需求分析与系统设计的面向对象推导过程

火龙果软件 发布于 2014-3-20

几年前写的了，这两天整理东西的时候又给翻出来了，当时是公司让给我给设计人员讲讲如何写面向对象的设计说明书，所以临时东拼西凑的弄了这么个东西，毕竟是用于内部培训的，有些东西都是直接从网上整段 COPY 的，最多就是用自己的话又修饰了一遍，在此说明一下，各位看到的时候，莫过多纠结于此 。

一. 引言

1.1 文档概要

概要很简单...

1.2 编写目的

解释设计说明书里应该写些什么，在写设计说明书之前应该给我什么，写完了设计说明书应该达到什么样的效果，或者换个说法，写完了设计说明书我能给代码开发人员什么。

1.3 背景

背景很复杂

1.4 定义

类型	名称	定义
缩写词	RUP	Rational Unified Process，统一软件开发过程，由IBM提出的基于面向对象且适应于大型项目的程序开发方法论
	OO	Object Oriented,面向对象
	XP	Extreme Programming极限编程 。一种认为轻量的软件开发方法论，强调架构，文档不如直接编程来得直接。适应于小型项目的开发实践
专门术语		

1.5 预期的读者和读者建议

实体名称	商品		
实体描述	每个商品都经有上架（录入商品信息），预购（放入购物车），卖出，退货和下架几个状态，详细请参看商品状态图		
属性名称	类型	精度	说明(属性的业务含义及业务规则)

商品编号	字符	12	商品类别编号(3位)+商品购入年份(4位)+流水号(5)位
商品分类	数字	3	商品的分类
名称	字符	100	商品的名称
价格	数字	12	商品的价格
折扣	数字	6	折扣价
产地	字符	100	商品产地
简介	字符	1000	商品的内容简介，上架时录入
状态	字符	1	商品的状态，可参看商品状态图

二．参考文献

《系统分析与设计》 JohnW.Satzinger Robert B.Jackson Stephen D.Burd 著朱群雄 汪晓勇 等译 机械工业出版社

《大型软件体系结构：使用 UML 实践指南》［美］Jeff Garland Richard Anthony 著 叶俊民汪望珠 等译 电子工业出版社

《设计模式精粹》［美］Alan Shalloway & James R.Trott 著熊节 译 清华大学出版社

《编写有效用例》［美］Alistair Cock Brun 著 机械工业出版社

《用例分析技术（原书第二版）》机械工业出版社

《OO 系统分析员之路 -- 用例分析系列》 coffeewoo

《RUP 文档模型》

三．内容

3.1 需求分析

虽说本文档是为设计说明书的编写进行服务。但我觉得如果要想明白设计阶段能够做些什么，就得知道前一阶段能够给我输入些什么，设计的前一阶段是需求。所以我们得搞明白需求能给我们输入些什么。当然要想搞明白需求能给我们输入些什么。就又得搞明白需求的最终文档是怎么一步一步分析出来的。要不然直接给我们一堆文字外加一堆 UML 图。我们也搞不明白究竟是些什么东西。所以本文档有必要从需求分析开始入手，一步一步推导到概要设计说明。详细设计本文档不作考虑。

本文档采用的是面向对象的推导过程。而非面向过程的推导过程。所以在开始推导之前先给大家测一下，看看你到底是面向对象的潮流人群呢，还是面向过程的遗老遗少。

如果你的分析习惯是在调研需求时最先弄清楚有多少业务流程，先画出业务流程图，然后顺藤摸瓜，找出业务流程中每一步骤的参与部门或岗位，弄清楚在这一步参与者所做的事情和填写表单的结果，并关心用户是如何把这份表单传给到下一个环节的。那么很不幸，你还在干着面向过程的事情。

如果你的分析习惯是在调研需求时最先弄清楚有多少部门，多少岗位，然后找到每一个岗位的业务代表，问他们类似的问题：你平时都做什么？这件事是谁交办的？做完了你需要通知或传达给谁？做这件事情时你都需要填写些什么表格？.... 那么恭喜你，你已经跟上了时代的潮流，进入了 OO 人群的行列！

当然是 OO 的先不要忙着得意，前路漫漫，还有的你走。是遗老遗少的也不用忙着灰心，改进为时不晚。

本文档主要采用了 RUP（统一软件开发过程）的思想。而非 XP（极限编程）的思想。采用 RUP 并不是说 RUP 就比 XP 好。这两个本身就无所谓谁好谁坏，因为在适应的对象上两者是完全不同的。对于中小型公司和中小型软件来说，XP 是非常有效的管理方法，它能大大降低管理、开发成本和技术风险。不过要是对于大公司和大型项目来说，XP 就不适用了，这时 RUP 却表现的非常出色。你能想象波音公司用 XP 的方法来开发 747 是一个什么情形吗？先不要管飞机将来是什么样子，反正先造一架出来，出问题了，摔了找找原因，改进改进，重构一下，再造一架.... 再摔了，没关系，咱们不怕变更，再造就是了。如果真是这样，恐怕波音公司早挂了。那 XP 什么情况下适用呢？如果你是一个杂货店的老板，不知道什么样的商品受欢迎，没关系，我们可以先各进一小批货，卖上一段时间，然后看顾客的反应，受欢迎的货品我们就多进一些，不受欢迎的嘛就少进一些或者不进，顺便再和顾客多多交流一下，直接问问他们喜欢什么，不喜欢什么。不断的改进，不断的完善。我想最后一定会顾客盈门的。但是假如这时这个老板非得先进行一下什么市场调研，再做个什么商

业方案，顺便再搞搞风险评估，最后再进行下客户关系研究，消费曲线分析。猛一点的，再加上个消费心理问卷.... 估计还没开业，就破产了。

在本文档中采用 RUP 的过程仅仅是因为他更全面，更严谨一些。另外对于一些面向对象里的基础理论，无论是对于 RUP 还是 XP 都还是适用的。在我看来 RUP 与 XP 的区别不在技术，仅在对于过程中对精细程度的把握尺度与迭代方式的不同而已。

注：使用的 RUP 的推导与分析和迭代过程，文档并不一定采用 RUP。

本文档主要讲的是推导过程，故对过程中的细节不作深入的描述。例如如何获取 “有效的” 用例以及获取用例时，所要考虑到的市场因素和风险因素，以及在需求分析和系统设计时的相关管理事项（例如需求会议召开，怎样与客户交流，进度的控制，成本）等不作进一步描述。相关内容可自行参考有关书籍和论文。

最后再啰嗦一句：我本身对于面向对象的理解也不是很深刻，因时间紧迫，还有很多东西未完全吃透，故错误肯定是在所难免的。如果不怕被误导那么你可以放心大胆的往下看了，如果怕误导，还是劝你就此打住比较合适。

下面我们正式开始推导过程。

寻找执行者与设计用例

RUP 是一种用例驱动的迭代开发过程。那么何为用例驱动呢？这个很好理解，看字面就能猜个八九分，就是一切从用例开始，然后一步一步推导出我们想要的结果。那么用例又是什么呢？这是我们提出来的第一问题

什么是用例 ？

我们先不忙着做出解答。先假设一段场景，AA 公司是一家转购公司，他们从不同的供货商那里进货后，然后再卖出去。最近他们想让我们去为他们开发一套系统，我们接到任务后给他们公司打了一个电话，他们为我们安排了一位销售经理进行电话交流。我们把这次交流的结果整理后，得到了如下的一段描述。

目前随着网络的普及，很多的上班族趋向于采用使用网络这种做在家里就可以购物的方式。所以我们公司准备开发一个网上的订单平台。客户可以 通过网络查看我们发布的产品并选购，然后通过银行或者其它付款方式付款给我们，我们则通过快递公司或者其它方式将产品直接送到客户填写的寄送地址。客户可以退货。并要求重新进货。当然可能需要额外的支付一笔费用。

看到这样一段描述，你可能觉得大致清楚要做的是什么样的东西了，但对于构建一个系统来说，这样的描述还差太多的东西，其中有很多的细节并没有表述出来。然而不幸的事，往往在前期你也就只能从客户那里挖出来这些东西了，尤其是你在调研时仅仅只和一个岗位的相关人员进行了交流。当然也许你的客户够专业，你们的交流使你获得了比这多的多的消息， 但无论如何， 你的经验会告诉你：你第一次获得的东西永远都不可能是完备的，你也永远不要奢望第一次就能讲全部的需求都搞到手。

那么获得了这样一段并不完备的描述后，我们应该做些什么呢？继续追问？NO，能获得这样的需求在目前看来我们已经做的足够好了。 自己去完善客户的需求并交付开发人员去开发？NO，要知道使用系统的是客户而不是你。也许你可以为客户设计一套流程，或者一个订单的样式，但那可能并不是客户所需要的，假如你认为客户不大会授权你去为他们在某些方面进行设计，你就千万不要去自作主张。当然绝大多数客户都是懒惰的，大部分的情况是他们非常乐于让我们先帮他们思考，然后他们在我们思考的基础上给出些意见就行了。是的，这就是我们下面要做的事－－分析这段需求，找出这段需求中的隐藏信息，分析出他们要做的这套系统究竟会涉及到哪些人（涉众），并简单的为每个涉众列出你所能想到的他所要做的事情，然后将这些以易于客户理解的方式告诉他们所有相关的人员，然后她们就会在此基础上提出她们自己的想法，你记录下她们的这些想法，对这些想法进行分析后将它们补充到你的需求里，之后再将你这个最新的分析后的需求提交给客户审查，如此迭代，直到客户说 OK，我要的就是这个东西。

