※ABeen※ 设计模式

学习摘要

目录

第一部分、 为什么要学习设计原则/模式 3
第二部分、 设计原则之关系 4
第三部分、 设计模式创建模式 5
3.1. 简单工厂模式 5
3.2. 工厂方法模式 6
3.3. 抽象工厂模式 7
3.4. 单例模式(Singleton) 9
3.5. 建造模式 11
3.6. 原始模型模式 12
第四部分、 设计模式结构模式 14
4.1. 适配器模式 15
4.2. 合成(Composite)模式 17
4.3. 装饰(Decorator)模式 18
4.4. 代理(Proxy)模式 20
4.5. 享元(Flyweight)模式 21
4.6. 门面(Facade)模式 24
4.7. 桥梁(Bridge)模式 24
第五部分、 设计模式行为模式 26
5.1. 不变模式 26
5.2. 策略模式 26
5.3. 模板方法模式 27
5.4. 观察者模式 28
5.5. 迭代子模式 30
5.6. 责任链模式 32
5.7. 命令模式 33
5.8. 备忘录模式 34
5.9. 状态模式 36
5.10. 访问者模式 37
5.11. 解释器模式 39
5.12. 调停者模式 39

第一部分、 为什么要学习设计原则/模式

首先考虑一款软件系统的生命周期。任何事物都要经历两次创造过程，软件项目系统也不例外。系统设计师拿到系统的设计需求后，首先系统存在于设计者们的头脑中，然后存在于设计图纸上，然后变成原型系统，最后变成真实的、可以交付客户使用的产品。此时这款软件系统在客户、设计师和开发人员眼中就如天仙妹一样动人心弦。

不久发生了变化，客户看到运行中的系统，提出一些“小小的”的修改要求，这些问题都是在系统设计需求中遗忘的。由于设计需求遗忘，设计者们只好采用一些权宜之计来满足客户的修改要求。这样一来仙妹就长了几个“青春痘”。时间一长“青春痘”越来越多，当这些“青春痘”当成了系统的重要组成部分后，此时这款系统软件的生命就应该结束，成了一堆腐烂的代码了。

为什么提点“小小的”修改就会成这样呢？ 原因当然很多，试想如果系统的可扩展性和可维护性做的好点，原设计者和维护者有一定的沟通，软件的生命周期可能会更长些。

因此出现了一个重要概念：支持可维护性的复用，也就是保持甚至提高系统的可维护性的同时，实现系统的复用。面向对象设计的复用可以帮你抓住这两只同事奔跑的兔子。然而在面向对象设计里，可维护性复用是以设计原则和设计模式为基础的。如果原设计者和维护者，都对面向对象设计有一定的理解的话，那么无形中就会达到一种潜意识的沟通。

第二部分、 设计原则之关系

开闭原则(OCP)、里氏代换原则(LSP)、依赖倒转原则(DIP)

开闭原则说：软件实体应该是扩展开放，对修改关闭。找到系统中的可变因素，并将之封装起来。

开闭原则的关键是抽象化，利用抽象化来达到开闭原则的目标。抽象化只所以是关键，因为只有很好的抽象化后，才能在此基础上导出具体化实现。

里氏代换原则中说，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。可以说里氏代换原则是对实现抽象化具体步骤的规范。

依赖倒转原则讲的是要依赖抽象，不要依赖具体。如果开闭原则是目标的话，那么依赖倒转原则便是实现这个目标的一种手段。

另外依赖倒转使用后，为了避免对具体类的直接使用可能会使用到对象工厂。由于依赖倒转还会导致大量类，对于不熟悉面向对象技术的工程师来说，维护这样的系统需要较好的面向对象设计的知识。

第三部分、 设计模式创建模式

创建模式是对类的实例化过程的抽象化，分为类的创建模式和对象的创建模式。

类的创建模式：

使用继承关系把类的创建过程延迟到子类，从而封装了客户端将得到哪些具体类的信息，并且隐藏了这些类的实例是如何创建和组合在一起的。

对象的创建模式：

把对象的创建过程动态的委派给另一个对象，从来动态的决定客户端将得到哪些具体类的实例，以及这些类是实例是如何创建和组合在一起的。

3.1. 简单工厂模式

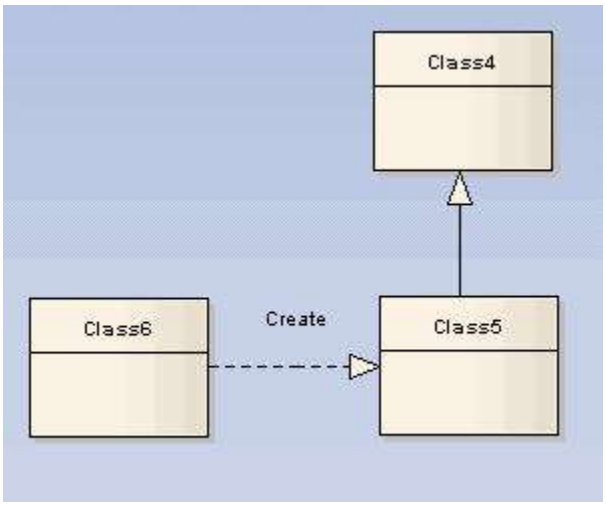
简单工厂模式是类的创建模式(静态工厂方法模式 Static Factory Method)，是由一个工厂对象决定创建出哪一种产品类的实例。

工厂是此模式的核心，它对客户封装了具体产品实例的创建过程，从而实现对责任的分割。但如果产品有复杂的多层结构时，工厂只有自己来负责所有产品的创建逻辑，形成一个无所不知的全能类。如果此类出问题整个系统都会受到影响。

优点：客户端相对独立于产品的创建过程，引入一款新产品时客户端无需修改。支持开闭原则。

缺点：对开闭原则支持不够。每引进一款产品，都要修改工厂对象。

简单工厂模式结构：



- 工厂类角色

  担任这个角色的是工厂方法模式的核心，含有与应用紧密相关的商业逻辑。工厂类在客户端的直接调用下创建产品对象，它往往由一个具体类实现。

- 抽象产品角色

  担任这个角色的类是工厂方法模式所创建对象的父类，或它们共同拥有的接口。抽象产品角色可以用一个接口或者抽象类实现。

- 具体产品角色

  工厂方法模式所创建的任何对象都是这个角色的实例，具体产品角色由一个具体类实现。

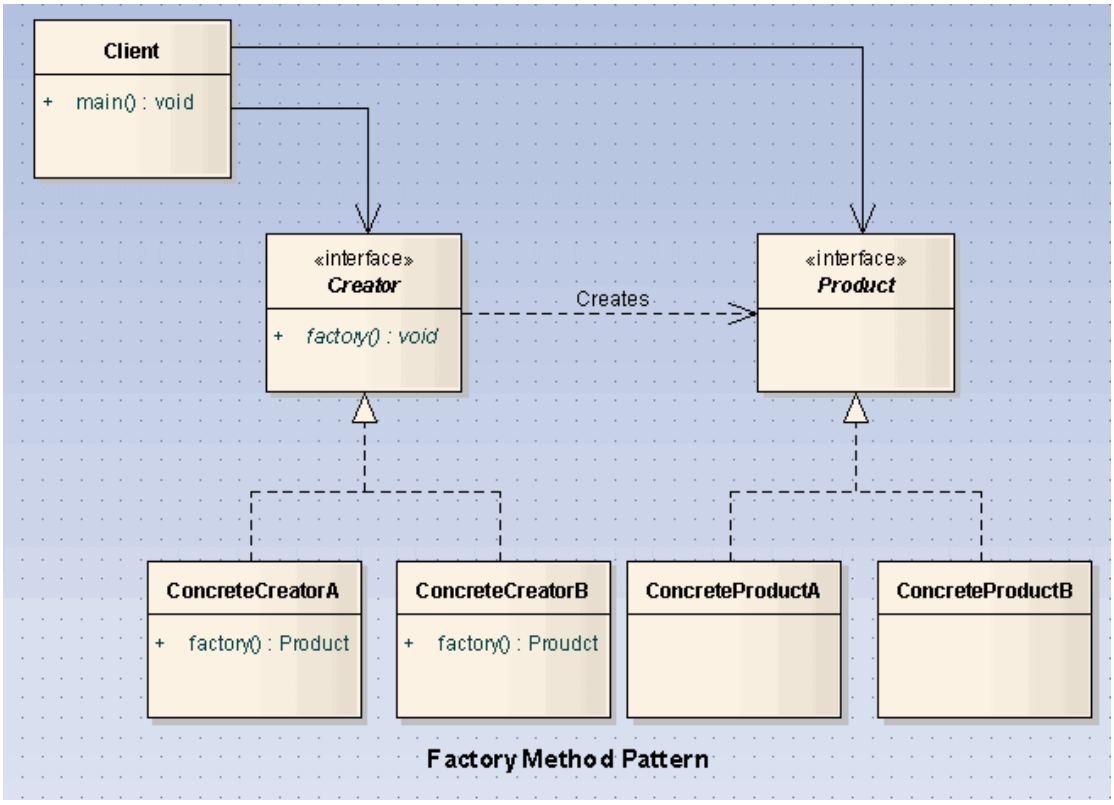
3.2. 工厂方法模式

工厂方法模式是类的创建模式，又叫做虚拟构造子模式或多态性工厂模式。工厂方法模式的用意是定义一个创建产品对象的工厂接口，将实际创建过程推延到子类中去。

优点：工厂方法比简单工厂更进了一步，将工厂角色抽象化产生出具体工厂角色，这样增加新的产品时，只需要增加对就产品类和具体工厂角色。

缺点：工厂方法模式主要解决面对一个产品等级结构的问题。如果面对多个产品等级结构，工厂方法会产生多个对应工厂等级结构。随着产品等级结构的增加，会产生出太多相似的工厂等级结构。显示不是我们想要的结果。

工厂方法模式结构



- 抽象工厂角色：

  这个角色是工厂模式的核心，它与应用程序逻辑无关。具体工厂角色实现这个接口。抽象工厂角色一般由接口或抽象类实现。

- 具体工厂角色：

  实现抽象工厂角色接口，与应用程序逻辑密切相关，由应用程序调用来创建产品对象。一般由具体类来实现。

- 抽象产品角色：

  产品对象共同拥有的接口。一般由接口或抽象类实现。

- 具体产品角色：

  实现抽象产品角色所声明的接口，工厂方法模式所创建的每一个对象都是某个具体产品角色实例。一般由具体类实现。

3.3. 抽象工厂模式

抽象工厂模式是对象的创建模式，是工厂方法模式的进一步推广。

抽象工厂模式是所有形态工厂模式中最为抽象和最具一般性的一种形态。抽

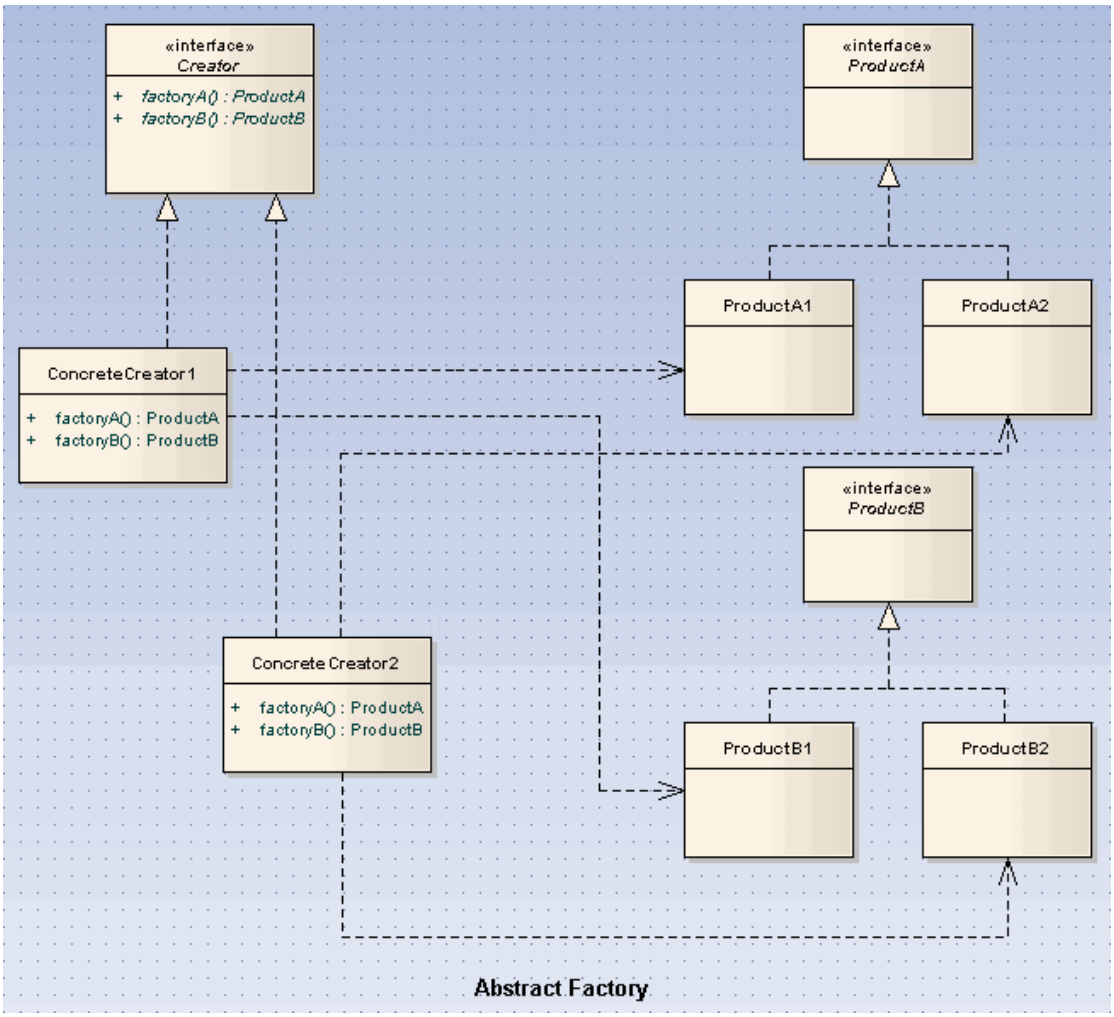
象工厂模式的用意是：向客户端提供一个接口，使得客户端在不必指定产品的具体类型的情况下，创建多个产品族中的产品对象。

