**一、依赖关系(Dependence)**

依赖关系（Dependence）：假设A类的变化引起了B类的变化，则说名B类依赖于A类。

•  依赖关系(Dependency) 是一种使用关系，特定事物的改变有可能会影响到使用该事物的其他事物，在需要表示一个事物使用另一个事物时使用依赖关系。大多数情况下，依       赖关系体现在某个类的方法使用另一个类的对象作为参数。

•   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。



1. **public class Driver**
2. {
3. **public void drive(Car car)**
4. {
5. car.move();
6. }
7. ……
8. }
9. **public class Car**
10. {
11. **public void move()**
12. {
13. ......
14. }
15. ……
16. }

依赖关系有如下三种情况：

1、A类是B类中的（某中方法的）局部变量；

2、A类是B类方法当中的一个参数；

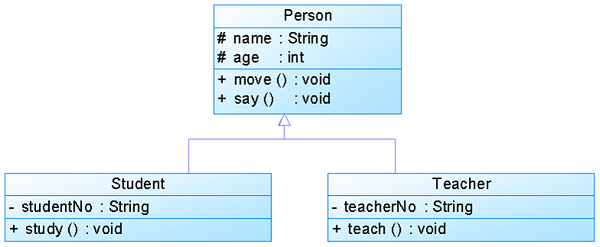
3、A类向B类发送消息，从而影响B类发生变化；

**二、泛化关系（Generalization）**

泛化关系（Generalization）：A是B和C的父类，B,C具有公共类（父类）A，说明A是B,C的一般化（概括，也称泛化）

•  泛化关系(Generalization)也就是继承关系，也称为“is-a-kind-of”关系，泛化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。

•  在代码实现时，使用面向对象的继承机制来实现泛化关系，如在Java语言中使用extends关键字、在C++/C#中使用冒号“：”来实现。



1. **public class Person**
2. {
3. **protected String name;**
4. **protected int age;**
5. **public void move()**
6. {
7. ……
8. }
9. **public void say()**
10. {
11. ……
12. }
13. }
14. **public class Student extends Person**
15. {
16. **private String studentNo;**
17. **public void study()**
18. {
19. ……
20. }
21. }

在UML当中，对泛化关系有三个要求：

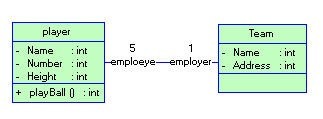
1、子类与父类应该完全一致，父类所具有的属性、操作，子类应该都有；

2、子类中除了与父类一致的信息以外，还包括额外的信息；

3、可以使用父类的实例的地方，也可以使用子类的实例；

**三、关联关系（Association）**

关联关系（Association）:类之间的联系，如客户和订单，每个订单对应特定的客户，每个客户对应一些特定的订单，再如篮球队员与球队之间的关联（下图所示）。



其中，关联两边的"employee"和“employer”标示了两者之间的关系，而数字表示两者的关系的限制，是关联两者之间的多重性。通常有“\*”（表示所有，不限），“1”（表示有且仅有一个），“0...”（表示0个或者多个），“0，1”（表示0个或者一个），“n...m”(表示n到m个都可以),“m...\*”（表示至少m个）。

•  关联关系(Association) 是类与类之间最常用的一种关系，它是一种结构化关系，用于表示一类对象与另一类对象之间有联系。

•  在UML类图中，用实线连接有关联的对象所对应的类，在使用Java、C#和C++等编程语言实现关联关系时，通常将一个类的对象作为另一个类的属性。

•  在使用类图表示关联关系时可以在关联线上标注角色名。

1)  双向关联: 默认情况下，关联是双向的。

https://img-my.csdn.net/uploads/201206/07/1339064456_6833.png

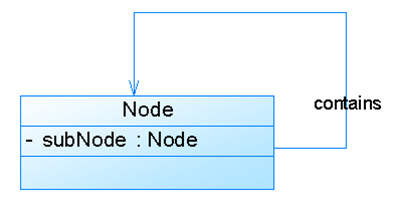
1. **public class Customer**
2. {
3. **private Product[] products;**
4. ……
5. }
6. **public class Product**
7. {
8. **private Customer customer;**
9. ……
10. }

2 ) 单向关联:类的关联关系也可以是单向的，单向关联用带箭头的实线表示.

https://img-my.csdn.net/uploads/201206/07/1339064543_2384.png

1. **public class Customer**
2. {
3. **private Address address;**
4. ……
5. }
7. **public class Address**
8. {
9. ……
10. }

3) 自关联: 在系统中可能会存在一些类的属性对象类型为该类本身，这种特殊的关联关系称为自关联。



1. **public class Node**
2. {
3. **private Node subNode;**
4. ……
5. }

4) 重数性关联: 重数性关联关系又称为多重性关联关系(Multiplicity)，表示一个类的对象与另一个类的对象连接的个数。在UML中多重性关系可以直接在关联直线上增加一个数字表示与之对应的另一个类的对象的个数。

|  |  |
| --- | --- |
| 表示方式 | 多重性说明 |
| 1..1 | 表示另一个类的一个对象只与一个该类对象有关系 |
| 0..\* | 表示另一个类的一个对象与零个或多个该类对象有关系 |
| 1..\* | 表示另一个类的一个对象与一个或多个该类对象有关系 |
| 0..1 | 表示另一个类的一个对象没有或只与一个该类对象有关系 |
| m..n | 表示另一个类的一个对象与最少m、最多n个该类对象有关系 (m<=n) |



1. **public class Form**
2. {
3. **private Button buttons[];**
4. ……
5. }
6. **public class Button**
7. {
8. …
9. }

**四、聚合关系（Aggregation）**

聚合关系（Aggregation）:表示的是整体和部分的关系，整体与部分 可以分开.

•  聚合关系(Aggregation) 表示一个整体与部分的关系。通常在定义一个整体类后，再去分析这个整体类的组成结构，从而找出一些成员类，该整体类和成员类之间就形成了聚合   关系。

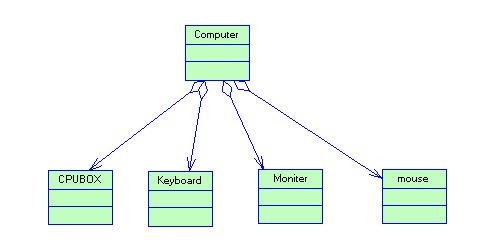
•  在聚合关系中，成员类是整体类的一部分，即成员对象是整体对象的一部分，但是成员对象可以脱离整体对象独立存在。在UML中，聚合关系用带空心菱形的直线表示。



1. **public class Car**
2. {
3. **private Engine engine;**
4. **public Car(Engine engine)**
5. {
6. **this.engine = engine;**
7. }
9. **public void setEngine(Engine engine)**
10. {
11. **this.engine = engine;**
12. }
13. ……
14. }
15. **public class Engine**
16. {
17. ……
18. }

如：电话机包括一个话筒

       电脑包括键盘、显示器，一台电脑可以和多个键盘、多个显示器搭配，确定键盘和显示器是可以和主机分开的，主机可以选择其他的键盘、显示器组成电脑；

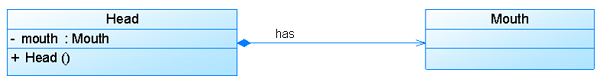


**五、组合关系（Composition）**

组合关系（Composition）:也是整体与部分的关系，但是整体与部分不可以分开.

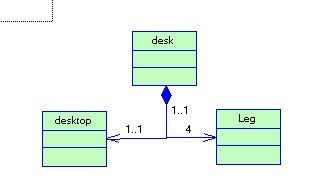
•  组合关系(Composition)也表示类之间整体和部分的关系，但是组合关系中部分和整体具有统一的生存期。一旦整体对象不存在，部分对象也将不存在，部分对象与整体对象之    间具有同生共死的关系。

•  在组合关系中，成员类是整体类的一部分，而且整体类可以控制成员类的生命周期，即成员类的存在依赖于整体类。在UML中，组合关系用带实心菱形的直线表示。



1. **public class Head**
2. {
3. **private Mouth mouth;**
4. **public Head()**
5. {
6. mouth = **new Mouth();**
7. }
8. ……
9. }
11. **public class Mouth**
12. {
13. ……
14. }

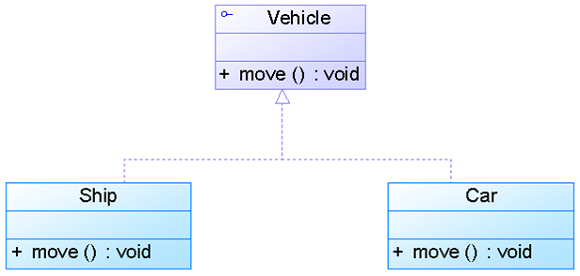
如：公司和部门，部门是部分，公司是整体，公司A的财务部不可能和公司B的财务部对换，就是说，公司A不能和自己的财务部分开； 人与人的心脏.



**六、实现关系（Implementation)**

实现关系（Implementation）：是用来规定接口和实线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。



1. **public interface Vehicle**
2. {
3. **public void move();**
4. }
5. **public class Ship implements Vehicle**
6. {
7. **public void move()**
8. {
9. ……
10. }
11. }
12. **public class Car implements Vehicle**
13. {
14. **public void move()**
15. {
16. ……
17. }
18. }

# UML的9中图例概述

作为一种建模语言，UML的定义包括UML语义和UML表示法两个部分。

l UML语义：描述基于UML的精确元模型定义。

l UML表示法：定义UML符号的表示法，为开发者或开发工具使用这些图形符号和文本语法为系统建模提供了标准。这些图形符号和文字所表达的是应用级的模型，在语义上它是UML元模型的实例。

标准建模语言UML可以由下列5类图来定义。

l 用例图：从用户角度描述系统功能，并指出各功能的操作者。

l 静态图：包括类图和对象图。类图描述系统中类的静态结构，不仅定义系统中的类，表示类之间的联系，如关联、依赖、聚合等，也包括类的属性和操作，类图描述的是一种静态关系，在系统的整个生命周期都是有效的。对象图是类图的实例，几乎使用与类图完全相同的标识。一个对象图是类图的一个实例。由于对象存在生命周期，因此对象图只能在系统某一时间段存在。

l 行为图：描述系统的动态模型和组成对象间的交互关系，包括状态图和活动图。状态图描述类的对象所有可能的状态以及事件发生时状态的转移条件，状态图是对类图的补充，活动图描述满足用例要求所要进行的活动以及活动间的约束关系，有利于识别并进行活动。

l 交互图：描述对象间的交互关系，包括时序图和协作图。时序图显示对象之间的动态合作关系，它强调对象之间消息发送的顺序，同时显示对象之间的交互；协作图描述对象间的协作关系，协作图跟时序图相似，显示对象间的动态合作关系。除显示信息交换外，协作图还显示对象以及它们之间的关系。如果强调时间和顺序，则使用时序图；如果强调上下级关系，则选择协作图。

l 实现图：包括组件图和部署图。组件图描述代码部件的物理结构及各部件之间的依赖关系，组件图有助于分析和理解部件之间的相互影响程度；部署图定义系统中软硬件的物理体系结构。

采用UML来设计系统时，第一步是描述需求；第二步根据需求建立系统的静态模型，以构造系统的结构；第三步是描述系统的行为。其中在第一步与第二步中所建立的模型都是静态的，包括用例图、类图、对象图、组件图和部署图等5种图形，是标准建模语言UML的静态建模机制。其中第三步中所建立的模型或者可以执行，或者表示执行时的时序状态或交互关系。它包括状态图、活动图、时序图和协作图等4种图形，是标准建模语言UML的动态建模机制。

首先对UML中的各个图的功用做一个简单介绍：

1、用例图

    描述角色以及角色与用例之间的连接关系。说明的是谁要使用系统，以及他们使用该系统可以做些什么。一个用例图包含了多个模型元素，如系统、参与者和用例，并且显示了这些元素之间的各种关系，如泛化、关联和依赖。

2、类图

    类图是描述系统中的类，以及各个类之间的关系的静态视图。能够让我们在正确编写代码以前对系统有一个全面的认识。类图是一种模型类型，确切的说，是一种静态模型类型。

3、对象图

    与类图极为相似，它是类图的实例，对象图显示类的多个对象实例，而不是实际的类。它描述的不是类之间的关系，而是对象之间的关系。

4、活动图

    描述用例要求所要进行的活动，以及活动间的约束关系，有利于识别并行活动。能够演示出系统中哪些地方存在功能，以及这些功能和系统中其他组件的功能如何共同满足前面使用用例图建模的商务需求。

5、状态图

    描述类的对象所有可能的状态，以及事件发生时状态的转移条件。可以捕获对象、子系统和系统的生命周期。他们可以告知一个对象可以拥有的状态，并且事件(如消息的接收、时间的流逝、错误、条件变为真等)会怎么随着时间的推移来影响这些状态。一个状态图应该连接到所有具有清晰的可标识状态和复杂行为的类；该图可以确定类的行为，以及该行为如何根据当前的状态变化，也可以展示哪些事件将会改变类的对象的状态。状态图是对类图的补充。

6、序列图（顺序图）

    序列图是用来显示你的参与者如何以一系列顺序的步骤与系统的对象交互的模型。顺序图可以用来展示对象之间是如何进行交互的。顺序图将显示的重点放在消息序列上，即强调消息是如何在对象之间被发送和接收的。

7、协作图

    和序列图相似，显示对象间的动态合作关系。可以看成是类图和顺序图的交集，协作图建模对象或者角色，以及它们彼此之间是如何通信的。如果强调时间和顺序，则使用序列图；如果强调上下级关系，则选择协作图；这两种图合称为交互图。

8、构件图 （组件图）

    描述代码构件的物理结构以及各种构建之间的依赖关系。用来建模软件的组件及其相互之间的关系，这些图由构件标记符和构件之间的关系构成。在组件图中，构件时软件单个组成部分，它可以是一个文件，产品、可执行文件和脚本等。

9、部署图 （配置图）

    是用来建模系统的物理部署。例如计算机和设备，以及它们之间是如何连接的。部署图的使用者是开发人员、系统集成人员和测试人员。

几种图的区别：

一：这九种模型图各有侧重，

1:用例图侧重描述用户需求，

2:类图侧重描述系统具体实现；

二：描述的方面都不相同，

1:类图描述的是系统的结构，

2:序列图描述的是系统的行为；

三：抽象的层次也不同，

1：构件图描述系统的模块结构，抽象层次较高，

2：类图是描述具体模块的结构，抽象层次一般，

3：对象图描述了具体的模块实现，抽象层次较低。

在有的文献书籍中，将这九种模型图分为三大类：

结构分类、动态行为和模型管理：

1：结构分类包括用例图、类图、对象图、构件图和部署图，

2：动态行为包括状态图、活动图、顺序图和协作图，

3：模型管理则包含类图。

画图说明

UML(统一建模语言)：是面向对象的可视化建模的一种语言。是数据库设计过程中，在E-R图（实体-联系图）的设计后的进一步建模。  
UML中有3种构造块：事物、关系和图，事物是对模型中最具有代表性的成分的抽象；关系是把事物结合在一起；图聚集了相关的的事物。具体关系图标如下：

说明：  
构件事物是名词，是模型的静态部分。  
行为事物是动态部分，表示行为。  
分组事物是组织部分。  
注释事物是解释部分。

依赖：一个事物变化会引起另一个事物变化。  
聚集：特殊的关联，描述整体与部分的组合关系。  
泛化：是一种特殊与一般的关系，如子元素（特殊）与父元素（一般），箭头指向父元素。  
实现：类元之间的关系，其中一个类元指定了由另一个类元保证执行的契约。一般用在接口和实现他们的类之间或用例和实现它们的协作之间。

UML提供9种视图：类图、对象图，用例图，序列图、协作图，状态图、活动图，构件图和部署图。

在UML系统开发中有三个主要的模型：

功能模型: 从用户的角度展示系统的功能，包括用例图。  
对象模型: 采用对象，属性，操作，关联等概念展示系统的结构和基础，包括类图。  
动态模型: 展现系统的内部行为。 包括序列图，活动图，状态图。

下面具体说明：

1.类图：描述一组对象、接口、协作等事物之间的关系。如下图(摘自网络)：

注：#表示protected，+表示Public，-表示private

2.对象图：描述一组对象之间的关系，是具有具体属性值和行为的一个具体事物，其是类图中所建事物实例的静态快照，其与类图的主要区别是一个是抽象的,而对象图是具体的。如下图(摘自网络)：

3.用例图：描述一组用例、参与者以及它们之间的关系，其展示的是该系统在它的外面环境中所提供的外部可见服务。如下图(摘自网络)：

4.交互图：包括序列图（顺序图）和协作图，两者对应，顺序图是强调消息时间顺序，有对象生命线和控制焦点。协作图是强调接收和发送消息的对象的结构组织，有路径和顺序号。如下图(摘自网络)：

序列图：

协作图：

5.状态图：展示了一个状态机，由状态、转换、事件和活动组成。强调事件行为的顺序。如下图(摘自网络)：

6.活动图：是一种特殊的状态图，实现一个活动到另一个活动的流程。如下图(摘自网络)：

7.构件图和部署图：构件图展示一组构件之间的组织和依赖关系，并以全局的模型展示出来。部署图是构件的配置及描述系统如何在硬件上部署。如下图(摘自网络)：

# UML中的几种关系详细解析

在UML类图中，常见的有以下几种关系: 泛化（Generalization）,  实现（Realization），关联（Association)，聚合（Aggregation），组合(Composition)，依赖(Dependency)

 　　1. 泛化（Generalization）

　　【泛化关系】：是一种继承关系，表示一般与特殊的关系，它指定了子类如何特化父类的所有特征和行为。例如：老虎是动物的一种，即有老虎的特性也有动物的共性。

　　【箭头指向】：带三角箭头的实线，箭头指向父类



　　2. 实现（Realization）

　　【实现关系】：是一种类与接口的关系，表示类是接口所有特征和行为的实现.