如果你没有和系统的所有相关者（涉众）进行过交谈，并获得了有用的信息。永远不要说你已经做好了需求

下面我们就具体讨论一下如何去分析这段需求。此时用例开始登场。当然本文档讲的是一个推导过程，不是一个迭代过程，故在下面的描述中我们不考虑迭代的因素，所有的分析都是假设在一个很顺利的环境中进行，这样也许会让你觉得条理更清晰一些（包括上面那段不完备的描述在这里我们都假设他已经是很完备的了）。

当然在继续往下介绍之前，为了更加专注于需求的功能要求，我会回避掉前面提到的一个概念 “涉众”，而改用“执行者” 代替。执行者是涉众的一个子集，涉众是同你要开发的系统相关的一切人或物，而执行者则是同你要开发的系统直接接触的一切人或物。从这两个定义上我们可以看出，涉众可能不是这个系统的直接操作者，例如一个用于控制机床运作的系统，财务部门可能不会去使用，但因为项目的款项是由财务部支出，所以财务部门是和本系统相关的人或物，所以它是涉众，但却不是执行者。

对于执行者的确定过程，本文档不再作相关的描述，下面只给出此描述里的执行者的相关 UML 图形。

RUP 的所有分析都是从确定执行者开始的。以后的所有分析也都是基于执行者，整个需求的获取与分析过程都是围绕着 某某执行者 做了 某某事 进行的。当然这也是面向对象的分析方式，非独 RUP。

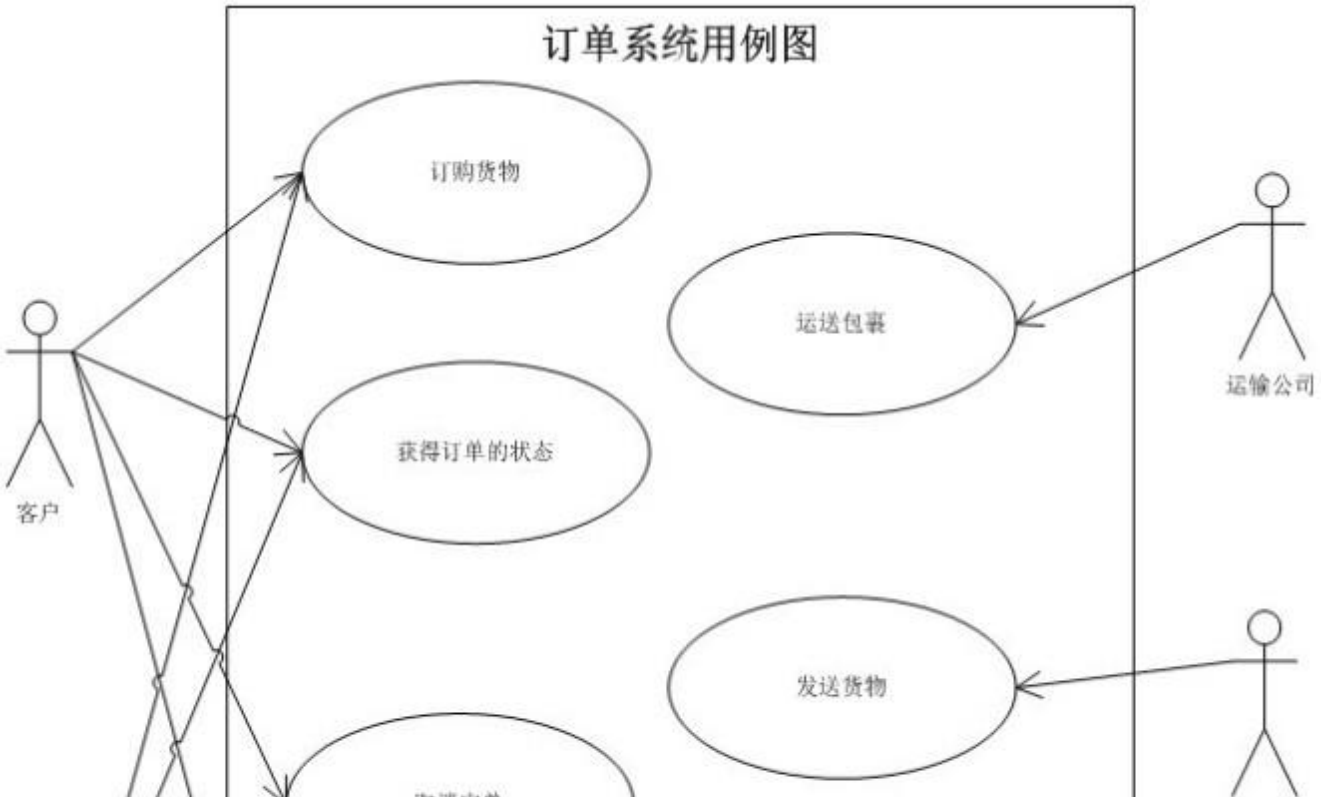
确定了执行者，那么下面我们就开始为每个执行者确定其所有的用例，这里已经是第三次提到用例了。当然在这里我还是不会给出他的定义，你目前要做的不是追问这个问题而是继续的往下看。然后在下面的分析中总结一下自己对用例的

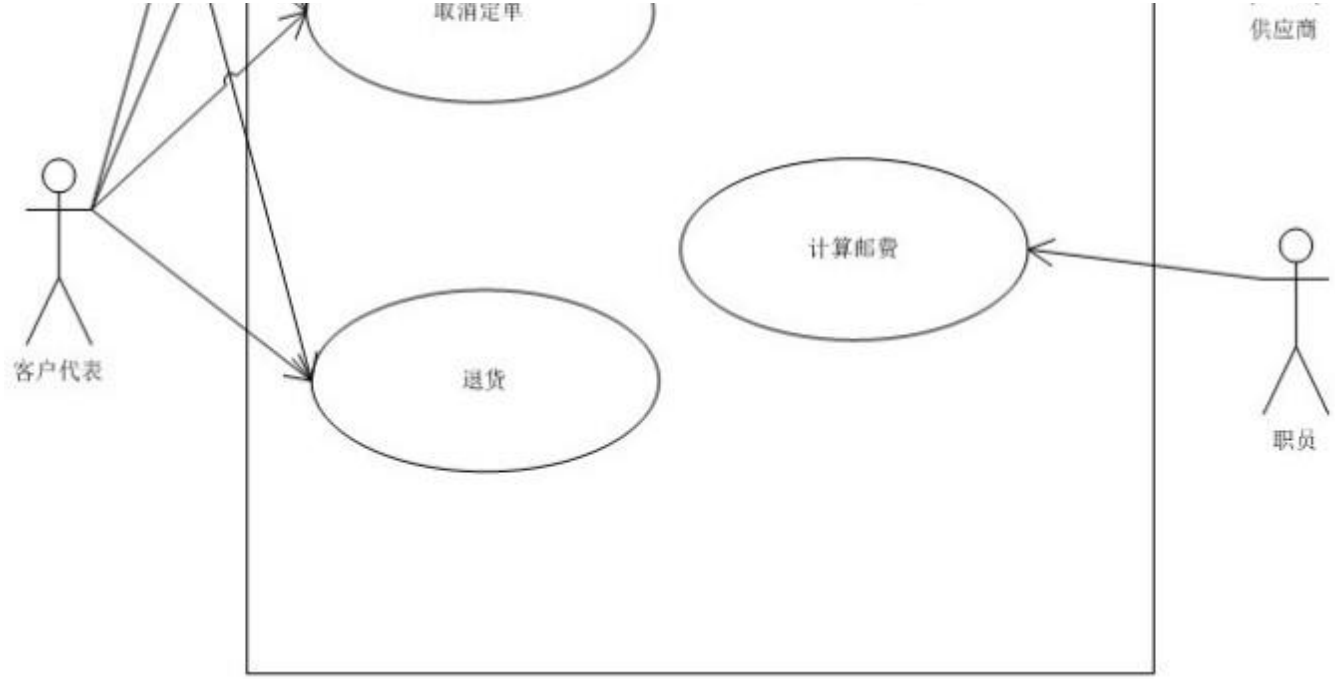
理解，并自己给出一个定义或者不断的修改你的定义。当然在最后。我会在适当的地方给出用例的定义。这时你可以拿它和你的定义进行比较。也许你会发现。其实你的定义比我给出的更加的精确，更加的好。当然，你也可以完全不用去管用例的定义究竟是个什么东西。的确对于用例来说，他的定义并不是必须的。任何的定义恐怕也都是不完备的。

好的， 下面我们让我们来开始提取用例吧，在提取之前请确保你能够再次认真的检查一下执行者，看看是否还有什么遗漏，确定没有后再动手，当然遗漏总是难免的， 这并不要紧， 我们还可以在后面的用例分析中再将它加进来。你现在只须做到一点即可：即是在用例分析前，你对执行者的确定已经做到了最好，这一点是很重要的。

对于执行者来说，一个用例应该是一个完整的任务。一个用例应该是在一个相对连续的时间内完成。如果有明显的时间断层，应该考虑一下把你的用例进行分解。成为多个用例。当然这并不是一定的，这涉及到了用例提取的粒度问题，这个问题相当的复杂，在此不作讨论，否则这个文档真要成为一本书了。不过还是建议你去查查相关的资料，因为这个也很重要。

