目录

[一、 设计模式介绍 1](#_Toc496110248)

[1. 简介 1](#_Toc496110249)

[2. 设计框架 1](#_Toc496110250)

[3. 设计原则 2](#_Toc496110251)

[4. 基本模式 4](#_Toc496110252)

1. 设计模式介绍

# 简介

设计模式（Design Pattern）是一套被反复使用、多数人知晓的、经过分类的、代码设计经验的总结。

使用设计模式的目的：为了代码可重用性、让代码更容易被他人理解、保证代码可靠性。 设计模式使代码编写真正工程化；设计模式是软件工程的基石脉络，如同大厦的结构一样。

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。

设计模式是一套被反复使用的、多数人知晓的、经过分类编目的、代码设计经验的总结。使用设计模式是为了重用代码、让代码更容易被他人理解、保证代码可靠性。毫无疑问，设计模式于己于他人于系统都是多赢的，设计模式使代码编制真正工程化，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周围不断重复发生的问题，以及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。

# 设计框架

可复用面向对象软件系统一般划分为两大类：应用程序工具箱和框架（Framework），我们平时开发的具体软件都是应用程序，Java的API属于工具箱；而框架是构成一类特定软件可复用设计的一组相互协作的类，EJB（EnterpriseJavaBeans）是Java应用于企业计算的框架。

框架通常定义了应用体系的整体结构类和对象的关系等等设计参数，以便于具体应用实现者能集中精力于应用本身的特定细节。框架主要记录软件应用中共同的设计决策，框架强调设计复用，因此框架设计中必然要使用设计模式。

另外，设计模式有助于对框架结构的理解，成熟的框架通常使用了多种设计模式,如果你熟悉这些设计模式，毫无疑问，你将迅速掌握框架的结构，我们一般开发者如果突然接触EJB、Spring等框架，会觉得特别难学、难掌握，那么转而先掌握设计模式，无疑是给了你剖析EJB或J2EE系统的一把利器。

# 设计原则

为什么要提倡“Design Pattern呢？根本原因是为了代码复用，增加可维护性。那么怎么才能实现代码复用呢？面向对象有几个原则：单一职责原则 （Single Responsiblity Principle SRP）开闭原则（Open Closed Principle，OCP）、里氏代换原则（Liskov Substitution Principle，LSP）、依赖倒转原则（Dependency Inversion Principle，DIP）、接口隔离原则（Interface Segregation Principle，ISP）、合成/聚合复用原则（Composite/Aggregate Reuse Principle，CARP）、最小知识原则（Principle of Least Knowledge，PLK，也叫迪米特法则）。开闭原则具有理想主义的色彩，它是面向对象设计的终极目标。其他几条，则可以看做是开闭原则的实现方法。

设计模式就是实现了这些原则，从而达到了代码复用、增加可维护性的目的。

开闭原则：实现热插拔，提高扩展性。

里氏代换原则：实现抽象的规范，实现子父类互相替换；

依赖倒转原则：针对接口编程，实现开闭原则的基础；

接口隔离原则：降低耦合度，接口单独设计，互相隔离；

迪米特法则，又称不知道原则：功能模块尽量独立；

合成复用原则：尽量使用聚合，组合，而不是继承；

## 3.1开闭原则

此原则是由Bertrand Meyer提出的。原文是：“Software entities should be open for extension,but closed for modification”。就是说模块应对扩展开放，而对修改关闭。模块应尽量在不修改原（是“原”，指原来的代码）代码的情况下进行扩展。

工厂模式是对具体产品进行扩展，有的项目可能需要更多的扩展性，要对这个“工厂”也进行扩展，那就成了“抽象工厂模式”。

## 3.2里氏代换原则

里氏代换原则是由Barbara Liskov提出的。如果调用的是父类的话，那么换成子类也完全可以运行。

可以说：里氏代换原则是继承复用的一个基础。

## 3.3依赖倒转原则

指在软件里面，把父类都替换成它的子类，程序的行为没有变化。简单的说，子类型能够替换掉它们的父类型。依赖性倒转其实可以说是面向对象设计的标志，用哪种语言编程并不是很重要。

## 3.4接口隔离原则

定制服务的例子，每一个接口应该是一种角色，不多不少，不干不该干的事，该干的事都要干。

## 3.5合成/聚合复用

合成/聚合复用原则（Composite/Aggregate Reuse Principle，CARP）经常又叫做合成复用原则。合成/聚合复用原则就是在一个新的对象里面使用一些已有的对象，使之成为新对象的一部分；新的对象通过向这些对象的委派达到复用已有功能的目的。它的设计原则是：要尽量使用合成/聚合，尽量不要使用继承。

