UML 与 Rational Rose 讲义

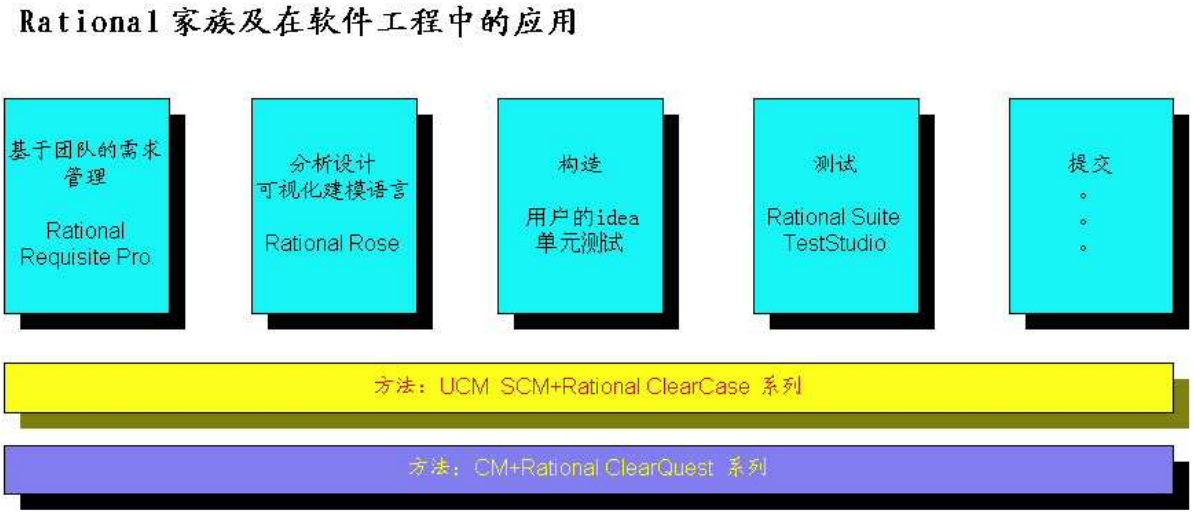
2002 年 8 月 2 日

目录

1. 第一周：概述

1.1 Rational 家族工具和应用方向介绍

1.1.1 用 Rational 工具和以往的软件开发方式，我们能改变什么？



- Rational ClearCase：提供版本控制，提供工作空间管理，建立管理和过程控制功能。
- Rational ClearQuest：定制缺陷和变更请求的信息域、过程、用户界面、查询、图表和报告等。
- Rational Requisite Pro：用户需求分析工具。
- Rational Rose：可视化的建模工具，将应用程序可视化、说明应用程序的完整结构或行为、创建一个模板引导你构建应用程序、将质量贯穿与整个开发生命周期、将开发过程中的所有决策信息整理归档。
- Rational Unified Process：是一个可以通过 Web 来使用的软件工程流程，可以提高团队的生产效率，并将最佳软件开发经验传递给所有团队成员。
- Rational SoDA：文档维护工具。
- Rational Suite TestStudio：全方位的质量测试，包括单元测试和自动化测试。

1.1.2 在纵向看：软件的开发过程分为需求、建模、构造、测试、提交五个阶段。

需求分析主要使用的工具为 Rational Requisite Pro

分析建模阶段主要使用的工具为：Rational Rose。

在系统构造阶段就是用基于面向对象的语言（如 Java、C++等）用程序员的经验、技能、好的想法合理的在体现前面二项分析的模型情况下具体实现软件的功能，并进行单元项的测试和修改。

测试阶段包括单元测试、整体联合测试、软件全方位的质量测试、满足项目需求的功能测试、

寻找应用程序缺陷的可靠性测试、查找响应时间瓶颈的性能测试等。并按测试的结果提出相应的改进建议。

提交阶段：包括为用户的安装、调试、培训、应用等具体行为。

1.1.3 从横行看：主要包括二大部分方法贯穿整个软件开发过程的始终。

方法一、UCM（全球变更管理）、SCM（软件配置管理），版本控制（Rational ClearCase）。

方法二、CM(变更管理)，工具为 Rational ClearQuest。

1.1.4 利用 Rational 的成功经验

1) 迭代化的开发
2) 基于构件的软件构架
3) 可视化建模(UML)
4) 持续地质量验证
5) 配置和变更管理

1.2 OOA&D 面向对象的基本原则：

面向对象机制是另一种观察应用程序的方式。利用面向对象方法，把应用程序分成许多小块(或对象)，这些对象是相互独立的。然后可以组合这些对象，建立应用程序。可以把它看成砌砖墙。第一步要建立或购买基本对象（各种砖块）。有了这些砖块后，就可以砌出砖墙了。在计算机领域中建立或购买基本对象后，就可以集成起来，生成新的应用程序。

面向对象机制的一个好处是可以一次性地建立组件，然后反复地使用。

那么面向对象机制与传统开发方法有什么不同呢？传统开发方法集中考虑系统要维护的信息。用这种方法时，我们要向用户询问他们需要什么信息，然后设计保存信息的数据库，提供输入信息的屏幕并打印显示信息的报表。换句话说，我们关注信息，而不是关注信息的作用或系统的功能。这种方法是以数据为中心的，多年来用它建立了成千上万个系统。

以数据为中心的模型适合数据设计和捕获信息。但用来设计商业应用程序就有问题。一个主要问题是系统要求随着时间不断变化。以数据为中心的系统可以方便地处理数据库变化，但很难实现商业规则变化和系统功能变化。

面向对象机制的开发正是要解决这个问题。利用面向对象机制，我们同时关注信息与功能。因此，我们可开发密切关注和适应信息与功能变化的系统。

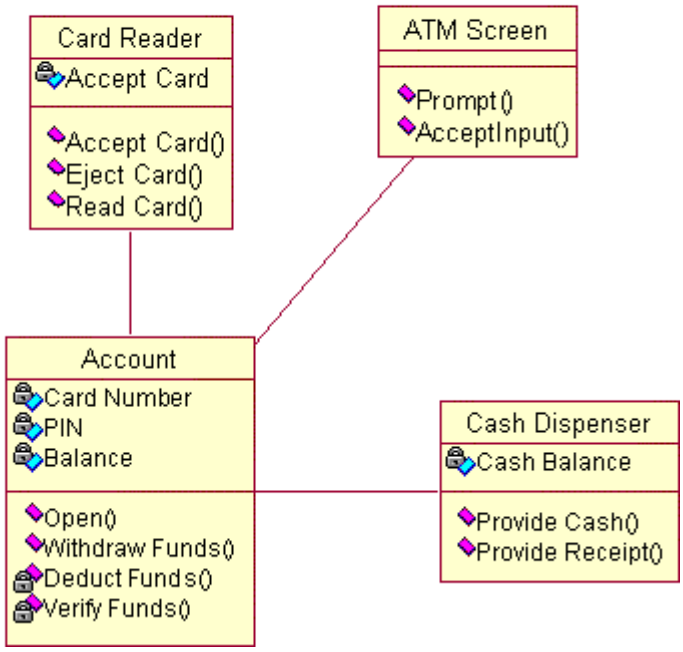
要实现灵活性带来的好处，只能通过设计好的面向对象系统。这就要求了解面向对象的基本原则：

a) 封装(Encapsulation)

在面向对象系统中，我们将信息与处理信息的功能组合起来，然后将其包装成对象，称为封装。另一种理解封装的方法就是把应用程序分解成较小的功能组件。

例如，我们把与银行有关的信息，如帐号、节余、客户名和功能：开户、销户、存取款。我们将这些信息与处理信息的功能封装成帐目对象。结果，银行系统对帐目的任何改变就会在帐目对象中实现。它是所有帐目信息与功能的集合。

银行模型：有银行信用帐户的客户，增加支票帐户的透支额只要改变 Account 类。

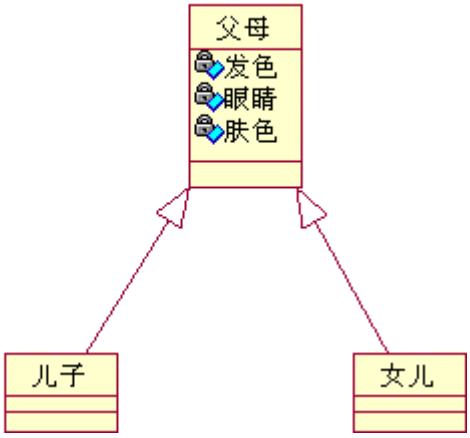


封装的另一好处是将系统改变的影响限制在对象内。

b) 继承(Inheritance)

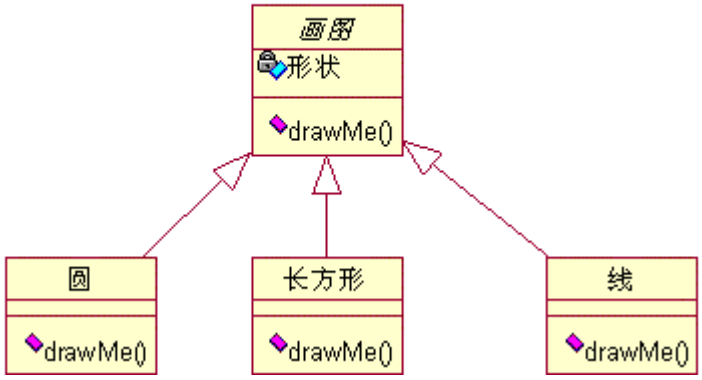
在面向对象系统中，继承机制可以根据旧对象生成新对象。子对象继承父对象的特性。

继承的主要好处之一是易于维护。（变化时只需改变父对象）



c) 多态(Polymorphism)

多态的定义是多种不同形式、阶段或类型发生的事，表示特定功能有多种形式或实现方法。



假如没有多态代码为

```
Function shape.drawMe()
{
case shape.Type
    case “Circle”
        shape.drawCircle();
    case” Rectangle”
        shape.drawRectangle();
    case” Line”
        shape.drawLine();
Endcase
}
```

