用例图

1. 参与者(Actor)

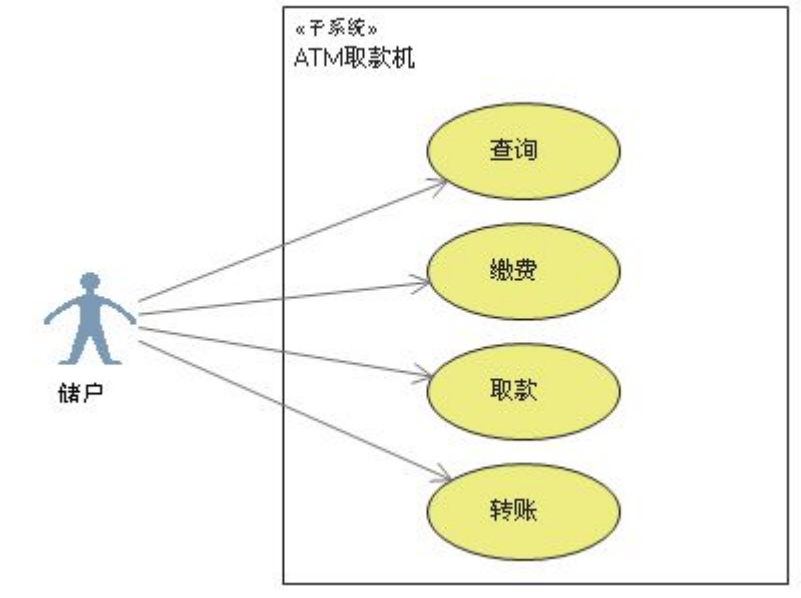
表示与您的应用程序或系统进行交互的用户、组织或外部系统。用一个小人表示。

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)

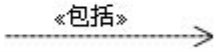
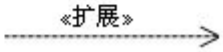
用来展示系统的一部分功能，这部分功能联系紧密。



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：

关系类型	说明	表示符号
关联	参与者与用例间的关系	
泛化	参与者之间或用例之间的关系	
包含	用例之间的关系	
扩展	用例之间的关系	

a. 关联(Association)

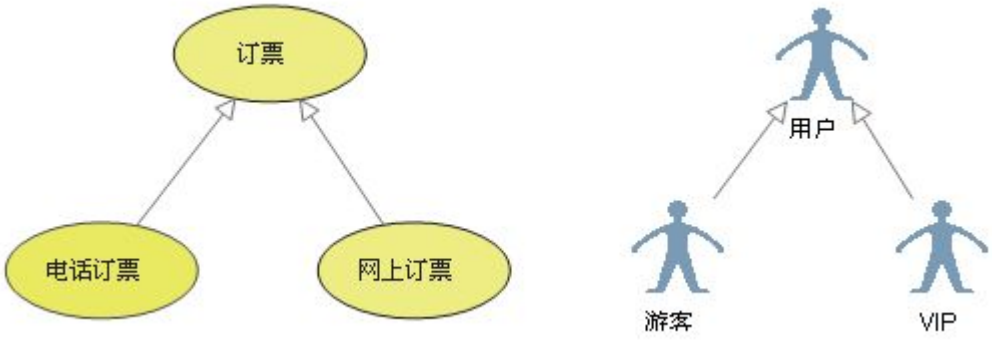
表示参与者与用例之间的通信，任何一方都可发送或接受消息。

【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。

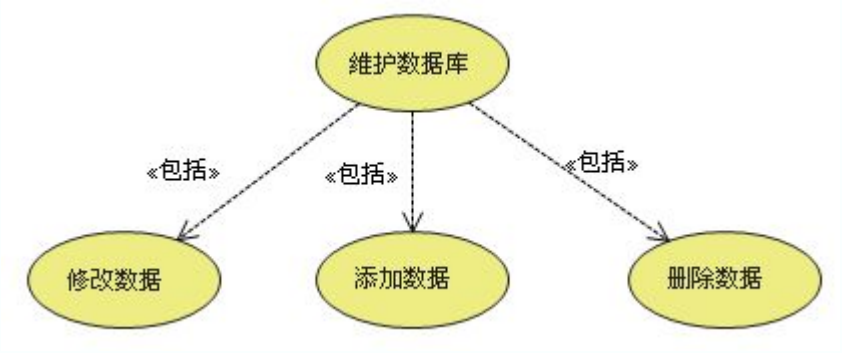
【箭头指向】：指向父用例



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。

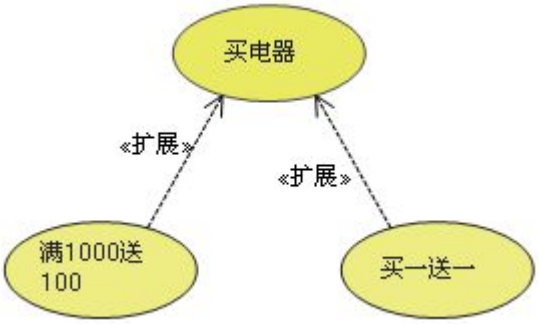
【箭头指向】：指向分解出来的功能用例



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。

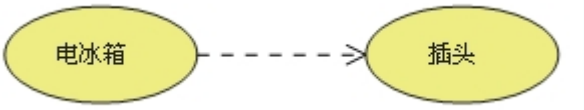
【箭头指向】：指向基础用例



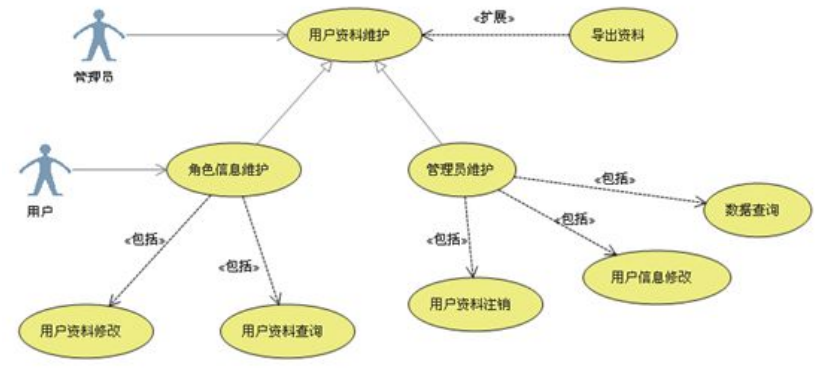
e. 依赖(Dependency)（不考）

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



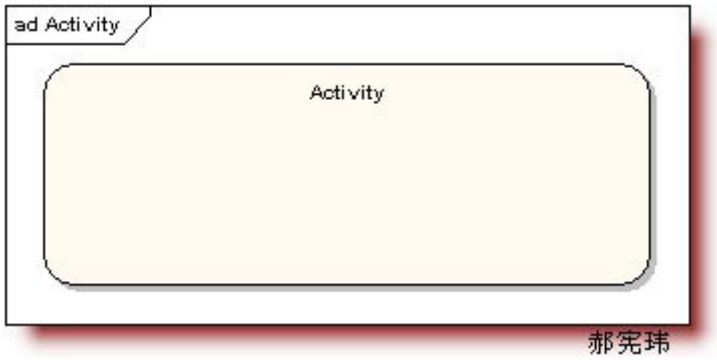
一个用例图示例：



活动图

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。

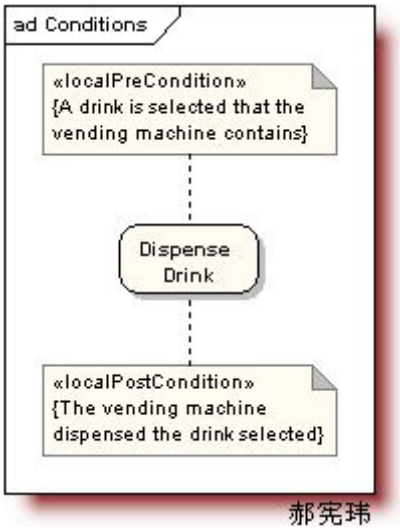


2、动作状态（Actions）

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：

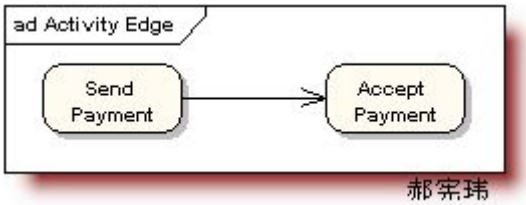
3、动作状态约束（Action Constraints）

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



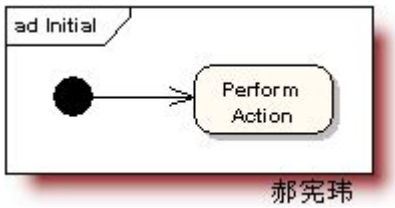
4、动作流（Control Flow）

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。



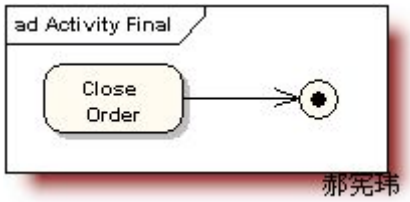
5、开始节点（Initial Node）

开始节点：表示成实心黑色圆点

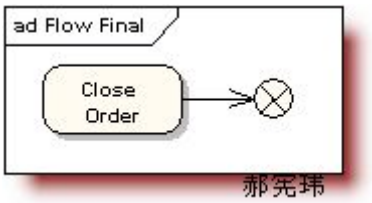


6、终止节点（Final Node）

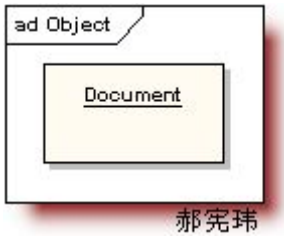
分为活动终止节点（activity final nodes）和流程终止节点（flow final nodes）。
活动终止节点表示整个活动的结束



而流程终止节点表示是子流程的结束。



7、对象（Objects）



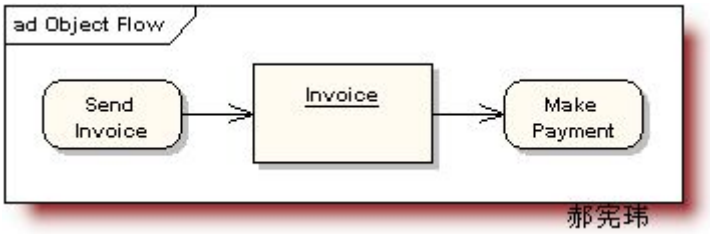
8、数据存储对象（DataStore）

使用关键字«datastore»

9、对象流（Object Flows）

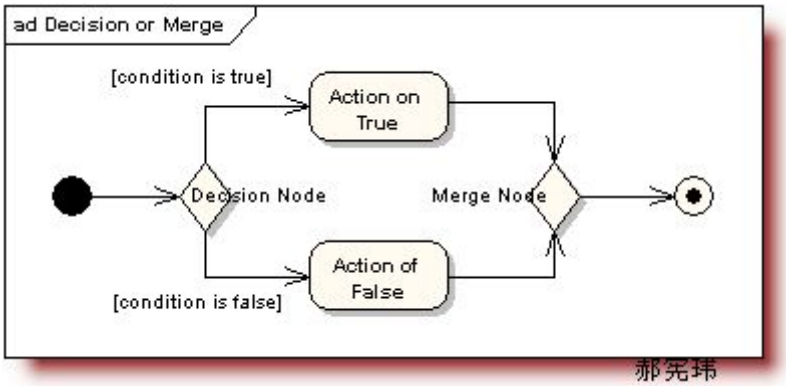
对象流是动作状态或者活动状态与对象之间的依赖关系，表示动作使用对象或动作对对象的影响。用活动图描述某个对象时，可以把涉及到的对象放置在活动图中并用一个依赖将其连接到进行创建、修改和撤销的动作状态或者活动状态上，对象的这种使用方法就构成了对象流。
对象流中的对象有以下特点：
(1)、一个对象可以由多个动作操作。
(2)、一个动作输出的对象可以作为另一个动作输入的对象。
(3)、在活动图中，同一个对象可以多次出现，它的每一次出现表面该对象正处于对象生存期的不同时间点。
对象流用带有箭头的虚线表示。如果箭头是从动作状态出发指向对象，则表示动作对对象施加了一定的影响。施加的影响包括创建、修改和撤销等。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。



