

# 设计模式

Design Patterns

---

- **设计模式分类**

- **创建型模式**
- **结构型模式**
- **行为型模式**

# 创建型模式

---

- **创建型模式的目的**

- 使系统独立于如何创建、组合和表示对象。
- 隐藏了这些类的实例是如何被创建和放在一起的

Abstract Factory

Builder

Factory Method

Prototype

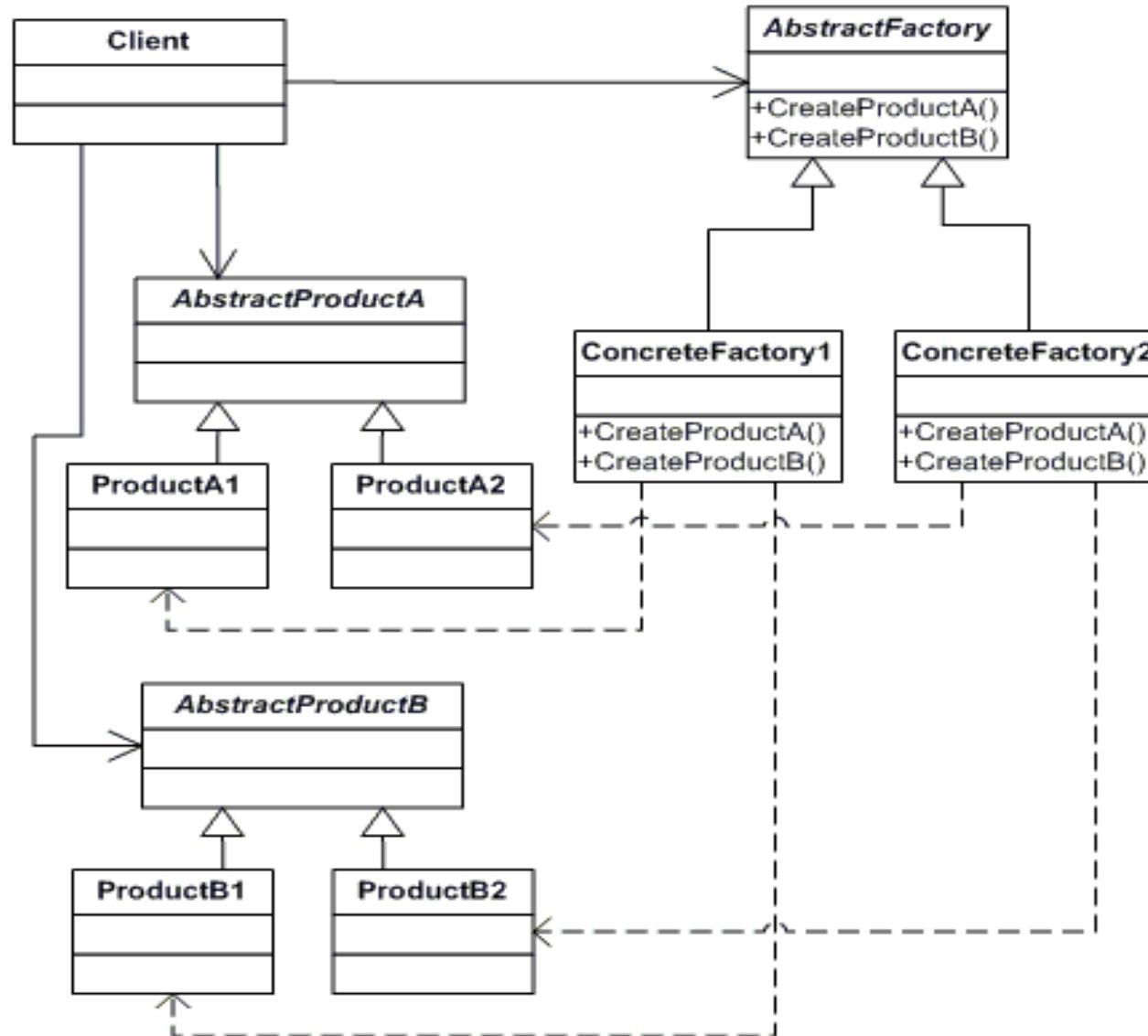
Singleton

# 创建型模式

---

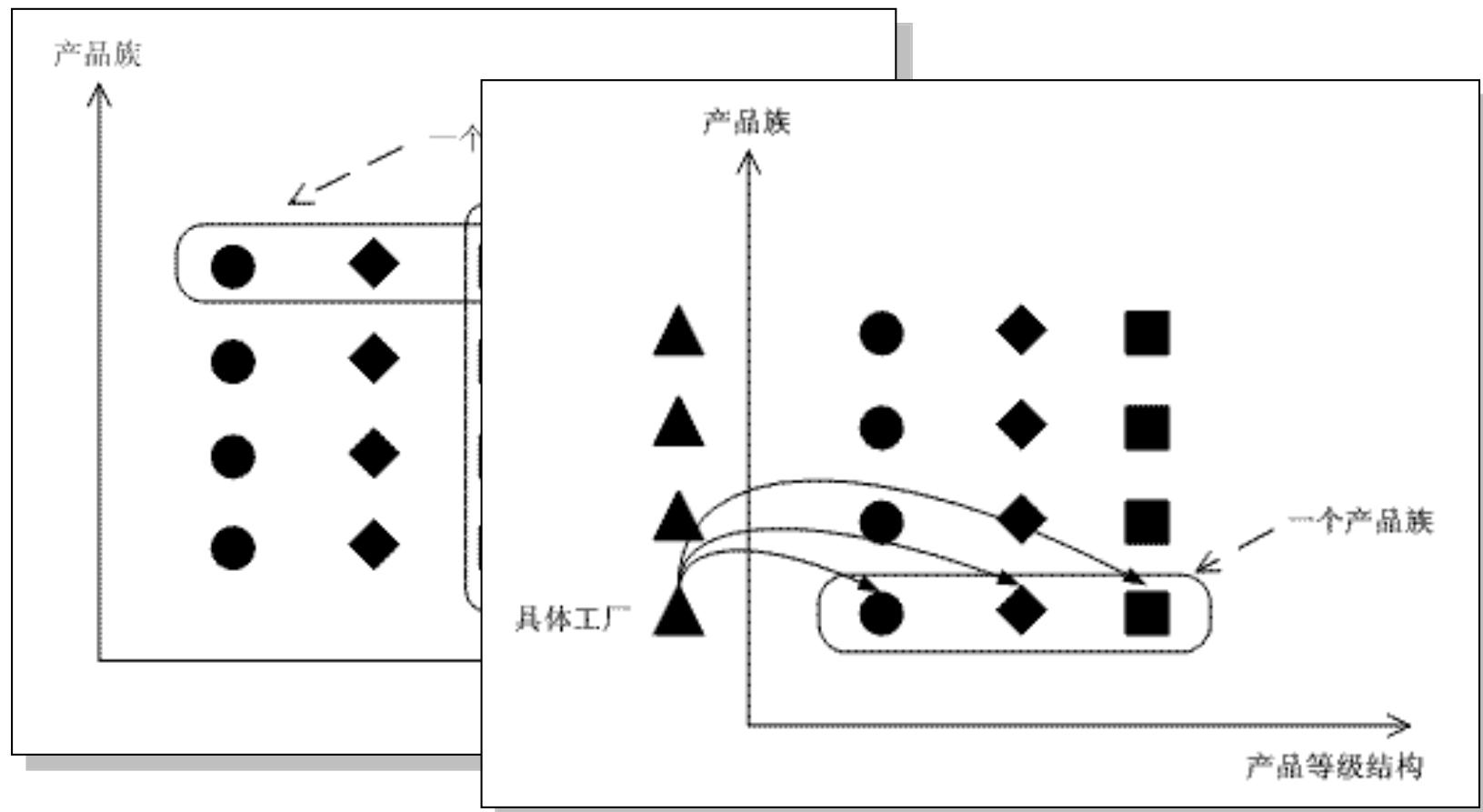
- **Abstract Factory ( 数据库的替换 )**
  - 意图
    - 提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。
  - 适用性
    - 一个系统要独立于它的产品的创建、组合和表示时。
    - 一个系统要用多个产品系列中的一个来配置时。
    - 要强调一系列相关的产品对象的设计以便进行联合使用时。
    - 提供一个产品类库，而只想显示它们的接口而不是实现时。

