

软件架构设计的思想与模式

中科院计算所培训中心 谢新华

第一章 软件架构设计思想与体系创建

在软件组织中，架构师的作用是举足轻重的。本课程针对企业开发最关注的问题深入研讨，抓住投入产出比这个企业的核心价值，讨论架构设计如何使这个核心价值得以实现。我们认为，一个设计如果必须高手云集才能生产出符合质量要求的产品，并不一定是好的架构。架构设计的目标是力争使用总体上能力一般的队伍，通过组织和设计的力量，生产出符合质量要求的产品，从投资回报的角度，两者效果是完全不一样的。另一方面，由于需求变更不可避免，而需求的变更必然造成设计调整进而造成总体投入的增加，这会极大的影响到投资回报，所以我们必须研究架构设计如何更好的适应变更，通过设计确保变更、维护与升级的成本下降。对这一系列问题的深入思考，成为现代软件架构设计的核心思维。

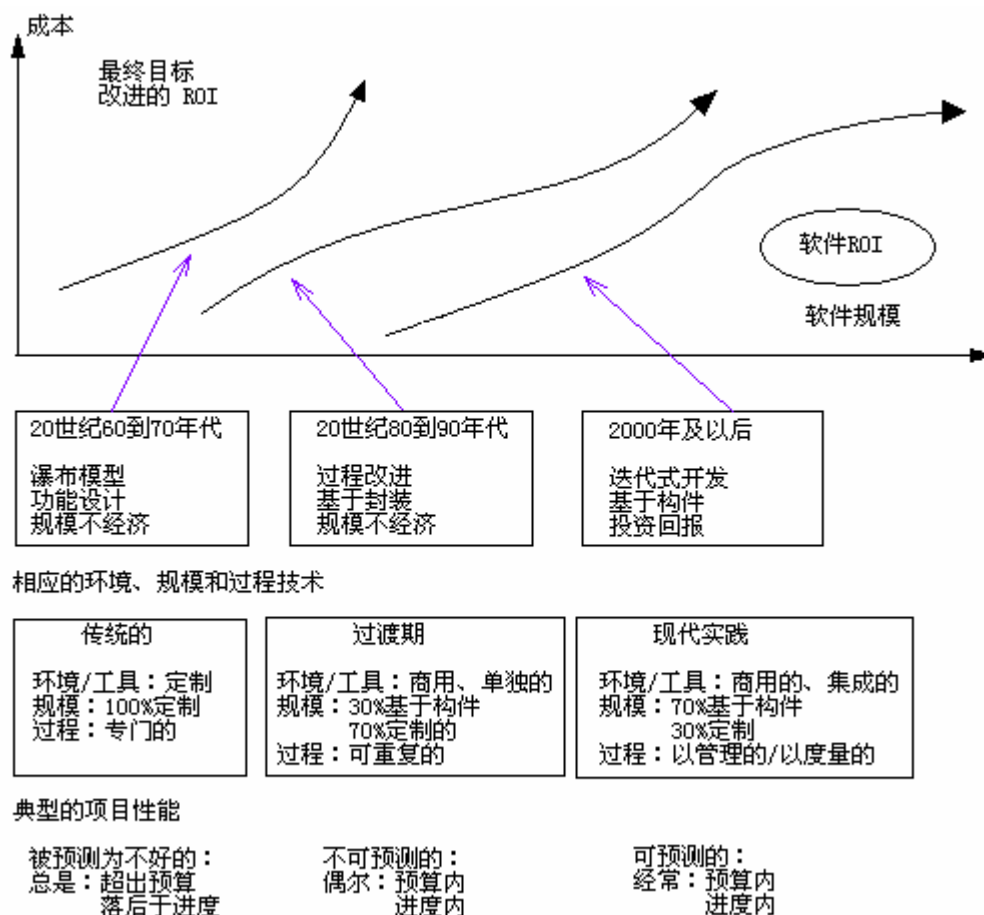
软件企业必须认真研究如何培养高水平的架构人员，但仅仅把架构设计作为一个孤立的节点来讨论，或者仅仅就架构谈架构的在一个很窄的思维空间中研究问题是没有意义的。任何设计都来自于目的，我们应该把架构设计放在整个项目过程的大环境下来研究，针对每个关键节点对设计的影响特点进行研讨，这样才可能真正理解架构设计真正精髓的东西，使未来的设计工作就会变得极有主动性和想象力。

随着经济全球化进程的不断推进，知识经济的时代已经到来。要增加软件产品的国际竞争力，软件质量作为企业发展的战略问题变得越来越重要，软件质量正被视为软件企业的生命。软件质量管理开始在软件组织内全面开展，强烈的质量意识正慢慢扎根于软件技术和管理人员的心灵深处，直至整个组织质量文化的形成，所以，如何设计高质量的软件产品，也成为软件架构设计的重要主题。

统计表明，软件质量问题 80%是由需求分析和架构设计两个环节造成的，因此，在需求分析的时候，我们必须研究如何充分理解用户需求，给各方面提供充分而有效的信息，在架构设计上，研究如何尽可能利用已有信息，合理组织技术方案，把人和任务作为一个重要因素进行考虑，在达到质量需求的基础上，使高的投资回报率成为可能，在项目过程管理上，如何与架构设计匹配协调，使技术方案的高质量实现成为可能，同时对于产品线架构和核心资产库构建的理论、方法、组织和技术给予足够的重视，这都需要项目经理、分析师、架构师具有很高的水平。

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

影响这个课程主体的主要思想，是 21 世纪是软件规模经济的时代，下图表达了工具、构件和过程的三个基本技术的进步，图中表达了在假定要求的质量和人员等级不变的情况下，投资回报（ROI）的关系，纵坐标表达了实现软件的单位成本（代码行、功能点），横坐标表达了软件规模，这里表示了随着时代的进步，同样规模的软件成本在大幅下降，投资回报在大幅上升。



伴随着非常大型的项目，比如由系统组成的系统，长生命力的产品以及多个相似项目的产品线，软件组织将达到更好的经济规模。这种规模软件经济的理念，对我的设计方法和思路提出了和过去完全不同的要求，重用、复用和变更促成为重要的主题。

架构设计应该从方法论的角度、从质量属性对架构设计影响的角度、从建立可度量的架构质量保证体系以及安全性和可扩展性的角度，在理论和实践两方面全方位研究问题。

1.1 软件架构师的角色和应掌握的知识体系

一、软件架构的定义与问题

软件架构（software architecture）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。

软件架构是对系统整体结构设计的刻划，包括全局组织与控制结构，构件间通讯、同步和数据访问的协议，设计元素间的功能分配，物理分布，设计元素集成，伸缩性和性能，设计选择等。

但是，所谓架构其实并不仅仅指的是软件产品体系结构设计，它还包括管理架构、过程架构以及质量保证架构等一系列问题的研究，因为高质量软件并不能只靠一个节点解决问题，而是需要有一个全面的解决方案。

重要的是要知道，一个软件系统的质量，很大程度上是由架构设计的质量决定的，所以，

研究架构设计的思想和方法，成为软件设计领域中一个十分引人注目的问题。但是很长时间以来，人们大都把目光关注在流程、方法、结构原理甚至编码的本身，而不太注意架构设计最本质的东西，思考的深度也欠深入，结果，很多产品即使设计出来，后期运行中也是问题百出。这就给我们提出了一个问题，架构设计的核心思维到底是什么？什么是好的架构设计呢？

首先，任何软件系统都是以满足需求作为目的，所以，好的架构设计必须以深入全面的需求分析作为基础，事实上架构设计是没有统一的模式的，任何模式只有针对问题才有意义。作为架构设计来说，必须对需求分析有足够的理解，这样才能有针对性地解决问题，才可能设计出真正优秀的产品来。但是，如果需求分析本身问题的信息就不足，那怎么可能设计出解决问题到位的架构来呢？

另一方面，任何架构思想的实现，都依赖于好的项目管理。项目管理必须与架构思想相匹配才能发挥作用。一个好的架构设计，必须关注成本，也必须关注时间，以期在约定的时间内、不超过规定的成本、生产出符合质量要求的软件产品来。因此，系统架构师应该仔细研究现代项目管理的思想和方法，吃透其中的精髓，根据自己的设计思想，提出项目管理的解决方案。

谈到项目管理，人们往往想到的是某种规范，某种文档模版，甚至某种流程。当然这些是重要的，不过，无论多么美好的规范，如果不能和自己的实际情况结合起来，尤其是和自己的架构特点结合起来，有的放矢的改进自己的过程，从而达到降低成本提高质量的目的，那什么样的规范都是没有意义的。

作为一个架构师来说，上述的讨论引发了三个核心思维，一个是架构设计的源泉来自于需求分析，第二个是架构设计重心和特点来自于质量需求（非功能性需求），第三个观点是，架构的实现依赖于好的项目管理。因此，软件架构设计是一个系统工程，它需要系统架构师有很宽的知识面，从需求分析、架构设计到类设计甚至代码实现一直到项目管理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。必须说明，软件系统设计的方法不是一个僵化的规则，关键是在实践中实事求是的摸索规律，从而找出符合实际达到要求的设计来。

二、软件复用和系统架构

现代软件架构设计的原则来自于软件复用，软件复用是指重复使用“为了复用目的而设计的软件”的过程。在过去的开发实践中，我们也可能会重复使用“并非为了复用目的而设计的软件”的过程，或者在一个应用系统的不同版本间重复使用代码的过程，这两类行为都不属于严格意义上的软件复用。

通过软件复用，在应用系统开发中可以充分地利用已有的开发成果，消除了包括分析、设计、编码、测试等在内的许多重复劳动，从而提高了软件开发的效率，同时，通过复用高质量的已有开发成果，避免了重新开发可能引入的错误，从而提高了软件的质量。

良好的软件架构提供了构件组装的基础和上下文。一个典型的软件架构是由系统中的构件、连接和约束构成的配置格局。从这个观点看，架构决不是概要设计，架构设计本身就可以看成一个项目，架构的结果应该是一个可执行、可验证的产品，也必须通过评审，为最终产品的复用与可持续发展打下框架上的基础。

研究软件构架有利于发现不同系统的高层共性、保证灵活和正确的系统设计、对系统的整体结构和全局属性进行规范约束、分析、验证和管理。将构架作为系统构造和演化的基础，可以实现大规模、系统化的软件复用。

研究软件架构对于进行高效的软件工程具有非常重要的意义：

- 通过对软件架构的研究,有利于发现不同系统在较高级别上的共同特性;
- 获得正确的架构对于进行正确的系统设计非常关键;
- 对各种软件架构的深入了解,使得软件工程师可以根据一些原则在不同的软件架构之间作出选择。

从架构的层次上表示系统,有利于系统较高级别性质的描述和分析。特别重要的是,在基于复用的软件开发中,为复用而开发的软件架构本身就可以作为一种大粒度的、抽象级别较高的软件构件进行复用,而且软件架构还为构件的组装提供了基础和上下文,对于成功的复用具有非常重要的意义。

软件架构研究如何快速、可靠地用可复用构件构造系统的方式,着眼于软件系统自身的整体结构和构件间的互联。其中主要包括:软件架构原理和风格,软件架构的描述和规范,特定领域软件架构,构件如何向软件构架的集成机制等。

软件开发实际上是从问题域向最终解决方案逐步映射和转换的过程,而特定领域软件架构(DSSA)和软件架构风格(Architectural Style)分别从问题域和软件解决方案两个方向提供了若干经过考验的候选转换路径。

特定领域软件架构 (DSSA):

这是一个领域中的所有应用系统所共有的体系结构,是针对领域模型中的领域需求给出的解决方案,也是识别、开发和组织特定领域可复用构件的基础。在国内外的金融、MIS、通讯和军事等领域中都开始注意到开发特定领域的软件架构和集成框架的重要性。

架构风格 (AS):

这是根据系统结构的组织模式确定了一组可以用于某种风格实例中的构件和连接方案,以及它们的拓扑结构、组装规则以及局部和全局约束,从而定义了一个面向系统结构的架构家族。软件架构风格与面向对象的设计模式或框架一样,为设计经验的复用提供了技术支持。

三、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。

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。另一方面,架构设计的结果主要依赖于对质量属性的理解,不同的质量需求往往可能得到一个完全不同架构设计,所以架构师对质量问题需要有一个透彻的理解。

软件架构设计是一个非常严肃、细致、敏感而且困难的工作,所以我们必须摒弃那些华而不实的名词堆砌,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。

1.3 在信息技术战略规划 (ITSP) 中的软件架构

好的架构设计,必须在信息系统战略规划 (ITSP) 的大环境下进行设计,才可能设计出真正优秀而且有价值的系统来,那么什么是信息系统战略规划呢?

信息技术战略规划 (ITSP) 的核心思想简述如下:

在信息时代知识经济的背景下,正确的结合 IT 规划,整合企业的核心竞争力,在新一

轮的产生、发展中取得更大的市场竞争力是必要的。

信息化的问题首先是企业管理层概念的问题。企业管理层的重视,和对信息化的高度认同是企业信息化的关键所在。当前国内很多企业管理层很关心资本运作的问题,而对很多国企业而言,管理层最关心的是扭亏增盈。信息化建设投入大、周期长、见效慢、风险高,往往不是企业需要优先解决的问题,导致管理层对信息化的重视程度不够,无法就信息化建设形成共识

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

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

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

如今国家正在大力推广企业信息化。然而人们大多从技术角度来谈论信息化和评价解决方案,他们往往脱离了企业的实际需要,以技术为本是不能根治企业疾病的。企业依然必须明确自己的核心竞争力。明确一切的活动和流程都是围绕让核心竞争力升值的过程。IT 规划意识如此,必须以企业核心、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,必须要在应用、结构、容量、通信能力、处理能力等方面具有较强的扩充性及进行产品升级换代的可能性。

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

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

Who: 为谁设计?

What: 要解决用户的什么问题？

Why: 为什么要解决这些用户问题？

这是一个被称之为 3W 的架构师核心思维，如果这个问题没搞清楚，就很快投入程序编写，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，甚至和企业目前的业务吻合得非常好，但为什么最终没有给企业带来实实在在的好处相反带来麻烦呢？一个专家感觉非常得意、技术上非常先进的系统，企业的使用者未见得感觉满意，这些情况在我的实践中屡见不鲜，即使一些知名的公司在设计的产品，往往都不能很好地把握，这足以证明我们必须下功夫来面对它。

那么，我们该怎么来做呢？

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

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

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

任何项目都是由项目的陈述开始的，在陈述的过程中比较难解决的问题是表达可度量的价值，所谓可度量价值实际上是一个预估，只是说由于加入了信息系统，解决了过去存在的问题，从预估的角度业务水平可能提升的一个度量数据。但并不等于说我管理依然是混乱的，只要有了这个信息系统，什么事情都不要做就可以达到这个水平了，这实际上是不切实际的幻想。所谓信息系统战略规划的本质，并不是说信息系统可以包打天下，而是说在整体规划下的信息系统，提升了我们的组织管理水平，减少了不必要的环节，提高了效率，通过全方面的努力，在可预测的未来，确实可以提升整体的经济效益，而且这个预测经过努力是可以达到的。

1.2 经典软件开发生命周期与过程

软件开发过程描述的是软件构造、部署还有维护的一种方法，成功的软件设计过程更多的是研究用户和市场，而不是技术本身。

经典的瀑布式过程大致的情况是这样的：

1) 收集市场数据，做市场分析。

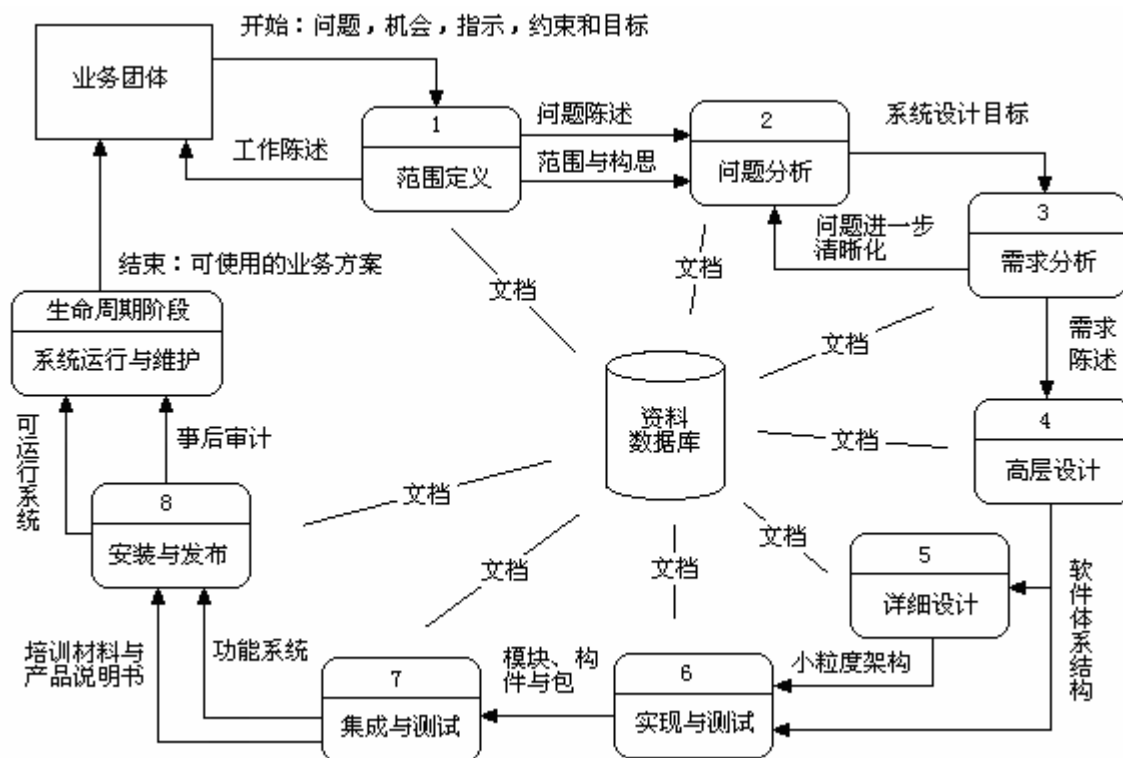
2) 确定用户，与用户交流，理解用户，理解用户的工作并与用户建立良好的关系，以便将来的设计和开发过程中经常得到他们的反馈意见。

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

4) 与用户交流进一步细化典型用户群，并写出场景脚本。

- 5) 确定软件的主要功能。
- 6) 确定这些功能的主次，并确定优先级。
- 7) 确定需求并写出说明书。
- 8) 由用户群检查需求说明书，看需求说明能不能满足用户的需要。
- 9) 进行软件架构设计。

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：



一、经典项目阶段

1. 范围定义阶段

范围定义主要考虑两个问题：首先是项目值不值得考虑。如果项目值得考虑，则确定项目的范围、目标、约束和限制条件，所需的参与者、预算和进度。这里主要由问题、机会和指示作为分析的根本。包括：

问题陈述：对问题、机会和指示的分类性描述，这是一个开发的合约。

约束条件：对约束条件的分类性描述。

在项目进行过程中，范围通常会变化，通过记录初始范围，我们就为（预算和进度）范围蔓延建立了一个基线。

2. 问题分析阶段

问题分析阶段是研究现有系统，分析发现的问题，促使项目团队更深入的研究和理解引发该项目的问题。分析员应该经常发现新问题，并回答这样一个问题：“解决这些问题的收益是不是超过了构造项目的开销？”

比如，我们定义了如下的改进准则：

把订单处理和交货之间的时间减少三天。

这时系统增加的复杂度带来的成本增加，是不是超过了我们能够认可的极限？

我们也可以把改进目标考虑成一种升级准则，新系统的升级准则可以向管理者口头汇报，也可以进行正式的书面报告。

3，需求分析阶段

需求分析的目的是回答下面这样一些问题：

系统给用户提供什么功能？必须收集什么数据？存储什么数据？期望达到什么样的性能等级？也就是系统需要做什么，而不是怎么做。

需求分析阶段需要定义业务需求，并且为它排序。需求分析也可能是开发阶段中最重要的阶段。只有当设计人员完全理解了需求之后，才可以开始设计工作。

不过，要注意避免分析瘫痪，也就是过多的系统建模极大的延缓了实现系统方案的进度。在这这就要求需求分析阶段，必须把握重点，解决最主要的问题。

把大问题切分为可以理解的小问题，是需求分析方法论的关键。

4，架构设计阶段

在这个阶段主要需要建立系统模型，这个模型又称之为逻辑设计模型。关于这个阶段需要注意的问题，将在后面详述。这个阶段主要是建立高层模型，系统和子系统的框架以及基于服务的层等。

5，决策分析阶段

当获得业务需求和架构系统模型以后，通常就具备了大量可选方案供我们决策分析，这里提出一些有关问题：

系统应该多大程度上应用计算机处理技术。

我们是购买组件还是自己编制。

是使用局域网还是使用 Web。

什么样的信息系统技术，会对这个系统有用。

这两个模型对于决策分析都是必要的输入，特别是架构多个备选方案，需要从项目团队不同的人那里，把不同的想法记录下来，最后确定选定方案。

要把决策的原因记录下来，其中包括：

技术可行性；

运行可行性；

经济可行性；

进度可行性；

风险可行性等。

6，物理设计和集成阶段

这个阶段也称之为详细设计，可以精细的把业务需求转换为系统模型。设计的过程包括：

按规格说明设计：这是一种最经典的方法，通过研究需求，生成设计的蓝图。

按原设计：通过构造不完整但可以工作的程序或者子系统，由用户和其他人员的反馈，不断地细化模型。实际上，通常是这两种极端方法的组合。

7，构造和测试阶段

构造和测试事实上就开始编码，在基于构件的设计和并行开发中，需要研究合理的测试用例和技术标准。开发人员在编码和测试中反映过来的问题是十分重要的，设计人员必须关注这些问题，用于部分的合理化系统。

8，安装和发布阶段

新系统往往与企业目前的行为方式相背离，所以分析员需要提供从老系统到新系统的平稳转换方案，需要准备一个转换方案，一般有一个平行使用期，在规定的时间内，突然切换到新系统。应该准备好用户手册和培训手册，必要的时候对用户进行培训。

事实上新系统会影响到原来的业务流程，分析员需要关注这种业务流程的变化对业务的

影响，以不断调整的姿态保证新系统的正常接入。

9、运行和维护阶段

一旦系统投入运行，就需要给运行提供不断地**系统支持**，在项目预算的时候，一定要把这笔费用计算进去。

在运行阶段收集用户反馈、表现出的问题以及业务需求的变化是十分重要的，这对早期研发升级版本提供了支持。系统一旦到了退役期，就可以迅速启碇一个新的系统开发过程来建立新项目。这个时候，在这个阶段收集到的信息，对新系统的设计及其有意义。

尽管现在有各种各样推荐的过程，但是这个基本的框架对于构造一个大型系统还是完全有必要的，我们的思维还是需要在稳定性和灵活性之间寻求平衡，包括开发过程也是如此。

二、软件开发生命周期增量模型

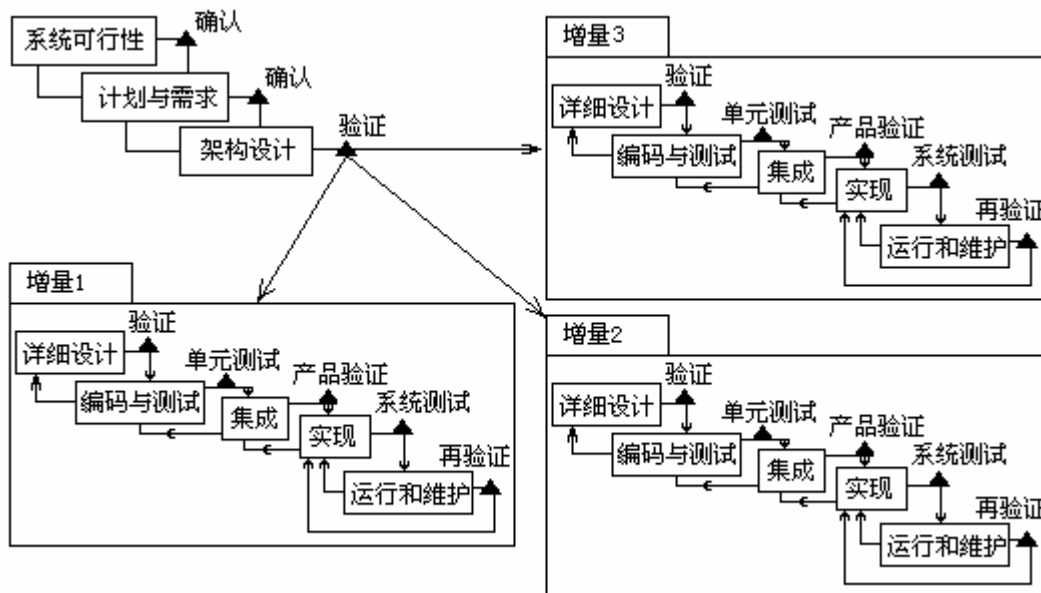
上述线性的或者瀑布的模型往往带来了很多问题，早期的模型要求在项目的第一阶段，在任何设计和实现工作之前，尽可能的推敲，把需求完全定义清楚，并把它稳定下来，并且实际开发前冻结需求，但历史证明这种方式是失败的，在项目很大的时候，冻结需求几乎没有可能。

对瀑布模型的一个关键性的改进，是所谓增量模型的出现。增量模型是指首先构建部分系统，再逐渐增加功能或者性能的过程。它降低了取得初始功能之前的成本，强调采用构建方法来帮助控制更改需求的影响，也提高了创建可操作系统的速度。

增量模型是综合了瀑布模型和原型化的产物，提倡以功能渐增方式开发软件，经验表明，这种增量模型在特大型项目和小型项目中同样适用。增量模型描述了为系统需求排定优先级然后分组实现的过程，每个后续版本都为先前版本增加了新功能。

在生命周期的早期阶段（计划、分析、设计），需要建立一个考虑了整个系统的架构，这个架构应该是具有强的可集成性的，后续的构件方式开发，都是建立在这个架构之上。剩下的生命周期阶段（编码、测试、交付），来实现每一个增量。

首先创建的应该是一组核心的功能，或者对于项目至关重要的最高优先级的系统，或者是能够降低风险的系统。随后基于核心功能反复扩展，逐步增加功能以提高性能。



这个过程也可以和其它模型结合（V型或者螺旋型），以降低风险和成本。

增量模型的优点：

- 整个项目的资金不会被提前消耗，因为首先开发和交付了主要功能和高风险功能。
- 每个增量交付一个可操作的产品。
- 每次增量交付过程中获取的经验，有利于后面的改进，客户也有机会对建立好的模型作出反应。
- 采用连续增量的方式，可把用户经验融入到细化的产品，这比完全重新开发要便宜得多。
- “分而治之”的策略，使问题分解成可管理的小部分，避免开发团队由于长时间的需求任务而感到沮丧。
- 通过同一个团队的工作来交付每个增量，保持所有团队处于工作状态，减少了员工的工作量，工作分布曲线通过项目中的时间阶段被拉平。
- 每次增量交付的结为，可以重新修订成本和进度的风险。
- 便于根据市场作出反应。
- 降低了失败和更改需求的风险。
- 更易于控制用户需求，因为每次开发的时间很短。
- 由于不是一步跳到未来，所以用户能逐步适应新技术。
- 切实的项目进展，有利于进度控制。
- 风险分布到几个更小的增量中，而不是集中于一个大型开发中。
- 由于用户能够从早期的增量中了解系统，所以更加理解后面增量中的需求。

增量开发必须注意的问题：

- 良好的可扩展性架构设计，是增量开发成功的基础。
- 由于一些模块必须在另一个模块之前完成，所以必须定义良好的接口。
- 与完整的系统相比，增量方式正式的回顾和评审更难于实现，所以必须定义可行的过程。
- 要避免把难题往后推，首先完成的应该是高风险和重要的部分。
- 客户必须认识到总体成本不会更低。
- 分析阶段采用总体目标而不是完整的需求定义，可能不适应管理。
- 需要更加良好的计划和设计，管理必须注意动态分配工作，技术人员必须注意相关因素的变化。

二、现代软件开发管理原理

对架构驱动的应用实践的总结，就形成一系列现代软件开发管理原则，这些现代原则与经典原则有很大的不同：

1， **把过程建立在构架优先的基础之上：**这要求在交付资源进行全面展开之前，就在需求、构架等重大设计决策和生命周期计划之间做出权衡。

2， **建立一个能尽早面对风险的迭代式生命周期过程：**对于今天高度复杂的软件系统，要按照顺序定义着那个个问题，设计整个解决方案、构造软件和测试最终产品几乎是不可能的。这就需要使用一个迭代过程，这个过程的方法很多，关键是定义好关键的里程碑，以及最终的目标，必须及早提出可能遇到的风险，以提高可预见性，避免昂贵的下游返工。

3， **设计方法向强调基于构件的开发转变：**从代码行至上转为基于构件开发的思想，可以大幅度减少开发的成本，构件是一个已经存在的代码行内聚集和，可以由原代码、也可以由可执行文件的格式提供，并且有定义好的接口和行为。

4， **建立一个变更管理环境：**迭代开发的动态特性，需要很好的变更控制环境，包括在共享产品上工作的不同团队之间并发的 workflows、客观需要的控制基线等。

5, **通过支持双向工程的工具增强变更的自由度**: 双向工程指的是位在不同格式(例如需求规格说明、设计模型、源代码、可执行代码、测试用例等)自动转化并同步工作信息所需要的一个环境支持,如果没有这种实质性的自动化支持,很难把迭代周期减少到可以管理的时间内。

例如: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。像 RequisitePro 这些工具,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。

6, **用严格的、基于模型的符号标记系统**: 具有严格语义的模型符号系统极其有意义,它可以减少人之上的歧义, UML 就是一种很好的符号语言。

7, **为过程配备工具进行客观的质量控制以及进展评估**: 必须对所有的中间产品进行质量评估,这些评估必须在度量的基础上进行,这些评估是在整体工程制品质量的基础上派生出来的。

8, **使用基于演示的方法评估中间制品**: 把产品的当前状态(不论是前期原型、基线架构或者是一个 bate 能力)转换为一个可以在相关场景下运行的演示,促使人们把注意力更早的集中在集成上,获得对设计折衷的更加切实的理解,并且尽早消除构架上的缺陷。

9, **计划在大量的使用场景中使用细节的进化等级进行中间发布**: 必须尽早的启动软件管理过程,项目增量迭代的进化必须与现有的对需求和构架的理解水平一致,因而采用内聚的使用场景(该场景是针对某些相互关联的问题设置),就成为组织需求、定义迭代内容、评估实现和组织验收测试的主要方法。

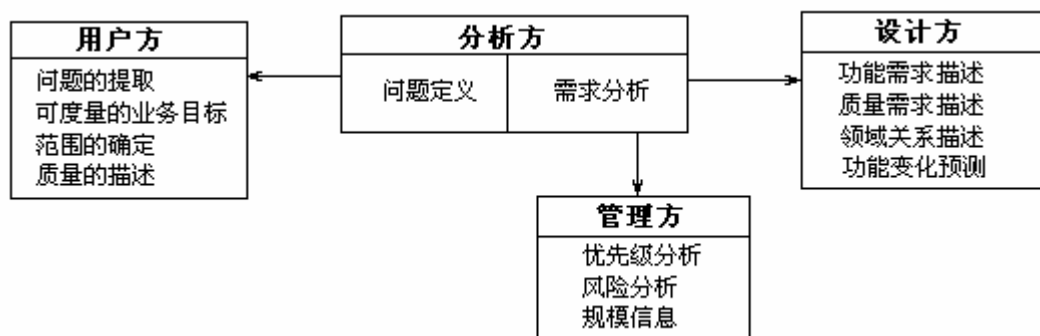
10, **建立一个经济上具有伸缩性的可配置的过程**: 没有一个过程是对所有软件开发都是合适的。一个实用的过程框架必须可以配置,以及支持更广泛的应用,这个过程必须应用共同的过程精神、广泛的过程自动化以及共同的构架模式和构件来确保规模经济和投资回报。

现代软件开发过程已经远离了传统的瀑布模型,尽管有一些变种,瀑布模型都要求开发过程依赖于前一阶段完成情况。而现代方法一般都要求在开发过程的前期迅速构造起一个初始版本,在这个版本中,强调的是高风险的领域、稳定基本架构、完善获取的需求。接着,开发就以迭代序列进行下去,构建核心的构架,直到达到期望的功能、性能和稳健性的等级。这种方法是通过不断的集成和完善需求、构架和计划减少了风险,而不是到了系统快要交付的时候,才惊讶的发现出现的错误。这种过程大大提高了管理上的难度,也需要管理者对问题理解的更加深刻。

第二章 需求抽取与业务建模

事实上架构设计是不可能独立存在的，架构设计提供的是用户需求的解决方案，所以一个架构师对需求分析的要点和关注点需要有深刻的理解，否则是不可能有好架构设计的。什么是需求呢？产品为用户在特定的背景中所必须满足的约束就是产品的需求，需求的表达一般是抽象的而且与技术无关的，这样主要是避免对技术方案产生影响，但架构设计的源泉来自于需求过程，也就是说，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，我们必须搞清楚，什么样的需求过程才会给架构设计提供足够的信息呢？

需求工程包括需求抽取、需求分析和需求管理，整个过程必须给用户方、管理方和设计方提供足够的而且是清晰的信息，对于分析方而言，各方面的要求主要集中在如下几个方面：



2.1 系统分析

一、从软件缺陷的产生看需求的重要性

软件的质量问题往往表现为缺陷（bug），软件缺陷的产生主要有两个原因：软件产品的特点和开发过程。例如：

需求不够明确，开发人员不太了解需求，不知道应该“做什么”和“不做什么”，常常做不合需求的事情，这方面的问题最多。

由于竞争激烈，过早使用新技术也容易产生问题。

有的企业为了在时间上取胜，认为实现很新、很酷的功能比质量更重要，因此时间上安排很紧，分析和设计的投入远远不够，测试也不到位，这是产生软件错误的主要原因之一。

除此以外，设计文档不清楚，文档本身就存在错误，沟通上存在问题，项目管理水平差，都可能导致问题。概括起来可以有七项原因：

- 项目期限的压力。
- 产品的复杂度。
- 开发人员的疲劳、压力或者受到干扰。
- 缺乏足够的知识、技能和经验。
- 不可解客户的需求。
- 缺乏动力。

从软件自身特点、团队工作和项目管理等多个方面进一步分析，就比较容易确定造成软件缺陷的一些原因细节，归纳如下：

1. 软件自身特点造成的问题

- 需求不清楚，导致设计目标偏离客户要求，从而引起功能或者产品特性上的缺陷。
- 系统结构非常复杂，而又无法设计成一个良好的层次结构或者组件结构，结果导致意想不到的问题或者维护、扩充上的困难。即使设计成良好的面向对象的系统，由于对象、类太多，很难完成多个对象相互作用的组合测试，而隐蔽着一些参数传递、方法调用、对象状态变化方面的问题。
- 新技术的采用，可能涉及实现没有考虑到的技术与系统兼容性问题。
- 对程序的逻辑路径或者数据范围的边界考虑不周，容易在边界条件下出错。
- 系统运行环境复杂，包括用户的各种操作方式及各种输入数据，容易引起一些特定用户环境下的问题，大用户量或者大数据量，也可能引发负载问题。
- 对于一些实时系统，如果时间同步设计不精确，也可能带来不协调、不一致的问题。
- 没有考虑系统崩溃后系统的自我恢复或者数据的异地备份等问题，从而存在系统安全性和可靠性的隐患。
- 由于通信端口多、存取加密手段的矛盾等，会造成系统安全性或适用性上的问题。

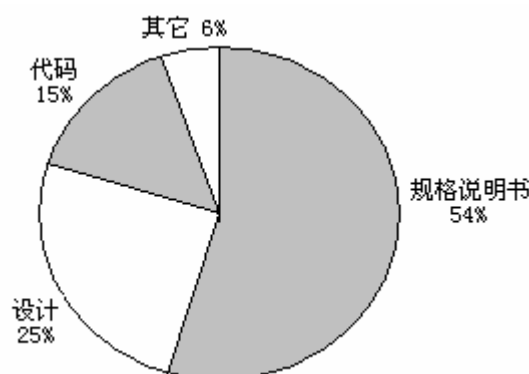
2, 软件项目管理的问题

- 缺乏质量文化，不重视质量计划，对质量、资源、任务、成本等的平衡把握不好，容易挤掉需求分析、评审、测试等时间。
- 系统分析对客户的需求不是太清楚，或者与用户的沟通存在一些困难。
- 开发周期短，需求分析、设计、编码与测试不能按照已经定义好的过程进行，工作不够充分，效果也就不完善、不准确，错误也就比较多。开发周期短，还给各类开发人员造成太大压力，引起一些人为的错误。
- 开发过程不够完善，存在太多的随意性，缺乏严谨的复审和评审机制，容易产生问题。
- 文档不完善，风险估计不足等。

3, 团队工作的问题

- 不同阶段的开发人员相互理解不一致，软件设计人员对需求分析结果的理解有偏差，编程人员对系统设计规格说明书中的某些内容重视不够，存在着误解。
- 设计或者编程上的一些假定或者依赖性，事先没有得到充分的沟通。
- 项目组成员技术水平参差不齐，新员工比较多，或者培训不足也容易造成问题。

软件缺陷虽然是由很多原因造成的，但从整个开发周期的统计结果来看，我们会意外的发现规格说明书是软件缺陷出现最多的地方，如下图所示。



换句话说，需求分析的不到位，是产生软件缺陷的最大原因。产生这种情况的原因如下：

- 用户一般是非计算机专业人员，软件开发人员与用户沟通存在着比较大的困难，对要开发的产品功能理解也不一致。
- 由于软件产品还没有设计、开发，完全靠想象去描述系统的实现结果，所以有些特

性还不够清晰。

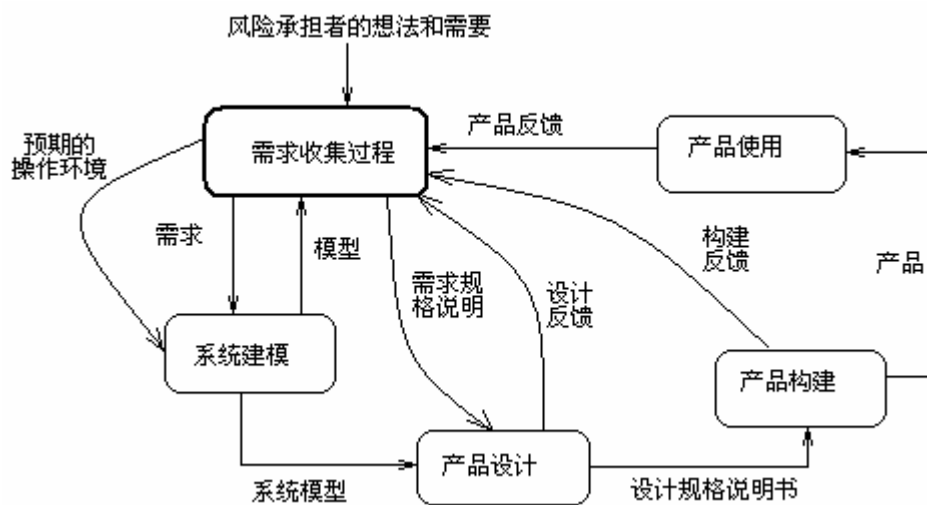
- 用户的需求总是在不断的变化,容易引起前后文、上下文的矛盾和需求描述的不一致。
- 需求分析没有得到足够的重视,在规格说明书的设计和写作上投入的人力、时间都不足。
- 没有在整个开发队伍中进行充分的沟通,有时只有架构师和项目经理得到比较多的信息。

因此,需求分析就显得非常重要,因为如果事情说不清楚,就没有办法做好。按出现问题的排位看,第二位是设计,第三位是编码。如果从软件开发各个阶段造成缺陷的分布来看,编码以后阶段的错误也要比前两个阶段少。正是对这个问题的理解,作为架构设计人员来说,有必要对需求分析的思想和方法有透彻的理解。

需求并不是加在项目上的额外负担,实际上可以使项目的进行更加顺利,如果还不太清楚要构建什么产品就开始构建了,那项目出现问题几乎是确定无疑的事情,但是,这并不等于需要完全理解需求以后才可以构建,也不意味着所有的需求都要写成书面的形式,而是意味着只有注意需求,才可能向用户提交有用和可用的产品。

需求的表达常常是抽象的,以一种与技术无关的方式表达,这样做的母的是为了避免解决方案对技术产生影响,需求是对业务方面的说明,而不能有任何技术实现上的偏好,产品设计的角色是把需求翻译成一个计划,按这个计划就可以构建出一个实体。

还有一点需要注意,好的需求意味着需求、设计和实现可以通过一系列的迭代循环来实现,每次迭代都会得到一些有用的功能,如下图所示。



所以,在产品构建好以后,需求不会冻结,下面我们对几个最重要的过程问题加以讨论。

二、需求的主要内容

需求是产品必须完成的事以及必须具备的品质,它主要包括如下几部分:

1, 功能性需求

功能性需求是为了向风险承担者提供的产品必须执行的动作。功能性需求源自于风险承担者需要完成的工作,几乎所有的动作,包括计算、检察、发布或者其它动作,都可以是一项功能需求。

功能性需求是用户给定业务背景下必须要做的事情。

2, 非功能性需求

非功能性需求描述了产品必需具备的属性和品质，在某些情况下，非功能性需求描绘了诸如观感、可用性、安全性和法律限制等需求，这些对于产品的成功是至关重要的。例如，产品必需在 0.25 秒之内便认出是朋友还是敌人。

非功能性需求通常在产品功能确定以后（但并非总是如此），也就是说，一旦知道产品要做的事（功能性需求），就可以确定它的行为方式，它需要具备什么品质，以及它的响应速度、可用性、可读性和安全性等。

3, 限制条件

限制条件是全局性的需求，又称之为“约束条件”，它可以是项目本身的限制，或者是对产品最终设计的限制，比如：产品必需在新的税务年度开始之前准备好。这个约束的效果是，分析师必须对需求分析进行限制，只包括那些在最后期限能够提供最大价值的需求。

又比如限制条件时针对产品最终设计和构造的：产品运行在 3G 手机上。结果不满足这样限制的解决方案是不可接受的。

所以，限制条件是另一种类型的需求。

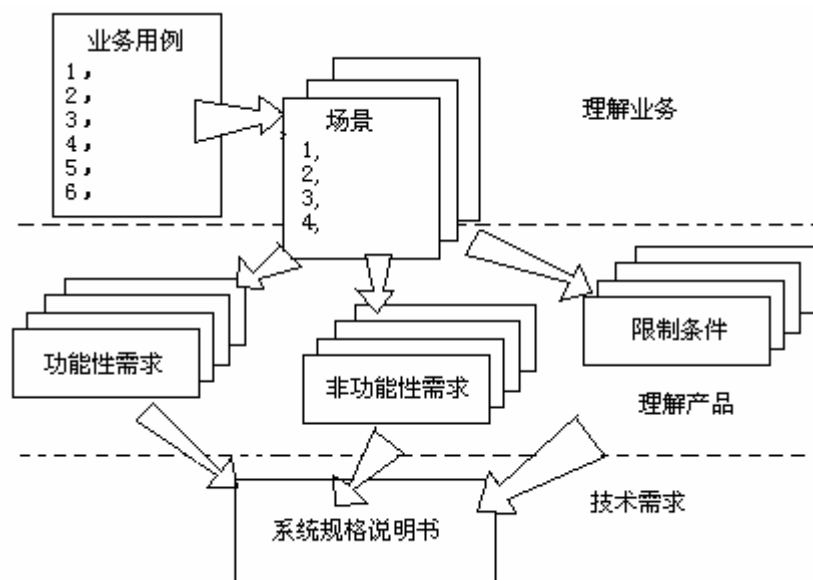
三、需求演进

在项目开始的时候，需求分析师关注的是产品将要支持的业务（或者称之为工作），在这个阶段，分析时研究场景及其它模型，帮助他们与风险承担者讨论在业务上应该在什么，在这个问题上达成一致意见，这时候的需求可以称之为“业务需求”。

随着对业务理解的加深，风险承担者对有助于自己工作的最佳产品做出了决定，现在需求分析师开始确定产品的详细功能，并且编写“产品需求”。

非功能性需求几乎是在同一个时间导出来的，与限制条件一起被记录了下来。此时，需求使用一种与技术无关的方式写下来，它规定了产品应该为工作做些什么，但没有规定工作应该用什么技术来实现。

当对他们的理解有足够的程度地时候，这些需求就可以交给设计者，他们添加产品的“技术需求”，然后为构建着提供最终的设计规格说明书。

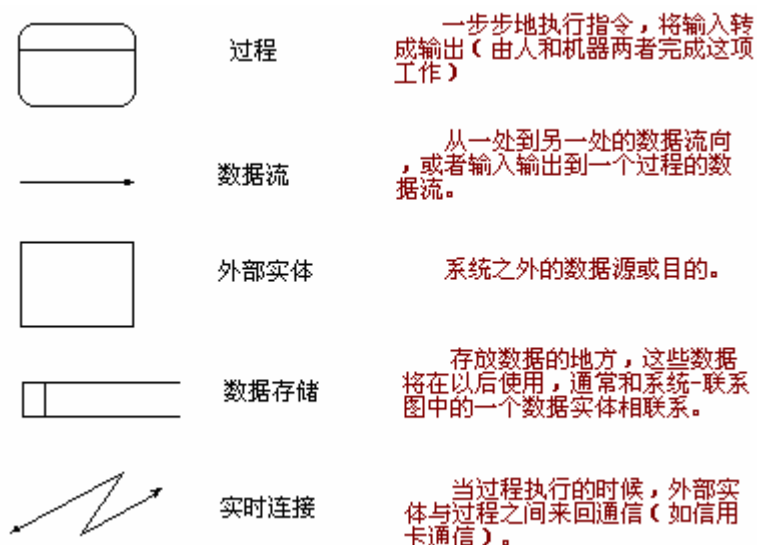


2.2 需求的过程定义

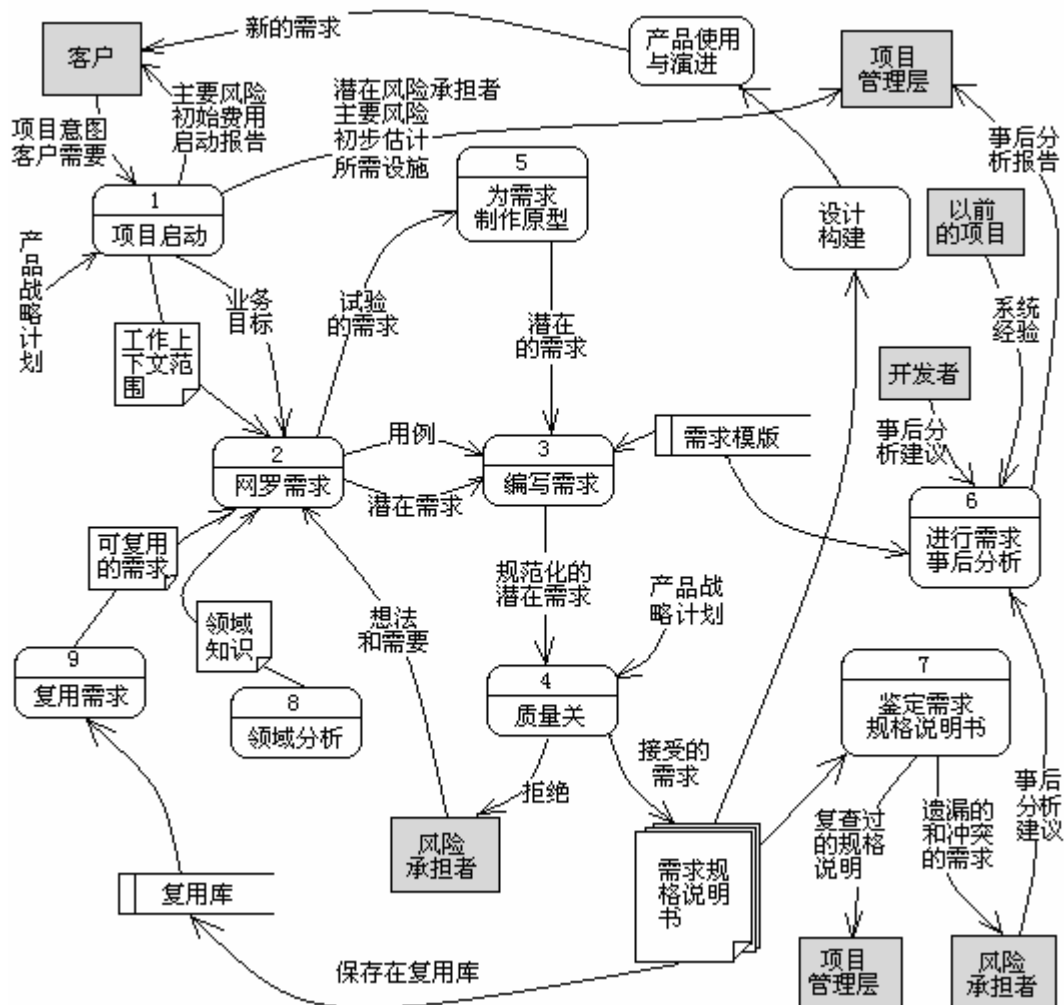
任何重要的工作都需要某种有序的过程，而且参与这项工作的人，必须知道为什么这个过程中有些步骤是重要的，知道哪些部分对他们的项目具有特定的重要意义，这就需要对需求过程进行定义。应当说明，这里我们定义的过程是为了得到提交产品的一个指南，而不是一个一成不变的程序，更不是一个必须的“做事情的方式”，我们可以从中理解很多有用的事情，从而更有效、更准确地收集需求。

需求过程在软件开发过程中是一个关键过程域，它建立了一套需求收集、建模、分析、和描述的循序、方法和模版，使需求可以以一种有序而深入的方式进行。但是没有什么过程是放之四海而皆准的，我们必须根据我们自己的实际情况来定义个改进我们的需求过程，我们不希望拿着课程中的过程就直接使用，而是仔细分析，创建一个构建相关产品的最佳方式，根据需要调用这个过程，使它符合您的项目和组织机构。

定义过程实际上就是定义软件分析的业务，这是一个成熟的软件组织必须要做的事情，在这个总体过程的基础之上，我们可以把每个过程在分解为相应的子过程，这样就可以使我们的思维活动变的可操作。事实上定义需求过程的本身，也是也是进行业务建模的一个训练，可以使用数据流图（DFD）表达，箭头表示数据的流动（或者过程的流动），DFD 只用了 5 个符号如下图所示。



和所有高层图一样，图中并不需要表示面面俱到的内容，而只是把最重要的关系表达出来，细节可以通过过程的分解来达到。



后面我们会围绕这个过程中的重要的子过程仔细分析，分析这些过程的细节（将要表达的过程都编了号），这样就可以把问题越来越深、越来越透彻的研究和分析清楚，所以，这个课程本身就是一个需求分析的训练，从而得到需求分析的正规、详尽而全面的知识。

需求过程是没有终点的，当项目和产品的一部分已经交付，用户开始使用它的时候，演进过程就开始了，人们会发现一些新的要求和用途，希望产品得到扩展，这就提出了新的需求。正因为产品自身有一个演进的过程，就可以决定先构建一个包含较少功能的早期版本，然后通过所计划的一系列发行版本来支持它。

同时也要注意围绕这个过程的人，这些人为了这个过程提供信息，或者从过程接收信息，这些人有一部分是风险承担者，也可以是对项目感兴趣的团队，他们具有收集产品须求所需要的知识。

需求过程不仅仅适用于从头开发产品的情况，对于迭代或者升级的产品也同样适用，本节的说们旨在对过程的各部分如何配合有个概貌的了解，但并不准备讨论若干细节。

下面我们对主要子过程加以说明。

2.3 项目启动

项目启动主要要确定三件事情：

- 产品实现的目标以及确定范围。
- 发现和确定主要风险承担者。

● 限制条件以及项目的可行性。

启动会议对项目进行准备，并且在开始详细的需求工作之前确保项目的可行性。它的特点是，主要风险承担者（客户、主要用户）、首席分析师、技术业务专家以及其他对项目的成功有关键作用的人聚在一起，对关键的项目问题达成一致意见。

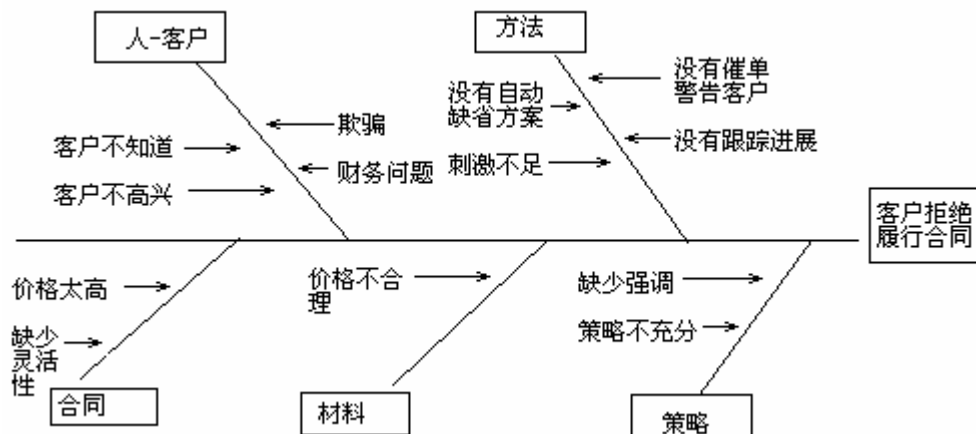
由首席分析师协调整个小组，在启动会议上，首先需要确定业务问题的范围，参与者在白板上画出系统上向文模型，以及系统如何与外界联系。

在业务问题差不多达成一致以后，小组开始确定风险承担者（项目中的利益相关人）。

项目启动也确认了项目的目标，小组也必须对项目是否值得开发，以及组织机构的能力等问题达成一致意见。

一、调查研究技术

调查研究事实上是一个需求获取过程，在这个过程中，一定要注意的是不要把症状当成问题，我们必须关注真正的问题到底在哪里，以期找到业务上的问题症结所在。在分析和讨论的过程中，可以使用一种称之为鱼骨图的方式，这是日本人首创的方法，它是一种确定、探索、何描述问题及其原因和结果的图形工具。



例如，对于客户拒绝履行合同的问题，可能有五个方面的原因，以及若干具体问题。在讨论会上，在白板上书写这样的图可以加深对问题的理解（请用数码相机拍下来）。利用这样的图可以清楚地描述可能出现的问题和应对措施。

所谓调查研究，是通过研究、面谈、调查表、抽样以及其它技术来收集关于问题、需求、偏好等问题的正式过程。在分析阶段，系统分析员应该了解一个企业和系统的术语、问题、机会、约束条件，以及优先级。一般来说，不管项目如何，都会有一个框架帮组我们在早期阶段收集事实。

我们可以对现有的文档、表和文件进行抽样来收集事实。

也可以通过观察工作环境甚至观察工作中的人来收集事实。

或者通过精心制定的调查表来收集事实。

也可以通过和客户面谈来收集事实。

面谈是一种重要的收集事实的方法，我们必须选择被接见者，制订问题方案，在面谈中学会聆听对方的表达，并且能够迅速抓住问题的本质。

需求必须通过一种形式化的方法记录和表达，以便于和客户沟通。

二、范围定义

高级系统分析员和项目经理通常领导这个阶段，在范围定义阶段我们需要做的事情是：

1，列出问题和机会：

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

2，协商项目的初步范围：

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

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

3，评估项目价值：

在这个任务中我们必须回答这样的问题：“这个项目看上去是不是值得？”项目的结果能够带来足够的价值低效开发费用吗？

4，计划项目进度表和预算：

如果项目值得做下去，就可以深入的规划项目了。初步的规划至少要包含以下几个部分：一个初步的主计划（包括进度和分配给整个项目的资源，这个计划会在项目的每个阶段结束的时候更新，又称之为基线计划），用于完成项目下一阶段（问题分析阶段）的详细计划和进度表。这个任务通常是项目经理的工作。

5，汇报项目计划：

当这些问题研究清楚以后，可以向项目指导部门（由主要业务管理人员和系统管理人员组成的委员会）汇报，并获得批准，至此，项目可以启动了。

三、问题分析

“除非你已经理解了它，否则不要试图修改它。”这就是问题分析阶段的任务。任何分析都来基于对问题的理解，一般没有办法跳过问题分析阶段，在问题分析阶段主要的任务是，充分研究和理解问题域中存在的问题、机会和约束条件。通过上下文图队整体的问题进行描述。这个阶段通常包含如下任务：

1，研究问题领域：

项目团队首先必须了解当前开发的系统需要解决什么问题，可以通过系统所有者、用户、系统分析人员的参与，从不同的侧面，不同的详细程度，不同的表述方式，不同的感知，不同的观点详细研究这个领域的业务、机会、指示和约束。不要吝啬开会，在白板上讨论并且用数码相机拍下讨论的过程十分重要，初期的准备是必要的，但讨论的结果和初期的准备会很不相同。

2，分析问题和机会：

通过因果分析得出对问题的真正理解。

3，分析业务过程：

业务过程分析应该避免对信息技术本身问题的关注，可以使中提出这样的问题：“如果目前的流程是完美的，还需要这个信息系统吗？”

在这个过程中还需要了解诸如：通过过程的数据量，每个过程的响应时间，系统中出现的任何延迟和瓶颈，这些问题对于系统改进是十分宝贵的资料。

4, 制定系统改进目标:

在上述问题已经清楚以后, 就可以制定改进目标, 也就是成功的准则应该按照目标进行度量。目标的制定应该清晰避免含糊, 应该便于测量而且是精确的、可度量的业务性能陈述, 比如:

下一年把不可收集的用户账号减少 50%;

当个工作站失效重新调度另一个产品所需时间减少 50%。

下面的描述是劣质的: “产生一个有拖欠债务的账号报告。” 而应该表述成: “更早的产生拖欠债务的账号报告, 以减少信誉损失 20%”。

系统改进目标可能受到约束条件调节, 比如:

进度: 系统必须在 4 月 15 日前运行。

成本: 系统成本不能超过 35 万。

技术: 所有的新系统必须使用 Oracle 数据库。

政策: 新系统必须使用“双倍递减余额”技术。

5, 修改项目计划:

由于深入研究的结果, 范围定义做出的项目基线计划一般会做出更改, 这种更改对计划的完善是很重要的。

6, 汇报调查结果和建议。

在项目启动阶段, 需要达成一个大家一致同意的目标, 并且能够清晰的、无歧义的、可度量的方式记录下来, 把项目的得益处量化, 工作就会有很大的回报。

项目的目的不应该仅仅是解决问题, 还要提供业务上的好处, 如果存在这种好处, 就必须度量它。

还有必须要粗略的估算成本, 在启动的时候估算成本是比较困难的, 并不需要使用十分精细的方法, 比较常用的方法是计算上下文模型中相邻系统的数目, 以及输入和输出数据流的数目, 可以大致的算出来“每个输入/输出数据流的平均费用”, 一般来说, 如果历史上有机构这个平均费用的数据, 大致费用的估计误差应该在可以接受的范围内, 总比凭空猜测规模好的多。另外还要预估项目的和风险, 最后决定项目的继续还是终止。

四、项目陈述

项目的陈述一般包括如下几个方面的考虑:

1, 问题: 不期望发生的情况, 它妨碍组织完整的实现其任务。

2, 机会: 能够改善组织的可能性。

3, 指示: 管理层、或者其它外部影响强加的新需求。

具体的问题陈述一般应该包括如下几个部分。

4, 项目的目标;

5, 项目概念;

6, 问题陈述;

7, 项目影响范围;

8, 项目构想;

9, 有关的约束: 业务限制; 技术限制;

2.4 网罗需求

启动会议以后, 分析师需要使用一系列的技巧网罗需求, 他们需要对启动会议上确定的

业务领域和工作进行学习，并且把业务的上下文分解成一些业务用例，每个业务用例都正确响应一个业务事件所需的功能。捕获需求让风险承担者密切参与是很有好处的，需求分析师和风险承担者一起确定，哪些是需要信息化系统来解决问题，当产品方方面面的特点被描述清楚以后，分析师就开始编写需求。

在网罗需求的时候，分析团队和用户一起工作，用户描述他们正在做以及他们希望做的工作，而分析师需要向用户（包括易用性专家、安全专家、操作人员等）咨询，使我们知道产品将来工作是什么样子的。

分析团队也可能需要帮助用户创造一个更好的工作方式，并且构造一个系统来支持这样更好的工作方式，基于这一点，可以举行有创造性的研讨会，团队成员和用户都可以发挥想象力，以创建一个更新、更好的最终产品的想法。

一、引出需求方法论问题

软件需求的引出有一系列的方法和共同活动的思想：

- 实际用户需要参与到需求收集的活动中。
- 应该确定所有的相关人员，每类相关人员的代表都应该参与需求收集。
- 模糊的需求应该被确认为原型化的候选者。
- 客户和发起人不一定是所交付的应用程序的使用者，但他们是支付需求工作和最终系统费用的实体。
- 应该确定可能限制系统共能性或者性能的领域约束。
- 每个相关人员都有其特有的偏见，其中包括对开发组织。
- 如果开发的仅仅是软件，就需要定义产品的目标环境（计算机结构、操作系统、通信等）。
- 如果开发的是整个系统（软件和硬件），在软件需求引出来之前，必须存在系统需求。
- 需求过程的输出包括一系列有功能、领域约束、使用案例和原型化（如果可能的话）组成的高层次上也对象。
- 应该记录每一需求的基本原理。
- 在有合适的人员时，引出方法可以最终做出决策。

下面我们讨论一些最重要的引出方法。

二、找出工作的本质

本质对于所有的项目、所有的工作领域和所有的工作层次都是重要的，只有理解了问题的本质，我们才可以发现正确的需求，并最终找到正确的解决方案，事实上，当发现了问题的本质的时候，解决方案通常是不言而喻的。

在网罗需求的时候，所听到的许多东西不可避免的是风险承担者关于“解决方案”的想法，而不是他们试图解决的问题的描述。需求分析师的任务，就是解释他们所说的，从而揭示出“本质”。工作的本质，代表了工作存在的根本原因。例如道路除冰调度系统的存在的根本原因，是准确的预测道路何时将结冰，怎样做到这一点（技术、设备、计算机和人）都是它的实现，并不是它的本质。

需求分析师必须能够分离问题的本质和所建议的解决方案，这样做的原因如下：首先，这样能够解决真正的问题，而不至于被实现细节拖着走。其次，这样也能够避免无意中重复实现过去的技术决定，几年前构建系统实现的技术方案，在今天不一定合适。

一个正待解决的问题或者业务策略，可以不掺入任何技术来描述，也可以使用多种技术方案来实现。如果发现了本质，就意味着理解了真正的工作，在大多数情况下，这种对本质的理解会带来更好的解决方案。

但是，现实中已经存在的技术方案可以作为我们发现本质的源泉，举个例子：

当走进一台 ATM 机的时候，希望进行的业务本质是什么？“从账户取钱”可能是个答案，至于如何插入银行卡、输入密码等，与银行所使用的技术有关，本质是访问了银行账户，并取走了钱。

应该注意技术的自由度有多大，考虑“访问账户并取走钱”，实际上存在多种技术可能性，但不论何种方式，但只有一个本质，这样的需求现在是怎样实现的，将来又会怎样实现，这些目前都不重要。

这就是说，网罗需求的时候，必须忽略要用到的技术，不论这些技术描述多么令人兴奋，我们还是要关心底层概念，所有的技术都是因为真正的工作而存在的。

如果需求中包含了技术元素，那这种技术就变成了一项需求，如果需求包含了实现方法，那它就是解决方案，而不是需求，假定有这么一条需求：“如果气象站传送数据失败，产品应该发出鸣叫声，屏幕上应该有一条闪烁的信息。”问题是是不是还有更好的方法？所以这样的需求应该这样来写：“如果气象站传送数据失败，产品应该发出警告。”

又比如，风险承担者希望使用一个触摸式地铁网地图，乘客选择目的站，产品计算相应的车费，作为一个辅助功能，还可以显示出到达目的站的最快路线。显然，这是个聪明的解决方案，但这并不是本质的表达应该是：“产品将接受乘客输入的目的地”。因为这种解决方案并不是唯一的方式，可能还有更好的方式。

重要的是，我们不应该事先确定下实现方式，不论自认为自己理解问题有多深，也不论某种技术多有吸引力。抛弃这种先入为主的思维方式并不容易，很多组织都发现这是最难传递给客户的概念之一，如果采用极限编程就更困难，因为 XP 的目标是早期得到解决方案，并很快就转入下一次迭代。所以是不是在前期认真地思考，仔细地体察用户解决问题的想法，一旦这样做，就不仅会得到做好的解决方案，解决方案也不会突然发生变化。

三、解决正确的问题

这里讨论的思考方式延续了对本质的理解，而且于所有的需求分析师都有关系。

我们常常看到这种情况，项目团队开始构建一个华而不实的产品，这个产品构建好了以后，尽管项目团队热心的推介一些功能，但用户从来没有使用过这个功能，为什么呢？因为它解决了错误的问题。

一个人在灯光下找钥匙，他觉得这里亮，但钥匙是掉在另一个街区的黑暗角落，那里才是潜伏着问题的区域。

例如，一个财务公司，它需要建立一个新系统，能够有效的重设口令。他们统计出原来系统为了建立信誉，其成本为每年几百万美元，他们希望新系统能减少这个成本。问题是新系统并不能建立信誉，真正的问题是：“顾客忘记了他们的口令怎么办？”

事实上发现正确的问题似乎比构建一个系统更难：例如，需要找到一种方法让用户记住他们的口令。问题是口令不是业务问题的一部分，他们是银行选择的技术。要解决的正确问题是：“允许用户安全的访问他们的账户，而且不要求顾客放具有强大的记忆力。”事实上能满足这样要求的技术是很多的。

如果我们在开始的时候思考产品而不是工作，我们常常会开始解决错误的问题，如果想产品的内部看，项目团队就不能看到更大的工作。

四、业务重组与创新的产品

我们主张，今天的需求分析师必须寻求改进客户工作的方法，这称之为业务重组，改进通常是通过创新来实现的。当与风险承担者一起工作的时候，他们需要改进现有的状况，需求分析师就需要创新。当在一个新的应用领域工作的时候，也需要创新，很显然，如果一个软件公司这样组织工作，创新就能让自己保持领先。

如果要在今天的商业市场上竞争，或者与大量上架销售软件竞争，或者与离岸外包公司竞争（通常可以廉价生产软件），或者与各类开放源代码竞争，那么创新是最大的支持者。不论是降低开发成本，还是沿着 CMMI 阶梯往上爬，但如果产品不能够镇住客户，就不能带来任何好处。

目前客户对软件产品的困惑到底在哪里呢？作为分析师来说，如果仅仅是要风险承担者说出他们的需求，自己仅仅是记录员，这在风险承担者很清楚的知道他们需要的东西的情况下是可以的。但是，我们的经验表明，除了极少数情况之外，人们并不知道他们想要的东西，直到看到它为止。许多现在人们认为理所当然的东西（手机、因特网、图形用户界面、短消息等），并不是来自于风险承担者的想像，而是来自于人们的创新，在它被发明之前并没有人要求这些东西。

所以，一个有创造力的公司，他们的工作并不仅仅是用限定的费用、按时给客户他们想要的东西，而是给他们从未梦想过的东西，当他得到的时候，他会认识到这是他一直想要的东西。

不要依赖于风险承担者准确的知道他们想要的东西，也不要依赖于他们对想要的东西提出要求，确实，构建许多产品是研究用户工作的结果，但如果这就是全部，那就没有改进什么东西，为了要前进一步，为了交付真正让客户满意的产品，就必须创新。

首先要考虑的是，仅仅把昨天的工作自动化，并不能让人发明一种更好的方法，把昨天的自动化进一步自动化同样也不能。只有重新思考业务用例，才能得到创新的产品，也能够在明天的市场竞争中取胜。这就是说，对于每个业务用例，必须理解本质，并确保风险承担者有类似的理解，当做到这一点的时候，创新就会接踵而来。

