面试题 1. 简述什么是设计模式？

推荐指数： ★★★ 试题难度： 初级 试题类型： 八股文 ▸

试题回答参考思路：

1：设计模式的概念：
设计模式（Design pattern）代表了最佳的实践，通常被有经验的面向对象的软件开发人员所采用。设计模式是软件开发人员在软件开发过程中面临的一般问题的解决方案。这些解决方案是众多软件开发人员经过相当长的一段时间的试验和错误总结出来的

2：个人理解：
设计模式并非是一种技术，而是在项目迭代的过程中，为了实现一些功能，设计了一些解决方案，将这些经验进行总结出来的一个模式体系，这个体系是在被反复使用的、多数人知晓的、经过分类编目的、代码设计经验的总结。

3：什么要使用设计模式呢?
用设计模式是为了复用代码、让代码更容易被他人理解、保证代码可靠性。 在目前的软件设计潮流初中毫无疑问，设计模式于己经应用与每个公司的各个项目中，设计模式使代码编写过程真正实现工程化，设计模式是软件工程的基石，如同大厦的一块块砖石一样。项目中合理地运用设计模式可以完美地解决很多问题，每种模式在现实中都有相应的原理来与之对应，每种模式都描述了一个在我们在开发过程中、各个业务中相似的且又不断重复发生的问题，以及提供了该问题的核心解决方案，这也是设计模式能被广泛应用的原因。

面试题 2. 叙述常见Java设计模式分类？

推荐指数： ★★★ 试题难度： 初级 试题类型： 八股文 ▸

试题回答参考思路：

常见Java设计模式分类
1：创建型模式，共五种：工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式。
2：结构型模式，共七种：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式。
3：行为型模式，共十一种：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。

面试题 3. Java 设计模式的六大原则？

推荐指数： ★★ 试题难度： 中级 试题类型： 八股文 ▸

试题回答参考思路：

1：开放封闭原则（Open Close Principle）
原则思想：尽量通过扩展软件实体来解决需求变化，而不是通过修改已有的代码来完成变化
描述：一个软件产品在生命周期内，都会发生变化，既然变化是一个既定的事实，我们就应

该在设计的时候尽量适应这些变化，以提高项目的稳定性和灵活性。
优点：单一原则告诉我们，每个类都有自己负责的职责，里氏替换原则不能破坏继承关系的体系。

2：里氏代换原则（Liskov Substitution Principle）
原则思想：使用的基类可以在任何地方使用继承的子类，完美的替换基类。
大概意思是：子类可以扩展父类的功能，但不能改变父类原有的功能。子类可以实现父类的抽象方法，但不能覆盖父类的非抽象方法，子类中可以增加自己特有的方法。
优点：增加程序的健壮性，即使增加了子类，原有的子类还可以继续运行，互不影响。

3：依赖倒转原则（Dependence Inversion Principle）依赖倒置原则的核心思想是面向接口编程.
依赖倒转原则要求我们在程序代码中传递参数时或在关联关系中，尽量引用层次高的抽象层类，
这个是开放封闭原则的基础，具体内容是：对接口编程，依赖于抽象而不依赖于具体。

4：接口隔离原则（Interface Segregation Principle）
这个原则的意思是：使用多个隔离的接口，比使用单个接口要好。还是一个降低类之间的耦合度的意思，从这儿我们看出，其实设计模式就是一个软件的设计思想，从大型软件架构出发，为了升级和维护方便。所以上文中多次出现：降低依赖，降低耦合。
例如：支付类的接口和订单类的接口，需要把这俩个类别的接口变成俩个隔离的接口

5：迪米特法则（最少知道原则）（Demeter Principle）
原则思想：一个对象应当对其他对象有尽可能少地了解，简称类间解耦
大概意思就是一个类尽量减少自己对其他对象的依赖，原则是低耦合，高内聚，只有使各个模块之间的耦合尽量的低，才能提高代码的复用率。
优点：低耦合，高内聚。

6：单一职责原则（Principle of single responsibility）
原则思想：一个方法只负责一件事情。
描述：单一职责原则很简单，一个方法 一个类只负责一个职责，各个职责的程序改动，不影响其它程序。 这是常识，几乎所有程序员都会遵循这个原则。
优点：降低类和类的耦合，提高可读性，增加可维护性和可拓展性，降低可变性的风险。

面试题 4. 简述对 MVC 的理解， MVC 有什么优缺点?

