UML

概念

UML的常用方式

- 系统需求
  - 在模型中捕获系统需求，帮助确认设计满足用户需要
- 系统结构和行为
  - 对系统的组成部分和它们之间的关系建模
  - 对系统中各个部分如何一起工作以满足系统的需求建模
- 系统部署
  - 对系统如何转移到现实世界中建模，描绘系统如何部署

建模语言的组成部分

- 伪代码、实际代码、图、文字描述
- 有助于描述系统的东西
- Notation（记号，表示法，符号）
- notation，表达模型的方式
- 对记号含义的描述成为语言的semantics
- 语义在语言的元模型中描述

为什么使用UML

1. 代码过于详尽
2. 非形式化语言会导致
    1. 罗嗦冗长
    2. 二义性
    3. 混乱

优点

- 形式化
- 简练
- 全面易理解
- 可伸缩，可测量
- 基于经验总结
- 有标准

模型和视图

> 建模工具所创建的一组图不是模式

模型：所有描述系统的元素，包括他们彼此的连接，构成了模型

图：一个图中不会一次性显示出所有模型的所有东西，图只是模型内容的一个视图

4+1视图模型

Logical view 逻辑视图

- 对系统组成部分的抽象描述
- 用于对系统由什么组成以及这些部分之间如何交互
- UML图：类图、对象图、状态机、交互图

Process view 过程视图

- 描述系统中的过程，帮助可视化系统中必须发生什么
- UML的活动图

Development view 开发视图

- 描述系统的部件如何被组织为模块和构件 （modules and components）
- 用于管理系统体系结构中的层次
- UML图：包图、构件图

Physical view 物理视图

- 描述系统的设计如何转变成为一组现实世界中的实体，即抽象的部件如何映射到最终部署的系统
- UML的部署图

Use case view 用例视图

- 描述从外部世界角度建模时看到的系统功能
- 描述想要让系统做些什么
- 上述四种视图都依赖用例视图作为指导
- 包含UML用例图、用例描述和用例概图

面向对象

Abstraction（抽象）

Inheritance（继承）

Polymorphism（多态）

Encapsulation（封装）

Message Sending（消息发送）

Associations（关联）

Multiplicity（重数）

Aggregation（聚集或者聚合）

Composition(组合或者组成）

类图

属性：

- 类的特性，一个类有零个或多个属性
- 属性会有取值范围，对象的每个属性一定会有一个值
- 类中，可以指定属性的类型，属性名和类型之间用冒号隔开
- 可以给属性一个缺省值

操作：

- 类的行为
- 操作名后必须有括号，括号里可以携带参数以及参数类型
- 有返回值的操作称之为函数

类图中多个属性和操作可以使用关键字进行分组

- 关键字使用书名号括起，独占一行

职责是对类必须做什么的描述——也就是说，它的属性和操作试图完成什么
约束即用花括号括起来的自由格式文本。括号内的文本指定类遵循的一条或多条规则

注释：使用折角矩形对类进行附加说明，与类之间用虚线连接

类之间关系

关联

类与类之间在逻辑上连接，称为关联

> 两个类、或者类与接口之间语义级别的一种强依赖关系
> 比如我和我的朋友，这种关系比依赖更强、不存在依赖关系的偶然性
> 关系不是临时性的，一般是长期性的，而且双方的关系一般是平等的
> 关联可以是单向、双向的
> 表现在代码层面，为被关联类B以类属性的形式出现在关联类A中，也可能是关联类A引用了一个类型为被关联类B的全局变量

- 每个类在关联中都会扮演角色
- 多个类可以连接到一个类
- 关联约束即在连接线上使用花括号声明约束
- 两个关联之间可以用约束{or}
- 关联也可以自成一个类
- 多重性：1对1，1对*，1对1..*，1对0.1，1对12..18，1对3，1对12,24
  - *表示0或多个
- 自关联：自身和自身产生关联，反射性关联

关联实现：

```
/**
  * Customer被Reservation关联:::Reservation实线指向Customer
  */
public class Reservation {
        private Customer customer ;
        public Reservation( Customer c ) {
            customer = c ;
        }
}
```

继承

- 没有父类的类叫做基类或根类
- 没有子类的类叫做叶子类
- 只有一个父类叫单继承
- 有多个父类叫多重继承

抽象类不提供对象，不能被实例化

依赖

> 一个类A使用到了另一个类B，而这种使用关系是具有偶然性的、临时性的、非常弱的，但是B类的变化会影响到A
> 比如某人要过河，需要借用一条船，此时人与船之间的关系就是依赖；
> 表现在代码层面，为类B作为参数被类A在某个method方法中使用

聚集聚合

有时，一个类由多个组件类组成。这是一种特殊类型的关系，称为聚合。组件及其构成的类处于部分-整体关联中。

> 部分和整体在生命周期上没有必然联系
>
> 聚合之间可以是整体对多部分的关系
> 部分和部分之间可以有{or}约束

组合

> 强调了整体与部分的生命周期是一致的，在组合关系中，整体与部分是不可分的，整体的生命周期结束也就意味着部分的生命周期结束
> 而聚合的整体和部分之间在生命周期上没有什么必然的联系

组成结构图

组合结构非常适合某些特定情景下的建模，组合结构图描述组合关系，也就是描述一个类的内部各组成部分之间的关系

- 内部结构
  - 显示一个类中包含的组成部分以及这些部分之间的关系
  - 由此可以显示上下文敏感的关系，或者在包含类中才有的关系
- 端口（Ports）
  - 显示如何在系统中通过端口使用一个类
- 协作（Collaborations）
  - 显示软件中的设计模式，或者对象如何协作达成一个目标

接口

操作的集合，这些操作只有声明，没有具体实现

供接口：

需接口：

用例图

概念

用例：用例是关于系统使用场景的集合
每一个场景都描述一系列事件

> UML规范中用例的定义用例是系统执行的一组操作的规范，它产生一个可观察的结果，通常对系统的一个或多个参与者或其他涉众有价值

用例的特征：

- 独立性
- 结果可观察、有意义
- 由参与者发起，反映了参与者的愿望
- 含有动宾短语

包含关系

多个用例的共同行为,从各个步骤序列中抽取公共步骤形成一个每个用例都要使用的附加用例

优点:

