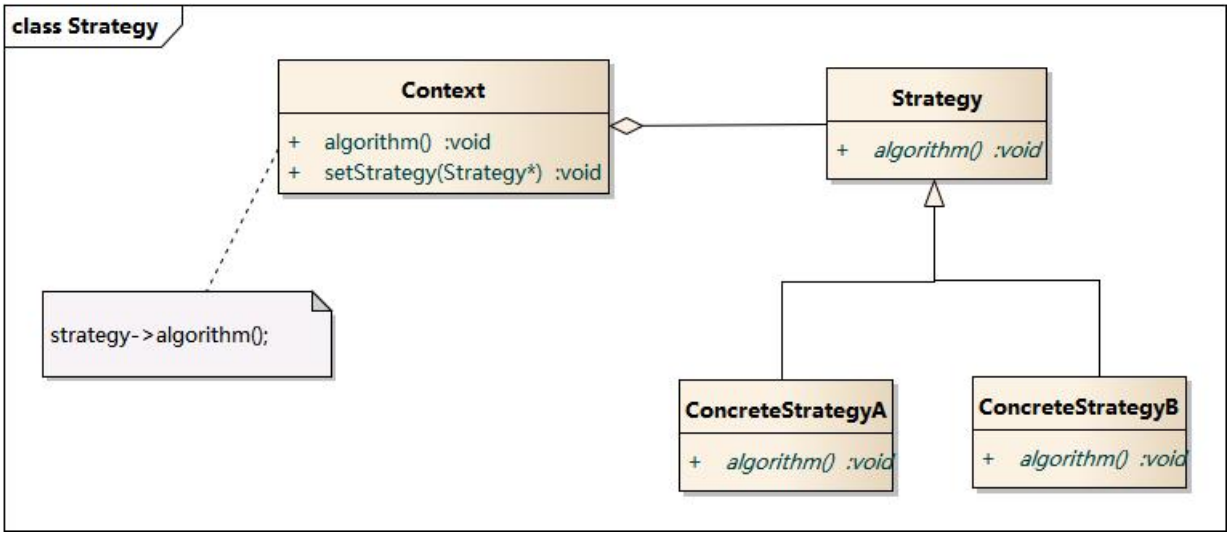
设计模式

算法 算法 设计模式

策略模式

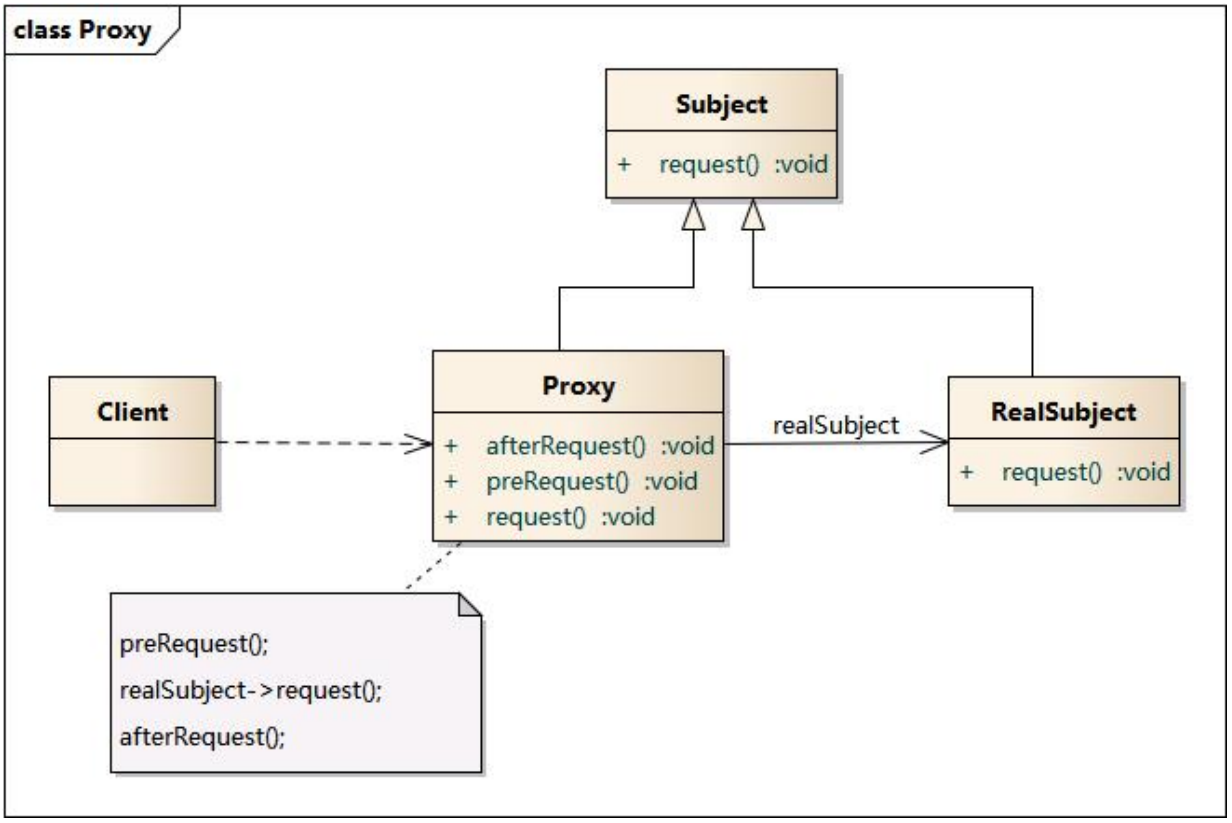
策略模式(Strategy Pattern)：定义一系列算法，将每一个算法封装起来，并让它们可以相互替换。策略模式让算法独立于使用它的客户而变化，也称为政策模式(Policy)。

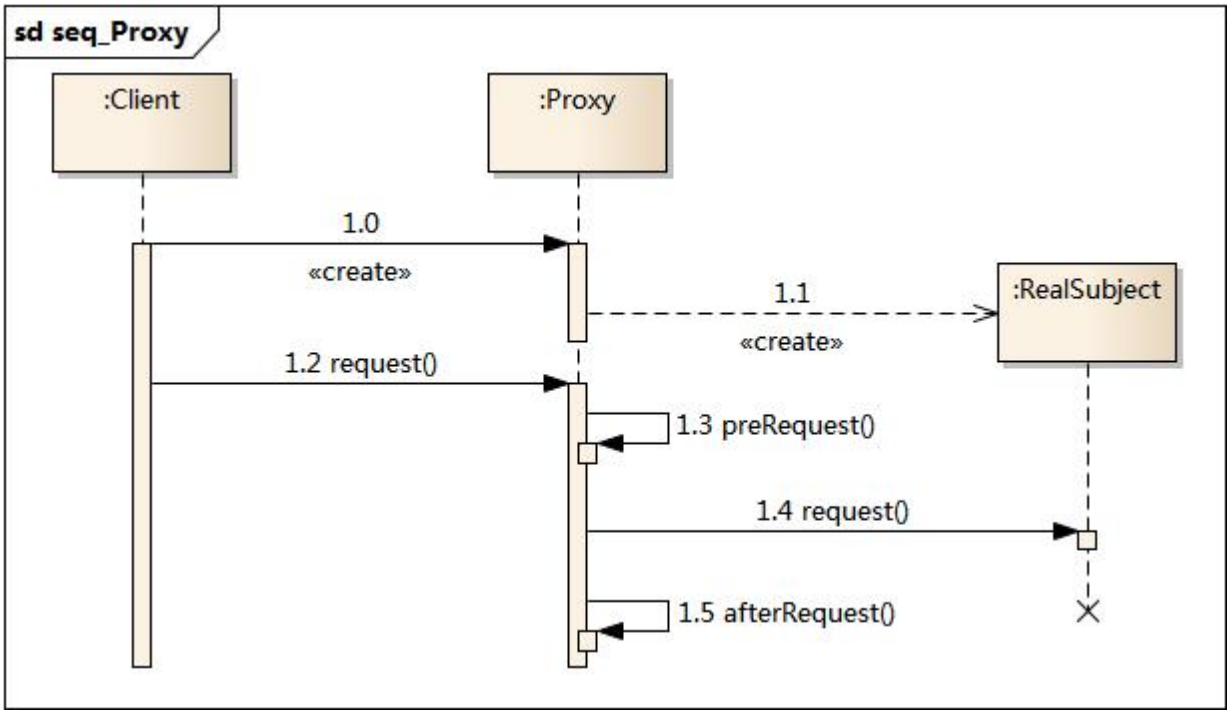


代理模式

在某些情况下，一个客户不想或者不能直接引用一个对 象，此时可以通过一个称之为“代理”的第三者来实现 间接引用。代理对象可以在客户端和目标对象之间起到 中介的作用，并且可以通过代理对象去掉客户不能看到 的内容和服务或者添加客户需要的额外服务。

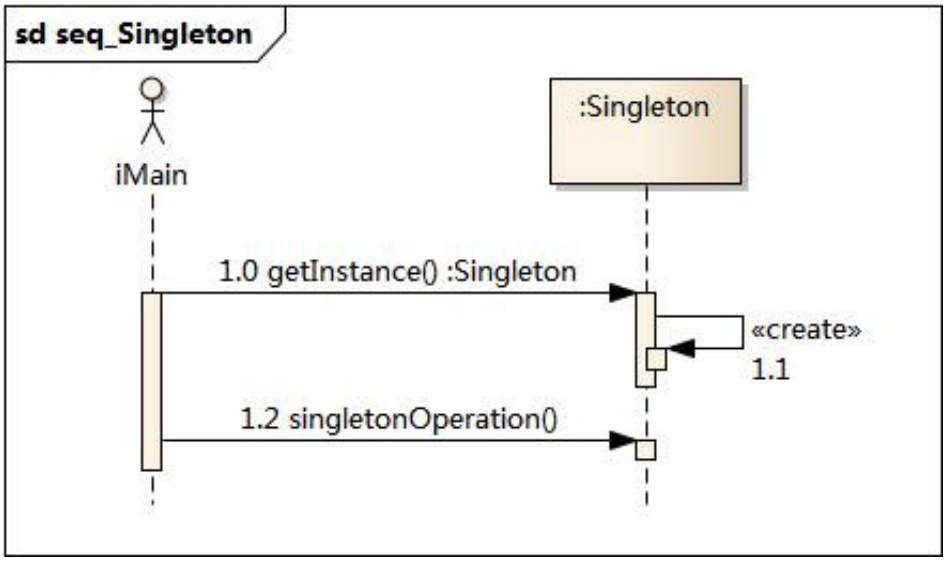
通过引入一个新的对象（如小图片和远程代理 对象）来实现对真实对象的操作或者将新的对象作为真实对象的一个替身，这种实现机制即 为代理模式，通过引入代理对象来间接访问一个对象，这就是代理模式的模式动机。
代理模式主要使用了 Java 的多态，干活的是被代理类，代理类主要是接活，你让我干活，好，我交给幕后的类去干，你满意就成，那怎么知道被代理类能不能干呢？同根就成，大家知根知底，你能做啥，我能做啥都清楚的很，同一个接口呗。

