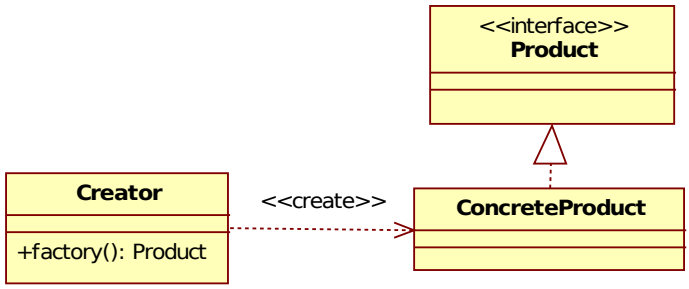
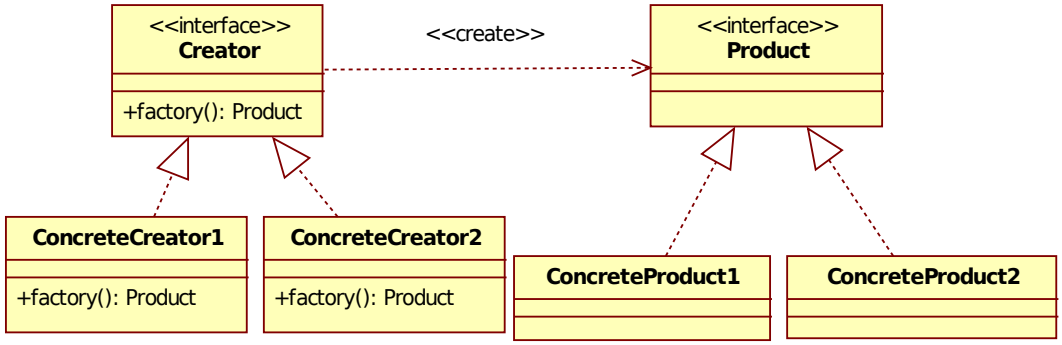
Factory 模式

1．简单工厂模式，又称静态工厂模式

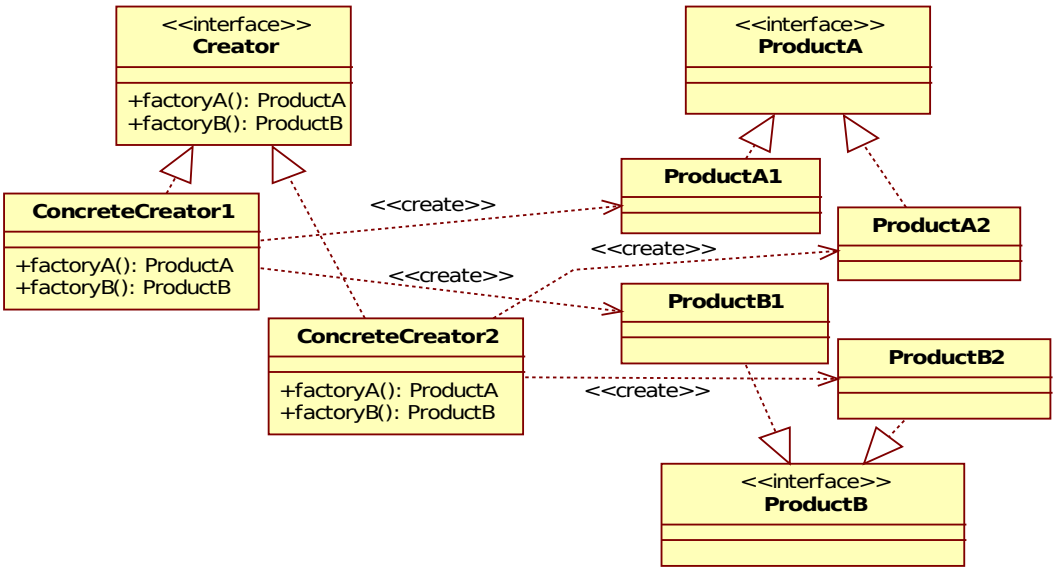


2．工厂方法模式



3．抽象工厂模式

抽象工厂模式与工厂方法模式的最大区别在于，工厂方法模式针对的是一个产品等级结构；而抽象工厂模式则需要面对多个产品等级结构。



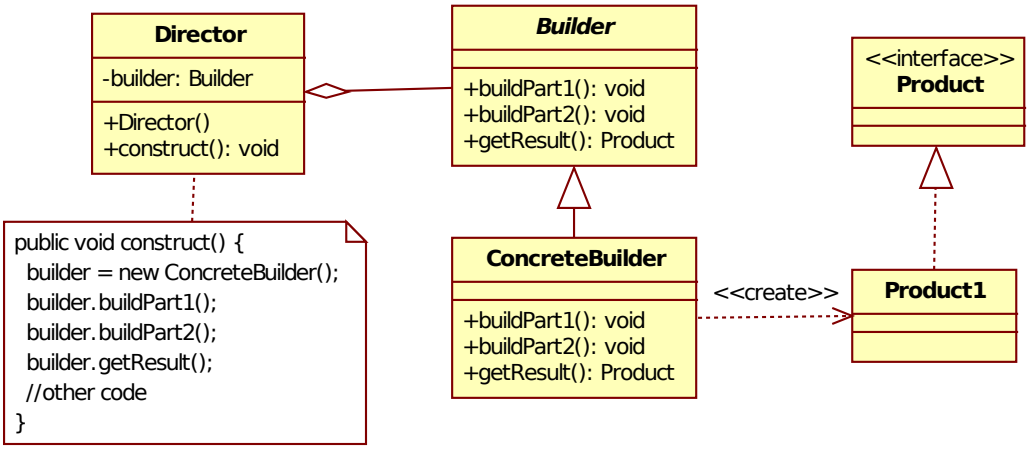
Singleton 模式

要点:

- 类只能有一个实例
- 必须自行创建这个实例
- 必须自行向外界提供这个实例

Builder 模式

Builder 模式利用一个 Director 对象和 ConcreteBuilder 对象一个一个地建造出所有的零件，从而建造出完整的 Product。Builder 模式将产品的结构和产品的零件建造过程对客户端隐藏起来，把对建造过程进行指挥的责任和具体的建造者零件的责任分割开来，达到责任划分和封装的目的。

