面向对象技术

面向对象的基本概念

- 面向对象=对象+分类+继承+通过消息的通信
- 1.对象的概念
- 在计算机系统中，对象是指一组属性以及这组属性上的专用操作的封装体。属性可以是一些数据，也可以是另一个对象。每个对象都有它自己的属性值，表示该对象的状态，用户只能看见对象封装界面上的信息，对象的内部实现对用户是隐蔽的。封装目的是使对象的使用者和生产者分离，使对象的定义和实现分开。一个对象通常可由3部分组成，分别是对象名、属性和操作（方法）。

面向对象的基本概念

- 2.类的概念
- 类是一组具有相同属性和相同操作的对象的集合。一个类中的每个对象都是这个类的一个实例（instance）。在分析和设计时，我们通常把注意力集中在类上，而不是具体的对象上。通常把一个类和这个类的所有对象称为类及对象或对象类。
- 一个类通常可由3部分组成，分别是类名、属性和操作（方法）。每个类一般都有实例，没有实例的类是抽象类。抽象类不能被实例化，也就是不能用new关键字去产生对象，抽象方法只需声明，而不需实现。抽象类的子类必须覆盖所有的抽象方法后才能被实例化，否则这个子类还是个抽象类。
  - 在Java语言中使用abstract class来定义抽象类。

面向对象的基本概念

- 3.继承
- 继承关系表示了对象间"is-a"的关系，即子类（派生类）是父类（超类、基类）的一种。继承是在某个类的层次关联中不同的类共享属性和操作的一种机制。一个父类可以有多个子类，这些子类都是父类的特例。父类描述了这些子类的公共属性和操作，子类还可以定义它自己的属性和操作。一个子类只有唯一的父类，这种继承称为单一继承。一个子类有多个父类，可以从多个父类中继承特性，这种继承称为多重继承。对于两个类A和B,如果A类是B类的子类，则说B类是A类的泛化。
-

面向对象的基本概念

- 继承是面向对象方法区别于其他方法的一个核心思想。继承机制实现的是父类和子类之间的关系，子类又能够作为其他类的父类，因此组织而成一个层次结构。在程序中，凡是引用父类对象的地方都可以使用子类对象来代替。

-

面向对象的基本概念

- 4、类的定义
- 类实际上就是由一组描述对象属性或状态的数据项和作用在这些数据项上的操作（或称为方法、成员函数等）构成的封装体。类的定义由关键字class打头，后跟类名，类名之后的括号内是类体，最后以分号";"结束。
- 类与C语言中的结构体大致相似，其不同之处在于类中规定了哪些成员可以访问，哪些成员不可以访问。这些都通过访问指明符予以说明。访问指明符有三种，分别是private、protected和public.
- （1）private 成员私有化，除了该类的成员函数以外，谁也不能访问该类

面向对象的基本概念

- （2）public 成员公有化，程序中的所有函数（不管是类内定义的还是类外定义的），都可以访问这些成员。
- （3）protected 成员受限保护，只有该类及该类的子类的成员函数才能够访问。在类的成员定义中，如果没有指明符，则系统默认为private.
- 要注意的是在C++语言中，一个类的成员是可以访问该类的所有成员的。
- 继承的限定也有三种，分别是private（私有继承）、protected（保护继承）和public（公有继承）。

- 在public继承时，派生类（子类）的public、private、protected型的成员函数可以访问基类中的public成员和protected成员，派生类的对象仅可访问基类中的public成员。
- 在private继承时，派生类的public、private、protected型的成员函数可以访问基类中的public成员和protected成员，但派生类的对象不可访问基类中的任何成员。
- 在protected继承时，派生类的public、private、protected型的成员函数可以访问基类中的public成员和protected成员，但派生类的对象不可访问基类中的任何成员。
- 使用class关键字定义类时，默认的继承方式是private,也就是说，当继承方式为private继承时，可以省略private.

面向对象的基本概念

- 另外，类的成员有动态和静态之分，默认情况下，为**动态成员**。如果在成员说明前加上*static*,则说明是静态成员。静态成员与对象的实例无关，只与类本身有关。它们用来实现类要封装的功能和数据，但不包括特定对象的功能和数据。
- **静态成员**包括静态方法和静态属性。 静态属性包含在类中要封装的数据，可以由所有类的实例共享。实际上，除了属于一个固定的类并限制访问方式外，类的静态属性非常类似于函数的全局变量。

面向对象的基本概念

- 5、多态性
- 多态性是指同一个操作作用于不同的对象可以有不同的解释，产生不同的执行结果。与多态性密切相关的一个概念就是动态绑定。传统的程序设计语言把过程调用与目标代码的连接放在程序运行前进行，称为静态绑定。而动态绑定则把这种连接推迟到运行时才进行。在运行过程中，当一个对象发送消息请求服务时，要根据接收对象的具体情况将请求的操作与实现的方法连接，即动态绑定。
- 在使用多态技术时，用户可以发送一个通用的消息，而实现的细节则由接收对象自行决定，这样同一消息就可以调用不同的方法。
- 多态有多种不同的形式，其中参数多态和包含多态称为通用多态，过载多态和强制多态成为特定多态。
-

面向对象的基本概念

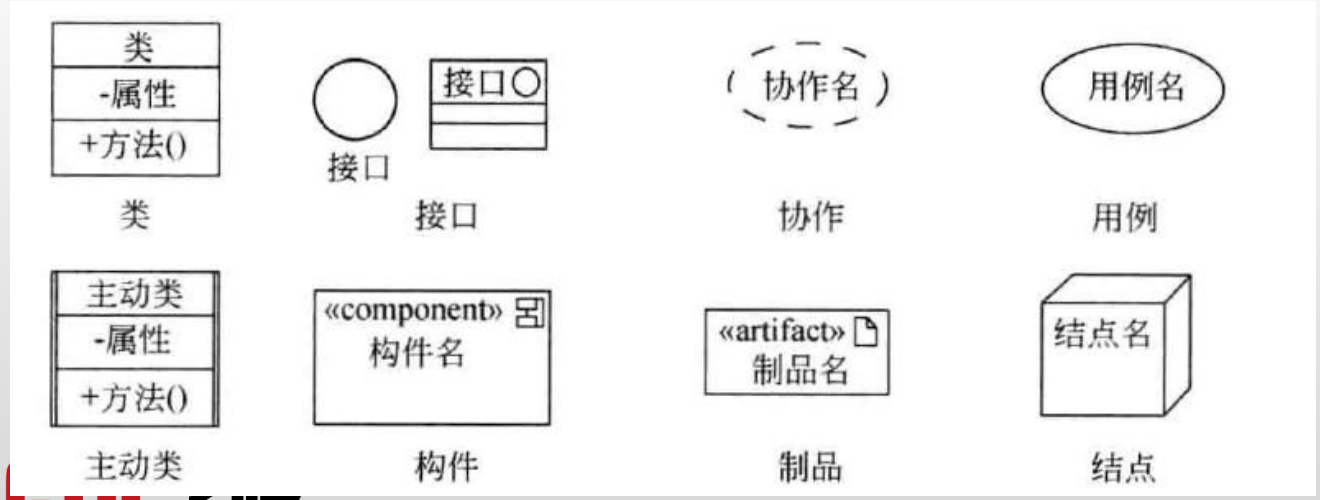
- （1）参数多态应用比较广泛，被称为最纯的多态。这是因为同一对象、函数或过程能以一致的形式用于不同的类型。
- （2）包含多态最常见的例子就是子类型化，即一个类型是另一类型的子类型。
- （3）过载多态是同一变量被用来表示不同的功能，通过上下文以决定一个类所代表的功能。即通过语法对不同语义的对象使用相同的名，编译能够消除这一模糊。
- （4）强制多态是通过语义操作把一个变元的类型加以变换，以符合一个函数的要求，如果不做这一强制性变换将出现类型错误。类型的变换可在编译时完成，通常是隐式地进行，当然也可以在动态运行时完成。

统一建模语言

- Unified Modeling Language (UML)又称统一建模语言或标准建模语言，它是一个支持模型化和软件系统开发的图形化语言，为软件开发的所有阶段提供模型化和可视化支持，包括由需求分析到规格，到构造和配置。

- UML中有4种事务：

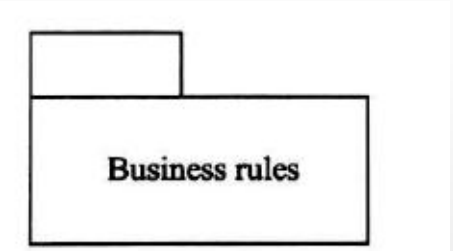
（1）结构事务：名词、静态部分、物理元素。



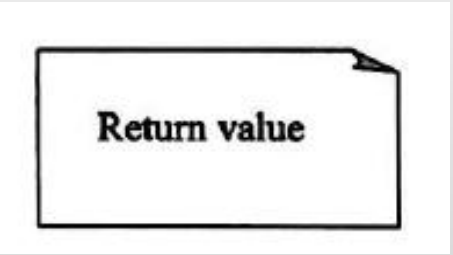
统一建模语言

- (2) 行为事务：动词、动态部分、行为。

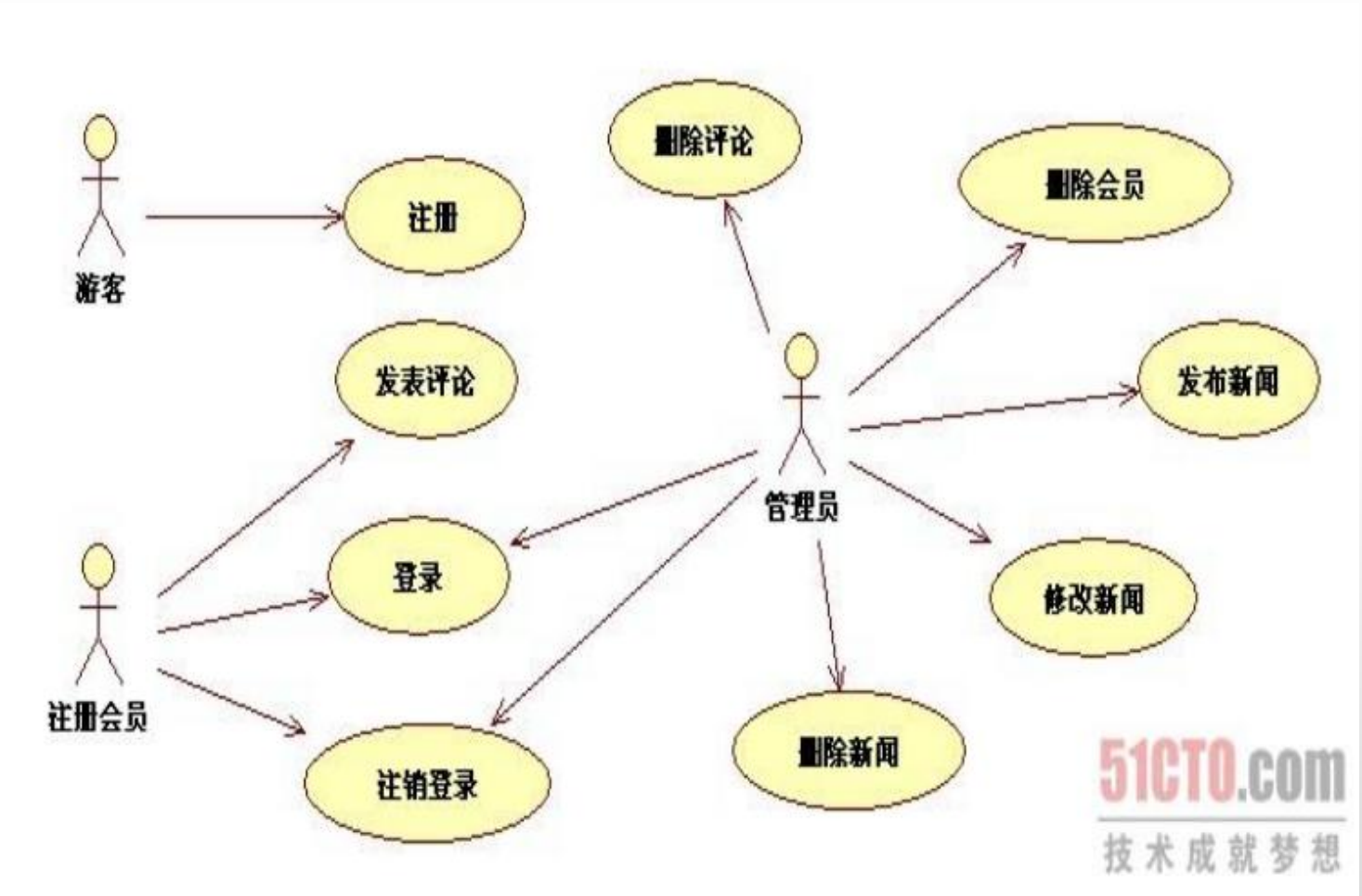
(3) 分组事务：包。



- (4) 注释事务：注解。



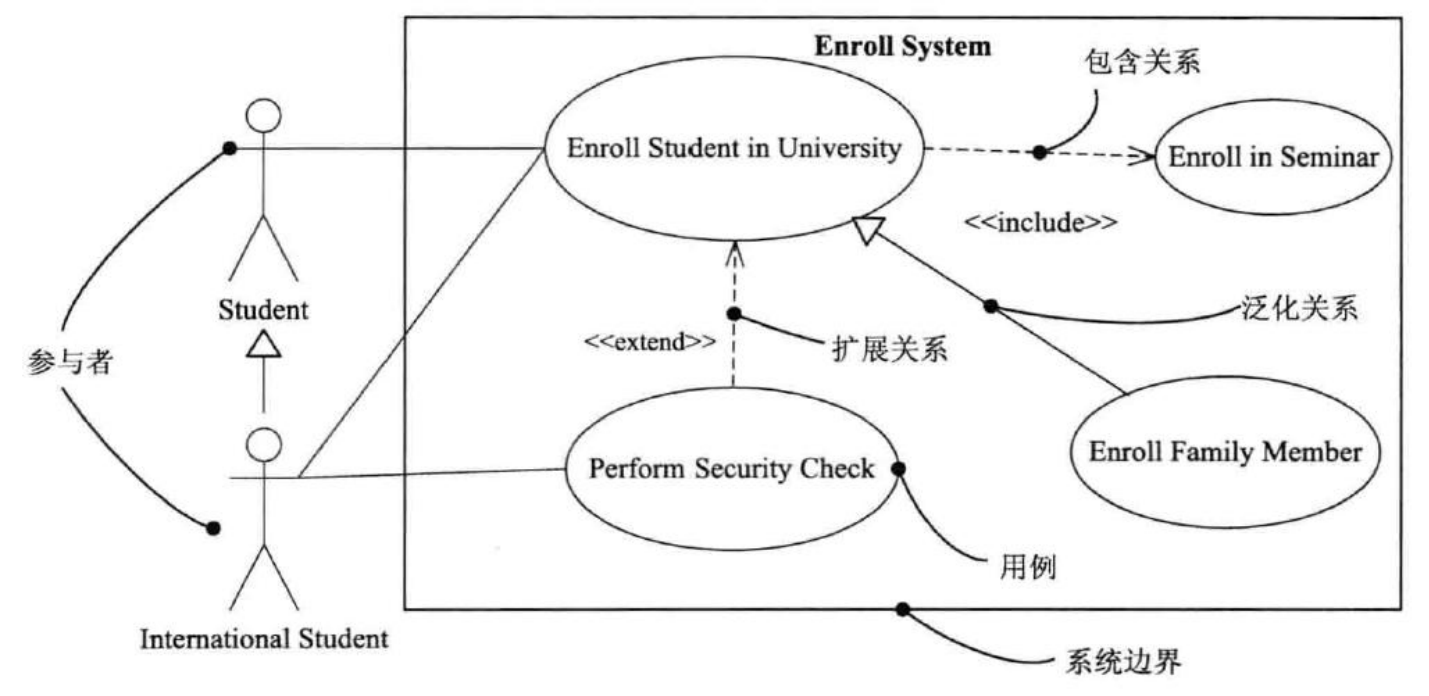
用例图 ：


游客
注册
删除评论
删除会员
发表评论
发布新闻
管理员
登录
修改新闻
注册会员
注销登录
删除新闻
51CTO.com
技术成就梦想

用例图

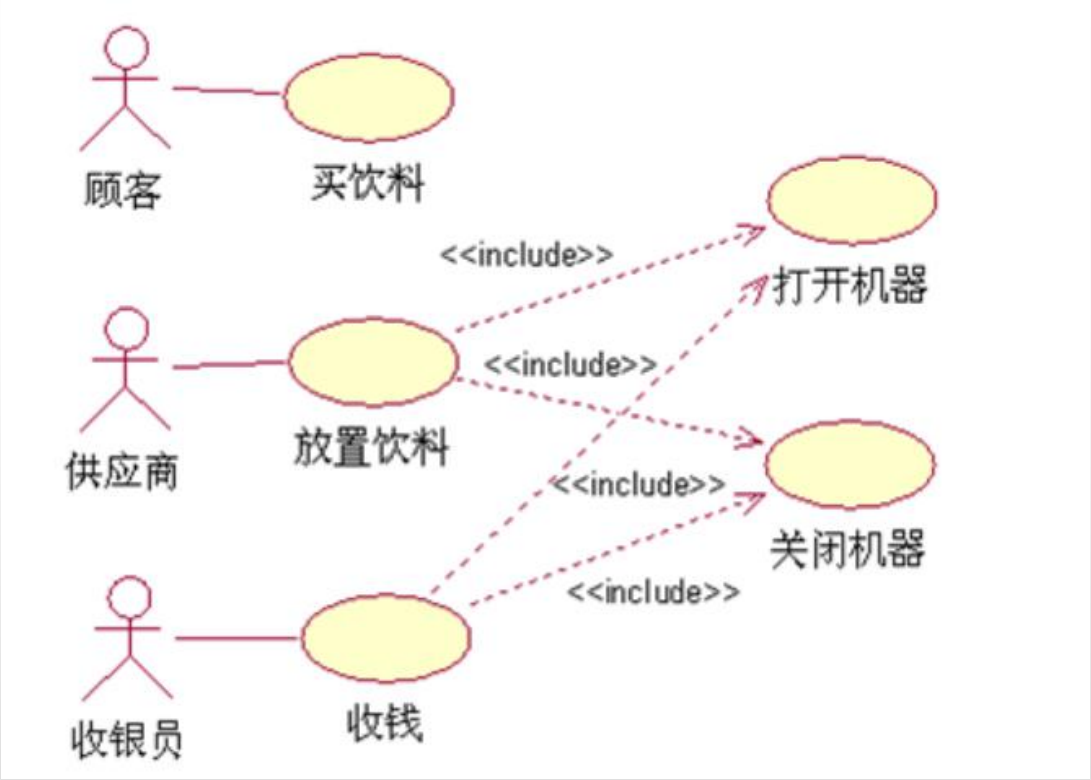
- 用例图是指由参与者（Actor）、用例（Use Case），边界以及它们之间的关系构成的用于描述系统功能的视图。用例图（User Case）是外部用户（被称为参与者）所能观察到的系统功能的模型图。用例图是系统的蓝图，用于需求分析阶段。用例图呈现了一些参与者，一些用例，以及它们之间的关系，主要用于对系统、子系统或类的功能行为进行建模。

用例图 ：


Enroll System
包含关系
Enroll Student in University
Enroll in Seminar
<<include>>
泛化关系
<<extend>>
扩展关系
Perform Security Check
Enroll Family Member
用例
Student
参与者
International Student
系统边界

用例之间的关系

- （1）包含(include)关系

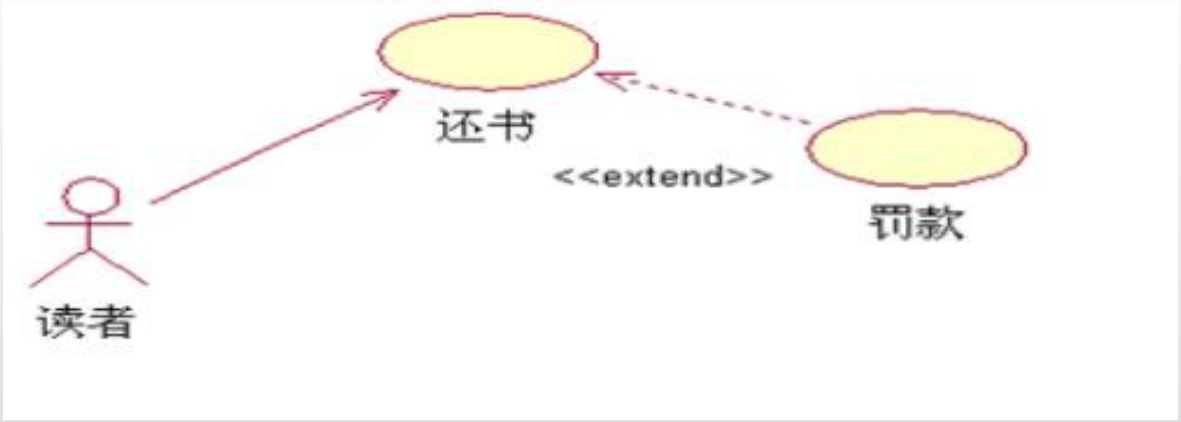


用例之间的关系

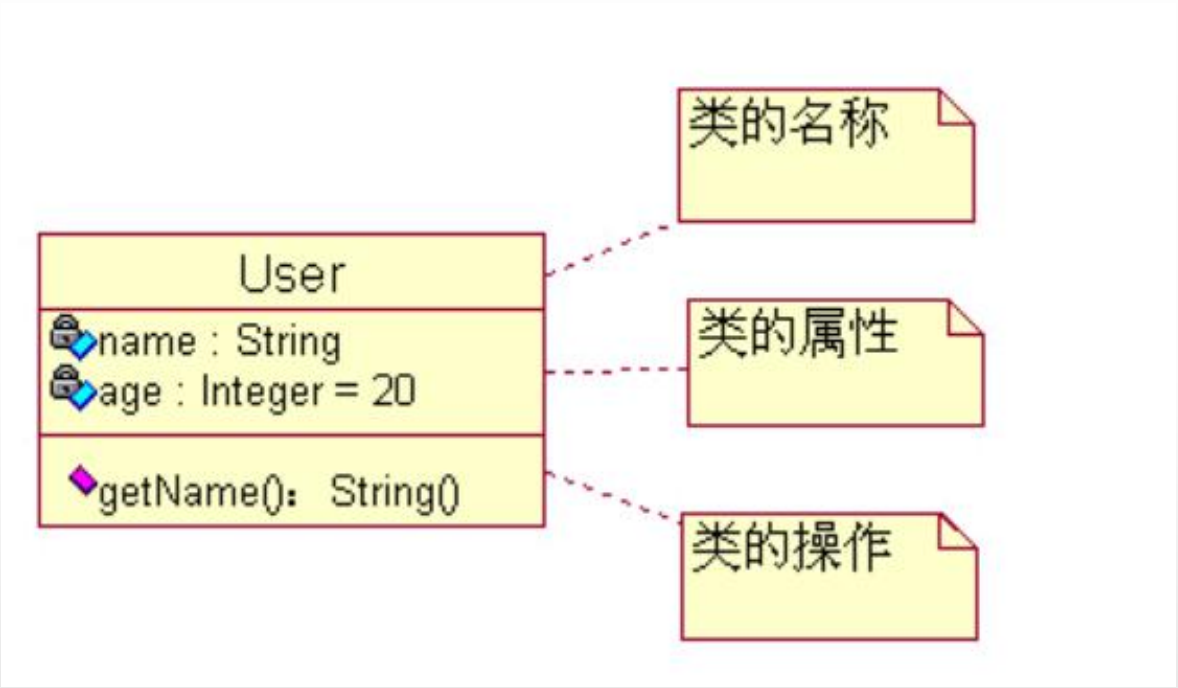
- （1）包含(include)关系
- 当两个或多个用例中共用一组相同的动作，这时可以将这组相同的动作抽出来作为一个独立的子用例，供多个基用例所共享。因为子用例被抽出，基用例并非一个完整的用例，所以include关系中的基用例必须和子用例一起使用才够完整，子用例也必然被执行。include关系在用例图中使用带箭头的虚线表示(在线上标注<<include>>)，箭头从基用例指向子用例。

用例之间的关系

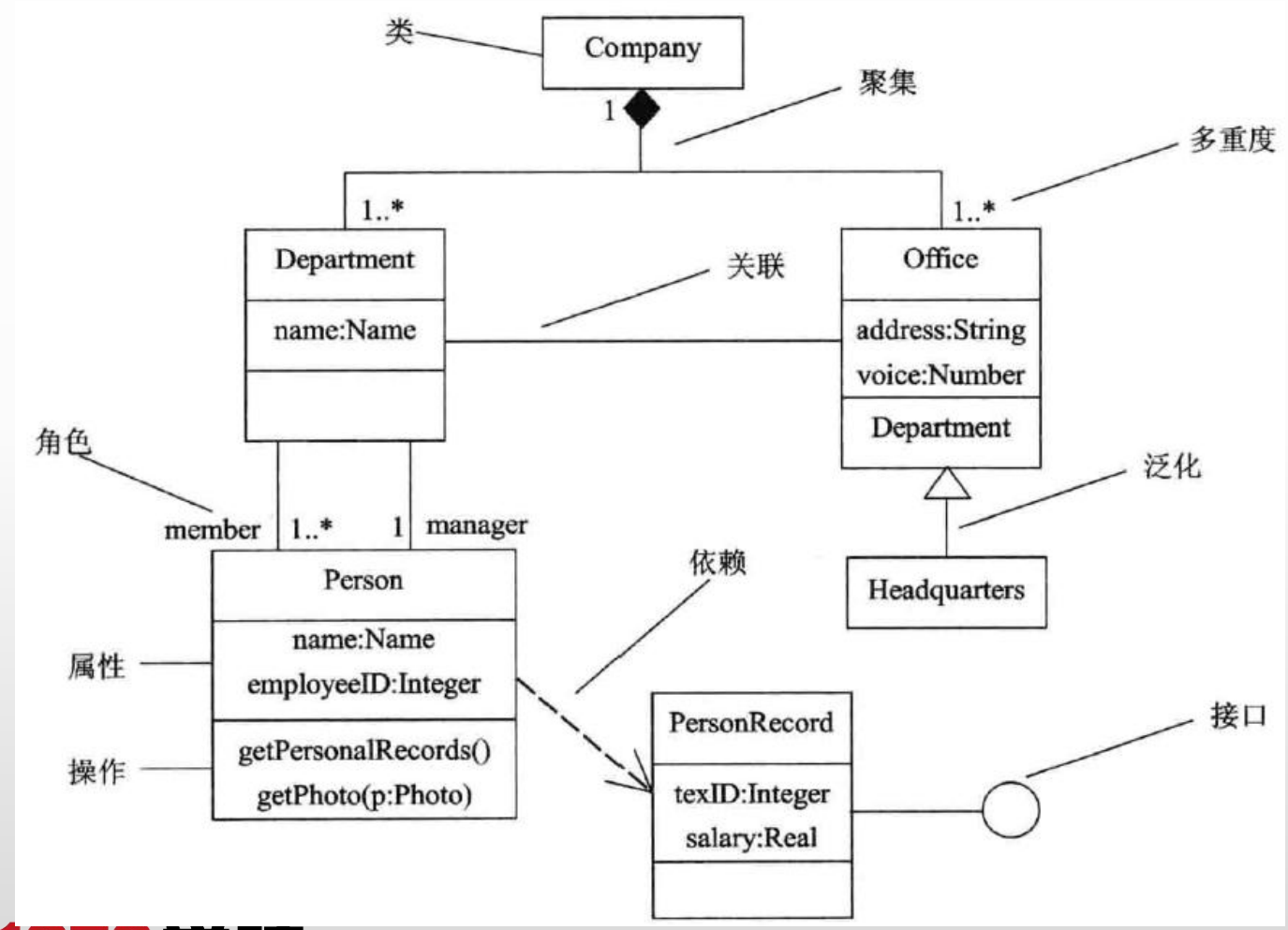
- （2）扩展(extend)关系
- 是对基用例的扩展，基用例是一个完整的用例，即使没有子用例的参与，也可以完成一个完整的功能。 extend关系在用例图中使用带箭头的虚线表示(在线上标注<<extend>>)，箭头从子用例指向基用例。