抽象工厂模式主要解决面对多个产品等级结构的问题。比工厂方法模式更进一层。

优点：增加新的产品族，只需要增加对就具体工厂类。

缺点：增加新的产品等级结构时，要修改所有工厂类，给每个工厂类都增加一个新的工厂方法。

抽象工厂模式结构



- 抽象工厂角色：

这个角色是模式的核心，它与应用系统逻辑无关。具体工厂角色必须实现这个接口。抽象工厂角色一般由接口或抽象类实现。

- 具体工厂角色：

这个角色在客户端调用下创建产品的实例。含有选择合适的产品对象的逻辑，

与应用系统的商业逻辑密切相关。一般由具体类实现。

- 抽象产品角色：

  模式创建产品对象共同拥有的接口。一般由接口或抽象类实现。

- 具体产品角色：

  模式所创建的任何产品对象都是某一具体产品类的实例。这是客户端最终需要的东西，内部一定充满了应用程序的商业逻辑。一般由具体类实现。

使用情形

1. 一个系统不应该依赖产品类如何被创建、组合和表达的细节，这对于所有形态的工厂模式都是重要的。
2. 系统的产品有多于一个产品族，而系统只消费其中某一族的产品。这是抽象工厂模式的用意。
3. 同属于同一产品族的产品是在一起使用的，这一约束必须在系统的设计中体现出来。
4. 系统提供一个产品类的库，所有的产品以同样的接口出现，从而使客户端不依赖于实现。

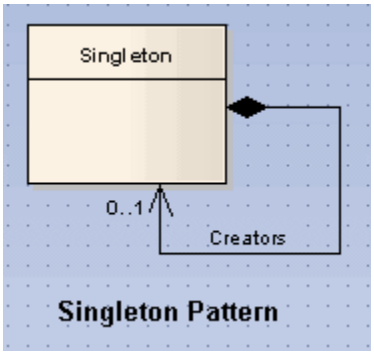
3.4. 单例模式(Singleton)

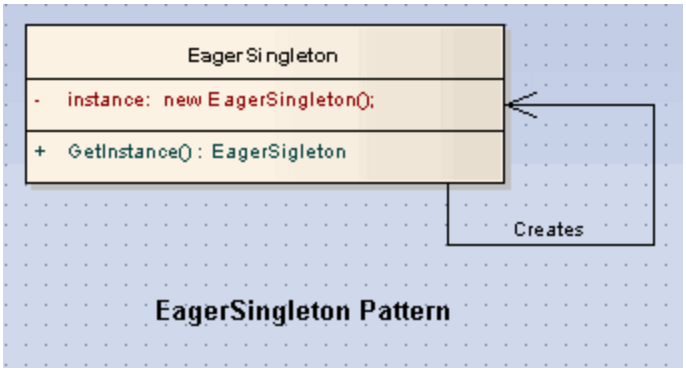
在一个系统要求一个类只能有一个实例的情况下才可以使用单例模式。如果一个类可以有几个实例共存，那么就没有必要使用单例模式。

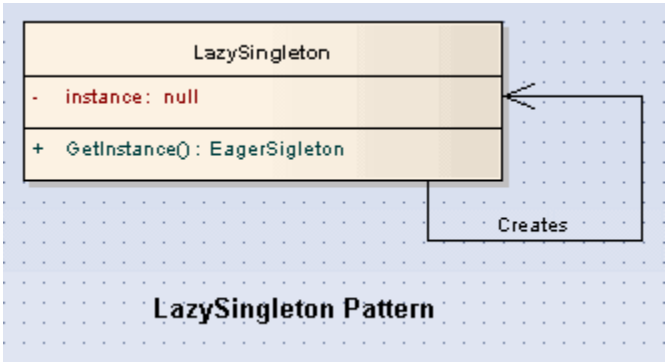
做为对象的创建模式，分为饿汉式单例类和懒汉式单例类。饿汉式式在自己被加载时就将自己实例化。懒汉式单例为在第一次被引用时将自己实例化。从资源利用效率来讲，饿汉式比懒汉式稍差些。从速度和反应时间角度来讲，饿汉式稍好些。懒汉式在初始化时要考虑多纯程问题。

单例模式的要点有三个：

- 一是某个类只能有一个实例；
- 二是它必须自己创建这个实例；
- 三是它必须自行向整个系统提供这个实例。



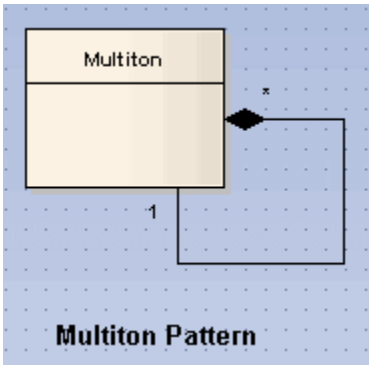




多例模式

所谓多例模式实际上是单例的自然推广。多例模式在应用时可分有上限多例模式和无上限多例模式。作为对象的创建模式，多例模式或多例类有以下特点：

- 可以有多个实例。
- 必须自己创建和管理自己的实例。
- 向处界提供自己的实例。



3.5. 建造模式

建造模式可以将一个产品的内部表象与产品的生成过程分割开来，从而可以使一个建造过程生成具有不同的内部表象的产品对象。

使用建造模式可以使客户端在不需要知道所生成的产品对象有哪些零件，每个产品对应的零件彼此有何不同，是怎么样建造和组合起来的。

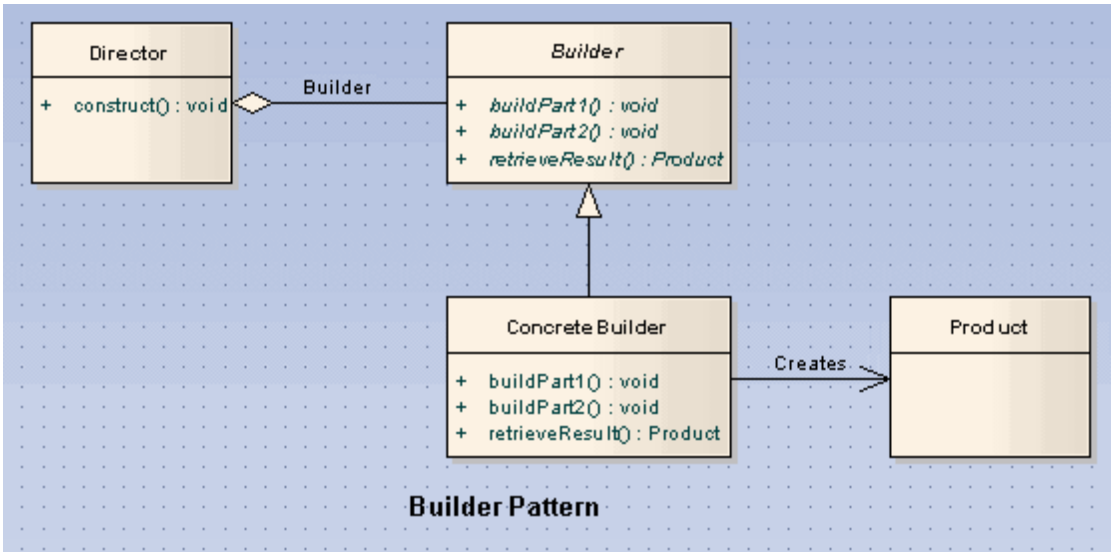
使用情况：

1. 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，也可以仅仅是一个对象的一个组成成分。
2. 产品对象的属性相互依赖，建造模式可以强制实行一种分步骤进行的建造过程。
3. 在对象的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是容易得到的。

达到效果：

1. 使得产品的内部表象可以独立地变化。使用建造模式可以使客户端不必知道产品内部组成的细节。
2. 每一个 Builder 都相对独立，而与其他的 Builder 无关。
3. 模式所建造的最终产品更易于控制。

