
架构师培训讲义

2006-12

本文档是中科院计算所的架构师培训讲义，版权归其所有。

目录

目录.....	2
第一章 软件架构设计思想与体系创建.....	1
第一节 软件架构师的角色和应掌握的知识体系.....	1
第二节 软件分析和设计的方法学问题.....	3
第三节 在信息技术战略规划（ITSP）中的软件架构.....	4
第四节 软件生命周期设计与统一软件开发过程.....	10
第二章 需求过程与分析的核心理论.....	26
第一节 需求过程在软件架构中的重要作用.....	26
第二节 面向过程的需求分析核心知识.....	28
第三节 面向对象的需求分析和用例.....	36
第四节 用例的事件流与用例文档.....	40
第五节 非功能性需求的识别.....	51
第六节 合理的应用活动分析.....	52
第三章 领域建模与系统行为分析.....	66
第一节 领域建模的思想和方法.....	66
第二节 领域模型的关联.....	73
第三节 领域模型的属性.....	75
第四节 泛化建模.....	76
第五节 精化领域建模及若干难以确定的要素.....	82
第六节 系统行为分析中必须关注的问题.....	88
第七节 迭代计划和风险管理.....	101
第八节 领域建模的实例分析.....	103
第四章 高层软件架构的设计.....	114
第一节 高层软件架构的规划.....	114
第二节 面向过程的架构设计.....	118
第三节 面向对象的架构设计.....	120
第四节 高层设计中的架构分析.....	122
第五节 高层架构设计中的层模式.....	126
第六节 框架设计的方法学问题.....	132
第七节 面向服务架构（SOA）.....	136
第八节 软件架构的质量描述和评估.....	140
第五章 详细设计阶段的数据库结构设计.....	144
第一节 关系数据库的结构设计.....	144
第二节 面向对象数据库设计.....	152
第三节 并发问题及其应对.....	157
第四节 处理事务.....	158
第五节 数据库结构设计案例.....	162
第六章 详细设计阶段的类结构设计.....	166
第一节 类结构设计中的通用职责分配软件模式.....	166
第二节 设计模式与软件架构.....	174
第三节 合理使用外观和适配器模式.....	176
第四节 封装变化的三种方式及评价.....	177

第五节	利用策略与工厂模式实现通用的框架.....	192
第六节	在团队并行开发中应用代理模式.....	222
第七节	利用观察者模式延长架构的生命周期.....	229
第八节	树状结构和链形结构的对象组织.....	232
第九节	委托技术与行为型设计模式.....	237
第十节	C 语言嵌入式系统应用设计模式的讨论.....	249

第一章 软件架构设计思想与体系创建

第一节 软件架构师的角色和应掌握的知识体系

一、软件架构

软件架构（software architecture）的一种定义是这样的：

架构是一组有关如下要素的重要决策：

软件系统的组织，构成系统的结构化元素，接口和它们相互协作的行为的选择，结构化元素和行为元素组合成粒度更大的子系统的的方式的选择，以及指导这一组织（元素及其接口、协作和组合方式）的架构风格的选择。

软件架构可以有多种定义，不管对软件架构如何定义，所有的定义都有一个共同的主题，那就是必须考虑诸如原理、组织、风格、模式、职责、协作、连接、系统的动机和主要子系统等大尺度方面的问题。

软件架构实际上是两个层面的事情，一个是设计构造一个完整的软件系统，这里的架构也称作软件体系结构（Software Architecture）。另一个层面是构造一个统一的共享的框架或者称架构（Framework），这种架构事实上是系统的一个基于服务的层。

软件架构在整个软件开发过程中，是处在软件体系结构设计阶段（设计），它的必要的输入，是来自需求工程（分析），而它的输出，是实现设计（编程），因此这是一个承上启下过程节点。

在软件开发中，架构既可以是名词，也可以是动词。

作为名词，架构包括上面所定义的内容。

作为动词，架构一部分是调研，一部分是设计，更清晰的，是架构调研和架构设计。

架构调研：

是指识别对系统存在或可能存在重大影响的功能性或非功能性需求（特别是非功能性需求），例如市场趋势、性能、成本、维护和系统演进等。广义上，是对系统的重大设计决策有特别影响的需求进行分析。

架构设计：

是对软件、硬件、网络、运营、政策等软件设计中的需求和要素进行决策。

在统一过程里面，架构调研和架构设计统称为架构分析。

软件架构设计是一个系统工程，它需要系统构架师有很宽的知识面，从需求分析、架构设计到类设计甚至代码实现都需要有透彻的理解，这之间的关系是你中有我我中有你，是不可能截然分开的。

在这个课程中，我会站在相对抽象的角度，对软件系统设计的思想和方法做一些讨论，这些观点，也是不少资深系统架构师经验的集合。必须说明，软件系统设计的方法不是一个僵化的规则，我表达的一些观点你也不一定赞成，这不要紧，关键是在实践中实事求是的摸索规律，从而

找出一些符合实际的方法来。

二、软件架构师的角色

尽管对软件架构师的角色有这样或那样的定义，但大体上下面几个职责是必需的。

- 1、技术负责，解决方案的提供者
- 2、与项目经理合作，制定计划，决定成员，组织团队
- 3、保证项目按计划走向完成

由于设计是由需求驱动的，所以，掌握需求分析的技巧，是一个好的架构师必备的能力。

三、软件架构师最难处理的问题

- 1、不是做什么，而是不做什么
- 2、不是从纯技术的角度来考虑整个项目
- 3、预见客户走向，早期决定技术研发
- 4、不能使用时髦但不可靠的技术

四、如何成长为一个好的系统架构师

架构师必须关注需求、分析需求，有人认为架构师只是在需求出来以后，把它的实现模型做出来就行了，真要是这样，那做一个架构师未免也太容易了。

事实上，现代迭代开发所有的驱动力都在于需求变更，如果架构师不关注需求，不关注和用户的讨论和沟通，那是很难设计出真正有用的东西来的。

软件架构设计是一个非常严肃、细致、敏感而且困难的工作，必须一点一滴认真做起，扎扎实实的努力，实实在在的积累经验，尤其是在失败中积累经验，这是一个软件架构师成功的必由之路。

为此，我们需要注意下面几点：

- 1、首先必须是一个好的程序员，技术上要强
- 2、知识结构：对象的观点，UML，RUP，设计模式
关键不是懂得了原理，而是灵活融合的应用
- 3、系统的观念：分析能力，把握抽象的能力
- 4、沟通能力：与客户沟通能力，与项目其它成员的沟通能力
- 5、知识面要广，把握行业流行趋势，但不要赶时髦
- 6、灵活机动，不能教条

五、几个观点

- 1、要承认软件是不完美的
- 2、要承认需求是不完全的
- 3、关键是拥抱变化而设计
- 4、各种性能标准，什么是架构师最关注的呢？
- 5、架构师最重要的素质：把握重点。

注意：灵活的把握，实事求是的分析，

善意和把握重点的沟通，
有先见性的设计，
这是一个优秀的系统构架师活的灵魂。

第二节 软件分析和设计的方法学问题

由于架构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架构思路。从方法学的角度来讲，目前分析和设计方法主要分为面向过程的方法与面向对象方法两种。

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的任务。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，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。

二、面向对象的方法

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的任务，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，逐步成长成为一种主要的和现代的分析与设计方式。

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。有人认为，学会了 UML 几张图或几个符号，就会对象方法了，这是个误会，UML 只是一个表达的工具，关键是在什么层面上去思考。

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。

一般来说，面向过程技术与面向对象技术是不能混用的，因为这两种设计技术基本思想是完全不同的，基本的原则也是不一样的，面向过程设计提供的是系统功能的体系结构，而面向对象技术是建立一系列交互对象的体系结构，思维不同，方式重点也必然不一样，这点是需要说清楚的。

在这个课程中，我们将沿着软件分析和设计的过程，重点研究面向对象的方法，同时与面向过程方法对比着，把思想方法和工作方法讨论清楚。

第三节 在信息技术战略规划（ITSP）中的软件架构

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，知识经济下，正确的结合 IT 规划，整合的核心竞争力，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

对企业管理信息化带来的管理模式变化不适应，又抵触心态，或者仅是为了形象问题，赶潮流搞信息化。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。

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。20 世纪末的电子商务热潮充分说明了这个问题。

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客户战略规划的需求，能够帮助客户解决实际问题的，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? What? 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，但为什么最终没有得到用户的广泛认可呢？一个专家感觉非常得意的东西，普通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时候，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？

很重要的问题是，设计的目的是为了生存，设计的源泉是来自于用户，满足用户的需求，能够帮助客户产生可度量的价值，又便于用户使用，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，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，用户也一定会喜欢。

其实这是个误会，人们购买软件产品，并不是购买它的技术本身，而是为了他的需求来购买，也就是说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。

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。

我们应该对市场和已有的产品作为模型来调查，搞清楚用户对产品的要求到底是什么？产品的设计应该来自于市场的调研，而不是对新技术的激情，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。典型的例子是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

给用户做一件事情，称为“有用的”；

如果一个功能是可以方便的使用的，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

泛泛地说体系结构设计的理念是没有用的，所以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，根本的本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。

1, 问题陈述

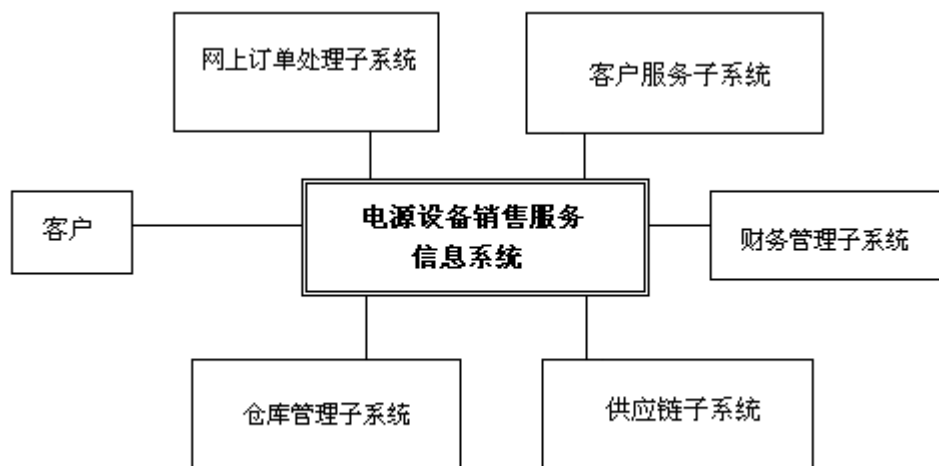
在研究这个陈述的时候,请体会这里是如何体现 **Who? What? Why?**这三个 W 的。

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	<p>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:</p> <ol style="list-style-type: none"> 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	<p>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。</p>
问题陈述	<p>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。</p> <p>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:</p> <ol style="list-style-type: none"> 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

	<p>易于变化特性。</p> <p>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。</p> <p>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。</p> <p>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。</p> <p>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。</p> <p>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。</p> <p>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。</p> <p>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。</p>
项目影响范围	<p>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:</p> <ol style="list-style-type: none"> 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 <ol style="list-style-type: none"> a, 潜在客户 b, 客户 c, 曾经的客户 d, 供应商 e, 生产厂 <p>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。</p>
项目构想	<p>信息技术战略规划要求该系统必须做到:</p> <ol style="list-style-type: none"> 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。

	<p>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。</p> <p>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。</p> <p>7, 提供改进的订单和应急配备设备客户的跟踪机制。</p>
业务限制	<p>新系统必须符合以下要求:</p> <p>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。</p> <p>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。</p> <p>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。</p> <p>4, 系统必须具备很好的可维护、可升级以及安全性。</p>
技术限制	<p>新系统必须符合下面信息技术架构标准:</p> <p>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)。</p> <p>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。</p> <p>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。</p> <p>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。</p> <p>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)。</p>
项目组织方式	<p>为此项目成立专门的开发管理班子, 职责为:</p> <p>1, 任命项目经理</p> <p>2, 任命首席构架师和分析师</p> <p>3, 任命项目经理推荐的项目团队</p> <p>4, 评审并批准项目发布的内容</p> <p>5, 确保项目符合管理层的想法</p> <p>6, 认可所有的范围、预算和计划变更</p> <p>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。</p> <p>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。</p>

2, 子系统组成



下面只列出了本课程作为案例的子系统说明。

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, 有供应链子系统获取供货信息。	使用桌面程序。

第四节 软件生命周期设计与统一软件开发过程

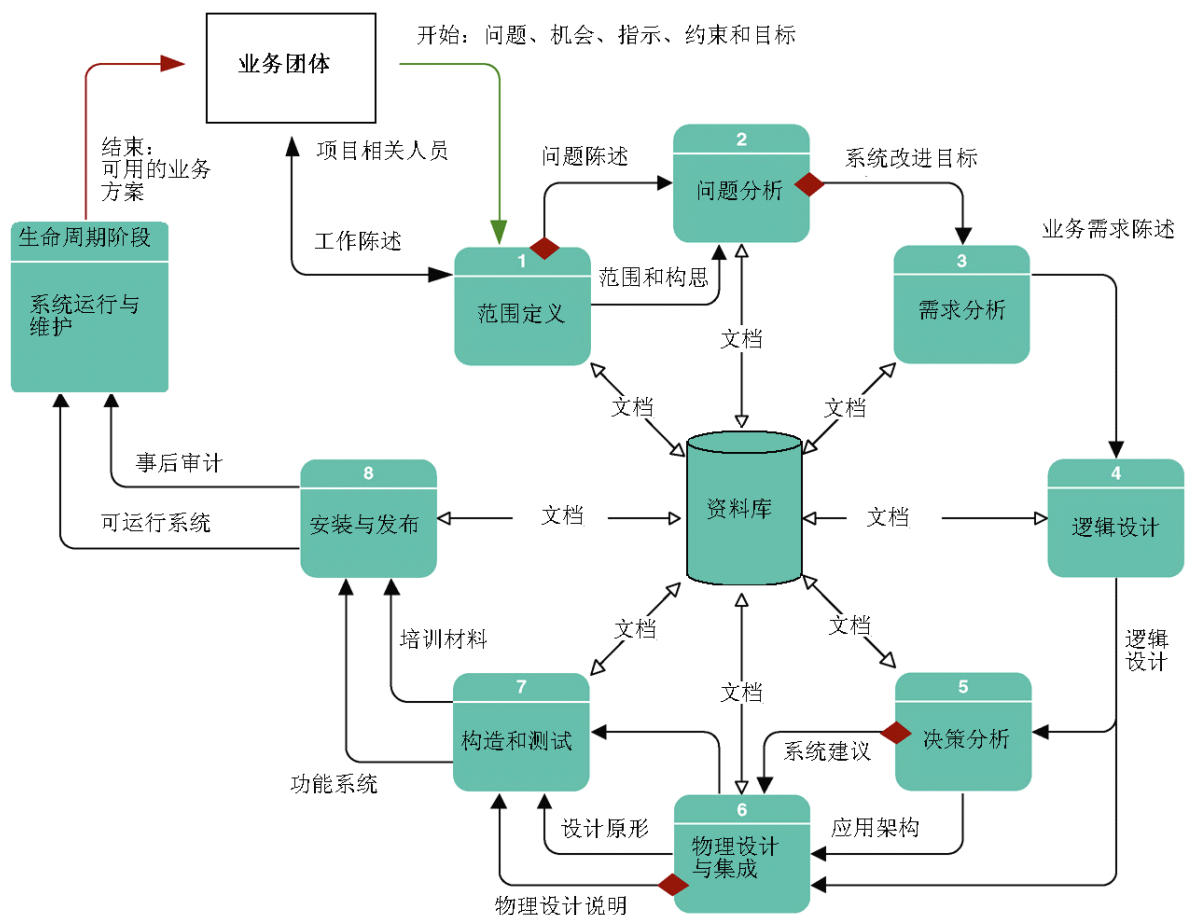
一、软件设计的生命周期与基本过程

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。

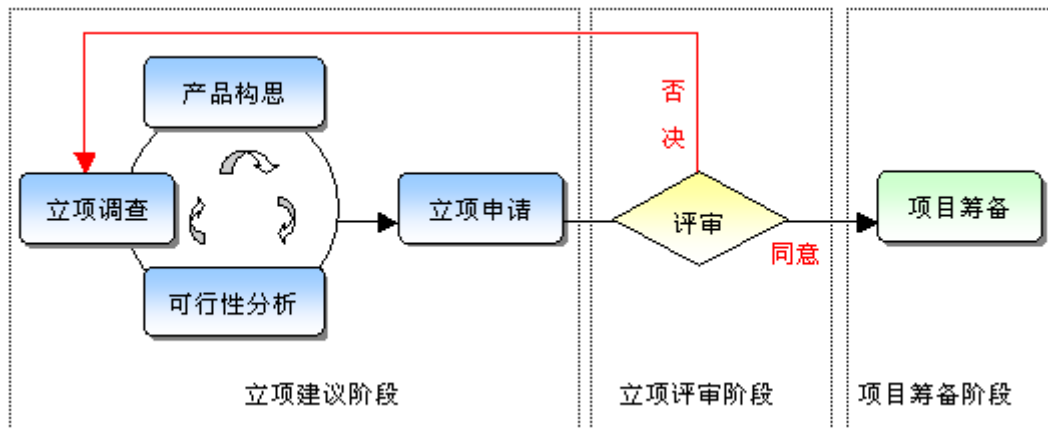
不成功的项目大多具有以下特点：

- 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。

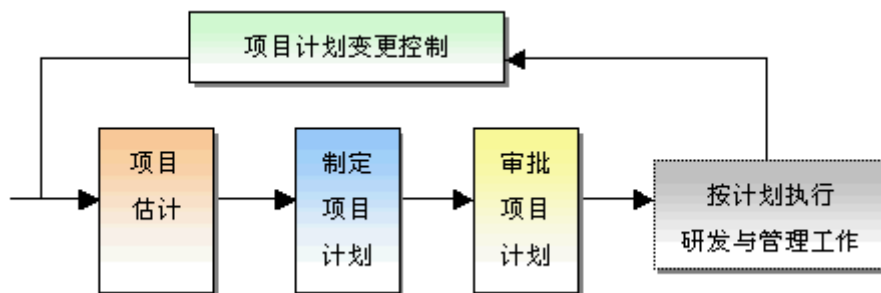
成功的项目，当从研究阶段到生产阶段的转变的时候，有十分明确的项目里程碑。
较前期的阶段，侧重于功能的实现，较后期的阶段，侧重于如和实现交付给客户的产品。

由此形成了如下过程。

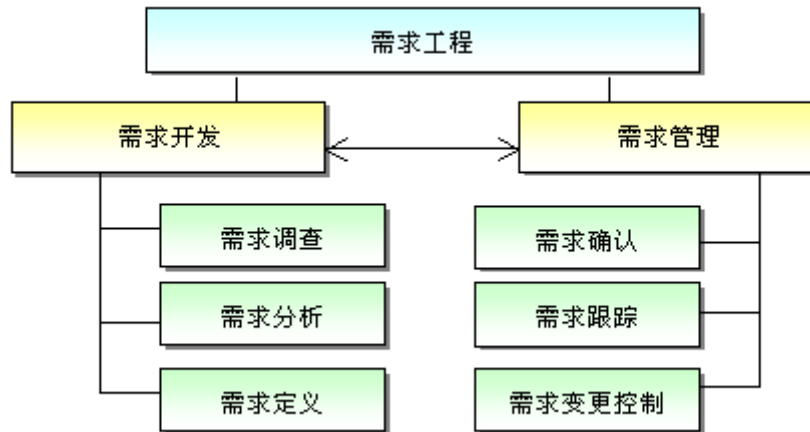
1，立项管理流程



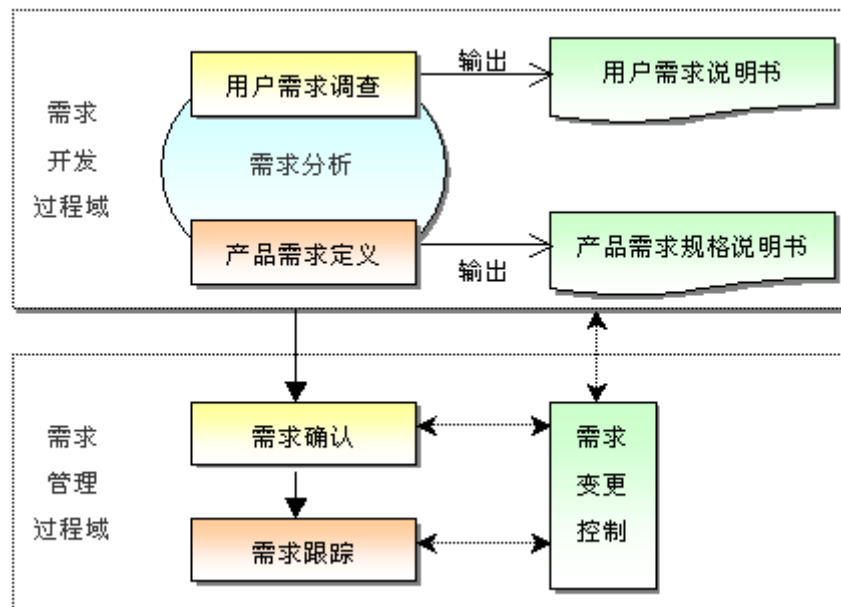
2，项目规划流程



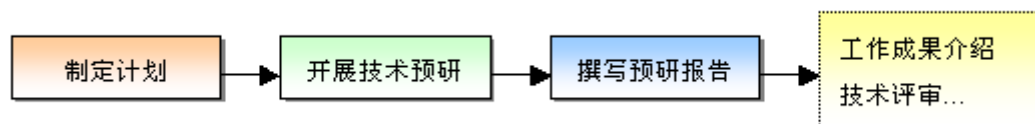
3，需求管理流程



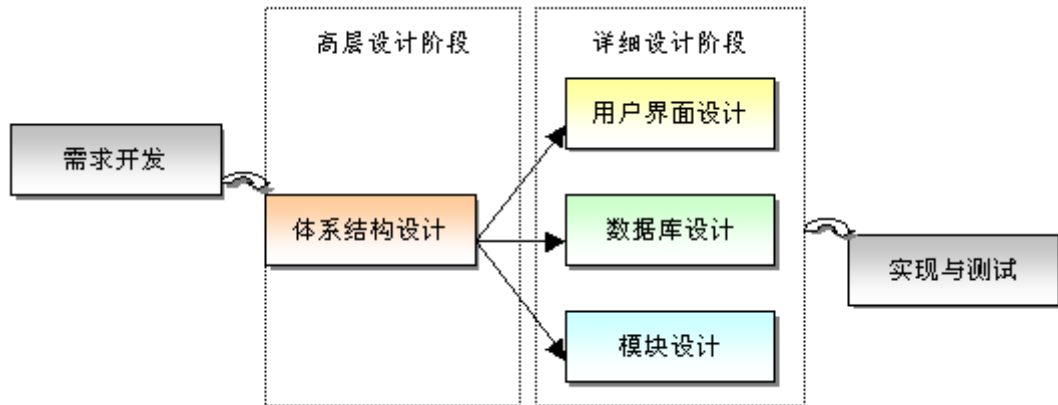
4，需求开发流程



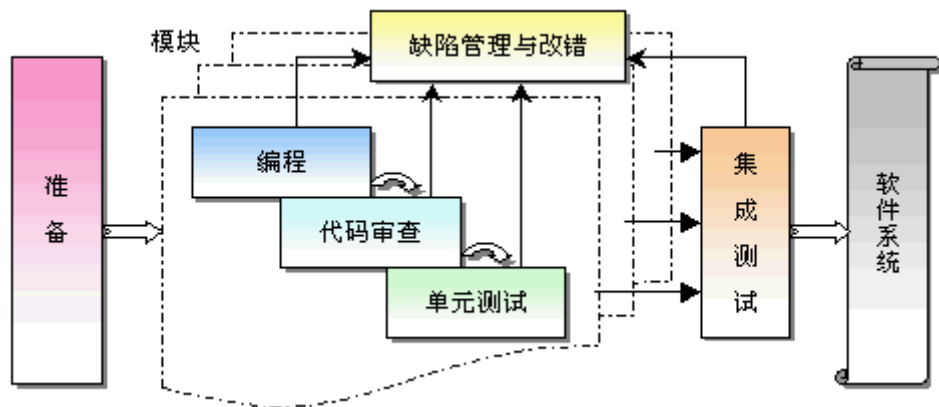
5，技术预研流程



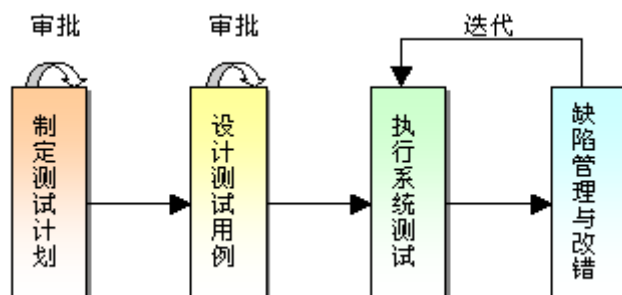
6，系统设计流程



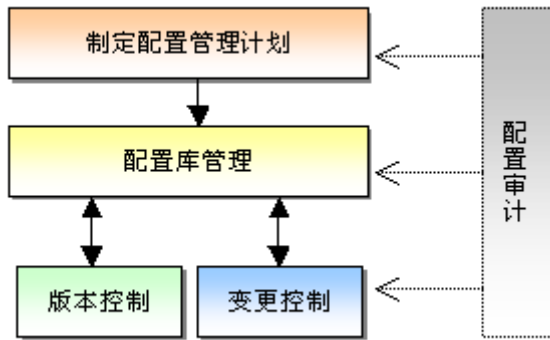
7, 实现与测试流程



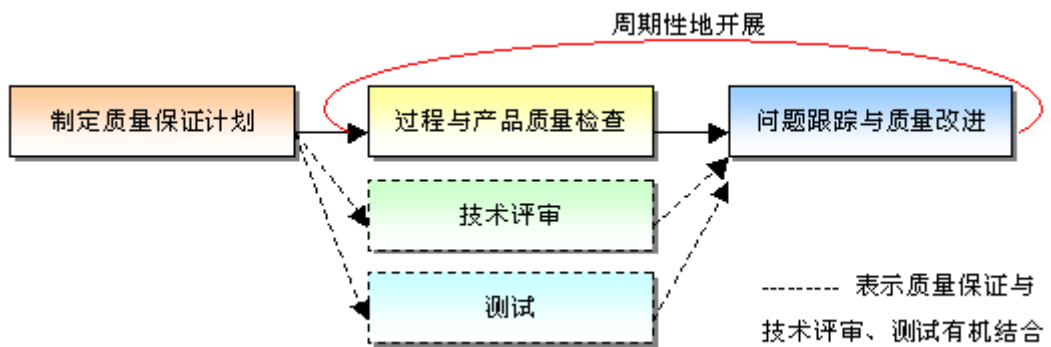
8, 系统测试流程



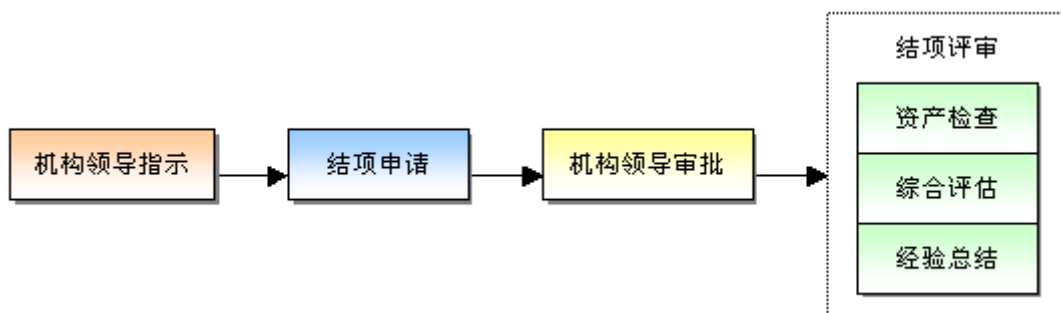
9, 配置管理流程



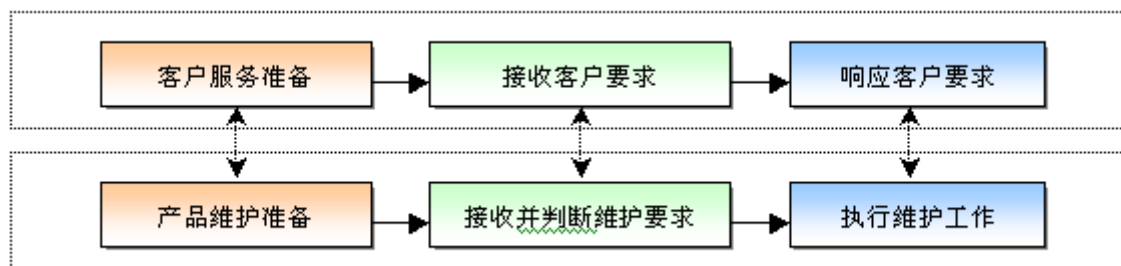
10, 质量保证流程



11, 结项管理流程



12, 服务与维护流程



二、顺序“瀑布”生命周期所带来的问题

经典的软件开发生命周期是顺序的、线性的或“瀑布”的生命周期，最常见的步骤如下：

- 1) 澄清、记录和承诺一组完整的已经冻结的需求。
- 2) 设计基于这个需求的一组系统。
- 3) 基于设计，实现系统。

但是，这样的开发过程往往带来了很多问题，这些问题促使了统一软件开发过程（UP）的提出，这是一种流行的构造面向对象系统的软件开发过程，并已经被广泛采纳。

统一过程（UP）把普遍接受的最佳实践（迭代生命周期和风险驱动开发），合并为内聚和具有良好文档的过程描述。而迭代开发是在统一过程中最有价值的实践，它是软件开发的一种巧妙的方法。

MIT Sloan Management Review（麻省理工学院项目管理评论）所刊载的一个为时两年对成功软件项目的研究报告指出，软件项目获得成功的共同因素，排在首位的是迭代开发，而不是瀑布过程。

（其它的因素是：

- 2，至少每天把新代码合并到整个系统，并且通过测试，对设计变更做出快速反应。
- 3，开发团队具备运作多个产品的工作经验。
- 4，很早就致力于构建和提供内聚的架构。

这四个因素中有三个在 UPO 中有显式的实践。）

为什么会是有这样的结果呢？

三、迭代开发希望能解决的问题

需求变更对项目过程的影响

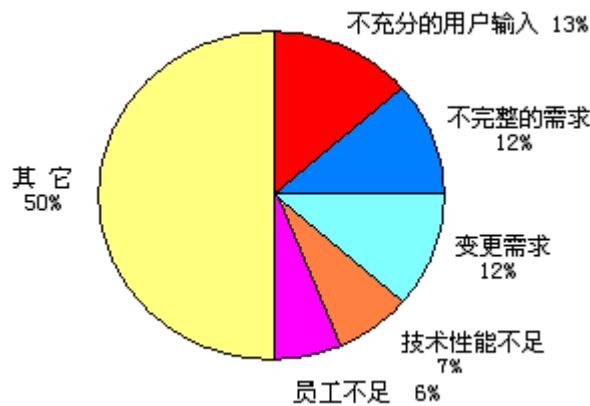
所谓需求，是系统必须提供的能力和必须遵从的条件。

需求最根本的挑战在于：

寻找、交流并记录什么是真正需要的，并能够向用户和开发团队讲解。

一个关于影响项目进展的因素的研究如下图。

（Jim Johnson 1994 Chaos:Charting the Seas of Information Technology. Published Report .The Standish Group）



可见 37% 的问题都与需求有关，这就需要“需求管理”。

早期需求管理是瀑布式的，也就是说在项目的第一阶段，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，并且实际开发前冻结需求，但历史证明这种方式是失败的，在项目很大的时候，冻结需求几乎没有可能。

正是这种困难，迫使人们在项目过程控制中，转而使用迭代式方法，所谓迭代式方法，是用一种条理化的方法来寻找、记录、组织和跟踪不断变化的需求。

这里的关键是“不断变化”这个词，把需求变化作为项目进展的不断驱动力，另一个重要的词是“寻找”，可以通过写出用例和召开需求研讨会等手段巧妙的得出需求。

整个问题都来自于，用户往往对自己的愿望并不清晰，需求会不断的变化。

所以，我们必须使用一种处理过程，这种处理过程的特点就在于适应这种需求的变化。

很多单位为什么放弃了多年使用面向过程方法，转而研究面向对象的方法，需求变化的困扰是一个主要原因，而面向对象的设计和分析，所有问题的焦点都聚焦于如何适应变化上，这是一个值得我们关注的课题。

统一过程（UP）提倡了几种最佳实践，其中影响力最大的实践，那就是迭代开发。

什么是迭代开发？

- 1) 开发被组织成一系列固定的短期（如四个星期）小项目，称为迭代。
- 2) 每次迭代都产生经过测试的，经过集成的，和可执行的系统。
- 3) 每次迭代都有自己的需求分析、设计、实现和测试活动。

迭代生命周期贯穿多个迭代，系统在这个过程中被持续扩展和精化。

系统迭代过程的驱动力有两个：

循环反馈；

适应调整。

随着一次又一次的迭代递进，系统增量式的发展完善，因此这一过程也称之为迭代增量开发。

我们来看一个简单的实例：

讨论一个两周的迭代。

星期一：分析和澄清本次迭代的任务和需求，同时由一人对上次迭代的代码进行逆向工程，形成 UML 并打印出重要的图。

星期二：在白板上进行分组设计工作，画出 UML 草图，使用数码相机记录，并写出一些伪代码和设计注释。

剩余的八天：

实现；

测试（单元测试、验收测试、可用性测试）；

进一步设计；

集成；

每日的构造工作；

系统测试及部分系统的稳定；

与项目相关人员进行演示和评估；

计划下一步迭代。

注意：

- 1) 不要匆忙编码。
- 2) 不要试图在编程前完善所有设计细节的长期持续的设计步骤。
- 3) 少量超前设计使用粗略和快速的 UML 可视建模来完成。
- 4) 开发人员大概用半天或一整天的时间进行分组的设计工作。

每次迭代的结果是一个可执行但不完善的系统，一般需要 10 到 15 次迭代系统才可能投入生产。

迭代输出不是实验性的，也不是即将丢弃的原型，迭代开发也不是原型开发。与之相反，它的输出是最终产品的子集。

一般来说，尽管每次迭代需要扩展新需求并增量扩展系统，但迭代偶尔也会重新审视现有的软件并对它进行改造，例如，并不增加一个子系统的新特性，而致力于提高它的性能。

四、初始阶段

大多数项目需要一个简短的初始阶段来考虑以下几种问题：

- 1) 项目的构想怎么样？业务的案例是什么？
- 2) 可行性怎么样？
- 3) 购买还是开发？
- 4) 粗略估计一下成本：一万到十万？还是上百万？
- 5) 项目是否停止或者继续进行？

想要定义构想和获得一个粗略的（不一定可靠）的预测结果，就必须研究需求。

但是，初始阶段的目标并不是要定义所有的需求，或产生一个真实可信的项目估计或计划。

简而言之，初始阶段的目标：

只进行一定的研究，得到未来新系统的可行性以及实现系统总体目标的合理判断，并确定是否值得继续深入研究系统即可。

深入研究是细化阶段的工作。

初始阶段的时间：

大多是一周到几周，太长就失去了初始的意义。

注意：重要的问题是，确定这个项目是不是值得深入的去研究。

在统一过程中，真正的项目研究在细化中。

下面列出初始阶段的工件以及各个工件需要完成的工作：

- 1) 构想和业务案例：
描述高层的目标和约束、业务案例、并提供一个执行摘要。
- 2) 用例模型
描述功能需求和非功能需求。
- 3) 补充规范
描述其它需求。
- 4) 术语表
关键的领域术语。
- 5) 风险列表和风险管理计划
描述业务、技术、资源和进度上的风险，以及如何减轻这些风险，以及如何应对。
- 6) 原型个概念验证
阐明构想，验证技术问题。
- 8) 迭代计划
描述第一此细化迭代中应该做些什么。
- 9) 阶段计划和软件开发计划
对细化阶段的持续时间和工作量进行抵精度的猜测，开发涉及的工具、人员、培训和其他资源。
- 10) 开发案例
描述为本项目定制的统一过程的步骤和工作，在统一过程中，总需要为项目定制一些步骤和工作。

在上面的工作中，需要我们关注的是所谓“用例模型”。

迭代开发的一个关键理解就是：

这些工件在初始阶段只是部分完成，在之后的迭代中再逐步精化和提炼，甚至在认定某个工件确实有用之前，根本无需来创建它，所以初始阶段进行的研究和工件内容都很少。

注意：

初始阶段会有一些编程，其目的是创建“概念验证”，通过面向用户的界面原型，来澄清一些需求，或者对关键的技术问题作一些编程实验。

初始阶段关注的是确实对项目有价值的工作，要放弃那些不必要的工作。工件的关键不是文档本身，而是其中蕴含的思想、分析和前期准备。

五、精化阶段，反馈和调整

注意：

迭代开发不是试图在实现以前完成所有的设计，“冻结”所有需求的变更。

而是用拥抱变更、适应调整的态度来作为迭代开发真正必要的驱动力。

但这并不表明迭代开发和 UP 鼓励不受控制的、反应式的“特性蔓延”驱动过程，事先，我们必须和相关人员认真探讨他的构想和需求，以及市场变化的时候，UP 如何平衡需求。另一方

面，我们也需要认清需求会变更这个事实。

每次迭代都选择一小组需求，并且快速的设计、实现和测试，在早期迭代中对需求和设计的选择可能并不准确，也可能不是最终的需求，但这样可以迅速得到来自用户、开发人员或者测试人员的反馈。

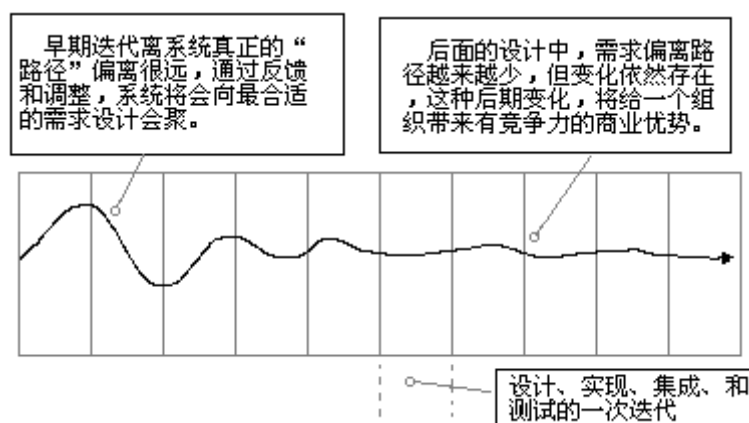
这种早期的反馈十分重要，这种来自实际构建和测试的反馈，提供了实际的洞察力，以及修改或者调整需求或设计的机会。

最终用户将有机会看到部分系统，并说：“不错.....但是.....”。

这种“不错.....但是.....”的循环，正是改进软件或者获得相关人员认可的巧妙方式，不过，注意这里并不是认可混乱的反应式开发。

另外，负载测试也可以验证部分设计是不是正确，这可以决定下次迭代是不是要改变核心架构，最好在早期解决和验证关键设计，以规避决策风险，迭代开发正好提供了这种机制。

因此，通过一系列有序的“构造-反馈-调整”循环，工作不断向前推进，早期迭代偏离“正确轨迹”会大于后来的迭代，随着时间的推移，系统将沿着这一轨迹推移。



六、迭代开发的优点

在早期而不是晚期缓解高风险（技术、需求、目标、可用性方面的风险）；

早期可见的发展；

早期反馈、用户参与和调整，会得到一个更接近用户真实需求的经过精化的系统；

可控复杂性，开发组不会被“分析瘫痪”或者长期复杂过程所淹没；

在一次迭代中所学到的知识，可以被有系统的运用于改进开发过程本身。

七、迭代的长度和时间分区

UP（或有经验的迭代开发人员）建议，迭代长度在 2-6 周之间比较好。

小步骤、快速反馈和调整，是迭代开发的主要思想。

长期迭代复杂性会变得不可收拾，反馈也会延迟，会增加项目风险。

短于两周的的迭代不足以产出和反馈。

一个关键思想：

时间分区是固定的，每次迭代不鼓励时间推迟，如果无法完成，可以除去一部分任务转移到下一次迭代中去。

大型开发团队（几百人）可能需要 6 周以上的迭代时间。

例：

20 世纪 90 年代，加拿大空中交通系统。

首席架构师：Philippe Kruchten

150 个程序员被组织到 6 个月的迭代中。

但是，即使在 6 个月的迭代中，10 – 20 人的子系统，仍然把任务分解成一连串 6 个为其一个月的迭代。

6 个月迭代是大型开发团队的特例，但不是准则。

UP 的建议普通的迭代应该在 2 – 6 周之间。

八、统一软件开发过程最佳实践和概念

UP 所欣赏和实践的主要思想是：短时间分区式的迭代和适应性开发。

UP 的另一个核心思想是使用对象技术。

其它一些 UP 的关键概念是：

在早期迭代中解决高风险和高价值的问题；

不断的让用户参与评估、反馈和需求；

在早期的迭代中建立内聚的核心架构；

不断的验证质量，提早、经常和实际的测试；

应用用例；

可视化建模（UML）；

仔细的管理需求；

实行变更请求和配置管理。

九、统一过程阶段和面向进度表的术语

一个 UP 项目跨越 4 个主要阶段：

1) 初始阶段：

大体构想、业务案例、范围、模糊评估。

2) 细化阶段：

已经精化的构想，核心架构的迭代实现，高风险的解决，大多数需求和范围的识别，更为现实的评估。

3) 构造阶段：

迭代实现遗留下来的风险较低的和比较容易的元素，准备部署。

4) 移交阶段：

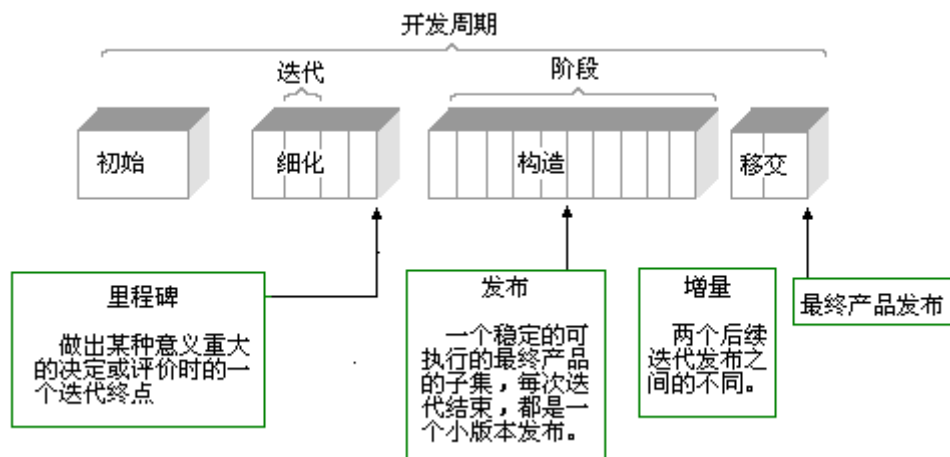
beta 测试，部署。

UP 和过去“瀑布”或顺序生命周期不同，它不是一开始就定义全部需求，然后进行全部或大部分设计。

初始阶段不是一个需求阶段，而是一个类似于可行性阶段，在这个时候需要进行充分的调查，以确定是否继续或终止项目。

同样，细化阶段也不是一个需求或设计阶段，而是一个迭代实现核心架构并降低高风险的阶段。

下图是 UP 中常用的面向进度表的术语，注意，一个开发周期（以系统投运作为结束标志）由多个迭代组成。



十、UP 流程（ workflow ）

UP 描述了流程（discipline），简短的说，流程是在一个主题域中的一组活动，例如需求分析中的活动。

工件（artifact），是对任何工作产品的通用术语，比如：代码，Web 图形，数据库模式，文本文档，图，模型等。

需要说明，在 2001 年，为了与 OMG SPEN 的国际化工作一致，过去的 UP 术语“工作流”（workflow）改成了流程（discipline），术语“工作流”（workflow）有新的含义，并且和 UP 的含义稍有不同，在一个特定的项目中，工作流是特定顺序的一组活动，可能跨越流程（工作的流转），但现在在 UP 中还有人使用“工作流”这个名词，尽管这并不严格正确。

Up 中有多个流程，但下面我们将专注于以下三个流程的建模。

1) 业务建模

在开发单独的应用的时候，业务建模包括领域对象建模。

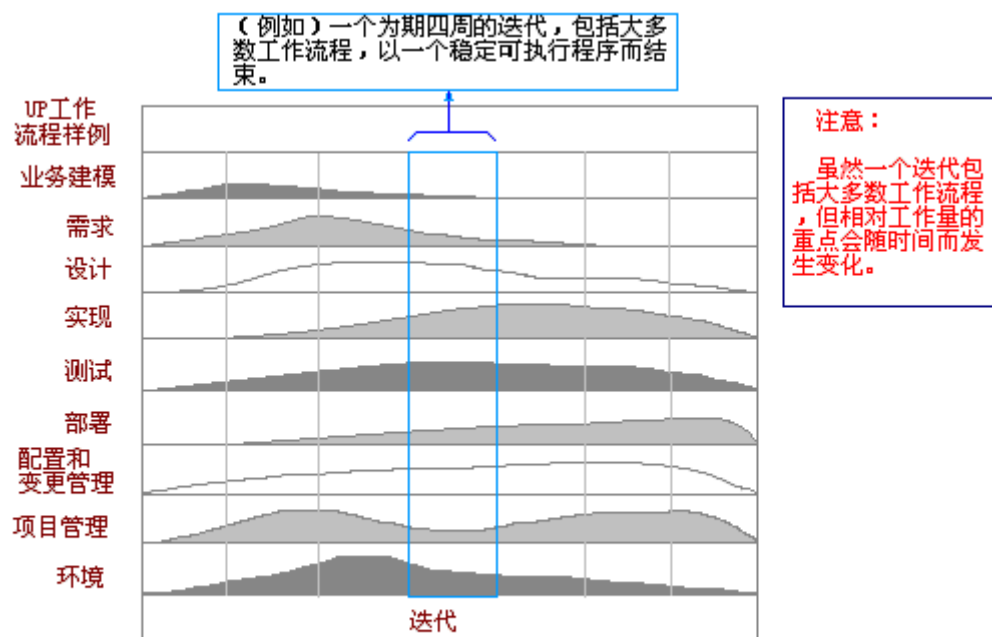
在从事大规模业务分析和业务过程再工程的时候，业务建模包括跨越整个企业的业务过程的动态建模。

2) 需求

对应的需求分析，比如写出用例，和识别非功能性需求。

3) 设计

设计的所有方面，包括总体框架、对象、数据库、网络连接等。
下图列出了更多的 UP 流程。



在图中：

实现：表示编程的构建系统，而不是部署。

环境：实质建立工具，并为项目定制过程，也就是说设置工具和过程环境。

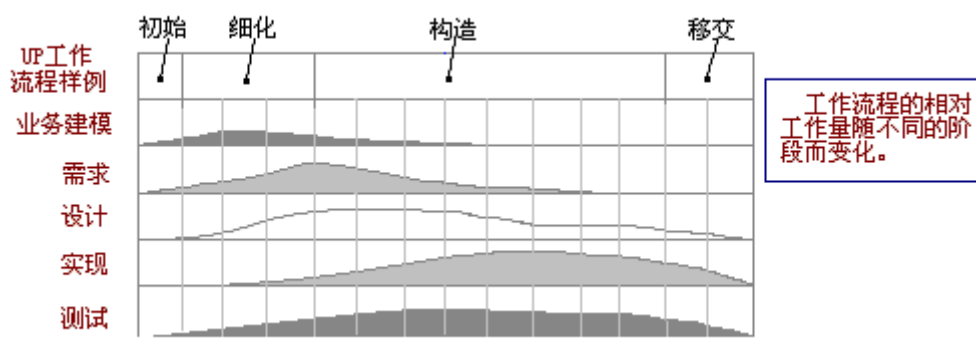
十一、流程和阶段

在上面的图中，在一次迭代中，工作会在大部分流程或全部流程中进行。但随着时间变化，这些流程的相对工作量会变化。

早期迭代更多的工作量在需求分析和设计。

后期迭代这种需求的变化会减少，表明需求和核心设计通过反馈和适应调整过程已经稳定。

对于 UP 阶段（初始、细化、构造、移交），各阶段相对工作量如下图。



比如，在细化阶段，迭代趋向于相对高层的需求和设计工作，尽管同时有一些实现。在构造阶段，工作的重点更多的是放在实现而非需求分析上。

十二、敏捷 UP

方法论者谈起过程是这样来区分的：

重量级过程和轻量级过程；

预测性过程和适应性过程。

重量级过程（heavy process）是一个贬义词，它的含义如下：

- 1) 在官僚气氛中创建工作。
- 2) 刻板和忧郁。
- 3) 细化的、长期的、详细的计划。
- 4) 预测的而不是适应性的。

预测性过程（predictive process）：

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。

预测性过程通常具有“瀑布”或顺序生命周期的特点：

- 1) 定义所有需求。
- 2) 定义详细的设计。
- 3) 实现。

适应性过程（adaptive process）：

认为变化是不可避免的驱动因素，鼓励灵活的改写。

它通常具有迭代生命周期。

敏捷过程（agile process）：

通常意味着轻量级和适应性过程，敏捷的反映不断变化的需要。

UP 的作者不想让 UP 成为重量级和预测性过程，尽管大量可选的工件和活动集中在这个方面导致这种印象，敏捷 UP 由以下的应用示例：

- 1) 推荐一小组 UP 活动和工件：

一般来说，应该尽可能保持简单。

- 2) UP 是一个迭代过程：

需求和设计实现之间并不是完全的，它们是基于反馈，通过一系列的可适应迭代完成。

- 3) 对整个项目没有详细计划：

项目的一个高层计划（阶段计划）确立项目完成日期和其它主要里程碑，但它没有详细描述这些里程碑的细粒度步骤。

详细计划（迭代计划）只是预先对迭代进行的更详细的计划，详细计划从迭代到迭代是可适应的。

强调相对小量工件和迭代开发，是敏捷 UP 的精髓。

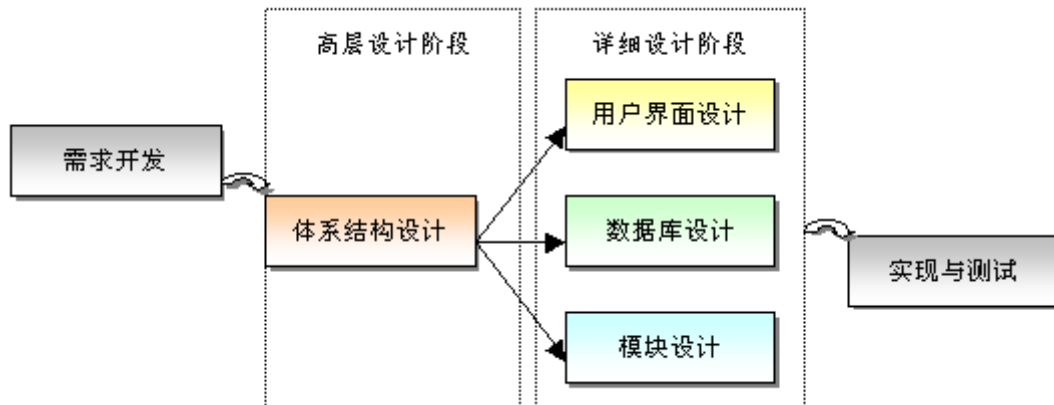
十三、何时你会知道你并不了解 UP

你如果是用如下方式之一处理问题，就表明你并不了解 UP。

- 1) 认为初始=需求，细化=设计，构造=实现。
- 2) 为细化的目的，是完整的细致的定义模型，这些模型在构造过程中会转化为代码。
- 3) 试图在开始设计或实现之前定义绝大部分需求。
- 4) 试图在开始实现之前定义绝大多数设计，试图在迭代编程和测试之前完整的定义和确认架构。
- 5) 在编程开始之前进行“长期”的需求和设计工作。
- 6) 认为合适的迭代长度是 4 个月，而不是四个星期（除非上百人开发一个项目）。
- 7) 认为 UML 绘图是完整而且详细的定义设计和模型，认为编程就是把这些设计图简单而机械的转变成代码。
- 8) 认为采用 UP 意味着实行大量可能的活动和创建许多文档，认为 UP 是一种形式化的繁琐的过程，这个过程有许多要遵守的步骤。
- 9) 试图从头到尾详细计划一个项目，试图投机的预测所有迭代和每次迭代可能发生的事情。
- 10) 在细化阶段结束之前，想要可信的项目计划和评估。

第二章 需求过程与分析的核心理论

架构设计过程分为**两个阶段**：高层设计阶段和详细设计阶段。



高层设计阶段的重点是软件系统的体系结构设计。详细设计阶段的重点是用户界面设计、数据库设计和模块设计。

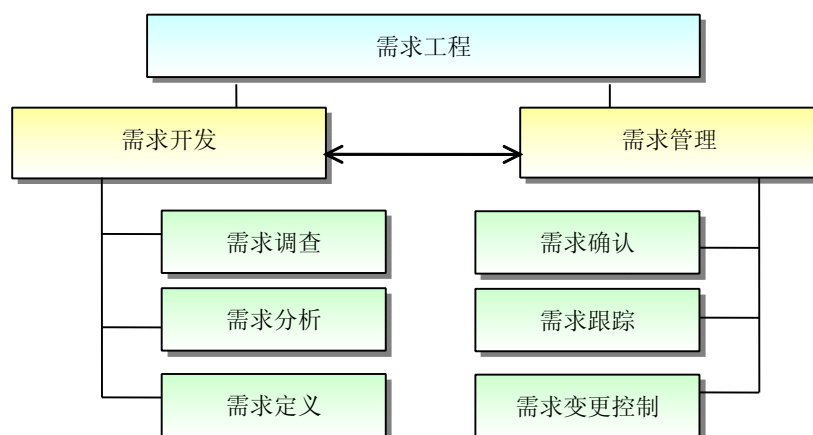
而高层设计的信息，主要来自于需求分析。

第一节 需求过程在软件架构中的重要作用

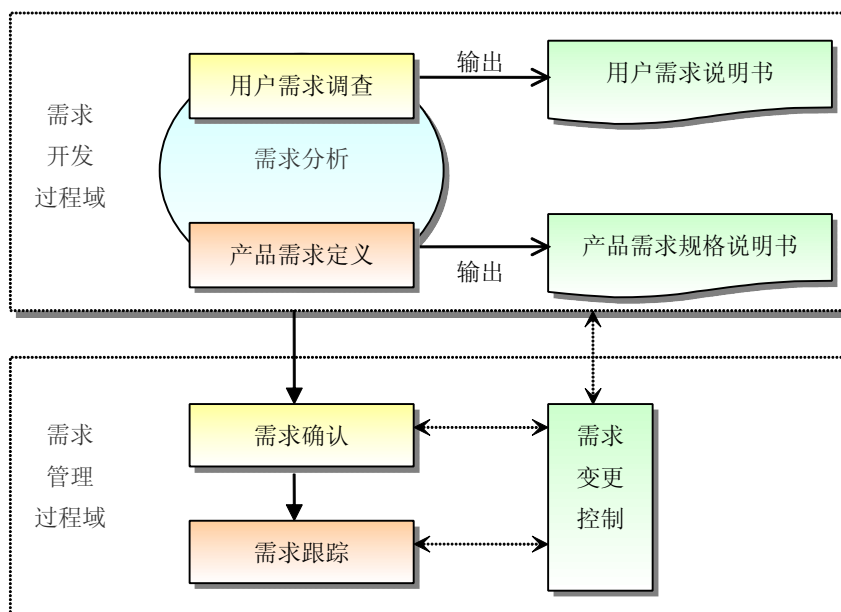
作为一个架构师，工作的主要舞台是系统设计，但设计的输入来自于需求工程，什么样的需求思想，就有什么样的架构思维。

这就是说，合理而且正确的需求分析过程，是架构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。需求工程结构图如下所示：



需求开发和需求管理的流程下所示。



其中，需求变更控制是指依据“变更申请—审批—更改—重新确认”的流程处理需求的变更，确保需求的变更不会失去控制而导致项目发生混乱。

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 (Implementation): 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)。

