# 设计模式之禅

# 迪米特法则

迪米特法则(Law of Demeter,LoD) 也称为最少知识原则(Least Knowledge Principle,LKP),

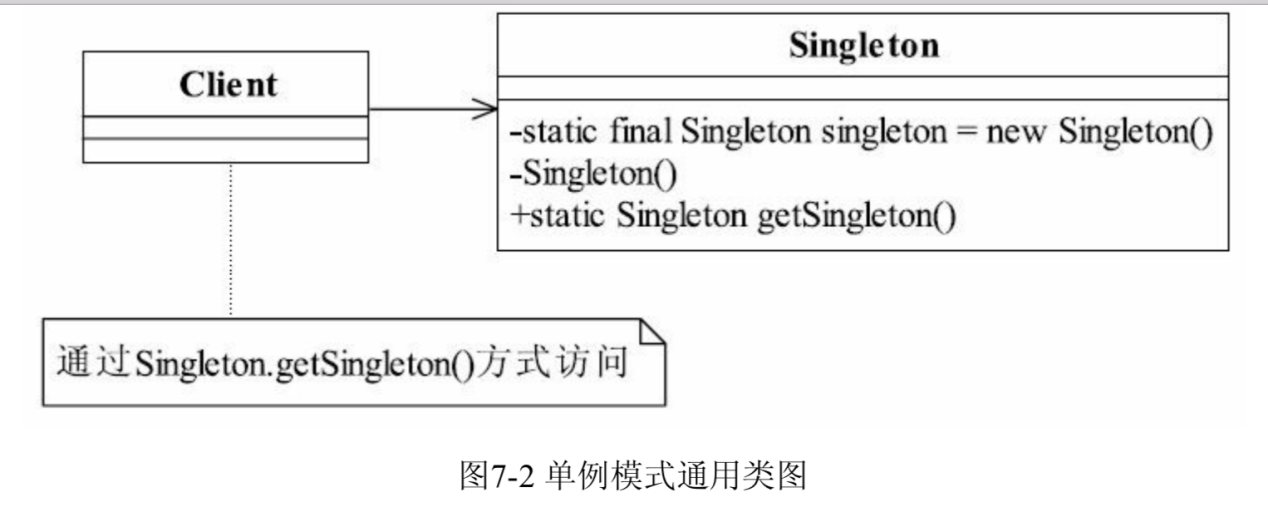
一个对象应该对其他对象有最少的了解。一个类应该对自己需要耦合的或调用的类知道的最少。

# 迪米特法则

# 单例模式

## 7.2 单例模式的定义

确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例



## 7.3 单例模式的应用

### 7.3.1 单例模式的优点

1. 内存中只有一个实例，减少了内存开支，特别是一个对象需要频地创建、销毁时，而且创建或销毁时性能又无法优化，单例模式的优势就非常明显
2. 由于单例模式只生成一个实例，所以减少了系统的性能开销，当一个对象的产生需要比较多的资源时，如读取配置，产生其他依赖对象时，则可以通过在应用启动时直接产生一个单例对象，然后永久驻留内存的方式来解决。
3. 单例模式可以避免对资源的多重占用，例如一个写文件动作，由于只有一个实例存在内存中，避免对同一个资源文件的同时写操作。
4. 单例模式可以在系统设置全局的访问点，优化和共享资源访问，例如可以设计一个单例类，负责所有数据表的映射处理。

### 7.3.2 单例模式的缺点

1. 单例模式一般没有接口，扩展很困难，若要扩展，除了修改代码基本上没有第二种途径可以实现。
2. 单例模式对测试是不利的。
3. 单例模式与单一职责原则有冲突。一个类应该只实现一个逻辑，而不关心它是否是单例的，是不是要单例取决于环境。

## 7.4 单例模式的实现方式

### 7.4.1 饿汉式

|  |
| --- |
| package com.book.study.zen.chapter07;  */\*\*  \* 饿汉式  \*/* public class SingletonObject1 {   private static final SingletonObject1 *instance* = new SingletonObject1();   private SingletonObject1() {  }   public static SingletonObject1 getInstance() {  return *instance*;  }  } |

### 7.4.2 懒汉式：线程不安全的

|  |
| --- |
| package com.book.study.zen.chapter07;  */\*\*  \* 懒汉式:存在多线程安全问题  \*/* public class SingletonObject2 {   private static SingletonObject2 *instance*;   private SingletonObject2(){  }   public static SingletonObject2 getInstance(){  if(null == *instance*){  *instance* = new SingletonObject2();  }  return SingletonObject2.*instance*;  }  } |

### 7.4.3 懒汉式：同步方法

|  |
| --- |
| package com.book.study.zen.chapter07;  */\*\*  \* 懒汉式:方法加同步，解决多线程问题  \*/* public class SingletonObject3 {   private static SingletonObject3 *instance*;   private SingletonObject3(){  }   public synchronized static SingletonObject3 getInstance(){  if(null == *instance*){  *instance* = new SingletonObject3();  }  return SingletonObject3.*instance*;  }   } |

### 7.4.4 懒汉式：double check 同步代码块

依旧存在线程安全问题

|  |
| --- |
| package com.book.study.zen.chapter07;  */\*\*  \* 懒汉式:double check方式解决多线程问题，实际上还是解决不了  \*/* public class SingletonObject4 {    private static SingletonObject4 *instance*;   private SingletonObject4() {   }   public static SingletonObject4 getInstance() {  if (null == *instance*) {  synchronized (SingletonObject4.class) {  if (null == *instance*) {  *instance* = new SingletonObject4();  }  }  }  return SingletonObject4.*instance*;  }  } |

### 7.4.5 懒汉式：double check 同步代码块 volatile关键字

|  |
| --- |
| package com.book.study.zen.chapter07;  */\*\*  \* 懒汉式：用volatile关键字解决double check问题  \*/* public class SingletonObject5 {   // 保证可见性、避免重排序  private static volatile SingletonObject5 *instance*;   private SingletonObject5() {   }   public static SingletonObject5 getInstance() {  if (null == *instance*) {  synchronized (SingletonObject5.class) {  if (null == *instance*) {  *instance* = new SingletonObject5();  }  }  }  return SingletonObject5.*instance*;  }   } |

### 7.4.6 静态内部类

|  |
| --- |
| package com.book.study.zen.chapter07;   */\*\*  \* 静态内部类的方式：当访问SingletonObject6的类时，会先初始化InstanceHolder，于是达到了访问的时候才初始化，而且由于是静态内部类实现的  \* 所以，  \*/* public class SingletonObject6 {   */\*\*  \* 1.类装载  \* 2.初始化  \*/* private SingletonObject6(){   }   private static class InstanceHolder{  private final static SingletonObject6 *intance* = new SingletonObject6();  }   public static SingletonObject6 getInstance(){  return InstanceHolder.*intance*;  }    } |

### 7.4.7 枚举1

|  |
| --- |
| package com.book.study.zen.chapter07;  public class SingletonObject7 {   private SingletonObject7(){   }   private enum Singleton{  *INSTANCE*,;   private final SingletonObject7 instance;   Singleton(){  instance = new SingletonObject7();  }   public SingletonObject7 getInstance(){  return instance;  }  }    public static SingletonObject7 getInstance(){  return Singleton.*INSTANCE*.getInstance();  }   } |

### 7.4.7 枚举2

|  |
| --- |
| package com.book.study.zen.chapter07;  public class DBConnection { }  package com.book.study.zen.chapter07;  */\*\*  \* 枚举实现单例模式  \*/* public enum SingletonObject8 {   *INSTANCE*,  ;   private final DBConnection instance;   private SingletonObject8() {  instance = new DBConnection();   }   public DBConnection getInstance() {  return instance;  }   } |

# 工厂方法模式

## 8.2 工厂方法模式的定义

定义一个用于创建对象的接口，让子类决定实例化哪一个类。工厂方法是一个类的实例化延迟到其子类。

|  |
| --- |
| package com.book.study.zen.chapter08;  public abstract class Product {   // 产品类的公共方法  public void method1(){  // 业务逻辑处理  }   // 抽象方法  public abstract void method2();  }  package com.book.study.zen.chapter08;  public class ConcreteProduct1 extends Product {    @Override  public void method2() {  // 业务逻辑处理  } }  package com.book.study.zen.chapter08;  public class ConcreteProduct2 extends Product {   @Override  public void method2() {  // 业务逻辑处理  } }  package com.book.study.zen.chapter08;  public abstract class Creator {   public abstract <T extends Product> T createProduct(Class<T> c);  }  package com.book.study.zen.chapter08;  public class ConcreteCreator extends Creator {   @Override  public <T extends Product> T createProduct(Class<T> c) {  Product product = null;  try {  product = (Product) Class.*forName*(c.getName()).newInstance();  }catch (Exception e){  // 异常处理   }  return (T) product;  } } |