推荐指数： ★★ 试题难度： 初级 试题类型： 八股文 ▸

试题回答参考思路：

MVC 设计模式（应用观察者模式的框架模式）

M: Model(Business process layer)，模型，操作数据的业务处理层，并独立于表现层(Independe ntof presentation)。
V: View(Presentation layer)，视图，通过客户端数据类型显示数据，并回显模型层的执行结果 。
C: Controller(Control layer)，控制器，也就是视图层和模型层桥梁，控制数据的流向，

接受视图层发出的事件，并重绘视图

MVC 框架的一种实现模型：
模型二 (Servlet-centric) ：
JSP+Servlet+JavaBean ，以控制为核心， JSP 只负责显示和收集数据， Sevlet ，连接视图和模
型，将视图层数据，发送给模型层， JavaBean ，分为业务类和数据实体，业务类处理业务 数
据，数据实体，承载数据，基本上大多数的项目都是使用这种 MVC 的实现模式。
StrutsMVC 框架 (Web application frameworks)
Struts 是使用 MVC 的实现模式二来实现的，也就是以控制器为核心。
Struts 提供了一些组件使用 MVC 开发应用程序：
Model ： Struts 没有提供 model 类。这个商业逻辑必须由 Web 应用程序的开发者以 JavaBean或 EJB 的形式提供
View ： Struts 提供了 action form 创建 form bean, 用于在 controlle r 和 view 间传输数据。此 外 ，Struts 提供了自定义 JSP 标签库，辅助开发者用 JSP 创建交互式的以表单为基础的应用程 序 ，应用程序资源文件保留了一些文本常量和错误消息，可转变为其它语言，可用于 JSP 中。
Controller ： Struts 提供了一个核心的控制器 ActionServlet ，通过这个核心的控制器来调用其他用户注册了的自定义的控制器 Action ，自定义 Action 需要符合 Struts 的自定义 Action 规范，还需要在 struts-config.xml 的特定配置文件中进行配置，接收 JSP 输入字段形成 Action
form ，然后调用一个 Action 控制器。 Action 控制器中提供了 model 的逻辑接口。

面试题 5. 简述什么是典型的软件三层结构？软件设计为什么要分层?

推荐指数： ★★★ 试题难度： 初级 试题类型： 八股文 ▸

试题回答参考思路：

软件的三层结构一般指的是 MVC
M 是model的简称是指实体层
V是view的简称是指视图层
C是Controller的简称是指控制层

MVC执行HTTP流程：
(1) Presentation layer （表示层）
(2) 表示逻辑（生成界面代码）
(3) 接收请求
(3) 处理业务层抛出的异常
(4) 负责规则验证（数据格式，数据非空等）
(5) 流程控制第 3 页 共 10 页
(2) Service layer （服务层 / 业务层）
(1) 封装业务逻辑处理，并且对外暴露接口
(2) 负责事务，安全等服务
(3) Persistence layer （持久层）
(1) 封装数据访问的逻辑，暴露接口