- 可以明确含义,一个用例的执行步骤中"包含"了另一个用例中的执行步骤
- 可以避免和通常的"使用"混淆,通过包含用例增进了用例的复用

扩展关系

对原有用例进行扩展,给已有的用例增加额外的步骤来建立新的用例

泛化关系

一般用于参与者之间的泛化，比如管理员是系统用户的一种

顺序图

对象用矩形框表示，其中是带下划线的对象名，消息用带箭头的实线表示，时间用垂直虚线

对象：每个对象向下伸展出垂直虚线，作为生命线。生命线上窄矩形条表示被激活，即对象正在进行操作，长度表示时间。

消息：一个对象到另一个对象的消息用跨越生命线的方式表示

> 调用消息：三角实心箭头实线
>
> 返回消息：箭头虚线
>
> 异步消息：箭头实线
> 把控制权交给接收者，不等待操作的完成

时间：顺序图是二维的，左右维数代表对象的布局，垂直的维数代表时间

消息队列中创建对象：使用`<<create>>`

交互事件：

活动图

活动图用于描述一个过程或操作的工作步骤

圆角矩形表示活动，比状态图标窄，更接近于椭圆

判定：菱形分开

初态、终止

并发：

信号：

泳道：将图分成多个平行的段，每个泳道顶部显示角色名，每个角色泳道负责自己的活动

状态图

对象改变了自己的状态来响应事件和时间的流逝

基本符号

起始状态：实心圆

状态：圆角矩形

活动：

入口动作entry：系统进入该状态时要发生的动作

出口动作exit：系统离开时要发生的动作

动作do：处于该状态时要发生的动作

事件：写在转移线上

保护条件：用中括号括起来，写在转移线旁边

终止状态：带外圈的实心圆

子状态

顺序子状态：在状态中顺序出现

并发子状态：并发状态之间用虚线隔开

设计模式

> 设计模式是一套被反复使用、多人知晓、经过分类编目的、代码设计的经验总结。对于代码的复用，经验的总结，增加代码的可靠性
>
> 基本要素：
> 模式名称：描述模式的问题、解决方法和效果
> 问题：描述在何时使用模式
> 解决方案：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
> 效果：描述了模式应用的效果及使用模式应权衡的问题
>
> 设计模式由四分帮建立--->GOF
> 1.针对接口编程
> 2.优先使用对象组合，而不是类继承
> 3.找到并封装变化点
>
> 设计模式的类别：
> 创建型：将对象的部分创建工作延迟到子类，而创建型对象模式则将它延迟到另一个对象中
> 结构型：使用继承机制组合类，而结构型对象模式则描述了对象的组装方式
> 行为型:使用继承描述算法和控制流，而行为型对象模式则描述一组对象怎样协作完成单个对象无法完成的任务

面向对象七大原则

设计模式对可维护性的支持

- 可扩展性：封装继承多态--->由开闭、里氏替换、依赖倒转、合成复用实现
- 灵活性：模块可复用，相对独立，与其他模块松耦合，该模块修改后，不会传递到其他模块--->由开闭、迪米特、接口隔离，单一实现
- 可插入性：复用后，新类代替旧类，容易实现--->由开闭、里氏替换、依赖倒转、合成复用实现

单一原则：类的职责单一，不将太多的职责放在一个类中

> 一个类承担的职责越多，它被复用的可能性越小，而且如果一个类承担的职责过多，就相当于将这些职责耦合在一起，当其中一个职责变化时，可能会影响其他职责的运作

开闭原则：软件实体对扩展开发，对修改封闭，即在不修改一个软件实体的基础上进行扩展

> 软件设计本身所追求的目标就是封装变化、降低耦合，而开放封闭原则正是对这一目标的最直接体现。
> 核心思想：对抽象编程，不对具体编程

里氏替换：子类完美继承父类所有的功能，所以一个可以接受基类对象的地方必然可以接受它的子类，即子类可以替代父类

> 派生类（子类）对象能够替换其基类（超类）对象被使用

依赖倒置：设计代码结构时，高层模块不应该依赖低层模块，二者都应该依赖其抽象，即针对抽象编程，不要针对具体

> 在进行业务设计时，与特定业务有关的依赖关系应该尽量依赖接口和抽象类，而不是依赖于具体类。具体类只负责相关业务的实现，修改具体类不影响与特定业务有关的依赖关系。降低客户与实现模块之间的耦合
>
> 依赖指的是调用关系，调用者依赖于被调用者
> 倒置指的是上层不依赖于下层，转而下层去依赖上层

接口分离：使用多个专门的接口来取代统一的接口

> 客户端不应该依赖于它不需要的接口，即将一个庞大冗杂的接口分割成更细小的接口，该接口只提供客户端需要的行为
>
> 进行接口拆分时，必须满足单一原则，且接口内方法越少越好

合成复用：系统中尽量多使用组合和聚合关系，少用甚至不要使用继承关系

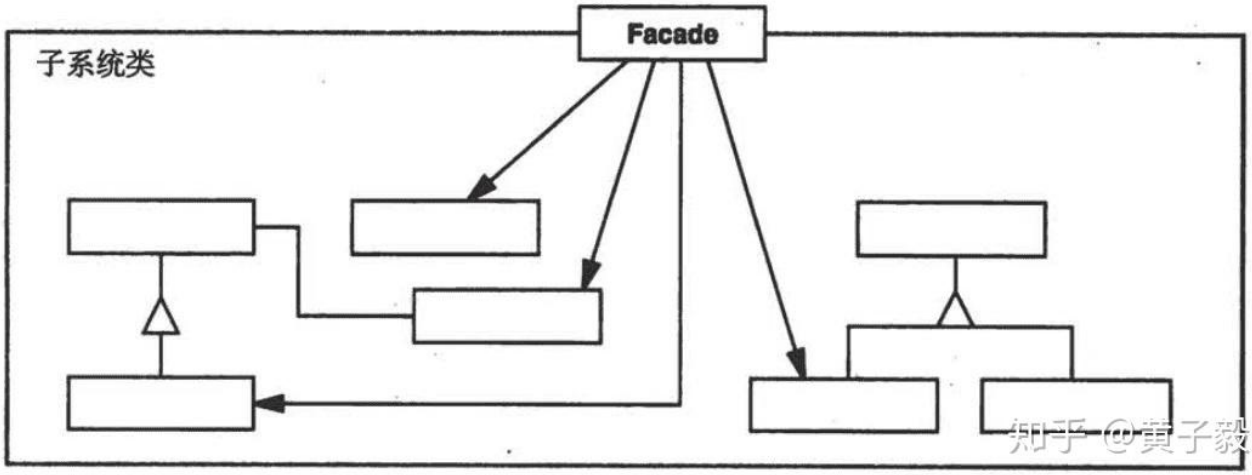
> 新的对象里通过关联关系来使用已有的对象；新对象通过委派调用已有对象的方法达到复用已有功能的目的
>
> 继承复用：实现简单，易于扩展。破坏系统封装性，灵活性低下，称之为白箱复用。需要严格满足里氏替换原则
>
> 组合聚合复用：耦合度低，可以动态进行，称之为黑箱复用

迪米特法则：最少了解，一个软件实体对其他实体的引用越少越好，或者说如果两个类不必彼此直接通信，即两个类之间应该通过引入第三个方进行间接交互

> 不要和陌生人说话
> 只与你的直接朋友通信
> 每个单位 对其他单位只有最少的知识，局限于与本单位密切相关的单位
>
> 朋友类别：
> 当前对象本身
> 以参数形式传入到当前对象方法中的对象
> 当前对象的成员对象
> 当前对象所创建的对象
>
> 狭义迪米特：可以降低类之间的耦合，但会增加方法的数量，造成系统的不同模块之间通信效率降低
> 两个类之间不必彼此直接通信，这两个类就不应当发送直接的相互作用，需要通信就要通过第三方
>
> 广义迪米特：对对象之间的信息流量、流向以及信息的影响的控制，主要对信息隐藏的控制。
> 信息隐藏：可以使各个子系统之间脱耦，允许独立开发、优化、使用和修改
>
> 主要用途：控制信息的过载
> 1.类的划分尽量创建松耦合的类
> 2.类的结构设计上每个类都应当尽量降低其成员变量和成员函数
> 3.类的设计上，只要有可能就应该设计成不变类
> 4.类的引用上，一个对象对其它对象的引用应当降低到最低