## 8.3 工厂方法模式的应用

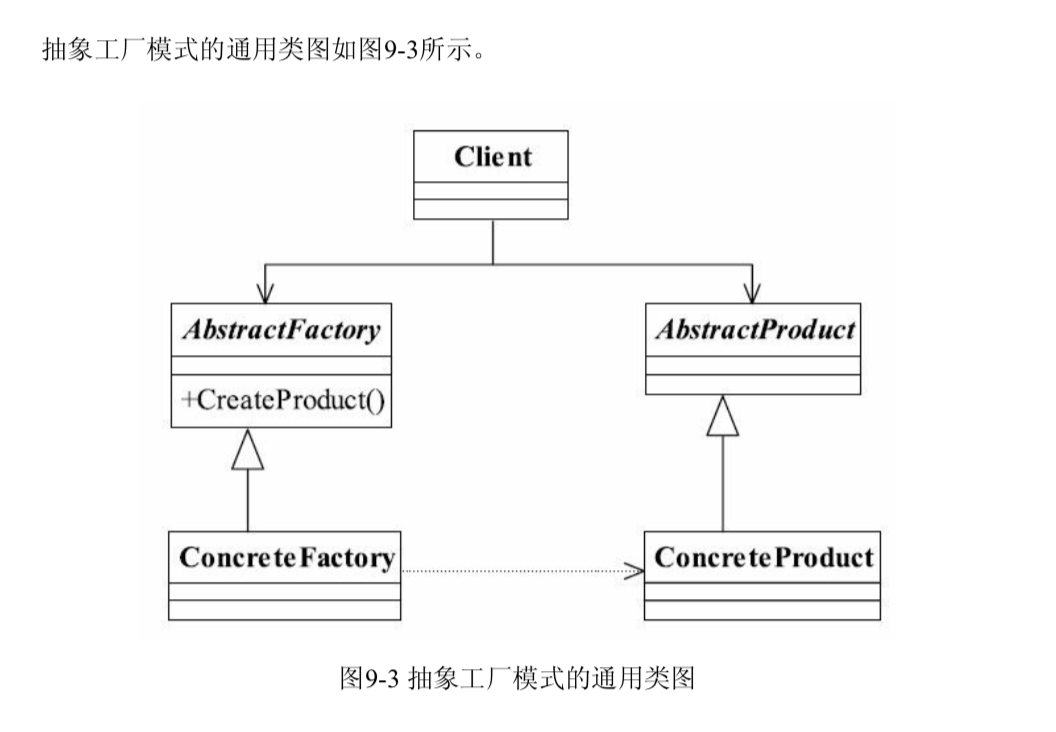
### 8.3.1 工厂方法模式的优点

1. 首先，良好的封装性，代码结构清晰。
2. 其次，工厂方法模式的扩展性非常优秀。
3. 再次，屏蔽产品类。
4. 工厂方法模式是典型的解耦框架。高层模块值不需要知道产品的抽象类，其他的实现类不用关心，复合迪米特法则，我不需要的就不要去交流。也符合依赖倒置原则。

# 抽象工厂

## 9.2 抽象工厂模式的定义

为创建一组相关或相互依赖的对象提供一个接口，而且无须指定他们的具体类



## 9.3 抽象工厂模式的应用

### 9.3.1 抽象工厂模式的优点

1. 封装性，每个产品的实现类不是高层模块要关心的。
2. 产品族类的约束为非公开状态。

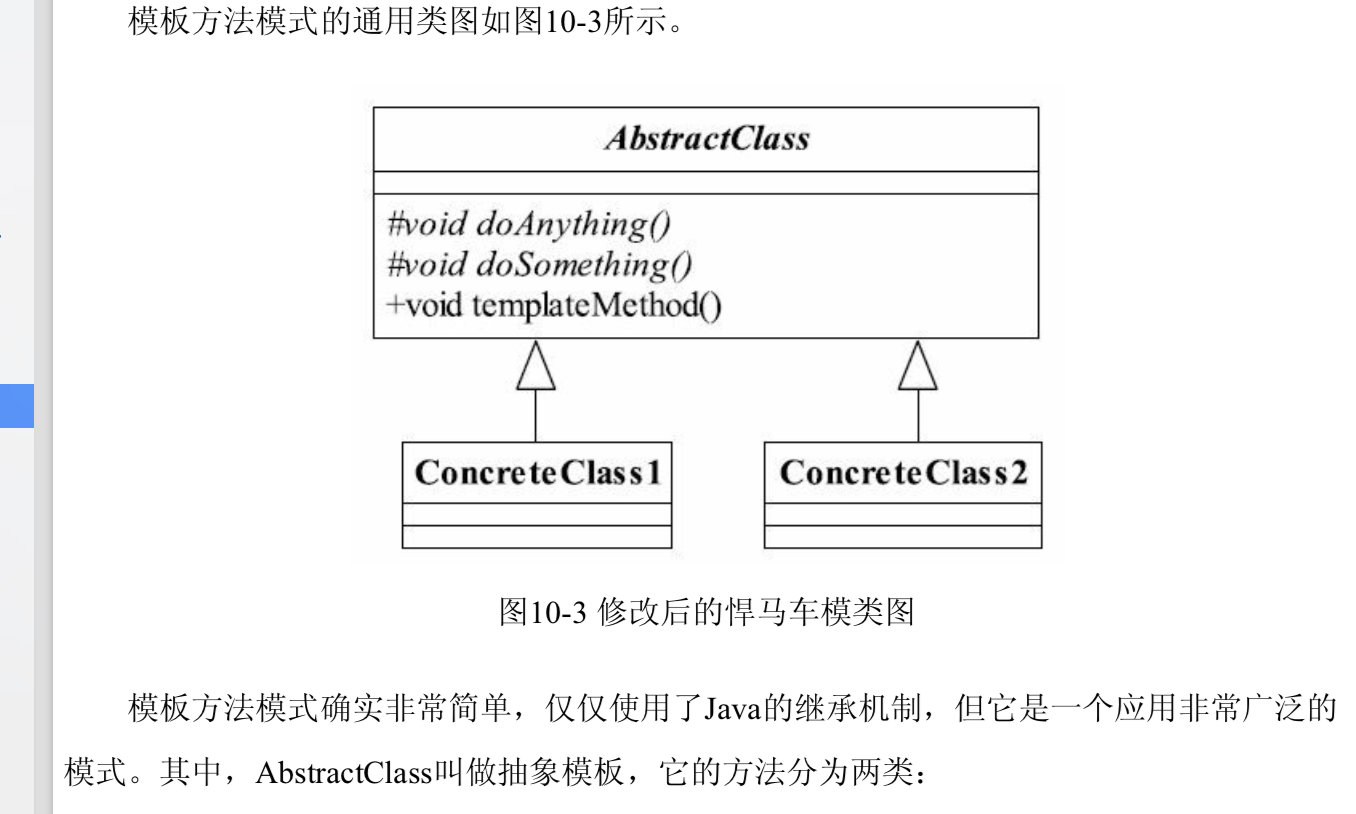
### 9.3.1 抽象工厂模式的缺点

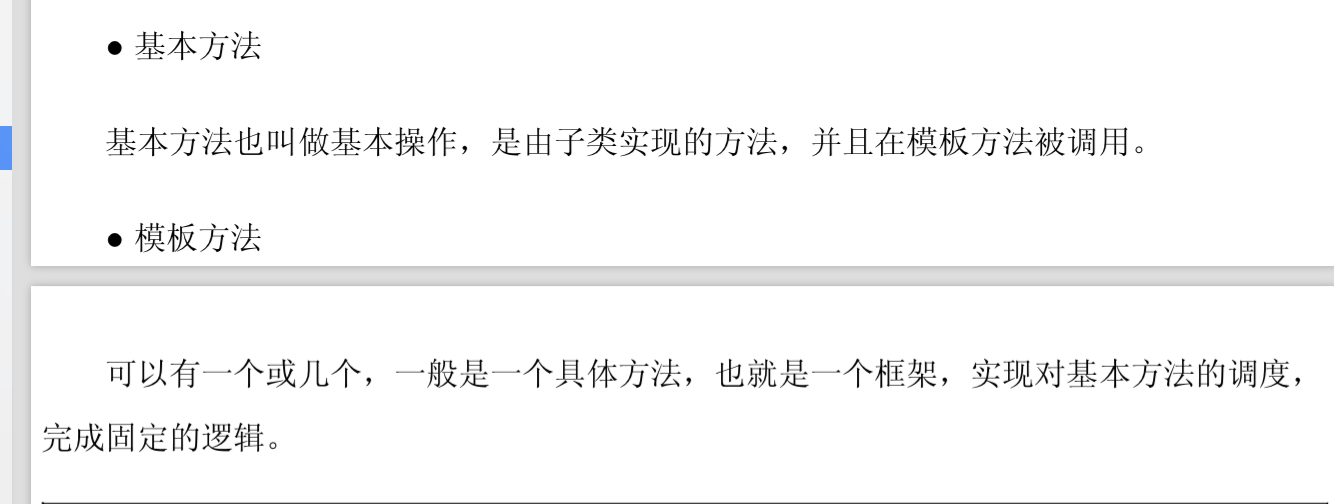
抽象工厂的模式的最大缺点就是产品族的扩展非常困难。

# 模板方法

## 10.2 模板方法的定义

定义一个操作中的算法的框架，而将一些步骤延迟到子类中。使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤