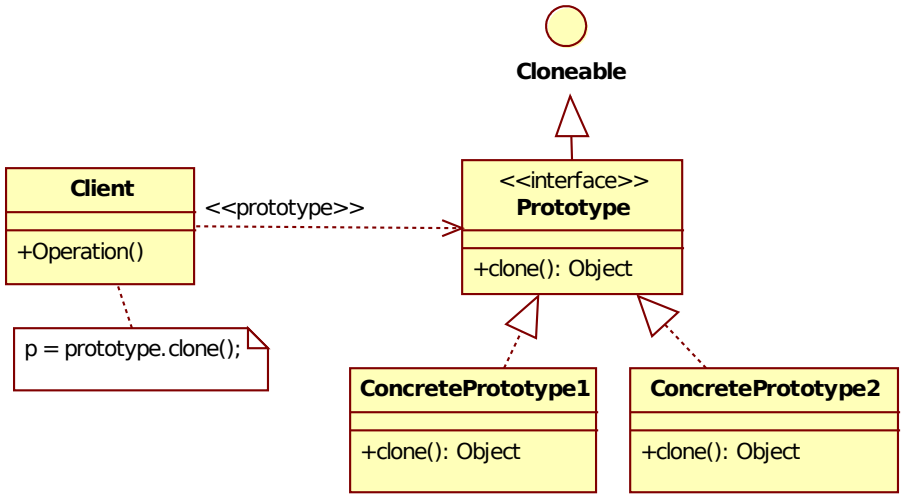


使用 Builder 模式的场合：

- 需要生成的产品对象有复杂的内部结构。每一个内部成分本身可以是对象，也可以紧紧是产品对象的一个组成部分。
- 需要生成的产品对象的属性相互以来。Builder 模式可以强制实行一种分步骤进行的建造过程，因此，如果产品对象的一个属性必须在另一个属性被赋值之后才可以被赋值，使用建造模式便是一个很好的设计思想。
- 在对象创建过程中会使用到系统中的其他一些对象，这些对象在产品对象的创建过程中不易得到。

Prototype 模式

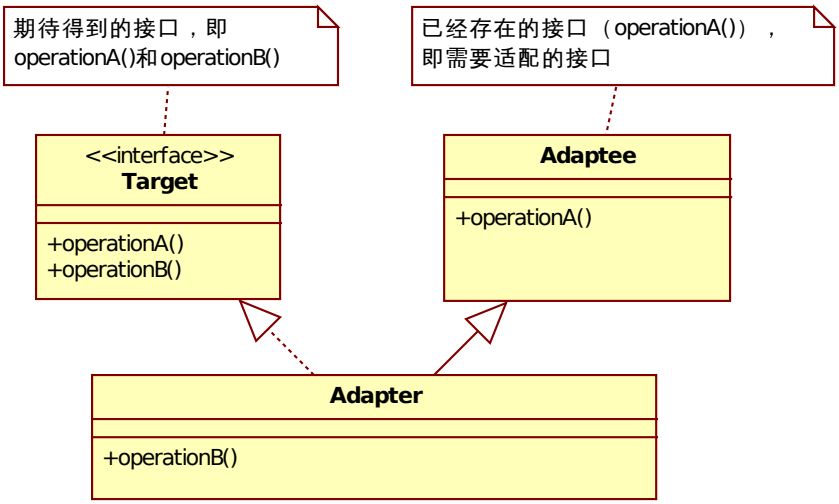
通过给出一个原型对象来指明所要创建的对象的类型，然后用赋值这个原型对象的办法创建出更多同类型的对象。



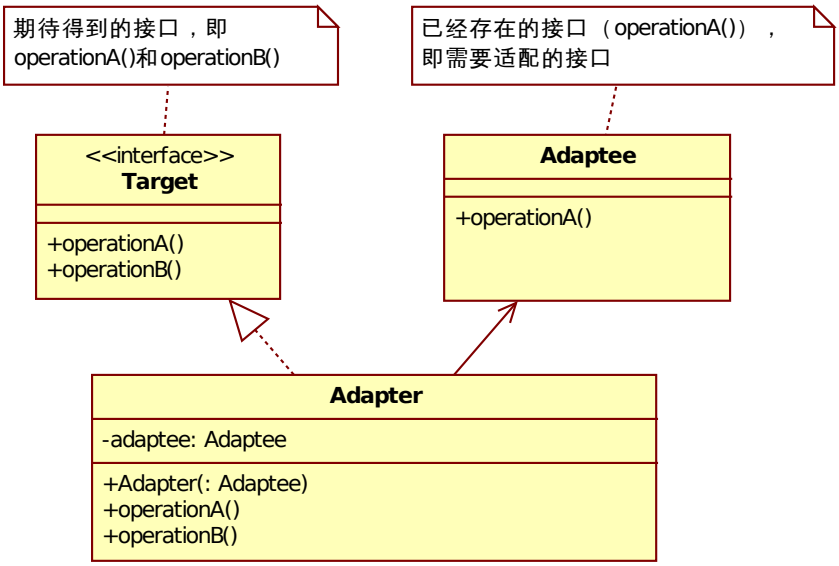
Adapter 模式

把一个类的接口变换成客户端所期待的另一种接口，从而使原本因接口不匹配而无法在一起工作的两个类能够在一起工作，也就是说把接口不同而功能相同或相近的多个接口加以转换。

1. 类的 Adapter 模式的结构



2. 对象的 Adapter 模式的结构



注意两种结构的区别：主要就是 Adaptee 和 Adapter 的关系，一个为继承关系，一个为依

赖关系。

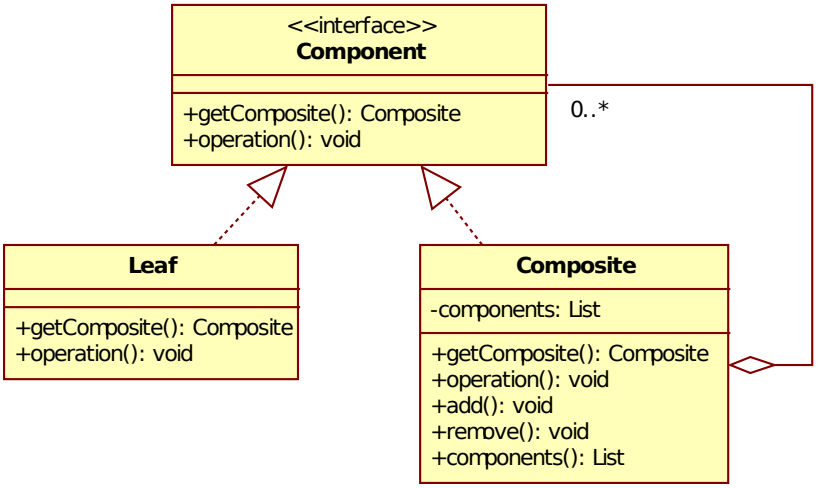
使用 Adapter 模式的场合：

- 系统需要使用现在的类，而此类的接口不符合系统的需要。
- 想要建立一个可以重复使用的类，用语与一些彼此之间没有太大关联的一些类，包括一些可能在将来引进的类一起工作。这些源类不一定有很复杂的接口。
- （对对象的 Adapter 模式而言）在设计里，需要改变多个已有的子类的接口，如果使用类的 Adapter 模式，就要针对每一个子类做一个 Adapter 类，而这不太实际。