下面我们给出上面 AA 公司经理的一段描述的用例图，这张是经过一次讨论后，得出的用例图，从图中可以看出来有些东西是在上面的那段描述中不曾出现的，这是讨论的结果，其中讨论的过程，在此不作叙述，怎么讨论也不准备花时间去写，这些方面的具体内容你可以自己去查看相关资料。



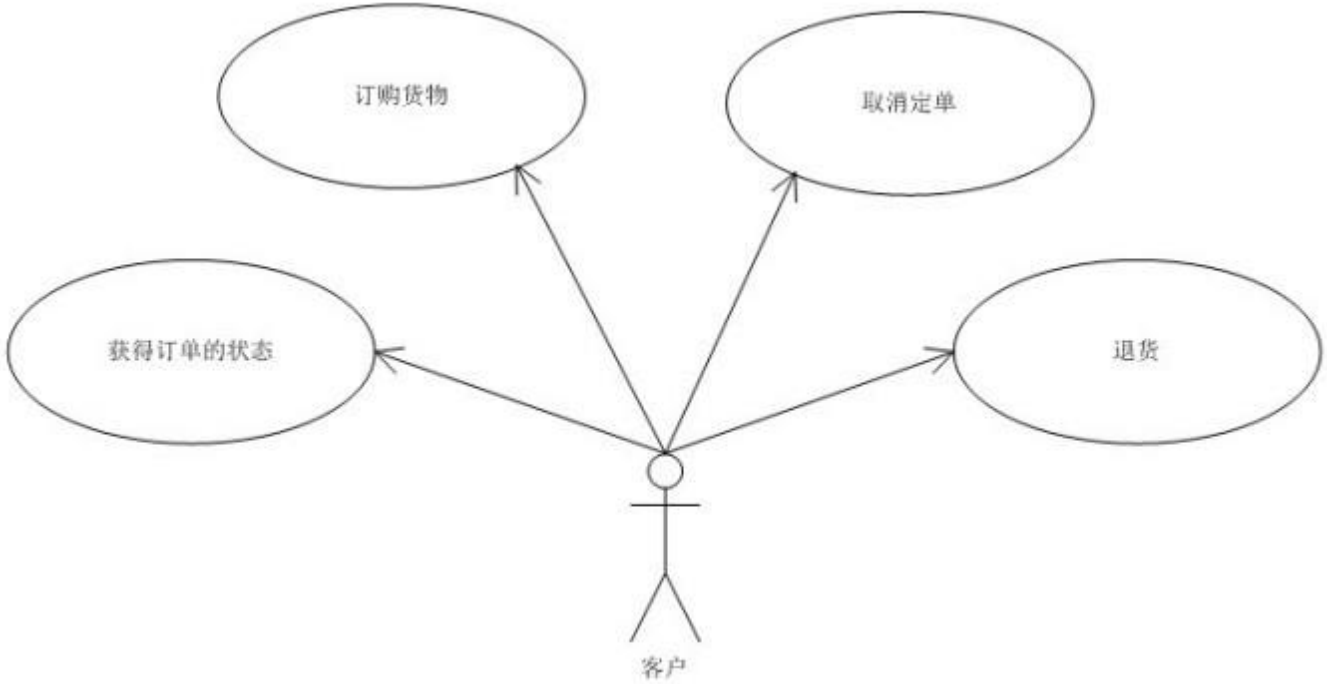


当然这张图仅仅只是经过一次讨论出来的结果，这个用例图也并不是完善的，其中很多东西并没有考虑到。例如财务的处理，银行的转帐等等都没有考虑进去。虽然如此，随时的画出用例图却是非常有必要的，因为任何一个未完成的用例图都是我们目前分析的成果，我们以后的分析也都以这张图作为基础与参考，这样会使我们更容易发现我们还有哪些方面没有考虑到，一旦发现了没有考虑到的事情，就马上更新这张图，将你的发现或者想法加进去，当然也要在合适的时间与地点将这张图拿过去与客户交流，并告诉他们这是我们目前分析出来的系统需求，有了这张图客户可以很容易的明白目前我们已经做了哪些，还有哪些我们没有考虑进去，这时他就会告诉我们他们的想法。我们再对他的想法加以整理，再一次的更新这张图。然后重复上述的步骤，直到客户说 OK。

当然还是为了以后说明的方便，在此文档中不描述迭代的过程，所以我们假设这种图已经是最终的用例图了。

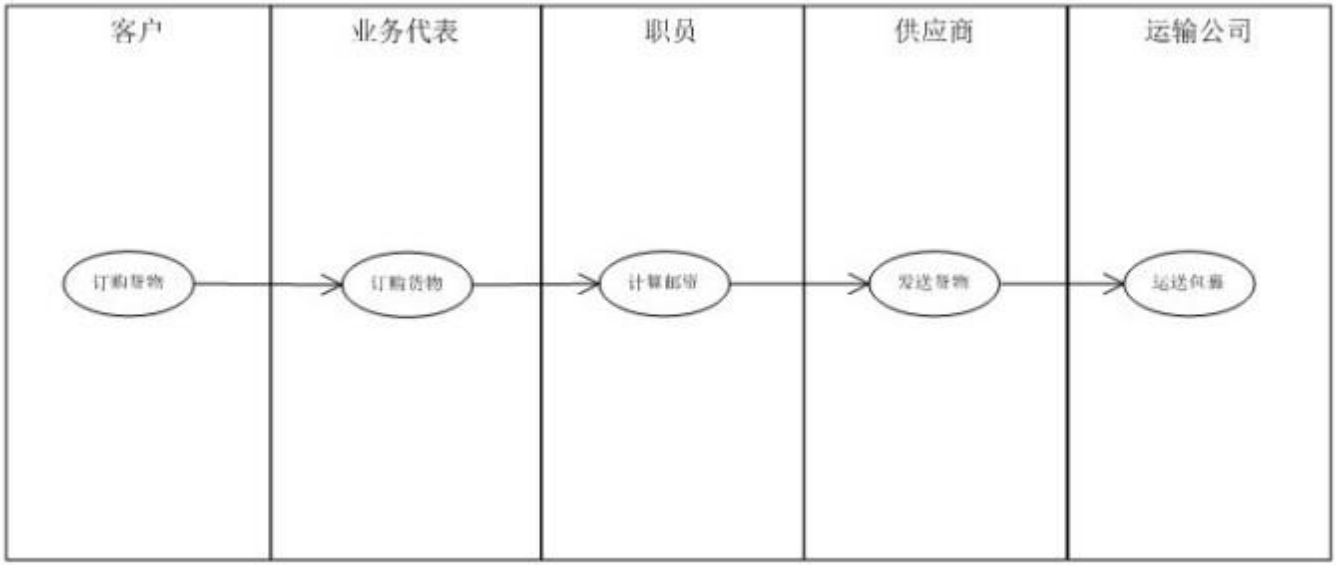
虽然这张用例图已经可以很清晰的描述了系统与所有相关执行者之间所进行的互动操作。但在实际的分析过程中，只使用这一种图往往是不够的，在实际的分析过程中，我们会发现 为每个执行者单独列出他所涉及的一切用例对于发现执行者还有什么操作没有加进去是很有用的，另外 为每个用例单独列出一张相关执行者 对于发现每个用例中还有那些相关人员没有考虑进去也是非常有用的。同时，对于一些典型的业务场景画一张业务场景图对于客户理解用例以及在发现业务的大体流程上还缺少什么环节也是很有必要的。所以下面我们分别帖出这三种图。