# Abstract Factory模式的结构



# 创建型模式

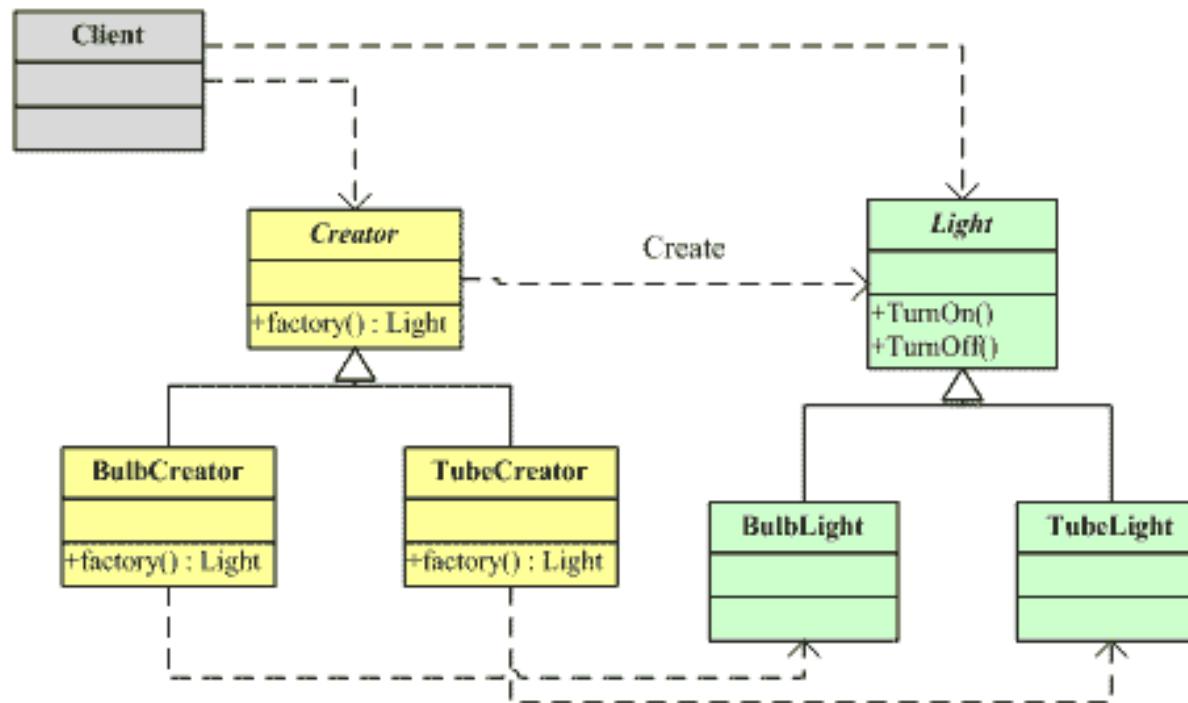
- **Abstract Factory**



- Factory Method (运算)

- 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。



- 当系统扩展需要添加新的产品对象时，仅仅需要添加一个具体对象以及一个具体工厂对象，原有工厂对象不需要进行任何修改，**也不需要修改客户端**，很好的符合了“开放 - 封闭”原则

# 创建型模式

## • Builder (建造小人)

### - 意图

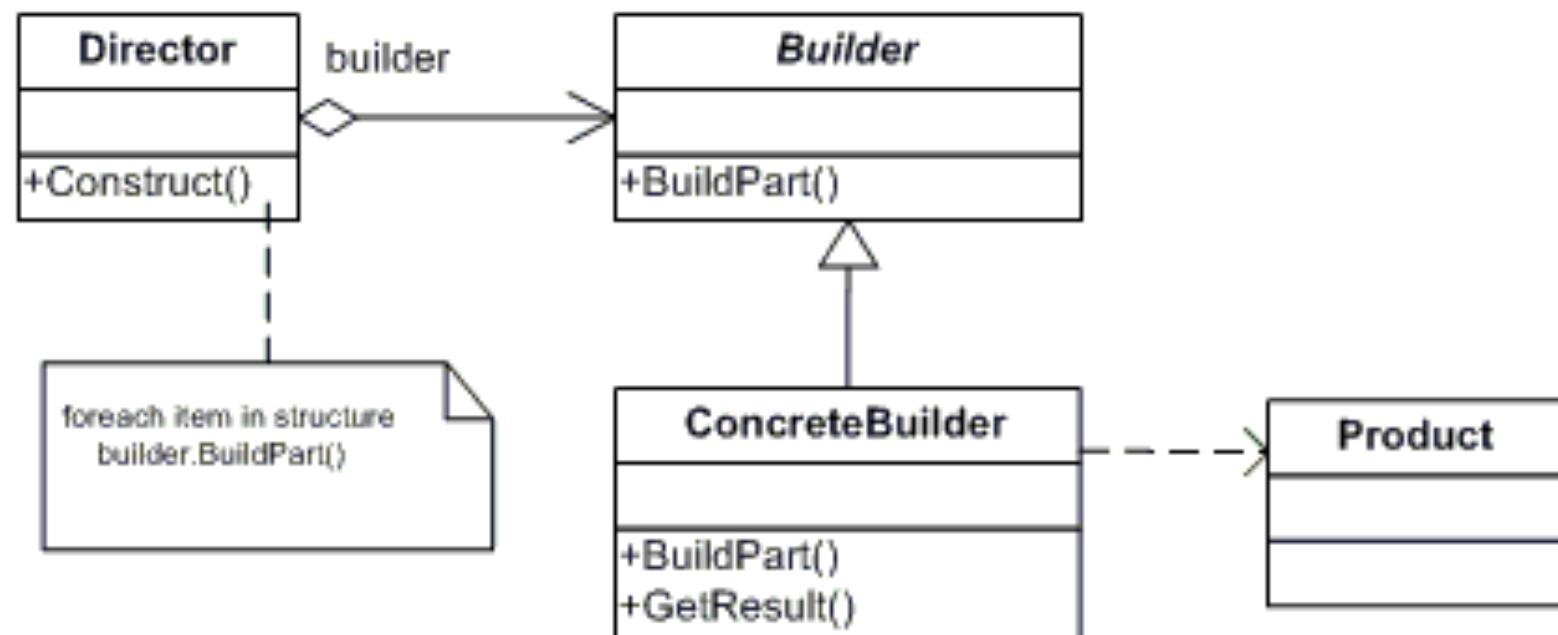
- 将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。

### - 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。

### - 效果

- 可以改变一个产品的内部表示。
- 将构造代码和表示代码分开。
- 可以对构造过程进行更精细的控制。

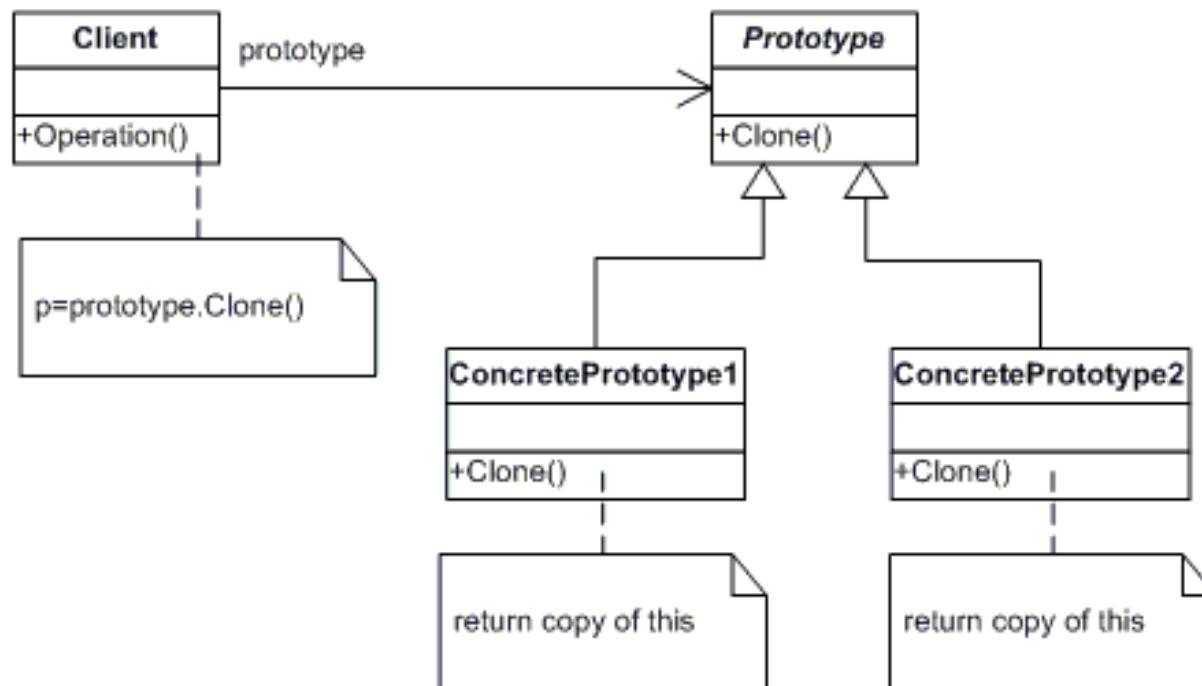


# 创建型模式

- Prototype (简历复印)

- 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。



# 创建型模式

---

- Prototype ( 原型 )

- 效果

- 可在运行时刻增加和删除产品。
    - 可以通过改变值来指定新产品。
    - 可以通过改变结构来指定新对象。
    - 减少了子类的构造。
    - 可以用类动态配置应用。

- 实现

- 使用一个原型管理器；
    - 实现克隆操作（浅拷贝和深拷贝）；
    - 初始化克隆对象；

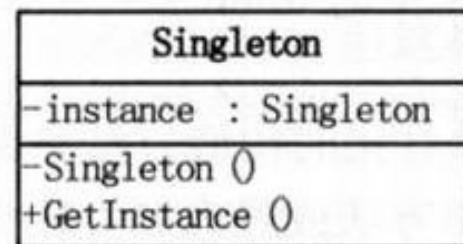
# 创建型模式

---

- **Singleton ( 打印机 )**
  - 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
    - 在一个系统要求一个类只有一个实例时才应当使用单例模式