外观模式

> 为了子系统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，使得子系统更加容易使用



> 参与者和协作者：为欲客户提供一个简化接口，使系统更容易时使用
>
> 效果：Facade模式简化了对所需子系统的使用过程。但是，由于Facade并不完整，因此客户可能无法使用某些功能

Facade不仅可以为方法调用创建更简单的接口，还可以减少客户必须处理的对象数量

> 假设有一个Client对象，这个对象必须处理Database , Model , Element类的对象。Client必须首先通过Database对象打开数据库，以得到一个对Model对象的引用，然后再向Model对象请求一个Element对象，最后向Element对象查询所需的信息。而如果我能创建一个DatabaseFacade类，让Client对象向它发出请求，那么上面的过程可能就会变得简单一些

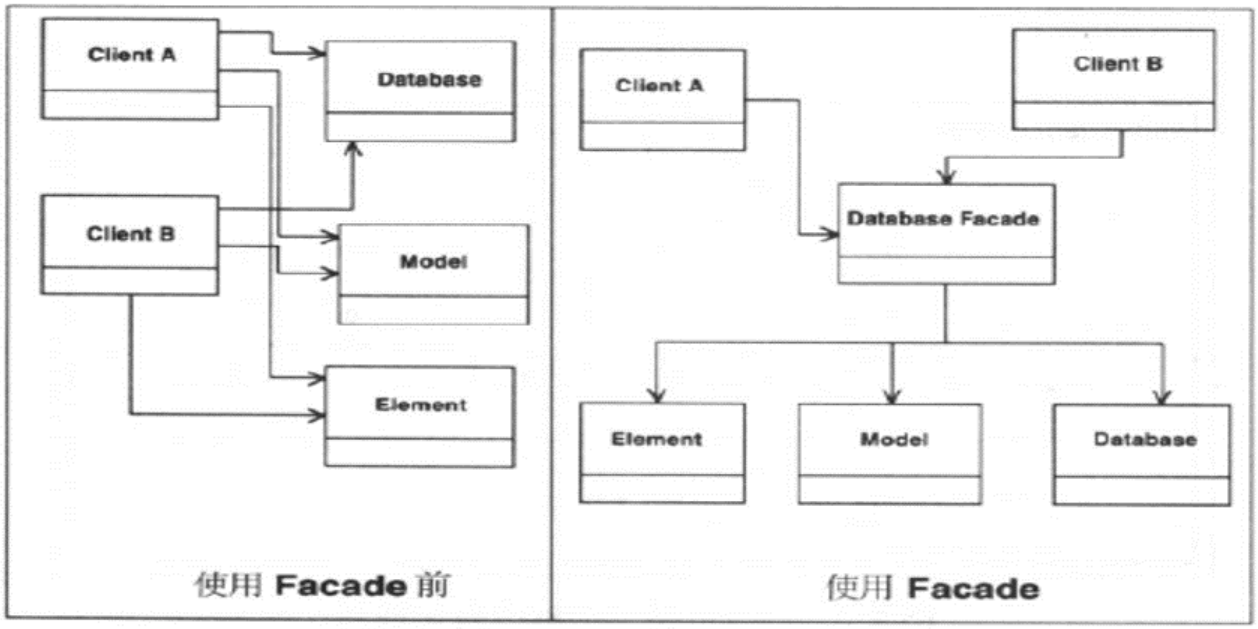


图 6-4　使用 Facade 模式减少客户需要处理的对象数量

> Facade模式也可以用于隐藏或包装原有的系统。 Facade可以把原有系统作为自己的私有成员。在这种情况下，原有系统与Facade类联系在一起，但使用Facade类的客户无法看到原有的系统

- 跟踪对系统的使用：强迫所有客户通过Facade使用原有系统，然后就可以很容易地监控对系统的使用了。
- 改变系统：把原有系统设计成Facade类的私有成员，那么只需做最少的工作就能切换一个新的系统。当然，我仍然需要做一些重要的工作，但至少我只需要改变一个地方(Facade类)的代码。

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

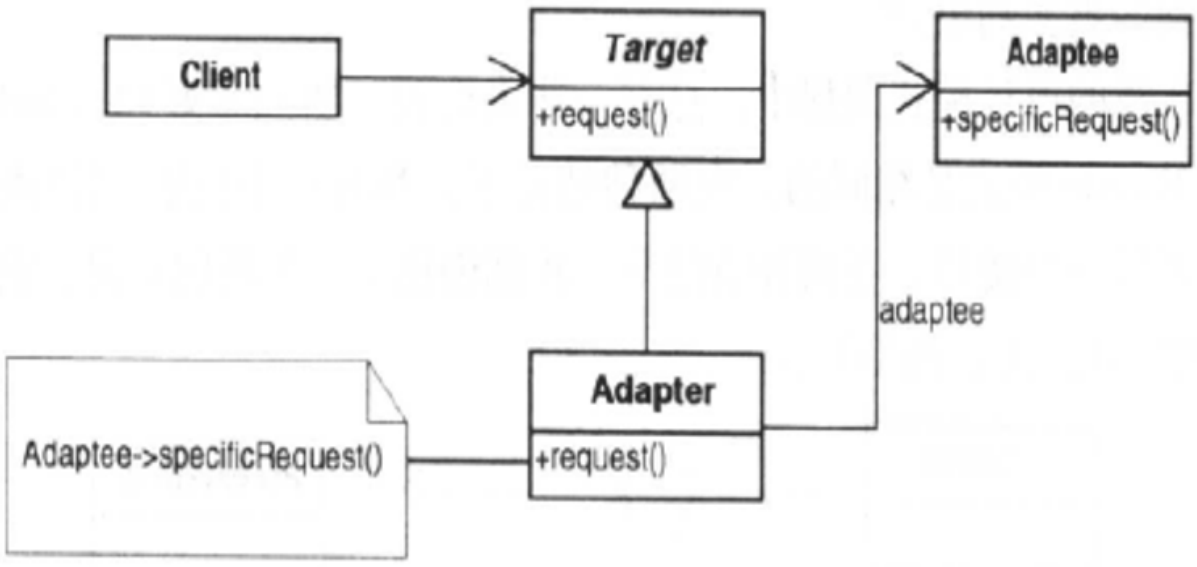


图7-6 Adapter模式的标准简化视图

> 参与者与协作者：Adapter改变了Adaptee的接口，使Adaptee与Adapter的基类Target匹配。这样Client就可以使用Adaptee了，好像它是Target类型

> 效果：Adapter模式使原有对象能够适应新的类结构，不受其接口的限制

> 被适配的对象可能不具备我想要的所有东西，在这种情况下，仍然可以使用Adapter模式，但它就不完全合适了
> 1.现存类已实现的那些功能可以被适配
> 2.现存对象没有实现的那些功能可以在适配对象中实现

> 对象Adapter模式：它依赖于一个对象(适配对象)包含另一个 对象(被适配对象)