为每个执行者列出他所涉及的所有用例（以客户为例，实际操作中你需要为所有执行者列出一张这样的图）



为每个用例列出所有相关的执行者（以订购货物为例，实际操作中你需要为每个用例列出一张这样的图）

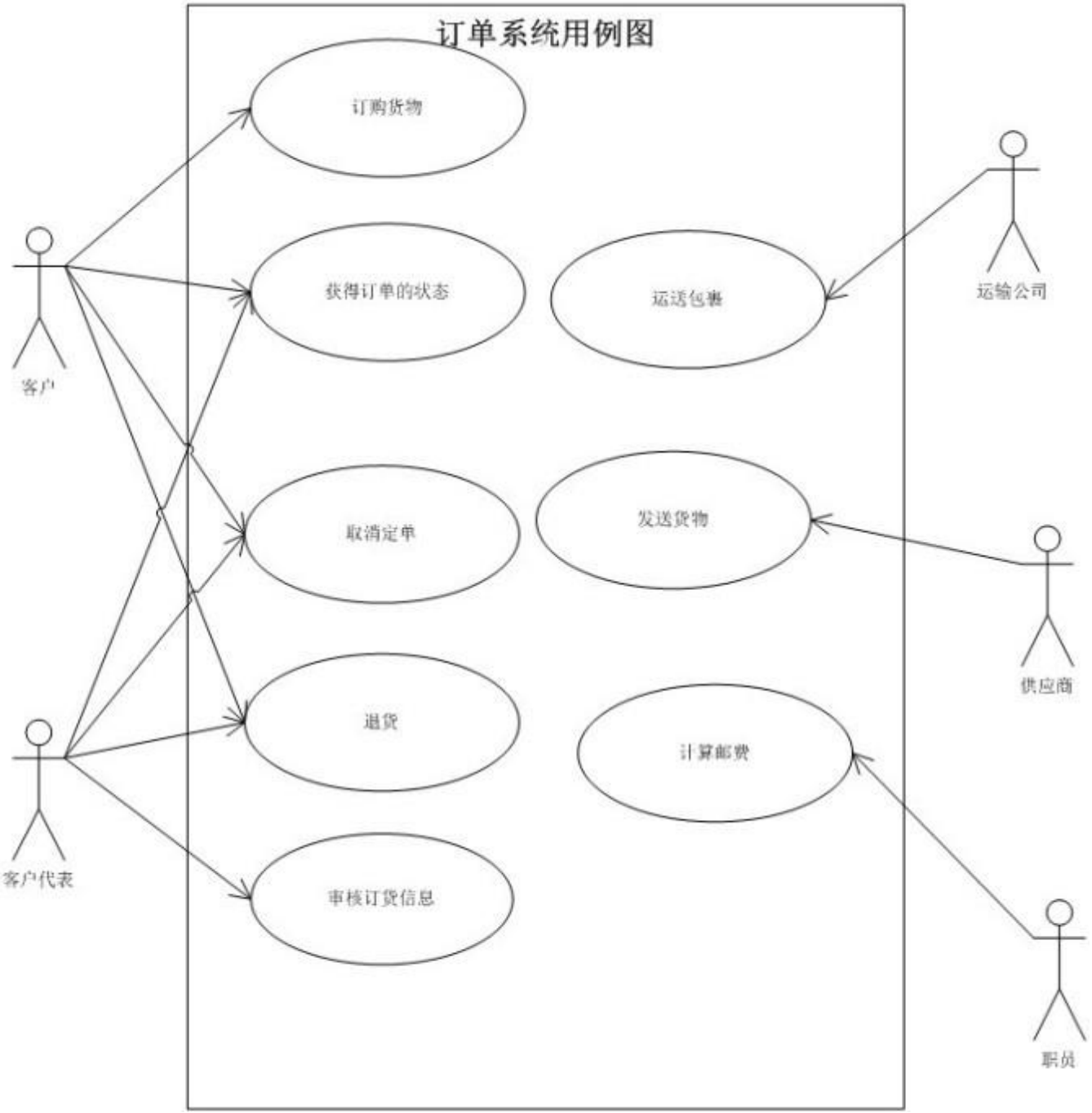
业务场景图（一次交易的场景）



我们在这里第一次接触到了场景这个概念，在下面我们还会遇到另外一种场景，用例场景。在这里还是不讲场景究竟是个什么定义，通过业务场景和用例场景，大家可以自己去比较一下之间的异同，然后自己给出一个定义。在这个场景图中，椭圆代表的是用例，方格的正式名称叫泳道，在业务场景中，每个泳道代表一个执行者，在用例场景中，会有所不同。大家注意一下之间的区别，同时大家在看这个场景图时，是不是发现了有两个订购货物用例，那是因为客户和业务代表都参与了这个用例。虽然在实际上订购货物的过程中，并不一定需要客户和客户代表必须在一个连续的时间内完成一系列动作（用例的一种粒度划分方法），对于传统的电话订购方式一个订购的过程可能会涉及客户和客户代表，但在网购当中这种用例设定是不准确的。事实上应当将它一分为二。一个是客户的订购货物用例，一个是客户代表的审核订购信息用例，但是本文档中为了展现用例分析中的更多细节，到目前为止，订购货物用例一直被视为电话订购的情况。但在后面的用例场景分析时，我们将重新将订购货物假定为只有客户一人参与的用例（网购时的情况）。另外从这张图中，我们可以明显的看出来许多的不协调，例如，这本身是一宗交易，却从头到尾，没有看到钱的影子，因此看了这张图后，我们就会考虑是否应该将电子支付系统这个执行者加进去。在实际的分析过程中，这个回答是肯定的，但在这里，我们先不忙着管它。至于原因，后面会给你回答

在进行下面的描述之前，我们将我们的用例图修正一下，使之更适应于网购

修正后如下:


订单系统用例图
订购货物
获得订单的状态
取消定单
退货
审核订货信息
运送包裹
发送货物
计算邮费
客户
客户代表
运输公司
供应商
职员

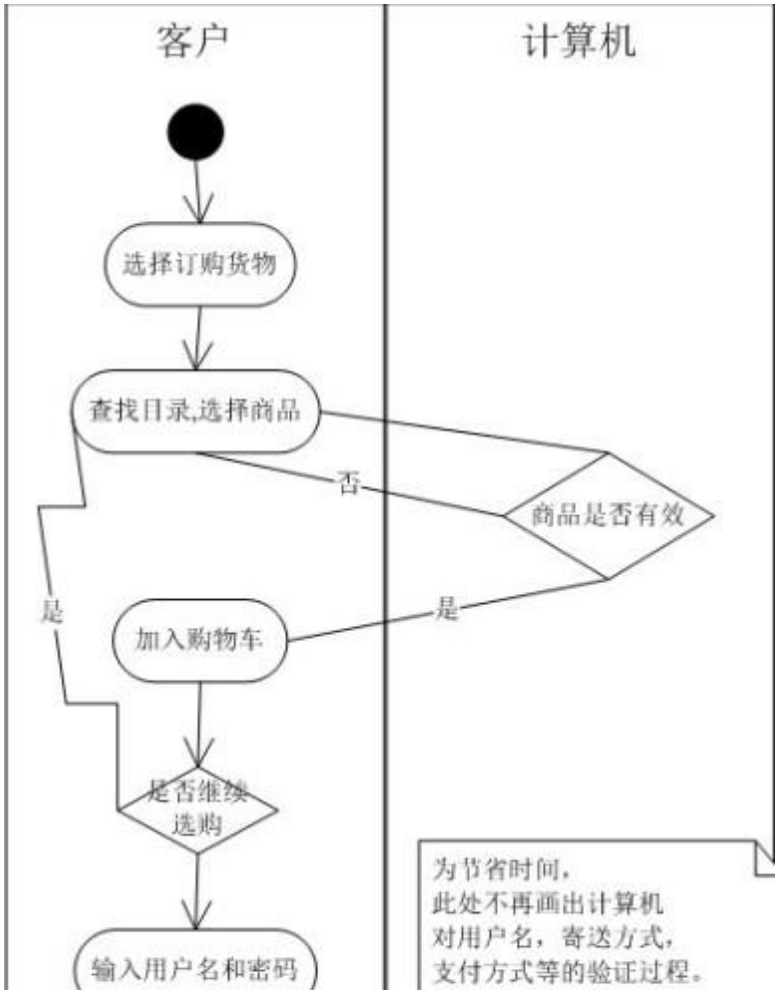
注意：本图只是对订购货物一处进行了修改，实际上取消定单等地方也应作相应修改，为了节省时间，就先这样了。当然，如果你修正了用例图，前面所涉及的一切用例相关的图形都应作相应的修改。

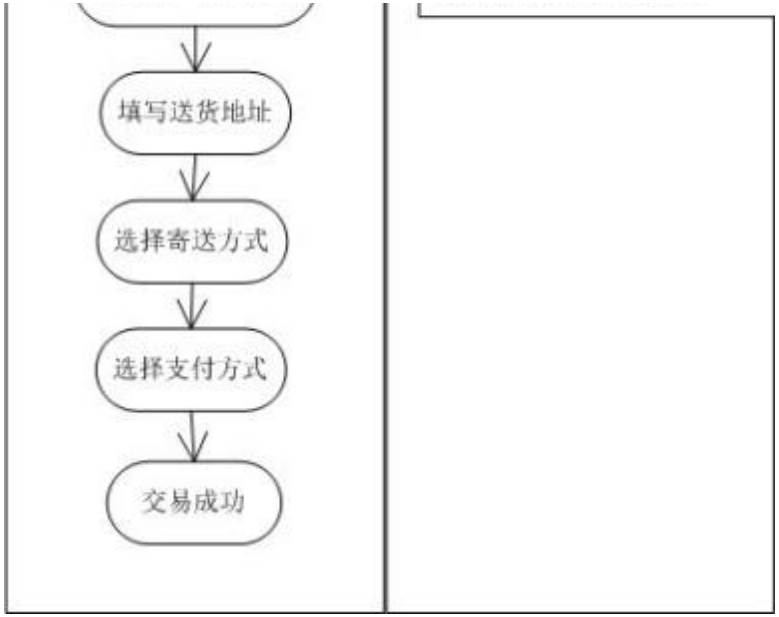
用例是非常复杂的，仅仅用 UML 里的一个椭圆来代替，似乎太过简单了点。所以我们现在有必要引入一张表格－－用例描述表对用例进行更加详尽的文字描述。用例描述表如下（以订购货物用例为例，实际中你应为每个用例都填写这样一张表格）

用例名称	订购货物用例
用例描述	客户通过此用例完成对商品的订购功能
执行者	客户
前置条件	客户已经打开主页
后置条件	返回订单编号给客户
主过程描述	1. 当客户在主页选择订购货物，用例开始 2. 客户查找目录，选择需要订购的产品 3. 对于每个商品 a）系统显示商口的图片价格等相关信息 b）客户将其加入购物车 c）系统自动计划购物车内的商品的价格总和 循环结束 4. 客户选择购买 5. 系统要求用户输入用户名和密码 6. 客户键入用户名和密码，并提交 7. 系统提示用户填写送货地址等相关信息 8. 客户添写送货地址等相关信息，并提交 9. 系统提示用户填写寄送方式 10. 用户选择寄送方式，并提交 11. 系统提示用户选择支付方式、 12. 客户选择使用银行卡支付，并填写相关信息，提交 13. 系统显示交易订单，并将其作为未完成的订单保存，并向电子支付系统要求支付。 14. 支付确认后，订单被标记为确认。并提示用户交易成功， 用例结束
分支过程描述	对于每个商品，在 a) 后选择放弃，回到 2 如果用户之间已经登录，在 4 处直接跳至 7 假如客户选择使用余额付款，在 10 处跳至 13 处
业务规则	如采用余额，用户余额不足，计算机显示余额和所需金额
补充说明	购物车中至少有一个货物，才能提交。