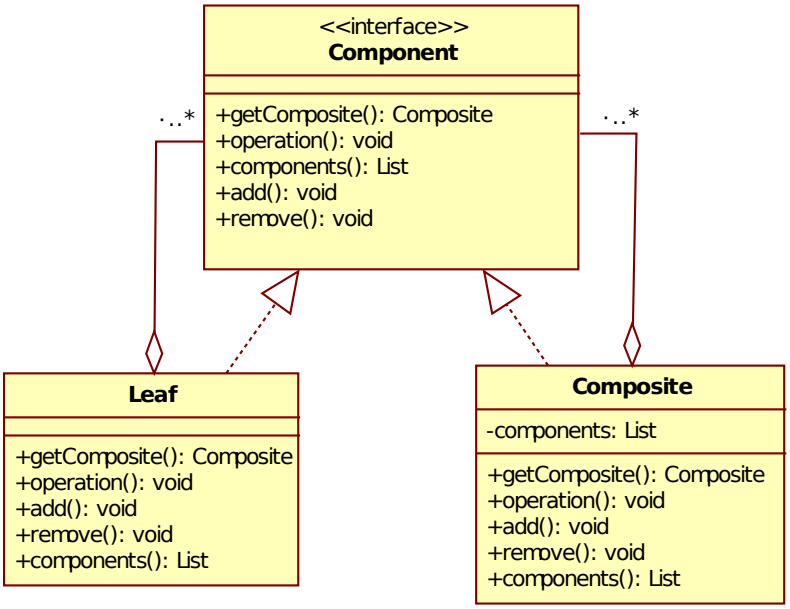
Composite 模式

把部分和整体的关系用树结构表示出来。Composite 模式使得客户端把一个个单独的成分对象和由他们符合而成的合成对象同等看待。

1. 安全式的 Composite 模式

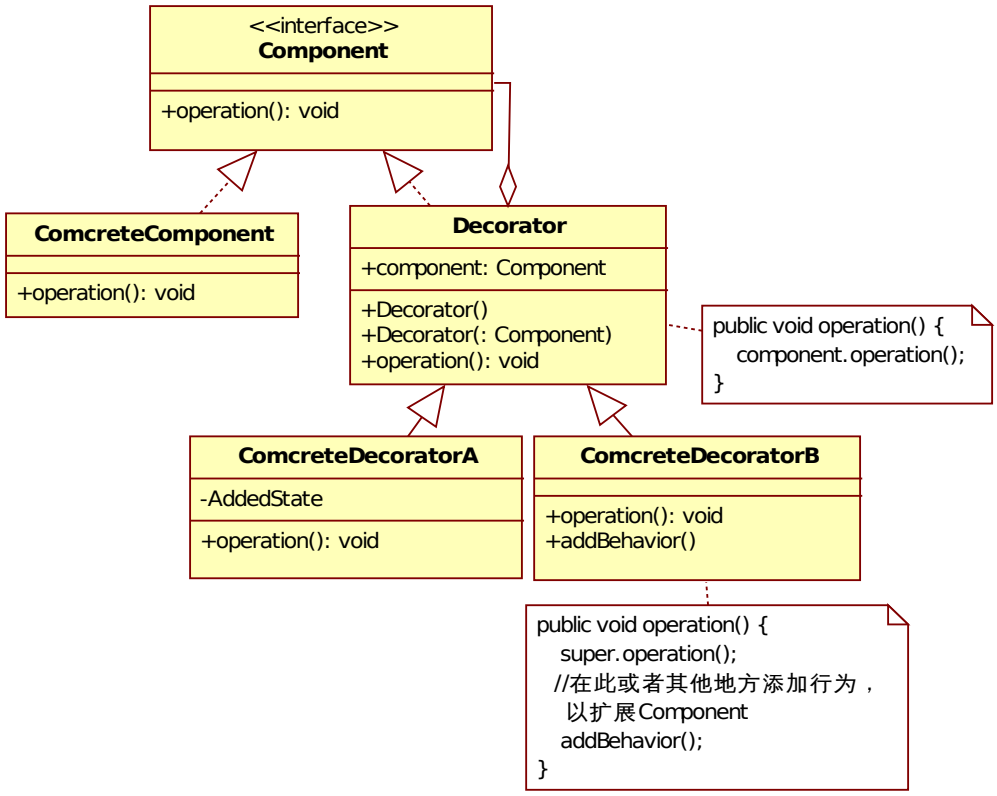


2. 透明式的 Composite 模式



Decorator 模式

此模式又称 Wrapper 模式。是以对客户端透明的方式扩展对象的功能，是继承关系的一个替代方案。

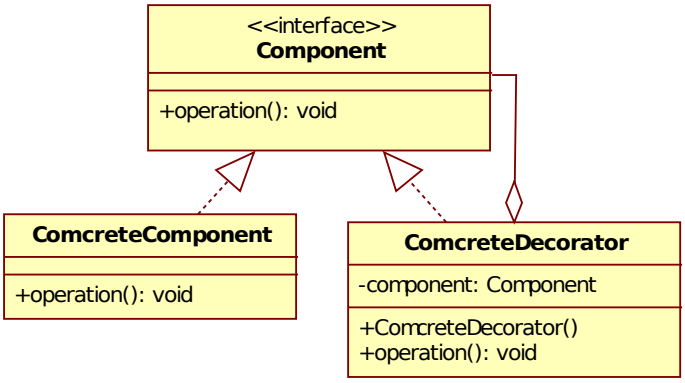


在以下情况下 Decorator 模式:

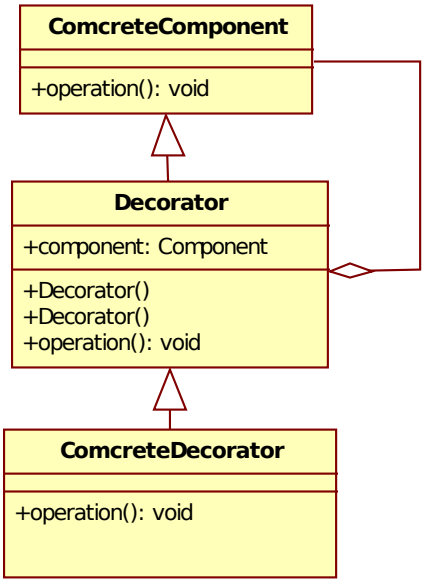
- 需要扩展一个类的功能，或给一个类增加附加责任。
- 需要动态地给一个对象增加功能，这些功能可以再动态的撤销。
- 需要增加由一些基本功能的排列组合个数的功能，从而使继承关系变得不现实。