> 类Adapter模式：使用多重继承实现Adapter模式

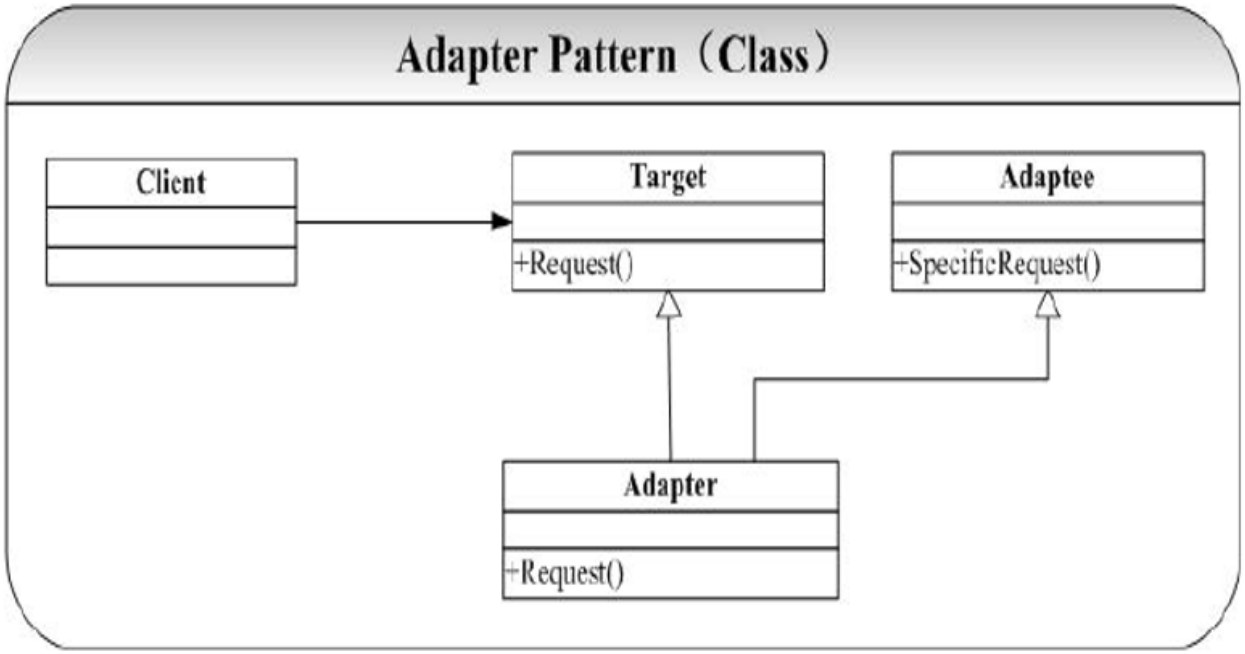


图 2-1：Adapter Pattern（类模式）结构图

Adapter 对比 Facade

	Facade模式	adapter模式
是否由现存的类	是	是
是否我们必须针对接口进行设计	否	是
一个对象是否需要多态行为	否	可能
是否需要一个更简单的接口	是	否

Facade模式简化接口，而Adapter模式将接口转换成另一个现有的接口

知识扩展

对象的传统看法与新的看法

> 原有观点：对象是具有方法的数据
>
> 新观点：对象是拥有责任的实体。这些责任定义了对象的行为
> 这样可以分两步构建软件:
> 建立一个初步的设计，不必操心所有的相关细节
> 实现该设计

封装的传统看法与新的看法

> 原有观点：封装是数据的隐藏

> 新观点：封装应该被想成“任何形式的隐藏”。换句话说，它可以隐藏数据。但它还可以隐藏实现细节、派生类、设计细节、实例化规则。
> 用这种方式看待封装的优点是:它给了一种更好的切分(分解) 程序的方法。封装层成为将要设计的接口。

继承的传统方式与新的方式

> 传统方式：类的复用
>
> 新的方式：对象分类的方法，将类按照相同行为分类

共性与可变性分析

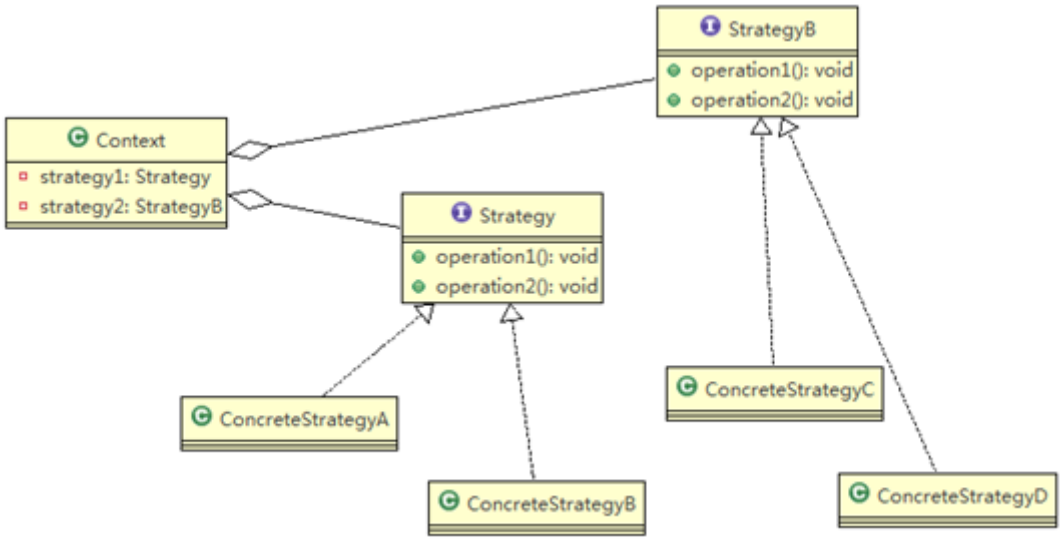
策略模式

> 考虑变更的设计
>
> 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。本模式使算法可独立于使用它的客户而变化

建立原则：

- 对象拥有责任
- 责任的不同的特定实现通过使用多态来表现
- 需要将几个不同的实现按照概念上相同的算法管理

若只有不发生变化的算法，则不需要Strategy模式



> 参与者协作：
> strategy规定如何使用不同的算法
> concreteStrategis实现这些不同的算法
> Context通过类型为Strategy的引用使用特定的CnncreteStrategy.
> Strategy与Context交互实现所选的算法(有时候Strategy必须向 context做查询)
>
> 效果：
> strategy模式定义了一系列的算法。
> switch语句或条件语句得到了避免。
> 必须以相同的方式调用所有的算法(它们必须拥有相同的接口)