类图


User
name : String
age : Integer = 20
getName(): String()
类的名称
类的属性
类的操作

类图


类
Company
聚集
1
多重度
1..*
1..*
Department
name:Name
关联
Office
address:String
voice:Number
Department
角色
泛化
member
1..*
1
manager
Person
name:Name
employeeID:Integer
getPersonalRecords()
getPhoto(p:Photo)
依赖
Headquarters
属性
操作
PersonRecord
texID:Integer
salary:Real
接口

类图

- 类图（Class diagram）展现了一组对象、接口、协作和它们之间的关系。类图是静态视图。
- 类图中包括：
- （1）类
- （2）接口
- （3）协作
- （4）依赖、泛化和关联关系

类图

- **使用类图的情况：**
- **（1）对系统的静态设计建模**
- **（2）对简单的协作建模**
- **（3）对逻辑数据库模式建模**

类图

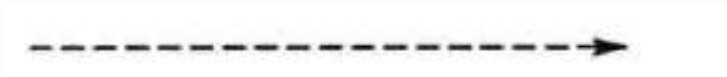
- 类的分类：
- (1) 实体类：实体类对应系统需求中的每个实体，它们通常需要保存在永久存储体中，一般使用数据库表或文件来记录，实体类既包括存储和传递数据的类，还包括操作数据的类。实体类来源于需求说明中的名词，如学生、商品等。

类的分类

- (2) 控制类：控制类用于体现应用程序的执行逻辑，提供相应的业务操作，将控制类抽象出来可以降低界面和数据库之间的耦合度。控制类用于对一个或几个用例所特有的控制行为进行建模。控制对象（控制类的实例）通常控制其他对象，因此它们的行为具有协调性质。控制类将用例的特有行为进行封装
- (3) 边界类：边界类用于对外部用户与系统之间的交互对象进行抽象，主要包括界面类，如对话框、窗口、菜单等。

类图中的关系

- 依赖(Dependency)



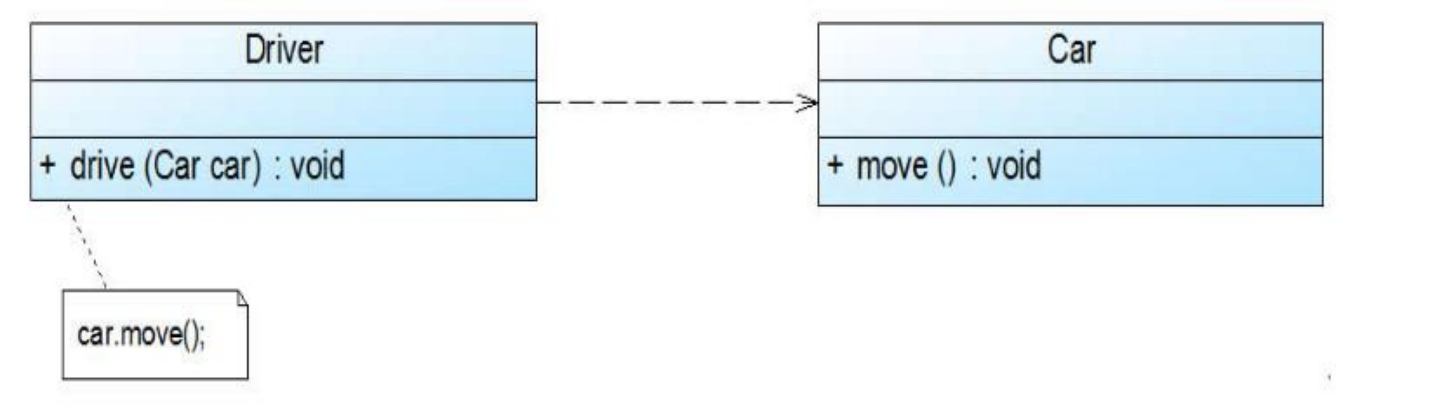
- 泛化（Generalization）

- 关联（Association）
  组合(Composition)
  聚合（Aggregation）

- 实现（Realization）

类图中的关系

- 依赖(Dependency)

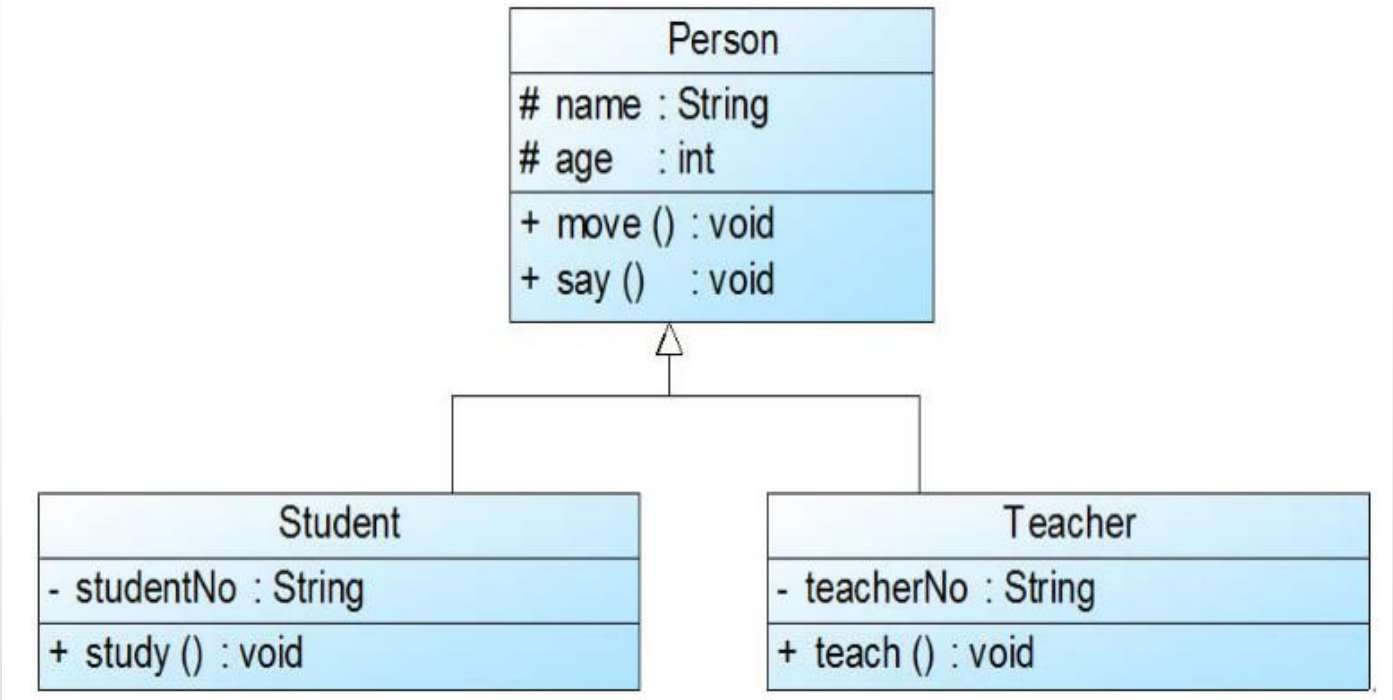


类图中的关系

- 依赖(Dependency)
- 是一种使用的关系，即一个类的实现需要另一个类的协助，所以要尽量不使用双向的互相依赖。可以简单的理解，就是一个类A使用到了另一个类B，而这种使用关系是具有偶然性的、临时性的、非常弱的，但是B类的变化会影响到A；比如某人要过河，需要借用一条船，此时人与船之间的关系就是依赖。在UML中，依赖关系用带箭头的虚线表示，由依赖的一方指向被依赖的一方。

类图中的关系

- 泛化(Generalization)关系

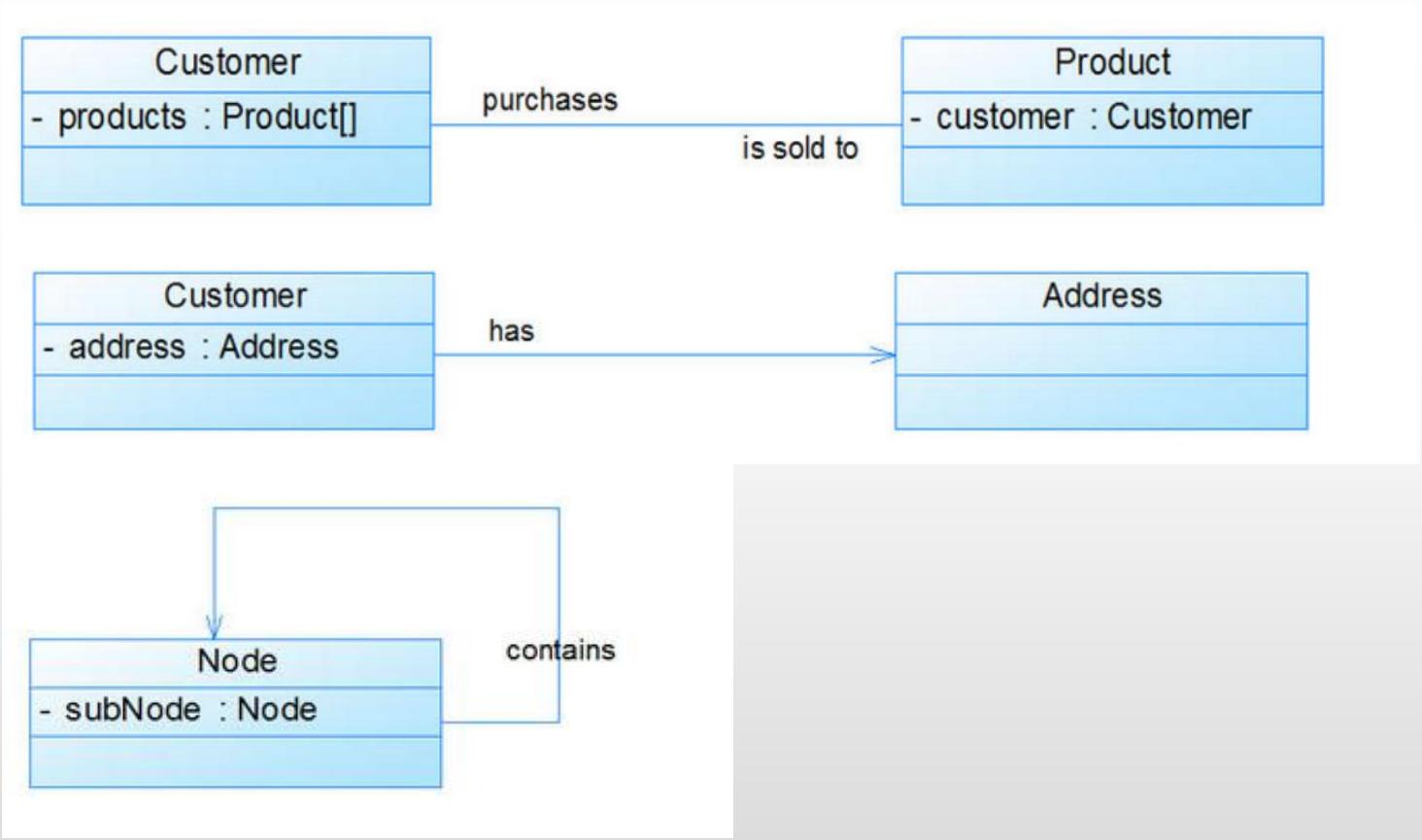


类图中的关系

- 也就是继承关系的反关系，用于描述父类与子类之间的关系，父类又称作基类或超类，子类又称作派生类。在UML中，泛化关系用带空心三角形的直线来表示。
- 子类继承自父类，父类是子类的泛化。

类图中的关系

- **关联（Association）**



类图中的关系

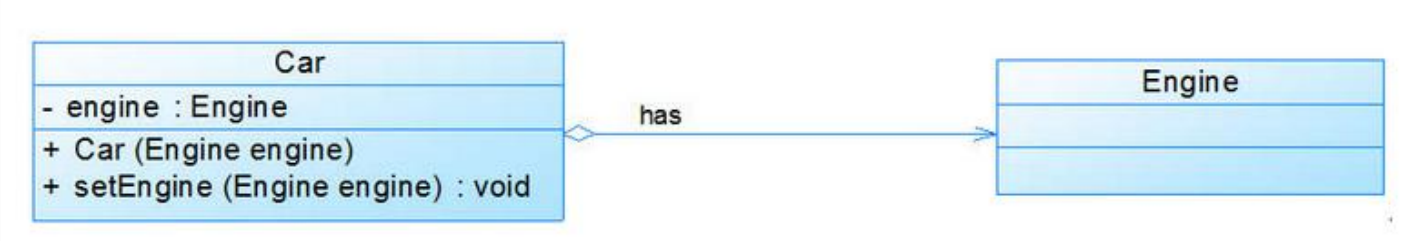
表示方式	多重性说明
1..1	表示另一个类的一个对象只与该类的一个对象有关系
0..*	表示另一个类的一个对象与该类的零个或多个对象有关系
1..*	表示另一个类的一个对象与该类的一个或多个对象有关系
0..1	表示另一个类的一个对象没有或只与该类的一个对象有关系
m..n	表示另一个类的一个对象与该类最少m，最多n个对象有关系 (m≤n)

类图中的关系

- 是一种拥有的关系，它使一个类知道另一个类的属性和方法；如：老师与学生，丈夫与妻子。
- 关联是类之间的结构关系，它描述了一组链，链是对象之间的连接。两个类之间可以有多个由不同角色标识的关联。关联可以是双向的，也可以是单向的。双向的关联可以有两个箭头或者没有箭头，单向的关联有一个箭头。
- (1) 双向关联。默认情况下，关联是双向的。
- (2) 单向关联
- (3)自关联
- (4)多重关联

类图中的关系

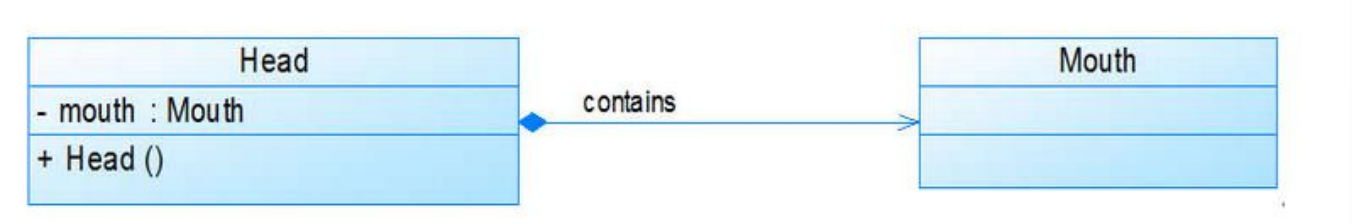
- 聚合（Aggregation）



- 聚合是关联关系的一种特例，他体现的是整体与部分、拥有的关系，即has-a的关系，此时整体与部分之间是可分离的，他们可以具有各自的生命周期，部分可以属于多个整体对象，也可以为多个整体对象共享在UML中，聚合关系用带空心菱形的直线表示。例如：汽车发动机(Engine)是汽车(Car)的组成部分，但是汽车发动机可以独立存在，因此，汽车和发动机是聚合关系，如图所示：

类图中的关系

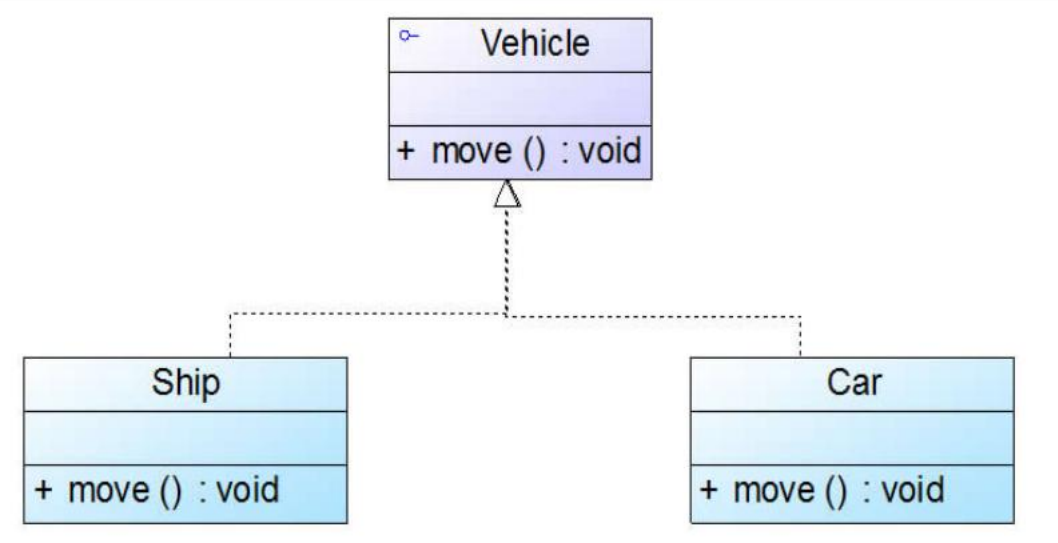
- 组合(Composition)



- 组合也是关联关系的一种特例，他体现的是一种contains-a的关系，这种关系比聚合更强，也称为强聚合；他同样体现整体与部分间的关系，但此时整体与部分是不可分的，整体的生命周期结束也就意味着部分的生命周期结束。在UML中，组合关系用带实心菱形的直线表示。

类图中的关系

- **实现关系（*Implementation*）**

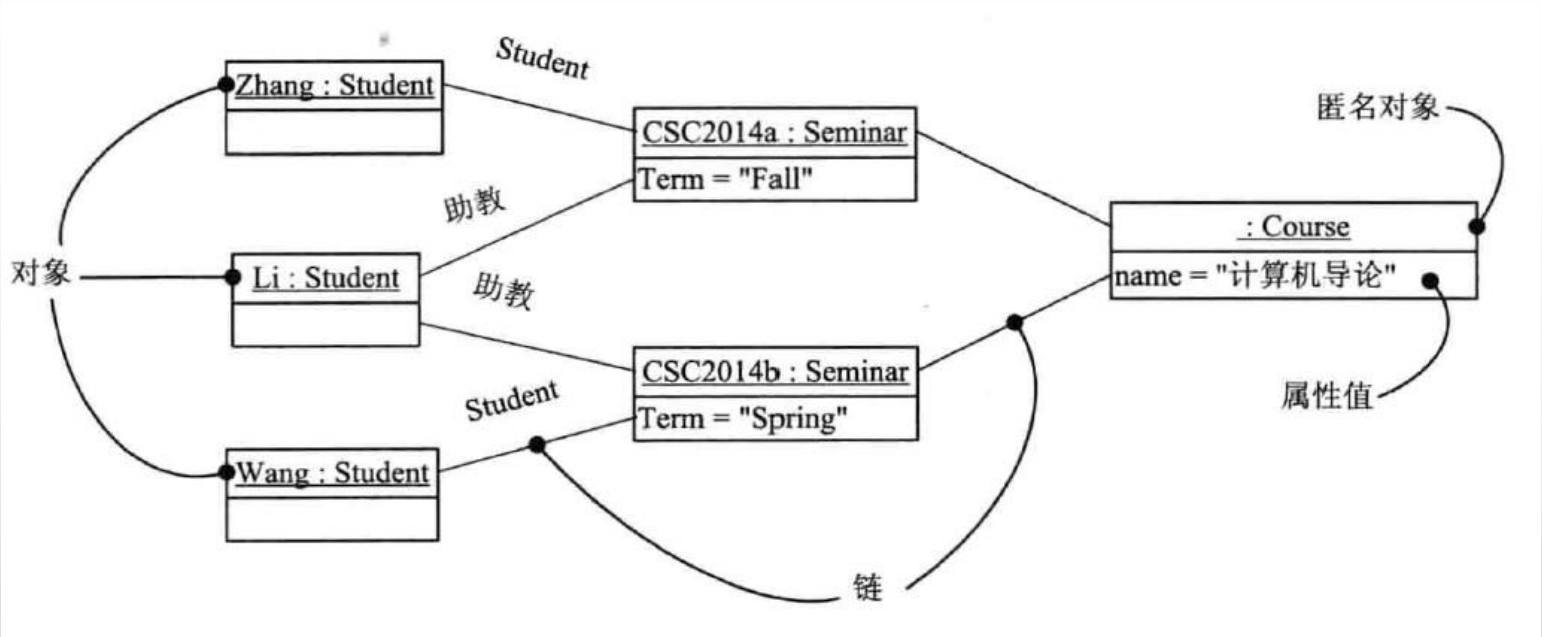


类图中的关系

- 是用来规定接口和实现接口的类或者构建结构的关系，接口是操作的集合，而这些操作就用于规定类或者构建的一种服务。
- 接口之间也可以有与类之间关系类似的继承关系和依赖关系，但是接口和类之间还存在一种实现关系(Realization)，在这种关系中，类实现了接口，类中的操作实现了接口中所 声明的操作。在UML中，类与接口之间的实现关系用带空心三角形的虚线来表示。例如：定义了一个交通工具接口Vehicle，包含一个抽象操作move()，在类Ship和类Car中都实现了该move()操作，不过具体的实现细节将会不一样，如图所示：

对象图

- 对象图(ObjectDiagram) 展现了某一时刻一组对象以及它们之间的关系，描述了在类图中所建立的事物的实例的静态快照。



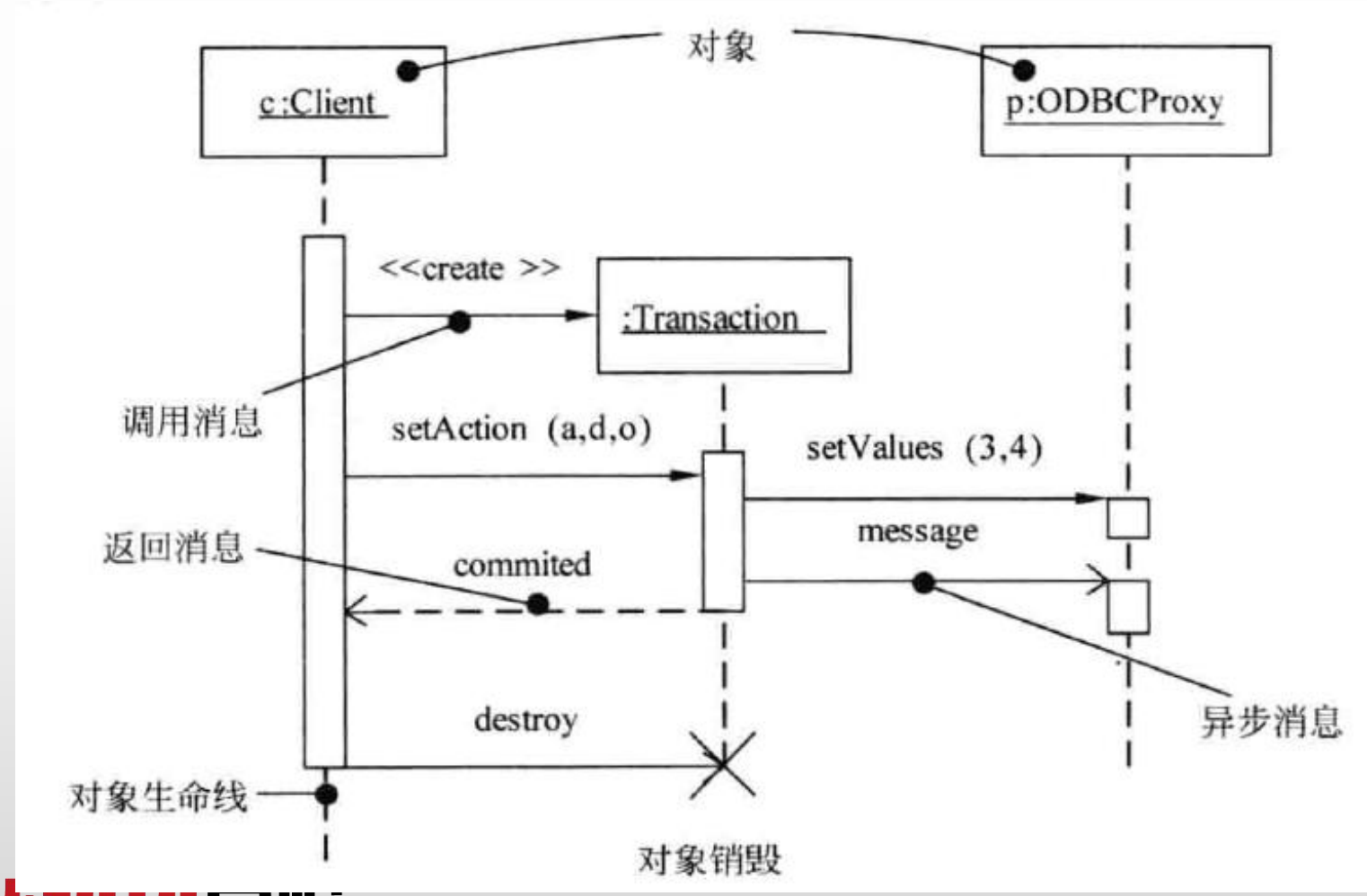
类图与对象图

类图	对象图
在类中包含三部分：类名、类的属性和类的操作	对象包含两个部分：对象的名称和对象的属性
类的名称栏只包含类名	对象的名称栏包含”对象名：类名“
类的属性栏定义了所有的属性特征	对象的属性栏定义了属性的当前值
类中列出了操作	对象图中不包含操作内容，因为对属于同一个类的对象，其操作是相同的
类中使用了关联连接，关联中使用名称、角色以及约束等特征定义	对象使用链进行连接，链中包含名称、角色
类代表的是对象的分类所以必须说明可以参与关联的对象的数目	对象代表的是单独的实体，所有的链都是一对一的，因此不涉及到多重性