Decorator 模式的简化

1. 没有抽象的 Decorator

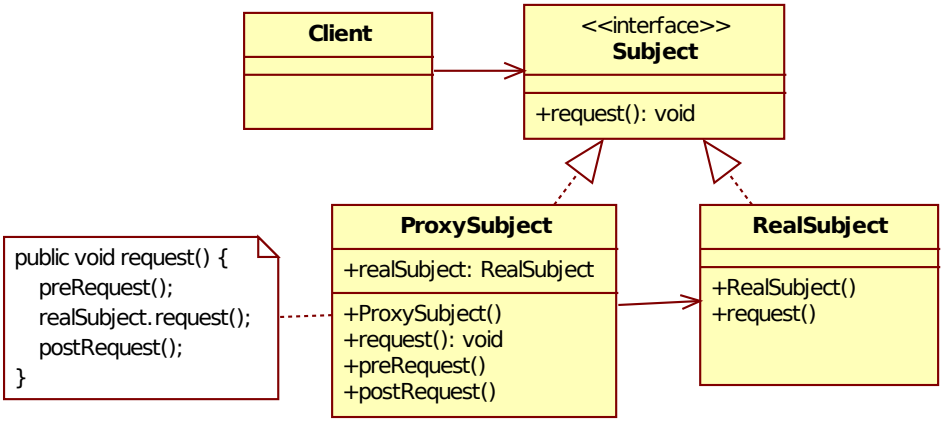


2. 没有抽象接口 Component



Proxy 模式

Proxy 模式是给某一个对象提供一个代理对象，并由代理对象控制对原对象的引用。



Flyweight 模式

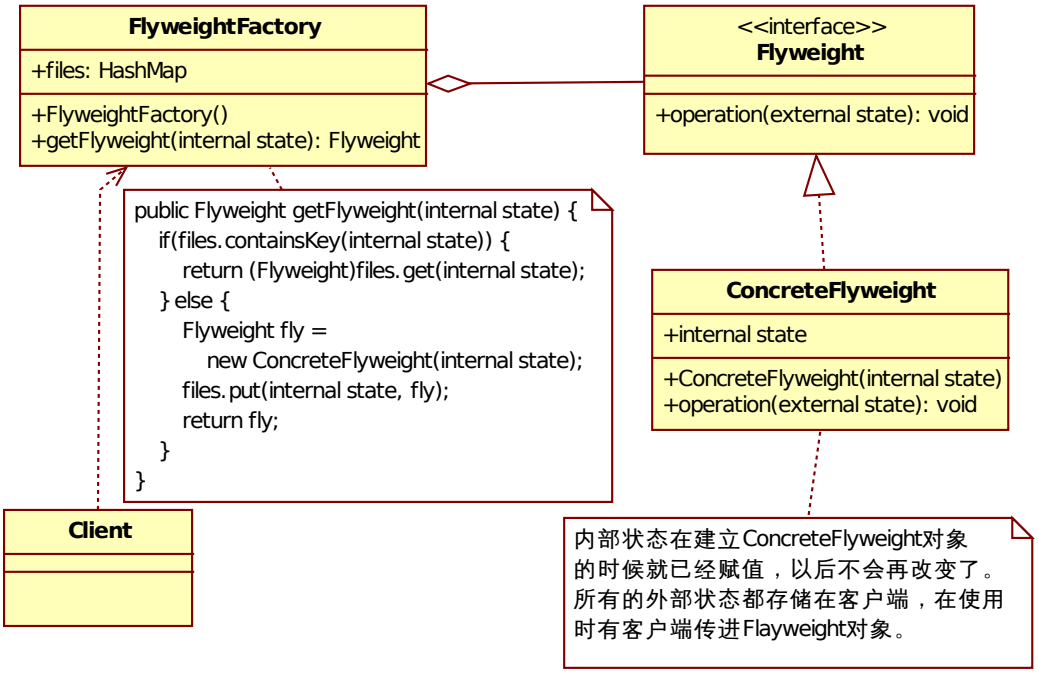
Flyweight 模式以共享的方式高效地支持大量的细粒度对象。Flyweight 对象能做到共享的关键是区分内部状态（Internal state）和外部状态（External state）。

内部状态是存储在 Flyweight 对象内部的，并且是不会随环境改变而有所不同的。因此，一个 Flyweight 可以具有内部状态并可以共享。

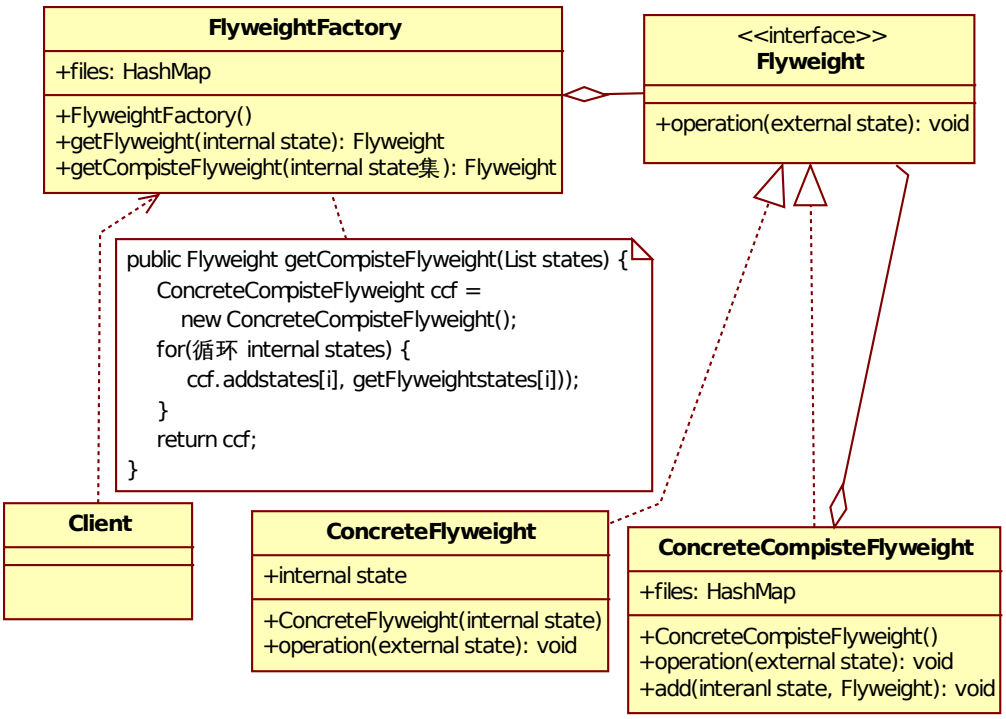
外部状态是随环境改变而改变的、不可以共享的状态。Flyweight 对象的外部状态必须有客户端保存，并在 Flyweight 对象被创建之后，在需要使用的时候再传入到 Flyweight 对象内部。

