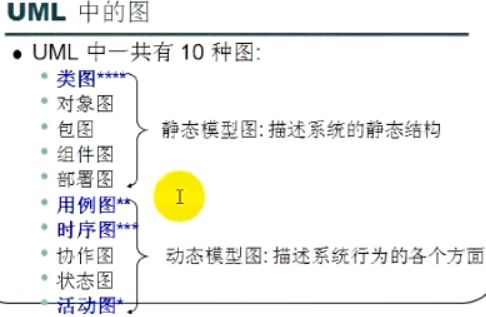
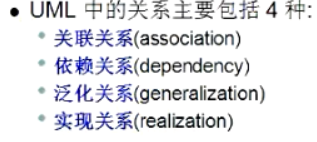
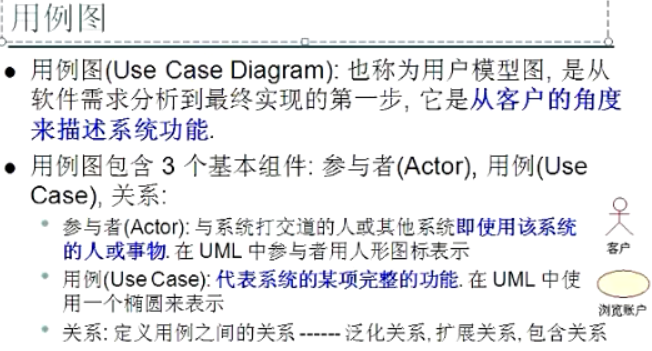
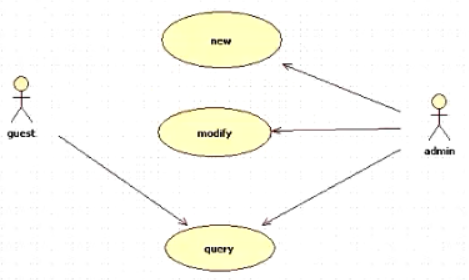
# UML(United Modeling Language，统一建模语言)：是一种基于面向对象的可视化建模语言。







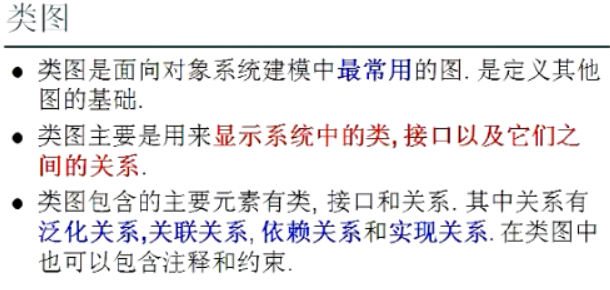
使用**StarUML**软件画图：



泛化关系：类似于父子类之间的继承关系。

包含关系：即类中的函数，对象的行为属性。

扩展关系：偶尔会执行另外一个功能。



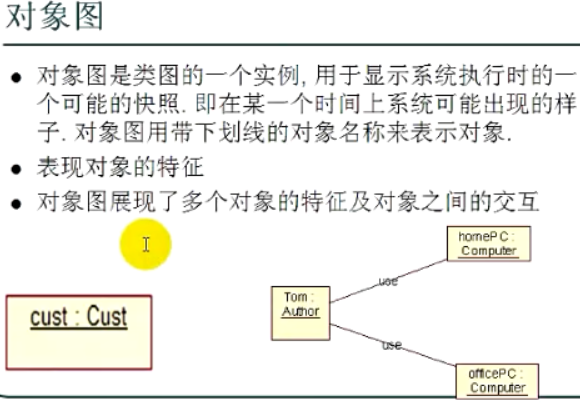
单个类，多个类，【类与类之间的关系】（**实现、泛化、依赖、关联**（一般关联、聚合、组合））。

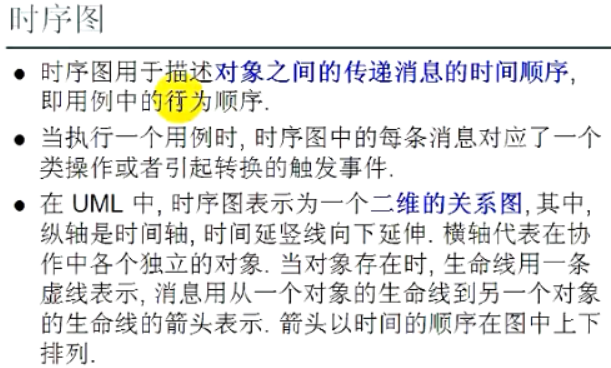
**依赖**关系：用虚线表示，一个类是另外一个类的函数参数或者函数返回值。

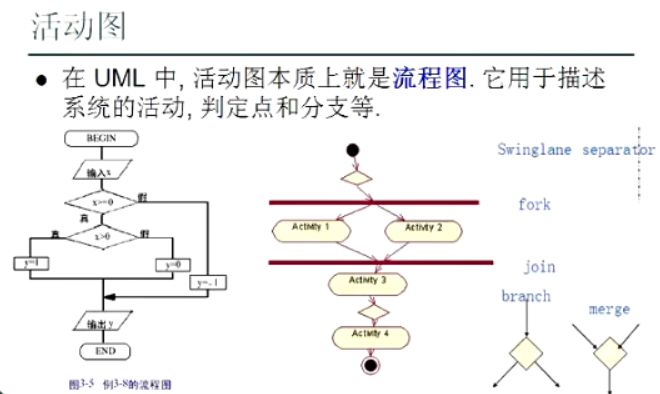
**关联**关系：用实线表示，一个类是另外一个类的成员变量。

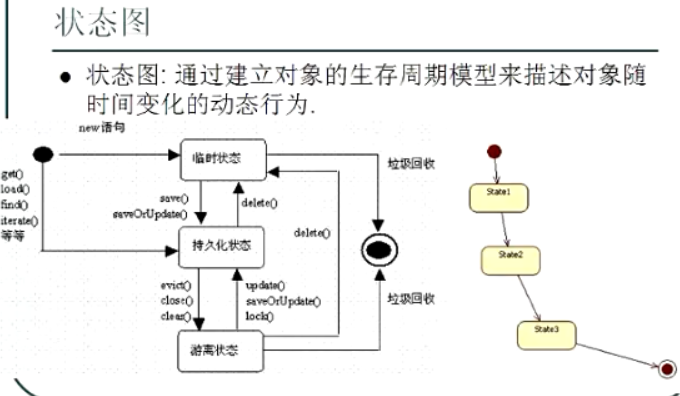
聚合关系：是**关联**关系的一种，用空菱形和实线表示，整体和部分的关系（汽车和发动机）。

