目录

软件系统分析与设计

中科院计算所培训中心 谢新华

前言

在软件工程领域中，分析与设计的作用举足轻重。本课程旨在从面向过程（结构化）和面向对象两种方法论的视角，全面探讨软件系统分析与设计的思想、方法和技术。作为软件工程学的基础课程，我们需要对开发过程各个节点之间的相互关系，以及方法论的基本内容作深入研究。但是，对于软件工程学中更深入的内容，将在更专业的“需求工程”、“软件体系结构”、“程序设计”等课程中会详加讨论。

作为分析与设计方法的基础性、贯穿性的课程，我们需要把一些共性的核心思想贯穿起来，例如结构化分析与设计、面向对象的分析与设计、UML 原理与应用等，使知识结构形成一个整体。这就为为学好软件工程学科其它课程提供了能力基础，所以学习好本课程，对于学习软件工程学全面的知识是非常必要的。

作为一个未来的软件设计人员，仅仅把分析和设计作为一个孤立的节点来讨论，或者在一个很窄的思维空间中研究是没有意义的。任何设计都来自于目的，所以课程中把分析与设计放在整个项目过程的大环境下来研究，针对每个关键节点的影响特点进行研讨，使学生真正理解分析与设计最精髓的东西，这就可能使学生在未来的分析与设计工作变得极有主动性和想象力。

在以质量为核心的现代软件工程方法中，课程中将引导学生关注细节，在细节中把握分析与设计的核心思想，例如，需求分析如何给设计方提供充分而有效的信息，在产品设计上，如何尽可能利用已有信息，合理组织技术方案，把人和任务作为一个重要因素进行考虑，使高的投资回报率成为可能。

课程中还关注了团队思维，软件分析与设计绝不是某个神秘人物冥思苦想然后又自鸣得意的产物，分析与设计应该是集体智慧的成果，软件设计与开发也应该是集体共通劳动的结果，重要的是各种相互矛盾的要求的合理平衡，这都需要有非常良好的方法，把团队的智慧集中起来，如何充分激发集体的智慧，也是一个分析师与设计师必须具备的能力。

在讨论分析和设计的问题的时候，还关注软件经济学这样一个重要的主题。任企业的核心价值就是投入产出比，讨论分析与设计的目的就是如何使这个核心价值得以实现，这就是软件经济学问题，软件经济学首先来自于军方项目，以大资源消耗为特点的军方项目更加关注成本和质量，这也是现代软件分析和设计人员必须关注的一个问题点。

本课程的特点是抓住软件工程各个领域之间的关系，从整体上和方法上解决问题，这对于相关课程的理解是非常有帮助的。

2010 年 9 月 于北京

第一章 软件系统分析与设计的基本思想

1.1 软件工程学的思维方式

首先我们必须理解本课程的教学目标：

1）打下经典和现代系统分析与设计方法的理论基础。

2）掌握系统分析与设计的实践知识。

3）建立团队合作共同设计的理念。

作为本门课程的基本理念，我们将不再关注最后的考试，而是把关注点放到过程中来，让学生在过程中逐步提升自己的能力，因此我们需要思考我们应该具备什么样的能力？

作为软件学院的学生，除了需要学习计算机程序设计以外，更要在整体上掌握软件系统分析与设计方法，需要对行业有更宽阔的视野，需要掌握项目开发的组织方法，需要具备团队协作的能力，需要有在应用层面实现创新的强烈欲望和思维方式。我们不但要关注技术，更要关注组织，尤其需要具备从系统的角度想问题的能力。我们应该拒绝平庸，使自己逐渐走向有向优秀。为此，我们首先在宏观上理解我们所处的行业，以及所需要的知识和技能。

一、IT 行业的昨天、今天和明天

当开始学习这个专业的时候，你能告诉我为什么要学习这个专业吗？这个专业未来发展的空间是什么呢？这个专业的特点是什么？我们以什么方式来学习这个专业呢？我们都知道，要取得成功，必须有经验直觉+三观分析（宏观、中观、微观），为了宏观上预见今后十年会发生什么，就要回顾过去十年发生了什么？一般地说，IT 经历过三个时代。

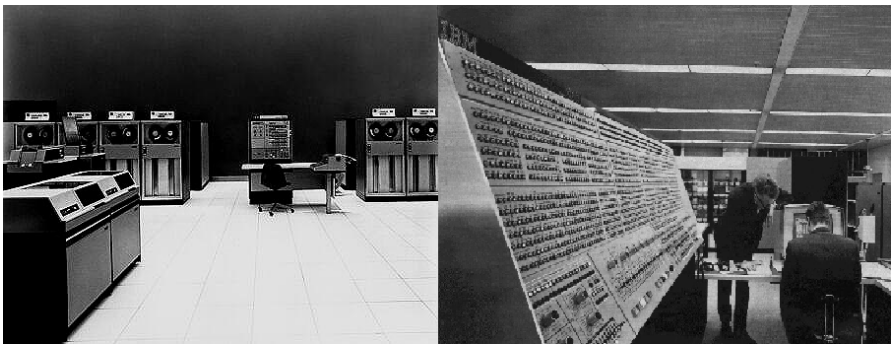
1，第一个时代，大机器时代

世界上第一台计算机，是 1946 年 2 月 14 日，美国宾夕法尼亚大学莫尔电机学院的“埃尼阿克”，但那还只是一种科学研究。


艾尼阿克计算机


莫克利博士

但这件事情引发了一个当时并不起眼的公司商业模式的想象。国际商用机器公司（IBM），原来只是一家做电子磅秤的公司，老沃森是一个天文爱好者，在 1937-1943 年间，他开始转向一种银行用的打孔计算器的开发，打下了后期发展的基础。到上个世纪 50 年代，老沃森退休，小沃森上台，当时从科学的领域数字计算机已经问世，小沃森的第一个重要决策，就是投资 50 亿美元，建造世界上第一台商用巨型计算机 S-360，从此世界进入了 IBM 时代！



这是一个伟大的事件，建造了一个计算机王国，当时 IBM 把第一台商用机算计放在最热闹的纽约第五大街的一个一楼，四面全部换成玻璃，里面大型机器灯光闪烁，工作人员穿着白大褂在机柜间忙碌，到现在，那些描述科学的卡通片作者的想象力，还是受到这种景象的影响。

此时，IBM 为天下共主，他的主流商业模式是垂直集成，软件是硬件的附属，使搭配硬件卖出去的，没有独立的地位。这个概念，到今天还在影响我国政府采购的模式。

IBM 凯歌四传，巨型机成为美国精神的象征。.

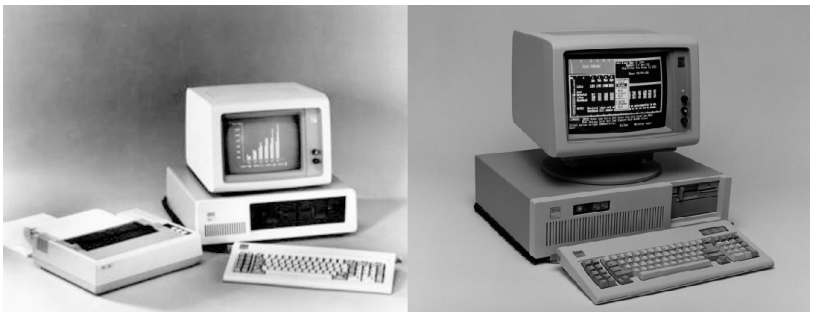
到了 70 年代，IBM 开始备受打击。

1）IBM 遭受的第一波打击：Unix 的出现

IBM 的主机业务被 Unix 引入的新观念（开放）打击,由此而来的 Sun、HP、DEC 群雄争霸对它形成重大打击。

2）IBM 遭受的第二波打击：微机的出现

1977 年的“苹果”引发了微机的出现，于是 IBM 就想招了，他想，我们也要做一个比苹果还要好地微机。IBM 为了比学赶帮苹果决定用最廉价的手段，快速撰出和苹果类似的机器，1981 年 8 月划时代的 IBM PC 诞生。



IBM PC 当初并不是他的主流产品（现在也不是），但它最大的贡献，是他的开放总线战略，引出了一大批兼容机厂家和它竞争，让世界进入微机时代，这个时代的一个显著特点，整个商业模式向水平方式发展，而软件终于成为 IT 行业的主导力量。

2，第二个时代，微软与软件的时代

IBM 选择微软 MS-DOS 作为操作系统，但微软还可以卖给兼容机厂家，这是典型的水平商业模式，从行业的角度，这是一个伟大的发展，但站在 IBM 的角度，这可是个致命错误，养虎

为患，使微软挫败 IBM 成为武林新盟主。

在 IBM 时代，软件是硬件的附属，使搭配硬件卖出去的，没有独立的地位。但是微软创造了一个很重要的商业模式：许可证模式。也就是软件作为一种独立于硬件的产品，顾客买一套电脑，就需要买一套预装的操作系统、字处理软件等，微软独创的这种 OEM 预装许可证模式，使微软成为第一个独立软件产业公司。

此后，为了保住自己的盟主地位，微软经历了多次战略转型。

1）微软第一次战略转型

1990 年代图形界面，与 IBM OS/2 达成协议，微软走低端，IBM 走高端，这是 IBM 的又一个致命错误，结果是人们只知道 Windows。

2）微软第二次战略转型

1995 年第一代互联网之争，Windows 捆绑 IE，经历世纪审判，最终打倒 Netscape，Netscape 重大决策失误，不是升级而是重写，丧失了市场机会。

3）微软第三次战略转型

2000 年应对已经成熟的 Java 解决方案与 Linux 及开源社区的挑战，微软用 Microsoft.Net 应战，.Net 主要关注网络服务模式、网络安全、互操作性（XML Web Service），这场大战如火如荼，Sun 公司这么一大堆好的解决方案，为什么难逃被 IBM 收购的命运？

4）现在，微软开始经历第四次战略转型

2005 年来自于第二代互联网及其商业模式的战略挑战，微软开始与 Google 进行全面战争，为什么会出现这个情况呢？似乎它们之间并没有多少关系呀？微软发现，在互联网模式下，广告支撑、宽带移动、云计算，草根性等低成本互联软件是新一代软件模型的特点，这自然也就成为微软目前的关注点。

这几次重大的转型，改变着人们对软件行业的看法，也改变着人们对软件商业模式的看法。

3，第三个时代，互联网时代

互联网的出现根本上改变了人们对软件本身的认识，表现为 Google 与微软争霸，主流商业模式向第三方广告模式转移。

软件行业第二个商业模式：20 世纪 90 年代互联网的兴起，形成订阅和租用模式，此时软件被看成一种服务，按照时间和使用次数来付租金。Saleaforce.com 是这一模式的大佬，在 CRM 领域迅速崛起。

到目前为止，互联网的商业概念分成两个阶段。

1）第一阶段 Web 1.0 以“信息”为中心

解决信息的可获得性与可交换性。它的典型应用方式为：门户，电子邮件，即时通信，信息存储。在这个阶段出现了四场商业大战：

- **浏览器之战**：网景与微软的互联网入口之战，网景输。
- **门户之战**：1998 年 AOL、雅虎和微软 MSN，争夺进入互联网的第一落点。

- **邮箱之战**：以 Hotmail 和 Yahoo 获胜
- **电子商务之战**：Amazon、eBay、Expedia、Priceline 是佼佼者。

值得一提的是，尽管中国没有赶上大机器时代，微机和软件时代沾了点尾巴。但是赶上了互联网时代，而且互联网行业是中国唯一个全行业把美国打得落花流水的行业。

- 几乎与美国同步，1998 年出现门户大战：新浪、搜狐、网易。
- 1999 年电子商务发芽，8848、阿里巴巴、异趣、携程、当当
- 尽管 2001 年 80%的第一代电子商务公司退市，2003 年随着 SARS 的入侵，又重新红火。
- 2004 年盛大上市，点燃网游的烈火，随后九城、巨人、金山相继上市，成为最挣钱的企业。

2）第二阶段 Web 2.0 以“知识”为中心

强调用户与用户的互动，把用户由“读”信息向“共同建设”发展。运行方式上，把“中心服务器”向“Web Services”转变。

软件行业第三个商业模式：广告模式或服务模式，这时软件就是服务，Google 是这个模式的大佬，它对用户免费，但放置搜索结果的第三方广告上付费。

在这个阶段也出现了四场商业大战：

- **搜索引擎之战**：Google、Yahoo、MSN，包括中国的百度，微软已经通过收购在中国进行火拼。
- **博客之战**：中国三大专业博客的争斗，新浪制胜。
- **视频之战**：各方都在烧钱（Google 以$16.5 亿收购 YouTube 为起点），不知道还有多少余钱可烧。
- **社区之战**：2007 年 10 月，微软$2.4 亿参股 Facebook 1.6%为起点，SNS（Social Networking Services）即社会化网络服务，也称之为 Social Network Site，即“社交网站”。1967 年，哈佛大学的心理学教授 Stanley Milgram(1934～1984)创立了六度分割理论（你和任何一个陌生人之间所间隔的人不会超过六个，也就是说，最多通过六个人你就能够认识任何一个陌生人）SNS 根据六度空间理论算出，网络的价值在于 N*N（N 是节点个数）。在中国：泛 SNS（51.com），线索型 SNS（校园、开心、海内），主题型（育婴的宝宝树等）。

Google 的盈利模式在于流量，流量越大，价值越高。

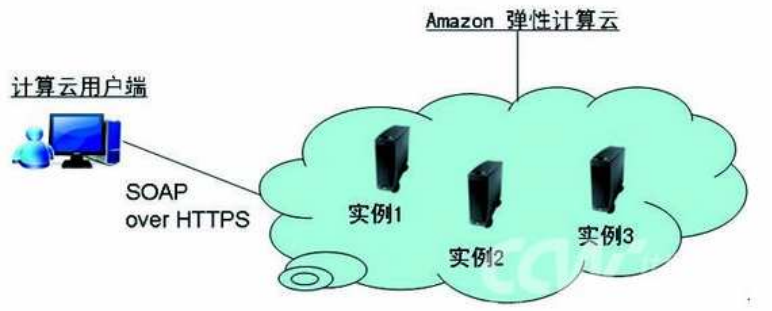
3）第三阶段 Web 3.0 可能以“情报”为中心

第三阶段的脉络还不是很清楚，一个比较强的信号就是，互联网的目标用户开始由广大普通用户转向企业和高端用户，可能以“情报”为中心，强调信息的目标、针对性、时效性、有效性，通过信息的高效整合和主动提炼，直接指导人的行为和决策。这时要强调软件的力量，强调软件与互联网的结合。互联网上的商业广告，可能会在这一阶段成为信息的最重要组成部分之一。

3，新一代互联网的走向

1）云计算

有几个重要的概念已经初见成效，首先就是云计算（Cloud Computing）。云计算是分布式处理（Distributed Computing）、并行处理（Parallel Computing）和网格计算（Grid Computing）的发展，或者说是这些计算机科学概念的商业实现。



亚马逊弹性计算云使用模式

云计算的基本原理是通过使计算分布在大量的分布式计算机上，而非本地计算机或远程服务器中，企业数据中心的运行将更与互联网相似。这使得企业能够将资源切换到需要的应用上，根据需求访问计算机和存储系统。

这是软件的一种革命性的举措，这就好比是从古老的单台发电机模式转向了电网分布式供电的模式。它意味着计算能力也可以作为一种商品进行流通，就像煤气、水电一样，取用方便，费用低廉。最大的不同在于，它是通过互联网进行传输的。云计算典型的商业模式如下图所示。

2）物联网

物联网（The Internet of things）顾名思义就是“物物相连的互联网”，其严格定义如下：通过射频识别（RFID）装置、红外感应器、全球定位系统、激光扫描器等信息传感设备，按照约定的协议，把任何物品与互联网相连接，进行信息交换和通信，以实现识别、定位、跟踪、监控和管理的一种网络。

理解物联网需要理解两层意思：

- 物联网的核心和基础仍然是互联网，是在互联网基础之上延伸和扩展的一种网络；
- 其用户端延伸和扩展到了任何物品与物品之间进行信息交换和通信。

“物联网”的概念打破了过去的传统思维。过去的思路一直是将物理基础设施和 IT 基础设

施分开：物理设施是机场、公路、建筑物等；而 IT 基础设施是数据中心，个人电脑、宽带等。而在“物联网”时代，钢筋混凝土、电缆将会与芯片、宽带整合为统一的基础设施，结果，整个世界就在它上面进行运转，其中包括经济管理、生产运行、社会管理乃至个人生活。

实现物联网需要掌握更广阔的知识。除了计算机科学与技术以外，还需要掌握包括通讯理论、控制理论以及管理理论等，包括：

- **计算机科学与技术：**也就是与互联网设计有关的理论与技术，这是首先需要掌握的知识，例如计算机原理、高级语言程序设计、网络理论、服务系统的设计与配置、数据库技术、软件工程理论与方法，IPV6 标准、XML/Web Service、分布式计算与存储技术（云计算、云存储）、嵌入式系统开发技术等。如果把物联网控制点的分布性，与云计算的分布性结合起来，就会产生无穷无尽的新能力，使整个网络应用产生革命性变化。
- **射频识别（RFID）：**这是物联网的关键技术，也是从物到网的接口，它是一种非接触识别技术，RFID 标签中存储着规范而具有互用性的信息，通过无线数据通信网络把它们自动采集到中央信息系统，实现物品(商品)的识别，进而通过开放性的计算机网络实现信息交换和共享，实现对物品的“透明”管理。本质上这是把一种称之为“标签”的芯片，嵌入被识别的物品中，作为被识别的电子标识，然后通过一种叫做读写器的装置读取标签的信息并将其传输到中央处理系统进行处理。这也是物联网专业标志性的课程。
- **信号与传输理论：**理解传感器信号的传输需要有一定的信号与传输理论作基础，例如调制、解调，信号分析、信号识别、模数与数模转换技术、数字信号处理。其中数字信号处理是关键知识，包括：数字滤波器设计、信号变换、信号检测、谱分析（包括离散傅里叶变换与快速傅里叶变换，卷积，相关性分析）、信号估计、压缩、识别等一系列的信号加工处理理论等。
- **控制理论：**由于“物联网”技术本质上是一个大规模远程控制体系，所以必须掌握一定的控制论基础，包括适用于单输入、单输出的线性定常系统的经典控制理论，适用于多输入、多输出，时变的或非线性系统的现代控制理论，离散控制理论，自适应控制理论以及智能控制技术等。
- **信息安全与网络安全技术：**包括信息安全的组织方法，网络攻击原理，安全防范技术，授权，密码学，传输层加解密技术（对称与不对称加解密原理与实现）等，这也是物联网实现的关键技术。
- **管理理论与系统工程论：**物联网技术的目标本质上是个管理问题，例如，从技术上实现识别、定位、跟踪、监控的目的是什么？还是在于有效管理。因此学生需要掌握基本的管理理论与系统工程思想，包括经典的社会组织和经济组织理论、管理过程的职能划分理论、行为科学（人的心理、行为对高效率实现组织目标的影响）、管理过程理论及科学管理理论（从系统的观点运用数学、统计学和计算机的技术，为现代管理决策提供科学依据）、系统理论（把管理业务看成相互联系的网络组织结构，通过模式分析，全面分析和研究企业和其他组织的管理活动和管理过程）等。

今天，中国的物联网在政府的推动下，正以令人难以想象的速度向前推进。

对行业的这些理解，对于我们思考未来的发展非常有意义。计算机行业仅仅走过了 60 年的道路，其变化之大已经超出了前人的想象。计算机已经成为国民经济乃至个人生活举足轻重的一部分。可以设想，在未来，计算机行业必将还会有更大的发展。

从过去，看到现在，再看到未来，我们就会知道我们所做的每一件事情的意义。我们应该用更广阔的视野来看待 IT 未来的发展，而不是仅仅蒙着头做软件。

我们国家未来的发展战略，信息化是放在一个重要的位置，有一个统计数字，我国 IT 业每年需要的软件工程师数量是 60 万，可是正规大学的计算机专业毕业生每年大概是 20 万，这就形成了一个巨大的人才缺口。

4，拥抱着变化的时代来思考

软件行业是今天变化最大的行业，不但技术方法，商业模式、应用领域也是经常发生翻天覆地的变化。因此，我们的知识在必要的时候要发生飞跃，但是，这种知识的飞跃必须是可靠的，是经过深思熟虑和实验的，是需要知道这个行业过去、现在与预测未来，同时也要反复思索，把自己的思维实践和这种知识的飞跃有机的结合起来。

我们必须“拥抱着变化来思考”，需求是在变化的，架构是在变化的，分析模式与设计模式也是在变化的，项目管理当然也是变化的。在这个大变动时期，给我们每个人提供了巨大的机会，也提出了巨大的挑战。IT 人员的优势在于他的智慧，而智慧的获得，需要实实在在的努力。

软件，是一个我们值得为它付出的行业！

二、软件工程的研究方法

1，软件工程的本质与思考方式

1）努力使复杂的事情变得简单

在研究软件工程这样一个复杂问题之前，我们不得不回过头来思考软件工程的本源。软件工程的目的是什么呢？从本质上，软件工程的目的就是想尽一切办法，把复杂的事情变得简单，变得可操作。什么样的事情复杂呢？那就是大大小小的事情纠缠在一起，一个人的脑子需要记很多事情，这就超出了人脑能够处理的能力范围。

无论是分析、设计还是组织方法，我们都是采取各种各样的方法来解决这个问题。从项目管理的角度来看我们怎么办呢？这就是任务的模块化，就像做积木，一个个的任务，每个任务都是独立的，要求都很单一，在完成这些任务的时候，不想别的事情，一心一意只把这些小任务完成好。。当我们有一大堆积木了以后就开始搭积木，搭积木的时候，就不考虑这些任务是什么细节，而只是利用接口，从更广阔的角度把他们串接起来。这就是业务流，考虑业务流的时候只考虑业务流，而不考虑任务细节。

这样就使每一件事情的考虑范围都不复杂，想大不想小，想小不想大。如果一个组织是这样来开发软件的，那么软件开发本身就很容易。

2）软件设计的分解方法

软件设计也是这样的，假定我们现在要考虑一个功能系统。首先我们得考虑现在都有一些什么方法呢？哪些方法我们可以直接使用呢？如果这些现成方法达不到我的要求，我们怎么来构思一个新方法呢？先只在这个层面考虑。如果我们决定了要自己设计一个方法，那么我们再粗略的考虑一下这个方法应该有什么样的特点，同时我们还定义一些名词和概念。然后，我们构造这个子功能系统的架构，哪些是组件？哪些是上层业务流？并且给他们恰当的命名。

有了这个架构，我们就静下心来，一个组件一个组件的设计，设计组件的时候只考虑组件本身，这样眼界很窄，所以看起来很复杂的事情做起来也很简单。

我们现在有了这些积木了，我们再把自己的思想提升一层，把这些组件串接成业务流。业务流实际上也是代码来串接的，但考虑业务流只考虑要达到的目的，就不去考虑每个积木是怎么做的。方法的声明其实就是接口，它只是告诉我们，什么名字呀？输入是什么东西呀？输出又是什么呢？有这些信息就足够了，我们就来搭它。

3）通过分解使关注点集中

这就是软件工程，不论是管理、分析还是设计，都是把所有的事情都分解开，一个问题一个问题解决，小问题关注细节，大问题关注系统，这样每件事情就都不难了。我们的工作需要有成就，关键就是要学会这种思考和处理问题的方法，努力使每一件事情都不难，每一件事情都可实现，这就是软件工程的本源。当我们认为软件工程化是个自找麻烦的事情的时候，那我们还没有

懂得它的本源，还没有真正的理解它。

2，我们的思维方式是分析还是综合

1）需求是以分析为主

在研究软件工程的时候，有一个问题时时困扰着人们，那就是我们思考问题的模式，是以分析为主还是以综合为主。当我们研究需求问题的时候，毫无疑义是以分析为主（人们已经给了它一个确切的定义，那就是需求分析）。但是，在设计这个层面，似乎就没那么简单。所谓分析，指的是当我们面对一个混沌的错综复杂的大问题的时候，我们把它一一的按子问题分解开，这样就使复杂问题就变得简单。然后仔细研究这个孤立问题的细节，当然还包括它和其它问题之间的关系（这称之为梳理）。这样一来，一个看似无法解决的问题，也就变得可以理解、可以处理、而且可以细腻的解决了。

2）设计的分解本质上属于分析

从这个角度上说，似乎研究设计问题的方法也是应该以分析为主。确实，把一个大的系统分解成子系统，每个子系统分解成功能块（组件），让他们处在合适的层次结构中，这本质上是属于分析的范畴，从方法论来说是一种演绎法的表现。

但是这还不够，系统设计事实上更需要一种综合能力。一个大型软件往往历经多个版本的历史变化，每一种产品都不会凭空从天上而降，设计师思考问题的方式会受到当前其它系统思想的感染，也会受到当前技术进步的影响，更会受到自己的历史经验、组织的管理特征、开发团队的能力等等更高层面问题的影响。

3）从整体上看设计更需要综合能力

一个系统设计，如果仅仅顾及到某个子系统或者某个模块的设计，在局部看也可能这样设计是正确的，也可能是精良的。但是从整体上看，如果各个子功能之间杂乱丛芜、互相矛盾，它仍然不能算作一个好的系统。这就需要有一种从整体上把握的能力，概括的说就是需要有一种综合能力，或者是站在更高的视角看问题的能力。另一方面，一个设计师需要把自己历史的经验，当前成功的方法和技术，各个系统之间的关系，从纵和横两方面归纳总结，把所有的问题高度压缩，形成一个简明的前后连贯的纲领，使这些知识与各种其他知识体系有一个可以互相比较的幅度与层次而得以升华，并且合理的应用到当前系统设计中去，才可能创造出更加有效的系统，从方法论来说这是一种归纳法的表现。

4）软件工程需要两种能力

一个成熟的软件工程人员需要这两种能力，而且需要融会贯通，思维方式是需要训练的，这两种思维模型必须在恰当的时候用在恰当的地方，所以一个高级软件工程人员比普通技术人员有更苛刻的要求，需要具有更多的能力。

三、对软件开发过程的认知

软件工程人员是需要一个方法学作为基础的，任何方法学都和它的应用背景有关。而这里所讨论的背景是一个最根本的问题，那就是软件开发到底是什么？

1，软件开发是到底工程还是创造？

软件开发到底是什么？它是一个工程实践，还是一个创造性过程呢？这是一个目前在软件工程界争论不休让人倍感困惑的问题。

1）以组织共同协调完成的项目必定是一个工程过程

大型软件项目是依靠一个组织来完成的，如果这个组织在管理上是无序的，开发过程是非正式和混乱的，计划期限和成本目标通常超限，项目的成功取决于个人英雄式的行为，在人员发生

变动时项目往往陷入灾难，那么，很难说这个项目会获得成功。从这个视点上说，软件开发应该是一项工程，软件工程的目的是使开发过程的各个要素，以一种统一协调的方式运转，保证时间、资金和质量这样的三角约束得以平衡，最终确保项目得以成功。

如果说我们软件开发是一项工程，那么就应该制定相应的工程标准、建立实施过程、制定项目计划以及提供可预见可重复性的基础设施。在计划实施过程中，要尽可能减少变更以保证计划的最终成功。

2）软件开发需要也应该是一个创造性过程

但是从另一方面来说，任何软件设计都是一个创造性过程。软件设计的目的，并不仅仅是用限定的费用、按时给客户他们想要的产品，而是要提供给用户从未梦想过的东西，当他得到的时候，他会认识到这是他一直想要的东西。这种软件产品设计方法论的思维核心，就是要突破已有的思维模式和行为模式，因为只有突破了这一点，我们的设计与产品才可能达到一个新的水准，达到从未有过的高度，只有创新才能让自己保持领先。。

创造性过程就要视需求变更为朋友而非敌人，因为变更可以激发更多的创造力，并且可以为客户创造更多的价值。另一方面，创造力往往更强调个人的价值，在过程中会不断修改自己的设计。对原有的业务过程也会在开发过程中再思考，对需求的理解也会随的开发的深入产生新的想法。

如果说软件是一项创造，那么就应该更加强调个人的作用，更加强调变化和修改。软件开发更类似一种写文章的不断修改和涂抹的过程，而不是一种大规模的军团有序行为。也可以说软件开发并不是一项工程行为。

综上所述，我们在研究软件架构之前，第一个需要突破的思想障碍就是：软件到底是什么？如果这个问题不解决，那么一切的方法论都成为无木之本。

3）软件开发既是工程又是创造

目前对软件开发方法论的尖锐对立和争论，从根本上说是一种以一概全的争论。我们认为，软件开发既是一个工程实践又是一个创造性过程。

- **创新的要求需要产品适应变化：**在软件开发的早期，我们必须强调创新，在这种创新背景下，软件不会产生可靠的结果，每个新设计都会包含大量的没有经过测试的新思想，而这些未经测试的新思想也经常会失败。因此，这种以创新为主的设计过程中，更加强调对变化的适应能力。
- **以工程为导向的产品开发需要严谨的过程：**当一个工程领域成熟以后，我们就有机会对大量经受了时间考验的设计进行组合和微小的修改，从思维方式上，就可以专注于解决良好定义的问题集合中的问题，而不是一切都从头开始。在这个阶段，我们将更加强调稳定的、一致的以及相互协调的软件工程过程。

4）大型软件项目的特点是工程背景中蕴含着创新：

但是，不论软件项目多么庞大和稳固，软件的创新仍然而且必须在继续，所以我们就可以把软件分成两的部分：

- **工程性过程：**大量应用经过考验的模式，通过恰当的组合和微小的修改达到目的。
- **创造性过程：**对新的各种各样的需求和设计方案进行探索。

在可行性研究阶段，软件开发大量的内容可能是属于创造性过程。而在系统开发和生产阶段，我们所遇到的问题大部分应该属于工程性过程。这样一来问题就比较清楚了。但是这两者是不可能截然分开的，更多的是工程性过程内蕴含着创造性过程。这样一来，我们在分析与设计的时候就应该思路清楚，能够很好的分开这两个部分，不同的背景应用不同的方法来解决。

5）软件工程人员要具备在复杂背景下思考问题的能力

一个软件工程人员的思维方式决不能死搬硬套，世界上的问题是千变万化的，但如果我们已经有了大量问题家族和解决方案集合，我们的解决方案就可能受到它的启发，这种启发往往是极

其珍贵的。

1.2 软件工程学概述

在软件领域，软件工程学是占主导地位的科学，而是是至关重要的工程实践方法，为什么呢？

如果我们仅仅是造一个狗窝，则不需要计划和图纸，直接用木板搭建起来就可以了。但如果是造一个两层楼，就需要仔细的设计。如果是造一个高层建筑，就需要经过仔细的分析、计算和设计。项目越大，管理、分析与架构设计所投入的成本就会越高，占的地位就会越重要。前期的低成本，会带来后期极大的高消耗。

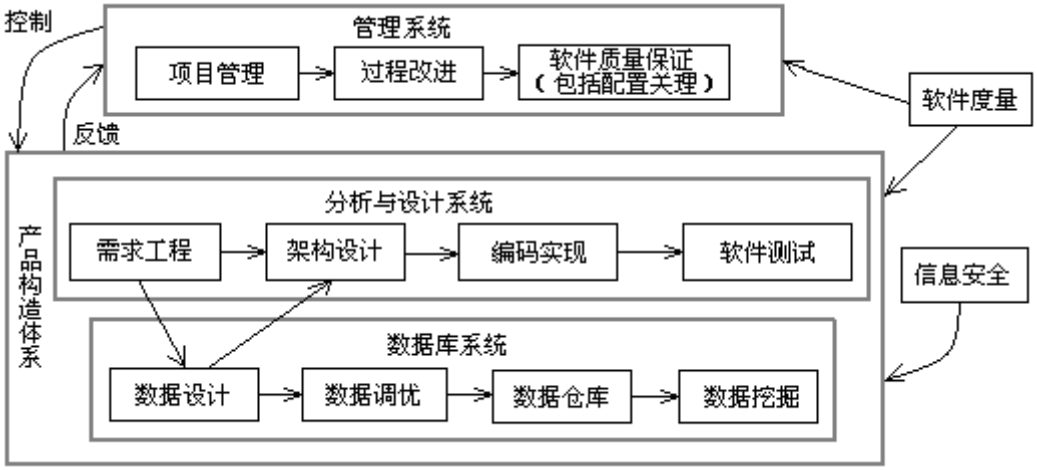
1，软件成熟阶段的特点是抓住质量：

目前国内软件行业的状况，编码这块其实还是比较强的，但是在软件工程化、质量控制、软件项目管理与过程这些方面其实非常弱，有些单位简直可以说是混乱，这些问题严重的阻碍了软件行业向更高的层次发展。究其原因，主要还是国内软件业长期处于初创阶段，一般初创阶段主要解决的是有和无的问题，在当初这个阶段，抓住技术和快速生产，迅速构建产品是必要的。

但是，今天的国内软件情况是从初创阶段向成熟阶段迈进，而一个行业的成熟主要表现在严格的管理、顺畅的流程、精髓的思考、高超的质量，在这个阶段质量成为企业的生命，但是，很多企业由于长期在游击队模型的熏陶下，随心所欲的开发几乎成了一种文化，不从思想上和方法上解决这个问题，企业再往前发展就困难了，其实很多企业都看到了这一点，关于软件开发正规化、工程化成为人们的共识，这就是为什么我们的课程几乎都处于这个层面的原因。

2，软件工程学是一个系统：

软件工程学并不是把某个孤立的节点做好就行了，而是一个系统，必须从系统工程的角度处理问题，除了关注每个节点以外，更要关注节点相互之间的关系，一个典型的软件工程系统知识点如下。



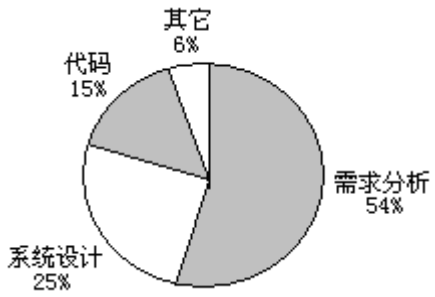
3，需求分析与架构设计是质量的关键：

没有软件工程并不是造不出软件，而是造不出高质量的软件。当然高质量软件的制造成本会比较高，但价值更高。同样功能的两个产品，往往会以 10 倍的差价卖出来，差的是什么？就是质量和创意，这两者到底哪个合算？

人们总是在问：“既然我们已经可以造出产品来了，为什么还要来什么规范？找那么多的麻烦？”我们必须知道，没有规则，随心所欲，并不是做不出产品，而是做不出高质量的产品！并

不是不能用，而是任何意外都可能发生崩溃性的结果。没有需求和架构设计不是做不出产品，但是做不出很好的有针对性的产品。

随着经济全球化进程的不断推进，要增加产品的国际竞争力，产品质量作为经济发展的战略问题变得越来越重要。软件质量问题相当程度上是以软件缺陷（defect）的形式出现的。软件缺陷是由很多原因造成的，但从整个开发周期的统计结果来看，我们会意外的发现需求分析是软件缺陷出现最多的地方，如下图所示。



换句话说，需求分析的不到位，是产生软件缺陷的最大原因。而系统分析与设计占了 Bug 的 89%，这个比例相当大。这样的统计分析告诉我们，要提高软件的质量，抓住分析和涉及两个环节就抓住了根。软件开发的过程应该遵循的原则是：软件需求工程活动和软件测试都贯穿整个软件开发生命周期。

4，只有理论的指导才会有精湛的实践：

现在软件行业的有些现状并不是对的，而是由初创时期的历史造成的，但现在已经到了必须改变的时候了。不过正规的思维方式和工作方式，在很多习惯于游击队模型的单位中很难一下子行得通，这就是有些说法并不是能马上发挥作用的原因，但并不等于这个东西不需要学。有人认为在软件开发生产部门工作，应该轻理论、重实践，我认为是片面的，我的观念是理论和实践相结合，没有理论永远不会有思想的高度，也不可能有灵活机变的实践。

这就是说，理论不是没有用，关键是要懂得它；没有理论，完全使用一种经验性的方法作实践，后面的路就会越来越窄；今天用不上不等于明天用不上；做一个课程就是为国内软件业的明天准备的，从现在的情况看，这个明天时间并不长，谁抓住了，谁就抓住了先机。

5，任何节点都要关注项目管理：

在所有的管理中，软件项目管理是最困难也是最具有挑战性的，它的难度主要表现在：首先，软件的成本主要是人力资源成本，而人是最具有不确定性的；其次，由于社会的快速发展，业务的变化必然引起需求的变化，需求变更成为影响软件质量的最重要因素；最后，软件技术也是在不断变化，具有很强的不确定性。这三者存在交集，所以没有什么软件项目是属于简单管理。

6，最需要的关注点是应对需求变更：

从目前大多数企业的情况看，需求的变更极大的影响了成本、时间和质量，而这种变更更多的又是来自于需求过程的随意性，所以，狠抓需求过程，是提高软件质量的一个关键点，而架构设计的一个重要的关注点，就是如何应对需求变更。软件行业现在也很关注复用的问题，过去软件复用主要是在代码段考虑问题，但路走到今天，人们终于发现复用应该从业务和需求开始，这就提出了需求复用和业务模式的概念，我们希望课程中体现这样一些观点和方法。

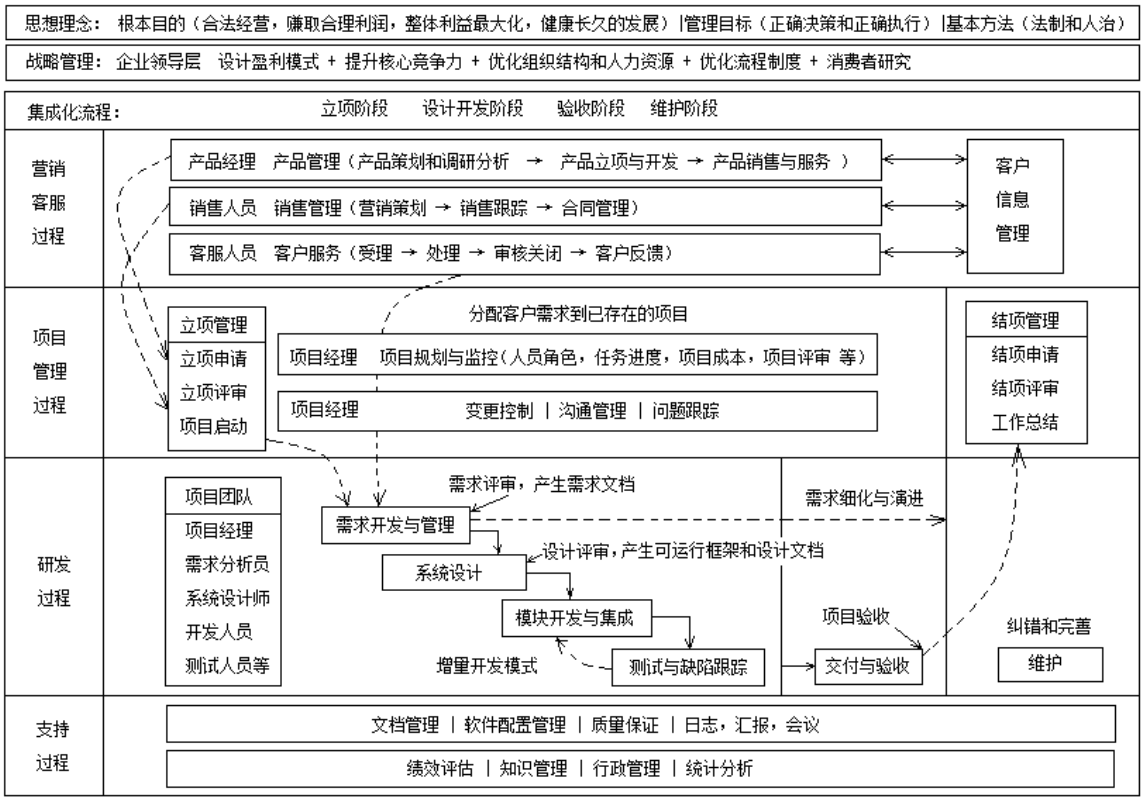
五年以前，国内软件企业的关注点还是如何写、如何做，在当时软件规模不是太大的情况下，这个关注点是正确的。但是，随着软件规模越来越大，越来越复杂，软件企业的关注点开始转向：项目管理与过程改进，软件质量控制与保证，软件需求分析与过程，软件架构设计与模式，软件

测试，软件度量等等这些与整体质量相关的问题成为新的关注点，这个关注点的转移是完全正确的。

1.3 软件生命周期设计与管理

一、集成化软件企业管理模型

良好分析与设计的第一要素就是“目的”，对于企业来说，最上层的就是企业的目的，或者说整个企业的运转可以看成一个顶层过程。我们应该看到任何过程都是为企业整体目标服务的。因此为了更好的完成项目，我们必须理解企业，理解项目团对与支持团队之间的关系，这样才可能更清晰的明确目标，把项目做的更好。一个完整的集成化软件企业管理模型，如下图所示。



二、基本的软件设计生命周期与过程

软件开发过程描述的是软件构造、部署还有维护的一种方法，成功的软件设计过程更多的是研究用户和市场，而不是技术本身。经典的瀑布式过程大致的情况是这样的：

1）收集市场数据，做市场分析。

2）确定用户，与用户交流，理解用户，理解用户的工作并与用户建立良好的关系，以便将来的设计和开发过程中经常得到他们的反馈意见。

3）建立典型用户群，通过对用户工作的了解，发现和自己设计工作有关的典型用户群。这些典型用户群应该能够描述用户工作中的一个或者几个重要环节。

4）与用户交流进一步细化典型用户群，并写出场景脚本。

5）确定软件的主要功能。

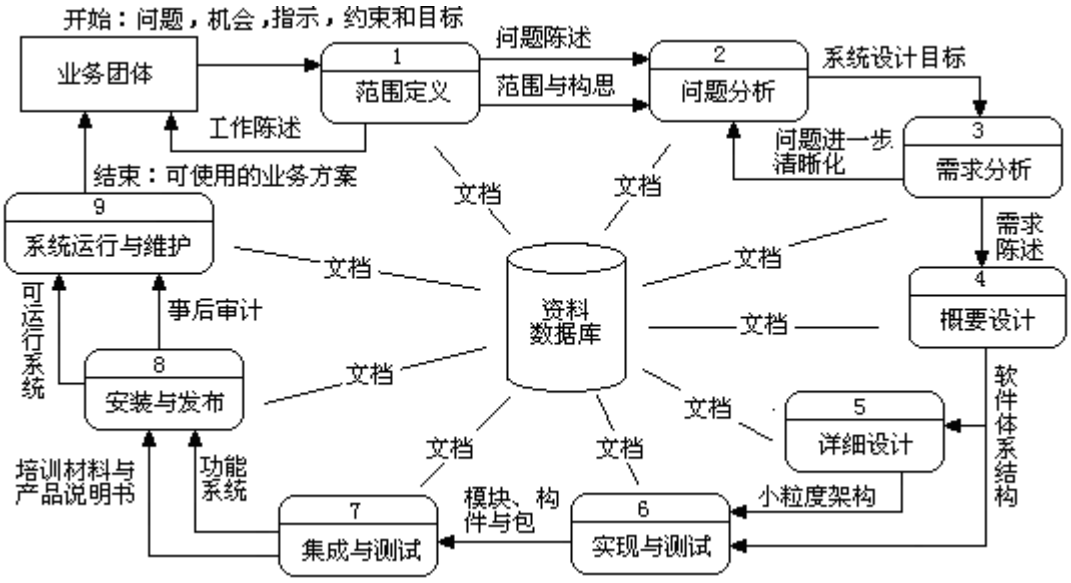
6）确定这些功能的主次，并确定优先级。

7）确定需求并写出说明书。

8）由用户群检查需求说明书，看需求说明能不能满足用户的需要。

9）进行软件体系结构设计。

可以看出来，在这个过程中软件分析占了软件设计很大一部分工作量，用户、市场、分析、设计，是整个软件设计中密不可分的几个部分，这样一个过程也称之为“瀑布式”过程，可以用图形描述如下：



成功的软件开发过程的最显著的特点，是把“研究和开发”活动与“生产”活动明确的分开。

不成功的项目大多具有以下特点：

1，过分强调研究和开发，进行太多的分析和书面研究，因此工程基线被推迟。这种状况，在传统的软件过程中倍受推崇，也是传统软件过程需要改进的原因。

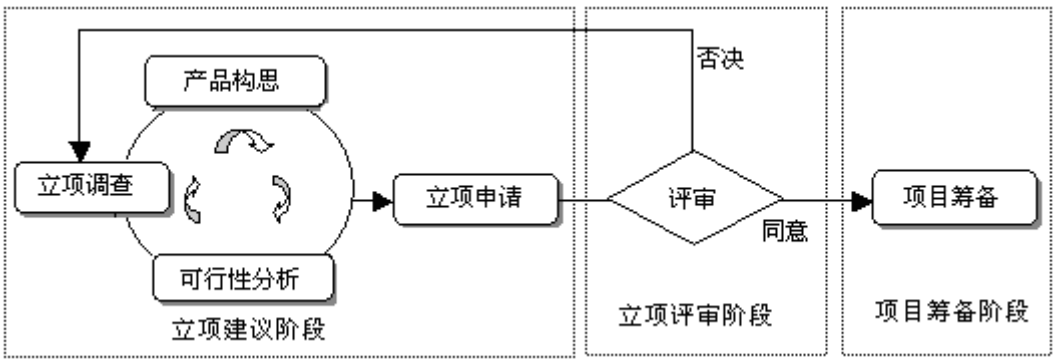
2，过分强调生产，匆忙做出判断和设计，编码人员过分卖力的做不成熟的编码，造成持续不断的删改。

成功的项目，当从研究阶段到生产阶段的转变的时候，有十分明确的项目里程碑。

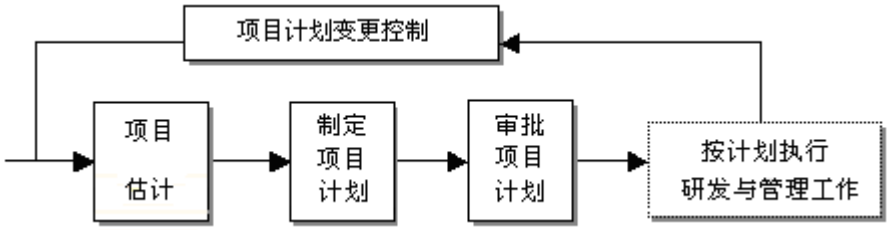
较前期的阶段，侧重于功能的实现，较后期的阶段，侧重于如和实现交付给客户的产品。

三、基本的项目管理流程

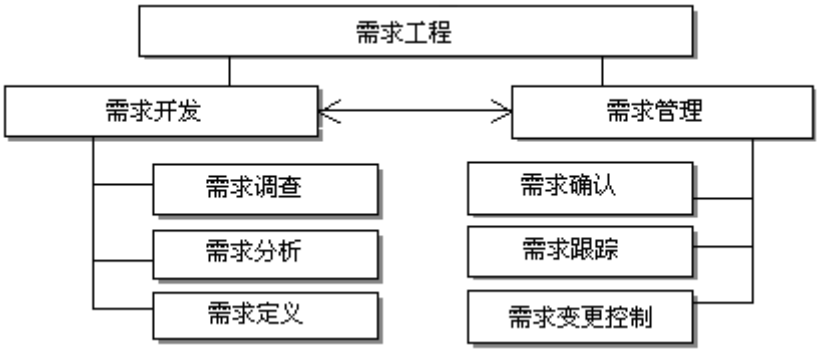
1，立项管理流程



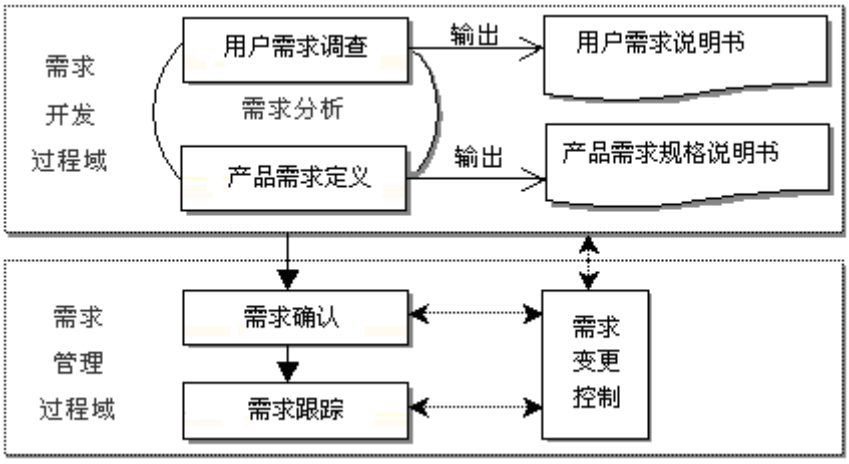
2，项目规划流程



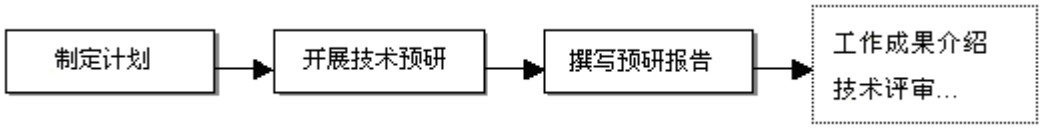
3，需求管理流程



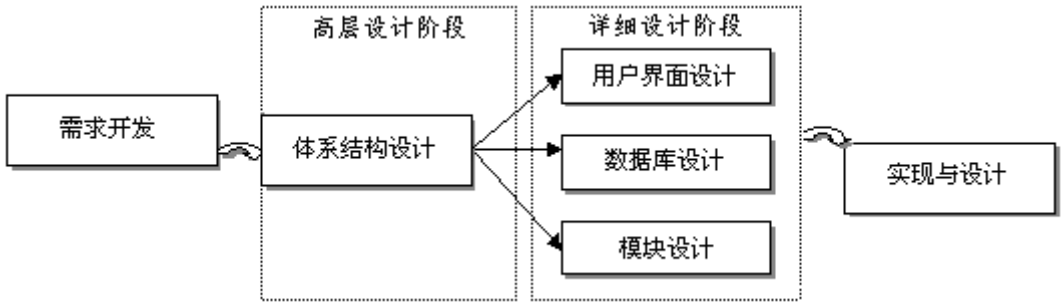
4，需求开发流程



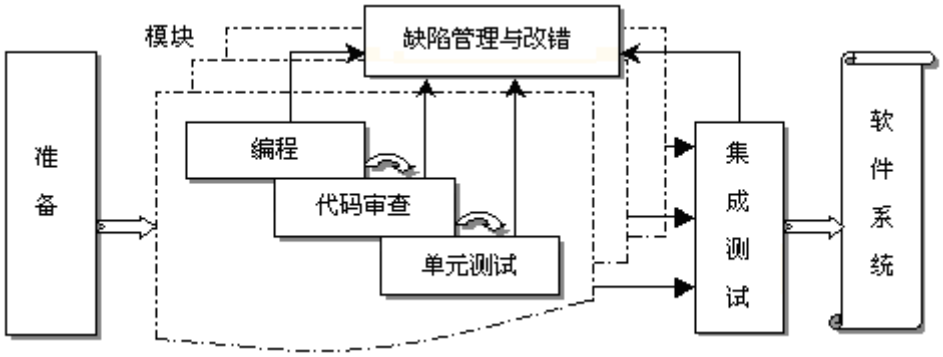
5，技术预研流程



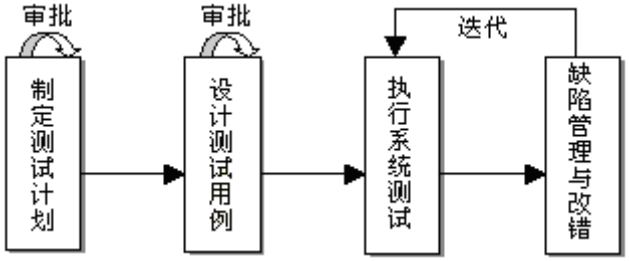
6，系统设计流程



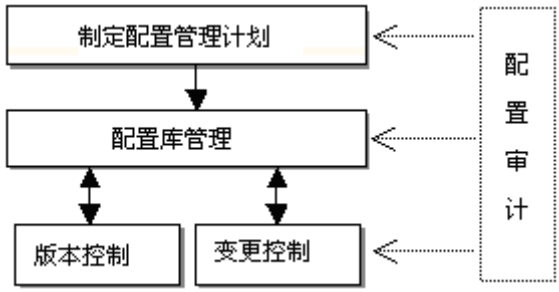
7，实现与测试流程



8，系统测试流程

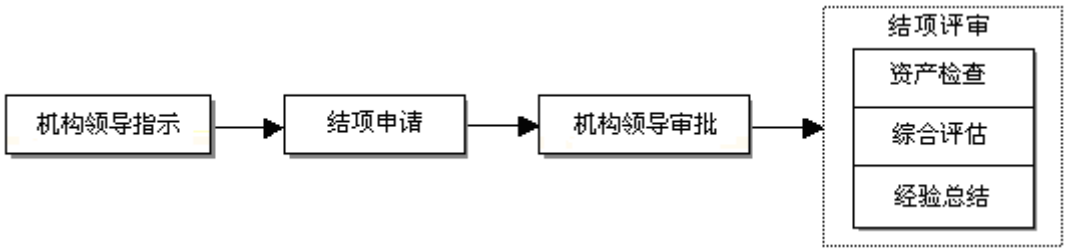


9，配置管理流程

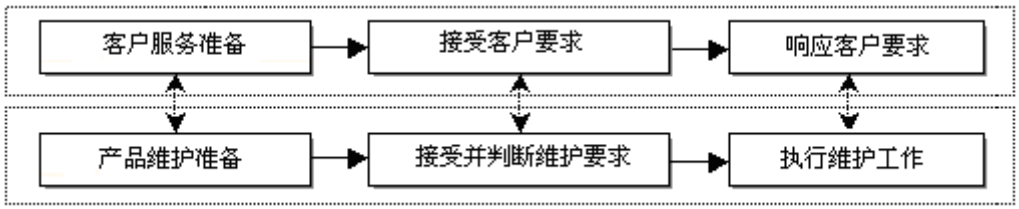


10，质量保证流程

11，结项管理流程



12，服务与维护流程



四、软件工程中的创新能力

一个优秀的软件工程人员第一个标志，就是具备创新能力。但创新不是突发奇想，而是需要对所考虑的问题深刻的分析与延伸。一个有创新能力的人应该善于从另外的领域借来想法，从而改进我们自己所面对的领域和问题。为了改进现有过程，我们可以大量吸收人类发展过程中最宝贵的思想，不断创建出更加有力的新方法。

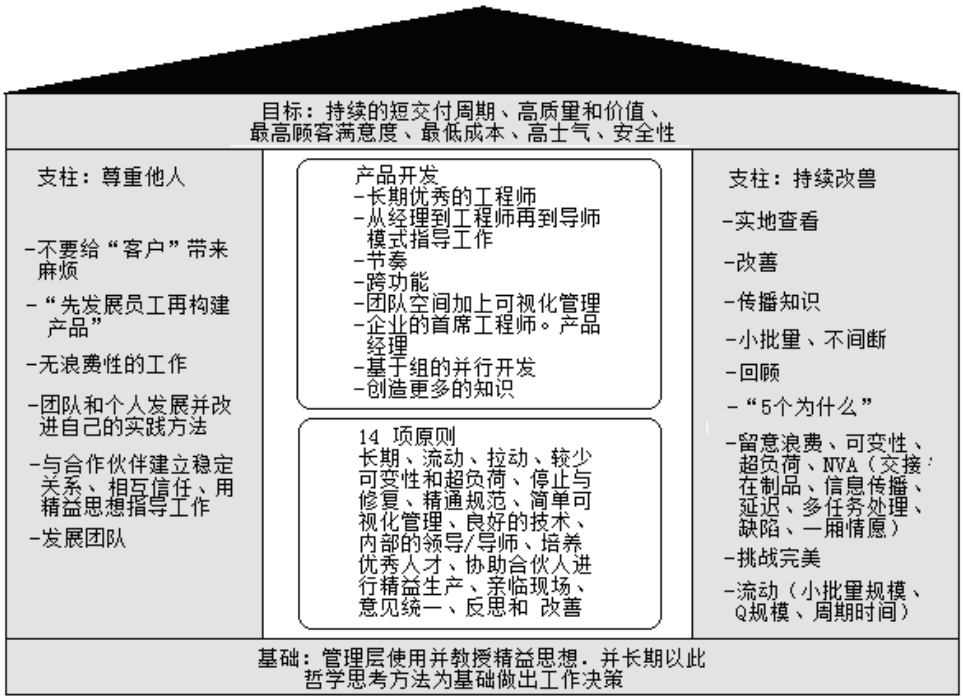
我们来看看似乎与软件不太相关的其他领域，是不是能够给我们提供有益的创新思考呢？

精益（Lean）是 MIT 研究人员确定的英文名称，用来描述丰田公司创建的丰田模式。丰田是一家具有强大适应力的公司，他们不断地自我提高：在 2006 年，丰田利润为 137 亿美元，与此同时，通用和福特公司则发生巨额亏损。2008 年，丰田超过通用公司成为销售量最大的公司，同时保持高利润率。丰田产品开发速度是一些竞争对手的两倍。丰田不断以社会和环境意识为基础进行创新，比如，开发混合动力技术等等。我们应该看到，伟大的业绩必然蕴含着先进的思想，丰田是怎么做到的呢？精益思想又会给软件过程的改进提供什么启示呢？

1，精益的支柱是什么？

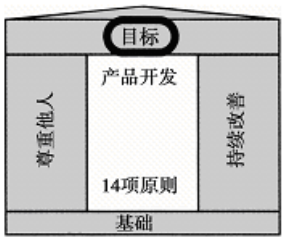
丰田模式可以通过其两大支柱简要总结为：持续改善和尊重他人。它描述述了丰田经商的基本方法：挑战一切事物。持续改善的真正价值是创造持续学习的气氛，和不仅接受变化而且拥抱变化的环境。这种环境只有在尊重他人的前提下才能实现，这就是丰田模式的两大支柱。下图利用“精益思想屋”总结了现代的丰田模式，它包括：

- 目标（屋顶）；
- 基础；
- 支柱一：尊重他人；
- 支柱二：持续改善；
- 14 项原则；
- 精益产品开发。



我们可以借用丰田的那个房子（环境）来问：什么是屋顶（目标）？什么是支柱？什么是基础？一步步思考就搭建了一个环境。继续探讨下去，我们就可以发现精益管理方法对软件项目管理来说，原来有这么大的启发作用。下面我们把这个环境中各个元素的内涵，以及如何在软件开发领域的应用做一个简要的讨论。

2，精益目标：持续快速交付价值



精益的目标是：持续快速交付价值，持续最短交付周期、（对人类和社会）高质童和价值、最高顾客满意度、最低成本、高士气、安全性。

显而易见，精益思想的整体或系统目标是以可持续且速度尽可能快地实现“概念兑现”或“定单兑现”。也就是说，所有过程以越来越短的周期快速交付价值，同时达到高质量和高士气。

丰田争取减少周期时间，并不是通过捷径、降低质量或以无法持续或不稳定的速度进行开发；而是通过不懈地持续改善来减少周期时间。这就需要公司具备真正意义上的“尊重他人”文化，保证员工在其中有挑战和改变现状的安全感。

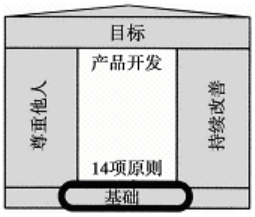
因此，精益的重点是在接力棒，而不是赛跑者。换句话说，精益思想改进的重点，是优化系统互相之间的关系，而不是仅仅改善某个节点。是去除瓶颈以加速交付客户产出价值，而不是进行最大限度利用人或机器的局部优化。这个“整体目标”是如何取得的呢？

- **开发**：通过学习在竞争中获胜，学习可以形成许多有用的知识，并进行有效的记忆和使用。
- **生产**：通过改进在竞争中获胜，将重点放在短周期、小批量和排队、停止和修复问题根

源、排除所有浪费（例如等待、交接等）上来。

3，精益基础：精益思想中的经理/导师

精益的基础是：管理层使用并传授精益思想，并将此长期哲理作为决策的基础。大多数丰田新员工在上岗前都要先经过几个月的培训。在这一期间，他们学习精益思想的基础，学习判断“浪费”，并在丰田公司的不同部门进行实践工作。这样．新的员工就具备了一下素质：

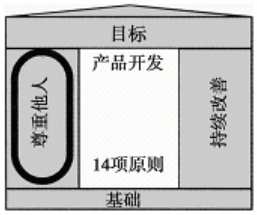


- 学会“纵观全局”。
- 学会观察精益思想在不同领域如何应用。
- 学会“改善”的思维模式（持续改善）。
- 领会丰田的核心原则：“实地查看”和“工作现场”。

实地查看（Go See）意思是指领导者（特别是经理们）要“亲身实地到工作现场观察”，而不是坐在办公桌前或相信可以从报告或数据中得知实际工作情况。这一点与意识到工作现场的重要性有关（亲身到实际生产价值的最前线）。

丰田的经理们都经过精益思想、持续改善、根源分析、变化性统计、系统思考的培训，并将这些思考工具传授给他人。从这一点上，我们深切体会到为了精益的成功实施，管理层必须具备的品质，有了这个品质，才可能在整个组织中获得有意义和持续的成功。

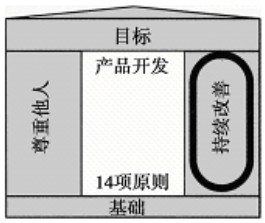
4，支柱一：尊重他人



尊重他人听起来比较朦胧，但是在丰田公司内部包含此概念的具体做法和文化。这些做法和文化广泛地反映出对道德的尊重和敏感、不让员工做浪费性的工作、真正的团队工作、指导发展高技能的员工、人性化工作和环境、安全和整洁的环境、哲学式完善管理团队。

5，支柱二：持续改善

持续改善是根据以下几个想法产生的：



- 实地查看
- 改善
- 完美挑战
- 工作流动性（包含在 14 项原则中）

1）实地查看（Go See)

走到源头，到真正产生价值的工作地点（现场），根据实际情况做出正确决策、建立统一意见、以最佳速度达到目标。

在许多管理文化中都找不到“实地查看”原则，但这是个关键和基本的原则。在丰田内部手册中，“实地查看”多次被丰田经理们引用。在精益思想文化中，所有的人特别是经理们（包括高级经理），不应当把时间都花费在自己的办公室或会议室中，而是应利用时间搞清楚实际清况，帮助改进（消除来自直接信息中的失真部分），管理层应当时常到工作地点考察并获取真正有用的信息。“真正工作前线”（现场）并不是指靠近实际工作场所的地方，也不是指走访其他的经理们。“工作”主要也不是指管理费用或会计等工作，而是指客户关心的增加价值的工作，包括工程、汽车设计、生产、客户服务。

实地查看的深层意思也被扩展开，它指解决问题的根源，而不是坐在办公桌后考虑问题。实地查看不仅单指走到源头并发现实际情况，做出直观决策，它也意味着（当你在实际工作地时）

为使目标达成一致，所进行的改进试验。实地查看的全面含义是让人（特别是经理们）经常性地接触实际生产价值的工作地点，与实际下作人员建立信任关系，帮助他们解决问题。

2）改善

为了改善所以进行无休止地改进。改善既是思考模式，又是实践方法。作为思考模式，它认为“我的任务就是做好我的工作，并提高我的工作”，并且“为了工作本身而不断地改进。” 改善更适用于作为实践方法，它是指：

- 选择同意尝试的实践团队和／或产品团体，直到他们彻底掌握该技巧；
- 不断地进行试验，直到找到最佳方法；
- 永久重复。

3）5 个为什么

“5 个为什么”是改善中简单并广泛使用的工具。它帮助培养解决问题和分析根源的能力。为了应对问题或缺陷，团队考虑“为什么？”至少 5 次。比如：

问题：开发人员没有进行代码重构来保持代码的可维护性。

- 为什么？我们对加快工作速度感到有压力。（第 1 个为什么）
- 为什么我们感到有压力？因为我们工作速度很慢。（第 2 个为什么）
- 为什么工作速度很慢？因为代码很复杂，很难开展工作。
- ……

这些问题可能会有多个相关的答案，因此可以创建“ 5 个为什么”图来显示答案分支，或利用更有条理性的“鱼骨图”来帮助分析。

“5 个为什么”的重点不在于技巧本身，也不是数字 5 ，而是丰田已经深入人心的“停止与修复”解决问题根源的思维模式和文化。培养员工成为深层问题的解决者，不是去适应问题，而是更深人地思考问题。“实地查看”与“5 个为什么”也有深层的联系：人们很容易猜错或给出的答案并不足以说服别人，除非他们亲自到发生问题的现场了解情况。

4）价值和浪费

价值 ：我们的行动需要在某一时刻创造出客户愿意购买的产品。换句话说，产品价值是由外部客户判断的。

浪费：不增值且消耗资源的其他时间或行动。浪费来自于负担过重的工人、瓶颈、等待、交接、一厢情愿、信息散布等。

在精益思想中，有一项数据分析就是估算价值率：

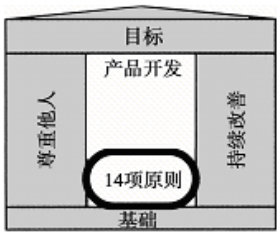
价值率＝总价值产生时间／总交付周期

5）没有决定性的过程

改善和横向传播知识的意义是：在工作中没有从中央过程团体发布的决定性或正确“确定”过程来遵循。 改善包括学习和掌握工作协定，但是它们都随着横向传播知识的模式普及和发展。

“领导怎么说我就怎么做!”有这种思维模式的人不会喜欢精益思想。引用丰田 CEO 的一句话“丰田模式的根基就是对现状的不满，你必须经常自问为什么我们要这么做。”

6，14 项原则



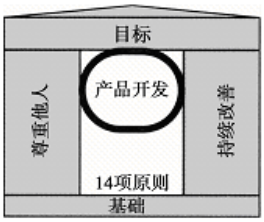
尊重他人和持续改善这两大支柱并不是精益原则的全部。还有其他强有力的精益原则组成了精益的整体系统，其中有些甚至是两大支柱中的重要元素。但是我们还需要把这些元素集合成系统，并且必须坚持每天实践使用。

下表列出了精益思想的 14 项原则。

	原则

1	管理层决策必须以长期哲理为基础．即使为此牺牲短期财务目标也在所不惜
2	做到流动，批量规模越来越小，周期越来越短，从而快速交付价值并暴露弱点
3	使用拉动系统．越晚做决定越好
4	均衡工作：减少变化和减轻负担以消除不平均
5	建立停止并修复的文化．教会每个员工部要有条不紊地研究问题
6	掌握规范（实践）以实现改善授权于员工
7	使用简单可视化管理揭露问题
8	只使用适合员工和过程的经过充分测试的技术
9	栽培那些充分理解工作、实践理念并教授他人的内部员工．让他们成为领导
10	栽培信奉公司理念的特殊人才和团队
11	重视公司的合伙人，挑战他们并帮助他们进行改善
12	亲身到工作现场查看，彻底了解情况并提供帮助
13	以共识为基础做决策，彻底考虑所有可能选择，快速实现
14	通过不断反思和改善，成为一个学习型的组织

7，精益产品开发



两大支柱和 14 项原则是精益思想的核心。但是，还有其他一些原则和实践是“学习在竞争中获胜”，对精益产品开发特别有效，这两个主要过程是：

- 产品开发；
- 生产。

丰田的产品开发与生产的效率非常高，他们是怎样达到这些的呢？答案就是：“以学习在竞争中获胜”。

精益产品开发（LPD）注重创造更多有用的知识和不断学习，而不是单纯的竞争。下图显示了精益实践是如何在 LPD 中实现“以学习在竞争中获胜”的。

1）高价值、低成本学习

不是所有的知识或信息都有价值，理想情况是创造经济价值的新信息。这就非常具有挑战性，因为这是个发现的过程，你会有所得也有所失。我们应该增加创建信息的价值，并降低创建信息的成本。

类似的，在软件开发中有些信息的发现是具有高价值的：

- **重视不确定的事物**：在优先准则是选择可以提前实现和测试那些具有不确定性或有风险的事物。如果结果的可预测性比较低，那么反馈的价值就很高。可预测的事物不能给我们带来更多的经验。相反，经验的积累往往来自那些不可预知的事物。
- **重视提早测试和反馈**：信息延迟就会付出代价，但非常不明智的做法是只在长生命周期末尾进行一次测试（在局部优化的错误引导下，认为这么做会降低测试成本）。如果在压力测试过程中发现问题，代价就会是非常高昂的。如果在开发进行了 18 个月后，发现某个主要架构决策是有缺陷的，那就是一场灾难。短周期和提早反馈回路是非常重要的，提早实现不可预测的事物，并在短周期内进行测试，那么就可以降低延迟产生的成本。

在软件开发中有些信息的发现是具有低成本的，这也包含降低学习的成本。比如：

- **重视大型自动测试**：用来了解缺陷和行为。设备费用比较高（相对于如果你现在使用人工测试），但是再执行成本几乎为零。
- **重视持续集成**：用来了解缺陷和缺乏同步的情况。通过在小批量中经常性的集成，由于受到集成大套代码带来的非线性结果影响，团队减少了平均额外成本。
- **重视得到专家指导和传播知识**：用来减少再发现的成本。

2）节奏

在生产和开发中，有规律的节奏是精益的原则，在精益生产中，它被称为“生产节拍”。。

节奏中有一些非常基本而且人性化的东西：人们喜欢或希望自己的生活和工作有节奏，并且喜欢或希望这些节奏是固定的。简单地说，工作有节奏可以提高预测、计划和协调的能力。在更深的层次，它也反映我们生活方式中的节奏。

假设某个团队工作没有节奏，却有无穷无尽的工作，不断的加班。他们可以在任何时候交付运转测试过的系统。假如他们想举行协调计划会议（因为有多个团队参与），还想举行回顾会议，这就会带来相当多的麻烦。

清晰节奏会带来很多益处：

- 时间周期加强了节奏。
- 研发工作通常是“模糊无界限”（或极少界限）的工作。当团队知道评审会议将在两周后（3 月 15 日）召开，这就给模糊的工作下定了界限并增加了团队的注意。
- 假设你在上大学，下周一要交作业。那么你什么时候开始做呢？很多人会说“快到周一的时候再说。”这就是学生综合症，时间周期正好起了平衡作用。
- 如果团队必须在两周内交付做完成的东西，那么现有工作方式中的浪费和低效将会变得非常明显。比如，并行开发（不是连续开发）导致更快价值交付、更短的反馈回路和其他的益处，因此时间周期创造了一种变革力量，向拥有跨功能特性团队的并行开发转变。
- 在大型多团队开发中，不同团队间的计划会议是非常必要的。在每个时间盒的第一天举行计划会议简化了他们的协调工作。
- 时间周期简化了时间安排，你知道什么时候该参与计划和评审会议。这对于繁忙的产品负责人来说非常有用，他可以提前将需要参加的活动记人日程表。
- 人可能对时间变化比对范围变化更敏感，“太晚了”比“这比我想要的要少”会留有更深的印象。

3）利用可视化管理的团队办公空间

精益产品开发鼓励使用团队办公空间，内部没有隔断或墙壁，跨功能的团队成员在这里共同工作，企业的首席工程师也在这里工作。墙上挂满了项目和工程信息的大型实物展示，来支持可视化管理。团队办公空间与团队成员在不同办公室或隔断中办公不同，在那样的环境下，团队成员存在沟通障碍。

4）有商业控制权和具有企业家精神的首席工程师

我见过很多产品开发团队中的管理者，通常不是掌握现代技术和具有极深工程造诣的工程师。丰田的做法非常不同。他们的产品开发是由优秀的首席工程师带领，该工程师具有“极高的技术造诣”，并且负责新产品获得商业成功。在丰田公司，产品和工程领导在具有企业家精神的首席工程师身上结合为一体，他了解市场、产品管理、利润和工程。

5）基于组的并行工程技术

在丰田中不是让一名工程师或一个团队创建一个系统的设计，而是由不同团队设计几个不同的系统并行实施 。这些不同选择的组合被深人研究、合并，最终过滤而出或聚合成一个方法，一开始还是很多不同选择的组合，然后变成较小的组合，以此类推。他们通过增加选择和组合而实现以学习在竞争中获胜。

8，精益生产经验可以帮助开发吗？

新产品开发（NPD）或研发（R&D）不是可预测的重复生产（制造），不恰当的假设两者具有相似性，正是错误应用 20 世纪初制造业的“规模经济”管理实践于研发的原因之一，比如，长生命周期开发和大批量需求传递，带来软件产品开发中的许多问题。

在精益生产中使用的一些原则和想法（包括短周期、小批量、停止与修复、可视化管理和排队论），在现代软件开发中都得到成功地应用。这是因为以互联网中心的现代软件开发更像是产品研发而不是大规模生产。离开“规模经济”方式转向小批量、无浪费的流动和灵活生产。但是

令人啼笑皆非的是这些并没有被许多高层研发管理者使用，他们还继续使用旧式“规模经济”制造管理的方法。

精益思想适用于大型产品开发的结果超过了我们的预想。精益思想不只是工具，如看板、可视化管理或排队管理、完全消除浪费等工具。而是依赖于经理/导师基础之上，以尊重他人和持续改善为支柱。成功地引人精益思想需要多年的努力，需要广泛传播知识和指导员工。最重要的是将这些要素集合成为系统，而且必须坚持每天持续实践。

1.4 软件分析和设计的方法学问题

由于设计的源泉来自于软件分析，不同的分析与设计方法，将会带来完全不同的设计思路。从方法学的角度来讲，“分解与抽象”是软件工程方法的思想精髓，目前分析和设计方法主要分为面向过程的方法（结构化方法）与面相对象方法两种。

一、面向过程的方法

面向过程方法又称为结构化方法，起源于 20 世纪 70 年代，主要由面向过程分析、面向过程设计和面向过程编程三部分组成。

面向过程分析：帮助开发人员定义系统需要做什么（处理需求），系统需要存储和使用那些数据（数据需求），系统需要什么样的输入和输出，以及如何把这些功能结合在一起来完成任务。面向过程分析的主要工具是**数据流图（DFD）**,这是一种显示面向过程分析中产生的输入、处理、存储和输出的图形模型。

在现代面向过程设计中，也引入了事件的概念。

面向过程设计：面向过程设计是为下列事务提供指导：程序集是什么，每个程序应该实现哪些功能能，如何把这些程序组成一张层次图。面向过程设计的主要工具是**结构图**，这是一种表达程序模块层次的图形模型。

面向过程编程：具有一个开始和结束的程序或者程序块，并且程序执行的每一步都由三部分组成：顺序、选择或者循环结构，实现这种思想的最典型的语言就是 C。

整个面向过程设计的根本**目标是**：把复杂的系统分解成简单模块的层次图。

面向过程的方法也称之为结构化方法，它的特点是：过程观点，源于对问题域处理过程的抽象，软件构成的基本组织单元模块，模块代表着一种变换，模块之间通过接口实现调用和数据传递，软件结构具有树状的层结构特征。

它的**局限性**在于：从分析到设计，模型之间的转换不能够平滑过渡，造成追溯与确认验证的困难。

二、面向对象的的方法

面向对象的方法由面向对象分析（OOA）、面向对象设计（OOD）以及面向对象编程（OOP）三部分组成。

面向对象方法与面向过程方法根本区别，是把信息系统看成一起工作来完成某项任务的对象集合，**而对象是系统对消息作出做出响应的事物**，所以面向对象方法中最值得关注的不是它该做什么，而是它如何做出反应，也就是消息，这是和面向过程方法的根本不同。

面向对象分析（OOA）：定义在系统中工作的所有类型的对象，并显示这些对象如何通过相互作用来完成任务，主要工具是**统一建模语言（UML）**。

面向对象设计（OOD）：定义在系统中人机进行通讯所必需的所有类型的对象，并对每种类型的对象进行细化，以便可以用一种具体的语言来实现这些对象。

面向对象编程（OOP）：用某种具体语言（C++、Java、C#、C 的对象模块等）来实现各种对象的行为，包括对象间的消息传递。

这里的关键是类图：用面向对象的方法显示系统中所有对象所属类的图形模型。

面向对象的方法起源于 20 世纪 60 年代挪威 Simula 编程语言的开发，80 年代建立了整体框架，90 年代由于 C++的崛起和 UML 被广泛认可，逐步成长为一种主要的和现代的分析和设计方式。

面向对象的方法和传统面向过程方法有很大不同，它的思维方式不是以设备结构为基础，而是利用可感知的对象来思考，对人而言，这是更加自然或者直观的。但是，如果只是把传统概念简单包装一下换成对象方法（比如封装），并不能得到实实在在的好处，反而使 OO 很难理解，面向对象的方法关注的是事件、重用和继承，关注的多态，它自己有一整套独特的思维方式，这和面向过程方法是根本不同的。

90 年代中期以后，这种关注带来了许多新的思维，有代表性的就是设计模式的提出，设计的质量更高，系统的优化空间更大，这就是说应用面向对象的方法，将会给我们提高设计质量带来巨大的好处

由于面向对象方法把对象看成系统对消息做出响应的事物，这种与面向过程完全不同的看待计算机系统的方法，必然导致完全不同的分析、设计和编程方式。

OO 方法**特点**：对象观点，源于现实问题域事物的映射，软件构成的基本组织单元是类，类封装了数据与行为，并可能具有自身的状态。通过消息实现不同对象的交互，从而完成系统应用的功能。软件结构呈现丰富的结构风格与模式。

有个问题，是不是使用面向过程的程序语言（比如 C），就一定要使用面向过程方法，实践表明并不是这样的，面向对象更多的是一种思维方式，面向过程的语言只需要略加改造就可以应用这种思想（继承、封装、多态），国外在这些方面有很多成功的案例和讨论，国内在一些大型嵌入式项目中也有很好的尝试。

三、面向构件的方法

做为方法论层面的讨论，我们还必须研究一下所谓面向构件的方法，构件也称为组件，它包含了许多关键理论，这些关键理论解决了当今许多备受挑剔的软件问题，这些理论包括：

1，构件基础设施

2，软件模式

3，软件体系结构

4，基于构件的软件开发

构件可以理解为面向对象软件技术的一种变体，它有四点原则区别于其它思想，封装、多态、后期绑定、安全。从这个角度来说，它和面向对象是类似的。不过它取消了对于继承的强调。

在面向构件的思想里，认为继承是个紧耦合的、白盒的关系，它对大多数打包和复用来说都是不合适的。构件通过直接调用其它对象或构件来实现功能的复用，而不是使用继承来实现它，事实上，在我们后面的讨论中，也会提到面向对象的方法中还是要优先使用组合而不是继承，但在构件方法中则完全摒弃了继承而是调用，在构件术语里，这些调用称作“代理”（delegation）。也就是说，类与构件最大的区别在于抽象颗粒度的不同，而前者具有继承特性而后者没有。

实现构件技术关键是需要一个规范，这个规范应该定义封装标准，或者说是构件设计的公共结构。理想状态这个规范应该是在行业以至全球范围内的标准，这样构建就可以在系统、企业乃至整个软件行业中被广泛复用。

构件利用组装来创建系统，在组装的过程中，可以把多个构件结合在一起创建一个比较大的实体，如果构件之间能够匹配用户的请求和服务的规范，它们就能进行交互而不需要额外的代码，这通常被称之为“即插即用”（Plug-and-Play），这也是后期绑定的一种形式。

构件方法并不是某种新的思想，它能不能实现，依赖于“构建基础设施”是不是能够创立，这种基础设施的最大特征，就是需要建立一个对语言和平台不敏感的通用标准。早在 20 世纪 90 年代，主要平台供应商就把它们的未来赌在了这个问题上，特别是微软的 COM、Sun 的 EJB、CORBA 的请求代理。

微软在整个 20 世纪 90 年代一直在推广构件对象模型（Component Object Model ,COM）。COM 是一种用于说明如何建立软件组件的规范，由于使用了统一的接口规范，不同的开发人员创建的 COM 组件，可被组合进不同的应用程序中，而且这些 COM 组件所使用的语言，可以是完全不同的。

OLE 和 ActiveX 技术都定义了基于 COM 的构件接口，COM 定义了一组 API 和一个二进制标准，让来自不同语言、不同平台的彼此独立的对象可以互相通信。

COM 技术实际上是一种分布式开发的技术标准，这对于一个开发软件的集体来说，是有很大的意义的。

DCOM（Distributed Component Object Model）分布式组件对象模型，后来发展为 COM+，这是一种分布式应用程序集成到网络的技术，一个分布式应用程序由多个进程组成，这些进程协作完成一项工作。DCOM 为 COM 组件之间的通信透明的提供可靠、安全和有效的支持 D，COM 可将其应用程序分布到最适合于其顾客和应用程序的位置。

Sun 的 Java 语言已经是一个在业界具有很大影响力的语言了，为了在跨平台应用领域取得竞争优势，它在 EJB 的基础上发展了一系列构件模型，确实这些发展为团队开发提供了便利，但它还是缺乏一项关键功能：易于和其它语言交互，对团体项目而言，这可能是一个严重的限制，而且 Java 应用程序编程接口（API）不是标准的，比如 JDBC 这样的流行技术，必须要供应商开发相应的 API 作为补充。

公共对象请求代理结构（CORBA）是一种用于分布式基础设施的开放系统标准，它得到了多个供应商、行业财团和正式实体的广泛支持，从一开始，CORBA 就支持对象和构件模型，可以支持面向消息的中间键和特定领域的 API 标准，甚至微软的或者 Sun 的产品，可以通过 CORBA IIOP 实现一定程度的互操作。但是，CORBA 也存在不足，比如内存泄漏、一致性、性能都存在一定的问题。

面向构件技术的特色在于：迅速、灵活、简洁，面向构件技术之于软件业的意义正如由生产流水线之于工业制造，是软件业发展的必然趋势。软件业发展到今天，已经不是那种个人花费一段时间即可完成的小软件。软件越来越复杂，时间越来越短，软件代码也从几百行到现在的上百万行。把这些代码分解成一些构件完成，可以减少软件系统中的变化因子。

1.5 在信息技术战略规划（ITSP）中的软件系统

一、利用信息技术战略规划整合客户需求

信息技术战略规划（ITSP）的核心思想简述如下：

在信息时代知识经济的背景下，正确的结合 IT 规划，整合企业的核心竞争力，在新一轮的产生、发展中取得更大的市场竞争力是必要的。

信息化的问题首先是企业管理层概念的问题。企业管理层的重视，和对信息化的高度认同是企业信息化的关键所在。当前国内很多企业管理层很关心资本运作的问题，而对很多国企业而言，

管理层最关心的是扭亏增盈。信息化建设投入大、周期长、见效慢、风险高，往往不是企业需要优先解决的问题，导致管理层对信息化的重视程度不够，无法就信息化建设形成共识

企业管理信息化必然带来管理模式的变化，如果对这种变化不适应，又抵触心态，或者仅是为了形象问题，赶潮流搞信息化。或者由于国家提出信息化带动工业化，信息化成为一种时髦，信息化工程往往成为企业的形象工程。结果软件体系结构的设计仅仅是企业目前业务流程的复制，并不可能给企业带来实实在在的好处。

有些公司缺乏统一完整的 IT 方向，希望上短平快的项目，立竿见影，跳过系统的一些必要发展阶段，导致系统后继无力，不了了之。由于方向不明确，企业内部充斥着各种各样满足于战术内容的小体系，并不能给企业带来大的好处。

有些公司对信息化建设的出发点不明确，在各个方案厂商铺天盖地的宣传下，不能很好的把握业务主线，仅是为了跟随潮流，既浪费了资源，同时也对后继的信息化造成了不良的影响，甚至直接导致“领导不重视”这样的后果。

如今国家正在大力推广企业信息化。然而人们大多从技术角度来谈论信息化和评价解决方案，他们往往脱离了企业的实际需要，以技术为本是不能根治企业疾病的。企业依然必须明确自己的核心竞争力。明确一切的活动和流程都是围绕让核心竞争力升值的过程。IT 规划意识如此，必须也企业核心、业务为本。再结合公司的实际情况。开发自己需要的系统。

信息化的建设难以对投入产出进行量化，难以进行绩效评估，CIO 们无法让企业管理切实感受到信息化带来的直接效益——经济效益、社会效益。

战略规划是一套方法论，用于企业的业务和 IT 的融合以及 IT 自身的规划。必须满足如下要求：

1.先进性：采用前瞻性、先进成熟的模型、方法、设备、标准、技术方案，使建议的企业信息方案既能反映当前世界先进水平，满足企业中长期发展规划，又能符合企业当前的发展步调，保持企业 IT 战略和企业战略的一致性。

2.开放性：为保证不同产品的协同运行、数据交换、信息共享，建议的系统必须具有良好的开放性，支持相应的国际标准和协议。

3.可靠性：建议的系统必须具有较强的容错能力和冗余设备份，整体可靠性高，保证不会因局部故障而引起整个系统瘫痪。

4.安全性：建议规划中必须考虑到系统必须具有高度的安全性和保密性，保证系统安全和数据安全，防止对系统各种形式的非法入侵。

5.实用性：规划中建议的系统相关必须提供友好的中文界面的规范的行业用语，并具有易管理、易维护等特点，便于业务人员进行业务处理，便于管理人员维护管理，便于领导层可及时了解各类统计分析信息。