外部状态不可以影响 Flyweight 对象的内部状态。它们是相互独立的。

1. 单纯 Flyweight 模式的结构



2. 复合 Flyweight 模式的结构



在以下所有的条件满足时，可以考虑使用 Flyweight 模式:

- 一个系统有大量的对象。
- 这些对象耗费大量的内存。
- 这些对象的状态中的大部分都可以外部化。
- 这些对象可以按照内部状态分成很多的组，当把外部对象从对象中删除时，每一个组都可以仅用一个对象代替。
- 软件系统不依赖于这些对象的身份，换言之，这些对象可以是不可分辨的。

Facade 模式

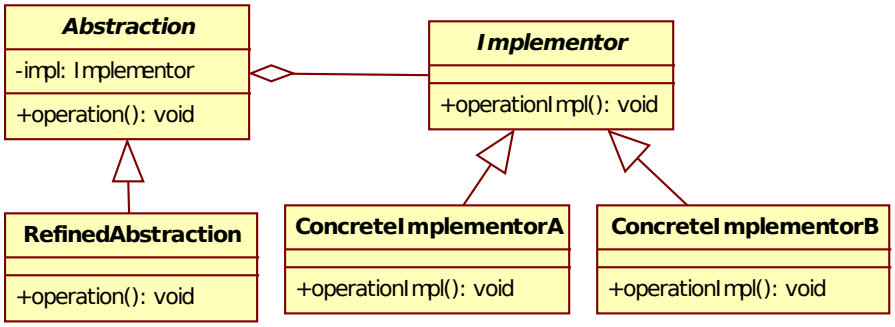
外部与一个子系统的通信必须通过一个统一的门面（Facade）对象进行，这就是 Façade 模式。

在以下情况下可以使用 Facade 模式：

- 为一个复杂子系统提供一个简单接口
  子系统往往因为不断演化而变得越来越复杂，使用 Facade 模式可以使得子系统更具有可复用性。
- 子系统的独立性
  一般而言，子系统和其他子系统之间 ·客户端和实现化层之间存在着很大的依赖性。引入 Facade 模式将一个子系统与它的客户端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。
- 层次化结构
  在构建一个层次化的系统时，可以使用 Facade 模式定义系统中每一层的入口。如果层与层之间是相互依赖的，则可以限定它们仅通过 Facade 模式进行通信，从而简化层与层之间的依赖关系。

Bridge 模式

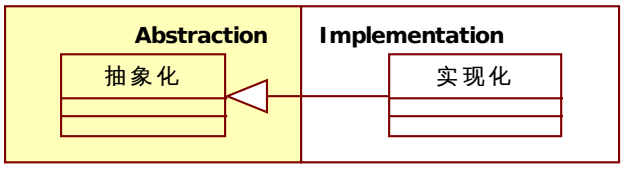
Bridge 模式的用意是“将抽象化与实现化解耦，使得二者可以独立地变化。”



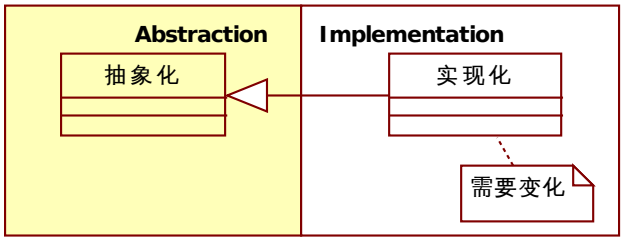
Bridge 模式比较难于理解，下面对其给出详细的解释。

“找到系统的可变因素，将之封装起来”，通常就叫做“对变化的封装”。对变化的封装实际上是达到“开-闭”原则的途径，与组合/聚合服用原则相辅相成。

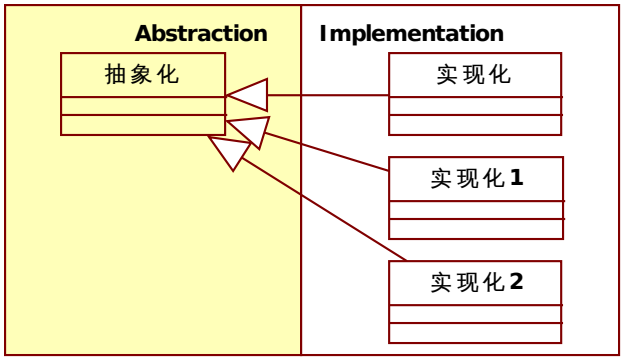
抽象化与实现化的最简单实现，也就是“开-闭”原则在类层次上的最简单实现，如下图所示。



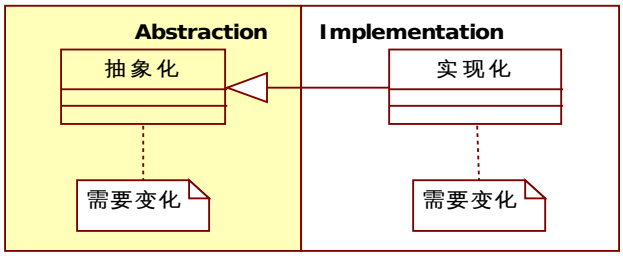
一般来说，一个继承结构中的第一层是抽象角色，封装了抽象的商业逻辑，这是系统中的不变的部分。第二层是实现角色，封装了设计中会变化的因素。这个实现允许实现化角色有多态性变化，如下图所示。



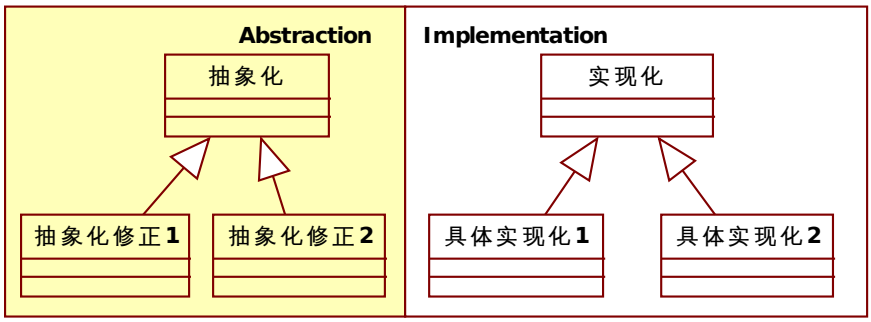
换言之，客户端可以持有抽象化类型的对象，而不在意对象的真实类型是“实现化”、“实现化 1”还是“实现化 2”，如下图所示。