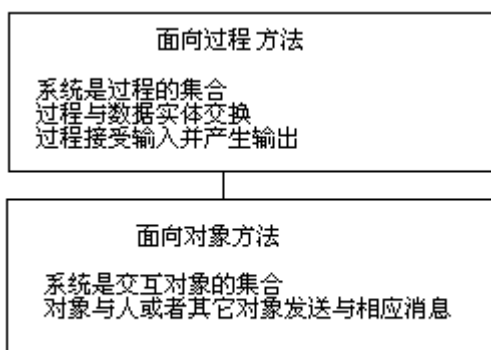
第二节 面向过程的需求分析核心知识

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）。

一、数据流程图 DFD

1, DFD 的符号

面向过程的理念是数据流，所以它的模型主要是数据流程图（DFD），它只用了 5 个符号。

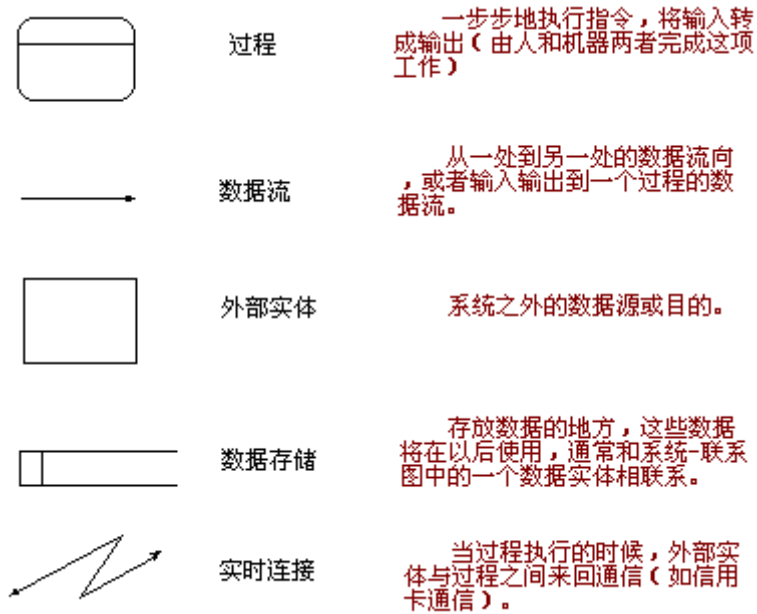
概念：

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。

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。

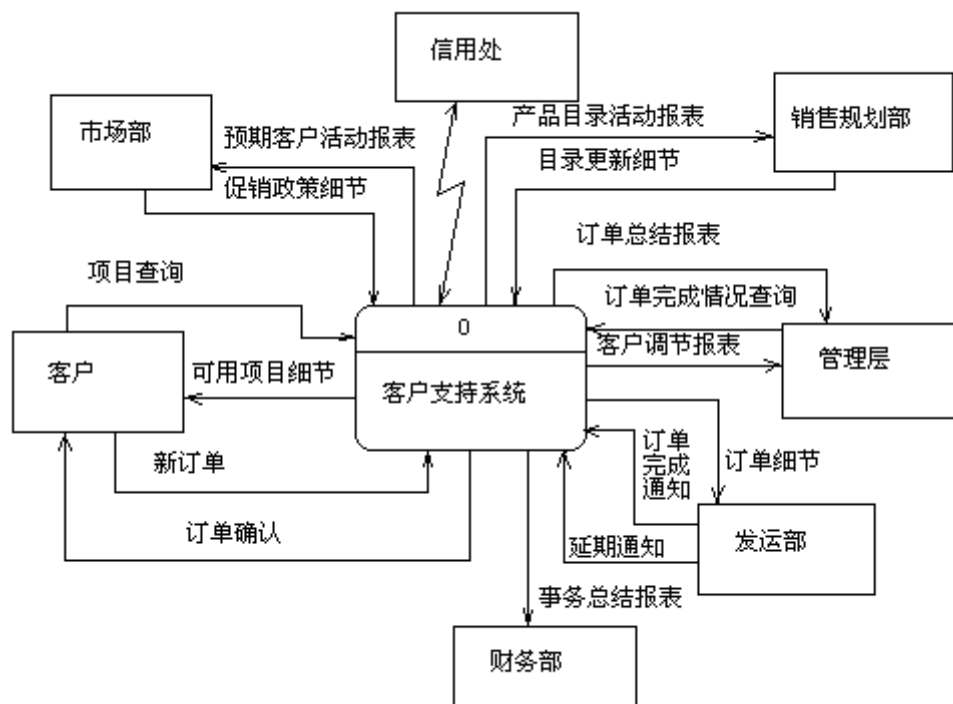


2. 关联图

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关联图简单例子：

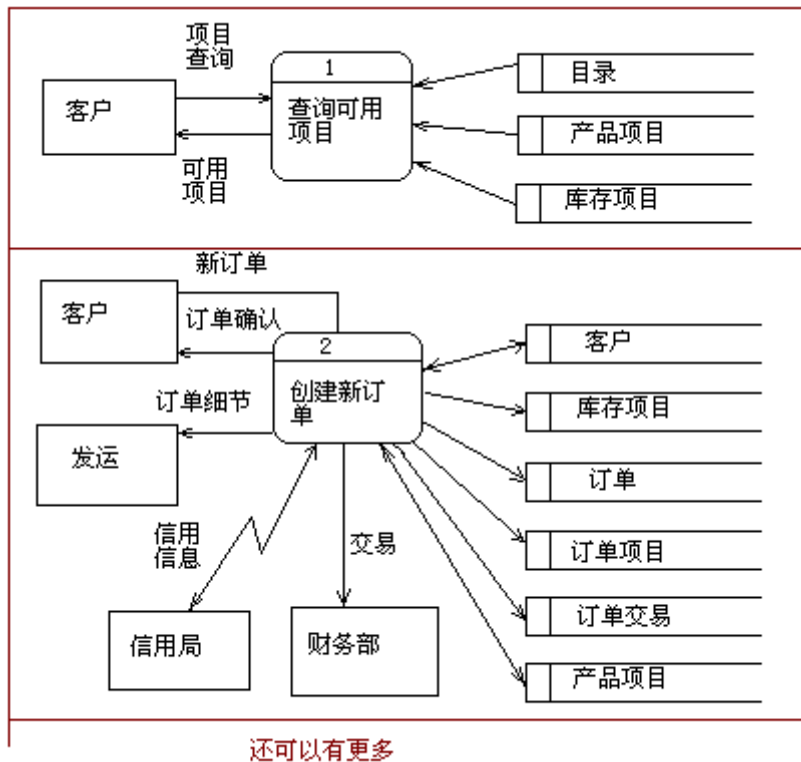
注意，箭头表示数据的流向。



二、事件划分的系统模型（0 层图）

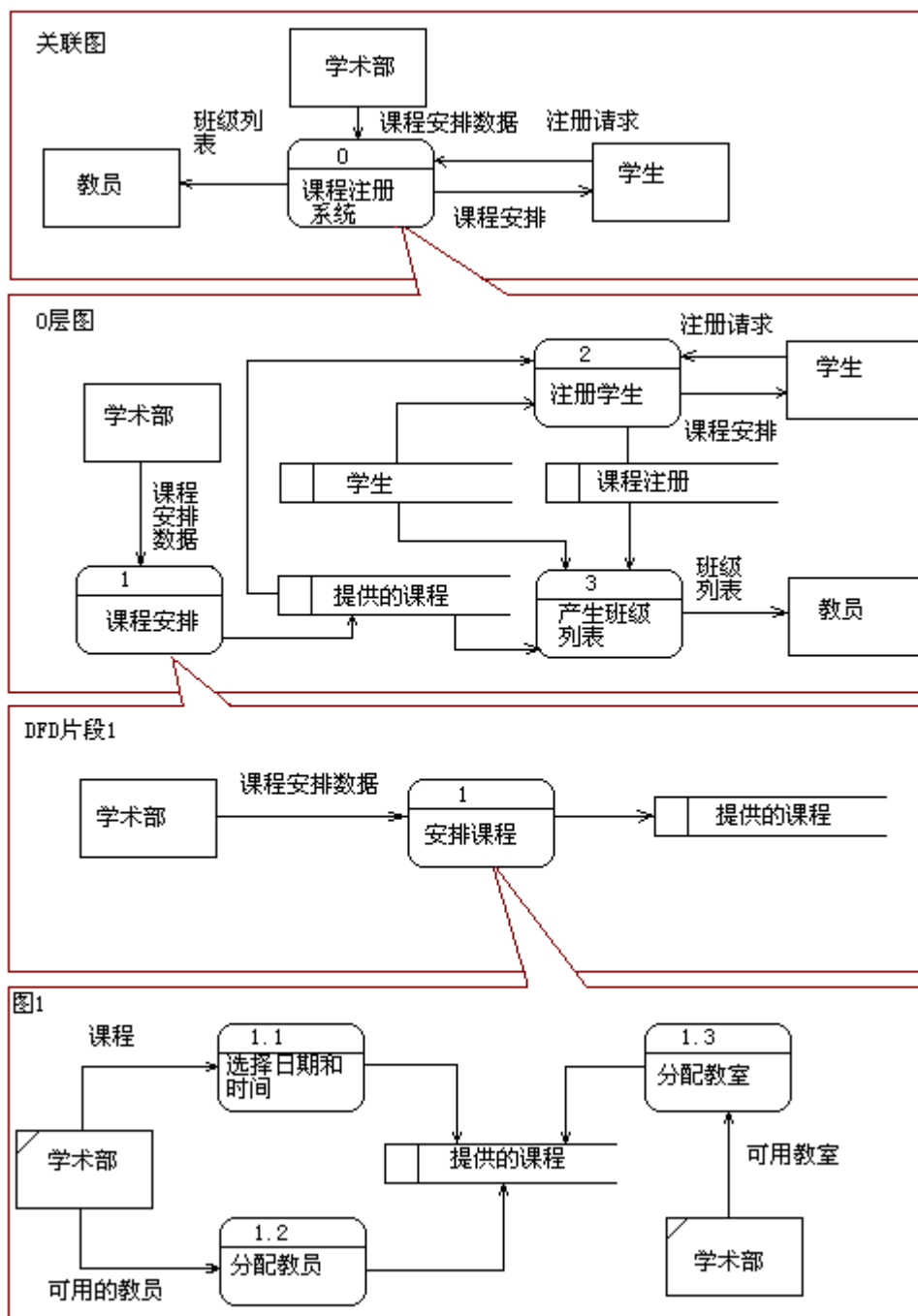
DFD 的细节称作片段，片段的组合有多种方式，现代过程分析也是以事件为基础，所以完

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。



三、分解过程

如果一个 DFD 片断包括更多的处理，可以把过程进行分解，以便作更详细的研究。



四、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。

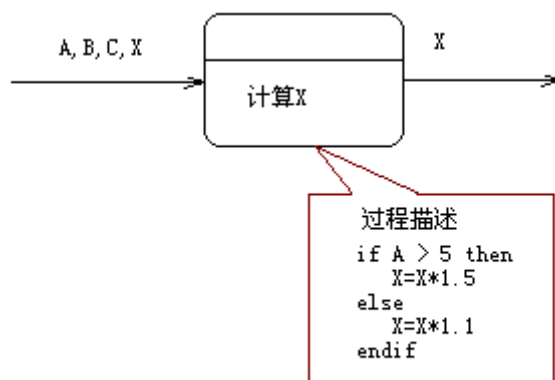
接口最小化：

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。

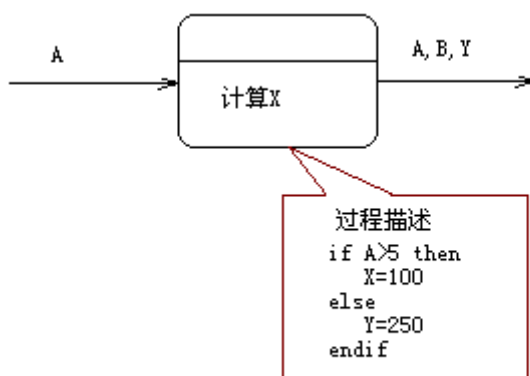
数据流一致性：

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”。



面向过程的分析已经有了一套完整而且成熟的方法，像决策表和决策树等等，这里就不再讨论了。

五、案例：订单处理子系统

1、问题陈述：

项目名称： 电源设备订单处理子系统 项目单位： TB 公司电源设备销售部 最后修改日期： ****年**月**日	
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。

1, 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。



2, 过程事件图

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, 这实际上是一个事件的上下文流图。在研究事件图的时候, 往往需要了街所有的数据存储。必要的时候, 数据库分析和设计可以提到前面先来完成, 我们将在后面的章节来讨论这个问题。要说明的是分析的时候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:

- 1) 输入及输入来源, 来源被描述为外部代理。
- 2) 输出及输出目的地, 目的地被描述为外部代理。

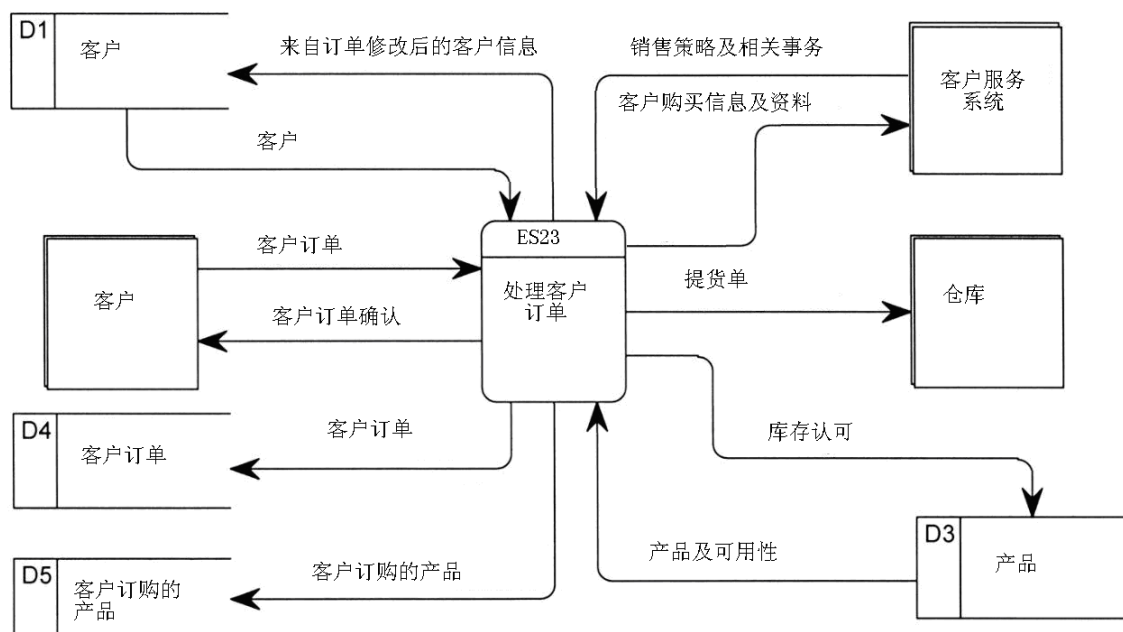
3) 必须读取记录的任何数据存储都应该被加入到事件图中, 事件流应该加入命名。

4) 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, 下面是一个简单的外部事件图。

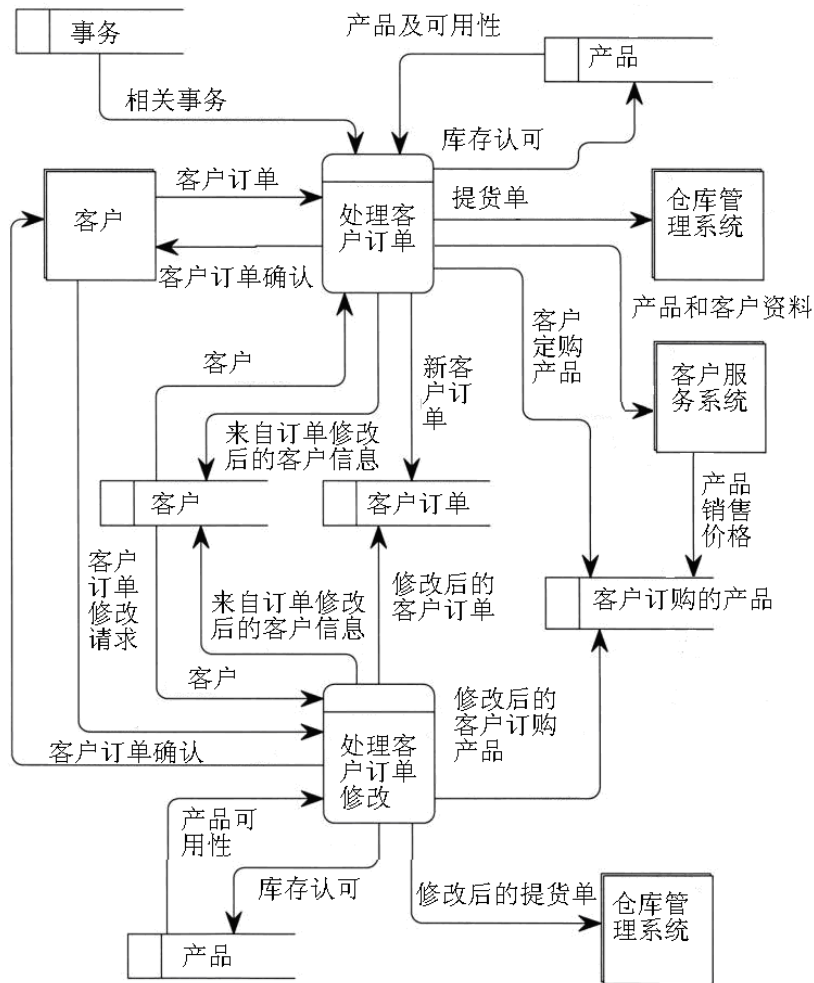
一个简化的“订单处理子系统”的过程事件图如下。

参与者	事件 (或者用例)	触发器	响应
客户	选择产品 (由 Web 页面驱动)	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”。



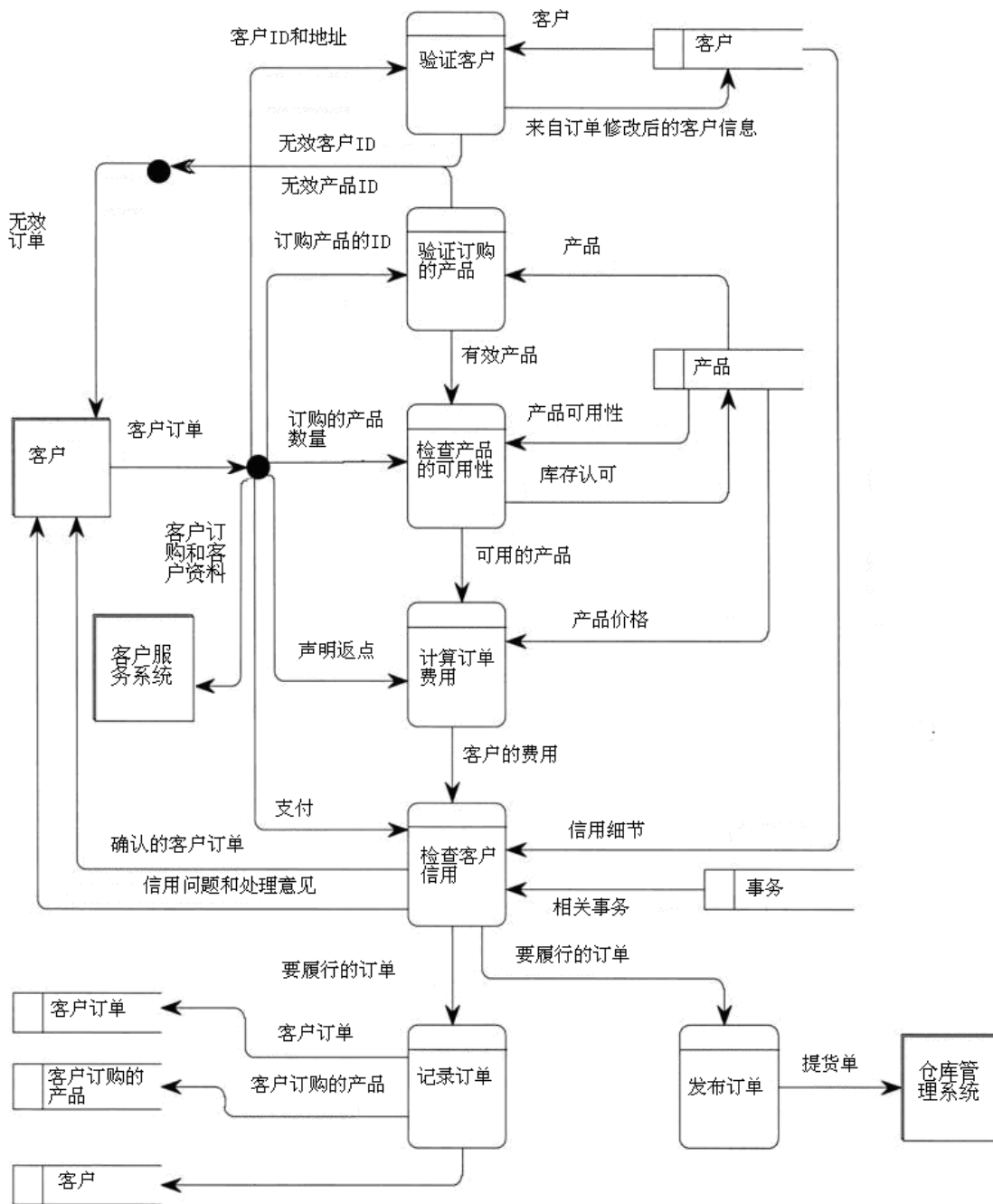
3. 系统 DFD 图

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, 所以, 构造一个或者多个系统或者子系统中所有事件相互关系的系统图也是有意义的。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。



4, 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。



5. 完整的规格说明

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。

第三节 面向对象的需求分析和用例

在面向对象的需求分析中，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。但是，还是有相应的方法来描述功能，这就是用例，这也是需求分析的重要部分。

一、用例及用例图的基本概念

用户一定会有自己的目标，并且希望计算机能够帮助他们实现这些目标。

用例就是表达如何使用系统达到目标的一组情节。

用例的几个概念

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。



参与者名 >

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”。通俗地讲，场景实际上是在说故事。一般来说，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。

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，是功能性行为。

参与者的发现：

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。

二、用例 (UseCase) 及其定义

1) 用例：

1. 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**，用例的图示如下。



<<用例名>>

2. 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。

3. 通常，用例侧重于功能，但不重点描述该功能的实现细节。
4. 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。

2) 用例的分类

1. 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）。

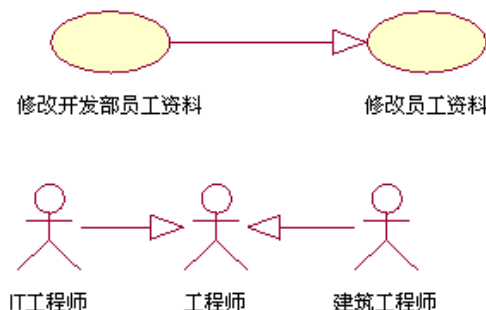
2. 系统用例（System Use Case）

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）。

3) 用例的联系（横向方面）

1. 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。

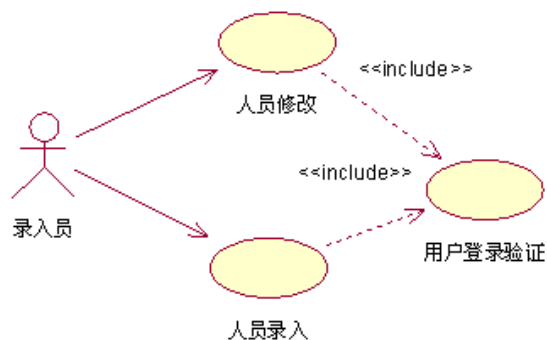


2. 包含关联（include）

包含关联指一个基本用例的行为包含了另一个用例的行为，这种关联是一种依赖关系，**被包含的用例不能独立存在，只能作为包含它的用例的一部分。**

例如，一个信息维护的模块，无论是录入人员信息还是修改人员信息，都必须对当前登录者进行验证，因而录入及修改人员信息这两个用例都用到了对当前用户的权限验证的用例。

其表示方法为用一条虚箭线从基本用例指向被包含的用例，并标有构造型<<include>>：

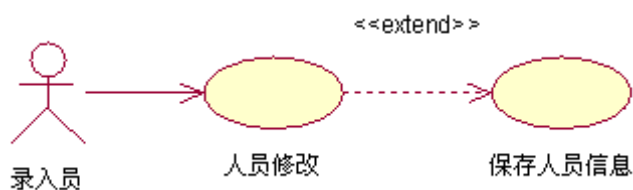


3. 扩展关联 (extend)

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。

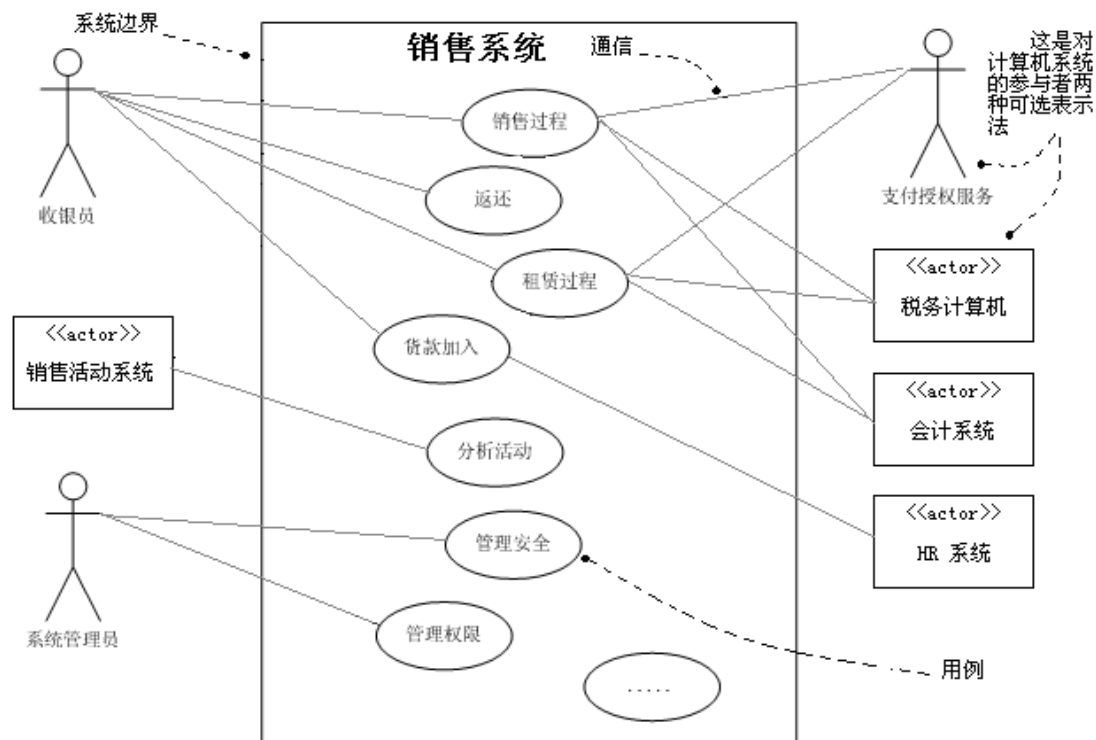
例如，**保存人员信息**用例可以是删除人员信息及新增和修改人员信息用例的扩展，它们之间存在着扩展关系。如果特定的条件发生，扩展用例的行为才能执行。

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，并在线上标注构造型<<extend>>:



三、用例图及其表达原则

用例图主要表现各个用例之间宽泛的关系，如下图所示。



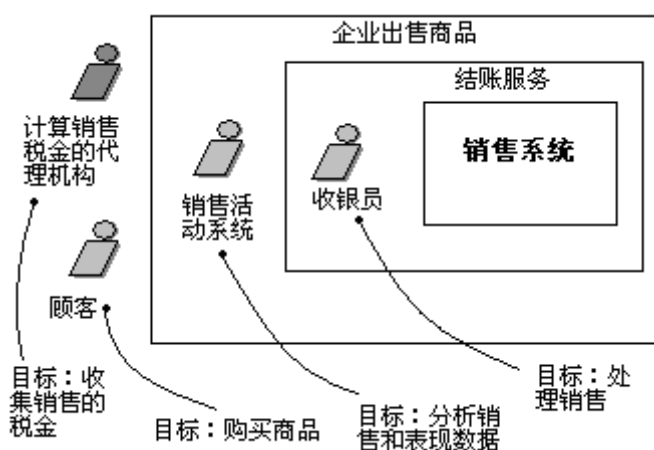
在上面的图中，计算机系统的参与者有两种表示法，其中有些人喜欢用不同于人型的方框表示外部计算机系统参与者，<<actor>>称作 UML 构造型，这是用某种方式分类元素的方式够造型的名称被书名符号包住，这本来就是法文印刷中表示引用的符号。

注意：主要参与者和用户目标和系统边界有关

有个问题，在“处理销售”用例中，为什么主要参与者是收银员而不是顾客呢？

这和系统边界有关，我们定义的销售系统边界，服务目标是收银员。

如果把系统边界定义为企业交款服务，那顾客就是一个主要参与者了。



我们也可以通过事件分析找出参与者。

第四节 用例的事件流与用例文档

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。

一、用例的事件流的组成

(1) 用例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。

二、描述用例的事件流的主要方式

1，面向过程语言：

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面向过程说明（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 workflow 模型、分析用例以及处理多线程应用程序的得力工具。

三、事件流描述文档的基本要求

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) 模板格式

用例的模版可以有不同的形式，关键是要表达清楚：

作者: _____

日期: _____

版本: _____

用例名:		用例类型
用例 ID:		
主要业务参与者:		
其它参与者:		
项目相关人员兴趣:		
描述:		
前置条件:		
后置条件:		
触发条件:		
基本流程:		
扩展流程:		
结束:		
业务规则:		
实现约束和说明:		
假设:		
待解决问题:		

对于上述一些时间流, 以及一些关键和重要的问题, 需要对用例进行详细设计, 这是需要使用文档而不是图。

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。

这就回答了一个问题, 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。

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

我们可以把主要成功场景做成“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.

3) 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标识和数量。

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。

4) 特殊需求

如果有一些与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 中可以插入新的业务规则。

经典的统一过程建议，第一次写出用例的时候，把特殊需求和用例写在一起。

但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。

5) 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”，需要在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。

五、示例：销售系统

这里为了把问题表达清楚，添加了一些项目。

举个例子。

POS 销售系统

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:	<p>收银员：希望能准确、快速的输入，而且没有支付错误。</p> <p>售货员：希望自动更新销售提成。</p> <p>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。</p> <p>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。</p> <p>政府税务机关：希望从每笔交易中抽取税金。</p> <p>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。</p>	
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:	<p>1. 顾客携带购买的商品到达 POS 机收费口</p> <p>2. 收银员开始一次新的销售</p> <p>3. 收银员输入商品标识</p> <p>4. ...</p> <p>重复 3 - 4 步，直到结束。</p> <p>5.</p> <p>.....</p> <p>10. 顾客携带商品和收据离开</p>	

替代流程	<p>*a.任何时刻，发生以下状况，系统将会崩溃。</p> <ol style="list-style-type: none"> 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。 <p>3a.非法标识</p> <ol style="list-style-type: none"> 1. 系统指示错误并拒绝输入。 <p>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</p> <ol style="list-style-type: none"> 2. 收银员可以输入商品类别的标识和数量。 <p>3-6a.顾客要求从已输入商品中减去一个商品</p> <ol style="list-style-type: none"> 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。 <p>.....</p> <p>7b.信用卡支付</p> <ol style="list-style-type: none"> 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 <p>2a 系统检测到和外部信用卡授权服务系统通信故障</p> <ol style="list-style-type: none"> 2. 系统向收银员指示发生了错误 <p>2.收银员向客户请求更换支付方式</p>
结束:	当客户完成支付，该用例结束。
特殊需求:	<ol style="list-style-type: none"> 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:	<p>3a. 商品标识可以用条码扫描也可以用键盘输入。</p> <p>3b. 商品标识可以采用 UPC、EAN、JAN、SKU 等不同的编码方式。</p> <p>7a. 信用卡账号信息可以使用读卡器或键盘输入。</p> <p>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。</p>
发生频率:	可能会持续发生。
待解决问题:	<ol style="list-style-type: none"> 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。

在统一过程中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、案例：订单处理子系统

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；

正确的：销售人员要 e-mail 给客户。

- 1，**客户**使用公司的 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。

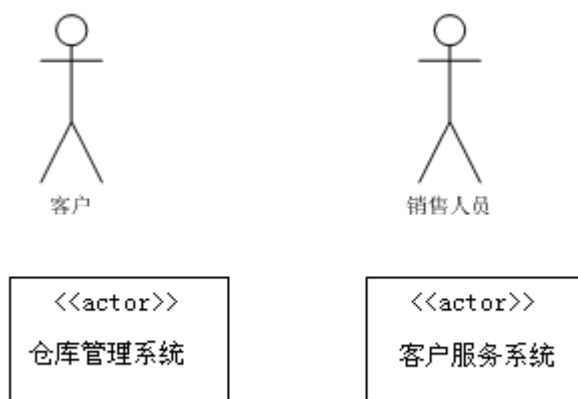
从中我们可以发现四个参与者：

客户；

销售人员；

仓库管理系统；

客户服务系统。



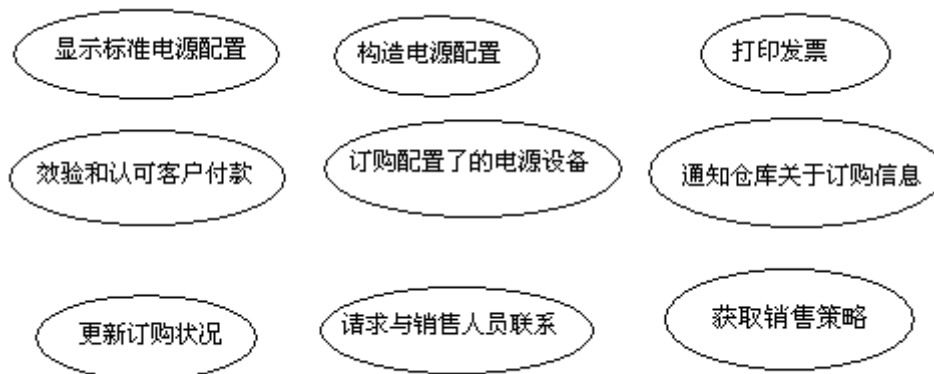
3、建立用例

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。

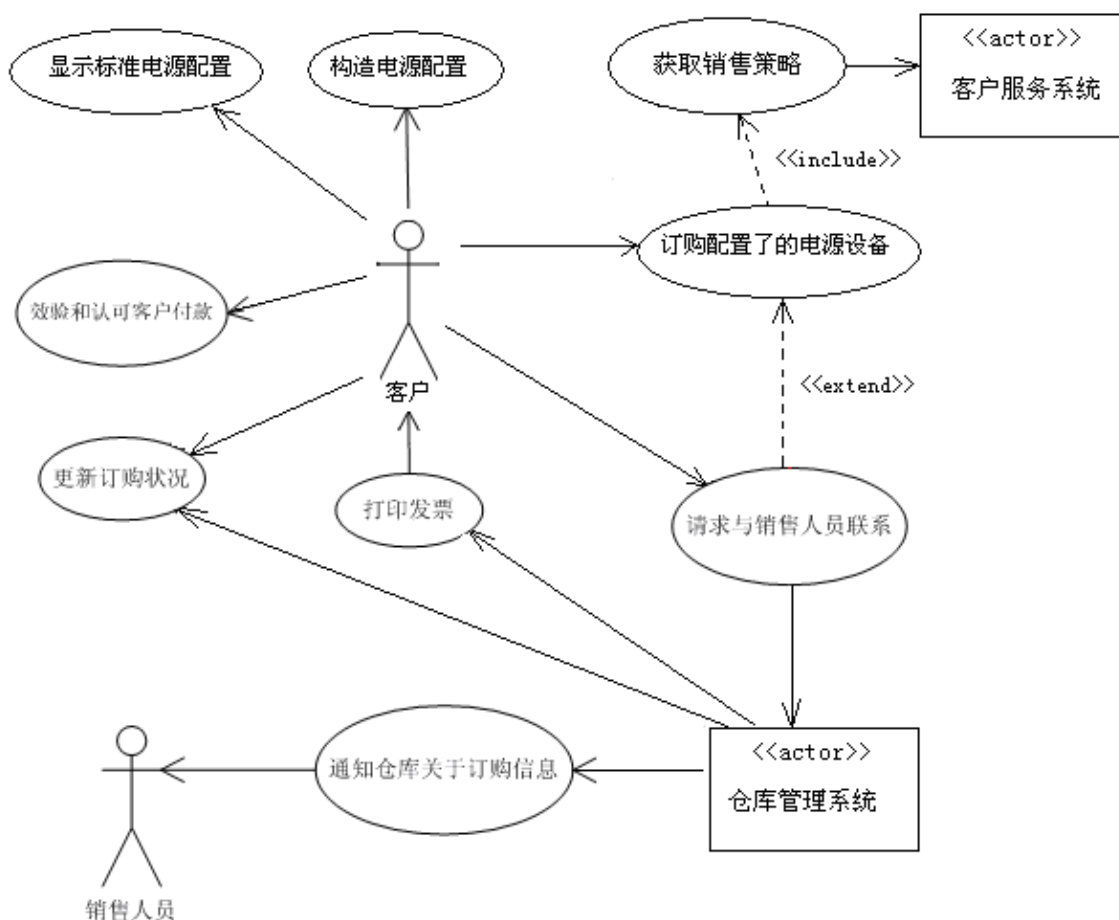
编号	需求	参与者	用例
1	客户 使用电源部的 Web 页面查看所选择的电源设备标配，同时显示价格。	客户	显示标准电源配置
2	客户 查看配置细节，可以更改配置。	客户	构造电源配置
3	系统由 客户服务系统 取得该 客户 的等级，以及由销售策略决定的折扣，同时计算价格。	客户 客户服务系统	获取销售策略
4	客户 选择订购，也可以要求 销售人员 在订单真正发出之前和自己联系，解释有关细节。	客户 销售人员	订购配置了的电源设备 请求与销售人员联系
5	要发出订单， 客户 必须填写表格，包括地址，付款细节（信用卡还是支票）等。	客户	订购配置了的电源设备 效验和认可客户付款
6	销售人员 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。用例文档大概 10 页左右，需要完整的表达用例的过程。

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:	<p>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。</p> <p>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。</p>	
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:	<p>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。</p> <p>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。</p> <p>3. 客户选择购买功能键来发送订单给系统。</p> <p>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。</p> <p>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。</p>	
扩展流程:	<p>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</p> <p>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。</p> <p>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。</p>	

七、用例的目标

1) 基本业务过程的应用

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，并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：

首先找出用户的目标，然后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。

第五节 非功能性需求的识别

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）

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。

第六节 合理的应用活动分析

一、为什么要应用活动建模

上面用文字表示用例事件流，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，活动图更善于表达流程和并发关系。

活动图表示了计算的步骤，每一步都是一个关于干什么事的状态。因为这个原因，执行步骤被称之为活动状态。

这个图表示了哪步应该按次序执行，哪步可以并行进行，从一个活动状态到另一个活动状态的转换，称之为“转换”。

如果用例文档完成了，活动状态可以从主要的和附加的流中间发现。

但是用例描述和活动模型之间存在着一些重要的区别，用例描述是从外部参与者的角度出发来编写的，而活动模型则采用内部系统的观点。

活动图也可以在用例编写以前，在一个高的抽象层次来理解业务进程。

活动

如果活动建模是为了表达可视化用例中活动的次序，则活动状态可以根据用例文档来建立，这时，活动应该从系统的角度，而不是参与者的角度来命名。

活动图形也可以表达活动状态和活动行为。

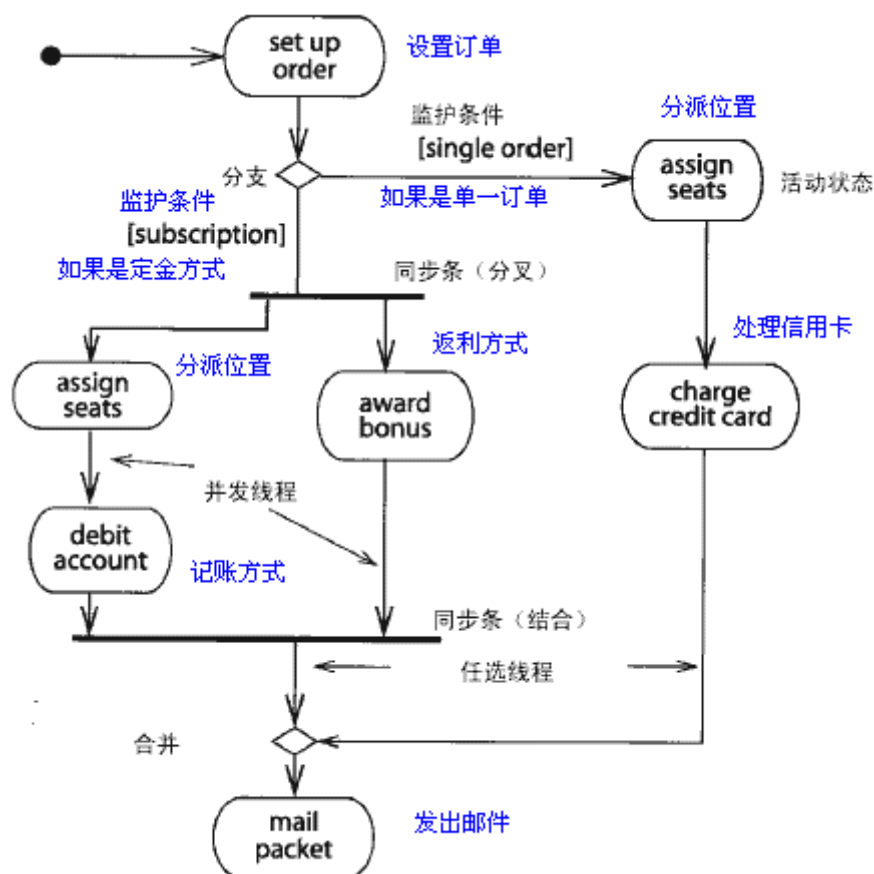
活动和行为的区别在于其时间跨度，活动是要花时间来完成的，而行为则可看成快照，被认为是不会花时间的。

活动视图（Activity Diagram）主要用于对计算流程和 workflows 建模，它很类似于面向过程建

模中的流程图。

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。比较适合建立 workflow 模型。

下面是一个订单处理的活动图。

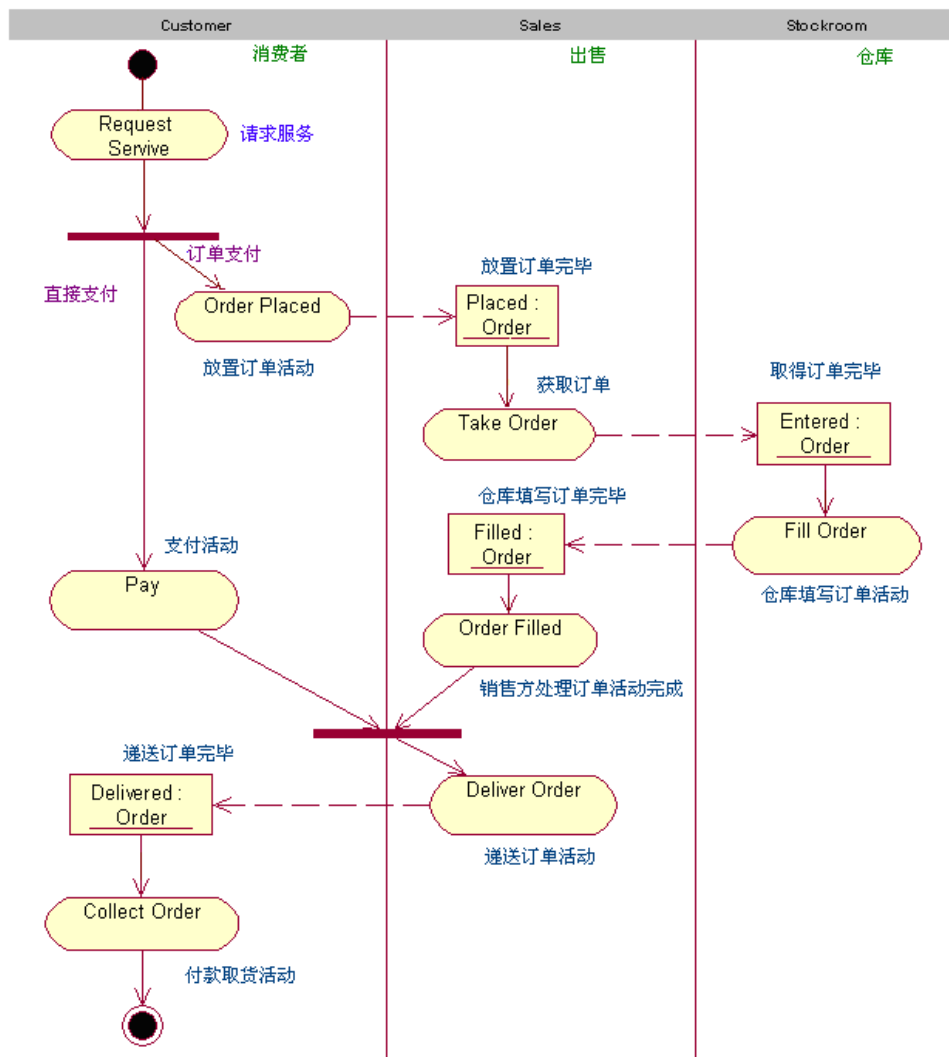


泳道：

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。

下图表示了一个销售过程的活动。



活动图并没有表示出计算处理过程中的全部细节内容。

它只是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设计工作。

二、案例：订单处理子系统

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。	显示当前配置； 获得客户请求

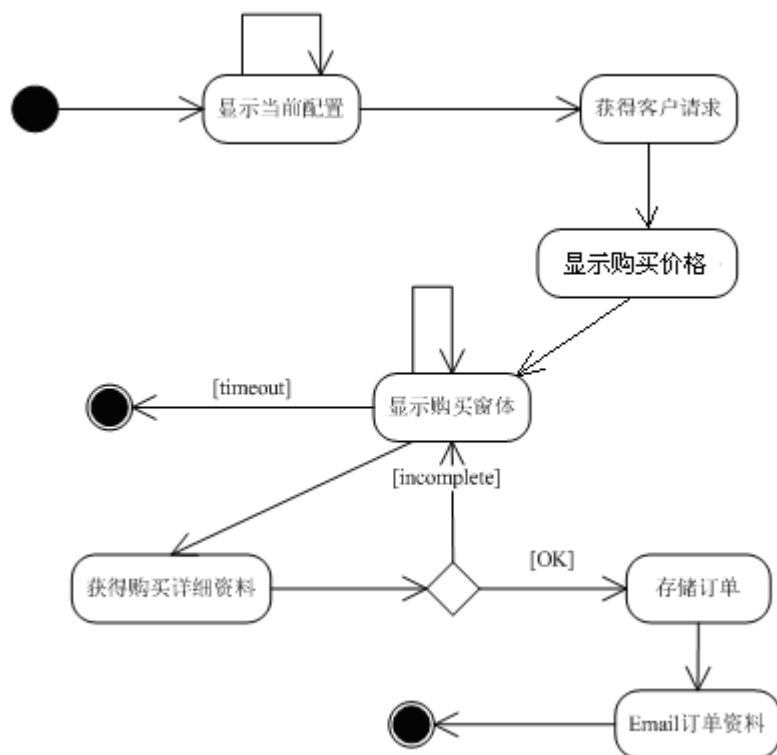
2	系统请求客户输入购买细节，包括销售人员的名字（如果知道的话）、运送信息（客户的名字和地址）、发票细节（如果运送地址不同的话）、付款方法（信用卡和支票）以及任何其它注释。	显示购买窗体
3	系统由销售服务系统取得客户的等级以及当前销售策略，计算客户购买的实际价格。	显示购买价格
4	客户选择 Purchase （购买，或者相似命名的）功能键来发送订单给 TB 公司。	获得购买详细资料
5	系统给购买订单赋予一个唯一的订单号码和客户账号，系统将订单信息存入数据库。	存储订单
6	系统把订单号和客户号与所有订单细节一起 e-mail 给客户，作为接受订单的确认。	Email 订单资料
7	客户在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。

三、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

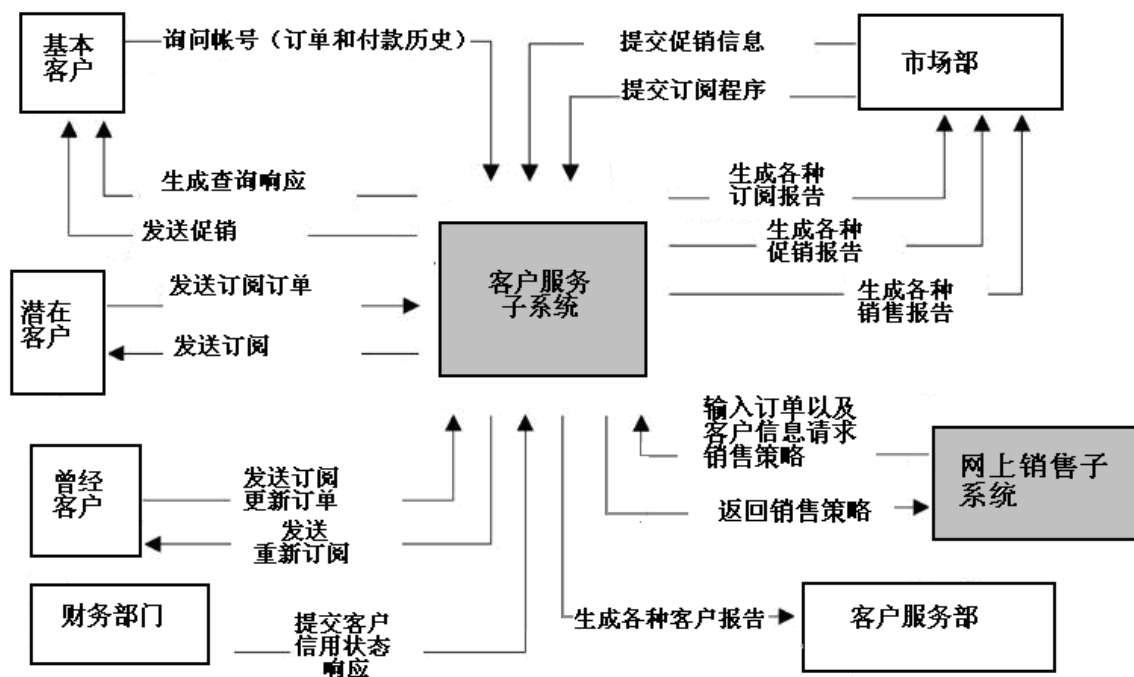
与已经建立的网上订购项目相比，这个新系统的最大特点，是把市场行为作为系统的重要功能，所以功能上和以前的系统有诸多不同。

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。



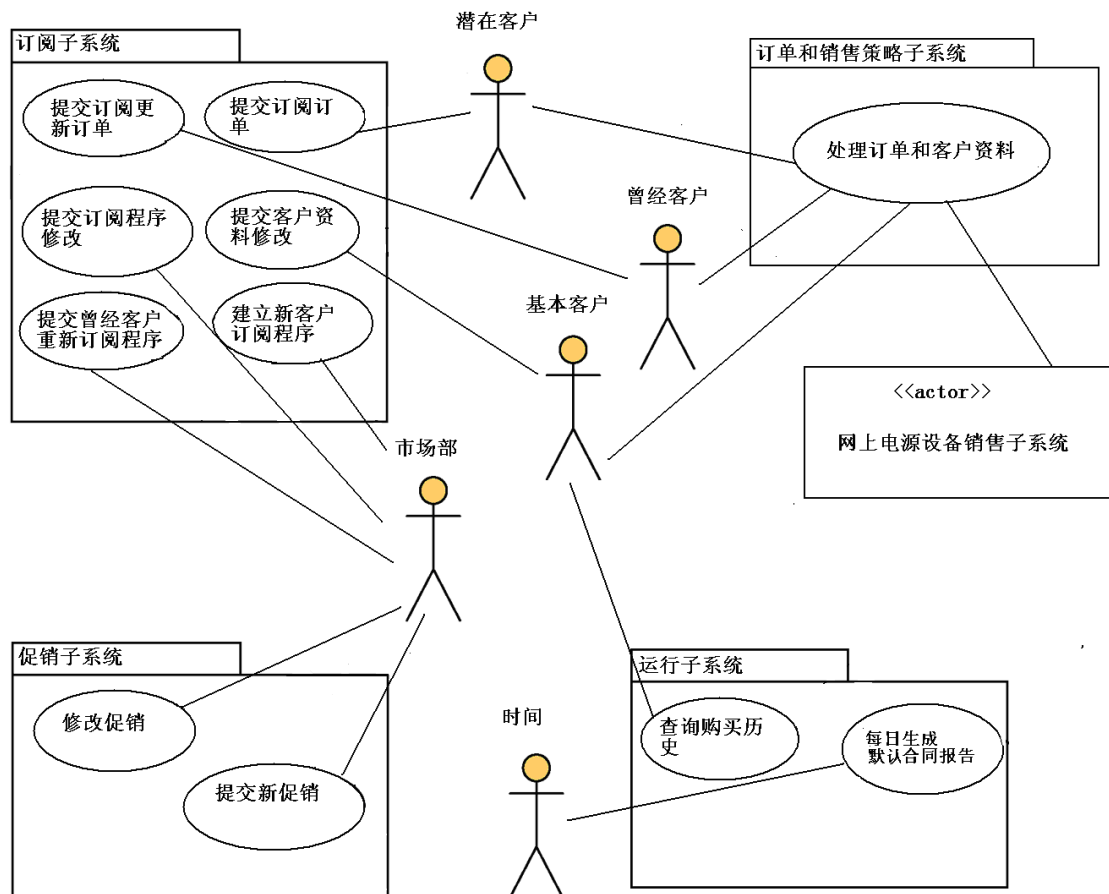
在这个系统中定义边界的时候，可以考虑把网上销售子系统暂时排除在外，也就是把网上销售子系统定义成外部接受者。我们可以建立一些用例并定义它的词汇。

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	潜在客户

		在两年内购买一定数量的本公司设备。	
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。

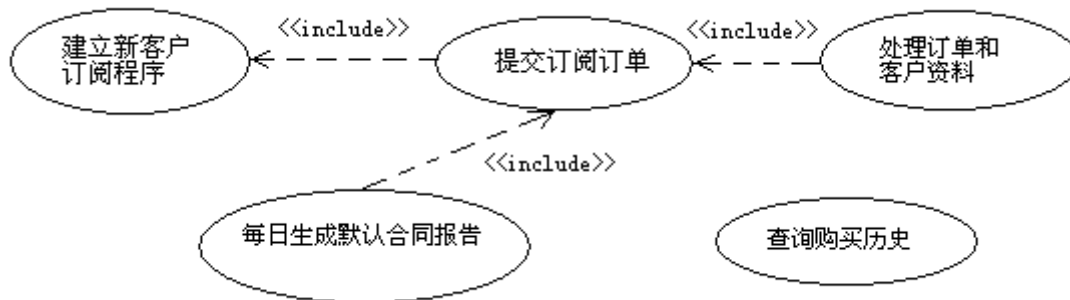
4，风险分析和优先级的考虑

为了发现最重要的用例，需要对用例的重要性或者开发风险进行评估，可以采用用例分级和评估矩阵来做这个初步分析，数据的来源可以采用项目相关人员和开发团队打分法来完成。

用例名称	分级标准 (1-5)						总分	优先级	构造周期
	1	2	3	4	5	6			
提交订阅订单	5	5	5	4	5	5	29	高	1
处理订单和客户资料	4	4	5	4	5	5	27	高	2
建立新客户订阅程序	4	5	5	3	5	5	27	高	1
每日生成默认合同报告	1	1	1	1	1	1	6	低	3
修改促销	2	2	3	3	4	4	18	中	2
提交新促销	3	2	3	4	2	1	15	低	2

从上面的表中，发现“提交订阅订单”优先级最高，应该首先开发。但这还不能完全确定，还需要考虑用例的依赖关系。

所谓用例的依赖关系，表达的是这个用例完成的状态（后置条件）恰恰是这个用例的前置条件。从下面的图可以看出来，建立“新客户订阅程序”虽然优先级并不是最高的，但它处于依赖关系的最前端，所以应该最先开发。

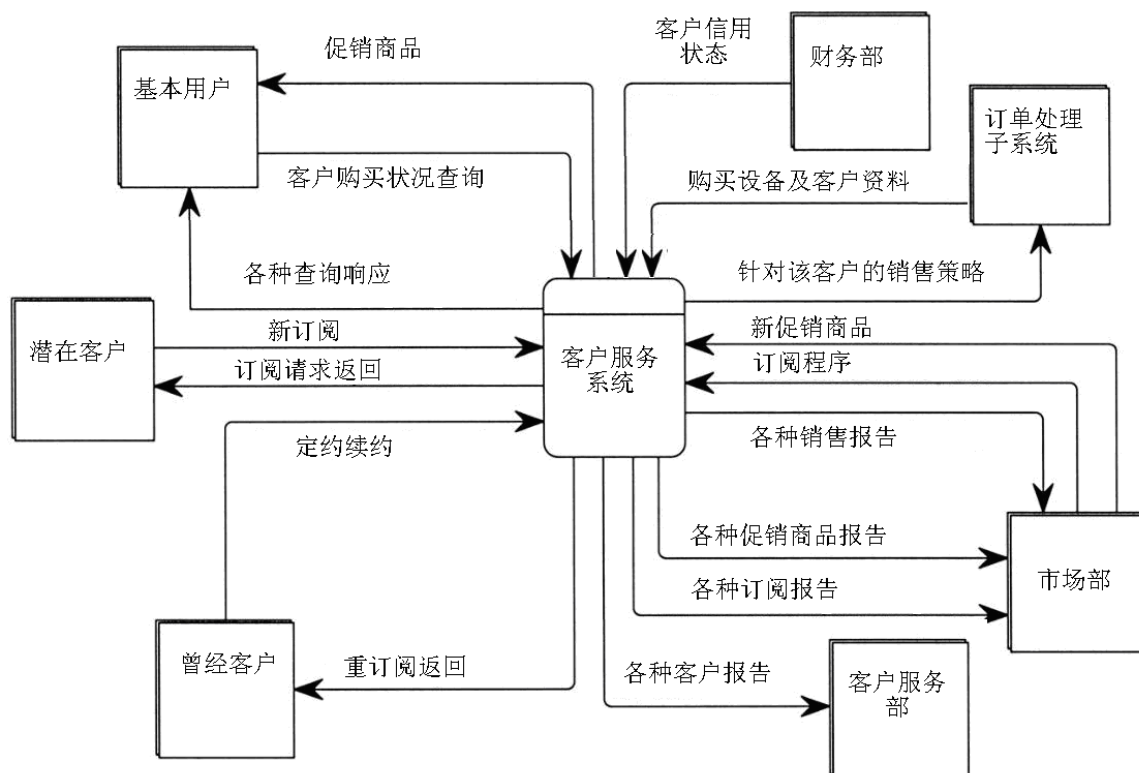


四、客户服务子系统的过程分析

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

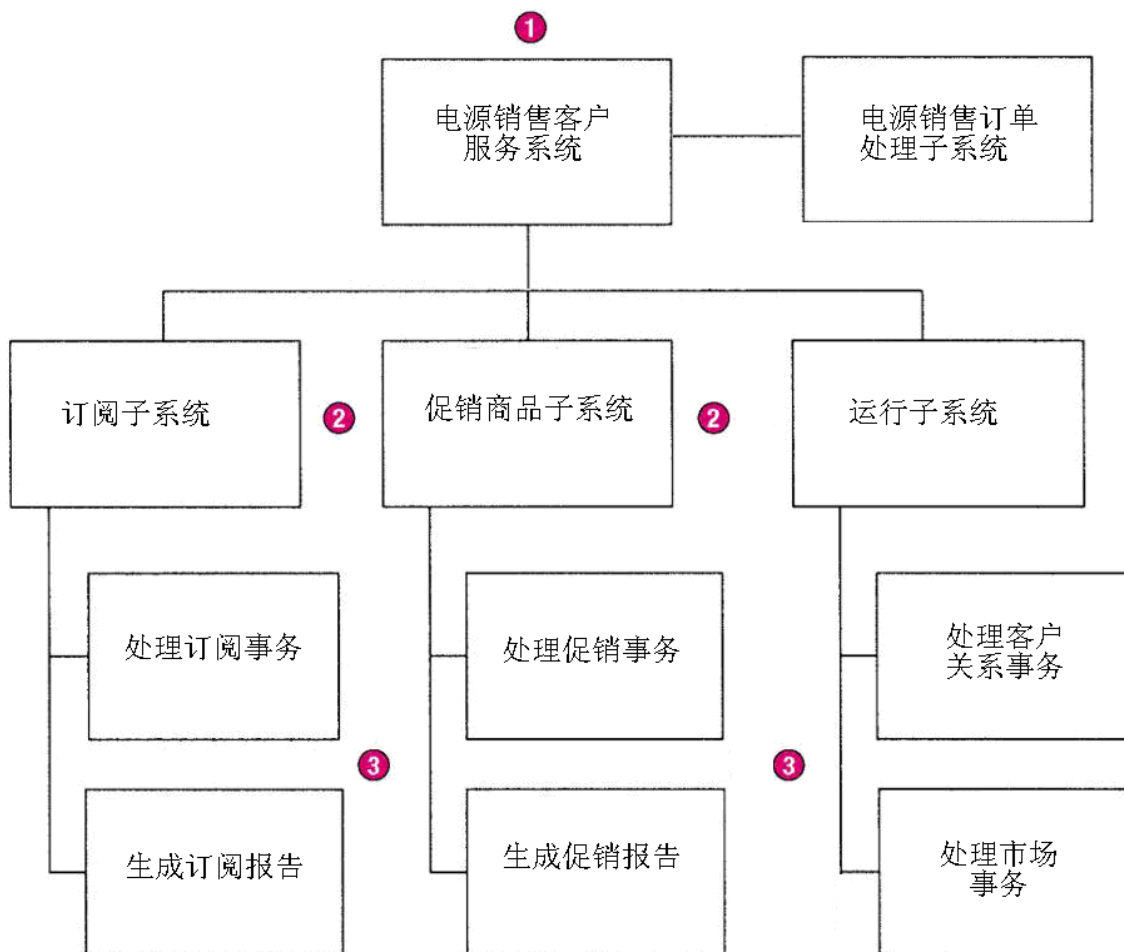
这个问题已经在前面讨论过，这里就直接画出这个系统的上下文来。



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程图（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：