多态代码为

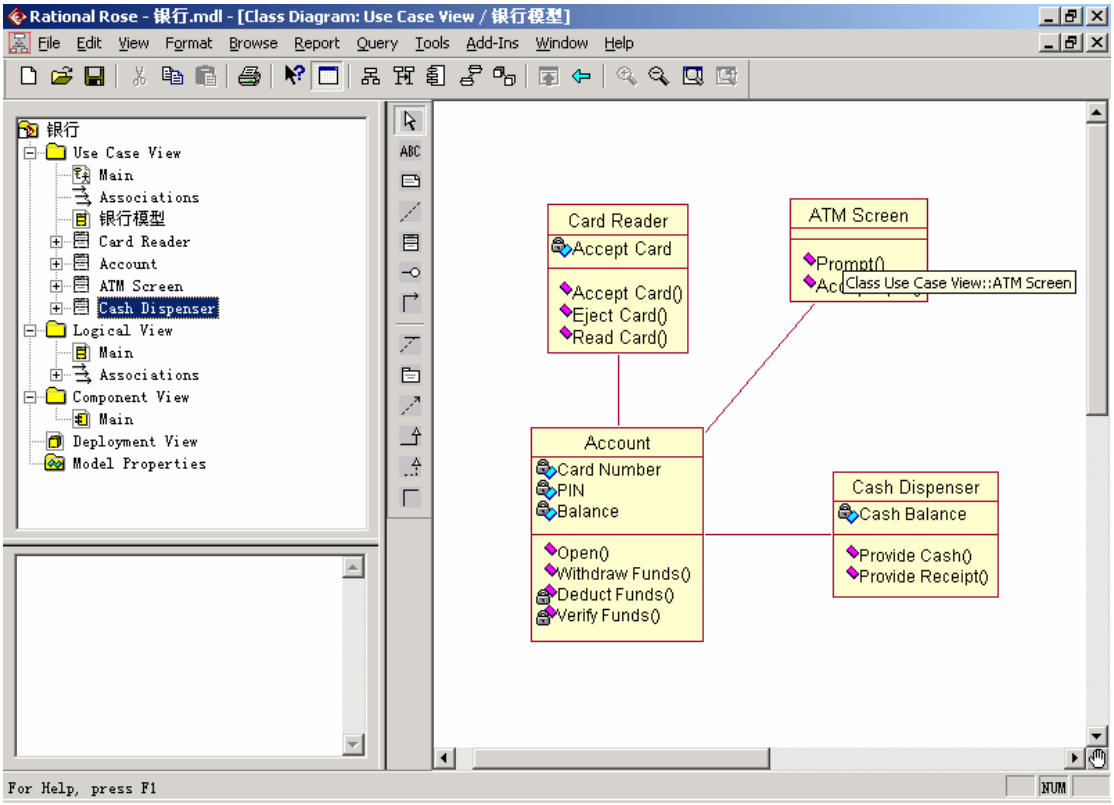
```
Function draw()
{
    shape.drawMe();
}
```

1.3 Rational Rose 的界面介绍

可视化建模将模型中的信息用标准图形元素直观地显示。

建立模型后，可以向所有感兴趣的方面显示这个模型，让他们对模型中的重要信息一目了然。

演示并解说 Rose 的主要功能和使用方法。



2. 第二周：静态建模：用例和用例图(Use Case Diagram)

2.1 角色和角色之间的关系

2.1.1 角色

角色(actor)是与系统交互的人或事。角色是一个群体概念，代表的是一类能使用某个功能的人或事，角色不是指某个个体。

“所谓系统交互”指的是角色向系统发送消息，从系统中接受消息，或是在系统中交换信息。只要使用用例，与系统相互交流的任何人或事都是角色。比如，某人使用系统中提供的用例，则该人是角色；与系统进行通讯（通过用例）的某种硬件设备也是角色。

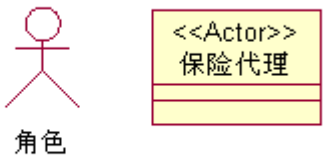
角色与系统进行通讯的收、发消息机制，与面向对象编程中的消息机制很像。角色是启动用例的前提条件，又称为刺激物(stimulus)。

角色可以分为：主要角色(primary actor)指执行系统主要功能的角色，次要角色(secondary actor)指使用系统的次要功能的角色，次要功能是指一般完成维护系统的功能(比如，管理数据库、通讯、备份等)。

或分为：主动角色（可以初始化用例），被动角色（不能初始化用例）仅仅参与一个或多个用例，在某个时刻与用例通讯。

2.1.2 发现角色：

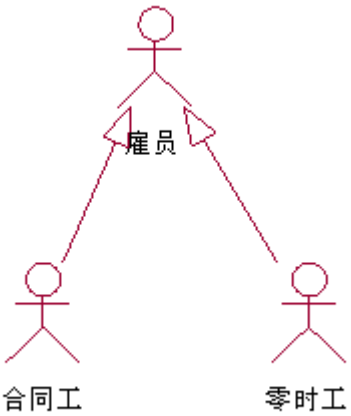
- 使用系统主要功能的人是谁（即主要角色）？
- 需要借助与系统完成日常工作的人是谁？
- 谁来维护、管理系统（次要角色），保证系统正常工作？
- 系统控制的硬件设备有哪些？
- 系统需要与哪些其他系统交互？其他系统包括计算机系统，也包括该系统将要使用的计算机中的其他应用软件。其他系统也分二类，一类是启动该系统，另一类是该系统要使用的系统。
- 对系统产生的结果感兴趣的人或事是哪些？



角色类的图示方式

2.1.3 角色之间的关系

角色是类所以拥有与类相同的关系描述：



把某些角色的共同行为（原角色中的部分行为），抽取出来表示成通用行为，且把它们描述成为超类(superclass)，即通用化关系。

这样在定义某一具体的角色时，仅仅把具体的角色所特有的那部分行为定义一下就行了，具体角色的通用行为则不必重新定义，只要继承超类中相应的行为即可。表示方式如上图。

2.2 用例和用例之间的关系

用例代表的是一个完整的功能。在 UML 中的用例是动作步骤的集合。

2.2.1 用例的特征：

用例总由角色初始化

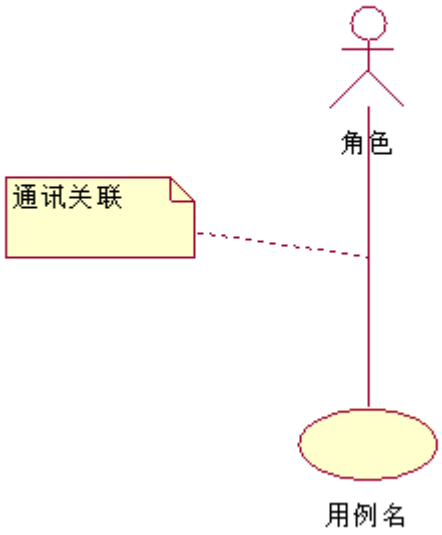
用例为角色提供值

用例具有完全性

2.2.2 发现用例

- 角色需要从系统中获得哪些功能？角色需要做什么？
- 角色需要读取、产生、删除、修改或存储系统中的某些消息吗？
- 系统中发生的事件需要通知角色吗？或者角色需要通知系统某件事吗？这些事件（功能）能干些什么？
- 如果用系统的新功能处理角色的日常工作是简单化了，还是提高了工作效率？
- 系统需要的输入/输出是什么信息？这些输入/输出信息从哪儿来到哪儿去？
- 系统当前的这种实现方法要解决的问题是什么（也许是用自动系统代替手工操作）？

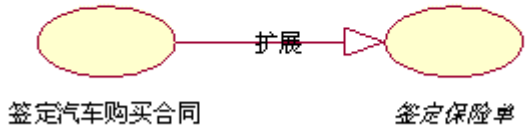
表示方式：



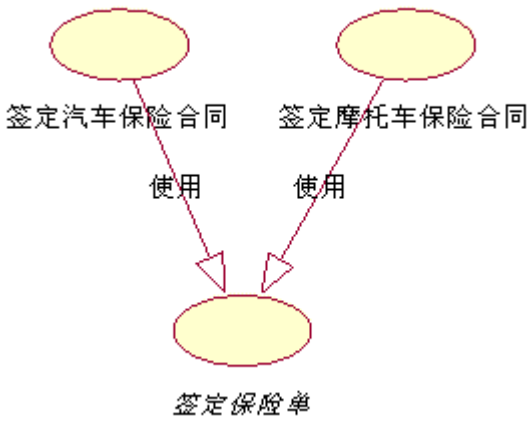
2.2.3 用例之间的关系

用例之间有扩展、使用、组合三种关系：扩展和使用是通用化关系的另一种体现。组合则把相关的关系打成包当作一个整体看待。

- 扩展关系是一个用例中加入一些新的动作后则构成了另一个用例。



- 当一个用例使用另一个用例时，这两个用例间就构成使用关系。



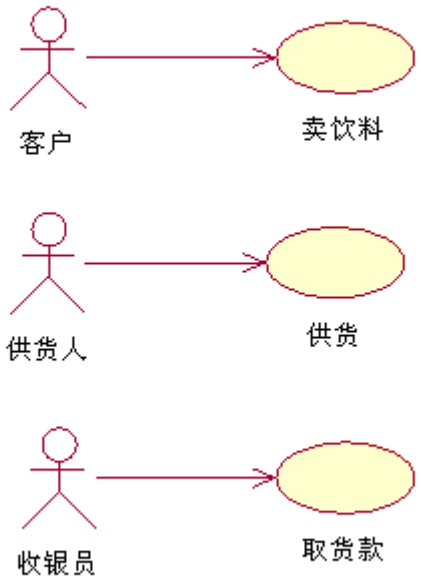
2.3 举例建立用例图的方法和文档的详细注释

2.3.1 描述用例

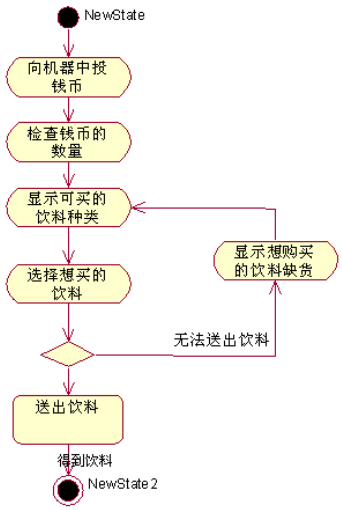
描述要点：描述用例时，应着重描述系统从外界看来会有什么样的行为，而不管该行为在系统内部是如何具体实现的，即只管外部能力，不管内部细节。