6.可扩充性：规划不仅要满足现有的业务需要，而且还应满足未来的业务发展，必须在应用、结构、容量、通信能力、处理能力等方面具有较强的扩充性及进行产品升级换代的可能性。

为了实现这样的规划，我们必须注意到，软件设计既是面对程序的技术，又是聚焦于人的艺术，成功的软件产品来自于合理的设计，而什么是合理的设计呢？

一个软件架构师最重要的问题，就是他所设计的产品必须是满足企业战略规划的需求，能够帮助企业解决实际问题的，因此一个合理的设计，首先要想的是：

Who：为谁设计？

What：要解决用户的什么问题？

Why：为什么要解决这些用户问题？

这是一个被称之为 3W 的架构师核心思维，如果这个问题没搞清楚，就很快的投入程序编写，那这样的软件在市场上是不可能获得成功的。

Who? Waht? Why? 这三个问题看似简单，但实际上落实起来是非常困难的。我们经常会看

到一些产品，看似想的面面俱到，功能强大，甚至和企业目前的业务吻合得非常好，但为什么最终没有给企业带来实实在在的好处相反带来麻烦呢？一个专家感觉非常得意、技术上非常先进的系统，企业的使用者未见得感觉满意，这些情况在我的实践中屡见不鲜，即使一些知名的公司在设计的产品，往往都不能很好地把握，这足以证明我们必须下功夫来面对它。

那么，我们该怎么来做呢？

为谁设计，表达的是我们必须认真研究企业的业务领域，研究企业本身的工作特点，通过虚心请教和深入研究，使我们对于企业本身的业务有深刻的理解，最后形成针对这个企业的实事求是的解决方案。

要解决用户的什么问题，表达的是我们必须把企业存在的问题提取出来，分析研究哪些问题是可以用信息化技术来解决的，采用信息化技术以后，企业的业务流程需要做什么样的更改，以及这些更改会带给企业什么样的正面和负面的影响。仅仅用计算机来复制企业现有业务流程是不可取的，关键是要做到因为信息系统的使用，使企业业务方式发生革命性的变化，使信息系统成为企业业务不可分割的一部分，而不仅仅是辅助工具。

为什么要解决这些用户问题，表达的是如何帮助企业产生可度量的价值，而这些价值是在研究企业目前存在的问题的基础上产生的，没有这些价值的产生，软件系统的投资是没有意义的。价值不可度量，企业领导者是不可能积极的支持信息化的。还要注意我们的设计必须便于用户使用，减少维护和培训的资源消耗，而且制作生产工艺尽可能简单，这就是设计之本。

二、错误设计的几个原因

但是我们的设计中，违背这样的原则的情况还是时有发生的，大致来说有这么几个原因。

1）新颖的技术成为设计之本

不少设计人员迷恋于新颖的技术，总是倾向于用刚刚流行的新鲜技术来设计他们的软件，他们总是认为只要用了新的技术，就能够写出最好的软件产品，企业用户也一定会喜欢，一定会给企业带来进步。其实这是个误会，企业购买软件产品，并不是购买它的技术本身，企业购买软件产品更多的是关注于改进它们自己的问题，也就是为了他们自己的需求来购买。这就是说是市场决定了产品的设计，而不是技术决定产品设计，这一点千万不要本末倒置。在信息技术战略规划中，我们应该十分关注产品为企业带来的核心竞争力的提高，并以此为设计目标来进行设计。

事实上美国每年倒闭的高科技公司，90%并不是因为技术落后而倒闭，而是因为没有正确的了解市场，换句话说，我们不能因为个人的兴趣而设计软件。

2）把软件当成自我表达的方式

由于软件工程师属于高智商群体，热衷于发明，热爱技术，这样往往不自觉地把软件设计当成自我表达的方式，用于表达自己的智慧，以及表达自己对于技术的理解。这样的结果往往是聪敏反被聪敏误。原因很简单，市场的规则往往决定了产品的命运，而不是技术本身。

我们应该对企业需求、市场状况以及已经存在的系统作为模型来调查，搞清楚企业对产品的要求到底是什么？产品的设计应该来自于对企业需求的调研，而不是我们自己对新技术的激情，新的技术只有用在合适的地方才有生命力，而不应该是一种无目的的自我表达。

新技术的采用只有在需要它的时候才有意义。

3）把软件设计成万能的

最可怕的是，把软件设计成万能的，几乎能满足一切需要，而忽略了技术上的可行性。

没有进行可行性分析的软件产品，通常会导致软件的失败，而且浪费大量的人力物力。一个技术上不成熟的产品流入市场，必将被市场淘汰。当我们开始以企业信息技术战略规划的思想来设计产品的时候，往往倾向于建立一个庞大的、面面俱到的、能解决企业一切问题的软件体系，这样的体系耗资巨大，时间周期很长，最终很可能无法完成。

合理的思维是在整个战略规划下，从可行性研究出发，构造一系列的基本上是相互独立的子系统，这些子系统之间预先定义稳定的接口，把企业运营规则的改变与子系统的投运有机的结合起来，逐步地螺旋形的发展和完善，最终达到提升整个企业核心竞争力的目的。在这样的模式下，产品的可扩充性成为设计的重要考虑方向。

庞大而万能的产品往往没有结果，典型的例子像日本的第五代计算机、语音翻译、人脸识别等等，听着好听，这些公司都倒闭了也是事实。

4）过分强调功能，而不是使用的方便性

企业信息化的目的时提升自己的核心竞争力，使用信息系统的人并不是也没有必要是计算机专家，因此使用的方便性是一个重要的指标。给用户做一件事情，称为“有用的”；如果一个功能是用户可以方便的使用的，称为“可用的”；这是两个完全不同的概念。如果过分强调“有用的”概念，把算法、系统等等放在思考问题的首位，而忽略了方便性，这样的软件往往并不能被企业用户接受。软件可用性往往和对用户心理研究是紧密相关的，具体落实在界面设计上，在软件工程界往往有一种轻视界面设计的倾向，其实这是错误的。

现在的问题在于，很多设计者往往只注意需求文档甚至文档格式，但不注意挖掘需求过程用户所表达的思想内涵，这也是导致不恰当设计的一个重要原因。

三、利用 ITSP 提升企业竞争力的案例陈述

为了对分析与设计有个直观的了解，整个课程贯穿一个简化的 TB 公司电源设备销售服务信息系统的案例，这个简化的例子可以认为是在信息技术战略规划（ITSP）项目中一个构思，根本的目的是企业利用信息化技术提升自己的销售业绩，信息化技术使企业利用信息系统对自己的业务过程和行为方式作充分改进成为可能，这种可能引发了企业建立信息化体系的强烈需求。通过这个案例的研究，我们还可以发现更多的方法学知识。

项目名称：电源设备销售服务信息系统 **项目单位**：TB 公司电源设备销售部 最后修改日期：****年**月**日	
项目目标	TB 公司电源设备销售部是一个以硬件系统配套设备提供商为基本客户的专业电源设备配套提供商，销售的设备来源一部分是自主生产，另一部分由多家国内外知名品牌的电源设备生产商组成供应链。本项目将开发新的业务过程以及相应的信息系统过程和服务，以支持该公司的产品和分级客户服务的战略。预期最终的系统将提供高度集成的过程和服务，这些过程和服务将跨越多个内部业务部门，直接到达客户。该项目预期将实现以下内容： 1，开发一个基于 Web 的网上销售系统，使销售渠道顺畅。 2，开发一个功能全面应用灵活的内部信息系统，使 TB 公司在高度竞争的市场中，带来显著的竞争优势。 3，与客户（硬件系统配套设备提供商）建立一种伙伴关系，其中包括购买电源方案、安装电源设备方案和定制电源设备配套方案，以产生竞争优势。注意，这个方案可能需要重新设计业务过程，这个替代方案假定客户的特殊需求可以和本公司的供应商（电源设备生产厂商）协作完成，并且允许供应商把这些修改加入到他们未来的产品中去，但是，设备生产商的改进将被合同限制在一定时间之后才能销售给本公司的其它竞争者。
项目概念	这个项目是在信息技术战略规划（ITSP）项目中构思的，战略信息系统规划为应用系统、数据库和网络（包括使用因特网作为战略平台）。因为市场被认为是优先级最高的考虑，所以，一个方便的网上订单处理系统，和一个跨部门的高度集成的客户服务系统被认为是最重要的系统之一，同时也和另一个高优先级的库存和供应链系统有合适的接口。
问题陈述	基于 Web 的网上订单系统主要用于是客户可以方便的自主组合订单。 客户服务系统主要与订单处理系统结合处理客户订单，并且可以按分级处理方式处理客户订阅，多年以来本公司主要使用 Microsoft Excel 作为主要的工具，所以现有的计算机处理只是一种初步的方式。该部门急需有一个方便的网上电源销售系统，另外灵活多变的销售策略是提升销售业绩的主要方法，但它们和现有的企业信息系统并不兼容，难以提升整

	体的效率，为此销售部门在讨论中提出了以下具体问题： 1，经常变化的产品组合导致的不兼容，应急配备的系统产生内部的低效率以及与客户关系复杂化。 2，产品构成的易于变化为建立新的基本客户创造了机会，这些易于变化的特征将会吸引未来的客户，但是目前的系统和工作方式并不支持这种易于变化特性。 3，在扩大业务的过程中由于积极的广告行为和基本客户量的大幅度增加，不久将会超出现有方式的实时事务处理的能力，向客户发货的延迟将会导致现金流动困难。 4，管理层建议实现一种“优惠级基本客户”制度，这种制度不能在现有的工作方式上实现，比如不同的业务人员接待具有“优惠级基本客户”资格的同一个客户的时候，难以使用相同的规则。 5，未付款的订单比两年前增加了 2%，发现主要原因是业务量增大以后，由于未付款客户检查过程繁复而遗漏是主要问题。 6，两年中，无效合同增加了 7%，根据分析主要原因是目前的工作方式对合同的有效性难以确认。 7，来自其它企业的竞争，迫使管理层提出了一个动态调整“优惠级基本客户”策略的方案，但现有的工作方式无法做这种动态的改变。 8，某些应急配备设备客户的订单没有得到及时处理，长时间的延时导致客户拒收或者后期处理耗尽仓库的库存。 9，TB 公司的管理层意识到，电源设备销售目前部分使用了计算机处理营销服务渠道，但由于与公司其它营销模式不匹配，难以提升整体效率，但是网上电源销售的成功坚定了管理层的信心，改变这个现状事实上已经成为该公司未来发展战略的一部分。
项目影响范围	这个交叉功能项目将支持或者影响以下的业务功能或外部团体： 1，营销 2，订阅 3，销售和订单录入（对所有的销售办事处，工作方式应该是相同的） 4，仓储（对不同地域的仓储中心可以实现统一调度） 5，库存控制和采购 6，发货和验收 7，所有的不同级别的客户服务 8，外部团体 a,潜在客户 b,客户 c,曾经的客户 d,供应商 e.生产厂 项目范围在项目执行过程中可能会发生变化，但第一阶段尽可能的清楚界定。
项目构想	信息技术战略规划要求该系统必须做到： 1，通过改进数据收集技术、方法、渠道和决策支持加速订阅和订单的处理，管理层希望原有的因特网销售系统略加改造就能够加入到本系统中去，成为本系统的一个子系统。 2，到****年底，未付款的订单减少到 2%。 3，到****年底，无效合同减少到 5%。 4，到****年底，在现有人力资源基本不变的情况下，订单处理能力提高 3 倍。 5，系统将可以协助完成动态调整“优惠级基本客户”策略的方案，同时可以检查合同结构，并支持在线的动态合同修改。 6，便于重新考虑引起客户抱怨的底层业务过程、程序和策略。 7，提供改进的订单和应急配备设备客户的跟踪机制。
业务限制	新系统必须符合以下要求： 1，系统的第一个版本必须在 9 个月内完成，后续的升级版本必须每 6 个月发布一次。 2，没有会计部门的同意，不得改变现有系统的任何文件和数据库结构。 3，作为 TB 公司通过 ISO 9000 认证这个战略目标的一部分，所有的业务过程都需要接受业务过程重构，以改进全面质量管理并支持持续改进。 4，系统必须具备很好的可维护、可升级以及安全性。
技术限制	新系统必须符合下面信息技术架构标准： 1，局域网架构为基于服务的三层架构，客户端为 Windows 应用程序，服务器操作系统为 Windows 2003 Server。因特网服务则使用 Internet Information Server（IIS）。 2，该项目要求开发一个或者多个企业数据库，数据库服务器操作系统为 Windows 2003 Server，希望能自动实现数据备份，从各种因素考虑，数据库采用 Sql Server 2005。

	3，该项目开发的应用系统采用 Microsoft.net 2005，首推的语言为 C#，在某些特殊的地方也可能使用 VC++编写的结构块，但是，并不拒绝在必要的情况下使用 Java 语言。 4，外部客户所使用的环境一律使用浏览器，但并不限制操作系统，对不同级别的客户，权力要实现限制。 5，内部员工一律使用 Windows XP 或升级版本，除了浏览器外，还可能使用桌面应用程序，需要预装 Microsoft.net Framework 2.0（或以上版本）。
项目组织方式	为此项目成立专门的开发管理班子，职责为： 1，任命项目经理 2，任命首席构架师和分析师 3，任命项目经理推荐的项目团队 4，评审并批准项目发布的内容 5，确保项目符合管理层的想法 6，认可所有的范围、预算和计划变更 要求项目团队每周召开情况工作会议，由项目经理组织，并把细节报告给管理班子。 开发团队需要建立自己的开发网站，其主要的内容为软件架构文档（Software Architecture Document SAD），用以公布项目决策的概要和架构的视图，便于开发成员了解项目的进程和主要思想。

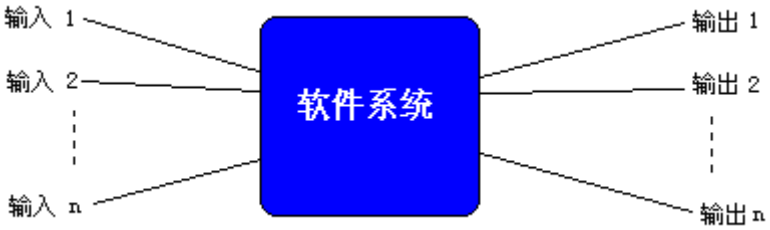
第二章 结构化分析的理论与实践

2.1 需求过程在软件开发中的重要作用

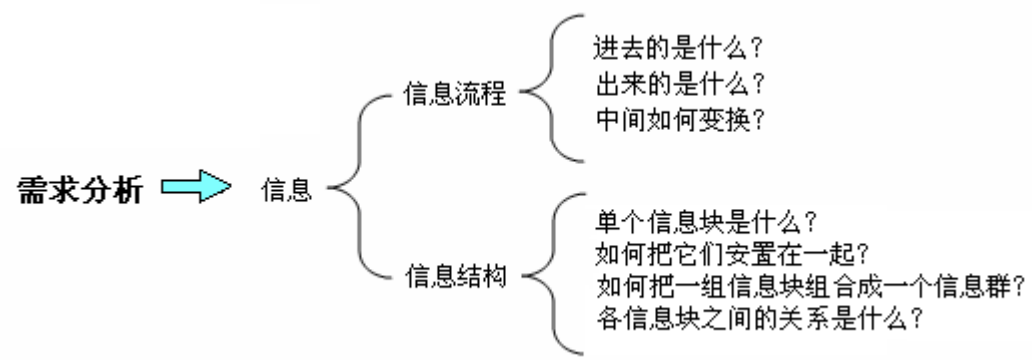
任何设计的输入来自于需求分析，什么样的需求思想，就有什么样的设计思维。这就是说，合理而且正确的需求分析过程，是系统设计过程的一个有机的组成部分，所以，我们首先必须讨论需求分析的领域建模的有关问题。

需求开发与需求管理是相辅相成的两类活动，它们共同构成完整的需求工程。

所谓需求分析，是把软件系统的全部功能被表示成一个单一的信息变换过程：



需求分析也是一个分解的过程。



软件需求分析人员应该具备的特征：

- 善于领会一些抽象的概念，重新整理使之成为各种逻辑成分，并根据各种逻辑成分综合出问题的解决办法；
- 善于从各种相互冲突或混淆的原始资料中吸取恰当的论据；
- 能够理解用户的环境及领域知识；
- 具备把系统的硬件和软件部分应用于用户环境的能力；
- 具备良好的书面和口头形式进行讨论和交换意见的能力；
- 具有“既能看到树木，又能看到森林”的能力。

需求的类型：

作为一个检查表，需求可以按照 FURPS+模型进行分类的，每个字母含义如下：

F：功能性（Functional）：特性、能力、安全性。

U：可用性（Usability）：人性化因素，帮助，文档。

R：可靠性（Reliability）：故障周期，可恢复性，可预测性。

P：性能（Performance）：响应时间，吞吐量，准确性，有效性，资源利用率。

S：可支持性（Supportability）：适应性，可维护性，国际化，可配置性。

+：辅助和次要的因素，比如：

实现（Implentation）：资源限制，语言和工具，硬件等。

接口（Interface）：与外部系统接口所加的约束。

操作（Operations）：系统操作环境中的管理。

包装（Packaging）：

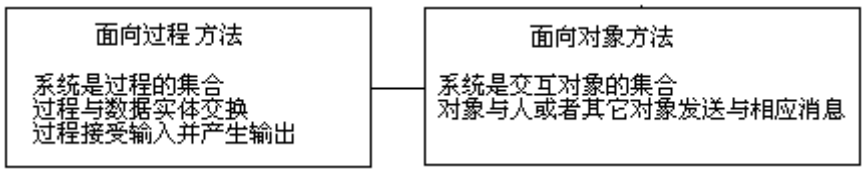
授权（Legal）：许可证或其它方式。

事实上，FURPS+模型并不是唯一的，但用它来作为需求范围检查表是很有效的，这可以降低考虑系统的时候遗漏某些因素的风险。还有一些分类方式和 FURPS+模型类似，比如 ISO9126，它主要来自于美国软件工程研究所(SEI)。

2.2 面向过程的需求分析

传统的面向过程需求分析与面向对象分析是不同的，传统方法把系统看成一个过程的集合体，由人和机器共同完成一个任务，计算机与数据交互、读出数据、进行处理又把结果写回到计算机里面去。

在讨论事件的时候，过程方法强调组件的过程模型。而对象方法把系统看成一个相互影响的对象集，对象重要的是具有行为（方法），行为发送消息请求另一个对象做事情，就本质而言，对象方法不包括计算机过程和数据文件，而是对象执行活动并记录下数据，当为系统响应建模的时候，对象方法包括响应模型、模型行为以及对象的交互。两种方式的不同点如下图：



下面我们先简单讨论一下面向过程分析的特点，一般来说，面向过程的分析必将导致面向过程的架构（设计）。面向过程的方法（OP）又称之为结构化方法（SA），它的特征如下：

1）采用“抽象”和“分解”两个基本手段，用抽象模型的概念，按照软件内部数据传递、变换关系，由顶向下逐层分解，直到找到满足功能需要的所有可实现的软件元素为止。

2）采用“分解”的方式来理解一个复杂系统，“分解”需要有描述手段，数据流程图就是作为描述信息流程和分解的手段而引入的。

一、数据流图 DFD

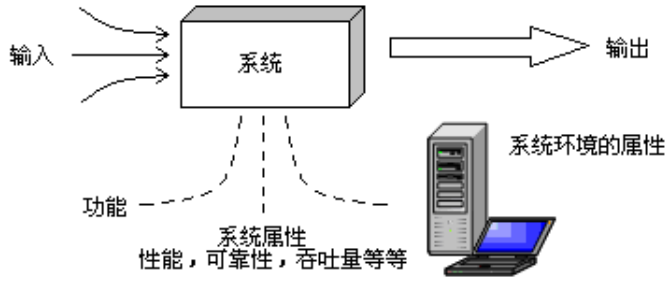
1，结构化分析以数据流为核心

我们探讨结构化分析为什么以数据流为核心。就需要考虑软件系统本质是什么，任何软件系统都是使用“输入 – 处理 – 输出”这样一个基本模型，这些模式比较适用于描述商业软件，它们中大多数依靠数据库。为什么结构化设计以数据流为核心进行分解呢？我们应该看到，任何软件系统都可以用五类事情来完全描述系统行为：

- **系统的输入**：不仅仅是式输入内容，还有输入设备、形式、外观和感觉等必要的细节。很多开发人员都发现了，在这个领域可能包括大量的细节，并且具有易变性，尤其是 GUI、多媒体或互联网的。
- **系统的输出**：对输出的描述，如语音输出或可视化显示等必需的支持，以及系统所产生信息的协议和格式。

- **系统的功能**：把输入映射到输出，以及它们不同的组合。
- **系统的属性**：典型的非功能需求，如可靠性、可维护性、可得到性以及吞吐量等开发人员必须考虑的问题。
- **系统的环境属性**：附加的非功能性需求，如系统在不同的操作系统、负载下的操作能力等等。

这种划分方法如下图所示。



这个图可以把需求分析需要研究的内容给描述清楚了，数据的处理和变换是软件系统的基本要素，研究数据的流动关系，可以使我们的分析更清楚的表达系统本质。

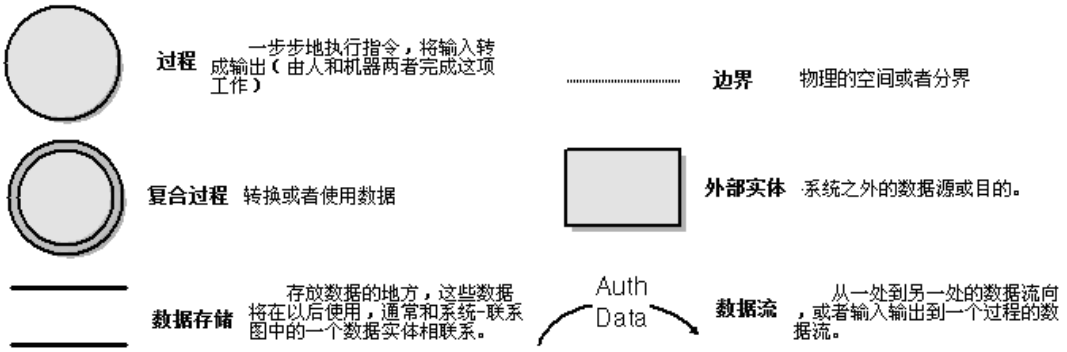
2，DFD 的符号

结构化分析的主要模型是数据流图（DFD），也称之为面向过程的方法。面向过程的方法在业务过程分析等方面具有独特的优势。由于分析与设计的第一步就是建议业务架构概念，所以深入理解数据流图是很有必要的，它的特点如下：

- 可以表示任何一个系统（人工的、自动的、或混合的）中的数据流；
- 每个可以加工的过程可能需要进一步分解以求得对问题的全面理解；
- 着重强调的是数据流而不是控制流。

DFD 只用了 6 个符号来表达概念：

- **外部实体**：在系统边界之外的个人和组织，它提供数据，或者接受数据输出。
- **过程**：在 DFD 中的一个符号，它代表数据输入转换到数据输出的算法或者程序。
- **复合过程**：代表过程的一种抽象，内部包含的多个子过程，例如把整个系统看成一个过程。
- **数据流**：在 DFD 中的箭头，它表示在过程、数据存储和外部实体间的数据移动。
- **数据存储**：保存数据的地方，将来一个或者多个过程来访问这些数据。
- **边界**：代表物理的空间或者分界。



二、产品范围的确定

在问题涉众达成一致意见以后，就需要集中精力在解决方案的边界上，以便解决一经发现的

问题。所谓解决方案的边界指的是解决方案与包围解决方案的现实世界之间的边界，我们必须明确，哪些是解决方案的一部分，哪些是通过系统与外界的接口进行的。这可以大致上圈定我们的工作范围。在初步的范围定义阶段我们需要做的事情是：

1，列出问题和机会：

需要关注以下几个方面的问题：紧急程度（什么时间实现？），可见性（系统在多大程度上对客户或执行管理层是可见的？），收益（新方案会增加或者减少多少支出？），优先权（那些问题是最重要的？如果预算出了问题，需要减掉哪些问题？），可能方案（用简单的方式表述：如，不改变令人满意的事物、快速修复、对现有系统改进、重新设计现有系统、设计一个新系统等等）

2，协商项目的初步范围：

包括什么类型数据描述了正被研究的系统？正被研究的系统包括什么业务过程？系统需要如何与其它用户、地点或者其它系统接口？

注意，项目的范围陈述可以描述成一个简单的列表，但不需要定义列表中的项目，也不十分关心精确的需求分析，尤其不关心任何费时的建模或者原型化。

在很多情况下系统的边界都很明显，但是有些情况下却不是那么清晰。比如在我们案例中说到的订单输入的例子，这个系统要集成在一个更大的系统中，分析人员必须确定它是否要和其它系统共享数据，新的系统是不是要在不同的主机或客户之间分布的，等等。

在问题分析过程中确定参与者是主要的分析步骤，这看起来很明显，但是我们如何才能找到参与者呢？下面的问题可以有所帮助：

- 谁会对系统提供信息？谁会在系统中使用信息？谁会从系统中删除信息？
- 谁是系统的操作者？
- 系统将会在哪里被使用？
- 系统从那里得到信息？
- 哪些外部系统要和系统进行交互？

3，利用内/外列表

事实上经验可以告诉我们，要判断一个主题是属于项目内还是项目外并不容易，而且常常会有两种截然相反的观点。这种决策有时还会花去很长的时间。我推荐大家在项目启动会议上利用一下所谓“内/外列表”，这种表共三列，第一列可以是任何内容，每当对某个主题是内还是外有疑问的时候，可以把它加入表格，并询问这属于项目“内”还是“外”？

在后期需求分析中，当队内外有疑问的时候，也参考这张表，并随着工作的进度对表格进行修改。下面是为“购买请求跟踪系统”建立的内外列表。

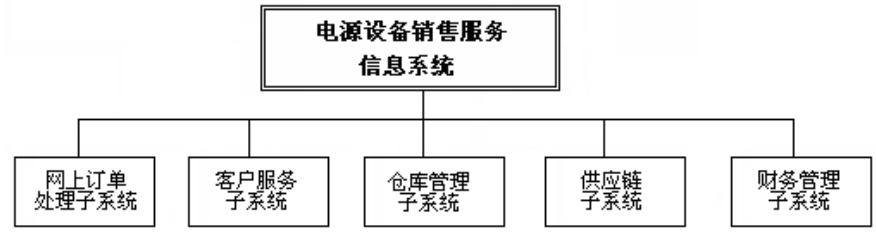
“内/外”列表		
主题	**内**	**外**
以任意形式开发票		√
产生请求报告（请求可能由买主、分部或某个人发起）	√	
把请求合并成一个队列	√	
部分发货、延迟发货、错误发货	√	
所有新的系统服务、软件	√	
系统中任何非软件部分		√
识别任何可用的已存在软件	√	
申请	√	

三、上下文图（Context Diagram）

上下文关系图表达的是未来系统的整体图像，我们来看一下将采取什么步骤完成这个视图更合理。

1，子系统组成

对于前面讨论过的电源销售系统，我们可以写出子系统的构成如下图所示。



下面列出了各个子系统说明。

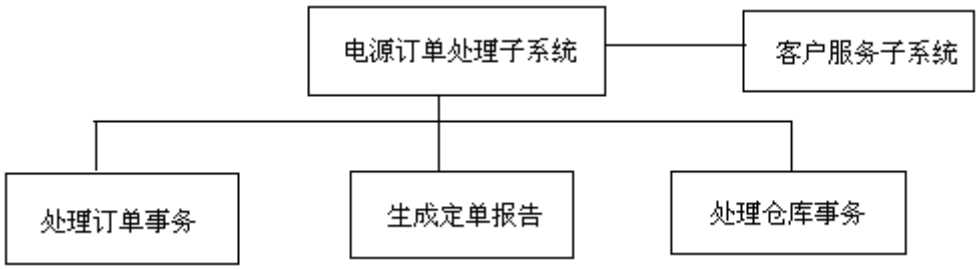
电源设备销售服务信息系统各子系统功能说明			
编号	**子系统**	**功能说明**	**要求**
1	订单处理子系统	1，客户直接利用因特网构建电源设备购买订单。 2，客户可以选择标准配置，也可以在线建立自己的配置。 3，对每一种配置，系统通过客户服务系统提供客户相关信息计算销售价格。 4，订单通过仓库服务系统获取发货策略。	订单生成使用 Web 页面。 内部服务使用桌面程序。
2	客户服务子系统	1， 实现客户分级管理策略。 2，处理客户订阅。 3，处理客户资料。 4，处理订单和客户资料。 5，查询客户购买历史。 6，处理促销方案。 7，每日生成默认合同报告。 8，向订单处理子系统发送价格处理结果。	客户订阅和查询使用 Web 页面。 内部处理使用桌面程序。
3	仓库管理子系统	1，处理库存。 2，处理进货。 3，决定发货策略。 4，由供应链子系统获取供货信息。 5，打印相应的报表。	使用桌面程序。

对于每个子系统，我们都需要进行详细的陈述，例如，订单处理子系统项目陈述如下：

项目名称：电源设备订单处例子系统 **项目单位**：TB 公司电源设备销售部 最后修改日期：2009 年 08 月 20 日	
系统目标	1，客户直接利用因特网购买电源设备，客户选择设备，设备分为普通不间断电源、服务器专用不间断电源和专业级不间断电源加自主供电设备等。 2，客户可以选择标准配置，也可以在线建立自己的配置。 3，可配置的构件显示在一个下拉列表中，对每一种配置，系统可以计算价格。
系统要求	1，发出订单时，客户需要填上运送和付款信息，系统可接受的付款方式为信用卡和支票，一旦订单输入，系统将向客户发送一个确认 e-mail 信息，并且附上订单细节，在等待电源设备送到的时候，客户可以在任何时候在线查到订单状态。 2，后端订单处理包括下面所需的步骤：由客户服务系统提供这个客户的等级以及根据等级和促销策略计算出的相应折扣方式，验证客户的信任度和付款方式，向仓库请求所订购的配置，打印发票，并且请求仓库将电源设备运送给客户。

2，功能分解图

我们希望把系统的目标和要求分解成一系列比较的大的功能模块，这就是功能分解图，功能分解显示了一个系统功能自顶向下的分层结构，也是我们绘制数据流图（DFD）的提纲。

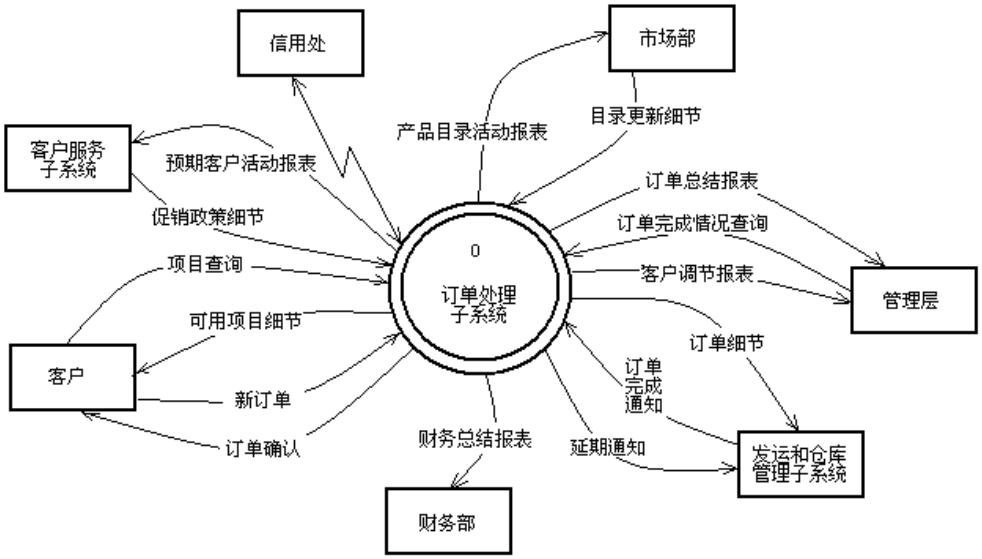


3，上下文关系图

对于每个子系统，我们需要集中精力绘制系统的上下文图，实际上这个图是一个系统宏观上的透视，或者是一个体系上的轮廓。能够表达清楚在与相邻系统相互作用的最重要的信息。图中并不需要事无巨细的绘制相邻系统细节，而是表达清楚什么是最重要也是人们最关心的信息。

系统内部在单个过程符号中概括所有处理活动的 DFD。下面是客户支持系统的上下文图简单例子，注意，箭头表示数据的流向。在初期的业务分析中，用数据流图表达上下文关系是有一定优势的。

在建立上下文关系图的时候，系统内部在单个过程符号中概括所有处理活动，而用有向线段表示数据的流向。这种表达可以使系统内外关系变得很清晰。



建立上下文图是系统分析的基础，它的作用在于：

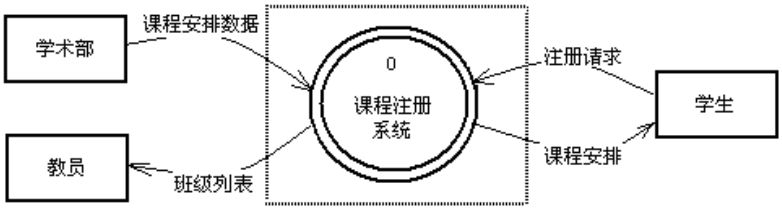
- 发现产品的外部事件；
- 发现产品的边界；
- 发现的产品的相邻系统

这对于由外及内、由上而下的分析思路是非常有帮助的。上下文模型最大的用途是用来刻划软件边界。

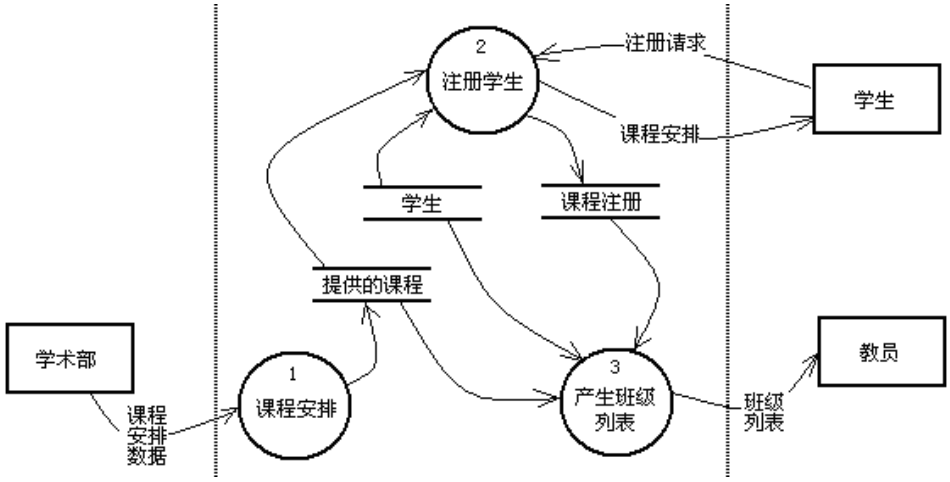
四、逐次分解的系统模型

1，基于过程划分的系统过程模型

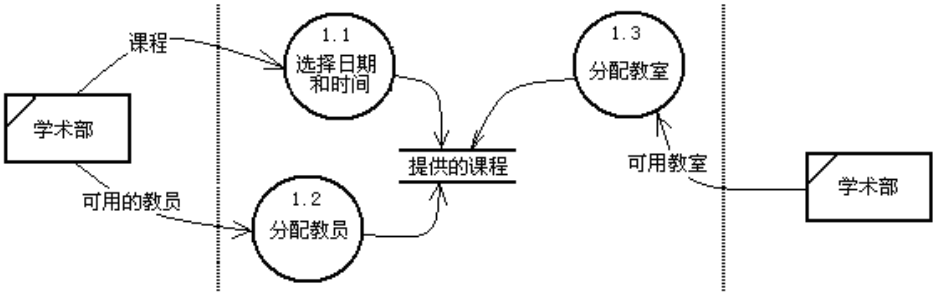
如果一个 DFD 片断包括更多的处理，可以把把每个过程逐步按层分解，以便作更详细的研究，这称之为过程分解。对上下文关系图进行的分解，称之为 1 级数据流图，例如，一个大学课程注册系统，其上下文关系（0 层图）如下：



在过程分解中，其 1 级数据流图可以作如下分解。



如果需要，可以再对过程“1”（课程安排）进行进一步分解，如下图所示。



这种分解的过程，将使我们对整个系统的业务模型由粗而精理解得非常清楚。

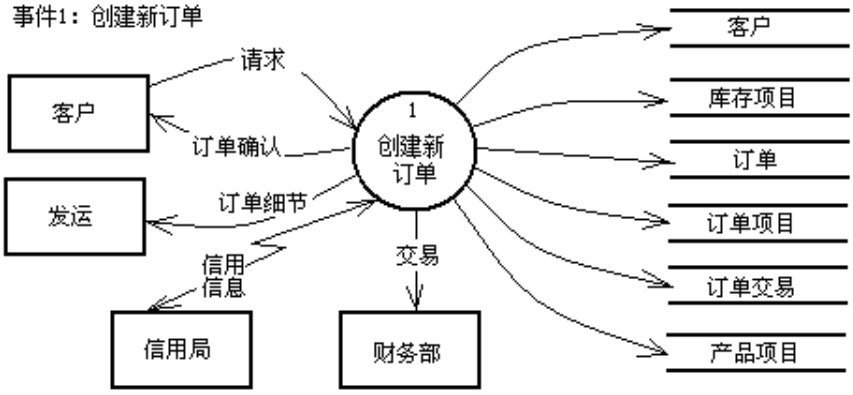
2，基于事件划分的系统过程模型

业务分析也可以使用事件驱动，现代过程分析也往往以事件为基础进行划分，其中每个事件包含一套处理过程。在研究事件图的时候，往往需要了解所有的数据存储。必要的时候，数据库分析和设计可以提到前面先来完成。要说明的是，在分析阶段并不需要数据库的详细设计，而只是把数据存储用实体的方式从大的方面规范清楚，以此作为详细设计的一个必要的输入。

大多数事件图包括一个单一过程，并且需要说明以下内容：

- 输入及输入来源，来源被描述为外部代理。
- 输出及输出目的地，目的地被描述为外部代理。
- 必须读取记录的任何数据存储都应该被加入到事件图中，事件流应该加入命名。
- 对数据的任何增、删、改、查都应该加入到事件流中，事件流应该加入命名。

事件图的敏感性和简单性，使它成为专家和用户沟通的强有力的工具，针对我们已经讨论过的“订单处理子系统”，可以对单个独立事件画出相应的数据流图。



3，事件分解的综合应用案例

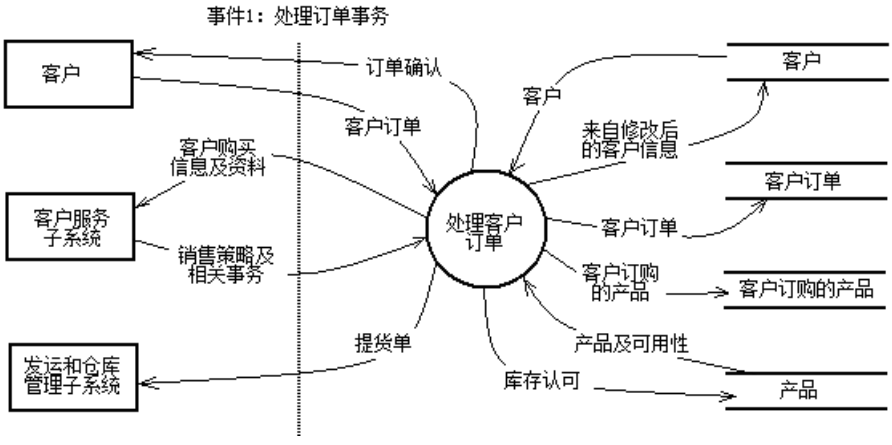
具体到设计过程，我们到底采用过程分解还是采用事件分解要根据情况。不过一般过程分解是静态的、先验的，比较适合预定义过程。而事件分解是动态的、更适合描述软件系统工作的，所以适合在软件系统分析中使用。事件的取得可以对输入数据流进行分析，根据它是否触发业务流程来判断是否是一个事件。很多情况下这两种分解方法是综合应用的，事件分解有助于表达相互隔离的事务，而某些情况下利用过程分解有助于进一步细化事务流程，目的只有一个，那就是把业务分析清楚。下面我们以上面讨论过的“订单处理子系统”，看综合应用事件分解的例子。

1）过程事件表与数据流图

现在我们的眼睛盯住具体的细节，为每个事件引发的事务绘制一个事件图，我们就可以建立一个事件的上下文流图。上述“订单处理子系统”的过程事件表如下。

参与者	事件	触发器	响应
客户	选择产品	产品查询	生成“目录描述”
客户	发出订单	新客户订单	生成“客户订单确认”，在数据库中创建“客户订单”和“客户订购的产品”。
客户	修改订单	客户订单修改请求	生成“客户订单确认”，修改数据库中 “客户订单”和“客户订购的产品”。
客户	取消订单	客户订单取消	生成“客户订单确认”，在数据库中逻辑的删除 “客户订单”和“客户订购的产品”。

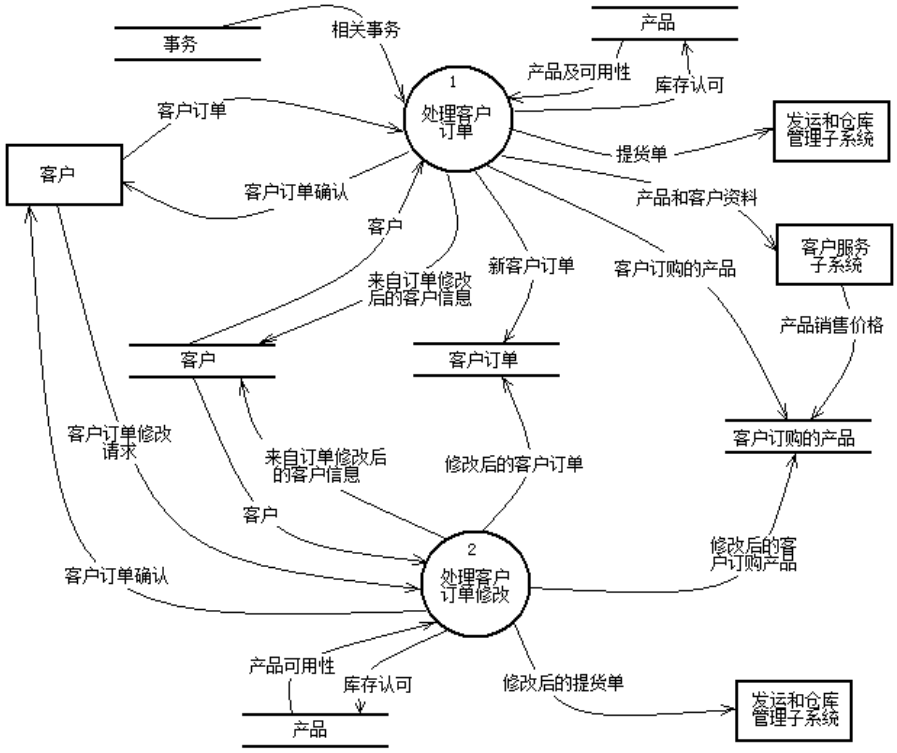
据此可以画出相应的过程事件图，每一个事件应该是一个单一的过程。



2）建立基于系统的业务模型

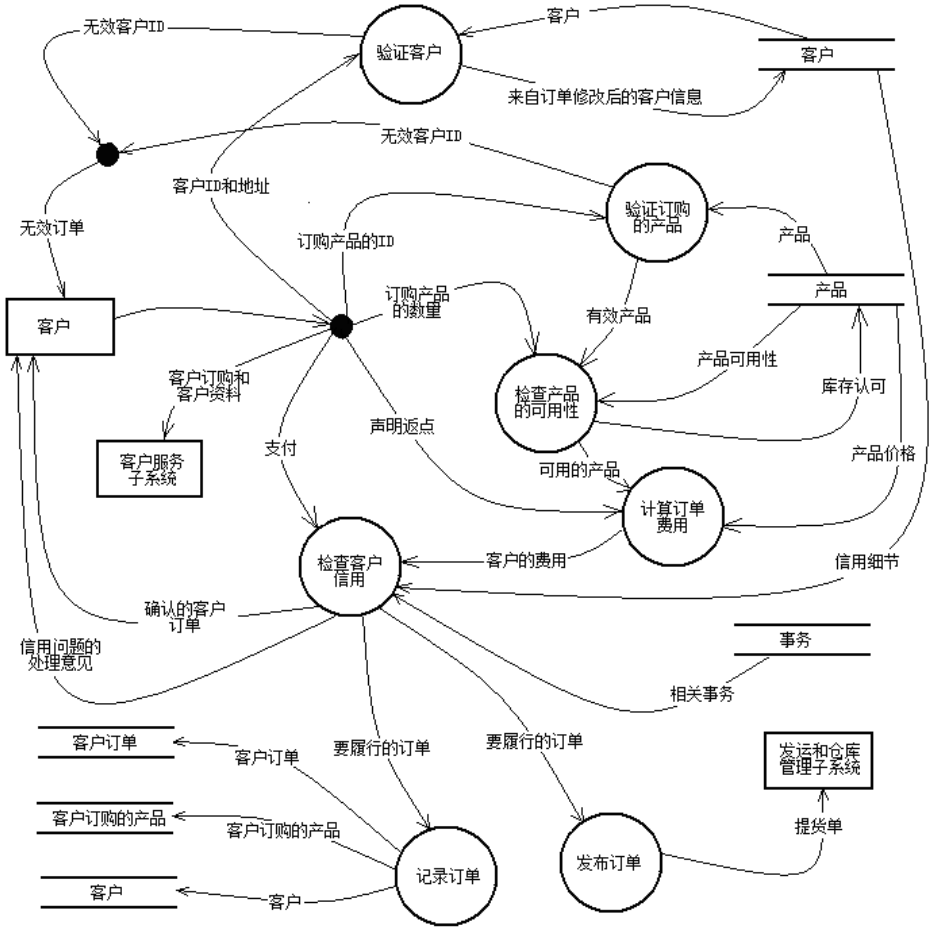
事件图并不是孤立存在的，它们集合在一起定义了系统和子系统。我们可以把单个过程进一步细化，构造一个或者多个系统或者子系统中所有事件的相互关系。这称之为建立基于系统的业务模型。从本质上说，它是对业务进行形式化分解，自顶向下、逐层细化。

从某种意义上说，在绘制系统图的时候必须平衡不同详细程度的事件图，以保证一致性和完整性，必要的时候可以扩展为多个 DFD。系统图更多的是从宏观的角度看为题，更多的考虑相互关系，这点很重要。



3）基本图

系统图中的某些重要的事件过程可以扩展为一个基本的数据流图，以揭示更多的细节，这对比较复杂的业务过程（比如订单处理特别重要），有些事件比较简单（比如报告生成），所以不需要进一步扩展。



DFD 的细节称作片段，片段的组合有多种方式，现代过程分析也是以事件为基础，所以完全集可以组合到一个事件划分系统模型或者称为 0 层图中去。其中，每个过程为一个事件的处理。

事件解首先需要定义来自系统外部的事件，在分解的过程中再发现由系统内部状态条件或时钟触发的内部事件。

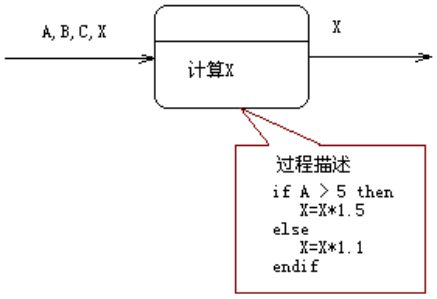
五、评估 DFD 的质量

高质量的 DFD 是可读的、内部一致的以及能准确表示系统需求的。

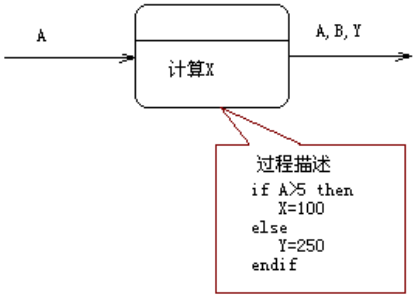
- **复杂性最小化：**人们对复杂信息处理是有局限性的，当太多的信息同时出现的时候，人们把这种现象称作**信息超量**。在这样的情况下，可以把信息划分为小的相对独立的子集，这样便于单独考察和理解信息，这也是建模最根本的目的。
- **7±2 原则：**这个原则也称之为 Mille 数，由心理学研究，一个人可同时记住或操纵的信息块的数目大约在 7 到 9 之间，也就是一个模型的过程数不要超过 7±2 个。另外数据流进、流出一个过程、数据存储或者数据元素的个数不要超过 7±2 个。这不是强制原则，但可以给我们提供一个警告。
- **接口最小化：**这里的接口是指一个问题或者描述中的一部分与其它部分的连接。源于 7±2 原则，连接数应该保持最小，如果超出了这个原则，可把一个过程分解为更多的过程，以使得分析简单。
- **数据流一致性：**通过分析数据流的不一致，可以找到错误。

下面的例子使用了过程描述（面向过程英语）来描述内部过程，流入的 B、C 没有任何处理，

也没有流出，被称之为“**黑洞**”。



下面的例子，流出的 B、Y 和内部的 X 并没有流入，被称之为“**奇迹**”。



推导数据流图的简单准则：

1）第一层数据流程图应当是基本的系统模型；

2）应当仔细说明原始的输入/输出文件；

3）所有箭头和过程均应当加上标注（使用有意义的名字）；

4）必须保持信息的连续性；

5）原则上每次只加工一个过程。

数据字典：

1）在数据流图中，所有的图形元素都进行了命名，所有名字的定义集中起来就构成一本数据字典。

2）数据字典最重要的用途是作为分析阶段的工具。在数据字典中建立的一组严密一致的定义，有助于改进分析员和用户之间的通信，因此将消除许多可能的误解。对数据的这一系列严密一致的定义也有助于改进在不同的开发人员之间或者不同开发小组之间的通信。如果要求所有开发人员都根据公共的数据字典描述数据或设计模块，则能避免许多麻烦的接口问题。

六、分层结构图

在初步分析的时候，对总体的需求结构作一个分解是很必要的，这种分解可以使各个问题之间的关系比较清楚。

1，信息结构

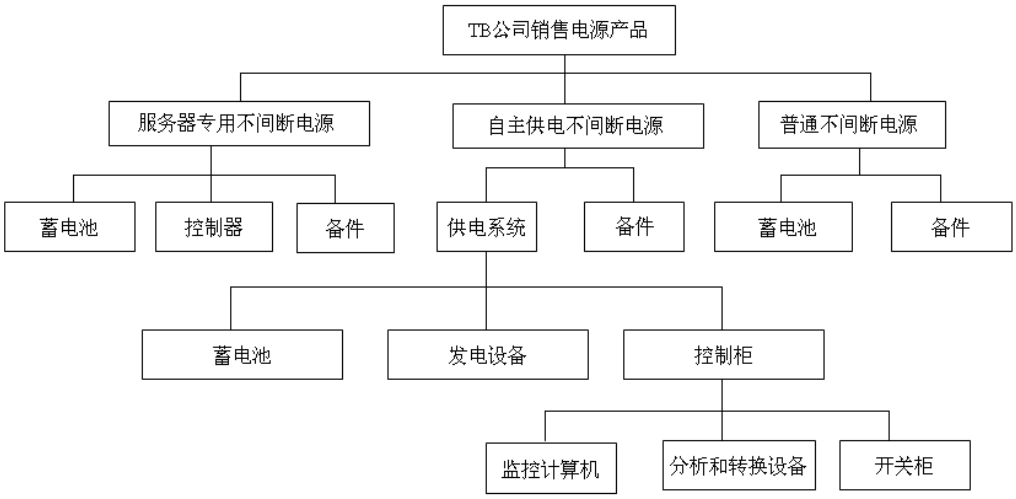
1）信息结构是各个数据成分之间逻辑关系的一种表示方法。

2）数据结构决定信息的组织、存取方法、结合性程度以及不同的处理方案。

3）典型的数据结构包括标量项、顺序向量、n 维空间、链接表等。

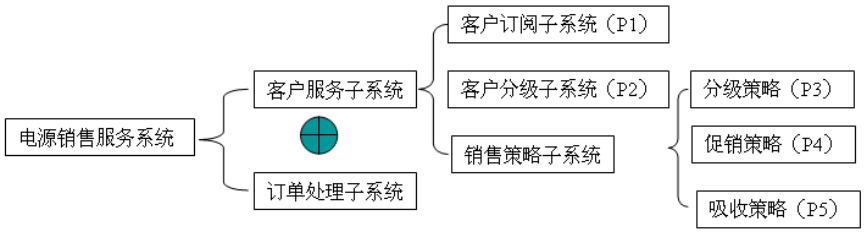
2，分层框图：

分层框图把信息用多层方框按照树形结构组织起来。在结构的顶层，用一个方框代表整个结构。下面各层由表示不同信息类别的方框组成，它们可以看成是上一层方框的子集。在该图的最低一层，每个框包含单独的数据实体。



七、Warnier 图：

Warnier 图把信息表示成一种树形数据结构。可以规定某些信息种类或信息量是重复性的，也可以说明在某一种类中信息是有条件出现的。



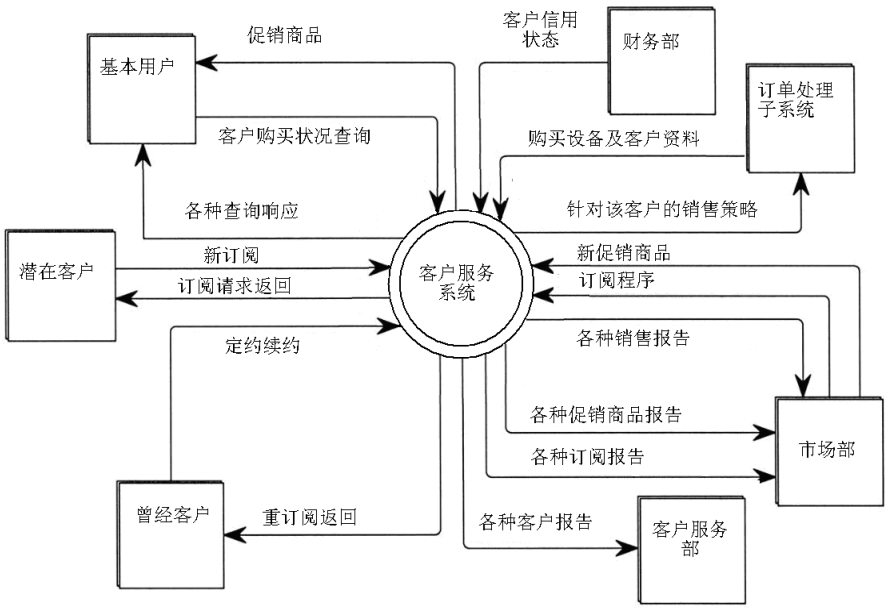
2.3 结构化分析案例

一、客户服务子系统的结构化分析

从方法学的角度，尽管很多情况下我们提倡面向对象的分析，但对于系统功能来说，面向过程的分析有时还是有其优点的，因为它直接对应于工作流程和系统的结构，事实上现代分析中，通过引入用例和事件的概念，使这种改进的过程分析仍然具有生命力，下面对于我们所研究的课题，采用过程来细化分析。

1，上下文数据流图

这个问题已经在前面讨论过，这里就直接画出这个系统的上下文来。



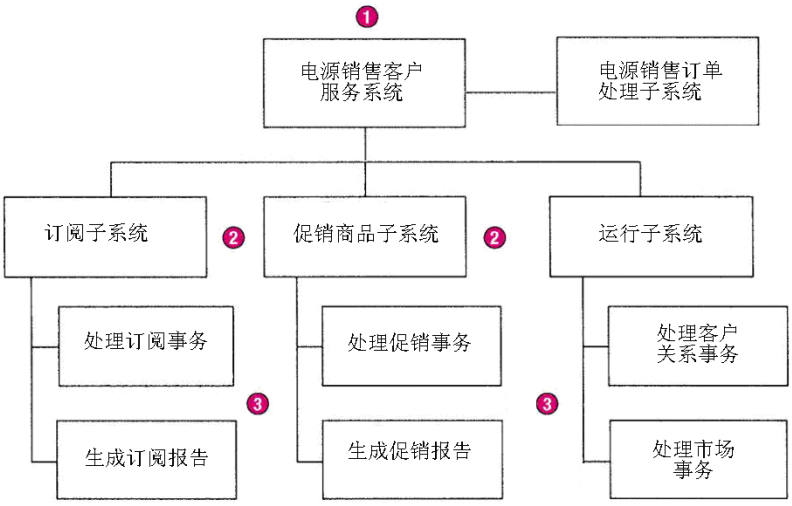
2，功能分解图

功能分解显示了一个系统自顶向下的分解结构，也为我们绘制数据流图（DFD）的提纲，下面是图中的数字含义：

1）整个系统。

2）系统最初的子系统/功能块。并不要求和实际的组织系统对应，分析员和用户越来越多的被要求忽略组织边界，构造并且理顺过程和数据共享的交叉功能系统。

3）区分系统的运行部分和报告部分。



3，事件响应或用例清单

我们可以借用面向对象的用例分析来进行这一步的研究，主要是确定系统必须响应什么业务事件，从本质上说，系统存在三类事件：

1）外部事件：该事件发生时，产生一个到系统的事件流。

2）时序事件：以时间为基础的处罚过程。

3）状态事件：系统由一个状态转换为另一个状态时触发的事件。

当使用用例分析的参与者的时候，引发事件的参与者，将成为 DFD 中的外部代理，而事件将由 DFD 中的某个过程来处理。下面，我们部分的列出在这里比较合用的用例清单。

参与者	事件（或者用例）	触发器	响应
市场部	制定一个新的客户关系订阅计划，把潜在客户变为基本客户。	新客户订阅程序	生成“订阅计划确认”，在数据库中创建合同。
市场部	制定一个新的客户关系订阅计划，以把曾经客户重新变为基本客户。	曾经客户订阅计划	生成“订阅计划确认”，在数据库中创建合同。
市场部	为当前客户改变订阅计划（例如延长优惠时间）	订阅计划改变	生成“合同修改确认”，修改数据库中合同。
（时间）	一个订阅计划过期。	（当前日期）	生成“合同修改确认”，在数据库中逻辑的删除（置空）合同。
市场部	在达到计划的过期日期之前取消一个订阅计划。	订阅计划取消	生成“合同修改确认”，在数据库中逻辑的删除（置空）合同。
客户	潜在客户通过订阅加入本系统，这个潜在客户至少答应在两年内购买一定数量的本公司设备。	新订阅	生成“合同目录修改确认”，在数据库中创建“客户”，在数据库中创建第一个“客户订单”和“客户订购的产品”。
客户	修改地址（包括电子邮件和密码）	地址修改	生成“客户目录修改确认”，修改数据库中的“客户”。
财务部	修改客户的信用状态。	信用状态修改	生成“信用目录修改确认”，修改数据库中的“客户”。
订单处理系统	输入订单和客户资料，获取针对具体客户的销售策略	客户确认产品	生成“客户订购产品优惠价格”发送给订单处理系统。
（时间）	市场部决定停止销售一个商品后 90 天。	（当前日期）	生成“目录修改确认”，在数据库中逻辑的删除（失效）“产品”。
（时间）	订单处理后 90 天	（当前日期）	在数据库中实际的删除 “客户订单”和“客户订购的产品”。
客户	查询自己购买历史记录（3 年为限）	客户购买查询	生成“客户购买历史”。
客户服务部	生成月末报告	（当前日期）	生成“月度销售分析报告” 生成“月度客户合同例外情况分析报告” 生成“客户关系分析报告”

认真把这个通过这个表思考清楚：

1）引发事件的参与者，它们将成为 DFD 的外部代理。

2）事件，它们由 DFD 的某个过程处理。

3）输入或者触发器，它们将成为 DFD 中的一个数据流或者控制流。

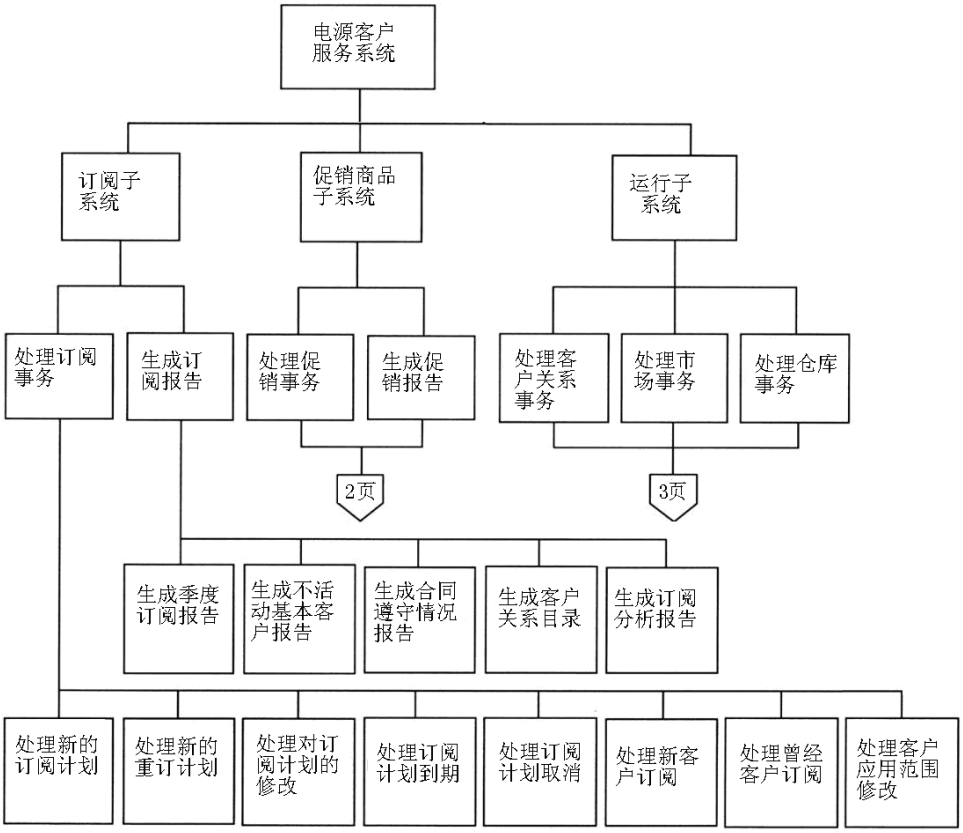
4）所有的输出响应，它们也将成为 DFD 中的数据流，注意我们是用括号来表达时序事件。

5）输出，注意这里不要暗示实现方式，当我们使用**报告**这个词的时候，并不一定指的是书面报告。输出包括了修改数据模型中存储的实体模型，比如创建新的实体实例、修改实体的现有实例以及删除实体实例等。

一个系统的用例可能很多，对于设计者来说详细的列出是非常有必要的，仔细研究这些用例，然后给每一个事件分配一个子系统功能，可以绘制出事件分解图。

4，事件分解图

为了进一步分解功能，我们把每个用例的事件处理过程分解到图中，这个图可以认为是前面功能分解图的一个细化，如果太复杂，必要的时候可以多页来绘制。这个图可以作为分析和设计的一个提纲。



5，事件图和系统图

现在我们的眼睛盯住具体的细节，为每个事件过程绘制一个事件图，它们集合在一起定义了系统和子系统，系统图更多的是从宏观的角度看为题，更多的考虑相互关系，具体的绘图学员可以自己来完成。

二、需求规格说明

上下文图、系统图、事件图和基本图的组合构成了过程模型，一个工艺良好的完整过程模型可以在最终用户和计算机软件设计者以及程序人员之间有效的沟通需求，消除大部分系统设计、编程和实施阶段出现的混淆。注意，完整的过程模型并不仅仅是这些图，更多的是文字说明，把图形和文字结合起来，设计就会非常的清晰而且避免歧义，这非常重要。一个完整的需求规格说明模版如下。

项目驱动：描述项目的理由与动机。

1，项目的目标：投资创建产品的理由以及希望取得的业务上的好处。

2，客户、顾客和其它风险承担者：产品涉及他们的利益以及对他们的影响。

3，产品的用户：预期的最终用户，以及他们对产品可用性的影响。

项目限制条件：加在项目和产品上的约束条件。

4，需求限制条件：项目的局限性和产品设计的约束条件。

5，命名标准和定义：项目的词汇表。

6，相关事实和假定：对产品产生一定影响的外部因素。

功能性需求：产品的功能。

7，工作的范围：针对的业务领域。

8，产品的范围：定义预期产生的边界。

9，功能与数据需求：产品必需作的事情以及所操作的数据。

非功能性需求：产品的品质。

10，观感需求：预期的外观。

11，易用性和人性化需求：产品让预期用户使用应该是什么样子的。

12，执行*需求：速度、大小、精度、人身安全、可靠性、健壮性、可伸缩性、持久性和容量等需求。

13，操作和环境需求：产品预期的操作环境。

14，可维护性和支持需求：产品的可改动性必须达到什么水平，以及需要什么样的支持。

15，安全性需求：产品的信息安全、保密性和完整性。

16，文化与政策需求：人和社会因素。

17，法律需求：满足适用的法律。

项目问题：这些是适用于构建产品的项目。

18，开放式问题：那些尚未解决的问题，可能对项目的成功有影响。

19，立即可用的解决方案：利用已有的组件，而不是重新开发。

20，新问题：引入新产品而带来的问题。

21，任务：把产品投入使用必须要做的事情。

22，迁移：从现有系统转换的任务。

23，风险：项目最有可能面对的风险。

24，费用：早期对构建产品的成本或者工作量的估计。

25，用户文档：创建用户指南或者文档的计划。

26，后续版本需求：可能在产品将来发行版本中包括的需求。

27，解决方案的想法：我们不想错失的设计想法。

在需求的规格描述中要避免二义性，避免发生描述上的歧义，例如：系统要有一定的安全保密措施。这种描述既无发定义也无法测试，所以是不合适的。

第三章 面向对象的建模与 UML

统一建模语言（Unified Modeling Language ，UML），是一种对软件密集型系统的制品，进行可视化、详述、构造和文档化的图形语言。

UML 给出了描绘系统蓝图的标准方法，其中既包括概念性的事务，如业务过程和系统功能，也包括具体的事务，用特定语言编写的类、数据库模式和可复用的软件构件。

面向对象的建模语言出现在 20 世纪 70 年代中期到 80 年代后期之间，那时，人们开始研究利用面向对象的思想进行分析和设计。到 1989 至 1994 年间，面向对象的方法增加到 50 种以上，用户能难找到一用完全满足他们要求的建模语言，于是人们开始寻求一些新的方法，其中著名的有如：OOSE（面向对象的软件工程）、OMT（对象建模技术）等。但每种方法都各有优缺点。

大量有决定意义的思想来自于 20 世纪 90 年代中期，到 1994 年 10 月，Rational 公司开始正式研究统一建模语言（UML），到 1996 年 6 月，发布了 UML0.9 版，在 1996 年全年在工程界收集反馈意见，大量的合作伙伴加入了协作，不久发布了 1.0 版。

1997 年 7 月，发布了 UML 1.1，并提交给了 OMG（对象管理组织），到 1997 年 11 月 14 日，被 OMG 接受，成为面向对象的标准。

1998 年发布了 UML 1.2，到 1998 年秋，发布 UML 1.3，它在技术上更趋完善。

UML 的出现对软件工程的影响是巨大的，它不但影响了设计方式，也影响了一些新一代语言的语言结构。

3.1 面向对象的建模

一、建模的重要性

过去软件设计经常遇到这样的事情，由于需求分析做得不够，只有一个大概的框架就开始编程了。当软件投运以后，用户提出各种各样的修改意见，有的是可以修改的，但有的是无法修改的，或者大面积的返工。最后只有两种可能，一个是软件不满意的“委屈”的运行，另一个就是被弃之不用。

因此，大型软件的制作必须精细的设计。而且必须是一个大的组织才能完成的，多个部门和人员协调的做好各自的工作。但如果没有事先精细的设计，扯皮推诿的现象会不断发生，软件组织将陷入无情无尽的会议的烦恼中。建模是什么呢？

模型是对现实的简化：

模型提供了系统蓝图，模型既可以是详细的计划，也可以从很高的层次考虑的系统总体计划。一个好的模型，应该包括那些有重要影响的主要元素，但忽略那些抽象水平不高的次要元素。

模型可以是结构性的，强调系统组织；也可以是行为性的，强调系统的动态方面。建模基本理由是为了更好的理解我们正在开发的系统。通过建模，要达到四个目的：

1）模型帮助我们按照实际情况或者按照我们所需要的样式对系统进行可视化；

2）模型允许我们详细说明系统的结构和行为；

3）模型给出一个指导我们构造系统的模板；

4）模型对我们做出的决策进行稳当化。

一个系统越大，越复杂，建模的重要性就越大。

归纳来说：

1，为什么要建模

1）必须对系统进行简化和抽象

目前软件系统也是一种非常复杂的系统，它的最终表现形式为可运行的目标代码。但是最终的软件代码一般是非常复杂的，会包含太多的细节信息，**直接阅读代码很难对系统有一个全面的了解**。我们需要有一个中间过程来得到这些结果，同时也需要对系统进行简化和抽象，这就是我们通常所说的系统设计。

2） 通过构建系统模型以对系统进行全面的分析和设计

利用统一建模语言 UML 来对系统结构进行全面的分析设计，即构建系统模型的过程，这就是可视化建模(Visual Modeling)。可视化建模技术已经成为一种成熟标准的软件开发技术规范。

2，什么是模型

1）模型是对现实世界的简化和抽象

现实世界中的系统是纷繁复杂的，直接去认识现实世界并且解决其中的问题是非常困难的。所以人们往往会构造一个模型来对现实世界中的复杂系统进行简化和抽象，通过这种简化和抽象来帮助设计人员加深对于系统的认知，在进行简化和抽象时我们抓住的是问题的本质，而过滤掉很多其他非本质的因素，从而帮助我们来简化问题的复杂性，有利于问题的解决。

2）各行各业都使用模型来辅助设计

模型在现实世界中大量存在，无论是研制飞机还是制造汽车，设计师们都会利用模型来研究目标课题的某一个侧面，如汽车的风阻系数、飞机机身的空气动力布局等等。在研发过程的大部分阶段中，设计师都不会去构造一个真实的系统来进行研究，因为这样的话成本太高了（或甚至是不可能的），同时问题本身没有得到足够的简化，很难找到问题的正确答案。

3）模型是沟通的手段

我们平时所见的模型有的是一种概念上的模型，如刚才提到的数学模型；有的是对实际系统外观的一个缩小，如轮船、飞机模型；还有的是对设计思想的一种展示，如建筑物的设计图纸等等。无论是哪一种模型，它的另外一个主要目的是帮助人们进行思想上的沟通，数学模型使别人了解你的逻辑思路，飞机模型向观众展示飞机的外观，设计图纸将设计师的设计思想传递给建筑工人。

4）模型可以精确地描述系统

语言和文字是人们进行沟通的主要手段，但语言和文字往往有二义性存在，较难保证人们的理解完全一致。所以在工程技术中，我们更多地是使用各种各样的模型来进行思想的沟通，模型可以精确地描述系统，同时保证整个系统开发过程的语义的一致性。

二、面向对象建模的方法

对于软件，可以有几种建模方法，最普通的方法是从算法的角度建模和从面向对象的角度建模。传统的软件开发是从算法的角度建模，所有的软件都用函数作为构造块，这种建模方法使设计人员把精力放在控制流程和对应的算法进行分析上，这种方法建立的模型是脆弱的，因为当需求发生变化的时候，这种模型将难以维护。

现代的软件开发采用面向对象的方式建模，所有的软件系统都用对象作为它的主要构造块，对面向对象系统进行可视化、评述和文档化，正是统一建模语言的（UML）的目的所在。为了更好的理解 UML 和应用 UML，我们必须对面向对象的语言体系有的比较好的理解。

1、面向对象的设计方法产生背景

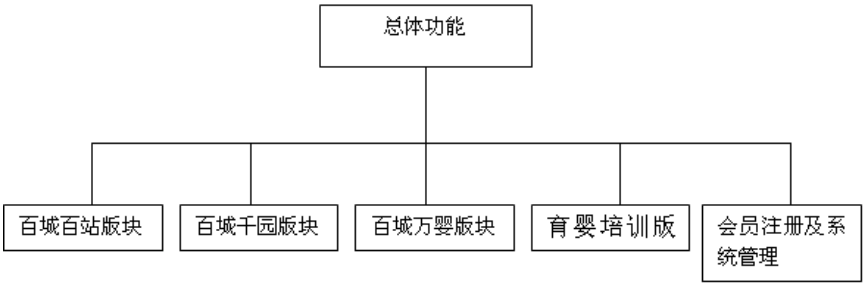
在 90 年代初，主要有结构化分析与设计方法和面向对象的方法， 而结构化分析与设计方法

主要是以数据为中心的方法（如采用实体关系图、数据流图等）；而面向对象设计方法则以对象为中心。

1）结构化方法

我们上面已经讨论过，结构化方法是一种基于功能分解的分析方法，并且自上向下的分解或分层。在结构化方法中首先要定义出需要哪些功能程序，每个程序应该实现哪些功能，然后按照某种方式把程序组织成一张图，该图称为结构图。

结构化分析与设计方法所具有的特点：自顶向下的分析和自底向上的开发实施；强调用户和用户参与的重要性；严格区分工作阶段。



结构化方法实际上就是按功能分解系统，比如设计一个工资系统，可以按功能划分成录入系统、打印系统、查询系统等等，这大约是传统程序员用得最多的一种方法。

2）面向对象方法

结构化方法所出现的问题：

在结构化方法中，用户使用软件的目的和过程的信息，都被直接抽象为了输入数据和得到反馈的数据。你只会看到用户做出输入什么数据然后得到什么数据输出的现象，至于用户在做出一件有什么业务意义的行为的信息在结构化模型中基本被抛弃。这些信息在面向对象的方法中得到保留并作为外部封装的信息。

面向对象方法的特点：

抽象性、封装性、继承性、多态性。

注意：

00 思想曾经遭受一些人的批评。理由是用户关心和理解的只是系统的功能，他不可能去学习 00 模型，所以虽然 00 建模缩小了分析设计和编码的鸿沟，但却拉大了和用户的距离。但是，随着面向对象的思想深入人心，使这一情况得到了大大的改观。

在 uml 中，用 00 建模的第一步是“用例”（也有称之为“用况”）的分析，“用例”体现了系统的功能单元。系统的外部人员或其它系统通过和“用例”交换消息来了解和使用系统的功能，弥补了 00 建模和用户之间的距离。

3.2 统一建模语言

一、UML 形成背景

1，UML 的主要特性

UML（Unified Modeling Language）**统一建模语言，**UML 是构建软件系统模型的标准化语言，因为

- **它提供了描述软件系统模型的概念和图形表示法**
- **同时由于它采用面向对象的技术、方法**，因此能准确方便地表达面向对象的概念，体现

面向对象的分析与设计风格。

- **它是编制软件蓝图的标准化语言**，用于对复杂软件系统的各种成分的可视化地说明和构造系统模型（建模是人类对客观世界和抽象事物之间联系的具体描述），以及建立软件文档。

因为**模型**的作用就是使复杂的信息关联简单易懂，它使我们容易洞察复杂堆砌而成的原始数据背后的规律，并能有效地使我们将系统需求映射到软件结构上去。

2，UML 的诞生

1）面向对象建模的标准语言的产生背景

在 UML 出现之前，人们已经开始采用面向对象的分析与设计，但是**很少有开发人员使用形象化的设计方法**，其主要原因就是缺乏统一的语言和语义来为复杂软件系统的组件定义、可视化、构建和编制文档。而 UML 的出现彻底的改变了这一现状，并成为了面向对象建模的标准语言。

2） 标准建模语言 UML 的出现

UML 的几位大师人物

Booch 是面向对象方法最早的倡导者之一，他提出了面向对象软件工程的概念。1991 年，他将以前面向 Ada 的工作扩展到整个面向对象设计领域。Booch 1993 比较适合于系统的设计和构造。

Rumbaugh 等人提出了面向对象的建模技术（OMT）方法，采用了面向对象的概念，并引入各种独立于语言的表示符。这种方法用对象模型、动态模型、功能模型和用例模型，共同完成对整个系统的建模，所定义的概念和符号可用于软件开发的分析、设计和实现的全过程，软件开发人员不必在开发过程的不同阶段进行概念和符号的转换。OMT-2 特别适用于分析和描述以数据为中心的信息系统。

Jacobson 于 1994 年提出了 OOSE 方法，其最大特点是面向用例(Use-Case)，并在用例的描述中引入了外部角色的概念。用例的概念是精确描述需求的重要武器，但用例贯穿于整个开发过程，包括对系统的测试和验证。OOSE 比较适合支持商业工程和需求分析。

面向对象的分析与设计(OOA＆D)方法的发展在 80 年代末至 90 年代中出现了一个高潮，UML 是这个高潮的产物。它不仅统一了 Booch、Rumbaugh 和 Jacobson 的表示方法，而且对其作了进一步的发展，并最终统一为大众所接受的标准建模语言。

3） 关于 UML 的形成

1994 年 10 月,Grady Booch 和 Jim Rumbaugh 开始致力于这一工作。他们首先将 Booch9 3 和 OMT-2 统一起来,并于 1995 年 10 月发布了第一个公开版本,称之为统一方法 UM 0.8(Unitied Method)。1995 年秋,OOSE 的创始人 Ivar Jacobson 加盟到这一工作。经过 Booch、Rumbaugh 和 Jacobson 三人的共同努力,于 1996 年 6 月和 10 月分别发布了两个新的版本,即 UML 0.9 和 UML 0.91,并将 UM 重新命名为 UML(Unified Modeling Language)。1996 年,一些机构将 UML 作为其商业策略已日趋明显。UML 的开发者得到了来自公众的正面反应,并倡议成立了 UML 成员协会,以完善、加强和促进 UML 的定义工作。当时的成员有 DEC、HP、I-Logix、 Itellicorp、 IBM、ICON Computing、MCI Systemhouse、Micr osoft、Oracle、Rational Software、TI 以及 Unisys。这一机

构对 UML 1.0(1997 年 1 月)及 UML 1.1(1997 年 11 月 17 日)的定义和发布起了重要的促进作用。

UML 是一种定义良好、易于表达、功能强大且普遍适用的建模语言。它溶入了软件工程领域的新思想、新方法和新技术。它的作用域不限于支持面向对象的分析与设计，还支持从需求分析开始的软件开发的全过程。

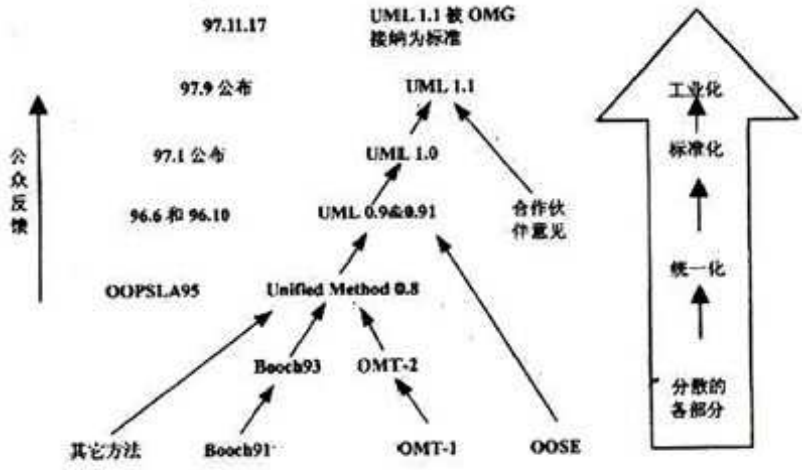


图 UML 的发展历程

面向对象技术和 UML 的发展过程可用上图来表示,标准建模语言的出现是其重要成果。在美国,截止 1996 年 10 月，UML 获得了工业界、科技界和应用界的广泛支持，已有 700 多个公司表示支持采用 UML 作为建模语言。1996 年底,UML 已稳占面向对象技术市场的 85%,成为可视化建模语言事实上的工业标准。1997 年 11 月 17 日，OMG 采纳 UML 1.1 作为基于面向对象技术的标准建模语言。UML 代表了面向对象方法的软件开发技术的发展方向,具有巨大的市场前景,也具有重大的经济价值和国防价值。

4）UML 语义：

UML 的描述基于精确元模型定义。元模型为 UML 的所有元素在语法和语义上提供了简单、一致、通用的定义性说明,使开发者能在语义上取得一致,消除了因人而异的最佳表达方法所造成的影响。此外 UML 还支持对元模型的扩展定义。

5）UML 表示法：

定义 UML 符号的表示法，从而为开发者或者开发工具使用这些图形符号和文本语法为系统建模提供了标准。在语义上它是 UML 元模型的实例

二、使用 UML 建模的意义

1，可以以图形的方式来展示系统的各个方面的特性

在工程设计中，工程师使用各种工程图来进行沟通。软件设计中通过使用 UML，**可以以 OO 的方式来进行系统的分析、设计**，并且已经被 OMG（Object Management Group）标准化了。

2，使用 UML 主要是基于如下的各种目的：

- **可以无歧义的理解：**因为在 UML 表示法中的每一个符号都有明确语义。这样，一个开发者可以用 UML 绘制一个模型，而另外一个开发者**可以无歧义的理解**这个模型
- **独立于实现语言的系统建模**：UML 不是一种可视化的编程语言，但用 UML 描述的模型可以与各种编程语言直接相连。可以进行正向工程和逆向工程两种工程-----独立于实现语言的系统建模。

- **UML 的文档化语言特性---产生各种制品**：一个健壮的软件组织除了生产可执行的代码之外，还要给出各种制品。通过这些制品的产出是度量和理解系统的关键。
- 可视化建模是一种交流和沟通工具，因为使用可视化建模可以抓住业务对象和业务逻辑

3，UML 最适于的过程

- 用例驱动的开发；
- 以体系结构为中心的开发；
- 迭代的和增量的开发。

由于 UML 的目标是以面向对象图的方式来描述任何类型的系统，因此其中最常用的是建立软件系统的模型，但它同样可以用于描述非软件领域的系统，如机械系统、企业机构或业务过程，以及处理复杂数据的信息系统、具有实时要求的工业系统或工业过程等。

三、UML 在软件开发过程中的应用

1，利用 UML 实现系统分析和设计的基本过程

- **3 个基本过程**

从应用的角度来看，当采用面向对象技术设计系统时，**首先是描述需求从而获得系统的主要功能；其次是根据需求建立系统的静态模型**，以构造系统的结构；最后则是**描述系统的行**为，从而描述出系统的各个对象之间的交互关系。

- **静态建模机制**

模型都是静态的，包括用例图、类图、包图、对象图、组件图和配置图等 5 个图形。所有这些都是标准建模语言 UML 的静态建模机制。

- **动态建模机制**

模型或者可以执行、或者表示执行时的时序状态或交互关系，它包括状态图、活动图、时序图和合作图等 4 个图形，所有这些都是标准建模语言 UML 的动态建模机制。

因此，标准建模语言主要可以归纳为静态建模机制和动态建模机制；它是一个通用的标准建模语言，可以对任何具有静态结构和动态行为的系统进行建模。

2，UML 适用于系统开发过程中从需求规格描述到系统完成后测试的不同阶段。

- **在需求分析阶段**

可以用用例来捕获用户需求。通过用例建模，描述对系统感兴趣的外部角色及其对系统(用例)的功能要求。

- **分析阶段**

主要关心问题域中的主要概念(如抽象、类和对象等)和机制，需要识别这些类以及它们相互间的关系，并用 UML 类图来描述。为实现用例,类之间需要协作，这可以用 UML 动态模型来描述。

- **在设计阶段**

只对问题域的对象(现实世界的概念)建模,而不考虑定义软件系统中技术细节的类(如处理用户接口、数据库、通讯和并行性等问题的类)。这些技术细节将在设计阶段引入,因此设计阶段为构造阶段提供更详细的规格说明。

- **编程(构造)是一个独立的阶段**

其任务是用面向对象编程语言将来自设计阶段的类转换成实际的代码。在用 UML 建立分析和设计模型时，应尽量避免考虑把模型转换成某种特定的编程语言。因为在早期阶段，模型仅仅是理解和分析系统结构的工具，过早考虑编码问题十分不利于建立简单正确的模型。

- **UML 模型还可作为测试阶段的依据**

系统通常需要经过单元测试、集成测试、系统测试和验收测试。不同的测试小组使用不同的 UML 图作为测试依据；

- 单元测试使用类图和类规格说明；
- 集成测试使用部件图和合作图；
- 系统测试使用用例图来验证系统的行为；
- 验收测试由用户进行，以验证系统测试的结果是否满足在分析阶段确定的需求。

3，统一建模中的“统一”含义

- 软件开发的整个生命周期（包括业务建模、用例建模、应用建模、数据建模）都可以用可视化建模技术统一起来。
- 在传统的开发技术中，这些步骤是由不同的技术完成的，如业务模型是由 IDEF 语言来描述，分析设计由数据流图来表示，数据库结构是用 ER 来定义等等。
- 在可视化建模技术中，所有的这些开发活动都可以由同一种语言 UML 来描述，这样可以大大增强团队的沟通，提高开发效率和软件质量。