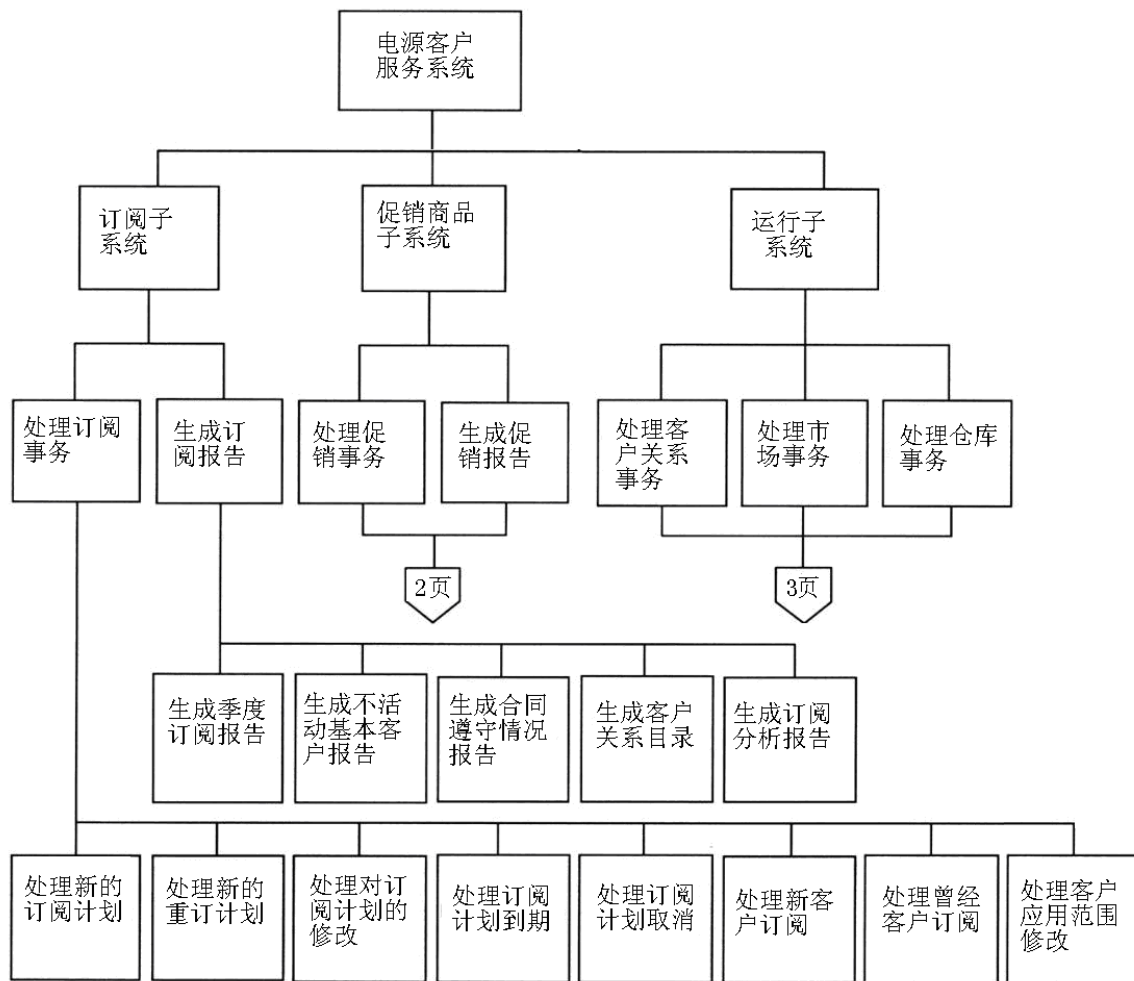
- 1) 引发事件的参与者，它们将成为 DFD 的外部代理。
- 2) 事件，它们由 DFD 的某个过程处理。
- 3) 输入或者触发器，它们将成为 DFD 中的一个数据流或者控制流。
- 4) 所有的输出响应，它们也将成为 DFD 中的数据流，注意我们是用括号来表达时序事件。
- 5) 输出，注意这里不要暗示实现方式，当我们使用**报告**这个词的时候，并不一定指的是书面报告。输出包括了修改数据模型中存储的实体模型，比如创建新的实体实例、修改实体的现有实例以及删除实体实例等。

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。

第三章 领域建模与系统行为分析

用例模型原则上不是面向对象的，它描述的是系统的功能，只是建立系统的最初的输入，为了更细腻的分析需求，从面向对象的角度，可以建立领域模型。

识别一个丰富的对象集或者领域类集，是面向对象分析的核心工作，做好这项工作，将会在设计和实现期间获得丰富的回报。

第一节 领域建模的思想和方法

领域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。

领域模型是说明问题域里（对建模者来说）有意义的**领域类**，它是面向对象分序的时候要创建的最重要的工作（必须说明，用例虽然也是一个重要的分析工作，但它并不是面向对象的，它是强调的概念的过程视图）。

一、领域建模的思想及其方法学问题

什么是“问题域”和“领域建模”？

问题域：

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

领域建模：

1. 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。
2. 对这些问题进行归纳、分析的过程就是**领域建模**（这个域，指的就是问题域）。

建立领域模型的好处：

1. 通过建立领域模型能够从现实的问题域中找到最有代表性的概念对象
 2. 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，是把一个相关的领域，分解为单个领域类或者对象（是一个我们能够理解的概念）。

领域模型是领域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：概念模型、领域对象模型、分析对象模型等。

在 UML 中，领域模型是不定义操作（方法）的一组类图来说明，它主要表达：

1. 领域对象或者领域类
2. 领域类之间的关联
3. 领域类的属性

属性用以表达对象的状态。

（1）三种领域类

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。

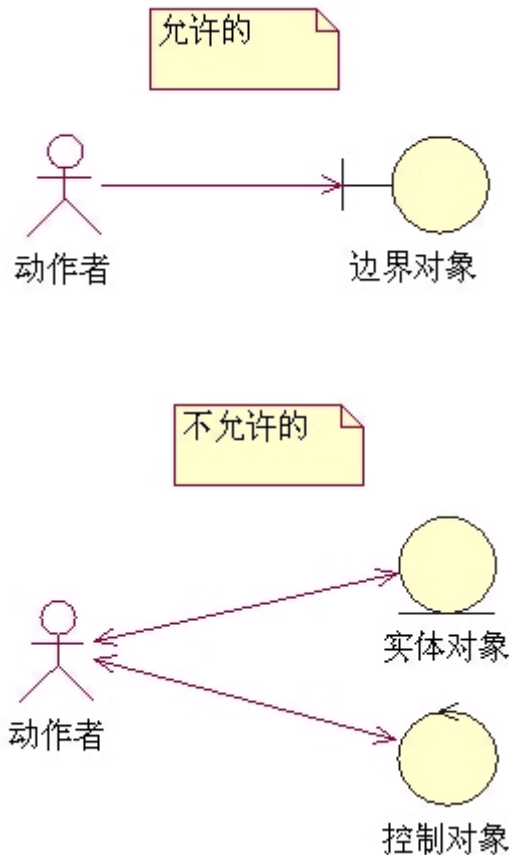
通过它可以表达和管理系统中的信息。在模型中, 系统中的关键概念以实体对象来表现。其主要的责任是: 业务行为的主要承载体

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

三种领域类的 UML 的图示如下:

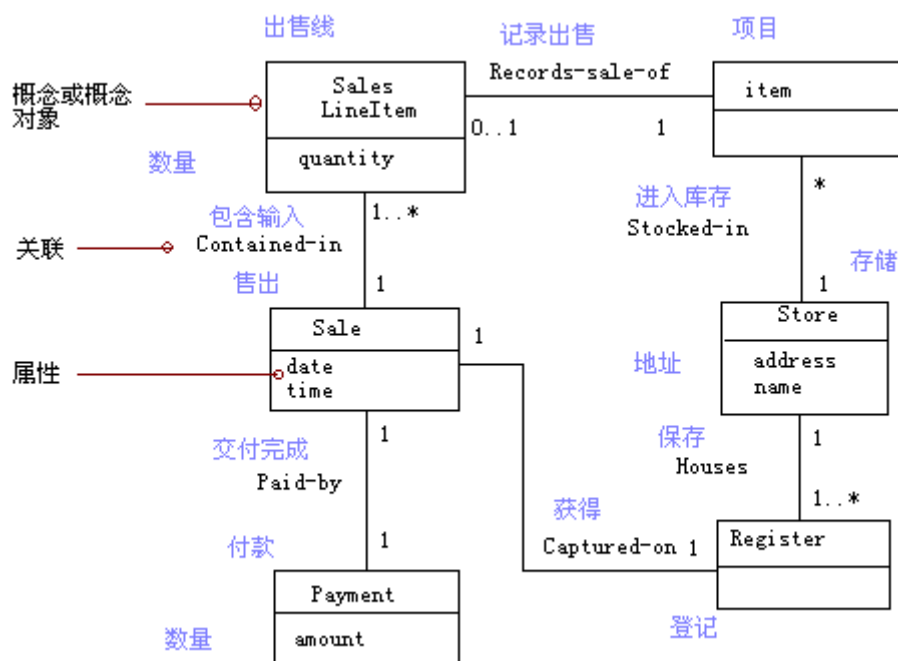


(2) 领域建模的简单例子

下面举个简单的例子, 说明领域建模的基本概念。

1) 问题的描述

例如：两个领域类 Payment（支付）Sale（售出）在领域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，领域模型实际上是可视化了领域中的单词或领域类，并且为这些单词建立了领域类。

也就是说，领域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) 领域模型不是软件组件的模型

领域模型视相关现实世界领域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领域模型中表述：

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，但领域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）。

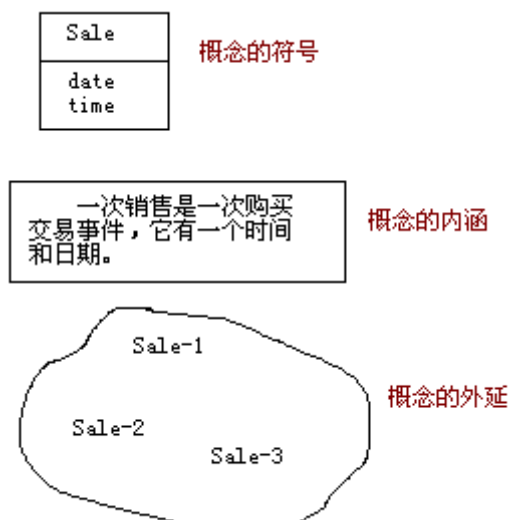
4) 领域类

领域模型表示领域中的领域类或词汇，一个不太准确的描述：一个领域类就是一个观点、事务或者对象。

比较准确的表达：

领域类可以按照它的符号、内涵和外延来考虑。

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



二、领域类的识别

我们的目标是在相关领域中创建有意义的领域类。

比如说创建“处理销售”用例中的相关领域类。

一般来说，用大量细粒度的领域类来充分描述领域模型，比粗略描述要好。

下面是识别领域类的一些指导原则：

- 不要认为领域模型中领域类越少越好，情况往往恰恰相反。
- 在初始识别阶段往往会漏掉一些领域类，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领域类的信息，或者因为领域类没有属性，就排除掉这个领域类。
- 无属性的领域类，或者在问题域里面仅仅担当行为的角色，而非信息的角色的领域类，都可以是有效的领域类。

1) 识别领域类的策略

下面提供了两种识别领域类的技巧。

1. 使用领域类分类列表。
2. 识别名词短语。

2) 使用领域类分类列表

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

领域类分类	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领域类

曾经有人提出了用名词短语分析找出领域类的方法，然后把它们作为候选的领域类或者属性。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领域类（“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：

把领域类分类，和词语分析一起使用。

三、领域建模的指导原则

1) 事物的命名和建模

领域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，也适用于领域模型

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：

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

2) 在识别领域类的时候一个常犯的错误

在建立领域模型的时候，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？还是单独的领域类 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，所以应该是个领域类。

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领域类，因为领域模型中，属性非常少见。

四、分析相似的领域类

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，在领域模型中，是不是应该用 Register 而不是 POST 吗？

我们应该知道，领域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 是一个领域中常见的术语，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，关键是看你的领域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。

五、为非现实世界建模

一些业务领域有自己独特的概念，只要这些概念是在业内被认可的，同样可以创建领域类，

比如在电信业可以建立这样的领域类：

消息（Message）、连接（Connection）、端口（Port）、对话（Dialog）、路由（Route）、协议（Protocol）等。

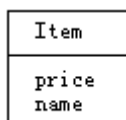
六、规格说明或者描述领域类

在领域模型中，对领域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，需要添加领域类的说明类：

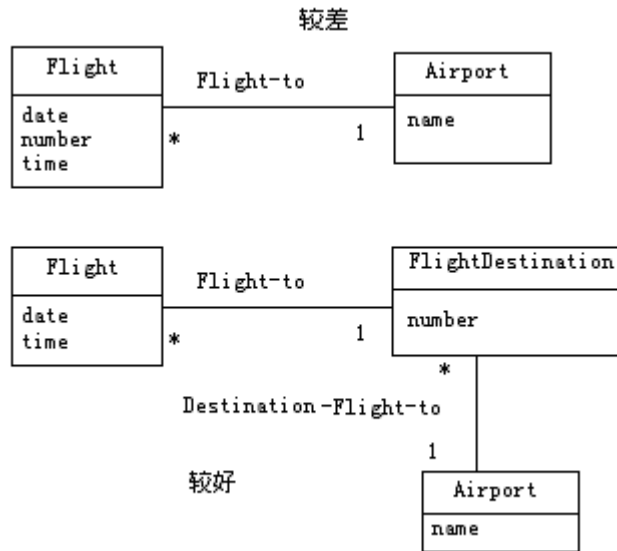
- 商品或服务的信息描述，独立于商品或者服务当前已经存在的任何实例。
- 删除所描述的事物，会导致维护信息的丢失。
- 希望减少冗余或者重复的信息。

2) 服务的描述

作为领域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。



七、统一过程中的领域模型

根据“统一过程的时间及其时间安排”的那张表，在 UP 中，通常在细化阶段开始并完成领域模型的建模。事实上，对于一个有经验的系统构架师，在每次迭代里开发领域模型，只需要几个小时就够了。

在 UP 中，有一个业务对象模型（BOM），但实际上并不通用，而领域模型实际上是 BOM 的一个正变的变体。在 RUP 中，BOM 是这样来定义的：它是业务员和业务实体如何相关联，以及为了完成业务如何写作的抽象。

BOM 可以用多种不同的图（类图、活动图、顺序图）来表示，这些图说明整个企业应该如何运行。

不过对于单个软件的应用，这似乎是个不太通用的活动。尽管它还有一些变通，但是在系统架构设计中，领域模型仍然是最为广泛的被采用的。

第二节 领域模型的关联

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领域模型，分而会使领域模型变的混乱，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。但在领域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

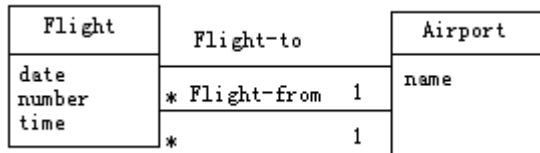
观点出发，也可能只有“1”或者“*”是合理的。

再一次提醒，发现领域类比发现关联更重要，花费在领域模型创建的大部分时间，应该被用于发现领域类，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。



第三节 领域模型的属性

发现和识别领域类的属性，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。

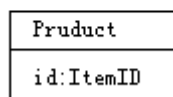
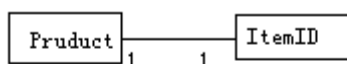
当然也可以使其它的一些必要的类型，比如：Color（颜色）、Address（地址）、PhoneNumber（电话号码）等。

二、非原始的数据类型类

在领域类中，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。

第四节 泛化建模

泛化和特化是概念建模的基本概念，另外，领域类的层次，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、领域模型的概念提取

UP 的领域模型，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领域模型的有关概念。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。

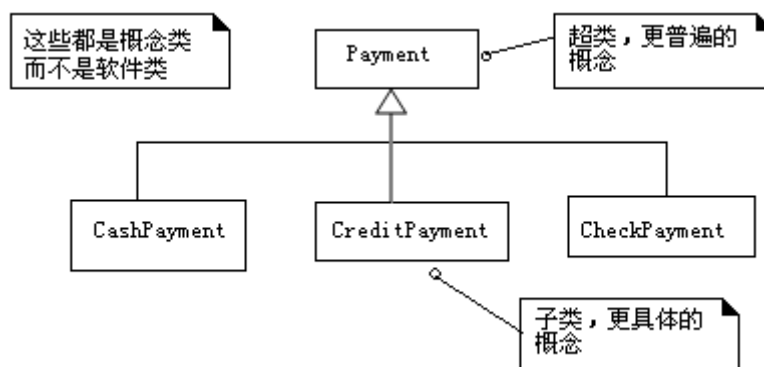
这些交易概念，不需要代表计算机中的记录，也不需要代表连线上的比特数据，它们代表了独立于执行方式的交易的抽象。

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，这里讨论的是领域类，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。

在领域模型中，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：

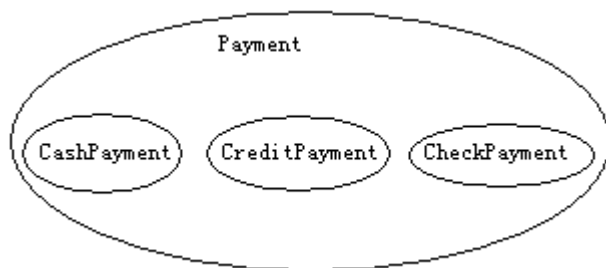
一个概念性超类的定义，比一个概念性子类的定义更为普遍或者范围更广。

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。

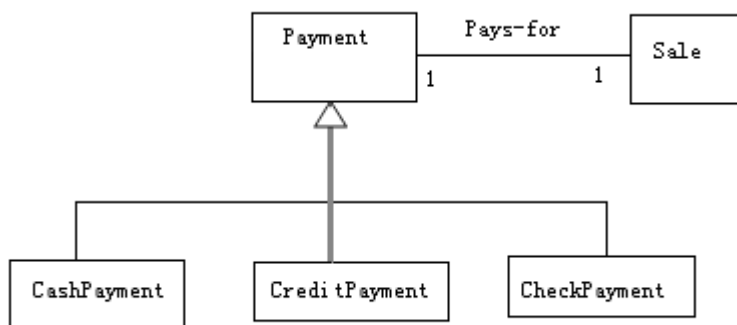
所有概念性子类集的成员，都是它们超类集的成员。



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 and 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）。

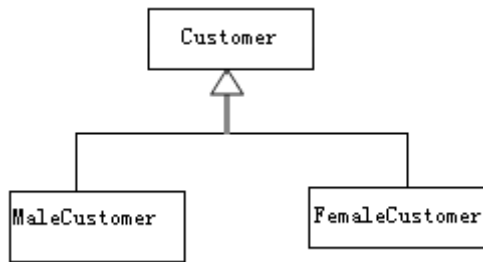
5) 什么是正确的概念性子类呢

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。

把一个领域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

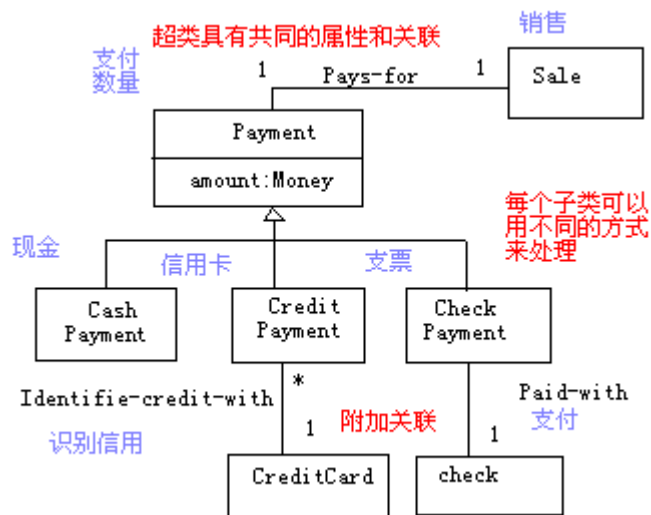
在多个潜在的子类之间，一旦发现共同特征，就可以暗示可以泛化得到一个超类。

下面是泛化和定义超类的动机：

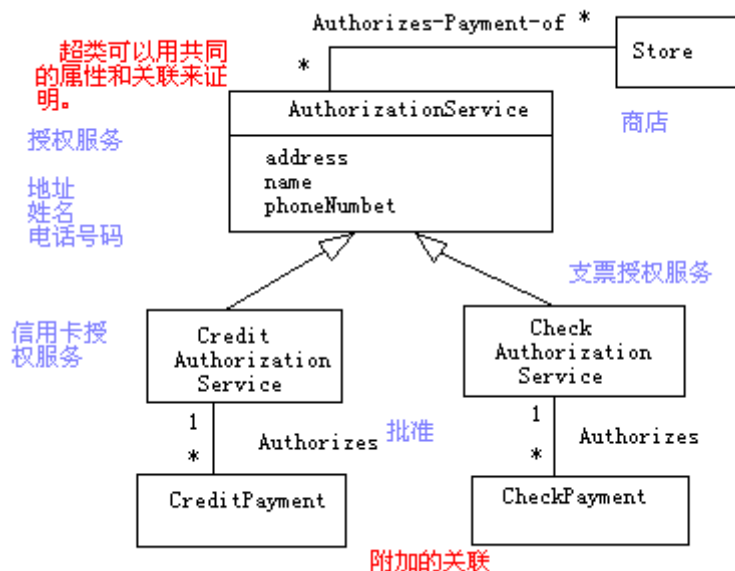
- 潜在的概念子类代表一个相似概念的变体。
- 子类遵守 100% 的 is-a 规则。
- 所有的子类具有共同的属性，可以提取出来并在超类中表示。
- 所有子类具有相同关联，可以提取并与超类相关。

8) 发现领域类的实例

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、抽象领域类

在领域模型里面，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，那么称它为抽象领域类。

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 是一个抽象的领域类。

第五节 精化领域建模及若干难以确定的要素

在前面讨论的基础上，我们需要把领域模型进一步的精化。

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？

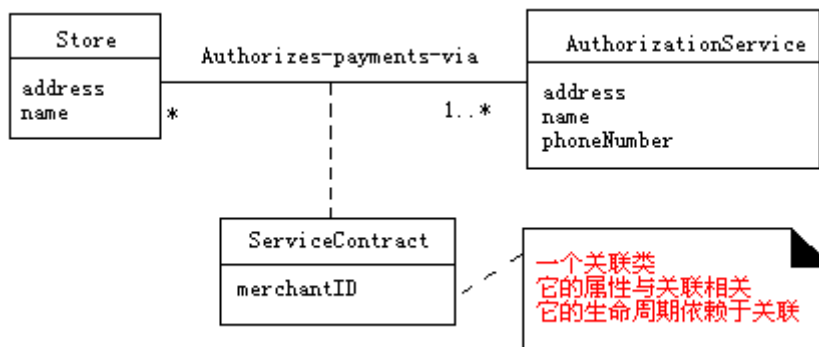
把 merchantID（店主 ID）置于 Store（商店）类中是不正确的，因为一个 Store 类会有多个 merchantID 值。

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：

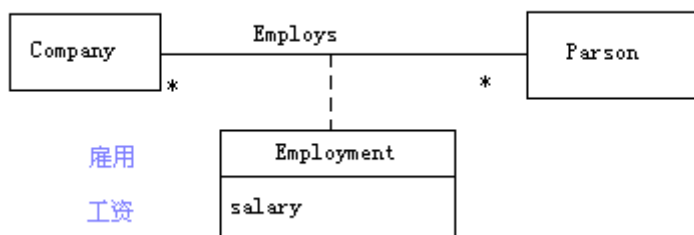
在概念建模中，如果一个类 C 的属性 A 同时具有多个值，那么不要把属性 A 放到类 C 中，而是把属性 A 放到与 C 关联的关联类里面去。



出现下面的情况可以在领域模型中加入关联类：

- 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。



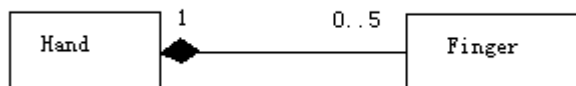
二、聚集和组合

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）。

但是在领域模型中，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，在领域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，在领域模型的关联问题上，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，来讨论领域建模的过程。

领域建模的目的，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，所以，这里的领域实体模型的建立，直接影响到数据库设计。

实体类定义的任何信息系统的本质，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。

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

寻找领域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？

它在应用领域的范围之内吗？

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 Order（订单）的区别到底在哪里？毕竟我们打算存储选择配置的电源，除非它的订单被提交。

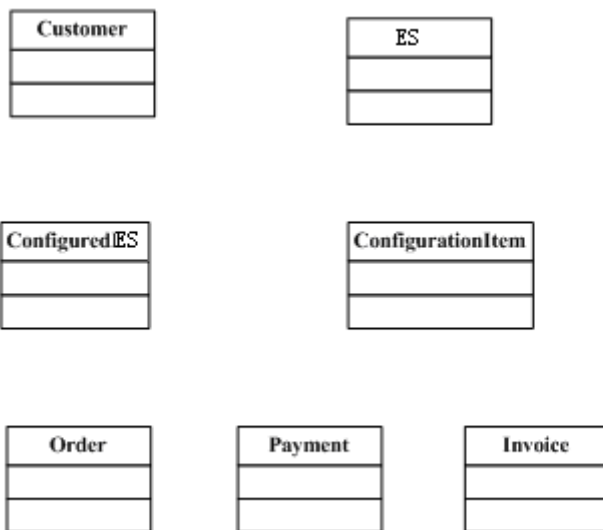
2，第 4 号和第 7 号需求中的 Shipment（出货）的含义是一样的吗？可能不一样，如果我们知道了运送是仓库的责任，这个类是不是还需要呢？

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？

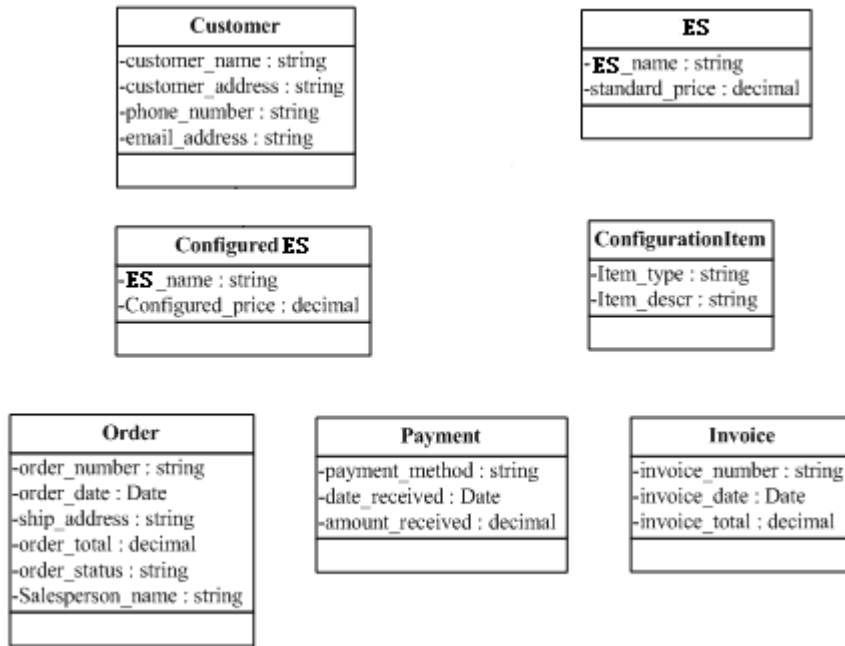
5，Salesperson（销售人员）是一个类，还是 Order（订单）还是 Invoice（发票）中的一个属性呢？

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。

但是在确定多重性的时候，可以作一些假定：

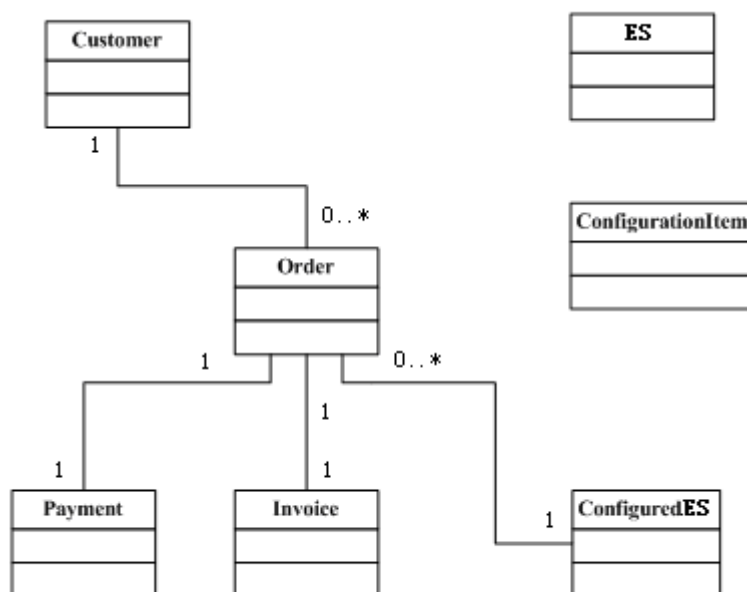
订单来自于单个客户，而客户可以提交多个订单。

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。

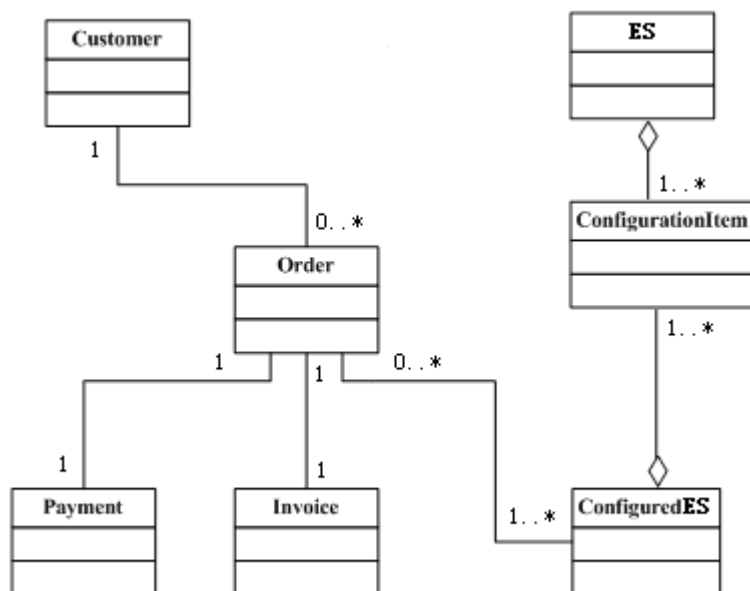


4) 发现聚集

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。

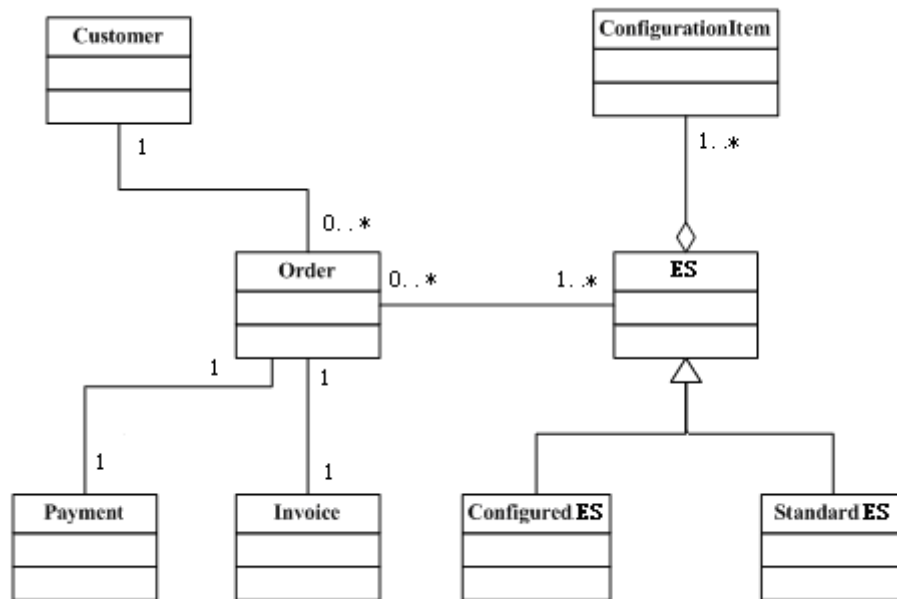
同样，一个自定义配置的电源也具有一个或者多个配置项目。



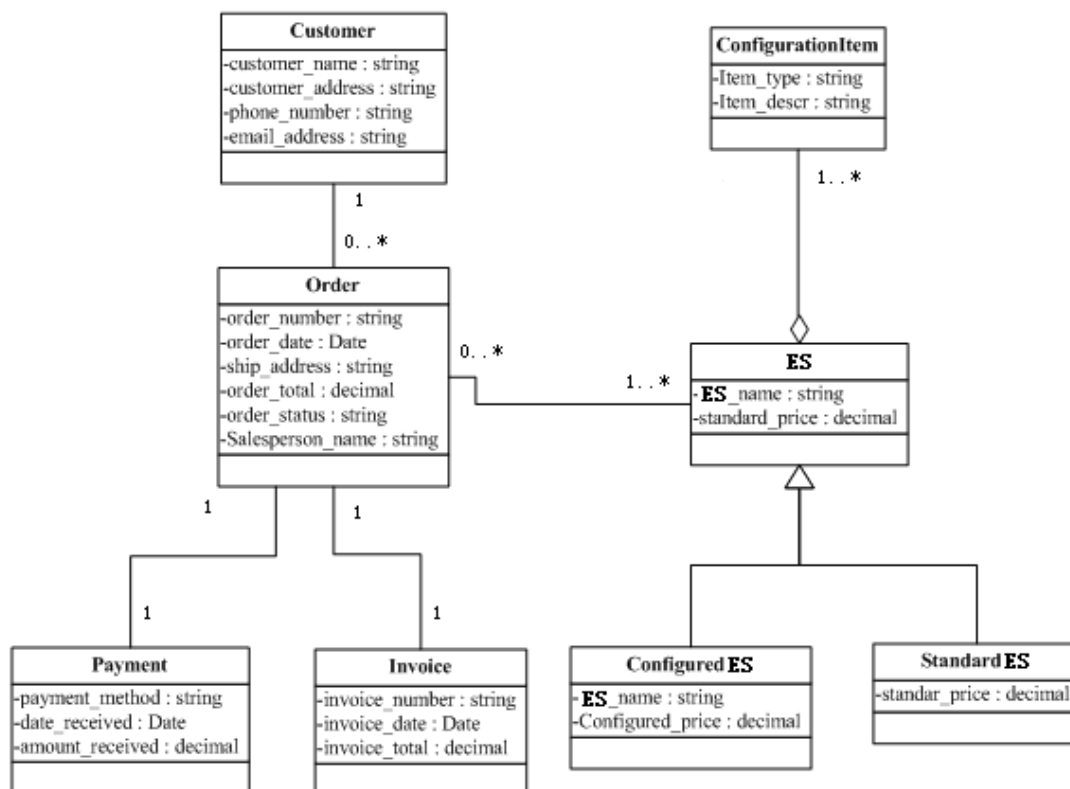
5) 泛化建模

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



第六节 系统行为分析中必须关注的问题

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）。

交互视图包括顺序图（Sequence Diagram）和合作图（Collaboration Diagram）和两种，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。

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、从整体的角度研究系统行为

用例描述外部参与者与我们创建的软件之间如何交互。

交互期间，参与者产生一个发送给系统的事件，通常要求系统响应这个操作。

比如：

收银员输入一个商品的 ID 的时候，收银员将请求 POS 系统记录商品的销售，这个请求将触发一个系统的操作。

为了表达这些外部参与者与系统交互的过程，使用 UML 顺序图是可取的。

一个系统顺序图（SSD），是一个用来表示用例特定场景、外部参与者产生的事件。所有的系统都被当作黑箱，图的重点是从参与者跨越到边界的事件。

系统顺序图是把系统作为黑箱来设计。

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

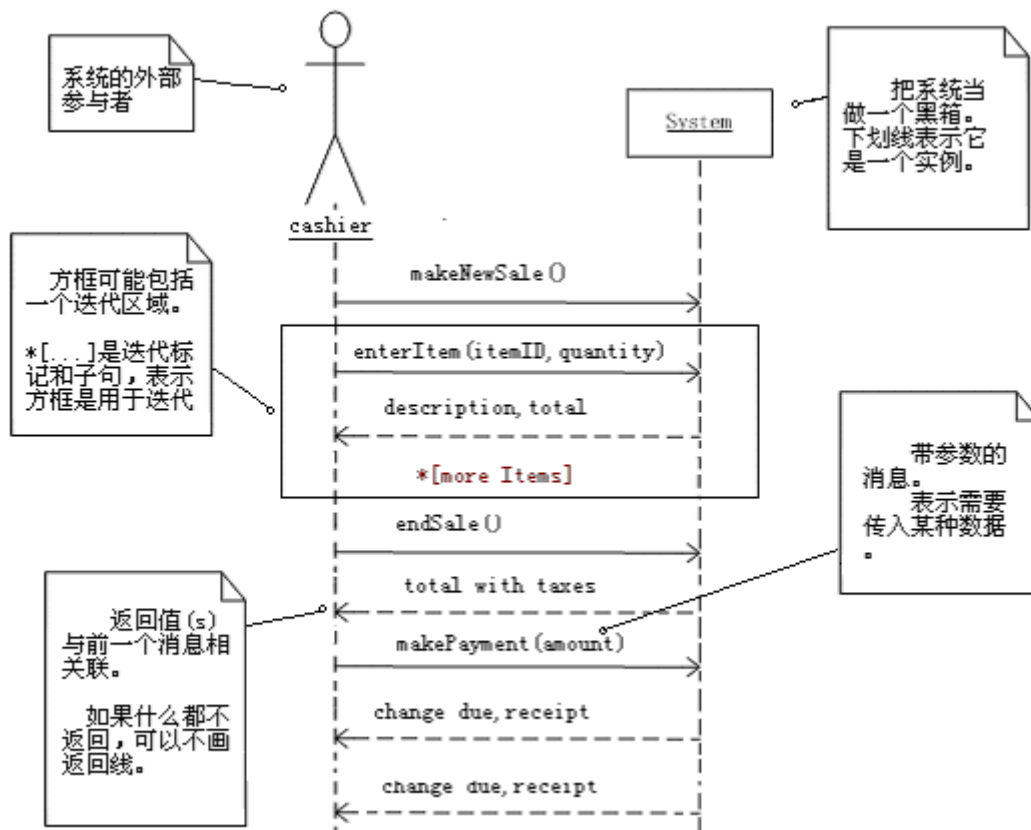
在协作中各独立对象的类元角色。类元角色用生命线表示。当对象存在时，角色用一条虚线表示，当对象的过程处于激活状态时，生命线是一个双道线。

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。

后面会讨论顺序图也可以说明交互软件对象的设计问题。

1) 一个 SSD 示例

下面的例子是用 SSD 显示参与者与系统（作为黑箱）直接的交互，参与者所产生的事件。



2) SSD 和用例

SSD 显示用例的一个场景中的系统事件，所以它产生自用例的考察。你可以直接把它和用例中的步骤对应起来。

3) 系统事件和系统边界

为了识别系统事件，必须象前面用例的讨论一样，清楚的定义系统的边界，由于软件开发的目的是，系统边界经常被定义为软件本身，在这样的语境下，系统事件是直接激活软件的外部系统事件。

在上面处理销售的例子中，由于顾客并没有直接参与 POS 系统的交互，所以顾客不是系统事件的参与者，只有收银员才是。

4) 命名系统事件及操作

因为名称强调事件的命令导向，所以事件以动词开头（add、enter、end、make）可以增加清晰度。

比如：

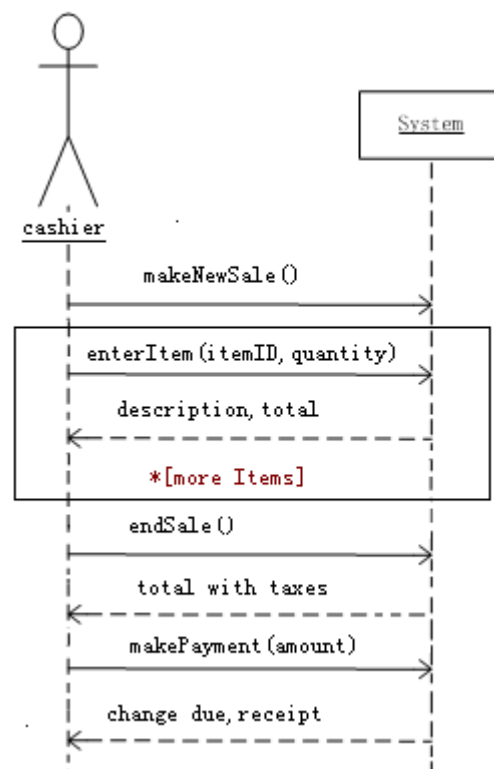
enterItem（加入项目）比 scan（激光扫描仪）要好。

5) 显示用例文本

事实上，这个顺序图是为了更清晰的表达问题，特别是为了更好的和用户交流，所以，很多情况下，希望能在顺序图中显示场景的用例片断，比如：

场景：

- 1, 顾客携带商品到POS机前结账。
- 2, 收银员开始一项新的操作。
- 3, 收银员输入商品的标示符。
- 4, 系统记录卖出商品的描述，价格和数量，多次选带，直到完成。
- 5, 系统提供税后总金额。
- 6, 顾客付款，系统处理支付。



6) SSD 和术语表

SSD 中显示的术语（操作、参数、返回值）是简练的，有的时候可能需要适当的解释，就应该在术语表中表达。

不过，如果讨论是工件的创建而不是编码，又没有必要建立享用的术语比哦澳是个令人怀疑的事情，除非用途和决策支持标明这真正有价值，就是一个不必要的工作。

二、统一过程中的 SSD

SSD 使用例模型中的一部分（用例中交互的可视化），但是在 UP 中并不认为一定要用 SSD 来描述。

不过，有的时候花个几分钟或半个小时创建 SSD 也不是完全没有用处。

一般的情况是：

初始：

在初始阶段一般不会使用 SSD。

细化：

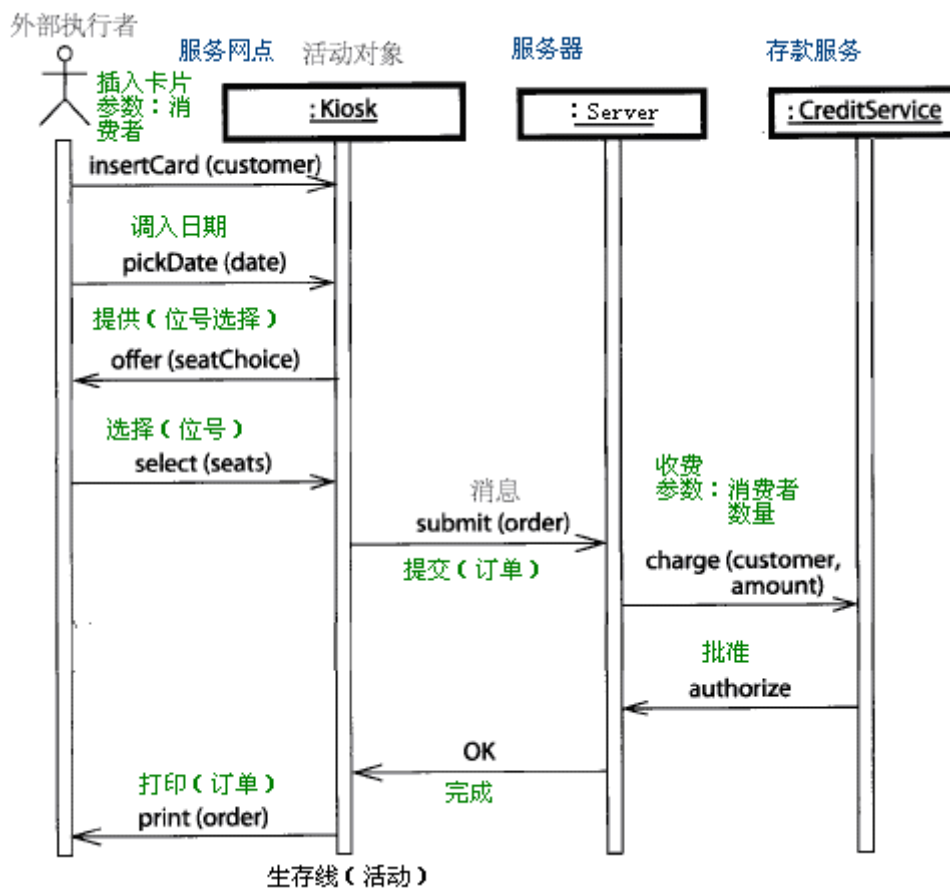
在细化阶段，大多数 SSD 可以被创建出来，用来识别系统事件的细节，以及澄清哪些是系统必须设计来处理的主要操作。

注意：

没有必要为所有场景创建 SSD，一般只是为当前迭代所选择的场景创建 SSD。

三、分析对象之间的行为

当需要更加细腻的分析的时候，可以用更详细的顺序图来分析。下图为带有异步消息的典型顺序图，这是一个服务网点的存款服务的用例。

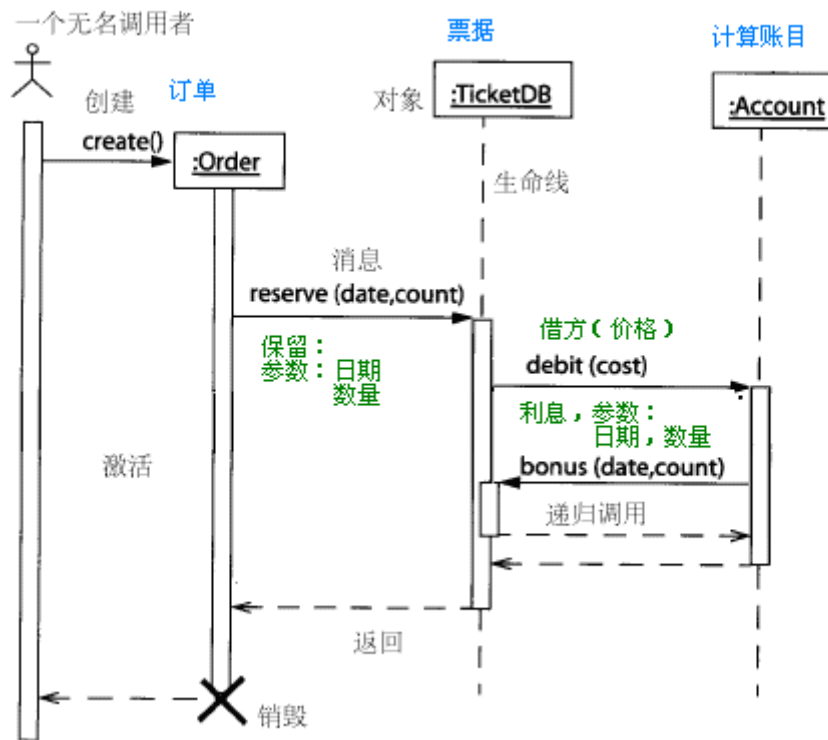


激活：

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。

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。



带有激活的顺序图

四、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。

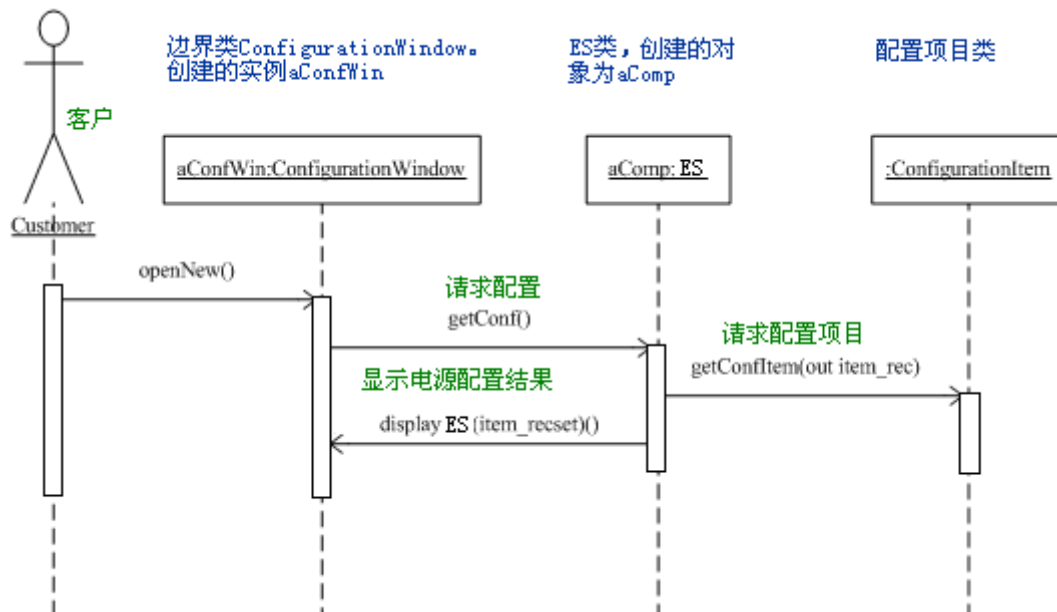
需要表达的交互操作关系如下：

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**。

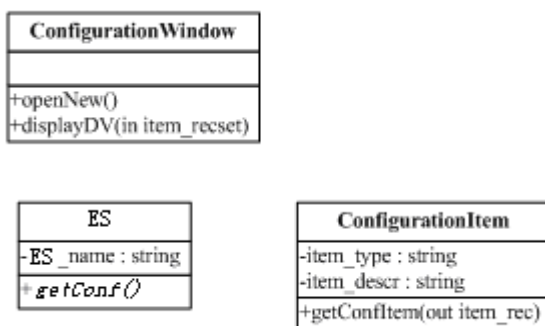
对象 **aConfWin** 需要“显示它自己”以及配置数据，为了这个目的，它发送一个消息 **getConf()** 给 **ConfiguredES** 类或者 **StandardES** 类，并且实例化一个对象 **aComp**。

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_reset**。

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，我们再将“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。

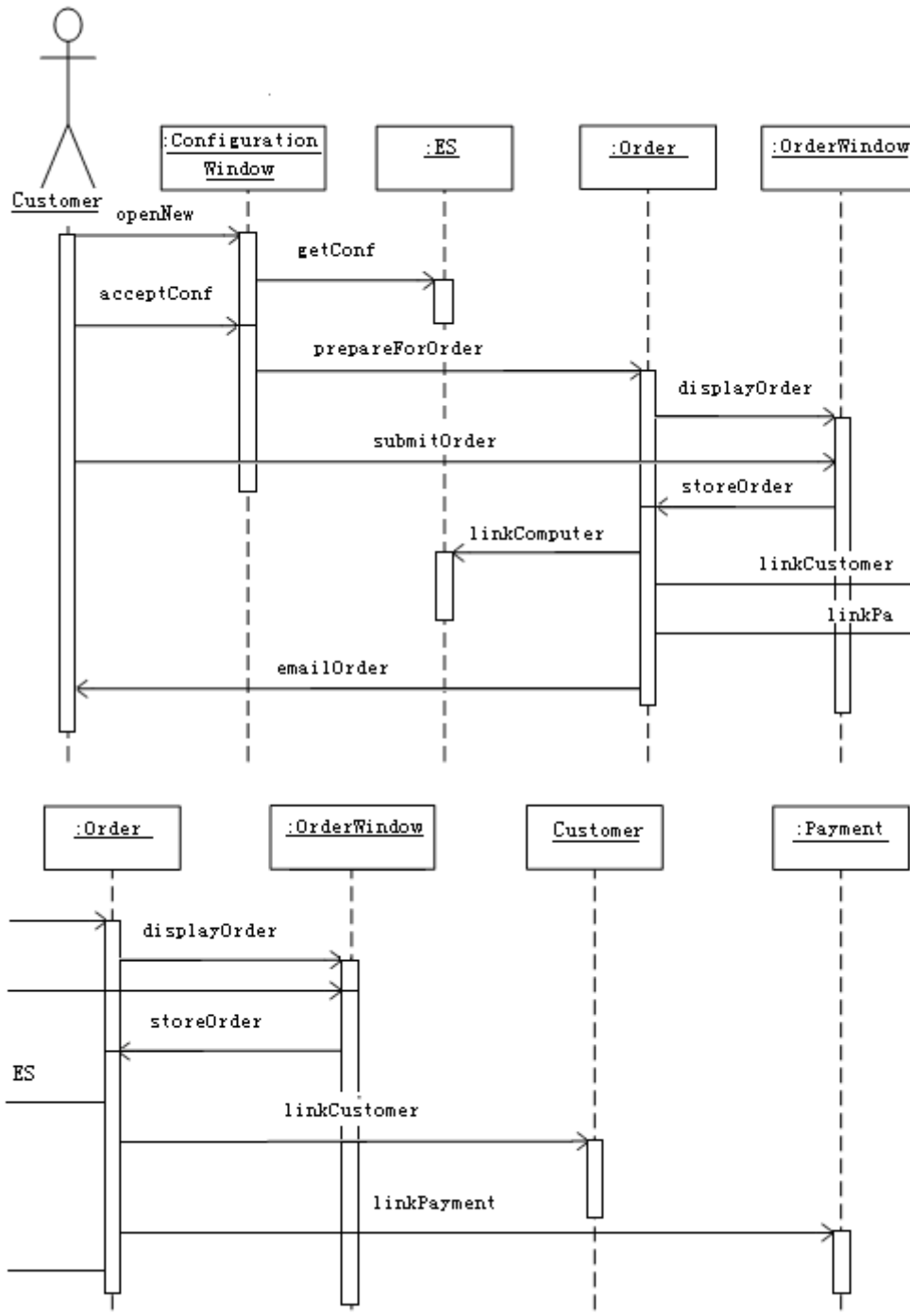
消息 acceptConf 产生发送给:Order 对象的 prepareForOrder 消息，这就创建了一个 Order 对象，它在:OrderWindow 中显示。

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。



五、用操作契约（Contract）增加用例细节

操作契约可以帮助定义系统行为，它是用概念对象的状态改变，来描述系统操作执行的结果。有的时候更详细的系统行为描述也是有价值的。

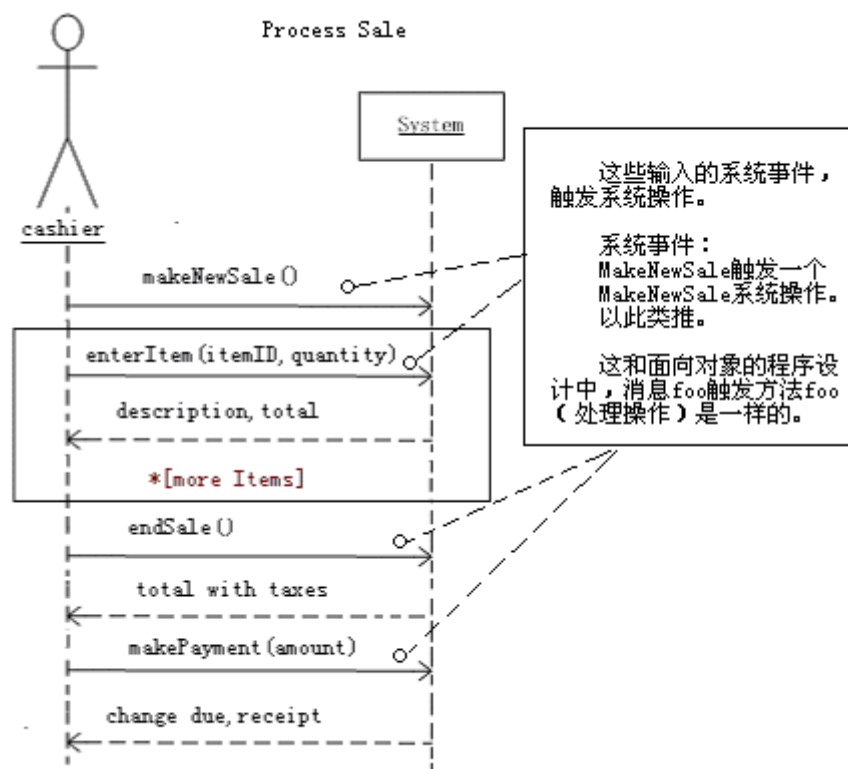
在执行一个系统操作以后，契约根据概念模型中对象的状态变化来描述详细的系统行为。

契约是为系统操作（system operation）而制定的，这种操作，是作为“黑箱”的系统，再它的公共接口中提供的。

贯穿所有用例的全体系统操作，定义了公共的接口。

而系统，作为一个整体，可以用一个类来表示。

我们看下面的例子：



契约 2: enterItem	
操作	enterItem(itemID:itemID,quantity:integer)
交叉引用	用例: Process Sale
前置条件	有一个销售正在进行
后置条件	1, 创建一个 SalesLineItem 实例 sli (创建实例) 2, sli 和当前的 Sale 形成关联 (关联形成) 3, sli.quantity 变成 quantity (属性修改) 4, 在 itemID 匹配的基础上, sli 和 ProductSpecification 形成关联 (关联形成)

1、契约条目的描述

契约名: 操作名	
操作	操作以及参数的名称
交叉引用	[可选]可能发生这个操作的用例
前置条件	在操作执行以前, 概念模型中系统或者对象状态的值得注意的假设, 它们在此操作的逻辑内不会得到测试, 而被假设为真。另外,

	他们并非无关紧要，而是要让阅读者了解有这个假设存在。
后置条件	操作完成以后，概念模型的状态，后面就会详细讨论。

2、后置条件

注意，在前面的例子中，每个后置条件都包含一个分类，比如“实例创建”（instance creation）或者“关联形成”（association formed），这是一个关键点。

后置条件描述概念模型中对象状态的变化，概念模型对象状态的变化包括：实例创建或删除、关联形成或断开、属性改变。

后置条件不是系统操作中要执行的动作，而是系统操作完成以后，为“真”的关于概念模型的声明。

关于关联断开：可以这样描述，“选定的 SaleLineItem 和 Sale 之间的关联就断开了”，还有“当偿还了贷款后，关联就断开了”等。

实例删除非常少见，这是人们一般不太关心强制的销毁一个事务。

后置条件一个特点是，它是声明性的，它描述的是状态变化，而不是一个动作的执行过程。

1) 分析细节

契约以声明状态变化的方式来表达问题，使它成为一个优秀的需求分析工具，在不需要说明操作如何实现的情况下，描述系统操作如何引起系统状态的变化。后置条件可以分析细粒度的信息，并指明系统的状态是什么。

这样就可以很好的分析我们所关心的一些细节问题。

2) 后置条件应该达到怎么样的详细程度

人们往往关心后置条件要书写到怎样的详细程度。

事实上，并不一定需要契约。

但假设需要一些契约，但产生完整、详细、详细、详细的后置条件是不可能也是不必要的。

这就是系统架构师的能力问题，如何抓住重点。

不过，在设计工作阶段，我们会发现一些细微的细节，这不见得是坏事，这些发现可以通知后面的迭代需求工作，这正是迭代开发的优点，一个近期的迭代，可以丰富后一个迭代的调研和分析工作。

经常在创建契约的时候发现，需要记录概念模型中出现的新的概念类、属性或者关联，我们不要受限于先前已经定义的概念模型，在思考操作契约并且有新的发现的时候，可以进一步丰富概念模型。

3、书写契约的指导原则

要创建契约的时候，必须：

- 1) SSD 识别系统操作
- 2) 对于那些复杂的，结果微妙的以及在用例中不清晰的系统操作，可以构造一个契约。
- 3) 要描述后置条件：
 - 实例的创建和删除
 - 属性修改

- 关联形成和断开

一些建议:

- 后置条件的陈述应该采用过去时态的声明语气 (was...), 以强调系统状态的变化, 而不是强调这种变化是如何设计实现的, 比如:

(较好) A SalesLineItem was created

(较差) Create a SalesLineItem

- 不要忘记在新创建的对象和已经存在的对象之间保持记忆关系。这个记忆关系就是两个对象之间的关联关系。例如, 当 enterItem 操作发生的时候, 会创建一个新的 SalesLineItem 实例, 这还不够, 当操作完成之后, 在新创建的实例和 Sale 之间, 还应该建立一个关联关系, 即:

SalesLineItem 和当前的 Sales 建立关联 (关联形成)

注意:

最常见的错误, 就是遗漏了关联的形式, 特别是在创建了新的实例以后, 往往需要和多个对象之间建立关联, 千万别忘了!