用例描述应包括下面几点：

1) 用例的目标：最终任务、结果等
2) 用例是怎样被启动的：何角色在何情况下启动。
3) 角色和用例之间的消息流：通知、修改、决定等
4) 用例的多种执行方案：
5) 用例怎样才算完成并把值传给了角色。



自动售货系统



顾客向自动售货机购货的过程

2.3.2 测试用例

有效性检查在系统开发之前。当用例模型构造完成后，开发者将模型交由用户讨论，由用户检查模型能否满足客户对系统的需求。发现问题产生修改，最终，用户和开发者之间对系统功能达成共识。反之，整个工程从头重来。

正确性测试保证系统的工作符合规格说明。

用具体的用例测试系统的行为，又称“漫游用例”。即指定一人扮演角色，另一人扮演系统。两者消息传递的推演发现不足。（recommend）

用用例描述本身测试，或称定义测试。

2.3.3 实现用例

用例用来描述系统应能实现的独立功能，实现用例就是在系统内部实现用例中所描述的动作，通过把用例描述的动作转化为对象之间的相互协作，完成用例的实现。

UML 中实现用例的基本思想是用协作表示用例，而协作又被细化为若干个图。协作的实现用脚本描述。

1) 用例实现为协作

协作是是实现用例内部依赖关系的解决方案。通过使用类、对象、类（或对象）之间的关系（协作中的关系称为上下文）和它们之间的交互实现需要的功能（协作实现的功能又称交互）。

2) 协作用若干个图表示

表示协作的图有协作图、序列图和活动图。这些图用于表示协作中的类（或对象）与类（或对象）之间的关系交互。在有些场合，一张协作图就完全能够反映出实际用例的协

作画面，而在另一些场合，只有把三种不同的图结合起来，才能反映协作状态。

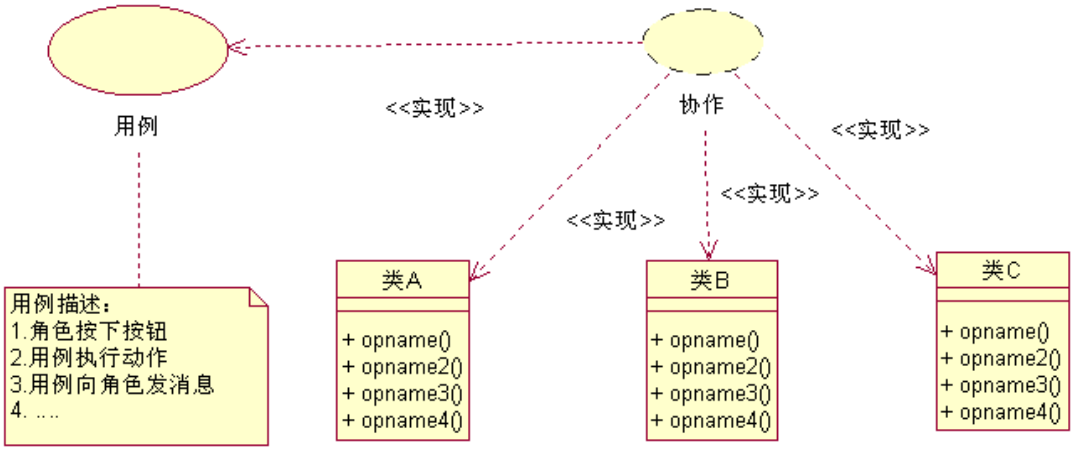
3) 协作的实例－－脚本

脚本是用例的一次具体执行过程，它代表了用例的一种使用方法。当把脚本看作用例的实例时，对角色而言，只需描述脚本的外部行为，也就是能够完成什么样的功能，而忽略完成该脚本的具体细节，从而到达帮助用户理解用例含义的目的：当把脚本看作协作的实例时，则要描述脚本的具体实现细节（利用类、操作和它们之间的通讯）。

实现用例的主要任务是把用例描述中的各个步骤和动作变换为协作中的类、类的操作和类之间的关系。具体说来，就是把用例中每个步骤完成的工作交给协作中的类来完成。实质上，每个步骤转换成类的操作，一个类中可以有多个操作。注意，一个类可以实现多个用例，也就是，一个类可以集成多种角色。

用例和它的实现（即协作）之间的关系可以用精化关系表示（下图为带箭头的点画线），也可以用用例工具中的不可见的超链实现。使用用例工具中的超链，能方便地将用例视图转换为协作或脚本。

若想成功地利用类表示用例描述，需要借助开发者的经验。通常，开发者必须反反复复地多次试验各种不同的可能情况，逐步完善解决方案，使该方案能够实现用例描述且灵活易扩展。如果将来用例描述改变了，只要将其对应的实现作简单地修改即可。



协作实现的类图示

另外，UML 的创始人 Jacobson 定义了三种类型的版类对象类（stereotype object type），利用这三个对象类描述用例的实现。每对象类能胜任的职责是：

边界对象(Boundary objects)：这种对象类紧靠系统的边界（虽然仍在系统内部），它负责与

系统外部的角色交互，在角色和系统内部的其它对象类之间传递消息。

实体对象(Entity objects)：这种对象类代表系统控制区域内的区域实体。它是被动对象，本身不能启动交互。在信息系统中，实体对象通常是持久的，存储在数据库中，实现对象也可以出现在多个用例中。

控制对象(Control objects)：这种对象类控制一组对象之间的交互。控制对象可以是刺激用例启动的角色，也可以用来实现若干个用例的普通序列。控制对象通常仅存在于用例执行阶段。

对象类各有自己的图标，可以用来图示协作和类图。

三种版类图示

3. 第三周：静态建模：类图和对象图(Class Diagram)

构成面向对象模型的基本元素有类(class)、对象(objects)、类与类之间的关系等。用面向对象的思想描述问题，能够把复杂的系统简单化、直观化，而且易于用面向对象语言编程实现，还方便日后对系统的维护工作。

类：是提供对象蓝图的项目。是包装信息和行为的项目。

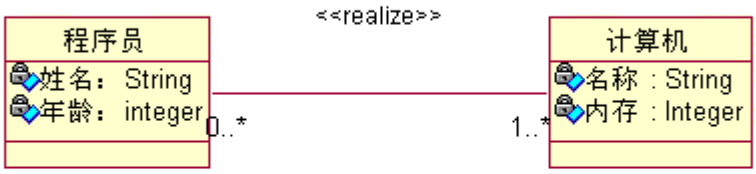
对象：是可以控制和操作的实体，它可以是一设备，一个组织或一个商务。

类是对象的抽象描述，它包括属性的描述和行为的描述二方面。

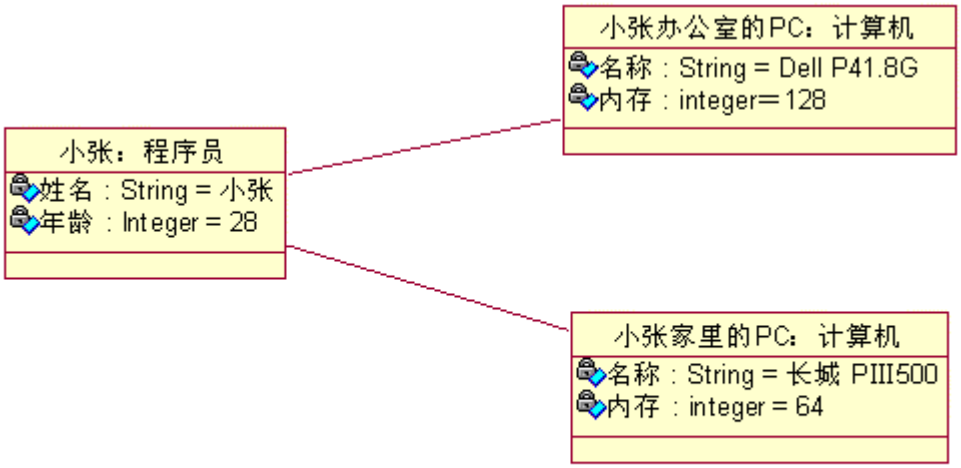
举例：

汽车类：具有功能有启动、行驶、制动，另外对汽车车身长度、汽缸容量等也有一定的要求。

汽车对象：桑塔纳、宝马、夏利等。



类图



对象图

类图是用类和它们之间的关系描述系统的一种图示，是从静态角度表示系统的，因此类图属于一种静态模型。

3.1 类的定义

建模者定义的类通常要有这样二个特点：一是使用来自问题域的概念，二是类的名字用该类实际代表的涵义命名。

准确地将系统要处理的数据抽象成类的属性，将处理数据的方法抽象为操作是一件不容易的事，下面列出一些可以帮助建模者定义类的问题：

有没有一定要存储或分析的信息？如果存在需要存储，分析或处理的信息，那么该信息有可能就是一个类。这里讲的信息可以是概念（该概念总在系统中出现）或事件或事务（它发生在某一时刻）。

有没有外部系统？如果有，外部系统可以看作类，该类可以是本系统所包含的类也可以是本系统与之交互的类。

有没有模板、类库、组件等？如果有通常应作为类。

系统中有被控制的设备吗？凡是与系统相连的任何设备都要有相应的类。通过这些类控制设备。

有无需要表示的组织机构？在计算机系统中表示组织机构通常用类，特别是构建商务模型时用得更多。

系统中有哪些角色？这些角色也可以看成类。比如，用户、系统操作员、客户等。

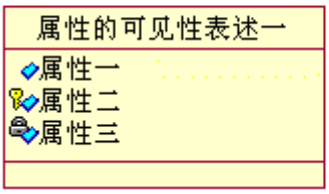
3.2 类图的属性、操作和使用方法

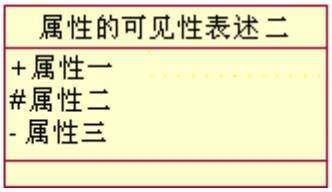
类的图示

属性类型可以是：整型、实型、布尔型、枚举型等基本类型。

属性的可见性通常分为：公有的(public)、私有的(private)和保护的（protected)。

表示方式：





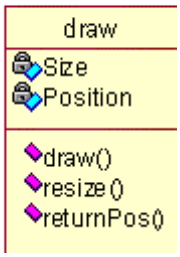
生成的代码（Java 语言）

```
public class classvisiblity
{
   public int att1;
   protected int att2;
   private int att3;

   public classvisiblity()
   {
   }
}
```