在创新问题上，我们来看看顾客的心理有什么变化。今天的顾客比以前知道的多，有办法得到以前得不到的消息，在因特网上可以找到最好的价格，并且在几乎所有的产品和服务之间作选择，地理位置不再重要，顾客可以在几乎世界上任何地方订购产品。顾客忠诚度曾经是传统公司的保命路线，现在正在消失，取而代之的顾客对服务和方便性的要求。因此，先产品的构建就必须考虑这个现实，产品是不是为组织或者顾客提供了更好的服务和更大的方便性呢？如果没有，所构建的产品就很可能很快地被抛弃掉。

如果我们要构建一个向顾客提供商品或者服务的产品，需要研究目前业务的问题是：它目前提供什么级别的服务和方便性？可以改进它吗？可以消除购买或者订购业务的一个步骤吗？可以替用户完成以前他自己完成的事情吗？注意，我们这里并不讨论设计，暂时忽略打算使用的技术，通过思考业务怎样做才能提供更好的服务和方便性，对工作的本质提出一些问题。

我们还可以考虑个客对他们的交易拥有多少控制权？人们有时候是喜欢，有时候是要求自己做一些工作，超市可以引导顾客自己扫描他们选购的商品，人们在家互动式的购物，在因特网上购买股票不需要中间交易商介入，旅行者自己订购旅馆和机票。让顾客参与给双方都带来了好处，商家减少了交易成本，消费者带来了更大的方便，并感觉对交易有控制权。对顾客的这些理解都可能影响到我们产品的创新。

顾客怎样和业务发生联系？人们酷爱联系，驾驶员冒着事故危险在行驶中接听电话，人们在路上行走眼睛只注意手机屏幕上的文字，手机忘在家几乎可能使人一天心神不宁，不管

什么年龄层次总是在深夜查看自己的 E-Mail，为什么？因为人们喜欢联系。所以，在创新的问题上是不是需要考虑人们的这个行为特点？顾客忠诚度卡、飞行里程计划、新闻短信、品牌消费卡等等，都是业务联系顾客的例子，这些东西是如何影响我们的创新的？

选择信息传播方式对创新的产品很重要，分析师需要和风险承担者一起工作，找到为用户或者顾客提供更好的消息的方式，提供更多更好的选择，人们还有一个先入为主的根深蒂固的特点，要研究如何先于自己的竞争者向顾客提供这些信息。

创新标志的现有的业务会发生重大重组，但产品不可能是完全创新的，它应该基于它支持的工作，但我们必须创造一种更好的方式来完成工作，也就是说，为顾客或用户提供更好的控制、更好的信息、更多的选择、更便捷的联系、更好的服务、更便捷的请求响应，以及可能为组织带来更低的成本以及更高的效益。

五、为需求制作原型

有时候分析师会遇到障碍，有些需求没有成型，甚至也没有办法向用户解释这些需求，或者分析师自己也不能理解它们，也许这个产品具有创新性，没有人真正知道需求是什么。在这个时候，原型就是最有效的方法了。

原型，是一个预期系统的小规模的，不完整的，但可以工作的示例。

所谓获取原型，是向用户提供一个快速而粗造的实现，以确定用户的业务需求。

原型可以使用开发工具快速的构造比如数据库系统，他可能包括错误检查、输入验证、安全等问题，原型往往表达的是系统最重要的部分或者是风险最大的部分。

原型的意图是向用户展示需求的一个模拟。但是，应该注意到，原型一般是在分析完成以后构造，如果没有定量的分析，你就不可能构造原型。有的时候，对于技术上的新框架，或者技术上的新方法，也可以构造原型并通过测试来验证想法。

有两种方法建立原型，一个高保真模型，其结果是工作软件的一部分。另一个是低保真模型，利用纸、白板，和自己熟悉的语言环境来构建。

六、面谈

在需求收集时间表中，面谈是一个重要的部分。过去，分析师仅仅与项目相关人员见面，询问他的需要，并希望以这种方式收集所有信息，遗憾的是，经验表明这种方法来发现需求是比较困难的。面谈有一些很重要而且很基本的步骤，这对掌握面谈的技巧是必须的：

1，提出问题

面谈从一组问题开始，并由此产生有关真正需求的多数知识，我们可以从服务请求、概念研究文档以及软件项目管理计划中，来提示并确立面谈问题的模版。

2，选择面谈者

多数情况下不可能与所有的相关人员面谈，因此必须选择其中的代表，他们应该有实际知识，比较易于接近，并且能够为问题提供可信的和可靠的答案，主要考虑一下人员：

初级人员：由于他们经验不多，并不能做出很多贡献，但他们可能有新鲜的观点，也可能有意想不到的信息。

中级相关人员：由于经验丰富，这些人员对领域操作性和技术性的问题，能够提供详细的理解，应该经常与项目领导进行面谈。

管理人员和其他特殊客户：CEO 有该领域的知识，并影响着项目的成功，他们无疑是面谈的重要对象，只要有可能，应该经常与执行发起人面谈。

理论家：如果有这样的相关人员，他们能够开阔你的视野。

系统典型“用户”：这样的相关人员是重要的，因为他们将花更多的时间与系统打交道。他们可能是革命精神的思想家，也可能对新系统抱有偏见。

“后起之秀”：这样的人将成为该领域的领导者，可以提供深刻的思想和信息。

人力资源的多样性，有助于验证所收集信息的可靠性。

3，计划联系方式

如果时间允许，请研究一下想要联系的人对项目的观点。每个人都很忙，所以对面谈的范围要有所控制，仅限于他们感兴趣的话题。

最初的联系方式是电话和电子邮件。

4，进行面谈

面谈可以通过电话、视频会议、MSN 或者 QQ，但最好的方式是通过面对面的谈话，能够回去信息的关键因素是，你和面谈之间建立了良好的氛围，一个舒适而且充满自信的环境。

1) 建立氛围

- 请求面谈这允许你在面谈期间记笔记。
- 诚恳的态度：对软件中的利害关系表现做出真正的兴趣。
- 倾听：完全关注对方，对方说话的时候，注意你的眼神不要漂移。
- 从小事入手，再进入敏感的事务。
- 直截了当，表现出诚实。
- 谦虚，没有人愿意和“无所不知”的人交谈。
- 不要跑题，除非你认为分享相关人员的经验很重要。
- 请求去看看将来要使用产品的环境。
- 提供未来的帮助和分项目报告，信息交换是互惠的。

2) 提出问题

问题应该简单、简洁，仅包括一个部分，复合的需求说明往往那个很难解释和测试。

3) 如果面谈中止

可能面谈者不能提供给你重要的信息，也要感谢他百忙之中参加面谈，应该留下联系电话，以便必要的时候可以联系你。有时面谈者不愿意给你提供信息，这时强调他的专门知识对系统的正面影响，可能会克服这个障碍。也可能需要临时改变方向，应该判断这个方向的意义，如果是有意义的，可能会提供意外的信息。

4) 结束会议

应该为下一次面谈打下基础：

- 询问面谈者是否有问题问你。
- 询问是否有其他应该提出的问题。
- 以未来问题或者兴趣点的方式，给你留下未来的目标。
- 询问你是否可以想到其他问题的时候和对方联系。
- 询问是否可以和其他人面谈。
- 联系资料来源，感谢他的参与。

七、需求项框架

每一个单独的需求都有一个结构，需求有一些组成部分，每部分都包含某方面的知识，每部分对于理解整个需求都是必要的。

由于希望和用户交谈的时候，找出一项需求的所有部分以后再找下一项的做法是不切合实际的，所以建议做一些小卡片，把需求框架打印在卡片上，在和用户交谈的时候，法线

一项就快速的记录下来，随着对需求理解的加深，逐步地完成这些卡片。在某一次访谈的时候可能谈得不是很清楚，卡片上的空白给我们提示了下一次访谈需要补充提出的问题。

这种做法给我们带来很大的灵活性，是我们在风险承担者那里收集需求的时候不至于带来很大的障碍。我们可以根据它所属的用例进行分组（见右上角的“事件/用例”编号），这样便于在桌面上查找某张卡片。

这样逐步地归纳整理不断完善，就可以形成一个很好的需求来，下面是这种卡片的模版。

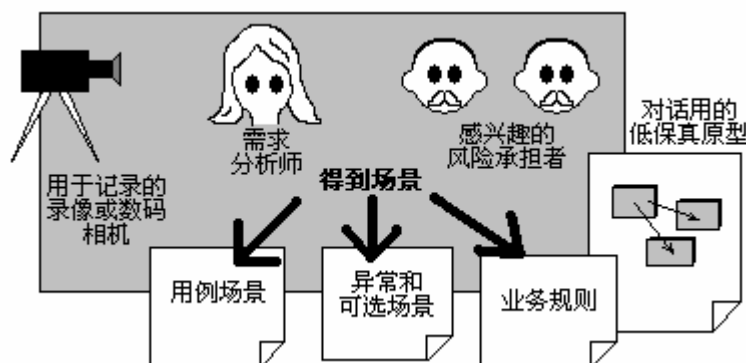
需求编号： 唯一 ID	需求类型： 模版小节编号	事件/用例编号： 需要这项需求的事件/用例清单
描述： 对需求意图的一句话概括		
理由： 这项需求的合理性		
来源： 提出这项需求的人		
验收标准： 这项需求的度量标准，从而可以测试解决方案是否满足了要求		
顾客满意度	顾客不满意度	冲突
如果这项需求成功实现，风险承担者的高兴程度（评分 1-5，1 为不感兴趣，5 为非常高兴）	如果这项需求没有成为座钟产品的一部分，风险承担者的不高兴程度（评分 1-5，1 为无所谓，5 为非常不高兴）	予这项需求冲突的需求
优先级： 顾客价值评分		
支持材料： 支出说明和解释这项需求的文档		
历史： 创建和变更情况		

八、业务用例研讨会

我们发现业务用例研讨会对大多数项目都是有用的，这些研讨会会让需求分析师与数量较少的、专门化的风险承担者一起工作。

我们已经讨论过，业务事件触发的响应，称之为业务用例，这种业务用例清楚的界定了根据对外界的响应的工作划分。在完成这种划分以后，需要做的事情就是描述和重新制定完成的工作，或期望完成的工作。需求分析师的工作是记录下这些活动，然后通过它们导出能完成工作提供最大帮助的产品需求。

每个业务事件都有一个或多个风险承担者，他们是这部分工作的专家，他们对这个业务用例的输出有特别的兴趣。研讨会的目的，就是通过这些专家把业务用例相关的知识传授给需求分析师。这些人把自己锁起来，召开类似 JAD（联合应用开发，后面会讨论）风格的研讨会，构建一些场景展示队业务事件做出正确相应所需要的动作。在理想情况下，感兴趣的风险承担者能够有效的沟通，描述他们对工作的理解，问问题，并给出对工作的期望。



后面我们会讨论用例的场景，也就是以一种结构化方式来描述业务用例的故事，我们建议在讨论的时候，把业务用例的场景作为谈话的中心，但在现在阶段，只需要把业务用例的

功能细分为一些合理的步骤就可以了。

会议中以场景作为手段，分析师与风险承担者一起工作，获取业务用例的知识：

- 业务用例的预测成果。
- 描述通过业务用例完成工作的一些场景。
- 异常场景描述了哪些事情可能出错，以及工作通过哪些活动来纠正它。
- 可选场景，展示工作允许的变化。
- 适用于该业务用例的业务规则，业务规则是管理的规定，可以在感兴趣的风险承担者的指导下，发现这些规则的文件，这些规则最终是业务用例和需求的一部分。
- 基本的产品用例，展示业务用例的多少部分由产品完成，在这个阶段，建立一个产品的轮廓是有好处的。
- 为这个业务用例构建的产品的可能用户特征。
- 低保真原型，用来帮助风险承担者把业务用例可视化，这个可以抛弃的原型并不打算在需求阶段结束后继续保存。

九、创造性研讨会

创造性研讨会是项目驱动力的一种自然的发展，目的是得到前面所讨论的创新产品，如果有大量的风险承担者将参与这个创新过程，可以使用这个方法。如果希望让风险承担者看到开发创新产品，而不只是重新创建一个同样的老系统的好处，也可以采用创造性研讨会。

越来越多的人认识到，在需求活动的早期进行创造性思考是一种有利的方式。当风险承担者们把自己从现状中释放出来，他们可以提出新的、从未梦想过的想法。

例如，当某空中管制中心决定调查未来空中交通控制系统的需求时，该机构发现难以超越目前的工作方式。于是他们与一家著名的咨询机构一起，举行了创造性研讨会，鼓励空中交通控制人员、飞行员、航空公司代表和系统开发者一起来思考将来的新需求。研讨会的成果是得到了几百条创新的需求，与会者都认为研讨会得到的需求对最终新的空中交通控制产品是重要的，几乎有令人吃惊的影响。他们也一致认为，如果不是通过创造性研讨会，这些需求可能永远难以实现。

举行创造性研讨会的最佳时间，是在接近项目开始时，在人们开始把他们的思维关注于解决方案之前。同时，我们需要有某种结构来约束这些新想法，否则事情将变得不切实际，与手头的问题相去甚远，下面是一些计划和举行创造性研讨会的技巧：

- 设定调查的范围，确定项目的目标，进行首次风险承担者分析；
- 设定调查的范围，利用业务事件来得到业务用例；
- 为研讨会制定计划，使用各种创新技巧，帮助人员提出和发现新的想法；
- 举行研讨会，以业务事件为中心把这些新思想串联起来。
- 在研讨会之后，对结果进行汇总，并反馈给研讨会的参与者。

创造性技巧的目的是鼓励人们进行创新。例如，一种方法是让不相关领域的一个专家向小组介绍他的工作，小组然后利用来自不相关领域的分析，触发对他们自己领域进行研究的想法。以空中交通控制系统为例，他们听取了一名电影编辑、一名音乐家、一名花纹设计师和一名厨师的介绍，人们发现，厨师对于配料的谈话是对控制交通控制系统有用的（当然中午也享受了他的手艺）。而电影编剧利用某电影的例子，解释了剧本是怎样作为设计和拍摄场景的基础的，她向我们展示了导演如何为剧本添加思想，并使故事板作为捕捉新想法的手段。按照这个演示，风险承担者们（大部分是空中交通控制人员）利用故事板技术作为启发和思想的来源，在他们自己的领域为项目创建了大型故事板。

其它的创造性技术包括把看起来无关的一些想法组合在一起，创造一个新想法。例如有

意义的例子是，把电话和照相机组合在一起，或者是把带有 GPS 功能的电话与语音驾驶行走向导组合在一起等，都创造了大量的新产品和新概念。

转换性思考是把一个想法应用到另外的领域，例如，把传统的拍卖销售转换为基于 Web 的拍卖销售。还有一个方法是消除约束，通过消除约束来探讨得到新的、不同的、也许有用的思想的可能性。

为空中交通控制系统举行的创造性研讨会所得到的思想，是由其它行业的思想和实践触发的，在听取专家的报告是受到启发，利用故事板作为产生新想法的手段，把所有这些想法作为跳板，实现对原来熟悉的、显而易见的工作方式的超越。

在创造性研讨会中，许多技巧都利用了头脑风暴会议的技巧。

十、头脑风暴会议

很多资料都谈到了头脑风暴这个技术，事实上头脑风暴是一个会议技术，通过收集会议成员的即兴想法，为具体问题找到解决方案。这种技术的立论依据是：“驳倒轻率的思想容易，提出新的设想难，不管这些想法多么离奇，也不应该一直不理睬它，不加以考虑和研究。”

一个死板教条的组织往往习惯于领导做指示，下属按部就班的执行，事实证明，这样的组织是没有生命力的，也是没有创造力的。

头脑风暴采用团队工作来产生想法和解决问题，这是一种群体决策技术，它的要点是，不进行评价或者解释说明，而是快速产生设想，任何设想都可以：离奇的、疯狂的或者不切实际的，设想产生后，再进行评价，而且不断地进行评价。

从需求的角度，前期生成的需求越多，后期发生的不愉快就越少，即使不能一次性的完成这些需求，在将来系统升级或者在设计中预留升级接口都是非常有意义的。从许多需求中选择，远比从头开始要好，这样一来，我们就可以从长长的清单中选择、分类、排序和删减。

Alex Osborn 指出：在一个群体中，人们想出设想，一个设想可能以另一个设想为基础，要欢迎所有的设想，不指责任何设想，更多数量的原创设想会产生更多数量的有用设想，单独一个人很少会有这样多的创造性设想。当减少限制以后，人们将更愿意提出有用的设想。

这种跳出思维框框的原创性技术是革命性的，头脑风暴现在几乎被世界上所有的大公司所采用，它已经成为“创造性设想”的同义词。头脑风暴使团队成员从社会角色的限制中解脱出来，因而能够产生尽量多的设想，鼓励所有的参与者说出任何设想，不管它显得多么“愚笨”都不会受到指责，因为参与者的任务是提出设想，而不是检查它们。

1) 头脑风暴会议的准备

- 确定并邀请参与者、记录员和会议领导。发布有关地点、时间、位置和会议预期长度等信息。一般预期的时间大约 2 个小时。
- 确定具体要开发的软件系统或者子系统，发布合适的文档（如概念研究总结，面谈总结，项目计划目标）给所有的团队成员。
- 定义头脑风暴会议的规则，发布给参与者。
- 保证室内有舒适的椅子，桌子，白板，投影仪，便签以及食物和饮料等，还需要给记录员准备数码相机，以记录白板上的实时讨论信息。

2) 领导规则

- 允许设想以口头形式（首选）和书面形式表达。
- 适当的时候允许沉默。
- 从所有参与者那里尽可能多地收集设想，产生设想的时候，不要指责或者判断。
- 说明会议程序。
- 不允许互相之间指责、讨论或者争论。

- 不允许嘲笑，也不要过于严肃。
- 鼓励异想天开或者夸张的设想。
- 确立目标和主题。
- 保持快速的节奏，以减少限制和评价。
- 帮助参与者变换和组合设想。
- 帮助参与者产生设想。
- 设置时间限制。
- 充分利用所分配的时间。

3) 记录员规则

- 不要具体描述设想，只需要捕捉他的精华。
- 如果需要，可以简单解释。
- 采用说话者的原话。
- 用简短的单词或者短语，记录下所陈述的设想。
- 必要的时候可以用数码相机记录白板上即时所画的图形。

4) 参与者规则

- 倾听并理解别人的设想，“站在别人的肩膀上”产生新的设想。
- 完全投入到过程中，并自由思考，充分展开想象的翅膀。
- 耐心，必要的时候保持沉默。
- 一次只提出一个设想。
- 不要怕麻烦，要提供解释，语言要简短干练。
- 不要评价他人的设想，不要指责和判断。
- 不要发出一些可能被别人理解为指责的暗示性语言。
- 不要打断别人，轮流发言。
- 不要太重视任何一种设想。
- 产生尽可能多的设想，短时间内，先数量后质量。
- 每一设想都要保持简洁，不能用简洁的语言表达的，往往是没有思考清楚的。
- 写下你的设想，并交给记录员。

5) 头脑风暴总体规则

- 欢迎所有的设想，没有错误的、愚蠢的、麻木的或者离题太远的设想。
- 提出的每一个设想都属于群体，而不是属于提出设想的人。
- 每个设想都有价值。
- 会议中要有适当的人数。
- 在头脑风暴完成之后再讨论设想。

6) 头脑风暴会议的后续工作

一般来说，会议完成之后又领导来编辑设想，建议加入一两个熟悉所开发应用程序的人。首先，建立一个设想清单，如果会议团队有时间，可以提供简短的匿名建议设想清单。

然后，去掉无用的和不完整的的设想，注意，不是扔掉，文档中是有记录的，只是从考虑问题的清单中去掉。采用有效的准则来删除设想，不要太轻易的删除，但如果是重复的，则要删除它。

把剩下的设想分成3类：

有效的：看起来是需要的，可测试的需求。

可能的：可以是需求。

似乎不是：可能不是有效的需求。

记录发现的需求，以输入到软件需求规范里面去，并根据重要性，依赖性以及风险进行

综合排序，这对于描述完整的需求是十分重要的。

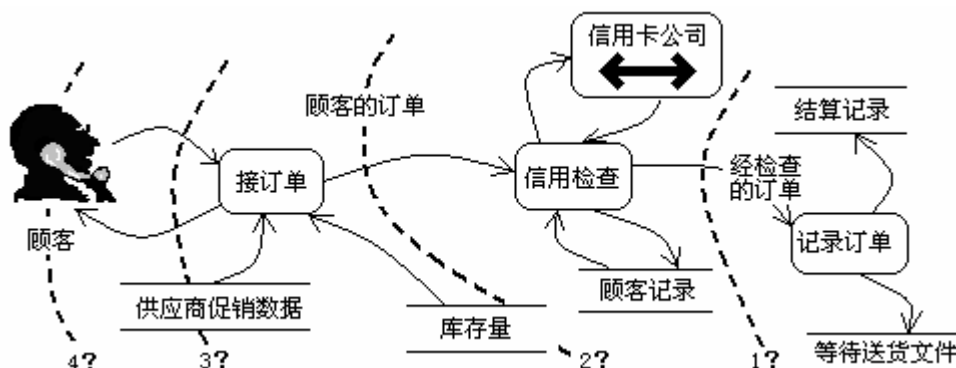
十一、确定产品应该是怎样的

当对工作有了比较深入而且条理化的理解之后，就可以思考“产品应该是怎样的？”这样一个问题了。遗憾的是许多项目开始的时候都有关于“产品应该是什么”的先入为主的概念，但不理解产品将成为其工作的一部分。这里我们看看为了发现优化的产品，我们可以做些什么？

对需求分析师来说，没有说出来但很重要的一项任务，就是确定工作将来应该是怎样的，以及产品是怎样才能对工作产生最大的帮助。产品是工作的一部分，工作是打算以某种程度改变的东西（通常是把它自动化），目标是找到优化的业务用例。因为业务用例通常是工作对外界服务请求的响应，所以，优化的响应就是以最低的时间、原材料或工作量成本（从组织的角度来说），提供最有价值的服务（从顾客的角度来说）。

需求分析师的任务是找到最佳的业务用例，最佳的业务用例总是最接近工作本质的那一个，有时候还包括一些创新。当对业务工作非常了解之后，就要决定工作的多少部分将由产品来完成。

重要的是先理解工作，然后把工作的一部分自动化，我们才能把自动化产品无缝的放到工作中去。让我们来看一个如何确定工作的多少部分将成为产品的例子。考虑下图的例子，一个顾客（相邻系统）正通过电话订购某种商品。那么，产品边界的最佳位置在哪里？或者换一种思考方式，怎样才算是最有用的产品？



采用 1 号边界的产品：一个操作员输入已经检查过的订单，通过信用卡公司授权以后，产品记录订单，这并不是一个好产品，因为大部分工作留给了操作员。

那么 2 号边界又如何？操作员通过电话接下订单，把它录入产品，其余的都可以自动化完成。这个方法不坏，但还可以做的更好。

3 号边界把订单接收自动化，可以是一个语音识别系统，也可以是通过按键处理的系统，也可以通过 Web 方式处理，通过网站在线方式订购。如果顾客没有问题或者不需要太复杂的方式响应，对销售公司来说可能是个好方案，但对于客户呢？他真正想要的是什么呢？他喜欢与人对话还是语音发生器对话？主要销售对象群体对 Web 方式接受吗？销售公司可以提供哪些服务来保持他的忠诚度？

我们在讨论业务事件的时候，应该寻找事件的真正起源，可以肯定地说，起源不在操作者那里，操作者只是业务响应的一部分，它是在相邻系统作了某些事情之后发生的。在上图的例子中，起源是客户发现缺少某种商品，他发现需要订购些什么，他曾经寻找了家里有没有这个商品，或者清点相关商品的数目，当他决定购买的时候他拿起了电话。所以，这个事件的起源在于他拿起电话前的大约 30 分钟。

从顾客的角度，能不能使这个事情变得更方便呢？如果销售公司知道这个顾客已有商品

的数目，并且知道他的消耗数目，顾客就根本不需要打电话了，公司会打电话给顾客通知他这种商品他快要消耗完了，同时安排恰当的时间送货。这个场景是把产品的边界扩展到了相邻系统思维的深处，从方便和服务的角度来说（站在顾客的角度），这是不是一个更好的想法呢？从提供服务来保持顾客的忠诚度来说（站在销售公司的角度），这是不是也是更好的想法呢？如果是，那我们的业务会做些什么样的改变呢？继续深入的思维和研究，可能会创造出更好的产品。

所以，我们应该通过某个工作事件的进一步思考，来设想产品应该是怎样的。检查业务事件，特别是那些由人发起的事件，当事件发生的时候，相邻系统都在做什么？可以扩展产品的范围以包括这些活动吗？

让我们考察一下某航空公司头等舱签入的过程，过去一直沿用的方法，是在乘客自己办理签入，背着包进入机场头等舱休息室后，头等场的服务即被触发开始。后来他们改变了做法，头等舱旅客由一辆豪华轿车送入机场，乘客不需要走入签入口，司机打电话通知该公司设置的直接签入口，报告要签入的行李数，乘客坐在车内就可以办理签入并拿到登机牌，并被送到头等舱休息室的入口。整个业务的变化在于，该航空公司认为，乘客离开家的时候用例就被触发了，而不是到达休息室门口。读顾客来说，这就提供了更好与更方便的服务，而公司得以维持了这些重要顾客的忠诚度。

我们的结论是，努力找到业务用例的真正起源，考虑自己的业务用例，他真的是在工作的边界上发生的吗？还是在它们到达组织之前就开始了？这些思考的结果，会对业务以至于产品的设计产生重大影响。

2.5 业务事件和业务用例

事实上，需求分析首先必须分析业务，而现实业务的特点本身就是过程的集合，所以，业务用例分析还是比较倾向于使用传统的面向过程需求分析，也就是把业务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，这样就构成了业务用例分析的基础。

一、业务需求分析阶段

在问题定义清楚以后，就可以进入业务需求分析阶段了。需求分析是给一个系统定义业务需求，这恐怕是一个开发过程中作重要的任务了。

注意，需求分析关注的是“什么”，而不是“如何”。需求分析要做的事情是：

定义需求：包括功能性，以及非功能性需求。

排列需求的优先次序；

修改项目计划。

交流需求陈述。

整个需求分析过程都需要用户的参与认可。需求过程真正重要的事情不是画图，而是写文档，要尽可能清晰的把需求定义清楚，并且请用户提出意见。

过程模型研究的问题范围主要是建立业务用例，下面我们先简单讨论一下面向过程分析的特点，它的特征如下：

1) 采用“抽象”和“分解”两个基本手段，用抽象模型的概念，按照具体业务内部数据传递、变换关系，由顶向下逐层分解，直到找到满足需要的所有可实现的业务元素为止。

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

二、做学徒是获取业务用例的好方法

事实上，获取业务用例最好的方法就是做学徒，在实际中观察业务是如何工作的。

不太可能许多用户能给你讲清楚他们在做什么，使开发者完全理解业务工作，并且捕做到他们所需要的需求。当某人给你解释一个业务的时候，他倾向于用行业内专用的抽象术语来解释，时间会非常短，也没有办法或者耐心演示一切。

这就需要分析这参加到对方工作中去，几乎所有人都善于把自己正在做的事情给你解释清楚。当用户描述自己的工作的時候，你不要叫他离开工作，而是坐在他的旁边，随着工作的进行，每一个步骤都给你解释清楚，你可以问他：“为什么要这么做？”“这是什么意思？”

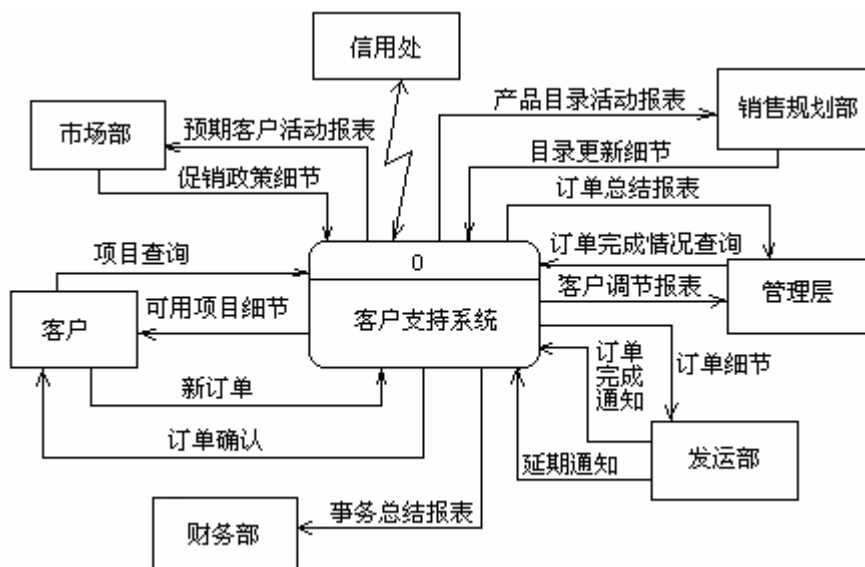
这种类似的师徒关系可以和当前的建模结合在一起，随着你对工作的观察和解释的展开，你就慢慢的勾勒出了业务的模型，以及他们是如何与其它的业务相连的。模型建立好了以后，你可以把它反馈到用户那里求得确认，并且对未肯定的地方提出问题。

分析师也可以利用这种师徒关系试验他的需求和设计思想，分析师可以向用户提出一个想法是不是可行？它是否会改进工作？这种沟通让用户觉得分析师不仅仅是观察着，而且慢慢的变成了个业务专家，这种尊重的氛围对于进一步建立分析模型是至关重要的。

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

三、业务的上下文范围与图

在确定范围的基础上，我们可以画出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顶层关联图简单例子，其中箭头表示数据的流向。



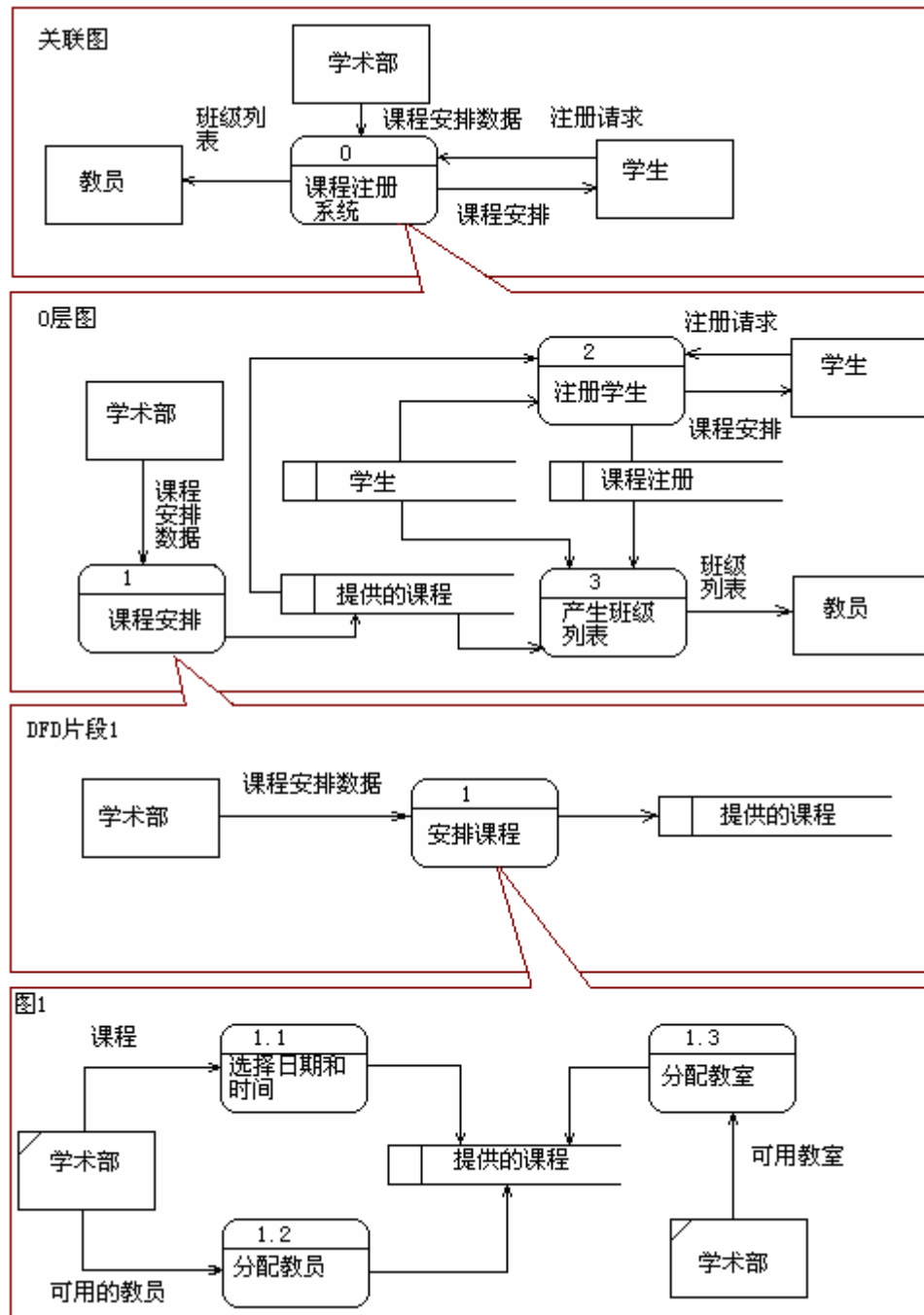
四、过程分解

过程分解是以一种事先定义的方式来规范工作流程的方法，我们经常在定义一个工作的时候采用过程分解的方法，比如在建立需求工程的时候，我们使用了典型的过程分解方法，使过程变得清晰明了，而且易于操作。同样道理，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。

过程分解实际上是一种工作划分方法，不过，过程分解一般是一个事先方法，当过程定义清楚一后，所有的参与者都必须遵循这个过程，当我们需要了解一个工作的时候，如果这个工作存在一个已经定义的过程，仔细阅读这个过程是非常有意义的，它可以使我们对整个工作方式有一个概貌的、框架性的理解。

如果不存在这个过程，则我们需要分析整个工作状况，描绘出整个工作的过程分解结构，并取得风险承担者的认同。这是理解工作的第一步，也是后续工作的基础。



但是，仅仅靠过程分解是不够的，因为这时候的过程研究还是属于静态的、理解性的、事实上可能还是事先的，甚至是主观的。对工作的进一步理解和分析，还需要进行动态的、更加客观的、更加深入的分析，这就要考虑利用事件及其业务响应来建立过程模型。

五、业务事件

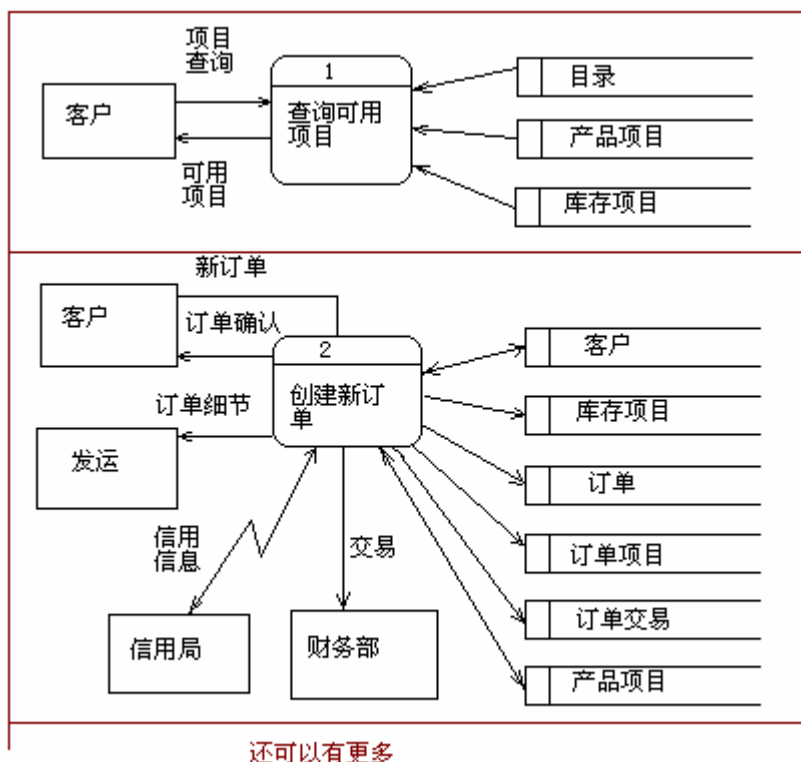
所有的系统或者工作都会对外部发生的事件做出响应，这些事件称为“业务事件”。例如顾客从书架上取出一本书，然后走到收银台，这样发生一个业务事件：“顾客想买一本书”。店员响应这个事件的做法是：扫描条形码，告知书价，向信用卡公司查询并要求认证，把款项记入收银机，把书放入袋子并递给您。

我们把对业务事件的响应称为“业务用例”。

在上面的例子中，向信用卡公司查询并要求认证事实上又发起了另一个事件，并且产生了另一个业务用例，也就是由事件把业务用例激活。

上面两个例子业务事件的产生是随机的，但也有时间触发的业务事件，比如事先确定的一个日期所发起，比如保险公司在某人的保单一周年的时候给他寄一份续保通知，银行每个月15号给他寄一份结算表等。通常对一个时间触发的业务响应是产生某种信息并发送到相邻系统，而且总是牵涉到存储数据。

DFD 的细节称作片段，以事件为基础的片断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。



在这个过程中，关键是要发现业务事件，这些事件让业务以某种方式做出反应，业务用例的每一层，都是一个事件响应的结果。

六、为什么业务事件和业务用例是好想法

对于许多一眼看起来相当明显的事情，我们在描述的时候往往会发生麻烦。但是我们的经验告诉我们，以客观的方式划分工作的价值，以及在规划解决方案之前先理解工作本身的价值，结果就会使我们发现真正的需求，而且会更快地发现它们。

事件是一种非主观方式来划分工作的，也就是确定工作对外界的响应。毕竟这是顾客看待业务的方式。无论在过程分解中内部的划分是怎么样的，外界都不感兴趣。类似的，任何工作当前的划分（比如过程分解），都是基于当初在具体情况下策略上的考虑，这些决定在新的自动化系统加入的时候可能不再有效，至少应该质疑它们，避免只是因为它们存在，就认为理应如此。不是从内部看，而是从外部看，这样我们就可以清楚地发现划分工作的最有效方式。

业务事件指出了哪些东西是在一起配合工作的，从而我们可以提交一些内聚的部分，并且使这些部分的接口最小化。同时，这也可以使工作的这一部分成为详细需求调研的基础。这些部分的依赖性越少，分析师就越有可能关注与这部分的细节，而不需要知道其它部分。

但是，使用也无用例最重要的原因，是研究业务事件到来的时候到底发生了什么。我们不能假定系统已经存在，再围绕着这个假定系统来研究业务，这非常危险，因为一开始就以产品为中心来研究问题，可能会忽略掉许多深入工作的其它部分，从而丧失了改进产品和业务的机会。

如果分析师不敢去问“为什么是这样的？”、“有没有改进的机会？”就可能使“业务化石”从一代产品传到下一代产品。

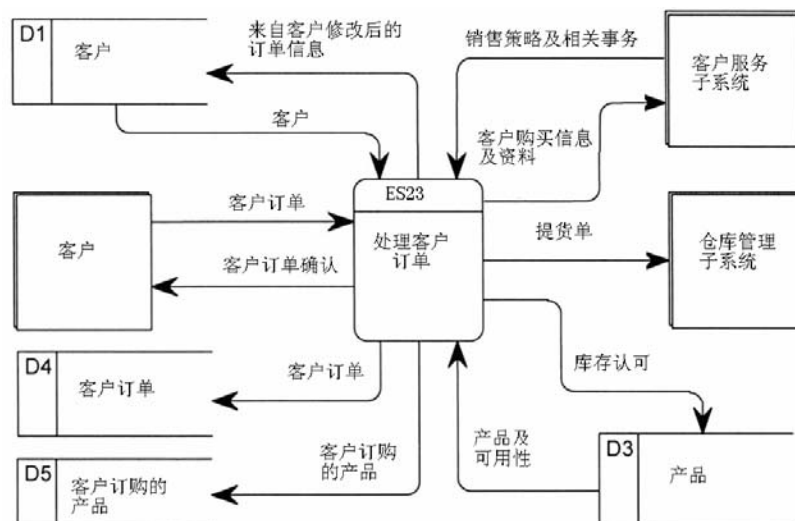
分析师必须超越那些显然的东西，这意味着理解工作真正的本质，当然如果太沉溺当前的工作方式也是危险的，有时候就难以发现业务事件，为了克服这种困难，我们建议通过一个非正式的过程来发现业务事件。

七、发现业务事件

业务事件是发生的一些事情，这些事情让工作作出某种反应，他们发生在工作的范围之外，或者是某个时间到达的时候。下面我们以一个订单处理子系统业务模型来讨论业务事件的发现问题，问题陈述如下：

项目名称：电源设备订单处理子系统	
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：由客户服务子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折扣方式应折，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。

这个系统的上下文图如下。



一般来说，在上下文范围图中每个数据流都与一个业务事件相联系，现在我们的眼睛盯住具体的细节，来发现每个事件。在研究事件的时候，往往需要了解所有的数据存储，但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。

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（输入）	响应（输出）
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”。

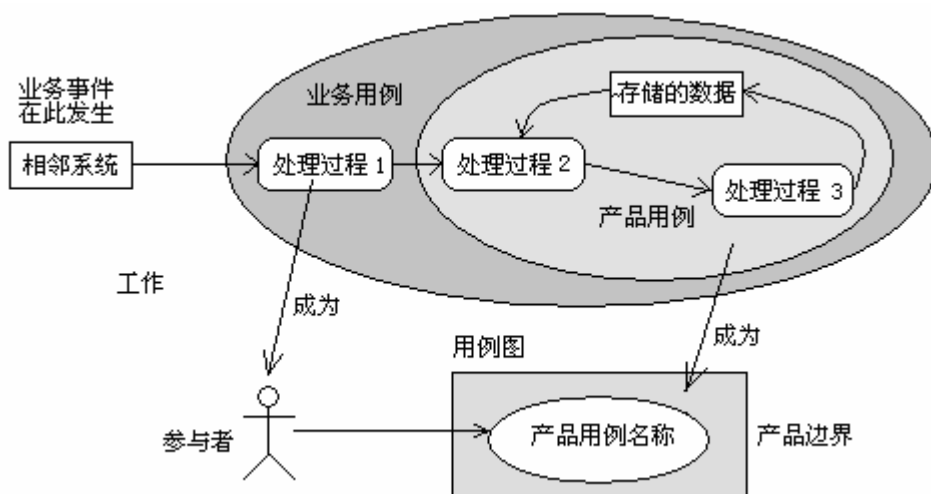
不可否认，确实需要一些工作知识才可能弄清楚这些业务事件，我们建议在项目的启动阶段就开始确定这些业务事件的过程，那时候风险承担者都在，他们都非常熟悉这些业务事件。但是需求分析师的兴趣在于工作是事件的响应，这就是对业务用例的分析。

第三章 产品用例与关注点捕获

3.1 产品用例的分析与场景

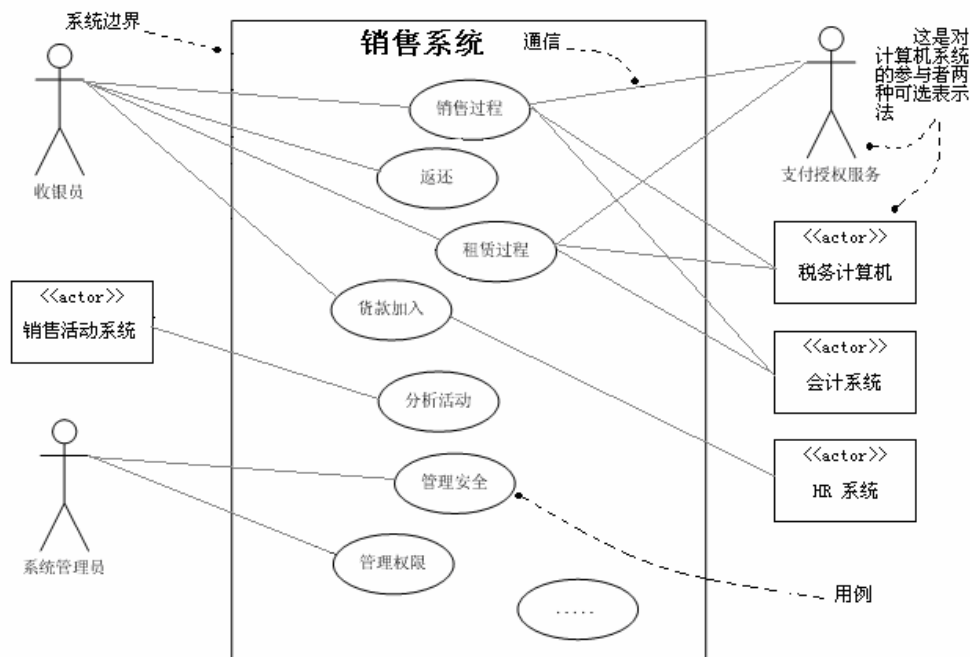
所谓产品用例，指的是站在产品的角度，仔细的研究和分析客户业务如何与产品融合，我们可以用用例和用例图来表达产品用例，这也是需求分析的重要部分。

作为分析过程，一般是在前述业务用例的基础上，进一步细化为产品用例，也就是说，在考察大范围业务的基础之上，进一步向产品需求集中，在某种程度上，它是由技术上的考虑集中的，所以，产品用例必须基于最初的业务事件和业务用例。



一、用例图对产品功能的表达

用例图主要表现各个用例之间宽泛的关系，而且主要表达技术层面的事情，或者称之为产品用例。如下图所示。



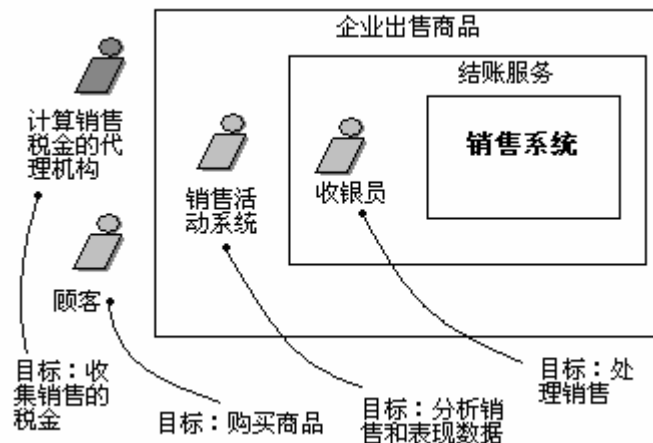
在上面的图中，计算机系统的参与者有两种表示法，其中有些人喜欢用不同于人型的方框表示外部计算机系统参与者，<<actor>>称作 UML 构造型，这是用某种方式分类元素的方式够造型的名称被书名符号包住，这本来就是法文印刷中表示引用的符号。

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。

3.2 用例的场景

用例的一个场景（说故事），用来研究一部分工作的分步骤情节，而这个情节或者情景必须用规定的格式来描述。由于场景是一个中性的媒体，所有人都能够理解，业务分析师用它来对工作所要做的事情取得一致意见，在取得一致意见以后，风险承担者将决定多少工作由产品来完成，然后产生一个参与者与产品交互的场景。

这种场景的描述，也可以理解为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。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。

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）。

二、场景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:

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:		

对于上述一些事件流, 以及一些关键和重要的问题, 需要对用例进行详细分析, 这是需要使用文档而不是图。

3.3 用例场景描述与结构

一, 用例事件流及其描述

系统所需的功能与行为, 可以通过一个或者多个用例来描述, 它是系统所有可能用途的总合。对于较大的系统, 可以把用例模型划分到不同的用例包里面去, 每个包包含针对一组参与者的一组用例, 形成一个个的功能区。

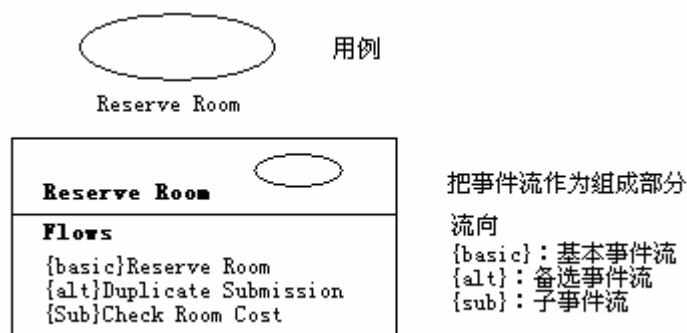
用例的场景是利用事件流, 以此可以实现用例的实例, 用例事件流的描述文档前面已经讨论过, 当文本不能很好的描述的时候, 可以使用活动图或者流程图来表达。

对于某些交互序列重复发生的事件流, 可以把这些部分独立描述, 定义成子事件流, 而子事件流的引用, 可以使用超级链接等实现技术来完成它。

设想一个“酒店管理系统”的简单例子, 我们描述一下场景(事件流)的文档, 主要表达的是对子事件流的引用。

用例名:	预定房间	用例类型
用例 ID:		
主要业务参与者:	客户	
描述:	该用例描述客户如何预定一个房间	
前置条件:	客户已经登录到系统	
后置条件:	成功预定之后，创建一个新的一项记录，特定时间的可用房间数将减少，如果预定失败，数据库不会发生任何改变。	
基本事件流:	1， 客户选择预定一个房间。 2， 系统显示酒店拥有的房间类型，以及它们的收费标准。 3， 客户核对房间的收费（S1）。 4， 客户对选择的房间提出预定。 5， 系统在数据库减少这个类型可预定房间的数目。 6， 系统基于提供的详细资料创建一个新的预定。 7， 系统显示预定确认号以及入住登记说明。 8， 用例结束。	
备选事件流:	A1. 重复预定 如果第 5 步存在相同的预定（同样的名字、e-mail、入住时间、离开时间）则系统显示原有的预定，并询问客户是否进行新的预定。 1， 如果客户想继续，则系统开始新的预定，用力重新开始。 2， 如果用户说明预定是重复的，用例结束。 A2.....	
子事件流:	S1， 核定房间收费。 1， 客户选择需要的房间类型，并说明将停留的时间。 2， 系统根据给出的周期，计算出总的费用并告知用户。	
特别需求:	系统必须能处理 5 个并发预定，每个预定所花时间不超过 20 秒。	

这种用例文档也可以用可视化的方式表达，比如下面的方式。



一般里说，如果需要概要的讨论系统中的用例，在抽象级别展现系统功能的关系，可以使用椭圆符号，这也是 UML 的标准符号。如果需要展现一个或者几个事件流的细节，比如在需求讨论会上，类元的标识符号更加合理，尽管它不是 UML 标准符号，但实践中非常有用。

在用例分析的时候，还需要避免在基础用例上分解出更小的用例，这称为“功能分解”，这是要避免的，因为一般情况过细的分解并不能向涉众提供真实的价值，而且很可能掉进分解的陷阱。如果发现有 200 个用例，一般已经掉进了功能分解的陷阱。所以，用例分解的时候一定要参考用户需要获取什么价值。

二，用例结构化

用例通过扩展、包含、泛化等关系作为对关注点之间进行建模的手段，称之为用例的结构化，比如如下图所示的情况，下面讨论这种结构化用例场景的描述规则。

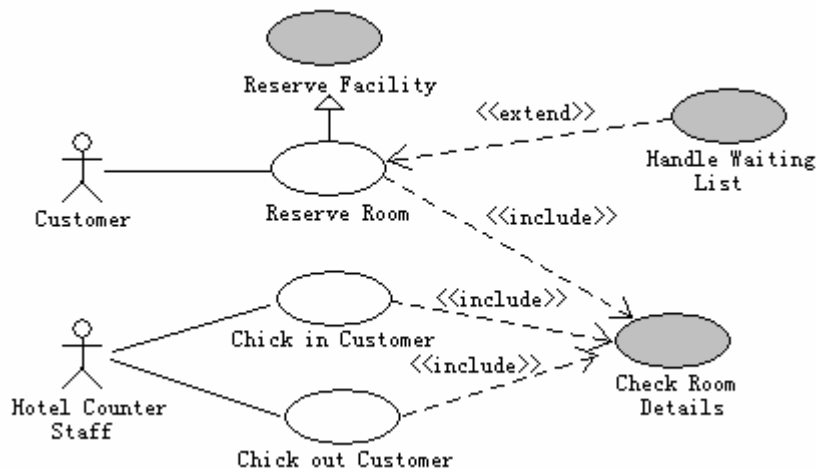
在用例建模的时候，不同用例之间的关系如下：

包含（include）：一个用例的实现使用另一个用例的实现。其图形表示方法为在用例图上用一条从基本用例指向包含用例的虚箭线表示，并在线上标注构造型<<include>>：

泛化（generalization）：一个用例的实现从另一个抽象的用例继承。

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

它们之间存在着扩展关系。如果特定的条件发生，扩展用例的行为才能执行。其图形表示同样用虚箭线表示，并在线上标注构造型<<extend>>：



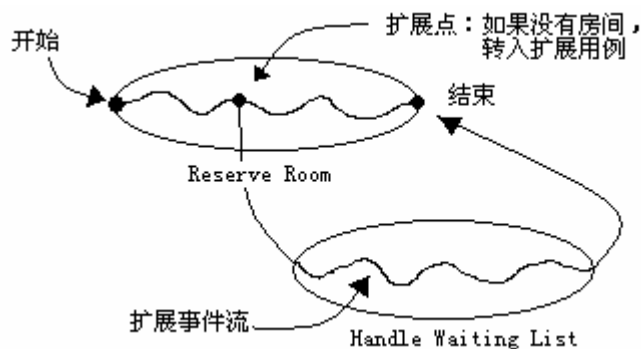
三，用例扩展关系

扩展用例包括一个或者多个扩展事件流，扩展事件流与备选事件流很相似，只不过它是在另一个用例中添加行为，例如上图情况：

用例名:	预定房间	用例类型
用例 ID:		
主要业务参与者:	客户	
描述:	该用例描述客户如何预定一个房间	
前置条件:	客户已经登录到系统	
后置条件:	成功预定之后，创建一个新的一项记录，特定时间的可用房间数将减少，如果预定失败，数据库不会发生任何改变。	
基本事件流:	1，客户选择预定一个房间。 2，系统显示酒店拥有的房间类型，以及它们的收费标准。 3，客户核对房间的收费（S1）。 4，客户对选择的房间提出预定。 5，系统在数据库减少这个类型可预定房间的数目（E1）。 6，系统基于提供的详细资料创建一个新的预定。 7，系统显示预定确认号以及入住登记说明。 8，用例结束。	
备选事件流:	
扩展点	E1：更新房间可用性 5.1 扩展点发生在基本流的第 5 步	

用例名:	处理等候列表	用例类型
用例 ID:		
基本事件流:	
扩展事件流:	EF1: 房间预定队列。 当没有客户所选择的房间的时候, 该扩展事件流则发生于用例“预定房间”的扩展点“更新房间可用性”上。 1, 创建一个等候预定, 并根据所选择的房间类型生成一个唯一标识符。 2, 把等候预定放入一个等候列表中。 3, 显示这个等候预定的唯一标识符。 4, 基用例结束。	

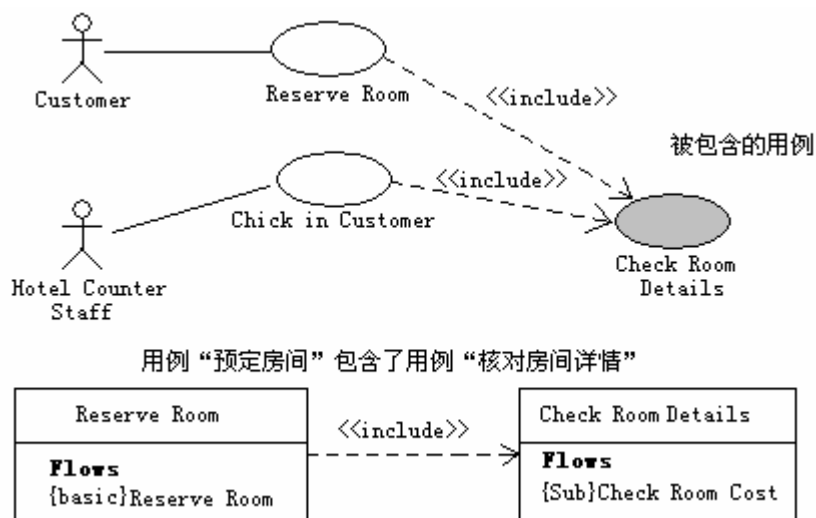
下图表示了这个扩展用例的存在发生了什么。



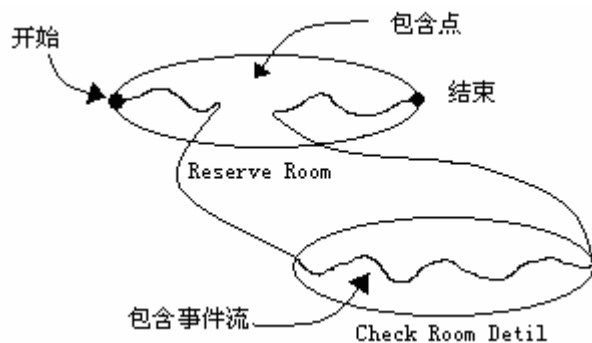
四, 用例包含关系

对于不同用例的相似步骤, 可以把这些公共事件抽取出来, 放在被包含的用例中, 其它用例可以引用这些被包含用例中的事件流。

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



{basic}标签表示这个事件可以由参与者启动, {sub}标签表示这个事件流只能被其它事件流包含或者引用, 它的执行过程如下。

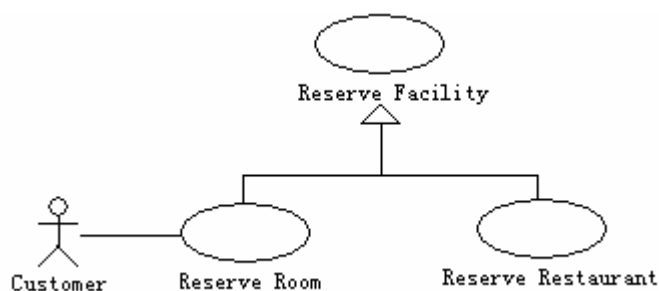


注意，包含并不是调用，而是在同一级别插入并展开。

五、用例泛化

泛化与类的泛化意义相同，表达用例之间存在“is-a-kind-of”关系。当一组用例拥有相同的事件流序列，或者相似的一组约束的时候，可以使用用例的泛化。

一般父用例是抽象的，而子用例将继承父用例的特性。比如“预定房间”或者“预定早餐”还可以有一套相同的其它服务的时候，可以使用泛化描述。



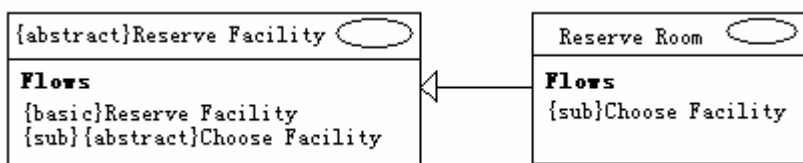
事件流的描述如下：

用例名:	预定服务	用例类型
用例 ID:		
基本事件流:	该用例开始于某个客户预定一个服务 1, 系统显示可提供的服务。 2, 客户 <u>选择一项服务</u> 。 3, 系统显示所选服务的总费用。 4, 系统在数据库中减去相应服务的总数量。 5, 系统为所选服务创建一个新的预定。 6, 系统显示预定确定号。 7, 用例结束。	
扩展事件流:	
子事件流:	{abstract} 选择服务	

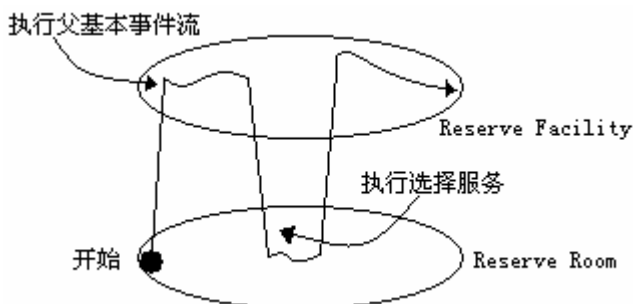
用例名:	预定房间	用例类型
用例 ID:		
基本事件流:		
子事件流:	S1, 选择服务 1, 客户选择预定房间。 2, 客户选择房间类型并说明停留时间。 3, 系统根据给出的周期计算出总的费用。	

在“预定服务”用例中有一个抽象子事件流，所以“预定房间”是一个抽象用例，而“选

择服务”的事件实现在“预定房间”用例中，用类表示的用例如下。



事件流执行的规则如下：实例化首先出现在子用例中，沿着基本事件流运行，如果子用例没有定义基本事件流，则沿着父基本事件流执行，上面的例子由于没有定义子基本事件流，所以沿着父基本事件流运行。在运行到“选择一项服务”的时候，执行子用例定义的“选择服务”子用例，如下图所示。



要注意到的问题是，首先不要混淆**泛化和扩展**，Java 中的关键字 `extend` 是泛化而非扩展，而用例中的扩展与泛化是完全不同的，扩展事件流只不过是挂接到原有用例事件流的某个位置上，以添加新的行为，而且是在同一层及解决问题。

另一个方面不能混淆**泛化和包含**，泛化和包含都是用于抽取用例的公共行为，因此容易混淆，其实它们是实现复用的两种不同手段，泛化关系要求父用例和子用例之间拥有“is-a-kind-of”关系，这样继承才有意义，但包含无需拥有这种关系。在使用包含的时候，当执行到使用公共行为的步骤时，必须明确指出包含的用例，而且指明使用哪个事件流。

3.4 捕获涉众关注点

理解涉众真实的关注点，是软件开发获得成功的关键，关注点源于不同的角色，并可能与系统的不同的方面相关。如何才能满足这些关注点，用例为描述系统提供了一种简单的方法，把用例分成两种主要的类型，它们是应用用例（application use case）和基础结构用例（infrastructure use case）。

应用用例描述用户如何与系统交互以实现预期的功能，而基础结构用例描述的是应用用例的每一步，如何添加诸如可用性、可靠性、性能、容错性等质量属性。

一，理解涉众关注点

如果要构建符合要求的系统，就必须保证需求是正确的，这并不意味着需要使用厚重的文档，而是文档必须抓住重点。

理解所规定的需求背后的动机，也就是涉众的关注点，这是十分重要的，如果发现涉众不断改变需求，就应该问自己一个问题，是否已经很好的理解了涉众的关注点？是否了解了预期的前景？即使回答是确定的，也必须不断地基于关注点来验证你的理解。

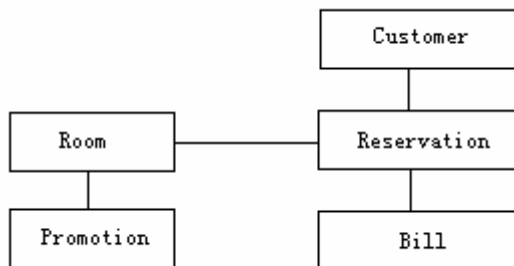
另外，还需要了解涉众的优先级，并不是所有的关注点都是同等重要的，优先级给定了哪些需求应该比其它需求更早开发，如果事情不顺利，应该知道哪些需求可以去掉，虽然功

能不大完全，但项目涉众还是可以接受。

1，理解问题域

需求工作所有的目的都集中在明确系统将要解决的问题上，所以必须能够诱导和探索出涉众的根本问题，必须了解业务领域知识，必须能够了解涉众的业务场景，必须标识出关键的功能。通称这些活动是并行的进行的，目标是描绘出涉众的真实需要。

系统能够正确地反映问题域是非常重要的，这可以把问题用“类”来描绘，这就是概念类或者称之为领域类，多这样的问题建模，称之为领域建模。领域类是只有名称和属性的类，不考虑行为，只关注类与类之间的关系。下图是针对“酒店管理系统”的简化的领域模型。



领域建模目的是通过对核心业务场景的分析，标示出系统必须保存与维护的那些信息，这样的模型只需要捕获核心概念及关系，并不需要向数据库设计中数据建模那样的过于考究细节，这个工作可以放在以后进行。

2，抽取系统特性

涉众或者项目发起人是从特定特性的投资回报的角度来证明是否需要一个新系统。特性是系统预期功能的高层描述，它可能是功能性的，也可能是非功能性的（性能、安全等），提出系统特性有很多方法，可以从面临的问题着手，也可以从分析业务流程着手，但不管什么方法，最终应该生成一个系统关键特性列表，例如：

酒店管理系统关键特性
<ol style="list-style-type: none">1，客户能够预定房间。2，前台员工可以为客户办理入住登记和结帐离店手续。3，酒店管理人员能够制定房间的收费标准，以及针对特定时段给予的优惠。4，会员消费时可以积累积分。5，当房间订满时，能够提供等候列表功能。6，必须能够处理不同类型的客户（散客、团队、会员）。7，可以通过不同渠道—代理、因特网或者电话来预订房间。8，系统应该能有以 Web 形式使用。9，在关系型数据库中存储所有的记录。10，方便审计，所有的时间都需要有日志。11，只有授权用户才可以使用其功能。12，为了提高可用性，系统必须能够记得用户常用的参数，并作为默认值。13，所有的查询操作都不超过 2 秒。

这个列表用简练的语句，概括定义了系统的需求，但并不需要涵盖所有的需求，不过，这个列表往往反映出用户的关注点。

另一方面，从开发者的角度，需要针对每个特性提出相应的验收标准，还需要呈请没有需求的确切含义，通过逐一提炼产生更长的需求列表，担任然不可避免的会出现遗漏。一个更有效的方法是分析系统的使用，发现这些特性是如何有效地置身于其中的，这就需要认真

对待用例的场景，事实上这也就是用例建模的一切。

我们需要把特性与用例场景对应起来，围绕这些特性来探索需求，并通过用例来捕获它们，通过用例，澄清与系统功能与质量都相关的涉众关注点。

3.5 功能性与非功能性需求

系统分析可以分为两类，功能性需求（FR）与非功能性需求（NFR），上述列表中 1-7 通常被认为功能性需求，而 8-13 被认为是非功能性需求。

现在一般分析主要把精力集中在功能性需求上，也就是系统必须做什么？同时每个用例场景事实上也是一个初步的测试用例，为了得到真正的测试用例，还需要添加一些测试数据，在开发初期，声称用例和测试用例是测试驱动设计所倡导的。

非功能性需求通常需要一些底层基础结构机制的支持，比如特性 12 需要一些基础结构机制来跟踪用户的输入，特性 13 需要用某些缓存基础机制来满足 2 秒钟做出响应的要求，因此，这些非功能需求会对处于基础层面的行为产生影响，进而极大的影响着架构设计。可以用用例对这些基础结构建模，为了区别其它用例，我们称之为**基础结构用例**，而其它的用例则称之为**应用用例**。基础结构用例考虑的是基础结构的关注点，而应用用例则考虑的是应用的关注点。

一、发现功能性需求

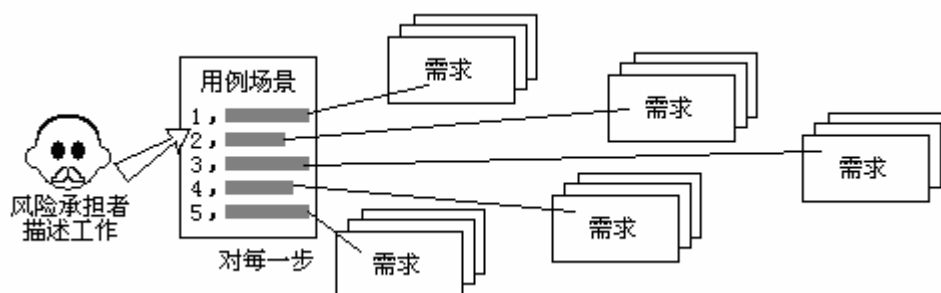
功能性需求是从业务的角度来看产品必须做的事情。当需求分析师与风险承担者交谈的时候，他们会描述这些活动，产品必须要执行这些活动以完成某部分的工作。功能性需求可以与任何可能用到的技术都无关，也就是说，它们是工作的功能本质。