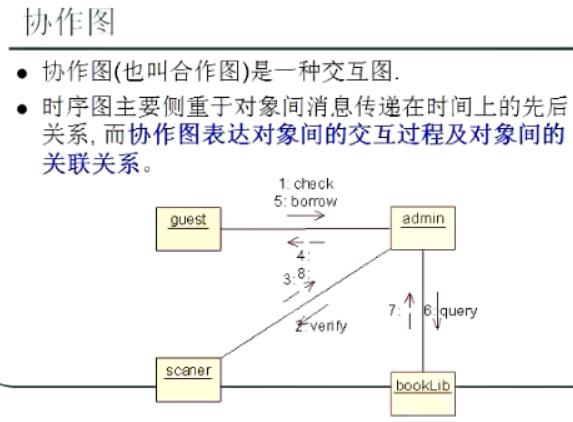
组合关系：是**关联**关系的一种，用实菱形和实线表示，生命体，整体和部分关系（人和器官）。





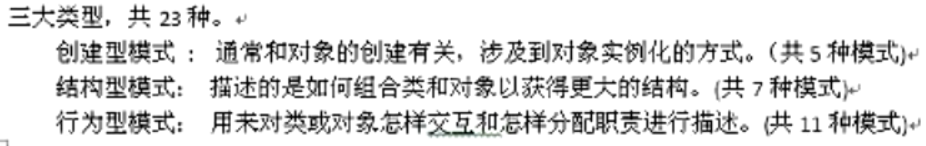


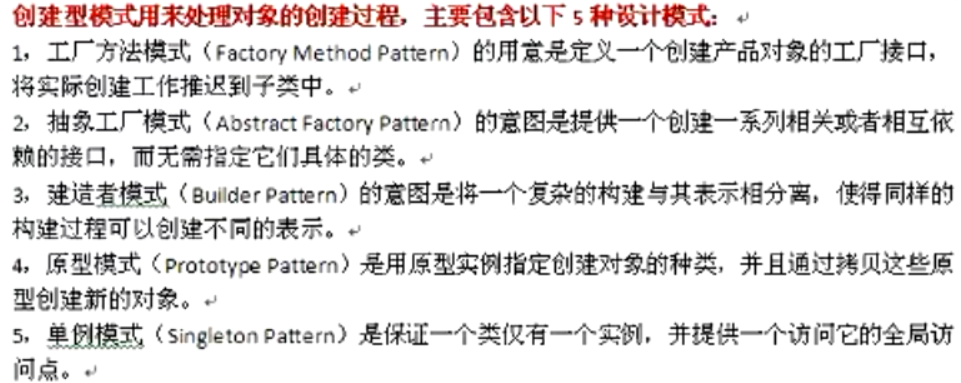


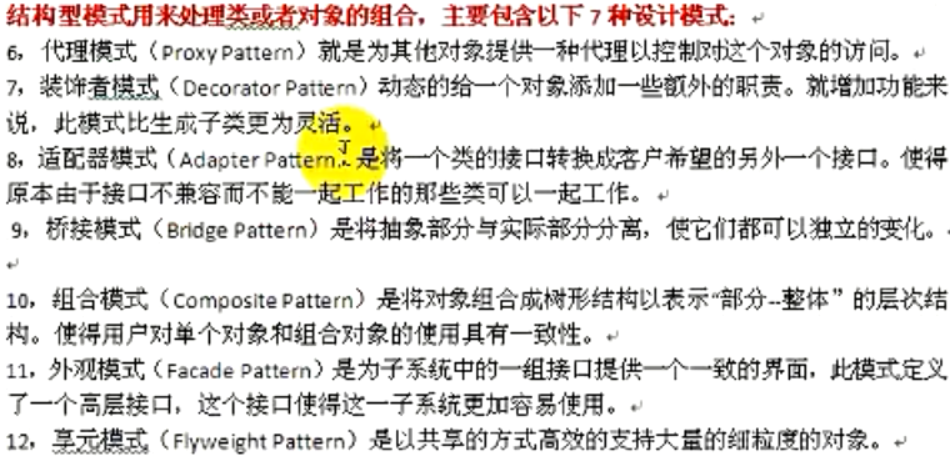


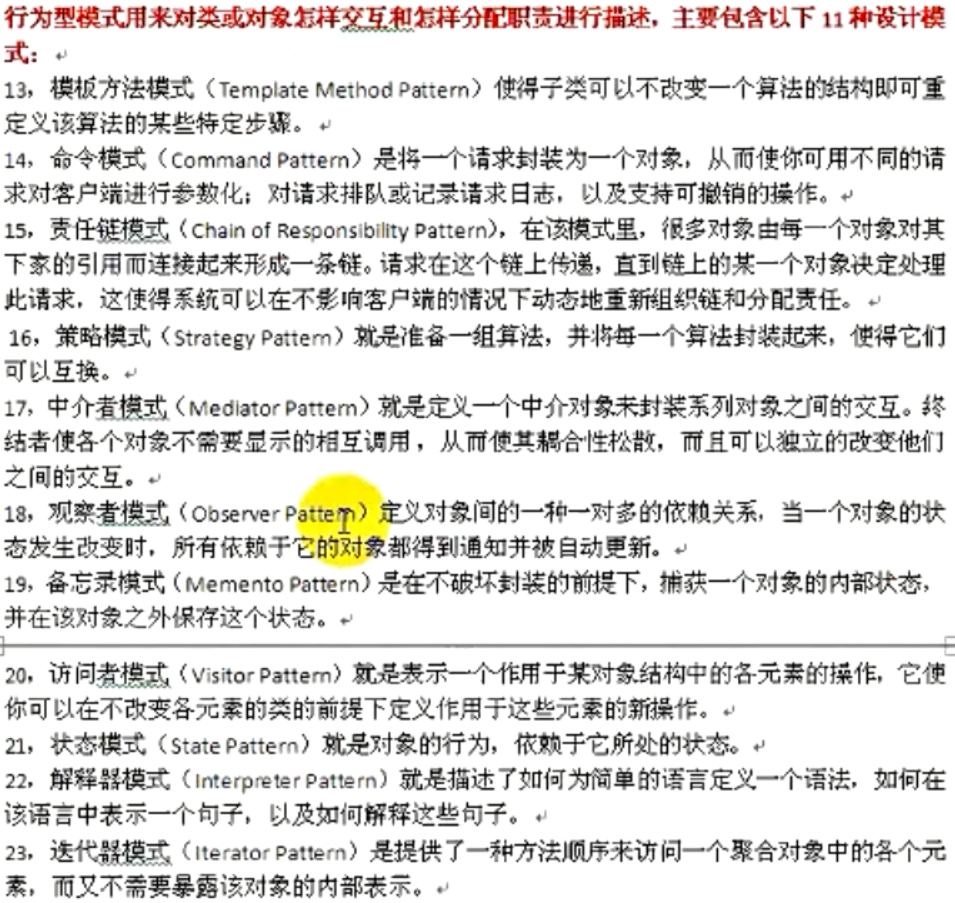
# 23种设计模式及其分类

1工厂方法模式、2抽象工厂模式、3建造者模式、4原型模式、5单例模式、6代理模式、7装饰者模式、8适配器模式、9桥接模式、10组合模式、11外观模式、12享元模式、13模板方法模式、14命令模式、15责任链模式、16策略模式、17中介者模式、18观察者模式、19备忘录模式、20访问者模式、21状态模式、22解释器模式、23迭代器模式









设计模式的基本原则：**1)开放性原则**、2)单一指责原则、**3)依赖倒置原则**、4)接口隔离原则、5)里氏替换原则、6)优先使用组合而不是继承原则、**7)迪米特法则**

# 一、创建型模式

## 1.1[单例模式](1单例模式（懒汉和饿汉）.cpp)

单例模式保证一个类只生成唯一的实例对象。即在整个程序空间，该类只存在一个实例对象。

实现：a) 构造函数私有化b) 提供一个全局的静态函数（全局访问点）c) 在类中定义一个静态指针，指向本类的变量的静态变量指针

## 1.2 [简单工厂模式](2.1简单工厂模式.cpp)和[工厂模式](2.2工厂模式.cpp)

**简单工厂模式**又叫做静态工厂方法模式。通过专门定义一个类来负责创建其他类的实例，被创建的实例通常都具有共同的父类。简单工厂模式不属于标准模式的一种。

1.工厂（Creator）角色

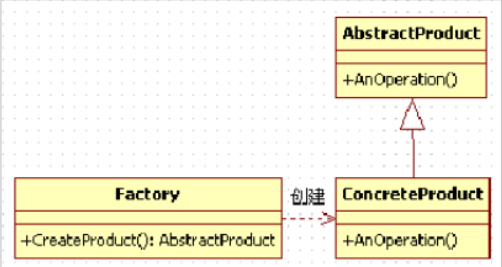
简单工厂模式的核心，实现创建所有实例的内部逻辑。工厂类可以被外界直接调用，创建返回所需的产品对象。

2.抽象（Product）角色

简单工厂模式所创建的所有对象的父类，它负责描述所有实例所共有的公共接口。

3.具体产品（ Concrete Product ）角色

简单工厂模式所创建的具体实例对象。



// 依赖 : 一个类的对象当另外一个类的函数参数或者是返回值

简单工厂模式的优缺点：