六、用协作图分析复杂操作

虽然用顺序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。

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。

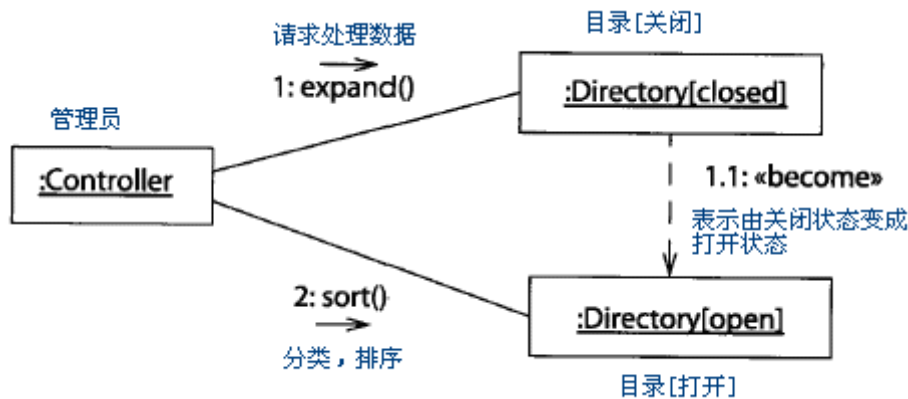
2、流

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, 这就是具有状态类的对象。同一个对象可以表示多次, 每次有不同的位置和状态。

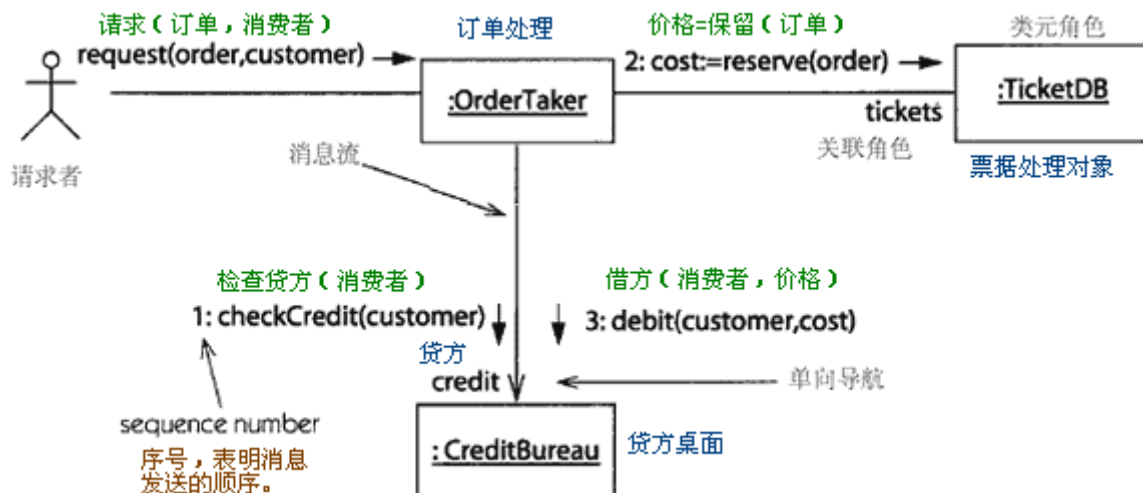
代表同一对象的不同对象符号可以用变成流联系起来。

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)。



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。

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；

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。

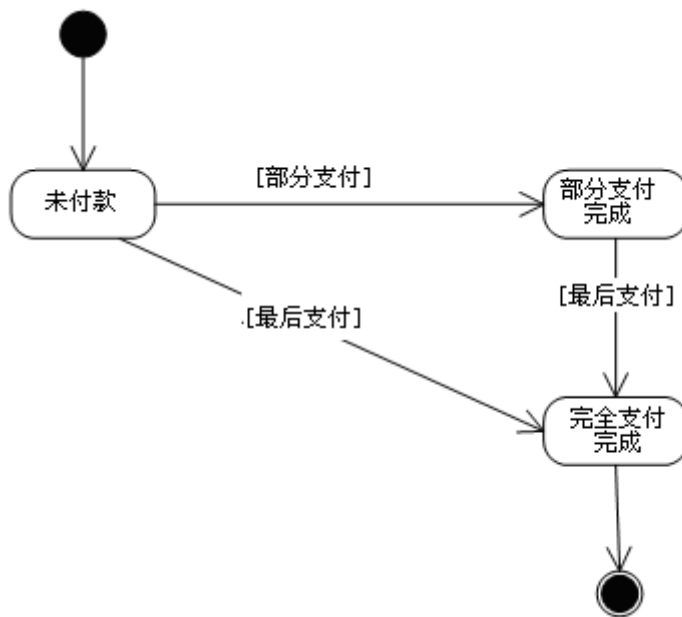
七、状态图及其讨论

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

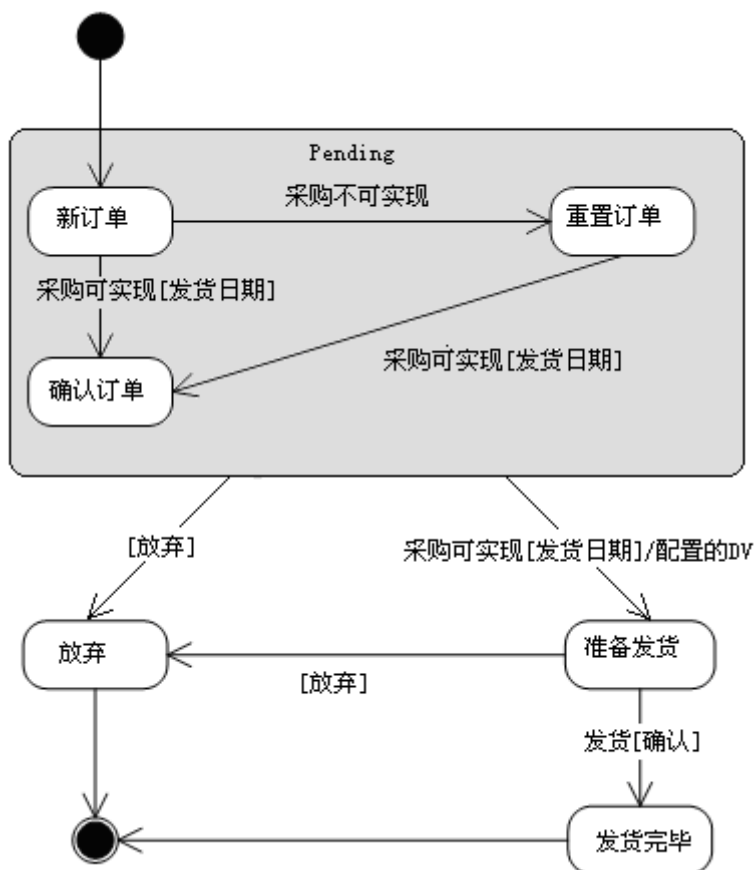
变迁的讨论，但是，有时候在最重要的类上进行这种讨论，也是有意义的，因为这可以为我们下一步系统设计打下了基础，我们来看下面的案例。

案例：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，看起来，这对细腻的描述订单的状态变迁是很有意义的。



第七节 迭代计划和风险管理

我们现在再把注意力回到迭代开发上来，迭代计划和项目管理是一个大问题，但为了能够实用，我们必须把问题简化，讨论一些关键问题。

所有的问题都集中在：

- 在下一个迭代中该做什么？
- 如何在迭代开发中跟踪需求？
- 如何组织项目工作？

一、区分需求的等级

1) 早期迭代的驱动因素：

风险、覆盖范围、重要性和技能发展。

在最早的迭代中，要根据风险、覆盖范围、重要性来组织迭代。

需求风险包括技术复杂度和其它因素，如工作方向的不确定性、拙劣的规范说明，行政问题或者可用性。

覆盖范围：

早期迭代至少要涉及系统的所有主要部分，可能还要充分或粗略的实现许多组件。

重要性：

具有很高业务价值的的功能，即使没有技术风险，在早期迭代中至少部分实现中要场景所需的主要功能。

有些工程的另一个驱动因素是技能培训，即帮组开发人员掌握新技术，在这些工程中，技能

培训是高优先级的驱动因素。

2) 如何进行等级划分

UP 是用例驱动的，所以具体实施是对用例进行等级划分。

另外，有些需求被描述为和特定用例无关的高优先级特性，它可能会跨越几个用例。这时需要描述在用例的补充规范中。

因此，优先级列表要包括用例和高层特征两个部分。

优先级列表		
需求	类型	优先级
Process	用例	1
Logging	特征	2
.....

3) 优先级划分的小组定位法

优先级可以是定性的，可以采取小组定位法。

可以在小组会上投票，事实上每次做迭代计划的时候都可以进行这种投票活动。

4) 优先级划分的定量法

需求和风险优先等级的划分，可能小组计点投票就已经足够了，不过这是模糊的定性方法，如果需要进行更定量的思考，需要进行以权重为基础的分类。

架构权重分析表					
需求	类型	AS	风险	重要性	权重和
Process Sale	UC	3	2	3	15
Logging	Feat	3	0	1	7
Handle Returns	UC	1	0	0	2
.....		
说明	<div>名称</div> <div>权值</div> <div>范围</div>				
	AS: 架构上的重要性		2	0 - 3	
	风险: 技术、复杂、创新		3	0 - 3	
	重要性: 早期较高的业务价值		1	0 - 3	

二、划分项目风险等级

划分全部项目风险等级的一个有效的方法，就是从成本、时间或者工时上估计它的发生频率和影响力。这种评估可以是定量的（通常需要深入的思考），也可以是简化定性的（小组讨论、投票，简单的分成：高、中、低三级）。

最糟糕的风险，是那些很可能发生又影响很大的风险，例如：

风险	可能性	影响程度	缓解方法
----	-----	------	------

缺乏熟练的面向对象开发人员	高	高	<ul style="list-style-type: none"> ● 阅读书籍 ● 雇用临时的咨询顾问 ● 课堂教学或者培训辅导 ● 两人一组编程
演示程序不满足即将产生的 Hamburg 的 POS 规范	中	高	<ul style="list-style-type: none"> ● 雇佣拥有 POS 系统开发经验的临时顾问。 ● 识别演示程序中已经实现的最严重的需求，并安排较高的优先级。 ● 最大限度的利用以完成的组件。

三、在迭代之间跟踪需求

我们已经很清楚，迭代开发中并非所有的场景都在第一次迭代中实现，一个复杂的场景，往往要花 6 个月采用多次为期两周的迭代来完成。每次迭代都处理新的场景或者场景的某些部分。在这样的情况下，必然出现了一个需求跟踪的问题，人们如何记录用例的哪些部分已经完成，哪些部分正在进行中，哪些部分还没有涉及呢？

这就需要用到需求工具。

Rational 的 RequisitePro 是一个需求跟踪工具，值得我们花时间掌握它来跟踪在迭代之间已部分完成的用例。

我们可以把 RequisitePro 和 Microsoft word 整合到一起，用 word 编辑需求，在 RequisitePro 中选择短语，并把它定义成被跟踪的需求。每个需求都有多种属性，如状态、风险等，RequisitePro 工具可以使我们可以在迭代之间跟踪用例的部分完成情况。这个工具的使用很简单也很有效。

第八节 领域建模的实例分析

一、如何通过领域模型来发现出类及其关系

建模实例一：我们使用一个猜数游戏来说明如何建立领域模型问题：输入一个数，如果猜中了显示“你猜中了啊”然后程序结束，如果猜的不对，系统则告诉你的数是太大还是太小，然后要求你重新输入新的数，直到猜中为止。

(1) 开始归纳问题——其实是描述出用例的事件流

+	-----	+
	系统应该准备一个正确答案	
	玩家可以输入一个答案	
	系统应该比较玩家输入的答案和正确答案	
	系统应该显示玩家每次输入的结果	
+	-----	+

在归纳问题时，要注意把握下面的几个基本要点

- **第一是不要涉及内部的流程**，别出现“如果输入不正确，就怎么怎么样”的句子，这些并不是正确的问题，正确的问题必须是明确的，清晰的，如果可能的话全部按照“什么可以干什么”的格式来写。

- **第二是不要一开始就进入细节**，包涵太多细节的问题将会是一个长长的清单，这种清单根本没什么用。尽量从最高一层分析，但也不要简单到“用户可以玩游戏”这种笼统的问题。
- **总之一一个原则是全面、清晰、明确**。要做好问题域分析完全取决于设计师的水平与能力，这就不是可以简单的看看书能达到的了。

(2) 获得名词列表-----为发现出类提供信息

把问题清单中的名词都提出来，得到一个名词列表，这就是类的来源（不过不忙，这只是初步过程）

+-----+		
	系统	
	玩家	
	正确答案	
	答案	
	游戏结果	
+-----+		

(3) 筛选名词-----除掉无关的名词

但要注意的是，不是名词列表中的所有的名词都能作为类的，接下来需要进行筛选。

- 玩家是参与者，应该放到用例图上去
- 系统太笼统，不能成为一个对象的名称
- 答案和正确答案容易混淆，但称为输入答案又容易被误解成一个动作，干脆叫做玩家答案
- 结果不明确，察看前面的需求，应该分解成错误信息和完成信息

(4) 根据名词列表发现出其中的类

最后，对前面的名词列表进行筛选完毕后，得到一个下面的名词列表

+-----+		
	正确答案	
	玩家答案	
	错误信息	
	完成信息	
+-----+		

在这个列表中缺少了系统，显得太单薄，回过头再仔细察看需求，应该引入一个游戏引擎，由它来充当调度者（控制类）。同时，我们再引入一些 Helper 类以实现游戏的交互（边界类）

+-----+		
	游戏引擎	
	正确答案	
	玩家答案	
	错误信息	
	完成信息	
	游戏的交互	
+-----+		

从而发现出问题域中所隐藏的分析类。但要注意的是，在这个阶段中所发现出的类，并不是最终要实现的类，而是一个分析类。利用分析类的主要目的是帮助设计师理清系统的关系，将需求文稿中错综复杂的关系整理成一个脉络清晰的完整系统。根据规则，应该包含有三种分析类：

边界类、实体类和控制类。在本问题中，它们分别是：

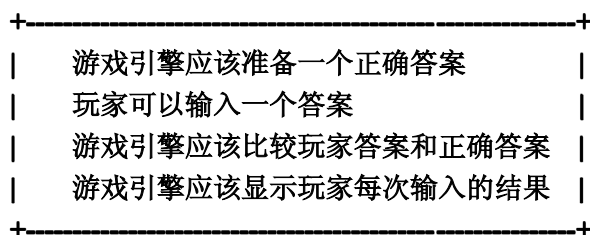
边界类：游戏的交互

实体类：正确答案、玩家答案、错误信息和完成信息

控制类：游戏引擎

(5) 进一步修改前面的问题域，以获得更清晰的需求描述

同时修改前面的问题域，现在系统已经明确是一个游戏引擎。这种替换当然是一种理想情况，通常都会发生分解和关联，那时候需要扩充问题域，有时候还需要建立新的问题域。



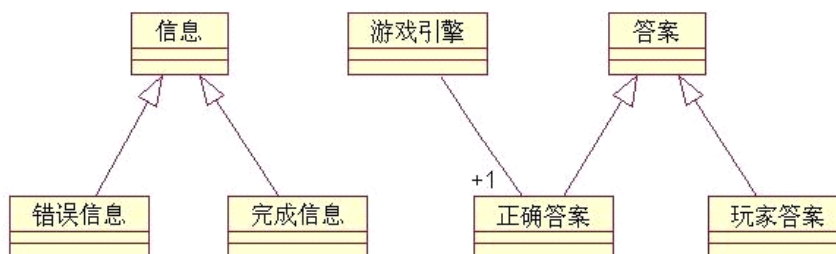
(6) 分析类的层次（纵向关联）

在前面，我们已经初步对问题域进行了分析，获得了一个类名的名词列表，接下来的工作则是要绘制这些分析类的层次。现在我们要用我们的专业知识来归纳类了。很明显，错误信息和成功信息需要一个基类，玩家答案和正确答案也是一样。

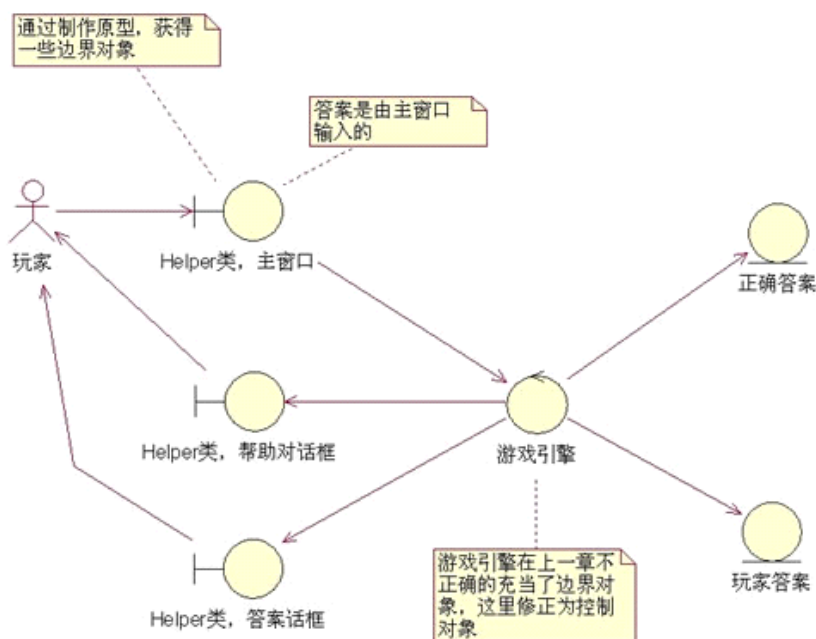


(7) 分析类之间的关联（横向关联）

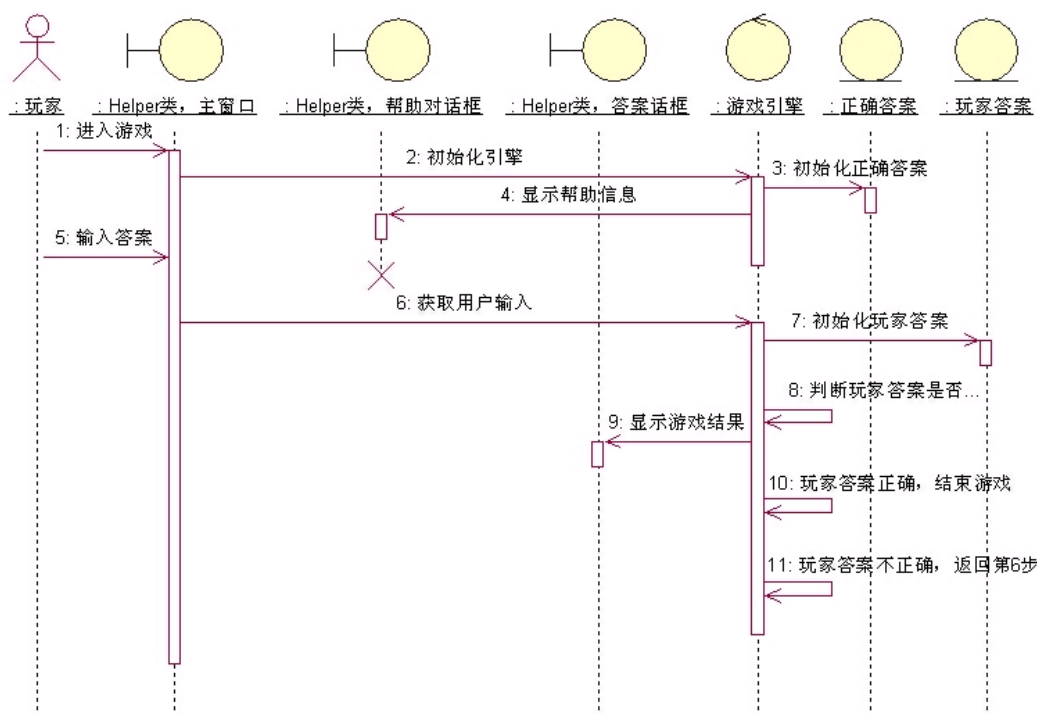
通过对项目中的类层次进行分析和描述以后，接下来则是要分析类之间的关联，同样我们也应该从需求文档和问题域中寻找线索。在本问题中，“游戏引擎应该准备一个正确答案”，即“游戏引擎”与“正确答案”之间有关联。



(8) 最后，设计出本问题例的类的分析图（关系说明——静态分析）



(9) 设计出游戏交互的顺序图——动态分析



(10) 设计出类中的属性和方法——进一步细化分析类以最终产生出设计的类

类的关联在类图已经完整地设计出后，应该为类分配方法和属性了，这将是以后的设计阶段的工作。

二、建立领域模型总结

(1) 可以从用例的事件流开始，看看事件流中的名词与名词短语，找出大量重要的域对象

以获得所要用到的类（抽取对应于业务实体或事件的名词），然后再分析这些类的属性、操作和它们之间的关系。

（2）将名词进行分类、抽取合适的类：同时将名词与名词短语成为对象与属性，而动词与动词短语成为操作与关联。

（3）审查类及属性

- 是否在系统责任之内
- 是否描述类对象的特征
- 是否存在冗余
- 是否有复杂结构的属性
- 根据对需求的理解进行细化

（4）所要注意的几点

● 素描类的特性

需要强调的是，在这一阶段对特定领域类的描述具有一定的素描性质。也就是说特定领域类的操作和属性不一定与最终实现时的定义一致。

因为此时还没有涉及到系统功能的具体实现，不可能准确、完整地定义它们。有一些操作需要在设计阶段细化时才能确定。

● 动态行为的描述

此外，为了描述领域类的动态行为，可以使用 UML 中的任何一种动态图(如顺序图、活动图、合作图、状态图)来描述。

三、建模实例二：某一网站领域模型的建立例

（1）用户所罗列出的一些需求

- 我需要做一个网站
- 我的文章、我最近的活动（新闻）、下载和留言
- 我希望把我平时的文章发布出来给用户看
- 我希望把我最近做的讲座放到我的活动栏目中来进行宣传
- 我还会把我的一些源代码和讲座录像打包发出去
- 我希望用户给我留言

（2）需求分析

● 功能性需求

- ✓ 网站可以提供我平时写的文章的链接
- ✓ 网站可以将我最近的活动情况发布出来
- ✓ 网站能够将我平时的源代码和讲座录像打包发出去
- ✓ 网站能够接受用户给我的留言

● 非功能性需求

网站要简单、美观和大方；浏览的速度尽可能快。

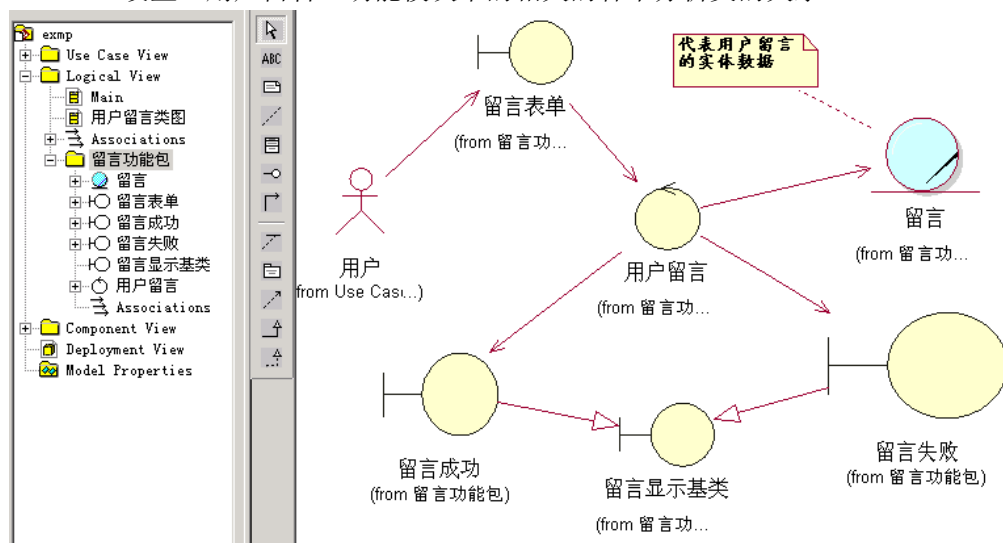
（3）找出名词短语——领域模型

网站	平时文章	最近活动新闻	平时源代码	讲座录像	用户留言	管理员	文章
讲座	活动	新闻栏目	源代码	讲座的录像包	浏览者	留言	

（4）发现类及类之间的关系

- 创建“用户留言”功能模块的包

- 在该包中分别添加各个分析类
- 设置“用户留言”功能模块中的相关的各个分析类的关系

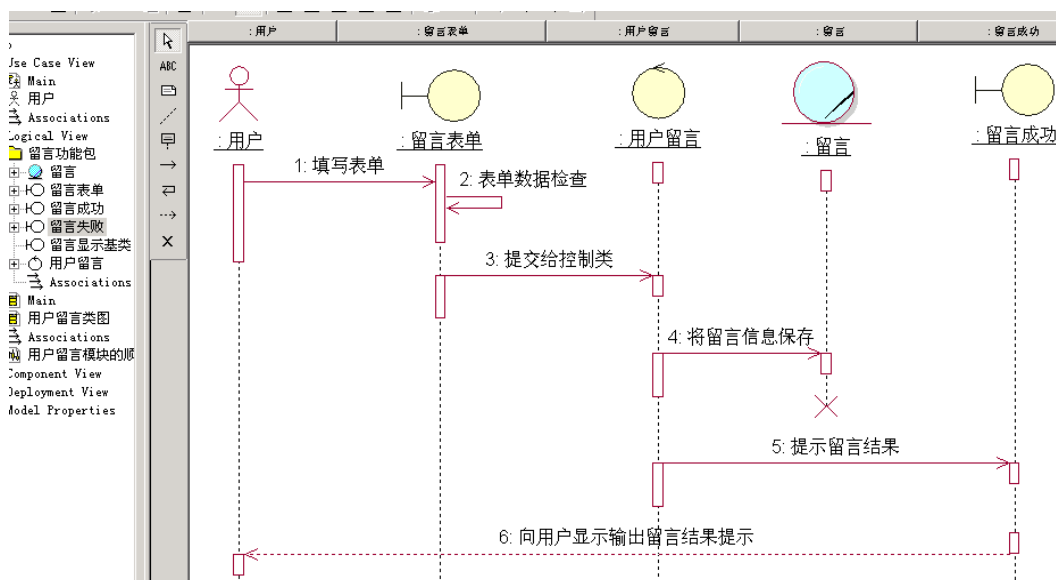


- 给某个类设置其属性和方法
- 再设置其属性的数据类型和访问限制
- 给该类添加方法
- 设置“用户留言”功能模块中的交互的顺序图——动态分析

名称为“用户留言模块的顺序图”

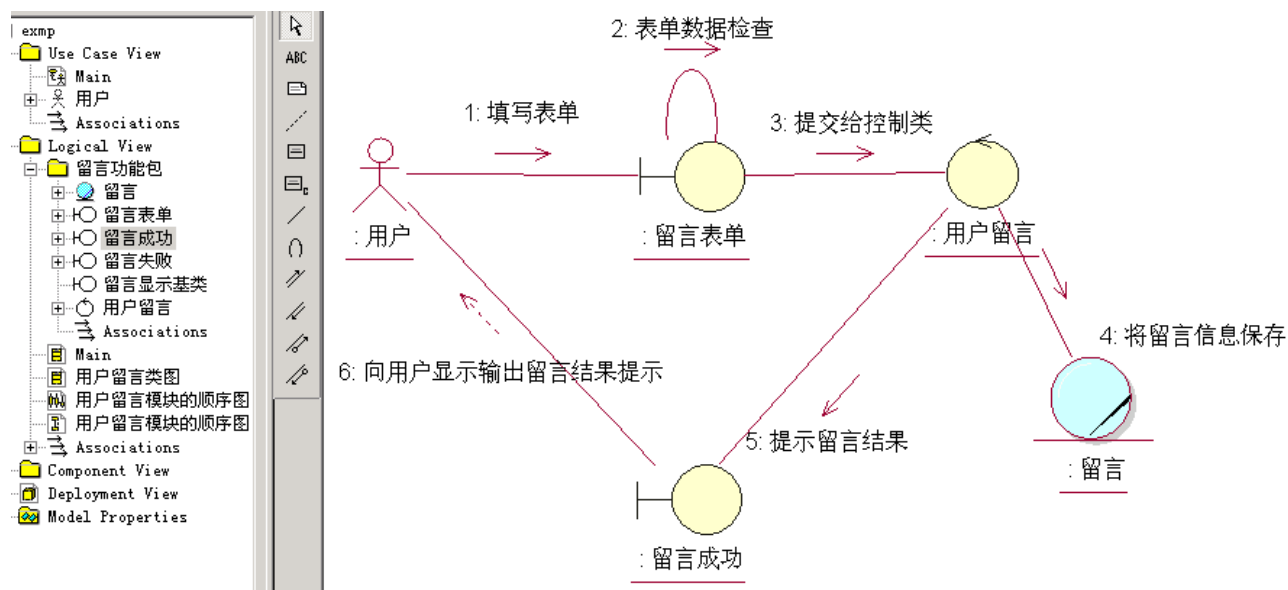
分别从用例图中拖动参与者“用户”和从“留言功能包”中拖动“留言表单”、“用户留言”和“留言”、“留言成功”等类到视图中。

然后再分别设计其消息的发送和返回的次序。



注意：在上面的顺序图中所出现的各个消息没有映射成对应类的操作方法，如“填写表单”消息没有映射成“留言表单”类的操作方法。因此在后面进行用 Rose 生成代码中进行“检查模型”时会出现错误。

- 根据顺序图创建出协作图



四、建模实例三

下面给出“铁路呼叫中心”项目的功能性和非功能性的需求，从而获得“问题域”中的相关的类；

(1) 呼叫中心项目的功能性需求

旅客应该能够通过系统进行车次信息的相关查询
 旅客应该可以通过系统进行相关车次余票的查询
 旅客应该能够通过系统进行订票
 旅客应该能够通过系统提供投诉意见
 旅客应该可以通过系统进行行包的办理
 系统应该可以取消订票
 系统应该可以记录旅客的操作步骤
 系统语音提示应该尽可能的简单清楚
 系统应该可以记录对方用户的电话号码等信息
 系统应该具有少量的人工坐席和大量的自动坐席

系统应该能够及时的响应客户订票，并通知订票结果
 系统应该可以自动地给旅客分配一个订票号和相应的订票密码
 系统应该可以按照车次查询时刻表
 系统应该可以按车次查询到站时刻
 系统应该可以根据始发站和到达站或途径站的车次信息

系统应该可以为团体客户提供团队订票业务
 系统应该可以为旅客提供原有订单的查询

(2) 呼叫中心项目的非功能性的需求

订票流程应该尽量的简单
 订票流程的操作时间应该尽量的短
 系统应该能够控制订票操作的时间
 系统应该能够取消故意骚扰系统的电话
 系统应该有预留接口可以方便的连接到其他的客服服务电话号码中去

(3) 找出名词短语——领域模型

旅客 呼叫系统 车次信息 车次余票 火车票 投诉意见 行包 操作步骤
 语音提示 用户信息 人工坐席 自动坐席 订票结果 订票号 订票密码 时刻
 表 到站时刻 始发站 到达站 途径站 团体客户 订票业务 订单

(4) 发现出类及类之间的关系

五、建模实例四

下面给出“网上订票”需求项目的功能性和非功能性的需求，从而获得“问题域”中的相关的类；

(1) 网上订票项目的功能性需求

系统应该可以允许用户进行注册
 系统应该可以允许进行登陆
 系统应该可以允许用户修改自己的个人信息
 系统应该可以允许进行网络在线订票
 系统应该可以允许可以根据车次查询列车时刻表
 系统应该可以根据始发站和到达站或途径站的车次信息
 系统应该可以查询票价
 系统应该可以查询既有订票信息
 系统应该可以取消订票
 系统应该可以查询晚点信息
 系统应该可以提供团体订票业务
 系统应该可以允许在线支付票款
 系统应该可以查询余票信息

(2) 网上订票项目的非功能性需求

网络响应速度应该尽量快（订票流程的操作时间应该尽量的短）
 用户填写的信息应该尽量的少，采用菜单选择和勾选方式（订票流程应该尽量的简单）
 系统应该有预留接口可以方便地连接到其他的客服服务电话号码中去

(3) 找出名词短语——领域模型

系统 用户 个人信息 火车票 车次 时刻表 始发站 到达站 途径站 票
 价 既有订票信息 晚点信息 团体订票业务 票款 余票信息

(4) 发现出类及类之间的关系

六、建模实例五

下面给出“ATM系统自动售票系统”需求项目的功能性和非功能性的需求，从而获得“问题域”中的相关的类；

(1) ATM系统自动售票系统的功能性需求

ATM系统应该可以接受纸币
 ATM系统可以接受信用卡
 ATM系统可以检验纸币的金额
 ATM应该可以验证信用卡的密码
 ATM系统应该可以进行伪钞的识别
 ATM系统必须可以卖票
 ATM系统可以验证购票的金额与信用卡的金额的比较
 ATM系统可以找零钱
 ATM系统应该可以订票？
 ATM可以打印交易的收据凭票
 ATM系统可以吐票
 ATM系统可以查阅车次信息
 ATM系统可以查询车票的席位余额
 ATM系统应该可以对帐
 ATM系统应该可以查询钱箱中的余额
 ATM系统应该可以取消购票
 ATM系统应该可以记录操作步骤
 ATM系统应该用触摸屏

(2) ATM系统自动售票系统的非功能性需求

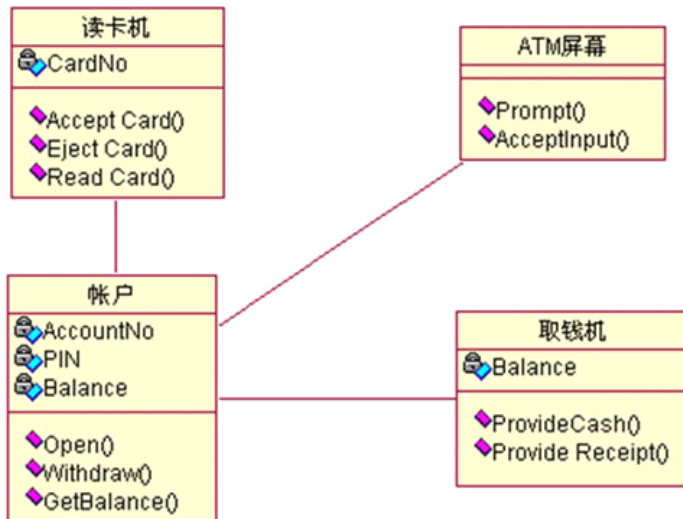
ATM系统应该可以语音提示
 ATM系统应该可以对于什么什么进行报警？
 ATM系统购票过程应该操作步骤简单
 ATM系统必须有监视功能
 ATM系统应该适合不通身高的人购票

(3) 找出名词短语——领域模型

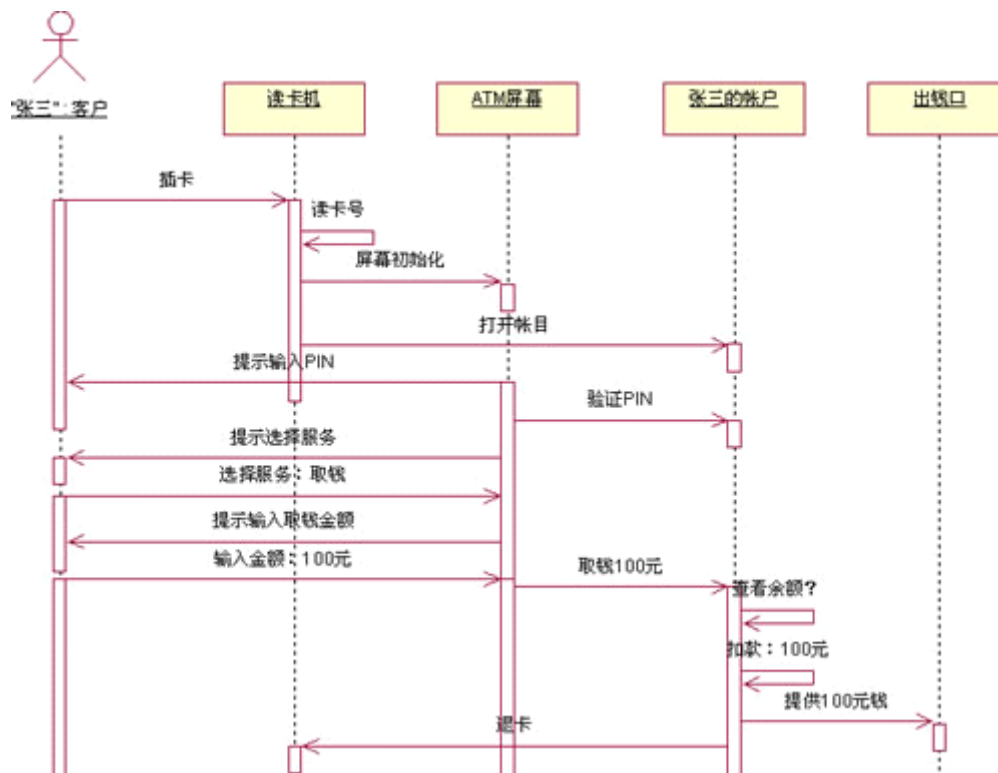
ATM 系统 纸币 信用卡 纸币的金额 信用卡的密码 伪钞 票 购票的金额
 零钱 打印凭条 车次 帐 钱箱的余额 操作步骤 触摸屏

(4) 发现类及类之间的关系

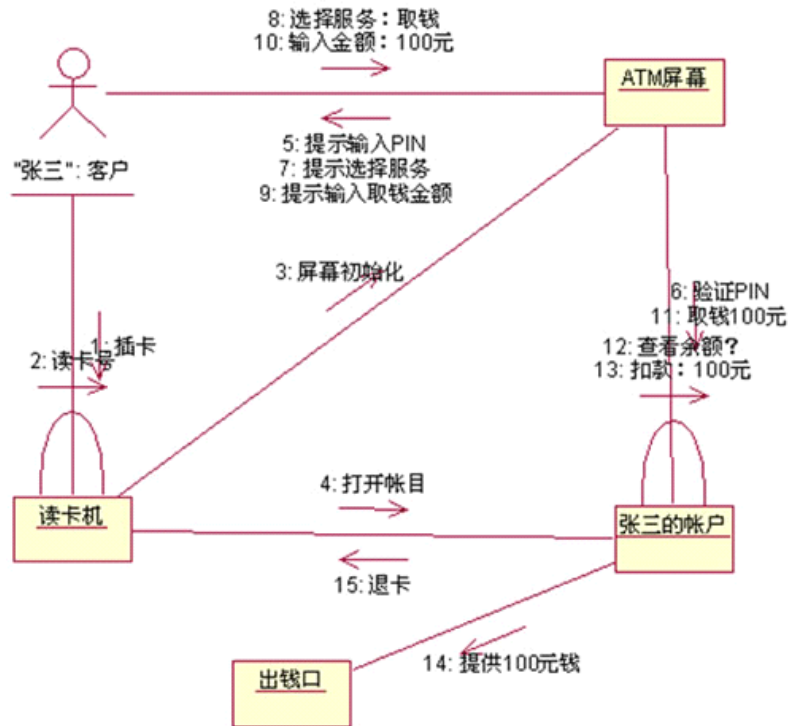
ATM系统的取钱使用案例的Class类图



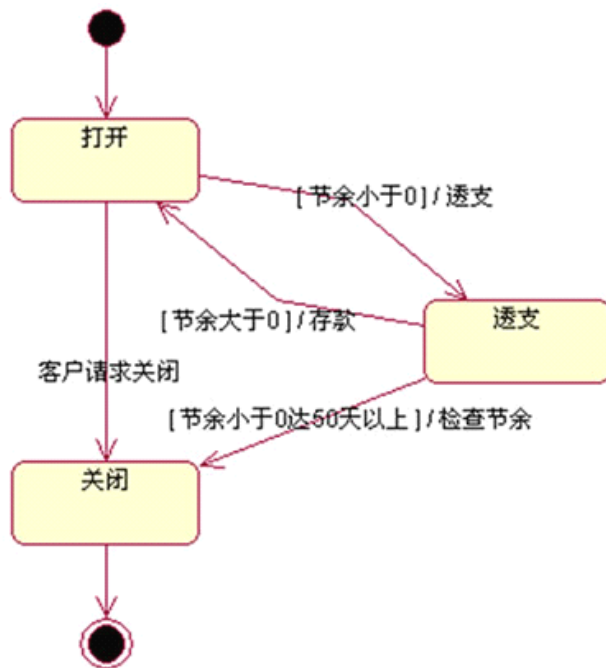
张三取款的顺序图



张三取100元钱的协作图



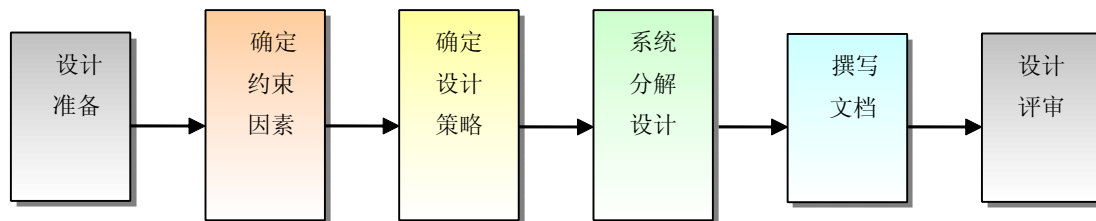
张三取款的状态图



第四章 高层软件架构的设计

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》。

这个阶段是系统架构师发挥作用的主要位置，高层架构设计过程设计流程如下。



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领域进行组织、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。

设计也称之为架构设计，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。

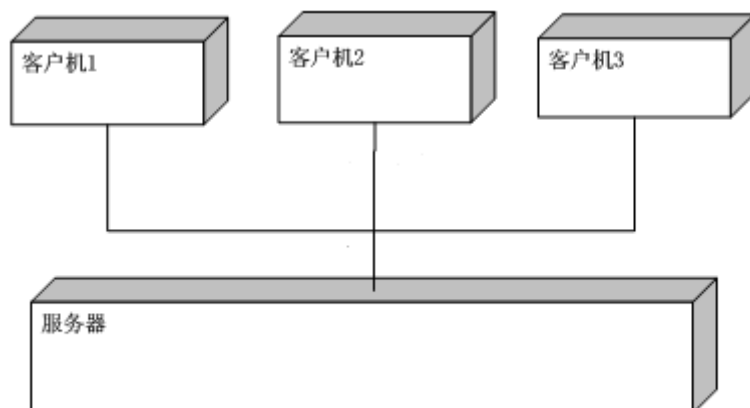
一般来说架构设计是一个高度技术的工作，一般不需要涉及太多的用户，但需要系统分析人员和部分开发人员参与。因为系统设计的输出就是开发的蓝图。

下面讨论在这一阶段一系列的原则和思想。

第一节 高层软件架构的规划

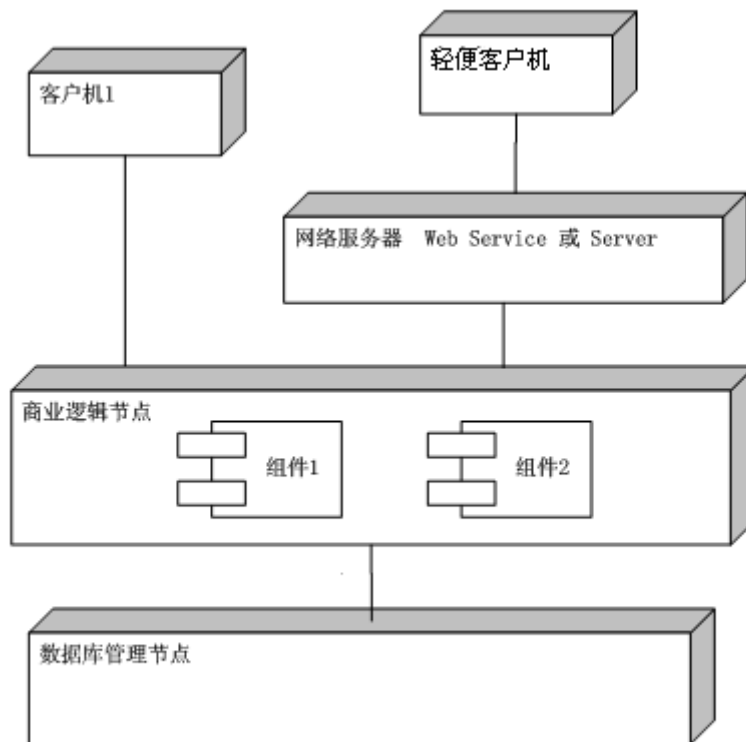
一、客户服务结构（C/S architecture）

这个结构可以用部署图来表示。

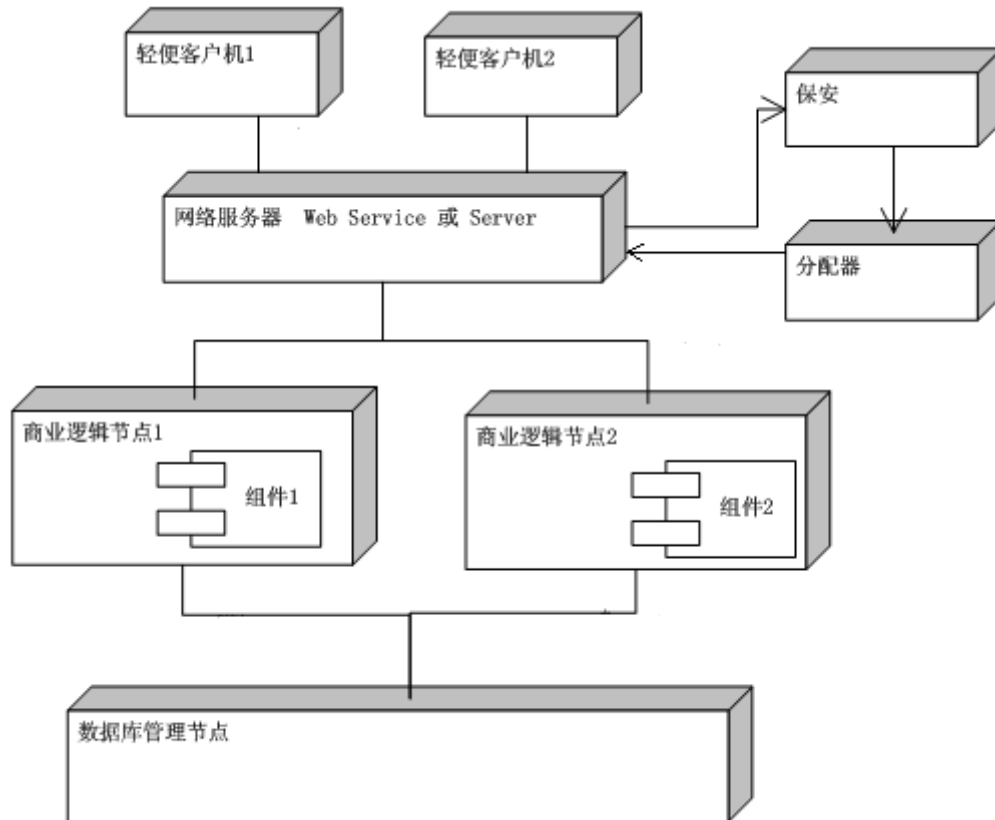


二、多级体系结构（four-tier architecture）

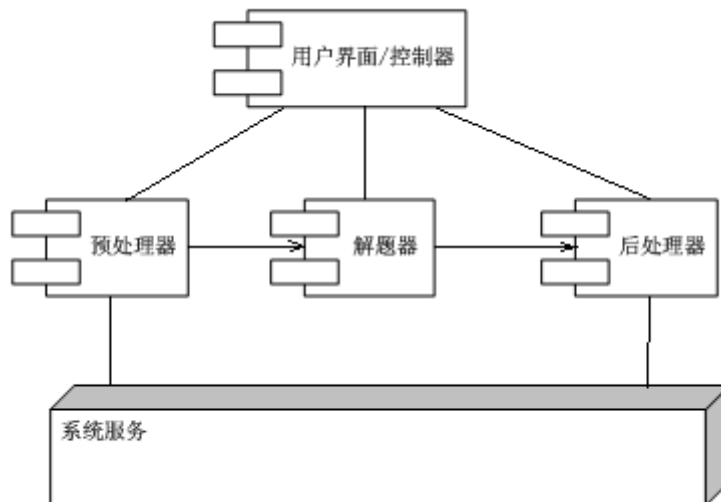
这里使用了组件图和部署图。



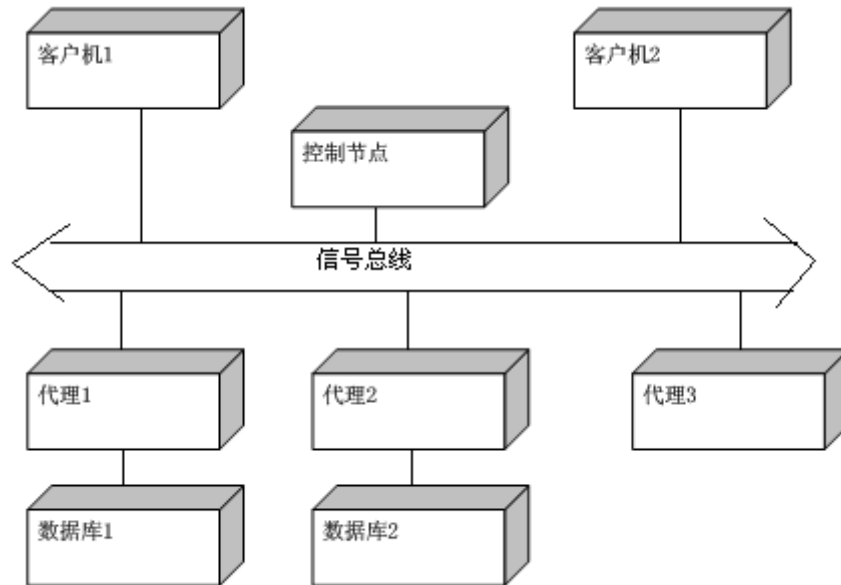
三、多级体系结构（串行法和团聚法）



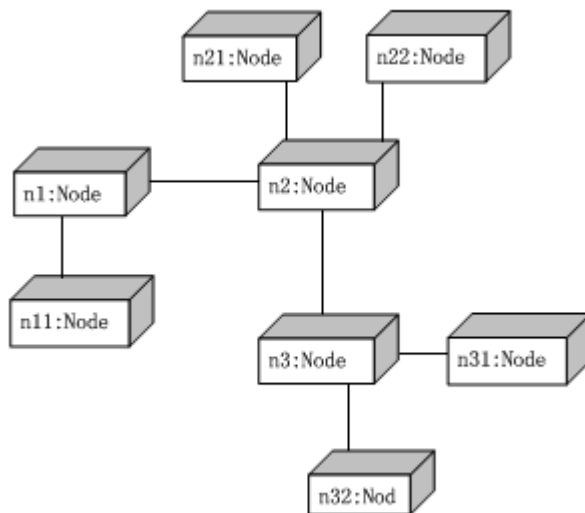
四、流处理体系结构（procedural prcessing architecture）



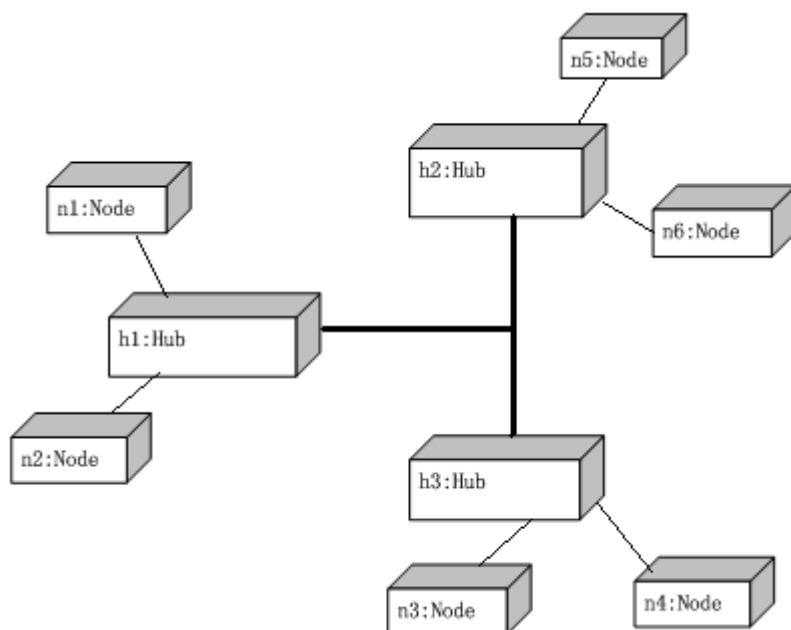
五、代理体系结构（agent architecture）



六、聚合体系结构（**aggregate architecture**）



七、联邦体系结构（**federation architecture**）



第二节 面向过程的架构设计

面向过程的架构设计，又称之为结构化设计。

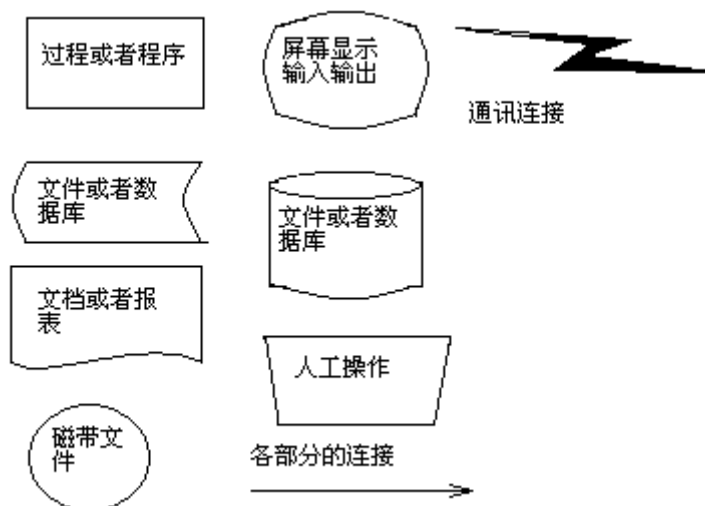
它使用“输入 - 处理 - 输出”这样一个基本模型，这些模式比较适用于描述商业软件，它们中大多数依靠数据库或者文件，并且不太需要复杂的实时处理。