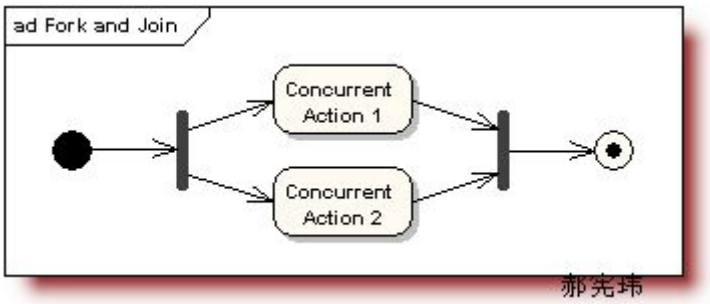
10、分支与合并（Decision and Merge Nodes）

分支与合并用菱形表示



11、分叉与汇合（Fork and Join Nodes）

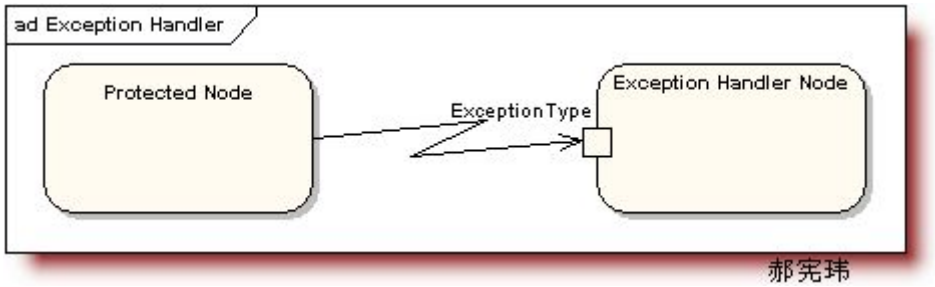
分为水平风向和垂直方向。



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。

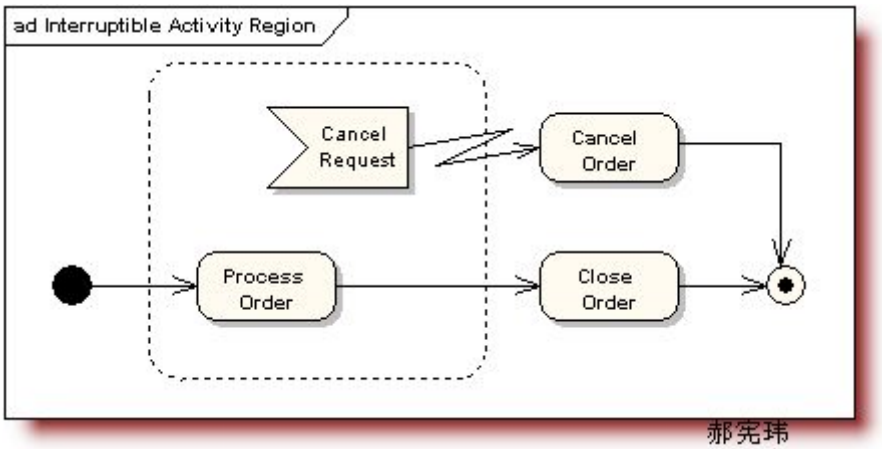
12、异常处理（Exception Handler）

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）

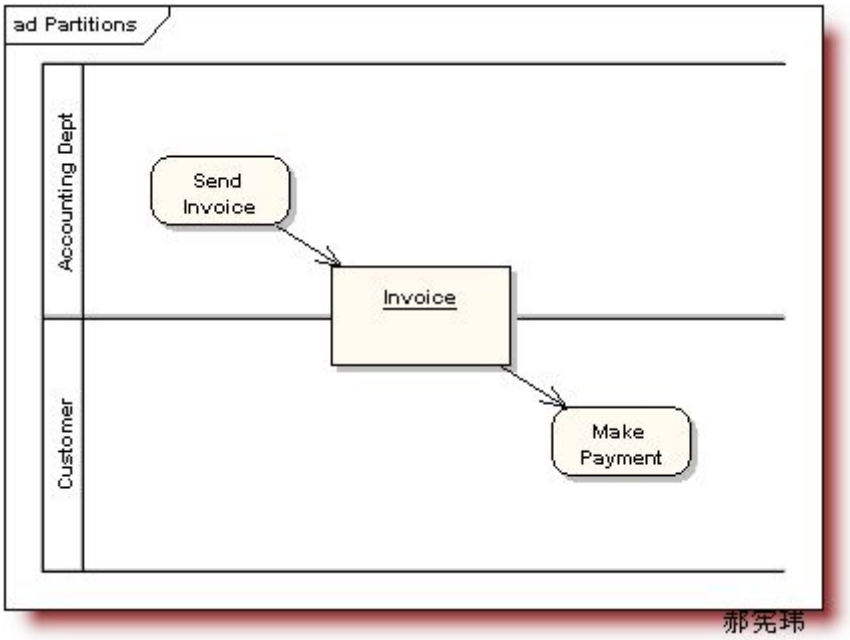
活动中断区域围绕一些可被中断的动作状态图。比如下图，正常情况下【Process Order】顺序流转到【Close Order】，订单处理流程完毕；但在【Process Order】过称中，会发送【Cancel Order】请求，这时会流转到【Cancel Order】，从而订单处理流程结束



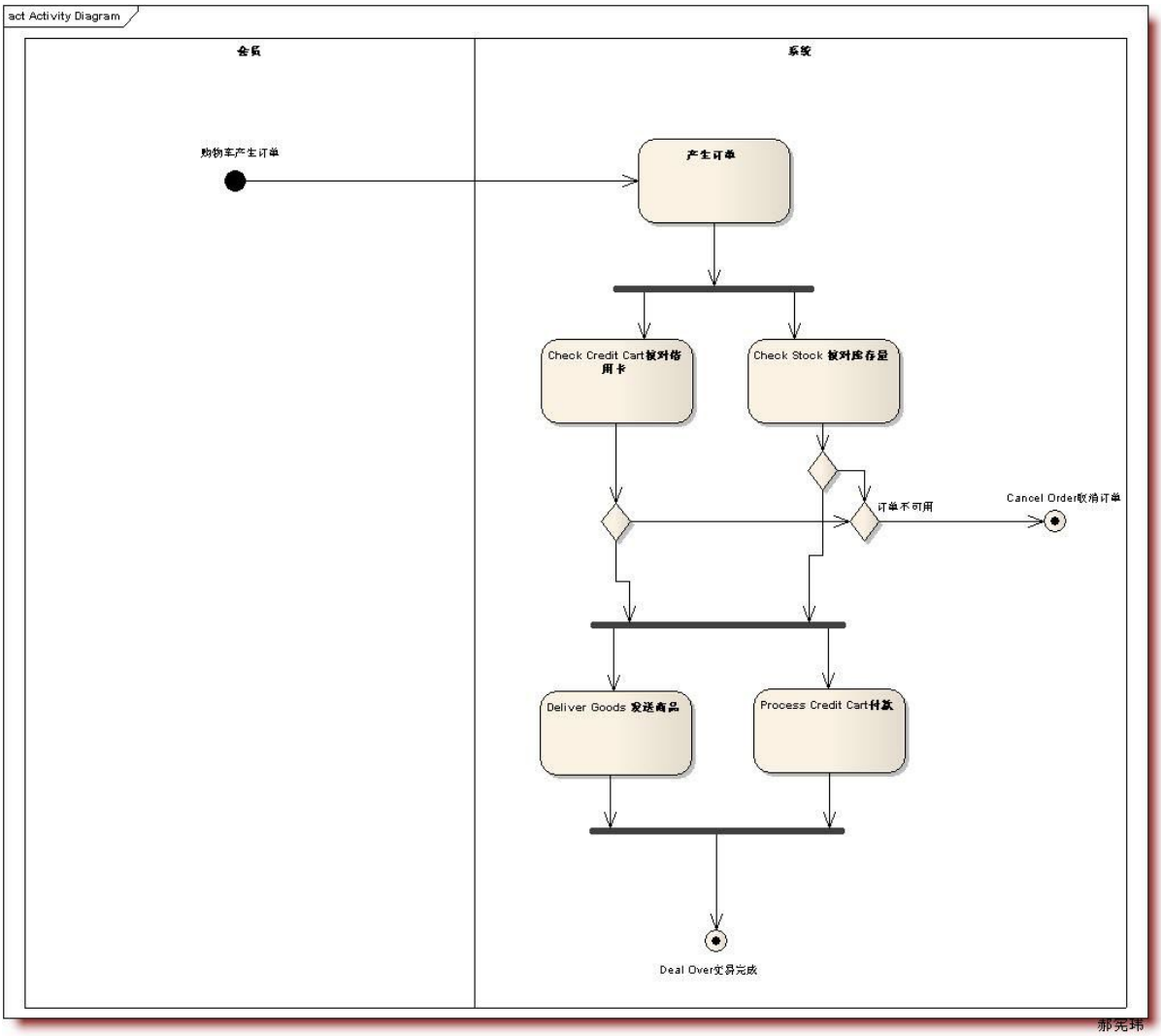
14、泳道（Partition）

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



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

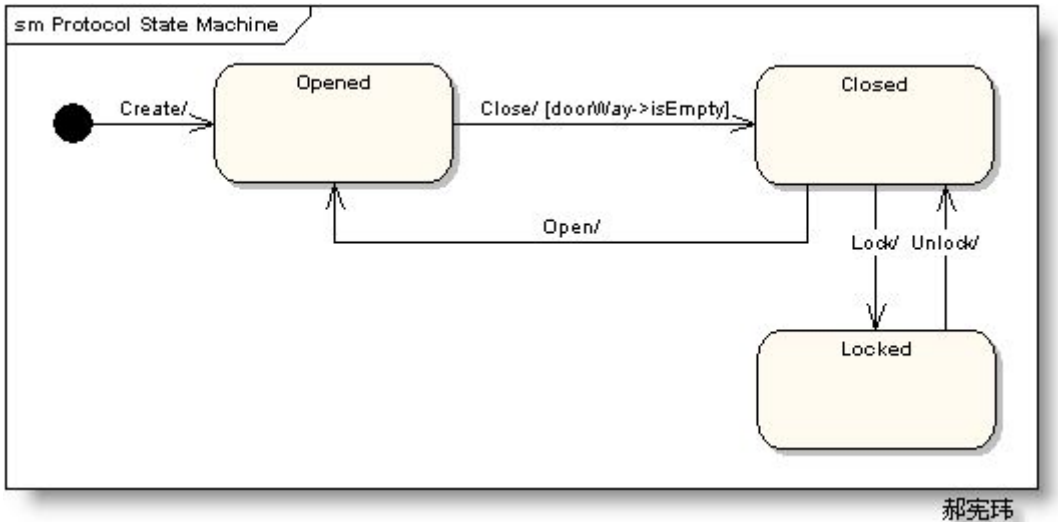
5、 分叉与汇合：【产生订单】份叉为检查库存量和会员支付金额是否足够，如果不足，取消订单，如过库存量和支付金额足够，发送商品和付款，最后汇合为订单完成。

状态图

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。

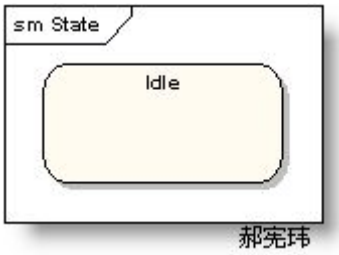
2、 转移（Transition）有警备条件（guard condition），比如只有 doorWay->isEmpty 条件满足时，才会响应事件。

