目录

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

前言

在软件组织中，高水平架构师队伍的作用举足轻重，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，其主要的思想如下：

1，随着经济全球化进程的不断推进，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，所以，如何设计高质量的软件产品，成为软件架构设计的重要主题。在架构设计上，我们应该研究如何尽可能在达到质量需求的基础上，使高投资回报率成为可能，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。

2，规模软件经济的理念，对设计方法和思路提出了完全不同的要求，重用和重构成为重要的主题。复用的思想，目前正经历从下游到上游延伸的过程，从设计模式延伸到分析模式、业务架构模式。软件架构决不是一个孤立的设计问题，一个好的架构师必须从业务领域、需求分析直至架构设计具有深刻而且现代的理解，能很好地实现各个节点之间的过渡，模型驱动的设计与开发是我们必须认真研究的问题。

3，在今天的企业环境下，变化就意味着胜出，因此软件产品开发中的需求变更不可避免，而需求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。

4，应对变更的架构设计必须对业务的变化规律进行深入分析，之中有两个不同粒度下的思考基点：一个是业务流程不变而业务单元可能变，相应的处理方法是合理应用设计模式构建软件框架；另一个是业务单元不变而业务流程可能变，处理方法就是适应业务流程敏捷性处理的面向服务的架构。这是从两个不同侧面讨论架构问题，需要我们有深入而现代的理解。

5，近年来，由于项目越来越大、越来越复杂，应对软件的易变性就不可能单单从架构设计本身解决问题，而需要有更加合理的项目过程，敏捷过程就是其中有代表性的新方法。但是敏捷开发的基础又是架构驱动，所以，我们必须研究敏捷过程下的架构设计问题。

6，为了延长软件架构的生命周期，提高已有架构资产的利用率，软件架构的恢复、重构已经成为业内的关注点，我们必须研究如何条理化的组织架构恢复和重构工作，使架构恢复和重构成为现实可能。

对上述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。这需要软件架构师具有很高的水平，才能使设计工作变得极有主动性和想象力。这一整套思想的实现，也构建了高质量软件系统坚实的基础。

本课程并不准备泛泛讨论软件架构设计一般方法与过程，而是针对上述核心问题和关键思考点，从系统的角度寻找相应的对策和解决方案。我们将会通过一系列精心选择的案例，从正反两个方面加以分析，多视角、全方位、在理论和实践两方面全面研究问题。通过本课程学习，希望学员在今后架构设计的实践中，在完成必须的功能性需求和性能指标的基础上，进一步优化架构设计，确保以低的开发成本达到高的质量要求，从而大大提高设计水平，为企业创造更高的可度量价值。

第一章 现代软件开发过程及架构策略

任何设计方案都与过程有关，过程的改进必然对架构设计产生深远影响。

1.1 软件架构设计师的的知识体系

一、软件架构的定义与问题

软件架构（software archiecture）也称之为软件体系结构，它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，一直以来，对于架构的理解有两个基本概念，一个称之为组成，另一个称之为决策。

- **组成：**架构的组成概念强调“计算机及组件之间的交互”。例如在的初步设计中，“表示层”和“业务层”是两个粗粒度的黑盒，当内部也表达了一些粒度比较细的组件的时候，这两个黑盒变成了“灰盒”。交互的概念表现在架构描述了它们之间的关系，例如数据如何读取、功能如何调用等。
- **决策：**架构决策不但表现了系统组织、元素、子系统的组织风格决策，还包括了非功能性需求的决策，例如对于可扩展性的决策，对于表示逻辑与业务逻辑变化的隔离，第三方工具包变化的隔离等，这就使架构有了弹性。

架构的**组成** 与**决策**是架构设计的两个基本概念，这两个概念并不矛盾，在架构设计中，往往是同时体现这两个概念，确保架构满足产品要求。由这两个概念出发，我们自然会提出：软件架构的核心思维到底是什么呢？

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全面深入的需求分析作为基础，根据需求来组织合理的产品架构。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。

其次，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，所以架构师的眼光一般都专注于质量属性上，应该根据产品质量属性的要求提出合理的架构决策。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，特别是发生变更的时候带来了很大的困难。这就给我们提出了一个问题，架构设计的思维到底是什么？

另一方面，任何架构思想的实现，必须与具体的项目组织相匹配才能发挥作用。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，提出合适的软件工程策略。反之，一个软件工程策略，也不可避免的也会影响到架构设计的特点。

上述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，架构整体特征应该考虑项目管理特征。因此，软件架构设计是一个系统工程，它需要系统构架师有很宽的知识面，从需求分析、架构设计到类设计甚至代码实现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、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，而对很多国内企业而言，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过程的复制，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。战略规划是一套方法论，用于企业的业务和 IT 的融合以及 IT 自身的规划。必须满足如下要求：

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

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

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

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

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

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

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

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

Who：为谁设计？

What：要解决用户的什么问题？

Why：为什么要解决这些用户问题？

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

Who? 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，企业的业务过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过程是不可取的，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，通过全方位的努力，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。

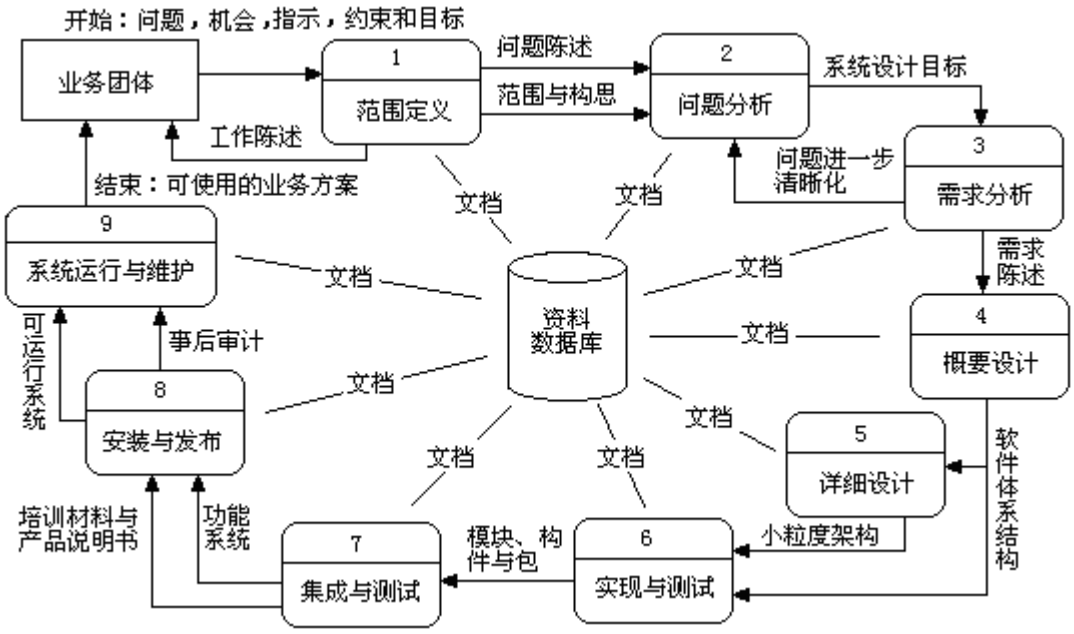
1.2 从线性模型到迭代模型

研究软件企业架构，不可避免的需要研究软件开发过程的有关问题，因为正是由于现代项目管理与过程管理的巨大变化，才对架构设计提出了新的要求，造成了现代架构设计与经典概要设计方法上根本的不同。所以，我们必须要仔细研究一下现代软件开发过程有什么变化呢？

一、经典软件开发过程模型

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，早在 20 世纪 70 年代，人们为了改变软件开发混乱、随意、无秩序的状态，提出了经典的瀑布式软件开发过程。在这样的模型下，人们发展了经典的项目管理方法，提出了诸如 PERT（计划评审技巧）、甘特图以及早期实现软件预估的一系列项目管理原则。

在经典项目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。模型要求在任何设计和实现工作之前，尽可能的推敲，把需求完全定义清楚，并把它稳定下来，并且实际开发前冻结需求。在概要设计阶段主要需要建立系统高层模型，建立系统和子系统的框架以及基于服务的层等。在详细设计阶段，可以精细的把业务需求转换为系统模型。然后在实现诸如编码、测试、系统集成以及发布等下游模 型。经典瀑布式过程的大致情况如下图所示。



二、经典项目过程导致失败的原因

经过 20 年的应用，人们发现经典项目过程存在很多问题，这迫使人们研究它存在的问题，以期找到解决方案。一个项目成功的关键是有良好的规划，产品在初期规划的时候，应该说明产品应该具备哪些能力？在什么时间完成？需要哪些资源？需要多少资源？这些信息有助于我们通过项目规划减少风险、降低产品目标的不确定性、为更好的决策提供支持，这是构建良好的项目管理必备的信息。但是，经典项目管理最困难的问题是项目难以规划，做出来的计划往往会出错。于是开发小组往往走向两个极端：要么是不做规划，这样的小组根本无法回答“你们什么时候能完成？”这样的问题。要么是在计划中投入大量的精力，直到自己确信计划是正确的，但这种“正确”往往是自欺其人的，这种计划可能更全面，但不意味着更准确或更有用。因为软件的特点不确定性是始终存在的。

尽管经典项目管理把费用、质量（功能目标）、时间设定为最重要的三角约束，遗憾的是很多人的研究都告诉我们，传统方法不一定会得出满意的答案，下面是很多人做的研究结果：

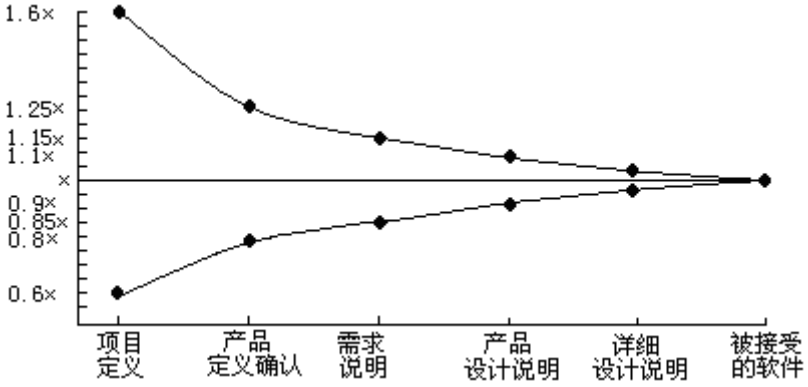
- 大约 2/3 的项目会显著超出费用预算（Lederer and Prasad 1992）。
- 产品中 64%的功能很少或者从不会被使用（Johnson 2002）。

- 一般项目花费的时间会超出进度表 100%（Stndish 2002）。

这就迫使我们仔细研究项目失败的原因，归纳起来可以有 5 个原因。

1）初期的估计偏差可能很大

下图显示了 Boehm 所考虑的不确定性在顺序开发过程不同点的范围，这个图称之为不确定性锥形。 图上表明，在项目定义阶段，对进度估计的偏差可能达到 60% ～160%，也就是说一个预 期 20 周完成的项目，实际花费可能在 12～32 周之间。而在写下需求之后，估计的偏差可能还在±15%，这时的 20 周预期可能实际上可能会花 17～23 周。这么大的预估偏差根本就无法实施有效的项目管理。



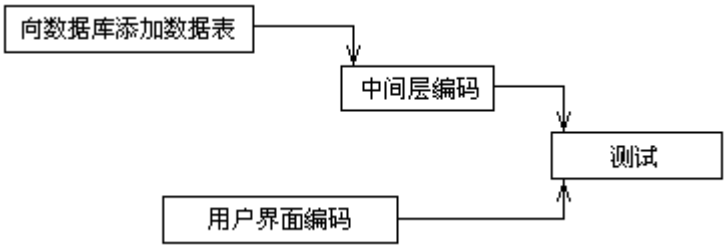
2）活动不会提前完成

设想某一个资深程序员正在为一个产品编制一个有趣的新功能，同时他还承担了一个为 CMMI 审核准备文档的任务，他会怎么样分配时间？毫无疑问，他会把大部分时间用于编制这个有趣的程序，而准备审核只留下了刚刚够用的时间，而且是到最后才不得不草草完成。实际上这种行为非常普遍，以至于有了一个名称叫帕金森定律（Parkinson’s Law，1993）：“工作总是要拖到最后一刻才完成。”

如果一项工作在甘特图上分配了 5 天，处理这个活动的员工一定会用满 5 天，即使能够提前完成，他也会通过增加一些花哨的修饰（镀金需求），或者尝试着用一些新的热点技术。在不少企业文化中，一项活动提前完成，往往被指责为当初他做的估计不准确，或者会另派其它的任务给他，为什么要冒这个风险来提前完成呢？人的本性就是如此：用多余的时间去做一些对自己有价值，但对别人不一定有用的事情。

3）延误沿着进度表向下传递

由于传统的计划是基于活动的，因此它们主要关注活动之间的依赖性。考察下面的甘特图，它显示了 4 项活动及它们之间的依赖关系。



如果需要提前完成测试，就要求一些事件的幸运巧合：

- 对中间层的编码提前完成，而它受到向数据库添加数据表这一活动完成时间的影响。
- 对用户界面的编码提前完成。
- 提前安排好测试人员。

关键之处在于，即使一个如此简单的案例中，在启动测试之前也有 3 件事情必须发生，

但是下面任何一件事情，都可能导致测试推迟：

- 用户界面编码结束时间延误。
- 对中间层编码花费时间超出计划。
- 中间层编码花费时间符合要求，但是向数据库添加数据表结束过晚，导致延误了测试时间。
- 未安排好测试人员。

也就是说提前启动测试需要完成很多事情，其中一件事情延误，都可能导致总体上的延误。

事实上我们已经确认了活动很少会提前完成，所以发生的绝大多数事情是延误，而这种延误会沿着进度表传递下去，最终导致后续项目很少会提前启动。

4）不按照优先级开发功能

传统项目管理方法不一定能够带来高价值产品的第三个原因是，制定的计划没有按照对用户或者客户所具有的价值大小来排列工作的优先级。很多传统的计划实际上假定工作都可以完成，工作的顺序主要是由依赖性来决定。但是随着项目结束时间的逼近，开发小组会匆忙放弃一些功能以跟上进度，由于不是按照功能优先级顺序开发的，某些被放弃的功能反而比交付的功能更重要。

5）忽视了不确定性

传统规划方法的第 4 个缺点，是不承认不确定性的存在。人们假定最初的需求分析就可以产生对产品来说完整的、完善的定义。我们假定用户在计划覆盖的整个时间内都不会改变想法，他们的观点也不会更细化，也不会提出新的需求。

与之类似，我们还忽视了如何构建产品这样的不确定性，某种技术在实现中才发现并不一定合适，但我们已经给它分配了确定的时间（两个星期），事实上我们不可能指望一开始就确定项目进程中所需要的所有活动，但我们又不愿意承认这一点。

在分析了传统项目管理方法所存在的问题以后，我们对那么多项目令人失望就不会感到奇怪了。基于活动的规划方法分散了我们对功能的注意力，而功能才是衡量客户价值的单元。本来项目的参与者满怀好意的把多任务处理当成解决延误的办法，结果却造成项目更加的延误。当进度表剩余时间不够的时候，不可避免地要放弃一些功能，由于开发人员是按照最有效的开发方式来安排进度的，因此放弃的功能对用户来说不一定是价值最低的。

忽视用户需求的不确定性，会导致虽然按时完成了任务，但很多后来发现的重要功能没有办法加入，或者即使加入了但要不是项目延误，或者不恰当的降低质量。

这是在这些分析的基础上，我们就必须考虑解决这些问题的办法。

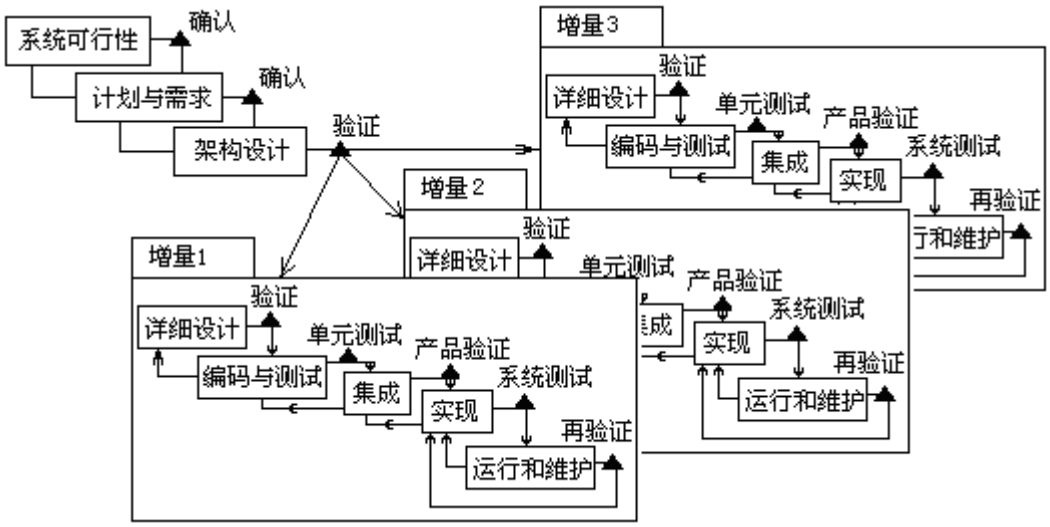
三、软件开发增量模型的提出

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。



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泪丧。
- 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，因为每次曾两开发的时间很短。
- 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。

1.3 大型项目敏捷模型中的架构设计

增量模型的提出，需要有一整套相应的项目管理理念与之匹配，人们通过多年的努力，从理论上和实践上作了艰苦的探索，提出了比较完整的敏捷项目管理方法。需要说明的是，敏捷开发并不是回归到当初混乱无计划开发状态，更不是赞成随意性。敏捷方法是在长期的实践中，对瀑布式模型存在问题的分析结果，也是从发现问题、分析问题到解决方案的过程。对管理水平上的要求是更高而不是更低。

一、敏捷开发的价值观

实际上敏捷开发运动在数年前就开始了，但它正式开始的标志是 2001 年 2 月的“敏捷宣言”（Agile Manifesto），这项宣言是由 17 位当时称之为“轻量级方法学家”所编写签署的，他们的价值观是：

- 个人与交互重于开发过程与工具；
- 可用的软件重于复杂的文档；
- 寻求客户的合作重于对合同的谈判；
- 对变化的响应重于始终遵循固定的计划。

个人与交互重于开发过程与工具的原因：

一个由优秀的人员组成但使用普通的工具，要比使用优秀的工具但由普通人组成、紊乱的小组做得更好。多年来人们花了很多时间试图建立一种过程，以便把人当作机器上的一个可以替代的齿轮，但结果却并不成功。敏捷过程是承认每个人都有特定的能力（以及缺点），并对之加以利用，而不是把所有的人当成一样来看待。

更重要的是，在这样的理念下，几个项目做下来，每个人的能力都从中得以提高。这种人的能力的提高，对公司是无价之宝。而不至于把人当成齿轮，随着时间的推移，人的能力慢慢被消耗掉，最后变成留之无用、弃之可惜的尴尬人物。

可用的软件重于复杂的文档的原因：

可用的软件可以帮助开发人员在每次迭代结束的时候，获得一个稳定的、逐渐增强的版本。从而允许项目尽早开始，并且更为频繁的收集对产品和开发过程的反馈。随着每次迭代完成软件的增长，以保证开发小组始终是处理最有价值的功能，而且这些功能可以满足用户的期待。

寻求客户的合作重于对合同的谈判的原因：

敏捷开发小组希望与项目有关的所有团体都在朝共同方向努力，合同谈判有时会在一开始就使小组和客户出于争执中。敏捷开发追求的是要么大家一起赢，要么大家一起输。换句话说，就是希望开发小组和客户在面对项目的时候，以一种合作的态度共同向目标前进。当然，合同是必需的，但是如何起草条款，往往影响到不同的团体是进行合作式的还是对抗式的努力。

对变化的响应重于始终遵循固定的计划的原因：

敏捷开发认为对变化进行响应的价值重于始终遵循固定的计划。他们最终的焦点是向用户交付尽可能多的价值。除了最简单的项目以外，用户不可能知道他们所需要的所有功能的每个细节。不可避免地在过程中会产生新的想法，也许今天看起来是必需的功能，明天就会觉得不那么重要了。随着小组获得更多的知识和经验，他们的进展速度会比开始的时候期望值慢或者快。对敏捷开发来说，一个计划是从某个角度对未来的看法，而具有多个不同的角度看问题是有可能的。

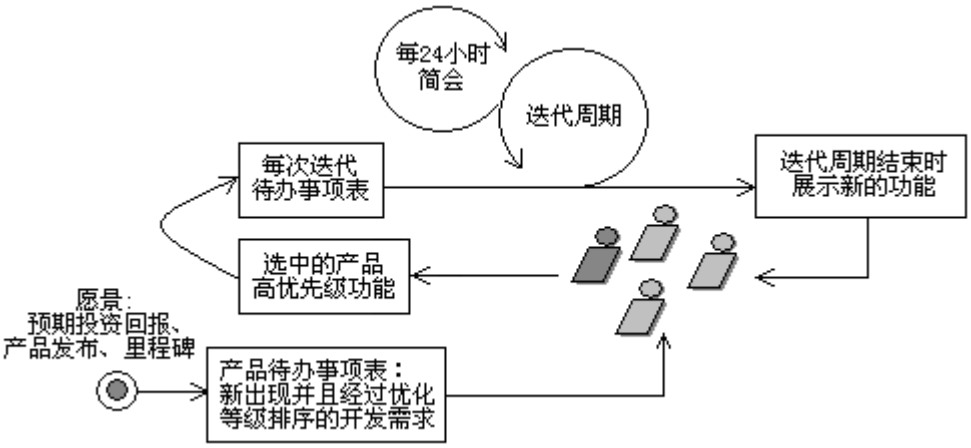
二、项目的敏捷开发方法

敏捷方法很多，包括 Scrum、极限编程、功能驱动开发以及统一过程（RUP）等多种法，

这些方法本质实际上是一样的，敏捷开发小组主要的工作方式可以归纳为：

- 作为一个整体工作；
- 按短迭代周期工作；
- 每次迭代交付一些成果；
- 关注业务优先级；
- 检查与调整。

下图是典型的敏捷过程总图，我们简要的介绍一下有关的特点。



1，敏捷小组作为一个整体工作

项目取得成功的关键在于，所有项目参与者都把自己看成朝向一个共同目标前进的团队的一员。“扔过去不管”的心理不是属于敏捷开发。设计师和架构师不会把程序设计“扔”给编码人员；编码人员也不会把只经过部分测试的代码“扔”给测试人员，一个成功的敏捷开发小组应该具有“我们一起参与其中的思想”， “帮助他人完成目标”这个理念是敏捷开发的根本管理文化。当然，尽管强调一个整体，小组中应该有一定的角色分配，各种敏捷开发方法角色的起名方案可能不同，但愿则基本上是一样的。

第一个角色是产品所有者，他的主要职责包括：

- 确认小组成员都在追求一个共同的目标前景；
- 确定功能的优先等级，以便总是处理最有价值的功能；
- 作出可以使项目的投入产生良好回报的决定。

产品所有者通常是公司的市场部门或者产品管理部门的人员，在开发内部使用的软件的时候，产品所有者可能是用户、用户的上级、分析师，也可能是为项目提供资金的人。

第二个角色是客户：

客户是做出决定为项目提供资金或者购买软件的人，在一个开发内部使用的软件项目中，客户通常是来自另一个团组或者部门的代表，在这样的项目中，产品所有者和客户的角色往往是重合的。对一个商业软件来说，客户就是购买这个软件的人。客户可能是也可能不是软件的用户（user），用户当然也是一个主要角色。

一个值得注意的角色是开发人员（developer）：所有开发软件的人，包括程序员、测试人员、数据库工程师、可用性专家、技术文档编写者、架构师、设计师、分析师，等等。

最后一个角色是项目经理：

在所有的敏捷开发项目中，项目经理的角色发生了变化，他更注重的是教导、引导而不是管理和控制。在某些敏捷开发项目中，项目经理也承担其它的角色，通常是开发人员，少数的时候也会担任产品所有者。

2，敏捷小组按短迭代周期工作

在敏捷项目中，总体上并没有什么上游阶段、下游阶段，你可以根据需要定义开发过程。在初始阶段可以有一个简短的分析、建模、设计，但只要项目真正开始，每次迭代都会做同样的工作（分析、设计、编码、测试。等等）。

迭代是受时间框限制的，也就是说即使放弃一些功能，也必须结束迭代。时间框一般很短，大部分是 2～4 周，在 Scrum 中采用的是 30 个日历天，也就是 4 周。

迭代的时间长度一般是固定的，但也有报告说，有的小组在迭代开始的时候选择合适的时间长度。

3，敏捷小组每次迭代交付一些成果

比选择特定迭代长度更重要的，是开发小组在一次迭代中要把一个以上的不太精确的需求声明，经过分析、设计、编码、测试，变成可交付的软件（称之为功能增量）。当然并不需要把每次迭代的结果交付给用户，但目标是可以交付，这就意味着每次迭代都会增加一些小功能，但增加的每个功能都要达到发布质量。每次迭代结束的时候让产品达到可交付状态十分重要，但这并不意味着要完成发布的全部工作，因为迭代的结果并不是真正发布产品。

假定一个小组需要在发布产品之前对软硬件进行为期两个月的“平均无故障时间”（MTBF）测试，他们不可能缩短这两个月的时间，但这个小组仍然是按照 4 周迭代，除了 MTBF 测试，其它都达到了发布状态。

4，敏捷小组关注业务优先级

敏捷开发小组从两个方面显示出他们对业务优先级的关注。

首先，他们按照产品所有者制定的顺序交付功能，而产品所有者一般会按照组织在项目上的投资回报最大化的方式来确定优先级，并且把它组织到产品发布中去。要达到这个目的，需要综合考虑开发小组的能力，以及所需功能的优先级来建立发布计划。在编写功能的时候，需要使工能的依赖性最小化。如果开发一个功能必须依赖其它 3 个功能，那产品所有者这就很难确定功能优先级。功能完全没有依赖是不太可能的，但把功能依赖性控制在最低程度还是相当可行的。

5，敏捷小组检查与调整

任何项目开始的时候所建立的计划，仅仅是一个当前的猜测。有很多事情可以让这样的计划失效：项目成员的增减，某种技术比预期的更好或更差，用户改变了想法，竞争者迫使我们做出不同的反应，等等。对此，敏捷小组不是害怕这种变化，而是把这种变化看成使最终软件更好地反映实际需要的一个机会。

每次新迭代开始，敏捷小组都会结合上一次迭代中获得新知识做出相应调整。如果认为一些因素可能会影响计划的准确性，也可能更改计划。比如发现某项工作比预计的更耗费时间，可能就会调整进展速度。

也许，用户看到交付的产品后改变了想法，这就产生反馈，比如他发现他更希望有另一项功能，或者某个功能并不像先前看得那么重。通过先期发布增加更多的用户希望的功能，或者减少某些低价值功能，就可以增加产品的价值。

迭代开发是在变与不变中寻求平衡，在迭代开始的时候寻求变，而在迭代开发期间不能改变，以期集中精力完成已经确定的工作。由于一次迭代的时间并不长，所以就使稳定性和易变性得到很好的平衡。在两次迭代期间改变优先级甚至功能本身，对于项目投资最大化是有益处的。

从这个观点来看，迭代周期的长度选择就比较重要了，因为两次迭代之间是提供变更的机会，周期太长，变更机会就可能失去；周期太短，会发生频繁变更，而且分析、设计、编码、测试这些工作都不容易做到位。综合考虑，对于一个复杂项目，迭代周期选择 4 周还是

有道理的。

在本节结束的时候，让我们看一下 MIT Sloan Management Review（麻省理工学院项目管理评论）所刊载的一篇为时两年对成功软件项目的研究报告，报告指出了软件项目获得成功的共同因素，排在首位的是迭代开发，而不是瀑布过程。其它的因素是：

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

从这个报告中所透露出的信息告诉我们，认真研究敏捷过程对软件项目的成功是非常有意义的，它的意义在于：

1）给开发小组的自组织提供了机会

经典项目管理就好比一个具备中央调度服务的航空管理系统，这个系统是自治的，而且是封闭的，但现实中更庞大的系统，似乎并不属于中央调度控制系统，但也同样也是有效的。假如我们开车到某个地方，我们可以任意选择所需要的路线，我们甚至不需要准确计算停车地点，只要我们遵守交通法规，驾驶员可以临时根据路况改变某个转弯点，在驾驶游戏规则的框架内，依照自身最大利益做出决策。

成千上万的驾驶者，并不需要中央控制和调度服务，人们仅仅在简单的交通法规的框架内，就可以完成综合起来看是更庞大的目标，很多事情从管理的角度只要抓住关键点，并不需要多么复杂的规则，往往会更有效。

随着系统复杂度的提高，中央控制和调度系统面临崩溃。仔细研究交通系统的特点，我们会发现这样的系统中独立的个体在一组适当的规则下运行，并不需要设计每个个体临时变更的方案，而每个个体只需要知道目标和大致的状况，他们完全可以利用自己的聪明才智来决定自己的行为。

2）缩短了反馈通道

敏捷过程有效运作的另一个原因是，它极大的缩短了用户与开发者、预测目标与实施状况、投资与回报之间的反馈回路。在面对不断变化的市场、业务过程以及不断发展的技术状态的时候，便需要有一种方法在比较短的时间内发展完善。

事实上，所有的过程改进都不同程度的使用着戴明循环，以研究问题、测试解决方案、评估结果，进而根据已知的事实来进行改进，这就称之为基于事实的决策模式，我们都知道，这比前端预测的决策方式更加有效。

3）易于集思广益

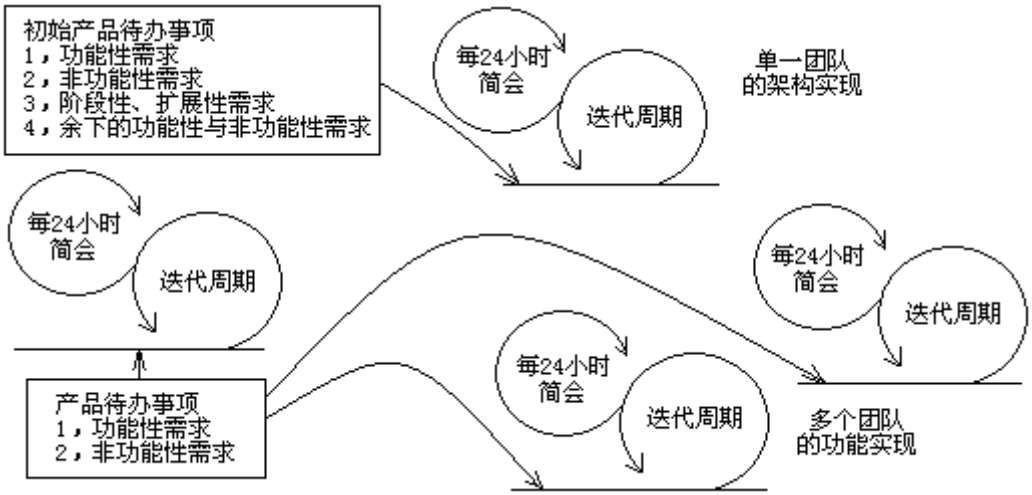
敏捷过程能有效应用的另一个原因在于，它可以就一个问题集思广益。我们的经验告诉我们，当一个问题发生的时候，总有某些人员知道问题所在，但他的观点却遭到忽视。例如航天飞机在起飞阶段发生爆炸，事后分析出了各种原因，但这种调查也提供给我们一个惊人的事实，就是部分工程师早就意识到这些潜在问题，却无法说服他人重视这个担忧。对这些事实的深入思考，促使我们研究我们应该采取何种管理系统，使前线工作人员的经验、观点及担忧得到充分的重视呢？

三、在大型敏捷项目多维度扩展下的架构

许多项目无法由一个小型团队完成，这种情况需要多团队合作。人们通过一系列的机制来协调多团队并行开发。两个或者两个以上的小型团队同时开发的项目称作“扩展项目”，协调这些项目的机制称作“扩展机制”，每个扩展项目有其自己的复杂性，需要独特的解决方案。扩展的核心是团队，一个 800 人的项目需要包含 100 个 8 人团队，我们将如何协调全部团队的工作呢？

敏捷过程扩展成功的关键：首先，在扩展前构建必需的基础设施和基础架构，一般设计

和构建基础架构需要经过几个迭代周期。第二，构建基础设施的同时，确保交付商业价值，这种商业价值也包括将来使用这个基础设施的应用案例。第三，其它团队可以在后期建立。优化原始团队的能力，向其它团队分派至少一名初始团队成员。同时还要注意，项目一开始就取得进展，对取得利益相关者的支持很重要，但应防止扩展速度过快。上述迭代扩展的方法，可以用下图表示。



但是不要认为在多维度扩展项目整体控制上还是要采用敏捷过程特有的自组织和自管理的规则。在总的控制上，采用类似预定义过程的分层管理结构在很多情况下都是合适的，这样可以降低管理上的复杂性，而且，总体方案上的变更本来就比较小，这样更容易协调和扩展项目。不管怎么说，一个大型复杂项目的框架被说成是易变的、无法预测的恐怕并不一定符合实际情况，所以书生气十足的建立全面敏捷管理很可能最终把事情搞坏。

大型项目的高级管理层也不可能达到与团队进行适应的频率和准确度，干预程度也往往不容易掌握，经过一些对比的尝试，感觉敏捷项目多维度扩展的高层管理还是经典项目管理理念比较有效，这是稳定性和灵活性的统一。

1.4 选择合适的软件工程策略

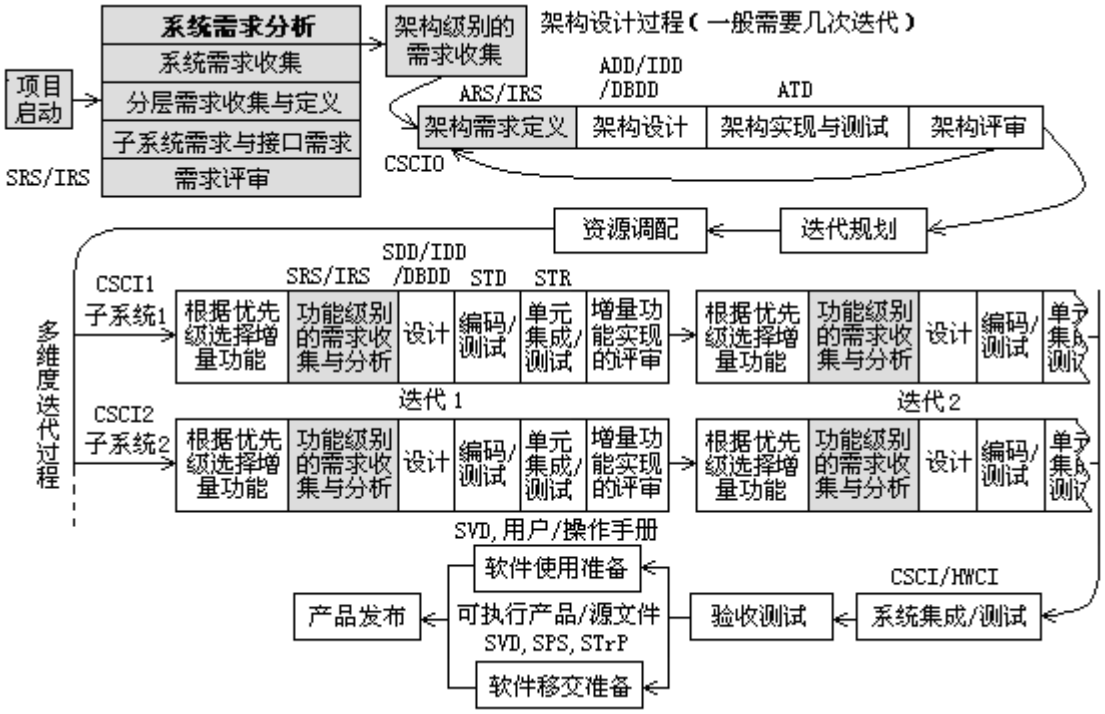
一、软件工程策略的分类

在项目很庞大的时候，需要选择合适的软件工程策略，有三种工程项目策略如下表所示：

三种工程项目策略的关键特性			
工程策略	**首先定义所有需求？**	**存在多次开发循环？**	**现场安装中间软件？**
一次性完成策略	是	否	否
增量式	是	是	可能
进化式	否	是	是

1，一次完成策略：

这是一种典型的“一次设计、一次通过”的策略，不论是原则上是采取线性过程还是迭代过程，在初期的需求完成以后，最后要交付完整的产品，如下图所示。



图中也标出了在各个阶段建议的文档，文档的略语表如下。

软件工程策略略语表			
CSCI	计算机软件配置项	DBDD	数据库设计说明
SRS	软件需求规格说明	STD	软件测试说明
IRS	接口需求规格说明	STR	软件测试报告
ARS	架构需求规格说明	HWCI	硬件配置项
ADD	架构设计说明	SVD	软件版本说明
ATD	架构测试说明	SPS	软件产品规格说明
SDD	软件设计说明	STrP	软件移交计划
IDD	接口设计说明		

在项目的早期，我们的精力集中于系统级的需求收集，在这个层面，我们关注于整个系统的功能和非功能需求，这时候的功能需求可以用“用户描述”来表达，非功能需求可以生成“补充规格说明”。对于规模比较庞大的“用户描述”，我们可以对这样的需求进行划分，从而引发一些派生的需求（例如子系统需求与接口需求等）。这个阶段的需求必须经过评审，以期把整个项目完整的定义下来，同时可以用这样的需求信息对项目进行早期的估计。

项目的第二个阶段，我们把精力集中于架构级别的需求收集，进一步精化和优化系统划分。这时候需求收集的重点是各个子系统和模块的需求定义，确定它们如何协调和通讯，以及架构级别的非功能性需求等。架构设计一般经过一到两次迭代，设计的结果需要经过评审，从而确定整个产品的体系结构已经完成。

在软件架构完成以后，需要根据已有的需求信息进行整个项目的规划和估计，针对每个子系统分配资源，组织适当数量的开发团队，使每个开发团队进入各自的迭代周期。

在每个迭代周期的开始，需要根据需求的优先级选择适当数量的的增量功能，在迭代开始以后，首先是对这些增量功能（以用户描述表达）进行详细的需求收集与分析，这个级别的需求是由迭代小组中的需求人员带领小组全体成员共同来完成，然后编写详细级别的需求文档并经过确认。以此为基础顺序的进入设计、编码、测试等各个阶段。每次迭代交付的是可以提交的功能增量。

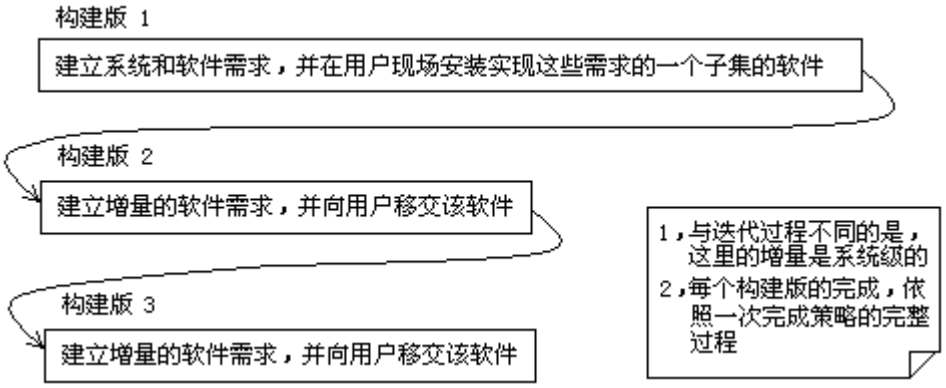
每个迭代期间，对需求变更是封闭的，但是在每次迭代开始“选择增量功能”阶段，需要对需求的变化进行收集和考虑，这个时候对需求变更是开放的。要注意的是，每次变更都

必须修改相应的需求文档，并做下记录。在这样复杂的过程中，良好的软件配置管理是非常必要的。

当所有的子系统都完成以后，还是需要进行最后的系统集成和测试，这可能要经过一到两次迭代，最后是验收测试和产品发布。

2，增量式策略：

这种策略的特点是确定用户需要和定义系统需求以后，按照构建版顺序完成其余的开发任务。第一个构建版纳入部分计划和能力，下一个构建版再增加一些能力等，直到系统全部完成。其中，每一个阶段与一次完成的策略是一样的。



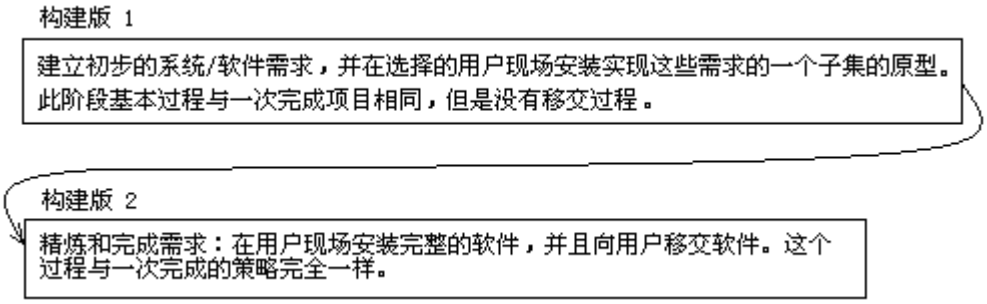
这种策略的本质是一种螺旋方式，开始于一个适度的、良好定义的问题，解决它，再在下一阶段扩展它。在开发的每一阶段，都有一个良好定义的而不是粗略的需求。每一阶段都为良好定义的问题产生一个可以证明是正确地解决方案。生成软件即使有漏洞也很少，从每个解决方案我们都可以知道在下一阶段如何来扩展这个问题，也就是改善需求。

这种策略最成功的应用，当数上个世纪 60 年代美国的阿波罗计划。如此庞大的计划不可能一开始就尝试所有的细节。于是，十年来 NASA 进行了多次发射，每次发射都力图解决登月计划问题集的每一个，例如太空生活供给问题，太空外活动问题，太空对接问题等。整个计划都是由需求来驱动，每次经验都对下次的需求设计有所改进。

向解决方向上增加解决方案的好处，使它可以解决我们从没有想出来的问题，当它成功的时候，能够发现我们从未像想到的问题，从而引发新的需求，而这些需求又是与用户的需要是吻合的。拥有良好需求定义的螺旋式方法，支持在问题域的每一个阶段设计软件的期望结果，也就有更大的机会在在早期发现错误，由于每一阶段都有一些良好定义的东西去测试，甚至实现某种模式，在后期设计中将会变得更加胸有成竹。

3，进化式策略：

进化式策略的一种是原型方式，它与增量式方法最大的区别是不能有一个良好定义的需求，即使是一个阶段也不可能。这是一种特殊的“增量式”策略，所不同的是承认用户的需求不完全清楚，不可能预先定义全部需求。采用此策略的时候，用户需要和需求预先部分的定义，然后在随后的构建版中逐步得精炼。



进化式策略目的是为需求和最终的软件规格说明书激发灵感。所以，需求本身并不是随意的，也不能与需求的严格定义相冲突。最终，我们还是需要通过观察每个构建版的行为来改进需求，也必须让程序严格的映射到需求。

原型方式实际上已经构建了一个模式，而这个模式是站在设计方的角度构建的，我们希望通过原型达到的，正是突破自己已经构建的模式，使产品真正符合用户的需要。一个优秀的作家往往把自己的草稿给一部分读者阅读，读者看到了这些草稿，就有了一个基础，激发了想象力，当这些读者提出各种各样的意见的时候，作家就可以换个思维方式面对自己的草稿，突破自己已经形成的思维模式，这样他就可以写出真正优秀的作品来。

进化式原型将是最终产品的一部分，所以从架构设计来说必须设计为易于升级和优化的，因此，我们应该重视软件系统性和完整性的设计原则，要达到进化型原型的质量要求并没有捷径。我们应该考虑进化型原型的第一次演变，因为它将作为实现需求中易于理解和稳定部分的试验性版本。从测试和首次使用中获得的信息将引起下一次软件原型的更新，正是这样不断增长并更新，使软件才能从一系列进化型原型逐渐发展为实现最终完整的产品。

二、利用风险分析选择合适的工程项目策略

工程项目策略由需方选择，但也可以由未来的或者已选定的开发方提议。下表说明了一种合适的策略所使用的风险分析方法，这种方法就是：列出每种策略的风险项（负面的）和机会项（正面的），为每个项确定风险的机会和等级（高、中、低）：根据风险和机会的权衡做出使用哪种策略的决定，表中所列的只是一个例子，实际分析可以采用其它的方法。记录在最后一行的“决定”表明选择了这种策略。

为确定适当的工程项目策略而进行的风险分析样例					
一次性完成设计		增量式策略		进化式策略	
风险项（反对这个策略的理由）	风险级别	风险项（反对这个策略的理由）	风险级别	风险项（反对这个策略的理由）	风险级别
1，没有很好地理解需求。 2，系统太大以至于不能一次性的完成。 3，预期任务和技术的快速变化可能会导致需求的改变。 4，现在可用的人力资源或者预算有限。	高 中 高 中	1，没有很好地理解需求。 2，在第一次交付的时候用户就要求所有的能力。 3，预期任务和技术的快速变化可能会导致需求的改变。	高 中 高	1，在第一次交付的时候用户就要求所有的能力。	中
机会项（支持这个策略的理由）	**机会级别**	**机会项（支持这个策略的理由）**	**机会级别**	**机会项（支持这个策略的理由）**	**机会级别**
1，在第一次交付的时候用户就要求所有的能力。 2，用户要求立即取消旧的系统。	中 高	1，需要早期能力。 2，系统自然的分解为增量。 3，财力/人力逐步的增加。	高 中 高	1，需要早期能力。 2，系统自然的分解为增量。 3，财力/人力逐步的增加。 4，为理解全部需求，	高 中 高 高

				需要对用户的反馈和对技术变更进行监视。	
				决定：使用这个策略	

工程项目的策略一般适用于整个系统，对系统中的软件可以采用相同的策略去处理，也可以采用不同的策略来得到。

我们可以看出来，无论是经典模型还是敏捷模型，或者是采用何种软件工程策略，作为设计的能力基础实际上是一样的，所区别的是在什么时候使用什么样的粒度来进行分析和设计，或者设计点在什么地方。敏捷模型希望所有团队成员都参与小粒度分析和设计，而不是把分析和设计仅仅看成几个分析师或者设计师的事情，这种方法和理念的不同，更需要所有团队成员具有分析和设计的知识基础。

优秀的设计来自于对问题重点的把握，来自于灵活应用基本方法到自己开发团队的过程中去，来自于对问题的深入理解。所以，在建立了整体框架性思想的指导之下，我们有必要讨论每个细节，对每个细节的知识、思想和能力基础做比较透彻的讨论。

小结：

有些人认为，中小型项目比较适合敏捷过程，而大型项目还是应该使用瀑布式过程比较正确，这是一个误解。在项目比较小的时候，或者比较简单的时候，不确定因素的影响其实是比较小的，这时候使用线性瀑布式过程还是比较合适的。但是当项目很大、很复杂的时候，不确定性因素的影响往往大到使项目无法进行下去的地步，这种情况才迫使人们研究敏捷过程。

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。

多维度敏捷过程的本质是：整体上的宏观规划采用线性过程，部分上的微观规划采用敏捷过程，但每一次迭代内部还是采用线性过程。这种动态和静态的结合，让所有的方法都用在合适的地方，加上架构驱动，这就足以使大型复杂项目得以很好地进行，管理上和规划上的难度也得到了很好的平衡。

第二章 从系统工程的角度构建架构

需求是设计的源泉，设计方案是对需求的应对。架构策略需要从系统工程的角度来思考。

在设计产品时，产品的需求决定了合适的架构方法。反过来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？”

经典瀑布式过程要求在需求开发的时候把所有需求都定义清楚，但在即便如此开发过程中仍然受到需求变更之苦。人们思维方式常常出现的缺失是：当一件事出现困难的时候，总是认为是自己还做得不够好，而不去考虑这件事做法的本身是不是有问题？人们总认为专家制定的规范是不会错的，而不去看一看专家在制定这些规范的时候做了什么假定？规范本身是不是随着社会的进步而变化着？当需求发生变更的时候，总认为自己做的还不够规范，于是进一步强化规范，明确条块分割，结果造成项目更加困难。正是这样的现实，才迫使人们回过头来想，需求发生变更的原因到底是什么？这才有敏捷过程的提出，而敏捷过程又对需求分析和架构设计提出了崭新的要求。这就是软件工程学者这几年如此活跃的原因。

多维度敏捷开发前期的需求开发并不需要太关注某个具体细节，更多的是从整体上考虑和分析问题。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架构。
- 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。

2.1 前景文档与设计方向

在敏捷模型下需求分析的特点，是在早期集中精力，致力于收集架构层面的需求，把重点放在整体性的功能以及功能之间关系的需求收集上。然后在每一个迭代周期，由各个开发小组自行对所选中的用户描述进行详细的需求分析和产品分析，并且在每两次迭代周期之间，由产品负责人根据内外环境的变化，提出变更想法，这就提供了变更的机会。这种方法，使每一阶段关注的重点能够集中精力完成，就有可能高效率的构造高质量的产品。

这个观点非常重要，在项目一开始过多关注某些细节，把精力不恰当地投入在一些细节上，很可能造成对整体关系投入精力不够，最后虽然某一部分实现的很好，但从整体上看可

能是一个失败的产品。组织软件开发与其说好像组织机器制造，更不如果好像组织一场战争。战争与软件开发有一个地方是相通的，那就是过程中充满了不确定因素。在初期构思战役时候，指挥员如果不注意各个部队之间的协调以及力量投放的合理性，而只是关注某些细节，那他就不是一个好的指挥员。如果所有的细节都想清楚了才能开始战争，那就很容易丧失机会。具体的细节的问题，完全可以交给下级部队的指挥员，在不破坏整体构思的情况下，根据现场情况相机处理。这些成功的方法，与敏捷模型的核心思维是一脉相承的。

前景文档（Vision Docunment）是在较高的抽象层次定义问题和解决方案，它使用一般的术语描述描述应用，包括对目标市场、系统用户以及应用特性的描述。前景文档是有效需求过程的关键成分，也可能是最重要的文档。因为一份简短甚至不完全的文档，都有助于项目成员朝向同一个目标工作。团队由此有了共同的目标和剧本，也可能会有共同的心理。如果团队的目标未知而且互相冲突，很快就会产生混乱。

前景文档获取用户需要、系统特性以及项目的其它要求。这样，前景文档的范围横跨需求金字塔的上两层，在较高的抽象级别上定义问题和解决方案。

对于软件产品，前景文档是项目的三个主要内部涉众进行讨论和协商的基础：

- 营销和项目管理团队，作为客户和用户的代理人，将最终解释项目发行成功的原因。
- 开发应用程序的项目团队。
- 管理团队，负责所有工作的商业结果。

前景文档是对产品或者应用中所有最重要的内容的简洁描述，他用简单的语言进行恰当的细节描述，确保项目的主要涉众易于审查和理解，下表概括了前景文档的轮廓。

前景文档模板
1，介绍
提供整个前景文档的概述。 **1.1 前景文档的目的** 文档的目的是收集、分析、定义高层用户需要和产品特性。 **1.2 产品综述** 阐述该应用系统的目的、版本以及要交付的新特性。 **1.3 参考** 这一部分应该列出在前景文档中引用的其它文档的全部清单。
2，用户描述
简要描述系统用户的观点。 **2.1 用户/市场统计** 总结决定产品动机的主要市场统计。 **2.2 用户剖析** 简要描述系统预期用户。 **2.3 用户环境** 描述在使用中包括应用程序和平台等成分的工作环境以及具体的使用模型。 **2.4 关键用户需要** 列出用户认可的关键问题或者需要。 **2.5 替代和竞争对手** 确定用户认为可得到的替代品。
3，产品综述
3.1 产品前景 提供系统或产品及其与外部环境接口的框图。 **3.2 产品定位陈述** 提供一个整体陈述，从最高层面总结产品在市场上的独特定位，推荐以下格式： 为了（目标客户）谁（陈述需要和机遇）“产品名”是一个（产品分类）它（产品名）（对主要优点的陈述，即激发购买热情的原因）不像（主要竞争替代品）我们的产品（对主要区别的陈述）。 **3.3 能力总结** 总结产品将提供的主要优点和特性 **客户利益　支持特性** 利益 1　特性 1 利益 2　特性 2

利益 3　　特性 3 **3.4 假定和相关条件** **3.5 成本和定价** 描述持续产品成本和预期产品定价点的任何点。
4，特性属性
描述将用来评估、跟踪、划分优先级以及管理特性的特性属性，以下是一些建议： 状态：建议的、批准的、合并的 优先级：累计投票结果，顺序分级，或者关键的、重要的、有用的。 工作量：低、中、高，团队周数，或人月数 风险：低、中、高 稳定性：低、中、高 目标版本：版本号 分配给：名字 原因：文本字段
5，产品特性
文档这一部分列出产品特性 **5.1 特性** **5.2 特性**
6，典型用例
描述一些典型用例，可以对架构有意义的，或者最方便帮助读者理解系统使用的用例。
7，其它产品需求
7.1 可应用标准 列出产品必须符合的标准。 **7.2 系统需求** 定义为支持应用所必需的所有系统需求，例如操作系统、网络性能等。 **7.3 许可证、安全和安装** 描述任何可能影响开发工作量，或可能产生独立安装软件需求的任何许可证、安全或安装的需求。 **7.4 性能需求** 用这一段细化性能需求
8，建档需求
这一部分描述所有为支持成功部署系统所需要开发的文档。 **8.1 用户手册** 描述用户手册的目标和内容 **8.2 在线帮助** 列出在线帮助、工具使用等等的需求。 **8.3 安装指南、配置和自述文件** **8.4 标记和打包**
9，词汇表

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

- 可以更多关注系统特性以及它如何体现用户需要，以便更好的理解系统的形状和形式。
- 可以对系统的完整性、一致性及其对环境的适应性进行评估。
- 在继续大量投入之前，可以利用这些信息决定可行性并管理系统的范围。
- 便于在选择迭代内容的时候，能够有一个整体的图像和便捷的用户描述表格。

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

2.2 架构层面的用例方法

一、用例的完整概念

在架构分析与设计中，最普遍使用的工具就是用例方法。用例是什么？用例是代表系统中各个项目相关人员就系统的行为达成的契约，用例描述了在不同条件下，系统对某一参与者的请求所作出的响应，参与者通过发起与系统的一次交互来实现某个目标。用例可以被用来记录需求，也可以描述业务过程，或者用来记录一个软件的行为需求。

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

要编写好一个用例，必须掌握三个概念：

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

需要注意的问题是，根据目标的不同，用例可以在不同的层次上描述，例如：

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

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

二、用例是规范行为的契约

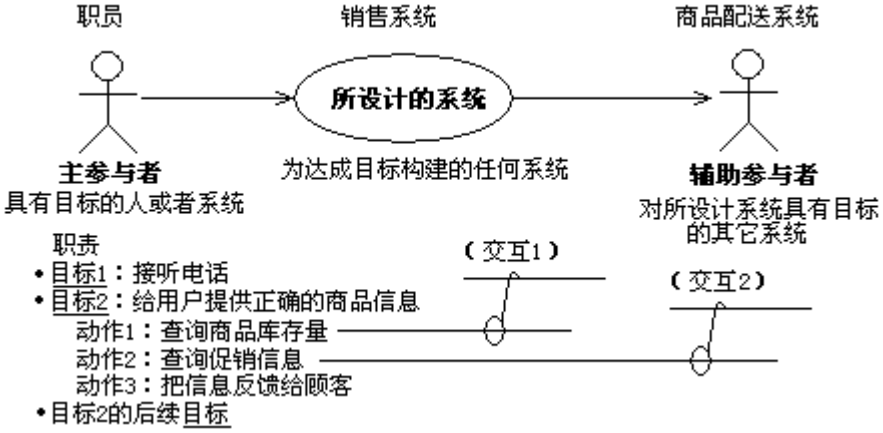
为了正确的开发系统，项目相关人员例如用户、需求分析人员、架构设计人员、编码人员、测试人员需要一个统一的契约，用例构成了契约中的行为部分。用例中的每个句子都有其价值，即它们各自描述了一项行为，该行为维护了或者增进了某些项目相关人员的利益。