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。

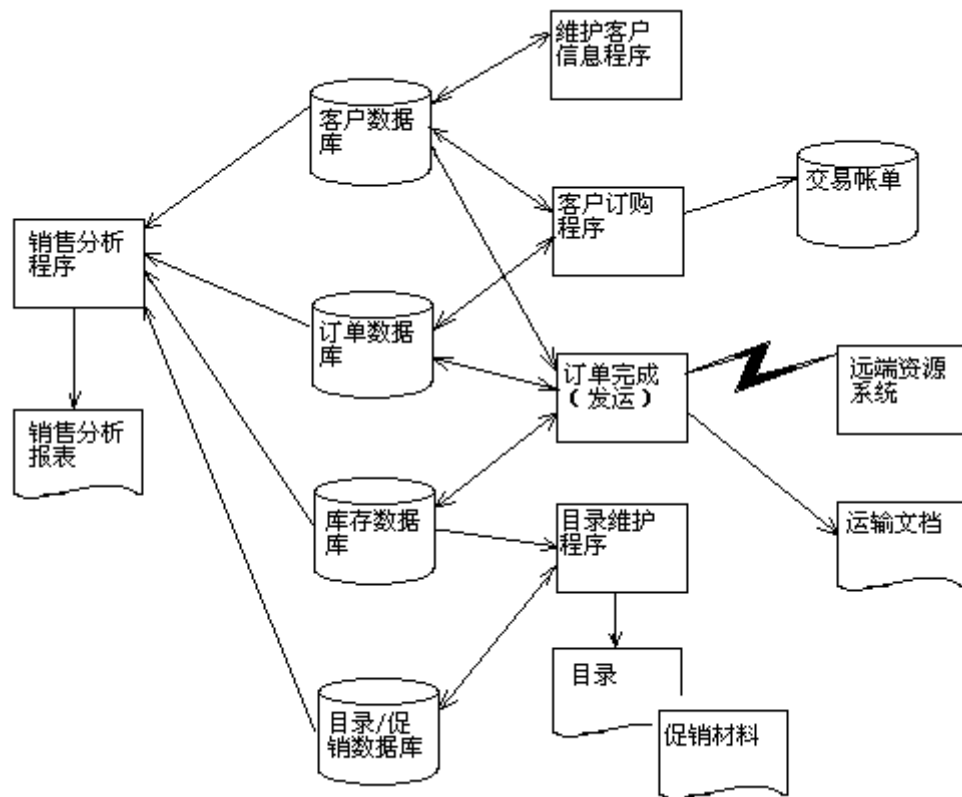
一、系统流程图

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。

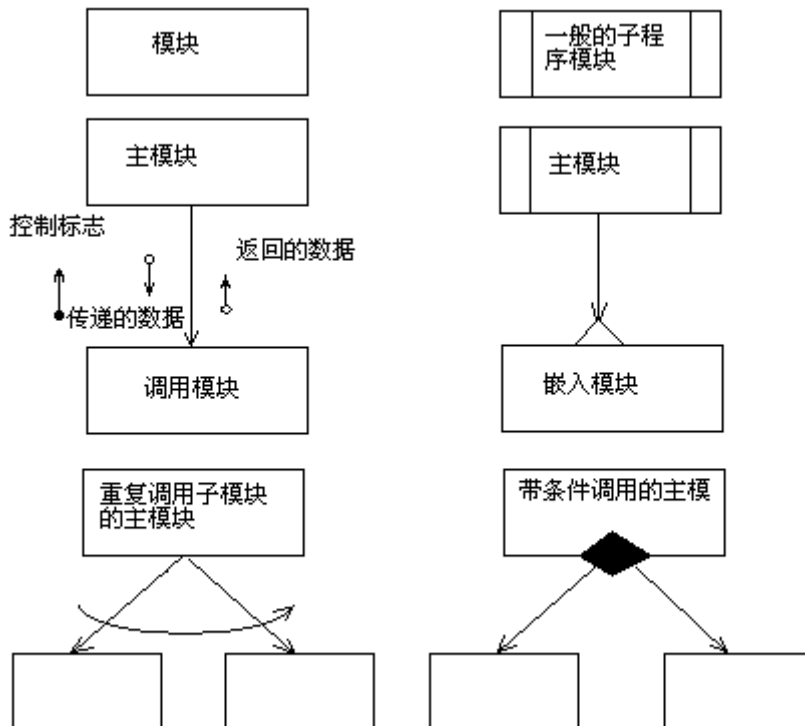


下面是一个销售系统的流程。

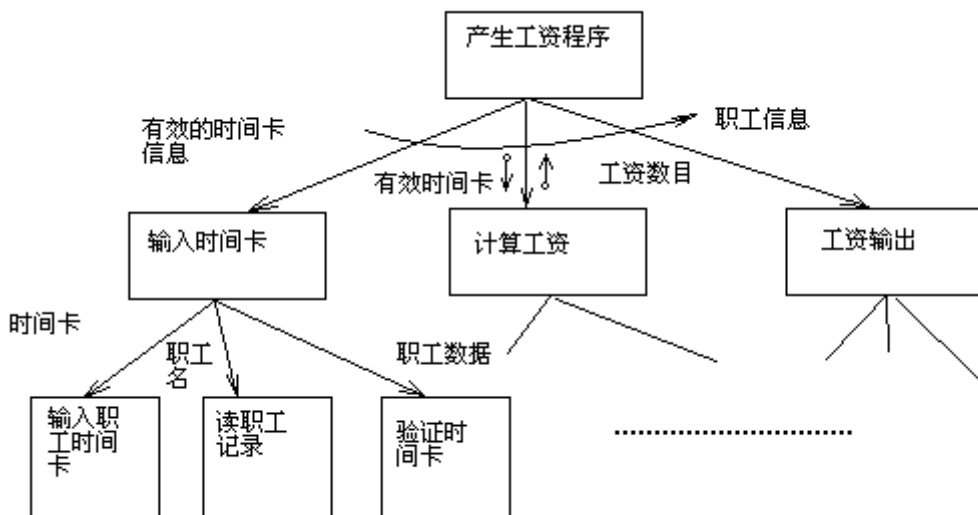


二、结构图及其应用

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。结构图的主要符号如下：



下面是一个工资系统的部分结构图。



三、模块算法设计（伪码）

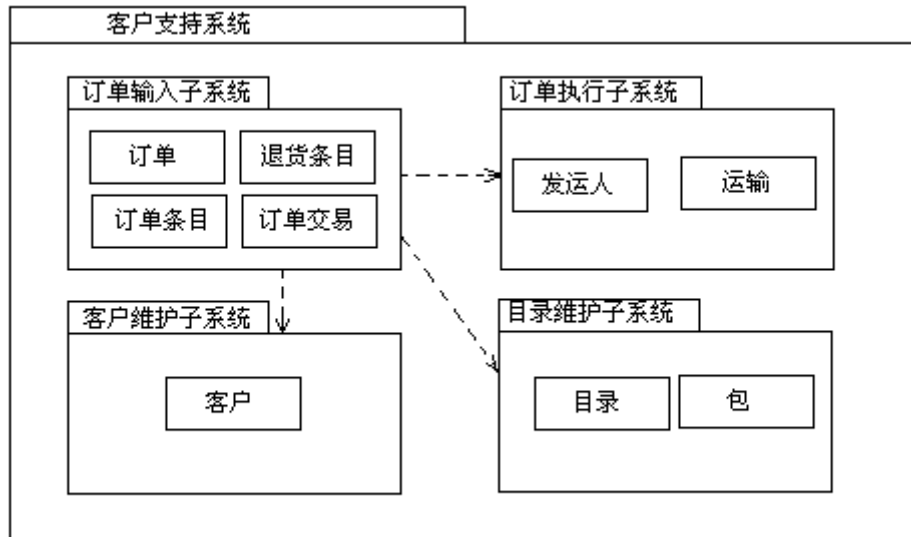
结构化设计的另一个需求，是描述每个模块的内部逻辑，我们可以用自己熟悉的语言来定义伪码（比如 C），使用伪码并不是写出程序，而是为了更清楚地描述模块级的逻辑。这样也可以避免各种图泛滥成灾。

第三节 面向对象的架构设计

在面向对象的设计中，关注点变成了消息和响应机制。

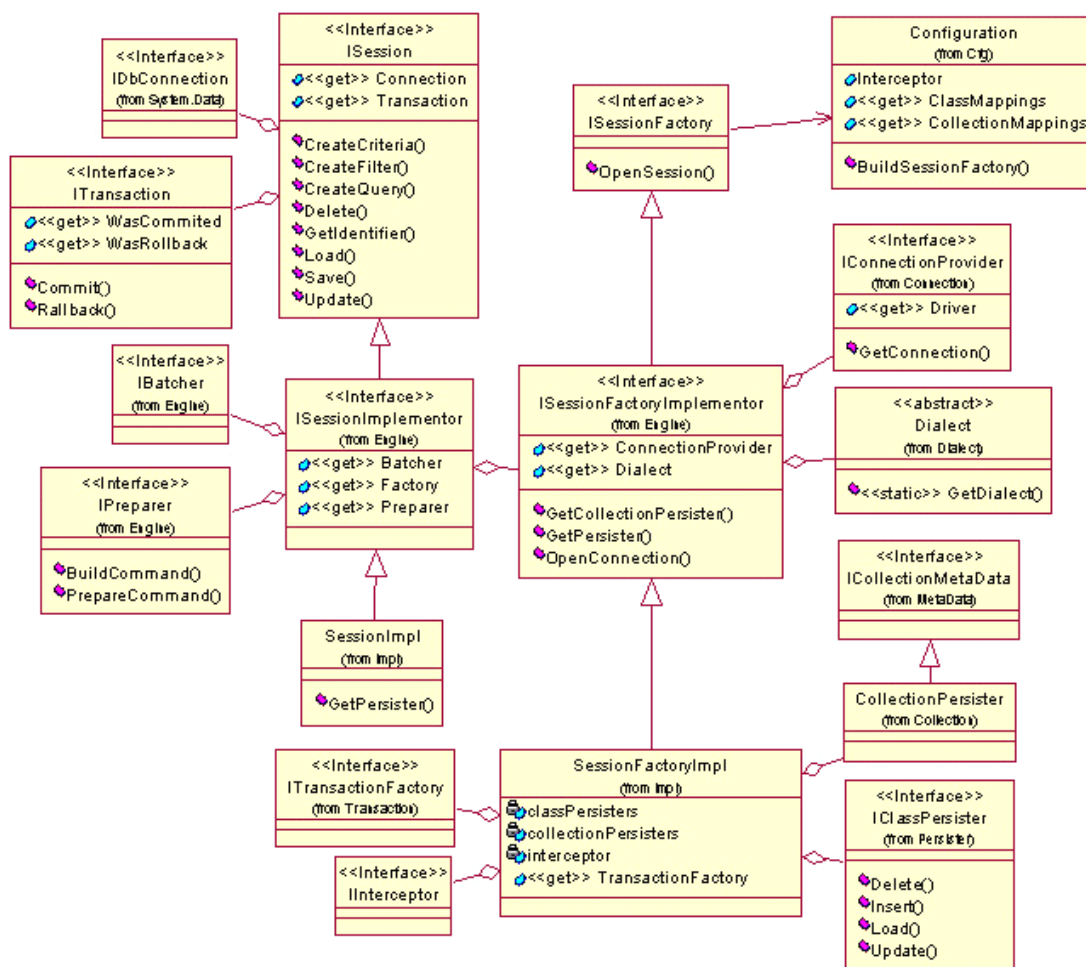
而我们由面向对象的分析转向面向对象的设计是一个自然的结果，在 OOA 中已经提供了足够的信息，

在高层设计阶段，我们可以用包图来建立体系架构。



在详细设计阶段，可以利用类图建立相应的体系结构。

下图是以类表达的典型的 Nhibernate 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。

下面我们来讨论有关问题。

第四节 高层设计中的架构分析

面向对象的设计并不是简单的把需求分析中的领域模型转换成设计模型就可以了，架构师必须在由需求分析获取架构因素，因此我们首先必须研究架构分析的问题。

另外，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，这是使用 OOD 必须关注的问题。面向对象的架构设计与结构化设计根本的不同，是非常注意实时信息，也就是消息和响应机制。另一方面，也非常注意代码重用，设计的目标往往是大型的、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。

架构分析的本质，是识别可能影响架构的因素，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，并掌握解决影响架构重要因素的众多方法。

架构分析是高优先级和大影响力的活动。

架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、架构分析

架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) 架构分析需要解决的问题

下面说明在架构级别上，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) 架构分析的一般步骤

架构分析有多种方法，大多数方法都是以下步骤的变体。

1. 辨识和分析影响架构的非功能性需求。
2. 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，这就是架构决策

二、识别和分析架构因素

1) 架构因素

任何需求对一个系统架构都有重要影响。

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，对架构最具影响的因素，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）决定了某个架构的独到之处，而不是功能性需求。

2) 质量场景

在架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”需要软件架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) 架构因素的描述

架构分析的一个重要目标，是了解架构因素的影响、优先级和可变性（灵活性以及未来演变的直接需要）。

因此，大多数架构方法，都提倡对以下信息建立一个架构因素表。

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，架构和其它因素	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。

在这里这么说明不等于它是唯一的或者最好的，但它对架构因素的分类很有效。

3) 架构因素和 UP 工件

在架构设计中，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。其隐含或者清晰的架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 <ul style="list-style-type: none"> ● 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. 商品的标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、架构因素的解析

架构设计的技巧就是根据权衡、相互依赖关系和优先级对架构因素的解决作出合适的选择。

但这还不全面，老练的架构师具有多种领域的知识（例如：架构样式和模式、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) 记录架构的可选方案、决定和动机

不管目前架构决策的原则有多少，事实上所有的架构方法都推荐记录：

可选的架构方案；决定；影响因素；显著问题；决定动机。

这些记录按不同的形式或者完善程度，被称之为：

技术备忘录；问题卡；架构途径文档。

技术备忘录的一个重要的方面就是动机或者原理，当开发者或者架构师以后需要修改系统的时候，架构师可能已经忘了他当初的设计依据（一个资深架构师同时带多个项目的情况非常常见），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，架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	
架构因素	<ul style="list-style-type: none"> ● 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

下面是指导做出架构决定目标：

1. 不可改变的约束，包括安全和法律方面的事务
2. 业务目标
3. 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，架构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

架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，是在架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，但这样的分离对于架构问题尤其突出。

因为系统的不同方面很广泛，而且架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：

1. 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。
2. 使用装饰器模式
3. 采用后编译器技术和面向方面的技术

有了架构分析的结果，我们就可以讨论高层架构设计本身的一系列原则了。

第五节 高层架构设计中的层模式

一、面向对象软件架构的优点

1) 面向对象软件架构的维和视图

一个系统的架构包括了多个维。

例如：

- 逻辑架构：描述系统的层、包、主要框架、类、接口和子系统的概念组织方式。
- 部署架构：描述系统的进程如何分配给处理单元和网络配置。

统一过程建议采用架构的六种视图（逻辑、部署等），我们后面会讨论这个问题。

2) 架构模式和模式的种类

架构模式是有关大尺度和粗粒度的设计，特别是应用在早期迭代过程（细化阶段）。也就是当主要的结构和连接建立起来的时候的一些原则。

二、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。

层模式与逻辑架构相关，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。

层为逻辑架构定义了一个 N 层模型，称之为分层架构 (Layers Architecture)，它作为模式得到了极为广泛的应用和引述。

框架 (framework):

一般来说框架具有如下的特征:

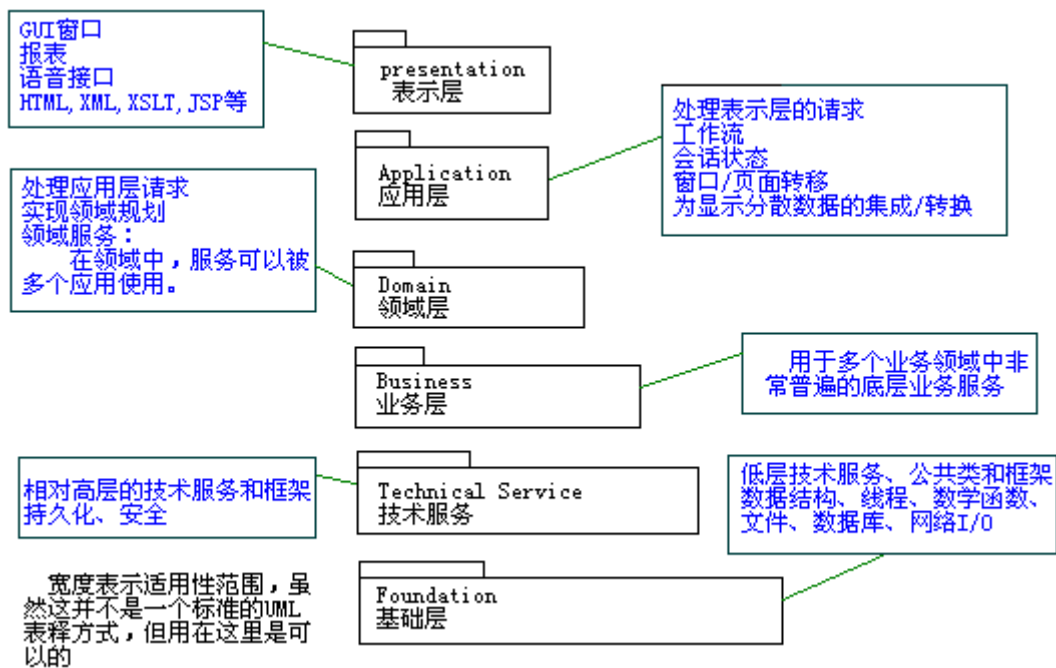
- 是内聚的类和接口的集合，它们协作提供了一个逻辑子系统的核心和不变部分的服务。
- 包含了某些特定的抽象类，它们定义了需要遵循的接口。
- 通常需要用户定义已经存在的框架类的子类，是客户得以扩展框架的服务。
- 所包含的抽象类可能同时包含抽象方法和实例方法。
- 遵循好莱坞原则: “别找我们，我们会找你。” 就是用户定义得类将从预定义的类中接受消息，这通常通过实现超类的抽象方法来实现。

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架构。



在具体架构设计的时候，可以建立比较详细的包图，但是，并不需要面面俱到。

架构视图的核心，是展示少数值得注意的元素。

统一过程的架构视图是这样告诫的：“选择一小组有意义的元素来传达主要的思想。”

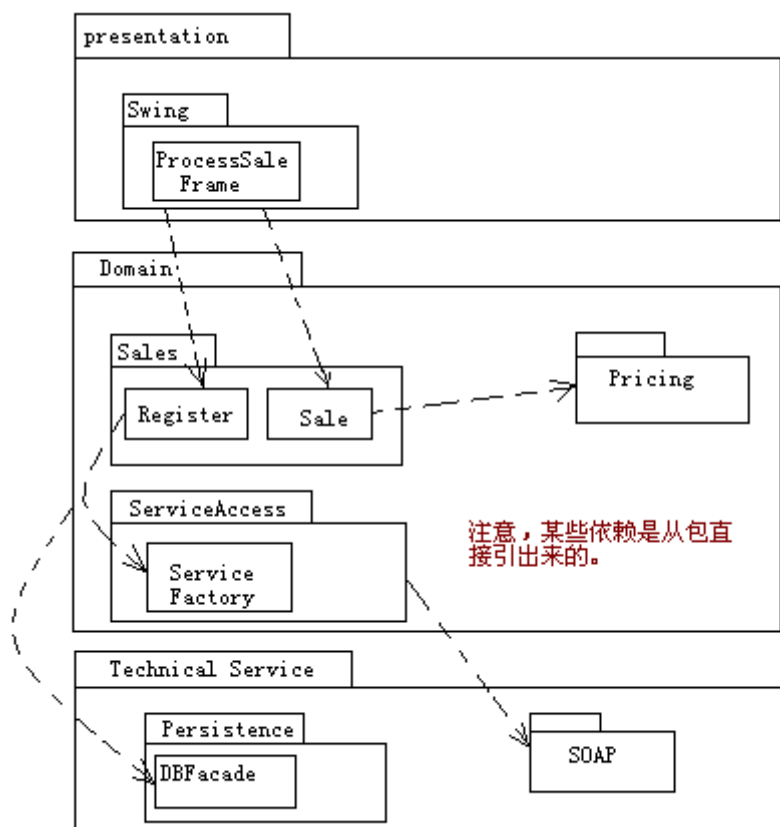
特别注意：

在面向对象的层模式中，底层的类不仅仅提供调用，更主要的是提供父类，很多情况下，需要使用配置文件来动态装配，这样才能适应不同的需求。

4) 层与层之间和包与包之间的耦合

逻辑架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：

在架构层面上，有两个设计上的决策：

1. 什么是系统的重要部分？
2. 它们是如何连接的？

在架构上，层模式对定义系统的重要部分给出了指导。

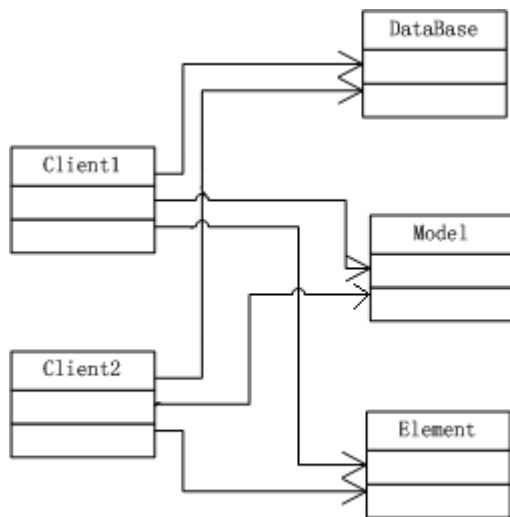
象外观、控制器、观察者这些模式，通常用于设计层与层、包与包之间的连接。

6) 外观模式

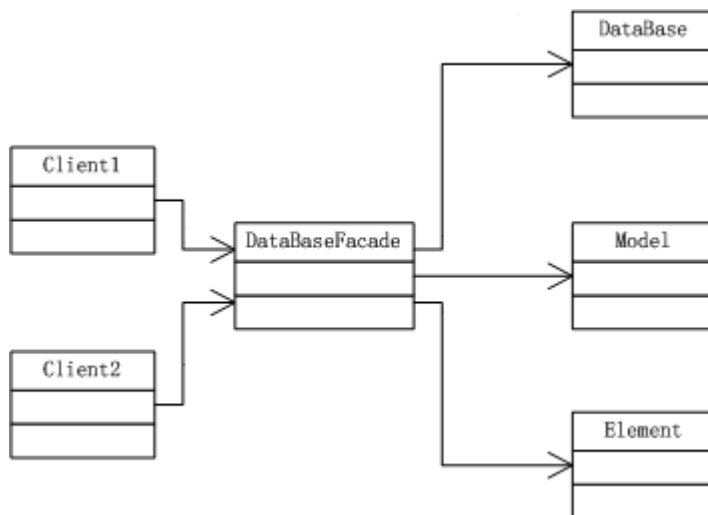
GoF 的外观模式，定义了一个公共的外观对象综合子系统的服务。

外观不应该表示大量子系统的操作，更确切的说，外观更适合表示少量的高层操作、粗粒度的服务。当外观呈现大量的底层操作的时候，会趋向于变得没有内聚力。

外观模式为了一组子系统提供一个一致的方式对外交互。这样就可以使客户和子系统绝缘，可以大大减少客户处理对象的数目，注意下图。



使用 Facade 之前



使用 Facade 之后

这样本来一个类的修改可能会影响一大片代码，而加了外观类以后只需要修改很少量的代码就可以了，使系统的高级维护成为可能。

7) 通过观察者实现向上协作

外观模式通常用于高层到底层的操作（底层提供外观，高层实现调用）。

当需要上层对底层的操作的时候，可以使用观察者模式。也就是上层响应底层的事件，但这个事件的执行代码由上层提供。

8) 层之间的松散耦合

大部分的多层架构不能像基于 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？

正如 GRASP 的受保护变化模式和低耦合模式的论述，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。

例如，建立一个 Java 应用程序，隐蔽对 Java 类库的访问有什么意义呢？

与类库的多个紧密耦合不太可能是个问题，因为它们是相对稳定而且无处不在的。

9) 应用层是可选的吗

如果存在应用层，那么它所包含的对象要负责了解客户端的会话状态，协调表示层和领域层，以及控制 workflow。

10) 在不同的层上模糊集合成员

一些元素必定只属于一个层，但有些元素却难以区分，特别是处在技术服务层和基础层之间，或者领域层和业务基础设施层之间的元素。其实这些层之间只存在模糊的差异，所以，“高层”、“低层”、“特殊”、“一般”这些术语是可以接受的。

开发组并不需要在限定的分类上做出决定，可以粗略的把一个元素归类到技术服务层或者基础层，也可以称之为“基础设施层”，注意，对于层并没有十分确定的命名习惯和约定，文献上各种命名上的矛盾是常见的，研究问题主要把握精髓，不要被这些表面的区别搞昏了头。

11) 使用反射动态装入对象

不论是 Java 还是 .NET，都很好的支持了反射，这样，建立一种通用对象容器成为可能，在详细设计的讨论中，我们会讨论一个这样的例子。

12) 利用工厂模式构建通用的创建者

为了保证层的通用型，在层中的必要部分，可以采用工厂模式创建对象，这个问题我们也会在详细设计的讨论中加以阐述。

13) 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。

三、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 Web 架构（比如 Struts），近来，这个术语（MVC）被分布式设计团体采纳，也应用在大规模的架构上，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。
- 允许新的视图方便地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。

第六节 框架设计的方法学问题

一、框架（Framework）的基本概念

事实已经表明，一个软件组织，合理的组织开发和使用框架，并且能跨越组织边界进行合作的能力越来越重要。软件架构使得您能够组合大量支撑产品和服务，一个共享的架构，可以使企业开发团队很方便的分解问题，从而确定：

哪些可以企业（或者开发组）内部解决；

哪些可以使用已有的服务。

当架构跨越组织的时候，你就可以调选公司内外组织的合作力量，获得共享架构带来的好处，在这样的情况下，架构设计组或者整个公司都需要掌握一些新的组织技能。

当公司内部存在产品线的时候，设计优秀的共享架构都可以带来实实在在的好处。

（1）框架

框架最简单的形式是指已开发过并已测试过的软件的程序块，这些程序块可以在多个软件开发工程中重用。框架提供了一个概括的体系结构模版，可以用这个模板来构建特定领域中的应用程序。

（2）为什么会出现应用框架

您只要细心地研究真实的应用程序，就会发现程序大致上由两类性质不同的组件组成，一类与程序要处理的具体事务密切相关，我们不妨把它们叫做业务组件；另一类是应用服务。人们自然会想要是把这些在不同应用程序中有共性的一些东西抽取出来，做成一个半成品程序，这样的半成品就是所谓的程序框架，再做一个新的东西时就不必白手起家，而是可以在这个基础上开始搭建。实际上，大型软件企业往往选择搭建自己的框架。

(3) 为什么要用框架？

因为软件系统发展到今天已经很复杂了，特别是服务器端软件，设计到的知识，内容，问题太多。在某些方面使用别人成熟的框架，就相当于让别人帮你完成一些基础工作，你只需要集中精力完成系统的业务逻辑设计。而且框架一般是成熟，稳健的，他可以处理系统很多细节问题，比如，事物处理，安全性，数据流控制等问题。还有框架一般都经过很多人使用，所以结构很好，所以扩展性也很好，而且它是不断升级的，你可以直接享受别人升级代码带来的好处。

目前一些常用的开源框架如下：



二、框架设计的核心原则

五个核心原则：

构想、预见、节奏、协作和简化。

1) 构想

构想原则说明了如何向架构的受益人描绘一幅一致的、有约束力的、以及灵活的未来图像。作为架构师，关键是要确保它提出来的架构设想与公司的业务目标相吻合，对于一个大型公司，做到这点事实上并不容易。

再次提醒，面向对象的架构，关键不是调用，而是继承和重用。

2) 节奏

节奏原则确保软件组织可以定期根据可预测的速度、内容和质量对工件进行取舍。在新颖的性能和规定的发布日期之间，有时候必须进行取舍，以确保发布日期，但这点可能和公司高层的设想不同，这如何解决呢？

3) 预见

首席架构师必须对未来发展走向有敏锐的洞察力，但这种预见往往和现行的标准有冲突，这就需要在两者间做出平衡，而这种平衡也是非常难处理的。

4) 协作

当首席架构师开始架构设计的时候，协作显得及其重要，我们一定要确保公司、周边合作者、领域架构师和开发组都能理解架构的关键思想，

5) 简化

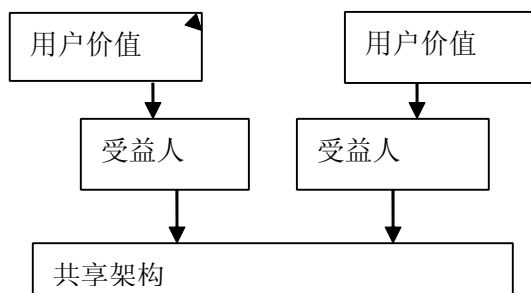
简化原则要求澄清并最小化架构与创建，当发现两个小组开发的构件有重叠的部分以后，应该可以考虑指定一个共享的构件，如何实现简化是构架师最值得关注的一个问题，非此架构设计的意义就显得不大。

这五个原则被称之为 **VRASP** 模型 (Vision, Rhythm, Anticipation, Partnering, Simplification)。这个模型重点在于软件架构的组织方面，事实上，软件架构师得以成功的最大障碍是组织问题而不是技术问题。

三、形成构想

1) 把价值映射为架构约束

这个问题的本质，是架构受益人如何把客户价值与架构约束捆绑。这里所谓受益人，实际上是指使用架构的开发者。



一致性和灵活性的考虑：

一致性的问题：是指受益人使用架构与期望值的符合程度。

灵活性的问题：是指受益人不破坏架构的情况下，利用共享框架的创建新的没有预见到的情况下的容易程度。

2) 产品线及其挑战

当产品线上有多个产品的时候，为了避免架构的设计被被动的拉向不同的方向，可以使用下面的三步方法：

- 1，清楚、简明的、阐述一条迫切的用户价值。
- 2，把用户价值映射为少数的能解决的问题。
- 3，把以上问题转译为的一组最小的约束条件。

遵循准则：

- 1，各方面的一致性
- 2，实施人员信任并使用架构
- 3，架构潜藏的知识对用户是可识别和可获得的

四、保证节奏

节奏能够克服复杂性，确保竞争优势。

节奏有三个元素：速度、内容和质量。

速度：一个团队和另一个团队之间（架构团队和开发工程师团队之间）同类型交接发生的频率，每次交接的时间越是可预测的，移交也就越容易管理。

内容：内容是指一个团体向另一个团体提供的价值。当软件架构师向开发团队提交的构架使开发团队获得了实实在在的好处，这个架构就被认为是有价值的。

质量：质量的含义是开发过程确保架构没有缺陷。

遵循准则：

- 1，经理们需要定期评估、同步和调整架构
- 2，架构用户需要对架构发布的内容和进度有高度的信心
- 3，通过节奏协调明确的活动

五、预测、验证和调整

架构师可以通过自己的经验和成功使用的架构，合理的猜测将来会发生什么。例如：

用户会有什么变化？

竞争形势会如何改变？

运行环境会如何改变？

架构的设计必须是能够适应这种可能的变化。

1) 验证：

在架构设计中，验证主要指的是对架构基础假定的测试，在架构成型以前，除非通过测试当初的基础假定是被确认的，否则就会发生代价高昂的错误。

2) 调整：

当预测结果发生变化的时候，你所面临的问题就是调整。

六、实现协作

在软件架构环境中，协作主要涉及对受益人的关系进行管理的过程。

合作并不能保证架构的受益人总是能和您保持一致，但是，当架构供应者发现他们有哪些功能他们没有一致的理解的时候，就必须用达成一致的方法来解决这个矛盾。

架构设计者应该和所有的架构受益人合作，使利益最大化，而不是仅仅靠合同和协议过日子。注意价值链的概念，每个行业都有自己的价值链，成功的构架设计者应该关注这些价值链。几个准则：

- 1，架构师需要了解谁是关键受益人，他们如何贡献价值，以及他们需要贡献什么。
- 2，和受益人之间达成明确和强制性的契约。
- 3，通过制度和非正式的规范强化合作。

七、简化

架构师和高级经理应该协力保持架构的平衡。

当某个新产品加入的时候，会大大增加架构的体积。架构师应该关注通用的服务，某个客户专用的能力不应该放入通用的架构里面。

架构师应该仔细考虑，极力找出隐蔽在多个不同需求中的公共元素。对于不能放弃的大型客户，需要仔细的谈判，提出多种解决方案供用户选择，而不是简单的行或者不行。

准则：

- 1，开发人员长期不断的使用架构的时候，减少了总成本和复杂性。
- 2，架构师明确理解关键最小需求，并构造多应用共享核心单元。
- 3，当不能被共享或者增加了不必要的复杂性的时候，应该把相关元素从核心移走。

在一个善于简化的组织中，开发人员会不断地清理核心，因为人人都知道复杂的核心所带来的麻烦。

第七节 面向服务架构（SOA）

面向服务的架构 (Service-Oriented Architecture SOA)是一种形式化的分离服务的架构风格。

面向服务的架构关注的是哪些是服务向用户提供的功能，哪些是需要这些功能的系统，这种分离，使用一种服务合约（Service Contract）的机制来完成的。

本质上来说，SOA 体现的是一种新的系统架构，SOA 的出现，将为整个企业级软件架构设计带来巨大的影响。

一、SOA 的优点

SOA 框架的特点是以服务为中心，它把应用程序划分成具有明确定义接口的模块，从而得到服务和应用程序之间相当松散的耦合。

在 SOA 中，服务供应商和消费者是两个独立的实体。

面向服务的架构的优点主要体现在以下几个方面：

- 降低应用开发费用。
- 降低维护费用。
- 增长的公司敏捷性。
- 生成对应用程序和设备的故障、中断更具免疫力的系统，提高整体的可靠性。
- 提供了一条应用系统的升级途径，对比使用单一的应用程序的时候，需要替换整个应用系统的标准升级方法，显然更为经济，更不容易失败。

二、SOA 的特性

SOA 有以下特性：

- 服务具有明确的接口（合约）与策略。
- 服务通常代表业务功能或者领域。
- 服务拥有模块化的设计。
- 服务被松散的耦合在一起。
- 服务是可以被发现的。
- 服务的位置对客户是透明的。
- 服务是独立于传输层的。
- 服务是独立于平台的。

SOA 可以通过很多方式来实现，但最常用的 SOA 是用 Web Service 来实现，这主要应为 Web Service 的独立于平台的特性和其它特性更符合 SOA 的规则。

1) 服务具有明确的接口与策略

明确定义服务具有的接口（合约）是 SOA 的核心定义。

合约应该包含两部分内容，一个是接口，另一个是业务策略。

普通对象概念的接口包括：

- 数据类型。
- 期望的输出。
- 必需的输入。
- 错误信息。

SOA 的合约扩大了接口的概念，包括：

- 所提供的功能。
- 需要的输入和期望的输出。

先决条件。

- 后置条件。
- 错误处理。
- 服务品质保证等。

关于业务策略，事实上服务的生产者和消费者都要定义策略，包括可靠性、可用性、安全性等等。

1. 所提供的功能

确切说明服务允许完成什么。

2. 期望的输入和输出

服务期待什么样的输入以及它能提供什么样的输出，对客户来说这是一个重要的信息。

3. 先决和后置条件

先决条件：

服务激活前存在的输入或者应用程序的状态，最常见的输入是安全口令。

后置条件：

请求被处理以后服务的状态。比如服务作为某个事物的一部分来调用的，这时候服务必须接到事务协调者的通知后才能完成这个事物的提交。

服务如何应对错误是绝对的后置条件，系统出错以后不同的错误系统应该是什么状态，这点

必须写清楚。

4, 错误处理

错误处理是另外一个需要在合约中说明的领域, 从 UML 的观点来看, 错误是一个通道, 所有错误都不返回参与者所期望的价值产品。

客户需要知道描述错误的数据结构或者其它信息。

5, 服务品质协议

服务品质 (QoS) 是可选的, 但确是合约的重要组成部分, 因为消费者很大程度上可以根据提供者提供的服务水准, 来选择它们的提供者。

服务品质包括诸如: 性能、多线程、容错之类问题。

6, 注册表

注册表 (Registry) 把所有的东西联系到了一起, 它是服务保存信息和登记信息的地方, 也是消费者找寻和履行合同的地方。

注册表这个术语有很多意思。一个注册表可以为消费者提供一个指定的查询标准, 来查找合约的机制。然后消费者将和服务联接在一起。

注册表可以由企业、独立来源、或者需要提供服务的其它业务组织来维护和提供, 所有的注册表都需要实现允许独立登记, 让消费者查找服务提供者并且和它们连接的应用程序编程接口 (API)。

注册表是把消费者和服务方分离开来的核心机制, 这种分离允许 SOA 增加需求能力, 并且提供连续可用的服务。

注册表并不一定需要包含合约, 注册表可以包括提供者所提供的服务以及合约地点的描述信息, 这样可以允许提供者在本地图维护自己的合约, 这样也可能更加方便。

2) 服务代表业务领域

服务可以用来建立各种各样的问题的领域, 即可以是企业领域, 也可以是技术领域。

SOA 真正的能力, 在与可以为企图领域建模, 因为业务服务通常比技术服务更加难以实现, 无论对内和对外都更加有价值, 所以 SOA 真正持久的价值在于建立一个重要业务过程的服务。

3) 服务拥有模块化设计

服务由模块组成, 模块花设计对 SOA 来说是很重要的, 模块可以被看作是一个执行具体、明确功能的软件和子系统。

模块应该表现为高的内聚性, 而且是完整的功能。

粗粒度做法是构造一个完整的转账模块, 这种模块提升了系统性能, 但减少了可重用性。但是, SOA 通过网络实现服务, 网络拥挤可能是主要矛盾, 因此, SOA 推荐的是粗粒度设计。这点非常重要。

4) 服务应该松散耦合

服务客户和服务提供者之间应该实现松散耦合, 也就是客户和提供者之间没有静态的、编译时刻的依赖关系。

服务把它履行职责的细节隐蔽起来。

这种隐蔽, 几乎大部分资料都是建议主要通过 GoF 的外观模式 (Facade) 实现。

5) 服务应该是可以被发现并且支持内省的

SOA 的灵活性和可复用性另外一个关键点, 就是动态发现和绑定的概念。

服务和客户之间没有任何静态连接，SOA 客户通过注册表来查找它们想要的功能，而不是使用编译的时候静态连接。因此，服务和客户都可以自由修改。

服务还可以在一个有限的时间内被提供，这就是说它们可以被租借，当客户超过有效时间以后，将会被迫转回到注册表，重新绑定合约或者选择另外的合约。

6) 服务是独立于传输机制的

客户使用网络来访问和使用服务，SOA 应该独立于访问服务的网络种类，服务独立于用来访问他的传输机制，意味着需要建立一个适配器来支持访问它的各种传输机制。通常情况下，适配器需要根据情况来构造（HTTP 或者 RMI），同一个适配器，也可以被多个服务所使用。

7) 服务的位置对客户是透明的

服务的位置对客户透明，实施上表达了客户调用服务的时候，并不需要关心服务具体的位置。这就使 SOA 在实现过程具有巨大的灵活性。服务可以被放到最方便的地方去，必要的时候（比如企业整顿），服务业可以放到第三方提供者那里。或者服务中断的时候，可以把服务请求转发到完全不同的另一个地点。

8) 服务应该是独立于平台的

服务应该独立于平台和操作系统。

对于 Web 服务来说，虽然在理论上，Java 和 .NET 使用着相同的协议和标准，因此，进行互操作是没有问题的，但实际上，由于 SOAP、协议中有很多模糊和未定义部分，所以，这之间的互操作还是存在不少问题，需要我们认真加以研究和试验。

三、构建 SOA 架构时应该注意的问题

当架构师基于 SOA 来构建一个企业级的系统架构的时候，一定要注意对原有系统架构中的集成需求进行细致的分析和整理。基于 SOA 的企业系统架构通常都是在现有系统架构投资的基础上发展起来的，我们并不需要彻底重新开发全部的子系统。

SOA 可以通过利用当前系统已有的资源(开发人员、软件语言、硬件平台、数据库和应用程序)来重复利用系统中现有的系统和资源。SOA 是一种可适应的、灵活的体系结构类型，基于 SOA 构建的系统架构可以在系统的开发和维护中缩短产品上市时间，因而可以降低企业系统开发的成本和风险。

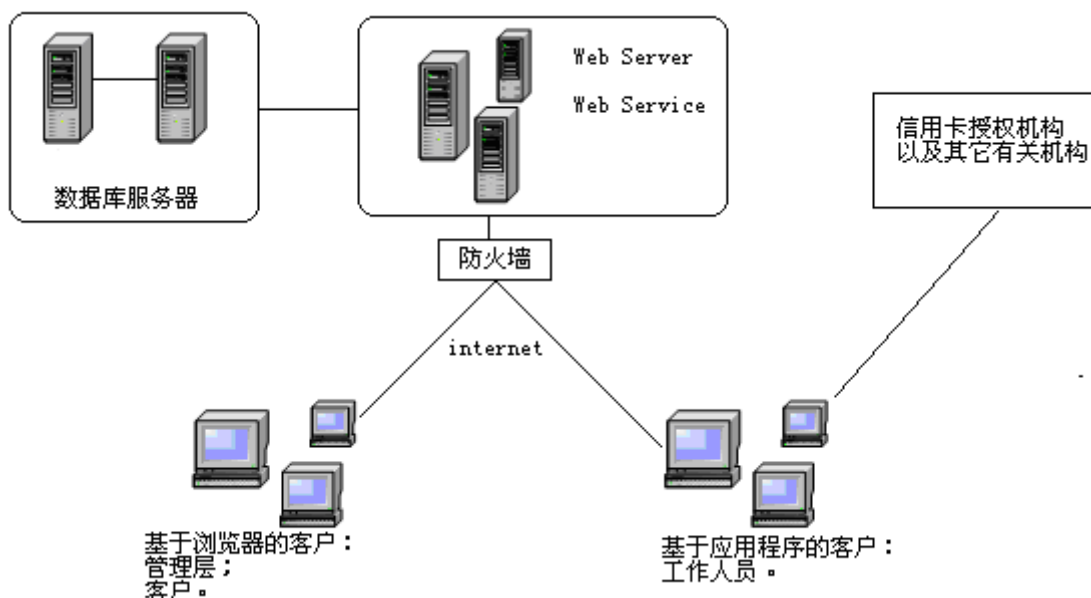
四、服务粒度的控制

当 SOA 架构师构建一个企业级的 SOA 系统架构的时候，关于系统中最重要元素，也就是 SOA 系统中的服务的构建有一点需要特别注意的地方，就是对于服务粒度的控制。

服务粒度的控制 SOA 系统中的服务粒度的控制是一项十分重要的设计任务。通常来说，对于将暴露在整个系统外部的服务推荐使用粗粒度的接口，而相对较细粒度的服务接口通常用于企业系统架构的内部。

五、案例：电源销售服务系统高层架构

这里只列出了初步的顶层架构。



设计中注意了几个问题：

- 1，对于管理层和客户，主要从使用方便性考虑，采用浏览器。
- 2，对于工作人员，因为要处理的内容比较复杂，采用应用程序，但是一个免维护的瘦客户端。
- 3，瘦客户端并不是指代码越少越好或者功能越少越好，而是把易变的、需要配置的、需要集中处理的内容转向应用程序服务器。事实上，客户端的功能越强，越能缓解服务器的压力。
- 4，某些特殊的专业通讯联系（比如信用卡授权机构），可以由客户端直接完成，并不一定一切都通过服务器，但需要向服务器提交必要的信息。
- 5，对于集中处理的部分，采用大粒度设计，以缓解网络压力。
- 6，由于服务方采用无状态模式，所以要严格控制客户调用信息的时间，对于需要长时间传输的信息，可以采用其它通道完成。
- 7，对于客户应用程序，某些不是十分大的，变化频度不是十分高的，调用频度比较高的数据，可以在客户端建立缓存，并且可以建立关联的映像表，这样就可以避免对最主要的数据处理的挤压，提高数据库的应用效率，但要考虑修改数据时候的并发策略。

第八节 软件架构的质量描述和评估

一、软件架构的创建原则

每个项目最开始的时候，是构架师最重要的时刻，在创建架构的过程中，可以依照下面的原则：

- 架构应该是精炼的；
- 架构应该是平易近人的；
- 架构应该是易读的；
- 架构应该是容易理解的；

架构应该是可信的；
架构不一定要完美无缺的；
不一定要一开始就做大的设计，如果在模型完善和实现之间作个选择的话，选择实现它；
做最简单和可行的事情，不要排除未来的需求；
架构是共享的财产；
让所有的涉众都参加进来，但还要能控制局面；
架构组的规模应该小；

软件架构需要做的事情是：

理解需求：

特别是非功能性需求，或者叫品质需求。

创建或者选择架构：

尽可能使用架构师熟悉的方法、技术和实践来满足，使用不为人所知的方法来创建架构是要冒一定风险的。

为了识别好的架构方案，可以采用下面的检查列表，作为可能误入歧途的早期预警征兆：

1) 架构被迫要求满足当前组织习惯。

当软件组织的习惯与需求不一致的时候，被迫满足组织的习惯，很容易误入歧途，很多情况下，适当的妥协是可以的，但架构的完整性应该尽可能保持。

2) 有太多的最高层架构组件（复杂性太高）。

如果这些组件的数目达到一定的级别（一般为 25 个），架构就已经太复杂了，以至于无法保证项目本身概念的完整性。

3) 某个特定需求超过了设计中的其它需求。

当某个目标高于其它一切需求的时候，那就要考虑这个目标可能只是投资商、项目经理、甚至是构架师的偏好，某个需求高于一切的时候，往往不能兼顾其它的需求。

4) 架构依赖于平台提供的选择。

实际上一个高度易用性的项目是不能受平台制约的。

5) 使用某些专用的组件，而不使用同样好的标准组件。

不要被花哨的设计和有趣的设计所迷惑，有很多能力可以使用标准组件来完成的。

6) 组件的定义划分来自于硬件的划分。

组件的设计过程是不考虑硬件结构的，否则软件就缺乏拓展性。

二、品质属性

软件架构的品质属性，可以通过下面几个方面来描述，请试试能不能定量的描述。

1) 性能

每个用例的预期响应时间是多少。

平均/最慢/最快的预期响应时间是多少。

需要使用哪些资源（CPU,局域网等）。
需要消耗多少资源。
使用什么样的资源分配策略。
预期的并行进程有多少个。
有没有特别耗时的计算过程。
服务器是单线程还是多线程。
有没有多个线程同时访问共享资源的问题，如果有，如何来管理。
不好的性能会在多大程度上影响易用性。
响应时间是同步的还是异步的。
系统在一天、一周或者一个月，系统性能变化是怎样的。
预期的系统负载增长是怎样的。

2) 可用性

系统故障有多大的影响。
如何识别是硬件故障还是软件故障。
系统发生故障后，能多快恢复。
在故障情况下，有没有备用系统可以接管。
如何才能知道，所有的关键功能已经被复制了呢。
如何进行备份，备份和恢复系统需要多长时间。
预期的正常工作时间是多少小时。
每个月预期的正常工作时间是多少。

3) 可靠性

软件或者硬件故障的影响是什么。
软件性能不好会影响可靠性吗。
不可靠的性能对业务有多大影响。
数据完整性会受到影响吗。

4) 功能

系统满足用户提出的所有功能需求了吗。
系统如何应付和适应非预期的需求。

5) 易用性

用户界面容易理解吗。
界面需要满足残疾人的需求吗。
开发人员觉得用来开发的工具是易用的和易理解的吗。

6) 可移植性

如果使用专用开发平台的话，用它的优点真的比缺点多吗。
建立一个独立层次的开销值得吗。
系统的可移植性应该在哪一级别来提供呢(应用程序、应用服务器、操作系统还是硬件级别)。

7) 可重用性

该系统是一系列的产品线的开始吗。

其它建造的系统有多少和现有系统有关呢？如果有，其他系统能重用吗。

哪些现有组件是可以重用的。

现有的框架和其它代码能够被重用吗。

其它应用程序可以使用这个系统的基础设施吗。

建立可重用的组件，代价、风险、好处是什么。

8) 集成性

于其它系统进行通信的技术是基于现行的标准吗。

组件的接口是一致的和容易理解的吗。

有解释组件接口的过程吗。

9) 可测试性

有可以测试语言类、组件和服务的工具、过程和技术吗。

框架中有可以进行单元测试的接口吗。

有自动测试工具可以用吗。

系统可以在测试器中运行吗。

10) 可分解性

系统是模块化的吗。

系统之间有序多依赖关系吗。

对一个模块的修改会影响其它模块吗。

11) 概念完整性

人们理解这个架构吗。是不是有人问很多很基本的问题呢。

架构中有没有自相矛盾的决策。

新的需求很容易加到架构中来吗。

12) 可完成性

有足够的时间、金钱和资源来建立架构基准和整个项目吗。

架构是不是太复杂。

架构足够的模块化来支持并行开发吗。

是不是有太多的技术风险呢。

上述问题，给我们提供了一个线索，在回答这些问题的时候，我们对初始设计的改进方向，就已经一目了然了。

第五章 详细设计阶段的数据库结构设计

详细设计阶段包括：数据库设计、模块设计和界面设计。

下面我们探讨一下数据库设计的有关问题。

第一节 关系数据库的结构设计

一、面向过程的设计与实体关系图

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


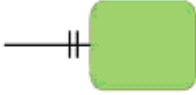
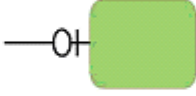



实体的描述特征称为**属性**，某些属性可以逻辑上被组合，称为**组合属性**，它在不同的数据建模语言中也被称作串联属性、合成属性或者数据结构。

3) 域

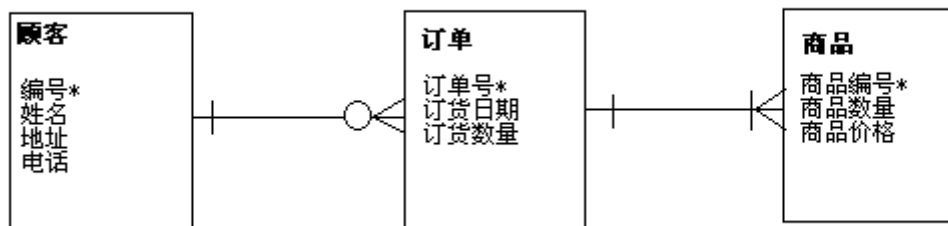
于是属性的一个参数，定义了这个属性所能定义的合法值。事实上这个值和使用的数据库特点有关。

4) 标识符

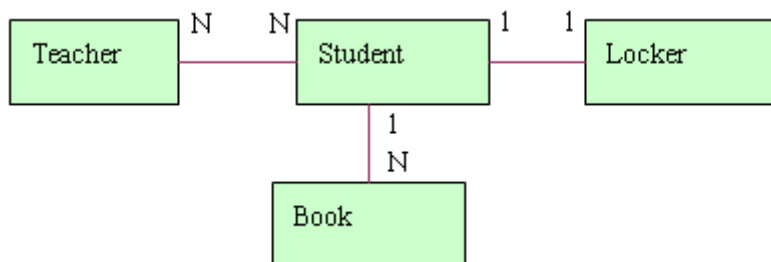
然后考虑实体之间的关系，关系基数符号如下：

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	 — or — 
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。



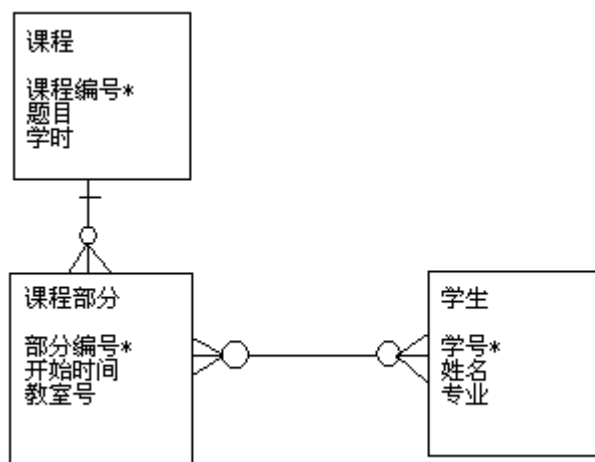
也就是一对一（1：1）或一对多（1：N）或多对多（N：N）关系。

比如：

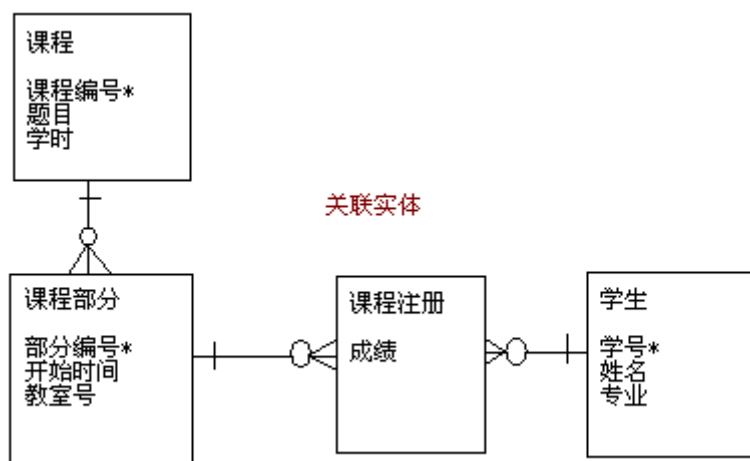
Student 和 Locker 之间可以是一个一对一关系，因为每个学生都有一个相应的锁柜，而每个锁柜只给一个学生使用。Student 和 Book 之间可以有一个一对多关系，因为每个学生可以有 multiple 本书，但每本书只归一个学生所有。Student 和 Teacher 之间有一个多对多关系，每个学生可以有多个教师授课，而每个教师又可以为多个学生讲课。

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、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用数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。

但我们应该如何选择主键呢？

智能键、常规键和代理键

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，逐渐重复也不是没可能，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)：没有重复子段和字段组的数据表结构。

函数相关：两个字段值之间一一对应。如果对于任意子段 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。

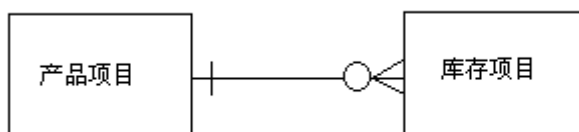
这里主要研究其它字段与关键字段的关系

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与“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与“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。

	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。

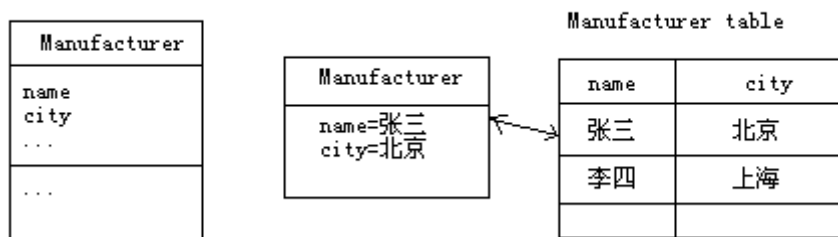
第二节 面向对象数据库设计

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

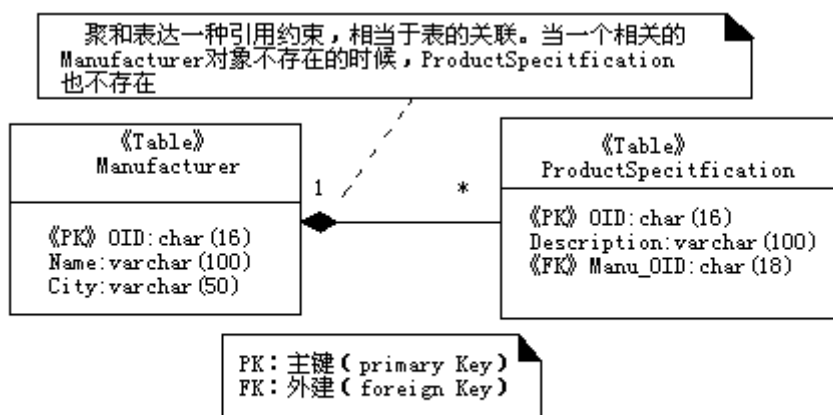


在 UML 中，虽然可以很好的表达类，但是，为了确切的表达数据，还需要有一些扩展，这个关于数据建模的扩展标准，已经提交给了 OMG 组织。

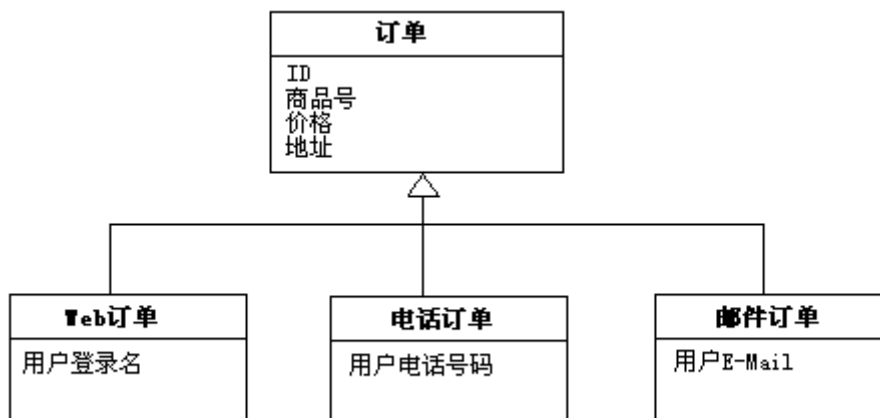
比如：PK：主键（primary Key）；

FK：外键（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。

二、持久化对象

目前常用的存储机制主要由两种。

对象数据库：

如果用对象数据库来存储和检索对象，就不需要第三方持久化服务，这是使用对象数据库吸引人的地方之一。

关系数据库：

由于RDB(关系型数据库)的流行，通常我们遇到的数据库都是RDB而不是更方便的ODB，这样一来，面向记录和面向对象的数据表示之间会有一系列的问题。往往我们会需要一个O-R映射服务。

其它：

出RDB以外，有时候我们还会使用其它存储机制(XML结构、层次数据库等)来存储数据，同样也需要有某种服务，来使这些机制和对象一起协调工作。

绝大多数应用，都需要从一个持久化存储机制(例如关系型数据库)存储和检索信息。

通常更好的办法，是购买或者获得工业的持久化框架，而不是自己开发。

开发工业级的数据库持久化 O-R（对象关系映射）服务需要数人年的时间，其中许多细节问题需要专门的专家。

三、解决方案：来自持久化框架的持久化服务

持久化框架（persistence framework）是多用途的、可重用的和可扩展的一组类型，它提供了支持持久化对象的功能。

持久化服务（persistence service）（或子系统）实际上是提供了这种服务，而且使用持久化框架来提供这种服务。

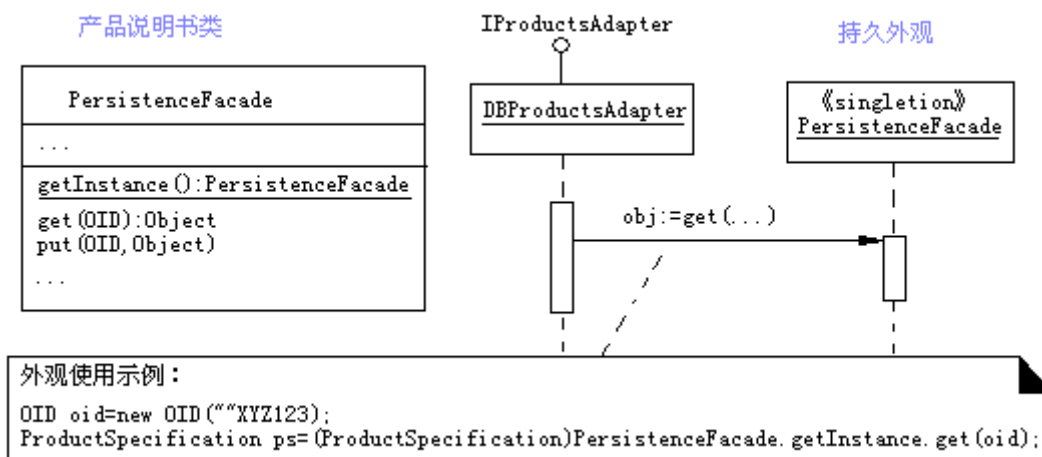
一般来说，持久化服务是属于技术服务层的子系统。

典型的比如：Hibernate。

这种持久化服务需要关注下面两个问题。

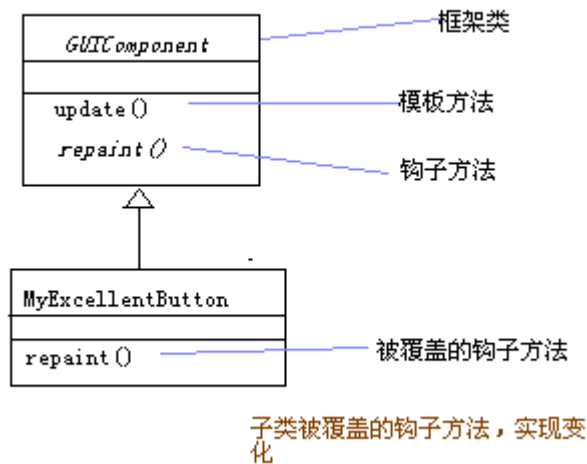
1、使用外观模式访问持久化服务

我们已经讨论过，外观模式是为子系统提供一个统一的接口，事实上，通过给定 ID，就可以返回一个对应的（代表一行）的对象。这里的外观可以是一个单件模式。



2、使用模板方法设计框架

是用模板方法，也可以实现持久类根据数据变化。



至少在目前关系型数据库还是非常流行的情况下，O-Rmapping 还是非常有意义的，因为这样一来，设计的时候就可以专注于对象的特点，可以使用抽象和泛化的方法来处理问题。