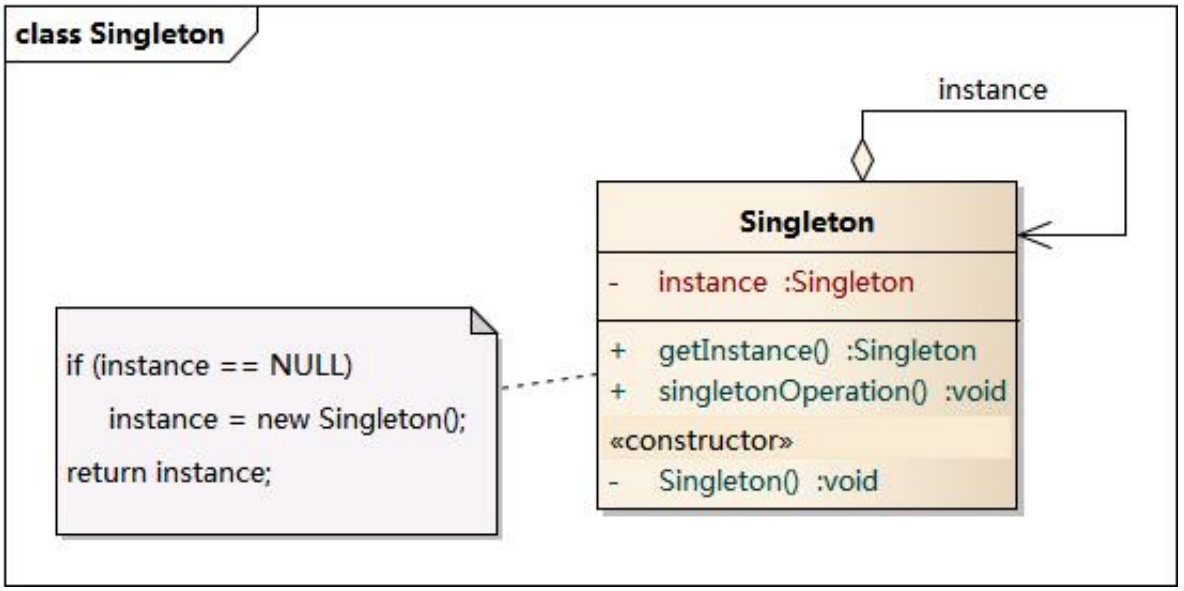




单例模式

单例模式很简单，就是在构造函数中多了加一个构造函数，访问权限是 private 的就可以了，这个模式是简单，但是简单中透着风险，风险？什么风险？在一个 B/S 项目中，每个 HTTP Request 请求到 J2EE的容器上后都创建了一个线程,每个线程都要创建同一个单例对象。

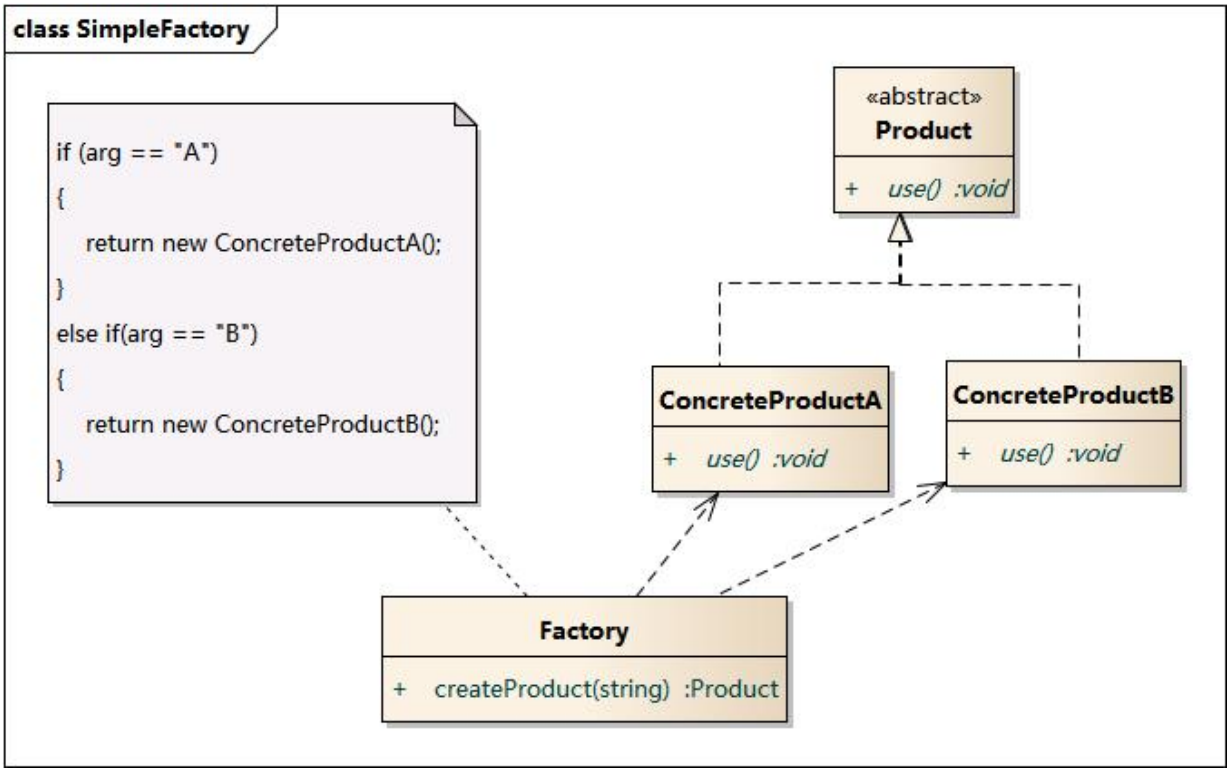


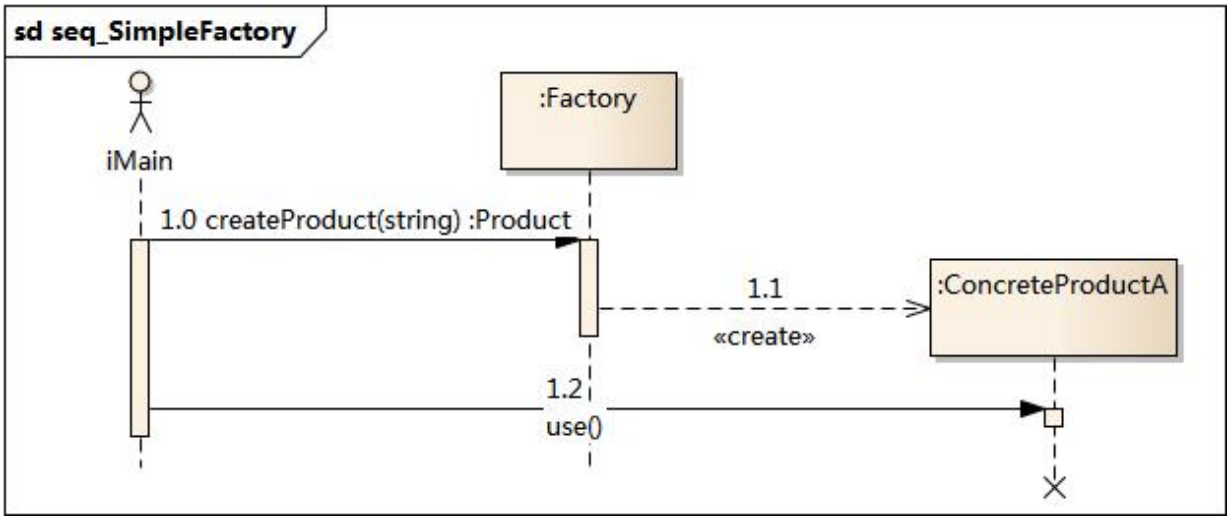


多例模式

工厂模式

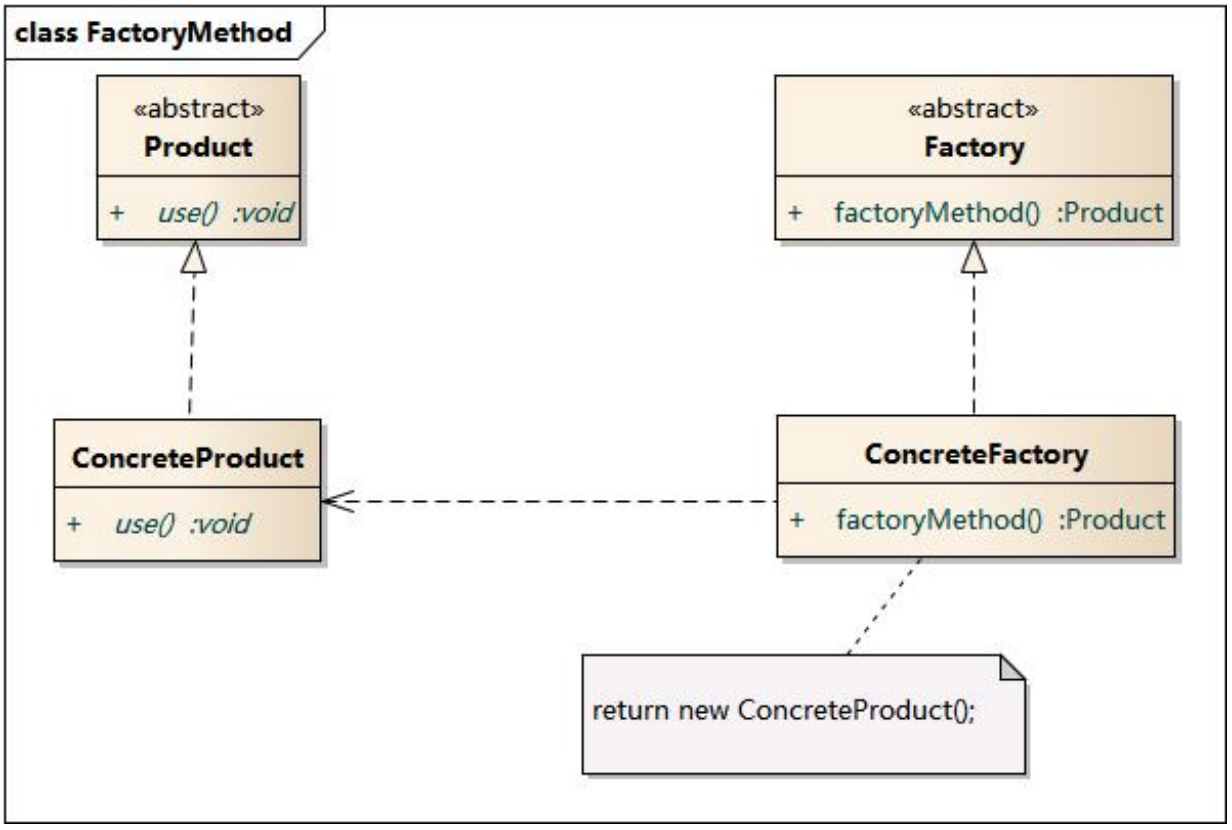
简单工厂模式

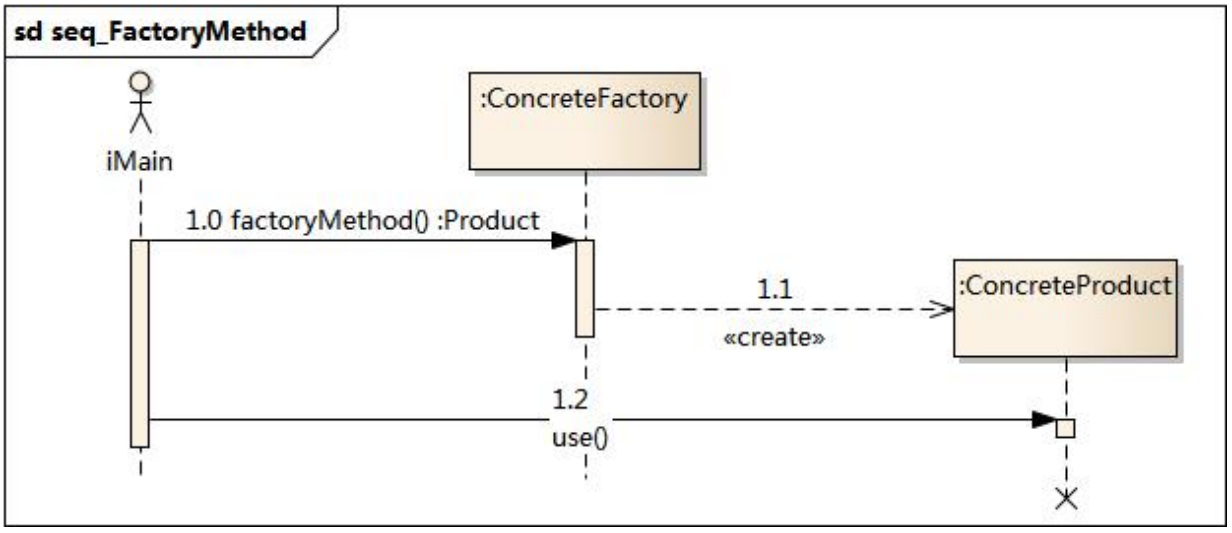




工厂方法模式

工厂方法模式(Factory Method Pattern)又称为工厂模式，也叫虚拟构造器(Virtual Constructor)模式或者多态工厂(Polymorphic Factory)模式，它属于类创建型模式。在工厂方法模式中，工厂父类负责定义创建产品对象的公共接口，而工厂子类则负责生成具体的产品对象，这样做的目的是将产品类的实例化操作延迟到工厂子类中完成，即通过工厂子类来确定究竟应该实例化哪一个具体产品类。