在这个模式中，工厂类是整个模式的关键所在。它包含必要的判断逻辑，能够根据外界给定的信息，决定究竟应该创建哪个具体类的对象。用户在使用时可以直接根据工厂类去创建所需的实例，而无需了解这些对象是如何创建以及如何组织的。有利于整个软件体系结构的优化。不难发现，简单工厂模式的缺点也正体现在其工厂类上，由于工厂类集中了所有实例的创建逻辑，所以“**高内聚**”方面做的并不好。另外，当系统中的具体产品类不断增多时，可能会出现要求工厂类也要做相应的修改，**扩展性并不很好。**

**工厂方法模式**又被称为多态工厂模式。其意义是定义一个创建产品对象的工厂接口，将实际创建工作推迟到子类当中。核心工厂类不再负责产品的创建，这样核心类成为一个抽象工厂角色，仅负责具体工厂子类必须实现的接口，这样进一步抽象化的好处是使得工厂方法模式可以使系统在不修改具体工厂角色的情况下引进新的产品。

1.**抽象工厂**（ Creator）角色

工厂方法模式的核心，任何工厂类都必须实现这个接口。

2.具体工厂（ Concrete Creator）角色

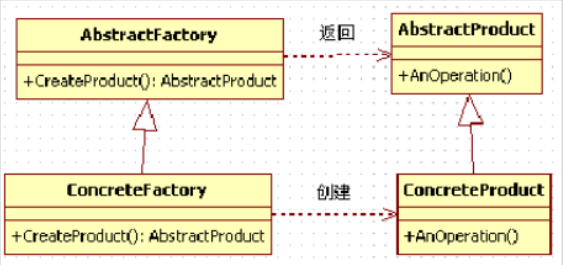
具体工厂类是抽象工厂的一个实现，负责实例化产品对象。

3.抽象（Product）角色

工厂方法模式所创建的所有对象的父类，它负责描述所有实例所共有的公共接口。

4.具体产品（ Concrete Product ）角色

工厂方法模式所创建的具体实例对象



工厂方法模式和简单工厂模式比较：

工厂方法模式与简单工厂模式在结构上的不同不是很明显。 工厂方法类的核心是一个抽象工厂类，而简单工厂模式把核心放在一个具体类上。工厂方法模式之所以有一个别名叫多态性工厂模式是因为**具体工厂类都有共同的接口**，或者有共同的抽象父类。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。而简单工厂模式在添加新产品对象后不得不修改工厂方法，扩展性不好。工厂方法模式退化后可以演变成简单工厂模式。

**“开放－封闭”通过添加代码的方式，不是通过修改代码的方式完成功能的增强。**

## 1.3 [抽象工厂模式](3抽象工厂模式.cpp)

抽象工厂模式是所有形态的工厂模式中最为抽象和最其一般性的。抽象工厂模式可以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，能够创建多个产品族的产品对象。

备注：工厂模式：要么生产香蕉、要么生产苹果、要么生产西红柿；但是不能同时生产一个产品组。抽象工厂：能同时生产一个产品族。 ===》抽象工厂存在原因

解释:具体工厂在开闭原则下，能生产香蕉 / 苹果 / 梨子; (产品等级结构)