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

1，具有目标的执行者之间交互

1）执行者具有目标

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



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

个子目标也就是辅助参与者必须履行的承诺。

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

2）目标可能失败

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

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

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

3）交互是复合的

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

a、查询商品库存量。

b、查询促销信息。

c、把信息反馈给顾客。

d，获取顾客要求。

在更高的抽象层次上，也可以把这个活动序列进行压缩，把它作为一个单独的步骤：

a，查询商品状态，获取顾客要求。

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

重要的是，通过对目标交互进行折叠，可以在一个很高的层次上表示系统行为，而通过把每个高层交互一点点地展开，也可以精细的刻划系统。

2，具有利益的项目相关人员之间的契约

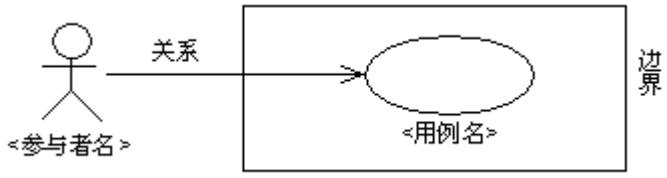
执行者和目标模型解释了如何编写用例中的句子，但没有涉及如何描述行为方面的内容。出于这个原因，需要对用例基于“基于项目相关人员之间的契约”这个观点进一步扩展。

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

3，用例图与文档

1）UML 中的用例图

在 UML 中利用了几个符号来表达用例中的元素，如下图所示。



参与者（actor）：具有行为能力的事务，可以是个人（由其扮演的角色来识别），计算机

系统，或者组织，在 UML 使用一个小人来表达。

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

用例：用例是关于单个参与者在与系统对话中所执行的处理行为的陈述序列。它表达了系统的**功能**和所提供的**服务**，在 UML 中，用例使用一个椭圆来表达。

2）用例图还是文档

许多人在编写用例的时候把精力集中在小人和椭圆上，却忽略了**文本**这个用例的最基本形式。不管怎么说，现在的情况是，很多人误以为椭圆就是用例，即使它根本没有传递什么信息。本课程多次在不同场合表达了这个观点，图比较适合表达关系，事无巨细用 200 个椭圆描绘一个开发需求，和什么都没做其实没有区别。

3）场景与用例事件流

场景（scenario）：是参与者和被讨论系统之间一系列特定的活动和交互，通常被称之为“用例的实例”。通俗地讲，场景实际上是在说故事。一般来说，一个用例就是描述参与者使用系统达成目标的时候一组相关的成功场景和失败场景的集合。

用例的一个场景（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，这就是事件流。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。

4，编写用例的简单步骤

在前期规划系统的时候，主要的着眼点还是放在建立体系的层面。尽管并不存在一个完美的过程来开发这个模型，但还是可以推荐一个简单而有效的步骤。

1）第一步：确定和描述参与者

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

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

2）第二步：确定用例，写一个简短的描述

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

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

要仔细思考用例的名字，通常用例是一个短语，以一个动词开始，说明参与者拿用例做了什么。有了名字，还需要提供一个简要地描述，概要的说明这个用例的工作。

3）第三步：确定参与者和用例的关系

尽管我们注意到只有一个参与者能够发起一个用例，但很多用例可能有多个参与者参与。在过程的这一步，要分析每个用例，看看有哪些参与者与它交互？审查每个参与者预期的行为，以验证参与者是否参与了所有必要的用例，是否达到了想要的结果。这个过程可能

非常复杂，需要团队成员一起来讨论，很快就会有大量的用例和参与者，通过用例图，可以使每个人都很好的理解系统。

4）第四步：简略描述单个用例

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

基本流：

- 什么事件发起该用例？
- 用例如何结束？
- 用例如何重复某些行为？

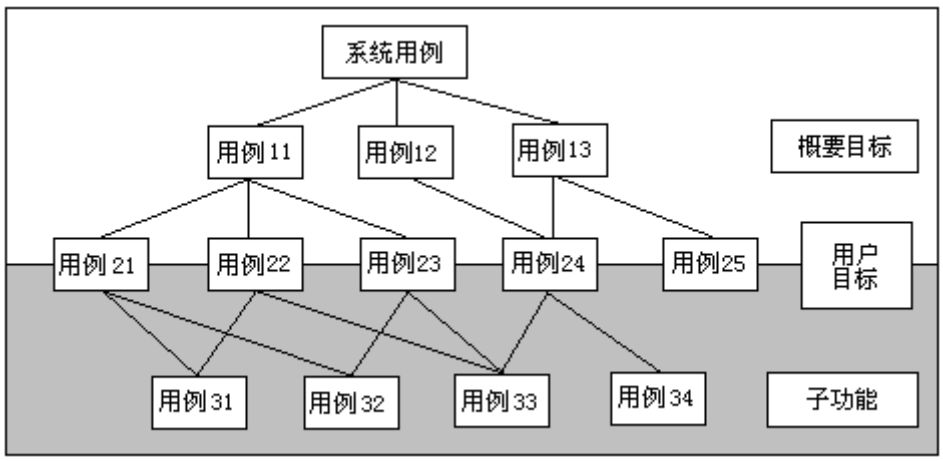
扩展流：

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

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

三、用例的目标层次

在架构分析的时候，我们一般会提出一个问题，我们的用例应该描述哪个层次上的目标呢？首先，我们需要给不同的目标命名，在下面的图中，水平线作为分界线，这条线上的目标我们称之为“用户目标”，用户目标之上的我们称之为“概要目标”，这条线之下的我们称之为“子功能”，如下图所示。



1，概要层次

概要层次包含多个概要目标，每个概要目标包含多个用户目标。主要在系统层次描述，包括如下各方面的功能：

- 显示用户目标运行的语境；
- 显示相关目标生命周期的顺序；
- 表达子系统和功能块黑盒层面的状态与行为。

- 为底层用例提供一个目录表；

对于系统层次的用例，我们可以参照下面的过程来进行：

1）以一个用户目标作为开始

2）考虑目标所提供的服务

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

3）定义与 A 相关的一个设计范围 S

保证 A 在 S 之外，并且给 S 命名。一般这样的设计范围有三种类型：

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

4）找出 A 在设计范围 S 中的所有用户目标

5）找出 A 对系统 S 具有的概要目标 G

6）编写概要级别的用例

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

7）用架构用例表达子系统和主要构件的行为

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

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

2，用户目标层次

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

1）使用用户描述反映初始需求

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

2）用户描述

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

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

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

3）用户描述和主题

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

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

4）确定主题的优先级

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

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

5）用户级别的用例

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

3，子功能目标层次

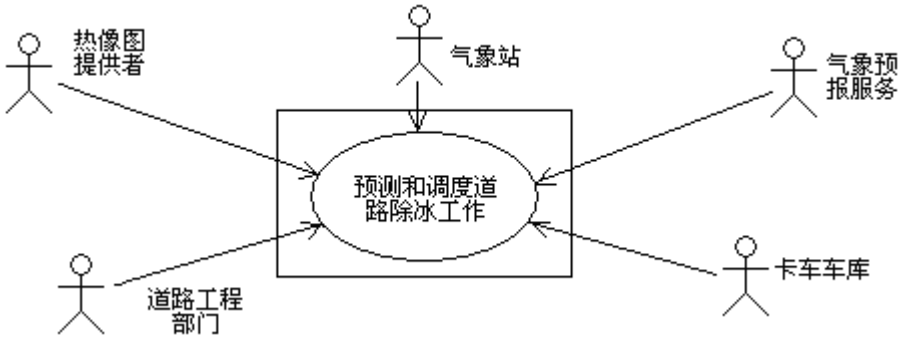
子功能层次的目标指的是那些实现用户目标时候可能会被用到的目标，只有当迫不得已的时候才会包含这些子功能用例，例如可读性方面的要求，或者有许多其它目标用到它们。

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

2.3 架构层面的需求分析

一、业务用例的分析

不论是预测性过程还是敏捷过程，需求都是从了解业务开始的，并且在建立业务模型的过程中发现和挖掘需求。建立业务模型首先是建立业务上下文图，也就是业务范围看成一个黑箱，表示所有相邻系统与业务范围的数据交互关系。例如我们需要了解一个“道路除冰工作系统”，我们可以把上下文图绘制如下。



我们知道，在任何情况下，图形都没有办法表达细节，但更容易表达关系。为了更精确的描述外部系统与系统之间数据流动关系，需要通过表格仔细列出每一个参与者与系统之间的数据流动，下表列出了这样的数据流清单。

道路除冰系统数据流清单			
编号	**参与者**	**输入数据流**	**输出数据流**
1	气象站	气象站读数	
2	气象预报服务	区域气象报告	
3	热像图提供者	热像图	
4	道路工程部门	改变的道路	
		新的气象站	
		改变的气象站	
			失效的气象站告警
5	卡车车库	卡车改变	修订的除冰调度计划
			道路除冰调度计划
		已处理的道路	
		卡车故障	修订的除冰调度计划
			对没有处理的道路进行提醒

然后，我们通过研究相邻系统与业务的数据交互，发现业务事件，然后列出一个业务事件清单，下表列出了这样的事件清单。

道路除冰系统事件清单			
编号	**事件名称**	**输入数据流**	**输出数据流**
1	气象站传送数据	气象站读数	
2	气象局预报天气	区域气象报告	
3	道路工程师通知改变的道路	改变的道路	
4	道路工程师安装了新的气象站	新的气象站	
5	道路工程师改变了气象站	改变的气象站	
6	到了测试气象站的时间		失效的气象站告警
7	卡车车库改变了卡车	卡车改变	修订的除冰调度计划
8	到了检查结冰道路的时间		道路除冰调度计划
9	卡车处理了一条道路	已处理的道路	
10	卡车车库报告卡车出问题	卡车故障	修订的除冰调度计划
11	到了监控道路除冰的时间		对没有处理的道路进行提醒

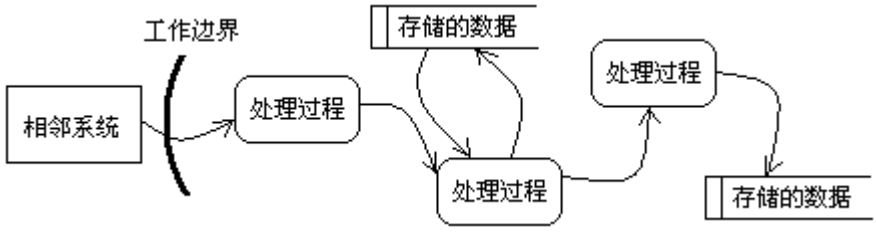
针对每个业务事件，有一个预先计划的对它的响应，被称之为“业务用例”。业务用例总是包含着一些可识别的过程，一些被存储的数据，产生一些输出，发送一些消息，或者是这些事件的组合。也就是说业务用例是一个功能单元，这些功能是编写功能需求与非功能需求的基础。

业务用例是方便研究的组合，可以把业务用例的工作相互隔离开来，因为它与其它业务基本上没有联系，因此不同的分析师可以调研不同的部分，不需要彼此之间一直保持沟通。实际上业务用例之间唯一的重合是它们之间存储的数据。

每个业务用例的相互隔离意味着可以找到一个或者多个这方面工作的专家，他们在您的帮助下可以准确而且详尽的描述这部分工作。可以描述正常情况（计划中的状态），也可以

描述异常情况（偏离计划的状态）及其处理办法。它们可以描述组织是如何相应业务事件的。

一个业务用例的处理应该是连续的，也就是说一个业务用例被触发以后，它将处理所有的事情，直到从逻辑上说无事可做为止（所有的功能都被执行、所有该存储的数据都已经存储、所有的相邻系统都已经得到通知），下图就是一个处理的例子。



二、产品边界的确定

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？

产品分析很重要的一项任务，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。

思考“产品应该是怎样的？”的直接结果，是最终确定产品的边界。在与风险承担者也进行了深入的交流以后，我们需要最终确定当前产品的边界并把它固定下来，否则产品将无法进入真正的设计，另一方面，最后确定的边界，也是作为下一步进行产品建模的必要输入。

在确定边界的时候，我们需要进行一些创新的思考，例如：从顾客的角度，能不能使这个事情变得更方便呢？从方便和服务的角度来说（站在顾客的角度），还有没有更好的想法呢？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），某种想法是不更好呢？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。

三、业务用例与产品用例

我们已经强调了理解工作，而不是理解产品的重要性。通过查看更大范围的工作，可以对业务需求提出更多的问题并且构建出更好的产品。研究业务用例，我们主要考虑的是工作要做哪些事情，相邻系统的期望和预期的结果。而产品用例，指的是建议的自动化产品所需要作出的响应。

在确定产品用例的时候，也是在选择与产品交互的参与者，精良的产品用例分析可以给后期工作带来巨大的好处，事实上引发了基于用例的开发这样一种概念。除了上面列出这一些，还可以在很多地方享受产品用例带来的好处：

- 它提供了一种手段，用于发现一些相关的需求并进行分组。
- 可以通过产品用例来计划实现版本。
- 测试人员可以把产品用例作为编写测试用例的输入信息。
- 产品用例为构建模拟原型提供了业务上的基础。
- 能够更早的响应变更，因为产品用例可以追踪到业务用例，然后追踪到业务事件。

在前期分析中，主要是在抽象层面构造用例模型，关注的是产品用例之间的关系，建立一个产品整体的图像，而不是对每个用例进行详细描述。在进入迭代过程之后，才必须对选

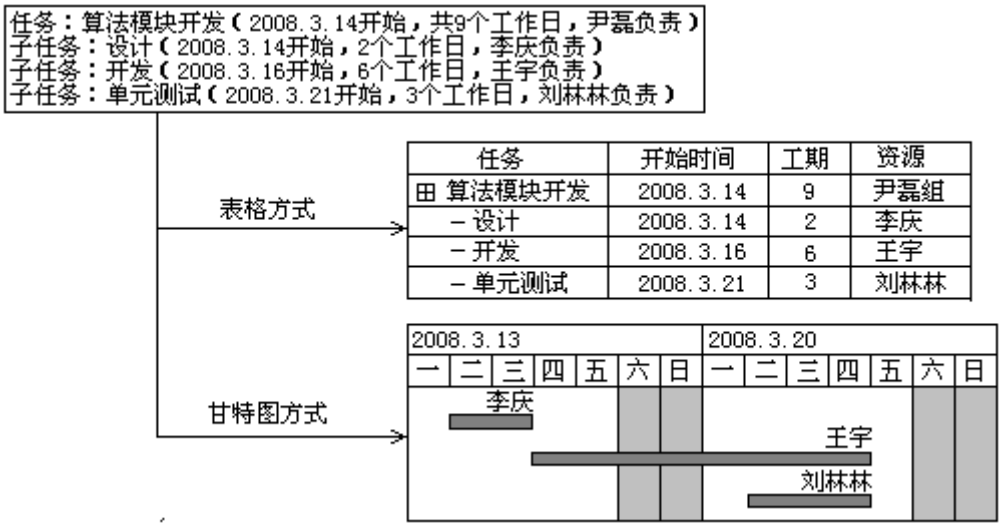
中的用例（或者用户描述）进行详细的分析和描述，从而进入完整的开发过程。

2.4 从问题域到用例模型

下面我们讨论一个简单但意义深刻的案例，以便更深入而细致地研究模型建立的思考过程，特别是研究其中的一些细微思考，这样将会为架构研究带来实感，从理论和实践两个方面讲清楚问题。

一、产品问题域与概念

设想我们需要开发一个项目，基本的问题域是由于各种现行工具不符合组织管理方法的要求，项目管理比较混乱繁杂，希望开发软件帮助项目人员在现有体制下管理好项目，用户简单的提出要求：“能够以甘特图的方式查看项目的起始时间、结束时间、任务承担者等信息。”对于这样的需求，我们首先给项目起名为 **PMT**（项目管理工具，project management tool）。我们最直观的想法是，项目至少应该有两种查看任务计划的方式，一种是表格，另一种是甘特图。

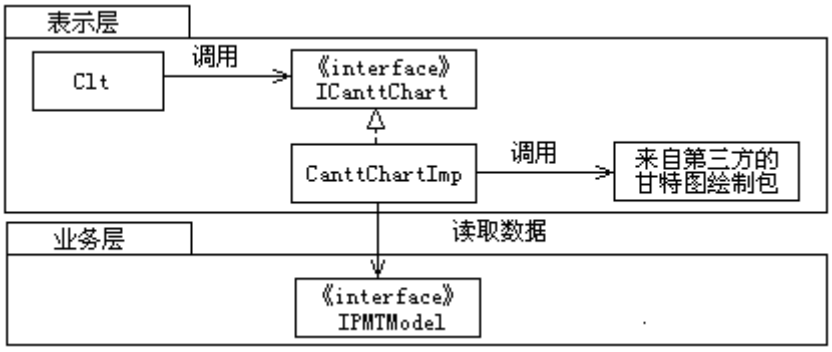


任务是如何计划的？是如何分配给项目成员的？这些东西与 PMT 如何与用户交互应该是没有关系的，根据这个分析，我们立刻想到采用 MVC 架构，把业务逻辑和表示逻辑分开。

在初期的构思中，有些粗粒度的技术问题也可以加以考虑，例如，“甘特图绘制”是一个技术问题，是自行开发还是采用现成的组件？

- 现在项目工期很紧，为什么不采用现成的甘特图绘制包？
- 短期决定采用第三方工具包，性能上和功能上可能不是最优的，所以并不希望 PMT 与工具包“绑死”。

基于上面的分析，架构师决定，采用第三方工具包，但是自主定义“甘特图绘制接口”，使系统与工具包隔离，这就是后面我们将会谈到的 Adapter 设计模式，如下图所示。



在这样的思考中，已经初步体现了**组成**与**决策**两个最基本的概念。

二、产品需求分析

1，项目启动：确定项目愿景

一个项目要被开发，要拨款立项，一定有它的业务目标，这就是需要有一个愿景。在愿景文档中，业务目标占有非常重要的地位。本项目的业务目标如下表所示，最终目标是提高软件项目开发的投入产出比。

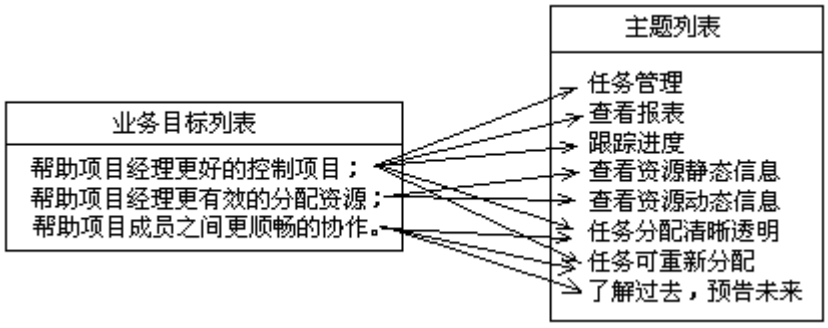
业务目标列表
● 帮助项目经理更好的控制项目； ● 帮助项目经理更有效的分配资源； ● 帮助项目成员之间更顺畅的协作。

2，从业务目标到主题列表

业务目标是客户方高层对于未来系统的展望，最终要落实到使用这套系统的人（最终用户）在实际操作中所需要的功能，这些功能被称之为“用户需求”。如果把业务目标直接向用户需求过渡，我们会发现这中间“跨度”过大，所以可以借助主题列表作为中间的跳板。

在我们讨论用户描述（user story）的概念的时候，我们说到了主题（theme）的概念，也就是一组相关的用户描述被结合在一起。在初期，我们可以以主题为单位考虑问题，在粗粒度范围内，考虑未来系统在大方向上应该具有哪些方面的特性，每个主题会有多个功能来支撑。

下图是从业务目标向主题过渡的示意图，其中，静态信息是指如职位、技能、特长等。动态信息是指如当前所承担的工作量等。

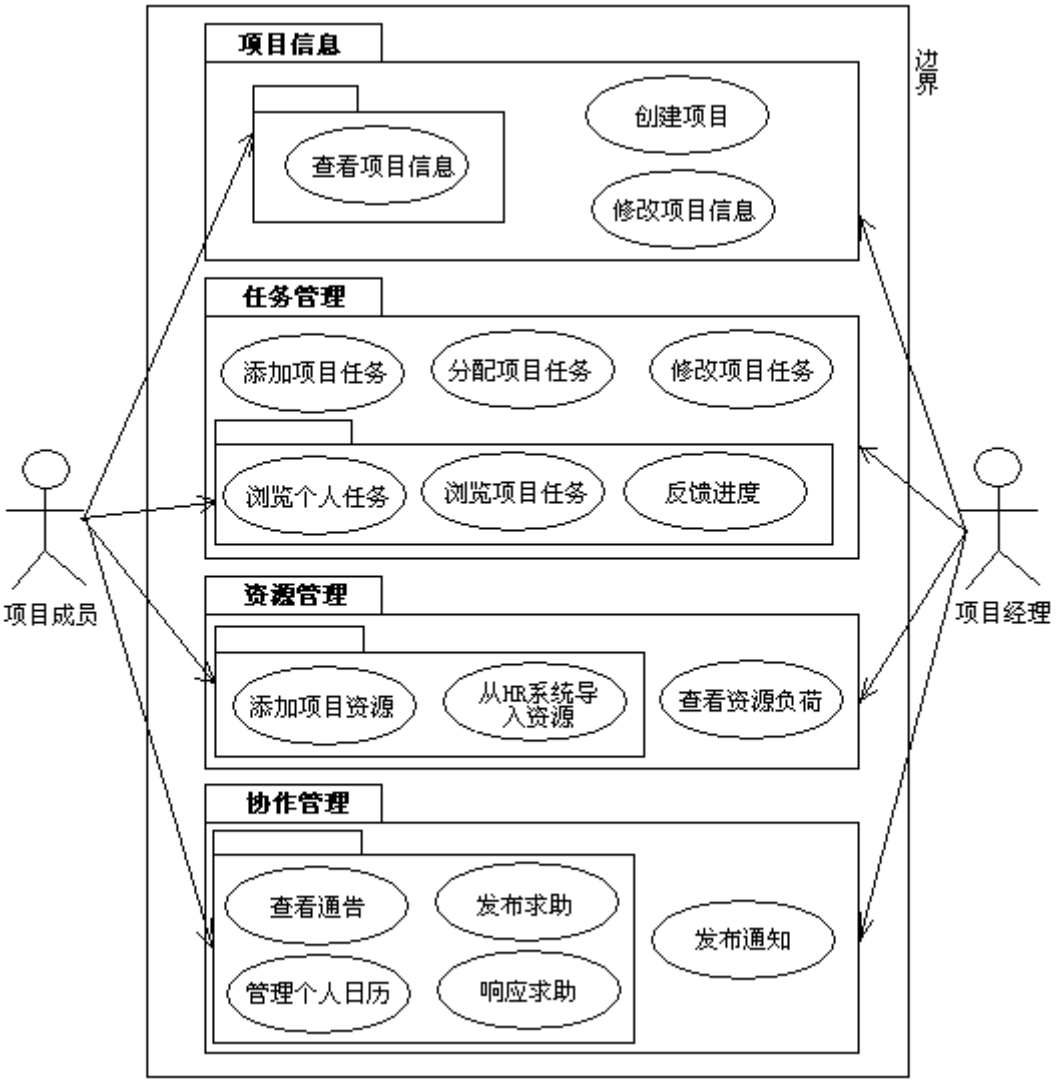


3，从主题列表到用例模型

所谓需求，就是用户需要软件做什么，把主题进一步分解，就可以列出一系列的用户描述（user story），一个用户描述的简要表达，一般都可以引申为一个用例（user story），通过

用例图把用例和参与者之间的关系，以及它们之间的关系表达清楚。

一个比较好的方法，是采用一张表列出“用户描述”表达每个用例（这里没有列出），然后使用用例图来表达关系，使用包来表达主题，如下图所示。



4，通过原型挖掘需求

在需求分析中与客户交流的过程中，困难的是客户难以把潜在需求描述清楚，造成后期需求不断变更。一个比较好的方法，就是在分析期间就开始界面设计，并试着把界面设计的草图用于和用户的交流中去，这就是一种原型方式，其好处是：增加实感，方便交流、帮助客户“发现”他们真正想要的东西。当然，这种原型界面并不需要放入需求文档，因为这是设计而不是需求。

为什么界面原型更为有用呢？分析一下主要有如下原因：

- 不论产品做什么，用户都是通过界面与产品打交道的；
- 产品的工作主要依赖于外部事件，界面是发起外部事件最多的部分；
- 界面是一种形象的图，通过图上的模拟操作，可以成为一种索引，以引发更深入地思考。
- 有了界面原型就避免了凭空想象，而凭空想象往往是更困难的。

在很多情况下，原型并不仅仅是纸质的，可能会有某些功能可演示的原型，不过一定要说明，原型的使用主要为了挖掘需求，从而加深用户的参与度，并不是最终产品的展示，也

要避免给用户带来某些不必要的误会。

5，软件需求规格说明

需求的最终成果是《需求规格说明》，这种规格说明实际上有多种形式，但必须表达清楚功能需求、非功能需求以及约束条件（也有把约束条件归于非功能需求的）。对于功能需求而言，除了重量级的描述功能需求列表以外，还可以用用例图、用户描述、用例场景的文档作为功能需求部分。非功能需求一般是整体的，所以需要单独列出，下面的简表表达了这个情况。

软件规格说明			
功能需求	非功能需求		
	运行期质量属性	开发期质量属性	约束
参见： ● 用例图 ● 用户描述 ● 用例场景	易用性； （测试条件） 互操作性； （可与 HR 系统、财务系统、软件配置管理系统、合作单位的 PM 系统等互操作） 跨平台运行； （考虑不同的 OS 和 DBMS）	可扩展性； （测试点）	客户群平台多样化； 应考虑外包趋势； 和其它系统交互数据。

注意：质量属性会严重影响设计与成本，所以提出质量属性应该按照如下三层次描述：

质量属性：易用性。

原因：如果 PMT 很难使用，反而增加了项目组的工作负担，不利于提高效率。

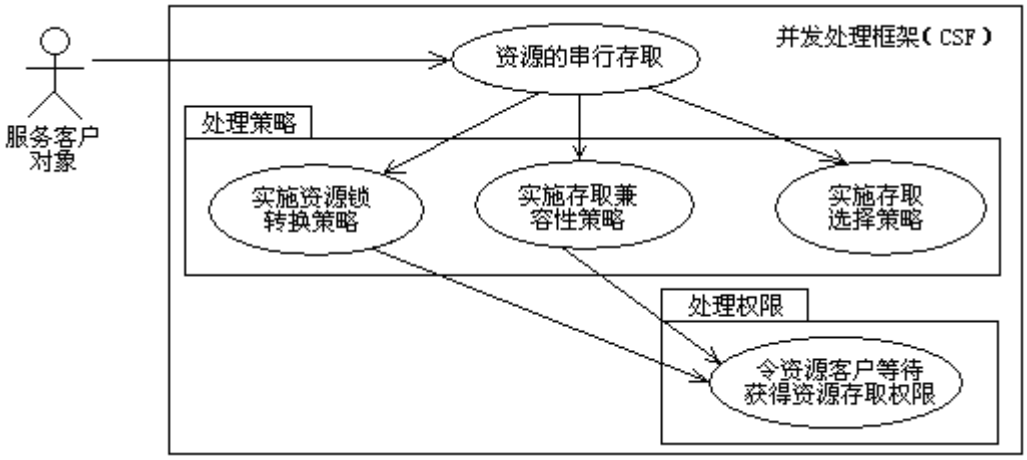
测试条件：在开发人员群体中，随机抽取 20 人，经过 20 分钟熟悉时间，80%的人能够在 5 分钟内处理一个功能。

三、架构层次的用例文档编写

为了能够实现架构设计，概要层次的用例可以帮助我们建立业务架构概念，利用概要层次的用例集中精力考虑了系统大的关系，专注于子系统和功能块“黑盒”层面的状态与行为，而有意的忽视了本身细节层面的内容，这在很多情况下是架构设计所需要的。但从另一方面来讲，作为架构层面的需求分析，很多内容又需要从细节层面考虑，特别是各个子系统和功能块之间交互行为的描述应该细致翔实，从这个范围来说，很多架构层面的需求分析又属于用户层面，因为这才有可能构建一个初始的架构基线。

正是基于这样两个方面的考虑，我们需要把架构层面的一些行为以可实现方式，比较详细地描述出来。这样一来，架构的用例文档编写就显得十分重要。这类文档的编写既要描述某些“黑箱”行为，更需要专注于用例之间行为关系的详细分析。

假定我们需要设计一个软件配置管理工具系统，其中有一个并发服务框架（Concurrency Service Framework，CSF）子系统，文件资源串行存取用例存在于这个框架之中，为了建立架构层面的用例，首先画出用例图结构如下。



该结构的目标是：为了确保服务客户处理文件最新版本的时候，不至于受到并发修改的困扰，客户对象使用 CSF 来保护代码中的临界区域，免受因为多线程而导致的非安全存取。作为框架设计必需的信息，用例中还列出了技术或数据的可变情况。

用例名:	资源的串行存取	**用例层次**
用例 ID:	C-1	用户目标
主要参与者:	服务客户对象	
范围:	并发服务框架（CSF）	
主事件流:	1，服务客户请求资源锁给予它特殊的资源存取权。 2，资源锁把控制权返回给服务客户以使它能够使用资源。 3，服务客户使用资源。 4，服务客户通知资源锁它对资源的使用已经完成。 5，服务客户结束后，资源锁处理善后事宜。	
扩展流:	2a，资源锁发现发现客户对资源已经具有存取权： 2a1，资源锁对请求实施资源锁转换策略（CS-1）。 2b，资源锁发现资源已经分配给他人使用： 2b1，资源锁实施兼容性策略（CS-2），使客户对资源也享有存取权。 2c，资源锁定的持续时间限制不为 0： 2c1，资源锁启动锁定计时器。 3a，在客户通知资源锁完成对资源的使用之前，锁定计时器计数时间到： 3a1，资源锁向客户进程发送一个异常信号。 3a2，失败！ 4a，资源锁发现服务客户上的所计数器非 0： 4a1，资源锁减少请求的引用个数。 4a2，成功！ 5a，资源锁发现资源当前未被使用： 5a1，资源锁实施存取选择策略（CS-3），给处于挂起状态的服务客户赋予存取权。 5b，锁定计时器仍然在运行： 5b1，资源锁取消锁定计时器的运行。	
技术和数据可变列表:	1，特殊的存取请求可以是： ● 独占存取 ● 共享存取 2c，资源锁的锁定超时时间可以通过以下因素指定： ● 服务客户 ● 资源锁定政策 ● 全局缺省值	

用例名:	实施资源锁转换策略	**用例层次**
用例 ID:	CS-1	子功能
主要参与者:	客户对象	

范围:	并发服务框架（CSF）：处理策略
主事件流:	1，资源锁验证请求的存取方式是独占存取。 2，资源锁验证服务客户已经具有共享存取权限。 3，资源锁验证没有服务客户等待更新其存取权限。 4，资源锁验证没有其他的服务客户在共享资源。 5，资源锁赋予服务客户对资源的独占存取权。 6，资源锁增加服务客户锁的个数。
扩展流:	1a，资源锁发现请求的存取方式是共享存取： 1a1，资源锁增加服务客户锁的个数。 1a2，成功！ 2a，资源锁发现服务客户已经具有共享存取权限： 2a1，资源锁增加服务客户锁的个数。 2a2，成功！ 3a，资源锁发现有另一个服务客户等待更新其存取权限： 3a1，通知服务客户不能赋予它所请求的存取权限。 3a2，失败！ 4a，资源锁发现有另一个服务客户正在使用资源： 4a1，资源锁令服务客户服务等待获得资源存取权限（CS-4）

用例名:	实施存取兼容性策略	**用例层次**
用例 ID:	CS-2	子功能
主要参与者:	服务客户对象	
范围:	并发服务框架（CSF）：处理策略	
主事件流:	1，资源锁验证请求的存取方式是共享存取。 2，资源锁验证当前对资源的所有使用都是共享存取。	
扩展流:	2a，资源锁发现请求的存取方式是独占存取： 2a1，资源锁令服务客户服务等待获得资源存取权限（CS-4） （进程的执行由锁服务策略恢复） 2b, 资源锁发现资源正在被以独占方式使用： 2b1，资源锁令服务客户服务等待获得资源存取权限（CS-4）	
可变情况:	1，兼容性准则可以改变。	

用例名:	实施存取选择策略	**用例层次**
用例 ID:	CS-3	子功能
主要参与者:	客户对象	
范围:	并发服务框架（CSF）：处理策略	
主事件流:	语境目标：资源锁必须决定哪一个等待请求（如果有的话）应该获得服务。 注意：这个策略是一个可变点。 1，资源锁选择等待时间最长的请求。 2，资源锁将存取权限赋予被选中请求，途经是使其进程进入可运行状态。	
扩展流:	1a，资源锁发现没有处于等待的请求： 1a1，成功！ 1b，资源锁发现有一个请求正等待从共享存取更新为独占存取： 1b1，资源锁选中这个等待更新的请求。 1c，资源锁选中一个请求共享存取的请求： 1c1，资源锁重复第一步，直到遇到一个独占请求。	
可变情况:	1，选择顺序准则可以改变。	

用例名:	令服务客户服务等待获得资源存取权限	**用例层次**
用例 ID:	CS-4	子功能
主要参与者:	客户对象	
范围:	并发服务框架（CSF）：处理权限	
主事件流:	1，资源锁将服务客户进程挂起。 2，服务客户等待直到再次被恢复运行。 3，服务客户进程被恢复运行。	

扩展流：	1a，资源锁发现一个等待超时已被指定： 1a1，资源锁启动计时器。 2a，等待计时器时间到： 2a1，通知服务客户其请求的存取权限不能得到。 2a2，失败！
技术和数据可变列表：	1a1，锁的等待超时时间可以通过下列因素指定： ● 服务客户 ● 资源锁定策略 ● 全局缺省值

2.5 从系统工程的角度分析与设计架构

一、应用系统工程帮助分析问题

在大多数需求分析的资料中，都提出来在整个需求分析的过程中，只是关注收集用户需要，而不要考虑解决方案。在某种意义上，这对防止初期的解决方案对后期设计的影响是有一定的意义的。但是，从另一个方面来说，把这个观念绝对化也是有害的。从架构设计的角度来看，软件架构的要求也是一种需求，可能也是第一阶段收集需求的时候最值得关注的需求。这就需要在需求分析的时候考虑到整体架构的因素，反过来这也会使需求过程更加趋于合理。

在多维度敏捷开发的模型下，第一阶段各个小组并没有成立，整体性的需求收集和架构小组是这一阶段最重要的团队，在这一阶段，团队成员综合各自的知识，从系统的角度分析和解决问题，往往更容易导致设计出一个高质量的产品，这就需要利用系统工程来帮助分析问题。

根据国际系统工程委员会（International Council on Systems Engineering，INCOSE）的说法：系统工程是一种交叉学科的方法，这种方法主要用于开发周期的早期侧重于定义客户的需要及所需的功能，并且为需求建挡，而后进行设计合成与系统确认，同时考虑所有方面的问题，包括：操作、性能、测试、制造、成本和进度、培训与支持以及配置。

系统工程把所有分支和专业小组集成一个团队，形成一个结构化开发过程，形成从概念到产品的到操作的转化，系统工程同时考虑客户的商业和技术需要，目标是生产出满足客户需要的高质量产品。

这段定义尽管比较长，但我们确实可以把系统工程看成一种问题分析技术，我们可以把这种技术应用在特定的环境中，帮助我们理解在系统中运行的应用软件之上的需求。

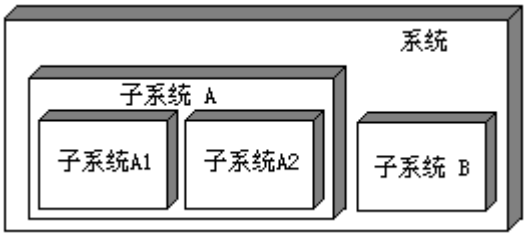
二、子系统、框架与软件架构

有两个系统级的概念是与架构的基本概念相伴而生的，一个是子系统，另一个是框架。

1，子系统与架构

子系统是伴随着软件复杂性的增长而日渐重要的一个概念，当软件规模越来越大，所有的软件系统都会划分为模块或者子系统进行开发。

任何复杂系统，都可以分解成小问题也就是子系统，每个子系统都可以合理的解释和证明、成功的设计与制造，然后在集成到整个系统中去。子系统还可以分解成子系统，这种分解（或逐步细化）过程会一直进行下去，如下图所示，例如 F22 战斗机被分解成 152 个子系统。



在下面的情况发生的时候，可以断定子系统的分解已经完成并且是正确的。

- 对功能进行分布和划分，对于以最小的成本和最大的灵活性来完成系统整体的功能来说是最优的。
- 每个子系统都可以被一个小型团队所定义、设计与开发。
- 每个子系统都可以使用可行的制造技术制造出来。
- 在成功地模拟了子系统的其它子系统的接口之后，每个子系统都可以单独进行可靠性测试。
- 对于子系统的各个物理方面（大小、重量、位置、分布）都要进行考虑，并在整个系统环境下加以优化。

子系统本身的开发也需要经过架构设计这一关。从高层来说，架构的重心主要是子系统之间的协作，从子系统来说，架构的重心主要是模块之间的协作。如果从粒度的角度来看问题：

- 粒度最小的单元是“类”。
- 几个类紧密协作形成“模块”。
- 完成相对独立功能的多个模块形成“子系统”。
- 多个子系统相互配合满足一个完整的需求，从而构成软件“系统”。
- 一个大型企业往往使用多套系统，多套系统通过互操作形成“集成系统”。

从子系统的层次结构来说：

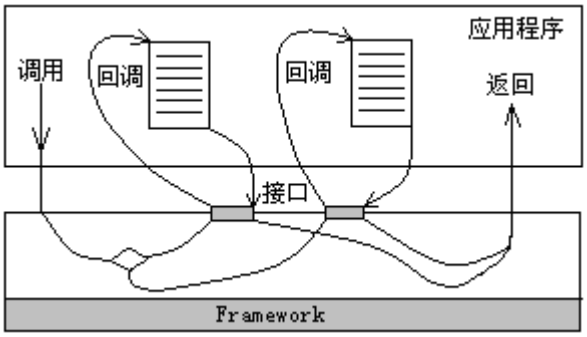
- 当进行高一层架构设计的时候，子系统成为一个原子单元、一个黑盒。
- 当进行子系统本身设计的时候，它才变成了一个结构复杂的白盒。
- 第三方组件可以看成一个原子单元，它是一个黑盒，我们主要应用它的服务。
- 当使用第三方框架的时候，需要仔细研究它的结构，这是因为这是使用的是它的结构，而不是它的服务。

2，框架与架构

框架是可以通过某种回调机制进行扩展的软件系统或者子系统。框架的概念主要来自于对“重用概率”的分析。一个软件单元被重用，单元粒度越大，重用概率越低，但是重用价值越大。反之，单元粒度越小，重用概率越高，但是重用价值越小。这个矛盾，仅仅通过分析是解决不了的。框架的智慧在于，在单元粒度比较大的情况下，追求高的重用概率。

人们对于架构（Architecture）和框架（Framework）实际上还存在很多混淆，认为架构就是框架。其实一句话就可以区别出来，框架是一个软件，但架构不是软件。

框架是一种特殊的软件，它不能提供完整无缺的解决方案，但为解决方案提供和很好的基础，它是一种系统或者子系统的半成品，框架中的服务可以提供最终应用系统的直接调用，而框架中的扩展点题共有开发人员定制的“可变化点”。

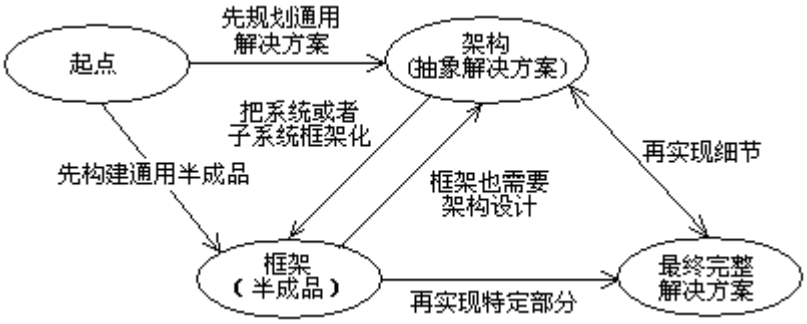


这样一来，一个比较大的框架单元，就可能有比较大的重用概率。

软件架构不是软件，而是关于软件如何设计的重要决策，软件架构解决的问题，是关于软件有几个部分，各部分静态关系与动态关系是如何交互的，经过完整的开发过程以后，这些决策将体现在最终系统中。在引入软件框架以后，软件架构决策往往会体现在框架设计之中。

软件架构是比具体代码更高一个抽象层次的概念，架构必须被代码体现和遵循，但任何一段具体代码都代表不了架构。

不论是架构技术还是框架技术，都是为了解决软件日益复杂所带来的困难，而采取的“分而治之”的结果。架构的思维是先大局后局部，这是一种问题在抽象层面地解决方案，首先考虑大局而忽略细节。框架的思维是先通用后专用，这是一种半成品，还需要通过后期的定制才能成为具体的软件。框架与架构设计的关系可以由下图表示。



框架和架构的关系可以总结为两个方面：

- 为了尽早验证架构设计，或者出于支持产品线开发的目的，可以把通用机制甚至整个架构以框架方式实现。
- 企业可能存在大量可重用框架，这些框架可能已经实现了架构所需的重要机制，或者对某个子系统提供了可扩展的半成品，最终软件架构可以借助这些框架来构造。

三、系统工程中的需求分配

从系统工程的角度考虑需求，就需要把子系统作为一个重要的实体加以考虑。首先赋予子系统功能描述（例如：子系统 B 将运行风速算法，并直接驱动前导显示器），然后再自上而下的分解需求。

1）关于派生的需求

在这样的情况下，很可能我们会生成一个派生的需求，它们必须被赋予子系统，典型情况下，派生需求可以分成两种：

- **子系统需求**：是子系统必须满足，但随最终用户未必有直接利益的那种需求（例如：子系统 A 必须计算飞行器的空速）。

- **接口需求**：是子系统为了完成整体任务，必须与另一个子系统进行通信产生的需求，子系统创建的同时也促进了子系统之间接口的创建。

但是这些子系统需求是真的需求吗？对最终用户来说，它可能并不是重要的。但对于需求人员来说，开发人员也是需求提供者的客户，这些需求对于开发人员是重要的，所以也是一种至关重要的需求。

还需要注意的是，虽然这些子系统需求对项目的成功至关重要，但他们是由系统的分解得来的，不同的分解方式会产生不同的派生需求。所以，在前期设计人员参加一些讨论还是非常有意义的。重要的是要认识到，对派生需求的描述会影响最终系统完成工作的能力，以及系统的可维护性和稳定性。

2）子系统分包

还有一种更加复杂的问题经常会发生，那就是子系统经常是由不同的团队来开发的，这也是我们生成子系统的目的之一。在这样的情况下，子系统需求与接口，就有可能成为团队之间的契约。更多的情况，子系统是由其它公司作为分包商开发的，这使得需求失去了其系统和技术的环境，改变需求变得很困难，项目会被它的短处所牵制，很多大型项目就是在这个问题上翻的船。

3）解决问题

这就是为什么要专门讨论从系统工程的角度研究需求的原因。我们可以做什么呢？在处理极其复杂的系统的时候，你可能需要考虑以下建议：

- 开发、理解和维护横跨子系统的高层需求和用例。它们描述了系统的整体功能，这些用例提供了系统整体的工作背景，要确保你没有“只见树木，不见森林。”他们还有助于确保系统架构设计支持最可能的使用场景。
- 把划分工作做得最好，并把功能限制在子系统范围内。在系统功能中使用对象技术原则：封装和信息隐蔽。根据合同建立接口，使用消息而不是数据共享。
- 可能的话，把软件作为一个整体来开发，而不是一些分开的部分。避免在接口的两端，为了双方的决策，软件都必须重构核心元素（对象）的状态。与硬件不同，软件需求在双方的分配并不是一个清晰的划分。
- 当对接口进行编码的时候，在接口两端采用共同的代码。否则如果有什么变化（比如优化）的话，两端的同步将会非常困难。另一方面，如果将来两个子系统之间的界限消失，比如系统工程师发现两个子系统可以合并，合并将会非常困难。
- 最后，看看能否找到一个资深工程师来帮助你实施系统工程，如果他以前做过类似的工作，它的经验将会对您有很大的帮助。此外这样做的副产品是，这样做有助于消除代沟。
- 在多团队开发的时候，最好是其中的一个团队来开发接口代码，否则，两个团队有很多工作是重复的。这样你将会确实的生成系统新的需求，包括接口。把接口作为正式的需求，可以避免很多集成上的问题。

四、组织复杂软硬件系统的需求

无论需求是表示成用户描述、特性列表、用例集、或者是其它的形式，都必须获取需求并形成文档。一个由多人参与的工作不可避免的存在交流问题，这就需要获取一份能够被复审和批准，大家都人认可的，用来参考的文档。传统上，利用需求规格说明这种大型文档，来获取和交流这种信息，但是敏捷开发基本理念似乎并不需要这样的大型文档。不过，对于大型复杂的敏捷项目，不可能没有需求规格说明就盲目的开始项目开发，关键是这种规格说明的粒度要合理。

关于如何组织需求信息有几个要点：

- 对于复杂项目，必须获取需求规格说明，并把它记录在文档、数据库或工具中。
- 不同类型的项目，要求不同的需求组织技术。
- 复杂系统需要每个子系统都有需求规格说明。

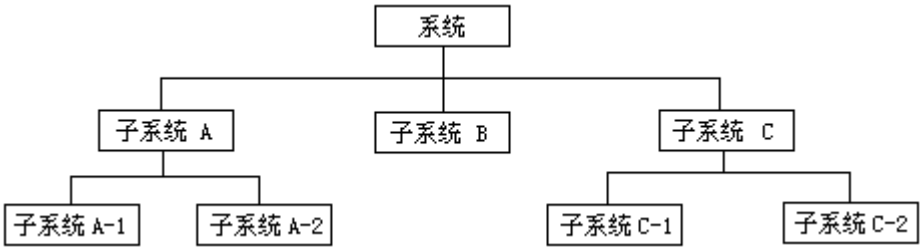
在项目的前期，我们只是宏观的讨论组织需求信息的有关问题。我们必须了解到，很少有需求能够在单个文档中定义，其原因是：

- 系统可能非常复杂，建档数量较大，需要有组织的和交互访问的技术。
- 该系统可能是相关产品系列的一个成员，没有什么文档可以包括所有的规格说明。
- 所构建的系统可能只是一个大型系统的子系统，仅能满足所确定需求的一个子集。
- 必须把市场和商业目标从产品的详细需求分离出来，这需要多个文档。
- 也可能在系统中建立制度或法规这样的需求，这些需求可以在其它地方进行建挡。

在以上例子中，都可能需要维护组成多个需求集的需求，每个需求集反映了一个特殊系统加上若干子系统的集合的需求，下面有一些这样的例子：

- 一个需求集用一般术语定义系统特性，这称之为**前景文档**（Vision Docunment）。而另一个需求集使用更专用的术语定义需求，这由**用例模型**和相关的**补充需求**组成。
- 一个“父”需求集定义整个“系统”的需求，包括硬件、软件、人员和过程，另一个需求集只定义软件子系统的需求。
- 一个需求集定义产品系列的全部需求集，另一个需求集只定义特定应用和特定版本的需求。

下面我们来看一个组织复杂软硬件系统的需求的案例。对于非常复杂的系统，唯一合理的方法是把它创建成由多个子系统组成的系统，这些子系统有时是其它子系统构成的系统。在特殊情况下，比如飞机控制系统，将包括上百个子系统，而每个子系统都有自己的硬件组件和软件。



1）创建系统级的需求规格说明

在这样的情况下，首先创建一个系统级的需求规格说明，在不了解或者不引用任何子系统的情况下来描述系统的功能行为。例如飞机油料的装载能力、爬升速度、飞行最大高度等等。

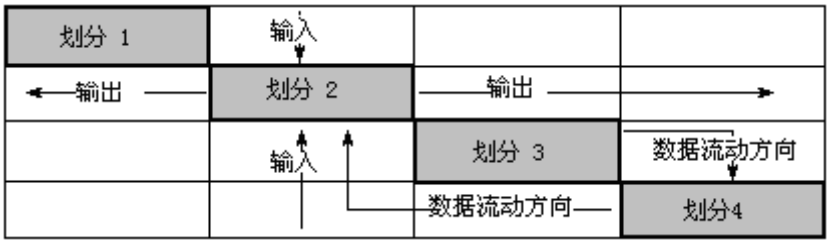
2）进行子系统划分

正如讨论过的，一旦对系统级的需求达成共识，就开始系统工程活动。我们可以把系统分解成多个子系统，描述子系统之间的接口，把系统级的需求分配到各个子系统，由此产生了系统架构描述，定义了系统划分以及系统之间的接口。原则上是可以任意确定划分大小的，一个可能的原则是，每个子系统可以由一个小型团队通过多次迭代来完成。

在系统分解的时候，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 图可以有效地对划分和接口进行评估，它是捕获子系统单元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。



下图是一个飞行控制系统 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。

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	状态数据
大气、地形及空气数据		环境状态数据	环境子系统

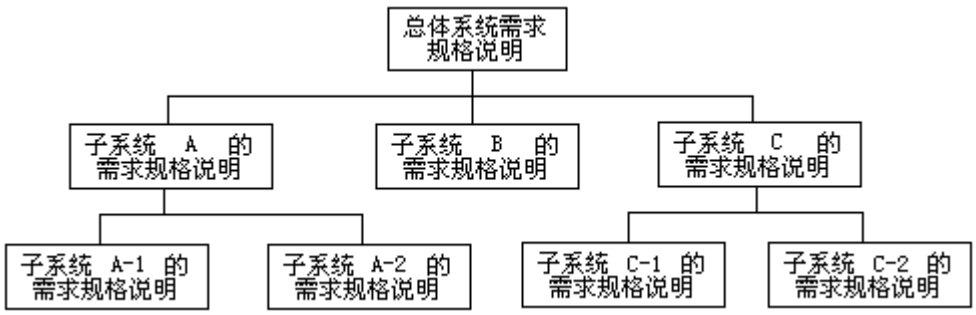
在分析阶段仔细考虑并稳定接口是非常必要的。系统分析师和设计师可以一直这样做下去，直到表示出接口中所有的原始对象为止。初步的考虑可能是混乱的没有条理的，但有了这样的矩阵就可以使自己的思维逐步的条理化，最后达到非常优秀的设计。

3）开发子系统需求规格说明

下一步，为每个子系统开发一个需求规格说明，这些规格说明应该在不引用它自己的子系统的情况下，完整地描述它的外部行为。在这个过程中会产生一类新的需求：派生需求。这类需求不是描述整个系统的外部行为，而是描述子系统的外部行为，因此，系统设计过程为组成系统的子系统创建了新的需求。特别是系统之间的接口成为关键需求。本质上这是子系统与另一个子系统之间的契约，或者是对子系统同意完成的事情的承诺。

4）对子系统进行划分

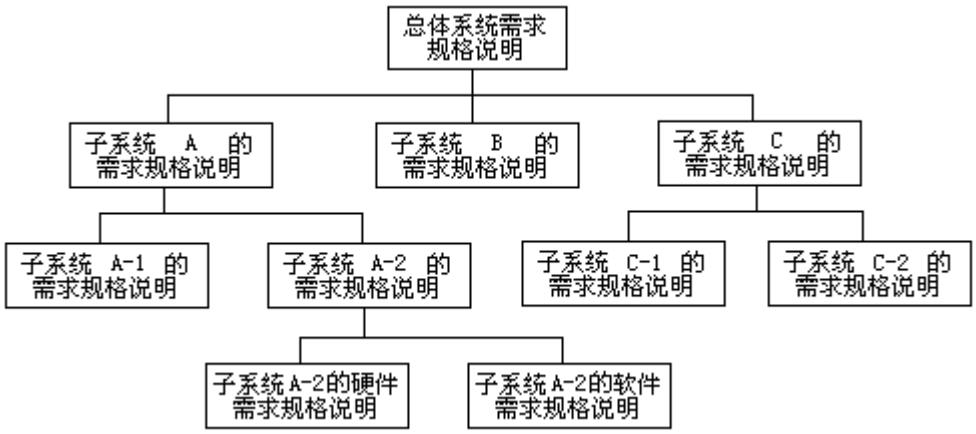
一旦需求达成一致，再次进行系统地分析和设计，如果需要，可以进一步把子系统分解成它自己的子系统，并且分别开发各个子系统的需求规格说明，如下图所示。



在每一层，都把来自上一层的需求规格说明分配给适当的下一级需求规格说明。例如，燃料容量需求分配给燃料控制系统和燃料储备系统，同时恰当的发现和定义新的需求。

5）最底层的需求规格说明

最底层的需求规格说明指的是那些不能再分割的需求规格说明，通常对应于仅仅是硬件或者仅仅是软件的系统，如下图所示。

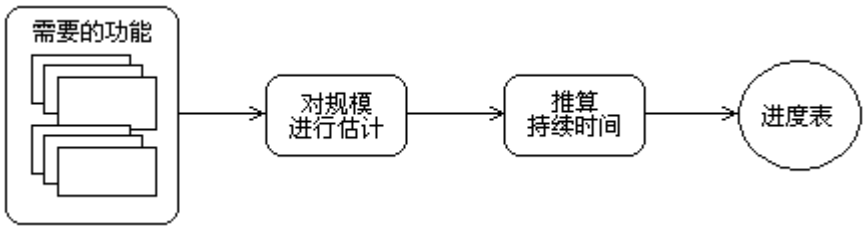


这些需求规格说明在开发进程中还会有一个演化的过程，使细节更容易理解。

2.6 利用规模的估计修正子系统划分

一、规模的估计

在上面的阐述中，如果划分大小的原则是，是每个子系统可以由一个小型团队通过多次迭代来完成，那就需要对分块的规模作出估计。另一方面，要实现好的项目的规划，其必要的输入就是规模估计以及持续时间的估计。一个基本的过程如下图所示。



曾经人们付出极大的努力构建了一些重量级估计方法，例如用功能点方法估计规模，用 CoCoMo 模型估计工作量和持续时间等。这些方法努力从数学的角度揭示它们之间内在的关系，对软件工程学的贡献是很大的。功能点分析法（Function Point Analysis，FPA）是 20 世纪 80 年代由 IBM 的 Albrecht 提出来的，它的意图在于度量值和软件的代码量无关。目前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。但是，这种重量级方法计算比较繁琐，需要的信息比较多，应该说对于经典瀑布式过程的还是比较合适的。但对于敏捷开发这种轻量级过程，由于原始信息本身就比较简约，变动比较大，就需要寻找一些轻量级的估计方法。

1，用描述点估计规模

度量理论的本质是构造映射的基准，通过基准使人们可以从事物的比较中获得对事物的理解。软件规模度量也是利用相同的原理。但是，敏捷方法给规模度量提出了新的难题，首先，用户描述是一种功能的抽象表达，细节信息不太清楚，很难用功能点这些重量级方法来计算。另一方面，敏捷方法提倡小组共同度量，这就更需要一种经验型方法，而不必请专业的估计师来度量。描述点规模度量是解决这个难题的方法之一。

2，描述点是相对的

我们先通过一个例子看看描述点度量的原理。假定我们需要比较 5 个用户描述的规模大小，而我们自己曾经做过类似功能 A 的产品，我们很熟悉它的工作量，我们希望用这个描述做基准，判断其它描述的相对大小。我们定义一个“描述点”作为度量标准，直观的感觉，它的规模应该是中间大小，我们把它的点数定为 5。

仔细的思考和比较，我们就可以给其它的描述给出一个合适的描述点数。功能 C 大约是它的两倍，所以给它 10；功能 E 接近功能 3，但比它稍小，所以给它 9；功能 D 是最小的，所以给它 1；在不知道细节的情况下，有时候需要猜测，比如功能 B 有多种做法，但一般的印象，最大的规模也比功能 A 小，最小的规模也比功能 D 大，这样比较的结果，分配给它 3 是合理的。这样得到的结果如下表所示。

序号	功能	描述点
1	功能 A	5
2	功能 B	3
3	功能 C	10
4	功能 D	1
5	功能 E	9

我们可以看出来，通过基准和比较，我们可以把一个模模糊糊的感觉，用量化的方式表达出来。这里有三个关键：

- 所有参与比较的最好在同一个数量级，比如航空管制系统是多少描述点？这就很难说了，一般来说越接近基准，越容易判断；
- 基准点最好在相对中间的位置，因为如果基准点偏高或者偏低，离开基准点比较远的部分误差就比较大；
- 就是有些情况下可以构造新的参照点，比如张三“身高点”是 5，李四是 8，王五比张三高，但给多少点呢？如果比李四高可以给 10，如果比李四底可以给 7，这样就避免了模糊性。