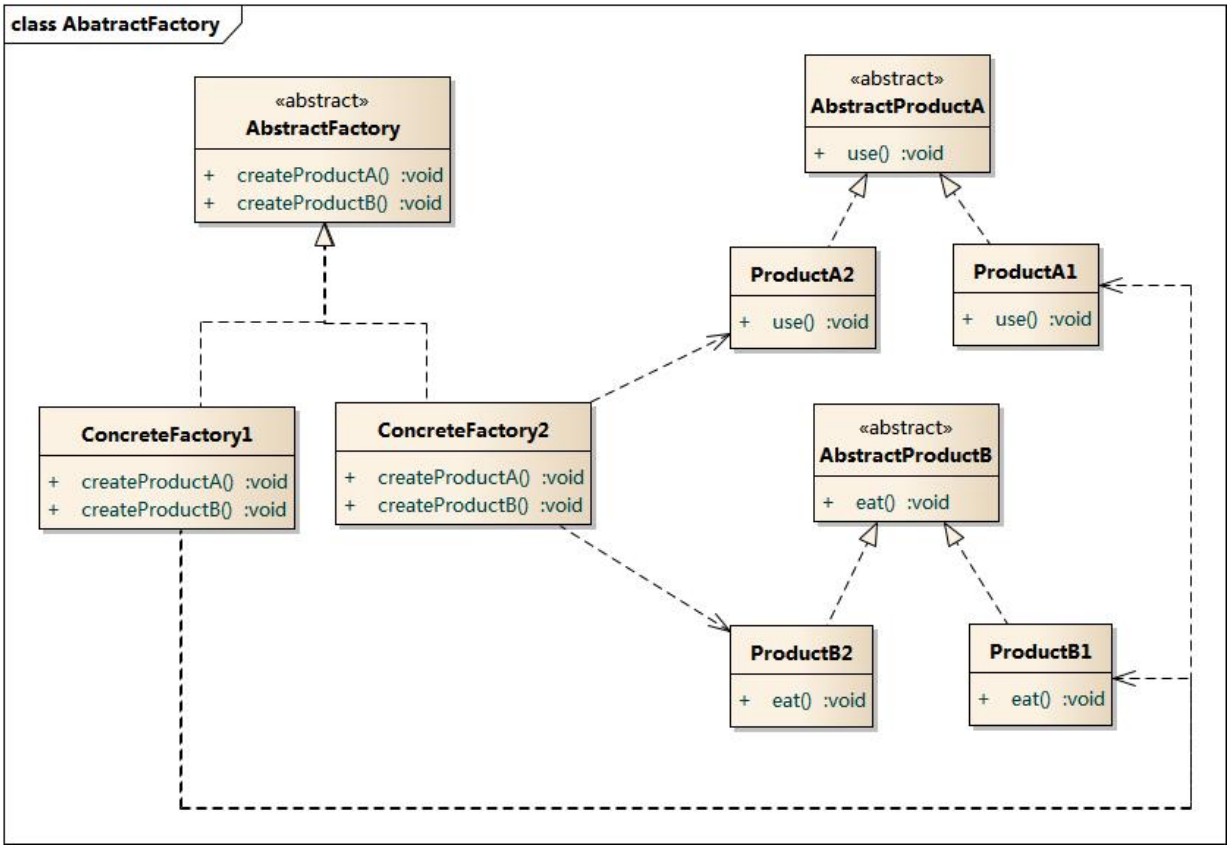


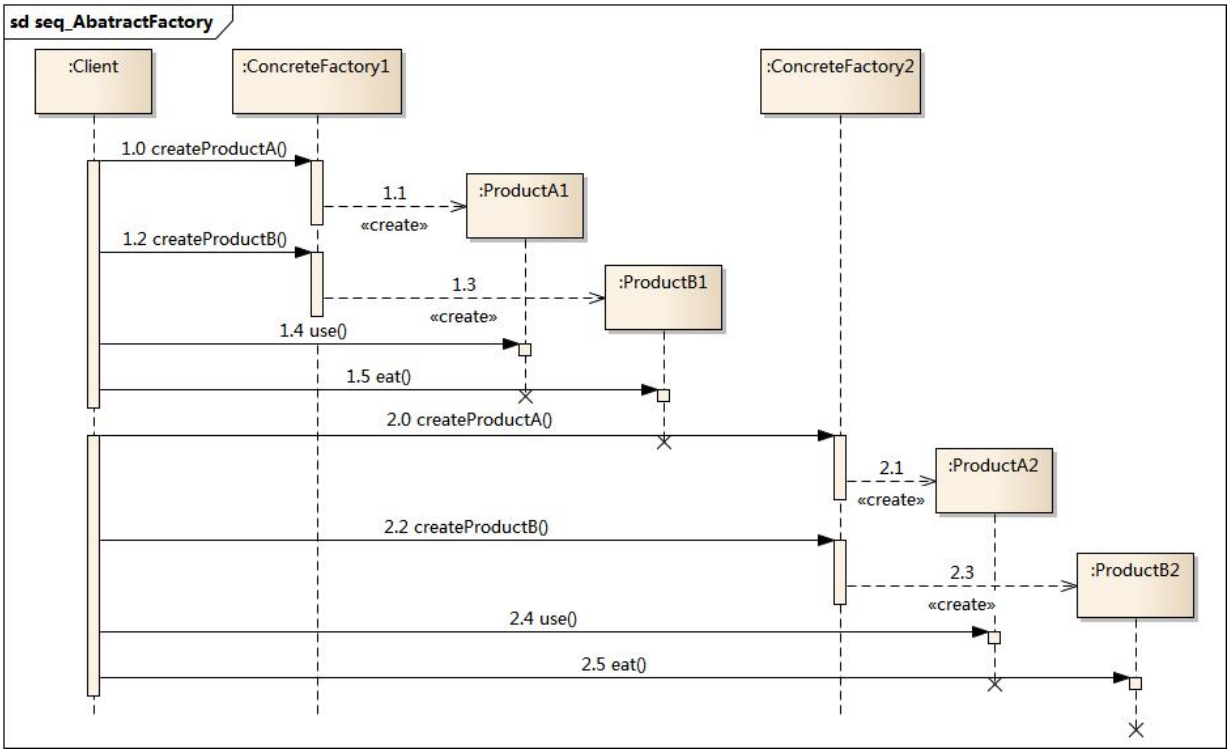
抽象工厂模式

在工厂方法模式中具体工厂负责生产具体的产品，每一个具体工厂对应一种具体产品，工厂方法也具有唯一性，一般情况下，一个具体工厂中只有一个工厂方法或者一组重载的工厂方法。但是有时候我们需要一个工厂可以提供多个产品对象，而不是单一的产品对象。

为了更清晰地理解工厂方法模式，需要先引入两个概念：

产品等级结构 ：产品等级结构即产品的继承结构，如一个抽象类是电视机，其子类有海尔电视机、海信电视机、TCL电视机，则抽象电视机与具体品牌的电视机之间构成了一个产品等级结构，抽象电视机是父类，而具体品牌的电视机是其子类。
产品族 ：在抽象工厂模式中，产品族是指由同一个工厂生产的，位于不同产品等级结构中的一组产品，如海尔电器工厂生产的海尔电视机、海尔电冰箱，海尔电视机位于电视机产品等级结构中，海尔电冰箱位于电冰箱产品等级结构中。
当系统所提供的工厂所需生产的具体产品并不是一个简单的对象，而是多个位于不同产品等级结构中属于不同类型的具体产品时需要使用抽象工厂模式。
抽象工厂模式是所有形式的工厂模式中最为抽象和最具一般性的一种形态。
抽象工厂模式与工厂方法模式最大的区别在于，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，一个工厂等级结构可以负责多个不同产品等级结构中的产品对象的创建 。当一个工厂等级结构可以创建出分属于不同产品等级结构的一个产品族中的所有对象时，抽象工厂模式比工厂方法模式更为简单、有效率。