注意：

1. 为了防止恶意的操作，一般模板方法都加上final关键字，不允许被覆盖
2. 抽象模板中的基本方法尽量设计为protected类型，复合迪米特法则，不需要暴露属性或方法精良不要设置为protected类型。实现类若非必要，尽量不要扩大父类中的访问权限。

## 10.3 模板方法模式的应用

优点：

1. 封装不变不封，扩展可变部分
2. 提取公共部分代码，便于维护
3. 行为由父类控制，子类实现

缺点：

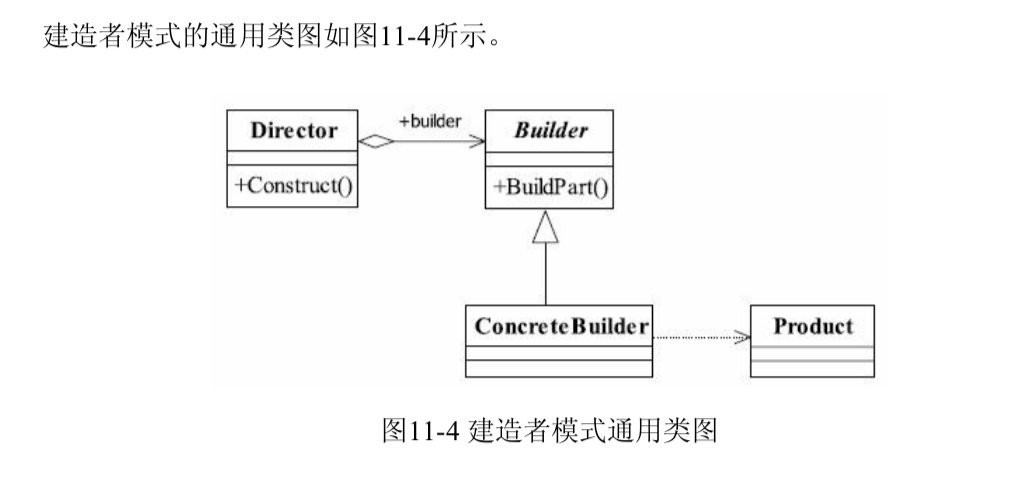
代码阅读起来有难度。

# 建造者模式

## 11.2 建造者模式的定义

建造者模式(Builder Pattern) 也叫做生成器模式，其定义如下：

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示



建造者模式中，4个角色：

1. Product产品类:

通常是实现了模板方法模式，也就是有模板方法和基本方法，这个参考第10章的模板方法模式。

1. Builder抽象建造者

规范产品的组建，一般是由子类实现。例子中的CarBuilder就属于抽象建造者。

1. ConcreteBuilder具体建造者

实现抽象类定意思的所有方法，并且返回一个组建好的对象。例子中的BenzBuilder和BMWBuilder就叔叔于具体建造者。

1. Director导演类

负责安排已有模块的顺序，然后告诉Builder开始建造。

## 11.3 建造者模式的优点

### 11.3.1 建造者模式的优点

1. 封装性

使用建造者模式可以使客户端不必知道产品内部组成的细节，

1. 建造者独立，容易扩展

BenzBuilder和BMWBuilder是相互独立的，对系统的扩展非常有利

1. 便于控制细节风险

由于具体的建造者是独立的，因此可以对建造过程逐步细化，而不对其他的模块产生任何影响

### 11.3.2 建造者模式的使用场景

1. 相同的方法，不同的执行顺序，产生不同的事件结果时，可以采用建造者模式
2. 多个部件或零件，都可以装配到一个对象中，但是产生的运行结果又不相同时，则可以使用该模式
3. 产品类非常复杂，或者产品类中的调用顺序不同产生了不同的效果，这个时候使用建造者模式非常合适
4. 在对象创建过程中会使用到系统中的一些其他对象，这些对象在产品对象的创建过程中不易得到时，也可以采用建造者模式封装该对象的创建过程。这种场景只能是一个补偿方法，因为一个对象不容易获得，而在设计阶段竟然没有发觉。而要通过创建者模式柔滑创建过程，本身已经违反了设计最初目标。

**建造者模式最主要的功能是基本方法的调用顺序安排。通俗的说就是零件的装配，顺序不同产生的对象也不同；而工厂方法侧重点事创建，创建零件是它的主要职责，组装顺序则不是它所关心的。**

## 11.4 建造者模式

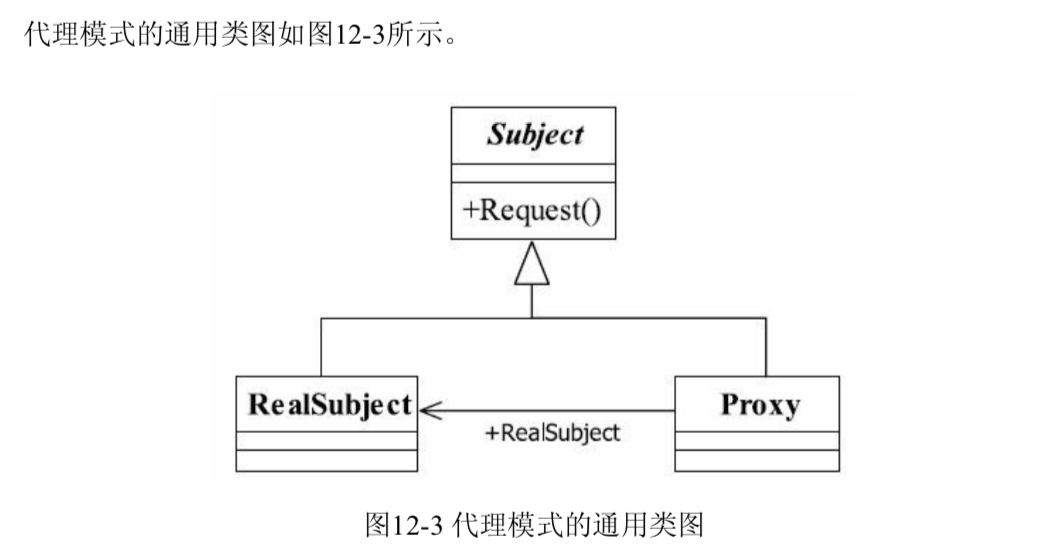
|  |
| --- |
| package com.book.study.zen.chapter11.demob;  public class Product {   public void doSomething(){  // 独立业务处理  }   }  package com.book.study.zen.chapter11.demob;  public abstract class Builder {   // 设置产品的不同部分，以获得不同的产品  public abstract void setPart();   // 建造产品  public abstract Product buildProduct();  }  package com.book.study.zen.chapter11.demob;  public class ConcreteProduct extends Builder {   private Product product = new Product();    @Override  public void setPart() {  */\*\*  \* 产品类内的逻辑处理  \*/* }   // 组建一个产品  @Override  public Product buildProduct() {  return product;  } }  package com.book.study.zen.chapter11.demob;  public class Director {   private Builder builder = new ConcreteProduct();   // 构建不同的产品  public Product getProduct() {  builder.setPart();   */\*\*  \* 设置不同的零件，产生不同的产品  \*/* return builder.buildProduct();  }   } |

# 代理模式

## 12.2 代理模式的定义

代理模式(Proxy pattern)是一个使用率非常高的模式，其定意思如下：

为其他对象提供一种代理以控制这个对象的访问。



1. Subject抽象主题角色

抽象主题类可以是抽象类也可以是接口，是一个最普通的业务类型定义，无特殊要求。

1. RealSubject具体主题角色

也叫做被委托角色，被代理角色。

1. Proxy代理主题角色
2. 也叫做委托类、代理类。它负责对这恩施角色的应用，把所有抽象主题类定义的方法限制委托给真实主题角色实现，并且在真实主题角色处理完毕前后做处理和善后工作

|  |
| --- |
| package com.book.study.zen.chapter12.demob;  public interface Subject {  // 定义一个方法  public void request(); }  package com.book.study.zen.chapter12.demob;  public class RealSubject implements Subject {   @Override  public void request() {  // 业务逻辑处理  } }  package com.book.study.zen.chapter12.demob;  public class Proxy implements Subject {   // 要代理哪个实现类  private Subject subject = null;   // 默认被代理者  public Proxy(){  this.subject = new Proxy();  }   // 通过构造函数传递代理者  @Override  public void request() {  this.before();  this.subject.request();  this.after();  }   // 预处理  private void after() {   }   // 善后处理  private void before() {   }   } |