(2) 提供方便的数据访问的方案（查询语言， API ，映射机制等）

Domain layer （域层）
(1) 业务对象以及业务关系的表示
(2) 处理简单的业务逻辑
(3) 域层的对象可以穿越表示层，业务层，持久层

面试题 6. 简述什么是单例模式，以及他解决的问题，应用的环境？

推荐指数： ★★★ 试题难度： 初级 试题类型： 八股文 ▸

试题回答参考思路：

一个类只允许创建一个实例对象，并提供访问其唯一的对象的方式。这个类就是一个单例类，这种设计模式叫作单例模式。
作用：避免频繁创建和销毁系统全局使用的对象。

单例模式的特点：
1：单例类只能有一个实例
2：单例类必须自己创建自己的唯一实例
3：单例类必须给所有其他对象提供这一实例的访问

应用场景：
全局唯一类，如 系统配置类、系统硬件资源访问类
序列号生成器
Web 计数器

面试题 7. 简述什么是工厂模式，以及他解决的问题，应用的环境？

推荐指数： ★★★★★ 试题难度： 中级 试题类型： 八股文 ▸

试题回答参考思路：

工厂模式是用来创建对象的一种最常用的设计模式，不暴露创建对象的具体逻辑，而是将将逻辑封装在一个函数中，那么这个函数就可以被视为一个工厂
其就像工厂一样重复的产生类似的产品，工厂模式只需要我们传入正确的参数，就能生产类似的产品

举个例子：

编程中，在一个 A 类中通过 new 的方式实例化了类 B，那么 A 类和 B 类之间就存在关联（耦合）
后期因为需要修改了 B 类的代码和使用方式，比如构造函数中传入参数，那么 A 类也要跟着修改，一个类的依赖可能影响不大，但若有多个类依赖了 B 类，那么这个工作量将会相当的大，容易出现修改错误，也会产生很多的重复代码，这无疑是件非常痛苦的事；
这种情况下，就需要将创建实例的工作从调用方（A类）中分离，与调用方解耦，也就是使

用工厂方法创建实例的工作封装起来（减少代码重复），由工厂管理对象的创建逻辑，调用方不需要知道具体的创建过程，只管使用，而降低调用者因为创建逻辑导致的错误

面试题 8. 简述什么是值对象模式，以及他解决的问题，应用的环境?

推荐指数： ★★★★ 试题难度： 中级 试题类型： 八股文 ▸

试题回答参考思路：

用来把一组数据封装成一个对象的模式。
1：解决问题：在远程方法的调用次数增加的时候，相关的应用程序性能将会有很大的下降。
2：解决方案：使用值对象的时候，可以通过仅仅一次方法调用来取得整个对象，而不是使用多次方法调用以得到对象中每个域的数值。
本质：就是把需要传递的多个值封装成一个对象一次性传过去

面试题 9. 请代码示例：值对象模式的实现方法？

推荐指数： ★★★ 试题难度： 高难 试题类型： 八股文 ▸

试题回答参考思路：

```
public class UserModel {
private String userId;
private String userName;
public void setUserId(String id) {
this.userId = id;
}
public String getUserId() {
return userId;
}
public void setUserName(String name) {
this.userName = name;
}
public String getUserName() {
return userName;
}
}
```

面试题 10. 请问什么是DAO模式？

推荐指数： ★★★ 试题难度： 初级 试题类型： 八股文 ▸

试题回答参考思路：

DAO：数据访问对象
解决问题：根据数据源不同，数据访问也不同。根据存储的类型（关系数据库、面向对象数据库、纯文件等）和供应商实现不同，持久性存储（如数据库）的访问差别也很大。如何对

存储层以外的模块屏蔽这些复杂性，以提供统一的调用存储实现。程序的分布式问题。
解决方案：是将数据访问逻辑抽象为特殊的资源，也就是说将系统资源的接口从其底层访问机制中隔离出来；通过将数据访问的调用打包，数据访问对象可以促进对于不同数据库类型和模式的数据访问。

DAO的本质：就是一层屏蔽一种变化。
本质：分层，是系统组件和数据源中间的适配器。（一层屏蔽一种变化）

面试题 11. 简述Spring开发中的哪里使用了工厂设计模式？

推荐指数： ★★★★★ 试题难度： 中级 试题类型： 八股文 ▸

试题回答参考思路：

Spring IOC
在Spring IOC容器创建bean的过程是使用了工厂设计模式
Spring中无论是通过xml配置还是通过配置类还是注解进行创建bean，大部分都是通过简单工厂来进行创建的。
当容器拿到了beanName和class类型后，动态的通过反射创建具体的某个对象，最后将创建的对象放到Map中。