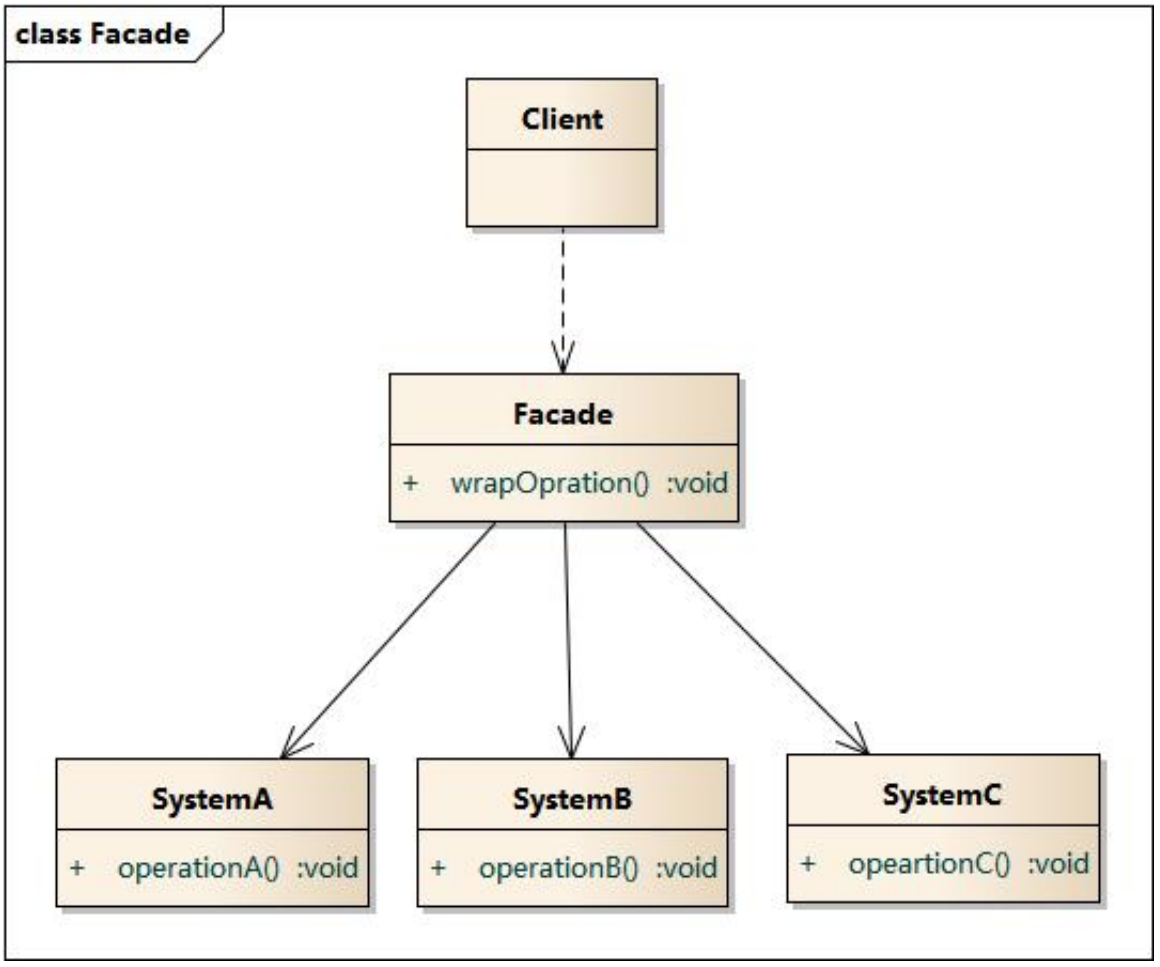


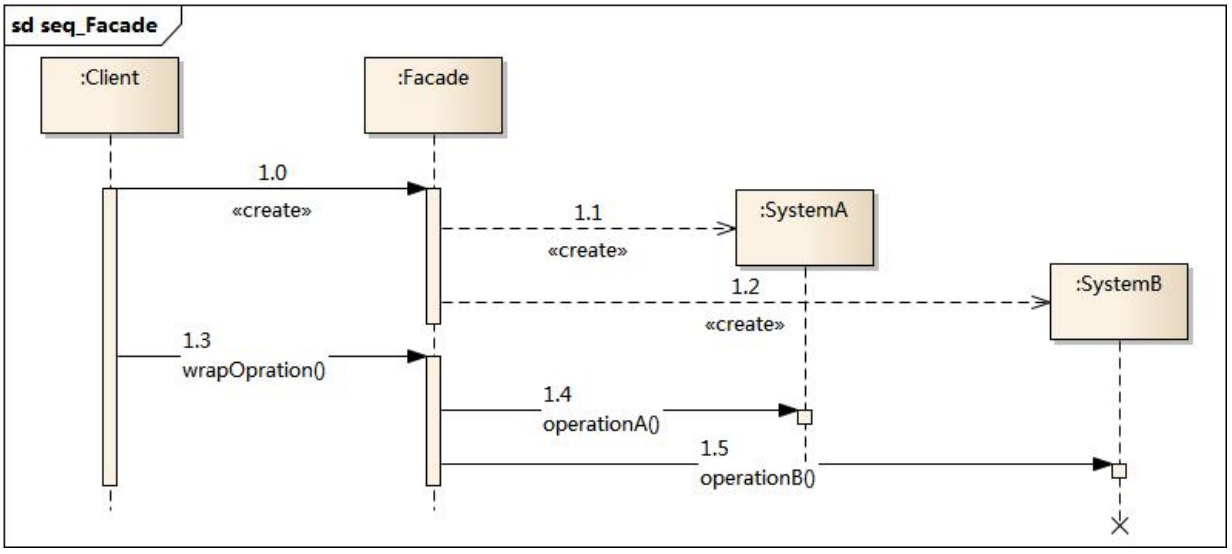


工厂模式符合 OCP 原则，也就是开闭原则，怎么说呢，比如就性别的问题，这个世界上还存在双性人，是男也是女的人，那这个就是要在我们的产品族中增加一类产品，同时再增加一个工厂就可以解决这个问题，不需要我再来实现了吧，很简单的大家自己画下类图，然后实现下。抽象工厂模式，还有一个非常大的有点，高内聚，低耦合，在一个较大的项目组，产品是由一批人定义开发的，但是提供其他成员访问的时候，只有工厂方法和产品的接口，也就是说只需要提供 Product Interface 和 Concrete Factory 就可以产生自己需要的对象和方法，Java 的高内聚低耦合的特性表现的一览无遗，哈哈。

门面模式【 Facade Pattern】外观模式

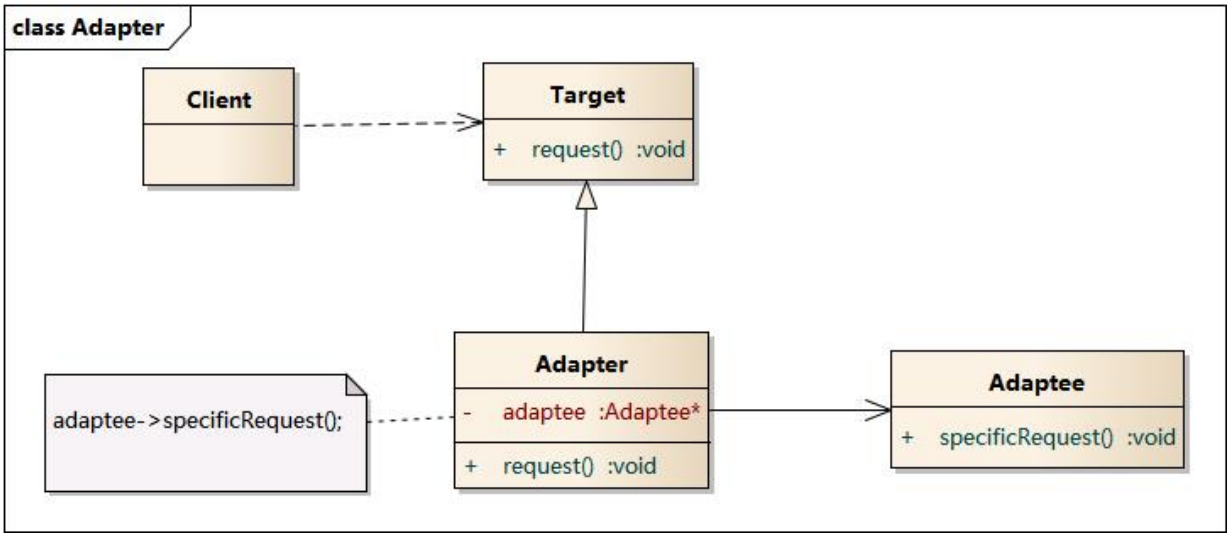
根据“单一职责原则”，在软件中将一个系统划分为若干个子系统有利于降低整个系统的复杂性，一个常见的设计目标是使子系统间的通信和相互依赖关系达到最小，而达到该目标的途径之一就是引入一个外观对象，它为子系统的访问提供了一个简单而单一的入口。 -外观模式也是“迪米特法则”的体现，通过引入一个新的外观类可以降低原有系统的复杂度，同时降低客户类与子系统类的耦合度。 - 外观模式要求一个子系统的外部与其内部的通信通过一个统一的外观对象进行，外观类将客户端与子系统的内部复杂性分隔开，使得客户端只需要与外观对象打交道，而不需要与子系统内部的很多对象打交道。 -外观模式的目的在于降低系统的复杂程度。 -外观模式从很大程度上提高了客户端使用的便捷性，使得客户端无须关心子系统的工作细节，通过外观角色即可调用相关功能。


class Facade
Client
Facade
+ wrapOpration() :void
SystemA
+ operationA() :void
SystemB
+ operationB() :void
SystemC
+ opeartionC() :void

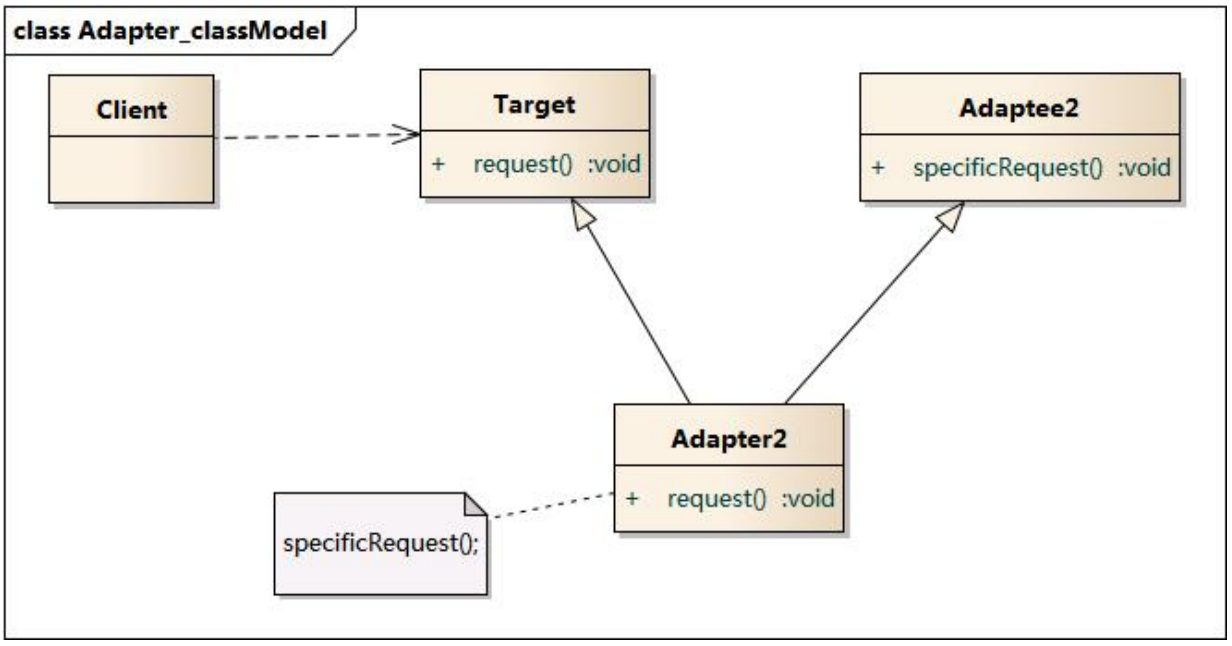

sd seq_Facade
:Client
:Facade
1.0
«create»
1.1
«create»
:SystemA
1.2
«create»
:SystemB
1.3
wrapOpration()
1.4
operationA()
1.5
operationB()

适配器模式

对象适配器



类适配器



将目标类和适配者类解耦，通过引入一个适配器类来重用现有的适配者类，而无须修改原有代码。
增加了类的透明性和复用性，将具体的实现封装在适配者类中，对于客户端类来说是透明的，而且提高了适配者的复用性。
灵活性和扩展性都非常好，通过使用配置文件，可以很方便地更换适配器，也可以在不修改原有代码的基础上增加新的适配器类，完全符合“开闭原则”。
类适配器模式还具有如下优点：
由于适配器类是适配者类的子类，因此可以在适配器类中置换一些适配者的方法，使得适配器的灵活性更强。
对象适配器模式还具有如下优点：
一个对象适配器可以把多个不同的适配者适配到同一个目标，也就是说，同一个适配器可以把适配者类和它的子类都适配到目标接口。