　　【箭头指向】：带三角箭头的虚线，箭头指向接口



　　3. 关联（Association)

　　【关联关系】：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。

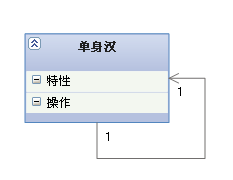
　　【代码体现】：成员变量

　　【箭头及指向】：带普通箭头的实心线，指向被拥有者



　　上图中，老师与学生是双向关联，老师有多名学生，学生也可能有多名老师。但学生与某课程间的关系为单向关联，一名学生可能要上多门课程，课程是个抽象的东西他不拥有学生。

　　下图为自身关联：



　　4. 聚合（Aggregation）

　　【聚合关系】：是整体与部分的关系，且部分可以离开整体而单独存在。如车和轮胎是整体和部分的关系，轮胎离开车仍然可以存在。

　　聚合关系是关联关系的一种，是强的关联关系；关联和聚合在语法上无法区分，必须考察具体的逻辑关系。

　　【代码体现】：成员变量

　　【箭头及指向】：带空心菱形的实心线，菱形指向整体



　　5. 组合(Composition)

　　【组合关系】：是整体与部分的关系，但部分不能离开整体而单独存在。如公司和部门是整体和部分的关系，没有公司就不存在部门。

       组合关系是关联关系的一种，是比聚合关系还要强的关系，它要求普通的聚合关系中代表整体的对象负责代表部分的对象的生命周期。

【代码体现】：成员变量

【箭头及指向】：带实心菱形的实线，菱形指向整体



　　6. 依赖(Dependency)

　　【依赖关系】：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.

　　【代码表现】：局部变量、方法的参数或者对静态方法的调用

　　【箭头及指向】：带箭头的虚线，指向被使用者



　　各种关系的强弱顺序：

　　泛化 = 实现 > 组合 > 聚合 > 关联 > 依赖

　　下面这张UML图，比较形象地展示了各种类图关系：



# UML几种图介绍

用例图

　　用例图描述了系统提供的一个功能单元。用例图的主要目的是帮助开发团队以一种可视化的方式理解系统的功能需求，包括基于基本流程的"角色"（actors，也就是与系统交互的其他实体）关系，以及系统内用例之间的关系。用例图一般表示出用例的组织关系 —— 要么是整个系统的全部用例，要么是完成具有功能（例如，所有安全管理相关的用例）的一组用例。要在用例图上显示某个用例，可绘制一个椭圆，然后将用例的名称放在椭圆的中心或椭圆下面的中间位置。要在用例图上绘制一个角色（表示一个系统用户），可绘制一个人形符号。角色和用例之间的关系使用简单的线段来描述，如图1所示。

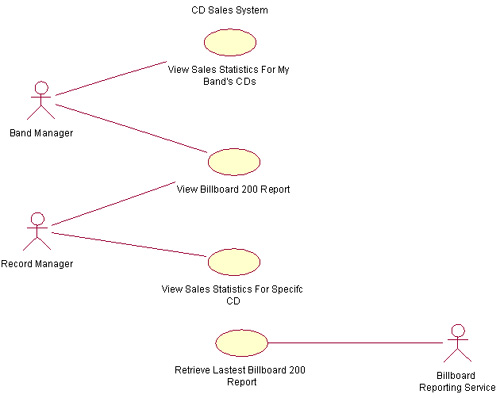


图1：示例用例图

　　图字（从上到下）：CD销售系统；查看乐队CD的销售统计；乐队经理；查看Billboard 200排行榜报告；唱片经理；查看特定CD的销售统计；检索最新的Billboard 200排行榜报告；排行榜报告服务

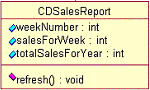
　　用例图通常用于表达系统或者系统范畴的高级功能。如图1所示，可以很容易看出该系统所提供的功能。这个系统允许乐队经理查看乐队CD的销售统计报告以及Billboard 200排行榜报告。它也允许唱片经理查看特定CD的销售统计报告和这些CD在Billboard 200排行榜的报告。这个图还告诉我们，系统将通过一个名为"排行榜报告服务"的外部系统提供Billboard排行榜报告。

　　此外，在用例图中，没有列出的用例表明了该系统不能完成的功能。例如，它不能提供给乐队经理收听Billboard 200上不同专辑中的歌曲的途径 —— 也就是说，系统没有引用一个叫做"收听Billboard 200上的歌曲"的用例。这种缺少不是一件小事。在用例图中提供清楚的、简要的用例描述，项目赞助商就很容易看出系统是否提供了必须的功能。

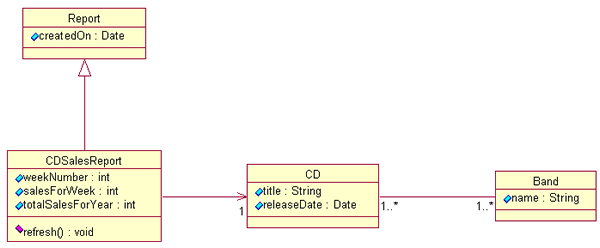
类图

　　类图表示不同的实体（人、事物和数据）如何彼此相关；换句话说，它显示了系统的静态结构。类图可用于表示逻辑类，逻辑类通常就是业务人员所谈及的事物种类 —— 摇滚乐队、CD、广播剧；或者贷款、住房抵押、汽车信贷以及利率。类图还可用于表示实现类，实现类就是程序员处理的实体。实现类图或许会与逻辑类图显示一些相同的类。然而，实现类图不会使用相同的属性来描述，因为它很可能具有对诸如Vector和HashMap这种事物的引用。

　　类在类图上使用包含三个部分的矩形来描述，如图2所示。最上面的部分显示类的名称，中间部分包含类的属性，最下面的部分包含类的操作（或者说"方法"）。

  
图2：类图中的示例类对象

　　根据我的经验，几乎每个开发人员都知道这个类图是什么，但是我发现大多数程序员都不能正确地描述类的关系。对于像图3这样的类图，您应该使用带有顶点指向父类的箭头的线段来绘制继承关系1，并且箭头应该是一个完全的三角形。如果两个类都彼此知道对方，则应该使用实线来表示关联关系；如果只有其中一个类知道该关联关系，则使用开箭头表示。

  
图3：一个完整的类图，包括了图2所示的类对象

　　在图3中，我们同时看到了继承关系和两个关联关系。CDSalesReport类继承自Report类。一个CDSalesReport类与一个CD类关联，但是CD类并不知道关于CDSalesReport类的任何信息。CD类和Band类都彼此知道对方，两个类彼此都可以与一个或者多个对方类相关联。

　　一个类图可以整合其他许多概念，这将在本系列文章的后续文章中介绍。

序列图

　　序列图显示具体用例（或者是用例的一部分）的详细流程。它几乎是自描述的，并且显示了流程中不同对象之间的调用关系，同时还可以很详细地显示对不同对象的不同调用。

　　序列图有两个维度：垂直维度以发生的时间顺序显示消息/调用的序列；水平维度显示消息被发送到的对象实例。

　　序列图的绘制非常简单。横跨图的顶部，每个框（参见图4）表示每个类的实例（对象）。在框中，类实例名称和类名称之间用空格/冒号/空格来分隔，例如，myReportGenerator : ReportGenerator。如果某个类实例向另一个类实例发送一条消息，则绘制一条具有指向接收类实例的开箭头的连线，并把消息/方法的名称放在连线上面。对于某些特别重要的消息，您可以绘制一条具有指向发起类实例的开箭头的虚线，将返回值标注在虚线上。就我而言，我总喜欢绘制出包括返回值的虚线，这些额外的信息可以使得序列图更易于阅读。

　　阅读序列图也非常简单。从左上角启动序列的"驱动"类实例开始，然后顺着每条消息往下阅读。记住：虽然图4所示的例子序列图显示了每条被发送消息的返回消息，但这只是可选的。

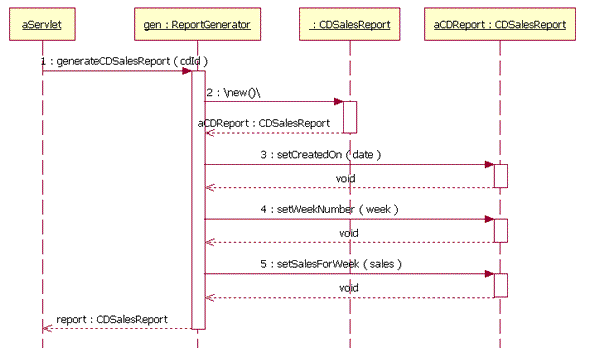
[](http://www.ibm.com/developerworks/rational/library/769b.html)

图4：一个示例序列图

　　通过阅读图4中的示例序列图，您可以明白如何创建一个CD销售报告（CD Sales Report）。其中的aServlet对象表示驱动类实例。aServlet向名为gen的ReportGenerator类实例发送一条消息。该消息被标为generateCDSalesReport，表示ReportGenerator对象实现了这个消息处理程序。进一步理解可发现，generateCDSalesReport消息标签在括号中包括了一个cdId，表明aServlet随该消息传递一个名为cdId的参数。当gen实例接收到一条generateCDSalesReport消息时，它会接着调用CDSalesReport类，并返回一个aCDReport的实例。然后gen实例对返回的aCDReport实例进行调用，在每次消息调用时向它传递参数。在该序列的结尾，gen实例向它的调用者aServlet返回一个aCDReport。

　　请注意：图4中的序列图相对于典型的序列图来说太详细了。然而，我认为它才是足够易于理解的，并且它显示了如何表示嵌套的调用。对于初级开发人员来说，有时把一个序列分解到这种详细程度是很有必要的，这有助于他们理解相关的内容。

状态图

　　状态图表示某个类所处的不同状态和该类的状态转换信息。有人可能会争论说每个类都有状态，但不是每个类都应该有一个状态图。只对"感兴趣的"状态的类（也就是说，在系统活动期间具有三个或更多潜在状态的类）才进行状态图描述。

　　如图5所示，状态图的符号集包括5个基本元素：初始起点，它使用实心圆来绘制；状态之间的转换，它使用具有开箭头的线段来绘制；状态，它使用圆角矩形来绘制；判断点，它使用空心圆来绘制；以及一个或者多个终止点，它们使用内部包含实心圆的圆来绘制。要绘制状态图，首先绘制起点和一条指向该类的初始状态的转换线段。状态本身可以在图上的任意位置绘制，然后只需使用状态转换线条将它们连接起来。

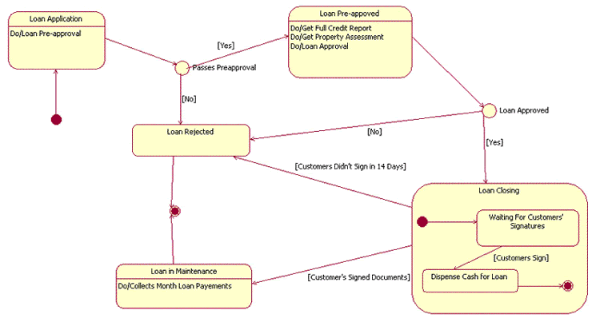
[](http://www.ibm.com/developerworks/rational/library/769c.html)  
 图5：显示类通过某个功能系统的各种状态的状态图

　　图5中的状态图显示了它们可以表达的一些潜在信息。例如，从中可以看出贷款处理系统最初处于Loan Application状态。当批准前（pre-approval）过程完成时，根据该过程的结果，或者转到Loan Pre-approved状态，或者转到Loan Rejected状态。这个判断（它是在转换过程期间做出的）使用一个判断点来表示 —— 即转换线条间的空心圆。通过该状态图可知，如果没有经过Loan Closing状态，贷款不可能从Loan Pre-Approved状态进入Loan in Maintenance状态。而且，所有贷款都将结束于Loan Rejected或者Loan in Maintenance状态。

活动图

　　活动图表示在处理某个活动时，两个或者更多类对象之间的[过程控制](https://www.baidu.com/s?wd=%E8%BF%87%E7%A8%8B%E6%8E%A7%E5%88%B6&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)流。活动图可用于在业务单元的级别上对更高级别的业务过程进行建模，或者对低级别的内部类操作进行建模。根据我的经验，活动图最适合用于对较高级别的过程建模，比如公司当前在如何运作业务，或者业务如何运作等。这是因为与序列图相比，活动图在表示上"不够技术性的"，但有业务头脑的人们往往能够更快速地理解它们。

　　活动图的符号集与状态图中使用的符号集类似。像状态图一样，活动图也从一个连接到初始活动的实心圆开始。活动是通过一个圆角矩形（活动的名称包含在其内）来表示的。活动可以通过转换线段连接到其他活动，或者连接到判断点，这些判断点连接到由判断点的条件所保护的不同活动。结束过程的活动连接到一个终止点（就像在状态图中一样）。作为一种选择，活动可以分组为泳道（swimlane），泳道用于表示实际执行活动的对象，如图6所示。

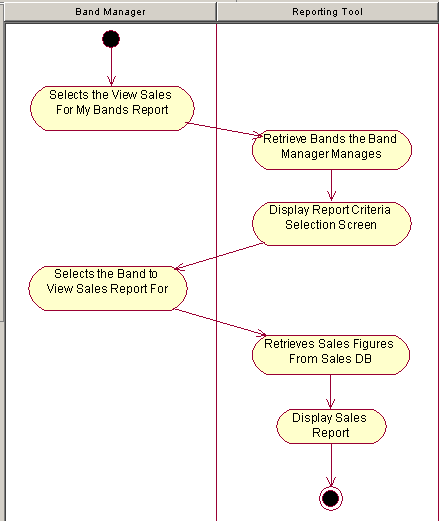


图6：活动图，具有两个泳道，表示两个对象的活动控制：乐队经理，以及报告工具

　　图字（沿箭头方向）：乐队经理；报告工具；选择"查看乐队的销售报告"；检索该乐队经理所管理的乐队；显示报告条件选择屏幕；选择要查看其销售报告的乐队；从销售数据库检索销售数据；显示销售报告。