> ConcreteStrategies与Context之间的交互可能需要在Context中加人getState之类的方法

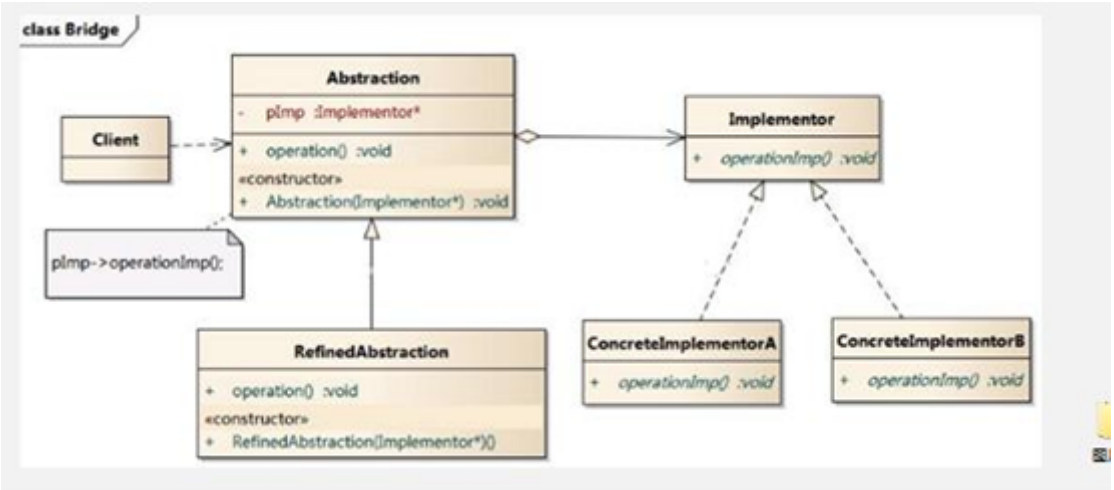
桥接模式

> 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
> 实现部分其实是指“抽象类的对象和用来实现抽象类的派生类的对象”(而不是抽象类的派生类，这些派生类被称为“具体类”)

在创建设计以处理变化的过程中，应该遵循两条基本策略:

- 发现并封装变化点
- 优先使用对象组合，而不是类继承

解决问题：一个抽象类的派生类必须使用多个实现，但不能出现类数量爆炸性增长



> 参与者与协作者：
> Abstraction为要实现的对象定义接口；
> Implementor类为具体的实现类定义接口；
> Abstraction的派生类使用Implementor的派生类，却无需知道自已具体使用哪一个ConcreteImplementor
>
> 效果：
> 实现与使用实现的对象解耦，提供了可扩展性，客户对象无需操心实现问题。

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

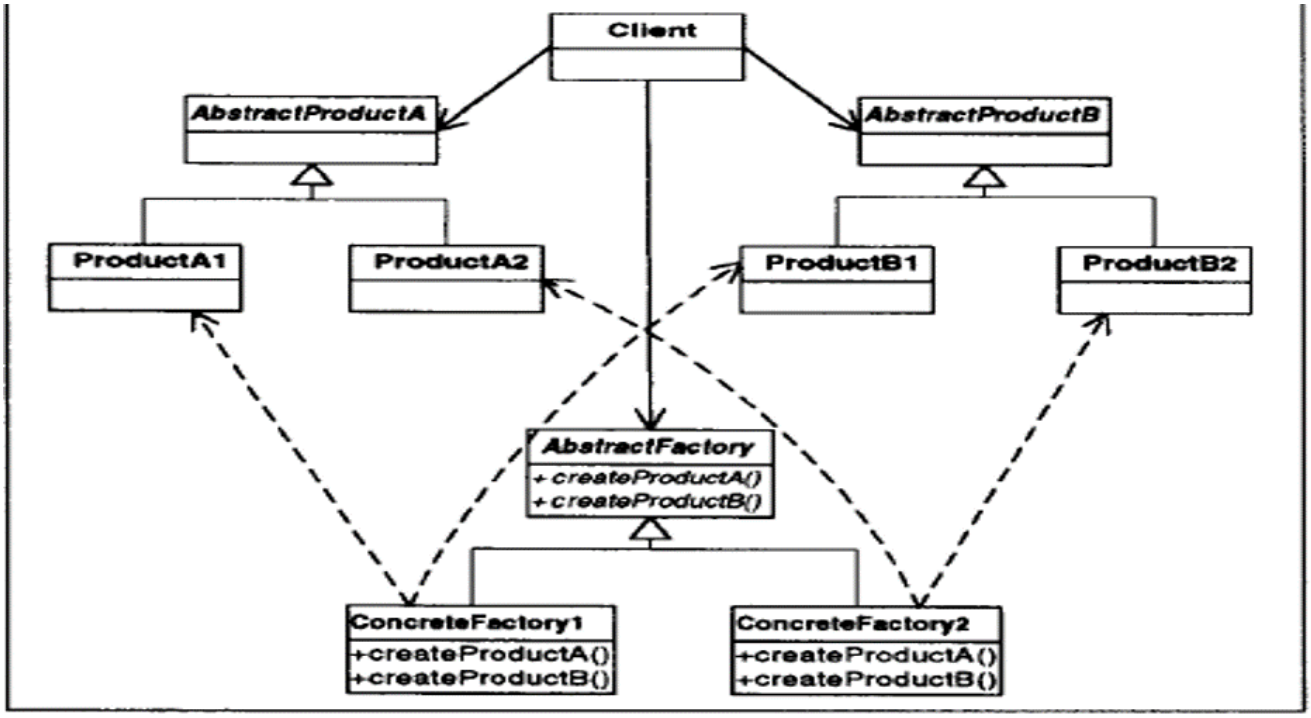


图 10-8　Abstract Factory 模式的标准简化视图

> 参与者与协作者：
> AbstractFactory为如何创建对象组的每个成员定义接口。
> 一般每个组都由独立的ConcreteFactory进行创建。
>
> 效果：
> 这个模式将“使用哪些对象’的规则与“如何使用这些对象”的逻辑分离开来

用模式思考

优秀的设计师遵循的原则：

- 背景优先：从问题出发考虑优先使用那个模式，然后已选择的模式就又成为背景的一部分
- 每次一个：每次只运用一个模式

装饰模式

> 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
>
> 可以创建以decorator对象—负责新的功能的对象—开始的一条对象“链”，并结束于最初的对象