根据这个原理，我们可以建立“用户描述”的“描述点”来度量每个功能的规模。一方面，由于软件开发对需求可追踪性的要求，一般用例和功能包是对应的，这就对用例的规模要求相对不要差别太大，这恰恰满足了一个数量级的条件。另一方面，当我们有一定的开发经验以后，一个典型的用例工作量到底有多少，还是有一定的数的，这就使“描述点”度量规模成为可能。

描述点是用于表达用户描述、功能或者其他工作的总体规模的度量单位。我们使用描述点进行估计的时候，对每个对象分配一个点值。我们分配的原始点值本身并不重要，重要的是这个值的相对大小，一个分配值是 2 的用户描述工作量是分配值是 1 的用户描述的 2 倍。

建议基准选择一个基本上处于中等的用户描述，然后各他分配一个基本上处于中间的描述点值。这个中间点值可以定为 5，然后可以比较基准描述点，或者与已经分配了点值的用户描述进行比较，来估计其它的用户描述。

研究显示我们对以基准数为核心相对大小这一类东西，同样数量级内的事情估计的准确度比较高。例如，假定在同一个街区，以一个停车场为基准，另一个停车场距离是它的几倍这类估计，一般估计的比较准确，但是要估计纽约的一个停车场距离是它的几倍，那误差就会相当大了。所以，我们希望大多数估计都是在一个数量级的范围内。

推荐的估计尺度是 1、2、3、4、5、8。这称之为斐波纳契数列，它的特点是两另两数之间的差距随着数的增大而增大。这个非线性数列反映了数值越大，估计的不确定性越高，因此非常便于使用。通过主题，开发小组可以减少估计的工作量。但是也要认识到，对主题的估计，不确定性要比更明确、更小的用户描述进行估计更高。

在最近几次迭代要实现的用户描述需要足够小，以便在一次迭代中完成，这是应该在一个数量级范围内估计这些对象，我们使用的序列是 1、2、3、5、8。离最近几次迭代更远的

用户描述可以当成主题，可以用 8 以外的数据来估计它，也就是：1、2、3、5、8、13、20、40、100。

某些情况下，这种描述点定义需要一些猜测，也需要一些工作经验，随着时间的推移，估计会越来越准确。

二、持续时间的估计

1，速度

一个没有单位的描述点估计为什么可能会有效呢？还需要知道一个概念就是“速度（velocity）”。速度是对开发小组进度律的度量，可以通过计算小组在一次迭代中完成的描述点数的总和来得到速度。比如，下组一次迭代完成了 3 个用户描述，每个描述点为 3，那么它的速度就是 15。比如小组在上一次迭代中完成了 10 个描述点，那么这次迭代中，有 2 个 5 点的用户描述，或者有 5 个 2 点的用户描述，都是可以完成的，这个估计应该是正确的。

如果把所有的预期描述点加起来，就会得到项目总的规模估计，用这个数值除以小组一次迭代能够完成的描述点数，就可以得到迭代次数，把这个时间反映在日历上，就可以得到进度表。敏捷规划的一个关键原则就是先估计规模，然后推算出持续时间。

2，利用速度修正估计误差

幸运的是，随着开发小组在项目的用户描述开发上取得进展，他们的速度在最初几次迭代就可以显示出来，使用描述点估计方法的美妙之处，就在于对速度的使用可以自我修正。假定开发小组要开发一个 200 个描述点的项目，他们最初估计一个迭代可以完成 25 点，这样他们认为 8 次迭代就可以了。但是项目开始以后他们发现一次迭代只能完成 20 点，他们立刻就可以判断出项目需要的迭代次数是 10 次而不是 8 次。这种修正早期就可以完成，并不需要等到项目接近尾声的时候仓促的加班，以降低质量换取时间的达到。

由于所有的数据都是无量纲的相对数据，所以要膨胀就整体膨胀，要缩小就整体缩小。用前期的数据外推出后期的结果，这从数学上来说也是合理的。把工作量与持续时间的估计隔离开，允许对它们独立的作估计，只需要通过计算和推算，这个区别很微妙但很重要。

2.7 迭代的建立架构基线

基础架构的建立，应该纳入发布规划和迭代规划，成为整个设计工作的一个环节。架构实现可能需要一个单一团队，架构的功能，也可以用用户描述来说明，但是问题更集中在顶层描述，主要是框架、对外通讯、内部框架之间的通讯，以及大的、集中在主题层面地描述。架构的成果也应该是一个可发布的、经过测试的版本，一个架构一般也需要经过一到两次迭代完成。

一、成功的软件架构设计

为了更好地进行架构设计，我们必须仔细研究一下，一个成功的软件架构设计应该具备什么品质，对这种品质的深入理解，可以使我们的设计目标更加明确而有指向性，研究问题的重点也更加突出。

1，成功软件架构的品质

成功的软件架构设计是高质量的，并且所花费的时间、技术决策等方面都能满足具体开发方法的要求，具体应该有如下品质：

- **良好的模块化**：每个模块职责明确，模块之间松耦合，模块内布高内聚，并且合理的实现了信息隐蔽。
- **适应功能需求变化，适应技术变化**：典型情况是，应该保持具体应用的相关模块和领域通用模块的分离，技术平台相关模块与具体技术的应用模块相分离，从而达到“隔离变化”的效果。
- **对系统动态运行有良好的规划**：应该标识出哪些是主动模块，哪些是被动模块，明确模块之间的调用和加锁机制，并说明关键的进程、线程、队列、消息等机制。
- **对数据的良好规划**：不仅仅包括数据的持久化存储，还包括数据传输、数据复制与数据同步等策略。
- **明确、灵活的部署规划**：这里往往牵涉到可移植性、可伸缩性、持续可用性以及互操作性等大型企业级软件特别关注的质量属性架构策略。

好的架构并不是“好的就是成功的”，而是“适合的才是成功的”。不适当的用时间换完美，最后会毁掉整个项目。回过头来看看我们为什么花了大量的时间讨论软件开发过程及其影响？除了明确我们初期需要在绝大部分技术细节都不清楚的情况下定义软件架构，还要考虑初期我们就需要搭建一个团队协作开发的基础，让不同的小组针对不同的子系统和模块深入下去，这种团队的秉性工作也意味着可能缩短项目工程的周期。

架构师并没有绝对的技术选择自由，还需要考虑经济性、技术复杂性、发展趋势以及团队水平等诸方面的因素。最终，架构师的工作成果就是为整个开发团队的工作提供足够的指导和限制，使他们沿着正确的方向进行下去。

下面，我们来探究一下成功架构设计的关键要素是什么，这些讨论将主导着本课程内容的演绎，是整个课程最核心的知识。

2，成功架构设计的关键要素

软件架构设计是以“需求规格说明书”为最主要的设计依据，首先勾勒出概念性的架构，再结合具体的技术平台制定实际架构方案的。在考虑架构设计的时候，必须注意如下关键要素。

1）是否遗漏了至关重要的非功能性需求

非功能需求是最重要的架构决定因素之一，所谓非功能需求主要包含两个部分：质量属性和约束条件。质量属性是软件系统整体的质量品质，所谓整体品质，就是它往往和大多数功能都有关系。比如易用性、可扩展、安全性、可靠性等。它们往往表现在整体上的品质，而不是哪个功能的“内部”。

非功能需求很多情况下存在一定的相互矛盾，所以只有少数几个质量属性在架构设计中是最重要的，它通常左右着架构风格的选择。另一方面，软件系统非功能性需求的的满足，仅凭编程级的努力是达不到的。比如为了提供高可靠性，往往涉及错误检测、预防、修复等，这就需要架构设计中非常重视非功能性需求。

功能是重要的，但如果仅仅盯着功能而忽视了非功能需求，则可能导致架构设计的失败。

2）是否适应数量巨大且频繁变化的需求

需求的变更既蕴藏着风险又包含了机遇。

之所以说是风险，是因为不存在不需要成本的需求变更。任何变更都意味着时间和金钱的消耗，并且在大量变更以后程序可能被搞得混乱不堪，势必引起 Bug 增多，产品质量下降。

另一方面，需求变更也蕴含着机遇，对软件架构师而言，这个机遇就意味着极力设计出更稳定的架构，最终，这个架构能够支持需求在一定范围内“随需而变”。

从更深层次来讲，现在商业环境的特点就是频繁变化，国际化带给我们的就是对未来越来越不可预测，技术的位置也不仅仅是服务，很多情况下技术在一定程度上还会拉动需求变更。

不考虑适应数量巨大而且频繁的需求变更，架构师就难以“从容”的进行架构设计，最终甚至被变化拖垮而无法关注大局。

3）能否从容设计架构的不同方面

架构师必须具有宏观思考的能力，避免涉及太多的技术细节，但这并不意味着架构设计是宽泛的甚至简单的。

软件架构必须对开发提供足够的指导和限制，这无疑意味着软件架构的工作是复杂的。架构师必须深入研究软件系统运行期间的情况，合理划分不同部分的职责，权衡轻重缓急，并制定相应的并行、分时、缓存和批处理等设计决策。

架构师必须掌握系统化的方法，对复杂问题“分而治之”，分解成独立的视图。然后再综合考虑各个视图之间的相互支持、互相影响等问题。

4）能否及早验证架构方案并做出了调整

现代软件开发非常注意早期缓解高风险。而架构设计又是软件开发风险比较高、也是最为关键的一环。架构设计的是否合理，将直接影响到软件项目最终是否能够得到成功。

毕竟，软件架构中包含了关于如何构建软件的一些最重要的设计决策，比如系统分为哪几个部分？各部分之间如何交互的？等等。这些设计决策能保证最终产品满足非功能需求（比如性能、可靠性、安全性）吗？

所以，在大规模开发之前，尽早构建架构的早期版本（架构基线），并通过对这个原型的测试评审，来评估这个架构是不是合理，这是非常重要的事情。

3，软件架构设计策略

策略对实践提供总体上的指导，对于有难度的工程（比如软件工程），或者有竞争性目标（软件中时间、质量、范围、成本之间存在竞争）而言，策略往往是制胜的关键。一定要注意，策略来自于问题，没有问题的策略是无目之本。下面，我们针对成功架构设计的四个要素，以此衍生出四个问题，作为讨论相应的策略的基础。这样的思考过程也可以成为我们研究其它架构问题的思考范例。我们先把关键点归纳成下面的表。

编号	关键点	问题	危害	策略	策略要点
1	是否遗漏了至关重要的非功能性需求	对需求的理解不系统、不全面、对非功能需求不够重视。	造成返工，项目失败	全面认识需求	弥补非功能需求的缺失
2	是否适应数量巨大且频繁变化的需求	对于时间和质量的矛盾，办法不足，处理草率。	耗时不少，质量不高	关键需求决定架构	把架构理解成概要设计，过于粗糙，不能适应实践要求
3	能否从容设计架构的不同方面	架构设计方案覆盖范围严重不足，许多关键决定被延迟，或者由实现人员仓促决定。	开发混乱，质量不高	多视图探寻架构	架构师开展系统化团队开发的基础，应该对不同的涉众提供指导和限制
4	能否及早验证架构方案并做出了调整	假设架构的方案是可行的，直到后期才发现问题，造成大规模的返工。	造成返工，项目失败	尽早验证架构	架构设计方案应该解决重大技术风险，并尽早验证架构

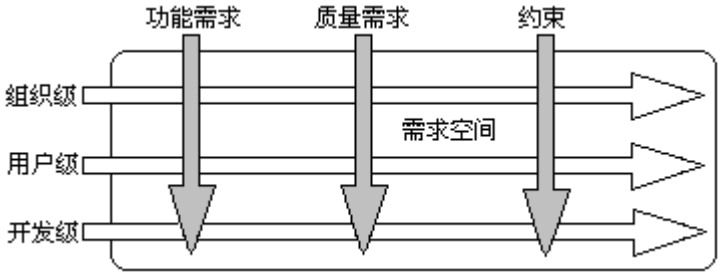
根据这张表，我们来讨论一下针对每个关键点的相应策略：

1）全面认识需求

既然软件架构强调的是整体，而整体的设计决策必须基于对需求的全面认识，所以全面认识需求，是软件架构的第一项需求。

但是全面认识需求并不等于“眉毛胡子一把抓”，而是需要对需求进行梳理清楚，在梳理的过程中把需求理解清楚。全面认识需求，就需要从不同级别来考察需求，这三个级别分别为：组织级、用户级、开发级。还需要对每个级别考虑不同类型的功能需求、质量属性、

约束，如下图所示。



一方面来说，需求是分层次的，对用户高层而言是帮助他们达到业务目标，最终用户而言，是辅助他们完成日常工作，对开发者而言，有着更多用户没有觉察到的“需求”需要实现。

关注需求的实践意义在于在需求之间建立可跟踪性，以免由于遗漏需求使软件达不到要求，或者一厢情愿的为用户制造没有实际意义的功能。当客户方老板说：“我希望这套软件能为我赚更多的钱。”这恰恰是一个商业目标，并会对需求和设计产生很大的影响。

总之通过需求分类，将有助于全面认识需求、分门别类的把握需求，从而设计出高质量的软件架构。

全面认识需求还有一层含义，那就是应当在深思熟虑之后作出合适的权衡和取舍。一方面众多质量属性往往会有冲突，我们必须权衡，另一方面，通过复杂的设计所支持的变化可能根本不会发生，这就造成过度设计，从而造成资源浪费并增加了开发的难度。

2）关键需求决定架构

软件架构师没有时间对所有的需求进行深入分析，这是现实，软件架构师没有必要对所有的需求进行深入分析，这是策略。

架构师要做的就是：接受现实，采取策略。关键需求决定架构有两个方面的内涵：

功能需求数量太多，应该控制架构设计的时候需要详细分析的用例个数。

另一方面，不同质量属性往往相互冲突，于是我们自然应该权衡那一部分质量属性是架构的重点设计目标。

在实际项目中，我们往往在项目的业务目标及核心需求达成共识之后就开始架构设计了，在这种情况下，关键需求决定架构的策略非常适合。

关键需求决定架构的策略有利于集中精力深入分析最为重要的需求，人的思维应对复杂问题的能力是有限的，当我们把问题分解、化简和转换，最后集中精力扑在相对较少的需求上的时候，可以更为深入地分析这些需求，有利于得到更加透彻的认识，从而设计出合理的架构。

例如，回顾前面讨论过的 PMT 软件项目，在全面分析需求的的基础上，下一步我们必须确定关键需求，以把握设计的重点。在这个阶段。我们应该多问几个为什么，例如：为什么“从 HR 系统导入资源”是架构设计的关键需求呢？这是因为这个功能涉及了 PMT 与外部设备接口的模块，而其它功能没有“覆盖”这一点。下表列出了 PMT 项目的关键需求子集。

关键需求			
功能需求	非功能需求		
	约束	运行期质量属性	开发期质量属性
创建项目 查看项目信息 添加项目任务 从HR系统导入资源 发布通知	客户群工作的平台多样化 成本收益考虑 应考虑外包趋势 和其它系统交换数据	跨平台运行 易用性 互操作性	可扩展性

事情简单勇者胜，事情复杂智者胜。

面对时间的压力和“软件的复杂性”，我们有理由质疑“先分析透彻所有需求，然后设计出架构”这个似乎天经地义的说法。通过全面认识需求，我们获得了需求的广度，这个阶段特别是要关注质量和约束等非功能性需求上。但要进行架构设计，对全面需求了然于心之后，必须抓住关键需求决定架构，这就是需求的深度。这种策略被哲学性的潜藏在广度和深度之间。

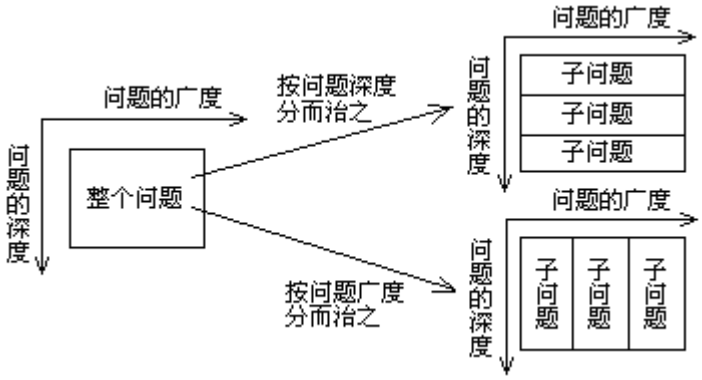
3）多视角探寻架构

利用多视角探寻架构，本质上属于分而治之的原理。当把复杂问题划分为多个相对简单的问题的时候，就可能对每个子系统强调不同重心，而心理学的研究早就表明，“强调能使经验活跃起来。”从而更有效地解决问题。

分而治之的策略有两种：

- **按问题深度分而治之**：也就是先不研究那么深、那么细，按问题复杂性的层次分而治之。
- **按问题广度分而治之**：先研究问题的一个部分，分割问题，各个击破。

下图表明了这两种情况。



接口与实现分离，实际上按照问题深度分而治之的例子。表示层、业务层、数据层单独处理，是按问题广度分而治之的例子。

4）尽早验证架构

应该尽早对软件架构设计方案进行开发和验证，而不仅仅向概要设计那样只是进行评审。

年具体而言：

- 应该真正通过编码把架构实现。
- 应该实际对架构基线进行测试，测试的重点是运行时的质量属性，如性能、可维护性、可伸缩性等。
- 要认真评估架构基线的实现过程，以对软件架构开发期的质量属性给出评价。
- 传统意义上的架构设计评审仍然是需要的，比如通过走查确保所有的功能需求能够被支持。
- 要对架构的不足之处及时进行调整，并再次进行验证。

要进行实际的测试，这一点很重要，否则，软件架构能否达到质量属性要求不得而知，这就埋下了众多技术风险。

在上述讨论的基础上，我们对软件架构的基本问题已经有了一个比较全面的了解，下面我们对问题进行分解，针对一些具体问题进行进一步的阐述。

二、建立弹性软件架构

对于基础架构，我还有几个观点要说明。在单一团队工作结束的时候，将建立系统早期

的、关键性的第一个版本，这是一个可执行的版本，我们称之为**架构基线**版本。在建立好架构基线之前可能要花费几个迭代周期，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的时候，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。

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许多可选的技术，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，把测试从被测单元中分离出来

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。

三、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成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户描述。这种识别主要是对系统需要做的事情进行界定、探索和发现。当确认了与架构有关的用户描述以后，需要把若干描述集合成一个个的主题。通过讨论，确定架构的基本方案，由于架构对产品的重要性，大多数情况下，这个方案还需要经过评审，然后才能进入架构设计迭代。此时，需要把用户描述转换为描述用例，这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、所使用的中间建和框架等等的影响。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

前面已经提到过，在架构设计的雏形完成以后，我们可以从以下四个视角优化架构：

- **从质量属性及其应对策略的视角**：根据非功能性需求和质量验收标准，提出整体上的体系结构策略。
- **从模块划分的视角**：进一步进行模块划分，从开发成本与集成成本两方面综合考虑问题，合理划分模块。

- **从解决缠绕和分层的视角**：分离出共享和缠绕部分，建立合理的分层结构
- **从构件化和软件复用的视角**：应用已经存在的基于复用的软件构造块，修改体系结构以适应这种构造块。考虑现有的结构能否有可以被将来复用的部分，按照构件的要求进行设计

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，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。

四、从质量属性及其应对策略的视角优化架构

通过从系统工程的角度分析与组织需求信息，我们已经由需求分析得到了一个框架性的架构雏形，但那只是架构的一个大概形状，在具体进行架构设计之前，架构师必须进行架构的进一步分析和设计，从不同的角度审视架构设计，多角度、多层次的修改架构设计方案，从而得到一个合理的架构基线产品。

对初步的架构轮廓作第一个方面的审视，是从需求分析中仔细思考质量需求对架构的要求，换句话说就是获取架构因素。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。

1，架构分析需要解决的问题

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。

这些工作，实现了对由系统工程学的角度提出的初步架构轮廓做第一次改进。

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。

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？

架构分析的一般步骤如下：

- 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

2，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	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。					

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。

3，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

非常多见），备忘录对理解当时的设计背后的动机极为有用。解释放弃被选方案的理由十分重要，在将来产品进化的过程中，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 ▪ 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：

- 不可改变的约束，包括安全和法律方面的事务
- 业务目标
- 其它全部目标

早期要决定是否应该避免保证未来的设计，应该实事求是的考虑，哪些是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。

五、从模块划分的视角优化架构

对初步的架构轮廓作第二个方面的审视，是考虑模块化的设计问题。也就是从架构的组成单元来说，定义清楚子系统以后，下一步就是定义模块。

1，模块化设计的概念

如何合理的进行模块设计呢？这里的关键是要保证模块的独立性。

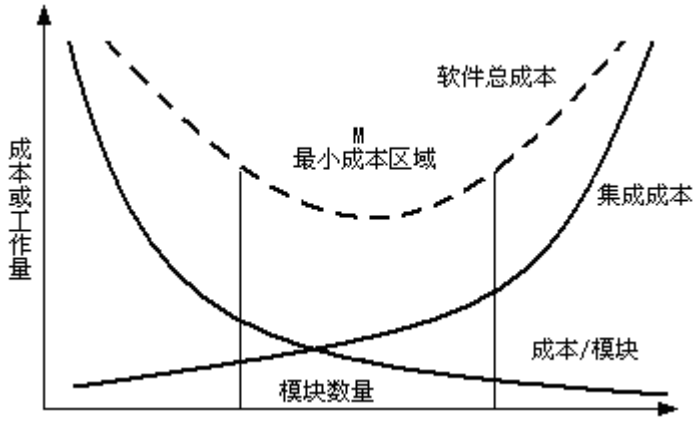
模块：模块是数据说明、可执行语句等程序对象的集合，是单独命名的并且可以通过名字来访问，例如过程、函数、子程序、宏等。

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：

- 一般来说，倾向于每个用户描述定义一个模块。我们应该努力使每个模块的大小差别在一个数量级之内。如果发现某个模块规模太大，就需要实现模块切割，然后针对这种切割的结果，反过来修改需求分析的时候用户描述（或者用例）的表达方式。这就是由设计引发的需求变更。
- 模块的大小一般以一个开发团队在一次迭代时间内能完成为好。模块切割方法与开发成本有关。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。



2，实现模块化的手段

- **抽象**：抽出事物的本质特性而暂时不考虑它们的细节。
- **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：

- 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。
- 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。
- 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。

3，模块化设计的一般准则

- 改进软件结构，提高模块独立性。
- 模块规模应该适中。大模块分解不充分；小模块使用开销大，接口复杂。
- 尽量减少高扇出结构的数目，随着深度的增加争取更多的扇入。扇出过大意味着模块过分复杂，需要控制和协调过多的下级模块。一般来说，顶层扇出高，中间扇出少，低层高扇入。
- 模块的作用范围保持在该模块的控制范围内。模块的作用范围是指该模块中一个判断所影响的所有其它模块；模块的控制范围指该模块本身以及所有直接或间接从属于它的模块。
- 力争降低模块接口的复杂程度模块接口的复杂性是引起软件错误的一个主要原因。接口设计应该使得信息传递简单并且与模块的功能一致。
- 设计单入口单出口的模块，以避免内容耦合，易于理解和维护。
- 模块的功能应该可以预测。相同的输入应该有相同的输出，否则难以理解、测试和维护。

六、从共享分层结构的视角优化架构

下面我们再从第三个方面审视架构设计，关注点在模块的**缠绕**和**分散**。所谓缠绕，指的

是一个模块是不是包含了多个模块不同的实现。所谓分散，一个模块的实现分散在多个不同的模块中。解决这些缠绕和分散问题，需要使用分层结构来解决。

1，层模式的问题与机会

在模块分割以后，就会发现有些功能块或者某些功能是共享的或者缠绕的，我们需要把这些模块或者功能提取出来，分成若干层次，这就是层模式。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。合理的分层也是架构师很大的一部分需要仔细设计的工作。

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。

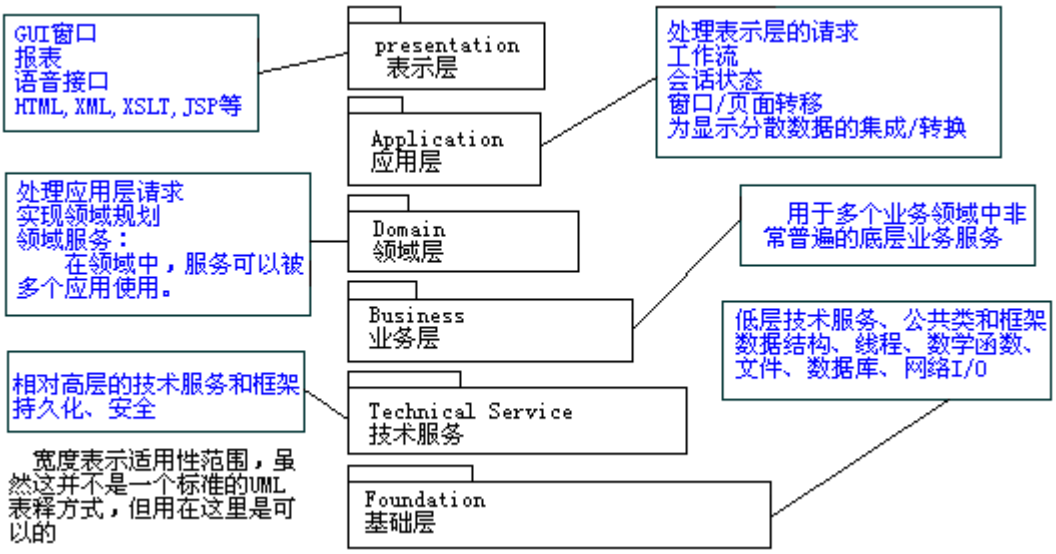
- 根据分离系统的多个具有清晰、内聚职责的设计原则，把系统大尺度的逻辑结构组织到不同的层中，每一层都具有独立和相关的职责，使得较低的层为低级和通用的服务，较高的层更多的为特定应用。
- 从较高的层到较低的层进行协作和耦合，避免从底层到高层的耦合。

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。作为分层架构的模式，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如下图所示，图中包的宽度表示的是应用范围。



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。

2，层模式的设计原则

分层并不是具有固定模式的，比如三层架构或者七层架构，而是需要根据情况合理布局。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。我们还需要注意到，分层主要解决共享和缠绕问题，而在设计时要考虑下层易变更问题，我们需要仔细考虑变化点和变更方式，合理应用设计模式，实现封装变化的结构形式，以此寻求架构解决方案，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。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合成员

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

“高层”、“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。

七、从软件复用与构件化的视角优化架构

事实上，经过从上面三个方面审视架构，我们已经建立了一个完整的而且比较良好的架构。但我们还需要从第四个方面在更高的层次审视我们的架构，需要考虑的又一个问题就是软件的复用。复用可以大大降低后期成本，提高整个软件系统的可升级性与可维护性。我们可以考虑哪些结构可以使用已经存在的可复用结构和产品，某些结构可以利用 GoF 的设计模式设计可复用的构件已备后期使用。还需要根据需求分析得出的易变点仔细设计产品结构，确保后期的变化不至于对产品带来太大的影响。而复用的一个重要的手段，就是面向构件的方法。

1，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。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。

2，面向构件的方法简述

构件也称为组件，面向构件的方法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：

- 构件基础设施
- 软件模式
- 软件架构

● 基于构件的软件开发

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

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

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。

3，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。

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

3）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。

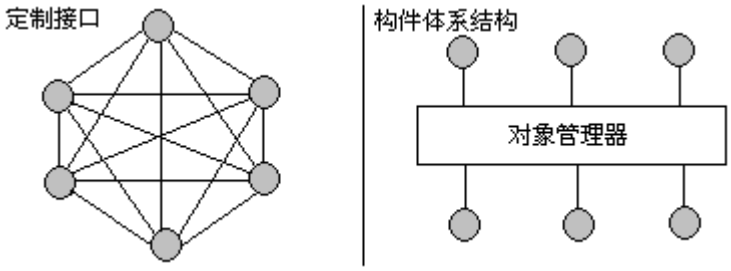
商业软件也可以在某个领域内实现专有，也就是提供应用程序接口（API）为应用软件

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。



为此，我们可以建立一个称之为对象管理器的层，用于统一协调各个对象的沟通，从可维护性角度这是一个比较好的结构，但是在某些特别强调效率的点可以避开它。系统的架构的一个重要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。

经过从四个方面审视我们的架构，经过分析、权衡、设计和部分编码，我们就可以得到一个稳固的架构。这个架构经过经过评审，就可以作为后期开发的基础架构。

2.9 软件架构设计的流程

综上所述，我们就可以比较条理化的建立软件架构设计的流程了。典型软件架构设计的流程如下图所示。

一、业务架构概念

在构建软件架构之前，架构师需要仔细研究如下几个问题：

- 系统是为什么目的而构建的？
- 系统投运后服务于哪些利益相关者的利益？
- 什么角色在什么时候操作或者维护系统？
- 业务系统实现方法是怎样的？

- 整个业务系统是如何依靠系统而运转的？

为了回答这些问题，需要仔细阅读需求分析文档中的业务模型建立、问题域及其解决构思、产品模型的构思等等前期文档，站在系统架构的角度，全面清晰的建立业务模型，包括组织结构关系、业务功能、业务流程、业务信息交互方法、业务地理分布、业务规则和约束条件。这个阶段的主要活动如下：

- 建立产品范围、目的、最终用户、业务背景等重要的初始信息。
- 建立完整的业务和系统的术语字典，确保项目相关人员理解上保持一致。
- 建立宏观层面业务的总体概念，明确总体流程、业务功能的边界、交互与协作方式，建立系统的概念模型。
- 汇总业务总的组织结构与协作职能关系。
- 分析业务的组成节点，以及节点间交互、协同与信息的依赖关系。
- 分析业务节点的事件、消息，以及由此引发的状态转换关系。
- 汇总业务运行的基本数据模型，以便于跟踪信息的流动与格式转换。分析业务数据的关联关系。
- 理解问题域以及系统需要解决的问题。
- 分析业务运作层面的基本业务规则与约束条件。

这个阶段的活动非常重要，架构师只有具备了这些相互关联的业务概念以后，才可能从这些概念中抽取恰当的架构因素。

二、产品架构概念

在理解业务的基础上，我们需要进一步思考产品架构的概念，这个阶段从活动的层面看实际上与建立业务架构概念是一样的，但是思维的重心转移到如下几个方面：

- 新系统投入运行以后，最高层面的业务会怎样运作？
- 新系统是如何解决原来工作系统的问题的？
- 新系统的投运，会对原来的组织结构划分发生什么样的影响？
- 由于新系统取代了原来的一些业务职能，业务节点的分布会发生怎样的变化？变化后的节点间的信息又是怎样交换与依赖的？
- 变化后的业务事件传递又会发生怎样的变化？
- 新的系统加入以后，哪些业务流程将会发生重大变化？哪些不会发生变化？
- 业务的状态转换关系将会如何随着新系统地加入而改变？
- 业务的数据模型将会如何随着业务流程的变化而变化的？
- 新系统地加入，将会如何影响新的业务规则和约束？

从这些角度出发，我们会重新构建未来新系统投运以后的业务规则，相应的新规则也需要建立，这就实现了业务过程的重构。

三、建立稳定的架构基线

在对业务领域与问题域有深刻的理解以后，我们需要继续研究如下一些问题：

- 这个复杂系统应该分成多少和哪些子系统？
- 子系统是如何分布在不同的业务节点或者物理节点上的？
- 这些分散的子系统将提供哪些接口？这些接口如何进行交互？
- 各个子系统需要交互哪些数据？
- 每个单独的子系统，所需要实现的功能有哪些？
- 整个系统对各个子系统有哪些功能、性能和质量上的要求？

“基线”这个词有两个意义：

- 这个阶段将会对整体架构策略做出重大的设计上的决策。一旦作出了这些决策，后续开发没有重大情况不允许变动。
- 这个阶段完成的工作，本身就是架构阶段的重要成果，需要广泛认同、集体遵守以及具备强制的约束力。

尽管在后期的演变中，这样的基线实际上还会不断的精化和优化，但最初下功夫构建稳定的架构基线是十分必要的。这个阶段的活动如下：

- 校验与确认前期所有的业务架构与产品架构的信息，必要的时候补充相应的信息。
- 修订和增补术语字典，确保所有的相关人员对术语有相同的认知。
- 把整个系统功能进行拆分，并且分解到不同的运行节点上，构建不同的系统集和子系统，在全局范围内划分接口与交互规则。
- 汇总系统/子系统接口信息，便于检索与浏览。
- 规划整个系统的通信链路、通信路径、通信网络等传输媒介。
- 把产品架构概念中的业务职能与系统功能相对应，从而确保满足业务要求。
- 分析系统/子系统在运行起动态协作需要交互的信息。
- 构建和模拟整个系统在业务环境下的动态特性，规划全系统内部状态变化过程、触发的事件及约束条件。
- 详细汇总各个子系统间信息传递的过程、内容以及其它辅助信息。
- 根据初始的数据模型构建数据物理模型。
- 汇总质量上对系统的要求，并把这些质量要求细化分解，量化到各个子系统中去。
- 构建整个系统与子系统的构建和演化计划，在迭代过程中构建整体项目规划和初始的迭代规划。
- 按固定时段预测技术的演化，汇总整个系统的应用技术及其演化。
- 分辨与汇总整个系统在不同阶段必须遵循的标准。
- 把业务约束映射到各个子系统，必要时附加 IT 业务约束。

四、子系统架构的设计与实现

通过上述各主要过程，我们已经实现了一个重要的总体架构基线。所有的子系统设计都是在这个庞大的架构基线约束下展开，至此，首席架构师逐渐淡出，让位于子系统架构设计师。子系统架构设计师的任务是继续分解、细化、设计各个子系统。在这个阶段，将会考虑更加细节的问题，为后来的构件设计与单元设计作准备，我们需要考虑的问题如下：

- 规划给该子系统的功能是否可行？
- 在整个子系统的范围内，又能分解成什么子功能集？
- 在整个子系统的范围内，又能把哪些子功能合并到某些构件中去？
- 这些构件与子功能集是如何通过接口与子系统衔接的？

事实上子系统架构设计本身就是一个完整的系统设计，所区别的是视野集中到子系统的范围内，这个阶段的活动如下：

- 校验与确认前期与该子系统相关的业务架构与产品架构的信息，必要的时候补充相应的信息。
- 增补与本子系统相关的术语字典，确保所有的相关人员对术语有相同的认知。
- 把整个子系统功能进行拆分，并且分解到不同的构件节点上，构建不同的子功能集，在子系统范围内划分接口与交互规则。
- 汇总子系统/构件接口信息，便于检索与浏览。
- 规划整个子系统的通信链路、通信路径、通信网络等传输媒介。

- 把产品架构概念中的子业务职能与构件功能相对应，从而确保满足业务要求。
- 分析子系统/构件在运行起动态协作需要交互的信息。
- 构建和模拟整个子系统在业务环境下的动态特性，规划子系统内部状态变化过程、触发的事件及约束条件。
- 详细汇总各个构件间信息传递的过程、内容以及其它辅助信息。
- 根据初始的数据模型构建子系统相关的更详细的数据物理模型。
- 根据质量上对子系统的要求，并把这些质量要求细化分解，量化到各个构件中去。
- 构建整子系统的构建和演化计划，在迭代过程中构建子系统项目规划和更详细地的迭代规划。
- 按固定时段预测技术的演化，汇总子系统的应用技术及其演化。
- 分辨与汇总子系统在不同阶段必须遵循的标准。
- 把业务约束映射到各个构件，必要时附加 IT 业务约束。

五、构件与实现单元的设计

进入构件设计阶段也就是进入了详细设计阶段。这个阶段主要的工作就是接口与功能设计。在迭代模型下，这个阶段很大程度上是在迭代过程中完成的，由某个设计人员带领全体开发团队进行分析和设计。

在这个过程中，我们应该考虑在小粒度架构中如何使产品需求变更不至于对产品质量造成影响，还需要考虑业务概念模型与产品功能块有相应的追溯和回溯关系。这些问题，我们将会在后面用适当的篇幅进行讨论。

小结：

大型复杂项目的成功依赖于合理的项目组织，这种组织概念包括人力资源的组织和产品架构的组织两个方面。敏捷项目管理为这两种合理的组织提供了思维基础，为项目的成功提供了保证。一个典型的例子是 20 世纪 90 年代加拿大空中交通系统项目（首席架构师：Philippe Kruchten）。150 个程序员被组织到 6 个月的长周期迭代中。但是，即使在 6 个月的迭代中，10～20 人的子系统，仍然把任务分解成一连串 6 个为期一个月的小迭代。正是这个项目的实践成功，Philippe Kruchten 才成为 RUP 的首倡者。

敏捷过程的提出直接影响到架构设计的核心思维，正是因为敏捷过程的提出，才有了架构驱动、弹性架构和骨架系统这些理念。也直接影响到需求分析方法、项目规划和估计方法等一系列领域的新思维。甚至推动了业务敏捷以及与之相适应的基于服务的架构的提出。这些观念的提出又更加推动了软件工程学向更高的层次发展。

下面我们将讨论几个专题，但讨论的时候希望研究一种思考问题的方法，从而为大家解决更广阔的问题提供一个思维的平台。这些专题并不是独立存在，而是融合在本章所讨论的各个阶段之中。再一次强调，方法和技术是会变化的，但优秀的思维方式是永恒的！

第三章 质量属性对架构策略的影响

架构设计的本质，是应对质量属性的解决方案，从而形成相应的架构策略。

在上一章关于架构分析的讨论中，我们简要地谈到了架构设计思维重心，很大程度应该放在对于质量需求的应对策略上。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。

构建一个好的系统架构，终极目标是使它具备形态之美，体系中每个部分相互配合，共同达到系统功能的完美实现。但是，系统架构师最需要考虑的问题，是这样的架构能不能达到系统的质量要求，这就是系统的设计品质而且这种品质又应该是可度量的。我们需要在如下两个点上考虑问题：

- 如何构建一个过程来考虑和完善系统的质量属性。
- 对于不同的质量需求，我们应该采取什么策略来满足要求。

下面我们分别加以讨论。

3.1 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

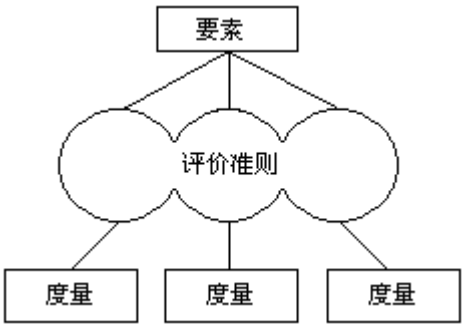
软件的质量由一系列质量要素组成，每一个质量要素又由一些衡量标准组成，每个衡量标准又由一些量度标准加以定量刻画。质量度量贯穿于软件工程的全过程以及软件交付之后，在软件交付之前的度量（内部属性）主要包括程序复杂性、模块的有效性和总的程序规模。在软件交付之后的度量（外部属性）则主要包括残存的缺陷数和系统的可维护性方面。

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：

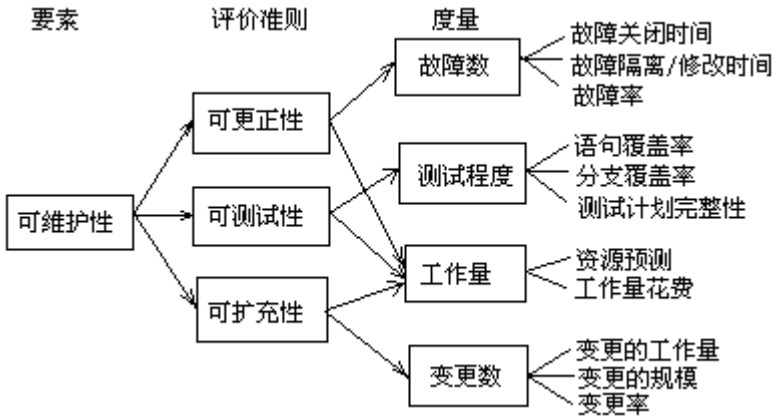
	特性	副特性
1	**功能性(functionality)**：与一组功能及其指定的性质的存在有关的一组属性。功能是指满足规定或隐含需求的那些功能。	**适合性(suitability)**：与对规定任务能否提供一组功能以及这组功能是否适合有相关的软件属性。 **准确性(accurateness)**：与能够得到正确或相符的结果或效果有关的软件属性。 **互用性(interoperability)**：与同其他指定系统进行交互操作的能力相关的软件属性。 **依从性(compli ance)**：使软件服从有关的标准、约定、法规及类似规定的软件属性。 **安全性(security)**：与避免对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)**：与在规定的一段时间内和和规定的条件下,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(fauIt 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直接受影响数据的能力，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	**资源特性(resource behavior)**：与软件执行其功能时，所使用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** **(maintaina bility)**：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(sta、bility)**：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)**：与软件可从某一缓建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关软件准备的处理或手段就能适应不同的规定环境有关的软件属性。 **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：



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。我们可以通过下面的检查表来描述和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）。

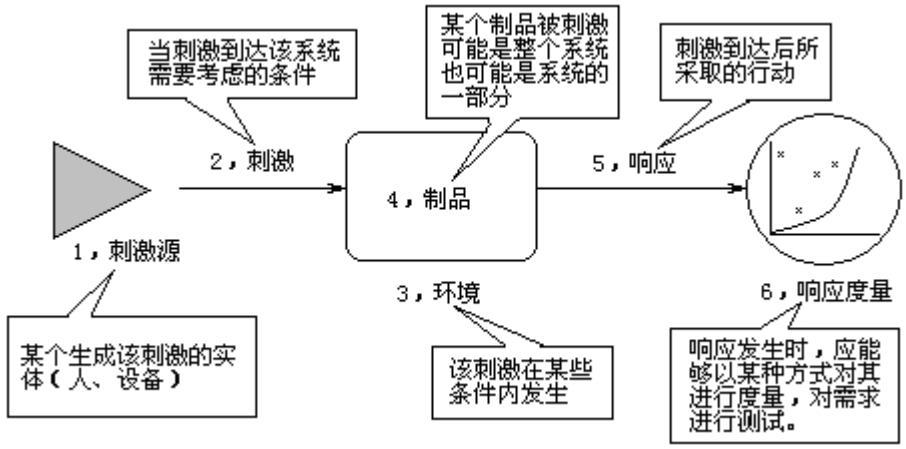
7	**可重用性**	该系统是一系列的产品线的开始吗。 其它建造的系统有多少和现有系统有关呢？如果有，其他系统能重用吗。 哪些现有构件是可以重用的。 现有的框架和其它代码能够被重用吗。 其它应用程序可以使用这个系统的基础设施吗。 建立可重用的构件，代价、风险、好处是什么。
8	**集成性**	于其它系统进行通信的技术是基于现行的标准吗。 构件的接口是一致的和容易理解的吗。 有解释构件接口的过程吗。
9	**可测试性**	有可以测试语言类、构件和服务的工具、过程和技术吗。 框架中有可以进行单元测试的接口吗。 有自动测试工具可以用吗。 系统可以在测试器中运行吗。
10	**可分解性**	系统是模块化的吗。 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。

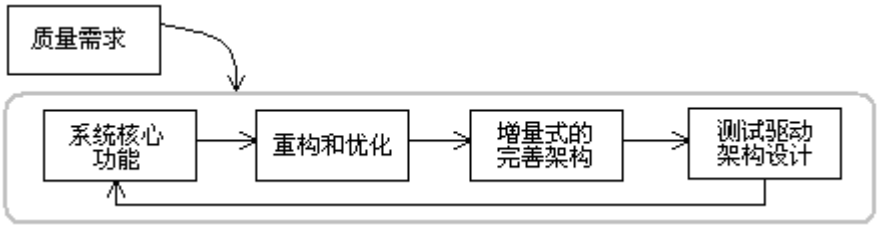


注意：于不同的系统，对于质量的关注点是不一样的，例如某个设计强调了高性能，另一个设计可能更强调高可靠性，针对这些质量属性就可以形成基本设计重点，我们称之为“解决方案”，解决方案是一种影响质量属性的设计决策，把各种解决方案打包集合，我们称之为“架构策略”。下面我们简要介绍一下几个最重要的质量属性的解决方案。

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。

3.2 应对质量属性的架构设计过程

在上一章我们已经构建了软件架构设计总体过程，为了强调针对质量属性的架构设计策略，我们还应该在每个节点加入如下子过程。



一、以核心功能为主进行架构设计

对于任何一个基础架构，我们都需要回答如下一些问题：

- 这个系统有哪些核心功能？
- 这些核心功能分布在哪些系统级的框架之内？
- 是什么样的质量要求才使我们这样分配功能分布？
- 这些系统功能互相交互的时候考虑了哪些质量要求？

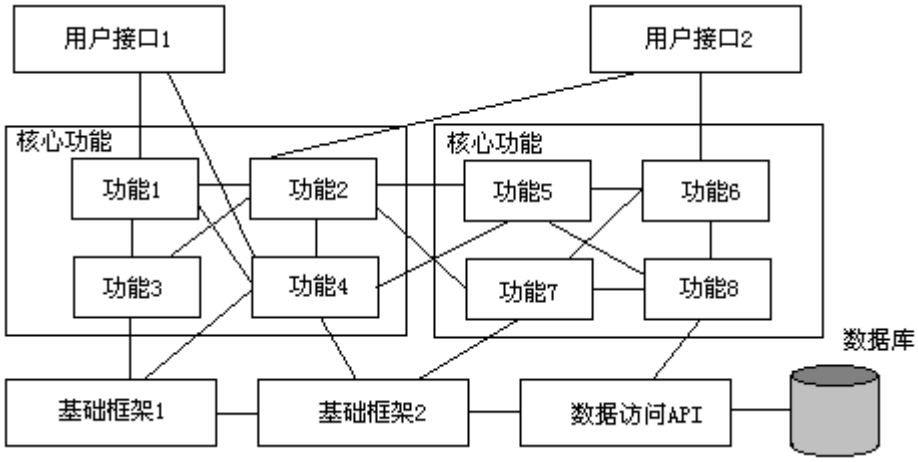
事实上，我们总是倾向于使用一些我们熟悉的架构，但是通用的架构很难让我们回答上述问题。为了确保设计的清晰，在我们已经存在一些熟悉的架构的基础上，需要按照下属步骤来考虑问题：

- 清晰的界定核心功能，例如工作流、业务处理单元等。
- 清晰的界定那些为了实现这些核心功能而必须提供的基础框架，例如通信、安全、并发等框架。
- 清晰的界定那些功能与功能，以及功能与框架之间的交互关系。
- 仔细考虑系统质量要求，根据质量要求合理分配核心功能的分布、基础框架的分布，是核心功能的协作与交互能够满足系统质量要求。

利用这些步骤，我们就可能在一定程度上，把一个系统架构构建成核心功能与基础框架规划清楚，特别是把核心功能与其他功能的交互关系，或者与基础框架的交互关系更加清楚的表达出来，这样也有利于进一步重构的体系。

二、以质量属性为依据进行重构和优化

通过上面这个步骤，实现了所谓的架构战略性的设计，但这还不够。一个好的架构总是经历从粗到精、逐步分解、逐步稳定的过程。仅仅从核心功能上划分并不一定能达到所要求的质量标准，我们还需要根据质量需求对架构重新拆分、合并、功能再分配等，这就重新构造了新的体系结构。例如，在以核心功能为主进行架构设计的时候，我们得到如下图所示的基本框架。



但是这个框架具有很大的危险性，因为各个功能块之间有很强的缠绕性，系统的可维护性、可实现性都很差。这样的系统从功能上可能已经达了要求，但是，它既无法维护，也无法扩展，当客户流量比较大的时候，性能可能也是个问题。为此，我们需要更加细致地从战术上考虑问题，仔细对整个架构进行重构和优化，整个优化过程可以依据两个方向：

- 自底向上，逐步合并各个功能，清理一些设计不合理的构造快，保证以一条清晰的核心功能为主线，这样我们才可以从目前的架构设计中，清晰的表达先前的所谓战略设计是什么。
- 自上而下，把系统的质量要求逐步分解，考虑这些分解出来的质量要求，选择最佳的解决方案，在适当的地方添加这些内容，确保系统既满足核心功能的需求，又满足了分解的质量要求。

这样两个方向反复思考和重构的结果，保证系统既有一条清晰的功能主线，更能满足质量要求，而且更着重以质量为主线考虑问题。

三、增量式的完善架构设计

一次性的完成高水平架构几乎没有可能性，因此，增量式迭代的完善架构几乎是一个普遍采用的方法。通过迭代，我们可以通过多次的提炼、重构和优化来使架构达到要求。我们建议采取如下步骤来实现：

- 把功能需求与质量需求按优先级排序。
- 估算完成每个需求和质量需求在架构阶段需要完成的时间。
- 决定迭代周期。
- 决定迭代次数。

- 把排定的功能需求与质量需求分配给每个迭代周期。
- 实施每次架构增量，以系统核心功能为主要线索完成任务。
- 重构和优化，以质量属性为核心优化架构。

四、以测试驱动架构设计

测试驱动的设计是保证设计质量的良好方法，我们可以通过下面的步骤来完成它。

- 首先，在进行每一次架构的增量、提炼与重构之前，需要进行初步的测试准备工作，也就是明确如何进行质量的测试。在系统重构完成之后，根据已有的系统核心功能，接口分布、协作以及系统质量要求，由测试人员帮助建立测试规约。
- 其次，所建立的测试规约，会被后来的设计、编码人员参考和执行。架构和设计人员必须保证后续人员完全遵循这些测试规约，也就是说架构人员还附带有监督职能。
- 在编码过程中的测试，将会发现许多初期没有考虑到的设计缺陷，架构人员需要根据这些测试报告中检测出来的缺陷，重构自己的架构。

所以，测试驱动的开发是迭代过程中优化架构的重要手段。上述三个步骤第一项是最困难也是最重要的，因为系统的质量完全在于你如何测量它。这就要求需求分析过程中，对于非功能性需求一定要定义测试条件，而测试条件的定义并不容易。

下面，我们进一步讨论对于不同的质量需求，应该采取什么样的策略来满足这些需求。注意，这些讨论会有一些实例，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。

3.3 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：

α =平均正常工作时间/（平均正常工作时间+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。

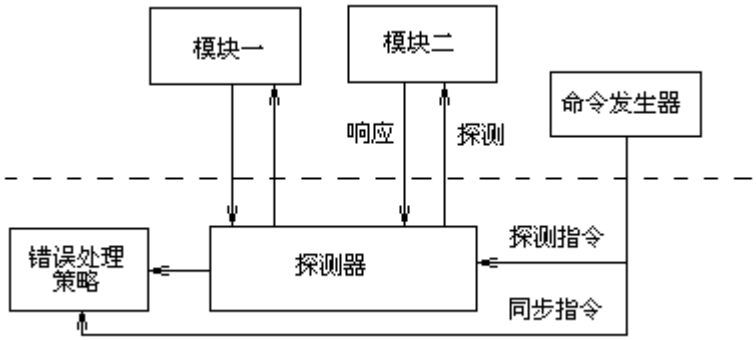
问题在于是不是使用了这些技术，设备的可靠性就一定提高了呢？事实证明并非如此，我们必须利用历史测量的数据进行数据处理，要对模块的易损性以及发生的位置等要素进行统计检验，从而找到最需要进行可靠性架构设计的位置，力争在比较少的成本下获取比较大的可靠性指标，这往往是架构决策的基础。

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）：

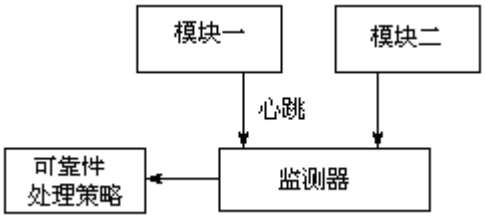
一个构件发出一个命令，并希望预定时间内收到一个审查构件的响应。和心跳方式相比，它的特点是发出检测命令是由专门的构件完成的。这个解决方案一般用在处理共同完成某项任务的一组构件内。一般情况下，“探测器”可以放在底层，它向相关软件进程发出命令，而高层命令发声器向底层探测器发出测量命令。在模块设计的时候，必须包括对于健康探测器探测命令的响应信息，这种响应应该力求对于各个模块是统一的，同时也要尽可能的简单。命令/响应机制一般以方法访问的方式工作，这在某些情况下可能是方便的。



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，但如果所有的备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同一个进程中处理相关问题。

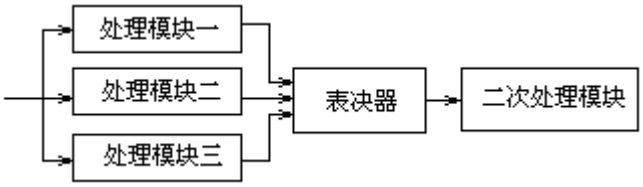
三、错误恢复

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的系统，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

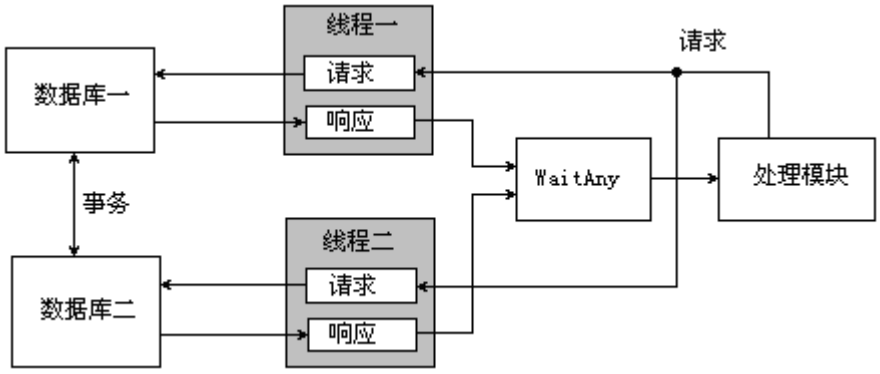
对于复杂的数据处理计算，例如快速傅里叶变换或者卷积、相关函数的计算、航路计算、及威胁计算以及应对策略计算等等，如果处理上出现了问题（错误），往往会带来灾难性的后果，在这种情况下，可以使用表决方案。它的特点是运行在冗余处理器上的每个过程都具有相同的输入，表决器采取某种策略取出数据，常用的是多数法，或者首选法。



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。

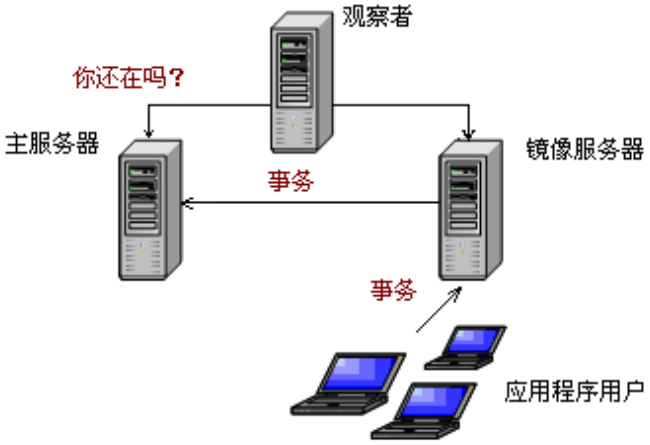
请看下面的场景：

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。

反映初始角色配置的数据库连接



反映“观察者”服务器转换为另两个服务器角色后的数据库镜像

注意，当发现主服务器有问题的时候，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。

3.4 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。

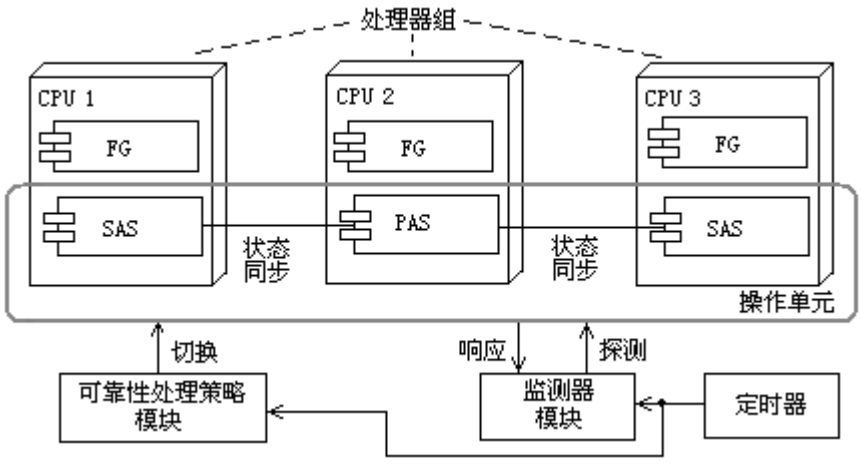
一、进程间提升可靠性的方法

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。

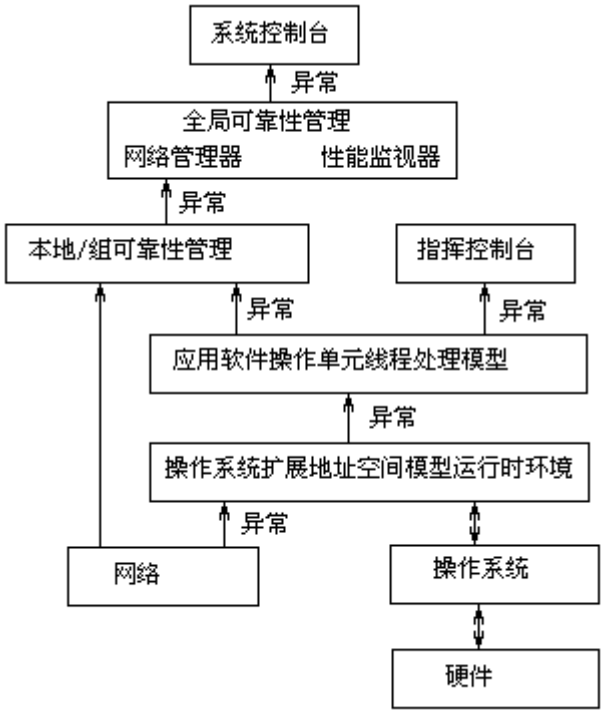
有经验的项目组成员可以按照这个步骤开展工作，为了加速开发，可以构造一个“代码模板”，以加快开发速度和减少错误。

