返回总目录

目　　录

第 3 章　静态建模：用例和用例图.....2

3.1　用　例　图.....3
3.2　系　　统.....4
3.3　角　　色.....4
3.4　用　　例.....6
3.5　描 述 用 例.....10
3.6　测 试 用 例.....12
3.7　实 现 用 例.....12
3.8　小　　结.....14

第 3 章　静态建模：用例和用例图

用例模型是把应满足用户需求的基本功能（集）聚合起来表示的强大工具。对于正在构造的新系统，用例描述系统应该作什么；对于已构造完毕的系统，用例则反映了系统能够完成什么样的功能。构建用例模型是通过开发者与客户（或最终使用者）共同协商完成的，他们要反复讨论需求的规格说明，达成共识，明确系统的基本功能，为后阶段的工作打下基础。

用例模型的基本组成部件是用例、角色和系统。用例用于描述系统的功能，也就是从外部用户的角度观察，系统应支持哪些功能，帮助分析人员理解系统的行为，它是对系统功能的宏观描述。一个完整的系统中通常包含若干个用例，每个用例具体说明应完成的功能，代表系统的所有基本功能（集）。角色是与系统进行交互的外部实体，它可以是系统用户，也可以是其它系统或硬件设备，总之，凡是需要与系统交互的任何东西都可以称作角色。系统的边界线以内的区域（即用例的活动区域）则抽象表示系统能够实现的所有基本功能。在一个基本功能（集）已经实现的系统中，系统运转的大致过程是：外部角色先初始化用例，然后用例执行其所代表的功能，执行完后用例便给角色返回一些值，这个值可以是角色需要的来自系统中的任何东西。

在用例模型中，系统仿佛是实现各种用例的“黑盒子”，我们只关心该系统实现了哪些功能，并不关心内部的具体实现细节（比如，系统是如何做的？用例是如何实现的？）。用例模型主要应用在工程开发的初期，进行系统需求分析时使用。通过分析描述使开发者在头脑中明确需要开发的系统功能有哪些。

引入用例的主要目的是：

- 确定系统应具备哪些功能，这些功能是否满足系统的需求（开发者与用户协商达成共识的东西）。
- 为系统的功能提供清晰一致的描述，以便为后续的开发工作打下良好的交流基础，方便开发人员传递需求的功能。
- 为系统验证工作打下基础。通过验证最终实现的系统能够执行的功能是否与最初需求的功能相一致，保证系统的实用性。
- 从需求的功能（用例）出发，提供跟踪进入系统中具体实现的类和方法，检查其是否正确的能力。特别是为复杂系统建模时，常用用例模型构造系统的简化版本（也就是精化系统的变化和扩展能力，使系统不要过于复杂），然后，利用该用例模型跟踪对系统的设计和实现有影响的用例。简化版本构造正确之后，通过扩展完成复杂系统的建模。

用例模型由用例图构成。用例图中显示角色、用例和用例之间的关系。用例图在宏观上给出模型的总体轮廓，而用例的真正实现细节描述则以文本的方式书写。用例图所表示的图形化的用例模型（可视化模型）本身并不能提供用例模型必需的所有信息。也就是说，从可视化的模型只能看出系统应具有哪些功能，每个功能的含义和具体实现步骤必须使用用例图和文本描述（它记录着实现步骤）。

在进行定义系统，发现角色和用例、描述用例、定义用例之间的关系，验证最终模型

的有效性等工作时，需要建立用例模型。从另一个角度来说，有各种不同的人员需要使用用例模型。客户（或最终用户）使用它，因为它详细说明了系统应有的功能（集）且描述了系统的使用方法，这样当客户选择执行某个操作之前，就能知道模型工作起来是否与他的愿望相符合；开发者使用它，因为它帮助开发者理解系统应该作些什么工作，为其将来的开发工作（比如，建造其它的模型、架构的设计和实现）奠定基础；系统集成和测试的人员使用它，因为它可用于验证被测试的实际系统与其用例图中说明的功能（集）是否一致；还有涉及市场、销售、技术支持和文档管理这些方面的人员也同样关心用例模型。

用例模型也就是系统的用例视图。用例视图在建模过程中居于非常重要的位置，影响着系统中其它视图（比如，逻辑和物理架构）的构建和解决方案（满足基本功能需求）的实现，因为它是客户和开发者共同协商反复讨论确定的系统基本功能（集）。

开发者既可以把用例视图用于构建一个新系统的功能视图，还可以把已有的用例视图修改或扩充后产生新的版本，也就是在现有的视图上加入新功能（即在视图中加入新的角色和用例）。