建造模式的结构



- 抽象建造者角色

用一个抽象接口，来规范出产品对象的各个组成成分的建造。一般此接口对应用程序商业逻辑无关。

- 具体建造者角色

此角色与应用程序逻辑密切相关，在应用程序调用下创建产品的实例。它实现抽象建造者所声明的接口，给出一步一步地完成创建产品实例的操作，完成后提供产品的实例。

- 导演者角色

担任这个角色的类调用具体建造者角色以创建产品对象，但此角色并没有产品类的具体知识，真正拥有产品类具体知识的是具体建造者角色。

- 产品角色

产品角色是要建造的复杂对象。一般一个系统中会有多于一个的产品类，而且这些产品类并不一定有共同的接口，完全可以是不相关联的。

3.6. 原始模型模式

原始模型模式属于对象的创建模式。通过给出一个原型对象来指明所要创建对象的类型，然后通过复制原型对象的方法来创建出更多的同类型的对象。

适用于产品结构可能会有经常性变化的系统。

优点

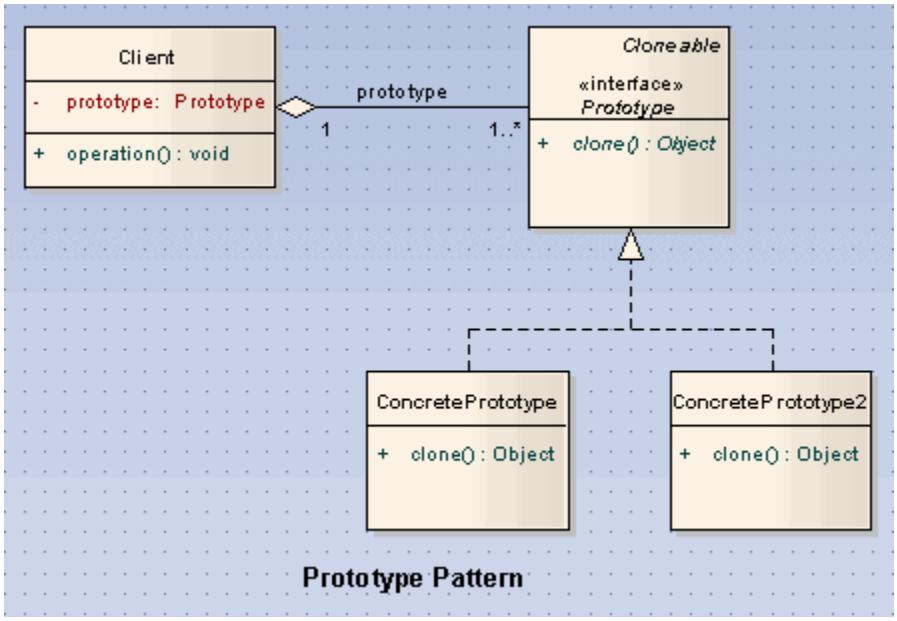
- 允许动态的增加或减少产品类。由于创建产品实例的方法是产品类内部具有

的，因此，增加新产品对整个结构没有影响。

- 产品类不需要非得有任何事先确定的等级结构，因为原始模型模式适用于任何的等级结构。

缺点

每一个类都必须配备一个克隆方法。配备克隆方法需要对类的功能进行通盘考虑，对新类问题不大。但对于已经有的类不太容易，特别是不支持串行化或引用含有循环结构的时候。



原始模型模式有两种表现形式：简单形式和登记形式。

简单形式的原始模型模式

- 客户角色

客户类提出创建对象的请求。

- 抽象原型角色

这是一个抽象角色，一般有接口或抽象类实现。此角色给出所有的具体原型类所需的接口。

- 具体原型角色

被复制的对象，此角色需要实现抽象的原型角色所要求的接口。

登记形式的原始模型模式

- 客户端角色

  提出创建对象的请求。
- 抽象原型角色

  给出所有的具体原型类所需的接口。一般由接口或抽象类实现。
- 具体原型角色

  被复制的对象，需要实现抽象的原型角色所需要的接口。
- 原型管理器角色

  创建具体原型类的对象，并记录每一个被创建的对象。

第四部分、 设计模式结构模式

结构模式描述如何将类或者类的对象结合在一起形成更大的结构。结构模式描述两种不同的东西：类与类的实例。结构模式可以分为：类的结构模式和对象的结构模式两种。

类的结构模式：类的结构模式使用继承来把类、接口等组合在一起，以形成

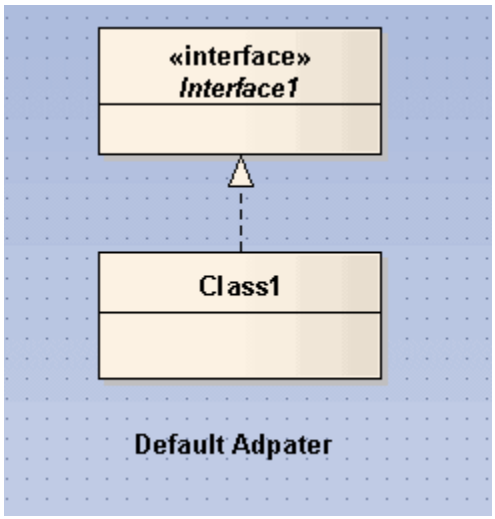
更大的结构。类的结构模式是静态的，比如类形式的适配器模式。

对象的结构模式：对象的结构模式描述怎么样把不同类型的对象组合在一起，以实现新的功能的方法。对象的结构模式是动态的。比如代理人模式。

4.1. 适配器模式

适配器模式(Adapter Pattern)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一起工作。

用意是将接口不同而功能相同或者相近的两个接口加以转化。

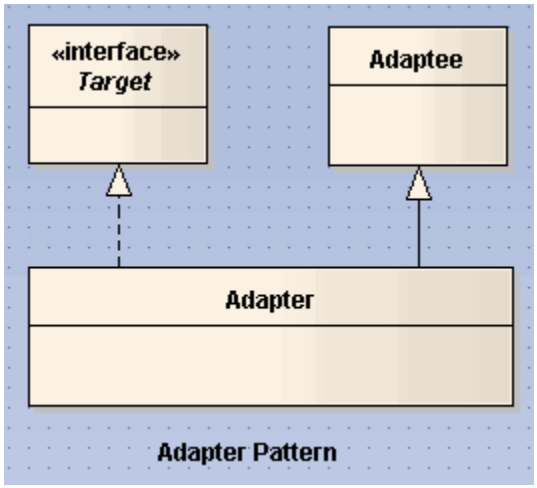


适配器模式有**类的适配器模式**和**对象的适配器**。

类的适配器模式

类的适配器模式是使用继承关系把被适配类的 API 转化成目标类的 API,适配模式是静态的。

结构图及角色



1. 目标角色

   所期待得到的接口。注意类的适配器模式，目标不可以是类。

2. 源角色

   现在需要适配的接口。

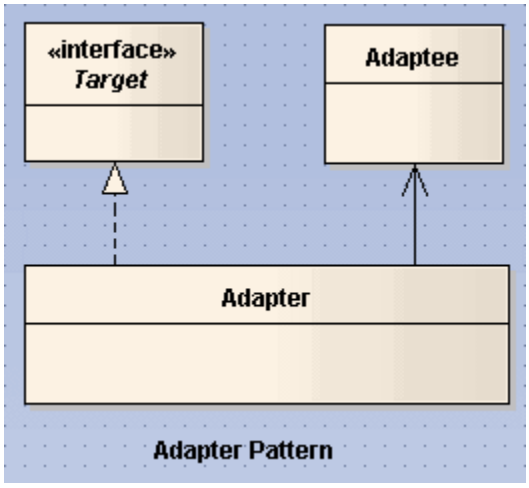
3. 适配器角色

   此角色是模式的核心。适配器角色把源接口转换成目标接口。显然这一角色不可以是接口，而必须是具体类。

对象的适配器模式

与类的适配器模式不同，对象的适配器模式不是使用继承关系连接到目标，而是使用委派关系。这决定这个适配器模式是对象的。

结构图及角色



1. 目标角色

所期待的角色，目标可以是具体的或抽象的类。

2. 源角色

   现有需要适配的接口。

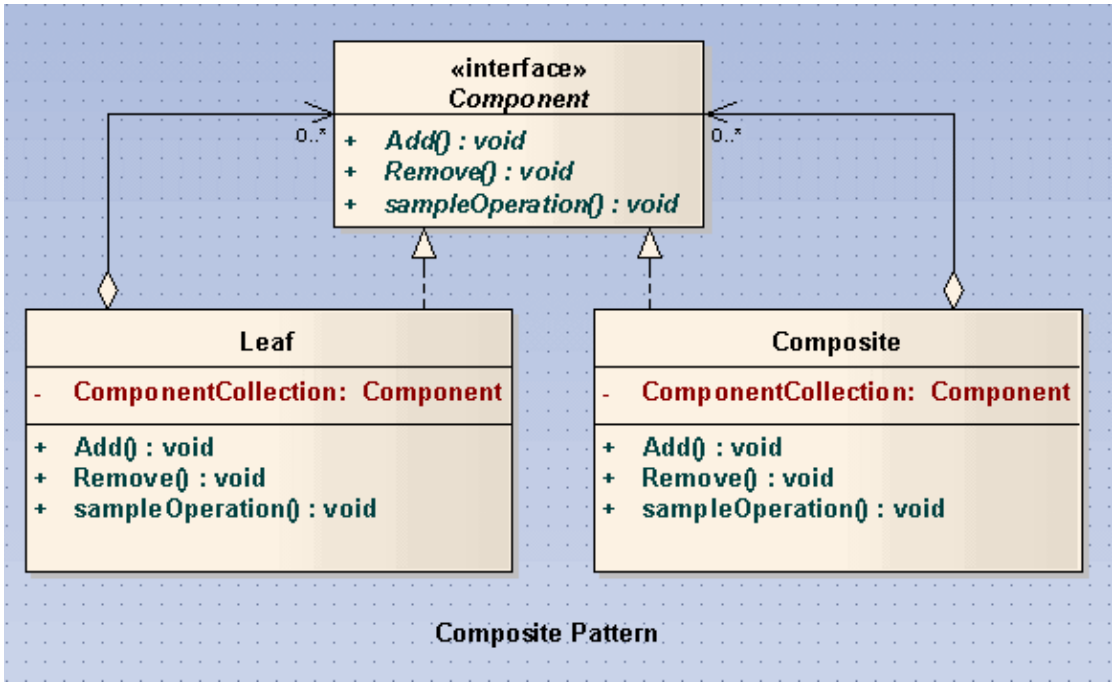
3. 适配器角色

   本模式的核心。把源接口转换成目标接口，显然必须是具体类。

4.2. 合成(Composite)模式

合成模式属于对象的结构模式，又叫部分－整体模式。合成模式将对象组织到树结构中，可以用来描述整体与部分的关系。可以使用客户端将单纯元素与复合元素同等看待。根据所实现接口的区别分为安全式和透明式的合成模式。

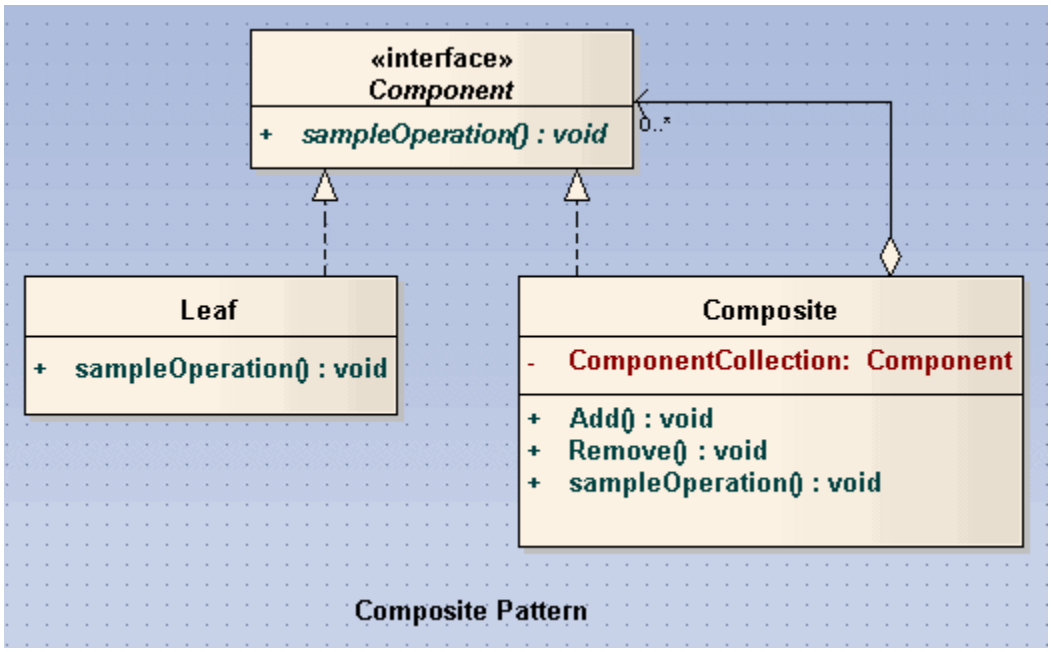
透明式的合成模式



透明式的合成模式抽象构件声明所有的用来管理子类对象的方法，使用客户对树叶类对象和合成类对象感觉没有区别，可以同等对待。缺点是树叶

和树枝在本质是有区别的，有可能发生编译时期正常，但运行时期才会出错。

安全方式的合成模式



安全方式的合成模式在抽象树枝角色里声明所有的用来管理子类对象的方法，这样客户端如果使用枝叶来管理子对象程序在编译时就不会通过。不会出现运时期错误。缺点不够透明，树叶类和树枝类将具有不同的接口。

模式结构图及角色

1. 抽象构件角色

   这是一个抽象角色，它给参加组合的对象规定一个接口。

2. 树叶构件角色

   代表参加组合的树叶对象。一个树叶没有下级的子对象，定义出参加组合的原始对象的行为。

3. 树枝构件角色

   代表参加组合的有子对象的对象，并给出树枝构件对象的行为。

4.3. 装饰(Decorator)模式

装饰模式又名包装模式。

装饰模式以对客户端透明的方式扩展对象的功能，是继承关系的一种替

代方案。装饰模式使用原来被装饰的类的一个子类的实例，把客户端的调用委派到被装饰类。装饰模式的关键在于这种扩展是完全透明的。

使用情况

1. 需要扩展一个类的功能，或给一个类增加附加责任。
2. 需要动态地给一个对象增加功能，这些功能可以再动态地撤销。
3. 需要增加由一些基本功能的排列组合而产生的非常大量的功能，从而使继承关系变得不现实。