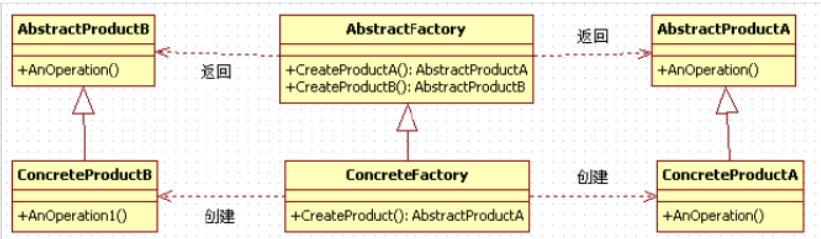
抽象工厂 :在开闭原则下，能生产：南方香蕉/苹果 /梨子（产品族）

北方香蕉 /苹果 /梨子

重要区别：

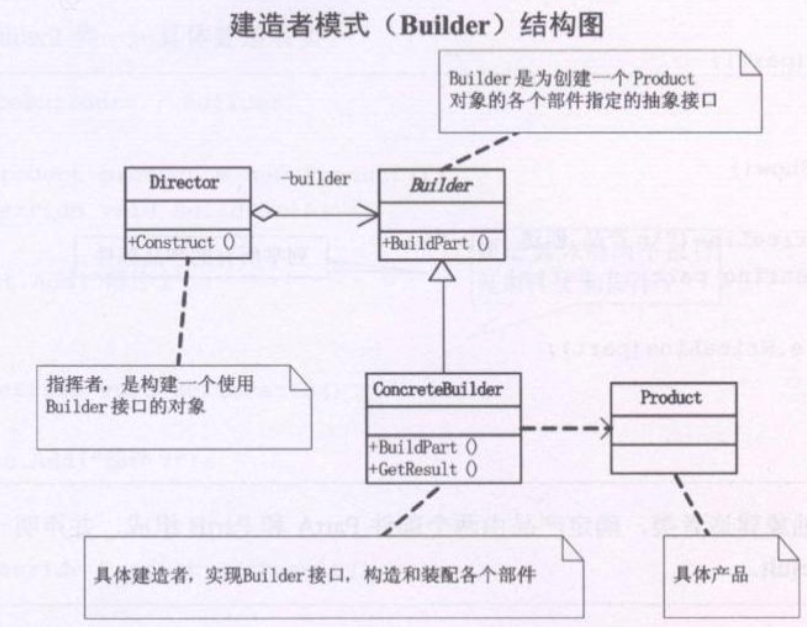
**工厂模式只能生产一个产品。 （要么香蕉、要么苹果）**

**抽象工厂可以一下生产一个产品族（里面有很多产品组成 ）（产品线）**



## 1.4 [建造者模式](4建造者模式.cpp)

**Builder 模式**是一种对象创建型模式之一，用来隐藏复合对象的创建过程，它把复合对象的创建过程加以抽象，通过子类继承和重载的方式，动态地创建具有复合属性的对象。



1)Builder ：为创建产品各个部分，统一抽象接口。

2) ConcreteBuilder ：具体的创建产品的各个部分，部分 A，部分 B，部分 C。

3) Director ：构造一个使用 Builder 接口的对象。

4) Product：表示被构造的复杂对象。

ConcreteBuilder 创建该产品的内部表示并定义它的装配过程，包含定义组成部件的类，包括将这些部件装配成最终产品的接口

适用情况：一个对象的构建比较复杂，**将一个对象的构建和对象的表示进行分离**。

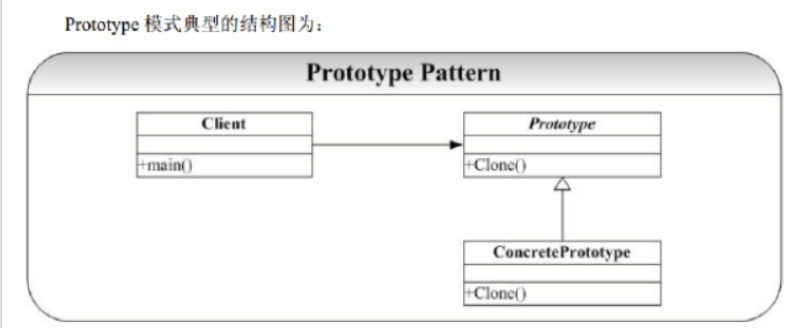
## 1.5 [原型模式](5原型模式.cpp) prototype

Prototype 模式采取复制原型对象来创建实例，其实例具有与原型一样的数据。

1）由原型对象自身创建目标对象。也就是说，对象创建这一动作发自原型对象本身。

2）目标对象是原型对象的一个克隆。也就是说，通过Prototype模式创建的对象，不仅仅与原型对象具有相同的结构，还与原型对象具有相同的值。

3）根据对象克隆深度层次的不同，有**浅度克隆与深度克隆**。

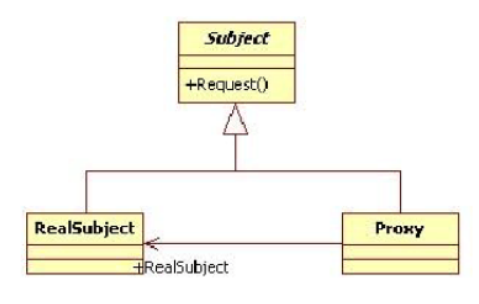


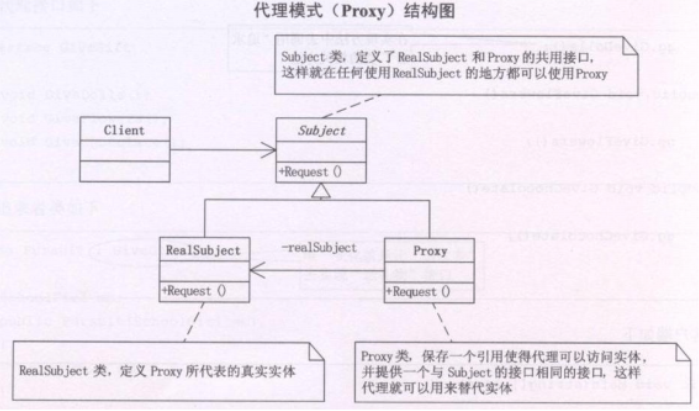
适用情况：一个复杂对象，具有**自我复制功能，统一一套接口**。

# 二、结构型模式

## 2.1 [代理模式](6代理模式.cpp)

Proxy 模式又叫做代理模式，是构造型的设计模式之一，它可以为其他对象提供一种代理（Proxy）以控制对这个对象的访问。所谓代理，是指具有与代理元 （被代理的对象） 具有相同的接口的类，客户端必须通过代理与被代理的目标类交互，而代理一般在交互的过程中（交互前后），进行某些特别的处理。