3.1 用 例 图

在 UML 语言中，用例模型（也就是用例视图）是用例图描述的。用例模型可以由若干个用例图组成。用例图中包含系统、角色和用例等三种模型元素。图示用例图时，既要画出三种模型元素，同时还要画出元素之间的各种关系（通用化、关联、依赖），如图 3-1 所示。每种元素的描述方式将在 3.3 和 3.4 节讲述。

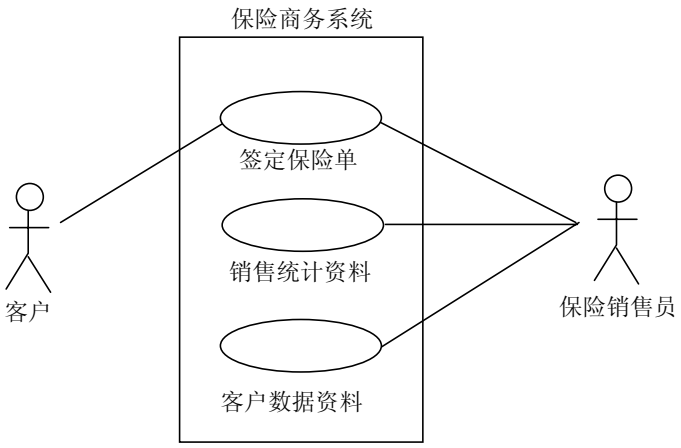

图 3-1 用例图示意

用例内容（即该用例所代表功能的具体实现过程）通常用普通的文字书写。在 UML 语言中，用例内容被看作用例元素的文档性质。另一描述用例内容的工具是活动图（关于活动图详见第五章）。用例图与活动图（比较正式的结构）相比，前者的描述更易被用户理解，也易于同其他用户交流信息。

3.2 系　　统

系统是用例模型的一个组成部分，代表的是一部机器或一个商务活动等等，而并不是真正实现的软件系统。系统的边界用来说明构建的用例模型的应用范围。比如，一台自助式售货机（被看作系统）应提供售货、供货、提取销售款等功能，这些功能在自动售货机之内的区域起作用，自动售货机之外的情况不考虑。准确定义系统的边界（功能）并不总是容易的事，因为严格地划分哪种任务最好由系统自动实现，哪种任务由其它系统或人工实现是很困难的。另外，系统最初的规模应有多大也应该考虑。一般的作法是，先识别出系统的基本功能（集），然后以此为基础定义一个稳定的、精确定义的系统架构，以后再不断地扩充系统功能，逐步完善。这样作的好处在于避免了一开始系统太大，需求分析不易明确，从而导致浪费大量的开发时间。

在建模初期，定义一些术语和定义是很有必要的。因为在描述系统、用例，或进行作用域分析（domain analysis）时，采用统一的术语和定义能够规范表述系统的含义，不致出现误解。当然，必要时可以随意扩充这些术语和定义。

用例图中的系统用一个长方框表示，系统的名字写在方框上或方框里面，方框内部还可以包含该系统中的用符号表示的用例。比如，图 3-1 中的保险商务系统，表示该系统的方框内还包含了三个用例（签订保险单、销售统计资料、客户数据资料）。

3.3 角　　色

角色（actor）是与系统交互的人或事。所谓“与系统交互”指的是角色向系统发送消息，从系统中接收消息，或是在系统中交换信息。只要使用用例，与系统互相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，则该人就是角色；与系统进行通信（通过用例）的某种硬件设备也是角色。

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。比如，在自动售货系统中，系统有售货、供货、提取销售款等功能，启动售货功能的是人，那么人就是角色，如果再把人具体化，则该人可以是张三（张三买矿泉水），也可以是李四（李四买可乐），但是张三和李四这些具体的个体对象不能称作角色。事实上，一个具体的人（比如，张三）在系统中可以具有多种不同的角色。比如，上述的自动售货系统中，张三既可以为售货机添加新物品（执行供货），也可以将售货机中的钱取走（执行提取销售款）。通常系统会对角色的行为有所约束，使其不能随便执行某些功能。比如，可以约束供货的人不能同时又是提取销售款的人，以免有舞弊行为。角色都有名字，它的名字反映了该角色的身份和行为（比如，顾客），注意，不能将角色的名字表示成角色的某个实例（比如，张三），也不能表示成角色所需完成的功能（比如，售货）。