二、状态图元素（State Diagram Elements）

1、状态（States）

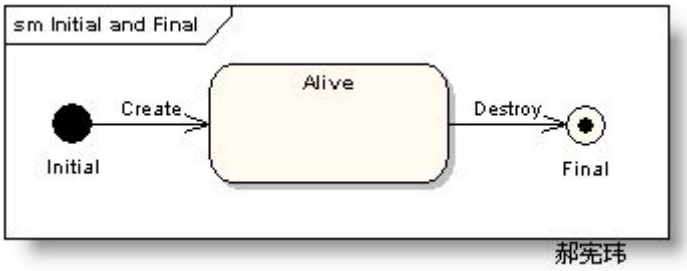
指在对象的生命周期中的某个条件或者状况，在此期间对象将满足某些条件、执行某些活动活活等待某些事件。所有对象都有状态，状态是对象执行了一系列活动的结果，当某个事件发生后，对象的状态将发生变化。

状态用圆角矩形表示



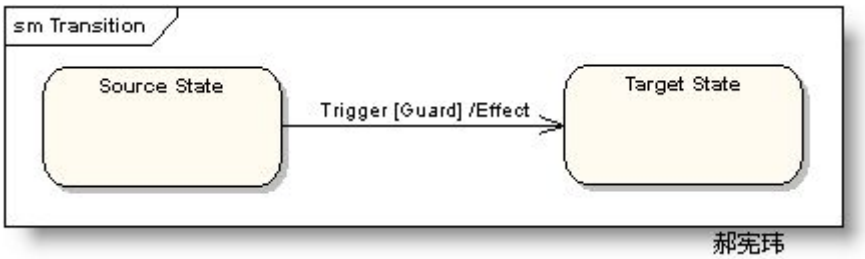
初态和终态（Initial and Final States）

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。

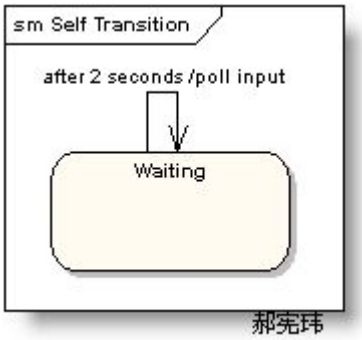
3、动作（State Actions）

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）

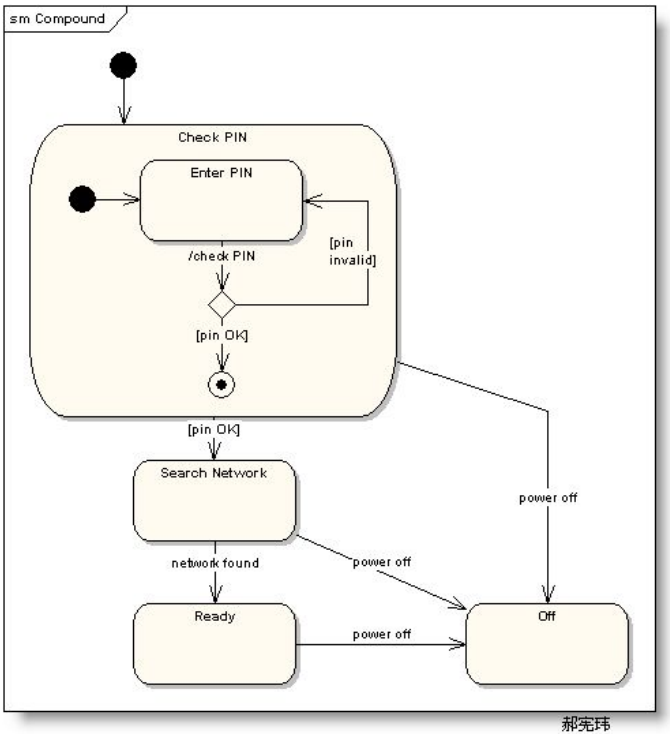
状态可以有返回自身状态的转移，称之为自身转移（Self-Transitions）



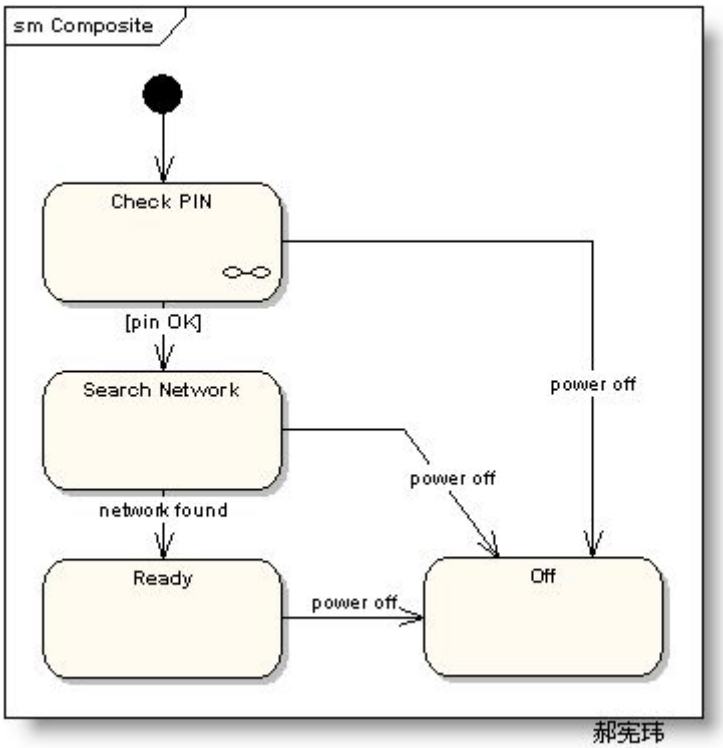
2S 后，Poll input 事件执行，转移到自己状态【Waiting】

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。



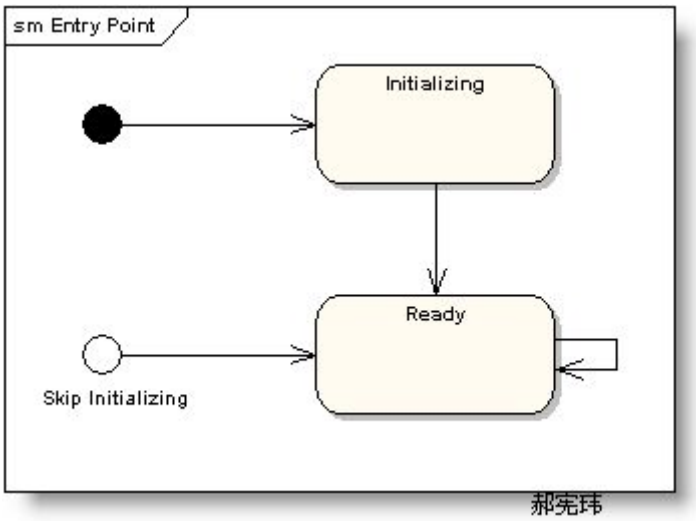
也可用以下方式进行描述



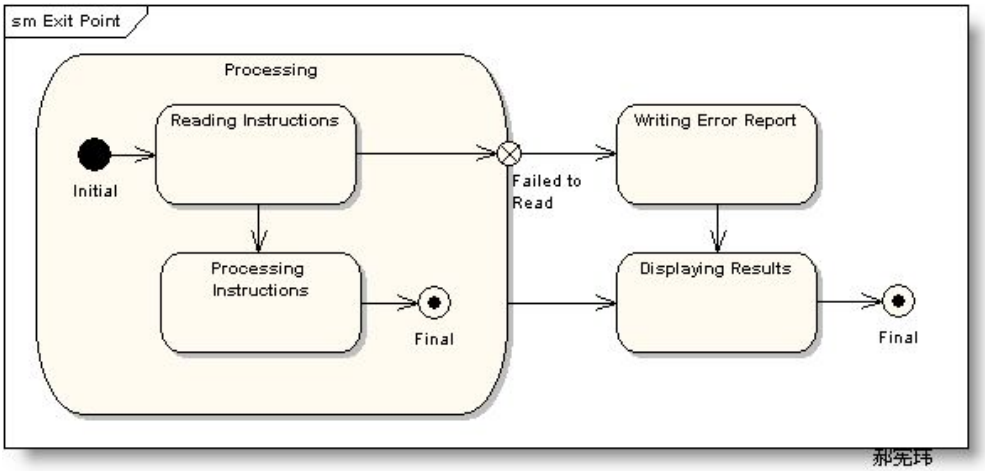
如上图，状态机【Check PIN】的细节被分割到另外一个图中了。

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）

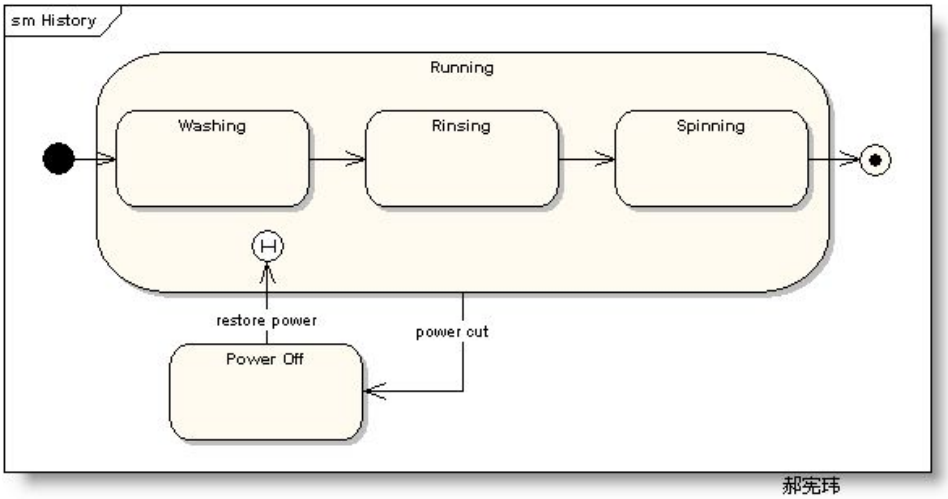


7、退出节点（Exit Point）



8、历史状态（History States）

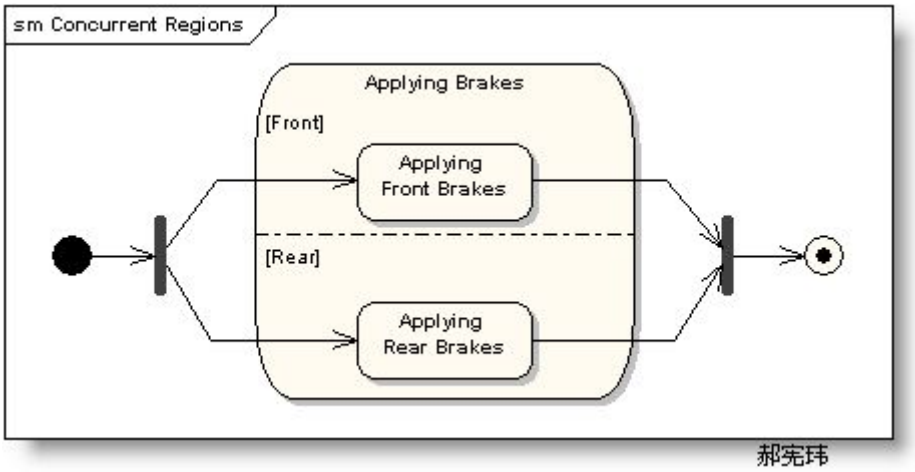
历史状态是一个伪状态（Pseudostate）,其目的是记住从组合状态中退出时所处的子状态，当再次进入组合状态，可直接进入这个子状态，而不是再次从组合状态的初态开始。