4，UML 用来描述模型的内容

UML 词汇表包含三种构造快：也即事物（Things）、关系（Relationships）和图（Diagrams）。

- 事务： 事务是对模型中最具代表性的成分的抽象；
- 关系： 关系是把事务结合在一起；
- 图： 图聚集了相关的事务。

而这 3 种内容的主要细分的内容如下所述。

3.3 统一建模语言中的事务、关系和图

一、UML 中的事务

在 UML 中有 4 种事务：

- 结构事务
- 行为事务
- 分组事务
- 注释事务

这些事务是 UML 种基本的面向对象构造快。用它们可以写出结构良好的模型。下面简单的介绍一下：

1）结构事物（Structure Things）主要包括 7 种，分别是类、接口、协作、用例、活动类、组件和节点。

- 类是具有相同属性、相同方法、相同语义和相同关系的一组对象的集合。
- 接口是指类或组件所提供的、可以完成特定功能的一组操作的集合，换句话说，接口描述了类或组件的对外的、可见的动作。
- 协作定义了交互的操作，是一些角色和其他元素一起工作，提供一些合作的动作。
- 用例定义了系统执行的一组操作，对特定的用户产生可以观察的结果。
- 活动类是对拥有线程并可发起控制活动的对象（往往称为主动对象）的抽象。
- 组件是物理上可替换的，实现了一个或多个接口的系统元素。

- 节点是一个物理元素，它在运行时存在，代表一个可计算的资源，如一台数据库服务器等。

2）行为事物（Behavior Things）主要有两种：交互和状态机；在 UML 图中，交互的消息通常画成带箭头的直线，状态机是对象的一个或多个状态的集合。

3）组织事物（Grouping Things）是 UML 模型中负责分组的部分，可以把它看作一个个盒子，每个盒子里面的对象关系相对复杂，而盒子与盒子之间的关系相对简单。组织事物只有一种，称为包；包是一种有组织地将一系列元素分组的机制。

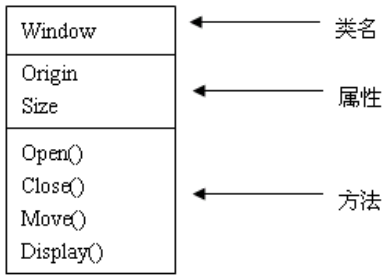
4）辅助事物（Annotation Things），也称注释事物，属于这一类的只有注释。注释即是 UML 模型的解释部分。在 UML 图中，一般表示为折起一角的矩形。

1，结构事务

结构事务（structural thing）通常是模型的静态部分，描述概念或物理元素。共有七种结构事务：

1）类（class）

是对一种相同属性、相同操作、相同关系和相同语义的对象的描述。一个类实现了一个和多个接口。下面是类的图形。



2）接口（interface）

描述一个类或构件的一个服务的操作集，也就是说，接口描述元素的外部可见行为。

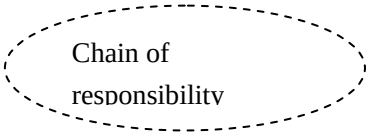
接口定义了一组操作的描述（也就是特征标记）。接口很少能单独存在，而是依附于实现接口的类或者构件。接口图形如下：


ISpellin

3）协作（collaboration）

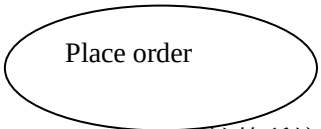
定义了一个交互，它是由一组共同工作，以提供某些协作行为的角色和其它元素构成的一个群体。这些协作行为大于所有元素各自行为的总和，因此，协作有结构，行为和维度。

一个给定的类可以参与几个协作，这些协作因而表现了系统构成模式的实现。其图形（责任的链条）如下：



4）用例（use case）

是对一组动作序列的描述，系统执行这些动作，将产生一个对特定的参与者有价值而且可观察的结果。用况用于对模型中的行为事务结构化，用况是通过协作实现的。其图形如下：



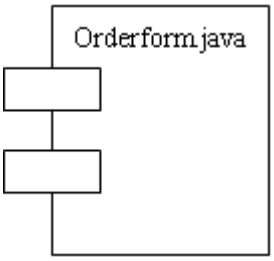
剩下的三种事务：主动类、构件和节点，都和类相似，也就是说它们也描述了一组具有相同属性、相同操作、相同关系和相同语义的对象，不过这三种事务和类的不同点也不少，而且对于面向对象某一方面的建模是必需的，因此需要单独处理。

5）主动类（active class）

其对象至少拥有一个进程或线程，因此它能够启动控制活动。主动类的对象所描述的元素的行为和其它元素的行为并发，除了这一点外，它和类是一样的。图形（注意外框为粗线）如下：

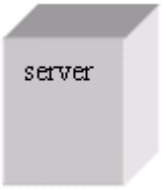
6）构件（component）

有的时候可能更通用的名字是组件。是系统中物理的、可替代的部件，它遵循并且提供一组接口的实现。在一个系统中可能会遇到不同的部件结构，比如：COM+或者 Java Beans 等。图形如下：



7）节点（node）

运行时存在的物理元素，它提供了一种可计算的物理资源，它通常至少需要一些记忆和处理能力。一个构件集可以驻留在一个节点内，也可以从一个节点迁移到另外一个节点。图形如下：



这七种元素即类、接口、协作、用况、主动类、构件和节点，是 UML 模型中可以包含的基本结构事务，它们也有一些变体，比如：参与者、信号、实用程序（一种类）、进程和线程（两种主动类）、应用、文档、文件、库、页和表（一种构件）等。

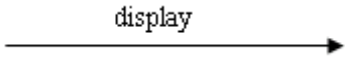
2，行为事务

行为事务（behavioral thing）是 UML 模型的动态部分。它是模型中的动词，描述了跨越时间和空间的行为。共有两类主要的行为事务：

1）交互（interaction）

它由在特定语境中共同完成一定任务的一组对象之间交换的消息组成。一个对象的行为或单个操作的行为，可以用一个交互来描述。交互涉及一些其它的元素，包括消息、动作序列和链。其中：

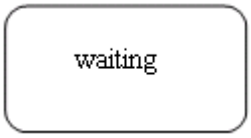
- 动作序列：由一个消息所引起的行为；
- 链：对象间的连接；
- 图形：有向线段，上边标注操作名。



2）状态机（state machine）

它描述了一个对象或一个交互在生命周期内响应事件所经历的状态序列。单个类或一组类之间协作的行为可以用状态机来描述。一个状态机涉及到一些其它的元素，包括状态、转换、事件和活动。其中：

- 转换：从一个状态到另一个状态的流；
- 事件：触发转换的事务；
- 活动：对一个转换的响应；
- 图形：状态的名称为等待。

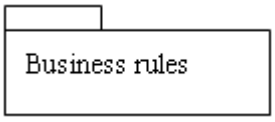


3，分组事务

分组事务（grouping thing）是 UML 模型的组织部分。它们是一些由模型分解成的“盒子”，在所有的分组事务中，最主要的分组事务是包。

包（package）是把元素组织成组的机制，这种机制具备多种用途，结构事务、行为事务甚至其它的分组事务，都可以放进包里面去。包不象构件（仅在运行的时候存在），它纯粹是概念上的（也就是说它仅在开发时存在）。

图形：（交易规则）

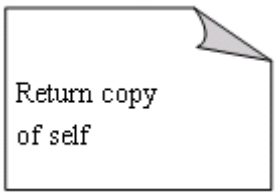


包还有一些变体，比如框架、模型和子系统等，它们是包的不同种类。

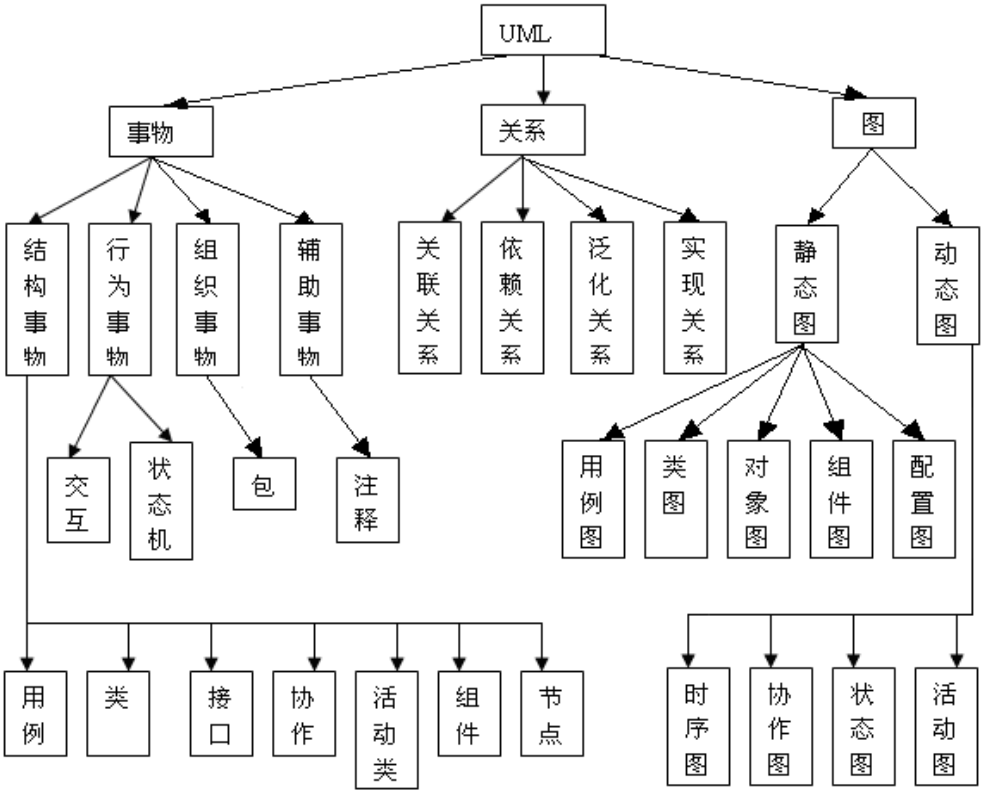
4，注释事务

注释事务（annotational thing）是 UML 模型的解释部分，这些注释事务可以用来描述、说明和标注模型的任何元素。有一种主要的注释事务，称之为注解（note），它是依附于一个或一组元素之上，对它进行约束或解释的简单符号。

图形：



5，3 种内容的图示

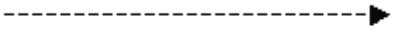


二、UML 中的四种关系

UML 中的四中关系包括：依赖、关联、实现和泛化。从语义上理解，关联、实现和泛化都是依赖关系，但因为他们有更特别的语义，所以在 UML 中被分离出来作为独立的关系。有的时候，也可以把关联关系延伸出来的聚合关系独立考虑，这四种关系说明如下。

1）依赖（dependency）

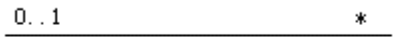
两个事物间的语义关系，其中一个事务（独立事务）发生变化会影响另一个事务（依赖事务）的语义对于两个对象 X、Y，如果对象 X 发生变化，可能会引起对另一个对象 Y 的变化，则称 Y 依赖于 X。例如在软件中，如果类 A 的一个操作调用类 B 的一个操作，且这两个类之间不存在其他关系，那么类 A 和类 B 之间是依赖关系。其符号是：



依赖（Dependencies）

2）关联（assoctation）

是一种结构关系，它描述了一组链，链是对象之间的连接，也就是指一种对象和另一种对象有联系。给定关联的两个类，可以从其中的一个类的对象访问到另一个类的相关对象。例如：在学校中，一个学生可以选修多门课程，一门课程可以由多个学生选修，那么学生和课程之间是关联关系。其符号是：

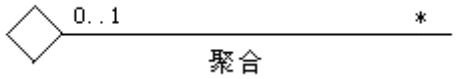


关联（Association）

3）聚合

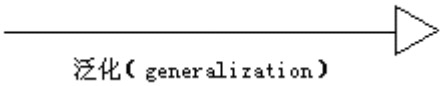
是一种特殊的关联，它描述了整体和部分之间的结构关系。例如：森林和树木之间是聚合关

系。其符号是：



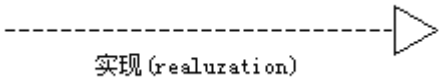
4）泛化（generalization 普遍化，一般化）

是一种“一般/特殊”关系，特殊元素（子元素）的对象可替代一般元素（父元素）的对象。用这种方法，子元素共享了父元素的结构和行为。例如：在 MFC 类库中，Window 类和 DialogBox 类之间是泛化关系。其符号入夏，图形上空心箭头指向父元素。



5）实现（realization）

是类元之间的语义关系，其中一个类元指定了由另一个类元保证执行的契约。或者说将一种模型元素（如类）与另一种模型元素（如接口）连接起来。在两种地方要用到实现关系：一种是接口和实现它们的类或者构件之间。另一种是在用例和实现它们的协作之间。其符号是：



这些关系是 UML 的基本关系构造，用它们可以写出结构良好的的模型。

三、UML 中的九种图

为了对系统可视化，图（diagram）是一组元素的图形表示，大多数情况下把图画成顶点（代表事务）和弧（代表关系）的连通图。

1） 5 种静态图

- 用例图（use case diagram），描述系统功能；
- 类图（class diagram），描述系统的静态结构；
- 对象图（object diagram），描述系统在某个时刻的静态结构；
- 组件图（component diagram），描述了实现系统的元素的组织；
- 配置图（deployment diagram），描述了环境元素的配置，并把实现系统的元素映射到配置上。

2） 4 种动态图

- 序列图（sequence diagram），按时间顺序描述系统元素间的交互；
- 协作图（Collaboration diagram），按照时间和空间顺序描述系统元素间的交互和它们之间的关系；
- 状态图（state diagram），描述了系统元素的状态条件和响应；
- 活动图（activity diagram），描述了系统元素的活动。

对上面的 9 种图，根据它们在不同的应用中，把 9 种图分成如下的五类模型视图：

- 用户模型视图：用例图
- 结构模型视图：类图、对象图
- 行为模型视图：序列图、协作图、状态图、活动图（动态图）
- 实现模型视图：组件图
- 环境模型视图：配置图

注意：其中最重要和使用最多的 3 种图是用例图(User Case Diagram)、时序图(Secquence

Diagram)和类图(Class Diagram)。

- **用例图是从用户的角度来描述系统的外部功能的图**：对不同的用户，系统应该有不同的功能，所以一个系统的用例图通常会有多个。
- **时序图是从系统实现的角度来描述每一个用例**：可见一个用例一般应该有一个对应的时序图它描述的是一个用例中用户与系统进行消息传递，系统各部件(类)之间进行消息传递的关系和顺序。时序图表现的是系统具体实施的静态逻辑，从它可以直接联系到代码中一个模块的具体实现逻辑，所以它对系统逻辑的具体实施是最重要的。
- **类图直接对应到源代码：**在面向对象的程序设计中，整个系统都是由类及其实例(对象)通过相互之间发送消息以及各自的消息处理函数来实现的。

时序图描述了各个类之间要发送的消息，类图则表明该如何来发送、接收和处理这些消息，这包括各种属性和方法。

四、主要 UML 图的说明

1、 类图

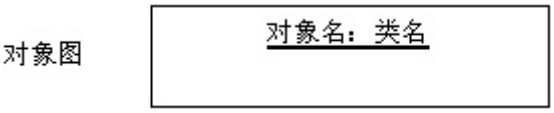
类图中的类和我们经常提到的面向对象软件设计与开发中的类是同一个概念，用来表示这么一个类的图我们就称之为类图。它主要是展示了系统或者领域中的实体以及实体之间的关联，类的 UML 图是一个矩形框。类图对系统分析有很大的帮助，它可以让系统分析员使用客户所采用的术语和客户交流，这样就可以促使客户说出所要解决的问题的重要细节。

2、 对象图

同类一样，对象在这里的概念与面向对象编程思想中的对象一样，它是类的实例。是具有具体属性值和行为的一个具体事物。

面向对象技术已经席卷了整个世界，事实也确实如此。作为一种程序设计方法，它的建立具有很多优点。基于构件的软件开发方法就是面向对象技术孕育出来的。采用这种方法建立一个系统时，首先建立一组类，然后通过增加已有构件的功能或者添加新的构件来逐步扩充系统，最后在建立一个新系统时，还可以重用已经建好的类。这样做可以大大削减系统开发时间。

使用 UML 可以建立起易于使用和易于理解的对象模型，以使程序员能够创建出这些模型所对应的软件。所以，UML 对基于类开发的全过程都有益处。对象图也是一个矩形，和类一样，但是对象名下面要带下划线。



3、 用例图

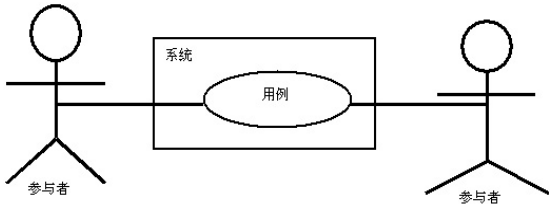
用例是用来描述潜在的用户所看到的系统的 UML 组件。

它是一个被称为参与者（可以是一个人、一个硬件设备、一段时间的流逝或者另一个系统）的实体所发起的场景的集合。用例的执行必须对发起改用例的参与者或者其他参与者产生影响。用例可以被重用。与用户会谈是导出用例的最好技术，当导出一个用例时，要注意到发起用例的牵制条件和产生影响的后置条件。

用例是一个强有力的工具，当使用 UML 可视化地表达出这些概念后用例甚至会变得更加强大。可视化允许你向用户现显示用例，他们能为你提供更多的信息。系统分析过程的一个目标是产生一组用例。此想法是要对用例进行分类整理，以便于引用。用例代表着用户的观点。当系统要进行升级时，用例目录可以作为进一步收集升级需求的基础。

用例是由参与者发起的，参与者（也许是发起者，但不是必须的）能够从用例的执行中获得有价值的事物。用例分析的一个好处是它能展现系统和外部世界的边界。参与者是典型的外部实体，而用例是典型的系统内部。参与者、用例和互连线共同组成了用例模型。如下图示。

在用例模型中，直立人形图标代表参与者，椭圆代表用例，参与者和用例之间的关联线代表两者之间的通信关系。



4、 状态图

在计算机系统中，当系统和用户（也可能是其他系统）交互的时候，组成系统的对象为了适应交互要经历必要的变化。

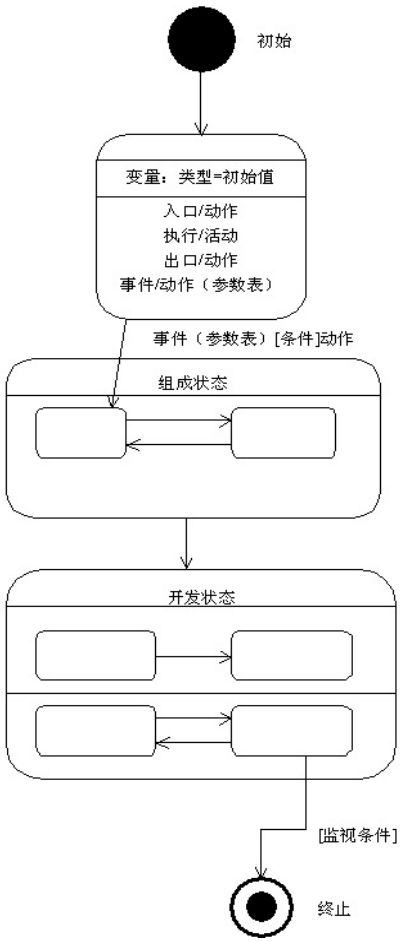
一种表征系统变化的方法可以说是对想改变了自己的状态以相应事件和时间的流逝。UML 状态图就是展示这种变化的工具，它描述了一个对象所处的可能状态及状态间的转移，并给出了状态变化序列的起点和终点。

要注意，状态图与以上提到的类图、对象图和用例图有着本质的不同。前 3 种图能够对一个系统或至少一组类、对象或用例建立模型，而状态图只是对单个对象建立模型。

状态图描述一段时间内对象所处的状态和状态的变化。状态的 UML 图标是一个圆角矩形，状态转移用状态之间的有向连线表示。

UML 状态图提供了多种表示法符号，并且包括了很多建模思想——如和对单个系统对象所经历的变化建模。或许对于很简单的问题建模时，这种类型的图可能很快就会变得很复杂，但是，事实上确实很需要状态图，因为它能帮助系统分析员、设计员和开发人员理解系统中对象的行为。

类图和对应的对象图只展示了系统的静态方面，他们展示的是系统的静态层次和关联，并能够告诉你系统的行为是什么，但它们不能说明这些行为的动态细节。



5、 顺序图

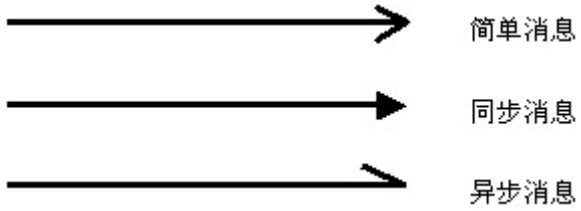
状态图的焦点是对象的状态，这只是图片的一小部分。UML 顺序图更进一步显示出随着时间的变化对象之间是如何通信的。

UML 顺序图在对象交互的表示中加入了时间维。

在顺序图中，对象位于图的顶部，从上到下表示时间的流逝，每个对象都有一个垂直向下的对象生命线，对象生命线上的窄矩形条代表激活——改对象某个操作的执行。可以沿着对象的生命线表示出对象的状态。

消息，有简单的、同步的或异步的三种，分别表示为如下图示的用连接对象生命线的带箭头的连线代表。

消息在垂直方向上的位置表示了该消息在交互序列中发生的时间，越靠近图顶部的消息发生的越早，越靠近底部的发生的越晚。

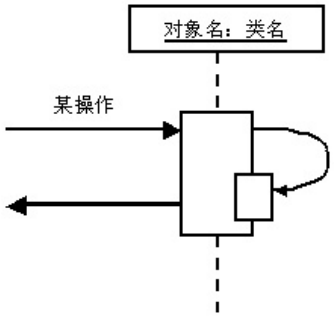


用例图可以只说明用例的一个实例（场景），或者可以表示一般的或者一个用例的所有场景。

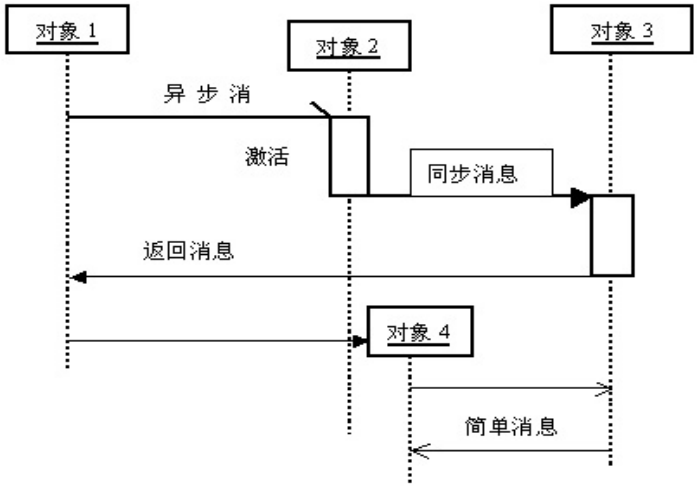
一般顺序图中通常提供了表示“if”条件语句和“while”循环语句的机会，每个“if”条件

语句要用方括号（[]）括起来，“while”循环语句也要用方括号（[]）括起来，并在左括号前面加一个星号。

同时，一般来说，对象可能会有一个调用自身的操作，即递归或自身调用。自身调用的表示是从一个激活框中引出消息线又重新回到这个激活框，并在该激活框中附加上一个小的矩形框，其表示方法如下图示。



下面是一个顺序图的表示。



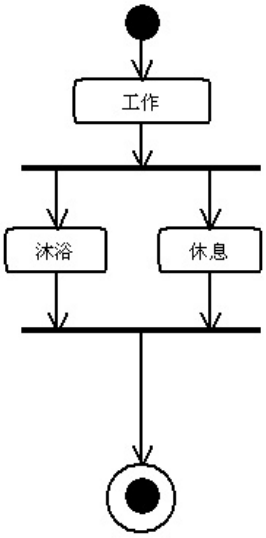
6、 活动图

UML 活动图是状态图的一种扩展形式，它展示出对象执行某种行为时或者在业务过程中所要经历的步骤和判定点。

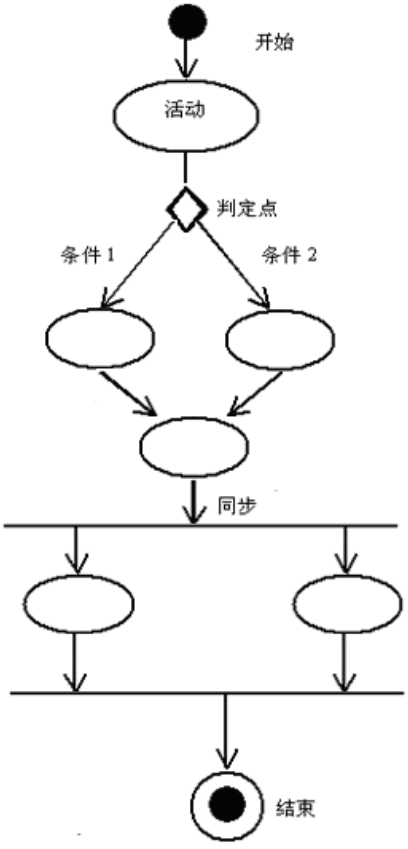
每个步骤（活动）用一个圆角矩形（比状态图更扁更圆）表示，菱形图标代表判定点。它很像程序设计课中学到的流程图。

UML 活动图可用于表达一个对象的操作和一个业务过程。活动图与状态图的主要区别是，状态图图出显示的是状态，而活动图突出显示的是活动。但它是状态图的扩展，有人认为活动图并不是是状态模型的一种，它仅仅反映了一种工作流程，这是不正确的。

当一个活动路径分成两个或多个路径时，可以用一个与路径垂直的粗实心线来代表路径的分支，两个并发路径的合并可以用相同的方式表达。并发活动表示法如下图。



活动图中可以显示出信号：发送信号的图符是一个凸无边形，接收信号的是一个凹无边形。在活动图中还可以表示出执行每个活动的角色，即通过将活动图划分为泳道——代表每个角色的平行段。还可以在活动图中出现其他图的图符并绘制混合图。一个活动图的大概情况可以表示为如下图示。



7、 协作图

对象图展示出对象和对象之间的静态关系，协作图是对象图的扩展。

协作图可视化地表示了对象之间随时间发生的交互，它除了展示对象之间的关联，还显示出对象之间的消息传递。

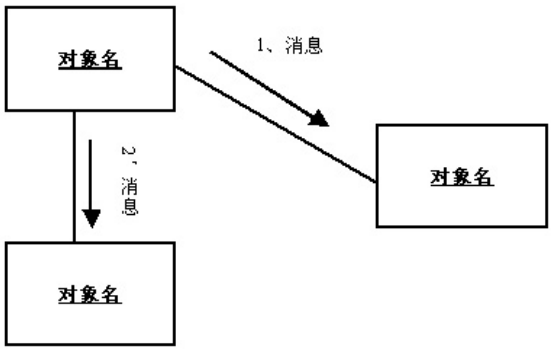
与顺序图一样，协作图也展示对象之间的交互关系，实际上，顺序图和协作图两者是语义等

价的。也就是说，这两种图表达的是同一种信息，并可以将顺序图转化为协作图，反之亦然。但是，顺序图强调的是交互的时间顺序，而协作图强调的是交互的语境和参与交互的对象的整体组织。

还可以从另一种角度来看两种图的定义，以区分这两种图：顺序图按照时间顺序布图，而协作图按照空间组织布图。

协作图中可以表示出一个对象按照指定的次序（或无次序）地向一组对象发送消息。还可以表示拥有消息控制流的主动对象，以及消息之间的同步。

协作图示例如下图。对象图标可以布置在图中的任何位置。对象间的连线代表了对象之间的关联和消息传递。每个消息箭头都带有一个消息序号，这些序号说明了该消息在交互序列中的序号。



8、 构件图

软件构件是软件系统的一个物理单元，它驻留在计算机中而不是只存在系统分析员的脑海里。像数据表、数据文件、可执行文件、动态链接库、文档等都可以称为构件。

至于构件和类的区别可以这样理解：构件是类的软件实施。类是代表一组属性和操作的抽象实体。类和构件的一个重要关系是：一个构件可以是多个类的实施。既然构件是驻留在计算机系统中的工作单元，对它建模是不是多此一举呢？我们说不是。

因为对构件和构件的关系建模具有如下意义：

- 使客户能够看到最终系统的结构；
- 让开发者有一个目标；
- 让编写技术文档和帮助文件的技术人员能够理解所写的文档是关于哪方面内容的；
- 利于重用等。

构件的一个重要方面是它具有潜在的重用性。在当今高节奏的商业竞技场中，你建造的系统发挥功能越快，在竞争中获得的利益就越多。如果在一个系统中所构造的构件在开发另一个系统时被重用，那么就越有利于获得这种竞争利益。

在建立构件的工作上花费一些时间和精力有助于今后的重用。在对软件实体进行建模的过程中，你可能会遇到三种类型的构件：

①、部署构件（Deployment Component），它形成了可执行系统的基础。例如动态链接库、二进制可执行体、ActiveX 控件等。

②、工作产品构件（Work Product Component），它是部署构件的来源，如数据文件和程序源代码。

③、执行构件（Execution Component），是可运行系统产生的结果。构件图中包括构件、接口和关系。当然前面介绍的其他类型的图标也可以加入到构件图中。

构件图的图标是一个左侧附有两个小矩形的大矩形框，如下图示。

构件的名字位于构件图标的中央。如果构件是一个包的成员，那么构件名之前要加上包的名

字，还可增加一些表达构件的细节信息。

构件图可以通过构件的接口来访问一个构件，构件的接口使一组操作集合。构件和接口之间的关系叫做实现关系。一个构件可以访问另一个构件提供的服务。当这样做的时候，它要使用导入接口，而实现服务接口的构件对访问它的服务的构件提供服务接口。

9、 部署图

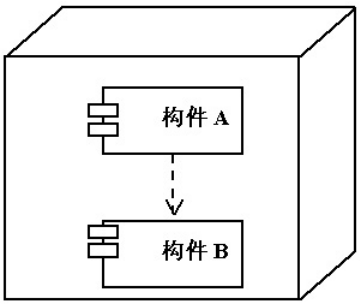
在大多数系统中，硬件也是一个重要方面。

现在的计算机领域，一个系统可能要包括无数种的操作平台，并且要有可能还要跨越很长的物理距离。

一个坚实的系统硬件部署图对系统设计来说是必不可少的。

从一开始的类的概念到上面的驻留在计算机中的软件构件，这里我们来看真实世界中的计算机硬件。

将 UML 部署图与整个系统集成到一起后将看到完整的物理结构图。系统是由节点组成的，每个节点用一个立方体表示。节点之间的连线代表两个立方体的连接。两种类型的节点分别是处理器（它可以执行软件构件）和设备（不能执行软件构件）。设备通常具有同外部世界通信的接口。部署图对建立网络结构的模型很有用处。



任何语言和符号，只有在应用中才有生命力，所以下面的讨论，我们并不准备以图为线索展开，而是以一个软件开发过程展开，仔细分析 UML 在软件开发各个过程中的应用，使图的应用成为一个活的应用，从而更加深刻的理解 UML 及其本质。

第四章 用例模型与系统分析

在面向对象的分析中，对象、事件和响应成为分析的主体，分析的着力点转向了交互。目前，描述功能和交互最重要的方法就是用例方法，这也是需求分析的重要组成部分。

4.1 深入理解用例方法

一、用例的完整概念

用例是什么？用例是代表系统中各个项目相关人员就系统的行为达成的契约，用例描述了在不同条件下，系统对某一参与者的请求所作出的响应，参与者通过发起与系统的一次交互来实现某个目标。

当然可以用流程图、顺序图等表示这种交互，但通常情况下用例是文本格式，因而更容易人员之间相互交流，而且对交互过程的表达也更加精确。

用例可以被用来记录需求，也可以描述业务过程，或者用来记录一个软件的行为需求。

一个编写很好的用例应该具有很好的可读性，它由多个句子组成，所有句子都采用同一种语法形式，也就是一个简单的执行步骤。这样一来阅读用例将变得很容易。

要编写好一个用例，必须掌握三个概念：

- **范围**：真正被讨论的系统是什么？
- **主要参与者**：谁要求实现他的目标？
- **层次**：目标的层次是高还是低？

需要注意的问题是，根据目标的不同，用例可以在不同的层次上描述，例如：

- **概要目标**：描写一个需要经过多次处理才能达成的目标；
- **用户目标**：描写经过一次处理就可以达成的目标；
- **子功能**：描述用户目标的一部分。

对于任何一个系统架构级别的用例，都会编写**黑盒**用例，这种用例将会专注于功能划分级别的行为，而不考虑内部细节。而对于业务过程的设计者，将会编写**白盒**用例，他会描述组织内部过程如何运作，技术开发者也需要利用白盒用例描述将来的系统具体工作情况。

在不同的层次，用例的描述方法将会很不相同。

1，需求与用例

如果把用例作为需求来编写，那么需要记住以下几点：

- **用例确实是需求**：不需要把用例转换为需求的其它形式，如果编写恰当，用例可以准确地对系统必须要做什么进行详细地描述。
- **用例不是所有的需求**：用例不能描述外部接口、数据格式、业务规则以及复杂的公式，用例也不能完整地描述非功能需求（质量属性和约束条件）的大部分内容。所以用例仅仅是需求的一部分，尽管是非常重要的一部分。
- **用例重在表达行为需求**：用例表达的主要是行为需求，而且是所有的行为需求。因为功能一般是用行为实现的，所以行为需求将包括多个功能需求。但是，其它还有很多的需求，例如：业务规则、词汇表、性能目标等都不属于行为需求之列。

2，用例作为项目连接结构

用例可以把许多其它需求细节连接在一起，为连接在一起的不同部分的信息提供了一个框架，用例也有助于用户把概要信息、业务规则、数据格式等需求内容进行交叉连接。

用例也有助于组织项目计划信息，如发布时间、优先级、小组的并行开发，也有助于设计小组跟踪项目进程，以及规划项目测试。虽然这些信息并不在用例之中，但都与用例有关。

3，用例的增值点

用例被广泛采用的主要原因：详细描述系统被使用的时候行为细节，使得用户能够明白新系统应该是什么样的。这有利于用户尽早对系统运行的细节进行判断：是欣然接受还是拒绝？但是应用用例还有更重要的原因：

1）便于对系统目标的描述

一般用例的命名被冠以系统的目标，并被收集整理成一个列表。这个列表声明了系统可以做什么，提示了系统的范围以及创建系统的目的。他成为项目不同的相关人员交流的一个工具。

用户代表、主管、资深开发人员和项目经理会对这个目标仔细研究，并由此对项目的费用和复杂性作出初步的估算，然后他们再一起协商首先开发哪些功能，如何组建开发小组。这个列表也可以成为进行复杂性、经费、时间和状态度量的框架，它收集了项目生命周期中各种各样的信息。

2）加强了对异常情况处理的描述

一般在描述功能和行为的时候，大部分情况都关注系统应该做什么，而往往忽视的是异常处理描述，从而发现许多以前没有考虑过的意外情况。当考虑例外情况的时候，通常总要和领域专家聚在一起，或者通过电话商量这种情况将如何动作，这样，就有可能发现新的项目相关人员、系统、目标和业务规则。

如果没有这些离散的用例步骤，以及对失败情况所作的集中讨论，那么许多错误情况就不可能在实际开发之前被发现。如果错误情况是在程序员编写代码的时候才被发现，这对发现新功能和新的业务规则来说就太晚了，那时业务专家已经走了，时间也非常紧迫，这时程序员只能想当然的按自己的想法编写相应代码，而不去寻求更理想的解决方案。

人们只要早期花不长的时间考虑异常步骤，就可以在后期得到更多的好处，且不说这已经挖掘出了潜在的需求，从而节省了大量的时间。

二、用例是规范行为的契约

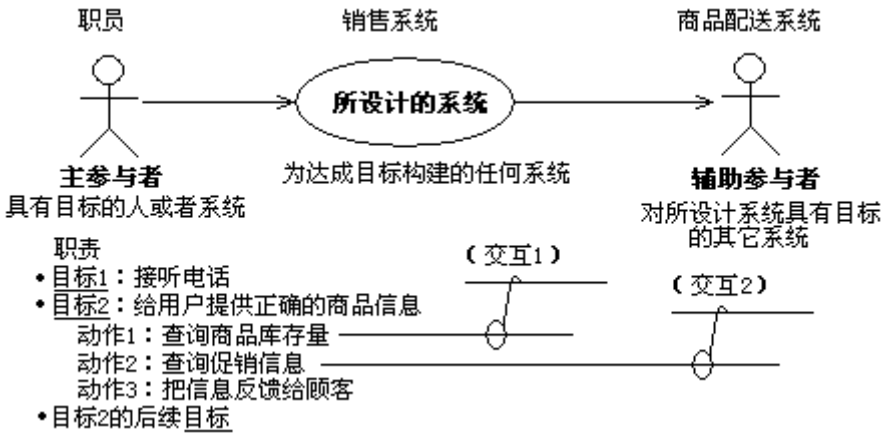
为了正确的开发系统，项目相关人员例如用户、需求分析人员、架构设计人员、编码人员、测试人员需要一个统一的契约，用例构成了契约中的行为部分。用例中的每个句子都有其价值，即它们各自描述了一项行为，该行为维护了或者增进了某些项目相关人员的利益。

因此，我们应该仅仅从具有某种目标的执行者之间交互行为的角度来考察一个用例。从概念上讲，首先是关注“执行者和目标的概念”，其次是关注“项目相关人员和利益”的概念。

1，具有目标的执行者之间交互

1）执行者具有目标

假设一个电话销售系统，职员负责处理电话购物请求（这个职员也就是主要执行者）。当客户打进电话来的时候，该职员就产生了一个目标：让计算机注册并启动这个请求。



在本例中，某些子目标要借助“辅助参与者”来实现，例如商品配送系统是原来就已经存在的业务系统，它所要完成的是一个子目标：根据订单，把正确的商品送到顾客手上。这个子目标也就是辅助参与者必须履行的承诺。

一般来说，最高的目标可以通过一系列的子目标来实现，但是不能这样无限细分下去，所以编写好的用例最大的困难，也就是目标粒度如何划分才是合理的？这需要我们对用例的分层有透彻的理解。

2）目标可能失败

如果职员记下顾客请求的时候，计算机崩溃了怎么办？这就需要为职员建立一个备选目标，例如用纸和笔记录顾客要求，并通过另外的工作流程使承诺能够实现，本质上这又出现了一个新的需求。

另一方面，系统在执行某些子目标的时候会遭遇失败，例如传送了错误的数据，或者商品配送系统可能由于某些状况不能运转（遭遇员工罢工？）。在这样的情况下，正常的工作流程无法进行，是采取备选方式呢？还是向顾客道歉？用什么方式道歉？

强调目标失败和思考目标失败后系统的反应，是用例被认为是出色的系统行为描述工具的原因之一，很多分析师和设计师都从中获得了重要的好处。

3）交互是复合的

最简单的交互是发送一条消息，但是更多的情况是一组消息序列或者场景，这称之为复合交互。例如上面的例子：

a、查询商品库存量。

b、查询促销信息。

c、把信息反馈给顾客。

d，获取顾客要求。

在更高的抽象层次上，也可以把这个活动序列进行压缩，把它作为一个单独的步骤：

a，查询商品状态，获取顾客要求。

因此，交互可以根据需要折叠和分解，就像目标能大能小一样。其中每一步都能展开成一个单独的用例，也可以把多个交互合并成一个用例，就像上面讨论的子目标问题是一个样的。

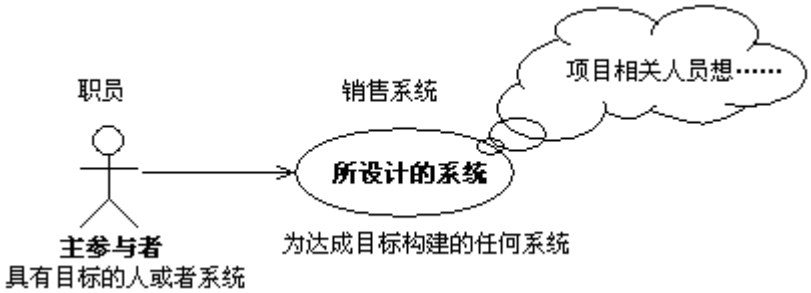
重要的是，通过对目标交互进行折叠，可以在一个很高的层次上表示系统行为，而通过把每个高层交互一点点地展开，也可以精细的刻划系统。用例常常被看成可以不断展开下去的故事，用例编写者的工作就是把这个故事书写的情节连冠、条理清楚，以便读者可以很舒适的在故事情节中切换、穿梭。

2，具有利益的项目相关人员之间的契约

执行者和目标模型解释了如何编写用例中的句子，但没有涉及如何描述行为方面的内容。出

于这个原因，需要对用例基于“基于项目相关人员之间的契约”这个观点进一步扩展。

例如：ATM 必须保留一个日志来记录所有的交互过程，以备发生争议的时候项目相关人员的利益能得到保护。它也记录其它信息，以便能够记录失败以前交易到底进行到什么程度？ATM 在交付现金以前，首先要核实账号持有人是否有足够的存款，以确保 ATM 不会向客户提供比他在银行实际存款数还多的现金。



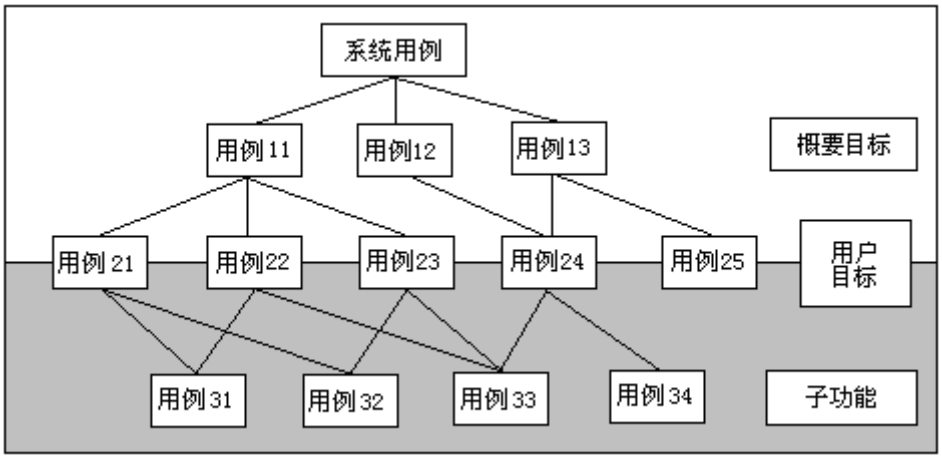
用例作为规范行为的契约，捕获了为满足项目相关人员的所有利益，并且权限于此。因此，要认真完成一个用例，就必须要列出所有的项目相关人员，根据用例的操作命名他们的利益，以及如果用例成功完成靠什么保证他们的利益。把这些搞清楚了，就可以编写用例的步骤，也就知道了用例步骤应该包括什么，不应该包括什么。

很多人并没有如此认真的编写用例，也幸运的完成了任务，很多东西是在软件开发的时候重新捕获得到。这种做法有时使项目进行得很顺利，但有时却会招致巨大的损失，这依赖于开发人员的水品。我们的任务是使项目开发不应该因为开发人员的水平波动对项目质量造成巨大影响。

三、用例的目标层次

我们很常见的情况时，需求分析写了 100 页，写满了用例文档，这些用例非常详细，但是这种冗长乏味的文档对编写者和阅读者都没有什么帮助，事实上如果描述清楚 6 个高层用例，就可以是用例更容易理解和处理。

这就给我们提出了一个问题，我们的用例应该描述哪个层次上的目标呢？首先，我们需要给不同的目标命名，在下面的图中，水平线作为分界线，这条线上的目标我们称之为“用户目标”，用户目标之上的我们称之为“概要目标”，这条线之下的我们称之为“子功能”，如下图所示。



1，概要层次

概要层次包含多个概要目标，每个概要目标包含多个用户目标。主要在系统层次描述，包括

如下各方面的功能：

- 显示用户目标运行的语境；
- 显示相关目标生命周期的顺序；
- 表达子系统和功能块黑盒层面的状态与行为。
- 为底层用例提供一个目录表；

对于系统层次的用例，我们可以参照下面的过程来进行：

1）以一个用户目标作为开始

2）考虑目标所提供的服务

这个目标对于主参与者 A（最好在组织外部）提供什么服务？而参与者 A 是我们想要收集用例的最终主参与者。

3）定义与 A 相关的一个设计范围 S

保证 A 在 S 之外，并且给 S 命名。一般这样的设计范围有三种类型：

- 某个实际存在的组织（公司等）；
- 参加整个系统的某个独立的软件系统；
- 被设计的具体软件系统。

4）找出 A 在设计范围 S 中的所有用户目标

5）找出 A 对系统 S 具有的概要目标 G

6）编写概要级别的用例

为 A 对系统 S 的目标 G 编写概要层次的用例，这些用例把一些概要级别的用例维系在一起。

7）用架构用例表达子系统和主要构件的行为

从本质上，每个概要层次的用例表达的是一些子系统、主要构件的黑箱级别的行为和状态描述，这样的用例也称为架构用例。用这些架构级别的用例在总体上把工作系统连起来很有帮助，编写出这样的用例以后，将有助于架构设计和实现，也有助于架构评审。基于这样的原因，我们极力推荐认真编写好最外层的用例。当然这样的用例并不包括系统所有的功能需求。

从上面的过程也可以发现，早期发现系统用户目标层次的能力（最好用用户描述表达），对于发现概要层次的目标是很有意义的。

2，用户目标层次

在用例编写上，我们最感兴趣也是需要倾注绝大部分精力的是用户目标，它是用户使用系统的目标。

1）使用用户描述反映初始需求

经典软件开发过程要求在任何设计和实现工作之前，尽可能的推敲，把需求完全定义清楚，并把它稳定下来。这一方面不太可能，另一方面这种重量级的需求分析往往缺乏足够的抽象，庞大的需求文档使选择搞优先级的需求变的不清晰而且困难。所以在敏捷开发中初始需求推荐使用一种轻量级的表达方式：用户描述（user story）。应该注意，一旦选中了若干描述进入迭代周期，就需要把这些用户描述用正规的方式把需求表达出来也就是说每个迭代周期都有需求分析过程，这个时候是也比较小，也更容易把需求描述清楚。

2）用户描述

由于敏捷开发小组要关注完成和交付具有用户价值的功能，而不是完成孤立的任务（把任务最终组合成有用户价值的功能）。问题是冗长的用户说明很难快速看清问题，一种比较好的表述需求的轻量级技术，是把一个用例先用一个简短的说明来表达，我们称之为用户描述，这是从客户的角度出发对功能的一个简短描述。一般的表达方式是：“作为（用户类型），我们希望可以（能力）以便（业务价值）”，例如：“作为购书者，我们希望可以根据 ISBN 找到一本书，以便更快找到正确的书。”

用户描述是轻量级的，并不需要一开始就把它们全部收集和记录下来，或者编写复杂的需求说明。不管怎么说，收集用户描述应该相对比较容易，每个描述一个卡片，产品所有者和开发人员都比较容易针对这样的简短描述进行交流。

一旦确定一次迭代需要完成哪些描述，就需要编写正规的需求文档，由于此时的视野比较小，处理这样的问题一般是没有太大困难的。

3）用户描述和主题

有时，一组相关的用户描述被结合在一起，比如把这些描述卡片用回形针别在一起，当作一个实体来看，我们常常称之为主题（theme）。还需要注意到的是，用户描述的主题往往构成了架构的单元。

一旦确定一次迭代需要完成哪些描述，就需要编写正规的需求文档，由于此时的视野比较小，处理这样的问题一般是没有太大困难的。

4）确定主题的优先级

即使我们有时间，也很少会有足够的时间来做所有的事情，所以需要确定优先级。尽管确定优先级的责任由整个开发小组共同承担，但成果由产品所有者享用。遗憾的是，估计少量的或者单个用户描述的价值是比较困难的，所以我们需要把若干用户描述或功能聚集到一些主题当中。然后根据用户描述和主题之间的相互关系，来确定它们的优先级。选择主题的时候，应该让它们能分别独立的定义一组对用户有价值的功能。

优先级确定看起来是比较简单的事，但是实践表明不少组织经常发生难以确定优先级的问题，这就需要有一些原则和方法，本课程会有专门的章节来讨论优先级确定的一些原则。

5）用户级别的用例

系统的价值是通过他对用户级别的目标的支持来判断的，构建系统的的人头脑中可能是一个更高层次的目标，用户目标只是实现更高层次的目标中的一部分，但是高层目标只有通过具体的用户目标来实现。初期的时候，应该尽可能收集基于用户描述表达的目标集，我们并不需要去真正的编写用例，后期则需要详细地描述它们。

3，子功能目标层次

子功能层次的目标指的是那些实现用户目标时候可能会被用到的目标，只有当迫不得已的时候才会包含这些子功能用例，例如可读性方面的要求，或者有许多其它目标用到它们。

子功能层次的目标往往形成用例的结构化，应该注意到，即使在最深层次的子功能，在系统之外也可能会有一个主参与者，但你不需要在这个方面多花精力，除非偶尔把它作为内部设计的议题来讨论，或者看上去缺少一个参与者的时候。更多的情况是，子功能会引用它的高层用例相同的主参与者。

4.2 静态用例模型

一、用例及用例图产生的技术背景概述

1）为什么会需要用例

在软件系统的分析与设计中，必须要了解并准确描述用户的功能需求，以便于确定建立的对象。很长时间以来，无论是**传统的软件开发方法还是面向对象的开发方法，都采用自然语言（如中文）来描述对系统的需求**

其缺点是没有统一的格式，缺乏描述的形式化，随意性较大，常常产生理解上的含混及不确定性。在这种背景下，有关专家提出了用例（Use Case）的概念及其图形表示方法——用例图，

这种方法很快得到广泛的应用。

所谓用例，实际上是描述参与者使用系统的场景。

用例是一种被广泛使用的发现和记录需求的（特别是功能性需求）的机制。写出用例，可以认为是使用系统时的情节，也就是写出实际语境中的需求。

2）用例模型的主要作用

你可以应用 UML 用例模型来开发一个精确的模型来表示系统的需求，然后以这些用例为基础来推动系统开发的其它方面。

用例的作用就好象是项链上的一条线，它将所有的珍珠绑定在一起。用例在最终的用户和系统需求之间建立起一座桥。它们可用来在功能需求和系统实现本身之间进行回溯。

用例也可以作为一个连接点，连接到一个详细的说明需求细节的用例文档。

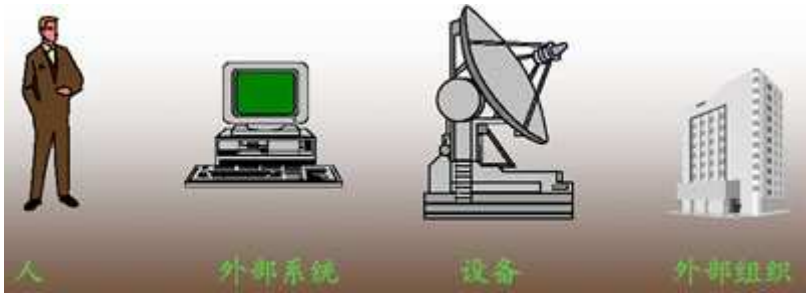
二、用例模型的基本组成

用例模型的基本组成部件包括：参与者、用例和系统。

1，参与者（Actor）

参与者（actor）：具有行为能力的事务，可以是个人（由其扮演的角色来识别），计算机系统，或者组织。参与者表示系统用户能扮演的角色(role)，这些参与者可能有三大类：系统用户、与所建系统交互的其他系统、时间。

- 系统用户：使用本系统的人
- 其他系统：可能是其他的计算机或者一些硬件或者甚至是其它软件系统
- 时间：时间作为参与者时，经过一定时间触发系统的某个事件。例如，ATM 机可能每天午夜运行一些协调处理。由于事件不在我们的控制之内，因此是个角色。



1） 参与者示例说明

设想一个项目，参与者主要有用户和系统统管理员，而管理员使用控制面板对系统和用户管理，也就是进行系统设置，管理用户、用户组、权限，查看系统访问日志及用户使用情况等的统计信息。

假如一个学校课程管理系统，有三个参与者在不同的应用中互动。这三个参与者分别是学生，讲师以及系统管理者。

学生参与者：使用了系统中浏览课程以及注册课程的功能；

系统管理者参与者：负责管理注册的学员，编排课程以及确认课程；

讲师参与者：主导课程的参与者，他可以浏览，开办以及移除课程(当然，必须是这个讲师自己的课程)

2）参与者图示

在 UML 中参与者的图示

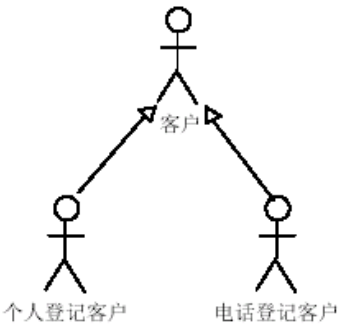
3)参与者的目标

我们必须表达清楚每个参与者的目标，用例的目的是使这些目标得以实现，如下是一个目标列表。

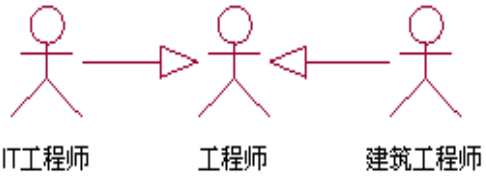
参与者	目标
收银员	处理销售
	处理租赁
	处理返回值
	入款
	出款
经理	启动
	关机
系统管理员	添加用户
	修改用户
	删除用户
	安全管理
	系统表管理
销售活动分析系统	分析销售活动和销售表现

4）参与者之间的关系---泛化（特化或者继承）关系

由于参与者是类，所以它拥有与类相同的继承关系描述，其 UML 的图示是用带空心三角形（箭头）的直线表示。在特殊的参与者中之需要给出其特殊的成员定义。

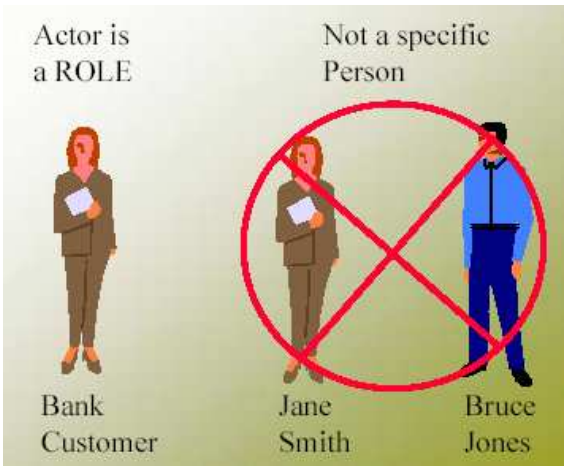


角色之间的通用化关系示例



4）所要注意的问题

- **参与者主要是指角色而非具体的个人**



- **用户与参与者之间的关系**
  - ✓ 一个用户可以抽象为多个参与者，如：张三即可以是网上书店的读者，也可以是管理员
  - ✓ 一个参与者可以包含多个用户，如：网上书店的读者可以是张三和李四

5）发现参与者对提供用例是非常有用的

通过实践，我们发现参与者对提供用例是非常有用的。因为面对一个大系统,要列出用例清单常常是十分困难。这时可先列出参与者清单,再对每个参与者列出它的用例,问题就会变得容易很多。

6）如何获得系统中的参与者

获取用例首先要找出系统的参与者。可以通过用户回答一些问题的答案来识别参与者。以下问题可供参考:

- 谁使用系统的主要功能(主要使用者)。
- 谁需要系统支持他们的日常工作。
- 谁来维护、管理使系统正常工作(辅助使用者)。
- 系统需要操纵哪些硬件。
- 系统需要与哪些其它系统交互,包含其它计算机系统和其它应用程序。
- 对系统产生的结果感兴趣的人或事物。

2，用例（UseCase）及其定义

- 用例是关于单个活动者在与系统对话中所执行的处理行为的陈述序列（Jacobson）。它表达了系统的**功能**和所提供的**服务**。
- 它描述了活动者给系统特定的刺激时系统的活动,**是活动者通过系统完成一个过程时出现的一组事件，最终以实现一种功能**。
- 通常，用例侧重于功能，但不重点描述该功能的实现细节；同时用例的大小划分一般以事件流在 10 个步骤左右为好。
- 所有的用例必须始于参与者（Actor），而且有些用例也结束于参与者。

1）用例的分类

业务用例（Business Use Case）

指系统提供的业务功能与参与者的交互，表现问题领域中各实体间的联系和业务往来活动（如果某个用例的范围包含了人以及由人组成的团队、部门、组织的活动，那么这个用例必然是业务用例）。

系统用例（System Use Case）

指参与者与系统的交互，它表现了系统的功能需求和动态行为（如果仅仅是一些软件、硬件、机电设备或由它们组成的系统，并不涉及到人的业务活动，那么该用例是系统用例）。

注意：用例确定的只是与用户交流的目的，而不是交流的手段

因为，客户并不需要了解执行者、用例这些概念。用例能告诉开发团队“去向客户了解什么”（目的），不能告诉你如何向客户去了解（手段）；

获得用例的手段可以有很多种，文档研究、问卷调查、访谈、观察、研究竞争对手、开会、原型、场景演示…，使用用例思维来指导这些交流手段，会使交流更有目的，更加高效。因为以场景方式表达的需求本来就比一条条列出的需求要便于交流。

2）如何确定系统中的用例

如何寻找项目中的用例？说法不一，见仁见智！标识和确定系统中的用例的一般做法是，针对问题领域中的事件流（业务工作流和系统运作的流程）编制场景（一个场景是描述行为的一个特定的动作序列），然后根据场景（可以将场景看作用例的实例）确定用例。一个系统中的用例的种类大致如下：

- 系统的开始和停止的用例。
- 系统维护的用例。
- 维护系统中存储的数据的用例。
- 修改系统行为的功能的用例。
- 系统中代表业务功能的用例

一旦获取了参与者,就可以对每个参与者提出问题以获取用例，以下问题可供参考:

- 参与者要求系统提供哪些功能(参与者需要做什么)?
- 参与者需要读、产生、删除、修改或存储的信息有哪些类型。
- 必须提醒参与者的系统事件有哪些?或者参与者必须提醒系统的事件有哪些?怎样把这些事件表示成用例中的功能?
- 为了完整地描述用例,还需要知道参与者的某些典型功能能否被系统自动实现?

尽管并不存在一个完美的过程来开发这个模型，但还是可以推荐一个简单而有效的步骤。当然，这些步骤并不是在开发的同一点发生，开发的迭代性决定了随着时间的变化，这些步骤会重复地进行，这个步骤可以分成五步。

第一步：确定和描述参与者

构建用例的第一步是确定所有与系统交互的参与者，从系统的上下文图中，我们可找到大部分参与者。在这一步中，我们可以考虑以下问题：

- 谁使用系统？
- 谁从系统得到消息？
- 谁向系统提供消息？
- 公司在什么地方使用系统？
- 谁支持和维护系统？
- 其它还有什么系统使用这个系统？

第二步：确定用例，写一个简短的描述

一旦决定了参与者，下一步就是决定参与者为了完成他们的工作使用的用例。我们可以通过按顺序界定每个参与者的特定目标来做这件事情：

- 参与者用系统做什么？
- 参与者是否在系统中创建、存储、改变、删除或者读取数据？
- 参与者是否需要通知系统相关的外部事件或改变？
- 是否需要通知参与者系统内发生的事情？

要仔细思考用例的名字，通常用例是一个短语，以一个动词开始，说明参与者拿用例做了什么。有了名字，还需要提供一个简要地描述，概要的说明这个用例的工作。这个描述与前面提到的用户描述（use story）是一样的，可以写成：“作为（参与者），我们希望可以（能力）以便（用例结果）”。把这些用户描述列成表，就可以在前期进行优先级的排序等其它工作。也可以把若干用户描述合并成主题，这样就可以进一步首先从广义上组织需求信息了。

这种简短的描述也便于与用户交流。

第三步：确定参与者和用例的关系

尽管我们注意到只有一个参与者能够发起一个用例，但很多用例可能有多个参与者参与。在过程的这一步，要分析每个用例，看看有哪些参与者与它交互？审查每个参与者预期的行为，以验证参与者是否参与了所有必要的用例，是否达到了想要的结果。这个过程可能非常复杂，需要团队成员一起来讨论，很快就会有大量的用例和参与者，通过用例图，可以使每个人都很好的理解系统。

第四步：简略描述单个用例

下一步是简略描述整个用例，从而在更深层的意义上理解系统的行为。所描述的包括基本流和扩展流。基本流一般只有一个，称为主事件流，扩展流（也称备选事件流）主要指的是正常的分支与异常事件。为了发现这些事件，需要问一下这些问题：

基本流：

- 什么事件发起该用例？
- 用例如何结束？
- 用例如何重复某些行为？

扩展流：

- 用例是否有可选项？
- 可能发生什么偶然事件？
- 可能发生什么变数？
- 什么可能出错？
- 什么不可能发生？
- 可能把什么资源锁住？

这个阶段的描述不需要很复杂，主要在宏观上表达问题，更细腻的描述可以在细化系统定义的时候完成。

第五步：细化用例

在前期是不需要做更细化的描述的，但是在生命周期的某个阶段，特别是在选定了某些用户描述开始迭代周期的时候，就应该把用例细化到最后一级的细节程度。这时有许多因素需要考虑，而每一种因素都可能在细化过程中起到一定的作用。

- **所有扩展流，包括异常事件**：这是一种相当直接的方法找到用例的扩展流。如果再开发中提出“如果……那么……”这类问题，扩展流中就需要加以全面考虑。所有异常都必须在用例中记录下来，否则系统不可能按照预期运行。
- **前置和后置条件**：细化的开始是确认系统控制行为的状态信息。这个状态信息是在用例的前置条件和后置条件中表示的。前置条件描述了用例开始的时候必须为真的事情，后置条件描述了用例离开的时候持续状态。

在上面的讨论中，我们把系统特性定义在较高的抽象级别上，并且在这个级别上对整个项目进行规划。这样做得好处是：

- 可以更多关注系统特性以及它如何体现用户需要，更好的理解系统的形状和形式。
- 可以对系统的完整性、一致性及其对环境的适应性进行评估。
- 在继续大量投入之前，可以利用这些信息决定可行性并管理系统的范围。

- 便于在选择迭代内容的时候，能够有一个整体的图像和便捷的用户描述表格。

此外，停留在较高的抽象层次，还有助于我们不至于过早的作出需求决策，我们可以很容易的加入自己的观点和价值观，也比较容易实现需求变更，在迭代开始的时候发生变更往往是不可避免的。

为了确保设计和编码产生完全符合需要的系统，就需要对选中的用例进行细化分析，这也会产生更多的需求管理信息，必须有效的组织和处理他们。在传统上意义上，需求规格说明种功能性和非功能性需求来自于对用例的细化描述。

在敏捷过程中，在选择了高优先级的用户描述，进入迭代周期以后，也需要对用例进行细化描述。具体采用那种方式要根据项目过程的需要，但不管哪种方法，对用户描述进行细化，也就是细化系统定义的能力是必需的。

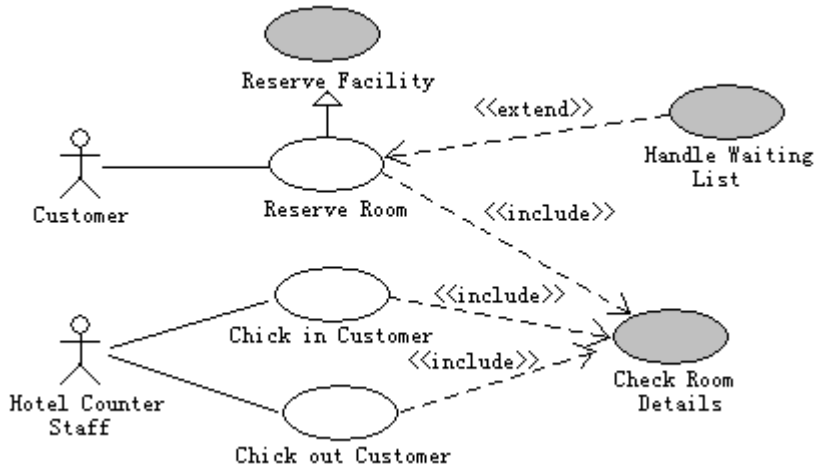
3）用例的命名

每个用例应有唯一的名称，同时应从客户的角度来命名，以帮助确定项目范围。用例的名称还应独立于实现方法。要避免使用例与特定实现方法或者语言平台相联系的短语。用例通常用动词 + 名词短语来命名

三、用例之间的关系

主要体现在纵向方面的层次化关系和横向方面的用例的关联性。用例之间的关系包括：包含、扩展与泛化，对例通过扩展、包含、泛化等关系作为对关注点之间进行建模的手段，称之为用例的结构化，比如如下图所示的情况，下面讨论这种结构化用例场景的描述规则。在用例建模的时候，不同用例之间的关系如下：

- **包含（include）**：一个用例的实现使用另一个用例的实现。其图形表示方法为在用例图上用一条从基本用例指向包含用例的虚箭线表示，并在线上标注购造型<<include>>：
- **泛化（generalization）**：一个用例的实现从另一个抽象的用例继承。
- **扩展（extend）**：扩展关联的基本含义与泛化关联类似，但是对于扩展用例有更多的规则，即基本的用例必须声明若干新的规则---扩展点（Extension Points），扩展用例只能在这些扩展点上增加新的行为并且基本用例不需要了解扩展用例的如何细节。它们之间存在着扩展关系。如果特定的条件发生，扩展用例的行为才能执行。其图形表示同样用虚箭线表示，并在线上标注购造型<<extend>>：



1，包含（include）关系

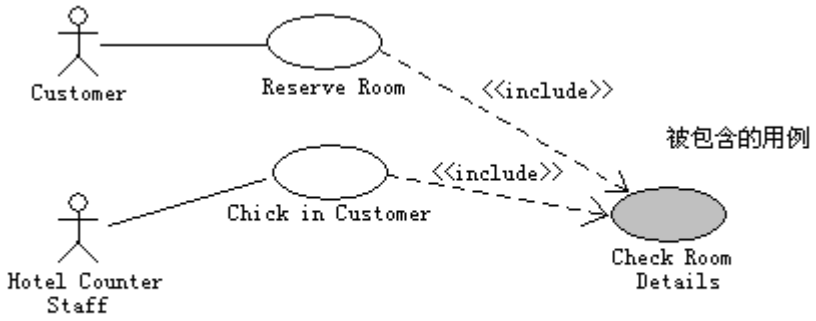
为了确保产品设计的品质，产品开发的可追踪性与回朔性要好，因此在用例设计的时候，我

们希望关注点相互独立，其重要性也不能相互比较，这就是所谓对等关注点。例如在酒店管理系统中，预订房间（Reserve Room）、登记入住（Check In Customer）、结账离开（Check Out Customer）都是对等关注点。

随着用例的细化和精化，团队会发现某些用户行为会在多个地方重复出现，事实上，很大一部分系统功能在多个地方重复出现时有可能的，比如输入口令进行用户确认。显然对相同的用户行为出现冗余的文档是不合适的，这就可以使用包含关系。包含的目的是在其它用例中引入用例。

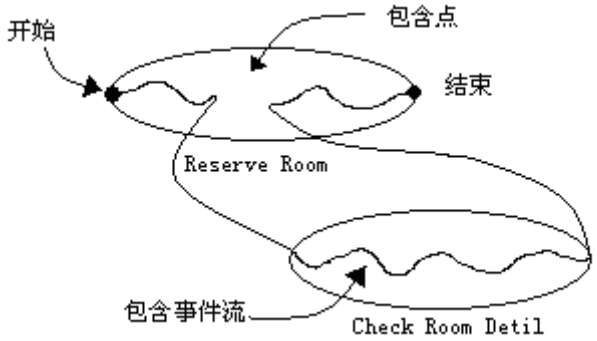
作为分析师，我们应该注意到问题越是前期发现和解决，总的风险就越小。所以设计师希望从分析的时候就做到使关注点相互分离。在分析的时候，对于不同用例的相似步骤，可以把这些公共事件抽取出来，放在被包含的用例中，其它用例可以引用这些被包含用例中的事件流。

例如，“预定房间”和“登记入住”都需要参与者核对房间的可用性、以及查询有没有可用的房间等，这可以增加一个“核对房间清单（Check Room Details）”的包含用例。



用例“预定房间”包含了用例“核对房间详情”

它的执行过程如下。、



注意，大部分对子用例的调用是以包含形式表现，这使得开发团队很容易掌握，因为这类似于软件中的子程序，如果用的合理，包含关系能简化开发和维护活动，也是很重要的一种结构。

2，扩展关联（extend）

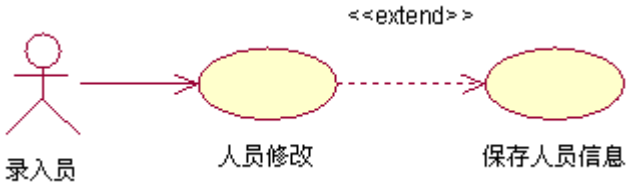
扩展关联的基本含义与泛化关联类似，但是对于扩展用例有更多的规则，即基本的用例必须声明若干新的规则---扩展点（Extension Points），扩展用例只能在这些扩展点上增加新的行为并且基本用例不需要了解扩展用例的如何细节。

例如，**保存人员信息**用例可以是删除人员信息及新增和修改人员信息用例的扩展，它们之间存在着扩展关系。如果特定的条件发生，扩展用例的行为才能执行。

因此，扩展用例为处理异常或者构建灵活的系统框架提供了一种有效的方法，同时采用泛化关联和扩展关联都可以分解一个用例，其一侧重于问题的特殊性，而另一个则侧重于问题的延续性。它们都能够便于分析设计人员简化复杂的系统。

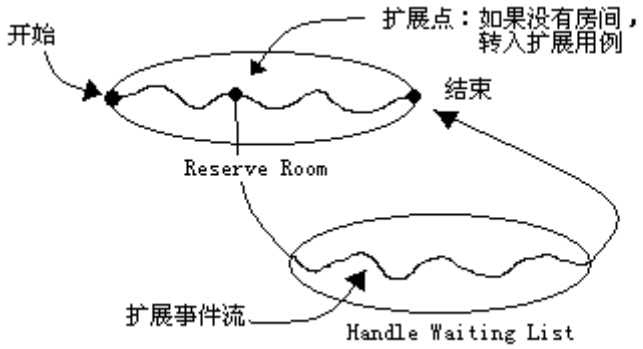
其图形表示方法为在用例图上用一条从基本用例指向扩展用例的虚箭线表示，并在线上标注

购造型<<extend>>：



例如，酒店管理系统中有一个功能“待分配房间的预订人等候名单”，如果没有房间，系统就会把客户放进这个“等候名单（Waiting list）”，因此，这个“Waiting list”就是“预订房间（Reserve Room）”的扩展。把扩展分离出来，可以使问题容易理解，这就就不至于被过多的问题所纠缠。

下图表示了这个扩展用例的存在发生了什么。



使用扩展用例的理由有三条：

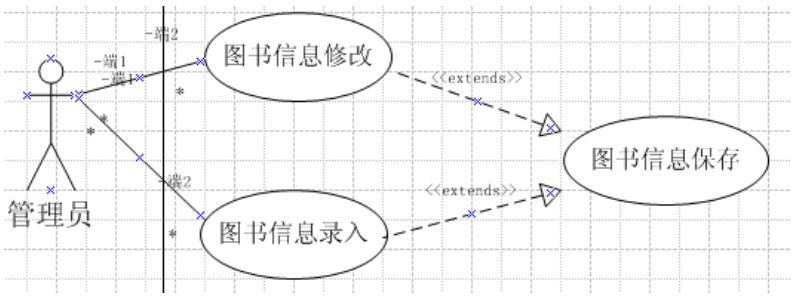
- 作为一个实体，扩展用例能够简化用例的维护，同时允许团队关注并细化扩展的功能，而不需要重读基本用例本身。
- 把扩展看成是用例开发，扩展点可以在基本用例中给出作为“通向未来特性”的途径。
- 扩展用例可以表示随意性的行为，而不是一个新的基本或扩展流程。

最后一条往往是最有用的。还需要注意需求的变更，标识出哪些需求将来可能变更，尽早提出一些解决办法，比如把可能变更的功能独立出来（至少建议这样做），这样就可以避免将来的需求变更带来不利的影响，所以，对于产品用例的考虑，我们可以在早期注意以下几个方面：

- 用例对应着功能包，所以用例的大小要合适。
- 注意到缠绕状态，标识出并且提出建议方案。
- 研究并且标识出易变性，把易变的功能独立起来。

从这些角度说，产品用例是在业务用例的基础上更进一步的思考。

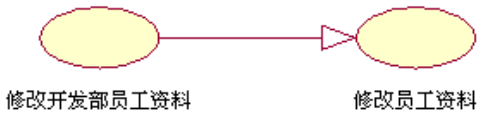
下面还列出了几个用例之间的扩展关系，例如，保存图书信息用例可以是图书信息修改和图书信息录入用例的扩展。



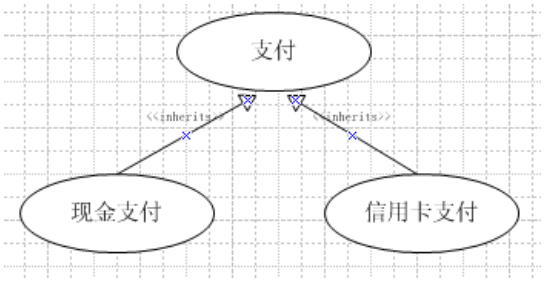
3，泛化关联

泛化关联代表一般与特殊的关系，它充分体现了面向对象的继承性：子类具有父类的所有属

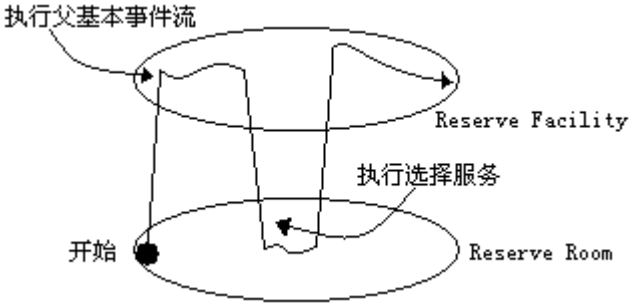
性，还可以拥有自己的属性特点及行为。泛化关联包括用例之间及活动着之间的关联关系。例如，修改员工资料和修改开发部员工资料就是用例的泛化关联。泛化关联用空心三角箭头的实线表示：其方向从特殊指向一般。



本项目中的几个用例之间的泛化关系



在“预定服务”用例中有一个抽象子事件流，所以“预定房间”是一个抽象用例，而“选择服务”的事件实现在“预定房间”用例中。事件流执行的规则如下：实例化首先出现在子用例中，沿着基本事件流运行，如果子用例没有定义基本事件流，则沿着父基本事件流执行，上面的例子由于没有定义子基本事件流，所以沿着父基本事件流运行。在运行到“选择一项服务”的时候，执行子用例定义的“选择服务”子用例，如下图所示。



4.3 UML 用例图的具体应用实例

一、确定项目中系统中的角色（参与者）的种类

在项目的设计中主要是要考虑有多少种不同类型的用户？都是哪几种类型的用户，用户的角色如何定义；用户的访问权限如何分配等。

- **在网上书店应用中**

根据应用的要求，**可能会有**

- ✓ 图书信息的浏览者(Visitor，没有进行购买行为的用户)

- ✓ 图书的购买者（读者用户）
- ✓ 收银员（如财务人员）
- ✓ 管理员和超级管理员(Administrator，系统管理员)。

系统管理员负责软件的初始化工作、权限分配、图书购入维护等工作

消费者使用系统查询系统查找需要的信息；

收银员使用销售系统根据消费者性质（会员、非会员）负责收款

- **在网上银行应用中**

根据应用的要求，可能会有**个人用户、企业用户、银行职员、财务人员以及管理员和超级管理员**(Administrator，系统管理员)。

- **确定角色的权限**

在设计的时候我们也已经把这些角色与相应的一些操作绑定在一起。如：

Publisher 拥有 Publish_Operation + Modify_Operation +Delete Operation

（出版者拥有：发布业务+修改业务+删除业务）

Visitor 拥有 Visit_Operation,

（访问者拥有：访问业务）

Reader 拥有 Visit_Operation + change ,

（阅读者拥有：访问业务+改变业务）

Manager 拥有 Visit Money Operation +Sum Money Operation

（管理者拥有：访问货币业务+合计货币业务）

Administrator 负责 ：

Create_User_Operation+

Delete_User_Operation+

Assign_Permission_Operation+

Deassign_Permission_Operation+

Assign_Role_Operation+

Deassign_Role_Operation

（系统管理员拥有：创建用户+删除用户+指派权限+重新分派权限+指派角色+重新分派角色的业务）

二、设计出项目系统中的各个模块的用例（UseCase）

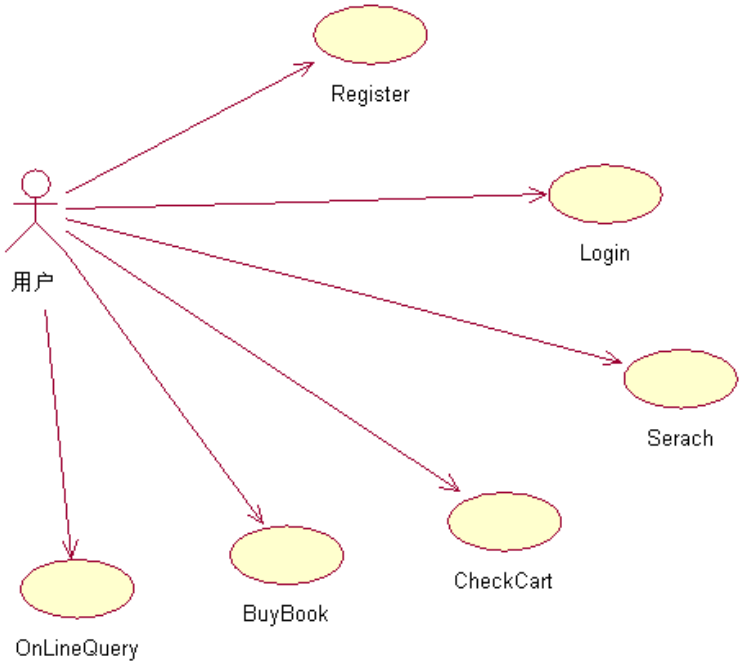
1）确定系统边界线

- ✓ 通过使用系统边界线矩形框来框定系统中的各个用例同时也通过它能够很清楚地划定内外部事物，因为在系统中所有的用例都应该放在矩形的内部，而在外部是所有该系统的活动者，并且它们被线连到用例。
- ✓ 在系统边界线内的每一件事物都是系统的一部分，而在系统边界线外的每一件事物都是系统的外部。

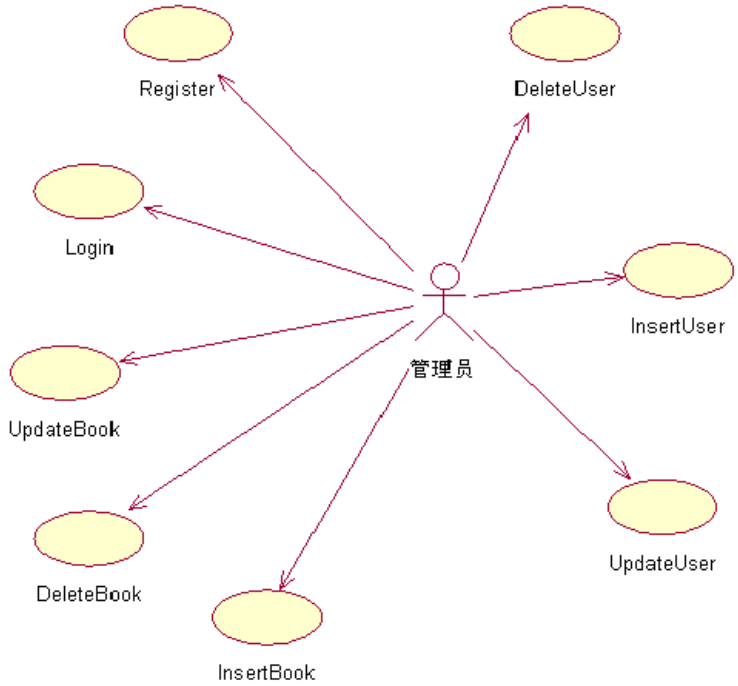
2） 用例图的主要作用

在需求分析阶段,可以利用用例图来捕获用户需求。通过用例建模，描述对系统感兴趣的外部角色及其对系统(用例)的功能要求。

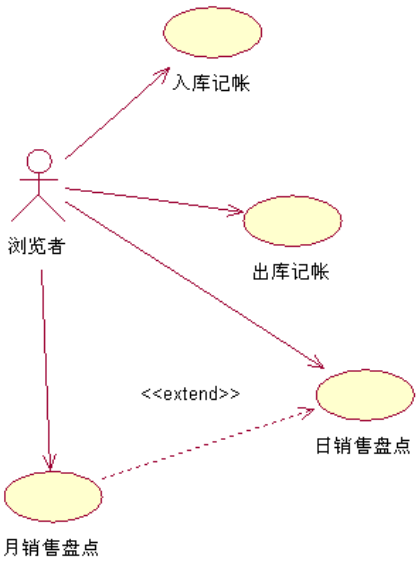
3） 网上书店的“网上书店前台用例图”如下



4） 本系统中的网上书店中后台管理员对用户进行管理的主要用例

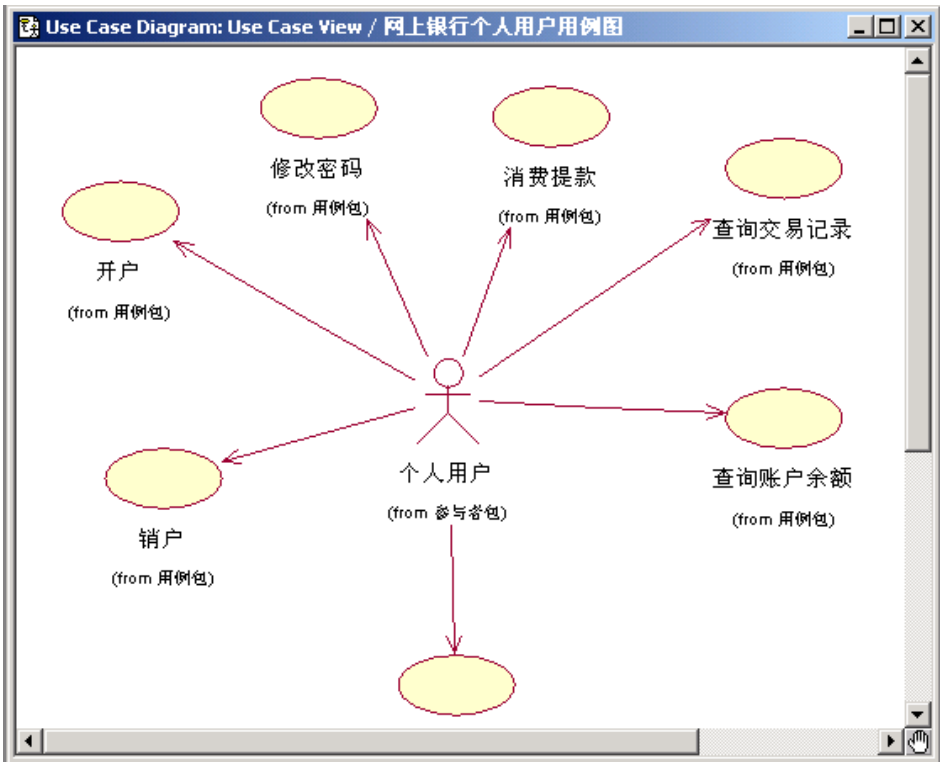


5） 网上书店中后台“收银员进行财务管理”的主要用例



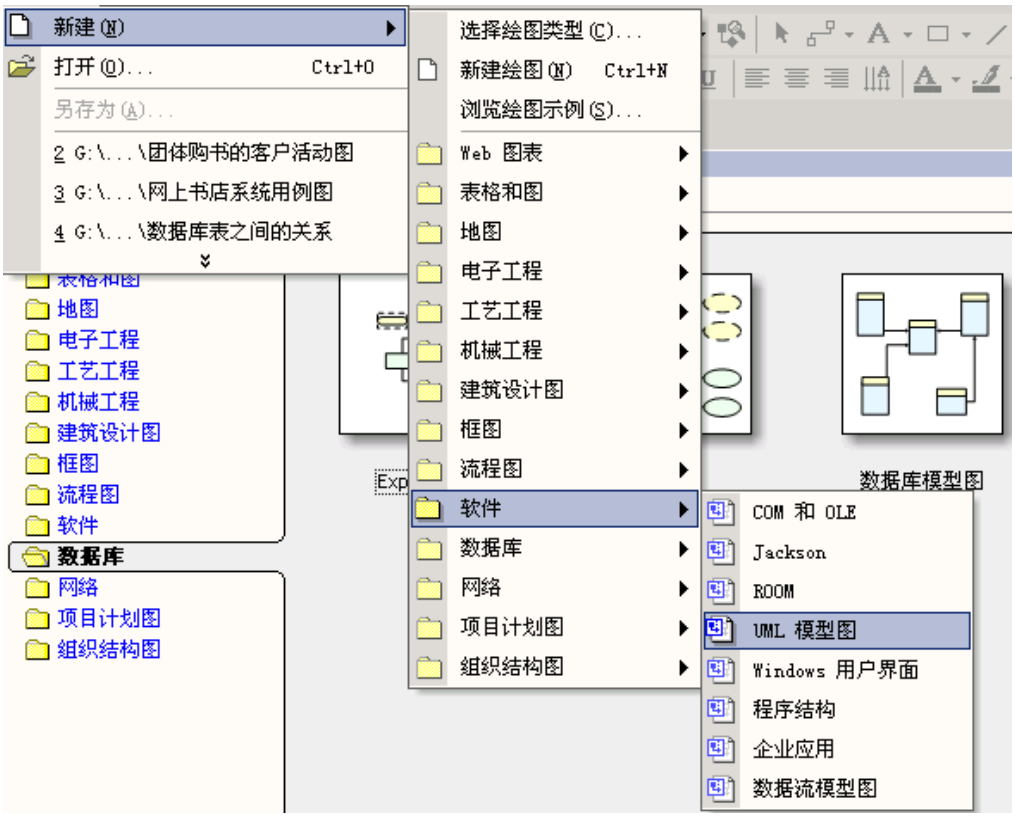
6） 网上银行的主要用例如下

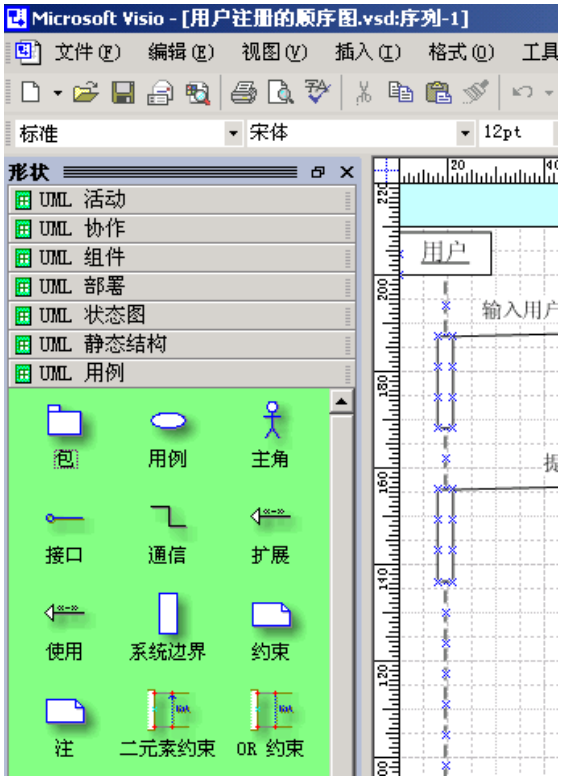
客户可以连接到 EJB 服务器并对储蓄账户进行各种操作，如开户、销户、存款、提款、查询交易记录、查询账户余额以及修改密码、转帐交易等。其用例图如下所示：