# 创建型模式

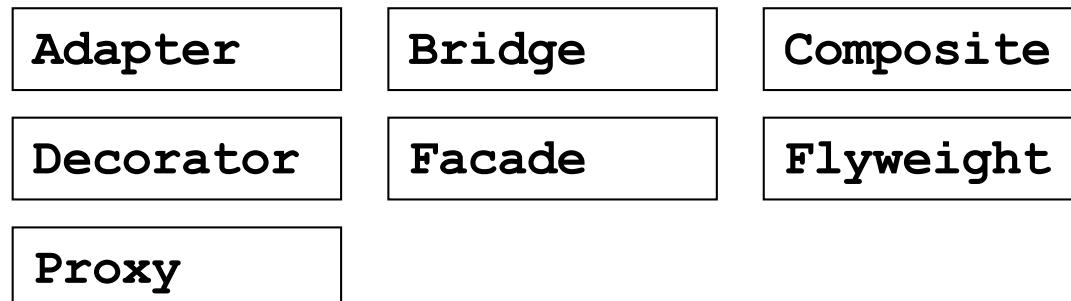
- Singleton



Singleton 类，定义一个GetInstance 操作，允许客户访问它的唯一实例。GetInstance 是一个静态方法，主要负责创建自己的唯一实例。

# 结构型模式

- 结构型模式的目的
  - 结构型模式涉及到如何组合类和对象以获得更大的结构。



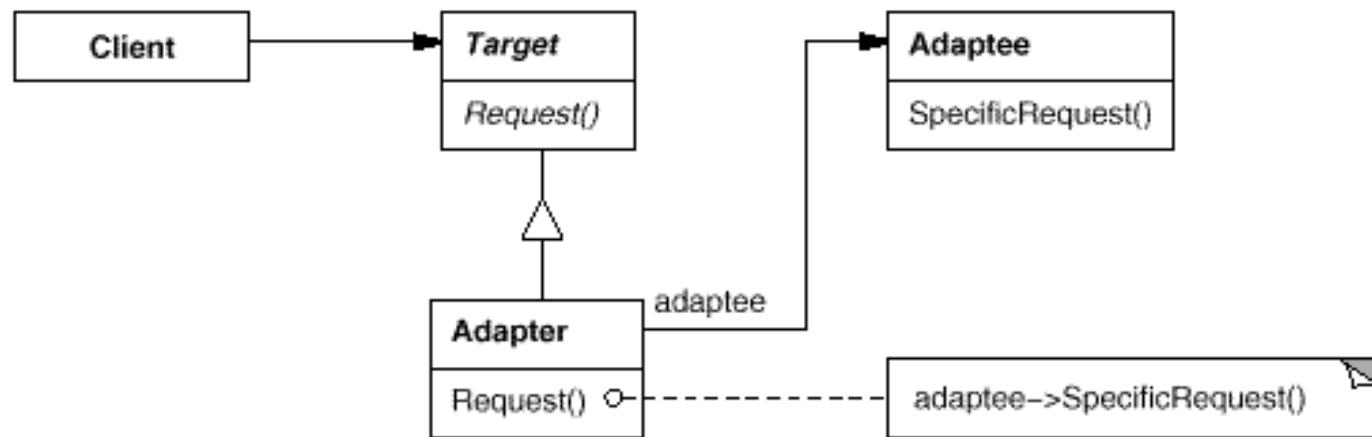
# 结构型模式

---

- Adapter ( NBA翻译 )
  - 意图
    - 将一个类的接口转换成客户希望的另外一个接口，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。

# 结构型模式

- **Adapter**



# 结构型模式

---

- Bridge ( 手机软件 )

- 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

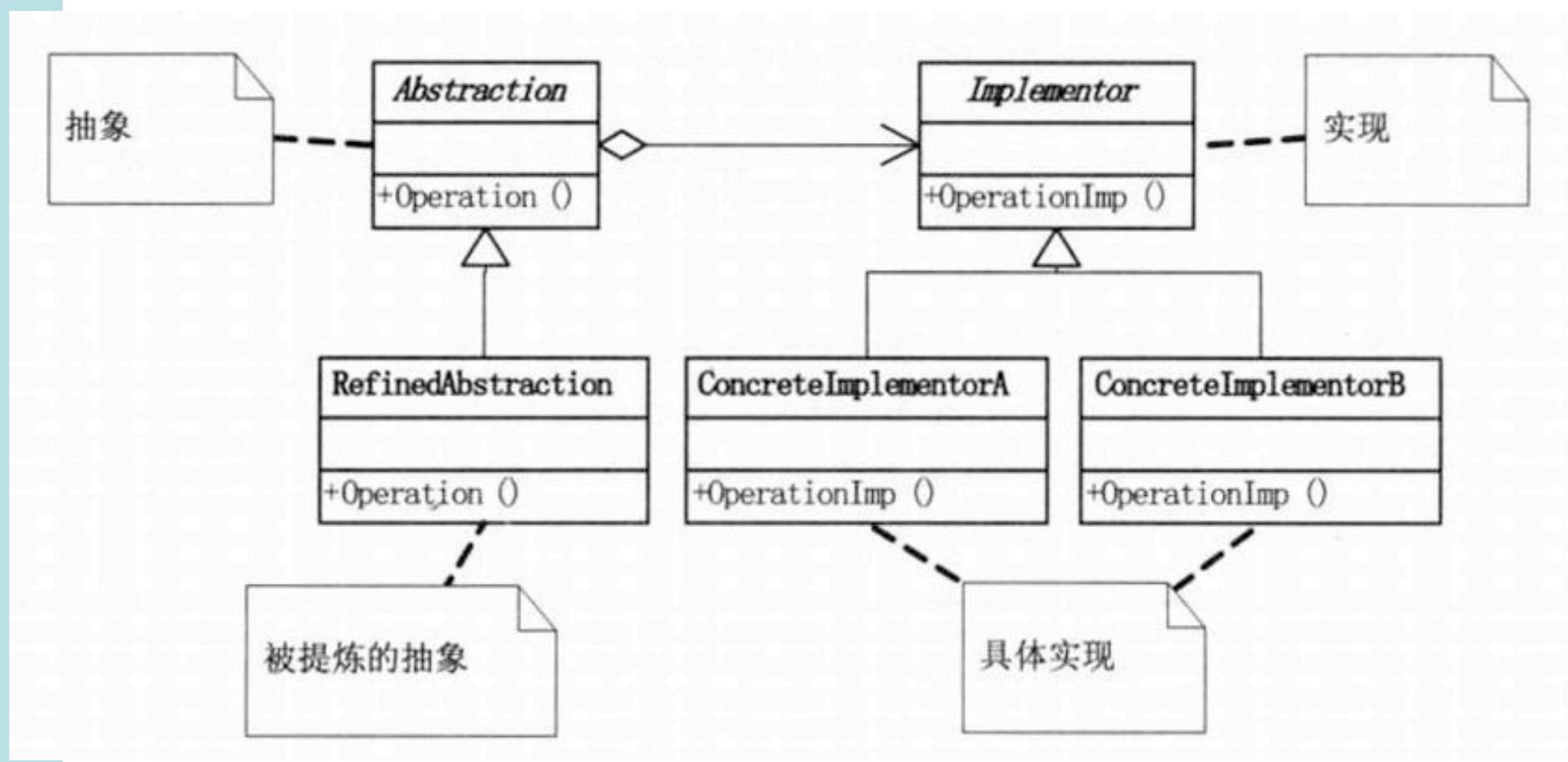
- 不希望在抽象和它的实现部分之间有一个固定的绑定关系。
    - 类的抽象以及它的实现都应该可以通过生成子类的方法加以扩充。
    - 对一个抽象的实现部分的修改应对客户不产生影响，即客户的代码不必重新编译。

- 效果

- 分离了接口及其实现部分。
    - 提高了可扩充性。
    - 实现了细节对客户透明。

# 结构型模式

- Bridge ( 桥接 )



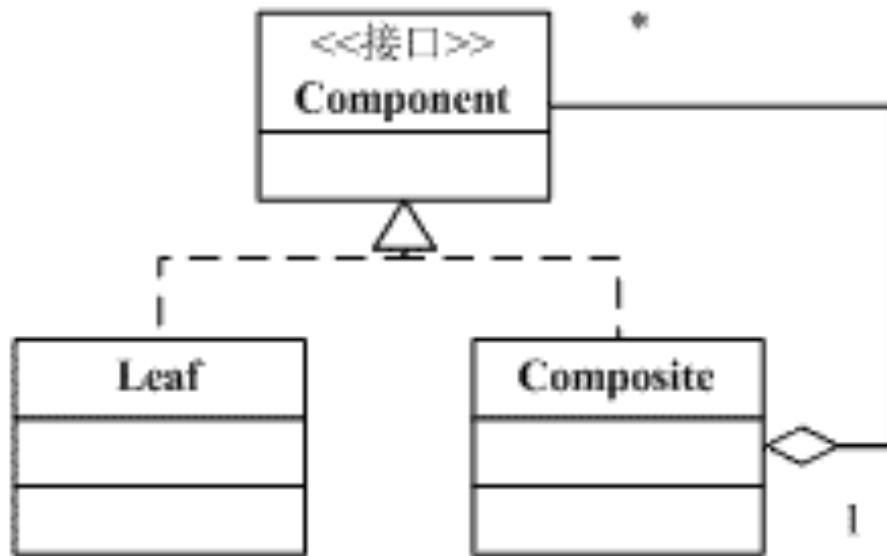
# 结构型模式

---

- Composite (公司与分公司)
  - 意图
    - 将对象组合成树形结构以表示“部分-整体”的层次结构。Composite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。

# 结构型模式

- Composite (组合)