角色与系统进行通信的收、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）。角色先发送消息给用例，初始化用例后，用例开始执行，在执行过程中，该用例也可能向一个或多个角色发送消息（可以是其它角

色，也可以是初始化该用例的角色）。

角色可以分成几个等级。主要角色（primary actor）指的是执行系统主要功能的角色，比如，在保险系统中主要角色是能够行使注册和管理保险大权的角色。次要角色（secondary actor）指的是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、通信、备份等）。比如，在保险系统中，能够检索该公司的一些基本统计数据的管理者或会员都属次要角色。将角色分级的主要目的是，保证把系统的所有功能表示出来。而主要功能是使用系统的角色最关心的部分。

角色也可以分成主动角色和被动角色。主动角色可以初始化用例，而被动角色则不行，仅仅参与一个或多个用例，在某个时刻与用例通信。

3.3.1 发现角色

通过回答下列的一些问题，可以帮助建模者发现角色。

- 使用系统主要功能的人是谁（即主要角色）？
- 需要借助于系统完成日常工作的人中谁？
- 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？
- 系统需要与哪些其它系统交互？其它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它应用软件。其它系统也分成二类，一类是启动该系统的系统，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？

在寻找系统用户的时候，不要把目光只停留在使用计算机的人员身上，直接或间接地与系统交互或从系统中获取信息的任何人和任何事都是用户。由于这里讨论的用例模型用于模型化一个商务（business）活动，所以角色通常指的是商务中的顾客，但是你要在头脑中保持清醒的认识——顾客的含义并不是计算机术语中的用户。

在完成了角色的识别工作之后，建模者就可以建立使用系统或与系统交互的实体（entities）了，即可以从角色的角度出发，考虑角色需要系统完成什么样的功能，从而建立角色需要的用例。

3.3.2 UML 中的角色

UML 中的角色是具有版类《角色》的类，该类的名字用角色的名字命名，用以反映角色的行为。角色类包含有属性、行为和描述角色的文档性质。UML 中用一个小人形图形表示角色类，小人的下方书写角色名字，如图 3-2 所示。图 3-2 左侧的矩形是具有版类《角色》的类（即角色类）的另一种图示方式，右侧的标准图示图标一般用在用例图中。

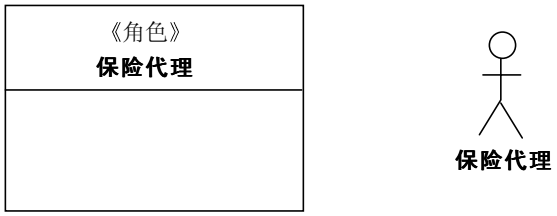

图 3-2　角色类的图示方法示例

3.3.3 角色之间的关系

由于角色是类，所以它拥有与类相同的关系描述（关于类中的关系的描述参见 4.3 节）。在用例图中，只用通用化关系描述若干个角色之间的行为。

通用化关系的含义是：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）。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。角色之间的通用化关系用带空心三角形（作为箭头）的直线表示，箭头端指向超类。图 3-3 示例的是保险业务中部分角色之间的关系，其中客户类就是超类，它描述了客户的基本行为（比如，选择险种），由于客户申请保险业务的方式可以不同，故又可以把客户具体分为二类：一类是用电话委托方式申请（用电话申请客户类表示），另一类则是亲自登门办理（用个人登记客户类表示）。显然，电话申请客户类与个人登记客户类的基本行为与客户类一致，这两个类的差别仅仅在于申请的方式不同，于是，在定义这两个类的行为时，基本行为可以从客户类中继承得到（从而不必重复定义），与客户类不同的行为则定义在各自的角色类中。

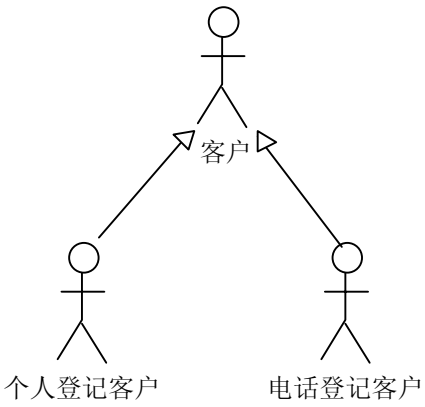

图 3-3　角色之间的通用化关系示例

3.4 用　　例

3.4.1 什么是用例

用例代表的是一个完整的功能。UML 中的用例是动作步骤的集合。动作（action）是系统的一次执行（能够给某个角色输出结果值）。与角色通信，或进行计算，或在系统内工作都可以称作动作。用例应支持多种可能发生的动作。比如，自动售货系统中，当顾客付款之后，系统自动送出顾客想要的饮料，这是一个动作；付款后，若需要的饮料无货，则提示可否买其它货物？或退款等等。系统中的每种可执行情况就是一个动作，每个动作由许多具体步骤实现。