三、在 Visio 中创建项目中的各个用例

1、Visio 中对 UML 的支持





2、在 Visio 中创建用例的方法

（1）在"文件"菜单上，依次指向"新建"、"软件"，然后单击"UML 模型图"；在树视图中，右击要包含用例图的包或子系统，再指向"新建"，然后单击"用例图"。这会出现一个空白页，并

且"UML 用例"模具将成为最顶部的模具。工作区将"用例"显示为水印。并将一个表示该图表的图标添加到树视图上。

（2）将"系统边界（可确定系统内部和外部之间的界限）"形状拖到绘图页上，双击该形状，然后输入系统的新名称，或按 DELETE 键删除现有名称。在绘图页上的形状外单击。

（3）从"用例"模具中将"用例"形状拖出并放在系统边界上，然后将"主角"形状拖到系统边界之外。

3、指出主角和用例之间的关系

（1）在用例图中，将"通信（定义了主角以何种方式参与到用例中来）"形状拖到绘图页上；将"通信"形状的一个端点粘附到"主角"形状的连接点上。将另一个端点粘附到"用例"形状的连接点上。

（2）如果您要添加箭头来表示信息流，请执行以下操作之一：

双击"通信"形状，然后在"关联"下单击您要编辑的一端，然后单击"属性"。

在"关联端"类别中，选中"IsNavigable"，单击"确定"，然后再单击"确定"。

右击"通信"形状，单击"形状显示选项"。在"端选项"下选择"端的导向性"，然后单击"确定"。

4、指出两个用例之间的使用关系

（1）在用例图中，将"使用（两个用例间的使用关系----指出用例 A 也可以包含用例 B 指定的行为）"关系形状拖到绘图页上；将"使用"端点（不带箭头）粘附到使用其他用例方式的"用例"形状的连接点上。

（2）将"使用"端点（带有箭头）粘附到正使用的用例的连接点上；双击"使用"形状，打开"UML 归纳属性"对话框。添加属性值，然后单击"确定"。

5、指出两个用例之间的扩展关系

（1）在用例图中，将"扩展"形状拖到绘图页上，将不带箭头的"扩展"端点 粘附到提供扩展的用例的连接点**用例的事件流** 上。

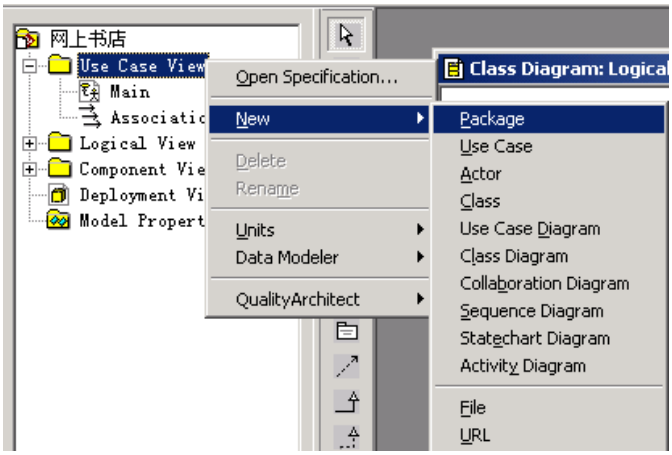
（2）将带有箭头的"扩展"端点粘附到基础用例的连接点上，双击"扩展"形状，打开"UML 归纳属性"对话框。添加属性值，然后单击"确定"。

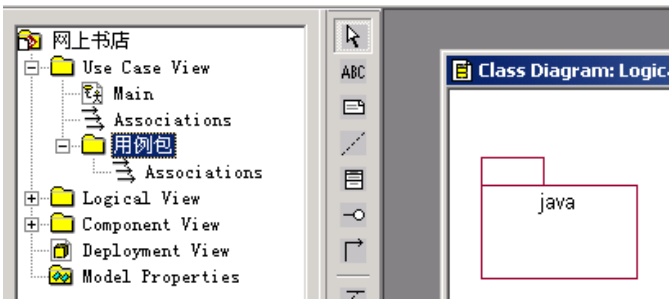
四、在 Rose 中创建项目中的各个用例

1、新建网上书店中的用例图

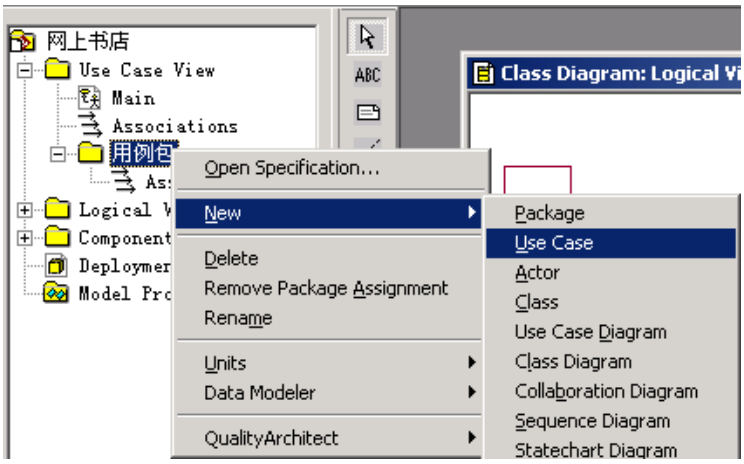
（1）新建各个用例

- **新建用例所在的包，包的名称为“用例包”**



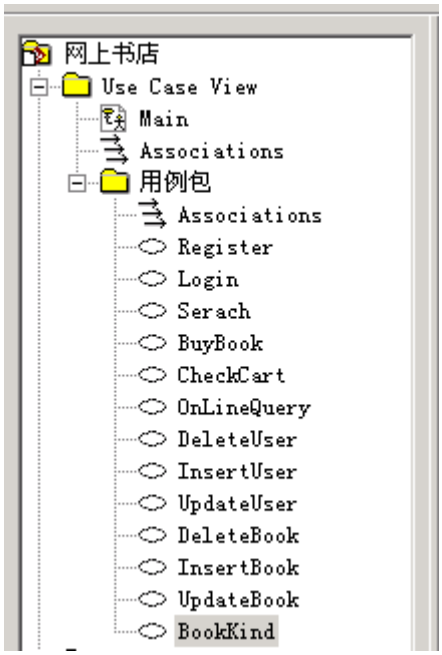


- **在该包中增加本项目中的用例**



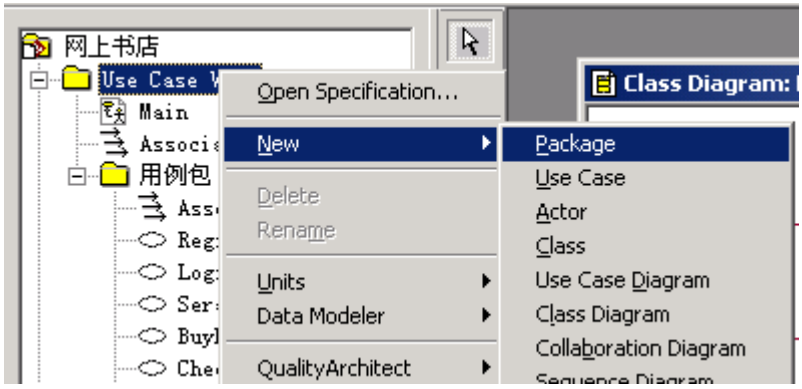
在 Rose 中分别添加各个用例，如：Register、Login、Serach、BuyBook、CheckCart、OnLineQuery、DeleteUser、InsertUser、UpdateUser、DeleteBook、InsertBook、UpdateBook、BookKind 等。

注意：上面的各个用例只涉及用户购买和管理员进行图书信息和用户管理的用例。



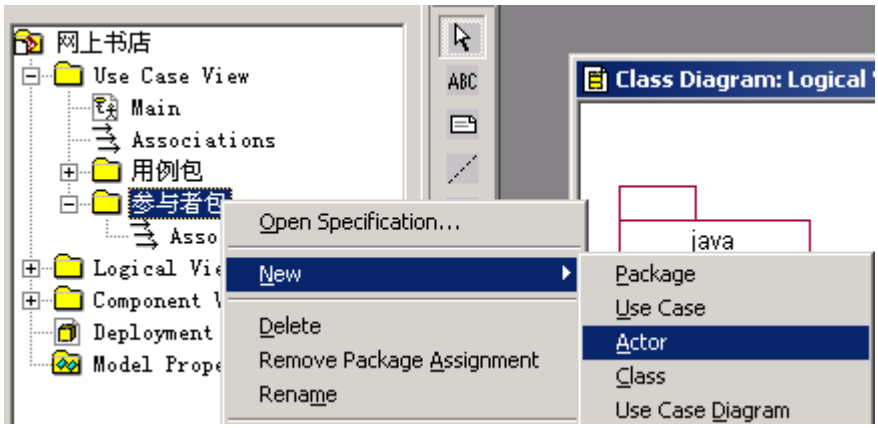
（2）新建参与者

- **新建参与者所在的包，包名称为“参与者包”**



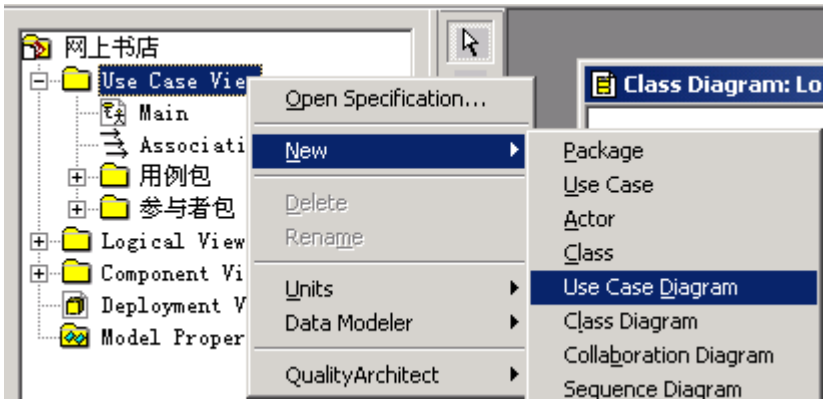
- **在该包中增加本项目中所需要的参与者**

在 Rose 中分别添加各个参与者，如：图书信息的浏览者(Visitor，没有进行购买行为的用户)、图书的购买者（读者用户）、收银员（如财务人员）和管理员、超级管理员(Administrator，系统管理员)等



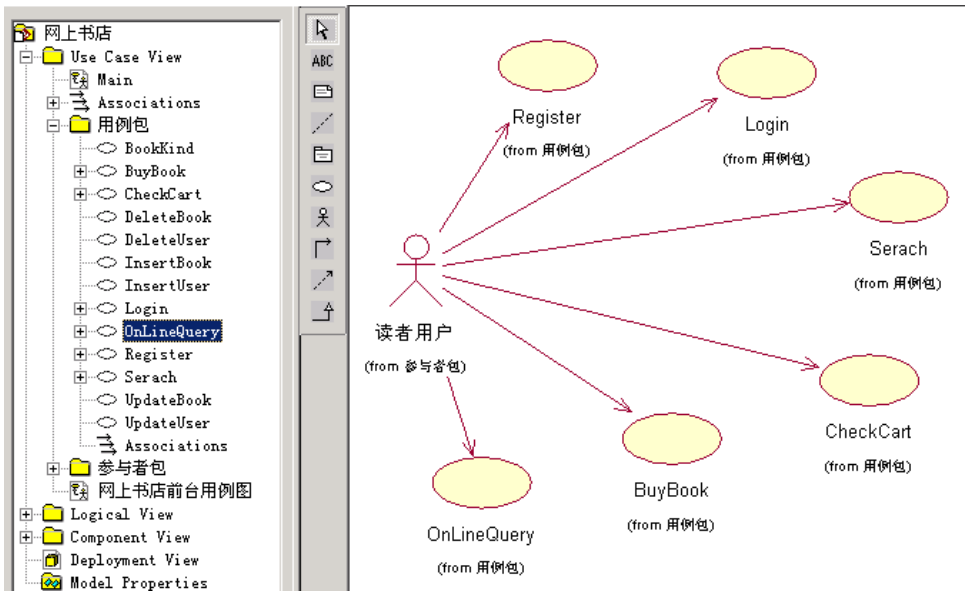
（3）新建“网上书店前台用例图”

- 右击以新建用例图

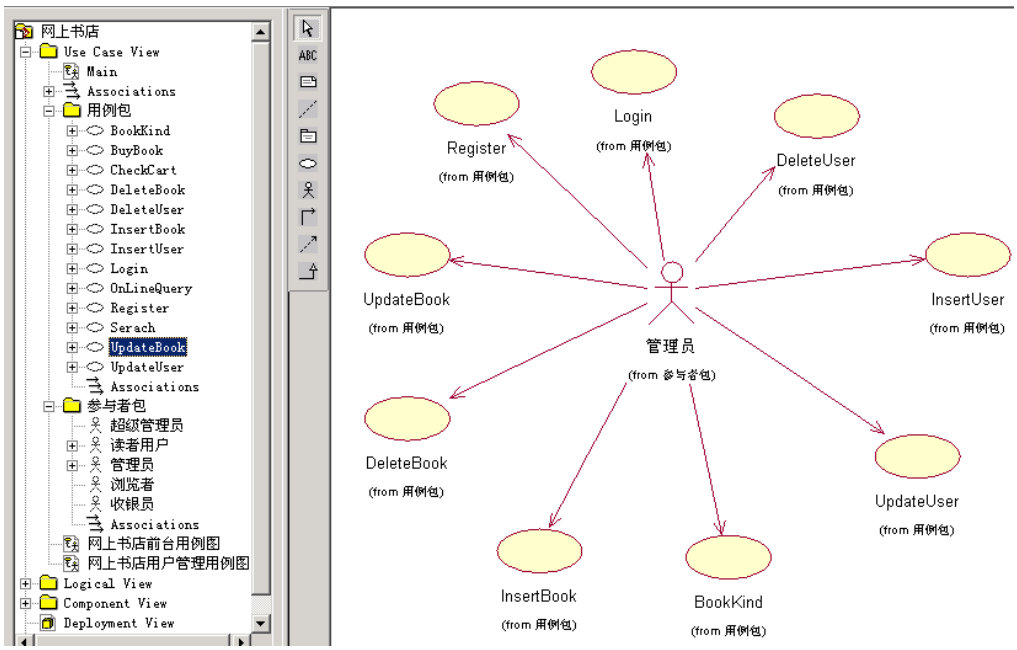


- 名称为“**网上书店前台用例图**”

- 在用例图中将前面所创建出的各个用例和参与者拖到用例图中，并点击“单向关联”箭头关联参与者和各个用例



（4）新建“网上书店用户管理用例图”

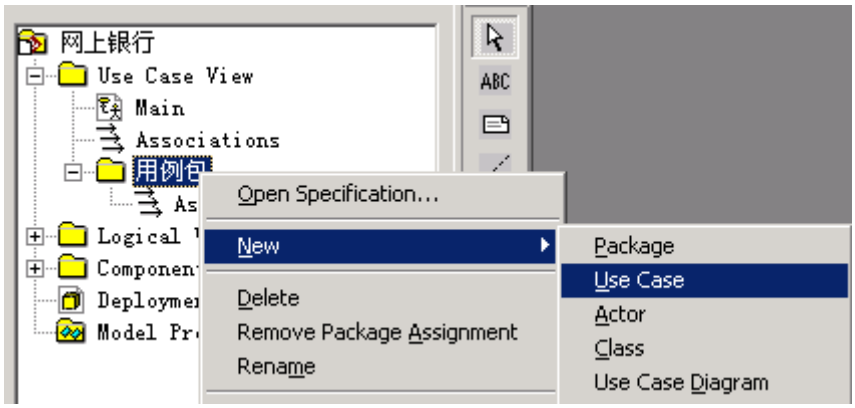


2、新建本项目中的网上银行中的用例图

（1）在“网上银行”模型文件中新建各个用例

- 新建用例的包，包的名称为“用例包”

- 在该包中新建各个用例

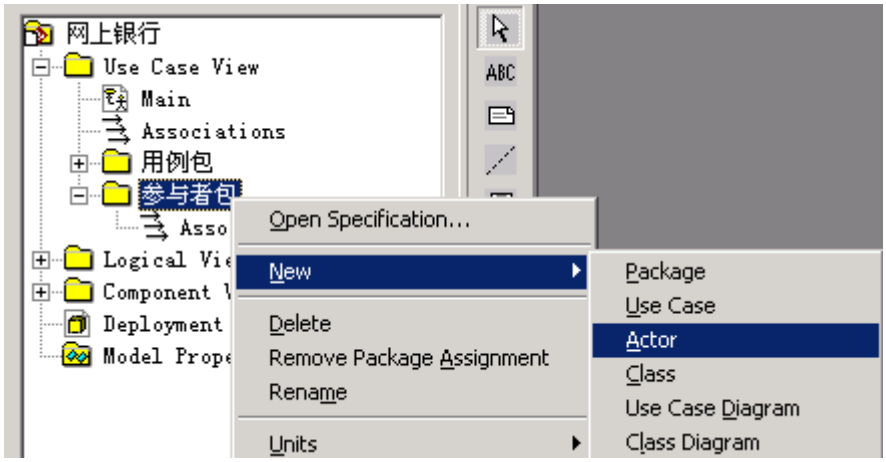


- 在 Rose 中分别添加各个用例，如：开户、消费提款、查询交易记录、查询账户余额以及修改密码、转帐交易、销户等。

（2）新建参与者

- **新建参与者所在的包，包名称为“参与者包”**

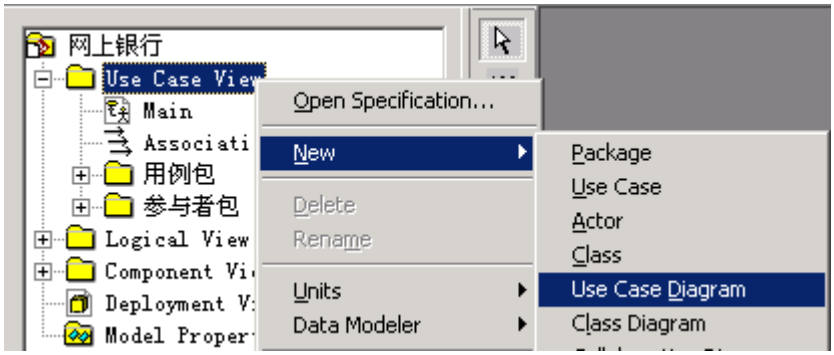
- **在该包中增加参与者**



在 Rose 中分别添加各个参与者，如：**个人用户、企业用户、银行职员、财务人员以及管理员和超级管理员**

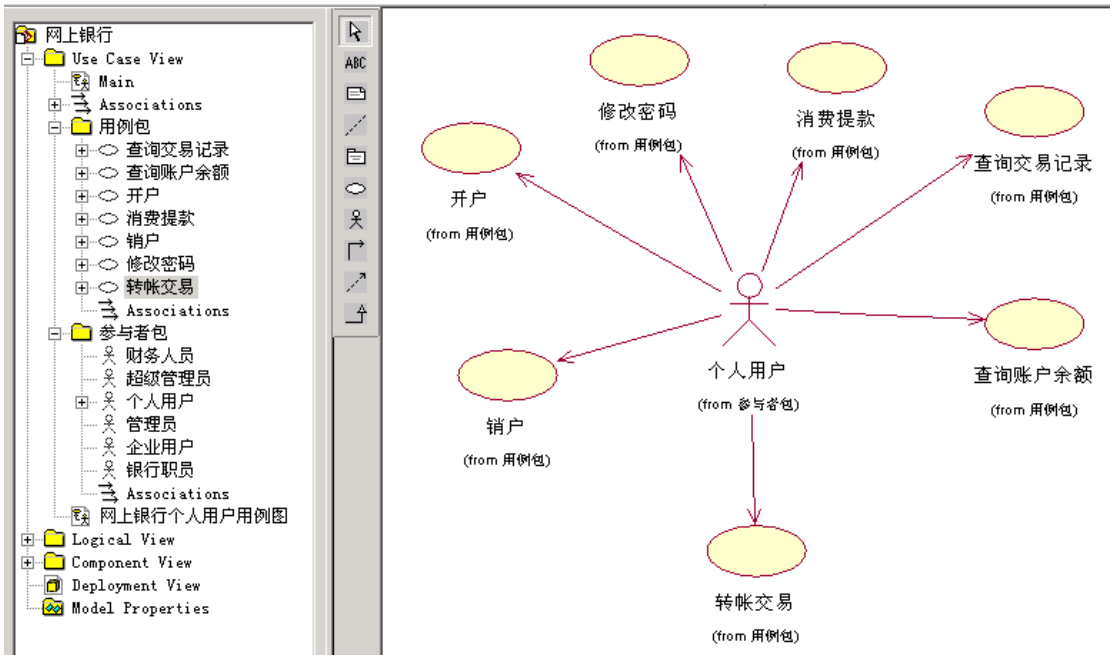
（3）新建用例图

- 右击以新建用例图



- 名称为“网上银行个人用户用例图”

- 在用例图中将前面所创建出的各个用例和参与者拖到用例图中，并点击“单向关联”箭头关联参与者和各个用例



4.4 UML 辅助网站规划和设计

一、案例背景

Web 网站往往具有复杂与高度动态的特点。为了让 Web 应用在短时间之内开始运作，开发周期应该尽量地短。许多时候，开发者直接进入编写代码这一阶段，却不去仔细考虑自己想要构造的是什么样的网站以及准备如何构造：服务器端代码往往是毫无准备的即兴式编写，数据库表也是随需随加，整个应用的体系有时候呈现一种无规划状态。然而，只要我们运用一些建模技术和软件工程技术，就能够让开发过程更加流畅，确保 Web 应用将来更容易维护。

下面介绍用 UML 为 Web 网站建模的一些方法。全面采用 UML 技术是一个复杂的过程，但 UML 的某些部分很容易使用，而且它能够帮助你用更少的时间构造出更好的系统。

为了示范 UML 在网站建设中的应用，我们将构造一个支持无线用户、提供各个地区天气报表和交通流量报表的网站。使我们对于 UML 建模的应用有一个概念型的了解。

二、规划阶段

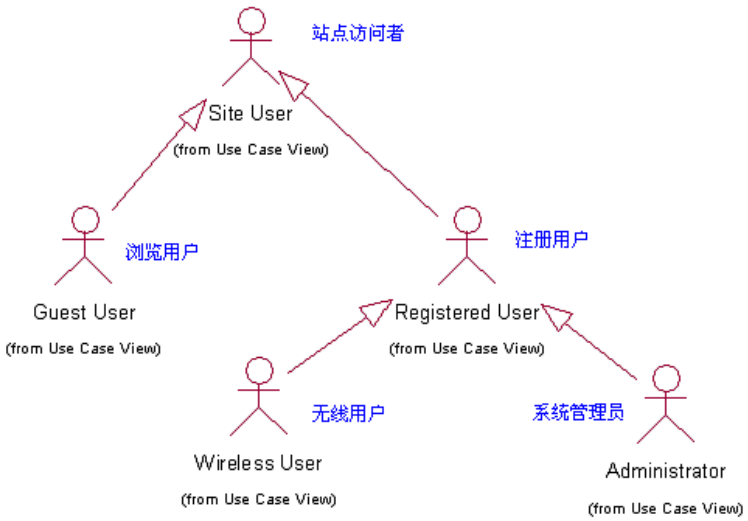
不论你是从头开始构造网站、移植网站还是增加某个重要的功能，为了确保设计决策的最优化，进行一些先期规划是必要的。如果你和其他人协作完成一项工程，就工作总量及其分配达成明确的共识具有不可估量的作用。在规划期间，你应该努力对系统的以下方面形成正确的认识：

- 用户和角色。
- 应用需求。
- 各个界面之间的转换流程。
- 要用到的工具和技术。

1，用户

了解使用系统的用户是很重要的。不仅系统分析要求你接触一些用户（通过问卷调查、email，或者面对面交谈），而且你经常还要让系统能够控制不同的用户角色和权限。通过对用户进行分类并了解他们的需求，你就可以找出线索来确定数据库的安全机制、功能限制方法、用户界面分组、培训和帮助需求、对具体内容的需求，甚至还可以从侧面了解到潜在广告客户的分布。

参与者/角色层次图如下：



上图显示了几组不同的网站用户（在 UML 中称为 Actor，即参与者）。在这里，最普通的用户类型（站点用户 Site User）位于图的顶端，实线箭头表示 generalization 关系（“泛化”关系），它表示 Site User 又可以具体分成两类用户：Guest（浏览用户），Registered（注册用户）。这两类用户共有的特征在“Site User”参与者中说明，而 Guest 和 Registered User 各自私有的特征则在

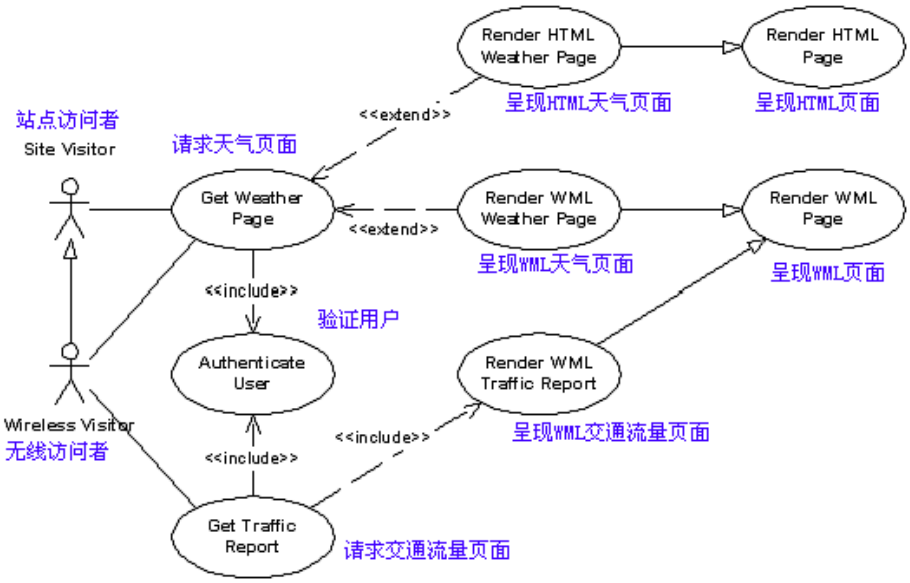
对应的参与者中说明。通常，你可以直接为参与者加上说明文档，无需单独编写说明用户的文档，但具体与你所用的 UML 工具有关。在本例中，Registered User 又可以细分为 Wireless（无线用户）和 Administrator（管理员用户）两种类型，系统对这些用户的处理方式应有所不同。

2，定义需求

在正式开始编写代码之前，你应该对准备构造一个怎样的系统有一个清晰的认识。虽然在编写代码的同时也可以逐步完成这一工作，而且这种做法也很有吸引力，但借助图形和文字资料事先集体进行讨论效率要高得多。

为网站编写详细的需求说明往往不那么合算，但你应该有时间画出几个草图、写下几段注解去说明网站准备提供的服务。这就要用到 Use Case 图（用例图）。

Use Case 可以看成一组功能——它可能对应网站上的一个页面、一个必须编写的程序，或者网站上可能发生的一个动作（比如，验证用户登录，改变用户的配置文件，清除过期的帐号，等等）。下面就是一个能够帮助你规划网站的 Use Case 图。注意，该图并没有显示出网站的所有 Use Case，通常我们需要多个 Use Case 图才能描述完整的网站功能。



即使是在这样一个简单的 Use Case 图中，我们也能够轻松地表达出大量的信息。例如：

include 关系说明两个 Use Case 包含同样的身份验证功能；

extend 关系说明天气页面可能以 WML 或者 HTML 格式显示；

泛化关系说明各个具体的表现过程将遵从“Render HTML Page”或者“Render WML Page”所描述的基本行为规则以达到维持统一的风格效果和统一宏观行为模式的目的。

上图也显示出无线用户能够访问网站中其他用户不能访问的某些区域。在这个 Use Case 图中，只有无线用户能够访问交通流量报表（Traffic Report）。这是因为我们已经得知只有在旅途中的移动用户才需要交通流量报表，而且不想再花时间把交通流量报表制作成其他标记语言形式。

由此，“Get Traffic Report”Use Case 不需要分成 WML 和 HTML 两种显示形式，它可以直接包含“Render WML Traffic Report”这个 Use Case。

一般地，你应该为这些 Use Case 加上简单的说明。具体地说，你应该描述每一个 Use Case 里将要发生什么，谁可以使用它，它如何启动、如何停止，以及某些时候可能发生的特殊事件（称为 variation，即变化）。

4.5 用例的事件流与用例行为分析

业务用例的动态建模主要用于分析系统行为，而行为是用事件引发的。下面，我们来说明如何展开用例的细节和逻辑流程，以便更好地理解用户的需求。

一、用例所涉及的主要事件

1，用例的事件流

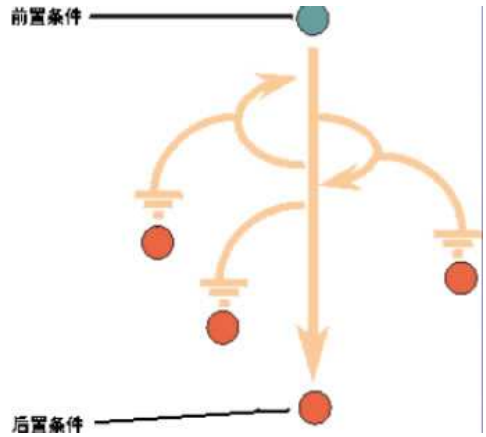
用例的事件流是对完成用例行为所需的事件的描述。事件流描述了系统应该做什么，而不是怎么做。可以通过一个清晰的，易被用户理解的时间流来说明一个用例的行为。在事件流中包括用例何时开始和结束，用例何时和参与者交互，什么对象被交互以及该行为的基本流和可选流。

其目的是对用例进一步细化。

二、一个用例的事件流的组成

1，事件流所应该包含的内容

- 简要说明：描述该使用案例的作用（可以不写出）；
- 前置条件：开始使用该用例之前必须满足的系统和环境的状态和条件（必要条件而不是充分条件）
- 主事件流：用例的正常流程（事件流是关注系统干什么，而不是怎么干），也称为用例的路径。
- 其它（备选）事件流：用例的非正常流程，如错误流程
- 后置条件:用例成功结束后系统应该具备的状态和条件(但不是每个用例都有后置条件)



2，主事件流（用例的路径）

可能包含有基本路径、备选路径、异常路径、成功路径和失败路径等几个方面的内容。

三、描述用例的事件流的主要方式

1，结构化语言

每个用例只描述没有大的分支的行为的单个线索，在事件流中要对事件流进行结构化说明（主事件流和备选事件流）。

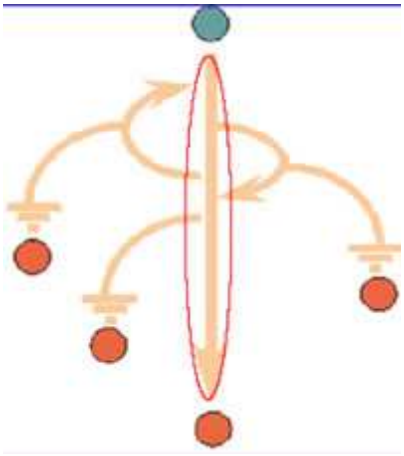
2， UML 的活动图（系统活动图---和用例活动图）

使用活动图可以表示由内部生成的动作驱动的事件流，活动图能提醒您注意并展示并行的和同时发生的活动。这使得活动图成为建立工作流模型、分析用例以及处理多线程应用程序的得力工具。

四、用例事件流描述的基本要求

1，首先写出基本的路径：

这是最主要的事情，因为它是用户最关心或者最想看到的内容。



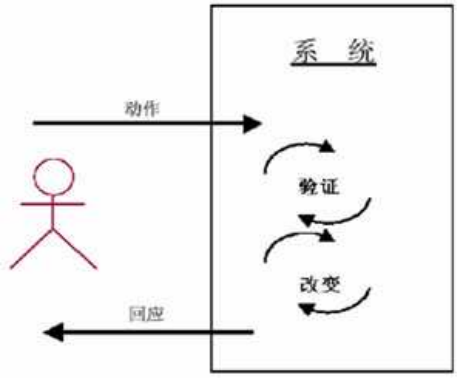
2，用例交互的四步曲

1）参与者产生某个行为动作；

2）然后系统对此动作进行响应；

3）响应成功后再根据动作的要求进行状态的改变；

4）最后将改变后的结果再回馈给参与者。



3，基本的要求

1）只写可观测的

如：

错误的：系统通过 ADO 建立数据库连接，传送 SQL 查询语句，从“用户信息表”中查询。。。

正确的：系统按照查询条件搜索用户信息

2）使用主动语句

错误的：系统从会员处获取用户名和密码，用户名和密码被验证

正确的：会员提交用户名和密码，系统验证用户名和密码

3）语句必须以参与者或者系统作为主语

如：

参与者 XXX

系统 XXXX

4）程序的逻辑处理：判断和循环

5）不要涉及界面的细节

如：

错误的：用户从下拉框中选择类别，并在相应的文本框中输入查询条件，最后用户点击“确定”按钮。。。

正确的：用户选择类别，并相应地输入查询条件，最后用户进行提交。。。

6）模板格式

用例名称	检索图书
参与者	图书管理员/读者
前置条件	图书管理员/读者已经登录
后置条件	系统已显示读者所需的图书
基本路径（主事件流）	1. 图书管理员/读者提交检索条件 2. 系统按检索条件检索图书 3. 系统显示搜索到的图书列表 4. 图书管理员/读者选中某个图书 5. 系统显示该图书的详细信息
扩展事件流	扩展 2a. 系统没有检索到所需图书： 2a1. 系统显示“没有找到适合检索条件的图书” 2a2. 用例结束
待解决问题	
补充说明	
注释	

4.6 用例文档的编写方法

对于上述一些事件流，以及一些关键和重要的问题，需要对用例行为详细分析，这时需要使用文档而不是图。

一、目标和情节

用户一定会有自己的目标（统一过程中也称之为“需要”），并且希望计算机能够帮助他们实现这些目标。

在多种获得系统需求的方法中，简单而熟悉的方法往往是最好的，因为这可以使目标和需求的定义和评估更加容易。

用例就是表达如何使用系统达到目标的一组情节。

场景（scenario）：是参与者和被讨论系统之间一系列特定的活动和交互，通常被称之为“用例的实例”。

通俗的讲，一个用例就是描述参与者使用系统达成目标的时候一组相关的成功场景和失败场景的集合。

处理返回，例：

成功场景（主事件流）：

一个顾客携带商品到达收费口。

收银员使用 POS 系统记录并返回每件商品的信息。

替代场景（扩展事件流）：

a）如果信用卡授权被拒绝，则通知顾客并询问顾客是否可以使用其它支付方式。

B）如果系统无法识别商品的标识码，则通知收银员并建议手工输入识别码。

C）如果系统检测到与外部税金计算系统通信失败，则…….

用例分析的关键是专注于“怎样才能使系统为用户提供可观测的数据，或帮助用户实现它们的目标”，而不是仅仅把系统需求用特性和功能的细目罗列出来。

在需求分析中，我们必须专注于考虑系统怎么才能增加价值和实现目标。

用例的主要思想是：为功能需求写出用例（而不是老式的为“系统会怎么做”的功能列表），在统一过程中用例是发现和定义需求的主要方法。在 FURPS+中，用例强调的是 F，也就是功能性行为。

二、用例的类型和格式

黑箱用例和系统职责：

黑箱用例是最普遍和推荐的用例，它对系统内部工作和设计不进行详细描述，而是描述系统应该具有哪些职责。

在面向对象的思想方法中，所谓封装性，正是指着这种思想。

优秀的软件开发模式中，一个很重要的方法就是避免需求分析阶段决定怎样去实现。

正确的描述：

系统记录销售信息。

错误的描述：

系统把销售信息写入数据库。

或者更糟：

系统为销售信息生成一条 SQL 的 INSERT 语句。

用例可以有几种形式：

1）简洁用例：

扼要的一段概括，比如：

一个顾客携带采购的商品到达收费口，收银员使用 POS 机记录每件商品，系统给出…..

2）临时用例：

非正式的段落格式，比如上面处理返回的例子。

3）详述用例：

最详细的描述格式，所有的步骤和其中的变化都被详细写出，包括辅助部分，例如“前置条件”、“成功后的保证”等。

三、如何编写用例

1，编写用例时，最应该注意的几种问题

- 编写功能需求（可以从文本和口头的功能需求中提取出来），而不是编写使用设想文本
- 描述属性与方法，而不是描述习惯用法
- 编写的用例过于简要并且把自己与用户界面完全隔离

- 回避详细的边界对象名称，同时从第三者的角度而不是用户角度编写用例，并用采用被动式
- 仅仅描述用户交互，而忽略系统响应

2，谁应该书写用例文档

- 最完美：业务人员接受训练，写出优美的用例文档
- **最现实**：业务人员提供素材，开发人员写用例文档
- 最糟糕：业务人员不管，完全由开发人员杜撰

3，用例示例点评一

1）错误用例

用例：提取现金
范围：ATM 系统
主执行者：储户
1.储户插入 ATM 卡，并键入密码
2.储户按“取款”按钮，并键入取款数目
3.储户取走现金、ATM 卡并拿走收据
4.储户离开

2）修正上面的用例

问题原因：没有系统（在上面的文字中没有出现 ATM 机器系统，这样无法了解储户也就是参与者与哪个系统在交互）。修正后：

范围：ATM 系统
主执行者：账户持有者
1.通过读卡机，储户插入 ATM 卡
2.ATM 系统从卡上读取银行 ID、账号、加密密码，并用主银行系统验证银行 ID 和帐号
3.储户键入密码，ATM 系统根据上面读出的卡上加密密码，对密码进行验证。
4.储户选择取款，并键入取款数量。
5.ATM 系统通知主银行系统，传递储户账号和取款数量，并接收返回的确认信息和储户账户余额。
6.ATM 系统输出现金、ATM 卡，显示账户余额的收据。
7.ATM 系统记录事务到日志文件。

4，用例示例点评二

1）错误用例

用例：提取现金
范围：ATM 系统
主执行者：储户
1.收集 ATM 卡，键入密码
2.收集取款事务类型
3.收集提取金额
4.验证账户上是否有足够储蓄金额
5.输出现金、收据和 ATM 卡
6.复位

2）修正上面的用例

问题原因：没有主持行者（也即没有出现参与者，这样无法了解谁在使用系统；并且描述的语句应该以参与者或者系统为主语）。修正后：

范围：ATM 系统
主执行者：账户持有者
1.通过读卡机，储户插入 ATM 卡
2.ATM 系统从卡上读取银行 ID、账号、加密密码，并用主银行系统验证银行 ID 和帐号
3.储户键入密码，ATM 系统根据上面读出的卡上加密密码，对密码进行验证。

4.储户选择取款，并键入取款数量。
5.ATM 系统通知主银行系统，传递储户账号和取款数量，并接收返回的确认信息和储户账户余额。
6.ATM 系统输出现金、ATM 卡，显示账户余额的收据。
7.ATM 系统记录事务到日志文件。

5，用例示例点评三

1）错误用例

用例：买东西
范围：采购应用系统
主执行者：顾客
1.系统显示输入 ID 及密码屏幕。
2.顾客键入 ID 和密码，然后按 OK。
3.系统验证顾客 ID 及密码，并在屏幕上显示个人信息。
4.顾客键入姓名、街道地址、城市、州、邮编、电话号码，然后按 OK。
5.系统验证是否为老客户
6.系统显示可用商品列表
7.顾客选取需要购买的商品及数量，完成时按 DONE。
8.系统通过库存辅助系统验证购买商品是否有足够库存。
9. … …

2）修正上面的用例

问题原因：过多用户接口细节的描述，有点太琐碎！修正后：

1. 顾客使用 ID 和密码进入系统
2. 系统验证顾客身份。
3. 顾客提供姓名、地址、电话号码
4. 系统验证顾客是否为老顾客
5. 顾客选择购买商品及相关数量
6. 系统由库存系统验证购买商品是否有足够库存。

四、用例文档中几个元素的解释

1，序言元素

要把最重要的元素放在一开始。而把一些不重要的“标题”材料放在末尾。

用户兴趣列表：

这个列表很重要也很实用。

用例作为行为的契约，扑获所有与满足客户兴趣有关的行为。

用例应该包含什么？答：

用例应该包含满足所有客户感兴趣的内容，另外，在写出用例所有部分之前，需要确定用户及其兴趣。

例如，如果我们没有列出“销售人员提成”的兴趣，在开始部分我们可能会漏掉这个职责。以客户兴趣作为视点来观察，会给我们提供一种彻底的、系统化的程序，用来发现和记录所有必需的行为。例：

项目相关人员的兴趣：

收银员：希望能够准确快速的输入，没有支付错误。

售货员：希望自动更新销售提成。等。

2，前置条件和后置条件（成功后的保证）

前置条件：

规定了用例一个场景开始之前必须为“真”的条件。

前置条件在用例中不会被检验，我们假定它已经被满足，通常前置条件是已经成功完成的其它用例的一个场景。

比如：

“系统已经被登录”或“收银员已经被识别和授权”。

但不要包括没有价值的条件，比如“系统已经被供电”。

前置条件主要表达读者应该引起警惕的或者值得注意的那些假设。

注意：前置条件用来描述的是条件和假设，既不是用例的输入，也不反映对用例所描述的工作的触发，这对于合理书写前置条件很重要。

后置条件：

后置条件也叫“成功后的保证”，表达了用例成功结束以后必须为真的条件，这个“保证”应该满足所有客户方的需要。

这里所有的客户方，指的是所有参与使用项目的人员。

例：

前置条件：收银员已经被识别和授权。

后置条件：存储销售信息，准确计算税金，更新账目和库存，记录提成，生成收据。

3，基本流程（主事件流）

我们可以把主要成功场景成做“基本流程”，它描述了能满足客户兴趣的典型成功路径。

一般它不包括任何条件和分支，而把条件和分支放在“扩展”部分说明。

场景记录以下三种步骤：

1）参与者之间的交互。

2）一个验证动作（通常由系统来完成）。

3）由系统完成的状态改变。

例：注意表示重复的习惯用法

主要成功场景：

1．顾客携带购买的商品到达 POS 机收费口

2．收银员开始一次新的销售

3．收银员输入商品标识

4．…..

重复 3－4 步，直到结束。

5．……

4，扩展

扩展又称之为“替代流程”，它说明了所有其它的场景和分支。

扩展往往比主要成功场景长而且复杂，这正是我们所希望的。在写完整用例的时候，基本流程加上扩展能满足几乎所有客户的兴趣。

扩展场景是从主要成功场景中分离出来的，所以标记方式应该相同，比如，第 3 步的一个扩展就被标记为 3a。

扩展：

3a.非法标识

1．系统指示错误并拒绝输入。

3b.多个具有相同类别的商品，不需要跟踪每个商品的唯一身份

1．收银员可以输入商品类别的表识和数量。

3c.顾客要求从艺术如商品中减去一个商品

1．收银员输入商品标识并将其删除。
2．系统显示更新后的累加值。
……..

一个“扩展”有两部分组成：条件和处理，处理的步骤可以有多个。
有时候扩展可能会非常复杂，这就需要用单个用例来完成扩展。
下面看看怎么标记扩展中的失败：

7b.信用卡支付
1.顾客输入信用卡账号
2.系统向外部信用卡授权服务系统请求支付验证
2a 系统检测到和外部信用卡授权服务系统通信故障
1. 系统向收银员知识发生了错误
2.收银员向客户请求更换支付方式
3…….

如果想要描述一个可能在任何一步（至少是绝大多数步骤）都会发生的条件，那么应该使用类似“*a”、“*b”这样的标记。

*a.任何时刻，发生一下状况，系统将会崩溃。
1．收银员重启系统，登录，请求恢复上次状态。
2．系统重建之前的状态。

5，特殊需求

如果有一些于这个用例有关的非功能性需求（质量属性或约束条件），那么应该把他们记录在一起。例：

特殊需求

- 在大型平板显示器上触摸屏界面，文本信息要能在一米之外看清。
- 90%信用卡授权机构的响应，应该能在 30 秒之内收到。
- 支持多种语言显示。
- 在步骤 2 和步骤 6 种可以插入新的业务规则。

现在更通用的做法是把所有非功能性需求放在“补充规范”中，这样更容易进行内容管理，也更容易读，因为这些需求通常要求在进行系统整体架构分析的时候通盘考虑。

6，技术和数据的变化列表

系统通常有一些技术上的变化是关于“应该怎样做”，而不是“应该做什么”，需要再用例中把这些变化记录下来。

比如，数据表示方案可能会有不同的变化，在列表中应该记录这种变化。

技术和数据的变化列表：

3a. 商品标识可以用条码扫描也可以用键盘输入。
3b. 商品表识可以采用 UPC、EAN、JAN、SKU 等不同的编码方式。
7a. 信用卡账号信息可以使用读卡器或键盘输入。
7b. 记录在纸面收据上的信用卡支付签名，但我们预测，两年内会有许多顾客希望使用数字签名。

五、示例：处理销售

作者:_____ **日期:**_____

版本: _____

用例名:	**Process Sale**	**用例类型** 业务需求
用例 ID:	TB-SALE2.00	
主要业务参与者:	收银员	
项目相关人员兴趣:	收银员：希望能准确、快速的输入，而且没有支付错误。 售货员：希望自动更新销售提成。 顾客：希望购买过程能够省力，并得到快速服务，希望得到购买证明，以便退货。 公司：希望准确的记录交易，并满足顾客要求，希望保证支付授权服务的信息被记录，希望有一定的容错性，即使某些服务不可用也能允许收款，希望能自动快速的更新账目和库存信息。 政府税务机关：希望从每笔交易中抽取税金。 支付授权服务：希望按照正确的格式和协议收到数字授权的请求，希望准确计算给商店的应付款。	
前置条件:	收银员已经被识别和授权。	
后置条件:	存储销售信息，准确计算税金，更新账目和库存，记录提成，生成收据。	
触发条件:	当客户开始验证购买的商品的时候，该用例被触发。	
基本流程:	1．顾客携带购买的商品到达 POS 机收费口 2．收银员开始一次新的销售 3．收银员输入商品标识 4．….. 重复 3 － 4 步，直到结束。 5．…… ……. 10．顾客携带商品和收据离开	
替代流程	*a.任何时刻，发生以下状况，系统将会崩溃。 1．收银员重启系统，登录，请求恢复上次状态。 2．系统重建之前的状态。 3a.非法标识 1．系统指示错误并拒绝输入。 3b.多个具有相同类别的商品，不需要跟踪每个商品的唯一身份 2．收银员可以输入商品类别的标识和数量。 3-6a.顾客要求从已输入商品中减去一个商品 1．收银员输入商品标识并将其删除。 2．系统显示更新后的累加值。 …….. 7b.信用卡支付 1.顾客输入信用卡账号 2.系统向外部信用卡授权服务系统请求支付验证 2a 系统检测到和外部信用卡授权服务系统通信故障 2. 系统向收银员指示发生了错误 2.收银员向客户请求更换支付方式	
结束:	当客户完成支付，该用例结束。	
特殊需求:	1，在大型平板显示器上触摸屏界面，文本信息要能在一米之外看清。 2，90%信用卡授权机构的响应，应该能在 30 秒之内收到。 3，支持多种语言显示。 4，在步骤 2 和步骤 6 种可以插入新的业务规则。	

技术和数据的变化列表：	3a. 商品标识可以用条码扫描也可以用键盘输入。 3b. 商品标识可以采用 UPC、EAN、JAN、SKU 等不同的编码方式。 7a. 信用卡账号信息可以使用读卡器或键盘输入。 7b. 记录在纸面收据上的信用卡支付签名，但我们预测，两年内会有许多顾客希望使用数字签名。
发生频率：	可能会持续发生。
待解决问题:	1，什么是税法的变化？ 2，研究远程服务的恢复问题 3，不同的业务需要什么样的定制？ 4，收银员是否必须在退出系统以后带走他的现金抽屉？ 5，顾客是否可以直接使用读卡器，而不需要收银员代劳？

这个例子虽然不完整，但是一个实际的例子，足以给我们提供一个真实的感受。

我们提倡一种简朴的书写风格，也就是不考虑用户界面，而专注于他们的意图，只对用户意图和系统职责这一级描述，这点很重要。

六、用例的目标

1）基本业务过程的用例

基本业务过程（EBP）是这样定义的：

由一个人在某个时间某个地点执行一项任务，这项任务是对某一业务事件的反应，而且可以增加可度量的业务价值，并且能够保持数据状态的一致。

其实这个定义过于教条，业务只能由一个人完成？两个人就不行？

所以对原则的理解不应该教条主义的处理问题，用例强调了能够增加可见的和可度量的业务价值，而且能够使系统和数据处于稳定和一致的状态中。

其实我们使用 EBP 原则的时候，主要是在对应用进行分析的时候来寻找主要用例。

2）用例和目标

参与者都有自己的目标（或需要），因此，一个 EBP 级别上的用例，通常被称之为一个**用户目标**级别上的用例。

因此，处理问题的过程应该是：

- 找出用户的目标。
- 为每个目标定义一个用例。

3）用 EBP 指导原则的实例

作为一个系统分析师，在需求分析会上可以这样了提出问题：

系统分析师：在使用 POS 系统的时候，你的目标是什么？

收银员：快速登录还有收款

系统分析师：你认为登录更高级别上的目标什么？

收银员：我要向系统证明身份，这样才能允许我使用系统

系统分析师：比这更高的目标呢？

收银员：防止盗窃、数据崩溃，不显示不宜公开的企业信息。

我们分析一下这段对话。

“防止盗窃、数据崩溃，不显示不宜公开的企业信息”实际上比起用户目标要高，可以认为是企业目标，所以在此不做考虑。

“我要向系统证明身份，这样才能允许我使用系统”看起来接近于用户目标，但它不是 EBP 级别上的行为，因为它不会增加可见的或者可以度量的业务价值，它是为期它目标服务的。

而“完成一次销售”是符合 EBP 原则的更高一级目标。

七、识别其它需求

仅仅写出用例还是不够的，还需要识别其它种类的需求，这些将被包含在补充规范中。例如：

简介：	
功能性：	日志和错误处理 安全性 人性因素
可靠性：	可恢复性
性能：	适应性 可配置性
实现约束：	比如开发的领导层坚持要使用 Java 开发。
采购的组件：	
免费开放源码的组件：	
接口：	值得注意的硬件和接口 触摸屏 条码扫描仪 数据打印机 信用卡读卡器 签名读取器（第一版中不支持）

还需要列出术语表：

术语	定义和相关信息
商品	销售的产品或服务
支付授权	一外部的支付授权服务提供验证，保证销售者得到支付
支付授权请求	以电子方式发送到支付授权系统的一组元素，通常是字符串，这些元素包括：商场 ID，顾客账号，数据和时戳

术语表也可以作为数据字典使用。

八、网上银行子项目中主要的用例说明

银行是与生活密切相关的机构，能为个人和企业提供存款、取款、转账等业务。在银行设立帐户的人或机构通常称为银行的客户。客户可以进行存款、取款、转账。经分析，可以获知系统至少应具有如下功能：

- 客户可以开户、销户、存钱、取钱、转账等亿万要求
- 并且一个银行可以有多个帐户，同时一个客户可以持有多个帐户

（1）参与者主要有

银行职员（Clerk）、客户（CustomerActor）、银行管理员（BankActor）

（2）主要的用例

- 客户登录（Login）
- 客户修改密码
- 存款（Deposit Fund）
- 取款（Withdraw Fund）
- 管理帐户（Maintain Account）
- 转账（Transfer Fund：银行内；银行间）

（3）银行职员登录用例的事件流描述

请见前面的网上书店中的用户登录的用例说明，基本类似。

（4）转账用例的事件流描述

用例名称：	客户转账
参与者：	银行职员

前置条件：	客户必须提供合法的银行身份信息并且银行职员已经登录到系统中
后置条件：	如果转账成功，则客户帐户中存款的金额发生变化。否则，系统状态不变。
主事件流	1、系统提示（银行职员）输入用户姓名、帐号、转账金额。 2、（银行职员）输入相关信息后提交。 3、系统确认资金转出帐户是否存在并有效（账号与户主一致、没有冻结帐户） 4、更新资金转出帐户的相关信息。 5、为资金转出账户建立相关信息。 6、存储转账记录
扩展事件流	1a、如果帐户不存在或无效，显示提示信息，用户可以重新输入或终止该用例。 3a、帐户中金额不足，显示提示信息，用户可以修改转账金额或终止用例（银行间转账）。

（5）用例“取钱”的事件流描述

用例名称：	客户取钱
参与者：	银行职员
前置条件：	客户必须提供合法的银行身份信息并且银行职员已经登录到系统中
后置条件：	如果用例成功，则客户帐户中存款的金额发生变化。否则，系统状态不变。
主事件流	1、系统提示（银行职员）输入用户姓名、帐号、取钱金额。 2、客户输入相关信息后提交。如果无效则执行其他事件流。 3、确定该帐户是否有足够的金额。如果余额不够，则告诉客户该帐户余额不足并结束该用例。
扩展事件流	1a、如果帐户不存在或无效，显示提示信息，用户可以重新输入或终止该用例。 3a、帐户中金额不足，显示提示信息，用户可以修改转账金额或终止用例（银行间转账）。

4.7 用例分析的案例

一、订单处理子系统案例

1、问题陈述：

与过程分析相同。

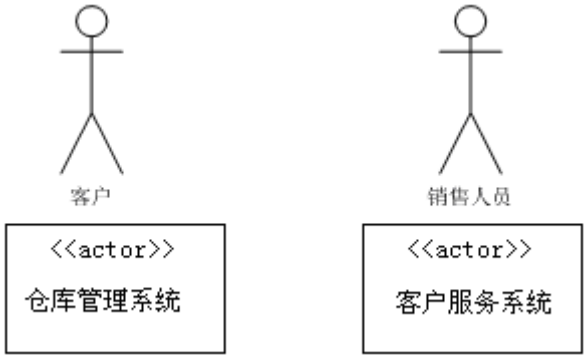
2、参与者

通过如下分析，把问题条理化，发现参与者。

注意，描述的时候不要使用隐语。比如“要 e-mail 给客户”有比较含糊，正确的写法是“销售人员要 e-mail 给客户”。

1，**客户**使用公司的的 Web 页面查看所选择的电源设备标配，同时显示价格。
2，客户查看配置细节，可以更改配置，同时计算价格。
3，客户选择订购，也可以要求**销售人员**在订单真正发出之前和自己联系，解释有关细节。
4，要发出订单，客户必须填写表格，包括地址，付款细节（信用卡还是支票）等。
5，客户订单送到系统之后， 系统由**客户服务系统**取得该客户的等级，以及由销售策略决定的折扣。
6，销售人员发送电子请求到**仓库管理系统**，并且附上配置细节。
7，事务的细节（包括订单号和客户帐户号），销售人员要 e-mail 给客户，使得客户可以在线查询订单状态。
8，仓库从销售人员处获取发票，并且向客户运送电源设备。

从中我们可以发现四个参与者：客户；销售人员；仓库管理系统；客户服务系统。



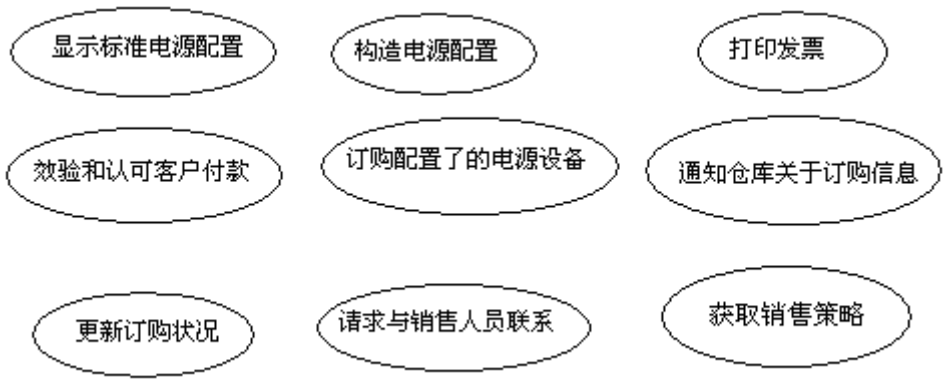
3、建立用例

用例表示一个完整的给用户传值的功能单元，不与用例通信的参与者是没有意义的，但用例可以只为泛化，而不与参与者通信。

我们可以建立一个表来分析，把功能需求赋予参与者和用例。注意，有些潜在的业务功能可能不在应用范围只内，它们不能被转换为用例，比如仓库装配电源设备并且把它运送给客户，这个任务已经超出这个系统的业务范围，不能作为用例。

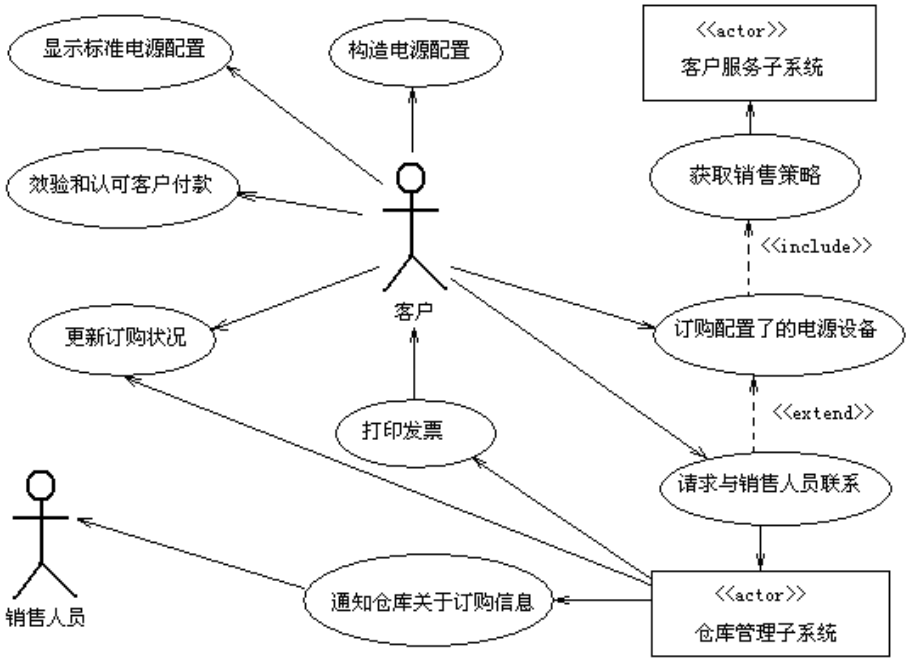
编号	需求	参与者	用例
1	**客户**使用电源部的 Web 页面查看所选择的电源设备标配，同时显示价格。	客户	显示标准电源配置
2	**客户**查看配置细节，可以更改配置。	客户	构造电源配置
3	系统由**客户服务系统**取得该**客户**的等级，以及由销售策略决定的折扣，同时计算价格。	客户 客户服务系统	获取销售策略
4	**客户**选择订购，也可以要求**销售人员**在订单真正发出之前和自己联系，解释有关细节。	客户 销售人员	订购配置了的电源设备 请求与销售人员联系
5	要发出订单，**客户**必须填写表格，包括地址，付款细节（信用卡还是支票）等。	客户	订购配置了的电源设备 效验和认可客户付款
6	**销售人员**发送电子请求到**仓库管理系统**，并且附上配置细节。	销售人员 仓库管理系统	通知仓库关于订购信息
7	事务的细节（包括订单号和客户帐户号），**销售人员**要 e-mail 给客户，使得**客户**可以在线查询订单状态。	销售人员 客户	订购配置了的电源设备 更新订购状况
8	**仓库管理系统**从**销售人员**处获取发票，并且向客户运送电源设备	仓库管理系统 销售人员	打印发票

可以建立如下用例：



4、用例图

用例图的作用是把用例赋给参与者，用例图是系统行为模型的主要可视化技术。还可以建立用例之间的关系，比如图中 Extend 表示“订购配置了的计算机”可以被“客户”扩展为“请求与销售人员联系”。而订购配置了的电源设备包含了（include）获取销售策略的行为，这种依赖关系，表达了获取销售策略的用例不能独立存在，只能作为包含它的订购配置了的电源设备用例的一部分。



5、编写用例文档

用例必须用**事件流**文档来描述，这个文档表达了系统必须做什么和参与者什么时候激活用例。

用例名:	订购配置了的电源设备	**用例类型** 业务用例
用例 ID:		
主要业务参与者:	客户	
描述：	该用例允许客户输入一份购物订单，该订单包括提供运送和发票地址，以及关于付款的详细情况。	
前置条件：	客户通过浏览器进入订单输入的 Web 页面，该页面显示已配置电源设备及价格的详细信息。 当客户在订单信息已经显示在屏幕上的时候，选择“客户”或者相似命名的功能键来确认订购所配置的电源设备的时候，该用例开始。	
后置条件：	如果用例成功，购物订单记录进系统的数据库，否则系统的状态不变。	
基本流程:	1．系统请求客户输入购买细节，包括销售人员的名字（如果知道的话）、运送信息（客户的名字和地址）、发票细节（如果运送地址不同的话）、付款方法（信用卡和支票）以及任何其它注释。 2．客户选择计算价格功能键来发送购买细节，系统通过客户服务系统获取这个等级的客户销售策略，然后计算购买价格。 3．客户选择购买功能键来发送订单给系统。 4．系统给购买订单赋与一个唯一的订单号码和客户账号，系统将订单信息存入数据库。 5．系统把订单号和客户号与所有订单细节一起 e-mail 给客户，作为接受订单的确认。	

扩展流程：	2a，客户对计算的价格有疑问，可以查阅客户服务系统的相应页面，以查询自己的客户等级及当前的销售策略 3a，客户在提供所有要求录入的信息之前，激活购买功能键，系统将显示错误信息，它要求提供所漏掉的信息。 3b，客户选择 Reset（或其它相似命名）功能来恢复一个空白的购物表格，系统允许客户重新输入信息。

二、客户服务子系统用例分析

下面，我们来研究一下另外一个重要的子系统，客户服务子系统的用例分析。

1，发现参与者

客户服务子系统主要实现客户分级管理策略，而且可以灵活的处理促销策略，从项目影响范围的研究中，我们可以发现参与者。

参与者词汇表	
参与者	描述
基本客户	已经有稳定业务往来的公司。
潜在客户	预计可能有业务往来的公司。
曾经客户	过去有业务往来，但是最近 6 个月内没有购买设备，但仍然保持良好的身份记录。
市场部	响应创建折扣和订阅程序，并为公司进行销售的组织部门。
客户服务部	按照合同为客户提供联系服务的组织部门。
财务部门	处理客户付款和收费，以及维护客户账户信息的组织部门。
时间	触发时序事件的参与者。
仓库管理子系统	存储和维护 TB 公司产品库存，并且处理顾客发货和退货的实体。
网上销售子系统	实现基于 Web 的电源产品销售。

2，确定业务需求用例

与已经建立的网上订购项目相比，这个新系统的最大特点，是把市场行为作为系统的重要功能，所以功能上和以前的系统有诸多不同。

首先我们需要记录项目的初始范围，也就是定义一个系统应该准备支持的业务方向，这就是所谓系统上下文数据流图，其方法是：

1，区分内部和外部，忽略内部工作，专注于外部功能。

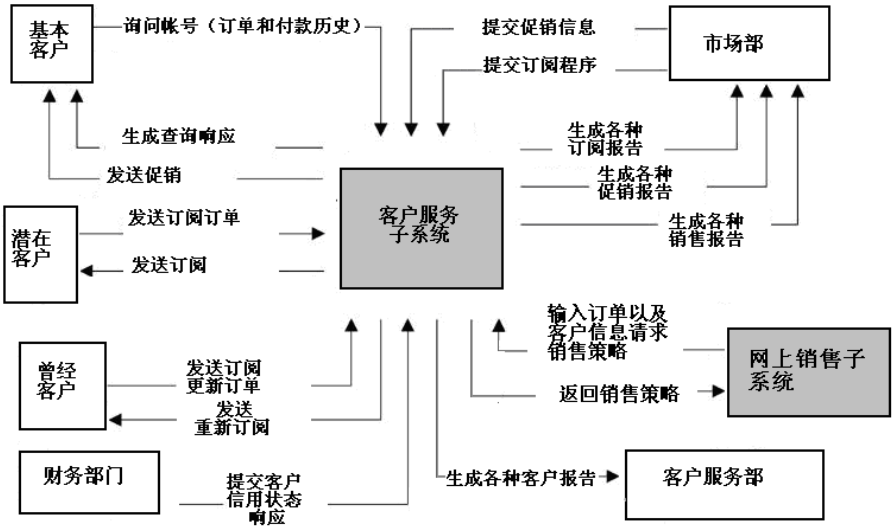
2，询问最终用户系统需要响应什么业务事务，这些业务事务就是系统的**净输入**。

3，询问最终用于系统需要有什么响应，这些相应就是系统的**净输出**。

4，确定外部数据存储，以实体的形式表达，数据库和文件一般是属于外部的。

注意：系统上下文并不是越复杂越好，而是要把握系统功能的重点，细节问题可以在后面的详细分析中解决。

下面是这个项目子系统的上下文。

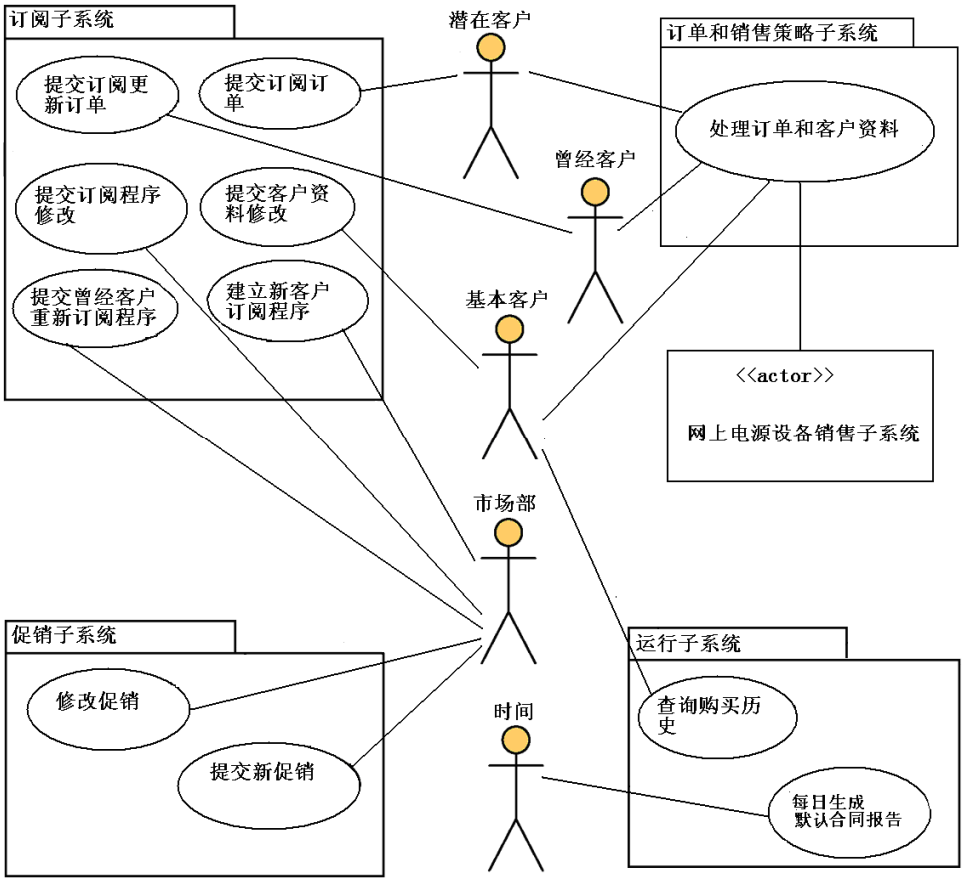


在这个系统中定义边界的时候，可以考虑把网上销售子系统暂时排除在外，也就是把网上销售子系统定义成外部接受者。我们可以建立一些用例并定义它的词汇。

编号	用例名称	用例描述	预期的参与者和角色
1	提交订阅订单	描述一个潜在客户通过订阅加入本系统，这个潜在客户至少答应在两年内购买一定数量的本公司设备。	潜在客户
2	提交订阅更新订单	描述一个过去的客户通过订阅加入本系统，这个客户过去是本公司的基本客户，尽管 6 个月内这个活动一度中断，但现在准备恢复商业活动。	曾经客户
3	提交客户资料修改	描述一个基本客户修改他的个人资料，包括公司地址、e-mail、密码和基本购买配置方式。	基本客户
4	处理订单和客户资料	描述一个客户通过网上销售子系统提交一个 TB 公司产品订单，以及有关的客户资料，等待客户服务子系统返回相关的折扣信息。	网上销售子系统 市场部
5	查询购买历史	描述一个基本客户查阅他三年内的购买历史。	基本客户
6	建立新客户订阅程序	描述市场部建立一个新的客户订阅计划（在什么情况下可以得到更大的优惠）来吸引新的客户。	市场部
7	提交订阅程序修改	描述市场部为基本客户修改订阅计划，比如优惠期的延长等等。	市场部
8	提交曾经客户重新订阅程序	描述市场部建立一个重新订阅计划，比如曾经客户可以更短的时间内享受到优惠，以吸引回曾经客户。	市场部
9	提交新促销	描述市场部建立一个新的促销计划，以吸引不同的客户来购买。需要注意，促销计划一般有专门的名称，以表明在某种情况下可以以特殊的价格来购买。这些促销产品可以集成到一个特殊的目录中在网上公布，并且通过 e-mail 发送给基本客户。	市场部
10	修改促销	描述市场部修改促销条件。	市场部
11	每日生成默认合同报告	描述每天生成一个报告，列出还没有达到“优惠级基本客户”购买量的基本客户，这些客户购买量以 30 天、60 天、120 天过期三个级别排序。	时间（发起） 客户服务部（外部接收者）

注：参与者没有标注的为主要业务参与者，表达他收到了某些可度量的价值。

这样就可以画出系统的用例图。注意这个图中，并没有致力于包含和依赖关系的研究，这是因为图形比较复杂的时候，有些事情单独考虑更有利，比如我们会在后面单独考虑依赖关系，并且以此生成开发策略。



3，撰写用例文档

为了更清楚的表达用例的事件流，需要写出用例文档，这里只列出了“处理订单和客户资料”用例的文档。

电源客户服务系统

作者:_____ **日期:** _____

版本: _____

用例名:	处理订单和客户资料	**用例类型** 业务需求
用例 ID:	TB-ES2.00	
主要业务参与者:	网上设备销售子系统	
其它参与者:	基本客户，曾经客户，潜在客户 销售部（外部接收者） 财务部（外部接收者）	
项目相关人员兴趣:	客户：对自己的级别能得到的优惠感兴趣。 市场部：对销售活动感兴趣，同时也为了计划新的促销。 采购部：对销售活动感兴趣，为了补充库存。 管理层：对销售活动感兴趣，为了评估公司业绩和客户满意度。	
描述:	该用例描述一个客户提交一个 TB 公司产品订单和客户资料，由客户服务系统根据客户级别和销售策略计算销售价格。 客户的资料信息以及他的帐号被验证，订单提交后系统需要查询客户所处于的级别，再根据促销策略，提供则扣信息，然后计算销售价格。	
前置条件:	客户已经登录，由网上设备销售子系统传送来客户资料和购买细节。	
后置条件:	订单被记录，向网上设备销售子系统发送具有折扣的付款信息。	
触发条件:	当新的客户订单被提交的时候，该用例被触发。	
基本流程:	1，由“网上设备销售子系统”传递过来客户提交的资料信息、订单信息	

	和支付信息。 2，系统验证所有信息。 3，根据客户信息验证客户优惠级别。 4，对于订购的每件产品，系统验证产品标识。 5，对每件产品，系统根据客户优惠级别和当前促销政策计算价格。 7，系统计算总价格 8，系统检查客户付款帐号的状态。 9，系统检查客户支付状况（如果是网上支付）。 10，系统记录订单信息，向“网上设备销售子系统”返回价格和优惠信息。
替代流程：	1a，客户没有提供足够的信息，通知客户重新提交。 4a，如果客户需要的产品和 TB 公司提供的商品不符，则要求客户澄清。 8a，如果客户帐号信用不良，则把订单挂起，并且通知客户，退出用例。 9a，如果标准支付方式无法完成，则通知客户，并希望客户提供另一种支付方式。
结束:	当“网上设备销售子系统”收到价格信息的时候，该用例结束。
实现约束和说明：	“网上设备销售子系统”客户为 Web 界面，内部工作人员为 GUI 界面。
待解决问题:	需要研究客户对优惠级别或者促销政策有疑问，能够快速的和有关销售人员联系，该销售人员能够迅速查阅信息，并作出解释。

4.8 活动视图（Activity Diagram）

上面用结构化语句表示的用例事件流，可以很细腻的表达一些用例的过程，但是，当用例的事件流比较复杂的时候，单纯用文字表达难以清楚的表示相互之间的关系，特别是一些并发关系，这时，可以借用活动图来表示。

一、活动视图概述

活动视图（Activity Diagram）是一种特殊形式的状态图，用于对计算流程和工作流程建模。活动图中的状态表示计算过程中所处的各种状态，而不是普通对象的状态。

通常，活动图假定在整个计算处理的过程中没有外部事件引起的中断，否则，普通的状态图更适于描述这种情况。活动图包含活动状态，活动状态表示过程中命令的执行或工作流程中活动的进行。与等待某一个事件发生的一般等待状态不同，活动状态等待计算处理工作的完成。

当活动完成后，执行流程转入到活动图中的下一个活动状态。当一个活动的前导活动完成时，活动图中的完成转换被激发。活动状态通常没有明确表示出引起活动转换的事件，当转换出现闭包循环时，活动状态会异常终止。

活动图也可以包含动作状态，它与活动状态有些相似，但是它们是原子活动并且当它们处于活动状态时不允许发生转换。

活动图可以包含并发线程的分叉控制。并发线程表示能被系统中的不同对象和人并发执行的活动。通常并发源于聚集，在聚集关系中每个对象有着它们自己的线程，这些线程可并发执行。并发活动可以同时执行也可以顺序执行。

活动图不仅能够表达顺序流程控制还能够表达并发流程控制，如果排除了这一点，活动图很像一个传统的流程图。

二、用例活动图

活动图是活动视图的表示法。它包括一些方便的速记符号，这些符号实际上可以用于任何状

态图，尽管活动图和状态图的混合表示法多数时候都很难看。

- 活动状态表示成带有圆形边线的矩形，它含有活动的描述（普通的状态盒为直边圆角）。
- 简单的完成转换用箭头表示。
- 分支表示转换的监护条件或具有多标记出口箭头的菱形。
- 控制的分叉和结合与状态图中的表示法相同，是进入或离开深色同步条的多个箭头。

使用活动图可以表示由内部生成的动作驱动的事件流，活动图能提醒您注意并展示并行的和同时发生的活动。这使得活动图成为建立工作流模型、分析用例以及处理多线程应用程序的得力工具。下面，我们考虑一个订单处理的活动图。

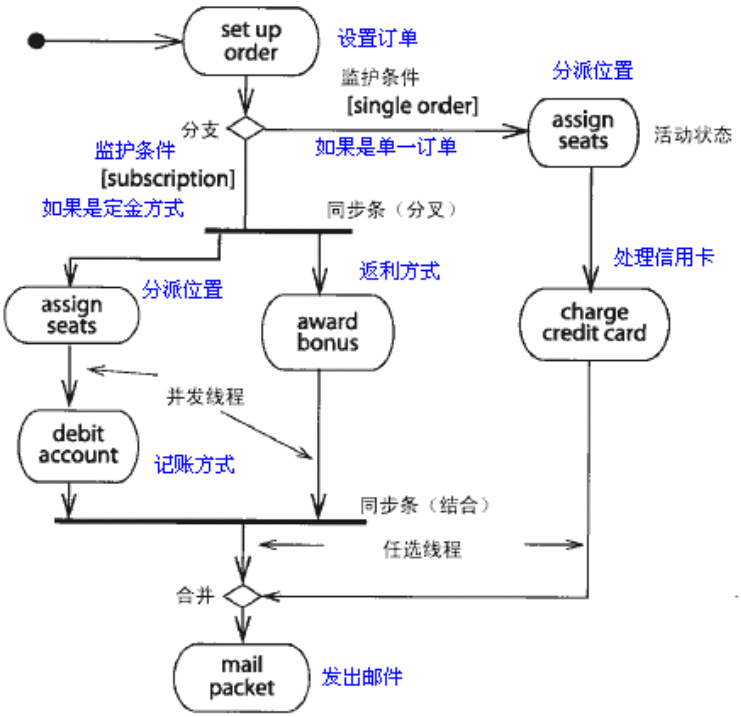
需求如下：

首先设置订单（setup order）。

如果是“单一订单”（single order）则：分派位置（assign seats）；处理信用卡（charge credit card）；发出邮件：mail packet

如果是定金方式（subscription）则：分派位置（assign seats）；有两种处理方式：第一种为计帐方式（debit account ）；第二种为返利方式（award bonus）；发出邮件 mail packet

据此可以画出具体的图。



为了表示外部事件必须被包含进来的情景，事件的接收可以被表示成转换的触发器或正在等待某信号的一个特殊内嵌符号。发送也可以同样的表示。然而，如果有许多事件驱动的转换，那么用一个普通的状态图表示更可取。

三、订单处理子系统的活动建模

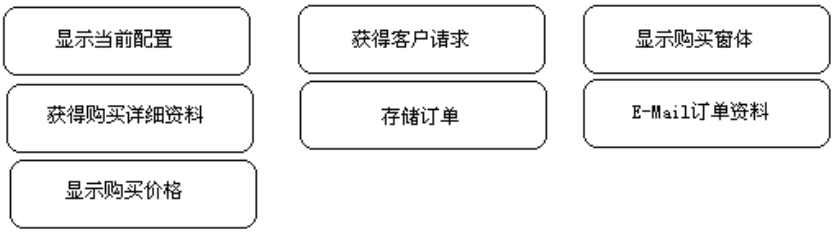
1，发现活动

针对上面已经建立了用例的关于电源设备采购的例子，我们从系统用例的描述，来找出活动，注意，活动是站在设备的角度来描述的。

编号	用例描述	活动状态
1	当客户在订单信息已经显示在屏幕上的时候，选择“客户”或者相似命	显示当前配置；

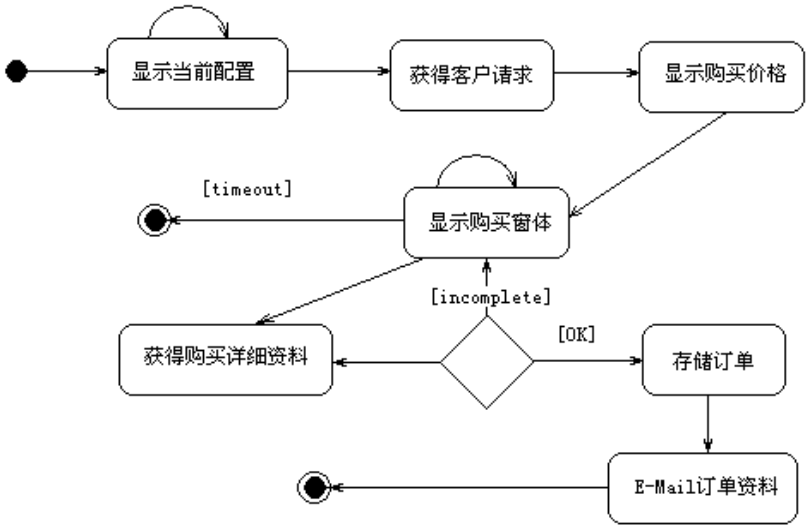
	名的功能键来确认订购所配置的电源设备的时候，该用例开始。	获得客户请求
2	系统请求客户输入购买细节，包括销售人员的名字（如果知道的话）、运送信息（客户的名字和地址）、发票细节（如果运送地址不同的话）、付款方法（信用卡和支票）以及任何其它注释。	显示购买窗体
3	系统由销售服务系统取得客户的等级以及当前销售策略，计算客户购买的实际价格。	显示购买价格
4	客户选择 Purchase（购买，或者相似命名的）功能键来发送订单给 TB 公司。	获得购买详细资料
5	系统给购买订单赋与一个唯一的订单号码和客户账号，系统将订单信息存入数据库。	存储订单
6	系统把订单号和客户号与所有订单细节一起 e-mail 给客户，作为接受订单的确认。	Email 订单资料
7	客户在提供所有要求录入的信息之前，激活 Purchase（或者相似命名的）功能键，系统将显示错误信息，它要求提供所漏掉的信息。	获得购买详细资料 显示购买窗体
8	客户选择 Reset（或其它相似命名）功能来恢复一个空白的购物表格，系统允许客户重新输入信息。	显示购买窗体