就是说要少用继承，多用合成关系来实现。我曾经这样写过程序：有几个类要与数据库打交道，就写了一个数据库操作的类，然后别的跟数据库打交道的类都继承这个。结果后来，我修改了数据库操作类的一个方法，每个类都需要改动。“牵一发而动全身”！面向对象是要把波动限制在尽量小的范围。

在Java中，应尽量针对Interface编程，而非实现类。这样，更换子类不会影响调用它方法的代码。要让各个类尽可能少的跟别人联系，“不要与陌生人说话”。这样，城门失火，才不至于殃及池鱼。扩展性和维护性才能提高。

## 3.6最少知识原则

也叫迪米特法则。不要和陌生人说话，即一个对象应对其他对象有尽可能少的了解。

## 3.7综述

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。

## 3.8模式名称

一个助记名，它用一两个词来描述模式的问题、解决方案和效果。命名一个新的模式增加了我们的设计词汇。设计模式允许我们在较高的抽象层次上进行设计。基于一个模式词汇表，我们自己以及同事之间就可以讨论模式并在编写文档时使用它们。模式名可以帮助我们思考，便于我们与其他人交流设计思想及设计结果。找到恰当的模式名也是我们设计模式编目工作的难点之一。

## 3.9问题

描述了应该在何时使用模式。它解释了设计问题和问题存在的前因后果，它可能描述了特定的设计问题，如怎样用对象表示算法等。也可能描述了导致不灵活设计的类或对象结构。有时候，问题部分会包括使用模式必须满足的一系列先决条件。

## 3.10解决方案

描述了设计的组成成分，它们之间的相互关系及各自的职责和协作方式。因为模式就像一个模板，可应用于多种不同场合，所以解决方案并不描述一个特定而具体的设计或实现，而是提供设计问题的抽象描述和怎样用一个具有一般意义的元素组合（类或对象组合）来解决这个问题。

## 3.11效果

描述了模式应用的效果及使用模式应权衡的问题。尽管我们描述设计决策时，并不总提到模式效果，但它们对于评价设计选择和理解使用模式的代价及好处具有重要意义。软件效果大多关注对时间和空间的衡量，它们也表述了语言和实现问题。因为复用是面向对象设计的要素之一，所以模式效果包括它对系统的灵活性、扩充性或可移植性的影响，显式地列出这些效果对理解和评价这些模式很有帮助。

# 基本模式

## 4.1综述

设计模式分为四种类型，共33种。

**创建型模式：5种 -->对象怎么来**

工厂模式（Factory Pattern）

抽象工厂模式（Abstract Factory Pattern）

单例模式（Singleton Pattern）

建造者模式（Builder Pattern）

这些设计模式提供了一种在创建对象的同时隐藏创建逻辑的方式，而不是使用 new 运算符直接实例化对象。这使得程序在判断针对某个给定实例需要创建哪些对象时更加灵活。

**结构型模式：8种 -->对象和谁有关**

原型模式（Prototype Pattern）

适配器模式（Adapter Pattern）

桥接模式（Bridge Pattern）

过滤器模式（Filter、Criteria Pattern）

组合模式（Composite Pattern）

装饰器模式（Decorator Pattern）

外观模式（Facade Pattern）

享元模式（Flyweight Pattern）

代理模式（Proxy Pattern）

这些设计模式关注类和对象的组合。继承的概念被用来组合接口和定义组合对象获得新功能的方式。

**行为型模式：12种 -->对象与对象在干嘛**

责任链模式（Chain of Responsibility Pattern）

命令模式（Command Pattern）

解释器模式（Interpreter Pattern）

迭代器模式（Iterator Pattern）

中介者模式（Mediator Pattern）

备忘录模式（Memento Pattern）

观察者模式（Observer Pattern）

状态模式（State Pattern）

空对象模式（Null Object Pattern）

策略模式（Strategy Pattern）

模板模式（Template Pattern）

访问者模式（Visitor Pattern）

这些设计模式特别关注对象之间的通信。

**J2EE 模式：8种 -->对象合起来要干嘛**

MVC 模式（MVC Pattern）

业务代表模式（Business Delegate Pattern）

组合实体模式（Composite Entity Pattern）

数据访问对象模式（Data Access Object Pattern）

前端控制器模式（Front Controller Pattern）

拦截过滤器模式（Intercepting Filter Pattern）

服务定位器模式（Service Locator Pattern）

传输对象模式（Transfer Object Pattern）

这些设计模式特别关注表示层。这些模式是由 Sun Java Center 鉴定的。



**按字典序排列简介如下：**

Abstract Factory（抽象工厂模式）：提供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。

