第一章 设计模式概述

1、 设计模式具有（D）的优点。

a) 提高系统性能
b) 减少类的数量，降低系统的规模
c) 减少代码开发工作量
d) 提升软件设计的质量

2、 在面向对象软件开发过程中，采用设计模式（C）。

a) 可以减少在设计和实现过程中需要创建的实例对象的数量
b) 可以保证程序的运行速度达到最优值
c) 可以复用相似问题的相同解决方案
d) 允许在非面向对象程序设计语言中使用面向对象的概念

3、 （B）都是行为型设计模式。

a) 组合模式、适配器模式和代理模式
b) 观察者模式、职责链模式和策略模式
c) 原型模式、建造者模式和单例模式
d) 迭代器模式、命令模式和桥接模式

4、 什么是设计模式？它包含哪些基本要素？

设计模式是在特定环境下为解决某一通用软件设计问题提供的一套定制的解决方案，该方案描述了对象和类之间的相互作用。

设计模式一般包含 模式名称、问题、目的、解决方案、效果、实例代码和相关设计模式等要素。模式名称通过一两个词来描述模式的问题、解决方案和效果，以便更好的理解模式并方便开发人员之间的交流，绝大多数模式都是根据其功能或模式结构来命名的。问题描述了应该在何时使用模式，它包含了设计中存在的问题以及问题存在的原因。解决方案描述了设计模式的组成成分，以及这些组成成分之间的相互关系，各自的职责和协作方式。效果描述了模式应用的效果以及在使用模式时应该权衡的问题。

5、 设计模式如何分类？每一类设计模式有何特点？

设计模式根据目的（模式时用来做什么的）可分为创建型、结构型和行为型 3 类。

a) 创建型模式主要用于创建对象。GOF 提供了 5 种创建型模式，分别是 Factory Method、Abstract Factory、Builder、Prototype 和 Singleton。
b) 结构型模式主要用于处理类和对象的组合。GOF 提供了 7 种结构型模式，分别是 Adapter、Bridge、Composite、Decorator、Facade、Flyweight 和 Proxy。
c) 行为型模式主要用于描述类和对象怎样交互和怎样分配职责。GOF 提供了 11 种行为型模式，分别是 Chain of Responsibility、Command、Interpreter、Iterator、Mediator、Memento、Observer、State、Strategy、Template Method 和 Visitor。

设计模式根据范围（即模式主要是用于处理类之间的关系还是处理对象之间的关系）可分为类模式和对象模式两种。

a) 类模式处理了类和子类之间的关系，这些关系通过继承建立，在编译时就被确定下来，是一种及静态关系。
b) 对象模式处理对象之间的关系，这些关系在运行时变化，更具动态性。

6、 设计模式具有哪些优点？

设计模式是从许多优秀的软件系统中总结出来的成功的、能够实现可维护性复用的设计方案，使用这些方案将避免做一些重复性的工作，而且可以设计出高质量的软件系统，

具体来说具有以下优点：

1) 设计模式融合了众多专家的经验，并以一种标准的形式供广大开发人员使用，它提供了一套通用的设计词汇和一些通用的语言以方便开发人员之间沟通和交流，使得设计方案更加通俗易懂。对于使用不同编程语言的开发和设计人员可以通过设计模式来交流系统设计方案，每一个模式都对应一个标准的解决方案，设计模式可以降低开发人员理解系统的复杂度。
2) 设计模式使人们可以更加简单、方便地复用成功的设计和体系结构，将已证实的技术表述成设计模式也会使新系统开发者更加容易理解其设计思路。设计模式使得重用成功的设计更加容易，并避免那些导致不可重用的设计方案。
3) 设计模式使得设计方案更加灵活、且易于修改。在很多设计模式中广泛使用了开闭原则、依赖倒转原则、迪米特法则等面向对象设计原则，使得系统具有较好的可维护性，真正实现可维护性复用。在软件开发中合理的使用设计模式可以使系统中的一些组成部分在其他系统中得以重用，而且在此基础上进行二次开发很方便。
4) 设计模式的使用将提高软件系统的开发效率和软件质量，并且在一定程度上节约设计成本。设计模式是一些通过多次实践得以证明的行之有效的解决方案，这些解决方案通常是针对某一类问题的最佳设计方案，因此可以帮助设计人员构造优秀的软件系统，并且直接重用这些设计经验，节省系统设计成本。
5) 设计模式有助于初学者更深入地理解面向对象思想，一方面可以帮助初学者更加方便地阅读和学习现有类库与其他系统中的源代码，另一方面还可以提高软件设计水平和代码质量。

7、除了设计模式之外，目前有不少人在从事“反模式”的研究，请查阅资料，了解“反模式”以及研究反模式的意义。

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、请查阅相关资料，了解在 JDK 中使用了哪些设计模式，在何处使用了何种设计模式，至少距离两个。

JDK 中部分设计模式使用示例列举如下：

1、 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()

2、 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.*

3、 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

第二章 面向对象设计原则

1、开闭原则是面向对象的可复用设计的基石，开闭原则是指一个软件实体应当对（**扩展**）开放，对（**修改**）关闭；里氏代换原则是指任何（**基类对象**）可以出现的地方，（**子类对象**）一定可以出现；依赖倒转原则就是要依赖于（**抽象**），而不要依赖于（**实现**），或者说要针对接口编程，不要针对实现编程。