把标识的活动画出来。



2，活动图

把活动用转换连线连接起来，就成为活动图。

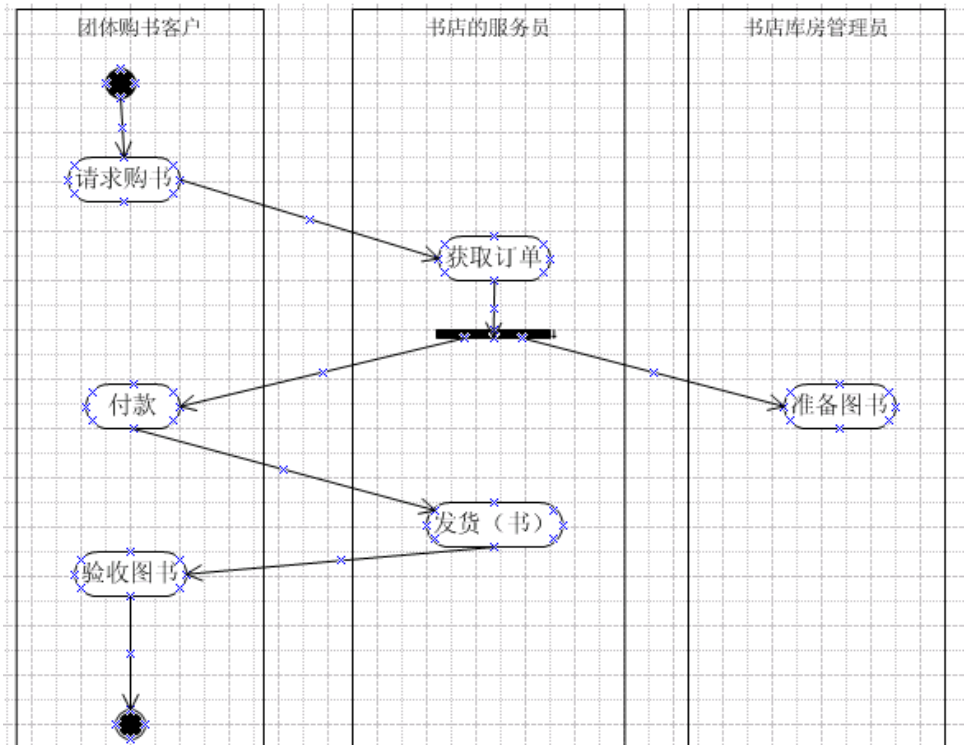


“显示当前配置”是初始活动状态，在这个活动上有一个**递归转换**的表达，描述在进行下一个活动以前，这个活动一直在反复执行。这个方式强调了这是活动而不是行为。

当活动转换为“显示购买窗体”的时候，timeout 将终止这个活动模型的执行，或者“获得购买详细资料”的活动被激活。如果购买资料不完全，系统又回到“显示购买窗体”，否则进入“存储订单”。并且接着进入“Email 订单资料”，然后活动结束。一般来说，只有“退出”活动状态被显示出来，一般活动内部的分支，可以推断出来。有时候重要的分支可以使用分支，并附上监护条件。

四、泳道

将模型中的活动按照职责组织起来通常很有用。例如，可以将一个商业组织处理的所有活动组织起来。这种分配可以通过将活动组织成用线分开的不同区域来表示。由于它们的外观的缘故，这些区域被称作泳道。 某个项目中团体客户购书的带泳道的用例活动图如下：



五、对象流 (Object Flow)

活动图除了能表示活动状态以外，也能表示对象的值流和控制流。在活动图中引入对象流的目的，是希望能更清楚的表达某些状态，而不是实际程序中的对象，这个概念是不同的。但是，这里引入的对象，对后期架构设计的内容将会有很大的影响。

对象流状态表示活动中输入或输出的对象。

- 对输出值而言，虚线箭头从活动指向对象流状态。
- 对输入值而言，虚线箭头从对象流状态指向活动。
- 如果活动有多个输出值或后继控制流，那么箭头背向分叉符号。
- 同样，多输入箭头指向结合符号。

活动和对象流状态都可以被分配到泳道中的活动图。

下面，我们用 Rose 来设置一个活动图，里面将包含活动和对象的状态。

基本的想法是，有一个 Order 类用于处理订单。

要求是：

有三个处理对象：Customer 消费者、Sales 出售 、Stockroom 仓库

由消费者“请求服务”（Request Servive）。

一个选项是：

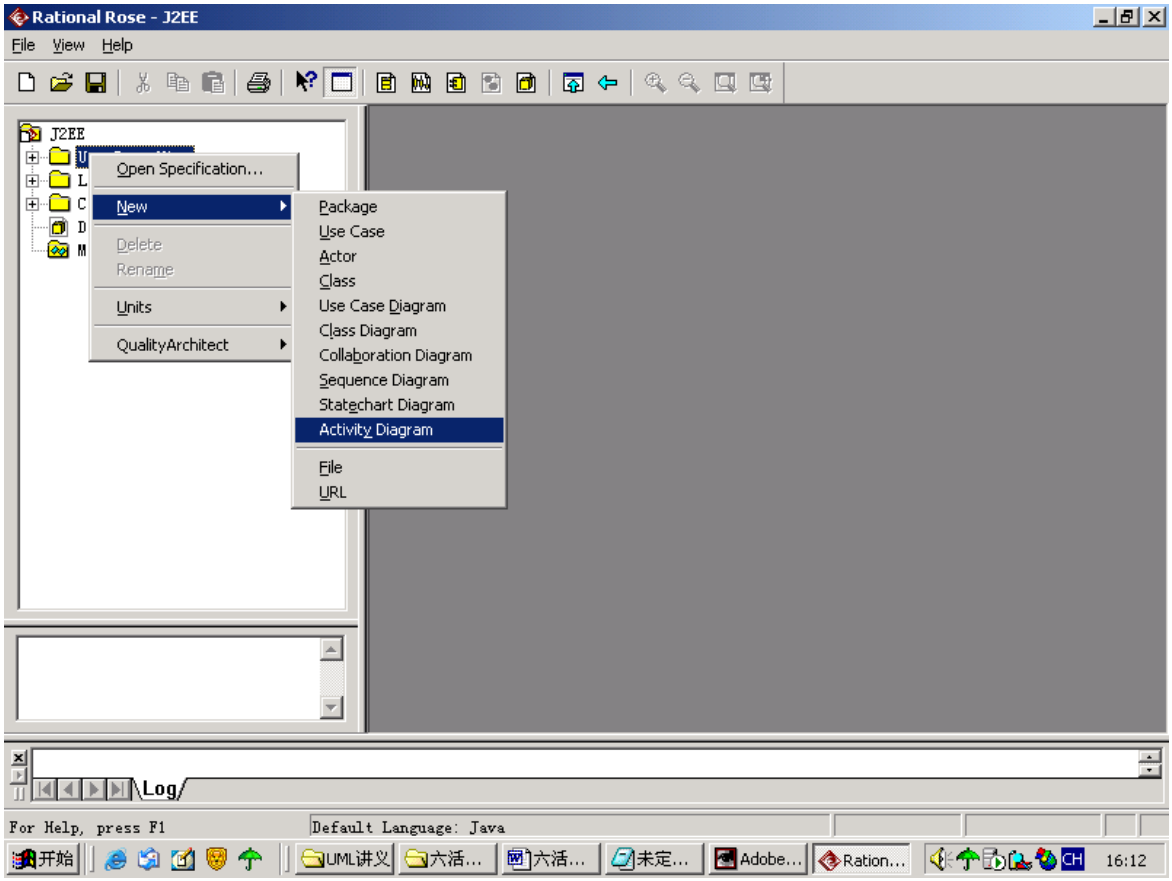
消费者直接支付（Pay）；

销售方递送订单（Deliver Order）

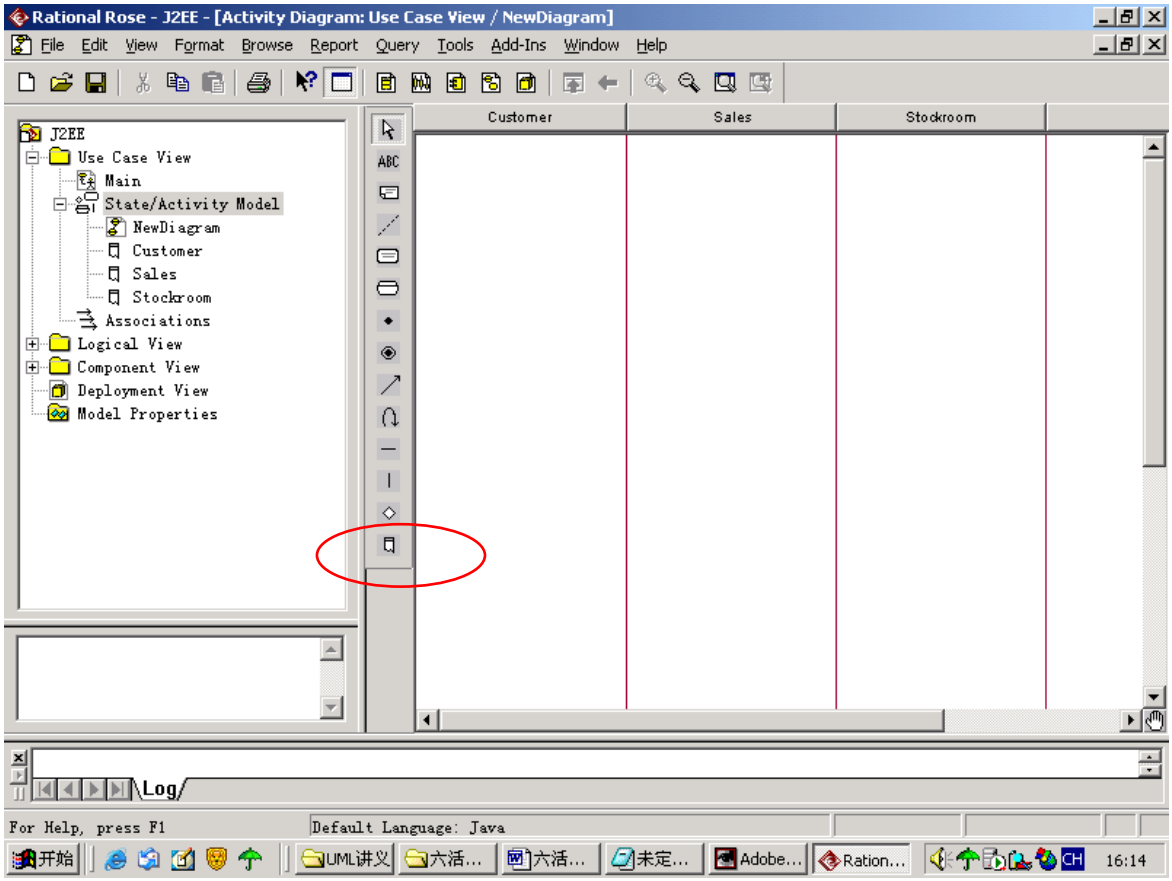
订单递送完毕（Order Delivered）后付款取获。

另一个选项是：

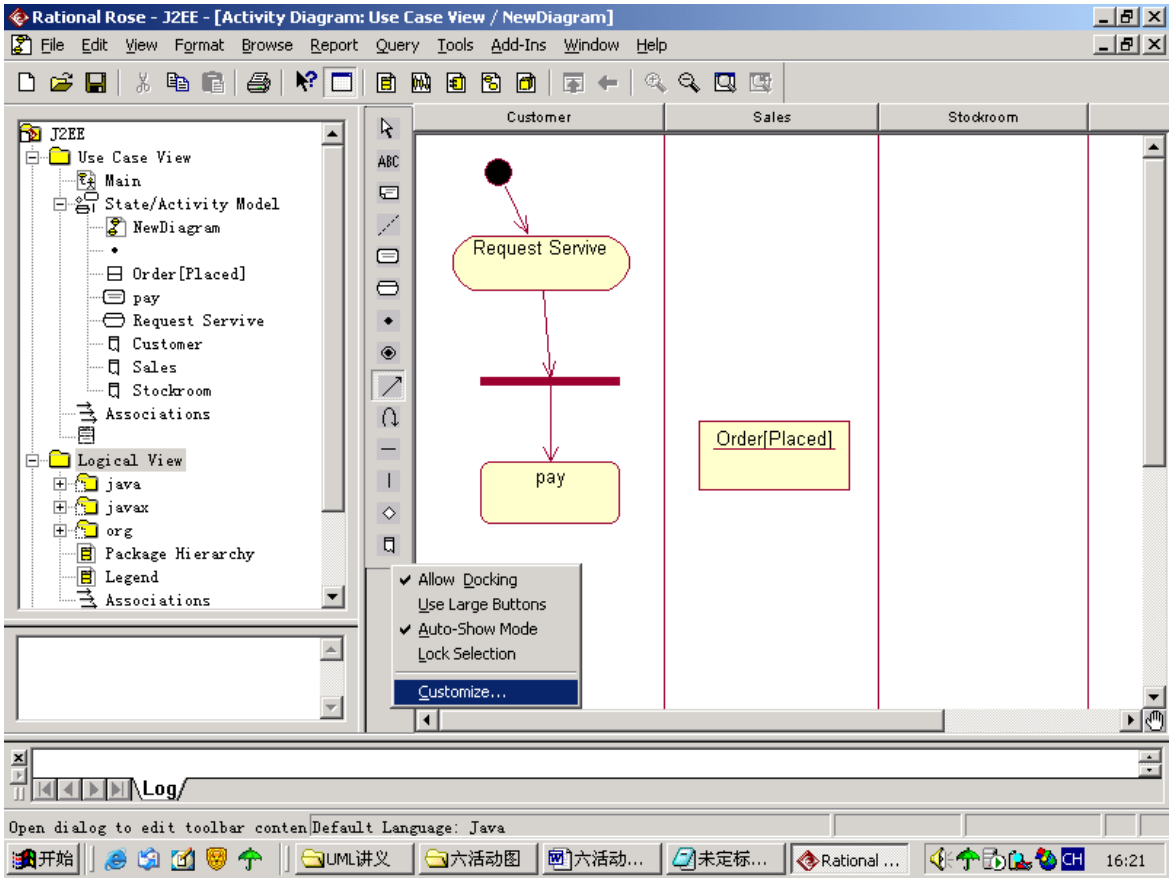
消费者使用订单，首先“放置订单”（Order Placed）
销售部由放置订单对象处理。
处理完毕后，状态变为“获得订单”（Take Order）。
再交由仓库方填写订单（Entered Order）。
填写完毕后，交由销售方完成订单的最后生成（Order Filled）。
销售方递送订单（Deliver Order）
订单递送完毕（Order Delivered）后付款取获。
由于需要三个通道处理问题，可以使用泳道。

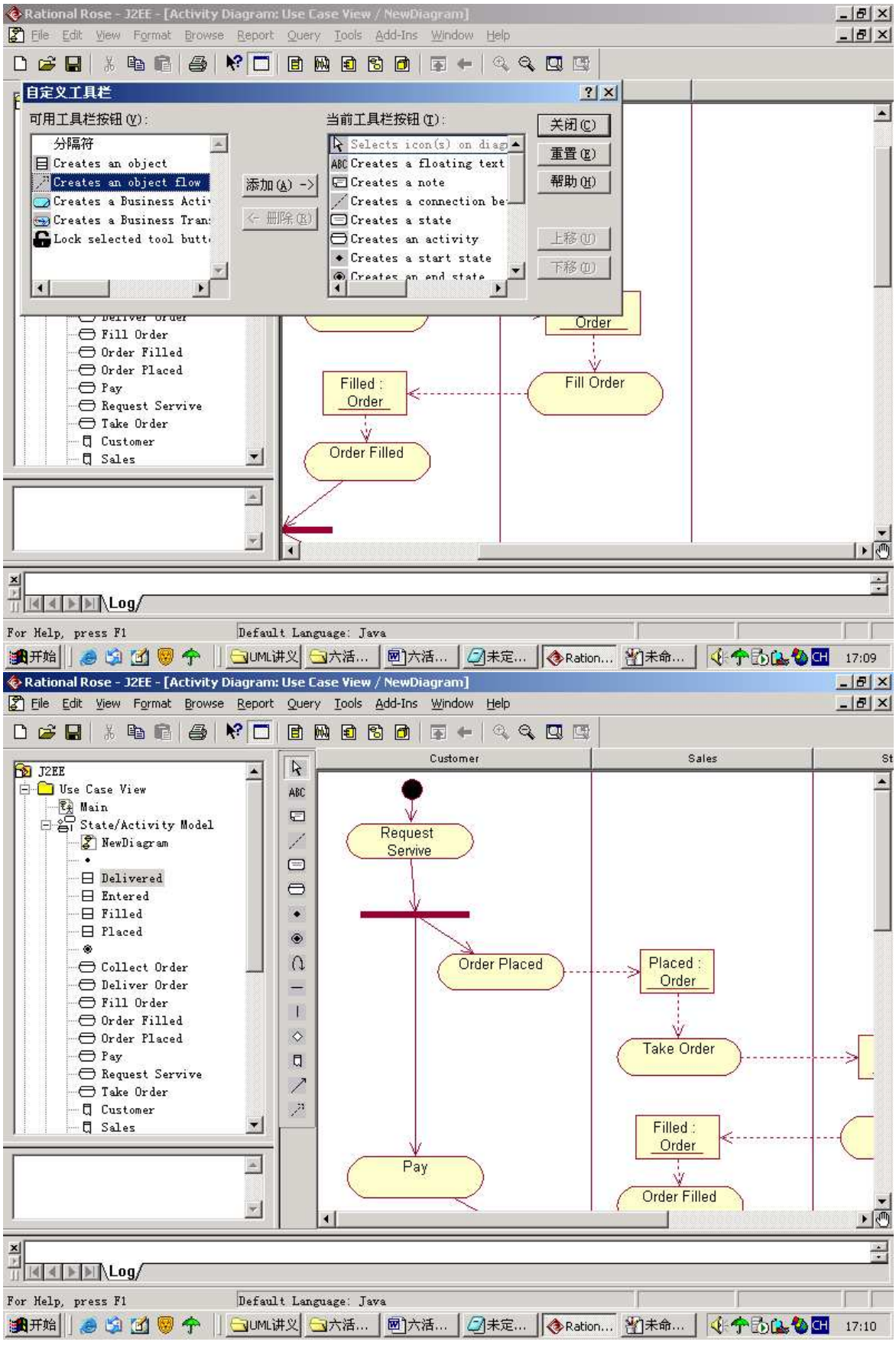


下面是设置泳道。

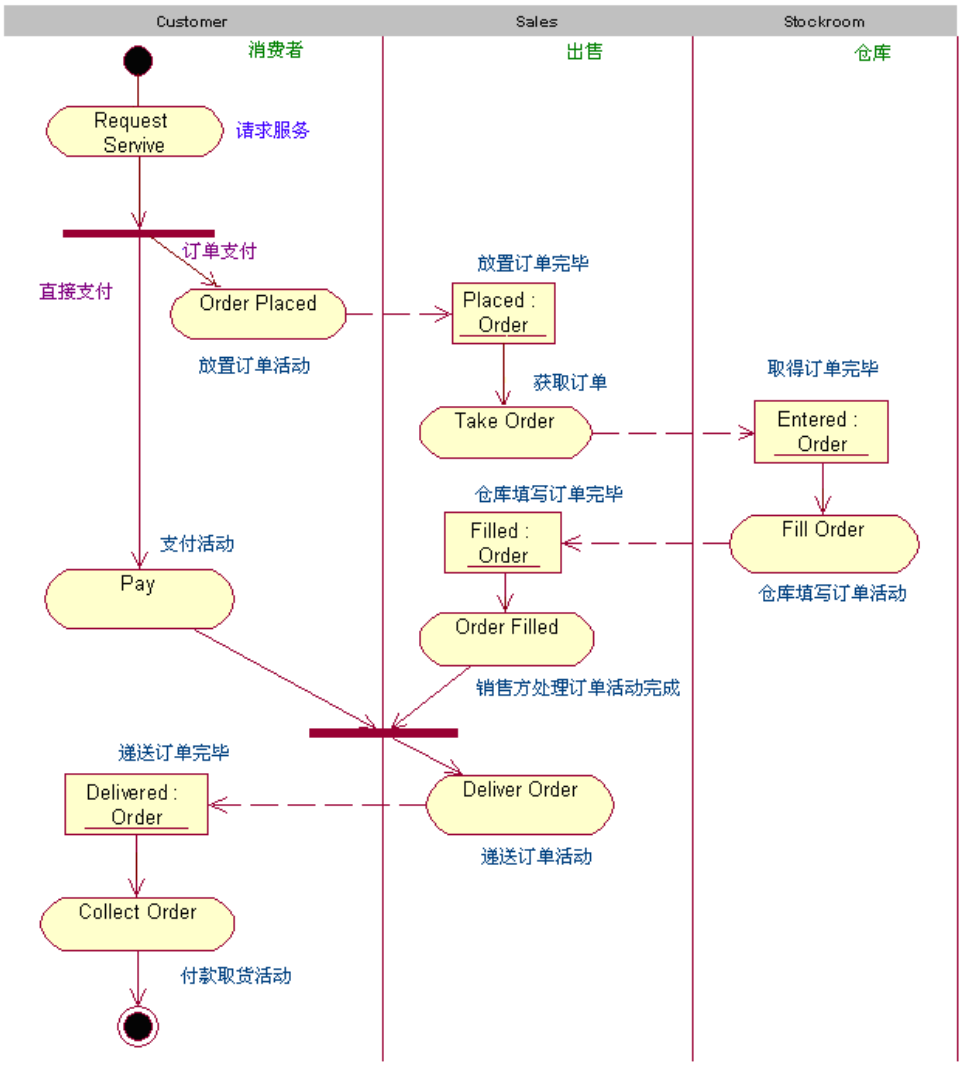


这里需要对象处理，需要加入“对象流”（Object Flow）关系。




Rational Rose - J2EE - [Activity Diagram: Use Case View / NewDiagram]
File Edit View Format Browse Report Query Tools Add-Ins Window Help
自定义工具栏
可用工具栏按钮(V):
分隔符
Creates an object
Creates an object flow
Creates a Business Acti
Creates a Business Tran
Lock selected tool butt
添加(A) ->
<- 删除(R)
当前工具栏按钮(T):
Selects icon(s) on diag
Creates a floating text
Creates a note
Creates a connection be
Creates a state
Creates an activity
Creates a start state
Creates an end state
关闭(C)
重置(E)
帮助(H)
上移(U)
下移(D)
Fill Order
Order Filled
Order Placed
Pay
Request Servive
Take Order
Customer
Sales
Order
Filled : Order
Fill Order
Order Filled
Log
For Help, press F1
Default Language: Java
开始
UML讲义
六活...
六活...
未定...
Ration...
未命...
17:09
Rational Rose - J2EE - [Activity Diagram: Use Case View / NewDiagram]
File Edit View Format Browse Report Query Tools Add-Ins Window Help
J2EE
Use Case View
Main
State/Activity Model
NewDiagram
Delivered
Entered
Filled
Placed
Collect Order
Deliver Order
Fill Order
Order Filled
Order Placed
Pay
Request Servive
Take Order
Customer
Sales
Customer
Sales
Request Servive
Order Placed
Placed : Order
Take Order
Filled : Order
Pay
Order Filled
Log
For Help, press F1
Default Language: Java
开始
UML讲义
六活...
六活...
未定...
Ration...
未命...
17:10

下图表示了泳道。



活动和其他图活动图没有表示出计算处理过程中的全部细节内容。它们表示了活动进行的流程但没表示出执行活动的对象。

活动图是设计工作的起点。为了完成设计，每个活动必须扩展细分成一个或多个操作，每个操作被指定到具体类。这种分配的结果引出了用于实现活动图的对合协的设计工作。

第五章 概念建模与系统分析

在用例分析的基础之上，识别一个丰富的对象集或者概念类集，是面向对象分析向设计过渡的重要工作，做好这项工作，将会在设计和实现期间获得丰富的回报。

5.1 概念建模的思想和方法

概念模型是作为设计软件对象的启发来源，也是后续工件的必须输入。概念模型是说明问题域里（对建模者来说）有意义的**概念类**，它是面向对象分序的时候要创建的最重要的工作。

一、概念建模的思想

什么是“问题域”和“概念建模”？

问题域:

现实世界中系统所要解决问题的概念为“问题域”，如“银行业务”属于“银行的问题域”。

概念建模:

- 我们设计一个系统，总是希望它能解决一些问题，这些问题总是会映射到现实问题和概念。
- **对这些问题进行归纳、分析的过程就是概念建模**（这个域，指的就是问题域）。

建立概念模型的好处：

- 通过建立概念模型能够从现实的问题域中找到最有代表性的概念对象；
- 并发现出其中的类和类之间的关系，因为所捕捉出的类是反馈问题域本质内容的信息。

经典的面向对象的分析或调研的步骤，是把一个相关的概念，分解为单个概念类或者对象（是一个我们能够理解的概念）。概念模型是概念类或者是我们感兴趣的现实对象的可视化表示，它们也被称之为：领域模型、领域对象模型、分析对象模型等。

在 UML 中，概念模型是不定义操作（方法）的一组类图来说明，它主要表达：

- 概念对象或者概念类；
- 概念类之间的关联；
- 概念类的属性。

建立概念模型的目的是建立对象职责分配的基础，对象职责分配必须依靠模型的的建立过程来发现，不建立模型，主要依靠对问题的理解来分配职责往往会得到一个错误的设计。

二、三种概念类

1，边界对象：参与者使用该对象与系统进行交流，也即边界对象代表系统的内部工作和它所处环境之间的交互。

边界对象将系统的其它部分和外部的相关事物隔离和保护起来。其主要的责任是：输入、输出和过滤。

2，实体对象：代表要保存到持续存储体中的信息。实体类通常用业务域中的术语命名。

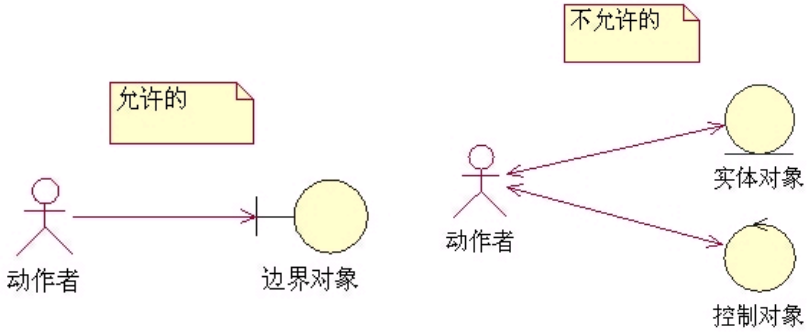
通过它可以表达和管理系统中的信息。在模型中，系统中的关键概念以实体对象来表现。其主要的责任是：业务行为的主要承载体

3，控制对象：它协调其他类的工作，每个用例通常有一个控制类，控制用例中的时间顺序。

它可能是与其它对象协作以实现用例的行为，控制类也称管理类。其主要的责任：控制事件流，负责为实体类分配责任。

有四个规则对应上面的三种分析类对象间的交互

1）用例的参与者只能与边界对象交互（这相当于结构化分析里面的自动化边界）



2）边界对象只能与控制对象和动作者交互（即不能直接访问实体对象）

3）实体对象只能与控制对象交互

4）控制对象可以和边界对象交互，也可以和实体交互，但是不能和动作者交互

三种概念类的 UML 的图示如下：

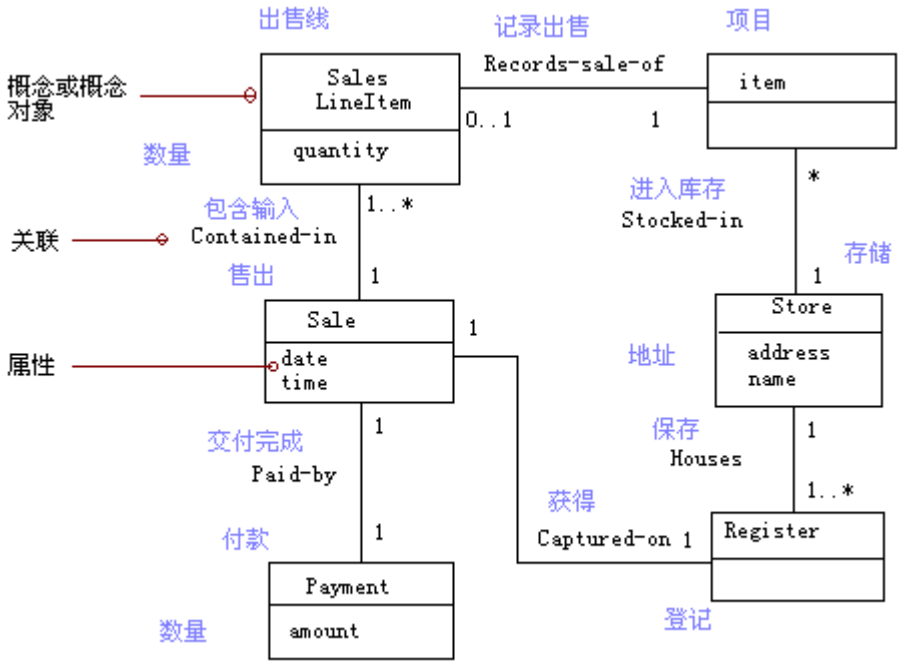
用这种图表达得更多的是一种概念性的行为，也称之为“鲁棒”图。另一方面，在概念建模的时候，我们也常常使用普通的类标示符来表达概念，下面的讨论我们主要使用这种方法。

三、概念建模的简单例子

下面举个简单的例子，说明概念建模的基本概念。

1）问题的描述

例如：两个概念类 Payment（支付）Sale（售出）在概念模型中以一种有意义的方式关联。



2）关键概念

仔细考察上面的图，可以看出，概念模型实际上是可视化了概念中的单词或概念类，并且为这些单词建立了概念类。

也就是说，概念模型是抽象了一个可视化字典。

模型展现了部分视图或抽象，而忽略了建模者不感兴趣的细节。

它充分利用了人类的特点—大脑善于可视化思维。

3）概念模型不是软件组件的模型

概念模型视相关现实世界概念中事务的可视化表示，不是 Java 或者 C#类这样的软件组件。

下面这些元素不适合在概念模型中表述：

1，软件工件（窗口或数据库）

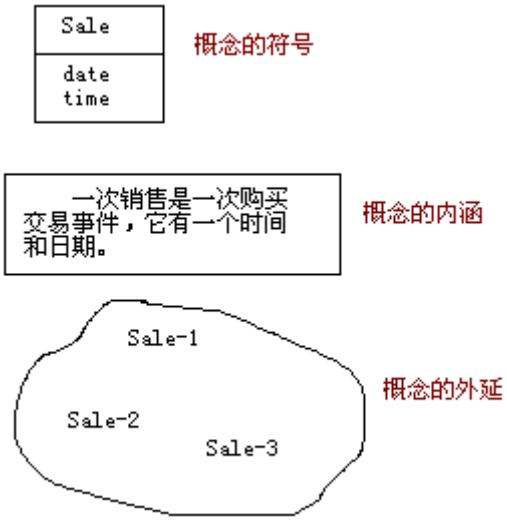
2，职责或者方法：方法是个纯粹的软件概念，在设计工作期间考虑对象职责是非常重要的，但概念模型不考虑这些问题，在这里考虑职责的正确方法是，给对象分配角色（比如收银员）。

4）概念类

概念模型表示概念中的概念类或词汇，一个不是太准确的描述：一个概念类就是一个观点、事务或者对象。比较准确的表达为：

概念类可以按照它的符号、内涵和外延来考虑。

- 符号：代表一个概念类的单词或者图片。
- 内涵：概念类的定义。
- 外延：概念类定义的一组实例。



5.2 概念类的识别

一、识别概念类

我们的目标是在相关概念中创建有意义的概念类，比如说创建“处理销售”用例中的相关概念类。

1，识别概念类的策略

下面提供了两种识别概念类的技巧。

- 使用概念类分类列表。
- 识别名词短语。

2，使用概念类分类列表

通过建立一个候选的概念类的列表，来开始建立模型。下面是一个从商店和航空订票概念中抽取出来的概念列表（注意，排列不考虑重要性）。

概念类分类	示例
物理或具体对象	（略）
事物的设计、描述、或规范	
位置	
交易	
交易项目	
人的角色	
其它事物的容器	
容器包含的元素	
在该计算机之外的其它计算机或电子机械系统	
抽象名词的概念	
组织	
过程（通常不表示一个概念，但可以被表示成一个概念）	
规则和政策	

分类	
有关工作、契约和法律事务的记录	
财务设施及服务	
手册、文档、引用论文、书籍	

3，根据名词短语识别找出概念类

曾经有人提出了用名词短语分析找出概念类的方法，然后把它们作为候选的概念类或者属性。使用这种方法必须十分小心，从名词机械的映射肯定是不行的，而且自然语言中的单词本来就是模棱两可的。不过，这仍然是灵感的另一种来源，比如，我们来看一看原来写出来的“处理销售”的用例：

> **基本流程：**
> 1．顾客携带购买的商品到达 POS 机收费口
> 2．收银员开始一次新的销售
> 3．收银员输入商品标识
> 4．…..
> 重复 3－4 步，直到结束。
> 5．……
> …….
> 10．顾客携带商品和收据离开。

仔细研究其中的名词，可以看到很多有用的概念类（“记账”、“提成”），也可能有些是属性，请研究我们后面要讨论的关于区分属性的和类的讨论。这种方法的缺点就是不精确，但对我们研究问题会非常有用。

推荐：把概念类分类，和词语分析一起使用。

一般来说，用大量细粒度的概念类来充分描述概念模型，比粗略描述要好。

下面是识别概念类的一些指导原则：

- 不要认为概念模型中概念类越少越好，情况往往恰恰相反。
- 在初始识别阶段往往会漏掉一些概念类，在后面考虑属性和关联的时候才会发现它，这是应该把它加上。
- 不要仅仅因为需求中没有要求保留一些概念类的信息，或者因为概念类没有属性，就排除掉这个概念类。

无属性的概念类，或者在问题域里面仅仅担当行为的角色，而非信息的角色的概念类，都可以是有效的概念类。

二、概念类识别的指导原则

1，事物的命名和建模

概念模型是问题域中的概念或这是事物的地图，所以地图绘制员的策略，也适用于概念模型的建模。

- 使用地域中已有的地名（和城市名相同）
- 排除不相关的特性（比如居民人数）
- 不添加不属于某个地方的事物（比如虚构的山川）

以此，我们建议使用如下的原则：

- 给概念模型建模，要使用问题域中的词汇。
- 把和当前不相关的概念类排除在问题域之外。
- 概念模型应该排除不在当前考虑下的问题域中的事物。

2，在识别概念类的时候一个常犯的错误

在建立概念模型的时候，最常犯的一个错误就是把原本是类的事物当作属性来处理。Store（商店）是 Sale（出售）的一个属性呢？还是单独的概念类 Store？大部分的属性有一个特征，就是它的性质是数字或者文本。而商店不是数字和文本，所以 Store 应该是个类。

另一个例子：

考虑一下飞机订票的问题，Destination（目的地）应该是 Flight（航班）的属性呢还是一个单独的类 Airport（包括属性 name）。在现实世界中，目的地机场并不是数字和文本，它是一个占地面积很大的事物，所以应该是个概念类。

建议：

如果我们举棋不定，最好把这样的事物当做一个单独的概念类，因为概念模型中，属性非常少见。

三、分析相似的概念类

有一些情况是比较不太容易处理的。

举个例子，我们来分析一下“Register（记录）”和“POST（终端）”这两个概念。POST 作为一个销售终端，可以是客户端任何终点的设备（用户 PC，无线 PDA），但早期商店是需要一个设备来记录（Register）销售。而 POST 实际上也需要这个能力。可见，Register 是一个更具抽象性的概念，在概念模型中，是不是应该用 Register 而不是 POST 吗？

我们应该知道，概念模型其实没有绝对正确和错误之分，只有可用性大小的区分。

根据绘图员原则，POST 是一个概念中常见的术语，从熟悉和传递信息的角度，POST 是一个有用的符号。但是，从模型的抽象和软件实现相互独立的目标来看，Register 是一个更具吸引力和可用性的表达，它可以方便的表达记录销售位置的概念，也可以表达不同的终端设备（如 POST）。

两种方式各具优点，关键是看你的概念类重点是表达什么信息。这也是一个架构师必备的能力—抓住重点。

四、为非现实世界建模

一些业务概念有自己独特的概念，只要这些概念是在业内被认可的，同样可以创建概念类，比如在电信业可以建立这样的概念类：消息（Message）、连接（Connection）、端口（Port）、对话（Dialog）、路由（Route）、协议（Protocol）等。

五、规格说明或者描述概念类

在概念模型中，对概念类作规格说明的需求是相当普遍的，因此它值得我们来强调。

假定有下面的情形：

- 一个 Item 实例代表商店中一个实际存在的商品
- 一个 Item 表达一个实际存在的商品，它有价格，ID 两个描述信息
- 每次卖掉一个商品，就从软件中删掉一个实例。

如果我们是这样来表达：

那很可能会认为随着商品的卖出，它的价格也删掉了，显然这是不对的。比较好的表达方式是这样的：

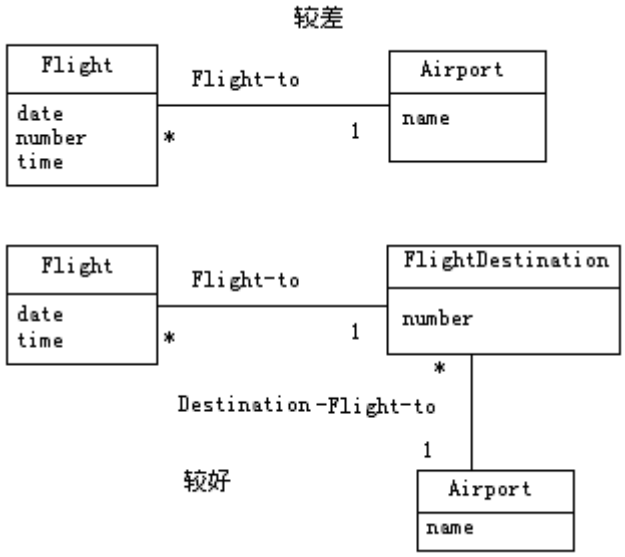
1）何时需要规格说明类

在下面的情况下，需要添加概念类的说明类：

- 商品或服务的信息描述，独立于商品或者服务当前已经存在的任何实例。
- 删除所描述的事物，会导致维护信息的丢失。
- 希望减少冗余或者重复的信息。

2）服务的描述

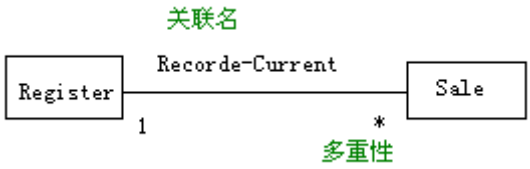
作为概念类的实例可以是一次服务而不是一件商品，比如航空公司的航班服务。假定航空公司由于事故取消了 6 个月的航班，这时它对应的 Flight（航班）软件对象也在计算机中删除了，那么，航空公司就不再有航班记录了。所以比较好的办法是添加一个 FlightDestination（航班目的）的规格描述类，请看下面的例子。



5.3 概念模型的关联

一、找出关联

关联，是类（事实上是实例）指示有意义或相关连接的一种关系。关联事实上表示是一种“知道”。如果不写箭头，关联的方向一般是“从上到下，从左到右”。



我们可以使用下面的表来找出关联

分类	示例
A 在物理上是 B 的一部分	（略）
A 在逻辑上是 B 的一部分	
A 在物理上包含在 B 中/依赖于 B	
A 在逻辑上包含在 B 中	
A 是对 B 的描述	
A 是交易或者报表 B 中的一项	
A 为 B 所知道/为 B 所记录/为 B 所扑获	
A 是 B 的一个成员	
A 是 B 的一个组织子单元	
A 使用或者管理 B	
A 与 B 通信	
A 与一个交易 B 有关	
A 是一个与另一个 B 有关的事物	
A 与 B 相邻	
A 为 B 所拥有	
A 是一个与 B 有关的事件	

二、关联的指导原则

- 把注意力集中在那些需要把概念之间的关系信息保持一段时间的关联（“需要知道”型关联）。
- 太多的关联不但不能有效的表示概念模型，分而会使概念模型变的混乱，有的时候发现某些关联很费时间，但带来的好处并不大。
- 避免显示冗余或者导出的关联。

三、角色和多重性

关联的每一端称之为“角色”。角色可选的具有：名称、多重性表达式、导航性。

多重性：

多重性表示一个实例，在一个特定的时刻，而不是一段时间内，可以和多个实例发生关联。

“*”表示多个。

“1”表示一个。

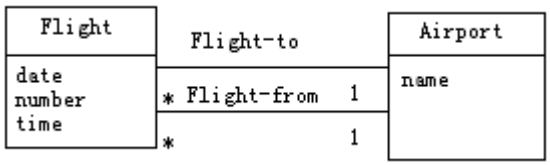
“0..1”表示 1 或者没有，比如一个商品在货架上，可能售出，也可能被丢掉了，这种情况，用“0..1”是合理的。问题是我们需要关心这样的观点吗？如果是数据库，可能表达这个数据存在，或者损坏。但在概念模型并不表示软件对象，通常我们只对我们有兴趣的内容建模，从这个观点出发，也可能只有“1”或者“*”是合理的。

再一次提醒，发现概念类比发现关联更重要，花费在概念模型创建的大部分时间，应该被用

于发现概念类，而不是关联。

四、两种类型之间的多重关联

两种类型之间的多重关联是可能存在的。比如航空公司的例子，Flight-to 和 Flight-from 可能会同时存在，应该把它们都标出来。



5.4 概念模型的属性

发现和识别概念类的属性，是很有意义的。属性是个逻辑对象的值。属性主要用于保留对象的状态。

一、有效的属性类型

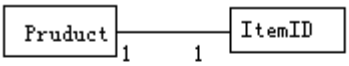
大部分属性应该是简单数据类型。当然也可以使其它的一些必要的类型，比如：Color（颜色）、Address（地址）、PhoneNumber（电话号码）等。

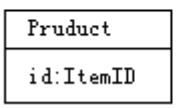
二、非原始的数据类型类

在概念类中，可以把原始数据类型改成非原始数据类型，请应用下面的指导原则：

- 由分开的段组成数据（电话号码，人名）
- 有些操作和它的数据有关，如分析和验证等（社会安全号码）
- 包含其它属性的数据（促销价格的开始和结束时间）
- 带有单位的数据值（支付金额有一个货币单位）
- 对带有上述性质的一个或多个抽象（商品条目标识符）

如果属性是一个数据类型，应该显示在属性框里面。





这两种表示都是可以的。

5.5 泛化建模

泛化和特化是概念建模的基本概念，另外，概念类的层次，往往是软件类层次的基本源泉，

软件类可以利用继承来减少代码的重复。

一、概念模型的概念提取

概念模型是在不断考虑迭代需求的相关概念的过程中发展起来的。很多人对概念建模都有一些细腻的建模问题的讨论。比如，有一个有一定作用的概念，就是概念分类表（Concept Category List）。

1，概念分类表

本次迭代所涉及的一些显著概念，可以列出一个表来。

类别	示例
物理或者实际的对象	CreditCard（信用卡），Check（支票）
事物的说明、设计或描述	
位置	
交易	CashPayment（现金支付） CreditPayment（信用卡支付） Check Payment（支票支付）
交易的项目	
人的角色	
其它事物的容器	
容器中的事物	
系统之外其它计算机或电子机械系统	CreditAuthorizationService（信用卡支付授权服务） CheckAuthorizationService（支票支付授权服务）
抽象名词概念	
组织	CreditAuthorizationService（信用卡支付授权服务） CheckAuthorizationService（支票支付授权服务）
事件	
规则及策略	
目录	
财务、工作、合约、法律事务的记录	AccountsReceivable 账目接受
金融工作和服务	
手册、书籍	

2，从用例中得到名词对照的概念

再次重申，不能机械的用名词与概念的对照来识别概念模型的有关概念。由于自然语言的模棱两可，文本中的相关概念并不总是明确和清晰的，因此我们必须判断并作合适的抽象处理。不过，由于名词和概念对照的直观性，它仍然是概念建模的一个实用技术。

每一次迭代，实际上都要反过来研究用例并进行需求分析，利用需求驱动项目进一步精化。在这次迭代中，我们将处理销售过程（Process Sale）用例的信用卡和支票的支付场景，下面显示扩展场景中一些名词对照的概念。

用例 1：Process Sale

……

扩展：

7b. 信用卡支付

1. 顾客输入**信用卡账号**信息。
2. 系统向外部**信用卡支付授权服务**系统发出**授权支付**请求和**批准支付**的请求
   - 2a. 系统检测到和外部信用卡授权服务系统通信故障
     1. 系统通知收银员发生了错误
     2. 收银员向客户请求更换支付方式
3. 系统收到**支付批准**并通知收银员

3a. 系统收到**支付拒绝**的通知
1．系统通知收银员系统拒绝支付
2．收银员要求顾客改变支付方式
4．系统记录**信用卡的支付情况**，包括支付授权
5．系统出示信用卡签名的输入方式
6．收银员要求顾客为信用卡支付签名，顾客签名
7c. 用支票支付
1． 顾客签发**支票**，将**支票**和**驾驶执照**一起交收银员
2． 收银员把驾驶执照号码记录在支票上，将其输入，发出**支票支付授权请求**。
3． 生成一个**支票支付请求**，将其发送到外部的**支票授权系统**。
4． 系统收到支票支付授权并通知收银员。
5． 系统记录**支票的支付情况**，其中包括**支付批准**。
……

其中，粗体的就是我们发现的概念名词。

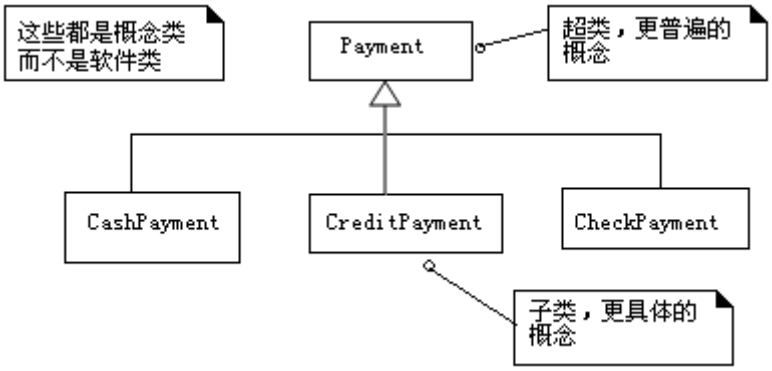
3，授权服务的交易

由名词对照，我们还可以发掘出诸如 CreditPaymentRequest（信用卡付款请求）以及 CreditApprovalReply（信用卡批准应答）这样一类概念，这些概念都可以视为外部服务的不同类型的交易。一般来说，识别这些交易非常有用，因为许多活动和处理都是围绕交易而进行的。

这些交易概念，不需要代表计算机中的记录，也不需要代表连线上的比特数据，它们代表了独立于执行方式的交易的抽象。比如，信用卡支付方式可以通过打电话或者计算机发出请求来完成。

二、泛化及其应用

CashPayment（现金支付）、CreditPayment（信用卡支付）和 Check Payment（支票支付）这几个概念非常接近，可以组织成一个泛化的类层次，其中超类 Payment（支付）具有更普遍的概念，而子类是一个更具体的概念。注意这里讨论的是概念类，而不是软件类。



泛化（generalization）是在多个概念之间识别共性，定义超类和子类关系的活动，它是构件概念分类的一种方式，并且在类的层次中得到说明。在概念模型中，识别超类和子类及其有价值，因为通过它们，我们就可以用更普遍、更细化和更抽象的方式来理解概念。从而使概念的表达简约，减少概念信息的重复。

三、定义概念性超类和子类

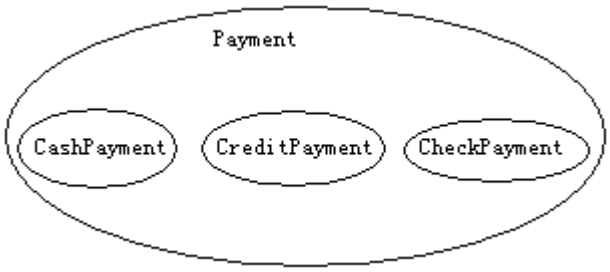
由于识别概念性的超类和子类具有价值，因此根据类定义和类集，准确的理解泛化、超类、子类是很有意义的，下面我们将讨论这些概念。

1，泛化和概念性类的定义

定义：一个概念性超类的定义，比一个概念性子类的定义更为普遍或者范围更广。
在前面的例子中，Payment（支付），是一个比具体的支付方法更为普遍的定义。

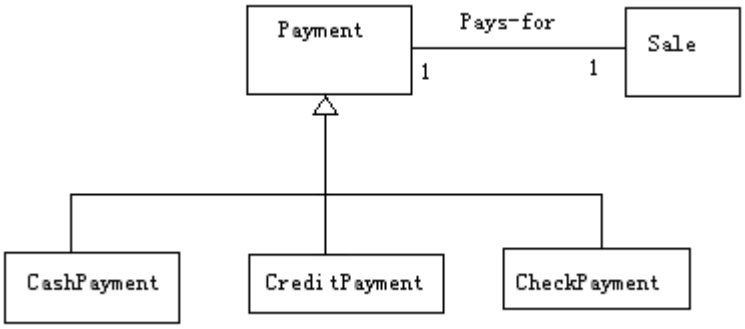
2，泛化与类集

概念性子类与概念性超类，在集的关系上是相关的。所有概念性子类集的成员，都是它们超类集的成员。



3，概念性子类定义的一致性

一旦创建了类的层次，有关超类的声明也将适用于子类。一般的说，子类和超类一致是一个“100%规则”，这种一致包括“属性”和“关联”。



4，概念性子类集的一致性

一个概念性子类应该是超类集中的一个成员。通俗的讲，概念性子类是超类的一种类型（is a kind of），这种表达也可以简称为 is-a。这种一致性称之为 is-a 规则。

所以，这样的陈述是可以的：“信用卡支付是一个支付”（CreditPayment is a Payment）。

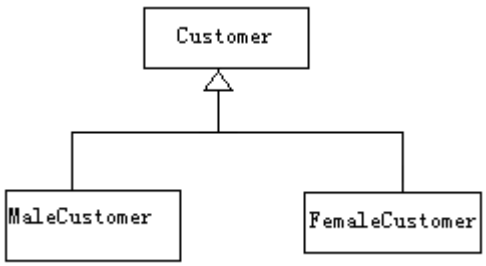
5，什么是正确的概念性子类呢

从上面的讨论，我们可以使用下面的测试，来定义一个正确的子类：

- 100%规则（定义的一致性）
- is-a 规则（集合成员关系的一致性）

6，何时定义一个概念性子类

举个例子，把顾客（Customer）划分为男顾客（MaleCustomer）和女顾客（FemaleCustomer），从正确性来说是可以的，但这样划分有意义吗？



这样划分是没有意义的，因此我们必须讨论动机问题。把一个概念类划分为不同子类的强烈动机为：

当满足如下条件之一的时候，为超类创建一个概念性的子类：

- 子类具有额外的相关属性。
- 子类具有额外的相关关联。
- 子类在运行、处理、反应或者操作等相关方式上，与超类或者其它子类不同。
- 子类代表一个活动的事务（例如：动物、机器人），它们与超类的其它子类在相关的行为方式上也不同。

由此看来，把顾客（Customer）划分为男顾客（MaleCustomer）和女顾客（FemaleCustomer）是不恰当的，因为它们没有额外的属性和关联，在运行（服务）方式上也没什么不同。尽管男人和女人的购物习惯不同，但对当前的用例来说不相关。这就是说，规则必须和我们研究的问题相结合。

7，何时定义一个概念性超类

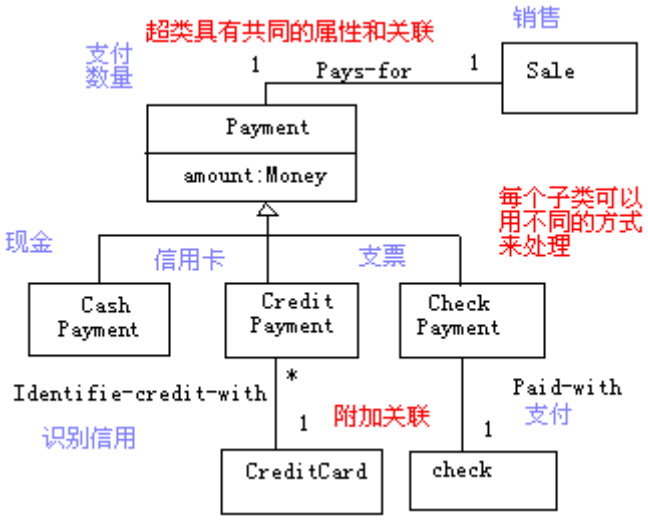
在多个潜在的子类之间，一旦发现共同特征，就可以暗示可以泛化得到一个超类。

下面是泛化和定义超类的动机：

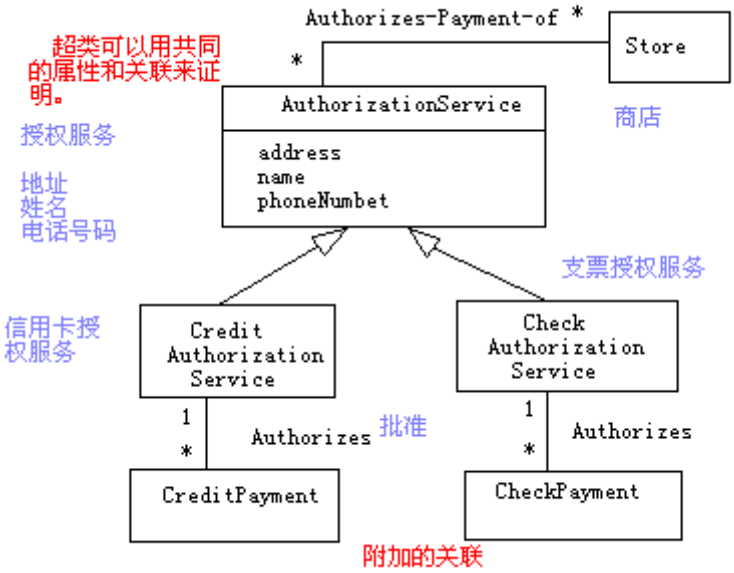
- 潜在的概念子类代表一个相似概念的变体。
- 子类遵守 100%的 is-a 规则。
- 所有的子类具有共同的属性，可以提取出来并在超类中表示。
- 所有子类具有相同关联，可以提取并与超类相关。

8，发现概念类的实例

Payment 类：



授权服务类：



注意，在构造超类的时候，层次不宜太多，关键是表达清晰。事实上，额外的泛化不会增加明显的价值，相反带来很大的负面影响，没有带来好处的复杂性是不可取的。

四、抽象概念类

在概念模型里面，识别抽象类是有用的，因为它们限制了哪些类可能具有具体的实例。如果一个类的成员，必须是它子类的成员，那么称它为抽象概念类。在上面的例子里，Payment 必须用更具体的 CreditPayment、CheckPayment 做实例，而 Payment 本身并不能实例化，所以 Payment 是一个抽象的概念类。

5.6 精化概念建模及若干难以确定的要素

在前面讨论的基础上，我们需要把概念模型进一步的精化。

一、关联类

下面的概念需求，需要考虑关联类的问题：

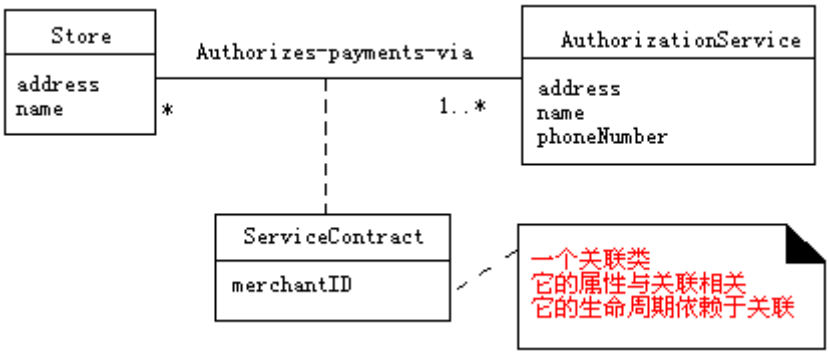
- 在通讯过程中，授权服务为每个商店分配一个店主 ID。
- 从商店发送到授权服务的支付请求，需要店主的 ID 以标识这个商店。
- 进一步，对每一个授权服务，每一个商店都有一个不同的店主 ID。

在 UP 的概念建模中，商店的 ID 的属性到地方在什么地方呢？把 merchantID（店主 ID）置于 Store（商店）类中是不正确的，因为一个 Store 类会有多个 merchantID 值。

注意，属性表示了对象的状态，因此，一个对象的属性，同时只能有一个值。

把 merchantID 置于 AuthorizationService（授权服务）类中也是不恰当的，因为它同样也有多个值。通过上面的讨论，我们就可以得出下面的建模原则：

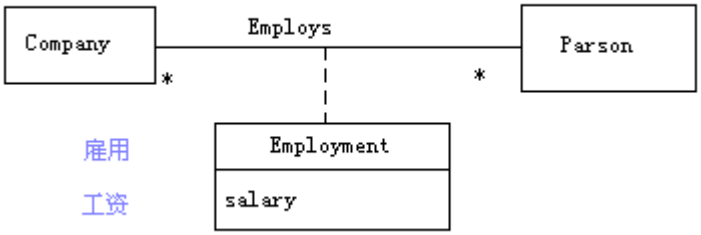
在概念建模中，如果一个类 C 的属性 A 同时具有多个值，那么不要把属性 A 放到类 C 中，而是把属性 A 放到与 C 关联的关联类里面去。



出现下面的情况可以在概念模型中加入关联类：

- 一个属性和关联有关。
- 关联类的生命周期依赖于关联。
- 在两个概念之间存在多对多的关联或者关联本身具有某些待表示的信息。

下面的例子，表达一个人可能受雇于多个公司。



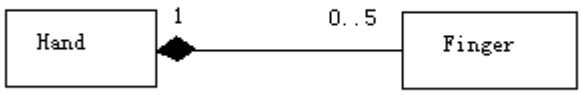
二、聚集和组合

聚集（Aggregation）是用于整体—部分关系建模的一种关联，整体称之为**组合**（composite）。

1）组合聚集

组和聚集用实心菱形表达，原来的意义，它表达部分的存在依赖于组合。

比如：一只手和手指，手指的存在依赖于手。



在设计模型中，组合的含义很清楚，它表达某一个对象的生命周期，依赖于另一个对象（全局性的定义，并且在构造的同时实例化）。但是在概念模型中，这种整体的创建对部分的影响并不太清楚（一个实际的销售并不可能创建实际的销售商品），所以，花功夫发现这种关系并没有什么实际意义。

2）共享聚集

共享聚集的图是空心菱形，它表示组合末端（菱形处）的多重性可能大于一。它的含义是，部分可能同时属于多个组合实例。在设计模型中，这种共享聚集和关联有几乎相同的实现方法（全局性的定义，但对象构造的时机待定）。但是，在实际的物理集合中，很少能找到相似的例子。

注意，发现和使用聚集在设计模型中相当有意义，但与设计模型不同，在概念模型中识别和说明聚集并不会产生深远影响。学院派的建模理论花了很多时间来讨论这些关联上的细腻差别，但是，很多经验丰富的建模者最终发现，它们在关联的细微含义上浪费了太多的无谓的时间。所以，在概念模型的关联问题上，我们将排除这种表示方式。

三、案例：订单处理子系统

我们还是以网上电源设备销售子系统的案例，来讨论概念建模的过程。

概念建模的目的，是用类来表达一个对象集，如果这个类是长久存在的（或者说是持久的）一个业务实体，比如，Customer、Order、Shipment 等等，这样的类通常被称为**实体类**。实体类往往代表着一个数据库的一个持久化对象。在这个问题域里面，我们将关注实体模型的建立，希望这个结果直接影响到数据库设计。

实体类定义了任何信息系统本质，所以需求分析上很大程度上是发现实体对象。不过，从功能上来说，发现其它的功能类也是必要的。有时候这样类的建模一直需要延续到设计阶段。一般来说，实体类的发现，需要在整体功能需求中发现。而实现类的发现，需要在用例文档中来发现。

1，从需求中寻找类

下面我们从功能需求的表中找到这个类。

编号	需求	实体类
1	**客户**使用制造商的 Web 页面查看所选择的**电源**设备标配，同时显示价格。	Customer 客户 ES: StandardConfiguration 标准配置 Product 产品
2	**客户**查看配置细节，可以更改配置，同时计算价格。	Customer 客户 ConfiguredES: ConfiguredProduct 配置的电源：配置的产品 ConfigurationItem 配置项目
3	**客户**选择订购，也可以要求**销售人员**在**订单**真正发出之前和自己联系，解释有关细节。	Customer 客户 ConfiguredES 配置的电源 Order 订单

		Salesperson 销售人员
4	要发出订单，**客户**必须填写表格，包括地址，付款细节（信用卡还是支票）等。	Customer 客户 Order 订单 Salesperson 销售人员 Shipment 出货 Invoice 发票 Payment 付款
5	客户订单送到系统之后，**销售人员**发送电子请求到**仓库**，并且附上配置细节。	Customer 客户 Order 订单 Salesperson 销售人员 ConfiguredES 配置的电源 ConfigurationItem 配置项目
6	事务的细节（包括订单号和客户帐户号），**销售人员**要 e-mail 给客户，使得**客户**可以在线查询订单状态。	Customer 客户 Order 订单 OrderStatus 订购状况
7	**仓库**从**销售人员**处获取发票，并且向客户运送电源设备	Invoice 发票 Shipment 出货

寻找概念类是一个迭代式的任务，需要反复思索，也可以试着回答下面的问题：

- 这个概念是一个数据容器吗？
- 它有取不同值的不同属性吗？
- 它已经有实体对象了吗？
- 它在应用概念的范围之内吗？

针对这张表我们还可以思考很多问题，比如：

1，ConfiguredES（自己选择配置的电源）和 Orde（订单）的区别到底在哪里？毕竟我们不打算存储选择配置的电源，除非它的订单被提交。

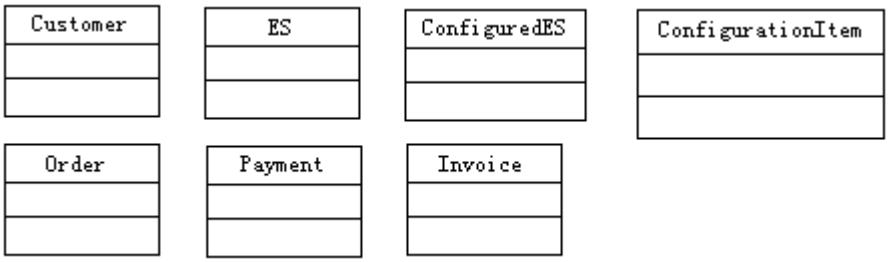
2，第 4 号和第 7 号需求中的 Shipment（出货）的含义是一样的吗？可能不一样，如果我们知道了运送是仓库的责任，这个类是不是还需要呢？

3，ConfigurationItem （配置项目）为什么不能是 ConfiguredES 中的一组属性呢？

4，OrderStatus（订购状况）为什么不能是 Order（订单）的一个属性呢？

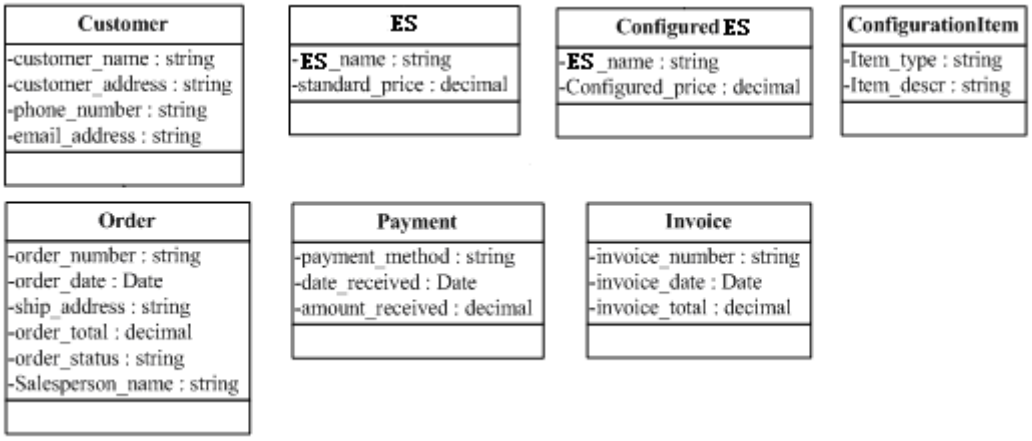
5，Salesperson（销售人员）是一个类，还是 Order（订单）还是 Invoice（发票）中的一个属性呢？

回答这些问题是不容易的，这需要对需求进行深入的研究，假定我们研究的结果出现了下面这些类（其中 Customer 实际上是用例中的参与者，所以是从用例的角度出现的）。



2，确定属性

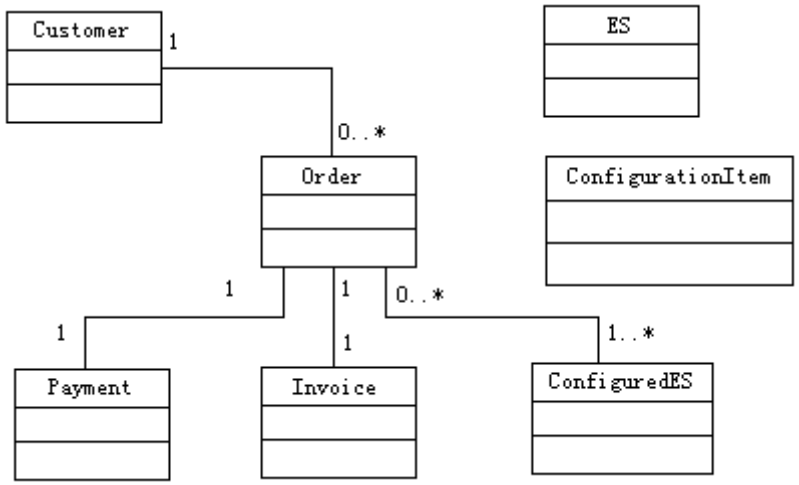
我们可以进一步定义一些属性，这些属性首先被想到的是原始类型的属性，其实定义属性确实是有很大的随意性的，这需要一些经验。



3，发现关联

我们可以根据需求的理解，来发现一些关联。在确定多重性的时候，可以作一些假定：

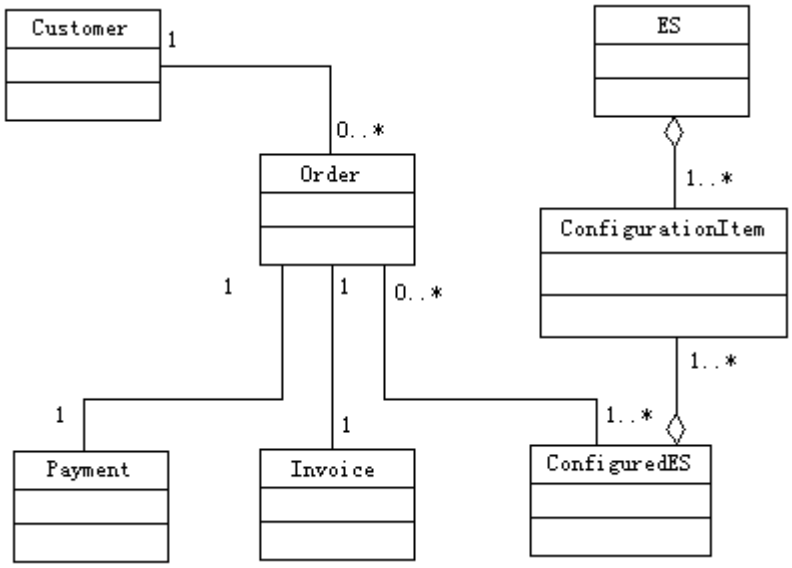
订单来自于单个客户，而客户可以提交多个订单。订单除非在付款已经被说明之后才被接受，因而是一对一关联。订单不一定要有一个所关联的发票，但发票总是和单个订单所联系。一个订单是为一个或者多个自配置电源建立的，一个自配置电源可以被订购多次或者一次也没有。这样就可以画出关联来。



4，发现聚集

聚集是关联更强的形式出现的，不过事实上并不一定需要刻意的发现聚集，这里用聚集表达主要是为了强调这种关联的重要。

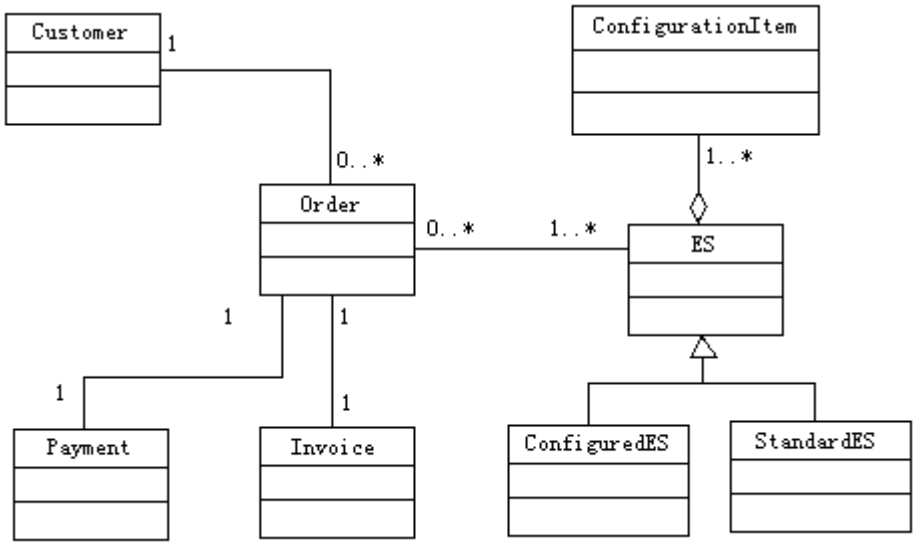
一个电源具备一个或者多个配置项目。同样，一个自定义配置的电源也具有一个或者多个配置项目。



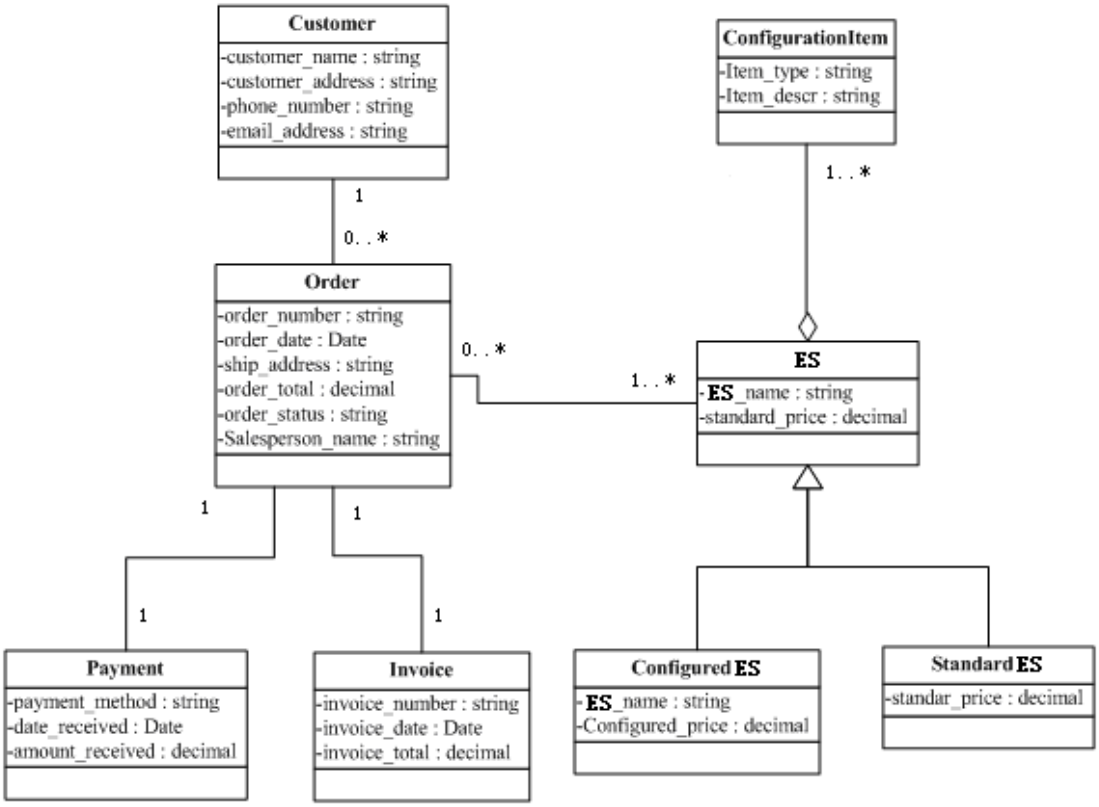
5，泛化建模

面向对象的分析很重视泛化建模，这样一来可以大大简化和清晰化所建立的模型。

在这里，电源（ES）变成了一个更抽象的类，两个子类为“标准配置电源”和“自定义配置的电源”。



6，包括属性的最后结果



5.7 系统行为分析

对一个软件应用程序进行逻辑设计之前，对系统进行研究，并把它的行为当作“黑箱”来考虑是有益的。**系统行为**描述一个系统做什么，而不解释系统如何做。描述系统行为一部分是靠顺序图，另外一部分是靠用例和系统契约（以后会加以讨论）。

交互视图主要包括顺序图（Sequence Diagram）和协作图（Collaboration Diagram），主要解决描述对象之间的交互问题。对象间的相互作用体现了对象的行为。

这种相互作用可以描述成两种互补的方式：

1）以独立的对象为中心进行考察；

2）以互相作用的一组对象为中心进行考察。

状态图的描述范围不宽，但它描述了对象深层次的行为，是单独考察每一个对象的“微缩”视图。对状态图的说明是精确的并且可直接用于代码。然而，在理解系统的整个功能时存在困难，因为状态图一个时刻只集中描述一个对象，要确定整个系统的行为必需同时结合多个状态图进行考察。交互视图更适合于描述一组对象的整体行为。交互视图是对象间协作关系的模型。

协作：

协作描述了在一定的语境中一组对象以及用以实现某些行为的这些对象间的相互作用。它描述了为实现某种目的而相互合作的“对象社会”。

交互：

交互是协作中的一个消息集合，这些消息被类元角色通过关联角色交换。当协作在运行时，受类元角色约束的对象通过受关联角色约束的连接交换消息实例。交互作用可对操作的执行、用例或其他行为实体建模，交互模型的建立过程即是对象职责分配的过程。

消息是两个对象之间的单路通信，从发送者到接收者的控制信息流。消息具有用于在对象间

传值的参数。消息可以是信号（一种明确的、命名的、对象间的异步通信）或调用（具有返回控制机制的操作的同步调用）。

创建一个新的对象在模型中被表达成一个事件，这个事件由创建对象所引起并由对象所在的类本身所接受。

创建事件：作为从顶层初始状态出发的转换的当前事件。对于新实例是可行的。

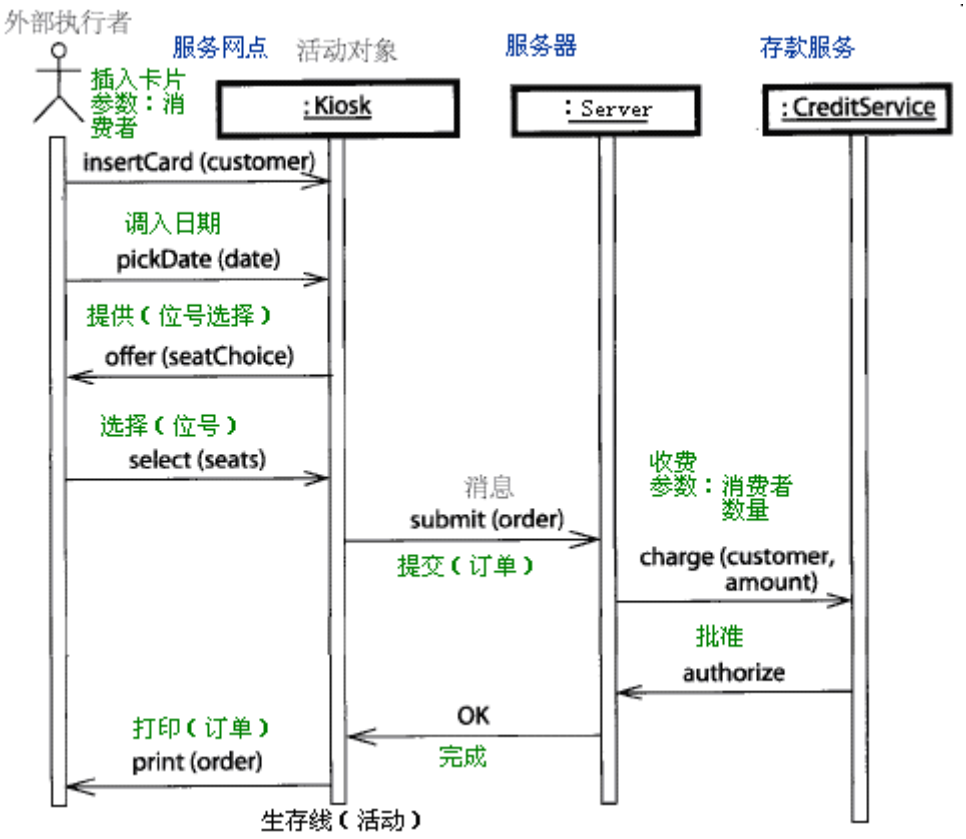
消息可以被组织成顺序的控制线程。分离的线程代表并发的几个消息集合。线程间的同步通过不同线程间消息的约束建模。同步结构能够对分叉控制、结合控制和分支控制建模。

消息序列可以用两种图来表示：顺序图（突出消息的时间顺序）和协作图（突出交换消息的对象间的关系）。

一、顺序图

顺序图将交互关系表示为一个二维图。纵向是时间轴，时间沿竖线向下延伸。横向轴代表了在协作中各独立对象的类元角色。类元角色用生命线表示。当对象存在时，角色用一条虚线表示，当对象的过程处于激活状态时，生命线是一个双道线。

消息用从一个对象的生命线到另一个对象生命线的箭头表示。箭头以时间顺序在图中从上到下排列。下图为带有异步消息的典型的顺序图，这是一个服务网点的存款服务的用例。



二、激活

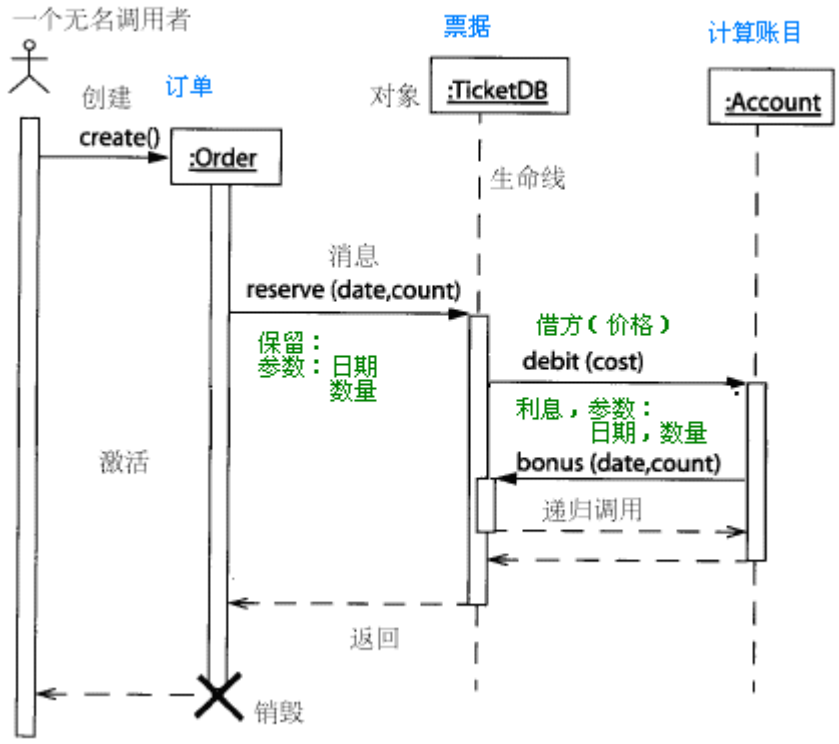
激活是过程的执行，包括它等待嵌套过程执行的时间。在顺序图中它用部分替换生命线的双道线表示。调用用指向由这个调用引起的激活的上部的箭头表示。当控制流程重新进入对象中的

一个操作递归时，递归调用发生，但是第二个调用是与第一个调用分离的激活。同一个对象中的递归或嵌套调用用激活框的叠加表示。

下图为含有过程控制流的一个顺序图，包括一个递归调用和一个对象的创建。

主动对象是激活栈中一组激活对象中的根对象。每个主动对象有由它自己的事件驱动控制线程，控制线程与其他主动对象并行执行。被主动对象所调用的对象是被动对象。它们只在被调用时接受控制，而当它们返回时将控制放弃。

如果几个并行控制线程有它们自己的利用嵌套调用的过程控制流，那么不同的线程必须用不同的线程名、颜色或其他方式辨别，以避免当两个线程在同一个对象中产生混乱。通常，在一个单独的图中最好不要混合使用过程调用和信号。



带有激活的顺序图

三、网上书店子项目案例

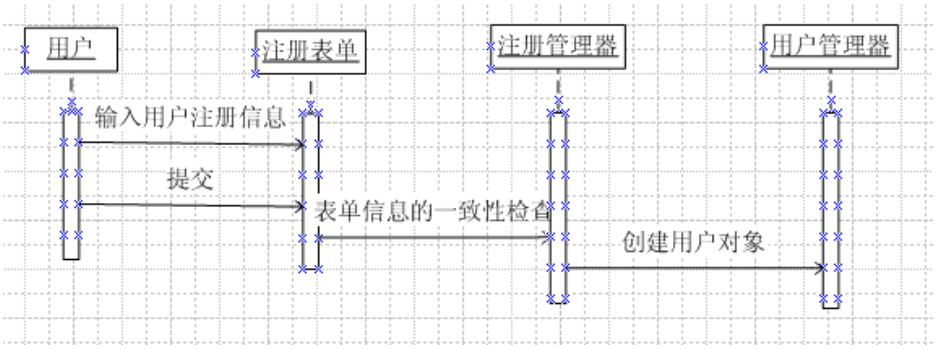
1）用户登录的用例事件流说明

用例名称：	用户登录
参与者：	用户
前置条件：	在此用例开始前，浏览者必须已拥有合法的用户名及密码。
后置条件：	用户名称和密码必须与数据库表中的对应字段匹配，包括字符的大小写。
主事件流	1、用户输入正确的用户名称和密码 2、后台业务组件进行验证 3、安全登录后，转向到可浏览的页面。
扩展事件流	1a、如果未输入用户的名称或者密码而是直接提交，则提示输入用户登录标识的文字信息，并再次跳转到登录页重新登录。 2a、如果用户的名称或者密码不能匹配，则提示登录失败的文字信息，并再次跳转到登录页重新登录。

2）用户登录的用例事件流说明的序列图

作为一种交互图，序列图显示参与交互作用的主角或对象，以及它们生成的按时间排序的事件。通常，序列图显示特定用例实例产生的事件。

序列图是一个二维图形，在其水平方向为对象维，沿水平方向排列参与交互的对象类角色；而纵向维代表时间，沿垂直向下方向按时间递增顺序列出各对象类角色所发出和接收的消息。



3）用户注册

用例名称：	用户注册
参与者：	用户
前置条件：	无（由于用户注册页是公开的，因此每位访问者都可以进行用户注册）。
后置条件：	用户所输入的个人信息满足本应用的各种格式要求并且在数据库表中没有重名的用户存在。
主事件流	1、用户输入个人的信息（包括用户名称和密码等） 2、后台业务组件首先获得用户的个人的信息 3、识别用户的名称是否已经存在 4、满足条件后。最后将用户的信息保存到数据库表中 5、注册成功后，提示用户并转向到用户登录页进行登录。
扩展事件流	1a、如果用户输入的个人信息不完整或者密码和确认密码不相同，则提示输入正确的注册信息的文字信息，并再次跳转到注册页重新注册。 1b、如果用户输入的个人信息为空，直接点击提交也将提示输入正确的注册信息的文字信息，并再次跳转到注册页重新注册。 3a、如果用户输入的用户名称已经在本网站中存在（重名），也将提示输入正确的注册信息的文字信息，并再次跳转到注册页重新注册。

这个用例的事件序列图，可以由学员自己画出。

4）管理员管理维护用户的信息

用例名称：	用户信息的维护
参与者：	管理员
前置条件：	在本用例开始前，管理员要登录到本系统中
后置条件：	用户的信息能够正确地被修改
主事件流	……
扩展事件流	……

请书写相应的主事件流和扩展事件流，并且画出相应事件序列图。

四、协作图（Collaboration Diagram）

虽然用序列图可以很好的表达操作和行为，而且时间关系也很清楚，但是，当关系非常复杂的时候，往往图就很复杂，这样反而不利于看清问题，所以，某些情况下使用协作图将会使表达更清楚。