属性仅仅表示了需要处理的数据，对数据的具体处理方法的描述则放在操作部分。操作通常又

称为函数，它是类的一个组成部分，只能作用于该类的对象上。从这一点也可以看出，类将事件和对数据进行处理的函数封装起来，形成一个完整的整体，这种机制非常符合问题本身的特性。




```
public class draw
{
   private int Size;
   private int Position;
   public draw()
   {
   }
   public void resize(integer percentX, integer percentY)
   {
   }
   public void returnPos()
   {
   }
}
```

使用基本类

将 case 工具的工作环境配置为某一具体编程语言，这样该语言本身所提供的基本类型就可以在 case 工具中使用了。

3.3 类图之间的关系

3.3.1 关联

关联(Associations)：是类之间的词法连接，在 Class 框图中用单线表示。

例学校新开三门选修课，每课的招生人数为 20 人表示如下：

关联和重数

代码：

```
public class student
{
   private int name;
   private int grade;
   public course theCourse[];

   public student()
   {
   }
}
public class course
{
   private int name;
   private int teacher;
   public student theStudent[];

   public course()
   {
   }
}
```

3.3.2 依赖性

依赖性(Dependencies)：依赖性总是单向的，显示一个类依赖于另一个类的定义。
比如：某个类中使用另一个类的对象作为操作中的参数，则这二个类之间就具有依赖性。

依赖性关系

精化关系

3.3.3 精化关系

精化关系用于表示同一事物的两种描述之间的关系。对同一事物的两种描述建立在不同的抽象层上。比如，定义了某数据类型，然后将其实现为某种语言中的类，那么抽象定义的类型与用语言实现的类之间就是精化关系，这种情况也被称为实现。

又如，对某事物的精确描述和粗糙描述。

精化关系常用于模型化表示同一事物的不同实现。比如，一个简单实现，一个比较复杂而高效的实现。

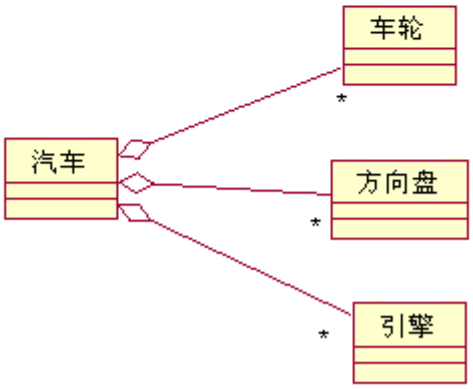
精化用于模型的协调。在对大型工程项目建模时，往往需要建立许多模型。然后用精化关系对这些模型进行协调。协调过程可以显示不同抽象层上的模型之间是怎样联系在一起的：显示来自不同建模阶段的模型之间具有什么样的关系；支持配置管理和模型之间的追溯。

3.3.4 聚合

聚合(Aggregations)：是强关联。积累关系是整体与个体间的关系。

聚合的示例

共享聚合



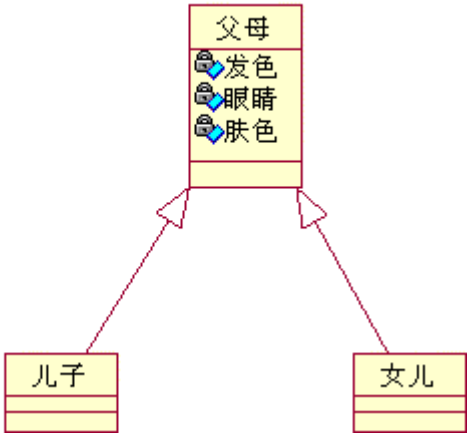
复合聚合

如果聚合关系中的处于部分方的对象同时参与了多个处于整体方对象的构成，则该聚合称为共享聚合。

如果构成整体类的部分类，完全隶属于整体类，则这样的聚合称为复合聚合。

3.3.5 一般化

一般化(Generalizations)：显示类之间的继承关系。



父类中的属性和操作又称作成员，不同可见性的成员在子类中随意使用：父类中的私有成员在子类中也是私有的，但是子类的对象不能存取父类中的私有成员。

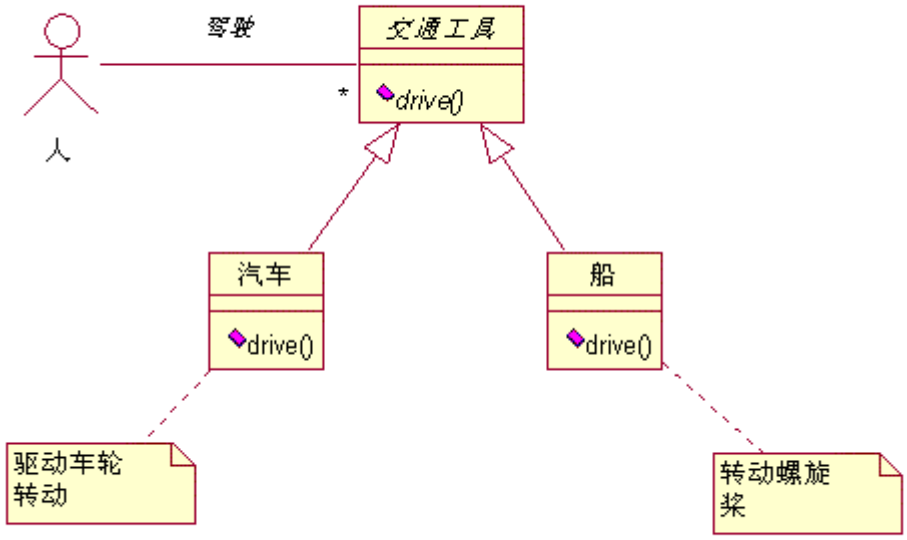
一个类中的私有成员都不允许外界元素对其作任何操作，这就到达了保护数据的目的。

如果既需要保护父类的成员（相当于私有的），又需要让其子类也能存取父类的成员，那么父类的成员的可见性应设为保护的。拥有保护可见性的成员只能被具有继承关系的类存取和操作。

类的继承关系可以是多层的。也就是说，一个子类本身还可以作另一个类的父类，层层继承下去。

没有具体对象的类称作抽象类。抽象类一般做为父类，用于描述其他类（子类）的公共属性和行为（操作）。

抽象类中一般都带有抽象的操作。抽象操作仅仅用来描述该抽象类的所有子类应有什么样的行为，抽象操作只标记出返回值、操作的名称和参数表，关于操作的具体实现细节并不详细书写出来，抽象操作的具体实现细节由继承抽象类的子类实现。

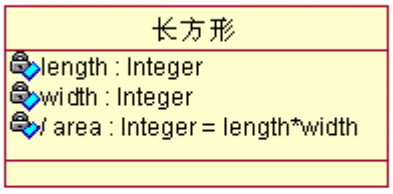


多态示例

当人执行（调用）drive()操作时，如果当时可用的对象是汽车，那么汽车轮子将被转动；如果当时可用的对象是船，那么螺旋桨将会动起来，这种在运动时可能执行的多种功能，称为多态。多态技术利用抽象类定义操作，而用子类定义处理该操作的方法，达到单一接口，多种功能的目的，在 C++语言中，多态利用虚拟函数或纯虚拟函数实现。

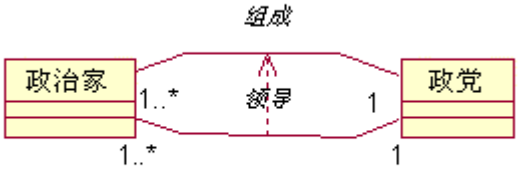
3.4 类图的约束和派生规则

UML 中的规则称为约束和派生。约束用于限制一个模型（多重、不相交、完全和不完全）。派生用于描述某种事物的产生规则，比如说，一个长方形面积可以由长和宽派生出来（两者之积）。一般说来，约束和派生能应用于任何模型元素，但最常用于属性、关联、继承、角色和时间。



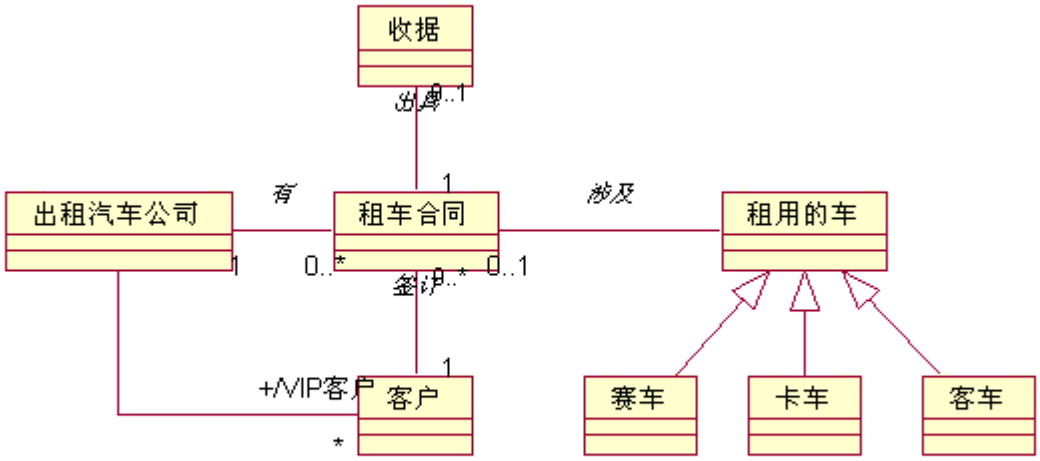
派生属性图示

关联关系可以被约束，也可以派生。如果一个关联是另一个关联的子集，则它们之间就会存在约束关联。下图中的“领导”关联中的政治家是“组成”关联中成员的子集。



约束关联图示

派生关联由现有的关联关系衍生而来，派生关联的名称前加一条斜线，下图表示的“VIP 客户”关联就是派生来的。因为出租车公司和很多客户有租车合同，这里面有些人很重要的，于是在公司和客户之间直接派生出关联关系。