subject（抽象主题角色）：真实主题与代理主题的共同接口。

RealSubject（真实主题角色）：定义了代理角色所代表的真实对象。

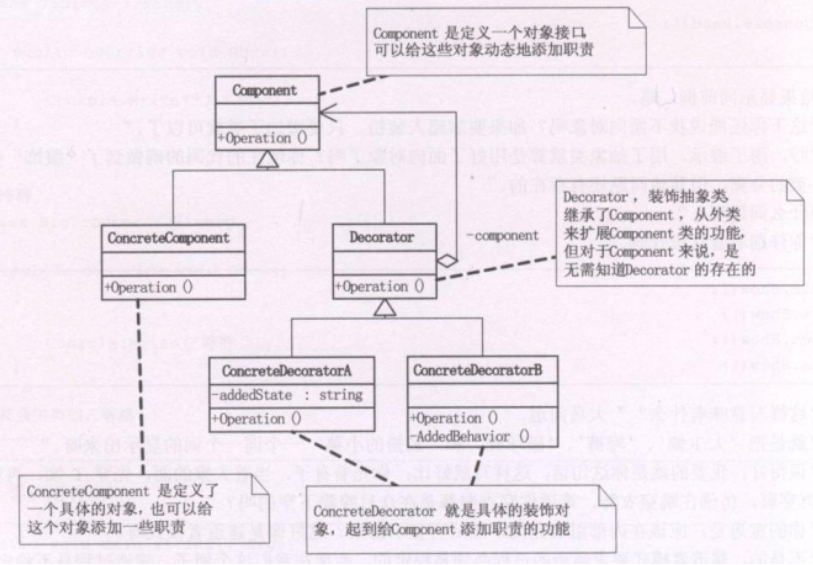
Proxy（代理主题角色）：含有对真实主题角色的引用，代理角色通常在将客户端调用传递给真是主题对象之前或者之后执行某些操作，而不是单纯返回真实的对象。

适合于：为其他对象提供一种代理以控制对这个对象的访问 。

提示： **a 中包含 b 类； a、 b 类实现协议类 protocol**

## 2.2 [装饰模式](7装饰模式.cpp)

装饰（Decorator）模式又叫做包装模式。通过一种对客户端透明的方式来扩展对象的功能，是继承关系的一个替换方案。装饰模式就是把要添加的附加功能分别放在单独的类中， 并让这个类包含它要装饰的对象，当需要执行时，客户端就可以有选择地、按顺序地使用装饰功能包装对象。



适用于：装饰者模式（ Decorator Pattern）**动态的给一个对象添加一些额外的职责**。就增加功能来说，此模式比生成子类更为灵活。

## 2.3 [适配器模式](8适配器模式.cpp) adapter

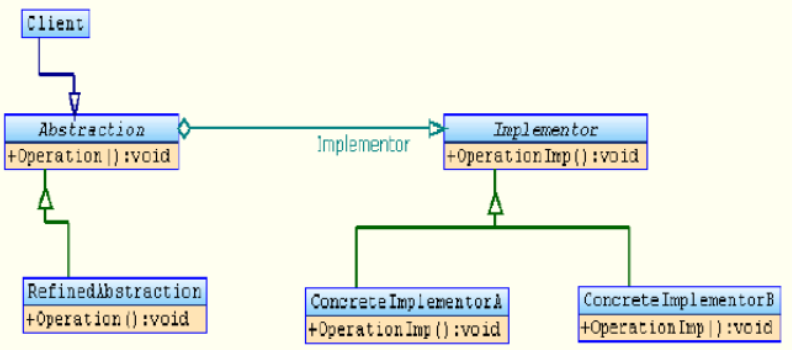
Adapter即适配器模式，通过 Adapter 模式可以改变已有类（或外部类）的接口形式。



适用于：是**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。

## 2.4[桥接模式](9桥接模式.cpp) bridge

Bridge模式又叫桥接模式，主要特点是**把抽象与行为实现分离开来**，从而可以保持各部分的独立性以及应对它们的功能扩展。

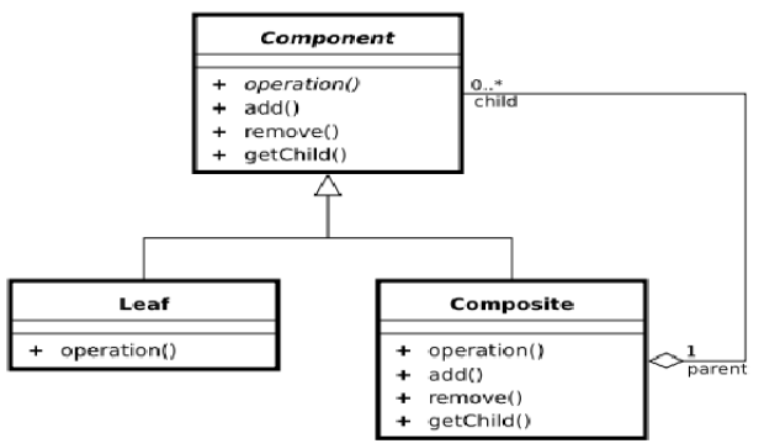


如：车安装发动机；不同型号的车，安装不同型号的发动机；

图形填颜色不同形状的图形，填充上不同的颜色。

## 2.5 [组合模式](10组合模式.cpp)

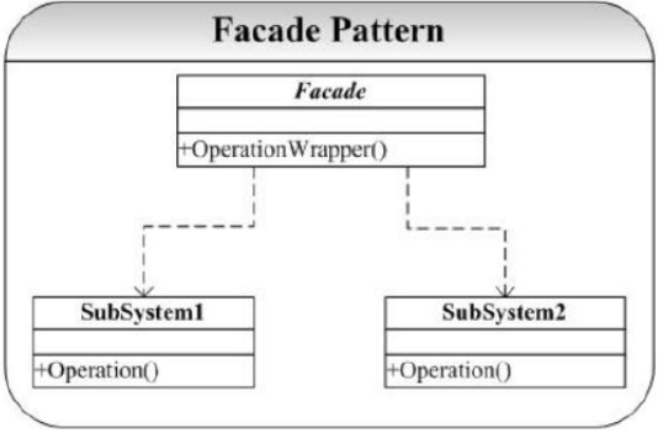
Composite 模式也叫组合模式，通过递归手段来构造树形的对象结构，并可以通过一个对象来访问整个对象树。