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。

3.5 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。

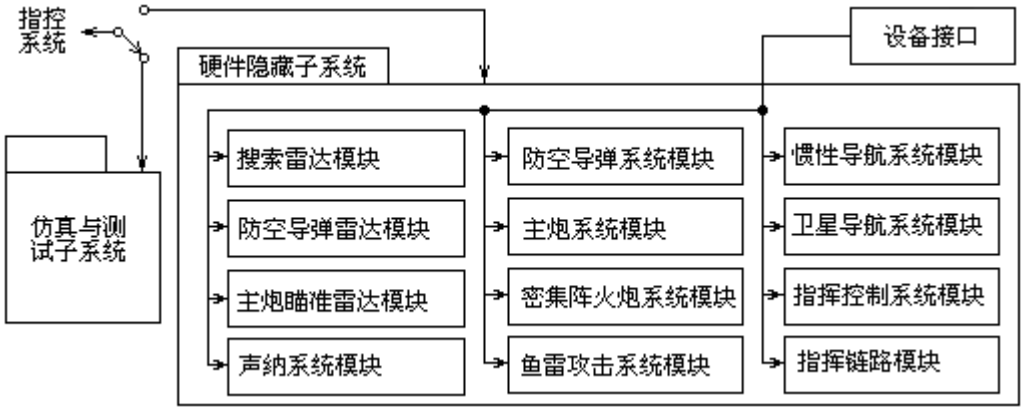
例如：舰用作战指挥平台由于武器单元经常升级，所以在模块设计的时候，主要考虑的问题是信息隐藏。很多分系统在主干系统的生命周期内可能会用新的型号或新原理组成的系统更换，这就需要把与这个系统交互的细节封装到某个模块中去，这个模块的接口只不过是一个抽象分系统。每个模块都提供了一组仅通过某个特定的访问过程与其它模块交互，如果换成了新式的分系统，只需要改变这个模块中的某些细节，对软件其它部分没有影响。

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。

由于易扩展性摆在了重要的位置，所以模块划分专门提供了硬件隐藏子系统，这个子系统的作用，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对象，也就是说这个子系统实现的是虚拟被控对象。



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 依靠于 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。

3.6 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。

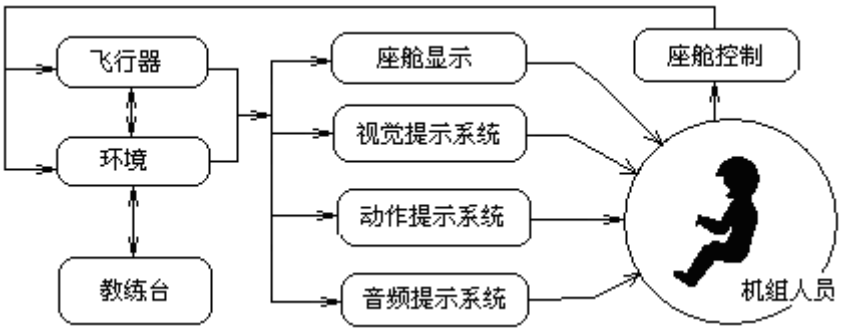
由于通信链路的加长，本来就因为规模大而不易实现的可集成性，实现的难度更大了。

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:

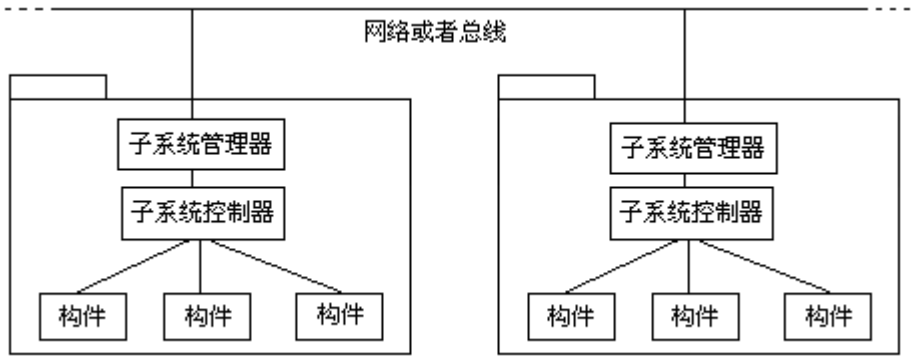
1，性能策略

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：

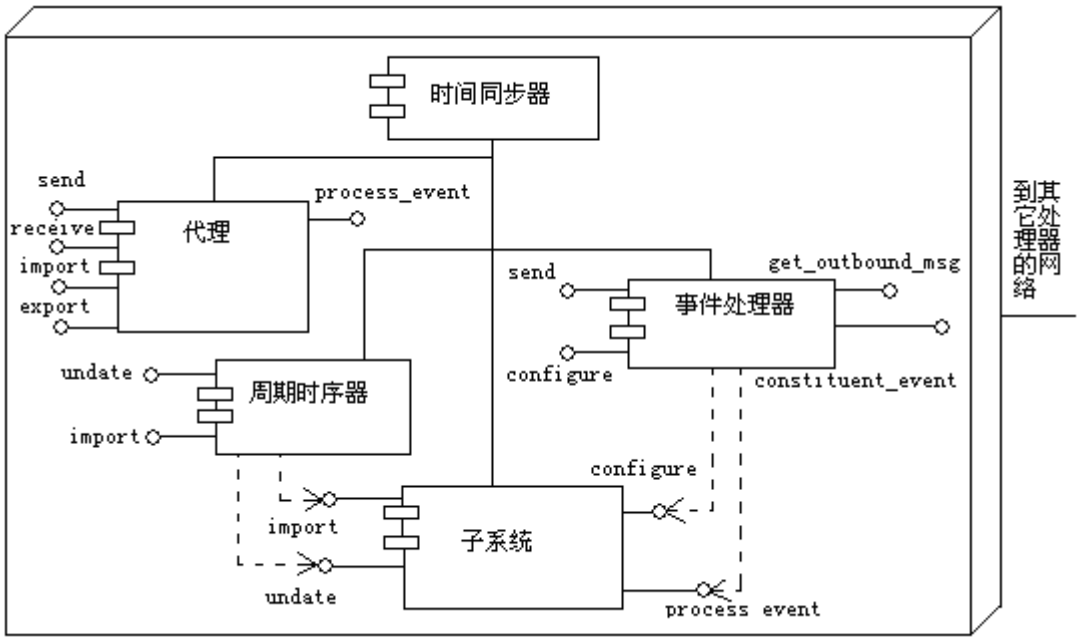
- 系统子结构的简单性和相似性。
- 把数据和控制信息的传递策略与运算分离开。
- 模块类型数量最少。
- 较少的系统级协调策略。
- 设计的透明性。

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。

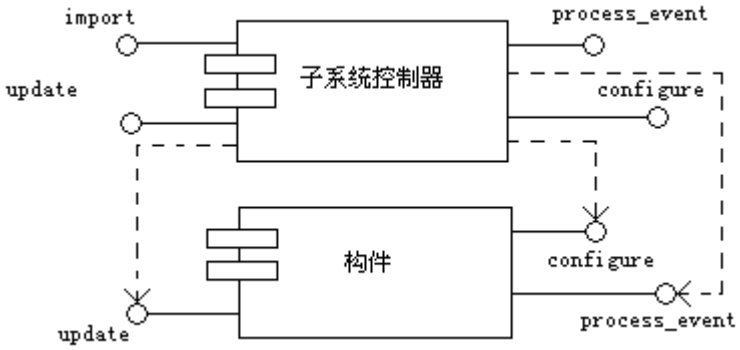
Send：子系统控制器使用它向其它子系统控制器发送事件。

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，已备 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快，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。

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。

3.7 基于质量属性的优化和重构

如果我们的目标仅仅是为了完成某些功能，实际上什么样的架构都可以的，但是如果考虑到应对质量属性的要求，那架构策略就显得非常重要而且具有某些特征。好的架构并不可能一次到位的完成设计，更多的是在一种迭代优化过程不断的完善，而这个优化过程又伴随着架构的一次又次的重构，前面所讨论过的从四个视角优化架构，本质上也是一个对架构的重构过程。为此我们有必要认真讨论架构层面的重构技术，为了对这个问题有更加透彻的理解，让我们先看一下最初的重构技术是如何定义的。

一、软件重构技术的本质

1，重构的定义

重构是以各种方式对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的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。

这些定义中最重要的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。

2，重构的原则

1）一个时刻只戴一顶帽子

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。

第一顶帽子是功能，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，目标是提高代码的结构质量。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顶帽子。

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戴一顶帽子。

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。

4，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。

软件系统重构广义上采取的步骤如下：

- 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。
- 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。
- 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。
- 遵循小步前进的原则实现重构操作。
- 在每步间运行测试以保证行为未被改变。
- 如有必要，对做过改变的接口修改测试代码。
- 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。

5，软件架构的重构

从上面的讨论还可以看出来，目前软件重构考虑的问题更多还是集中在代码阶段，但对于架构层面的重构技术却研究的并不充分。而作为一个架构师，所考虑的重构问题应该专注在架构层面。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。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。换句话说，重构的目标是使架构在保证功能行为的基础上，满足质量属性的要求。

如果让我们进行一次架构和设计重构，基本的操作思路就是按照下述的步骤贯穿整个系统的各个子系统、构件和服务：

- **问题及背景识别**：首先需要识别系统架构和设计中有哪些是可以优化和改进的部分，并澄清存在的问题、问题背景或原因。
- **改进方案**：思考作为一个系统架构人员，面对这样的质量属性问题时应该采取怎样的设计方案。同时，参照业界经典的问题解决方案（架构重构模式）也是极其有效的一种手段。
- **风格沿袭**：将系统中存在类似问题的其他部分，用类似统一的方案来解决。这样做的结果，会使系统架构和设计非常匀称和平衡。

上述的架构重构步骤，非常像是在使用一系列的设计模式来解决系统设计问题。其实，要想掌握架构和设计重构技术，就是要经过多次实践，总结经验后，最终能够提炼出一些“重构模式”的过程。即掌握问题的现象，并寻求问题的最佳解决方式。

二、重构模式

下面，我们针对系统架构和设计中的“坏味”（注：“坏味”是 Martin Fowler 的一个著名概念），分别总结出的一些“重构模式”，看看这些模式是如何把这些设计“坏味”去掉的。

1，实体重新命名

问题：进行架构和设计时所界定出来的系统组成元素（子系统、构件、模块等）名称使用混乱，不能很好地表达该系统元素的用途或语义，使系统结构难以理解。事实上，一个架构发展的历程中，命名的逐渐混乱，是促使架构老化的重要原因。

解决方案：

- 在系统全局范围内，使用事先定义的命名规则和编码规则。
- 尽量使用容易使项目相关人员理解的含有明显语义的名称来给系统元素命名。
- 改变系统元素名称时，需要考虑该元素与其他系统元素间的引用关系，使元素名称包含引用的概念。

举例：

重构时，我们可以使用适当的名称来表达两个构件或服务间的关系，如下图所示。

2，转移重复元素

问题：系统架构和设计中，同样的功能或系统元素屡次出现在很多其他系统元素中。这样会降低系统的可理解性，并使这些相同功能的代码散布在整个系统设计当中，无论从设计或编码的角度上，都无疑增加了理解和维护的复杂性。

解决方案：根据面向对象的“封装性”原则，一些共用的系统元素可以考虑封装起来，这样方便系统其他部分对其的重用。同时，这样的架构和设计将使系统结构描述更加简捷直接。我们需要做的事情是：

- 识别这些散布在系统结构当中的公共功能或公共任务。
- 汇总有哪些系统元素会利用这些公共部分。
- 尝试将这些共用部分从各个系统元素中转移到一个封装起来的系统元素中。

也就是说，我们可以使用一个新的构件来封装共享的功能，并移除原先那些构件中相同的设计逻辑；取而代之的是以调用的方式来满足各个构件的功能引用。

3, 利用抽象的层次结构

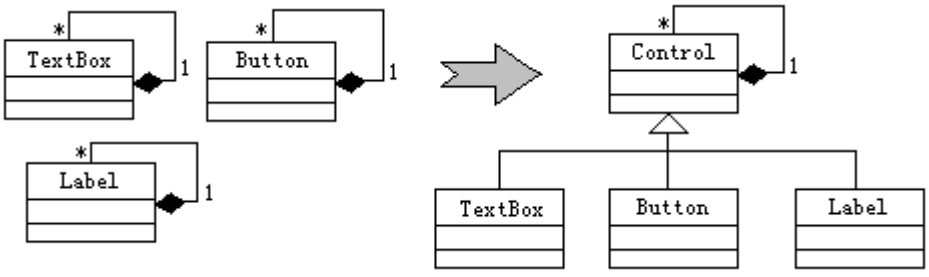
问题：架构重构期间，我们会发现有些设计单元所实现的功能非常相近。这些重复出现的相似设计单元以及后继的相似功能的代码，会严重干扰系统架构一些概念的一致性，从而降低了架构描述的简捷和可读性。进一步可能会导致系统调用时需要和各种对象变种进行紧耦合，从而形成难以应对的设计结构。

解决方案：

为了保持系统架构和设计中相似概念的一致性，应该考虑使用数据抽象的方式来使对象之间形成自然的层次结构关系。这里需要注意 Barbar Liskov 著名的 LSP（Liskov Substitution Principle，替换原则），这个原则的原始表述是：使用指针或引用基类的函数，必须使应用的时候不需要知道它的派生类的对象。一般的处理方法如下：

- 利用 LSP 为指导原则，引入一个“通用抽象”来封装那些相似通用的功能。
- 定义一个以“通用抽象”为基础的层次结构。
- 将其他具体变种的架构设计单元，按照层次结构依次进行衍生。

举例：在.NET 架构中，每个控件都需要存在一个集合，用以记录下一层控件的引用。按照 LSP 原则，可以组成以组合为特征的抽象层次结构，以便衍生其他特定的具体元素，如下图所示。



4，以适配（Adaption）方式替代中介方式（Mediation）

问题：

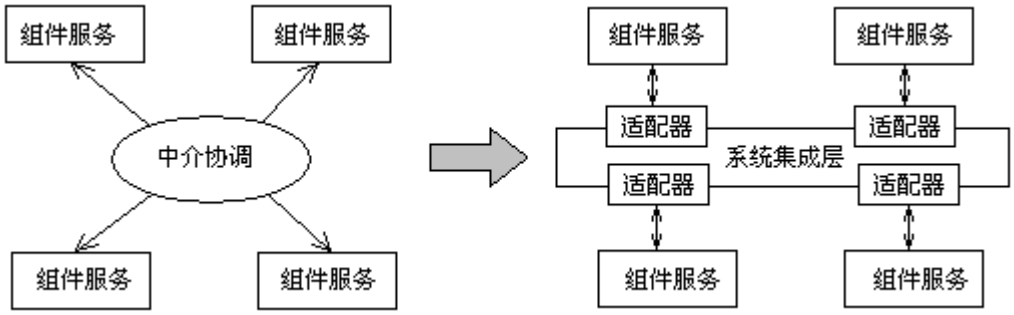
架构重构期间，我们会发现一些过分使用集中式设计的情况。当对等系统实体之间的交互顺序比较复杂的时候，适当地使用一些中介协调器（Mediator）不失为一种好的选择。但是，这样集中管理的架构方式，对系统的可扩展能力、可修改能力有着明显的抑制作用。因为每增加一个新的系统实体，我们就必须修改一系列的中介协调器（Mediator），这明显非常不利于系统的维护。

解决方案：

为了替代集中管理的中介方式系统设计，可以采用如下办法：

- 在系统架构中增加一个“系统集成层”，系统实体（例如：构件或服务）完全是以插件的方式集成在这个“集成层”中。
- 将这些对等实体的调用逻辑转移到该集成层之外的其他系统代码位置当中，系统通过“系统集成层”来调用相应的构件或服务。

举例：将中介方式的架构设计转换为适配器方式，明显增强了系统的可扩展能力，如下图所示。



5，合并子系统

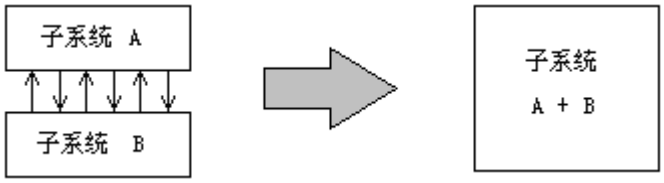
问题：

架构重构时，我们经常会见到一个系统内的各个子系统之间存在着紧耦合关系。根据高内聚低耦合的原则，一个系统中的各个子系统之间，在架构时应该尽量保持相对的松耦合关系；同时，一个子系统内的各个构件之间可以保持紧耦合关系。如果子系统间的耦合关系太高，对系统整体架构而言，系统进行维护和修改的代价就会非常高，并且重用一个子系统也会变得非常困难。这完全是由于紧耦合造成了子系统间复杂关系的结果。

解决方案：

一个大规模系统中存在一些子系统，这些子系统间或多或少存在着相互耦合的关系。两个子系统间的这种紧耦合关系，其实就意味着这两者是为了同样的功能目的而协作。这时可以考虑将这两个子系统进行合并。

举例：与其将紧耦合分成两个子系统，倒不如将其合并。这符合“子系统间松耦合，子系统内紧耦合”的设计原则。



6，强化层间调用

问题：

如果我们重构系统，经常会发现出现跨层进行层间调用的现象。这种设计的出现，完全打破了 Layer 架构风格的优势，很容易造成架构的混乱。

解决方案：

目前系统架构经常会采用 Layer 的架构风格，这样分层的结构非常适于系统功能的切分，使系统结构清晰，从而方便了未来的维护。当出现越层调用的情况，需要重新构建层间关系，使系统保持严格逐层调用的架构风范。

注意：解决越层调用并不是一件非常容易的事，有时甚至需要进行商业逻辑重构才能解

决。

7，以消息通信替代远程方法调用

问题：

重构系统时，经常会发现大量使用远程方法调用（RPC，Remote Procedure Call）方式进行通信，这无疑是一种最常见的通信实现方式。即便是在前端应用层可能需要的是一个异步的通信方式，但是后端通信层的实现往往还是利用 RPC 同步方式进行实现。

然而，在一个分布式系统中，如果存在基于事件的异步通信需要，最佳的通信机制还是采用 Message 形式的中间件方式。

解决方案：

分布式系统中异步通信方式是一种非常有利的通信机制。系统通信方式从同步机制向真正的异步方式进行转变，可以采用渐进的办法来重构：对现有系统中仍旧需要同步通信方式的，保留这种架构方式。同时，在系统架构中增加一个“消息中间件层”，把允许异步消息传输的通信转向这个消息中间件层。

举例：两种通行方式并存的局面，会随着后续功能的修改和新功能的加入，逐渐修改为依赖于“消息中间件层”的异步通信方式。

8，以缓存方式优化资源利用

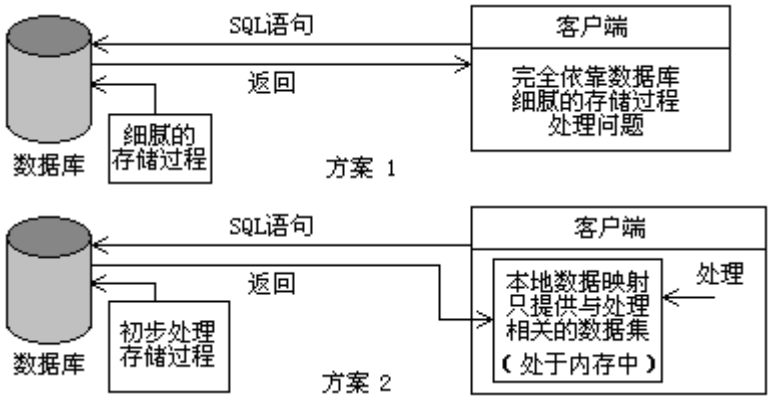
问题：

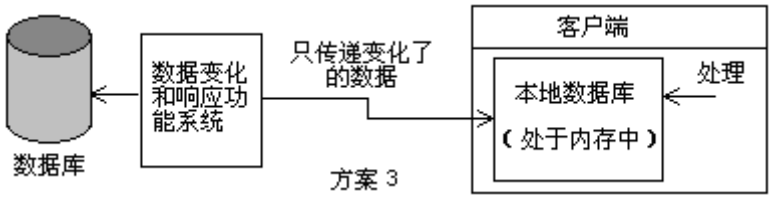
重构系统时，经常会发现系统性能存在着严重的问题。进一步分析就会发现，系统性能的瓶颈正是因为大量的系统实体对一个共享资源进行频繁存取所致。因为资源每次存取都会经过资源创建、分配等一系列非常消耗性能的动作。如果提高了资源存取的效率，系统性能就能够明显得到改善。这在一些实时系统的设计方面有着极为重要的意义。

解决方案：

系统中往往有很多共享的资源，在资源访问方面尽量优化，是克服系统性能问题的一个重要手段。利用 Caching 来缓存利用率较高的资源，同时定义一个适当的资源分配原则来使系统各个实体能够共享该缓存的资源。对于实时系统，采用基于内存的所谓“内存数据库”是一个比较广泛采用的方法。

举例：在子系统当中，有意识地将访问频繁的资源暂时存放在访问速度较高的缓存 Cache 中，从而避免了不停地进行资源创建这种消耗系统性能的动作。我们可以来看看三种数据访问形式，并比较它们的优缺点。





9，避免构件接口的膨胀

问题：

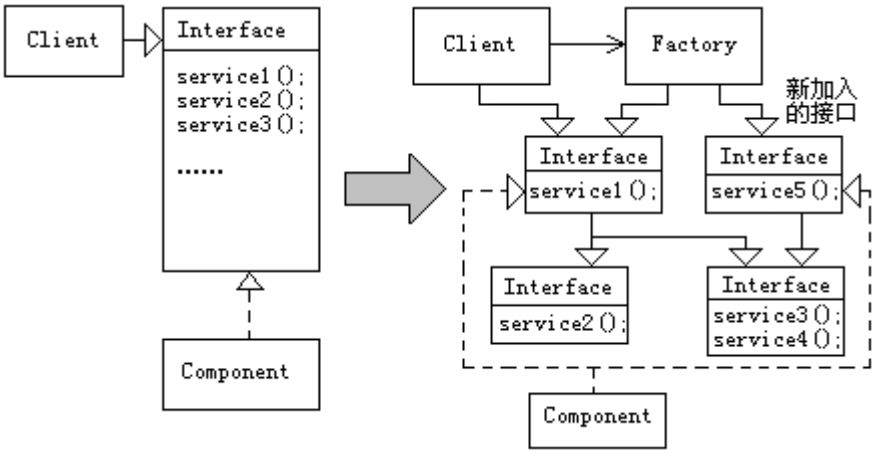
现实中，一个软件应用，经常会出现利用接口这样的抽象方式，来尽可能地使该接口的功能抽象满足各种各样的应用要求。一个经历了数年维护的系统，随着新功能的增加，系统构件的接口也会随之变化，这就导致了构件接口的膨胀。

试图满足各种要求的接口设计，会导致系统的可用性、可维护性、可管理性明显降低。最严重的情况是，这样膨胀的接口可能会导致系统代码的混乱。

解决方案：

我们必须承认这样一个事实，那就是构件的接口（Interface）是会膨胀的。可以考虑引入一个接口的层次结构关系，再提供一个公共的接口导航方式，以便客户端代码可以在一个层次结构的接口关系中寻找相应的功能实现。

举例：将一个膨胀的构件接口重构为层次化的结构，并提供寻找具体功能实现的导航机制，如下图所示。



10，保持架构和设计的对称

问题：

在初步设计的系统中，我们经常能够看到有些子系统的消息处理机制是使用“观察者设计模式”，将消息与消息处理进行衔接。但是有的子系统却又使用 Hard Code 的方式，大量地使用判断代码将消息与消息处理以编码的方式捆绑在一起。有些构件设计只照顾了资源的获取，而在构件任务结束后没有释放资源。这些都是现实中我们经常能够看到的所谓“不对称”的架构和设计方式。

结构匀称的系统，是指在系统全局范围内，在各个子系统、构件层面，尽量利用相同的架构概念和设计手段来解决那些相似的问题。这样的匀称，对维护人员理解系统结构非常重要。反之，一个系统如果使用了不对称的架构和设计，往往会造成系统概念的混乱，从而使后续维护非常困难。

解决方案：

如果系统架构是以对称的方式来匀称地组织系统内结构和关系，在系统维护期间，将非

常有利于问题的识别和修改工作，其方法为：

- 系统重构期间，首先需要识别系统中那些重要的战略和战术层面的设计抉择。
- 将这些设计抉择背景相似的汇总为一类。然后对相同类型的设计抉择统一考虑使用相同的架构原理或设计手段。

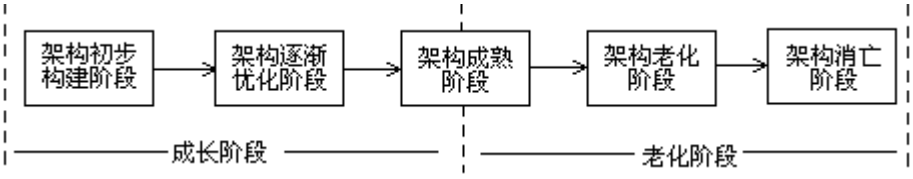
举例：将相似背景的设计抉择点汇总，并采用相同的 Observer 机制来进行事件的分发和处理。可能是一种更加对称的方法。

软件重构是架构的一个优化过程，也是一个敏感而且风险比较大的过程，所以必须遵循小步前进、一个时间只解决一个问题的原则，逐步的使架构得到优化。

3.8 软件架构的恢复

大量的软件工程实践告诉我们，一个软件系统从开始研发到最终消亡的整个生命周期过程中，前期的架构、设计、编码、测试所付出的成本只占所有工作的 20%~40%。事实上绝大部分的工作重点，需要花费到已经投入生产系统的后续维护上。很多产品研发的过程，大部分任务可能就是在一些成熟产品或框架的基础上，从事修改、增补等维护方面的工作。由于大量的技术架构是由最初的架构和设计团队来完成的，并且产品维护历史有了很长的跨度，结果就可能会出现许多问题。

软件架构的重复研发是一个很大的问题，国外很多大型软件企业，其发展历史非常久远，产品演化的周期非常漫长，因此在系统恢复和重构问题上积累了丰富的实践经验。让我们首先看一下一个软件架构的生命周期，如下图所示。



什么情况下可以判断架构已经进入了老化阶段呢？如果已经出现了如下现象：

- 系统的文档已经部分丢失和残缺。
- 研发团队对系统的认知非常有限。这包括缺乏系统架构、系统设计、系统实施、系统功能、系统覆盖的关键业务、系统测试用例等。
- 每当修改系统中存在的 Bug 时，莫名其妙的新 Bug 就会屡次出现。
- 每当试图增加新的系统功能时，整个系统的集成运行总会发生意想不到的问题。
- 如果仔细分析当前系统代码，会发现代码之间的关系非常混乱且难以维护。

一般就可能意味着产品已经开始从成熟期向老化期过渡，系统已经处于“架构老化阶段”。解决这些问题的办法，就是进行产品的“系统架构恢复和重构（System Architecture Recovery and Refactory）”。一个系统架构恢复和重构的主要目的，首先是把现有系统已经含糊的架构和设计进行梳理并恢复到一定清晰的程度。然后进行架构重构和优化，以便于日后的扩展和修改等维护工作。通常，恢复和重构可以从下述几个阶段来依次展开：

- 确立反向工程与正向工程的概念。
- 架构和设计恢复。
- 架构和设计重构。
- 系统代码重构。

为了说清楚这个问题，我们首先需要对架构恢复中的重构进行定义。

一、架构恢复层面的重构技术

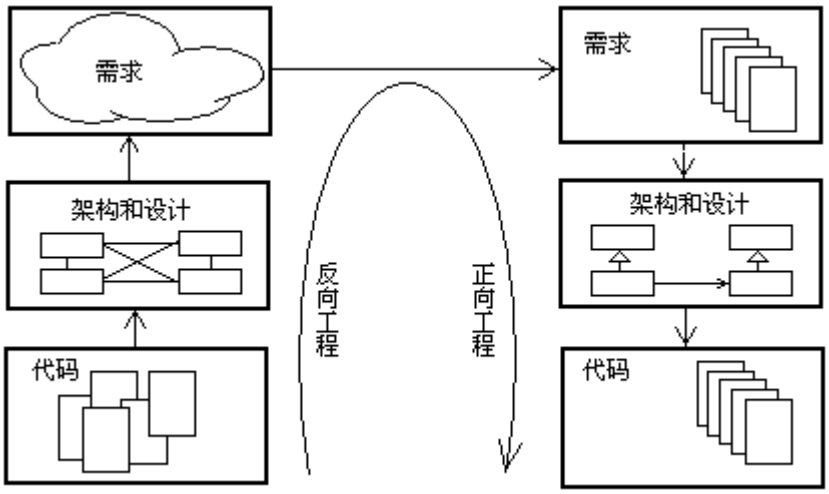
我们已经多次谈到了软件重构技术，在架构恢复过程中，我们大量使用的基本技术仍然是软件重构，为了延缓一个系统产品不可抗拒的衰老趋势，我们可以采取如下步骤。

- **确立反向工程与正向工程的概念**：要在团队中正确的建立反向和正向工程的理论概念，这样才能很好地帮助目标的界定和实现。
- **架构和设计恢复**：进行架构和设计恢复工作，有效地保证了我们的团队是工作在一个清晰并认知一致的架构基线上。
- **架构与设计重构**：为了改变现有架构中存在的那些不能满足维护要求的设计问题，进行架构和设计的调整是非常有必要的。
- **软件代码重构**：原先的代码是不允许完全抛弃的，我们可以通过代码重构来合理地重用那些有用的代码。

下面依据这个步骤，对各个步骤中的问题与解决方案作一个讨论。

二、反向工程和正向工程

系统重组（Reengineering）是架构恢复和重构的核心概念，也是最复杂的一个总体概念。下图表达了一次完整的系统重组过程，我们可以看出来，过程被划分成了两个主要阶段：反向工程阶段和正向工程阶段。



1）反向工程阶段

反向工程（Reverse-Engineering ）阶段主要包括以下活动：

- **系统分析和架构恢复**：该活动的主要目的是能够使我们全面掌握当前系统的架构基线，即主要回答这样一个问题：我们面对的是一个什么系统？
- **SWOT 分析**：针对当前所面对的系统，从架构的角度来分析其优点、弱点、改进机会、面临威胁等。
- **进行抉择**：前两个活动分析的结果，会成为我们进行抉择的输入信息。现在，该是我们抉择系统中哪些部分该保留，哪些部分该修改，而哪些部分可以彻底抛弃的时候。

2）正向工程阶段

反向工程任务完成之后，我们已经完全明确了当前系统代码的功能及关系、系统架构和设计的结构、满足的主线业务情节等重要信息。然后，我们会进入到正向工程阶段，其主要步骤如下：

- **建立新的架构远景**：为崭新的系统确立架构远景，其中包含了以往可重用的架构组成部分及计划添加、修改、替换的部分。

- **建立新的架构基线**：遵循“将可重用的或新的架构部分逐渐组装进系统架构当中，然后立即进行重构”这样的原则，采取逐渐提炼和优化的风格，进行架构和设计的重构，并最终逐步形成崭新的架构基线。
- **进行代码重构**：遵循“将可重用的或新的代码部分逐渐组装进系统代码当中，然后立即进行重构”这样的原则，采取逐渐提炼和优化的风格，进行代码的重构，并最终逐步形成崭新的系统代码基线。

三、架构和设计恢复

1，架构和设计恢复的整体概念

所谓架构和设计恢复，指的是利用现有的资源（现存的代码、文档、测试用例等）来抽取关键的架构信息，从而恢复一个系统原有的风貌。只有在这样的基础之上，维护人员才可能完全理解一个系统的当前状况，为未来计划中的系统修改做好铺垫工作。从反向工程的概念来看，架构和设计恢复就是反向工程的核心任务。

但是，单从这样一个经典定义，我们还没有办法知道到底如何计划、管理和执行架构恢复工作。我们回忆一下就可以发现这样一个现象，目前我们对系统研发过程的研究，大部分时间局限在正向研发时的方法论上，往往忽视了一个系统进入维护或运营阶段所经常出现的架构恢复的要求。但是，情况正在悄悄的改变，目前架构恢复过程的研究开始逐渐展开。最近十几年的探索过程基本上集中在下面几个层面上：

- 系统概念和需求恢复。
- 系统功能恢复。
- 系统架构和设计恢复。
- 系统编码实现恢复。

很明显，这四个层面间的次序，是一种逐渐深入、逐渐细化的层次关系。这也就意味着，在不同的层面上，可以进行不同程度的系统恢复工作。在我们进一步探讨如何进行一次架构和设计恢复前，需要先将这四个层面进行一下说明：

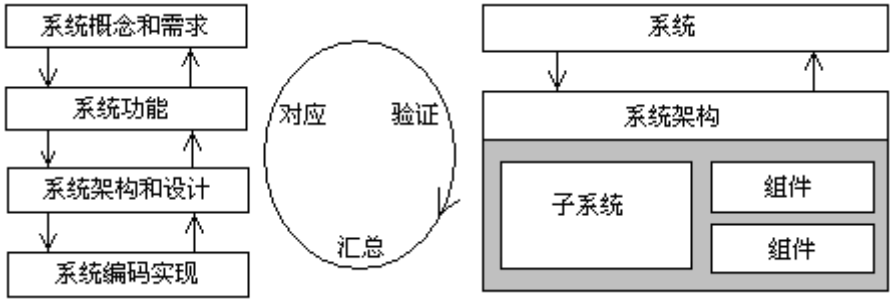
系统概念和需求恢复：系统概念可以从两种视角来衡量。如果假定我们是一个系统的用户，那么一个系统到底能帮助我完成什么工作？这就是系统用户建立起来的对一个系统的概念。如果我们从系统架构、设计，甚至是编码人员的视角去看一个系统，系统概念就会成为一个个主要的功能点。同时，这些功能点是怎样协作来完成要求的任务也是需要建立起来的系统概念之一。系统需求，则代表了用户指定的系统必须完成的任务以及必须满足的质量方面的要求。

系统功能恢复：是遵照客户所指定的需求，经过设计和实施后系统所具备的能力。

系统架构和设计恢复：为了满足所要求的功能而构建出的系统组成结构及关系，这时可以利用的材料主要是原先系统架构设计中的文档、代码、模板等。

系统编码实现恢复：是一个系统设计经过编程语言最终实现的过程。这时可以利用的材料主要是原先系统编码实现时产生的源代码文件、开发文件目录、培训材料、用户手册等。

如果我们进行一次系统架构和设计的重构，通常会需要把上述四个层面的核心内容完全恢复出来。如果只是完成了其中一两个层面的恢复工作，将仍然无法勾勒出一个系统的全貌。没有这样的全貌，就根本无法满足我们后续系统维护的要求。所以，我们需要一个更加系统化的架构恢复方法。一个比较理想的架构恢复过程如下图所示。



可以看出，我们其实最需要的系统架构恢复过程，是一个覆盖了从“系统概念和需求”到“系统编码实现”四个层面，并且经过了恢复信息的汇总、对应和交叉验证的过程之后，最终恢复成为一个完整的系统架构的系统化过程。

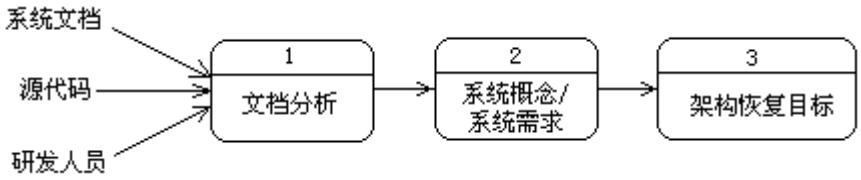
现在，一个架构恢复的总目标和大致恢复过程的轮廓已经确立了。那么为了实现这样一个理想中的系统架构恢复，我们应该怎样进行实际操作呢？从过程的角度上来讲，我们可以进一步将架构恢复划分为这样五个主要过程阶段：

- 系统概念和需求恢复阶段。
- 系统功能恢复阶段。
- 系统架构和设计恢复阶段。
- 系统编码恢复阶段。
- 汇总对应和验证阶段。

下面对每个过程阶段加以说明。

2，系统概念和需求恢复

“系统概念和需求恢复”工作是系统架构恢复的第一个阶段。该阶段的主要目的，是以当前可以搜集到的那些系统相关的文档（例如：系统的用户使用手册等）为出发点，最终帮助我们建立起对一个完整系统的概念和认知，其中包括系统服务的商业业务领域、系统提供的服务内容、系统的行为的轮廓、用户在该系统初创时需求方面的要求等，从而在一个架构恢复正式开始前，明确地界定此次架构恢复的目标。



从图中可以看出，该阶段的工作基本上由四个主要的活动顺序组成：搜集系统当前保留的信息、进行初步系统信息分析、建立系统概念和系统需求、与管理层一起确立架构恢复目标。

1）搜集系统信息

一个系统从开始创建，经过了数年的连续维护，必然会有一些重要的文档或信息保留了下来。作为后续架构恢复的基础，这些记录过去历史的可视性信息将有助于回答诸如“这个系统到底是什么”之类的问题。

系统信息搜集，就是要通过各种渠道，去搜集和汇总那些与系统相关的信息，例如系统源代码、架构和设计文档、需求说明书、测试计划、集成和系统测试结果、测试用例等。除了研发期间的文档，那些系统维护期间的文档也需要进行汇总，例如系统维护记录、系统维护手册等。从客户使用的视角来看，用户使用手册等信息也非常有助于我们理解该系统的商业功能。如果该组织的知识管理做得非常好，那么这些信息的搜集还是非常容易的。

为了快速建立起对该系统更加深入的认知，与系统的利益相关人员及研发人员的沟通也

是一种非常有效和直接的方式。如果条件允许的话，我们还可以利用各种方式，与当年的系统研发人员取得联系，然后进行简短的沟通，这无疑会对我们目前的工作有极大的帮助。

在大多数情况下，我们无法找到当年的研发人员。这时，从事系统后续维护工作的架构和设计人员、编码和测试人员等，就是我们可以重点走访的对象。从他们那里，我们肯定可以听到许多更加深入和直观的有关当前系统的论述（有时甚至是抱怨）。

系统所保留的历史信息、维护人员的访谈信息，都成为下一步我们分析和理解的基础。

2）系统信息分析

经常遇到的尴尬是，我们手中能够掌握的有关系统的信息，大多数是有关于如何完成某些特定要求或功能的文档，例如：系统设计最终应用了一个怎样的算法来解决系统资源的调度问题。虽然这类信息在一定程度上回答了一些系统相关片段的“How”问题。但是，当前阶段我们的首要任务是要明确“整体是一个怎样的系统”，即我们需要知道的是整体系统的“What”问题。

所以，我们需要以这些可能不完整的系统信息文档为出发点，通过逐渐分析、汇总及拼凑，最终需要回答“这是一个什么系统”的问题，也就是说，我们需要专注于系统的问题域，并且认真体味当年系统设计者处理这些问题的思维过程。

这样分析的结果，会在一定层面上帮助我们建立起该系统一个比较宏观但完整的概念：例如该系统能够提供的主要商业服务或商业功能（我们目前并不关心这些服务在系统中是如何实现的这样的问题）、系统运行时的主要行为、系统的主要组成元素（子系统、主要服务或构件等）、系统物理的分布、系统组成元素的职能和相互关联、系统与商业组织运作的结合等。

经过这样的分析，一个完整的系统概念已经在我们的脑海中建立了起来，我们已经在很大程度上清楚所面对的是一个什么系统。

3）系统概念建立

通过上面分析的结果，我们已经明确了系统的整体轮廓。现在，需要将这样的系统概念以适当的方式描述出来。毕竟，头脑中建立起来的概念可能并不是一个可视的、系统化的描述。我们需要以文字、图表来叙述该系统存在的商业背景、主要功能及商业目的、系统主要输入和输出、系统提供的用户界面等。

这种书面的方式，更加适合于今后工作中的交流，并且该过程本身也是一个不断完善的系统化工作。

4）确定目标

在具体展开一次系统架构恢复前，还需要根据目前所构建出的系统概念，与管理层一起确定架构恢复的目标和范围。多年的的实践告诉我们，如果没有这样一次范围限定的活动，势必会造成恢复工作持续时间长、目标和范围模糊、恢复效率低下等问题。

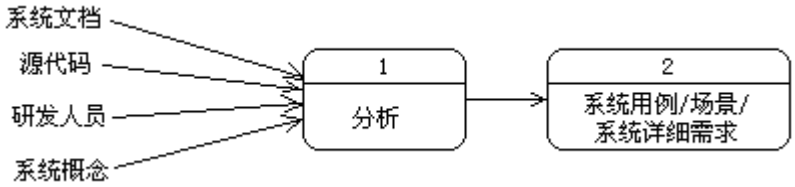
对一个大规模复杂系统的反向工程，尤其需要进行时间和范围的限定，否则会导致一次反向工程中的产出物甚至比你要进行架构恢复的系统所拥有的文档数量还多，这显然适得其反。

进行目标和范围限定时，一定要以目标为导向，结合前期系统的整体“图像”来界定此次架构恢复的范围。例如：此次架构恢复是要将该系统的功能进行全面的恢复，从而明确该系统所涵盖的各种功能以及相应功能的应用主线场景和各个可能的分支；如果是对系统中某个子系统的架构和设计进行恢复，我们主要就会利用该子系统的源代码或设计文档来进行进一步的工作。

3，系统功能恢复

系统架构恢复的第二个阶段，就是要完整地恢复一个系统所覆盖的各种商业业务功能。系统功能的全貌，有助于我们理解一个系统为什么会进行不同的架构和设计抉择。同时，系

统功能的理解也帮助我们对当时系统构建时客户所提出的需求进行分析和汇总。这样，在很大程度上，我们将一个系统的用户需求和系统功能进行了完整的映射，如图下所示。



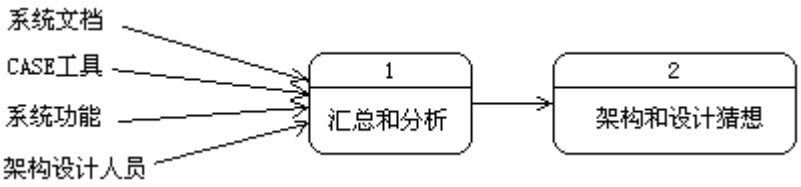
系统功能恢复与系统概念恢复阶段相比，是一个更加深入的分析过程。通过文档、源代码、系统测试用例、系统运行界面的输入输出、系统的数据流、系统控制流、相关人员的访谈等各种方式的结合，试图进一步细化并完整汇总系统所支持的用例（Use Case）；同时继续细化前期获得的部分系统需求，使用户需求更加丰满。

仅仅是系统用例，还不能够复原系统的全貌，需要逐步完善系统所支持的各种业务和系统场景。只有将系统的功能细化到支持场景的程度上，才能真正在细节这样的层面上完善系统功能的原貌。

4，系统架构和设计恢复

基于我们已经建立起来的系统概念、搜集汇总的用户需求、细化和分析的系统功能这三类主要信息，再结合当前遗留的架构设计文档，我们可以开始进入到系统架构和设计恢复阶段的工作了。

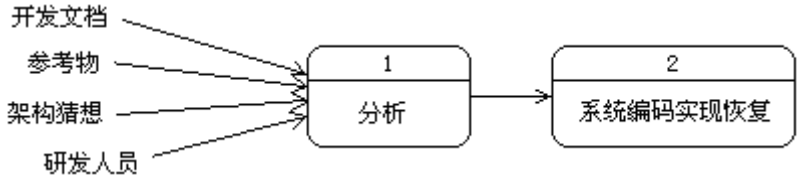
系统架构和设计的恢复是一个滚动添加和修改的过程。之所以用“滚动”来描述架构和设计的恢复，就是因为架构和设计的恢复其实是一个猜想、验证、修改的循序渐进的过程。但是，作为“滚动”的第一步，一个合乎逻辑的系统架构和设计猜想是走向成功的基础，如下图所示。



架构和设计恢复的核心工作，就是一个架构猜想的建立和验证的过程。首先，通过与维护工作人员中涉及架构和设计的技术人员的沟通，结合系统设计文档的进一步深入分析，已经在很大程度上具备了做出一个系统架构和设计猜想的条件。这时我们再结合一些计算机辅助软件工程 工具，就能从系统源代码中抽取那些与架构和设计相关的结构方面的信息。例如：系统的接口信息、构件间的调用关系或构件执行顺序、系统的数据流、系统的控制流、代码执行顺序等。这样就逐步形成了一个完整的、符合商业需求及当前收集的系统功能要求的系统架构和设计猜想。

5，系统编码实现恢复

系统编码实现方面的恢复是最低层面的系统恢复工作。根据前期建立的系统概念、汇总的系统功能、系统架构和设计的猜想，我们可以开始重现当前系统代码级的原貌，如下图所示。



系统代码级的恢复，主要是将系统代码的结构和关系进行恢复。

举一个经常出现的例子，例如一个经过多年维护的系统，系统代码经常会出现代码目录结构混乱（应用程序的代码包经常和中间件或编程框架的代码混杂在一起）的问题，以至于维护人员为了编译一个小小的修改，需要将很多无关的代码添加进来，才能最终编译成功。

通常，系统实现代码间的结构及相互之间的关系，可以从下面几个方面得到有用的线索：

- **开发人员的文档**：在开发人员的详细设计文档和源代码中，经常有很多有用的解释信息，这些信息有助于我们理解代码间的组成关系和协作行为联系。
- **代码调试**：核心代码的调试会有效地帮助我们理解系统实现代码间的执行顺序和代码的各种异常分支，这也是搞清代码结构关系的重要方式。
- **系统参看物**：可视的用户界面变化（例如页面前后次序）、系统输出的报表、系统与外部交互的过程等，都是我们对代码执行期间的动态关系进行推敲的工具。
- **辅助工具**：有些反向工程工具可以帮助我们解析系统代码，抽取一些静态信息（例如：Attributes 及功能方法），或者形成构件结构视图和依赖关系视图，这在很大程度上也助于我们的分析。

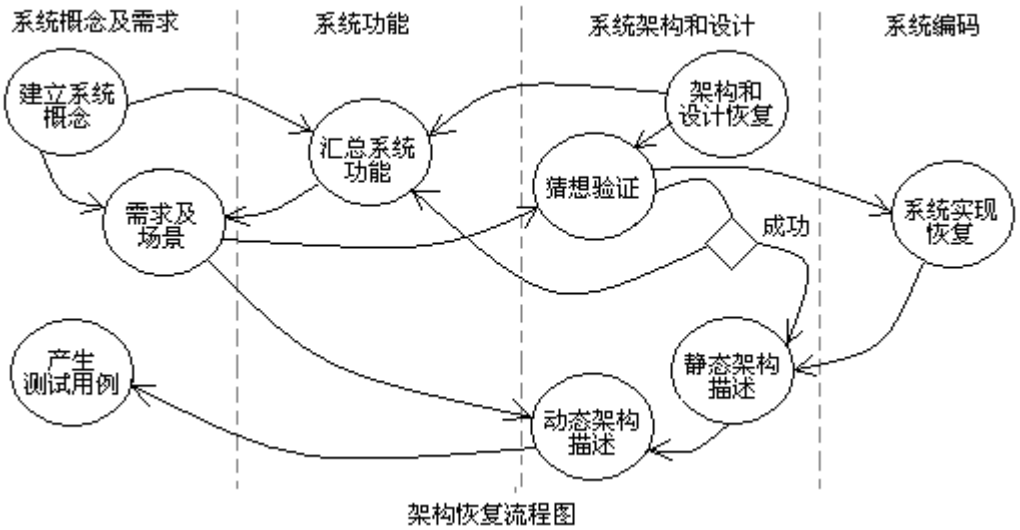
在系统编码实现级完成了恢复工作后，一个完整的代码级系统静态视图将会展现在我们面前。这样的视图无疑在后续系统代码维护时期成为我们重要的参照依据之一。

6，汇总对应和验证恢复

一旦上述四个层面的恢复工作各自完成到一定的深度后，就需要进入一个汇总、对应和验证的过程中来。这是因为，我们需要将相对割裂的四层的工作结果进行关联。例如：系统功能的描述，需要对应到那些相应的实现代码上。而那些实现代码，需要对应到架构和设计的某个模块或构件上。同时，这样的对应往往可以帮助我们更正系统架构和设计猜想中那些与实现代码完全脱节的错误或遗漏。

作为系统恢复的最后一个动作，还必须将恢复过程中的各种文档进行汇总和整理，并且形成系统化的互联关系，从而通过“系统恢复文档集”的方式，将一致的、相互联系的、真实的系统原貌展现给系统维护方。

系统架构和设计恢复是一个不断递增的、系统化的复原过程。作为一个完整的流程，我们可以参考如下图所示的动态表现方式的系统架构恢复流程，再根据自身的要求，来进行范围明确的架构恢复：



架构恢复流程图

作为一个完整的架构恢复流程，从上图中我们可以看到，除了需要构建一个架构的静态结构描述外，架构的动态描述也是非常重要的辅助信息。构建架构的动态描述，非常有利于我们创建一些测试用例。这样做的目的，就是使用当前的运行系统来检测我们系统恢复的质

量。

虽然架构和设计恢复是系统反向工程中的核心任务。但是，除了进行架构恢复外，通常我们还需要在恢复系统原貌的基础上，进一步分析当前系统结构上的优缺点及改进机会。架构恢复不仅仅是为了恢复原貌，也不仅仅是为了使架构更容易维护，更重要的是一次创新思考的机会。

例如：哪些系统组成部分已经不能继续使用、哪些系统组成部分需要进行修改、哪些接口需要重新制定、哪些是坏死部分并且已经完全没有重用价值等。只有这样一个系统反向工程的结果，才真正能够帮助系统维护管理层进行未来系统改进时的抉择。

四、架构恢复阶段的设计重构

前期反向工程中的架构和设计恢复，已经把一个系统的原始风貌完全还原了出来。这样的一个系统架构和代码的基线，将成为下一步我们进行正向工程的良好起点。换句话讲，反向工程虽然做了很多分析工作，但是不会对既有系统做任何改动。从正向工程开始，就是我们大胆进行系统变革的时刻。

正向工程中的核心任务主要由两个方面的重构动作组成：

- 系统架构和设计的重构。
- 系统代码的重构。

应该注意到，目前大多数软件人员依然把重构的重点放在了系统代码上，而忽视了架构和设计重构的重要性。作为一个系统架构师，除了应该重视代码重构外，更应该重视系统架构和设计重构的研究和实践，

系统架构和设计重构，有一点与 Martin Fowler 的系统代码重构的表达相似：如果我们纵览一遍老系统的代码，直觉上就能够嗅到很多系统代码的“坏味”。类似地，当我们纵览一遍恢复出来的系统架构和设计时，同样也能察觉到很多系统架构和设计的“坏味”，例如：

- 架构设计严重违反了“不能重复自己”（DRY，Don’t Repeat Yourself）原则。
- 系统元素命名混乱，无法分辨其在系统内的职责和角色。
- 大量使用中央集中处理的架构机制。
- 系统中各部分相似的问题却使用不同的设计方式来处理。
- 无法顺利使用子系统内的部分功能。
- 子系统或构件的结构不断膨胀。
- 直接暴露子系统或构件的内部功能。
- 分散的系统配置，造成了手工逐点配置的大量工作。
- 系统资源访问方式不佳，严重影响系统性能。
- 系统内同步和异步通信方式使用混乱。
- 系统元素间没有形成双向联系，无法互相调用。
- 系统接口的“契约”没有严格定义，造成调用子系统和构件的行为没有严格按照接口的“契约”执行。
- 子系统间功能划分不合理，造成子系统间紧耦合。
- 系统结构逻辑没有严格按照“层”的架构风格设计。

架构和设计重构的核心任务，其实就是不断地去否定以往的系统架构和设计的原则，力图找到架构和设计中的“坏味”，然后在不改变系统行为的基础上，调整系统的内部结构，最终使其达到优化的目的。这里应用的一些重构模式，这些模式与架构设计阶段的要求是一致的。

重构经常是牵一发而动全身的一个改造动作。所以，在实际操作过程中，应该尽量采取小批量递增的方式，逐步进行系统设计和代码的改善。力图一次就完成一个翻天覆地的变化，

一定会出现适得其反的结果。

完成了正向工程中系统架构和设计重构、代码重构、文档重构、流程重构等这样的工作后，正向工程的主要任务就已经圆满完成了。

为了将一个维护多年并且现状混乱的老系统翻新改造，延长其生命力，我们已经走过了从反向工程到正向工程的艰难过程。真可谓天下没有免费的午餐，付出了巨大的代价，才能完成这样一个治疗老系统的任务。

3.9 架构评审与决策

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。

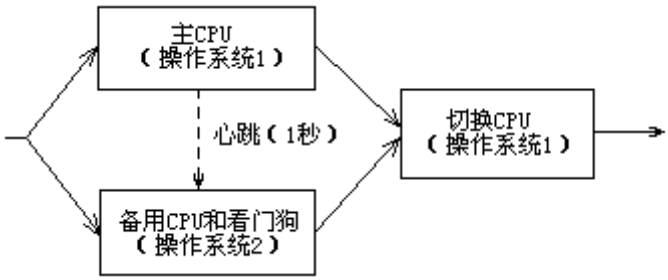
评估小组的目标是，确信这个方法能够达到质量属性的要求。

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	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）。 最坏情况下，4 秒钟（运算状态的最长时间）内完成恢复。 根据心跳和看门狗的速度，保证 2 秒之内检测到故障。 看门狗简单可靠（已经过验证）。 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）
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的时候，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。

3.10 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，并且着重于描述高层需求，是架构设计成功的必要条件，因此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。

第四章 软件架构的模型驱动与演化

模型驱动的开发可以保证关注点前后一致，用例不仅仅是一种需求方法，而且是设计、测试的驱动因素。架构的演化和重构，是架构整个生命周期中必须认真研究的问题。

在一个迭代周期内，我们必须要注意到经典的软件工程学原理不但适用而且是必须坚持的。所以一个迭代周期将贯穿整个分析、设计、编码、测试和集成过程。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。为了保证关注点的前后一致，模型驱动的开发是个很好的理念，这种理念会使整个开发变得非常有序而且有效。

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”。

软件架构设计并不仅仅发生在项目的早期阶段，在整个项目过程中都可能迭代的考虑架构优化的问题。尤其是在敏捷过程中，每个迭代周期中都可能发生功能和非功能需求的收集和设计，我们的目标是使这个过程具有流畅之美，确保整个项目能够按要求完成。

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，面向过程、面向对象、面向构件以及面向方面的方法并不是相互矛盾甚至相互对立的，而是以各种思维方式和技术手段，研究处理关注点独立性的方法。研究整个软件开发过程中的架构设计，对于提升软件整体的质量是非常有意义的。