面试题 12. 简述什么是代理模式？

推荐指数： ★★★ 试题难度： 初级 试题类型： 八股文 ▸

试题回答参考思路：

代理模式是常见的设计模式之一，顾名思义，代理模式就是代理对象具备真实对象的功能，并代替真实对象完成相应操作，并能够在操作执行的前后，对操作进行增强处理。（为真实对象提供代理，然后供其他对象通过代理访问真实对象）

举个例子来说明：假如说我现在想买一辆二手车，虽然我可以自己去找车源，做质量检测等一系列的车辆过户流程，但是这确实太浪费我得时间和精力了。我只是想买一辆车而已为什么我还要额外做这么多事呢？于是我就通过中介公司来买车，他们来给我找车源，帮我办理车辆过户流程，我只是负责选择自己喜欢的车，然后付钱就可以了

面试题 13. 请列举代理模式Java应用场景？

推荐指数： ★★★★ 试题难度： 高难 试题类型： 八股文 ▸

试题回答参考思路：

Spring AOP、日志打印、异常处理、事务控制、权限控制等

面试题 14. 简述什么是原型模式？

推荐指数： ★★★ 试题难度： 中级 试题类型： 八股文 ▸

试题回答参考思路：

1.什么是原型模式
原型设计模式简单来说就是克隆

原型表明了有一个样板实例，这个原型是可定制的。原型模式多用于创建复杂的或者构造耗时的实例，因为这种情况下，复制一个已经存在的实例可使程序运行更高效。

2.原型模式的应用场景
类初始化需要消化非常多的资源，这个资源包括数据、硬件资源等。这时我们就可以通过原型拷贝避免这些消耗。
通过new产生的一个对象需要非常繁琐的数据准备或者权限，这时可以使用原型模式。
一个对象需要提供给其他对象访问，而且各个调用者可能都需要修改其值时，可以考虑使用原型模式拷贝多个对象供调用者使用，即保护性拷贝。
我们Spring框架中的多例就是使用原型

面试题 15. 请简述Java中原型模式的使用方式？

推荐指数： ★★★★ 试题难度： 中级 试题类型： 八股文 ▸

试题回答参考思路：

实现Cloneable接口。在java语言有一个Cloneable接口，它的作用只有一个，就是在运行时通知虚拟机可以安全地在实现了此接口的类上使用clone方法。在java虚拟机中，只有实现了这个接口的类才可以被拷贝，否则在运行时会抛出CloneNotSupportedException异常。

重写Object类中的clone方法。Java中，所有类的父类都是Object类，Object类中有一个clone方法，作用是返回对象的一个拷贝，但是其作用域protected类型的，一般的类无法调用，因此Prototype类需要将clone方法的作用域修改为public类型

面试题 16. 简述什么是观察者模式?

推荐指数： ★★ 试题难度： 中级 试题类型： 八股文 ▸

试题回答参考思路：

观察者模式又叫发布者订阅者模式，它定义了一对多的关系，让多个观察者对象同时监听某一个主体对象，这个主体对象发生变化时就会通知所有的观察者对象，使得他们能够自己更新自己
如：你订阅游戏主播，当主播开播的时候他就会给你推送开播消息；
使用观察者模式的好处：
1.1支持简单的广播通信，自动通知所有已订阅的对象。
1.2页面载入后目标元素容易和观察者者存在一种动态关联，增加了灵活性。
1.3目标对象与观察者之间的抽象耦合关系能够单独扩展及运用

面试题 17. 请列举观察者模式应用场景？

推荐指数： ★★ 试题难度： 中级 试题类型： 八股文 ▸

试题回答参考思路：

1.当一个对象状态的改变需要改变其他对象， 或实际对象是事先未知的或动态变化的时，可使用观察者模式。
2.当应用中的一些对象必须观察其他对象时， 可使用该模式。 但仅能在有限时间内或特定情况下使用。

面试题 18. 请Java代码实现观察者模式的案例？

推荐指数： ★★★★ 试题难度： 高难 试题类型： 八股文 ▸

试题回答参考思路：

抽象被观察者

```
//（抽象被观察者）接口, 让（具体观察者）WeatherData 来实现
public interface Subject {
public void registerObserver(Observer o);
public void removeObserver(Observer o);
public void notifyObservers();
}
```