在为功能性需求设计解决方案的时候，设计者加入了与解决方案中用到的技术需求，技术需求有时候与业务需求混在一起，两者一起被称之为“功能性”需求，但这里的功能性需求指的是描述产品或业务必须做的事情，它们与产品中使用的任何技术都无关。

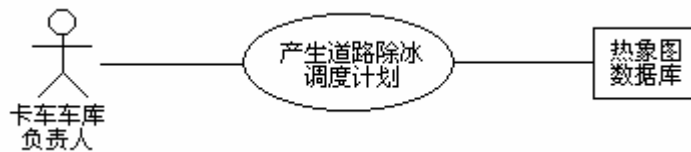
需求规格说明的目的，是作为要构建的产品的一份合约，因此，功能性需求必须十分详细的描述预期的产品将执行哪些活动。为了满足这个指标，功能性需求必须包含足够的细节，让开发能够构造出正确的产品，使需求分析师与风险承担者的误解减少到最低程度。

有些工件揭示了产品的功能性，最明显的就是场景，通过影响工作的业务事件对工作进行的划分，然后得到场景，对于每个业务事件，会有一个业务用例，并导致一个产品用例，场景表达了完成用例功能的一系列步骤。

场景中的步骤是风险承担者能够识别的，因为这是用风险承担者熟悉的语言描述的。同时由于它可能抽象程度比较高，封装了产品功能的细节，可以把每一步的细节当作它的功能性需求，现在就是通过编写功能性需求来展现这些细节，下图展示了渐进式的得到用例的功能性需求。



下面看一个产品用例的例子，来看这个过程是怎样实现的。在道路除冰系统中，有一个名为“产生道路除冰调度表”的用例，该用例参与者是卡车库负责人，用例图如下。



用例名:	产生道路除冰调度表	用例类型
用例 ID:		
主要业务参与者:	卡车车库负责人	
基本事件流:	1, 工程师提供调度日期和区域标识符。 2, 产品选择相关的热象图。 3, 产品运用热象图, 区域温度读数和气象预报来预测该区域每个道路的温度。 4, 产品预测哪条道路会结冰, 何时会结冰。 5, 产品从相关的车库中调度可用的卡车。 6, 产品向工程师提供建议的时间调度表。	

当你与风险承担者对这些步骤取得一致意见以后，对每个步骤问一个问题，为了完成这些步骤，产品必须做些什么？

例如，产品用例的第一步：

1, 工程师提供调度日期和区域标识符

第一项功能需求就非常明显了：产品接收一个调度日期。

当你问风险承担者关于调度日期有没有特殊的要求的时候，他们会告诉你调度日期不会比当前日期超出两天，这项信息提供了另一项功能需求：如果调度日期不是今天也不是明天，产品将发出警告。

进一步的另一项需求是：产品将接收一个有效的地区标识符。

问一下“有效”的含义，就会发现另一项需求。如果标识符标识了工程师负责的一个地区，那它就是有效的。如果它与工程师所希望的地区的标识符相符，它也是有效的。这导致了另外两项功能性需求：

产品将验证这个地区是在产品安装覆盖的区域的除冰职责之内。

产品将验证这个地区是工程师所希望的地区。

从一个步骤中导出的需求的数量并不重要，但是经验告诉我们，它通常少于 6 个。如果每个步骤只发现了一个需求，这表明这个场景的粒度太细，要么就是用例很复杂。

发现功能需求的目标是要发现足够的功能需求，让开发者能够构建出客户希望的产品，也是参与者用它来完成工作的产品。

让我们考虑一下例子中的另一个用例步骤：

4, 产品预测哪条道路会结冰，何时会结冰。

用例场景的这一步导致了 3 项功能需求：

产品将确定地区中的何处预计会结冰。

产品将确定通过地区的哪些路段预计将会结冰。

产品将确定何时这些路段会结冰。

现在以同样的方式，处理场景中的每个步骤，当穷尽了所有步骤以后，就为这个用例写好了功能性需求。通过与一组同事一起对需求复查了一遍，来测试是否完成了这个用例。在这个过程中，你应该能够清楚地展示为参与者提供了正确的结果的结果的用例。

二、发现非功能性需求

在某些情况下，产品的非功能性方面是做产品的主要原因，某些产品的功能可能是一样的，但是，在易用性、性能和可靠性方面下的功夫，也可能是新产品最主要的需求。非功能性需求并不改变产品的功能，也就是说不管增加多少属性，功能需求都会保持不变。但是更复杂的是，非功能性需求也可能会为产品增加功能，比如产品可能更易于使用、更安全或者更直观。

我们来考察一下在后续的系统设计中架构设计的观点，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，事实上非功能需求就是质量需求，这些信息对于产品架构设计来说，是最值得关注的，例如：

- 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？

任何非功能性需求对一个系统设计都有重要影响。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，对架构设计最具影响的因素，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。

正是这些特点，在需求分析的时候必须关注非功能性需求的问题。非功能性需求很大程度上定义了产品的质量属性，如果开发者和用户没有详细地讨论新技术方法所牵涉到可能的性能，从而会导致了用户期望与产品实际性能之间的期望差异。比起仅仅满足客户所要求的功能，软件的成功似乎更为重要。

三、用例与非功能性需求

用户总是强调确定他们的功能、行为或需求—软件让他们做的事情。除此之外，用户对产品如何良好地运转抱有许多期望。这些特性包括：产品的易用程度如何，执行速度如何，可靠性如何，当发生异常情况时，系统如何处理。这些被称为软件质量属性(或质量因素)的特性是系统非功能（也叫非行为）部分的需求。

质量属性是很难定义的，并且他们经常造成开发者设计的产品和客户满意的产品之间的差异。事实上真正的现实系统中，在决定系统的成功或失败的因素中，满足非功能需求往往比满足功能需求更为重要。优秀的软件产品反映了这些竞争性质量特性的优化平衡。如果你在需求的获取阶段不去探索客户对质量的期望，那么产品满足了他们的要求，这只能说你很幸运。但更多的可能是客户失望和开发者沮丧。

虽然，在需求获取阶段客户所提出的信息中包含提供了一些关于重要质量特性的线索，但客户通常不能主动提出他们的非功能期望。用户说软件必须“健壮”，“可靠”或“高效”时，这是很技巧地指出他们所需要的东西。从多方面考虑，质量必须由客户和那些构造测试和维护软件的人员来定义。探索用户隐含期望的问题可以导致对质量目标的描述，并且制定可以帮助开发者创建完美产品的标准。

对非功能性需求的分类没什么神奇之处，可以自由制定自己的分类。如果难以确定一向需求是属于哪一个类，就不要在这点上花太多的时间。如果有疑问，就把它归到最有可能的那一类，或者它似乎属于两个类型的时候，就让它有两个类型，少数需求的类型不确定也不会带来什么害处。

产品用例代表当工作响应一个业务事件的时候，产品所作的工作，它被分解成一些步骤，针对这些步骤，可以确定功能性需求。某些非功能性需求可以直接与一项功能需求联系起来，某些非功能需求适用于整个用例，下图展示了功能于相关的非功能需求的这种联系。在图中，用例有三项功能性需求，每一项都有一些非功能属性。用例作为一个整体，必须满足一定的易用性需求，而观感需求与整个产品有关。



四、非功能性需求类型与软件质量模型

非功能性需求有很多类型，大部分情况下与产品的质量属性有关，比较常用的是：

- 观感需求：产品的外观精神实质。
- 易用性和人性化需求：产品的易用性程度，以及更好的用户体验所需要的特殊可用性考虑。
- 执行需求：功能是实现必须多块、多可靠、能有多少处理量、多精确。
- 操作和环境需求：产品的操作环境，以及对该操作环境必须考虑的问题。
- 安全性需求：产品的安全性、保密性与可恢复性。
- 文化和政策需求：对于产品的操作所设计的人文化和习惯带来的特殊需求。
- 法律需求：哪些法律和表顺适用于该产品。

在描述非功能需求的时候，应该表达这个需求的描述和理由，先记录下风险承担者对这项需求的意图，将来会为每项需求加上一个测量指标（称之为验收标准）。

例如，对于观感需求，当仔细研究了产品的目标受众以后，风险承担者希望产品对受众有吸引力，可能会要求如下的一些：

- 显而易见的使用简单性；
- 平易近人，这样人们就会不犹豫的使用它；
- 权威性，这样用户会依赖它和信任它；
- 与客户的其它产品兼容；
- 对儿童或者其它特定群体有吸引力；
- 不显眼，这样人们就不会注意到它；
- 创新兵表现出艺术的方式；
- 看上去很专业；
- 令人兴奋；
- 酷，

或者人们认为易用性是实符合用户能力及其对使用体验的期望，我们可以看一下需求规格说明中对用户的分类，他们需要哪种类型的人来完成他们的工作？他们需要哪种类型的产

品来完成他们的工作？产品的易用性对用户使用产品的工作效率、错误率以及用户对新产品的接受程度都会产生影响。所以在编写易用性需求之前，需要考察客户需要产品达到什么目标。

比如：该产品应该易于使用。

这个需求就太模糊了，可以采取下面的写法：

产品应该对公众易于使用，他们也许不能看懂英语。

产品应该对具有认证资格的机械工程师使用。

下面是一种描述方式：

描述：产品应该为个人顾问提供一种喜欢的工作方式。

理由：为了树立起顾问对产品的信心。

验收标准：在经过 8 周的熟悉期以后，75% 的顾问将愿意使用该系统。

还要注意“易于使用”和“易于学习”有轻微的不同，易于使用的产品着眼点是出促进持续的工作效率，他们在使用前可能经过了培训。易于学习的产品设计着眼点是针对不太经常使用的功能，他们可能忘了它的用法，但很快能够学会并使用，比如报表处理等。

下面是一种描述方式：

描述：当未经培训的公众第一次尝试使用该产品的时候，它应该易于学习。

理由：潜在用户可能从未使用过这类产品。

验收标准：由公众组成的测试小组中，90% 的人在第一次使用该产品的时候，能在 45 秒内成功地通过产品购买一张票。

易用性还可以包括：

- 用户的接受率或者采用率；
- 通过引入该产品而导致的的生产效率的提高；
- 错误率（或者错误率的降级）；
- 被不说英语的人采用；
- 对残障人士的可用性；
- 被没有计算机使用经验的人使用（这是一项重要的，但往往被遗忘的考虑）。

对于操作和环境的需求，需要研究当产品与伙伴产品合作的时候，或者与其它信息系统接口的时候，或者与其它提供信息的系统接口的时候，也许要写出操作需求，这些系统在用例图中表现为参与者，在上下文模型中表现为相邻系统。

安全性可能是最难指明的一种需求，并且如果不正确的话可能会给产品带来极大的风险，一般来说，安全性包括 3 个方面：

1) 保密性：

在编写这类需求的时候，实际上是在指明允许的访问（授权），在什么情况下授权是有效的，对每个得到授权的用户来说，允许访问哪些数据和功能。

2) 可得性：

得到授权的用户可以不受组织的访问数据，有时候也需要考虑防止数据丢失或者恢复丢失数据的需求。

3) 完整性：

产品所保存的数据与相邻系统发送给产品的数据应该保持一致，除此以外可能还需要一些完整性检查：

- 防止得到授权的用户以不希望的方式使用。
- 进行审计以检测不正确的用法。
- 在一些非正式的事件之后（断电、异常操作、火灾、水灾、爆炸），验证产品的文完整性。

4) 审计:

必要的时候,可以把审计列在需求规格说明的安全部分中,大多数会计产品都有审计方面的需求,审计需求要求产品包括一份审计跟踪记录,审计需求的准确实现必须与审计师沟通,例如可以这样写:

描述: 产品应该保留法定的时间内所交易的日志。

理由: 这是审计师所要求的。

3.6 验收标准

“验收”意味着解决方案完全满足了需求,但是为了测试解决方案是否满足需求,需求本身必须是可测量的。

一、验收需要标准的原因

如果产品有一项需求,要执行某个功能或者具备某种属性,那么测试活动就必须展示产品确实执行了该功能,或者具备了该项期望的属性。为了进行这样的测试,需求必须有一个测试基准,这样测试者才可能把提交的产品与最初的需求进行比较。测试基准就是验收标准,也就是产品的某种定量描述,它说明了产品必须达到的标准。

作为系统给架构师,我们有理由认为,当他知道了产品的验收标准以后,他就会按标准构建,例如验收条件是:在水下 15 米处连续工作 15 小时。那么他们不太可能使用不防水的材料构建这个产品。

根据一个公认的测量指标对一项需求进行测试,其中最难的是定义一个测量指标,如果风险承担者要求产品“优秀”,那么必须找到一种方式来测量它“优秀”的程度。当然,这项指标必须得到风险承担者的同意,你可能不知道他说的“优秀”是什么意思,但与风险承担者就测量标准达成一致意见以后,构建者就可以创造出正确的产品,测试者也就可以证明它是正确的产品。

于是,我们需要发现一些“优秀”的测量指标,你可以提问“优秀”指的是什么?如果回答是:“职员喜欢使用它”,进一步的探究就会是:“产品应该职员本能的喜欢它,毫不犹豫的使用它。”

这样的回答就给出了一些可以测量的东西:在见到产品品质前的犹豫,到完全采用它需要多少时间?或者运用一段时间以后满意度(通过员工调查表),当与风险承担者达成一致意见以后,就有了一个“优秀产品”的测试标准。

要记住,所有的需求都可以测量,你要做的只是找到合适的尺度。

二、测量的尺度

测量的尺度是用于测试产品符合程度的单位,如果需求是针对一定的操作速度,那么尺度就是完成给定动作所需要的时间。对于一个易用性要求,可以测量学习产品所需要的时间,或者达到公认的能力水平的时间。对于可升级性,尺度可能使升级一项功能所需要的成本(钱币),或者是工作量(人/月)。几乎所有的东西都有测量尺度,关键是我们使用正确的尺度。

三、理由

理由就是需求的原因,或者存在的道理,我们发现,为需求加上理由,会使理解真实的

需求变的更加容易。风险承担者常常会告诉你一个解决方案，而不是他们真实的需求，或者他们会告诉你一个模糊的需求，以至于没什么用处。

当用户说需要一个“优秀”的产品的时候，一个理由就是用户乐于使用，这时候的测试条件前面已经讨论过了。但如果理由是“容易使用”，那对应的需求就有了完全不同的意义，导致相应的验收标准也不相同。

理由不仅仅是帮助你发现验收标准的指导，还可能帮助你发现混合在一起的模糊的需求，比如前面说的“优秀”的产品，几个风险承担者可能是不同的理由，比如：易于使用、令人兴奋、还有用户愿意再次使用它。这样就会有三项需求，每项都有不同的不同的属性，以及相应的测量标准。

在向风险承担者询问理由的时候，最常说的词可能就是“为什么？”这是对的。

需求分析师的职责就是问为什么，不断地问为什么，直到理解了需求的含义。但事情还没有完，我们还需要做很多工作来测量需求的真正含义，这就是为什么需要导出验收标准。

为了找到合适的验收标准，要从分析得到的需求描述和理由开始，例如：

描述：产品应该让购买者容易找到他选择的音乐。

这比较模糊，让我们再看看理由：

理由：音乐购买者习惯了方便，不能忍受很不方便的找到音轨。

那么，速度成了合理的测量单位，而且与音乐购买者有关。市场人员调查相关群体，得到的结论是说：购买者应该在 3 个动作之内找到音轨，其中 10 秒钟是忍受极限。为了比对手做得更好，于是导致如下验收标准：

验收标准：平均音乐购买者应该能够在 6 秒钟之内，通过不超过 3 个动作，定位任意一段音乐。

由于真实世界、技术甚至成本的限制，某些验收标准可能不能实现，这是对验收标准就需要进行调整，以适应产品的操作环境、使用意图和客户的预算。请把这些调整堪称业务误差，因为我们可以肯定某些用户的操作水平会在平均水平之下，所以可以将验收标准调整如下：

验收标准：90%的音乐购买者应该能够在 6 秒钟之内，通过不超过 3 个动作，定位任意一段音乐。

四、非功能需求的验收标准

非功能需求是产品必须具备的品质，某些非功能需求看上去很难量化，但是总可以把它加上数字标准，如果不能对一项需求进行量化和测量，它就不一定是一项真的需求。它可能是多项需求合并在一起了，或者没有仔细考虑好、没有理由、偶然的，也许根本就不是需求。对这样的需求，或者是想办法量化它，或者是删除它。

例如，有这样的需求：

描述：产品应该对用户友好。

初看起来很模糊，但是可以找到测量它的方法，可以问一下用户，在他们心目中，“用户友好”的准确含义是什么？是易于学习，还是易于使用，或者是吸引人，还是其他含义？

假定用户呈请了他们的意图，比如：“我希望我的员工能够快速学会使用这个产品。”这就说明了一个尺度（快速），比如培训的时间等，我们可以这样写：

验收标准：新用户第一次使用该产品的时候，应该能在 30 分钟内完成数据的添加、更改、删除操作。

请注意，这个测量标准定义的是具体用户关于“产品友好”的含义，但不是所有“产品友好”的含义。像前面关于“优秀”的含义，下面的表达式可测量的：

验收标准：在引入该产品的 3 个月内，60% 的用户应该用它完成工作，在这些用户之中，应该有大于 75% 的用户对产品表示赞许。

要注意验收标准对需求是起到澄清作用的，通过讨论测试尺度，需求从一个模糊的、二义性的意图，变成了一个完整的、可测量的需求。风险承担者在一开始是不太可能用这样的方式表达自己的意图的，建议你顺着这样的流程走下去，这样并不会因为需要可测量就放慢自己的需求收集过程，而是可以更深入的了解风险承担者的意图以及理由。通过分析，可以编写对验收标准的最佳诠释。

1，产品是否失败

验收标准可以以一个问题来确定：“什么会被认为产品失败？”假定有如下需求：

描述：产品必须在可接受的时间内产品道路除冰调度计划。

显然测量的尺度是时间，那么如果客户告诉我们超过 15 分钟才产生计划是不能接受的，就可以得到如下验收标准（加上了业务误差）：

验收标准：在 90% 的情况下，工程师应在在 15 分钟之内得到产品成生的道路除冰调度计划，任何情况下都不得超过该时间的 20 秒。

但是有的时候对测量标准不能达成一致意见，这种情况就需要考虑需求是不是有几项需求组成？能不能分解成不同测量方式的几个独立的需求。极端的情况是需求根本不切实际，比如：“该产品应该足够好，如果我的祖母还在世，她将会喜欢它。”显然这个需求是没有办法测试的。

2，主观测试

某些需求需要通过主观测试来进行测量。例如，有一项文化需求：“不冒犯任何团体。”那么验收标准可能是这样的：

验收标准：产品应该不会让测试组 85% 的人感觉被冒犯。测试组有可能与产品法生联系的人员代表组成，测试组所代表的业务团体相感觉被冒犯的不超过 10%。

不能指望 100% 的人通过测试，在这个情况下，业务误差保护了产品不至于受到少数极端观点的攻击，同时又允许对“感觉被冒犯”进行测量。

你可以使用原型而不是最终产品来进行这类测试，这样可能更有成本效益。

另外，百分比数字不是任意制定的，它必须来自于经验数据，并且得到用户的认可，这种认可包括用户对这一目标的原因有很好的理解。

3，观感需求

观感需求是关于产品的外观和行为的精神、情绪和风格。

很多用户要求产品具备本公司的代表颜色，这种要求要么是坚持品牌标准，要么是希望强化顾客的认知，但这两种情况稍有不同。

坚持品牌标准：

验收标准：产品应该有市场部领导认证符合今年的公司品牌标准。

强化顾客认知：

验收标准：60% 的目标用户在第一次见到该产品的 5 秒钟内，就意识到该产品属于该公司。

观感需求虽然是对意图的感觉式说明，但经过反复询问理由，还是可以找到可测量的方面。

4，易用性和人性化需求

易用性和人性化需求规定了产品对目标用户的方便性与合适性。下面来看一个例子：

描述：产品应该是直观的。

为了测量“直观的”，必须考虑直观是针对什么样的用户而言的，如果用户是道路除冰工程师，他们拥有工程学位，并拥有气象方面的经验。

理由：工程之必须认为产品直观，否则他就不愿意使用它。

有了这个理由，“直观”就具备了不同的意义。

验收标准：在首次使用该产品时，在不参考产品以外的帮助的情况下，道路工程师应该能够在 10 分钟内得到一封正确的除冰预报。

有时候“直观”值得是易于学习，这是必须询问可以花多少时间用于培训，从而得到这样的验收标准。

验收标准：在经过一天的培训以后，10 个道路工程师中应该有 9 个能够成功的完成[选择的任务清单]。

易用性需求的验收标准也可以使用量化来完成给定任务完成时间、允许的差错率、用户的满意率、易用性实验室的评分等，重要的事情是发现需求的真正含义，验收标准能正确测量这个含义，从而确认需求的含义。

5，执行需求

执行需求是关于产品的速度、精度、荣俩格等方面的需求，大多数情况下，执行需求的本质将支出测量的尺度是什么，例如：

验收标准：95%的情况下，响应时间不超过 1.5 秒，其它情况下不超过 4 秒。

可访问性：

验收标准：在系统投运的前 3 个月中，在早上 8:00 至晚上 8:00 之间，产品的可用时间应该达到 98%。

验收标准也可以使一个范围，例如：

验收标准：产品应该允许每小时 3000 次下载，但是每小时 5000 次更好。

使用范围的原因是防止开发者构建过于昂贵的产品，在预算与设计限制之间取得合理的平衡。

6，安全性需求

安全性需求包括了产品的许多方面，例如：

描述：只有使用 A 类登陆的工程师能够进行增加、修改或者删除气象站的数据。

验收标准：在 1000 次气象数据的增加、修改或者删除中，全部由 A 类登陆的工程师完成，没有例外。

至于数据的一致性等等问题，都属于安全性考虑的范围，这里就不再讨论了。

五、功能性需求的验收标准

1，功能需求验收标准的描述

一般来说，对于功能需求来说，不存在测量的尺度，动作要么完成，要么没有完成。功能需求可能是不同类型的动作，例如：

描述：产品应该记录气象站的数据。

理由：准备除冰调度表需要这些读数。

验收标准：记录下来的气象站数据应该与负责传输这些数据的气象站数据相同。

在这里，验收标准并没有说如何被测试，这种陈述保证了测试者用它来确保符合需求。

如果功能需求是进行某种计算，则验收标准就应该指出：“计算结果应该与某机构的看法一致……”

功能需求的一般原则是：验收标准确保功能被正确执行，从中导出测试用例。

2，测试用例

你可能会发现，做了这么多事情，你就可以为功能需求编写测试用例了。极限编程的观点是：先编写测试代码，再编写产品代码。基本思想是强制程序员关注于了解所有功能的成

功标准。

3.7 编写完整的规格说明

系统开发的一个主要问题是对问题的误解,所以需求分析师必须以一种可测试的方式写下需求,确保提出需求的风险承担者理解并且同意这些需求,然后再传递给后面的开发者。

分析师编写需求是为了确保确保个方面对需要做的事情达成一致的理解,虽然这可能是个负担,但我们发现这是记录与沟通需求的唯一方法。由于需求是业务需求,所以需要用业务语言来描述,这就需要分析师与风险承担者一起来编写需求。这样,不懂技术的风险承担者才可能理解并检验这些需求的正确性。

当然编写需求并不是一项独立的活动,它需要其它的一些活动支持(网罗需求、制作原型、质量关),但是为了正确理解编写需求的方法,我们暂且单独的来看它。

编写功能需求的另一个方法是建模,可以有多种模型可供选择,关键是“写出”需求和各项功能的验收标准,并且与风险承担者达成一致的意见,否则任何需求都没有意义。

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。

需求规格说明的模版如下:

项目驱动: 描述项目的理由与动机。

- 1, 项目的目标: 投资创建产品的理由以及希望取得的业务上的好处。
- 2, 客户、顾客和其它风险承担者: 产品涉及他们的利益以及对他们的影响。
- 3, 产品的用户: 预期的最终用户, 以及他们对产品可用性的影响。

项目限制条件: 加在项目和产品上的约束条件。

- 4, 需求限制条件: 项目的局限性和产品设计的约束条件。
- 5, 命名标准和定义: 项目的词汇表。
- 6, 相关事实和假定: 对产品产生一定影响的外部因素。

功能性需求: 产品的功能。

- 7, 工作的范围: 针对的业务领域。
- 8, 产品的范围: 定义预期产生的边界。
- 9, 功能与数据需求: 产品必需作的事情以及所操作的数据。

非功能性需求: 产品的品质。

- 10, 观感需求: 预期的外观。
- 11, 易用性和人性化需求: 产品让预期用户使用应该是什么样子的。
- 12, 执行需求: 速度、大小、精度、人身安全、可靠性、健壮性、可伸缩性、持久性和容量等需求。

13, 操作和环境需求: 产品预期的操作环境。

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

15, 安全性需求: 产品的信息安全、保密性和完整性。

16, 文化与政策需求: 人和社会因素。

17, 法律需求: 满足适用的法律。

项目问题: 这些是适用于构建产品的项目。

18, 开放式问题: 那些尚未解决的问题, 可能对项目的成功有影响。

19, 立即可用的解决方案: 利用已有的组件, 而不是重新开发。

- 20, 新问题: 引入新产品而带来的问题。
- 21, 任务: 把产品投入使用必须要做的事情。
- 22, 迁移: 从现有系统转换的任务。
- 23, 风险: 项目最有可能面对的风险。
- 24, 费用: 早期对构建产品的成本或者工作量的估计。
- 25, 用户文档: 创建用户指南或者文档的计划。
- 26, 后续版本需求: 可能在产品将来发行版本中包括的需求。
- 27, 解决方案的想法: 我们不想错失的设计想法。

3.8 质量关

质量关是对需求进行测试, 是对每项需求成为需求规格说明之前必须通过的一个节点。质量关通常由一到两个人组成, 可能是需求分析师或者测试人员, 只有他们有权允许需求通过质量关。在允许需求加入规格说明之前, 他们一起检查每项需求的完整性、相关性、可测试性、一致性、可跟踪性以及其它的一些质量属性。

质量关的一项任务, 就是确保每项需求都有一个验收标准, 验收标准是对需求的一种度量, 是需求可理解并且可测试。

建立质量关的另一个原因是防止需求遗漏, 需求有时候进入规格说明以后没有人知道它来自何处, 或者它对产品带来什么价值, 避免这些问题的唯一方法就是建立质量关。

3.9 复查规格说明

质量关存在的目的, 是把不好的需求拒之门外, 但他一次只处理一项需求, 当考虑需求规格说明是否完整的时候, 应该对他进行复查。最终的复查会检查需求是否存在遗漏或者矛盾的需求, 保证所有的需求相互一致。简而言之, 复查工作是确保规格说明书是完整的和恰当的, 这样就可以转向下一个开发阶段。

复查也为重新评估产品的费用和风险提供了机会, 既然拥有了一份完整的需求, 对产品的了解就会比起动会议更多。在需求规格说明完成之后, 对产品的范围和功能有了一个准确的认识, 所以此时就是重新更准确的估计产品规模的时候了, 另外诸如可选择的解决方案架构、预估费用等都可以更清晰的完成。

这个阶段有些需求会引入巨大的风险, 开发方应该评价自己的能力, 并且预估风险。

软件需求说明的审查主要需要审查下面几个问题:

- 1. 审查需求的一致性
- 2. 审查需求的现实性
- 3. 审查需求的完整性和有效性

软件需求说明审查中的问题:

- 1, 所规定的软件目标和任务与系统的目标和任务相符合吗?
- 2, 与所有系统成分的重要接口都已被描述了吗?
- 3, 研制项目的数据流图、数据字典、数据结构充分确定了吗?
- 4, 图表都清楚吗? 每个图表在不加补充说明的情况下能被理解吗?
- 5, 主要功能在规定的范围之内吗? 每一种功能被充分说明了吗?
- 6, 设计的限制条件是现实的吗?
- 7, 开发的技术风险是什么?
- 8, 考虑过软件需求的其他方案吗?

- 9, 检验标准是否详细? 他们能否确认系统是成功的?
- 10, 有无遗漏、重复或不一致的地方?
- 11, 用户是否审查了初步的用户手册?
- 12, 软件计划中的估算是否需要修改?

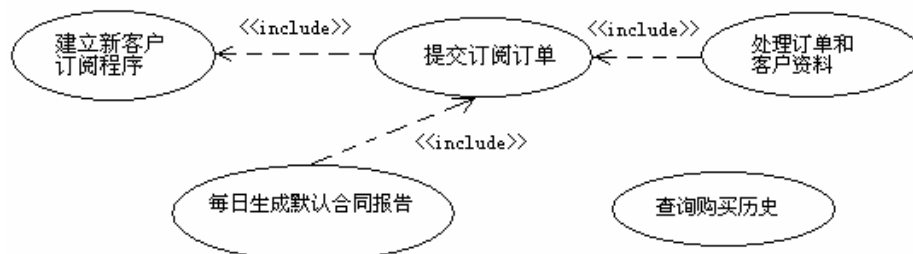
3.10 风险分析和优先级的考虑

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。



在优先级问题上, 还要考虑功能的关注点、重要性和影响性, 综合起来考虑问题。

3.11 模式与需求复用

我们已经说过, 2000 年以后, 软件产品走上了规模经济的时代, 随着产品规模的上升, 软件组织已经有可能改善投资回报 (ROI)。伴随着非常大型的项目, 软件组织将达到更好的经济规模。正是这种规模软件经济的理念, 在分析与设计方法的思路, 提出了和过去完全不同的要求, 重用、复用和变更成为一个重要的主题。

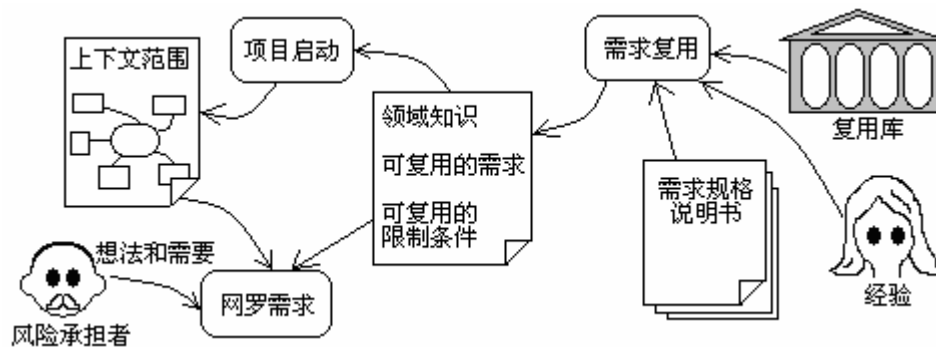
一、什么是复用需求

尽管产品有许多它们自己的特殊功能, 但我们构建的产品并不完全是独一无二的。如果

发现某人在某处开发了一个产品，包含了某些与您的工作有密切关系的需求，利用已经完成的工作，需求收集者的效率将会大幅度提高。所以，我们强烈建议在需求项目的早期阶段就寻找那些已经写下来的可复用的需求。

但是这个需求并不可能原搬照套，必须对它做很多工作，成功的需求复用始于一种组织文化，这样的文化有意识的鼓励复用而不是重新发明，比如产品线的建立以及需求描述的规范化。如果组织是这种态度，你就可以准备好在自己的需求过程中包含需求复用了。

可复用需求可以有几种来源：规格说明书的复用库、其它类似的或相同领域需求规格说明，或者非正式的来自于其他人的经验，如下图所示。



如果需要知道是否有相关的可复用需求，需要先对正在调查的工作有所了解，在召开启动会议的时候，要特别注意需求规格说明的前 7 个部分：

- 1，项目的目标：组织中是否存在其它的项目与本项目一致，或者实际上包含相同的领域？
- 2，客户、顾客和其它风险承担者：可以复用一份客户、顾客和其它风险承担者分析表格吗？
- 3，产品的用户：其它产品是否涉及相同的用户，从而具有类似的易用性需求？同时要参考风险承担者图或者表格。
- 4，需求限制条件：限制套间是不是也在其它项目中指定？
- 5，命名标准和定义：可以使用其他人的词汇表，而不需要发明新的词汇。
- 6，相关事实和假定：注意最近一些项目的相关事实，它们可能也适用于现在的项目，对其它项目的假定也适用于本项目吗？
- 7，工作的范围：本项目很可能成为组织正在开发的其它项目的相邻系统，要利用其他工作上下文模型已经建立好的接口。

在寻找潜在可复用需求的时候，不要急于说现在的项目与此前的项目根本不同，不错，主题方面会不同，但如果不去看名字，而关注一下底层功能是不是一样呢？能不能少作调整就可以用在自己的需求说明书中呢？

启动会议中风险承担者是很好的可复用组件的来源，要向风险承担者询问关于其它包含项目的相关知识和文档，要考虑是不是有其他人调查过一些主题领域，这些领域与当前项目有没有重叠？要仔细检查相关启动会议的提交产物，它为确定可复用知识提供了关注点。

使用一个规范过程来编写规格说明的好处之一，就是自然得到了将来项目更容易服用的需求。

获取可复用需求的一个重要方法，就是利用同事非正式的经验，这些经验可能不是直接可用，但是稍加改变就可以用在当前项目中。另外从已有的需求规格说明和领域模型，都可能获取有用的知识，这需要你有抽象的能力，能透过主题来发现可复用的组件，这里面有一个关键的思维就是模式的思想。

二、需求模式

模式是一种指南，当你试图重复某项工作的时候，或者近似的重复某项工作的时候，它给出了一种可遵循的形式。模式的概念最初来自于 Alexander 关于建筑学领域的探讨，Christopher Alexander（克里斯多弗·亚历山大）作为建筑师自问：“质量可以客观评价吗？”

你认为美的东西，对所有观看者都是美的吗？”比如，你设计了一个房子的入口，有什么理由认为这个这个入口的位置就是美的呢？

Alexander 认为，建筑学系统确实存在这样一种客观规则，他认为评价一个建筑是否美观，并不仅仅是观察者品味问题，我们可以构造一个评价系统，客观评价它的美学问题。

问题的核心是：只要把注意力关注在解决同样问题的不同解决方案，以缩小自己的关注点，从而找到优质设计的相似点，而这些相似之处称之为“模式”。

到上个世纪 90 年代中期，一群优秀的开发者（Gamma、Helm、Johnson 和 Vissides）偶尔发现了 Alexander 关于“设计模式”的研究成果，他们发现，建筑学中的“设计模式”理论也同样适合软件设计。因为软件设计中也同样存在不断重复出现，可以用某种相同方式解决的问题，也可以按照某种模式进行识别，并且可以在这个模式的基础上创建特定的解决方案。这就是当今在软件设计领域非常有影响的 GoF 的设计模式。

我们注意到在 GoF 的定义中，对于每个模式，都必须有如下基本要素：

项目	描述
名称	每个模式都有一个独一无二的名称，人们用名称来鉴别模式。
意图	模式的目的是。
问题	模式试图解决的问题。
解决方案	对于自己出现的场景的问题，模式怎样提供一个解决方案。
参与者和协作者	模式包括的实体。
效果	使用模式的效果，使用模式同时研究其约束。
实现	怎样实现模式，注意：实现只是模式的具体表现形式，而不能象模式本身那样去分析。
GoF 参考	在四人团的书中得到更多的信息的位置。

这样就很好地避免了模式的二义性，提升了复用的可能。

从需求的角度，同样也存在着相似的问题，例如某一个具体书店卖书的需求过程：确定价格、计算税金、收款、包书、谢谢顾客，如果不考虑哪个具体的项目，把它总结成卖书的模式（假定存在的话），将来任何卖书活动都可以使用这个模式，这将得到很好的回报，因为使用这个模式就不需要每次都发明卖书的活动。

模式改进了需求规格说明的精确性和完整性，通过寻找适用于您的项目的模式，复用了一些其它项目的知识，减少了产生规格说明书的时间。记住，模式是一种抽象，可能需要一些工作使它适应你的需要，但是这样一来，你可以对他人的工作有更深刻的见解。

下面，让我们看看模式是怎样应用在需求上的。

三、业务事件模式

先来看一个需求模式的例子，然后我们再讨论如何创建模式以及如何把它们应用于将来的项目，这个模式是基于对一个业务事件的响应：

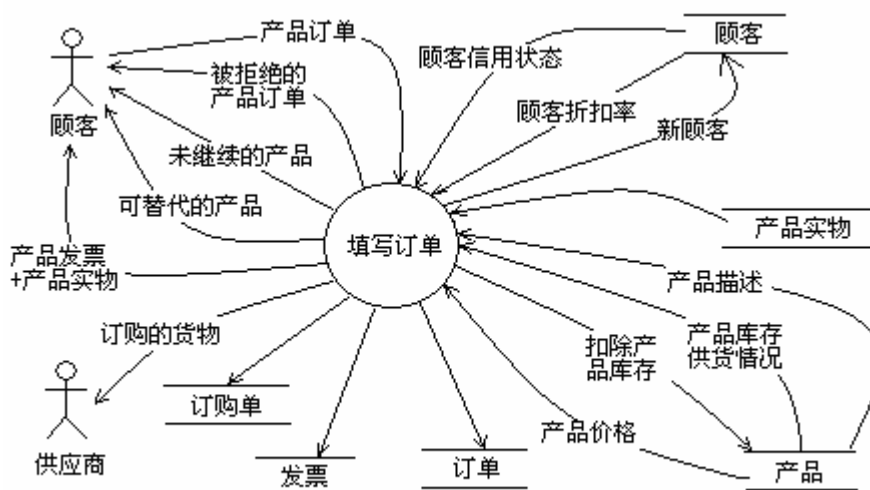
<p>模式名称：客户希望购买一件产品。</p> <p>上下文：从客户那里接受订单，提供或订购该产品，并开具发票的模式。</p> <p>要点：一个组织有来自顾客的要求，希望提供物品或者服务。不能满足顾客要求可能导致顾客寻找其他供应商，有时在收到了订单时却没有产品。</p> <p>解决方案：下面的上下文范围模型、事件响应模型和类图定义了该模式的组成部分。每个参与者、过程、数据流和数据存储、业务对象和关联都在后面所附的文本中有详细定义，文本中使用的名称与模式中使用的保持一致。</p>

注：这里的上下文表达该模式相关的边界情况，也相应于希望模式提供解决方案的场景。

1，事件响应上下文

下图所示的上下文范围模型是对该模式讲述的主题的一个总结，您可以通过查阅该图可

以知道该模式的细节是否与您正在做的事情相关。



这个上下文图定义了顾客希望购买一件产品的模式边界

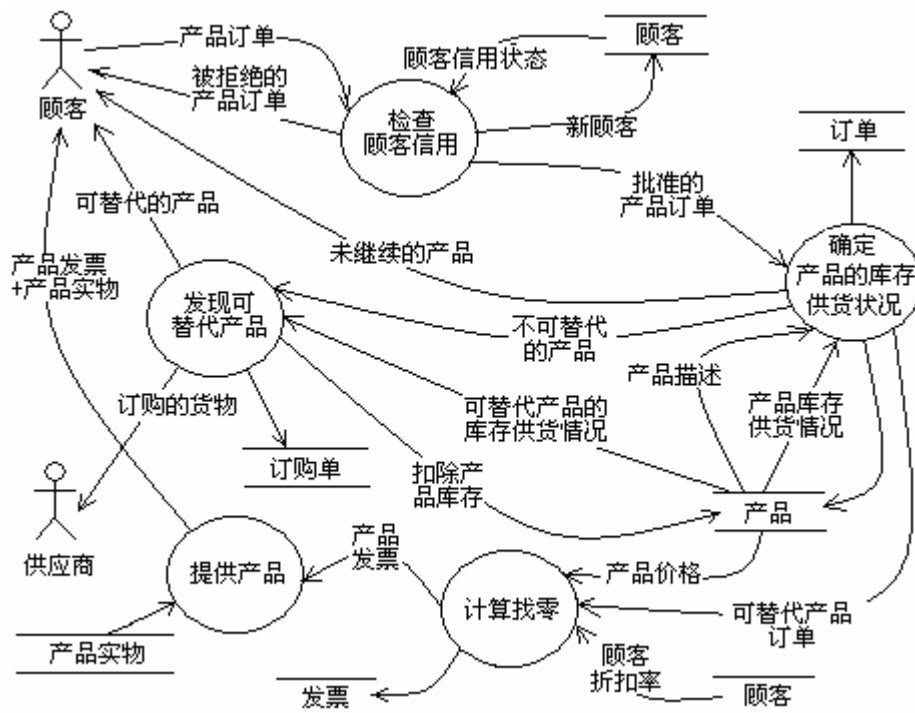
图中，上下文边界上有一些数据流（或者实物流），表明了该模式所做的工作。如果这些数据流中的大部分符合您的业务事件输入和输出，那么该模式就有可能用在项目中。

了解了该模是不是适合使用以后，就可以进一步了解细节，这可以通过一些不同的方式来表达，使用的技巧取决于对模式了解的深度和广度，例如：

- 从顾客发出产品订单开始所发生的事情的逐步文字描述。
- 所有与订单填写相关的单个需求的正式定义。
- 一个细节化的模型，在指定单独的需求之前，把该模式分解为一些子模式以及它们的依赖关系。

2. 事件响应的处理

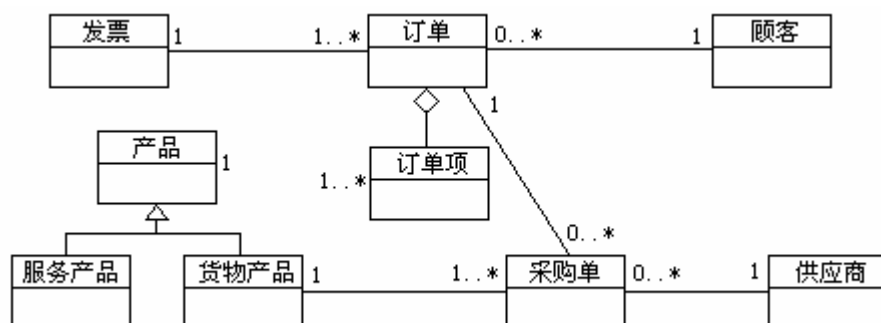
下图反映了一个大的模式如何被分解成一些子模式。从这个图中，我们可以确定潜在可复用的需求组。例如，图中展示了一个名为“计算找零”的子模式，它与其它子模式交互。不论何时，只要我们要为确定任何类型的找零确定需求的时候，都可以独立的使用这个子模式。子模式之间的互动表明，当我们对计算找零的模式感兴趣的时候，其它模式也可能与我们相关。描述子模式可以有多种方式，包括 UML 活动图、顺序图或场景说明。



说明：该图把顾客希望购买一件产品这一模式分解为 5 个子模式（一组功能相关的需求）。同时展示了它们之间的依赖关系。每个子模式（用圆圈表示）通过命名的数据流与其它子过程、数据存储或者相邻系统相联。每个子过程也包含一定数量的需求。这个模型没有任意强制指定处理过程的顺序，而是关注于处理过程的依赖关系。例如，我们可以看到“确定产品库存供货情况”过程依赖于“检查顾客信用”过程，为什么？因为前一过程需要知道“批准的产品订单”才能开始工作。

3. 事件响应的数据

下面的类图向我们展示了参与客户希望购买一件产品之一模式的对象以及它们之间的联系。我们可以利用面向对象的方式把每个对象特有的属性和操作组织在一起。例如，目标产品有一些独特的属性，诸如名称和价格；类似的，它有一些独特的操作，诸如计算折扣、发现库存水平等。如果有关于产品这个对象的这些知识，就意味着无论何时需要制定一个产品需求的时候，我们就可以复用这个知识的一部分或者全部。在课程后面我们讨论建立分析模型的问题时，会对这个问题有更深入的研究。



说明：该类图展示了一些对象和它们之间的关系，这些都是客户希望购买一件产品这一模式的一部分。请考虑这个图所表示的业务规则：顾客可能下 0 份或者多份订单，每份订单

都需要开发票。一份订单包含一些订单项，每个订单项是一项服务或一些产品。只有产品可以向提供商订购。现在请考虑在哪些情况下这些业务规则、数据或者过程可以被复用？

强调一下，可以使用各种模型对数据建模，这里使用的是类图，也可以使用实体-关系模型，这与组织中分析的习惯有比较大的关系。

四、通过抽象形成模式

上面我们讨论的需求模式是很多业务事件的分析结果，这些业务事件通常来自于不同的组织，实例中业务事件是客户等待购买产品这一主题，我们通过抽象来捕获这类业务事件共同的处理策略，从而推导出这个模式。因此，这个模式包含的业务策略将适合于大多数顾客希望购买某种产品的情况。如果现在项目中包含了一个中心内容是顾客希望购买某种东西的业务事件，那么，这个模式是一个符合实际的起始点。

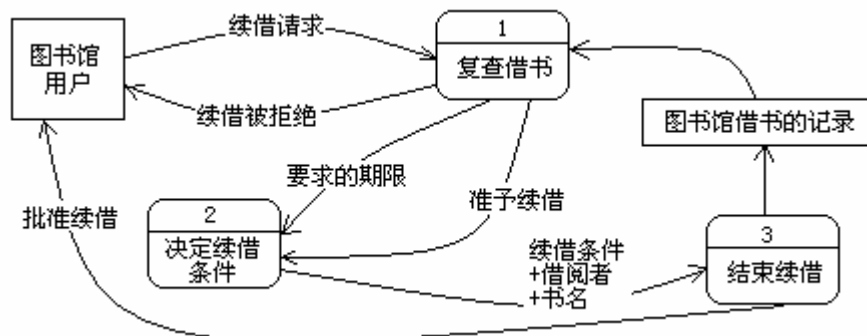
建立模式需要透过特性看到共性：要把眼光从组织现在使用的技术上移开，看到被处理的业务策略，考虑工作也不要从它当前的具体形式考虑，而要把它看成将来也可以在其它领域实现的工作模型。

当然会存在很多模式，涉及了许多业务事件和主题领域，为了把它们记录下来以备查找，我们可以根据下面的模板按一致的方式把它们组织起来。

模式名称：一个描述性的名称，它使得我们更容易和其他人交流。
要点：该模式存在的理由。
上下文：该模式相关的边界情况。
解决方案：通过文字、图片和对其它文档的引用来描述该模式。
相关模式：可能与该模式一起应用的其它模式；可能有助于理解该模式的其它模式。

1. 特定领域的模式

假定目前的工作是关于一个图书馆的系统，在上下文范围中几乎肯定有这样的业务事件：图书馆用户希望续借图书。下图展示了对该事件的系统响应模式，当一个图书馆用户提交了一个续借请求的时候，系统的响应要么是拒绝续借，要么是批准续借。

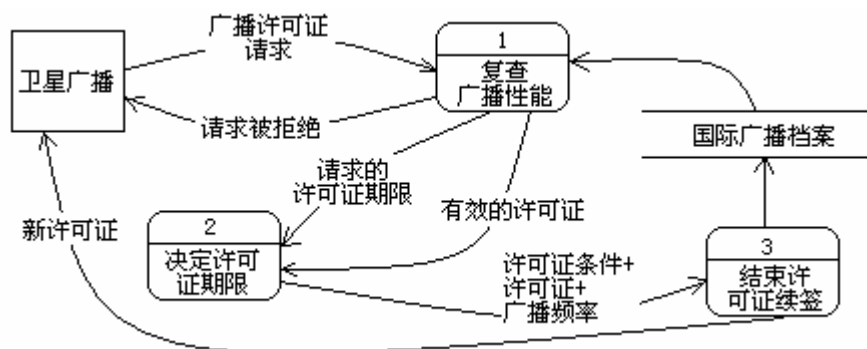


在图书馆领域的项目上的工作导致了一份详细的需求规格说明，可用于创建一个特定的产品。这项工作的一项副产品是识别了某些有用的需求模式，一些与业务相关的需求，这些都可以在图书馆领域的其它项目中复用。

使用了一致的原则来确定需求之后，就可以有机会让其他人更有机会接触它，从而使它可以复用。如果开始另一个图书馆项目，已经编写好的规格说明将是一个好的起点，它们通常是这个领域可复用需求的巨大来源。

现在想象目前的工作是一个极不同的领域项目，例如“卫星广播”。这个上下文范围中有一个业务事件是“卫星广播希望对许可证续约”。当卫星广播者提交了广播许可证请求之

后，系统的响应要么是拒绝该请求，要么是提供新的许可证。



当你在为卫星广播项目的需求工作的时候，也会发现图书馆项目的需求模式，可能在这个项目中复用。

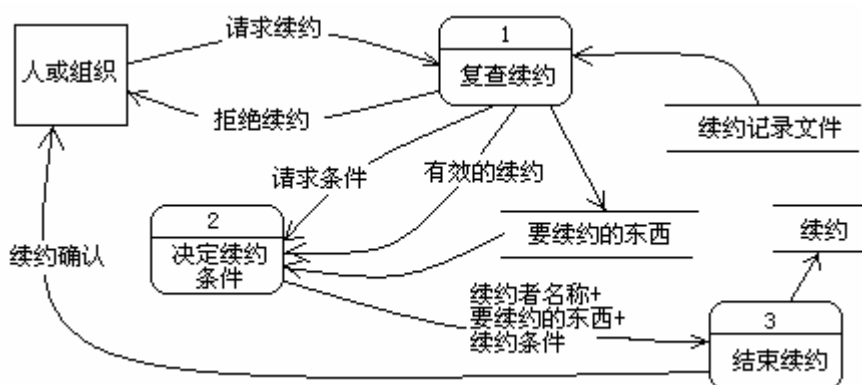
现在让我们把问题看得更深入一些，我们已经讨论了在特定主题领域确定和复用需求模式的想法，但是我们如何才能够在最初的领域之外使用模式呢？

2. 跨领域的模式

第一眼看上去，图书馆续借和卫星广播许可证续约对事件的响应很不一样，它们的不同之处是它们来自于不同的领域。但是再仔细研究它们的响应，就会发现它们之间有比较大的相似之处，如果发现了相似之处，我们就有机会导出更抽象的模式，可以适合更多的其它领域。

“续借”和“广播许可证”都是要续约东西，但其中有很大的相似之处：每件要续约的东西都有唯一的标示符、标准的续约周期以及续约费用。

下图是我们对这两个业务事件的处理策略进行抽象所得到的结果。我们是用抽象来确定共同的特征，这意味着透过事物的表象来发现有用的相似之处或者分类。它 also 意味着可以忽略一些特征以发现共同之处。



忽略实际物品和主题领域，我们就可以把注意力集中在两个不同系统所执行的底层操作上，我们这样做是为了发现相似之处，以便于利用。

确定和使用模式的技能与一些能力有关：

- 从不同的抽象层次来看待工作的能力；
- 按不同的方式进行分类的能力；
- 发现望远镜与注满水的玻璃半球都是放大镜的能力；
- 指出显然不同的事情之间相似之处的能力；
- 以抽象的方式来看问题的能力。

五、领域分析

领域分析是对一个主题领域的通用知识进行调查、记录并规范化的活动。可以把领域分析看成是非项目系统分析，它的要点是了解有关的业务策略、数据和功能，而不是构建什么东西。在该领域所获得的知识，将用于该领域内任何构建产品的项目，并将得到复用。

领域分析工作进行的方式，与常规系统分析是一样的。也就是说与领域专家一起工作，已提取他们到目前为止还是自然的知识。把它们记录下来，以便其他分析师复用。这说明，常规的分析模型（事件响应模型、类图、状态模式、数据字典等）是最有用的，因为这些模型在分析工作中最常见。重要的是，要点不是重新发现已经存在的知识，而是复用知识的模型。

事实上到底哪些东西组成了“领域”总是个问题，只是简单的说“银行”、“保险”、“医学”是不够的，对领域的定义存在于它与其它领域的接口之中，定义一个领域是进入和离开的东西，而不是它的名字。我们来看本章第一个上下文图，进入和离开的数据流定义了这个领域的范围，而不是它的名字“顾客希望购买一件产品”。建立了这个边界之后，确定有用的业务事件、数据和功能就变得比较容易了。

领域分析是一项长期的任务，但只有得到的知识是可以复用的，它才是有意义的。在领域分析上的投资就像其它投资一样，必须有一个很好的想法，这才能使投资有回报。从领域分析来看，如果领域知识被应用于该领域的数个项目之中，投资就得到了回报，要记住，领域只是被复用的次数是没有限制的。

六、复用的趋势

为系统的某个方面建立一个模型以后，就会使这方面变得可视，一旦它们可视，就有机会被复用。在对领域可视化建模中，越来越倾向于用类来描述概念，所建立的模型被称之为分析模型，这也是面向对象分析的一种趋势。

对象的使用不断增长，使人们在谈论系统知识的时候更加正式和一致，这种一致性提高了对复用可能性的认识。如果我们以更一致的方式来表述我们的知识，我们就有可能更广泛的交流，就有机会不止一次的使用它。面向对象的分析有利于复用的另一个原因，使它导致了普通符号表示方法的一致。

早在 1993 年，第二届国际软件复用大会在意大利 Lucca 召开时，当时会上提交的大多数论文，是讨论代码、设计和结构的复用问题，很少有人谈论在开发周期的早期（比如需求分析）阶段进行复用的思想。在这以后，在面向对象模式方面有一些重大的突破，有代表性的，是 1995 年出版了一本对模式影响深刻的书籍：Gamma、Helm、Johnson 和 Vissides 的《Design Patterns: Elements of Reusable Object-Oriented Software, 1995》（中文版《设计模式：可复用面向对象软件基础》，机械工业出版社，2000 年 9 月）。

时过境迁，今天复用的实践正在往上游发展，架构设计的复用成为必须考虑的问题，需求的复用成为趋势，这种关注点上的改变带来了更大的回报。人们为此改变了整个开发思想，架构优先与基于用例的开发就是这种思想的体现，人们不再把分析、设计、实现作为孤立的工作，而是作为一个整体来看，这就是我们在后面要谈到的用例模块的概念。

今天，我们正开始把发现和管理可复用的模式作为一项真正的工作，这将会通过需求复用带来实实在在的好处。

第四章 从需求开发到架构设计

4.1 需求开发向设计规划的转化

有经验的项目经理和开发人员都知道把软件需求转化为健壮的设计和合理的项目规划的重要性。由于需求定义了项目预期的成果，所以你的项目规划、预测和进度安排都必须以软件需求为基础。

一、需求和进度安排

许多软件工程实行“从右到左的进度安排”，此时，规定了发行产品的具体日期而后定义产品的需求。当开发者要实现预期质量标准下所有要求的功能时，他们常常不能按时完成项目。在做出详细的规划和约定之前定义软件需求是更现实的。

然而，如果你在需求的哪些部分能适应进度安排的限制哪些部分不能适应进度安排的限制这一问题上还有商量余地的话，那么“从设计到进度安排”的策略是可以起作用的。

对于复杂的系统，软件仅是最终产品的一部分时，只有在产品级（系统）需求产生以后，才能建立高层的进度安排。然后，将系统需求分解并分配到各个不同的软硬件子系统中。从这一点上看，就可以以不同来源（包括市场、销售、客户服务以及开发）的输入为基础建立起一致的产品发行日期。如果存在进度安排的约束条件，那么具有交叉功能的开发小组必须在功能、质量和费用上作出合理的决策。

你可能考虑按阶段规划和提供项目资金。需求探索作为第一阶段将提供足够的信息，使你能为一个或更多的构造阶段进行现实的规划和预测。具有不确定需求的项目也可以从反复或渐增的软件开发生存期中得以改善。定义需求的优先级可以使你判断出哪些功能应包括在首发版中，哪些功能放到随后发行的版本中。

软件项目可能经常不能达到预定的目标，这是因为开发人员和其他项目参与者是拙劣规划者，倒不是因为他们是拙劣的软件工程师。主要的规划失误包括：忽略公共（用）的项目任务，低估了要花费的工作量和时间，没有考虑项目风险，并且没有考虑返工所需的时间。

正确的项目规划需要以下元素：

- 根据对需求的清楚理解来估计产品规模的大小。
- 根据历史记录了解开发小组的工作效率。
- 需要一张综合的任务列表以完整实现和验证每一特性或用例。
- 有效的预测和规划过程。
- 经验。

二、需求和预估

项目估计（预估）的第一步就是要把需求和软件产品规模的大小相联系。你可以根据文本需求、图形分析模型、原型或用户界面设计来预估产品的大小。虽然对于软件大小没有完善的度量标准，但以下给出了一些常用的度量标准：

功能点和特性点的多少。

图形用户界面（GUI）元素的数量、类型和复杂度。

用于实现特定需求所需的源代码行数。

对象类的数量或者其它面向对象系统的衡量标准。

单个可测试需求的数量。

所有这些方法都可用于预估软件的大小,但不管采用什么方法你都必须根据经验进行选择。如果你没有记录当前项目所完成的真正结果,并和你所预估的进行比较,利用那些知识来提高你的预估能力,那么你所预测的将永远是一种猜测。

积累数据是需要时间的,所以你可以把度量软件大小的标准与实际的开发工作量相联系。你的目标是建立可以使你从需求文档中预估整个软件大小的方程(等式)或者借助归纳等解决问题的方法。

另一种方法是从需求中预测代码行、功能点或图形用户界面元素的数量,并使用商业预估软件作出人员水平与开发进度的合理安排。

含糊的和不确定的需求必定会引起你在软件大小预估中的不确定性,从而导致你的工作量和进度安排预估的不确定。因为在项目的早期阶段,需求的不确定性是不可避免的,所以在进度安排中应包括临时的事件并要合理预算资金以适应一些需求的增加和可能的超限。一个合理的住房改造项目包括了对不可预料的偶然事件,在你的项目中是否也要如法炮制呢?

花一点时间考虑一下,哪些标准最适合你所主持的项目。应该意识到开发时间与产品大小或开发小组的大小没有线性关系。在产品大小、工作量、开发时间、生产率和人员技术积累时间之间是存在着复杂的关系。

理解这种关系可以防止你陷入进度安排或人员任务安排承诺的误区(以前没有类似项目成功完成过),否则,项目开发将不会成功完成。

三、从需求到设计和编码

需求和设计之间存在差别,但尽量使你的规格说明的具体实现无倾向性。理想情况是:在设计上的考虑不应该歪曲对预期系统的描述。

需求开发和规格说明应该强调对预期系统外部行为的理解和描述。让设计者和开发者参与需求审查以判断需求是否可以作为设计的基础。

不同的软件设计方法常常都会满足最终需求,而设计方法会随着性能、有效性、健壮性以及所采用的技术上的不同而变化。如果你直接从需求规格说明跳到编码阶段,你所设计的软件将会是空中阁楼,其可能的结果只能是结构性很差的一个软件。在构造软件之前,你应该仔细考虑构造系统的最有效的方法。考虑一下其它的设计方案将有助于确保开发人员遵从所提出的设计约束或遵从与设计有关的质量属性规格说明。

例如有这样一个项目,进行了完整的需求分析,建立了详细描述模拟摄像系统行为的8个变换过程的数据流程图。经过大量的需求分析后,我们并没有直接进行源代码的编写工作。而是以数据流程图为表示方法,创建了一个设计模型。我们立刻意识到模型中有三个步骤使用了相同的计算算法,另外三个使用不同的方程集,而剩下的两个步骤共享三分之一集合。分析模型代表了用户和开发小组对我们正在解决的问题的理解,而设计模型则描绘了我们应该如何构造系统。

通过设计期间的仔细思考,我们把核心问题简化了60%,把8个复杂的计算集合减少到3个。如果我们在需求分析之后立刻进行编码,那么在构造阶段必定会出现代码重复。但是,由于及早发现了可简化问题,我们节省了许多时间和金钱。设计上的返工比编码返工可能要效率高一些。

以需求为基础,反复设计将产生优良成果。当你得到更多的信息或额外的思想时,用不同的方法进行设计可以精细化你最初的概念。设计上的失误将导致软件系统难以维护和扩充,最终会导致不能满足客户在性能和可靠性上的目标。在把需求转化为设计时你所花的时间将

是对建立高质量、健壮性产品的关键的投资。

在设计产品时，产品的需求和质量属性决定了所采用的合适的架构方法。研究和评审所提出的架构是另一种解释需求的方法，并且会使需求更加明确。两种方法都围绕着这样一种思维过程：“如果我正确理解需求，那么这种方法可以满足这种需求。既然我手中有一个最初的架构（或原型），它是否有助于我更好地理解需求呢？”

在你可以开始实现各个部分需求前，不必为整个产品进行完整、详细的设计。然而，在你进行编码前，必须设计好每个部分。设计规划将有益于大难度项目（有许多内部组件接口和交互作用的系统和开发人员无经验的项目）。然而，下面介绍的步骤将有益于所有的项目：

- 应该为在维护过程中起支撑作用的子系统和软件组件建立一个坚固的体系结构。
- 明确需要创建的对象类或功能模块，定义他们的接口、功能范围以及与其它代码单元的
- 根据强内聚、松耦合和信息隐藏的良好设计原则定义每个代码单元的预期功能。
- 确保你的设计满足了所有的功能需求并且不包括任何不必要的功能。

当开发者把需求转化为设计和代码时，他们将会遇到不确定和混淆的地方。理想情况下，开发者可沿着发生的问题回溯至客户并获得解决方案。

如果不能马上解决问题，那么开发者所做出的任何假设，猜想或解释都要编写成文档记录下来，并由客户代表评审。如果遇到许多诸如此类的问题，那么就说明开发者在实现需求之前，这些需求还不十分清晰或具体。在这种情况下，最好安排一两个开发人员对剩余的需求进行评审后才能使开发工作继续进行。

4.2 构造弹性软件架构

决定软件产品质量最重要的因素是软件架构，在架构设计中对我们最严峻的挑战是，我们如何合理的组织技术方案，把人和任务作为一个重要的因素进行考虑，使整体上高的投资回报率成为可能，所以我们的思维集中在两个问题上，第一问题，我们设计的架构如何确保以低的开发成本达到高的质量要求。第二个问题，如何避免需求变更或者后期升级，造成产品开发成本的大幅度上升。

从这个观点上说，一个好的架构要确保不同类型的关注点互相独立，当其中一个发生改变的时候，不至于影响系统的其它部分。架构师应该致力于标识系统的关键用例来构建这个架构，通过分析这些关键用例，可以创建一个**弹性架构**，也就是说各个不同类型的关注点保持独立，而系统中的一部分发生变化时，对其余部分的影响要最小。架构设计也必须满足性能、可靠性等系统的关注点。

架构将在系统早期的、关键性的第一个版本中展现，这是一个可执行的版本，我们称之为**架构基线**版本。在建立架构基线之前可能要花费几个迭代，但完成以后，将会证实你的假设、以及系统开发方法，也就可以降低风险，基于这个架构，其它部分的开发速度将会大大加快。

一、好的架构的特点

什么是好的架构？如何构建好的架构呢？这是每一个软件架构师时时在问自己的问题。毫无疑问，一个好的架构必须满足性能、可靠性等系统级的关注点，它必须是易于理解的，以便能跟踪出架构中哪一部分实现了哪些需求或者用例。每个类（或者包）都必须扮演已定义的角色，并且很好的履行这个角色全部的、而且仅仅属于这个角色的职责。职责应该是少量的，并且在类之间没有重复。

好的架构必须使每个关注点互相分离,尽可能使系统一部分的改变不至于影响到其它部分,即使有一定影响,也要清晰的识别出哪些部分需要改变,如果需要扩展架构,影响应该最小化,已经可以工作的部分应该可以继续下去。

1, 分离功能性需求

一般的希望保持功能性需求之间是分离的,功能表明了不同最终用户的关注点,并且可能互相独立的发展,所以不希望一个功能的改变会影响到其它。功能性需求一般是站在问题域的高度来表达的,因此很自然的希望系统特定功能从领域中分离出来,这样,就便于把系统适配到类似的领域中。另一方面,一些功能需求会以其它功能需求扩展的形式来定义,这样更需要它们互相独立。

2, 从功能需求中分离出非功能性需求

非功能性需求通常标识所期望的系统质量属性:安全、性能、可靠性等等,这就需要通过一些基础结构机制来完成,比如,需要一些授权、验证以及加密机制来实现安全性;需要缓存、负载均衡机制来满足性能要求。通常,这些基础结构机制需要在许多类中添加一小部分行为(方法),这就意味着与基础结构机制实现的一点变动都会造成巨大的影响,因此,要使功能需求与非功能需求之间保持分离。

3, 分离平台特性

现在的系统运行在多种技术之上,比如身份验证的基础结构机制就可能有許多可选的技术,这些技术经常是与厂商有关的,当一个厂商把它的技术升级到一个新的、更好的版本的时候,如果你的系统适紧密依赖于这项技术前一个版本的,那么进行升级就并不那么容易,所以要使平台特性与系统保持独立。

4, 把测试从被侧单元中分离出来

作为完成一项测试的一部分工作,你必须采用一些控制措施和方法(调试、跟踪、日志等),这些控制措施是保证系统运行流程符合测试要求的规程。这些方法是为了在系统执行的过程中提取信息,以确认系统确实是按照预期的测试流程执行的。

这些控制措施和方法,经常需要在测试过程中向系统内插入一些与系统一起运行的代码,在擦拭完成以后这些代码将会被删去,因此,希望把测试的实现与被测的系统分离开来。

二、建立架构基线的步骤

良好的架构必须尽早建立,技术在理论上,都没有办法通过重构等增量技术,把一个糟糕的架构改造成一个好的架构,在实践中就更难了。事实上如果重构的成本超过了管理者所能接受的程度,他就宁可简单的重写代码而不是重构。所以,一个良好的架构需要在建立忱本很小的时候建立起来,事实表明,优先架构会获得良好的投资回报,它减少了项目执行过程中的重新设计和做无用功。如果已经建立了一个良好的架构,就需要持续的进行重新评估,并且做一些必要的完善和重构。

1, 架构基线

架构是最终系统的一个早期版本,也称之为架构基线。架构基线是整个系统的子集,我们称之为**骨架系统**(skinny system)。这个骨架系统包含了项目结束时的“完整”系统所具有的模型的一个版本,它包含了相同的子系统、组件和节点的“骨架(skeleton)”,但并非所有的“肌肉(musculature)”都已齐全。

不管怎么说,骨架系统确实具有行为,并且是可执行的代码,可能会需要对结构和状态作某些细微的改变,在精化阶段或者架构设计迭代结束的时候,我们就可以得到一个稳定的架构了。

尽管骨架系统（架构基线）通常只包含 5%~15% 的最终代码，但他已经足够验证你所做的关键设计了，更为重要的，你必须确认骨架系统能够成长为一个完整的系统，应该有一个书面的架构描述文档，但现在，这个文档必须通过架构基线来验证和确认。

2，用例驱动的架构基线

架构基线是由关键用例子集驱动建立起来的，我们称这个子集为架构重要用例，在你提取出这些架构重要用例之前，必须尽可能收集存在的信息，以识别出系统所有的用例。注意：识别用例于描述用例并不一样，识别是对系统需要做的事情进行界定、探索 and 发现。描述用例则是对用例中的流程和步骤进行细化，描述用例会贯穿项目的整个生命周期，而识别用例则必须尽可能尽早的完成。

在这些识别出的用例中，确认哪些是最重要的，所谓“重要”，意味着它们组合在一起，可以覆盖所有需要作出的关键决策：

- 演练系统的关键功能和特性。
- 涵盖大部分的功能性、基础结构、平台特性等方面的风险。
- 突出系统中一些高复杂性和高风险的部分。
- 是系统剩余部分的基础。

架构重要用例列表应该包含应用用例和基础结构用例，要注意对其中一些具有技术相似性或者交互相似性的用例，可以选择有一个用例作为代表，只要解决了其中的一个用例，就可以解决其它用例了。

当挑出了架构重要用例以后，就可以分析它们当中的关键场景，通过分析用例场景，更好的理解系统需要做什么以及系统的元素是如何交互的，通过对系统的理解，就可以定义和评估架构，这个过程将不断迭代的形成一个稳定的架构。

稳定就意味着系统关键风险已经被解决，并且所做的决定可以基本上满足着手开发系统剩余部分的需要，这个架构不仅受架构重要用例的影响，还受使用的平台、必须集成的遗留系统、标准和方针、分布性需求、所使用的中间件和框架等等的影响。通过用例，可以评估所做的选择是否足够，并且发现哪些方面还需要完善。