2、关于单一职责原则，以下叙述错误的是（C）

a) 一个类只负责一个功能领域中的相应职责。
b) 就一个类而言，应该有且仅有一个引起它变化的原因。
c) 一个类承担的职责越多，越容易复用，被复用的可能性就越大。
d) 当一个类承担的职责过多时需要将职责进行分离，将不同的职责封装在不同的类中。

3、以下关于面向对象设计的叙述中错误的是（D）

a) 高层模块不应该依赖于底层模块
b) 抽象不应该依赖于细节
c) 细节可以依赖于抽象
d) 高层模块无法不依赖于底层模块

4、在系统设计中应用迪米特法则，以下叙述有误的是（D）

a) 在类的划分上应该尽量创建松耦合的类，类的耦合度越低，复用越容易
b) 如果两个类之间不必彼此直接通信，那么这两个类就不用当直接的相互作用
c) 在对其他类的引用上，一个对象对其他对象的应当应该当降低到最低
d) 在类的设计上，只要有可能，一个类型应该尽量设计成抽象类或接口，且成员变量和成员函数的访问权限最好设置为公开的（public）。

5、 有人将面向对象设计原则简单地归为 3 条：①封装变化点②对接口进行编程③多使用组合而不是继承。请查阅相关资料并结合本章所学内容谈谈对这三条原则的理解。

“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”。

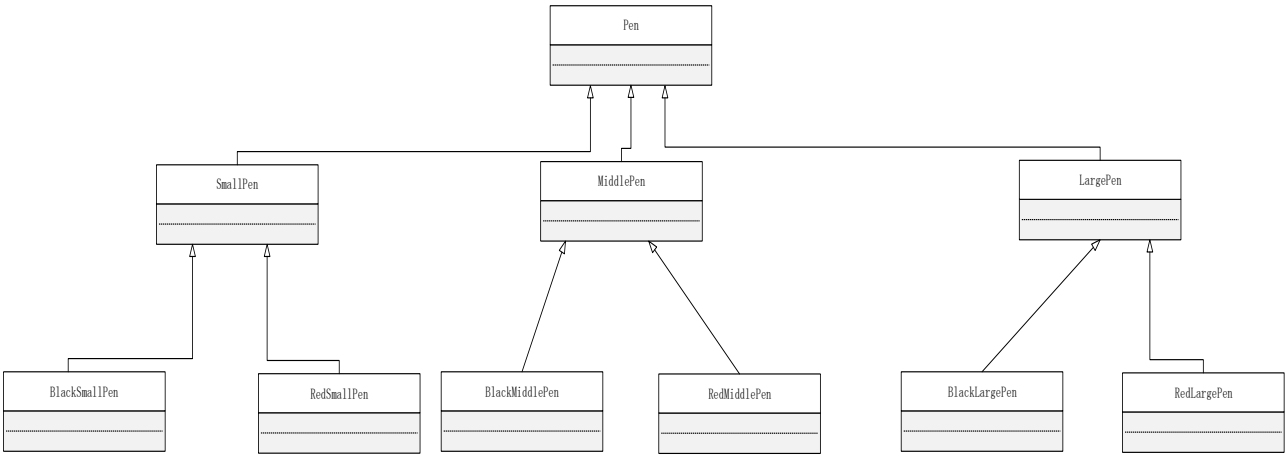
6、 结合本章所学习的面向对象设计原则谈谈对类和接口“粒度”理解。

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、 结合面向对象设计原则分析正方形是否为长方形的子类？

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：

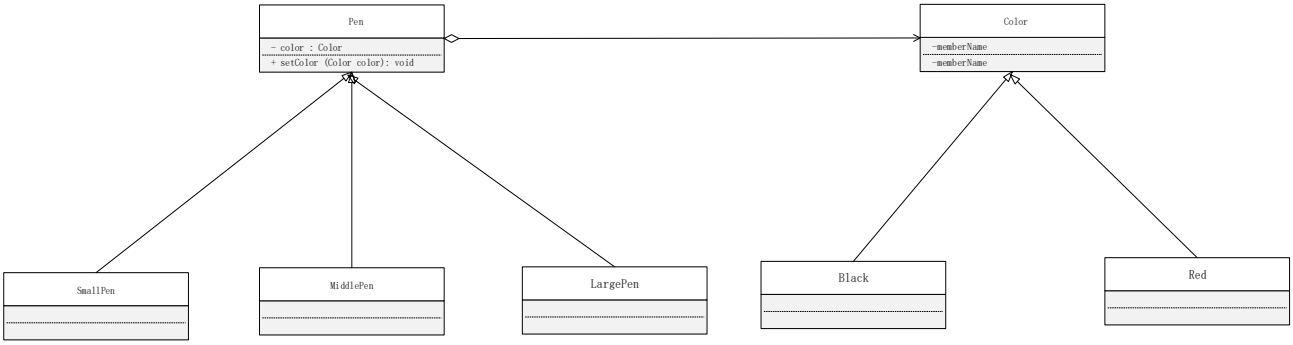
8、 在某绘图软件中提供了多种大小不同的画笔，并且可以给画笔指定不同的颜色，某设计人员针对画笔的结构设计了如图所示的初始类图。



通过仔细分析，设计人员发现该类图存在非常严重的的问题，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。例如增加一种绿色，则对应每一种大小的笔都需要增加一支绿色的笔，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的都较为方便。