用例具有以下的特征：

（1）用例总由角色初始化

用例所代表的功能必须由角色激活，而后才能执行。一般情况下，角色可能并没有意识到初始化了一个用例。换句话说，角色需要系统完成的功能，其实都是通过用例具体完成的，角色一定会直接或间接地命令系统执行用例。

（2）用例为角色提供值

用例必须为角色提供实在的值，虽然这个值并不总是重要的，但是能被角色识别。

（3）用例具有完全性

用例是一个完整的描述。虽然编程实现时，一个用例可以被分解为多个小用例（函数），每个小用例之间互相调用执行，一个小用例可以先执行完毕，但是该小用例执行结束并不能说这个用例执行结束。也就是说，不管用例内部的小用例是如何通信工作的，只有最终产生了返回给角色的结果值，才能说用例执行完毕。

用例和角色之间也有连接关系，用例和角色之间的关系属于关联（association），又称作通信关联（communication association）。这种关联表明哪种角色能与该用例通信。关联关系是双向的一对一关系，即角色可以与用例通信，用例也可以与角色通信。

用例的命名方式与角色相似，通常用用例实际执行功能的名字命名，比如：签定保险单、修改注册人等。用例的名称一般由多个词组成，通过词组反映出用例的含义，这也符合我们通常习惯的“见名知义”的约定。

注意，用例表示的也是一个类，而不是某个具体的实例。用例描述了它代表的功能的各个方面，也就是包含了用例执行期间可能发生的种种情况。比如：多种方案选择、错误处理、例外处理等。用例的实例（也是一种动作）代表系统的一种实际使用方法，这个实例通常又叫做脚本（scenario）。脚本是系统的一次具体的执行路线。比如，在自动售货系统中，张三投入钱币希望购买矿泉水，系统收到消息后将矿泉水送出的过程，就是一个脚本。又如，李四投币买矿泉水，但矿泉水卖完了，于是系统给出提示信息并把钱退还给李四，这个过程也是一个脚本。当然，还可以提供当需要买的东西缺货时，提示客户买其它东西而不立即退钱的脚本。

3.4.2 发现用例

实际上，从识别角色起，发现用例的过程就已经已开始了。对于已识别的角色，通过询问下列问题就可发现用例：

- 角色需要从系统中获得哪种功能？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种信息吗？
- 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？
- 还有一些与当前角色可能无关的问题，也能帮助建模者发现用例，例如：
- 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）？

3.4.3 UML 中的用例

UML 中的用例用椭圆形表示，用例的名字写在椭圆的内部或下方。用例位于系统边界的内部。角色与用例之间的关联关系（或通信关联关系）用一条直线表示，如图 3-4 所示。