Adapter（适配器模式）：将一个类的接口转换成客户希望的另外一个接口。Adapter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。

Bridge（桥接模式）：将抽象部分与它的实现部分分离，使它们都可以独立地变化。

Builder（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示。

Chain of Responsibility（责任链模式）：为解除请求的发送者和接收者之间耦合，而使多个对象都有机会处理这个请求。将这些对象连成一条链，并沿着这条链传递该请求，直到有一个对象处理它。

Command（命令模式）：将一个请求封装为一个对象，从而使你可用不同的请求对客户进行参数化；对请求排队或记录请求日志，以及支持可取消的操作。

Composite（组合模式）：将对象组合成树形结构以表示“部分-整体”的层次结构。它使得客户对单个对象和复合对象的使用具有一致性。

Decorator（装饰模式）：动态地给一个对象添加一些额外的职责。就扩展功能而言， 它比生成子类方式更为灵活。

Facade（外观模式）：为子系统中的一组接口提供一个一致的界面，Facade模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。

Factory Method（工厂模式）：定义一个用于创建对象的接口，让子类决定将哪一个类实例化。Factory Method使一个类的实例化延迟到其子类。

Flyweight（享元模式）：运用共享技术有效地支持大量细粒度的对象。

Interpreter（解析器模式）：给定一个语言, 定义它的文法的一种表示，并定义一个解释器, 该解释器使用该表示来解释语言中的句子。

Iterator（迭代器模式）：提供一种方法顺序访问一个聚合对象中各个元素，而又不需暴露该对象的内部表示。

Mediator（中介模式）：用一个中介对象来封装一系列的对象交互。中介者使各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。

Memento（备忘录模式）：在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。这样以后就可将该对象恢复到保存的状态。

Observer（观察者模式）：定义对象间的一种一对多的依赖关系,以便当一个对象的状态发生改变时,所有依赖于它的对象都得到通知并自动刷新。

Prototype（原型模式）：用原型实例指定创建对象的种类，并且通过拷贝这个原型来创建新的对象。

Proxy（代理模式）：为其他对象提供一个代理以控制对这个对象的访问。

Singleton（单例模式）：保证一个类仅有一个实例，并提供一个访问它的全局访问点。 单例模式是最简单的设计模式之一，但是对于Java的开发者来说，它却有很多缺陷。在九月的专栏中，David Geary探讨了单例模式以及在面对多线程（multi-threading）、类装载器（class loaders）和序列化（serialization）时如何处理这些缺陷。

State（状态模式）：允许一个对象在其内部状态改变时改变它的行为。对象看起来似乎修改了它所属的类。

Strategy（策略模式）：定义一系列的算法,把它们一个个封装起来, 并且使它们可相互替换。本模式使得算法的变化可独立于使用它的客户。

Template Method（模板方法模式）：定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。Template Method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。

Visitor（访问者模式）：表示一个作用于某对象结构中的各元素的操作。它使你可以在不改变各元素的类的前提下定义作用于这些元素的新操作。

## 4.2工厂模式（Factory）

**意图**

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。Factory Method 使一个类的实例化延迟到其子类。