3，迭代地建立架构基线

对于一个复杂系统，最终建立一个稳定架构之前需要经过几个迭代，由于这些迭代聚焦于开发架构，有时也被称之为架构迭代，或者称作精化迭代（在统一过程中的术语）。

每次架构迭代中必须处理所有的架构关注点，虽然不可能在每次迭代中处理所有的关注点，但需要全面的考虑它们，每次迭代过程都产生一个增量，解决一部分架构关注点。

迭代一直需要持续到所有的架构关注点都已经解决，在迭代结束时所得到的早期版本（骨架系统）需要经过测试和运行来证实结果。伴随着架构基线的产生需要一个架构描述文档，这个文档将成为后期开发小组工作的指南，也是后续迭代中必须遵循的依据。

架构描述还必须经过评审，已确定架构是否合理，这个描述文档还应该附上一张修定表已说明历史的演变，它同时也说明了重要的决策。在架构迭代中，由于需要作出决策，所以进展一般是比较缓慢的，一旦完成了架构迭代，后续的生产率就会明显提高，所以投入到架构迭代中的时间是非常值得的。

4.2 系统设计的应用架构策略

事实上所有的信息系统都是一个应用架构，不同的组织使用不同的策略来确定应用架构，下面介绍两种最常用的方法。

一、企业应用架构策略

在企业应用架构策略中，组织开发一个企业级的信息技术架构，所有的后续开发项目，都应该遵循这个架构，这个架构定义了以下内容：

- 1) 约束网络、数据、接口和过程技术以及开发工具（包括软件、硬件、客户端与服务端）。
- 2) 集成已存在系统和技术、或者新系统到应用架构中的策略。
- 3) 连续地检查应用架构的正确性和适应性的不断进行的过程。
- 4) 研究新技术并推荐将其纳入应用架构的不断进行的过程。
- 5) 对已经实现的应用架构修改请求的分析过程。

初始的企业应用架构通常是作为独立的项目开发的，或者是作为战略信息系统规划项目的一部分开发的。对应用架构的维护工作，通常由一个常设的信息技术研究组担负。在认可了应用架构以后，期望每个信息系统开发项目都根据这个架构使用或者选择技术。在大多数情况下，这会极大的简化系统开发或者架构阶段的工作。

二、战术应用架构策略

如果没有企业级的应用架构，每个项目就需要定义自己的架构。这种架构形式对于开发人员使用新技术有更大的自由度，不过需要进行三方面的可行性研究：

- 1) 技术可行性研究：对该技术的成熟度的度量，该技术对于正在设计的项目适应度的度量，或者该技术与其它技术以其工作能力的度量。
 - 2) 运行可行性研究：企业管理人员和用户对于该技术的熟悉程度，以及管理人员和支持人员对于该技术的熟悉程度的度量。
 - 3) 经济可行性：使用该技术从经济的角度是否合算的度量。
- 这些度量对于架构风格的选择是非常有意义的。

4.3 模块化架构设计策略

一、模块化设计的概念

模块：

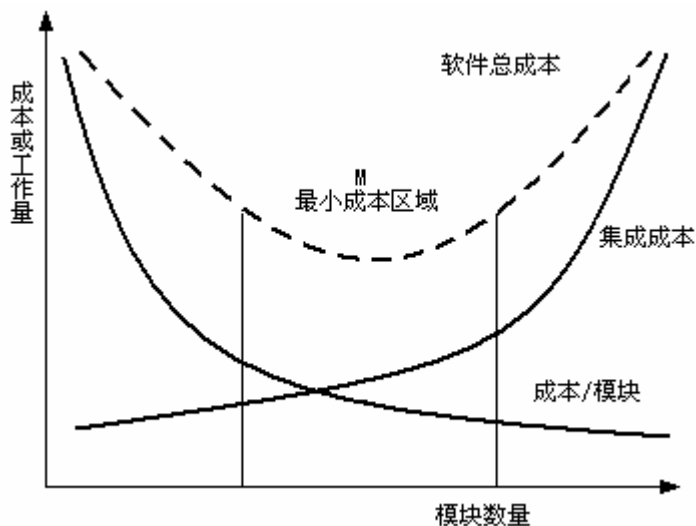
模块是数据说明、可执行语句等程序对象的集合，是单独命名的并且可以通过名字来访问，例如过程、函数、子程序、宏等。

模块化：

软件被划分成独立命名和可独立访问的被称作模块的构件，每个模块完成一个子功能，它们集成到一起可以满足问题需求。

利用模块化解决方案的注意事项：

我们可以这样来思考模块化对软件工作量和成本的影响。实际的情况见下图，随着模块数量的增加，开发成本减低，但是系统集成的成本增加，所以最小成本的区域在一个合适的区间。



也就是说，模块并不是越多越好，只有模块数量适当的时候，总体成本才可能下降。

二、实现模块化的手段

1，抽象：抽出事物的本质特性而暂时不考虑它们的细节。

2，信息隐蔽：应该这样设计和确定模块，使得一个模块内包含的信息（过程和数据）对于不需要这些信息的模块来说，是不可访问的。

模块独立性问题：

1，模块独立是指开发具有独立功能而且和其它模块之间没有过多的相互作用的模块。

2，模块独立的意义：

- 1) 功能分割，简化接口，易于多人合作开发同一软件；
- 2) 独立的模块易于测试和维护。

3，模块独立程度的衡量标准：

- 1) 耦合性：对一个软件结构内不同模块间互连程度的度量。
- 2) 内聚性：标志一个模块内各个处理元素彼此结合的紧密程度，理想的内聚模块只做一件事情。

耦合分类：

1，无任何连接：两个模块中的每一个都能独立地工作而不需要另一个的存在（最低耦合）。

2，数据耦合：两个模块彼此通过参数交换信息，且交换的仅仅是数据（低耦合）。

3，控制耦合：两个模块之间传递的信息有控制成分（中耦合）。

4，公共环境耦合：两个或多个模块通过一个公共环境相互作用：

- 1) 一个存数据，一个取数据（低耦合）；
- 2) 都存取数据（低--中之间）。

5，内容耦合（一般比较高）：

- 1) 一个模块访问另一个模块的内部数据；
- 2) 两个模块有一部分程序代码重叠；
- 3) 一个模块不通过正常入口而转移到另一个的内部；
- 4) 一个模块有多个入口（意味着该模块有多个功能）。

内聚分类：

- 1, 功能内聚: 一个模块完成一个且仅完成一个功能(高)。
- 2, 顺序内聚: 模块中的每个元素都是与同一功能紧密相关, 一个元素的输出是下一个元素的输入(高)。
- 3, 信息内聚: 模块内所有元素都引用相同的输入或输出数据集合(中)。
- 4, 时间内聚: 一组任务必须在同一段时间内执行(低)。
- 5, 逻辑内聚: 一组任务在逻辑上同属一类, 例如均为输出(低)。
- 6, 偶然内聚: 一组任务关系松散(低)。

关于耦合性和内聚性的设计原则:

- 1, 力争尽可能弱的耦合性: 尽量使用数据耦合, 少用控制耦合, 限制公共环境耦合的范围, 完全不用内容耦合。
- 2, 力争尽可能高的内聚性: 力争尽可能高的内聚性, 并能识别出低内聚性。

4.4 面向构件的方法

一、面向构件的方法简述

作为弹性架构的最初实现方法, 就是面向构件的方法, 构件也称为组件, 它包含了许多关键理论, 这些关键理论解决了当今许多备受挑剔的软件问题, 这些理论包括:

- 1, 构件基础设施
- 2, 软件模式
- 3, 软件架构
- 4, 基于构件的软件开发

构件可以理解面向对象软件技术的一种变体, 它有四原则区别于其它思想, 封装、多态、后期绑定、安全。从这个角度来说, 它和面向对象是类似的。不过它取消了对继承的强调。

在面向构件的思想里, 认为继承是个紧耦合的、白盒的关系, 它对大多数打包和复用来说都是不合适的。构件通过直接调用其它对象或构件来实现功能的复用, 而不是使用继承来实现它, 事实上, 在我们后面的讨论中, 也会提到面向对象的方法中还是要优先使用组合而不是继承, 但在构件方法中则完全摒弃了继承而是调用, 在构件术语里, 这些调用称作“代理”(delegation)。

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

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

构件方法并不是某种新的思想, 它能不能实现, 依赖于“构建基础设施”是不是能够创立, 这种基础设施的最大特征, 就是需要建立一个对语言 and 平台不敏感的通用标准, 构件的想法值得深思。最近几年又兴起了面向服务的架构(Service Oriented Architecture, SOA), 它能帮助企业有效地使用 IT 资源, 使 IT 系统灵活配合业务需求。为了使客户能够更加简单地实现向面向服务架构的转变, 现在提出了一种新的服务构件模型。它提供了一种统一的调用方式, 从而使得客户可以把不同的构件都可以通过一种标准的接口来封装和调用。这种服务构件的编程模型可以大大简化客户的编程, 提高应用的灵活性, 这就是面向服务构件的架构 SCA(Service Component Architecture, SCA)。

二、面向构件的软件模式

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

1、面向构件方法模式

面向构件技术的思想基础在软件复用，技术基础是根据软件复用思想设计的众多构件。面向构件将软件系统开发的重心移向如何把应用系统分解成稳定、灵活、可重用的构件和如何利用已有构件库组装出随需而变的应用软件。

基于面向构件的架构可以描述为：系统=框架+构件+组建。框架是所有构件的支撑框架；每个构件实现系统的每个具体功能；组建，可以视为构件的插入顺序，不同构件的组成顺序不同，其实现的整体功能也就不同。

面向构件技术将把软件开发分成几种：框架开发设计、构件开发设计、组装，如果用现代的工业生产做比喻，框架设计就是基本的生产机器的开发研究，构件开发就是零件的生产，组装就是把零件组装成汽车、飞机等等各种产品。

面向构件的方法也存在一些必须解决的问题。

(1) 构件的质量

面向构件的软件开发技术将系统分解成不同的构件，因此，构件是否高效稳定势必影响整个系统性能。

(2) 标准化和知识产权限制了构件的复用

标准化和知识产权分别从技术角度和法律角度限制了构件的复用。如何让构件得以更好的储存复用，降低劳动重复劳动量应该从这两方面考虑。

(3) 应用框架技术还不够成熟完善

(4) 关于构件组建技术

要把珍珠串成项链，选择牢固可用的链子是重要的。同样的，如何把独立的构件组建成一个可用系统，组建技术同样也是举足轻重的。

2、面向构件开发的不足之处

(1) 系统资源耗费

从软件性能角度看，用面向构件技术开发的软件并不是最佳的。除了有比较大的代码冗余外，因为它的灵活性在很大程度上是以空间和时间等为代价实现的。

(2) 面向构件开发的风险

从细节来看，构件将构件的实现细节完全封装，如果没有好的文档支持，有可能导致构件的使用结果不是使用者预期的。比如，构件使用者对某构件的出错机制认识不够

三、开放式系统技术

专用软件是由单个供应商生产的不符合统一标准的产品。这些单个供应商通过版本更换来控制软件的形式与功能。但是谁这系统越来越复杂，当一个系统建立起来以后，往往更倾向于依赖于通用的商业软件，这种依赖往往成为内部软件复用的非常有效的形式。正是这种状态，我们需要讨论一下开放式系统技术这个问题。

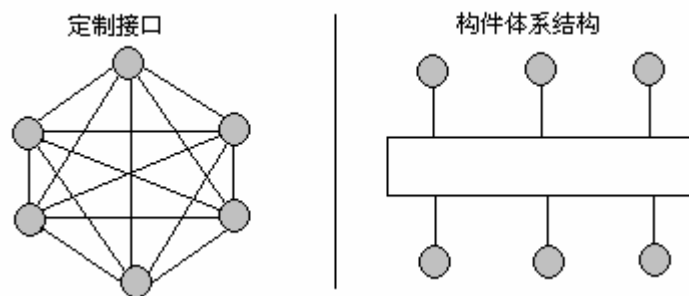
商业软件成为复用的有效形式，主要原因是规模经济的作用，通用的商业软件的质量，往往超过终端用户自主开发能力。

商业软件也可以在某个领域内实现专有，也就是提供应用程序接口（API）为应用软件提供服务。当软件不断升级的过程中，这些接口的复杂性可能超出用户的需要，这就需要有复杂性的控制。

一个解决方案就是实现开放式系统技术，开放式系统技术与专有技术有根本的不同。在开放式系统技术中，由供应商组织来开发独立于专有实现的规范，所有的供应商实现的接口都严格的一致，并且规定了统一的术语，这样就可以是软件的生命周期得以延长，这种开放式系统技术特别适合于面向对象的方法。

为了使商业技术能适应各种应用需求，对软件开发和安装就有一定的要求，这种要求称之为配置（profiling），适当的配置软件的嵌入也称为开放是软件的一个特点。

从架构的角度来看，不少系统结构具有大量的一对一接口和复杂的相互连接关系，这种模型被称之为“烟囱”模型，当系统规模增大以后，这种关系会以平方律的速度增加，复杂性的增加会带来相当多的问题，尤其是升级和修改越来越难以进行，而系统的可扩展性恰恰是开发成本的重要部分。



所以，系统的架构的一个重要的原则就是对软件接口定义一个已经规划的定义，为系统集成方案提供更高的统一性，软件的子系统部分要由应用程序接口来定义。这样，就削弱了模块之间的依赖性，这样的系统就比较容易扩展和维护，并且支持大规模的集成。

4.5 高层设计中的架构分析

架构设计并不是简单的把分析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。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，事实上非功能需求就是质量需求，这些信息对于架构设计来说，是最值得关注的。

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。

- 辨识和分析影响架构的非功能性需求。
- 对于那些具有重要影响的需求而言，分析可选方案，并做出处理这些影响的决定，这就是架构决策

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。

4，从需求文档中获取架构因素

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 <ul style="list-style-type: none"> 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做出合适的选择。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。

4.6 高层架构设计中的层模式

一、层模式的问题与机会

层模式是构造弹性架构的基础, 好的架构几乎都是在层这个模式基础上建立起来的。分层不是目的, 分层必须把分离性与易变性、灵活性结合起来, 这也是设计层模式有挑战性的问题, 也是最近几年最吸引人的研究领域之一。

1, 问题:

- 如果系统的许多部分高度的耦合, 源代码的变化将波及整个系统。
- 如果应用逻辑与用户接口捆绑在一起, 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。
- 如果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, 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。
- 在系统的不同部分高度的耦合的情况下, 难以对不同开发者清晰界定各自的工作界限。
- 如果高度耦合混合在系统的各个方面, 则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。

2, 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。

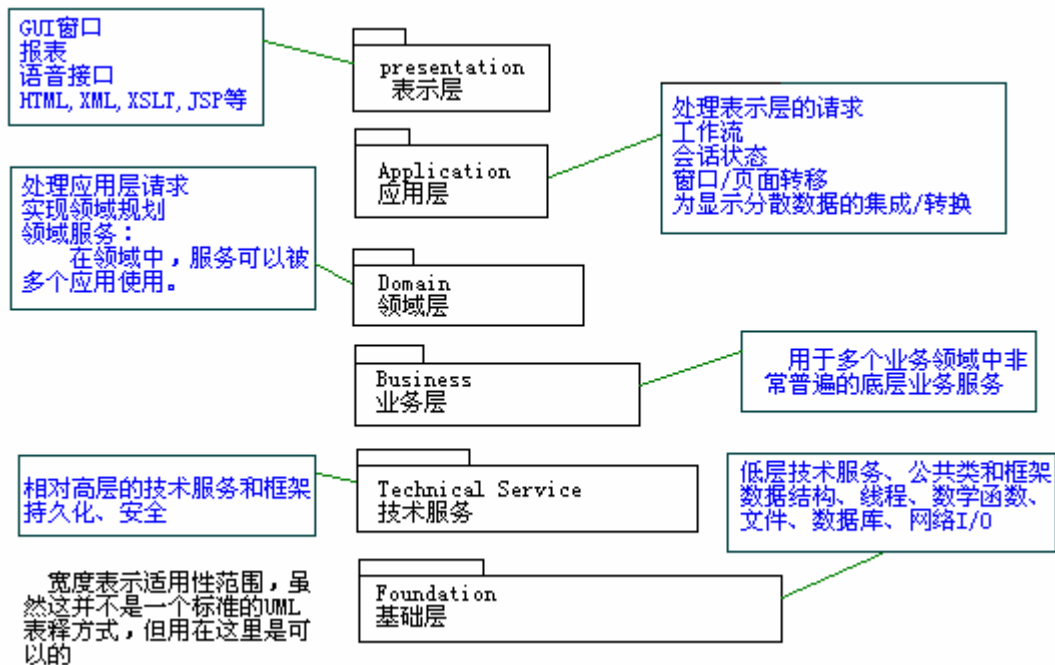
作为层架构的模式, 我们必须提到框架 (framework) 的概念, 一般来说框架具有如下的特征:

- 是内聚的类和接口的集合, 它们协作提供了一个逻辑子系统的核心和不变部分的服务。
- 包含了某些特定的抽象类, 它们定义了需要遵循的接口。
- 通常需要用户定义已经存在的框架类的子类, 是客户得以扩展框架的服务。
- 所包含的抽象类可能同时包含抽象方法和实例方法。

- 遵循好莱坞原则：“别找我们，我们会找你。”就是用户定义得类将从预定义的类中接受消息，这通常通过实现超类的抽象方法来实现。

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，或者说选择一小组有意义的元素来传达主要的思想。

二、层模式的设计原则

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。架构设计的重点，是从质量保证的角度，寻求架构解决方案，只要可以考虑如下一些策略。

1. 协作：

在架构层面上，有两个设计上的决策：

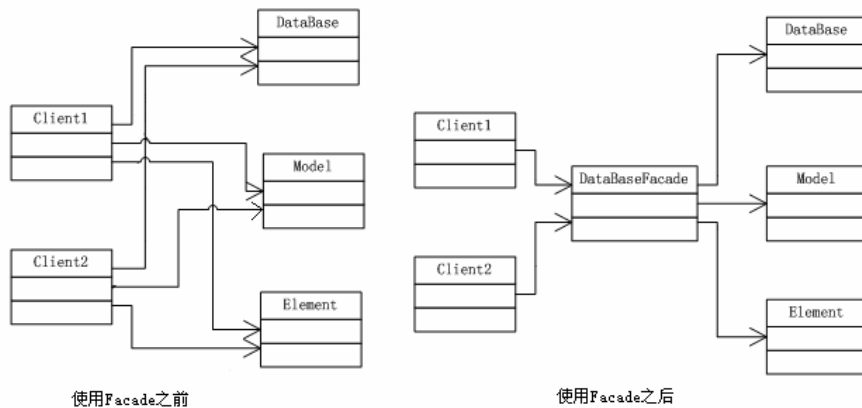
- 什么是系统的重要部分？
- 它们是如何连接的？

在架构上，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。

2. 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。



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。

3. 通过观察者模式（Observer）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。

4. 在不同的层上模糊集成员

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”、“底层”、“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。

5. 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。

4.7 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，事实上，SOA 是对构建弹性架构强烈的需求背景下发展起来的。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本地维护自己的合约, 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, 在与可以为企业领域建模, 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、灵活的架构类型，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。

第五章 用例驱动与基于方面的架构设计

软件开发需要太多的问题需要关注，当系统设计的时候，必须处理和平衡许多困难的关注点，系统如何达到预期的功能？如何达到性能和可靠性的要求？如何处理平台特性等。在设计的时候，我们会发现许多混合在一起的代码，对象中的很多内容都不是实现预期必须达到的要求，所以必须对设计进行改进，使其更加模块化。

把问题分解到更小的部分，在计算机科学中被称之为关注点分离，理想情况下，希望把不同的关注点清晰的分离到不同的模块中去，并且能够互相独立，也就说说把模块当成一个个关注点的集合，一个个的深入研究和开发，然后再把这些软件模块组合到一起

成功的关注点分离必须尽早开始，依照涉众的关注点来收集系统的需求，但当关注点很多的时候，任何一个组件都没有办法满足要求，这就构成了影响多个构件的横切关注点，由于面向对象的方法没有办法使关注点分离，这就造成了面向对象方法的局限性。

构件技术虽然解决了易扩展的问题，但对于不同涉众的关注点，仍然没有很好的方法有条理的解决这个问题，由于这种冗余代码段存在于许多操作和类中，一个功能的改变可能会影响很多类。这种冗余被称之为“横切关注点”。

尽管构件技术在采用层次化风格建立复杂系统方面表现优秀，但无法使横切关注点分离一直保持到编码阶段，往系统中添加一个新的关注点（一组需求）将会变得很麻烦。

正是这些问题的存在，促使人们提出了面向方面（Aspect-Oriented）的方法，其基础是面向方面的编程（AOP），AOP 提供了一种手段，可以把横切关注点的实现代码分离出来，使程序更加容易理解和维护。面向方面的方法必须贯穿于从需求到分析、设计、实现和测试的全过程中，这就是面向方面的软件开发（AOSD），它的目标是，使功能性需求、非功能性需求、平台特性等许多不同的关注点更好的模块化，从而使它们互相独立，并且一直延伸到软件开发的全过程。

在软件产业化进程中，软件工程学是目前研究最活跃的领域之一，人们极力的从组织学、方法学、技术学、经济学乃至数学的领域，多角度全方位的研究和探索，新的思想层出不穷。下面的讨论有些正处于探索的进程之中，介绍这些思想对开阔我们的视野非常帮助。

从方法论的角度，面向方面的方法与已经存在的面向过程、面向对象以及面向构件的方法并不是相互矛盾甚至相互对立的，而是以一种新的思维方式和技术手段，研究处理关注点独立性的方法。面向方面的技术是今天强调弹性软件架构与骨架系统理念的自然结果，同时使产品线概念取得更大的技术支持，这些研究，对于提升软件整体的质量是非常有意义的。

5.1、用例驱动的分析模型

对产品用例的进一步分析，就产生了分析模型，分析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。分析模型是说明问题域里（对建模者来说）有意义的**分析类**，必须说明，用例虽然也是一个重要的分析工作，它是强调的概念的过程视图，而分析模型强调的是用例场景内概念的关系视图，所以，它的输入是用例的场景，也是一个对功能精化分析的过程。

一、分析建模：

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、分析的过程就是分析建模（这个域，指的就是问题域）。

建立分析模型的好处：

- 1, 通过建立分析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。

经典的面向对象分析的步骤，是把一个相关的领域，分解为单个分析类或者对象（是一个我们能够理解的概念）。分析模型是分析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。分析模型是不定义操作（方法）的一组类图来说明，它主要表达：

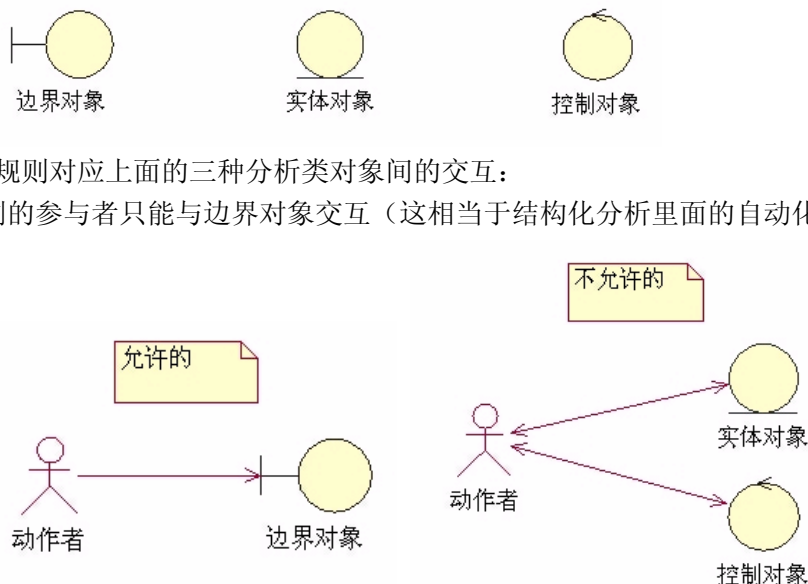
- 1, 分析对象或者分析类；
- 2, 分析类之间的关联；
- 3, 分析类的属性。

三种分析类：

1, 边界类：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, 实体类：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, 控制类：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。三种分析类的 UML 的图示如下：



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；

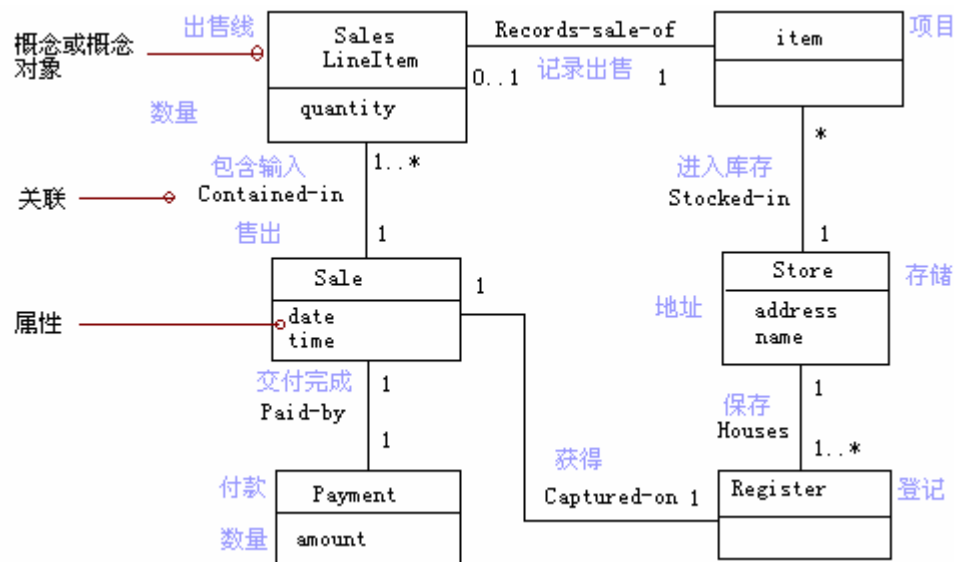
4, 控制对象可以和边界对象交互，也可以和实体交互，但是不能和动作者交互。

二、分析建模的简单例子

下面举个简单的例子，说明分析建模的基本概念。

1) 问题的描述

例如：两个分析类 Payment（支付）Sale（售出）在分析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，分析模型实际上是可视化了分析中的单词或分析类，并且为这些单词建立了分析类。也就是说，分析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) 分析模型不是软件组件的模型

分析模型视相关现实世界分析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分析模型中表述：

1，软件工件（窗口或数据库）

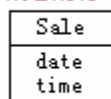
2，职责或者方法：方法是个纯粹的软件概念，在设计工作期间考虑对象职责是非常重要的，但分析模型不考虑这些问题，在这里考虑职责的正确方法是，给对象分配角色（比如收银员）。

4) 分析类

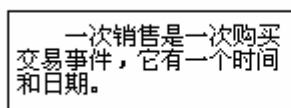
分析模型表示分析中的分析类或词汇，一个不太准确的描述：一个分析类就是一个观点、事务或者对象。比较准确的表达为：分析类可以按照它的符号、内涵和外延来考虑。

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

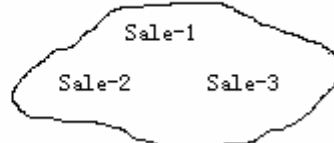
概念的符号



概念的内涵



概念的外延



三、分析类的识别

1, 识别分析类

我们的目标是在相关分析中创建有意义的分析类，比如说创建“处理销售”用例中的相关分析类。一个方法是通过建立一个候选的分析类的列表，来开始建立模型。但是，更多的是使用名词短语分析找出分析类的方法，然后把它们作为候选的分析类或者属性。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分析类（“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：把分析类分类，和词语分析一起使用。

一般来说，用大量细粒度的分析类来充分描述分析模型，比粗略描述要好。下面是识别分析类的一些指导原则：

- 不要认为分析模型中分析类越少越好，情况往往恰恰相反。
- 在初始识别阶段往往会漏掉一些分析类，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分析类的信息，或者因为分析类没有属性，就排除掉这个分析类。

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

2, 分析类识别的指导原则

1) 事物的命名和建模

分析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，也适用于分析模型的建模。

- 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：

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

2) 在识别分析类的时候一个常犯的错误

在建立分析模型的时候，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？还是单独的分析类 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，所以应该是个分析类。

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分析类，因为分析模型中，属性非常少见。

3，分析相似的分析类

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，在分析模型中，是不是应该用 **Register** 而不是 **POST** 吗？

我们应该知道，分析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** 是一个分析中常见的术语，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，关键是看你的分析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。

4，为非现实世界建模

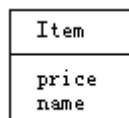
一些业务分析有自己独特的概念，只要这些概念是在业内被认可的，同样可以创建分析类，比如在电信业可以建立这样的分析类：消息（**Message**）、连接（**Connection**）、端口（**Port**）、对话（**Dialog**）、路由（**Route**）、协议（**Protocol**）等。

5，规格说明或者描述分析类

在分析模型中，对分析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

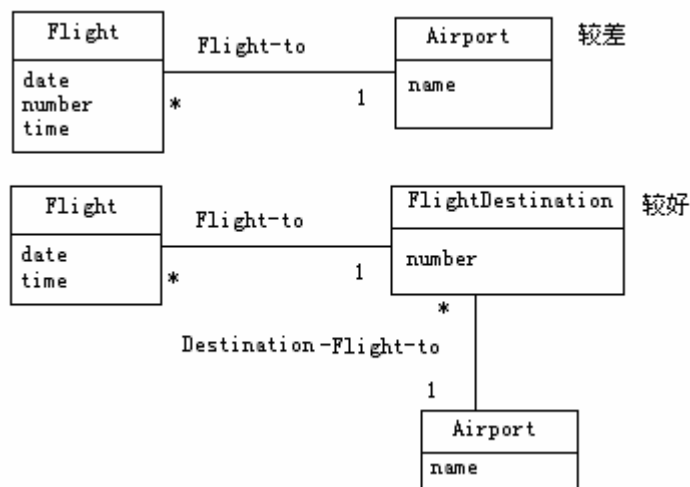
在下面的情况下，需要添加分析类的说明类：

- 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

作为分析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

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。



四、分析模型的属性

发现和识别分析类的属性，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。

1，有效的属性类型

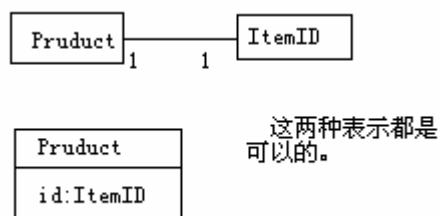
大部分属性应该是简单数据类型。当然也可以使其它的一些必要的类型，比如：**Color**（颜色）、**Address**（地址）、**PhoneNumber**（电话号码）等。

2，非原始的数据类型类型

在分析类中，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。



五、分析模型的关联

1，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 所捕获	
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	

2, 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分析模型,分而会使分析模型变的混乱,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。

3, 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。

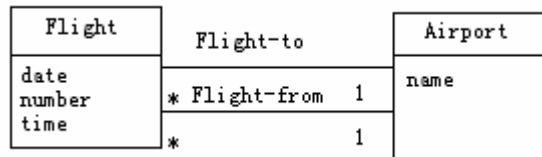
“1”表示一个。

“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。但在分析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,发现分析类比发现关联更重要,花费在分析模型创建的大部分时间,应该被用于发现分析类,而不是关联。

4, 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。

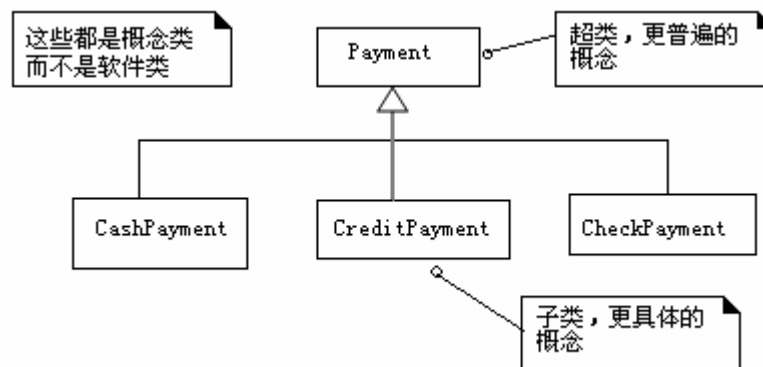


六、分析模型的泛化建模

泛化和特化是概念建模的基本概念，另外，分析类的层次，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。

1. 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。注意这里讨论的是分析类，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。在分析模型中，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。

2. 定义概念性超类和子类

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

1) 泛化和概念性类的定义

定义：一个概念性超类的定义，比一个概念性子类的定义更为普遍或者范围更广。

在前面的例子中，Payment（支付），是一个比具体的支付方法更为普遍的定义。

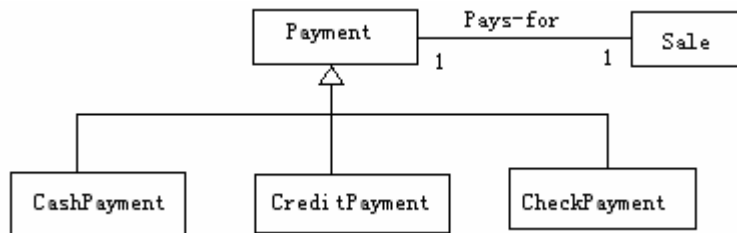
2) 泛化与类集

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



3) 概念性子类定义的一致性

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



4) 概念性子类集的一致性

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

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

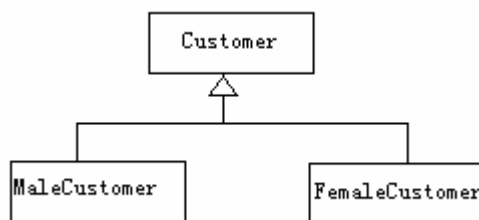
5) 什么是正确的概念性子类呢

从上面的讨论，我们可以使用下面的测试，来定义一个正确的子类：

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

6) 何时定义一个概念性子类

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



这样划分是没有意义的，因此我们必须讨论动机问题。把一个分析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。

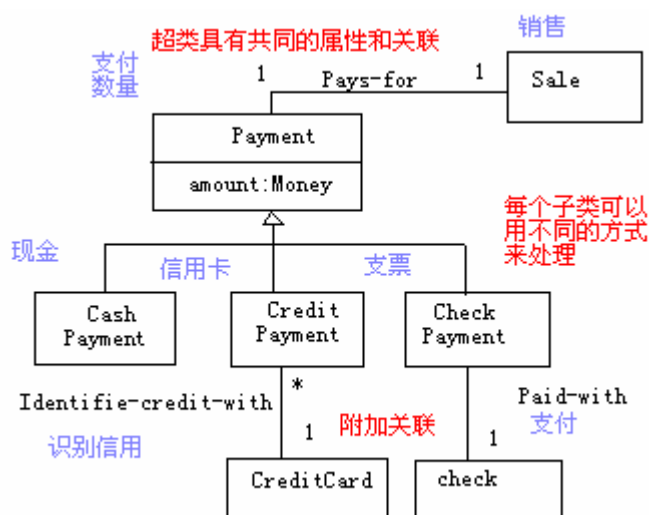
7) 何时定义一个概念性超类

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) 发现分析类的实例

Payment 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。

5.2 设计和实现模型

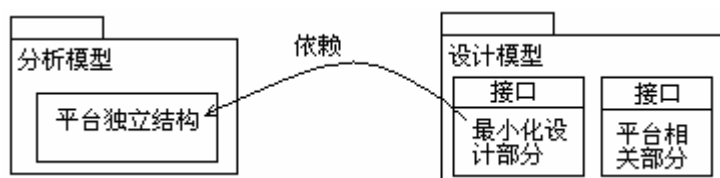
一、从分析模型到设计模型

设计模型是对分析模型的精练，它涉及了可执行平台的特殊细节，因此设计模型必须对许多重要的细节进行规划，它必须满足多种实现语言和技术的需求。它需要组织跨越多个处理节点间的系统部署，由于设计模型比分析模型更加复杂，因此，把分析模型与设计模型分开是明智之举，一般用分析模型来定义高层结构，用设计模型来细化结构、合并细节。

由于设计模型是对分析模型的细化，因此希望分析模型中的结构继续在设计模型中得以保持，这也意味着在设计元素结构中的包与分析元素结构中的包相对应，同时也可以发现设计模型中的类也是派生于分析模型中的类。同样，在分析阶段标识的用例切片也会在设计阶段细化。

事实上，分析模型中的结构都是与平台无关的，但是设计模型的结构可以分成两个部分：

- 最小化设计：对应于与平台特性无关的分析模型部分，是系统能力的基本表达。它包含相应的边界类、控制类和实体类。
- 平台相关部分：与平台技术有关的部分。



这种清晰的分离，对于摆脱平台的限制极其重要，它也将改进系统的可移植性。另外，在设计模型中还会包括在边界类、控制类和实体类之间传递消息或者数据的类，以及负责处理异常的类等，因此，设计模型会比分析模型有更多的设计元素。

上面提到的边界类、控制类和实体类，或者它们的组合，将演变为设计中的构件，这些构件将拥有来自于分析类职责的接口。我们称这些接口为**最小化设计接口**。

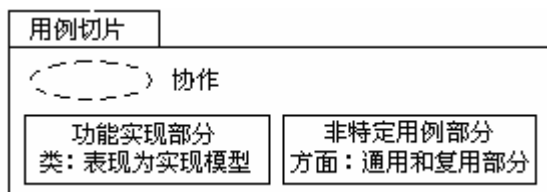
而对于平台相关部分，将拥有**平台相关接口**，比如 web 的 HTTP 接口，或者 Web Service

的远程接口等。

二、用例模型横切于模型

当我们实现一个用例的时候，必须表示出所需要的类，以及这些类的特性（属性、方法和关系），如果我们引进一个“用例切片”的概念，这个切片把**功能实现部分**放在一个模型中（设计模型），而**通用的和复用的部分**则保存在一种**非特定用例部分**中，把所有这些切片的叠加将构成系统的设计模型。这种方式，将会使问题比较清晰，避免不必要的混乱和遗漏。我们针对上面的用例切片的概念，可以进一步细化和规范。一个用例切片应该包含三个方面的内容：

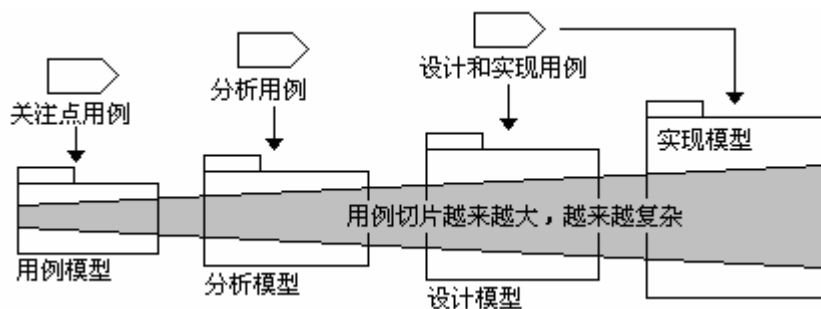
- **协作**：我们知道，用例是从系统的外部来描述系统的，但是协作是从内部视角来描述系统，在描述上可以表达类与类之间具体的协作关系。同时还必须对它命名，一般来说，协作的名字需要和用例一致，以防止理解上的歧义。
- **类**：这些类是特定于该用例实现，是本质上与其它用例分离的部分。
- **方面**：是该用例实现的时候对原有类的扩展，也是可能发生缠绕的部分。



用例切片可以层次化的组成设计模型，这个概念最好的比喻就是投影仪，切片类似于叠加在一起的幻灯片，设计模型类似于屏幕，我们可以独立的制作每个幻灯片，但需要一些协调工作，以保证它们内容的一致性，每个幻灯片的内容可以不同，但显示的位置应该在规定的地方。软件开发是围绕着模型的构建进行的，大概的步骤如下：

- 首先通过用例模型捕获涉众关注点。
- 然后把用例模型提炼为分析模型，这就是从高层视角对系统的描述。
- 通过设计模型来决定系统会运行于什么平台。
- 实现模型就是系统的代码实现。

设计系统应该逐个用例的进行，通过日益增多的模型来获取用例，提炼并且实现它。当完成一个用例的工作以后，就可以在一个包中（称之为用例模块）交付与这个用例相关的所有工件（包括分析、设计、实现和测试文档）。一个用例模块由每种模型的用户切片组成。用例模块可以单独开发，然后再合并成完整的系统。也就是说当构建一个系统的时候，需要逐个用例的进行构建，首先识别系统用例，然后一次一个地处理用例，详细描述它、分析它、设计它、并且实现它，当你在不同的模型中构建每个用例的时候，也会在不同的模型中更新相应的用例切片，这些用例切片如下图中阴影所示。



例如当工作于一个用例切片的时候（例如设计阶段用例切片），应该从这个用例切片的

上游（对应于分析阶段用例切片）开始做一些精化，添加一些内容。因此每个下游用例片都比上游切片更大也更复杂。模型也同样的下游模型比上游模型更大而且更复杂，这是因为模型也需要考虑越来越多的问题。

1，保持用例模型中的结构

通过各种不同的模型逐渐对用例切片进行精化，以完成对系统的开发，我们主要是通过这些模型的用例切片，使所有的下游模型都保持用例模型中的结构，这样做的理由如下：

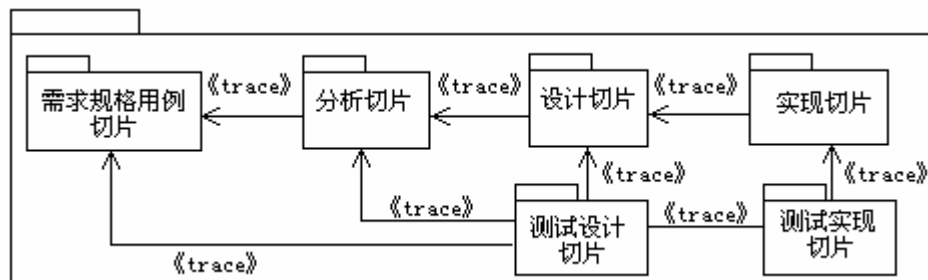
- **帮助我们理解下游模型：**可以通过查看相对应的用例，识别每个用例的切片到底完成什么功能。
- **便于在前后模型中转换：**软件开发并不是从用例到实现的线性过程，而是在前后不同的模型中转换和更新，保持模型中的结构，有助于在受控环境下无缝的转换模型，先完成一部分用例描述，接着转入分析来研究交互，然后回过头来完善用例，以澄清原来缺少的涉众关注点细节。也许在设计的时候，还可能回到需求来澄清该用例描述的特定需求，在建立系统架构基线的时候，也可能会前后切换。
- **保持模型一致：**如果下游模型与上游模型不同，您可能需要花时间去理解完全不同的结构，并且维护这种模型之间的映射关系，由于并没有一种形式化的方法解决这个问题，因此难以在模型中保持一致。

还需要说明的是，并不需要把设计结构引入到用例模型里面去，用例只是用涉众可以理解的方式结构化了他们的关注点，而设计将反映出涉众如何感知系统，用例驱动了设计模型，这就是为什么我们把这种开发方式称之为用例驱动开发的原因。

2，用例模块包括用例结构

既然是基于所有针对各个用例的不同模型的切片进行了工作，就可以把这些切片放到一个单独的包中，我们称这样的包为“**用例模块**”，这个模块包括了各种模型的切片以及它们的依赖关系，当开发一个由用例模块组成的系统的时候，我们可以把每个用例模块视为一个单独的项目，在硬盘或者某个中心开发库中，某根目录对应于用例模块，而子目录对应于用例模块中的每个切片。

下图是一个用例模块的示意图，而表示的<<trace>>的依赖关系，表明上游模型得出的一个下游模型存在着一些规则，作为开发团队必须遵循的开发指南和原则。这种用例模块，也可以成为软件产品线的一个元素。进一步抽取掉模块的领域相关部分，使它具备通用性，并且适当的命名，加入应用场景和使用案例，就可以发展成一个**用例模式**。这种用例模式对用例切片的复用极其有帮助。



在软件开发中，**可追溯性**是一个很重要的概念，可追溯性表明上游模型元素与下游模型元素存在着链接，这可以帮助你判断是不是所有的需求都已经实现，需求的变化会对什么产

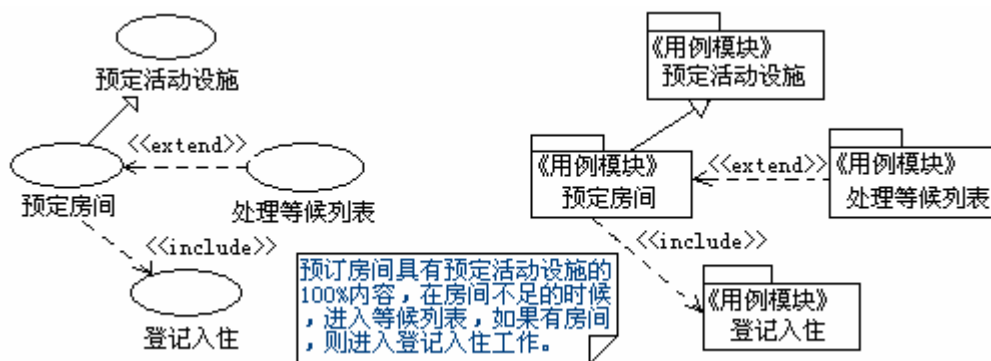
生影响，通常上游元素的修改会影响到下游元素。

如果上游和下游元素遵循着不同的结构化原则，结果就会花费很多精力维护模型间的可追溯性，这在大型项目中显得尤其困难，但在我们保持模型结构的原则下，可追溯性的维护工作将显著减少，这是因为模型本身就是基于相同结构的。

过去的设计方法，易变形和可维护性问题的解决，是在分析之后的设计过程中完成的，造成了设计模型与分析模型之间的可追溯性很差。但在面向方面的软件设计理念中，这个问题的解决是在分析的时候就提出来了，通过模型的一致性达到了问题解决模型的可追溯性，这就大大提升了软件设计的质量。

3，用例模块之间的关系

正如用例之间存在关系一样，用例模块之间也存在类似的关系，用例模块之间的关系于用例之间的关系相同，也就是泛化、包含与扩展。

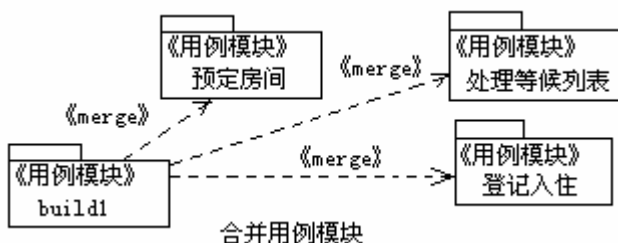


用例模块的关系可以用于两个互补的方面，从正向工程的视角，可以从用例之间的关系派生出用例模块之间的关系，换句话说，用例模块保留了用例之间的关系，具备了一致性。

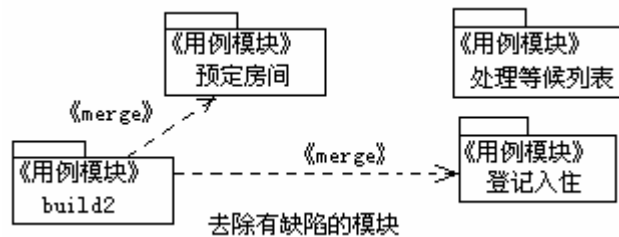
其次，用例模块之间的关系，可以作为检测访问违例的手段，也就是检查某个用例模块中的一个元素与另一个模块的另一个元素之间是不是存在非法关系。通过观察用例模块之间的关系，可以尽早发现违例，也可以尽早进行必要的修改。

4，合并和配置用例模块

完成了基于各个用例模块的工作以后，就可以把它们合并到各个发布版本中，假定我们为某次发布版本分配了三个用例模块，就可以有一个名字为 build1 的合成用例模块。



如果由于某种原因发现“处理等候列表”有许多缺陷，可以去除这个模块，构成一个发布版本 build2 先供用户使用，而我们单独对 build1 进行排查和测试。



在传统的做法中，是直接修改原有类的代码，可能会把废弃的代码留在系统中，也会使原有类引入更多的混乱，同时也可能引入更多的错误。

用例模块是从不同的视角描述系统，这种从一个模型到另一个模型构造的关系，有助于维护模型的一致性，也更容易指导下游模型元素的导出。请注意，不要逐个模型的构造系统，而应该逐个用例模块的构建，用例模块是软件开发工作的一个单位，它把各个不同模型中与某个用例相关的元素局部化于一个单一包中，可以采用迭代的方式独立的开发。

5.3 使用方面技术解决关注点分离问题

在软件开发中，使关注点保持分离是十分重要的，它可以帮助你复杂的问题分解为更小的部分，并独立的解决它们。当它发展为大型系统的时候，这是构建它的唯一方法。如果没有办法使关注点保持分离，随着系统的演化，复杂性将会不断增加，另一方面，通过保持关注点分离，系统也会变得更易于理解、维护和扩展。

新的模块化技术应该能够包括分析、设计、编码和测试的整个生命周期中，使横切关注点保持分离，要达到这种模块性，就需要有两个方面的技术：**关注点分离技术**，以及**关注点合成技术**。这两种技术的合理应用，是面向方面架构设计的关键。

面向方面的本质与用例驱动开发是相同的，也是首先使用用例为横切关注点建模，然后按照在类之上进行叠加的方式来设计用例，这种叠加方式称为用例切片和用例模块，接着使用方面技术来组合用例切片和用例模块以构建整个系统的完整模型。

使用用例切片和用例模块进行系统开发，能够对横切关注点进行清晰的分离，在演化和扩展中，使每个切片更容易复用，而且这种切片之间不会互相干扰，使用这一方法，将获得更好的可维护、可扩展、以及更高的性能。

事实上在我们架构设计中，一直在强调封装变化、基于扩展的设计等等，但面向方面把这些思想进一步提升，变成了一种方法论和技术实现的结合，而且，面向方面的编程（AOP），又使这种思想的实现变得容易和可能，但是，这种分离与合成的广泛使用，如果没有仔细的、经过慎重考虑的、无歧义的设计描述相结合，将会使整个产品结构变得极其混乱更加难以处理，所以我们必须从方面的视角，从分析到设计全面的、规范化的思考和描述问题，也就是说，面向方面的问题需要全方位的解决才可能发挥作用，这就需要仔细研究面向方面解决问题的基本思路。

同样，这种研究对于提升普通的语言系统设计质量也非常有意义。

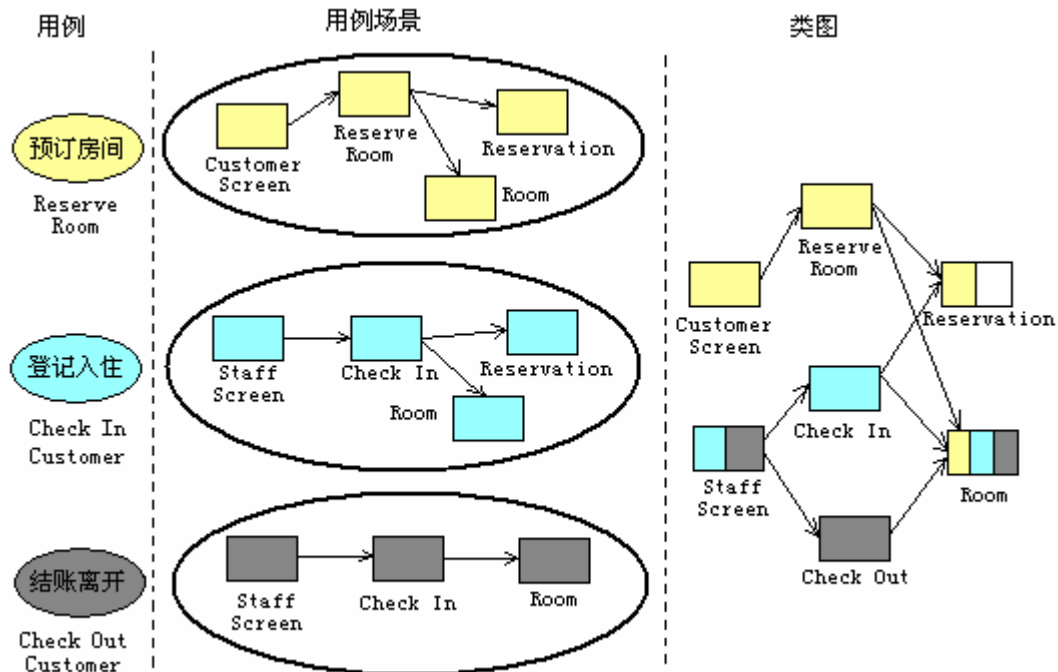
一、使关注点相互分离

尽管在分析的时候我们强调了关注点分离，但是，一旦进入设计，这种分离仍然会不可避免地造成交叠，我们下面来研究两种横切关注点及其问题：

1, 对等（peer）关注点

所谓对等关注点，就是关注点相互独立，其重要性也不能相互比较。比如 ATM 机中，存款、转账和取款都是对等关注点。又例如一个简单的例子，酒店管理系统预订房间(Reserve Room)、登记入住(Check In Customer)、结账离开(Check Out Customer)都是对等关注点。

对等关注点本身并不互相依存，但在系统中实现对等关注点的时候，就会出现明显的交叠，如下图所示。



这张图可以看出构件在保持对等关注点分离方面的局限性，结果就是结构的混乱状态和分散。这种交叠又包括两种情况：

1) 缠绕状态 (tangling): 一个构件包含满足不同关注点的实现（亦即编码），比如上图中，Room 包含了三个不同关注点的实现，这就意味着开发人员需要理解一组不同的关注点，构件也变得更加不易被理解。注意不要把缠绕状态与复用混为一谈，复用是指相同的代码和行为在不同的上下文中使用，而一个构件中缠绕着多个关注点，使这种模块更加难以复用。

2) 分散 (Scattering): 一个关注点的实现分散在多个构件中，比如：关注点登记入住(Check In Customer)，在四个构件中添加了额外的行为，如果需求发生变化，或者设计发生修改，就必须修改多个构件。更重要的，分散使系统内部组织不易理解，例如，不容易通过阅读一个（或者几个）构件的源代码来理解系统，如果一个关注点类发生变化，会有多个类需要修改，更不容易进行改进，对大型系统尤其如此。

2. 扩展 (extension)

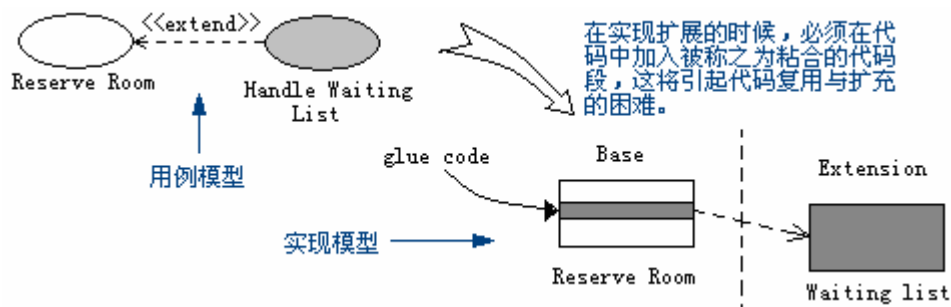
在用例的结构化分析中，有时候会引入扩展，这就是第二种需要研究的横切关注点。扩展是在基础(base)构件上定义的构件，用来表示附加的服务或者功能。

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

但是，虽然描述问题的时候这种扩展很有意义，但实现的时候却有问题。

下图把 Reserve Room 作为基础组件，在添加 Waiting list 扩展组件的时候，还需要在 Reserve Room 中增加一个代码段，来连接或者调用 Waiting list，这个代码段被称为粘合代码

(glue code),



结果，在发生更改的时候，这种粘合代码对原来的代码段形成很大的干扰，比如，如果需要添加新的扩展，在原来的程序中适当地地方就要加入对应的粘合代码，如果系统设计的不合理，系统的扩展性就会受到很大影响。

3. 分离关注点的解决方案

人们一直在研究关注点分离的技术，要达到这种模块性，需要两方面的基础：

关注点分离技术（Concern Separation Technique）：为了使关注点保持分离，必须对关注点进行建模和结构化，同时还需要一种在设计和实现阶段保持分离的技术。

关注点合成机制（Concern Composition Mechanism）：某些时候，需要把关注点合成，这需要一些执行自动化方法，比如需要的时候，转而执行扩展。

现有的类、模块等技术可以在一定程度上使关注点保持分离，但是出现贯穿多个类和组件的横切关注点的时候，就需要另外的方法来实现模块化了。

二、使用方面技术解决关注点分离问题

面向方面是使横切关注点更好的分离的一组技术，所以有时候也称高层关注点分离。而AOP提供了面向方面的实现基础，比较典型的是Java语言的一个扩展AspectJ，它提供了关注点分离和模块化的新构造，主要是：

Intertype（自动排版）：声明允许把新的特性（属性、方法及关系）合成到已有的类中。

Advices（通知）：提供了一种在AOP中通过pointcut（切割点）指定的扩展点上扩展现有操作的手段。

Aspects（方面）：只是一种构造块，用于组织Intertype声明和Advices。Aspects可以是其它Aspects的泛化。

AOP的合成机制使你可以在类的外部定义方法，这个概念在大多数人来说还是新的，但是如果不加限制的在现有操作中插入Advices，或者在现有类中加入Intertype声明，最终反而会得到一个难以扩展的、缝补出来的系统。

AOP提供的强大的合成机制，可以更好的分离关注点，更好的实现模块化。因此，必须认真构思，到底是把行为设计成类还是方面呢？也就是说，如何分清关注点是源于哪个方面或者类。这就需要在分析和设计中仔细思考

另一方面，不要以为只有使用AOP才可能使用面向方面的方法，这种方法的思想可以用任何语言完成，编程只是一种实现手段，仅仅从编程上思考问题是不可能设计出优秀的产品的。

三、通过叠加用例切片来构建系统

为了确保从分析、设计到实现全的过程中关注点保持分离，我们必须有一种形式化的方法来表达这种分离结构，并且能够利用这种表达方式无歧义的研究、调整以及实现。从整体上看，我们把整个系统分为元素结构与用例结构两部分，也就是说一个模型可以包括两个结构：

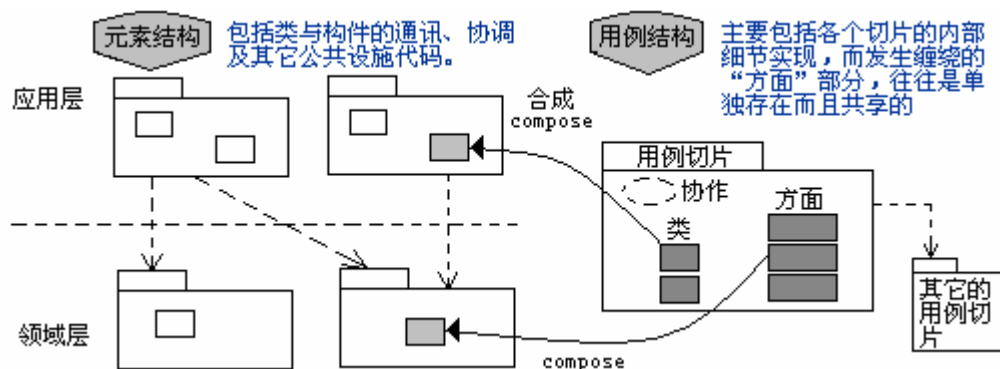
1) 元素结构

这个结构用来标识元素与组织元素。一个弹性架构的设计模型，是由层次化的元素组成（类、接口、包），这个层次结构可以称之为元素结构。

2) 用例结构

用来定义这些元素内容的结构。用例结构描述哪个用例切片（或者叠加图）放在第一位，哪个放在下一个等，这将会构成一个结构，它是用例切片的层次化结构，

我们可以使用例切片与元素结构相互分离，它构建在元素结构之上，用来定义扩展的内容，这种合成如下图所示。



在图的右边，可以看到用例结构，为简单起见，这里只列出了一个用例切片，它可以用构造型《use-case slice》表示。用例切片包含要合成（通过方面编织）到模型（包含到元素结构的引用）中的元素扩展，用方面（aspect）来表示。

正如图中所看到的那样，元素结构仅仅是一种标识元素位于模型何处的空盒子，它的具体内容（例如行为），将在合成的时候通过用例切片来填充。

如果说早期的概念，开发人员必须从不同的用例中收集每个类的职责，然后才能开发这个类的话，必然引发缠绕和分散，现在把用例结构从元素结构中分离，就可以使用例的分离得以保持。

四，使用对等用例保持分离

基于上面所讨论的分析方法，在对用例进行分析以及建立用例切片的时候，就需要非常清晰的提取出每个用例到底有哪些类是该用例独占的，哪些类只是部分定义的，需要表达成扩展（方面）。这样一来，我们就可以使用对等用例使关注点保持分离，但每个用例切片不一定拥有完整的类，而是在类中存在称之为类扩展（方面）的那一部分。在前期建立分析模型的时候，把这些信息都提取出来时非常有好处的。

我们还是讨论关于宾馆的例子，现在可以通过一个矩阵研究三个用例中的切片使用情况，我们得到了 ReserveRoom 切片、CheckInCustomer 切片和 CheckOutCustomer 切片，以及它们相应的分析类。

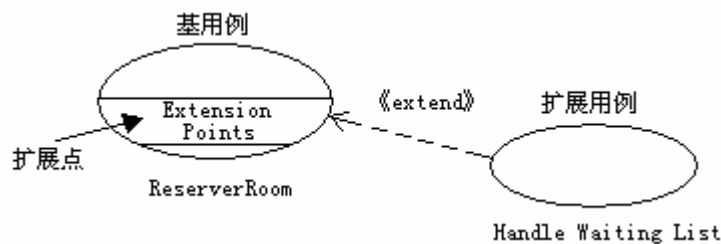
类 用例切片			顾客 屏幕	工作人 员屏幕	预订 房间	登记 入住	结账 离开	记录 信息	房间
			Customer Screen	Staff Screen	Resrver Room	Check In	Check Out	Reservati on	Room
1	预订 房间	Reserve Room	×		×			×	×
2	登记 入住	Check In Customer		×		×		×	×
3	结账 离开	Check Out Customer		×			×		×
缠绕状态			1	2, 3	1	2	1	1, 2	1, 2, 3

每个用例切片都包含针对用例实现的、只是其中一部分的类定义（也就是类扩展），如果你想得到完整的类定义，需要做的事情就是合并类扩展（把同名字的类合并在一起）。比如三个切片都有个 **Room** 扩展，那么可以合并成一个扩展的 **Room** 类。

五、使用扩展用例保持分离

也可以使用扩展使用例保持分离。假如需要在“预定房间”的用例中加入一个功能，当预定房间的时候如果没有空房间，则把客户加入等候列表，以此生成一个用例“处理等候列表”（Handle Waiting List），这时候可以使用扩展用例来实现，也就是说“处理等候列表”从基用例“预定房间”扩展。

基用例必须声明若干新的规则扩展点（Extension Points），扩展用例是在不改变基用例的前提下，向它的一组扩展点中添加行为。



扩展用例会插入一组活动，这组活动就是扩展用例事件流，它负责把客户放进等候列表，相对于基用例，是一个独立的关注点。

过去这种思考方式由于难以实现而不太常用，但在 AOP 中可以很好的实现这种模型，在 AOP 中，join point（连接点）是与扩展点相对应的概念，join point 是使用事先定义的语法（调用方法、异常等）说明程序执行流的一个点。而 pointcut 是比扩展点更大的概念，可以一次在多个类中定义多个扩展点（也就是 join point）的引用，这对于一些横切于多个类的基础机制（例如认证、日志等）特别有用。

在 AOP 中，一个扩展用例事件流将被实现为 advice（通知），它等价于用例切片中的术语“操作扩展”，这是对原有操作的模块化扩展，用于执行不同于操作主要职责的行为。

对于上面的用例，可以画出切片图如下，在元素结构中多了一个 **WaitingList** 类，它是 **HandleWaitList** 用例实现所需要的类。

类 用例切片			顾客 屏幕	工作人 员屏幕	预订 房间	房间	等待 列表
			Customer Screen	Staff Screen	Resrver Room	Room	Waiting List
1	预订 房间	Reserve Room			×	×	
2	处理 列表	Handle Waiting List	✓	✓	×	×	×
缠绕状态					1, 2	1, 2	1
在补充位置定义切割点			2	2			

表中 HandleWaitList 用例切片还有两个操作扩展，分别定义在类 CustomerScreen 和 ReserverRoom 中，如果想完成类定义，或者完整的类的操作，就需要合并这两个类的切片。

5.4 基于用例切片使对等用例保持分离

下面，我们需要对用例模块和用例切片的描述作更深入的讨论。在前面分析阶段产品建模中，我们已经对涉众关注点进行了正确的建模，接下来就希望在设计和实现阶段保持这种分离，前面我们已经引入一种新的“用例切片”（use case slice）的模块化单元来保持这种分离，用例切片包括每个用例实现特定的类，以及现有的类的扩展。

在这样的设计模型下，我们就会有一个元素模型，这个模型事实上是一个弹性架构，包括层、包、子系统的分级元素组织，而且具备最基本的组织和关系。另外我们还会有一个用例结构，利用用例切片在元素模型的元素上叠加各种行为。

过去，开发人员直接基于元素结构来实现系统，元素结构内包括各种行为，其结果就是关注点与这些元素互相缠绕。基于用例切片的概念，开发人员就可以互相独立的实现每个用例，而让开发环境来组合用例切片，构成设计模型中完整的元素集。对这个问题的深入理解，即使不使用面向方面的编程语言，也同样可以实现这些思想。

我们已经讨论过，基于用例可以在需求阶段使关注点保持分离，但这还不够，我们的问题是必须在设计和实现阶段继续保持这种分离。上面我们已经分析了，虽然对等用例之间并没有什么关系，但它们的实现会调用一些共享的类，或者在类的某些部分在用例之间共享和复用，这样就发生了缠绕。

用例切片是对模型中特定于某个用例的部分进行模块化，它使用方面作为合成机制，在这些共享的类的基础上，扩展了特定于某个用例的特性（属性、关系和操作）。

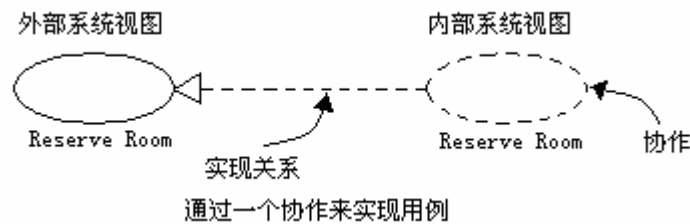
一、实现对等用例

从用例建模的角度，对等用例之间没有任何关系（即不存在包含、扩展与泛化关系），这样，就有可能把细化对等用例交给不同的人，但是，当实现对等用例的时候，会发现它们影响了相同的类（这也是称之为“对等”的原因）。这中间就会出现协调上的问题，所以，当开始实现对等用例的时候，我们仍然希望能够独立的、并行的工作。

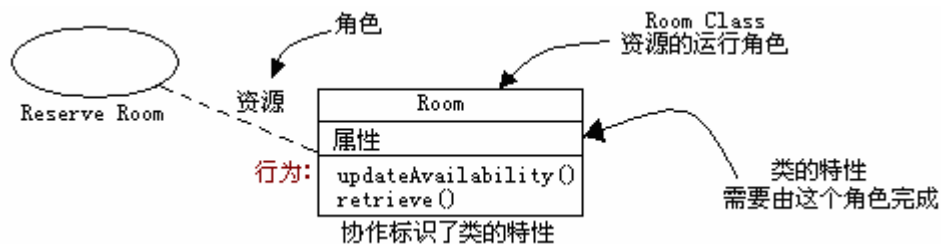
1、通过协作实现用例

用例的实现一般建模为**协作**，用例是从系统外部的视角来描述参与者与用例的交互，而并不描述系统内部的类是如何通过协作来实现用例的。协作定义了一组类的实例，通过协作进行建模，使它们相互协作完成任务，这个任务可以是具体应用的实现，也可以是基础结构的实现。所以用例和协作是事物的两个方面，用例是从外部视角描述系统，而协作是从内部视角来描述系统。