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）

第三章 简单工厂模式

1、 在简单工厂模式中，如果需要增加新的具体产品，通常需要修改（C）的源代码。

a) 抽象产品类
b) 其他具体产品类
c) 工厂类
d) 客户类

2、 以下关于简单工厂模式的叙述错误的是（C）。

a) 简单工厂模式可以根据参数的不同返回不同产品类的实例。
b) 简单工厂模式专门定义一个类来负责创建其他类的实例，被创建的实例通常都具有共同的父类。
c) 简单工厂模式可以减少系统中类的个数，简化系统的设计，使得系统更易于理解
d) 系统的扩展困难，在添加新的产品类时需要修改工厂的业务逻辑，违背了开闭原则

3、 以下代码使用了（A）模式。

```
public abstract class Product{
    public abstract void process();
}
public class ConcreteProductA extends Product{
    public void process(){…}
}
public class ConcreteProductB extends Product{
    public void process(){…}
}
public class Factory{
    public static Product creatProduct(char type){
       switch(type){
         case  'A' :
            return new ConcreteProductA();break;
         case  'B' :
            return new ConcreteProductB();break;
         …
        }
    }
}
```

A、Simple Factory B、Factory Method C、Abstract FactoryD、未用任何设计模式

第四章 工厂方法模式

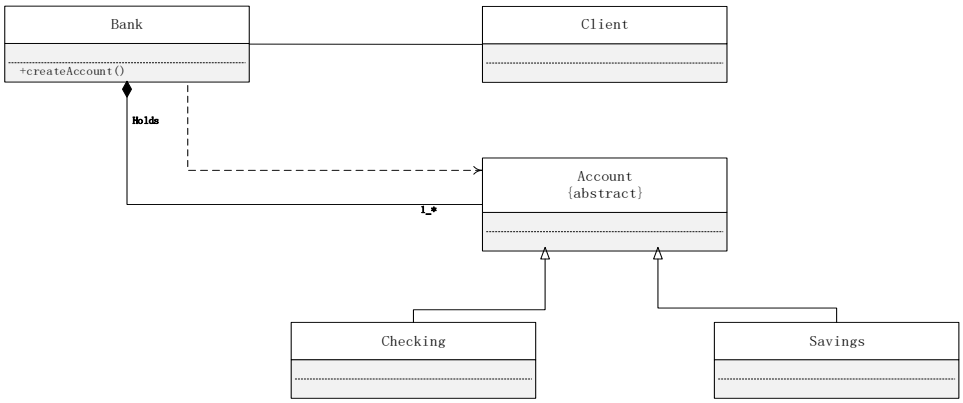
1、 不同品牌的手机应该有不同的公司制造，三星公司生产三星手机，苹果公司生产苹果手机。该场景蕴含了（B）设计模式。

(a)Simple Factory (b）Factory Method
(c)Abstract Factory (d)Builder

2、 在以下关于工厂方法模式的叙述错误的是（D）。

a） 在工厂方法模式中引入抽象工厂类，而具体产品的创建延迟到具体工厂中实现
b） 工厂方法模式添加新的产品对象很容易，无需对原有的系统进行修改，符合开闭原则
c） 工厂方法模式存在的问题是在添加新产品时需要编写新的具体产品类，而且还要提供与之对应的具体工厂类，随着类个数的增加会给系统带来一些额外的开销
d） 工厂方法模式是所有形式的工厂模式中最为抽象和最具一般性的一种形态，工厂方法模式退化后可以演变成抽象工厂模式

3、 某银行系统采用工厂模式描述其不同账户之间的关系，设计出的类图如图所示。其中与工厂模式中的 Creator 角色相对应的类是（Bank），与 Product 角色相对应的类是（Account）



题3
某银行系统类图

第五章 抽象工厂模式

1、某公司要开发一个图表显示系统，在该系统中曲线图生成器可以创建曲线图、曲线图图例和曲线图数据标签；柱状图生成器可以创建柱状图、柱状图图例和柱状图数据标签。用户要求可以很方便地增加新的类型的图形，系统具备较好的可扩展能力。针对这种需求，公司采用（D）最为恰当。

A 桥接模式　　B 适配器模式　　C 策略模式　　D 抽象工厂模式

2、以下关于抽象工厂模式的叙述错误的是（D）

A. 抽象工厂模式提供了一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。
B. 当系统中有多于一个产品族是可以考虑使用抽象工厂模式
C. 当一个工厂等级结构可以创建出属于不同产品等级结构的一个产品族中的所有对象时，抽象工厂模式比工厂方法模式更为简单、有效率
D. 抽象工厂模式符合开闭原则，增加新的产品族和新的产品等级结构都很方便

3、关于抽象工厂模式中的产品族和产品等级结构的叙述错误的是（A）

A. 产品等级结构是从不同的产品族中任意选取产品组成的层次结构
B. 产品族是指位于不同产品等级结构、功能相关的产品组成的家族
C. 抽象工厂模式是指一个工厂等级结构可以创建出分属于不同产品等级结构的一个产品族中的所有对象
D. 工厂方法模式对应唯一一个产品等级结构，而抽象工厂模式需要面对多个产品等级结构