可以看到在这个用例表中，我们已经详细的描述了用例的细节与步骤。其中的主过程描述又叫主成功用例场景，每个分支过程则独自构成一个分支用例场景。同时我们注意一下。主过程描述中的用粗字体表示的那个 “电子支付系统”。是不是有点熟悉，在讨论业务场景时，曾经提到了他，并且当时并没有把他加入执行者中去，为的就是在此处说明一个问题，就是在写用例描述表时，我们也能够发现一些曾经忽略的执行者或者用例。此时一旦发现，我们同样必须及时的去更新用例图。以使你的用例图更加的完善更加的接近于客户的要求。

虽然用一堆生涩的词汇和句子是可以完全的表述一个用例，但是不免让人觉得有点难以接受。没有什么太直观的印象，因此我们有必要再引入一些图形。以使之形象化。用例场景图 正好能够满足这个要求. 每个用例场景图，对应着一个用例场景（一个用例有多个场景，一个主场景＋多个分支场景），在此我们只画出主用例场景的 用例场景图，分支场景将不再绘出，各位可自行去绘。





经过了反复的迭代，分析以及与客户的交流，目前我们已经得到所有的用例以及他的场景，并一一用 UML 使之图形化。同时这些用例及场景也已得到了客户的认可，并被认为已经能够很好的描述出他们所希望的系统的样子。那么下面我们可以放心的进入建模的下一个环节——分析业务实体了，当然在进入下一环节之前，我还得履行此节开篇的诺言，给出用例的定义。

用例：还是不做介绍了，可能会误导。

补充说明：在发现用例，尤其是粒度比较大的用例时，画出事件表，对于分析有很大的帮助。详细情况，请自己查看相关的文档。

设计业务实体

从上一节里，我们继承了诸多的成果，例如用例，场景以及花了 N 长时间绘制的那一堆用例图和场景图。其中我们需要的是用例描述表和用例场景图。这两个可以帮助我们很好的去发现业务实体。

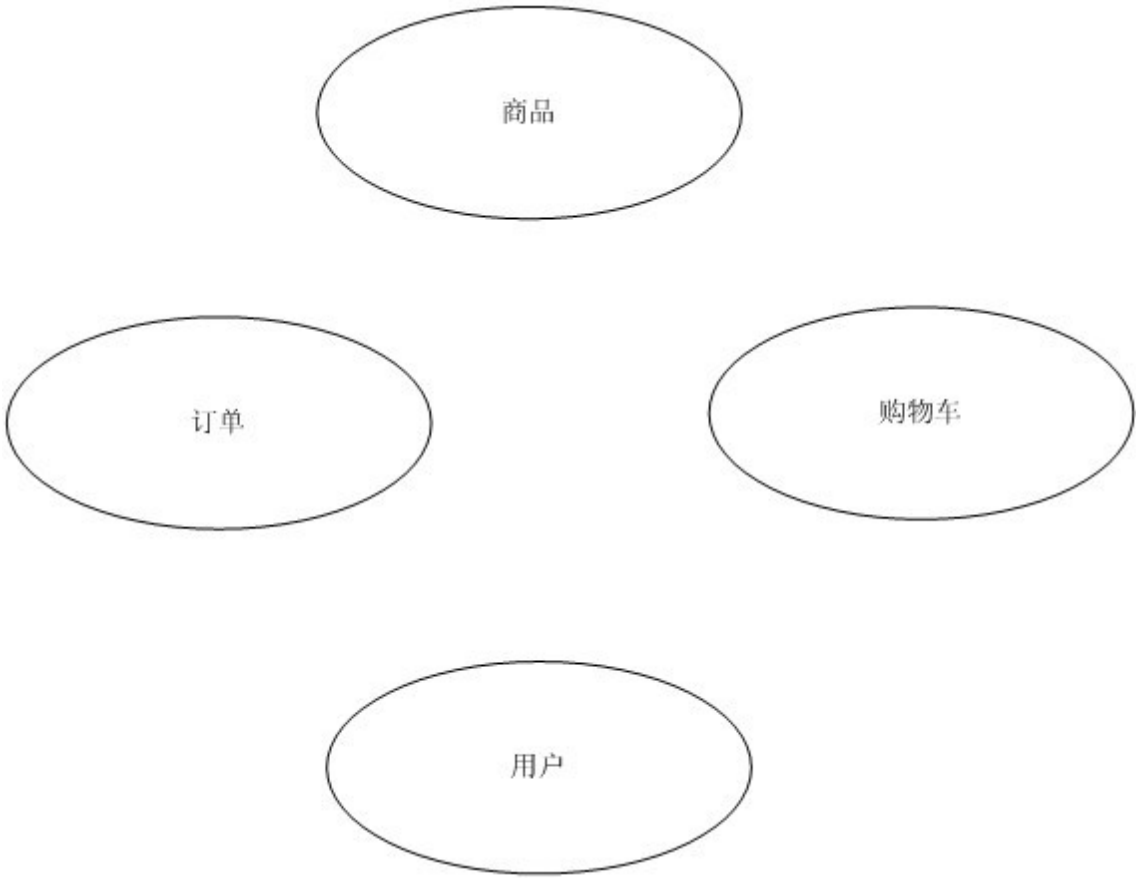
业务实体的命名其实很准确，但是不太容易被程序员所理解。假如我讲对象的话可能会让你感觉更亲切一些。但是我们要明白一点。业务实体和对象虽然有着密切的关系，但却也有着本质的区别。对象确切的说是面向对象编程的一个术语，是和计算机密切相关的，是一个设计阶段的概念。而业务实体则仍是一个需求阶段的概念，需求文档是要给客户看的。所以业务实体不会去考虑抽象，设计模式等。假如你在需求文档里弄出个什么 FACTORY，或者 FA?ADE 模式。就好比财务人员给你弄出些什么复式记帐法，借贷平衡，借方科目，贷方科目。亦或科技人员给你弄出些什么量子理论，测不准原理。绝一点的哲学家再给你弄出些个什么奥卡姆剃刀，证伪主义一样。肯定会搞的客户一头雾水，不知所云。

业务实体是需求阶段的概念，对象是设计阶段的概念

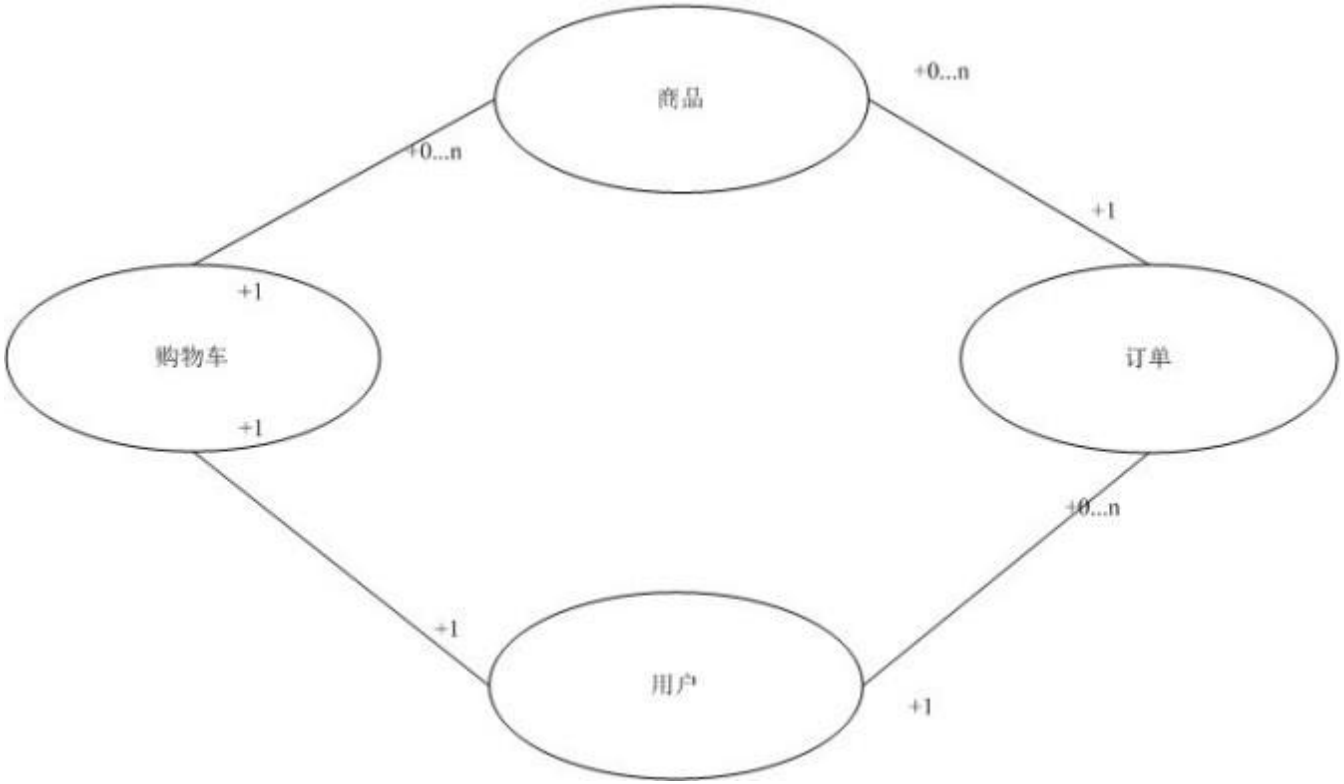
要对业务实体进行分析，我们先要从用例描述表和用例场景图中分析出来究竟有哪些业务实体。这是一项技巧性很强的工作。有些简单的一眼就能看出来（名词），有些则隐藏的很深，这需要有相当的经验和对实体的敏锐感知能力以及认真仔细的态度。这不是本文档要讨论的。本文档只讲到这个步骤你要提取业务实体了就行了。