# 结构型模式

---

- **Decorator (小菜扮靓)**

- **意图**

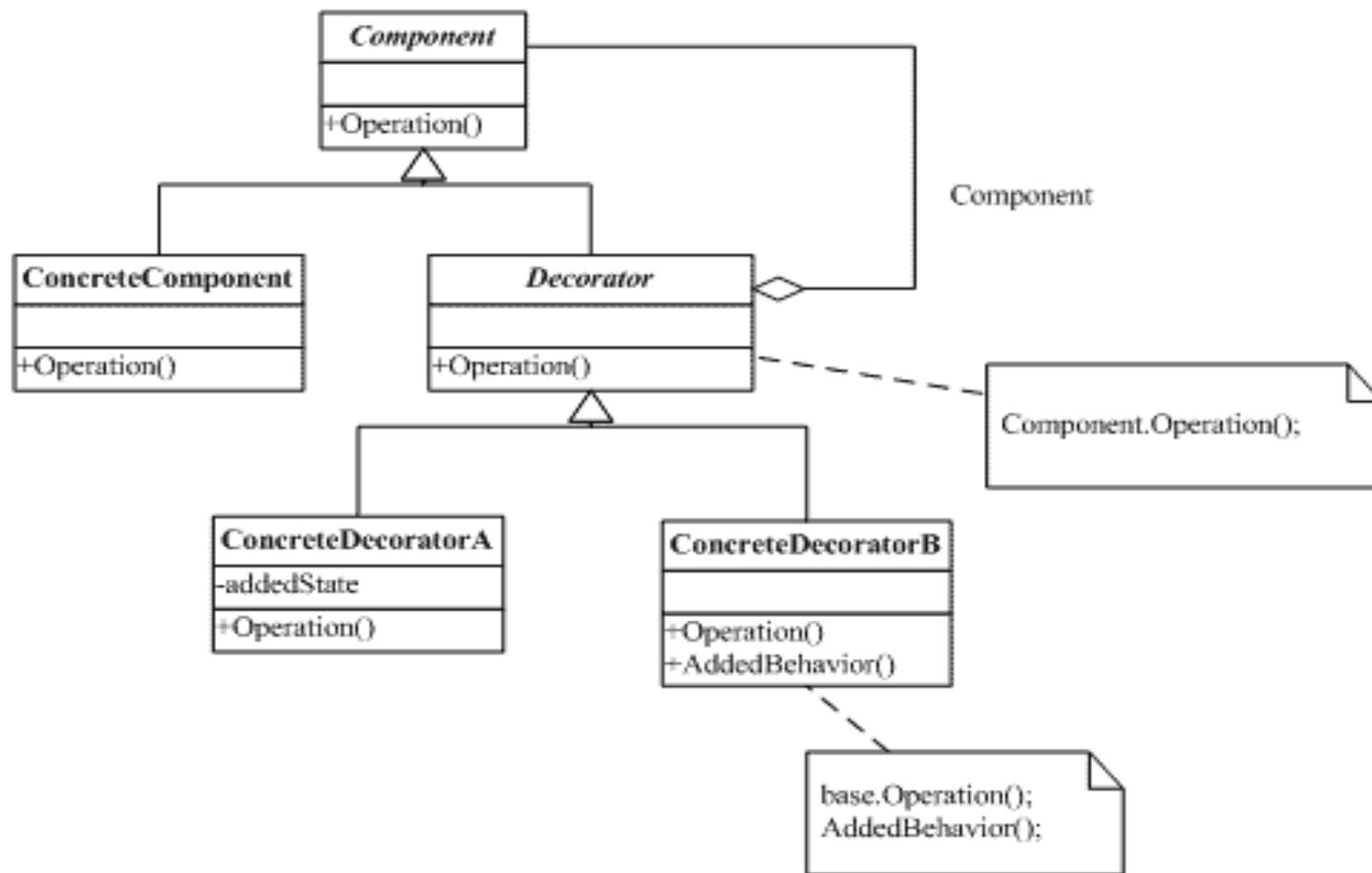
- 动态地给一个对象添加一些额外的职责。就增加功能来说，Decorator模式相比生成子类更为灵活。

- **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。

# 结构型模式

- **Decorator ( 装饰 )**



# 结构型模式

---

- **Facade ( 投资基金 )**

- 意图

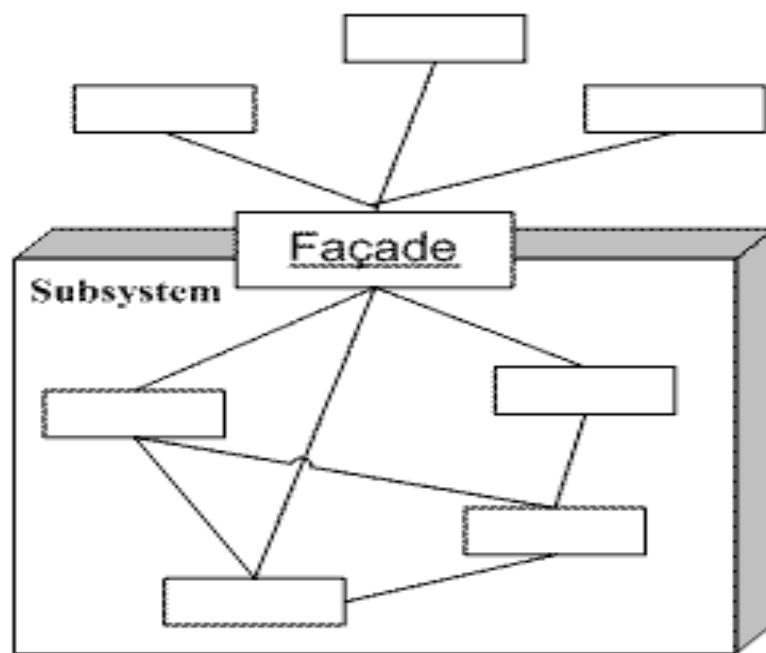
- 为子系统中的一组接口提供一个一致的界面，Facade模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。

- 适用性

- 为一个复杂子系统提供一个简单接口。
    - 当客户程序与抽象类的实现部分之间存在着很大的依赖性时，引入Facade将这个子系统与客户以及其他子系统分离，可以提高子系统的独立性和可移植性。
    - 在构建一个层次结构的子系统时，使用Facade模式定义子系统中每层的入口点。让相互依赖的子系统之间仅通过Facade进行通讯，可以简化它们之间的依赖关系。

# 结构型模式

- **Facade ( 外观 )**



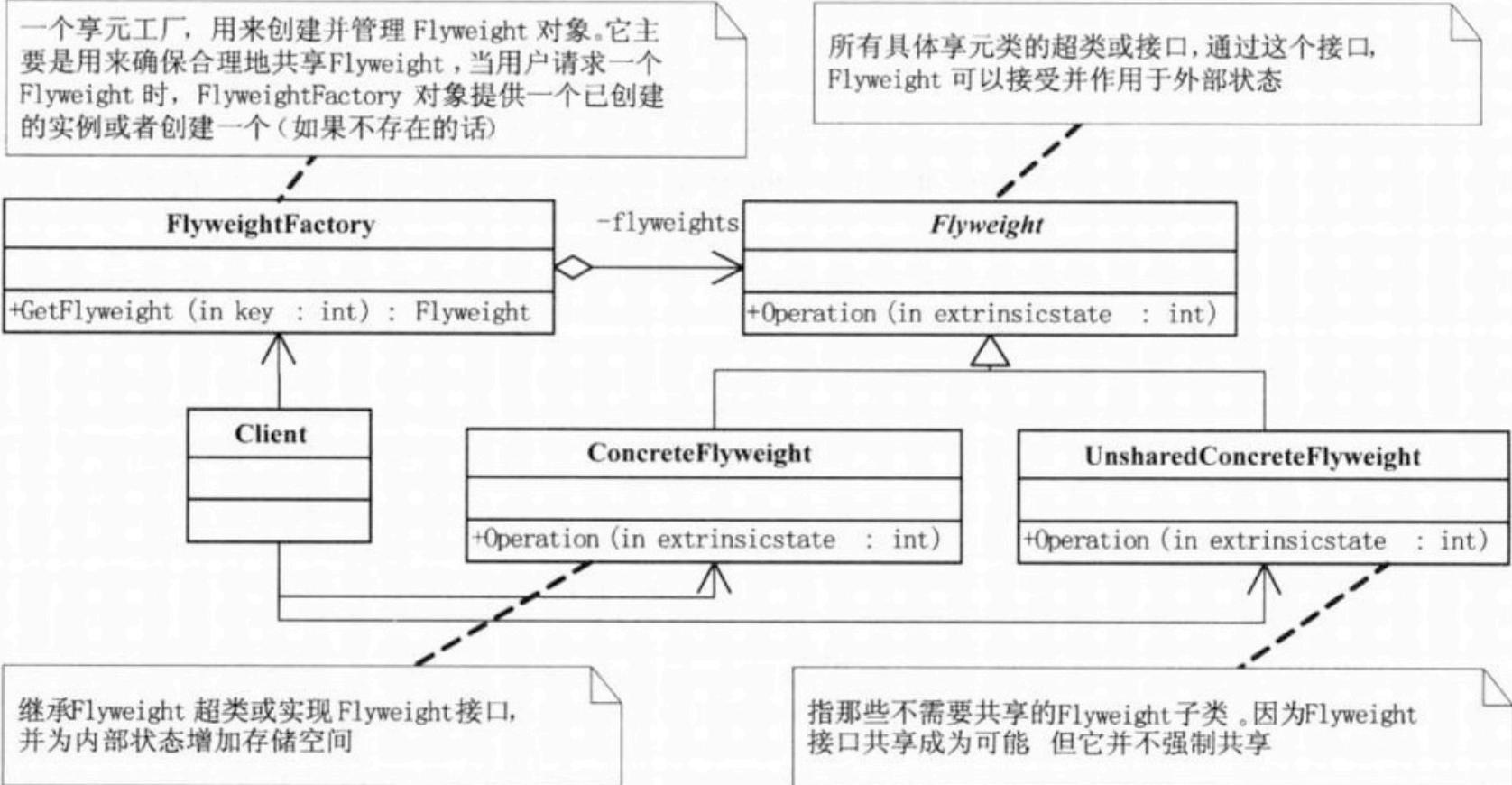
# 结构型模式

---

- Flyweight ( 网站共享、围棋 )
  - 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剔除时，每一个组都可以仅用一个对象代替。