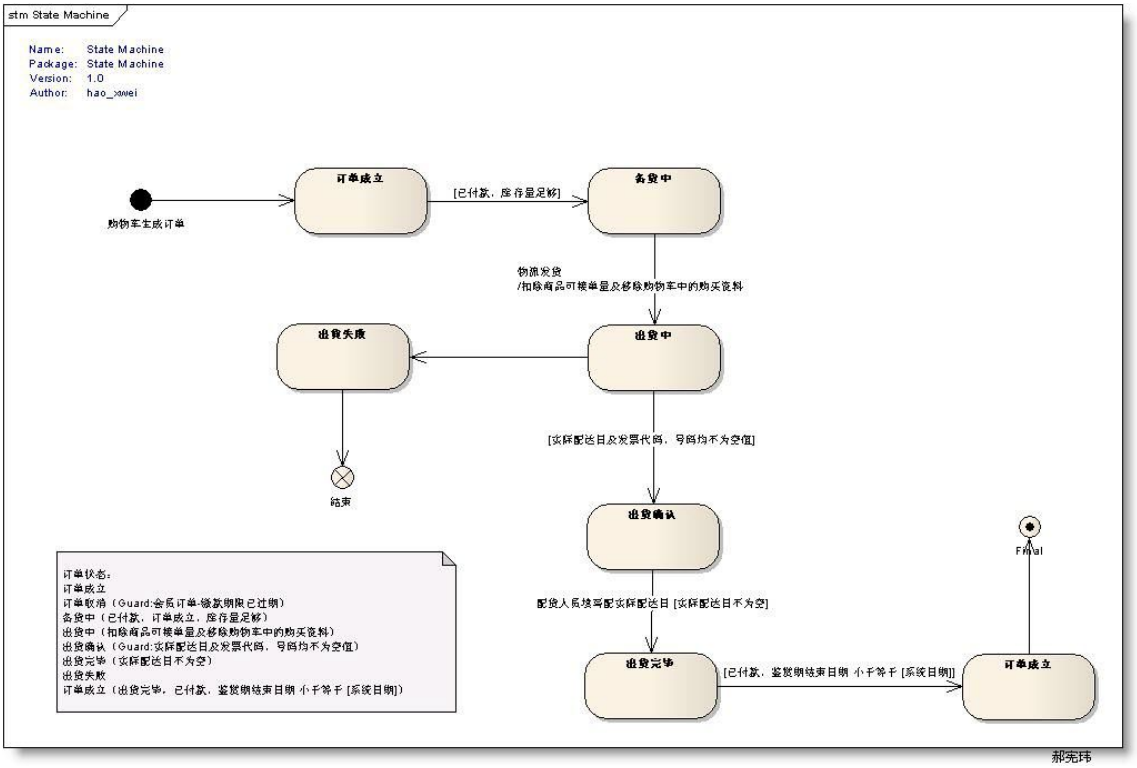
在上图的状态图中，正常的状态顺序是:【Washing】- >【Rinsing】->【Spinning】。
如果是从状态【Rinsing】突然停电（Power Cut）退出 ,洗衣机停止工作进入状态【Power Off】,当电力恢复时直接进入状态【Running】。

9、并发区域（Concurrent Regions）

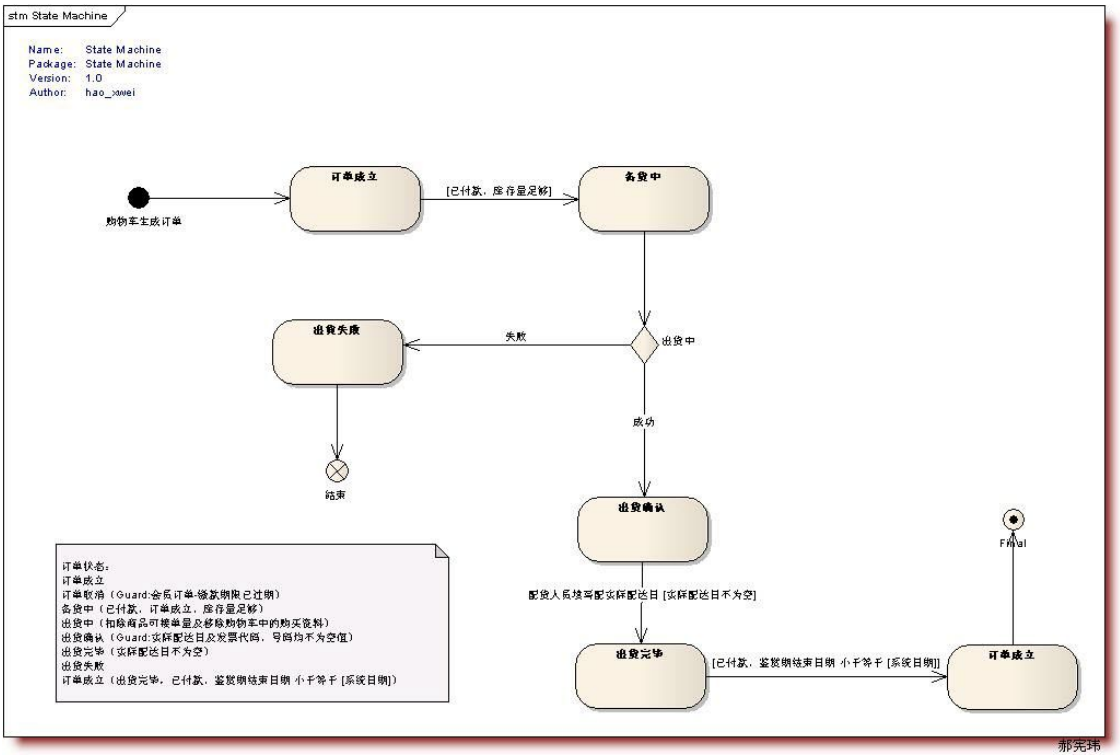
状态图可以分为区域，而区域又包括退出或者当前执行的子状态。说明组合状态在某一时刻可以同时达到多个子状态。如下图刹车系统，同时进入前刹车【Applying Front Brakes】状态和后刹车【Applying Rear Brakes】状态。



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]）

状态图和活动图的区别

状态图是描述某一对象的状态转化的，它主要表现的是该对象的状态。从状态图中可以看出，该对象在接受了外界的某种刺激之后，会做出什么样的反应。描述的是一个对象的事情。可以说是对类图的一种补充，帮助开发者完善某一类。

活动图是描述系统在执行某一用例时的具体步骤的，它主要表现的是系统的动作。从活动图中可以看出，系统是如何一步一步的完成用例规约的，主要用于业务建模阶段。活动图描述的是整个系统的事情。可以说活动图是对用例图的一种细化，帮助开发者理解业务领域。

比如说：
学校用的学生选课系统。

在系统中，学生是一个对象（UML 中的对象，不是编程语言中的对象），那么学生“未登录”“已登录”“未完成选课”“已完成选课”“已选 XX 课”“未选 XX 课”等都是学生的状态。描述这些状态之间是如何转化的，就要用状态图。
而学生选课的这个动作涉及到：学生、课程、教师、学生课表等多个对象。同时这个动作也是学生选课系统的一个用例，所以要描述它就要用到活动图。