派生关联图示

3.5 类图的接口、组件、包和模板

UML 中的包、组件和类也可以定义接口，利用接口说明包、组件和类能够支持的行为。在建模时，接口起到非常重要的作用，因为模型元素之间的相互协作都是通过接口进行的。一个结构良好的系统，其接口必然也定义得非常规范。

接口通常被描述为抽象操作，也就是只用标识（返回值、操作名称、参数表）说明它的行为，而真正实现部分放在使用该接口的元素中。这样，应用该接口的不同元素就可以对接口采用不同的实现方法。在执行过程中，调用该接口的对象看到的仅仅是接口，而不管其他事情，比如，该接口是由哪个类实现的，怎样实现的，都有哪些类实现了该接口等。通俗地讲，接口的具体实现过程、方法，对调用该接口的对象是透明的。

接口在类图中表示为一个带接口名称的小圆圈。

3.6 类图怎样生成 java 代码框架

1) 选定要转换的类图
2) 选择转换的语言我们选 tools/java
3) 定义工程目录
4) 语法检查（若不通过则模型有逻辑错误查看 log）
5) 生成框架代码
6) 浏览代码

3.7 类的 java 代码生成类图

1) 选定要转换的类源代码（*.java）
2) 选择转换的语言我们选 tools/java
3) 定义工程目录
4) Reverse Engineer Java 生成类图

4. 第四周：动态建模：序列图和协作图(Sequence & Callaboration Diagram)

在面向对象的编程中，两个对象之间的交互表现为一个对象发送一个消息给另一个对象。在这里，有一点很重要，不能仅仅从字面上理解“消息”这个词，因为消息是在通讯协议中发送的。通常情况下，当一个对象调用另一个对象中的操作时，消息是通过一个简单的操作调用来实现；当操作执行完成时，控制和执行结果返回给调用者。消息也可能是通过一些通讯机制在网络上或一台计算机内部发送的真正的报文。在所有动态图（序列图、协作图、状态图、活动图）中，消息是作为对象间的一种通讯方式来表示的。具体来说，消息是连接发送者和接收者的一根箭头线。

消息可分为如下几种类型：

简单(Simple)：这个选项指定消息在单个控制线程中运行。

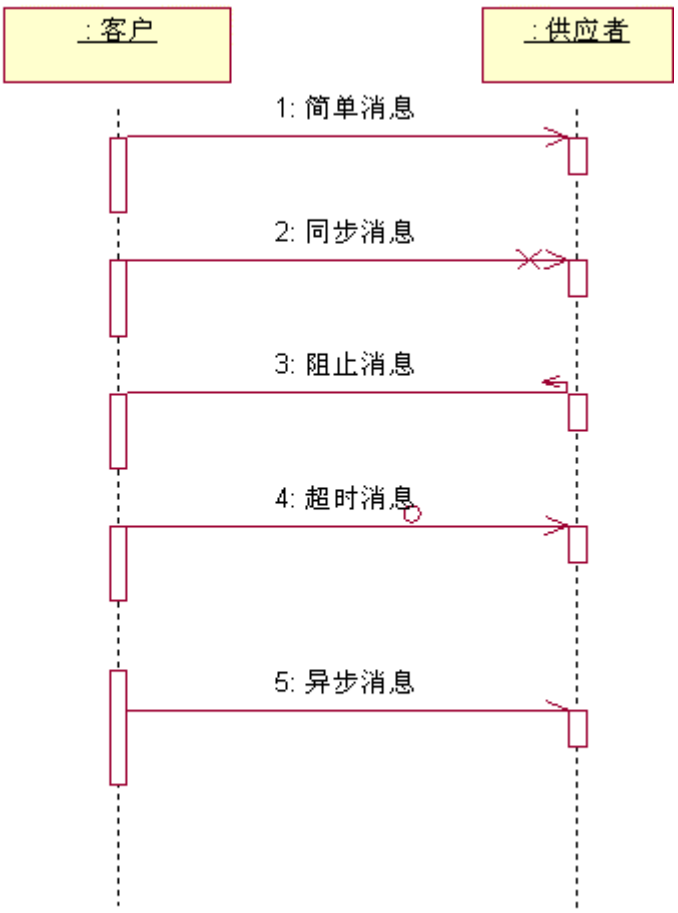
同步(Synchronous)：这个选项用于客户发出消息后等待供应者。

阻止(Balking)：使用这个选项时，客户发出消息给供应者。如果供应者无法立即接受消息，则客户放弃这个消息。

超时(Timeout)：使用这个选项时，客户发出消息给供应者并等待指定的时间。如果供应者无法在指定时间内接收消息，则客户放弃这个消息。

异步(Asynchronous)：使用这个选项时，客户发出消息给供应者然后客户继续处理，不等待消息是否接收。

表示方式为：



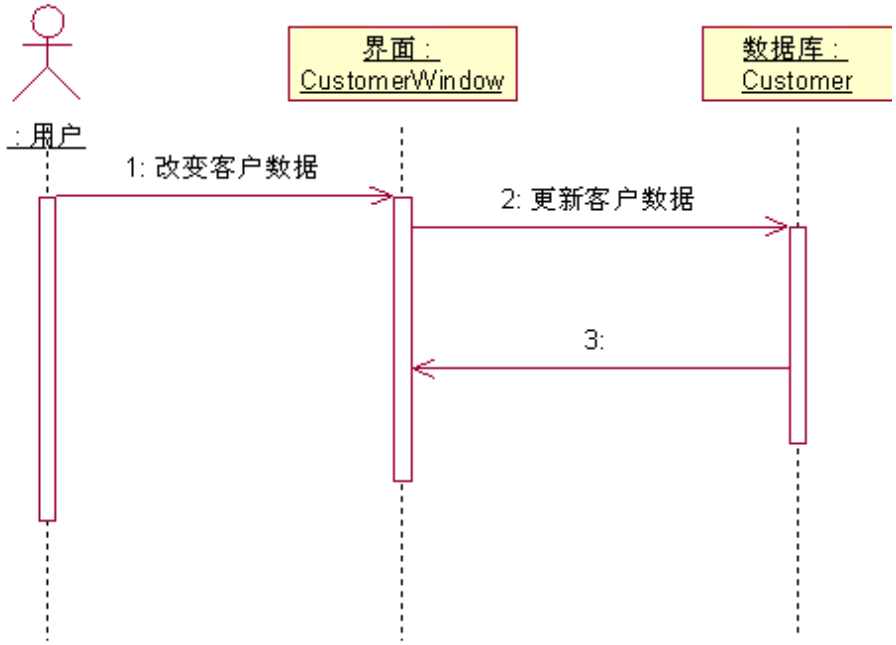
消息的类型和表示方式

4.1 序列图

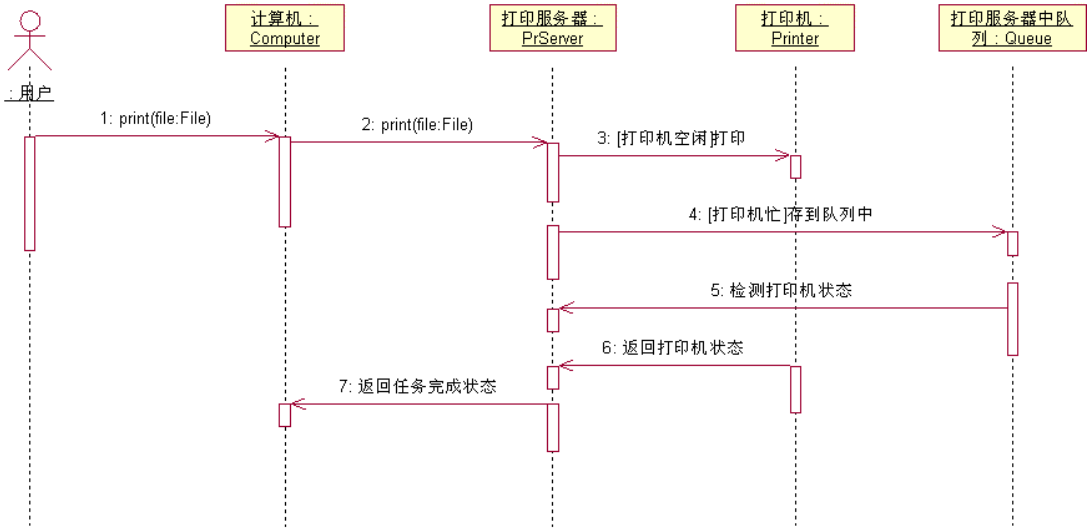
序列图：描述对象如何相互交互和通讯。序列图中的最重要的是时间。通过序列图，可以看出为了完成某种功能一组对象如何发送和接收一序列消息。

4.1.1 序列图的格式和并发事件

有二种使用序列图的方式：一般格式和实例格式。实例格式详细描述一次可能的交互。实例格式没有任何条件、分枝或循环，它仅仅显示选定的情节的交互。而一般格式则描述所有的情节，因此，包括了分枝、条件和循环。



更新客户数据的实例格式图示



打印过程的一般格式图示

一条消息是一次对象间的通讯，通讯所传递的消息是期望某种动作发生。通常情况下，接收到一条消息被认为是一个事件。消息可以是信号，操作调用或其他类似的东西（如，C++中的RPC(Remote Procedure Calls)或 Java 中的 RMI(Remote Method Invocation)）。