优点

装饰模式与继承关系的目的都是扩展对象的功能，但是装饰模式可以提供比继承更多的灵活性。

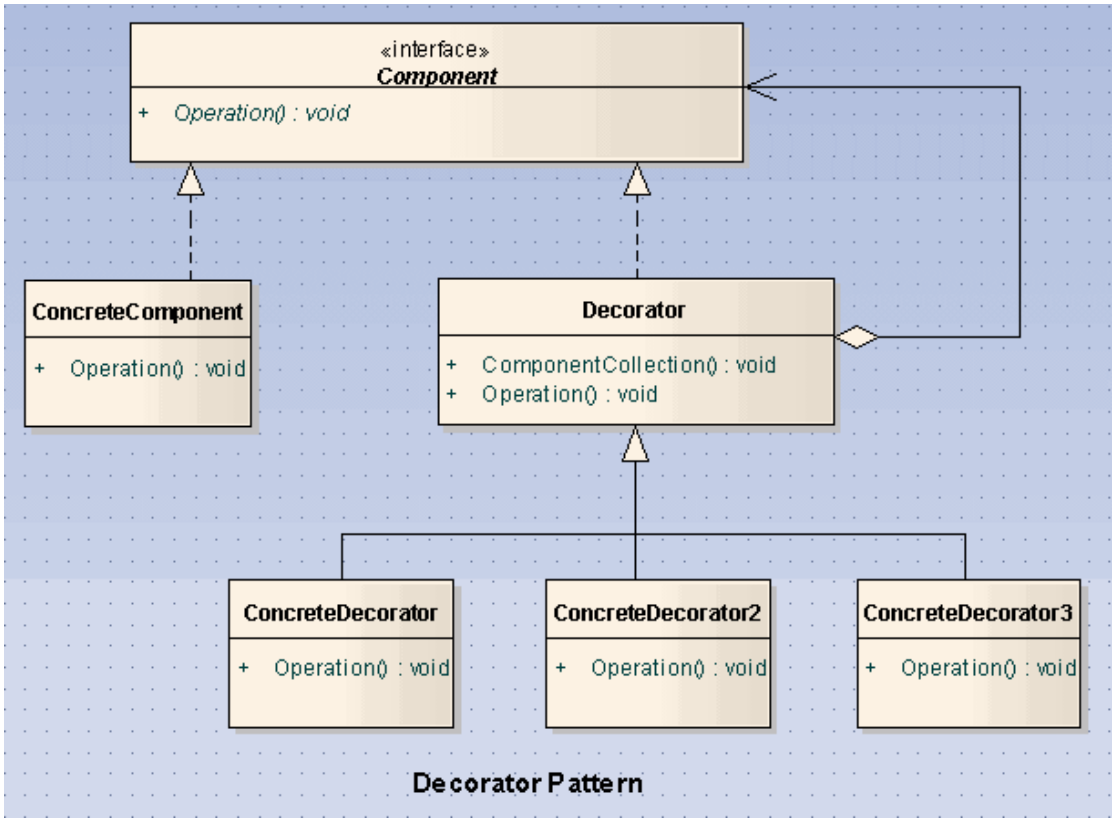
通过使用不同的具体装饰类以及这些装饰类的排列组合，设计师可以创造出很多不同行为的组合。

缺点

比继承关系需要较少数目的类，但会产生更多的对象，使得查错变得困难，特别是这些对象看上去都很像。

装饰模式和适配器模式都有一个别名，即包装模式，但是这两个模式是很不一样的。适配器模式的用意是要改变所考虑的对象的接口而不一定改变对象的性能，而装饰模式的用意是要保持接口，从而增加所考虑对象的性能。

结构及角色



- 抽象构件角色

  给出一个抽象接口，以规范准备接收附加责任的对象。

- 具体构件角色

  定义一个将要接收附加责任的对象。

- 装饰角色

  持有一个构件对象的实例，并定义一个与抽象构件接口一致的接口。

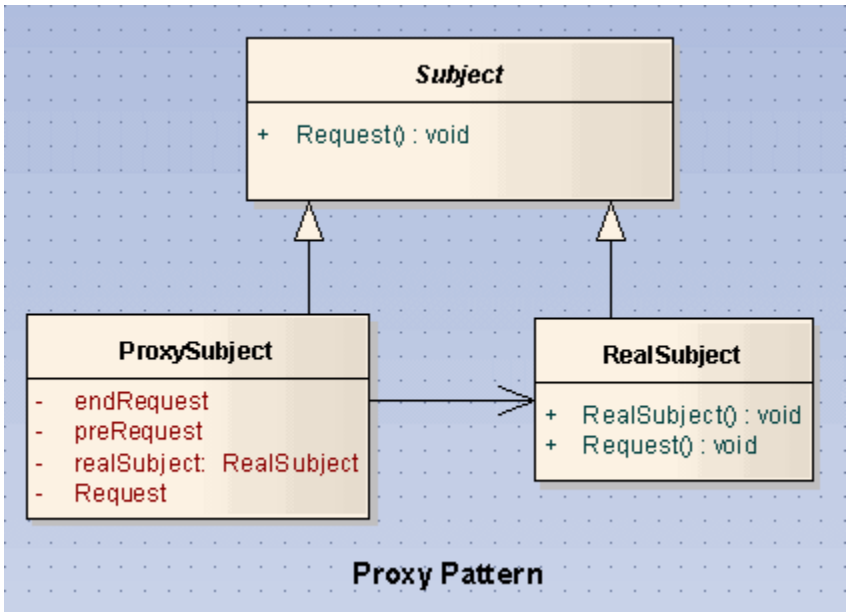
- 具体装饰角色

  负责给构件对象“贴上”附加的责任。

4.4. 代理(Proxy)模式

代理模式是对象的结构模式。代理模式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。顾名思义，所谓代理就是一个人或者一个机构代表另一个人或者一个机构采取行动。

代理结构及角色



- 抽象主题角色

  声明了真实主题和代理主题的共同接口，这样一来任何可以使用真实主题的地方都可以使用代理主题。

- 代理主题角色

  代理主题角色内部含有对真实主题的引用，从而可以在任何时候操作真实主题对象。通常在将客户端的请求传给真实主题之前或之后，都要执行某个操作，而不是单纯的将调用传递给真实主题对象。

- 真实主题角色

  定义了代理角色所代表的真实对象。

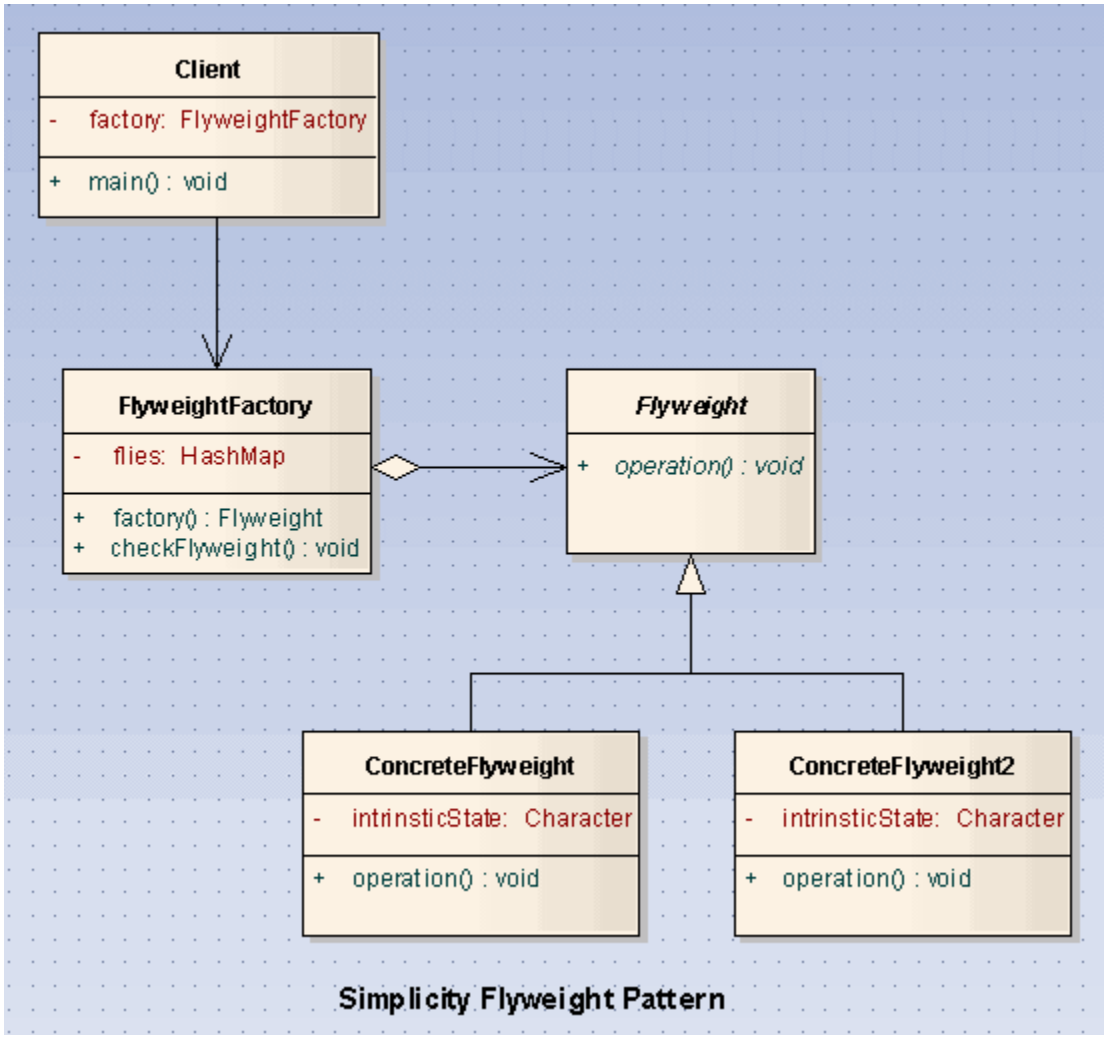
4.5. 享元(Flyweight)模式

享元模式是对象的结构模式。享元模式以共享的方式高效地支持大量的细粒度对象。根据享元对象的内部表象，享元模式可以分成单纯享元模式和复合享元模式两种形式。享元对象不一定非要是不变对象。一个享元对象之所以能够共享，是因为它只含有可以共享的状态，而没有不可以共享的状态，这是应用享元模式的前提。享元工厂实际上提供了一个缓存机制，判断对象是否存在然后创建或直接返回对象。

优缺点

使用享元模式可以大幅度降低内存中对象的数量。但是为了可以享元，需要将一些状态外部化， 这使得程序逻辑变的复杂，而且读取外部状态使得运行时间稍微变长。

单纯享元模式结构



- 抽象享元角色

  所有具体享元角色的基类。

- 具体享元角色

  实现抽象享元角色所规定的接口。如果有内蕴状态的话，必须为内蕴状态提供存储空间。内蕴状态必须与对象所处的周围环境无关，从而使得享元对象可以在系统内共享。

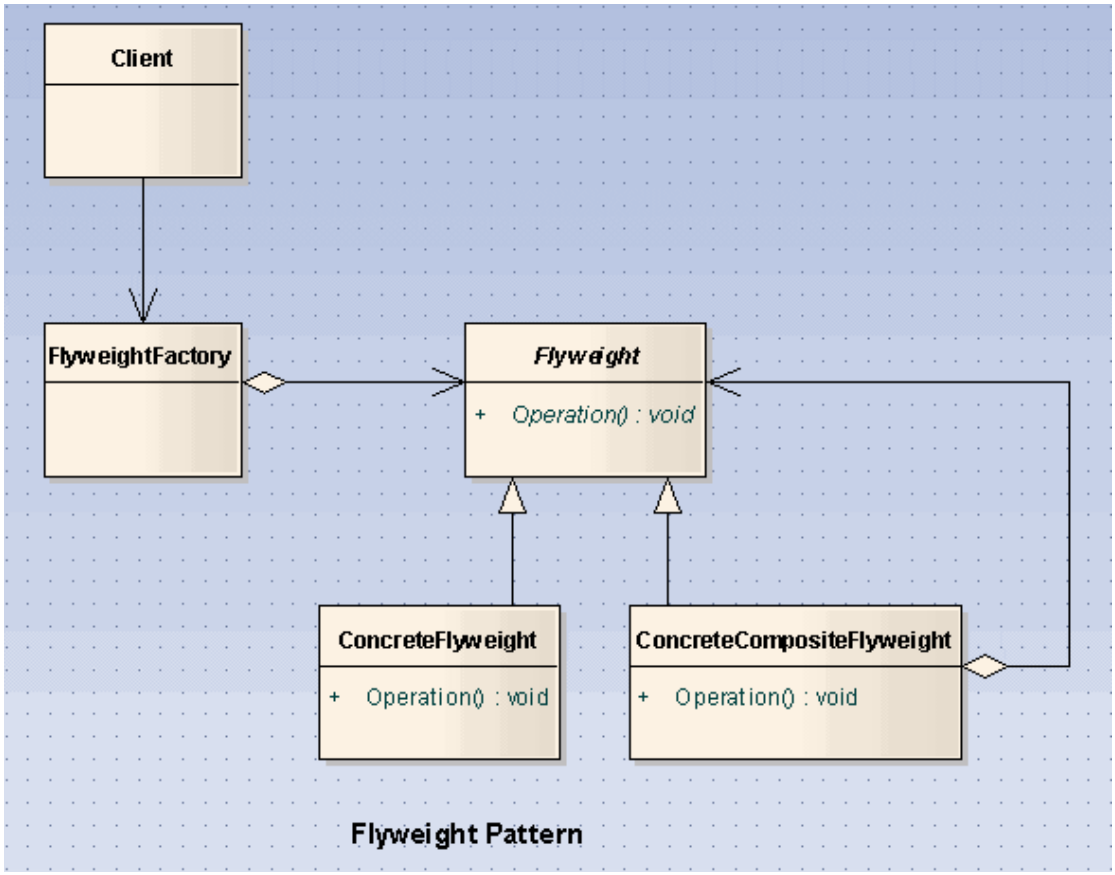
- 享元工厂角色

负责创建和管理享元角色。保证享元对象可以被适当的共享，检查系统中是否有复合要求的对象。如果要直接近回，否则创建并返回。

- 客户端角色

本角色需要维护一个对所有享元对象的引用，自行存储所有享元对象的外蕴状态。

复合享元模式结构



- 抽象享元角色

所有具体享元角色的基类。

- 具体享元角色

实现抽象享元角色所规定的接口。如果有内蕴状态的话，必须为内蕴状态提供存储空间。内蕴状态必须与对象所处的周围环境无关，从而使得享元对象可以在系统内共享。又称单纯具体享元角色。

