# 概述

## UML简介

UML (Unified Modeling Language)为面向对象软件设计提供统一的、标准的、可视化的建模语言。适用于描述以用例为驱动，以体系结构为中心的软件设计的全过程。

UML的定义包括UML语义和UML表示法两个部分。

UML语义：UML对语义的描述使开发者能在语义上取得一致认识，消除了因人而异的表达方法所造成的影响。

UML表示法：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图形符号和文本语法为系统建模提供了标准。

## UML的构成

事物(Things)：UML模型中最基本的构成元素，是具有代表性的成分的抽象

关系(Relationships)：关系把事物紧密联系在一起

图(Diagrams )：图是事物和关系的可视化表示

### UML事物

UML包含4种事物：构件事物、行为事物、分组事物、注释事物。

1. **构件事物： UML模型的静态部分，描述概念或物理元素**

它包括以下几种：

* 类：具有相同属性相同操作 相同关系相同语义的对象的描述
* 接口：描述元素的外部可见行为，即服务集合的定义说明
* 协作：描述了一组事物间的相互作用的集合
* 用例：代表一个系统或系统的一部分行为，是一组动作序列的集合
* 构件：系统中物理存在，可替换的部件
* 节点：运行时存在的物理元素

另外，参与者、信号应用、文档库、页表等都是上述基本事物的变体

1. **行为事物：UML模型图的动态部分，描述跨越空间和时间的行为**

* 交互：实现某功能的一组构件事物之间的消息的集合，涉及消息、动作序列、链接
* 状态机：描述事物或交互在生命周期内响应事件所经历的状态序列

1. **分组事物： UML模型图的组织部分，描述事物的组织结构**

* 包： 把元素组织成组的机制

1. **注释事物： UML模型的解释部分，用来对模型中的元素进行说明，解释**

* 注解 ：对元素进行约束或解释的简单符号

### UML关系

1. **依赖**

依赖(dependency)是两个事物之间的语义关系，其中一个事物(独立事物)发生变化，会影响到另一个事物(依赖事物)的语义

1. **关联**

关联(association)是一种结构关系，它指明一个事物的对象与另一个事物的对象间的联系。

1. **泛化**

泛化(generalization)是一种特殊/一般的关系。也可以看作是常说的继承关系

1. **实现**

实现(realization)是类元之间的语义关系，其中的一个类元指定了由另一个类元保证执行的契约

### UML图

1. **结构型（Structure Diagram）**

* 类图（Class Diagram）
* 对象图（Object Diagram）
* 构件图（Component Diagram）
* 部署图（Deployment Diagram）
* 包图（Package Diagram）

1. **行为型（Behavior Diagram）**
   * 活动图（Activity Diagram）
   * 状态机图（State Machine Diagram）
   * 顺序图（Sequence Diagram）
   * 通信图（Communication Diagram）
   * 用例图（Use Case Diagram）
   * 时序图（Timing Diagram）

## UML绘图工具

* RationalRose
* Microsoft Office Visio
* PowerDesigner
* Star UML
* Argo UML

# 类图（Class Diagram）

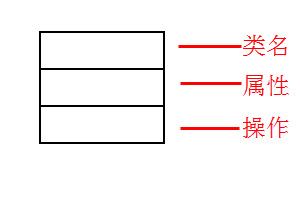
## 类图的概念

类图以反映类的结构(属性、操作)以及类之间的关系为主要目的，描述了软件系统的结构，是一种静态建模方法

类图中的“类”与面向对象语言中的“类”的概念是对应的，是对现实世界中的事物的抽象

## 类图的组成

类图由三部分组成：类名、属性和操作（方法），其中类名是必须有的。



类如果有属性，则每一个属性都必须有一个名字，另外还可以有其它的描述信息，如可见性、数据类型、缺省值等

类如果有操作，则每一个操作也都有一个名字，其它可选的信息包括可见性、参数的名字、参数类型、参数缺省值和操作的返回值的类型等



在UML类图中，类一般由三部分组成：

(1) 第一部分是类名：每个类都必须有一个名字，类名是一个字符串。

(2) 第二部分是类的属性(Attributes)：属性是指类的性质，即类的成员变量。一个类可以有任意多个属性，也可以没有属性

UML规定属性的表示方式为：

可见性 名称:类型 [ = 缺省值 ]

其中：

“可见性”表示该属性对于类外的元素而言是否可见，包括公有(public)、私有(private)和受保护(protected)三种，在类图中分别用符号+、-和#表示。

“名称”表示属性名，用一个字符串表示。

“类型”表示属性的数据类型，可以是基本数据类型，也可以是用户自定义类型。

“缺省值”是一个可选项，即属性的初始值。

(3) 第三部分是类的操作(Operations)：操作是类的任意一个实例对象都可以使用的行为，是类的成员方法。

UML规定操作的表示方式为：

可见性 名称(参数列表) [ : 返回类型]