第六章 建造者模式

1、 以下关于建造者模式的叙述错误的是（D）。

A. 建造者模式将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表述。

B. 建造者模式允许用户只通过指定复杂对象的类型和内容就可以创建它们，而不需要知道内部的而具体构建细节

C. 当需要创建的产品对象有着复杂的内部结构时可以考虑使用建造者模式

D. 在建造者模式中，各个具体的建造者之间通常具有较强的依赖关系，可通过指挥者类组装成一个完整的产品对象返回给用户。

2、 当需要创建的产品具有复杂的内部结构时，为了逐步构造完整的对象，并使得对象的创建更具有灵活性，可以使用（C）。

A 抽象工厂模式　　B 原型模式　　C 建造者模式　　D 单例模式

3、 关于建造者模式中的 Director 类的描述错误的是（D）

A. Director 类隔离了客户类及创建过程

B. 在建造者模式中客户类指导 Director 类去生成对象或者合成一些类，并逐步构造一个复杂对象

C. Director 类构建一个抽象建造者 Builder 子类的对象。

D. Director 与抽象工厂模式中的工厂类很相似，负责返回一个产品族中的所有产品

第七章 原型模式

1、 关于 java 语言中的 clone()方法，以下叙述错误的是（A）

A. 对于对象 x,都有 x.clone()==x

B. 对于对象 x,都有 x.clone().getClass()==x,getClass()

C. 对于对象 x 的成员对象 member,都有 x.clone().getMember()==x.getMember()

D. 对于对象 x 的成员对象 member，都有 x.clone().getMember().getClass()==x.getMember().getClass()

2、 以下关于原型模型的叙述错误的是（D）

A. 原型模式通过给出一个原型对象来指明所要创建对象的类型，然后用复制这个对象的方法创建出更多的对象

B. 浅克隆仅仅复制所考虑的对象，而不复制它所引用的对象，也就是其中的成员变量并不复制

C. 在原型模式中实现深克隆时常常需要编写较为复杂的代码

D. 在原型模式中，不需要为每一个类配备有一个克隆方法，因此对于原型模式的扩展很灵活，对于已有类的改造也较为方便

3、 某公司要开发一个即时聊天软件，用户在聊天过程中可以与多位好友共同聊天，在私聊时将产生多个聊天窗口，为了提高聊天窗口的创建效率，要求根据第一个窗口快速创建其他窗口。针对这种需求，采用（C）进行设计最为合适。

A 享元模式　　B 单例模式　　C 原型模式　　D 组合模式

第八章 单例模式

1、 在（B）时可使用单例模式。

A. 隔离菜单项对象的创建和使用

B. 防止一个资源管理器窗口被实例化多次

C. 使用一个已有的查找算法而不想修改既有代码

D. 不能创建子类，需要扩展一个数据过滤类

2、 以下关于单例模式的描述正确的是（B）。

A. 它描述了只有一个方法的类的集合。

B. 它能够保证一个类只产生一个唯一的实例。

C. 它描述了只有一个属性的类的集合。

D. 它能够保证一个类的方法只能被一个唯一的类调用。

3、 以下（B）不是单例模式的要点。

A. 某个类只有一个实例

B. 单例类不能被继承

C. 必须自行创建单个实例

D. 必须自行向整个系统提供单个实例

4、 分析并理解饿汉式单例和懒汉式单例的异同。

1）延迟加载：饿汉式单例模式在类被加载时就将自己实例化；懒汉式单例模式在第一次使用时被创建，无需一直占用系统资源，实现了延迟加载。

2）线程安全：饿汉式单例模式线程安全吗，懒汉式单例模式线程不安全

3）响应时间：由于饿汉式单例模式的单例对象在类加载是对象就得以创建所以比懒汉式单例模式响应时间短。

5、 什么是双重检查锁定？为什么进行双重检查锁定？Java 如何实现双重检查锁定？

6、 分别采用双重检查锁定和 IoDh 实现 8.3 节的负载均衡器。

```
/**
*双重检查锁定
*
*/
public class Singleton {
    private volatile static Singleton instance = null;

    private Singleton(){}

    public static Singleton getInstance(){
        //第一重判断
        if (instance==null){
            //锁定代码块
            synchronized (Singleton.class){
                //第二重判断
                if (instance==null){
                    instance=new Singleton();//创建单例实例
                }
```

饿汉式单例类不能实现延迟加载，不管将来用不用始终占用内存；懒汉式单例类线程安全控制繁琐，而且性能受影响；通过使用 IoDH 既可以实现延迟加载，又可以保证线程安全，不影响系统性

`` } } return instance; } } ```	能，不失为最好的 java 语言单例模式的实现方式；其缺点你是与编程语言本身的特性相关，很多面向对象语言并不支持 IoDh。
`` /** * IODH：Initialization on Demand Holder */ public class Singleton{ private Singleton(){} //静态内部类 private static class HoldClass{ private final static Singleton instance = new Singleton(); } public static Singleton getInstance(){ return HoldClass.instance; } } ```	

第九章 适配器模式

1、 (B)将一个类的接口装换成客户希望的另一个接口，使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。

A) 命令模式
B) 适配器模式
C) 策略模式
D) 单例模式

2、 以下关于适配器模式叙述错误的是（C）

A) 适配器模式将一个接口转换成客户希望的另外一个接口，使得原本接口不兼容的那些类可以要一起工作。
B) 在类适配器中 Adapter 和 Adaptee 是继承关系，而在对象适配器中 Adapter 和 Adaptee 是关联关系。
C) 类适配器比对象适配器灵活，在 java 语言中可以通过类适配器一次适配多个适配者类。
D) 适配器可以在不修改原来的适配者接口 Adaptee 的情况下将一个类的接口和另一个类的接口匹配起来。

3、 现需要开发一个文件转换软件，将文件由一种格式转换为另一种格式。例如将 XML 文件转换为 PDF 文件，将 DOC 文件转换为 TXT 文件，有些文件格式转换代码已经存在，为了将已有的代码应用于新软件而不修改软件的整体结构，可以使用（A）设计模式进行系统设计。