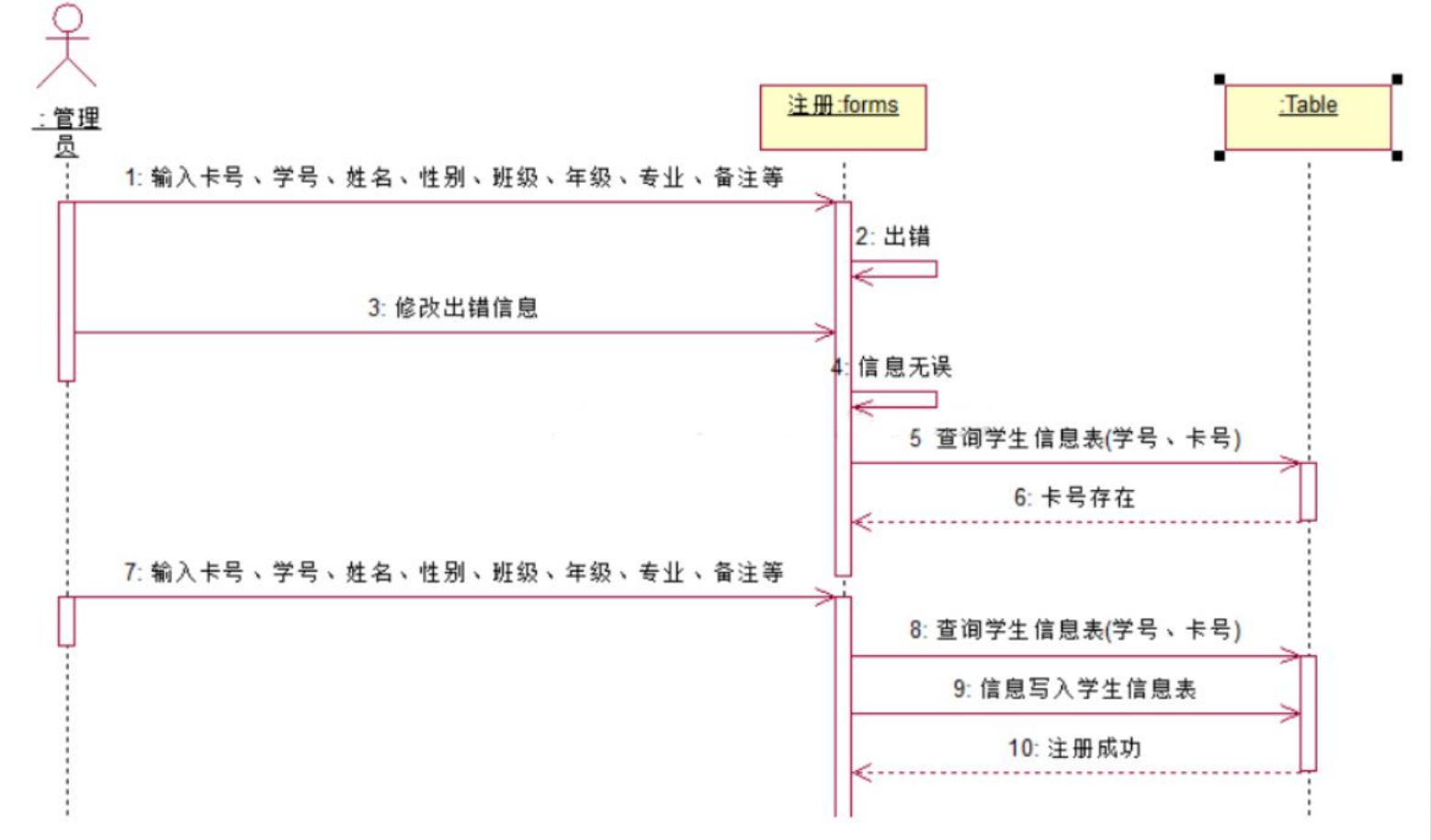
交互图

- 交互图表现为序列图、通信图、交互概览图和计时图。用于动态建模。
- 序列图强调消息时间顺序的交互
- 通信图强调接收和发送信息的对象的结构组织的交互
- 交互概览图强调控制流的交互图
- 计时图适合嵌入式系统建模的交互图

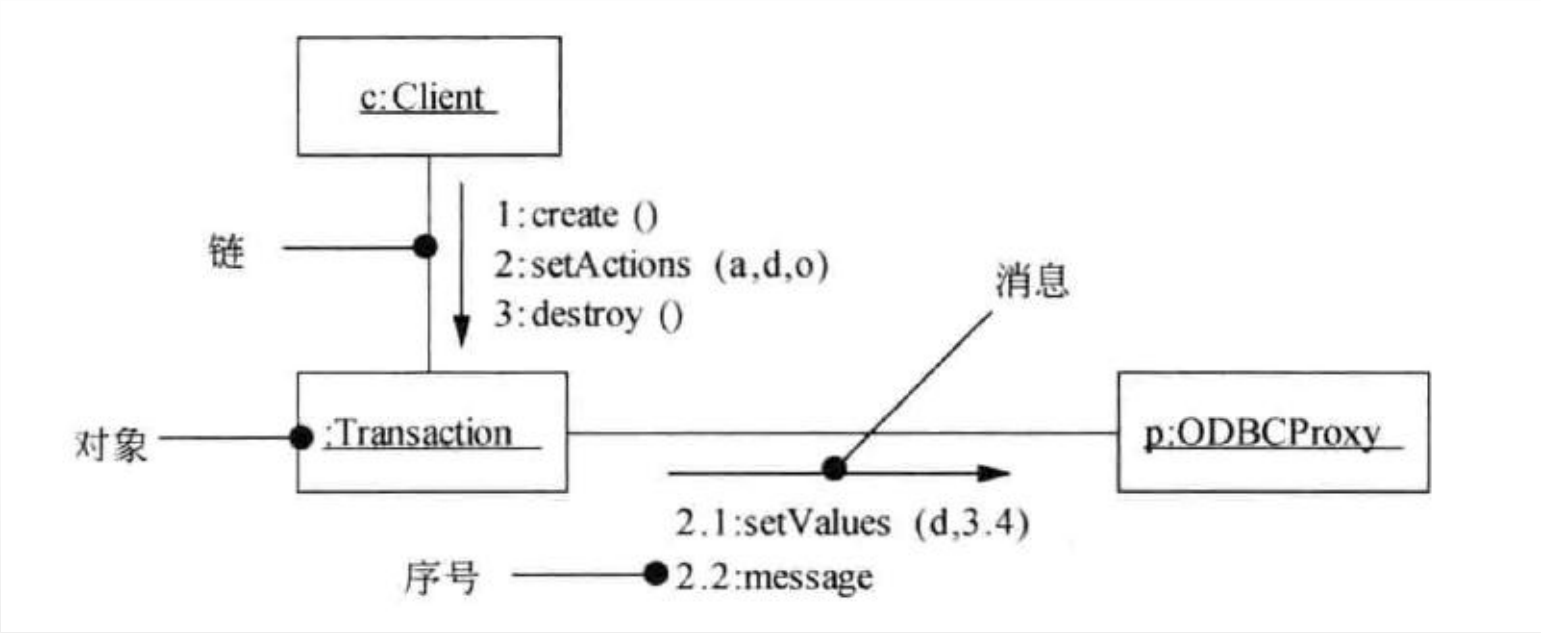
交互图——序列图



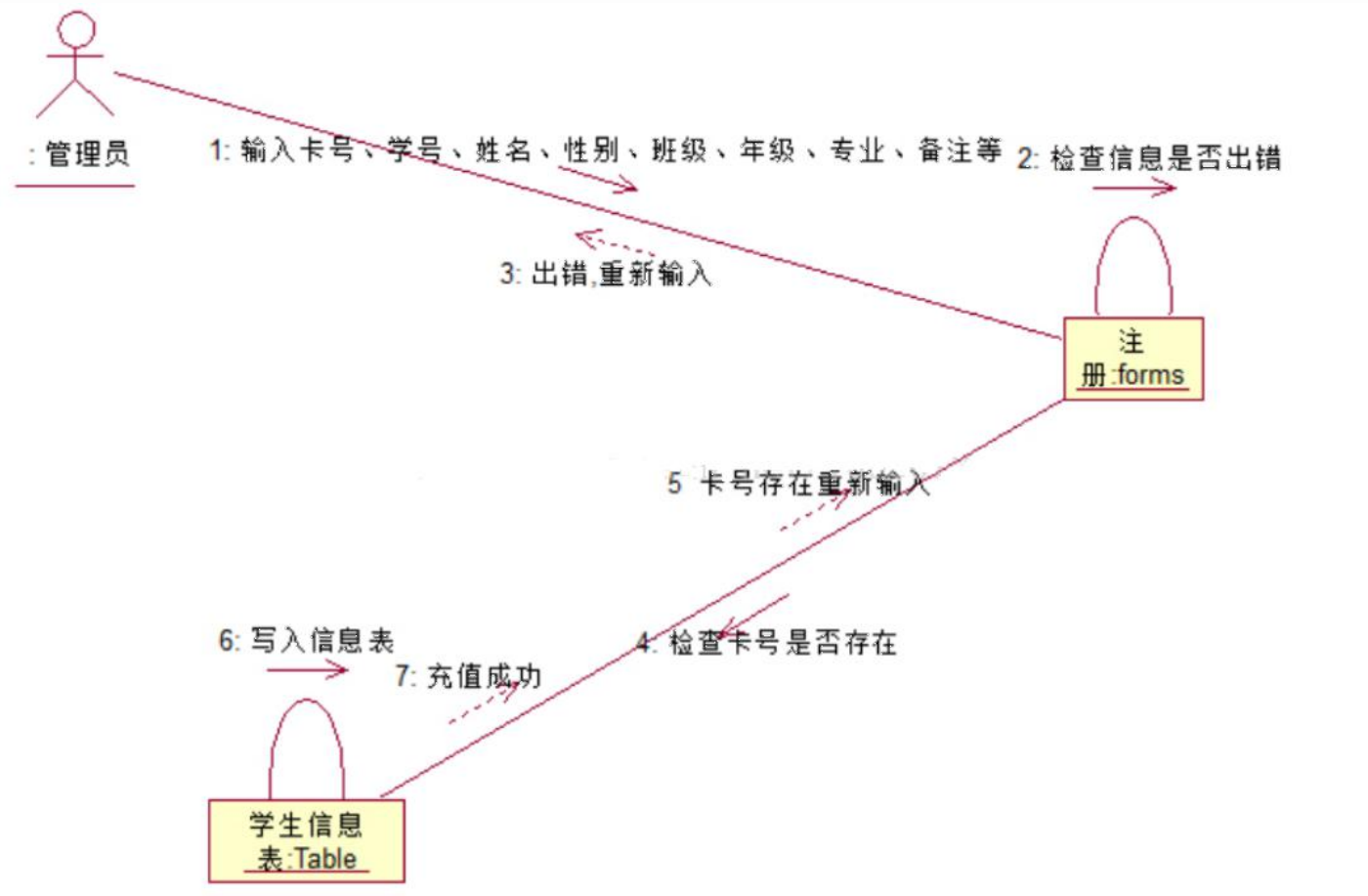
交互图——序列图



交互图——通信图（协作图）



交互图——通信图（协作图）

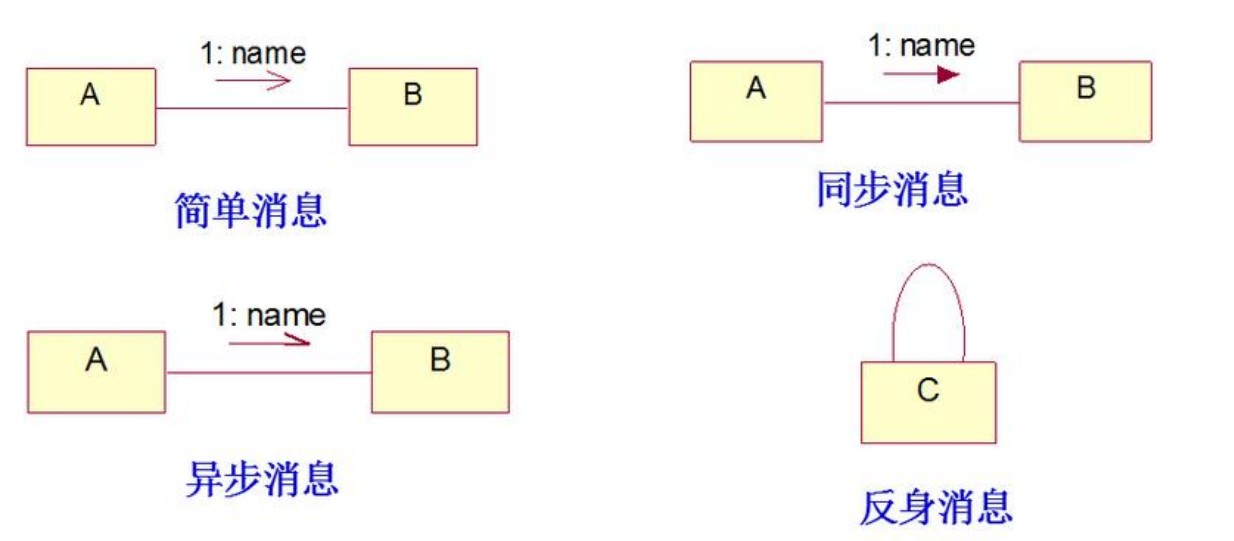


交互图——通信图（协作图）

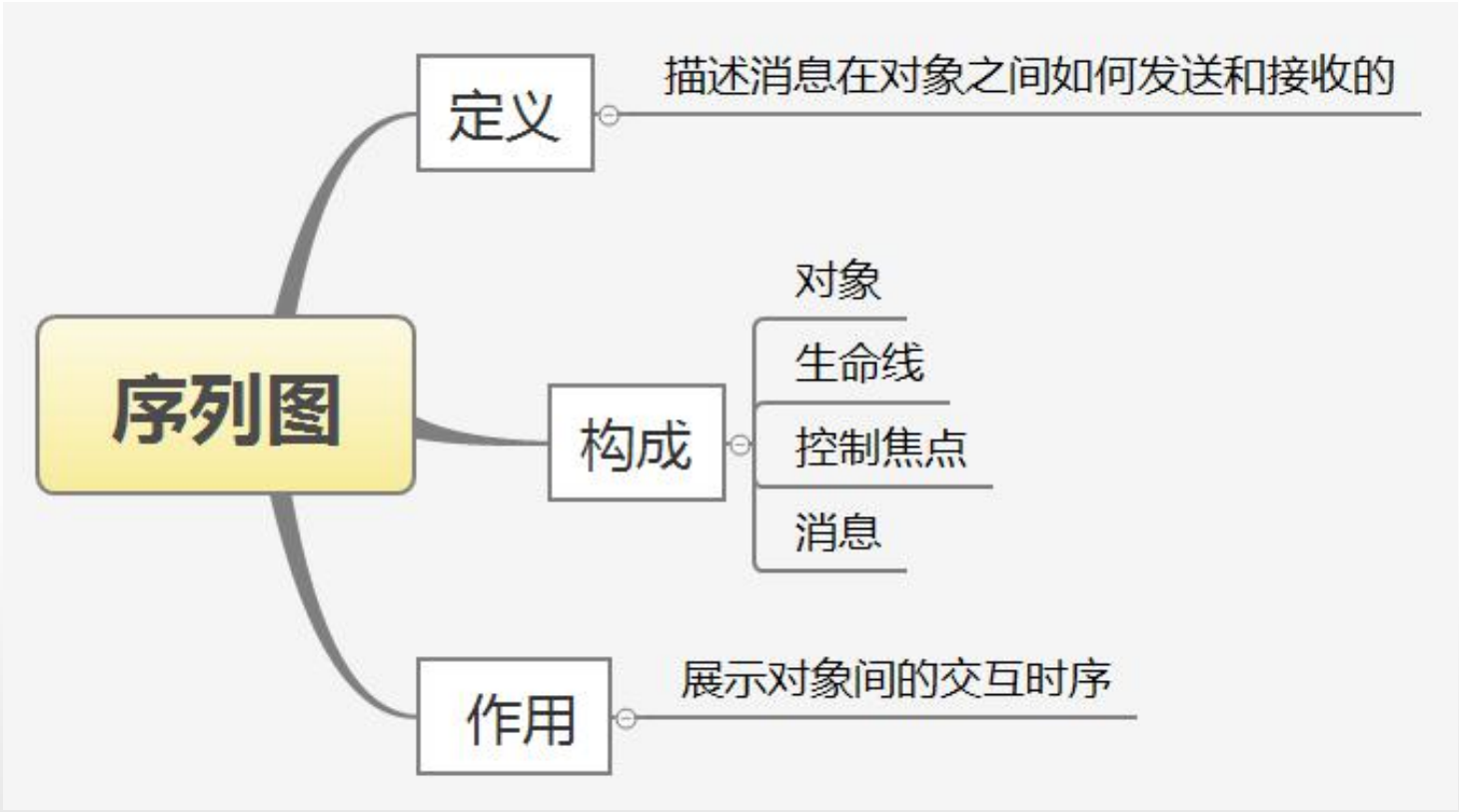
对象：图中的矩形元素即为对象，其中冒号前面部分为对象名，后面为类名，表示类的一个实例。