　　该活动图中有两个泳道，因为有两个对象控制着各自的活动：乐队经理和报告工具。整个过程首先从乐队经理选择查看他的乐队销售报告开始。然后报告工具检索并显示他管理的所有乐队，并要求他从中选择一个乐队。在乐队经理选择一个乐队之后，报告工具就检索销售信息并显示销售报告。该活动图表明，显示报告是整个过程中的最后一步。

组件图

　　组件图提供系统的物理视图。它的用途是显示系统中的软件对其他软件组件（例如，库函数）的依赖关系。组件图可以在一个非常高的层次上显示，从而仅显示粗粒度的组件，也可以在组件包层次2上显示。

　　组件图的建模最适合通过例子来描述。图7显示了4个组件：Reporting Tool、Billboard Service、Servlet 2.2 API和JDBC API。从Reporting Tool组件指向Billboard Service、Servlet 2.2 API和JDBC API组件的带箭头的线段，表示Reporting Tool依赖于那三个组件。

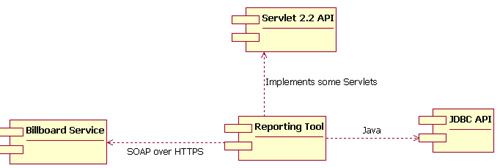


图7：组件图显示了系统中各种软件组件的依赖关系

部署图

　　部署图表示该软件系统如何部署到硬件环境中。它的用途是显示该系统不同的组件将在何处物理地运行，以及它们将如何彼此通信。因为部署图是对物理运行情况进行建模，系统的生产人员就可以很好地利用这种图。

　　部署图中的符号包括组件图中所使用的符号元素，另外还增加了几个符号，包括节点的概念。一个节点可以代表一台物理机器，或代表一个虚拟机器节点（例如，一个大型机节点）。要对节点进行建模，只需绘制一个三维立方体，节点的名称位于立方体的顶部。所使用的命名约定与序列图中相同：[实例名称] : [实例类型]（例如，"w3reporting.myco.com : Application Server"）。

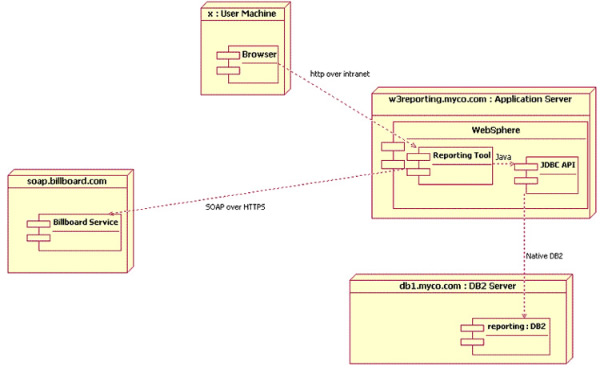
[](http://www.ibm.com/developerworks/rational/library/769c.html)

　　图8：部署图

　　图字：由于Reporting Tool组件绘制在IBM WebSphere内部，后者又绘制在节点w3.reporting.myco.com内部，因而我们知道，用户将通过运行在本地机器上的浏览器来访问Reporting Tool，浏览器通过公司intranet上的HTTP协议与Reporting Tool建立连接。

　　图8中的部署图表明，用户使用运行在本地机器上的浏览器访问Reporting Tool，并通过公司intranet上的HTTP协议连接到Reporting Tool组件。这个工具实际运行在名为w3reporting.myco.com的Application Server上。这个图还表明Reporting Tool组件绘制在IBM WebSphere内部，后者又绘制在w3.reporting.myco.com节点内部。Reporting Tool使用Java语言通过IBM DB2数据库的JDBC接口连接到它的报告数据库上，然后该接口又使用本地DB2通信方式，与运行在名为db1.myco.com的服务器上实际的DB2数据库通信。除了与报告数据库通信外，Report Tool组件还通过HTTPS上的SOAP与Billboard Service进行通信。

结束语

　　尽管本文仅提供了对统一建模语言UML的简要介绍，但还是鼓励大家把从这里学到的基本信息应用到自己的项目中，同时更深入地钻研UML。已经有多种软件工具可以帮助您把UML图集成到软件开发过程中，不过即使没有自动化的工具，您也可以使用白板上的标记或者纸和笔来手工绘制UML图，仍然会[获益匪浅](https://www.baidu.com/s?wd=%E8%8E%B7%E7%9B%8A%E5%8C%AA%E6%B5%85&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)。

# UML之用例图解析

用例图主要用来描述“用户、需求、系统功能单元”之间的关系。它展示了一个外部用户能够观察到的系统功能模型图。

　　【用途】：帮助开发团队以一种可视化的方式理解系统的功能需求。

　　用例图所包含的元素如下：

　　1. 参与者(Actor)

　　表示与您的应用程序或系统进行交互的用户、组织或外部系统。用一个小人表示。



　　2. 用例(Use Case)

 　　用例就是外部可见的系统功能，对系统提供的服务进行描述。用椭圆表示。



　　3. 子系统(Subsystem)

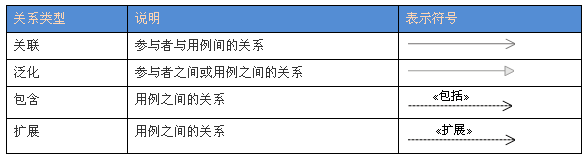
　　用来展示系统的一部分功能，这部分功能联系紧密。



　　4. 关系

　　用例图中涉及的关系有：关联、泛化、包含、扩展。

　　如下表所示：



　　a. 关联(Association)

　　表示参与者与用例之间的通信，任何一方都可发送或接受消息。

　　【箭头指向】：指向消息接收方



　　b. 泛化(Inheritance)

　　就是通常理解的继承关系，子用例和父用例相似，但表现出更特别的行为；子用例将继承父用例的所有结构、行为和关系。子用例可以使用父用例的一段行为，也可以重载它。父用例通常是抽象的。

　　【箭头指向】：指向父用例



　　c. 包含(Include)

　　包含关系用来把一个较复杂用例所表示的功能分解成较小的步骤。

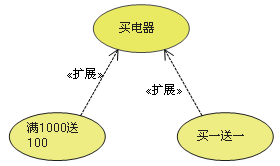
　　【箭头指向】：指向分解出来的功能用例z



　　d. 扩展(Extend)

　　扩展关系是指用例功能的延伸，相当于为基础用例提供一个附加功能。

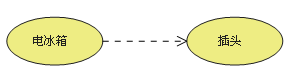
　　【箭头指向】：指向基础用例



　　e. 依赖(Dependency)

　　以上4种关系，是UML定义的标准关系。但VS2010的用例模型图中，添加了依赖关系，用带箭头的虚线表示，表示源用例依赖于目标用例。

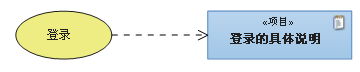
　　【箭头指向】：指向被依赖项



　　5. 项目(Artifact)

　　用例图虽然是用来帮助人们形象地理解功能需求，但却没多少人能够通看懂它。很多时候跟用户交流甚至用Excel都比用例图强，VS2010中引入了“项目”这样一个元素，以便让开发人员能够在用例图中链接一个普通文档。

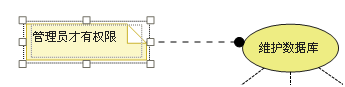
　　用依赖关系把某个用例依赖到项目上：



　　然后把项目-》属性 的Hyperlink设置到你的文档上；

　　这样当你在用例图上双击项目时，就会打开相关联的文档。

　　6. 注释(Comment)



　　包含(include)、扩展(extend)、泛化(Inheritance) 的区别：

　　条件性：泛化中的子用例和include中的被包含的用例会无条件发生，而extend中的延伸用例的发生是有条件的；

　　直接性：泛化中的子用例和extend中的延伸用例为参与者提供直接服务，而include中被包含的用例为参与者提供间接服务。

　　对extend而言，延伸用例并不包含基础用例的内容，基础用例也不包含延伸用例的内容。

　　对Inheritance而言，子用例包含基础用例的所有内容及其和其他用例或参与者之间的关系；

　　一个用例图示例：



　　牢骚：

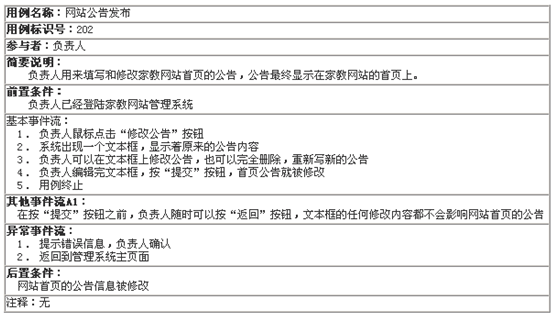
　　感觉用例图还不成熟，并不能很好地表达系统的需求， 没有UML背景的用户几乎不知道画的是些什么。

　　其次，包含关系、扩展关系的箭头符号竟然是同样的箭头，仅靠上方写个文字来加以区别，翻译成其他语言的话，几乎就不知道代表什么意思。扩展关系的箭头朝向也很难理解，为何要指向基用例，而不指向扩展用例。

　　VS2010添加的“项目”元素，是个很好的创新，能够在用例图中关联word, excel这些文档。但为什么不把这些功能直接集成到用例里面，双击用例就弹出一份文档岂不更容易理解，非要画蛇添足地加一个元件，仅仅为了提供个链接功能。

　　用例描述表：

　　鉴于用列图并不能清楚地表达功能需求，开发中大家通常用描述表来补充某些不易表达的用例，下图的表给大家提供一个参考：



# UML之序列图解析

序列图主要用于展示对象之间交互的顺序。

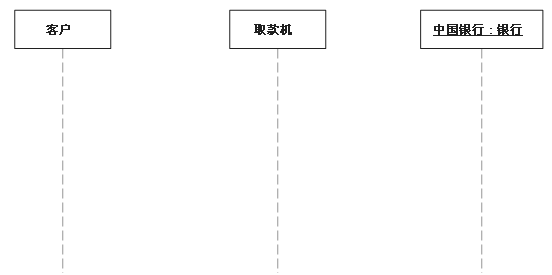
　　序列图将交互关系表示为一个二维图。纵向是时间轴，时间沿竖线向下延伸。横向轴代表了在协作中各独立对象的类元角色。类元角色用生命线表示。当对象存在时，角色用一条虚线表示，当对象的过程处于激活状态时，生命线是一个双道线。

　　消息用从一个对象的生命线到另一个对象生命线的箭头表示。箭头以时间顺序在图中从上到下排列。

　　序列图中涉及的元素：

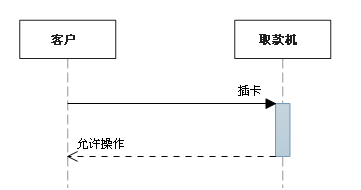
　　1. 生命线：

　　生命线名称可带下划线。当使用下划线时，意味着序列图中的生命线代表一个类的特定实例。



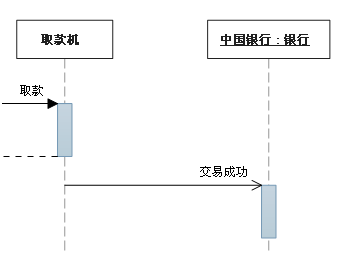
　　2. 同步消息

　　发送人在它继续之前，将等待同步消息响应。

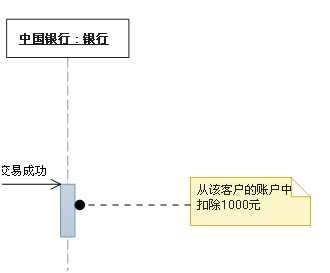


　　3. 异步消息

　　在发送方继续之前，无需等待响应的消息。

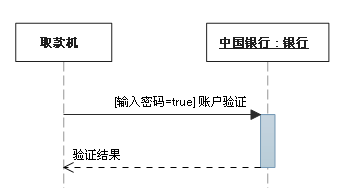


　　4. 注释



　　5. 约束

　　约束的符号很简单；格式是: [Boolean Test]



　　6. 组合片段

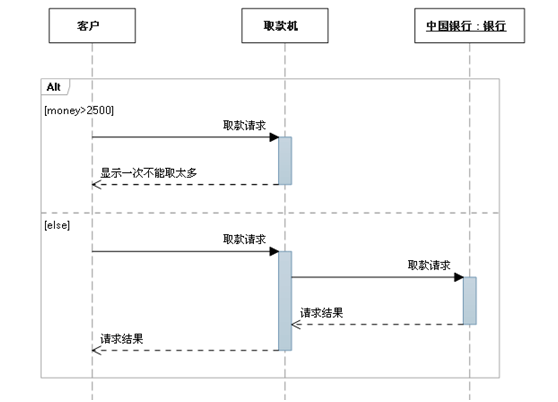
　　组合片段用来解决交互执行的条件及方式。它允许在序列图中直接表示逻辑组件，用于通过指定条件或子进程的应用区域，为任何生命线的任何部分定义特殊条件和子进程。

　　常用的组合片段有：

　　抉择（Alt）

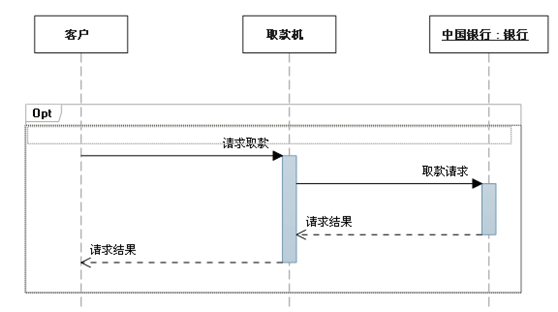
　　抉择用来指明在两个或更多的消息序列之间的互斥的选择，相当于经典的if..else..。

　　抉择在任何场合下只发生一个序列。 可以在每个片段中设置一个临界来指示该片段可以运行的条件。else 的临界指示其他任何临界都不为 True 时应运行的片段。如果所有临界都为 False 并且没有 else，则不执行任何片段。



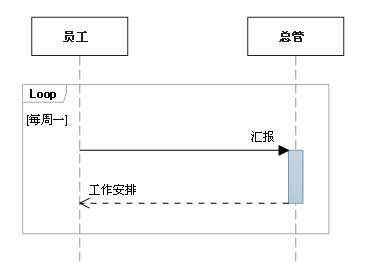
　　选项（Opt）

　　包含一个可能发生或不发生的序列

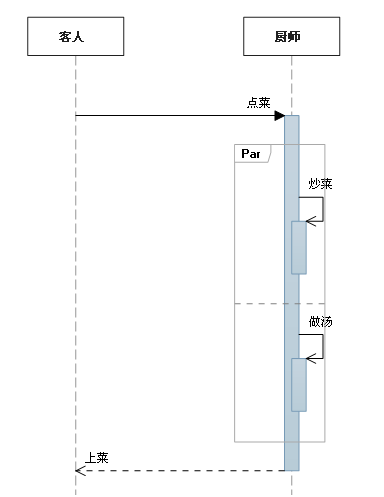


　　循环（Loop）

　　片段重复一定次数。 可以在临界中指示片段重复的条件。



　　并行（Par）



　　下表列出了常用的组合片段：

|  |  |  |
| --- | --- | --- |
| 片段类型 | 名称 | 说明 |
| Opt | 选项 | 包含一个可能发生或可能不发生的序列。 可以在临界中指定序列发生的条件。 |
| Alt | 抉择 | 包含一个片段列表，这些片段包含备选消息序列。 在任何场合下只发生一个序列。  可以在每个片段中设置一个临界来指示该片段可以运行的条件。 else 的临界指示其他任何临界都不为 True 时应运行的片段。 如果所有临界都为 False 并且没有 else，则不执行任何片段。 |
| Loop | 循环 | 片段重复一定次数。 可以在临界中指示片段重复的条件。  Loop 组合片段具有“Min”和“Max”属性，它们指示片段可以重复的最小和最大次数。 默认值是无限制。 |
| Break | 中断 | 如果执行此片段，则放弃序列的其余部分。 可以使用临界来指示发生中断的条件。 |
| Par | 并行 | 并行处理。 片段中的事件可以交错。 |
| Critical | 关键 | 用在 Par 或 Seq 片段中。 指示此片段中的消息不得与其他消息交错。 |
| Seq | 弱顺序 | 有两个或更多操作数片段。 涉及同一生命线的消息必须以片段的顺序发生。 如果消息涉及的生命线不同，来自不同片段的消息可能会并行交错。 |
| Strict | 强顺序 | 有两个或更多操作数片段。 这些片段必须按给定顺序发生。 |