# 结构型模式

## • Flyweight (享元)

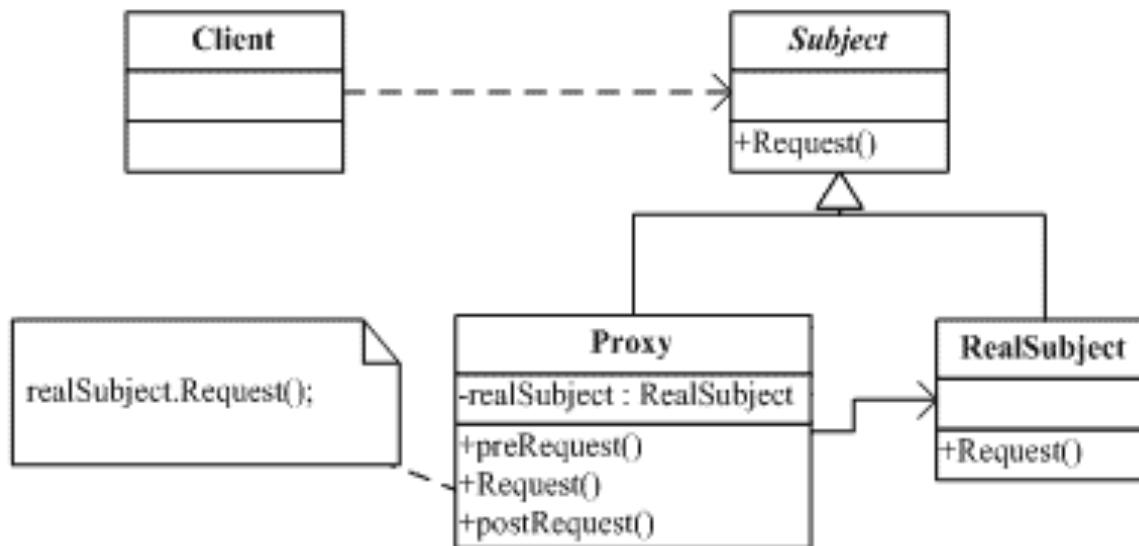


# 结构型模式

- 
- **Proxy ( 追求者-代理-被追求者 )**
    - 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
      - 1) **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。
      - 2) **虚代理 ( Virtual Proxy )** 根据需要创建开销很大的对象。
      - 3) **保护代理 ( Protection Proxy )** 控制对原始对象的访问。
      - 4) **智能引用 ( 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。

# 结构型模式

- **Proxy (代理)**



# 结构型模式

---

- 结构型模式的讨论

- Adapter和Bridge都涉及到从自身以外的接口向另一对象发送请求，Adapter是为了解决两个接口不匹配，Bridge则是对抽象接口与其实现部分进行桥接。Adapter通常在类设计好之后实施，Bridge则在类设计之前实施。
- Adapter使两个已经存在的接口协同工作，并不定义新的接口，而Facade则定义了一个新的接口。
- Composite和Decorator都是基于递归组合来组织可变数目的对象，但Composite的目的是使多个相关对象能够被当作一个对象来处理，Decorator的目的则是不需要生成子类即可给对象添加职责。
- Decorator和Proxy都描述了为对象提供间接引用，但Proxy不能动态地添加或分离性质，也不是为递归组合而设计的，是由实体完成关键功能，Proxy控制其访问，而Decorator则是在组件基本功能之外完成附加的功能。

# 行为模式

- 行为模式的目的

- 行为模式涉及到算法和对象间职责的分配。
- 行为模式描述对象或类之间的通信模式，刻画了在运行时难以跟踪的复杂的控制流。

Chain of Responsibility

Command

Interpreter

Iterator

Mediator

Memento

Observer

State

Strategy

Template Method

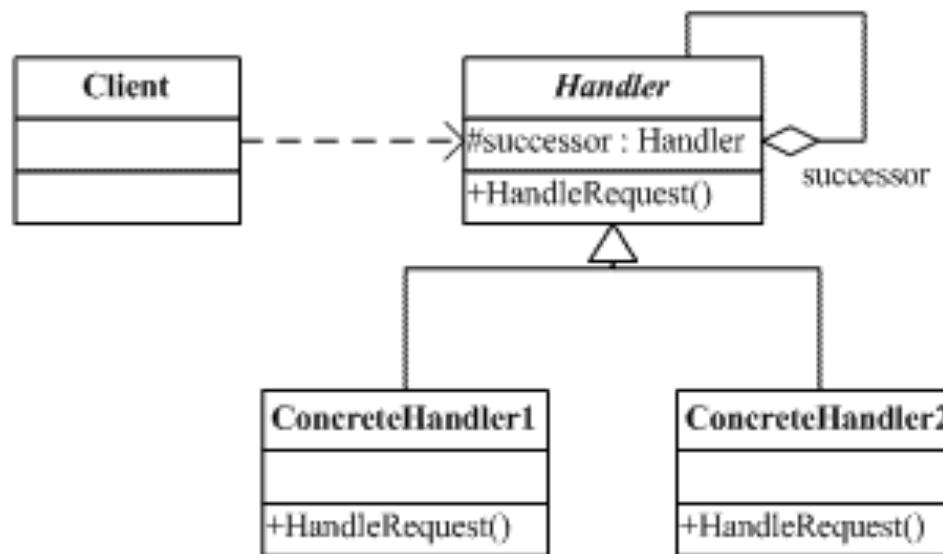
Visitor

# 行为模式

- **Chain of Responsibility (击鼓传花)**
  - 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
    - 有多个对象可以处理一个请求，哪个对象处理该请求在运行时刻自动确定。
    - 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。
  - 效果
    - 降低了对象间的耦合度。
    - 增强了给对象分配响应（职责）的灵活性。

# 行为模式

- Chain of Responsibility



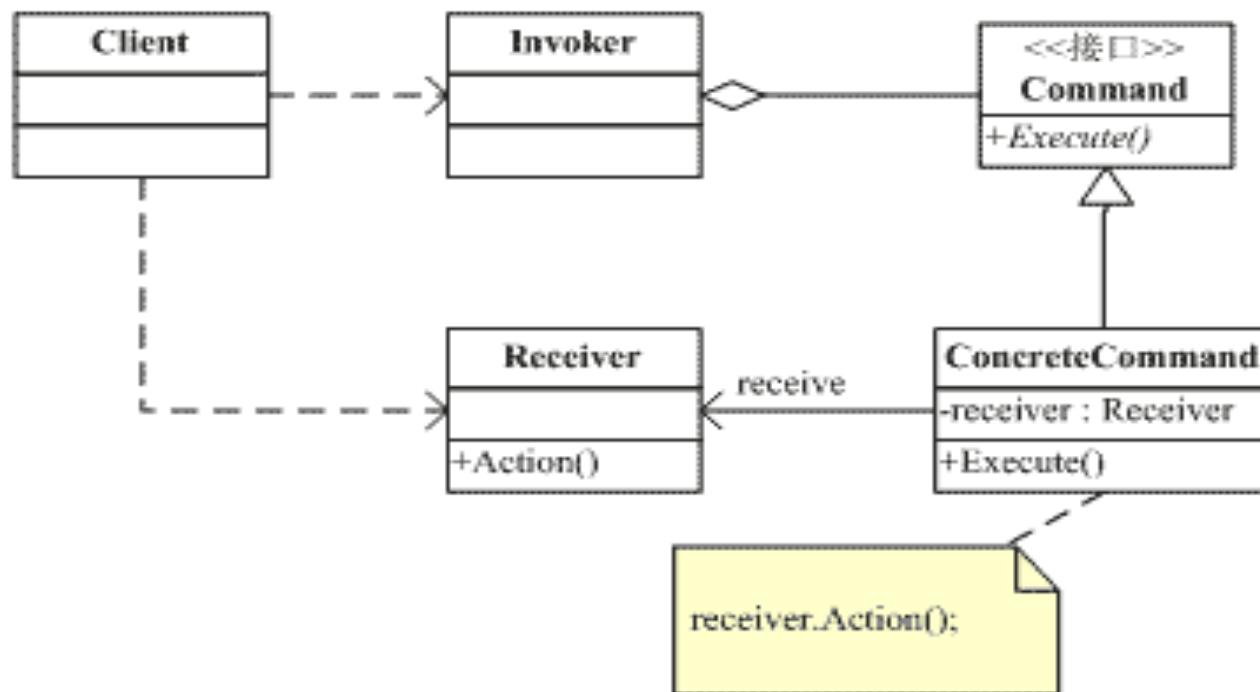
# 行为模式

---

- Command ( 烤羊肉串 )
  - 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，以及支持可撤销的操作。
  - 适用性
    - 抽象出待执行的动作以参数化某对象。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。
    - 支持取消操作。
    - 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。