## 12.3 代理模式的应用

### 12.3.1 代理模式的优点

1. 职责清晰

真实的角色就是实际的业务逻辑，不用关系其他非本质职责的事务，通过后期的代理完成一件事务，附带的结果就是编程简洁清晰

1. 高扩展性

具体主题角色是随时都会发生变化的，只要它实现了接口，

1. 智能化

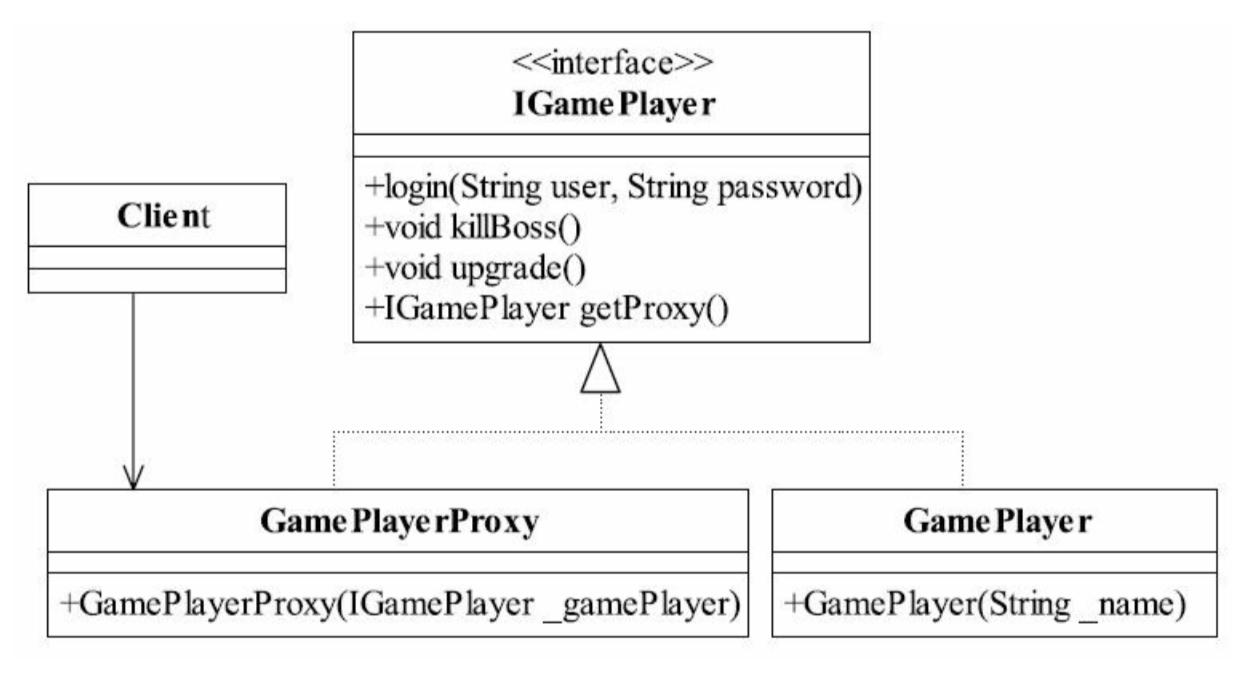
## 12.4 代理模式的扩展

### 12.4.1 普通代理

|  |
| --- |
| package com.book.study.zen.chapter12.democ;  import com.book.study.zen.chapter12.IGamePlayer;  public class GamePlayer implements IGamePlayer {   private String name = "";    // 构造函数限制谁能创建对象，并同时传递姓名  public GamePlayer(IGamePlayer \_gamePlayer, String \_name) throws Exception {  if (\_gamePlayer == null) {  throw new Exception("不能创建真实角色");  } else {  this.name = \_name;  }  }    // 进游戏之前你肯定要登陆吧，这是一个必要条件  @Override  public void login(String user, String password) {  System.*out*.println("登陆名为" + user + "的用户" + this.name + "登陆成功!");  }   // 打怪，最期望的就是杀老怪  @Override  public void killBoss() {  System.*out*.println(this.name + "在打怪");  }   // 升级，升级有很多方法，花钱买是一种，做任务也是一种  @Override  public void upgrade() {  System.*out*.println(this.name + "又升了一级");  } } |

|  |
| --- |
| package com.book.study.zen.chapter12.democ;  import com.book.study.zen.chapter12.IGamePlayer;  public class GamePlayerProxy implements IGamePlayer {   private IGamePlayer gamePlayer = null;   // 通过构造函数传递要对谁进行代练   public GamePlayerProxy(String name) {  try {  this.gamePlayer = new GamePlayer(this,name);  }catch (Exception e){  // *TODO 异常处理* }   }   @Override  public void login(String user, String password) {  this.gamePlayer.login(user,password);  }   // 通过构造函数传递要对谁进行代练  @Override  public void killBoss() {  this.gamePlayer.killBoss();  }   // 通过构造函数传递要对谁进行代练  @Override  public void upgrade() {  this.gamePlayer.upgrade();  } } |

### 12.4.2 强制代理

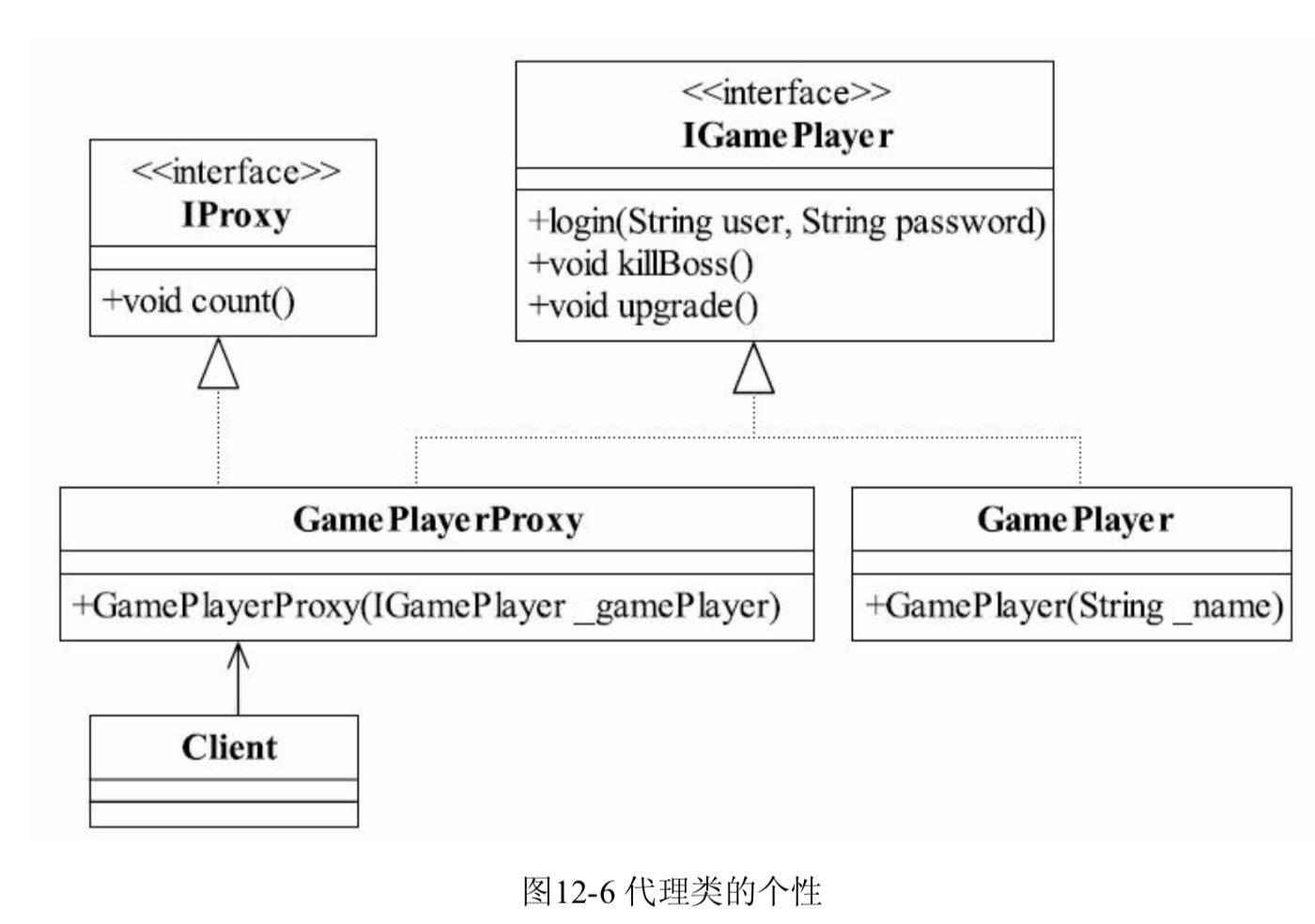


|  |
| --- |
| package com.book.study.zen.chapter12;  public interface INewGamePlayer {   // 登陆游戏  public void login(String user, String password);   // 杀怪，网络游戏的主要特色  public void killBoss();   // 升级  public void upgrade();    public INewGamePlayer getProxy();  } |