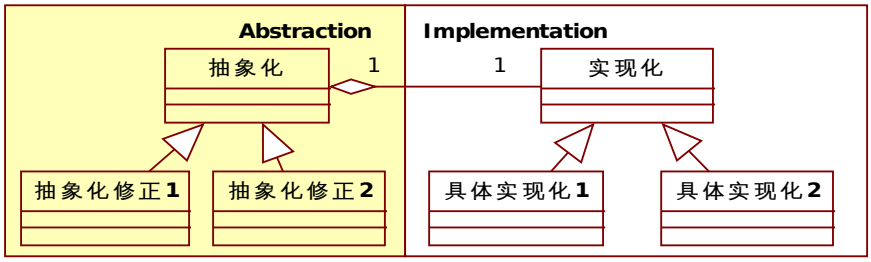
显然，每一个继承关系都封装了一个变化因素，而一个继承关系不应当同时处理两个变化因素。换言之，这种简单实现不能够处理抽象化与实现化都面临变化的情况，如下图所示。



上图中的两个变化因素应当是彼此独立的，可以在不影响另一者的情况下独立演化。比如，下面的两个等级结构分别封装了自己的变化因素，由于每一个变化因素都是可以通过静态关系表达的，因此分别使用继承关系实现，如下图所示。



那么在抽象化与实现化之间的变化怎么办呢？正确的设计方案应当是使用两个独立的等级结构封装两个独立的变化因素，并在他们之间使用聚合关系，以此达到功能复用的目的。



根据上面的分析，在以下情况下应当使用 Bridge 模式：

- 如果一个系统需要在构件的抽象化角色和具体化角色之间增加更多的灵活性，避免在两个层次之间建立静态的联系。
- 设计要求实现化角色的任何改变不应当影响客户端，或者说实现化角色的改变对客户端是完全透明的。
- 一个构件有多于一个的抽象化角色和实现化角色，系统需要他们之间进行动态耦合。
- 虽然在系统中使用继承是没有问题的，但是由于抽象化角色和具体化角色需要独立变化，设计要求需要独立管理这两者。

Immutable 模式

一个对象的状态在对象被创建之后就不再变化，这就是 Immutable 模式。

弱不变模式

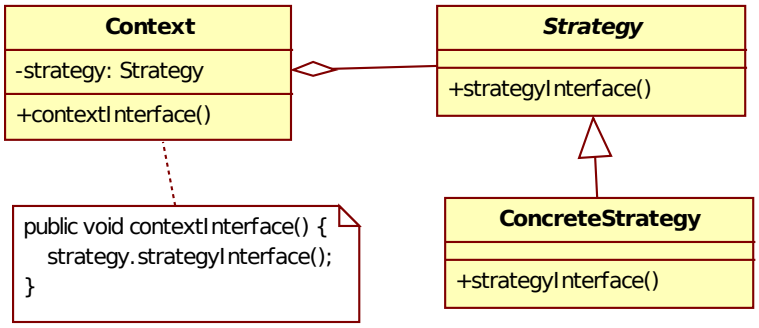
一个类的实例的状态是不可变化的，但是这个类的子类的实例具有可能会变化的状态。

强不变模式

一个类的实例的状态是不可变化的，同时这个类的子类的实例也具有不可变化的状态。

Strategy 模式

Strategy 模式就是针对一组算法，将每一个算法封装到具有共同接口的独立的类中，从而使得它们可以相互替换。



Strategy 模式的实现有以下这些需要注意的地方。

- 经常见到的是，所有的具体策略类都有一些公有的行为。这时候，就应当把这些公有的行为放到共有的抽象策略角色 Strategy 类中。当然这时候抽象策略角色必须要用 java 抽象类实现，而不能使用 java 接口。
- Strategy 模式在每一个时刻都只能使用一个策略对象，但是有的时候一个应用程序同时与几个策略对象相联系。换言之，在应用程序启动时，所有的策略对象就已经被创立出来，而应用程序可以在几个策略对象之间调换。

在下面的情况下应当考虑使用 Strategy 模式：

- 如果在一个系统里面有许多类，它们之间的区别仅在于它们的行为，那么使用 Strategy 模式可以动态地让一个对象在许多行为中选择一种行为。
- 一个系统需要动态地在几种算法中选择一个。
- 一个系统的算法使用的数据不可以让客户端知道。
- 如果一个对象有很多的行为，如果不用恰当的模式，这些行为就只好使用多重的条件选择语句来实现。此时，使用 Strategy 模式，把这些行为转移到相应的具体策略类里面，就可以避免使用难以维护的多重条件选择语句，并体现 OO 设计的概念。

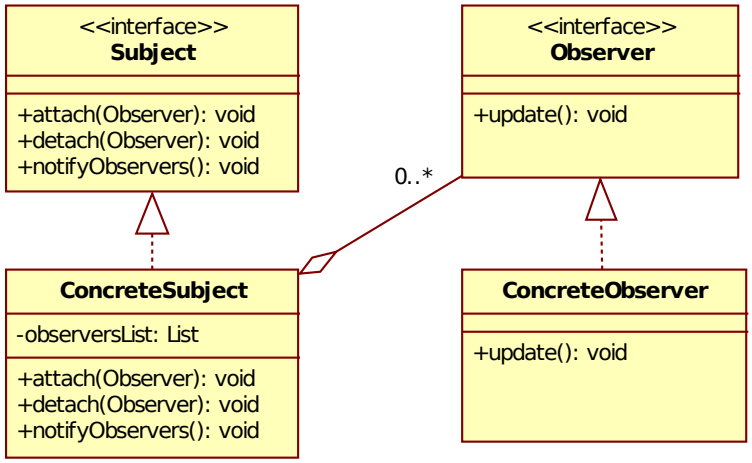
Template Method 模式

准备一个抽象类，将部分逻辑以具体方法以及具体构造子的形式实现，然后声明一些抽象方法来迫使子类实现剩余的逻辑。不同的子类可以以不同的方式实现这些抽象方法，从而对剩余的逻辑有不同的实现。这就是 Template Method 模式的用意。