顺序图（时序图）

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。

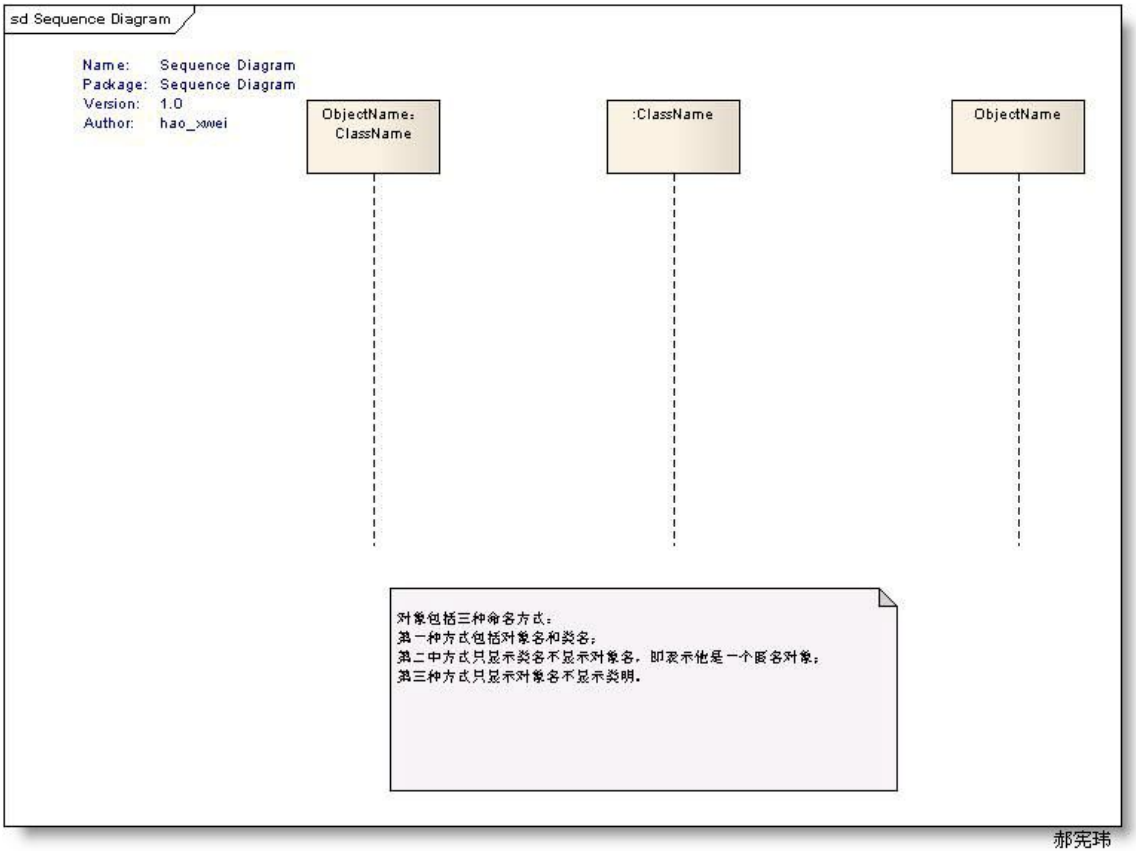
二、时序图元素（Sequence Diagram Elements）

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。

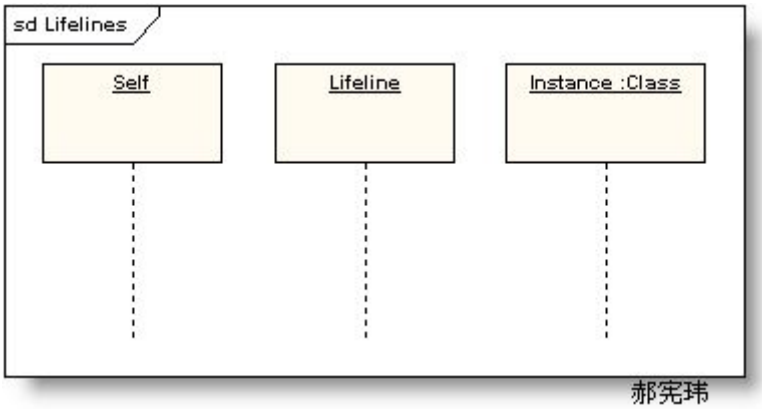
对象（Object）

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。



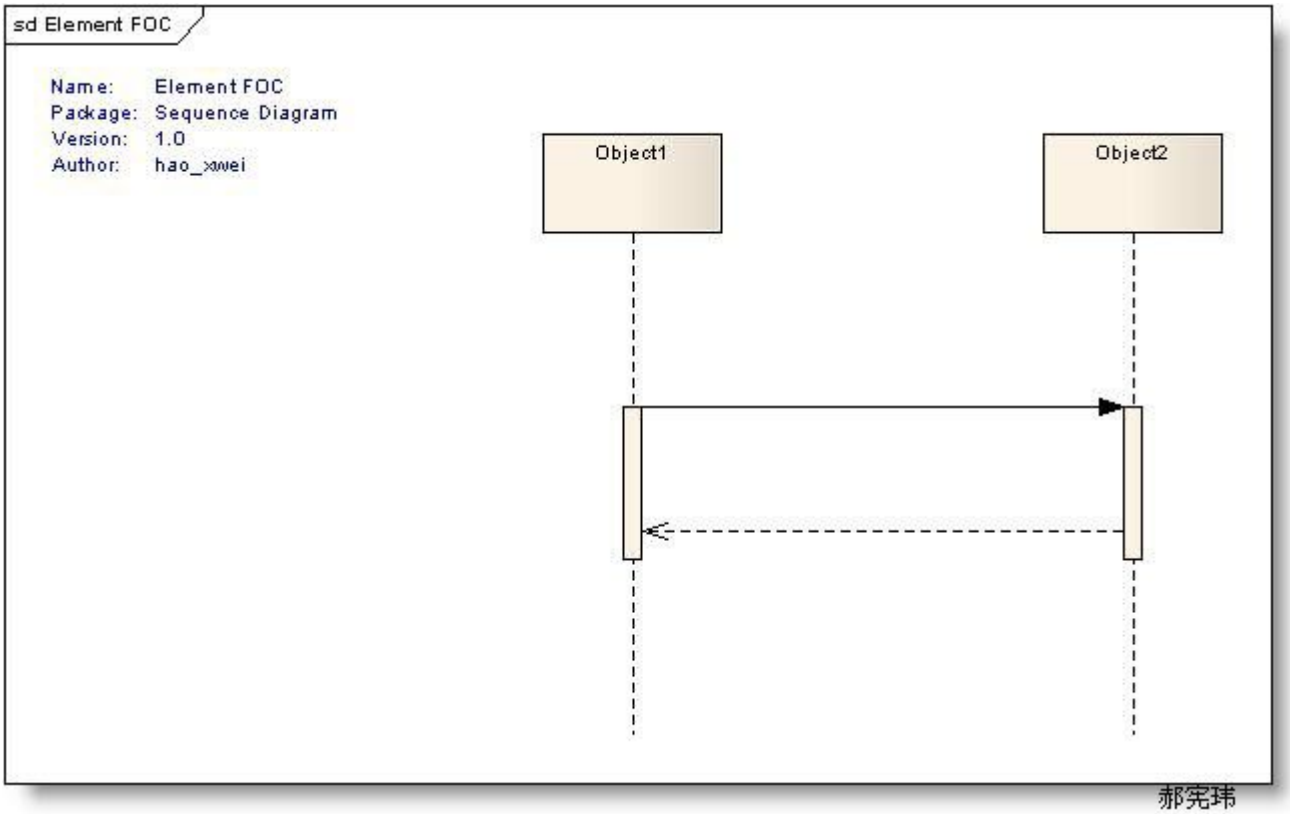
生命线（Lifeline）

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



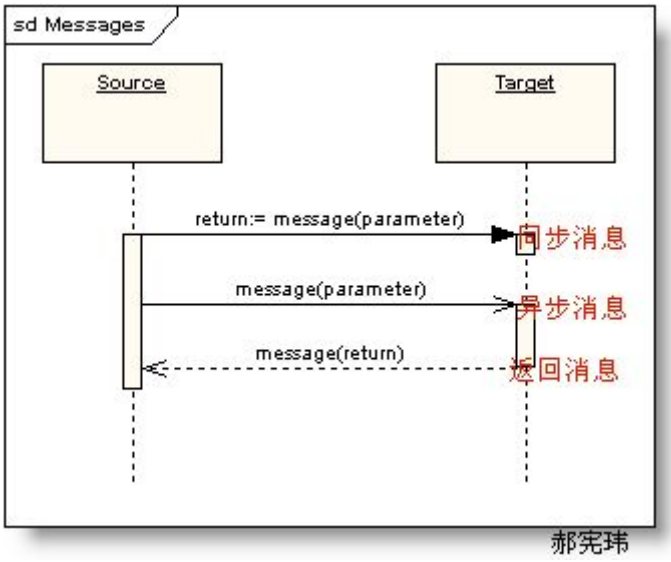
控制焦点（Focus of Control）

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）

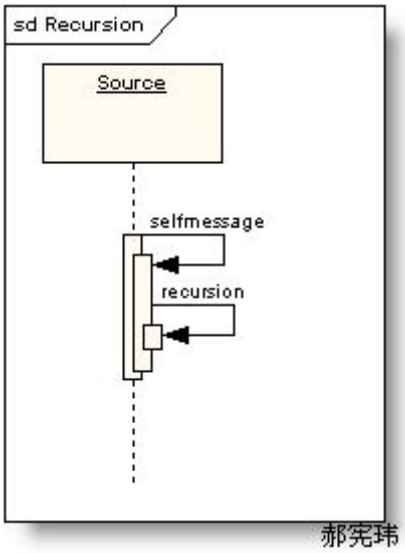
消息发送者通过消息把信号传递给消息的接收者，然后继续自己的活动，不等待接受者返回消息或者控制。异步消息的接收者和发送者是并发工作的。

返回消息（Return Message）

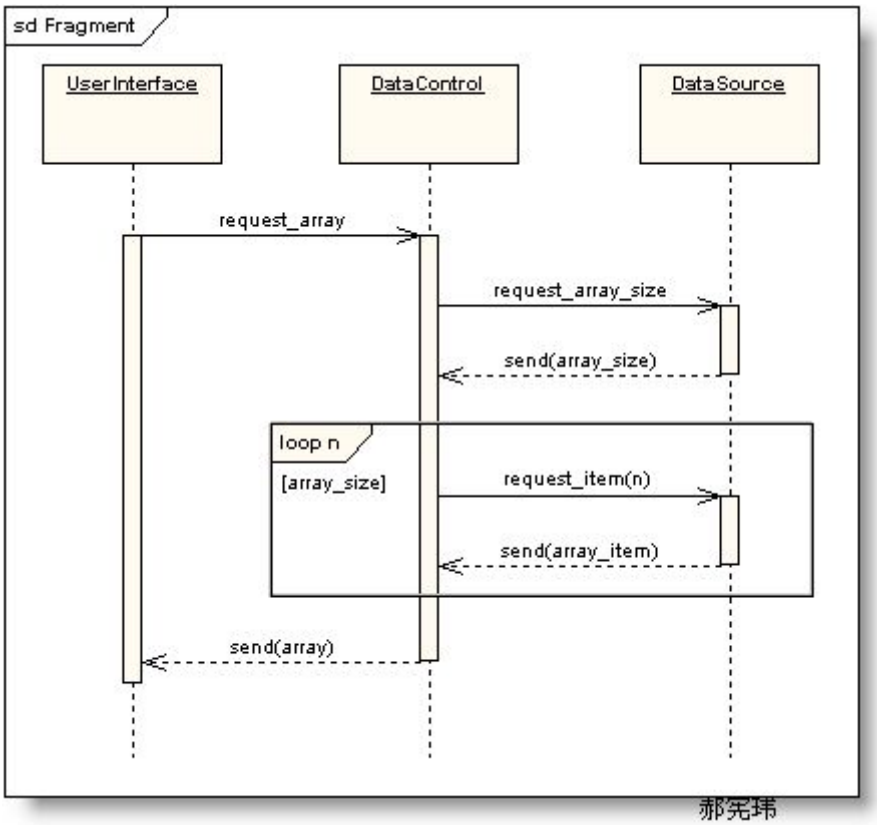
返回消息表示从过程调用返回

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



- Alternative fragment（denoted “alt”） 与 if…then…else 对应
- Option fragment (denoted “opt”) 与 Switch 对应
- Parallel fragment (denoted “par”) 表示同时发生
- 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

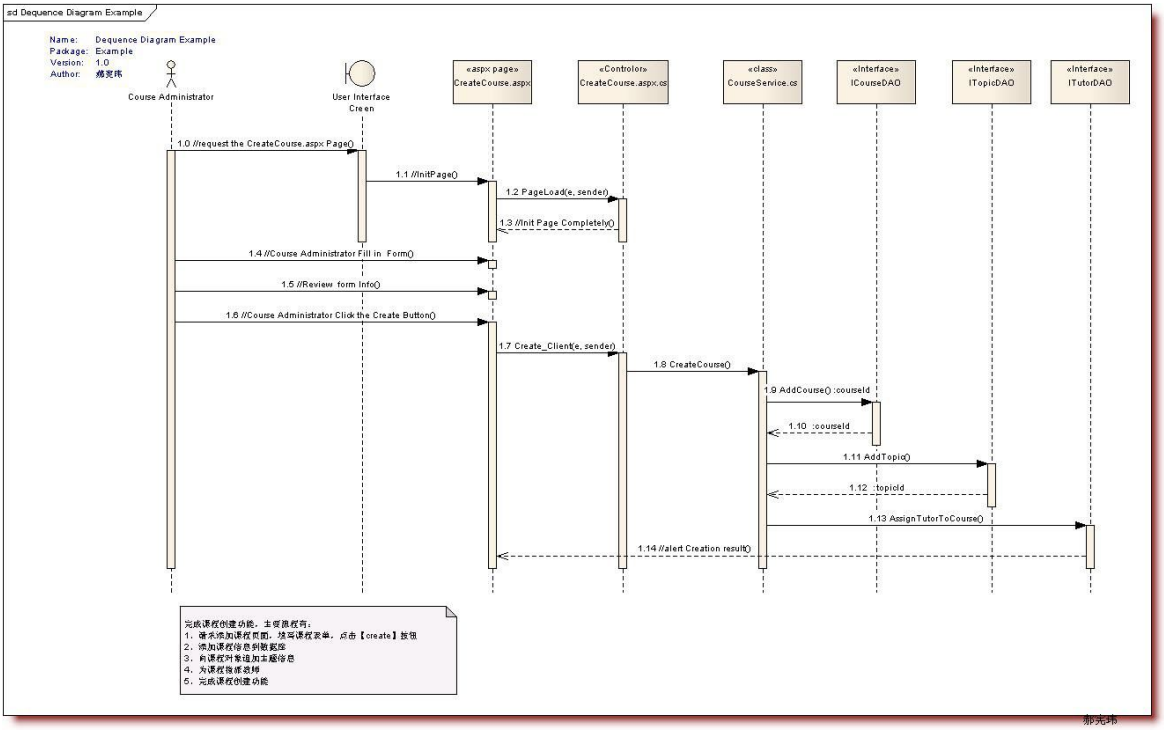
2、添加课程信息到数据库

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

包图（MVC 架构）

在 UML 的建模机制中，模型的组织是通过包来实现的。包把建立的各种模型组织起来，形成各种功能或用途的模块，并可以控制包中元素的可见性以及描述包之间的依赖关系。通过这种方式，系统模型的实现者可在高层把握系统的结构。

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。

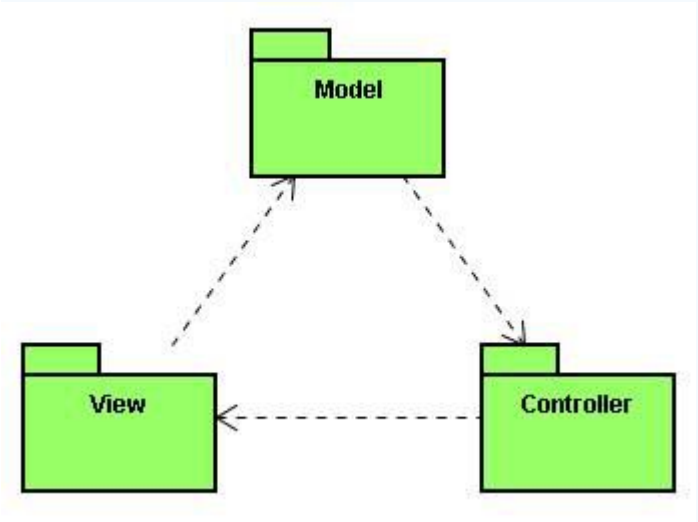
包图由两个矩形表示（单选项卡），包图名称写在大矩形中间。除包名外，还有元素，元素可见性，包构造型及包间的关系组成。