- 复合享元角色

此角色所代表的对象是不可以共享的，但可分解成多个本身可以共享的单纯享元对象。又称不可共享的享元对象。

- 享元工厂角色

  负责创建和管理享元角色。保证享元对象可以被适当的共享，检查系统中是否有复合要求的对象。如果要直接近回，否则创建并返回。

- 客户端角色

  本角色需要自行存储所有享元对象的外蕴状态。

4.6. 门面(Facade)模式

门面模式是对象的结构模式。

门面模式要求一个子系统的外部与其内部的通信必须通过一个统一的门面对象进行。门面模式提供了一个高层次的接口，将客户端和子系统应用程序逻辑隔离，使得子系统更易于使用。对于服务层逻辑复杂的系统更为有效。

门面模式的结构

- 门面角色

  此角色知晓相关 (一个或多个) 子系统的功能和责任。将所有从客户端的请求委派到相关的子系统去。

- 子系统角色

  每个子系统都可以被客户端直接调用，或者被门面角色调用。子系统并不知道门面的存在，对于子系统而言，门面仅仅是另外一个客户端。

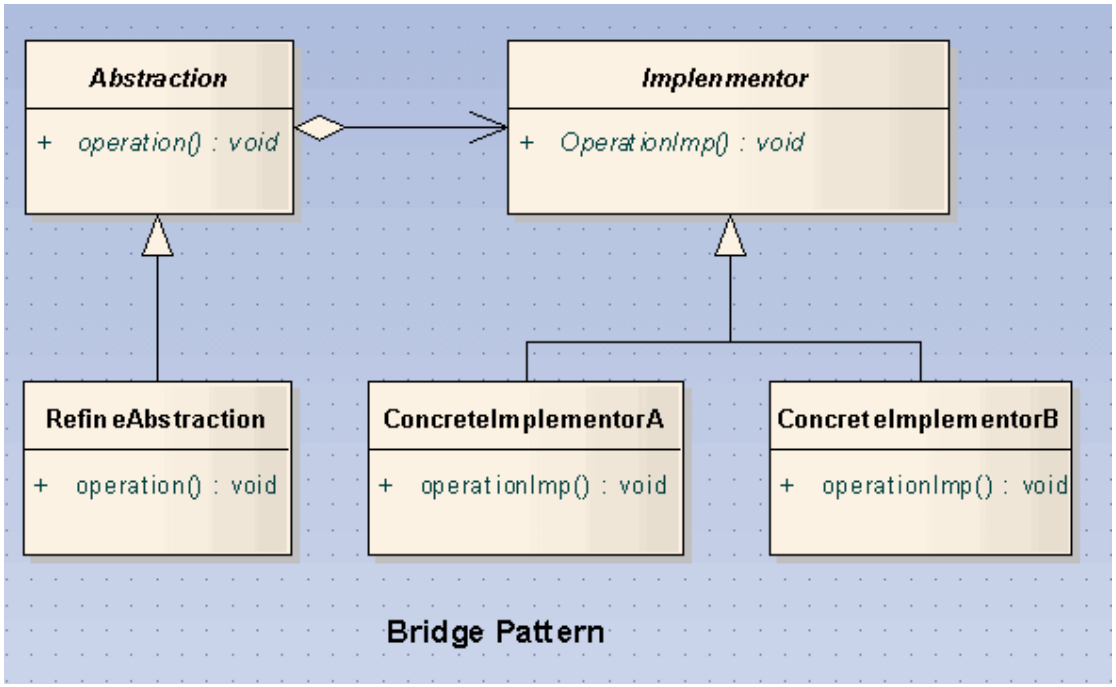
4.7. 桥梁(Bridge)模式

桥梁模式是对象的结构模式，又称为柄体模式或接口模式。(ABeen 更喜欢柄体模式这个名字，感觉更形象的表达其意图)。抽象化角化就像是一个水杯的手柄，而实现化角色和具体实现化角色就像是水杯的杯身。手柄控制杯身，由此得名“柄体模式”。

桥梁模式的用意是将抽象化和实现化脱藕，使得二者可以独立地变化。

桥梁模式所谓的脱藕，实际上指在一个软件系统的抽象化和实现化之间使用组合/聚合关系而不是继承关系，从而使这两者可以相对独立的变化。桥梁模式多用在抽象部分和实现部分都多变的情况下。桥梁模式很好的符合开闭原则和组合复用原则。

桥梁模式结构



- 抽象化角色

  抽象化给出的定义，并保存一个对实现化对象的引用。

- 修正抽象化角色

  扩展抽象化角色，改变和修正父类对抽象化的定义。

- 实现化角色

  给出实现化角色的接口，必须指出的是这个接口不一定和抽象化角色的接口定义相同，这两个接口可以非常不一样。实现化应当只给出底层操作接口，而抽象化角色应当只给出基于底层操作的更高一层的操作。

- 具体实现化角色

  给出实现化角色接品的具体实现。

第五部分、 设计模式行为模式

行为模式主要是责任和算法的抽象化。行为模式不仅仅是关于类和对象的，而且是关于它们之间的相互作用的。

行为模式分为类的行为模式和对象的行为模式两种。

类的行为模式

类的行为模式使用继承关系在几个类之间分配行为。

对象的行为模式

对象的行为模式则使用对象的聚合来分配行为。

5.1. -不变模式

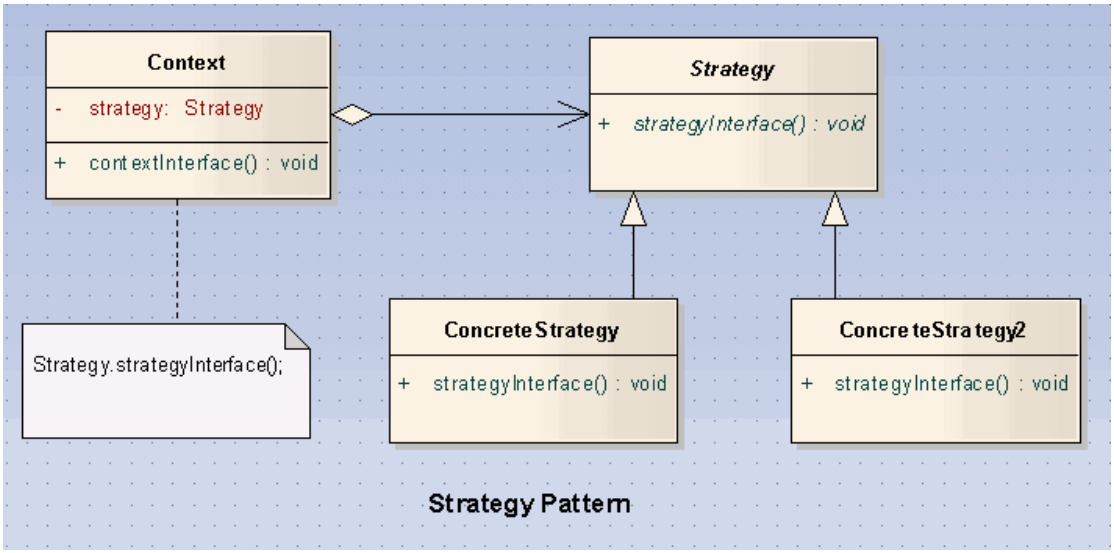
一个对象的状态在对象被创建之后就不再变化，这就是所谓的不变模式。采用不变模式的类一旦初始化后，其状态就不再改变了。这意味着不变模式不能提供任何的修改内部属性的方法，一旦构造函数构造完后其内部状态就保持不变。又分为强不变模式和弱不变模式，区别在于其子类是否也是不变的，准确的来说是其子类也不能修改父不变类的状态。当然为了方便我们可以索性设置为 sealed。

不变类的好处在于允许任何多的对象共享，不需要在多线程访问的时候进行同步。缺点在于一旦要修改不变对象，只有重新创建一个新的实例。需要频繁修改的对象不能使用不变模式。享元模式中的享元对象多为不变类。

5.2. 策略模式

策略模式属于对象的行为模式。其用意是针对一组算法，将每一个算法封装到具有共同接口的独立类中，从而使用得它们可以相互替换。策略模式使得算法可以在不影响到客户端的情况下发生变化。

结构及角色



- 环境角色(Context)

  持有一个抽象策略角色 Strategy 类的引用。

- 抽象策略(Strategy)角色

  这是一个抽象角色，通常由一个接口或抽象类实现。此角色给出所有的具体策略类所需的接口。

- 具体策略(Concrete Strategy)角色

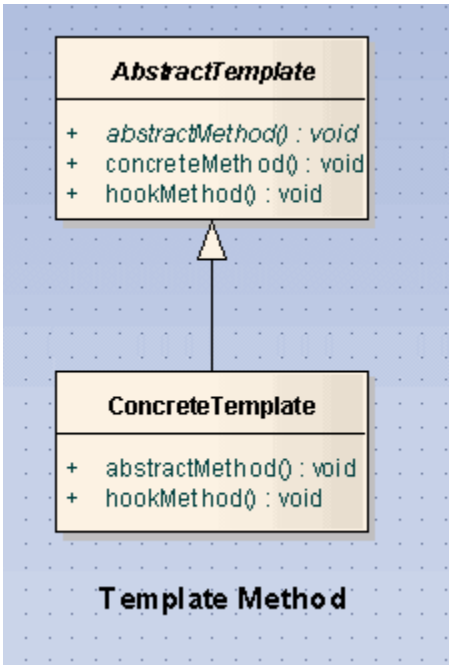
  包装相关算法或行为。

5.3. 模板方法模式

模版方法模式是类的行为模式。

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。这就是模板方法的用意。

结构及角色



- 抽象模板(Abstract Template)角色

  定义一个或多个抽象操作，以便让子类实现。这些抽象操作叫做基本操作，它们是一个顶级逻辑的组成步骤。

  定义并实现一个模板方法。这个模板方法一般是一个具体方法，它给出了一个顶级逻辑骨架，而逻辑的组成步骤在相就的操作中，推迟到子类实现。顶级逻辑也有可能调用一些具体方法。

- 具体模板(Concrete Template)角色

  实现父类所定义的一个或多个抽象方法，它们是一个顶级逻辑的组成步骤。

  每一个抽象模板角色都可以有任意多个具体模板角色与之对应，而每一个具体模板角色都可以给出这些抽象方法的不同实现，从而使得顶级逻辑的实现各不相同。

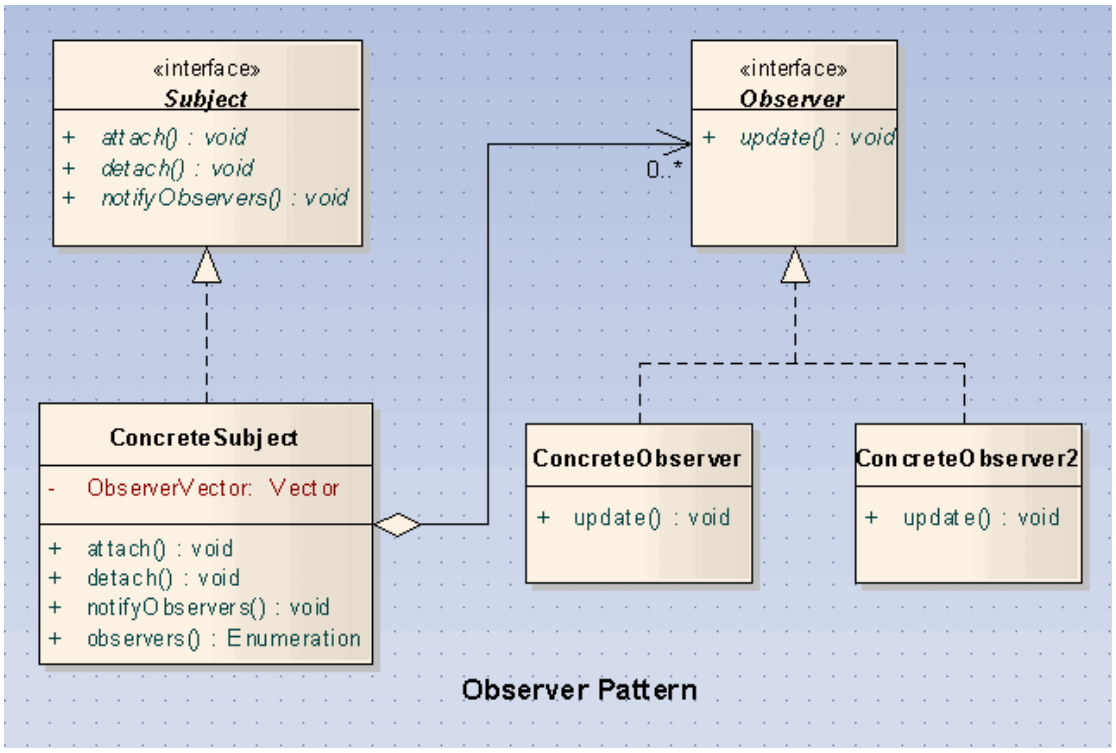
5.4. 观察者模式

观察者模式对象的行为模式，又叫做发布－订阅模式(Publish/Subscribe)、模型－视图(Model/View)模式、源－监听器(Source/Listener)模式或从属者(Dependents)模式。