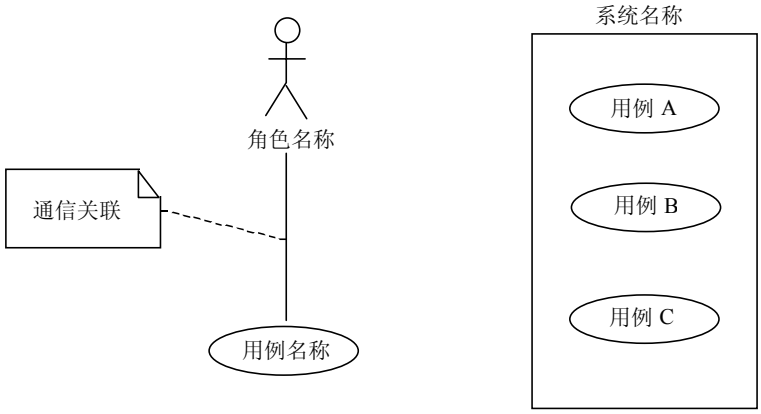

图 3-4　用例图示例

图 3-5 给出的是某自动售货系统的用例图。

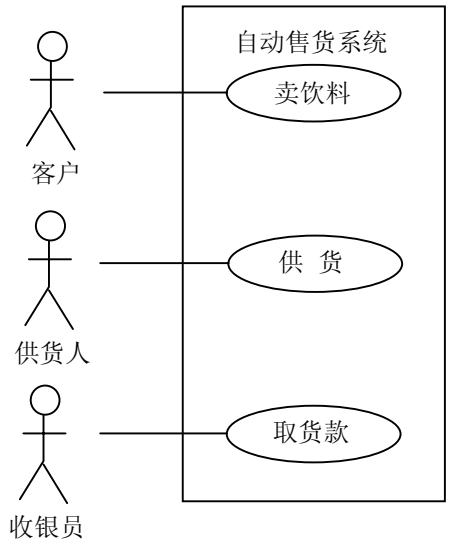

图 3-5　自动售货系统用例图

3.4.4　用例之间的关系

用例之间有扩展、使用、组合三种关系。扩展和使用是继承关系（即通用化关系）的另一种体现形式。组合则是把相关的用例打成包（package），当作一个整体看待。

1．扩展关系

一个用例中加入一些新的动作后则构成了另一个用例，这两个用例之间的关系就是通用化关系，又称扩展关系。后者通过继承前者的一些行为得来，前者通常称为通用化用例。后者常称为扩展用例。扩展用例可以根据需要有选择地继承通用化用例的部分行为。扩展用例也一定具有完全性。

由于用例的具体功能通常采用普通的文字描述（书写），因此，从文字中划分哪些行为是从通用化用例中继承而来的，哪些行为是在用例中重新定义的（作为用例本身的具体

行为），哪些行为是添加到通用化用例中（扩展通用化用例）的，都比较困难。

引入扩展用例的好处在于：便于处理通用化用例中不易描述的某些具体情况；便于扩展系统，提高系统性能，减少不必要的重复工作。用例之间的扩展关系可图示为带版类《扩展》的通用化关系，如图 3-6 所示。

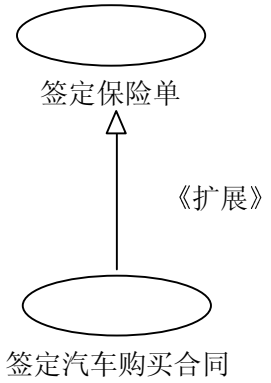

图 3-6　扩展关系图示

2．使用关系

一个用例使用另一个用例时，这两个用例之间就构成了使用关系。一般情况下，如果若干个用例的某些行为都是相同的，则可以把这些相同的行为提取出来单独作成一个用例，这个用例称为抽象用例。这样，当某个用例使用该抽象用例时，就好象这个用例包含了抽象用例的所有行为。

用例之间的使用关系被图示为带版类《使用》的通用化关系，如图 3-7 所示。

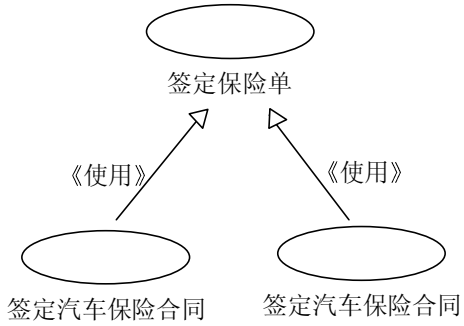

图 3-7　使用关系图示

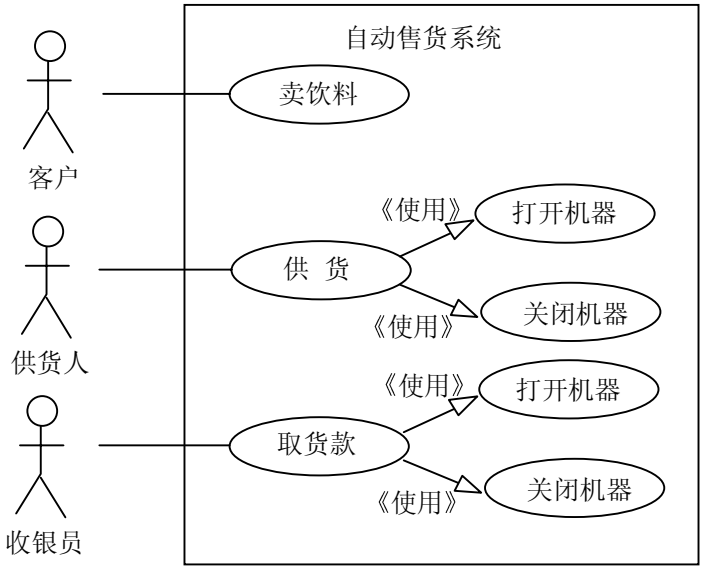

图 3-8　自动售货系统用例模型

比如，自动售货系统中，“供货”和“提取销售款”这两个用例的开始动作都是去掉

机器保险并打开它，最后的动作一定是关闭机器并加保险，于是可以从这两个用例中把开始的动作抽象成“打开机器”用例，把最后的动作抽象成“关闭机器”用例，那么“供货”和“提取销售款”用例在执行时一定要使用上述的两个抽象用例，它们之间便构成了使用关系。使用关系用带版类《使用》的通用化关系表示，如图 3-8 所示。