模板方法模式

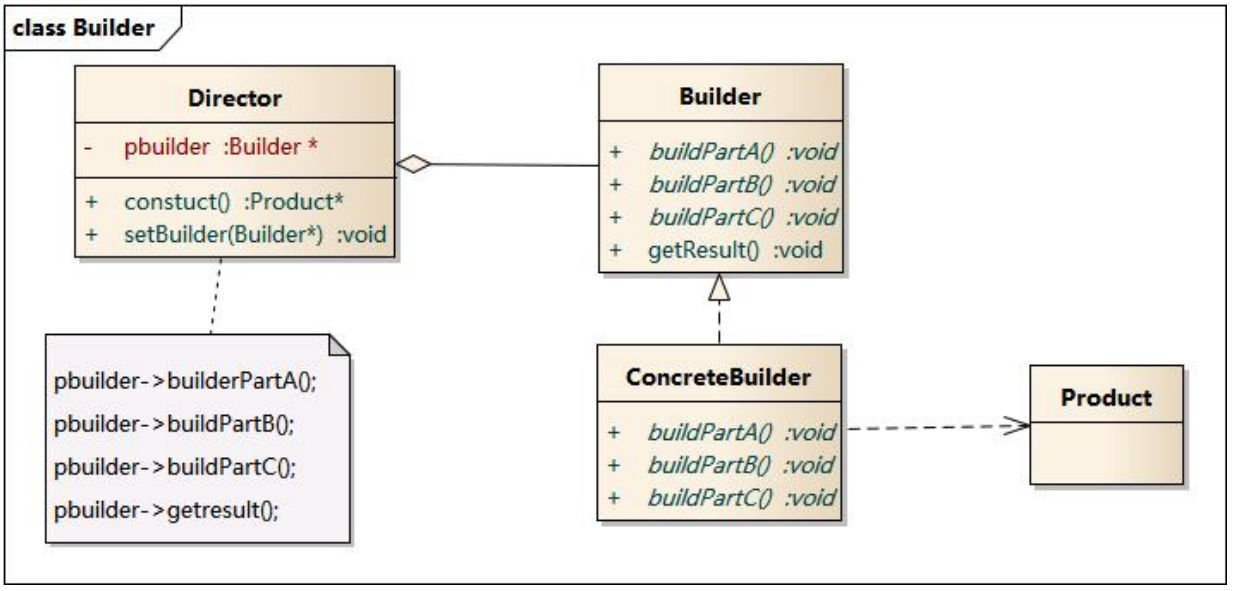
模板方法模式就是在模板方法中按照一个的规则和顺序调用基本方法。

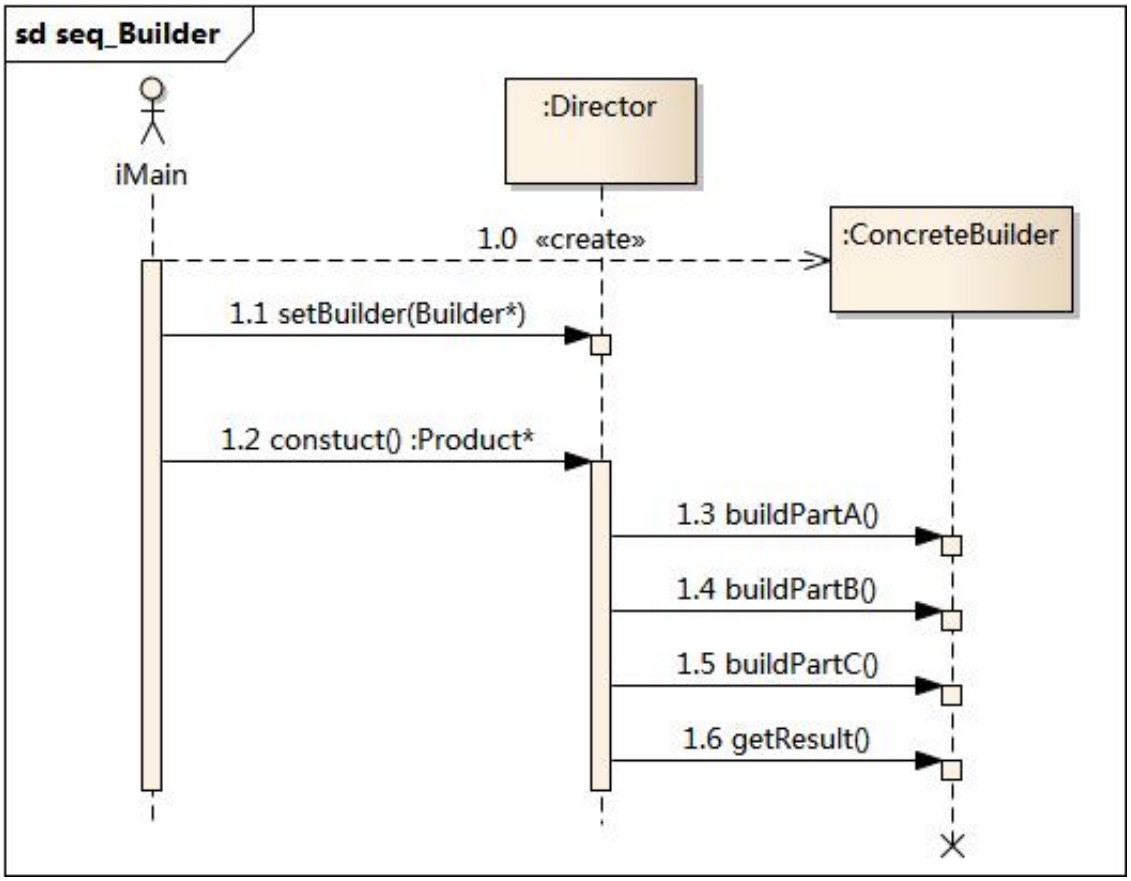
建造者模式

复杂对象相当于一辆有待建造的汽车，而对象的属性相当于汽车的部件，建造产品的过程就相当于组合部件的过程。由于组合部件的过程很复杂，因此，这些部件的组合过程往往被“外部化”到一个称作建造者的对象里，建造者返还给客户端的是一个已经建造完毕的完整产品对象，而用户无须关心该对象所包含的属性以及它们的组装方式，这就是建造者模式的模式动机。

造者模式(Builder Pattern)：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。

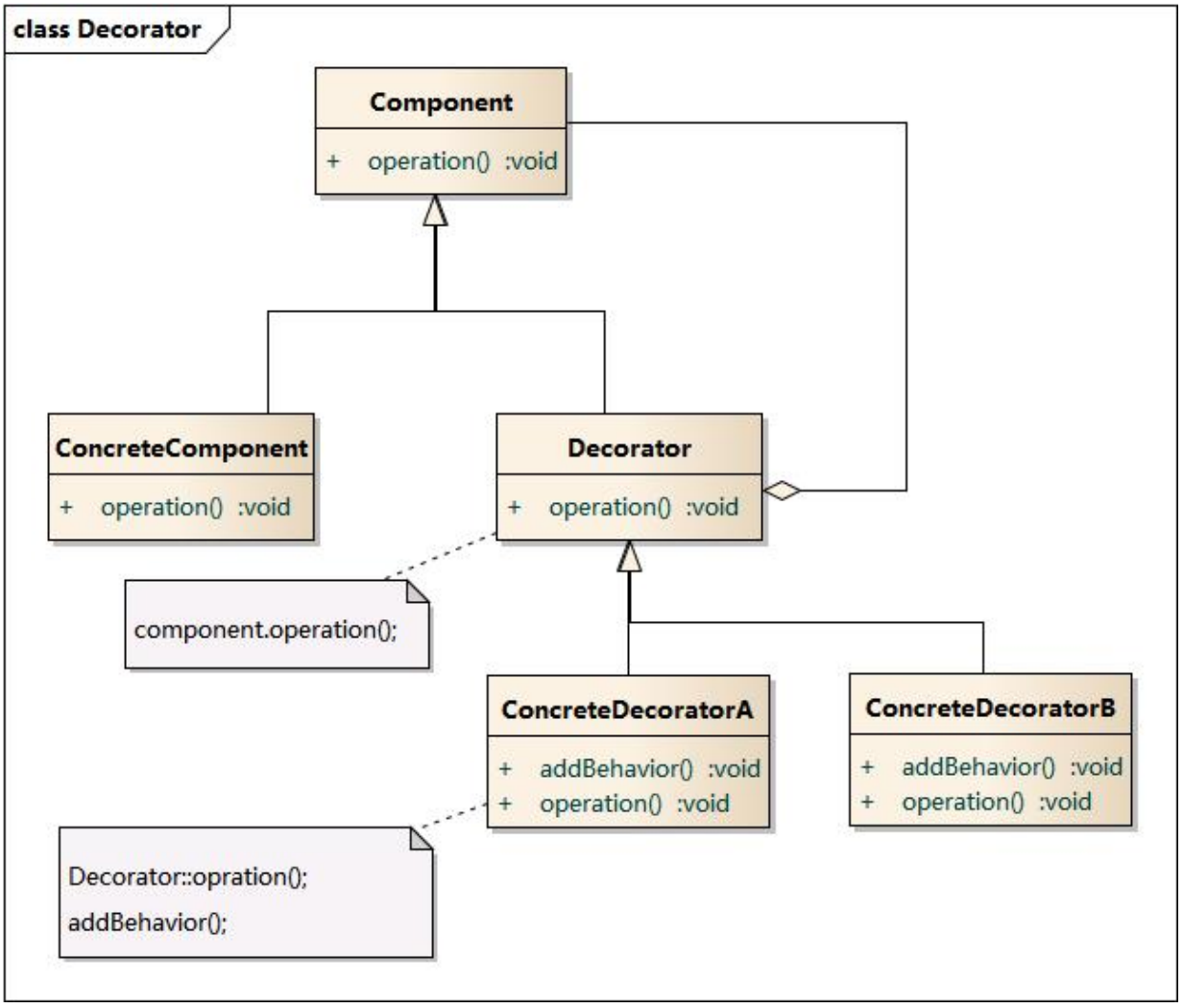
建造者模式是一步一步创建一个复杂的对象，它允许用户只通过指定复杂对象的类型和内容就可以构建它们，用户不需要知道内部的具体构建细节。建造者模式属于对象创建型模式。根据中文翻译的不同，建造者模式又可以称为生成器模式。

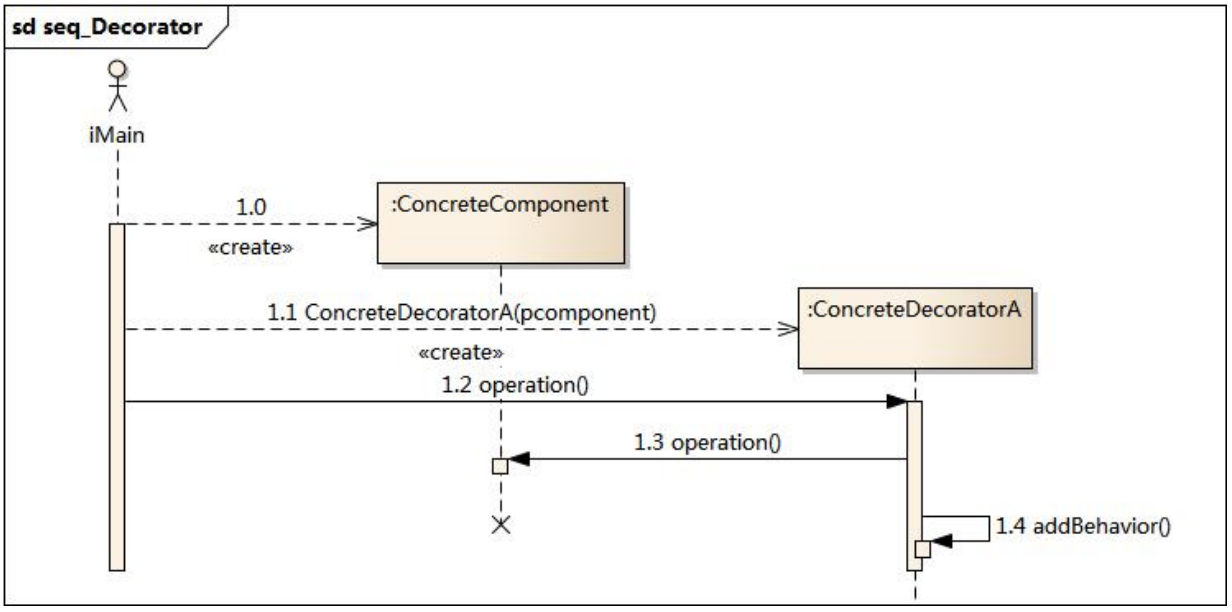




装饰模式

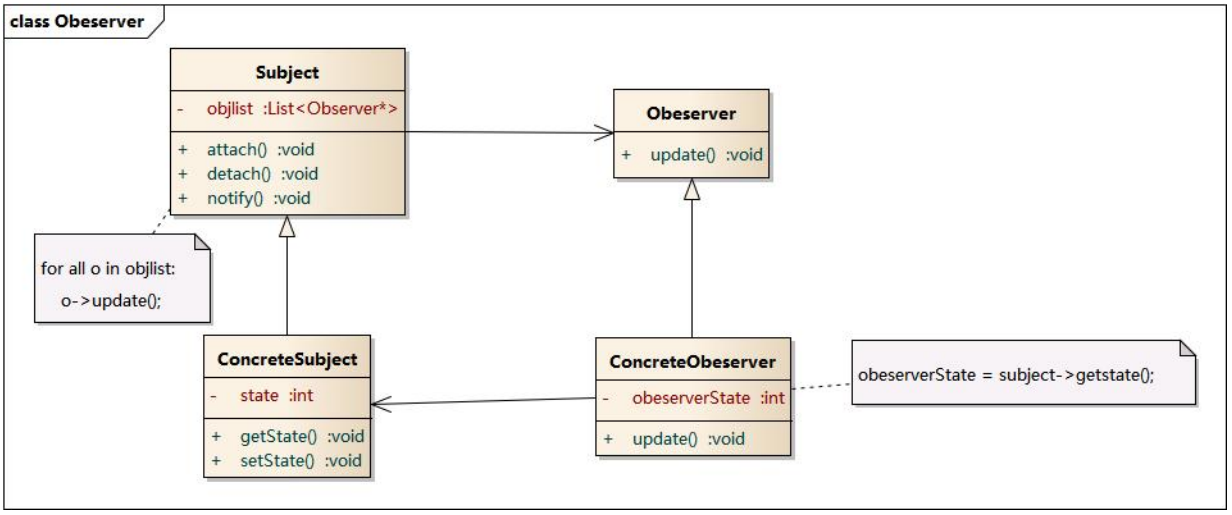
装饰模式(Decorator Pattern) ：动态地给一个对象增加一些额外的职责(Responsibility)，就增加对象功能来说，装饰模式比生成子类实现更为灵活。其别名也可以称为包装器(Wrapper)，与适配器模式的别名相同，但它们适用于不同的场合。根据翻译的不同，装饰模式也有人称之为“油漆工模式”，它是一种对象结构型模式。