在包下可以创建各种模型元素：类，接口，构件，节点，用例，图以及其它包。

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。

例：



构件图

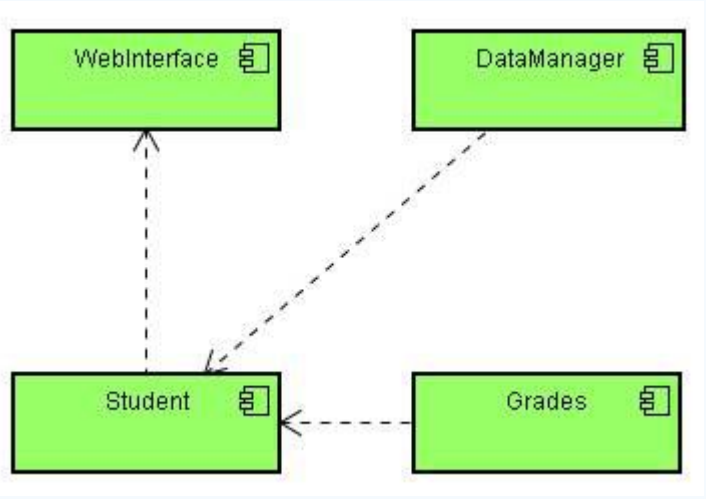
构件图提供系统物理视图，在一个非常高的层次上显示系统中的构件与构件之间的依赖关系。

将系统中可重用的模块封装为具有可替代性的物理单元，就是构件。它是一个系统或子系统中的封装单元，提供一个或多个接口，是系统高层的可重用部件。构件作为系统中的一个物理实现单元，包括：软件代码（源码，二进制代码，可执行文件，脚本，命令行等），带有身份标识并且有物理实体的文件（文档，数据库）。

标准构件用左边有两个小矩形的大矩形表示，构件名在大矩形内部。构件有不同的类型。

构件图是用来表示系统中构件与构件之间，类或接口与构件之间的关系图。其中，构建图之间的关系表现为依赖关系，定义的类或接口与类之间的关系表现为依赖关系或实现关系。

例：（astah 中的构件图与 rational rose2003 中的不同）



部署图

部署图描述了一个系统运行时的硬件节点，在这些节点上运行的软件构件将在何处物理运行以及它们将如何彼此通信的静态视图。部署图包括两种基本模型元素：节点和节点间的连接。每个模型中，仅包含一个部署图。节点包括两种类型：处理器和设备。

处理器指本身具有计算能力且能执行各各软件的节点，如服务器。处理器具有处理能力，所以在描述处理器方面应当包含了处理器的调度和进程。调度指在处理器处理其进程中为实现一定的目的而对共同使用的资源进行时间分配。调度方式包含：抢占，无优先级，循环，算法控制，手动执行。进程表示一个单独的控制纯种，是系统中一个重量级的并发和执行单元。

设备指本身不具备处理能力的节点，如打印机。

连接用来表示两个节点之间的硬件连接。节点之间的连接可以通过光缆直接进行，或通过卫星等方式非直接连接，通常连接都是双向的。连接用实线表示，实线上可加连接名和构造型。

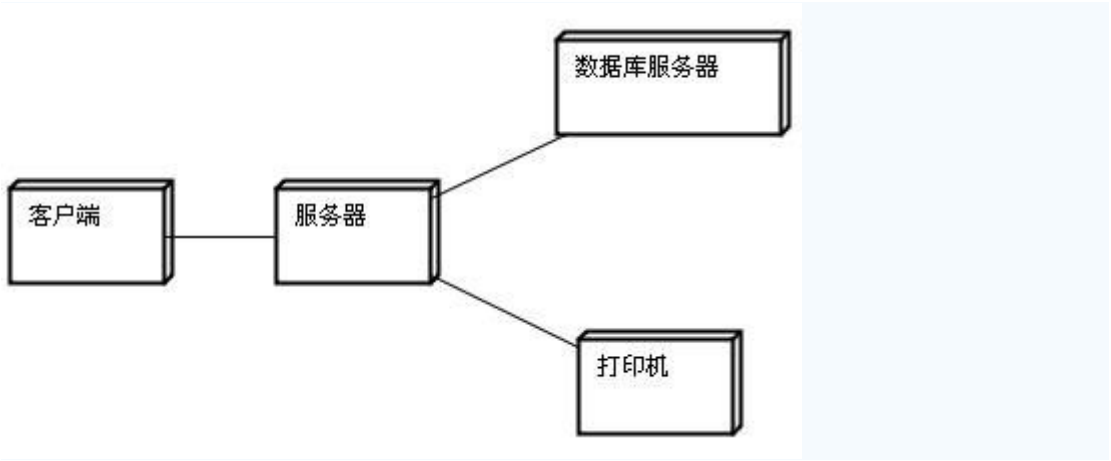
系统开发人员和部署人员可以利用部署图去了解系统的物理运行情况。如果，开发的软件系统只需在一台计算机上运行，且使用的标准设备，则不需要为它画出系统部署图。部署图只需要给那些复杂的物理运行情况进行建模。

部署图显示了系统的硬件，安装在硬件上的软件，用于连接硬件的各种协议和中间件等。

部署模型的目的：

描述一个具体应用的主要部署结构，通过对各种硬件，在硬件中的软件以及各种连接协议的显示，可以很好的描述系统是如何部署的；平衡系统运行时的计算资源分布；可以通过连接描述组织的硬件网络结构或者是嵌入式系统等具有多种硬件和软件相关的系统运行模型。

例：（astah 中部署图）



类图

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)

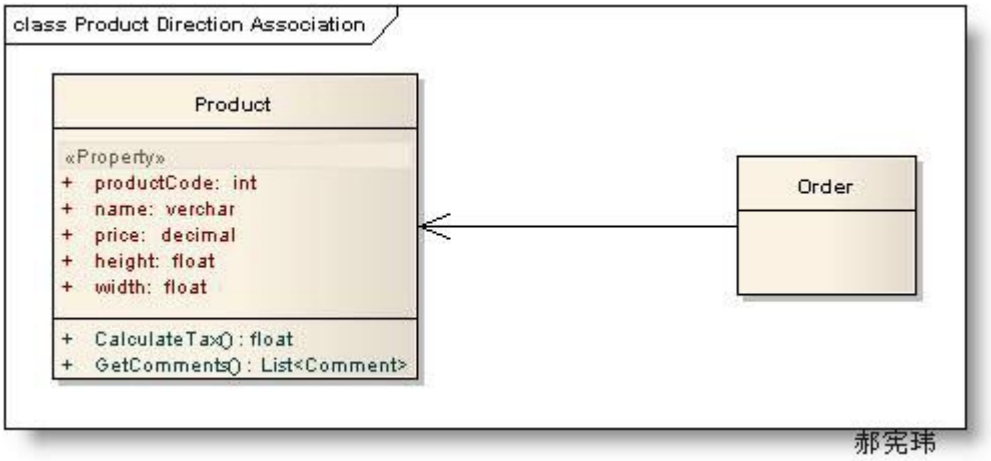
两个相对独立的对象，当一个对象的实例与另外一个对象的特定实例存在固定关系时，这两个对象之间就存在关联关系。

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
{
}
```

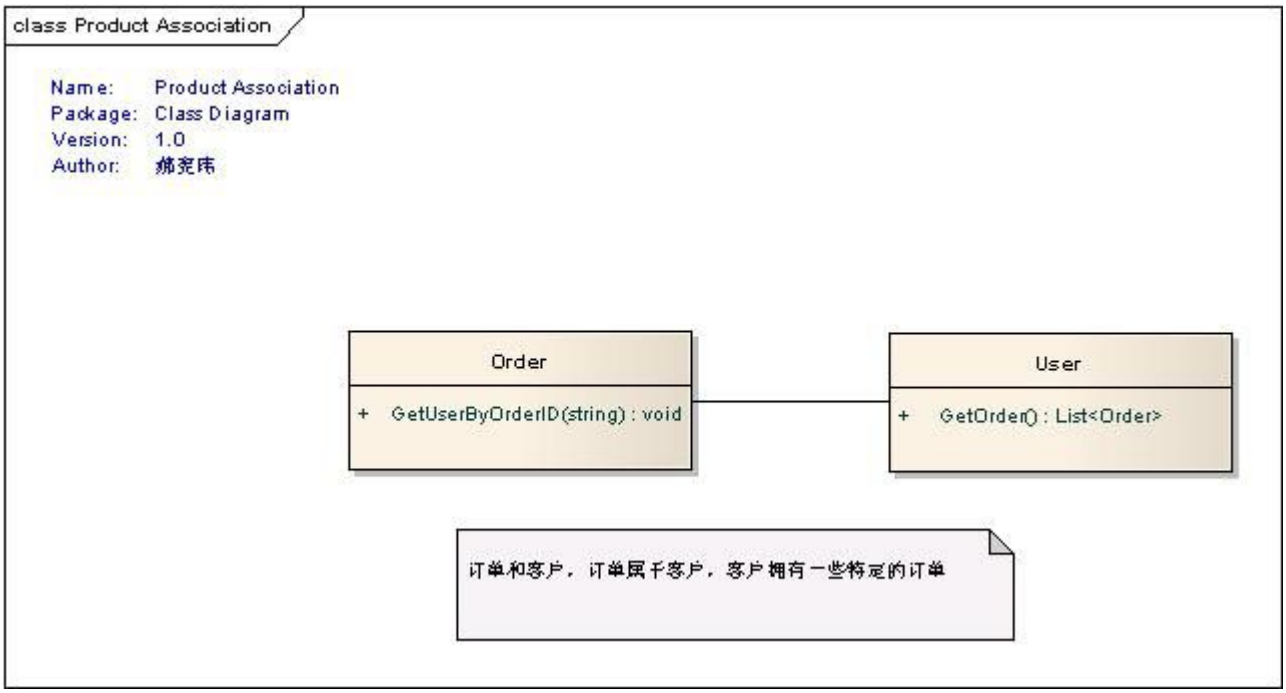
代码表现为：Order(A1)中有 Product(A2)的变量或者引用

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



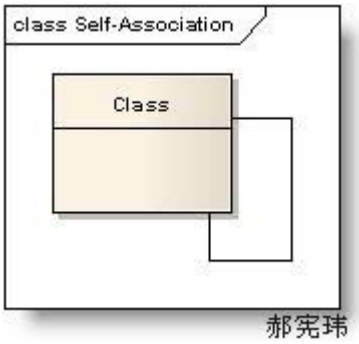
C#代码

```
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

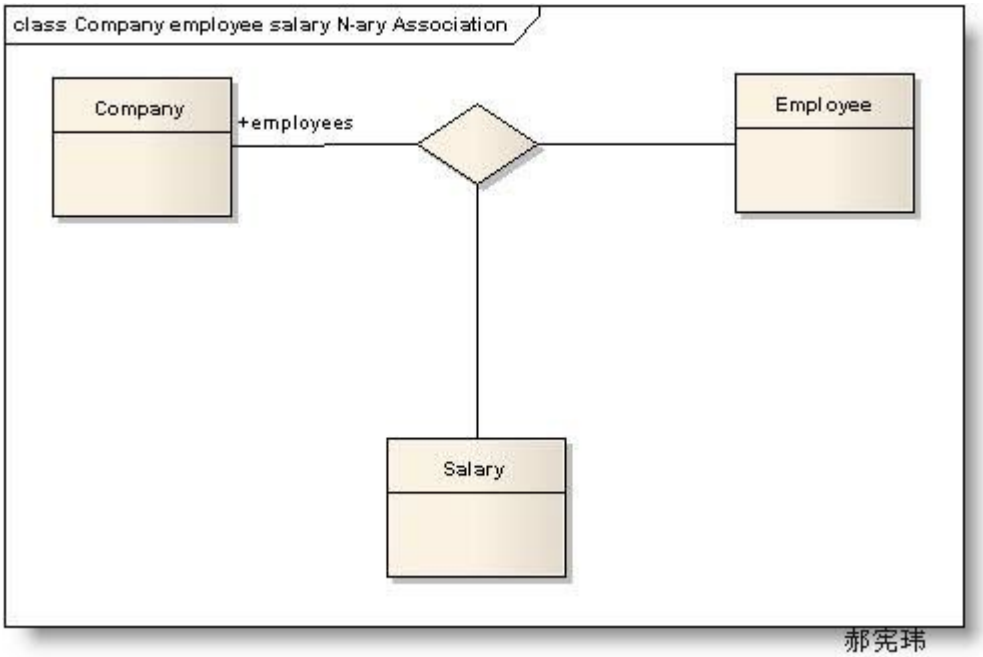


4、多维关联(N-ary Association)

