《Head First 设计模式》笔记

-- Dawei Min

1 模式

1.1 策略模式

策略模式：定义了算法簇，分别封装起来，让他们之间可以互相替换，此模式让算法的变化独立于使用算法的客户。

1.2 观察者模式

观察者（Observer）模式定义了对象之间的一对多依赖，这样一来，当一个对象改变状态时，它的所有依赖者都会收到通知并自动更新。

1.3 装饰者模式

装饰者模式：动态地将责任附加到对象上。若要扩展功能，装饰者提供了比继承更有弹性的替代方案。

装饰者一般对组件的客户是透明的，除非客户程序依赖于组件的具体类型。

装饰者会导致设计中出现许多小对象，如果过度使用，会让程序变得很复杂。

经典案例：Java I/O

1.4 工厂模式

工厂方法模式：定义了一个创建对象的接口，但由子类决定要实例化的类是哪一个。工厂方法让类的实例化推迟到子类。

抽象工厂模式：提供一个接口，用于创建相关或依赖对象的家族，而不需要明确指定具体类。

创建对象的区别：工厂方法模式采用的方法是继承，抽象工厂模式通过对象组合。

1.5 单件模式

单件模式（Singleton）：确保一个类只有一个实例，并提供一个全局访问点。

1.6 命令模式

命令模式：将“请求”封装成对象，以便使用不同的请求、队列或者日志来参数化其他对象。命令模式也支持可撤销的操作。

应用：日程安排（Scheduler）、线程池、工作队列等。

1.7 适配器模式

适配器模式：将一个类的接口，转换成客户期望的另一个接口。适配器让原本接口不兼容的类可以合作无间。

1.8 外观模式

外观模式（Facade）：提供了一个统一的接口，用来访问子系统同中的一群接口。外观定义了一个高层接口，让子系统更容易使用。

特点：

外观没有“封装”子系统，外观只是提供简化的接口。所以用户如果觉得有必要，依然可以使用子系统的类。

外观可以附加“聪明”的功能，让使用子系统更方便（这里的“聪明”的功能个人理解为：通过子系统也能实现的，但是外观把调用的逻辑等实现了）。

每个子系统可以创建多个外观。

外观模式允许你将客户从组件的子系统中解耦。如果客户只是使用外观，那么修改子系统的实现，只要外观不变即可。

与适配器模式的差异主要在意图上。适配器的意图是将接口转换成不同接口，而外观意图是简化接口。

1.9 模板方法模式

模板方法模式：在一个方法中定义一个算法的骨架，而将一些步骤延迟到子类中。模板使得子类可以在不改变算法结构的情况下，重新定义算法中的某些步骤。

1.10 迭代器模式

迭代器模式：提供一种方法顺序访问一个聚合对象中的各个元素，而又不暴露其内部的表示。

1.11 组合模式

组合模式（Composite Pattern）允许你将对象组合成树型结构来表现“整体/部分”层次结构。组合能让客户以一致的方式处理个别对象以及对象组合。

1.12 状态模式

状态模式：允许对象在内部状态改变时改变它的行为，对象看起来好像修改了它的类。

1.13 代理模式

代理模式为另一个对象提供一个替身或占位符以控制对这个对象的访问。

使用代理模式创建代表（representative）对象，让代表对象控制某对象的访问，被代表的对象可以是远程对象、创建开销大的对象或需要安全控制的对象。

装饰者模式、适配器模式、代理模式的区别：

装饰者模式为对象增加行为，适配器会改变对象的接口，代理实现相同的接口控制对象的访问。

1.14 复合模式

复合模式结合两个或以上的模式，组成一个解决方案，解决一再发生的一般性问题。

例如：MVC 用了观察者模式、策略模式和组合模式。

1.15 桥接模式

桥接模式（Bridge Pattern）不只改变你的实现，也改变你的抽象。

抽象和实现可以独立扩展，不会影响到对方。适合使用在需要跨越多个平台的图形和窗口系统上。缺点是增加了复杂度。

1.16 生成器模式

生成器模式（Builder Pattern）封装一个产品的构造过程，并允许按步骤构造。

1.17 责任链模式

当你想要让一个以上的对象有机会能够处理某个请求的时候，就使用责任链模式（Chain of Responsibility Pattern）。