下图中“内部系统视图”代表一个实现，协作用虚线箭头表示，表示一个实现关系。



一个协作标识了扮演实现用例的的不同角色的类，例如下图的角色由 Room 类来扮演，协作需要类的一组特性（属性、方法及关系）来扮演这些角色，我们也可以在 Room 的基础上加入新的操作（行为），比如：更新提供数据（updateAvailability()）、取回数据（retrieve()）等。



实现一个用例，需要标识出参与实现的类及所需的角色，通过分析用例文档中的每一步骤，判断需要哪些类，为不同的类规定角色和职责，并且分析是如何调用其它类的实例的，也可以通过交互图表达消息传递的顺序。

2. 对等用例实现之间的交迭

在对等用例的情况下，如果有的类只是针对自己，是不存在交迭操作的。但是前面已经讨论过，以“登记入住”和“预订房间”两个用例讨论，其中有的类是针对自己的，比如 ReserveRoomHandler（预定房间处理类）和 CheckInHandler（登记入住处理类）。但是，也有的类是共享的，比如 Room（房间类），这样就构成了交迭（交迭类）。另外，Room 类的扩展中的“取回数据”操作（retrieve（））也将用于所有的协作，这也构成了交迭（或者称之为交迭操作）。

在没有面向方面的技术的时候，开发人员必须在实现每个类之前，整理它的扩展，这些类有些是原有的类，有的需要在其中添加新的特性。有的是新的类，原先还没有标识出来，因而设计人员就需要在设计模型适当的包中创建它。结果就使用例的实现变得分散，类模型变得混乱，正是这种状况，就需要通过一种方法，使特定于每个用例的部分保持分离。

二、使用例特定部分保持分离

为了使模型中的用例在实现的时候仍然保持模块化，需要引入一种新的模块化单元，它包含了特定于某个用例实现的元素，我们把这种模块化单元称之为用例切片。用例切片的特点是它的内容针对的是某个用例，是对模型（这里是设计模型）的切割或者切片。

我们已经说明了用例切片应该包括以下个内容：

- 描述用例实现的一个协作。
- 特定于该用例实现的类。
- 特定于该用例实现的对原有类的扩展。

为了把这个概念搞清楚，我们用“预定房间”这个用例来举例说明。对于每个用例而言，

在设计模型中都有一个用例切片，用例“预定房间”就有相应的“预定房间”用例切片，如下图所示。



它是一个特定类型的包，用构造型<<use case slice>>表示，用例切片的名字与对应的用例相同，协作的命名规范也是相同的，在具体的描述时，需要用图形把各个类之间的协作关系表达出来。

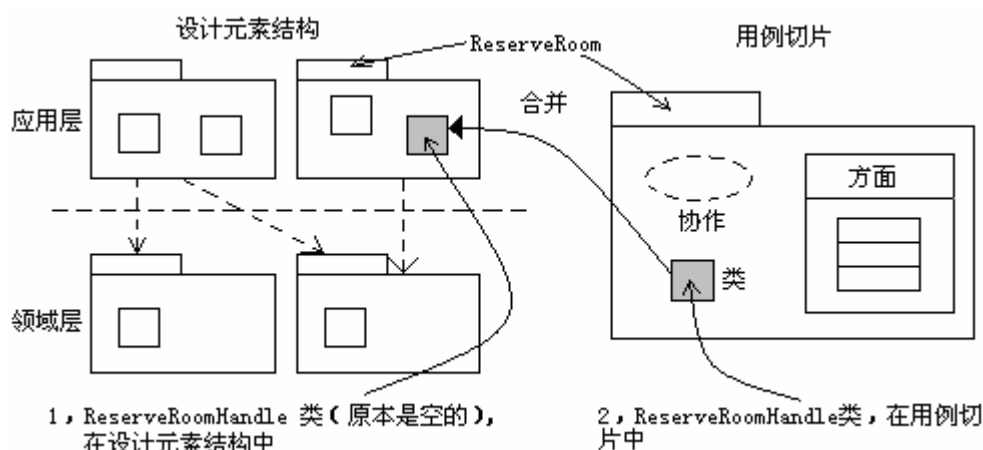
“预定房间”用例切片包含 ReserveRoomHandle 类和 Room 类扩展，在类中标示出相应的行为，不论是类还是方面，一般都有多个。这就需要有一个机制把 Room 类的片断合并到原有的 Room 类中去，这可以通过 AOP 的方面机制来完成，例如，通过 intertype 声明和 advice 来实现。

用例切片和幻灯片很近似，而且组合也是简单的把它们叠在一起。

对软件设计而言，设计模型的命名空间可以确保用例切片正确组合，下面我们将讨论怎么把用例切片合并到设计模型里面去。

1. 合并特定于某用例的类

以 ReserveRoomHandler 类为例，看一看特定于某个用例实现的类是如何合并到设计模型中的。假定我们使用层（layer）和包来组织设计模型，则可能有一个应用层，它包含一个客户应用包，ReserveRoomHandler 类应该添加在这个包内。



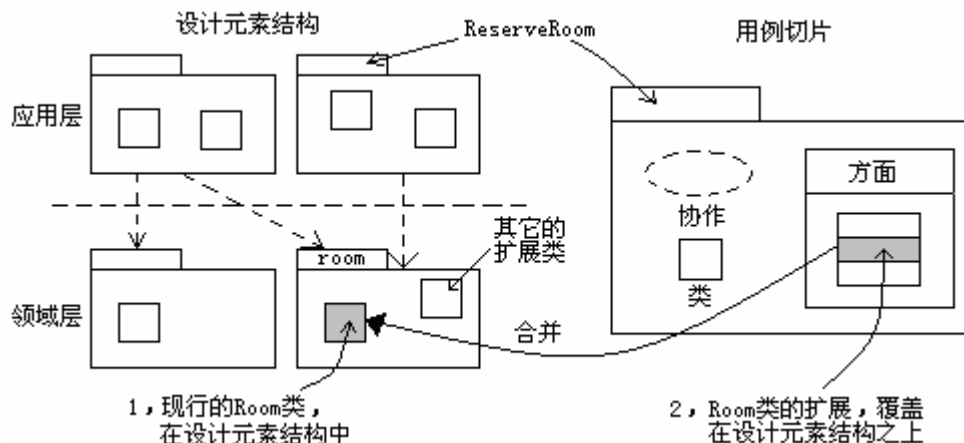
在设计元素结构中，唯一的标识出 ReserveRoomHandle 类（图中的 1），此时这个类还是空的，只有名字，我们可以给设计模型设置“命名空间”或者包名来赋予全名称，接下来可以利用合成机制，把用例切片（图中的 2）中的 ReserveRoomHandle 合并到设计元素结构中。

在 AspectJ 中，可以很容易的在类的外部定义方法，然后通过 Intertype 声明把新的特性（属性、方法及关系）合成到已有的类中。另一方面，后面我们要讨论的设计模式，也有一

些实现这种思想的模式，这都是值得我们去研究的。

2, 合并用例特定的类扩展

下面要考虑如何在设计元素结构中现有类上合并类的扩展。在 AOP 之前，传统编程语言并没有提供这个功能，现在 AOP 可以通过 `intertype` 声明，向原有的类添加特性，要添加在原有类的特性，收集在一个类的扩展中，在下图中用“合并”箭头说明。



在我们的例子中，**Room** 类就是在设计元素结构中的原有类，它原本已经叠加在了设计元素结构中，现在我们要叠加上新额外的类扩展。假定，**Room** 类位于领域层的 **room** 包中，并表示出了 **Room** 类，我们希望把“预定房间”用例切片的 **Room** 类扩展叠加到实际元素结构中的 **Room** 类中。

在 AOP 中，可以使用带 `intertype` 声明的 `aspect` 实现，这个 `aspect` 应用在用例“预定房间”的实现中，所以我们把它命名为 `ResverRoom`，下面表达了在现行的 `Room` 类中是如何添加这个操作的。

//AspectJ 中的方面"预定房间": ReserveRoom.java

// ReserveRoom 方面是位于应用层的包 app.customer 中(与 ReserveRoomHandler 等类在一个包)

```
package app.customer;
import domain.room.Room;
public aspect ReserveRoom{
    //它扩展的是 Room 类,、
    //并且在设计结构元素的 Room 类中添加了一个操作 updateAvailability()。
    public void Room.updateAvailability(){
        //具体代码,
    }
}
```

在这个例子中，**aspect**（方面）包含了一个 **intertype** 声明，它是只针对一个类的，但是，通常要把多个操作合并到多个类中去，这就需要对这些 **intertype** 操作进行组织，也就是把相同类添加的特性归集到一起，这种做法很有效。

另外，对每个独立的类所拥有的全部 **aspect**（方面）进行可视化和理解十分重要，对每个原有类的 **advice**（通知）和 **intertype** 声明进行准确的建模极其有利，同时在标识出协作中的角色的时候，这些类的扩展也会自然的派生出来。

5.5 使用 Pointcut 使扩展保持分离

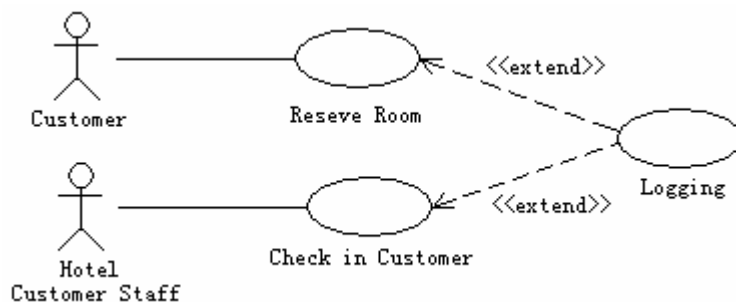
在关注点分离问题上，除了考虑对等用例以外，还需要考虑扩展用例的问题。

扩展用例是用例的一种，它可以使额外的功能与现有的用例保持分离。扩展用例的实现需要在原有的操作中定义额外的行为，这在传统的技術中往往会引起混乱。通过面向方面的技术，用例切片可以使操作扩展与现有的操作保持分离，并且在编译或者执行期间由 Pointcut 在所执行的点上执行。Pointcut 是可以参数化的，它可以在一个或者多个点中执行相同的扩展，这种参数化可以与模版化结合起来，以解决更大范围内的问题。下面我们来讨论有关问题。

一、实现扩展用例

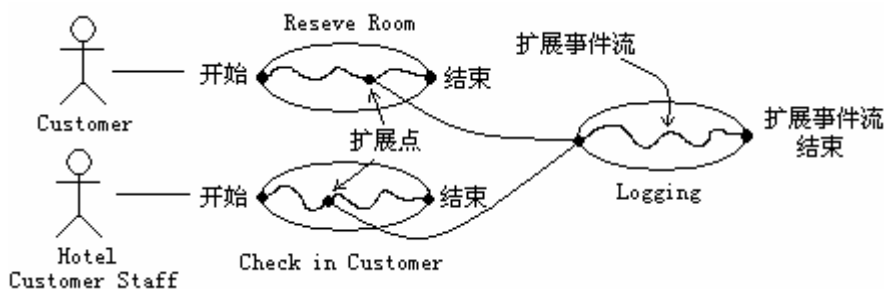
在用例分析的时候，我们已经知道扩展（extend）的目的是在用例指定的扩展点处，添加额外的行为（称之为扩展事件流），通过扩展 pointcut 来指定扩展点，我们可以使扩展在需求阶段保持分离。

下面的简单例子是一个“记录日志”的用例，这是一个基础结构机制，它对用例“预定房间”和“登记入住”进行扩展，如下图所示。



扩展“记录日志”的目的是，统计不同类型房间的请求次数，以及请求是否成功，这些信息可以为后面分析酒店不同类型的房间受欢迎的程度提供基础。

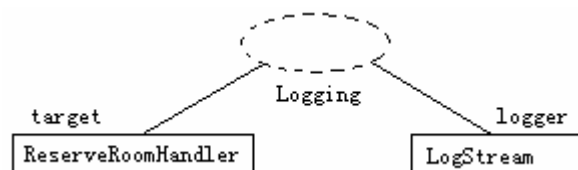
用例的事件流如下图所示。



二、使扩展用例的实现保持模块化与扩展

1. 用例实现保持模块化

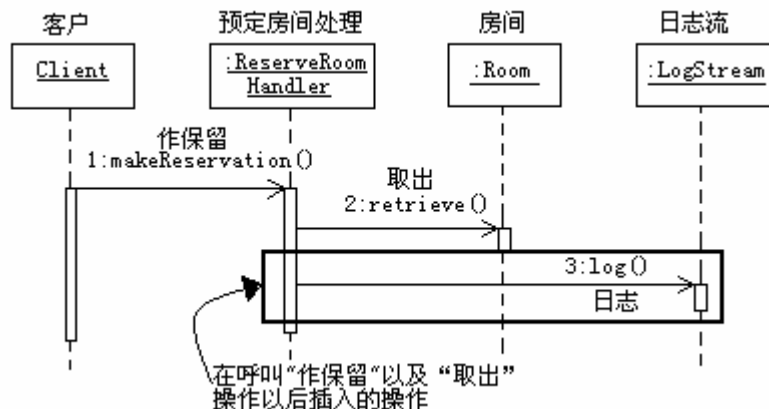
在扩展用例 Logging 的实现中，需要具备两个角色，一个是记录者（logger），另一个角色是目标（target），两个角色的工作解释如下。



记录者 (logger): 扮演这个角色的类是 `LogStream`，负责把申请房间的次数保持到持久化的数据库中去，它中间包含一个保留数据的 `log()` 操作。

目标 (target): 在 `LogStream` 类把信息保留之前，首先需要获得请求信息，这需要通过 `ReserveRoomHandler` 类来处理，它属于目标角色。只需要在其中添加新的扩展，就可以获得请求信息。

现在要做的就是识别出具体的执行点，把这个执行点上的操作合并到目标上，下图表达了用例实现“预定房间”的交互信息。



当某个客户调用 `ReserveRoomHandler` 对象中的 `makeReservation()` 操作以后，将调用 `retrieve()` 操作。这也表明了扩展将在 `makeReservation()` 操作之中进行。

在这个交互的基础上，我们叠加了“记录日志”操作扩展，上图中用一个方框标记了这一部分，这个方框也说明了额外的行为会在什么位置执行。它是在完成 `Room.retrieve()` 调用之后，在 `makeReservation()` 操作中发生的。实际上，不管 `Room` 对象中的“取出数据” (`retrieve()`) 操作是否完成，操作扩展都将调用 `LogStream` 对象中的 `log()` 操作，完成请求信息的存储工作。

下面我们来描述在面向方面的设计技术中，如何来标记这个扩展过程。

2. 操作扩展

一个操作扩展是由位于某个操作内、特定于某个用例实现的行为组成的。在这个例子中，操作扩展只用于调用 `LogStream` 对象中的 `Log()` 操作，操作扩展对应于 AOP 中的 `advice`。

在标识执行操作扩展的位置时，必须注意两件事，也就是结构上下文和行为上下文。

结构上下文: 这个上下文是对调用方的描述，它描述的是操作扩展是叠加到元素结构中的什么位置，也就是在哪个包、哪个类以及哪个操作上执行。在这个例子中，操作扩展是在 `ReserveRoomHandler` 类的 `makeReservation()` 操作中执行。

行为上下文: 这个上下文是对扩展方的描述，它在操作扩展中标示出一个执行点，表明操作扩展是在什么位置对原有操作进行扩展。这个执行点与用例建模中的扩展点，以及 AOP 中的连接点 (`join points`) 的概念是相对应的。在这个例子中，只关注了 `ReserveRoomHandler` 类的 `Log()` 调用，也可能你还会关注还有哪些类调用了这个操作。通常，执行上下文会牵涉到调用堆栈的概念。

操作扩展声明: 正是这样一些要求，操作扩展的声明应该表明操作扩展会对哪个现有的操作进行扩展，所以操作扩展的描述应该包括以下三个部分：

结构上下文+行为上下文+操作扩展

所以在上面的图中，操作扩展“记录日志”的声明就应该是：

```
makeReservation() { after call (Room.retrieve()) logData }
```

说明：

第一部分 `makeReservation()`：说明操作扩展在哪个现有操作中执行，我们不需要把它表示成 `ReserveRoomHandler`. `makeReservation()`，这是因为操作扩展本身就是放在一个 **aspect** 里的类扩展中，所以只要标明操作扩展属于该类中的哪个操作就行了。

第二部分 **call** (`Room.retrieve()`) `Room retrieve`：表示扩展点是位于 `Room` 类中的 `retrieve()` 操作。这个表达式就是 AOP 中的 **pointcut**。关键字 **after** 是一个修饰符，相对于 **pointcut**，用来定义操作扩展是在什么地方执行。这个操作扩展的语义是：其执行点在 `makeReservation()` 操作中，具体的说就是 `makeReservation()` 调用 `Room` 实例中的 `retrieve()` 操作之后。

第三部分是 **logData**：说明操作扩展实际上是完成什么功能。

下图描述了完整的扩展用例切片“记录日志”，其中包括 **Logging** 协作，`LogStream` 类，以及名字为 **Logging** 的 **aspect**。注意，`LogStream` 类是位于 **aspect** **Logging** 之外的，这是因为，`LogStream` 类是为了增加日志功能而新加的。



3, Pointcut

我们再仔细看一下表达式 `call(Room.retrieve())`，在调用 `Room` 类中的 `retrieve()` 操作的时候，它直接引用了现有的 `makeReservation()` 操作中的扩展点，但这种直接引用是易变的，例如，如果 `retrieve()` 操作改名为 `read()`，那么所引用的扩展点将失效。因此，这样对扩展点的直接引用并不好。

更好的方法是，为扩展点起一个有意义的名字，说明当执行操作扩展的时候，是位于哪个原有的操作，然后再单独定义一个实际扩展点。在这个例子中，主要是跟踪对于 `Room` 类的调用，因此命名为 `roomCall` 更有意思。而你可以在扩展点的定义中使用这个名字。扩展点的命名实际上就是定义 **pointcut**。因此，扩展表达式声明就成为。

```
makeReservation() { after call (<roomCall>) logData }
```

符号 `{}` 说明，在操作扩展声明中的 `roomCall` 是一个参数，它被绑定为不同的值。这可以用 **aspect** 中的 **pointcut** 表达式来实现。另一方面，这个参数会应用在整体用例切片中，在这种情况下用例切片就成为一个模版。

下图展示了修订后的“记录日志”用例切片，它采 **pointcut** 来引用位于 `call(Room.retrieve())` 的连接点（也就是扩展名），名字为 `roomCall` 的 **pointcut** 的定义如下所示：

```
roomCall= call(Room.retrieve())
```

包含 **pointcut** 的用例切片的“日志记录”如下图所示。



上图中的 Logging 的 **aspect** 实现如下：

//AspectJ 中的方面"记录日志"：Logging.java

//Logging 方面是位于基础结构层的包 logging 中

```
package infra.logging.;
```

//这个方面也引用了 ReserveRoomHandle 类，因此需要把它 import 进来。

```
import app.customer.ReserveRoomHandle;
```

// roomCall 包括到一个 Room 类的引用，因此需要把它 import 进来。

```
import domain.room.Room;
```

```
public aspect Logging{
```

```
    //定义了一个名字为 roomCall 的 pointcut。
```

```
    pointcut roomCall();
```

```
        //下面是 roomCall 的定义，是用&&连接符定义的更小的 pointcut。
```

```
        // makeReservation 中可能的连接点。
```

```
        withincode(void ResveRoomHandler.makeReservation())
```

```
            && call(void Room.retrieve()); // Room 中 retrieve()操作可能的调用。
```

```
        //把操作扩展实现一个 advice，
```

```
        //关键字 after 说明了操作扩展的位置在名字为 roomCall 的 pointcut 之后。
```

```
        after():roomCall(){
```

```
            //具体代码，
```

```
        }
```

```
}
```

用例切片和方面技术为模型驱动架构提供了有效的解决方案，而面向方面的软件开发的理论和实践，现在仍然在发展之中。在项目的早期建立一个弹性架构至关重要，其目标是让系统健壮并且减少需求变化所带来的系统大量修改。它同时也让系统容易理解，从面向方面的观点来看，一个弹性系统让你更容易定义 pointcut，因为需要扩展的所有类和职责都被局部化了。

建立描述系统的模型结构是迭代的，首先从一些与平台无关的结构开始，然后逐个分析架构重要的用例，这样做的时候，会逐渐补充和完善已经存在的结构，并且把平台相关的元素融合到结构中，当遍历了所有的架构重要用例以后，就可以建立一个相当有弹性的架构了。即使不使用面向方面的语言，我们也可以从面向方面的讨论中获取重要的营养，使我们的系统构建的更加健壮而且有弹性。

第六章 基于质量属性的架构设计

软件架构设计最关注的问题也架构师的思维重心，应该放在质量需求上，因为所谓架构设计，事实上就是对质量需求的应对，例如，随着需求变更越来越频繁越来越不可避免，在质量需求中就提出可维护性的要求，对这种要求的强烈程度，往往就决定了某一种架构设计的特征，而这个特征表现在对所付出代价的权衡。为了对软件质量进行度量，必须对影响软件质量的要素进行度量，并建立实用的软件质量体系或模型，在需求分析中，必须对软件质量需求提出可度量的模型，这是一个好的架构设计最重要的基础。

注意，下面的讨论会有一些实例，它们都来自于国外公开发表的资料，主要是为了表达问题方便，并不牵涉到任何具体的设计。

6.1 质量度量模型与质量属性场景

一、三层次软件质量度量模型

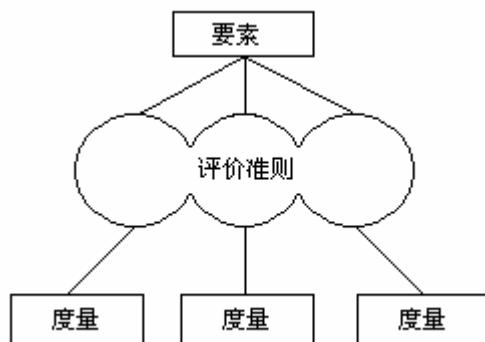
软件的质量由一系列质量要素组成，每一个质量要素又由一些衡量标准组成，每个衡量标准又由一些量度标准加以定量刻画。质量度量贯穿于软件工程的全过程以及软件交付之后，在软件交付之前的度量（内部属性）主要包括程序复杂性、模块的有效性和总的程序规模。在软件交付之后的度量（外部属性）则主要包括残存的缺陷数和系统的可维护性方面。

ISO 9126 称为“软件产品评价：质量特性及使用指南”。在这个标准中，把软件质量定义为“与软件产品满足声明的或隐含的需求能力有关的特性和特性的总和”，可以分为 6 个特性，即：功能性、可靠性、效率、可使用性、可维护性以及可移植性。这 6 大特性，每个特性包括一系列副特性，其中各质量特性和质量子特性的含义如下：

	特性	副特性
1	功能性(functionality) ：与一组功能及其指定的性质的存在有关的一组属性。功能是指满足规定或隐含需求的那些功能。	适合性(suitability) ：与对规定任务能否提供一组功能以及这组功能是否适合有相关的软件属性。 准确性(accurateness) ：与能够得到正确或相符的结果或效果有关的软件属性。 互用性(interoperability) ：与同其他指定系统进行交互操作的能力相关的软件属性。 依从性(compliance) ：使软件服从有关的标准、约定、法规及类似规定的软件属性。 安全性(security) ：与避免对程序及数据的非授权故意或意外访问的能力有关的软件属性。
2	可靠性(reliability) ：与在规定的时间内和在规定的条件下，软件维持其性能的有关能力。	成熟性(maturity) ：与由软件故障引起实效的频度有关的软件属性。 容错性(fault tolerance) ：与在软件错误或违反指定接口的情况下，维持指定的性能水平的能力有关的软件属性。 易恢复性(recoverability) ：与在故障发生后，重新建立其性能水平并恢复受影响数据的能力，以及为达到此目的所需的时间和努力有关的软件属性。
3	易使用性(usability) ：与为使用所需的努力和由一组规定或隐含的用户对这样所作的个别评价有关的一组属性。	易理解性(understandability) ：与用户为理解逻辑概念及其应用所付出的劳动有关的软件属性。 易学性(learnability) ：与用户为学习其应用(例如操作控制、输入、输出)所付出的努力相关的软件属性。 易操作性(operability) ：与用户为进行操作和操作控制所付出的努力有关的软件属性。
4	效率(efficiency) ：在规定条	资源特性(resource behavior) ：与软件执行其功能时，所使用

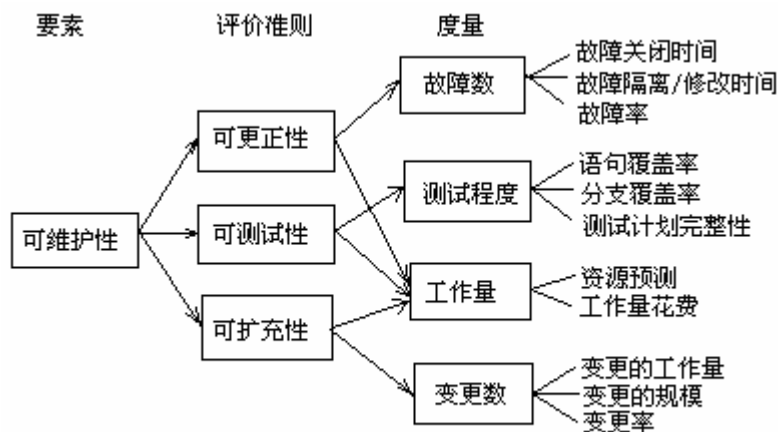
	件下,软件的性能水平与所用资源量之间的关系有软件属性	的资源量以及使用资源的持续时间有关的软件属性。
5	可维护性(maintainability) : 与进行规定的修改所需要的努力有关的一组属性。	易分析性(analyzability) : 与为诊断缺陷、失效原因或判定待修改的部分所需努力有关的软件属性。 易改变性(changeability) : 与进行修改、排错或适应环境变换所需努力有关的软件属性。 稳定性(stability) : 与修改造成未预料效果的风险有关的软件属性。 易测试性(testability) : 为确认经修改软件所需努力有关的软件属性。
6	可移植性(portability) : 与软件可从某一环境转移到另一环境的能力有关的一属性。	适应性(adaptability) : 与一软件无需采用有别于为该软件准备的软件属性。 易安装性(installability) : 与在指定环境下安装软件所需努力有关的软件属性。 一致性(conformance) : 使软件服从与可移植性有关的标准或约定的软件属性。 易替换性(replaceability) : 与一软件在该软件环境中用来替代指定的其他软件的

其软件质量模型包括 3 层,即高层:软件质量需求评价准则(SQRC);中层:软件质量设计评价准则(SQDC);低层:软件质量度量评价准则(SQMC),也就是:质量要素(factor)、评价准则(criteria)、度量(metric)。



ISO 认为这 6 个特性是全面的,也就是说软件质量的任何一部分都可以用这 6 个特性中的一个,或者多个特性的某些方面来描述。这 6 个特性中的每一个都定义为“一组与软件相关方面有关的属性”,并且能细化为多级子特性。

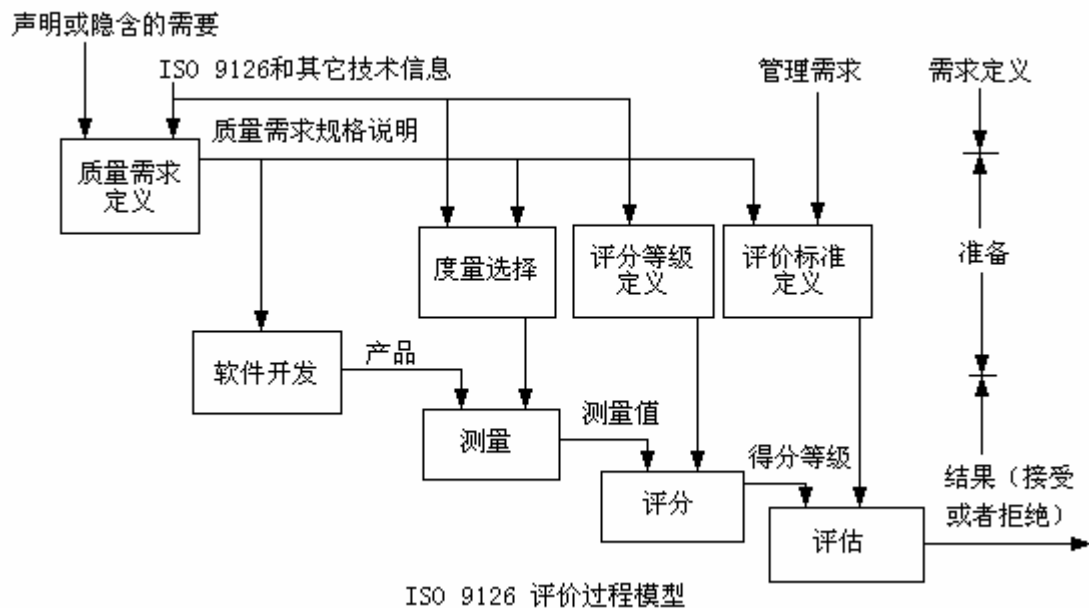
例如:



可维护的分解图

ISO 9126 标准也定义了评价软件质量的过程,这个过程如下图所示,事实上这个过程

是有一些批判的言论的，但参考 ISO 9126 模型和框架来建立符合自己软件特点的质量评估体系，仍然会对提高软件的质量水平发挥重要的作用。



把软件的质量度量引申到架构设计，就衍生出了架构的质量属性和其实现的问题。我们可以通过下面几个方面来描述，来构造自己的质量度量模型。

软件架构的质量属性		
编号	名称	内容
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。 哪些现有构件是可以重用的。 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。 建立可重用的构件，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。 构件的接口是一致的和容易理解的吗。 有解释构件接口的过程吗。
9	可测试性	有可以测试语言类、构件和服务的工具、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。

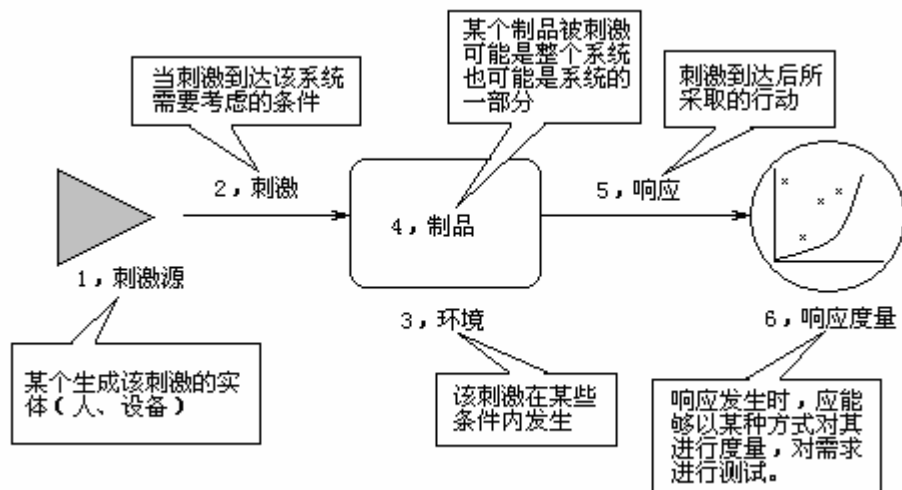
上述问题，给我们提供了一个线索，在回答这些问题的时候，就可以利用它建立具体软件的架构质量标准。同时也给我们初始设计的改进方向，提供了一个一目了然的依据。这些内容，可以成为我们制定评估表格的基础。还要注意，在高层架构设计中，初步的验证性实验或者说原型项目是完全必要的，这种概念和方法的验证可以极大的规避风险，因为如果初期的架构思想出现了本质性的缺陷，对整个设计来说可能就是灾难性的。

当我们建立了架构的质量属性以后，就需要对每一个质量属性进行进一步的分解和研究，从而找到针对具体质量属性的解决方案，以此综合出架构策略，这种对具体质量属性的细化描述，我们称之为质量属性的场景。

二、软件架构质量属性的场景

软件的质量属性决定了架构风格，因此在上述整体质量属性研究的基础之上，我们必须把问题集中一下，在架构设计的过程中，对于某一个具体的质量属性，我们应该如何来描述它呢？描述质量属性的方法是很多的，我们可以根据三层次质量度量模型，构造出更切合实际的架构设计质量属性场景来。

针对具体质量属性的需求，我们定义的质量属性场景由以下六个部分组成。



注意：于不同的系统，对于质量的关注点是不一样的，例如某个设计强调了高性能，另一个设计可能更强调高可靠性，针对这些质量属性就可以形成基本设计重点，我们称之为“解决方案”，解决方案是一种影响质量属性的设计决策，把各种解决方案打包集合，我们称之为“架构策略”。下面我们简要介绍一下几个最重要的质量属性的解决方案。

下面我们通过几个设计中最为关注的可靠性、可维护性以及可集成性问题，来深入的讨论这个问题。

6.2 可靠性质量解决方案

如果一个项目把可靠性放在了十分重要的位置，那么开发这样的系统就必须花力气研究可靠性解决方案。关于可靠性是这样表述的，当系统不再提供与其规范一致的服务的时候，错误就发生了。错误或其组合，可能会导致故障的发生。恰当的解决方案可以阻止错误发展为故障，至少能够把错误的影响限制在一定的范围之内。

在软件开发中实施可靠性是很昂贵的，因此对可靠性问题必须谨慎，我们必须制定可靠性计划，借此能够解决具体风险，并降低成本，可靠性问题必须兼顾投资回报的影响，并却反馈到软件项目计划中去，可靠性计划大概需要研究如下内容并作出决策。

可靠性计划		
序号	可靠性问题	解决方案
1	产品可靠性需求	
2	错误预测途径	定义功能描述 定义错误 错误失效分类计划 客户可靠性需求 研究折衷方案 确定产品可靠性目标
3	错误预防途径	在设计构件中分配可靠性 满足可靠性目标的工程过程 基于功能描述的资源计划 错误引入和传播管理计划 软件可靠性度量计划
4	错误排除途径	定义操作描述 可靠性增长测试计划 测试过程跟踪计划 附加测试计划

		可靠性目标证明过程
5	容错途径	监督现场可靠性目标 跟踪客户对可靠性的满意度 通过监督可靠性来安排新特性的引入 通过可靠性措施引导产品和过程改进

一、可靠性质量属性场景

针对可靠性所关注的问题，我们需要仔细研究：如何检测系统发生的故障，系统故障发生的频度，出现故障的时候会有什么情况，允许系统有多长时间非正常运行，什么时候可以安全的出现故障，如何防止故障的发生，以及发生故障的时候需要哪种通知等。

在这个问题上，我们需要把故障和错误区分开来，一般来说，系统用户可能会观察到故障，但不能观察到错误（异常），当用户可以看到错误的时候，实际上错误已经发展到了故障。错误是可以实现自动修复的，如果在用户没有观察到的情况下实现了错误修复，则就没有发生故障。系统的可靠性一般定义成系统正常运行的比率：

$$\alpha = \text{平均正常工作时间} / (\text{平均正常工作时间} + \text{平均修复时间})$$

注意，在计算可靠性的时候，一般不计算预定的停机时间，即使用户在这个时间可能得不到服务，但由于停机是预定的，所以可以不予考虑，可靠性的一般场景如下，我们可以根据具体情况来设计具体的场景。

可靠性的一般场景		
序号	场景	可能的值
1	源	系统内部，系统外部
2	刺激	错误： 疏忽：构件未能对某个输入作响应。 崩溃：构件不断遭受疏忽的错误。 时间：构件做出了响应，但响应时间太早或者太迟。 响应：构件用一个不正确的值作响应。
3	制品	指定系统可靠性的资源：处理器，通信通道，进程，永久存储。
4	环境	当出现错误或者故障的时候，系统状态的期望值。比如：如果看到了一系列的错误，可能希望关掉系统。如果看到了第一个错误，可能希望只是对功能的降级。
5	响应	系统出现故障的时候，应该具备的反应，比如： 记录故障，通知适当的各方，禁止导致故障的事件源，在一段时间内不可用，在降级模式下运行等等。
6	响应度量	系统必须可用的时间间隔，可用时间，系统在降级模式下运行的时间间隔，修复时间等。

为了解决已定义的可靠性问题，需要有可靠性解决方案，例如，当发生错误的时候，可靠性解决方案的目标就是屏蔽错误或者进行修改。

维持可靠性的方法都包括某种类型的冗余，以及用来检测故障的某种类型的健康监视，以及当检测到故障的时候某种类型的恢复。有些情况下，监测和恢复是自动进行的，有些情况下是手动进行的。这种模块或者设备的冗余加上健康监测，往往成为高可靠性系统的特征。换句话说高可靠性的产品都是昂贵的。

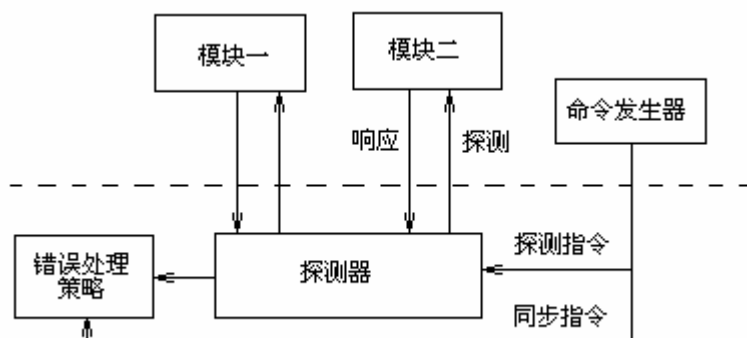
问题在于是不是使用了这些技术，设备的可靠性就一定提高了呢？事实证明并非如此，我们必须利用历史测量的数据进行数据处理，要对模块的易损性以及发生的位置等要素进行统计检验，从而找到最需要进行可靠性架构设计的位置，力争在比较少的成本下获取比较大的可靠性指标，这往往是架构决策的基础。

二、健康监测

事实上无论代码写的多么优秀，各种问题考虑得多么全面，但系统发生故障的可能性还是存在的，作为模块或者设备的冗余配置，恰当的健康监测是判断模块是否工作正常的基础架构。

1. 命令/响应 (ping/echo):

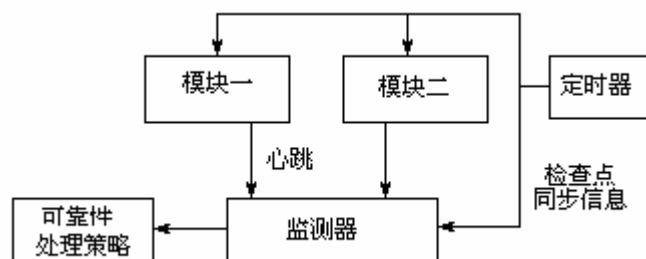
一个构件发出一个命令，并希望预定时间内收到一个审查构件的响应。和心跳方式相比，它的特点是发出检测命令是由专门的构件完成的。这个解决方案一般用在处理共同完成某项任务的一组构件内。一般情况下，“探测器”可以放在底层，它向相关软件进程发出命令，而高层命令发声器向底层探测器发出测量命令。在模块设计的时候，必须包括对于健康探测器探测命令的响应信息，这种响应应该力求对于各个模块是统一的，同时也要尽可能的简单。命令/响应机制一般以方法访问的方式工作，这在某些情况下可能是方便的。



健康监测可以远程进行，但更多的情况是在本地进行，和远程探测相比这种方式所需要的网络带宽比较少。

2. 心跳 (dead man) 计时器:

“心跳”是一种主动检测方案，由被测构件主动发出一个自检结果信息（心跳），由另一个构件收听这个信息，如果心跳失败，则可以判断这个构件失灵，并通知按照可靠性策略纠正错误构件。心跳也可以同时传递数据，比如传递上一次交易的日志。心跳一般以事件机制工作，统一的定时器并不一定是必须的，但如果所有的被测模块按统一时间工作，给可靠性处理策略将会带来好处。



上述两种情况，被测模块都可以处于不同的进程中。

3. 异常:

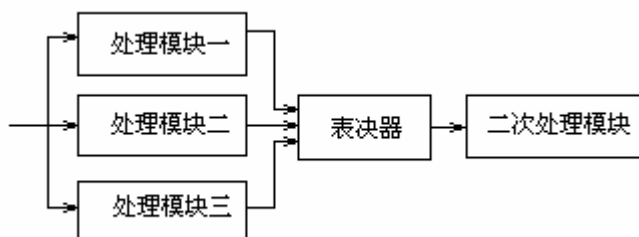
异常处理是一种最常用的程序内处理方案，当出现异常的时候，异常处理程序将会在一个进程中处理相关问题。

三、错误恢复

高可靠性的系统往往伴随着系统冗余，必然带来了造价提高和系统复杂性提高，但在航空设备、航空管制或者战场指挥这样要求高的但数据相对比较简单系统，使用恰当的系统冗余是合适的，但对于大量的长时间数据处理这样的问题，就需要作一些限制。

1, 表决

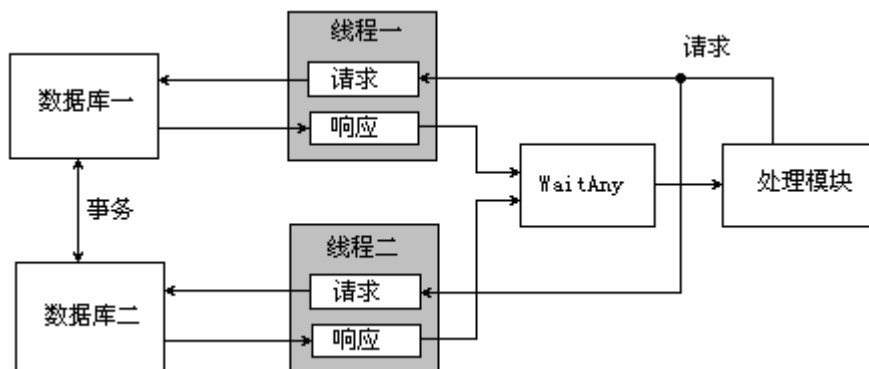
对于复杂的数据处理计算，例如快速傅里叶变换或者卷积、相关函数的计算、航路计算、及威胁计算以及应对策略计算等等，如果处理上出现了问题（错误），往往会带来灾难性的后果，在这种情况下，可以使用表决方案。它的特点是运行在冗余处理器上的每个过程都具有相同的输入，表决器采取某种策略取出数据，常用的是多数法，或者首选法。



极端情况是冗余构件是由不同小组开发，并且是在不同的平台上运行，当然这样开发和维护是很昂贵的，仅用在要求非常高的环境中，比如作战指挥和分析系统、飞行器的控制等。

2, 主动冗余（热重启）

所有的构件都以并行方式对事件做出响应，但只有第一个构件的响应被使用，而其他的响应被丢弃。比如在数据库系统中，这种情况使切换时间只有几毫秒，但资源消耗却成倍增加。



3, 被动冗余（暖重启/双冗余/三冗余）

由一个主要构件对事件做出响应，并通知其它备用构件进行状态更新，一旦错误发生，将自动切换到备用构件上，在此之前，系统必须保证备用状态是新的。切换的时机可以由备用构件决定，也可以由其它构件决定。该解决方案依赖于可靠的接管，有时候周期性切换可以提高可靠性。

例如，作为一个典型的例子，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？

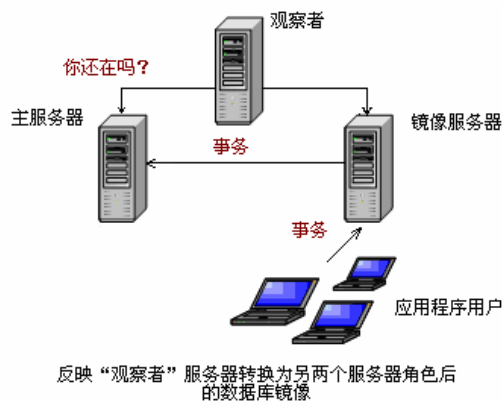
首先，在停止和重启服务器的情况下，我们可以清除连接池并重新建立它。但如何才能保证结果和服务器关闭之前完全相同呢？这就要用到容错恢复技术了。

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。



注意，当发现主服务器有问题的时候，用户方需要自动清除连接池，并转而使用备用服务器。

4. 备件

备件是在计算机系统中配置了用于更换不同故障的构件。在出现故障的时候，必须转入适当的软件配置，然后重启计算机。这个方法必须记录持久设备的状态变化，以使接入的设备能以适当的状态工作。有一些重新引入模块的修复解决方案，包括 shadow 操作、状态再同步以及回滚。

5. shadow 操作

以前出现故障的构件，可以短时间内以“shadow 模式”运行，以确保在恢复该构件前，模仿工作构件的行为。

6. 状态再同步

不论是主动还是被动的冗余解决方案，都要求所恢复的构件在重新提供服务前更新其状态更新的方法取决于可承受的停机时间、更新规模以及更新所要求的消息数量。如果可能的话，最好用一条消息包含它的状态。

7. 检查点/回滚

检查点就是记录已创建的状态，一般可以定期进行，也可以对某种消息进行响应。如果故障是以不同寻常的方式发生的，可以用上一个检查点拍了快照以后所发生的事务日志来恢复系统。

四、错误预防

1. 从服务中删除

这个解决方案是从操作中删除系统的一个构件，以支持某些活动来防止预期发生的故障。比如在周期性切换的被动冗余方案，可以重新启动构件，以防止内存泄漏导致系统故障发生。

如果从服务中删除是自动的，就需要以架构策略来支持它。如果是手动的，就需要对系统进行设计以支持这个动作。

2, 事务

事务可以保证多步数据处理的一致性。

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。

3, 进程监视器

一旦检测到进程中存在错误，进程监视器就可以删除非执行进程，并为该进程创建一个新的实例，再把它初始化一个适当的状态。

6.3 基于高可靠性的架构设计

假定某一个大型系统的设计提出了极高的可靠性要求，因此在架构设计的时候，就需要针对可靠性问题讨论具体的解决方案。

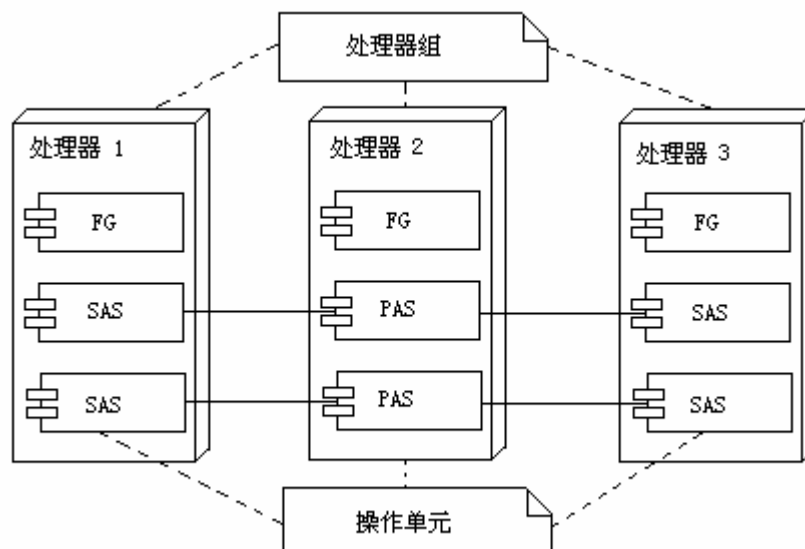
一、进程间提升可靠性的方法

大型系统一般是按照多处理器环境设计的，逻辑上组成处理器组，处理器组的目的是运行一个或者多个应用程序的副本，这一思想对于支持容错性和可靠性是非常重要的。在多个运行副本中，一个为主，称为主地址空间（PAS），其它的为辅，称为备用地址空间（SAS）。

一个主地址空间，和相应的备用地址空间的集合称为操作单元，某个操作单元完全驻留在同一处理器组的处理器中，一个处理器组最多可以包含 4 个处理器。

没有用这种容错方式实现的系统称为功能组，功能组可以根据需要出现在各个处理器上。

应用程序根据可靠性的需要，可以是操作单元，也可以是功能组（FG）。



操作单元的接管过程如下：

操作单元的 PAS 代表整个操作单元接收消息并做出响应，然后 PAS 对自己的状态和相应的 SAS 的状态进行更新（可能会向 SAS 发送多条消息）。

如果 PAS 出现了故障将按如下步骤完成接替工作：

- 1，把一个 SAS 提升为新的 PAS。
- 2，为 PAS 重新设置该操作单元以及客户机的关系（实际上是操作单中的一个固定的列表），在这个过程中，PAS 要向各个客户机发送消息，内容为：原来提供服务的操作单元发生故障，正在等待 PAS 发送消息吗？如果是，新的 PAS 就处理收到的任何服务请求。
- 3，启动一个新的 SAS，代替原有的 PAS。
- 4，新启动的 SAS 把自己的存在告知于新的 PAS，新的 PAS 就会向这个 SAS 发送消息，使它保持最新的状态。

如果在 SAS 内部发生错误，就需要在另外的处理器上启动新的 SAS，新的 SAS 要与 PAS 协调工作，并接受状态信息。

仔细研究这样的需求和动作，就可以设计出合适的架构来。

如果需要添加一个新的操作单元，需要采用如下步骤进行设计：

- 1，确定必要的输入数据及所在的位置。
- 2，确定哪些操作单元需要用到这个新的操作单元的输出数据。
- 3，以一种非循环方式把这个操作单元的通信模式加入到整个系统中，以避免死锁。
- 4，设计消息，实现所期望的数据流。
- 5，确定内部状态数据（包括从 PAS 到 SAS 的状态数据）。
- 6，把状态数据划分为能够很好的适应网络要求的消息。
- 7，定义必须用到的消息类型。
- 8，规划 PAS 失败的时候的切换，要对更新数据作合理的规划，保证能够完全的反映出各种状态。
- 9，保证切换发生的时候数据的一致性。
- 10，保证需要完成的处理步骤，能在一次系统“心跳”的时间内完成。
- 11，规划和其它操作单元的数据共享和数据锁定协议。

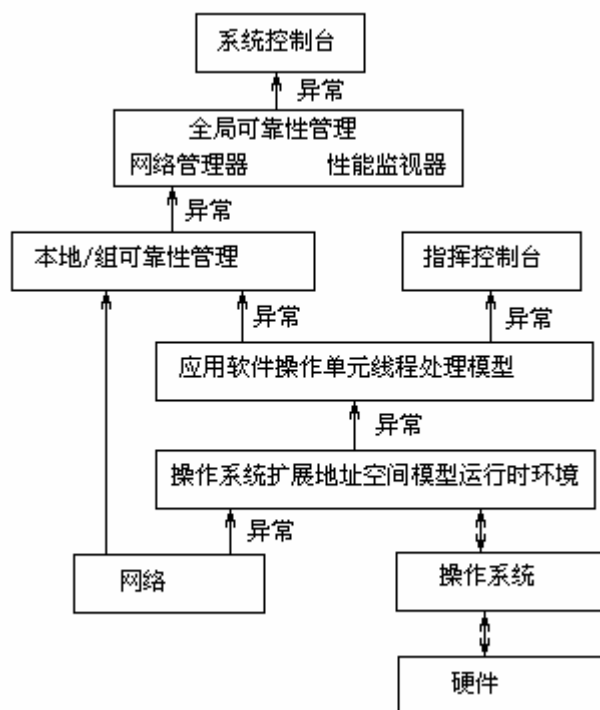
有经验的项目组成员可以按照这个步骤开展工作，为了加速开发，可以构造一个“代码模板”，以加快开发速度和减少错误。

无论是客户端还是服务方，都可以通过这个方法提升模块的可靠性。

二、保证可靠性的分层结构

对于可靠性极高的系统，由于不能系统出现故障的时候冷启动，而是需要尽可能快地切换到备用构件上，这就出现了新的架构层次结构，称之为容错层次结构，该结构描述了如何检错、如何隔离、如何恢复等，它主要用于捕获应用程序交互的错误并从中恢复。

容错层次结构包括“本地可靠性管理器”和驻留在系统管理控制台上的“全局可靠性管理器”，它们之间要实现通信，以及报告当前状态和接受控制命令，它通过内部时间同步器把本处理器的时钟和其它处理器的时钟同步。



该系统的容错层次结构提供了多种不同层次的错误检测和恢复机制，在每个层次上都能异步的进行如下工作：

- 1，检测自身、同级实体和底层实体的错误。
- 2，处理来自底层的例外情况。
- 3，诊断、恢复、报告或者提交例外情况。

高可靠性系统要比普通系统复杂的多也昂贵的多，但是在例如飞机交通管制系统、指挥控制系统等这些需要高可靠性的场合，这样的设计是值得的，实际效果也是非常好的。即使普通的系统，在恰当的节点上恰当的应用高可靠性解决方案，有的时候也是必要的。

6.4 可维护性解决方案

可维护性直接影响到可扩展、可修改性能。由于需求变更在设计中的不可避免性，加之迭代开发使用需求变更为驱动力，更重要的是系统升级是必然的，所以，可维护性是我们架构布局需要十分注意的问题。

一、可维护性质量属性场景

可维护性关注的是有关变更的成本问题，它提出两个关注点：

1，可以修改什么（制品）？

可以修改系统的任何方面，包括系统的功能、平台、环境、质量属性以及其容量。应该注意到，泛泛的按照可维护性策略设计，往往系统的开发成本可能会超出所能接受的范围，因为一个可维护性很好的系统，其开发成本就会比较高，系统性能也可能比较低。所以，必须在需求阶段很好的定义变更预期，没有这个定义，设计就会非常盲目的。

2，何时进行变更和由谁进行变更（环境）？

过去常见的就是修改源代码，但这样的做法一般是不合理的。现在提出的问题不仅仅是何时变更的问题，也要提出由谁进行变更的问题，不同的变更方案，可以由开发人员、最终

用户或者系统管理员来进行，这就需要有相应的设计策略。

可维护性的一般场景如下。

可维护性的一般场景		
序号	场景	可能的值
1	源	开发人员、最终用户、系统管理员
2	刺激	希望增加/删除/修改/改变功能、质量属性、容量
3	制品	系统用户界面、平台、环境或者与目标系统交互的系统
4	环境	运行时、编译时、构建时、设计时
5	响应	查找架构中需要修改的位置，进行修改且不会影响其它功能，对所做的修改进行测试，部署所做的修改。
6	响应度量	根据所影响的元素度量：成本、努力、资金。 该修改对于其它功能或者质量所造成影响的程度。

可维护性解决方案，目的是发生变更的时候，直接影响的模块数量最少，因此这个解决方案也可以称作“局部化修改”解决方案。另一个设计目标，就是在局部化修改的时候，不要引起“连锁反应”。第三个设计目标，是控制部署时间的成本，我们称之为“延迟绑定时间”。下面我们对这三个解决方案进行讨论。

二、局部化修改

一般来说，如果把修改限制在一小组模块之内，最后就可能会降低成本特别是维护成本。这就需要在设计期为模块分配好责任，把预期的变更限制在一定的范围之内，可以采用的方案如下。

1，维持语义的一致性

语义的一致性是指模块中的责任能够协调一致的工作，不需要过多的依赖其它模块。也就是设计模块的时候，必须确保内聚度指标。

虽然耦合性和内聚度指标可以在一定程度上度量语义一致性，但设计的时候还需要仔细考虑预期的变更，也就是根据一组预期的变更来度量语义一致性。其中的一个解决方案就是“抽象通用服务”。通过一个通用服务模块可以支持可重用性，但“抽象通用服务”可支持可维护性，而且这种修改不会影响其它用户。所以，考虑问题的时候不仅仅要研究语义一致性，也要研究防止连锁反应。

2，预期期望的变更

考虑所预期的变更的集合，可以给我们提供特定的责任分配方法，我们需要回答下面的问题：

“对于每次变更，系统的分解结构是不是限定了完成变更所需要修改的模块集合？”

与此相关的问题是：

“根本不同的变更，会影响相同的模块吗？”

从某种意义上说，这个方案是维持语义一致性对于变更的进一步深入。尽管变更是很难预期的，但是如果在系统分析的时候能够深入研究这个问题，则对于设计质量的提高是非常有意义的。

3，泛化该模块

这里的泛化与 UML 中的泛化并不完全是一层意思，这里所说的是，模块设计的越通用，发生变更对模块影响就越小。

4，限制可能的选择

当修改的范围很大的时候，可能会影响很多模块，限制各个模块可能的选择，会降低这

些修改所造成的影响，这就需要在设计的时候对模块功能和它的可变范围进行限制。

三、防止连锁反应

连锁反应指的是对一个模块的修改，需要更改修改并没有直接影响到的模块。在设计的时候必须仔细研究这个问题，这是模块分割的一个重要依据，我们来思考下面几个情况。设两个模块 A 和 B，我们从如下几个方面讨论模块间的依赖性：

(1) 语法或语义

假定要使 B 正确编译（或执行），必须使用 A 产生的数据类型（或语义），而且必须保证两者数据类型（或语义）一致，这就发生了语法或者语义的依赖性。

(2) 顺序

数据顺序：要使 B 正确执行，必须按一个固定顺序接受由 A 产生的数据。

控制顺序：要使 B 正确执行，A 必须在一定的时间限制内执行（比如，在 B 执行前，A 执行的时间不能超过 5 毫秒）。

这就发生了顺序上的依赖性。

(3) A 的一个接口身份

A 可以有多个接口，要使 B 正确编译和执行，这个接口的身份、名称或者句柄必须与 B 假定的一致，这里接口的身份、名称或者句柄被称之为方法（或者函数）的签名。这就发生了接口身份上的依赖性。

(4) A 在运行时的位置

要使 B 正确执行，A 运行时的位置必须与 B 假定的一致。这就发生了运行位置的一致性。

(5) A 提供的服务/数据的质量

要使 B 正确执行，涉及 A 所提供的数据或者服务的质量，必须与 B 的假定一致，比如，某个传感器提供的数据必须有一定的准确性，以使 B 的算法能够正常运行，这就发生了服务或者数据质量上的依赖性。

(6) A 必须存在

要使 B 正确执行，A 必须存在，例如，如果 B 请求对象 A 提供的服务，如果 A 不存在，则 B 就不能正常执行，这就发生了存在上的一致性。

(7) A 的资源行为

要使 B 正确执行，A 的资源行为必须与 B 假定的一致。比如 A 必须使用与 B 相同的内存，这就发生了资源行为的依赖性。

借助于对依赖性的理解，在设计得时候我们必须仔细考虑这些风险，把重要的位置中的风险列出来，逐个考虑解决方案，特别是用于防止某些类型的连锁反应的解决方案。这样一来，设计的质量就会比较高。

为了解决连锁反应的问题，我们可以采用如下一些策略。

1, 信息隐蔽

信息隐藏就是把某个子系统的责任分解成更小的部分，并选择哪些信息是公有的，哪些是私有的。可以通过指定的接口获得公有责任。信息隐藏的目的时把变更隔离在一个模块内，防止变更扩散到其它的模块内，这是防止变更扩散最早的技术，由于它使用变更预期作为分解的基础，到今天仍然是最重要的技术。

2, 维持现有接口

如果 B 依赖于 A 的一个接口的名字和签名，则应该维持这个接口不变（条件是 B 对 A 不应该有语义依赖性）。在高层设计的时候，必须仔细设计接口，考虑全面以及充分利用预期变更的信息，确保接口的稳定性。

实现这个解决方案的模式包括。

- **添加接口：**大多数编程语言都允许多个接口，当添加新的服务或者数据的时候，从而使现有的接口保持不变并提供相同的签名。
- **添加适配器：**给 A 添加一个适配器，该适配器把 A 包装起来，并提供 A 原来的签名。
- **提供一个占位程序 A：**如果程序修改要求删除 A，但 B 依赖于 A 的签名，那么，可以设计一个占位程序，虚拟的提供 A 的接口和行为，当然这也是为了可修改性付出的代价。

3，限制通信路径

限制与模块共享数据的模块，也就是说，减少使用由给定模块所产生的数据的模块数量，就会减少连锁反应。在实时性要求很高的场合，可以把重要的数据在本地建立映像，只在发生更改的时候才改变这个本地数据集的数据，这样就可以限制通信的数量。

四、推迟绑定时间

后期绑定和允许非开发人员进行修改，可以使系统的可维护性大大提升，推迟绑定时间还能够使最终用户或者系统管理员进行设置，或者提供影响行为的输入。

后期绑定需要系统的设计上做好准备，解决方案包括：

- **运行时注册：**即支持即插即用操作。
- **配置文件：**目的是启动时设置参数。
- **多态：**允许方法调用的后期绑定。
- **构件更换：**允许载入时绑定。
- **遵守已定义的协议：**允许独立进程的运行时绑定。

6.5 基于高可维护性的架构设计

如果一个系统强调可维护性，设计中就会有若干特点。首先各个分系统应该具备极高的独立性，分系统与主干系统之间的数据信息与控制信息应该仔细设计，确保信息比较少而且稳定。主干系统的任务是调度，并不真正处于每个分系统的控制环路之中，而主要是作为各个分系统的一个集中处理和应对策略的分析控制中心，这样的布局有利于保证各个分系统的独立性和稳定性，也可以保证在发生损坏的时候，各个分系统仍然能够独立工作，这也是一种典型的集散系统方案。如果分系统的升级频度很高，这样的设计也就强调了可维护性。

一、问题的陈述

主干系统的输入主要用来获取和管理各个独立分系统的数据。

而该主干系统的输出包括各种显示设备，数据的集中处理结果，以及向各个分系统发出相应的指令。系统要求具备计算和处理能力。假想的大粒度数据流图如下：

二、高层结构

在模块设计的时候，主要考虑的问题是信息隐藏。很多分系统在主干系统的生命周期内可能会用新的型号或新原理组成的系统更换，这就需要把与这个系统交互的细节封装到某个模块中去，这个模块的接口只不过是一个抽象分系统。每个模块都提供了一组仅通过某个特定的访问过程与其它模块交互，如果换成了新式的分系统，这需要改变这个模块中的某些细节，对软件其它部分没有影响。

在模块划分的时候遵循了如下原则：

- 每个模块的结构尽可能简单。
- 使用不不需要知道其它模块的细节。
- 对设计修改的容易程度与发生修改的频度有合理的对应关系。
- 把软件系统作的比较大的修改，可分解成各个模块的一组独立的修改，程序员不需要相互交流，新老版本的组合、测试不应该有困难。

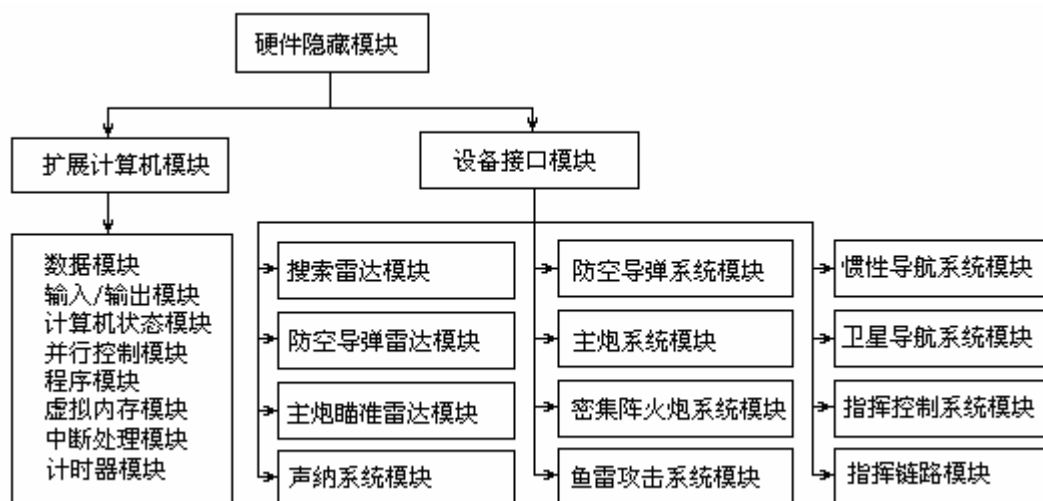
对于可维护性质量指标的实现，采用如下解决方案。

目标	如何实现
易于改变：分系统、平台、符号、输入	信息隐藏。
了解未来可能的改动	正式的评审过程，借助相关领域专家的经验。
为各开发小组分配任务，使各小组之间交流很少。	按层次结构组织各模块，每个开发小组负责开发一个二级模块及其所有子模块。

由于易扩展性摆在了重要的位置，所以模块划分以隐藏模块为基础，以此得到对于这个系统来说很特殊的模块划分。

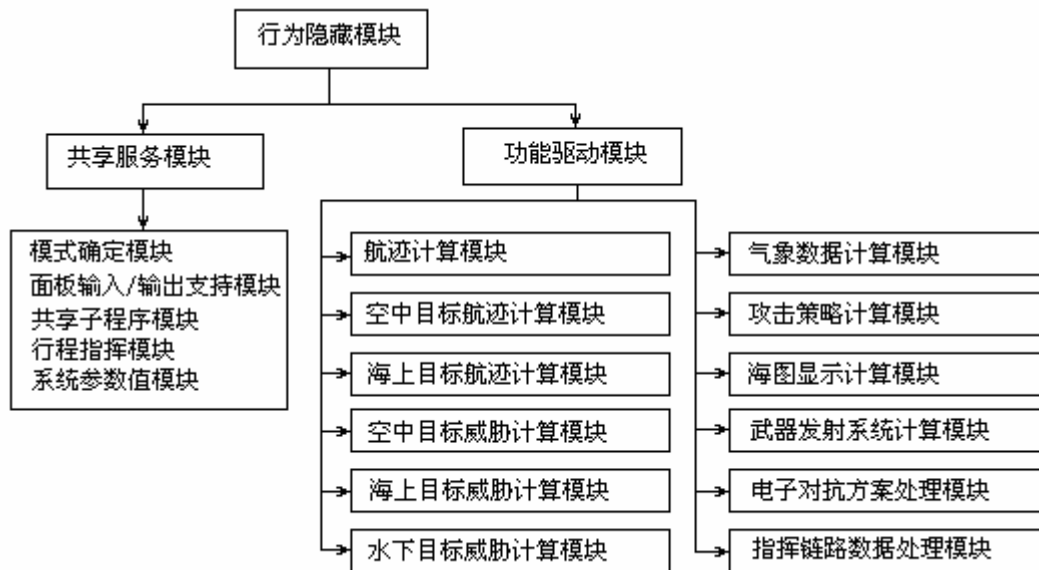
1. 硬件隐藏模块

这个模块的作用，是保证当硬件的任何一部分被替换的时候，需要修改的仅仅是这个过程，也就是说这个模块实现的虚拟分系统。



2. 行为隐藏模块

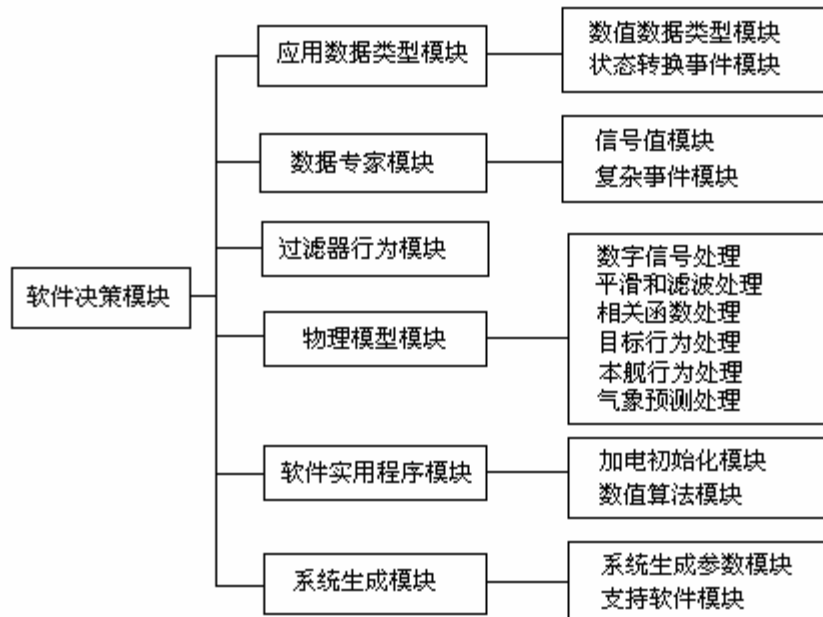
行为隐藏模块包括当影响系统行为的需求发生变化的时候，可以修改的过程，这个模块主要实现系统功能性需求的隐藏，这个过程也包括了应该发送给硬件隐藏模块提供的虚拟输出设备的数据。