4.1 产品用例的细化分析

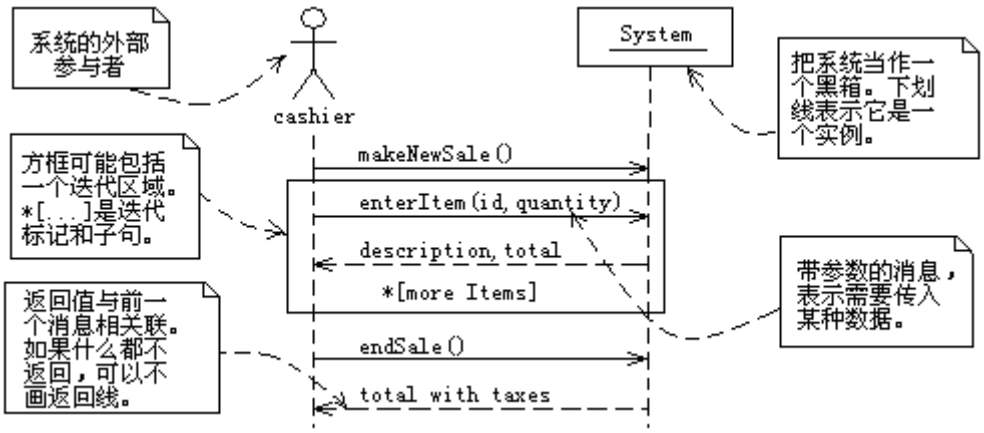
一旦选中了部分用户描述进入迭代过程，为了确保设计和编码产生完全符合需要的系统，就要对选中的描述进行细化分析，这实际上是一个细化的需求分析过程，会产生更多的需求信息，必须有效的组织和处理它们。

一、从系统的角度研究事件及行为

产品用例的细化分析，首先应该从系统的角度研究外部参与者与我们将要创建的软件系统之间如何交互。交互期间，参与者产生一个发送给系统的事件，通常要求系统响应这个操作。为了更仔细的研究这些外部参与者与系统交互的过程，使用系统顺序图（SSD）是可取的，在这里所有的系统都被当作黑箱，图的重点是从参与者跨越到边界的事件。

根据分析的需要，在 SSD 中可以考虑每个相邻系统与系统的交互过程，从而发现和定义

尽可能多的事件，发现事件的粒度可以根据具体情况来决定。下面的例子是用 SSD 显示参与者与系统（作为黑箱）直接的交互，参与者所产生的事件。



产品用例不仅仅是把业务用例“翻译”成产品用例，更重要的是为设计提供想法，用例不仅仅是一种需求开发，还是一种用于驱动整个软件开发生命周期的软件工程技术。用例驱动是假定软件开发过程是模型驱动的，最简单的情况是包括概念模型、设计模型和实现模型，一般来说对于每次迭代，开发小组都需要最如下的事情：

- 寻找用例并详细说明它们。
- 设计每个用例。
- 设计并实现每个类。
- 测试每个用例。

我们前面已经讨论过用例场景的详细描述，我们已经知道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，易被用户理解的事件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备选流。基本的场景描述文档的内容如下：

- **用例名**：这是唯一的名称，应该反映用例的主要工作。
- **主要参与者**：请求系统提供一项服务的主要项目相关人员。
- **描述**：简要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。
- **扩展流**：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）。

关于前置条件和后置条件，可以参考下面一些指导方针。

- 前置后置条件所描述的状态应该是用户能观察到的。可观察的状态例如“用户已在系统注册”或“用户已打开文档”等。
- 前置条件是用例启动时候的约束，但不是用例启动时候的事件。
- 虽然你可以在某一个用例层次上定义前置和后置条件，但它们不是只限于一个流程。
- 无论执行哪个扩展流，后置条件都应该为假，它仅对主事件流成为真。比如在后置条件中这样概括：“动作完成，或者出现错误，动作未能执行”，而不是“动作已经完成”。

- 后置条件对描述用例来说是重要的工具，你首先可以定义用例要获得什么后置条件，然后再考虑如何达成这个条件（所需要的事件流程）。

二、子事件流

对于某些在同一个用例中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。下面是一个“酒店管理系统”的简单例子，主要表达的是对子事件流的引用。

用例名:	预定房间	**用例层次**
用例 ID:		用户目标
主要参与者:	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。	
主事件流:	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。	
扩展流：	5a. **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。 5a1，如果客户想继续，则系统开始新的预定，用例重新开始。 5a2，如果用户说明预定是重复的，用例结束。 5b……	
子事件流：	S1，**核定房间收费**。 S1-1，客户选择需要的房间类型，并说明将停留的时间。 S1-2，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。	

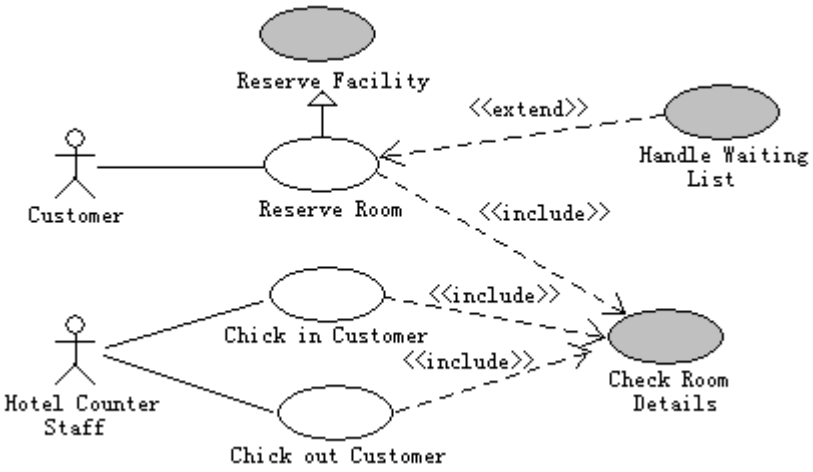
三、用例结构化及其文档描述

对于大多数用户和系统描述来说，上面的细化步骤再充分不过了，多数情况可以合理的确定、精化主流程和替换流程，确定前置和后置条件，为系统提供了充分而广泛地描述。但是随着应用的复杂性不断的增长和演变，就可能会遇到用例之间的结构化关系，下面我们来讨论有关问题。

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，并在线上标注购造型<<include>>：
- **泛化（generalization）**：一个用例的实现从另一个抽象的用例继承。
- **扩展（extend）**：扩展关联的基本含义与泛化关联类似，但是对于扩展用例有更多的规则，即基本的用例必须声明若干新的规则---扩展点（Extension Points），扩展用例只能在这些扩展点上增加新的行为并且基本用例不需要了解扩展用例的如何细节。

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



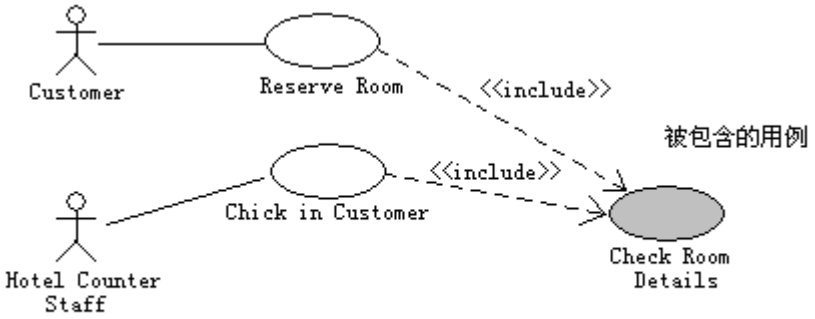
四、包含（imclude）

为了确保产品设计的品质，产品开发的可追踪性与回朔性要好，因此在用例设计的时候，我们希望关注点相互独立，其重要性也不能相互比较，这就是所谓对等关注点。例如在酒店管理系统中，预订房间（Reserve Room）、登记入住（Check In Customer）、结账离开（Check Out Customer）都是对等关注点。

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

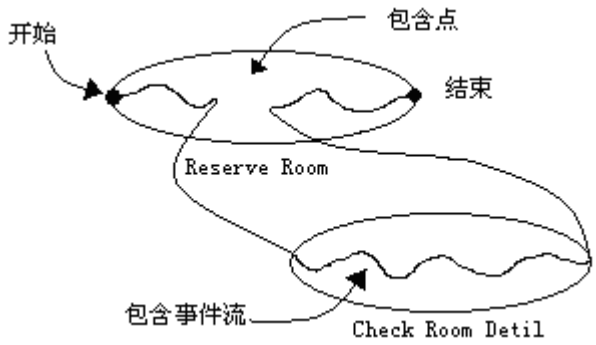
作为分析师，我们应该注意到问题越是前期发现和解决，总的风险就越小。所以设计师希望从分析的时候就做到使关注点相互分离。在分析的时候，对于不同用例的相似步骤，可以把这些公共事件抽取出来，放在被包含的用例中，其它用例可以引用这些被包含用例中的事件流。

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



用例“预定房间”包含了用例“核对房间详情”

它的执行过程如下。、



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

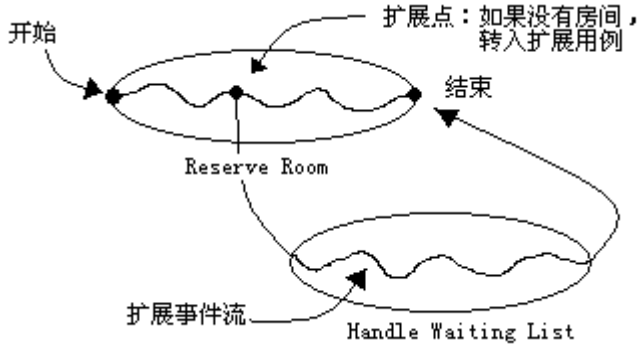
五、扩展（extension）

1，用例的扩展关系

随着系统随时间的不断演变，需要附加一些特性和功能来满足新的和当前的用户需要。假定有这样的情况，一个项目范围是团队一次迭代所能完成规模的两倍，你就需要砍掉一部分功能，把某些用例推到下一个周期完成。在第一个迭代周期完成一个主要的功能，下一个迭代周期完成一些扩展的功能。

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。

下图表示了这个扩展用例的存在发生了什么。



使用扩展用例的理由有三条：

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

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

- 用例对应着功能包，所以用例的大小要合适。

- 注意到缠绕状态，标识出并且提出建议方案。
- 研究并且标识出易变性，把易变的功能独立起来。

从这些角度说，产品用例是在业务用例的基础上更进一步的思考。

2，用例扩展关系的场景描述

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，例如在“酒店管理系统”中：

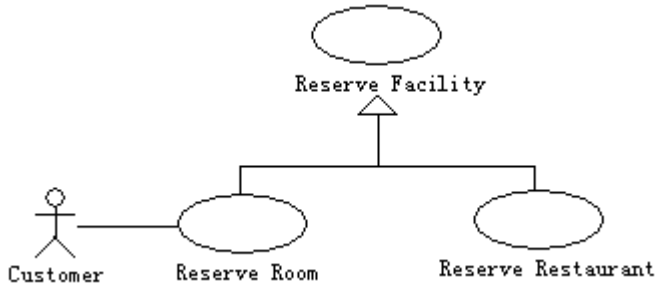
用例名:	预定房间	**用例层次**
用例 ID:	C-2	用户目标
主要参与者:	客户	
描述:	该用例描述客户如何预定一个房间	
前置条件:	客户已经登录到系统	
后置条件:	成功预定之后，创建一个新的一项记录，特定时间的可用房间数将减少，如果预定失败，数据库不会发生任何改变。	
主事件流:	1，客户选择预定一个房间。 2，系统显示酒店拥有的房间类型，以及它们的收费标准。 3，客户核对房间的收费（S1）。 4，客户对选择的房间提出预定。 5，系统在数据库减少这个类型可预定房间的数目（Ea）。 6，系统基于提供的详细资料创建一个新的预定。 7，系统显示预定确认号以及入住登记说明。 8，用例结束。	
扩展流:	……	
扩展点	Ea：更新房间可用性 Ea1，当没有客户所选择的房间的时候，该扩展事件流发生。 扩展事件流为房间预订队列（CS-5-EF1）	

用例名:	处理等候列表	**用例层次**
用例 ID:	CS-5	子功能
基本事件流:	……	
扩展事件流:	**EF1：房间预定队列。** 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。	

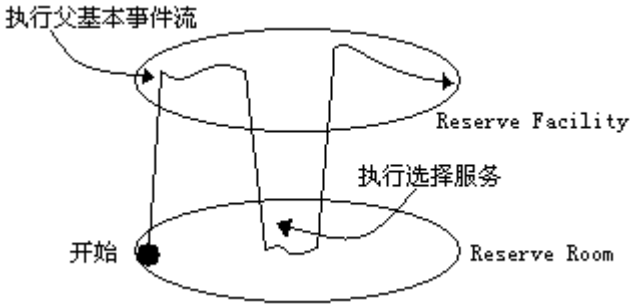
六、用例的泛化关系及场景描述

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。



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



事件流的描述如下：

用例名:	预定服务	**用例层次**
用例 ID:	C-3	用户目标
主事件流:	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，客户选择一项服务（S1）。 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。	
扩展流：	……	
子事件流：	S1：{abstract}选择服务	

用例名:	预定房间	**用例层次**
用例 ID:	C-4	用户目标
主事件流:		
子事件流：	C-3-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。	

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，泛化关系要求付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。

4.2 领域模型的建立

作为从需求到架构的第一个过渡，在具体进行架构设计之前，首先认真研究一下业务领域，并且建立领域模型是十分必要的。在现代软件开发中，领域建模已经是大家公认的最佳实践之一，领域建模决定了产品功能的范围，因此也影响着软件的可扩展性，但是，在领域建模的应用上，还是存在一些误解：

- **时间太紧是不是真有必要？**很多人并不否认领域建模的重要性，但是时间太紧是不是还是由需求直接进入设计及较快捷，甚至直接进入编码可能是更紧急的事务。这是要从思想上解决的问题，领域模型是对需求向设计的第一步过渡，也是可扩展性最重要的一步，这是不能绕开的。
- **需求的文字似乎没有办法直接过渡到视图**：这是比较常见的问题，我们应该注意几个情况：一方面，领域建模应该是业务专家（需求人员）与设计人员（架构师）交互的结果。另一方面，领域建模也是深入研究需求文档的过程，没有这种研究，后期的设计就可能走向一个错误的方向。

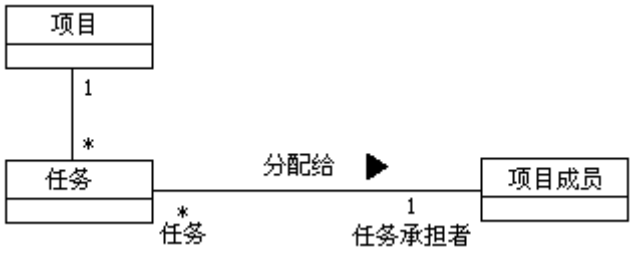
一、领域模型的初步建立

应该着重强调，领域模型并不是某个人冥思苦想做出来的，它需要领域建模小组有效地开展工作，通过讨论可以使设计人员对业务问题理解得更透彻。下面我们通过前面多次使用的“项目管理工具（PMT）”软件设计的简单例子，看看实际过程中是如何边讨论、边建立模型、边使问题的研究更深入的。

我们先从最基本的问题入手，逐步的建立静态领域模型。首先遇到的第一个领域问题，就是软件项目管理到底是怎样进行的？

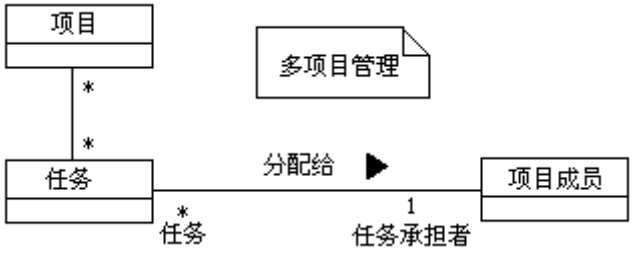
1）项目的开发特点是分解成许多任务，分配下去

2）每个任务只能分配给唯一的人，而一个人可以负责多个任务



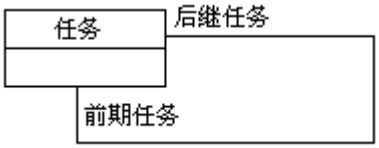
3）现在已经可以实践“多项目管理”，多个项目可以共享某各子任务

例如：很多企业级应用会用到一些与领域无关的模块，例如身份验证、消息机制等。恐怕我们的产品需要支持这样的特性。

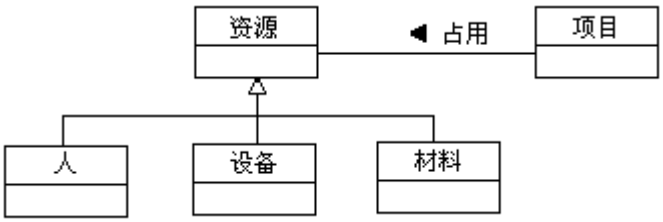


4）项目分解成多个任务以后，需要为任务排定日期

排定日期的时候还要考虑依赖关系。

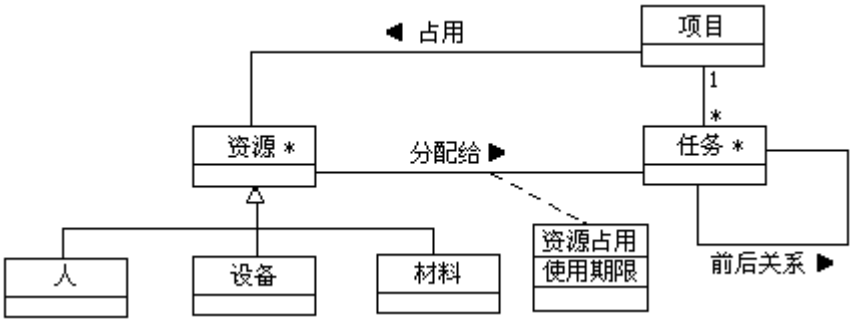


5）我们的项目涉及的资源有多种，比如人、设备、材料等

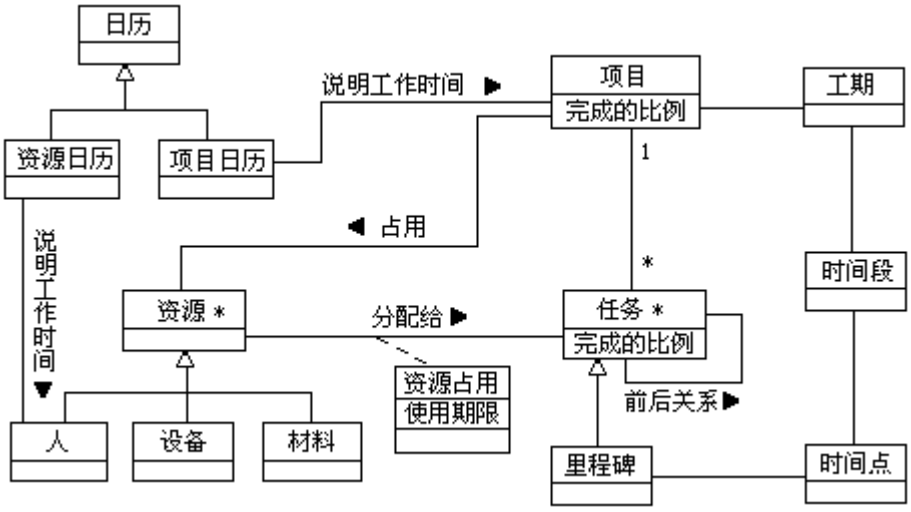


6）从逻辑上看，资源是项目占用的，但对资源的使用要具体分配到任务

我们已经明确了是任务在消耗资源，而且还要明确资源具体的使用期限，这样项目、任务、资源的关系越来越清楚了。



把上面的研究结合起来，这就得到了一个比较清楚的领域模型了，如下图所示。

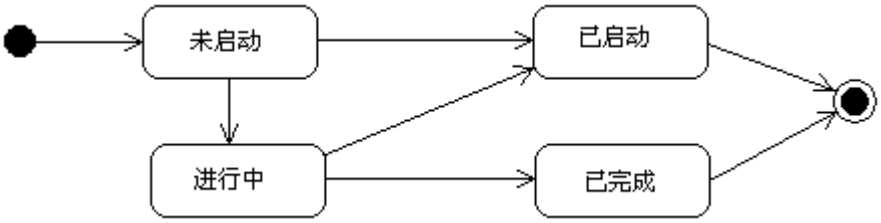


二、领域模型的行为和状态

静态领域建模成功以后，我们需要进一步讨论模型的行为和状态。

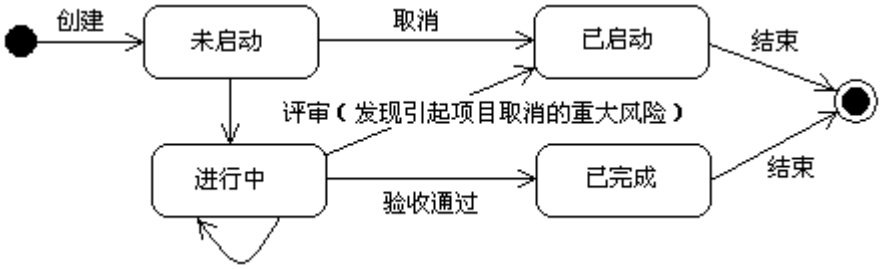
1）PMT 应该能够跟踪每个项目的状态

这些状态应该包括未启动、进行中、已完成、以取消等。



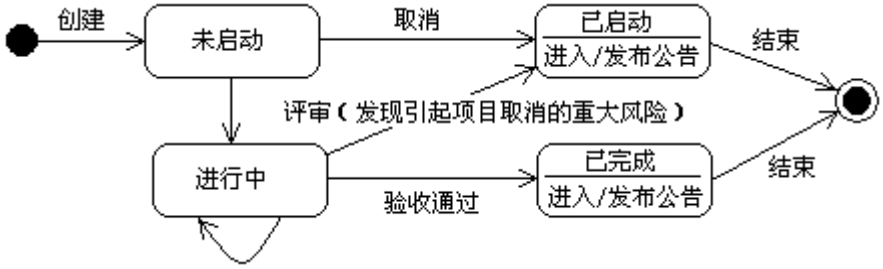
2）我们必须明确一个项目不同的状态是因何而发生变化的

可以修改得到下面的图。



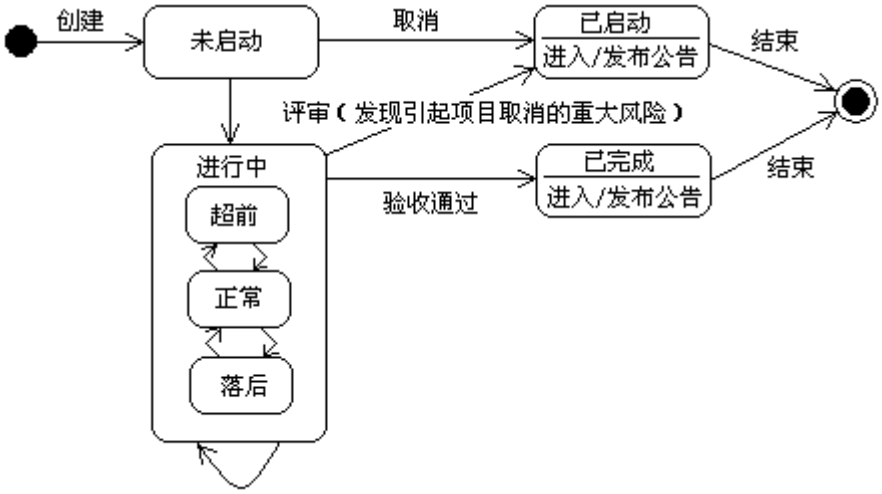
3）项目的每个状态，会对项目团队产生什么样的影响呢？

项目完成与项目取消这样的重大事件，应该通知所有的人知道，这就可能与公告板的功能有关。



4）这个系统必须能报告项目的进度情况

这些进度应该包括超前、正常、滞后。我们的领域模型已经有了时间点、工作量等领域，通过它们可以很容易实现项目进度状态的算法。



从上面的过程我们应该看的非常清楚，团队的合作与讨论是项目成功的关键。

4.3 概念性架构设计及模型

经过上面从多个视角分析的基础之上，我们就可以满怀信心地开始产品设计了。很多人在面对需求文档的时候，对于如何把需求文档转变为设计文档感觉很茫然，概念建模就是试图解决这个问题。概念建模是一次由内而外造就软件的过程。很多项目，正是因为模型设计不尽合理，任何需求的变更或者功能增加都可能引起一连串的问题。换句话说，这是因为忽视了概念建模这个“内”，而仅仅重视了编写程序这个“外”，多半是要出问题的。

概念建模是通过分析用例场景中的事件流，识别出实现用例规定的功能所需的主要对象和职责，形成以职责模型为主的初步设计。概念模型是说明问题域里（对建模者来说）有意义的**概念类,**作为设计的第一步，我们需要针对每个用例进行分析，就产生了概念模型，概念模型是作为设计软件对象的启发来源，也是后续设计的必须输入。

一、概念建模:

我们设计一个系统，总是希望它能解决一些问题，这些问题总是会映射到现实问题和概念。对这些问题进行归纳、分析的过程就是概念建模。概念建模专注于分析问题域本身，发掘重要的业务领域概念，并且建立业务领域概念之间的关系，建立概念模型的好处是：

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

概念模型是概念类或者是我们感兴趣的现实对象的可视化表示。它们也被称之为：分析模型、领域对象模型、分析对象模型等。概念模型是不定义操作（方法）的一组类图来说明，它主要表达：

- 概念对象或者概念类；
- 概念类之间的关联；
- 概念类的属性。

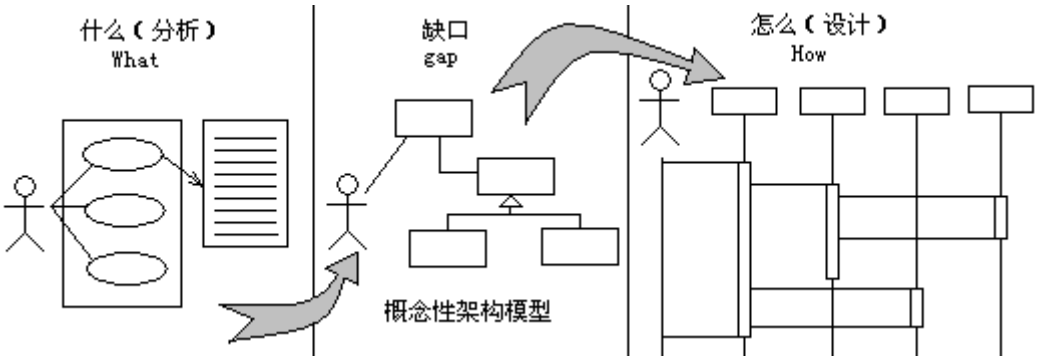
很多人在需求分析之后卡壳，不知道怎么做了。更可怕的是仅仅粗读一遍需求分析文档就开始设计了，结果设计结果与需求有很大的距离。

的确，用例场景描述了待开发软件的使用方法，但却没有以类、包、组件和子系统等元素的形式来描述系统的内部结构，用例场景向这些设计概念的过渡之所以困难，是由于下面这些原因：

- 用例是面向问题域的，设计是面向机器域的，这两个“空间”之间思维方式不同，但又存在映射。
- 用例技术本身不是面向对象的，而设计应该是面向对象的，这是两种不同的思考问题方式。
- 用例场景采用自然语言描述，而设计采用形式化的模型描述，描述的手段也不同。

概念建模就是希望能够填补分析与设计之间的缺口，所建立的概念性架构设计不考虑平台相关结构，只是专注于与产品功能相关的部分，从某种意义上也是产品的最小化设计部分。这种概念性架构为后期架构的优化和重构打下了一个很好的基础，它符合重构的基本原则，也就是小步前进，一个时间只戴一顶帽子。

事实上从用例场景到对象设计的过渡可以采用不同的技术，但概念建模是一种比较好的方法，关于这个缺口的填补如下图所示。

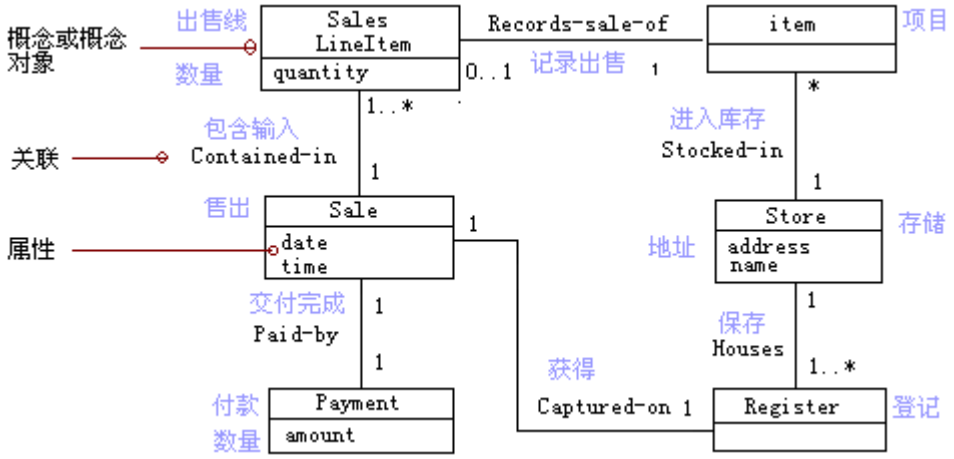


二、概念建模的基础案例

下面举个简单的例子，说明概念建模的基本概念。

1）问题的描述

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



2）关键概念

仔细考察上面的图，可以看出，概念模型实际上是可视化了分析中的单词或概念类，并且为这些单词建立了概念类。也就是说，概念模型是抽象了一个可视化字典。模型展现了部分视图或抽象，而忽略了建模者不感兴趣的细节。它充分利用了人类的特点—大脑善于可视化思维。

3）概念模型不是软件组件的模型

概念模型视相关现实世界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概念模型中表述：

1，软件工件（窗口或数据库）

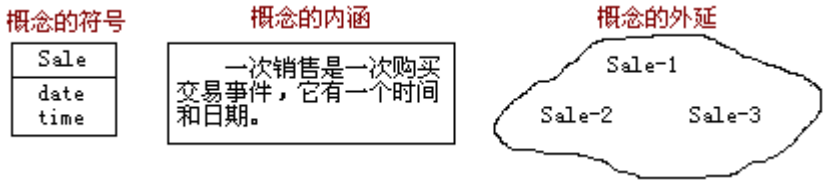
2，职责或者方法：方法是个纯粹的软件概念，在设计工作期间考虑对象职责是非常重要的，但概念模型不考虑这些问题，在这里考虑职责的正确方法是，给对象分配角色（比如收银员）。

4）概念类

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

- 符号：代表一个概念类的单词或者图片。
- 内涵：概念类的定义。

- 外延：概念类定义的一组实例。



三、概念类的识别

1，识别概念类

我们的目标是在相关分析中创建有意义的概念类，比如说创建“处理销售”用例中的相关概念类。一个方法是通过建立一个候选的概念类的列表，来开始建立模型。但是，更多的是使用名词短语分析找出概念类的方法，然后把它们作为候选的概念类或者属性。使用这种方法必须十分小心，从名词机械的映射肯定是不行的，而且自然语言中的单词本来就是模棱两可的。不过，这仍然是灵感的另一种来源。

一般来说，用大量细粒度的概念类来充分描述概念模型，比粗略描述要好。下面是识别概念类的一些指导原则：

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

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

2，概念类识别的指导原则

1）事物的命名和建模

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

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

以此，我们建议使用如下的原则：

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

2）在识别概念类的时候一个常犯的错误

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

另一个例子：考虑一下飞机订票的问题，Destination（目的地）应该是 Flight（航班）的属性呢还是一个单独的类 Airport（包括属性 name）。在现实世界中，目的地机场并不是数字和文本，它是一个占地面积很大的事物，所以应该是个概念类。

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

3，分析相似的概念类

有一些情况是比较不太容易处理的。举个例子，我们来分析一下“Register（记录）”和

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

我们应该知道，概念模型其实没有绝对正确和错误之分，只有可用性大小的区分。根据绘图员原则，POST 是一个分析中常见的术语，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，关键是看你的概念类重点是表达什么信息。这也是一个架构师必备的能力—抓住重点。

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，同样可以创建概念类，比如在电信业可以建立这样的概念类：消息（Message）、连接（Connection）、端口（Port）、对话（Dialog）、路由（Route）、协议（Protocol）等。

5，规格说明或者描述概念类

在概念模型中，对概念类作规格说明的需求是相当普遍的，因此它值得我们来强调。假定有下面的情形：

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

如果我们是这样来表达：

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

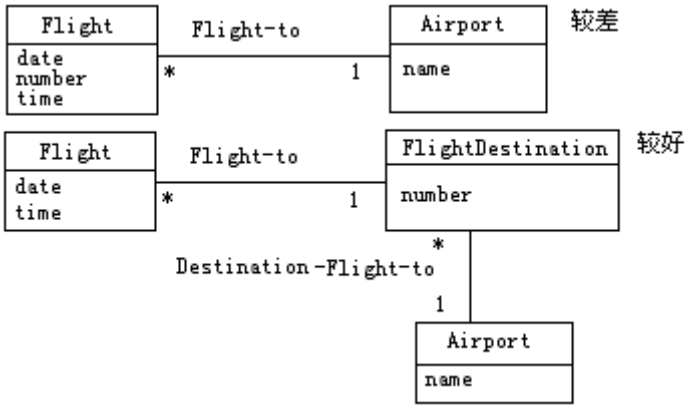
1）何时需要规格说明类

在下面的情况下，需要添加概念类的说明类：

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

2）服务的描述

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



四、概念模型的属性

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

1，有效的属性类型

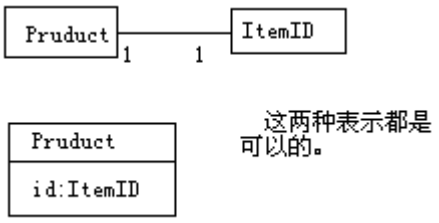
大部分属性应该是简单数据类型。当然也可以使其它的一些必要的类型，比如：Color（颜色）、Address（地址）、PhoneNumber（电话号码）等。

2，非原始的数据类型类

在概念类中，可以把原始数据类型改成非原始数据类型，请应用下面的指导原则：

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

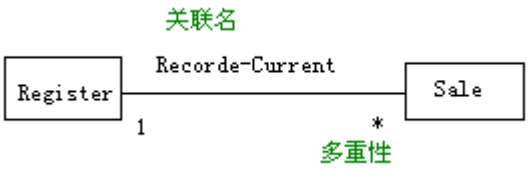
如果属性是一个数据类型，应该显示在属性框里面。



五、概念模型的关联

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”。



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概念模型，分而会使概念模型变的混乱，有的时候发现某些关联很费时间，但带来的好处并不大。
- 避免显示冗余或者导出的关联。

3，角色和多重性

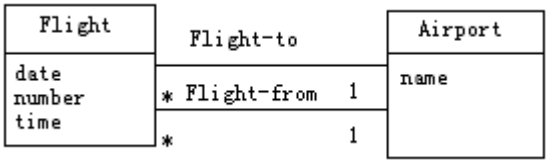
关联的每一端称之为“角色”。角色可选的具有：名称、多重性表达式、导航性。

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。

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

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。

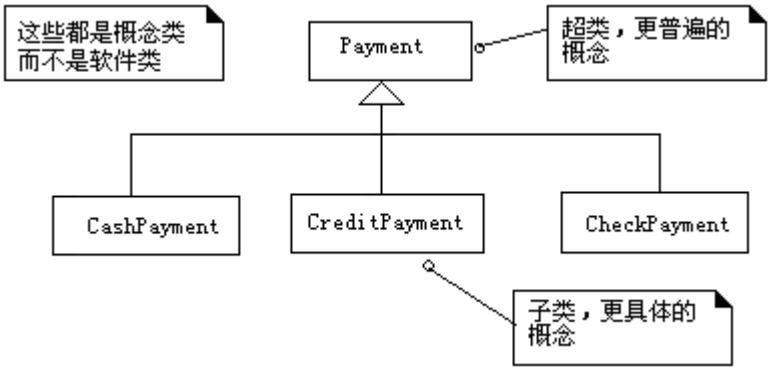


六、概念模型的泛化建模

泛化和特化是概念建模的基本概念，另外，概念类的层次，往往是软件类层次的基本源泉，软件类可以利用继承来减少代码的重复。

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。注意这里讨论的是概念类，而不是软件类。



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

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。

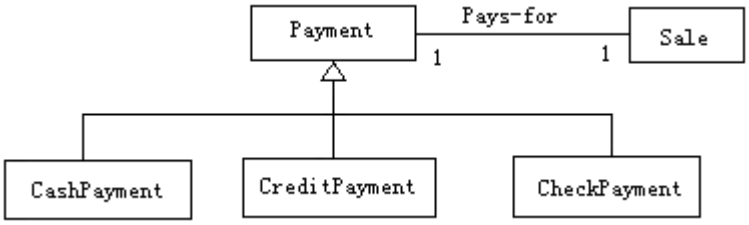
在前面的例子中，Payment（支付），是一个比具体的支付方法更为普遍的定义。

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）。

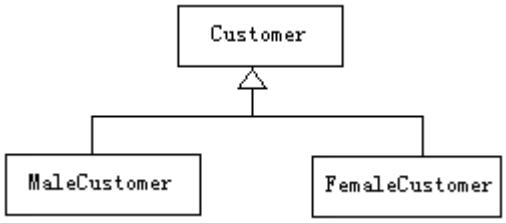
5）什么是正确的概念性子类呢

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。把一个概念类划分为不同子类的强烈动机为：当满足如下条件之一的时候，为超类创建一个概念性的子类：

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

由此看来，把顾客（Customer）划分为男顾客（MaleCustomer）和女顾客（FemaleCustomer）是不恰当的，因为它们没有额外的属性和关联，在运行（服务）方式上也没什么不同。尽管

男人和女人的购物习惯不同，但对当前的用例来说不相关。这就是说，规则必须和我们研究的问题相结合。

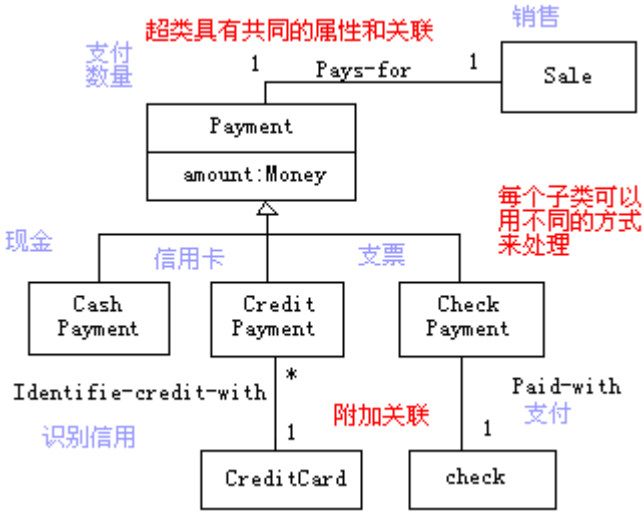
7）何时定义一个概念性超类

在多个潜在的子类之间，一旦发现共同特征，就可以暗示可以泛化得到一个超类。下面是泛化和定义超类的动机：

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

8）发现概念类的实例

Payment 类：



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。

4.4 行为模型与 GRASP 设计模式

在概念模型中是不考虑行为或者消息的，而主要考虑对象之间的静态关联，以及对象之间的泛化关系。当概念模型建立起来以后，就需要仔细研究对象之间的交互，这需要一个比较清晰的思路。利用著名的 GRASP 模式（General Responsibility Assignment Software Patterns 通用职责分配软件模式）可以作为思考的基础。尽管我们平时设计中已经在使用职责的概念，但是应用模式将会使我们的设计思路更加清晰，考虑问题也更加完整。

一、根据职责设计对象

什么是职责呢？从对象的角度来看，职责与一个对象的义务相关联，职责主要分为两种类型：

1）了解型（knowing）

职责包括：

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

2）行为型（doing）

职责包括：

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

职责是对象设计过程中，被分配给对象的类的。

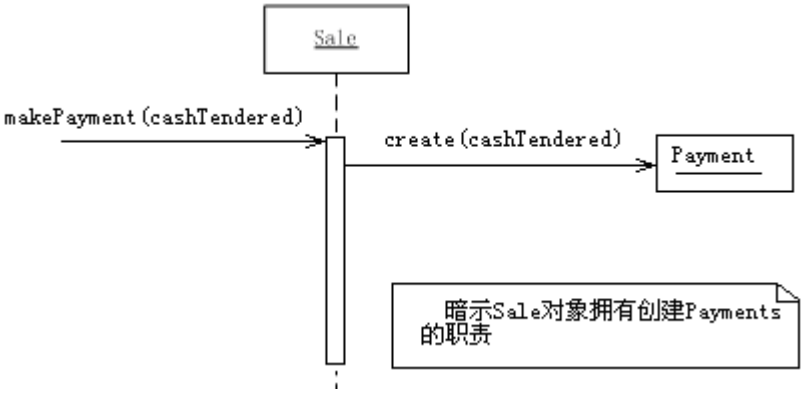
例：一个 Sale 类负责产生一个 SalesLineItems 对象（行为型职责）；

一个 Sale 类负责了解 Sale 类的 total 的值（了解型职责）。

我们常常能从概念模型推理出了解型相关的职责，这是因为概念模型实际上展示了对象的属性和相互关联。职责和方法不是相同的事物，但可以用执行方法来履行职责。职责是通过使用方法来实现的。

二、职责和交互图

虽然我们讨论的目的是为了给对象分配职责的时候提供一个基本的原则，但职责分配只有在编程的时候才能最终完成。在 UML 中，职责分配到何处（通过方法来实现）这样的问题，贯穿了交互图生成的整个过程。请看下面的图。



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

三、信息专家模式

解决方案：

将职责分配给拥有履行一个职责所必需信息的类，也就是信息专家。

问题：

在开始分配职责的时候，首先要清晰的陈述职责。假定某个类需要知道一次销售的总额。根据信息专家模式，我们应该寻找一个对象类，它具有计算总额所需要的信息。

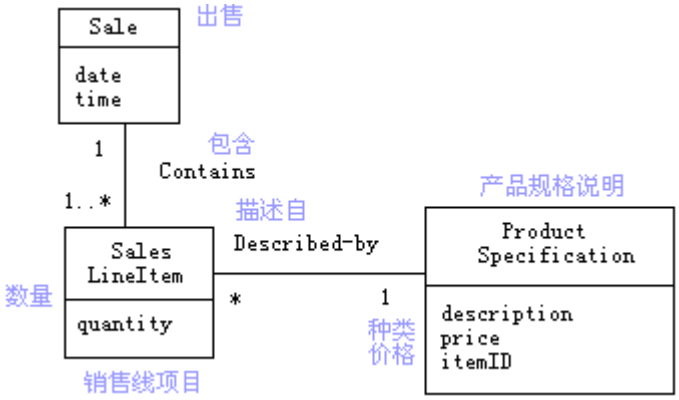
关键：

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

如果设计模型中存在相关的类，先在设计模型中查看。

如果设计模相中不存在相关的类，则查看概念模型，试着应用或者扩展概念模型，得出相应的概念类。

我们下面来讨论一个例子。假定有如下概念模型。



到底谁是信息专家呢？

如果我们需要确定销售总额。

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

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

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

上面已经提到，在创建交互图语境的时候，常常出现职责分配的问题。

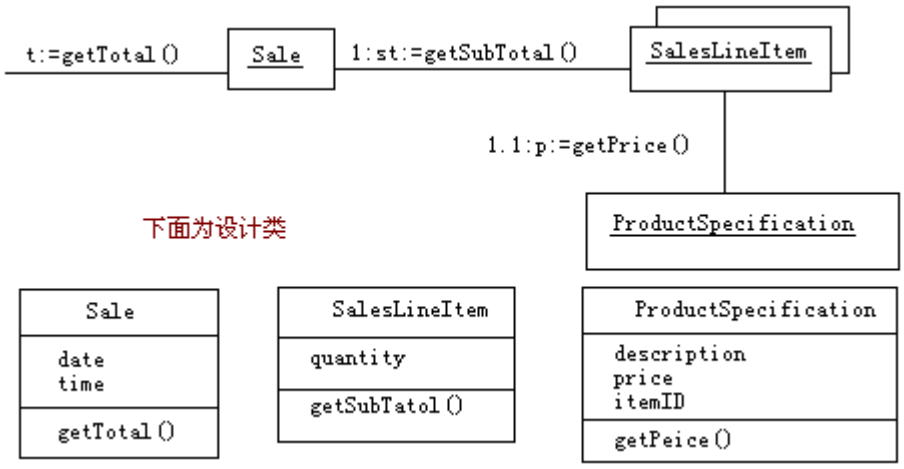
设想我们正在绘设计模型图，并且在为对象分配职责，从软件的角度，我们关注一下：

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

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

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

这样的思考，我们就在概念模型的基础上，得到了设计模型。



注意：

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

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

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

四、创建者模式

解决方案：

如果符合下面一个或者多个条件，则可以把创建类 A 的职责分配给类 B。

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

如果符合多个条件，类 B 聚合或者包含类 A 的条件优先。

问题：

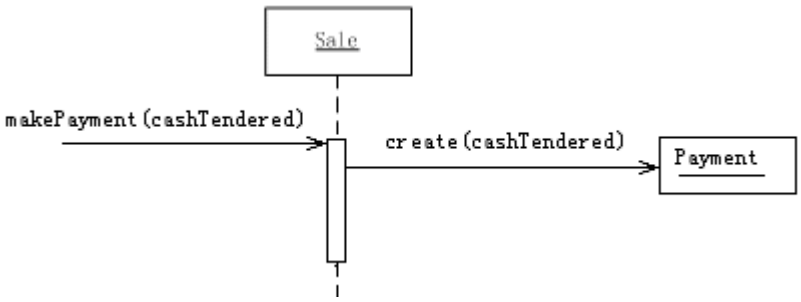
谁应该负责产生类的实例？

创建对象是面向对象系统最普遍的活动之一，因此，拥有一个分配创建对象职责的通用原则是非常有用的。如果职责分配合理，设计就能降低耦合度，提高设计的清晰度、封装性和重用性。

讨论：

创建者模式指导怎样分配和创建对象（一个非常重要的任务）相关的职责。

通过下面的交互图，我们立刻就能发现 Sale 具备 Payment 创建者的职责。



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

限制：

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

五、低耦合模式

解决方案：

分配一个职责，是的保持低耦合度。

问题：

怎样支持低的依赖性，减少变更带来的影响，提高重用性？

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

元素包括类、子系统、系统等。

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

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

示例：

我们来看一下 POS 机的例子，有如下三个类。

Payment（付款）、Register（登记）、Sale（销售）。

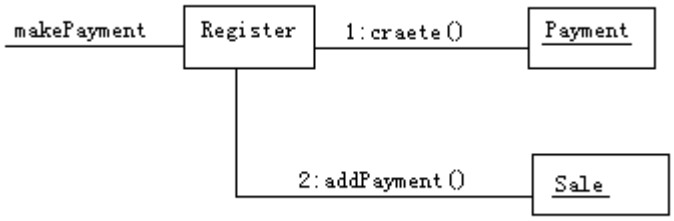
要求：创建一个 Payment 类的实例，并且与 Sale 相关联。哪个类更适合完成这项工作呢？

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

第一方案：

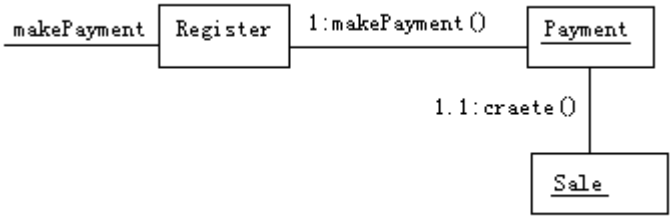
由 Register 构造一个 Payment 对象。

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



第二方案：

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



两种方案到底那种支持低的耦合度呢？

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

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

单纯从耦合度来考虑，第二种方案更优。

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

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：

- 一个存数据，一个取数据（低耦合）；
- 都存取数据（低--中之间）。

5，内容耦合（一般比较高）：

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

讨论：

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

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

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

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

低耦合度的需要，导致了一个著名的设计原则，那就是优先使用组合而不是继承。但这样又会导致许多臃肿、复杂而且设计低劣的类的产生。所以，一个优秀的设计师，关键是用一种深入理解和权衡利弊的态度来面对设计。设计师的灵魂不是记住了多少原则，而是能灵活合理的使用这些原则，这就需要在大量的设计实践中总结经验，特别是在失败中总结教训，来形成自己的设计理念。

六、高内聚模式

解决方案：

分配一个职责，使得保持高的内聚。

问题：

怎么样才能使得复杂性可以管理？

从对象设计的角度，内聚是一个元素的职责被关联和关注的强弱尺度。如果一个元素具有很多紧密相关的职责，而且只完成有限的功能，那这个元素就是高度内聚的。这些元素包括类、子系统等。一个具有低内聚的类会执行许多互不相关的事物，或者完成太多的功能，这样的类是不可取的，因为它们会导致以下问题：

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

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

示例：

我们还是来看一下刚刚讨论过的 POS 机的例子，有如下三个类。

Payment（付款）

Register（登记）

Sale（销售）

要求：创建一个 Payment 类的实例，并且与 Sale 相关联。

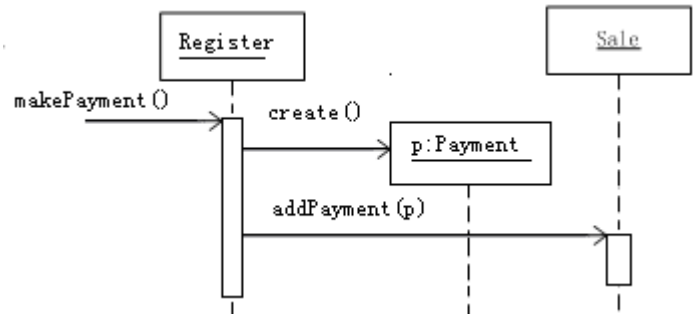
哪个类更适合完成这项工作呢？

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

第一方案：

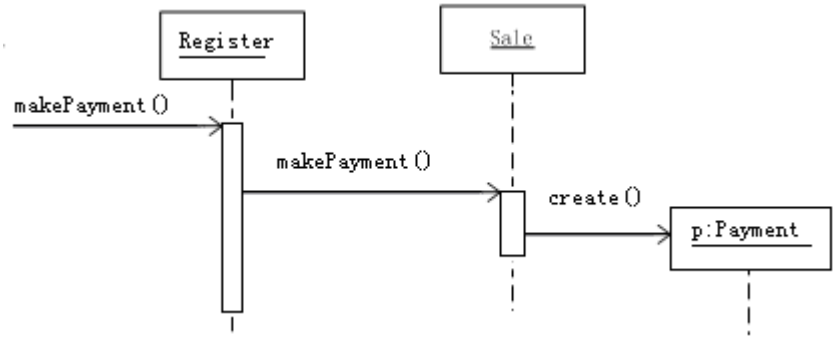
由 Register 构造一个 Payment 对象。

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



第二方案：

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



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

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

从开发技巧的角度，至少有一个开发者要去考虑内聚所产生的影响。一般来说，高的内聚往往导致低的耦合度。

讨论：

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

同样，它往往会和其它的设计原则相抵触，因此必须综合考虑。

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

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

一个类具有高内聚是非常有利的，因为它对于理解、维护和重用都相对比较容易。

内聚分类：

- 功能内聚：一个模块完成一个且仅完成一个功能（高）。
- 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）。
- 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）。
- 时间内聚：一组任务必须在同一段时间内执行（低）。

- 逻辑内聚：一组任务在逻辑上同属一类，例如均为输出（低）。
- 偶然内聚：一组任务关系松散（低）。

限制：

少数情况下，接受低内聚是合理的。比如，把 SQL 专家编写的语句综合在一个类里面，这就可以使程序设计专家不必要特别关注 SQL 语句该怎么写。又比如，远程对象处理，利用很多细粒度的接口与客户联系，造成网络流量大幅度增加而降低性能，就不如把能力封装起来，做一个粗粒度的接口给客户，大部分工作在远程对象内部完成，减少远程调用的次数。

关于耦合性和内聚性的设计原则：

- 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。
- 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。

七、控制需求与状态转换关系

在上面的讨论中，我们仅仅考虑了每个事件发生后的业务序列，以及计算机能够告诉系统发生了什么。在大多数情况下这已经足够定义动态信息问题了。但是对于复杂的控制问题，仅仅使用用例来描述就显得不足，本节会提出若干方法来描述一些附加规则。

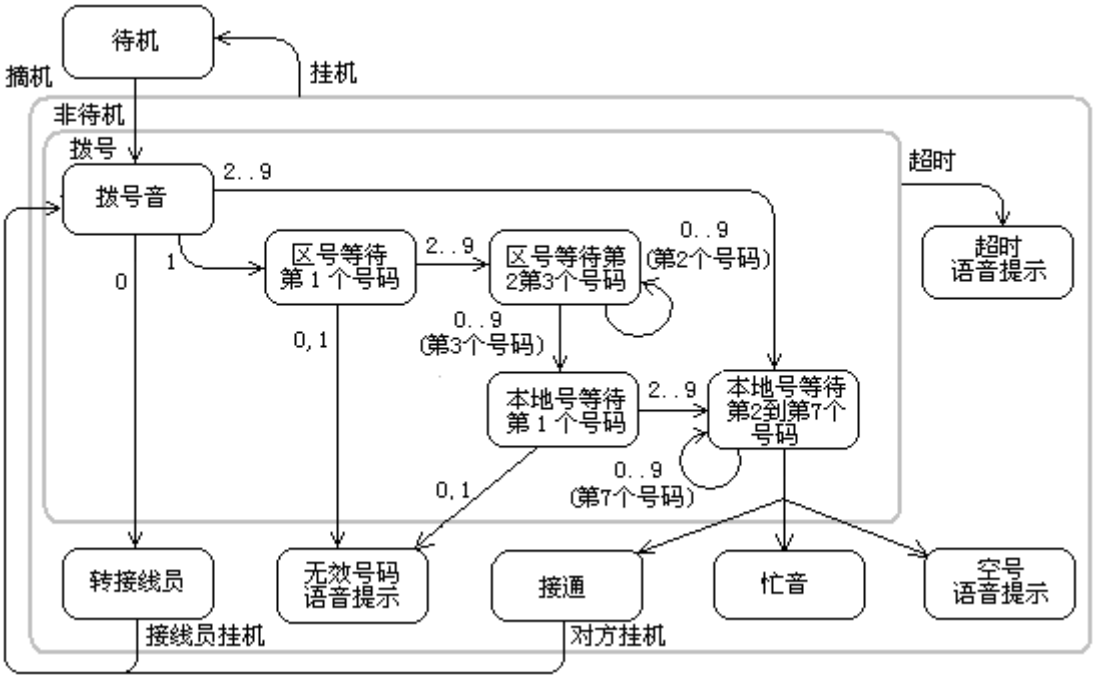
1，状态转换的描述

软件问题绝大多数都是讨论对象在不同的时间具有不同的状态，为了控制它们的状态，就需要知道什么会引起它们改变状态。对于某些类型的对象，使它们改变状态的规则可以很好地用数学方程式表达，例如飞机控制系统可以通过一组差分方程，根据空气密度、风的速度以及升降舵、副翼和方向舵的位置，依据一组复杂的数学方程式，计算出飞机如何运动或者是它当前的状态。得出控制这类对象的规格说明是控制理论的一部分，也是高度专业化的。

另一方面，更多的对象是执行一些离散的事件，当动作结束的时候，可能从一个状态转换为另一个状态，这里的状态非常少，少到可以用文本书写它，例如前面用例文档的后置条件，就包含了这层意思。

但是很多情况下状态转换处于这两者之间，它们具备相当的复杂性和相关性，很多设计者做出的状态转换图并没有很好的定义开发，引起这种图的价值也受到质疑，从高层设计的角度就需要把这些状态转换进行简化和条理化，最终可以定义软件产品开发。

下面通过一个电话线路的行为作为例子，讨论如何把一个复杂的状态转换图转变成一个条理化的设计文档。所讨论的复杂控制的状态转换图如下。这里描述的事件包括拨号、挂断、举起听筒、长时间不做任何事情（超时）、另一方挂断电话。



为了控制复杂性，需要对状态进行分类、合并与条理化，图中已经显示了几种方法：

1）超状态：

如果许多状态在相同的事件触发现具有相同的结果，可以组合这些状态成为一个超状态，并把它们封装在一个方框中。比如，**挂机**事件可以在 11 个不同状态下发生，但每一种情况下的结果都是相同的，所以可以把它们合并成一个“非待机”超状态。**超时**事件可应用在拨号超状态下的所有状态上。用灰色的线框表示超状态，不但可以减少由紧密排列的平行线所造成的视觉混淆，更可以达到归纳整理的目的。