　　有关如何解释序列的片段

　　默认情况下，序列图表明可能发生的一系列消息。 在运行的系统中，可能会出现您未选择显示在关系图上的其他消息。

　　以下片段类型可用于更改此释义：

|  |  |  |
| --- | --- | --- |
| 片段类型 | 名称 | 说明 |
| Consider | 考虑 | 指定此片段描述的消息列表。 其他消息可发生在运行的系统中，但对此描述来说意义不大。  在“Messages”属性中键入该列表。 |
| Ignore | 忽略 | 此片段未描述的消息列表。 这些消息可发生在运行的系统中，但对此描述来说意义不大。  在“Messages”属性中键入该列表。 |
| Assert | 断言 | 操作数片段指定唯一有效的序列。 通常用在 Consider 或 Ignore 片段中。 |
| Neg | 否定 | 此片段中显示的序列不得发生。 通常用在 Consider 或 Ignore 片段中。 |

# UML之活动图解析

一、活动图的组成元素 Activity Diagram Element

1、活动状态图（Activity）

2、动作状态（Actions)

3、动作状态约束（Action Constraints）

4、动作流（Control Flow）

5、开始节点（Initial Node）

6、终止节点（Final Node）

7、对象（Objects）

8、数据存储对象（DataStore）

9、对象流（Object Flows）

10、分支与合并（Decision and Merge Nodes）

11、分叉与汇合（Fork and Join Nodes）

12、异常处理（Exception Handler）

13、活动中断区域（Interruptible Activity Region）

14、泳道（Partition）

二、活动图案例分析

三、总结

活动图是UML用于对系统的动态行为建模的另一种常用工具，它描述活动的顺序，展现从一个活动到另一个活动的控制流。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。

## 一、活动图的组成元素 Activity Diagram Element

### 1、活动状态图（Activity）

活动状态用于表达状态机中的非原子的运行，其特点如下：

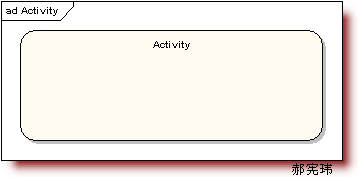
(1)、活动状态可以分解成其他子活动或者动作状态。

(2)、活动状态的内部活动可以用另一个活动图来表示。

(3)、和动作状态不同，活动状态可以有入口动作和出口动作，也可以有内部转移。

(4)、动作状态是活动状态的一个特例，如果某个活动状态只包括一个动作，那么它就是一个动作状态。

UML中活动状态和动作状态的图标相同，但是活动状态可以在图标中给出入口动作和出口动作等信息。



### 2、动作状态（Actions）

动作状态是指原子的，不可中断的动作，并在此动作完成后通过完成转换转向另一个状态。动作状态有如下特点：

(1)、动作状态是原子的，它是构造活动图的最小单位。

(2)、动作状态是不可中断的。

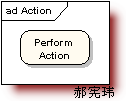
(3)、动作状态是瞬时的行为。

(4)、动作状态可以有入转换，入转换既可以是动作流，也可以是对象流。动作状态至少有一条出转换，这条转换以内部的完成为起点，与外部事件无关。

(5)、动作状态与状态图中的状态不同，它不能有入口动作和出口动作，更不能有内部转移。

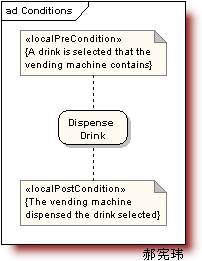
(6)、在一张活动图中，动作状态允许多处出现。

UML中的动作状态图用平滑的圆角矩形表示，如下：



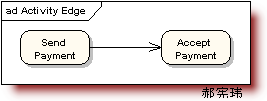
### 3、动作状态约束（Action Constraints）

动作状态约束：用来约束动作状态。如下图展示了动作状态的前置条件和后置条件

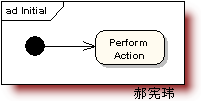


### 4、动作流（Control Flow）

动作之间的转换称之为动作流，活动图的转换用带箭头的直线表示，箭头的方向指向转入的方向。

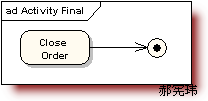


5、开始节点（Initial Node）  
开始节点：表示成实心黑色圆点

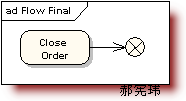


6、终止节点（Final Node）  
分为活动终止节点（activity final nodes）和流程终止节点（flow final nodes）。

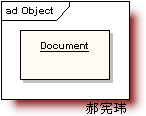
活动终止节点表示整个活动的结束



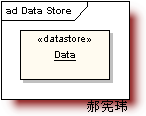
而流程终止节点表示是子流程的结束。



7、对象（Objects）



8、数据存储对象（DataStore）



使用关键字«datastore»

9、对象流（Object Flows）

对象流是动作状态或者活动状态与对象之间的依赖关系，表示动作使用对象或动作对对象的影响。用活动图描述某个对象时，可以把涉及到的对象放置在活动图中并用一个依赖将其连接到进行创建、修改和撤销的动作状态或者活动状态上，对象的这种使用方法就构成了对象流。

对象流中的对象有以下特点：

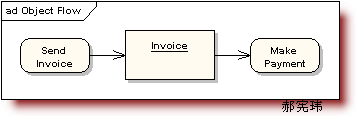
(1)、一个对象可以由多个动作操作。

(2)、一个动作输出的对象可以作为另一个动作输入的对象。

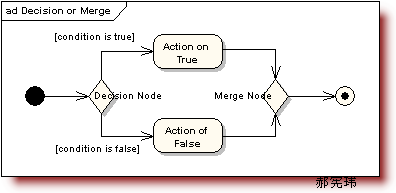
(3)、在活动图中，同一个对象可以多次出现，它的每一次出现表面该对象正处于对象生存期的不同时间点。

对象流用带有箭头的虚线表示。如果箭头是从动作状态出发指向对象，则表示动作对对象施加了一定的影响。施加的影响包括创建、修改和撤销等。如果箭头从对象指向动作状态，则表示该动作使用对象流所指向的对象。

状态图中的对象用矩形表示，矩形内是该对象的名称，名称下的方括号表明对象此时的状态。

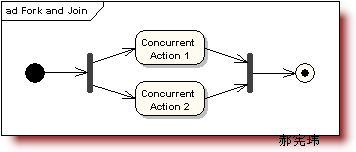


10、分支与合并（Decision and Merge Nodes）  
分支与合并用菱形表示



11、分叉与汇合（Fork and Join Nodes）

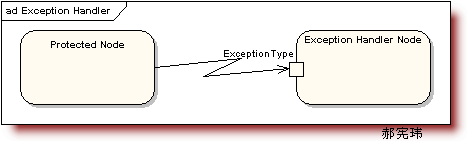
分为水平风向和垂直方向。



对象在运行时可能会存在两个或多个并发运行的控制流，为了对并发的控制流建模，UML中引入了分叉与汇合的概念。分叉用于将动作流分为两个或多个并发运行的分支，而汇合则用于同步这些并发分支，以达到共同完成一项事务的目的。

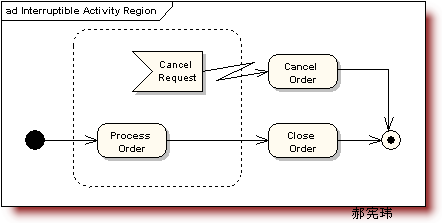
12、异常处理（Exception Handler）

当受保护的活动发生异常时，触发异常处理节点。



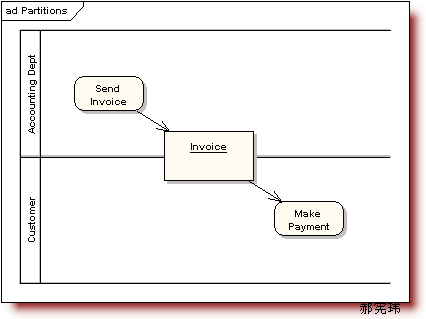
13、活动中断区域（Interruptible Activity Region）

活动中断区域围绕一些可被中断的动作状态图。比如下图，正常情况下【Process Order】顺序流转到【Close Order】，订单处理流程完毕；但在【Process Order】过称中，会发送【Cancel Order】请求，这时会流转到【Cancel Order】，从而订单处理流程结束



14、泳道（Partition）  
泳道将活动图中的活动划分为若干组，并把每一组指定给负责这组活动的业务组织，即对象。在活动图中，泳道区分了负责活动的对象，它明确地表示了哪些活动是由哪些对象进行的。在包含泳道的活动图中，每个活动只能明确地属于一个泳道。

泳道是用垂直实线绘出，垂直线分隔的区域就是泳道。在泳道的上方可以给出泳道的名字或对象的名字，该对象负责泳道内的全部活动。泳道没有顺序，不同泳道中的活动既可以顺序进行也可以并发进行，动作流和对象流允许穿越分隔线。



## 二、活动图案例分析

## http://images.cnblogs.com/cnblogs_com/ywqu/Activity%20Diagram/NActivityDiagram.jpg

1、  泳道分为：会员泳道和系统泳道。会员选择商品并加入购物车，系统完成订单生成及其支付完毕。

2、  开始节点：会员添加商品到购物车，点击【订单确认】，开始交于系统处理订单流程

3、  结束节点：商品发送完毕和付款成功，订单处理流程结束

4、  活动状态：产生订单、Check Credit Cart核对信用卡、Check Stock 核对库存量、Deliver Goods 发送商品、Process Credit Cart付款

5、  分叉与汇合：【产生订单】份叉为检查库存量和会员支付金额是否足够，如果不足，取消订单，如过库存量和支付金额足够，发送商品和付款，最后汇合为订单完成。

## 三、总结

活动图描述的是对象活动的顺序关系所遵循的规则，它着重表现的是系统的行为，而非系统的处理过程。活动图能够表示并发活动的情形，活动图是面向对象的。

# UML之关系图解析

一、简介

# 二、类的构成

# 三、类之间的关系（Relationship）

# 1、单向关联

# 2、双向关联

# 3、自身关联

# 4、多维关联(N-ary Association)

# 5、泛化(Generalization)

# 6、依赖(Dependency)

# 7、聚合(Aggregation)

# 8、组合（Composite）

# 四、总结

## 一、简介

类是对象的集合，展示了对象的结构以及与系统的交互行为。类主要有属性（Attribute）和方法（Method）构成,属性代表对象的状态，如果属性被保存到数据库，此称之为“持久化”；方法代表对象的操作行为，类具有继承关系，可以继承于父类，也可以与其他的Class进行交互。

       类图展示了系统的逻辑结构，类和接口的关系。

二、类的构成

类主要有属性和方法构成。比如商品属性有：名称、价格、高度、宽度等；商品的方法有：计算税率，获得商品的评价等等。如下图



## 三、类之间的关系（Relationship）

关联(Association)

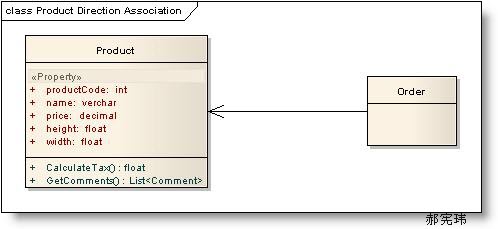
两个相对独立的对象，当一个对象的实例与另外一个对象的特定实例存在固定关系时，这两个对象之间就存在关联关系。

### 1、单向关联

A1->A2: 表示A1认识A2，A1知道A2的存在，A1可以调用A2中的方法和属性

场景：订单和商品，订单中包括商品，但是商品并不了解订单的存在。

类与类之间的单向关联图:



C#代码:

Public class Order

{

       Public List<Product> order;

Public void AddOrder(Product product )

       {

              order.Add(product);

}

}

Public Class Product

{

}

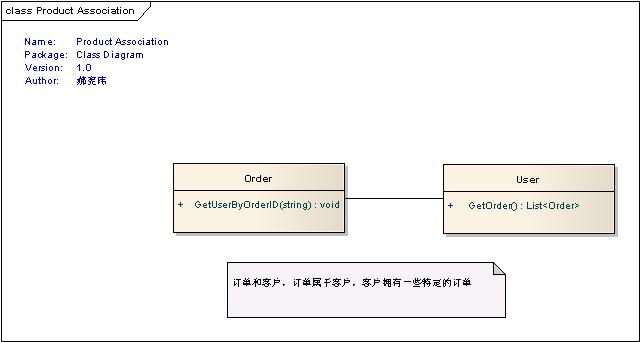
代码表现为：Order(A1)中有Product(A2)的变量或者引用

### 2、双向关联

B1-B2: 表示B1认识B2，B1知道B2的存在，B1可以调用B2中的方法和属性；同样B2也知道B1的存在，B2也可以调用B1的方法和属性。

场景：订单和客户，订单属于客户，客户拥有一些特定的订单

类与类之间的双向关联图



C#代码

Public class User

{

       Public List<Order> GetOrder()

       {

}      return new List<Order>();

}

Public Class Order

{

       Public User GetUserByOrderID(string OrderId )

       {

              Return new User();

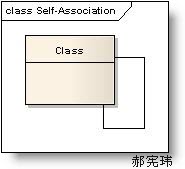
}

}

### 3、自身关联

同一个类对象之间的关联

类与类之间自身关联图

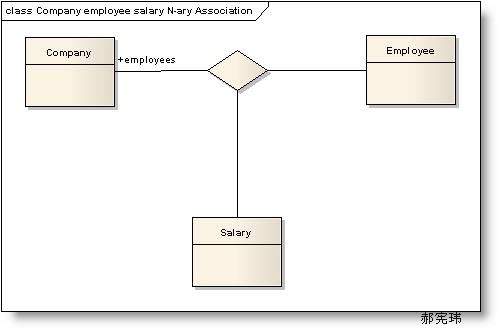


### 4、多维关联(N-ary Association)

多个对象之间存在关联

场景：公司雇用员工，同时公司需要支付工资给员工

类与类之间的多维关联图：

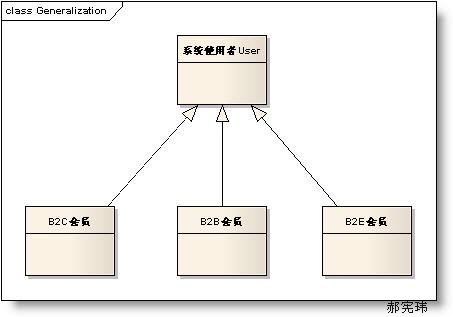


### 5、泛化(Generalization)

类与类的继承关系，类与接口的实现关系。

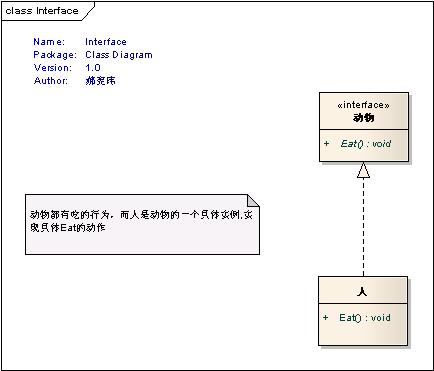
场景：父与子、动物与人、植物与树、系统使用者与B2C会员和B2E会员的关系

类与类之间的泛化图：



系统的使用者包括：B2C会员、B2B会员和B2E会员。

接口的实现，动物都有吃的行为，而人是动物的一个具体实例,实现具体Eat的动作



### 6、依赖(Dependency)

类A要完成某个功能必须引用类B，则A与B存在依赖关系，依赖关系是弱的关联关系。C#不建议双相依赖，也就是相互引用

场景：本来人与电脑没有关系的，但由于偶然的机会，人需要用电脑写程序，这时候人就依赖于电脑。

类与类的依赖关系图

在程序中一般为 using 引用。

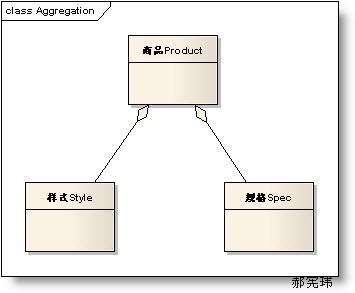
### http://images.cnblogs.com/cnblogs_com/ywqu/Class%20Diagram/NameDenpendecy.jpg