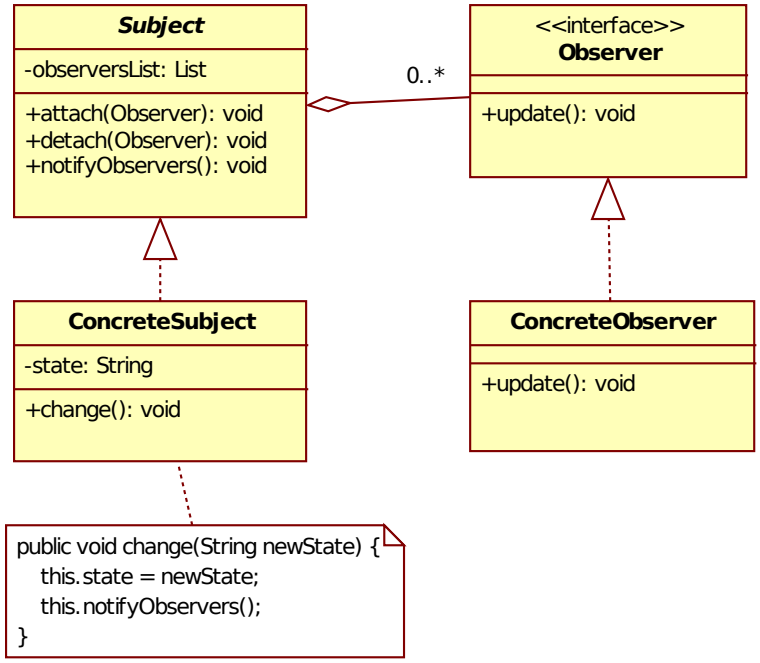
Observer 模式

Observer 模式定义了一种一对多的依赖关系，让多个观察着对象同时监听某一个主题对象。这个主题对象在状态上发生变化时，会通知所有观察者对象，使它们能够自动更新自己。

1．简单的实现方案



2．另一种实现方案

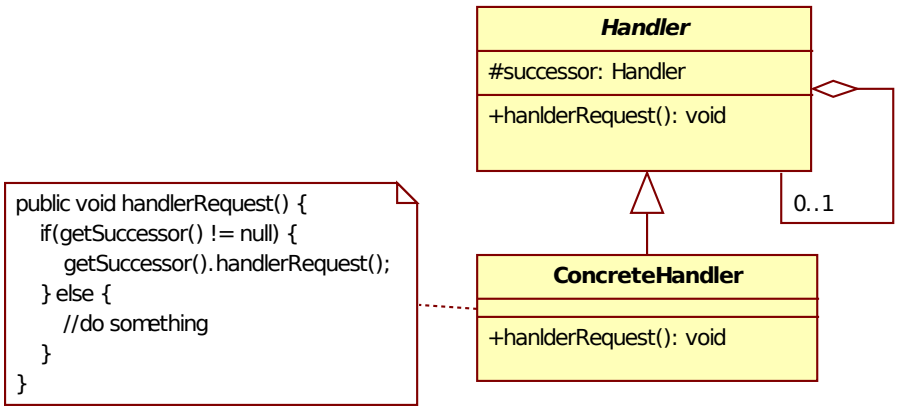


Iterator 模式

Iterator 模式可以顺序地访问一个聚集中的元素而不必暴露聚集的内部表示。

Chain of Responsibility 模式

在责任链模式中，很多对象由每一个对象对其下家的引用而连接起来形成一条链。请求在这个链上传递，直到链上的某一个对象决定处理此请求。发出这个请求的客户端并不知道链上的哪一个对象最终处理这个请求，这使得系统可以在不影响客户端的情况下动态地重新组织链和分配任务。

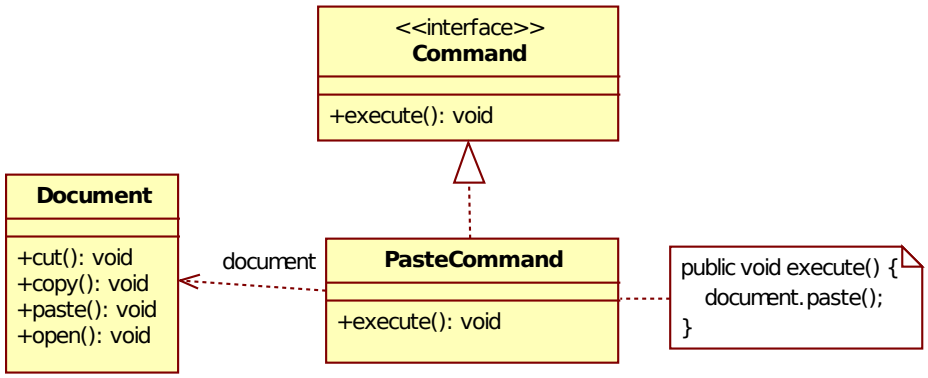


Command 模式

Command 模式把一个请求或者操作封装到一个对象中。Command 模式允许系统使用不同的请求把客户端参数化，对请求排队或者记录请求日志，可以提供命令的撤销和恢复功能。

Command 模式是对命令的封装。Command 模式把发出命令的责任和执行命令的责任分隔开，委派给不同的对象。

每一个命令都是一个操作：请求的一方发出请求要求执行一个操作；接收的一方收到请求，并执行操作。Command 模式允许请求的一方和接收的一方独立开来，使得请求的一方不必知道接收请求的一方的接口，更不必知道请求是怎么被接收，以及操作是否被执行、何时被执行，以及是怎么被执行的。