# 行为模式

- Command (命令)



# 行为模式

---

- Command ( 命令 )
  - 效果
    - 将调用操作的对象与实现操作的对象解耦。
    - Command对象也可以被操纵和扩展。
    - 可以将多个Command装配成一个复合Command。
    - 无需改变已有的类，可以容易地增加新Command。

# 行为模式

---

- Interpreter (自学)
  - 意图
    - 根据语言的文法，定义一个解释器，用来解释语言中的句子。
  - 适用性
    - 当有一个语言需要解释执行，并且该语言中的句子可以表示为一个抽象语法树时。  
当满足以下情况时，解释器模式的效果最好：
      - 文法简单。
      - 效率不是一个关键问题。

# 行为模式

---

- **Iterator ( 迭代器 )**
  - **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个和多种遍历。
    - 为遍历不同的聚合结构提供一个统一的接口。

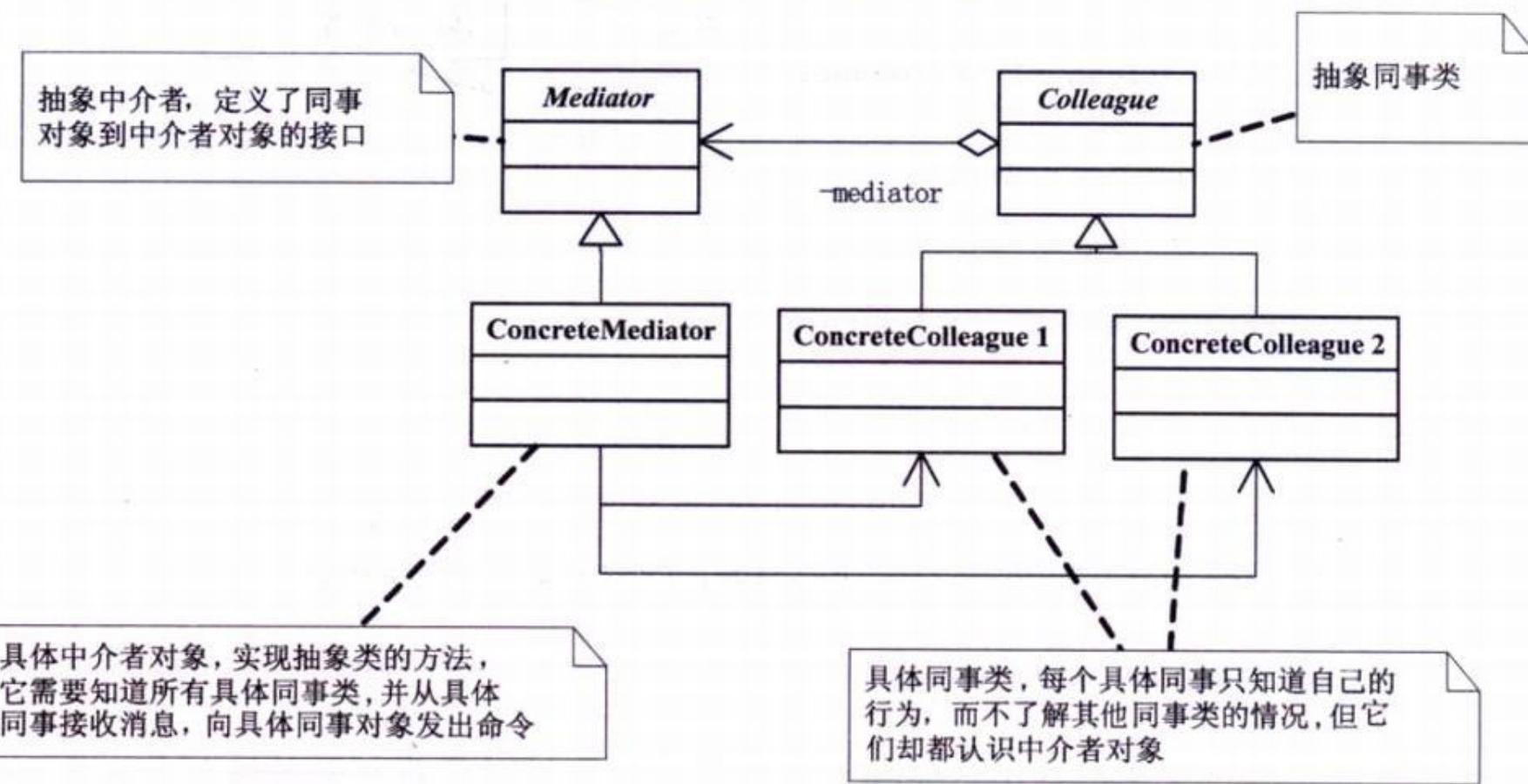
# 行为模式

---

- **Mediator (世界需要和平)**
  - 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。

# 行为模式

- Mediator (中介者)



# 行为模式

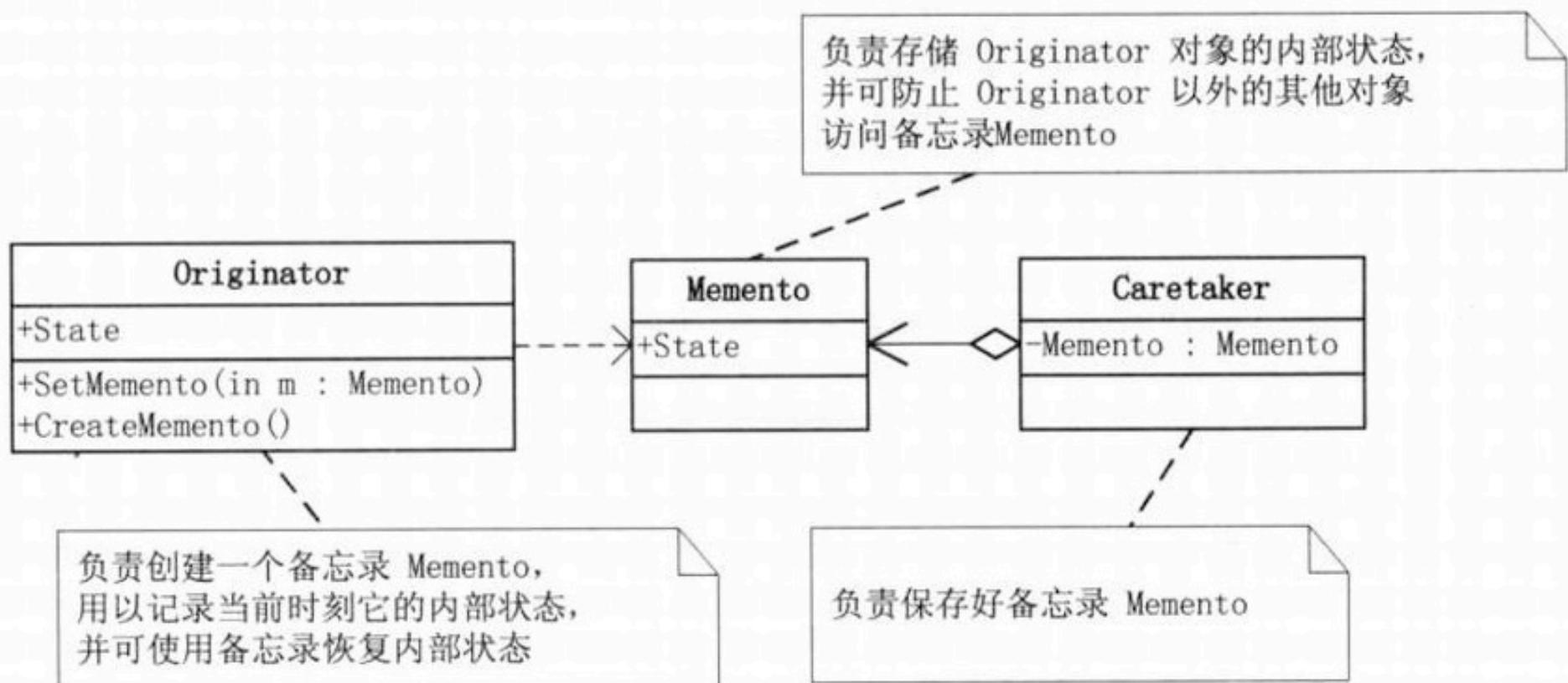
---

- **Memento ( 游戏存档 )**
  - 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原先保存的状态。
  - 适用性
    - Memento模式比较适用于功能比较复杂的，但需要维护或记录属性历史的类。

# 行为模式

- Memento (备忘录)