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：

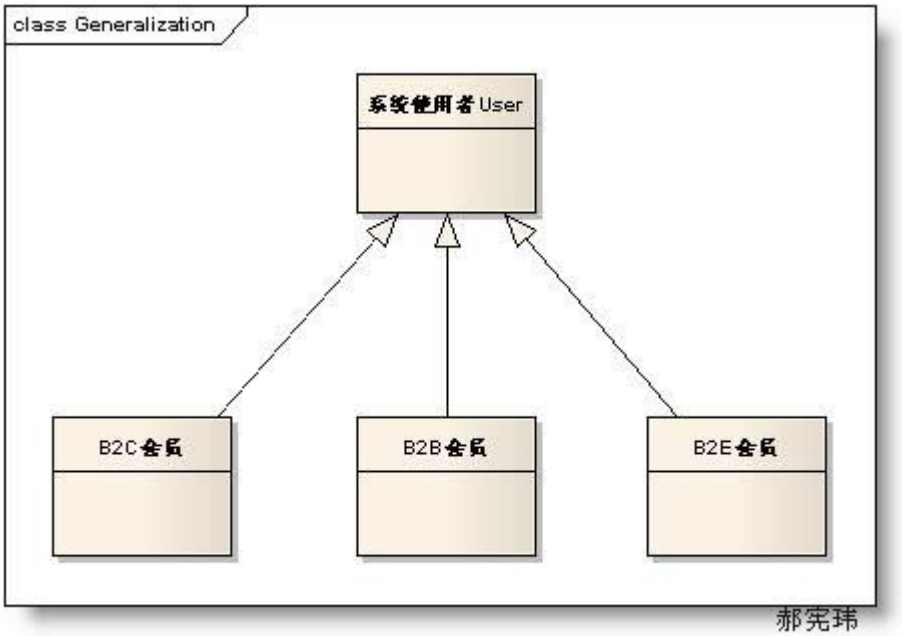


5、泛化(Generalization)

类与类的继承关系，类与接口的实现关系。

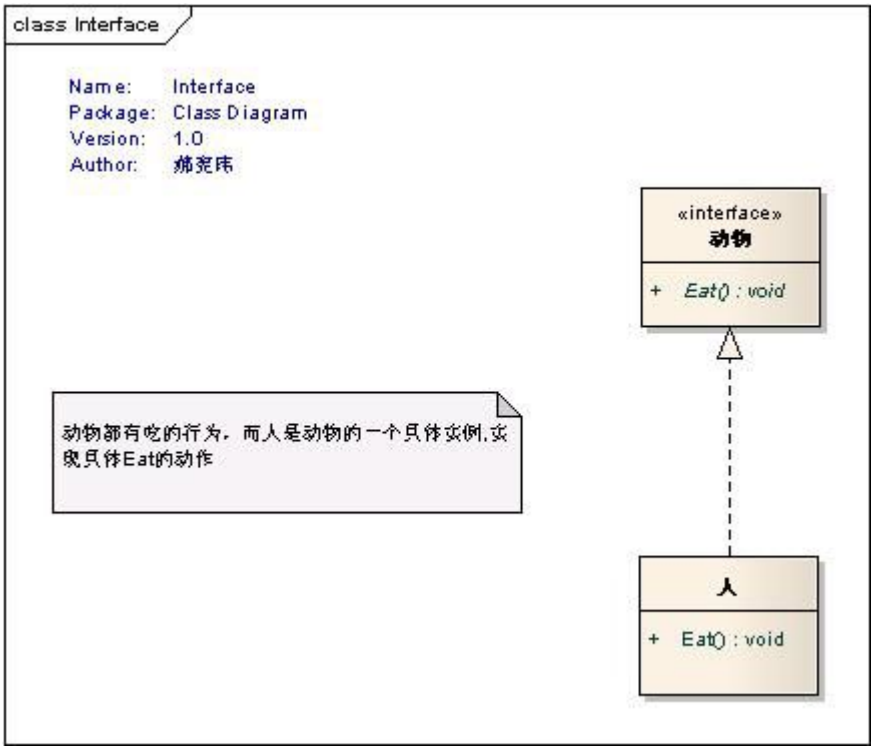
场景：父与子、动物与人、植物与树、系统使用者与 B2C 会员和 B2E 会员的关系

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



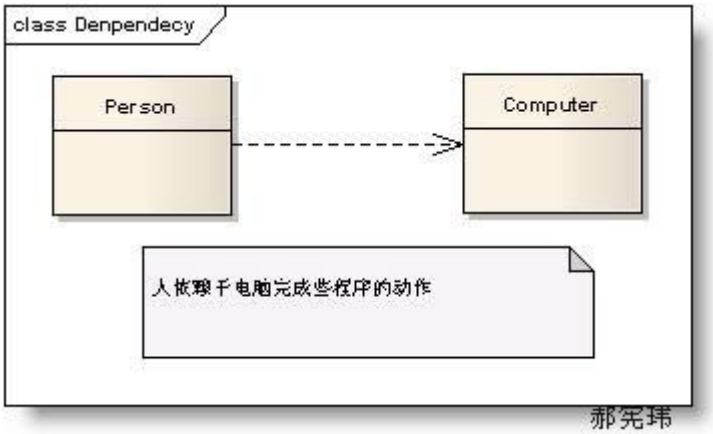
6、依赖(Dependency)

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。

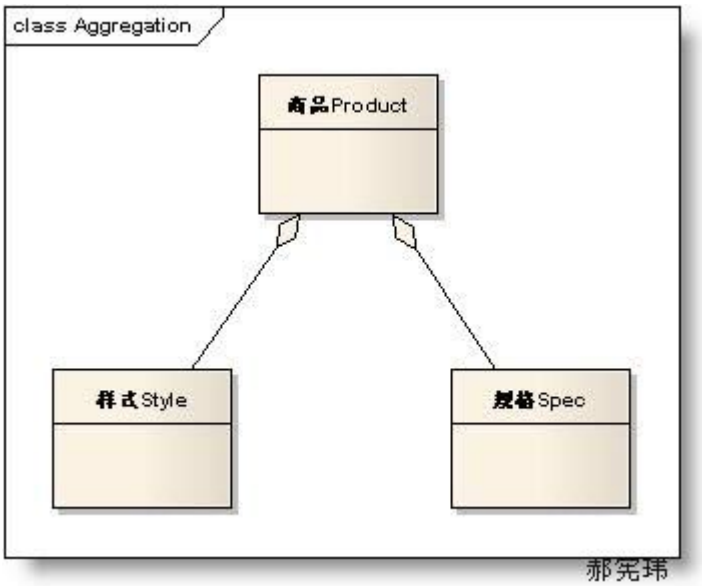


7、聚合(Aggregation)

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

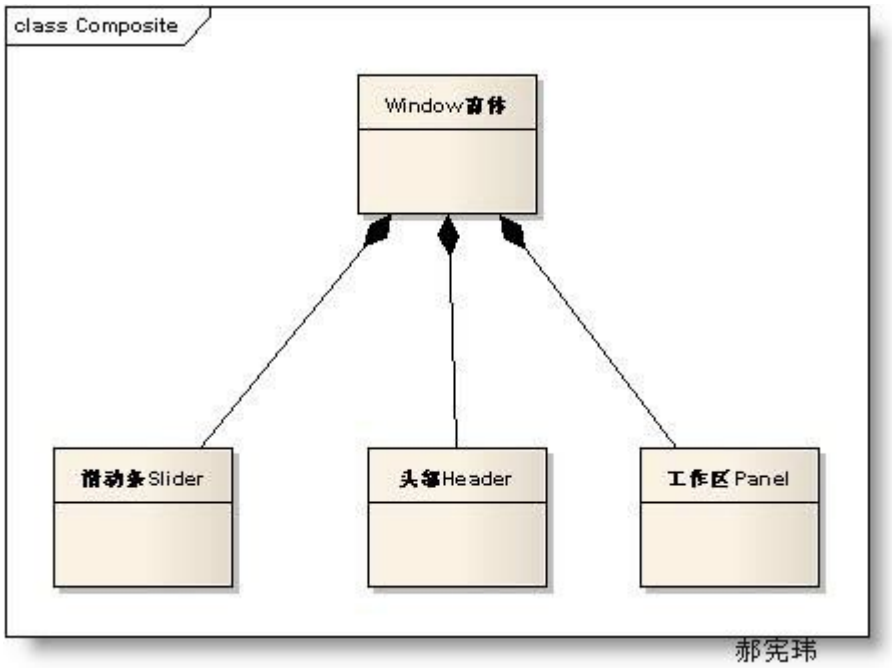


8、组合（Composite）

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



领域模型

用包（Package）组织领域模型

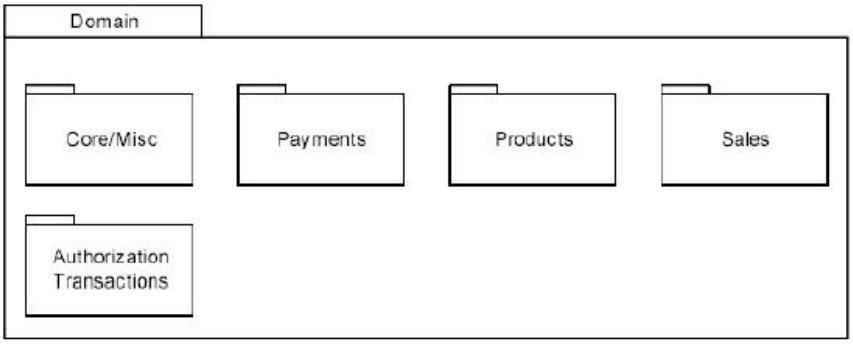
有时候，一个领域模型会变得非常大，这时我们可以把按照相关的概念将其划分成不同的包。