### 7、聚合(Aggregation)

当对象A被加入到对象B中，成为对象B的组成部分时，对象B和对象A之间为聚合关系。聚合是关联关系的一种，是较强的关联关系，强调的是整体与部分之间的关系。

场景：商品和他的规格、样式就是聚合关系。

类与类的聚合关系图

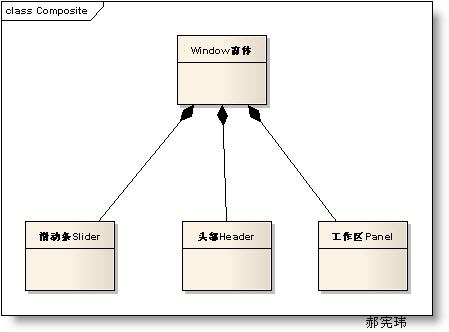


### 8、组合（Composite）

       对象A包含对象B，对象B离开对象A没有实际意义。是一种更强的关联关系。人包含手，手离开人的躯体就失去了它应有的作用。

场景： Window窗体由滑动条slider、头部Header 和工作区Panel组合而成。

类与类的组合关系图



## 四、总结

## 本文针对类之间常用的关系进行了简单的描述，主要有：关联关系、泛化、依赖、聚合和组合。

# UML之状态图解析

状态图目录：

一、状态图简介（Brief introduction）

二、状态图元素（State Diagram Elements）

1、状态（States）

2、转移（Transitions）

3、动作（State Actions）

4、自身转移（Self-Transitions）

5、组合状态（Compound States）

6、进入节点（Entry Point）

7、退出节点（Exit Point）

8、历史状态（History States）

9、并发区域（Concurrent Regions）

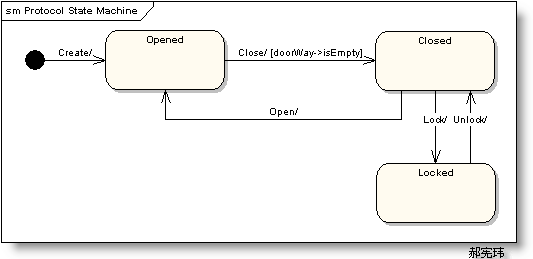
三、状态图案例分析（State Diagram Example Analysis）

      四、总结（Summary）

#### 一、状态图简介（Brief introduction）

## 

状态图（Statechart Diagram）主要用于描述一个对象在其生存期间的动态行为，表现为一个对象所经历的状态序列，引起状态转移的事件（Event），以及因状态转移而伴随的动作（Action）。一般可以用状态机对一个对象的生命周期建模，状态图用于显示状态机（State Machine Diagram），重点在与描述状态图的控制流。  
如下图例子，状态机描述了门对象的生存期间的状态序列，引起转移的事件，以及因状态转移而伴随的动作（Action）.



状态有Opened、Closed、Locked。

事件有 Open、Close、Lock和Unlock。

注意：

1、             并不是所有的事件都会引起状态的转移，比如当门是处于【Opened】状态，不能进行【Lock】事件。

2、             转移（Transition）有警备条件（guard condition），比如只有doorWay->isEmpty 条件满足时，才会响应事件。

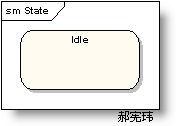
#### 二、状态图元素（State Diagram Elements）

## 

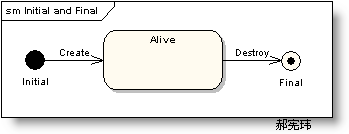
### 1、状态（States）

    指在对象的生命周期中的某个条件或者状况，在此期间对象将满足某些条件、执行某些活动活活等待某些事件。所有对象都有状态，状态是对象执行了一系列活动的结果，当某个事件发生后，对象的状态将发生变化。

状态用圆角矩形表示

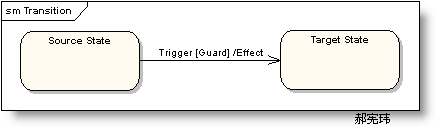


初态和终态（Initial and Final States）  
初态用实心圆点表示，终态用圆形内嵌圆点表示。



### 2、转移（Transitions）

    转移（Transitions）是两个状态之间的一种关系，表示对象将在源状态（Source State）中执行一定的动作，并在某个特定事件发生而且某个特定的警界条件满足时进入目标状态（Target State）



      事件标记（Trigger）：是转移的诱因，可以是一个信号，事件、条件变化（a change in some condition）和时间表达式。

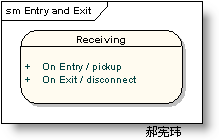
      警界条件（Guard Condition）：当警界条件满足时，事件才会引发转移（Transition）。

      结果（Effect）：对象状态转移后的结果。

### 3、动作（State Actions）

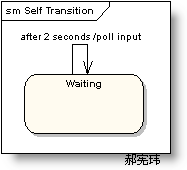
动作（Actions）是一个可执行的原子操作,也就是说动作是不可中断的，其执行时间是可忽略不计的。

在上例中，对象状态转移后的结果显示在转移线上，如果目标状态有许多转移，而且每个转移有相同的结果，这时把转移后的结果（Effect）展示在目标状态中（Target State）更好一些，可以定义进入动作（Entry Action ）和退出动作（Exit Action），如下图



### 4、自身转移（Self-Transitions）

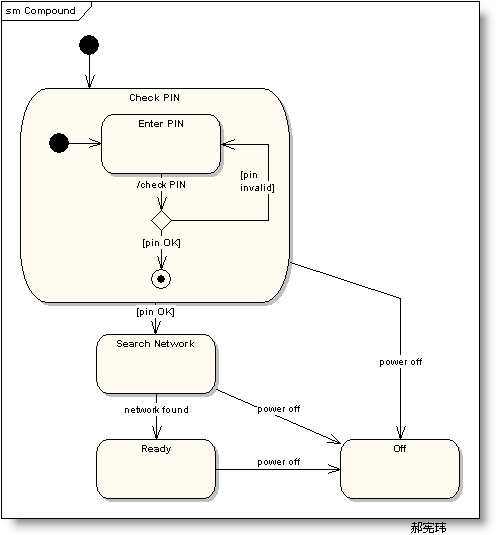
    状态可以有返回自身状态的转移，称之为自身转移（Self-Transitions）



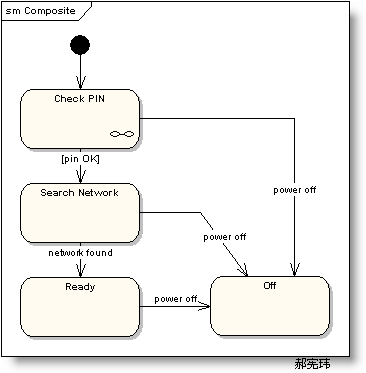
2S后，Poll input事件执行，转移到自己状态【Waiting】

### 5、组合状态（Compound States）

    嵌套在另外一个状态中的状态称之为子状态（sub-state）,一个含有子状态的状态被称作组合状态（Compound States）. 如下图，【Check PIN】是组合状态，【Enter PIN】是子状态。



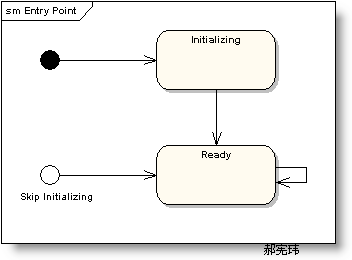
也可用以下方式进行描述



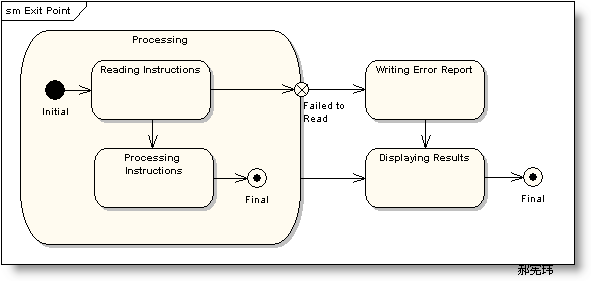
如上图，状态机【Check PIN】的细节被分割到另外一个图中了。

### 6、进入节点（Entry Point）

    如下图所示，由于一些原因并不会执行初始化（initialization），而是直接通过一个节点进入状态【Ready】，则此节点称之为进入节点（Entry Point）

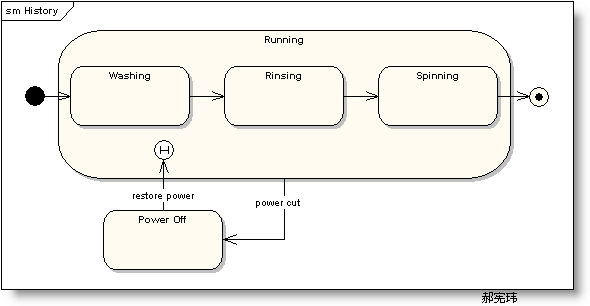


### 7、退出节点（Exit Point）



### 8、历史状态（History States）

    历史状态是一个伪状态（Pseudostate）,其目的是记住从组合状态中退出时所处的子状态，当再次进入组合状态，可直接进入这个子状态，而不是再次从组合状态的初态开始。

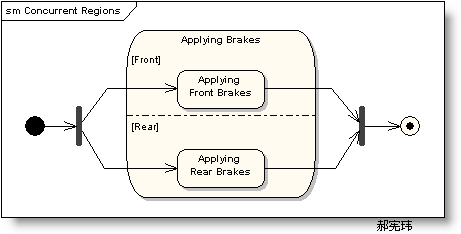


在上图的状态图中，正常的状态顺序是:【Washing】- >【Rinsing】->【Spinning】。

如果是从状态【Rinsing】突然停电（Power Cut）退出，,洗衣机停止工作进入状态【Power Off】，当电力恢复时直接进入状态【Running】。

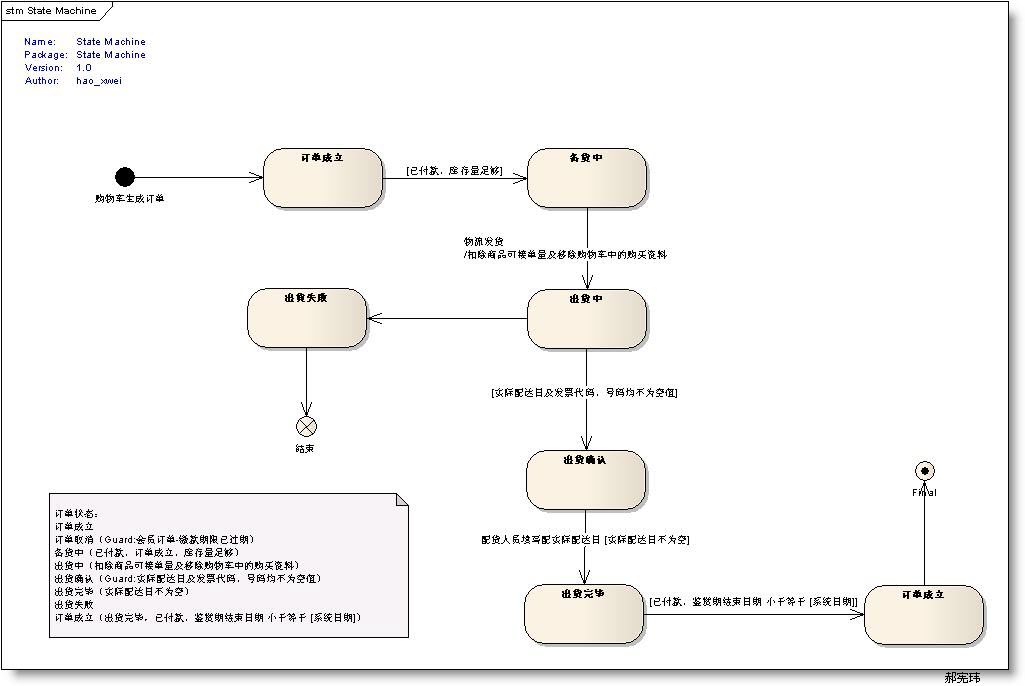
### 9、并发区域（Concurrent Regions）

    状态图可以分为区域，而区域又包括退出或者当前执行的子状态。说明组合状态在某一时刻可以同时达到多个子状态。如下图刹车系统，同时进入前刹车【Applying Front Brakes】状态和后刹车【Applying Rear Brakes】状态。

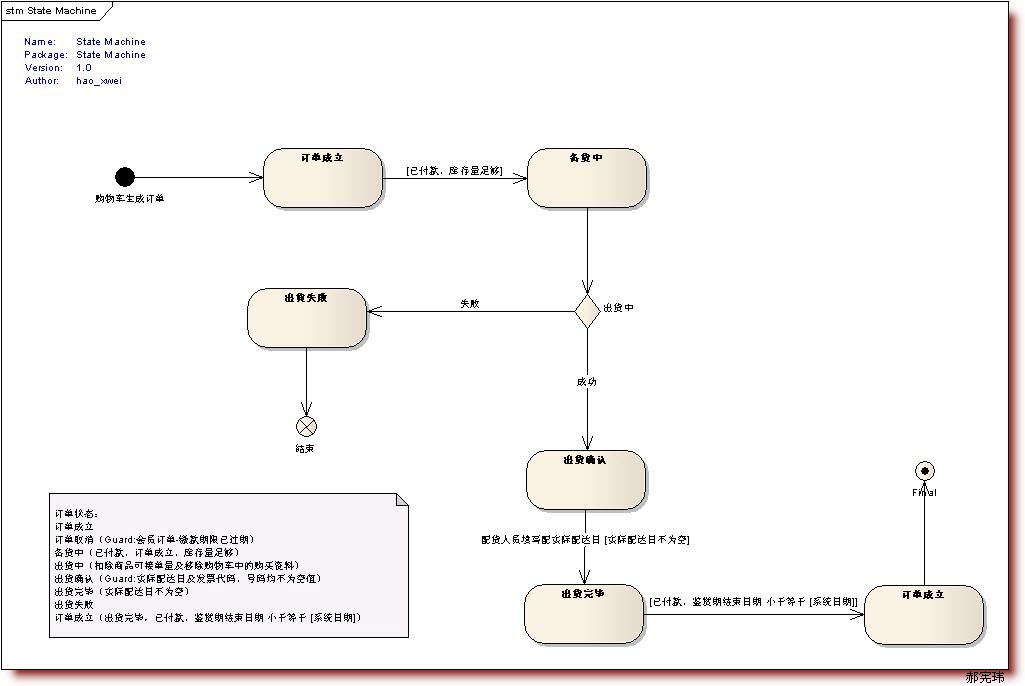


#### 三、状态图案例分析（State Diagram Example Analysis）

## 



按照blink518的建议（“出货中”是属于条件分支应该使用Decision），改成如下图也是很好的做法：



订单成立状态主要有：

订单成立

订单取消（Guard:会员订单-缴款期限已过期）

备货中（Guard:已付款、订单成立、库存量足够）

出货中（Effect:扣除商品可接单量及移除购物车中的购买资料）

出货确认（Guard:实际配达日及发票代码、号码均不为空值）

出货完毕（Guard:实际配达日不为空）

出货失败

订单成立（Guard:出货完毕，已付款、鉴赏期结束日期 小于等于 [系统日期]）

分析：

1、购物车生成订单进入状态【订单成立】

2、系统检测订单已经付款并且库存量足够，则进入状态【备货中】

3、物流发货，进入状态【发货中】，状态转移为【发货中】后，需要做的操作有“扣除商品可接单量及移除购物车中的购买资料”