A) 适配器模式

B) 组合模式
C) 外观模式
D) 桥接模式

4、 在对象适配器中，适配器类和适配者类之间的关系为（A）

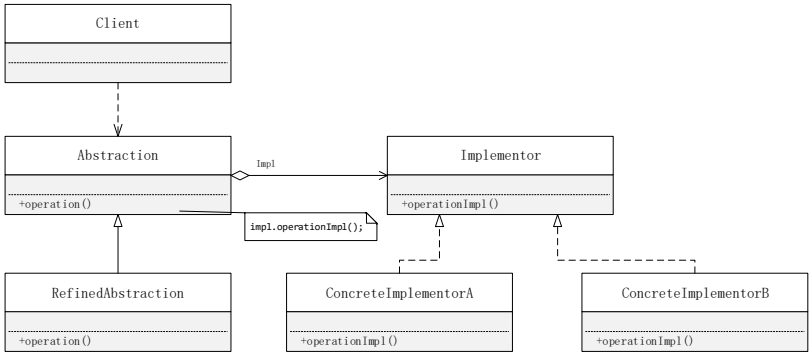
A) 关联关系
B) 依赖关系
C) 继承关系
D) 实现关系

5、 在对象适配器中一个适配器能否适配多个适配者？如果能，应该如何实现？如果不能，请说明原因。如果是类适配器呢？

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。

第十章 桥接模式

1、 （①B）设计模式将抽象部分与它的实现部分分离，使它们都可以独立变化。下图所示为改设计模式的类图，其中（②D）用于定义实现部分的接口。



① A、Singleton B、Bridge C、Composite D、Façade
② A、Abstrarction B、ConcreteImplementorA
C、ConcreteImplementorB D、Implementor

2、 以下关于桥接模式的叙述错误的是（C）

A) 桥接模式的用意是将抽象化与实现化脱耦，使得两者可以独立变化
B) 桥接模式将继承关系转换为关联关系，从而降低系统的耦合度
C) 桥接模式可以动态的给一个对象增加功能，这些功能也可以动态的撤销
D) 桥接模式可以从接口中分离实现功能，使得设计更具扩展性

3、 (C)不是桥接模式所使用的的场景。

A) 一个可以跨平台并支持多种格式的文件编辑器
B) 一个支持多数据源的报表生成工具，可以用不同的图形方式显示报表信息
C) 一个可动态选择排序算法的数据操作工具
D) 一个支持多种编程语言的跨平台开发工具

4、 如果系统中存在两个以上的变化维度，是否可以使用桥接模式进行处理？如果可以，系统该如何设计？

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。

第十一章 组合模式

1、 一个树形文件系统体现了(B)模式。

A. 装饰　　B.组合　　C.桥接　　D.代理

2、 以下关于组合模式的叙述错误的是(B)

A) 组合模式对叶子对象和组合对象的使用具有一致性

B) 组合模式可以很方便的保证在一个容器中只能有特定的构件

C) 组合模式将对象组织到树形结构中，可以用来描述整体与部分的关系

D) 组合模式使得可以很方便的在组合体中加入新的对象构件，客户端不需要因为加入新的对象构件而改变类库代码

3、 现需要开发一个 XML 文档处理软件，可以根据关键字查询指定内容，用户可以在 XML 中任意选取某一个节点作为作为查询的起始节点，无需关心该节点所处的层次结构。针对该需求可以使用（C）模式进行设计。

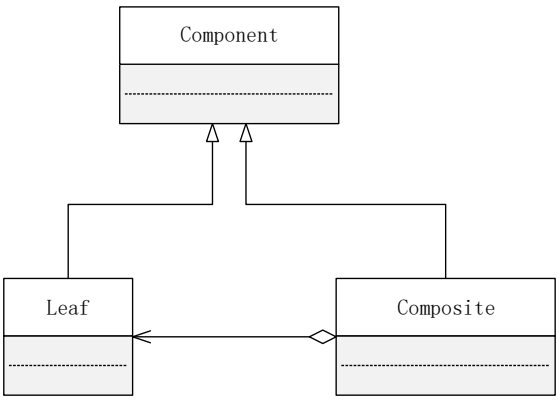
A) 抽象工厂模式

B) 享元模式

C) 组合模式

D) 策略模式

4、 在组合模式结构图中，如果聚合关联关系不是从 Composite 到 Component，而是从 Composite 到 Leaf，如图所示，会产生怎样的结果？



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。

第十二章 装饰模式

1、 当不能采用生成子类的的方法进行扩充时可采用（D）设计模式动态的给它一个对象添加一些额外的职责。

A.外观　　B.单例　　C.参与者　　D.装饰

2、 以下（C）不是装饰模式的适用条件。

A. 要扩展一个类的功能或给一个类增加附加责任。

B. 要动态的给一个对象增加功能，这些功能还可以动态撤销
C. 要动态的组合多于一个的抽象化角色和实现化角色
D. 要通过一些基本功能的组合产生复杂的功能，而不适用继承

3、 半透明装饰模式能否实现对同一个对象的多次装饰？为什么？

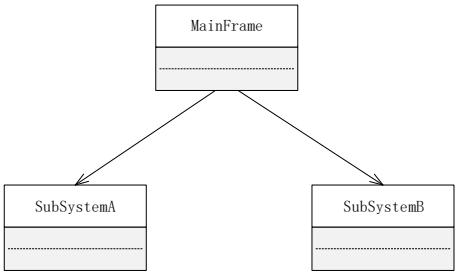
不能实现对用一个对象的多次装饰。因为在半透明装饰模式中，使用具体装饰类来声明装饰之后的对象，具体装饰类中新增的方法并未在抽象构件类中声明，这样做的优点在于装饰后客户端可以单独调用在具体装饰类中新增的业务方法，但是将导致无法调用到之前装饰时新增的方法，只能调用到最后一次装饰时具体装饰类中新增加的方法，故对同一个对象实施多次装饰没有任何意义。