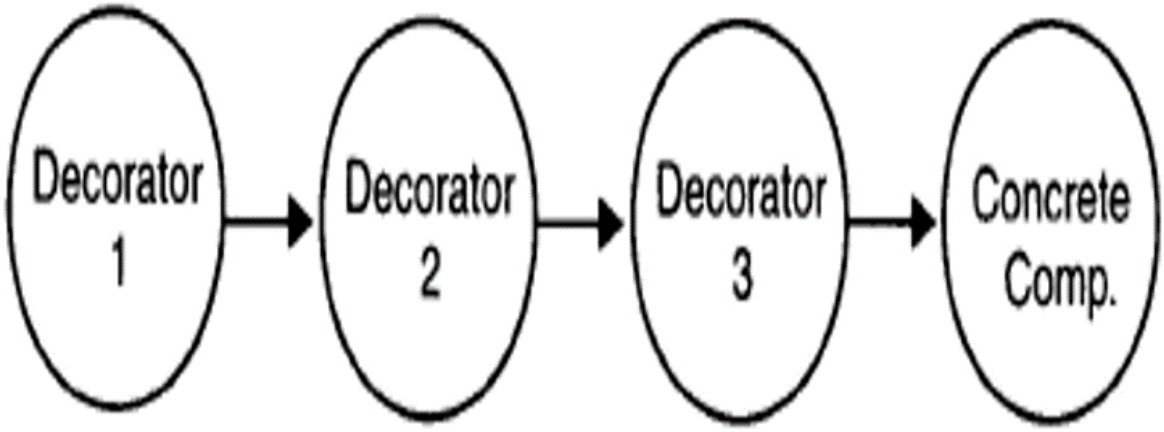
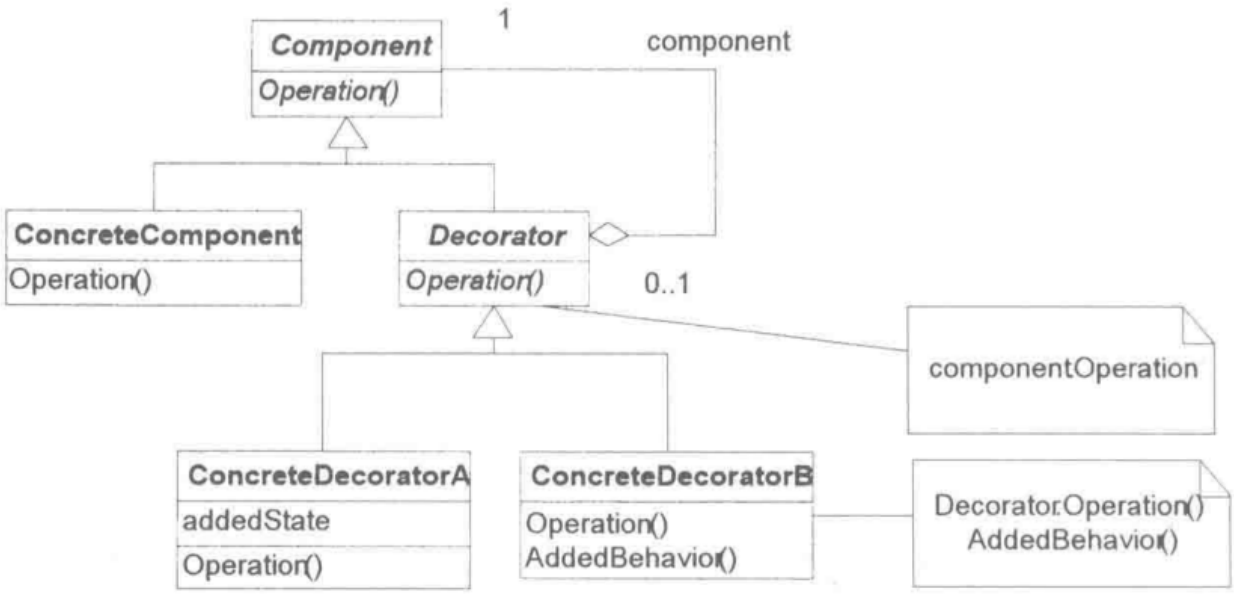


图 15-3 Decorator 链



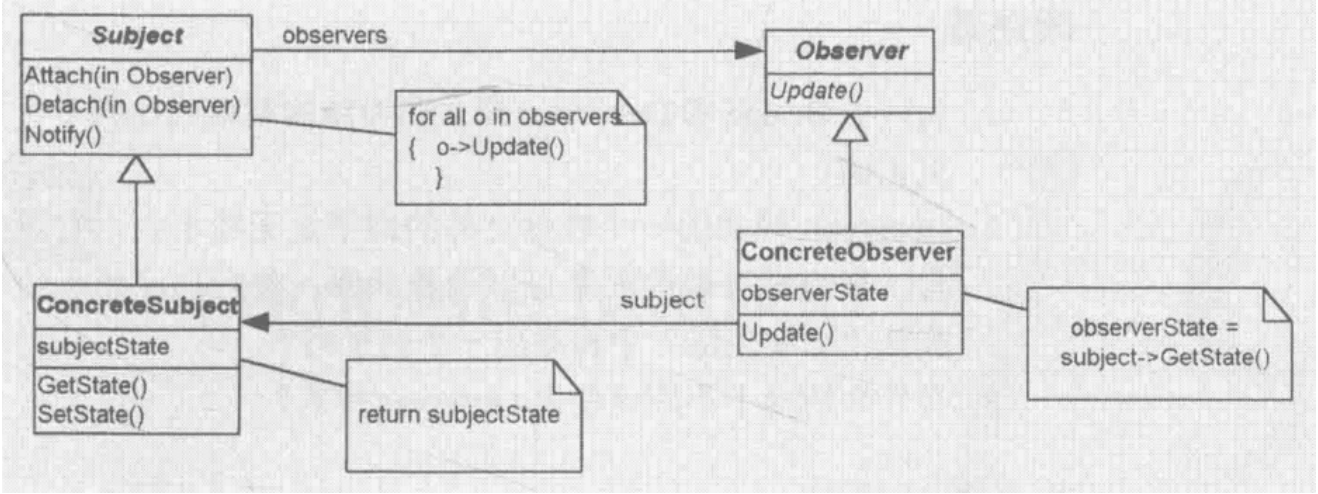
参与者与协作者：
ConcreteComponent让Decorator对象为自己添加功能
有时候用ConcreteComponent的派生类提供核心功能，在这种情况下ConcreteComponent类就不再是具体的，而是抽象的Component类定义了所有这些类所使用的接口

效果：
所添加的功能放在小对象中。好处是可以在ConcreteComponent对象的功能之前或之后动态添加功能
注意：虽然装饰对象可以在被装饰对象之前或之后添加功能，但对象链总是终于ConcreteComponent对象