4、发货完毕后，状态分为【出货确认】和状态【出货失败】，如果状态是【出货失败】，则【结束】，如果状态为【出货确认】，则进入下一步。

5、配货人员填写实际配达日期，进入状态【出货完毕】。

6、如果”已付款、鉴赏期结束日期 小于等于 [系统日期]”,则【订单成立】。

#### 四、总结（Summary）

## 

       状态图重点在于描述对象的状态及其状态之间的转移，状态图的基本元素主要有：状态、转移、动作、自身转移、组合状态、进入节点、退出节点、历史状态、并发区域等，状态中的事件分为调用事件（Call）、变化事件（Change）、时间事件（Time）和信号事件(Singal)。最后以实例对状态对进行了分析。

# UML之时序图解析

 一、时序图简介（Brief introduction）

       二、时序图元素（Sequence Diagram Elements）

角色（Actor）

对象（Object）

生命线（Lifeline）

控制焦点（Focus of Control）

消息（Message）

自关联消息（Self-Message）

Combined Fragments

   三、时序图实例分析（Sequece Diagram Example Analysis）

时序图场景

时序图实例

时序图实例分析

       四、总结（Summary）

## 一、时序图简介（Brief introduction）

       时序图（Sequence Diagram）是显示对象之间交互的图，这些对象是按时间顺序排列的。顺序图中显示的是参与交互的对象及其对象之间消息交互的顺序。时序图中包括的建模元素主要有：对象（Actor）、生命线（Lifeline）、控制焦点（Focus of control）、消息（Message）等等。

## 二、时序图元素（Sequence Diagram Elements）

### 角色（Actor）

   系统角色，可以是人、及其甚至其他的系统或者子系统。

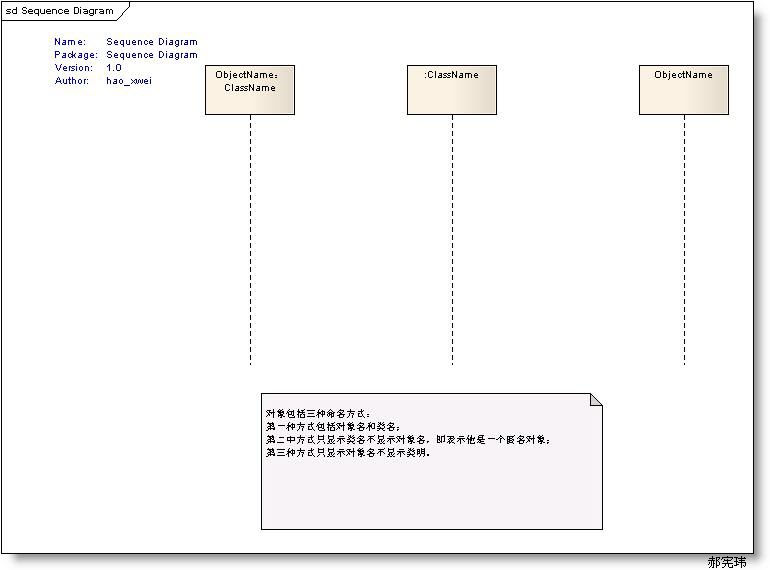
### 对象（Object）

  对象包括三种命名方式：

  第一种方式包括对象名和类名；

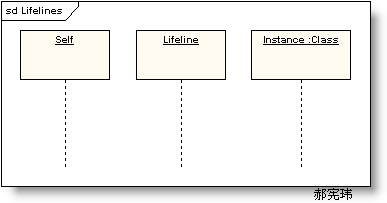
  第二中方式只显示类名不显示对象名，即表示他是一个匿名对象；

  第三种方式只显示对象名不显示类明。



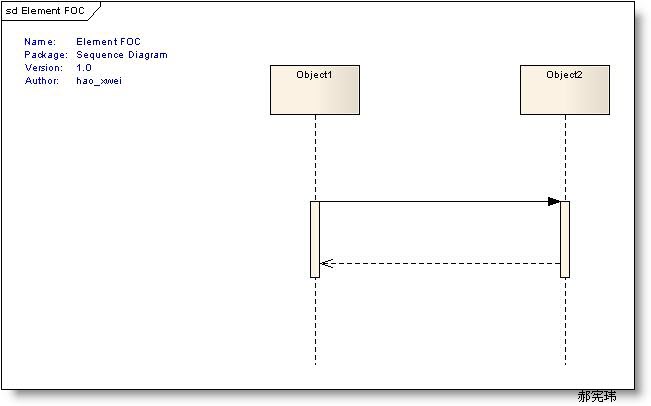
### 生命线（Lifeline）

  生命线在顺序图中表示为从对象图标向下延伸的一条虚线，表示对象存在的时间，如下图



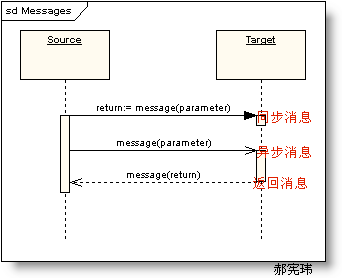
### 控制焦点（Focus of Control）

  控制焦点是顺序图中表示时间段的符号，在这个时间段内对象将执行相应的操作。用小矩形表示，如下图。



### 消息（Message）

  消息一般分为同步消息（Synchronous Message），异步消息（Asynchronous Message）和返回消息（Return Message）.如下图所示：



  同步消息=调用消息（Synchronous Message）

  消息的发送者把控制传递给消息的接收者，然后停止活动，等待消息的接收者放弃或者返回控制。用来表示同步的意义。

  异步消息（Asynchronous Message）

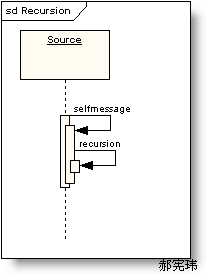
  消息发送者通过消息把信号传递给消息的接收者，然后继续自己的活动，不等待接受者返回消息或者控制。异步消息的接收者和发送者是并发工作的。

  返回消息（Return Message）

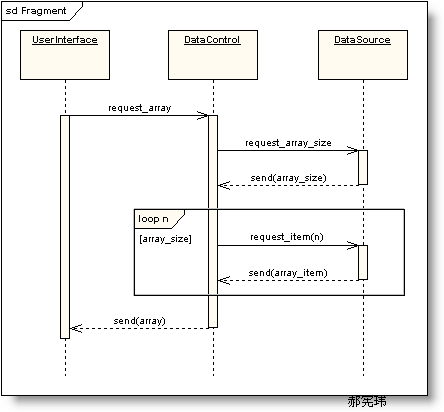
  返回消息表示从过程调用返回

### 自关联消息（Self-Message）

  表示方法的自身调用以及一个对象内的一个方法调用另外一个方法。



### Combined Fragments



  Ø         Alternative fragment（denoted “alt”） 与 if…then…else对应

  Ø         Option fragment (denoted “opt”) 与 Switch对应

  Ø         Parallel fragment (denoted “par”) 表示同时发生

  Ø         Loop fragment(denoted “loop”) 与 for 或者 Foreach对应

## 三、时序图实例分析（Sequece Diagram Example Analysis）

### 时序图场景

完成课程创建功能，主要流程有：

1、请求添加课程页面，填写课程表单，点击【create】按钮

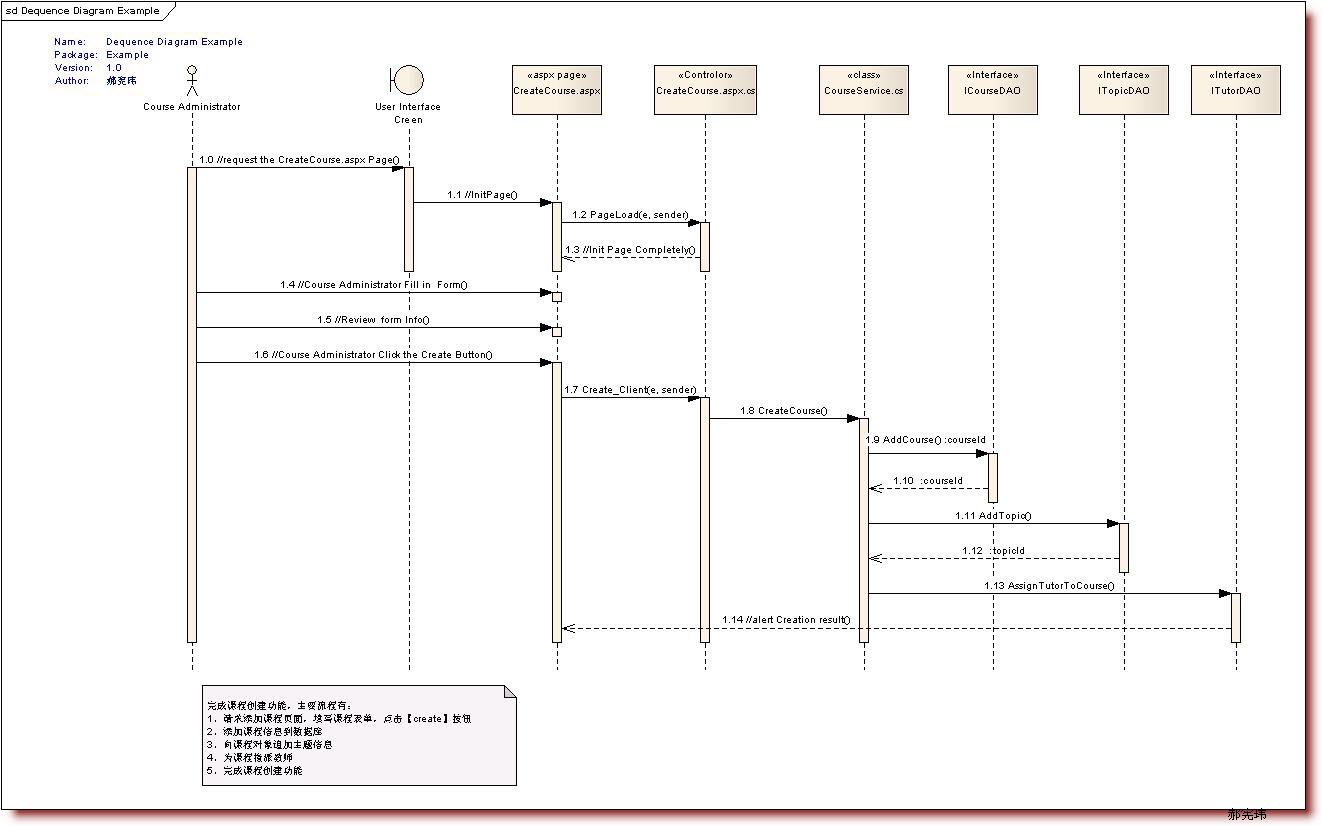
2、添加课程信息到数据库

3、向课程对象追加主题信息

4、为课程指派教师

5、完成课程创建功能

时序图实例



### 时序图实例分析

1、序号1.0-1.3  完成页面的初始化

2、序号1.4-1.5  课程管理员填充课程表单

3、序号1.6-1.7  课程管理员点击【Create】按钮，并响应点击事件

4、序号1.8     Service层创建课程

5、序号1.9-1.10 添加课程到数据库，并返回课程编号CourseId

6、序号1.11-1.12 添加课程主题到数据库，并返回主题编号topicId

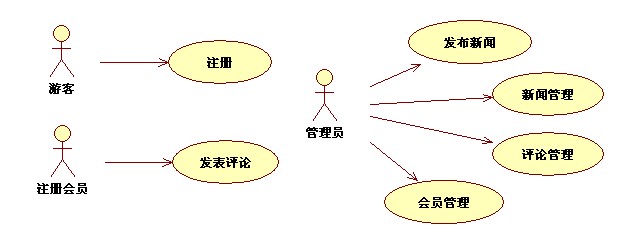
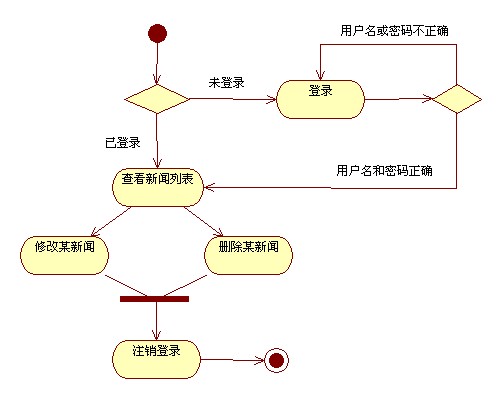
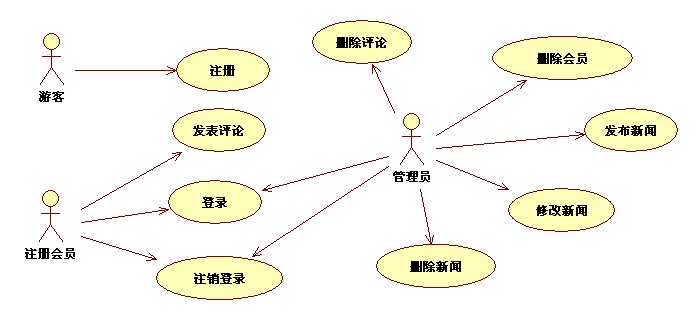
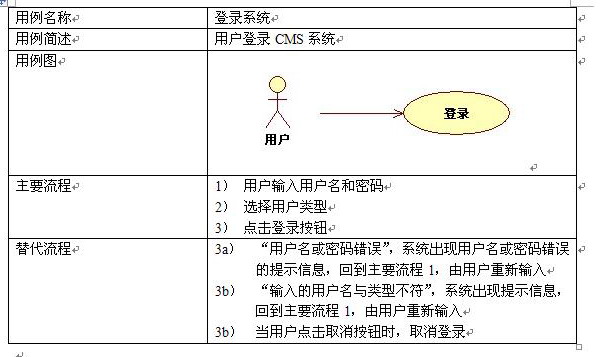
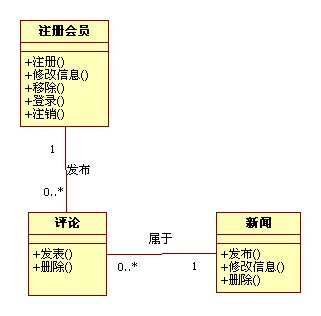
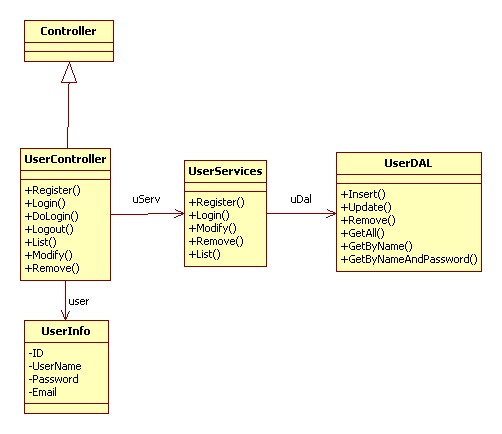
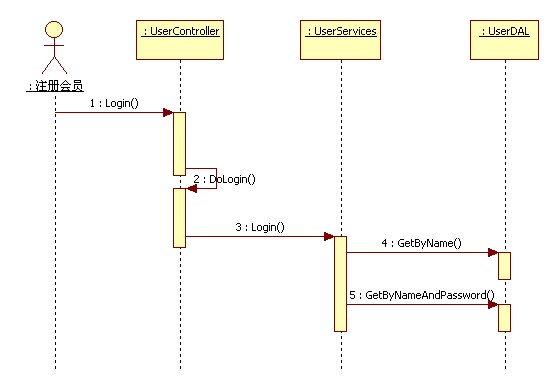
7、序号1.13         给课程指派教师

8、序号1.14         向界面抛创建课程成功与否的消息

## 四、总结（Summary）

       时序图（Sequence Diagram）是显示对象之间交互的图，这些对象是按时间顺序排列的。顺序图中显示的是参与交互的对象及其对象之间消息交互的顺序。时序图中包括的建模元素主要有：对象（Actor）、生命线（Lifeline）、控制焦点（Focus of control）、消息（Message）等等。最后，以课程创建功能演示一时序图实例。