2）对状态分类：

尽管区号中每个数字与本地号码的组合会有不同的状态，还是可以把它们（区号、本地号）各分解成两个状态，尽管还有更多的状态没有表达出来，但作为表达整体状态关系的图，这就足够了。

3）利用文本描述：

在图中可以省略任何您想省略的东西，因为还有利用文本解释它们的机会，图形越复杂越棘手，就需要要更多地考虑用文本来描述状态转换，详细的图形描述往往使图更混乱。记住，我们的任务是用最简单的方式描述对象和它的状态，让开发者容易去理解，而不是强迫每个描述都成为标准的图形符号，文本是永远值得信赖的东西。为了用文本描述对象和它的状态，就需要包含一下信息：

- 所有状态的列表。
- 对于每个状态，对象在这个状态中作了什么？或者有关这个状态外部能检测到的差异。例如：对于“车库开启者”这个对象，一个状态是“打开”，对象做的事情是“用马达把门拉开”。又比如灯泡的状态是“开”，外部能检测到的差异是“灯泡闪亮”。
- 对于每个状态，哪个事件是可能的，以及对于一个可能的事件，在那个状态下对象如何响应，对象执行的行为是什么？在那个事件后对象的行为是什么？
- 在状态转换图中没有表现出来的附加状态信息，如缩位拨号。
- 哪个状态是开始状态？
- 哪个状态（或多个）是结束状态？

下面就是电话拨号状态图的文本描述。

除了图中所显示的状态，还有五个状态变量应用到电话线路中	
区号	最多 3 个数字的字符串，如果呼叫的是长途，区号要被拨打。
本地号	最多 7 位数字的字符串，在区号内部的电话号码。
定时器	一个 60 秒的定时器，其状态有 2 个，运行（倒计时）；关闭。
传送	传送电话音频信号，状态有两个：传送；关闭。
接收	接收传送过来的音频信号，状态有两个：接收；关闭。
事件列表	
摘机	回路关闭，只能发生在待机状态。
挂机	回路打开，只能发生在待机状态以外的其它状态。
超时	定时器倒计数为 0。
0，1，2，3，4，5，6，7，8，9	按键拨打脉冲数字。
对方挂机	只能发生在通话中，对方挂机。

状态和响应			
状态	**事件**	**行为**	**下一状态**
待机 （传送、接收、定时器处于关闭状态）	0..9	—	待机
	摘机	—	拨号者
	超时	（假定不发生）关闭定时器	待机
拨号音 （传送处于关闭状态，接收拨号音，状态进入时，设置定时器为 60 秒）	0	—	转接线员
	1	—	区内号，等待第 1 个号码
	2..9	区内号位数字	区内号，等待第 2 到第 7 个号码
	挂机	—	待机
	超时	—	超时语音提示
区号等待第 1 个号码 （传送、接收处于关闭状态，定时器运行）	0,1	—	空号语音提示
	2..9	区号为数字	区号等待第 2 第 3 个号码
	挂机	—	待机
	超时	—	超时语音提示
区号等待第 2、第 3 个号码 （传送、接收处于关闭状态，定时器运行）	0..9	添加数字到区号	如果区号已经有 3 位号码，区内号等待第 1 个号码，否则区号等待第 2 第 3 个号码
	挂机	—	待机
	超时	—	超时语音提示
（其它状态为了简洁而省略了）			
接通到另一方 （传送到本地号码所指定的线路，如果未拨区号就是本地号码，否则就是区号，从相应线路接收，定时器关闭）	0..9	—	接通到另一方
	挂机	—	待机
	超时	（假定没有发生）定时器关闭	接通到另一方

请注意，这个表说明了每一状态超时将会发生什么情况，如果仅仅看状态图，就很容易忽略这个情况。有些状态表现不明显，但通过一张表就可以系统的关注每种可能的情况。

状态图可以让人更好的理解总体情况，但文本不容易让人理解总体情况，但可以很好的理解每个状态的细节，每次一个事件的仔细阅读，线性的引导阅读者从一个事情到另一个事情，特别是图中难以表达的细节，在文本中可以很详细的表达。

对于某些相同的响应（比如超时），也可以用一个专门的表来描述。

虽然这张表已经表达了状态图希望表达的所有东西，而且更详细而确切，但是图的作用还是很大的，因为在文本中，状态之间的关系难以表达，典型的情况是：

读者首先从文本读取一个状态的信息，然后参照图去研究什么状态可以转换到这个状态，然后在文本中再读取一点，再回到图中查找。如果没有图，要么读者需要在自己的大脑中形成自己的视觉化场景，要么纯粹抽象的理解状态转换，这两种情况都加重了读者的负担，尤其是很少有人具备纯粹抽象的处理问题的本领。最终是开发出的产品结果与当初的要求有很大的出入。

所有这些还有一个重要的暗示，很多人喜欢使用各种各样的 CASE 工具来辅助分析与设计，这当然无可非议，但是工具可以减轻重复劳动，并不能帮助我们思考，尤其不能帮助我们站在读者的角度处理问题。你必须站在读者的角度考虑元素安排，图的重点要突出，表达的思想要清楚，图的布局应该和谐，读者的眼睛应该很容易跟踪图的流向，并且和文本的顺序有一定的内在联系。发起事件的关键状态（比如拨号）应该放在左上角，而不要被其它各种状态包围。对于一些特殊的异常情况（比如超时提示），也不要和其它状态排列在一起。

在画状态转换图的时候，一个需要时刻铭记于心的原则是，仅仅忠实地事无巨细的画出转换关系是不够的，如果你画了一个混乱的图，相当于你什么都没画，因为文本已经提供了完整的描述。如果有了一个混乱的图而没有文本，那也就更不需要操心书写文本了，因为这相当于你什么都没做。从这个角度来说，设计做到一定的层次更像一种艺术。

2，命名状态和事件

状态与事件在命名上要有明显的区分，基本的命名原则如下：

事件的名称应该是动词或者名词（或者作为动词或名词使用的短语），需要能够清晰地体现事件在特定时间发生和结果，比如挂机（hang up）。

状态的名称应该是形容词或名词（或者作为形容词或名词使用的短语），能够清晰地体现能持续一段时间的状态。比如灯泡的状态：开（on）或者关（off）。

用形容词命名状态的一个重要的类型是分词：使用分词可以使一个动词转换成一个形容词，现在分词一般是动词加 ing，过去分词与过去式是一样的，比如：connected。当然也有一些不规则动词的过去式是不同的，比如：broken。下面的表示一些命名状态和事件的很有用的词汇。

状态	事件
开始（start）	开始（start）
在标题段落中（in header segment）	创建（create）
目标获取（target acquired）	获取目标（acquire target）
已获得密码（got password）	得到密码或密码（get password or just password）
已察觉入侵（detected intrusion ）	察觉入侵或入侵（detect intrusion or just intrusion）
已接收到确认或已确认（received confirmation or confirmed）	接受确认或确认（receive confirmation or just confirmation）
等待确认（awaiting confirmation）	情况变更（status changes）
已做（done）	中止（abort）

其中，开始（start）同时出现在两栏中，因为它经常对事件和状态都有用，但不能描述同一个对象，“开始（start）”是一个好名称。作为状态：它描述没有经历任何事件的状态。作为事件：作为状态图描述一个发起一个过程的事件。

3，状态转换的四种解释

状态转换有一些基本的不确定性，可以有意用下面的四种方式中的一种来解释。

从任何状态下出现的事件，是当对象处于那个状态时的：

- **唯一可能事件**：没有显示的事件不可能发生。
- **对象如何响应的事件**：没有显示的事件没有响应，或者不可能在此发生。
- **唯一允许的事件**：没有显示的事件，系统必须阻止它发生。
- **对象对事件的期望响应**：没有显示的事件，要么不可能在此发生，要么期望的相应忽略了它。

前两种解释是对于问题域的描述：第一种是对于所有可能事件集合的描述；第二种是对因果关系的描述。后两种解释作为描述性陈述：针对的是设计决策，或者需求规格说明。

如果你正在写自动电话拨号机的设计文档，那么前面的自动电话拨号机状态图纯粹就是描述性的，开发者会根据这个状态转换关系设计出满足需要的设计来。

如果你正在写控制电话公司连接呼叫的设备的设计文档，那自动电话拨号机状态图就是说明性的，文档需要据此加上其它的描述性陈述，说明对于不同的电话线、连接传送、接收通道都是些什么事件，创建关于信道状态变更的需求说明。

为了避免模糊性，在描述的时候可以加上某些情态动词，比如：“必须”或者“应当”。必要的时候，也可以使用命令方式描述事件和响应，比如：

“按下数字 2..9：添加数字到区内号码。”

要注意描述的模糊性，往往是产品开发最后失败的根源。

八、产品行为问题的归纳总结

任何软件都需要关注引发期望结果的行为。期望结果往往是某种行为的功能，所以在软件需求模型中，可能有三种类型的问题需要书写：

- **自发行为**：这些行为在问题域中发生，比如：按下复印机按钮。
- **直接行为**：软件能直接初始化的行为，比如复印机加电。直接行为是共享现象，它同时是软件的行为，也是问题域的行为。
- **间接行为**：由其它行为引发的行为。

相同的行为并不总是引发相同的结果，比如打印机滚筒可能成功地输送一张纸，也可能失败。“行为”和“事件”还是有区别的，时间表达的是一个很短暂的行为，通常反映了某种触发。而行为是更宽泛的，具有明确的开始和结束，每个行为需要书写的信息如下：

- **因果关系类型**：不是使用诸如自发、直接、间接这些稍微深奥的词汇，而是简单的把每种类型的行为组合到一起，比如：微处理器可以产生下列行为。
- **行为中涉及的所有类型对象**：实施那个行为的对象是什么？被行为影响的对象是什么？
- **行为所具有的参数**：行为的属性从一个实例到另一个实例有所不同，它需要的参数是什么？会有哪些状态发生改变？
- **在间接行为的情况下，引发此行为的条件或事件**：什么时间发生？什么对象引发的？
- **如果只有一个特定的条件为真，此行为就一直发生**：可以描述：“只要……就发生。”
- **行为的持续时间**：除非时间足够短可以忽略，就需要表述行为持续时间。
- **行为的所有可能结果**：对于每一种被影响的对象，可以有什么效果？
- **如果可能存在一个以上的结果**：软件是如何或能否监测到哪个行为真正发生？

从本质上看，前面所讨论过的用例文档就是描述一种行为（也是一种功能），其中前置条件与后置条件具备某种定义参数和状态的作用。如果某些情况下需要专注于行为的描述，也可以单独写出行为文档，比如：

收集货物	
参与者	货物收集者，货物，存储位置
参数	一个或多个货物清单以及它们的存储位置
什么时候发生	已打印的订单（显示获取清单和存储位置）是在打印机 A 处，以及货物收集者分遣订单
持续时间	从货物收集者分遣订单开始少于 5 分钟，在多数情况下如果超过 10 分钟就有问题。
可能的结果	1）货物收集者找到货物并带他们到包装站。 2）货物收集者寻找货物，但没有找到。 3）货物收集者没有找到订单/或没有搜索货物。 10 分钟后，可以认为 2 或者 3 发生了。

描述行为应该简明扼要抓住重点，在行为比较复杂情况下也可以使用顺序图或者状态图来表达，并辅以相应的文档。我们没有必要为设计师呈现一个犹如智力拼图一般的行为描述。

事实上在功能需求与非功能需求的描述中，已经定义了大部分行为问题。

4.5 设计模型和实现模型

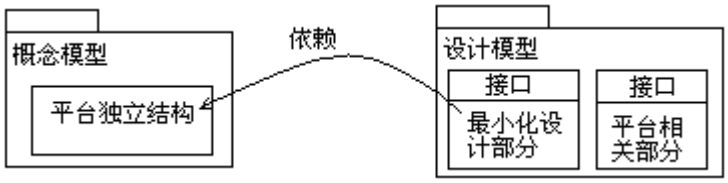
一、从概念模型到设计模型

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

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

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

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



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

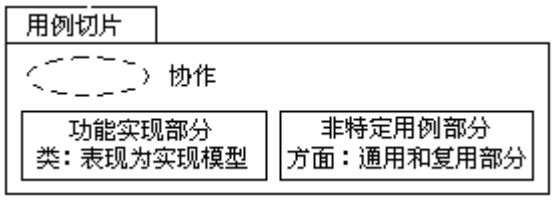
上面提到的边界类、控制类和实体类，或者它们的组合，将演变为设计中的构件，这些构件将拥有来自于概念类职责的接口。我们称这些接口为**最小化设计接口**。

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

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：

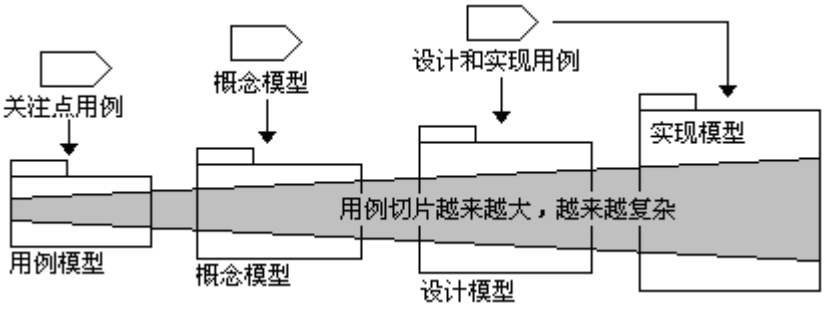
- **协作**：我们知道，用例是从系统的外部来描述系统的，但是协作是从内部视角来描述系统，在描述上可以表达类与类之间具体的协作关系。同时还必须对它命名，一般来说，协作的名字需要和用例一致，以防止理解上的歧义。
- **类**：这些类是特定于该用例实现，是本质上与其它用例分离的部分。
- **方面**：是该用例实现的时候对原有类的扩展，也是可能发生缠绕的部分。



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概念模型，这就是从高层视角对系统的描述。
- 通过设计模型来决定系统会运行于什么平台。
- 实现模型就是系统的代码实现。

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



例如当工作于一个用例切片的时候（例如设计阶段用例切片），应该从这个用例切片的上游（对应于分析阶段用例切片）开始做一些精化，添加一些内容。因此每个下游用例片都比上游切片更大也更复杂。模型也同样的是下游模型比上游模型更大而且更复杂，这是因为模型也需要考虑越来越多的问题。

1，保持用例模型中的结构

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

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

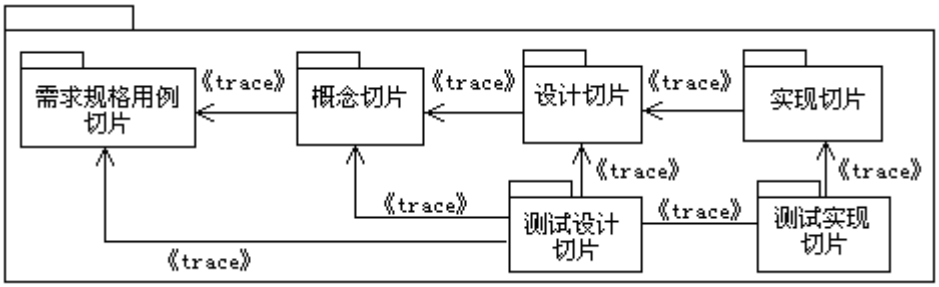
同的结构，并且维护这种模型之间的映射关系，由于并没有一种形式化的方法解决这个问题，因此难以在模型中保持一致。

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

2，用例模块包括用例结构

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

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

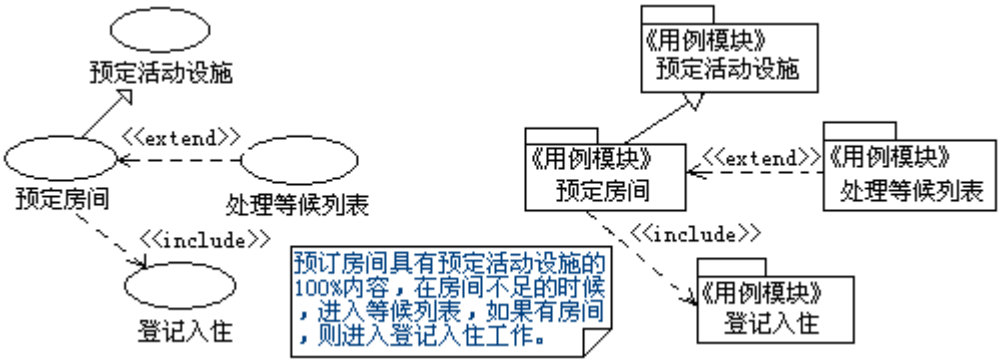


在软件开发中，**可追朔性**是一个很重要的概念，可追朔性表明上游模型元素与下游模型元素存在着链接，这可以帮助你判断是不是所有的需求都已经实现，需求的变化会对什么产生影响，通常上游元素的修改会影响到下游元素。

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

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。

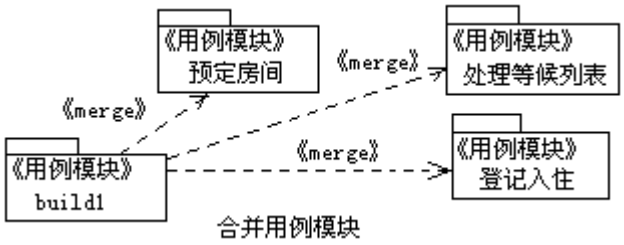


用例模块的关系可以用于两个互补的方面，从正向工程的视角，可以从用例之间的关系派生出用例模块之间的关系，换句话说，用例模块保留了用例之间的关系，具备了关系的一致性。

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。

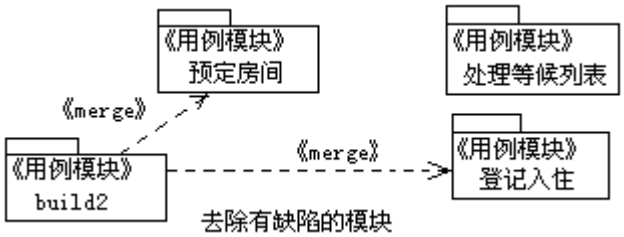
4，合并和配置用例模块

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。



合并用例模块

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。



去除有缺陷的模块

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。

4.6 关注点的分散、缠绕与合并

在软件开发中，使关注点保持分离是十分重要的，它可以帮助你把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。

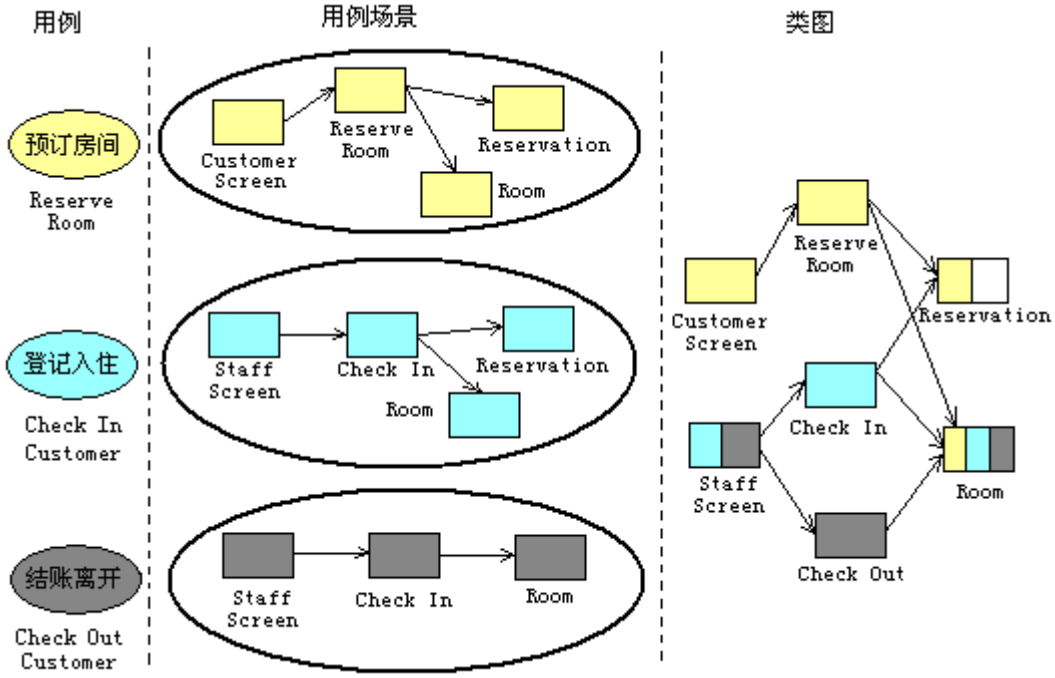
一、使关注点相互分离

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。但是，我们还是发现，尽管在分析的时候我们强

调了关注点分离，一旦进入设计，这种分离仍然会不可不可避免地造成交叠，我们下面来研究这种关注点分离中的问题：

在分析的时候，我们力图使关注点成为**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。

二、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观注点保持分离，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：

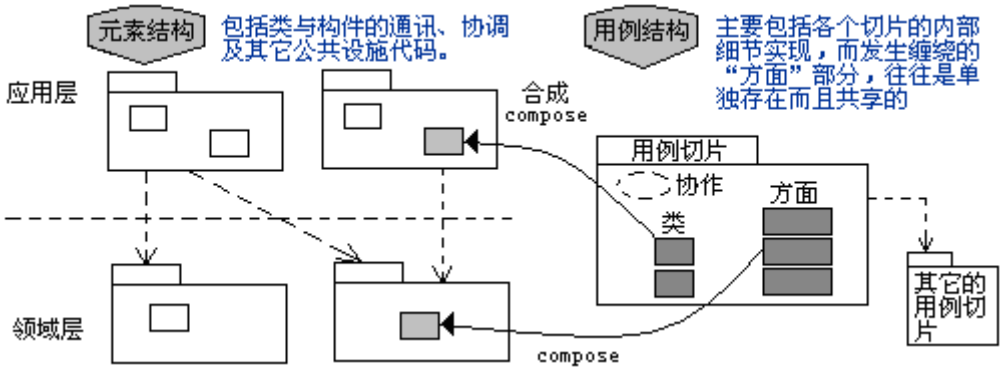
1）元素结构

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。

三、合并类的扩展

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，通过合适的方法组合用例切片，构成设计模型中完整的元素集，合成的方法，可以使用面向方面的编程语言，也可以使用其它方法实现这些思想。

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）。

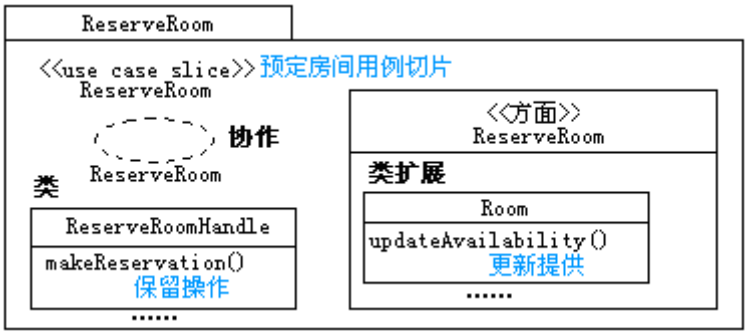
1，使用例特定部分保持分离

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。

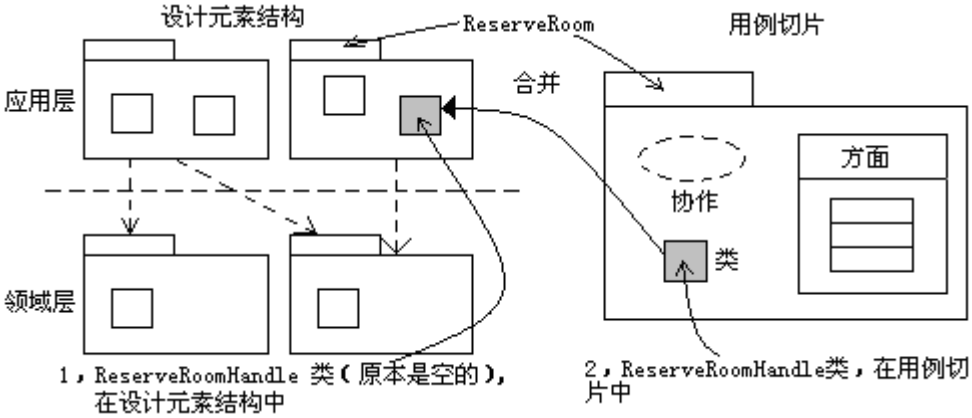
“预定房间”用例切片包含 ReserveRoomHandle 类和 Room 类扩展，在类中标示出相应的行为，不论是类还是方面，一般都有多个。我们可以建立某种机制把 Room 类的片断合并到原有的 Room 类中去，例如，我们可以通过 AOP 的方面机制，通过 intertype 声明和 advice 来实现。

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。

2，合并特定于某用例的类与扩展

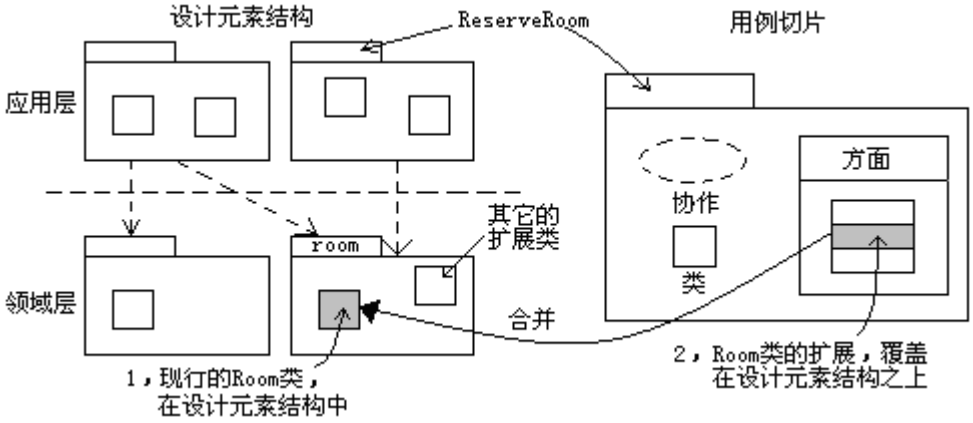
以 ReserveRoomHandler 类为例，看一看特定于某个用例实现的类是如何合并到设计模型中的。假定我们使用层（layer）和包来组织设计模型，则可能有一个应用层，它包含一个客户应用包，ReserveRoomHandler 类应该添加在这个包内。



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。

我们可以考虑合并类的扩展，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。



利用面向方面的技术，我们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，利用框架技术，通过后面我们要讨论的设计模式，也可以实现这种分离与合并的结合，重要的是发现缠绕的规律，并根据这些规律寻找解决方案，这些都是值得我们去研究的。

4.7 从产品模型到测试模型

模型驱动的开发不仅仅影响到分析与设计，更影响到测试。模型驱动的测试方案，可以以更小的成本实现更有效的测试，另一方面，测试驱动的开发又可以确保开发不至于偏离正确的方向，从而大大提高产品质量。在敏捷开发中，由于测试是在每个迭代过程中快速完成的，这种测试策略将显得更有效。

在传统过程模型上，测试人员在很晚的时间才进入开发过程，他们得到一个规格说明的最小集合，对被测系统他们得到的是一个未知的“黑盒”，在论证任务的时候，他们往往会问：

- 到底系统要做什么？以什么顺序做？
- 怎样创建和记录一组测试场景来检查系统的功能？
- 我怎么知道什么时候已经完整的测试了系统？
- 还有没有什么其它的事情应该做或者不应该做？

到最后他们还会问一个问题：“有什么办法使测试工作早一点开始，以便早一点发现问题？”在敏捷开发模型下，这个矛盾将更加突出，因为迭代周期很短，甚至很多工作都不是专职的测试人员来做，任何把测试工作向前的移动都是有意义的。

如果测试团队采用测试用例来做这件事，除了黑盒以外，还可能发现如下资产：

- 一个完整地用例集合，记录着有序的事件顺序，这些事件记录着系统如何与用户交互，并向用户提交结果。
- 一个用例模型，记录着系统所有用例，以及它们如何交互，什么参与者驱动这些用例。
- 在一个用例场景中，基本事件流和备选事件流定义了系统“如果……那么”的行为。
- 前置条件和后置条件的描述。
- 定义系统非功能性需求的补充规格说明。

这就可以看出来，用例技术构建了一组可以用来驱动测试过程的资产，无论是开发效率和产品质量都可以由此得到很大的提高。

一、测试用例的概念

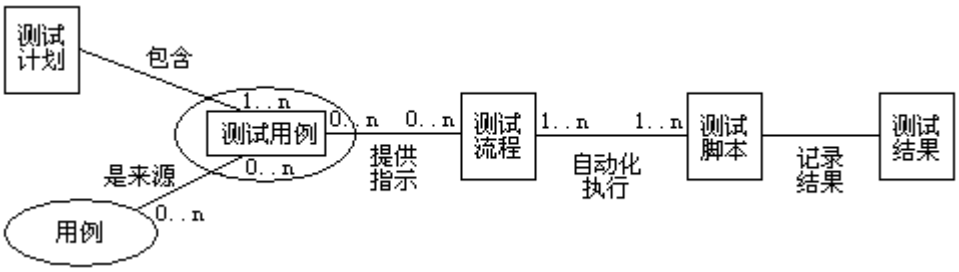
用例本身并不是测试用例，但是用例驱动了测试过程，为了转化这些资产为系统测试打下了基础，仍然需要一些严肃地分析工作，首先让我们定义一些名词。

1）公共测试名词

- **测试计划**：包括项目测试的目的和目标信息，此外测试计划确定实现和执行测试的策略和所需要的资源。
- **测试用例**：一组为了特定目标（如执行特定程序路径或验证符合特定需求）开发的测试输入、执行条件和预期结果。
- **测试流程**：一组为给定测试用例的安装、执行和结果估计的详细指令。
- **测试脚本**：一个自动化执行测试程序（或测试程序的一部分）的软件脚本。
- **测试覆盖**：定义了一次测试或者一组测试，解决给定系统或组件所制定测试用例的程度。
- **测试项**：一个测试对象的构建（Build）。
- **测试结果**：在测试执行中获取的数据集，用于计算测试的不同关键度量。

2）测试工件的关系

可以看出来，为了实现和管理一个全面的测试过程，需要相当多的测试工件，下图展示了这些工件之间的关系。



测试过程的基础是测试计划，其中包含了测试策略，并且引用或者包含测试用例本身。用例是潜在测试用例的来源。对每个测试用例来说，都有一个或者多个测试流程定义如何执行特定的测试用例。测试用例的执行可以是手工的，也可以运行一套测试脚本。测试的结果记录在测试的结果集中。

3）测试用例的作用

测试活动的中枢是测试用例。测试用例的创建和执行本身是测试活动组成的一大部分，它们设计和执行的质量，以及测试的彻底性，决定了最终结果的质量。

- 测试用例决定了设计和开发测试流程的基础。
- 测试的“深度”与测试用例的数量成正比。
- 测试工作量的规模与测试用例的数量成正比。
- 测试设计和开发以及所需的资源，很大程度上由所要求的测试用例控制。

在项目的先启、精化和构建阶段中创建的用例，可以作为这个过程的输入，由于测试的大部分工作量在预定义和实现测试用例，这就实现了把测试工作向前移动的目的。

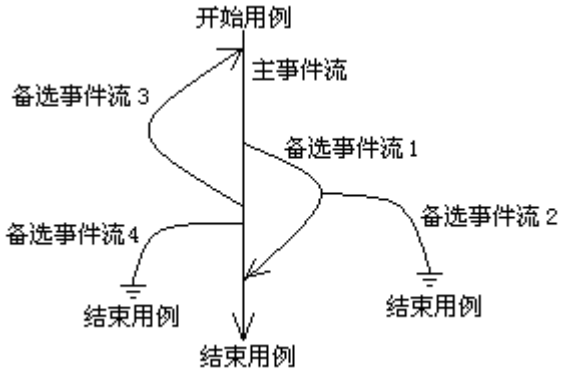
二、从用例得到测试用例

用例和测试用例有不同的起源，并服务于尽管相关但却不同的目的，所以从用例到测试用例并不简单，但还是有合理的步骤，首先我们定义一下场景的概念：

场景：或用例的一个实例，是一个用例的执行，其中特定用例以特定方式执行该用例。

场景可能有多个，如下图所示，用户可能走主事件流，也可能走备选事件流 1 和 2，然

后异常退出。每个路径都可以是被执行和测试的场景或实例。



既然我们已经定义了用例场景的概念，就可以提出一个四步的过程来完成这个目标。

1）第一步：确定用例场景

因为用例和场景之间是一对多关系，我们可以把基本流域备选流之间的关系用一个矩阵表达出来，假定已经有上面的用例，可以写出场景矩阵。

场景矩阵				
场景编号	**开始流**	**备选流**	**下一个备选流**	**下一个备选流**
1	基本流			
2	基本流	备选流 1		
3	基本流	备选流 1	备选流 2	
4	基本流	备选流 3		
5	基本流	备选流 3	备选流 1	
6	基本流	备选流 3	备选流 1	备选流 2
7	基本流	备选流 4		
8	基本流	备选流 3	备选流 4	

注意到我们描述的用例还不是太复杂，就产生了相当数量的场景。在很多情况下，测试人员需要设计一个既认识到测试所有的场景不现实，同时又有足够测试的测试策略。在烤炉策略的时候，首先列出所有的场景是必要的。

另外，测试人员也要认识到，并不是所有的场景在原来的用例中都有描述，场景发现的过程要与开发团队交互地进行，这样做有两个原因：

- 用例开发是用于实现的，没有百分之百穷尽，其详细程度对测试来说可能不够。
- 测试团队的审查过程将通过执行用例创建新的发现场景，有的甚至在设计的时候都没有考虑到，所以就会发生修改设计。

这也是我们在生命周期方法中选择迭代模型的原因之一，因为它允许我们有效的计划和管理这个过程。测试团队审查用例并发现漏洞，或者附加备选流程将可能产生更好的系统。

2）第二步：确定测试用例

公司的测试过程千差万别，但测试用例都应该包括要实施的测试参数，包含测试的条件和预期的结果。下面的表就是一个公共的格式，使用一个矩阵，表达场景、条件、数据、预期和实际值。

测试特定场景的矩阵						
测试用例编号	**场景/条件**	**数值 1**	**数值 2**	**数值 N**	**预期结果**	**实际结果**
1	场景 1					
2	场景 2					
3	场景 2					

注意，上面的表中一个场景可能产生多个测试用例（见用例 2，3），这是因为一个场景可能会有多种逻辑成分。假定有一个关于自定义照明策略的用例：

户主为一周的每天输入最多 7 种照明序列，系统用一个蜂鸣声确认每个输入。

这个简单步骤将产生两个测试用例，如下表所示。

测试用例编号	场景/条件	描述	预期结果
1	场景 6	少于 7 个序列的输入	保存序列 系统蜂鸣提示
2	场景 6	尝试输入 8 个序列	出错

此外，这个过程中我们还发现了一个歧义性必须解决：“如果户主想输入多于 7 个，系统将怎样解决？”于是，测试团队和开发团队一起来讨论这件事情，这就是我们迭代发现过程的本质。

3）第三步：确定测试条件

下一步是在测试用例中确定引发执行这个测试用例的特定条件。也就是考虑一下，什么条件引起用户在一个用例中执行特定事件的序列呢？

在这个过程中，测试人员要搜索用户步骤，发现引发特定测试用例的特定数据条件、分支等。每发现一个条件，测试人员都在矩阵中输入一个新的列表示这样的条件。

在这个过程中，只要简单的创建一个列，表明对于这个条件将发生哪些状态（有效、无效、不可用）就足够了。

- 有效（V）:为执行基本流，这个条件必须为真。
- 无效（I）:这个条件将激活备选流，引发特定场景。
- 不可用（N/A）:所确定的条件无法应用于测试用例。

我们来看一个简单的“控制灯”的用例，这里有三个改变系统行为的条件要考虑：

- 按下按钮少于 1 秒。
- 按下按钮超过 1 秒。
- 按下按钮超过 1 秒后松开。

它们将分别触发场景 1，2，3。下面列出这个用例描述。

用例举例：控制灯		
用例名:	控制灯	**用例类型：**
业务参与者:	住户，灯排	
描述：	这个用例规定了开灯和关灯的方式，以及如何根据用户按下开关的时间长短来变明或变暗。这里“按钮”指的是“开/关/变暗”按钮。	
前置条件：	按钮必须是能变暗的，同时能被编程控制某个灯排。	
后置条件：	系统记住了按钮的明暗程度	
基本流程:	当住户按住按钮的时候，开始基本流。 当住户按住按钮但在定时周期结束前松开按钮，系统按下面方式改变灯的状态： 1，如果灯是开着的，那么关灯。 2，如果灯是关着的，那么把灯开到最近记忆的明亮程度上。	
备选流程：	当住户按下按钮超过 1 秒钟，启动备选流。 当住户按住按钮的时候： 1，灯的明亮程度以最大值的 10%的速度平滑增加。 2，当到最大值以后，灯的明亮程度以最大值的 10%的速度平滑减弱。 3，当到最小值以后，重复步骤 1。 当住户松开按钮的时候： 4，用例终止，并把亮度停留在当前等级。	
特殊需求:	性能：对于住户任何可能动作，从控制板到系统的响应时间小于 50 毫秒。	

现在我们创建一个测试矩阵，把每种情况预期的结果记录下来。

有确定条件的控制灯测试用例							
测试用例编号	**场景**	**描述**	**条件：按下按钮少于 1 秒**	**条件：按下按钮超过 1 秒**	**条件：按下按钮超过 1 秒后松开**	**条件**	**预期结果**
1	1	基本流:按下按钮后在 1 秒之内松开按钮	V	I	不适用	灯开	灯灭了

2	1	基本流:按下按钮后在 1 秒之内松开按钮	V	I	不适用	灯灭	灯亮了
3	2	备选流:持续按下按钮超过 1 秒	I	V	不适用	N/A	明亮程度连续上升或下降
4	3	备选流:按下按钮持续超过 1 秒以后松开按钮	I	I	V	不适用	灯停留在最后的亮度

4）第四步：增加数据值完成测试用例

我们已经有了很好的进展，现在来确定完全的测试一个用例所需要的所有条件。用例只是对条件、场景、路径的描述，并没有具体的值，所以还需要到补充规范去找到一些有效的数据范围、接口协议等等信息。

这恰恰也是利用测试用例解决当初的用例定义的需求的时候了，这也包括把最大/最小性能、最大/最小数据范围、最大/最小负载的定义和自行期间的数据量结合起来。

一旦确定了数据范围，就可以把它填入测试用例的矩阵，如下表所示。

有确定条件的控制灯测试用例							
测试用例编号	**场景**	**描述**	**条件：按下按钮少于 1 秒**	**条件：按下按钮超过 1 秒**	**条件：按下按钮超过 1 秒后松开**	**条件**	**预期结果**
1	1	基本流:按下按钮后在 1 秒之内松开按钮	<1 秒以 0.1 秒的间隔	I	不适用	灯开	灯灭了
2	1	基本流:按下按钮后在 1 秒之内松开按钮	<1 秒以 0.1 秒的间隔	I	不适用	灯灭	灯亮了
3	2	备选流:持续按下按钮超过 1 秒	I	1～60 秒	不适用	N/A	明亮程度连续上升或下降
4	3	备选流:按下按钮持续超过 1 秒以后松开按钮	I	I	V	不适用	灯停留在最后的亮度

三、管理测试覆盖

显然，即使简单的用例也可能产生大量的测试用例，所以许多应用程序彻底的测试不太可行。但是在高要求的系统中，不但要求彻底测试，还需要返回一次。如果没有工具的话，就会带来很大的工作量。下面的几条指南可以提供一些帮助：

- 选择最恰当、最重要、最关键的测试用例进行最彻底的测试。通常这些用例是主要的用户接口。另外从架构上看很重要，或者对用户来说呈现一种冒险或困难，也可能存在没有发现的缺陷的地方。
- 在成本、风险和验证该用例的必要性之间进行权衡，并据此选择要测试的用例。
- 使用在迭代计划中已经使用的优先级算法来确定对用户的相关重要性。

4.8 通过优先级评价发现设计重点

当我们致力于提升自己能力的时候，就发现会经历这样几个阶段：首先是熟悉情况，了解各种各样的问题，在这个过程中满满的获得成功和失败的经验。其次是仔细总结和思考，想办法把各种各样的问题和经验进行梳理，把问题的逻辑过程进行条理性的描述，这样对很

多问题的解决就提供了有效的方法。

很多人只做到了这一步，但是真正有能力的人会再仔细考虑，所有这些问题哪些是最重要的？哪些是最关键的？哪些是可以先放一放的？哪些是必须优先解决的？抓住了重点，就可能抓住了纲，使事情解决起来更加的快速有效。

作为设计也是这样的，仅仅发现和列出了所有的功能这只是第一步，进一步的挖掘我们需要对所有的功能优先级评价，从而发现最重要的功能，这也是分析和设计的精化过程，对软件设计来说是很重要的一步。

在确定重点功能的时候，粒度也是一个问题，面对大批的事无巨细的功能，耗费巨大的精力对每个细小的功能进行优先级排序是得不偿失的，我建议优先级讨论的粒度应该粗一些，比如若干功能组合成的能力（或者说是主题），这样的归纳本身也是一个对需求进行梳理的过程

一、确定能力的价值

能力的价值其实很难确定，当产品负责人提出“要根据业务价值确定优先级”的伟大建议的时候，业务价值到底是什么？因此需要有更确切的指导原则，我们必须考虑 4 个因素：

- 获得这些能力所带来的经济价值。
- 开发新能力所需要的成本。
- 开发新能力需要学习知识的量及重要性。
- 开发这些能力所减少的风险。

由于多数项目都考虑节约开支与赚钱，所以前两个因素往往比较受到重视，但为了最优的确定优先级，对于学习与风险的考虑也是重要的。

1）价值

确定优先级的第一个因素是能力的经济价值，获得了能力的能力，可以让公司获得或者节省多少钱？这本身就具有“要根据业务价值确定优先级”的含义。

确定能力价值的理想方法，使估计它在一段时间内（几个月或者几年）所带来的经济影响。确定能力的经济回报是很困难的事情，它通常要求对新产品的销售数量、每次销售的平均价格（包括后续销售和维护协议）、销售上升的时机等都进行估计，这种估计难度颇高。因此常常用一些替代方式对价值进行估计，由于现有用户合意性常常与价值有关，所以常用的方法是用合意性的非经济度量来表示价值，这是后面我们会讨论的问题。

2）成本

很自然的，开发能力的成本是优先级的重要考虑因素。很多能力看起来不错，但了解倒它的成本之后可能会改变主意。由一个常常被忽视的问题，就是成本会随时间变化。假定开发某个能力现在需要 4 周的时间，但 6 个月以后我们会由于所获得的知识的改变，还需要花额外的 3 周对已经开发的能力进行改变，那就需要等一等。或者 6 个月之后预计我们会发现一种更简单的方法来开发这个能力，那为什么不等一等呢？

3）新知识

在很多项目中，整体工作中的大部分是花在对新知识的追寻上，重要的是承认这种努力，并把它看作是项目的一个重要组成部分。获取新知识很重要，因为项目开始的时候，我们并不知道项目结束前我们所需要知道的所有的事情。在开发过程中所形成的知识包括两类：

- **关于产品的知识**：这是关于将要开发的能力的知识，一个小组关于产品的知识越多，就能更好的作出产品特性和能力的决策。
- **关于项目的知识**：这是关于如何建立产品的知识，例如将要使用的技术、技能、小组共同工作所需的相关知识。

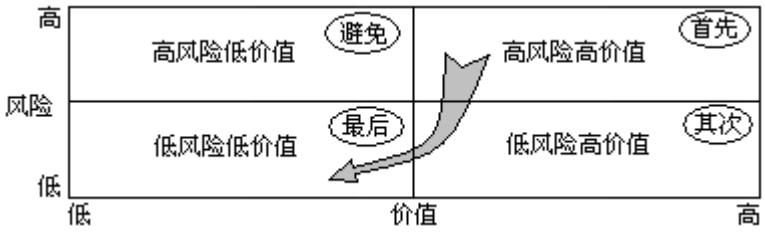
4）风险

与新知识的概念密切相关的是风险。几乎所有的项目都蕴藏着大量风险。风险指的是目前尚未发生但是可能发生，而且会妨碍或者限制项目成功的事情，在项目中不同的风险包括：

- 进度风险
- 成本风险
- 能力风险

此外，风险还可以分为技术风险和商业风险。能力上的高风险和高价值存在着典型的竞争，要在它们之间做出选择先要考虑每种方法的缺点。风险驱动的开发可能花了很多努力，到最后才发现这个能力其实是不必要的。价值驱动的开发首先开发了很多高价值能力，到最后才在一个不经意的地方发生危及整个项目的风险，造成项目崩溃。这两种情况其实都有可能发生。

解决的办法是不要让风险也不要让价值在确定优先级的时候占绝对主要地位。我们考虑一下下面的关于风险价值关系 4 象限图。开发的优先级是从高风险高价值开始，沿着下图所示的曲线安排比较合理。



图中表达的方法是要“避免”高风险低价值的能力，但这个避免不是绝对的，今天被认为是要避免的，6 个月后随着开发的进展，或者风险降低，或者价值提高，这个能力可能会处于首先要处理的位置。

要综合 4 个优先级因素，基本的工作方法是这样的：

首先考虑能力的价值与现在就开发它所需的成本之间的关系，这会给您一个初始的优先级顺序，具有高价值/成本比的主体应该首先完成。

其次，考虑其它优先级因素把主体向前或者向后移动。假定一个能力根据价值/成本比具有中等优先级，但发现这个能力有很大的技术风险，这会导致这个能力在进度表上往前移动。

初始排序并不是很正式的，甚至只是产品负责人在一张纸上的涂鸦，然后产品负责人向开发小组提出自己的想法，小组作出恰当的评估，最后由产品负责人确定最后的顺序

二、确定合意性优先级

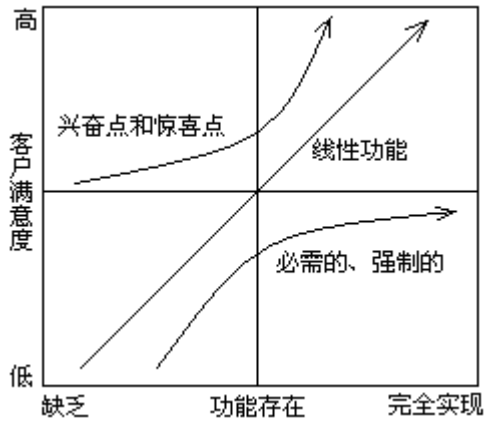
产品的价值很大程度上来自于客户满意度，因此在考虑合意性优先级的时候，我们需要研究产品能力与用户满意度的关系。

1）客户满意度的 Kano 模型

最先提出为新产品发布提供优先级指导的，是日本管理学家狩野纪昭（Noriaki Kano），他的方法是把能力分成 3 类：

- 作为阈值的能力，或者说必需的能力。
- 线性能力。
- 兴奋点和惊喜点。

这几种能力与客户满意度的关系见下图，这个图称之为 Kano 模型。



阈值能力是产品要成功必须具备的那些能力，也常常也称之为必需的能力。改善阈值能力和增加阈值能力的数量对客户满意度并没有多大的影响。

由于产品要有那些必需的能力才能在市场上生存，应该强调优先开发所有那些阈值能力。但并不是一定要在第一次迭代中就开发所有产品必须能力，但是由于用户把这些能力看成强制性的，他们必须要在发布产品之前变成可用。从图上可以看出来，有的时候只需要满足必须能力的部分实现就可以了，因为对必须能力一定程度的支持以后，客户满意度的上升幅度就趋于平缓。比如用记事本写文章，它只有单次撤销，当然有象 Word 那样的多次撤销能力也很好，没有也可以，即使加上了多次撤销客户对记事本的能力满意度也不会提高多少。

线性能力是一个处于“越多越好”状态的能力。这里客户满意度会随着这部分内容的上升而直线上升，而产品的价格也与线性特性相关。

最后，兴奋点和惊喜点是那些提供了很高的满意度，并常常可以为产品增加额外价格的那些能力。但是缺少兴奋点和惊喜点并不会让客户满意度降到中性以下。兴奋点的特点是，用户会喜欢它，但如果没有它用户也不会拒绝这个产品。实际上兴奋点和惊喜点也被称之为未被了解的需求，因为在大家没有很好的研究它的时候，并不知道自己需要它。

一般来说的优先级是：首先是阈值能力，其次是现行能力，最后的兴奋点能力。

2）用 Kano 模型评估能力

在敏捷开发项目中使用 Kano 模型最简单的方法，是考虑每个能力，对它所属的类型作出有根据的猜测。不过更好的办法是与客户或者用户进行讨论，确定每个能力的类型。一般来说，只需要向少至 20～30 个人进行一次书面问卷，就可以准确的知道需求的优先级。Kano 建议通过两个问题来确定一个能力的分类。

第一个问题是能力存在形式：如果一个产品中有这个能力用户会怎样？

第二个问题是能力缺失形式：如果一个产品中没有这个能力用户会怎样？

每个问题都可以通过 5 点的度量方式进行回答：

（1）我希望这样

（2）我预期就是这样

（3）我没有意见

（4）我可以忍受这样

（5）我不希望这样

例如，我们正考虑要建立一个运动员网站 SwimStats，假定我们正在考虑 3 个新能力：

- 查看一张图，显示一个运动员在过去的赛季中某个项目的成绩能力。
- 让运动员发布自传简历的能力。
- 让注册的网站成员上传照片的能力。

要确定这些能力属于哪个类别，就需要对用户进行调查，可以问他们以下问题：

合意性调查表

	题目	1	2	3	4	5
1	如果可以绘制一个运动员在过去赛季中某个项目上的成绩，您觉得怎么样？	√				
2	如果不可以绘制一个运动员在过去赛季中某个项目上的成绩，您觉得怎么样？					√
3	如果有让运动员发布自传简历的能力，您觉得怎么样？		√			
4	如果没有让运动员发布自传简历的能力，您觉得怎么样？					√
5	如果有让注册的网站成员上传照片的能力，您觉得怎么样？	√				
6	如果没有让注册的网站成员上传照片的能力，您觉得怎么样？			√		
说明	1，我希望这样 2，我预期就是这样 3，我没有意见 4，我可以忍受这样 5，我不希望这样					

对这一叠调查表，需要一种方法，对用户的观点做出评估。

2）将答案分类

我们可以使用一个分析矩阵对每个人的回答做出判断，如下表所示。

答案分析						
		能力缺失				
		希望	预期	无意见	忍受	不希望
能力存在	希望	Q	E	E	E	L
	预期	R	I	I	I	M
	无意见	R	I	I	I	M
	忍受	R	I	I	I	M
	不希望	R	R	R	R	Q
M 必须的 L 线性的 E 兴奋点 R 反对的 Q 存在疑问的 I 无所谓的						

在这个分析表上，如果用户对这个能力的存在认为他预期、无意见、可以忍受，但是不希望没有，这一般可以认为是阈值能力。如果用户希望有这个能力，而且不希望没有这个能力，那么这是一个线性能力，客户满意度会随着这样的能力增多而增加，这个能力也是强制性的，例如上面关于绘图的回答。如果客户希望有这个能力，但对没有这个能力它的反映是预期、无意见、可以忍受，那这一般是一个兴奋点能力。

最后把每个人的分析统计在一张表上，就可以看出每个能力的性质了，如下表所示。

用户调查结果分析							
能力	E	L	M	I	R	Q	类别
对项目成绩作图	18.4	**43.8**	22.8	12.8	1.7	0.5	线性
上传照片	8.3	30.9	**54.3**	4.2	1.4	0.9	必须
发布自传简历	**39.1**	14.8	**36.6**	8.2	0.2	1.1	兴奋点，必须

可见，上传照片是必须的，对成绩作图是线性能力。对于发布自传简历，一部分人认为是必须，还有一部分人感觉是一个兴奋点，这就需要对结论进一步分析。

但是用户可能会有不一致的回答，比如第一个问题是：我不希望这样；对第二个问题他也回答：我不希望这样。这就比较难处理了。如果是个别情况，就可以把它作为奇点去掉，如果是比较普遍的情况，您可以考虑按照某些因素对客户分组，比如把小公司与大公司分开，或者按照客户在公司的角色进行区分，就会发现这类回答是有一定群体性的，再来分析他们的回答得到更加深入的结论。某些情况下也可以利用专家来判断，开发小组共同对下一次发布所考虑的每个能力进行评估，通过价值和成本的分析来订出不同能力的权重。

4.9 设计文档编写的若干建议

一、为什么要书写文档

很常见的情况是，开发人员对于书写文档有种种非议，但是一个正规的软件项目，即使带来了表面上的工作效率降低，也必须花费精力书写文档，为什么呢？

任何时候，只要是两个人以上参与项目，都离不开口头交流，但是每个人在传递这个口

头交流的内容的时候，都会把原来的意思歪曲一点点，人类的记忆能力就是这样的。一个软件项目永远离不开口头交流，我们不必要去停止这种交流方式，但是当开发人员比较多的时候，就可能带来很多麻烦，某些情况需要大家都知道，需要其他人员与之交互等等，解决的办法就是把所有的情况记录下来，或者是要强调书写文档。利用文档记录有五个好处：

1）拓展人脑所掌握的记忆范围

在任何一个大到必须编写文档的项目中，信息的含量都会超过一个人可以掌握的规模，书面文字可以供将来参考，而不会像记忆一样慢慢的褪去。

2）为项目团队所有成员提供相同的信息

书面文档阅读的时候内容是相同的，所有的人员都阅读相同的材料，这是任何人对人的口头交流无法达到的效果。

3）减少项目人员流动的困难

项目中发生人事变动是正常的，当老成员离开，新成员加入的时候，任何口头交流都很难让人很快地掌握情况，一份很好的文档可以让他花更少的时间融入到团队中去。

4）保护智力资产

在大多数情况下，项目中只有少数的人理解问题域，他们知道是否需要变更，软件是否存在某些漏洞，内心中对软件是不是有什么新观点。如果这些人把他们掌握的信息以正规的方式记录下来，项目对他们的依赖就会减小，如果他们获得了更好的机会，他们的智力资产也不会被带走。

5）帮助书写人员更好的理解问题域

以书面形式描述设计方案，将促使相关人员花精力以比口头交流更标准和更准确的方式表达。人们在写作过程中可以发现自己对某些问题的理解存在漏洞，甚至概念上也是不完整的。难怪人们说：“文档本身并不重要，重要的是写文档的过程。”当然，这样写出来的文档对别人也是重要的。

很多公司实际上是很严肃地对待文档的，但他们并没有从文档中得到好处，相反在实践中口头交流仍然是主要手段，其关键原因是所书写的文档并不是为了开发而是为了应付检查。从格式上看很漂亮，内容面面俱到，但在文档组织上把信息分散到各个地方，不容易查找，使文档提供的好处尽失。如果我们希望项目开发小组很好的使用文档中的信息，就需要考虑什么内容是开发小组最关注的，怎么组织文档才是对开发来说最容易查找，如何编写才是最容易阅读和有效的

二、设计文档编写的建议

1，防止编写成智力拼图

许多文档写成了类似智力拼图的玩具，很多不同的相关定义与内容散落在文档的不同地方。读者不会预料到在文档的其它地方会不会还有相关说明。为了阅读这样的文档，人们需要把一切都记在脑子里，智力拼图是把散落在各处的小块拼在一起，但人的大脑很难做到这一点。

要记住，任何大的文档都不同程度的具有智力拼图性质，但我们的任务是减少智力拼图块的数量。不要让阅读者在一个地方看到的信息块做出了推论，而在另一个地方才发现这个理解是错误的，这就很危险。

组织文档的原则，大多数情况下是防止出现智力拼图的情况，如果需要，可以把文档中对某个名称的定义和描述都找出来，统一放在某个固定的、唯一的位置上，必要的时候还可以建立术语表。如果无法将相关信息都集中在一个章节，就需要增加引用页面，这样读者就不会迷惑了。

我们这个课程讲义的编写本身就是一个文档化过程，除了定义清楚问题域以外，整个课

程的思考方式是基于问题、研究对策、寻找解决方案，而不是依据于某个人说过什么，某个标准是什么（何况标准也是在变化的）。在书写方式上，以读者为中心，力求通俗易懂、内容连贯、条理清楚。在内容组织上，力图减少不必要的拼图游戏，减少不必要的前后查阅。相似的概念和名词，只要在各个章节中，都力图自成逻辑体系，使得读起来省力清晰。