业务实体一般包括 实实在在的事物、角色、组织部门、设备、交互行为、地点位置等

下面我给出一张业务实体图（只根据订购货物用例简单的分析了一下，实体不是很全，实际当中，你要把所有的实体都分析出来。本文档只简单演示一下而已）



分析出来了有哪些业务实体。下一步就得分析业务实体的关系。图如下



关系主要有一元（回归）关系，二元关系，三元关系，N 元关系，其中二元关系比较重要。其具体又分一对多，一对一，依赖，多对多等，具体请查阅相关资料。

理出来关系后。下一次我们开始分析业务实体的属性。所谓属性就是有关业务实体的一条特定信息。

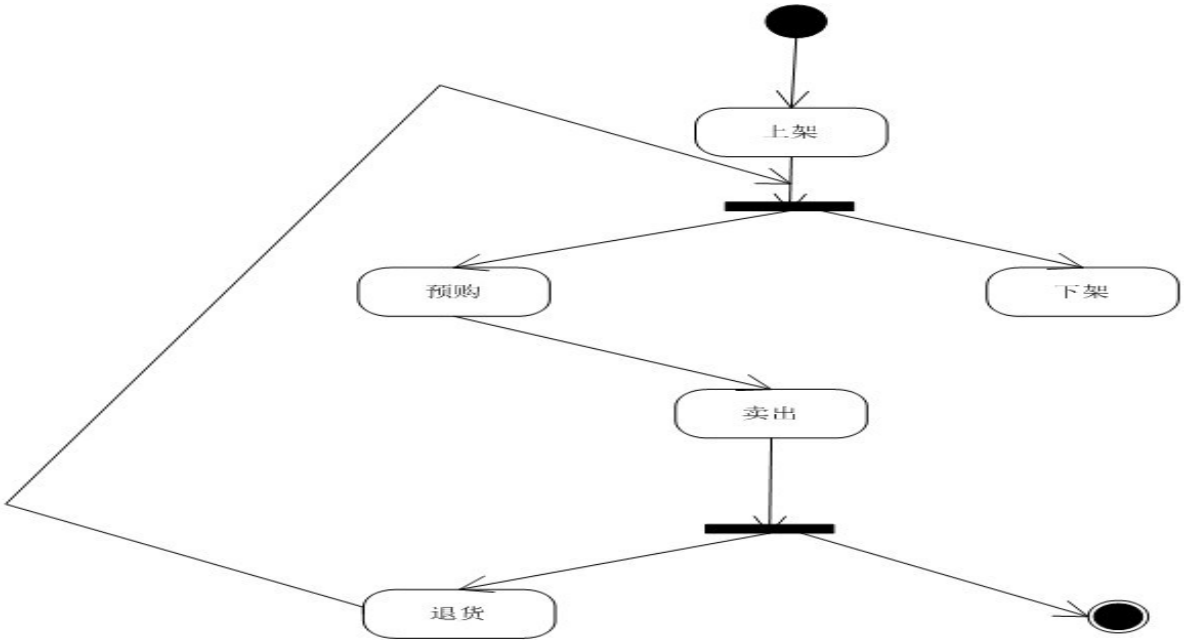
属性：有关业务实体的一条特定信息

对于业务实体的属性在需求分析阶段一般不采用 UML 图形的方式，而是采用领域模型（业务实体）说明表的形式表示。领域模型说明表如下（以商品为例，实际中应为每个实体写一份这样的说明）

实体名称	商品
实体描述	每个商品都经有上架（录入商品信息），预购（放入购物车），卖出，退货和下架几个状态，详细请参看商品状态图

属性名称	类型	精度	说明(属性的业务含义及业务规则)
商品编号	字符	12	商品类别编号(3位)+商品购入年份(4位)+流水号(5)位
商品分类	数字	3	商品的分类
名称	字符	100	商品的名称
价格	数字	12	商品的价格
折扣	数字	6	折扣价
产地	字符	100	商品产地
简介	字符	1000	商品的内容简介，上架时录入
状态	字符	1	商品的状态，可参看商品状态图

下面给出说明书表提到的商品状态图。



当然并不是每个实体都必须要画出一个状态图，对于那些只有两三个状态或者没有状态的实体。完全可以不画状态图。但对于有较多状态的实体，建议还是不要偷懒的好。

好了，万里长征终于走了一半了。到目前为止，整个需求分析阶段算是基本完成了。回顾一下前面的叙述，从提取执行者，到获得用例，再而根据用例写出业务场景，再对用例进行详细的分析从而勾勒出用例场景，最后再到发现业务实体，并理清他们之间的关系，同时对他们的属性进行挖掘。我们经过了一个从朦胧到初步认识的过程，在这段过程中，我们积累了很多的成果，也学会了画许多样的 UML 图形，但是一切还是显得那样杂乱，还是显得那样没有条理性，现在我急需的是一个文档，对上面所述的一切进行归类与整理，以使之更容易被人所明白与理解。这个文档就是需求说明书。本文档提供一个需求说明书的例子（此文档仅作例子，具体需求说明书版式，需采用公司统一标准），见附件 1。需求说明书也将作为输出文档。输出到下一个阶段 -- 设计阶段。当然实际的 RUP 要输出的东西更多，具体可参见 RUP 相关资料。

3.2 系统设计

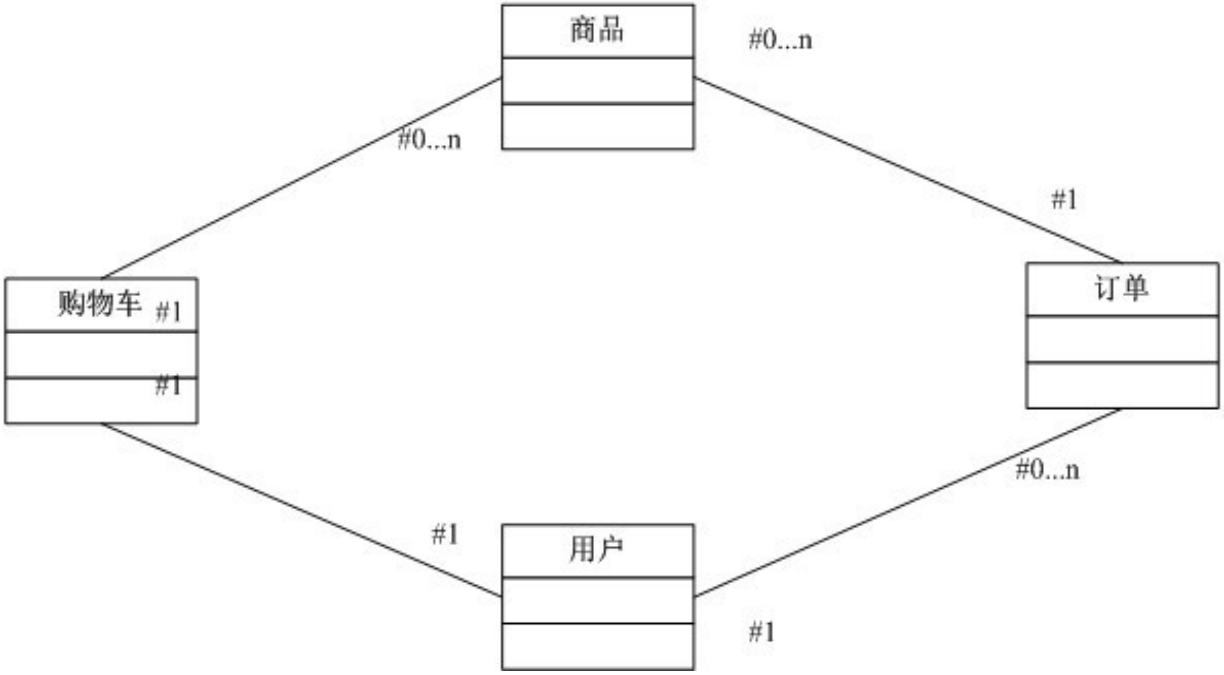
划分子系统

这一过程理论上是属于需求分析阶段，当用例越积越多时，就已经有必要对用例进行整理与分类，将功能相尽或者相关度较高的用例归入一个比较大一点的类里，从而形成组件，再继续归类，从而形成子系统。当然这一步，也可以放到设计阶段来完成。所以将这一部分的内容写到了这里。如果需求阶段已经划分了子系统，在设计阶段直接继承，如果没有，则需在设计阶段的第一环节划分出子系统。对此我不准备再作过多的叙述，各位可参考相关的资料。接口也同样。须各位自行去查考相关的资料，这毕竟只是一个文档，不是一本书。