其中：

“可见性”的定义与属性的可见性定义相同。

“名称”即方法名，用一个字符串表示。

“参数列表”表示方法的参数，其语法与属性的定义相似，参数个数是任意的，多个参数之间用逗号“，”隔开。

“返回类型”是一个可选项，表示方法的返回值类型，依赖于具体的编程语言，可以是基本数据类型，也可以是用户自定义类型，还可以是空类型(void)，如果是构造方法，则无返回类型。

## 类图之间的关系表示

### 关联关系

关联(Association)关系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

1. **双向关联**

默认情况下，关联是双向的。例如：顾客(Customer)购买商品(Product)并拥有商品，反之，卖出的商品总有某个顾客与之相关联。因此，Customer类和Product类之间具有双向关联关系，如图2所示：



代码：

|  |
| --- |
| public class Customer {  private Product[] products;  ……  }  public class Product {  private Customer customer;  ……  } |

1. **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。例如：顾客(Customer)拥有地址(Address)，则Customer类与Address类具有单向关联关系，如图3所示：



代码：

|  |
| --- |
| public class Customer {  private Address address;  ……  }  public class Address {  ……  } |

1. **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。例如：一个节点类(Node)的成员又是节点Node类型的对象。



|  |
| --- |
| public class Node {  private Node subNode;  ……  } |

1. **多重性关联**

多重性关联关系又称为重数性(Multiplicity)关联关系，表示两个关联对象在数量上的对应关系。在UML中，对象之间的多重性可以直接在关联直线上用一个数字或一个数字范围表示。

|  |  |
| --- | --- |
| 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) |

1. **聚合关系**

聚合(Aggregation)关系表示整体与部分的关系。在聚合关系中，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。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6所示：



1. 组合关系

组合(Composition)关系也表示类之间整体和部分的关系，但是在组合关系中整体对象可以控制成员对象的生命周期，一旦整体对象不存在，成员对象也将不存在，成员对象与整体对象之间具有同生共死的关系。在UML中，组合关系用带实心菱形的直线表示。例如：人的头(Head)与嘴巴(Mouth)，嘴巴是头的组成部分之一，而且如果头没了，嘴巴也就没了，因此头和嘴巴是组合关系，如图7所示：



### 依赖关系

依赖(Dependency)关系是一种使用关系，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。

例如：

驾驶员开车，在Driver类的drive()方法中将Car类型的对象car作为一个参数传递，以便在drive()方法中能够调用car的move()方法，且驾驶员的drive()方法依赖车的move()方法，因此类Driver依赖类Car，

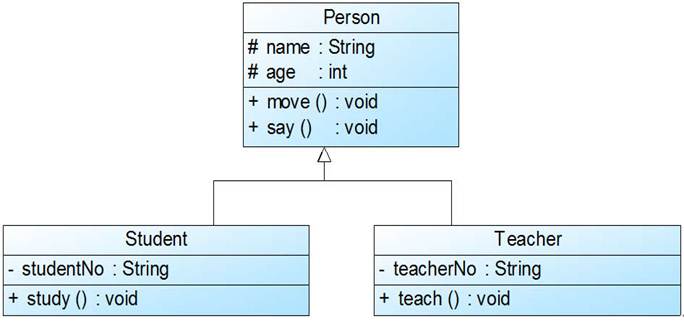


在系统实施阶段，依赖关系通常通过三种方式来实现，第一种也是最常用的一种方式是如图1所示的将一个类的对象作为另一个类中方法的参数，第二种方式是在一个类的方法中将另一个类的对象作为其局部变量，第三种方式是在一个类的方法中调用另一个类的静态方法。图1对应的Java代码片段如下：

|  |
| --- |
| public class Driver {  public void drive(Car car) {  car.move();  }  ……  }  public class Car {  public void move() {  ......  }  ……  } |

### 泛化关系

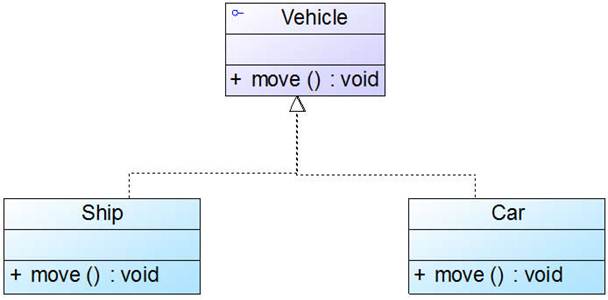
泛化(Generalization)关系也就是继承关系，用于描述父类与子类之间的关系，父类又称作基类或超类，子类又称作派生类。在UML中，泛化关系用带空心三角形的直线来表示。在代码实现时，我们使用面向对象的继承机制来实现泛化关系，如在Java语言中使用extends关键字、在C++/C#中使用冒号“：”来实现。例如：Student类和Teacher类都是Person类的子类，Student类和Teacher类继承了Person类的属性和方法，Person类的属性包含姓名(name)和年龄(age)，每一个Student和Teacher也都具有这两个属性，另外Student类增加了属性学号(studentNo)，Teacher类增加了属性教师编号(teacherNo)，Person类的方法包括行走move()和说话say()，Student类和Teacher类继承了这两个方法，而且Student类还新增方法study()，Teacher类还新增方法teach()。如图2所示：