这些特征，在文档的编写中都是需要注意的。

2，防止编写成鸭叫文档

有些文档编写出来为什么相关人员不愿意看呢？为什么自认为似乎已经描述的很清楚的文档，在读者感觉模糊不清难以阅读呢？我们来看看下面的描述：“机票预定数据验证函数应该验证机票预定数据”。这样的句子确实使人莫名其妙，这到底是什么意思？如何测试它？这很容易使人联想到一种鸭叫演讲，单调、温顺、平庸、符合官方标准，但是叫人提不起精神，看起来描述了很多，但实际上什么也没有描述。

把自己写的句子大声念出来，如果发现自己写了很多毫无疑义的句子，只是一味的迎合标准要求，就要考虑重新框定问题，使文档写出来具有精神，读者读起来也有精神。

3，不要为了文档而文档

软件质量保证使每个开发组织都很关注的，为了保持一致的软件质量，我们需要一套独立定义的质量准则，所以，过程的定义与文档规范就成为很多组织很关注的内容。但也会发生文档的使用与功能正确实现无关的情况，由于文档记录要与文档标准保持一致，结果使用起来将非常繁琐而散乱。

这种书写文档的理念只是为了应付检查，而不是促进效率与更好的开发，是为了文档的本身，是为了文档而文档的书写方式。书写这种文档的目的只有一个，那就是证明我们正在做和官方保持一致的事情，如果有什么差错，那不是我的责任。至于项目能不能完成，那不是我考虑的问题。

很多文档都没有被任何人阅读，其原因是人们无法理解他，不适合相关人员的习惯，也没有办法把文档与自己的开发工作联系起来。当然，确实很多评审只关注文档格式而不是文档有效性，只关注文档字数多少而不是内容的思想性。这种东西不是单靠开发单位能解决的，如果这样的评审是一定存在的，那一个建议就是：开发两套文档，一套用于评审，一套用于开发小组内部。这看起来多费了时间，但所费的时间是值得的，总比把应付评审的格式化文档用于实际开发从效率上要高的多。试试看，就会发现这个建议的意义。

4，图与文的协调

谈到设计文档就免不了使用图形表达，不过在图和文的协调上需要下点功夫。图比较适合说明关系，而文字比较适合表达细节。这两者是不可互相替代的。作为设计，如果没有图，相关人员对问题域就可能不能得到一个整体的概念。但是没有详尽的文字表达就不可能定义开发，也不可能使开发指向我们所需要的方向。图不需要面面俱到只需要表达重点需要表达的内容，而文字的详细程度应该足以定义开发的关键点，但也要给开发人员留出适当的发挥空间。

有的单位热衷于把类图直接生成代码，这是一个误会，也是把图用错了地方。代码是事无巨细的，而且与编码员个人的特征有很大关系。图没有必要画得那么细，也不可能画的那么细。

画图要表现出聪明，要说明每个结构策略的原因，也要说明所采用的策略所带来的负面影响，仅仅按照某个模式来做设计那是初学者的事情，一个高级设计人员的思维方式，是突破一切框框，目的就是一切。

第五章 软件复用与框架技术

在考虑变与不变分离的时候，应该看到很多情况下业务流程具有相似性，而业务单元却各自独立的变化，这种分析对于复用和框架技术尤其重要。

前面已经讨论过，框架是可以通过某种回调机制进行扩展的软件系统或者子系统。框架的概念主要来自于对“重用概率”的分析：单元粒度越大，重用概率越低，但是重用价值越大。反之，单元粒度越小，重用概率越高，但是重用价值越小。框架的智慧在于，在单元粒度比较大的情况下，追求高的重用概率。

框架是一种特殊的软件，它不能提供完整无缺的解决方案，但为解决方案提供和很好的基础，它是一种系统或者子系统的半成品，框架中的服务可以提供最终应用系统的直接调用，而框架中的扩展点题共有开发人员定制的“可变化点”。从而达到一个比较大的框架单元，就可能有比较大的重用概率。

在引入软件框架以后，软件架构决策往往会体现在框架设计之中。软件架构是比具体代码更高一个抽象层次的概念，架构必须被代码体现和遵循，但任何个一段具体代码都代表不了架构。

研究框架技术，首先得从需求模式开始研究。

5.1 利用模式重构问题域与架构

一、对功能分解的再讨论

功能分解方法有时候也称之为“自顶向下逐步求精法”，这是一种长期主导软件业的分析思想，其基本原理在于如果一个功能对人脑来说太大，以至于不能立刻勾画或者实现它，那么就把它重复分解成小到足以能够处理的子功能。

我们应该看到，功能分解只是诸多问题解决技术的一种，如果一个分析人员不去关注历史上各个项目类似功能的成功实现方案，只是一味的就功能谈功能，往往不能得到很有效的划分方法。我们应该看到，把高层问题分成子问题事实上可以有多种划分方法，而且早期也没有办法判断哪种分解方法更好更有效。只是在开始设计以后，才能够评价一种特定的分解在实现上是不是合理，但到了那个时候，去纠正原先的高层的错误已经太迟了。因此，比较常用的办法是：

- 尽可能收集和应用已经存在的、经过时间考验的划分和设计。
- 把新的方法融入到老的是设计中去，而不是总是另起炉灶。

二、利用模式解决划分中的困难

把上面的思想进一步延伸，就可以形成一种“模式”的概念。所谓“模式”强调的是某种功能单元可能被使用上百次，但使用的方式却不尽相同。模式是一种灵活的思想，运用它却需要智慧和想象力，因为没有两种完全一样的功能需求。

“模式”这个词最初来自于 Christopher Alexander（克里斯多弗. 亚历山大），在城市规划和建设的过程中，他发现许多相同的原则，在 Pattern Language（模式语言）这本书中，他把建筑学中大量简单事务划分为若干模式（咖啡馆、街角杂货铺、天窗、与腰等高的柜台等）当拥有丰富的模式就相当于拥有的大量的词汇，有助于使设计更完美。这个思想同样可以用

在软件分析和设计中，当拥有大量模式的时候，就可以使自顶向下的分解变得更加主动。

在软件开发的早期，由于强调创新，软件领域内不会产生可靠的结果，每个新设计都会包含大量的没有经过测试的新思想，而这些未经测试的新思想经常会失败。

当一个工程领域成熟以后，我们就有机会对大量经受了时间考验的设计进行组合和微小的修改，从思维方式上，就可以专注于解决良好定义的问题集合中的问题，而不是一切都从头开始。就好比做变压器，现在由于已经有了经过考验的模式，只需要告诉他基本参数，他就可以用相同的形式做出用户需要的变压器，而不要从磁通量开始计算和设计。

不过，软件的创新仍然在继续，所以我们就可以把软件分成两的部分：

- 条理性工程：应用经过考验的模式，通过恰当的组合和微小的修改达到目的。
- 探索性工程：对新的各种各样的设计的非结构化探索。

当我们在分析系统的时候能够很好的分开这两个部分，整个分析和设计过程就可能变得及其有智慧。模式的意义并不在于死搬硬套，世界上的问题是千变万化的，但如果我们已经有了大量问题家族和解决方案集合，我们的解决方案就可能受到它的启发，这种启发往往是极其珍贵的。

三、模式的合成与分解

如果已经存在大量的分析和设计模式，就可以使我们在较高层次上的功能分解变得充满信心。没有经验的分析员在分解的时候总是试探性的尝试使用各种分解方法，希望从中得到某种启发，有经验的人往往可以根据已知的模式进行恰当的组合，使将来的程序运转起来更加简单。

有经验的设计师在实现功能分解的时候好像与别人没什么不同，但是由于他们掌握了大量的历史信息，直到某种划分在实现上更加可靠和方便。他们力图把高层次的模式分解成子模式，总是把这种子模式与已知的设计结合起来。这种抽象问题的分解演绎成一个能够构造的组件，结果就使复杂问题的解决有了保障。

四、发现需求的变化规律

今天对软件开发影响最大的，恐怕就数需求变更了。需求的蠕变，可能使开发成本上升，交付时间推后，产品质量下降，所以关注变更无论对分析和设计都是相当重要的。

从需求分析的角度来看，首先要做到的是尽可能全面地收集需求，分析需求，确保不是由于前期需求分析的漏洞造成后期无畏的变更，当然如果发生了这种变更，无疑责任是在需求分析师和他的团队身上。

但是很多情况下需求变更是客观存在的，这是今天竞争激烈，变化越来越快的商业环境所决定的。如何灵活快速适应变化甚至创新求变，成为企业生存的头等大事。当业务发生变化的时候，难道需求不会发生变化吗？复杂性和易变性是信息技术（IT）必须面对的现实，无论是构建新的应用、替换现有的应用，以及及时处理各种维护与改进的要求，都是处理各种复杂情况，这种对于复杂性和易变性的挖掘和发现，是对产品开发的巨大挑战。考虑业务的易变性问题，需要把变和不变进行恰当的分离，大致的情况可以分成两种情况：

- 很多情况下业务流程具有相似性，而业务单元却各自独立的变化，这在业务粒度比较小的时候尤其普遍。
- 另一种经常遇到的情况，是业务单元具有相似性，而业务流程将会发生改变的情况，这在业务粒度比较大的时候很常见。

如果需求人员能够很深入地把这些变化规律找出来，设计人员就可以采取适当的设计措施，使产品有针对性地适时、快速的响应变化。这种具备恰当、明智的应变措施的产品，将

可以在变化的环境中高质量长周期的运行。但如果需求人员不能挖掘出这种规律，那设计人员的设计将会非常盲目，技术措施的应对点也会不对，虽然费了很多努力，但是设计无法与客观世界的变化特点融合。一般来说，盲目的应用设计模式只会使产品的性能变得更糟糕。

所以，从现代的观点来看，需求分析不仅仅是发现功能性、非功能性需求。也需要关注客观世界的变化规律，这也是一种需求，可能对后期产品架构设计造成重大影响。

5.2 需求模式

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。

在上个世纪 90 年代中期，一群优秀的开发者（Gamma、Helm、Johnson 和 Vissides）发现软件设计中存在不断重复出现，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。
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。

一、通过业务事件发现模式

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：

模式名称：客户希望购买一件产品。

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。

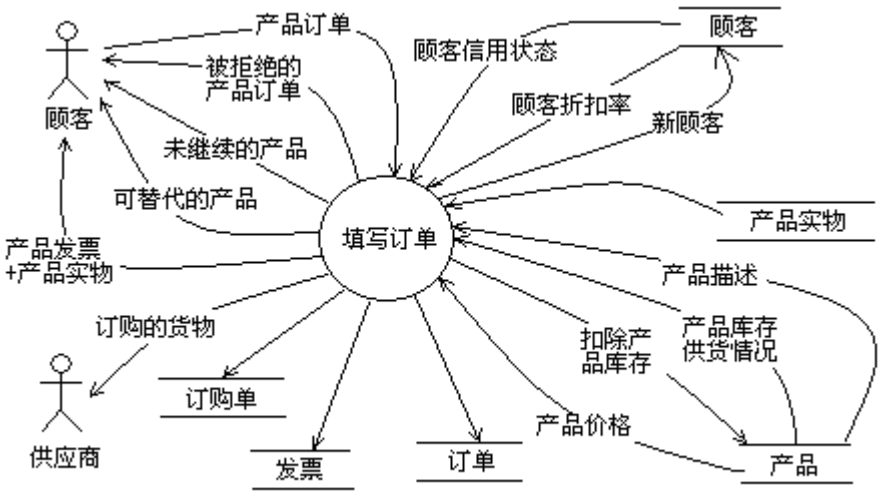
要点：一个组织有来自顾客的要求，希望提供物品或者服务。不能满足顾客要求可能导致顾客寻找其他供应商，有时在收到了订单时却没有产品。

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。

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。

一、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

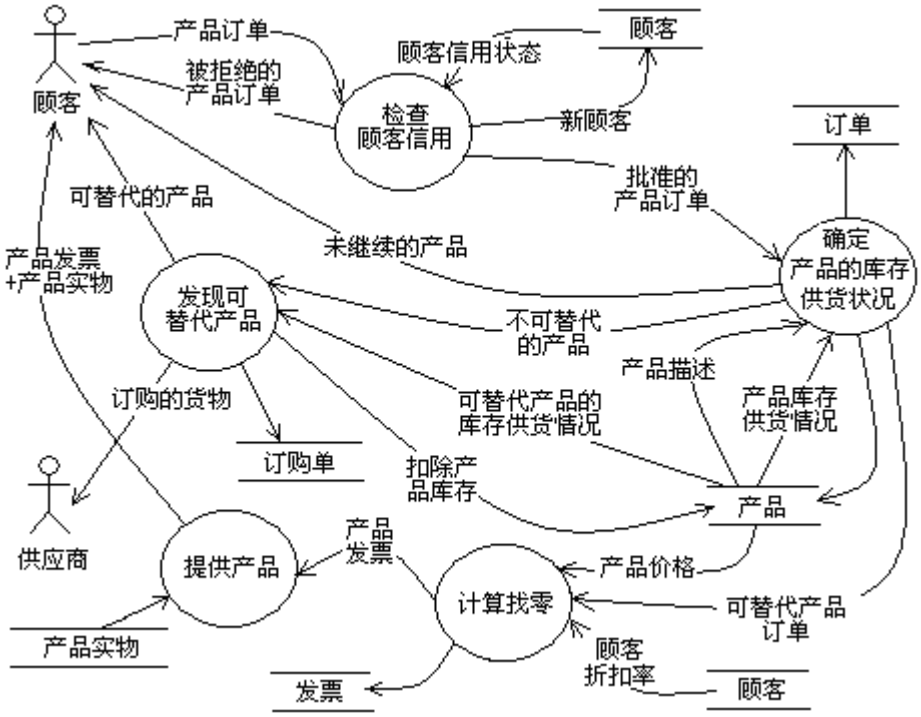
图中，上下文边界上有一些数据流（或者实物流），表明了该模式所做的工作。如果这些数据流中的大部分符合您的业务事件输入和输出，那么该模式就有可能用在项目中。

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。

二、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。



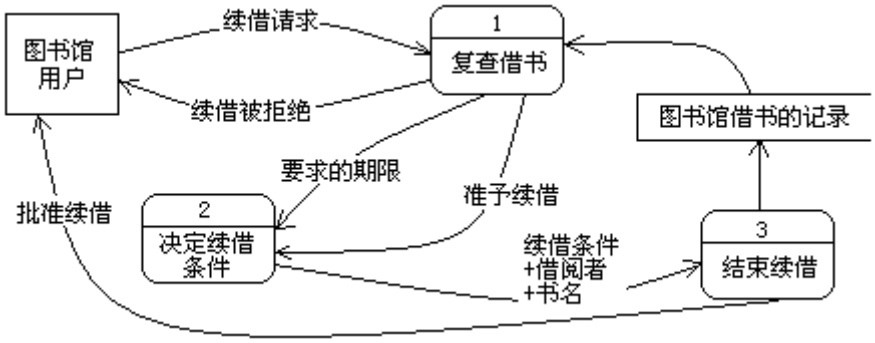
说明：该图把顾客希望购买一件产品这一模式分解为 5 个子模式（一组功能相关的需求）。同时展示了它们之间的依赖关系。每个子模式（用圆圈表示）通过命名的数据流与其它子过程、数据存储或者相邻系统相联。每个子过程也包含一定数量的需求。这个模型没有任意强制指定处理过程的顺序，而是关注于处理过程的依赖关系。例如，我们可以看到“确定产品库存供货情况”过程依赖于“检查顾客信用”过程，为什么？因为前一过程需要知道“批准的产品订单”才能开始工作。

上面的例子涉及了许多业务事件和主题领域，为了把它们记录下来已备查找，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。

三、特定领域的模式

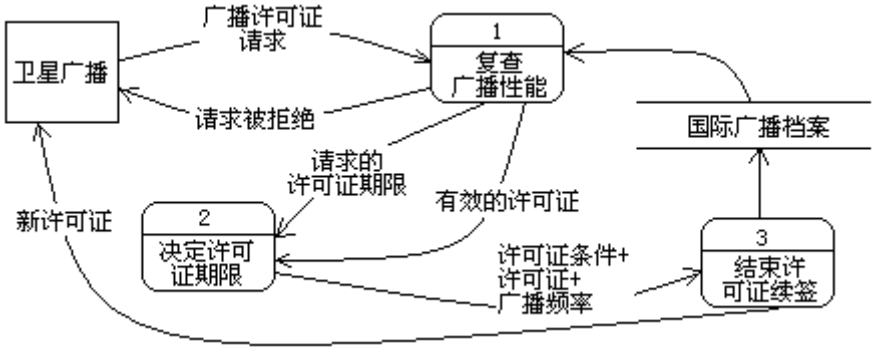
假定目前的工作是关于一个图书馆的系统，在上下文范围中几乎肯定有这样的业务事件：图书馆用户希望续借图书。下图展示了对该事件的系统响应模式，当一个图书馆用户提交了一个续借请求的时候，系统的响应要么是拒绝续借，要么是批准续借。



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了需求之后，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。

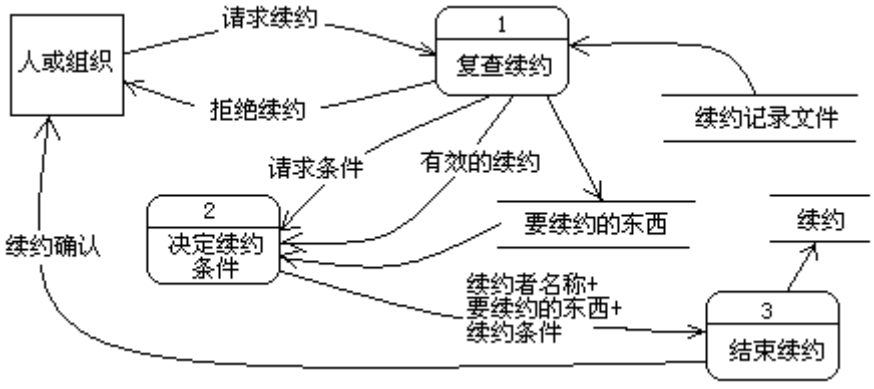
现在让我们把问题看得更深入一些，我们已经讨论了在特定主题领域确定和复用需求模式的想法，但是我们如何才能够在最初的领域之外使用模式呢？

四、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。它也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、设计模式

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？

- 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。
- 如果我们编写的软件能面向未来，那将永远处在设计阶段。
- 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：

- 针对接口编程而不是针对实现编程。
- 优先使用对象组合，而不是类的继承。
- 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：

- 开放-封闭原则；
- 从场景进行设计的原则；
- 包容变化的原则。

本章的题目是“适应业务单元变化的架构策略”，换句话说，就是在框架的层面讨论架构问题，我们的目标是建立一种结构，使“通过某种回调机制进行扩展的软件系统”成为可能，这样的架构，就有可能在比较大的粒度下提高重用概率，也使产品适应变化的能力得到提高。

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。

六、代码重构的问题与解决方案

虽然经过了系统架构和设计的重构，系统的结构已经得到了很大程度的改善。但是，最终我们还需要进行一个更低层面但绝对重要的重构工作，这就是系统代码重构。

我们在浏览一个系统代码后，通过经验及直觉就能发现的一些“坏味”，例如：

- 代码的方法过大。
- 系统中重复的代码过多。
- 类的子类中存在大量相同方法。

- 代码中过多的注释。
- 参数列表太长。

那么一般我们应该选择怎样的时机去解决这些问题呢？通常，有如下四种时刻最合适进行代码重构。反之，如果随意选择代码重构的时刻，则会使系统代码更加混乱：

- 当我们试图将新的功能代码集成进系统代码前。
- 当调试系统代码发现 Bug 时。
- 当我们进行常规代码评审时。
- 系统架构和设计重构结束后。

需要强调一点，在我们真正动手前，明确代码重构通常所遵循的原则也有着极为重要的意义：

- 必须创建相应的大量测试用例和测试脚本。
- 代码重构要遵循小步调的工作规模（小计划、小构想、小改动、小测试）。
- 先拿问题最严重或最危险的部分开刀。
- 一定要利用测试进行验证。如果测试失败，就需要重复进行上述动作。

执行代码重构，不是简单地依靠自己的直觉进行代码修改。往往还需要参考业界在代码重构实践经验的基础上，提炼出来的那些类似于设计模式的“代码重构模式”来进行工作。下面我们以几个最基本的代码重构模式作为示例，来看看前人总结的有用的重构经验。

1，方法抽取

问题：

浏览一个 Class 的代码，经常会发现该 Class 不同的方法代码中，会重复出现很多完全一致的代码。这些重复的代码与各个方法内的功能代码混杂在一起，非常难以理解这个 Class 的功能结构。

解决方案：

代码结构需要简捷明了，否则混杂在一起的代码难以理解，可以将这些重复出现的代码抽取出来，汇总在该 Class 适当的新方法中。原先调用的位置，只留下对新方法的调用代码。

2，使用有语义的变量名称

问题：

浏览系统代码，经常遇到的一个普遍问题，就是代码中大量使用了一些没有明显语义的变量，例如：代码中声明了一个变量 a，但是当你看到这个变量时，并不能立即明白其用途。

解决方案：

代码的可读性在很大程度上决定了系统代码的可维护性。根据统计数据显示，超过 50% 的代码重构时间都花在了理解原有代码上。因此，定义变量时，尽量使用一些与变量用途或使用逻辑相关的名称。

在实际的代码重构过程中，可以使用一些工具来进行辅助。许多开发环境都集成了重构的部分功能，这些工具在一定程度上可以帮助我们来避免不必要的编码错误，帮助检查代码的一致性，提高了重构的效率。但是，正如 Booch 曾经说过的一句名言：“依赖工具的人，将最终被工具所玩弄”。由于代码重构具有明显的经验要求，所以绝大部分还是需要依靠重构人员的智慧和经验。

无论是架构和设计重构，还是系统代码重构，都是相互依赖和相互联系的。如果我们进行架构和设计重构时，没有系统代码作为支撑，就不能完全掌握系统内部结构的关系。同时，一次增量式的小规模设计重构，非常有可能成为触发后续系统代码相应部分的重构动作。例如：我们对系统内一些元素的名称进行了修改，那么我们自然对相应的代码（包名称、类名称、方法名称等）进行修改。所以，在进行架构和设计重构时，如果能够详细记录设计变化

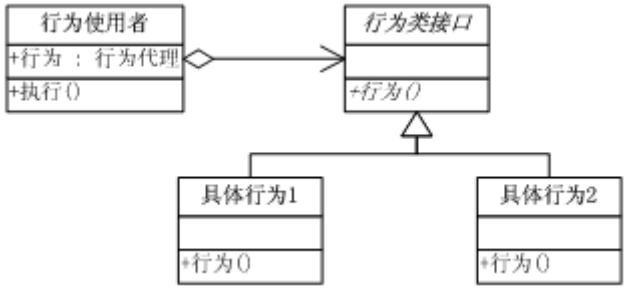
所涉及的代码，将对后续的代码重构非常重要。

下面我们会更详细的讨论这个问题。

七、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经有的对象。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。



5.3 处理类或者接口的变化

有一些变化包括类的变化或者接口的变化，当预测这种变化可能存在的时候，可以使用外观模式或者适配器模式。

一、外观模式（Facade）

1，意图

外观模式定义了一个把子系统的一组接口集成在一起的高层接口，以提供一个一致的处理方式，其它系统可以方便的调用子系统中的功能，而忽略子系统内部发生的变化。

2，使用场合

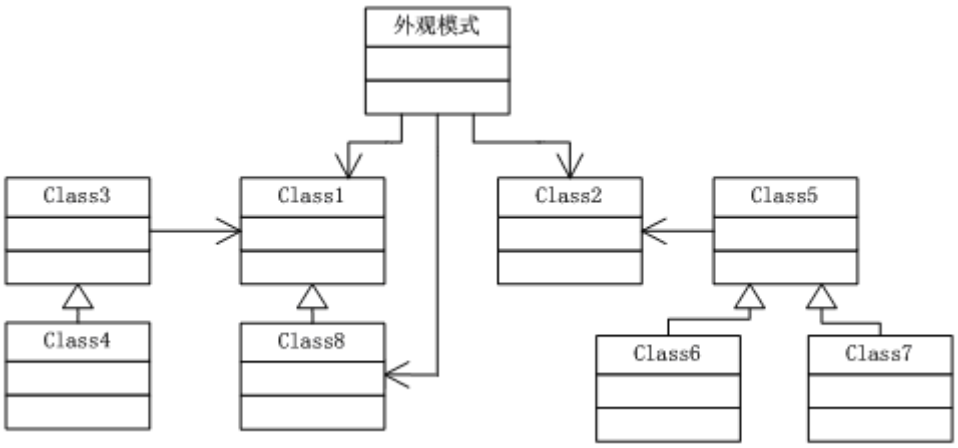
1）为一个比较复杂的子系统，提供一个简单的接口。

2）把客户程序和子系统的实现部分分离，提高子系统的独立性和可移植性。

3）简化子系统的依赖关系

3，结构

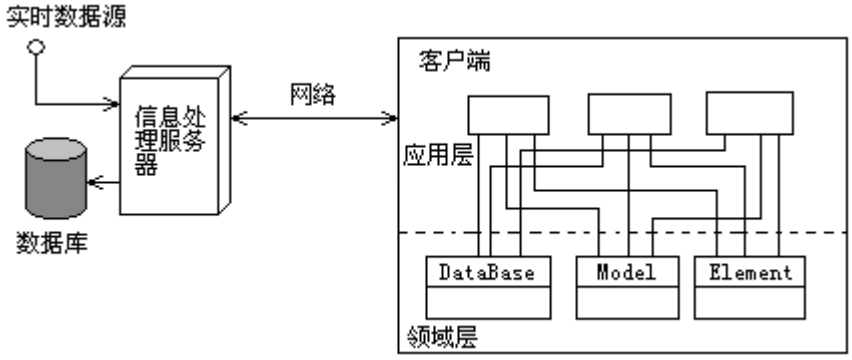
下图是外观模式的结构，由于该模式的引入，所以外界访问通过这个统一的接口进行，系统的复杂性得以降低。



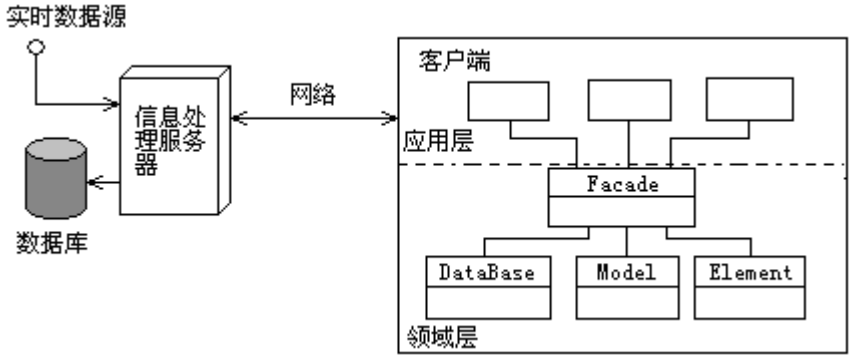
4，使用效果

外观模式为用户提供了使用子系统组件的统一的接口，使用户减少了处理对象的数目，并且使子系统使用简单。使用外观模式使子系统和客户之间实现松散耦合关系，由于用户针对接口编程，因此子系统的变化不会影响到客户的变化，而且有助于分层架构的实现。

如图所示某信息系统的总体体系结构图，其中客户端使用了分层结构。



在这个结构中，应用层与领域层的连接关系非常混乱，领域层中类的任何变化都可能引起应用层大量的未知变化，使系统的升级时成本很高，可维护性大幅降低。由于这种混乱是在类的（或者模块）的层面上，为了解决这个问题，我们可以使用外观模式，如下图所示。



这种模式一个显而易见的好处，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，这就使系统的高级维护成为可能。

二、适配器模式（Adapter）

在系统之间集成的时候，最常见的问题是接口不一致，很多能满足功能的软件模块，由于接口不同，而导致无法使用。在这种情况下可以使用适配器模式。

1，意图

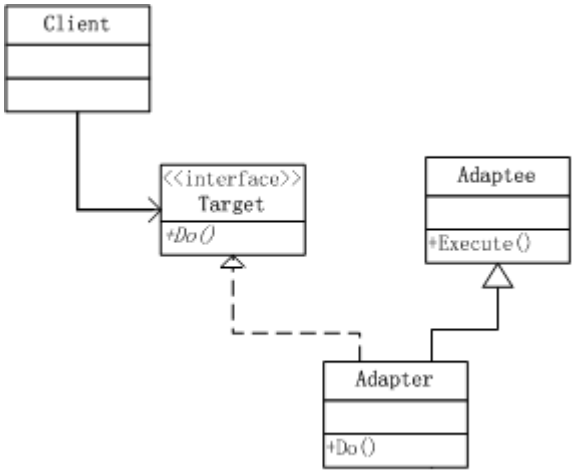
适配器模式的含义在于，把一个类的接口转换为另一个接口，使原本不兼容而不能一起工作的类能够一起工作。

2，结构

适配器有类适配器和对象适配器两种类型，二者的意图相同，只是实现的方法和适用的情况不同。类适配器采用继承的方法来实现，而对象适配器采用组合的方法来实现。

1）类适配器

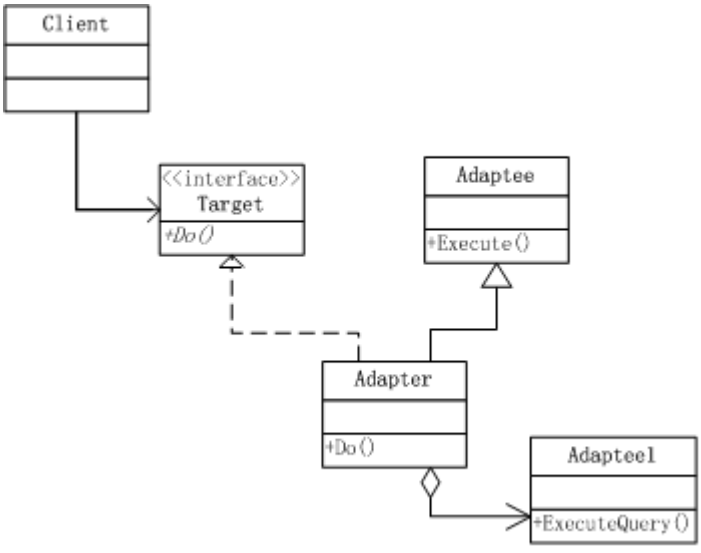
类适配器采用多重继承对一个接口与另一个接口进行匹配，其结构如下。



这种方式有一个限制，由于大部分面向对象语言不承认多重继承，所以当 Adapter 类需要继承多个类的时候就没有办法实现，这时，可以使用对象适配器。

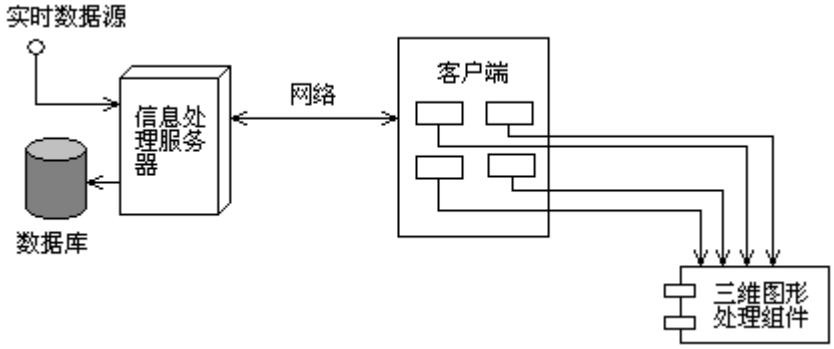
2）对象适配器

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

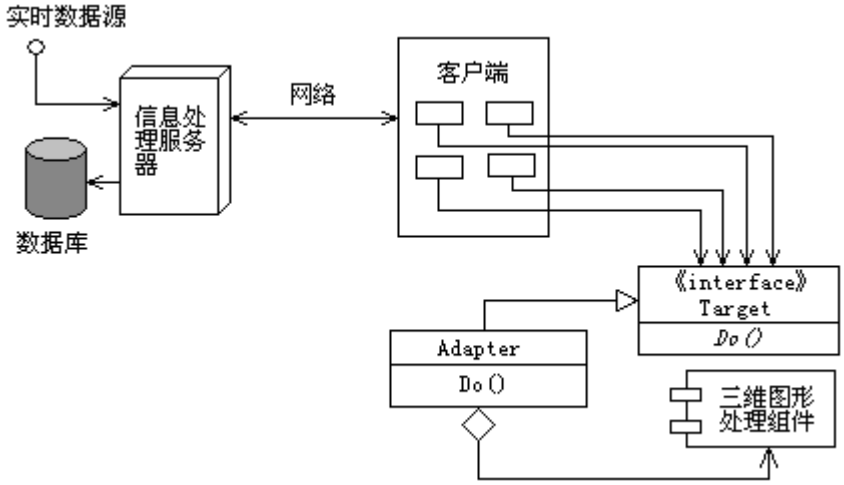


3，使用场合

当需要使用一个已经存在的类，但接口与设计要求不符合的时候，使用适配器模式可以是一个解决方案。例如，如图所示某信息系统的总体体系结构图，其中“三维图形处理组件”是外包开发，该组件接口随着版本升级可能发生改变。由于接口变化将会引起客户端多处发生改变，使系统的升级时成本很高，可维护性大幅降低。



可以使用设配器模式来屏蔽接口的变化，如下图所示。



5.4 封装业务单元的变化

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。如果我们遇到的是业务流程不变，但是业务单元可能改变，这种分离可以有多种方式，一般来说可以从纵向、横向以及外围三个方面考虑。

一、模板方法（Template Method）

1、意图

软件复用的关键是寻找相似性，在很多情况下，相似性表现为业务流程相似，但是业务单元具有特殊性。如果是这种情况，可以定义一个操作中的业务流程骨架，而将一些业务单元的实现延伸到子类中去，使得子类可以不改变一个业务流程的的结构，即可重新定义该业务流程的某些特定业务单元。这里需要复用的是业务流程的结构，也就是操作步骤，而步骤的实现（或者是业务单元的实现）可以在子类中完成。

2、使用场合

1）一次性实现一个业务流程的不变部分，并且将可变的行为留给子类来完成。

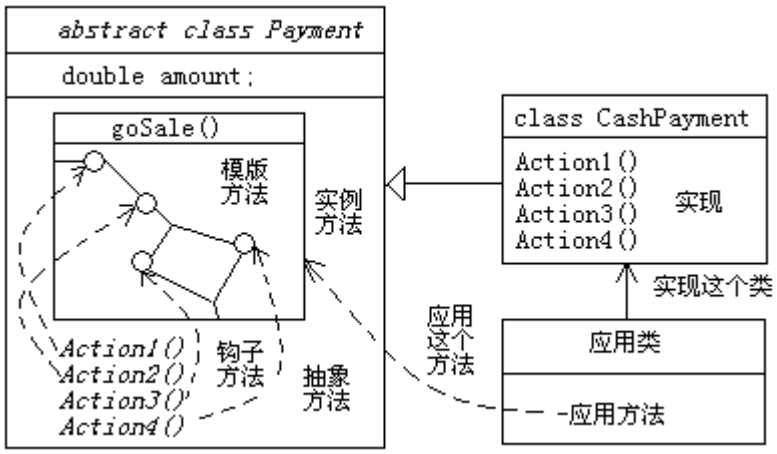
2）各子类公共的行为应该被提取出来并集中到一个公共父类中以避免代码的重复。首先识别现有业务的不同之处，并且把不同部分分离为新的操作，最后，用一个调用这些新的操作的模板方法来替换这些不同的代码。

3）控制子类的扩展。

3、结构

模板方法的结构是使用一个抽象类，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：

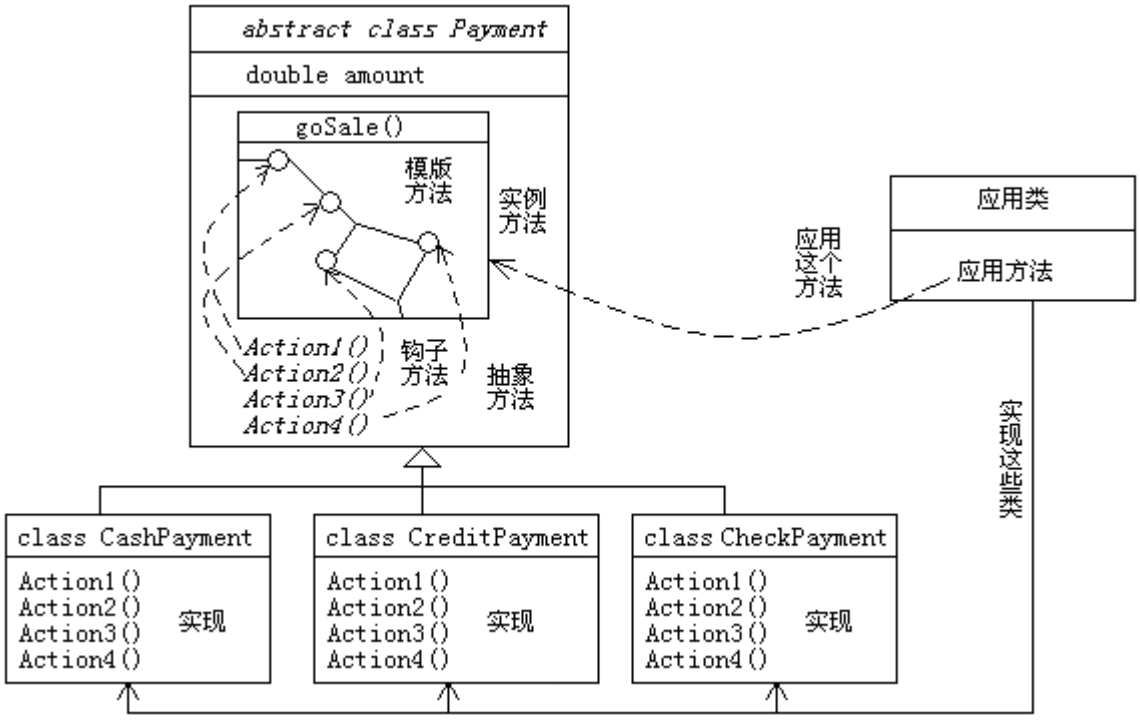
```
public abstract class Payment{
    private double amount;
    public double   getAmount(){
        return amount;
    }
    public void setAmount(double value){
        amount = value;
    }
    public String goSale(){
        String x = "不变的流程一   ";
        x += Action();      //可变的流程
        x += 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、简单工厂模式（Simpleness Factory）

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

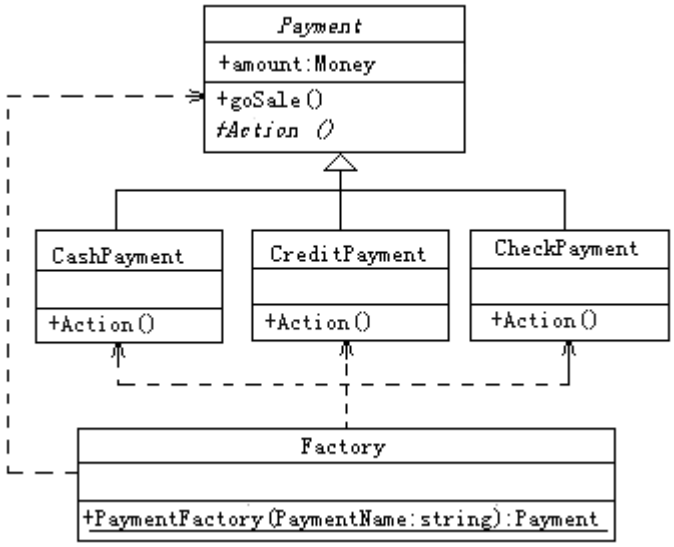
简单工厂实例化的类具有相同的接口，类的个数有限而且基本上不需要扩展的时候，可以使用简单工厂。

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(){
```

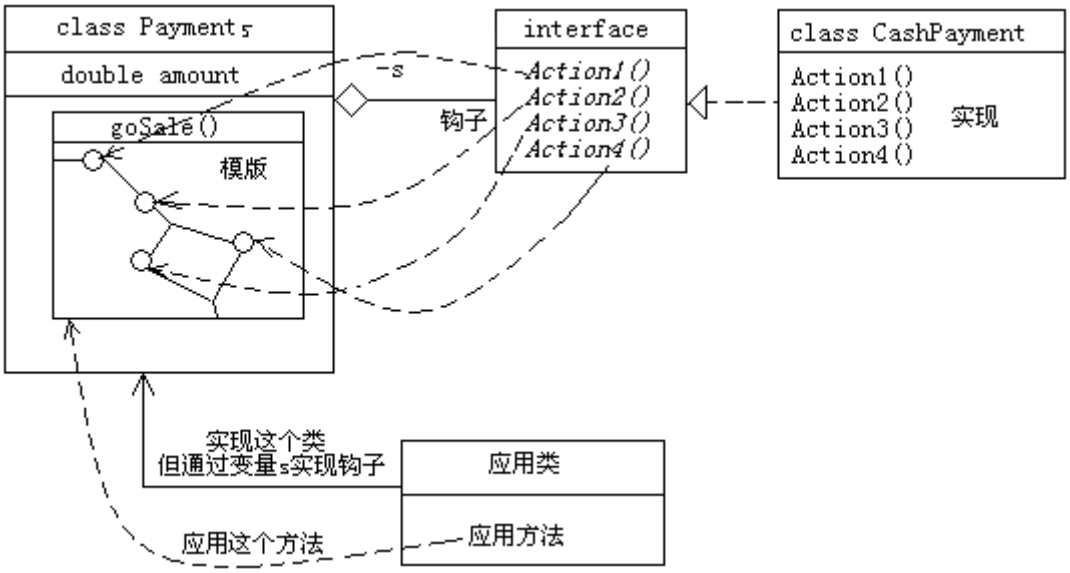
```
        String x = "不变的流程一   ";
        x += Action();      //可变的流程
        x += amount + ", 正在查询库存状态";   //属性和不变的流程二
        return x;
    }
    public abstract String Action();
}

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、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。

我们还是通过上面的关于支付的简单例子可以说明它的原理。



显然，它具备和模版方法相类似的能力，但是不采用继承。

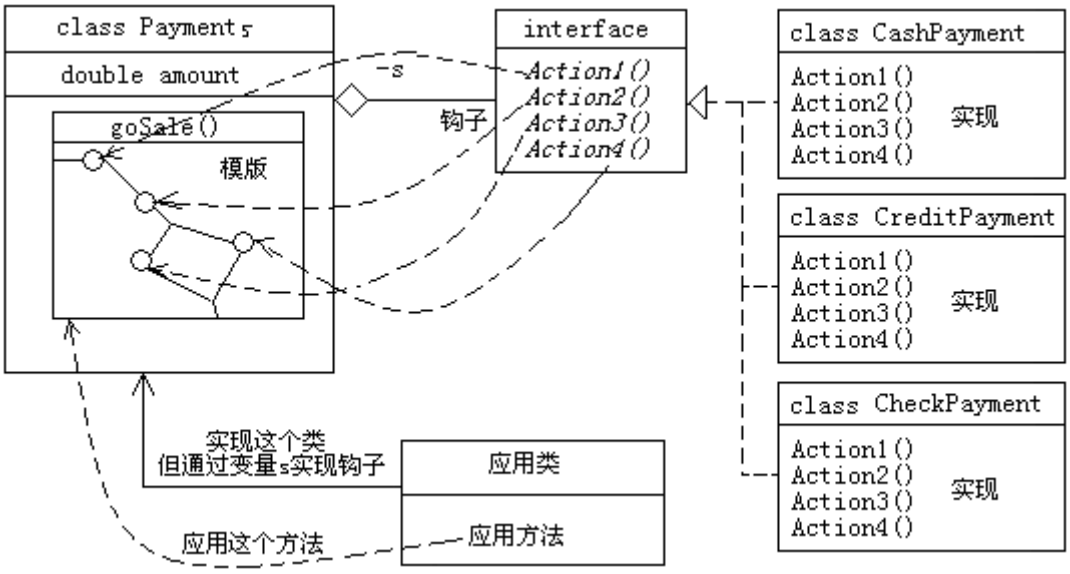
```
public class Payment{
   private double amount;
   public double  getAmount(){
      return amount;
   }
   public void setAmount(double value){
      amount = value;
   }
   private Implementor imp;
   public void setImp(Implementor s){
        imp=s;
   }
   public String goSale(){
      String 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，您将怎样处理呢？




```
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、装饰器模式（Decorator）

有的时候，希望实现一个基本的核心代码快，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。

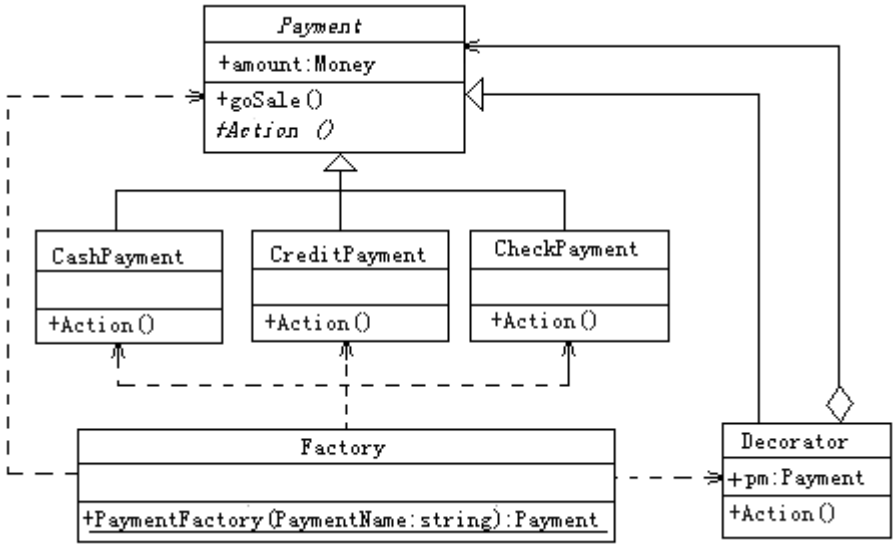
1、意图

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。

5.5 利用观察者模式处理业务单元的变化

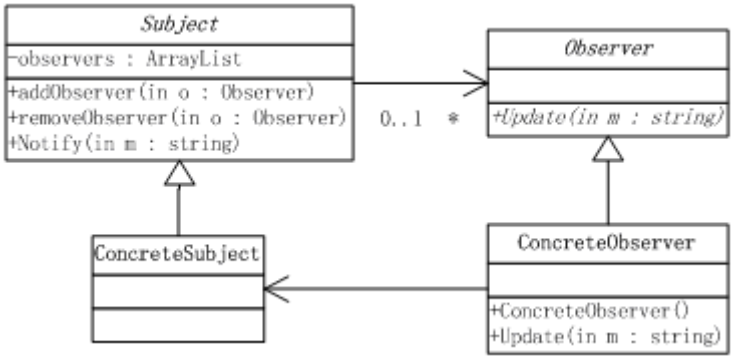
当需要上层对底层的操作的时候，可以使用观察者模式实现向上协作。也就是上层响应底层的事件，但这个事件的执行代码由上层提供。

1、意图：

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

2、结构

传统的观察者模式结构如下。



3，举例：

```
// 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 WriteTextEvett 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());
  }
}
```

5.6 利用策略与工厂模式实现通用的框架

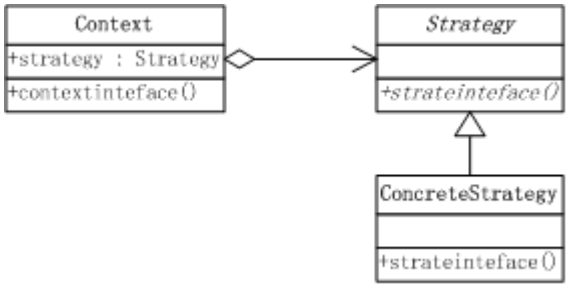
一、应用策略模式提升层的通用性

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、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。

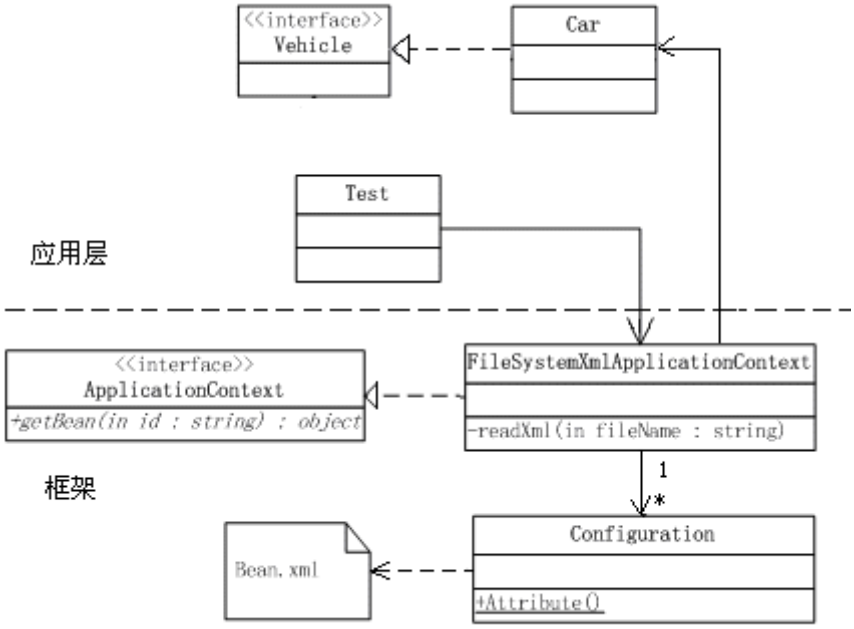
问题：

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{

    //用一个哈西表保留从 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 把数据放入哈西表
           hs=Configuration.Attribute(fileName,"bean","property");
    }

    public Object getBean(String id) throws Exception{
         //由 id 取出内部的哈西表对象
        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"));
     }
}
```

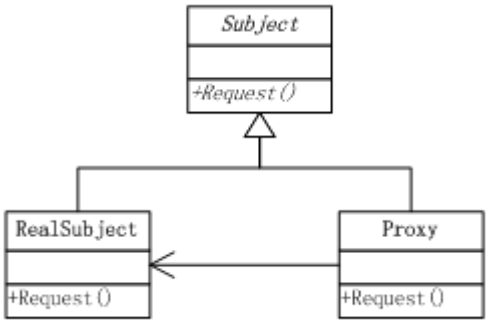
我们发现，在加入新的类的时候，使用方法几乎不变。通过上面的讨论，我们当对框架实现技术有了一个基本的理解。

5.7 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。

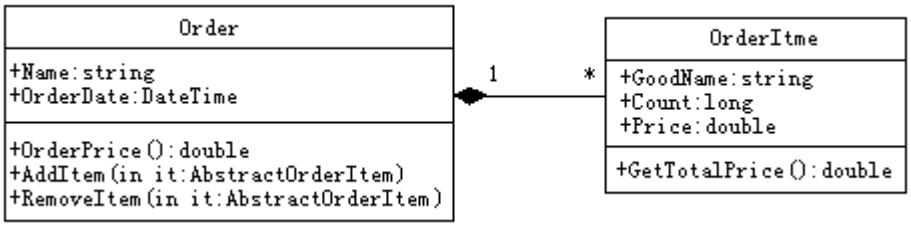


二、在团队并行开发中使用代理模式

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：Ordre 包括若干 OrderItme，订单的总价是每个订单项之和。

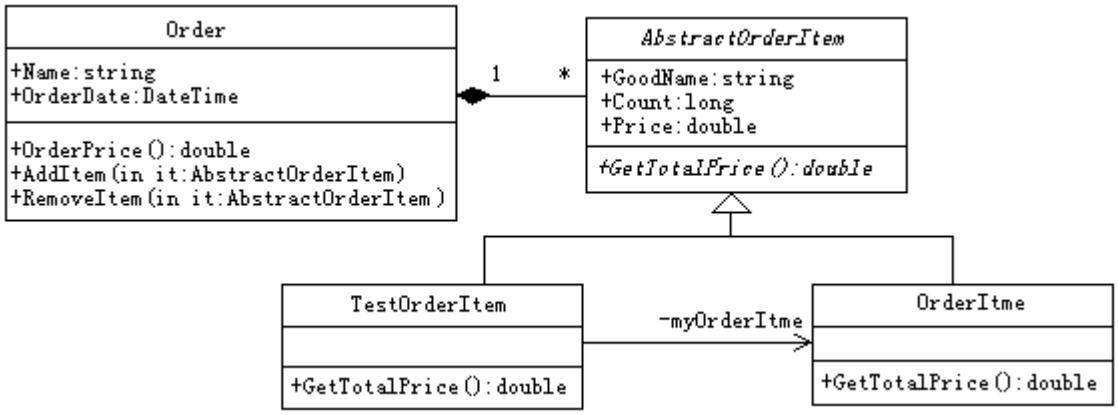
在两个开发组共同完成这个项目的情况下，如果 OrderItme 没有完成，Ordre 也就没有办法测试。一个简单的办法，是 Ordre 开发的时候屏蔽 OrderItme 调用，但这样代码完成的时候需要做大量的垃圾清理工作，显然这是不合适的，我们的问题是，如何把测试代码和实际代码分开，这样更便于测试，而且可以很好的集成。

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

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

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

这些需求牵涉到代理模式的应用，现在可以把代理结构画清楚。



这就很好的解决了问题，实例：

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.getStackTrace();}
    }
}
```

5.8 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。

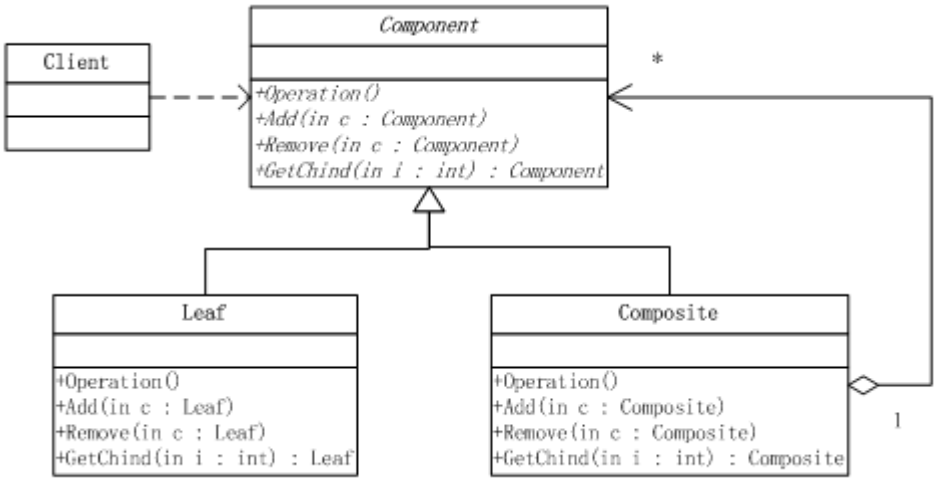
一、树状结构：组合模式

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。

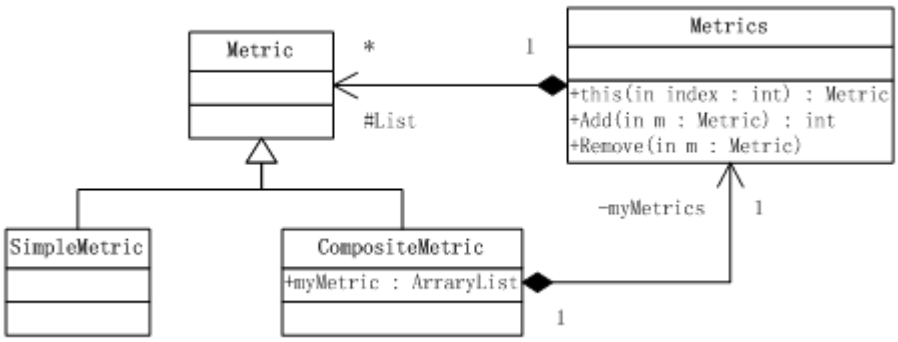
3）可以很容易的增加类型，由于新类型符合相同的接口，因此不需要改动客户代码。

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



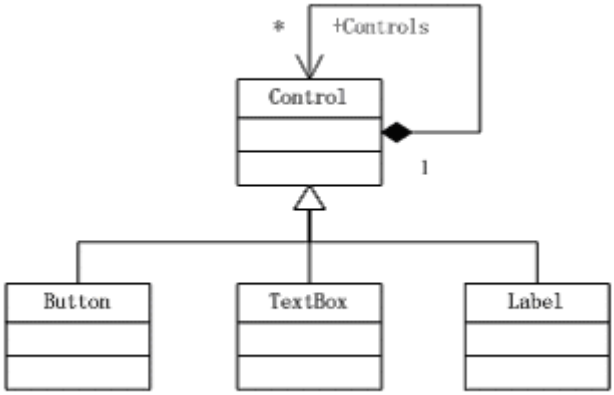
这里有自己定义一个表示控件聚合的 Metrics 对象，由这个对象放置类型（比如 Metric），采用强制类型集合的优点为：

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。

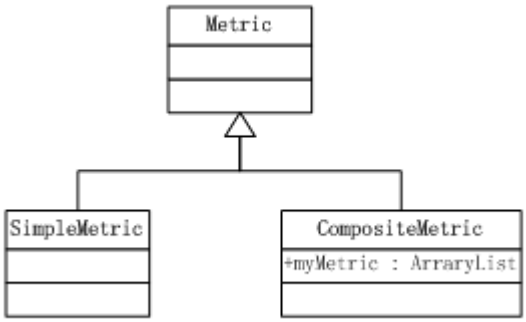
这种方式的缺点是需要自行定制集合，编码比较复杂。

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：

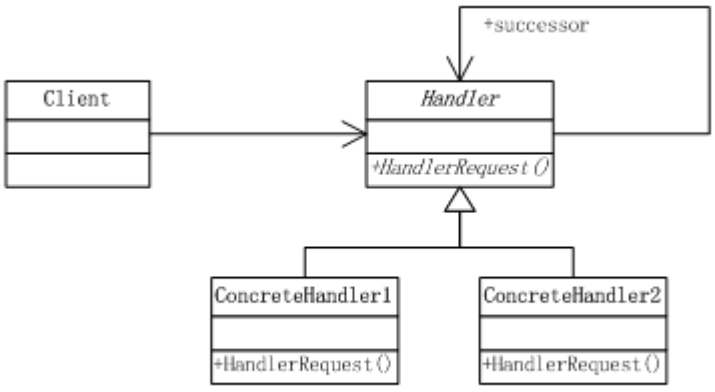
1）有多个对象处理请求，到底怎么处理在运行时确定。

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;
        ct1=m;
        for (int i=1;i<10;i++){
            ct=new ConcreteHandler1();
            ct.s=i;
            ct1.successor=ct;
            ct1=ct;
        }
    }
    private void button1_Click(object sender, System.EventArgs e){
        //显示
        see(m,10);
    }
    void see(Handler m,int b){
        if (m !=null){
            int s=m.HandlerRequest(b);
            listBox1.Items.Add(s);
            m=m.successor;
            see(m,s);
        }
    }
}
```