3.5 描 述 用 例

正如我们前面曾提到过的，图形化表示的用例本身不能提供该用例所具有的全部信息，因此还必需描述用例不可能反映在图形上的信息。通常用文字描述用例的这些信息。用例的描述其实是一个关于角色与系统如何交互的规格说明，该规格说明要清晰明了，没有二义性。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。

用例的描述应包括下面几个方面：

（1）用例的目标

用例的最终任务是什么？想得到什么样的结果？即每个用例的目标一定要明确。

（2）用例是怎样被启动的

哪个角色在怎样的情况下启动执行用例。比如：张三渴了，张三买矿泉水。“渴了”是使张三买矿泉水的原因。

（3）角色和用例之间的消息流

角色和用例之间的哪些消息是用来通知对方的？哪些是修改或检索信息的？哪些是帮助用例做决定的？系统和角色之间的主消息流描述了什么问题？系统使用或修改了哪些实体？

（4）用例的多种执行方案

在不同的条件或特殊情况下，用例能依当时条件选择一种合适的执行方案。注意，并不需要非常详细地描述各种可选的方案，它们可以隐含在动作的主要流程中。具体的出错处理可以用脚本描述。

（5）用例怎样才算完成并把值传给了角色

描述中应明确指出在什么情况下用例才能被看作完成，当用例被看作完成时要把结果值传给角色。

需要强调的是，描述用例仅仅是为了站在外部用户的角度识别系统能完成什么样的工作，至于系统内部是如何实现该用例的（用什么算法等）则不用考虑。描述用例的文字一定要清楚，前后一致，避免使用复杂的易引起误解的句子，方便用户理解用例和验证用例。

用例也可以用活动图描述，即描述角色和用例之间的交互（如图 3-9 所示）。活动图（第五章中详叙）中显示各个活动的顺序和导致下一个活动执行的决策（decision）。对用户来说，用例视图更易于使用。

对于已经包含完全性和通用性描述的用例来说，还可以再补充描述一些实际的脚本，用脚本说明用例被实例化后系统的实际工作状况，帮助用户理解复杂的用例。注意，脚本描述只是一个补充物，不能替代用例描述。

