## 第一章 设计模式概述

1. **设计模式具有（D）的优点。**
   1. 提高系统性能
   2. 减少类的数量，降低系统的规模
   3. 减少代码开发工作量
   4. 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）。**
   1. 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
   2. 可以保证程序的运行速度达到最优值
   3. 可以复用相似问题的相同解决方案
   4. 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。**
   1. 组合模式、适配器模式和代理模式
   2. 观察者模式、职责链模式和策略模式
   3. 原型模式、建造者模式和单例模式
   4. 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。

1. **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。

1. 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。
2. 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。
3. 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。

1. 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。
2. 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。
3. **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。
6. **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。

1. **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()

* 1. 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

* 1. 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()

* 1. 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)

* 1. 单例模式 (Singleton)

• java.lang.Runtime#getRuntime()

• java.awt.Desktop#getDesktop()

1. 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()

* 1. 桥接模式(Bridge)

• AWT (提供了抽象层映射于实际的操作系统)  
• JDBC

* 1. 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)

* 1. 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]()

* 1. 外观模式(Facade)

• java.lang.Class  
• javax.faces.webapp.FacesServlet

* 1. 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)

* 1. 代理模式(Proxy)

• java.lang.reflect.Proxy  
• java.rmi.\*

1. 行为型模式：
   1. 职责链模式(Chain of Responsibility)

• java.util.logging.Logger#log()  
• javax.servlet.Filter#doFilter()

* 1. 命令模式(Command)

• java.lang.Runnable  
• javax.swing.Action

* 1. 解释器模式(Interpreter)

• java.util.Pattern  
• java.text.Normalizer  
• java.text.Format  
• javax.el.ELResolver

* 1. 迭代器模式(Iterator)

• java.util.Iterator  
• java.util.Enumeration

* 1. 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()

* 1. 备忘录模式(Memento)

• java.util.Date  
• java.io.Serializable  
• javax.faces.component.StateHolder

* 1. 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

* 1. 状态模式(State)

• java.util.Iterator  
• javax.faces.lifecycle.LifeCycle#execute()

* 1. 策略模式(Strategy)

• java.util.Comparator#compare()  
• javax.servlet.http.HttpServlet  
• javax.servlet.Filter#doFilter()

* 1. 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()

* 1. 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）**
   1. 一个类只负责一个功能领域中的相应职责。
   2. 就一个类而言，应该有且仅有一个引起它变化的原因。
   3. 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。
   4. 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）**
   1. 高层模块不应该依赖于底层模块
   2. 抽象不应该依赖于细节
   3. 细节可以依赖于抽象
   4. 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）**
   1. 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
   2. 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
   3. 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
   4. 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”。

1. **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。

1. **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：

1. **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。**
   1. 抽象产品类
   2. 其他具体产品类
   3. 工厂类
   4. 客户类
2. **以下关于简单工厂模式的叙述错误的是（C）。**
   1. 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。
   2. 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。
   3. 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
   4. 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

1. **在以下关于工厂方法模式的叙述错误的是（D）。**
   1. 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
   2. 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
   3. 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
   4. 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
2. **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）**



## 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

1. **以下关于抽象工厂模式的叙述错误的是（D）**
   1. 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。
   2. 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
   3. 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
   4. 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
2. **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）**
   1. 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
   2. 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
   3. 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
   4. 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）。**
   1. 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。
   2. 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
   3. 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
   4. 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

1. **关于建造者模式中的Director类的描述错误的是（D）**
   1. Director类隔离了客户类及创建过程
   2. 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
   3. Director类构建一个抽象建造者Builder子类的对象。
   4. 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）**
   1. 对于对象x,都有x.clone()==x
   2. 对于对象x,都有x.clone().getClass()==x,getClass()
   3. 对于对象x的成员对象member,都有x.clone().getMember()==x.getMember()
   4. 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）**
   1. 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
   2. 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
   3. 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
   4. 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。**
   1. 隔离菜单项对象的创建和使用
   2. 防止一个资源管理器窗口被实例化多次
   3. 使用一个已有的查找算法而不想修改既有代码
   4. 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）。**
   1. 它描述了只有一个方法的类的集合。
   2. 它能够保证一个类只产生一个唯一的实例。
   3. 它描述了只有一个属性的类的集合。
   4. 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。**
   1. 某个类只有一个实例
   2. 单例类不能被继承
   3. 必须自行创建单个实例
   4. 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。

1. **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？**
2. **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* }  }  }  **return** *instance*;  } } | **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*;  } }** |