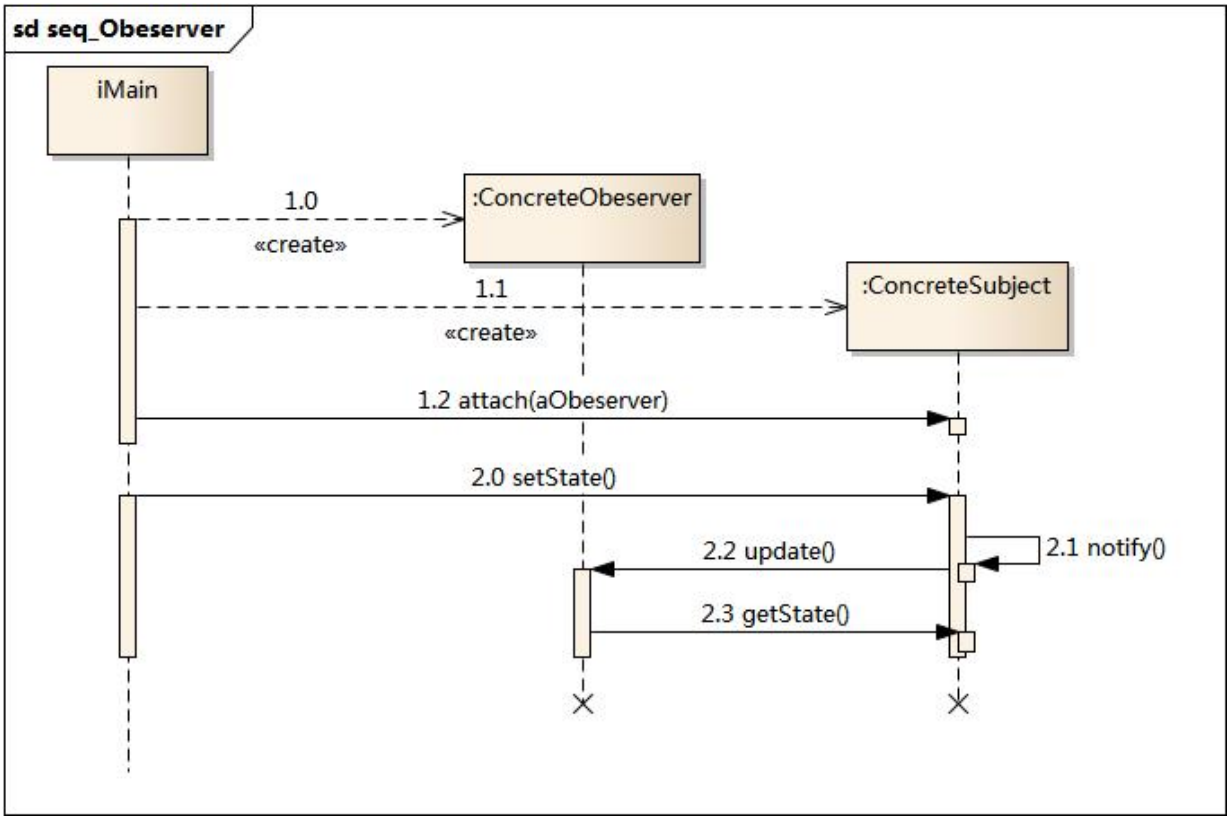




观察者模式

建立一种对象与对象之间的依赖关系，一个对象发生改变时将自动通知其他对象，其他对象将相应做出反应。在此，发生改变的对象称为观察目标，而被通知的对象称为观察者，一个观察目标可以对应多个观察者，而且这些观察者之间没有相互联系，可以根据需要增加和删除观察者，使得系统更易于扩展，这就是观察者模式的模式动机。（四年级成绩单的例子）



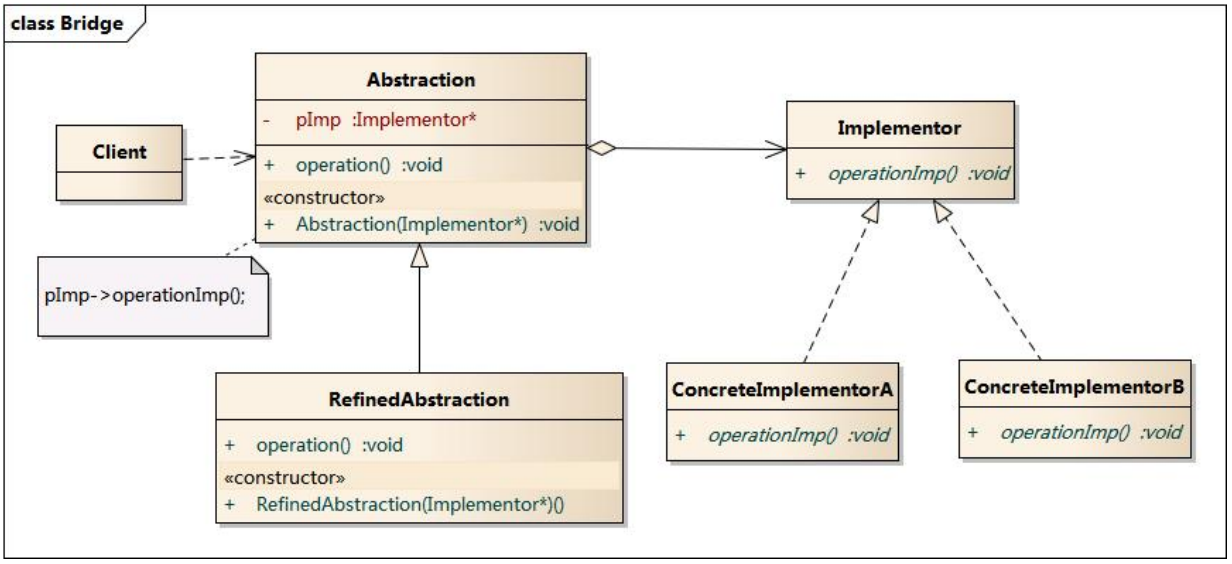


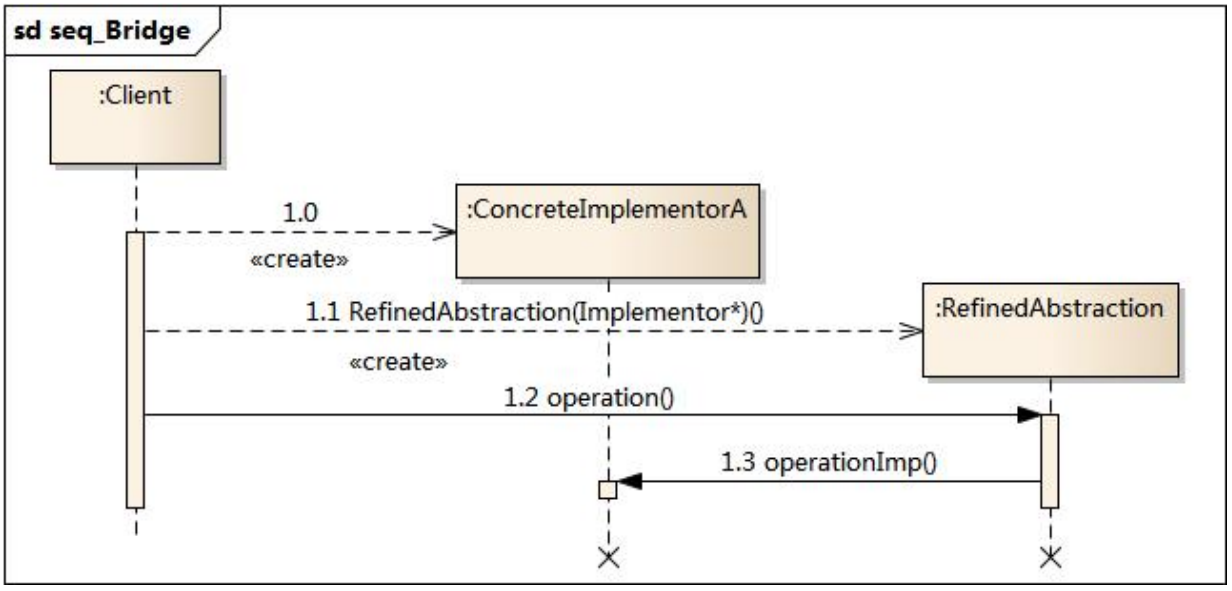
桥接模式

设想如果要绘制矩形、圆形、椭圆、正方形，我们至少需要4个形状类，但是如果绘制的图形需要具有不同的颜色，如红色、绿色、蓝色等，此时至少有如下两种设计方案：

第一种设计方案是为每一种形状都提供一套各种颜色的版本。
第二种设计方案是根据实际需要对形状和颜色进行组合
对于有两个变化维度（即两个变化的原因）的系统，采用方案二来进行设计系统中类的个数更少，且系统扩展更为方便。设计方案二即是桥接模式的应用。桥接模式将继承关系转换为关联关系，从而降低了类与类之间的耦合，减少了代码编写量。