适用于：**单个对象和组合对象的使用具有一致性**。将对象组合成树形结构以表示“部分--整体”。

## 2.6 [外观模式](11外观模式.cpp) facade

Facade模式也叫外观模式，它为一组具有类似功能的类群，比如类库，子系统等，提供一个一致的简单的界面。



适用于：**为子系统中统一一套接口，让子系统更加容易使用**。

## 2.7 [享元模式](12享元模式.cpp) flyweight

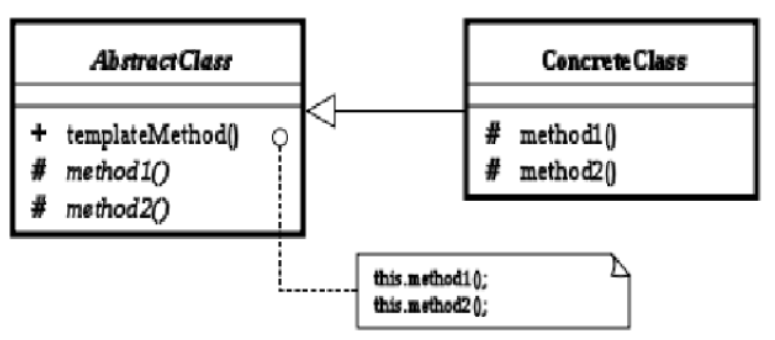
Flyweight 模式也叫享元模式，通过与其他类似对象**共享数据来减小内存**占用。



# 三、行为型模式

## 3.1 [模板模式](13模板模式.cpp) template

它把具有特定步骤算法中的某些必要的处理委让给抽象方法，通过子类继承对抽象方法的不同实现改变整个算法的行为。

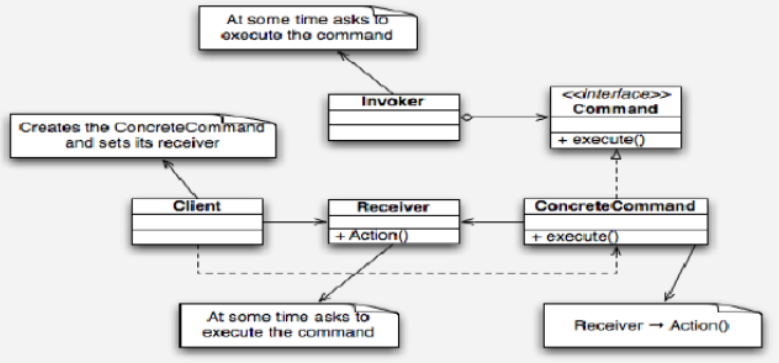


先定义好类的实现过程，抽象化具体细节，然后再对细节进行具体化。

## 3.2 [命令模式](14命令模式.cpp) command

Command 模式也叫命令模式，是行为设计模式的一种。Command模式通过被称为Command 的类封装了对目标对象的调用行为以及调用参数。

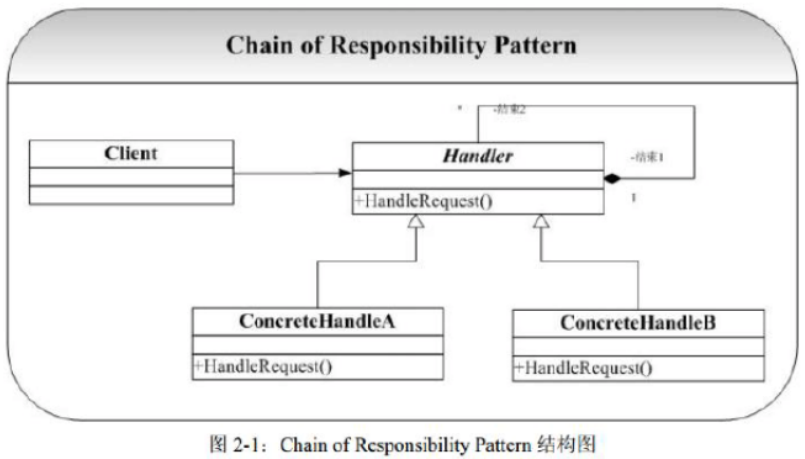
在面向对象的程序设计中， 一个对象调用另一个对象， 一般情况下的调用过程是： 创建目标对象实例；设置调用参数；调用目标对象的方法。但在有些情况下有必要使用一个专门的类对这种调用过程加以封装，我们把这种专门的类称作 command 类。整个调用过程比较繁杂，或者存在多处这种调用。这时，使用 Command 类对该调用加以封装，便于功能的再利用。调用前后需要对调用参数进行某些处理。调用前后需要进行某些额外处理，比如日志，缓存，记录历史操作等。



适用于：是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端进行参数化；对请求排队或记录请求日志，以及支持可撤销的操作。

## 3.3 [责任链模式](15责任链模式.cpp)

Chain of Responsibility（CoR） 模式也叫职责链模式或者职责连锁模式，该模式构造一系列分别担当不同的职责的类的对象来共同完成一个任务，这些类的对象之间像链条一样紧密相连，所以被称作职责链模式。



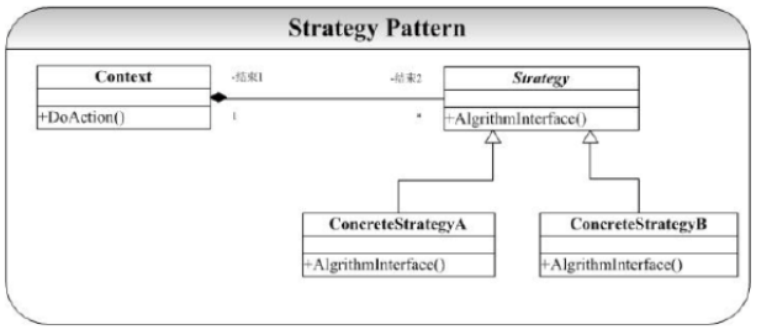
优点：1、责任的分担。 2、可以根据需要自由组合工作流程。3、类与类之间可以以松耦合的形式加以组织。