链接：用两个对象之间的单一线条表示，用来在通信图中关联对象，目的是让消息在不同系统对象之间传递。可以理解链接是公路，消息是车。

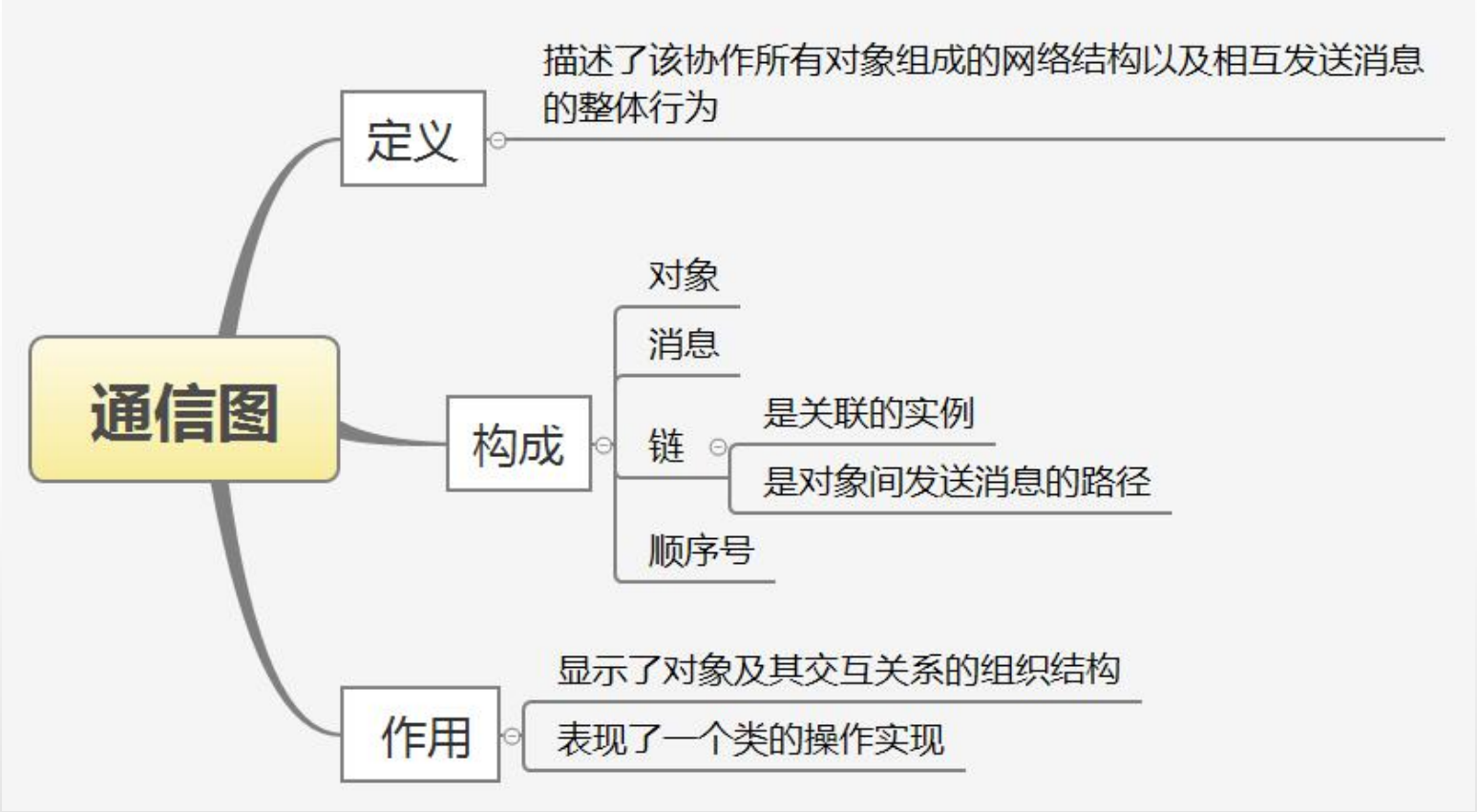
消息：通信图中对象之间通信的方式。



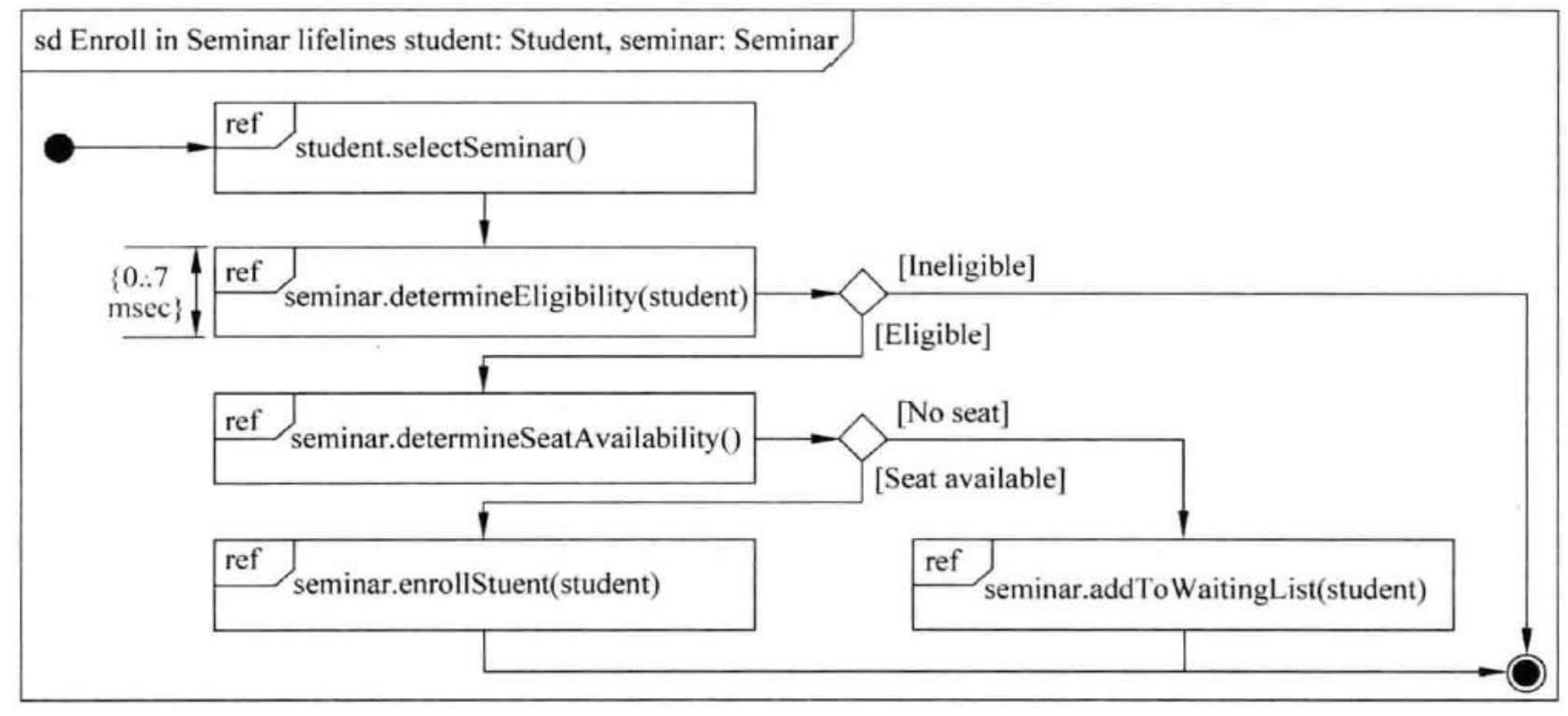
序列图与协作图的区别



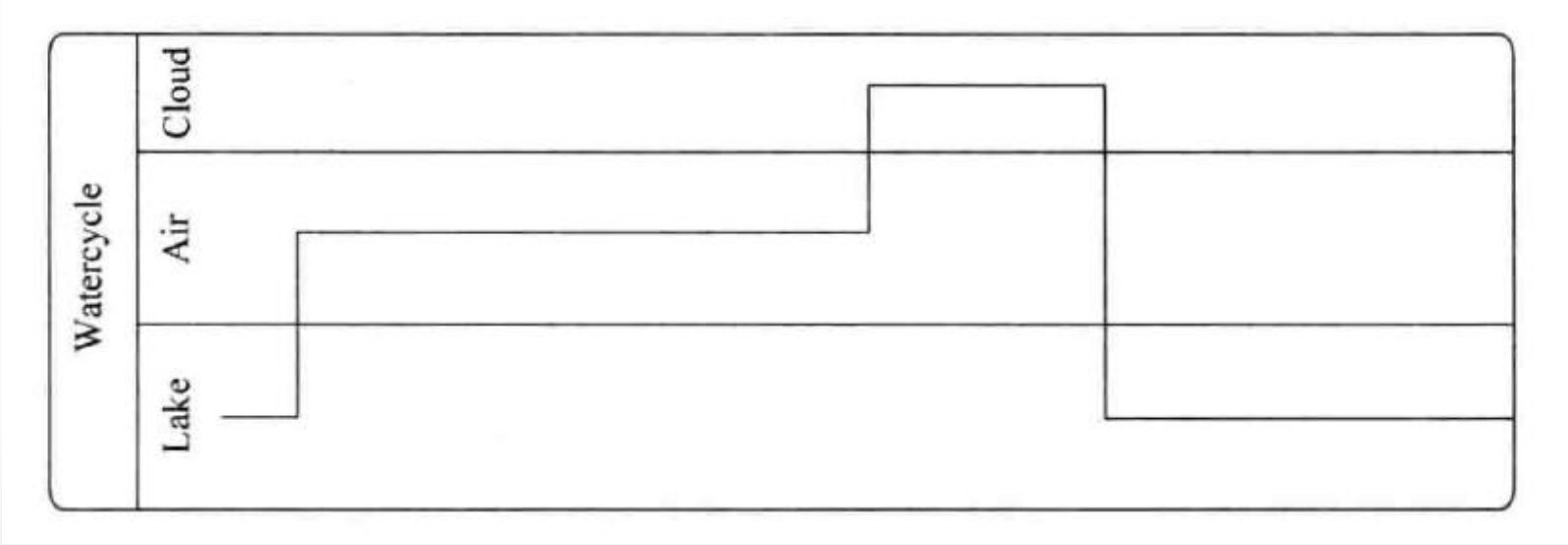
序列图与协作图的区别



交互图——交互概览图

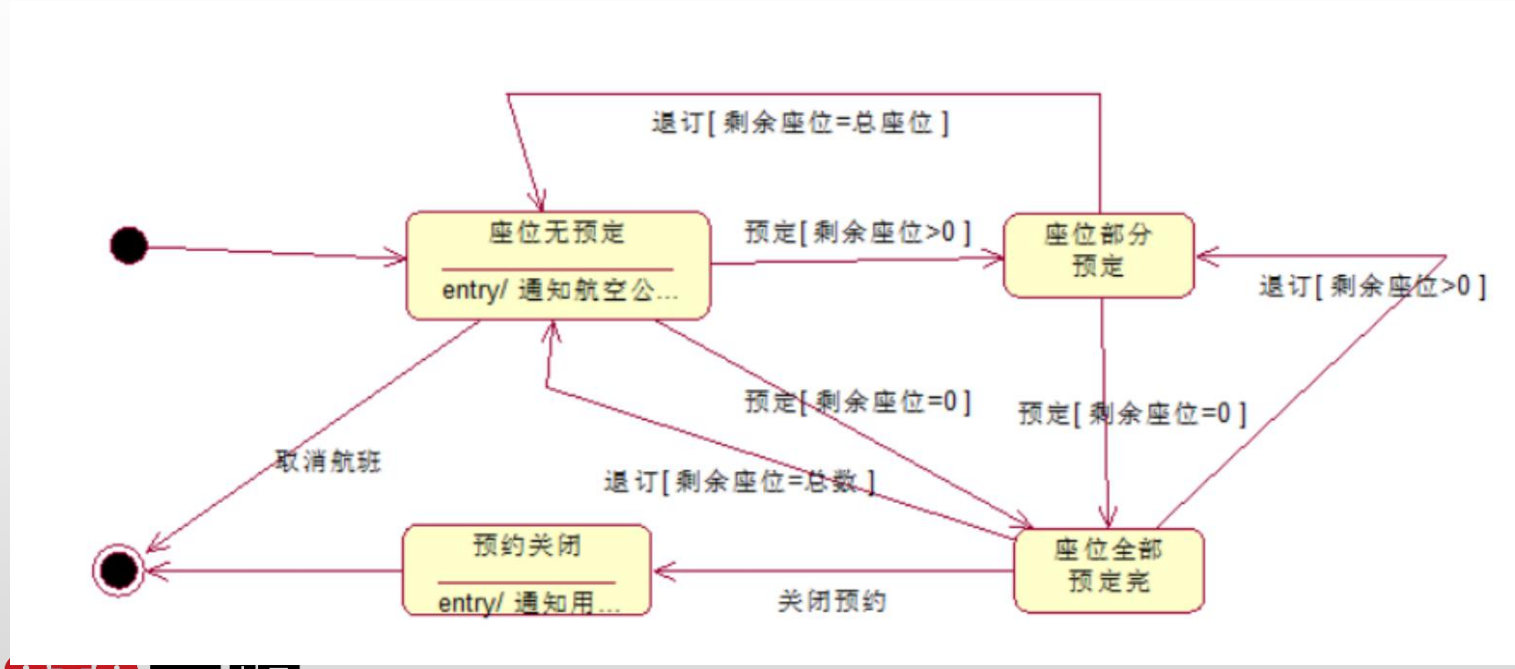


交互图——计时图



状态图

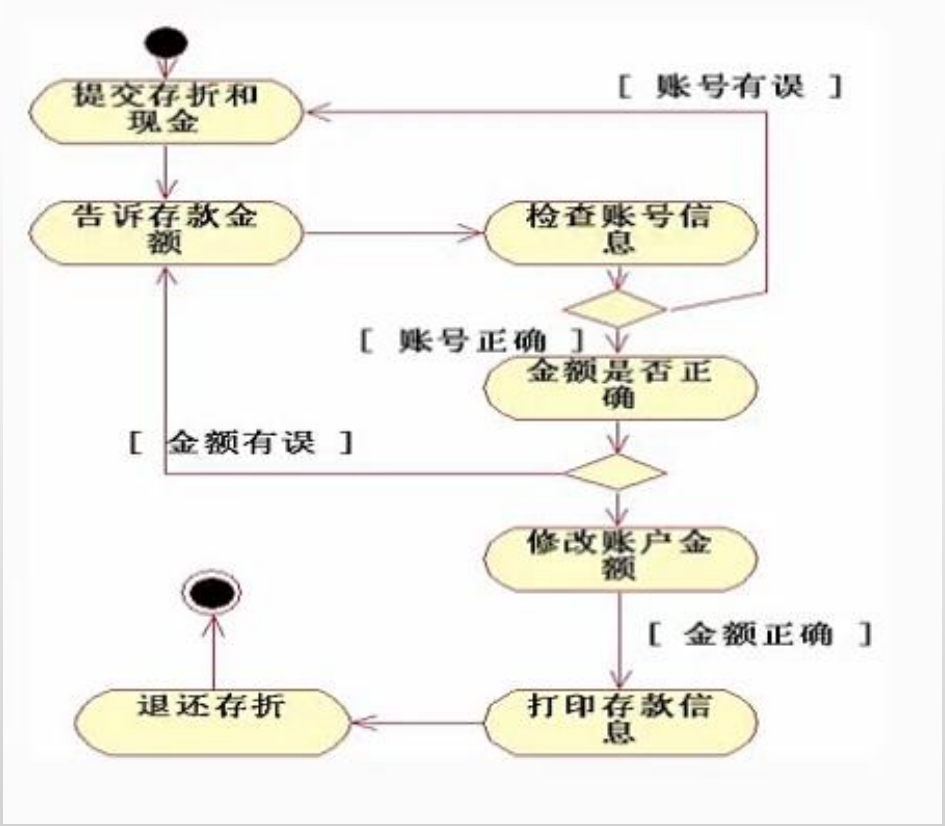
- 用来描述一个特定的对象所有可能的状态,以及由于各种事件的发生而引起的状态之间的转移和变化。用于对系统的**动态方面建模**。



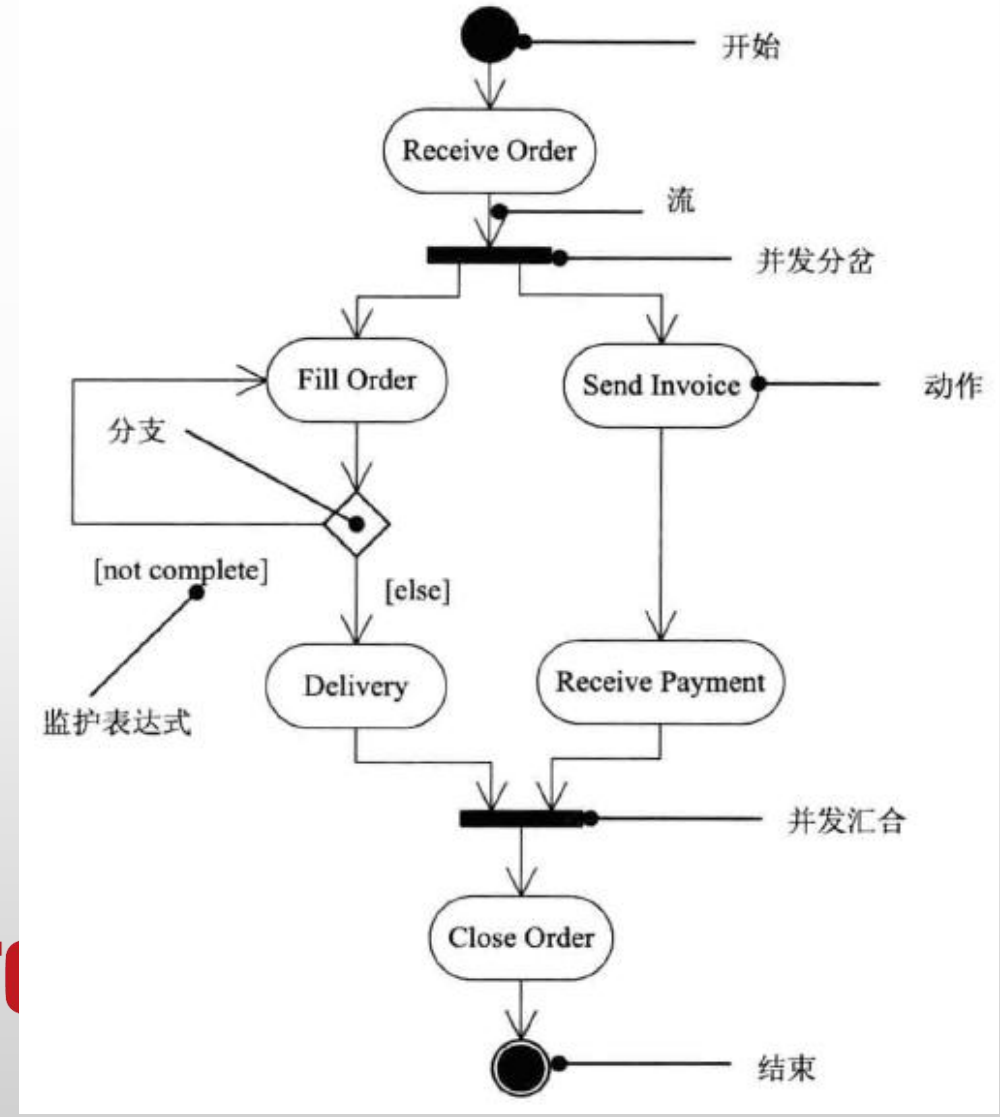
活动图

- 将进程或其他计算的结构展示为计算内部一步步的控制流和数据流，主要用来描述系统的动态视图。活动图在本质上是一种流程图。活动图着重表现从一个活动到另一个活动的控制流，是内部处理驱动的流程。

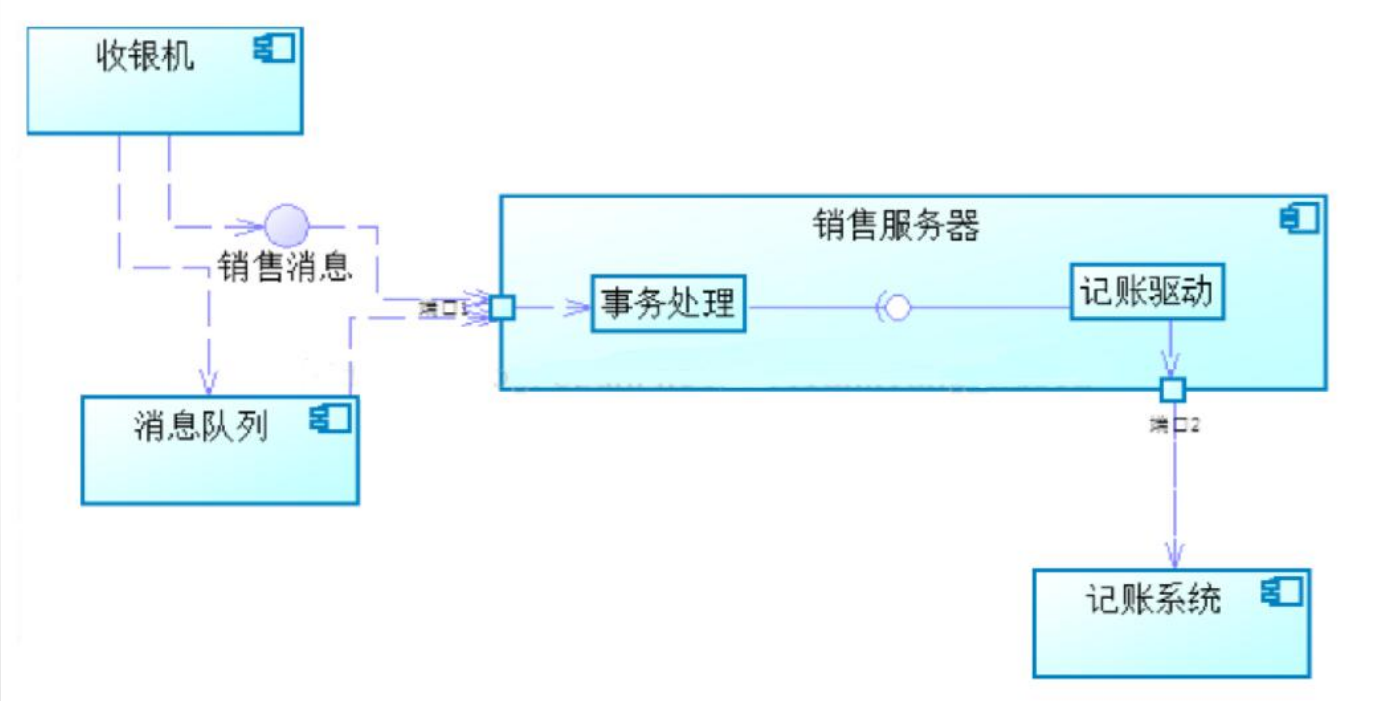
- 活动图主要描述
- 行为的动作，
- 状态图主要描述
- 行为的结果。



活动图


开始
Receive Order
流
并发分岔
Fill Order
Send Invoice
动作
分支
[not complete]
[else]
监护表达式
Delivery
Receive Payment
并发汇合
Close Order
结束

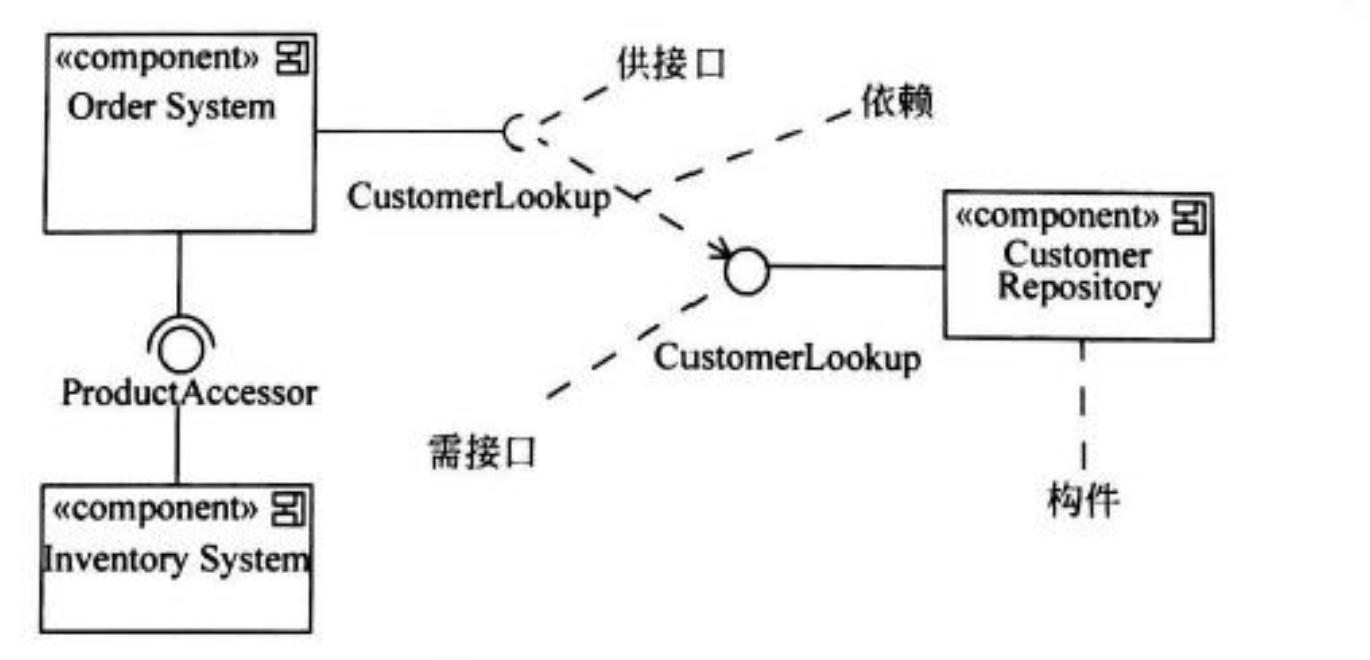
构件图（组件图）



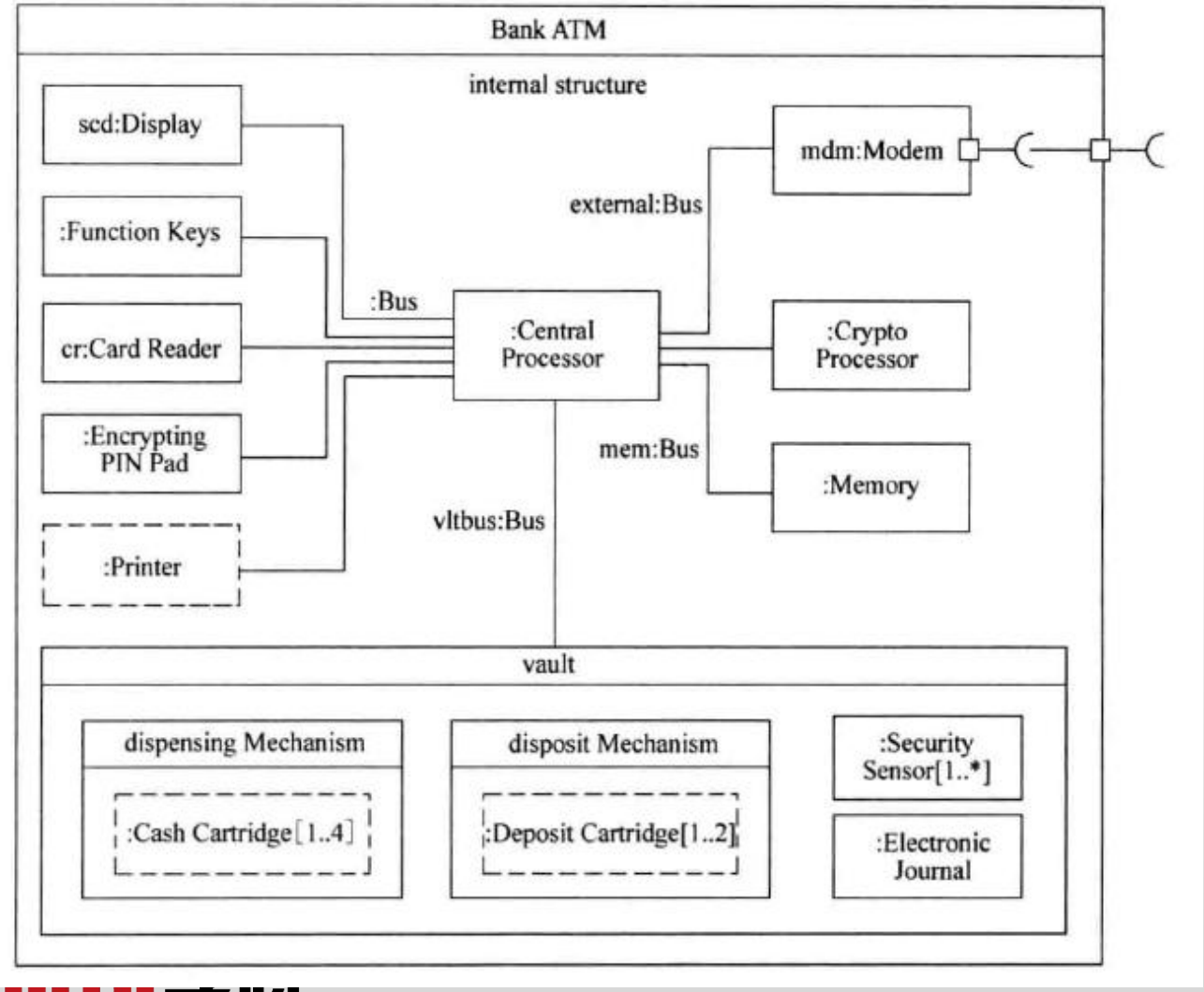
构件图（组件图）

- 使用构件图的思想是复用。就像是我们盖房子，当房子的大体框架建好之后，剩下的门和窗户家具之类的直接拿来安装上即可，不需要再从新制作，直接拿来复用的思想。这些门和窗户就相当于一个个的构件。
- 构件有一下几种类型：
- （1）部署构件：dll文件、exe文件、com+对象、CORBA对象、EJB、动态Web页和数据库表等。
- （2）工作产品构件：源代码文件、数据文件等
- （3）执行构件：系统执行后得到的构件。

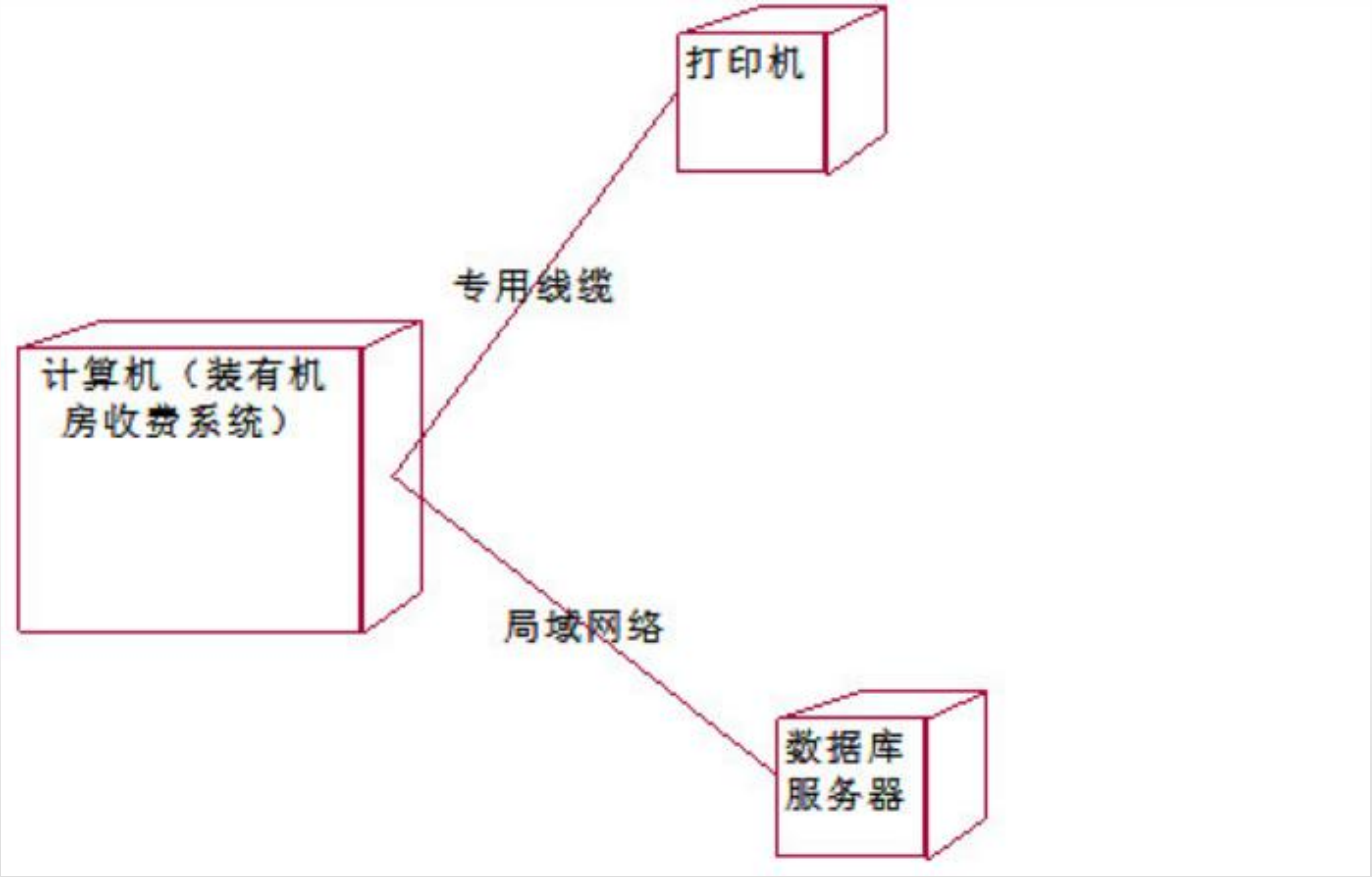
构件图（组件图）



组合结构图


Bank ATM
internal structure
scd:Display
:Function Keys
cr:Card Reader
:Encrypting PIN Pad
:Printer
:Bus
:Central Processor
mdm:Modem
external:Bus
:Crypto Processor
mem:Bus
:Memory
vltbus:Bus
vault
dispensing Mechanism
:Cash Cartridge[1..4]
disposit Mechanism
:Deposit Cartridge[1..2]
:Security Sensor[1..*]
:Electronic Journal

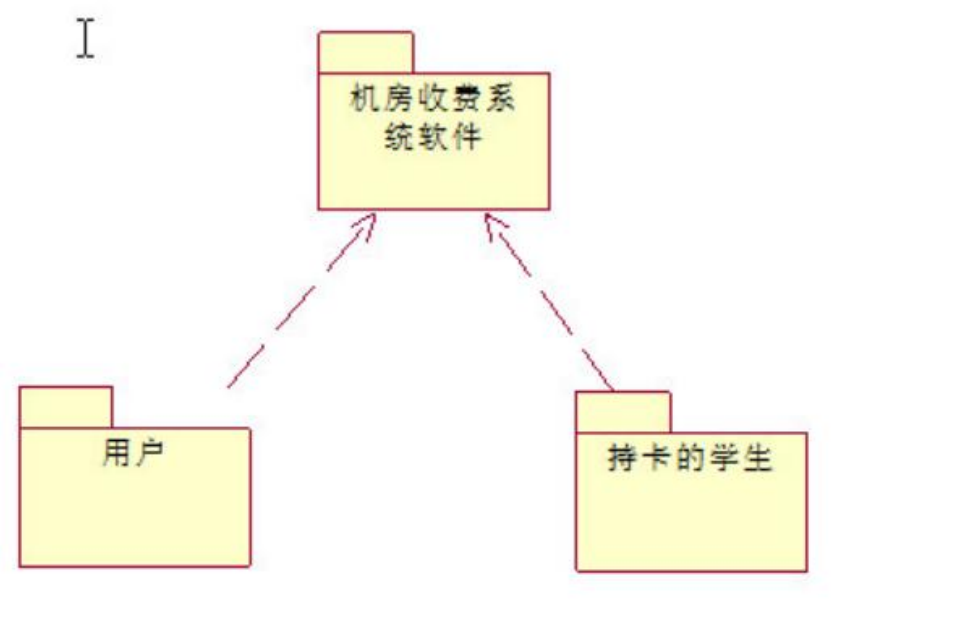
部署图



部署图

- 是用来显示系统中软件和硬件的物理架构。
- 从部署图中，可以了解到软件和硬件组件之间的物理关系以及处理节点的组件分布情况。使用部署图可以显示运行时系统的结构，同时还传达构成应用程序的硬件和软件元素的配置和部署方式。

包图



包图

- 在 UML 中用类似于文件夹的符号表示的模型元素的组合。包图是一种维护和描述系统总体结构的模型的重要建模工具，通过对包中各个包以及包之间关系的描述，展现出系统的模块与模块之间的依赖关系。
- 包图的作用：包图可以描述需求，设计的高阶概况；包图通过合理规划自身功能反应系统的高层架构，在逻辑上将系统进行模块化分解；包图最终是组织源码的方式。
- 一个包图可以由任何一种UML图组成，通常是UML用例图或是UML类图。
- 包被描述成文件夹，可以用于UML任何一种的图上。
- 包图只是把某些类放在一个包中，因此可以看做是类图的一种。

uml分类

- （1）静态模型（系统结构）
- 用例图、类图、对象图、构件图、部署图

- （2）动态模型（系统行为）
- 状态图、活动图、顺序图、协作图

典型真题

- （ ）多态是指操作（方法）具有相同的名称、且在不同的上下文中所代表的含义不同。
- A.参数
- B.包含
- C.过载
- D.强制

典型真题

- 试题分析
- 参数多态：应用广泛、最纯的多态。
- 包含多态：同样的操作可用于一个类型及其子类型。包含多态一般需要进行运行时的类型检查。
- 强制多态：编译程序通过语义操作，把操作对象的类型强行加以变换，以符合函数或操作符的要求。
- 过载多态：同一个名（操作符、函数名）在不同的上下文中有不同的类型。
- 本题应该选择C选项过载多态。
- 试题答案：C