**适用性**

当一个类不知道它所必须创建的对象的类的时候。

当一个类希望由它的子类来指定它所创建的对象的时候。

当类将创建对象的职责委托给多个帮助子类中的某一个，并且你希望将哪一个帮助子类是代理者这一信息局部化的时候。

## 4.3抽象工厂模式（Abstract Factory）

**意图**

提供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。

**适用性**

一个系统要独立于它的产品的创建、组合和表示时。

一个系统要由多个产品系列中的一个来配置时。

当你要强调一系列相关的产品对象的设计以便进行联合使用时。

当你提供一个产品类库，而只想显示它们的接口而不是实现时。

## 4.4建造者模式

**意图**

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。

**适用性**

当创建复杂对象的算法应该独立于该对象的组成部分以及它们的装配方式时。

当构造过程必须允许被构造的对象有不同的表示时。

## 4.5原型模式

**意图**

用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。

**适用性**

当要实例化的类是在运行时刻指定时，例如，通过动态装载；或者

为了避免创建一个与产品类层次平行的工厂类层次时；或者

当一个类的实例只能有几个不同状态组合中的一种时。建立相应数目的原型并克隆它们可能比每次用合适的状态手工实例化该类更方便一些。

## 4.6单例模式

**意图**

保证一个类仅有一个实例，并提供一个访问它的全局访问点。

**适用性**

当类只能有一个实例而且客户可以从一个众所周知的访问点访问它时。

当这个唯一实例应该是通过子类化可扩展的，并且客户应该无需更改代码就能使用一个扩展的实例时。

## 4.7适配器模式

**意图**

将一个类的接口转换成另外一个客户希望的接口。Adapter 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。

**适用性**

你想使用一个已经存在的类，而它的接口不符合你的需求。

你想创建一个可以复用的类，该类可以与其他不相关的类或不可预见的类（即那些接口可能不一定兼容的类）协同工作。

（仅适用于对象Adapter）你想使用一些已经存在的子类，但是不可能对每一个都进行子类化以匹配它们的接口。对象适配器可以适配它的父类接口。

## 4.8桥接模式

**意图**

将抽象部分与它的实现部分分离，使它们都可以独立地变化。

**适用性**

你不希望在抽象和它的实现部分之间有一个固定的绑定关系。例如这种情况可能是因为，在程序运行时刻实现部分应可以被选择或者切换。

类的抽象以及它的实现都应该可以通过生成子类的方法加以扩充。这时B r i d g e 模式使你可以对不同的抽象接口和实现部分进行组合，并分别对它们进行扩充。

对一个抽象的实现部分的修改应对客户不产生影响，即客户的代码不必重新编译。

（C++）你想对客户完全隐藏抽象的实现部分。在C++中，类的表示在类接口中是可见的。

有许多类要生成。这样一种类层次结构说明你必须将一个对象分解成两个部分。Rumbaugh称这种类层次结构为“嵌套的普化”（nested generalizations ）。

你想在多个对象间共享实现（可能使用引用计数），但同时要求客户并不知道这一点。一个简单的例子便是Coplien的String类，在这个类中多个对象可以共享同一个字符串表示（StringRep）。

## 4.9组合模式

**意图**

将对象组合成树形结构以表示“部分-整体”的层次结构。C o m p o s i t e 使得用户对单个对象和组合对象的使用具有一致性。

**适用性**

你想表示对象的部分—整体层次结构。

你希望用户忽略组合对象与单个对象的不同，用户将统一地使用组合结构中的所有对象。

## 4.10装饰模式

**意图**

动态地给一个对象添加一些额外的职责。就增加功能来说，Decorator模式相比生成子类更为灵活。

**适用性**

在不影响其他对象的情况下，以动态、透明的方式给单个对象添加职责。

处理那些可以撤消的职责。

当不能采用生成子类的方法进行扩充时。一种情况是，可能有大量独立的扩展，为支持每一种组合将产生大量的子类，使得子类数目呈爆炸性增长。另一种情况可能是因为类定义被隐藏，或类定义不能用于生成子类。

## 4.11门面模式

**意图**

为子系统中的一组接口提供一个一致的界面，Facade模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。

**适用性**

当你要为一个复杂子系统提供一个简单接口时。子系统往往因为不断演化而变得越来越复杂。大多数模式使用时都会产生更多更小的类。这使得子系统更具可重用性，也更容易对子系统进行定制，但这也给那些不需要定制子系统的用户带来一些使用上的困难。Facade可以提供一个简单的缺省视图，这一视图对大多数用户来说已经足够，而那些需要更多的可定制性的用户可以越过Facade层。

客户程序与抽象类的实现部分之间存在着很大的依赖性。引入Facade将这个子系统与客户以及其他的子系统分离，可以提高子系统的独立性和可移植性。

当你需要构建一个层次结构的子系统时，使用门面模式定义子系统中每层的入口点。如果子系统之间是相互依赖的，你可以让它们仅通过Facade进行通讯，从而简化了它们之间的依赖关系。