缺点：因为处理时以链的形式在对象间传递消息，实现方式不同可能会影响处理速度。

适用于：链条式处理事情。工作流程化、消息处理流程化、事物流程化。

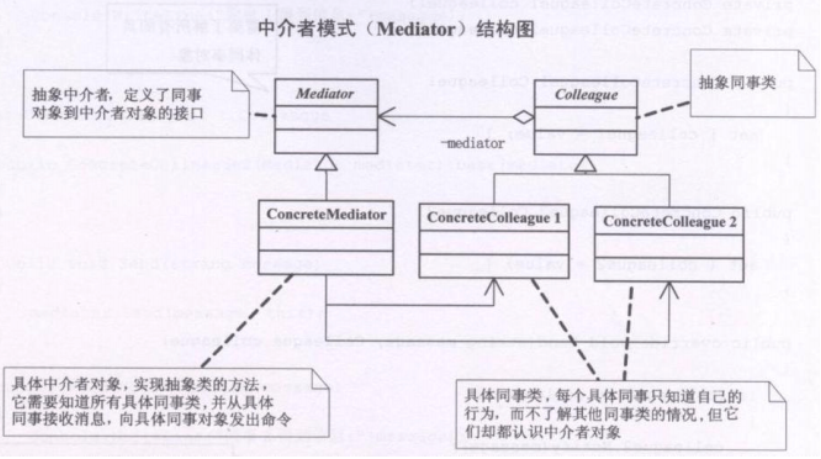
## 3.4 [策略模式](16策略模式.cpp) strategy

策略模式是对一系列的算法加以封装，为所有算法定义一个抽象的算法接口，并通过继承该抽象算法接口对所有的算法加以封装和实现， 具体的算法选择交由客户端决定（策略）。 Strategy 模式主要用来平滑地处理算法的切换。



## 3.5 [中介者模式](17中介者模式.cpp) mediator

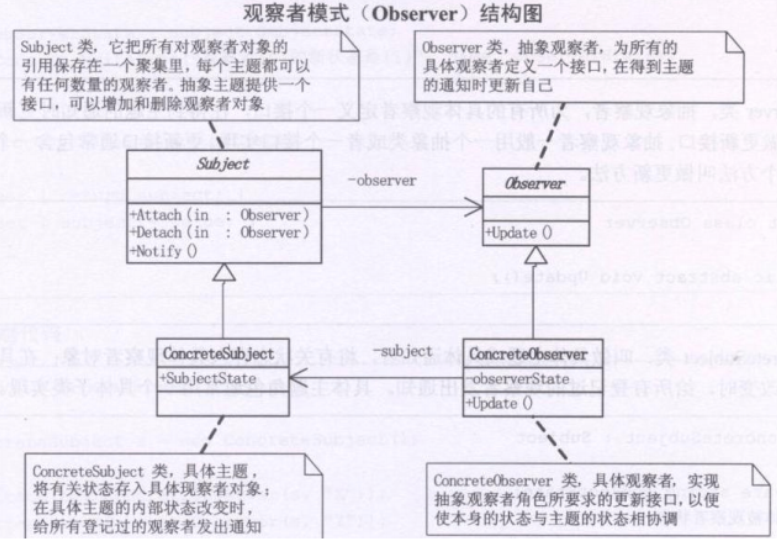
Mediator 模式也叫中介者模式，类之间的交互行为被统一放在Mediator的对象中，对象通过Mediator对象同其他对象交互，Mediator对象起着控制器的作用。



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示的相互引用，从而降低耦合；而且可以独立地改变它们之间的交互。

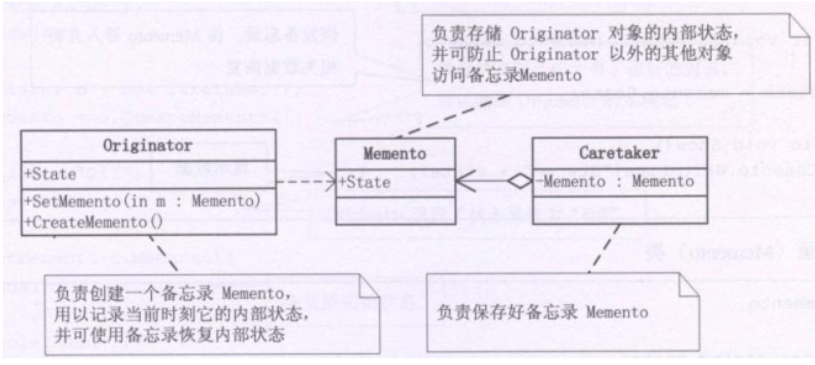
## 3.6 [观察者模式](18观察者模式.cpp) observer

Observer模式是行为模式之一，它的作用是当一个对象的状态发生变化时，能够自动通知其他关联对象，自动刷新对象状态。Observer模式提供给关联对象一种同步通信的手段，使某个对象与依赖它的其他对象之间保持状态同步。



## 3.7 [备忘录模式](19备忘录模式.cpp) memento

Memento 模式也叫备忘录模式，它的作用是保存对象的内部状态，并在需要的时候（ undo/rollback ）恢复对象以前的状态。

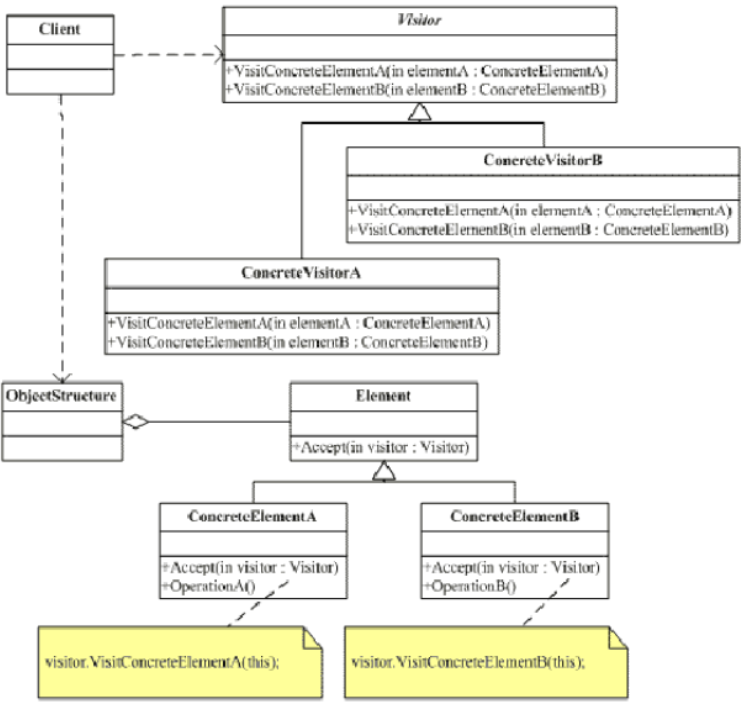


Originator （原生者）：需要被保存状态以便恢复的那个对象。Memento （备忘录）：该对象由 Originator 创建，主要用来保存 Originator 的内部状态。Caretaker（管理者）:负责在适当的时间保存 /恢复 Originator 对象的状态。