|  |
| --- |
| package com.book.study.zen.chapter12.demod;  import com.book.study.zen.chapter12.INewGamePlayer;  public class GamePlayer implements INewGamePlayer {   private String name = "";   private INewGamePlayer proxy = null;   public GamePlayer(String name) {  this.name = name;  }   @Override  public void login(String user, String password) {  if (this.isProxy()) {  System.*out*.println("登陆名为" + user + "的用户" + this.name + "登陆成功!");   } else {  System.*out*.println("请使用指定的代理访问");  }  }   private boolean isProxy() {  return this.proxy != null;  }   @Override  public void killBoss() {  if (this.isProxy()) {  System.*out*.println(this.name + "在打怪");  } else {  System.*out*.println("请使用指定的代理访问");  }  }   @Override  public void upgrade() {  if (this.isProxy()) {  System.*out*.println(this.name + " 又升了一级");  } else {  System.*out*.println("请使用指定的代理访问");  }   }   @Override  public INewGamePlayer getProxy() {  this.proxy = new GamePlayerProxy(this);  return this.proxy;  } } |

|  |
| --- |
| package com.book.study.zen.chapter12.demod;   import com.book.study.zen.chapter12.INewGamePlayer;  public class GamePlayerProxy implements INewGamePlayer {    private INewGamePlayer gamePlayer = null;   public GamePlayerProxy(INewGamePlayer gamePlayer) {  this.gamePlayer = gamePlayer;  }   @Override  public void login(String user, String password) {  this.gamePlayer.login(user,password);  }   @Override  public void killBoss() {  this.gamePlayer.killBoss();  }   @Override  public void upgrade() {  this.gamePlayer.upgrade();  }   @Override  public INewGamePlayer getProxy() {  return this;  } } |

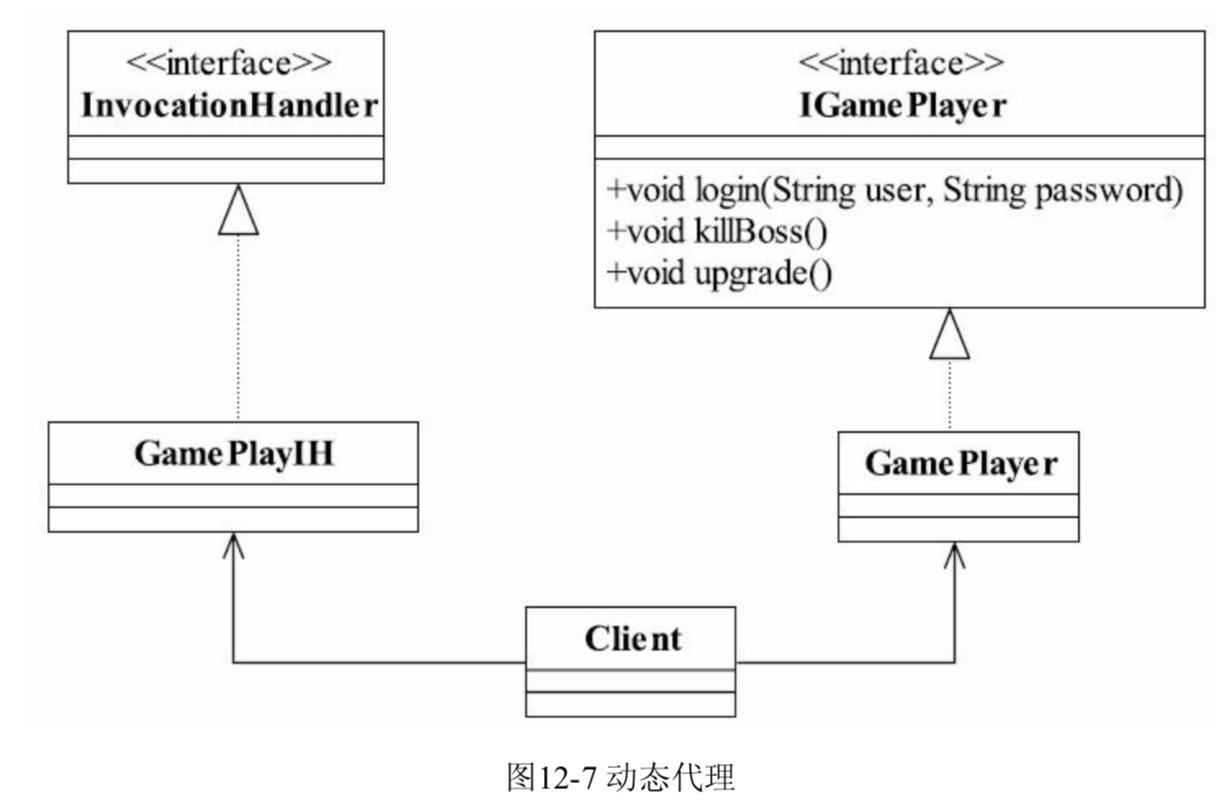
### 12.4.3 代理是有个性的



|  |
| --- |
| package com.book.study.zen.chapter12.demoe;  import com.book.study.zen.chapter12.IGamePlayer;  public class GamePlayer implements IGamePlayer {   private String name = "";    // 构造函数限制谁能创建对象，并同时传递姓名  public GamePlayer(String \_name) {  this.name = \_name;  }    // 进游戏之前你肯定要登陆吧，这是一个必要条件  @Override  public void login(String user, String password) {  System.*out*.println("登陆名为" + user + "的用户" + this.name + "登陆成功!");  }   // 打怪，最期望的就是杀老怪  @Override  public void killBoss() {  System.*out*.println(this.name + "在打怪");  }   // 升级，升级有很多方法，花钱买是一种，做任务也是一种  @Override  public void upgrade() {  System.*out*.println(this.name + "又升了一级");  } } |

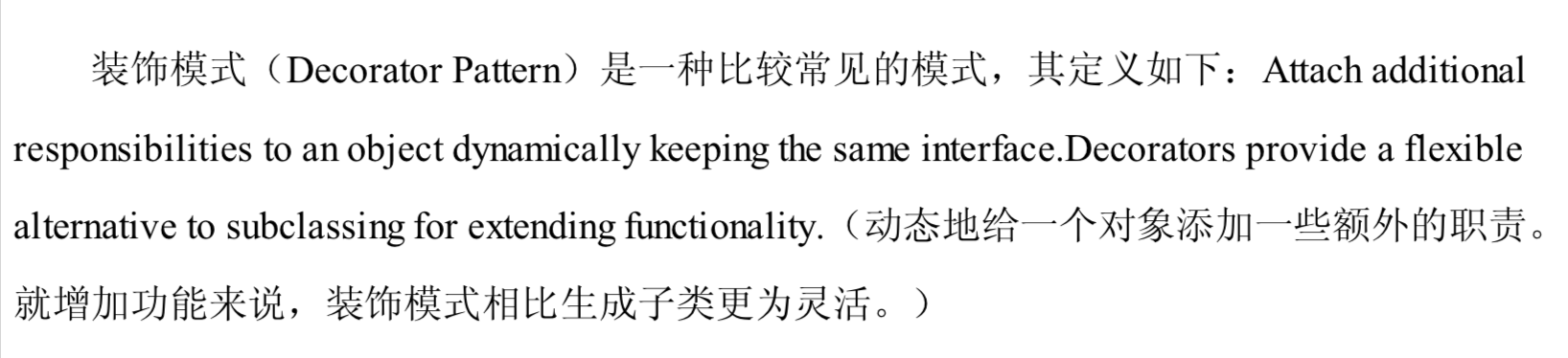
|  |
| --- |
| package com.book.study.zen.chapter12.democ;  import com.book.study.zen.chapter12.IGamePlayer;  public class GamePlayerProxy implements IGamePlayer {   private IGamePlayer gamePlayer = null;   // 通过构造函数传递要对谁进行代练   public GamePlayerProxy(String name) {  try {  this.gamePlayer = new GamePlayer(this,name);  }catch (Exception e){  // *TODO 异常处理* }   }   @Override  public void login(String user, String password) {  this.gamePlayer.login(user,password);  }   // 通过构造函数传递要对谁进行代练  @Override  public void killBoss() {  this.gamePlayer.killBoss();  }   // 通过构造函数传递要对谁进行代练  @Override  public void upgrade() {  this.gamePlayer.upgrade();  } } |

