UML 用例图

Engineering Notebook
C++报告
98.11-98.12

翻译：杨健

UML 的一个重要部分是画用例图的功能。在工程的分析阶段用例图被用来鉴别和划分系统功能。它们把系统分成动作者(actor)和用例。

动作者(actor)表示系统用户能扮演的角色(role)。这些用户可能是人，可能是其他的计算机，一些硬件，或者甚至是其它软件系统。唯一的标准是它们必须要在被划分进用例的系统部分以外。它们必须能刺激系统部分，并接收返回。

用例描述了当动作者之一给系统特定的刺激时系统的活动。这些活动被文本描述。它描述了触发用例的刺激的本质，输入和输出到其他活动者，和转换输入到输出的活动。用例文本通常也描述每一个活动在特殊的活动线时可能的错误，和系统应采取的补救措施。

例如，考虑销售系统的一个网点。活动者之一是客户和另一个是销售店员。这正是一个来自该系统的用例：

用例 1：销售店员结算一件商品

1. 顾客放商品到柜台
2. 店员用 UPC 读取器扫描商品上的 UPC 码
3. 系统在数据库中查找 UPC 码获得商品的描述和价格
4. 系统发出可听见的蜂鸣
5. 系统用声音宣布商品的描述和价格
6. 系统加价格和商品类型到当前发票
7. 系统加价格到调节税小计

错误情况 1：UPC 代码不能读取

如果在第 2 步之后，UPC 码无效或没有正确读取，系统发出一声巨响。

错误情况 2：商品不在数据库中

如果在第 3 步之后，不能在数据库中找的该 UPC，闪动终端上的“手动输入”按钮。接收店员输入的价格和税码，设商品描述为“未知商品”。进入第 4 步。

无疑销售系统的一个网点有比这更多的用例。的确，一个复杂系统，这数目可达上千。系统的全部功能可象这样在用例中被描述。这使得用例成为很强的分析工具。

画用例图

图 1 显示在 UML 图标形式下上面的用例看起来可能的样子。用例自身被画成一个椭圆。活动者被画成一个小人。活动者被用线连到用例。

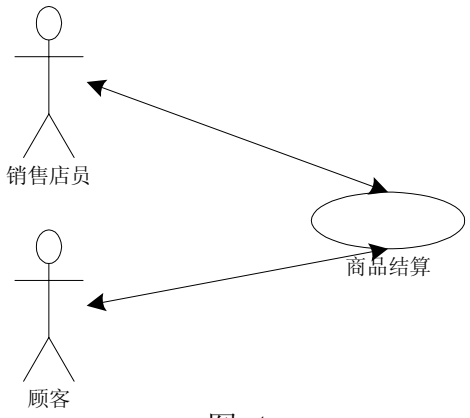

图 1

显然这是一个漂亮而简单的标记;但还不止这些。用例和活动者图标能被组装到大型的“系统边框图”。这种图在一个矩形中显示系统中所有的用例。矩形的外部是所有该系统的活动者，并且它们被线连到用例。这箱子表示系统的边框；也就是，它显示了一个特定系统内的所有用例。箱内的每一件事物都是系统的一部分。箱外的每一件事物都是系统的外部。见图2。

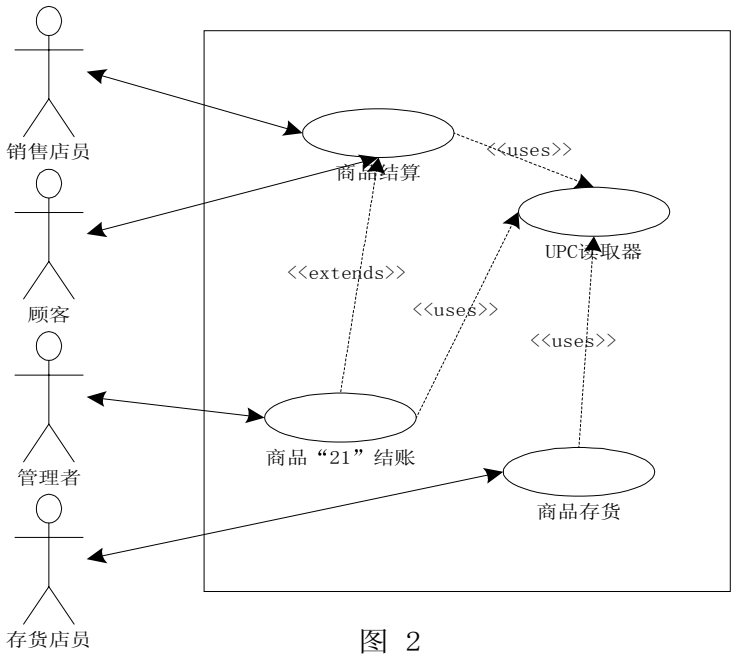

图 2

除了显示系统边框以外，还有更多的活动者，和更多的用例；这图也显示了用例之间的一些关系。这些关系是<<uses>>和<<extends>>。使用(uses)关系是最简单的关系。注意它在图2中出现了两次，一次从商品结算到UPC读取器，另一次从商品存货到UPC读取器。用例UPC读取器描述活动者和系统的行为，当活动者用UPC读取器读取产品条码时。