适用于：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，这样就可以将以后的对象状态恢复到先前保存的状态。适用于功能比较复杂的，但需要记录或维护属性历史的类；或者需要保存的属性只是众多属性中的一小部分时 Originator 可以根据保存的 Memo 还原到前一状态。

## 3.8 [访问者模式](20访问者模式.cpp) visitor

Visitor 模式也叫访问者模式，它分离对象的数据和行为，使用 Visitor模式，可以不修改已有类的情况下，增加新的操作角色和职责。



抽象访问者（ Visitor）角色：声明了一个或者多个访问操作，形成所有的具体元素角色必须实现的接口。具体访问者（ ConcreteVisitor ）角色。抽象节点（ Element）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（ ConcreteElement ）角色。结构对象（ ObiectStructure ）角色：有如下的一些责任，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让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，如列（List）或集合（Set）。

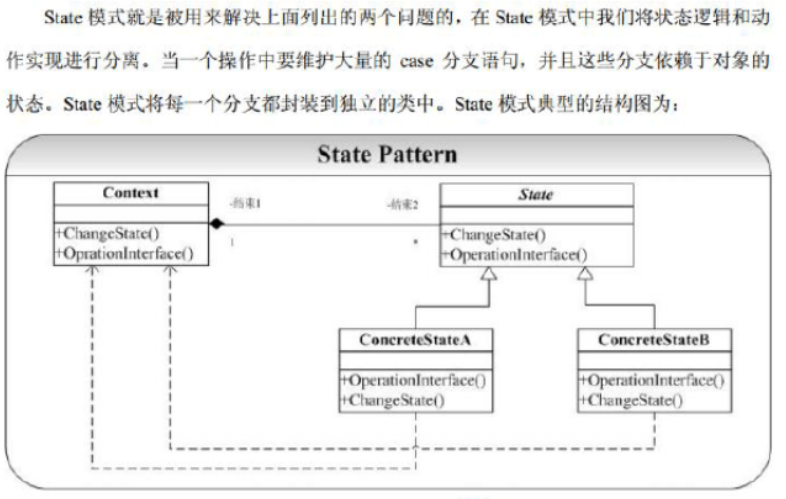
适用于：把数据结构和作用于数据结构上的操作进行解耦合;适用于数据结构比较稳定的场合。

案例需求：比如有一个公园，有一到多个不同的组成部分；该公园存在多个访问者：清洁工A负责打扫公园的A部分，清洁工B负责打扫公园的B部分，公园的管理者负责检点各项事务是否完成，上级领导可以视察公园等等。也就是说，对于同一个公园，不同的访问者有不同的行为操作，而且访问者的种类也可能需要根据时间的推移而变化（行为的扩展性）。

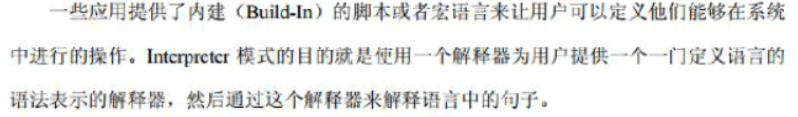
## 3.9 [状态模式](21状态模式.cpp) state

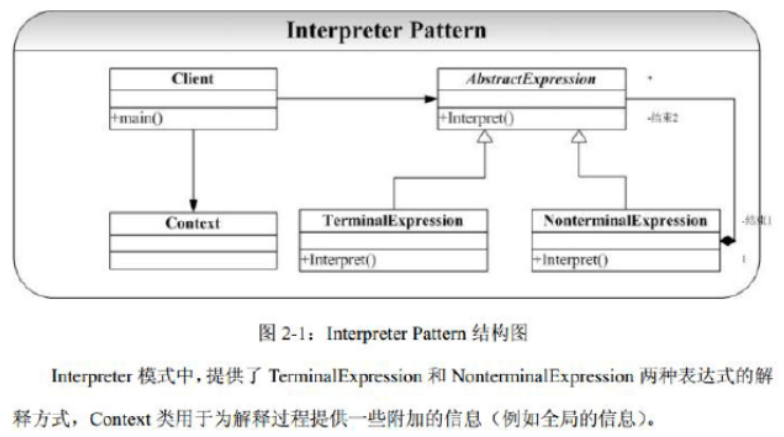
State 模式也叫状态模式，允许通过改变对象的内部状态而改变对象的行为，这个对象表现得就好像修改了它的类一样。状态模式主要解决的是当控制一个对象状态转换的条件表达式过于复杂时的情况。把状态的判断逻辑转译到表现不同状态的一系列类当中，可以把复杂的判断逻辑简化。

每个人、事物在不同的状态下有不同的行为，而一个状态会在不同的表现下转换到另一个状态。



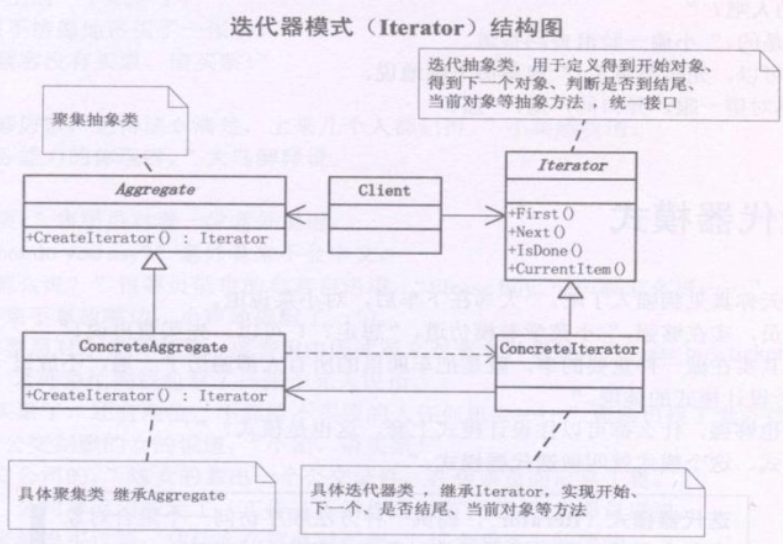
## 3.10 [解释模式](22解释模式.cpp) interpret





## 3.11 [迭代器模式](23迭代器模式.cpp) iterator

Iterator 模式也叫迭代模式，它把对容器中包含的内部对象的访问委让给外部类，使用Iterator（遍历）按顺序进行遍历访问的设计模式。



在迭代器中持有一个集合的引用；所以通过迭代器，就可以访问集合