第十三章 外观模式

1、 已知某子系统为外界提供功能服务，但该子系统存在很多粒度十分小的类，不便被外界系统该直接使用，采用（A）设计模式可以定义一个高层接口，这个接口使得这一子系统更加容易使用。

A.Façade(外观) B.Singleton(单例) C.Participant(参与者) D.Decorator(装饰)

2、 下图是（D）模式实例的结构图。



A.桥接(Bridge) B.工厂方法(Factory Method)
C.模板方法(Template) D.外观(Facade)

3、 以下关于外观模式的叙述错误的是（C）

A. 在外观模式中，一个子系统的外部与其内部的通信可以通过一个统一的外观对象进行
B. 在增加外观对象之后，客户类只需要直接和外观对象交互即可，与子系统类之间的复杂引用关系有外观类对象来实现，降低了系统的耦合度
C. 外观模式可以很好的限制客户类使用子系统，对客户类访问子系统做限制可以提高系统的灵活性
D. 可以为一个系统提供多个外观类

第十四章 享元模式

1、 当应用程序由于大量使用的对象造成很大的内存开销时，可以采用（C）设计模式运用共享技术来有效的支持大量细粒度对象的重用。

A.外观 B.组合 C.享元 D.适配器

2、 在享元模式中，外部状态是指（D）

A. 享元对象可共享的所有状态
B. 享元对象可共享的部分状态

C. 由享元对象自己保存和维护的状态
D. 由客户端保存和维护的状态

3、 以下关于享元模式的叙述错误的是（C）

A. 享元模式运用共享技术有效的支持大量细粒度对象的复用
B. 在享元模式中可以多次使用某个对象，通过引入外部状态使得这些对象可以有所差异
C. 享元对象能够做到共享的关键是引入了享元池，在享元池中通过克隆方法向客户端返回所需对象
D. 在享元模式中，外部状态是随环境改变而改变、不可以共享的状态，内部状态是不随环境改变而改变、可以共享的状态

第十五章 代理模式

1、 Windows 操作系统中应用快捷方式是（A）模式的应用实例。

A.代理　　B.组合　　C.装饰　　D.外观

2、 以下关于代理模式的是叙述错误的是（B）

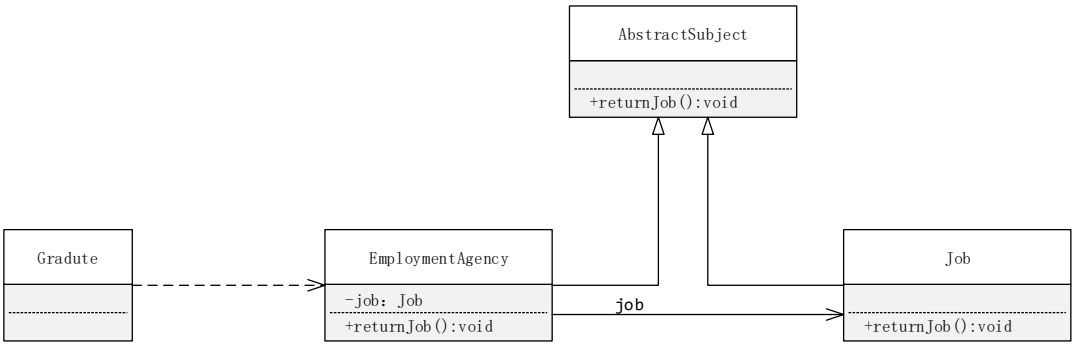
A. 代理模式能够协调调用者和被调用者，从而在一定程度上降低系统的耦合度
B. 控制对一个对象的访问，给不同的用户提供不同级别的使用权限时可以考虑使用远程代理模式
C. 代理模式的缺点是请求处理的速度会变慢，并且实现代理模式需要额外的工作
D. 代理模式给某一个对象提供一个代理，并由代理对象控制对原对象的引用