## 4.12享元模式

**意图**

运用共享技术有效地支持大量细粒度的对象。

**适用性**

一个应用程序使用了大量的对象。

完全由于使用大量的对象，造成很大的存储开销。

对象的大多数状态都可变为外部状态。

如果删除对象的外部状态，那么可以用相对较少的共享对象取代很多组对象。

应用程序不依赖于对象标识。由于Flyweight对象可以被共享，对于概念上明显有别的对象，标识测试将返回真值。

## 4.13代理模式

**意图**

为其他对象提供一种代理以控制对这个对象的访问。

**适用性**

在需要用比较通用和复杂的对象指针代替简单的指针的时候，使用Proxy模式。下面是一些可以使用Proxy模式常见情况：

远程代理（Remote Proxy）为一个对象在不同的地址空间提供局部代表。

虚代理（Virtual Proxy）根据需要创建开销很大的对象。

保护代理（Protection Proxy）控制对原始对象的访问。保护代理用于对象应该有不同 的访问权限的时候。

智能指引（Smart Reference）取代了简单的指针，它在访问对象时执行一些附加操作。 它的典型用途包括：

1.对指向实际对象的引用计数，这样当该对象没有引用时，可以自动释放它（也称为SmartPointers）。

2.当第一次引用一个持久对象时，将它装入内存。

3.在访问一个实际对象前，检查是否已经锁定了它，以确保其他对象不能改变它。

## 4.14职责链模式

**意图**

使多个对象都有机会处理请求，从而避免请求的发送者和接收者之间的耦合关系。将这些对象连成一条链，并沿着这条链传递该请求，直到有一个对象处理它为止。

**适用性**

有多个的对象可以处理一个请求，哪个对象处理该请求运行时刻自动确定。

你想在不明确指定接收者的情况下，向多个对象中的一个提交一个请求。

可处理一个请求的对象集合应被动态指定。

## 4.15命令模式

**意图**

将一个请求封装为一个对象，从而使你可用不同的请求对客户进行参数化；对请求排队或记录请求日志，以及支持可取消的操作

**适用性**

像上面讨论的MenuItem对象那样，抽象出待执行的动作以参数化某对象。你可用过程语言中的回调（callback）函数表达这种参数化机制。所谓回调函数是指函数先在某处注册，而它将在稍后某个需要的时候被调用。Command模式是回调机制的一个面向对象的替代品。

在不同的时刻指定、排列和执行请求。一个Command对象可以有一个与初始请求无关的生存期。如果一个请求的接收者可用一种与地址空间无关的方式表达，那么就可将负责该请求的命令对象传送给另一个不同的进程并在那儿实现该请求。

支持取消操作。Command的Execute操作可在实施操作前将状态存储起来，在取消操作时这个状态用来消除该操作的影响。Command接口必须添加一个Execute操作，该操作取消上一次Execute调用的效果。执行的命令被存储在一个历史列表中。可通过向后和向前遍历这一列表并分别调用Unexecute和Execute来实现重数不限的“取消”和“重做”。

支持修改日志，这样当系统崩溃时，这些修改可以被重做一遍。在Command接口中添加装载操作和存储操作，可以用来保持变动的一个一致的修改日志。从崩溃中恢复的过程包括从磁盘中重新读入记录下来的命令并用Execute操作重新执行它们。

用构建在原语操作上的高层操作构造一个系统。这样一种结构在支持事务（Transaction）的信息系统中很常见。一个事务封装了对数据的一组变动。Command模式提供了对事务进行建模的方法。Command有一个公共的接口，使得你可以用同一种方式调用所有的事务。同时使用该模式也易于添加新事务以扩展系统。

## 4.16解析器模式

**意图**

给定一个语言，定义它的文法的一种表示，并定义一个解释器，这个解释器使用该表示来解释语言中的句子。

**适用性**

当有一个语言需要解释执行, 并且你可将该语言中的句子表示为一个抽象语法树时，可使用解释器模式。而当存在以下情况时该模式效果最好：

1.该文法简单对于复杂的文法, 文法的类层次变得庞大而无法管理。此时语法分析程序生成器这样的工具是更好的选择。它们无需构建抽象语法树即可解释表达式, 这样可以节省空间而且还可能节省时间。

2.效率不是一个关键问题最高效的解释器通常不是通过直接解释语法分析树实现的, 而是首先将它们转换成另一种形式。例如，正则表达式通常被转换成状态机。但即使在这种情况下, 转换器仍可用解释器模式实现, 该模式仍是有用的。