# 实例演示 -- 基于UML的面向对象分析与设计

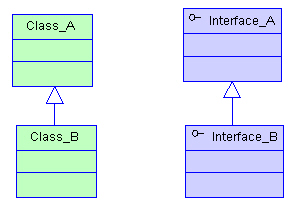
   摘要  
      本文以实例的方式，展示了如何使用UML进行面向对象的分析与设计。本文将假设读者对UML、面向对象等领域的基本内容已了然于胸，所以将不会过多阐述，而将重点放在应用过程上。本文的目的是通过一个完整的实例，展现基于UML的OOA&D过程 的一个简化模式，帮助朋友们更好的认识UML在OOA&D中起的作用。  
      前言  
      经常听到有朋友抱怨，说学了UML不知该怎么用，或者画了UML却觉得没什么作用。其实，就UML本身来说，它只是一种交流工具，它作为一种标准化交流符号，在OOA&D过程中开发人员间甚至开发人员与客户之间传递信息。另外，UML也可以看做是OO思想的一种表现形式，可以说“OO是神，而UML是型”。所以，想用好UML，扎实的OO思想基础是必不可少的。然而，在UML应用到开发过程中时，还是有一定的模式可以遵循的。（注意，是模式而不是教条，我下面给出的流程只是一个启发式过程，而不是说一定要遵循这个流程。）下面，我们通过一个CMS系统的分析设计实例，看看如何将UML应用到实际的开发中。  
      1.从需求到业务用例图  
      OOA&D的第一步，就是了解用户需求，并将其转换为业务用例图。我们的CMS系统需求非常简单，大致课做如下描述：这个系统主要用来发布新闻，管理员只需要一个，登录后可以在后台发布新闻。任何人可以浏览新闻，浏览者可以注册成为系统会员，注册后可对新闻进行评论。管理员在后台可以对新闻、评论、注册会员进行管理，如修改、删除等。  
      通过以上需求描述，我们画出如下的业务用例图：  
      这里要注意三点：  
      1.业务用例是仅从系统业务角度关注的用例，而不是具体系统的用例。它描述的是“该实现什么业务”，而不是“系统该提供什么操作”。例如，在实际系统中，“登录”肯定要作为一个用例，但是这是软件系统中的操作，而用户所关注的业务是不包含“登录”的。  
      2.业务用例仅包含客户“感兴趣”的内容。  
      3.业务用例所有的用例名应该让客户能看懂，如果某个用例的名字客户看不懂什么意思，它也许就不适合作为业务用例。  
      2.从业务用例图到活动图  
      完成了业务用例图后，我们要为每一个业务用例绘制一幅活动图。活动图描述了这个业务用例中，用户可能会进行的操作序列。活动图有个很重要的使命：从业务用例分析出系统用例。例如，下面是“新闻管理”的活动图：  
      可以看到，一个“新闻管理”这个业务用例，分解出N多系统操作。这里要特别注意这些操作，其中很多“活动”都很可能是一个系统用例（当然，不是每个都是）。例如，由这个活动图可以看出，系统中至少要包含以下备选系统用例：登录、注销登录、查看新闻列表、修改新闻、删除新闻。  
      这样，将每个业务用例都绘制出相应的活动图，再将其中的“活动”整合，就得出所有备选系统用例。  
      3.从活动图到系统用例图  
      找出所有的备选系统用例后，我们要对他们进行合并和筛选。合并就是将相同的用例合并成一个，筛选就是将不符合系统用例条件的备选用例去掉。  
      一个系统用例应该是实际使用系统的用户所进行的一个操作，例如，“查看新闻列表”就不能算一个系统用例，因为他只是某系统用例的一个序列项。  
      最终我们得出的系统用例图如下：  
      4.从系统用例图到用例规约  
      得出系统用例图后，我们应该对每一个系统用例给出用例规约。关于用例规约，没有一个通用的格式，大家可以按照习惯的格式进行编写。对用例规约唯一的要求就是“清晰易懂”。  
      下面给出“登录”这个系统用例的一个规约：  
      5.绘制业务领域类图  
      完成了上面几步，下面应该是绘制业务领域类图了。所谓业务领域类图要描述一下三点：  
      1.系统中有哪些实体。  
      2.这些实体能做什么操作。  
      3.实体间的关系。  
      这里要特别强调：这里的实体不是Actor，而是Actor使用系统时使用的所调用的实体，是处在系统边界之内的实体。例如，管理员就没有作为一个实体出现在这里，因为管理员处在系统边界之外，它所有的工作都可以通过调用这三个类的方法完成。并且，这里的“注册会员”实体也不是刚才用例图中注册会员这个Actor，而是作为一个系统内的业务实体，供Actor们使用的。例如，其中的注册功能是给注册会员这个Actor使用，而移除则是给管理员这个Actor使用的。  
      理解以上这段话非常重要，我经常看到由于混淆了实体和Actor的关系而导致画出的领域类图不准确或职责分配不准确。  
      大家可能还注意到，我们这里没有给出每个实体的属性。其实，在领域分析阶段，实体的属性并不重要，重要的是找出实体的操作。    
      6.绘制实现类图  
      以上这几步，就是分析的过程。而下面的步骤就是设计了。  
      设计没有分析那么好描述，因为分析是“客户面”，它只关心系统本身的功能和业务，而不关心任何和计算机有关的东西。但是，设计和平台、语言、开发模型等内容关系紧密，因而很难找出一个一致的过程。但是，一般在设计过程中实现类图是要绘制的。  
      实现类图和领域类图不一样，它描述的是真正系统的静态结构，是和最后的代码完全一致的。因此，它和平台关系密切，必须准确给出系统中的实体类、控制类、界面类、接口等元素以及其中的关系。因此，实现类图是很复杂的，而且是平台技术有关的。所以，我在这里不可能给出一个准确的实现类图，不过为了描述，我还是给出一个简化了的实现类图，当然，它是不准确的，而只是从形式上给出实现类图的样子。  
      我们假设这个系统建构于.NET 3.5平台上，并且使用ASP.NET MVC作为表示层，整体使用三层架构。那么，用户模块体系的实现类图大体是这样子（不准确）：  
      7.绘制序列图  
      有了静态结构，我们还要给出动态结构，这样，才能看清系统间的类是如何交互的，从而有效帮助程序员进行编码工作。  
      上图给出的是用户登录的序列图。首先注册会员作为Actor，调用UserController的Login方法启动序列，然后序列按图示步骤执行。其中UserServices作为业务组件，首先调用数据访问组件的GetByName确定用户是否存在，如果存在，再调用GetByNameAndPassword确定输入密码是否是此用户的密码。从而完成业务功能。  
      要注意，序列图在实际中是很多的，几乎每个类方法都配有相应的序列图。  
      8.后面的步骤  
      在完成了上面的过程后，就可以进行编码、调试、测试等工作了。但这些已经超出了本文讨论的范围。  
      总结  
      本文简要给出了使用UML进行OOA&D的过程。当然，由于示例较小，而且本人水平有限，所以给出的相关内容可能不是很准确。而且软件分析设计本来就不是一个固定模式的过程，随着系统的不同整个过程会有变化。本文只是想起到一个抛砖引玉的作用，让朋友们大致了解UML的使用流程。至于实际的分析设计，还需要深入的学习和实践的积累。

# UML类图关系（泛化 、继承、实现、依赖、关联、聚合、组合）

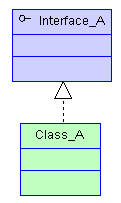
继承、实现、依赖、关联、聚合、组合的联系与区别

分别介绍这几种关系：

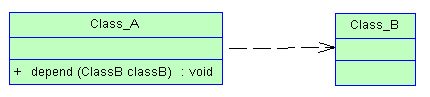
##### 继承

指的是一个类（称为子类、子接口）继承另外的一个类（称为父类、父接口）的功能，并可以增加它自己的新功能的能力，继承是类与类或者接口与接口之间最常见的关系；在Java中此类关系通过关键字extends明确标识，在设计时一般没有争议性；  


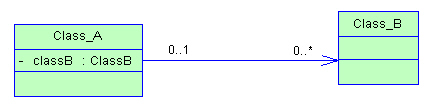
##### 实现

指的是一个class类实现interface接口（可以是多个）的功能；实现是类与接口之间最常见的关系；在Java中此类关系通过关键字implements明确标识，在设计时一般没有争议性；  


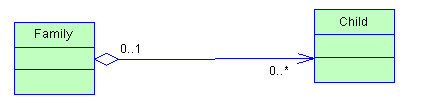
##### 依赖

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；表现在代码层面，为类B作为参数被类A在某个method方法中使用；  


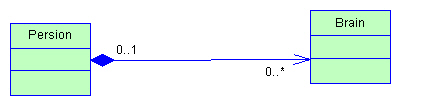
##### 关联

他体现的是两个类、或者类与接口之间语义级别的一种强依赖关系，比如我和我的朋友；这种关系比依赖更强、不存在依赖关系的偶然性、关系也不是临时性的，一般是长期性的，而且双方的关系一般是平等的、关联可以是单向、双向的；表现在代码层面，为被关联类B以类属性的形式出现在关联类A中，也可能是关联类A引用了一个类型为被关联类B的全局变量；  


##### 聚合

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；比如计算机与CPU、公司与员工的关系等；表现在代码层面，和关联关系是一致的，只能从语义级别来区分；  


##### 组合

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；比如你和你的大脑；表现在代码层面，和关联关系是一致的，只能从语义级别来区分；  


对于继承、实现这两种关系没多少疑问，他们体现的是一种类与类、或者类与接口间的纵向关系；其他的四者关系则体现的是类与类、或者类与接口间的引用、横向关系，是比较难区分的，有很多事物间的关系要想准备定位是很难的，前面也提到，这几种关系都是语义级别的，所以从代码层面并不能完全区分各种关系；

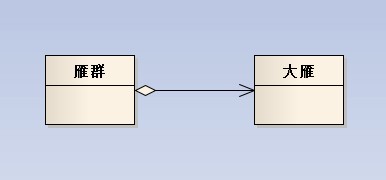
但总的来说，后几种关系所表现的强弱程度依次为：组合>聚合>关联>依赖；

聚合跟组合其实都属于关联 只不过它们是两种特殊的关联 因为本是同根生 所以它们之间难免会有相似之处 下面让我们一起来看一下它们之间有何不同

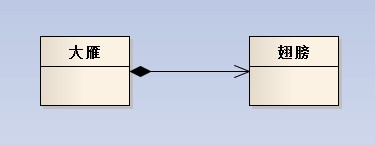
聚合与组合的概念相信不用我在此赘述大家就已经了解了 下面直接上例子

程老师的《大话》里举大那个大雁的例子很贴切 在此我就借用一下 大雁喜欢热闹害怕孤独 所以它们一直过着群居的生活 这样就有了雁群 每一只大雁都有自己的雁群 每个雁群都有好多大雁 大雁与雁群的这种关系就可以称之为聚合 另外每只大雁都有两只翅膀 大雁与雁翅的关系就叫做组合 有此可见 聚合的关系明显没有组合紧密 大雁不会因为它们的群主将雁群解散而无法生存 而雁翅就无法脱离大雁而单独生存——组合关系的类具有相同的生命周期

聚合关系图：



组合关系图：



从从代码上看这两种关系的区别在于：

构造函数不同

雁群类：

1. **public class GooseGroup{**
2. **private Goose goose;**
3. **public GooseGroup(Goose goose){**
4. **this.goose = goose;**
5. }
6. }

大雁类：

1. **public class Goose{**
2. **private Wings wings;**
3. **public Goose(){**
4. wings=**new Wings();**
5. }
6. }

聚合关系的类里含有另一个类作为参数   
雁群类（GooseGroup）的构造函数中要用到大雁（Goose）作为参数把值传进来 大雁类（Goose）可以脱离雁群类而独立存在   
组合关系的类里含有另一个类的实例化   
大雁类（Goose）在实例化之前 一定要先实例化翅膀类（Wings） 两个类紧密耦合在一起 它们有相同的生命周期 翅膀类（Wings）不可以脱离大雁类（Goose）而独立存在  
信息的封装性不同   
在聚合关系中，客户端可以同时了解雁群类和大雁类，因为他们都是独立的   
而在组合关系中，客户端只认识大雁类，根本就不知道翅膀类的存在，因为翅膀类被严密的封装在大雁类中。

-------------------------------------------------------------------------------------------------------

UML-泛化、关联、聚合、组合、依赖

一、泛化关系（generalization）

1.说明

表示类与类之间的继承关系，接口与接口之间的继承关系，或类对接口的实现关系。一般化的关系是从子类指向父类的，与继承或实现的方法相反。

2.例图

3.表现

父类 父类实例＝new 子类();

4.举例

class Animal{};

class Tigger : public Animal{};

class Dog : public Animal{};

Animal\* pAnimal = new Dog;

二、关联关系（association）

1.说明

对于两个相对独立的对象，当一个对象的实例与另一个对象的一些特定实例存在固定的对应关系时，这两个对象之间为关联关系。

表示类与类之间的联接，有双向关联和单向关联，双向关联有两个箭头或者没有箭头，单向关联有一个箭头，表示关联的方向。

关联关系以实例变量的形式存在，在每一个关联的端点，还可以有一个基数(multiplicity),表明这一端点的类可以有几个实例。

2.例图

3.表现

双向关联在代码的表现为双方都拥有对方的一个指针，当然也可以是引用或者是值。

关联关系是使用实例变量来实现。

4.举例

//eg.1

//单向关联

class Person{};

class Friend

{

Person\* mpPerson;

};

//eg.2

//双向关联

class A;

class B

{

A\* pA;

};

class A

{

B\* pB;

};

//eg.3

//自身关联

class C

{

C\* pC;

};

三、聚合关系（aggregation）

1.说明：

关联关系的一种，是强的关联关系。聚合是整体和个体的关系。聚合关系也是通过实例变量实现的。例如汽车、发动机、轮胎，一个汽车对象由一个发动机对象，四个轮胎对象组成。

当类之间有整体-部分关系的时候，我们就可以使用组合或者聚合。

2.例图

3.表现

与关联关系一样，聚合关系也是通过实例变量来实现这样关系的。关联关系和聚合关系来语法上是没办法区分的，从语义上才能更好的区分两者的区别。

4.举例

class CPU{};

class Memory{};

class Computer

{

CPU\* mpCPU;

Memory\* mpMemory;

};

四、组合关系（合成关系）（composition）

1.说明：

合成关系也是关联关系的一种，是比聚合关系更强的关系。合成关系是不能共享的。例如人有四肢、头等。

表示类之间整体和部分的关系，组合关系中部分和整体具有统一的生存期。一旦整体对象不存在，部分对象也将不存在。部分对象与整体对象之间具有共生死的关系。

2.例图

3.表现

4.举例

//同聚合关系，不过说语义不同

class Leg{};

class Arm{};

class Person

{

Leg mLeg;

Arm mArm;

};

五、依赖关系（Dependency）

1.说明：

对于两个相对独立的对象，当一个对象负责构造另一个对象的实例，或者依赖另一个对象的服务时，这两个对象之间主要体现为依赖关系。

与关联关系不同的是，依赖关系是以参数变量的形式传入到依赖类中的，依赖是单向的，要避免双向依赖。一般来说，不应该存在双向依赖。

依赖是一种弱关联，只要一个类用到另一个类，但是和另一个类的关系不是太明显的时候（可以说是“uses”了那个类），就可以把这种关系看成是依赖。

2.例图

3.表现

依赖关系表现在局部变量，方法的参数，以及对静态方法的调用

4.举例

class Car{};

class House{};

class Person

{

void buy(Car& car){}

void buy(House\* pHouse){}

};

六、关系之间的区别

1.聚合与组合

（1）聚合与组合都是一种结合关系，只是额外具有整体-部分的意涵。

（2）部件的生命周期不同

聚合关系中，整件不会拥有部件的生命周期，所以整件删除时，部件不会被删除。再者，多个整件可以共享同一个部件。   
组合关系中，整件拥有部件的生命周期，所以整件删除时，部件一定会跟着删除。而且，多个整件不可以同时间共享同一个部件。