观察者模式定义了一种一对多的依赖关系，让多个观察者对象同时监听

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，使它们自动更新自己。

结构及角色



- 抽象主题(Subject)角色

  主题把所有对观察者对象的引用保存到一个聚集里，每个主题都可以有任何数量的观察者。抽象主题提供一个接口，可以增加和删除观察者对象，主题角色又叫做抽象被观察者角色，一般由一个抽象类或接口实现。

- 抽象观察者(Observer)角色

  为所有具体观察者定义一个接口，在得到主题的通知进更新自己。这个接口叫作更新接口。此角色一般由一个抽象类或接口实现。

- 具体主题(Concrete Subject)角色

  将有关状态存入具体观察者对象，在具体主题的内部状态发生变化时，通知所有登记过的观察者对象。以叫做具体被观察者角色。一般由一个具体子类实现。

- 具体观察者(Concrete Observer)角色

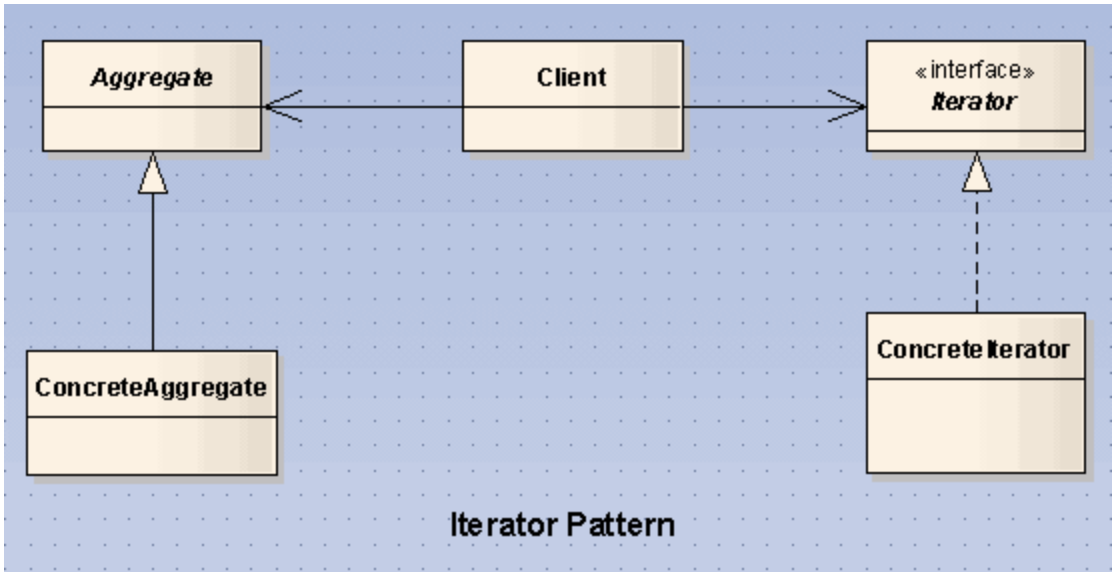
  存储与主题的状态自恰的状态。具体观察者角色实现抽象观察者所声明的更新接口，以便使本身状态与主题状态相协调。如果需要，可以保存一个指向

具体主题对象的引用。一般由一个具体类来实现。

5.5. 迭代子模式

迭代子(Iterator)模式是对象的行为模式，又叫做游标(Cursor)模式。迭代子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象。

迭代子模式的一般性结构



1. 抽象迭代子(Iterator)角色

   定义出遍历元素所需的接口。

2. 具体迭代子(Concrete Iterator)角色

   此角色实现抽象迭代子角色所定义的接口，并保持迭代过程的游标位置。

3. 抽象聚集(Aggregate)角色

   此抽象角色给出创建迭代子对象的接口。

4. 具体聚集(Concrete Aggregate)角色

   实现抽象聚集角色所定义的接口，返回一个合适的具体迭代子实例。

5. 客户端(Client)角色

   持有对聚集及其迭代子对象的引用，调用迭代子对象的迭代接口，也有可能通过迭代子操作聚集元素的增加和删除。

宽接口和窄接口

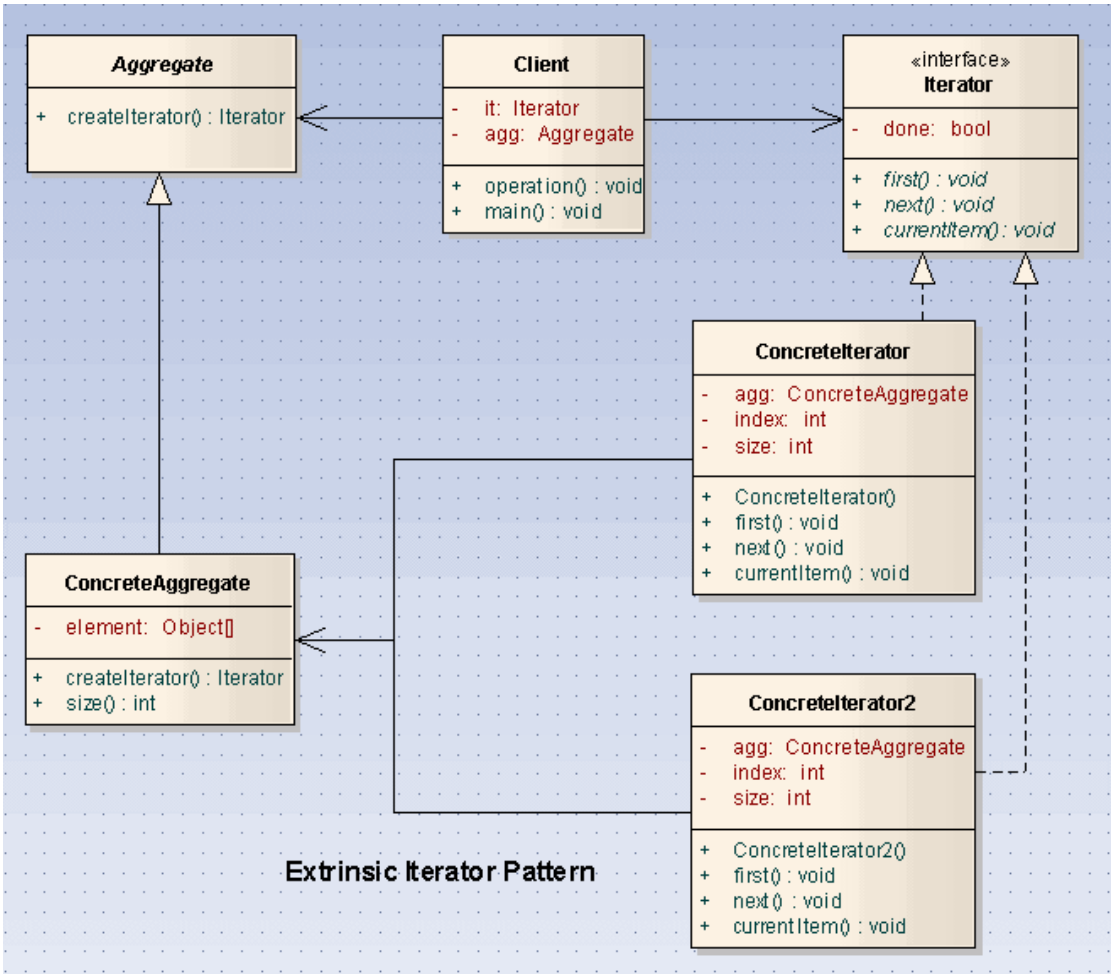
如果一个聚焦的接口提供了可以用来修改聚集元素的方法，这个接口就是所谓的宽接口；

如果一个聚集的接口没有提供修改聚集元素的方法，这样的接口就是所谓的窄接口。

白箱聚集

如果聚焦对所有对象提供宽接口的话，这样会破坏聚焦对象的封装，这种聚集叫做白箱聚集。白箱聚焦向外界提供同样的宽接口。

由于聚集自己实现迭代逻辑，并向外部提供适当的接口，使得迭代可以从外部控制聚焦的迭代过程。这样一样迭代子所控制的仅仅是一个游标而已。这种迭代子叫做游标迭代子。由于迭代子是在聚焦的结构之外的，因此这样的迭代子以叫做外禀迭代子(Extrinsic Iterator)。

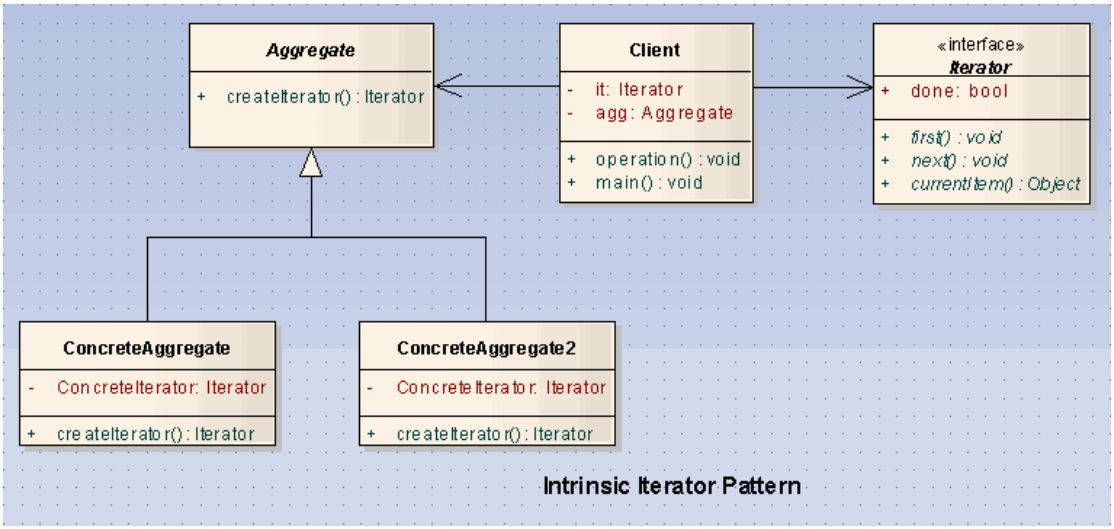


黑箱聚集

聚焦对象为迭代子对象提供一个宽接口，而为其他对象提供一个窄接口。

换言之，聚焦对象的内部结构应当对迭代子对象适当公开，以便迭代子对象能够对聚集对象有足够的了解，从而可以进行迭代操作。但是，聚集对象应当避免向其他对象提供这些方法，因为其他对象应当经过迭代子对象进行这些工作，而不是直接操控聚集对象。这样同时保证聚集对象的封装和迭代功能的方案叫做黑箱实现方案。

由于迭代子是聚集的内部类，迭代子可以自由访问聚集元素，所以迭代子可以自行迭代功能并控制对聚集的迭代逻辑。由于迭代子是在聚集的结构之内定义的，因此这样的迭代子以叫做内禀迭代子(Intrinsic Iterator)。