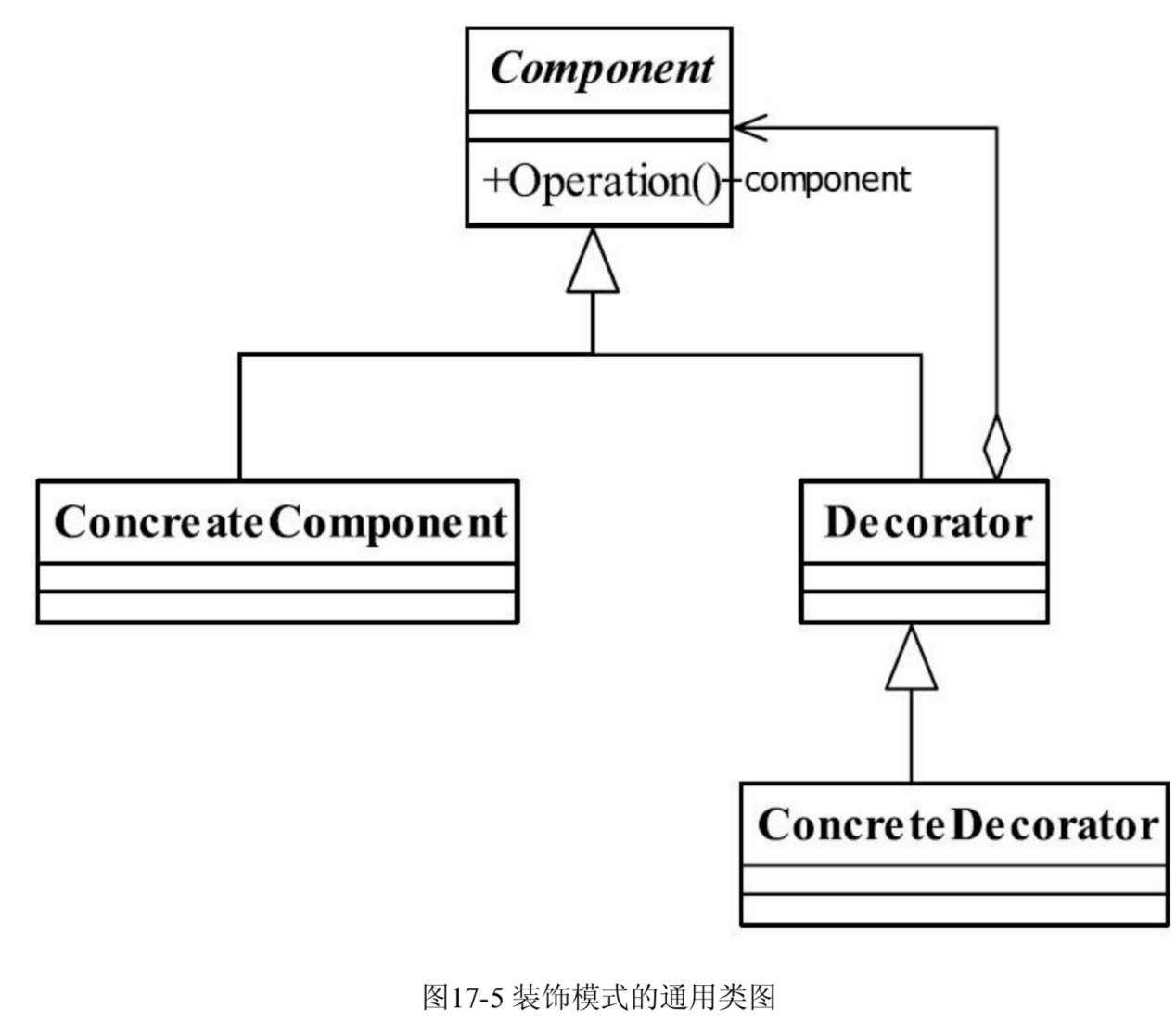
### 12.4.3 动态代理

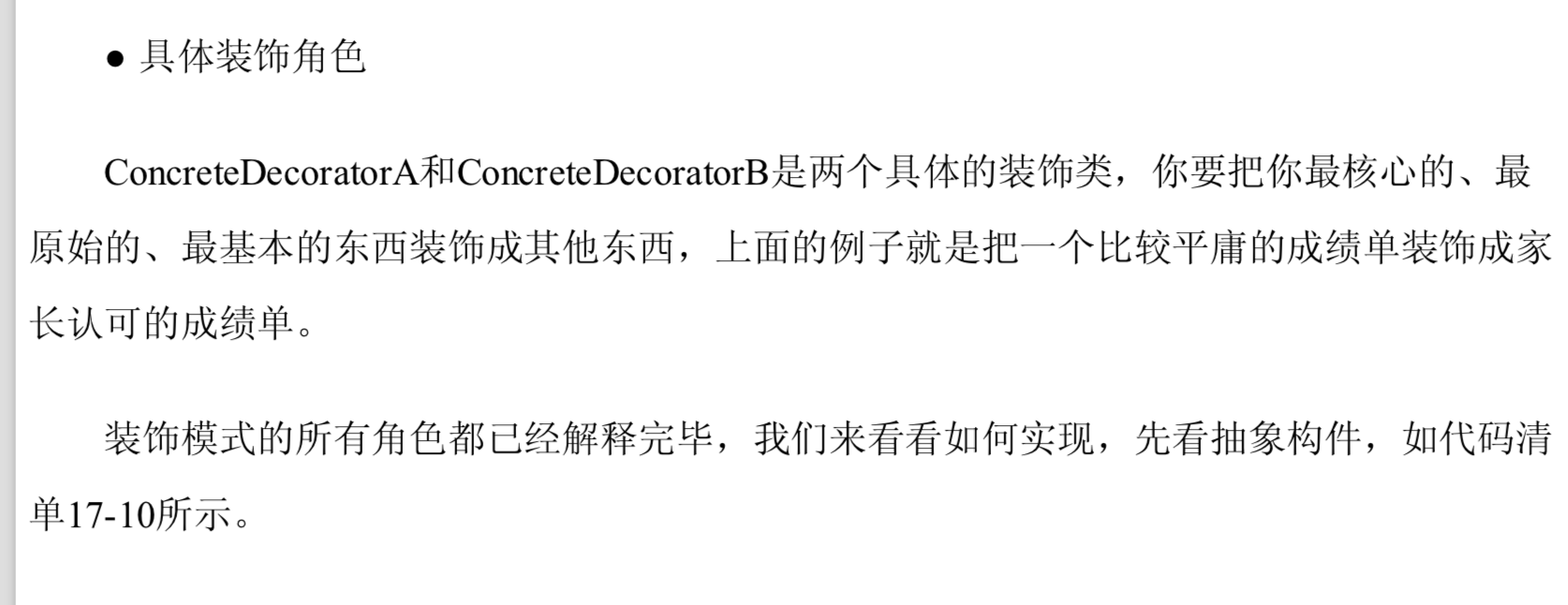


# 装饰者模式

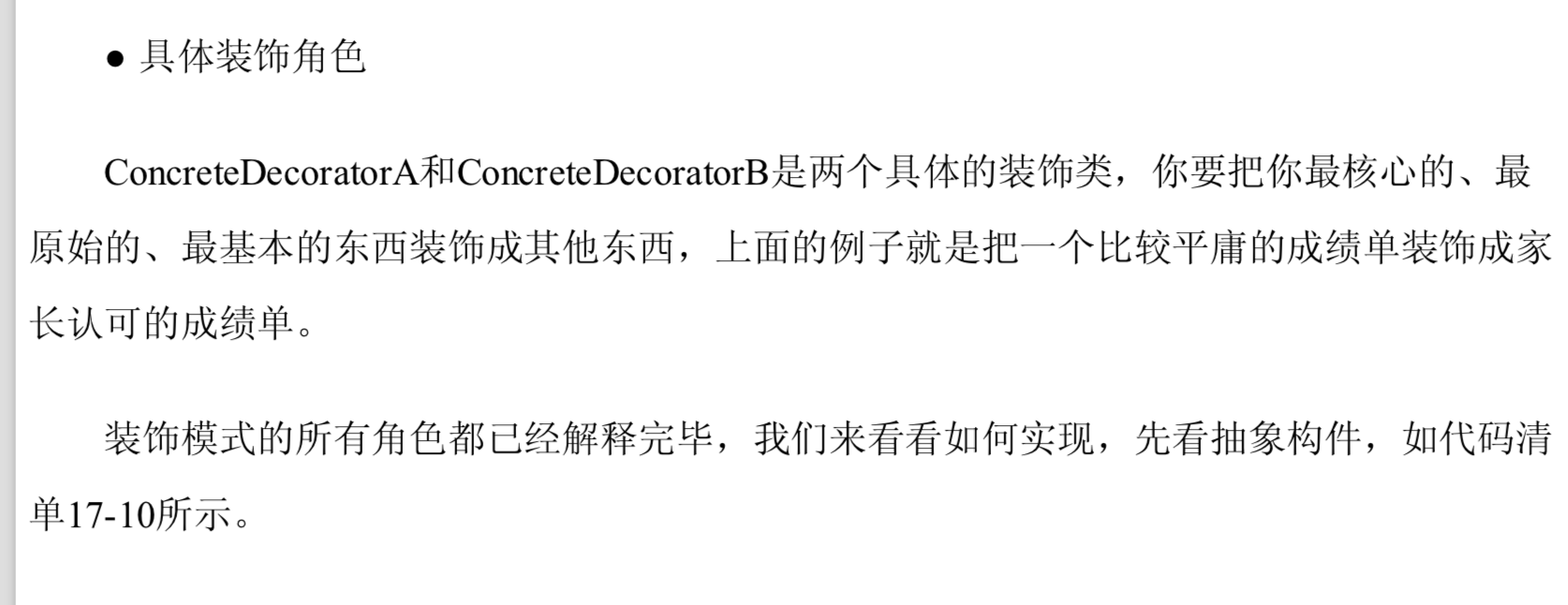
动态地给一个对象添加一些额外的职责

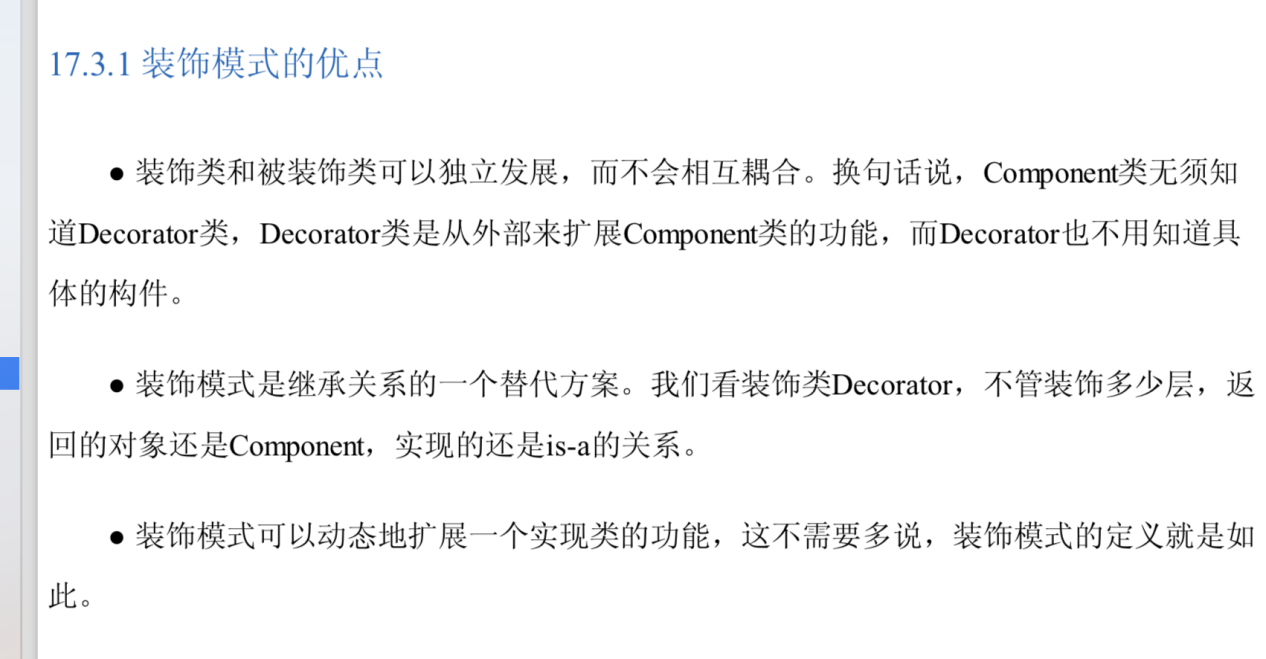