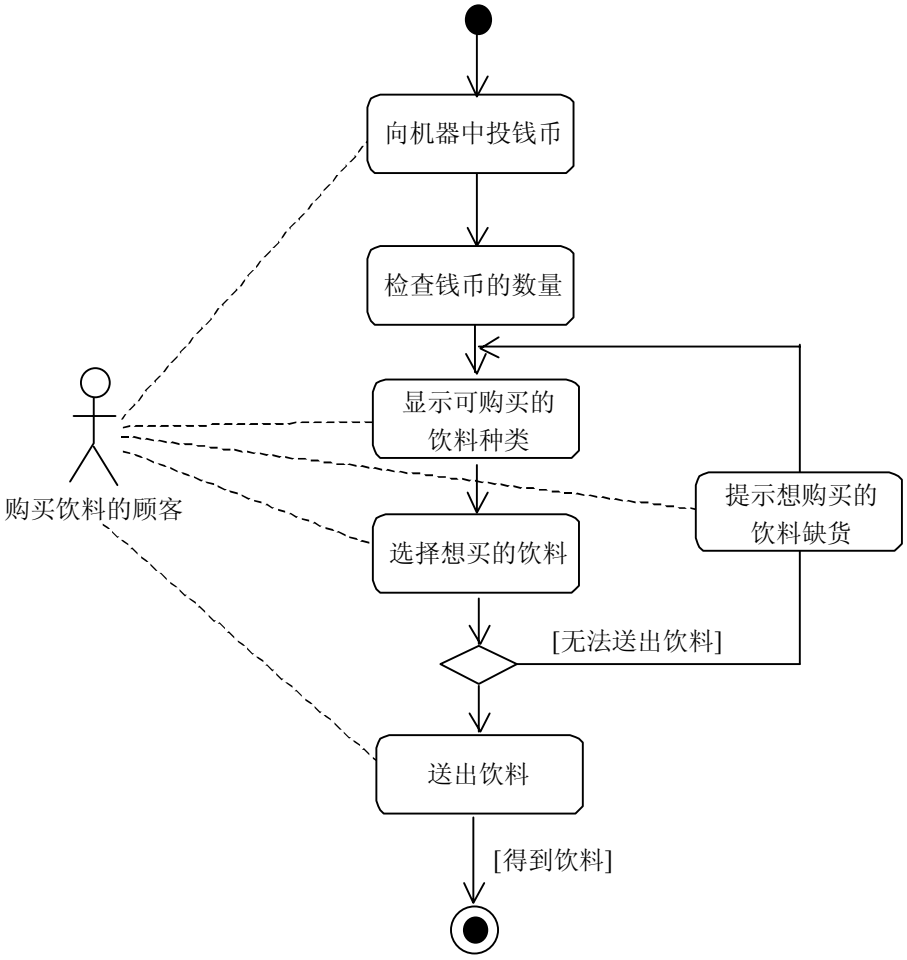

图 3-9　活动图示例

对某个用例的描述完成之后，可以用一个具体的活动跟踪一下，检查用例中描述的关系能否被识别。在执行这个活动时，可以通过回答下列问题找出不足之处。

- 用例中的所有角色与该用例都有关联关系吗？
- 若干个角色的通用行为与基类角色（从若干个角色的行为中抽象出的最普通的那部分）的行为是否相似？
- 表示同一个活动流的多个用例之间是否存在相似性，它们是否能用使用关系描述为一个用例。
- 用例扩展关系中描述的特殊情况存在吗？
- 有没有存在无任何通信关联的角色或用例？如果有，那么用例一定存在问题，否则为什么还要这个角色。
- 有没有遗漏与需求的功能相对应的用例？如果有，那么就要新建一个用例。

注意，不要把所有的用例建好之后，再去识别用例中的关系是否正确。这种做法有时会导致错误。

3.6 测 试 用 例

用例可用于测试系统的正确性和有效性。正确性表明系统的实现符合规格说明。有效性保证开发的系统是用户真正需要的系统。

有效性检查一般在系统开发之前进行。当用例模型构造完成后，开发者将模型交给用户讨论，由用户检查模型能否满足他们对系统的需求。在此期间，各种问题和想法还会产生，比如，修改用例的不足之处，或在用例中添加新功能。最终，用户和开发者之间对系统的功能达成共识。有效性检查也可以在系统测试阶段进行，如果发现了系统不能满足用户需求的问题，那么整个工程或许会要从头重来。

正确性测试保证系统的工作符合规格说明。常用的测试方法有二种，一种是用具体的用例测试系统的行为，又称“漫游用例”；另一种是用用例描述本身测试，或称定义测试。这两种方法相比，第一种方法更好一些。

第一种测试方法的基本思想是用人模拟系统的行为。大致过程如下：指定一个人扮演具体用例中的角色，另一个人扮演系统。扮演角色的人首先说出角色应传给系统的消息，然后系统接收消息开始执行，在系统执行过程中，扮演系统的人说出他正在做的工作是什么。通过角色模拟，开发者可以从扮演者那里得知用例的不足之处。比如，发现哪些情况漏掉了，哪些动作描述得还不够详细。扮演系统行为的人洞察力越强，用例测试的效果就越好。因此，可以让每个人分别多次扮演各个角色或系统，从而为建模者提供更多的信息，减少用例描述的遗漏和含糊不清。当所有的角色都被扮演，且所有的用例都以此方式执行过了，那么对用例模型的测试就算完成了。

3.7 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，而协作又被细化为用若干个图。协作的实现用脚本描述。具体内容是：

（1）用例实现为协作

协作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）。协作的图示符号为椭圆，椭圆内部或下方标识协作的名字。

（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）之间的关系和交互。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，才能反映协作状况。

（3）协作的实例——脚本

前文已经讲过，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，而忽略完成该角本的具体细节，从而达到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信）。

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所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，一个类可以用来实现多个用例，也就是，一个类可以集成多种角色。比如，自动售货系统中，负责“供货”和“提取货款”的角色就可以定义为同一个类（当然，从安全角度讲，最好不要定义为同一个类）。

用例和它的实现（即协作）之间的关系可以用精化关系表示（图示为带箭头的点画线），也可以用 CASE 工具中的不可见的超链实现。使用 CASE 工具中的超链，能方便地将用例视图转换为协作或脚本。图 3-10 表示一个用例的实现，从图中可以看出若干个类加到了协作中。为了实现用例，用例描述中的每个步骤的职责还需转换为类与类之间的协作（用关系和操作描述）。