典型真题

- 在某销售系统中，客户采用扫描二维码进行支付。若采用面向对象方法开发该销售系统，则客户类属于（ ）类， 二维码类属于（ ）类。

- A.接口　　B.实体　　C.控制　　D.状态

- A.接口　　B.实体　　C.控制　　D.状态

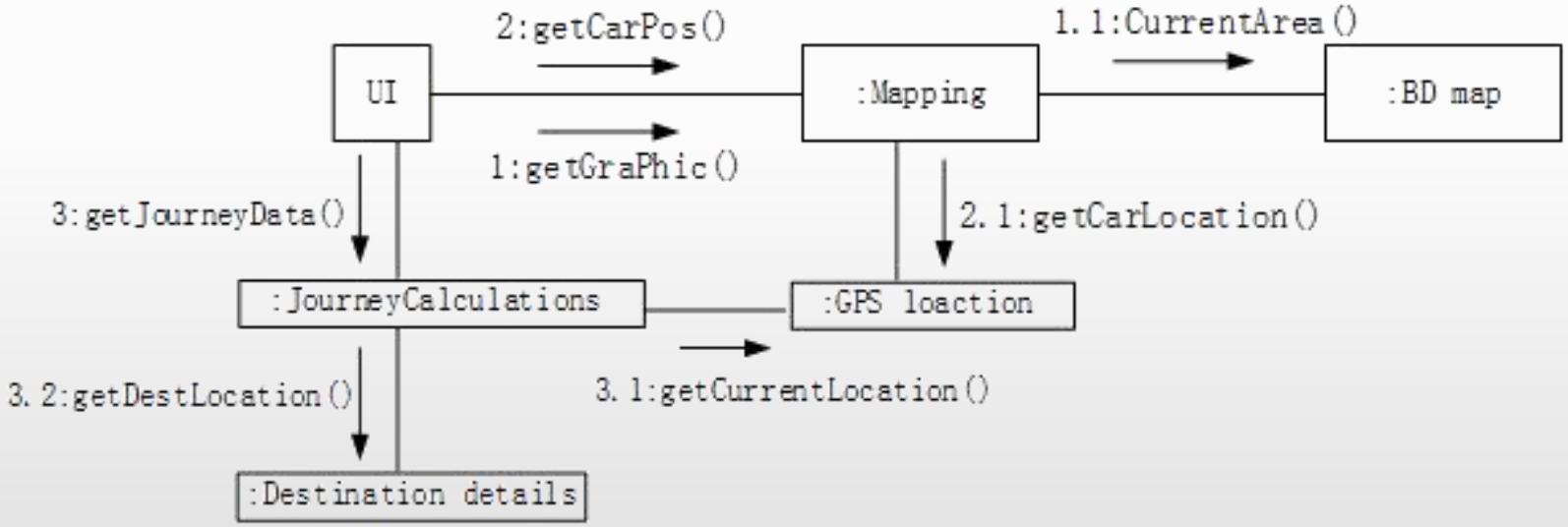
典型真题

- 试题分析
- 类可以分为三种：实体类、接口类（边界类）和控制类。实体类的对象表示现实世界中真实的实体，如人、物等。接口类（边界类）的对象为用户提供一种与系统合作交互的方式，分为人和系统两大类，其中人的接口可以是显示屏、窗口、Web窗体、对话框、菜单、列表框、其他显示控制、条形码、二维码或者用户与系统交互的其他方法。系统接口涉及到把数据发送到其他系统，或者从其他系统接收数据。控制类的对象用来控制活动流，充当协调者。

- 试题答案：B、A

典型真题

- 如下所示的图为UML的（ ），用于展示某汽车导航系统中（ ）。
  Mapping对象获取汽车当前位置（GPS Location）的消息为
  （ ）。



典型真题

- A.类图　　B.组件图　　C.通信图　　D.部署图

- A.对象之间的消息流及其顺序
- B.完成任务所进行的活动流
- C.对象的状态转换及其时间顺序
- D.对象之间消息的时间顺序

- A.1: getGraphic()　　B. 2: getCarPos()
- C. 1.1: CurrentArea()　　D.2. 1: getCarLocation()

典型真题

- 试题分析
- 通信图（communication diagram）。通信图是一种交互图，它强调收发消息的对象或参与者的结构组织。顺序图和通信图表达了类似的基本概念，但它们所强调的概念不同，顺序图强调的是时序，通信图强调的是对象之间的组织结构（关系）；获取汽车当前位置的消息为2.1：getCarLocation()。
- 试题答案：C、A、D

设计模式

- 设计模式（Design Pattern）是一套被反复使用、多数人知晓的、经过分类的、代码设计经验的总结。
- 使用设计模式的目的：为了代码可重用性、让代码更容易被他人理解、保证代码可靠性。 设计模式使代码编写真正工程化；设计模式是软件工程的基石脉络，如同大厦的结构一样。

设计模式

创建型模式	用于创建对象	工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式。共五种。口诀：单抽元件（建）厂
结构型模式	处理类或对象的组合	适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式。共七种。口诀：外侨（桥）组员（元）戴（代）配饰。
行为型模式	描述类与对象怎样交互、怎样分配职责	策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。共十一种。口诀：观摩（模）对（迭）策，责令解放（访），戒（介）忘台（态）。

设计模式

	创 建 型	结 构 型	行 为 型
类	Factory Method	Adapter（类）	Interpreter Template Method
对象	Abstract Factory Builder Prototype Singleton	Adapter（对象） Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

设计模式

目的	设计模式	简要说明	可改版的方面
创建型	Abstract Factory 抽象工厂	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。	产品对象族
	Builder 建造者	将一个复杂的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。	如何建立一个组合对象
	Factory Method （类） 工厂方法	定义一个用于创建对象的接口，让子类决定将哪一个类实例化。使一个类的实例化延迟到其子类。	实例化子类对象
	Prototype 原型	用原型实例指定创建对象的种类，并且通过拷贝这个原型来创建新的对象。	实例化类的对象
	Singleton 单例	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。	类的单个实例

目的	设计模式	简要说明	可改版的方面
结构型	Adapter（类）适配器	将一个类的接口转换成客户希望的另外一个接口。Adapter模式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。	与对象的接口
	Bridge 桥接	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。	对象的实现
	Composite 组合模式	将对象组合成树形结构以表示“部分-整体”的层次结构。Composite使得客户对单个对象和复合对象的使用具有一致性。	对象的结构和组合
	Decorator 装饰模式	动态地给一个对象添加一些额外的职责。就扩展功能而言，Decorator模式比生成子类方式更为灵活。	无子类对象责任
	Facade 外观模式	为子系统中的一组接口提供一个一致的界面，外观模式通过提供一个高层接口，隔离了外部系统与子系统间复杂的交互过程，使得复杂系统的子系统更易使用。。	与子系统的接口

目的	设计模式	简要说明	可改版的方面
结构型	*Flyweight* 享元模式	运用共享技术有效地支持大量细粒度的对象。	对象的存储代价
	Proxy 代理模式	为其他对象提供一种代理以控制对这个对象的访问。 代理模式使用代理对象完成用户请求，屏蔽用户对真实对象的访问。。	如何访问对象 对象位置

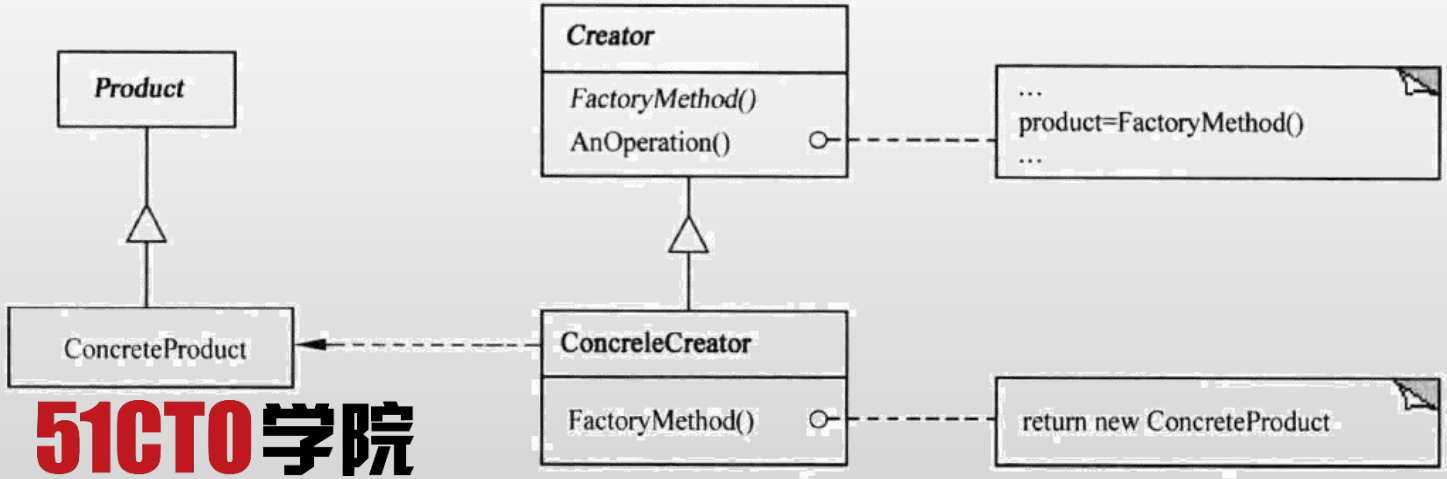
目的	设计模式	简要说明	可改变的方面
行为型	*Chain of Responsibility* 责任链模式	避免请求发送者与接收者耦合在一起，让多个对象都有可能接收请求，将这些对象连接成一条链，并且沿着这条链传递请求，直到有对象处理它为止。	可满足请求的对象
	Iterator 迭代器模式	提供一种方法顺序访问一个聚合对象中各个元素，而又无须暴露该对象的内部表示。	如何访问、遍历聚集的元素
	Mediator 中介者模式	用一个中介对象来封装一系列的对象交互，中介者使各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。	对象之间如何交互以及哪些对象交互
	Memento 备忘录模式	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。这样就可以将该对象恢复到原先保存的状态。	何时及哪些私有信息存储在对象之外
	Observer 观察者模式	观察者模式定义了对象间的一种一对多依赖关系，使得每当一个对象改变状态，则所有依赖于它的对象都会得到通知并被自动更新。	信赖于另一对象的对象数量，信对象如何保持最新数据

目的	设计模式	简要说明	可改版的方面
行为型	State 状态模式	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。	对象的状态
	Strategy 策略模式	策略模式定义了一系列的算法，并将每一个算法封装起来，而且使它们还可以相互替换。策略模式让算法独立于使用它的客户而独立变化。	算法
	Command 命令模式	将一个请求封装成一个对象，从而使得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。	何时及如何满足一个请求
	Interpreter （类） 解释器模式	给定一个语言，定义它的文法的一种表示，并定义一个解释器，这个解释器使用该表示来解释语言中的句子。	语言的语法和解释

目的	设计模式	简要说明	可改版的方面
行为型	Template Method （类） 模板方法	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。模板方法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。	算法的步骤
	Visitor 访问者模式	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与这些元素的新操作。即对于某个对象或者一组对象，不同的访问者，产生的结果不同，执行操作也不同。	无须改变其类而可应用于对象的操作

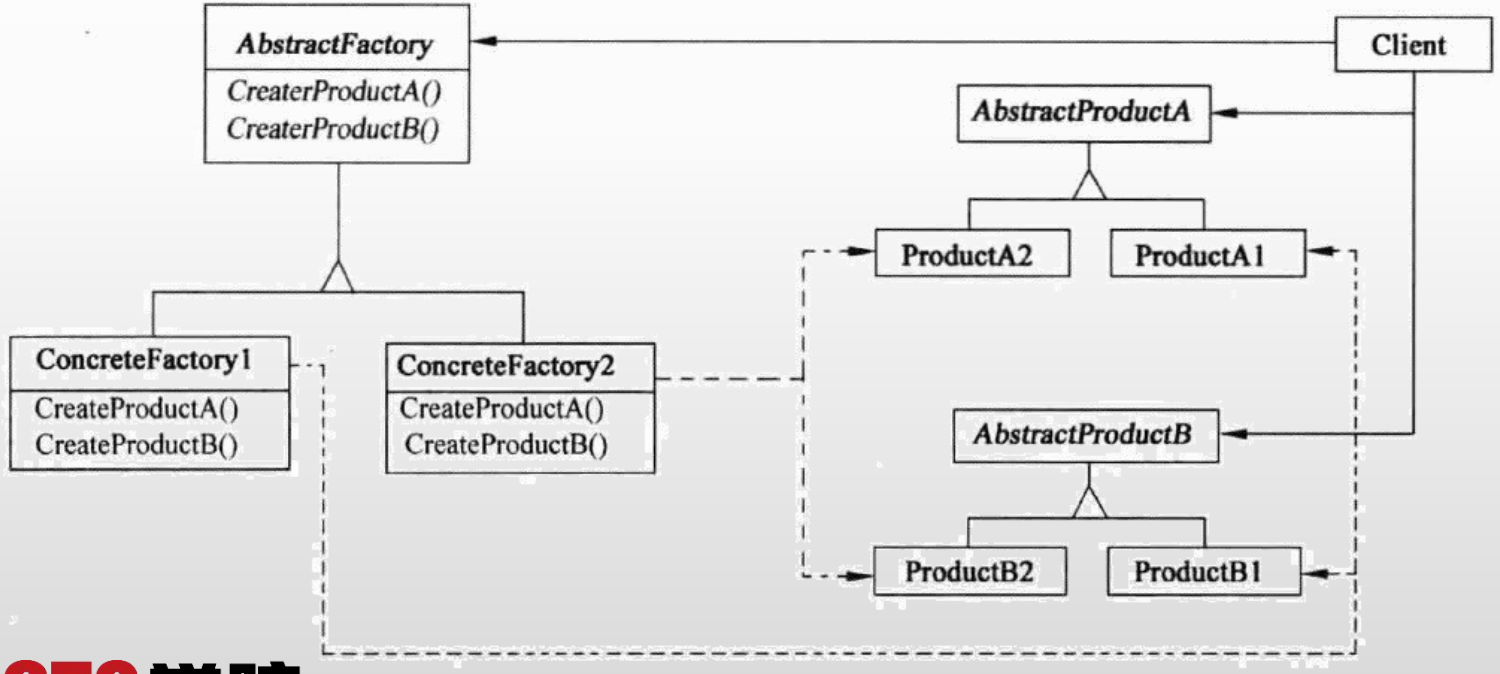
Factory Method工厂方法模式

- 定义一个用于创建对象的接口，让子类决定将哪一个类实例化。Factory Method使一个类的实例化延迟到其子类。
- 在此模式中，工厂父类负责定义创建产品对象的公共接口，而工厂子类负责生产具体的产品对象，使一个类的实例化延迟到其子类，由子类来确定实例化哪个具体的产品类。



Abstract Factory抽象工厂模式：

- **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。**



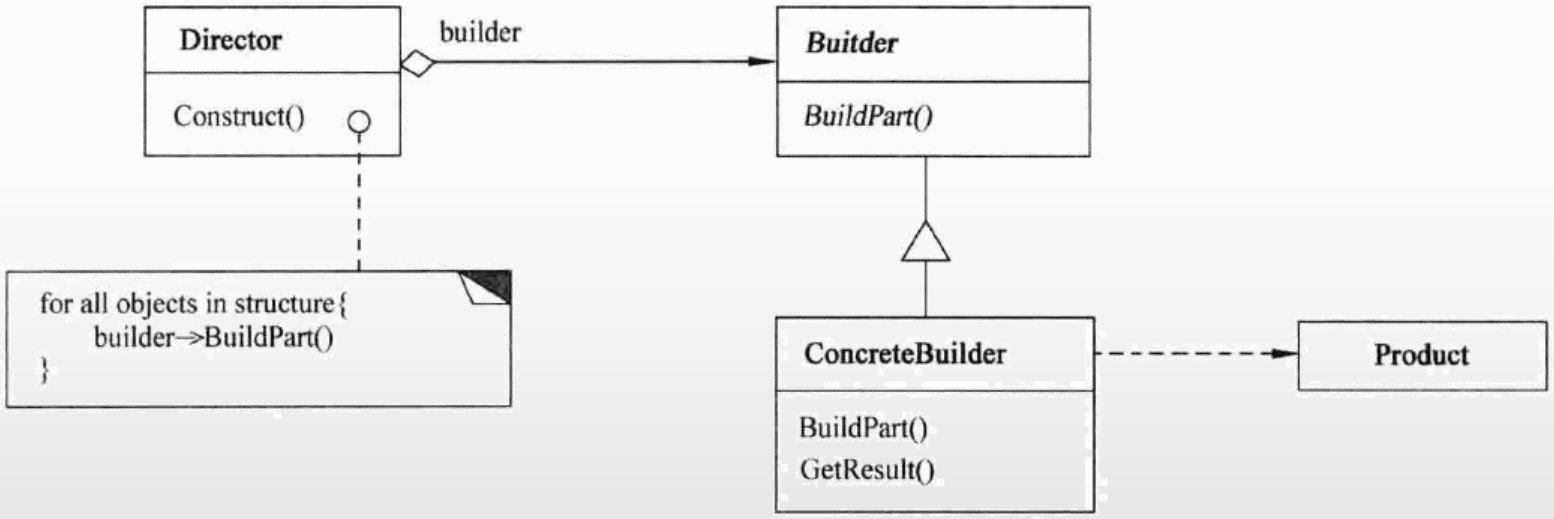
建造者模式（Builder）

- 将一个复杂的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。
- 典型的KFC儿童餐包括一个主食，一个辅食，一杯饮料和一个玩具（例如汉堡、炸鸡、可乐和玩具车）。这些在不同的儿童餐中可以是不同的，但是组合成儿童餐的过程是相同的。
- 使用建造者模式的好处：
- 1.使用建造者模式可以使客户端不必知道产品内部组成的细节。
- 2.具体的建造者类之间是相互独立的，对系统的扩展非常有利。
- 3.由于具体的建造者是独立的，因此可以对建造过程逐步细化，而不对其他的模块产生任何影响。

建造者模式（Builder）

- Builder：抽象建造者。它声明为创建一个Product对象的各个部件指定的抽象接口。
- ConcreteBuilder：具体建造者。实现抽象接口，构建和装配各个部件。
- Director：指挥者。构建一个使用Builder接口的对象。它主要是用于创建一个复杂的对象，它主要有两个作用，一是：隔离了客户与对象的生产过程，二是：负责控制产品对象的生产过程。
- Product：产品角色。一个具体的产品对象。

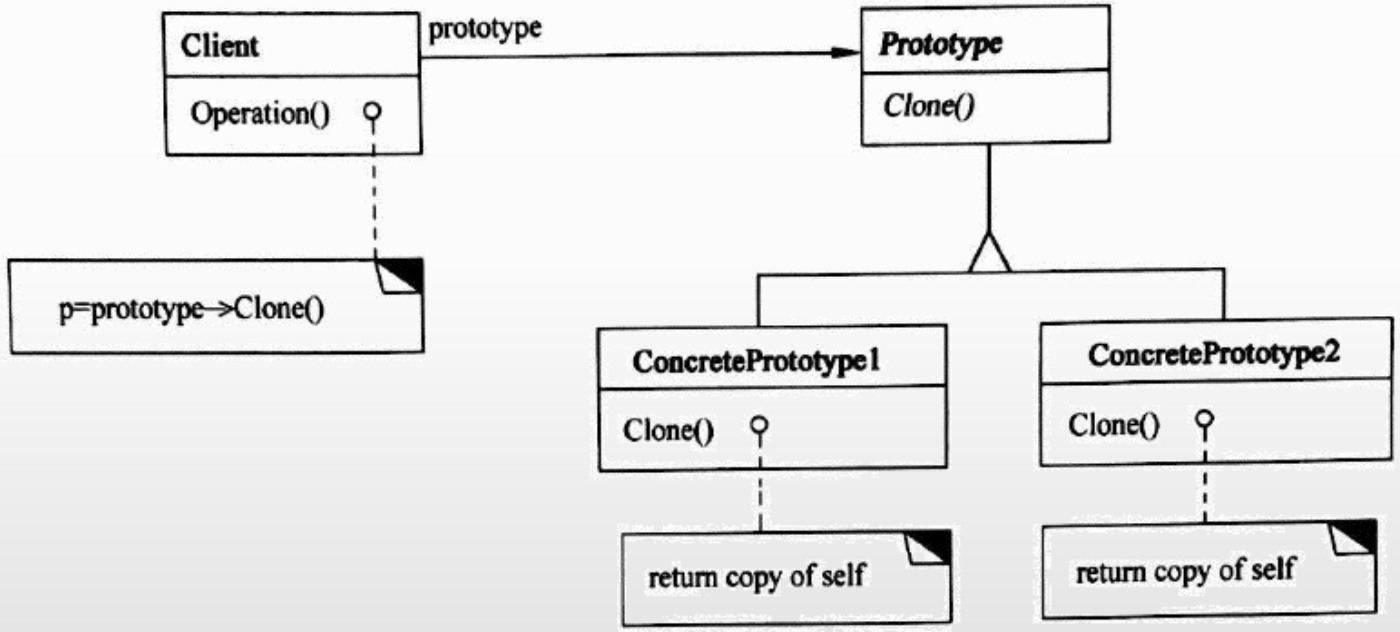
建造者模式（Builder）



Prototype原型模式：

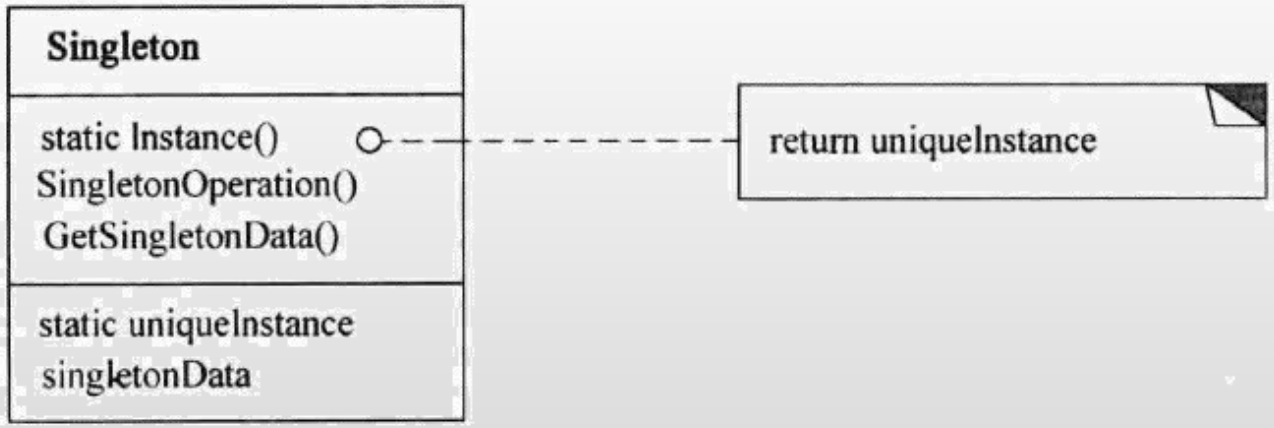
- 用原型实例指定创建对象的种类，并且通过拷贝这个原型来创建新的对象。
- 原型模式类似于细胞分裂，细胞在一定条件下，由一个分裂成2个，再由2个分裂成4个……，这个原始的细胞决定了分裂出来的细胞的组成结构。Prototype类中包括一个clone方法，Client调用其拷贝方法clone即可得到实例，不需要手工去创建实例。

Prototype原型模式：



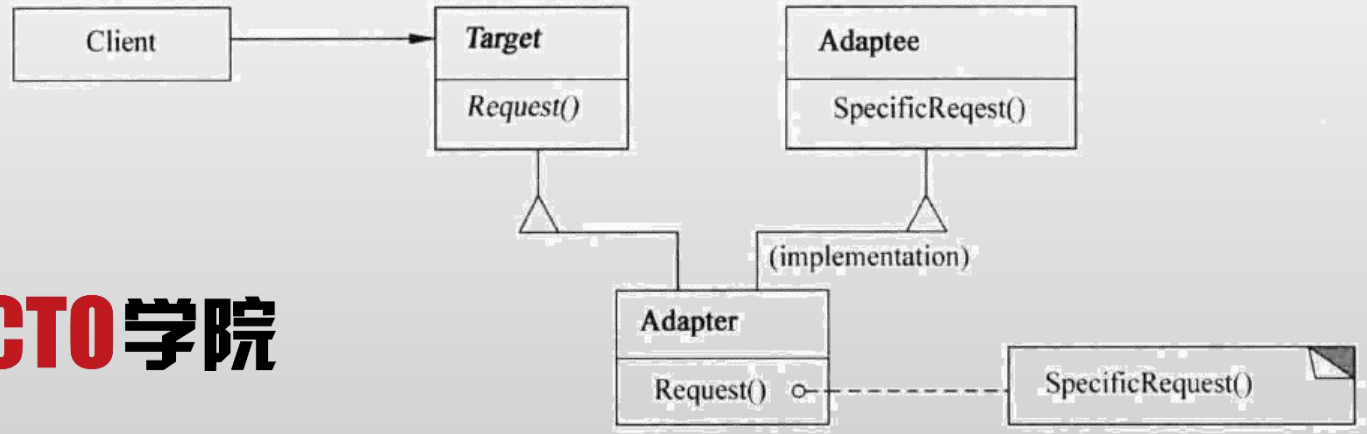
Singleton单例模式：

- 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。在计算机系统中，线程池、缓存、日志对象、对话框、打印机、显卡的驱动程序对象常被设计成单例。



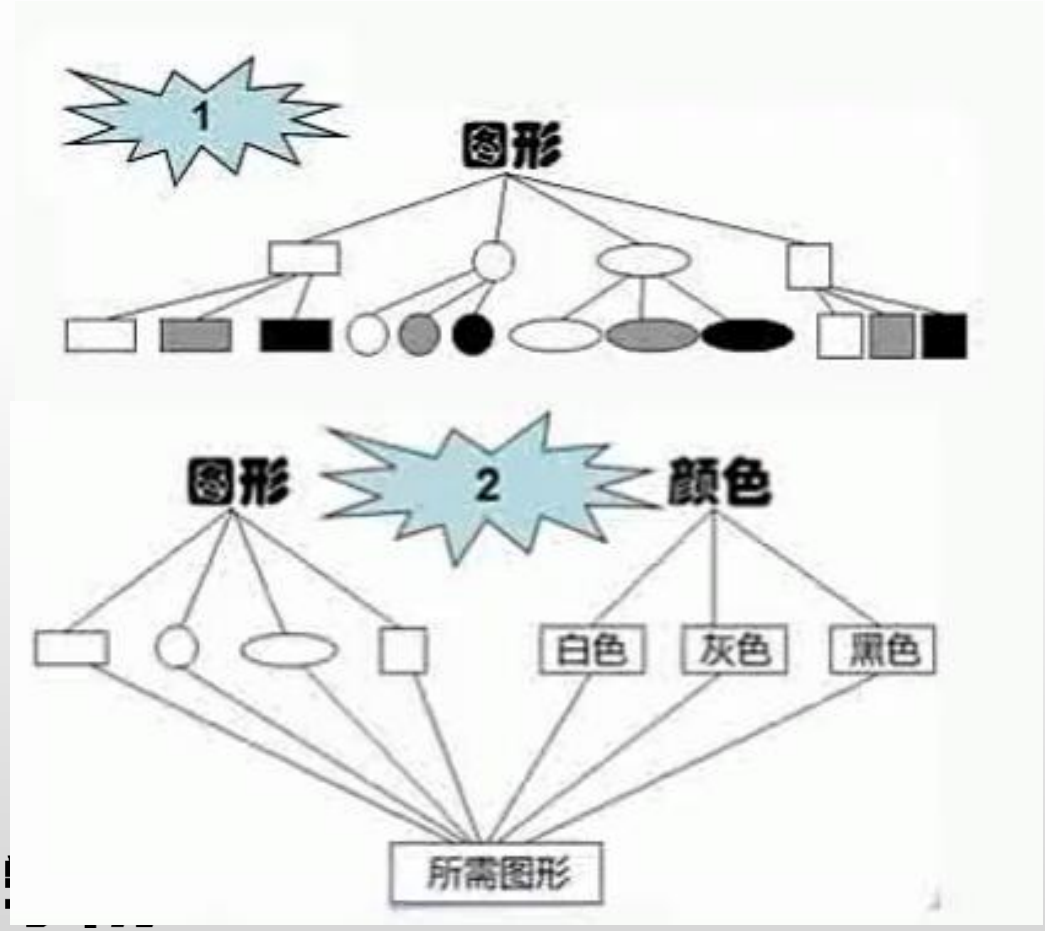
Adapter适配器模式：

- 将一个类的接口转换成客户希望的另外一个接口。Adapter模式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。如：电源适配器。
- Adapter模式适用于：
- 想使用一个已经存在的类，而它的接口不符合你的需求。
- 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（接口不一定兼容的类）协同工作。