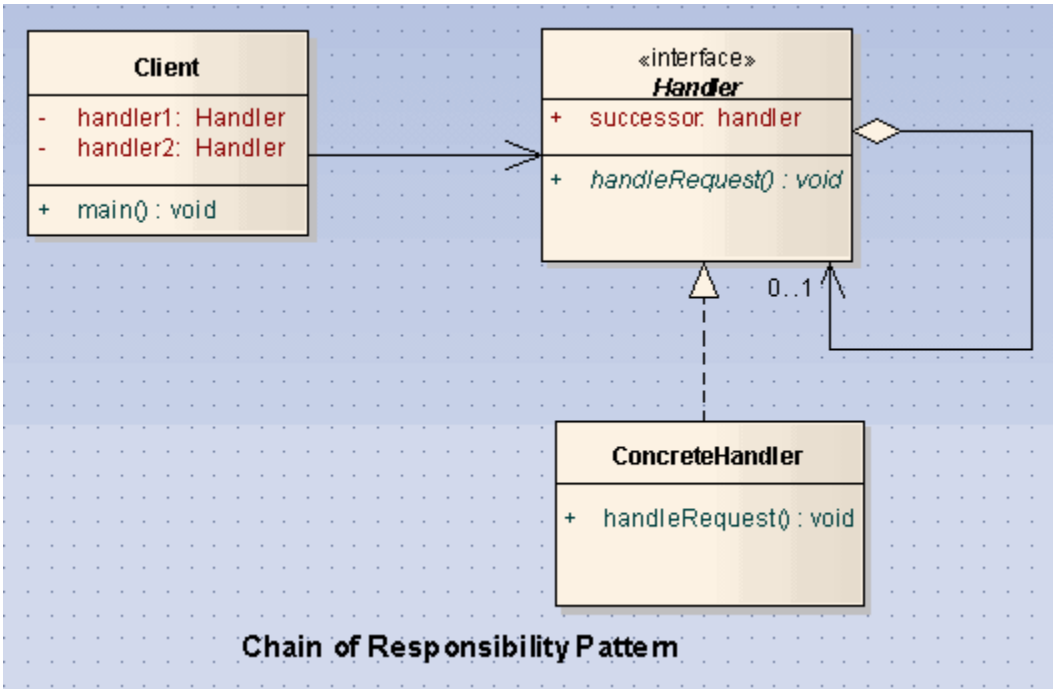
5.6. 责任链模式

责任链模式是对象的行为模式。

在责任链模式里，很多对象由每一个对象对其下家的引用而连接起来开成一条链。请求在这个链上传递，直到链上的某个对象决定处理些请求。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责任。

纯的责任链要求只有一个处理者处理，但是在现实中不太可能。往往是某个处理者处理部分，然后多个处理者共同完成处理。

结构及角色



- 抽象处理者(Handler)角色

  定义出一个处理请求的接口。如果需要，接口可以定义出一个方法，以设定和返回对下家的引用。此角色通常由抽象类或接口实现。

- 具体处理者(Concrete　Handler)角色

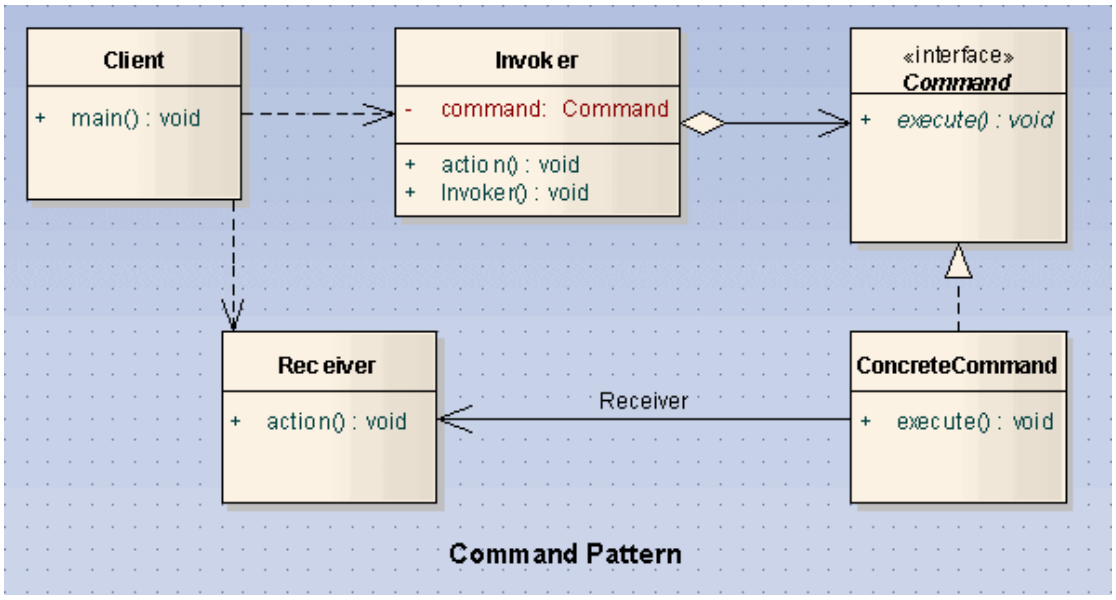
  具体处理者接到请求后，可以选择将请求处理掉，或者将请求传给下家。由于具体处理者持有对下家的引用，因此，如果需要，具体处理者可以访问下家。

5.7. 命令模式

命令模式是对象的行为模式。以称为行动模式或交易模式。

命令模式把一个请求或者操作封装到一个对象中，命令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。

结构及角色



- 客户(Client)角色

  创建一个具体命令对象并确定其接收者。

- 抽象命令(Command)角色

  声明所有具体命令角色必须实现的接口，一般由抽角类或接口实现。

- 具体命令(Concrete Command)角色

  实现抽象命令角色所声明的接口。并定义一个接收者和行为之间的弱耦合，负责调用接收者的相应操作。

- 请求者(Invoker)角色

  负责调用命令对象执行请求，相关的方法叫做行动方法。

- 接收者(Receiver)角色

  负责具体实施和执行一个请求。任何一个类都可以成为接收者，实施和执行请求的方法叫做行动方法。

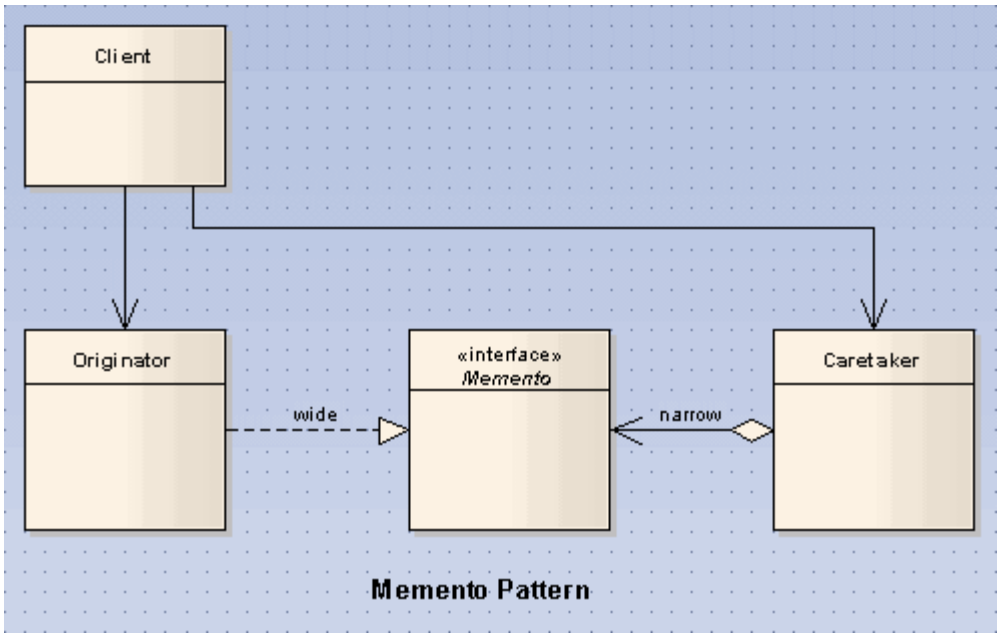
5.8. 备忘录模式

备忘录(Memento Pattern)模式以叫做快照模式或 Token 模式，是对象的行为模式。

备忘录对象是一个用来存储另外一个对象内部状态的快照的对象。其用意是在不破坏封装的条件下，将一个对象的状态捕捉住，并外部化存储起来，从而可

以在将来合适的时候把这个对象还原到存储起来的状态。

结构及角色



- 备忘录(Memento)角色

  将发起人对象的内部状态存储起来。备忘录可以根据发起人对象的判断来决定存储多少发起人对象的内部状态。

  备忘录可以保护其内容不被发起人对象之外的任何对象所读取。

- 发起人(Originator)角色

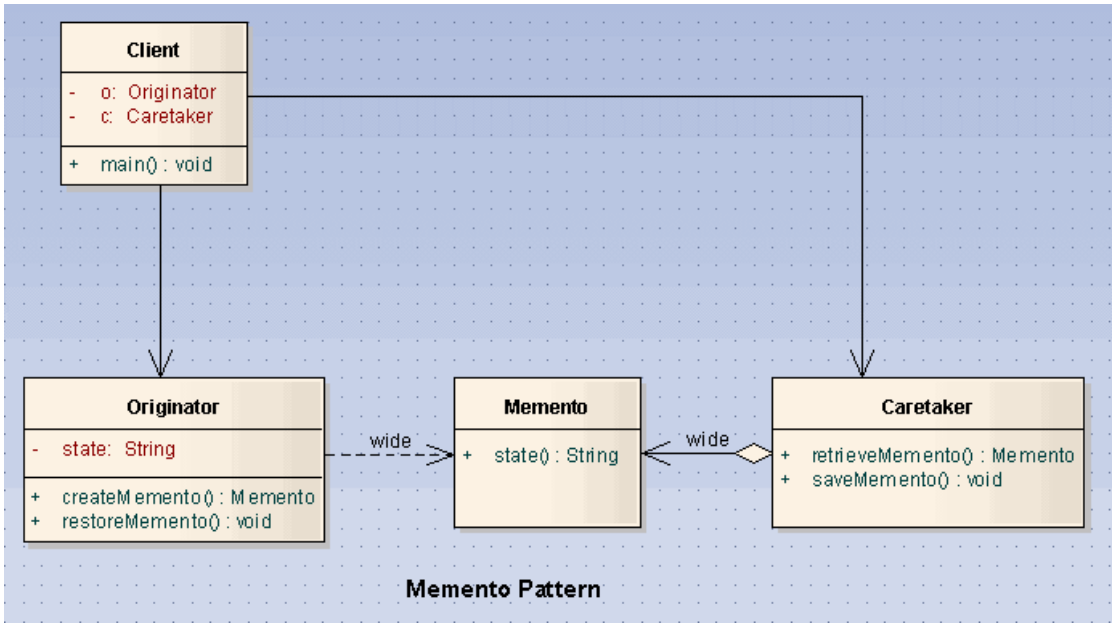
  创建一个含有当前内部状态的备忘录对象。

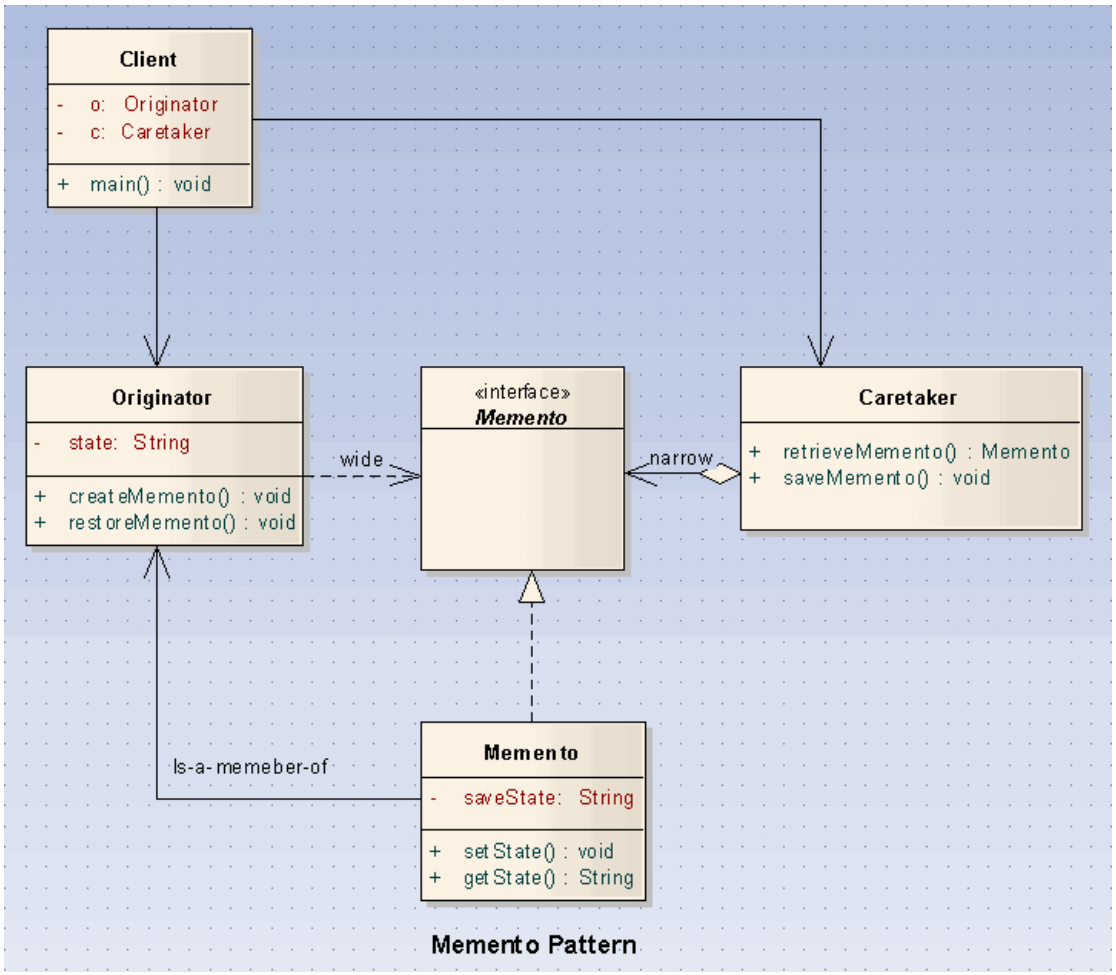
  使用备忘录对象来存储其内部状态。

- 负责人(Caretaker)角色

  负责保存备忘录对象。

  不检查备忘录对象的内容。


Client
- o: Originator
- c: Caretaker
+ main() : void
Originator
- state: String
+ createMemento() : Memento
+ restoreMemento() : void
wide
Memento
+ state() : String
wide
Caretaker
+ retrieveMemento() : Memento
+ saveMemento() : void
Memento Pattern