备忘录模式 (Memento) 结构图



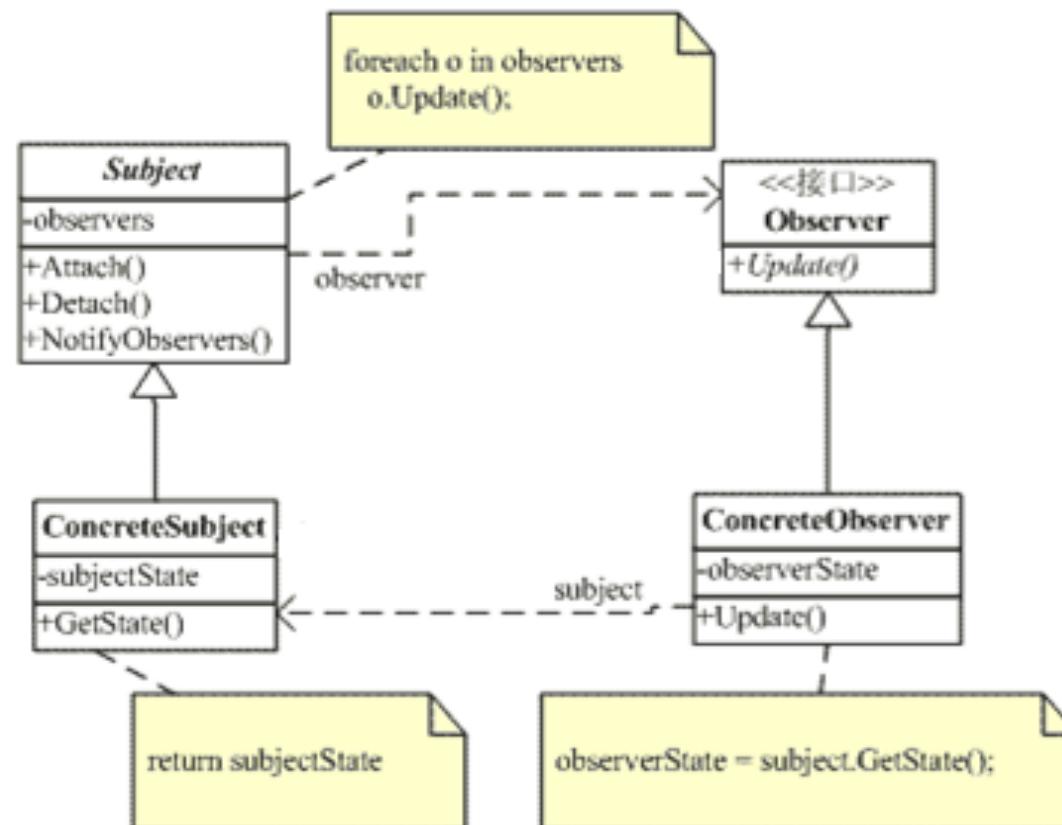
# 行为模式

---

- **Observer (老板回来，得到通知)**
  - 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，将这两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，不希望这些对象是紧密耦合的。

# 行为模式

- Observer (观察者)



# 行为模式

---

- **State (状态)**

- **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**

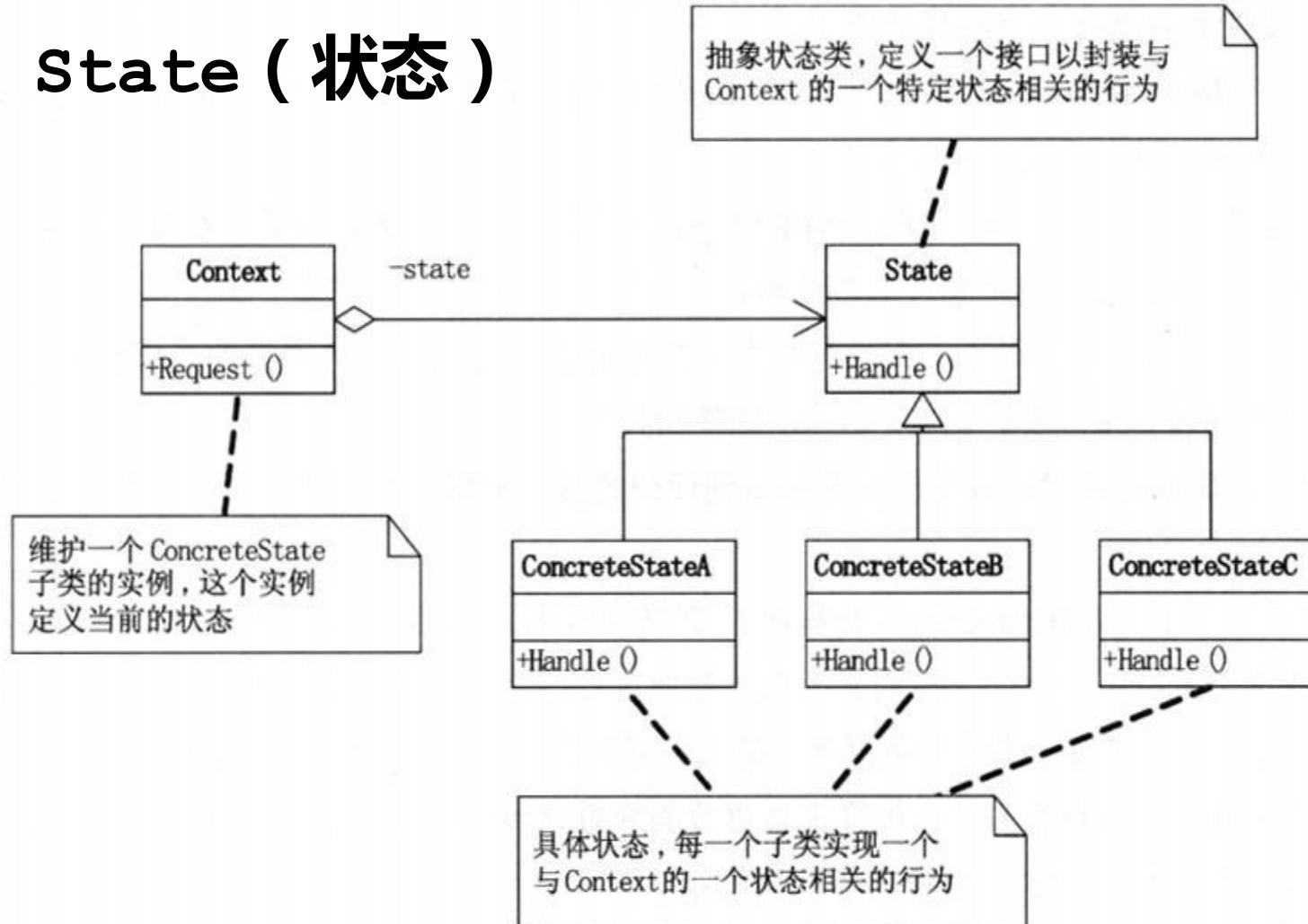
- 一个对象的行为取决于它的状态，并且它必须在运行时刻根据状态改变它的行为。

- 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。

# 行为模式

## • State ( 状态 )



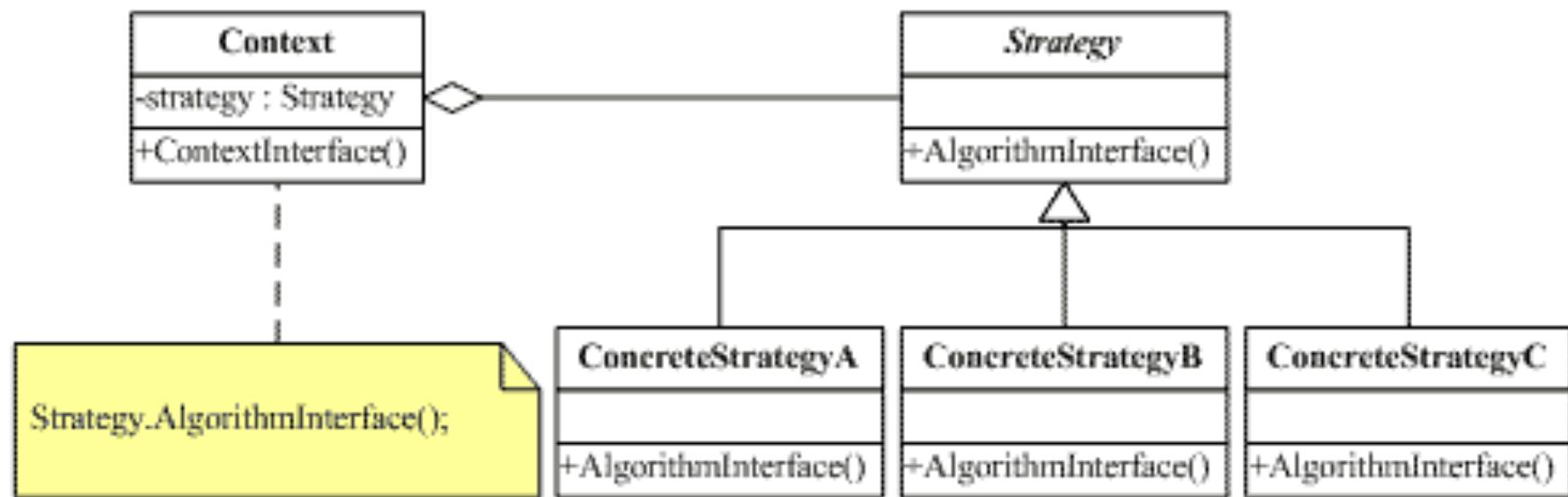
# 行为模式

---

- **Strategy ( 商场促销 )**
  - 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。
    - 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。

# 行为模式

- Strategy (策略)



