## 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