3. 软件决策模块

软件决策模块把基于数学定律、物理事实、编程思想（什么样的算法速度更快？）的软件设计决策隐藏起来，该模块隐藏的信息，在需求文档中是没有记录的，模块对外的接口是由设计人员决定的，所以这种模块明显区别于其它模块。

- **应用数据类型模块：**主要是提供适合各个系统的数据类型，比如距离、时间长度、角度等，这主要是实现数据类型的转换。它隐藏的是数据转换的方法。
- **数据专家模块：**大多数数据都由专门的模块产生，并保留在本地，它从数据生产过程获得数据值，集中提供相应系统工作状态的所有数据。数据专家是一个“中间人”，它控制数据更新的频度和来源，并且只在数据发生变化的才更新。
- **过滤器行为模块：**提供过滤器的数字化模型，用以对噪声等进行过滤。
- **物理模型模块：**不直接测量，但对其它的测量数据进行综合计算得到的估计，这是对这些模型计算的隐藏。
- **软件实用程序模块：**包括多个程序员编写的实用函数、资源监视程序等。
- **系统生成模块：**隐藏的是直到系统生成的时候才做的决策，这些决策包括系统生成参数的确定，以及对某一模块不同实现方式的选择。



这样的设计，在分系统不断更新的情况下，能够很容易的升级和维护。这种分解对于设计文档的编写、联机配置文件的编制、测试计划的制定、编程小组的组织、评审过程的安排、项目进展的确定等都产生了重要的影响。

6.6 基于高可集成性的架构设计

对可维护性的进一步强调，衍生出可集成性的质量需求，很多系统具有很强的分布性，而且还必须时常升级和更新，这就对开发和维护这样的软件系统，提出了巨大的挑战。例如，我们希望所设计的系统能够毫无困难的在各种不同的要求、不同的组合甚至不同的背景下都能够很好的实现集成，而这种集成需要付出的代价有很小，这就把可集成性提高到系统设计的主要的位置上来。

在软件系统质量属性中，可集成性的描述并不是总是被强调的，不过，在由分散小组或单独的组织开发的大型系统，可集成性往往成为驱动因素。事实上一些可集成性的目的也是可维护性，它的解决方案可以有如下一些：

- 使接口较小、简单、稳定；
- 遵守已定义的协议；
- 松散耦合使元素间的依赖较小；
- 使用构件框架；
- 使用已有版本的接口等。

下面我们通过一个假想的系统，研究一下在这些原则的指引下，得出的架构模式能否具有较高的可集成性，而且能满足软件必须具备的其它质量属性。飞行模拟系统是当前最复杂的系统之一，它具有很强的分布性，严格的时间要求，而且还必须时常升级和更新，这就对开发和维护这样的软件系统，提出了巨大的挑战。

一、问题的陈述

飞行训练模拟系统有三种作用：第一，训练飞行人员和机组人员。第二，对环境的模拟，包括空气、各种威胁、武器和其它飞机的影响等，第三，对教练的模拟，根据训练目的，提

前下发文字材料，教练可以实时设置环境，比如设备故障、暴风导致的湍流等。

事实上飞行环境模型可以达到任意的逼真程度，比如对于气压高度表而言的空气压力影响，可以考虑上升气流和下降气流，当地天气模式，甚至附近飞机的存在也可能产生湍流。这样而言，飞行模拟器就需要强大的计算能力，但这样逼真的模式是不是一定能提高飞行员技能，一直还是存在争议的，因此，性能这个质量属性是影响架构特征的主要因素。

飞行模拟器运行涉及多种状态，其中包括：

- **运行状态：**这是系统的正常操作状态。
- **配置状态：**对于当前的训练科目发生变化时的状态。
- **停止状态：**指停止当前的模拟。
- **重放状态：**指没有机组人员干预的情况下，重放某次模拟。这种重放可以用于演示，也可以用于研究自己的操作是否恰当。

飞行模拟系统具有以下 4 大特征：

1，实时性要求高

为保证逼真，飞行模拟系统的执行必须保证非常高的帧频，比如“转弯”这个动作，所有的仪表、景色以及座舱控制的动作都应该与实际情况一致，而且保证很好的协调。

2，连续的开发和修改

不论是军用还是民用飞机，都处于不断的修改、更新的过程之中，因此，飞行模拟系统也是与此同步不断的修改的。更重要的是，这种修改对于软件设计来说，完全是被动的、不可预知的。

3，规模大、复杂程度高

随着飞机的性能越来越好，飞行模拟器的规模成指数增长的趋势。

4，在分散的地理位置上开发

军用飞机模拟系统一般以分布式的方式开发，造成这种情况至少有两个原因。一个是技术方面的，也就是不同的部件需要相当不同的专业的知识。另一个是政治上的，这种大规模高科技的产品，本来就是各方面争夺的对象。

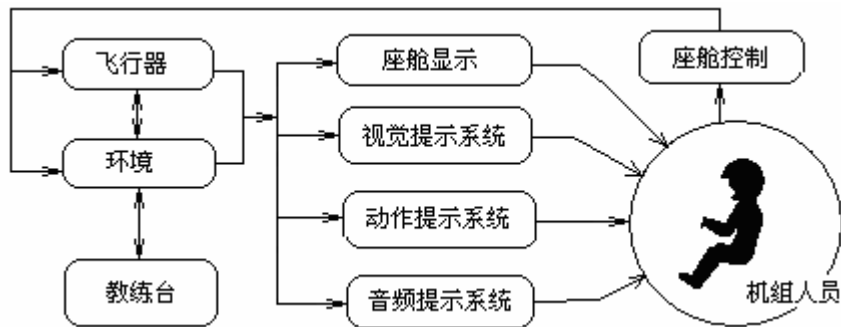
由于通信链路的加长，本来就因为规模大而不易实现的可集成性，实现的难度更大了。

二、架构解决方案

飞行模拟系统的上述特征，迫使架构设计必须考虑以下两个问题：

- **强调架构的可集成和可修改性：**事实上系统测试、集成和修改的代价超过了开发成本，所以架构设计应该非常强调可集成和可修改性。
- **软件结构不需要和飞机结构严格对应：**把运行时效率作为首要目标，因此软件结构不需要和飞机结构严格对应，而是和动作相对应。比如作“转向”动作，实际的动作是踩方向舵，摆动副翼，同时控制升降舵防止掉高。而在飞行模拟系统中，为了保证实时性，可以构造一个“转向模块”，同样，其它的动作也有相应的模块，换句话说，模块的建立是来自于对机组人员的操作进行考查，把任务的组件模型化。采用这种方案可以极大的减少执行计算所需要的通信，但物理模块可能交叉出现在不同的模块中，造成集成的困难。

下图给出了飞行模拟器的参考模型。



对应于需求和质量的特征，可以采用以下三个构架策略：

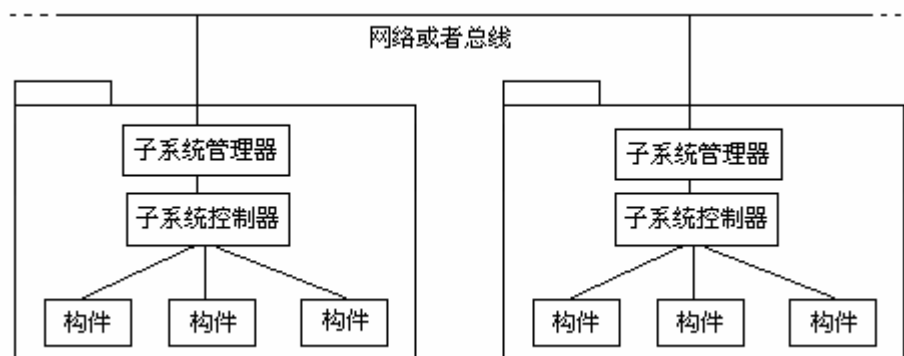
1，性能策略

这是一个关键质量目标，这主要通过时间调度策略来完成。管理程序所调用的每个子系统，都有运行时间的限制，而且还确定了硬件规模，以保证容许子系统的这种时间要求。

实时性能要求控制循环分配给子系统的时间，要少于模拟系统的一个操作周期，架构设计中，要保证每一个激励响应的流程合理（时间最短），单元测试中，响应时间要作为技术参数存在。为保证实时性，代码或者模块的冗余是允许的。

2，可集成性策略

由于强调可集成性要求，这里采用了所谓结构化模型，它有意识的把子系统之间数据连接和控制连接应该控制在最小。



首先从构件级别来说，子系统同级构件不能直接传递数据和信息，任何数据和控制信息的传递，都必须通过子控制器来完成。子系统控制器的数据在内部是一致的，要把另外的构件集成到子系统，比新构件直接与其它构件通信要简单得多。也就是说，集成问题的规模不再与构件数量成指数关系，而降低为线性关系。

其次从子系统级别来说，在集成两个子系统的时候，各子系统的构件都不能直接交互，所以问题又简化了，只要保证两个子系统之间的数据传递一致性就可以了。添加新的子系统可能会影响到其它子系统，但子系统的数量要比构件少，所以问题的复杂性不会太大。

所以，在结构化模型中，通过有意识的限制可能连接的数量，可集成问题得到了简化。这种限制的代价，是子系统控制器经常成为各个构件公用的纯粹的数据通道，而且也提高了复杂性和性能的开销。不过在实践中，这种方法的收益远远超过了成本，这些收益包括创建了一个能够进行增量式开发和更轻松的集成的骨架系统。

3，可维护性策略

对于结构化模型来说,设计人员和维护人员只需要理解少数几个基本构件配置就可以了。并且由于功能的局部化,使得某次修改只涉及少数几个子系统控制器或者构件,就使可维护性得到提高。下面针对可集成性,来考虑几个具体问题。

三、结构化模型的架构模式

针对可集成性的具有挑战性的问题，可以引入一种称之为“结构化模型”的架构模式，它突出强调以下几个方面：

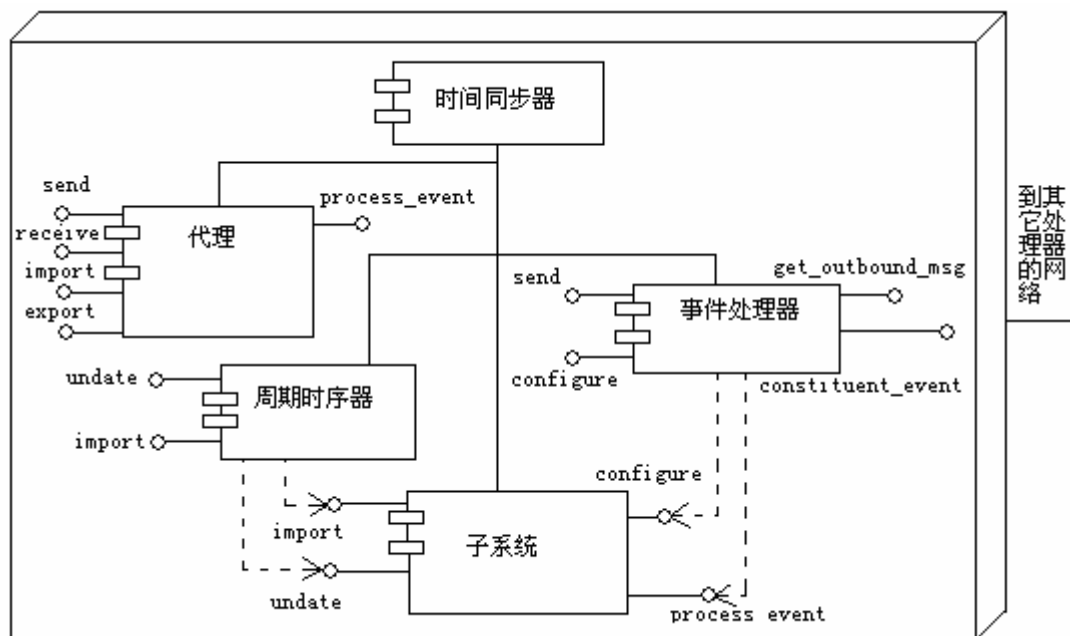
- 系统子结构的简单性和相似性。
- 把数据和控制信息的传递策略与运算分离开。
- 模块类型数量最少。
- 较少的系统级协调策略。
- 设计的透明性。

结构化模型的架构模式，包括一组元素和对元素运行时的协作配置。大型系统结构化模型可能需要多个处理器运行，所以各个部分需要相互协调和通信，在粗粒度上，可以把结构化模型架构样式分成两大部分：

- **管理部分:**用以处理协调问题,包括子系统的实时调度、处理器之间的同步、从指挥控制台上对事件进行管理、数据共享、数据完整性等等。
- **应用部分:**处理各个武器系统的运算,对系统建模等。它由各个子系统和构件完成。

四、子系统管理部分的模块

下图给出了子系统管理部分的结构化模型,各构件的功能简述如下。



1, 时间同步器

时间同步器是系统调度机制的基础，它负责维持模拟系统的内部时钟。它接受来自其他三个部分的数据和控制信息，也负责和其它部分保持时间上的同步。

2, 周期时序器

用于完成子系统所要做的所有周期性的处理，也包括按照固定的周期调用某些子系统。

它向时间同步器提供两种操作：

import：请求周期时序器调用子系统的 **import** 操作。

update：请求周期时序器调用子系统的 **update** 操作。

它具备组织“调度”信息和通过某个分发机制实现周期对某些子系统调用。

3，事件处理器

事件处理器用以协调子系统所做的所有非周期处理。它涉及 4 种操作：

Configure：启动新的任务。

Condtituuent_event：某个实例针对某个模块的特定实例时使用。

Get_outbound_msg：时间同步器要在系统操作状态下执行非周期处理的时候使用。

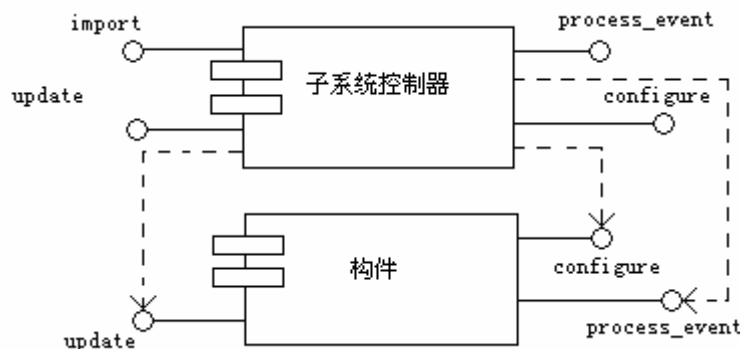
Send：子系统控制器使用它向其它子系统控制器发送事件。

4，代理

完成各个模型之间的系统级通讯。

五、子系统应用模块

子系统应用部分只有两大部分，即子系统控制器和构件，如下图所示。



特点：

各个子系统控制器之间可以互相传递数据，但只能向其子构件传递数据。

构件只能与它的父构件传递数据（或者控制信息），但不能向其它任何构件传递数据（或者反馈信息）。这个规则是防止把数据或者控制信息传递给同级的构件。

这些限制的基本思想，就是通过消除构件实例之间除父子关系之外的所有耦合情况，来简化集成和维护工作，维护或者集成的影响，由上一级子系统来调解，这就是“限制通信”解决方案的一个例子。

1，子系统控制器

子系统控制器用来把相关子模块的一组功能联系起来，完成以下功能：

- 实现对子系统整体的模拟。
- 调解系统和子系统之间的控制和非周期通信。

因为结构化模型限制了控制器构件之间的通信，所以子系统控制器必须在他的构件和其它的构件之间建立起逻辑上的联系。

为了建立相应的状态，并且在时间上与系统协调，它还具备两个功能：

- 为了解决数据一致性和时间连贯性的问题，检索入站连接的数据值，并把它保存在本地。
- 稳定构件的模拟算法，并报告这个子系统当前是不是稳定的。

此外，子系统控制器还需要支持训练任务参数的重新配置，它分别通过周期时序器和事件处理器周期的和非周期的实现这个功能。

周期操作：

update: 周期性的接受所需要的数据源，并且向构件播送改变信息。

import: 周期性的读入入站数据，并把它保存在本地，以备 **update** 使用。

非周期操作：

process_event: 要求子系统控制器对某个具体的事件做出反应。

configure: 非周期的处理系统操作状态（如初始化），该操作用以建立一组命名的条件，如某些设备的配置或任务等。

2，控制器构件

子系统控制器构件对外部系统来说，可能是对自身系统的模拟。构件之间的所有的逻辑交互都由子系统控制器来完成。

六、系统设计中需要关注的问题

在系统设计进行模块切分的时候，需要关注以下几个问题。

1，系统的骨架化

对于一个庞大的系统，如果设计规格不加以控制，则会给将来的集成和维护带来极大的困难。但在这个例子中，仅仅使用了 6 个模块类型（构件、子系统控制器、时间同步器、周期时序器、事件处理器以及代理），就可以对这么大的系统进行完整的描述。这就使得架构很容易创建、理解、集成、发展和修改。

更重要的是，如果采用一组标准模式，我们就可以创建一个骨架系统，为此创建出规格表、代码模版和描述这些模式的示例程序。这样一来，就允许一致性分析。

架构师还可以坚持设计和开发人员仅仅使用所提供的构建块，这虽然听起来有些苛刻，但这样一来，就可以把设计人员从系统总的功能实现的关注中解脱出来，构件的标准化必然带来可集成性的提高。

2，功能分配给构件的原则

把功能分配给构件的时候，需要考虑如下原则：

- 实际物理系统的各个部分应该与软件系统很好的对应，这为我们提供了真实世界的概念模型。通过对各个分系统交互的理解，也可以帮助我们更好的理解软件各部分交互的方式。这对于用户和评审也很有帮助。
- 要理解未来分系统更新换代的规律，比如整体换装设备需要做哪些变化？这种理解可以帮助我们设计模块的范围，以使将来系统升级时的更改局部化。
- 努力降低系统接口的数量和规模，这来自于各部分更强的功能内聚，把最大的接口放在各部分之内而不是各部分之间。

3，功能分解中的接口设计

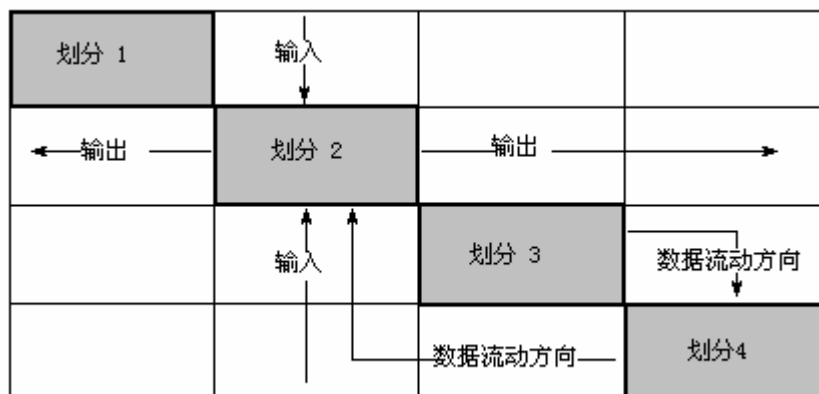
在功能分解的时候，必须对系统划分和接口设计进行仔细研究，确保接口有效而且最少。一个有效的方法是使用 n^2 图， n^2 图可以有效地对模块和接口进行评估，它是捕获模块的数如何输出的一个好办法。下面是一个 n^2 的例子，它的特点是：

主对角线方向是系统的划分；

各划分的输出显示在行中；

各划分的输入显示在列中；

数据从一个划分到另一个划分是按照右→下→左→上的路径传递的。



下图是一个假想的飞机仿真系统 n^2 图，为简化表示，图中没有像正规图形那样表示出接口的名称和类型。在设计阶段仔细设计并稳定接口是非常必要的。系统设计师可以一直这样做下去，直到表示出接口中所有的原始对象为止。

初步的设计可能是混乱的没有条理的，但有了这样的矩阵就可以使自己的思维逐步的条理化，最后达到非常优秀的设计。

动力学子系统	负载	飞行器状态向量	飞行器位置
能源	飞机机身子系统	能源	
惯性状态	负载	航空电子设备子系统	所有权播送
大气、地形及空气数据		环境状态数据	环境子系统

这里讨论的假想案例，旨在说明当系统对性能、可靠性与可修改性提出比较苛刻的要求的时候，我们如何能合理设计架构，使项目能够在节约成本的情况下实现这些质量属性。成本的节约可能表现在现场安装小组只有以前所要求的一半，因为他们可以更容易的查找和纠正问题。

设计方案通过以下方式实现了这些质量属性：

限制结构化模型架构模式中的模块类型配备的数量，限制模块类型之间的通信，根据飞机预期变更的信息分解功能。从度量的角度，主要表现在现场测试描述（即测试问题）的大幅度减少，开发人员还发现，采用这种方法更容易纠正问题。

6.7 软件产品线的应用

产品线系统(Product Line System)的定义为：一组软件密集型系统，它们共享一个公共的、可管理的特性集，以规定的方式使用公共核心资产集开发，来满足某个特定市场或者任务的具体需要。

产品线系统是一种产品开发的组织方式。产品线集中体现了软件复用思想。经验表明，单靠技术方法并不能保证成功的产品线生产能力，经济、组织、管理和过程在建立和维护产品线中起到了关键作用。一个产品线是共享一组共同设计及标准的产品族，从市场角度看是

在某市场片断中的一组相似的产品。建立产品线是根据生产的经济学，使产品族可复用构件能达到最大限度的复用目的。产品线方法可以通过各种可复用软件构件，如需求分析、产品需求规格说明、架构设计、代码构件、文档、测试策略和计划、测试案例和数据、开发人员的知识和技能、过程、方法及工具等，支持最大限度的软件复用。产品线也是基于在相同产品价格条件下提高竞争力的商业考虑。

换句话说，软件产品线的本质，是在生产软件产品的家族的时候，以一种规范的、策略性的方式重用资产。所以，它的特点就是一组可重用的资产，它包括一个基本架构以及一些的可剪裁、可填充的元素。此外，它还包括设计文档、用户手册、项目管理制品（包括预算和进度）以及测试计划和测试用例。当成功建立产品线了以后，可以把可重用的资产保存在“核心资产库”中去，由于可以使用到多个系统，可以明显的降低成本和缩短开发时间。

基于产品线的开发代表了软件工程的一个创新的、不断发展的概念，对于不同客户的需求，如何使产品线的部件具有更大的灵活性，以简化这种针对不同的客户的创建，是研究产品线问题值得注意的问题。

产品线中可重的要素很多，包括：

1，**需求**：大多数需求与早期开发的产品线系统需求是基本一致的，可以减少需求工程的工作量。

2，**架构设计**：如果产品线的架构是集中了最聪明能干的工程师投入的大量时间，所确定的架构就已经排除了系统的质量缺陷（性能、可靠性、可修改性等），因此新产品开发可以在相同的架构上完成，而且可以确保质量。

3，**元素**：软件元素包括接口及其文档，测试计划和规程等，如果这些元素是适用的，新系统的搭建就只是代码的具体实现和重用已有代码，要注意，最重要的可重用的元素集就是用户接口，它代表了很多关键的设计决策。

4，**建模和分析**：性能分析、可集成性分析、分布式系统问题（如证明没有死锁）、容错方案以及网络负载策略都可以在产品中重用。在构建新系统的时候，可以认为这些问题都已经解决了。

5，**样本原型系统**：一个产品线系统应该有一个高质量的演示原型以及关于性能、安全性、保密性和可靠性的高质量工程设计原型。

软件产品线依赖于重用，但为什么到今天为止软件重用总是得不到承诺的那么多好处呢？关键是元素集放什么和放多少的问题并没有真正得到解决。软件产品线事实上是确定一个策略，用以对架构进行定义，确定功能，了解质量属性，仔细考虑只有构建的时候需要重用的元素才放到重用库中去。过少则没有意义，过多则难以应用，产品线将依赖战略规划来发挥作用。产品线架构设计的基本步骤如下。

一、确定范围

所谓确定范围，使定义哪些系统属于这个产品线，哪些不属于这个产品线。这个范围也代表了组织对于可预测的将来，我们将会开发什么样的产品的最佳预测。如果范围过小，则达不到好的投资回报，如果范围过大，则利用核心资产库开发单个产品的工作量就太大了。

确定范围并不在于发现共性，而在于发现可以充分利用的共性，这才可以极大的降低系统构建的成本。而且，在确定范围的时候，也不是仅仅考虑相似的产品，在不同的产品线上可能存在共同的功能块，这也是需要关注的问题。

在核心资产库中，软件架构是重中之重，而一个可以在几乎所有产品线中不同产品可以通用的架构，设计的关键是架构设计中有一组明确允许可以发生变化的，所以，识别允许的变化是架构设计责任的一部分。架构设计必须研究清楚，什么是必须保持不变的，以及允

许发生什么样的变化和怎样应对这种变化，而这种变化往往是新系统带有重要特点的一部分。当然，架构本身是属于不变的那一部分。

二、确定变化点

由于产品可能会以很多方式发生变化，在产品线设计之初，我们必须确定在产品线的需求分析中获取变化点。其中包括特性、平台、用户接口、质量属性以及目标市场等。其次，在产品线的架构设计中，我们还可以获取其它的变化点，最后，在产品线的实现过程中，对变化点可能会带来新的灵感。这是必要，因为某些决策只有在获取更多的信息之后才能确定。

三、支持变化点

架构设计中支持变化点的方式很多，举例如下：

- 在我们面向对象的设计模式中，利用泛化和特化可以实现这种变化。它的特点是系统具有相同的接口但具有不同的行为。
- 把扩展点构建到元素的实现中，也就是放在可以安全的添加额外的行为和功能的的地方。
- 可以通过元素、子系统或子系统集合引入构建参数来完成，这里可能需要使用配置文件，必要的时候可以使用反射。

对于保存在核心资产库中的架构，必须为它编写文档，重点是表达变化点和应用变化点的原理，也应该描述架构实例化的过程，也就是如何应用变化点。文档中必须指出有效的和无效的应用实例，以避免应用上的错误。产品线架构必须经过评估，以确保这个架构是有效的和应用安全的。

6.8 基于产品线的架构设计

不少组织利用产品线架构取得了显著的经济效益，实现产品线的架构设计是一个系统工程，并不是只要把历史上的元素保存下来就可以的，很重要的是我们的组织形式和开发方式要和产品线方式向配合。

产品线系统已有成功的应用实例。典型的是美国空军电子系统中心(ESC)和瑞典 CelsiusTech System 公司的产品线系统。

在 ESC 采用的产品线方法中，主要有五个关键性的组织，即外围的用户、SPO（系统项目办公室），和产品线内部的系统构架组、产品线工程中心和产品线构件支持组。SPO 是直接和用户打交道的组织，由它来决定是开发一个新系统还是从已有的系统升级。SPO 和用户都是产品的客户，它们介入了整个开发阶段，在系统从原型演化到最终部署的过程中进行监控和认证。产品线内部的三个关键性组织分别是系统构架组(SAG)、产品线工程中心(PLEC)、产品线构件支持组(PLAS)。

CelsiusTech 系统公司是瑞典主要的指挥与控制系统供应商，下面我们在这个公司使用产品线的例子，来讨论他们从最初的开发模式引入产品线的过程的有启发性的案例，对这个问题进行分析。

一、开发产品线的动因

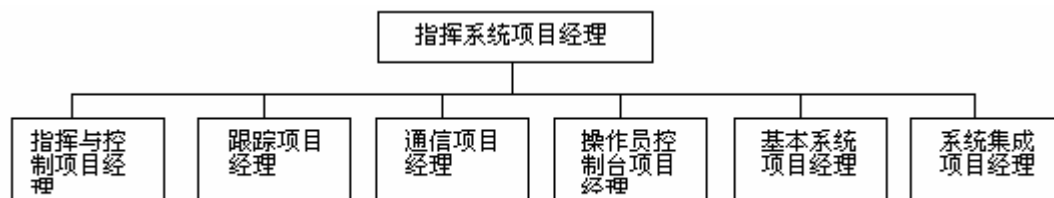
该公司最初的产品，是把具体的武器系统中模拟计算单元，以嵌入式数字单元代替，

它采用的是传统开发方法，但取得了很好的效果。但是后来，该公司同时接到了两份合同，也就是瑞典海军和丹麦海军的舰船系统，两个系统都需要很强的容错性和分散性，比以往开发的系统要复杂得多。

该公司早期的经验，是生产单个系统中的计算单元，但是以前开发单个系统的方法需要大量的投资和人力，在当时的情况下就已经出现了不能保证进度费用超支的问题，面对这样大型的，特别两个结构类似的大型综合系统的并行开发情况，对管理层和高层技术人员提出了严峻的挑战。因此公司的管理者和高级技术人员经过认真研究，决定根本上改变开发模式，采用一种新的产品族系列的开发方法，这就是 SS2000 产品线。

二、组织结构的变更

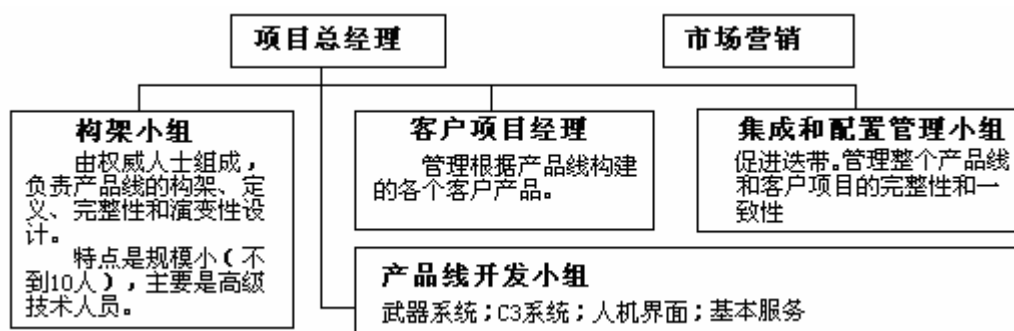
开发方式和理念的改变，必须以组织结构的改变为基础。该公司原来的项目开发组织系统，是以系统功能结构为基础的，以功能领域为基础的项目经理，自开始到系统集成都对整个阶段的人力资源有使用权，基本上如下图：



注：每个项目经理下辖开发小组和集成小组

在系统比较小的时候，这个组织也还是成功的，但还是发现了很多问题，比如为了减少各功能领域的通信，把所分配的系统需求和接口写入文档，但往往接口的不兼容需要到系统集成的时候才能发现，使得在职责分配上浪费了很多时间。系统的集成和安装也耗费了过多的时间。

对于一个大型的系统性的项目，这可能会带来了更多的问题。后来该公司调整了组织结构，他的特点是构架组、产品线开发组和集成组的分离。首先，成立了一个强有力、小规模、、主要注意技术问题的架构小组，用于设计和管理公司的产品线。在这个阶段，主要注意力放在了架构小组，并直接向总经理汇报。



该架构小组主要对以下方面负有责任，并直接向总经理汇报：

- 产品线概念和原则的创建。
- 各层及对外接口的确定。
- 接口的定义、完整性和受控演变。
- 系统功能在各层上的分配。
- 通用机制或者服务的确定。
- 诸如异常处理、进程间通信协议等通讯机制的确定、原型的建立与实施。

- 向项目开发人员传达产品线概念和原则。

该公司最初的架构是由两个高级工程师经过两个星期的研究以后提出来的，包括 125 个系统功能和上面所提出的问题。后来随着产品的开发和升级，又扩充到 200 个功能，整个架构小组稳定为 10 位高级工程师，到现在为止这个构架仍然是现有产品的基本框架。

关于集成和配置管理小组主要负责如下工作：

- 单元测试策略、测试计划和测试用例的开发。
- 所有测试的协调。
- 增量式构建计划的开发。
- 有效子系统的集成和发布。
- 开发和版本库的配置管理。
- 软件交付介质的制作（比如用户手册等）。

而产品线的开发小组，主要是子系统或者模块级的设计和开发。

也就是说，构架组负责产品线系统构架的定义和演化。产品线开发组负责根据产品线系统构架，生产和管理可复用构件。集成组则根据具体客户的需求，利用产品线系统构架和可复用构件进行具体的系统集成。

随着系统的成熟，各个组织的关注点也发生的变化，比如集成组的注意力逐步转入对若干同时开发的系统集成和版本管理，更多地关心各个客户系统上对测试计划和数据集的重用。而产品线开发小组也不再非常强调技术的成熟，而是把重点放在如何立业客户的业务过程和更改客户需求之上。

三、架构解决方案

从架构的角度，我们首先考虑产品最重要的一些质量需求：

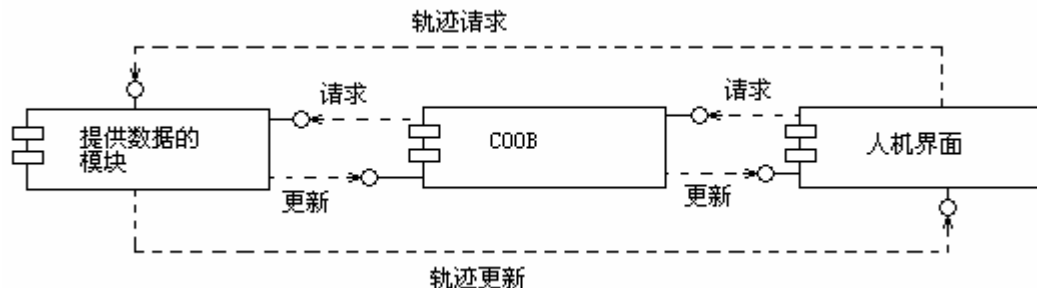
- 性能：系统必须能够对不断到达的传感器输入做相应，并且能控制所要求的分系统。
- 可修改性：对于计算平台、操作系统、添加和更换传感器和分系统、人机接口需求和通信协议的变化，架构都需要具有健壮性。系统应该能不破坏架构和其它部分的情况下，把现有模块更换成针对某个特定系统的模块。
- 安全性、可靠性和可用性。
- 可测试性：每个系统都必须是可集成和可测试的，以快速发现、隔离和纠正错误。

1. 进程视图

事实上，在分布式计算平台上，是把系统构建成一组通信进程，这个问题的研究需要使用进程视图。进程视图的关注点是性能，我们还需要关注诸如：死锁、通信协议、容错性（防止通信线路出现问题）、网络管理及防阻塞的考虑等，我们必须制定一些约定，这些约定不但是分布式系统而且对于构件的开发都是必要的。比如我们提出的约定如下：

- 构件之间的通信是通过强类型消息传递的。抽象数据类型和实施操纵的程序是由传递消息的构件提供的。它使得各个构件可以在不考虑其它构件对数据表示细节的情况下独立编写。其实强类型也包括了具体的数据类型可以在使用方由特定的文件确定。
- 进程间通信协议支持本地独立应用程序之间的数据传输协议，这样就可以保证不考虑各个应用程序所在的物理位置，这种处理器分配的“匿名性”，可以保证应用程序可以在处理器之间迁移。

数据生产者不需要在了解数据使用者的情况下独立编写，数据的维护和更新从概念上是与数据的使用者相分离的，也就是这是一个黑箱模式，数据的主要使用者是人机界面(HCI)。包括存储库的构件称作通用对象管理器(Common Object Manager, COOB)，它的作用如下所示，数据基本上应该通过 COOB 调整，但有时候为了性能，也不排除少数的绕过 COOB 的情况。比如目标的轨迹信息(该目标的历史数据)更新频率特别快，所以它绕过了 COOB。



关于数据生产的约定包括：

- 各个数据应用模块都有本地的数据保留库（称之为实时数据库），仅当数据改动的时候才发送数据去更新这样的“实时数据库”，这样可以保证不必要的消息进入网络。构件拥有自己已更改的数据，这就避免了数据的争用。
- 数据以面向对象的抽象形式表现出来，以便把程序与实现的细节隔离开来，数据必须是强类型的。
- 由于数据是分布式的，所以对访问请求的时间很短。
- 从较长的时间来看，数据在整个系统中是一致的，但允许短时间的不一致。

与网络相关的约定如下：

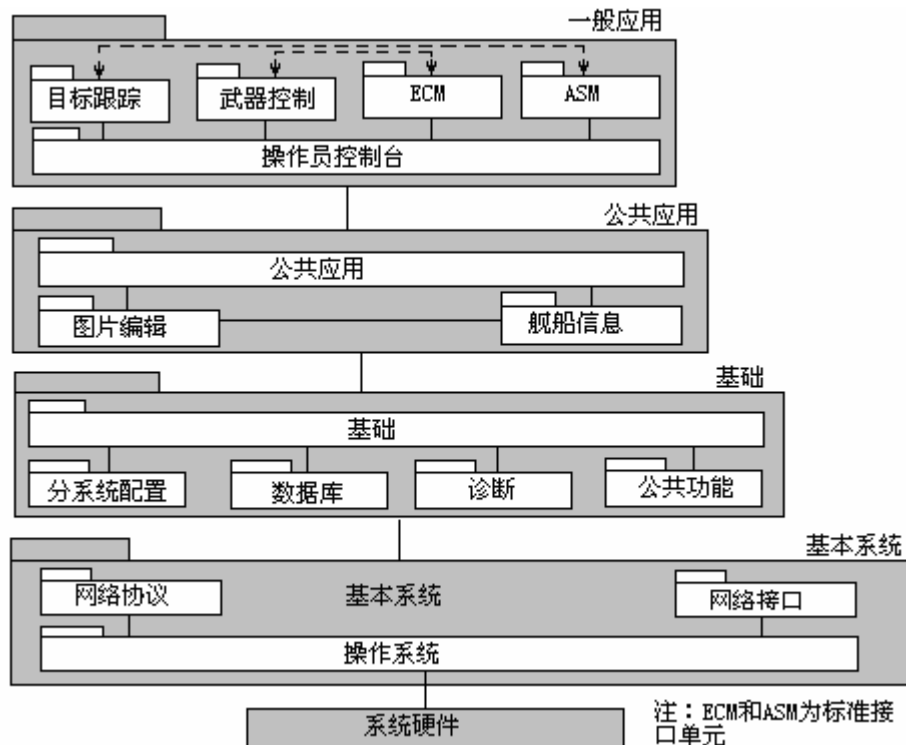
- 从设计上保证了网络负载比较小，这需要设计的时候对数据流作大量的工作，确保所传递的数据都是必要的。
- 数据通道需要具有一定的防错性，尽可能在应用程序内部解决出现的错误。
- 允许应用程序偶尔错过某次更新，例如舰船位置，某次方位数据可能会错过，可以事后根据前后数据进行更新，也就是需要有某种数据“记忆”功能。

这些约定，就可以保证模块在不考虑自己所不能控制的可变部分的情况下独立编写，使模块更具通用型，而且模块的类型限制在比较少的数量，可以为不同系统直接使用。

2，分层视图

该系统实现层的原则如下：

- 模块的分组大致是按照所封装的信息类型划分的。一般来说，当这种类型的信息形式发生变化时，只改变这一个层的内容。比如，关于网络、通信协议发生变化时，必须改动的模块，放在一个层中。
- 层排列的顺序，越是依赖于硬件的层放在底端，越是针对具体应用的层越放在上端。
- 对层的访问限制，模块只能访问同一层和下一层的模块。



3. 模块分解视图

本系统由大约 30 个系统功能组组成，其中每个功能组又包含大约 20 个左右的系统功能。系统功能组是围绕几个大的功能领域组织起来的，它们包括：

- 指挥、控制和通信。
- 武器控制。
- 系统内部通信和计算机环境接口的机制。
- 人机界面。

四、产品线架构的应用

为了实现规定的质量需求，该系统采用了各种相关解决方案。

编号	需求	实现方式	相关解决方案
1	性能	严格的网络通信协议； 把软件编写为一组进程，以使并发最大化； 把软件编写的独立于位置的，通过重新调整位置来调整性能； 绕过 COOB 以实现高速数据交换； 仅在改变和分配的时候发送数据，以使响应时间最短。	引入并发 减少需求 多个副本 增加资源
2	可靠性、可用性和安全性	冗余的 LAN；容错的软件；标准的异常协议； 把软件编写的独立于位置的，是能够在出现故障的时候移植； 数据具有严格的所有权，以防止多人争抢改写数据库的情况。	异常处理 主动冗余 状态再同步 事务
3	可修改性（包括产生新的成员）	严格使用基于消息的通信，提供与实现细节隔离的接口； 把软件编写的独立于位置的，分层提供了跨平台、网络拓扑、网络协议的可移植性； 由于 COOB 的存在，数据的生产者和使用者	语义一致性 预期期望的变更 泛化模块 抽象公共服务 接口稳定性

		不需要了解对方； 大量使用元素参数化； 系统功能和系统功能组提供了语义一致性。	配置文件 构件可更换 遵守已定的协议
4	可测试性	严格的数据所有权、元素的语义一致性以及强大的接口定义，简化了测试中错误责任的发现。	把接口与实现分离

通过上述的一系列措施，SS2000 产品线方法取得了很好的效果，使 CelsiusTech 系统公司获得了巨大的成功，表现在硬件与软件的费用比例从过去的 35:65 变成了 80:20。由于软件的费用得到了良好的控制，该公司现在已经把精力投入到了降低硬件费用上。

使用产品线方法的好处是显而易见的，它将以更小的代价和资源，更快地开发出系统。由于使用的是高度可靠、性能经过验证的可复用构件，产品线系统的质量将大大提高。而且产品线方法还开拓了一个广阔的构件市场，这将极大地促进软件构件业的发展。

下面我们对所采用的架构策略进行总结：

1，把架构作为基础：

事实上不考虑商业、组织结构方面的问题，仅仅靠架构是不足以形成产品线的。在实现产品线的过程中，抽象和分层十分重要。通过抽象，可以把可能改变的决策封装在接口边界之内，当未来发生变化时，不需要改变资产库中的内容。

为了合理的设计，设计者必须对对象领域非常熟悉而且有透彻的理解，对变化、差异等要理解的十分清楚，这样才可能是产品线用于多个差异化的具体产品中。

2，在开发新系统的同时维护资产库

由于产品线架构应用于多个客户目标系统，所以随着需求的变更，产品线架构也会不断变化，所以可充裕资产也会不断更新。但是，不允许可重用模块以独立于产品线的发展沿着独立于产品线的方向发展。但是，可以形成针对某些具体类型的产品集，不过在配置方式上要保证统一和灵活。

3，模块的参数化

在某些情况下，模块的处理可以使用符号来定义参数，以使模型具有通用性。但参数过多也会带来非法操作和冲突的问题，也要注意测试中测试这些参数带来的影响。不过，确实很少有报告说错误操作是由于参数设置不正确造成的。

由于该公司的管理层对产品线方法进行了全面的支持，特别是注意到了从组织结构上与构架方式匹配，从而取得了巨大的经济效益。在这个过程中，关键是管理层赋予了架构小组在整个设计中的权威中心，这样就实现了对概念完整性的维护。他们认识到，了解多个领域的知识，并且具有软件工程技巧的架构师对多个产品线的创建至关重要，同时，新产品开发及产品线改进中，仍然需要领域专家的协助。

五、产品线架构的障碍

应该看到，产品线方法仍然面临着很大的挑战和障碍，这主要体现在：

- **文化上：**产品线策略意味着软件组织和管理者对他们产品开发的直接控制减少了，对其它组织的依赖增加了，这意味着一种思想观念上的转变。
- **战略计划上：**产品线的规划不仅是对一组相关系统的管理过程，还需要考虑用户的长期需要和现有产品线的能力，对未来的发展作出长远规划。
- **需要折衷：**产品线方法需要用户作出折衷，是“独自开发一个恰好是我所需要的系统”，还是“利用产品线开发一个与我的需要非常接近的系统，但可以节省开发的

代价和时间”。

- **资源的所有权**：产品线构件归谁“所有”？在目前的体制下，这的确是一个容易引起纠纷的问题。这就需要整个软件开发和管理组织的重心从当前的程序获取转移到商业化的构件开发上来。
- **提拔和奖励**：当前的开发方法主要是提拔和奖励那些提交了最终系统的开发人员，采用产品线方法后还应该提拔和奖励那些开发构件和促进了产品线开发中构件复用的人员。

六、复用成熟度模型(RMM)

产品线架构依赖于软件复用，对于一个软件企业重复使用“为了复用目的而设计的软件”的能力，受 SEI 的能力成熟度模型(Capability Maturity Model, 缩写为 CMM)的启发，已出现了几个复用成熟度模型(Reuse Maturity Model, 缩写为 RMM)，作为对企业内复用水平层次的度量。

例如，在 IBM 的 RMM 中，将企业的软件复用水平分为五级。随着复用成熟度级别的提高，复用的范围、复用使用的工具和复用的对象都有变化：

1) **初始级(Initial)**：不协调的复用努力。复用是个人的行为，没有库的支持，主要的复用对象是子程序和宏。

2) **监控级(Monitored)**：管理上知道复用，但不作为重点。复用是小组的行为，有非正式的、无监控的数据库，复用的对象包括模块和包。

3) **协调级(Coordinated)**：鼓励复用，但没有投资。复用的范围包括整个部门，有配置管理和构件文档的数据库，复用的对象包括子系统、模式和框架。

4) **计划级(Planned)**：存在组织上的复用支持。在项目级别支持复用，有复用库，复用的对象包括应用生成器。

5) **固有级(Ingrained)**：规范化的复用支持。复用成为整个企业范围的行为，有一组领域相关的复用库，复用的对象包括 DSSA。

HP 的 RMM 将复用成熟度与复用率联系起来，也分为五级：

- 1) **无复用**：-20%至 20%的复用率；
- 2) **挖掘整理**：15%至 50%的复用率；
- 3) **计划复用**：30%至 40%的复用率；
- 4) **系统化复用**：50%至 70%的复用率；
- 5) **面向领域的复用**：80%至 90%的复用率。

复用问题是今天软件工程界热衷于研究的话题，对未来软件工程的走势有重要的影响。

6.9 架构决策

当架构设计基本完成而且已经编制了文档以后，需要做的一件最重要的工作就是架构决策，我们必须知道我们构建的架构对系统重要的质量属性产生的影响，以及对架构方案的取舍做出定。评估大型系统的架构是一项复杂任务，它的困难在于：

- 1，大型系统由于结构庞大，要在有限的的时间里理解这个构架非常困难。
- 2，计算机系统的目的是支持商业目标，评估的时候需要把这些目标和技术支持联系起来。
- 3，大型系统通常都有多个涉众，需要在有限的的时间里把这些涉众的观点仔细管理并加以评估，有的时候是非常困难的。

在进行架构评估的时候，有一些方法论可以帮助我们解决这些问题，比如我们可以使用 ATAM（构架权衡分析方法）是一种评估构架的综合全面的方法，这种方法不仅可以揭示出架构满足特定质量目标的情况，而且可以使我们更清楚地认识到质量目标之间的关系。

ATAM 用以获取系统以及构架的业务目标，并且使用这些目标，通过涉众参与使评估人员把注意力放在为实现这个目标最重要的构架部分上。

一、ATAM 的参与人员

ATAM 要求如下 3 个小组参与合作：

1，评估小组：

该小组时所评估架构项目外部的小组，通常由 3~5 人组成，每个人可能会扮演不同的角色。这个小组可能是常设的，也可能临时组织的。可能从理解架构的人员中挑选出来，也可能是外部咨询人员。在任何情况下，他们必须有能力、没有偏见而且私下也没有其它工作要做的外部人员。评估小组具体的角色如下：

- **评估小组负责人：**准备评估；与评估客户协调；签署评估合同；组建评估小组；最终报告的生成与提交。
- **评估负责人：**负责评估工作；促进场景的得出；管理场景的选择及优先级的过程；促进对照架构的场景评估，为现场分析提供帮助。
- **场景记录员：**在得出场景的过程中，把场景写到白板上，描述场景必须用已达成一致的措辞，如果没有想出这样的措辞，先终止讨论，直到想出来为止。
- **进展书记员：**以电子形式记录评估进展情况；捕获原始场景；捕获促使每个场景的问题；捕获与场景对应的构架解决方案；打印出采用场景的列表并分发。
- **计时员：**帮助评估人员使评估工作按时进行，评估过程中控制用在每个场景上的时间。
- **过程观察员：**记录如何改进评估过程，以及评估如何偏离了原计划。通常不发表意见，但可以在过程中向评估负责人提出经过谨慎考虑的基于过程的建议。
- **过程监督者：**帮助评估负责人记住并执行评估方法的各个步骤。
- **提问者：**提出涉众可能没想到的关于架构的问题。

2，项目决策者：

这些人对开发项目有发言权，并有权要求进行某些改变，他们通常包括项目管理人员、承担开发费用的客户代表也可以列入其中，架构设计师必须参与评估。

3，构架涉众：

涉众包括开发人员、测试人员、维护人员、性能工程师、用户以及与该项目交互的其它项目人员，他们的任务是，清晰而且明白无误地说明架构必须满足的具体质量指标，例如可修改性、安全性以及可靠性等。

二、ATAM 的结果

ATAM 评估至少应该包括如下结果：

- **一个简洁的架构表述：**通常的架构文档是对象模型、接口及其签名的列表。但 ATAM 要求在一个小时内表述构架，这就要求有一个简洁、可理解的架构表述。
- **表述清楚的业务目标：**业务目标能否表述清楚，关系到架构能否无歧义的实现，对于开发小组这尤其重要。
- **用场景集合捕获的质量需求：**业务目标导致质量需求，一些重要的质量需求是用场

景的形式来捕获的。

- **架构决策到质量需求的映射:** 可以根据构架决策所支持或者所阻碍的质量属性来解释构架决策, 对于 ATAM 期间分析的每个质量场景, 确定哪些有助于实现该质量场景的架构决策。
- **所确定的敏感点和权衡点集合:** 这是对一个或者多个质量场景有显著影响的构架决策。比如, 备份数据库是一个架构决策, 它正面的影响了可靠性, 是一个关于可靠性的敏感点。但是保持备份消耗了系统资源, 又负面的影响了系统性能。因此它是可靠性与性能的权衡点, 这个决策的风险在于, 性能成本是不是超过了规定范围。
- **有风险决策和无风险决策:** ATAM 中有风险决策的定义是: 根据所陈述质量属性的需求, 可能导致不期望的结果的架构决策。无风险决策与之对应, 被认为是安全的架构决策。
- **风险主题的集合:** 分析完成以后, 评估小组把发现的所有风险的集合列表, 进而寻找确定架构中系统弱点的总的主题, 如果不采取措施, 这些风险主题将会影响项目的业务目标。

三、ATAM 的阶段

ATAM 的活动分四个阶段:

- **第 0 阶段:** 为合作关系和准备阶段, 评估小组负责人和主要项目决策者进行非正式会议, 以确定此次评估的细节, 项目代表向评估人简要概述项目, 以使评估小组具备适当的专业技术人员的协助。另外对于会议的地点、时间以及后勤保障需要实现达成一致, 对于需要什么样的架构文档也需要达成一致。
- **第 1 和第 2 阶段:** 为评估阶段, 第一阶段, 评估小组和项目决策者会晤 (通常一天时间), 以开始信息收集和分析工作。第二阶段, 构架涉众加入到评估中, 分析继续进行 (一般用两天时间), 后面会详细讨论这两个阶段的步骤。
- **第 3 阶段:** 小组需要生成一个最终的书面报告。在总结会议中, 需要讨论哪些活动比较理想, 还有什么需要自我检查和改进的问题, 以使评估工作一次比一次更好。

下面对第 2 和第 3 阶段进行讨论, 这就是评估阶段的步骤。

第 1 步: ATAM 方法的表述

评估负责人需要向参加会议的项目代表介绍 ATAM, 说明每个人要参与的过程, 回答有关问题, 负责人需要使用一个简单的演示来简要描述 ATAM 步骤和评估的结果。

第 2 步: 商业动机的表述

评估小组成员需要了解促成这个项目开发的主要商业动机, 介绍的问题包括:

- 系统最重要的功能。
- 相关技术、管理、经济和政治的限制。
- 该项目相关的商业目标。主要的涉众。
- 架构驱动因素, 也就是形成该构架的主要质量属性目标。

第 3 步: 构架的表述

设计师表述架构的本质, 在表述中, 设计师首先必须谈到架构的约束条件, 以及满足这些条件所使用的模式。在表述中应该抓住重点和本质, 讲述构架最重要的方面, 而不需要面面俱到甚至关注自己感兴趣但不太重要的方面。在讲述的时候尽可能使用视图而不是文字。

第 4 步: 对架构方法进行分类

通过对每个质量属性使用的模式, 可以对架构方法进行分类, 评估小组应该对这些模式和方法进行明确的命名。应该注意到每个方法都影响特定的质量属性, 但也可能造成其它不

良的影响。比如分层模式提供了可移植性，但很可能是牺牲性能为代价的。

这个分类表应该由记录员记录下来，以供所有人传阅。

第5步：生成质量属性效用树

由于质量属性极大的影响了架构设计，而且不同的架构设计方法对不同的质量属性的影响可能是矛盾的，比如对安全性要求极高的系统，性能极高的架构可能完全是错误的。

所以，我们必须对质量属性的优先级进行分配，可以生成质量属性效用树，这种树实际上是一个鱼骨图，从不同的动机中寻找最主要的原因，具体的方法在系统分析中已经讨论过了，这里不再重复。

第6步：分析架构方法

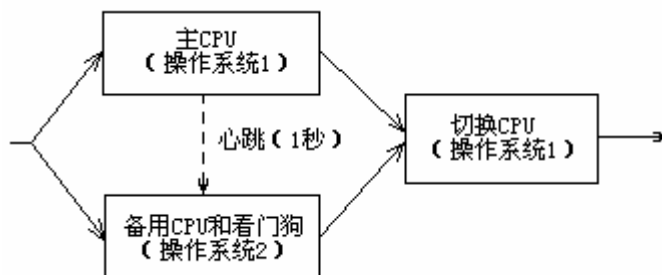
在这里，评估小组每次分析一个最高优先级的场景。方法是设计师解释架构如何支持这个场景，小组成员通过提问探查设计师用来实现场景的构架方法。在这个过程中，评估小组把相关架构决策编成文档，确定它的有风险决策、无风险决策、敏感点、权衡点，并且对它进行分类。必要时说明设计师是如何避开这种架构的弱点并保证该方法满足要求的。

评估小组的目标是，确信这个方法能够达到质量属性的要求。

下面给出一个场景构架方法分析的表格，需要整体上把有风险决策、无风险决策、敏感点、权衡点列成单独的表，表中 R8、T3、S4 等代表这个表中的条目。

场景号:	场景：检测主交换机的硬件故障并从中恢复过来			
属性	可用性			
环境	正常操作			
刺激	某个 CPU 不能正常工作了			
响应	切换可用概率是：0.999999			
架构决策	敏感点	权衡点	有风险决策	无风险决策
备用 CPU	S2		R8	
无备用数据通道	S3	T3	R9	
看门狗	S4			N12
心跳	S5			N13
故障切换路由	S6			N14
推理	<p>通过使用不同的硬件和操作系统，保证不出现通用模式故障（参见风险决策 R8）。</p> <p>最坏情况下，4 秒钟（运算状态的最长时间）内完成恢复。</p> <p>根据心跳和看门狗的速度，保证 2 秒之内检测到故障。</p> <p>看门狗简单可靠（已经过验证）。</p> <p>由于缺少备用数据通道，因此可用性存在风险（参见有风险决策 R9）</p>			
架构图	见附件 1			

附件 1：架构图



至此第一阶段结束，评估小组需要对所获得的知识进行总结，并在 1 到 2 周的中断时间内与设计师进行非正式会晤（通常用电话形式），如果必要，在这个阶段可以分析更多的场

景，也可以解决已澄清的问题。

当评估决策者准备就绪，把评估组成员再召集到一起的时候，第二阶段就开始了。

此时应该重复前面的第一步，重申在这个阶段每个人所扮演的角色，而且需要概述一下第 2~6 步的结果，并给每个人一份有风险决策、无风险决策、敏感点、权衡点的当前列表，当大家熟悉了以评估结果以后，就可以进行下面的 3 步了。

第 7 步：集体讨论并确定架构的优先级

上面第 5 步生成的质量属性效用树，主要是为了了解架构设计师是如何看待质量属性构架驱动因素的，对场景进行集体讨论，则是为了了解多数涉众的看法，在参与评估的人员比较多时，对场景进行集体讨论非常有效，可以创造出一个人的想法激发其他人灵感的氛围。

这种讨论过程，能够促进相互交流、发挥创造性、并起到表达参评人员共同意愿的作用。如果集体讨论确定了优先级的一组场景与效用树进行比较，如果相同，则说明设计师所想的与大多数涉众的想法基本是一致的。如果又发现了其它驱动场景，则可能存在一个风险，表明涉众与设计师的想法不一致。

在集体讨论中，不同的涉众对场景的要求可能是不一样的，比如，维护人员可能会更关注易修改性，最终用户可能会关注一个功能或者易操作性，这就需要讨论和平衡。我们鼓励涉众考虑在效用树中尚未分析过的场景，或者是在前面的分析中被认为没有引起足够重视的场景。

确定了场景以后，就必须划分优先级，我们必须注意把有限的评估时间用在最重要的地方。首先把涉众认为代表相同行为或者质量属性的场景合并起来，然后投票。

投票的方法是，每个涉众拿到总场景数 30% 的选票（此数只入不舍，比如 20 个场景，每个人 6 张选票），投票时可以任意使用这些选票，可以全投给一个场景，也可以一个场景投一张。

每个涉众都要公开投票，然后评估负责人对场景进行排序，选择“某得票数之上的”场景，供后续步骤使用。

第 8 步：分析架构方法

对于已经收集的若干场景并且确定了优先级以后，评估小组引导设计师实现第 7 步中得到的优先级最高的场景。设计师对相关的构架决策如何有助于实现每个场景作出解释。理想情况下，设计师使用已经讨论过的构架方法对这些场景作出解释。

第 9 步：结果的表述

最后，把 ATAM 分析中得到的各种信息进行归纳总结，并且呈现给涉众。一般来说应该有一个书面报告，包括商业环境、促成该构架的主要需求、约束条件和架构策略等，然后表述如下结果：

- 已经写了文档的构架方法。
- 经过讨论得到的场景和优先级。
- 效用树。
- 所发现的风险决策。
- 已经编成文档的无风险决策。
- 所发现的敏感点和权衡点。

我们在评估过程中得到了这些结果，并且对它进行了讨论分类，同时还可以根据一些常见的基本问题或者系统缺陷把风险分解为风险主题，比如没有足够的重视文档编写、未提供备份能力或者为提高可用性等。

在这个过程中，很可能把某个经理认为是技术层面的风险，提高为他该关心的某个问题的威胁，这都可能改变后期很多问题的做法。

6.10 关于架构的重要结论

通过以上讨论，我们得到了一些关于架构设计的有启发性的结论：

1，架构设计是保证软件质量的最重要的要素之一，因此必须由有多年工作经验、对相关领域知识了解透彻、对质量和项目管理理解深入的人来担任系统构架的工作，架构人员的想象力是至关重要的。

2，需求分析的系统全面，是架构设计的必要条件，因为我们必须花大力气改善需求分析工作，尤其是需求分析人员必须受过很深的专业训练，对问题点的把握要准确。

3，软件质量标准决定了架构的风格，所谓高质量软件对于不同的用户要求是完全不一样的，因此架构设计必须把质量问题放在重要位置认真研究，以寻找恰当的架构策略。

4，研究质量就必须把每个质量属性细化，建立质量属性场景，并对每个场景确定解决方案，最后综合出架构策略，最后的策略应该经过权衡，把设计的关注点集中在最主要的方向上。

5，优秀的产品线可以带来巨大的经济效益，但产品线并不仅仅是技术，必须以合适的组织形式与之协调，与此同时，项目管理与过程也必须依据产品线方式进行过程改进，这样才可能使架构的特点得到充分的发挥。

第七章 设计模式与小粒度架构设计

在高层架构设计中，我们已经详尽的讨论了为达到系统的质量目标而采取的架构策略，但是这一策略的实现，必须依靠小粒度架构的良好设计。

小粒度设计阶段考虑的主要是模块结构设计，它着力于解决如下问题：

- 1，给出软件结构中各模块的内部过程描述。
- 2，模块的内部过程描述也就是模块内部的算法设计。
- 3，模块设计有时需要导出一些算法设计表示，由此可以直接而简单地导出程序代码。

模块结构设计直接对应于实现编码，因此设计质量直接影响着软件的质量。为了合理的规划模块之间的关系以及模块内部的各个类之间的关系，就需要提出一些原则，这就是设计模式提出的原因和背景。设计模式也称之为微观架构模式，这种小尺度的对象和框架的设计，有时候也需要资深程序人员参与。

7.1 软件重构技术

关于系统可维护、可集成性的实践，引发了人们对于这类问题理论上研究的兴趣，一个直接的成果就是软件重构技术的提出。软件重构技术是对软件的内部结构所作的一种改变，这种改变在可观察行为不变的条件下使软件更容易理解，而且修改更廉价。

一、为什么要研究重构技术

在软件开发与维护的长期实践中，人们普遍认识到的一个事实是代码太容易变坏。

坏的代码总是趋向于有更大的类、更长的方法、更多的开关语句和更深的条件嵌套。特别是那些初看相似细看又不同的代码泛滥于整个系统：条件表达式，循环结构、集合枚举…。

全部代码都以非常密集的样式被书写，你根本看不到这种代码还有什么良好的设计。

当项目进行或者完成之后，需求被改变，功能被增加，程序员改变代码，所有这些情形都倾向于在非常紧的工期下发生。这些因素意味着代码不被维护，只是被扩展，导致难于阅读和理解的复杂代码。随着时间的漂移，代码中只有很少的一部分代表了系统的设计。

如果要在这样一种系统上添加功能，第一个反应应该是拒绝。因为这样的代码难以理解，更不要说对它加以修改。存在的第二种可能性是抛弃现在的系统，然后从头编写。如果项目的规模较大，或者系统已经在运行，不可能进行重新编写，这时人们就得面对第三种选择，硬着头皮进行维护，进行修改和扩充。

维护人员通常采取一种快速和消极的方法。如果系统有问题，那么就尽快地找到问题，直接修改它。如果要增加一项新功能，就会从原来的系统中找到一块相近的代码，拷贝出来，作些修改，再粘贴进去。这样一来，系统变得越来越难以理解，维护越来越困难、越来越昂贵。系统变成了一个十足的大泥潭。每个人都不愿意看到这种情况，但奇怪的是，这样的情形却一次又一次地在现实中不断地出现。

所有维修都倾向于破坏结构，增加了系统的凌乱。在修理原始设计缺陷上所花的时间越来越少，越来越多的时间是花在修理由早期修理所引入的错误上。随着时间的流逝，系统变得越来越混乱。迟早修理变得不可能而停止下来。

软件重构技术的动机是研究主要包括软件重用、软件维护、和软件的重新组织 (Software Restructuring)。

1, 软件重用 (Software Reuse)

为了降低开发软件的高成本,软件重用的研究是为了使一个系统开发的知识再次容易地用于另一个软件系统的开发。然而,可重用软件经常需要很多设计迭代。使软件更容易改变将使设计迭代更简单。这样的软件将更可重用。

抽象、封装、继承、多态和模块性这样的面向对象程序设计特性给软件再利用提供了基础结构,以鼓励再利用现有的代码,而不是从头开始编码。这样,通过添加新类或者在现存类上添加操作能对软件作出某些改变,而使软件的大部分保持不变。

除代码级再利用之外,设计级的再利用也是研究的内容。而且,从长期的观点来看,人们认识到设计级的再利用更为重要。面向对象应用框架 (framework) 是这种研究努力的结果。框架是抽象和具体类的集合。从而,能够添加新的子类对它进行重定义。因此,框架支持抽象级,并允许部分的规范说明。

2, 软件维护 (Software Maintenance)

软件重用与软件维护紧密相关。维护是软件生产所有方面中最为困难的。主要的理由在于维护包容了软件过程所有其他阶段的各个方面内容。在软件生命周期中,在维护上所花的时间比任何其它阶段都更多。实际上,现行软件的维护工作量能占到全部开发工作量的 60 % 以上。软件维护经常需要重新组织软件。

3, 软件重新组织 (Software Restructuring)

不适当的设计方法学,缺乏开发和维护标准等诸如此类的很多因素都能导致拙劣的软件结构。只存在一些不对代码进行更改的方法。例如,人们能够在软件再工程期间从代码和现有的文档出发,重新创建软件的结构。

二、重构的定义

重构是以各种方式对一对象设计进行重新安排使之更灵活并且 / 或者可重用的过程。效率和可维护性可能是进行重构最重要的理由。

重构定义为名词形式和动词形式两部分:

重构 (Refactoring, 名词): 是对软件的内部结构所作的一种改变,这种改变在可观察行为不变条件下使软件更容易理解,而且修改更廉价。

重构 (Refactor, 动词): 应用一系列不改变软件可观察行为的重构操作对软件进行重新组织。

这些定义中最重要方面是不改变软件系统的可观察行为,并且改变软件结构是朝着更好的设计和更能理解,而且可重用的方向进行。

重构的名词形式就是说重构是对软件内部结构的改变,这种改变的前提是不能改变程序的可观察的行为,这种改变的目的就是为了让它更容易理解,更容易被修改。

动词形式则突出重构是一种软件重构行为,这种重构的方法就是应用一系列的重构操作。

三、重构的原则

1, 一个时刻只做一件事情

如果你使用重构开发软件,你把开发时间分给两种不同的活动:增加功能和重构。

增加功能时,你不应该改变任何已经存在的代码,你只是在增加新功能。这个时候,你增加新的测试,然后让这些新测试能够通过。当你换一顶帽子重构时,你要记住你不应该增加任何新功能,你只是在重构代码。你不会增加新的测试。只有当重构改变了一个原先代码

的接口时才改变某些测试。

在一个软件的开发过程中，你可能频繁地交换这两件工作。

关于两件工作交换的故事不断地发生在日常开发中，但是不管你做哪件工作，一定要记住一个时刻只做一件事情。

2. 小步前进

保持代码的可观察行为不变称为重构的安全性。重构的另一个原则是小步前进，即每一步总是做很少的工作，每做少量修改，就进行测试，保证重构的程序是安全的。如果你一次做了太多的修改，那么就有可能介入很多的错误，代码将难以调试。

这些细小的步骤包括：确定需要重构的位置，编写并运行单元测试，找到合适的重构并进行实施，运行单元测试，修改单元测试，运行所有的单元测试和功能测试等。如果按照小步前进的方式去做重构，那么出错的机会可能就很小。

小步前进使得对每一步重构进行证明成为可能，最终通过组合这些证明，可以从更高层次上来证明这些重构的安全性和正确性。

四、重构的目标和本质

从概念上讲重构的目标是对软件系统的设计进行重新组织，使系统满足某些通用设计的准则，并具有容易扩展系统功能的结构或者形状。在重构应用与实践中，设计模式（design pattern）为重构提供了一个明确的目标。

重构的实质是在保持可观察行为不变的前提下，为提高软件的可理解性，可扩展性和可重用性而对软件进行的修改。

五、重构的组成与步骤

重构由许多小的步骤组成。当一次对系统作了很多改变时，在此过程中也极有可能引入许多错误。但产生这些错误的的时间和地点是不可再现的。如果以小步前进的方式实现对系统的改变，并在每一步后运行测试的话，错误就有可能在它引入系统后的测试中立即表现出来。然后对每步的结果进行检查，如果有问题，可撤消此步所作的改变。在复原之后，可以采取更小的步骤前进。

重构软件系统采取的步骤如下：

1. 确定需要重构的位置。可以通过理解，扩展，或者重新组织系统来发现问题，或者通过查看实际代码中的代码味道（code smell）来确定位置，或者通过某些代码分析工具。
2. 如果所考虑的代码的单元测试存在的话，就运行该单元测试看看能否正确的完成。否则，编写所有必要的单元测试并运行它们。
3. 通过思考或者查看重构分类目录，找出能够明显被应用的重构操作。
4. 遵循小步前进的原则实现重构操作。
5. 在每步间运行测试以保证行为未被改变。
6. 如有必要，对作过改变的接口修改测试代码。
7. 当重构操作被成功地用于重新组织代码，再次运行测试，集成并运行全部的单元测试和功能测试。