在我们的第一个用例中，读取UPC的活动需要出现在商品结算用例的描述中。然而，既然存货店员也要读取UPC条码，当统计货架上的货物时，这相同的活动必是商品存货用例的一部分。我们可以使用<<uses>>关系表示其同时属于两个用例，而不必为这行为做两次描写。这样做后我们可以象下面这样修改商品结算用例的描述：

用例 1：销售店员结算一件商品

1． 顾客放商品到柜台

2． <<uses>>UPC 读取器

3． 系统在数据库中查找 UPC 码获得商品的描述和价格

4． 系统发出可听见的蜂鸣

5． 系统用声音宣布商品的描述和价格

6． 系统加价格和商品类型到当前发票

7． 系统加价格到调节税小计

如此，<<uses>>关系非常象一个函数调用或一个子过程。以这种方式使用的用例称为抽象用例，因为它不能单独存在，而必须被其它用例使用。

另一个有趣的关系是<<extends>>关系在商品结算和商品“21”结算之间。在许多店铺，21 岁以下的销售店员不允许结算酒类商品。当一个 21 岁以下的销售店员看到一个就类商品，这店员叫“21”在 P.A.系统。不久一个管理者走过来读取这酒类商品的 UPC 码。这表示了一个变化，用例可能要进行两条路径选择。

首先，我们可以加“if”语句到商品结算用例，那么现在看起来是这样的：

用例 1：销售店员结算一件商品

1． 顾客放商品到柜台

2． If 商品是酒类

2．1．呼叫“21”在 P.A.系统

2．2．等待管理者

2．3．管理者<<uses>>UPC 读取器

2．4．进到步骤 4

3．<<uses>>UPC 读取器

4．系统在数据库中查找 UPC 码获得商品的描述和价格

5．系统发出可听见的蜂鸣

6．系统用声音宣布商品的描述和价格

7．系统加价格和商品类型到当前发票

8． 系统加价格到调节税小计

虽然这样能很好的工作，但还有一些不利之处。还记得 Open Closed Priniple(OCP)吗？(参考 1996 年的 Engineering Notebook)那是说，我们不想去建立当需求改变时就必须修改的软件。在一个设计良好的软件，一个需求改变将引起增加新代码而不是去修改老的代码。同样的规则被用在用例的功能说明。当需求改变，我们想去增加新的用例，而不改变老的已存在的用例。

因此，我们更愿意使用<<extends>>关系，而不在用例中使用“if”语句。这关系允许我们规定一个新的用例装载命令覆盖和修改需要扩展的用例。因此在图 2 中结算商品“21”的用例覆盖和扩展了结算商品的用例。这结算商品用例的文本看起来可能象下面：

用例 2：结算商品“21”

1． 用以下替代商品结算用例的步骤 2：

1．1．呼叫“21”在 P.A.系统

1．2．等待管理者

1．3．管理者<<uses>>UPC 读取器

这就达到了允许增加新特征到用例模型而又不修改已存在的老用例的目的。

扩展点

注意到用例结算商品“21”直接提到用例商品结算。这是很不幸的。如果我们想扩展另一些用例的同样需要修改的步骤，那该怎么办？我们不得不要有和被扩展用例同样数量的扩展用例。而所有的扩展用例几乎是相同的。

对这问题我们能采取的补救措施是在被扩展用例中增加扩展点。扩展点是一个简单的标识名用来标识被扩展用例中用于调用扩展用例的位置。因此，我们的两个用例看起来可能象下面这样：

用例 1：销售店员结算一件商品

1. 顾客放商品到柜台
2. XP21：店员用 UPC 读取器扫描商品上的 UPC 码
3. 系统在数据库中查找 UPC 码获得商品的描述和价格
4. 系统发出可听见的蜂鸣
5. 系统用声音宣布商品的描述和价格
6. 系统加价格和商品类型到当前发票
7. 系统加价格到调节税小计

用例 2：结算商品“21”

2. 用以下替代被扩展用例的 XP21
   - 2．1．呼叫“21”在 P.A.系统
   - 2．2．等待管理者
   - 2．3．管理者<<uses>>UPC 读取器

文本管理

文档维护的一个问题是当单独一个需求改变，可能影响到功能说明书中的许多地方。的确，有时在一个功能说明书中的冗余信息量可能很高，引起重大的维护问题。用例和它们关系的目的是去管理功能说明书中的文本描述，从而减少冗余信息。通过合理的构造用例和它们的关系，你能建立永远都不需要修改多于一处的功能说明书。对于大工程，这可能是一个巨大的收获。

用例的结构不是它要表示的软件的结构

用例图并没告诉你许多东西。它们传达了用例的结构，但没有告诉你它们内部的文档。同样，当它们被和文本描述分开时，它们不再是那么有趣的文档了。最多这图提供了一组相关的漂亮路标，那么读者能重构给定情况的整个文档，通过追踪<<uses>>和<<extends>>关系，在前者中插入文档，在后者中修改文档。

结论

当划分系统功能时，用例是强有力的分析工具。用例关系和协作图帮助分析用例结构，那么它们的文本描述装载最小的冗余信息；因而使整个文档更容易维护。但用例不是设计工具。它们没有规定最终软件的结构，它们也没有隐含了任何类和对象的存在。它们是以完全与软件设计相分离的形式书写的纯粹的功能描述。