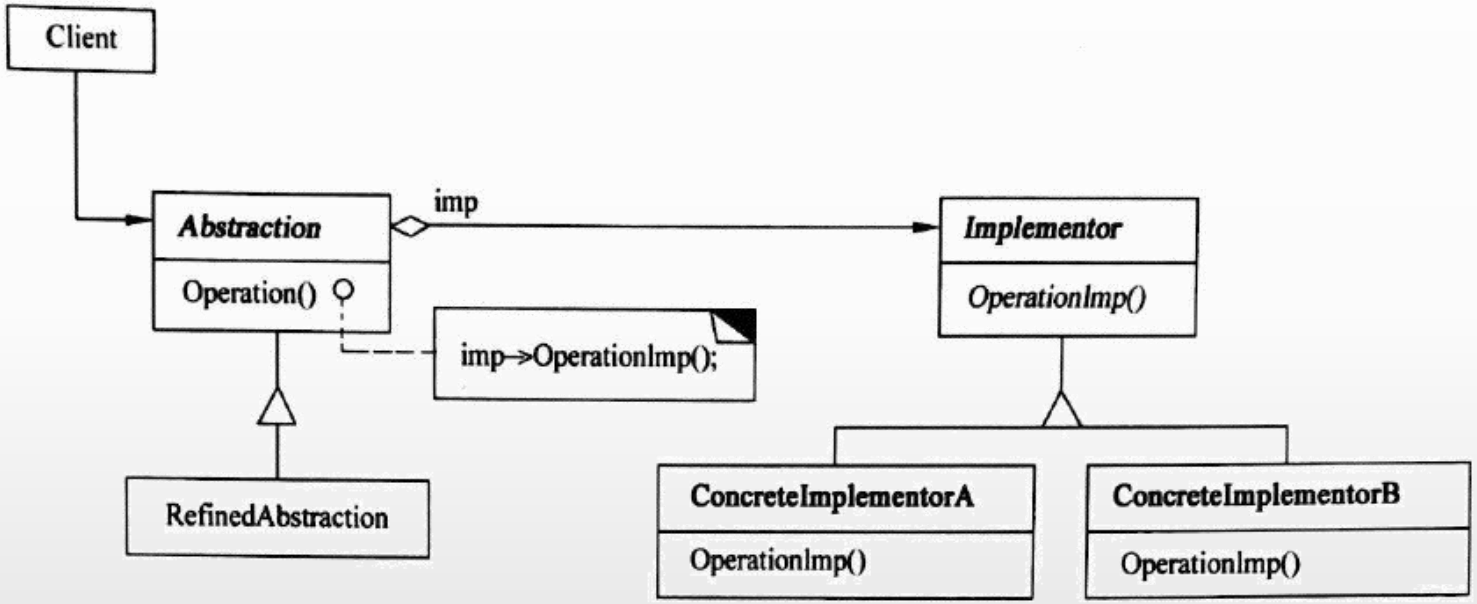


Bridge桥接模式：

- 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。



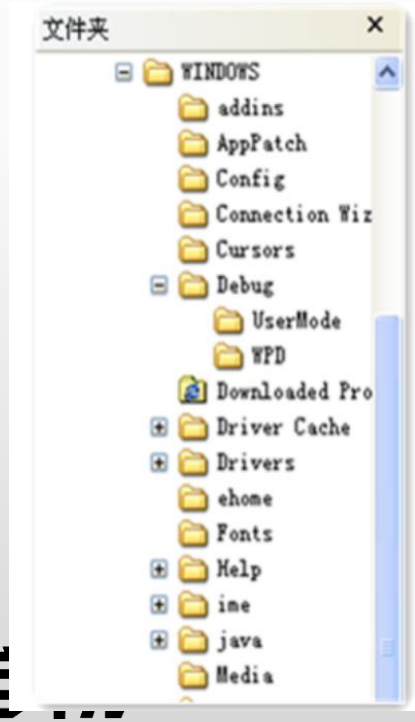
Bridge桥接模式：

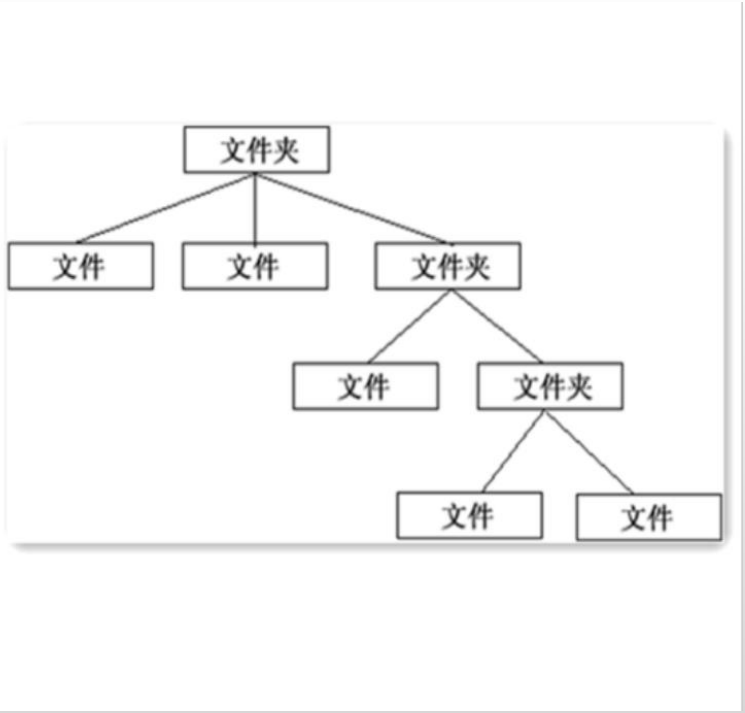


Composite组合模式：

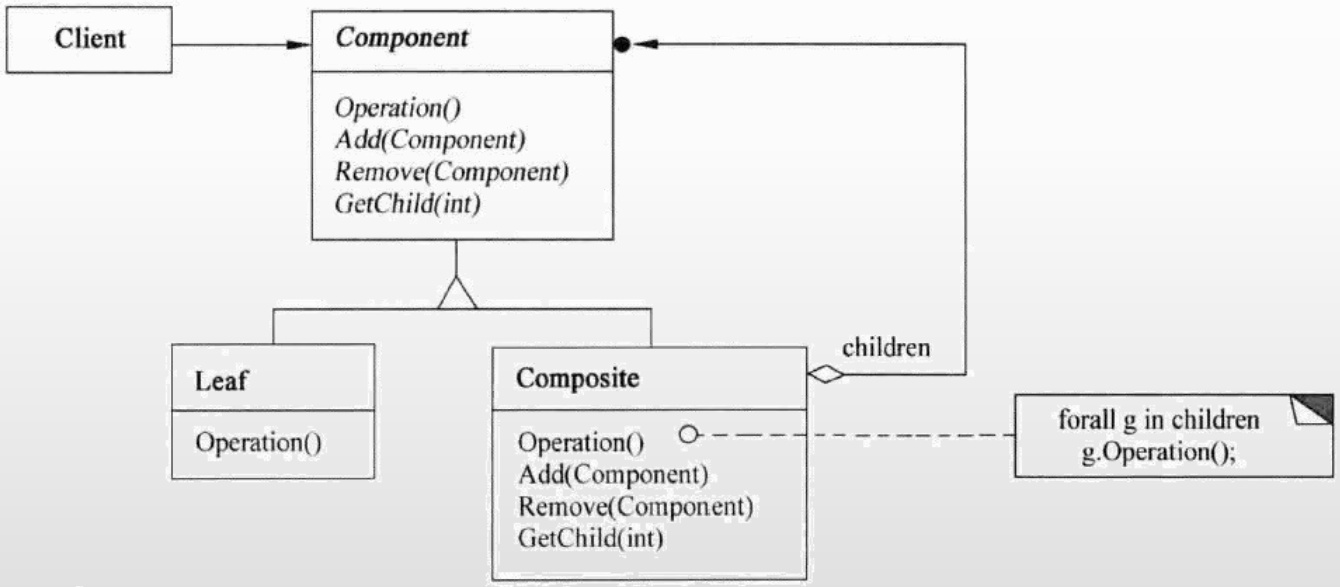
- 将对象组合成树形结构以表示“部分-整体”的层次结构。
  Composite使得客户对单个对象和复合对象的使用具有一致性。

- 组合模式描述了如何将容器对象和叶子对象进行递归组合，使得用户在使用时无须对它们进行区分，可以一致地对待容器对象和叶子对象。





Composite组合模式：

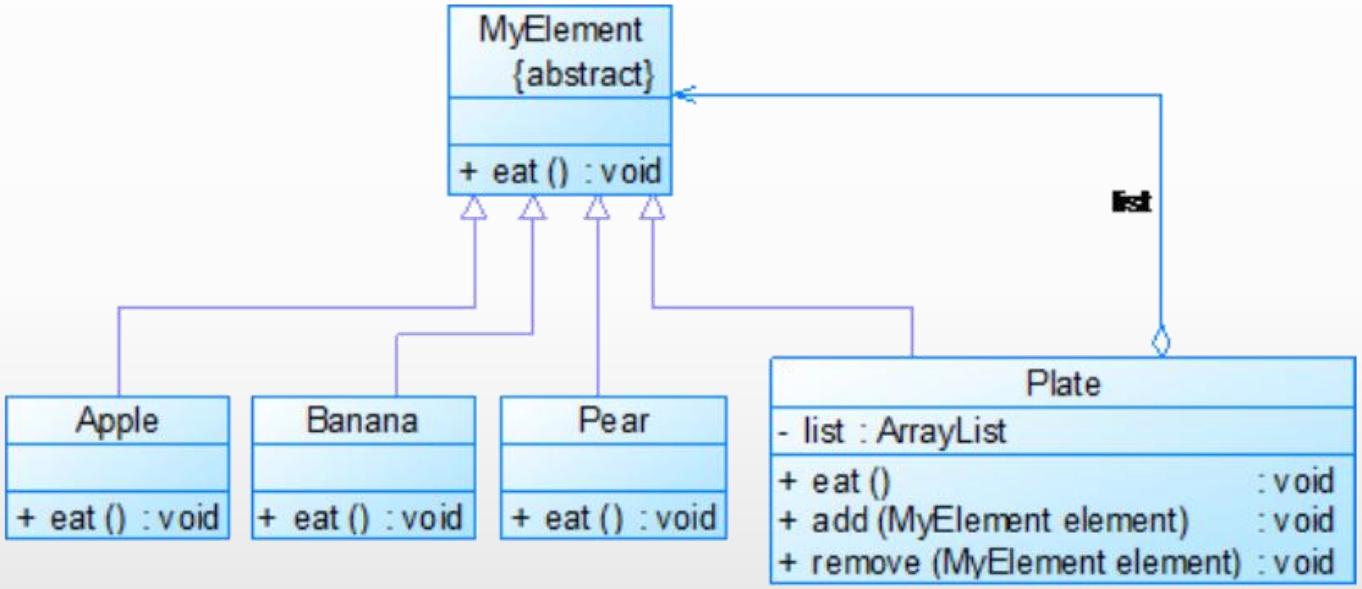


Composite组合模式：

- 在水果盘(Plate)中有一些水果，如苹果(Apple)、香蕉(Banana)、梨子(Pear)，当然大水果盘中还可以有小水果盘，现需要对盘中的水果进行遍历（吃），当然如果对一个水果盘执行“吃”方法，实际上就是吃其中的水果。
- 我们可以不管是大水果盘还是小水果盘，还是水果，都给它看做一个组件，该组件可以定义为抽象类，并且定义基本的抽象的吃方法，添加组件（组件可以是水果也可以是盘子）的方法，然后让水果类和盘子类分别继承组件类，并根据需要重写一些方法。

Composite组合模式：

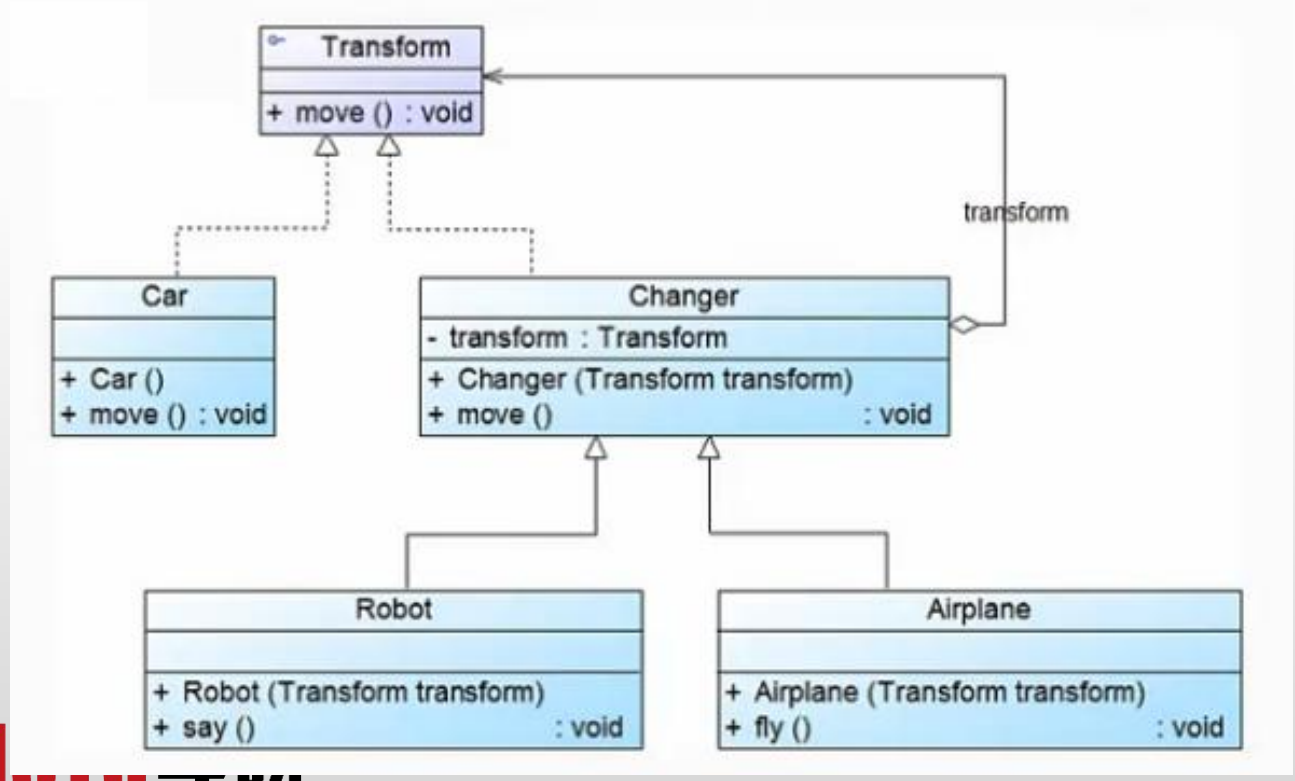
类图：



Decorator装饰模式：

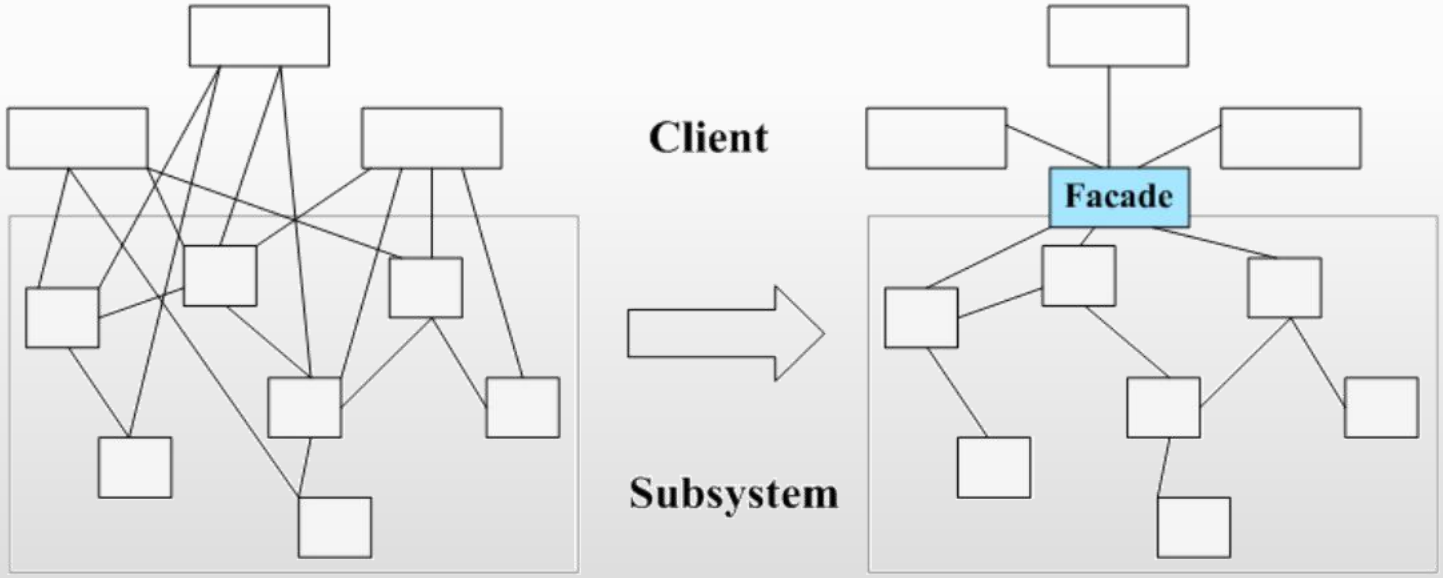
- 动态地给一个对象添加一些额外的职责。就扩展功能而言，Decorator模式比生成子类方式更为灵活。是在不改变原类文件和使用继承的情况下，动态地扩展一个对象的功能。比如，我们有一杯“茉莉茶”，现在加上一颗“柠檬”，那我们就有了一杯“柠檬茉莉花茶”。“柠檬”作为一个装饰者，提供了“茉莉茶”本身没有的清爽口感。当然，这也带来了一定的负担，你需要花更多的“钱”。

- 装饰者模式通过组合的方式扩展对象的特性，这种方式允许我们在任何时候对对象的功能进行扩展甚至是运行时扩展，而若我们用继承来完成对类的扩展则只能在编译阶段实现，所以在某些时候装饰者模式比继承（inheritance）要更加灵活。

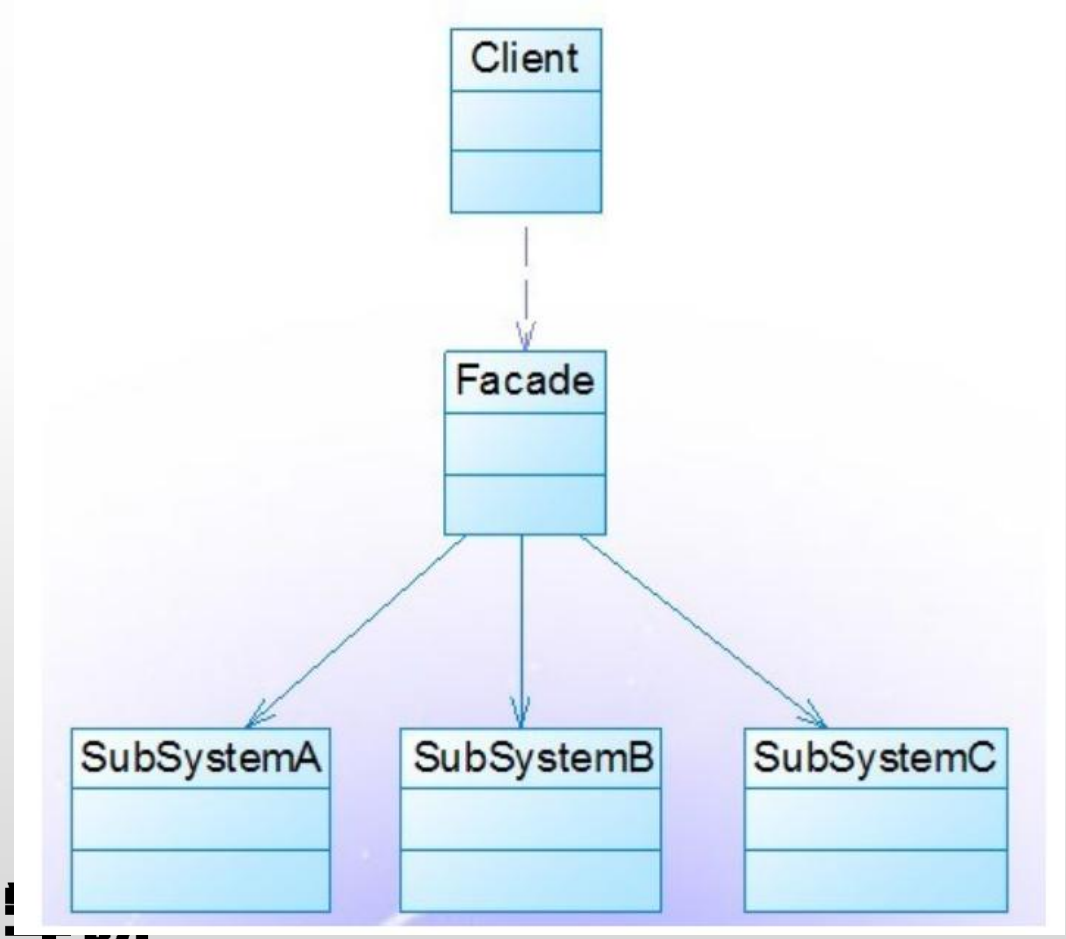


Facade 外观模式：

- **为子系统中的一组接口提供一个一致的界面，外观模式通过提供一个高层接口，隔离了外部系统与子系统间复杂的交互过程，使得复杂系统的子系统更易使用。**



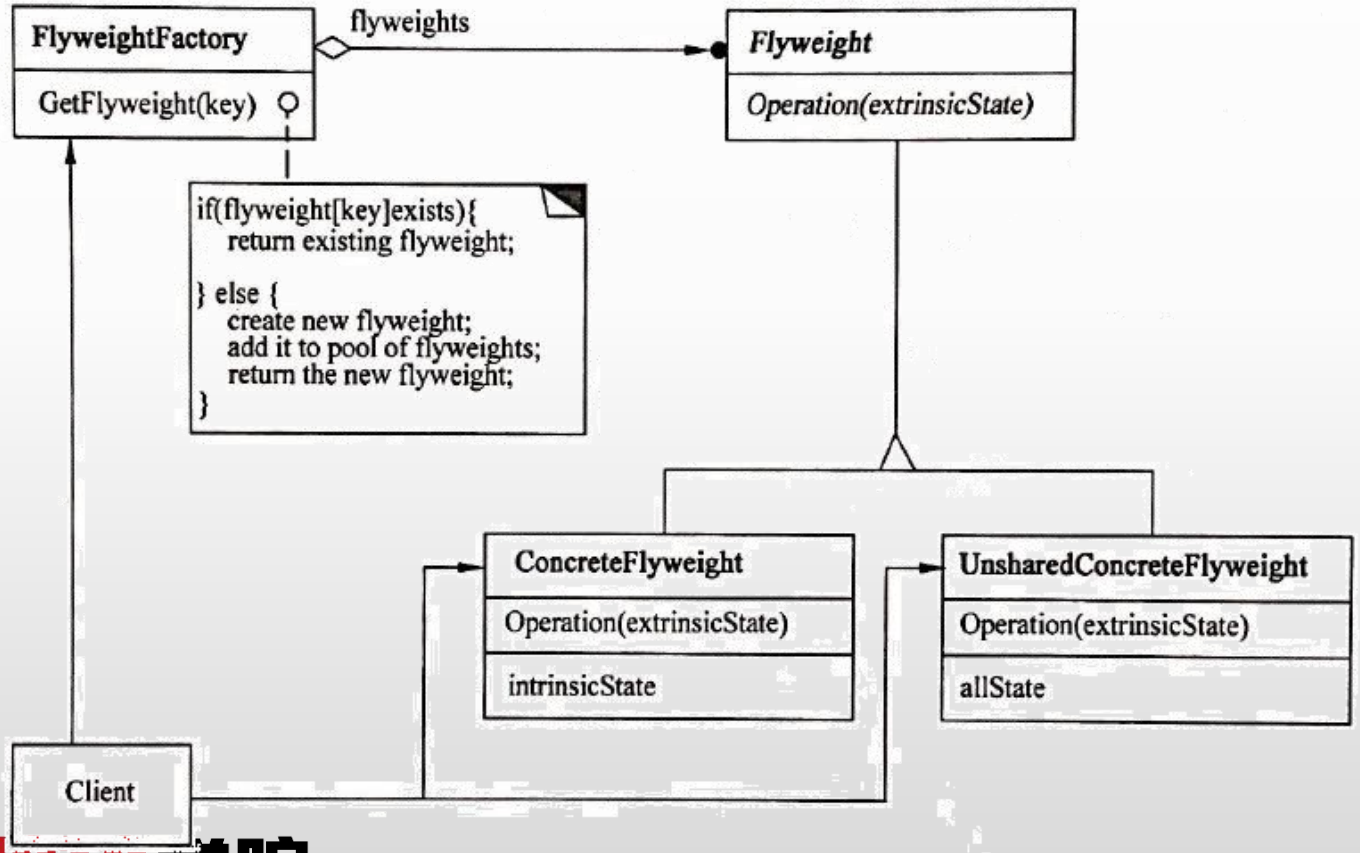
Facade 外观模式：



享元模式（Flyweight Pattern）：

- 运用共享技术有效地支持大量细粒度的对象。
- 如：内衣工厂有100种男士内衣、100中女士内衣，要求给每种内衣拍照。如果不使用享元模式则需要200个塑料模特；使用享元模式，只需要男女各1个模特。
- 又或者围棋游戏中，每个棋子都是白色或者黑色并且大小一样只是位置不同。如果每个棋子都用一个独立的对象存储，那么和上面一样会造成大量的浪费。
- 享元模式使系统使用少量的对象，而这些对象都很相似，状态变化很小，可以实现对象的多次复用。又称为轻量级模式。
- 享元模式通过共享技术实现相同或相似对象的重用。