六、重构的优点与风险

重构增加了具有某种缺点的程序的价值。难读的程序难修改。具有重复逻辑的程序难修

改。带有复杂条件逻辑的程序更难修改。重构虽然需要更多的“额外工作”，但是它给我们带来的各种好处显然值得我们做出这样的努力。

1, 简化测试

一个好的重构实现能够减少对新设计的测试量。因为重构的每一步都保持可观察的行为，也就是保持系统的所有单元测试都能顺利通过。所以只有发生改变的代码需要测试。这种增量测试使得所有的后续测试都建立在坚实的基础之上，整个系统测试的复杂性大大降低。

2, 增进软件可理解性

程序编写是人的活动，人首先要理解才能行动。所以，源代码的另一个作用就是用于交流的工具。其他人可能会在几个月之后修改代码，如果连理解代码都做不到，又如何完成所需的修改呢？

我们通常会忘掉源代码的这种用处，尽管它可能是源代码更重要的用处。不然，我们为什么发展高级语言、面向对象语言，为什么我们不直接使用汇编语言甚至是机器语言来编写程序？

如果一个人够理解我们的代码，他可能只需要一天的时间完成一个增加功能的任务，而如果不理解我们的代码，可能需要花上一个礼拜或更长的时间。

这里的问题是，我们在编写代码的时候不但需要考虑计算机 CPU 的想法，更要把以后的开发者放在心上，除非，你写代码的唯一目的就是把它丢掉。

重构可以使得你的代码更容易理解。原因在于重构支持更小的类、更短的方法、更少的局部变量、更小的系统耦合，重构要求你更加小心自己的命名机制，让名字反映出你的意图。如果哪一块代码太复杂以至于难于理解，你都需要对它进行重构。

3, 改善软件设计

大部分的软件工程师认为代码仅仅是设计的附属物。但我们必须正视的情况是，程序设计在实现阶段完成的正是代码，而不是你脑袋里或纸上的设计。而绝大多数的故障也产生于编码阶段。在当时或者以后找出这些故障被事实证明是非常昂贵的。

很多方法学强调把注意力放在分析和设计阶段，把实现定义为按照设计进行编码，以努力防止产生故障。这些方法学认为通过分析和设计的严格化就能产生更高质量的软件。然而，这仅仅是理论，实际上是不可能的。

另一方面，即使一开始的设计是完好的，随着用户对系统使用的深入，新的需求可能会被加入，旧的需求会被修改、删除。设计不可能完全预料到这些变化。一旦实现开始偏离最初的设计，那么它的代码将不受控制，从而不可避免地开始腐化（decay）。代码加入越多，腐化的速度越快。如果没有办法让设计尽可能地与实现保持一致，那么这种腐化的最后结果就是代码不得不被抛弃。

在传统的软件方法中，一旦开发到了实现阶段，就很难对设计做出变化。由于程序员编写的代码倾向于腐化。它慢慢地偏离最初的设计，代码偏离最初的设计越远，反映设计的代码就越难被看到，代码就越容易变坏。重构有助于把贬值的代码变得有用，并获得好的设计。

4, 利于找出错误

软件调试中最困难和最费时间的工作是定位错误。重构不仅帮助理解代码，同时也可以帮助发现错误。因为故障隐藏在系统中一些非常微妙的地方，清理设计和代码使故障被带到明处。

重构使得程序代码的结构更清晰，每一个时刻可以更集中关注专一的数据和行为，这会使你的工作量大大减少，效率大大提高。同时，由于重构要求小步前进，并且对每一步都进行严格的测试，这样会使得错误很容易地浮现出来。所以重构期间进行的测试对于捕捉由于测试不足而早期未被发现的故障是很有用的。

5, 加速开发

重构的以上优点是明显的, 重构能够加速开发的优点却不那么明显。人们甚至认为重构会减慢开发速度。因为需要编写额外的单元测试, 还要去重构那些本来就能够工作的代码。

事实上, 重构确实能够加快软件的开发速度。一个好设计的基本前提就是允许更快的软件开发。如果没有一个好的设计, 一开始可能可以开发得很快, 但是随着功能的增加, 代码逐渐腐化, 结构渐渐失去。每次需要加入新功能时, 必须花大量的时间去理解原来的代码, 修改原来代码的错误, 改变一项功能需要花费越来越长的时间。

重构支持好的结构、设计和理解性, 它让人们更快地开发软件。因为它可以防止软件的腐化, 甚至用于改进设计。

6, 有助于代码复审

重构也能对代码复审做出贡献。它被用于使复审的代码更容易理解, 这种范围更广的理解导致更有用的建议。当把这些建议作为展开重构的起点, 由复审者即刻进行实现的话, 代码复审就能交付具体的结果。通过重构, 重建设计的结构, 代码变得更容易阅读, 代码复审变得更成功。

7, 增加灵活性

在传统的开发中, 需要的灵活性必须预设并体现在系统的设计中, 这使得灵活性成为必需付出较高代价的事情。如果使用重构技术, 灵活性不再是开发过程中强加的约束。

重构降低初始设计的复杂程度。模式有其成本(间接性、复杂化), 因此设计应该达到需求所要求的灵活性, 而不是越灵活越好。

如果在设计期间试图介入太多以后可能需要的灵活性, 就会产生不必要的复杂和错误。重构能够以多种方式扩展设计。他鼓励为手头的任务建立刚好适合的解决方案, 当新的需求来到时, 可以通过重构扩展设计。

七、重构的不足和风险

1, 重构的成本

重构的成本取决于所处的环境支不支持某些重构的基本原则。因为重构极大地依赖于每一小步后的测试, 因而拥有一套可靠的单元测试工具就能大量地降低手工测试成本。

应用重构涉及改变接口、名字、参数表等等, 更新项目文档的成本不应低估。这就导致整个系统设计的改变。由于在重构步骤之间老接口首先被保持, 以后逐渐被新的所扩展, 所以, 全部测试都不用改变, 应该在整个重构期间内运行。之后当老的接口被取消, 而支持新接口时, 测试也必须被更新。

2, 重构的风险

由重构引起的软件失效可能具有严重的后果。因此不应该轻易地对待重构, 必须小心奕奕, 并在在头脑中想着可能发生的问题。小步前进, 每步后进行测试, 以可预见的方式作改变。

重新组织不要与添加新功能混在一起工作, 这些重构原则使得不可能在重构时引入错误。尽管如此, 程序员不可能不犯错误。

因此在重构应用于系统之后, 系统不仅应该通过必不可少的单元测试, 而且也应该通过回归功能测试, 以表明重构后的系统与上一次的运行没有差别。即使在重构时引入了错误, 由于重构使系统具有良好结构, 不带有重复代码, 清晰的层次, 以及小的单元, 所以追踪这些错误(一旦它们表现出来)将变得容易得多。

事实上, 重构的一个重要的风险在于, 需要一支高水平的程序员队伍, 他们必须对重构技术有深入的理解, 而不至于使系统不受控的偏移到不希望的方向, 这对过程控制的水平提

出了更高的要求。

7.2 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，我们需要研究一下软件重构技术。而重构技术影响最大而且最成功的应用，要数 GoF 的 23 种设计模式，在 GoF 中，把设计模式分为结构型、创建型和行为型三大类，从不同的角度讨论了软件重构的方法。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：

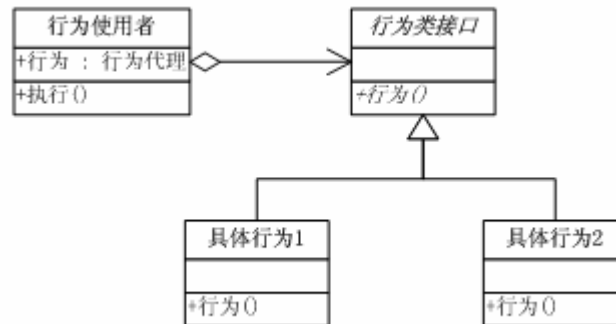
- 1) 开放-封闭原则；
- 2) 从场景进行设计的原则；
- 3) 包容变化的原则。

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 Java 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（C#、C++）来讨论这些问题。

7.3 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();
    }
}
```

7.4 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

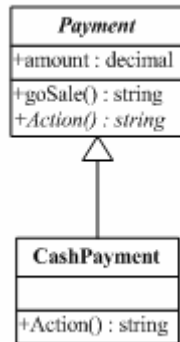
- 1) 一次性实现一个算法的不变部分，并且将可变的行为留给子类来完成。
- 2) 各子类公共的行为应该被提取出来并集中到一个公共父类中以避免代码的重复。首先识别现有代码的不同之处，并且把不同部分分离为新的操作，最后，用一个调用这些新的

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 double amount;
    public double  getAmount(){
        return amount;
    }
    public void setAmount(double value){
        amount = value;
    }
    public 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();
}

class CashPayment extends Payment{
    public String Action(){
        return "现金支付";
    }
}

测试：
  
```

```

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

```
Payment o;  
o = new CashPayment();  
o.setAmount(555);  
System.out.println(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 extends Payment {  
    public String Action() {  
        return "信用卡支付, 联系支付机构";  
    }  
}  
  
class CheckPayment extends Payment {  
    public String Action() {  
        return "支票支付, 联系财务部门";  
    }  
}
```

调用:

```
public class Test {  
    public static void main (String[] args) {  
        Payment o;  
        o = new CashPayment();  
        o.setAmount(555);  
        System.out.println(o.goSale());  
        o = new CreditPayment();  
        o.setAmount(555);  
        System.out.println(o.goSale());  
        o = new CheckPayment();  
        o.setAmount(555);  
        System.out.println(o.goSale());  
    }  
}
```

二、简单工厂 (Simpleness Factory) 模式

1、意图

简单工厂的作用是实例化对象, 而不需要客户了解这个对象属于哪个具体的子类。

在 GoF 的设计模式中并没有简单工厂, 而是把它作为工厂方法的一个特例加以解释, 可以这样来理解, 简单工厂是参数化的工厂方法, 由于它可以处理粒度比较大的问题, 所以还是单独列出来比较有利。

2、使用场合和效果

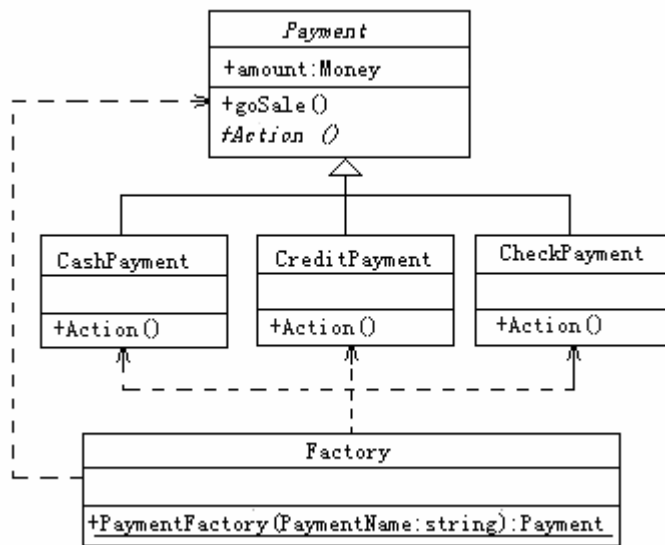
简单工厂实例化的类具有相同的接口, 类的个数有限而且基本上不需要扩展的时候, 可以使用简单工厂。

使用简单工厂的优点是用户可以根据参数获得类的实例，避免了直接实例化类的麻烦，降低了系统的耦合度。缺点是实例化的类型在编译的时候已经确定，如果增加新的类，需要修改工厂。

简单工厂需要知道所有要生成的类型，在子类过多或者子类层次过多的时候并不适合使用。

3、结构

通常简单工厂不需要实例化，而是采用静态方法来实现。下面是一个简单工厂的基本框架，Factory 类为工厂类，里面的静态方法 `PaymentFactory` 决定了实例化哪个子类，而在这个方法里面，通过多分支语句来控制具体实现的子类。



// Payment.java

```

public abstract class Payment{
    private double amount;
    public double getAmount(){
        return amount;
    }
    public void setAmount(double value){
        amount = value;
    }
    public 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();
}
  
```

```

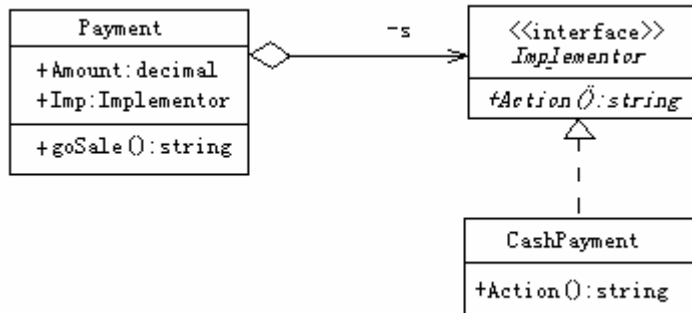
class CashPayment extends Payment{
    public String Action(){
        return "现金支付";
    }
}
  
```

```
    }  
}  
  
// CreditPayment.java  
public class CreditPayment extends Payment{  
    public String Action(){  
        return "信用卡支付,联系支付机构";  
    }  
}  
  
class CheckPayment extends Payment{  
    public String Action(){  
        return "支票支付, 联系财务部门";  
    }  
}  
  
//这是一个工厂类 Factory.java  
public class Factory{  
    public static Payment PaymentFactory(String PaymentName){  
        Payment mdb=null;  
        if (PaymentName.equals("现金"))  
            mdb=new CashPayment();  
        else if (PaymentName.equals("信用卡"))  
            mdb=new CreditPayment();  
        else if (PaymentName.equals("支票"))  
            mdb=new CheckPayment();  
        return mdb;  
    }  
}  
  
调用:  
public class Test{  
  
    public static void main (String[] args){  
        Payment o;  
        o = Factory.PaymentFactory("现金");  
        o.setAmount(555);  
        System.out.println(o.goSale());  
        o = Factory.PaymentFactory("信用卡");  
        o.setAmount(555);  
        System.out.println(o.goSale());  
        o = Factory.PaymentFactory("支票");  
        o.setAmount(555);  
        System.out.println(o.goSale());  
    }  
}
```

三、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。

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 double amount;
    public double  getAmount(){
        return amount;
    }
    public void setAmount(double value){
        amount = value;
    }
    private Implementor imp;
    public void setImp(Implementor s){
        imp=s;
    }
    public String goSale(){
        String x = "不变的流程一 ";
        x += imp.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;
    }
}

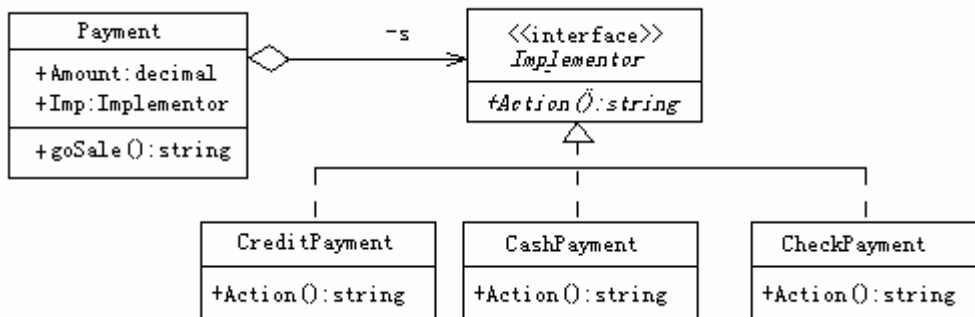
interface Implementor{
    public String Action();
}

class CashPayment  implements Implementor{
    public String Action(){
        return "现金支付";
    }
}
  
```

调用：

```
public class Test{
public static void main (String[] args){
    Payment o=new Payment();
    o.setImp(new CashPayment());
    o.setAmount(555);
    System.out.println(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 implements Implementor{
    public String Action() {
        return "信用卡支付, 联系支付机构";
    }
}
class CheckPayment implements Implementor{
    public String Action() {
        return "支票支付, 联系财务部门";
    }
}
```

调用：

```
public class Test{
public static void main (String[] args){
    Payment o=new Payment();
    o.setImp(new CashPayment());
    o.setAmount(555);
    System.out.println(o.goSale());
    o.setImp(new CreditPayment());
    o.setAmount(555);
    System.out.println(o.goSale());
    o.setImp(new CheckPayment());
    o.setAmount(555);
}
```

```

System.out.println(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。

四、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，是使用超类，但是超类使用了继承而提升了耦合性。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。

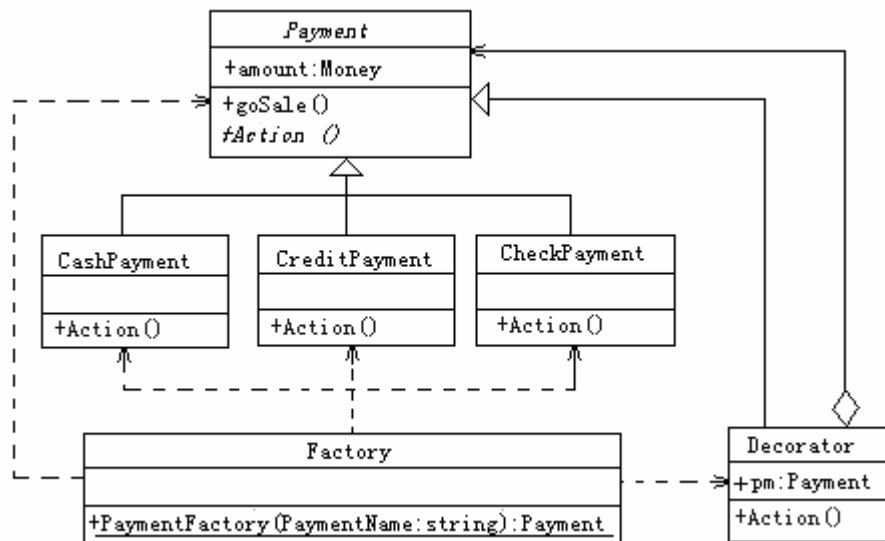
1、意图

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。

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。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。



//装饰类

```

public class Decorator extends Payment {
    private String strName;
    public Decorator(String strName) {
        this.strName = strName;
    }
    private Payment pm;
    public void setPm(Payment value) {

```

```
        pm = value;
    }
    public String Action() {
        //在执行原来的代码之前，显示提示框
        System.out.println(strName);
        return pm.Action();
    }
}
```

而工厂类：

//这是一个工厂类

```
public class Factory{
    public static Payment PaymentFactory(String PaymentName) {
        Payment mdb=null;
        if (PaymentName.equals("现金"))
            mdb=new CashPayment();
        else if (PaymentName.equals("信用卡"))
            mdb=new CreditPayment();
        else if (PaymentName.equals("支票"))
            mdb=new CheckPayment();
        //return mdb;
        Decorator m=new Decorator(PaymentName);
        m.setPm(mdb);
        return m;
    }
}
```

可以说，这是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。

7.5 利用观察者模式延长架构的生命周期

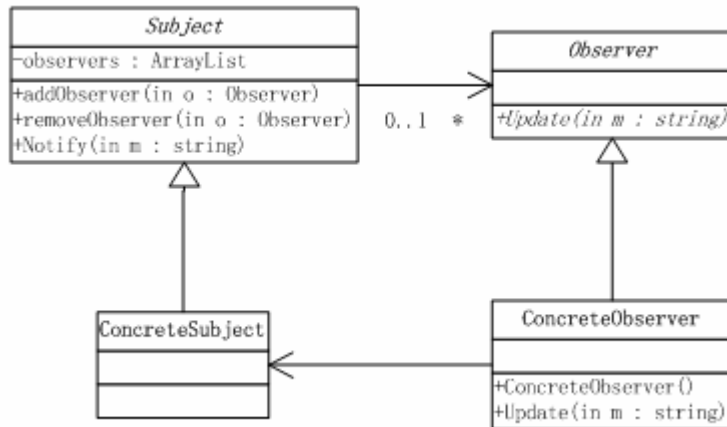
当需要上层对底层的操作的时候，可以使用观察者模式实现向上协作。也就是上层响应底层的事件，但这个事件的执行代码由上层提供。

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，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:

// PaymentEvent.java 传入数据的类

```
import java.util.*;
```

```
public class PaymentEvent extends EventObject {
```

```
    private String Text;    //定义 Text 内部变量
```

```
    //构造函数
```

```
    public PaymentEvent(Object source,String Text) {
```

```
        super(source);
```

```
        this.Text=Text;    //接收从外部传入的变量
```

```
    }
```

```
    public String getText(){
```

```
        return Text;        //让外部方法获取用户输入字符串
```

```
    }
```

```
}
```

//监听器接口

//PaymentListener.java

```
import java.util.*;
```

```
public interface PaymentListener extends EventListener {
```

```
    public String Action(PaymentEvent e);
```

```
}
```

// Payment.java 平台类

```
import java.util.*;
```

```
public class Payment{
```

```
    private double amount;
```

```
    public double getAmount() {
```

```
        return amount;
```

```
    }
```

```
    public void set(double value){
```

```
        amount = value;
```

```
    }
```

```
    //事件编程
```

```
    private ArrayList elv; //事件侦听列表对象
```

```
//增加事件侦听器
public void addPaymentListener(PaymentListener m) {
    //如果表是空的，先建立它的对象
    if (elv==null){
        elv=new ArrayList();
    }
    //如果这个侦听不存在，则添加它
    if (!elv.contains(m)){
        elv.add(m);
    }
}
//删除事件侦听器
public void removePaymentListener(PaymentListener m) {
    if (elv!= null && elv.contains(m)) {
        elv.remove(m);
    }
}
//点火 ReadText 方法
protected String fireAction(PaymentEvent e) {
    String m=e.getText();
    if (elv != null) {
        //激活每一个侦听器的 WriteTextEvent 事件
        for (int i = 0; i < elv.size(); i++) {
            PaymentListener s=(PaymentListener) elv.get(i);
            m+=s.Action(e);
        }
    }
    return m;
}
public String goSale() {
    String x = "不变的流程一 ";
    PaymentEvent m=new PaymentEvent(this,x);
    x = fireAction(m); //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: Test.java

```
import java.util.*;
public class Test {
    //入口
    public static void main(String[] args) {
```

```
Payment o1 = new Payment();
Payment o2 = new Payment();
Payment o3 = new Payment();
o1.addPaymentListener(new PaymentListener() {
    public String Action(PaymentEvent e) {
        return e.getText()+" 现金支付 ";
    }
});
o2.addPaymentListener(new PaymentListener() {
    public String Action(PaymentEvent e) {
        return e.getText()+" 信用卡支付 ";
    }
});
o3.addPaymentListener(new PaymentListener() {
    public String Action(PaymentEvent e) {
        return e.getText()+" 支票支付 ";
    }
});
o1.set(777);
o2.set(777);
o3.set(777);
System.out.println(o1.goSale());
System.out.println(o2.goSale());
System.out.println(o3.goSale());
}
```

7.6 利用策略与工厂模式实现通用的框架

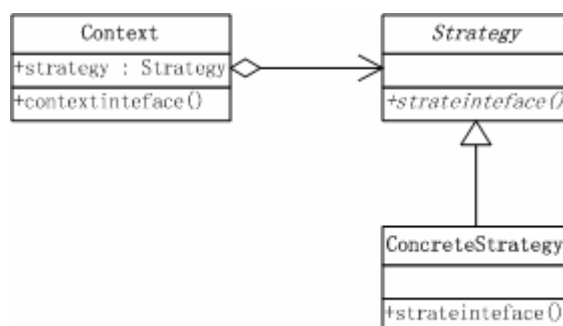
一、应用策略模式提升层的通用性

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。

二、示例：利用反射实现通用框架

目标：

构造一个 Bean 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。

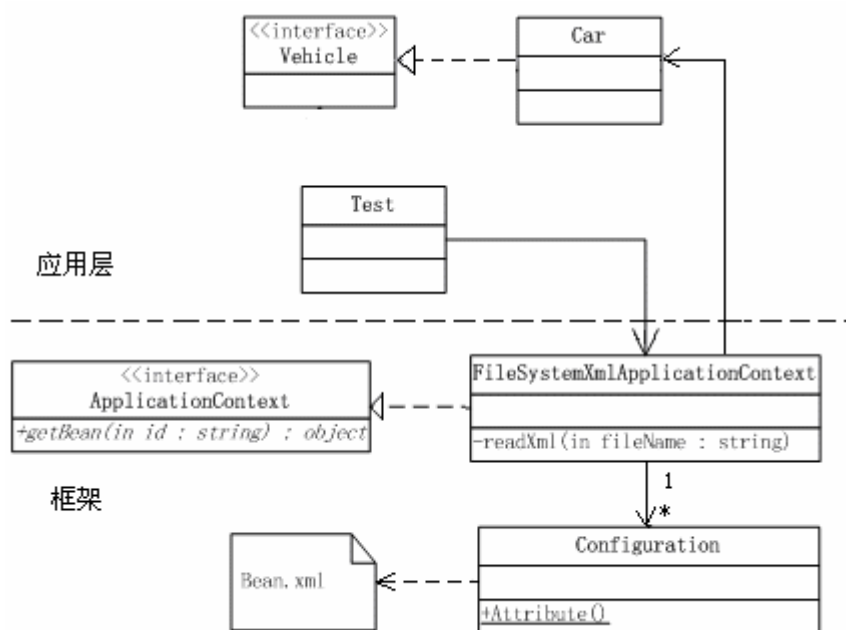
问题：

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：



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(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
node1=((Element)elemList1.item(j)).getElementsByTagName("value");
            String value=node1.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("d:\\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("d:\\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"));
        springdemo.Vehicle s2=(springdemo.Vehicle)m.getBean("Train");
        System.out.println(s2.execute("你的"));
    }
}
```

我们发现，在加入新的类的时候，使用方法几乎不变。通过上面的讨论，我们当对框架实现技术有了一个基本的理解。

7.7 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：

```
class CShapeSingleton{
    private static CShapeSingleton mySingleton=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 CShapeSingleton()
    {}
    //这个方法是调用的入口
    public static CShapeSingleton ShapeInstance(){
        if (mySingleton==null){
            mySingleton=new CShapeSingleton();
        }
        return mySingleton;
    }
    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;
    }
}
```

//调用代码：

```
public class MySingleton{
    public static void main (String[] args){
        CShapeSingleton m1;
        CShapeSingleton m2;
        CShapeSingleton m3;
        CShapeSingleton m4;
        m1=CShapeSingleton.ShapeInstance();
        m2=CShapeSingleton.ShapeInstance();
        m3=CShapeSingleton.ShapeInstance();
        m4=CShapeSingleton.ShapeInstance();
        System.out.println(m1.Count());
        System.out.println(m2.Count());
        System.out.println(m3.Count());
        System.out.println(m4.Count());
    }
}
```

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。

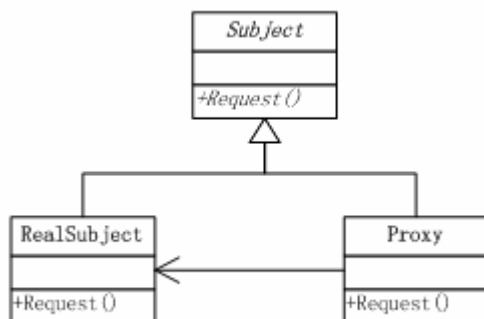
因此何时使用单件，要具体情况具体分析，不能一概而论。

7.8 代理模式的应用

一、代理模式简述

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。

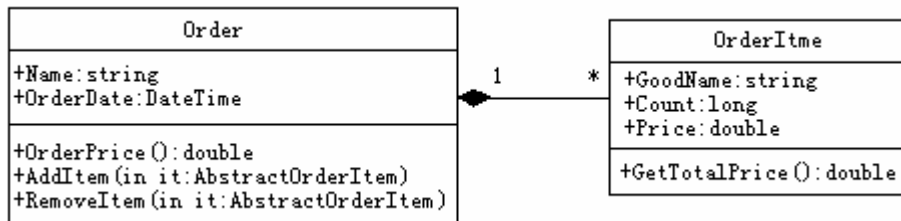


二、案例：在团队并行开发中使用代理模式

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。

假定在我们设计的订单处理子系统中，**Ordre**（订单）类在计算订购项目金额总合的时候，需要由客户服务子系统根据客户级别和销售策略来计算实际收取的费用，而客户服务系统由**OrderItme**（订单项）类提供这个服务，显然这是由另一个开发组完成。



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

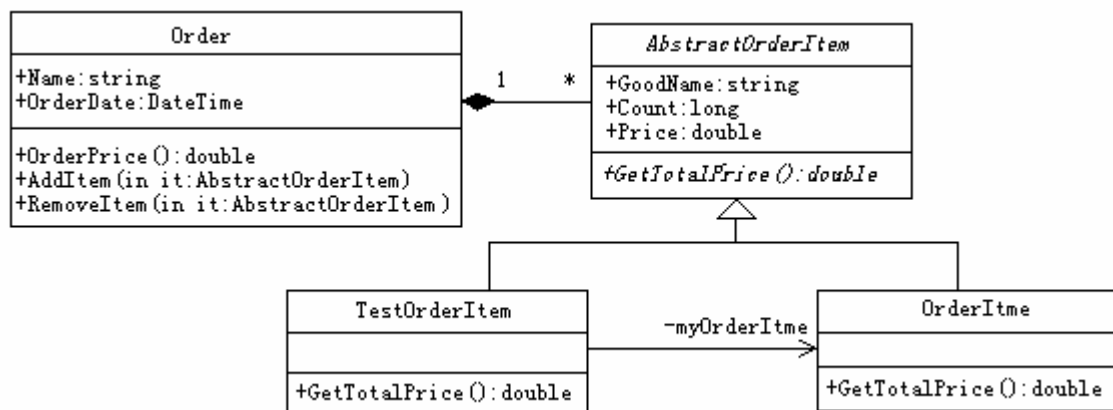
在两个开发组共同完成这个项目的情况下，如果 OrderItem 没有完成，Order 也就没有办法测试。一个简单的办法，是 Order 开发的时候屏蔽 OrderItem 调用，但这样代码完成的时候需要做大量的垃圾清理工作，显然这是不合适的，我们的问题是，如何把测试代码和实际代码分开，这样更便于测试，而且可以很好的集成。

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：

在配置文件 Bean.xml 中加一个元素：

```

<appSettings>
    <add key="istest" value="false" />
</appSettings>
  
```

编写抽象的定单类：

```

package demo;
//这是统一的接口
  
```

```
public abstract class AbstractOrderItem{
    private String goodName;
    private long count;
    private double price;
    public void setGoodName(String m_GoodName){
        goodName=m_GoodName;
    }
    public String getGoodName(){
        return goodName;
    }
    public void setCount(long m_Count){
        count=m_Count;
    }
    public long getCount(){
        return count;
    }
    public void setPrice(double m_Price){
        price=m_Price;
    }
    public double getPrice(){
        return price;
    }
    //价格求和，这个计算方式是另外的人编写的
    public abstract double GetTotalPrice() throws Exception;
}
```

编写订单代码：

```
package demo;
//处理订单代码
public class Order{
    public String Name;
    //public DateTime OrderDate;
    private java.util.ArrayList oitems;
    public Order(){
        oitems=new java.util.ArrayList();
    }
    public void AddItem(AbstractOrderItem it){
        oitems.add(it);
    }
    public void RemoveItem(AbstractOrderItem it){
        oitems.remove(it);
    }
    public double OrderPrice() throws Exception{
```

```

        AbstractOrderItem it;
        double op=0;
        for (int i=0;i<oitems.size();i++){
            it=(AbstractOrderItem)oitems.get(i);
            op+=it.GetTotalPrice();
        }
        return op;
    }
    public java.util.ArrayList GetOrder(){
        return oitems;
    }
}

```

假定为另一开发组编写的定单处理代码：

```

package demo;
//由另外的人编写的处理代码
//这里只处理合计
public class OrderItme extends AbstractOrderItem{
    public double GetTotalPrice(){
        return this.getCount()*this.getPrice();
    }
}

```

编写测试代理：

```

package demo;
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.*;

//这是一个代理类，由配置文件决定状态
public class TestOrderItem extends AbstractOrderItem{
    private OrderItme myOrderItme=null;
    public double GetTotalPrice() throws Exception{
        //读配置文件，看是不是处于测试状态
        String istest=Configuration.Attribute("d:\\Bean.xml","appSettings","add");
        if (istest.equals("true")){
            //这个返回的数据称之为"占位"
            return 1000;
        }
    }
}

```

```

        else{
            myOrderItme=new OrderItme();
            myOrderItme.setGoodName(this.getGoodName());
            myOrderItme.setCount(this.getCount());
            myOrderItme.setPrice(this.getPrice());
            return myOrderItme.GetTotalPrice();
        }
    }
}

//这是一个专门用于读配置文件的类
class Configuration{
    public static String Attribute(String configname,
        String mostlyelem,
        String childmostlyelem) throws Exception{
        //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);
        //获取二级标记子节点
        Element node=(Element)elemList.item(0);
        NodeList elemList1 =node.getElementsByTagName(childmostlyelem);
        //获取这个节点的属性集合
        NamedNodeMap ac = elemList1.item(0).getAttributes();
        String strOut=ac.getNamedItem("value").getNodeValue();
        return strOut;
    }
}

```

应用代码:

```

package demo;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Test{
    public static void main (String[] args) throws Exception{
        Frame1 mf=new Frame1();
        mf.show();
    }
}

```

```
}  
}
```

```
class Frame1 extends JFrame implements java.awt.event.ActionListener{
```

```
    JPanel contentPane;
```

```
    //声明 DefaultListModel 类,并使用该类访问列表的数据
```

```
    DefaultListModel listData=new DefaultListModel();
```

```
    JList jList1 = new JList(listData);
```

```
    JTextField jTextField1 = new JTextField();
```

```
    JTextField jTextField2 = new JTextField();
```

```
    JTextField jTextField3 = new JTextField();
```

```
    JButton jButton1 = new JButton();
```

```
    JButton jButton2 = new JButton();
```

```
    Order o=new Order();
```

```
    //Construct the frame
```

```
    public Frame1() {
```

```
        enableEvents(AWTEvent.WINDOW_EVENT_MASK);
```

```
        try {
```

```
            jInit();
```

```
        }
```

```
        catch(Exception e) {
```

```
            e.printStackTrace();
```

```
        }
```

```
    }
```

```
    //Component initialization
```

```
    private void jInit() throws Exception {
```

```
        contentPane = (JPanel) this.getContentPane();
```

```
        contentPane.setLayout(null);
```

```
        this.setSize(new Dimension(337, 240));
```

```
        this.setTitle("测试实例");
```

```
        jList1.setBounds(new Rectangle(13, 16, 187, 171));
```

```
        jTextField1.setText("名称");
```

```
        jTextField1.setBounds(new Rectangle(212, 53, 102, 21));
```

```
        jTextField2.setText("0");
```

```
        jTextField2.setBounds(new Rectangle(212, 89, 103, 24));
```

```
        jTextField3.setText("0");
```

```
        jTextField3.setBounds(new Rectangle(212, 123, 105, 21));
```

```
        jButton1.setBounds(new Rectangle(236, 17, 77, 22));
```

```
        jButton1.setText("加入");
```

```
        jButton1.addActionListener(this);
```

```
        jButton2.setBounds(new Rectangle(238, 161, 77, 23));
```

```
        jButton2.setText("显示");
```

```
        jButton2.addActionListener(this);
```

```
        contentPane.add(jList1, null);
```



```
contentPane.add(jTextField3, null);
contentPane.add(jTextField2, null);
contentPane.add(jTextField1, null);
contentPane.add(jButton1, null);
contentPane.add(jButton2, null);
}

//Overridden so we can exit when window is closed
protected void processWindowEvent(WindowEvent e) {
    super.processWindowEvent(e);
    if (e.getID() == WindowEvent.WINDOW_CLOSING) {
        System.exit(0);
    }
}

public void actionPerformed(ActionEvent e) {
    try{
        if (e.getSource().equals(jButton1)){
            TestOrderItem t1=new TestOrderItem();
            t1.setGoodName(jTextField1.getText());
            t1.setCount(Long.parseLong(jTextField2.getText()));
            t1.setPrice(Double.parseDouble(jTextField3.getText()));
            o.AddItem(t1);
        }
        else if (e.getSource().equals(jButton2)){
            listData.clear();
            java.util.ArrayList m=o.GetOrder();
            for (int i=0;i<m.size();i++){
                AbstractOrderItem s=(AbstractOrderItem)m.get(i);
                listData.addElement(s.getGoodName()+" "+String.valueOf(s.getCount())+" "+String.valueOf(s.getPrice()));
            }
            listData.addElement("合计: "+o.OrderPrice());
            System.out.println(o.OrderPrice());
        }
    }
    catch(Exception e1)
    {e1.printStackTrace();}
}
```

7.9 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。

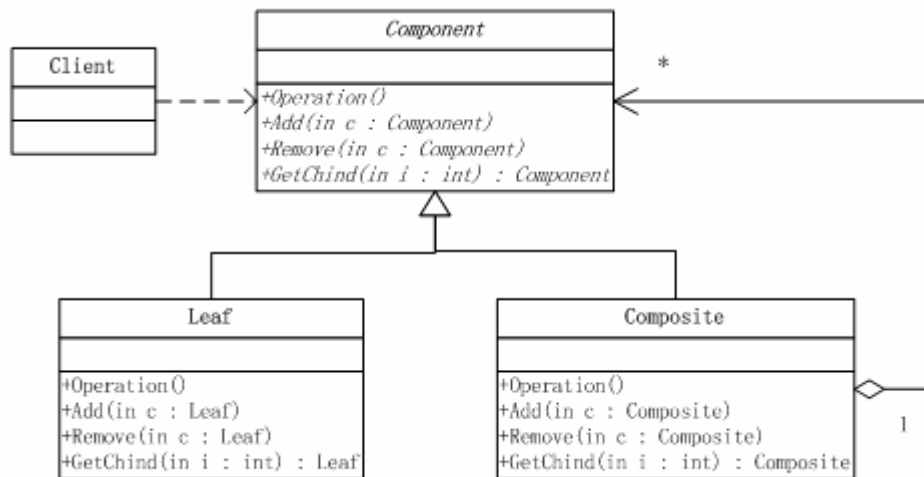
一、树状结构：组合模式

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。

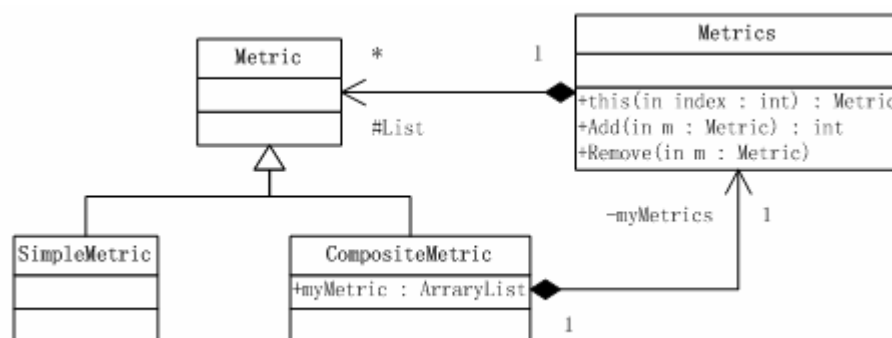
3) 可以很容易的增加类型,由于新类型符合相同的接口,因此不需要改动客户代码。

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



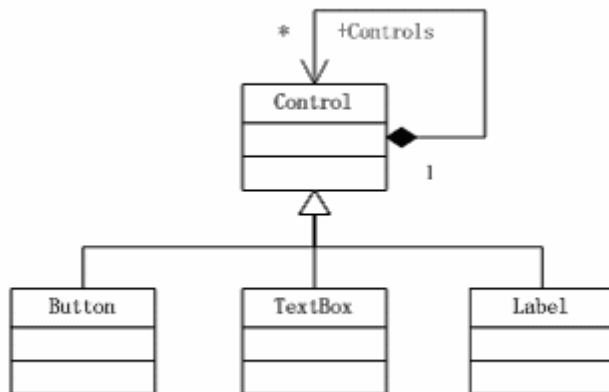
这里有自己定义一个表示控件聚合的 **Metrics** 对象, 由这个对象放置类型(比如 **Metric**), 采用强制类型集合的优点为:

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。

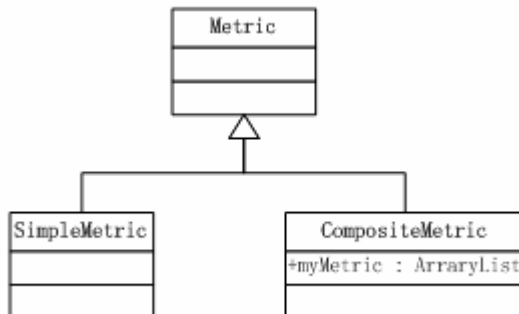
这种方式的缺点是需要自行定制集合, 编码比较复杂。

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:

1) 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。

2) 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。

例:

```
package demo;
public class XMLMetric{
    private String name;
    public String getName(){
        return name;
    }
}
```

```
}
public void setName(String name){
    this.name=name;
}
private String topage;
public String getTopage(){
    return topage;
}
public void setTopage(String topage){
    this.topage=topage;
}
private XMLMetric pmetric;
public XMLMetric getPmetric(){
    return pmetric;
}
public void setPmetric(XMLMetric pmetric){
    this.pmetric=pmetric;
}
private arrXMLMetric metrics;
public arrXMLMetric getMetrics(){
    return metrics;
}
public void setMetrics(arrXMLMetric metrics){
    this.metrics=metrics;
}
public XMLMetric(){
    metrics=new arrXMLMetric();
}

class arrXMLMetric extends java.util.Vector{

    public XMLMetric getXMLMetric(int idx){
        return (XMLMetric)this.get(idx);
    }

    public void setXMLMetric(int idx,XMLMetric m){
        this.add(idx,m);
    }

    public boolean Add(XMLMetric m){
        return this.add(m);
    }

    public void Remove(XMLMetric m){
        this.remove(m);
    }
}
```

```
}
```

测试代码:

```
package demo;
```

```
public class Test {
```

```
    public static void main(String[] args) {
```

```
        XMLMetric1 m1=new XMLMetric1();
```

```
        m1.setName("第一点 0.0");
```

```
        XMLMetric2 m2=new XMLMetric2();
```

```
        m2.setName("第二点 0.1");
```

```
        m1.getMetrics().Add(m2);
```

```
        XMLMetric1 m3=new XMLMetric1();
```

```
        m3.setName("第三点 1.0");
```

```
        m2.getMetrics().Add(m3);
```

```
        XMLMetric2 m4=new XMLMetric2();
```

```
        m4.setName("第四点 0.2");
```

```
        m1.getMetrics().Add(m4);
```

```
        XMLMetric1 m5=new XMLMetric1();
```

```
        m5.setName("第五点 1.1");
```

```
        m2.getMetrics().Add(m5);
```

```
        XMLMetric2 m6=new XMLMetric2();
```

```
        m6.setName("第六点 2.0");
```

```
        m3.getMetrics().Add(m6);
```

```
        Test m=new Test();
```

```
        System.out.println(m1.getName());
```

```
        m.addToTree(m1,0);
```

```
}
```

```
private void addToTree(XMLMetric c,int k){
```

```
    //k 为当前层次数
```

```
    for (int i=0;i<c.getMetrics().size();i++){
```

```
        String s;
```

```
        if (k==0) s="";
```

```
        else if (k==1) s="  ";
```

```
        else if (k==2) s="    ";
```

```
        else if (k==3) s="      ";
```

```
        else s="";
```

```
        System.out.print(s+c.getMetrics().getXMLMetric(i).getName()+" : ");
```

```
        //判断 c 是不是 XMLMetric1 或者 XMLMetric2 的一个实例
```

```
        if (c instanceof XMLMetric1)
```

```
            System.out.println(((XMLMetric1)c).hello());
```

```
        else if (c instanceof XMLMetric2)
            System.out.println(((XMLMetric2)c).hello());
        else
            System.out.println("原始行为");
        //实现递归遍历
        addToTree(c.getMetrics().getXMLMetric(i),k+1);
    }
}

class XMLMetric1 extends XMLMetric{
    public String hello(){
        return "第一种行为";
    }
}

class XMLMetric2 extends XMLMetric{
    public String hello(){
        return "第二种行为";
    }
}
```

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。

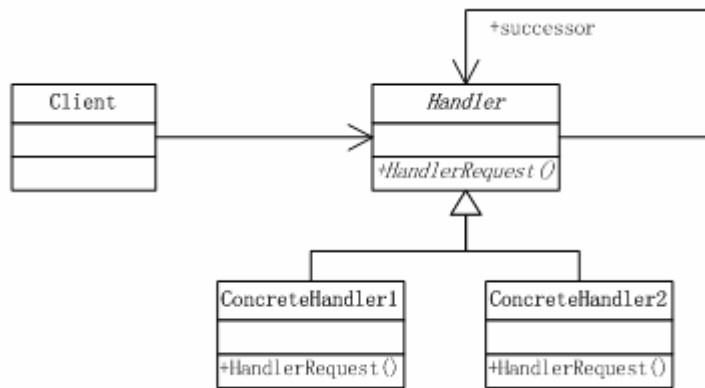
2、使用场合

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;
        ctl=m;
        for (int i=1;i<10;i++){
            ct=new ConcreteHandler1();
            ct.s=i;
            ctl.successor=ct;
            ctl=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);
        }
    }
}
```

我们需要注意，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。

7.10 代码结构的质量度量

一、控制流结构及模型

当我们需要测量程序或者算法的控制流结构的时候。对控制流的建模可以采用一种有向图，称之为**控制流图**（control-flow graph）或者称**流图**（flowgraph）。图中，每个节点，代表一条程序语句，每段弧（或有向边）表示从一条语句到另一条语句的控制流。

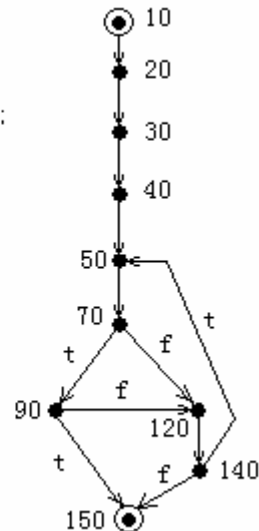
下图表示了一个简单程序 A 及其对应流图 F(A)的合理表示。

所谓合理，指的是人们总是不太清楚如何把程序 A 映射成流图 F(A)。流图是定义控制流结构度量的一个良好模型，因为它对程序的许多结构性质进行了清晰的展示。


```

10  double P=10;
20  double div = 10;
30  double Lim = Math.Abs(Math.Sqrt(P));
40  double Flag = P / div - Math.Abs(P / div);
50  do
60  {
70      if (Flag = 0 || div==Lim)
80      {
90          if (Flag != 0 || P > 4)
100             break;
110         }
120         div += 1;
130     }
140     while (div < 10);
150 //end

```



我们先来讨论一下与图相关的各种术语：

图 (graph)： 有一组点（或结点）和线段（或者边）组成的结构体系。

有向图 (directed graph)： 每个边都指定了方向，用箭头表示。

弧 (arc)： 把有向边称为弧。通常把弧写成序数 $\langle x, y \rangle$ ，其中， x 和 y 是形成弧两端的结点，方向是从 x 到 y 。

入度 (in-degree)： 到达一个节点的弧的数量。

出度 (out-degree)： 离开一个节点的弧的数量。

路径 (path)： 是指前后连续的有向边序列，在这个序列中，经过某些边的次数可能不止一次。

简单路径 (simple path)： 指不会出现重复边的路径。

例如：在上面的例子里，结点 70 的**入度**是 1，**出度**是 2。

流图 (flow graph)： 所谓流图是指这样一种图，即图中有两个结点，**开始结点 (start node)** 和**终止节点 (stop node)**，开始节点入度为零，终止节点出度为零，每个结点都位于从开始节点到终止结点的某条路径上。

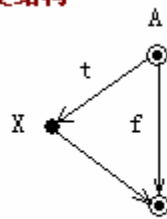
过程节点 (procedure node)： 出度等于 1 的结点成为过程节点。

谓词节点 (predicate node)： 除过程节点之外的所有节点（除终止节点之外）。

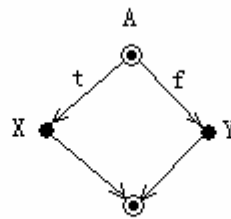
我们为程序结构建模的过程中，有些流图是经常见到的，有必要对它进行特殊的命名。

顺序结构

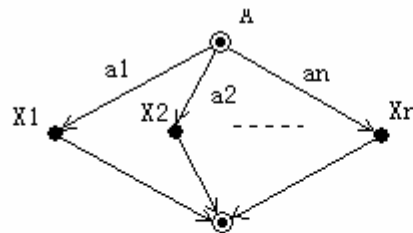


分支结构**D0 或者 D0(A, X)**

if (A) { X }

**D1 或者 D1(A, X, Y)**

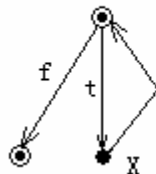
if (A) { X } else { Y }

**Cn 或者 Cn(A, X1, X2, ... Xn)**

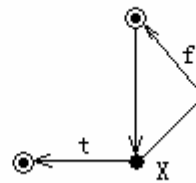
```

switch (A)
{
    case a1:
        X1
        break;
    case a2:
        X2
        break;
    .....
    case an:
        Xn
        break;
}

```

循环结构**D2 或者 D2(A, X)**

while (A) { X }

**D3 或者 D3(A, X)**

```

do { X }
while (A);

```

上述的流图有个很重要的共同性质，就是它可以当作结构化程序的“积木”来使用，为了理解这个问题，必须对构建流图的各种方式进行形式化定义。

1. 循序与嵌套

我们只能根据两种合法操作来对旧的流图构建新的流图，**循序连接**（sequencing）和**嵌套**（nesting），在程序结构方面，这两种操作都有合法的解释。

顺序连接操作：

设 F1 和 F2 是两个流图，则 F1 与 F2 的顺序连接操作，是把 F1 的终止点与 F2 的开始点合并所形成的流图。

记为（可选任何一种方式）：

F1;F2

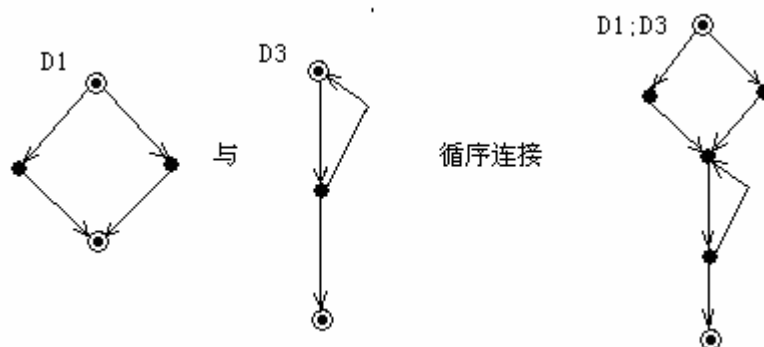
或

Seq(F1,F2)

或

P2(F1,F2)

下面的例子，为流图 D1 和 D3 的顺序连接操作。



嵌套连接操作:

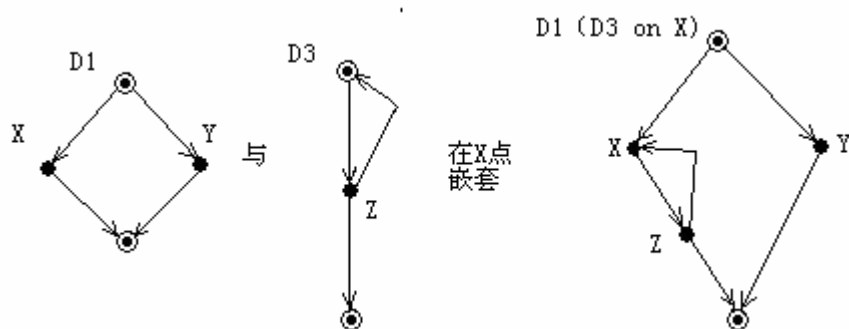
设 $F1$ 与 $F2$ 是两个流图, $F1$ 有一个过程节点 X , 则, $F2$ 在 X 点嵌套于 $F1$ 上, 是指在 $F1$ 的基础上, 通过用整个 $F2$ 替换从 x 开始的弧而形成的流图, 我们把这个结果记为:

$F1(F2 \text{ on } X)$

在确信不会对图中节点产生混淆的情况下, 也可以记为:

$F1(F2)$

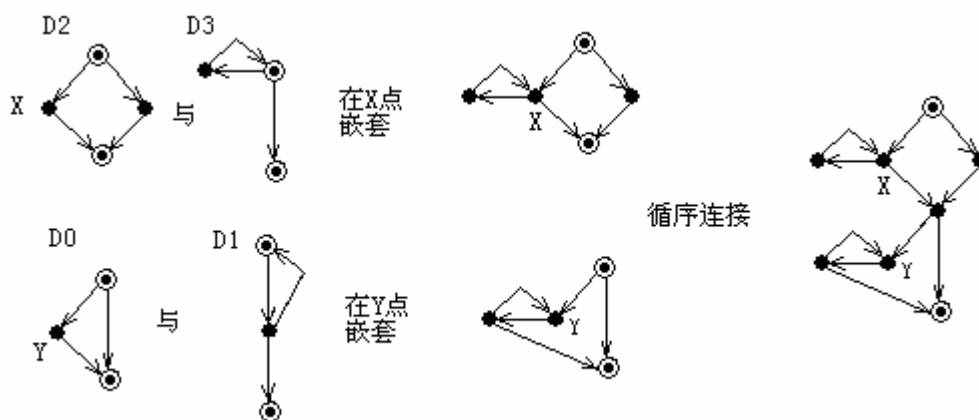
下图显示的是 $D3$ 嵌套于 $D1$ 的情况。



由于本课程的许多例子, 嵌套的实际节点是无关紧要的, 所以一般记为:

$F(F1, F2, \dots, F_n)$

下图显示了流图的构造过程, 这个过程进行了多次嵌套和顺序连接操作。



素流图 (prime flowgraph):

是指不能用顺序连接或者嵌套来进行有用分解的流图。

在上面的例子中，合并前的图都是素流图，而合并后的图都不是素流图，因为可以分解。

2, 结构化的一般概念

结构化程序设计的定义：如果程序的结构只是由三种情况组成，即**顺序**（sequence）、**选择**（selection）和**迭代**（iteration），则称这个程序是结构化的。这个原理可以追溯到 Böhm 和 Jacopini 于 1966 年提出来的经典理论。这个理论表明，任何算法都可以用三种结构来实现。

后来有些人士曾经不太恰当的认为，这个结论无异是在说，只要程序中不出现 goto 语句，这个程序就是结构化的，今天的事实证明这并不恰当，因为无论面向过程还是面向对象的语言，goto 语句还是作为一个重要的成员存在，在构造一些特殊的结构的时候，还在发挥作用。尽管一般来说，goto 语句还是少用为好。

我们希望对某个独立的程序评估，以确定它是否结构化的，最原始的定义对解决这个问题帮助并不大，我们需要一个能够确定任意流图的结构化等级的机制。

为此，我们需要引入更多的、与素流图的图族 S（结构化）相关的术语。

如果图族满足下面的递归规则，则可以说这个图族是 **S 结构化的**（S-structured），或者更简单的说，图族中的所有**元素**都是 **S 图**（S-graph）。

- 1, 所有的元素都是 S 结构化的。
- 2, 如果 F 和 F' 都是 S 结构化的流图，则下列各项亦然：
 - a) F;F'
 - b) F(F') 只要对 F' 嵌套于 F 进行了定义。

- 3, 只有通过对上述步骤有限次的应用生成的图才是 S 结构化的。

前面我们关于基本结构化流图的说明，已经定义了几个元素符号 Pn（顺序结构）、D（if-then 分支）、D1（if-then-else 分支）、Cn（case 分支）、D2（while 循环）、D3（repeat 循环），这些元素称之为**基本 S 图**（basic S-graph），我们可以自由的使用者写基本单元来构造结构化单元。

在结构化程序设计文献中，定义：

$$S^D = \{P1, D0, D2\}$$

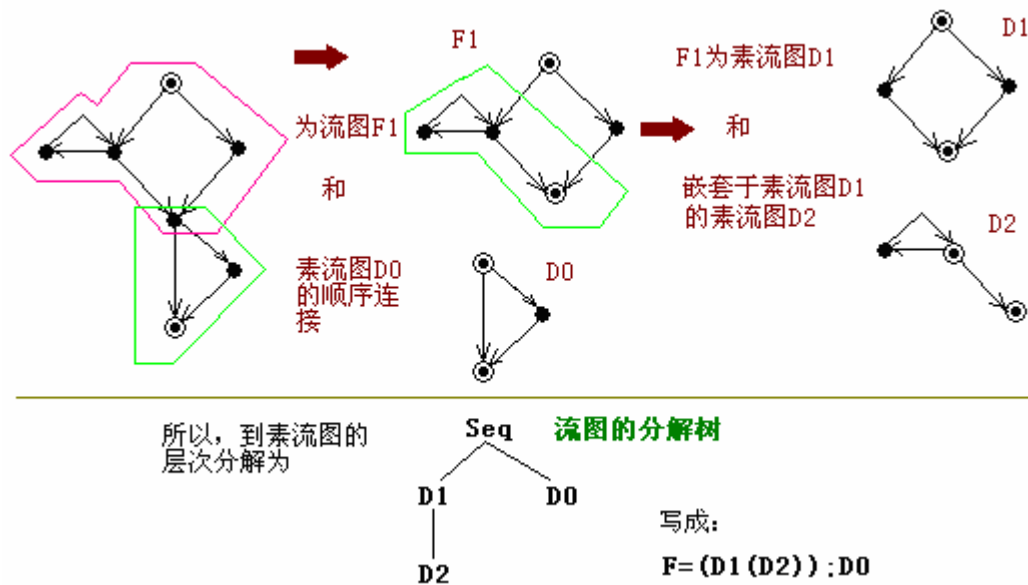
这个定义称之为 D 结构化（D-structured），Böhm 和 Jacopini 的研究曾表明，所有的算法都可以编码为 S^D 图，不过现在还是把比 D 结构化更大的集合作为 S 结构化的集合，也就是 S 结构化单元是集合：

$$S = \{P1, P2, \dots, D1, D2, \dots, C1, C2, \dots\}$$

3, 素分解

Fenton 和 Whitty 的研究表明，我们可以把任意一个流图与**分解树**（decomposition tree）相联系，来描述如何通过素流图进行顺序连接和嵌套操作来构建图的。

下图是根据给定的流图来确定分解树的简单例子。



研究表明，我们不但总是能够把流图分解成素流图，而且还能确信这种分解始终是唯一。这个结论被称之为**素分解定律**（prime decomposition theorem）。

唯一定义的素分解树，是对程序控制结构的一种定义性描述。

当然，对于一个比较大的流图来说，实现素分解并不是容易的事情，手工完成这种计算也是不切实际的，好在许多商业软件能够自动完成这项工作。

素分解的唯一性定律，为我们研究软件结构复杂性度量提供了一个手段，一般来说，程序包含的素流图越多，素流图的嵌套越深，程序结构往往也就越复杂。

二、软件复杂性及度量原则

软件复杂性度量：

开发规模相同、复杂性不同的软件，花费的时间和成本会有很大差异。我们可以从六个方面描述软件复杂性

- ①理解程序的难度；
- ②纠错、维护程序的难度；
- ③向他人解释程序的难度；
- ④按指定方法修改程序的难度；
- ⑤根据设计文件编写程序的工作量；
- ⑥执行程序时需要资源的程度。

它反映了软件的可理解性、模块性、简洁性等属性。

软件复杂性度量的原则：

- 1.软件复杂性与程序大小的关系不是线性的；
- 2.控制结构复杂的程序较复杂；
- 3.数据结构复杂的程序较复杂；
- 4.转向语句使用不当的程序较复杂；
- 5.循环结构比选择结构复杂；选择结构又比顺序结构复杂；
- 6.语句、数据、子程序和模块在程序中的次序对复杂性有影响；
- 7.全程变量、非局部变量较多时，程序较复杂；

- 8.参数按地址调用比按值调用复杂;
- 9.函数副作用比显式参数传递难于理解;
- 10.具有不同作用的变量共用一个名字时较难理解;
- 11.模块间、过程间联系密切的程序比较复杂;
- 12.嵌套越深程序越复杂。

三、层次化度量

我们已经知道,唯一定义的素分解树,是对程序控制结构的一种定义性描述。所以我们就可以对素流图、顺序连接以及嵌套操作所产生的影响进行复杂性度量方面的研究。例:

假定我们需要正式测量“嵌套深度”这个直观的概念。假设我们已经用流图 F 对程序结构建模,想要计算 F 的嵌套深度 α 。

我们可以用素流图、顺序连接和嵌套来表示 α 。

讨论如下:

素流图: 素流图 $P1$ 的嵌套深度等于 0, 其他任何素流图的嵌套深度等于 1。

所以, $\alpha(P1)=0$ 。如果 F 是一个不等于 $P1$ 的素流图, 则 $\alpha(F)=1$ 。

顺序连接: 顺序连接 $F1, \dots, Fn$ 的嵌套深度等于 F_i 中最大嵌套深度值。

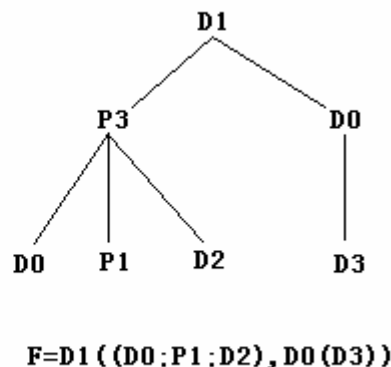
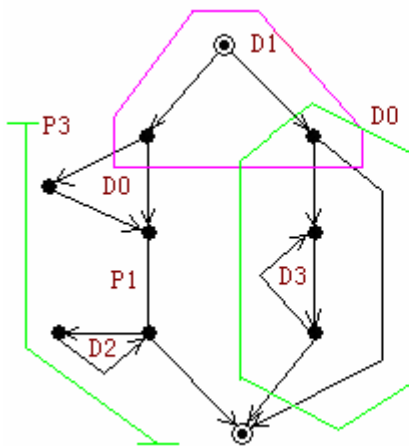
所以:

$$\alpha(F1;F2;\dots;Fn)=\max(\alpha(F1), \alpha(F2), \dots, \alpha(Fn))$$

嵌套: 流图 $F(F1;F2;\dots;Fn)$ 的嵌套深度, 等于 F_i 中的最大嵌套深度值加一, 因为 F 本身就存在一个嵌套层次。所以:

$$\alpha(F(F1;F2;\dots;Fn))=1+\max(\alpha(F1), \alpha(F2), \dots, \alpha(Fn))$$

例如, 我们来考虑如下图所示的结构流程, 和它的结构树。



我们已经知道结构树的表达式为:

$$F=D1((D0;P1,D2),D0(D3))$$

所以可以计算:

$$\begin{aligned} \alpha(F) &= \alpha(D1((D0;P1,D2),D0(D3))) \\ &= 1+\max(\alpha(D0;P1,D2), \alpha(D0(D3))) \quad \{\text{嵌套规则}\} \\ &= 1+\max(\max(\alpha(D0), \alpha(P1), \alpha(D2)), 1+\alpha(D3)) \quad \{\text{顺序连接规则和嵌套规则}\} \\ &= 1+\max(\max(1,0,1),2) \\ &= 3 \end{aligned}$$

直观地看，嵌套深度也是 3。

这种把素流图、顺序规则和嵌套规则分层处理的方式，使我们可以定义一个度量 m ，称之为**层次化度量**（hierarchical measure）。

1，素分解规则：实际上是对要研究问题的定义，把这条规则命名为 M1。

2，顺序连接函数：把这条规则命名为 M2。

3，嵌套函数 h ：把这条规则命名为 M3。

例：我们来研究一下由结构树得到代码行长度 v 的清晰定义。

M1: $v(P1)=1$ ，而且每个 $F \neq P1$ 的素流图（不是顺序结构），都有 $v(F)=n+1$ ，其中 n 是 F 中过程节点的数量。

说明：

过程结点的长度（它通常和没有控制流的语句相对应）等于 1，例如：

```
double Flag = P / div - Math.Abs(P / div);
```

有 n 个控制结点的素流图长度（它通常包含有 n 个控制语句的控制语句）等于 $n+1$ ，例如：

```
if (P<10)
{
    textBox1.Text = P.ToString();
    div += 1;
}
```

在这个例子中，直观地可以数出来： $n=2$ ，总行数为 3。

这和我们实际的映像是一致的。

M2: $v(F1;F2;...;Fn)=\sum v(Fi)$ ，顺序结构的语句数等于每个结构的语句数之和。

M3: $v(F(F1,F2,...Fn))=1+\sum v(Fi)$ ，对于每个 $F \neq P1$ 的素流图。嵌套结构的语句数，等于每个嵌套单元的语句数之和加 1。

后两个表达应该不会引起争议。还是用上面的例子：

$$\begin{aligned} v(F) &= v(D1((D0;P1;D2),D0(D3))) \\ &= 1+(v(D0;P1;D3)+v(D0(D3))) \\ &= 1+(v(D0)+v(P1)+v(D2)+(!+v(D3))) \\ &= 1+(3+1+2)+(1+2) \\ &= 10 \end{aligned}$$

由这个原理，我们可以得到一系列的能够捕获具体特征的、简单但又重要的层次化度量，从某种意义上说，这些度量测量的都是“复杂性”。例如：

结点数度量 n ：

M1: 对于任何一个素流图 F ，都有 $n(F)=F$ 中的结点数。

M2: $n(F1;F2;...;Fn)=\sum n(Fi)-k+1$

M3: 对于任何一个素流图 F ，都有：

$$n(F(F1,F2,...Fn))=n(F)+\sum n(Fi)-2k$$

边数度量 e ：

M1: 对于任何一个素流图 F ，都有 $e(F)=F$ 中的边数。

M2: $e(F1;F2;...;Fn)=\sum e(Fi)$

M3: 对于任何一个素流图 F ，都有：

$$e(F(F1,F2,...Fn))=e(F)+\sum e(Fi)-n$$

“最大素流图”度量 k ：

M1: 对于任何一个素流图 F , 都有 $k(F)=F$ 中的谓词数。

M2: $k(F_1;F_2;....F_n)=\max(k(F_1), k(F_2), \dots k(F_n))$

M3: 对于任何一个素流图 F , 都有:

$$k(F(F_1,F_2,...F_n))=\max(k(F),k(F_1),...k(F_n))$$

还有一些度量就不再列出了。

四、McCabe 圈复杂性度量

不管怎么说, 直接使用素分解进行复杂性度量还是比较困难的, 使用中还是希望找到一种简单直接的方法来解决这个问题, 另外, 人们希望找到一种方法, 把边数、结点数以及素流图数综合在一起考虑, **McCabe** 圈复杂性度量为我们解决这个问题提供了可能。

McCabe 圈复杂性度量是对层次化度量深入研究的结果, 并且提供了一个简单易行的表达式, 下面我们来研究一下这个问题。

我们已经详细的研究了**控制流图** (或称程序控制结构图), 我们知道:

程序结构对应于有一个入口结点和一个出口结点的有向图。

图中每个结点对应一个语句或一个顺序流程的程序代码块。

弧对应于程序中的转移。

由入口结点可以到达图中每个结点, 并且从图中每个结点都可以到达出口结点。

McCabe 用程序控制结构图的圈数 (巡回秩数) $V(G)$ 作为程序结构复杂性的度量

$$V(F) = e - n + 2$$

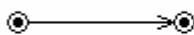
其中: e 为结构图的边数

n 为结构图的结点数

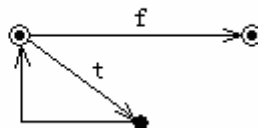
可以证明 $V(G)$ 等于结构图中有界或无界的封闭区域个数

例: 计算如图所示程序控制结构图的 $V(F)$ 值。

1, 顺序结构



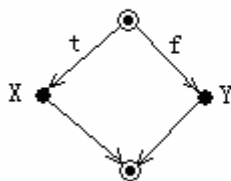
2, while结构



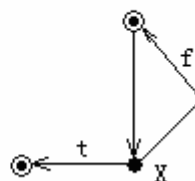
(1) $e=1, n=2, v=1$;

(2) $e=3, n=3, v=2$;

3, 选择结构

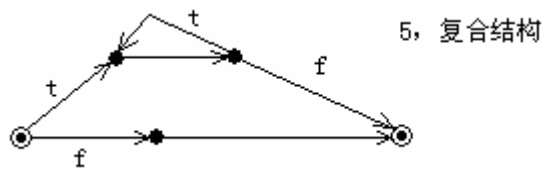


4, repeat结构



(3) $e=4, n=4, v=2$;

(4) $e=3, n=3, v=2$;



(5) $e=6, n=5, v=3$.