观察者模式

基于发布订阅，通过依赖进行：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

有时候，一个将成为观察者的类可能已经存在了。在这种情况下，我可能不希望修改它。如果是这样，我可以使用Adapter模式简单地对它进行适配



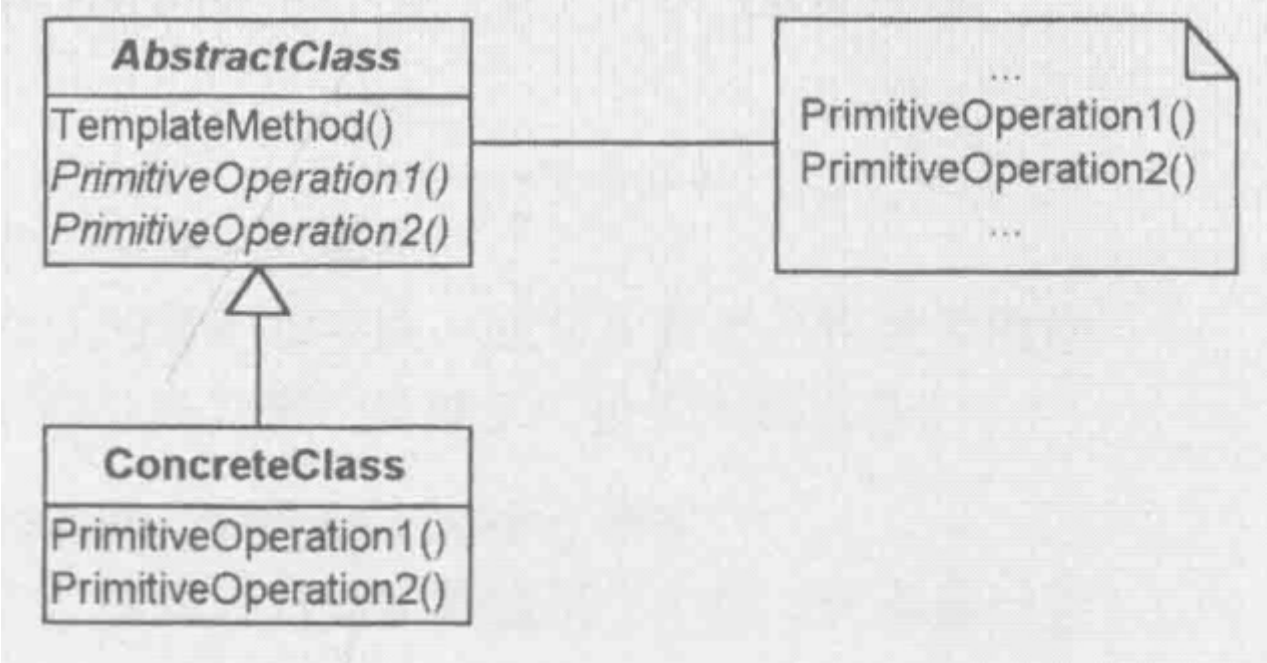
> 参与者与协作者：
> Subject知道自己的Observer，因为Observer要向它注册。
> Subject必须在所监视的事件发生时通知Observer。
> Observer负责向subject注册，以及在得到通知时从Subject处获取信息
>
> 效果：
> 如果某些Observer只对事件的一个子集感兴趣，那么Subject可能会告诉它们不需要知道的事件。
> 如果subject通知Observer, Observer还返回请求更多信息，则可能需要额外的通信。

模板模式

> 帮助从不同的步骤中抽象出一个通用过程的模式
>
> 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
>
> 简单来说就是把不同操作中提取出相同部分进行算法封装，减少代码量、提高可读性、可维护性

避免模板重复方式：

- 首先，重构方法，提取要修改的代码
- 其次，创建一个基类包含不变的方法
- 编写MySecondClass。请注意现在添加新功能除了工厂类（也只是可能)之外，将不会影响任何其他代码



> 参与者与协作者：
> Template Method模式由一个抽象类组成，这个抽象类定义了需要覆盖的基本TemplateMethod方法。
> 每个从这个抽象类派生的具体类将为此模板实现新方法
>
> 效果：
> 模板提供了一个很好的代码复用平台。它还有助于确保所需步骤的实现
> 它将每个Concrete类的覆盖步骤绑定起来，因此只有在这些变化总是并且只能一起发生时，才应该使用Template Method模式

单例模式

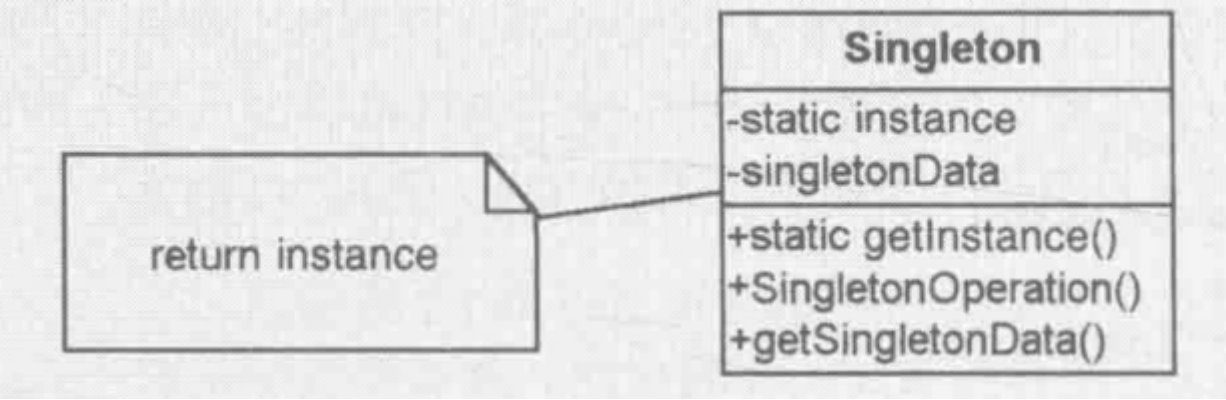
> 保证一个类仅有一个实例，并提供一个访问它的全局访问点

工作方式：

- 用一个特定的方法实例化需要的对象
- 调用这个方法时，检查对象是否已经实例化
  - 如果已经实例化，方法仅仅返回对该对象的一个引用
  - 如果尚未实例化，这个方法将对象实例化并返回新实例的引用
- 构造函数是保护或私有的

用途：

- 需要反复使用同样的对象，由于性能的原因，不希望反复地实例化这些对象，然后又将它们销毁
- 不在开始就实例化，最好的方法是在需要时进行实例化，但只进行一次实例化
- 不希望创建一个独立的对象来记住“已经实例化了哪些对象”。而是让这些对象自己负责处理自己的单次实例化



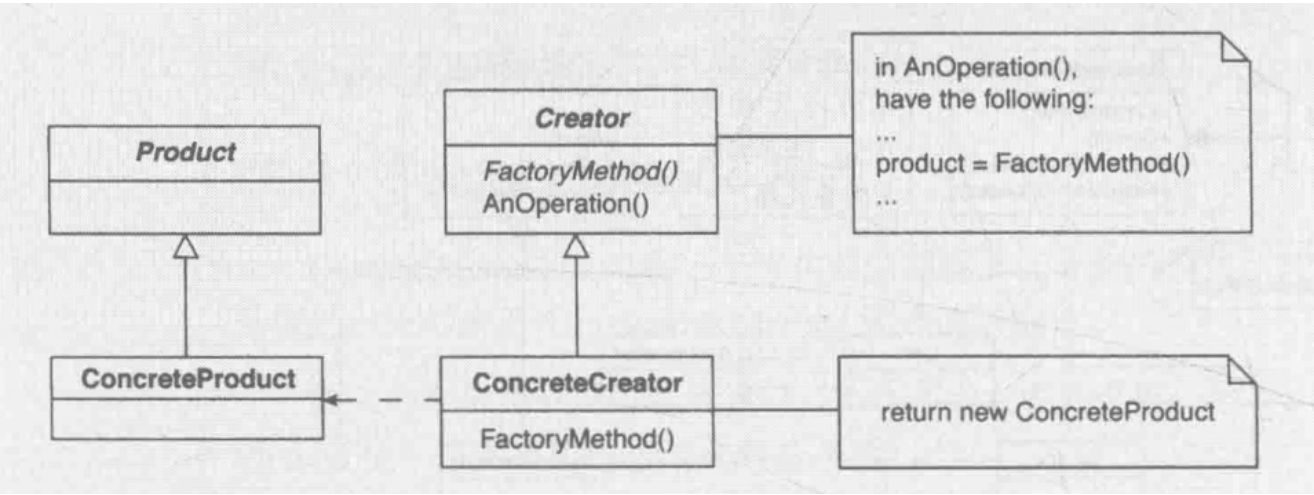
问题：几个不同的客户对象需要引用同一对象，而且希望确保这种类型的对象数目不超过一个