六、数据库的关联设计

在设计数据库系统的时候，必须作出如下重要决策：

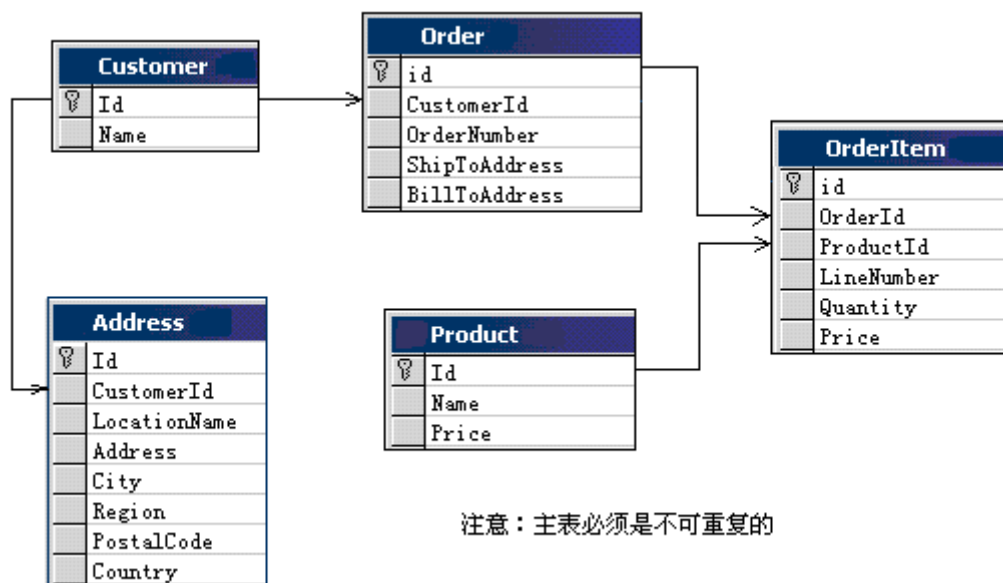
应该加载多少数据？

在涉及多个关联表的时候，如何更新数据？

实现何种类型主键（最重要的决策问题）？

我们下面将逐一进行讨论。

设想我们有一个简化的订单数据库，该数据库包含 5 个关联表。



1) 应加载什么数据

数据选择:

应该只加载用户需要处理的非连接数据,几乎所有的情况下只需要获取数据库的一个数据子集。

数据量:

数据量的选择会影响加载时间、更新时间、以及内存需求量。记住,非连接对象是一个基于内存的对象。因此要注意所获取的数据量大小,如果不能确信获取这些数据是必要的,就不要获取它。

分割数据:

根据数据对象的使用目的,最好把数据分割成多个部分,并分别存入相应的本地数据集对象。

例如,当用一个非连接对象保留数据的时候,这个对象会包括所有 5 个表。

再仔细分析一下,首先假定主体是销售部门,销售部门的主要关心点是客户:

针对特定客户,一个客户在 **Customer** (消费者) 和 **Address** (地址) 表只有一行信息,但是 **Order** (订单) 表和 **OrderItem** (订单项) 却可以包含多行与该客户相关的信息(一个客户可以有多个订单)。这样,我们可以把订单和客户信息放在一个 **DataSet** 里面 (**Customer DataSet**)。

如何处理 **Product** (产品) 表呢? 不同的情况可能对 **Product** 表的要求是不一样的。

如果假设本地数据集对象包含该客户在某个时刻的全部数据,则 **Product** 表可能只包含与 **OrderItem** 表相关的行。

同样,如果希望能够为不同的产品增加更多的订单,又可能必须保证 **Product** 表是完整的,因此,可能需要再次将该表存入原先其所属的本地数据集对象中。这样就能独立的在客户之间传递产品列表。

我们希望能够删除 **Product** 表中不再使用的产品,但不强制删除它们在 **OrderItem** 表中的引用。

由于情况比较复杂,我们可以专门设置一个本地数据集 (**Product DataSet**) 来装载 **Product** 表。也就是使用一个本地数据集保留客户数据,另一个本地数据集保留产品数据。

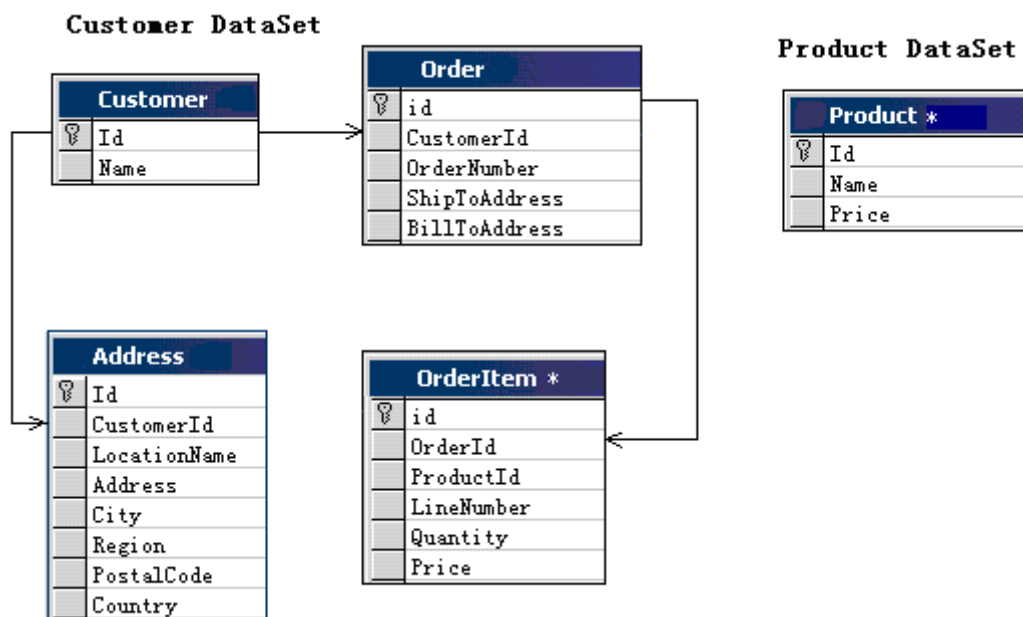
记住,不能在不同的本地数据集对象的数据表之间建立外键约束。但可以很容易的通过 **OrderItem** 表的 **ProductId** 定位 **Product** 数据,如下图所示。

在下载数据的顺序上,首先决定 **Order** 的要求,再由它的 **Id** 决定下载哪些 **OrderItem** 内容,以及由 **CustomerId** 决定下载哪些 **Customer** 和 **Address**。

由 **OrderItem** 的 **ProductId** 决定下载哪些 **Product**。

这样就可以免掉许多无效数据的下载。

当然,上面这些讨论并不是规则,需要具体情况具体分析。



第三节 并发问题及其应对

数据访问的一个挑战性问题，就是多个用户可能同时访问数据库同一个数据。比如一个用户正在编辑数据，另一个用户正在利用这个数据形成报表。

这就要注意两个问题：

- 1) 防止两个用户同时编辑数据；
- 2) 防止报表显示不完整的数据。

并发是使多个用户访问数据库，并得到一个相互一致的数据视图的能力。通过锁定必须的行、表或数据库，防止用户访问可能不一致的数据。

例如：

假定小张正在贷方帐户 A 和借方帐户 B 之间进行一项转账业务。

而此时小李正在帐户 A 中提取资金，则小张看到的帐户 A 的余额应该是多少呢？如果小张的事务还没有完成，如何处理小李的事务呢？

解决方案：

在一个事务还没有完成的时候，数据库服务器通过锁定数据库来接决这样的问题，在小张的事务还没有完成的时候，小李必须等待小张的事务完成以后才能进行操作，我们的目标是尽可能快地处理事务，并使锁定等待的时间尽可能短。

但是，我们如果使用非连接对象，把一部分数据库数据复制到客户段，等修改后再发回数据库，而把这个期间作为一个完整的事务，就会产生严重的锁定问题。

正是这种情况，迫使非连接对象获得数据的时候不启动一项事务（因而不会锁定），但是在提交数据的时候，检查和处理多个更新操作引起的冲突。

设想有一个表“账务”，包含字段为编号、姓名、金额。

把数据读入 DataTable 表以后，DataRow 对象将包括 Current 版本和 Original 版本数据。

更新命令如下：

```
Sql="UPDATE 账务 SET 姓名 = @姓名, 金额 = @金额
WHERE (编号 = @Original_编号) AND (姓名 = @Original_姓名 OR @Original_姓名
IS NULL AND 姓名 IS NULL)
AND (金额 = @Original_金额 OR @Original_金额 IS NULL AND 金额 IS NULL);
SELECT 编号, 姓名, 金额 FROM 账务 WHERE (编号 = @编号)";
```

这条命令的 **Where** 子句指出，只有在数据库当前列值与原来的列值都相同的时候，才会执行这条命令，这可能是解决并发冲突最安全、最容易的办法。此外还要注意，在提交一行数据的时候，所有的列值都将改变。

这时，小张修改金额，提交应该没有问题

但小李修改姓名，提交发生并发错误

但这样的策略是不是正确呢？毕竟两个人改的不是同一列数据，这样处理起来比较简单（一行数据全部改变），如果希望修改这个行为，可以改变 **Sql** 语句。

解决并发问题的策略

如何解决并发冲突是一个商业决策，下面列出了一些主要决策原则：

- **时间优先**：第一次更新优先，也称为“时间顺序优先”，只保留第一次更新的结果，上面的例子实现了这个策略。
- **时间优先**：最后一次更新优先，也称为“逆时间顺序优先”，只保留最后一次更新结果，这个方法最简单，因为可以把 **Where** 子句的内容全部去掉。
- **角色优先**：卖方人员优于买方人员，因为卖方人员更了解产品。这个方式实现起来难度较大，因为必须知道每个用户的角色，而且如果角色相同，还是应该保留实现顺序优先作为备用机制。
- **位置优先**：位置优先，总店优于分店，这个方式必须知道角色位置，事实上权限往往决定了位置，所以可以用权限优先来代替这个策略，同样也需要用顺序优先作为备用机制。
- **用户解决冲突**：当发生冲突的时候，弹出一个冲突解决界面，由用户决定下一步处理方式，这种方式虽然直观，事实上并不常用，而且出现问题的机率也比较大。

一般来说，对于顾客数据可采用角色优先策略，对于账目数据可采用位置优先策略。

第四节 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，但事务将控制和维持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？

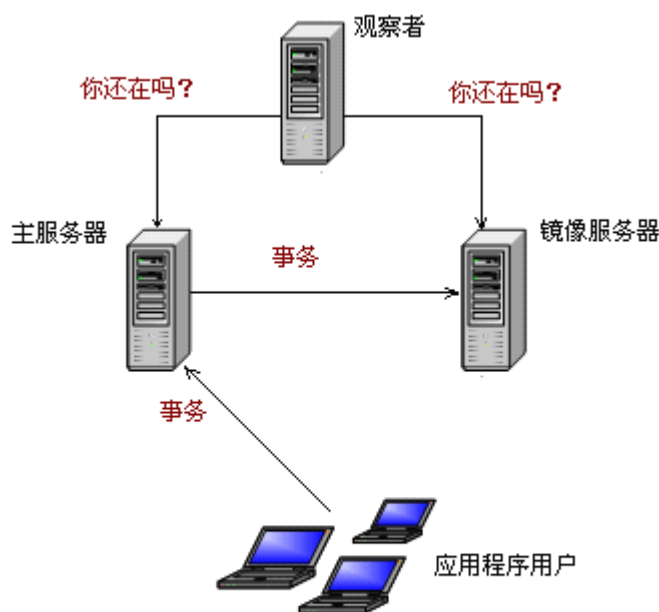
这就要用到容错恢复技术了。

请看下面的场景：

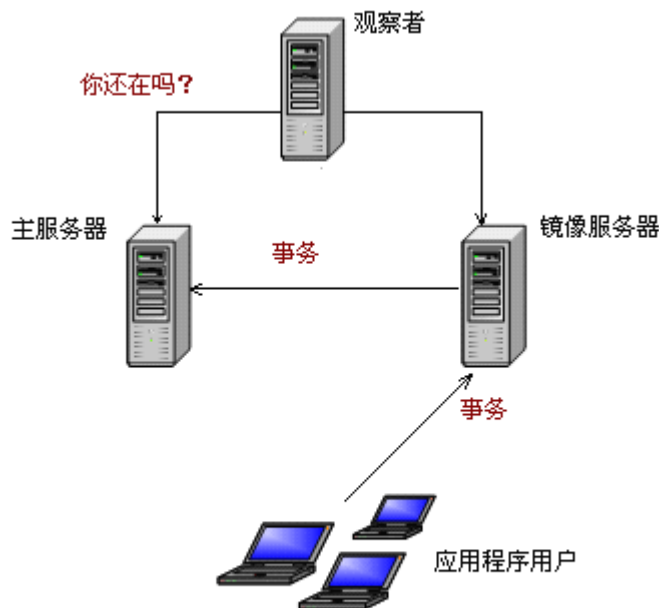
使用三台数据库服务器，“主服务器”、“镜像服务器”、“观察者服务器”，使用数据库镜像的时候，客户只和主服务器联系，镜像服务器处于数据恢复状态（不能进行任何访问），当向主服务器提交一个事务的时候，也会把这个事物发给镜像服务器。

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。

第五节 数据库结构设计案例

我们用前面讨论的“电源设备销售客户服务子系统”作为例子，来具体分析一下数据库设计的有关问题，这个案例的说明见于第三章第八节综合案例的研究，表达了TB公司电源设备销售部为了实现信息技术战略规划（ITSP）构思的销售服务系统，我们用这个案例来具体说明一下数据库的设计方法。

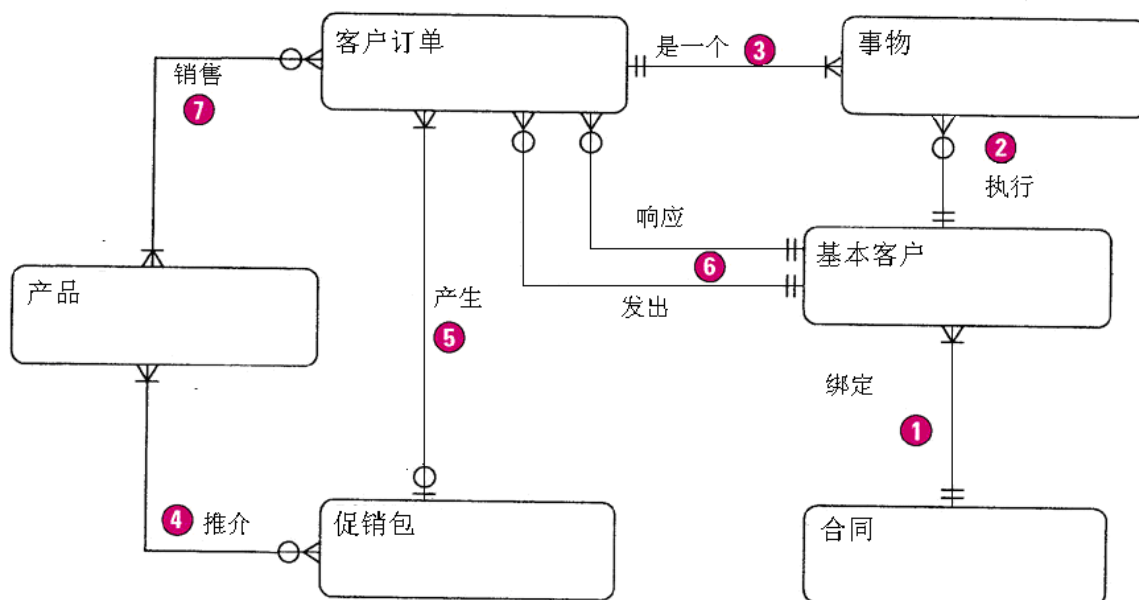
一、获取实体

仔细研究需求分析文档，从而可以获取实体，注意实体要按照业务词汇来定义，而不要使用技术词汇来定义，例如，为了简化问题，这里分析上面项目的订单和促销活动部分，在这一部分里，订单和促销活动联系在一起，而其它的购买方式暂时不予考虑。在这样的情况下，本项目定义的实体如下。

实体名称	业务定义
合同	合同是记录客户购买产品的状况，当购买产品达到一定数额的时候，可以得到规定比例的返点。
基本客户	基本客户是目前比较活跃的客户，一是公司必须关注的客户群体，促销活动主要针对的是这样的客户群。
客户订单	基本客户发出的订单记录，
事务	客户服务系统必须响应的一个业务事件。
产品	可以用于销售的电源产品。
促销	由市场部组织的促销活动，在促销期间，对于不同类型的客户可以提供不同的产品价格。

二、领域数据模型

利用我们已经讨论过的领域数据模型（早期称作上下文数据模型），我们可以根据对业务的理解，建立包括业务实体类和它们之间的自然关系，也就是构造了如下关系。



整个过程关键是对业务的理解，没有对业务的理解系统的分析和设计简直是不可能的，下面我们做一些说明：

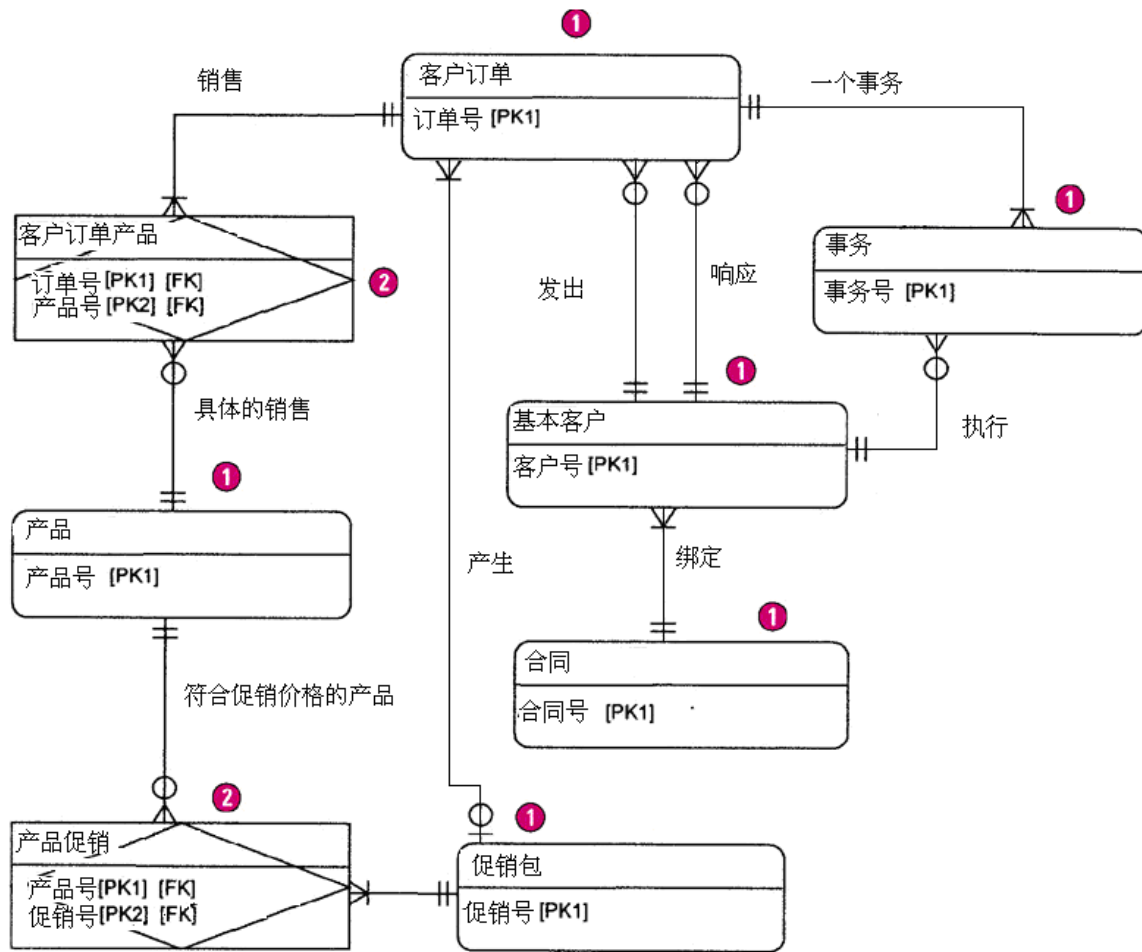
- 1，合同绑定一个或多个基本客户，反之一一个基本客户只绑定一个合同。
- 2，一个客户执行零个、一个或者多个事务，反之一一个给定事务只能一个客户执行。
- 3，一个客户订单是一个事务，事实上一个客户订单也可能是多个事务（新增订单、删除订单、修改订单等），反之一一个事务只能对于对应一个订单。
- 4，一个促销包用于打包推广一个或者多个产品，反之一一个产品可能在一个或者多个促销包中存在，注意这是个多对多关系。
- 5，一个促销包会产生多个客户订单，反之一一个订单为零个或者一个促销包。零个的原因在于，有的客户不符合促销打包的条件，将在另外的系统中解决。
- 6，如果不同的关系沟通不同的业务事件和关联，那么两个实体间允许存在多个关系。比如对于促销包所产生的订单，一个客户可以响应零个、一个或多个订单。另外，客户也可以主动发出零个、一个或者多个订单。当然这样的关系也可以直接写出“发出 or 响应”来表达。
- 7，一个订单销售一个或多个产品，反之一一个产品可以有零个、一个或者多个客户，注意这是个多对多关系。

这样一个数据模型构建过程，事实上也加深了分析人员对于业务模型的理解，这是非常有意义的。

三、基于键的数据模型

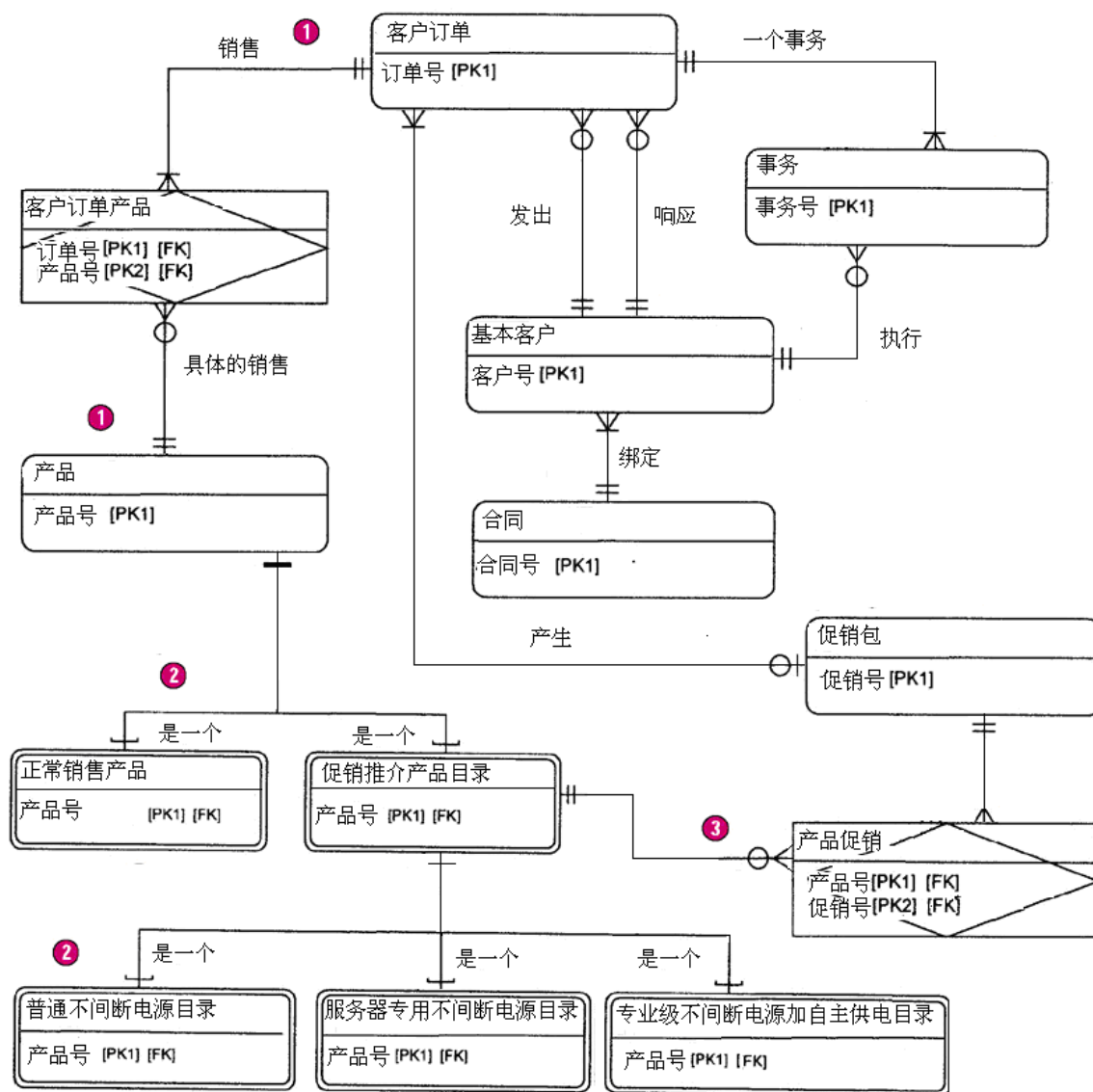
一旦领域模型被构造起来，我们就可以考虑给模型建立主键，我们已经讨论了关于主键的有关问题，这里需要说明的，一般来说尽可能用简单的单一属性作为主键。

对于多对多关系，需要增加一个关联表。



四、概化层次体系的数据模型

如果促销政策比较复杂，可以考虑把促销商品通过泛化实现层次体系。



五、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。

第六章 详细设计阶段的类结构设计

详细设计阶段的一个十分重要的问题，就是进行类设计。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。应用在实现架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架构模式。

第一节 类结构设计中的通用职责分配软件模式

GRASP 模式（General Responsibility Assignment Software Patterns 通用职责分配软件模式）能够帮助我们理解基本的对象设计技术，并且用一种系统的、可推理的、可说明的方式来应用设计理论。

一、根据职责设计对象

职责：职责与一个对象的义务相关联，职责主要分为两种类型：

1) 了解型 (knowing)

- 了解私有的封装数据；
- 了解相关联的相关对象；
- 了解能够派生或者计算的事物。

2) 行为型 (doing)

- 自身执行一些行为，如建造一个对象或者进行一个计算；
- 在其它对象中进行初始化操作；
- 在其它对象中控制或者协调各项活动。

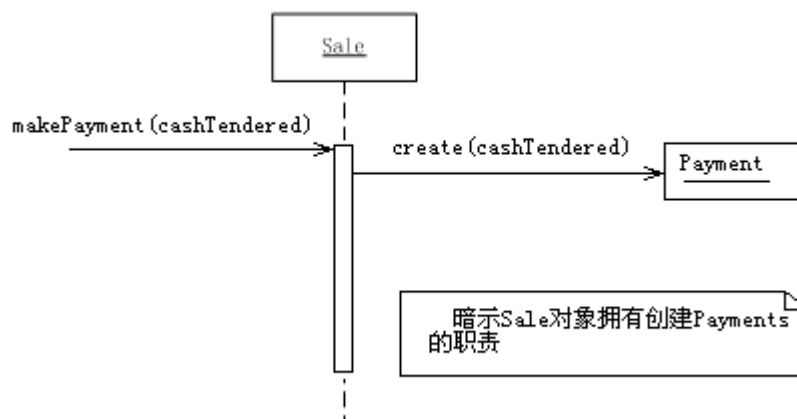
职责是对象设计过程中，被分配给对象的类的。

我们常常能从领域模型推理出了解型相关的职责，这是因为领域模型实际上展示了对象的属性和相互关联。

二、职责和交互图

在 UML 中，职责分配到何处（通过操作来实现）这样的问题，贯穿了交互图生成的整个过程。

例如：



所以，当交互图创建的时候，实际上已经为对象分配了职责，这体现到交互图就是发送消息到不同的对象。

三、在职责分配中的通用原则

总结一下：

- 巧妙的职责分配在对象设计中非常重要。
- 决定职责如何分配的行为常常在创建交互图之后发生，当然也会贯穿于编程过程。
- 模式是已经命名的问题/解决方案组合，它把与职责分配有关的好的建议和原则汇编成文。

四、信息专家模式

解决方案：

将职责分配给拥有履行一个职责所必需信息的类，也就是信息专家。

问题：

在开始分配职责的时候，首先要清晰的陈述职责。

假定某个类需要知道一次销售的总额。

根据专家模式，我们应该寻找一个对象类，它具有计算总额所需要的信息。

关键：

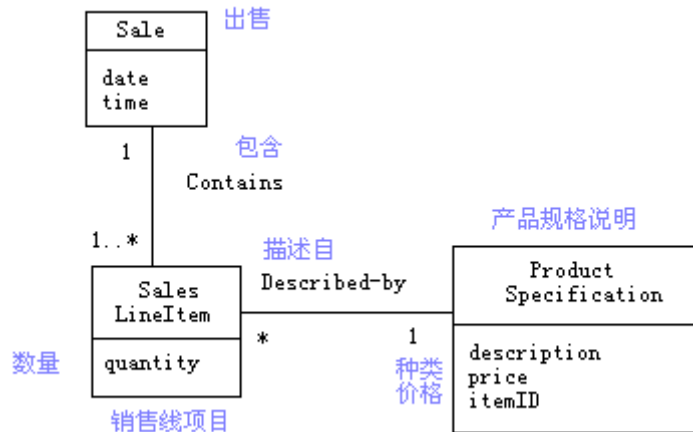
使用概念模型（现实世界领域的概念类）还是设计模型（软件类）来分析所具有所需信息的类呢？

答：

1. 如果设计模型中存在相关的类，先在设计模型中查看。
2. 如果设计模型中不存在相关的类，则查看概念模型，试着应用或者扩展概念模型，得出相应的概念类。

我们下面来讨论一下这个例子。

假定有如下概念模型。



到底谁是信息专家呢？

如果我们需要确定销售总额。

可以看出来，一个 **Sale** 类的实例，将包括“销售线项目”和“产品规格说明”的全部信息。也就是说，**Sale** 类是一个关于销售总额的合适的信息专家。

而 **SalesLineItem** 可以确定子销售额，这就是确定子销售额的信息专家。

进一步，**ProductSpecification** 能确定价格等，它就是“产品规格说明”的信息专家。

上面已经提到，在创建交互图语境的时候，常常出现职责分配的问题。

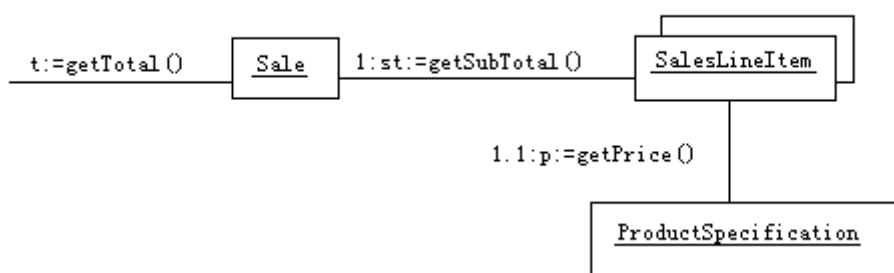
设想我们正在绘设计模型图，并且在为对象分配职责，从软件的角度，我们关注一下：

为了得到总额信息，需要向 **Sale** 发出请求总额请求，于是 **Sale** 得到了 `getTotal` 方法。

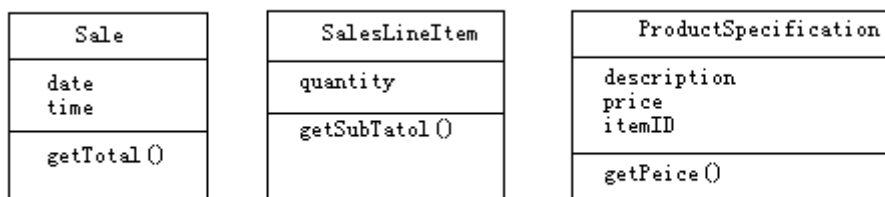
而销售需要取得数量信息，就要向“销售线项目”发出请求，这就在 **SalesLineItem** 得到了 `getSubtotal` 方法。

而销售线项目需要向“产品规格说明”取得价格信息，这就在 **ProductSpecification** 类得到了 `getPrice` 方法。

这样的思考，我们就在概念模型的基础上，得到了设计模型。



下面为设计类



注意:

职责的实现需要信息，而信息往往分布在不同的对象中，这就意味着需要许多“部分”的信息专家来协作完成一个任务。

信息专家模式于现实世界具有相似性，它往往导致这样的设计：软件对象完成它所代表的现实世界对象的机械操作。

但是，某些情况下专家模式所描述的解决方案并不合适，这主要会造成耦合性和内聚性的一些问题。后面我们会加以讨论。

五、创建者模式

解决方案:

如果符合下面一个或者多个条件，则可以把创建类 A 的职责分配给类 B。

- 1，类 B 聚和类 A 的对象。
- 2，类 B 包含类 A 的对象。
- 3，类 B 记录类 A 的对象的实例。
- 4，类 B 密切使用类 A 的对象。
- 5，类 B 初始化数据并在创建类 A 的实例的时候传递给类 A（因此，类 B 是创建类 A 实例的一个专家）。

如果符合多个条件，类 B 聚合或者包含类 A 的条件优先。

问题:

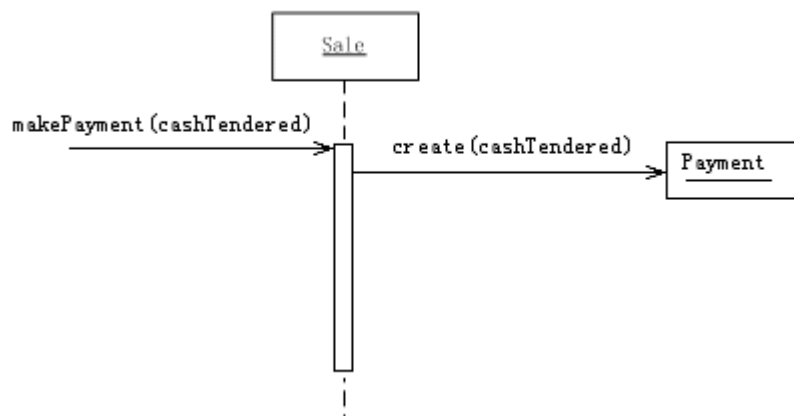
谁应该负责产生类的实例？

创建对象是面向对象系统最普遍的活动之一，因此，拥有一个分配创建对象职责的通用原则

是非常有用的。如果职责分配合理，设计就能降低耦合度，提高设计的清晰度、封装性和重用性。

讨论：

创建者模式指导怎样分配和创建对象（一个非常重要的任务）相关的职责。
通过下面的交互图，我们立刻就能发现 Sale 具备 Payment 创建者的职责。



创建者模式的一个基本目的，就是找到一个在任何情况下都与被创建对象相关联的创建者，选择这样的类作为创建者能支持低耦合。

限制：

创建过程经常非常复杂，在这种情况下，最好的办法是把创建委托给一个工厂，而不是使用创建者模式所建议的类。

六、低耦合模式

解决方案：

分配一个职责，是的保持低耦合度。

问题：

怎样支持低的依赖性，减少变更带来的影响，提高重用性？

耦合（coupling）是测量一个元素连接、了解或者依赖其它元素强弱的尺度。具有低耦合的元素不过多的依赖其它的元素，“过多”这个词和元素所处的语境有关，需要进行考查。

元素包括类、子系统、系统等。

具有高耦合性地类过多的依赖其它的类，设计这种高耦合的类是不受欢迎的。因为它可能出现以下问题：

- 相关类的变化强制局部变化。
- 当元素分离出来的时候很难理解
- 因为使用高耦合类的时候需要它所依赖的类，所以很难重用。

示例：

我们来看一下订单处理子系统中的一个例子，有如下三个类。

Payment（付款）

Register（登记）

Sale（销售）

要求：创建一个 Payment 类的实例，并且与 Sale 相关联。

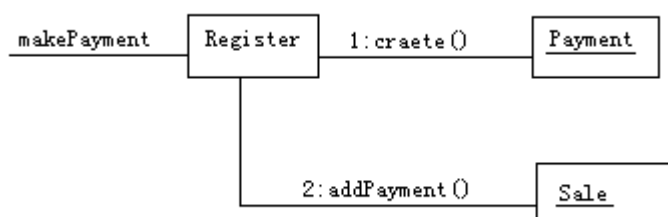
哪个类更适合完成这项工作呢？

创建者模式认为，Register 记录了现实世界中的一次 Payment，因此建议用 Register 作为创建者。

第一方案：

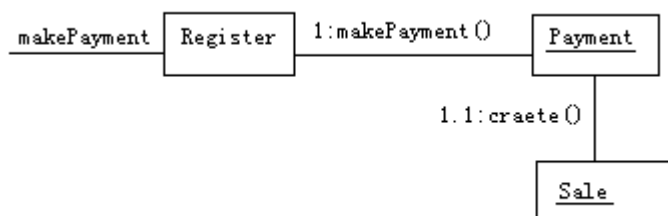
由 Register 构造一个 Payment 对象。

再由 Register 把构造的 Payment 实例通过 addPayment 消息发送给 Sale 对象。



第二方案：

由 Register 向 Sale 提供付款信息（通过 makePayment 消息），再由 Sale 创建 Payment 对象。



两种方案到底那种支持低的耦合度呢？

第一方案，Register 构造一个 Payment 对象，增加了 Register 与 Payment 对象的耦合度。

第二方案，Payment 对象是由 Sale 创建的，因此并没有增加 Register 与 Payment 对象的耦合度。

单纯从耦合度来考虑，第二种方案更优。

在实际工作中，耦合度往往和其它模式是矛盾的。但耦合性是提高设计质量必须考虑的一个因素。

讨论：

在确定设计方案的过程中，低耦合是一个应该时刻铭记于心的原则。它是一个应该时常考虑的设计目标，在设计者评估设计方案的时候，低耦合也是一个评估原则。

低耦合使类的设计更独立，减少类的变更带来的不良影响，但是，我们会时时发现低耦合的要求，是和其它面向对象的设计要求是矛盾的，这就不能把它看成唯一的原则，而是众多原则中的一个重要的原则。

比如继承性必然导致高的耦合性，但不用继承性，这就失去了面向对象设计最重要的特点。

没有绝对的尺度来衡量耦合度，关键是开发者能够估计出来，当前的耦合度会不会导致问题。事实上越是表面上简单而且一般化的类，往往具有强的可重用性和低的耦合度。

低耦合度的需要，导致了一个著名的设计原则，那就是优先使用组合而不是继承。但这样又会导致许多臃肿、复杂而且设计低劣的类的产生。

所以，一个优秀的设计师，关键是用一种深入理解和权衡利弊的态度来面对设计。

设计师的灵魂不是记住了多少原则，而是能灵活合理的使用这些原则，这就需要在大量的设计实践中总结经验，特别是在失败中总结教训，来形成自己的设计理念。

设计师水平的差距，正在于此。

七、高内聚模式

解决方案：

分配一个职责，使得保持高的内聚。

问题：

怎样才能使得复杂性可以管理？

从对象设计的角度，内聚是一个元素的职责被关联和关注的强弱尺度。如果一个元素具有很多紧密相关的职责，而且只完成有限的功能，那这个元素就是高度内聚的。这些元素包括类、子系统等等。

一个具有低内聚的类会执行许多互不相关的事物，或者完成太多的功能，这样的类是不可取的，因为它们会导致以下问题：

- 1， 难于理解。
- 2， 难于重用。
- 3， 难于维护。
- 4， 系统脆弱，常常受到变化带来的困扰。

低内聚类常常代表抽象化的“大粒度对象”，或者承担着本来可以委托给其它对象的职责。

示例：

我们还是来看一下刚刚讨论过的订单处理子系统的例子，有如下三个类。

Payment（付款）

Register（登记）

Sale（销售）

要求：创建一个 Payment 类的实例，并且与 Sale 相关联。

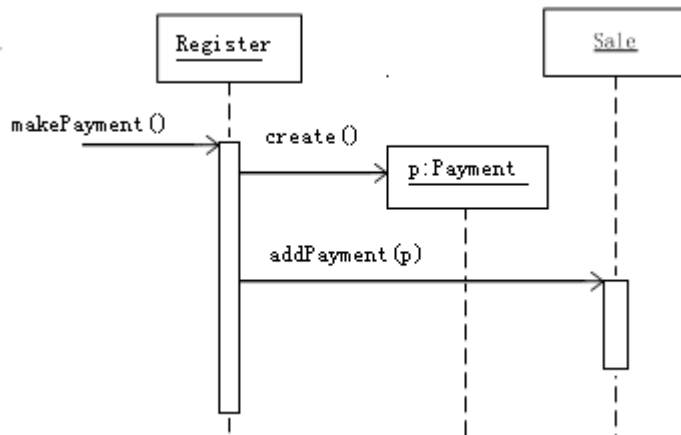
哪个类更适合完成这项工作呢？

创建者模式认为，Register 记录了现实世界中的一次 Payment，因此建议用 Register 作为创建者。

第一方案：

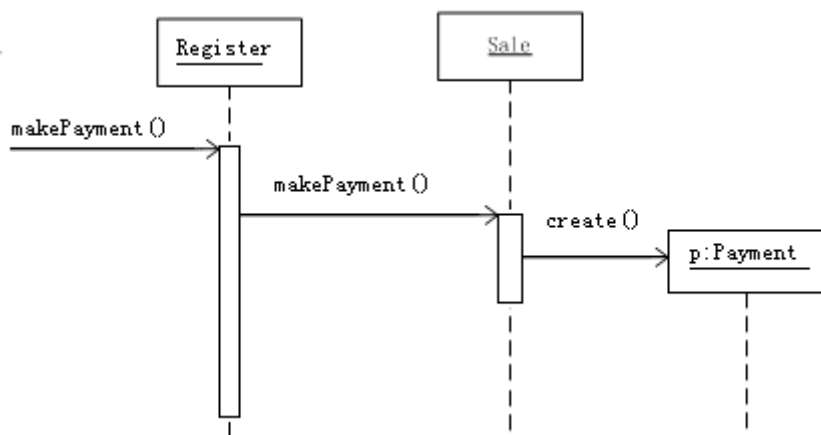
由 Register 构造一个 Payment 对象。

再由 Register 把构造的 Payment 实例通过 addPayment 消息发送给 Sale 对象。



第二方案：

由 Register 向 Sale 提供付款信息（通过 makePayment 消息），再由 Sale 创建 Payment 对象。



在第一个方案中，由于 Register 要执行多个任务，在任务很多的时候，就会显得十分臃肿，这种要执行多个任务的类，内聚是比较低的。

在第二种方案里面，由于创建 Payment 对象的任务，委托给了 Sale，每个类的任务都比较简单而且单一，这就实现了高的内聚性。

从开发技巧的角度，至少有一个开发者要去考虑内聚所产生的影响。

一般来说，高的内聚往往导致低的耦合度。

讨论：

和低耦合性模式一样，高内聚模式在制定设计方案的过程中，一个应该时刻铭记于心的原则。

同样，它往往会和其它的设计原则相抵触，因此必须综合考虑。

Grady Booch 是建模的大师级人物，它在描述高内聚的定义的时候是这样说的：“一个组件（比如类）的所有元素，共同协作提供一些良好受限的行为。”

根据经验，一个具有高内聚的类，具有数目相对较少的方法，和紧密相关的功能。它并不完成太多的工作，当需要实现的任务过大的时候，可以和其它的对象协作来分担过大的工作量。

一个类具有高内聚是非常有利的，因为它对于理解、维护和重用都相对比较容易。

限制：

少数情况下，接受低内聚是合理的。

比如，把 SQL 专家编写的语句综合在一个类里面，这就可以使程序设计专家不必要特别关注 SQL 语句该怎么写。

又比如，远程对象处理，利用很多细粒度的接口与客户联系，造成网络流量大幅度增加而降低性能，就不如把能力封装起来，做一个粗粒度的接口给客户，大部分工作在远程对象内部完成，减少远程调用的次数。

第二节 设计模式与软件架构

一、设计模式

在模块设计阶段，最关键的问题是，用户需求是变化的，我们的设计如何适应这种变化呢？

- 1，如果我们试图发现事情怎样变化，那我们将永远停留在分析阶段。
- 2，如果我们编写的软件能面向未来，那将永远处在设计阶段。
- 3，我们的时间和预算不允许我们面向未来设计软件。过分的分析和过分的的设计，事实上被称之为“分析瘫痪”。

如果我们预料到变化将要发生，而且也预料到将会在哪里发生。这样就形成了几个原则：

- 1，针对接口编程而不是针对实现编程。
- 2，优先使用对象组合，而不是类的继承。
- 3，考虑您的设计哪些是可变的，注意，不是考虑什么会迫使您的设计改变，而是考虑要素变化的时候，不会引起重新设计。

也就是说，封装变化的概念是模块设计的主题。

解决这个问题，最著名的要数 GoF 的 23 种模式，在 GoF 中，把设计模式分为结构型、创建型和行为型三大类。

本课程假定学员已经熟悉这 23 个模式，因此主要从设计的角度讨论如何正确选用恰当的设计模式。

整个讨论依据三个原则：

- 1) 开放-封闭原则
- 2) 从场景进行设计的原则
- 3) 包容变化的原则

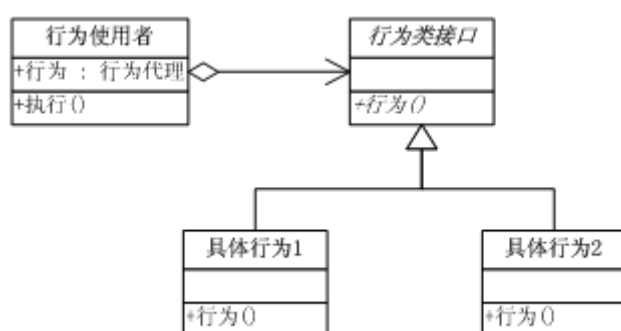
下面的讨论会有一些代码例子，尽管在详细设计的时候，并不考虑代码实现的，但任何架构设计思想如果没有代码实现做基础，将成为无木之本，所以后面的几个例子我们还是把代码实现表示出来，举这些例子的目的并不是提供样板，而是希望更深入的描述想法。

另外，所用的例子大部分使用 C#来编写，这主要因为希望表达比较简单，但这不是必要的，可以用任何面向对象的语言（Java、C++）来讨论这些问题。

二、封装变化与面向接口编程

设计模式分为结构型、构造型和行为型三种问题域，我们来看一下行为型设计模式，行为型设计模式的要点之一是“封装变化”，这类模式充分体现了面向对象的设计的抽象性。在这类模式中，“动作”或者叫“行为”，被抽象后封装为对象或者为方法接口。通过这种抽象，将使“动作”的对象和动作本身分开，从而达到降低耦合性的效果。这样一来，使行为对象可以容易的被维护，而且可以通过类的继承实现扩展。

行为型模式大多数涉及两种对象，即封装可变化特征的新对象，和使用这些新对象的已有的对象。二者之间通过对象组合在一起工作。如果不使用这些模式，这些新对象的功能就会变成这些已有对象的难以分割的一部分。因此，大多数行为型模式具有如下结构。



下面是上述结构的代码片断：

```

public abstract class 行为类接口
{
    public abstract void 行为();
}
public class 具体行为1:行为类接口
{
    public override void 行为()
    {
    }
}
public class 行为使用者
{
    public 行为类接口 我的行为;
    public 行为使用者()
    {
        我的行为=new 具体行为1();
    }
    public void 执行()
    {
        我的行为. 行为();
    }
}
  
```



```
}
}
```

第三节 合理使用外观和适配器模式

一、使用外观模式（Facade）使用户和子系统绝缘

外观模式为了一组子系统提供一个一致的方式对外交互。这样就可以使客户和子系统绝缘，可以大大减少客户处理对象的数目，我们已经讨论过这个主题，这里就不再讨论了。

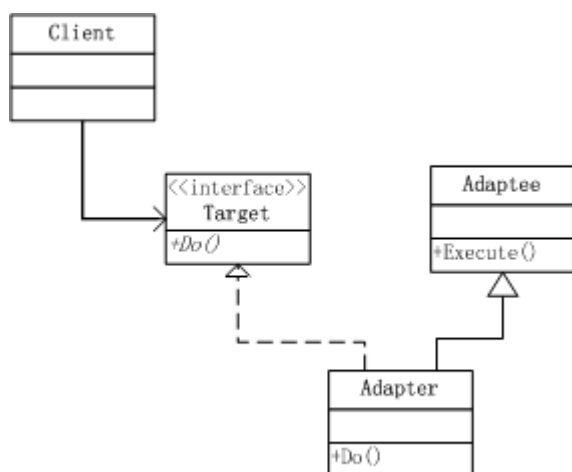
二、使用适配器模式（Adapter）调适接口

在系统之间集成的时候，最常见的问题是接口不一致，很多能满足功能的软件模块，由于接口不同，而导致无法使用。

适配器模式的含义在于，把一个类的接口转换为另一个接口，使原本不兼容而不能一起工作的类能够一起工作。适配器有类适配器和对象适配器两种类型，二者的意图相同，只是实现的方法和适用的情况不同。类适配器采用继承的方法来实现，而对象适配器采用组合的方法来实现。

1) 类适配器

类适配器采用多重继承对一个接口与另一个接口进行匹配，其结构如下。

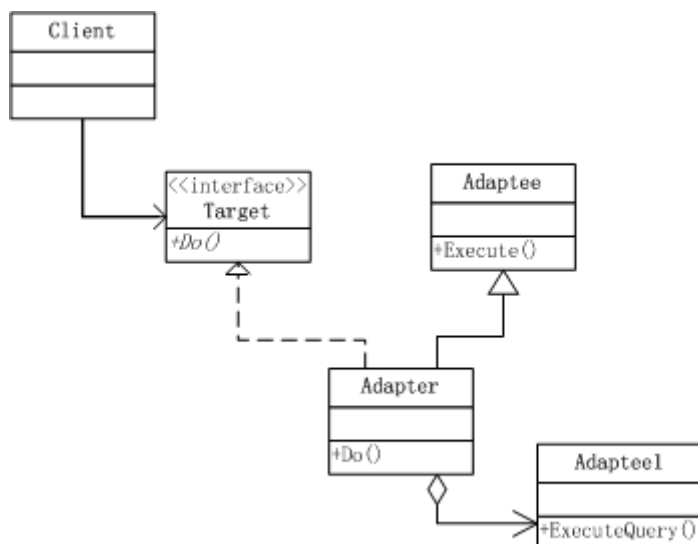


由于 Adapter 类继承了 Adaptee 类，通过接口的 Do() 方法，我们可以使用 Adaptee 中的 Execute 方法。

这种方式当语言不承认多重继承的时候就没有办法实现，这时可以使用对象适配器。

2) 对象适配器

对象适配器采用对象组合，通过引用一个类与另一个类的接口，来实现对多个类的适配。



第四节 封装变化的三种方式及评价

设计可升级的架构，关键是要把模块中不变部分与预测可变部分分开，以防止升级过程中对基本代码的干扰。这种分开可以有多种方式，一般来说可以从纵向、横向以及外围三个方面考虑。

一、纵向处理：模板方法（Template Method）

1、意图

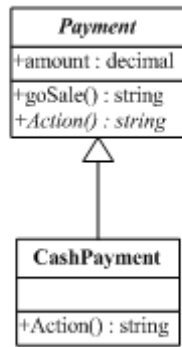
定义一个操作中的算法骨架，而将一些步骤延伸到子类中去，使得子类可以不改变一个算法的结构，即可重新定义改算法的某些特定步骤。这里需要复用的使算法的结构，也就是步骤，而步骤的实现可以在子类中完成。

2、使用场合

- 1) 一次性实现一个算法的不变部分，并且将可变的行为留给子类来完成。
- 2) 各子类公共的行为应该被提取出来并集中到一个公共父类中以避免代码的重复。首先识别现有代码的不同之处，并且把不同部分分离为新的操作，最后，用一个调用这些新的操作的模板方法来替换这些不同的代码。
- 3) 控制子类的扩展。

3、结构

模板方法的结构如下：



在抽象类中定义模板方法的关键是：

在一个非抽象方法中调用调用抽象方法，而这些抽象方法在子类中具体实现。

代码：

```

public abstract class Payment
{
    private decimal amount;

    public decimal Amount
    {
        get
        {
            return amount;
        }
        set
        {
            amount = value;
        }
    }

    public virtual string goSale()
    {
        string x = "不变的流程一 ";
        x += Action(); //可变的流程
        x += amount + ", 正在查询库存状态"; //属性和不变的流程二
        return x;
    }

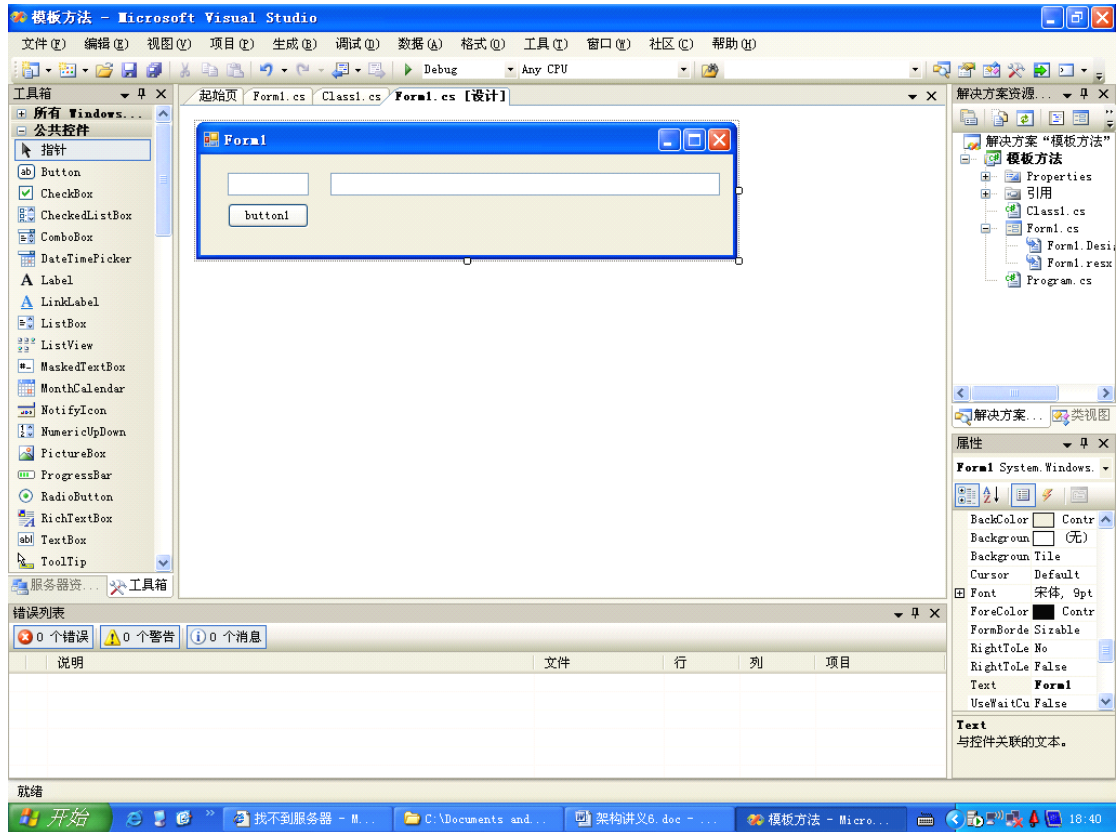
    public abstract string Action();
}

public class CashPayment : Payment
{
    public override string Action()
  
```

```

{
    return "现金支付";
}
}

```



调用:

```

Payment o;

private void button1_Click(object sender, EventArgs e)
{
    o = new CashPayment();
    if (textBox1.Text != "")
    {
        o.Amount = decimal.Parse(textBox1.Text);
    }
    textBox2.Text = o.goSale();
}

```

假定系统已经投运，用户提出新的需求，要求加上信用卡支付和支票支付，可以这样写：

```

public class CreditPayment : Payment

```

```
{  
    public override string Action()  
    {  
        return "信用卡支付, 联系支付机构";  
    }  
}
```

```
public class CheckPayment : Payment  
{  
    public override string Action()  
    {  
        return "支票支付, 联系财务部门";  
    }  
}
```

调用:

```
Payment o;  
//现金方式  
private void button1_Click(object sender, EventArgs e)  
{  
    o = new CashPayment();  
    if (textBox1.Text != "")  
    {  
        o.Amount = decimal.Parse(textBox1.Text);  
    }  
    textBox2.Text = o.goSale();  
}  
//信用卡方式  
private void button2_Click(object sender, EventArgs e)  
{  
    o = new CreditPayment();  
    if (textBox1.Text != "")  
    {  
        o.Amount = decimal.Parse(textBox1.Text);  
    }  
    textBox2.Text = o.goSale();  
}  
//支票方式  
private void button3_Click(object sender, EventArgs e)  
{  
    o = new CheckPayment();  
    if (textBox1.Text != "")  
    {  
        o.Amount = decimal.Parse(textBox1.Text);  
    }  
}
```

```

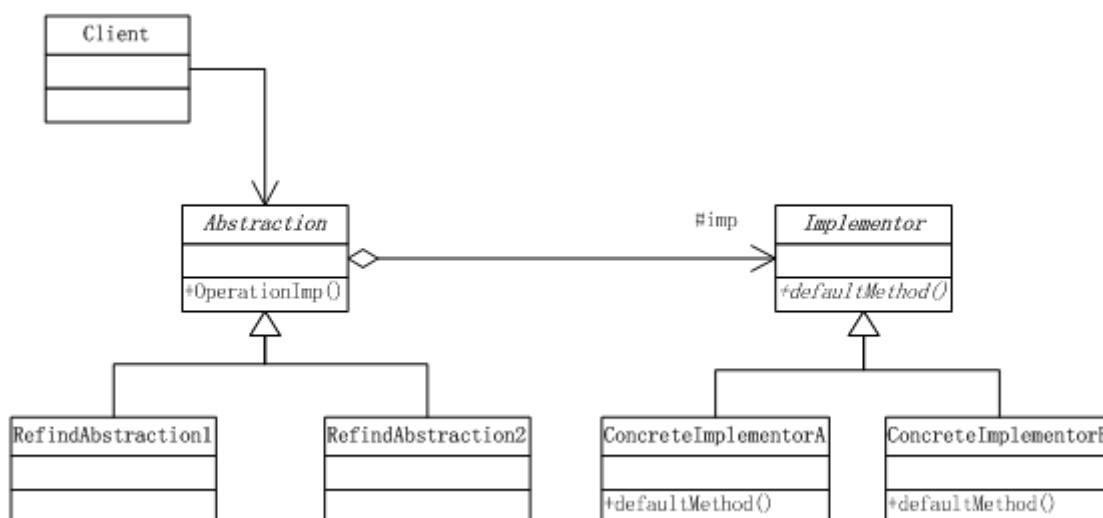
    }
    textBox2.Text = o.goSale();
}

```

二、横向处理：桥接模式（Bridge）

模板方法是利用继承来完成切割，当对耦合性要求比较高，无法使用继承的时候，可以横向切割，也就是使用桥接模式。

桥接模式结构如下图。



其中：

Abstraction: 定义抽象类的接口，并维护 Implementor 接口的指针。

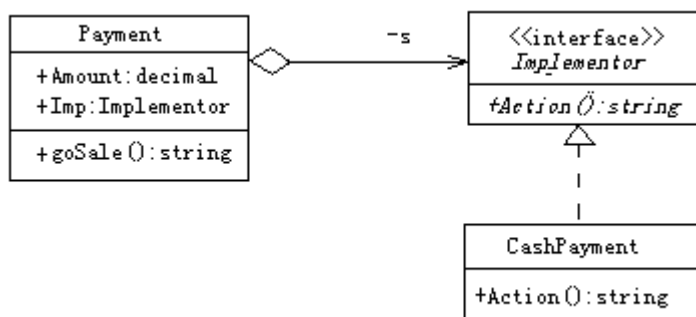
其内部有一个 OperationImp 实例方法，但使用 Implementor 接口的方法。