McCabe 建议把 $V(F)$ 作为模块规模的定量指标, 一个模块 $V(F)$ 的值不要大于 10 当 $V(F) > 10$ 时, 模块内部结构就会变得复杂, 给编码和测试带来困难。

程序中分枝结构数和循环结构数增加时, 控制结构图的区域数增加, $V(F)$ 的值增大, 程序的结构会变得更复杂。

结构化程序设计控制流力求从高层指向低层, 从低层指向高层的流向, 会增加封闭区域的个数, 反方向的控制流向越多 $V(F)$ 越大, 程序结构越复杂。

McCabe 圈数结构复杂度度量使用起来非常方便, 在实践中也有很好的应用。

从测量理论的角度, 让人十分怀疑的是, 这个假设与复杂性的直观关系是相对应的吗? 我们无法说明这个度量就是复杂性度量。但是, 用圈数确定程序或者模块的测试和维护难度的有用的指示器是非常有效的。在这个背景下, $V(F)$ 可以用于对产品的质量保证。

例: Grady 曾经报告过 HP 公司的一个研究项目, 对 850000 行 FORTRAN 代码的每个模块计算了圈数。研究人员发现, 在模块圈数和需要更新的次数之间存在着紧密联系。在对更新次数超过三次的模块上造成的成本和进度影响进行研究以后, 研究小组得出结论, 模块中允许的最大圈数为 15。

例: Channel Tunnel 铁路系统中的软件质量保证规程要求: 如果 Logiscope 工具发现模块的圈数超过 20, 或者模块的语句数超过 50, 就会认为不合格。

7.12 软件架构挖掘

仔细研究软件架构设计的方法, 我们可以发现, 软件架构设计是一个知识积累的过程, 为了有效的增加知识积累的能力, 我们可以利用一个附加实践, 那就是所谓架构挖掘。

一、架构挖掘过程

1, 自顶向下和自底向上

自顶向下的设计方法, 强调的是把设计视图或者需求文档这样一些抽象的概念, 进一步转化为具体的设计和实现。这事实上是一种预先方法, 在实现之前必须产生设计计划。

而在自底向上的设计方法, 一个新的设计是从基本的程序或有关部分开始创建的, 在递增变化以设计复用方面, 往往更具备生产效率。自底向上方法实际上是一种事后方法, 文档往往是根据已经建立的结构完成的。

一般来说我们推荐自顶向下方法, 它体现了一个系统有计划的开发, 这样更便于构建有效的系统。但是, 不论愿不愿意, 系统当初的设计是不可能一成不变的, 这是一个螺旋上升的迭代过程, 在这个过程中, 架构师会对应用问题有更深入的理解, 不断创建许多新的设计。

自底向上的方法可以认为是这个主体过程的一个补充, 事实上大多数信息系统都存在预先设计, 有些设计存在于已经完成的系统中, 利用这个信息, 架构师就可以在前期建立有效的原型, 对这些信息进行抽取, 把抽取的结果应用于软件架构的过程, 称之为**架构挖掘**。

架构挖掘是利用现存设计与实践经验来创建新的架构,通过回顾大量的实现细节,发现、抽取、提炼设计知识。

2, 架构挖掘过程

架构挖掘开始之前,首先需要识别一组与设计问题相关的代表性技术,这些技术的搜寻可以用各种方法进行(拜访专家、技术研讨会、网上搜索等),也可以由针对性地进行技术预研。

第一步:对典型技术建模,产生相关的软件接口规范。

第二步:已经挖掘的设计被一般化用于创建一个共同的接口规范。

在这一步,我们不是要得到一个最简约的典型设计,而是应该有一个更加良好的解决方案。

第三步:提炼设计,提炼的驱动力:架构师的评定、非正式走查、回顾过程、新的需求、其它挖掘研究等。

二、架构挖掘的方法学问题

1, 挖掘的适用性

挖掘的目的事实上并不是具体产品,事实上挖掘的真正目标是架构师的启迪。这对于降低风险和提高架构的质量是有明显好处的。如果对问题和先前的解决方案能有成熟的理解,架构师就可以着手准备架构设计了。

挖掘的另一个结果是接口模型,尽管这不是正规的产品,但可以应用于架构的创造性设计的方法中。挖掘工作对于高风险的或者有着广泛影响的重大设计,是很有意义的,它可以提高设计的质量和复用性。一般来说,一个典型技术的挖掘研究几天就可以完成,经过几个挖掘研究之后,就可以满怀信心地开始设计了。

架构师的知识主要来自于多年的实践积累,如果我们不注意把这种积累充分挖掘出来,总是针对每个新项目重新开始,往往产生一些不成熟的、习惯性的设计,这就增加了设计的风险。

2, 水平与垂直设计元素

在设计模式的讨论中,我们知道建立一个在未来可以适应变化的系统从技术上是可行的,这种设计被称之为水平设计元素,它的特点是重点考虑软件复用。而针对一个唯一的需求实现硬编码,被称之为垂直设计元素,它的特点是只考虑具体实现。

架构设计的一个重要目标,就是在水平与垂直设计元素的考虑上寻求平衡,这是一个往往让人感到迷惑的问题。作为程序员一般比较偏好垂直设计,因为这可以使用一种直接的方式编码,他们的思维是:既然这样就能解决问题,为什么还要采用其它的方式呢?

但作为构架师,就需要关注具有长期影响的其它设计要点。经验告诉我们,需求的变化是频繁的,而我们也知道了一些设计方法可以灵活的适应这些变化。这样,我们就可以采用某种能够合理管理变化的方法,方法是:

- 1) 列出可能的变化源以及它们对设计的影响,这称之为“架构评估”。
- 2) 研究哪些局部变化会导致全局问题。
- 3) 做出细粒度决策来适应这种变化,这些决策很多来自于直觉。
- 4) 通过实践来平衡对灵活性的要求。

现在我们已经知道,我们可以很容易得把架构设计的很灵活,但合理的架构是基于共识和平衡设计的。过于灵活的设计存在着一些潜在的不良后果:

- 1) 效率低

高度灵活的设计在一个接口两边都需要额外的运行处理,特别是使用了反射这样的技术

尤其如此，动态装入、动态参数很容易在接口操作上出现两个数量级以上的延迟。使用分布式体系或者 SOA 架构都可能造成低效率的后果。如果架构强调了质量标准，这种低效率的不可容忍可能会迫使你放弃灵活性。

2) 可理解性差

如果架构太灵活，往往使开发人员难以理解你的架构，结果造成开发的困境。最坏的情况是开发人员自己做了某些假定，是最后的结果和你的设想大相径庭。所以架构设计中的灵活性应该在开发人员能够理解的前提下实现，而且需要和开发人员加强沟通。

3) 冗余编码

灵活性设计必然导致冗余编码，这样的冗余带来的好处是在变化的过程中不需要修改架构。但是代价是编码量确实提高了，这就提高了开发成本，这是需要做出某种平衡的。

4) 过多的文档化约定

灵活性设计往往需要配置文件，或者用文档来规定约束，这种应用上的复杂性如果不加以控制，这就会使软件集成的难度增加。在硬编码和使用约定之间的权衡，需要一个准确的设计平衡点。

3. 水平设计元素

尽管垂直设计非常受程序人员欢迎而且高效方便，架构师还是应该把眼光落在水平设计元素上。在需求工程完成以后，架构的设计基本上是属于垂直方法，也就是根据需求来构架体系，接着，合理的消除垂直设计元素成为一个架构师能力的体现，关键是合理。

需求经过领域分析以后，软件设计就开始了。领域分析可以在一个给定的问题域里面，帮助架构师确定水平元素和垂直元素。就设计方法而言，由分析人员那里得到的知识和经验尤为重要。

一个好的水平设计通常能满足多个应用要求，不过过分的通用性一般也是很难达到的，这就需要一个平衡。事实上一般化的问题往往比特定问题更容易解决，这是因为特定问题往往把人的思维局限于具体细节中，而一般问题可以从具体的细节中解脱出来，但是复用仍然是对一个架构师水平的考验。

三、职责驱动的开发

在企业应用架构设计过程中，我们要通过特殊的形式讨论职责驱动的设计和开发。职责驱动的开发是指根据子系统或者构件所应该具有的功能上的职责，对其进行分析和设计。

在一个系统中，子系统或者构件的职责集相互正交。如果新设计的子系统要承担的职责已经存在了，就可以把这个职责委派给已经拥有这个职责的子系统、构件或者实例。这种技术以非常小的委派的代价，最大限度地增加了复用。

这个过程的结果之一，就是为子系统创建软件规范。它以独特的格式区别了接口文档和实现文档。因为接口要被其它子系统所使用，所以相较于封装在子系统内的类而言，要求更加不容易变动，这就是一个十分重要的设计原则，接口要保持稳定，高层架构设计的时候，也需要下很大的工夫规范接口。

在软件开发的时候，如果开发人员不能时常碰头，或者极端的某个子系统是由外包的形式开发的，设计文档就更应该类似于设计规范，它比一般的描述方式要求更加严密。设计规范应该更清楚地将子系统、构件间的接口，与详细描述构成系统的子系统内部实现部分加以区分，其目的是尽可能的减少设计的二义性。

尽管如果类设计中方法、参数及数据结构足够详细，本来是不需要设计规范的。但实际上设计是在不断改进，利用软件规范明确标定接口，就可以防止设计改动的时候带来问题。

一旦拥有了设计规范，架构时就可以和开发团队交流思想，通过“介绍”迅速沟通，必

要的时候也可以单独指导。在这样的“介绍”中所取得的反馈，也可以用来改进自己的架构，甚至可以委托开发小组进行小粒度设计和实验，当然这些设计和实验必须符合设计规范，而且是在构架师指导下进行的。

四、架构的可追踪性

一般来说，垂直设计的可追踪性是比较好的，因为它的实现细节与外部需求是紧密联系在一起。但是当架构被设计成水平结构的时候，可追踪性就变得不明显了。解决的办法是通过内部场景显示内部设计是如何支持外部需求的，每个场景都对应于一个重要的外部功能的执行。典型情况下一个水平设计元素与多个外部场景有关，而不仅仅描述单个需求。

第八章 软件经济学与架构设计

20 世纪 60 到 70 年代的软件方法，是典型的手工工艺时代，到 20 世纪 80-90 年代，软件产业成熟了并且变得更具工程规模，但这个时候的软件还是研究密集型的，由规模不经济所统治，2000 年以后，软件业正在迅速的向规模经济为主流的产品密集型的方法发展。这是在这样的背景下，软件分析师与架构师就必须研究软件经济学及其有关的问题。

8.1 软件经济学

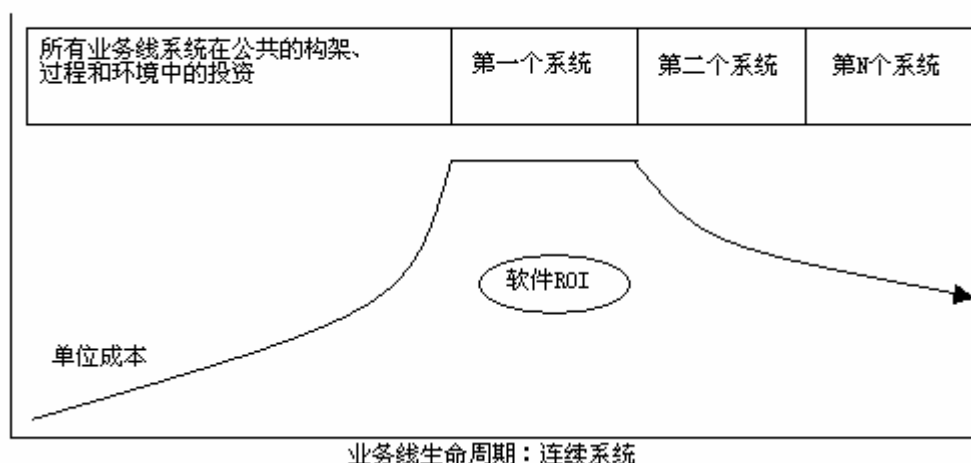
多数软件成本模型被抽象成五个基本参数的函数：规模、过程、人员、环境和所要求的质量。

- 1，最终产品的规模，通常以源代码行数或者功能点数来量化。
 - 2，用于生产最终产品的过程，特别是避免无附加值活动的过程的能力（返工、官僚主义的拖延、沟通开销）。
 - 3，软件工程人员的能力，尤其是项目应用领域的经验。
 - 4，环境，有效地软件开发和使过程自动化的工具或技术。
 - 5，所要求的产品质量，包括产品的特性、性能、可靠性和适应性。
- 这些参数之间的关系和所估计的成本可表达为：

$$\text{工作量} = \text{人员} \times \text{环境} \times \text{质量} \times \text{规模}^{\text{过程}}$$

如果过程指数大于 1.0，则软件就是规模不经济的（随着规模的增加，工作量大幅度增加）。系统的规模效益主要表现在项目的后续期，当架构设计合理的时候，随着后续系统的开发，软件系统的投资回报则上升。下面的图表达了各种领域生命周期的后续工作中，如何达到投资回报（ROI）的情况。

在一个业务线中实现ROI



8.2 软件规模预估

当需求分析完成以后，特别是当功能确定以后，作为管理方需要的信息，第一个遇到的问题就是如何预估项目的规模。目前很多项目是凭经验来估算的，这种经验估算事实上具有

很大的主观性。比较科学的项目估算方法：一种是经典的**代码行规模度量**，另一种是现在可能更流行也更实用的**功能点规模度量**。但是，规模度量更多的是一种预估，它是以历史数据作为基准的，如果在过去开发中历史数据保留不全面，是不可能进行正确的规模预估的。

一、代码行规模度量

1、传统的代码度量

最常用的源代码程序长度的度量是代码行（LOC）的数量。但里面有些问题，关键是要采用何种策略来处理：一般认为，不应该计算空白行和注释行。惠普（Hewlett-Packard, HP）公司定义，代码行是不带注释行和空白行的任何源语句（包括程序头、声明和任何可执行和不可执行的语句）。惠普公司的定义得到了广泛的认可，按照这个定义，我们把不带注释的行缩写为 NCLOC，又称其为有效代码行（ELOC）。

另外，很可能给人造成误解，似乎编写代码不需要关注注释行，这当然是错误的。因此，我们建议采用一种折中的方法，定义注释行为 CLOC，则总长度为：

$$\text{LOC} = \text{NCLOC} + \text{CLOC}$$

这个定义还可以派生出一个有用的度量：

$$\text{CLOC/LOC}$$

表达的是注释行的密度，或者是程序对自己说明的程度。

从实现长度测量的自动化方面，LOC 肯定要比 NCLOC 的难度低，所以，把 LOC 和 NCLOC 两个度量都收集起来是很有用处的。

2、处理非文本和外部代码

另一方面，语言依赖性可能会成为问题，大多数代码行估算都作了这样的假定，也就是软件代码完全是由纯文本组成的，这在 1990 年以前是正确的，但是，现在可视化编程是如此普及，编程环境使很多问题得以简化，单纯的依靠代码行是没有办法很好的估计工作量。

例如：C#编程环境可以在没有写多少代码的情况下，很快的构造复杂的界面，只是在具体操作的时候才需要编写代码，甚至在没有写一程序的情况下，轻而易举的生成 200k 左右的代码，这就带来了两个测量问题：

- 1) 如何计算非文本对象的长度度量？
- 2) 如何计算在外部构造的组件长度度量？

面向对象的开发技术，还为测量长度提供了新的方法。

与计算代码行比较起来，统计对象和方法的个数能够获得更精确的生产率估计值。有人做过“用对象和方法来计算规模度量”的类似报道。例如，Lorenz 在 IBM 做研究的时候发现，平均起来，每个类含有 20 个方法，C++ 平均的方法规模为 24 行。一个类平均包含 6 个对象属性或者实例变量。

他同时还注意到，规模还会随着系统地应用类型的不同而发生变化，例如，原型类含有 10 到 15 个方法，每个方法含有 5 到 10 行代码。而生产类则含有 20 到 30 个方法，每个方法含有 10 到 20 行代码。

工作于美国马里兰州巴尔的摩城“太空望远镜科学研究所，STSI”的 Williams 曾对面向对象软件规模做过研究（这项研究只是整个研究“识别功能性和生产率测量中的关键因素”的一部分），他建议用类的个数、功能点数和类的交互数，来对规模进行表述。

用软件代码行数估算软件规模简单易行，但它的缺点也很明显：

- 1) 代码行数的估算依赖于程序设计语言的功能和表达能力；
- 2) 采用代码行估算方法会对设计精巧的软件项目产生不利的影响；

- 3) 在软件项目开发前或开发初期估算它的代码行数十分困难;
- 4) 代码行估算只适用于过程式程序设计语言, 对非过程式的程序设计语言不太适用等等。

3. 预测长度

长度, 尤其是源代码的长度, 是许多预测模型需要的而且做深入研究的属性。

从直观的角度, 把生命周期后期产品的长度与前期产品的长度进行关联来对长度进行预测, 具有很大的吸引力。尤其需要注意, 通过计算相似项目从规格说明 (或者设计) 长度, 到代码长度的**膨胀率** (expansion ratio) 的中位数, 可以从一定意义上预测长度。

例: 我们可以用设计规模来预测结果代码的规模, 设计代码的膨胀率为:

膨胀率=设计规模 / 代码规模

在模块设计阶段, 可以用下面的公式来估计长度 (用 LOC 表示):

$$LOC = \alpha \sum_{i=1}^m S_i$$

其中:

S_i : 模块 i 的设计规模。

m : 模块数量。

α : 采用同样习惯的, 历史项目记录下来的模块膨胀率。

为生命周期内的所有文档准备一套良好而稳定的长度度量, 其好处不言而喻。规格说明的和设计的长度, 最通用的计算方式是“页”, 也有把其中的组成元素分开计量, 比如把规格说明分为三个视图、四类图表和六个原子对象。

视图	图表	原子对象
功能视图	数据流图	圆圈
	数据字典	数据元素
数据视图	实体关系图	对象、关系
状态视图	状态转换图	状态、转换

不过这种计算设计规模的方式, 还没有什么统一的方法, 设计思想和方法本身也还在发展变化之中, 所以, 必须自己对规模进行定义, 重要的是定义必须统一。我们可以明确生命周期内不同阶段的长度度量之间的关系, 并进行更准确地表示, 从而改进自己的预测工作。

三、功能点度量

很多软件工程师都认为, 长度是一种误导, 代码行度量依赖于程序设计语言, 尤其是在面向对象方法普及的今天, 代码行更没有办法真实的表达系统的规模。所以, 他们认为, 产品内在的**功能点**数量, 能更好的描述产品的规模状况。

由 Albrecht 提出的功能点分析法 (Function Point Analysis, FPA) 是一个很好的方法。功能点度量的意图在于度量值和软件的代码量无关。另外, 功能点还倾向于应用于软件的整个生命周期, 从需求分析一直到以后的升级维护。目前功能点很多国家广泛流传, 原因在于它赋予了软件行业解决严肃的经济学问题的能力。21 世纪初, 功能点度量已经成为研究很多重要软件课题的标准, 它包括:

- 生产力基线和基准;
- 质量基线和基准;
- 过程改进经济学;
- 合同外包等。

经验证明,应用功能点分析来度量软件的规模是非常可靠的,尤其是在项目估计、变更管理、生产率度量和功能需求的沟通等方面。更重要的是,既然功能点是从用户的角度按功能来表达开发的产品,所以对于实时的嵌入式编码和面向对象的编程方法都可以很好的应用。

我们知道,理解客户需求最好的办法是站在客户的角度分析软件系统产生的结果,从而来确定客户关心的问题。因此,功能点分析的一个主要的目标就是从用户的角度定义系统的能力,让项目组和客户可以使用同一方法定义功能需求。

Albrecht 的工作量估计方法,主要建立在功能点概念的基础之上的。正如这个概念所暗示的,功能点(function point)主要用来测量系统中(由规格说明描述的)功能的数量。计算功能点的时候,并不要求规格说明一定要与特定的规格说明模型或者技术的规定相吻合。

功能点分析法(Function Point Analysis, FPA)中认为整个项目的规模由三部份构成:信息处理规模,技术复杂度,环境因素。

1, 信息处理规模

前面已经说到,从用户的观点来看,系统是从五个基本方面帮助他们进行工作的,这就是功能点的五个要素。在计算功能点(FP)的时候,首先要计算未调整的功能点数(UFC),为此,需要根据软件的表示方式确定下列各项的数目:

- 数据处理功能

1) 内部逻辑文件:系统内的逻辑主文件。

2) 外部逻辑文件:到其他系统的可机读的接口。

- 事务处理功能

1) 外部输入:由用户提供的项,主要用来描述各种面向应用程序的数据(比如文件名和菜单选项)。这些项不包括查询,因为要对查询进行单独的计算。

2) 外部输出:向用户提供的项,主要用来生成个中面向应用程序的数据(比如各种报告和消息,而不是单独的部分)。

3) 外部查询:需要做出响应的交互输入。

进一步解释,数据处理规模:

内部逻辑文件(ILF):这是第一项数据功能,使客户可以使用他们负责维护的数据。例如驾驶员在起飞前通过显示器输入一系列航行数据,这些数据保存在一个文件里备用,并可以在执行过程中进行修改。因此,驾驶员负责维护包含航行信息的文件。数据在系统中的逻辑分组是由最终用户维护的,我们把它叫做“内部逻辑文件”。

外部界面文件(EIF):这是第二项数据功能,也和数据的逻辑分组有关。在这种情况下,用户不负责维护数据,数据在另一系统中驻留由其他用户进行维护。该数据只供系统用户参考使用。例如:飞行中,驾驶员可能需要参考某卫星或地面定位系统的定位数据。驾驶员不负责更新这些数据但要参考使用。这样,这些只供参考使用的其他系统的数据分组就称为外部界面文件。

以下的功能的重点是客户端可以处理 ILF 和 EIF 中数据的能力,主要包括:维护、查询、输出数据。这类功能称为事务功能。

外部输入(EI):这是第一项事务功能,是指用户可以根据需要通过增、删、改来维护内部逻辑文件。例如,驾驶员在飞行前和飞行中可以增加、删除、修改航行信息。此时,飞行员使用的功能是外部输入。外部输入使用户可以维护 ILF。

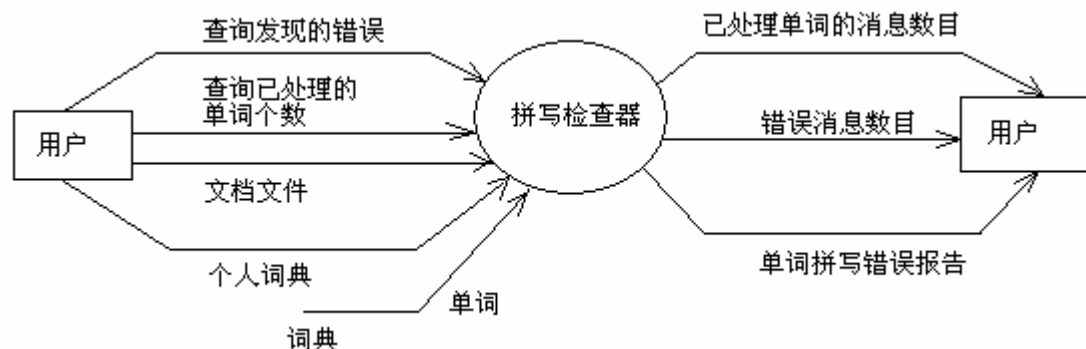
外部输出(E0):下一项事务功能是使用户可以输出结果。例如:驾驶员能够分别显示地面速度、实际气流速度和经过校准的气流速度。显示结果就是经过调用维护数据和参考数据获得的。在功能点术语中,显示的结果就称为“外部输出”。

外部查询 (EQ): 最后一项事务功能是指用户可以通过计算机系统选择特定的数据并显示结果。为了获得这项结果, 用户要输入选择信息抓取符合条件的数据。此时没有对数据的处理, 是直接从所在的文件抓取信息。例如: 驾驶员要显示预先设置的地形图, 输出的结果就是直接从信息存贮位置提取的信息; 这里我们称作“外部查询”。

例: 下图描述的是一个简单的拼写检查器, 请对未调整的功能点数进行计算。

拼写检查器规格说明:

检查器接收的输入是文档文件和可选的个人词典。检查器会列出所有未包含在这些文件中的单词。用户可以在处理的任何阶段查询已处理的单词个数, 以及发现的拼写错误个数。



A, 外部输入个数=2

B, 外部输出个数=3

C, 外部查询个数=2

D, 外部文件个数=2

E, 内部文件个数=1

然后, 根据个人的判断为各项指定“复杂性”等级, 等级的权值如下表:

项	加权系数		
	简单	适中	复杂
EI 外部输入	3	4	6
EO 外部输出	4	5	7
EQ 外部查询	3	4	6
EIF 外部逻辑文件	7	10	15
ILF 内部逻辑文件	5	7	10

理论上一共有 15 个不同中类的项 (五类中都有三个复杂性等级), 先用个类的项数乘以该类的权重, 再对所有的项目求和, 就得到了未调整的功能点数 (UFC):

$$UFC = \sum_{i=1}^{15} ((\text{第}i\text{类项数}) \times \text{权值})$$

例: 以上面的检查器为例, 假定每个项复杂性相同, 那么未调整的功能点的 UFC 值为:

$$UFC = 4A + 5B + 4C + 10D + 10E = 61$$

如果我们知道词典文件和单词拼写错误报告项的等级为“复杂”, 那么:

$$UFC = 4A + (5 \times 2 + 7 \times 1) + 4C + 10D + 10E = 63$$

2. 技术复杂度

技术复杂度指系统实现的复杂度, 按照系统特徵分为 14 个方面: 数据通讯, 分布式数据处理, 性能, 硬件负荷, 事务频度, 联机数据输入, 界面复杂度, 联机更新, 内部处理复

杂度，代码复用考虑，转换和安装，备份和恢复，多平台考虑，易用性。依据每个方面的打分。所以说功能点计算的下一个步骤是，计算**调整功能点数（FP）**，也就是用 **UFC** 乘以**技术复杂性因子 TCF**，也有称作调整系数 VAF(Value Adjustment Factor)。这个因子涉及到下表的十四个起作用的因素：

技术复杂性系数的组成部分	
名称	说明
F1: 可靠的备份与恢复	系统的数据有可靠的备份，出现故障后能够迅速恢复。
F2: 数据通信	应用系统中的数据和控制信息通过通讯设施发送或接收。
F3: 分布式的功能	应用系统在其应用范围内具有分布式数据或处理功能。
F4: 性能	运行客户提出或批准的应用系统的运行目标，可以是：响应速度、处理量、对设计、开发、运行和支持的影响（或可能的影响）。
F5: 大业务量配置	大用户量配置对设计有特殊要求，是应用系统的一个特性。
F6: 在线数据入口	系统中包括在线数据输入和控制信息功能。
F7: 易操作性	应用系统具有易操作性。系统测试阶段，提供了可以有效启动、备份和恢复规程。
F8: 在线更新	应用系统包括在线更新内部逻辑文件的功能。
F9: 最终用户效率	在线功能强调了对用户效率的要求。
F10: 复杂处理	应用系统有进行复杂处理的特点。
F11: 可复用性	应用系统中的应用和代码经过特殊设计、开发和支持，可以在其他应用系统中复用。
F12: 易安装性	应用系统的转换和安装容易。系统测试阶段提供了转换和安装计划和/或转换工具
F13: 多场所	应用系统经特殊设计、开发、支持可以在多个组织、多个地点安装。
F14: 容易变更	应用系统经特殊设计、开发、支持，可以支持变更。

对表中的每一个组成部分从 0 到 5 评级，其中：

0 对系统没有影响；1 影响不明显；2 略有影响；3 中等影响；4 明显影响；5 强烈影响，对正在构造的系统起关键作用。

再运用下面的公式，把 14 个评级结果合并为一个最终的技术复杂性因子：

$$TCF = 0.65 + 0.01 \sum_{i=1}^{14} F_i$$

其中：Fi 的变化范围为 0 到 5。

这个因子的变化范围是 0.65（所有的 Fi 都是 0）到 1.35（所有的 Fi 值都是 5）。功能点的最终计算结果是：

FP=UFC*TCF

这就是总的系统内在规模。

例，还是上面的例子，通过对规格说明的了解，我们做了合理的假定：

F3、F5、F9、F11、F12、F13 值为 0；

F1、F2、F6、F7、F8、F14 值为 3；

F4、F10 值为 5。

则：**TCF=0.65+0.01(18+10)=0.93**

由于 UFC 的值为 63，所以

FP=63*0.93=59

这就计算出了功能点数据。

3，环境因素

环境因素指由外部因素决定的系统开发效率。这个数据和开发采用的工具，是否有过类

似开发经验，项目大小等有关。该值的作用是将功能点转换为时间。对于中小型项目(<3000FP)，对于一个有经验的 Team，开发效率一般为 50 - 100 FP/人月。

考虑到环境因素后，可以得到整个项目所需的时间，即整体项目规模。对于不同的项目，信息处理规模、技术复杂度的计算会有所不同。而开发效率也会随程序员不同而不同。我们可以基于 FPA 标准，根据自身实际情况，经过 1-2 个项目的调整，可以对之后的项目工作量做出比较正确的估计

8.3 成本估计

一、软件成本估计的建议

在软件成本估计的一个关键问题，是缺乏对于迭代式开发方法的案例，尽管很多模型声称他是用于迭代式开发项目，但很少看到使用迭代式开发定义完整的、成功的例子，这就给我们的研究提供了一个空间。

另一个问题是，即使在同一个组织，由于定义不一致，收集相似项目的数据是非常困难的，这就需要我们早期就定义清楚处理方法和理论基础，而且过程需要尽可能统一。我们必须解决好下面这些问题：

- 1，使用哪一种成本估计模型。
- 2，使用代码行还是功能点来度量软件的规模。
- 3，是什么形成了一个好的估计。

这些问题稳定了之后，必须坚持在一个稳定的、比较的长的时间内各个项目都是用相似的方法收集和保存数据，这样才能保证这些数据对后期项目有效。

不论是使用代码行还是功能点来度量软件的规模都是可以的，尽管不少专家认为代码行已经不适用现代软件开发的特征，不过映射成代码的数量（尽管不是真正的行数）给人感觉还是不错的。但是，由于面向对象的技术、构件、源代码自动生成等技术的使用，代码行现在越来越成为一个有歧义的度量。有些不成熟的组织认为不使用软件度量也是可以的，实践证明这个观点是错误的。

一般来说，在项目的初期，用功能点比较合适，在后面阶段，代码行也可能是更合适的度量，当然一个组织坚持使用一种也是可以的。

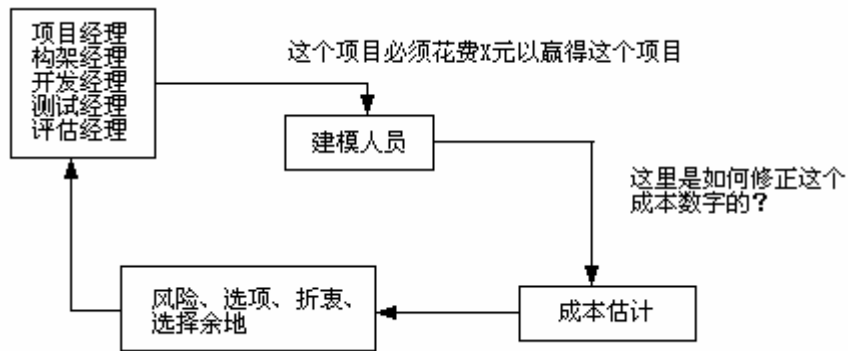
在成本估计之前，有三个问题必须确定下来：

- 1，使用哪种成本估计模型。
- 2，使用代码行还是功能点来度量软件的规模。
- 3，是什么形成了一个好的估计。

不管怎么说，事实上现在还是存在一种观点，就是不用软件度量也是可以的，但事实表明，这种观点是由于能力不够或者组织不成熟造成的，不论是代码行还是功能点来度量软件的规模，都比这个观点好的多。

许多软件专家认为代码行标准是没有用的，但是把代码段描述成若干行代码的规模，感觉上还是不错的。当然，功能点方法更适合于面向对象的系统，特别是信息系统。不管怎么说使用任何一种方法度量总比什么都不用好。

虽然 DeMarco 建议把估计责任分配给一组指定的人员，也就是成立专门的估计团队，但是由一个有能力理解力强的团队进行预估（由项目经理、软件构架师、软件分析师、开发部门经理和测试部门经理组成），也可能是更好的方法，这需要通过几次估计和敏感性分析进行迭代，下面是这样工作的流程。



概括来说，优秀的成本估计具有以下特征：

- 1，它由执行这个项目的经理、构架团队、开发团队和测试团队构思并且支持。
- 2，所有项目相关人员都同意它，并且把它看成一个有雄心但是可以实现的估计。
- 3，它基于一个基础可信的、定义良好的软件成本模型。
- 4，它基于一个相关项目经验的数据库，该数据库包括相似的过程、相似的技术、相似的环境、相似的质量需求和相似的人员。
- 5，它的定义足够细腻，以理解它的相关风险域，并客观的评估它的成功概率。

二、自底向上或自顶向下的估计

从专业的角度，我们知道成本和工作量的估计是非常不准确的，很多组织在项目的开始阶段投入了大量的时间和工作进行预测，但是由于历史记录并不全面，你很难确定估计是不是准确，以及这个过程如何进行剪裁来适应预测，没有这样一些数据，我们就没有进行合理估计的基础。估计技术可以是下面的两种方式：

1) 自底向上估计：

从产品或者任务的最底层开始，提供每一部分的估计，然后把底层的估计值组合成高层的估计值。无论如何，自底向上不是严格的分解技术，它是通过组合几个小的、部分的模型来逐步构造整个模型。或者通过合并较大产品或过程的每个部分，应用类推技术进行类推估计。

2) 自顶向下估计：

从整个过程或者产品开始，做出整体估计后，再计算与整体相关联的组成部分估计，这种方法也可以应用上面四种估计技术。

三、评价估计的准确性

一旦项目开始进行，我们就有机会对实际值和早期所作的估算值进行比较，以判断这种估计技术是不是可以接受。假设 E 是我们的估算值，而 A 是实际值，则计算估算的**相对误差**（relative error）为：

$$RE = (A-E) / A$$

因此，这是一个介于 0 到 1 之间的数，估计值大于实际值，相对误差是负数，而估计值小于估算值，相对误差是正数。通常，我们必须计算许多估计的相对误差，而不是单独进行一次估计。例如，我们通常想知道对一个大的开发项目团队工作量的估计是不是准确，这里，我们定义了 n 个项目的**平均相对误差**（mean relative error）：

$$\overline{RE} = \frac{1}{n} \sum_{i=1}^N RE_i$$

因为相对误差是有符号的，求和中很可能平衡掉一些数据，所有我们还了一定以**相对误差量级**（magnitude of the relative error），数值上为 RE 的绝对值，表示为 MRE。

平均相对误差量级（mean magnitude of the relative error）为：

$$\overline{MRE} = \frac{1}{n} \sum_{i=1}^N MRE_i$$

如果平均相对误差量级很小，则我们的预测一般是正确的，Conte、Dunsmore 和 Shen 建议，平均相对误差量级的可接受水平是 ≤ 0.25 。

他们使用这个概念定义了一个**预测质量**（prediction quality）的度量，具有 n 个项目的集合，k 是平均相对误差量级 $\leq q$ 的项目个数，其中 q 为可接受水平，则：

$$PRED(q) = k / n$$

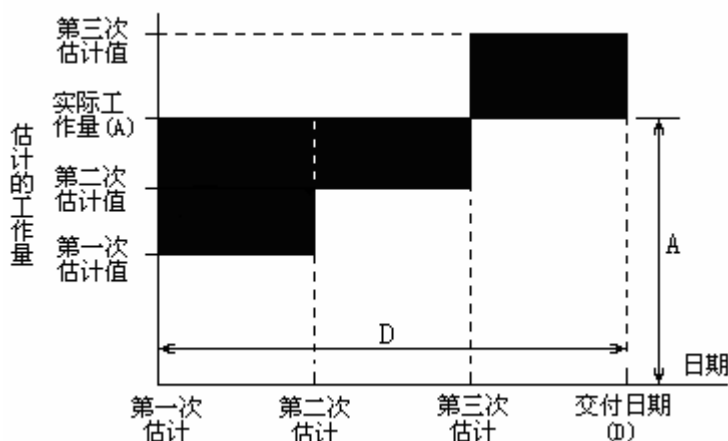
例如， $PRED(0.25)=0.47$ ，表示 47% 的预测值在它的可接受水平 25% 的误差范围内。

Conte、Dunsmore 和 Shen 建议 $PRED(0.25)$ 的数值，如果至少是 75%，则这个估计技术是可以接受的。

DeMarco 建议在评价预测过程准确性方面，使用**估计因子**（estimating quality factor, EQF）。

在他们的技术中，估计在整个过程中被反复计算，因为很多信息是在最终产品中获得的，理想情况下，随着项目的进展，估计值应该能越来越接近实际。

如下图所示的例子，图中阴影部分是估计值和实际值之差。



如果我们将阴影部分的面积，除以 $A \times D$ ，根据估计值对于实际值的覆盖程度，我们就得到估计过程有效性的总度量。

四、工作量和成本模型

分解或者建模方法及上恰当的类推分析方法，往往那个可以比较准确的估计成本。通过建立底层活动或者产品间的工作量关系（如分解法），或者通过构造对工作量或成本其关键作用的参数模型（比如建模），在软件工程师把估计值与实际值相比较的时候，他们就拥有可以用来检验的东西。

也就是说，预测的目标不仅仅是提供一个估计，而是提供一个能够提供准确估计的过程。

通过在过程合成一个模型，估计师检查模型和准确性之间的关系，以便能够调整模型，提高未来预测的准确性。

按照这种观点，分解法也是一种建模技术，它可以获取关系，然后按照分解模型来合成各个属性的和，这也是建模公式最初的形式。

工作量的估计主要是用两种模型：

成本模型：

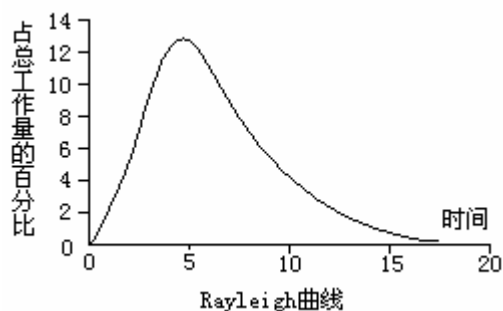
该模型提供工作量和成本的直接估计，大部分成本模型都是基于能够反映整个成本的经验数据。通常这些模型都有一个基本输入（比如产品规模度量），以及大量的二级调整因子，这种二级因子称为成本驱动器，成本驱动器是项目、过程、产品在某种方式上影响工作量和持续时间的资源特性。驱动器的值可以用来调整由基本因子提供的最初估计。

后面要谈到的 CoCoMo 模型就是一个经验成本模型。

约束模型：

显示了随时间流逝两个或者多个参数之间的关系，这些参数是工作量、持续时间或者人员水平等。这种曲线被称之为 Rayleigh 曲线。

典型的 Rayleigh 曲线是 Norden 在检验了 IBM 的很多项目（不一定是软件项目），他发现工作量与过程时间有某种规律性的关系，虽然这是一个经验值，但在很多关于工作量和持续时间的模型中，作为工作量和开发时间关系的基础。



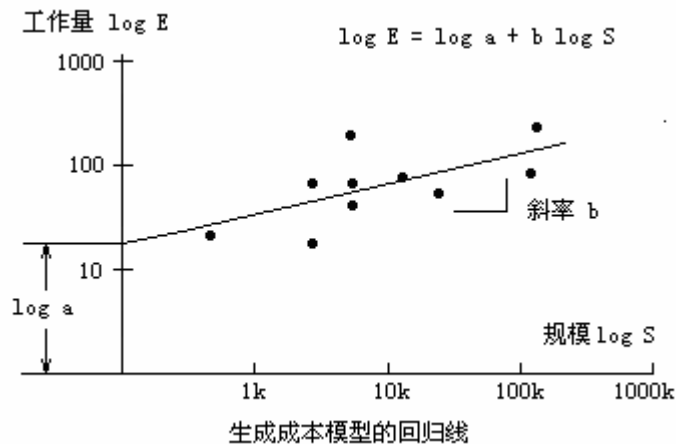
五、基于回归技术的模型

成本模型最基本的构造方法是使用回归技术，这个研究也为我们构造自己的估计模型奠定了基础。通过对大量项目的统计，我们可以以此建立一个基本公式，然后估计就可以通过其它二级成本因子进行调整。

下面我们讨论一下如何根据统计数据，通过线性回归确定基本公式的过程。

假定，我们对大量以代码行为基础的规模度量 S ，和人月为基础的工作量 E 的数据进行了统计，我们发现，把两个变量都取对数以后，有了明显线性的关系。也就是说，在两个轴都取对数以后，有了最小的残差。

线性回归本身的计算方法可以参照最经典的数理统计书籍，由于计算量比较大，在数据比较多时，可以为此编写一个程序计算。假定绘出的图如下。



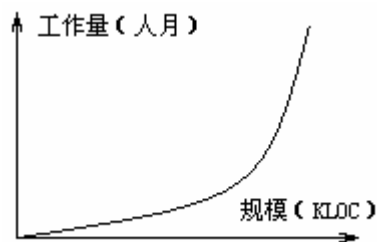
根据这张图，我们可以写出回归直线的表达式：

$$\log E = \log a + b \log S$$

把这个式子由对数域转换到实数域，就变成了指数关系式：

$$E = a S^b$$

上式就是根据回归直线计算出的规模（以代码行记）与工作量（以人月记）之间的数学模型。我们可以推广出来作为后一步的估计公式使用。这个式子表明，规模和工作量之间，具有明显的指数率关系。这也与我们平时观察到的经验规律是吻合的。



当然，线性回归残差是肯定存在的，所以估计的精确度是需要通过其它方面的因素调整的。这就需要积累历史数据以便于比较，以及通过估计实践提升自己的经验。由于通常都存在一个显著的残差，所以基于回归的建模下一步就是明确造成预测值和实际工作量之间差异的因素。

例如：你可能发现，对于相同规模的项目，80%的工作量差异是由于编程小组的经验造成的。这样，你就可以确定一个附加参数，把这个参数添加到模型中成为成本驱动器。在根据它的影响分配加权因子。

例如，你可以根据实际情况，定义一个工作经验（以工作年限记）权值：

低级（小于 8 年）=1.3；

中级（8 到 10 年）=1.0；

高级（大于 10 年）=0.7；

这些权值应该是经验数据，而不是依靠专家的判断。

令 F 是工作经验权值，则工作量计算公式就变成：

$$E = (a S^b) F$$

注意，只有当每个调整因子都是独立的时候，这个公式才是正确的。例：

Bailey 和 Basili 提出一种可以从你自己的数据中推导出成本模型的方法，他们把这项技术应用到美国国家航空航天局戈达德航天中心收集的数据上，然后生成了一个模型，这是文献报道中最准确的模型之一。他们强调，这是一种可以转移到其它开发环境中的技术，而不

是案例模型本身。

Bailey 和 Basili 通过对 18 个相似的大型项目应用，演示了这个方法是如何应用的，从这个项目数据中得到的基本工作量公式为：

$$E = 5.5 + 0.73 S^{1.16}$$

按照预测值和实际值进行调整，根据下表的属性，给出一个技术调整因子。

Bailey 和 Basili 模型技术调整因子		
总的方法论 (METH)	累计复杂性 (CPLX)	累计经验 (EXP)
树图	用户接口复杂性	程序员的资质
自顶向下设计	应用复杂性	程序员对机器的经验
正式文档纪录	程序流复杂性	程序员对语言的经验
首席程序员小组	内部通信复杂性	程序员对应用的经验
正式培训	数据库复杂性	团队经验
设计形式体系	外部通信复杂性	
代码阅读	基于客户需求的程序设计变更	
单元开发文件夹		
正式测试计划		

表中的每一个项都给出评分，从 0（未出现）到 5（非常重要），由项目经理做出判断。然后每个列求和，这些何用最小二乘法回归来拟合公式：

$$\text{调整后的工作量} = a\text{METH} + b\text{CPLX} + c\text{EXP} + d$$

当然，合理的调整因子必须经过多个项目过程，分析预测值和实际值的误差，才能趋向于准确。

六、CoCoMo 模型

在 20 世纪 70 年代，Boehm 开始研究从加利福尼亚 TRW 咨询公司的大量项目中收集的数据。并且研究使用这些数据规律。1981 年 Boehm 根据多年研究，导出了**构造性成本模型**（COnatructive COst MOdel, CoCoMo）。Boehm 是第一个从经济学的角度看待软件工程学的研究者。

1、原始的 CoCoMo 模型

原始的 CoCoMo 实际上是三个模型的集合，分为基本、中间、详细三个层次，分别用于软件开发的三个不同阶段。当对项目了解很少的时候，使用**基本模型**。用以估算整个系统的工作量(包括软件维护)和软件开发所需要的时间。明确需求以后，使用**中级模型**。用于估算各个子系统的工作量和开发时间。当设计完成以后，使用**高级模型**（或者称**详细模型**）。用于估算独立的软部件，如子系统内部的各个模块。这三个模型都是静态、单变量模型，具有相同的形式。

$$E = a S^b F$$

其中：E 是按人月计算的工作量。

S 是按千行交付的原指令数目（KDSI）的测量规模。

F 是调整因子（在基本模型中为 1）。

Boehm 把软件划分为有机式（又称组织型）、半分离式（又称半独立型）和嵌入式三类，允许不同应用领域和复杂程度的软件按照三类软件的适用范围选取相应的参数

组织式系统：包括数据处理，一般要使用数据库，集中在事务处理和数据恢复上，包括银行和账务系统等都是有机式系统。

嵌入式系统：包含一个集成在硬件的大系统中的实时软件，比如导弹制导系统、水压传感系统等。

半分离式系统：介于有机式和嵌入式之间的系统。

三种类型的系统工作量的 a 和 b 的数值见下表。

方式	A	b
组织式	2.4	1.05
半分离式	3.0	1.12
嵌入式	3.6	1.20

通过选用开发方式和使用适当的工作量公式，CoCoMo 生成了一个工作量的初步估计。

例如：为了预测一个主要的电话交换系统的工作量，我们得知系统将需要大约 5000KKDSI。因为软件是实时的，而且是大型复杂硬件系统的一部分，所以它是一个嵌入式系统，初步的工作量估计为：

$$3.6(5000)^{1.2}=100000 \text{ 人月}$$

由工作量可以进一步计算出预测的开发时间，对于开发时间有

$$D = a E^b$$

其中：

D：以月计算的开发时间。

E：以人月计算的工作量。

而 a 和 b 的数值见下表

方式	a	b
组织式	2.5	0.38
半分离式	2.5	0.35
嵌入式	2.5	0.32

例：针对一个 3000 人月的基于数据库的信息系统项目，其开发时间为：

$$2.5(3000)^{0.38}=52 \text{ 月}$$

需要的人数：

$$3000/52 \approx 58$$

也就是说，需要 58 人，工作 52 个月完成。

下面讨论一下**中间 CoCoMo 模型**。中间 CoCoMo 模型可以使用基本模型参数，也可以使用经过调整的参数如下表：

方式	a	B
组织式	3.2	1.05
半分离式	3.0	1.12
嵌入式	2.8	1.20

在工作量估计公式中乘以工作量调节因子 EAF

$$E = a S^b * EAF$$

工作量调节因子 EAF 与**软件产品属性、过程属性、计算机属性、人员属性**有关

原始 COCOMO 的成本驱动器		
软件产品属性	软件可靠性 软件复杂性 数据库的规模	
过程属性	软件开发方法的能力 软件工具的质量和数量 软件开发的进度要求	
资源属性	计算机属性	人员属性
	程序执行时间 程序占用内存的大小 软件开发环境的变化 软件开发环境的响应速度	分析员的能力 程序员的能力 有关应用领域的经验 开发环境的经验 程序设计语言的经验

四种属性共 15 个要素。

每个要素调节因子 F_i , $i=1,2,\dots,15$, 的值分为:

很低、低、正常、高、很高、极高, 共六级。

正常情况下 $F_i=1$ 。

Boehm 推荐的 F_i 值范围

(0.70, 0.85, 1.00, 1.15, 1.30, 1.65)

当 15 个 F_i 的值选定后, EAF 的计算如下

$$EAF = F_1 * F_2 * \dots * F_{15}$$

调节因子集的定义和调节因子定值是由统计结果和经验决定的。不同的软件开发组织, 在不同的历史时期, 随着环境的变化, 这些数据可能改变。

使用中间 CoCoMo 模型可以估算开发软件产品的工作量, 比较各种开发方案的工作量。

工作量、开发时间和项目开发人数计算实例:

假定我们需要估计一个目标代码行数为 33.3 KLOC 的计算机辅助设计 (CAD) 项目。

CAD 软件开发属于中等规模、半独立型

我们可以查到 $a=3.0$, $b=1.12$ 。

代入公式计算:

$$\begin{aligned} E &= 3.0 S^{1.12} \\ &= 3.0(33.3)^{1.12} \\ &= 152 \text{ PM} \end{aligned}$$

将 E 的估算值代入公式来计算开发时间:

取 $a=2.5$ $d=0.35$

$$\begin{aligned} D &= 2.5 E^{0.35} \\ &= 2.5(152)^{0.35} \\ &= 14.5 \text{ M} \end{aligned}$$

建议参加项目开发的人数

$$N = E/D = 152/14.5 \approx 11$$

例中计算出来的 11 人是粗略估计

在软件项目开发过程中, 11 个人不可能都有相同的能力和个性, 相同的经验和知识结构, 并且在软件开发的各个阶段对人的要求也不同。

资源 R 的静态多变量模型:

以静态单变量 CoCoMo 模型为基础, 还可以定义估算资源 R 的静态多变量模型

$$R = \sum a_i * e_i * b_i$$

其中: e_i 表示软件第 i 个特性

a_i, b_i 是与软件第 i 个特性有关的常数, 通常由实验数据确定。

2, CoCoMo 2.0

从 CoCoMo 基本模型开发后的许多年后, 软件工程技术发生了很大的变化, 新的生命周期过程已经实现, 面向对象的开发已经成为主流。这样, CoCoMo 模型对于这些新技术是不灵活和不准确的, 为此, Boehm 定义了 CoCoMo 的升级版, 也就是 CoCoMo 2.0。

CoCoMo 2.0 允许我们使用不同规模的模型, 它的估计过程是基于任何开发项目的三个主要阶段, 原始的 CoCoMo 模型是把交付的源代码行数作为输入, 新模型认为, 在开发周期的早期阶段获得代码行计数是不可能的。

在阶段一, 项目通常通过构造原型来解决具有高风险的问题: 包括用户界面、软件系统的交互、性能和技术成熟度。这里, 由于对所考虑的最终产品可能规模了解很少, 所以, 在 CoCoMo 2.0 中使用对象点来计算规模。其中包括服务器数据表的个数、客户机数据表的个

数、以及重用以前项目的屏幕和报告的百分比。

在阶段二，随着开发工作的进展制定相应决策，但是设计人员必须开发出可以二选一的架构和操作概念。这个阶段，还没有足够的信息来支持细粒度的工作量和持续时间的度量，但由于功能点估计在需求阶段已经获取了功能性，因此，他们提供了比对象点更丰富的系统描述。

在阶段三，系统开发已经开始，我们已经知道了更多的信息，这个阶段符合原始的 CoCoMo 模型，程序规模可以用代码行来计数，很多成本因子也可以用适当的方式来估计。

除了规模，这三个阶段还存在其它的一些差异，下表对他进行了总结，例如 CoCoMo 2.0 合并了重用模型、考虑了维护和损坏、需求变更等等。

成本驱动器与 CoCoMo 相似，但也有了一些新的内容，还有一些重新计算的值，但是由于 CoCoMo 2.0 是新发布的，还没有关于它的准确性的数据报道。

两种版本的 CoCoMo 模型对比				
模型方面	原始 COCOMO	COCOMO 2.0		
		阶段一	阶段二	阶段三
规模	交付的源指令(DSI) 或 源代码行数(SLOC)	对象点	功能点 (FP) 和语言	FP 和语言 或者 SLOC
重用	相当的 SLOC	隐含在模型中	未修改的重用% 修改的重用% (取决于功能)	等于作为其它变量函数的 SLOC
损坏	需求变更性	隐含在模型中	损坏的%	损坏的%
维护	每年的改动(ACT)= 增加%+修改%	对象点	重用模型	重用模型
规模 (b) 在 工作量公式 中	组织式: 1.05 半分离式: 1.12 嵌入式: 1.20	1.0	1.02 到 1.26 取决于: 先例, 一致性。早期 架构, 风险解决方案, 团队内聚力, SEI 成熟度。	1.02 到 1.26 取决于: 先例, 一致性。早期 架构, 风险解决方案, 团队内聚力, SEI 成熟度。
产品成本驱动器	可靠性 数据库大小 产品复杂性	没有	复杂性, 所要求的可 重用性。	可靠性, 数据库大小, 所需文档记录, 产品复杂性。
平台成本驱动器	程序执行时间 程序占用内存大小 软件开发环境变化 软件开发环境响应 速度	没有	平台难度。	执行时间约束, 主存约束, 虚拟机易变性。
人员成本驱动器	分析员的能力 程序员的能力 有关应用领域经验 开发环境的经验 程序设计语言经验	没有	人员能力和经验	分析员能力, 应用 经验, 程序员能力, 虚拟机经验, 编程语言和工具 使用经验, 人员的 连贯性。
过程成本驱动器	软件开发方法能力 软件工具的质量和 数量 软件开发进度要求	没有	所要求的开发进度, 开发环境。	软件工具的使用, 所要求的开发进 度, 多场所开发。

8.4 改进的软件经济学

对软件开发进行经济上的改进是比较困难的，只对某一方面改进效果也是有限的，只有全面地改进软件过程的各个方面，才可能获得明显经济上的好处。下面对此提几个建议：

1, 缩小软件规模

不管怎么说, 基于投资回报 (ROI) 最显著的方法, 是缩小软件的规模, 也就是尽可能的基于构件的开发。但要注意当构件规模减小的时候, 构件之间的通信量就会加大, 而且系统集成的成本就会增加。

2, 复用

复用确实可以提高投资回报率, 问题在于开发可复用的构件成本并不低, 一个能够在不同项目中应用的构件, 其开发成本不能低估, 仅仅开发可复用的构件, 并不一定就能降低成本。因此, 开发可复用的构件, 必须考虑是不是有广泛的客户基础来获得经济利益。

3, 改进软件过程

软件过程的质量会强烈的影响需要的工作量, 也会影响产品的进度。过程的改进至少有 3 个方面:

我们使用一个 N 步的过程并改进每一步的效率。

我们使用一个 N 步的过程, 删去一些步骤, 以至于现在只有 M 步。

我们使用一个 N 步的过程, 并使得要执行的活动和所用的资源尽量保持一致。

很多过程改进都强调的是第一个方面, 但本课程强调的是第二和第三方面, 这里面大有潜力可挖, 尤其是过程改进的目标是以最小的迭代次数获得完备解, 并尽可能消除下游废品和返工。

每一次出现废品和返工都会导致一系列的重做任务, 比如在测试的时候发现了设计缺陷, 重做设计将会导致产品延期和费用增加, 如何压缩这些任务系列呢? 很多过程都是理想情况, 但真要出现这个问题怎么办呢?

我们需要对工程活动进行管理, 以使出现废品和返工的时候不会对任何项目相关人员的取胜条件造成影响, 这应该成为多数过程改进的根本前提。

4, 改进团队的有效性

长期以来, 人们一直认为人员上的差异是生产力上波动的最大原因。但实际上只建立优秀人员组成的团队并不现实。人才的配备主要掌握平衡的原则, 项目经理要让高度熟练的人员处于重要的位置上, 并注意以下箴言:

- 只要项目管理良好, 使用一般的工程团队也能成功。
- 管理失当的项目, 即使使用专家级的团队, 也几乎无法成功。
- 只要系统构架良好, 使用一般的软件团队也能构建。
- 构架差劲的系统, 即使使用专家级的开发团队, 也会难于实现。

这样一来, 就获得了团队组织的一些基本的原则:

- 顶尖人才原则: 使用更好的和更少的人员, 保证团队有合理的规模, 人多和少都是不利的, 而一个队伍中的所谓顶尖人才应该严加限制, 而对于明显不合格的员工, 应该坚决淘汰。
- 工作匹配原则: 把任务分配给技能和动力都匹配的合适人员。在团队中, 有经验的程序员常常希望被提升到设计师或者经理的位置上, 而团队领导似乎也认为这种提升是一种激励, 但两者技能上的需求是不一样的, 结果工作的失败往往对程序员和领导双重打击。这样的例子举不胜举。
- 职业发展原则: 帮助员工自我实现的组织, 最终将获得最好的成绩。表现良好的员工往往总是能在任何环境中自我实现, 组织可以帮助也可以阻止员工的自我实现, 而组织的作用最能帮助中等和中等以下的成员, 这些帮助主要是通过培训来达到。
- 团队平衡的原则: 选择与其他人互相补充的, 协调一致的员工。团队平衡很重要, 只要任何一方失去平衡, 团队就可能处于危险之中。

软件开发是一项集体运动, 必须培养一种团队的合作的, 而不是追求个人成功的氛围,

在上面五项原则中，团队平衡和工作匹配应该是最主要的目标，因为顶尖人才原则和逐步淘汰的原则必须在团队平衡的原则下实施。

8.5 变更管理和预期管理

事实上在开发过程中对成本影响最大的莫过于需求变更，因此如何正确的实现变更管理就是一个值得很好研究的问题，而做好变更管理的前提，是有一个很好的科学的预期管理，对于架构师来说，这个问题值得研究。

一、变更管理

即使计划过程的早期已经就一个正确完成工作的陈述达成了一致，范围增长失去控制的情况仍然十分常见。范围增长称之为“变更”。很多公司都注意到了，变更常常是客户和信息系统开发组织争论的焦点，因为他们不同意一个特定的功能，是对初始协议的改变还是部分改变。

范围变化的不可避免性使我们需要一个正式的策略和过程来处理变更，以及变更对进展和预算的影响。

变更管理（change management）策略是建立一个过程，以管理在项目期间出现的变化，这个过程用来保护项目经理和团队，使其不会由于范围变化导致的进度和预算超支的困扰。

变更可能是各种事件和因素的结果，例如：

- 在定义初始范围的时候得疏忽。
- 对初始范围的错误理解（用于希望的产品比原来交流的更复杂）。
- 产生新需求的外部事件，比如政府规定。
- 组织结构变化，比如合并、收购和伙伴关系，这些变化产生了新的业务问题和机会（参与者也变化了）。
- 可以得到更好的技术。
- 偏离规划技术，这导致了不期望的和明显的企业组织结构、文化或者过程的变化。
- 用户管理层希望系统实现比他们原先要求的或者同意的功能更多的功能。
- 管理层消减了经费，或者提出了更早的最后期限。

变更管理包含一组程序来记录变化请求，并且根据变化的预期影响来定义需要考虑的步骤。大多数变更管理一个或者多个项目相关人员（例如系统拥有者、用户、分析员、设计人员和程序人员）使用一个变更请求表格。理想情况下，这种请求由变更控制委员会（Change Control Board, CCB）来考虑，他们负责批准或者驳回变更请求。CCB 的成员应该包含内部团队人员，也包含于项目利益相关的外部人员，CCB 的决策应该建立在效果分析的基础之上。在更严肃的组织中，CCB 被称作配置控制委员会（Configuration Control Board, CCB），它有一系列重要的职责，其中也包括控制变更，我们会在后面“配置管理”中详细讨论这个问题。

评价业务变化的重要性可以利用可行性效果分析，要考虑变更对进度的影响，以及变更对项目预算和对长期运营费的影响。最后，变更管理中重点在于管理客户的预期，下面我们将介绍一个简单但是概念坚实的框架，用来管理预期以及预期对项目进度和预算的影响。

二、预期管理

有经验的项目管理者都知道，要管理好系统所有者、客户对项目的预期，比管理好项目

的费用、进度、人员或者质量要困难得多。在这一节我们介绍一个称之为**预期管理矩阵**的简单工具，这个局正最初是由麦道公司顾问 Phil Friedlander 博士提出来的。

每个项目都存在目标和约束，在一个理想世界里，管理层通常希望优化费用、进度、范围和质量等每个参数。但现实情况表明，你不可能优化所有参数，这就需要考虑一个对管理层来说可行又可接受的平衡，这就是预期管理矩阵的主要目的，它辅助项目管理人员理解变化的项目参数的动态影响。基本矩阵为下图所示的 3 乘 3 的方阵：

对于优先权来说：

- 最大或最小：对一个给定项目来说，被认为最重要的成功度量。
- 受限的：在一个项目三种成功度量中是第二重要的。
- 可接受的：在一个项目三种成功度量中是最不重要的。

优先权 成功度量	最大或最小	受限的	可接受的
费用			
进度			
范围和/或质量			

理想情况是给三个度量赋予同样的优先权，而且经验表明这三种度量会自动趋于一个平衡，比如你增加了范围或者质量的要求，自然就会花费比较多的时间或者费用。

填表规则是：对于任何项目 9 个单元格只能填 3 个，而且行列都只能有一个×。

我们下面开看一个典型的例子（这是来自于 Phil Friedlander 博士的非常有启发性的例子）。

1961 年美国总统肯尼迪启动了一个 10 年内把人送到月球并安全返回的大型项目。下面看这个项目实际的预期。

1，系统所有者（公众）既有范围预期、也有质量预期。项目范围（或需求）是成功的把人送上月球，项目质量度量是把人安全返回。因为公众对这个新的太空计划的预期不会太少了，它不得不作为第一优先权，也就是把最大安全和最小风险作为第一优先权。

2，项目启动的时候，当时苏联已经走在空间竞赛的前面，为了国家的尊严，第二优先权给予了 10 年内完成任务，这种项目约束没有必要提前，但也不能错过最后的期限。

3，默认情况下，第三优先权赋予了费用（1961 年估计为 200 亿美元）。把费用标成为第三优先权，不是说费用不用控制，只是说在项目范围和质量需求可能不得不接受费用超支。

优先权 成功度量	最大或最小	受限的	可接受的
费用 ● 200 亿美元（估计）			×
进度 ● 1969 年 12 月 31 号（最后期限）		×	
范围和/或质量 ● 将人送上月球 ● 安全地将人返回	×		

历史的情况是，项目在 1969 年完成，花费超过了 300 亿美元，费用超支 50%。费用超支意味着项目失败吗？相反，大多数人认为这个项目是一个巨大的成功。政府成功地管理了公众的预期，即实现了最大的安全和最小的风险，而且满足了最后期限，这对于费用超支来说是一个可接受的权衡。系统开发项目经理可以从这个平衡艺术中上一堂有价值的课。

在任何项目的开始阶段，项目经理应该向系统所有者介绍预期矩阵的概念，并且应该和系统所有者一起填写这个矩阵。对于大多数项目，很难在矩阵中记录下所有的范围和质量需求，相反它们应该在列在工作陈述中，而矩阵中应该列出最后的期限和费用估计。

项目经理并不设置优先权，他只是强制执行矩阵规则。这听起来容易但做起来难，很多项目经理不理解这样做的理由，应该告诉他们，如果不能最大化三个度量，就需要知道他们

的优先权，这有助于我们作出明智的抉择。

如果系统所有者拒绝设置优先权，那么这个工具就没什么用处了，唯一的作用就是问题成为灾难之前记录下你的担心。要知道拒绝设置优先权的系统所有者往往就是拒绝深入思考的所有者，而这样的系统所有者恰恰就是最可能给项目经理提出不可实现的项目目标的管理人员，正如 Friedlander 博士指出来的：“你可以不相信地心引力，但你还是会像其他人一样重重的摔在地上。”

让我们假定你已经有了一个满足前面提到的规则的项目管理矩阵，它将如何帮助你管理预期呢？在一般的系统开发项目过程中，优先权是不稳定的，各种变化因素（比如经济、政府策略和公司策略）都可能改变优先权，比如预算可能变得更受约束或者不那么受约束，最后期限可能变得重要或者不那么重要，更多的情况是，需求增加了或者变化了。这些变化以某种方式影响所有的成功度量，解决这个问题的办法是管理预期，而不论项目参数如何变化。

假定你已经有了如下的优先权矩阵。

优先权 成功度量	最大或最小	受限的	可接受的
费用		×	
进度			×
范围和/或质量	×		

- 1，在项目开始阶段明确需求范围和/或质量给予了最高优先权。
- 2，为项目建立一个固定的最大预算
- 3，你同意力争希望的最后期限，但项目所有者接受了现实，如果必须把某些工作后延，可以对它进行调整。

现在假定系统分析期间发现了重要的和未预料的业务问题，对这些问题分析将使项目进度延后，而且这些新需求又扩展新的用户需求，作为项目经理你将如何应对？

首先，对进度延误不要过度反应，这是可以接受的。问题是新的需求加入将会增加费用，而费用是受限的，这就有了一个需要解决的问题，这就使“费用”这个度量处于危险的状态。系统所有者需要知道哪些度量处于危险状态，然后和项目经理一起来讨论这个问题，有几种解决方案。



- **可以重新分配资源（费用、进度）：**也许系统所有者可以在其它地方找到更多的经费，这样度量优先级可以保持不变，注意修改是建立在进度已经延迟的基础之上的。
- **可以增加预算：**预算可以在计划外的进度延迟抹平，比如把项目延迟到下一个财政年度，就可以分配到额外的经费，而不必从现有的经费中调配，这个方案如下图所示。

优先权 成功度量	最大或最小	受限的	可接受的
费用 调整预算		×+ 增加预算	
进度 调整最后期限			×- 延长最后期限
范围和/或质量 调整项目范围	×+ 接受扩充的需求		

- **可以重新排列需求的优先次序：**可以把某些需求推迟到第二版实现，这样就减少了用户需求，如果预算不能增加，这个替代方案将是合适的。
- **可以改变度量优先权。**

只有系统所有者可以改变度量优先权。例如，也许系统所有者认为，对增加的需求投入额外的经费是值得的，他们同意分配足够的经费来满足需求，但同时调整优先权，使费用最小成为最高优先权（如下图的第一步）。

这样一来就违反了矩阵的规则，因此把范围和质量调整为受限的（下图的第二步）。结果，这个系统变成保证费用最小，但冻结需求的增长，并且仍然接受进度的延误。

成功度量 \ 优先权	最大或最小	受限的	可接受的
费用	第一步 		
进度			x
范围和/或质量	第二步 		

关于优先权变化还有三条说明：

首先，一个项目的优先权可以多次变化，只要保持矩阵的平衡，就可以任意改变预期。

其次，预期管理可以通过组合优先权变化和资源调整来实现。

最后，即使项目符合进度，系统所有者也可以改变优先权，例如：在一个正在进行的项目中，政府可能强制一个不可折中的最后期限，这就突然把进度变成最大约束，使我们不得不调整以平衡矩阵。预期管理是一个简单的工具，但有时简单的工具往往是最有效的工具！

结束语：时代呼唤优秀的软件架构师

在软件组织中，架构师的作用是举足轻重的，当企业把一个方向的生命线托付给你的时候，责任也是重大的，因此架构师必须十分谨慎和细致，最后我给你提如下一些建议：

1，架构师的知识结构

- 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

2，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。

3，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。

4，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。

5, 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。

6, 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。

7, 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。

8、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。知识经济的时代在呼唤优秀的软件架构师，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。