具体被观察者

```
/**
* 类是核心
* 1. 包含最新的天气情况信息
* 2. 含有 观察者集合，使用ArrayList管理
* 3. 当数据有更新时，就主动的调用 ArrayList, 通知所有的（接入方）就看到最新的信息
* @author Administrator
*
*/
///具体被观察者
public class WeatherData implements Subject {
private float temperatrue;
private float pressure;
private float humidity;
//观察者集合
private ArrayList observers;
//加入新的第三方
public WeatherData() {
observers = new ArrayList();
}
public float getTemperature() {
return temperatrue;
}
public float getPressure() {
return pressure;
}
```

```
public float getHumidity() {
return humidity;
}
public void dataChange() {
//调用 接入方的 update
notifyObservers();
}
//当数据有更新时，就调用 setData
public void setData(float temperature, float pressure, float humidity) {
this.temperatrue = temperature;
this.pressure = pressure;
this.humidity = humidity;
//调用dataChange， 将最新的信息 推送给 接入方 currentConditions
dataChange();
}
//注册一个观察者
@Override
public void registerObserver(Observer o) {
// TODO Auto-generated method stub
observers.add(o);
}
//移除一个观察者
@Override
public void removeObserver(Observer o) {
// TODO Auto-generated method stub
if(observers.contains(o)) {
observers.remove(o);
}
}
//遍历所有的观察者，并通知
@Override
public void notifyObservers() {
// TODO Auto-generated method stub
for (int i = 0; i < observers.size(); i++) {
observers.get(i).update(this.temperatrue, this.pressure, this.humidity);
}
}
}
```

抽象观察者

```
//抽象观察者接口，由具体观察者来实现（）
public interface Observer {
public void update(float temperature, float pressure, float humidity);
}
```

具体观察者

```
//具体观察者
```

```
public class BaiduSite implements Observer {
// 温度，气压，湿度
private float temperature;
private float pressure;
private float humidity;
// 更新 天气情况，是由 WeatherData 来调用，我使用推送模式
public void update(float temperature, float pressure, float humidity) {
this.temperature = temperature;
this.pressure = pressure;
this.humidity = humidity;
display();
}
// 显示
public void display() {
System.out.println("===百度网站====");
System.out.println("***百度网站 气温：" + temperature + "***");
System.out.println("***百度网站 气压: " + pressure + "***");
System.out.println("***百度网站 湿度: " + humidity + "***");
}
}
//具体观察者
public class CurrentConditions implements Observer {
// 温度，气压，湿度
private float temperature;
private float pressure;
private float humidity;
// 更新 天气情况，是由 WeatherData 来调用，我使用推送模式
public void update(float temperature, float pressure, float humidity) {
this.temperature = temperature;
this.pressure = pressure;
this.humidity = humidity;
display();
}
// 显示
public void display() {
System.out.println("***Today mTemperature: " + temperature + "***");
System.out.println("***Today mPressure: " + pressure + "***");
System.out.println("***Today mHumidity: " + humidity + "***");
}
}
```

测试

```
public static void main(String[] args) {
// TODO Auto-generated method stub
//创建一个WeatherData
WeatherData weatherData = new WeatherData();
//创建观察者
```

```
CurrentConditions currentConditions = new CurrentConditions();
BaiduSite baiduSite = new BaiduSite();
//注册到weatherData
weatherData.registerObserver(currentConditions);
weatherData.registerObserver(baiduSite);
//测试
System.out.println("通知各个注册的观察者, 看看信息");
weatherData.setData(10f, 100f, 30.3f);
weatherData.removeObserver(currentConditions);
//测试
System.out.println();
System.out.println("通知各个注册的观察者, 看看信息");
weatherData.setData(10f, 100f, 30.3f);
}
```

面试题 19. 简述什么是装饰模式?