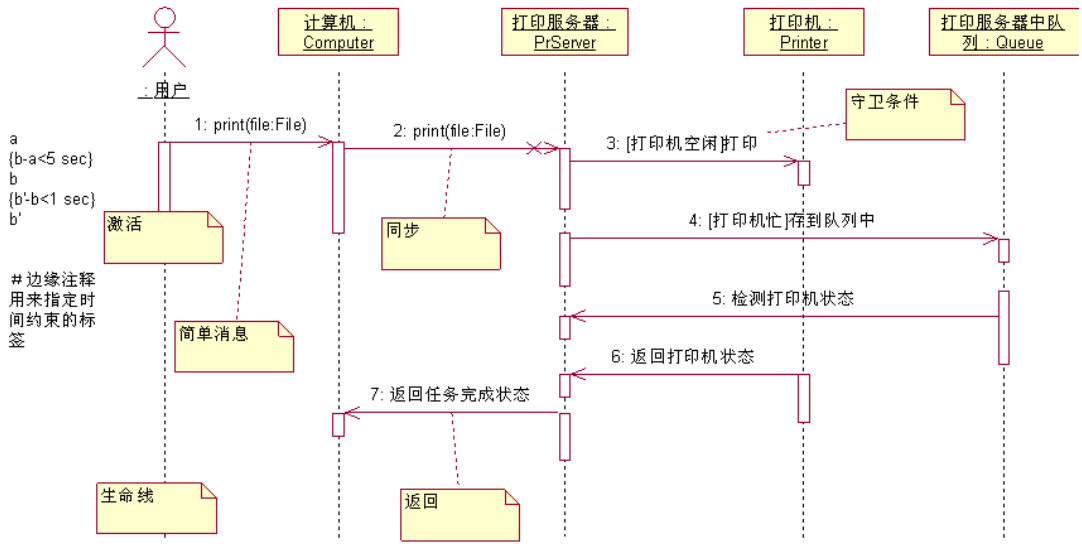
当对象收到一条消息时，活动就开始，称之为“激活(activation)”。激活显示控制的焦点，对象及时地在某一点执行。一个被激活的对象要么执行自己的代码或等待另一个对象返回结果。“激活”

用对象的生命线上的窄的矩形来表示。生命线表示一个对象在一个特定时间内的存在，它是一条从上到下的虚线。消息用对象的生命线间的箭头线表示。每一条消息可以有一个说明，内容包括名称和参数。消息可以有序列号。同时消息可以有条件约束。

在某些系统中，对象并发执行，每一个对象有一条自己的控制线程。如果系统使用这样的并发对象，则通过激活、异步消息和活动对象来表示。

4.1.2 序列图定义迭代和约束的标签

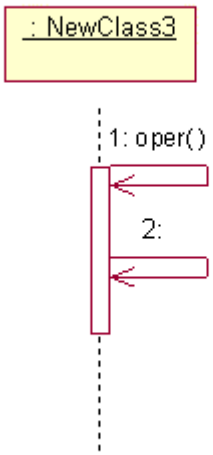
在对象图的左边和右边可以有标签和注释。标签可以是任何类型，如定时标记，激活过程中的动作描述、约束等等。



定义迭代和约束的标签

4.1.3 序列图的递归方式

在很多算法中，递归是一种常用的技术。当一个操作调用它自己时就是递归，下图中递归为激活它自己。当一个操作调用它本身时，消息总是同步的，因而可以用序列图中同样的方法来标记它。用一条简单消息来表示返回。

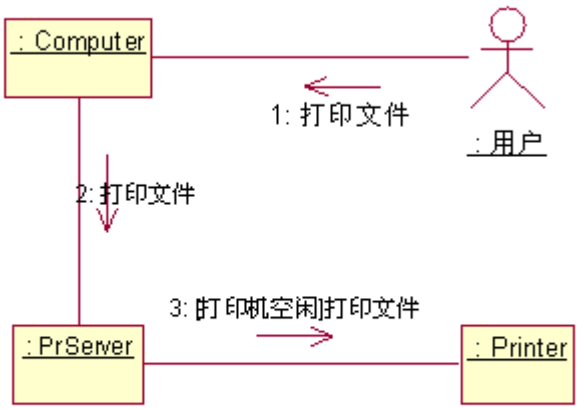


递归示图

4.2 协作图

协作图：描述对象交互，但侧重于空间的协作，意即明确地给出对象间的关系（链接）。

4.2.1 协作图的格式和消息流



协作图示

用户发送打印消息给计算机，计算机发送一个消息给打印服务器，如果打印机空闲打印服务器发送打印消息给打印机。

协作图中的消息标签用下面的语法规则来书写：

前缀　守卫条件 序列表达式　返回值：＝说明

1：[z:=1..n]1.2.3.7:prim:=nextPrim(prim)

4.2.2 协作图的链接

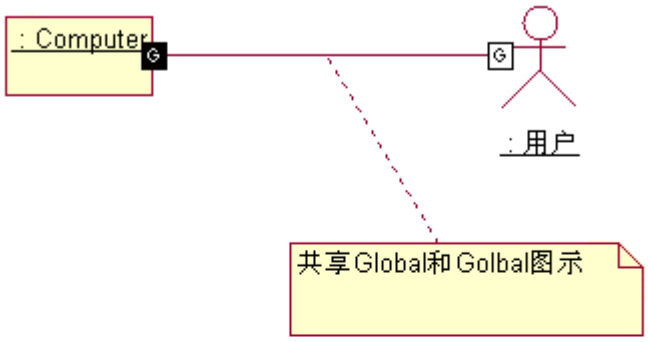
一条链接是两个对象间的连接。链接上的任何对象的角色均作为链接的端点，与链接的量词在一起。量词和角色均在对象类的类图中说明。有些版类可能会同链接上的角色（链接角色）放在一起：Global,Local,Parameter,Field。

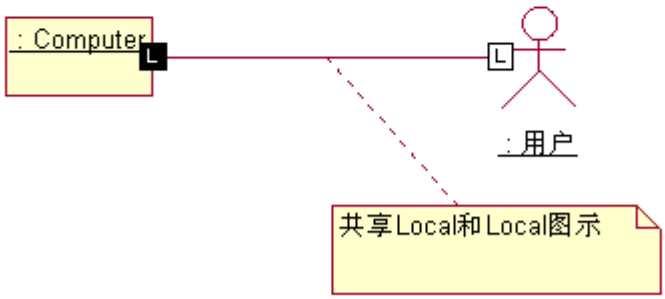
Global：是加在链接角色上的约束，说明与对象对应的实例是可见的，因为它是在全局范围内（可以通过系统范围内的全局名来访问对象）。

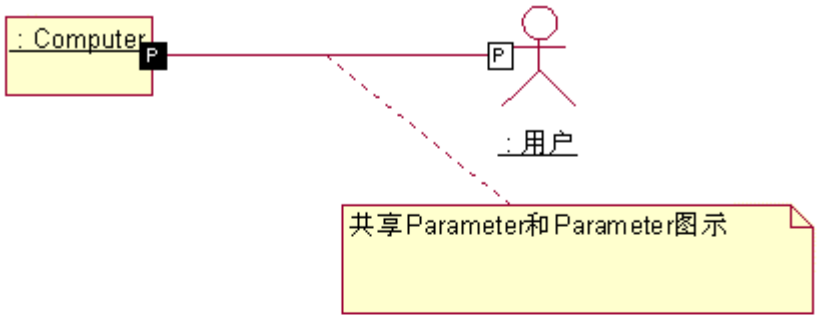
Local：是加在链接角色上的约束，说明与对象对应的实例是可见的，因为它是操作中的局部变量。

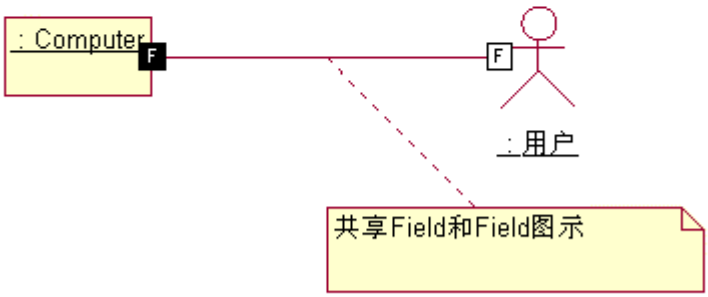
Parameter：是加在链接角色上的约束，说明与对象对应的实例是可见的，因为它是操作中的一个参数。

Field：加在链接角色上的约束。因为它是操作中的“局部变量”。


: Computer
G
G
: 用户
共享Global和Golbal图示


: Computer
L
L
: 用户
共享Local和Local图示


: Computer
P
P
: 用户
共享Parameter和Parameter图示


: Computer
F
F
: 用户
共享Field和Field图示

版类和链接上的角色

4.2.3 对象的生命周期

在给一个协作中对象被创建和破坏的完成过程。

4.3 从序列图转换为协作图的方式

1) 选中需转换的序列图
2) 点击工具栏 Browse/Create Collaboration Diagram 或直接按 F5
3) 生成协作图

4.4 从协作图转换为序列图的方式

1) 选中需转换的协作图
2) 点击工具栏 Browse/Create Sequence Diagram 或直接按 F5
3) 生成序列图

5. 第五周：动态建模：状态图/活动图(Statechart / Activity Diagram)

5.1 状态图

状态图：描述对象在生命周期内处于哪些状态，每一种状态的行为以及什么样的事件引起对象状态发生改变；例如，一张发票可以是已付（状态 paid）和未付（unpaid）。

5.1.1 状态和转移

所有对象均有状态：状态是对象操作的前一次活动的结果，通常情况下，状态由对象的属性以及指向其他对象的链接来决定的。类的状态由类中的指定属性来说明或对象的状态由对象中的通用属性的值来确定。