要说明的一点是，在需求阶段划分出来的子系统，在设计阶段需要再审查一下，进行进一步的优化。优化的过程贯穿于整个设计阶段。接口也同样。这就是跨阶段的迭代，当然要尽量避免这种跨阶段的迭代。如果你有过多的跨阶段迭代，就有必要重新审查一下你的需求分析做的是不是有什么问题了 。

设计” 类”

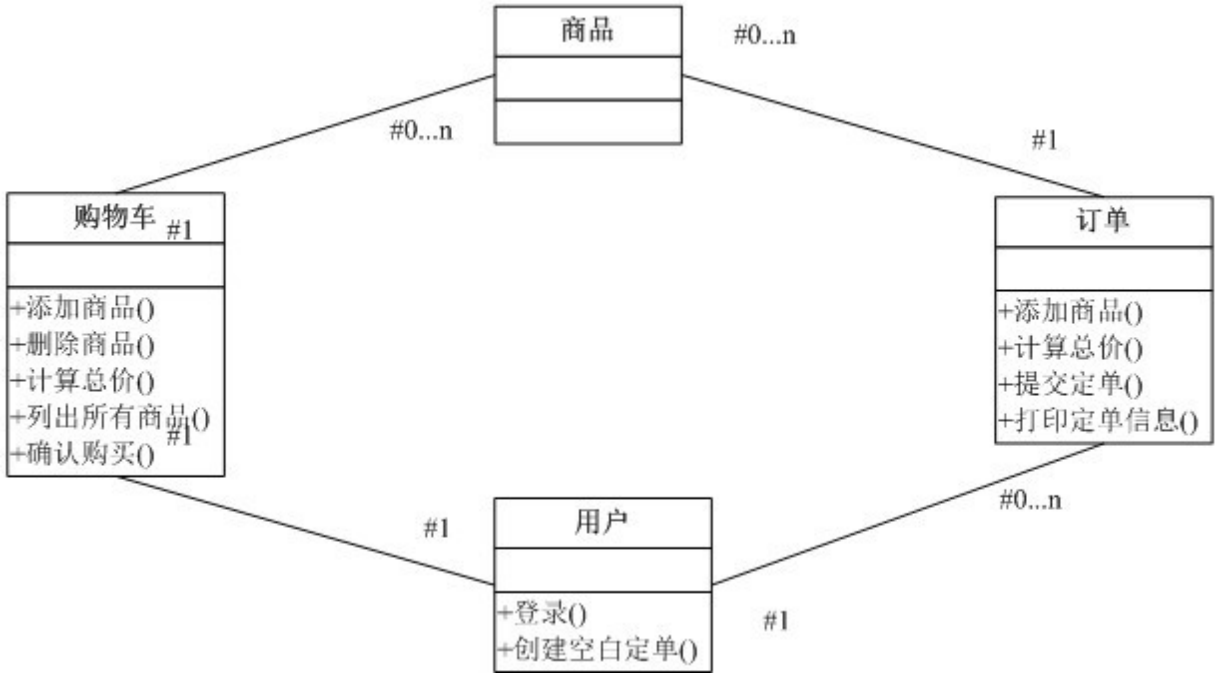
类的设计是从分析业务实体（领域模型）开始的。最初的类的设计就是简单的将需求分析阶段分析出来的业务实体用类的形式表现出来。当然业务实体之间的关系，以及业务实体的属性，也都应一并继承下来。做完了这些后我们要做的就是分析类的形为，类的形为也叫类的方法，有些书籍上也叫类的责任或者消息。当然现在我们先不着急做这一步，先将上面分析出来的业务实体 转成类的形式 再讲。类图如下



属性信息我就不在这里表述出来了，实在是太烦了，时间也紧。

有了这张图，下面我们开始在他的基础上分析一下类的形为。分析类的形为的具体方法不在本文档里进行讨论，相关的资料和书籍 N 多，可以自己去看。

下面仅列出经过简单分析后重新更新过的类图，在这个类图中加入了形为



当然这个类图并不是完善的，也不一定是合理的，在此仅仅只是用来举例而已，各位不用太过深究了。在进行类的行为分析时一定要综合所有的和类有关的用例进行分析。同时记住一个很重要的观点，就是类只对自己负责。用户类绝不会去添加商品，因为添加商品不是用户的责任，而应该是购物车或者订单的责任。购物车类也绝不会去登录。因为登录是用户的责任。而不是购物车的责任。

不知道你是否还记得在进行用例分析时，用例分析完成后，引入了业务场景和用例场景的概念。使之可以对用例的细节进行分析，同时在分析业务场景和用例场景的同时，又进一步的完善了用例。在类中也一样。在把类分析出来后，便开始分析他的形为。然后根据形为开始分析他们之间的交互，在分析交互的过程中进一步的去完善你的类。迭代，又是迭代。是的。在 RUP 中，迭代是无处不在的。

记住，从需求分析继承过来的类并不是最终的类，需求分析阶段主要面向的还是客户，直接继承过来的类还仅仅停留在业务的层面上，但在进行设计时我们却不得不考虑到业务在计算机里的实现问题，因此会对这些类进行调整，优化，扩展，以使之更适应于计算机的实现。这些调整包括运用面向对象的一些特性，例如继承，重载等对类都行重构。在重构的同时。可能会出现许多与具体业务无关的类，虽说他们和业务没有什么关系。但在使一个业务在计算机中得以实现时却是必不可少的。所以这些类你也必须得如实的反应于你更新后的类图上。当然现在我们还是先不着急去管这些，目前紧要的还是进行类的交互分析。

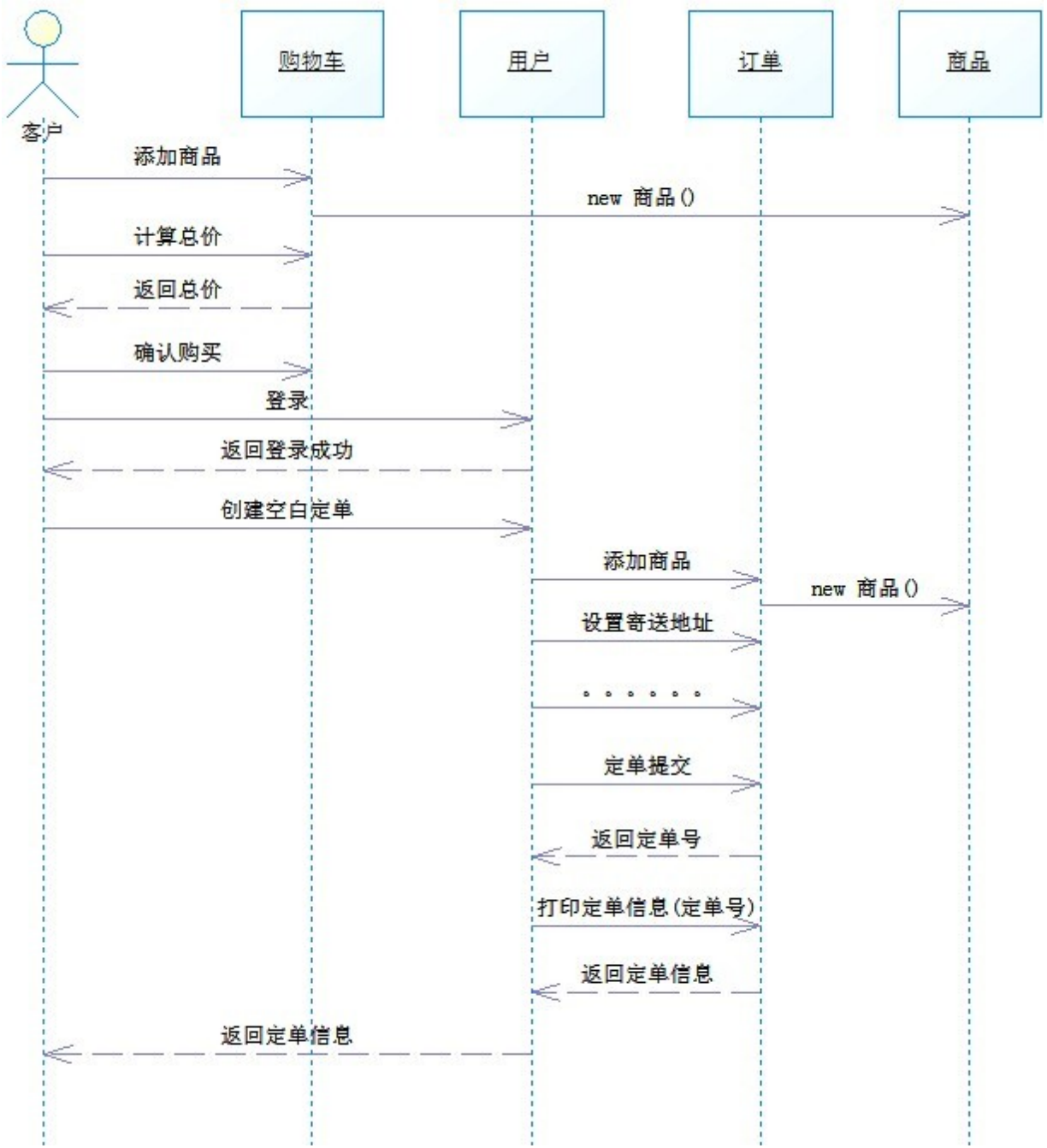
在类的交互分析中最常用到的是顺序图和状态图。至于交互图爱画不画

一般情况下，一个用例的一个场景必须对应一个顺序图，顺序图是为了表现用例中所涉及的类之间的消息传递的过程，这个表述可能让你有点不明白，换种说法，顺序图就是表现用例中所涉及的类之间的调用顺序。其实第一种表述中的消息指的就是类的方法。前面也已经指出过类的方法的表述有很多种，消息，责任，行为。但是无论怎么表述，内容都还是一样的，只是不同的表述用于不同的语境而已。这个知道即可。