## 4.17迭代器模式

**意图**

提供一种方法顺序访问一个聚合对象中各个元素, 而又不需暴露该对象的内部表示。

**适用性**

访问一个聚合对象的内容而无需暴露它的内部表示。

支持对聚合对象的多种遍历。

为遍历不同的聚合结构提供一个统一的接口（即, 支持多态迭代）。

## 4.18中介模式

**意图**

用一个中介对象来封装一系列的对象交互。中介者使各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。

**适用性**

一组对象以定义良好但是复杂的方式进行通信。产生的相互依赖关系结构混乱且难以理解。

一个对象引用其他很多对象并且直接与这些对象通信,导致难以复用该对象。

想定制一个分布在多个类中的行为，而又不想生成太多的子类。

## 4.19备忘录模式

**意图**

在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。这样以后就可将该对象恢复到保存的状态。

**适用性**

必须保存一个对象在某一个时刻的(部分)状态, 这样以后需要时它才能恢复到先前的状态。

如果一个用接口来让其它对象直接得到这些状态，将会暴露对象的实现细节并破坏对象的封装性。

## 4.20观察者模式

**意图**

定义对象间的一种一对多的依赖关系,当一个对象的状态发生改变时, 所有依赖于它的对象都得到通知并被自动更新。

**适用性**

当一个抽象模型有两个方面, 其中一个方面依赖于另一方面。将这二者封装在独立的对象中以使它们可以各自独立地改变和复用。

当对一个对象的改变需要同时改变其它对象, 而不知道具体有多少对象有待改变。

当一个对象必须通知其它对象，而它又不能假定其它对象是谁。换言之，你不希望这些对象是紧密耦合的。

## 4.21状态模式

**意图**

允许一个对象在其内部状态改变时改变它的行为。对象看起来似乎修改了它的类。

**适用性**

一个对象的行为取决于它的状态, 并且它必须在运行时刻根据状态改变它的行为。

一个操作中含有庞大的多分支的条件语句，且这些分支依赖于该对象的状态。这个状态通常用一个或多个枚举常量表示。通常, 有多个操作包含这一相同的条件结构。State模式将每一个条件分支放入一个独立的类中。这使得你可以根据对象自身的情况将对象的状态作为一个对象，这一对象可以不依赖于其他对象而独立变化。

## 4.22策略模式

**意图**

定义一系列的算法,把它们一个个封装起来, 并且使它们可相互替换。本模式使得算法可独立于使用它的客户而变化。

**适用性**

许多相关的类仅仅是行为有异。“策略”提供了一种用多个行为中的一个行为来配置一个类的方法。

需要使用一个算法的不同变体。例如，你可能会定义一些反映不同的空间/时间权衡的算法。当这些变体实现为一个算法的类层次时，可以使用策略模式。

算法使用客户不应该知道的数据。可使用策略模式以避免暴露复杂的、与算法相关的数据结构。

一个类定义了多种行为, 并且这些行为在这个类的操作中以多个条件语句的形式出现。将相关的条件分支移入它们各自的Strategy类中以代替这些条件语句。

## 4.23模版模式

**意图**

定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。Te m p l a t e M e t h o d 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。

**适用性**

一次性实现一个算法的不变的部分，并将可变的行为留给子类来实现。

各子类中公共的行为应被提取出来并集中到一个公共父类中以避免代码重复。这是Opdyke和Johnson所描述过的“重分解以一般化”的一个很好的例子。首先识别现有代码中的不同之处，并且将不同之处分离为新的操作。最后，用一个调用这些新的操作的模板方法来替换这些不同的代码。

控制子类扩展。模板方法只在特定点调用“hook”操作，这样就只允许在这些点进行扩展。

## 4.24访问者模式

**意图**

表示一个作用于某对象结构中的各元素的操作。它使你可以在不改变各元素的类的前提下定义作用于这些元素的新操作。

**适用性**

一个对象结构包含很多类对象，它们有不同的接口，而你想对这些对象实施一些依赖于其具体类的操作。

需要对一个对象结构中的对象进行很多不同的并且不相关的操作，而你想避免让这些操作“污染”这些对象的类。Visitor使得你可以将相关的操作集中起来定义在一个类中。当该对象结构被很多应用共享时，用Visitor模式让每个应用仅包含需要用到的操作。

定义对象结构的类很少改变，但经常需要在此结构上定义新的操作。改变对象结构类需要重定义对所有访问者的接口，这可能需要很大的代价。如果对象结构类经常改变，那么可能还是在这些类中定义这些操作较好。