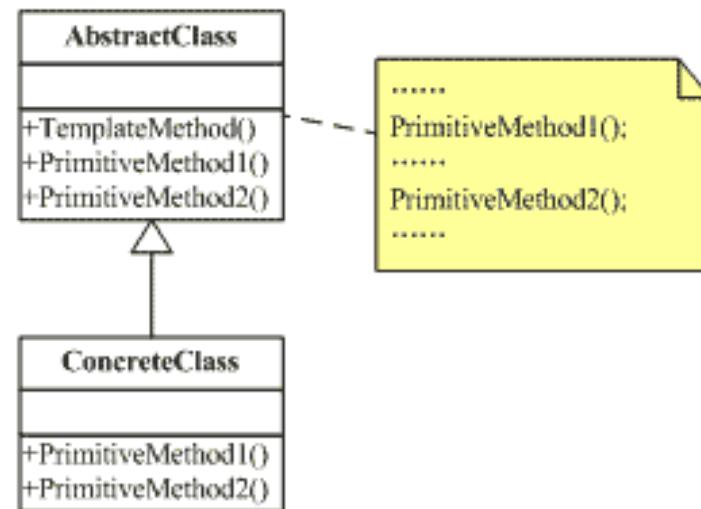
# 行为模式

---

- **Template Method (考试考题)**
  - 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。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。
  - 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。

# 行为模式

- **Template Method ( 模板方法 )**



# 行为模式

---

- **Visitor (男人-女人)**

- **意图**

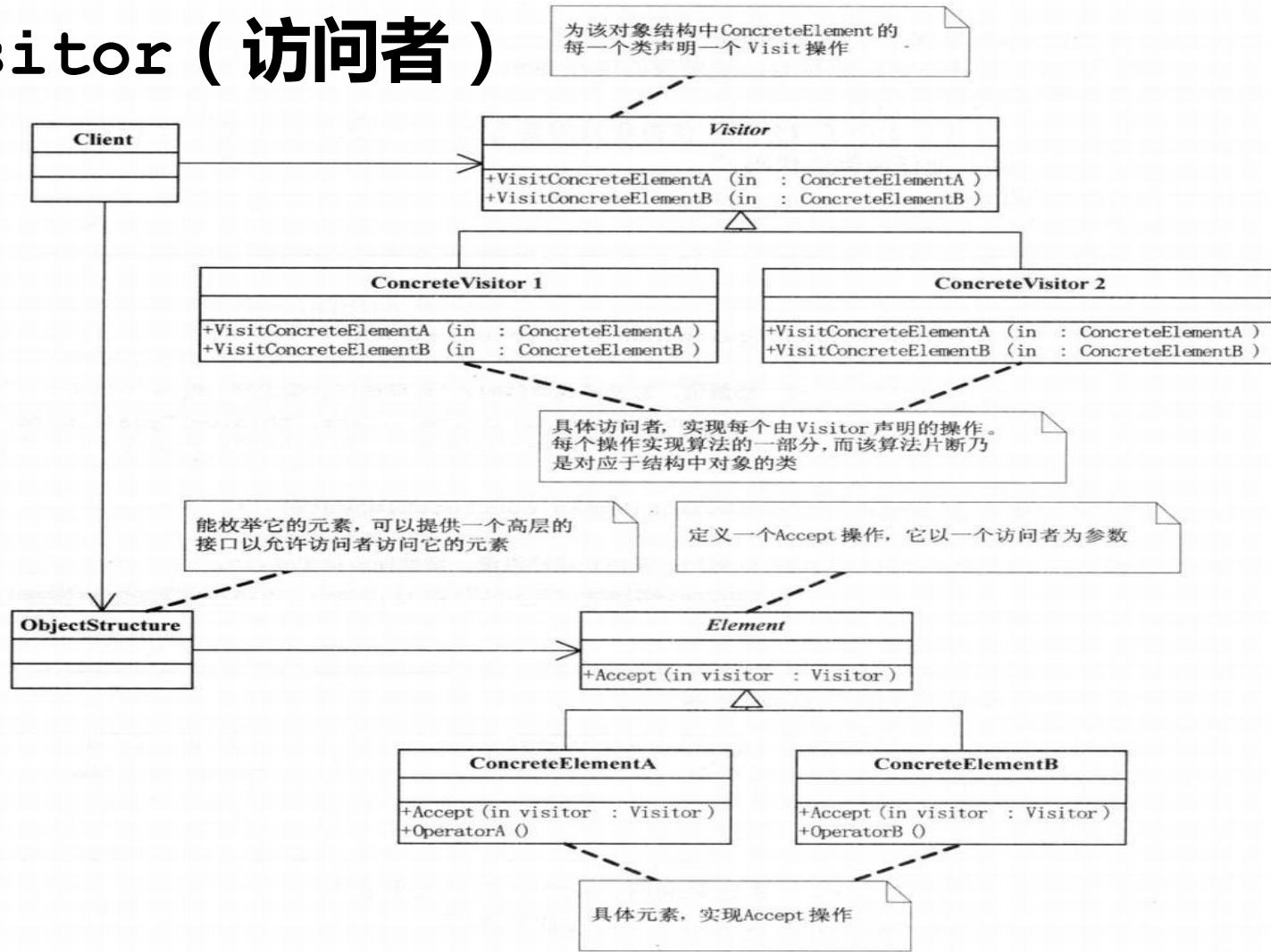
- 表示一个作用于某对象结构中的各元素的操作。它使你可以在不改变各元素的类的前提下定义作用于这些元素的新操作。

- **适用性**

- 访问者模式的目的是要把处理从数据结构分离出来
    - 访问者模式仅应当在被访问的类结构非常稳定的情况下使用

# 行为模式

## • Visitor ( 访问者 )



# 设计模式六个基本原则

---

- OO设计根本的指导原则是提高可维护性和可复用性。这些原则主要有：
  - **开放封闭**
  - **依赖倒转**
  - **里氏代换**
  - **合成/聚合复用**
  - **迪米特**
  - **单一职责**
  - **接口隔离原则**

# 开闭原则

- 一个软件实体应该对扩展开放，对修改关闭。  
在设计一个模块的时候，应当使这个模块可以在不被修改的前提下被扩展。
- 如何做到既不修改，又可以扩展？  
解决问题的关键在于抽象化：  
在C#语言里，可以给出一个或多个抽象C#类或C#接口，规定出所有的具体类必须提供的方法特征作为系统设计的抽象层。这个抽象层预见了所有的可能扩展，因此，在任何扩展情况下都不会改变。这就使得系统的抽象层不需要修改，从而满足了-对修改关闭。  
同时，由于从抽象层导出一个或多个新的具体类可以改变系统的行为，因此系统的设计对扩展是开放的。

- **开闭原则实际上是“对可变性的封闭原则”：找到一个系统的可变因素，将之封装起来。这个原则意味着两点：**
  - 1) 一个可变性不应当散落在代码的很多角落里，而应当被封装到一个对象里面。同一种可变性的不同表象意味着同一个继承等级结构中的具体子类。
  - 2) 一种可变性不应当与另一种可变性混合在一起。（所有类图的继承结构一般不会超过两层，不然就意味着将两种不同的可变性混合在了一起。）

**开闭原则是总的原则，其它几条是开闭原则的手段和工具。**

# 依赖倒转原则

---

- 依赖倒转原则讲的是：要依赖于抽象，不要信赖于实现。
- 开闭原则是目标，而达到这一目标的手段是依赖倒转原则。

# 里氏代换原则

---

- 任何基类可以出现的地方，子类一定可以出现。

开闭原则的关键步骤是抽象化。而基类与子类的继承关系就是抽象化的具体体现，里氏代换原则是对实现抽象化的具体步骤的规范

# 合成 / 聚合复用原则

- 要尽量使用合成 / 聚合，而不是继承关系达到复用的目的。

**合成 / 聚合原则**要求我们首先考虑合成 / 聚合关系，里氏代换原则要求在使用继承时，必须确定这个继承关系符合一定的条件（继承是用来封装变化的；任何基类可以出现的地方，子类一定可以出现。）

**合成 / 聚合原则**就是在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。

# 迪米特原则

---

- 一个软件实体应当尽可能少的其他实体发生相互作用。模块之间的交互要少。这样做的结果是当系统的功能需要扩展时，会相对更容易地做到对修改的关闭。
- 一个对象应当对其他对象有尽可能少的了解。

# 接口隔离原则

- 应当为客户端提供尽可能小的单独接口，而不是提供大的总接口。即：使用多个专门的接口比使用单一的总接口要好。

接口隔离原则与迪米特都是对一个软件实体与其他的软件实体的通信限制。迪米特原则要求尽可能地限制通信的宽度和深度，接口隔离原则要求通信的宽度尽可能地窄。这样做的结果使一个软件系统在功能扩展过程当中，不会将修改的压力传递到其他对象。

# 杂谈

---

- GoF在《设计模式》一书中提到，如果不是一个有经验的面向对象设计人员，建议从最简单最常用的设计模式入门，比如 AbstractFactory模式、Adapater模式、Composite模式、Decorator模式、Factory模式、Observer模式、Strategy 模式、Template模式等。
  - 加上几个我觉得在开发中会很有用的模式：  
Singleton模式、Facade模式和Bridge模式。
- 懂了设计模式，你就懂了面向对象分析和设计（OOA/D）的精要。“道可道，非常道”。道不远人，设计模式亦然如此。

- 面向对象系统的分析和设计实际上追求的就是两点，一是高内聚（Cohesion），而是低耦合（Coupling）。这也是我们软件设计所追求的，因此无论是OO中的封装、继承、多态，还是我们的设计模式的原则和实例都是在为了这两个目标努力着、贡献着

- 设计模式体现的是一种思想，而思想则是指导行为的一切，理解和掌握了设计模式，并不是说记住了23种（或更多）设计场景和解决策略（实际上这也是很重要的一笔财富），实际接受的应该是一种思想的熏陶和洗礼，等这种思想融入到了你的思想中后，你就会不自觉地使用这种思想去进行你的设计和开发，这一切才是最重要的。