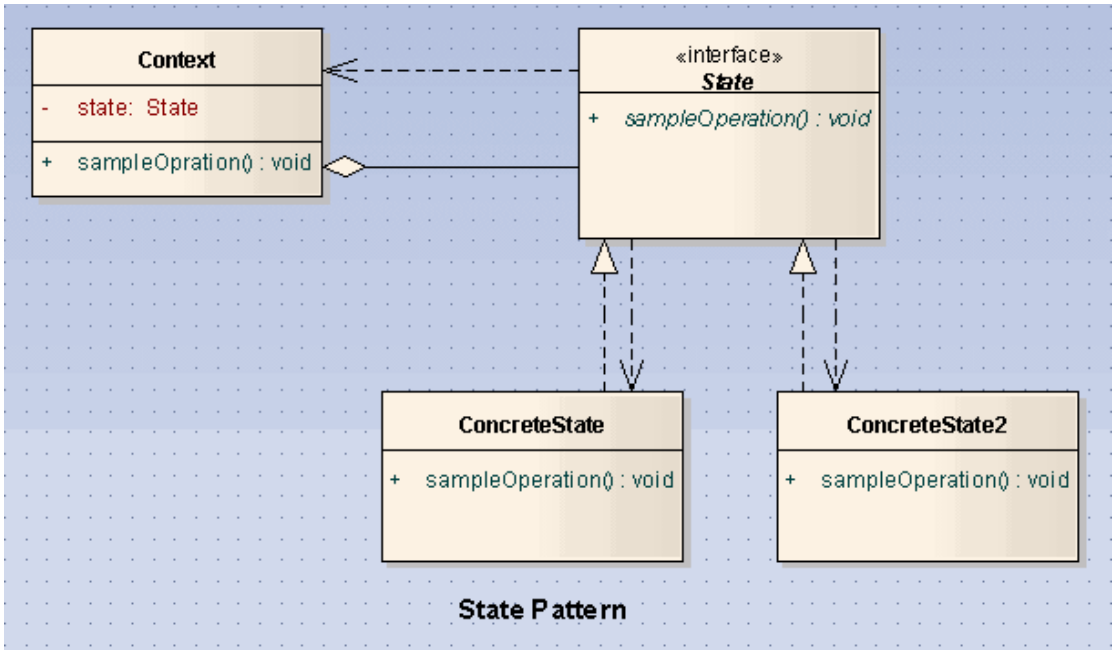

Client
- o: Originator
- c: Caretaker
+ main() : void
Originator
- state: String
+ createMemento() : void
+ restoreMemento() : void
wide
«interface»
Memento
narrow
Caretaker
+ retrieveMemento() : Memento
+ saveMemento() : void
ls-a-memeber-of
Memento
- saveState: String
+ setState() : void
+ getState() : String
Memento Pattern

5.9. 状态模式

状态模式以称状态对象模式，是对象的行为模式。

状态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改变了它的类一样。状态模式把所研究的对象的行为包装在不同的状态对象里，每一个状态对象都属于一个抽象状态类的一个子类。

结构及角色



- 抽象状态(State)角色

  定义一个接口，用以封装环境对象的一个待定状态所对应的行为。

- 具体状态(Concrete State)角色

  每一个具体状态都实现了环境的一个状态所对应的行为。

- 环境(Context)角色

  定义客户端所感兴趣的接口，并且保留一个具体状态类的实例。这个具体状态类的实例给出环境对象的现有状态。

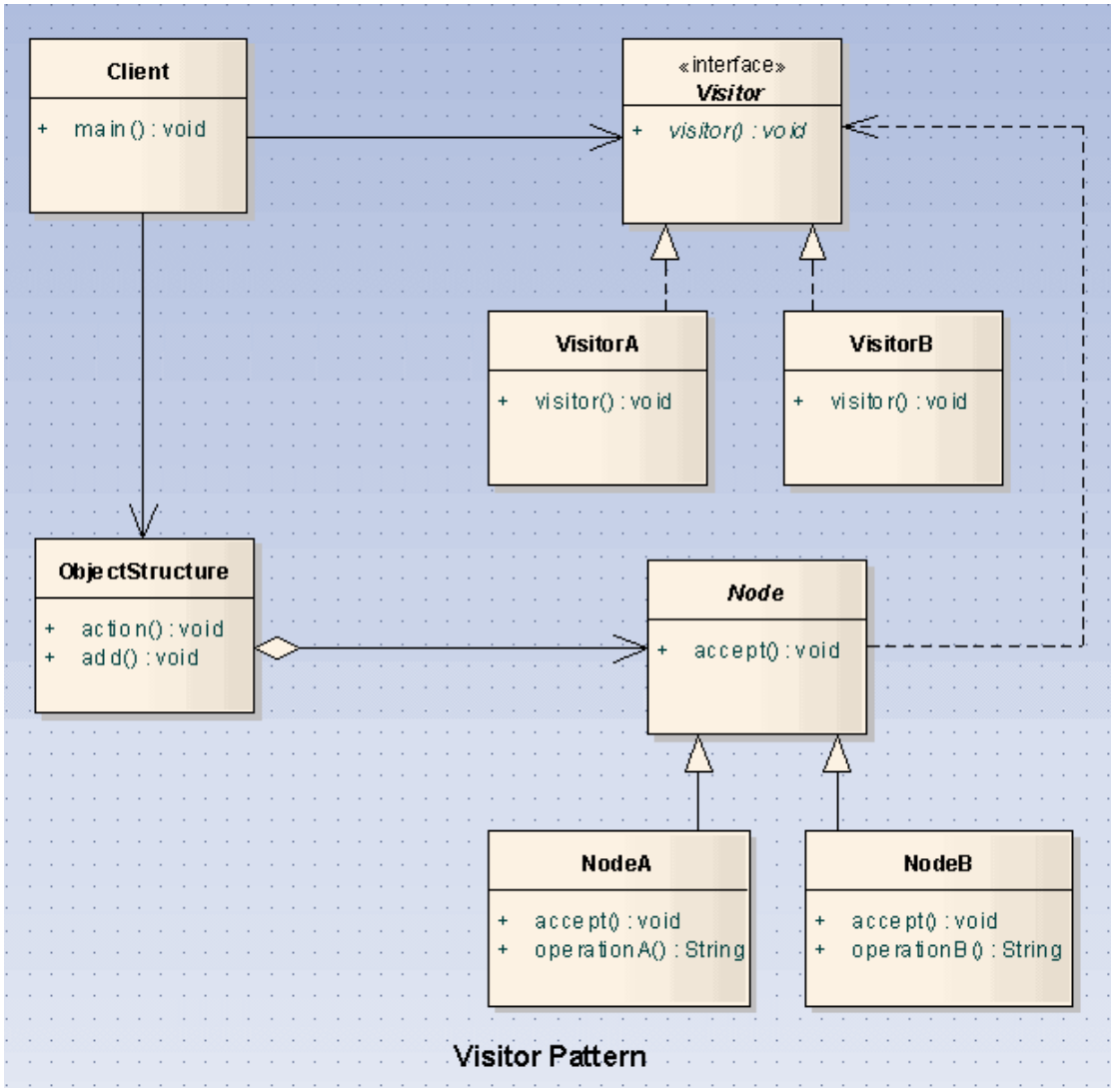
5.10. 访问者模式

访问者模式是对象的行为模式。目的是封装一些施加于某种数据结构元素之上的操作。一旦这些操作需要修改的话，接受这个操作的数据结构则可以保有持不变。

访问者模式适用于数据结构相对未定的系统，它把数据结构和作用于结构上

的操作之间的耦合解脱开，使得操作集合可以相对自由地演化。

结构及角色



- 抽象访问者(Visitor)角色

  声明一个或多个访问操作，形成所有的具体元素角色必须实现的接口。

- 具体访问者(Concrete Visitor)角色

  实现抽象访问者角色所声明的接口，也就是抽象访问者所声明的各个访问操作。

- 抽象节点(Node)角色

  声明一个接受操作，接受一个访问者对象作为一个参量。

- 具体节点(Concrete Node)角色

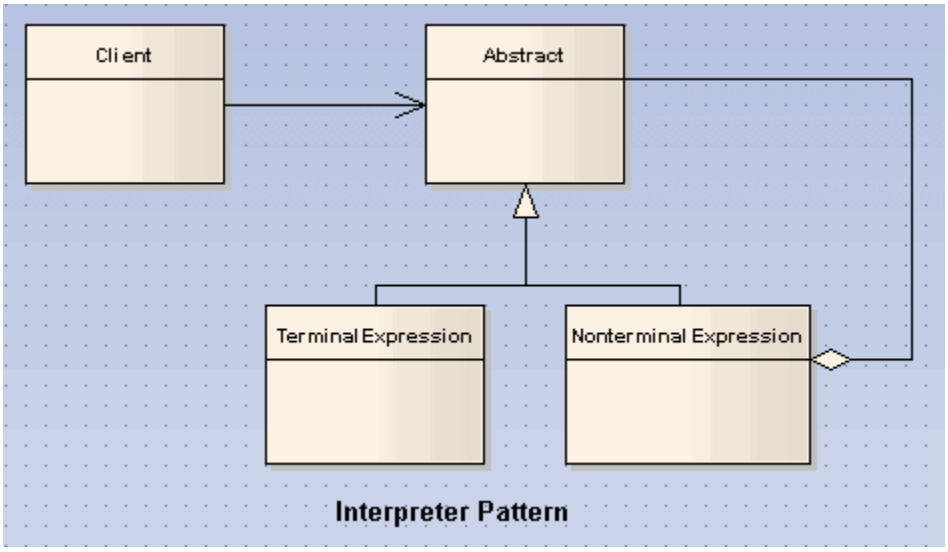
实现了抽象节点角色所规定的接受操作。

- 结构对象(Object Structure)角色

可以遍历结构中的所有元素；如果需要，提供一个高层次的接口访问者对象可以访问每一个元素；如果需要，可以设计成一个复合对象或者一个聚集。

5.11. 解释器模式

解释器模式是类的行为模式。给定一个语言之后，解释器模式可以定义出其方法的一种表示，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。



5.12. 调停者模式

调停者模式是对象的行为模式。

调停者模式包装了一系统对象相互作用的方法，使得这些对象不必明显互相引用。从而使它们可以较松散的耦合。当这些对象中的某些对角之间的相互作用发生改变时，不会立既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地变化。

结构及角色

- 抽象调停者角色

定义出同事对象到调停者对象的接品，其中主要的方法是一个或者多个事件

方法。一般由抽象类或接口实现。

- 具体调停者角色

  从抽角调停者继承而来，实现了抽象超类所声明的事件方法。此角色知道所有的具体同事类，它从具体同事对象接收消息、向具体同事对象发现命令。

- 抽象同事类角色

  定义出调停者到同事对象的接口。同事对象只知道调停者而不知道其余的同事对象。一般而言，由抽象类或接口实现。

- 具体同事类角色

  实现抽象同事类所规定的接口。每一个具体同事类都很清楚它自己在小范围内的行为，而不知道它在在范围内的目的。