|  |
| --- |
| //父类  public class Person {  protected String name;  protected int age;  public void move() {  ……  }  public void say() {  ……  }  }  //子类  public class Student extends Person {  private String studentNo;  public void study() {  ……  }  }  //子类  public class Teacher extends Person {  private String teacherNo;  public void teach() {  ……  }  } |

### 接口与实现关系

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(Realization)关系，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4所示：



|  |
| --- |
| public interface Vehicle {  public void move();  }  public class Ship implements Vehicle {  public void move() {  ……  }  }  public class Car implements Vehicle {  public void move() {  ……  }  } |

# 对象图（Object Diagram）

### 概念

对象：类（Class）的实例（Instance）就是对象（Object）。

类图代表了一类事物，那么对象图就代表着某个具体的事物。

对象图应用比较少，一般使用类图表示。

### 对象图组成元素

类图和对象图的表示方式区别



### 对象图实例



“A:公司”表示对象A是公司这类的实例，如果是：“:公司”则表示这是公司这类的实例，但是没有具体的名称。

# 构件图（Component Diagram）

# 部署图（Deployment Diagram）

# 包图（Package Diagram）

# 活动图（Activity Diagram）

## 概念

※ 描述系统的动态行为。

※ 包含活动状态(ActionState)，活动状态是指业务用例的一个执行步骤或一个操作，不是普通对象的状态。

※ 活动图适合描述在没有外部事件触发的情况下的系统内部的逻辑执行过程；否则，状态图更容易描述。

※ 类似于传统意义上的流程图。

※ 活动图主要用于：

业务建模时，用于详述业务用例，描述一项业务的执行过程；

设计时，描述操作的流程。

## 活动图事物

开始状态(Initial State):实心的原点，表示流程的开始。  
 结束状态（Final State）：空心的原点，表示流程的结束。  
活动（Activity）：圆角矩形，流程中的一个步骤。  
判断（Decision）：菱形，从菱形开始的分支，叫做判断。

合并（Merge）：菱形，分支流程汇聚到一个菱形上。  
监护（Guard）：  
泳道（Swimlane）：垂直泳道和水平泳道

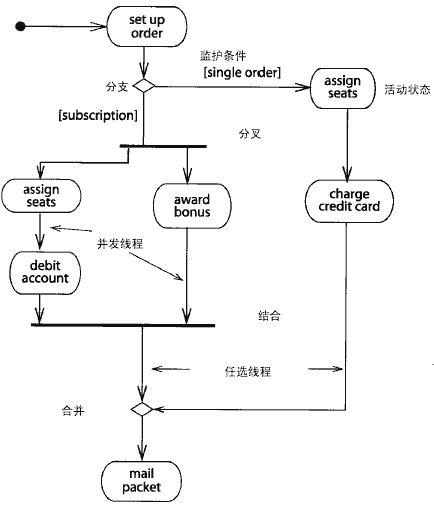
|  |  |  |
| --- | --- | --- |
| 活动 (ActionState) | 动作的执行 |  |
| 起点 (InitialState) | 活动图的开始 |  |
| 终点(FinalState) | 活动图的终点 |  |
| 对象流(ObjectFlowState) | 活动之间的交换的信息 |  |
| 发送信号(signalSending) | 活动过程中发送事件，触发另一活动流程 |  |
| 接收信号(SignalReceipt) | 活动过程中接收事件，接收到信号的活动流程开始执行 |  |
| 泳道(SwimLane) | 活动的负责者 |  |

## 活动图关系

|  |  |  |
| --- | --- | --- |
| 迁移(transition) | 活动的完成与新活动的开始 |  |
| 分支(junction point) | 根据条件，控制执行方向 |  |
| 分叉(fork) | 以下的活动可并发执行 |  |
| 结合(join) | 以上的并发活动再此结合 |  |

## 活动图实例

### 一般活动图





### 泳道活动图

# 状态机图（State Machine Diagram）

# 顺序图（Sequence Diagram）

# 通信图（Communication Diagram）

# 用例图（Use Case Diagram）

概念

用例图是被称为参与者的外部用户所能观察到的系统功能的模型图。 (《UML参考手册》)

用例图列出系统中的用例和系统外的参与者，并显示哪个参与者参与了哪个用例的执行

(或称为发起了哪个用例)。

用例图多用于静态建模阶段(主要是业务建模和需求建模)。

# 时序图（Timing Diagram）

# 总结