RefindAbstraction: 扩充 Abstraction 类定义的接口，重要的是需要实例化 Implementor 接口。

Implementor: 定义实现类的接口，关键是内部有一个 defaultMethod 方法，这个方法会被 OperationImp 实例方法使用。

ConcreteImplementor: 实现 Implementor 接口，这是定义它的具体实现。

我们通过上面的关于支付的简单例子可以说明它的原理。



```
public class Payment
{
    private decimal amount;

    public decimal Amount
    {
        get
        {
            return amount;
        }
        set
        {
            amount = value;
        }
    }

    Implementor s;

    public Implementor Imp
    {
        set
        {
            s = value;
        }
    }

    public virtual string goSale()
    {
        string x = "不变的流程一 ";
        x += s.Action();    //可变的流程
        x += amount + ", 正在查询库存状态"; //属性和不变的流程二
        return x;
    }
}

public interface Implementor
{
    string Action();
}

public class CashPayment : Implementor
{
    public string Action()
    {

```

```

        return "现金支付";
    }
}

```

调用:

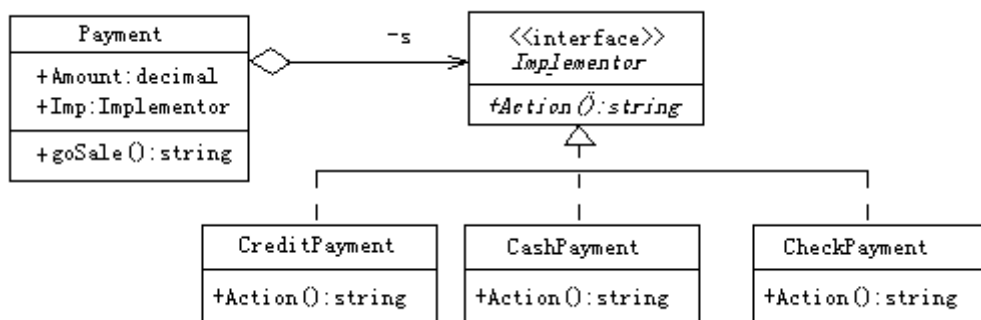
```
Payment o=new Payment();
```

```

private void button1_Click(object sender, EventArgs e)
{
    o.Imp = new CashPayment();
    if (textBox1.Text != "")
    {
        o.Amount = decimal.Parse(textBox1.Text);
    }
    textBox2.Text = o.goSale();
}

```

假定系统投运以后, 需要修改性能, 可以直接加入新的类:



```

public class CreditPayment : Implementor
{
    public string Action()
    {
        return "信用卡支付, 联系机构";
    }
}

```

```

public class CheckPayment : Implementor
{
    public string Action()
    {
        return "支票支付, 联系财务部";
    }
}

```


调用：

```
Payment o=new Payment();

private void button1_Click(object sender, EventArgs e)
{
    o.Imp = new CashPayment();
    if (textBox1.Text != "")
    {
        o.Amount = decimal.Parse(textBox1.Text);
    }
    textBox2.Text = o.goSale();
}

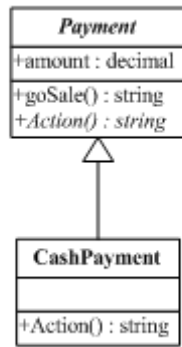
private void button2_Click(object sender, EventArgs e)
{
    o.Imp = new CreditPayment();
    if (textBox1.Text != "")
    {
        o.Amount = decimal.Parse(textBox1.Text);
    }
    textBox2.Text = o.goSale();
}

private void button3_Click(object sender, EventArgs e)
{
    o.Imp = new CheckPayment();
    if (textBox1.Text != "")
    {
        o.Amount = decimal.Parse(textBox1.Text);
    }
    textBox2.Text = o.goSale();
}
```

这样就减少了系统的耦合性。而在系统升级的时候，并不需要改变原来的代码。

三、核心和外围：装饰器模式（Decorator）

有的时候，希望实现一个基本的核心代码块，由外围代码实现专用性能的包装，最简单的方法，是使用继承。



```
public abstract class Payment
{
    private decimal amount;

    public decimal Amount
    {
        get
        {
            return amount;
        }
        set
        {
            amount = value;
        }
    }

    public virtual string goSale()
    {
        return Action() + "完成, 金额为" + amount + ", 正在查询库存状态";
    }

    public abstract string Action();
}

public class CashPayment : Payment
{
    public override string Action()
    {
        return "现金支付";
    }
}
```

```
}

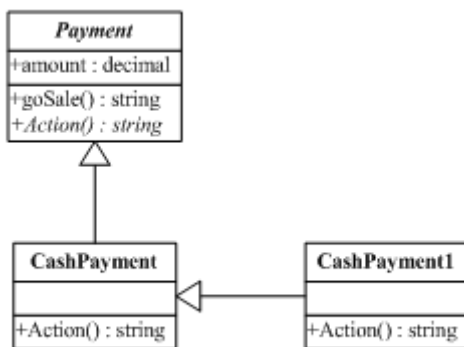
```

实现：

```
Payment o;
private void button1_Click(object sender, EventArgs e)
{
    o = new CashPayment1();
    o.Amount = decimal.Parse(textBox1.Text);
    textBox2.Text = o.goSale();
}

```

加入继承：



```
public class CashPayment1 : CashPayment
{
    public override string Action()
    {
        //在执行原来的代码之前，弹出提示框
        System.Windows.Forms.MessageBox.Show("现金支付");
        return base.Action();
    }
}

```

实现：

```
Payment o;
private void button1_Click(object sender, EventArgs e)
{
    o = new CashPayment1();
    o.Amount = decimal.Parse(textBox1.Text);
    textBox2.Text = o.goSale();
}

```

缺点：

继承层次多一层，提升了耦合性。

当实现类比较多时，实现起来就比较复杂。

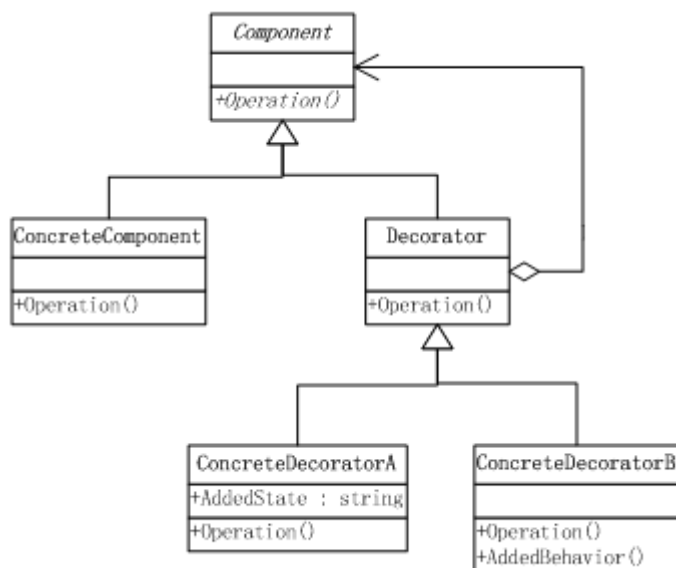
在这样的情况下，也可以使用装饰器模式，这是用组合取代继承的一个很好的方式。

1、意图

事实上，上面所要解决的意图可以归结为“在不改变对象的前提下，动态增加它的功能”，也就是说，我们不希望改变原有的类，或者采用创建子类的方式来增加功能，在这种情况下，可以采用装饰模式。

2、结构

装饰器结构的一个重要的特点是，它继承于一个抽象类，但它又使用这个抽象类的聚合（即装饰类对象可以包含抽象类对象），恰当的设计，可以达到我们提出来的目的。



模式中的参与者如下：

Component (组成): 定义一个对象接口，可以动态添加这些对象的功能，其中包括 **Operation**(业务)方法。

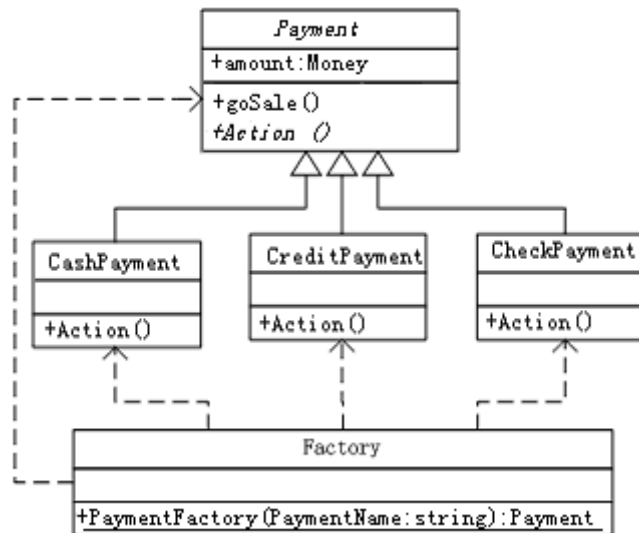
ConcreteComponent(具体组成): 定义一个对象，可以为它添加一些功能。

Decorator(装饰): 维持一个对 **Component** 对象的引用，并定义与 **Component** 接口一致的接口。

ConcreteDecorator (具体装饰): 为组件添加功能。它可能包括 **AddedBehavior**(更多的行为)和 **AddedState**(更多的状态)。

举个例子：

假定我们已经构造了一个基于支付的简单工厂模式的系统。



```

using System;
using System.Collections.Generic;
using System.Text;

namespace PaymentDemo
{
    public abstract class Payment
    {
        private decimal amount;

        public decimal Amount
        {
            get
            {
                return amount;
            }
            set
            {
                amount = value;
            }
        }

        public string goSale()
        {
            return Action() + "完成, 金额为" + amount + ", 正在查询库存状态";
        }
    }
}

```

```
    public abstract string Action();  
}
```

```
public class CashPayment : Payment  
{  
    public override string Action()  
    {  
        return "现金支付";  
    }  
}
```

```
public class CreditPayment : Payment  
{  
    public override string Action()  
    {  
        return "信用卡支付";  
    }  
}
```

```
public class CheckPayment : Payment  
{  
    public override string Action()  
    {  
        return "支票支付";  
    }  
}
```

```
}
```

工厂类，注意这是独立的模块。

```
using System;  
using System.Collections.Generic;  
using System.Text;  
  
namespace PaymentDemo  
{  
    //这是一个工厂类  
    public class Factory  
    {  
        public static Payment PaymentFactory(string PaymentName)  
        {  
            Payment mdb=null;  
            switch (PaymentName)
```

```

        {
            case "现金":
                mdb=new CashPayment();
                break;
            case "信用卡":
                mdb=new CreditPayment();
                break;
            case "支票":
                mdb=new CheckPayment();
                break;
        }
        return mdb;
    }
}
}

```

调用:

```
Payment obj;
```

```

private void button1_Click(object sender, EventArgs e)
{
    obj = Factory.PaymentFactory("现金");
    obj.Amount = decimal.Parse(textBox1.Text);
    textBox2.Text = obj.goSale();
}

```

```

private void button2_Click(object sender, EventArgs e)
{
    obj = Factory.PaymentFactory("信用卡");
    obj.Amount = decimal.Parse(textBox1.Text);
    textBox2.Text = obj.goSale();
}

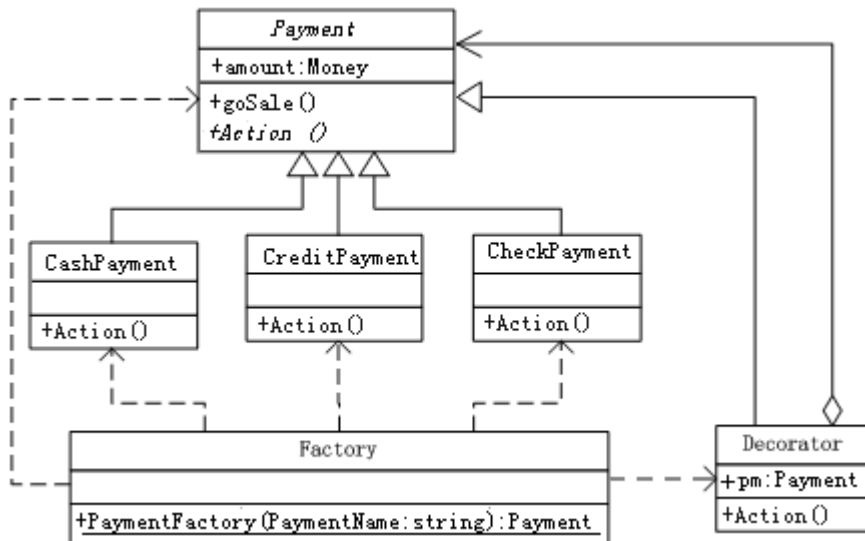
```

```

private void button3_Click(object sender, EventArgs e)
{
    obj = Factory.PaymentFactory("支票");
    obj.Amount = decimal.Parse(textBox1.Text);
    textBox2.Text = obj.goSale();
}

```

现在需要每个类在调用方法 goSale()的时候,除了完成原来的功能以外,先弹出一个对话框,显示工厂的名称,而且不需要改变来的系统,为此,在工厂类的模块种添加一个装饰类 Decorator,同时略微的改写一下工厂类的代码。



//这是一个工厂类

```

public class Factory
{
    public static Payment PaymentFactory(string PaymentName)
    {
        Payment mdb=null;
        switch (PaymentName)
        {
            case "现金":
                mdb=new CashPayment();
                break;
            case "信用卡":
                mdb=new CreditPayment();
                break;
            case "支票":
                mdb=new CheckPayment();
                break;
        }
        //return mdb;

        //下面是实现装饰模式新加的代码
        Decorator obj = new Decorator(PaymentName);
        obj.Pm = mdb;
        return obj;
    }
}
    
```

//装饰类

```

public class Decorator : Payment
    
```



```
{
    string strName;
    public Decorator(string strName)
    {
        this.strName = strName;
    }
    Payment pm;
    public Payment Pm
    {
        get
        {
            return pm;
        }
        set
        {
            pm = value;
        }
    }

    public override string Action()
    {
        //在执行原来的代码之前，弹出提示框
        System.Windows.Forms.MessageBox.Show(strName);
        return pm.Action();
    }
}
```

这就可以在用户不知晓的情况下，也不更改原来的类的情况下，改变了性能。

第五节 利用策略与工厂模式实现通用的框架

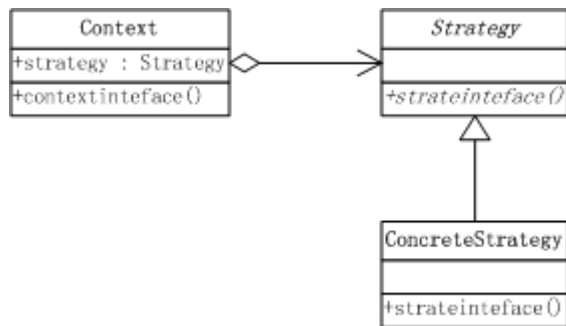
一、应用策略模式提升层的通用性

1、意图

将算法封装，使系统可以更换或扩展算法，策略模式的关键是所有子类的目标一致。

2、结构

策略模式的结构如下。



其中：Strategy（策略）：抽象类，定义需要支持的算法接口，策略由上下文接口调用。

3、示例：

目标：在 C# 下构造通用的框架，就需要使用 XML 配置文件技术。

构造一个类容器框架，可以动态装入和构造对象，装入类可以使用配置文件，这里利用了反射技术。

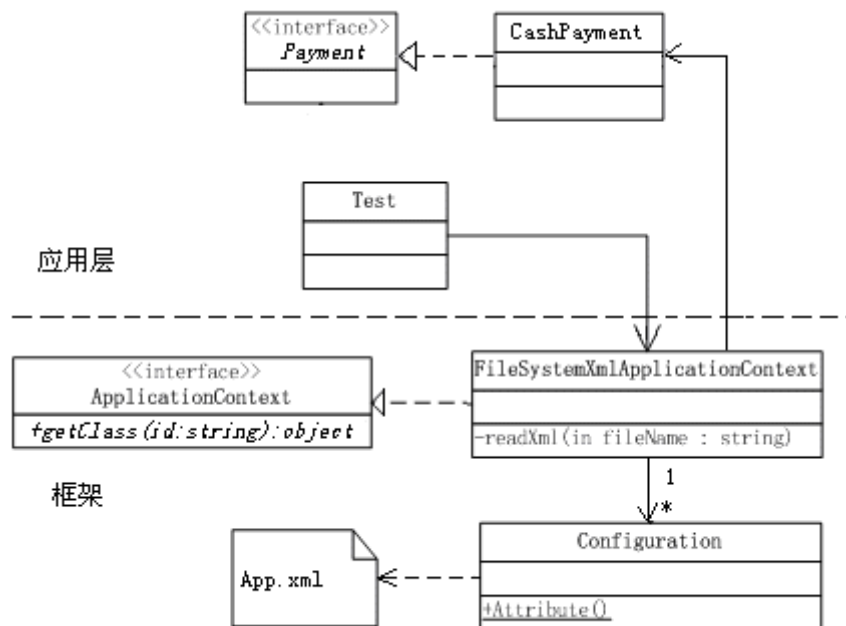
问题：

如何动态构造对象，集中管理对象。

解决方案：

策略模式，XML 文档读入，反射的应用。

我们现在要处理的架构如下：



App.xml 文档的结构如下。

```

<configuration>
<description>说明</description>
<class id="标记" type="类名,dll文件名">
  <property name="属性名">
    <value>属性值</value>
  
```

```

</property>
.....
</class>
</configuration>

```

应用程序上下文接口: ApplicationContext

只有一个方法, 也就是由用户提供的id提供类的实例。

代码:

```

using System;
using System.Collections.Generic;
using System.Text;
using System.Windows.Forms;
using System.Collections;
using System.Xml;
using System.Reflection;

namespace 处理类
{
    public interface ApplicationContext
    {
        Object getClass(string id);
    }

    public class FileSystemXmlApplicationContext : ApplicationContext
    {
        //用一个哈希表保留从XML读来的数据
        private Hashtable hs = new Hashtable();

        public FileSystemXmlApplicationContext(string fileName)
        {
            try
            {
                readXml(fileName);
            }
            catch (Exception e)
            {
                MessageBox.Show(e.ToString());
            }
        }

        //私有的读XML方法。
        private void readXml(String fileName)
        {
            //读XML把数据放入哈希表

```

```

        hs = Configuration.Attribute(fileName, "class", "property");
    }

    public Object getClass(string id)
    {
        //由id取出内部的哈希表对象
        Hashtable hsb = (Hashtable)hs[id];
        Object obj = null;
        string m = hsb["type"].ToString();

        int i = m.IndexOf(',');
        string classname = m.Substring(0, i);
        string filename = m.Substring(i + 1) + ".dll";

        //利用反射动态构造对象
        //定义一个公共语言运行库应用程序构造块
        System.Reflection.Assembly MyDll;
        Type[] types;
        MyDll = Assembly.LoadFrom(filename);
        types = MyDll.GetTypes();
        obj = MyDll.CreateInstance(classname);
        Type t = obj.GetType();
        IEnumerator em = hsb.GetEnumerator();
        //指针放在第一个之前
        em.Reset();

        while (em.MoveNext())
        {
            DictionaryEntry s1 = (DictionaryEntry)em.Current;
            if (s1.Key.ToString() != "type")
            {
                string pname = "set_" + s1.Key.ToString();
                t.InvokeMember(pname, BindingFlags.InvokeMethod, null,
obj, new object[] { s1.Value });
            }
        }
        return obj;
    }
}

//这是一个专门用于读配置文件的类
class Configuration
{
    public static Hashtable Attribute(String configname,

```

```
String mostlyelem,
String childmostlyelem)
{
    Hashtable hs = new Hashtable();
    XmlDocument doc = new XmlDocument();
    doc.Load(configname);

    //建立所有元素的列表
    XmlElement root = doc.DocumentElement;

    //把所有的主要标记都找出来放到节点列表中
    XmlNodeList elemList = root.GetElementsByTagName(mostlyelem);

    for (int i = 0; i < elemList.Count; i++)
    {
        //获取这个节点的属性集合
        XmlAttributeCollection ac = elemList[i].Attributes;

        //构造一个表，记录属性和类的名字
        Hashtable hs1 = new Hashtable();
        hs1.Add("type", ac["type"].Value);

        //获取二级标记子节点
        XmlNodeList elemList1 =
        ((XmlElement)elemList[i]).GetElementsByTagName(childmostlyelem);

        for (int j = 0; j < elemList1.Count; j++)
        {
            //获取这个节点的属性集合
            XmlAttributeCollection ac1 = elemList1[j].Attributes;
            string key = ac1["name"].Value;

            XmlNodeList e1 =
            ((XmlElement)elemList1[j]).GetElementsByTagName("value");
            string value = e1[0].InnerText;
            hs1.Add(key, value);
        }
        hs.Add(ac["id"].Value, hs1);
    }
    return hs;
}
}
```

做一个抽象类类库：AbstractPayment.dll

```
using System;
using System.Collections.Generic;
using System.Text;

namespace AbstractPayment
{
    public abstract class Payment
    {
        private decimal amount;

        public string Amount
        {
            get
            {
                return amount.ToString();
            }
            set
            {
                amount = decimal.Parse(value);
            }
        }

        public virtual string goSale()
        {
            return Action() + "完成, 金额为" + amount + ", 正在查询库存状态";
        }

        public abstract string Action();
    }
}
```

实现类：Cash.dll

```
using System;
using System.Collections.Generic;
using System.Text;

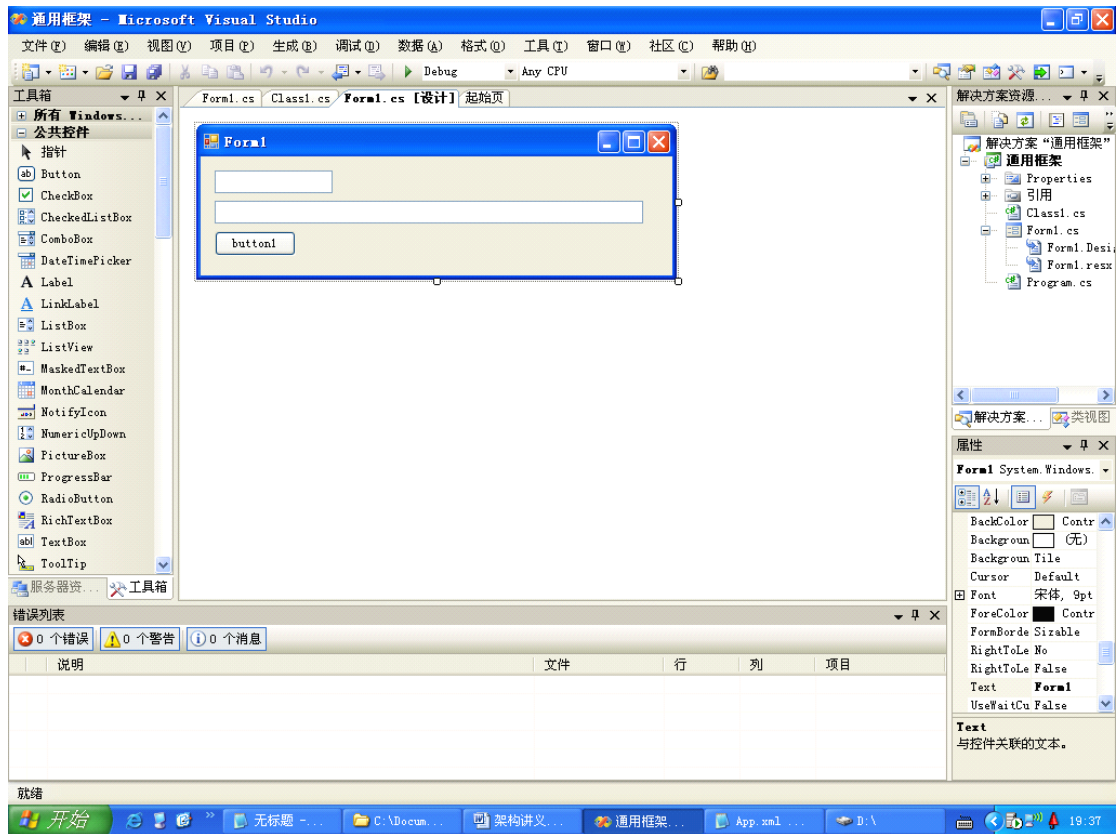
namespace Cash
{
    public class CashPayment : AbstractPayment.Payment
    {
        public override string Action()
        {
            // TODO: Implement the Action method
        }
    }
}
```

```

    {
        return "现金支付";
    }
}
}

```

界面：



添加引用：AbstractPayment.dll

配置文件：App.xml

```

<configuration>
<description></description>
<class id="A" type="Cash.CashPayment,d:\Cash">
  <property name="Amount">
    <value>124</value>
  </property>
</class>
</configuration>

```

使用：

```
ApplicationContext s;
```

```
private void Form1_Load(object sender, EventArgs e)
{
    s = new FileSystemXmlApplicationContext("d:\\App.xml");
}

private void button1_Click(object sender, EventArgs e)
{
    AbstractPayment.Payment m = (AbstractPayment.Payment)s.getClass("A");
    if (textBox1.Text != "")
    {
        m.Amount = textBox1.Text;
    }
    textBox2.Text = m.goSale();
}
```

如果需要添加两个新的实现类: CreditCheck.dll

```
using System;
using System.Collections.Generic;
using System.Text;

namespace CreditCheck
{
    public class CreditPayment : AbstractPayment.Payment
    {
        public override string Action()
        {
            return "信用卡支付";
        }
    }

    public class CheckPayment : AbstractPayment.Payment
    {
        public override string Action()
        {
            return "支票支付";
        }
    }
}
```


把这个类放在当前 D 盘根目录下：

配置文件：

```
<configuration>
<description></description>
<class id="A" type="Cash.CashPayment,d:\Cash">
  <property name="Amount">
    <value>124</value>
  </property>
</class>
<class id="B" type="CreditCheck.CreditPayment,d:\CreditCheck">
  <property name="Amount">
    <value>52000</value>
  </property>
</class>
<class id="C" type="CreditCheck.CheckPayment,d:\CreditCheck">
  <property name="Amount">
    <value>63000</value>
  </property>
</class>
</configuration>
```

实现：

```
ApplicationContext s;

private void Form1_Load(object sender, EventArgs e)
{
    s = new FileSystemXmlApplicationContext("d:\\App.xml");
}

private void button1_Click(object sender, EventArgs e)
{
    AbstractPayment.Payment m = (AbstractPayment.Payment)s.getClass("A");
    if (textBox1.Text != "")
    {
        m.Amount = textBox1.Text;
    }
    textBox2.Text = m.goSale();
}

private void button2_Click(object sender, EventArgs e)
{
}
```

```

        AbstractPayment.Payment m = (AbstractPayment.Payment)s.getClass("B");
        if (textBox1.Text != "")
        {
            m.Amount = textBox1.Text;
        }
        textBox2.Text = m.goSale();
    }

    private void button3_Click(object sender, EventArgs e)
    {
        AbstractPayment.Payment m = (AbstractPayment.Payment)s.getClass("C");
        if (textBox1.Text != "")
        {
            m.Amount = textBox1.Text;
        }
        textBox2.Text = m.goSale();
    }
}

```

同样的原理，Java 实现的 Bean 容器框架

Bean.xml 文档的结构如下。

```

<beans>
<description>说明</description>
<bean id="标记"    class="类名">
    <property name="属性名">
        <value>内容</value>
    </property>
    .....
</bean>
</beans>

```

应用程序上下文接口：ApplicationContext.java

只有一个方法，也就是由用户提供的 id 提供 Bean 的实例。

```
package springdemo;
```

```

public interface ApplicationContext
{
    public Object getBean(String id) throws Exception;
}

```

上下文实现类：FileSystemXmlApplicationContext.java

```
package springdemo;

import java.util.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;
import java.io.*;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.*;

public class FileSystemXmlApplicationContext
    implements ApplicationContext
{

    //用一个哈希表保留从 XML 读来的数据
    private Hashtable hs=new Hashtable();

    public FileSystemXmlApplicationContext()
    {}

    public FileSystemXmlApplicationContext(String fileName)
    {
        try
        {
            readXml(fileName);
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
    //私有的读 XML 方法。
    private void readXml(String fileName) throws Exception
    {
        //读 XML 把数据放入哈希表
        hs=Configuration.Attribute(fileName,"bean","property");
    }

    public Object getBean(String id) throws Exception
    {
        //由 id 取出内部的哈希表对象
        Hashtable hsb=(Hashtable)hs.get(id);
```

```

//利用反射动态构造对象
Object obj =Class.forName(hsb.get("class").toString()).newInstance();
java.util.Enumeration hsNames1 =hsb.keys();
//利用反射写入属性的值
while (hsNames1.hasMoreElements())
{
    //写入利用 Set 方法
    String ka=(String)hsNames1.nextElement();
    if (! ka.equals("class"))
    {
        //写入属性值为字符串
        String m1="String";
        Class[] a1={m1.getClass()};
        //拼接方法的名字
        String sa1=ka.substring(0,1).toUpperCase();
        sa1="set"+sa1+ka.substring(1);
        //动态调用方法
        java.lang.reflect.Method fm=obj.getClass().getMethod(sa1,a1);
        Object[] a2={hsb.get(ka)};
        //通过 set 方法写入属性
        fm.invoke(obj,a2);
    }
}
return obj;
}
}

//这是一个专门用于读配置文件的类
class Configuration
{
    public static Hashtable Attribute(String configname,
        String mostlyelem,
        String childmostlyelem) throws Exception
    {
        Hashtable hs=new Hashtable();
        //建立文档，需要一个工厂
        DocumentBuilderFactory factory=DocumentBuilderFactory.newInstance();
        DocumentBuilder builder=factory.newDocumentBuilder();
        Document doc=builder.parse(configname);

        //建立所有元素的列表
        Element root = doc.getDocumentElement();

        //把所有的主要标记都找出来放到节点列表中
        NodeList elemList = root.getElementsByTagName(mostlyelem);

```

```

for (int i=0; i < elemList.getLength(); i++)
{
    //获取这个节点的属性集合
    NamedNodeMap ac = elemList.item(i).getAttributes();
    //构造一个表，记录属性和类的名字
    Hashtable hs1=new Hashtable();
    hs1.put("class",ac.getNamedItem("class").getNodeValue());
    //获取二级标记子节点
    Element node=(Element)elemList.item(i);
    //获取第二级节点的集合
    NodeList elemList1 =node.getElementsByTagName("childmostlyelem");
    for (int j=0; j < elemList1.getLength(); j++)
    {
        //获取这个节点的属性集合
        NamedNodeMap ac1 = elemList1.item(j).getAttributes();
        String key=ac1.getNamedItem("name").getNodeValue();
        NodeList
        nodeList=((Element)elemList1.item(j)).getElementsByTagName("value");
        String value=nodeList.item(0).getFirstChild().getNodeValue();
        hs1.put(key,value);
    }
    hs.put(ac.getNamedItem("id").getNodeValue(),hs1);
}
return hs;
}
}

```

做一个程序实验一下。

首先做一个关于交通工具的接口： Vehicle.java

```

package springdemo;

public interface Vehicle
{
    public String execute(String str);
    public String getMessage();
    public void setMessage(String str);
}

```

做一个实现类： Car.java

```

package springdemo;

public class Car implements Vehicle

```

```

{
    private String message="";
    private String x;

    public String getMessage()
    {
        return message;
    }
    public void setMessage(String str)
    {
        message = str;
    }

    public String execute(String str)
    {
        return getMessage() + str+"汽车在公路上开";
    }
}

```

Bean.xml 文档。

```

<beans>
<description>Spring Quick Start</description>
<bean id="Car"    class="springdemo.Car">
    <property name="message">
        <value>hello!</value>
    </property>
</bean>
</beans>

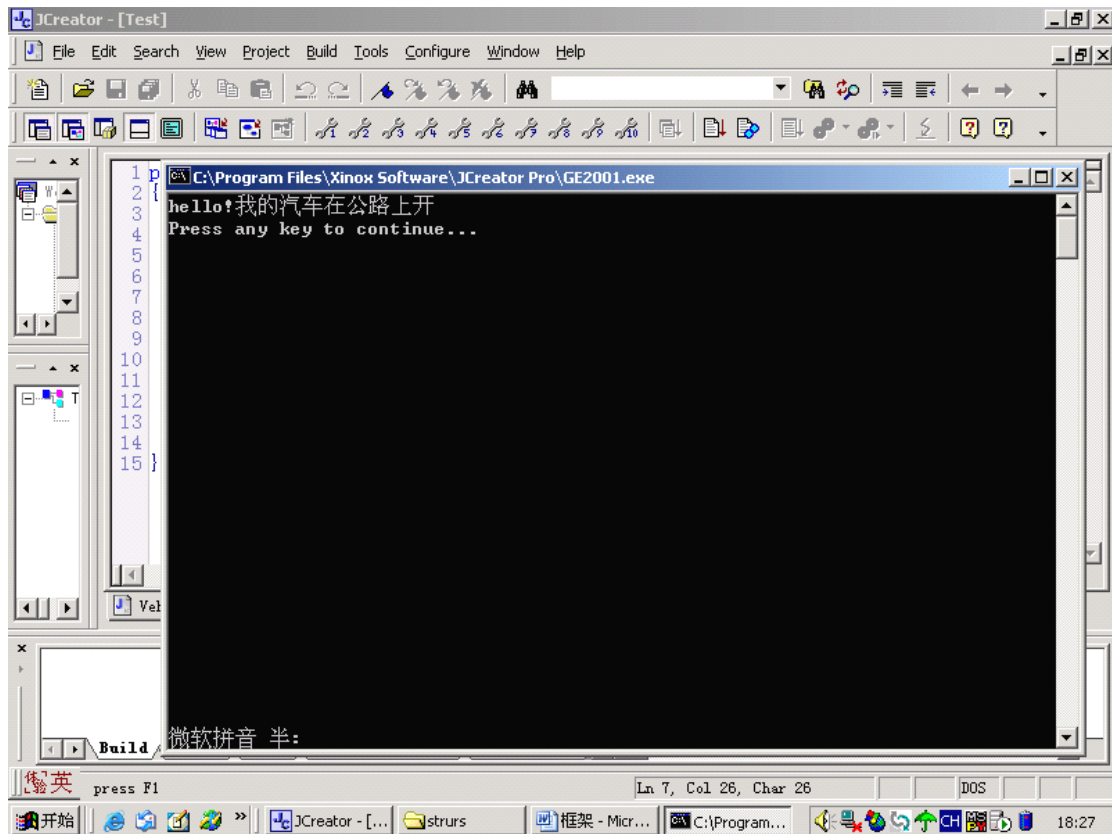
```

测试：Test.java

```

public class Test
{
    public static void main (String[] args) throws Exception
    {
        springdemo.ApplicationContext m=
            new springdemo.FileSystemXmlApplicationContext("Bean.xml");
        //实现类，使用标记 Car
        springdemo.Vehicle s1=(springdemo.Vehicle)m.getBean("Car");
        System.out.println(s1.execute("我的"));
    }
}

```



基于接口编程将使系统具备很好的扩充性。

再做一个类：Train.java

package springdemo;

```
public class Train implements Vehicle
{
    private String message="";

    public String getMessage()
    {
        return message;
    }
    public void setMessage(String str)
    {
        message = str;
    }

    public String execute(String str)
    {
        return getMessage() + str+"火车在铁路上走";
    }
}
```

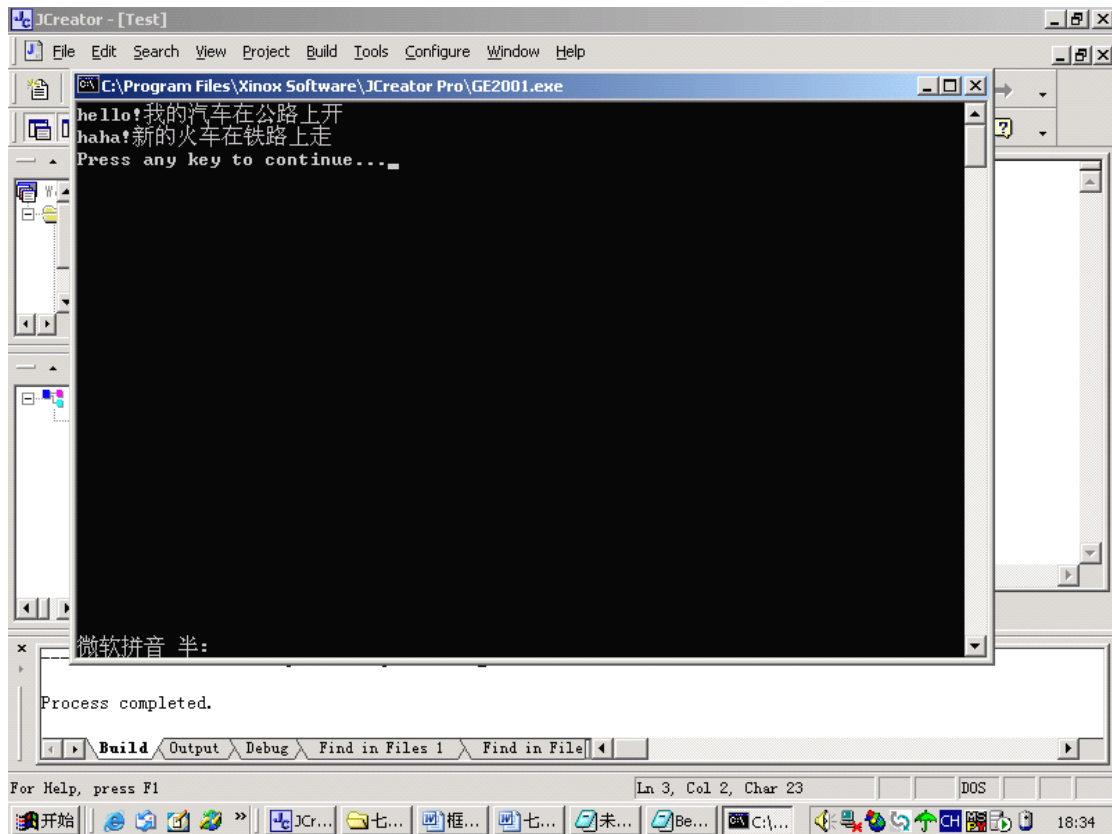
Bean.xml 改动如下。

```
<beans>
<description>Spring Quick Start</description>
<bean id="Car"    class="springdemo.Car">
    <property name="message">
        <value>hello!</value>
    </property>
</bean>
<bean id="Train"  class="springdemo.Train">
    <property name="message">
        <value>haha!</value>
    </property>
</bean>

</beans>
```

改动一下 Test.java

```
public class Test
{
    public static void main (String[] args) throws Exception
    {
        springdemo.ApplicationContext m=
            new springdemo.FileSystemXmlApplicationContext("Bean.xml");
        springdemo.Vehicle s1=(springdemo.Vehicle)m.getBean("Car");
        System.out.println(s1.execute("我的"));
        s1=(springdemo.Vehicle)m.getBean("Train");
        System.out.println(s1.execute("新的"));
    }
}
```

我们发现，在加入新的类的时候，使用方法几乎不变。

再做一组不同的接口和类来看一看：Idemo.java

```
package springdemo;
```

```
public interface IDemo
{
    public void setX(String x);
    public void setY(String y);
    public double Sum();
}
```

实现类：Demo.java

```
package springdemo;
```

```
public class Demo implements IDemo
{
    private String x;
    private String y;

    public void setX(String x)
    {
```

```

        this.x = x;
    }
    public void setY(String y)
    {
        this.y = y;
    }
    public double Sum()
    {
        return Double.parseDouble(x)+Double.parseDouble(y);
    }
}

```

Bean.xml 改动如下:

```

<beans>
<description>Spring Quick Start</description>

<bean id="Car"    class="springdemo.Car">
    <property name="message">
        <value>hello!</value>
    </property>
</bean>
<bean id="Train"  class="springdemo.Train">
    <property name="message">
        <value>haha!</value>
    </property>
</bean>
<bean id="demo"   class="springdemo.Demo">
    <property name="x">
        <value>20</value>
    </property>
    <property name="y">
        <value>30</value>
    </property>
</bean>

</beans>

```

改写 Test.java

```

public class Test
{
    public static void main (String[] args) throws Exception
    {

```

```

springdemo.ApplicationContext m=
    new springdemo.FileSystemXmlApplicationContext("Bean.xml");

springdemo.Vehicle s1=(springdemo.Vehicle)m.getBean("Car");

System.out.println(s1.execute("我的"));

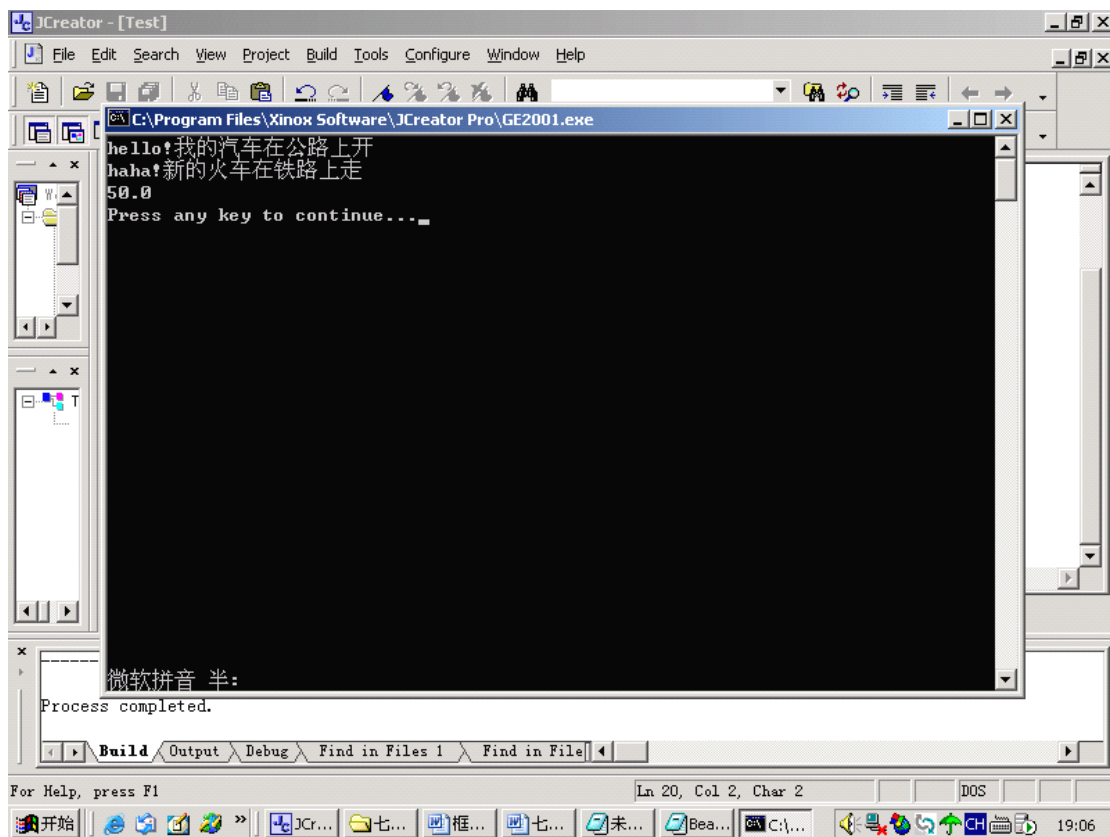
s1=(springdemo.Vehicle)m.getBean("Train");

System.out.println(s1.execute("新的"));

springdemo.IDemo s2=(springdemo.IDemo)m.getBean("demo");

System.out.println(s2.Sum());
}
}

```



二、创建者工厂的合理应用

实例：用.NET 数据提供者作为例子，讨论实现通用数据访问工厂的方法。

目的：

利用工厂实现与数据库无关的数据访问框架。

问题：

由于数据提供者是针对不同类型数据库的，造成升级和维护相当困难。现在需要把与数据库相关的部分独立出来，使应用程序不因数据库不同而有所变化。

解决方案：

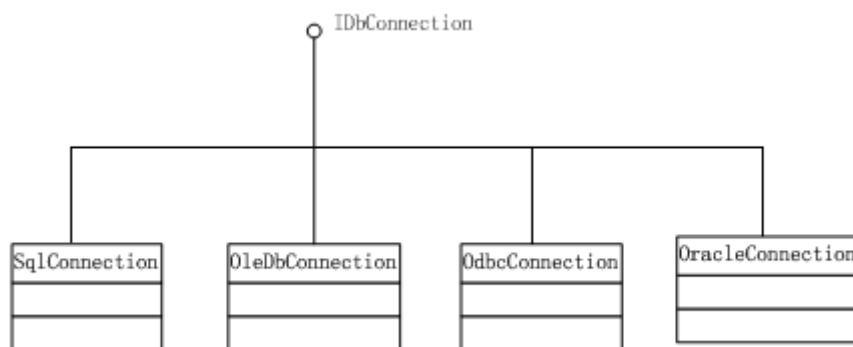
配置文件读取，不同提供者的集中处理，后期的可扩展性。

.NET 提供了一组连接对象。

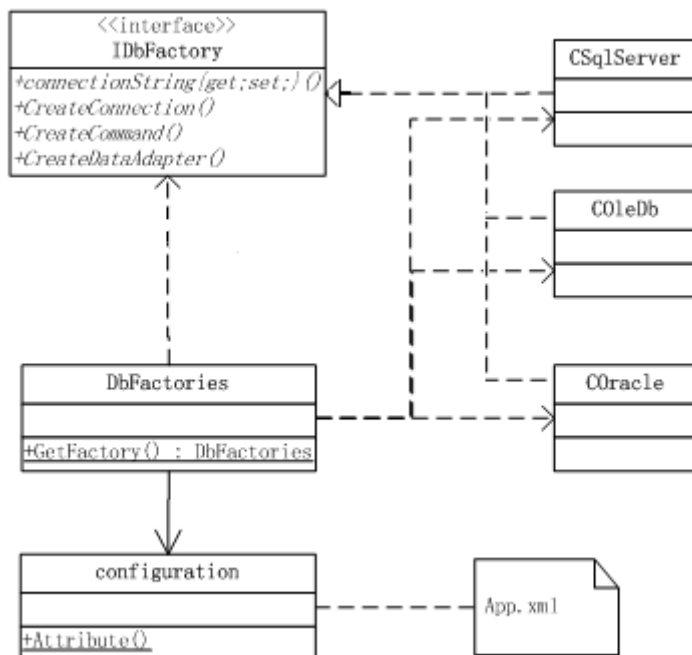
Connection 对象用于连接数据库，它被称之为提供者。

连接类对象的关系：

在.NET 2003，它的继承关系是这样的。



当数据提供者要求非常灵活的时候，尤其是对于领域中的层设计，需要考虑更多的问题。下面的例子，希望用一个配置文件来决定提供者的参数，而实现代码不希望由于数据库的变化而变化，而且当系统升级的时候，不应该有很大的困难。



配置文档：App.xml

```

<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <connectionStrings>
    <add name="PubsData" providerName="System.Data.SqlClient"
      connectionString="data source=zktb; initial catalog=奖金数据库;persist security info=false;
workstation id= COMMONOR-02A84C; packet size=4096;UID=sa;PWD=;Max Pool Size=50;"/>

    <add name="Nothing" providerName="System.Data.OleDb"
      connectionString="Provider=Microsoft.Jet.OLEDB.4.0;Data Source=c:\\奖金数据库.mdb"/>

    <add name="Bank" providerName="System.Data.SqlClient"
      connectionString="data source=zktb; initial catalog=Bank;persist security info=false;
workstation id= COMMONOR-02A84C; packet size=4096;UID=sa;PWD=;Max Pool Size=50;"/>

  </connectionStrings>
  <appSettings>
    <add key="provider" value="SqlClient"/>
  </appSettings>
</configuration>

```

类代码:

```

using System.Data;
using System.Data.Common;
using System.Data.OleDb;
using System.Data.SqlClient;
using System.Data.OracleClient;

namespace DbComm
{
    //工厂接口
    public interface IDbFactory
    {
        string connectionString
        {
            get;
            set;
        }
        IDbConnection CreateConnection();
        IDbCommand CreateCommand();
        DbDataAdapter CreateDataAdapter(IDbCommand comm);
    }

    //工厂类, 工厂方法模式

```

```

public class DbFactories
{
    public static IDbFactory GetFactory(string name)
    {
        string providerName=null;
        string connectionString=null;

        ArrayList
al=configuration.Attribute("c:\\App.xml","connectionStrings","add");

        for (int i=0;i<al.Count;i++)
        {
            Hashtable hs=(Hashtable)al[i];
            if (hs["name"].ToString().Equals(name))
            {
                providerName=hs["providerName"].ToString();
                connectionString=hs["connectionString"].ToString();
                break;
            }
        }

        IDbFactory da=null;

        switch (providerName)
        {
            case "System.Data.SqlClient":
                da=new CSqlServer();
                break;
            case "System.Data.OleDb":
                da=new COleDb();
                break;
            case "System.Data.Oracle":
                da=new COracle();
                break;
        }
        da.connectionString=connectionString;
        return da;
    }
}

//这是一个专门用于读配置文件的类
public class configuration
{
    public static ArrayList Attribute(string configname,

```

```

        string mostlyelem,
        string childmostlyelem)
    {
        ArrayList al=new ArrayList();

        XmlDocument doc = new XmlDocument();
        doc.Load(configname);

        //建立所有元素的列表
        XmlElement root = doc.DocumentElement;

        //把所有的主要标记都找出来放到节点列表中
        XmlNodeList elemList = root.GetElementsByTagName(mostlyelem);

        for (int i=0; i < elemList.Count; i++)
        {
            //获取二级标记子节点
            XmlNodeList elemList1 =
            ((XmlElement)elemList[i]).GetElementsByTagName(childmostlyelem);
            //获取这个节点的数目
            int N=elemList1.Count;
            for (int j=0; j < elemList1.Count; j++)
            {
                Hashtable hs=new Hashtable();
                //获取这个节点的属性集合
                XmlAttributeCollection ac = elemList1[j].Attributes;
                for( int k = 0; k < ac.Count; k++ )
                {
                    hs.Add(ac[k].Name, ac[k].Value);
                }
                al.Add(hs);
            }
        }
        return al;
    }
}

//SqlServer实现类
public class CSqlServer:IDbFactory
{
    private string strConn;

    public string connectionString
    {
        get
    }

```

```

        {
            return this.strConn;
        }
        set
        {
            this.strConn=value;
        }
    }
    public IDbConnection CreateConnection()
    {
        SqlConnection conn=new SqlConnection(strConn);
        return conn;
    }
    public IDbCommand CreateCommand()
    {
        return new SqlCommand();
    }
    public DbDataAdapter CreateDataAdapter(IDbCommand comm)
    {
        SqlDataAdapter adp=new SqlDataAdapter();
        adp.SelectCommand=(SqlCommand)comm;
        SqlCommandBuilder cb=new SqlCommandBuilder(adp);
        return adp;
    }
}
//OleDb实现类
public class OleDb:IDbFactory
{
    private string strConn;

    public string connectionString
    {
        get
        {
            return this.strConn;
        }
        set
        {
            this.strConn=value;
        }
    }

    public IDbConnection CreateConnection()
    {

```

```
        OleDbConnection conn=new OleDbConnection(strConn);
        return conn;
    }
    public IDbCommand CreateCommand()
    {
        return new OleDbCommand();
    }
    public DbDataAdapter CreateDataAdapter(IDbCommand comm)
    {
        OleDbDataAdapter adp=new OleDbDataAdapter();
        adp.SelectCommand=(OleDbCommand)comm;
        OleDbCommandBuilder cb=new OleDbCommandBuilder(adp);
        return adp;
    }
}
//Oracle实现类
public class COracle:IDbFactory
{
    private string strConn;

    public string connectionString
    {
        get
        {
            return this.strConn;
        }
        set
        {
            this.strConn=value;
        }
    }

    public IDbConnection CreateConnection()
    {
        OracleConnection conn=new OracleConnection(strConn);
        return conn;
    }
    public IDbCommand CreateCommand()
    {
        return new OracleCommand();
    }
    public DbDataAdapter CreateDataAdapter(IDbCommand comm)
    {
        OracleDataAdapter adp=new OracleDataAdapter();
```

```

        adp.SelectCommand=(OracleCommand) comm;
        OracleCommandBuilder cb=new OracleCommandBuilder(adp);
        return adp;
    }
}
}

```

应用:

```

System.Data.Common.DbDataAdapter adp;
DataTable dt=new DataTable();

//处理数据
DataTable callData(string name,string query)
{
    DataTable dt=new DataTable();
    try
    {
        DbComm.IDbFactory df=DbComm.DbFactories.GetFactory(name);
        IDbConnection conn=df.CreateConnection();
        IDbCommand comm=df.CreateCommand();
        comm.CommandText=query;
        comm.Connection=conn;
        adp=df.CreateDataAdapter(comm);
        adp.Fill(dt);
    }
    catch
    {
        MessageBox.Show("错误,可能配置文件不对");
    }
    return dt;
}

//调SqlServer
private void button1_Click(object sender, System.EventArgs e)
{
    dt=callData("PubsData","select * from 奖金");
    dataGrid1.DataSource=dt;
}

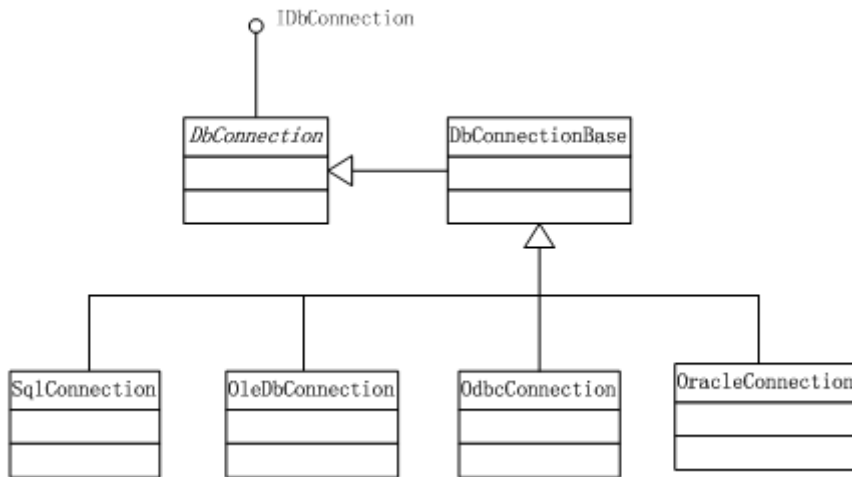
//调OleDb
private void button2_Click(object sender, System.EventArgs e)
{
    dt=callData("Nothing","select * from 奖金");
    dataGrid1.DataSource=dt;
}

//提交

```

```
private void button3_Click(object sender, System.EventArgs e)
{
    adp.Update(dt);
}
```

.NET 2005 为了扩充功能，在接口和实现类之间又加了一个抽象类（DbConnection 等）。



为了更好的实现上面类似的性能，提供了一套基于工厂的对象：

System.Data.Common.DbProviderFactory
System.Data.Common.DbConnectionStringBuilder
System.Data.Common.DbProviderFactories

等一系列的类，我们看一下在.NET 2005 上实现的几个例子，有了上面的基础，这几个类的理解当不会感到困难。

配置文档 App.config 和上面的例子几乎是相同的。

等一系列的类，我们看一下在.NET 2005 上实现的几个例子，有了上面的基础，这几个类的理解当不会感到困难。

配置文档 App.config
 和上面的例子是一模一样的。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <connectionStrings>
    <clear/>
    <add name="PubsData" providerName="System.Data.SqlClient"
        connectionString="data source=COMMONOR-02A84C; initial catalog=奖金数据库;persist
security info=false; workstation id= COMMONOR-02A84C; packet size=4096;UID=sa;PWD=;Max Pool
```

```

Size=50;"/>

<add name="Nothing" providerName="System.Data.OleDb"
    connectionString="Provider=Microsoft.Jet.OLEDB.4.0;Data Source=c:\\虚拟公司.mdb"/>

<add name="Bank" providerName="System.Data.SqlClient"
    connectionString="data source=COMMONOR-02A84C; initial catalog=Bank;persist security
info=false; workstation id= COMMONOR-02A84C; packet size=4096;UID=sa;PWD=;Max Pool Size=50;"/>

</connectionStrings>
<appSettings>
    <add key="provider" value="SqlClient"/>
</appSettings>
</configuration>

```

实现代码:

```

private void button4_Click(object sender, EventArgs e)
{
    DataTable dt = null;
    //返回一个 DataTable,
    //其中包含有关实现 DbProviderFactory 的所有已安装提供程序的信
    dt = System.Data.Common.DbProviderFactories.GetFactoryClasses();
    dataGridView1.DataSource = dt;
}

//表示一组方法, 这些方法用于创建提供程序对数据源类的实现的实例。
//命名空间: System.Data.Common
//程序集: System.Data (在 system.data.dll 中)
System.Data.Common.DbProviderFactory factory =
System.Data.SqlClient.SqlClientFactory.Instance;

public System.Data.Common.DbConnection GetProConn()
{
    System.Data.Common.DbConnection conn = factory.CreateConnection();
    System.Configuration.ConnectionStringSettings
        publ = System.Configuration.ConfigurationManager.ConnectionStrings["PubsData"];
    conn.ConnectionString = publ.ConnectionString;
    return conn;
}

private void button5_Click(object sender, EventArgs e)
{

```

```

        System.Data.Common.DbCommand comm = factory.CreateCommand();
        comm.Connection = GetProConn();
        comm.CommandText = "select * from 奖金";
        comm.Connection.Open();
        DataTable dt = new DataTable();
        dt.Load(comm.ExecuteReader());
        comm.Connection.Close();
        dataGridView1.DataSource = dt;
    }

    //与提供者无关的数据访问
    private void SeeData(string protext, string tablename)
    {
        System.Configuration.ConnectionStringSettings
            publ = System.Configuration.ConfigurationManager.ConnectionStrings[protext];
        System.Data.Common.DbProviderFactory factory =
            System.Data.Common.DbProviderFactories.GetFactory(publ.ProviderName);
        System.Data.Common.DbConnectionStringBuilder bld =
            factory.CreateConnectionStringBuilder();
        bld.ConnectionString = publ.ConnectionString;
        System.Data.Common.DbConnection cn = factory.CreateConnection();
        cn.ConnectionString = bld.ConnectionString;
        System.Data.Common.DbDataAdapter da = factory.CreateDataAdapter();
        System.Data.Common.DbCommand cmd = factory.CreateCommand();
        cmd.CommandText = "select * from " + tablename;
        cmd.CommandType = CommandType.Text;
        cmd.Connection = cn;
        da.SelectCommand = cmd;
        System.Data.Common.DbCommandBuilder cb = factory.CreateCommandBuilder();
        cb.DataAdapter = da;
        DataSet ds = new DataSet();
        da.Fill(ds, "auth");
        dataGridView1.DataSource = ds;
        dataGridView1.DataMember = "auth";
    }

    private void button6_Click(object sender, EventArgs e)
    {
        SeeData("PubsData", "奖金");
    }

    private void button7_Click(object sender, EventArgs e)
    {

```

```

        SeeData("Nothing", "Orders");
    }

```

利用上面的讨论过的原理，还可以把代码进一步封装，形成一个基于领域的层，再给予接口编程的原则下，系统的升级性能是非常好的。

三、单件模式的应用问题

有时候，我们需要一个全局唯一的连接对象，这个对象可以管理多个通信会话，在使用这个对象的时候，不关心它是否实例化及其内部如何调度，这种情况很多，例如串口通信和数据库访问对象等等，这些情况都可以采用单件模式。

1、意图

单件模式保证应用只有一个全局唯一的实例，并且提供一个访问它的全局访问点。

2、使用场合

当类只能有一个实例存在，并且可以在全局访问的时候，这个唯一的实例应该可以通过子类实现扩展，而且用户无需更改代码即可以使用。

3、结构

单件模式的结构非常简单，包括防止其它对象创建实例的私有构造函数，保持唯一实例的私有变量和全局变量访问接口等，请看下面的例子：