推荐指数: ★★★ 试题难度: 初级 试题类型: 八股文 ▸

试题回答参考思路:

装饰模式（Decorator Pattern） 也称为 包装模式（Wrapper Pattern），是结构型设计模式之一，其使用一种对客户端透明的方式来动态地扩展对象的功能，经常作为 继承 的一种替代方案之一。
装饰模式 的核心：功能扩展。
使用 装饰模式 可以透明且动态地扩展类的功能。

面试题 20. 请Java代码实现装饰者模式的案例？

推荐指数: ★★★★ 试题难度: 中级 试题类型: 八股文 ▸

试题回答参考思路:

以 装饰模式 的角度来看待上述例子，人 属于 Component 角色；男人 属于 ConcreteComponent 角色，因此，男人 是被装饰对象；穿衣服 是功能扩展，属于 Decorator 角色；而 穿裤子，穿内衣，穿外套 是3个具体功能扩展，均属于 ConcreteDecorator 角色;

```
class Client {
public static void main(String[] args) {
IPerson person = new Man();
person.dress();

System.out.println("----------------------");
System.out.println("增加裤子适配器");
person = new TrousersDecorator(person);
person.dress();
```

```
System.out.println("----------------------");
System.out.println("再增加内衣适配器");
person = new UnderClothesDecorator(person);
person.dress();

System.out.println("----------------------");
System.out.println("再增加外套适配器");
person = new OvercoatDecorator(person);
person.dress();
}

// 抽象组件（Component）
interface IPerson {
void dress();
}

// 具体组件（ConcreteComponent）,即被修饰者
static class Man implements IPerson {

@Override
public void dress() {
System.out.println("穿了内裤!");
}
}

// 抽象适配器（Decorator）,接收一个具体的Component，本身也是一个Component
static abstract class ClothesDecorator implements IPerson {
protected IPerson mPerson;

// 构造方法强制子类构造必须传入一个IPerson
public ClothesDecorator(IPerson person) {
this.mPerson = person;
}

@Override
public void dress() {
this.mPerson.dress();
}
}

//具体装饰器（ConcreteDecorator）：裤子装饰器
static class TrousersDecorator extends ClothesDecorator {

public TrousersDecorator(IPerson person) {
super(person);
}
```

```
@Override
public void dress() {
super.dress();
this.dressTrousers();
}

private void dressTrousers() {
System.out.println("穿上裤子了!");
}
}

//具体装饰器（ConcreteDecorator）：内衣装饰器
static class UnderClothesDecorator extends ClothesDecorator {
public UnderClothesDecorator(IPerson person){
super(person);
}

@Override
public void dress() {
super.dress();
this.dressUnderClothes();
}

private void dressUnderClothes(){
System.out.println("穿上内衣了!");
}
}

//具体装饰器（ConcreteDecorator）：外套装饰器
static class OvercoatDecorator extends ClothesDecorator {
public OvercoatDecorator(IPerson person){
super(person);
}

@Override
public void dress() {
super.dress();
this.dressOvercoat();
}

private void dressOvercoat(){
System.out.println("穿上外套了!");
}
}
}
```

上面的代码中Man这个IPerson本身只穿了件内裤，衣裳不整，不能入目-_-，因此我们使用装饰器分别为其增加了穿裤子，内衣，外套的功能，最终完成了穿整套衣服的功能。

代码运行结果如下：

```
穿了内裤!
----------------------
增加裤子适配器
穿了内裤!
穿上裤子了!
----------------------
再增加内衣适配器
穿了内裤!
穿上裤子了!
穿上内衣了!
```

上面代码中客户是为new Man()这个IPerson一件一件的进行衣服试穿，太浪费时间了，我们完全可以使用装饰器嵌套（因为装饰器接收一个IPerson，而自己同时也是一个IPerson，因此完全支持嵌套）模式，这样一次性就穿完，代码如下：

```
class Client {
public static void main(String[] args) {
IPerson person = new OvercoatDecorator(new UnderClothesDecorator(new TrousersDecorator(new Man())));
person.dress();
}
```