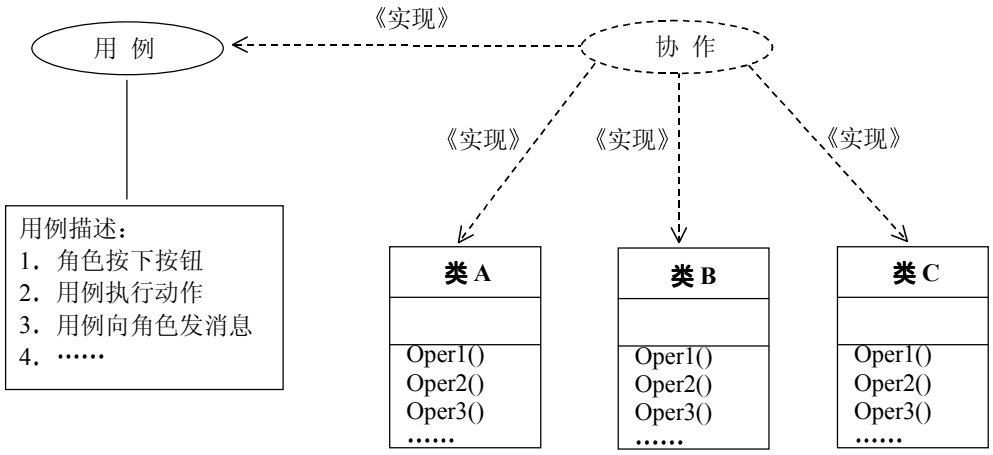

图 3-10　协作实现的类示例

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。

另外，UML 的创始人 Jacobson 定义了三种类型的版类对象类（stereotype object types）：边界对象（又称接口对象）、控制对象和实体对象。还可以利用这三个对象类描述用例的实现。每个对象类能胜任的职责是：

（1）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。

（2）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。

（3）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，存储在数据库存中，实体对象也可以出现在多个用例中。

上述三个对象类各有自己的图标，可以用来图示协作和类图。在定义了不同的对象类和详细说明了一个协作之后，也可以用一个具体的活动测试用例，确认对象的实现方式，以便这些类重用在其它用例中，Jacobson 将这种开发过程称为用例驱动的开发过程。

实现用例有多种不同的方法。不同的实现方法为实现用例的类分配不同的功能。有些方法采用先建立作用域分析模型（用于显示所有的作用域类及其它们之间的关系），然后才处理用例，为分析模型中的类分配应完成的功能，在这个过程中，有时会修改分析模型或在模型中添加新的类。另外一些方法则把用例当作发现类的基础，在为类分配应完成功能的同时，逐步建立作用域分析模型。总之，不论采用何种方法，对开发者来说，要明白这种实现工作将是反反复复的过程。也就是说，当利用用例为类分配功能时，或许可以发现类图中的错误和遗漏，这时，必须返回去修改类图或者建一个新类来反映用例的本意。当然，在某些情况下，可能必须修改用例图，因为初期建造的用例图不一定完全正确地描述系统的功能，随着开发者对系统功能的深入理解，原图之中的不足之处就可能暴露出来。

最后一点需要说明的是，并不是所有面向对象的方法都提供用例图。比如，有的方法只提供类和对象等静态结构，忽略系统开发过程中功能性和动态性方面的描述。

3.8 小　　结

用例模型是描述系统基本功能的工具。用例模型用角色、用例和系统描述。角色代表外部实体，比如用户、硬件或另一个与系统交互的外部系统。角色启动用例并与其通信，执行中的用例是一个动作序列。用例一定要给角色传递一个确切的（tangible）值。用例的具体执行过程用文本文件描述。角色和用例都是类，角色通过关联与一个或多个用例相连接，角色和用例都有通用化关系，这样超类中的通用行为可以被一个或多个专门化的子类继承。用例模型（即用例视图）由一个或多个 UML 用例图描述。

用例图用协作实现。协作描述了类（或对象）、类与类之间的关系和交互（显示类之间怎样交互才能实现一个具体的功能）。协作用活动图、协作图和交互图描述。实现用例时，用例中的每个动作的责任交给协作中的类完成，通常是由类中的操作完成的。脚本是用例或协作的实例，用于显示一次具体的执行过程。当把脚本看作用例的例子时，脚本仅仅描述用例和外部角色之间的交互，不涉及交互的具体实现细节；当把脚本看作协作的实例时，则实现系统内部类（或对象）之间的交互，即实现具体的实现细节。