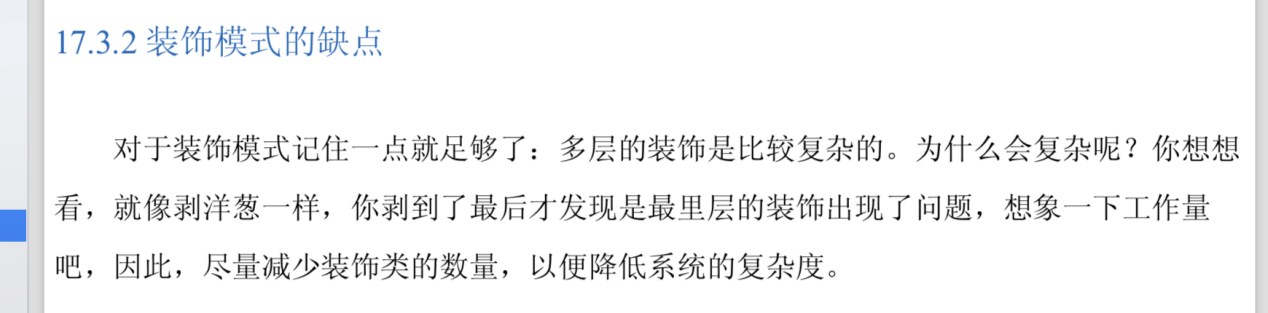


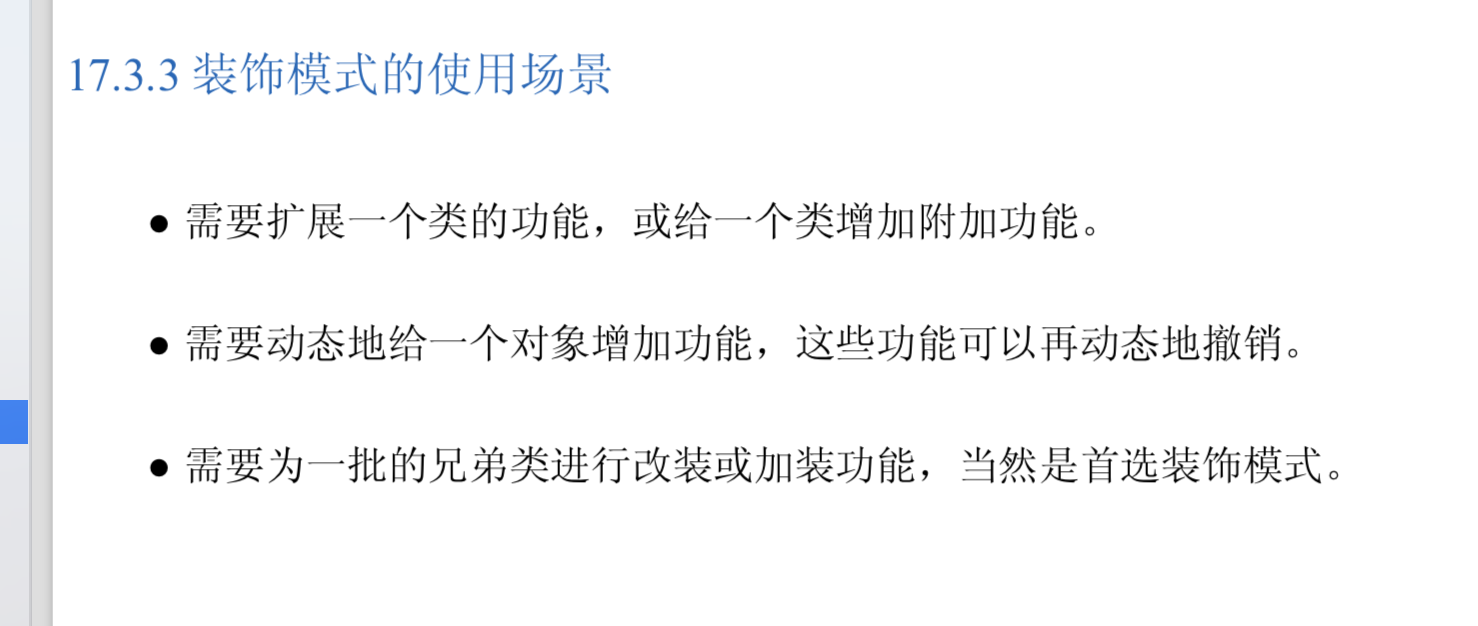




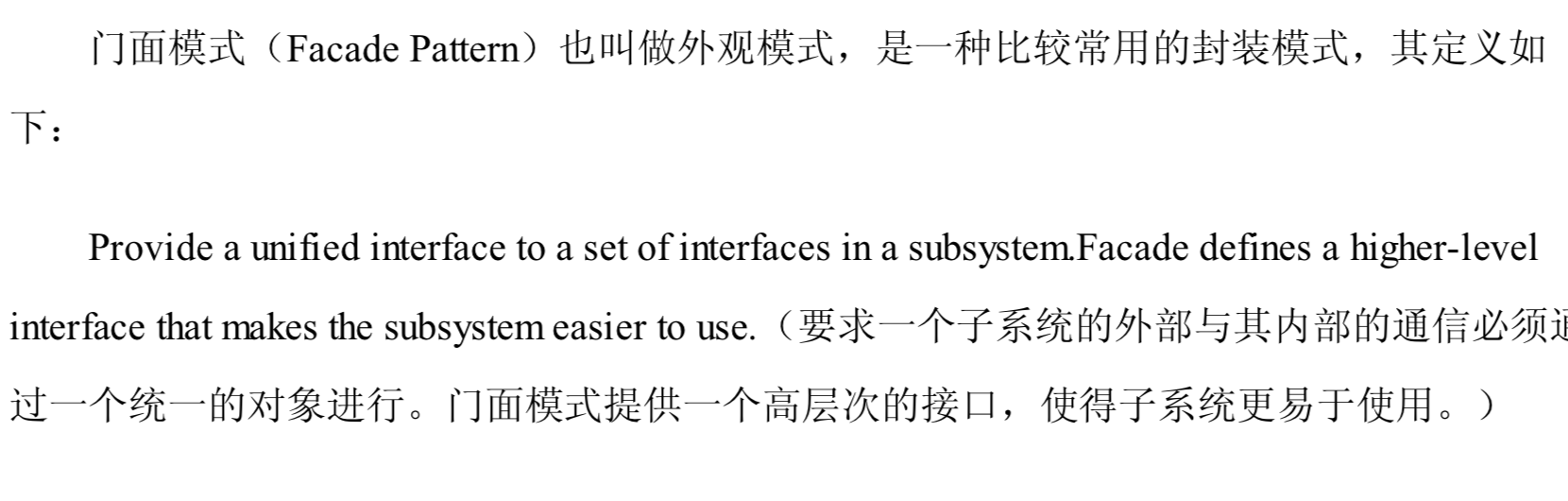


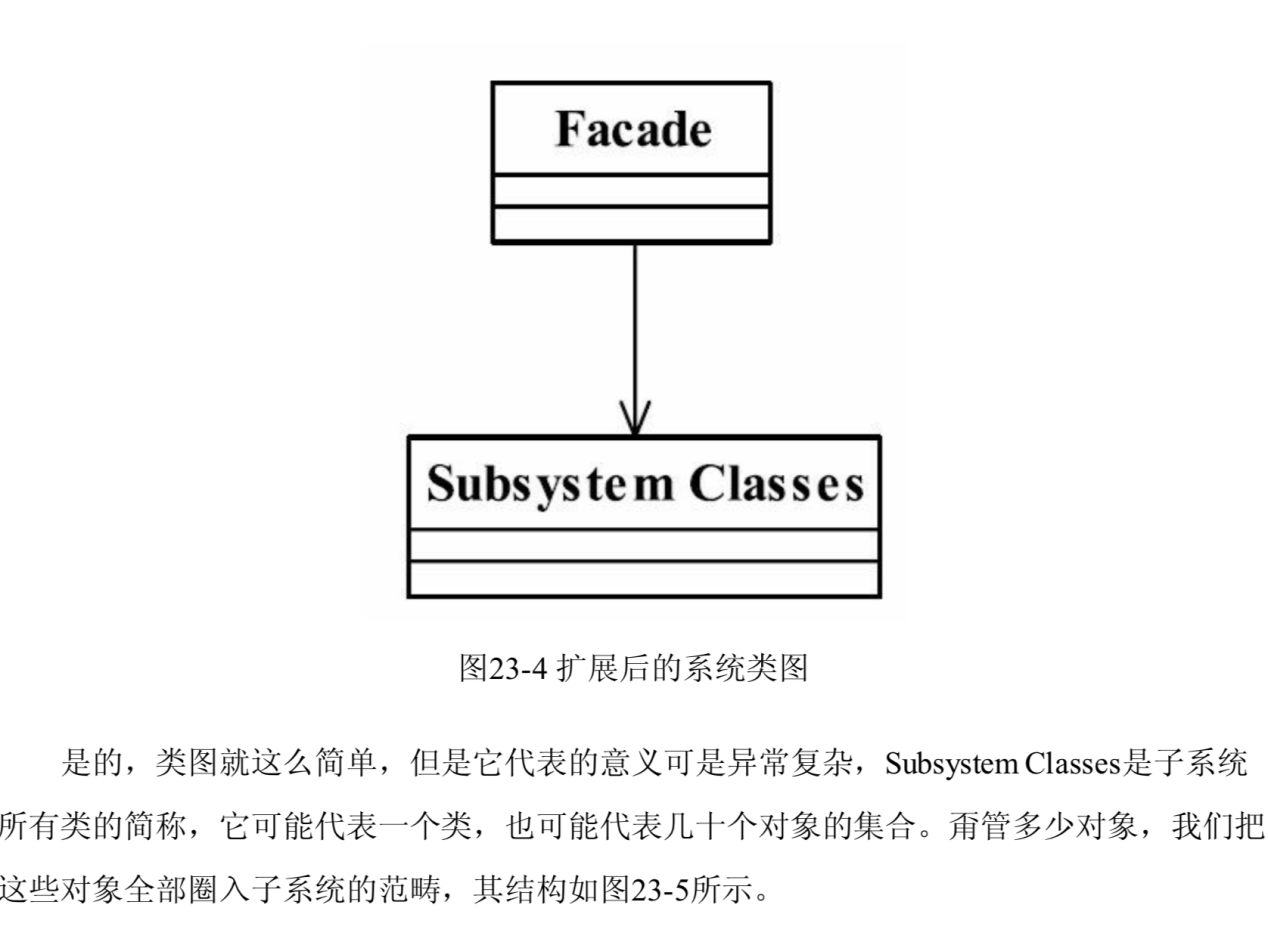






# 门面模式

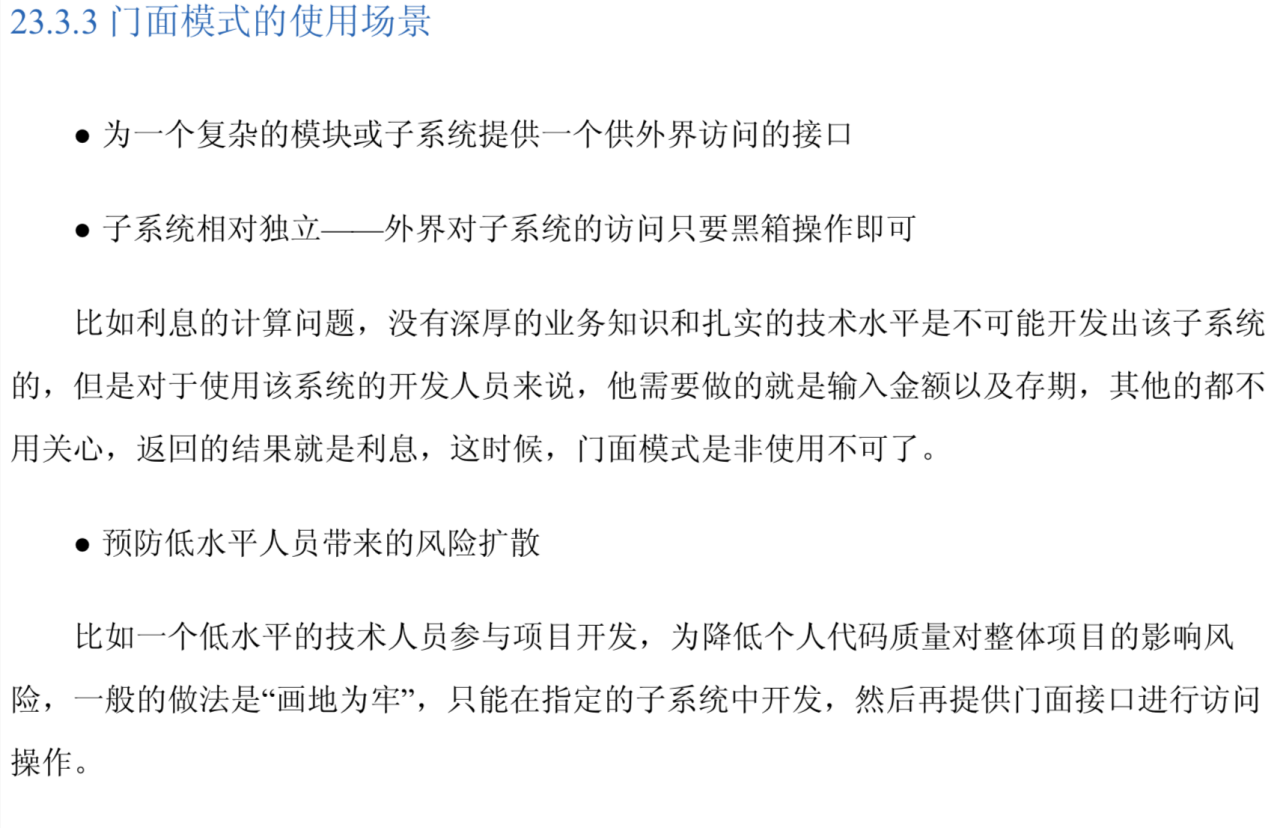




优点：

1. 减少系统的相互依赖
2. 提高了灵活性
3. 提高安全性

缺点：不符合开闭原则，对修改关闭，对扩展开放



案例：tomcat的Request，Response就使用了门面模式，netty的ByteBuf对ByteBuffer等的封装，也是门面模式。其实传统的mvc架构中的。controller，service，dao，这三层也体现了门面模式，controller是对外暴露的服务，service组合dao层处理业务逻辑。