协作图（也称合作图、交互图）是一种类图，它包含类元角色和关联角色，而不仅仅是类元

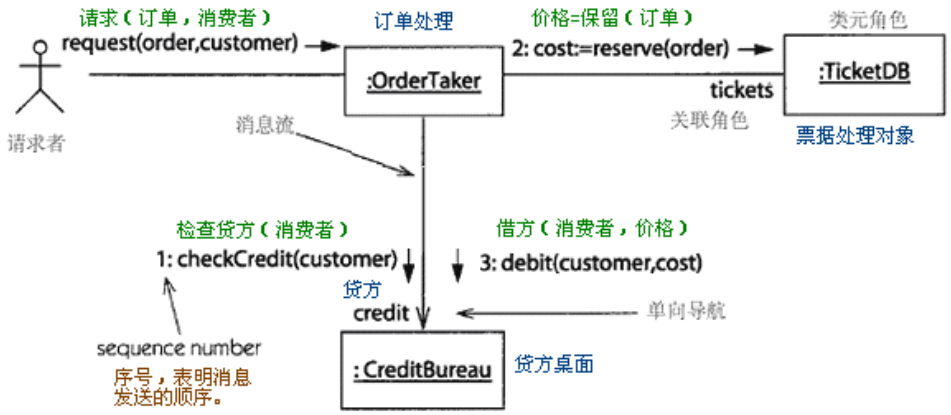
和关联。类元角色和关联角色描述了对象的配置和当一个协作的实例执行时可能出现的连接。

换而言之，协作图只对相互间具有交互作用的对象和对象间的关联建模，而忽略了其他对象和关联。注意，我们并不需要对系统所有的部分绘制协作图，而只是对最主要最复杂的部分做这样的讨论。

当协作被实例化时，对象受限于类元角色，连接受限于关联角色。关联角色也可以被各种不同的临时连接所担当，例如过程参量或局部过程变量。

连接符号可以使用构造型表示临时连接（《 parameter 》或《 local 》）或调用同一个对象（《 self 》）。虽然整个系统中可能有其他的对象，但只有涉及到协作的对象才会被表示出来。

下图为一个协作图。



可以将对象标识成四个组：

存在于整个交互作用中的对象；

在交互作用中创建的对象（使用约束 {new} ）；

在交互作用中销毁的对象（使用约束 {destroyed} ）；

在交互作用中创建并销毁的对象（使用约束 {transient} ）。

设计时可以首先表示操作开始时可得的对象和连接，然后决定控制如何流向图中正确的对象去实现操作。

虽然协作直接表现了操作的实现，它们也可以表示整个类的实现。在这种使用中，它表示了用来实现类的所有操作的语境。这这使得对象在不同的操作中可以担当多种角色。这种视图可以通过描述对象所有操作的协作的联合来构造。

1，消息

消息可以用依附于链接的带标记的箭头表示。每个消息包括一个顺序号、一张可选的前任消息的表、一个可选的监护条件、一个名字和参量表、可选的返回值表。顺序号包括线程的名字（可选）。同一个线程内的所有消息按照顺序排列。除非有一个明显的顺序依赖关系，不同线程内的消息是并行的。各种实现的细节会被加入，如同步与异步消息的区别。

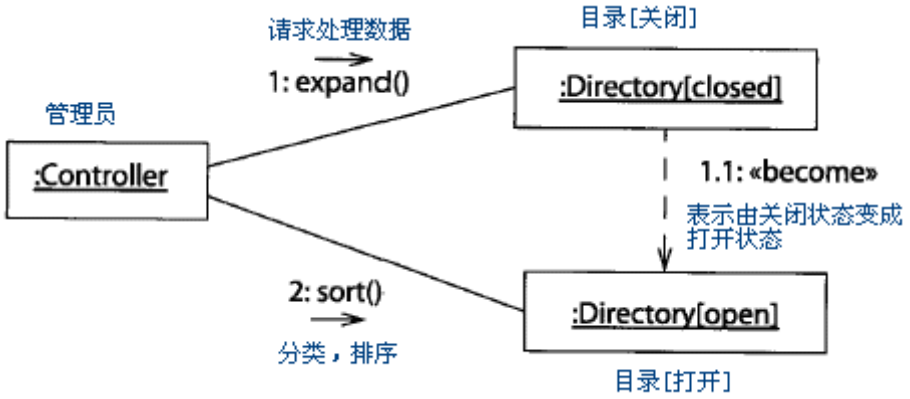
2，流

通常，在完整的操作中协作图包含对象的符号。然而，有时对象具有不同的状态并且必须弄明确表达出来。

例如，一个对象可以改变位置，或者在不同的时刻它的关联有很大区别。对象可以用它的类与它所处的状态表示即具有状态类的对象。同一个对象可以表示多次，每次有不同的位置和状态。

代表同一对象的不同对象符号可以用变成流联系起来。

变成流是从一个对象状态到另一个的转换。它用带有构造型《 become 》的箭头表示，并且可以用顺序号标记表示它何时出现（如下图所示）。



变成流也可以用来表示一个对象从一个位置到另一个位置的迁移。

构造型《 copy 》不经常出现，它表示通过拷贝另一个对象值而得到的一个对象值。

下表表示了几种对象流的关系。

流	功能	表示法
变成	从一个对象值变化到另一个对象值	<<become>>
拷贝	拷贝一个对象，从此以后，该对象为独立对象	<<copy>>

3，协作图与顺序图

协作图和顺序图都表示出了对象间的交互作用，但是它们侧重点不同。顺序图清楚地表示了交互作用中的时间顺序，但没有明确表示对象间的关系。协作图清楚地表示了对象间的关系，但时间顺序必须从顺序号获得。顺序图常常用于表示方案，而协作图用于过程的详细设计。

五、案例：订单处理子系统

交互图很多情况下是对于活动图的深入研究构建起来的，比如对于已经讨论的设备购置案例，我们仔细的研究活动图，对“显示当前配置”的活动作更加细腻的研究。它牵涉到三个类：

ConfigurationWindow，这是一个界面类；

ES 类，这是 ConfiguredES 类或者 StandardES 类。

ConfigurationItem 类，这是一个配置项目类。

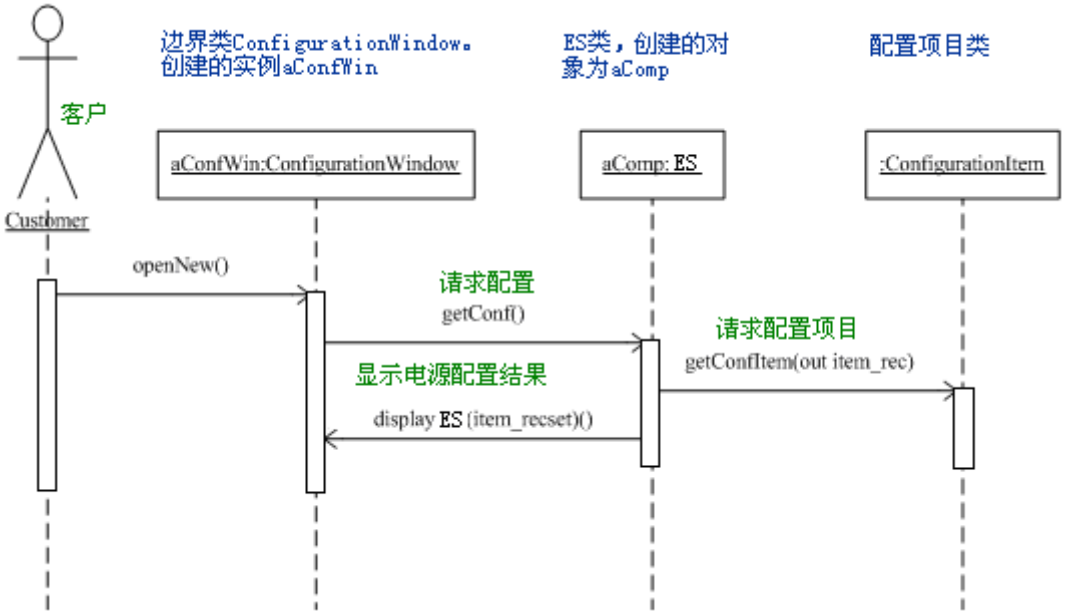
需要表达的交互操作关系如下：

外部参与者 Customer 先选择电源的配置，然后消息 openNew()发送给一个界面类 ConfigurationWindow，这个消息导致创建一个实例 aConfWin。

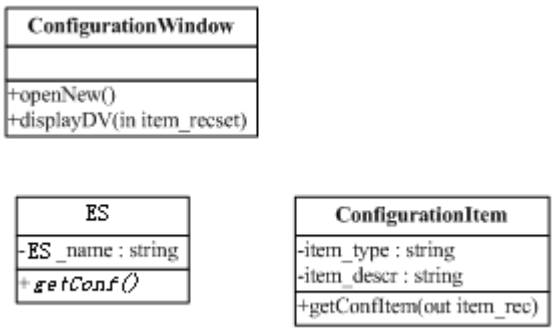
对象 aConfWin 需要“显示它自己”以及配置数据，为了这个目的，它发送一个消息 getConf()给 ConfiguredES 类或者 StandardES 类，并且实例化一个对象 aComp。

对象 aComp 使用输出参数 item_rec 根据 ConfigurationItem 对象“组合它自己”，然后批量发送配置项到 aConfWin。消息 displayES 的参数为 item_recset。

现在对象 aConfWin 能显示自己了，对应的交互图如下。



对这样的序列图的研究，可以为相应的类提供方法打下基础，比如受影响的三个类可以构造相应的方法。



一般来说，可以为每个用例构造一个单独的序列图，这种对象之间相互关系的研究，为好的建模提供了重要的基础。下面，我们再为“订购配置了的电源设备”这个用例，创造序列图，这个序列图表达了跨越几个用例的行为。

描述：

在开始两个消息的作用，我们在上面的序列图已经说明过了。

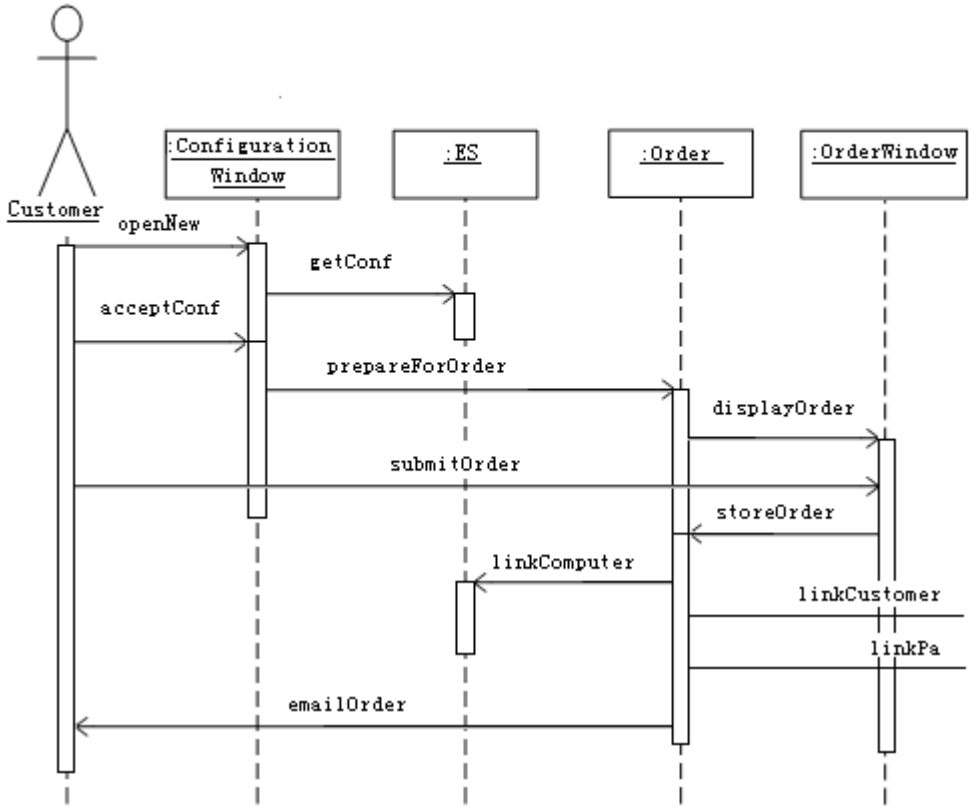
消息 acceptConf 产生发送给:Order 对象的 prepareForOrder 消息，这就创建了一个 Order 对象，它在:OrderWindow 中显示。

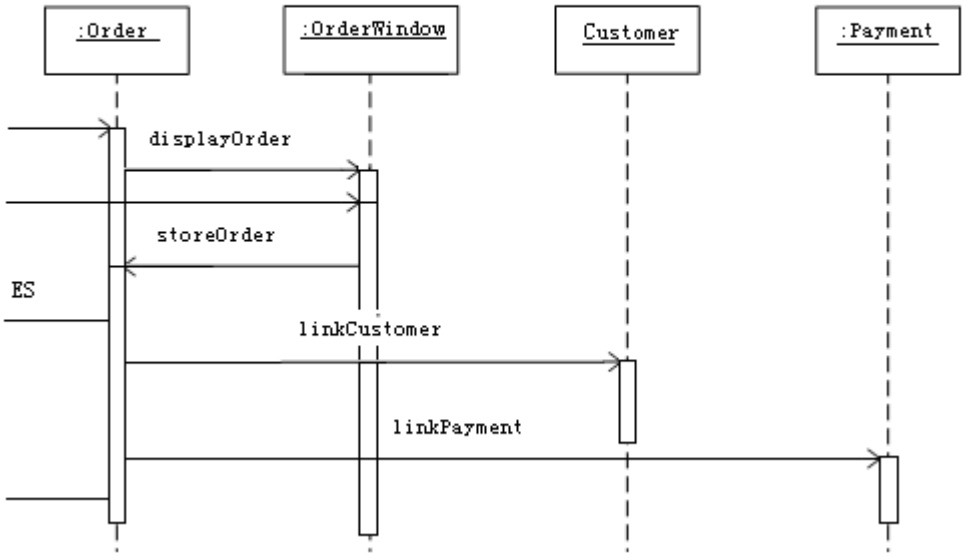
针对客户对预定细节的接受情况（submitOrder）,:OrderWindow 触发（storeOrede）一个永久的:Order 对象的创建，然后，这个:Order 对象把它自己链接到所预定的:ES 和相关的:Customer 和:Payment 对象上，一旦这个对象被永久地链接起来，这个:Order 对象就发送 emailOrder 消息给外部参与者:Customer。

注意，:Customer 机作为外部参与者对象又作为内部类对象的双重用法，这在建模中是经常出现的矛盾，客户对系统来说既是外部的又是内部的，作为外部的，它与系统交互，作为内部的，客户信息又必须保持在系统里面，以识别一个外部客户是否是系统已经知道的一个合法内部实体。

参照前面的活动图，可以画出序列图。这个序列图分成两部分，Order 和 OrderWindow 的生命线在这两部分中被重复使用。

这里只显示消息的激活，消息的返回是隐式的，同时也不需要指出参数。





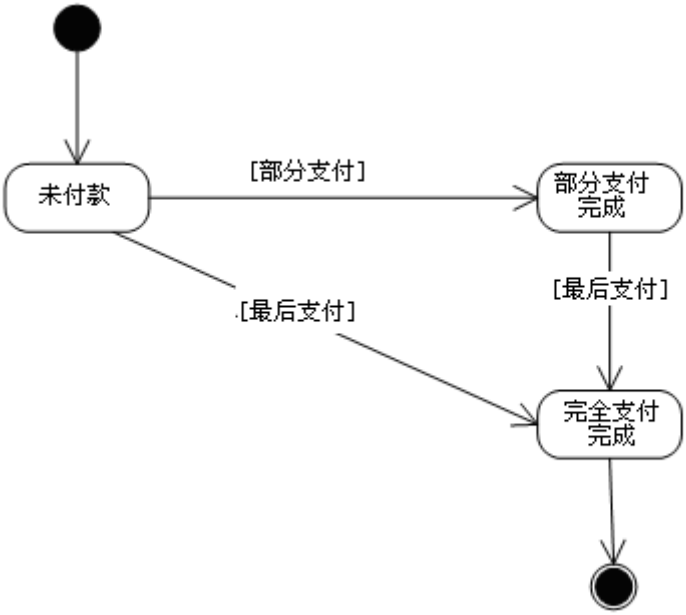
六、状态图及其讨论

状态图可以表达一个对象的状态变迁，事实上，在分析的时候，常常并不总是需要这种状态变迁的讨论，但是，有时候在最重要的类上进行这种讨论，也是有意义的，因为这可以为我们下一步系统设计打下了基础，我们来看下面的案例。

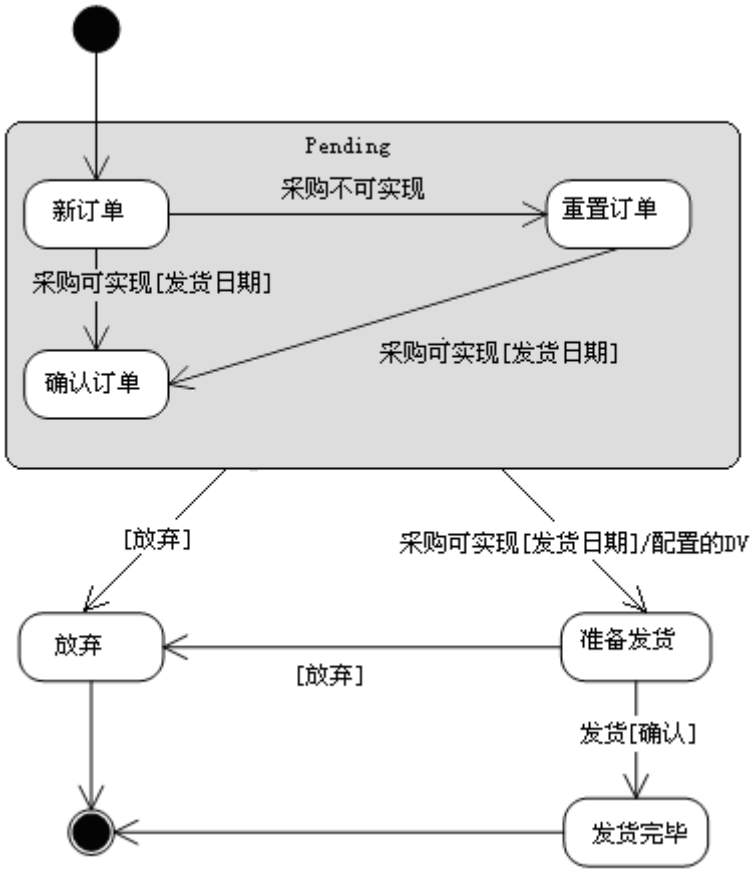
1，订单处理子系统

考虑 Invoice 对象的状态， 有两种支付方式，初始状态是未付款，形成发票有两种可能的状

态变迁，可以用下面的状态图表达。



另一方面，我们可以给 Order 对象画出状态图，一个新订单可能有两种状态变迁，可以看出来，这对细腻的描述订单的状态变迁是很有意义的。



2，网站设计运行时的系统模型

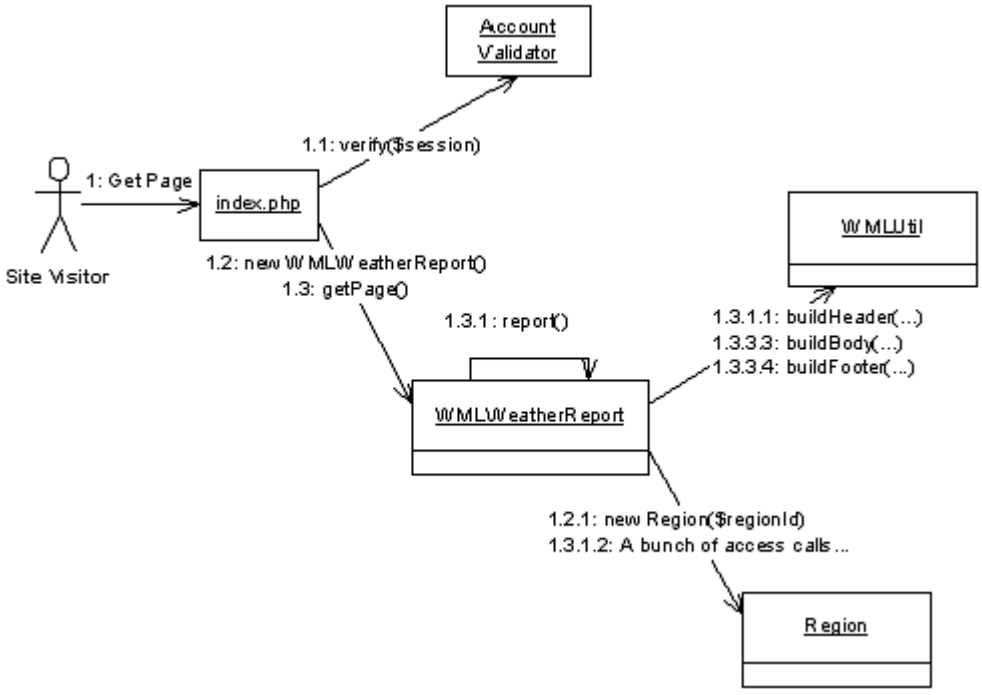
有些时候，我们需要显示出应用的各个部件如何在运行时协作完成任务。前面的类图显示了

类之间的关系，但它没有显示出调用出现的次序，也没有显示出来自一个函数的结果可能决定下一次调用的目标。为了在更动态的层面上描述系统，UML 提供了许多其他类型的图。

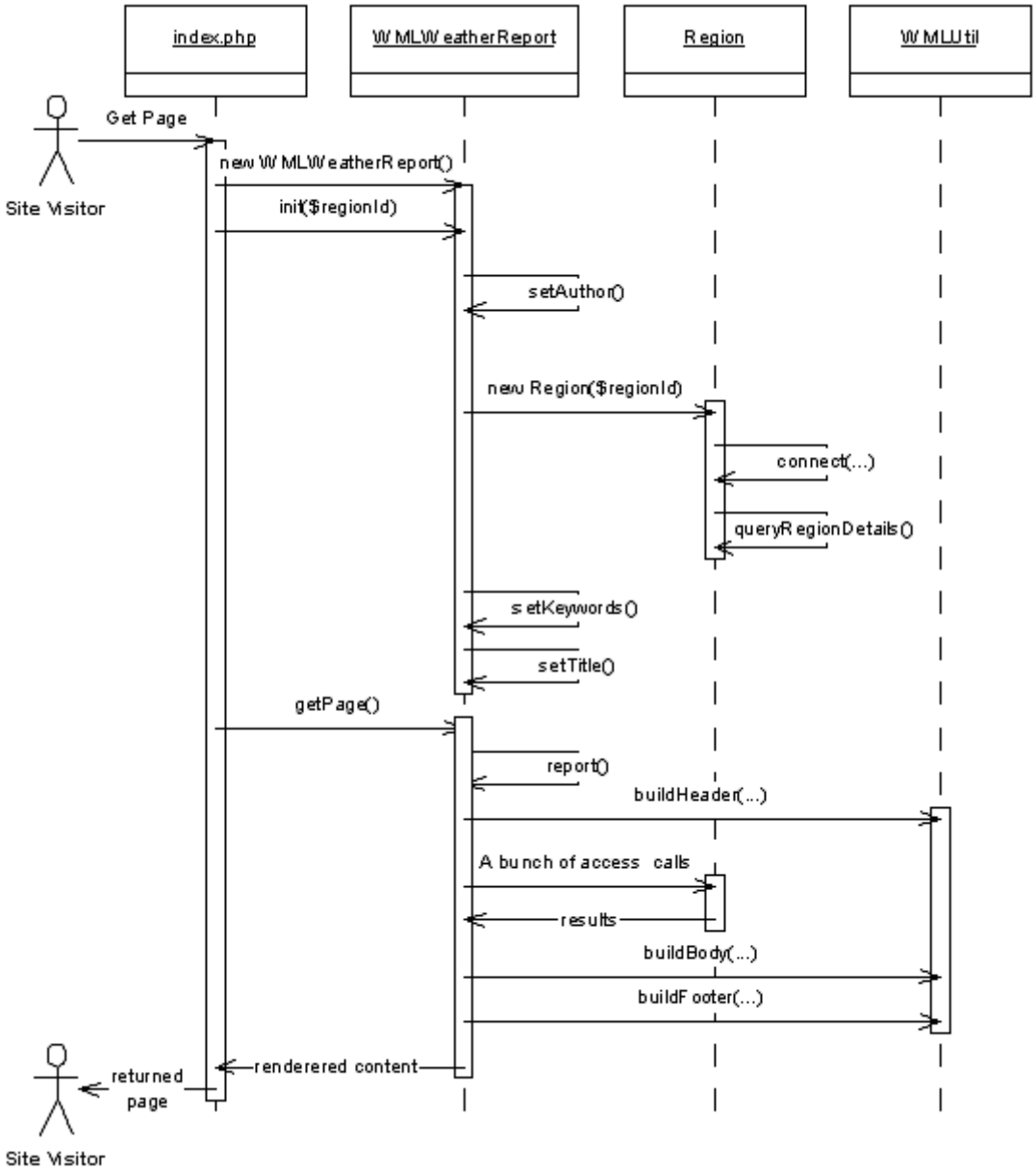
对于 Web 网站设计来说，情节图（Scenario Diagram）特别有用。

情节图分成两种：协作图（Collaboration Diagram），序列图（Sequence Diagram）。一般地，我们不会建立系统所有交互过程的模型，情节图只用来描述系统最复杂的部分，或用来概括出代码的一般调用模式。例如，我们可能要示范特定的页面如何与验证用户身份的代码协作，或者要显示页面如何调用公用代码（工具性的框架代码）以保持统一的外观和风格。

协作图和序列图分别举例如下。



上面的协作图显示了从 Web 网站获取天气报表的一般过程。注意该图忽略了一些不重要的方法，因为我们只对处理过程中的关键步骤感兴趣。你可以根据编号“1”到“1.3.3.4”找出各个函数的执行次序。一些人喜欢以“1，2，3，……”形式对执行步骤编号，但一般而言，用“1，1.1，1.2，2，2.1，……”的形式显示出调用栈的深度是一种更好的选择，这种编号方式能够更清楚地显示出程序的控制转换过程。例如，上图显示出 report()方法调用了 WMLUtil 以及 Region 对象中的许多方法：在通过一系列的查询和内容生成函数为指定地区生成报表之前，我们调用了 WMLUtil 中的 buildHeader(...)函数；最后我们调用的是 WMLUtil 模块的 buildFooter(...)，然后返回 report()方法，最后返回 getPage()。你可以为协作图加上更多的细节说明，比如返回值、约束、条件等。



就图形所传达的信息而言，次序图和协作图非常相似。事实上，许多 UML 建模工具能够从协作图生成次序图，或者相反。次序图与协作图的主要不同之处在于：在次序图上，事件的发生次序一目了然，非常直观。另外，次序图中还可以加入生存周期和时间方面的详细信息，比如延迟、线程并发、对象的构造和删除等。

在决定选用次序图还是协作图的时候，考虑以下几点有助于你作出最合适的选择：

如果要显示代码中与时间或线程密切相关的问题，选择次序图。

如果要显示对象之间的交互模式，选择协作图。

如果要显示几个或者大量对象之间的交互过程，选择次序图。

如果要显示少量对象之间的大量消息传递或交互过程，选择协作图。

5.8 系统优先级分析

一、为什么要设定需求的优先级

当客户的期望很高、开发时间比较短并且资源有限时，我们必须尽早确定所交付的产品应该具备的最重要的功能。合理建立每个功能的相对重要性，有助于你我们规划软件的开发重点，使我们可以以最少的费用提供产品的最大功能。如果我们选择的软件工程策略是迭代或者增量式开发，那么设定优先级就更加重要，因为在这些开发中，交付进度安排很紧迫并且不可改变日期，在每个迭代周期中，都需要考虑排除或推迟一些不重要的功能。

每个项目负责人都必须权衡合理的项目范围、进度、预算、人力资源以及质量目标的约束。比较常用的权衡的方法是：当接受一个新的高优先级的需求或者其它项目环境变化时，删除低优先级的需求，或者把它们推迟到下一版本中去实现。如果客户没有以重要性和紧迫性来区分它们的需求，那么项目负责人就必须自己作出这样的决策。但这样一来就会有风险，因为客户可能并不赞成项目经理所设定的优先级。

在需求评审过程中，由客户与开发部门负责人一起来确定哪些需求必须包括在首发版中，而哪些需求可以延期实现是一个很好的策略。当项目有很多选择可以完成一个成功的产品时，应该尽早在项目中设定其优先级。

但是实际情况并不这么简单，让每一个客户都来决定他们的需求中哪一些是最重要的，这实际上很难做到，要在众多具有不同期望的客户之间达成一致意见就更难了。因为人们心中都存在个人的利益，并且他们并不总能与其它群体的利益相妥协。这就需要一些方法对需求的优先级进行分析，这样更便于就优先级问题达成共识。

确定优先级是比较复杂的，因为它牵涉到不同的因素，而这些因素常常互相冲突。优先级问题上，还要考虑功能的关注点、重要性和影响性，综合起来考虑问题，软件需求优先级应该综合考虑其它各种因素在过程中动态调整。确定优先级牵涉到不同的因素，而这些因素常常互相冲突，风险承担者也可能会有不同的目的，达成一致往往比较困难。因此，我们可以通过一些工程方法来综合考虑各种因素，通过这些方法可以消除一些情感、政策以及处理过程中的推测，使考虑问题比较全面。

二、不同角色的人处理优先级

优先级设定的困难来自于两个方面：

- **客户方面**：客户的心理往往是：“我需要所有的特性，只要以某种方式使它发生即可。”如果用户知道低优先级需求可能不会实现，那么就很难说服用户设定需求的优先级。
- **开发者方面**：开发者的概念是，既然需求被写入软件需求规格说明，那么就应该不遗余力地去实现这些需求。开发者更喜欢避开优先级设定的原因，在于他们觉得建立优先级与他们要向经理表示的“我们可以全部完成产品”的态度相冲突。

但是在现实中，一些特性确实比其它特性更重要，把握需求的重点对项目的成功极其有意义。例如：在项目接近尾声时，开发者可能会需要抛弃掉一些不必要的功能以保证按时完工，在最后阶段匆忙决定可能会使项目陷入灾难，而在项目的早期阶段设定优先级将有助于我们在项目过程中逐步作出相互协调的决策。如果在项目的后半期我们感觉需要考虑哪些是低优先级需求，但这时我们已经实现了将近一半的特性，那这将是一种浪费。

我们也不能把优先级设定问题完全推给客户，因为客户自己设定优先级，大部分情况下他们会把 85 %的需求设定为高优先级，10 %的需求设定为中等优先级，5 %的需求设定为低优先级。这种设定方式和没有设定优先级并没有太大的区别。正是这样一个背景，我们需要以一种更专业的眼光来审视优先级设定的方方面面。

第六章 数据库结构设计

详细设计阶段包括：数据库设计、模块设计和界面设计。下面我们探讨一下与数据库设计的有关问题。

6.1 关系型数据库的结构设计

一、面向过程的设计与实体关系图

业务领域中的事物分析是系统数据分析与设计的基础，也就是说事物分析所获得的概念类图是实体-关系（ER）模型建立的基础。一个关系模式是对某类实体记录集框架结构的形式化描述。在一项业务中，最稳定而不易变化的业务成分是实体间相互的关联关系，正因为如此，在初期设计的时候，我们需要花费更多的精力来把数据的关联结构设计完善。

数据建模中，关系约束有两种，它们是基数与强制性限定。关系数据库的单元是表，在面向过程的设计中，建立一个关系数据库的第一步，需要仔细考虑实体-关系图（ERD），给每个实体建立一张表，每张表的数据域要与已经定义的实体相一致。然后，可以为每个表建立一个主键，如果没有合适的字段作为主键，可以自己创造一个，主键的数据必须是唯一的。

1）实体

实体指的是某些事物，企业需要存储有关这些事物的数据。实体实例表达的是实体的具体值。

2）属性

实体的描述特形称为**属性**，某些属性可以逻辑上被组合，称为**组合属性**，它在不同的数据建模语言中也被称作串联属性、合成属性或者数据结构。

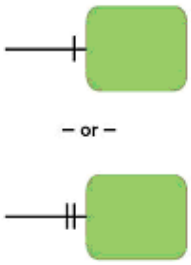
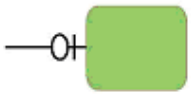
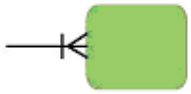
3）域

域是属性的一个参数，定义了这个属性所能定义的合法值。事实上这个值和使用的数据库特点有关。

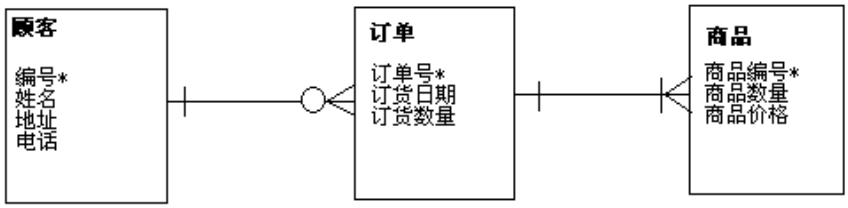
4）标识符

每一个实体和属性必须有一个唯一的名字。

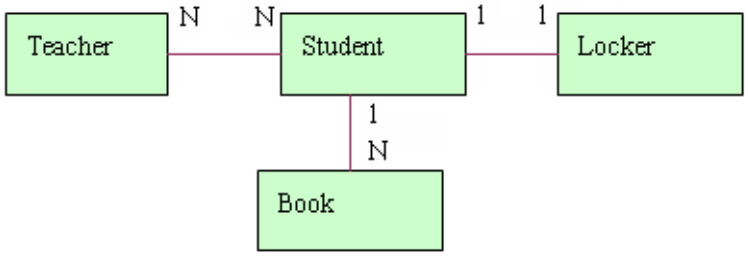
然后考虑实体之间的关系，关系基数符号如下：

基数含义	最小实例数	最大实例数	图形化符号
正好一个（一个）且只有一个	1	1	
零个或一个	0	1	
一个或多个	1	多个（>1）	
零个、一个或多个	0	多个（>1）	
大于一个	>1	>1	

以此可以建立表与表之间的关系：



实体之间的关系有三种，最简单的而且最有代表性的例子也就是学生、老师、锁柜和书的关系。



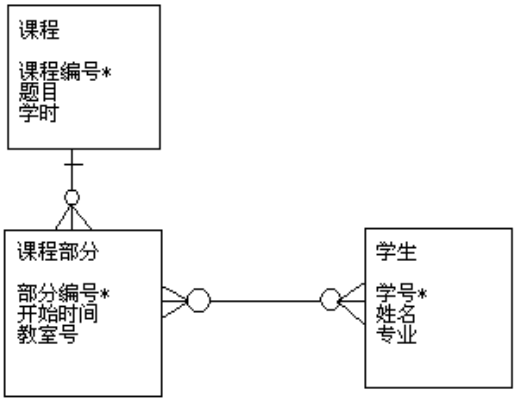
也就是一对一（1：1）或一对多（1：N）或多对多（N：N）关系。

比如：

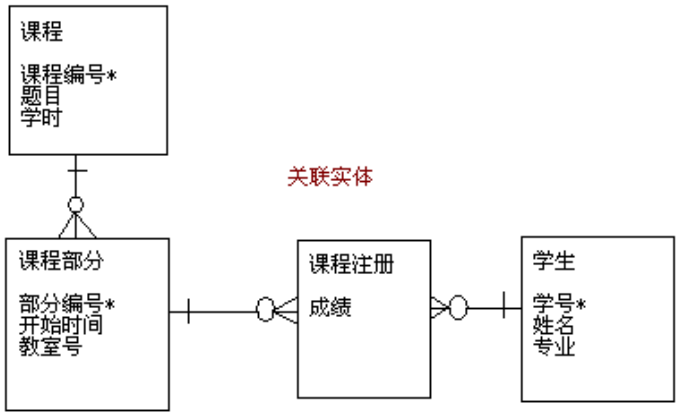
Student 和 Locker 之间可以是一个一对一关系，因为每个学生都有一个相应的锁柜，而每个锁柜只给一个学生使用。Student 和 Book 之间可以有一个一对多关系，因为每个学生可以有多本书，但每本书只归一个学生所有。Student 和 Teacher 之间有一个多对多关系，每个学生可以有多个教师授课，而每个教师又可以为多个学生讲课。

注意，一个实体可以参与多个关系，而每个关系可以有不同的对应关系。

这里还有一个问题，就是多对多关系需要增加一个关联实体，比如如下的多对多关系：



我们会发现，学生某门课的成绩应该放在什么地方呢？尽管模型中表达了学生选修了某门课程，但没有放置这门课程成绩的地方，所以需要增加一个关联实体。



这样，我们就可以得到设计关系数据库的一般步骤：

1，为每个实体类型建立一张表。

2，为每张表选择一个主键（如果需要，可以定义一个）。

3，增加外部码以建立一对多关系。

4，建立几个新表来表示多对多关系。

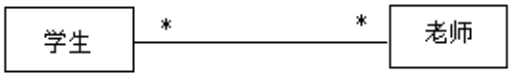
5，定义参照完整性约束。

6，评价模式质量，并进行必要的改进。

7，为每个字段选择适当的数据类型和取值范围。

二、关于多对多关系的讨论

多对多关系是一种比较难以处理的关系，典型的场景是：一个学生（Student)有多个老师（Teacher)，一个老师有多个学生。



象学生选课，并且记录成绩，就是这种场景。其实学生、老师、锁柜、书这个关系，使数据库问题的根本关系，弄清楚这个问题，其他任何问题都可以迎刃而解了。

我们可以先把两个表建立起来，如下表所示，其中 A 和 B 都是具体的值。

学生表	
StudentID[PK]	**其他**
A1	
A2	
A3	
A4	
A5	
A6	

老师表	
TeacherID[PK]	**其他**
B1	
B2	
B3	
B4	
B5	

可以有两个方案解决这类问题.

第一方案：

增加一个关联表，每边都增加一个外键，而关联表是主键，用关联表的主键建立这种联系（这是我原来在论文数据结构中建立的关系）。

学生表	
StudentID[PK]	**Task[FK]**
A1	M
A2	N
A3	N
A4	M
A5	K
A6	M

关联表
Task[PK]
M
N
K

老师表	
TeacherID[PK]	**Task[FK]**
B1	M
B2	K
B3	M
B4	N
B5	K

这种表结构的多重性关系如下。

这种关联的建立方法如下：

首先在关联表上建立几种模式（比如何种销售策略），然后在两边的表中选择自己是属于何种模式的，在外键上填上相应的符号。

它的工作原理我们可以用表中的实际数据理解：

从学生看：

A1：B1，B3

A2：B4

A3：B4

A4：B1，B3

A5：B2，B5

A6：B2，B5

从老师看

B1：A1，A4

B2：A5

B3：A1，A4，A6

B4：A2，A3

B5：A5

可见，多对多关系确实建立起来了。

但是这种方案有个很大的缺点，那就是它只适合于几个比较固定的模态，而且两个表之间的对应关系不能重复，不灵活，对于像销售策略和商品这类关系，这种方法也可能是合适的。但如果需要一种复杂多变的场合，某个人可能需要多种选择，比如学生自由选课，或者任务分配，一个任务多个人，以个人承担多种任务，这种方法就不合适了。

所以这种方案只适合某些极其特殊的场合。

第二方案

增加一个关联表，用两边的主键建立**联合主键**，在联合主键中，每个主键各自是可以重复的，但是两者的联合是不可重复的。

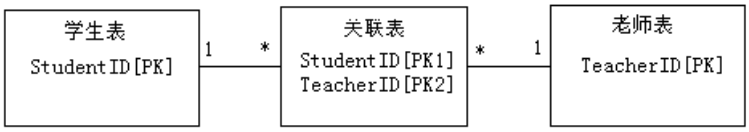
这种模态是很自由的，可以形成各种复杂的组合，而且两张主表中不需要添加外键字段。

学生表	
StudentID[PK]	
A1	
A2	
A3	
A4	
A5	
A6	

关联表	
StudentID[PK]	**TeacherID[PK]**
A1	B1
A4	B2
A2	B1
A1	B4
A5	B1
A3	B1
A2	B2
A1	B3
A3	B4
A4	B4
A5	B4
A4	B3
A6	B2
A3	B5
A6	B4
A5	B5

老师表	
TeacherID[PK]	
B1	
B2	
B3	
B4	
B5	

从这个表上看，关联表的长度一般要大于两边各自的表，所以多重性关系如下。



注意和第一方案相反，多的关系是指向关联表的。

关联表的建立方法是：

首先确定老师列表，由学生决定选择哪些老师（自由选课），每做一个决定，关联表就增加一行，当然也可以把选中的老师去掉，表现为删除关联表中的相应行，

这种表的查询方法是，先从与自己主键匹配的关联表主键列找到所有属于自己的行，然后看

对应的另一方。因此，我们可以写出。

从学生看：

A1：B1，B3，B4

A2：B1，B2

A3：B1，B4，B5

A4：B2，B3，B4

A5：B1，B4，B5

A6：B2，B4

从老师看

B1：A1，A2，A3，A5

B2：A2，A4，A6

B3：A1，A4

B4：A1，A3，A4，A5，A6

B5：A3，A5

显然具有相当灵活的关系。

这种方法的优点是极其灵活，可以处理各种情况，缺点是关联表的联合主键处理上要麻烦一些，必须确保两者合并后不能重复，但是，只要实行了正确的设置，目前数据库管理系统都有自动处理这类问题能力，所以一般问题不大。

关联表还可以有更多的字段，比如学生成绩，就可以存放在关联表中，作为一个专门的字段存在。所有这些查询方法都可以由程序来实现，最后展示给用户的，实际上是经过组合的，他所需要的表格样式，从本质上看，电脑就是在复现我们人自己的思考问题方式。

三、执行参照完整性

建立关系主要需要建立主键和外键的关系，执行参照完整性表达了外键和主键间一致的状态。执行参照完整性表达的是：一个一致的关系数据库状态，每个外键的值必须有一个主键值与之对应。

规则：

1，当建立一个包含外键的记录的时候，应确保主表中相应的主键值要存在。

2，当删除一条记录的时候，要确保所有相应外键的记录也被删除。

3，当更改一个主键值的时候，要确保所有相应表外键值也跟随改变。

三、评价模式质量

一个高质量的数据模型应该具备以下特点，

1，表中每行数据及主键是唯一的。

2，冗余数据较少。

3，容易实现将来数据模型的改变

不过，提高数据库设计质量的方法有很多，但量化方法又很少，很大程度上依赖于设计师的经验和判断，下面提供几个注意点。

1，行和关键字的唯一性

主键是能够唯一定义一行数据的一列或者多列，主键中的列值不能为 null，主键为数据库引擎提供了获取使据库表中某个特定行的方法，主键还用于保证引用的完整性。如果多个用户同时插入数据，则必须保证不会出现重复的主键。

由于主键是必须存在的，而主键是不可重复的，显然表中的每一行也都是唯一的，这就是所有关系数据库模型都有一个基本要求，那就是主键和表中的行是唯一的。

但我们应该如何来选择主键呢？

智能键、常规建和代理键

智能键是一种基于商业数据表示的键，例如 SKU（Stock Keeping Unit 常用保存单元）就是智能键的例子。它定义一个 10 个字符的字段（Char(10)），它的可能分配如下，前 4 个字符为供应商代号，随后 3 个字符保存产品类型代号，最后 3 个字符保存一个序列号。

常规键由现有商业数据中一个或多个列组成，比如社会保险号。

尽管智能键和常规键不尽相同，但他们都是用商业相关数据组成，下面统一成为智能键。

代理键是由系统生成的，与商业数据无关，比如自动增值列（Identity），GUID（globally unique identifier 全局唯一代码，16 字符），这是通过取值算法得到的键值，后面将称之为 GUID 键。

下面的例子包含三张表（作者 Author，书籍 Book，而 AuthorBook 是一张多对多的连接表，因为一个书籍可能由多个作者完成）。

注意，在代理键完成的时候，需要多增加一个列值，这是因为代理键是系统自动生成，用户不可见的。

数据大小

使用智能键或代理键数据的大小是不一样的，因此一定要计算键的使用引起数据量的变化，显然，基于 int 的自动增加列数据量最小，但也要注意，int 的最大值是有限的，而且不便于移动数据，这些都是考虑的因素。

键的可见性

智能键是可见的而且是有意义的，而代理键一般不对用户开放而且是无意义的，从维护的角度来说，似乎智能键更优，因此，如果智能键的数据确实存在，可以考虑智能键。

在非连接对象确保唯一性

在非连接对象中，两个人同时加入行智能键重复的几率是存在的，但代理键不可能重复。而自动增加列值存在着诸多限制（移动性，最大值），所以 GUID 键是最合适的。

在数据库中移动数据

自动增加列值事实上无法在数据库中移动数据，智能键除非仔细设计，主键重复也不是没可能，比较好的是 GUID。

使用的方便性

自动增加键是最方便的，但由于上面种种讨论，并不推荐使用，智能键需要多列组成，但方便性可以接受，GUID 的使用往往叫人不太习惯，但合理的设计以后，这并不是问题，所以 GUID 主键还是最常用的。

2，数据模型的灵活性

在关系型数据库最早的规范当中，数据库的灵活性和可维护性是最主要的目标。如果对数据库模式进行更改，对已经存在的数据内容和结构造成的影响最小，那么就可以认为这个关系数据库模型是灵活的而且是可维护的。

比如，增加一个新的实体，不需要对原有的表进行重新定义。增加一个新的一对多关系，只要求给已存在的表添加一个外部码。增加一个新的多对多关系，只需要在模式中添加一个单独的

新表。在判断数据模型的灵活性的时候，要特别注意属于冗余的影响，一般来说，数据存在多个地方，那么在进行增、删、改、查操作的时候会增加额外的操作，而且维护上也更复杂和低效，在操作失败的时候，数据不一致的危险性也会更大。

多表关联的时候，外键是必须的，但这样也会造成关键字段操作的复杂性。

关系型数据库管理系统通过参照完整性的约束来保证主键和外键一致，但并没有自动化的方法来保证冗余数据项的一致，为了避免关键字段的数据冗余，可以采用数据库的规范化。

数据库规范化是用来评价关系数据库模式质量的有效技术，它可以确定一个数据库模式是不是包含了任何错误冗余，它基本的表述如下：

第 1 范式（1NF）：没有重复字段和字段组的数据库表结构。

函数相关：两个字段**值**之间一一对应。如果对于任意字段 B 的值有而且只有一个 A 的值与之对应，则称 A 函数相关 B。

这里主要研究的是字段内容，保证表中数据没有冗余。

第 2 范式（2NF）：每个非关键字段对于主键或者主键组函数相关。

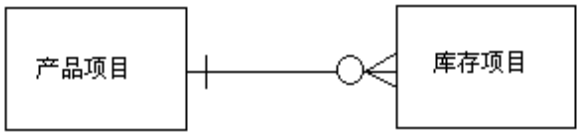
这里主要研究其它字段与关键字段的关系

第 3 范式（3NF）：各个非关键字段之间不能函数相关。

这个范式保证了数据没有冗余。

下面对这些概念作一些解释。

设想有这样的实体，并以此构造了两张表。



表“产品项目”中的数据，位置是“demo”中

编号	产品名称	标准价格
1244	宝莱	170000
1245	捷达	90000

表“库存项目”中的数据，位置是“demo”中、“(local)”上

编号	产品编号	颜色	配置	附加价格
8211	1244	红	标准手动	0
8212	1244	兰	标准自动	20000
8213	1244	白	豪华自动	50000
8214	1245	黑	标准自动	20000

第一范式：

这是对表格行定义的一个结构限制，事实上关系数据库管理系统本身就是在一张表中拒绝由两个相同的字段的，所以要实现第一范式并不困难。

函数相关：

这是一个比较难以描述以及应用的概念。

比如考察“产品项目”表中“编号”和“标准价格”两个字段，现在已知“编号”是一个内部主键，在表中一定是唯一的，为了判断“标准价格”是不是函数相关“编号”，只需要描述：对于字段“编号”的值有而且只有一个“标准价格”的值与之对应，则“标准价格”函数相关“编号”。

我们来考察一下对于“产品项目”表这个表述对不对？事实上只要字段“编号”数据是唯一的，这个表述就是正确的，也就是“标准价格”函数相关“编号”。

一个不太确切但很简单的方式如下，“产品项目”表中对于每个产品产品价格应该是唯一的，这就是函数相关的，如果每个产品有多种价格，这就不是函数相关的。

第二范式：

为了判断“产品项目”表是不是属于第二范式，我们必须首先判断它是不是第一范式，因为它不包括重复的字段，所以它是属于第一范式，然后我们需要判断每个非关键字段都函数相关于“标准价格”（也就是每个字段都来替换函数相关定义中的 A），如果每个非关键字段都函数相关于“标准价格”，那“产品项目”表就是属于第二范式。

当主键是由两个或者多个字段组成的时候，判断表是不是第二范式就比较复杂，例如考虑如下的“目录产品”表，这个表示为了表示“销售目录”表和“产品项目”表之间的多对多关系。所以，表达这个关系的表的主键由“销售目录”的主键（“目录号”）和“产品项目”的主键（“产品号”）组成，这个表还包含了一个非关键字段公布价格。受市场影响，公布的价格往往是浮动的。

表“目录产品”中的数据，位置是“demo”中

目录号	产品号	公布价格
22	1244	150000
23	1244	140000
23	1245	80000
24	1245	70000

如果这张表属于第二范式，那么非主关键字段“公布价格”必定函数相关于“目录号”与“产品号”组合。我们可以通过替换函数相关定义中的词语来验证函数相关：

对于“目录号”与“产品号”组合的值有而且只有一个“公布价格”的值与之对应，则“公布价格”函数相关“目录号”与“产品号”组合。

分析这样的语句正确性还是需要技巧的，因为你必须考虑“产品目录”表中所有所有可能出现的关键字值得组合。比较简单的分析的方法不是机械的对照，而是考虑这个实体本身的问题。

一个产品可能在多个不同的销售目录中出现，如果在不同的目录中它的价格不同，那么上述的陈述就是正确的。如果产品不论在哪个目录中，它的价格是相同的（或者说一个价格数据对应于多个目录），那上述的陈述就是错误的，而且这个表不是第二范式。所以正确的判断不是依赖于陈述，而往往是依赖于对问题本身的理解。

如果非关键字段只是函数相关于主键组的一部分，那么这个非关键字段必须从当前表中移出去，并且放在另一个表中。

例如如下的“目录产品”表，增加了一个目录的“发布日期”字段，显然，这个字段函数相关于“目录号”，但不函数相关于“产品号”，也就是同一个“产品号”可能在多个“发布日期”中使用，这就会造成冗余，这个表不属于第二范式。

表“目录产品”中的数据，位置是“demo”中

目录号	产品号	价格	发布日期
22	1244	150000	2006-1-1
23	1244	140000	2006-6-1
23	1245	80000	2006-6-1
24	1245	70000	2006-10-1

正确的做法是把“发布日期”移出来，放在“销售目录”这张表中，这时候就正确了。

表“目录产品”中的数据，位置是“demo”中

	目录号	产品号	价格
	22	1244	150000
	23	1244	140000
	23	1245	80000
	24	1245	70000

表“销售目录”中的数据，位置是“demo”中

	目录号	发布日期
	22	2006-1-1
	23	2006-6-1
	24	2006-10-1
	25	2006-11-1
▶	26	2006-12-1

要判断一张表是不是第三范式，则必须考虑每一个非关键字段是不是函数相关于其它的非关键字段，如果是，就要考虑结构上的修正。

对于一个大型表来说，这实际上是一个非常复杂的工作，而且工作量随着字段数的增加而快速增。当非关键字段数是 N 时，要考虑的函数相关数目为 N*（N-1）。而且要注意函数相关要两方面考虑（即 A 相关 B，B 相关 A）。

我们来考虑下面一个简单的“职工状况”表。

表“职工状况”中的数据，位置是“demo”中、“(local)”上

	编号	住址	部门	部门编号
	8266	北京市科学院南路888号	企管部	1010
	8267	北京市海淀区234号	电源部	1011
	8268	北京市朝阳区458号	培训部	1012
	8269	北京市宣武区2112号	培训部	1012

这张表有三个非关键字段，所以需要考虑六个函数相关：

“部门”与“住址”？一个住址可能有多个部门的人，不是。

“住址”与“部门”？一个部门相同住址的可能不止一个，不是。

“部门编号”与“住址”？一个住址可能牵涉到的部门编号是多个，不是。

“住址”与“部门编号”？一个部门编号有相同住址的可能不止一个，不是。

“部门编号”与“部门”？一个部门只有一个部门编号，是。

“部门”与“部门编号”？一个部门编号只对应一个部门，是。

注意，有时候两方面考虑只有一个是，比如“省份”和“邮政编码”，一个省份有多个邮政编码，而一个邮政编码只能对应一个省份。

这样一来，当部门有多个人的时候就会出现大量的冗余，解决的办法是再增加一张表，表达“部门”和“部门编号”的对应关系。

表“职工状况”中的数据，位置是“demo”中、“(local)”上

	编号	住址	部门编号
	8266	北京市科学院南路888号	1010
	8267	北京市海淀区234号	1011
	8268	北京市朝阳区458号	1012
	8269	北京市宣武区2112号	1012

表“部门编码”中的数据，位置是“demo”中

	部门编号	部门
	1010	企管部
	1011	电源部
	1012	培训部
	1012	培训部

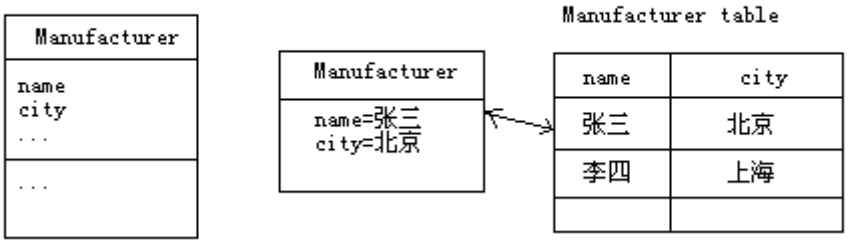
如果出现了需要几个字段数据计算出结果的字段，也不属于第三范式，可以把这个字段取消，由调用方通过程序来解决。

6.2 面向对象数据库设计

一、模式：把数据对象表表示成类

把对象表示成表（Representing Objects as Tables）的模式建议，在 RDB 中对每一个持久化对象类定义一个表，对象的原始数据类型（数字、字符串、布尔值）的属性映射为列。如果对象只有原始数据类型的属性，就可以直接映射。但对象如果还包含了其它复杂对象的引用属性，事情就比较复杂，因为关系型模型需要的值是原子性的（第一范式），所以除非有充分的例有，尽可能不要这样来做。

关于厂商表的映射关系

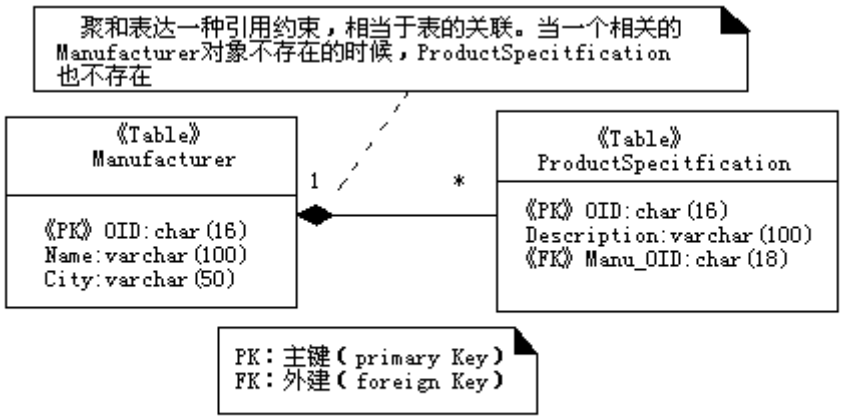


在 UML 中，虽然可以很好的表达类，但是，为了确切的表达数据，还需要有一些扩展，这个关于数据建模的扩展标准，已经提交给了 OMG 组织。

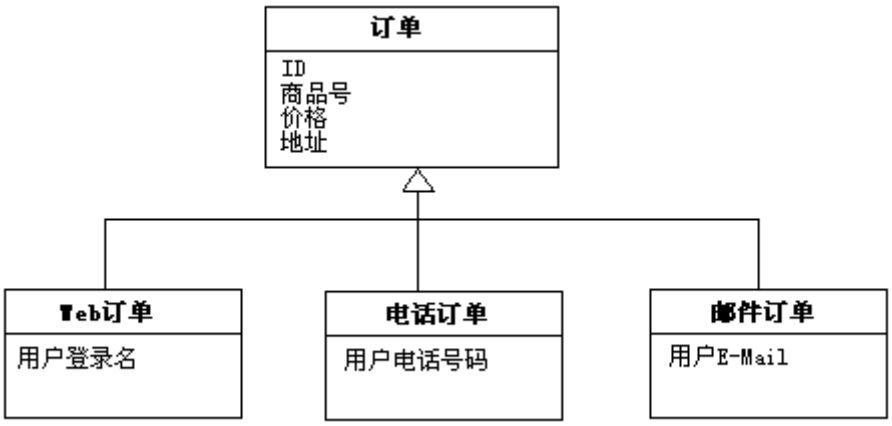
比如：PK：主键（primary Key）；

FK：外建（foreign Key）。

等。



注意，如果仅仅把对象表达成类，和基于结构的思维实际上一样的，面向对象的数据表达最大的特点是数据具有继承性，这是和面向过程的设计完全不同的地方。也就是说，对象数据库设计的时候，和关系数据库很大的区别，在于类可以实现继承，比如如下的例子。



这为我们优化系统提供了更大的思维空间。

6.3 处理事务

一、为什么要关注事务处理

执行事务事务是一组组合成逻辑工作单元的操作，虽然系统中可能会出错，但事务将控制和维护事务中每个操作的一致性和完整性。

例如，在将资金从一个帐户转移到另一个帐户的银行应用中，一个帐户将一定的金额贷记到一个数据库表中，同时另一个帐户将相同的金额借记到另一个数据库表中。由于计算机可能会因停电、网络中断等而出现故障，因此有可能更新了一个表中的行，但没有更新另一个表中的行。如果数据库支持事务，则可以将数据库操作组成一个事务，以防止因这些事件而使数据库出现不一致。如果事务中的某个点发生故障，则所有更新都可以回滚到事务开始之前的状态。如果没有发生故障，则通过以完成状态提交事务来完成更新。

在一个操作中，如果牵涉到多个永久存储，而且是多步完成，并且需要修改数据的时候，就一定要考虑加上事务处理。

二、事务处理的基本概念

事务是一个原子工作单位，必须完整地完成其中的所有工作，如果提交事务，则事务执行成功，如果终止事务，则事务执行失败。事务具备以下 4 个关键属性：原子性、一致性、孤立性和持久性，这被称之为 ACID 属性。

- **原子性**：事务的工作不能划分为更小的部分，尽管其中包括了多条 Sql 语句，但它们要么全部执行，要么都不执行。这意味着出现一个错误的时候，将全体恢复到启动事务前的状态。
- **一致性**：事务必须操作一致性的视图，并且必须使数据处于一致的状态，事务再提交之前，它的工作绝不会影响到其他的事务。
- **孤立性**：事务必须是独立运行的实体，一个事务不会影响到其它正在执行的事务。
- **持久性**：在提交一个事务的时候，必须永久性的存储它，以免发生停电或系统失败丢失事务。在重新供电或者恢复系统以后，将只会恢复已经提交的事务，而退回没有提交的事务。

1）并发模型和数据库锁定

数据库使用数据库锁定机制来防止事务互相影响，以实现事务的一致性和孤立性，事物在访问数据的时候，将强制锁定数据，而其它要访问的事物将强制处于等待状态，这说明长时间的运行事务是不可取的，这会严重影响系统性能和可测量性。这种用锁定来阻止访问的做法，称为“悲观的”并发模型。

在“乐观的”并发模型中，将不使用锁，而是检查数据在读取之后是不是发生了变化，如果发生了变化，则抛出一个异常，由商业逻辑进行恢复，前面 DataAdapter 的 UpDate 方法就是使用了这种模型，但它无法解决我们在前面应行的例子中出现的问题。

2）事务的孤立级别

实现完全孤立的事务当然好，但代价太高，完全孤立性要求只有在锁定事务的情况下，才能读写任何数据，甚至锁定将要读取的数据。

根据应用程序的目的，可能并不需要实现完全的孤立性，通过调整事务的孤立级别，就可以减少使用锁定的次数，并提高可测量性和性能，事物孤立性将影响下列操作：

- **Dirty 读取操作**：该操作能够读取还没有提交的数据，在退回一个添加数据的事务的时候，该操作会造成大问题。
- **Nonrepeatable 读取操作**：该操作指一个事务多次读取同一行数据的时候，另一个事务可以在第一个事务读取数据期间，修改这行数据。
- **Phantom 读取操作**：该操作指一个事务多次读取同一个行集的时候，另一个事务可以在第一个事务读取数据期间，插入或删除这个行集中的行。

下表列出了典型数据库中的孤立级别。

级别	Dirty 读取操作	Nonrepeatable 读取操作	Phantom 读取操作	并发模型
Read Uncommitted	是	是	是	无
Read committed With Locks	否	是	是	“悲观的”并发模型
Read committed With Snapshots	否	是	是	“乐观的”并发模型
Repeatable Read	否	否	是	“悲观的”

				并发模型
Snapshot	否	否	否	“乐观的” 并发模型
Serializable	否	否	否	“悲观的” 并发模型

下面对各个级别进行讨论：

- **Read Uncommitted 级别**：其它事务的修改情况将对某个事务的查询造成影响，如果设置为该级别，则在读取该数据的时候，即不会获取锁，也不愿意使用锁。
- **Read committed With Locks 级别**：这是 Sql Server 的默认设置，已提交的更新在事务间是可见的，长时间运行的查询，不需要实时保持一致。
- **Read committed With Snapshots 级别**：已提交的更新在事务间是可见的，如果设置为该级别，则不会获取锁。但行数据的版本信息将用于跟踪行数据的修改情况，长时间运行的查询需要实时保持一致，该级别将带来使用版本存储区的开销，版本存储区可以提高吞吐量，并且减少对锁的依赖。
- **Repeatable Read 级别**：在一个事务中，所有的读取操作都是一致的，其它事务不能影响该事务的查询结果，因为在完成该事务并取消锁定之前，其它事务一直处于等待状态。该级别主要用在读取数据后希望同一个事物中修改数据的情况。
- **Snapshot 级别**：在需要精确的执行长时间运行的查询和多语句事务的时候，如果不准备更新事务，则使用该事务。使用这个级别的时候，不会获取读取操作的锁，以防止其它事务修改数据，因为在读取快照（Snapshot）并提交修改数据的事务之前，其它事务看不到修改情况，数据可以在该级别事务中进行修改，但是在快照事务（Snapshot transaction）启动后，可能和更新相同数据的事务发生冲突。
- **Serializable 级别**：在所访问的行集上放置一个范围锁，这是一个多行锁，在完成事务之前，防止其它用户更新数据集或者在数据集中插入行。在事务的生命周期中，保持数据的一致性和正确性，这是孤立级别中的最高级别，因为这个级别要使用大量的锁，所以只是在必要的时候才考虑使用这个级别。

在提交 Update 或者 Delete 语句后，版本存储区将保存行版本记录，直到提交所有的活动记录位置，事实上，在提交或者结束下列事务类型之前，版本存储区将一直保存行版本记录。

- **在 Snapshot 孤立级别下运行的事务。**
- **在 Read committed With Snapshots 孤立级别下运行的事务。**
- **在提交事务之前启动的所有其它事务。**

三、使用容错恢复技术

在准备产品化应用程序的时候，如何知道数据库是不是能够经受众多用户对应用程序反复的使用呢？如果数据库服务器关机，会出现什么情况呢？如果数据库服务器需要快速重启，会出现什么情况呢？

首先，在停止和重启服务器的情况下，我们可以清除连接池并重新建立它。

但如何才能保证结果和服务器关闭之前完全相同呢？

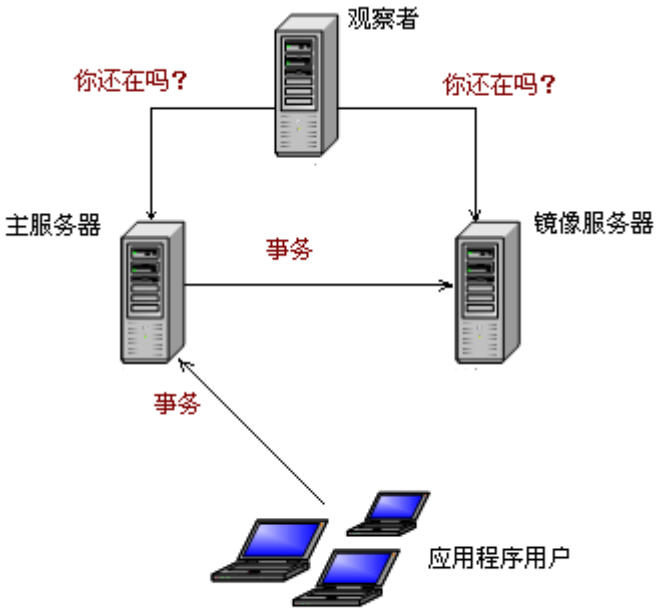
这就要用到容错恢复技术了。

请看下面的场景：

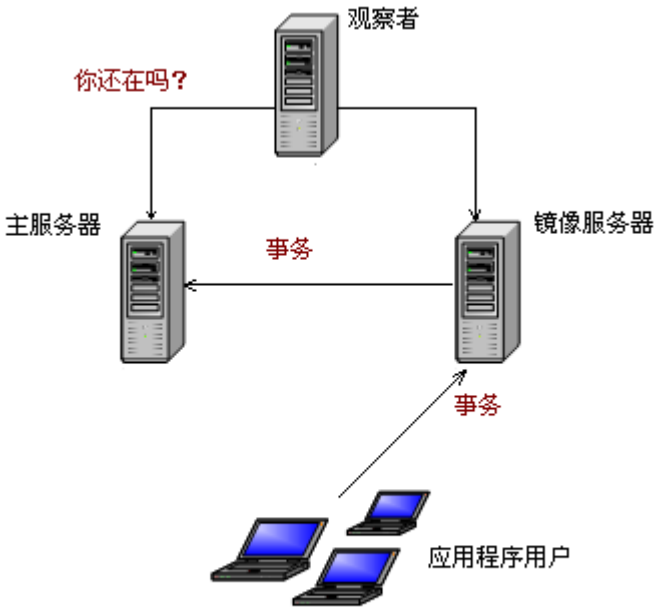
使用三台数据库服务器，“主服务器”、“镜像服务器”、“观察者服务器”，使用数据库镜像的时候，客户只和主服务器联系，镜像服务器处于数据恢复状态（不能进行任何访问），当向主服务器提交一个事务的时候，也会把这个事物发给镜像服务器。

观察者服务器只是观看主服务器和镜像服务器是不是正在正常工作。

在主服务器关机的时候，观察者自动把镜像服务器切换为主服务器，见下面两张图。



反映初始角色配置的数据库连接



反映“观察者”服务器转换为另两个服务器角色后的数据库镜像

注意，当发现主服务器有问题的时候，则自动清除连接池，并转而使用备用服务器。

6.4 数据库结构设计案例

一、PDM 文件管理系统的基本要求

产品数据管理（Product Data Management ， PDM）是机械设计行业常用的管理软件，文挡管理是 PDM 的核心功能，它由数据（元数据）以及指向描述产品不同方面的物埋数据和文件的指针组成，它为 PDM 控制环境和外部（用户和应用系统）之间的传递数据提供一种安全的手段。

为了建立正确的数据模型，我们首先需要了解 PDM 的系统在企业实施中的主要要求。如果对这种要求不了解，是无法设计出正确的数据结构的。因此，在进行设计以前，仔细研究产品的需求极其重要。我们构建的这个分布式数据和文裆管理功能，应该允许用户迅速无缝地访问企业的产品信息，而不用考虑用户和过据的物理位置。这些功能包括：

1）文件的检入（Check-in）和检出（Check-out）；

2）按属性搜索机制；

3）动态浏览 / 导航能力；

4）分布式文件管理；

5）安全机制（记录锁定，域锁定）。

这些要求与目前流行的各种 PDM 文件系统是一致的，为了更好的发挥 PDM 管理功能，使项目能以有序受控的方式进行开发，系统还需要具备项目管理能力，下面我们主要介绍 PDM 项目管理部分数据结构的设计过程。

二、领域模型的初步建立

很多建立数据结构的过程是比较随意的，这样很难构建一个合理的数据模型。问题是在一个项目中，数据模型是应该最具稳定性的，因此我们的做法是，从建立领域模型开始，通过讨论使设计人员对业务问题慢慢理解的更透彻。边讨论、边建立模型、边使问题的研究更深入，最后构建一个合理的数据模型，本论文撰写者认为，描述这样的过程属于一种分析与设计方法论的研讨，比直接给出设计结论更有意义。

PDM 的一个重要的内容是项目管理与过程管理，因此，我们先从项目管理最基本的问题入手，逐步的建立静态领域模型。首先遇到的第一个领域问题，就是项目管理到底是怎样进行的？

1）任何项目开发特点是先定义里程碑，每个里程碑包含多个任务

我们可以画出里程碑和任务之间的关系，如图 3-3 所示。

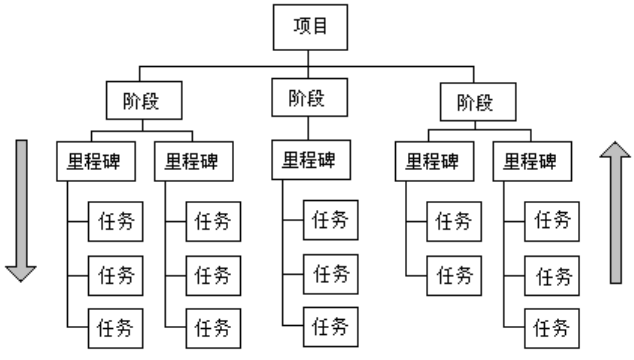


图 3-3 里程碑与任务

现在用类来表示表示概念，那就是领域模型，如图 3-4 所示。

图 3-4 里程碑与任务的领域模型

一个里程碑有多个任务，而且每个任务需要排定日期。

2）每个任务可以分配多个资源，而每个资源也可以提供给多个任务

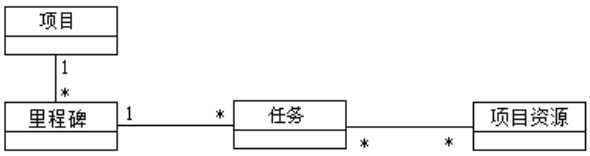


图 3-5 考虑资源的领域模型

3）在 PDM 系统中，每个任务会关系到多个设计图纸和设计方案

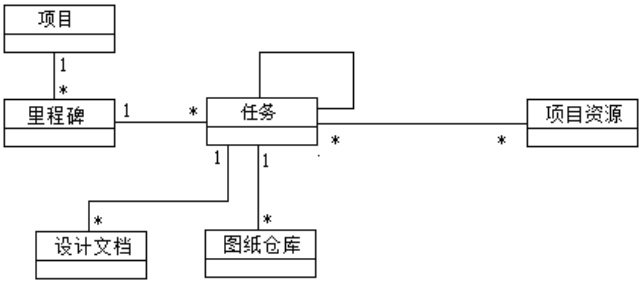


图 3-6 每个任务都可以牵涉到多个设计与图纸

4）事实上，项目涉及的资源有多种，比如人、设备、材料等

不论资源是人员、材料还是设备，它们都需要有属于这个项目资源的基本属性，所以这是一个父子关系。

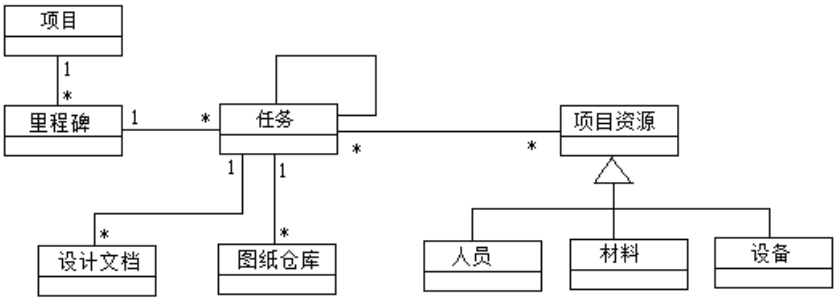


图 3-7 资源包括人员、材料与设备

5）每个里程碑除了有时间、费用和质量三点控制外，必要的时候还会有变更通知

每个里程碑可能有多个变更通知。

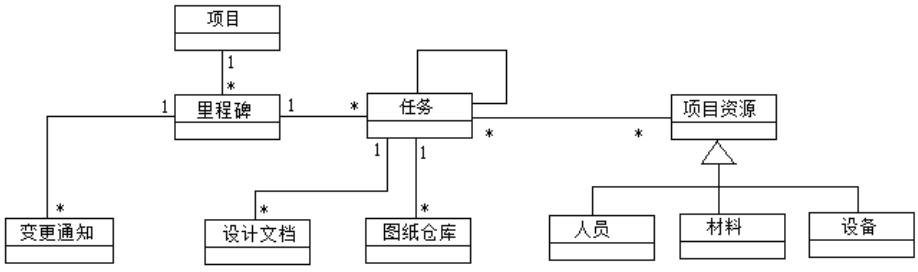


图 3-8 每个里程碑都可能发生多个设计变更

6）有变更通知就需要有变更执行情况记录

每个变更通知，可能需要有多个执行情况记录。

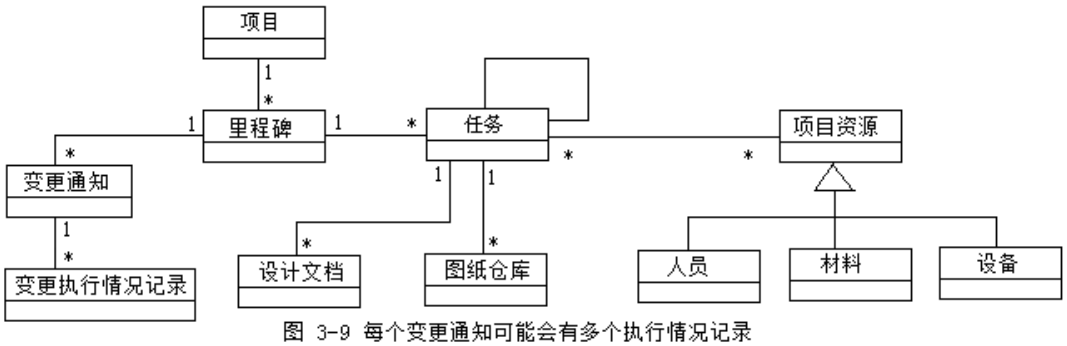


图 3-9 每个变更通知可能会有多个执行情况记录

7）图纸仓库应该包括图纸索引与图纸库两部分

图纸库可以使用 PDM 系统的图纸库，任何 PDM 系统不论功能上多么不一样，这个图纸库是必须存在的。由于图纸库一般都比较大，为了查询方便，一般都使用图纸索引。考虑到图纸的版本演变，以及必要的 Check-in/Check-out 机制，每份图纸会有多个版本。而设计文档也需要同样的版本控制，这都是设计文件配置管理所必需的。由于图纸的复杂性，本项目不采用只保留变化部分的方法，而是记录整个图纸的编办变化演化过程，这样设计上比较现实，而且也符合实际情况。为了保证串行读写机制可以实行，数据中必须记录每个版本的状态，以识别文档当前是处于等待修改、已经在被人修改、还是正在被阅读等状态，使系统可以分配使用文件的人为保留型、非保留型等权限。

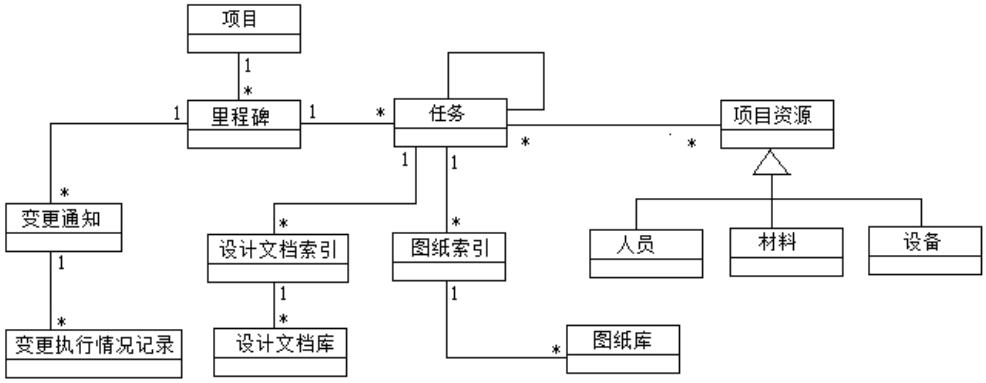


图 3-10 设计文档与图纸可能会有多个版本，需要纳入配置管理体系

把上面的研究结合起来，考虑到我们将使用关系型数据库，屏蔽掉项目资源的父子关系，我们就得到了一个比较清楚的领域模型了。这里，当使用多对多关系的时候，必须添加一个关联表，使这种关系的实际实现成为可能。

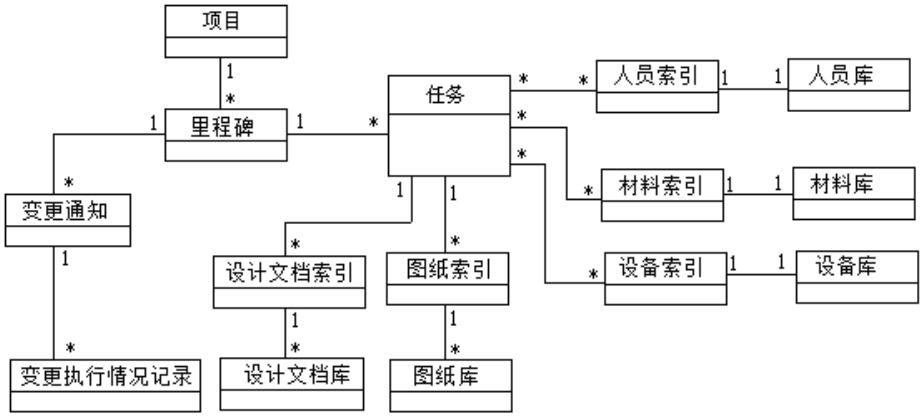


图 3-11 从关系数据库的角度重新考虑结构

在资源部分（人员、材料、设备）使用索引与库分开的 1 对 1 关系的原因，是索引记录的与本项

目相关的信息，而库中记录的是资源本身更通用的信息。这样的数据结构便于在更广泛的领域中使用。对于多对多关系，需要建立一个关联表，根据不同的用途，关联表的方案可以有多种。我们考虑在本项目情况下，无论人员、设备还是材料，与任务的关系比较复杂，所以关联表决定采用“联合主键”的方式。

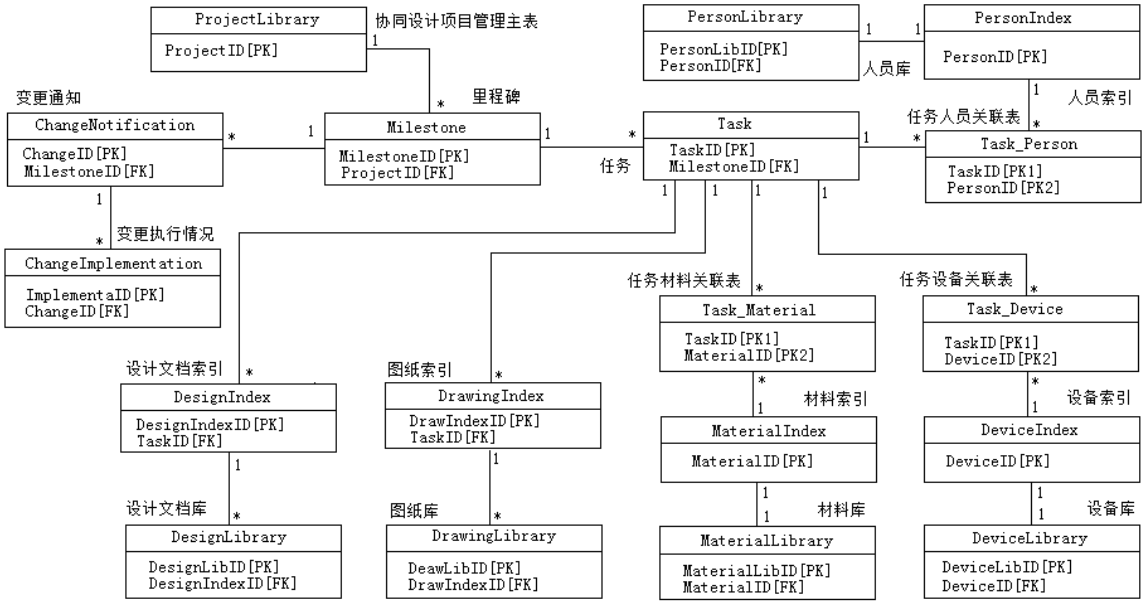
此种关联表的建立方法我们以人员为例：

首先确定人员列表，由任务决定选择哪些人员（自由选择），每做一个决定，关联表就增加一行，当然也可以把选中的人员去掉，表现为删除关联表中的相应行，

这种表结构的查询方法是，不论是以任务为基点还是以人员为基点，先从与主键匹配的关联表主键列找到所有与相应字段匹配的行，然后查询对应的另一方，就可以在另一个表中把所有对应记录找出来。

三、数据模型的建立

依据我们已经建立的领域模型，再仔细思考每个领域类的属性（也就是每一张表的字段），就可以建立一个初步的 PDM 协同设计系统关于项目管理部分的数据模型了，如下图所示,图中只表达了各个表之间的关系，以及为建立关系有关的主键和外键，以表达图中最主要的关系。在字段比较复杂的时候，字段的细节还是用表格表达比较合适。



各表的字段说明如下，此表经过了简化，只反映最主要的内容。

ProjectLibrary			
协同设计项目管理主表			
序号	名称	类型	说明
1	ProjectID	CHAR(30)	主键（PK1）
2	Name	NCHAR(30)	项目名称
3	StartDate	DATE	项目开始日期
4	EndDate	DATE	项目结束日期
5	Responsibl	NCHAR(20)	项目负责人
6	Units	NCHAR(40)	项目单位
7	Department	NCHAR(40)	项目部门
8	State	NCHAR(225)	项目状态

Milestone			
里程碑			
序号	名称	类型	说明
1	MilestoneID	CHAR(30)	主键（PK1）
2	ProjectID	CHAR(30)	外键（FK）
3	Description	NCHAR(225)	里程碑说明
4	Time	DATE	里程碑检查时间
5	Check	NVARCHAR2	里程碑检查要点
6	Shortcomings	NCHAR(225)	里程碑检查问题
7	Improve	BLOB	里程碑检查改进方案

ChangeNotification			
变更通知			
序号	名称	类型	说明
1	ChangeID	CHAR(30)	主键（PK）
2	MilestoneID	CHAR(30)	外键（FK）
3	Notice	NVARCHAR2	设计变更通知
4	TaskBook	BLOB	设计变更任务书
5	Time	DATE	变更时间
6	Location	NCHAR(225)	变更任务位置
7	Published	NCHAR(20)	变更下达人
8	Perform	NCHAR(20)	变更执行人

ChangeImplementation			
变更执行情况			
序号	名称	类型	说明
1	ImplementaID	CHAR(30)	主键（PK）
2	ChangeID	CHAR(30)	外键（FK）
3	Situation	NVARCHAR2	执行状态
4	PerDate	DATE	执行日期
5	Problems	BLOB	存在问题记录
6	Person	NCHAR(20)	变更执行人

Task			
任务			
序号	名称	类型	说明
1	TaskID	CHAR(30)	主键（PK）
2	MilestoneID	CHAR(30)	外键（FK1）
3	Description	NVARCHAR2	任务说明
4	Time	CHAR(20)	任务时间段
5	StartDate	DATE	任务开始日期
6	EndDate	DATE	任务结束日期
7	Person	NCHAR(20)	任务承担人
8	TaskState	NVARCHAR2	任务完成状态
9	Problems	NVARCHAR2	任务存在问题
10	Improve	BLOB	问题解决方案

DesignIndex			
设计文档索引			
序号	名称	类型	说明
1	DesignIndexID	CHAR(30)	主键（PK）
2	TaskID	CHAR(30)	外键（FK）
3	DesignNo	CHAR(50)	文档编号
4	DesignName	NCHAR(50)	文档名称

5	TaskBook	BLOB	设计任务书
6	DesignPerson	NCHAR(20)	设计人

DesignLibrary			
设计文档库			
序号	名称	类型	说明
1	DesignLibID	CHAR(30)	主键（PK）
2	DesignIndexID	CHAR(30)	外键（FK）
3	Version	CHAR(30)	文档版本号
4	Description	BLOB	文档说明
5	ModifiedPerson	NCHAR(20)	修改人
6	Modified	BLOB	文档修改记录
7	File	BLOB	设计文档
8	CheckState	CHAR(8)	当前状态（为检入/检出机制设置）

DrawingIndex			
图纸索引			
序号	名称	类型	说明
1	DrawIndexID	CHAR(30)	主键（PK）
2	TaskID	CHAR(30)	外键（FK）
3	DrawNo	CHAR(40)	图纸编号
4	Name	NCHAR(50)	图纸名称
5	TaskBook	BLOB	图纸任务书
6	DesignPerson	NCHAR(20)	设计人