```

using System;

namespace 单件模式
{
    public class CShapeSingletion
    {
        private static CShapeSingletion mySingletion=null;
        //为了防止用户实例化对象，这里把构造函数设为私有的
        private CShapeSingletion()
        {}
        //这个方法是调用的入口
        public static CShapeSingletion Instance()
        {
            if (mySingletion==null)
            {
                mySingletion=new CShapeSingletion();
            }
            return mySingletion;
        }
        private int intCount=0;
        //计数器，虽然是实例方法，但这里的表现类同静态
        public int Count()
        {
            intCount+=1;
        }
    }
}

```

```

        return intCount;
    }
}

private void button1_Click(object sender, System.EventArgs e)
{
    label11.Text=CShapeSingleton.Instance().Count().ToString();
}

```

4、效果

单件提供了全局唯一的访问入口，因此比较容易控制可能发生的冲突。

单件是对静态函数的一种改进，首先避免了全局变量对系统的污染，其次它可以有子类，业可以定义虚函数，因此它具有多态性，而类中的静态方法是不能定义成虚函数的。

单件模式也可以定义成多件，即允许有多个受控的实例存在。

单件模式维护了自身的实例化，在使用的时候是安全的，一个全局对象无法避免创建多个实例，系统资源会被大量占用，更糟糕的是会出现逻辑问题，当访问象串口这样的资源的时候，会发生冲突。

5、单件与实用类中的静态方法

实用类提供了系统公用的静态方法，并且也经常采用私有的构造函数，和单件不同，它没有实例，其中的方法都是静态方法。

实用类和单件的区别如下：

- 1) 实用类不保留状态，仅提供功能。
- 2) 实用类不提供多态性，而单件可以有子类。
- 3) 单件是对象，而实用类只是方法的集合。

应该说在实际应用中，实用类的应用更加广泛，但是在涉及对象的情况下需要使用单件，例如，能不能用实用类代替抽象工厂呢？如果用传统的方式显然不行，因为实用类没有多态性，会导致每个工厂的接口不同，在这个情况下，必须把工厂对象作为单件。

因此何时使用单件，要具体情况具体分析，不能一概而论。

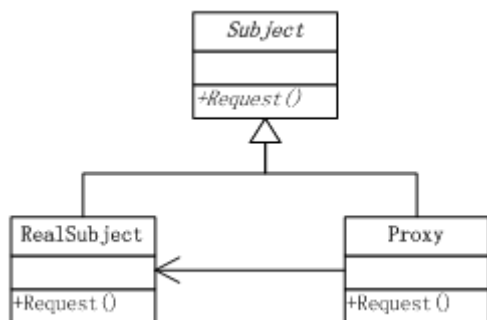
第六节 在团队并行开发中应用代理模式

代理模式的意图，是为其它对象提供一个代理，以控制对这个对象的访问。

首先作为代理对象必须与被代理对象有相同的接口，换句话说，用户不能因为使不使用代理而做改变。

其次，需要通过代理控制对对象的访问，这时，对于不需要代理的客户，被代理对象应该是不透明的，否则谈不上代理。

下图是代理模式的结构。

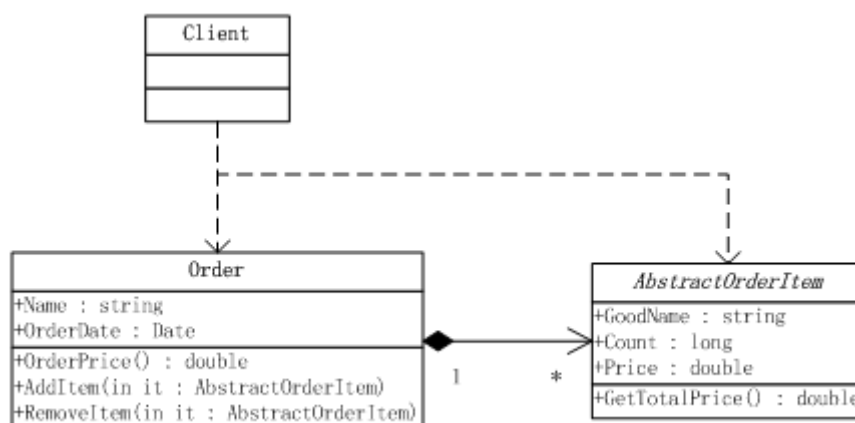


实例：测试中的“占位”对象

软件开发需要协同工作，希望开发进度能够得到保证，为此需要合理划分软件，每个成员完成自己的模块，为同伴留下相应的接口。

在开发过程中，需要不断的测试。然而，由于软件模块之间需要相互调用，对某一模块的测试，又需要其它模块的配合。而且在模块开发过程中也不可能完全同步，从而给测试带来了问题。

假定，有一个系统，其中 Ordre（订单）和 OrderItme（订单项）的 UML 图如下。



其中：Ordre 包括若干 OrderItme，订单的总价是每个订单项之和。

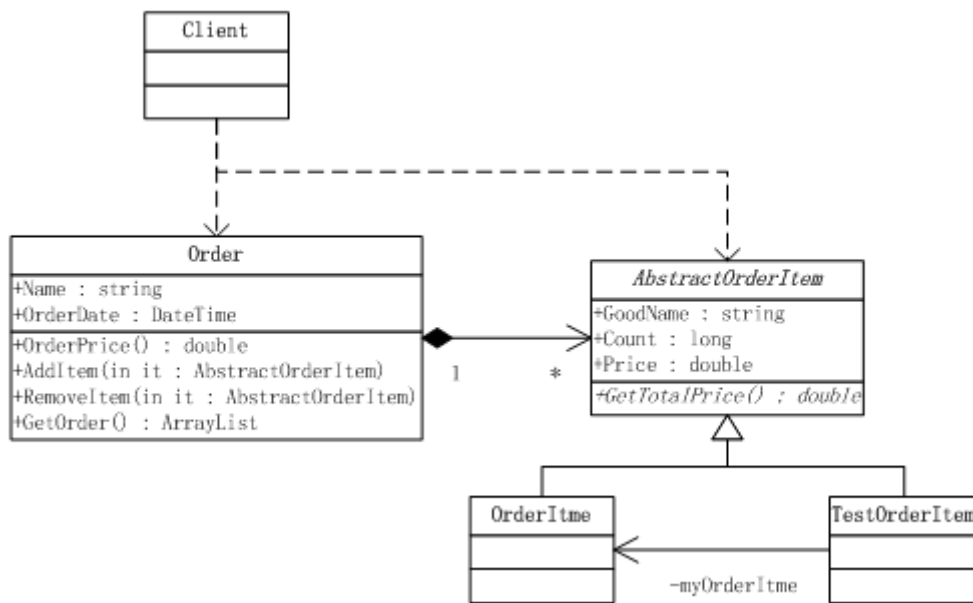
假定这是由两个开发组完成的，如果 OrderItme 没有完成，Ordre 也就没有办法测试。一个简单的办法，是 Ordre 开发的时候屏蔽 OrderItme 调用，但这样代码完成的时候需要做大量的垃圾清理工作，显然这是不合适的，我们的问题是，如何把测试代码和实际代码分开，这样更便于测试，而且可以很好的集成。

如果我们把 OrderItem 抽象为一个接口或一个抽象类，实现部分有两个平行的子类，一个是真正的 OrderItem，另一个是供测试用的 TestOrderItem，在这个类中编写测试代码，我们称之为 Mock。

这时，Order 可以使用 TestOrderItem，测试。当 OrderItem 完成以后，有需要使用 OrderItem 进行集成测试，如果 OrderItem 还要修改，又需要转回 TestOrderItem。

我们希望只用一个参数就可以完成这种切换，比如在配置文件中，测试设为 true，而正常使用为 false。

这些需求牵涉到代理模式的应用，现在可以把代理结构画清楚。



这就很好的解决了问题。

实例：

首先编写一个配置文件：config.xml

```

<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="istest" value="false" />
  </appSettings>
</configuration>

```

代码：

```

using System;
using System.IO;
using System.Xml;
using System.Collections;

```

namespace 代理一

```

{
    //这是统一的接口
    public abstract class AbstractOrderItem
    {
        private string m_GoodName;
        private long m_Count;
        private double m_Price;
        public virtual string GoodName
        {

```

```
        get
        {
            return m_GoodName;
        }
        set
        {
            m_GoodName=value;
        }
    }
    public virtual long Count
    {
        get
        {
            return m_Count;
        }
        set
        {
            m_Count=value;
        }
    }
    public virtual double Price
    {
        get
        {
            return m_Price;
        }
        set
        {
            m_Price=value;
        }
    }
    //价格求和，这个计算方式是另外的人编写的
    public abstract double GetTotalPrice();
}
//处理订单的代码
public class Order
{
    public string Name;
    public DateTime OrderDate;
    private ArrayList oitems;

    public Order()
    {
        oitems=new System.Collections.ArrayList();
    }
}
```

```

    }
    public void AddItem(AbstractOrderItem it)
    {
        oitems.Add(it);
    }
    public void RemoveItem(AbstractOrderItem it)
    {
        oitems.Remove(it);
    }
    public double OrderPrice()
    {
        AbstractOrderItem it;
        double op=0;
        for (int i=0;i<oitems.Count;i++)
        {
            it=(AbstractOrderItem)oitems[i];
            op+=it.GetTotalPrice();
        }
        return op;
    }
    public ArrayList GetOrder()
    {
        return oitems;
    }
}
//由另外的队伍编写的处理代码
//主要需要调用客户服务的计算方法，这里只处理合计
public class OrderItem:AbstractOrderItem
{
    public override double GetTotalPrice()
    {
        return this.Count*this.Price;
    }
}
//这是一个专门用于读配置文件的类
public class configuration
{
    public static string[] Attribute(string configname,
        string mostlyelem,
        string childmostlyelem)
    {
        ArrayList al=new ArrayList();

        XmlDocument doc = new XmlDocument();

```

```

doc.Load(configname);

//建立所有元素的列表
XmlElement root = doc.DocumentElement;

//把所有的主要标记都找出来放到节点列表中
XmlNodeList elemList = root.GetElementsByTagName(mostlyelem);

for (int i=0; i < elemList.Count; i++)
{
    //获取二级标记子节点
    XmlNodeList elemList1 =
((XmlElement)elemList[i]).GetElementsByTagName(childmostlyelem);
    //获取这个节点的数目
    int N=elemList1.Count;
    for (int j=0; j < elemList1.Count; j++)
    {
        //获取这个节点的属性集合
        XmlAttributeCollection ac = elemList1[j].Attributes;
        for( int k = 0; k < ac.Count; k++ )
        {
            if (ac[k].Name=="value")
            {
                al.Add(ac[k].Value);
            }
        }
    }

    string[] strOut=new string[al.Count];
    al.CopyTo(strOut,0);
    return strOut;
}
}

//这是一个代理类，由配置文件决定状态
public class TestOrderItem:AbstractOrderItem
{
    private OrderItme myOrderItme=null;

    public override double GetTotalPrice()
    {
        //读配置文件，看是不是处于测试状态
        string[] istest=configuration.Attribute("config.xml","appSettings","add");

        bool s=bool.Parse(istest[0]);
    }
}

```

```

        if (s)
        {
            //这个返回的数据称之为“占位”
            return 1000;
        }
        else
        {
            myOrderItme=new OrderItme();
            myOrderItme.GoodName=this.GoodName;
            myOrderItme.Count=this.Count;
            myOrderItme.Price=this.Price;
            return myOrderItme.GetTotalPrice();
        }
    }
}

Order o=new Order();
//加入
private void button1_Click(object sender, System.EventArgs e)
{
    TestOrderItem t1=new TestOrderItem();
    t1.GoodName=textBox1.Text;
    t1.Count=long.Parse(textBox2.Text);
    t1.Price=double.Parse(textBox3.Text);
    o.AddItem(t1);
}
//显示
private void button2_Click(object sender, System.EventArgs e)
{
    listBox1.Items.Clear();
    ArrayList m=o.GetOrder();
    for (int i=0;i<m.Count;i++)
    {
        AbstractOrderItem s=(AbstractOrderItem)m[i];
        listBox1.Items.Add(s.GoodName+" "+s.Count.ToString()+"
"+s.Price.ToString());
    }
    listBox1.Items.Add("合计: "+o.OrderPrice());
}
}

```

第七节 利用观察者模式延长架构的生命周期

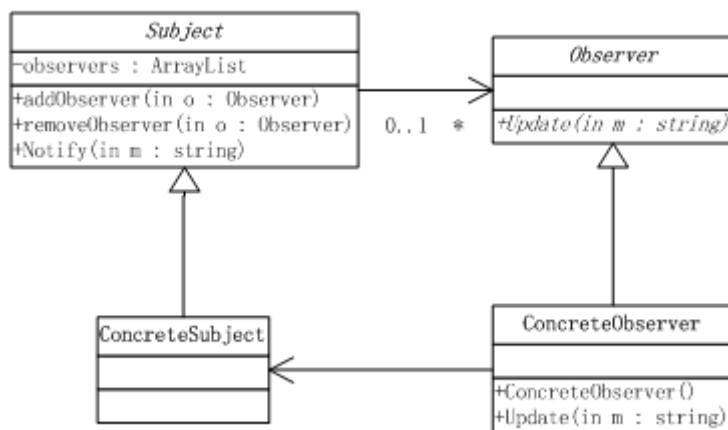
当需要上层对底层的操作的时候，可以使用观察者模式实现向上协作。也就是上层响应底层的事件，但这个事件的执行代码由上层提供。

1、意图：

定义对象一对多的依赖关系，当一个对象发生变化时候，所有依赖它的对象都得到通知并且被自动更新。

2、结构

传统的观察者模式结构如下。



3、举例：

```

public class Payment
{

    private decimal amount;

    public decimal Amount
    {
        get
        {
            return amount;
        }
        set
        {
            amount = value;
        }
    }

    public delegate string PersonAction(string x);
}

```

```

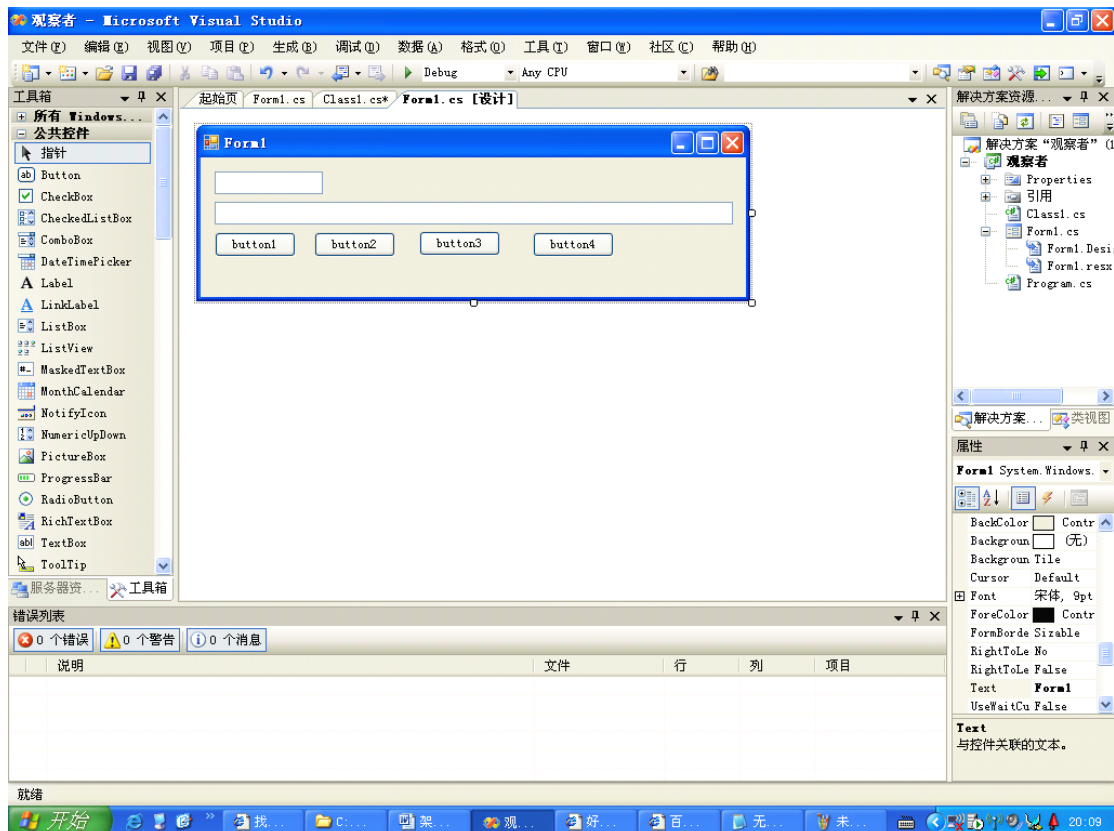
//定义事件
public event PersonAction Action;

protected virtual string onAction(string x1)
{
    if (Action != null)
        return Action(x1);
    else
        return x1;
}

public virtual string goSale()
{
    string x = "不变的流程一 ";
    x = onAction(x);    //可变的流
    x += amount + ", 正在查询库存状态"; //属性和不变的流程二
    return x;
}
}

```

调用:



```
private Payment o1 = new Payment();
```

```
private Payment o2 = new Payment();
private Payment o3 = new Payment();
private Payment o4 = new Payment();

private void Form1_Load(object sender, EventArgs e)
{
    o2.Action += new Payment.PersonAction(Cash);
    o3.Action += new Payment.PersonAction(Credit);
    o4.Action += new Payment.PersonAction(Check);
}

private string Cash(string x)
{
    return x + "现金支付 ";
}

private string Credit(string x)
{
    return x + "信用卡支付, 联系机构 ";
}

private string Check(string x)
{
    return x + "支票支付, 联系财务部 ";
}

//没有事件
private void button1_Click(object sender, EventArgs e)
{
    if (textBox1.Text != "")
    {
        o1.Amount = decimal.Parse(textBox1.Text);
    }
    textBox2.Text = o1.goSale();
}

//现金支付
private void button2_Click(object sender, EventArgs e)
{
    if (textBox1.Text != "")
    {
        o2.Amount = decimal.Parse(textBox1.Text);
    }
    textBox2.Text = o2.goSale();
}
```



```
//信用卡支付
private void button3_Click(object sender, EventArgs e)
{
    if (textBox1.Text != "")
    {
        o3.Amount = decimal.Parse(textBox1.Text);
    }
    textBox2.Text = o3.goSale();
}

//支票支付
private void button4_Click(object sender, EventArgs e)
{
    if (textBox1.Text != "")
    {
        o4.Amount = decimal.Parse(textBox1.Text);
    }
    textBox2.Text = o4.goSale();
}
```

第八节 树状结构和链形结构的对象组织

对象的组织方式，可以是树状结构和链形结构两种。

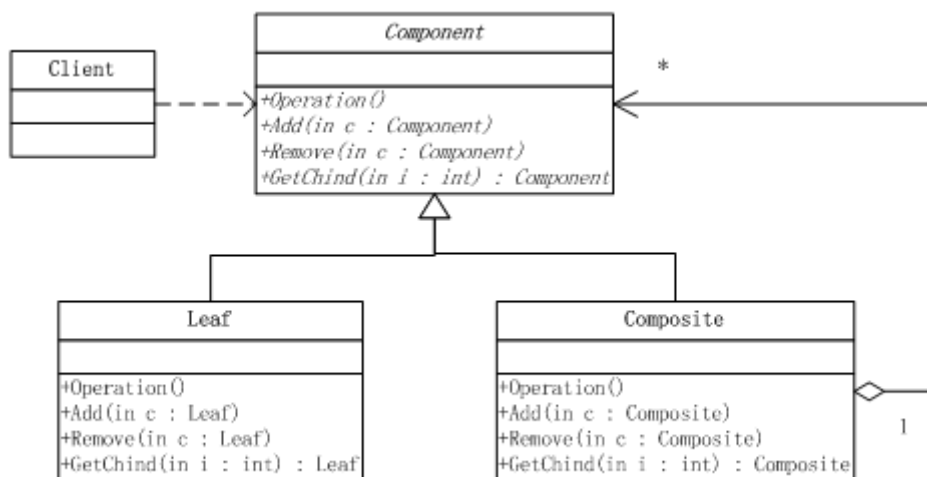
一、树状结构：组合模式

组合可以说是非常常见的一种结构，我们经常会遇到一些装配关系，从数据结构上来说，这些关系往往表达为一种树状结构，这就用到了组合模式。

它的意图是，把对象组合成树形结构来表示“部分-整体”关系，使得用户对单个对象和组合对象的使用具有一致性。

1、结构

组合模式的结构可以有多种形式，一个最典型的结构如下。



2、效果

使用组合模式有以下优点：

1) 组合对象可以由基本对象和其它组合对象构成，这样，采用有限的基本对象就可以构造数量众多的组合对象。

2) 组合对象和基本对象有相同的接口，这样操作组合对象和操作基本对象基本相同，这样就可以大大简化客户代码。

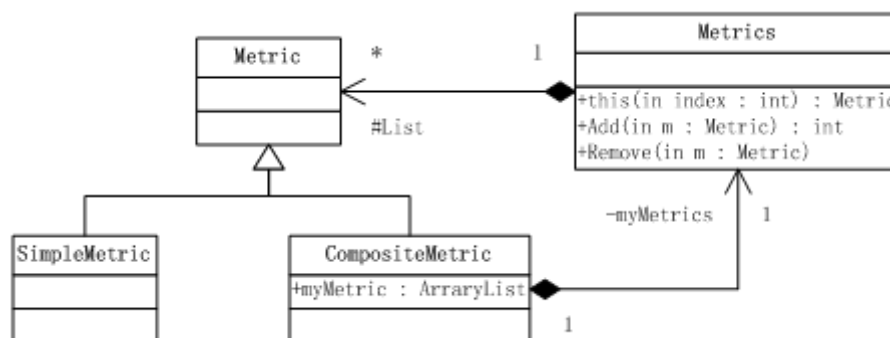
3) 可以很容易的增加类型，由于新类型符合相同的接口，因此不需要改动客户代码。

采用组合方式的代价是，由于组合对象和基本对象的接口相同，所以程序不能依赖具体的类型，不过这个问题本身并不大。

3、组合模式的不同实现方式

组合模式可以有多种实现方式，下面列出三种。

1) 强制类型集合



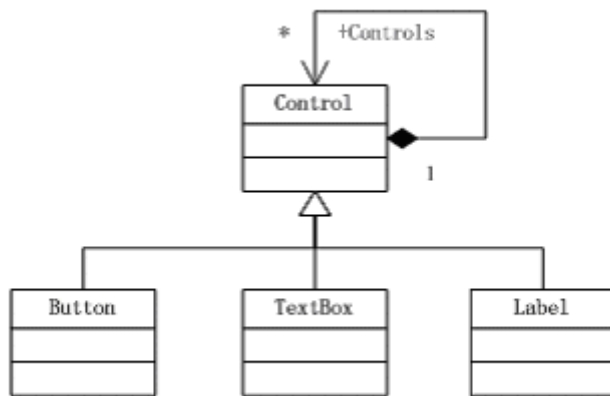
这里自定义一个表示控件聚合的 **Metrics** 对象，由这个对象放置类型（比如 **Metric**），采用强制类型集合的优点为：

代码含义清楚：集合中的类型是确定的；

不容易出错：由于集合中的类型是确定的，所以有类型错误在编译的时候可以早期发现。

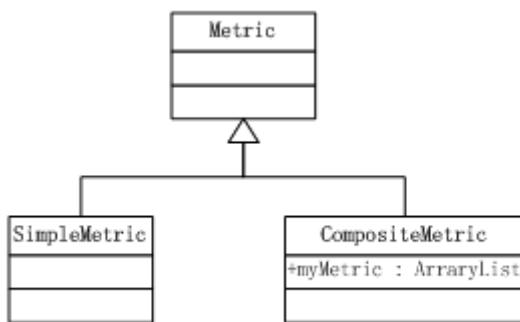
这种方式的缺点是需要自行定制集合，编码比较复杂。

2) 基础节点和复合节点相同



这种方式集合就在基础节点中，便于构造复杂的数据结构。

3) 非强制类型集合



非强制类型集合主要是采用象 ArrayList 这类集合，它的数据类型是 Object，因此可以保留任何数据类型，由于 .NET 中具备大量的可供选择的集合类，编码比较方便。

缺点是：

代码不够清晰，特别是 ArrayList 内部保留的数据结构往往看得不清楚。

需要进行类型转换，这样有些问题只有在运行中才会暴露出来，增加了调试的难度。

另外一个问题，组合模式往往需要遍历数据，这需要使用递归方法。

二、链形结构：职责链模式

当算法牵涉到一种链型运算，而且不希望处理过程中有过多的循环和条件选择语句，并且希望比较容易的扩充文法，可以采用职责链模式。

1、意图

使多个对象都有机会处理请求，避免请求的发送者和接收者之间的耦合关系，可以把这些对象链成一个链，并且沿着这个链来传递请求，直到处理完为止。

当处理方式需要动态宽展的时候，职责链是一个很好的方式。

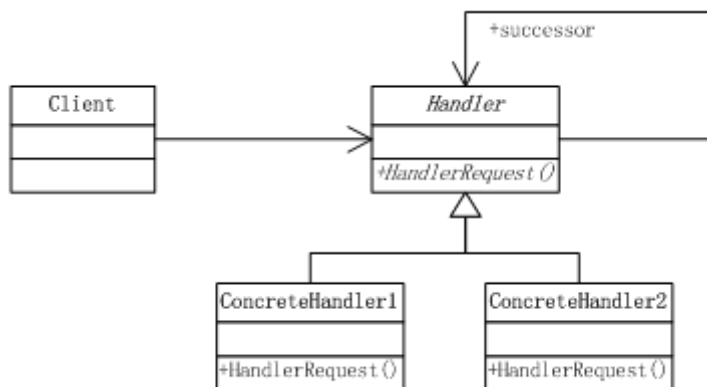
2、使用场合

以下情况可以使用职责链模式：

- 1) 有多个对象处理请求，到底怎么处理在运行时确定。
- 2) 希望在不明确指定接收者的情况下，向多个对象中的一个提交请求。
- 3) 可处理一个请求的对象集合应该被动态指定。

3、结构

职责链的结构如下。



其中：

Handler 处理者

方法：HandlerRequest 处理请求

ConcreteHandler 具体处理者

关联变量：successor （后续）是组成链所必需。

4、实例

下面的实例反映了上面职责链的工作过程，注意链是在运行中建立的。

```

using System;
using System.Windows.Forms;

namespace 基本职责链
{
    public abstract class Handler
    {
        public Handler successor;
        public int s;
        public abstract int HandlerRequest(int k);
    }
    public class ConcreteHandler1:Handler
    {

```

```

        public override int HandlerRequest(int k)
        {
            int c=k+s;
            return c;
        }
    }
}

```

调用

```

Handler m;

private void Form1_Load(object sender, System.EventArgs e)
{
    //建立职责链
    Handler ct=new ConcreteHandler1();
    Handler ct1=null;
    ct.s=0;
    m=ct;
    ct1=m;
    for (int i=1;i<10;i++)
    {
        ct=new ConcreteHandler1();
        ct.s=i;
        ct1.successor=ct;
        ct1=ct;
    }
}

private void button1_Click(object sender, System.EventArgs e)
{
    //显示
    see(m, 10);
}

void see(Handler m, int b)
{
    if (m !=null)
    {
        int s=m.HandlerRequest(b);
        listBox1.Items.Add(s);
        m=m.successor;
        see(m, s);
    }
}
}

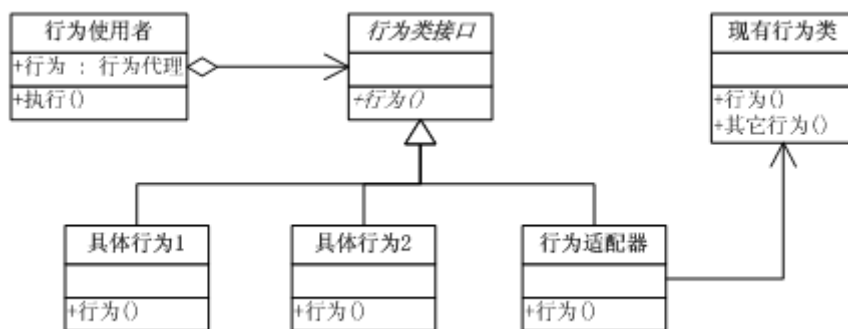
```

第九节 委托技术与行为型设计模式

一、委托技术的使用场合

上面关于行为模式的讨论，关键是行为方法的多态性实现，行为使用者引用行为类接口的目的，仅仅是为了获得行为方法，这样，为了使用行为，使用者必须依赖行为类接口。尽管这样耦合性已经很小了，但有些情况下仍然不很方便。

假如，现有一个类实现了行为的抽象方法，但没有实现行为类接口，我们要使用它的“行为”方法，这时，就必须引入一个行为类适配器，从而使系统的复杂性增加了，下面就是这样一种结构。



代码：

```

public class 现有行为类
{
    public void 行为()
    {
    }
    public void 其它行为()
    {
    }
}

public class 行为适配器:行为类接口
{
    public override void 行为()
    {
        现有行为类 m=new 现有行为类();
        m.行为();
    }
}

public class 行为使用者
{
    public 行为类接口 我的行为;
}
  
```

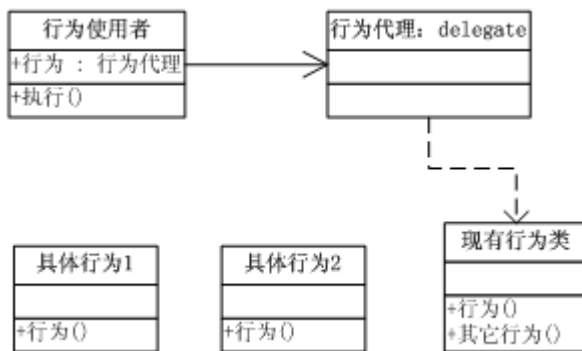
```

public 行为使用者()
{
    我的行为=new 行为适配器();
}
public void 执行()
{
    我的行为.行为();
}
}

```

这会使问题变得复杂。

如果行为类接口的引入单存是为了行为扩充，我们可以用委托来代替行为类接口，也就是这些继承关系都不需要存在，而是用委托来定义行为的格式。



```

public class 具体行为1
{
    public void 行为()
    {
    }
}
public class 具体行为2
{
    public void 行为()
    {
    }
}
public class 现有行为类
{
    public void 行为()
    {
    }
    public void 其它行为()
    {
    }
}

```

```

    }
}
public delegate void 行为代理();
public class 行为使用者
{
    public 行为代理 行为代理行为;
    public 行为使用者()
    {
        行为代理行为=new 行为代理((new 现有行为类()).行为);
    }
    public void 执行()
    {
        行为代理行为();
    }
}

```

同样可以实现上面的目的，但系统的耦合度大幅度的降下来了。在只关心方法的场合，由“行为代理”来决定方法的格式，这时各个类并不需要一致的接口。

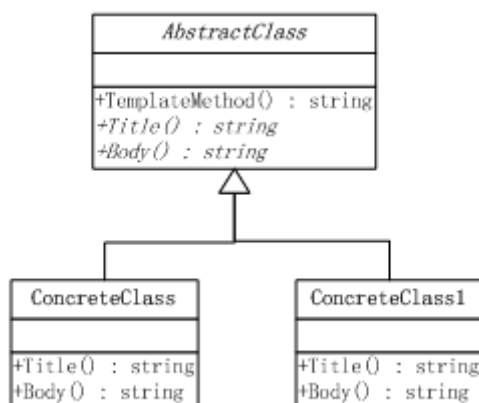
采用委托技术一个现实的缺点是，现有的 UML 标准并不包括委托结构，因此在静态图上并不能很好的表达出来，同时，本身系统的结构关系也不太清楚，这给调试和测试带来了难度。

当然，面向对象的理论是发展的，传统的面向对象理论多态性主要是通过继承或者是通过接口实现的，但是，委托同样也是实现多态性的一种方式（同样，还可以通过反射实现多态性），这也表明了传统的理论并不是一成不变的道理。

二、采用委托技术实现模板方法结构

在前面相关的章节已经讨论了一些设计模式使用委托的情况，这里我们把它们放在一起，有助于更深的理解如何使用委托。

我们已经知道，传统的模板方法结构和实现如下：



```

public abstract class AbstractClass
{

```



```
public virtual string TemplateMethod()
{
    string s="";
    s=s+Title();
    s=s+Body();
    return s;
}

public abstract string Title();
public abstract string Body();
}

public class ConcreteClass: AbstractClass
{
    public override string Title()
    {
        return "我是名称";
    }
    public override string Body()
    {
        return "我是内容";
    }
}

public class ConcreteClass1: AbstractClass
{
    public override string Title()
    {
        return "新的名称";
    }
    public override string Body()
    {
        return "新的内容";
    }
}
```

调用:

```
AbstractClass obj;
private void button1_Click(object sender, System.EventArgs e)
{
    obj=new ConcreteClass();
    label1.Text=obj.TemplateMethod();
}
```

```

private void button2_Click(object sender, System.EventArgs e)
{
    obj=new ConcreteClass1();
    label1.Text=obj.TemplateMethod();
}

```

但我们也可以利用委托来实现模板方法，类的结构并没有多大变化，只是不需要定义成抽象类，也不需要继承，这样也就减少了耦合性。

```

//首先定义两个委托，决定方法调用的格式
public delegate string DTitle();
public delegate string DBody();

public class TemClass
{
    //模板类，基于委托的实现
    public string TemplateMethod()
    {
        string s="";
        s=s+Title()+" ";
        s=s+Body();
        return s;
    }
    public DTitle Title;
    public DBody Body;
}

//实现类，并不需要继承，但被调用的方法格式要一致
public class ConcreteClass1
{
    public string myTitle1()
    {
        return "我是名称";
    }
    public string myBody1()
    {
        return "我是内容";
    }
}

public class ConcreteClass2
{
    public string myTitle2()
    {

```

```

        return "新的名称";
    }
    public string myBody2()
    {
        return "新的内容";
    }
}

```

调用:

```

//构造模板的实例
TemClass tm=new TemClass();

private void button1_Click(object sender, System.EventArgs e)
{
    //利用委托定位到需要处理的方法指针
    ConcreteClass1 csc=new ConcreteClass1();
    tm.Title=new DTitle(csc.myTitle1);
    tm.Body=new DBody(csc.myBody1);
    //实现
    label1.Text=tm.TemplateMethod();
}

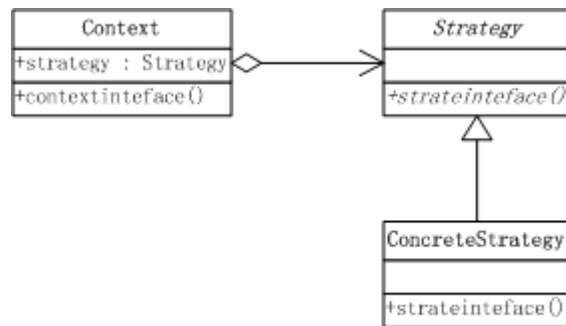
private void button2_Click(object sender, System.EventArgs e)
{
    //利用委托定位到需要处理的方法指针
    ConcreteClass2 csc=new ConcreteClass2();
    tm.Title=new DTitle(csc.myTitle2);
    tm.Body=new DBody(csc.myBody2);
    //实现
    label1.Text=tm.TemplateMethod();
}

```

显然，使用委托减少了类的层次结构，同时也减少了耦合性。
很多情况下，模板方法和工厂方法一起使用，可以达到非常好的效果。

三、采用委托实现策略模式

策略模式的结构如下:



策略模式的目的是使某种算法独立于调用算法的对象,在这种模式中,Context 依赖 Strategy 接口,这就有可能遇到和上面的讨论同样的问题,也就是某个现存对象实现了算法,但并不符合 Strategy 接口的规则,这就需要引入适配器。

应用委托技术可以避免这种情况,也就是用委托来代替 Strategy 接口。

```

using System;
using System.Collections;

namespace 策略和委托
{
    //构造委托取代接口
    public delegate string Strateinteface(string m);
    //包含算法的具体实现
    //注意,不使用继承,而且方法的名字也不同
    public class ConcreteStrategy1
    {
        public string myinteface(string m)
        {
            return m+":算法一";
        }
    }
    public class ConcreteStrategy2
    {
        public string yourinteface(string ma)
        {
            return ma+":算法二";
        }
    }

    public class Context
    {
        public Strateinteface strateinteface;
        //聚合可以用集合实现
        private Hashtable hs=new Hashtable();
    }
}
  
```

```

    public Context()
    {
        hs.Add("s1", new Strateinteface((new ConcreteStrategy1()).myinteface));
        hs.Add("s2", new Strateinteface((new ConcreteStrategy2()).yourinteface));
    }

    public string contextinteface(string KeyName, string strIn)
    {
        strateinteface=(Strateinteface)hs[KeyName];
        return strateinteface(strIn);
    }
}

```

实现:

```

Context c=new Context();
private void button1_Click(object sender, System.EventArgs e)
{
    textBox1.Text=c.contextinteface("s1","王小丫");
}

private void button2_Click(object sender, System.EventArgs e)
{
    textBox1.Text=c.contextinteface("s2","黄建翔");
}

```

四、用委托和事件机制实现观察者模式

.NET 提供了处理聚集的基本方法和事件响应方法，并且引入了委托的概念，因此，>NET 天然的支持观察者模式。

下面我们用一个最简单的例子，说明它是如何处理问题的。

首先我们必须定义一个委托，这个委托事实上提供了对方法方法的指针调用，也定义了被调用的方法必须遵循的形式：

```
public delegate void LoopEvent(int i,int j);
```

下面是例子，请注意代码中的注释。

```

using System;
using System.Windows.Forms;
using System.Drawing;

namespace 观察者
{
    //主题，相当于Subject

```

```
public class DoLoop
{
    public delegate void LoopEvent(int i,int j);
    public event LoopEvent le;
    public void BeginLoop(int LoopNumber)
    {
        for (int i=0;i<LoopNumber;i++)
        {
            for (int j=0;j<LoopNumber;j++)
            {
                le(i, j);
            }
        }
    }
}

//观察者显示窗口
public class Display:Form
{
    private Button button1=new Button();
    private TextBox textBox1=new TextBox();
    private Label label1=new Label();
    public Display()
    {
        this.label1.Location = new Point(72, 8);
        this.label1.Size = new Size(112, 24);
        this.label1.Text = "label1";
        this.button1.Location = new Point(160, 48);
        this.button1.Size = new Size(96, 24);
        this.button1.Text = "测量";
        this.button1.Click += new System.EventHandler(this.button1_Click);
        this.textBox1.Location = new Point(16, 48);
        this.ClientSize = new Size(272, 85);
        this.Controls.Add(this.textBox1);
        this.Controls.Add(this.button1);
        this.Controls.Add(this.label1);
        this.Text = "Display";
    }

    //这是观察者显示方法
    public void UpdateDisplay(int i,int j)
    {
        this.label1.Text=i.ToString()+"-"+j.ToString();
    }

    //测量，这是本身工作的事件，和观察者模式无关
    private void button1_Click(object sender, System.EventArgs e)
```

```

        {
            textBox1.Text=label1.Text;
        }
    }
}

```

调用:

```

//定义主题对象
DoLoop dl;
private void Form2_Load(object sender, System.EventArgs e)
{
    dl=new DoLoop();
}
//实现委托, 这个定义把观察者和主题联系了起来
private void button1_Click(object sender, System.EventArgs e)
{
    Display f=new Display();
    dl.le+=new DoLoop.LoopEvent(f.UpdateDisplay);
    f.Show();
}

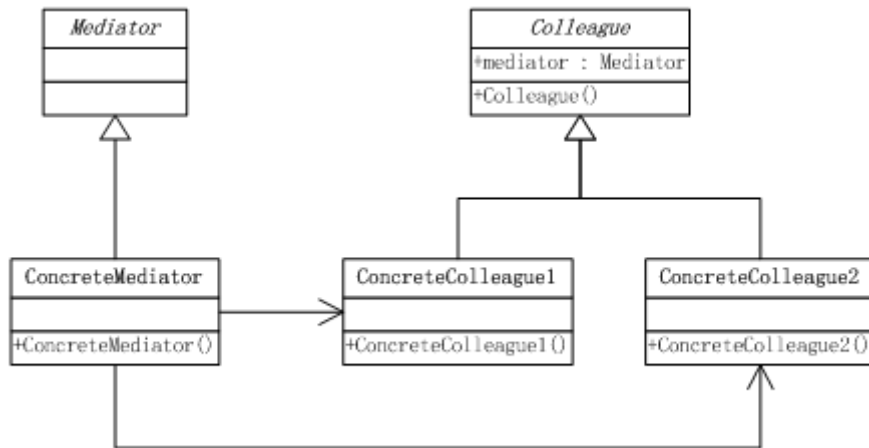
//可以把每个对象放在不同的线程中
//主题发送信息的时候, 各个对象本身还能正常工作
private void button2_Click(object sender, System.EventArgs e)
{
    Thread t=new Thread(new ThreadStart(aaa));
    t.Start();
}

void aaa()
{
    dl.BeginLoop(100);
}
}

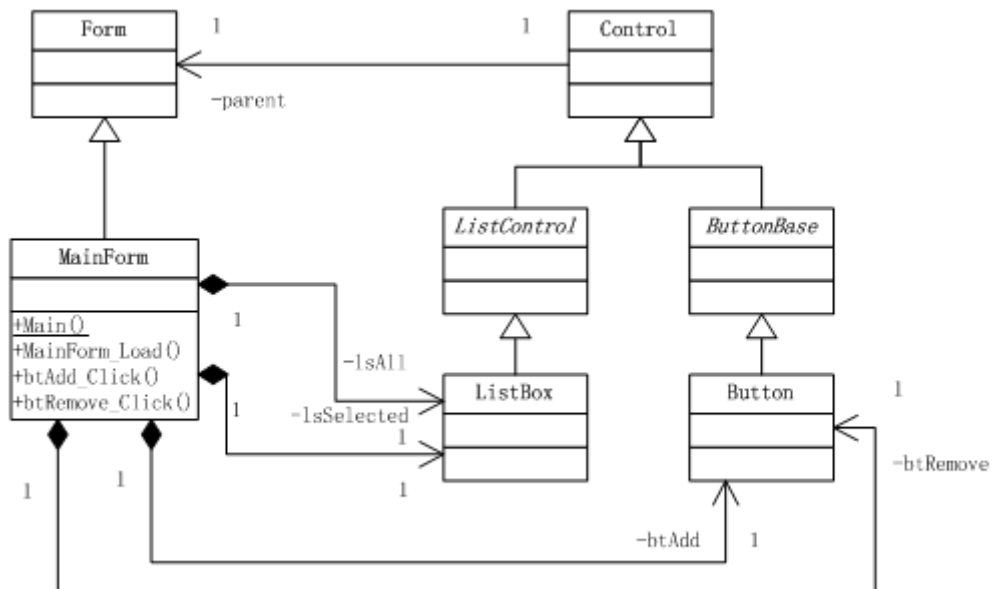
```

五、采用委托技术简化中介者模式

传统的中介者模式中可以看到 Colleague 对 Mediator 的引用, 引用的目的是通知 Mediator 自己要执行的操作。这样结构的缺点是 Colleague 对 Mediator 之间是紧耦合的。在.NET 开发的时候, 可以使用委托技术, 这样就解开了 Colleague 对 Mediator 的依赖, 所以.NET 开发这种技术用的很多, 静态结构可以变成下面的情况。



实例：窗体和控件



[STAThread]

```
static void Main()
```

```
{
```

```
    Application.Run(new MainForm());
```

```
}
```

```
private void MainForm_Load(object sender, System.EventArgs e)
```

```
{
```

```
    lsAll.Items.Add("中国");
```

```
    lsAll.Items.Add("美国");
```



```
lsAll.Items.Add("英国");
lsAll.Items.Add("俄罗斯");
lsAll.Items.Add("法国");
lsAll.Items.Add("德国");
lsAll.SelectionMode=System.Windows.Forms.SelectionMode.MultiSimple;
lsSelected.SelectionMode=System.Windows.Forms.SelectionMode.MultiSimple;
}

private void btAdd_Click(object sender, System.EventArgs e)
{
    for (int i=0;i<lsAll.SelectedItems.Count;i++)
    {
        lsSelected.Items.Add(lsAll.SelectedItems[i]);
    }
    for (int i=lsAll.SelectedItems.Count-1;i>=0;i--)
    {
        lsAll.Items.Remove(lsAll.SelectedItems[i]);
    }
}

private void btRemove_Click(object sender, System.EventArgs e)
{
    for (int i=0;i<lsSelected.SelectedItems.Count;i++)
    {
        lsAll.Items.Add(lsSelected.SelectedItems[i]);
    }
    for (int i=lsSelected.SelectedItems.Count-1;i>=0;i--)
    {
        lsSelected.Items.Remove(lsSelected.SelectedItems[i]);
    }
}
```



需要注意的问题是，Windows 本身就是一个中介者，所以大多数情况下并没有必要再定义一个中介者。之所以提出这个问题，是因为我们看到大多数介绍中介者模式的文章，均以人机界面为例子，但他们往往用实例生成 `ListBox` 和 `Button` 的子类，使生成的子类互相调用，然后又做一个 `Mediator` 解耦。但 `ListBox` 和 `Button` 本身并没有耦合性，生成一个子类人为地加入耦合性显然是多此一举，而且增加了程序调试的难度，这是一种设计过度的典型的例子。

这样的程序不但没有提高性能，相反更难维护。

之所以如此，是因为很多人受 GoF 的《设计模式》书中列举的那个例子影响，加上读书不求甚解，依猫画虎，其实这是学习设计模式最忌讳的事情。我们应该知道，《设计模式》形成的那个年代，很多待解决的问题在当前大多数平台上都得到解决了，时代在发展，我们不需要再走回头路。

六、设计模式的发展

委托可以使方法接口做为最小的接口单元使用，从而避免了不必要的继承，减少了类的层次，这也是用组合代替继承的最好的体现。

但是，委托也使设计更难理解，因为对象是动态建立的。

随着技术的进步，设计模式是需要随着时代的进步而进步的。从某种意义上讲，GoF 的“设计模式”从概念上的意义，要超过对实际开发上的指导，很多当初难以解决的问题，现在已经可以轻而易举的解决了，模式的实现更加洗练，这就是我们设计的时候不能死搬硬套的原因。

第十节 C 语言嵌入式系统应用设计模式的讨论

上面关于软件架构的讨论，特别是在详细设计的过程，完全是基于面向对象的技术的，显而易见，利用面向对象的基本概念，像封装性、继承性尤其是多态性，可以大大提升系统的可升级可维护性能，成为现代架构设计的主体。

但是，C 语言嵌入式系统编程中，软件架构的理论能适用吗？这也是许多人感觉困惑的问题。其实仔细研究面向对象开发的本质，可以发现在面向过程的语言体系中，完全可以应用面向对象

的思想，下面谈得仅仅是我个人的看法，供参考。

一、利用指针调用函数实现多态性

仔细研究一下上面关于委托应用的问题，也可以发现利用指针，完全可以实现基于接口编程的多态性的，而 23 个模式本质上是利用了多态性。调用函数的指针在声明的时候，本质上是规定了被调函数的格式，这和接口的作用是一样的。

例：有两个方法 max 和 min：

```
int max(int x,int y)
{
    int z;
    if (x>y) z=x;
    else
        z=y;
    return(z);
}
int min(int x,int y)
{
    int z;
    if (x<y) z=x;
    else
        z=y;
    return(z);
}
//定义max函数的指针
int (*p)(int,int);
p=max;
c=(*p)(5,6);

p=min;
c=(*p)(5,6);
```

二、C 的面向对象化

在面向对象的语言里面，出现了类的概念。类是对特定数据的特定操作的集合体。类包含了两个范畴：数据和操作。而 C 语言中的 struct 仅仅是数据的集合，我们可以利用函数指针将 struct 模拟为一个包含数据和操作的“类”。下面的 C 程序模拟了一个最简单的“类”：

```
#ifndef C_Class
#define C_Class struct
#endif
C_Class A
{
```

```

C_Class A *A_this; /* this 指针 */
void (*Foo)(C_Class A *A_this); /* 行为: 函数指针 */
int a; /* 数据 */
int b;
};

```

我们完全可以利用 C 语言模拟出面向对象的三个特性: 封装、继承和多态, 但是更多的时候, 我们只是需要将数据与行为封装以解决软件结构混乱的问题。C 模拟面向对象思想的目的不在于模拟行为本身, 而在于解决某些情况下使用 C 语言编程时程序整体框架结构分散、数据和函数脱节的问题。

在这样的情况下, 面向对象的理论和优点安全可以在 C 语言嵌入式系统编程中得到应用, 你甚至于还可以编写比较紧凑的层, 供二次开发使用。

三、C 的模块划分

C 语言由于其特点, 模块划分并不像典型的面向对象语言 (Java、C#) 那么清楚, 但我们还是可以认真的去研究, 结合实际搞好模块的划分。

模块划分是指怎样合理的将一个很大的软件划分为一系列功能独立的部分, 通过合作完成系统的需求, 这也是架构设计的核心知识。C 语言作为一种结构化的程序设计语言, 在模块的划分上主要依据功能,

但在面向对象的架构设计中, 功能块的概念实际上被弱化了, 但在 C 语言设计中, 这个思想仍然是可行的, C 语言模块化程序设计需要注意以下几个问题:

- (1) 模块即是一个 .c 文件和一个 .h 文件的结合, 头文件 (.h) 中是对于该模块接口的声明。
- (2) 某模块提供给其它模块调用的外部函数及数据需在 .h 中文件中冠以 extern 关键字声明。
- (3) 模块内的函数和全局变量需在 .c 文件开头冠以 static 关键字声明。
- (4) 永远不要在 .h 文件中定义变量, 定义变量和声明变量的区别在于定义会产生内存分配的操作, 是汇编阶段的概念; 而声明则只是告诉包含该声明的模块在连接阶段从其它模块寻找外部函数和变量。如:

```

/*module1.h*/
int a = 5; /* 在模块 1 的.h 文件中定义 int a */

/*module1.c*/
#include "module1.h" /* 在模块 1 中包含模块 1 的.h 文件 */

/*module2.c*/
#include "module1.h" /* 在模块 2 中包含模块 1 的.h 文件 */

/*module3.c*/
#include "module1.h" /* 在模块 3 中包含模块 1 的.h 文件 */

```

以上程序的结果是在模块 1、2、3 中都定义了整型变量 a, a 在不同的模块中对应不同的地

址单元，其实我们从来不需要这样的程序。正确的做法是：

```
/*module1.h*/
extern int a; /* 在模块 1 的.h 文件中声明 int a */

/*module1.c*/
#include "module1.h" /* 在模块 1 中包含模块 1 的.h 文件 */
int a = 5; /* 在模块 1 的.c 文件中定义 int a */

/*module2.c*/
#include "module1.h" /* 在模块 2 中包含模块 1 的.h 文件 */

/*module3.c*/
#include "module1.h" /* 在模块 3 中包含模块 1 的.h 文件 */
```

这样如果模块 1、2、3 操作 a 的话，对应的是同一片内存单元。

一个嵌入式系统通常包括两类模块：

- (1) 硬件驱动模块，一种特定硬件对应一个模块；
- (2) 软件功能模块，其模块的划分应满足低耦合、高内聚的要求。

也就是前面所述的原则这里都是适用的。

四、多任务系统和单任务系统

所谓“单任务系统”是指该系统不能支持多任务并发操作，宏观串行地执行一个任务。而多任务系统则可以宏观并行（微观上可能串行）地“同时”执行多个任务。

多任务的并发执行通常依赖于一个多任务操作系统（OS），多任务 OS 的核心是系统调度器，它使用任务控制块（TCB）来管理任务调度功能。TCB 包括任务的当前状态、优先级、要等待的事件或资源、任务程序码的起始地址、初始堆栈指针等信息。

调度器在任务被激活时，要用到这些信息。此外，TCB 还被用来存放任务的“上下文”（context）。任务的上下文就是当一个执行中的任务被停止时，所要保存的所有信息。通常，上下文就是计算机当前的状态，也即各个寄存器的内容。当发生任务切换时，当前运行的任务的上下文被存入 TCB，并将要被执行的任务的上下文从它的 TCB 中取出，放入各个寄存器中。

究竟选择多任务还是单任务方式，依赖于软件的体系是否庞大。例如，绝大多数手机程序都是多任务的，但也有些小灵通的协议栈是单任务的，没有操作系统，它们的主程序轮流调用各个软件模块的处理程序，模拟多任务环境。

五、合理应用中断服务程序

中断是嵌入式系统中重要的组成部分，但是在标准 C 中不包含中断。许多编译开发商在标准 C 上增加了对中断的支持，提供新的关键字用于标示中断服务程序（ISR），类似于 `__interrupt`、`#program interrupt` 等。当一个函数被定义为 ISR 的时候，编译器会自动为该函数增加中断服务程序所需要的中断现场入栈和出栈代码。

中断服务程序需要满足如下要求：

- (1) 不能返回值;
- (2) 不能向 ISR 传递参数;
- (3) ISR 应该尽可能的短小精悍;
- (4) printf(char * lpFormatString,...)函数会带来重入和性能问题,不能在 ISR 中采用。

例如,在项目的开发中,我们设计了一个队列,在中断服务程序中,只是将中断类型添加到该队列中,在主程序的死循环中不断扫描中断队列是否有中断,有则取出队列中的第一个中断类型,进行相应处理。

```
/* 存放中断的队列 */
typedef struct tagIntQueue
{
    int intType; /* 中断类型 */
    struct tagIntQueue *next;
}IntQueue;

IntQueue lpIntQueueHead;

__interrupt ISRexample ()
{
    int intType;
    intType = GetSystemType();
    QueueAddTail(lpIntQueueHead, intType); /* 在队列尾加入新的中断 */
}
```

在主程序循环中判断是否有中断:

```
While(1)
{
    If( !IsIntQueueEmpty() )
    {
        intType = GetFirstInt();
        switch(intType) /* 是不是很像 WIN32 程序的消息解析函数? */
        {
            /* 对,我们的中断类型解析很类似于消息驱动 */
            case xxx: /* 我们称其为"中断驱动"吧? */
                ...
                break;
            case xxx:
                ...
                break;
            ...
        }
    }
}
```

按上述方法设计的中断服务程序很小，实际的工作都交由主程序执行了。

六、硬件驱动模块的设计

一个硬件驱动模块通常应包括如下函数：

- (1) 中断服务程序 ISR
- (2) 硬件初始化
 - a. 修改寄存器，设置硬件参数（如 UART 应设置其波特率，AD/DA 设备应设置其采样速率等）；
 - b. 将中断服务程序入口地址写入中断向量表：

```
/* 设置中断向量表 */
m_myPtr = make_far_pointer(0l); /* 返回 void far 型指针 void far * */
m_myPtr += ITYPE_UART; /* ITYPE_UART: uart 中断服务程序 */
/* 相对于中断向量表首地址的偏移 */
*m_myPtr = &UART_Isr; /* UART_Isr: UART 的中断服务程序 */
```

- (3) 设置 CPU 针对该硬件的控制线

- a. 如果控制线可作 PIO（可编程 I/O）和控制信号用，则设置 CPU 内部对应寄存器使其作为控制信号；

- b. 设置 CPU 内部的针对该设备的中断屏蔽位，设置中断方式（电平触发还是边缘触发）。

(4) 提供一系列针对该设备的操作接口函数。例如，对于 LCD，其驱动模块应提供绘制像素、画线、绘制矩阵、显示字符点阵等函数；而对于实时钟，其驱动模块则需提供获取时间、设置时间等函数。

可见，C 语言嵌入式系统架构确实有自己的特点，考虑问题的重点也不太一样，但是只要深刻理解架构设计的基本思想，理解它的精神实质，结合自己的情况，走符合自己实际的路子，一样能设计出性能相当好的架构来。

只是单纯从形式上看问题是永远不会有前途的。

第十一节 架构师在软件架构设计中的作用

在软件组织中，架构是的作用是举足轻重的，为了尽快成长为一个成熟的软件架构师，最后我给你提如下一些建议：

1，聚焦于人，而不是工艺技术

事实上，软件开发是一个交流游戏，你必须保证人们能彼此有效的沟通（开发人员、项目涉众）。架构师要以最高效的可能方式与客户和开发人员一起工作和讨论，白板上书写讲解、视频会议、电子邮件都是有效的交流手段。

2，保持简单

建议“最简单的解决方案就是最好的”，你不应该过度制作软件。在架构设计上，你不应该描述用户并不真正需要的附加特性一个辅助的原则就是：“模型来自于目的”。

这个原则引发了两个核心的实践。

第一个就是描述模型的文档力求简单明了，切中要害。

第二个就是架构设计避免不必要的复杂性，以减少不必要的开发、测试和维护工作。

你的架构和文档只要足够好，并不需要完美无缺，事实上也做不到，因为建模的原则就是“拥抱变化”。你的文档应该重点突出，这样可以增加受众理解它的机会，同样这个文档会不断更新，

因此如何管理好文档显得十分重要。

3, 迭代和递增的工作

这种迭代和递增的工作,对项目管理和软件产品开发,事实上提出了更高的要求,你必须时时检验你的项目进展,不要使它偏离了方向。

4, 亲自动手

考察一下你遇见过的“架构师”,最棒的那一个一定是需要的时候立刻卷起袖子参加到核心软件开发中的那个人。架构师首先必须是编程专家,这是没有错的。

积极参与开发,和开发人员一起工作,帮助他们理解架构,并在实践中试用它,会带来很多好处。

- 你会很快发现你的想法是否可行。
- 你增加了项目组理解架构的机会。
- 你从项目组使用的工具和技术,以及业务领域获得了经验,提高了你自己对正在进行的架构事务的理解。
- 你获得了具体的反馈,用它来提高架构水平。
- 你获得客户和主要开发人员的尊重,这很重要。
- 你可以指导项目组的开发人员建模和小粒度架构。

5, 在开口谈论之前先实践

不要作无谓的空谈和争论,你可以写一个满足你的技术主张的小版本,来保证你的方案切实可行,这个小版本只研究最最关键的技术。这些主张只写够用的代码就行了,来验证你的方案切实可行。这会减少你的技术风险,因为你让技术决策基于已知的事实,而不是美好的猜想。

6, 让架构吸引你的客户

架构师需要很好的与客户沟通,让客户理解你的工作的价值,他如果明白你的架构工作会帮助他们的任务,那他们就会很乐意的和你一起工作。架构师的这种与客户沟通的技巧极其重要,因为如果客户认为你在浪费他的时间,那他就会想方设法回避你。你的架构描述能不能吸引客户,也成了建模是不是能顺利进行的关键。

7、架构师面对时代的考验

年轻人需要成长为合格的架构师,需要扎扎实实的从基础做起,不断提升自己的能力,并不是听过几个课程,就能够成为一个合格的架构师的。架构师必须善于学习,一个人最大的投资莫过于对自己的投资,每周花三个小时时间用于学习是完全必要的。

架构师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,同时要反复思索,把自己的思维实践和这种知识的飞跃有机的结合起来。

架构师要更看重企业所要解决的问题。

架构师要学会在保证性能的前提下,寻找更简单的解决方案。

做一个好的架构师并不是一个容易的事情,这需要我们付出极其艰苦的努力。

我这个课程的主题,就是“拥抱变化”,需求是在变化的,架构是在变化的,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。

好的架构师的优势在于他的智慧,而智慧的获得,需要实实在在的努力。