桥梁模式的优点就是类间解耦，我们上面已经提到，两个角色都可以自己的扩展下去，不会相互影响，这个
也符合 OCP 原则。

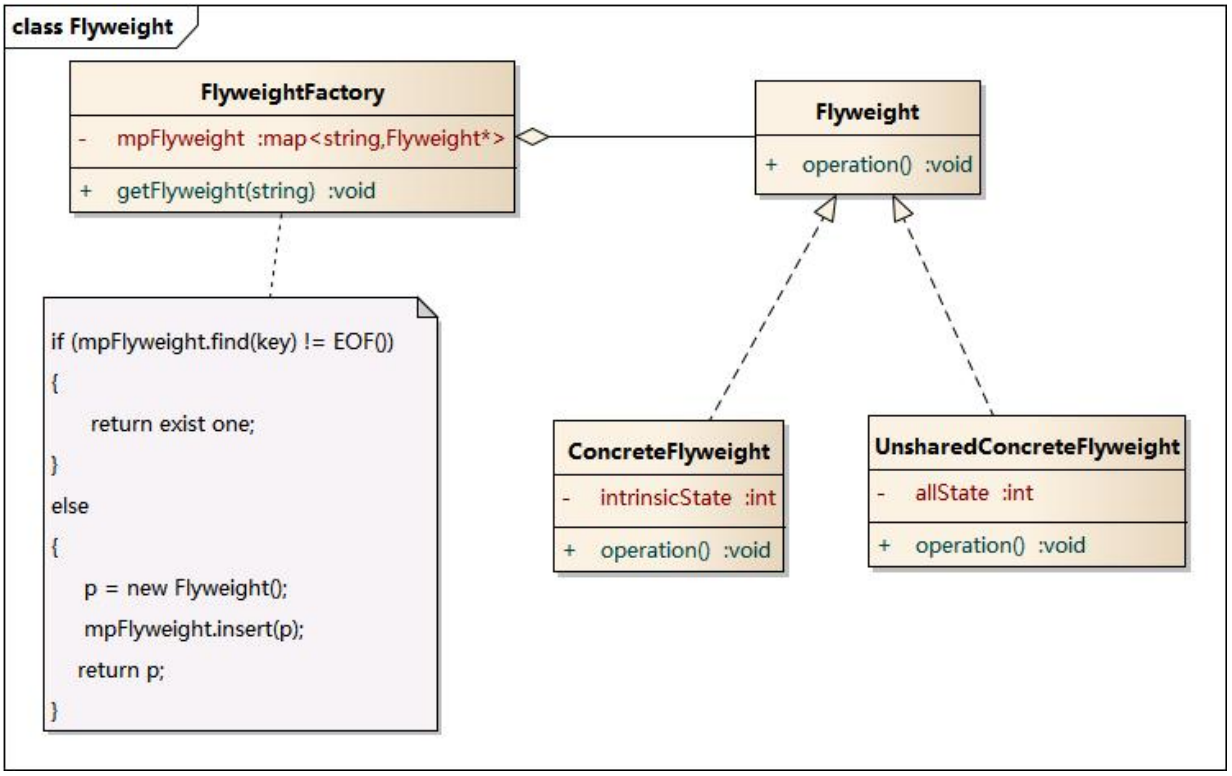


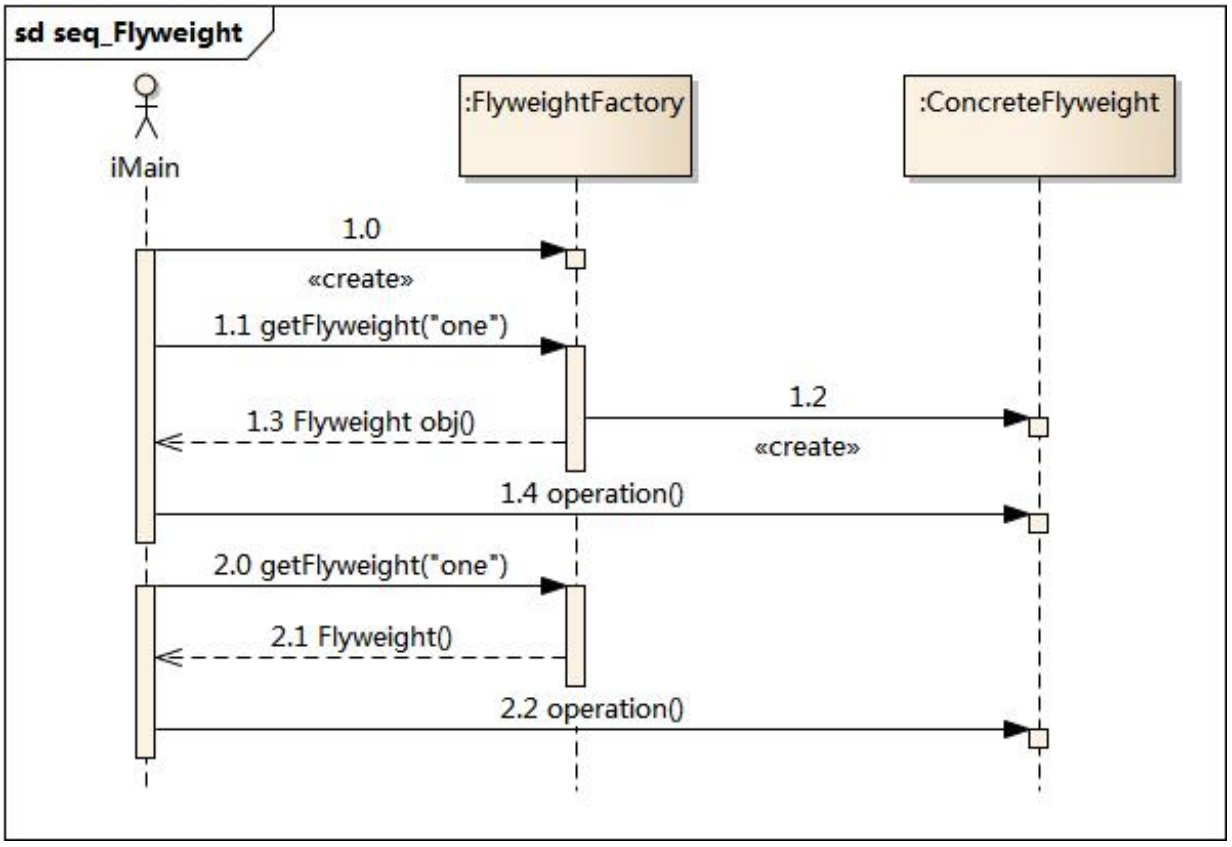


享元模式

面向对象技术可以很好地解决一些灵活性或可扩展性问题，但在很多情况下需要在系统中增加类和对象的个数。当对象数量太多时，将导致运行代价过高，带来性能下降等问题。
享元模式正是为解决这一类问题而诞生的。享元模式通过共享技术实现相同或相似对象的重用。
在享元模式中可以共享的相同内容称为内部状态(IntrinsicState)，而那些需要外部环境来设置的不能共享的内容称为外部状态(Extrinsic State)，由于区分了内部状态和外部状态，因此可以通过设置不同的外部状态使得相同的对象可以具有一些不同的特征，而相同的内部状态是可以共享的。
在享元模式中通常会出现工厂模式，需要创建一个享元工厂来负责维护一个享元池(Flyweight Pool)用于存储具有相同内部状态的享元对象。
在享元模式中共享的是享元对象的内部状态，外部状态需要通过环境来设置。在实际使用中，能够共享的内部状态是有限的，因此享元对象一般都设计为较小的对象，它所包含的内部状态较少，这种对象也称为细粒度对象。享元模式的目的就是使用共享技术来实现大量细粒度对象的复用。

Flyweight: 抽象享元类
ConcreteFlyweight: 具体享元类
UnsharedConcreteFlyweight: 非共享具体享元类
FlyweightFactory: 享元工厂类





1、享元模式的优点在于它能够极大的减少系统中对象的个数。 2、享元模式由于使用了外部状态，外部状态相对独立，不会影响到内部状态，所以享元模式使得享元对象能够在不同的环境被共享。

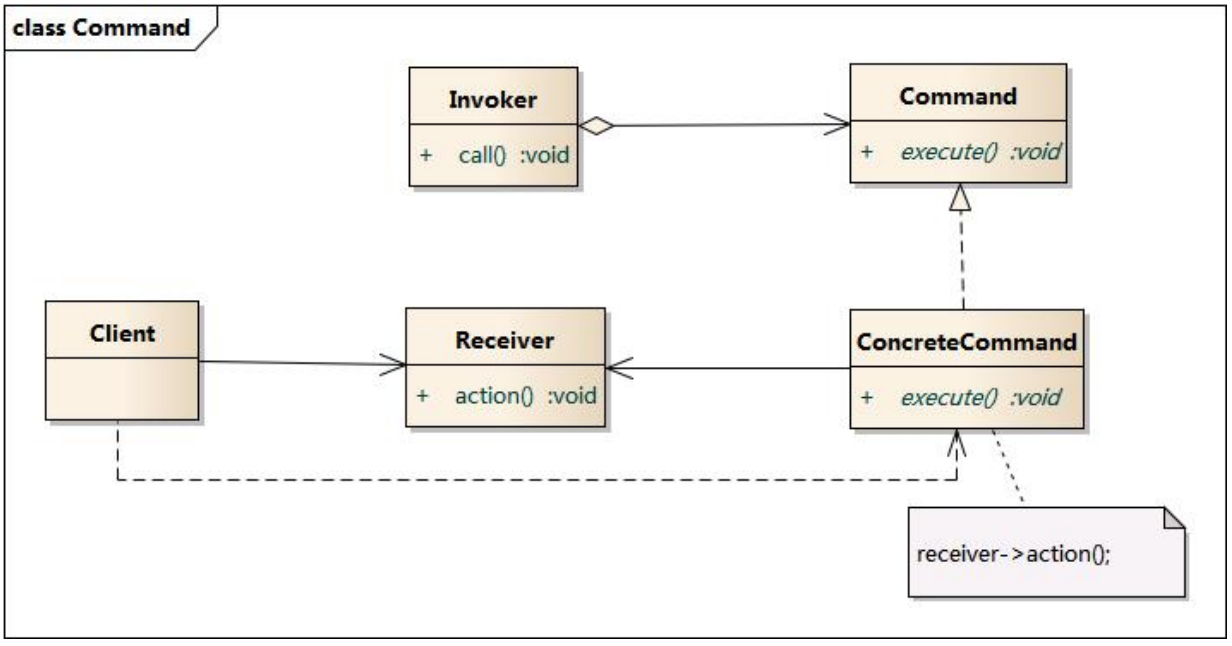
命令模式

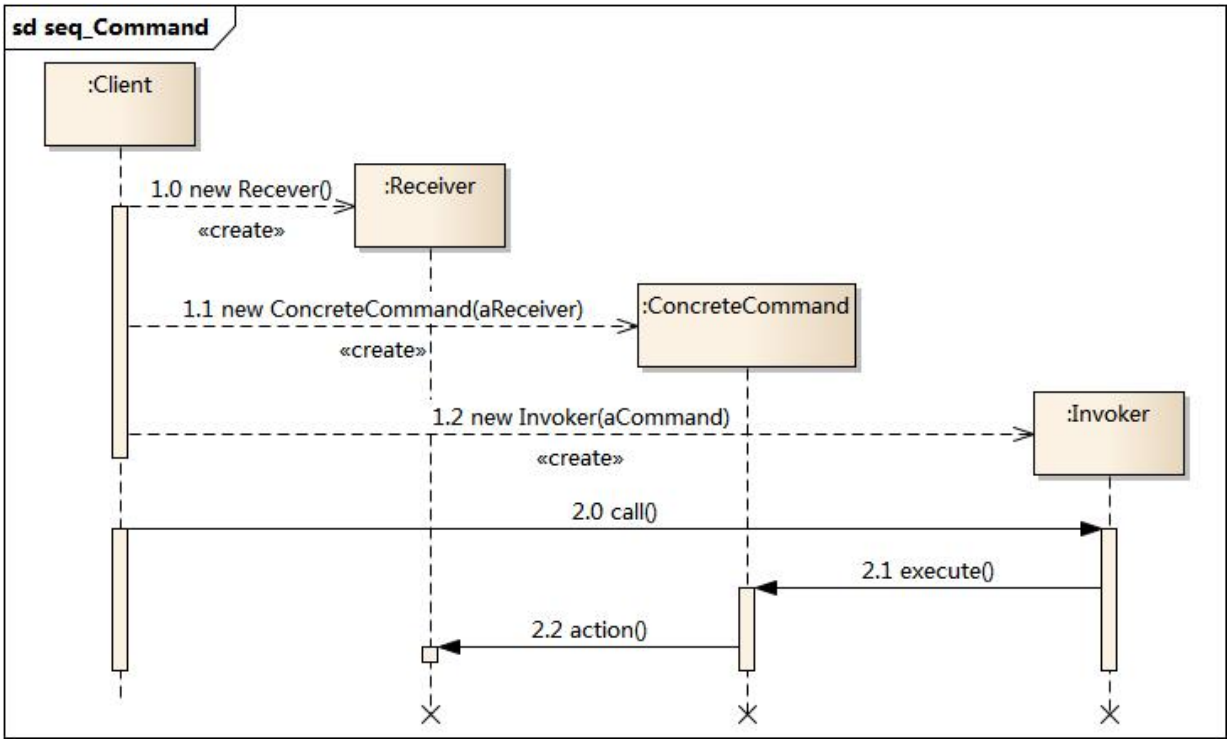
在软件设计中，我们经常需要向某些对象发送请求，但是并不知道请求的接收者是谁，也不知道被请求的操作是哪个，我们只需在程序运行时指定具体的请求接收者即可，此时，可以使用命令模式来进行设计，使得请求发送者与请求接收者消除彼此之间的耦合，让对象之间的调用关系更加灵活。

命令模式可以对发送者和接收者完全解耦，发送者与接收者之间没有直接引用关系，发送请求的对象只需要知道如何发送请求，而不必知道如何完成请求。这就是命令模式的模式动机。

命令模式(Command Pattern)：将一个请求封装为一个对象，从而使我们可用不同的请求对客户进行参数化；对请求排队或者记录请求日志，以及支持可撤销的操作。命令模式是一种对象行为型模式，其别名为动作(Action)模式或事务(Transaction)模式。