DrawingLibrary			
图纸库			
序号	名称	类型	说明
1	DeawLibID	CHAR(30)	主键（PK）
2	DrawIndexID	CHAR(30)	外键（FK）
3	Version	CHAR(30)	图纸版本号
4	Description	NVARCHAR2	图纸说明
5	ModifiedPerson	NCHAR(20)	修改人
6	Modified	NVARCHAR2	修改记录
7	Draw	BLOB	图纸
8	Video	BLOB	视频
9	CheckState	CHAR(8)	当前状态（为检入/检出机制设置）

Task_Person			
任务人员关联表			
序号	名称	类型	说明
1	TaskID	CHAR(30)	主键 1（PK1）
2	PersonID	CHAR(30)	主键 2（PK2）

PersonIndex			
人员索引			
序号	名称	类型	说明
1	PersonID	CHAR(30)	主键（PK）
2	TaskBrief	CHAR(30)	任务简述
3	TaskBook	BLOB	任务书
4	Responsibilities	NVARCHAR2	工作职责

PersonLibrary
人员库

序号	名称	类型	说明
1	PersonLibID	CHAR(30)	主键（PK）
2	PersonID	CHAR(30)	外键（FK）
3	PersonNo	CHAR(20)	人员编号
4	Name	NCHAR(20)	姓名
5	Sex	NCHAR(10)	性别
6	Age	DATE	成生年月
7	Education	NCHAR(20)	学历
8	EmployData	DATE	参加工作时间
9	Position	NCHAR(30)	职务
10	Department	MCHAR(30)	所在部门
11	Wage	NUMBER	工资
12	Remark	NVARCHAR2	备注

Task_Material			
任务材料关联表			
序号	名称	类型	说明
1	TaskID	CHAR(30)	主键 1（PK1）
2	MaterialID	CHAR(30)	主键 2（PK2）

MaterialIndex			
材料索引			
序号	名称	类型	说明
1	MaterialID	CHAR(30)	主键（PK）
2	MaterUse	CHAR(30)	材料用途
3	MaterAmount	CHAR(50)	材料用量
4	TRAC	BLOB	材料调拨单
5	MaterHelp	BLOB	材料使用说明

MaterialLibrary			
材料库			
序号	名称	类型	说明
1	MaterialLibID	CHAR(30)	主键（PK）
2	MaterialID	CHAR(30)	外键（FK）
3	MaterNo	CHAR(30)	材料编号
4	Name	NHAR(50)	材料名称
5	Description	BLOB	材料说明
6	Consumption	BLOB	消耗记录
7	Details	BLOB	材料明细

Task_Device			
任务设备关联表			
序号	名称	类型	说明
1	TaskID	CHAR(30)	主键 1（PK1）
2	DeviceID	CHAR(30)	主键 2（PK2）

DeviceIndex			
设备索引			
序号	名称	类型	说明
1	DeviceID	CHAR(30)	主键（PK）
2	Reasons	CHAR(30)	设备使用原因
3	TRAC	BLOB	设备调拨单
4	Units	NCHAR(50)	设备使用单位
5	Duty	NCHAR(20)	设备责任人

DeviceLibrary			
设备库			
序号	名称	类型	说明
1	DeviceLibID	CHAR(30)	主键（PK）
2	DeviceID	CHAR(30)	外键（FK）
3	DeviceNo	CHAR(40)	设备编号
4	DeviceName	NCHAR(50)	设备名称
5	Description	BLOB	设备说明
6	State	NCHAR(125)	设备状态
7	StartDate	DATE	设备启用时间
8	EndDate	DATE	设备终止时间
9	Depreciation	NVARCHAR2	设备折旧状况
10	Details	BLOB	设备明细

我们从这个 PDM 系统数据结构的设计中应该体会到，在面对一个复杂系统的时候，没有章法的胡乱把数据库表结构做出来是不可取的。由于数据库结构是系统希望最为稳定的部分，所以认真研究深入体会整个结构和概念，形成正确的设计思想十分重要。我们可以发现，正确进行领域建模，把领域模型通过适当的变换，再过渡到实际数据模型是一个很好的方法，这可以使数据建模达到事半功倍的效果，也为未来的设计能力的提升打下了方法论的基础。

四、具有完整属性的数据模型以及规范化分析

体系结构完成以后，可以考虑写出属性，每个属性对应一个字段。由于这个例子太复杂，而且详细写出属性并不能给我们提供更多的知识，所以这里就不再讨论了。

然后必须进行规范化分析，也就是根据第 1 范式（1NF），函数相关，第 2 范式（2NF），第 3 范式（3NF）对表结构和数据进行检查，修改其中的不合理成分，这样数据库分析和设计可以告一段落。上面的分析过程不论是面向过程还是面向对象，方法上几乎是相同的。

第七章 高层软件体系结构的设计

在高层设计阶段，主要工作是分析与设计软件的体系结构。通过系统分解，确定子系统的功能和子系统之间的关系，以及模块的功能和模块之间的关系，产生《体系结构设计报告》。

在分析阶段，我们建立模型表示真实的世界，以便理解业务过程以及这个过程中所要用到的信息。基本上说，分析首先是分解，把复杂信息需求的综合问题，分解成易于理解的多个小问题。然后通过建立需求模型来对问题概念进行组织、构造并且编制文档。分析建模过程必须要用户参与，并且需要用户解释需求，并且验证建立的模型是否正确。

体系结构设计实际上也是个建模过程，它把分析阶段得出的信息也就是需求模型，转换为称之为**解决方案**的模型。一般来说体系结构设计是一个高度技术的工作，一般不需要涉及太多的用户，但需要系统分析人员和部分开发人员参与。因为系统设计的输出就是开发的蓝图。

下面讨论在这一阶段一系列的原则和思想。

7.1 高层体系结构设计的基本原则

所谓软件设计是研究下面几个问题：

1，软件设计是把软件需求转变为软件的具体方案

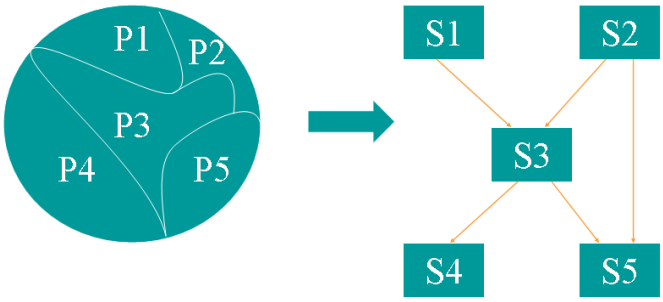
2，软件设计包括两个阶段：高层设计（或者称概要设计）和详细设计

3，高层设计根据软件需求所确定的信息流程或信息结构，导出软件的总体表示。

一、软件体系结构设计的基本方法

1，软件体系结构是一种层次化的表示，其指出了由需求分析隐含地确定的某一问题的软件解法的各个元素（称之为模块）之间的相互控制关系。

2，软件体系结构的演变从确定问题开始，当该问题的每个部分用一个或多个软件加以解决以后，整个问题的解决方案也就有了。



软件体系结构设计方法：

1）逐步精化----自顶向下设计方法

2）面向过程的体系结构设计

面向过程的体系结构设计的基础建立在三种能够构成结构化程序的逻辑构造（顺序，选择，重复）上。

3）面向数据的设计方法

面向数据流的设计，面向数据结构的设计。

4）面向对象的设计方法

二、软件体系结构的度量和术语

1，深度：表示控制的层数。
2，宽度：表示控制（同一层次）总跨度。
3，扇出数：指由一个模块直接控制的其它模块的数目。
4，扇入数：指有多少个模块直接控制一个给定的模块。
5，上级模块。
6，下级模块。

程序过程：

程序过程是用于描述每个模块的操作细节，是关于模块算法的详细描述，它应当包括处理的顺序、精确的判定位置、重复的操作以及数据组织和结构等。

模块：

模块是数据说明、可执行语句等程序对象的集合，是单独命名的并且可以通过名字来访问，例如过程、函数、子程序、宏等。

模块化：

软件被划分成独立命名和可独立访问的被称作模块的构件，每个模块完成一个子功能，它们集成到一起可以满足问题需求。

实现模块化的手段：

1，抽象：抽出事物的本质特性而暂时不考虑它们的细节。

2，信息隐蔽：应该这样设计和确定模块，使得一个模块内包含的信息（过程和数据）对于不需要这些信息的模块来说，是不可访问的。

模块独立性：

1，模块独立是指开发具有独立功能而且和其它模块之间没有过多的相互作用的模块。
2，模块独立的意义：
1）功能分割，简化接口，易于多人合作开发同一软件；
2）独立的模块易于测试和维护。
3，模块独立程度的衡量标准：
1）耦合性：对一个软件结构内不同模块间互连程度的度量。
2）内聚性：标志一个模块内各个处理元素彼此结合的紧密程度，理想的内聚模块只做一件事情。

耦合分类：

1，无任何连接：两个模块中的每一个都能独立地工作而不需要另一个的存在（最低耦合）。
2，数据耦合：两个模块彼此通过参数交换信息，且交换的仅仅是数据（低耦合）。
3，控制耦合：两个模块之间传递的信息有控制成分（中耦合）。
4，公共环境耦合：两个或多个模块通过一个公共环境相互作用：
1）一个存数据，一个取数据（低耦合）；
2）都存取数据（低--中之间）。
5，内容耦合（一般比较高）：
1）一个模块访问另一个模块的内部数据；
2）两个模块有一部分程序代码重叠；
3）一个模块不通过正常入口而转移到另一个的内部；
4）一个模块有多个入口（意味着该模块有多个功能）。

内聚分类：

1，功能内聚：一个模块完成一个且仅完成一个功能（高）。

2，顺序内聚：模块中的每个元素都是与同一功能紧密相关，一个元素的输出是下一个元素的输入（高）。

3，信息内聚：模块内所有元素都引用相同的输入或输出数据集合（中）。

4，时间内聚：一组任务必须在同一段时间内执行（低）。

5，逻辑内聚：一组任务在逻辑上同属一类，例如均为输出（低）。

6，偶然内聚：一组任务关系松散（低）。

关于耦合性和内聚性的设计原则：

1，在模块职责分配上低耦合模式的主要目的在于：支持低的依赖性，减少变更带来的影响，提高重用性

2，力争尽可能弱的耦合性：尽量使用数据耦合，少用控制耦合，限制公共环境耦合的范围，完全不用内容耦合。

3，力争尽可能高的内聚性：力争尽可能高的内聚性，并能识别出低内聚性。

三、高层体系结构设计的一般准则

1，改进软件结构，提高模块独立性。

2，模块规模应该适中（最好能写在一页纸上）。

大模块分解不充分；小模块使用开销大，接口复杂。

3，尽量减少高扇出结构的数目，随着深度的增加争取更多的扇入。

扇出过大意味着模块过分复杂，需要控制和协调过多的下级模块。一般来说，顶层扇出高，中间扇出少，低层高扇入。

4，模块的作用范围保持在该模块的控制范围内。

模块的作用范围是指该模块中一个判断所影响的所有其它模块;模块的控制范围指该模块本身以及所有直接或间接从属于它的模块。

5，力争降低模块接口的复杂程度

模块接口的复杂性是引起软件错误的一个主要原因。接口设计应该使得信息传递简单并且与模块的功能一致。

6，设计单入口单出口的模块

避免内容耦合，易于理解和维护。

7，模块的功能应该可以预测

相同的输入应该有相同的输出，否则难以理解、测试和维护。

7.2 系统设计的应用体系结构策略

事实上所有的信息系统都是一个应用体系结构，不同的组织使用不同的策略来确定应用体系结构，下面介绍两种最常用的方法。

一、企业应用体系结构策略

在企业应用体系结构策略中，组织开发一个企业及的信息技术体系结构，所有的后续开发项目，都应该遵循这个体系结构，这个体系结构定义了以下内容：

1）约束网络、数据、接口和过程技术以及开发工具（包括软件、硬件、客户端与服务器）。

2）集成已存在系统和技术、或者新系统到应用体系结构中的策略。

3）连续地检查应用体系结构的正确性和适应性的不断进行的过程。

4）研究新技术并推荐将其纳入应用体系结构的的不断进行的过程。

5）对已经实现的应用体系结构修改请求的分析过程。

初始的企业应用体系结构通常是作为独立的项目开发的，或者是作为战略信息系统规划项目的一部分开发的。对应用体系结构的维护工作，通常由一个常设的信息技术研究组担负。在认可了应用体系结构以后，期望每个信息系统开发项目都根据这个体系结构使用或者选择技术。在大多数情况下，这会极大的简化系统开发或者体系结构阶段的工作。

二、战术应用体系结构策略

如果没有企业级的应用体系结构，每个项目就需要定义自己的体系结构。这种体系结构形式对于开发人员使用新技术有更大的自由度，不过需要进行三方面的可行性研究：

1）技术可行性研究：对该技术的成熟度的度量，该技术对于正在设计的项目适应度的度量，或者该技术与其它技术以其工作能力的度量。

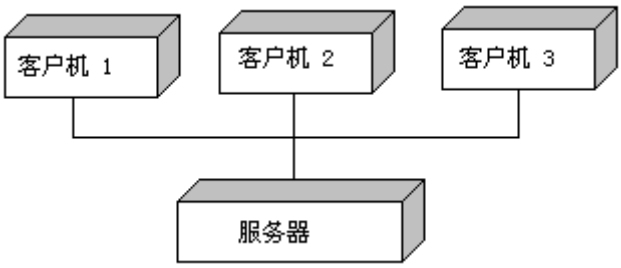
2）运行可行性研究：企业管理人员和用户对于该技术的熟悉程度，以及管理人员和支持人员对于该技术的熟悉程度的度量。

3）经济可行性：使用该技术从经济的角度是否合算的度量。

这些度量对于体系结构风格的选择是非常有意义的。

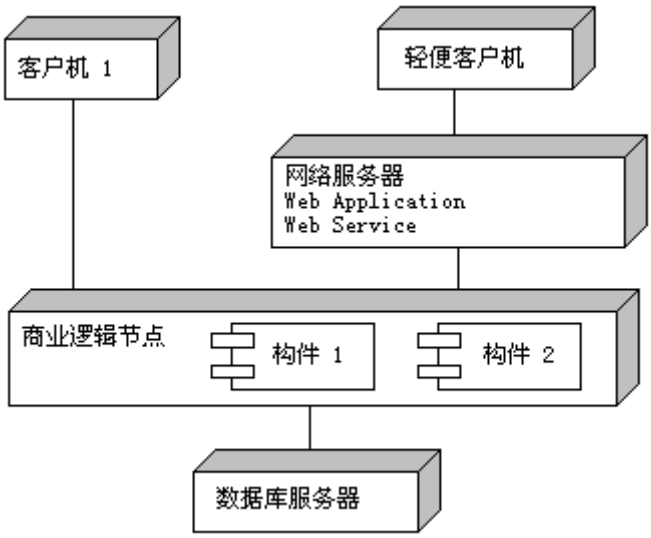
7.3 高层软件体系结构的规划

一、客户服务结构（C/S architecture）



客户/服务体系结构是支持应用系统的软件技术发展的结果，面对日益膨胀的应用业务处理，客户/服务体系结构的发展已经成为信息技术膨胀中的重要因素。客户/服务体系最初的动因是文件服务器，后来发展为数据库服务器，当初是为了解决大型机技术里数据和管理完全集中的缺点而创立的，但由于维护性和升级性上的种种缺点，目前正在被 N 层面向构件的解决方案所替代。

二、多级体系结构（four-tier architecture）



N 层面向构件的解决方案，主要基于这样几点考虑：

1）减少客户机的维护量，因为前台程序比较简单；

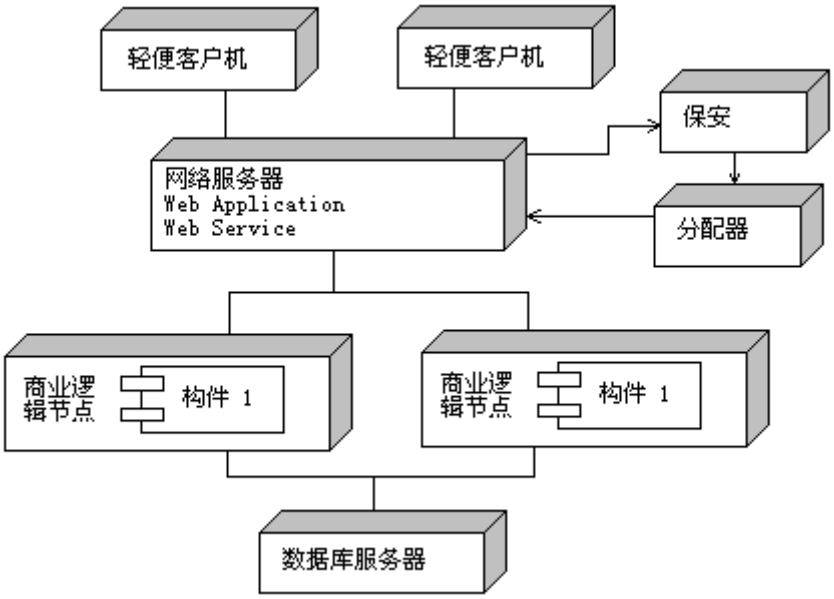
把企业逻辑封装在通用的中间件应用服务器中，不同的客户都可以共享同一个中间层（包括Web），而不必每个客户都单独实现企业规则，避免了重复开发和维护的麻烦。由于客户程序相当瘦（这就是现在流行的瘦客户机概念），无论是开发还是发布，都变得简单了。

2）便于升级，当中间件升级的时候，客户程序可能不需要变化；

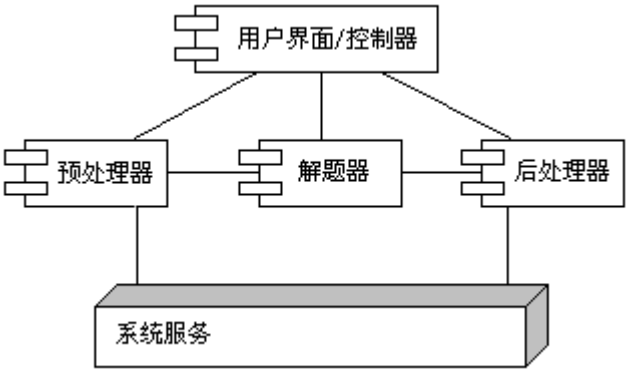
3）实现了分布式数据处理，把一个应用程序分布在几台机器上运行，可以提高应用程序的性能，也可以把敏感部分封装在中间件，为不同的用户设置不同的访问权限，增强了安全性。

4）减少直接连接数据库的用户数目，减少费用。

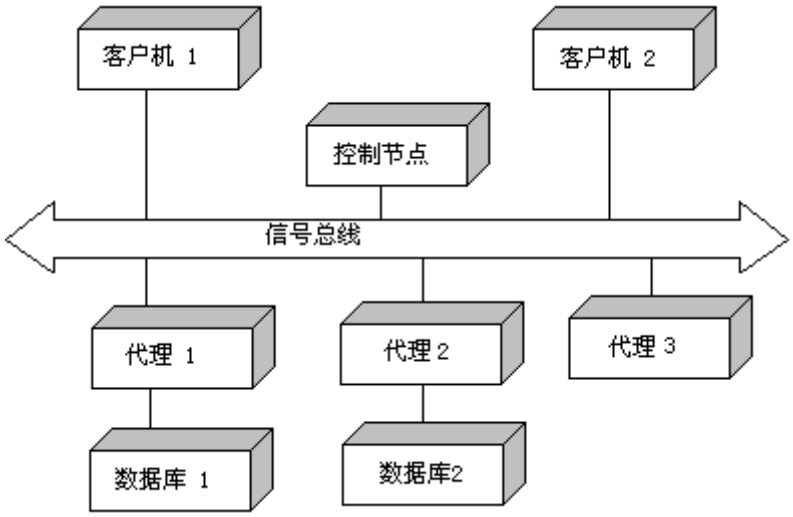
三、多级体系结构（串行法和团聚法）



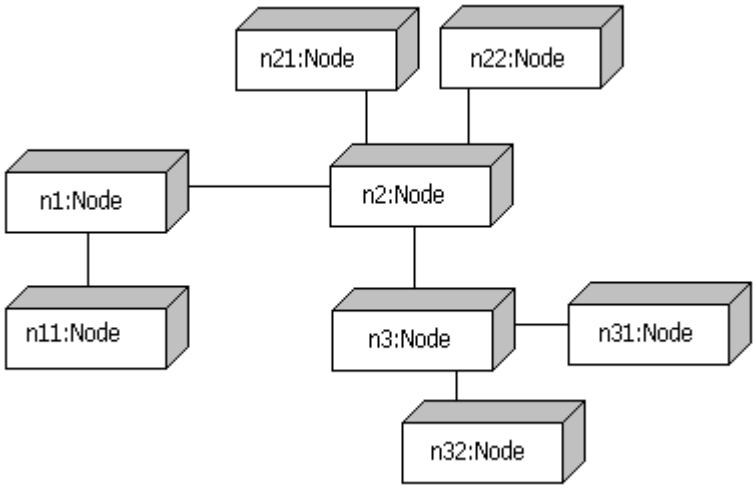
四、流处理体系结构（procedural prcessing architecture）



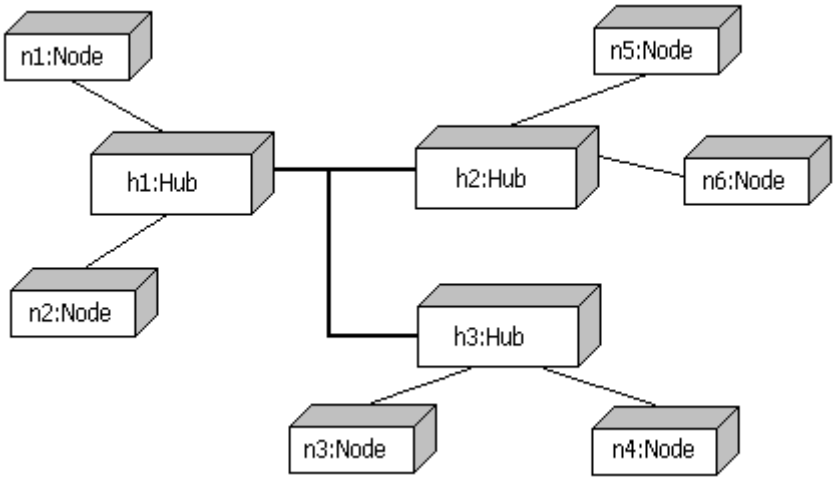
五、代理体系结构（agent architecture）



六、聚合体系结构（aggregate architecture）



七、联邦体系结构（federation architecture）



7.4 面向过程的体系结构设计

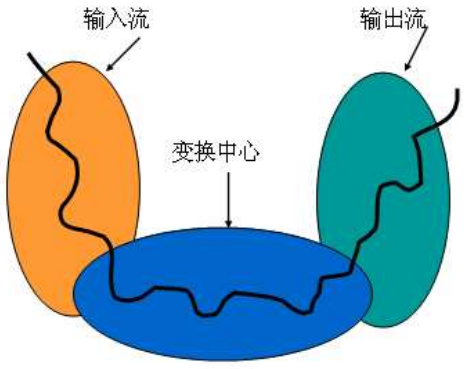
面向过程的体系结构设计，又称之为结构化设计。设计的结果，需要把数据流程图（DFD）转换成相应的模块。它使用“输入 – 处理 – 输出”这样一个基本模型，这些模式比较适用于描述商业软件，它们中大多数依靠数据库或者文件，并且不太需要复杂的实时处理。我们可以使用基于数据流的设计，也可以使用流程图来记录各个子系统的结构，系统流程图标识了每个程序，以及他们存取的数据。

一、基于数据流的设计

1）面向数据流的设计方法把信息流映射成软件结构
2）信息流的类型决定了映射的方法
3）信息流有两种类型：变换流和事务流。

1，变换流：

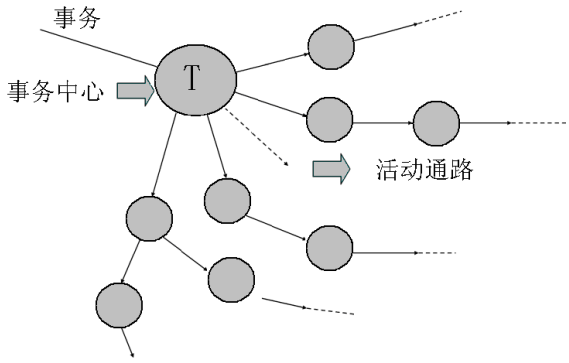
信息沿输入通路进入系统，同时由外部形式变换成内部形式。进入系统的信息通过变换中心，经过加工处理以后再沿着输出通路变换成外部形式离开系统。



2，事务流：

事务流的特点是数据沿着接收通路把外部世界的信息转换成一个事务项，然后，计算该事务

项的值，根据它的值激励起多条活动通路中的一条数据流。发出多条通路的信息流中枢被称为“事务中心”。



3，变换型分析：

第 1 步　　复查基本系统模型。

第 2 步　　复查并精化数据流图。

第 3 步　　确定数据流图具有变换特性还是事务特性。

第 4 步　　确定输入流和输出流的边界，从而孤立出变换中心。

第 5 步　　完成“第一级分解”。

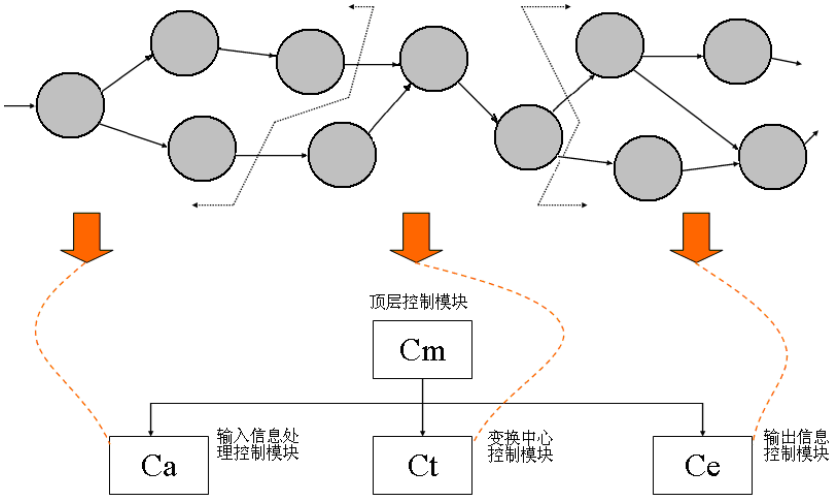
软件结构代表对控制的自顶向下的分配，所谓分解就是分配控制的过程。

对于变换流，数据图将被映射成一个特殊的软件结构，这个结构控制输入、变换和输出信息等处理过程：位于软件结构最顶层的控制模块 Cm 协调下述从属的控制功能：

（1）输入信息处理控制模块 Ca，协调对所有输入数据的接收；

（2）变换中心控制模块 Ct，管理对内部形式的数据的所有操作；

（3）输出信息控制模块 Ce，协调输出信息的产生过程。



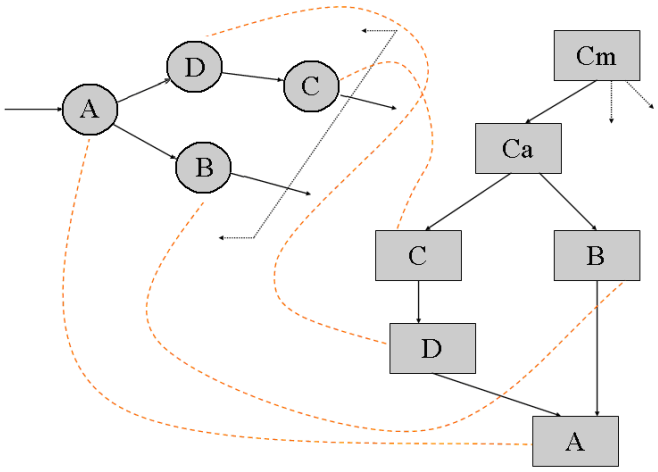
第 6 步　　完成“第二级分解”。

把数据流图中的每一个处理映射成软件结构中一个适当的模块：

1）从变换中心的边界开始沿着输入通路向外移动，把输入通路中每个处理映射成软件结构中 Ca 控制下的一个低层模块；

2）然后沿输出通路向外移动，把输出通路中每个处理映射成直接或间接受 Ce 控制的一个低层模块；

3）最后把变换中心内的每个处理映射成受 Ct 控制的一个模块。



分析这张图的时候，我们需要注意到数据流程图与结构图的箭头含义是不同的，数据流程图的箭头表达的是数据的流向，而结构图的箭头表达的是自顶向下的结构关系。当需要在结构图中表达数据和控制的流向时，就必然的希望增加某些表示符号。

第 7 步　　使用耦合性和内聚性规则对得到的软件结构进一步精化。

4）事务型分析：

第 1 步　　复查基本系统模型。

第 2 步　　复查并精化数据流图。

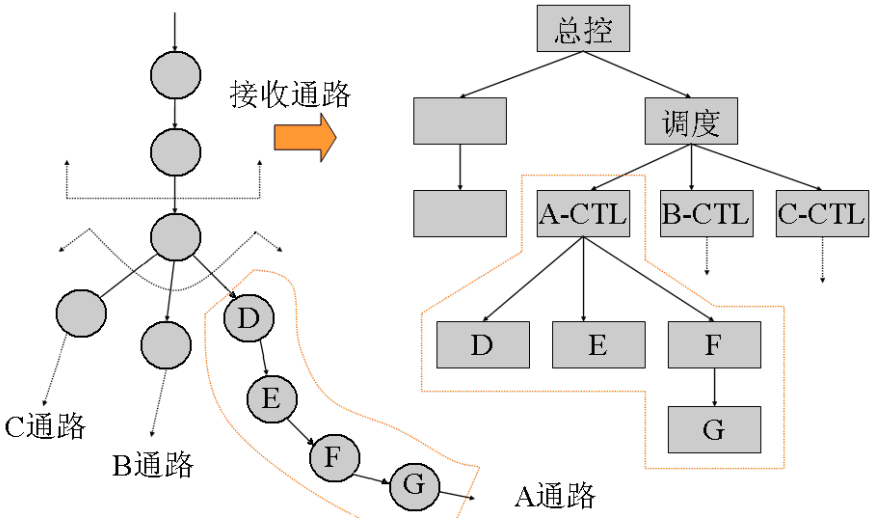
第 3 步　　确定数据流图具有变换特性还是事务特性。

第 4 步　　确定事务中心和每个活动通路的流程特征。

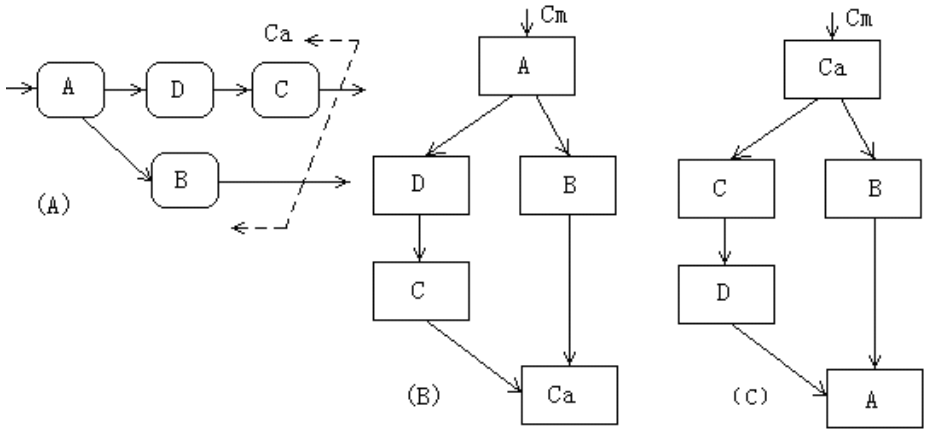
第 5 步　　把数据流图映射成一个适合于事务处理的软件结构。

第 6 步　　对事务中心的结构和每个活动通路的结构进行分解、合并和改进。

第 7 步　　使用耦合性和内聚性规则对得到的软件结构进一步精化。



上述这种映射变换关系有时候会发生错误，很多人把过程图直接映射为结构图，这就会出现问题。如下图所示：位于软件结构最顶层的控制模块 Cm 协调从属的控制功能，而 Ca 为输入信息处理模块，在图 A 中，从变换中心的边界开始沿着输入通路向外移动，如果把输入通路中每个过程映射成图 B 所示的软件结构就错了，正确的应该如图 C 所示。

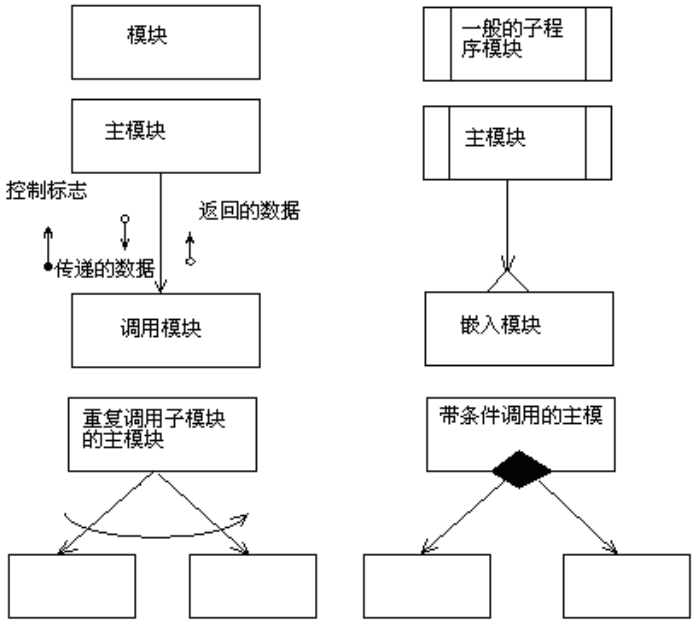


二、模块算法设计（伪码）

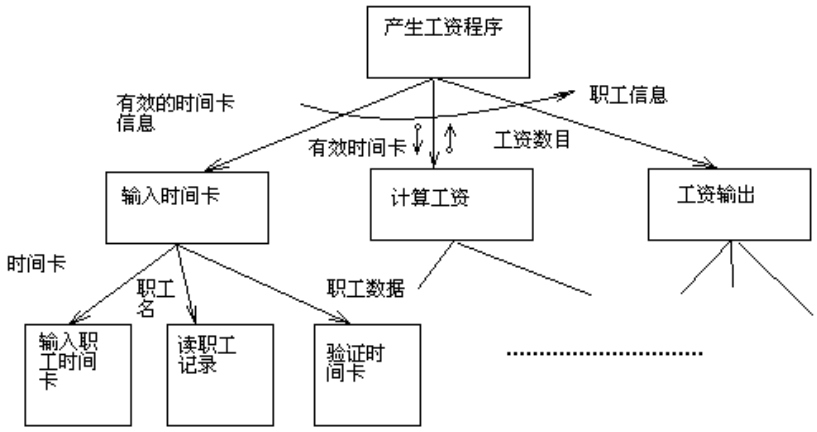
结构 化设计的另一个需求，是描述每个模块的内部逻辑，我们可以用自己熟悉的语言来定义伪码（比如 C），使用伪码并不是写出程序，而是为了更清楚地描述模块级的逻辑。这样也可以避免各种图泛滥成灾。

三、结构图的进一步细化

结构化设计的基本任务，是自顶向下的分解任务，结构图是用来展示计算机程序模块之间的层次关系。在上面的讨论中，我们只表达了自顶向下的结构关系，但并没有表达结构中的控制和数据的走向，为了更清楚的表达这些内容，需要给结构图增加一些符号，目前常用的结构图主要符号如下：



下面是一个工资系统的部分结构图。



四、基于数据结构的设计

1，基于数据结构的设计方法用信息结构导出程序过程

2，基于数据结构的设计过程分为如下几步：

1）分析数据结构的特性；

2）用一些基本类型（如：顺序，选择和重复）来描述数据；

3）把数据结构表示映射成软件的控制层次；

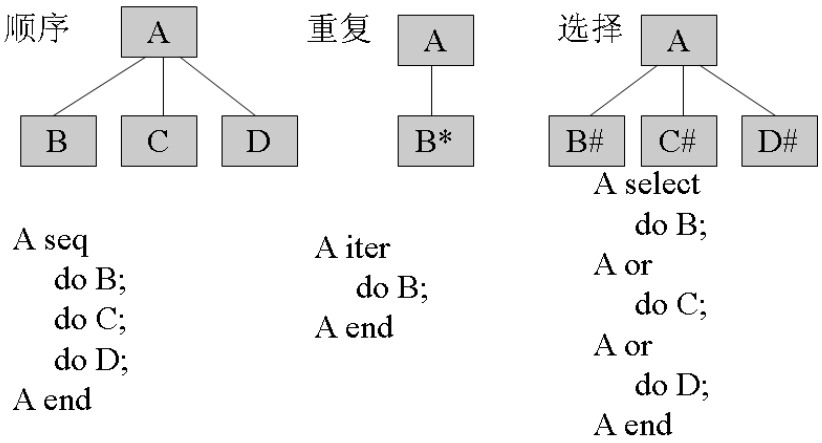
4）利用一组规则改进软件的层次结构；

5）最后得到软件的过程性描述。

五、Jackson 方法

1，Jackson 方法是一种基于数据结构的设计方法，同时也是一种非常典型的面向过程方法，它的精髓在于：应该把问题分解成仅用三种结构化形式（顺序，选择和重复）来表示的层次结构。

2，Jackson 方法包括一种数据结构符号和一组映射或转换步骤。



Jackson 图的特点：

1，便于表示层次结构，而且是对结构进行自顶向下分解的有力工具；

2，形象直观，可读性好；

3，既能表示数据结构，又能表示程序结构。

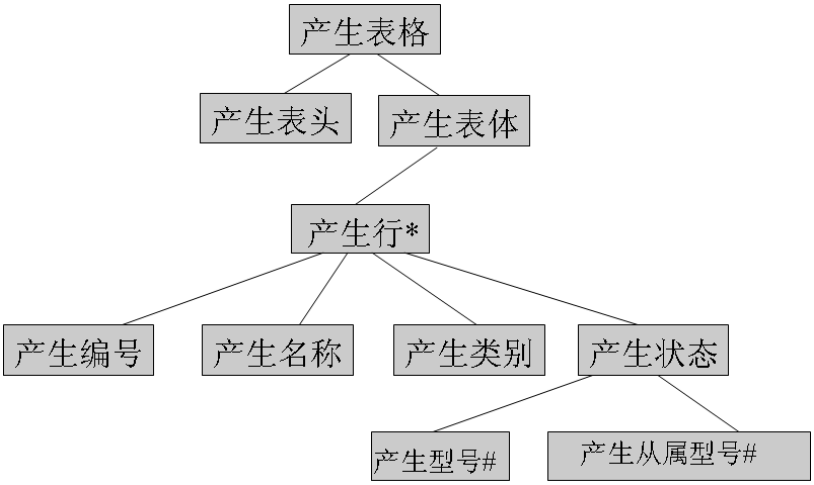
建立程序结构例：

设想我们需要在电源仓库管理子系统中设计一个报表打印程序。表格如下：

编号	名称	类别	状态

这里类别可以是“主机”或“附件”两种。“状态”一项，如果是主机则印出它的“型号”，如果是附件则印出它的“从属主机型号”。

描述数据结构的 Jackson 图：

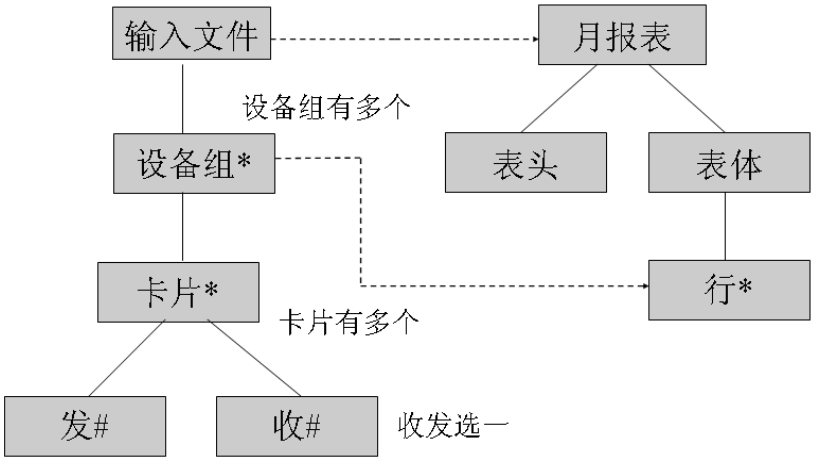


案例：电源设备报表打印

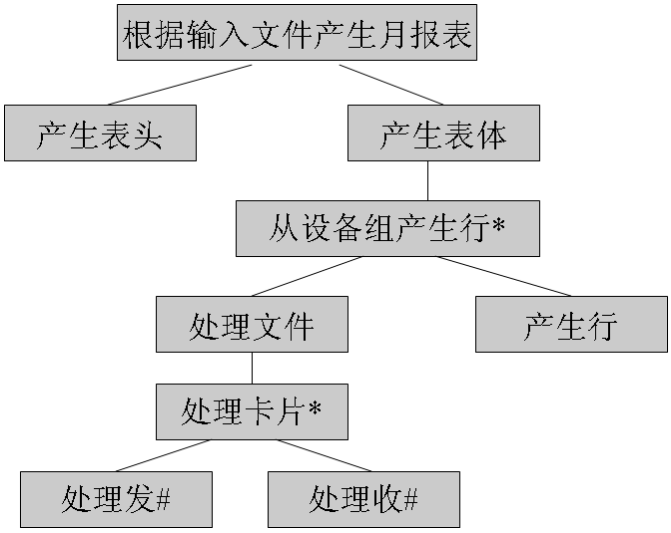
在电源销售服务系统中，我们考虑一下仓库管理系统中的报表打印模块。

电源设备仓库中存放了多种设备，每种设备的每次变动（收到或发出）都有一张卡片作出记录，库存管理系统每月要根据这些卡片打印一张月报表，列出各种设备在这个月中库存量的净变化。

首先画出描述数据结构的 Jackson 图：



再考虑相应的描述程序结构的 Jackson 图：



Jackson 方法的基本步骤：

1，分析并确定输入数据和输出数据的逻辑结构，并用 Jackson 图描述这些数据结构；

2，找出输入数据和输出数据结构中有对应关系的数据单元。所谓对应关系是指有直接的因果关系，在程序中可以同时处理的数据单元（对于重复出现的数据单元必须重复的次数相同才可能有对应关系）；

3，用下述三条规则从描述数据结构的 Jackson 图导出描述程序结构的 Jackson 图：

第一，为每对有对应关系的数据单元，按照它们在数据结构图中的层次在程序结构图的相应层次画一个处理框（注意，若这对数据单元在输入数据结构和输出数据结构中所处的层次不同，则和它们对应的处理框在程序结构图中所处的层次与它们之中在数据结构图中层次低的那个对应）；

第二，根据输入数据结构中剩余的每个数据单元所处的层次，在程序结构图中的相应层次分别为它们画上对应的处理框；

第三，根据输出数据结构中剩余的每个数据单元所处的层次，在程序结构图中的相应层次分别为它们画上对应的处理框。

4，列出所有操作和条件（包括分支条件和循环结束条件），并且把它们分配到程序结构图的适当位置。

5，用伪码标示程序。

例：输入一个文件 FIPT，此文件只包含三种记录类型 T1、T2 和 T3，现在要对该文件作如下处理：

（1）统计出现的第一个 T1 类型的记录前的记录总数（计数 A），我们称之为“前缀”。

（2）显示第一个 T1 类型的记录；

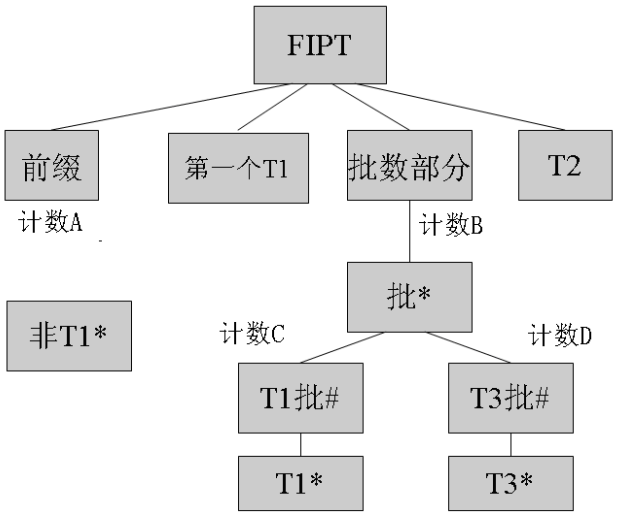
（3）计算第一个 T1 类型的记录后的记录批数（一批记录指一串连续的 T1 类型，的记录或一串连续的 T3 类型的记录（计数 B）；

（4）显示最后一个记录，最后一个记录是在第一个 T1 类型的记录后的第一个 T2 类型的记录。这就完成了第一个层次的顺序处理。

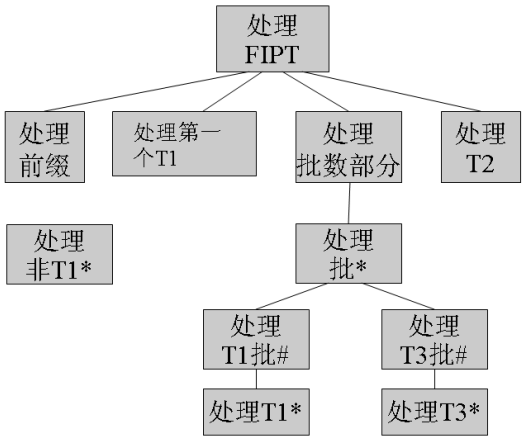
（5）统计在第一个 T1 类型的记录后出现的 T1 类型记录的总数（计数 C）；

（6）计算在第一个 T1 类型的记录后的 T3 类型记录的批数（计数 D）。

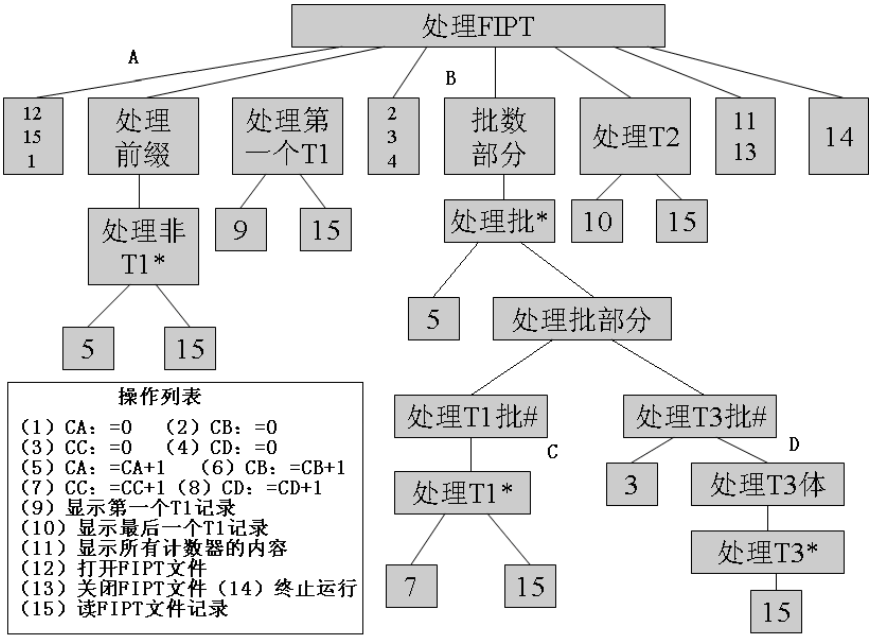
首先画出描述数据结构的 Jackson 图：



再考虑相应的描述程序结构的 Jackson 图：

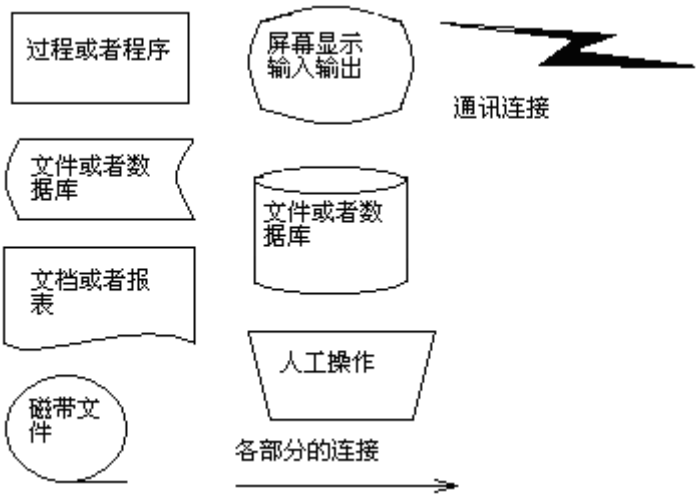


对结构图进一步细化，列出所有的操作，其中 CA、CB、CC、CD 为四个计数变量，图中的数字对应操作列表中的一个操作。

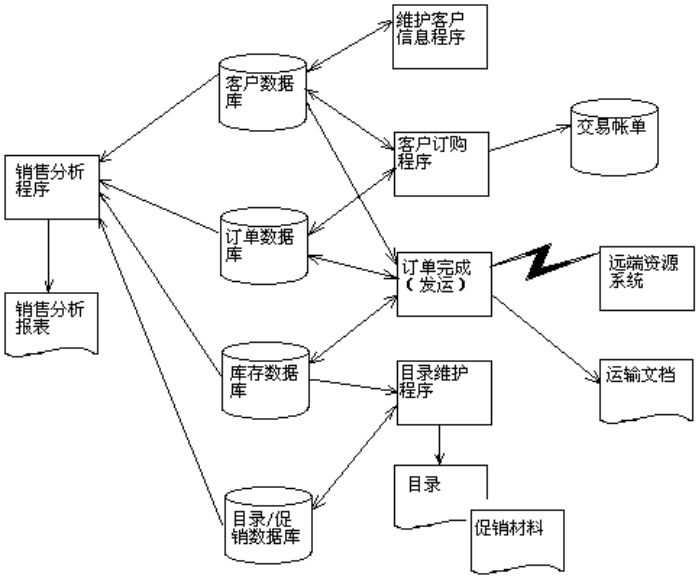


六、系统流程图

系统流程图是用图形的方式描述哪些子系统是系统自动完成的，哪些是需要人工参与的，并且显示了数据流和控制流。系统流程图主要描述大的信息系统，这种大的信息系统由单个的子系统和大量的程序块组成。绘制流程图使用的主要符号如下，也可以有其它的变体。

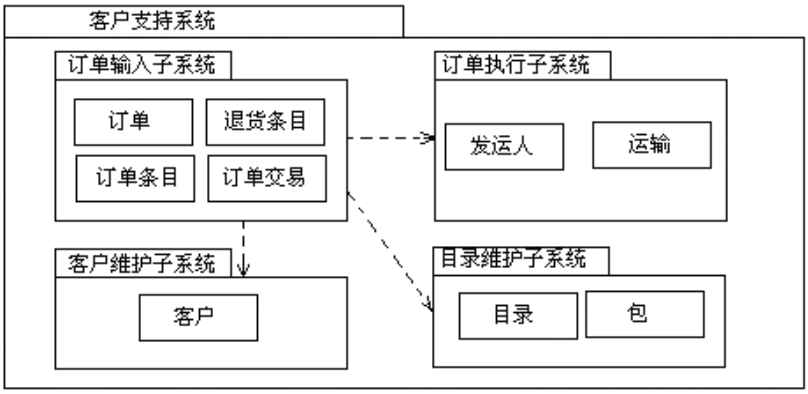


下面是一个销售系统的流程。



7.5 面向对象的体系结构设计

在面向对象的设计中，关注点变成了消息和响应机制。而我们由面向对象的分析转向面向对象的设计是一个自然的结果，在 OOA 中已经提供了足够多的信息，在高层设计阶段，我们可以用包图来建立体系体系结构。



在详细设计阶段，可以利用类图建立相应的体系结构。在设计的各个阶段，在必要的重点位置，我们还可以用顺序图或者协作图来描述一些最重要的消息机制。面向对象的设计不仅仅是根据功能性和非功能性需求建立一些相应的结构，更重要的是要分析一些潜在问题，通过种种设计技巧，提升系统的整体性能。高层体系结构设计很大的工作量在于划分子系统，子系统的划分主要有两个原则：

- 按功能区分子系统。
- 按结构区分子系统。

我们必须注意到，面向对象的设计与面向过程的设计从思想体系上是根本不同的，面向过程的设计着力点在于建立一个稳定的体系结构，整个构建的体系是自顶向下金字塔结构，所以它的体系更倾向于一种稳定性，但灵活性不足。而面向对象的设计，更倾向于建立一种灵活机变，便于修改便于升级的体系，设计中强调的是软件的重用和重构，设计着力点往往强调的是自底向上的服务，以及可升级和可维护性。同时也非常注意实时信息，或者说是消息和响应机制。这种设计的灵活性如果处理不好，也有可能破坏体系结构的稳定性造成体系结构的失败。

1）面向过程与面向对象的方法论区别

从方法论的角度，面向过程的设计更多的是属于重量级过程（heavy process），它的特点是：

- **细化的、长期的、详细的计划**：试图在相对长期的时间（例如项目的大部分时间）内详细的计划和预测活动和资源（人员）分配。
- **预测的而不是适应性的**：需要在前期定义所有需求，定义详细的设计，然后才开始计数实现。

而面向对象的设计更强调一种轻量级过程，或者是适应性过程（adaptive process），它的特点是：

- 认为变化是不可避免的驱动因素，鼓励灵活的改写。
- 整个设计是在迭代变更中完成的，整个体系结构应该适应这种变更。
- 鼓励程序人员参与设计。

所以，当我们使用面向对象的方法来设计的时候，应该把注意力放在这个地方。

面向过程和面向对象两种方法各有优缺点，事实上在设计中，稳定性和灵活性的平衡一直是我们最需要关注的问题，所以这两种方法应该有一个恰当的融合。

由于面向对象设计的成熟和发展，已经形成了一系列的重要设计原则和方法，这些原则和方法可以大大的提高我们的设计质量，设计的目标往往是建立一个大型的、分布式的、可升级、可维护而且是安全的体系，这也对设计者提出了更高的要求。

2）职责定义的原则

将系统职责分解并分配到系统元素一级，详细定义系统元素的职责是软件设计的核心任务。采用 OO 方法进行软件设计时类的职责定义应该根据以下原则来确定。

- 软件体系结构风格和设计模式所确定的结构已经蕴含了部分职责分解的方案；

- 用例描述规格中的后置条件定义了用例的操作契约，它是类职责定义的重要参考；
- 类的交互模型本身就蕴含了一种职责分解的方案，一个类响应和发出的消息列表可以成为定义类成员方法的直接来源和依据；
- 信息隐蔽是分配对象职责需要重要考虑的因素，即对象自己的数据应由对象自己来处理，别人不能直接干预和操作。

3）模型驱动的开发层次

从软件整个生命周期过程来看，软件开发的产品过程是一个模型驱动的过程，也是一个不断认识问题并描述问题，进而给出问题解决方案的演进过程。建模一般要由上至下分层进行，下层是上层问题的解决方案，它们之间存在内容上的过渡和追溯关系，往往成为软件确认与验证的依据软件开发的一般建模过程的层次和模型如下：

- **业务层**：对业务进行建模，描述业务过程、数据以及业务行为。可对应使用活动图、活动图以及行为模型进行刻画，而业务任务可通过业务级的 UseCase 模型进行刻画。
- **系统层**：定义软、硬件系统元素边界及其元素的职责分工与接口联系。重点对系统元素之间的协作关系与接口进行刻画，可通过系统级的交互与协作模型进行建模。
- **概要设计层**：对软件系统体系结构和软件组成结构进行建模。架构建模可使用组件模型进行描述，系统结构层面的建模包括：系统功能模型可选用系统级 UseCase 刻画，而系统静态逻辑结构可使用类/对象模型进行刻画，系统行为可选用交互模型进行刻画。
- **详细设计层**：从实现要求出发进行软件结构的设计与优化，对软件元素实现方案进行建模，使之成为软件编码的蓝图。通常在行为细致分析的基础上建立完整的设计类模型，完善类的内容设计。
- **实施层**：对软件物理组成构件及其关系进行建模，刻画软件物理配置结构（组件模型），并描述物理组件在网络拓扑结构中的部署结构（部署模型）。

7.6 高层设计中的体系结构分析

面向对象的设计并不是简单的把需求分析中的概念模型转换成设计模型就可以了，体系结构师必须在由需求分析获取体系结构因素，因此我们首先必须研究体系结构分析的问题。

体系结构分析的本质，是识别可能影响体系结构的因素，了解它的易变性和优先级，并解决这些问题。其难点是，应该了解提出了什么问题，权衡这些问题，并掌握解决影响体系结构重要因素的众多方法。体系结构分析是高优先级和大影响力的活动。

体系结构分析对如下的工作而言是有价值的：

- 降低遗漏系统设计核心部分的风险；
- 避免对低优先级的问题花费过多的精力；
- 为业务目标定位产品。

一、体系结构分析

体系结构分析是在功能性需求过程中，有关识别非功能性需求的活动。

1）体系结构分析需要解决的问题

下面说明在体系结构级别上，需要解决的诸多问题的一些示例：

- 可靠性和容错性需求是如何影响设计的？
- 购买的子组件的许可成本将如何影响收益？
- 分布式服务如何影响有关软件质量需求和功能需求的？

- 适应性和可配置性是如何影响设计的？

2）体系结构分析的一般步骤

体系结构分析有多种方法，大多数方法都是以下步骤的变体。

- 辨识和分析影响体系结构的非功能性需求。
- 对于那些具有重要影响的需求而言，分析可选方案，并做出处理这些影响的决定，这就是体系结构决策

二、识别和分析体系结构因素

1，体系结构因素

任何需求对一个系统体系结构都有重要影响。这些影响包括可靠性、时间表、技能和成本的约束。比如，在时间紧迫、技能有限同时资金充足的情况下，更好的办法是购买和外包，而不是内部开发所有的组件。然而，对体系结构最具影响的的因素，包含功能、可靠性、性能、支持性、实现和接口。通常是非功能性属性（如可靠性和性能）决定了某个体系结构的独到之处，而不是功能性需求。

2，质量场景

在体系结构因素分析期间定义质量需求的时候，推荐应用质量场景。

它定义了可量化（至少是可观测）的响应，并且因此可以验证。质量场景很少使用模糊的不具度量意义的描述，比如“系统要易于修改”。质量场景用<激发因素><可量化响应>的形式作简短的描述，如：

- 当销售额发送到远程计税服务器计算税金的时候，“大多数”时候必须 2 秒之内返回。这一结果是在“平均”负载条件下测量的。
- 当系统测试志愿者提交一个错误报告的时候，要在一个工作日内通过电话回复。

这里，“大多数”和“平均”需要软件体系结构师作进一步的调查和定义。质量场景直到做到真的可测试的时候，才是真正有效的。这就意味着需要有一个详细的说明。

3，体系结构因素的描述

体系结构分析的一个重要目标，是了解体系结构因素的影响、优先级和可变性（灵活性以及未来演变的直接需要）。因此，大多数方法都提倡对以下信息建立一个体系结构因素表。

因素	测量和质量场景	可变性（当前灵活性和未来演化）	因素（和其变化）对客户的影响，体系结构和其它因素	获取成功的优先级	困难或风险
可靠性 --- 可恢复性					
从远程服务失败中恢复。	当远程服务失败的时候，侦听到远程服务重新在线的一分钟内，重新与之建立联系，在产品环境下实现正常的存储装载。	当前灵活性—我们的 SME 认为直到重新建立连接前，本地客户简化的服务是可以接受的（也是可取的）。 演化—在 2 年之内，一些零售商可能选择支付本地完全复制远程服务的功能（如税金计算器）。可能性？高。	对大规模设计影响大。 零售商确实不愿意远程服务失败，因为这将限制或阻止它们使用 POS 进行销售。	高	低
……	……	……	……		
注：SME 表示主题专家。					

请注意上面的分类方法：可靠性—可恢复性。

4，从需求文档中获取体系结构因素

在体系结构设计中，中心功能需求库就是用例，它的构想和补充规范，都是创建因素表的重要源泉。在用例中，特殊需求、技术变化、未决问题应该被反复审核。其隐含或者清晰的体系结构因素要被统一整理到补充规范里面去。例如：

用例 1：Process Sale
主要成功场景 1．…… 特殊需求 ▪ 90%的信用授权应该在 30 秒内响应 ▪ 无论如何，当远程服务如库存系统失败的时候，我们需要强健的恢复措施。 ▪ …… 技术和数据变化表 2a. 商品的表识可以通过条形码扫描器或者是键盘输入。 …… 未决问题 ▪ 税法的变化是什么？ ▪ 研究远程服务的恢复问题。

三、体系结构因素的解析

体系结构设计的技巧就是根据权衡、相互依赖关系和优先级对体系结构因素的解决作出合适的选择。

但这还不全面，老练的体系结构师具有多种概念的知识（例如：体系结构样式和模式、技术、产品、缺陷和趋势），并且能把这些知识应用在它们的决定中。

1，记录体系结构的可选方案、决定和动机

不管目前体系结构决策的原则有多少，事实上所有的体系结构方法都推荐记录：

可选的体系结构方案；决定；影响因素；显著问题；决定动机。

这些记录按不同的的形式或者完善程度，被称之为：技术备忘录；问题卡；体系结构途径文档。技术备忘录的一个重要的的方面就是动机或者原理，当开发者或者体系结构师以后需要修改系统的时候，体系结构师可能已经忘了他当初的设计依据（一个资深体系结构师同时带多个项目的情况非常多见），备忘录对理解当时的设计背后的动机极为有用。解释放弃被选方案的理由十分重要，在将来产品进化的过程中，体系结构师也许需要重新考虑这些备选方案，至少知道当初有些什么备选方案，为什么选中了其中之一。技术备忘录的格式并不重要，关键是简单、清楚、表达信息完整。

技术备忘录
问题：可靠性---从远程服务故障中恢复
解决方案概要：通过使用查询服务实现位置透明，实现从远程到本地的故障恢复和本地服务的部分复制
体系结构因素 ▪ 从远程服务中可靠恢复 ▪ 从远程产品数据库的故障中可靠恢复 解决方案 在服务工厂创建一个适配器…… 动机 零售商不想停止零售活动…… 遗留问题 无

考虑过的备选方案 与远程服务厂商签订“黄金级”服务协议……

2，优先级

下面是指导做出体系结构决定目标：

- 不可改变的约束，包括安全和法律方面的事务；
- 业务目标；
- 其它全部目标。

早期要决定是否应该避免保证未来的设计，应该实事求是的考虑，那些将要推迟到未来的场景，有多少代码需要改变？工作量将是多少？仔细考虑潜在的变更将有助于揭示什么是首要考虑的重要问题。一个低耦合高内聚的产品，往往比较容易适应将来的变化，但也要仔细分析这样付出的代价，在这个问题上，设计师的掂量往往是决定这个项目的生命线。

3，系统不同方面的分离和影响的局部化

体系结构分析的另外一个基本原则，就是实现分离系统的不同方向。

系统不同方向的分离，是在体系结构级别上关于低耦合和高内聚的一种大尺度思考方法。虽然它们也应用在小尺度对象上，但这样的分离对于体系结构问题尤其突出。因为系统的不同方面很广泛，而且体系结构的解决方案涉及重要的设计选择。至少有三个实现系统不同方面分离的大尺度技术：

- 把系统的一个方面模块化到一个独立的组件（如子系统）中并且调用它的服务；
- 使用装饰器模式；
- 采用后编译器技术和面向方面的技术。

有了体系结构分析的结果，我们就可以讨论高层体系结构设计本身的一系列原则了。

7.7 高层体系结构设计中的层模式

一、层模式

1，解决方案：

层模式（Layers pattern）的基本思想很简单。

- 根据分离系统的多个具有清晰、内聚职责的设计原则，把系统大尺度的逻辑结构组织到不同的层中，每一层都具有独立和相关的职责，使得较低的层为低级和通用的服务，较高的层更多的为特定应用。
- 从较高的层到较低的层进行协作和耦合，避免从底层到高层的耦合。

层是一个大尺度的元素，通常由一些包或者子系统组装而成。

层模式与逻辑体系结构相关，也就是说，它描述了设计元素概念上的组织，但不是它物理上的包或者部署。层为逻辑体系结构定义了一个 N 层模型，称之为分层体系结构（Layers Architecture），它作为模式得到了极为广泛的应用和引述。

作为层体系结构的模式，我们必须提到框架（framework）的概念：

一般来说框架具有如下的特征：

- 是内聚的类和接口的集合，它们协作提供了一个逻辑子系统的核心和不变部分的服务。
- 包含了某些特定的抽象类，它们定义了需要遵循的接口。
- 通常需要用户定义已经存在的框架类的子类，是客户得以扩展框架的服务。
- 所包含的抽象类可能同时包含抽象方法和实例方法。

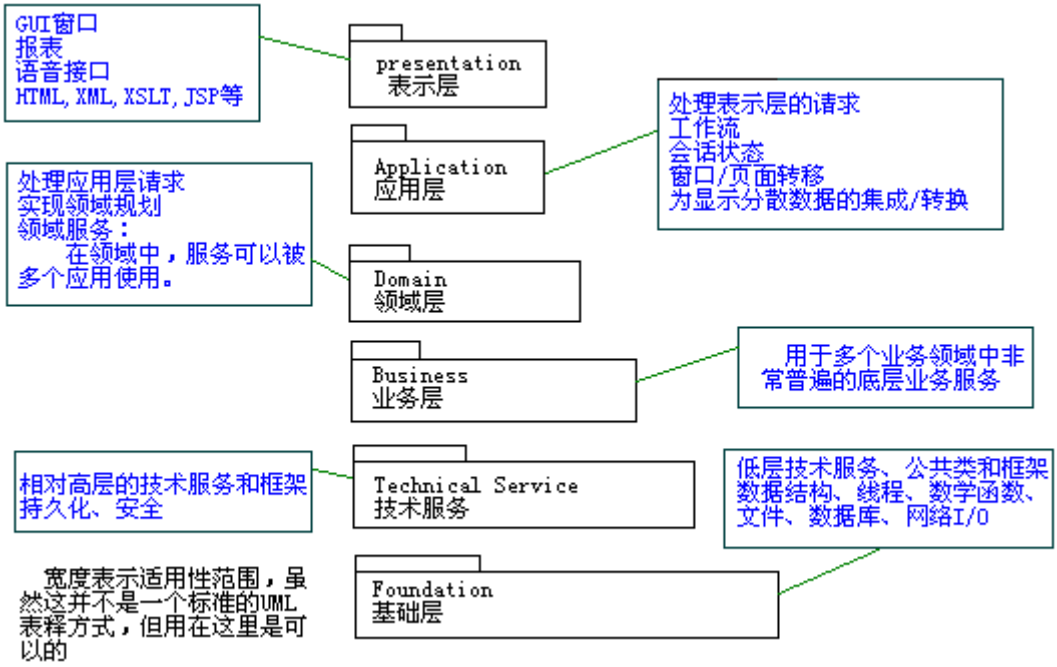
- 遵循好**莱坞原则**:“别找我们，我们会找你。”就是用户定义得类将从预定义的类中接受消息，这通常通过实现超类的抽象方法来实现。

2，问题：

- 由于系统的许多部分高度的耦合，因此源代码的变化将波及整个系统。
- 由于应用逻辑与用户接口捆绑在一起，因此这些应用逻辑在其它不同的接口上无法重用，也无法分布到另一个处理节点上。
- 由于潜在的通用技术服务或业务逻辑，与更具体的应用逻辑捆绑在一起，因此这些通用技术服务或者业务逻辑无法被重用，或者分布到其它的节点，或者被不同的实现简单的替换。
- 由于系统的不同部分高度的耦合，因此难以对不同开发者清晰界定格子的工作界限。
- 由于高度耦合混合了系统的各个方面，因此改进应用程序的功能，扩展系统，以及使用新技术进行升级往往是艰苦和代价高昂的。

3，示例：

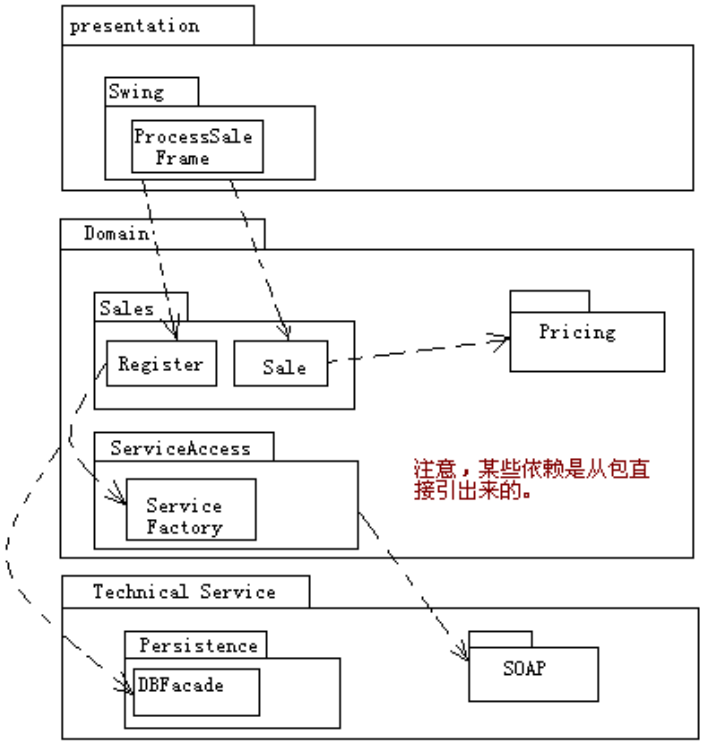
信息系统一般分层逻辑体系结构。



在具体体系结构设计的时候，可以建立比较详细的包图，但是，并不需要面面俱到。体系结构视图的核心，是展示少数值得注意的元素。

4，层与层之间和包与包之间的耦合

逻辑体系结构还可以包含更多的信息，用来描述层与层以及包与包之间值得注意的耦合关系。在这里，可以用依赖关系表达耦合，但并不是确切的依赖关系，而仅仅是强调一般的依赖关系。



有时候也可以不画出类，专注于包之间的依赖关系。

5，协作：

在体系结构层面上，有两个设计上的决策：

- **什么是系统的重要部分？**
- **它们是如何连接的？**

在体系结构上，层模式对定义系统的重要部分给出了指导。象外观、控制器、观察者这些模式，通常用于设计层与层、包与包之间的连接。

大部分的多层体系结构不能像基于 OSI 7 层模型的网络协议一样，上一层只能调用下一层。相反，在信息系统中的分层通常是“松散的分层”或者说是“透明的分层”。一个层的元素可以和多个其它层的元素协作或耦合。对于一般层之间耦合的观点是：

- 所有较高的层都可以依赖于技术服务层和基础层

  比如在 Java 中，所有的层都依赖于 java.util 包元素。
- 依赖于业务基础设施层的领域层是要首先考虑的。
- 表示层发出对应用层的调用，应用层再对领域层进行服务调用。除非不存在应用层，表示层一般是不直接对领域层进行调用的。
- 如果应用是一个单进程的“桌面”程序，那么领域层的软件对象对于表示层、应用层和更底层（如技术服务层）可以直接可见或者在中间传递。
- 另一方面，对于一个分布式系统，那么领域层对象的可序列化复制对象（通常称之为值对象或者数据容纳对象）通常可以被传递到表示层。在这种情况下，领域层被部署到另一台服务器上，客户端节点得到服务器数据的拷贝。

现在的问题是，与技术服务层和基础层的关系不是很危险吗？事实上耦合本身并不是个问题，但是与变化点和演化点的耦合是不稳定的而且是难于修正的。几乎没有任何理由，去抽象和隐蔽某些不太可能变化的因素，即使这些因素可能变化，这种变化所产生的影响也是微乎其微的。

6，层模式的优点：

- 层模式可以分离系统不同方面的考虑，这样就减少了系统的耦合和依赖，提高了内聚性，增加了潜在的重用性，并且增加了系统设计上的清晰度。
- 封装和分解了相关的复杂性。
- 一些层的实现可以被新的实现替代，一般来说，技术服务层或者基础层这些比较低层的层不能替换，而表示层、应用层和领域层可能进行替换。
- 较低的层包含了可重用的功能。
- 一些层可能是分布式的（主要是领域层和技术服务层）。
- 由于逻辑上划分比较清楚，有助于多个小组开发。

二、模型-视图分离原则

这个原则我们已经讨论了多次，但这里还是有必要总结一下。

非窗口类如何与窗口类通信？推荐的做法，是其它组件不和窗口对象直接耦合。因为窗口和特定的应用有关，耦合太强不利于重用。

这就是模型--视图分离的原则。

模型—视图分离（Model – View Separation）的原则，已经发展为模型-视图-控制器（Model-View-Controller MVC）模式的一个关键原则。MVC 起源于一个小规模的 Web 体系结构（比如 Struts），近来，这个术语（MVC）被分布式设计团体采纳，也应用在大规模的体系结构上，模型指领域层，视图指表示层，控制器指应用层的工作流对象。

模型—视图分离原则的动机包括：

- 为关注领域处理而不是用户界面而定义内聚的模型。
- 允许分离模型和用户界面层的开发。
- 最小化界面因为需求变更给领域层带来的影响。
- 允许新的视图方便的连接到领域层而不影响领域层。
- 允许在同一模型对象上有多个联立的视图。
- 允许模型层的运行独立于用户界面层。
- 允许方便的把模型层简单的连接到另一个用户界面框架上。

7.8 体系结构设计案例

一、系统总体概念性架构设计

对于我们在数据模型一张已经讨论过的具有合作方协同设计能力的 PDM 系统，我们提出了如下基本设计要求：

1）以互联网为整个设计的基础设施，但要加强安全机制的设计。

2）该系统是以“机械设计研究所”为中心建立 PDM 系统扩展功能，应该与原有的 PDM 系统互联。

3）其它合作方，设置单独的 Web Service 组件（如果原来为 C/S 架构，则设置单独的 Web 服务器），保证对数据和功能实现远程调用。

4）合作方相互之间也可以在必要的情况下通过 Web Service 相互沟通。

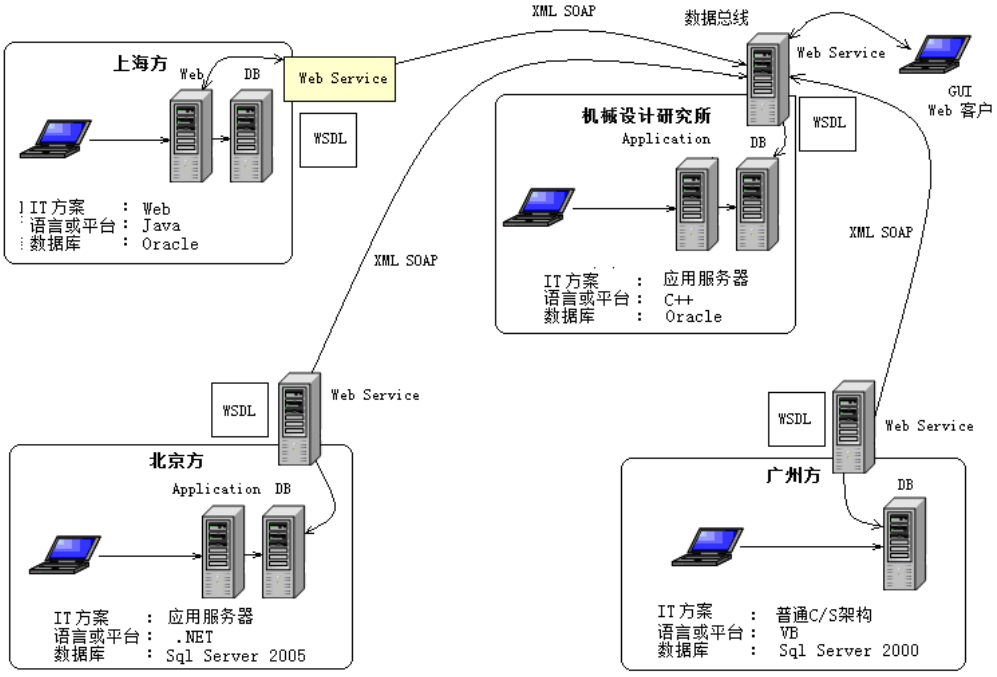
5）各合作方的 Web Service 组件只处理与合作项目相关的数据和文件，其它与此不相关的信息则完全绝缘，确保合作方商业秘密的安全性。

6）该系统是模块化的，应该很容易组合新的服务和业务模式。

7）所有传输数据协议一律采用 XML，确保跨平台跨语言应用，不需要改变合作方内部的任何工作方式。

8）我们有理由认为采用 XML 格式可能会有效率问题，但是该系统大量的工作是用于设计过程控制与管理，本身的设计过程都使用基于内部的处于封闭状态的 PDM 系统，因此大部分相关数据的传输速率应该在可接受范围内的。至于少量大型工艺文件，考虑到合作方协同设计主要是在里程碑点上的传输，而不是日常的频繁传输，所以速率问题应该不大。

本项目的基本构思是在多个分散点合理的使用 Web Service 技术，并且通过中央基于 Web 的 PDM 系统进行了数据集成（这也可以认为是 Web Services 最后那个 s 的具体体现），这就有可能在合作项目的范围内实现基于服务的数据总线，这将大大提高整个体系的效能，有利于克服异地合作设计的困难。我们设计的总体结构如下图所示，这称之为总体概念性架构设计。



在这个图中，合作方本身的工作模型并没有改变，合作方的 Web Service 服务器只是为着建立数据总线的通讯，与合作有关的数据将直接保留在本地数据库相应的区中，这个数据区将直接与 Web Service 服务器有关应用程序相连。其他的数据将和这个服务器绝缘，以保护本地数据的安全。

与合作有关的业务，可以通过专门设计的 GUI Web 客户端程序，应用设置在“机械设计研究所”的总 Web 服务器功能，而不需要在每个服务方设置过份复杂的应用程序。应用 GUI Web 客户端程序主要因为某些应用可能比较复杂（比如工艺图纸显示，复杂业务流程的定义与跟踪等），但是对于某些简单数据的人机交互，也可以直接使用浏览器，因此设计中需要把这两种应用模型分开。

二、PDM 系统主要需求

PDM（产品数据管理系统 Product Data Management）是用来管理所有与产品相关信息（包括零件信息、配置、文档、CAD 文件、结构、权限信息等）和所有与产品相关过程（包括过程定义和管理）的技术。本项目在这个基础上，要求利用了 Web Services 技术进一步实现分布式异构环境下合作方协同设计，因此，对项目提出了新的要求。根据需要，本产品主要需求如下。

1，实现协同模式下的产品项目管理与过程管理

本项目需要构建专门的协同设计管理模块，其基本功能需求如下：

R1-1）产品应该具备项目的创建、修改、查询、审批、统计等能力。

R1-2）产品应该提供项目人员和组织机构的定义和修改，并能够对合作方进行适当的监控。

R1-3）产品应该在对项目人员和组织机构有效管理的基础上，实现对各类人员角色的指派。

R1-4）产品应该在人员角色确定后，规定其对产品数据操作权限。

R1-5）产品应该具备协同项目开发过程定义的能力。

R1-6）产品应该保证用户能够自定义过程单元，并且能够把这些单元连结成适当的工作流，能定义工作流每个单元完成后需要提交的设计对象（部件、零件、文档等）。

R1-7）产品应该具备过程管理的手段，能够建立任务列表，并记录每个列表的执行信息。工作流程管理的主要功能包括:

R1-8）产品应该具备协同项目开发任务定义与过程监控能力。

R1-9）产品应该有效交互信息，能够根据工作进展情况，向有关人员提供相关信息和解决方案。

2，实现工程图档及设计文档的有效管理与检索

R2-1）产品应该在数据库中建立合理的工程图档管理数据结构。

R2-2）产品应该构件有效的工程图档管理功能。

R2-3）产品应该根据用户定义的信息项完成图档基本信息的录入与编辑。

R2-4）产品应该建立图档基本信息与图档文件的清晰的连接关系。

R2-5）产品应该实现图档文件的批量入库和交互入库方式。

R2-6）产品对于指定的图档文件从数据库中释放，并传送到客户端进行操作，应该支持 Check-in/Check-out 功能，以保证文件的完整性和一致性。

R2-7）产品显示模块应该可以浏览和显示多种常见格式的文件。

R2-8）产品应该为用户提供快速、方便的批注功能，支持使用各种用于批注的实体（复线、指引文字和云状线等）。批注文件可存放在独立的文件中,充分保护原始文件。

3，实现产品设计与图档的配置管理与变更管理

R3-1）产品应该建立产品结构树，该树的节点与文档对象应该有清晰的可视化关系。

R3-2）产品应该对设计文档与图档的版本演化有管理能力与可视化表达能力。

R3-3）产品应该针对设计中的不同批次或同一批次的不同阶段（如设计、工艺、制造与组装等）,生成的产品结构信息，生成不同的视图。

R3-4）产品应该能够查询与浏览品零部件之间的层次关系，并用图视方式显示产品各种配置信息的变化，包括结构的改变、各种版本的演化。

R3-5）配置管理与变更管理应该能够对产品的各版本数据提供冻结、释放、复制等操作。

R3-6）产品对文档或图纸进行编码规则应该符合企业编码规则，这个规则在系统中应该是可以订制的。

4，产品的非功能需求

R4-1）产品的安全性应该保证权限控制的有效和安全。

R4-2）敏感数据在网络上传输的时候，应该保证即使发生截获也不应该被解读。

R4-3）产品的易使用性要好，尽可能不要改变各个合作单位原来的工作方式

R4-4）产品对于不属于合作伙伴协作设计的内容，不应该在系统中出现，以确保各个企业的

商业秘密的安全。

三、系统总体体系结构设计

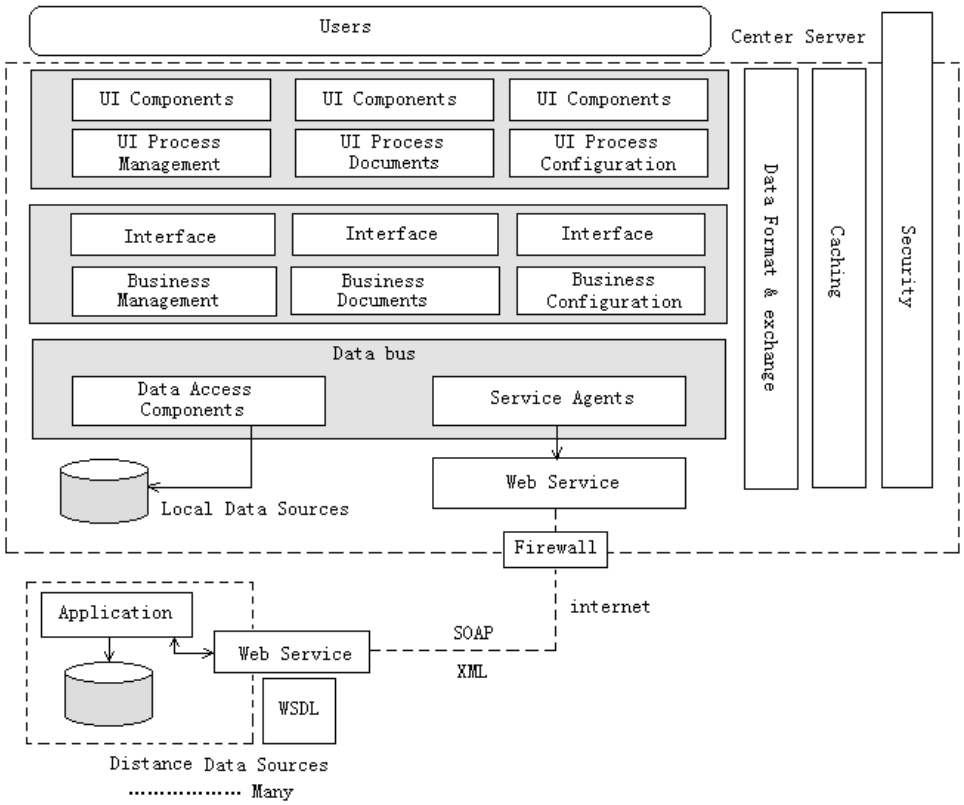
为了使文件管理发挥更大的作用，并且减少开发上的难度，本项目在设计的时候，共分成三个子系统，每个子系统分配给一个独立的开发组完成，它们包括：

项目管理与过程管理子系统（Project Management and Process Management，PM&PM）。

工程图档与文档管理子系统（Engineering Drawing and Document Management，Ed&DM）。

配置管理与变更管理子系统(Configuration Management and Change Management，CM&CM）。

各子系统要求设计成具有独立系统架构的完整系统，各子系统之间不得交互，他们只能通过共享的数据总线（Data Bus）进行交互，这样就减少了各子系统之间的偶合性，增加了子系统的内聚度，减少了开发、集成、调试、维护以及后期升级的难度。系统的整体结构关系如下图所示。