当某些事情发生的对象的状态发生改变，我们称改变对象状态的事情为“事件”，如下图的付了支票。

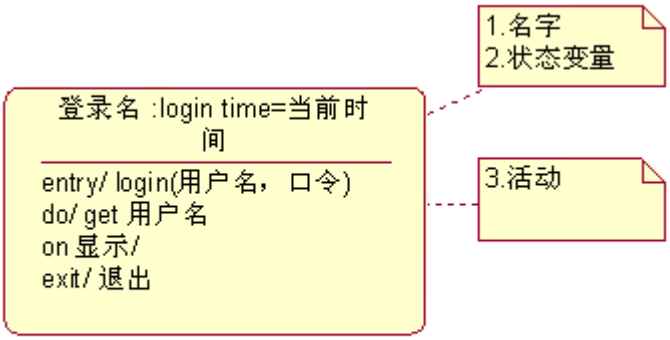
动态性表现在两个方面：交互和内部状态的改变。

交互描述对象的外部行为以及对象如何与其他对象交换信息（通过发送消息或链接和断链到其他对象）。

内部状态改变描述对象是如何改变其状态的，例如，对象的内部属性值。状态图用来显示对象对事件的反应以及对象状态的改变。

支票状态改变图示

当支票对象被创建时，它的状态为未付，当某人付了一张支票，则支票对象的状态从未付转移到已付。

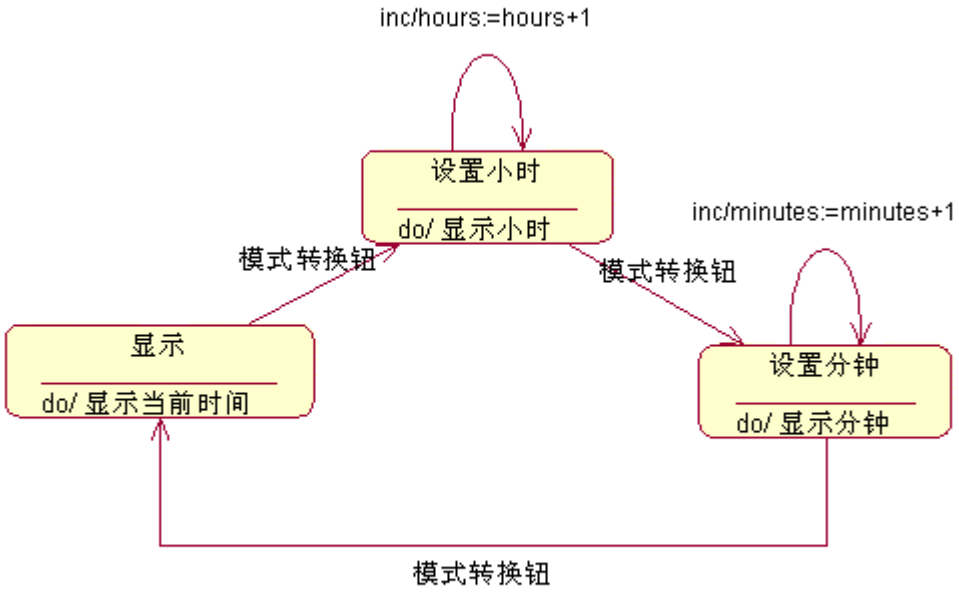


状态三个组成部分

一个状态一般包含三个部分，第一部分为状态的名称，如已付、未付、空闲、移动。第二部分为可选的状态变量的变量名和变量值。第三部分为可选的活动表，列出有关的事件和活动。

5.1.2 事件

“事件”指的是发生的且一起某些动作执行的事情。



电子表状态图示

该对象有三个状态：正常显示状态，两个设置时钟的状态（分别为小时和分钟）。

UML 中有四类事件：

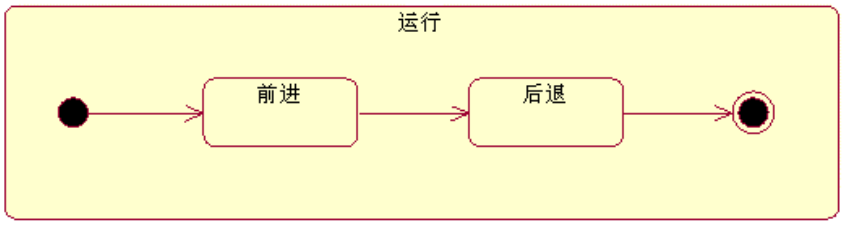
1) 条件成真：即状态转移上的守卫条件。
2) 收到另一个对象中的信号：信号本身也是一个对象。在状态转移中，表示为事件说明。这类事件也称作消息。
3) 收到另一个对象（或对象本身）的操作调用：在状态转移中，表示为事件说明。这类事件也称为消息。
4) 经过指定时间间隔：通常情况下，时间是从另一个指定事件（通常是当前状态的入口）或一给定时间段之后开始计算的。在状态转移中，表示为时间表达式。

了解事件的一些基本语义是很重要的。首先，事件是触发状态转移的触发器，并且一个事件只能被处理一次。如果一个事件可能触发多个状态转移，则只能有一个状态转移发生（具体触发哪一个是未定义的）。如果事件发生了，且状态转移的守卫条件为假，则事件被忽略（也不保存事件，如果守卫条件在后来成真则触发状态转移）。

类可以接收和发送消息，也就是说，可以调用操作或发送、接收信号。状态转移中的事件说明可以用来描述消息的发送和接收。当操作被调用时，操作开始执行并产生结果当一个信号被发送时，接收者接收该信号对象并使用它。信号是普通的类，但仅用来发送信号：它们表示系统中的对象间的发送单元。

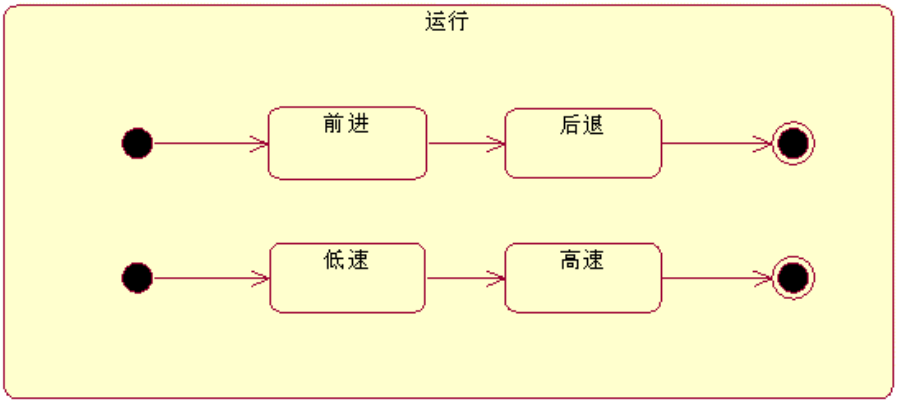
5.1.3 状态图与子状态

状态可能有嵌套的子状态，且子状态可以在另一个状态图。子状态又可分为两种：与子状态(and-substate)，或子状态(or-substate)。与子状态指的是一个状态可以有子状态，但是一次只能有一个子状态，如图。一辆处于运行状态的车，它的运行态可以有两个子状态：前进和后退，它们是或子状态，因为它们不能同时为真。嵌套的子状态可以显示在另一个状态图中，方法是在初始状态图中扩展运行状态。



或子状态

另一方面，运行态可能有多个并行的子状态（与子状态）：前进和低速，前进和高速，后退和低速，后退和高速。当一个状态有与子状态且它们中的几个可以同时为真时，表示一个状态既有与子状态也有或子状态，下图。与子状态也称为并行状态，可以用来抽象并行线程的状态。

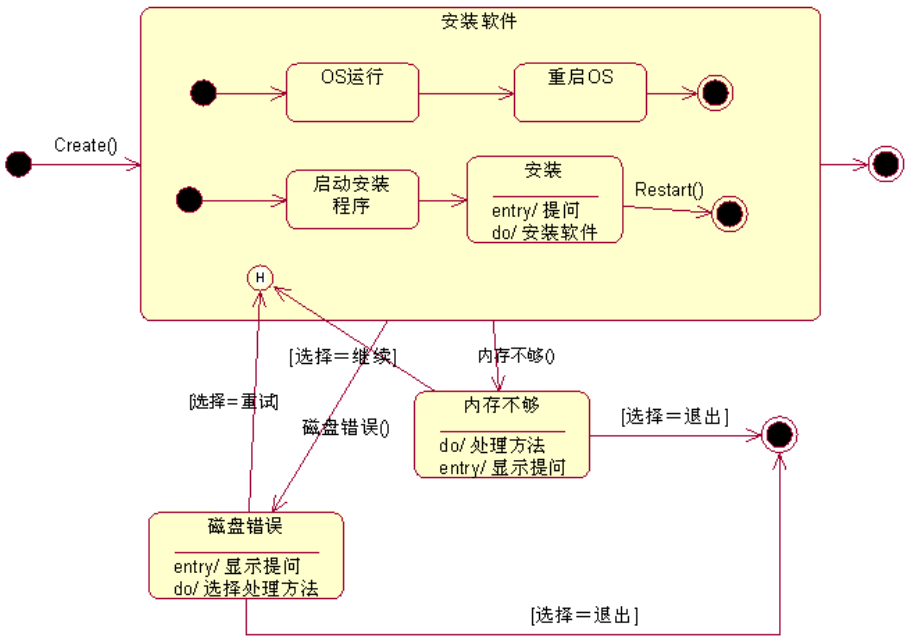


与子状态

5.1.4 历史指示器

历史指示器被用来存储内部状态，例如，当对象处于某一状态，经过一段时间后可能会返回到该状态，则可以用历史指示器来保存该状态。可以将历史指示器应用到状态区。如果到历史指示器的状态转移被激活，则对象恢复到该区域内的原来的状态。历史指示器用空心圆中放一个‘H’来