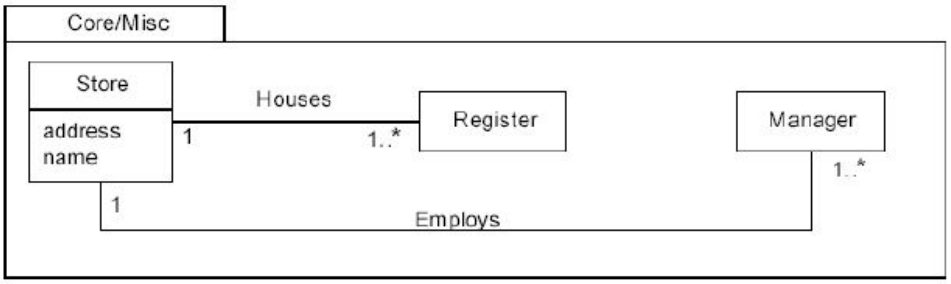
包的划分

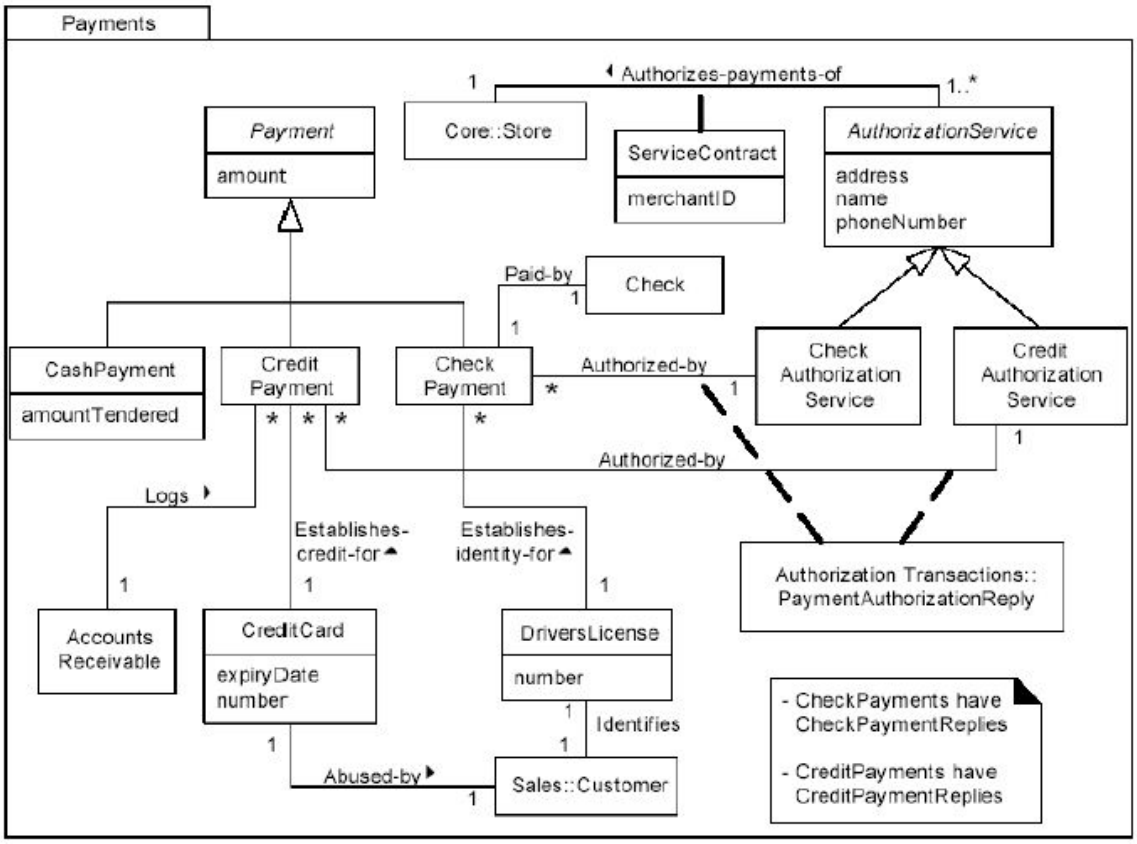
我们可以将如下一些元素组合成一个单独的包：

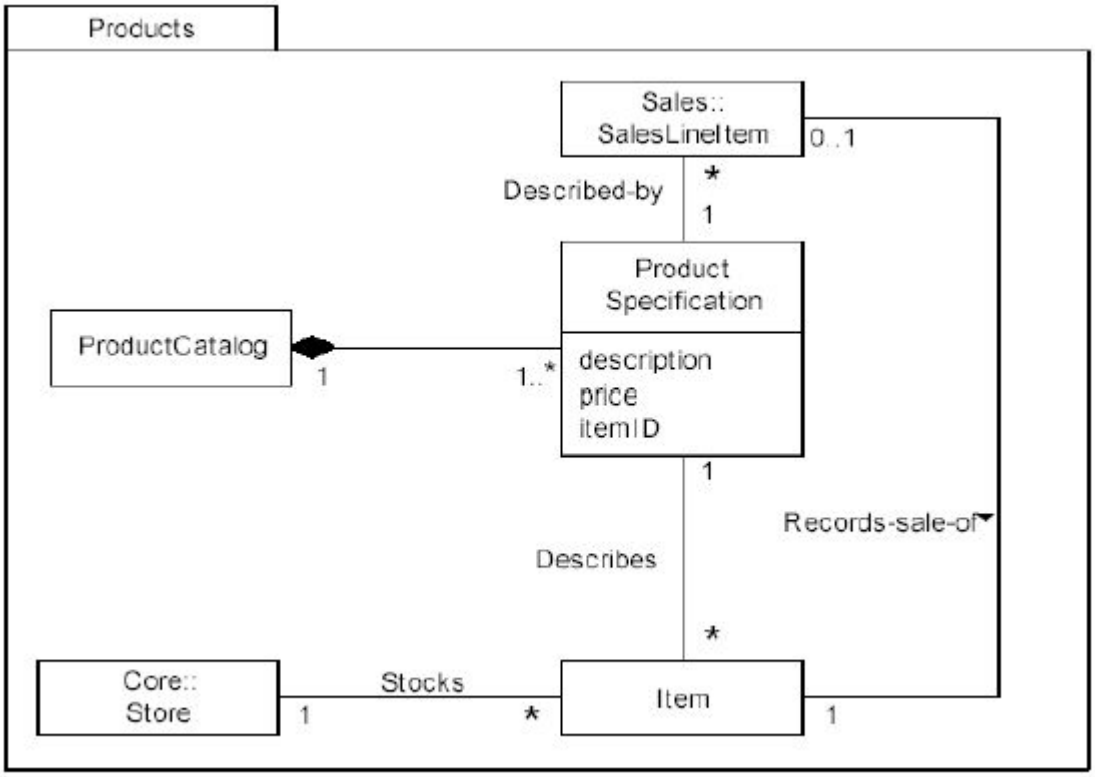
- 在概念或用途上，它们属于同一个主题范围；
- 它们在同一个类层次结构上；
- 参与了同一个用例；
- 之间有非常强烈的关联关系。

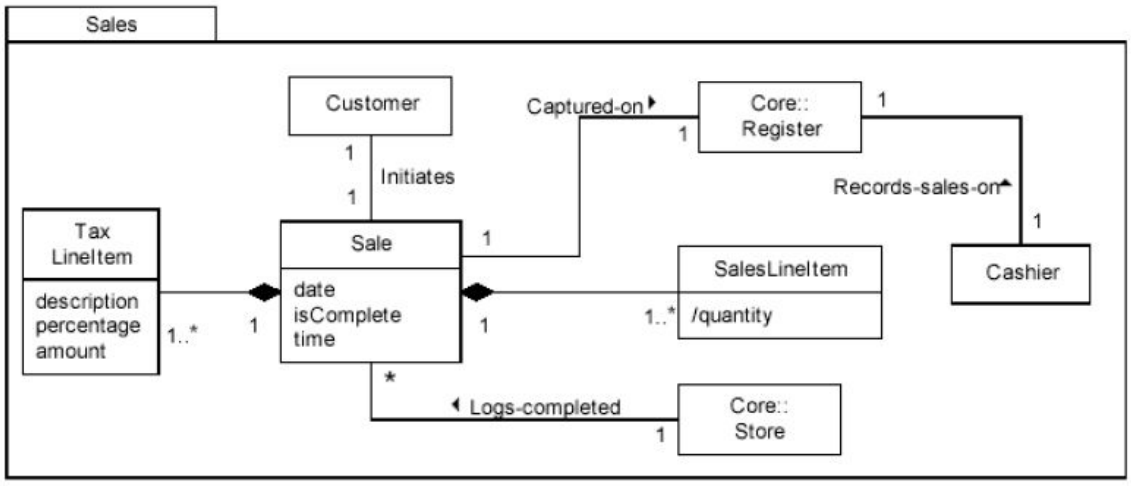
例如，我们将领域划分为 Core/Misc, Payments, Products, Sales 和 Authorization Transation 五个包，每个包中的类图又如下所示。

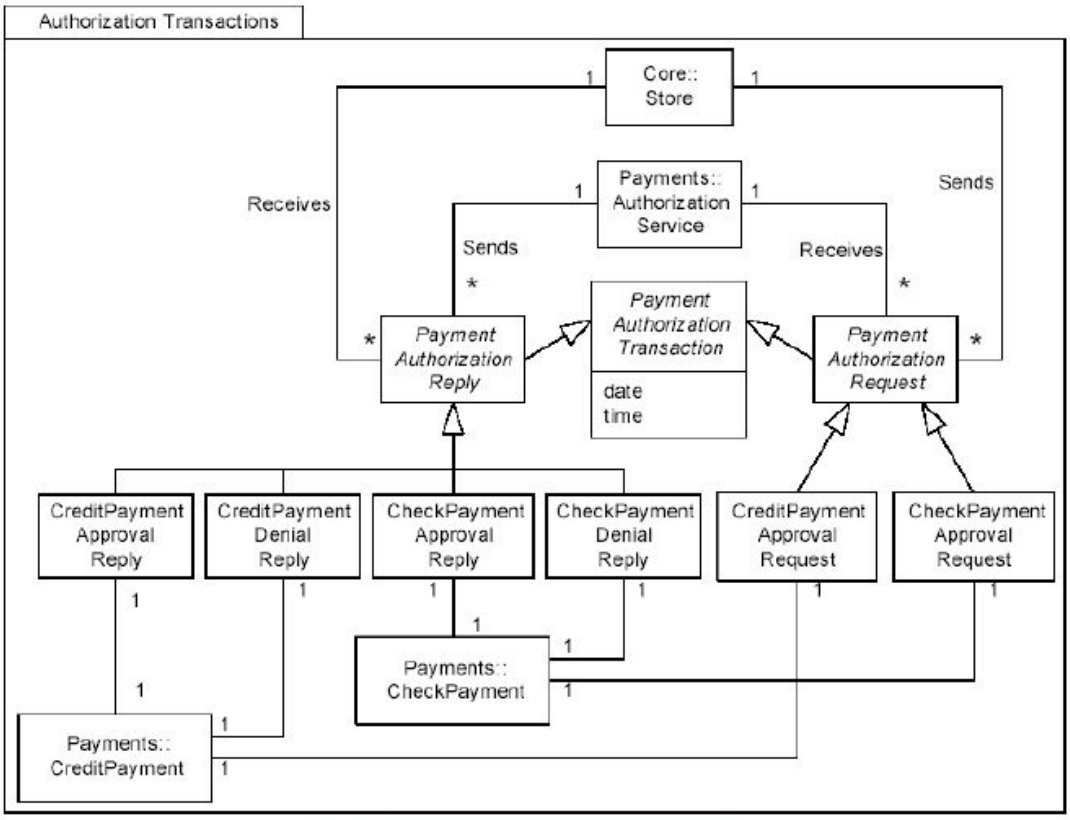

Domain
Core/Misc
Payments
Products
Sales
Authorization Transactions


Core/Misc
Store
address
name
Houses
1
1..*
Register
Manager
1..*
1
Employs


Payments
1
Authorizes-payments-of
1..*
Payment
amount
Core::Store
ServiceContract
merchantID
AuthorizationService
address
name
phoneNumber
Paid-by
1
Check
1
CashPayment
amountTendered
Credit Payment
Check Payment
Authorized-by
*
1
Check Authorization Service
Credit Authorization Service
1
* * *
*
Authorized-by
Logs
Establishes-credit-for
Establishes-identity-for
1
1
1
Accounts Receivable
CreditCard
expiryDate
number
DriversLicense
number
Authorization Transactions::
PaymentAuthorizationReply
1
1
Identifies
1
1
Abused-by
1
Sales::Customer
- CheckPayments have CheckPaymentReplies
- CreditPayments have CreditPaymentReplies


Products
Sales::
SalesLineItem
0..1
Described-by
*
1
Product
Specification
description
price
itemID
ProductCatalog
1
1..*
1
Records-sale-of
Describes
*
Core::
Store
Stocks
1
*
Item
1


Sales
Customer
1
1
Initiates
Captured-on
1
Core::
Register
1
Records-sales-on
1
Cashier
Tax
LineItem
description
percentage
amount
1..*
1
Sale
date
isComplete
time
1
1
SalesLineItem
1..*
/quantity
*
Logs-completed
1
Core::
Store


Authorization Transactions
Core::
Store
Payments::
Authorization
Service
Receives
Sends
Sends
Receives
Payment
Authorization
Reply
Payment
Authorization
Transaction
date
time
Payment
Authorization
Request
CreditPayment
Approval
Reply
CreditPayment
Denial
Reply
CheckPayment
Approval
Reply
CheckPayment
Denial
Reply
CreditPayment
Approval
Request
CheckPayment
Approval
Request
Payments::
CheckPayment
Payments::
CreditPayment