（3）聚合关系是“has-a”关系，组合关系是“contains-a”关系。

2.关联和聚合

（1）表现在代码层面，和关联关系是一致的，只能从语义级别来区分。

（2）关联和聚合的区别主要在语义上，关联的两个对象之间一般是平等的，例如你是我的朋友，聚合则一般不是平等的。

（3）关联是一种结构化的关系，指一种对象和另一种对象有联系。

（4）关联和聚合是视问题域而定的，例如在关心汽车的领域里，轮胎是一定要组合在汽车类中的，因为它离开了汽车就没有意义了。但是在卖轮胎的店铺业务里，就算轮胎离开了汽车，它也是有意义的，这就可以用聚合了。

3.关联和依赖

（1）关联关系中，体现的是两个类、或者类与接口之间语义级别的一种强依赖关系，比如我和我的朋友；这种关系比依赖更强、不存在依赖关系的偶然性、关系也不是临时性的，一般是长期性的，而且双方的关系一般是平等的。

（2）依赖关系中，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。

4.综合比较

这几种关系都是语义级别的，所以从代码层面并不能完全区分各种关系；但总的来说，后几种关系所表现的强弱程度依次为：

组合>聚合>关联>依赖；

-----------------------------------------------------------------------------------------------------------------------------------------------

UML 线条 箭头

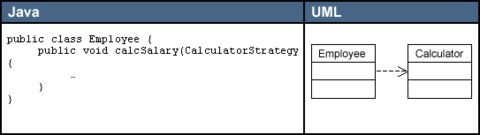
关系

后面的例子将针对某个具体目的来独立地展示各种关系。虽然语法无误，但这些例子可进一步精炼，在它们的有效范围内包括更多的语义。

依赖（Dependency）

实体之间一个“使用”关系暗示一个实体的规范发生变化后，可能影响依赖于它的其他实例（图D）。 更具体地说，它可转换为对不在实例作用域内的一个类或对象的任何类型的引用。其中包括一个局部变量，对通过方法调用而获得的一个对象的引用（如下例所 示），或者对一个类的静态方法的引用（同时不存在那个类的一个实例）。也可利用“依赖”来表示包和包之间的关系。由于包中含有类，所以你可根据那些包中的 各个类之间的关系，表示出包和包的关系。

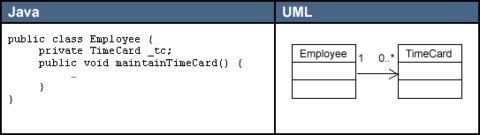
图D



关联（Association）

实体之间的一个结构化关系表明对象是相互连接的。箭头是可选的，它用于指定导航能力。如果没有箭头，暗示是一种双向的导航能力。在Java中，关联（图E） 转换为一个实例作用域的变量，就像图E的“Java”区域所展示的代码那样。可为一个关联附加其他修饰符。多重性（Multiplicity）修饰符暗示 着实例之间的关系。在示范代码中，Employee可以有0个或更多的TimeCard对象。但是，每个TimeCard只从属于单独一个 Employee。

图E

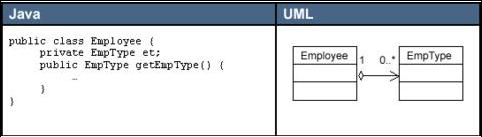


聚合（Aggregation）

聚合（图F）是关联的一种形式，代表两个类之间的整体/局部关系。聚合暗示着整体在概念上处于比局部更高的一个级别，而关联暗示两个类在概念上位于相同的级别。聚合也转换成Java中的一个实例作用域变量。

关联和聚合的区别纯粹是概念上的，而且严格反映在语义上。聚合还暗示着实例图中不存在回路。换言之，只能是一种单向关系。

图F

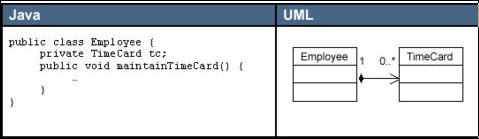


合成（Composition）

合成 （图G） 是聚合的一种特殊形式，暗示“局部”在“整体”内部的生存期职责。合成也是非共享的。所以，虽然局部不一定要随整体的销毁而被销毁，但整体要么负责保持局 部的存活状态，要么负责将其销毁。局部不可与其他整体共享。但是，整体可将所有权转交给另一个对象，后者随即将承担生存期职责。

Employee和TimeCard的关系或许更适合表示成“合成”，而不是表示成“关联”。

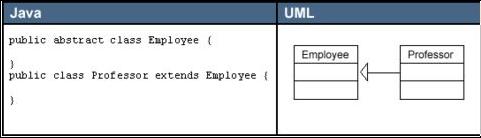
图G



泛化（Generalization）

泛化（图H）表示一个更泛化的元素和一个更具体的元素之间的关系。泛化是用于对继承进行建模的UML元素。在Java中，用extends关键字来直接表示这种关系。

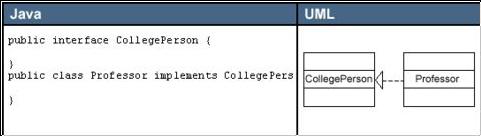
图H



实现（Realization）

实例（图I）关系指定两个实体之间的一个合同。换言之，一个实体定义一个合同，而另一个实体保证履行该合同。对Java应用程序进行建模时，实现关系可直接用implements关键字来表示。

图I



-----------------------------------------------------------------------------------------------------------------------------------------------

UML类图关系主要有关联，依赖，泛化，实现等，那么它们的表示方法你是否熟悉，本文就像大家介绍一下UML类图关系的表示方法。

AD：

本节和大家一起学习一下UML类图关系的表示方法，主要包括关联，聚合，泛化，实现，依赖等内容，希望通过本节的学习大家对UML类图关系的表示方法有一定的掌握。下面是具体介绍。

UML基础

1：UML类间关系的种类

2：关联

UML类图关系中关联描述了系统中对象或实例之间的离散连接，关联带有系统中各个对象之间关系的信息。

2.1关联表示法

2.2聚集与组合

3：泛化，继承【Generalization】

UML类图关系中泛化关系是类元的一般描述和具体描述之间的关系，具体描述建立在一般描述的基础之上，并对其进行了扩展。

4：实现【realization】

UML类图关系中实现关系将一种模型元素（如类）与另一种模型元素（如接口）连接起来，其中接口只是行为的说明而不是结构或者实现。

5：依赖【Dependence】

UML类图关系中依赖表示两个或多个模型元素之间语义上的关系。它只将模型元素本身连接起来而不需要用一组实例来表达它的意思。它表示了这样一种情形，提供者的某些变化会要求或指示依赖关系中客户的变化。

5.1依赖的种类

访问：允许一个包访问另一个包【access】

绑定：为模板参数赋值以生成一个新的模型元素【bind】

调用：声明一个类调用其他类的方法【call】

导出：声明一个实例可以从另一个实例中到处【derive】

友元：允许一个元素访问另一个元素而不论被访问元素的可见性【friend】

引入：允许一个包访问另一个包的内容并未被访问包的组成部分添加别名【import】

实例化：关于一个类的方法生成了另一个类的实例的生命【instantate】

参数：一个操作和他参数之间的关系【parameter】

实现：说明和其实之间的映射关系【realize】

精化：声明具有两个不同层次上元素的映射关系【refine】

发送：信号发送者和信号接受者之间的关系【send】

跟踪：声明不同模型中元素之间的连接，没有映射精确【trace】

使用：声明使用一个模型元素需要已存在的另一个模型元素，这样才能正确实现使用者的功能（调用，实例化，参数，发送）【use】

6：约束

UML类图关系中约束可以用来表示各种非局部的关系，如关联路径上的限制。约束尤其可以用来表述存在特性（存在X则C条件成立）和通用特性（对于Y中的所有y，条件D必须成立）。

7：实例

实例是有身份标识的运行实体，即它可以与其他运行实体相区分。它在任何时刻都有一个值，随着对实例进行操作值也会被改变。

-----------------------------------------------------------------------------------------------------------------------------------------------

类与类之间存在以下关系:   
(1)泛化(Generalization)   
(2)关联(Association)   
(3)依赖(Dependency)   
(4)聚合(Aggregation)   
UML图与应用代码例子:   
1.泛化(Generalization)   
[泛化]   
表示类与类之间的继承关系，接口与接口之间的继承关系，或类对接口的实现关系。一般化的关系是从子类指向父类的，与继承或实现的方法相反。   
[具体表现]   
父类 父类实例＝new 子类()   
[UML图](图1.1)   
图1.1 Animal类与Tiger类,Dog类的泛化关系   
[代码表现]   
class Animal{}   
class Tiger extends Animal{}   
public class Test   
{   
public void test()   
{   
Animal a=new Tiger();   
}   
}   
2.依赖(Dependency)   
[依赖]   
对于两个相对独立的对象，当一个对象负责构造另一个对象的实例，或者依赖另一个对象的服务时，这两个对象之间主要体现为依赖关系。   
[具体表现]   
依赖关系表现在局部变量，方法的参数，以及对静态方法的调用   
[现实例子]   
比如说你要去拧螺丝，你是不是要借助(也就是依赖)螺丝刀(Screwdriver)来帮助你完成拧螺丝(screw)的工作   
[UML表现](图1.2)   
图1.2 Person类与Screwdriver类的依赖关系   
[代码表现]   
public class Person{   
/\*\* 拧螺丝 \*/   
public void screw(Screwdriver screwdriver){   
screwdriver.screw();   
}   
}   
3.关联(Association)   
[关联]   
对于两个相对独立的对象，当一个对象的实例与另一个对象的一些特定实例存在固定的对应关系时，这两个对象之间为关联关系。   
[具体表现]   
关联关系是使用实例变量来实现   
[现实例子]   
比如客户和订单，每个订单对应特定的客户，每个客户对应一些特定的订单；再例如公司和员工，每个公司对应一些特定的员工，每个员工对应一特定的公司   
[UML图] (图1.3)   
图1.3 公司和员工的关联关系   
[代码表现]   
public class Company{   
private Employee employee;   
public Employee getEmployee(){   
return employee;   
}   
public void setEmployee(Employee employee){   
this.employee=employee;   
}   
//公司运作   
public void run(){   
employee.startWorking();   
}   
}   
(4)聚合（Aggregation）   
[聚合]   
当对象A被加入到对象B中，成为对象B的组成部分时，对象B和对象A之间为聚集关系。聚合是关联关系的一种，是较强的关联关系，强调的是整体与部分之间的关系。   
[具体表现]   
与关联关系一样，聚合关系也是通过实例变量来实现这样关系的。关联关系和聚合关系来语法上是没办法区分的，从语义上才能更好的区分两者的区别。   
[关联与聚合的区别]   
(1)关联关系所涉及的两个对象是处在同一个层次上的。比如人和自行车就是一种关联关系，而不是聚合关系，因为人不是由自行车组成的。   
聚合关系涉及的两个对象处于不平等的层次上，一个代表整体，一个代表部分。比如电脑和它的显示器、键盘、主板以及内存就是聚集关系，因为主板是电脑的组成部分。   
(2)对于具有聚集关系（尤其是强聚集关系）的两个对象，整体对象会制约它的组成对象的生命周期。部分类的对象不能单独存在，它的生命周期依赖于整体类的 对象的生命周期，当整体消失，部分也就随之消失。比如张三的电脑被偷了，那么电脑的所有组件也不存在了，除非张三事先把一些电脑的组件（比如硬盘和内存） 拆了下来。  
[UML图](图1.4)   
图1.3 电脑和组件的聚合关系   
[代码表现]   
public class Computer{   
private CPU cpu;   
public CPU getCPU(){   
return cpu;   
}   
public void setCPU(CPU cpu){   
this.cpu=cpu;   
}   
//开启电脑   
public void start(){   
//cpu运作   
cpu.run();   
}   
}

-----------------------------------------------------------------------------------------------------------------------------------------------

##### 类图及类图中的关系

1.类图和对象图

类图(Class Diagram)是显示出类、接口以及他们之间的静态结构与关系的图。其中最基本的单元是类或接口。

类图不但可以表示类(或者接口)之间的关系，也可以表示对象之间的关系。下面是一个典型的类图：

类图一般分为几个部分：类名、属性、方法。下面分别讲解。

(1)类名

上面的Car就是类名，如果类名是正体字，则说明该类是一个具体的类，如果类名是斜体字，则说明类是一个抽象类abstract。

(2)属性列表

属性可以是public、protected、private。public前面的图标是菱形，protected对应的是菱形加钥匙，private对应的是菱形加锁。当然，这只是一种表现方式。我是用的是Rational Rose，如果用的是别的软件，还可能使用+、-、#表示：+代表public、-代表private、#代表protected。

(3)方法列表

方法可以是public、protected、private。public前面的图标是菱形，protected对应的是菱形加钥匙，private对应的是菱形加锁。当然，这只是一种表现方式。我是用的是Rational Rose，如果用的是别的软件，还可能使用+、-、#表示：+代表public、-代表private、#代表protected。

对于静态属性，属性名会加上一条下划线。如上图所示。

此外，类图既能表示类之间的关系，还能表示对象之间的关系。二者的区别是：对象图中对象名下面会加上一条下划线。

2.类图中的关系

(1)Generalization：泛化、一般化

Generalization表示的是类与类之间的继承关系、接口与接口之间的继承关系、类与接口之间的实现关系。如果体现到Java语言中，那就是反应extends和implements关键字。其典型类图如下所示：

(2)Association：关联关系

关联关系描述的是类与类之间的连接，他表示一个类知道另一个类的属性和方法。关联关系可以是单向的或者双向的。在Java语言中，单向的关联关系是通过以实例变量的方式持有被关联对象的引用来实现的。一般来说是不建议使用双向的关联关系的。下面举例介绍单向的关联关系。

上面的类图表现的是骑手和马之间的关系。Rider中有一个实例变量类型是Horse。

每个连接都会有两个端点，上面的Rider和Horse就是端点，且每个端点都可以有(optional)一个基数(multiplicity)，表示这个类可以有几个实例。这个类似于数据库中的1：n、m：n这些关系。我们可以给上面的例子加上基数：

上面表示的是骑手与马之间的1对n关系。

(3)Aggregation：聚合关系

聚合关系是关联关系的一部分，是非常强的关联关系。聚合关系表现的更多的是整体与部分的关系。例如汽车和车门、发动机之间的关系。如图所示：

与关联关系一样，聚合关系也是通过实例变量实现的。单纯从语法的角度基本上无法判断出关联关系和聚合关系。

(4)Composition:组合关系

组合关系同样也是关联关系中的一种，这种关系是比聚合关系更加强的关系。我们前面提到，聚合关系表现的是整体与部分之间的关系，组合关系是在聚合关系的基础上，表示不可分割的整体与部分之间的关系。也就是说表示整体的对象需要负责表示部分的对象的生命周期。

“代表整体的对象负责保持代表部分的对象的存活，在一些情况下负责将代表部分的对象湮灭掉。代表整体的对象某些时候可以将代表部分的对象传递给另外一个对象，并由它负责代表部分的对象的生命周期。换言之，代表部分的对象同一时刻只能与一个对象构成组合关系。并且由后者排他的负责其生命周期。”——[《Java与模式》](https://www.baidu.com/s?wd=%E3%80%8AJava%E4%B8%8E%E6%A8%A1%E5%BC%8F%E3%80%8B&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)

我们以人和手臂的关系举例，组合关系的类图如下：

(5)Dependency：依赖关系

依赖关系表示一个类依赖于另一个类的定义。依赖关系是单方向的。人吃苹果，那么人依赖苹果。类图如下：

一般来说，被依赖的对象往往是以局部变量、方法参数的形式存在于来对象中，与关联关系不同，它不会以成员变量的形式存在于以来对象中。这一点值得注意。另外，每一个依赖都有一个名称。上面这个依赖关系的名称就是eats。