表示。可以有多个指向历史指示器的状态转移，但没有从历史指示器的状态转移。



历史指示器图示

5.2 活动图

活动图也是描述对象交互的，但侧重于工作的描述。当对象相互交互时，需要执行一些工作或活动。这些活动以及它们的出现顺序就是活动图所要描述的。

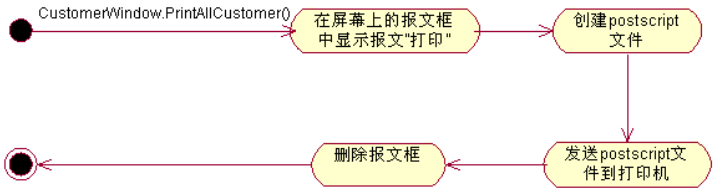
活的图(activity diagram)显示动作及结果。活动图着重描述操作（方法）实现中所完成的工作以及用例实例或对象中的活动。活动图是状态图的一个变种，与状态图的目的有一些小的差别，活动图的主要目的是描述动作（执行的工作和活动）及对象状态改变的结果。当状态中的动作被执行（不象正常的状态图，它不需指定任何事件）时，活动图中的状态（称为动作状态）直接转移到下一个阶段。活动图和状态图的另一个区别是活动图中的动作可以放在泳道中。泳道聚合一组活动，并指定负责人和所属组织。活的图是另一种描述交互的方式，描述采取何种动作，做什么（对象状态改变），何时发生（动作序列），以及在何处发生（泳道）。

活动图可以用作下述目的：

- 描述一个操作执行过程中（操作实现的实例化）所完成的工作（动作）。这是活动图最常见的用途。
- 描述对象内部的工作。
- 显示如何执行一组组相关的动作，以及这些动作如何影响它们周围的对象。
- 显示用例的实例是如何执行动作以及如何改变对象状态。
- 说明一次商务活动中的角色、工作流、组织和对象是如何工作的。

5.2.1 活动图的动作和转移

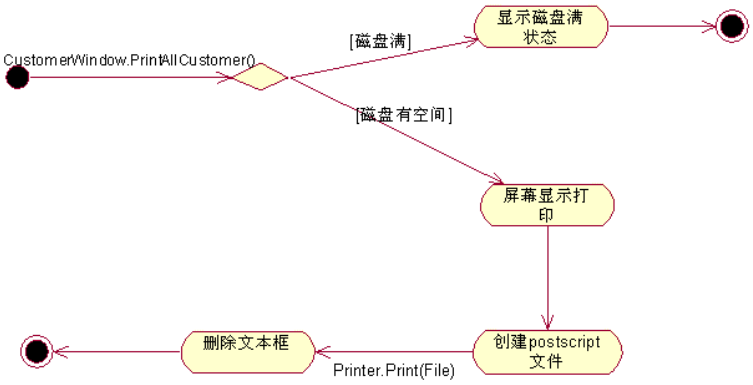
执行动作就会产生结果。可以用一组相关动作来描述操作的实现，然后将这些动作转换成代码行。



动作和转移

上图中当客户调用 PrintCustomer()操作，动作开始，第一个动作是在屏幕上的报文框中显示报文“打印”消息框，第二个动作创建 postscript 文件，第三个动作是发送 postscript 文件到打印机，第四个动作是删除报文框。转移是自动进行的，只要源状态中的动作被执行，转移就开始。

在动作内，一个文本串被用来说明动作或采取的动作。除了事件外，动作之间的转移的描述方法与状态图中所用的方法一致。事件可能只与从起点到第一个动作之间的转移联系在一起。动作之间的转移用箭头来表示，箭头上可能还带有守卫条件，发送短句和动作表达式。通常情况下，箭头上不带任何东西，表示只要动作状态中的所有活动一完成转移就开始。



多个转移

用守卫条件来约束转移，只有守卫条件为真时转移才可以开始。守卫条件的语法规则与状态图中的守卫条件的语法规则一致。用守卫条件决策用菱形符号来表示决策点。上图所示，一个决策符号可以有一个或多个进入转移，两个或更多的带有守卫条件的发出转移。正常情况下，出转移中的一个总是真。

当调用 PrintAllCustomer()操作，动作开始，在决策点有二个守卫条件发出转移，当磁盘满

时在屏幕上显示磁盘满状态，动作结束。当磁盘未满时在屏幕上显示打印动作，接着动作创建 postscript 文件，调用 Printer.print(File)操作，最后动作删除文本框动作结束。

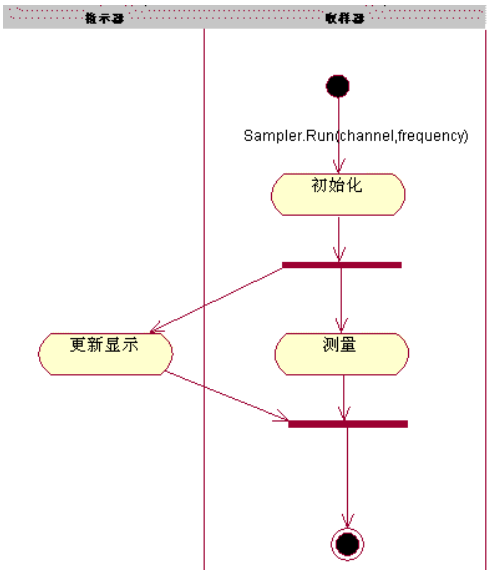
5.2.2 活动图的泳道

泳道用来组合活动。通常情况下，根据活动的功能来组合。

泳道有以下几个目的：

1) 直接显示动作在哪一个对象中执行。

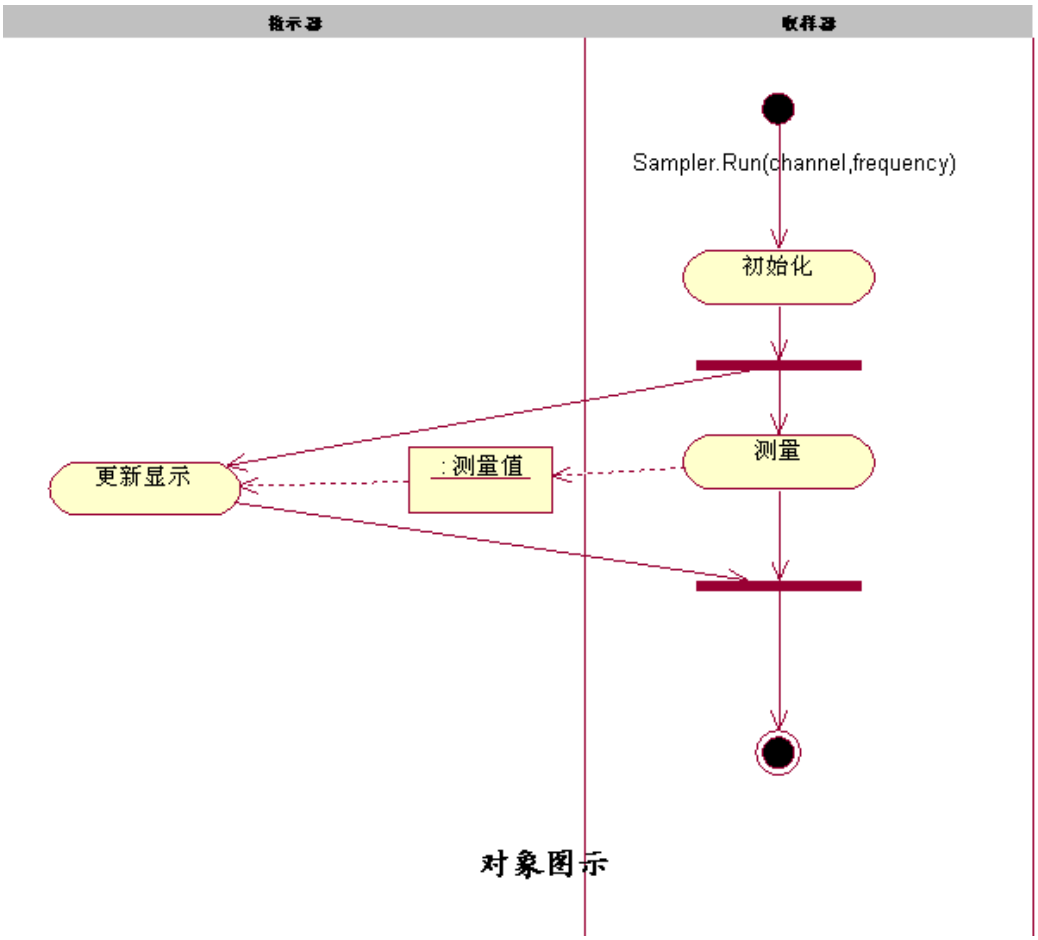
2) 显示执行的是一项组织工作的哪一部分。



当 Run 操作被调用时，发生的第一个动作是初始化。然后是并发执行的两个动作，更新显示和测量。动作更新显示在指示器内执行。动作初始化和测量在取样器内执行。

5.2.3 活动图的对象

对象可以在活动图中显示。对象可以作为动作的输入或输出，或简单地表示指定动作对对象的影响。对象用对象矩形符号来表示，在矩形的内部有对象名或类名。当一个对象是一个动作的输入时，用一个从对象指向动作的虚线箭头来表示；当对象是一个动作的输出时，用一个从动作指向对象的虚线箭头来表示。当表示一个动作对一个对象有影响时，只需用一条对象与动作间的虚线来表示。作为一个可选项，可以将对象的状态用中括号括起来放在类名的下面，如[planned]，[bought]，[filled]，等等。



对象图示

5.2.4 活动图的信号

有两个与信号有关的符号，一个表示发送信号，另一个表示接收信号。发送符号对应于与转移联系在一起的发送短句。同发送短句一样，发送和接收符号均同转移联系在一起。但是从图形角度看，转移又分为两种：发送信号的转移和接收信号的转移。

发送和接收符号可以同消息的发送对象或接收对象联系在一起。具体表示方法是从发送或接收符号画一条虚线箭头到对象。如果是发送符号，则箭头指向对象；如果是接收符号，则箭头指向接收符号。

6. 第六周：图书馆信息系统 UML 实例

6.1 需求

6.2 分析 - 用例图

6.3 建模及设计 - 类图

6.4 建模及设计 - 状态图

6.5 建模及设计 - 序列图

6.6 详细设计 - 类包

6.7 详细设计 - 详细的类图

6.8 详细设计 - 关键对象的状态图

6.9 详细设计 - 关键对象的序列图

6.10 详细设计 - 关键对象的协作图

6.11 详细设计 - 组件图

6.12 接口的设计

6.13 转成 java 并 encoding 实现

6.14 测试和配置 - 展开图

6.15 总结

殷连华

2002 年 8 月 2 日