结果如下：

```
穿了内裤!
穿上裤子了!
穿上内衣了!
穿上外套了!
```

面试题 21. 简述Java什么是适配器模式？

推荐指数：★★★ 试题难度：中级 试题类型：八股文 ▸

试题回答参考思路：

适配器模式是一种结构型设计模式，其用途是将一个类的接口转换成客户端所期望的另一种接口。适配器模式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。

适配器模式的实现方式
适配器模式通过创建一个实现目标接口的适配器类来实现，该适配器类存有一个对源类的实例的引用，并将请求重定向到源类的方法。通过这种方式，适配器类可以将目标接口和源接口之间的差异隐藏起来，使得它们可以协同工作。

面试题 22. 请Java代码实现适配器模式的案例？

推荐指数： ★★★★ 试题难度： 高难 试题类型： 八股文 ▸

试题回答参考思路：

假设我们有一个MP3播放器和一个普通CD播放器，现在需要将普通CD播放器接口转换为MP3播放器接口，以便让普通CD播放器能够兼容MP3播放器。这个问题可以通过适配器模式来解决。

首先，定义一个目标接口MediaPlayer，该接口定义了MP3播放器所需的所有方法：

```
public interface MediaPlayer {
public void play(String audioType, String fileName);
}
```

然后，定义一个源类AdvancedMediaPlayer，该类定义了普通CD播放器的接口：

```
public interface AdvancedMediaPlayer {
public void playVlc(String fileName);
public void playMp4(String fileName);
}
```

为了将普通CD播放器的接口转换为MP3播放器的接口，我们需要创建一个适配器类MediaAdapter，该类实现了MediaPlayer接口，并使用AdvancedMediaPlayer接口来实现其方法：

```
public class MediaAdapter implements MediaPlayer {
AdvancedMediaPlayer advancedMusicPlayer;
public MediaAdapter(String audioType){
if(audioType.equalsIgnoreCase("vlc") ){
advancedMusicPlayer = new VlcPlayer();
} else if (audioType.equalsIgnoreCase("mp4")){
advancedMusicPlayer = new Mp4Player();
}
}
@Override
public void play(String audioType, String fileName) {
if(audioType.equalsIgnoreCase("vlc")){
advancedMusicPlayer.playVlc(fileName);
}else if(audioType.equalsIgnoreCase("mp4")){
advancedMusicPlayer.playMp4(fileName);
}
}
}
```

最后，我们定义一个客户端类AudioPlayer，该类实现了MediaPlayer接口，并使用MediaAdapter来适配普通CD播放器的接口：

```
public class AudioPlayer implements MediaPlayer {
MediaAdapter mediaAdapter;
@Override
public void play(String audioType, String fileName) {
//播放 mp3 音乐文件的内置支持
if(audioType.equalsIgnoreCase("mp3")){
System.out.println("Playing mp3 file. Name: " + fileName);
}
```

```
//mediaAdapter 提供了播放其他文件格式的支持
else if(audioType.equalsIgnoreCase("vlc")
|| audioType.equalsIgnoreCase("mp4")){
mediaAdapter = new MediaAdapter(audioType);
mediaAdapter.play(audioType, fileName);
}
else{
System.out.println("Invalid media. " + audioType + " format not supported");
}
}
}
```

现在我们可以使用AudioPlayer来播放MP3、VLC或MP4格式的音频文件，无论它们是由MP3播放器还是普通CD播放器提供的。

```
public static void main(String[] args) {
AudioPlayer audioPlayer = new AudioPlayer();
audioPlayer.play("mp3", "beyond the horizon.mp3");
audioPlayer.play("mp4", "alone.mp4");
audioPlayer.play("vlc", "far far away.vlc");
audioPlayer.play("avi", "mind me.avi");
}
```

输出结果如下：

```
Playing mp3 file. Name: beyond the horizon.mp3
Playing mp4 file. Name: alone.mp4
Playing vlc file. Name: far far away.vlc
Invalid media. avi format not supported
```

从输出结果可以看出，适配器模式使得普通CD播放器可以与MP3播放器的接口兼容，从而可以被AudioPlayer所使用