3、 代理模式有多种类型，其中智能引用代理是指（D）。

A. 为某一个目标操作的结果提供临时的存储空间，以便多个客户端可以共享这些结果
B. 保护目标不让恶意用户接近
C. 使几个用户能够同时使用一个对象而没有冲突
D. 当一个对象被引用时提供一些额外的操作，例如将此对象被调用的次数记录下来

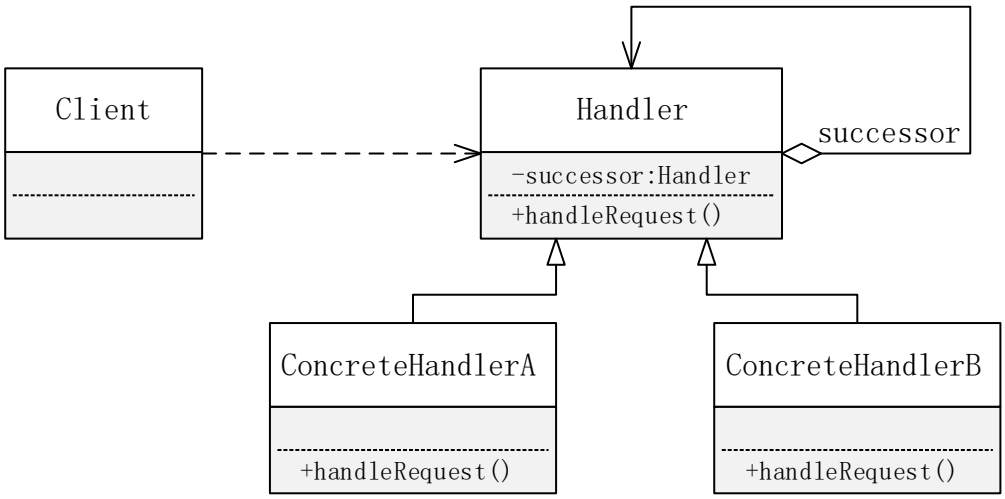
4、 **毕业生通过职业介绍所介绍找工作，请问该过程蕴含了那种设计模式，绘制了相应的类图。**

代理模式



第十六章 职责链模式

1、 下图描述了一种设计模式，该设计模式不可以（B）。



A. 动态决定由一组对象中的某个对象处理该请求。
B. 动态决定处理一个请求的对象集合，并高效地处理一个请求。
C. 使多个对象都有机会处理请求，避免请求的发送者和接收者间的耦合关系
D. 将对象连成一条链，并沿着该链传递请求。

2、 接力赛跑体现了（A）设计模式。

A. 职责链模式
B. 命令模式
C. 备忘录模式
D. 工厂方法

3、 Java 语言中的异常处理机制是职责链模式的一个应用实例，编写一个包含多个 catch 字句的程序，理解异常处理的实现过程，并判断此处使用的是纯职责链模式还是不纯职责链模式。

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。

第十七章 命令模式

1、 以下关于命令模式的叙述错误的是（D）。

A、 命令模式将一个请求封装为一个对象，从而可用不同的请求对客户进行参数化
B、 命令模式可以将请求发送者和请求接收者解耦
C、 使用命令模式会导致某些系统有过多的具体命令类，导致在有些系统中命令模式变得不切实际
D、 命令模式是对命令的封装，命令模式把发出命令的责任和执行命令的责任集中在一个同一个类中，委派给统一的类进行处理

2、 在(C)是无须使用命令模式。

A、 实现撤销操作和恢复操作
B、 将请求的发送者和接收者解耦
C、 不改变聚合类的前提下定义作用于聚合中元素的新操作
D、 不同的时间指定请求，并将请求排队

第十八章 解释器模式

1、 **对于一个语法不是特别复杂的计算机语言，可以考虑使用(D)模式进行设计。**
   A、 模板方法
   B、 命令
   C、 访问者
   D、 解释器
2、 **关于解释器模式，以下叙述有误的是（C）。**
   A、 当一个待解释的语言中的句子可以表示为一颗抽象语法树时可以使用解释器模式
   B、 在解释器模式中使用类来表示文法规则，可以很方便地改变或者扩展文法
   C、 解释器模式既适用于文法简单的小语言，也适用于文法非常复杂的语言解析
   D、 需要自定义一个小语言，如一些简单的控制指令时，可以考虑使用解释器模式。

第十九章 迭代器模式

1、 **迭代器模式用于处理具有(B)性质的类。**
   A、 抽象
   B、 聚集
   C、 单例
   D、 共享
2、 **以下关于迭代器模式的叙述错误的是（D）。**
   A、 迭代器模式是一种方法来访问聚合对象，而无需暴露这个对象的内部表示。
   B、 迭代器模式支持以不同的方式遍历一个聚合对象
   C、 迭代器模式定义了一个访问聚合元素的接口，并且可以跟踪当前遍历的元素，了解哪些元素已经遍历过而哪些没有
   D、 在抽象聚合类中定义了访问和遍历元素的方法，并在具体聚合类中实现这些方法
3、 **在迭代器模式中将数据存储和数据遍历分离，数据存储由聚合类负责，数据遍历由迭代器负责，这种设计方案是（C）的具体应用。**
   A、 依赖倒转原则
   B、 接口隔离原则
   C、 单一职责原则
   D、 合成复用原则

第二十章 中介者模式

1、 **在图形界面系统开发中，如果界面组件之间存在较为复杂的相互调用关系，为了降低界面组件之间的耦合度，让它们不产生直接的相互引用，可以使用（C）设计模式。**
   A、 组合
   B、 适配器
   C、 中介者
   D、 状态
2、 **在中介者模式中通过中介者将同事类解耦，这是(A)的具体应用。**

A、 迪米特法则
B、 接口隔离原则
C、 里氏代换原则
D、 合成服用原则

3、 以下关于中介者模式的叙述错误的是（B）。

A、 中介者模式用一个中介对象来封装一系列的对象交互
B、 中介者模式和观察者模式均可以用于降低系统的耦合度，中介者模式用于处理对象之间一对多的调用关系，而观察者模式用于处理多对多的调用关系
C、 中介者模式简化了对象之间的交互，将原本难以理解的网状结构转换成相对简单的星型结构。
D、 中介者将原本分布于多个对象间的行为集中在一起，改变这些行为只需要生成新的中介者子类即可，这使得各个同事类可被重用。