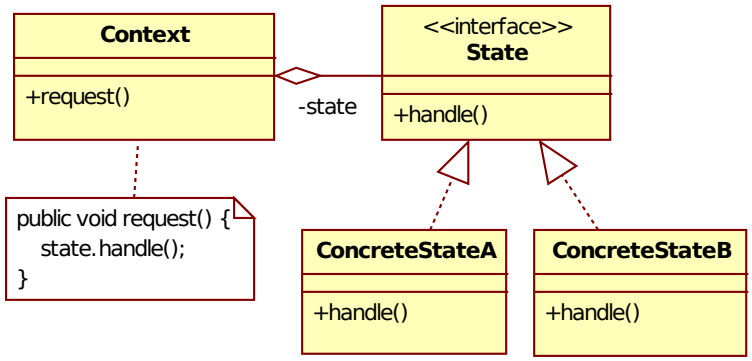


Memento 模式

Memento 模式是一个保存另外一个对象内部状态拷贝的对象，这样以后就可以将该对象恢复到原先保存的状态。

State 模式

State 模式允许一个对象在其内部状态改变的时候改变其行为。这个对象看上去就像是改变了它的类一样。



在以下各种情况下可以使用状态模式：

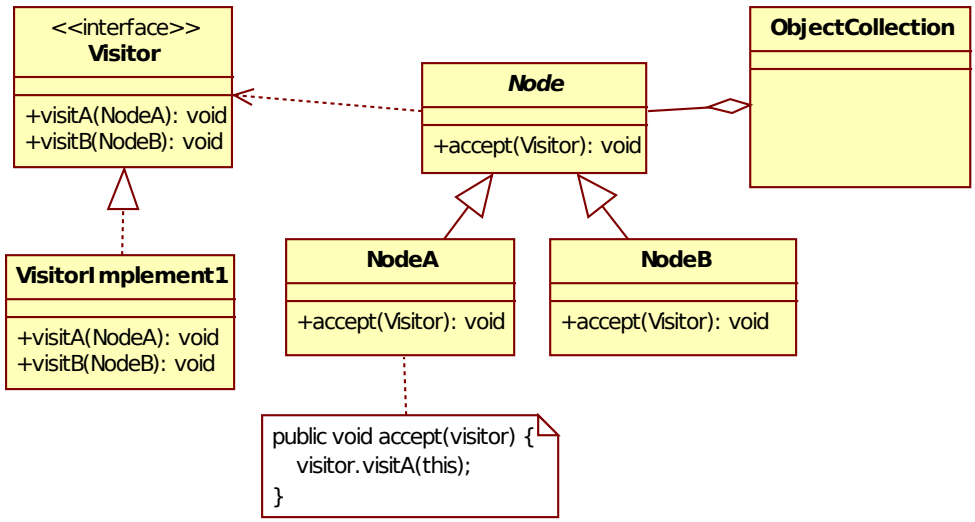
- 一个对象的行为依赖于它所处的状态，对象的行为必须随着其状态的改变而改变。
- 对象在某个方法里依赖于一重或多重的条件转移语句，其中有大量的代码。State 模式把条件转移语句的每一个分支都包装到一个单独的类里。这使得这些条件转移分支能够以类的方式独立存在和演化，维护这些独立的类也就不再影响到系统的其他部分。

State 模式与 Strategy 模式的区别

- 一个简单的方法就是考察环境角色是否有明显的状态和状态的过渡。如果环境角色只有一个状态，那么就应当使用 Strategy 模式。Strategy 模式的特点是：一旦环境角色选择了一个具体策略类，那么在整个环境类的生命周期里它都不会改变这个具体策略类。而 State 模式则适用于另一种情况，即环境角色有明显的状态转移。
- 另一个微妙的差别在于，Strategy 模式的环境类自己选择一个具体策略类；而 State 模式的环境类是被外在原因放进一个具体状态中的。
- Strategy 模式所选的策略往往并不明显地告诉客户端它所选择的具体策略；而状态模式则相反，在 State 模式里面，环境角色所处的状态是明显告诉给客户端的。

Visitor 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。



网文引用：

1. Reflect on the Visitor design pattern (Implement visitors in Java, using reflection)

http://www.javaworld.com/javaworld/javatips/jw-javatip98.html

2. 引用于 jdon.com 的论坛，作者：bang

在很多场合，我们使用 RTTI 实际是为了达到更加抽象的目的，但是抽象本身的一个问题就是无法从抽象中找到“原来的我”了。

比如一个接口抽象了很多具体的继承类，在很多地方我们都是和接口直接打交道，这样做的前提是必须在接口中声明所有需要使用的方法，这样做可以对付大部分问题。

以上是关于接口的多态，这样做的前提是，所有对象都是继承同一个接口，如果对象分属不同的抽象接口，怎么做?

Visitor 模式是对在多个抽象的对象群的一种特殊处理，适合在这样一个场景：

有一堆身份各异对象(通常是数据状态类)，这些状态类有一些特征：被动的(数据都是被动的)，需要等待外界来操作或推动。

那么现在外界有一个动作来准备操作这些状态类了，但是走到面前，突然傻眼，分不清楚谁是谁啊？这些状态类可能属于不同类型的接口，怎么办?

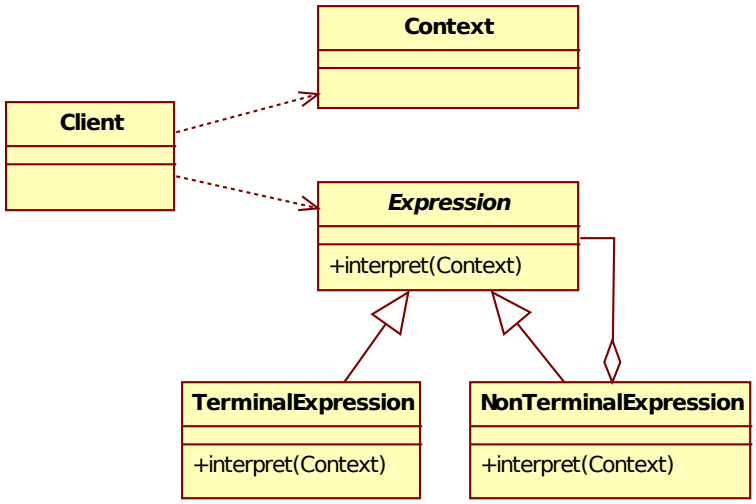
解决思路，当然把这些状态类再统一到一个接口下就可以操作了。

当然，这时需要这些状态类做些修饰，有个“对外开放”的姿态，再实现一个统一接口 Visitable，这个接口中提供的方法就更抽象：Accept()。

其实 Accept 方法的具体实现是采取了 Adapter 模式,因为各个状态类都分属不同的接口，总不能为接受访问修改这些接口吧?

Interpreter 模式

Interpreter 模式可以定义出其文法的一种表示，并同时提供一个解释器。客户端可以使用这个解释器来解释这个语言中的句子。



Interpreter 模式的几个要点

- Interpreter 模式的应用场合是 Interpreter 模式应用中的难点，只有满足“业务规则频繁变化，且类似的模式不断重复出现，并且容易抽象为语法规则的问题”才适合使用 Interpreter 模式。
- 使用 Interpreter 模式来表示文法规则，从而可以使用面向对象技巧来方便地“扩展”文法。
- Interpreter 模式比较适合简单的文法表示，对于复杂的文法表示，Interpreter 模式会产生比较大的类层次结构，需要求助于语法分析生成器这样的标准工具。

Mediator 模式

Mediator 模式包装了一系列对象相互作用的方式，使得这些对象不必相互明显引用。从而使他们可以较松散地耦合。当这些对象中的某些对象之间的相互作用发生改变时，不会立即影响到其他的一些对象之间的相互作用。从而保证这些相互作用可以彼此独立变化。