经常被使用在窗口系统中，处理鼠标和键盘子类的事件。

1.18 蝇量模式

如果想让某个类的一个实例能用来提供许多“虚拟实例”，就使用蝇量模式（Flyweight Pattern）。

减少运行时对象实例的个数，节省内存。但是单个的逻辑实例将无法拥有独立而不同的行为。

1.19 解释器模式

使用解释器模式（Interpreter Pattern）为语言创建解释器。

当你需要实现一个简单的语言时可以使用解释器。当语法的规则数目太大时，这个模式可能会变得非常繁杂。

1.20 中介者模式

使用中介者模式（Mediator Pattern）来集中相关对象之间复杂的沟通和控制方式。

中介者常常被用来协调相关的 GUI 组件。中介者模式的缺点是，如果设计不当，中介者对象本身会变得过于复杂。

1.21 备忘录模式

当你需要让对象返回之前的状态时（例如，你的用户请求“撤销”），就使用备忘录模式（MementoPattern）。

在 Java 中，可以考虑使用序列化（serialization）机制存储系统的状态。

1.22 原型模式

当创建给定类的实例的过程很昂贵或很复杂时，就使用原型模式（Prototype Pattern）。

当一个复杂的类层次中，当系统必须从其中的许多类型创建新对象时，可以

考虑原型。但是对象的复制有时相当复杂。

1.23 访问者模式

当你想要为一个对象的组合增加新的能力，且封装并不重要时，就使用访问者模式（Visitor Pattern）。

当采用访问者模式的时候，就会打破组合类的封装。因为游走的功能牵涉其中，所以对组合结构的改变就更加困难。

2 原则

2.1 针对接口编程

针对接口编程而不是针对实现编程。“针对接口编程”这句话，可以更明确地说成“变量的声明类型应该是超类型，通常是一个抽象类或者接口或者是一个接口”。针对接口编程时利用多态特性。

2.2 多用组合，少用继承

组合可以是系统具有很大的弹性，可以“在运行时动态得改变行为”。

2.3 松耦合设计

为了交互对象之间的松耦合设计而努力。松耦合的设计之所以能让我们建立有弹性的 OO 系统，能够应对变化，是因为**对象之间的互相依赖降到了最低**。

观察者模式提供了一种对象设计，让主题和观察者之间松耦合。

2.4 开放-关闭原则

类应该对扩展开放，对修改关闭。

我们的目标是运行类容易扩展，在不修改现有代码的情况下，就可以搭配新的行为。虽然似乎有点矛盾，但是确实有些技术可以实现。在选择需要被扩展的代码部分要小心。每个地方都采用开放-关闭原则是一种浪费，也没有必要，还会导致代码变得复杂且难以理解。

比如：装饰者模式、观察者模式。

2.5 依赖倒置原则

依赖倒置原则（Dependency Inversion）：要依赖抽象，不要依赖具体类。

这个原则说明了：不能让高层组件依赖低层组件，“两者”都应该依赖于抽象。

比如：工厂模式。

2.6 最小知识原则

最小知识原则（Least Knowledge）：当我们设计一个系统时，对任何对象都要注意它交互的类有哪些，不要让多个类耦合在一起，免得修改一部分会影响到其它部分。

反例：return　a.getB().getC();

正例：return a.getC();

如果我们调用从另一个调用中返回的对象的方法，相当于向另一个对象的子部分发请求，增加了我们直接认识的对象数目。

做法

在对象方法内，我们只应该调用属于以下范围的方法：

（1）该对象本身

（2）方法传递进来参数的对象

（3）此方法创建或实例化的任何对象（方法内的局部对象）、

（4）对象的任何组件（组件：HAS-A 关系）

2.7 好莱坞原则

别调用（打电话给）我们，我们会调用（打电话给）你。高层组件对待低层组件的方式是“别调用我们，我们会调用你”。

例如：模板方法模式。

2.8 单一责任

一个类应该只有一个引起变化的原因。

如果一个类有两个改变的原因，那么这会使得将来该类的变化机率上升，而当它真的改变时，你的设计同时有两个方面将会受影响。迭代器模式就是将遍历的责任从集合类中分离出来。