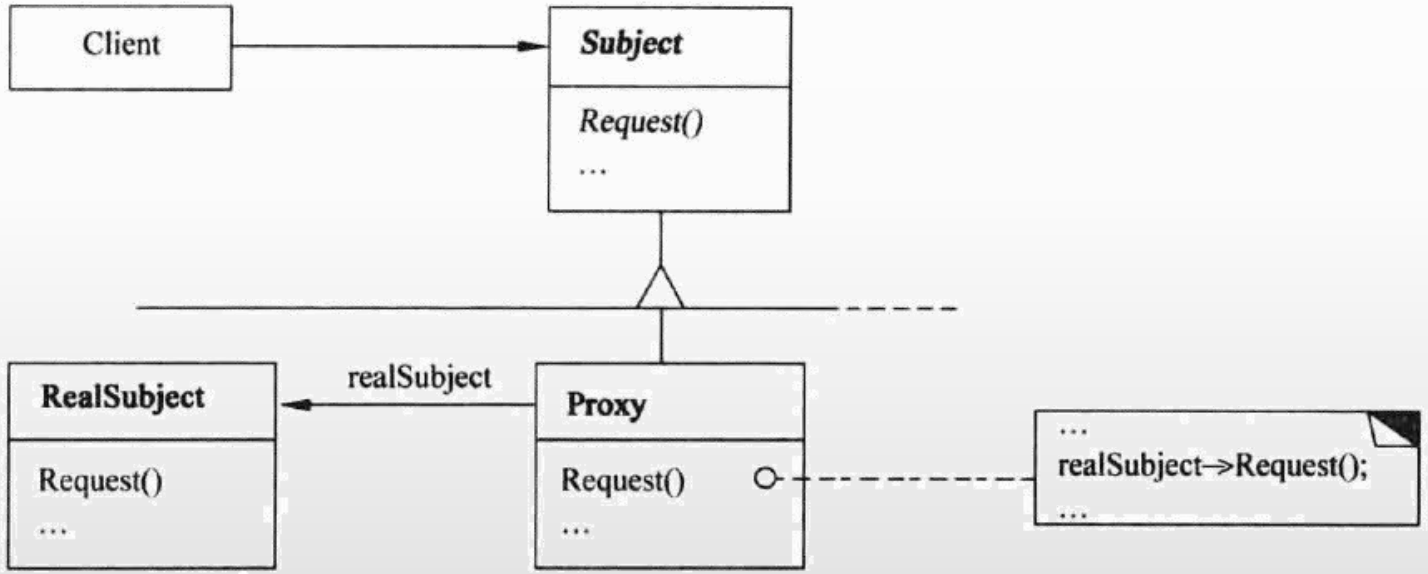
享元模式（Flyweight Pattern）：



代理模式（Proxy Pattern）：

- 为其他对象提供一种代理以控制对这个对象的访问。
- 代理模式使用代理对象完成用户请求，屏蔽用户对真实对象的访问。现实世界的代理人被授权执行当事人的一些事宜，无需当事人出面，从第三方的角度看，似乎当事人并不存在，因为他只和代理人通信。而事实上代理人是要有当事人的授权，并且在核心问题上还需要请示当事人。
- 代理模式和适配器模式应该说很相像，但是他们的区别也很明显，代理模式和被代理者的接口是同一个，只是使用中客户访问不到被代理者，所以利用代理间接的访问，而适配器模式，是因为接口不同，为了让用户使用到统一的接口，把原先的对象通过适配器让用户统一的使用，大多数运用在代码维护的后期，或者借用第三方库的情况下，而外观模式，是大家经常无意中使用的，就是把错综复杂的子系统关系封装起来，然后提供一个简单的接口给客户使用，就类似于一个转接口。

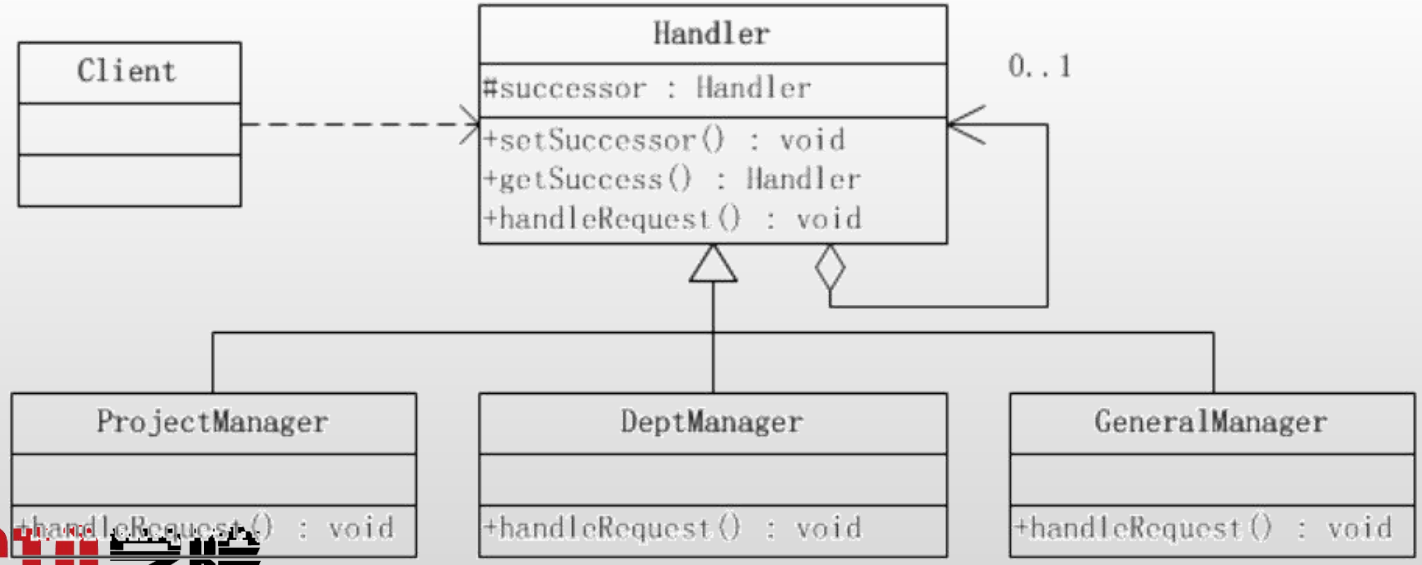
代理模式（Proxy Pattern）：



责任链模式（Chain of Responsibility Pattern）：

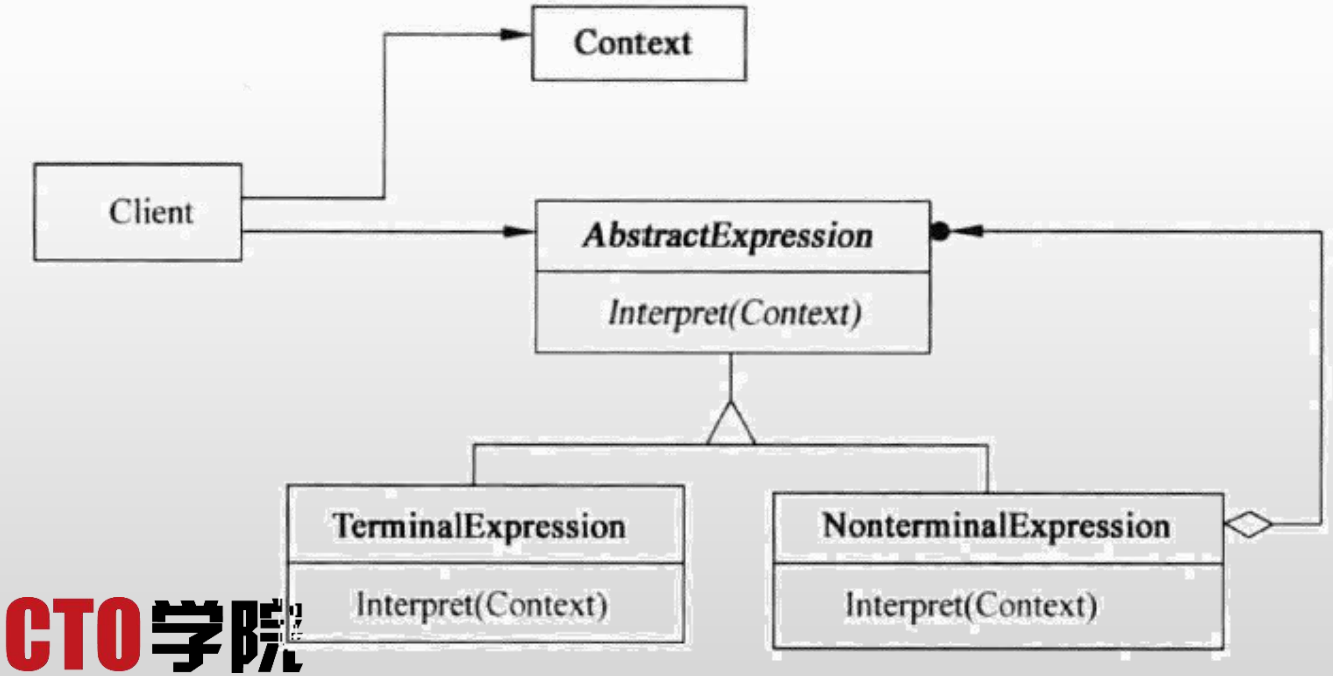
- 避免请求发送者与接收者耦合在一起，让多个对象都有可能接收请求，将这些对象连接成一条链，并且沿着这条链传递请求，直到有对象处理它为止。
- 如申请报销费用，大致流程一般是，由申请人先填写申请单，然后交给领导审批，如果申请批准下来，领导会通知申请人审批通过，然后申请人去财务领取费用，如果没有批准下来，领导会通知申请人审批未通过，此事也就此作罢。
- 不同级别的领导，对于审批的额度是不一样的，比如，项目经理只能审批500元以内的申请；部门经理能审批1000元以内的申请；而总经理可以审核任意额度的申请。

- 当某人提出报销申请的请求后，该请求会经由项目经理、部门经理、总经理之中的某一位领导来进行相应的处理，但是提出申请的人并不知道最终会由谁来处理他的请求，一般申请人是把自己的申请提交给项目经理，或许最后是由总经理来处理他的请求。申请人只要直接与项目经理交互就可以，其余的工作在黑盒中，究竟流程是怎样的，最后是由谁审批通过的，申请人无需关心。



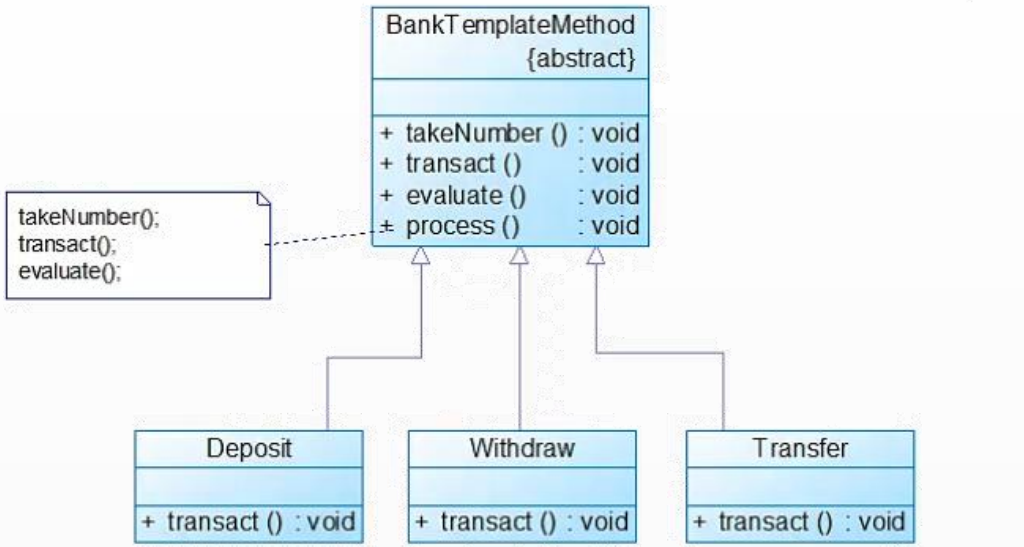
解释器模式（Interpreter Pattern）：

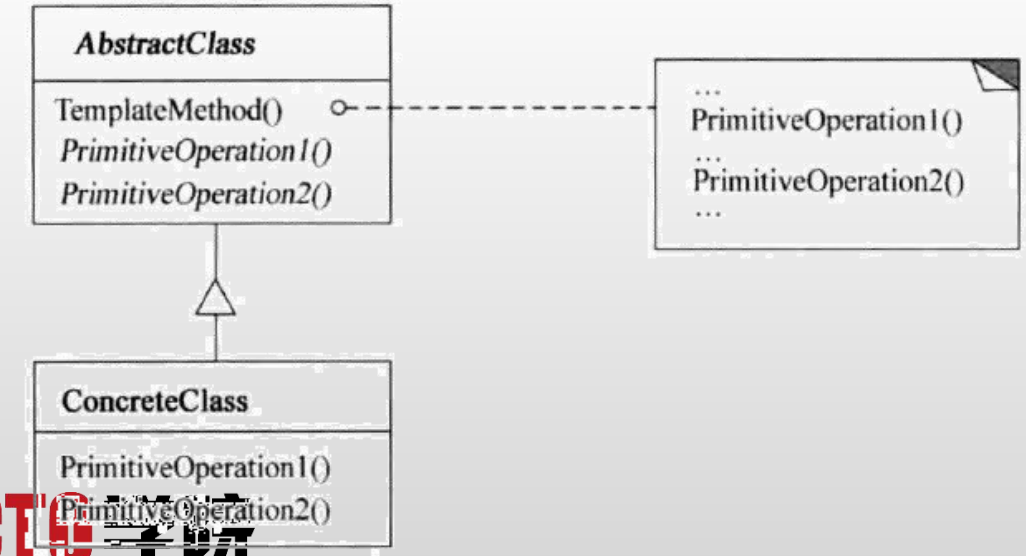
- 给定一个语言，定义它的文法的一种表示，并定义一个解释器，
  这个解释器使用该表示来解释语言中的句子。
- 例如：构造一个语言解释器，使得系统可以执行整数间的乘、除和求模运算。如用户输入表达式”2*3/4%5”



模板方法（Template Pattern）：

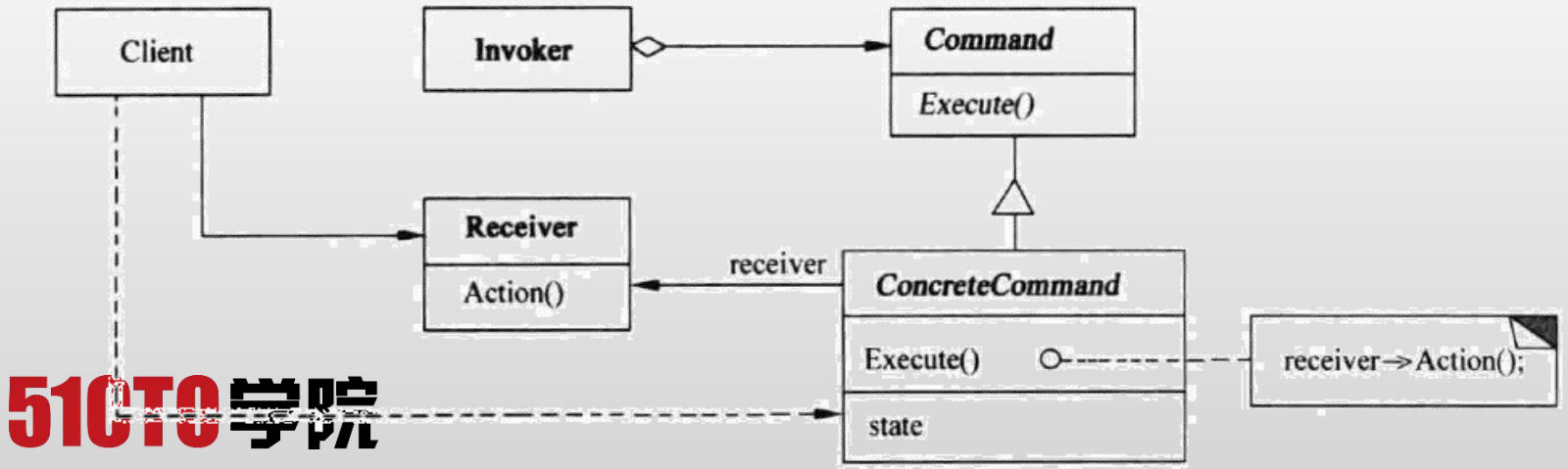
- 定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。模板方法使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。
- 在父类中定义一个完成该事情的总方法，按照完成事件需要的步骤去调用其每个步骤的实现方法。每个步骤的具体实现，由子类完成。
- 在银行办理业务的时候，步骤一般是，排队取号、等待叫号、办理业务、为银行柜员打分，这个流程无论是办理存款、取款还是转账业务都是一样的。


BankTemplateMethod
{abstract}
+ takeNumber () : void
+ transact () : void
+ evaluate () : void
± process () : void
takeNumber();
transact();
evaluate();
Deposit
+ transact () : void
Withdraw
+ transact () : void
Transfer
+ transact () : void


AbstractClass
TemplateMethod()
PrimitiveOperation1()
PrimitiveOperation2()
...
PrimitiveOperation1()
...
PrimitiveOperation2()
...
ConcreteClass
PrimitiveOperation1()
PrimitiveOperation2()

命令模式（Command Pattern）：

- 将一个请求封装成一个对象，从而使得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。
- 命令模式将发出命令的责任和执行命令的责任分割开。请求的一方不必知道接收请求的一方的接口，也不必知道请求是怎么被接收的，以及操作是否被执行、何时被执行以及怎样被执行的。

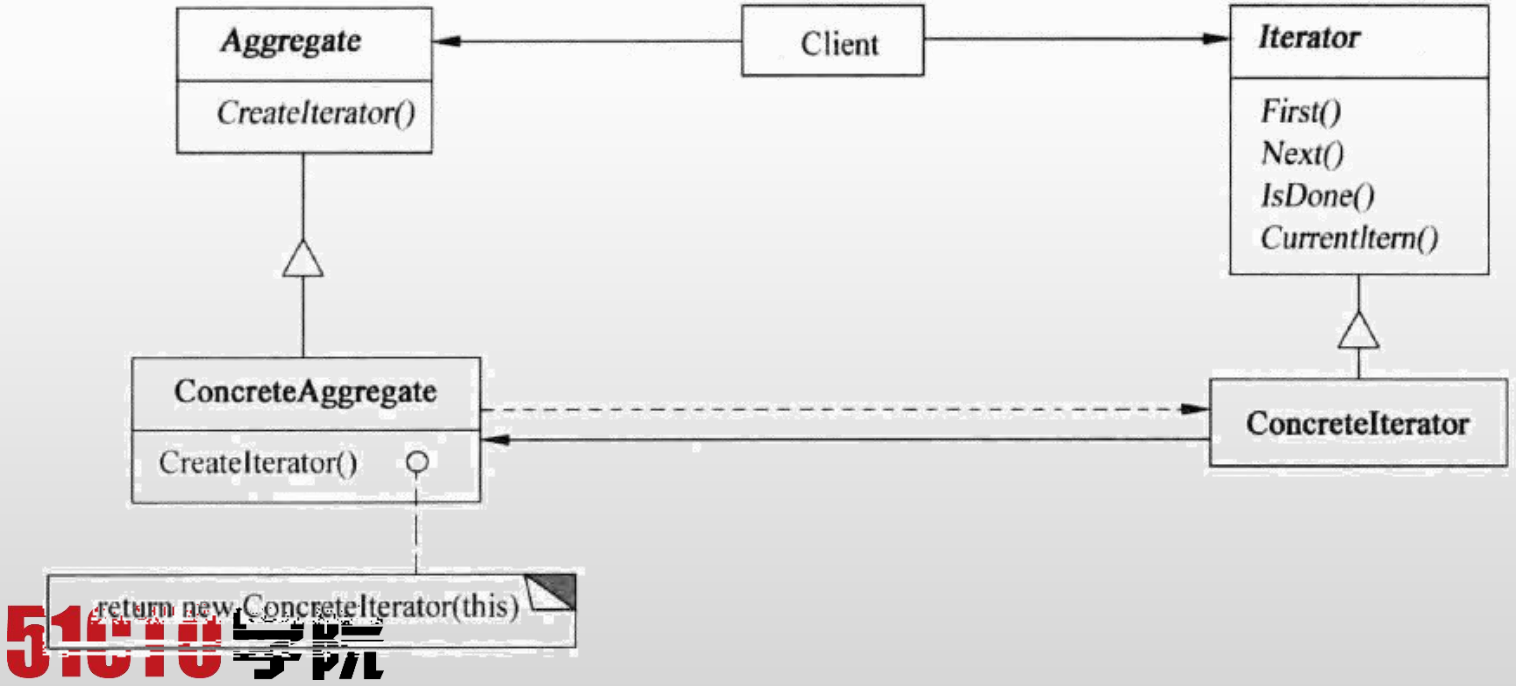


命令模式（Command Pattern）：

- 这就好像以前电视没有遥控的时候，如果你想更换频道的话，必须跑到电视机前，手动按换台按钮，如果台很多的话，而你家的电视又比较古老，那么你就得更辛苦一点，得手动调出来其他频道。
- 现在有了遥控器，解放了我们，我们（相当于请求者），想换台，不用跑来跑去的调台，而是向电视机发出换台的请求，遥控器接收到我们的请求后，对电视发出具体的命令，是换台，关机，还是开机。这样，我们把请求传给遥控器，遥控器来具体执行需要哪些命令。这样，我们就和电视机解耦了。虽然结果是一样的，都是换了台，但是过程确截然不同。我们和电视之间没有任何联系了。而联系只存在于遥控器和具体的命令，以及命令和电视之间了。

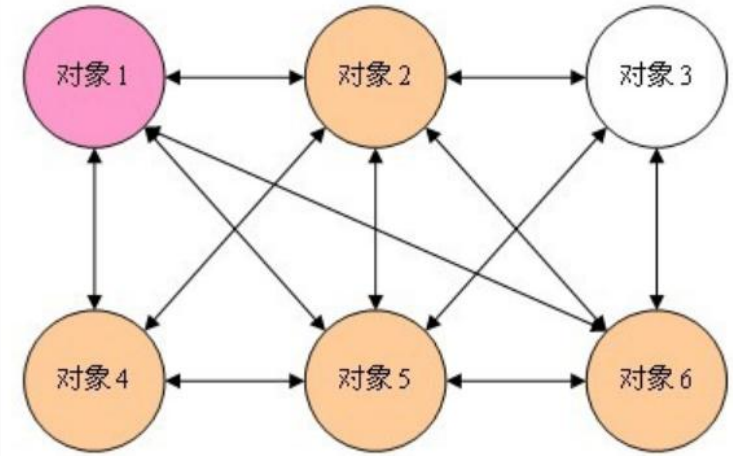
迭代器模式（Iterator Pattern）：

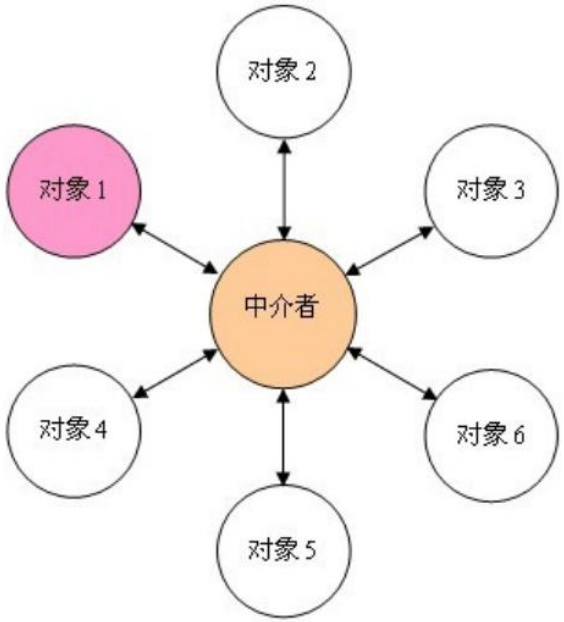
- **提供一种方法顺序访问一个聚合对象中各个元素，而又无须暴露该对象的内部表示。**
- **如看电视换频道，我们可以遍历所有的频道而不需要知道频道间是如何转换的。**

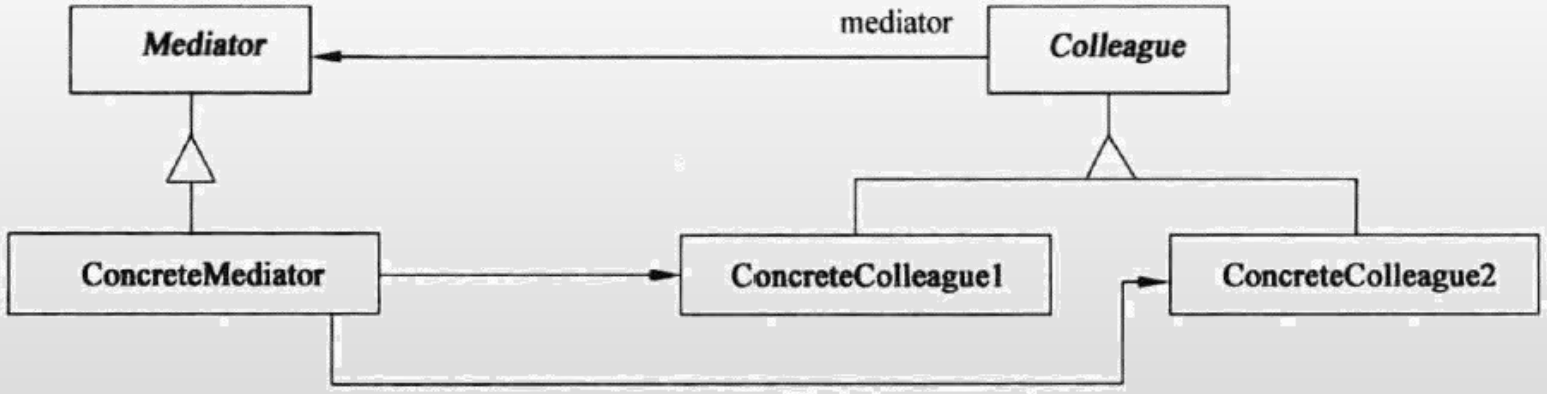


中介者模式（Mediator Pattern）：

- **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。**
- **比如系统和各个硬件，系统作为中介者，各个硬件作为同事者，当一个同事的状态发生改变的时候，不需要告诉其他每个硬件自己发生了变化，只需要告诉中介者系统，系统会通知每个硬件某个硬件发生了改变，其他的硬件会做出相应的变化；**


对象 1
对象 2
对象 3
对象 4
对象 5
对象 6


对象 2
对象 1
对象 3
中介者
对象 4
对象 6
对象 5


Mediator
mediator
Colleague
ConcreteMediator
ConcreteColleague1
ConcreteColleague2