Command: 抽象命令类
ConcreteCommand: 具体命令类
Invoker: 调用者
Receiver: 接收者
Client:客户类





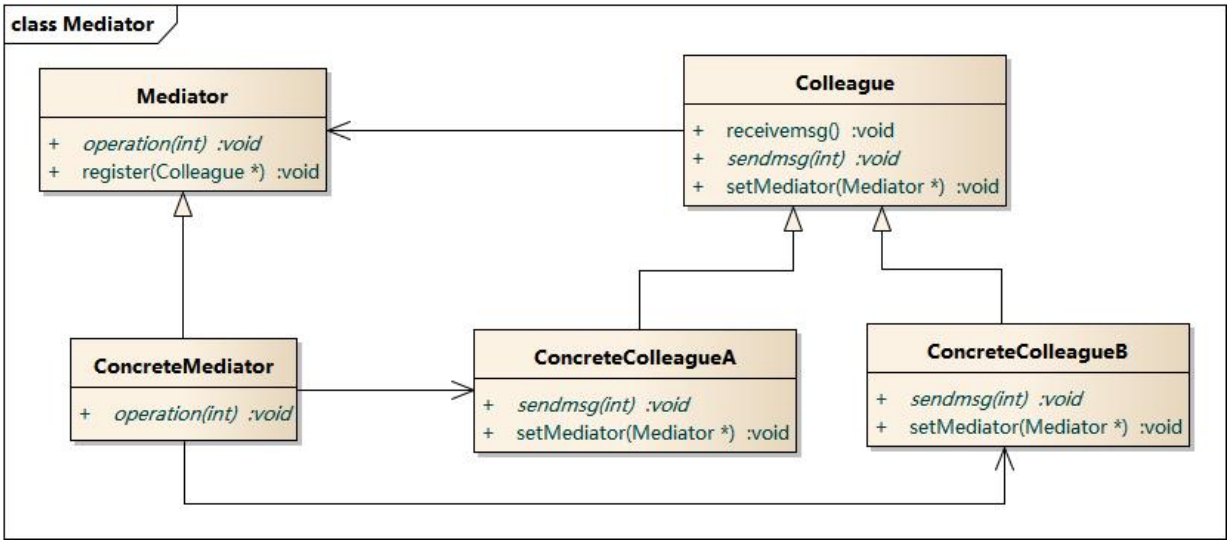
命令模式的优点

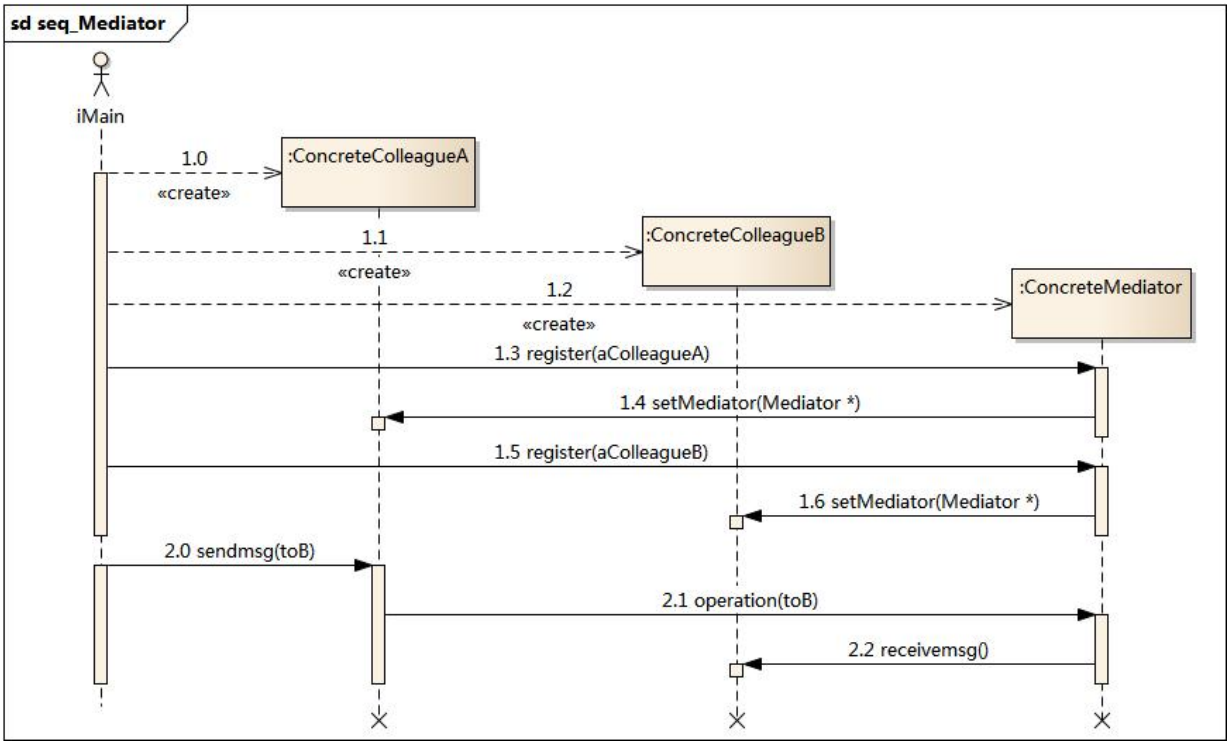
降低系统的耦合度。
新的命令可以很容易地加入到系统中。
可以比较容易地设计一个命令队列和宏命令（组合命令）。
可以方便地实现对请求的Undo和Redo。

可能导致具体命令太多

中介者模式

中介者模式(Mediator Pattern)定义：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。中介者模式又称为调停者模式，它是一种对象行为型模式。





抽象中介者（Mediator）角色：抽象中介者角色定义统一的接口用于各同事角色之间的通信。

具体中介者（Concrete Mediator）角色：具体中介者角色通过协调各同事角色实现协作行为，因此它

必须依赖于各个同事角色。

同事（Colleague）角色：每一个同事角色都知道中介者角色，而且与其他的同事角色通信的时候，一

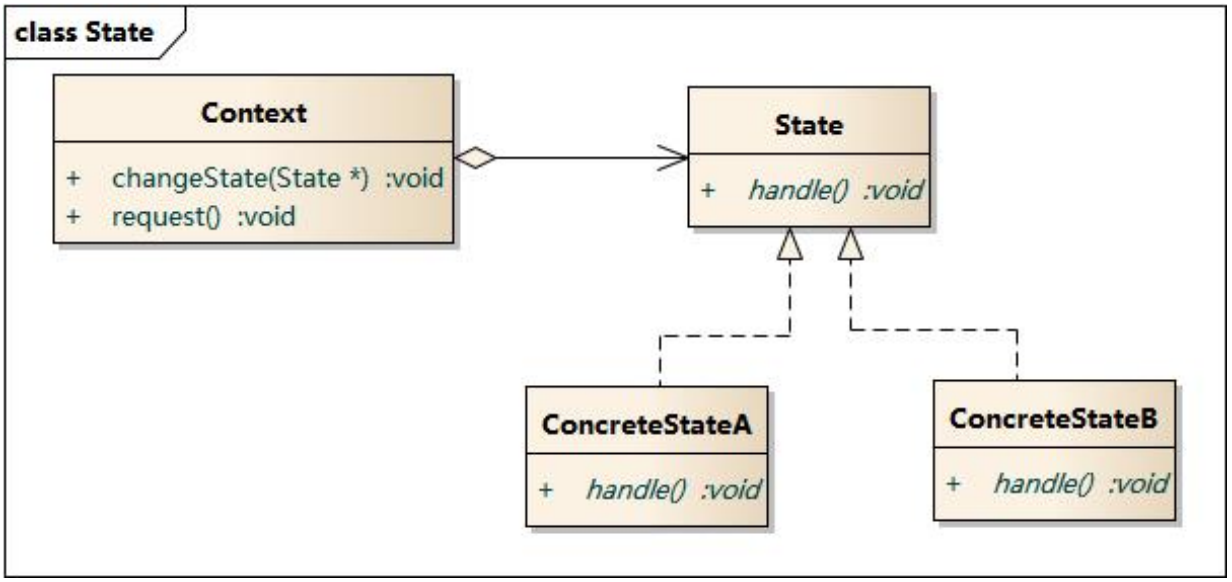
定要通过中介者角色协作。每个同事类的行为分为两种：一种是同事本身的行为，比如改变对象本身的状

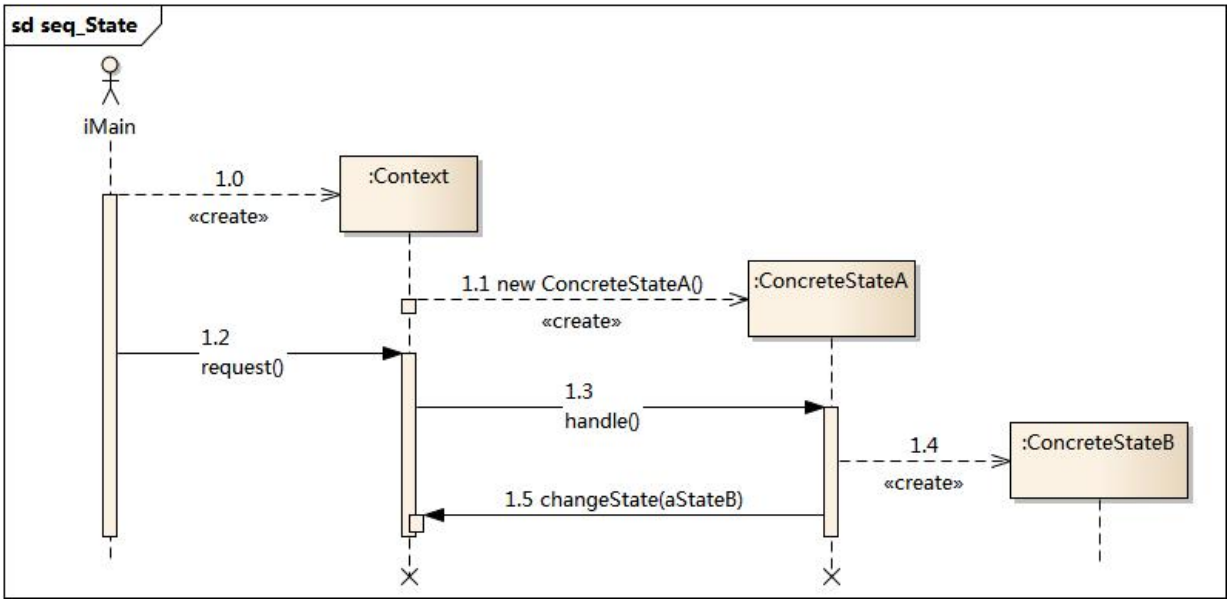
态，处理自己的行为等等，这种方法叫做自发行为(Self-Method)，与其他的同事类或中介者没有任何的依

赖；第二种是是必须依赖中介者才能完成的行为，叫做依赖方法（Dep-Method）。

状态模式

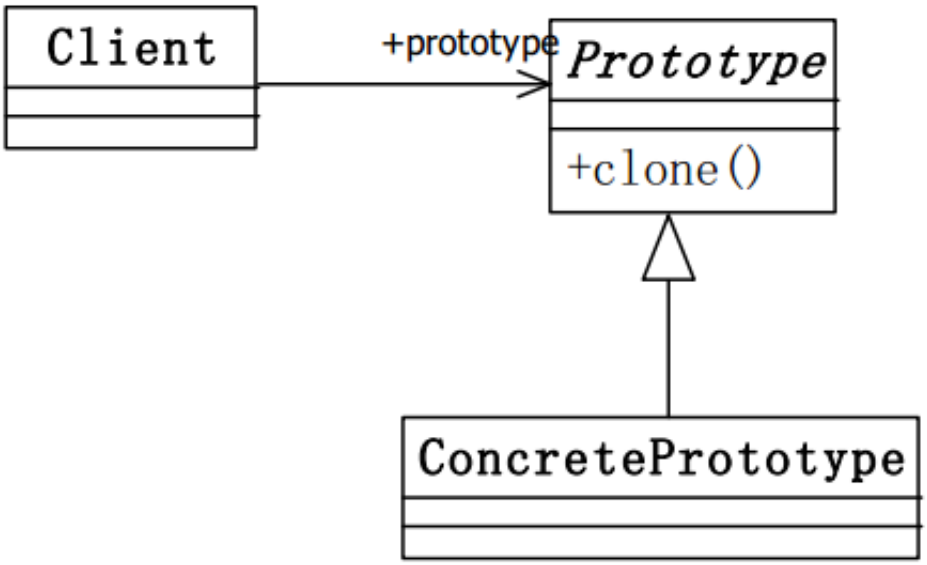
状态模式(State Pattern) ：允许一个对象在其内部状态改变时改变它的行为，对象看起来似乎修改了它的类。其别名为状态对象(Objects for States)，状态模式是一种对象行为型模式。





原型模式

不通过 new 关键字来产生一个对象，而是通过对象拷贝来实现的模式就叫做原型模式。


Client
+prototype
Prototype
+clone()
ConcretePrototype