第二十一章 备忘录模式

1、 很多软件都提供了撤销功能，(B)设计模式可以用于实现该功能。

A、 中介者
B、 备忘录
C、 迭代器
D、 观察者

2、 以下关于备忘录模式叙述错的是（D）。

A、 备忘录模式的作用就是在不破坏封装的前提下捕获一个对象的内部状态，并在该对象之外保存这个状态，这样可以在以后将对象恢复到原先保存的状态。
B、 备忘录模式提供了一种状态恢复的实现机制，使得用户可以很方便地回到一个特定的历史步骤。
C、 备忘录模式的缺点是在于资源消耗太大，如果类的成员变量太多，就不可避免的地占用大量的内存，而且每保存一次对象的状态都需要消耗内存资源。
D、 备忘录模式属于对象行为型模式，负责人向原发器请求一个备忘录，保留一段时间后将其送回负责人，负责人负责对备忘录的内容进行操作和检查。

第二十二章 观察者模式

1、 （D）设计模式定义了对象之间一种一对多的依赖关系，以便当一个对象的状态发生改变时所有依赖于它的对象都得到通知并自动刷新。

A、 适配器
B、 迭代器
C、 原型
D、 观察者

2、 在观察者模式中，(A)。

A、 一个 subject 对象可对应于多个 Observer 对象
B、 Subject 只能有一个 ConcreteSubject 子类
C、 Observer 只能有一个 ConcreteObserver 子类
D、 一个 Subject 对象至少对应一个 Observer 对象

3、 下面这句话隐含着(C)设计模式。“我和妹妹跟妈妈说：“妈妈，我和妹妹在院子里玩。饭做好了叫我一下””

A、 适配器

B、 职责链

C、 观察者

D、 迭代器

第二十三章 状态模式

1、 以下关于状态模式的叙述错误的是（D）

A、 状态模式允许一个对象在其内部状态改变时改变它的行为。对象看起来似乎修改了它的类。

B、 状态模式中引入了一个抽象类来专门表示对象的状态,而具体的状态都继承了该类,并实现了不同状态的行为，包括各种状态之间的转换。

C、 状态模式使得状态的改变更加清晰明了，也容易创建对象的新状态。

D、 状态模式完全符合开闭原则，增加新的状态类无需对原有类库进行任何修改。

2、 场景(C)不是状态模式的实例。

A、 银行账户根据余额不同用于不同的存取款操作

B、 游戏软件中根据虚拟角色的不同拥有不同的权限

C、 某软件在不同的操作系统中程序不同的外观

D、 在会员系统中会员等级操作可以实现不同的行为

3、 分析一下代码：

```
public class TestXYZ{
    private int behaviour;
    //Getter and Setter
    public void handleAll(){
        if (behaviour == 0) {//do something
        }else if (behaviour ==1) {//do something
        }else if (behaviour ==2) {//do something
        }else if (behaviour ==3) {//do something
        }...some more esle if ...
    }
}
```

为了提高代码的可扩展性和健壮性，可以使用(D)设计模式进行重构。

A、 访问者

B、 外观

C、 备忘录

D、 状态

第二十四章 策略模式

1、 在某系统中用户可自行动态地选择某种排序算法来实现某功能，该系统的设计可以使用（B）设计模式。

A、 状态
B、 策略
C、 模板方法
D、 工厂方法

2、 以下关于策略模式的叙述错误的是（B）。

A、 策略模式是对算法的包装，它把算法的责任和算法本身分隔开，委派给不同的对象管理
B、 在 Context 类中维护了所有 ConcreteStrategy 的引用实例
C、 策略模式让算法独立于使用它的客户而变化
D、 在策略模式中中定义了一些列算法，并将每一个算法封装起来，让它们可以相互替换

3、 以下关于策略模式的优缺点的描述错误的是（A）

A、 在策略模式中客户端无需知道所有的策略类，系统必须自行提供一个策略类
B、 策略模式可以避免使用多重条件转移语句
C、 策略模式会导致产生大量的策略类
D、 策略模式提供了管理相关算法族的方法

4、 在策略模式中一个环境类能否对应多个不同的策略等级结构？如何设计？

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。

第二十五章 模板方法模式

1、 某系统中的某子模块需要为其他模块提供访问不同数据库系统的功能，这些数据库系统提供的访问接口有一定的差异，但访问过程都是相同的。例如先连接数据库，再打开数据库，最后对数据进行查询，可使用(C)设计模式抽象出相同的数据库访问过程。

A、 观察者
B、 访问者
C、 模板方法
D、 策略

2、 以下关于模板方法模式的叙述错误的是（B）

A、 模板方法模式定义了一个操作中算法的骨架，而将一些步骤延迟到子类中
B、 模板方法模式是一种对象行为型模式
C、 模板方法模式使得子类可以不改变一个算法的结构即可重新定义该算法的某些特定步骤
D、 模板方法模式不仅可以调用原始操作，还可以调用定义于 AbstractClass 中的方法或其他对象中的方法

3、 在模板方法模式中，钩子方法如何实现子类控制父类的行为？

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。

第二十六章 访问者模式

1、 关于访问者模式中的对象结构，以下描述错误的是（A）

A、 它实现了 accept()方法，该操作以一个具体访问者作为参数

B、 可以提供一个高层次的接口以允许访问者访问它的元素

C、 可以是一个组合模式或一个集合

D、 能够枚举其中包含的元素

2、 以下关于访问者模式的叙述错误的是(D)

A、 访问者模式表示一个作用域某对象结构中的各元素的操作

B、 访问者模式使用用户可以在不改变各元素的类的前提下定义作用于这些元素的新操作

C、 再访问者模式中 ObjectStructure 提供一个高层接口以允许访问者访问它的元素

D、 在访问者模式中增加新的元素很容易

3、 什么是双重分派机制？如何用代码实现。