系统的数据总线通过 Web Service 技术，隔离了远程异构数据的物理位置、数据格式等信息，把本地数据和远程数据结合起来，使用者与开发者并不需要知道这些远程异构数据源的具体情况，这就大大减少了开发和应用上的难度。系统除了提供共享的数据总线以外，还提供了公用的数据格式与交换、缓存和安全机制，提高了模块的可复用性

系统在设计中还注意到，系统采用垂直分层，水平分模块，力争结构非常清晰。垂直方向基本按照表示层、业务层和持久化三个层次划分，使关注点分离功能分割清晰，而且通过接口分解了模块之间的耦合性，便于系统维护。

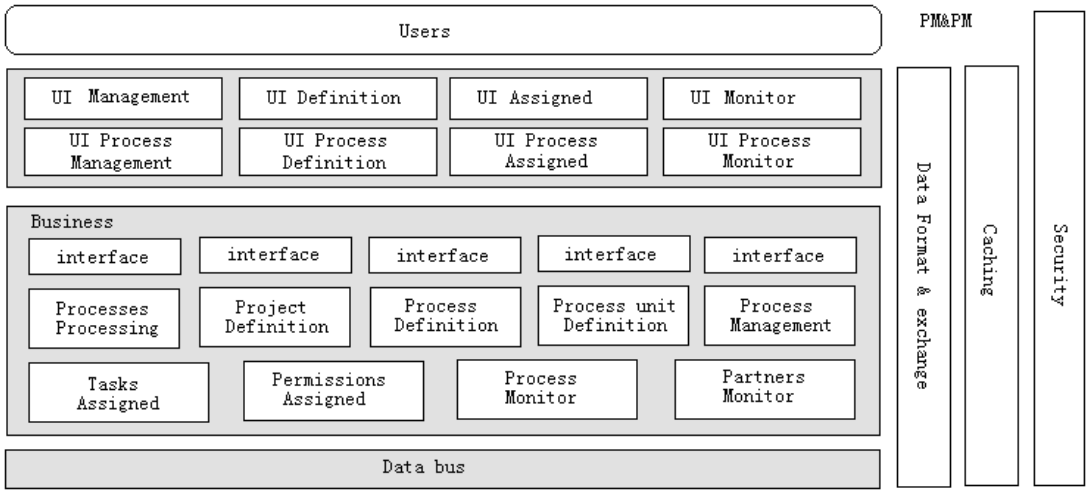
具体地说：在表示层，按垂直方向分离了用户接口组件和用户接口过程组件；在业务层，按照统一的接口对外，水平分离了业务流程、业务组件和业务实体；在持久化层，水平分离了数据访问组件以及服务代理，实现了统一的数据总线机制，这样一来，就使这个体系结构非常清晰。

四、子系统体系结构设计

本系统要求子系统是具有独立体系结构的系统，各子系统之间通过数据总线进行交互，这种规定确保了设计与开发的独立性，下面我们简要介绍各个子系统的高层体系结构。

1，项目管理与过程管理子系统（PM&PM）

本子系统是这个项目最具有特征的部分，由于合作方协同设计的要求，本子系统需要对本设计项目与合作方进行统一的无缝的项目管理与监控，所以具备一般 PDM 系统所不具备的功能。子系统的下层是分为模块，也就是独立的业务单元，项目设计规则要求，各模块是独立设计的，模块之间不能直接交互，而只能通过接口用规则的方法交互，项目的这个要求，就确保了模块的高内聚与低偶合，确保了后期的升级和维护成本比较低。该子系统的顶层架构如下图所示。



各个模块的功能简单描述如下。

业务层:

- Processes Processing：业务流程处理主模块，包括创建、修改、查询、审批、统计流程
- Project Definition：项目组织定义业务模块。
- Process Definition：协同项目开发过程定义业务模块。
- Process unit Definition：过程单元定义业务模块。
- Process Management：过程管理业务模块。
- Tasks Assigned：人员任务指派业务模块。
- Process Monitor：过程监控业务模块。
- Permissions Assigned：操作权限指派业务模块。
- Partners to Monitor：合作方监控业务模块。

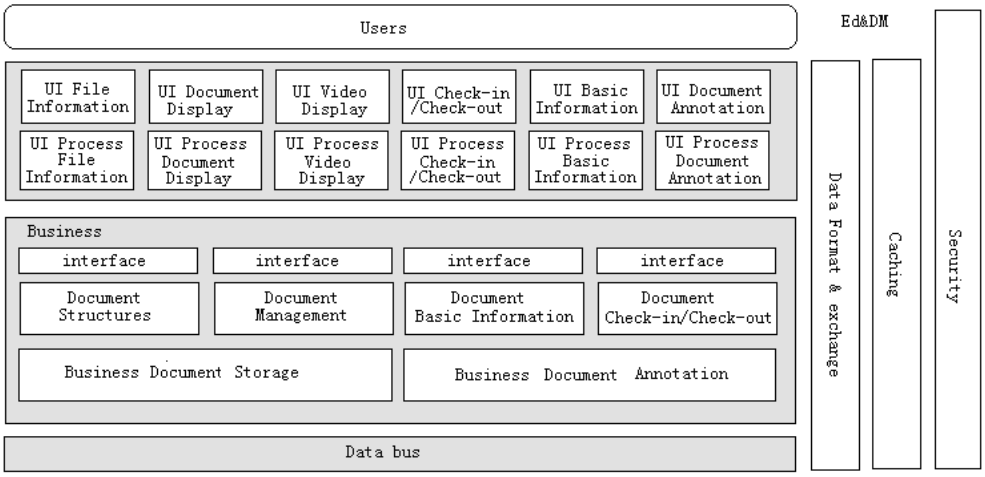
表示层：

- UI Management：管理交互模块。
- UI Definition：定义交互模块。
- UI Assigned：指派交互模块。
- UI Monitor：监控交互模块。

2，工程图档与文档管理子系统（Ed&DM）

工程图档与文档管理子系统是这个项目最重要的部分，它也是 PDM 系统的核心功能，主要用于检索、修改、变更工艺过程中所需要的各类设计文档与图形文档，其中可能还包括三维演示

视频文档，本子系统的顶层架构如下图所示。



各个模块的功能简单描述如下。

业务层：

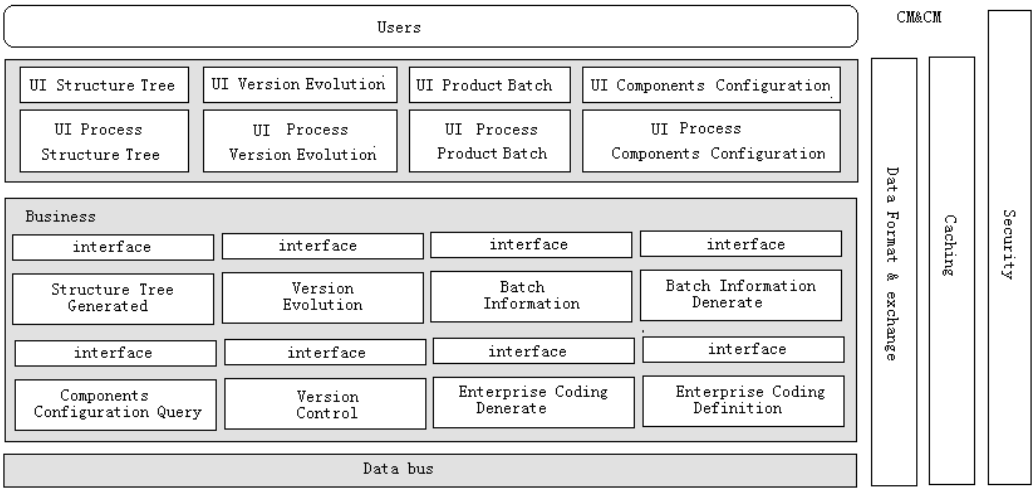
- Document Structures：工程图档结构建立业务模块。
- Document Management：工程图档管理业务模块。
- Basic Information：工程图档基本信息录入业务模块。
- Check-in/Check-out：工程图档检入/检出业务模块。
- Document Storage：工程图档入库业务模块。
- Document Annotation：工程图档批注业务模块。

表示层：

- UI File Information：工程图档信息显示交互模块。
- UI Image Display：图形显示交互模块。
- UI Video Display：工程视频显示交互模块。
- UI Check-in/Check-out：工程图档检入/检出交互模块。
- UI Basic Information：工程图档基本信息录入交互模块。
- UI Document Annotation：工程图档批注交互模块。

3，配置管理与变更管理子系统（CM&CM）

本子系统是这个项目开发中最困难也是最具挑战的部分，它主要用于处理整个工程文档的演化与版本控制，其中包括可视化的版本跟踪，企业编码的生成与应用，批次文档的查询与组合，以及产品零部件的配置等重要信息处理。可以说，PDM 文件处理，只有和配置管理系统结合起来，才可能发挥更大的作用。而变更管理是新产品设计与发布的重要一环，PDM 文件系统必须对变更管理提供有效的支持。本子系统的顶层架构如下图所示。



各个模块的功能简单描述如下。

业务层：

- Structure Tree Generated：结构树生成业务模块。
- Version Evolution ：版本演化生成业务模块。
- Batch Information ：产品批次文档信息查询业务模块。
- Batch Information Denerate：产品批次信息生成业务模块。
- Components Configuration Query ：产品零部件配置查询业务模块。
- Version Control：版本数据控制业务模块。
- Enterprise Coding Denerate： 企业编码生成业务模块。
- Enterprise Coding Definition ：企业编码定义业务模块。

表示层：

- UI Structure Tree：产品结构树显示与交互模块。
- UI Version Evolution： 产品版本演化显示与交互模块。
- UI Product Batch ：产品批次文档信息显示与交互模块。
- UI Components Configuration： 产品零部件配置显示与交互模块。

由于 Web Service 技术本身比较成熟，所以论文中就不再涉及诸如 Web Service 原理和应用方面的问题了。

五、系统设计原则

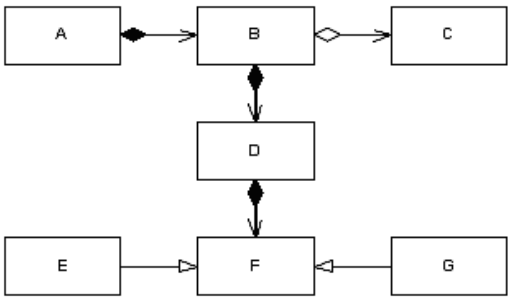
UML 只是一个工具。如果使用得法，UML 能够帮助我们轻松地构造出更好的系统。然而，要设计出优秀的系统，关键仍在于要有一个好的设计原则或理念。

“提高类的内聚力，减少不同类之间的联系”这一点在谈到好的面向对象设计原则时经常被反复引用。

一个内聚的类包含那些在目标和作用域上都紧密相关的行为和信息。它意味着你不应该把构造 UI 的代码和实现数学算法的代码混合到一起，你应该尽力把所有与用户紧密相关的信息封装到 UserAccount 类。

内聚式设计是一个重要的设计原则，原因有很多：它有助于减少类之间的依赖关系，使得设计更直观、更容易理解，方便了向其他开发者介绍整个设计，减少了开发者同一时刻需要操作的类的数量，等等。例如，如果你要改变网站的用户身份验证机制，只修改单个文件中的一个类无疑要比修改多个文件、多个类更加方便。

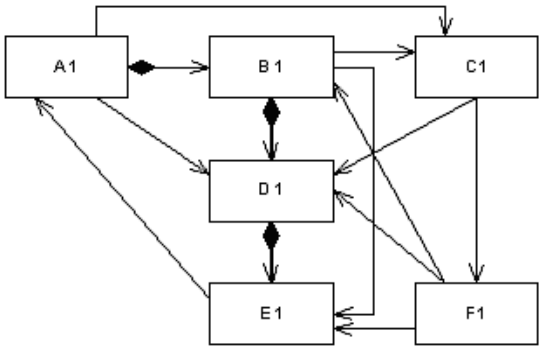
“减少不同类之间的联系”意味着使类或者文件之间的交互减到最少。它不仅使得整个设计容易理解，而且也方便了代码的维护。请考虑下面这个例子：



设计实例 A

除非深入了解了上述各个类的用途，要估计这些类的内聚程度是不可能的。然而，从这些类之间的关系可以看出，这个设计方案已经成功地减少了不同类之间的联系。类之间的交互被减到了最少，从而使得系统的行为很容易理解。

更重要的是，修改任意一个类时受影响的类数量都减到了最少（例如，修改 D 类只直接影响 B 类）。另外，要访问 D 类中的功能，我们无需知道任何有关 E、F 或 G 类的情况。作为比较，请考虑下图：



设计实例 B

显然，在这个设计实例中，类之间的联系是相当紧密的。一旦对 D1 类作了修改，为了检查这种修改对其他类的影响，我们必须对其他类进行广泛的测试。

只有在实践中不断锻炼才能避免出现过于复杂的设计，但注意以下几点有助于达到这一目标：

1．提高类的内聚力。不要把密切相关的功能分散到多个文件和类之中。

2．采用直观、有意义的名字。如果其他人不能了解类、函数或者变量的作用，不管类的结构是多么完美，整个设计仍缺乏直观性。过多地采用缩写词会影响设计的可理解性。

3．不要害怕改写代码。有些时候，在几个类之间移动一些函数能够大大地简化代码。

4．类应该保持紧凑、简洁。代码膨胀是类缺乏内聚力的一种征兆。过于庞大的类、模块或者文件往往缺乏明确的用途和目标。

5．让其他人复查你的设计。其他人可能有新的想法，或者为你指出你以为显而易见但别人却不能明白的问题。

6．在早期设计阶段不要考虑太多的性能问题。与一个笨拙的、为了昨天所出现的问题而优化的设计相比，一个简洁、经过精心调整的设计更容易进行性能优化。注意这并不是建议把性能问题抛到脑后，而是建议把细节优化问题留到工程后期考虑。

7.9 分析与设计的可追朔性

一、需求跟踪

我发现很多初涉开发的人都会有这样的一个尴尬的开发经历，其实是忽略了一个需求，所以在我自认为完成编程后，不得不返工编写额外的代码。就是因为忘了，后来不得不返工。如果忽略某几个需求造成用户不满意或发布一个不符合要求的产品，那就不仅仅是尴尬了。在某种程度上，需求跟踪提供了一个表明与合同或说明一致的方法。更进一步，需求跟踪可以改善产品质量，降低维护成本，而且很容易实现重用。

需求跟踪是个要求手工操作且劳动强度很大的任务，要求组织提供支持。随着系统开发的进行和维护的执行，要保持关联链信息与实际一致。跟踪能力信息一旦过时，可能再也不会重建它了。由于这些原因，应该正确使用需求跟踪能力。下面是在项目中使用需求跟踪能力的一些好处：

- 审核跟踪能力信息可以帮助审核确保所有需求被应用。
- 变更影响分析跟踪能力信息在增、删、改需求时可以确保不忽略每个受到影响的系统元素。
- 维护可靠的跟踪能力信息使得维护时能正确、完整地实施变更，从而提高生产率。要是一下子不能为整个系统建立跟踪能力信息，一次可以只建立一部分，再逐渐增加。从系统的一部分着手建立，先列表需求，然后记录跟踪能力链，再逐渐拓展。
- 项目跟踪在开发中，认真记录跟踪能力数据，就可以获得计划功能当前实现状态的记录。还未出现的联系链意味着没有相应的产品部件。
- 再设计（重新建造） 你可以列出传统系统中将要替换的功能，记录它们在新系统的需求和软件组件中的位置。通过定义跟踪能力信息链提供一种方法收集从一个现成系统的反向工程中所学到的方法。
- 重复利用跟踪信息可以帮助你在新系统中对相同的功能利用旧系统相关资源。例如：功能设计、相关需求、代码、测试等。
- 减小风险使部件互连关系文档化可减少由于一名关键成员离开项目带来的风险。
- 测试测试模块、需求、代码段之间的联系链可以在测试出错时指出最可能有问题的代码段。

以上所述许多是长期利益，减少了整个产品生存期费用，但同时要注意到由于积累和管理跟踪能力信息增加了开发成本。这个问题应该这样来看，把增加的费用当作一项投资，这笔投资可以使你发布令人满意同时更容易维护的产品。尽管很难计算，但这笔投资在每一次修改、扩展或代替产品时都会有所体现。如果在开发工程中收集信息，定义跟踪能力联系链一点也不难，但要在整个系统完成后再实施代价确实很大。

二、需求跟踪能力矩阵

表示需求和别的系统元素之间的联系链的最普遍方式是使用需求跟踪能力矩阵。下表展示了这种矩阵，这是一个“化学制品跟踪系统”实例的跟踪能力矩阵的一部分。

这个表说明了每个功能性需求向后连接一个特定的用例，向前连接一个或多个设计、代码和测试元素。设计元素可以是模型中的对象，例如数据流图、关系数据模型中的表单、或对象类。代码参考可以是类中的方法，源代码文件名、过程或函数。加上更多的列项就可以拓展到与其它工作产品的关联，例如在线帮助文档。包括越多的细节就越花时间，但同时很容易得到相关联的软件元素，在做变更影响分析和维护时就可以节省时间。

一种需求跟踪能力矩阵				
用例	功能需求量	设计元素	代码	测试实例
UC-28	catalog.query.sort	class	catalog.sort()	search.7
		catalog		search.8
UC-29	catalog.query..import	class	catalog..import()	search.8
		catalog	catalog.validate()	search.13
				search.14

跟踪能力联系链可以定义各种系统元素类型间的一对一，一对多，多对多关系。

上表允许在一个表单元中填入几个元素来实现这些特征。这里是一些可能的分类：

- 一对一：一个代码模块应用一个设计元素。
- 一对多：多个测试实例验证一个功能需求。
- 多对多：每个用例导致多个功能性需求，而一些功能性需求常拥有几个用例。

手工创建需求跟踪能力矩阵是一个应该养成的习惯，即使对小项目也很有效。一旦确立用例基准，就准备在矩阵中添加每个用例演化成的功能性需求。随着软件设计、构造、测试开发的进展不断更新矩阵。例如，在实现某一功能需求后，你可以更新它在矩阵中的设计和代码单元，将需求状态设置为“已完成”。

需求跟踪矩阵记录了不同阶段工作产品之间内容的追溯关系，对于系统有效性验证、需求变更控制、测试用例设计都需要参考需求跟踪矩阵，但是对于最初的项目计划，由于需求跟踪并没有建立起来，所以不需要参考它。

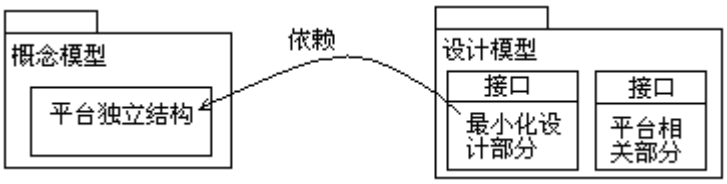
三、从概念模型到设计模型

设计模型是对概念模型的精练，它涉及了可执行平台的特殊细节，因此设计模型必须对许多重要的细节进行规划，它必须满足多种实现语言和技术的需求。它需要组织跨越多个处理节点间的系统部署，由于设计模型比概念模型更加复杂，因此，把概念模型与设计模型分开是明智之举，一般用概念模型来定义高层结构，用设计模型来细化结构、合并细节。

由于设计模型是对概念模型的细化，因此希望概念模型中的结构继续在设计模型中得以保持，这也意味着在设计元素结构中的包与分析元素结构中的包相对应，同时也可以发现设计模型中的类也是派生于概念模型中的类。同样，在分析阶段标识的用例切片也会在设计阶段细化。

事实上，概念模型中的结构都是与平台无关的，但是设计模型的结构可以分成两个部分：

- 最小化设计：对应于与平台特性无关的概念模型部分，是系统能力的基本表达。它包含相应的边界类、控制类和实体类。
- 平台相关部分：与平台技术有关的部分。



这种清晰的分离，对于摆脱平台的限制极其重要，它也将改进系统的可移植性。另外，在设计模型中还会包括在边界类、控制类和实体类之间传递消息或者数据的类，以及负责处理异常的类等，因此，设计模型会比概念模型有更多的设计元素。

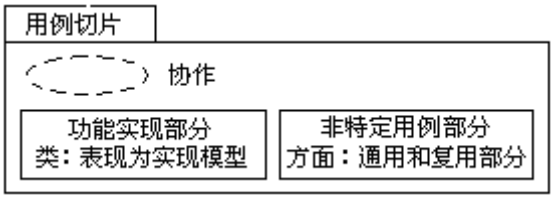
上面提到的边界类、控制类和实体类，或者它们的组合，将演变为设计中的构件，这些构件将拥有来自于概念类职责的接口。我们称这些接口为**最小化设计接口**。

而对于平台相关部分，将拥有**平台相关接口**，比如 web 的 HTTP 接口，或者 Web Service 的远程接口等。

四、用例模型横切于模型

当我们实现一个用例的时候，必须表示出所需要的类，以及这些类的特性（属性、方法和关系），如果我们引进一个“**用例切片**”的概念，这个切片把**功能实现部分**放在一个模型中（设计模型），而**通用的和复用**的部分则保存在一种**非特定用例部分**中，把所有这些切片的叠加将构成系统的设计模型。这种方式，将会使问题比较清晰，避免不必要的混乱和遗漏。我们针对上面的用例切片的概念，可以进一步细化和规范。一个用例切片应该包含三个方面的内容：

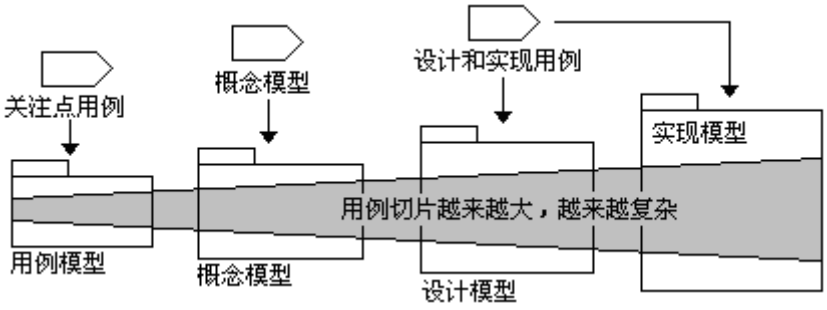
- **协作**：我们知道，用例是从系统的外部来描述系统的，但是协作是从内部视角来描述系统，在描述上可以表达类与类之间具体的协作关系。同时还必须对它命名，一般来说，协作的名字需要和用例一致，以防止理解上的歧义。
- **类**：这些类是特定于该用例实现，是本质上与其它用例分离的部分。
- **方面**：是该用例实现的时候对原有类的扩展，也是可能发生缠绕的部分。



用例切片可以层次化的组成设计模型，这个概念最好的比喻就是投影仪，切片类似于叠加在一起的幻灯片，设计模型类似于屏幕，我们可以独立的制作每个幻灯片，但需要一些协调工作，以保证它们内容的一致性，每个幻灯片的内容可以不同，但显示的位置应该在规定的地方。软件开发是围绕着模型的构建进行的，大概的步骤如下：

- 首先通过用例模型捕获涉众关注点。
- 然后把用例模型提炼为概念模型，这就是从高层视角对系统的描述。
- 通过设计模型来决定系统会运行于什么平台。
- 实现模型就是系统的代码实现。

设计系统应该逐个用例的进行，通过日益增多的模型来获取用例，提炼并且实现它。当完成一个用例的工作以后，就可以在一个包中（称之为用例模块）交付与这个用例相关的所有工件（包括分析、设计、实现和测试文档）。一个用例模块由每种模型的用例切片组成。用例模块可以单独开发，然后再合并成完整的系统。也就是说当构建一个系统的时候，需要逐个用例的进行构建，首先识别系统用例，然后一次一个地处理用例，详细描述它、分析它、设计它、并且实现它，当你在不同的模型中构建每个用例的时候，也会在不同的模型中更新相应的用例切片，这些用例切片如下图中阴影所示。



例如当工作于一个用例切片的时候（例如设计阶段用例切片），应该从这个用例切片的上游（对应于分析阶段用例切片）开始做一些精化，添加一些内容。因此每个下游用例片都比上游切片更大也更复杂。模型也同样的是下游模型比上游模型更大而且更复杂，这是因为模型也需要考

虑越来越多的问题。

1，保持用例模型中的结构

通过各种不同的模型逐渐对用例切片进行精化，以完成对系统的开发，我们主要是通过这些模型的用例切片，使所有的下游模型都保持用例模型中的结构，这样做的理由如下：

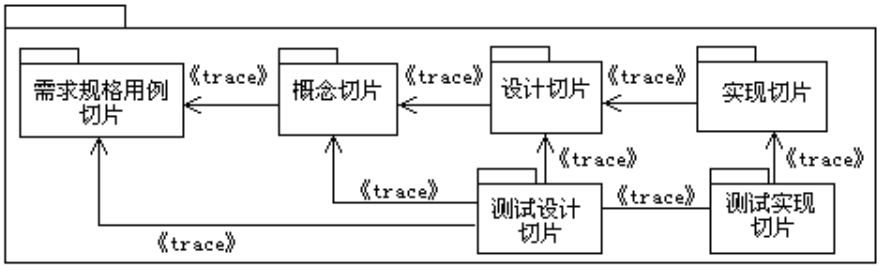
- **帮助我们理解下游模型**：可以通过查看相对应的用例，识别每个用例的切片到底完成什么功能。
- **便于在前后模型中转换**：软件开发并不是从用例到实现的线性过程，而是在前后不同的模型中转换和更新，保持模型中的结构，有助于在受控环境下无缝的转换模型，先完成一部分用例描述，接着转入分析来研究交互，然后回过头来完善用例，以澄清原来缺少的涉众关注点细节。也许在设计的时候，还可能回到需求来澄清该用例描述的特定需求，在建立系统架构基线的时候，也可能会前后切换。
- **保持模型一致**：如果下游模型与上游模型不同，您就可能需要花时间去理解完全不同的结构，并且维护这种模型之间的映射关系，由于并没有一种形式化的方法解决这个问题，因此难以在模型中保持一致。

还需要说明的是，并不需要把设计结构引入到用例模型里面去，用例只是用涉众可以理解的方式结构化了他们的关注点，而设计将反映出涉众如何感知系统，用例驱动了设计模型，这就是为什么我们把这种开发方式称之为用例驱动开发的原因。

2，用例模块包括用例结构

既然是基于所有针对各个用例的不同模型的切片进行了工作，就可以把这些切片放到一个单独的包中，我们称这样的包为“**用例模快**”，这个模块包括了各种模型的切片以及它们的依赖关系，当开发一个由用例模快组成的系统的时候，我们可以把每个用例模块视为一个单独的项目，在硬盘或者某个中心开发库中，某根目录对应于用例模块，而子目录对应于用例模块中的每个切片。

下图是一个用例模块的示意图，而表示的<<trace>>的依赖关系，表明上游模型得出的一个下游模型存在着一些规则，作为开发团队必须遵循的开发指南和原则。这种用例模块，也可以成为软件产品线的一个元素。进一步抽取掉模块的领域相关部分，使它具备通用性，并且适当的命名，加入应用场景和使用案例，就可以发展成一个**用例模式**。这种用例模式对用例切片的复用极其有帮助。



在软件开发中，**可追朔性**是一个很重要的概念，可追朔性表明上游模型元素与下游模型元素存在着链接，这可以帮助你判断是不是所有的需求都已经实现，需求的变化会对什么产生影响，通常上游元素的修改会影响到下游元素。

如果上游和下游元素遵循着不同的结构化原则，结果就会花费很多精力维护模型间的可追朔性，这在大型项目中显得尤其困难，但在我们保持模型结构的原则下，可追朔性的维护工作将显

著减少，这是因为模型本身就是基于相同结构的。

过去的设计方法，易变形和可维护性问题的解决，是在分析之后的设计过程中完成的，造成了设计模型与概念模型之间的可追朔性很差。但在面向方面的软件设计理念中，这个问题的解决是在分析的时候就提出来了，通过模型的一致性达到了问题解决模型的可追朔性，这就大大提升了软件设计的质量。

第八章 模块结构设计

详细设计阶段的模块结构设计是解决如下问题：

1，给出软件结构中各模块的内部过程描述

2，模块的内部过程描述也就是模块内部的算法设计

3，详细设计有时需要导出一种算法设计表示，由此可以直接而简单地导出程序代码

在面向对象的方法中，详细设计阶段的一个十分重要的问题，是进行类设计。类设计直接对应于实现设计，它的设计质量直接影响着软件的质量，所以这个阶段是十分重要的。这就给我们提出了一个问题，类是如何确定的，如何合理的规划类，这就要给我们提出一些原则，或者是一些模式。设计模式是有关中小尺度的对象和框架的设计。应用在实现体系结构模式定义的大尺度的连接解决方案中。也适合于任何局部的详细设计。设计模式也称之为微观体系结构模式。

8.1 面向过程的详细设计

使用结构化构造（即用顺序、选择和重复三种程序结构）表示程序过程，降低程序的复杂性，从而提高可靠性、易测试性和易维护性。

一、详细设计工具

图形工具：将过程细节用图来表示，在图中，逻辑结构用具体的图形表示，包括：流程图、方块图、PAD 图等。

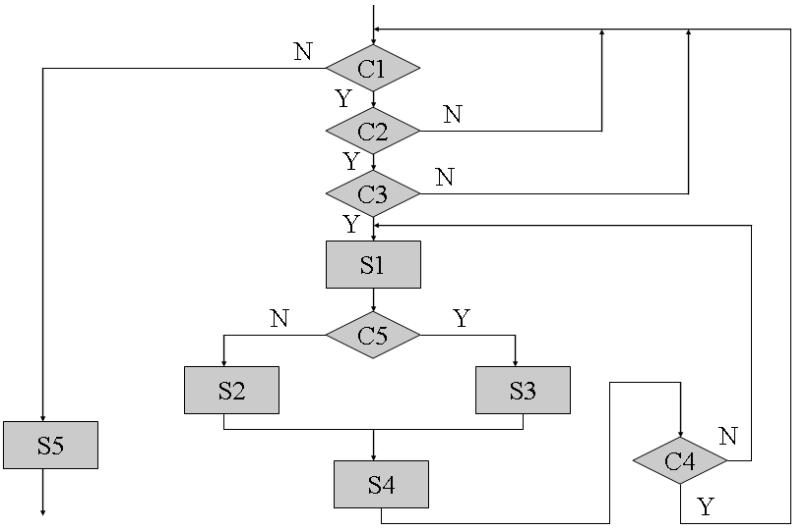
列表工具：利用表来表示过程细节，表列出了各种操作和相应的条件

语言工具：用类语言（伪码）表示过程的细节，很接近编程语言

二、流程图

符号：方框表示处理步骤、菱形表示逻辑判断、箭头表示控制流。

注意：用流程图表示过程细节时，要注意不要乱用箭头，否则会使结构不清晰



流程图的主要缺点：

1，流程图本质上不是逐步求精的好工具，它诱使程序员过早地考虑程序的控制流程，而不去考虑程序的全局结构。

2，流程图中用箭头代表控制流，因此程序员不受任何约束，可以完全不顾结构程序设计的精神，随意转移控制。

3，流程图不易表示数据结构。

三、方块图（N-S 图）

1，研制方块图的目的是：既要制定一种图形工具，又不允许它违反结构化原则。

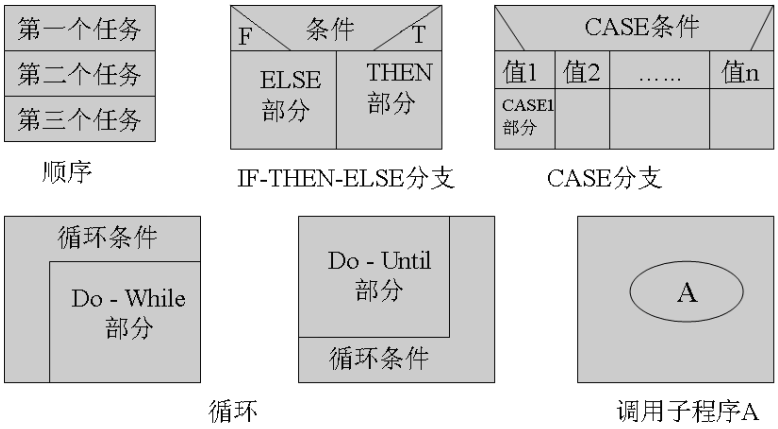
2，方块图具有以下特点：

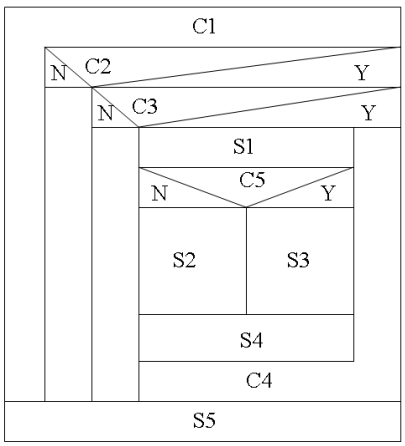
（1）功能域（即某一具体构造的功能范围）有明确的规定，并且很只观地从图形表示中看出来；

（2）想随意分支或转移是不可能的；

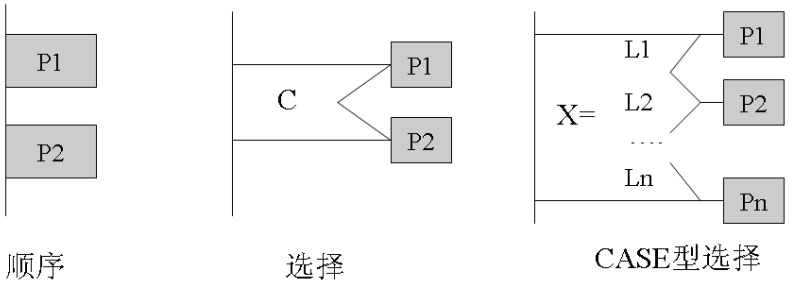
（3）局部数据和全程数据的作用域可以很容易确定；

（4）容易表示出递归结构。

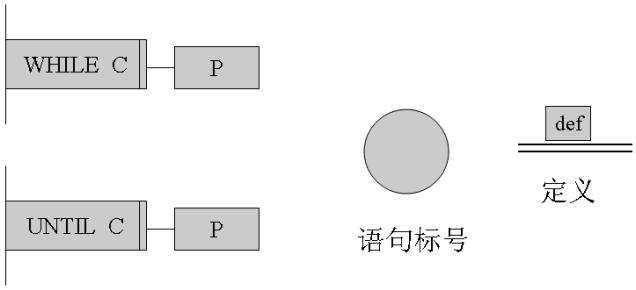




四、PAD 图（Problem Analysis Diagram）



顺序　　选择　　CASE型选择



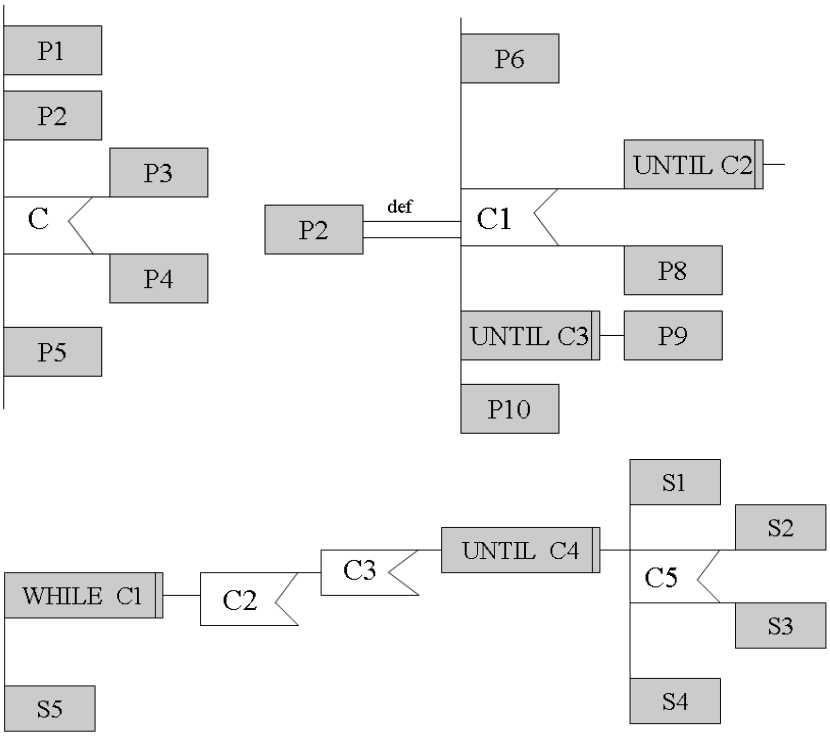
语句标号　　定义

循环

PAD 图的特点：

1，使用表示结构化控制结构的 PAD 符号所设计出的程序必然是结构化程序。

2，PAD 图所描述的程序结构十分清晰，图中最左面的竖线是程序的主线，即第一层结构，随着程序层次的增加，PAD 图逐渐向右延伸，每增加一个层次，图形向右扩展一条竖线，PAD 图中的竖线的总条数就是程序的层次数。



3，用 PAD 图表现程序逻辑，易读、易懂、易记，PAD 图是二维树形结构的图形，程序从图中最左竖线上端的结点开始执行，自上而下，从左向右顺序执行，遍历所有结点。

4，容易将 PAD 图转换成高级语言源程序，这种转换可用软件工具自动完成。

5，既可以用于表示程序逻辑，也可用于描述数据结构。

6，PAD 图的符号具有支持自顶向下、逐步求精方法的作用。开始时设计者可以定义一个抽象的程序，随着设计工作的深入而用 def 符号逐步增加细节，直至完成详细设计。

五、软件设计说明书

当详细设计完成以后，必须形成软件设计说明书，这个说明书应该表达的内容可以用下面的表来表示。

编号	名称	内容
1	范围	1）系统的目标和作为系统元素的软件的作用； 2）硬件、软件与人机接口； 3）主要的软件功能； 4）外部定义的数据库； 5）主要的设计约束与限制。
2	参考文档	1）现有的软件文档； 2）系统文档； 3）外购产品文档（硬件或软件）； 4）技术参考资料。
3	设计说明	1）数据说明 信息流的复审，信息结构的复审。 2）导出的软件结构 3）结构内的接口
4	模块（对每一个模块）	1）处理说明 2）接口说明 3）设计语言（或其他）的说明 4）使用的模块 5）数据的组织 6）注解
5	文件结构和全程数据	1）外部文件结构 逻辑结构，逻辑记录说明，存取方法。 2）全程数据 3）文件和数据的交叉引用
6	需求与模块的对照表	
7	测试的准备	测试大纲，组装策略，专门的考虑。
8	装配	专门的程序覆盖考虑，转录考虑。
9	专门注解	
10	附录	

8.2 UML 类图与代码结构设计

在类结构设计阶段，我们需要对 UML 类图与代码的关系有个清晰而且透彻的理解。下面我们分别加以讨论。

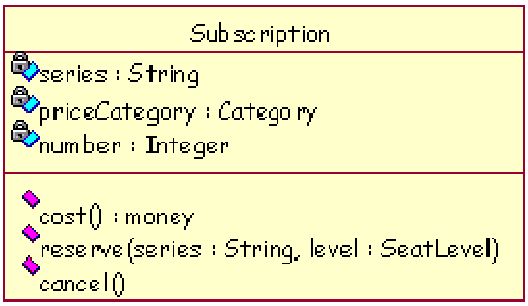
一、类

类代表了被建模的应用领域中的离散概念—物理实体（如飞机）、商业事物（如一份订单）、逻辑事物（如广播计划）、应用事物（如取消键）、计算机领域的事物（如哈希表）或行为事物（如一项任务）。类是有着相同结构、行为和关系的一组对象的描述符号。所用的属性与操作都被附在类或其他类元上。类是面向对象系统组织结构的核心。

对象是具有身份、状态和可激发行为的离散实体。对象是用来构造实际运行系统的个体；类

是用来理解和描述众多个别对象的个别概念。

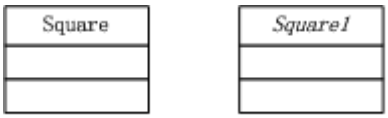
类定义了一组有着状态和行为的对象。属性和关联用来描述状态。属性通常用没有身份的纯数据值表示，如数字和字符串。关联则用有身份的对象之间的关系表示。个体行为由操作来描述，方法是操作的实现。对象的生命期由附加给类的状态机来描述。类的表示法是一个矩形，由带有类名、属性和操作的分格框组成。



一组类可以用泛化关系和建立在其内的继承机制分享公用的状态和行为描述。泛化使更具体的类（子类）与含有几个子类公同特性的更普通的类（超类）联系起来。一个类可以有零个或多个父类（超类）和零个或多个后代（子类）。一个类从它的双亲和祖先那里继承状态和行为描述，并且定义它的后代所继承的状态和行为描述。

类在它的包含者内有唯一的名字，这个包含者通常可能是一个包但有时也可能是另一个类。类对它的包含者来说是可见性，可见性说明它如何被位于它的可见者之外的类所利用。类的多重性说明了有多少个实例可以存在，通常情况下，可以有多个（零个或多个，没有明确限制），但在执行过程中一个实例只属于一个类。

1，类的名称：



上图中第二个斜体表示抽象类。

Java 语言中的类的简单定义形式：

[修饰符] class 类名 [extends 超类名] [implements 接口名]{
    内容
}

其中：

修饰符：public、abstract、final ，修饰符可以有多个。

超类名为这个类所继承的父类，缺省为 object。

Implements 为实现接口。

在 C#中，可以简单的用“逗号”表达：

[修饰符] class 类名 [: 超类名] [,接口名]{
    内容
}

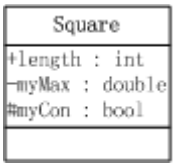
其中：

修饰符：public、abstract 等修饰符可以有多个。

用关键字 abstract 修饰的类或方法称作抽象的类或方法，抽象类是一种没有完全定义的类，只有抽象类才能才能有抽象方法，抽象类不能创建实例。

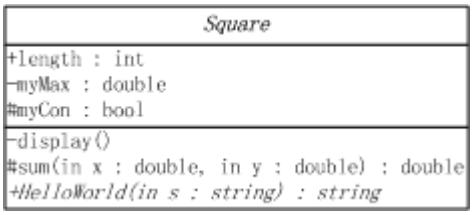
2，类的数据成员：

数据成员又称之为特性（有时也称之为属性）。



第一个为公有，第二个为私有，第三个为受保护的。

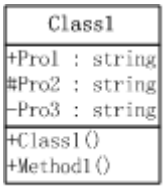
3，类的方法：



类本身是抽象类。

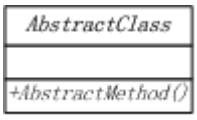
第一个是私有，第二个是受保护的，第三个是公有，但是是抽象的。

再比如：




```
public class Class1{
    public string Pro1;
    protected string Pro2;
    private string Pro3;
    public void Class1(){
    }
    public void Method1(){
    }
}
```

4，抽象类和抽象成员

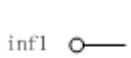


```
public abstract class AbstractClass{
    public abstract void AbstractMethod();
}
```

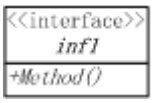
抽象类在开发中是一种占有非常重要地位的类，抽象方法只能声明在抽象类里面。使用抽象类的主要原因是：有时候父类声明的一些方法，只能由它的子类完成，如果将这个类声明成抽象的，就可以防止这个类被实例化。

二、接口

接口是在没有给出对象的实现和状态的情况下对对象行为的描述。接口包含操作但不包含属性，并且它没有对外界可见的关联。一个或多个类或构件可以实现一个接口，并且每个类都可以实现接口中的操作。



或者




```
public interface infl{
    void Method();
}
```

三、数据类型

数据类型用以描述缺少身份的简单数据值。数据类型包括数字、字符串、枚举型数值。数据类型通过值来传递，并且是不可变的实体。数据类型没有属性，但是可以有操作。操作不改变数据值，但是可以把数据值作为结果返回。

四、含义分层

类可以存在于模型的几种含义层中，包括分析层、设计层和实现层。当表现真实世界的概念时，说明实际的状态、关系和行为是很重要的。但是像信息隐藏、有效性、可见性和方法这些实现方面的概念与真实世界的概念无关（它们是设计层的概念）。分析层的类代表了在应用域中的逻辑概念或应用本身。分析层模型应该尽可能少地表示建模的系统，并在不涉及到执行和构造的情况下，充分说明系统的必要逻辑组成。

当表示高层次的设计时，有些概念与类直接相关，这些概念包括特定类的状态定位、对象之间导航的效率、外部行为和内部实现的分离和准确操作的描述。设计层类表示了将状态信息和其上的操作封装于一个离散的单元，它说明了关键的设计决定、信息定位和对象的功能。设计层类包含真实世界和计算机系统两方面的内容。

最后，当实现程序代码时，类的形式与所选择的语言紧密相关。如果一个通用类的功能不能直接用语言来实现，那么不得不放弃它们。实现层类直接与程序代码相对应。

同一个系统可以容纳多个层次的类，面向实现的类会在模型中实现更逻辑化的类。一个实现类表示用特定程序设计语言声明一个类，它得到了一个按照语言所要求的准确格式的类。然而，在许多情况下，分析、设计和实现信息可嵌套在一个单独的类中。

五、关系

类元之间的关系有关联、泛化、流及各种形式的依赖关系，这在 UML 概述里面已经讨论过，这里所强调的，是在 UML 图上所标出的关系，到底与具体的代码是如何联系的，对这个问题的理解，对于良好的设计十分重要。

1，关联

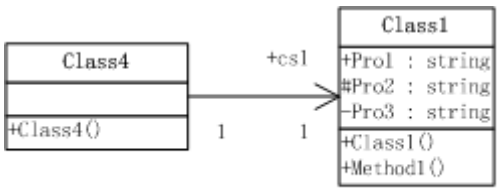
关联描述了系统中对象或实例之间的离散连接。关联将一个含有两个或多个有序表的类元，在允许复制的情况下连接起来。最普通的关联是一对类元之间的二元关联。关联的实例之一是链。每个链由一组对象（一个有序列表）构成，每个对象来自于相应的类。二元链包含一对对象。

关联关系一般性的描述如下：

1）关联关系

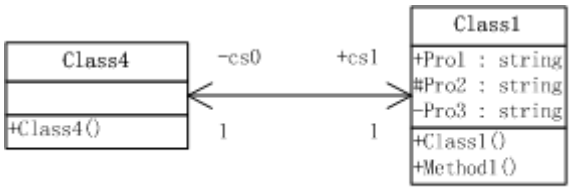
类之间的关系，表示一个类“知道”另一个类，用实线表示关联，关联可以是单向的，也可以是双向的，关联的方向用箭头表示。

每个端可以有基数，表示这一端类的实例个数。




```
public class Class4{
    public Class1 cs1;
    public void Class4(){
    }
  }
```

双向关联：

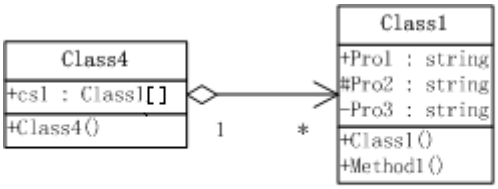



```
public class Class1{
    public string Pro1;
    protected string Pro2;
    private string Pro3;

    private Class4 cs0;

    public void Class1(){
    }
```

```
    public void Method1(){
    }
}

public class Class4{
    public Class1 cs1;
    public void Class4(){
    }
}
```

2）聚合关系

表示一种弱的拥有关系，即 A 对象可以包含 B 对象，但 B 对象不是 A 对象的一部分。



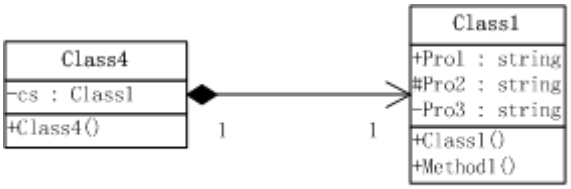

```
public class Class4{
    public Class1[] cs1;
    public void Class4(){
    }
}
```

聚合在逻辑关系上可以用“汽车-停车场”关系来说明，停车场中有汽车，但汽车不是停车场的一部分，汽车和停车场没有整体和一般的关系。

聚合关系的对象之间没有依赖关系，也就是对象的创建和消失没有绝对的先后顺序，在 C# 中，聚合关系和关联关系实现上是相同的。

3）组合关系

也称为复合或者合成关系，组合关系具有严格的“部分-整体”关系，部分和整体生命周期是相同的。

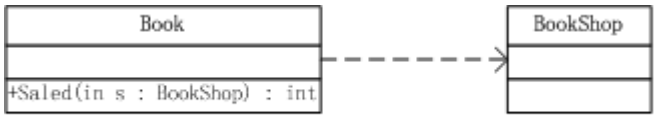



```
public class Class4{
    private Class1 cs;
    public void Class4(){
      cs=new Class1();
    }
}
```

在严格的组合关系中，类之间有强依赖关系，在很多情况下，代表整体的对象需要保持部分对象的存活。

4）依赖关系

依赖关系表示类出现在局部变量或者方法的参数中，或者是类的静态方法被调用。




```
public class Book{
    public int Saled(BookShop s){
    }
}
```

或者：

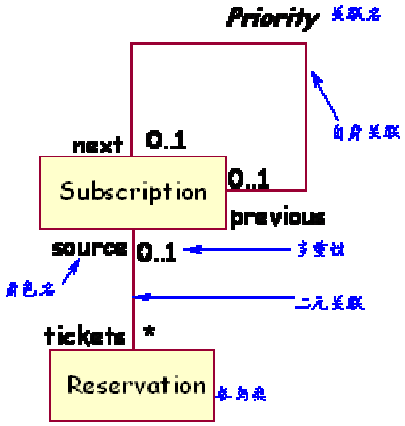
```
public class Book{
    public int Saled(){
        BookShop s=new BookShop();
    }
}
```

关联带有系统中各个对象之间关系的信息。当系统执行时，对象之间的连接被建立和销毁。关联关系是整个系统中使用的“胶粘剂”，如果没有它，那么只剩下不能一起工作的孤立的类。

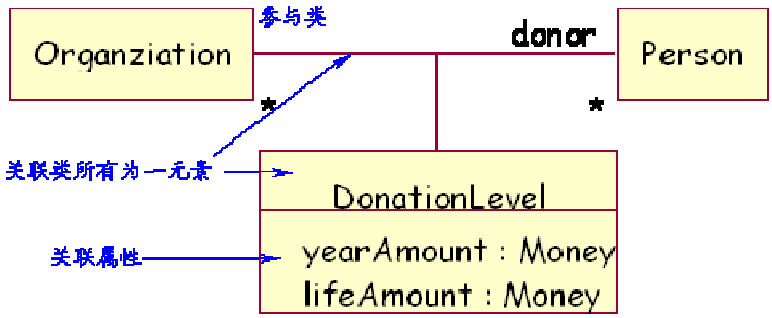
在关联中如果同一个类出现不止一次，那么一个单独的对象就可以与自己关联。如果同一个类在一个关联中出现两次，那么两个实例就不必是同一个对象，通常的情况都如此。

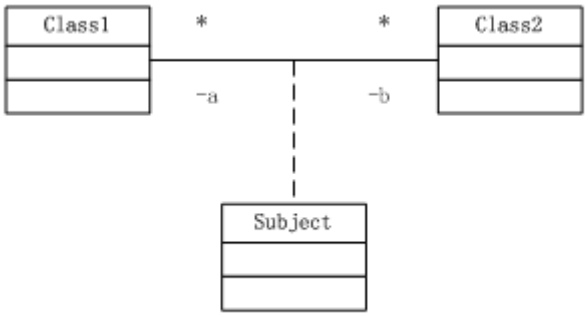
一个类的关联的任何一个连接点都叫做关联端，与类有关的许多信息都附在它的端点上。关联端有名字（角色名）和可见性等特性，而最重要的特性则是多重性，重性对于二元关联很重要，因为定义 n 元关联很复杂。

二元关联用一条连接两个类的连线表示。如下图所示，连线上有相互关联的角色名而多重性则加在各个端点上。



如果一个关联既是类又是关联，即它是一个关联类，那么这个关联可以有它自己的属性，如下图。

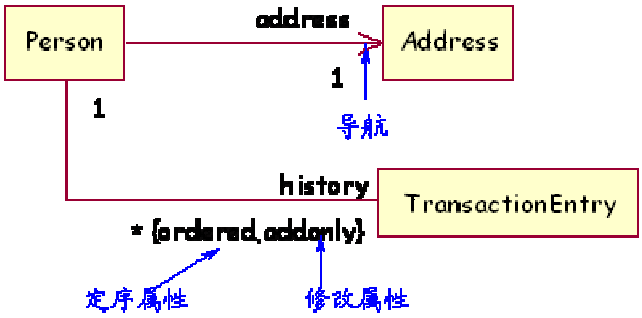





```
public class Class1{
}
public class Class2{
}

public class Subject{
    private Class1 a = null;
    private Class2 b = null;
}
```

六、关联关系进一步讨论

在分析阶段，关联表示对象之间的逻辑关系。没有必要指定方向或者关心如何去实现它们。应该尽量避免多余的关联，因为它们不会增加任何逻辑信息。在设计阶段，关联用来说明关于数据结构的设计决定和类之间职责的分离。此时，关联的方向性很重要，而且为了提高对象的存取效率和对特定类信息的定位，也可引入一些必要的多余关联。然而，在该建模阶段，关联不应该等于 C++ 语言中的指针。在设计阶段带有导航性的关联表示对一个类有用的状态信息，而且它们能够以多种方式映射到程序设计语言当中。关联可以用一个指针、被嵌套的类甚至完全独立的表对象来实现。其他几种设计属性包括可见性和链的可修改性。 下图表示了一些关联的设计特性。



1. 聚集和组成

聚集表示部分与整体关系的关联，它用端点带有空菱形的线段表示，空菱形与聚集类相连接。组成是更强形式的关联，整体有管理部分的特有的职责，它用一个实菱形物附在组成端表示。每个表示部分的类与表示整体的类之间有单独的关联，但是为了方便起见，连线结合在一起，现在整组关联就像一棵树。 下图表示了聚集关联和组成关联。

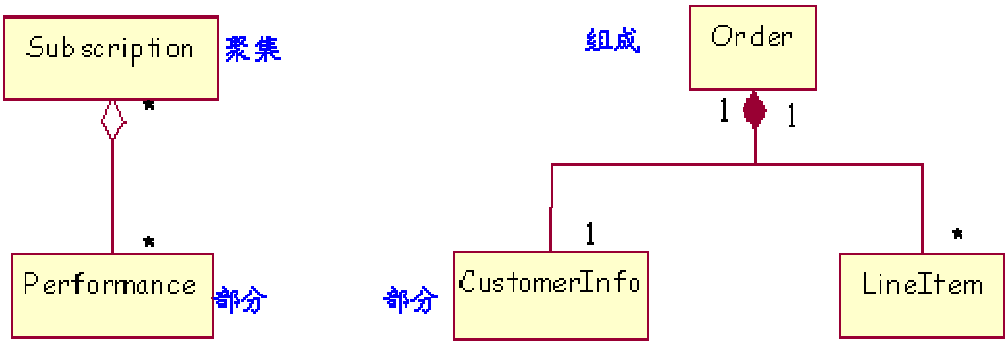


图 聚集和组成

2. 链

链是关联的一个实例。链即所涉及对象的一个有序表，每个对象都必须是关联中对应类的实例或此类后代的实例。系统中的链组成了系统的部分状态。链并不独立于对象而存在，它们从与之相关的对象中得到自己的身份（在数据库术语中，对象列表是链的键）。在概念上，关联与相关类明显不同。而在实际中，关联通常用相关类的指针来实现，但它们可以作为与其相连的类分离的包含体对象来实现。

3. 双向性

关联的不同端很容易辨认，哪怕它们都是同一种类。这仅仅意味着同一个类的不同对象是可以相互联系的。正是因为两端是可区分的，所以关联是不对称的（除了个别的例子外），且两个端点也是不能互相交换的。在通常情形下这是一个共识：就像动词短语中的主语和宾语不能互换一样。关联有时被认为是双向性的，这意味着逻辑关系在两个方向上都起作用。这个观点经常被错误理解，甚至包括一些方法学家。这并不意味着每个类“了解”其他类，或者说在实现中类与类之间可以互相访问。这仅仅意味着任何逻辑关系都有其反向性，无论这个反向性容不容易计算。如果关联只在一个方向横穿而不能在另一个方向横穿，那么关联就被认为有导航性。

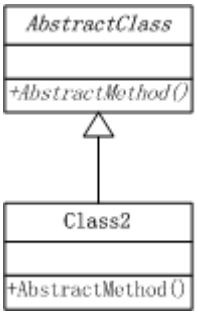
为什么使用基本模型，而不用编程语言中流行的指针来表示关联？原因是模型试图说明系统

实现的目的。如果两个类之间的关系在模型中用一对指针来表示，那么这两个指针仍然相关。关联方法表明关系在两个方向都有意义，而不管它们是如何实现的。将关联转化成一对用于实现的指针很容易，但是很难说明这两个指针是彼此互逆的，除非这是模型的一部分。

七、泛化

1，泛化

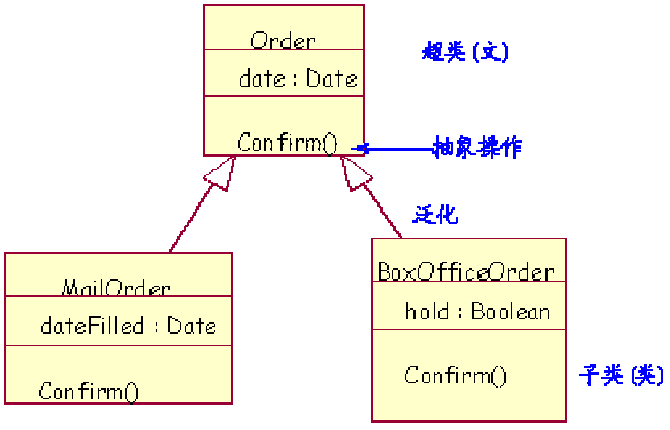
泛化关系实际上表达的是类的继承。




```
//AbstractClass的泛化是Class2
public class Class2  : AbstractClass{
    public void AbstractMethod()
    {
    }
}
```

泛化关系是类元的一般描述和具体描述之间的关系，具体描述建立在一般描述的基础之上，并对其进行了扩展。具体描述与一般描述完全一致所有特性、成员和关系，并且包含补充的信息。例如，抵押是借贷中具体的一种，抵押保持了借贷的基本特性并且加入了附加的特性，如房子可以作为借贷的一种抵押品。一般描述被称作父，具体描述被称作子如借贷是父而抵押则是子。泛化在类元（类、接口、数据类型、用例、参与者、信号等等）、包、状态机和其他元素中使用。在类中，术语超类和子类代表父和子。

泛化用从子指向父的箭头表示，指向父的是一个空三角形（如下图表示）。多个泛化关系可以用箭头线组成的树来表示，每一个分支指向一个子类。



泛化的用途

泛化有两个用途。第一个用途是用来定义下列情况：当一个变量（如参数或过程变量）被声明承载某个给定类的值时，可使用类（或其他元素）的实例作为值，这被称作可替代性原则（由 Barbara Liskov 提出）。该原则表明无论何时祖先被声明了，则后代的一个实例可以被使用。例如，如果一个变量被声明拥有借贷，那么一个抵押对象就是一个合法的值。

泛化使得多态操作成为可能，即操作的实现是由它们所使用的对象的类，而不是由调用者确定的。这是因为一个父类可以有许多子类，每个子类都可实现定义在类整体集中的同一操作的不同变体。例如，在抵押和汽车借贷上计算利息会有所不同，它们中的每一个都是父类借贷中计算利息的变形。一个变量被声明拥有父类，接着任何子类的一个对象可以被使用，并且它们中的任何一个都有着自己独特的操作。这一点特别有用，因为在不需要改变现有多态调用的情况下就可以加入新的类。例如，一种新的借贷可被新增加进来，而现存的用来计算利息操作的代码仍然可用。一个多态操作可在父类中声明但无实现，其后代类需补充该操作的实现。这种不完整操作是抽象的（其名称用斜体表示）。

泛化的另一个用途是在共享祖先所定义的成分的前提下允许它自身定义增加的描述，这被称作继承。继承是一种机制，通过该机制类的对象的描述从类及其祖先的声明部分聚集起来。继承允许描述的共享部分只被声明一次而可以被许多类所共享，而不是在每个类中重复声明并使用它，这种共享机制减小了模型的规模。更重要的是，它减少了为了模型的更新而必须做的改变和意外的前后定义不一致。对于其他成分，如状态、信号和用例，继承通过相似的方法起作用。

2，继承

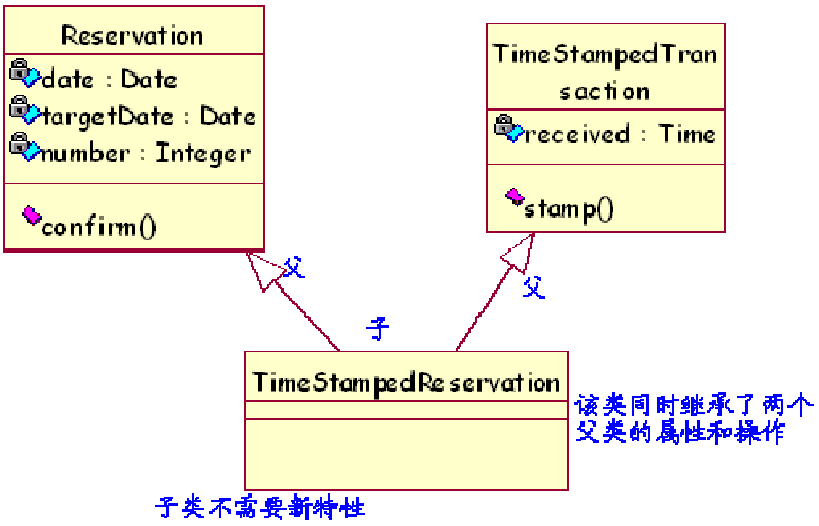
每一种泛化元素都有一组继承特性。对于任何模型元素的包括约束。对类元而言，它们同样包括一些特性（如属性、操作和信号接收）和关联中的参与者。一个子类继承了它的所有祖先的可继承的特性。它的完整特性包括继承特性和直接声明的特性。

对类元而言，没有具有相同特征标记的属性会被多次声明，无论直接的或继承的，否则将发生冲突，且模型形式错误。换而言之，祖先声明过的属性不能被后代再次声明。如果类的接口一致（具有同样的参数、约束和含义），操作可在多个类中声明。附加的声明是多余的。一个方法在层次结构中可以被多个类声明，附在后代上的方法替代（重载）在任何祖先中声明过的具有相同特征标记的方法。如果一个方法的两个或多个副本被一个类继承（通过不同类的多重继承），那么它们会发生冲突并且模型形式错误（一些编程语言允许显式选定其中的一种方法。我们发现如果在后代类中重新定义方法会更简单、安全）。元素中的约束是元素本身及它所有祖先的约束的联合体，如果它们存在不一致，那么模型形式错误。

在一个具体的类中，每一个继承或声明的操作都必须有一个已定义的方法，无论是直接定义或从祖先那里继承而来的。

3，**多重继承**

如果一个类元有多个父类，那么它从每一父类那里都可得到继承信息（如下图）。它的特征（属性、操作和信号）是它的所有父类特征的联合。如果同一个类作为父类出现在多条路径上，那么它的每一个成员中只有它的一个拷贝。如果有着同样特征的特性被两个类声明，而这两个类不是从同一祖先那里继承来的（即独立声明），那么声明会发生冲突并且模型形式错误。因为经验告诉我们设计者应自行解决这个问题，所以 UML 不提供这种情形的冲突解决方案。像 Eiffel 这样的语言允许冲突被程序设计者明确地解决，这比隐式的冲突解决原则要安全，而这些原则经常使开发者大吃一惊。



八、单分类和多重分类

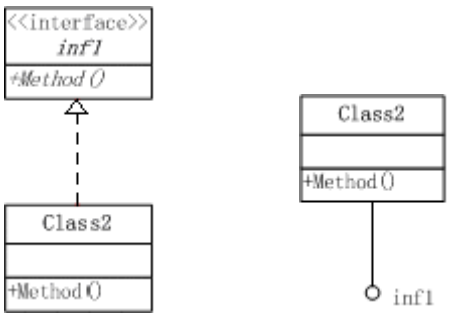
在最简单的形式中，一个对象仅属于一个类，许多面向对象的语言有这种限制。一个对象仅属于一个类并没有逻辑上的必要性，我们只要从多个角度同时观察一下真实世界的对象就可以发现这一点。在 UML 更概括的形式中，一个对象可以有一个或多个类。对象看起来就好像它属于一个隐式类，而这个类是每个直接父类的子类 — 多重继承可以免去再声明一个新类，这可提高效率。

九、静态与动态类元

在最简单的形式中，一个对象在被创建后不能改变它的类。我们再次说明，这种限制并没有逻辑上的必要性，而是最初目地是使面向对象编程语言的实现更容易些。在更普遍的形式下，一个对象可以动态改变它的类，这么做会得到或失去一些属性或关联。如果对象失去了它们，那么在它们中的信息也就失去了并且过后也不能被恢复，哪怕这个对象变回了原来的类。如果这个对象得到了属性或关联，那么它们必须在改变时就初始化，就像初始化一个新对象一样。当多重分类和动态分类一起使用时，一个对象就可以在它的生命期内得到或失去类。动态类有时被称作角色或类型。一个常见的建模模式是每个对象有一个唯一的静态的固有类（即不能在对象的生命期内改变的类），加上零个或多个可以在对象生命期内加入或移走的角色类。固有类描述了这个对象的基本特性，而角色类描述了暂时的特性。虽然许多程序设计语言不支持类声明中的多重动态分类，然而它仍然是一个很有用的建模概念，并且可以被映射到关联上。

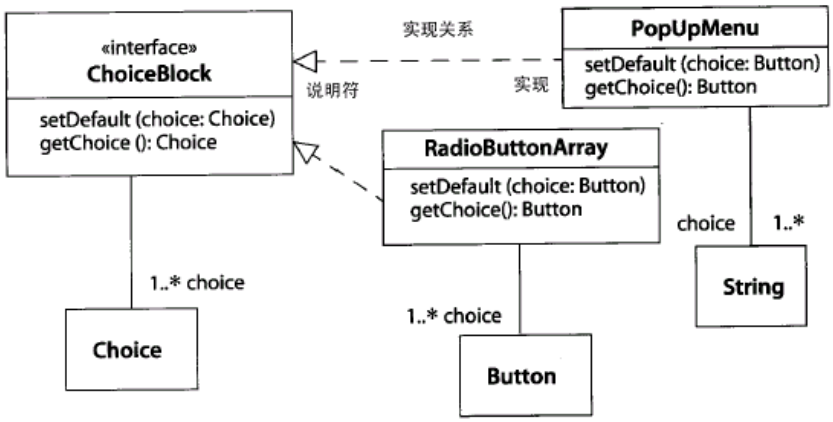
十、实现

实现接口。



实现关系将一种模型元素（如类）与另一种模型元素（如接口）连接起来，其中接口只是行为的说明而不是结构或者实现。客户必须至少支持提供者的所有操作（通过继承或者直接声明）。虽然实现关系意味着要有像接口这样的说明元素，它也可以用一个具体的实现元素来暗示它的说明（而不是它的实现）必须被支持。例如，这可以用来表示类的一个优化形式和一个简单低效的形式之间的关系。

泛化和实现关系都可以将一般描述与具体描述联系起来。泛化将在同一语义层上的元素连接起来（如，在同一抽象层），并且通常在同一模型内。实现关系将在不同语义层内的元素连接起来（如，一个分析类和一个设计类；一个接口与一个类），并且通常建立在不同的模型内。在不同发展阶段可能有两个或更多的类等级，这些类等级的元素通过实现关系联系起来。两个等级无需具有相同的形式，因为实现的类可能具有实现依赖关系，而这种依赖关系与具体类是不相关的。实现关系用一条带封闭空箭头的虚线来表示（如下图），且与泛化的符号很相像。



用一种特殊的折叠符号来表示接口（无内容）以及实现接口的类或构件。接口用一个圆圈表示，它通过实线附在表示类元的矩形上（如下图）。

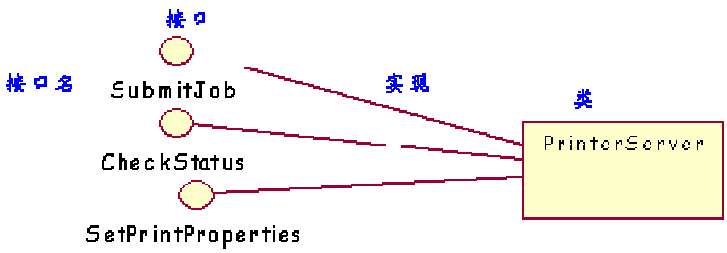


图　接口和实现图标

十一、实例

实例是有身份标识的运行实体，即它可以与其他运行实体相区分。它在任何时刻都有一个值，随着对实例进行操作值也会被改变。

模型的用途之一是描述一个系统的可能状态和它们的行为。模型是对潜在性的描述，对可能存在的对象集和对象经历的可能行为历史的描述。静态视图定义和限制了运行系统值的可能配置。动态视图定义了运行系统从一个配置传递到另一个的途径。总之，静态视图和建立在其上的各种动态视图定义了系统的结构和行为。

在某一时刻一个系统特定的静态配置叫做快照。快照包括对象和其他实例、数值和链。对象是类的实例，是完全描述它的类的直接实例和那个类的祖先的间接实例（如果允许，重分类对象可能是多个类的直接实例）。同样，链是关联的实例，数值是数据类型的实例。

对象对于它的类的每个属性有一个数据值，每个属性值必须与属性的数据类型相匹配。如果属性有可选的或多重的多重性，那么属性可以有零个或多个值。链包含有多个值，每一个值是一个给定类的或给定类的后代的对象的引用。对象和链必须遵从它们的类或关联的约束（其中既包括明确的约束又包括如多重性的内嵌的约束）。

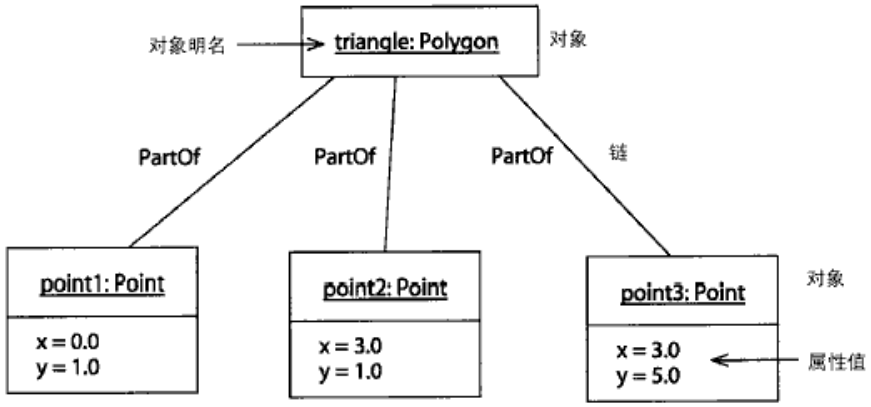
如果系统的每个实例是一个形式良好的系统模型的一些元素的实例，并且实例满足模型的所有约束，则说明系统的状态是有效的系统实例。

静态视图定义了一组能够在单独快照中存在的对象、值和链。原则上，任何与静态图相一致的对象和链的结合都是一个模型可能的配置。但这不意味着每个可能的快照能够或将要出现。某些快照可能在静态下合法但在系统的动态图下可能不会被动态地达到。UML 的行为部分描述了快照的有效顺序，快照可能作为部和内外部行为影响的结果出现。动态图定义了系统如何从一个快照转换到另一个快照。

十二、对象图

快照的图是系统在某一时刻的图像。因为它包含对象的图像，因此也被叫做对象图。作为系统的样本它是有用的，如可以用来说明复杂的数据结构或一系列的快照中表示行为（如下图）。请记住所有的快照都是系统的样本，而不是系统的定义。系统结构和行为在定义视图中定义，且建立定义视图是建模和设计的目标。

静态视图描述了可能出现的实例。除了样本外，实际的实例不总是直接在模型中出现。



上面这些概念，对于 UML 在设计中的应用是非常重要的。

8.3 设计模式简介

在软件分析与设计中，大量的使用着“模式”的概念。所谓“模式”强调的是某种功能单元可能被使用上百次，但使用的方式却不尽相同。模式是一种灵活的思想，运用它却需要智慧和想象力，因为没有两种完全一样的功能需求。

“模式”这个词最初来自于 Christopher Alexander（克里斯多弗. 亚历山大），在城市规划和建设的过程中，他发现许多相同的原则，在 Pattern Language（模式语言）这本书中，他把建筑学中大量简单事务划分为若干模式（咖啡馆、街角杂货铺、天窗、与腰等高的柜台等）当拥有丰富的模式就相当于拥有的大量的词汇，有助于使设计更完美。

这个思想同样可以用在软件分析与设计中，当拥有大量模式的时候，就可以使自顶向下的分解变得更加主动。

我们不能认为任何以面向对象方法构造的软件，都便于实现软件复用，而模式使这种复用成为可能。设计模式是一种设计结构的复用，设计模式实现了“隔离问题，封装变化，利于系统扩展与互连”的目的，设计模式也有助于类的职责分配。

设计模式并不是反映了业务领域**事物**之间特有的共性结构，而是抽取了具体的领域特征，根据**问题**的共性特点，寻找到的解决方案的共性结构，因此，它具备更广阔的复用可能。

8.4 算法结构复杂性分析及度量

一、控制流结构及模型

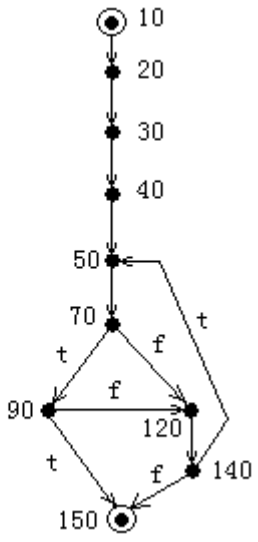
1，控制流结构

当我们需要测量程序或者算法的控制流结构的时候。对控制流的建模可以采用一种有向图，称之为**控制流图**（control-flow graph）或者称**流图**（flowgraph）。图中，每个节点，代表一条程序语句，每段弧（或有向边）表示从一条语句到另一条语句的控制流。

下图表示了一个简单程序 A 及其对应流图 F(A)的合理表示。

所谓合理，指的是人们总是不太清楚如何把程序 A 映射成流图 F(A)。流图是定义控制流结构度量的一个良好模型，因为它对程序的许多结构性质进行了清晰的展示。

```
10      double P=10;
20      double div = 10;
30      double Lim = Math.Abs(Math.Sqrt(P));
40      double Flag = P / div - Math.Abs(P / div);
50      do
60      {
70          if (Flag = 0 || div==Lim)
80          {
90              if (Flag != 0 || P > 4)
100              break;
110         }
120         div += 1;
130     }
140     while (div < 10);
150     //end
```



我们先来讨论一下与图相关的各种术语：

图（graph）：有一组点（或结点）和线段（或者边）组成的结构体系。

有向图（directed graph）：每个边都指定了方向，用箭头表示。

弧（arc）：把有向边称为弧。通常把弧写成序数<x,y>，其中，x 和 y 是形成弧两端的结点，方向是从 x 到 y。

入度（in-degree）：到达一个节点的弧的数量。

出度（out-degree）：离开一个节点的弧的数量。

路径（path）：是指前后连续的有向边序列，在这个序列中，经过某些边的次数可能不止一次。

简单路径（simple path）：指不会出现重复边的路径。

例如：在上面的例子里，结点 70 的**入度**是 1，**出度**是 2。

流图（flow graph）：所谓流图是指这样一种图，即图中有两个结点，**开始结点**（start node）和**终止节点**（stop node），开始节点入度为零，终止结点出度为零，每个结点都位于从开始节点到终止结点的某条路径上。

过程节点（procedure node）：出度等于 1 的结点成为过程节点。

谓词节点（predicate node）：除过程节点之外的所有节点（除终止节点之外）。

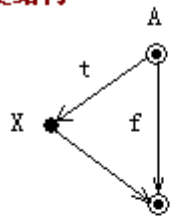
我们为程序结构建模的过程中，有些流图是经常见到的，有必要对它进行特殊的命名。

顺序结构

Pn 或者 Pn（X1，X2,...Xn）

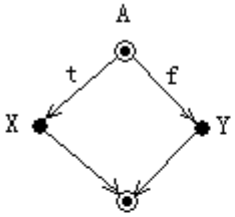
X1;X2;...Xn

分支结构



D0 或者 D0(A,X)

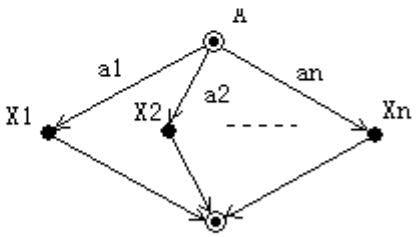
if (A) { X }



D1 或者 D1（A,X,Y）

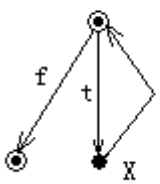
if (A) { X } else { Y }

Cn 或者 Cn(A,X1,X2,...Xn)



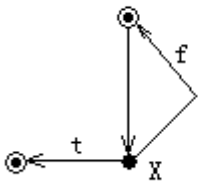

```
switch (A)
{
     case a1:
         X1
         break;
     case a2:
         X2
         break;
     .....
     case an:
         Xn
         break;
}
```

循环结构



D2 或者 D2(A,X)

while (A) { X }



D3 或者 D3(A,X)

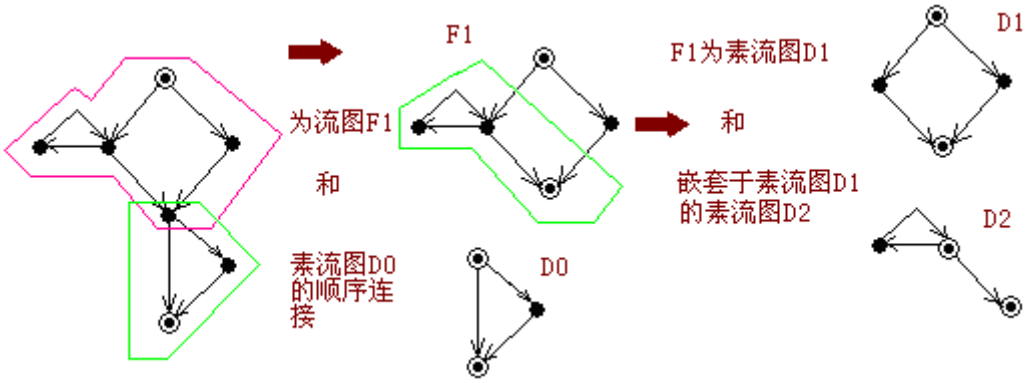
```
do{ X }
while (A);
```

上述的流图有个很重要的共同性质，就是它可以当作结构化程序的“积木”来使用，为了理解这个问题，必须对构建流图的各种方式进行形式化定义。

2，素分解

Fenton 和 Whitty 的研究表明，我们可以把任意一个流图与**分解树**（decomposition tree）相联系，来描述如何通过素流图进行顺序连接和嵌套操作来构建图的。

下图是根据给定的流图来确定分解树的简单例子。



研究表明，我们不但总是能够把流图分解成素流图，而且还能确信这种分解始终是唯一的。这个结论被称之为**素分解定律**（prime decomposition theorem）。

唯一定义的素分解树，是对程序控制结构的一种定义性描述。

当然，对于一个比较大的流图来说，实现素分解并不是容易的事情，手工完成这种计算也是不切实际的，好在许多商业软件能够自动完成这项工作。

素分解的唯一性定律，为我们研究软件结构复杂性度量提供了一个手段，一般来说，程序包含的素流图越多，素流图的嵌套越深，程序结构往往也就越复杂。

二、程序复杂性及度量原则

软件复杂性度量：

开发规模相同、复杂性不同的软件，花费的时间和成本会有很大差异。我们可以从六个方面描述软件复杂性

①理解程序的难度；

②纠错、维护程序的难度；

③向他人解释程序的难度；

④按指定方法修改程序的难度；

⑤根据设计文件编写程序的工作量；

⑥执行程序时需要资源的程度。

它反映了软件的可理解性、模块性、简洁性等属性。

软件复杂性度量的原则：

1.软件复杂性与程序大小的关系不是线性的；
2.控制结构复杂的程序较复杂；
3.数据结构复杂的程序较复杂；
4.转向语句使用不当的程序较复杂；
5.循环结构比选择结构复杂；选择结构又比顺序结构复杂；
6.语句、数据、子程序和模块在程序中的次序对复杂性有影响；
7.全程变量、非局部变量较多时，程序较复杂；
8.参数按地址调用比按值调用复杂；
9.函数副作用比显式参数传递难于理解；
10.具有不同作用的变量共用一个名字时较难理解；
11.模块间、过程间联系密切的程序比较复杂；
12.嵌套越深程序越复杂。

三、McCabe 圈复杂性度量

不管怎么说，直接使用素分解进行复杂性度量还是比较困难的，使用中还是希望找到一种简单直接的方法来解决这个问题，另外，人们希望找到一种方法，把边数、结点数以及素流图数综合在一起考虑，**McCabe** 圈复杂性度量为我们解决这个问题提供了可能。

McCabe 圈复杂性度量是对层次化度量深入研究的结果，并且提供了一个简单易行的表达式，下面我们来研究一下这个问题。

我们已经详细的研究了**控制流图**（或称程序控制结构图），我们知道：

程序结构对应于有一个入口结点和一个出口结点的有向图。

图中每个结点对应一个语句或一个顺序流程的程序代码块。

弧对应于程序中的转移。

由入口结点可以到达图中每个结点，并且从图中每个结点都可以到达出口结点。

McCabe 用程序控制结构图的圈数（巡回秩数）V(G)作为程序结构复杂性的度量

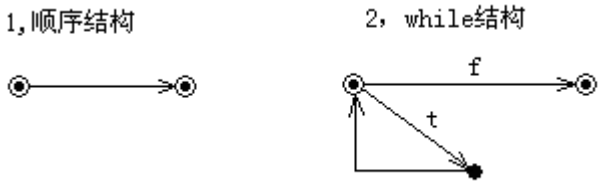
$$V(F) = e-n+2$$

其中：e 为结构图的边数

n 为结构图的结点数

可以证明 V(G)等于结构图中有界或无界的封闭区域个数

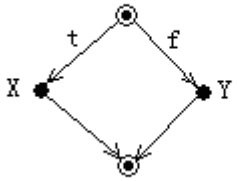
例：计算如图所示程序控制结构图的 V(F)值。



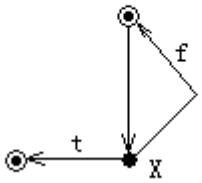
(1) e=1,n=2,v=1;

(2) e=3,n=3,v=2;

3,选择结构

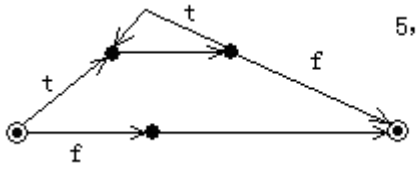


4，repeat结构



(3) e=4,n=4,v=2；

(4) e=3,n=3,v=2;



5，复合结构

(5) e=6,n=5,v=3.

McCabe 建议把 V(F)作为模块规模的定量指标，一个模块 V(F)的值不要大于 10 当 V(F)>10 时，模块内部结构就会变得复杂，给编码和测试带来困难。

程序中分枝结构数和循环结构数增加时，控制结构图的区域数增加，V(F)的值增大，程序的结构会变得更复杂。

结构化程序设计控制流力求从高层指向低层，从低层指向高层的流向，会增加封闭区域的个数，反方向的控制流向越多 V(F)越大，程序结构越复杂。

McCabe 圈数结构复杂度度量使用起来非常方便，在实践中也有很好的应用，但是让人十分怀疑的是，这个假设与复杂性的的直观关系是相对应的吗？我们无法说明这个度量就是复杂性度量。但是，用圈数确定程序或者模块的测试和维护难度的有用的指示器是非常有效的。在这个背景下，V(F)可以用于对产品的质量保证。

例：Grady 曾经报告过 HP 公司的一个研究项目，对 850000 行 FORTRAN 代码的每个模块计算了圈数。研究人员发现，在模块圈数和需要更新的次数之间存在着紧密联系。在对更新次数超过三次的模块上造成的成本和进度影响进行研究以后，研究小组得出结论，模块中允许的最大圈数为 15。

例：Channel Tunnel 铁路系统中的软件质量保证规程要求：如果 Logiscope 工具发现模块的圈数超过 20，或者模块的语句数超过 50，就会认为不合格。

总结：

本课程全面地从面向过程（结构化）和面向对象两种方法论的领域，讨论了软件系统分析与设计的方法。作为软件工程学的基础课程，我们对各个节点之间的相互关系，以及方法上的基本内容作了深入地探讨。但是，对于软件工程学中更深入的内容，我们会在更专业的“需求工程”、“软件体系结构”、“程序设计”等课程中会详加研究。

必须注意，作为分析与设计方法的基础研究课程，为学好软件工程学科其它课程提供了的共同能力基础，所以学习好本课程，对于学习软件工程学全面的知识是非常必要的。