参与者与协作者：
C1ient对象只能通过getInstance方法创建Singleton实例

效果：
C1ient对象无需操心是否已存在Singleton实例

工厂方法模式

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



问题：
一个类需要实例化另一个类的派生类，但不知道是哪一个
FactoryMhod允许派生类进行决策

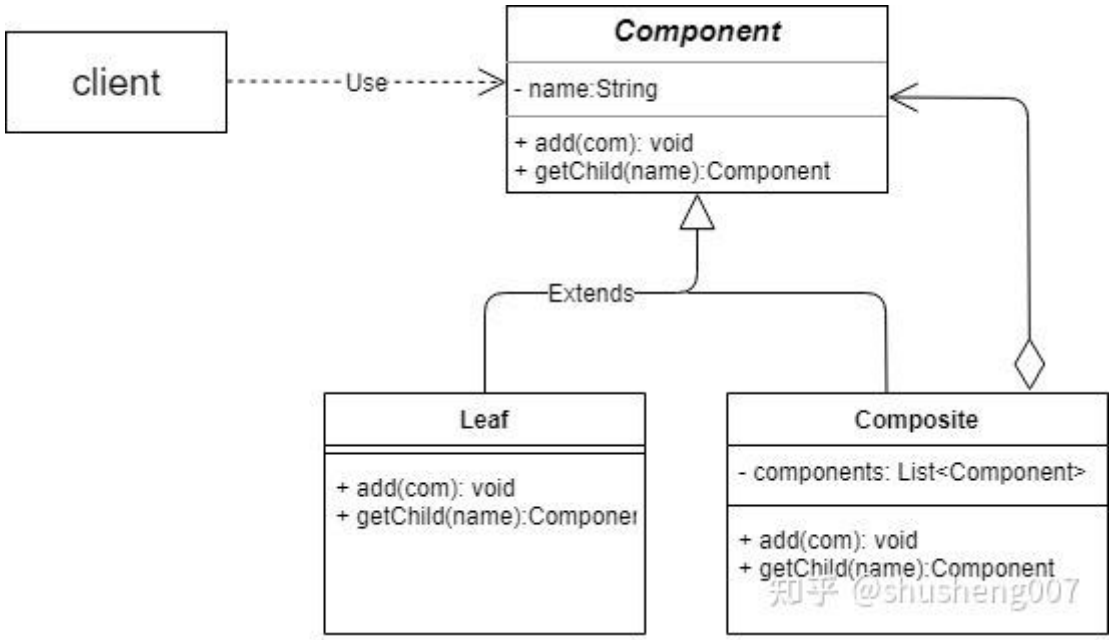
参与者与协作者：
product是工厂方法所创建的对象类型的接口。
Creator是定义工厂方法的接口

效果：
客户将需要派生Creator，以创建一个特定的ConcreteProduct对象。

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。
组合模式使得用户对单个对象和组合对象的使用具有一致性

重点在组合逻辑关系，即部分整体



参与者：
Component为组合中的对象声明接口，在适当的情况下，实现所有类共有接口的默认行为。
声明一个接口用于访问和管理Component子部件
Leaf 在组合中表示叶子结点对象，叶子结点没有子结点，定义图元对象的行为
Composite 定义有子部件的那些部件的行为，用来存储子部件，在Component接口中实现与子部件有关操作，如增加(add)和删除(remove)等

协作者：
用户使用Component类接口与组合结构中的对象进行交互。
如果接收者是一个叶节点，则直接处理请求，如果接收者是composite，它通常将请求发送给它的子部件。

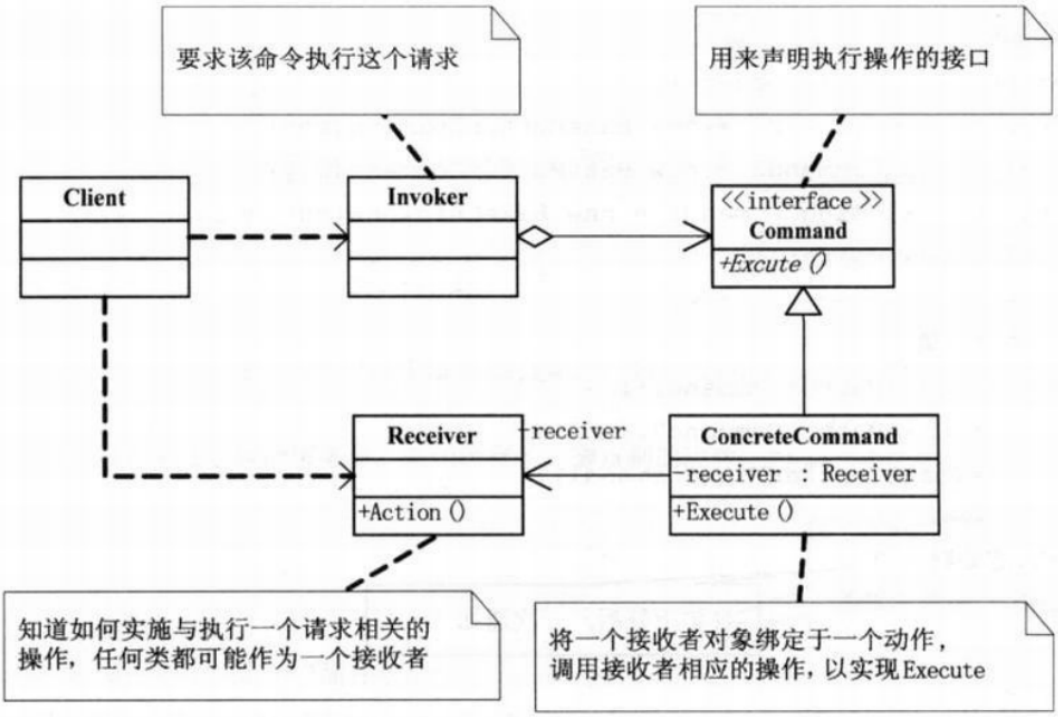
效果：
定义了包含基本对象和组合对象的类层次结构
减化了客户代码
更容易增加新类型的组件
设计更一般化

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，以及支持可撤销的操作

命令模式（Command），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，以及支持可撤销的操作。[DP]

命令模式（Command）结构图



参与者：

Command：定义命令的接口，声明执行的方法

ConcreteCommand：命令接口实现对象，是“虚”的实现;通常会持有接收者，并调用接收者的功能来完成命令要执行的操作

Receiver：接收者，真正执行命令的对象。任何类都可能成为一个接收者，只要它能够实现命令要求实现的相应功能

Invoker：要求命令对象执行请求，通常会持有命令对象，可以持有很多的命令对象。使用命令对象的入口

Client：创建具体的命令对象，并且设置命令对象的接收者。可以称之为装配者，因为真正使用命令的客户端是从Invoker来触发执行

效果：

将调用操作的对象与知道如何实现该操作的对象解耦

Command对象可像其他的对象一样被操纵和扩展

可将多个命令装配成一个复合命令

增加新的command很容易，因为无需改变已有的类