状态图和顺序图有所不同，顺序图的主体是用例，而状态图的主体是类。状态图表现了一个类在其参与的所有用例中的状态的改变及其触发条件。状态图异常复杂，要想真正的理解恐怕还得你去找相关的资料多多的研究一下。这里不再作说明。、

下面我们延续上面的例子，绘制一张订购货物的主场景的顺序图，状态图太复杂了，而且前面也没有列举其它的用例，所以在这里就不画了。

订购货物主成功场景顺序图


客户
购物车
用户
订单
商品
添加商品
new 商品()
计算总价
返回总价
确认购买
登录
返回登录成功
创建空白定单
添加商品
new 商品()
设置寄送地址
. . . . . .
定单提交
返回定单号
打印定单信息(定单号)
返回定单信息
返回定单信息

在这张图中，并没有画出查找目录的相关交互过程。那是因为本文档在进行业务实体分析时，考虑到只是举个例子而已，故只简单的画了几个实体，思考的并不是很全面。不小心将商品目录这个业务实体给丢掉了。但又为了免于唐突，在这里也就没有把和他相关的交互过程表现出来，要不然突然冒出来一个没见过的类对象。反而更容易让人糊涂。虽然这个业务实体很重要，但在这里我就不因为他而去更新之前的相关图例了。这实在是太费事了。大家知道怎么回事就成了。当然在实际的操作过程中，这种思想可要不得。一旦你发现了这种问题，一定要一直的向上追朔，直到将所有相关的内容和图例全部更新为止。

当然实际情况下，这张序列图对于开发人员来说无疑是一个噩梦，这个序列图所表现的那个用例涉及了过多的间断性动作，导致了这个序列图中很多动作都是不连续的，如果用 WEB 开发的术语来讲，就是这个序列图中包含了过多的请求：在一个页面只有一个 HTTP 连接的情况下页面刷新一次叫一个请求，这样在实现时就需要技术人员去把这个序列图分解，然而这个分解过程是很痛苦的，当然这不是开发人员的错，造成这种情况的原因是从一开始（进行用例分析）的时候我们就没有遵照前面提到的一条原则——一个用例最好是表示一个不间断的动作。当然这个问题并不是不可解决的，因为只要将原先的那个庞大的用例进行分解就行了，当然这也并不是说，这条用例就没有存在的必要了，事实上恰恰相反，这种粗粒度的用例却是必不可少的，因为粒度过细，就会导致用例膨胀，如果没有这种粗粒度用例进行归类的话，会导致非常大的混乱，不易让人理解。这就产生矛盾了，一方面你说一个用例最好表示一个不间断的动作，一方面又说大粒度的用例必不可少，^_! 你这不是自己打自己嘴巴吗？事实也确实该打嘴巴，当然打归打问题还得解决，解决的方法其实很简单，只须再引入一些概念即可，即是用例的包含与继承关系，大用例包含多个小用例，当然包含的原始目的是为了用例的重用，但根据目前我的实际经验，把一个大用例完全分解成各个小用例，然后大用例包含所有的分解后的小用例，这些小用例可以继续分解，直到用例是表示一个连续的动作（在 WEB 中则最有可能的是一次请求，或叫一次会话）为止，也就是说叶子节点上的用例都表示一个连续的动作。绘制序列图时，也只需绘制叶子节点上的用例即可。当然以上是个人经验之总结，对不对我也不太清楚，各位仔细掂量，附 用例与用例之间的包含与扩展的原始意义 作为参考

用例与用例之间的包含与扩展的原始意义：包含关系表示一种从属关系，即子用例是主用例中相对独立的、必须调用的一部分功能。在用例分析中，我们应当将多个用例都共有的、相对独立的功能提取出来形成一个子用例，为日后代码复用提供有力保障。扩展关系表示一个功能是对另一个功能的扩展，即被扩展功能不一定调用扩展功能，但扩展功能是对被扩展功能的加强与延伸。在绘制用例关系时，包含关系应绘制成从主用例指向子用例的虚线箭头，并标注为“include”，表示主用例包含子用例；扩展关系应绘制成从扩展用例指向被扩展用例的虚线箭头，并标注为“extend”，表示扩展用例是对被扩展用例的扩展。虚线箭头在 UML 中代表的是一种依赖关系，即客户元素了解供应者，并且供应者的变化会影响到客户元素（依赖是从客户元素指向供应者的）。

通过对顺序图和状态图的分析，可以让我们更有效的去完善类的属性以及它的方法。从而，使我们的设计更加的接近于客户的要求。但这还远远不够，因为目前我们仅仅是停留在完善的层次上，还不能说是一个优秀的设计。甚至连成功的设计都还算不上。因为到目前为止我们还没有考虑到复用的问题。

复用大致分为三个层次。从低到高依次为，代码级复用，组件级复用，服务级复用。

1. 代码级的复用：主要依靠面向对象的特性，如继承，重载，抽象等。此级别复用的执行者是代码开发人员。（注：建议用组合来代替继承）

2. 组件级的复用：主要依赖于设计模式，例如 FACTORY，FA?ADE，ABSTRACT，FACTORY 等。这一层面的复用的执行者一般为设计人员。

3. 服务级的复用：这个复用的层次最高，理解也最为困难，如有兴趣可自行查看关于 SOA 以及云计算的相关信息。个人认为这个粒度太粗，把握起来很困难，实际运行时，见机行事。

对于复用的其它相关知识，本文档就不再做什么深入的讨论了，各位也可自行查阅相关的资料，只要记住考虑复用是在已经发现了尽可能完善的类后进行的就可以了。当然这也不一定，如果你的经验够丰富，可能不自觉的在设计前期就已经将复用考虑进去了，所以我们还是具体情况具体对待吧，不用搞的那么死。

数据库相关

按着上面所述的步骤，经过一次又一次的迭代，一次又一次的反复。终于兴奋的发现，原来我们也可以做出优秀的设计。于是乎我们迫不及待的将这些设计的成果发给了开发人员，并自信的对他们说如果照着这个做，保证能做出来一个优秀的系统。然而事实呢，开发人员拿到我们这些所谓的优秀成果。恐怕除了骂爹骂娘骂你祖宗十八代，其它的什么事也做不了。的确，经过上面的迭代，我们的设计看起来已经非常的优秀了，但事实的情况是：根本无法实现。因为到目前为止我们至少还少考虑了一个极其重要的部分。那就是持久化支持，当然这个名词听起来有点玄乎。说白了，就是怎么保存数据。再白一点就是讲你的这个设计怎么和数据库扯上关系（当然你也可以使用其它存储方法，但这里我们只考虑普遍情况）。

当然这里我也只是讲出有这么一个过程。具体怎么映射（扯关系就叫映射），是一个类映射成一张表呢，还是一个类映射成多张表呢，亦或多个类映射成一张表呢。又或者怎么处理有继承关系的类到数据表的映射等等，诸如此类，请自己查阅相关文档。当然处理完成后。别忘了画些 ER 图啊什么的，这是必须的。

又经过了一个漫长的发现，分析，改正并不断迭代的过程。终于，概要设计的所有东西我们都分析出来了，也完善了。什么子系统，组件，接口，类，以及他们之间的交互，包括数据库都写的很详细很清楚并且很明白了，同时又不辞劳苦的用 UML 工具，将这些所对应的什么子系统图，组件图，接口图，类图，类顺序图，类状态图，甚至类交互图，包括 ER 图也都画出来了。然而你却惊奇的发现。当你再次把这些东西发给开发人员，然后告诉他们，你去实现吧的时候，他们仍然骂了你的祖宗十八代。的确，这次你是没有把数据库设计给拉下，但如此混乱的一堆东西，恐怕也是没几个人能够接受的吧。所以下面我们有必要象整理分析成果时一样对设计成果也进行一番整理。这并不是件麻烦的事，有现成的格式让你套－－概要设计说明书。按着概要设计书的目录，将你之前那一堆毫无章法，东一个西一沱的所谓优秀设计的混乱成果们，分门别类放进去，然后你就可以将这个概念设计说明书放心的传到下一个环节，顺利的结束这段折磨人的旅途了。当然如果下一环节是详细设计说明书的话，恐怕还得逮着你。但一般情况下，详细设计说明书的情况不多，一般都是直接到开发人员哪里去了。

结语：正如开篇所言，推导过程不是一个一成不变的东西，以上的东西仅供参考而已。

开发软件不是生产产品，当然开发软件也不是搞艺术设计，确切的讲，应该是介于这两者之间的东西。鼓励创意的同时又约束部分的形为、需要流程而又不局限于流程。这就是软件设计：一个世界上最差劲的艺术设计，外加一个世界上最糟糕的生产过程。

相关文章

锻造软件需求人员的六大要素

开发需求路线图：重要的是规划

需求用例模板

如何做好需求变更管理？流程规范

相关文档

需求分析方法

需求分析与建模 (两个周期)

如何将客户需求转化为技术需求

产品生命周期管理

相关课程

面向产品的需求分析与管理

面向产品的需求分析与管理

非功能需求分析与管理

用户体验 & 界面设计

软件需求分析与管理

分享到