备忘录模式（Memento Pattern）：

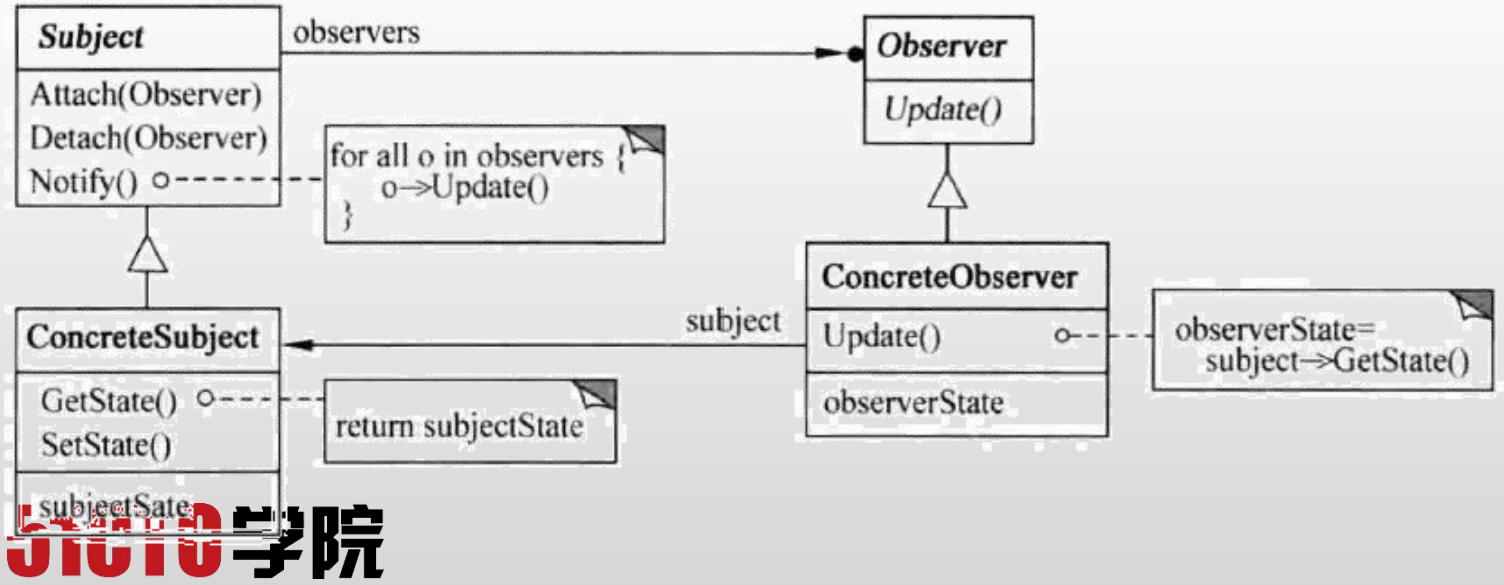
- 在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。这样就可以将该对象恢复到原先保存的状态。
  如word的撤销键， 简单的说备忘录模式就是在想让对象回到原来某个时间点的状态时，可以通过撤销来简单的实现。

备忘录模式（Memento Pattern）：

- Originator可以根据需要决定Memento存储自己的哪些内部状态。
- Memento(备忘录)：负责存储Originator对象的内部状态，并可以防止Originator以外的其他对象访问备忘录。
- Caretaker(管理者):负责备忘录Memento，不能对Memento的内容进行访问或者操作。
- 涉及角色：
- Originator(发起人)：负责创建一个备忘录Memento，用以记录当前时刻自身的内部状态，并可使用备忘录恢复内部状态。

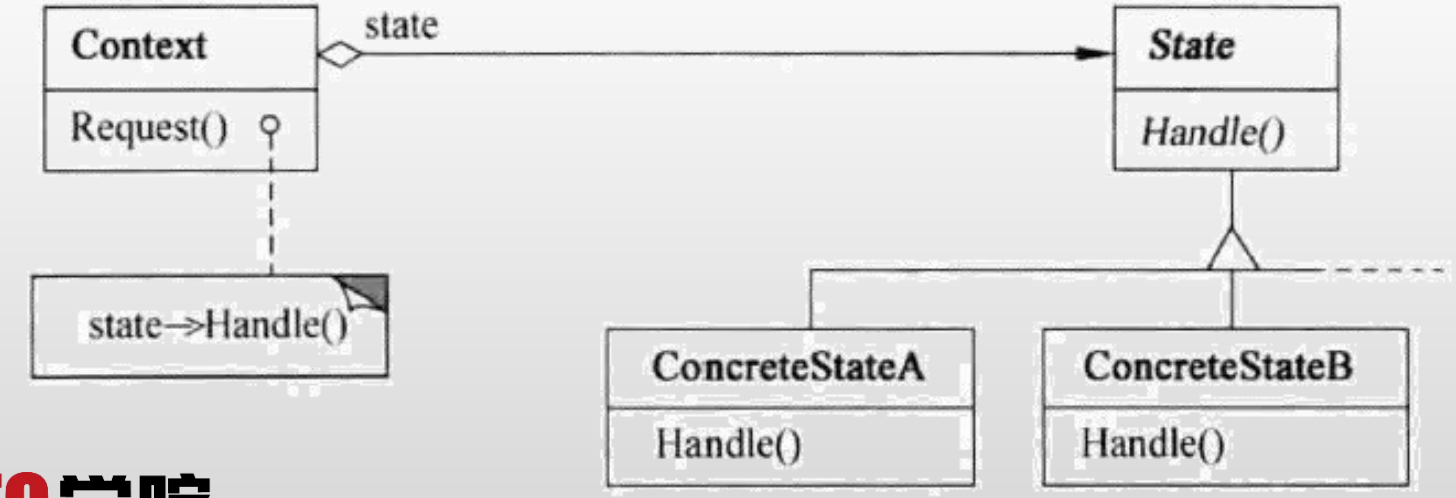
观察者模式（Observer Pattern）：

- 观察者模式定义了对象间的一种一对多依赖关系，使得每当一个对象改变状态，则所有依赖于它的对象都会得到通知并被自动更新。
- 发生改变的对象称为观察目标，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。

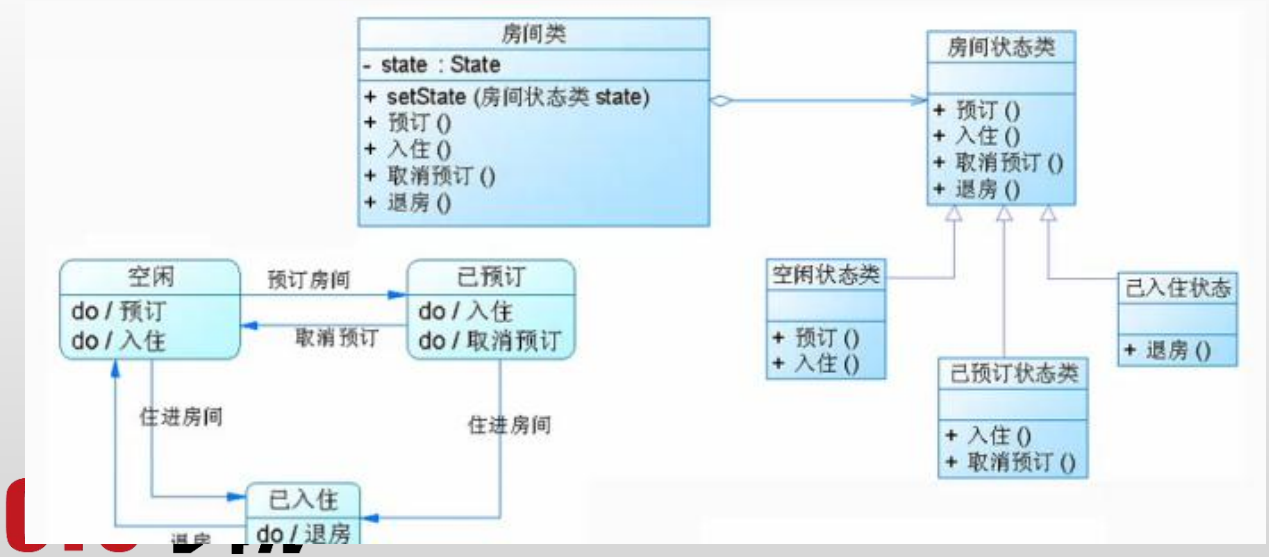


状态模式（State Pattern）：

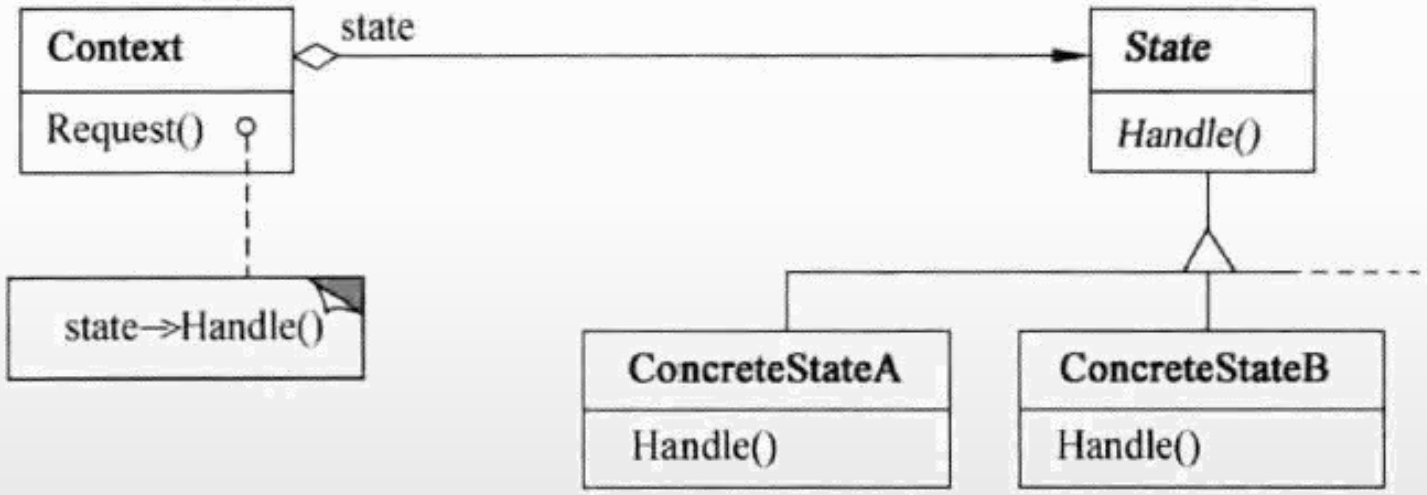
- 对于对象内部的状态，允许其在不同的状态下，拥有不同的行为，对状态单独封装成类。
- 例如：我有一个订单，订单状态有未提交，待审核，审核通过，审核失败四个状态，订单的变化应该是：点击提交订单时，订单状态由未提交变成待审核；点击审核（待审核的订单）订单，可以变成审核通过或审核失败。



- 对于对象内部的状态，允许其在不同的状态下，拥有不同的行为，对状态单独封装成类。
- 例如：我有一个订单，订单状态有未提交，待审核，审核通过，审核失败四个状态，订单的变化应该是：点击提交订单时，订单状态由未提交变成待审核；点击审核（待审核的订单）订单，可以变成审核通过或审核失败。

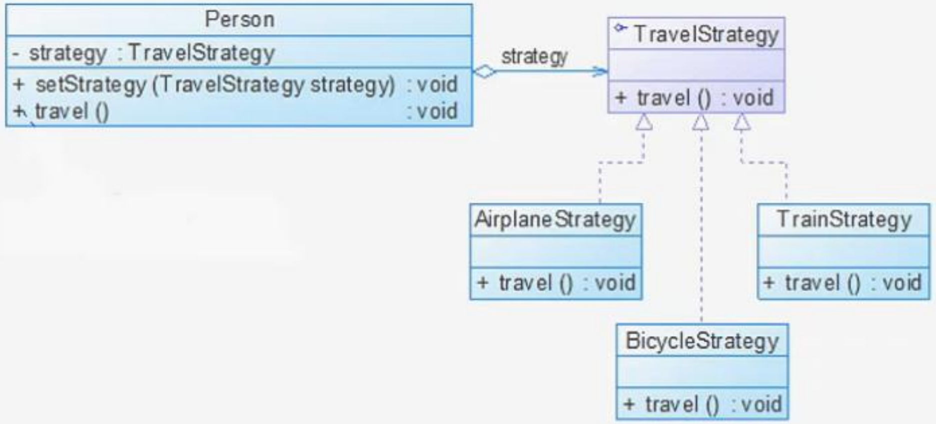


状态模式（State Pattern）：

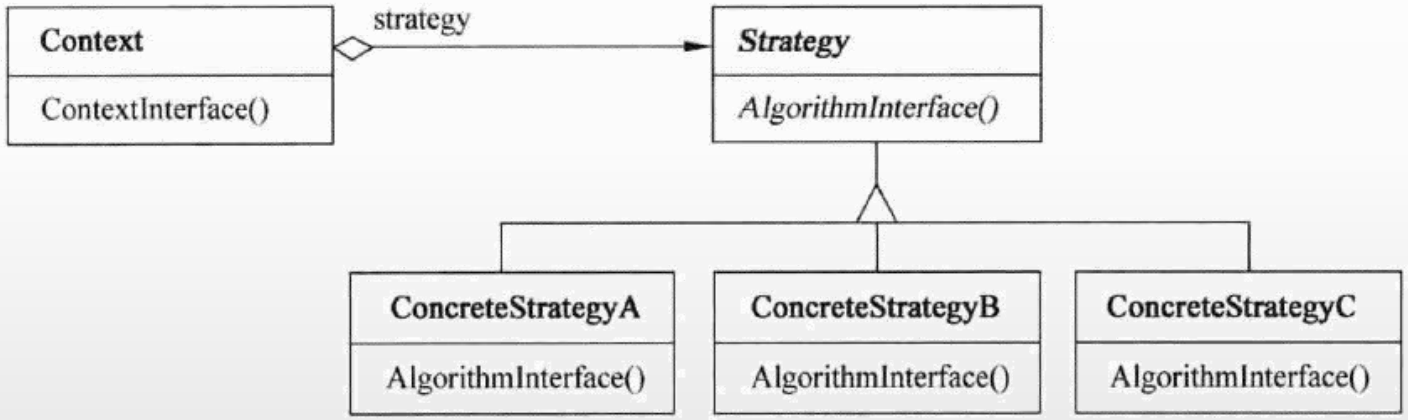


策略模式（Strategy Pattern）：

- 策略模式定义了一系列的算法，并将每一个算法封装起来，而且使它们还可以相互替换。策略模式让算法独立于使用它的客户而独立变化。
- 出行旅游：我们可以有几个策略可以考虑：可以骑自行车，汽车，做火车，飞机。每个策略都可以得到相同的结果，但是它们使用了不同的资源。选择策略的依据是费用，时间，使用工具还有每种方式的方便程度 。

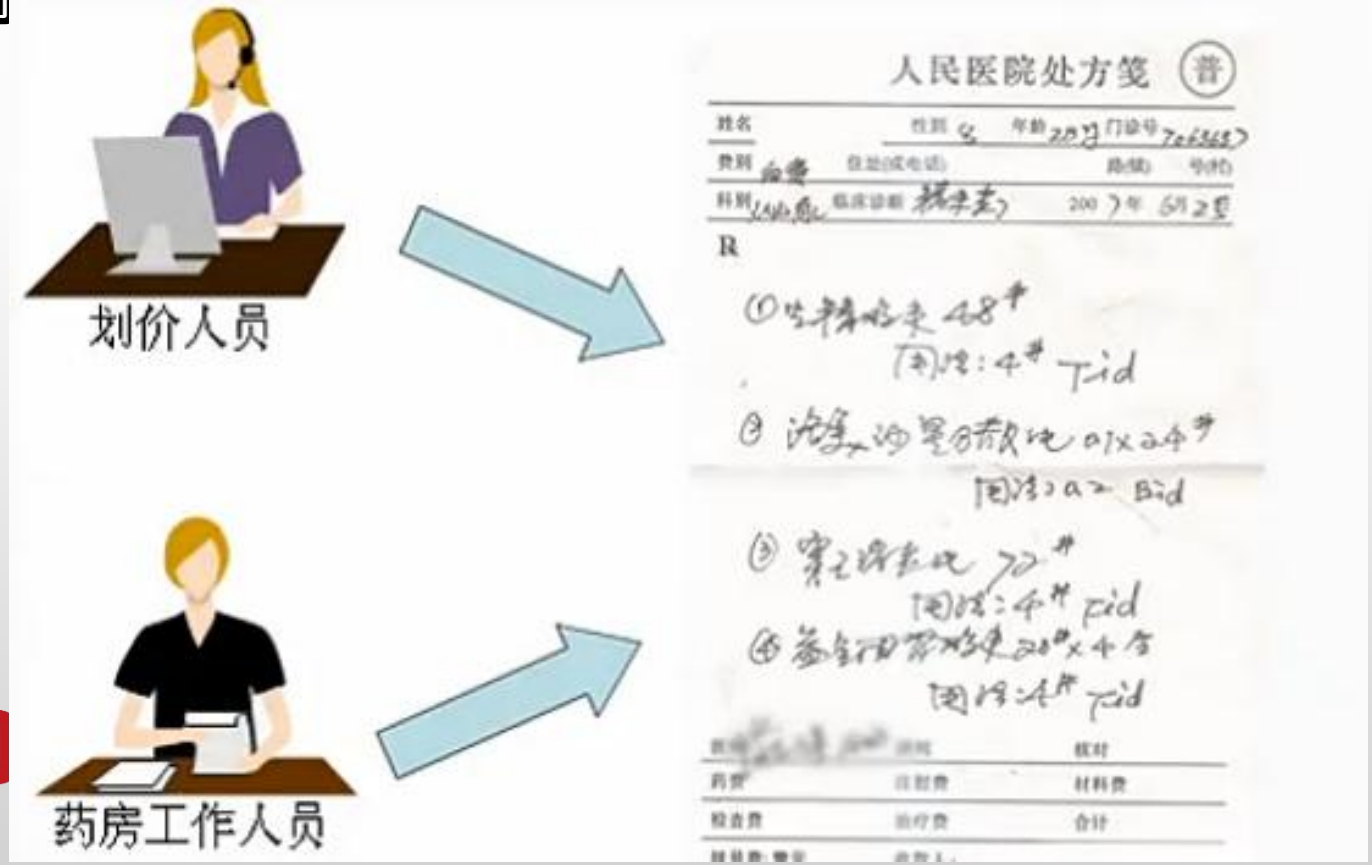


策略模式（Strategy Pattern）：

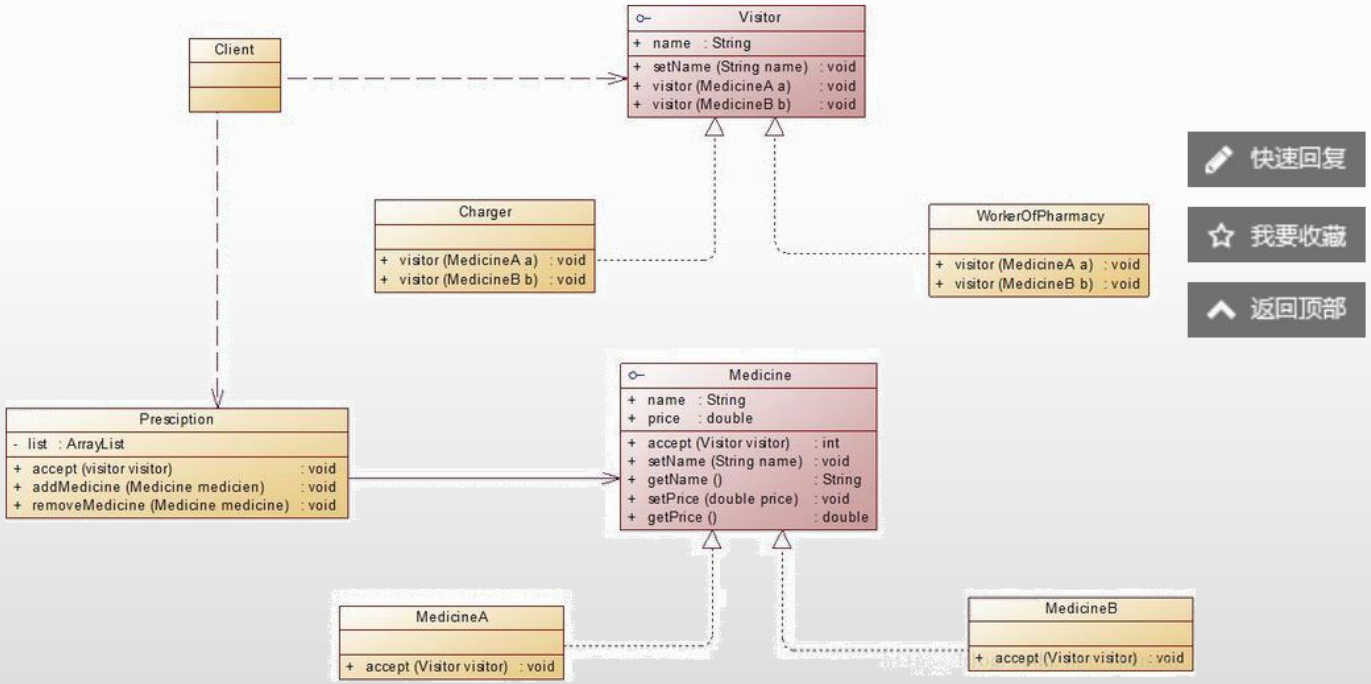


访问者模式（Visitor Pattern）：

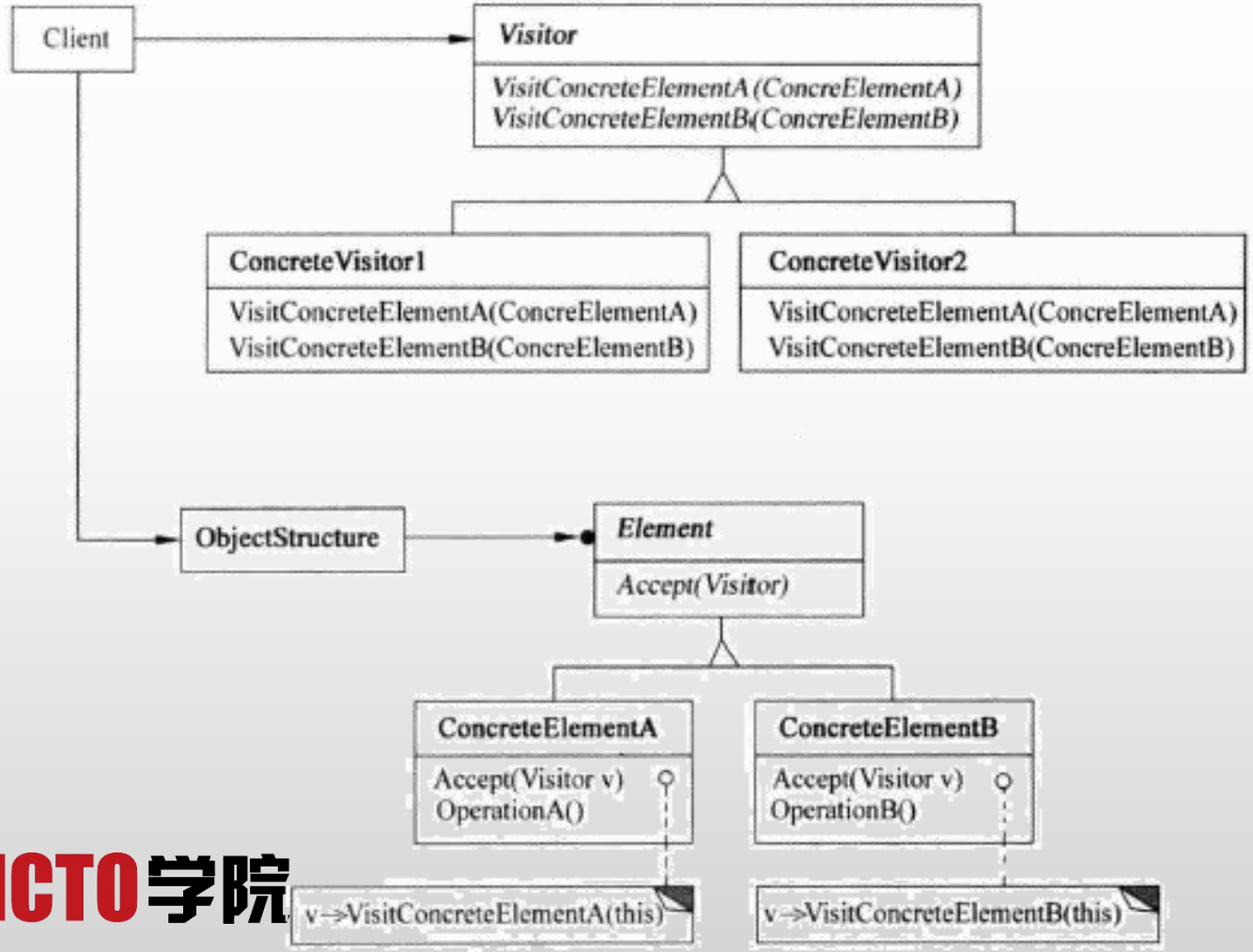
- **表示一个作用于某对象结构中的各元素的操作。它使你可以在不改变各元素的类的前提下定义作用与这些元素的新操作。即对于某个对象或者一组对象，不同的访问者，产生的结果不同，执行操作也不同**



访问者模式（Visitor Pattern）：



访问者模式（Visitor Pattern）：



典型真题

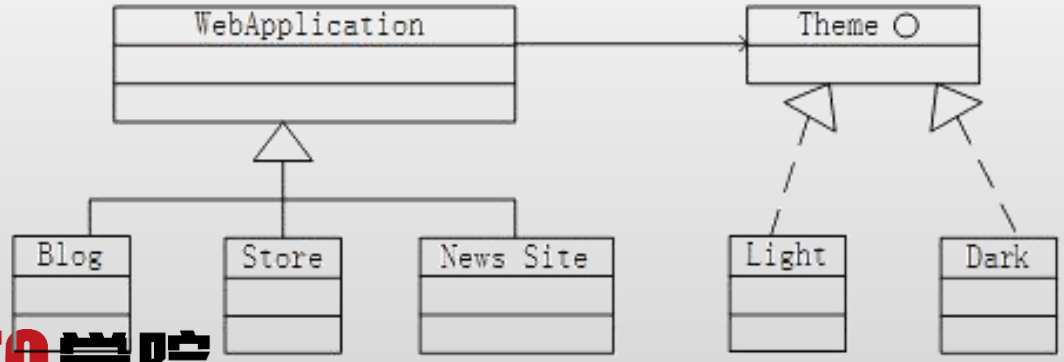
- （ ）设计模式将一个请求封装为一个对象，从而使得可以用不同的请求对客户进行参数化，对请求排队或记录请求日志，以及支持可撤销的操作。
- A.命令（Command）
- B.责任链（Chain of Responsibility）
- C.观察者（Observer）
- D.策略（Strategy）

典型真题

- 试题分析
- 命令模式的特点为：将一个请求封装为一个对象，从而可用不同的请求对客户进行参数化，将请求排队或记录请求日志，支持可撤销的操作。本题描述为命令模式。
- 职责链模式（Chain of Responsibility）：通过给多个对象处理请求的机会，减少请求的发送者与接收者之间的耦合。将接收对象链接起来，在链中传递请求，直到有一个对象处理这个请求。
- 观察者模式（Observer）：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自动更新。
- 答案：A

典型真题

- 假设现在要创建一个Web应用框架，基于此框架能够创建不同的具体Web应用，比如博客、新闻网站和网上商店等；并可以为每个Web应用创建不同的主题样式，如浅色或深色等。这一业务需求的类图设计适合采用（ ）模式（如下图所示）。其中（ ）是客户程序使用的主要接口，维护对主题类型的引用。此模式为（ ），体现的最主要的意图是（ ）。



- A.观察者（Observer）　　B.访问者（Visitor）
- C.策略（Strategy）　　D.桥接（Bridge）
- A. WebApplication　　B. Blog
- C. Theme　　D. Light
- A.创建型对象模式　　B.结构型对象模式
- C.行为型类模式　　D.行为型对象模式
- A.将抽象部分与其实现部分分离，使它们都可以独立地变化
- B.动态地给一个对象添加一些额外的职责
- C.为其他对象提供一种代理以控制对这个对象的访问
- D.将一个类的接口转换成客户希望的另外一个接口
- 答案：D、A、B、A

技术成就梦想