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。

5.9 基于产品线的架构设计

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。

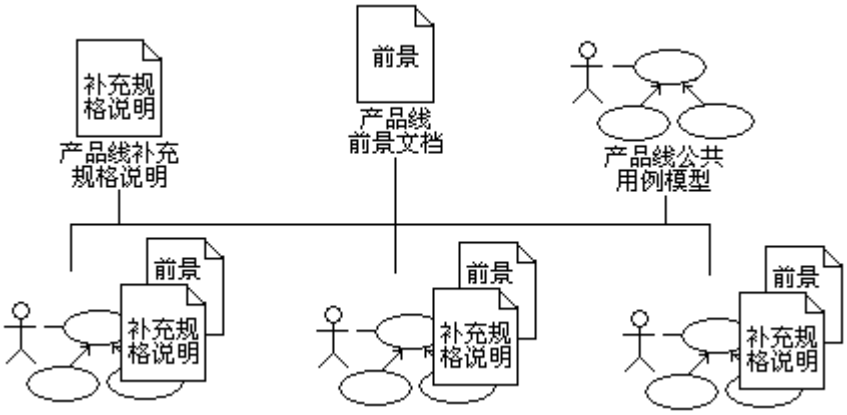
软件产品线依赖于重用，但为什么到今天为止软件重用总是得不到承诺的那么多好处呢？关键是元素集放什么和放多少的问题并没有真正得到解决。软件产品线事实上是确定一个策略，用以对架构进行定义，确定功能，了解质量属性，仔细考虑只有构建的时候需要重用的元素才放到重用库中去。过少则没有意义，过多则难以应用，产品线将依赖战略规划来发挥作用。产品线架构设计的基本步骤如下。

一、组织产品线的需求

许多产业构造了一系列在功能上相近的产品集，但每种产品又包含了某些独特的特性，被称之为产品线。假定，正在构建一套软件系统，每个产品都有共享的功能，在使用过程中需要共享数据，或者与其它部分进行通讯。在这种情况下，可以用如下方法组织需求：

- 开发产品系列的前景文档，描述产品共同的工作方式以及共享的特性。
- 为了更好的理解共享用法的模型，也应该设计一套用例，先是用户如何与共同运行的不同应用自建交互。
- 开发定义关于共享功能的特殊需求的公共软件需求，例如，公共 GUI 和通信协议。
- 为系列中的每个产品开发前景文档、补充规格说明以及定义特殊功能的用例模型。

组织的结果如下图所示。



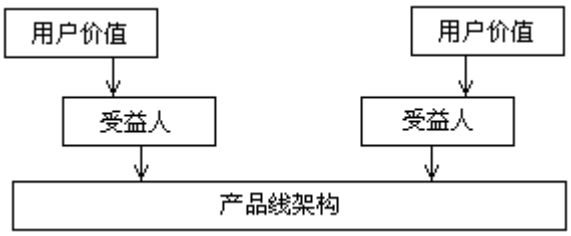
前景文档（Vision Docunment）是在较高的抽象层次定义问题和解决方案，它使用一般的术语描述应用，包括对目标市场、系统用户以及应用特性的描述。前景文档是有效需求过程的关键成分，也可能是最重要的文档。因为一份简短甚至不完全的文档，都有助于项目成员朝向同一个目标工作。团队由此有了共同的目标和剧本，也可能会有共同的心理。如果团队的目标未知而且互相冲突，很快就会产生混乱。

前景文档获取用户需要、系统特性以及项目的其它要求。这样，前景文档的范围横跨需求金字塔的上两层，在较高的抽象级别上定义问题和解决方案。

由于产品可能会以很多方式发生变化，因此在需求分析中需要尽可能获取变化点，这些变化点包括特性、平台、用户接口、质量属性以及目标市场等。除此以外，我们还要考虑如下问题。

1）把价值映射为约束

这个问题的本质，是产品线架构的受益人如何把客户价值与架构约束捆绑。这里所谓受益人，实际上是指使用架构的开发队伍。



2）一致性和灵活性的考虑

一致性的问题：是指受益人使用架构与期望值的符合程度。

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

3）防止产品线架构被拉向不同的方向

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

- 清楚、简明的、阐述一条迫切的用户价值。
- 把用户价值映射为少数的能解决的问题。
- 把以上问题转译为一组最小的约束条件。

遵循准则：

- 各方面的一致性。
- 实施人员信任并使用架构。

- 架构潜藏的知识对用户是可识别和可获得的。

二、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。

三、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。

5.10 产品线架构的案例

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，下面我们以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。

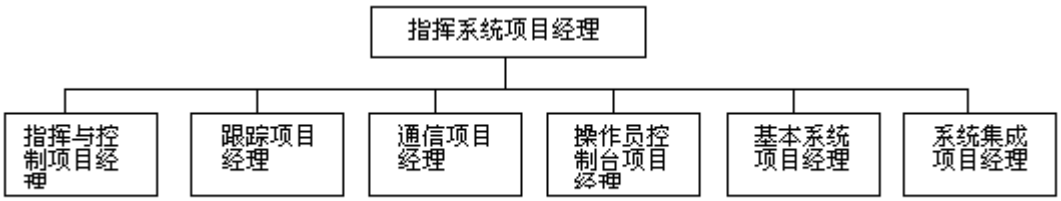
一、开发产品线的动因

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

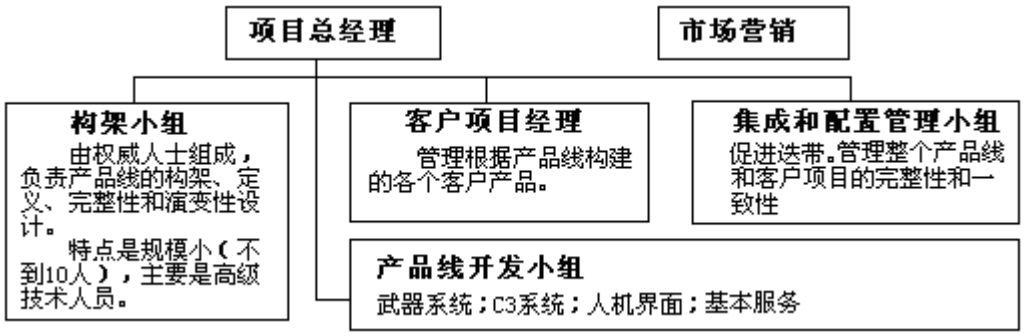
开发方式和理念的改变，必须以组织结构的改变为基础。该公司原来的项目开发组织系统，是以系统功能结构为基础的，以功能领域为基础的项目经理，自开始到系统集成都对整个阶段的人力资源有使用权，基本上如下图：



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、小规模的、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，主要注意力放在了这个架构小组，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：

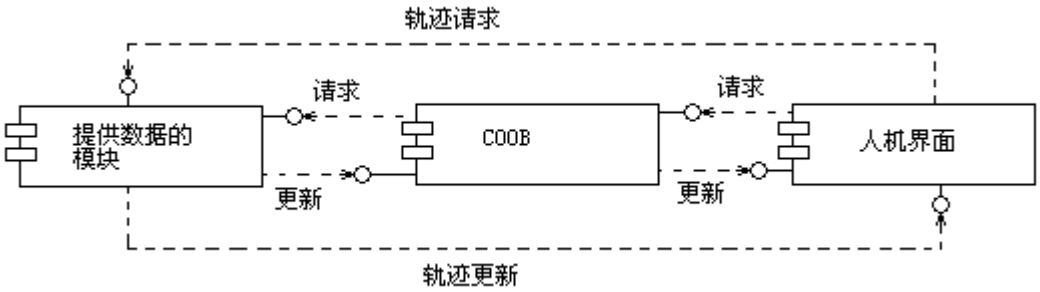
- 性能：系统必须能够对不断到达的传感器输入做响应，并且能控制所要求的分系统。
- 可修改性：对于计算平台、操作系统、添加和更换传感器和分系统、人机接口需求和通信协议的变化，架构都需要具有健壮性。系统应该能不破坏架构和其它部分的情况下，把现有模块更换成针对某个特定系统的模块。
- 安全性、可靠性和可用性。
- 可测试性：每个系统都必须是可集成和可测试的，以快速发现、隔离和纠正错误。

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：

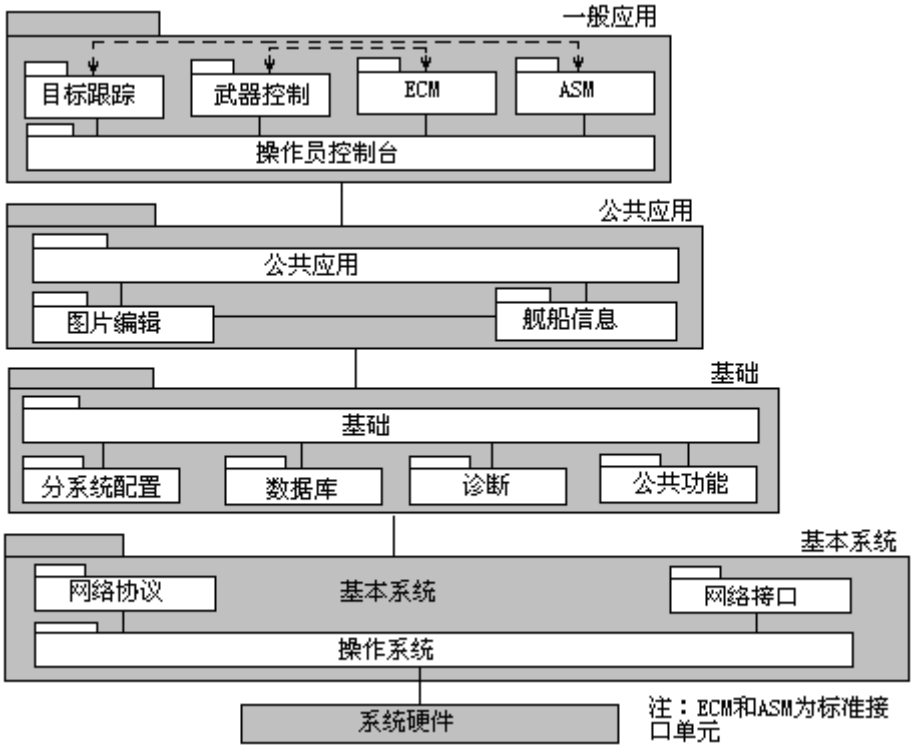
- 从设计上保证了网络负载比较小，这需要设计的时候对数据流作大量的工作，确保所传递的数据都是必要的。
- 数据通道需要具有一定的防错性，尽可能在应用程序内部解决出现的错误。
- 允许应用程序偶尔错过某次更新，例如舰船位置，某次方位数据可能会错过，可以事后根据前后数据进行更新，也就是需要有某种数据“记忆”功能。

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的时候，只改变这一个层的内容。比如，关于网络、通信协议发生变化的时候，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。	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 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，当未来发生变化的时候，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，也要注意到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。

第六章 业务流程敏捷性与面向服务的架构

在大粒度设计的时候，经常遇到的情况是业务单元不变而业务流程可能发生变化，面向服务架构的本质，是实现业务敏捷。这是一场软件开发和架构的革命。

在经典软件工程理论中，不管是瀑布方法还是迭代方法，都是从需求分析做起，一步一步构建起形形色色的软件系统。但是，需求变更像一个挥之不去的阴影，时刻伴随着系统左右。每一个实际应用系统的开发者都饱尝了在系统进入开发阶段、测试阶段，甚至上线阶段遭遇应接不暇的需求变更的极端痛苦。

复杂性和易变性是信息技术（IT）必须面对的现实，无论是构建新的应用、替换现有的应用，以及及时处理各种维护与改进的要求，都是处理各种复杂情况，这种对于复杂性和易变性的应对，是对软件架构师的巨大挑战。

上一章从封装变化的角度研究软件复用的问题，我们主要考虑业务流程具有相似性，而业务单元可能发生变更的情况。但是在大粒度设计的手，经常遇到的情况是业务单元具有相似性，而业务流程将会发生改变的情况，这种情况下应该采取什么策略呢？ 最初的想法实际上还是比较简单的，也就是把业务单元做成基础结构，再增加一个层实现这些单元的编排，当业务流程发生变化的时候，只要改变流程编排就可以了。如果在同一个平台上，这种实现还是比较方便的。但是现实的问题是，在软件系统发展的历程中，业务单元可能不是同一种语言、不属于同一种平台甚至不在同一个物理位置。要把这些历史上形成的业务单元整合在一起，就需要新的标准和方法。如何解决这一问题呢？能否来一场软件开发和架构的革命？SOA 架构的提出，就是被人看成这样的一场革命。其实质就是要将系统业务流程模型与系统实现单元分割开来。

在过去 30 年中，人们为了解决 IT 复杂性和易变性问题作了相当大的努力，人们终于认识到，如果所有的应用都使用公共的编程接口，以及如果建立恰当的互操作协议的话，则有助于降低 IT 的复杂性，提升系统应对变更的能力，已有的功能也更容易的被再次利用，这就是功能或者业务层面的复用。

在软件复用向上游发展的进程中，到今天人们已经开始把大量的精力放到考虑业务级复用的问题，面向服务的开发（Service-Oriented Development）所承诺带给 IT 界的，就是这种功能级别或者业务级别的复用，而如果部署采用面向服务的架构 (Service-Oriented Architecture SOA)，将更有利用建立各种新的战略方案，这些方案包括：

- 快速应用集成；
- 自动化业务流程；
- 支持多渠道服务的应用（包括固定设备和移动设备）。

面向服务的架构是一种形式化的方式分离服务的架构风格，事实上，SOA 是在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，使用一种服务契约（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。

6.1 面向服务的架构的本质

一、业务流程的敏捷性需求带来的挑战

SOA 的本质一句话就是实现业务敏捷（Busines Agility），因此，首先让我们看一下什么是业务敏捷。企业发展来自于市场的不断开拓、战略不断调整、经营方针不断的完善、管理水平不断的提高。在今天，要实现上述所有的工作，都必须依靠 IT 系统的有效支持，因为依照传统的方式，已经没有办法满足企业的发展与竞争的需求了。

敏捷性包括几个重要的因素：变化、变化的速度和变化的质量。业务的敏捷性让一个企业能够适时、快速的响应变化，从而有可能采取恰当、明智的应变措施，这是今天竞争激烈，变化越来越快的商业环境所决定的。如何灵活快速适应变化甚至创新求变，成为企业生存的头等大事。

SOA 要解决的根本问题是，如何保护企业的现有资产，推动业务敏捷，换句话说就是重用，把企业已经有的应用系统资产，用标准的、高效的和便利的方式集成起来，使企业更好的应对市场的变化，对业务需求的变化作出快速的反应。

业务敏捷性意味着快速的决定和行动，这受到很多因素的影响，除了掌握信息、分析论证、对政治气候的理解等等因素外，还要看到目前企业的现状是一大群人和一大堆信息系统条块分割的各自完成各自的业务，计算机系统相互隔离，不能协调工作。这样一来，整体上会处于一种混乱状态，很难达成业务的敏捷性。

我们应该看到，业务敏捷性取决于企业信息的自由流动、服务和业务流程，而这些都要求 IT 系统能够满足业务的变更，决不能因为业务变更造成诸如仓储系统重新构建、人力资源系统重新编写、财务系统重新构建这样一类基本上做不到的事情。所以，为了应对业务敏捷的需求，IT 系统必须应对两个基本的挑战：

1，能够统一描述各种业务、业务对象与业务模型，这些业务对象和业务模型需要很容易被组合或者重组。

2，对于大企业的平台异构性所带来的复杂性要有充分的理解，目前企业系统一般是多平台（IBM、BEA、Microsoft、Oracle）和多技术（J2EE、.NET、遗留技术）构成的，而且业务会涉及到企业内部、外部环境、供应商和客户等，因此就需要更好的互联技术来满足异构系统之间的信息交互。

解决这类问题，给软件架构的思想、方法和技术带来了前所未有的挑战。SOA 的概念，就是在这种大背景下提出来的，这种概念引入的诸如粗粒度、松耦合、接口以及业务组件已经成了软件架构人员耳熟能详的概念，但这些概念如何为企业创造价值呢？

二、SOA 一些概念的澄清

1，什么是服务

一般来说，服务包含 4 个主要方面：提供、使用、说明、中介。我们先从一个例子来讨论什么是服务，银行出纳员要为银行客户提供服务，不同的出纳员可能要提供不同的服务，典型的银行服务包括：

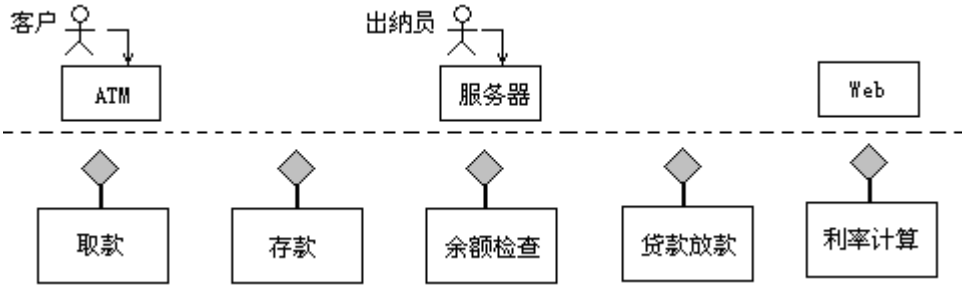
- 账户管理（开户与消户）；
- 贷款（处理申请、查询条款与细则、同意放款）；
- 提款、存款与转账；
- 外汇兑换。

尽管存在多个提供一组相同服务的出纳员，不过对于客户而言，他们只关心能否完成服务，至于银行怎么安排则与他无关，如果是一个复杂的交易，客户可能需要与多个出纳员接触才能完成处理，这就是一个业务流程流。

实现银行业务自动化的 IT 系统存在于银行内部，但服务是通过出纳员最终提供给客户的，因此，IT 系统所实现的服务，必须与出纳员向客户提供的服务一致，并且必须对出纳员提供支持。如果“IT 系统提供服务的定义”与“业务功能与业务流程”一致，那么 IT 系统

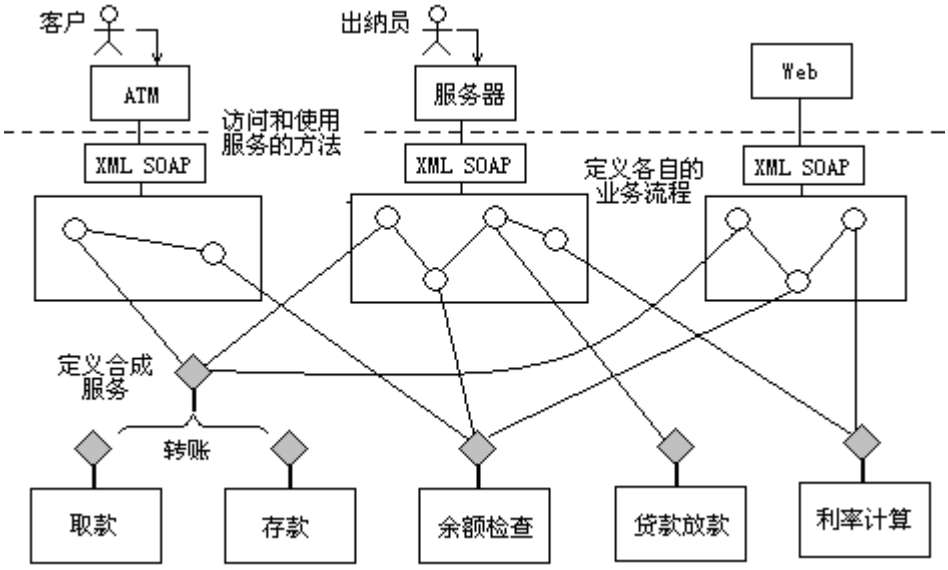
就更容易支持业务目标。

例如 ATM 机、办公网络上的出纳员、以及 Web 用户需要使用相同的服务。虚线以下是基础结构，而虚线以上是三种业务对象，其中：ATM 机业务包括转账和余额检查。出纳员业务包括转账、余额检查、贷款放款、利率计算。Web 包括转账、余额检查、利率计算。



这些业务单元实际上是不变的，但业务流程在不同的应用中可能发生改变。而且我们需要达成的软件系统服务的实现环境并不重要，重要的是服务本身。

首先我们需要定义原子服务，图中每一个基本的业务单元都可以看成一个原子服务。然后定义合成服务（例如把存款与取款服务组合成一个转账服务）。接下来定义各自的业务流程，这就是实现业务流程的重新编排。最后构建服务消费者访问和使用服务的方法



这样一来，就搭建了一个典型的面向服务的架构，这样的服务部署满足了不同客户的需要。

软件服务的定义与银行提供的业务服务相一致，目的是确保业务运营的流畅，并有助于实现战略目标（比如除了柜台方式以外，还允许通过 ATM 和 Web 方式使用银行服务）。复杂的服务可以由多个服务组合而成。在 SOA 环境中部署服务，可以使服务的合成更加容易，而这种合成的应用，也可以发布为服务，供人或者 IT 系统使用。

从 IT 的角度来看，服务就是机器可读的消息（接收和返回）描述的网络位置，也就是服务是由它所支持的消息交换模式定义的，消息中包含着数据具有的相应模式（schema），模式用于在服务请求者和服务提供者之间建立契约（contract）。其它的一些原数据项分别描述了服务的网络地址、所支持的操作、以及对可靠性、安全性以及事务性方面的要求。

2，SOA 的定义

在构建 IT 架构（特别是企业级架构）时，我们的目标始终是：支持业务流程并对业务变化做出响应。在最近几年中，出现了一些构建系统架构的新方法，这些方法主要围绕功能

单元（称为服务）来构建复杂的系统。

Web 服务也对以上这几个方面提供基于系统和标准的支持。因此，Web 服务具有无与伦比的敏捷性这一优点。例如，使用 Web 服务基础设施可以在运行时更改服务提供者，而不影响使用者。某个系统本身要被称为基于 SOA 的系统，应具备以下特性：

- 业务流程映射到软件服务；因此，业务可通过软件进行跟踪。
- 存在一种基础结构，支持上述服务的 4 个不同方面。这样服务级别就具有高度的敏捷性。
- 服务是监视和管理单元。因此，一个人可以跟踪业务流程的操作属性和问题。

SOA 并不是一个新概念，有人就将 CORBA 和 DCOM 等组件模型看成 SOA 架构的前身。早在 1996 年，Gartner Group 就已经提出了 SOA 的预言。不过那个时候仅仅是一个“预言”，当时的软件发展水平和信息化程度还不足以支撑这样的概念走进实质性应用阶段。到了近一两年，SOA 的技术实现手段渐渐成熟了。在 IBM、BEA、HP 等软件巨头的极力推动下，才得以慢慢风行起来。Gartner 为 SOA 描述的愿景目标是实现实时企业（Real-Time Enterprise）。

关于 SOA，目前尚未有一个统一的、业界广泛接受的定义。一般认为：面向服务的架构是一个组件模型（SOA），它将应用程序的不同功能单元——服务（service），通过服务间定义良好的接口和契约（contract）联系起来。接口采用中立的方式定义，独立于具体实现服务的硬件平台、操作系统和编程语言，使得构建在这样的系统中的服务可以使用统一和标准的方式进行通信。这种具有中立接口的定义（没有强制绑定到特定的实现上）的特征被称为服务之间的松耦合。

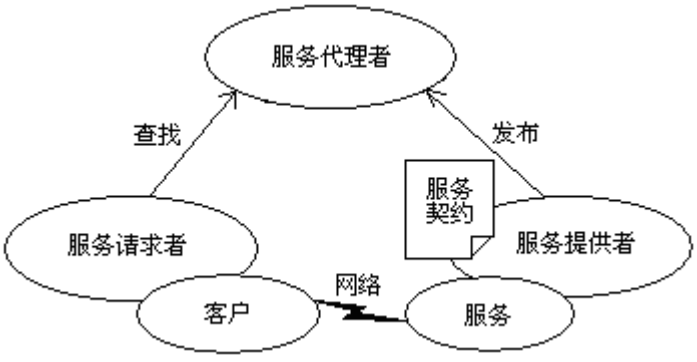
从这个定义中，我们看到下面两点：

- 它是一种软件系统架构。 SOA 不是一种语言，也不是一种具体的技术，更不是一种产品，而是一种软件系统架构。它尝试给出在特定环境下推荐采用的一种架构，从这个角度上来说，它其实更像一种架构模式(Pattern)，是一种理念架构，是人们面向应用服务的解决方案框架。
- 服务（service）是整个 SOA 实现的核心。SOA 架构的基本元素是服务，SOA 指定一组实体（服务提供者、服务消费者、服务注册表、服务条款、服务代理和服务契约），这些实体详细说明了如何提供和消费服务。遵循 SOA 观点的系统必须要有服务，这些服务是可互操作的、独立的、模块化的、位置明确的、松耦合的，并且可以通过网络查找其地址。

正是从这个观点出发，SOA 设计主要是从方法论层面考虑问题，重点是考虑如何把业务流程映射到软件服务，以及业务如何规划，以达到足够而且恰当的业务敏捷。当然，这种映射也需要通过一定的技术手段来完成，但是技术手段是多种多样的。

3，三种角色的关系

下图是 W3C 给出的 SOA 模型中三种不同角色的关系示意图。其中：



服务是一个自包含的、无状态（stateless）的实体，可以由多个组件组成。它通过事先定

义的界面响应服务请求。它也可以执行诸如编辑和处理事务（transaction）等离散性任务。服务本身并不依赖于其他函数和过程的状态。用什么技术实现服务，并不在其定义中加以限制。

服务提供者（service provider）：也称之为服务生产者，它主要提供符合契约（contract）的服务，并将它们发布到服务代理。

服务请求者（service consumer）：也叫服务使用者、消费者，它发现并调用其他的软件服务来提供商业解决方案。从概念上来说，SOA 本质上是将网络、传输协议和安全细节留给特定的实现来处理。服务请求者通常称为客户端，但是，也可以是终端用户应用程序或别的服务。

服务代理者（service broker）：作为储存库、电话黄页或票据交换所，产生由服务提供者发布的软件接口。

这三种 SOA 参与者：服务提供者、服务代理者以及服务请求者通过 3 个基本操作：发布（publish）、查找（find）、绑定（bind）相互作用。服务提供者向服务代理者发布服务。服务请求者通过服务代理者查找所需的服务，并绑定到这些服务上。服务提供者和服务请求者之间可以交互。

所谓服务的无状态，是指服务不依赖于任何事先设定的条件，是状态无关的（state-free）。在 SOA 架构中，一个服务不会依赖于其他服务的状态。 它们从客户端接受服务请求。因为服务是无状态的，它们可以被编排(orchestrated)和序列化（sequenced）成多个序列 (有时还采用流水线机制) ，以执行商业逻辑。编排指的是序列化服务并提供数据处理逻辑。但不包括数据的展现功能。

3，SOA 的特征

基于上面的讨论，我们给出 SOA 的下面一些特征：

1）服务的封装（encapsulation）：将服务封装成用于业务流程的可重用组件的应用程序函数。它提供信息或简化业务数据从一个有效的、一致的状态向另一个状态的转变。封装隐藏了复杂性。服务的 API 保持不变，使得用户远离具体实施上的变更。

2）服务的重用（reuse）：服务的可重用性设计显著地降低了成本。为了实现可重用性，服务只工作在特定处理过程的上下文（context）中，独立于底层实现和客户需求的变更。

3）服务的互操作(interoperability)：互操作并不是一个新概念。在 CORBA、DCOM、web service 中就已经采用互操作技术。在 SOA 中，通过服务之间既定的通信协议进行互操作。主要有同步和异步两种通信机制。SOA 提供服务的互操作特性更有利于其在多种场合被重用。

4）服务是自治的（Autonomous）功能实体：服务是由组件组成的组合模块，是自包含和模块化的。

SOA 非常强调架构中提供服务的功能实体的完全独立自主的能力。传统的组件技术，如.NET Remoting、EJB、COM 或者 CORBA，都需要有一个宿主(Host 或者 Server)来存放和管理这些功能实体；当这些宿主运行结束时，这些组件的寿命也随之结束。这样当宿主本身或者其他功能部分出现问题的时候，在该宿主上运行的其他应用服务就会受到影响。

SOA 架构中非常强调实体自我管理和恢复能力。常见的用来进行自我恢复的技术，比如事务处理(Transaction)、消息队列(Message Queue)冗余部署(Redundant Deployment)和集群系统(Cluster)在 SOA 中都起到至关重要的作用。

5）服务之间的松耦合度(Loosly Coupled)：服务请求者到服务提供者的绑定与服务之间应该是松耦合的。这就意味着，服务请求者不知道提供者实现的技术细节，比如程序设计语言、部署平台，等等。服务请求者往往通过消息调用操作，请求消息和响应，而不是通过使用 API 和文件格式。

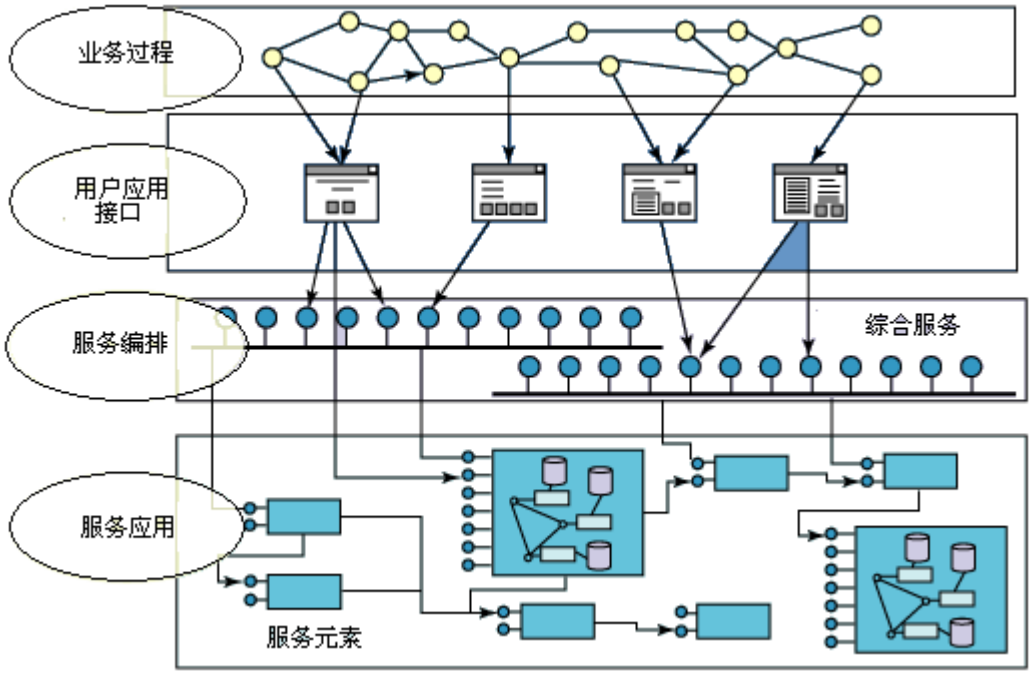
这个松耦合使会话一端的软件可以在不影响另一端的情况下发生改变，前提是消息模式

保持不变。在一个极端的情况下，服务提供者可以将以前基于遗留代码（如 COBOL）的实现完全用基于 Java 语言的新代码取代，同时又不对服务请求者造成任何影响。这种情况是真实的，只要新代码支持相同的通信协议。

6）服务是位置透明的(location transparency)：服务是针对业务需求设计的。需要反映需求的变化，即所谓敏捷（agility）设计。要想真正实现业务与服务的分离，就必须使得服务的设计和部署对用户来说是完全透明的。也就是说，用户完全不必知道响应自己需求的服务的位置，甚至不必知道具体是哪个服务参与了响应。

4，SOA 应用程序

下图从应用程序角度展示了企业级 SOA 所包含的元素。业务流程由用户界面应用程序和服务应用程序进行部分和完全支持。业务流程中的一个步骤或者通过人工执行，或者得到用户界面应用程序的支持。用户界面应用程序实现了许多宏工作流，而且它们还使用实现业务功能的服务。



在服务编排层，组合服务是通过编排语言（例如业务流程执行语言（Business Process Execution Language，BPEL））定义的。组合服务的编排通过基本服务 定义其流程和组成。编排层应由支持图形规范的编排工具提供支持，例如 IBM WebSphere® Business Integration Modeler 和 IBM Rational® Application Developer。

基本服务（由服务编排层使用，也由用户界面应用程序使用）通过服务应用程序实现。而服务实现又可以调用其他服务，这些服务通常来自另外的服务应用程序。

Jason Bloomberg 在其《Principles of SOA》中指出，SOA 的实践必须遵循以下原则：

- **业务驱动服务，服务驱动技术**：从本质上说，在抽象层次上，服务位于业务和技术中间。面向服务的架构设计师一方面必须理解在业务需求和可以提供的服务之间的动态关系;另一方面，同样要理解服务与提供这些服务的底层技术之间的关系。
- **业务敏捷是基本的业务需求**：SOA 考虑的是下一个抽象层次：提供响应变化需求的能力是新的“元需求”，而不是处理一些业务上的固定不变的需求。从硬件系统以上的整个架构都必须满足业务敏捷的需求，因为，在 SOA 中任何的瓶颈都会影响到整个 IT 环境的灵活性。
- **一个成功的 SOA 总在变化之中**：SOA 工作的场景，更像是一个活的生物体，而不

是像传统所说的“盖一栋房子”。IT 环境惟一不变的就是变化，因此面向服务架构设计师的工作永远不会结束。对于习惯于盖房子的设计师来说，要转向设计一个活的生物体要求有崭新的思维方式。SOA 的基础还是一些类似的架构准则

5，SOA 常见误区

在 SOA 领域中有一些常见误区，现说明如下：

1）认为 SOA=Web Service：

Web 服务通常指的是基于 SOAP/HTTP 的一种服务，这些服务通常是实践 SOA 所定义服务的一种技术形式，它提供了分布式环境下卓越的互操作能力，但实现 SOA 的方法还很多，并不仅仅是 Web 服务一种。SOA 构架是独立于技术实现的。SOA 并不必用 Web Services 来实现，相反，Web Services 也并不一定遵循 SOA 标准。

不过，Web Services 的特性十分适合用来实现 SOA 架构。Web Services 之间能够交换带结构的文档（比如 XML），这些文档可能包含完全异构的数据信息。这些文档可以同时附带关于数据的数据：元数据（metadata）。换句话说，Web Services 可以有较粗的粒度，这样较粗的粒度正好可以构成 SOA 中服务的粒度。

说到底，两者是相交的圆，SOA 服务和 Web Services 之间的区别还在于设计。SOA 概念并没有确切地定义服务具体如何交互，而仅仅定义了服务如何相互理解。其中的区别也就是定义如何执行流程的战略与如何执行流程的战术之间的区别。而另一方面，Web Services 在需要交互的服务之间如何传递消息有具体的指导原则;从战术上实现 SOA 模型是通过 HTTP 传递的 SOAP 消息中最常见的 SOA 模型。因而，从本质上讲，Web Services 是实现 SOA 的具体方式之一。

2）认为 SOA 是一种特殊的分布式组件对象（Components Objects）：

SOA 中的服务与组件对象的相似之处在于：都有一个或多个接口，并且，服务发布者和使用者都遵守这些接口。

不同之处在于：SOA 是关于模式（schemas）的，组件对象是关于对象类型（object types）的；SOA 通过像 SOAP 这样的标准消息机制（messages）来实现通信，而组件对象通过方法调用（method calls）来交互。与 CORBA 中的接口定义语言 IDL (Interface Definition Language) 相比，SOA 在 WSDL (Web Services Definition Language) 中采用 XML，会显得更加普遍和通用。

联系之处在于：服务最终还是通过类和组件对象来实现的。

SOA 被认为是传统紧耦合的、面向对象的模型的替代者。像通用对象代理架构 CORBA (Common Object Request Broker Architecture)和分布式组件对象模型 DCOM (Distributed Component Object Model)。在 SOA 中，单个服务可以用面向对象方法来设计，但是，整个 SOA 的设计却是面向服务的。下面的表格中给出了 SOA 与分布式组件架构的不同点。

SOA 与分布式组件架构的区别		
编号	**分布式组件架构**	**面向服务的架构**
1	面向功能	面向流程
2	设计目的是实现需求	设计目的是适应变化
3	开发周期长	交互式和重用性开发
4	成本为中心	业务为中心
5	应用阻塞	服务协调
6	紧密耦合	敏捷的和松耦合
7	同构技术	异构技术
8	面向对象	面向消息
9	需要更深入的了解实施细节	独立与实施细节

3）认为新系统 SOA 没有用武之地：对于新构建的系统，如果需要得到业务敏捷性的话，这就使 SOA 具有了用武之地。SOA 通过更好地让 IT 和业务融合在一起，借助于企业架构、

业务建模、SOA 监管以及一些新的设计原则，使支持这种风格的新技术来达成 IT 的灵活性，可以更好的支持业务敏捷性。

4）认为 SOA=BPM：业务流程管理（Business Process Management，BPM）与 SOA 的关系紧密，但并不是一件事。BPM 的目的是业务优化，这种优化需要 IT 支持，SOA 很好的能提供这种支持；反过来 BPM 在业务建模和业务规则方面也能给 SOA 提供很好的支持，为 SOA 达成业务敏捷性带来良好的基础。

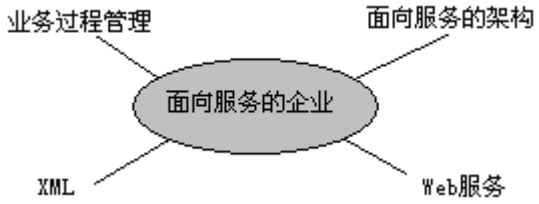
6.2 面向服务的架构所牵涉到的问题

尽管 SOA 的思想是与技术无关的，但是任何思想没有技术实现的支持，那也仅仅是一个幻想。由于 Web 服务的 广泛普及，给 SOA 的应用打下了坚实的基础，使这种架构思想的实现成为可能，其中 SOAP 已经成为连接不同应用的标准互操作协议，WSDL（Web Service Description Language）已经成为访问任何应用的标准互操作协议。这样一来，使 Web Service 成为构建下一代 IT 基础设施 SOA 的公认比较优秀的平台。也正是这个原因，本课程中在讨论 SOA 有关问题的时候，还是通过 Web 服务的有关标准来具体描述，以避免空泛的描述，使我们的研究落到实处。

但是，我们还是要强调，SOA 架构设计与其说是技术问题，还不如说是战略问题，更需要设计师对业务有更深刻的理解，下面简要介绍 SOA 设计中需要关注的几个问题。

一、面向服务的企业

对 Web 服务的广泛接受，将对面向服务的企业产生巨大的影响，这会显著增加组织的机动性、加快新产品、新服务的上市速度，降低 IT 的成本，同时提高运营效率。多种代表行业趋势的概念与实现发生了重大的变化，他们关注的问题如下图所示，这是共同造就面向服务企业的关键技术。



1，可扩展的标记语言（XML）

这是一种跨企业的（更广泛的）、公共的、中立的数据格式，它提供了：

- 与编程语言、开发环境以及软件系统无关的标准数据类型与结构；
- 对于业务文档定义和业务信息（比如标准的行业词汇）交换的通用工具；
- 已经存在广泛的 XML 处理软件（解析器、查询器、转换器等）。

2，Web 服务（Web Service）

这是一种基于 XML 技术，用于传递消息、描述服务、发现服务以及其它的扩充功能，它提供了：

- 各种被广泛采用的、用于分布式计算的接口描述，以及通过消息进行文档交换的开放式标准；
- 与下层执行技术和应用平台的无关性；
- 企业级的服务质量（安全性、可靠性、事务性等）的可扩展性；

- 对合成应用（业务流程、多渠道服务、快速集成等）的支持。

3，面向服务的架构（SOA）

一套用于实现应用间的互操作，以及重用 IT 资产的方法，它具有以下特征：

- 对架构方面（治理、过程、建模、工具）的强烈关注；
- 具有恰当的抽象层次，有利于促进业务需求与技术能力的配合与协调，以及致力于创建可重用、粗粒度的业务功能；
- 是一种适于快速、方便的构建新应用部署的基础设施；
- 一个用于常见业务与 IT 功能的可重用复用库。

4，业务流程管理（Business Process Management，BPM）

用于自动化业务操作的方法和技术，它包括：

- 清晰地描述业务流程，以便于理解、改进和优化；
- 易于针对业务需求的变更，快速修改业务流程；
- 把原来由人工完成的业务流程自动化，并实施相应的业务规则；
- 为决策者提供有关业务流程的实时信息与分析。

这些技术都已经对业务计算的一个或者多个方面产生了深远影响，现在要做的就是把它们融合起来，就可以提供一个综合平台，这个平台有助于获得面向服务的优点，并且在 IT 系统的发展进程中走向下一个阶段。

二、面向服务的开发

目前，软件厂商已经广泛接受了“采用 Web 服务进行面向服务开发”这么一种模式。面向服务开发是对前面已经讨论过的面向过程、面向对象、面向方面等开发方法的补充。它具有以下优点：

- **重用**：创建可重用用各种业务应用的服务能力；
- **效率**：通过组合现有服务，以快速创建新的服务和业务应用的能力，这样就可以集中精力于数据共享，而不是底层实现的能力；
- **与技术的松耦合**：独立于服务的执行环境进行服务建模的能力。例如，紧紧盯以服务能够收发的消息，而不需要考虑具体的技术实现。
- **职责的划分**：可以令业务人员和技术人员分别关注业务问题和技术问题，双方通过服务契约进行协同。

当 SOA 架构师构建一个企业级的 SOA 系统架构的时候，关于系统中最重要的元素，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，由于服务的访问通常是远程的，所以，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。

应用服务来设计、开发和部署应用，需要在思考方式上发生重大转变，为了帮助这一转变的完成，我们可以把 IT 部门的职责划分为两个部分：

- **创建服务**：处理服务所涉及的复杂的下层技术，确保 Web 服务的描述与服务消费者的需要相一致，而且双方共享着应该有的数据。
- **使用服务**：组装新的合成应用（Composite Applications）和业务流程流（Business Process Flows），确保共享数据以及业务流程流能够准确反映业务的运营和战略需求。

在项目层次上，架构师通常要指导可重用服务的开发，并确定一种存放、管理和检索服

务描述的方法。可重用的服务层把业务操作（比如“获取客户信息”、“下订单”）与下层软件平台的实现差异相隔离（就象浏览器把服务器的实现语言的差异相隔离一样），这样，就有可能具备把可重用的服务快速组合成更大服务的能力，这样，组织就具备了使过程自动化和快速适应环境的优点。

事实上，定义可重用的服务是面向服务最重要的方面，要实现服务的最高价值，必须在开发的时候，就考虑与其它服务的互操作，并且通过与其它服务的组合来构建应用。这种思想上的转变，可能需要某个处于领导职位上的人协调完成检查设计，以确保它们与新的 IT 目标一致。

三、SOA 的服务抽象

1，SOA 的三个抽象级别

从概念上讲，SOA 中有三个主要的抽象级别：

- **操作**：代表单个逻辑工作单元（LUW）的事务。执行操作通常会导致读、写或修改一个或多个持久性数据。SOA 操作可以直接与面向对象 (OO) 的方法相比。它们都有特定的结构化接口，并且返回结构化的响应。同方法一样，特定操作的执行可能涉及调用附加的操作。
- **服务**：代表操作的逻辑分组。服务可以分层，以降低耦合度和复杂性。一个服务的粒度（granularity）大小也与系统的性能息息相关。粒度太小，会增加服务间互操作通信的开销；粒度太大，又会影响服务面对需求变化的敏捷性。
- **业务流程**：为实现特定业务目标而执行的一组长期运行的动作或活动。业务流程通常包括多个业务调用。

在 SOA 中，业务流程包括依据一组业务规则按照有序序列执行的一系列操作。操作的排序、选择和执行称为服务或流程编排。典型的情况是调用已编排服务来响应业务事件。从建模的观点来看，由此带来的挑战是如何描述设计良好的操作、服务和流程抽象的特征，以及如何系统地构造它们。这些涉及服务建模、特征抽取的问题已经成为现阶段人们关注的焦点。

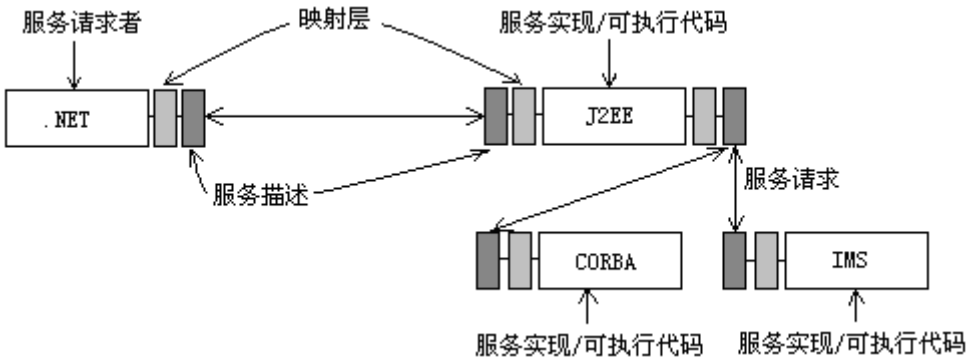
2，服务抽象

一个可复用的服务包括服务的描述、服务的实现以及服务的映射等。

服务的实现（service implementation）：任何提供 Web 服务支持的执行环境都可以认为是服务的实现，服务的实现也称之为可执行代理（executable agent），它负责实现服务处理模型，在执行环境中运行，这里的可执行环境通常是某种编程语言。

服务的描述（service description）：它通常与可执行代理是分开的，一个服务描述可以由多个不同的代理与之相关联，而一个可执行代理也可以支持多个服务描述。

映射层（maping layer）：有时也称之为转换层，实现与执行环境的分离，映射层通常以代理（proxy）或者桩（stub）的形式出现，映射层负责接收消息、转换 XML 到本地格式、把数据派送到可执行代理。

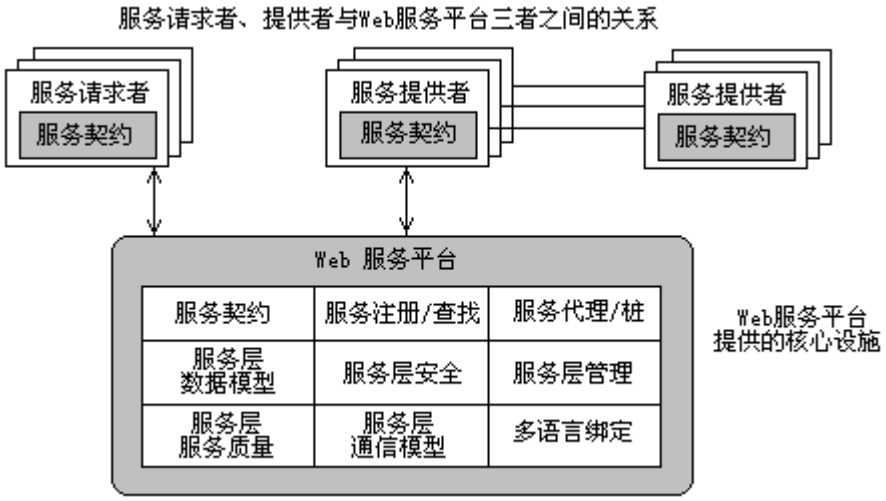


我们已经讨论过，Web 服务区分请求者和提供这两种角色：

服务请求者：通过向服务器发送一条消息来启动服务的执行。

服务提供者：收到消息以后执行服务，然后把结果返回给服务请求者。

服务抽象的好处在于，易于访问不同类型的服务，比如新开发的服务、经过包装的传统应用，或者由其它服务合成的应用等。



有时候服务提供者也可能是服务请求者，也就是说，服务请求者与服务提供者可以被组织成一种 N 层结构。

四、解读 SOAP 和 WSDL

Web 服务中两个关键性的标准是 SOAP 与 WSDL（Web Service Description Language），SOAP 协议是一个标准的通信协议，已经为软件业所接受。

1，解读 SOAP

让我们讨论一个例子，服务程序为 C#代码：

```
using System.Web;
using System.Web.Services;
using System.Web.Services.Protocols;
public class UserPass:SoapHeader{
public string PassWord;
}
public UserPass theHeader;
```

```
[WebMethod(), SoapHeader("theHeader")]
public string GetPassWord(){
        return "您送过来的是：" + theHeader.PassWord;
}
[WebMethod()]
public int Sum(int x, int y) {
        return x + y;
}
```

下面，我们来看看这个 SOAP 消息是怎么组织的。

1）SOAP 包封：

<soap:Envelope></soap:Envelope>定义了一个 SOAP 包封，它包装了 SOAP 消息的全部内容。

2）SOAP 头：

SOAP 头是选配的，使用头必须构造一个继承于 SoapHeader 的类。它的用途是要包括对于应用程序可能是非常重要的非方法调用的附加信息，比如验证、事务等等。它包含同步一个双方提交情境中一些合作伙伴的消息。

下面是一个请求的 SOAP 信息：

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Header>
    <UserPass xmlns="http://tempuri.org/">
      <PassWord>string</PassWord>
    </UserPass>
  </soap:Header>
  <soap:Body>
    <GetPassWord xmlns="http://tempuri.org/" />
  </soap:Body>
</soap:Envelope>
```

而响应信息可以是这样的格式：

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <GetPassWordResponse xmlns="http://tempuri.org/">
      <GetPassWordResult>string</GetPassWordResult>
    </GetPassWordResponse>
```

```
  </soap:Body>
</soap:Envelope>
```

其中：

<soap:Header> </soap:Header>为 SOAP 头标记

<soap:Body></soap:Body>为 SOAP 体标记

使用头元素必须遵循以下几个规定：

- 头元素是选配的，但一旦使用，就必须放在整个包封的第一个，也就是<soap:Envelope>标记的后面，否则就是错误的。
- 头元素必须用命名空间修饰，比如：

  <UserPass xmlns="http://tempuri.org/">

  </<UserPass >

3）SOAP 体：

SOAP 体中包含了双方需要传递的数据，由于 GetPassWord 方法是个无参数请求，所以请求中并不需要代参数，对于 Sum 方法而言，请求中就需要加入参数：

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <Sum xmlns="http://tempuri.org/">
      <x>5</x>
      <y>6</y>
    </Sum>
  </soap:Body>
</soap:Envelope>
```

对应的响应格式如下：

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <SumResponse xmlns="http://tempuri.org/">
      <SumResult>11</SumResult>
    </SumResponse>
  </soap:Body>
</soap:Envelope>
```

2，解读 WSDL

WSDL 是一个契约，你必须用某种方式向客户解释如何通过编程来调用这些服务，至少，

您的客户需要知道它必须传递什么参数，以及可以得到什么返回结果。也需要知道 Web 服务的实际地址，以及如何访问这个 Web 服务。这就需要一个机制，WSDL 协议就是为了解决这个问题而设立的。为了了解 WSDL，我们可以解读上面的例子生成的 WSDL 文档。

1）definitions（定义）

整个 WSDL 封装在 definitions 下。它首先定义了文档其它部分使用的最通用的协议、数据类型、模式的命名空间。必须说明，命名空间并不是必需的，但它有助于消除一些含糊之处，并可以避免名字上的冲突。

```
<?xml version="1.0" encoding="utf-8" ?>
 <definitions xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
                xmlns:soap=http://schemas.xmlsoap.org/wsdl/soap/
                 xmlns:s="http://www.w3.org/2001/XMLSchema"
                 xmlns:s0="http://tempuri.org/"
                 xmlns:soapenc=http://schemas.xmlsoap.org/soap/encoding/
               xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
               xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
               targetNamespace="http://tempuri.org/" xmlns="http://schemas.xmlsoap.org/wsdl/">
```

2）类型部分

类型部分定义了在 Web 服务中使用了哪些数据类型，以及可以从那里找到这些数据类型。事实上，这很象一般编程上的数据类型定义，也执行先定义后使用的原则。

```
<types>
<s:schema elementFormDefault="qualified"
            targetNamespace="http://tempuri.org/">
    <s:element name="GetPassWord">
      <s:complexType />
    </s:element>
    <s:element name="GetPassWordResponse">
      <s:complexType>
        <s:sequence>
            <s:element minOccurs="0" maxOccurs="1"
                      name="GetPassWordResult" type="s:string" />
        </s:sequence>
      </s:complexType>
    </s:element>
    <s:element name="UserPass" type="s0:UserPass" />
    <s:complexType name="UserPass">
      <s:sequence>
        <s:element minOccurs="0" maxOccurs="1" name="PassWord" type="s:string" />
      </s:sequence>
    </s:complexType>
    <s:element name="Sum">
      <s:complexType>
        <s:sequence>
```

```
            <s:element minOccurs="1" maxOccurs="1" name="x" type="s:int" />
            <s:element minOccurs="1" maxOccurs="1" name="y" type="s:int" />
          </s:sequence>
        </s:complexType>
      </s:element>
    <s:element name="SumResponse">
      <s:complexType>
        <s:sequence>
          <s:element minOccurs="1" maxOccurs="1" name="SumResult" type="s:int" />
        </s:sequence>
      </s:complexType>
    </s:element>
    <s:element name="string" nillable="true" type="s:string" />
    <s:element name="int" type="s:int" />
</s:schema>
</types>
```

3）消息部分

消息部分通过把特定消息的组成部分组成在一起，抽象的定义了送入和送出的消息。，基本上它是作为类型部分显式定义的参数具体化为参数传递的方式。

```
    <message name="GetPassWordSoapIn">
        <part name="parameters" element="s0:GetPassWord" />
    </message>
    <message name="GetPassWordSoapOut">
        <part name="parameters"
                element="s0:GetPassWordResponse" />
    </message>
    <message name="GetPassWordUserPass">
        <part name="UserPass" element="s0:UserPass" />
    </message>
    <message name="SumSoapIn">
        <part name="parameters" element="s0:Sum" />
    </message>
    <message name="SumSoapOut">
        <part name="parameters" element="s0:SumResponse" />
    </message>
    <message name="SumHttpGetIn">
        <part name="x" type="s:string" />
        <part name="y" type="s:string" />
    </message>
    <message name="SumHttpGetOut">
        <part name="Body" element="s0:int" />
    </message>
    <message name="SumHttpPostIn">
```

```
        <part name="x" type="s:string" />
        <part name="y" type="s:string" />
    </message>
    <message name="SumHttpPostOut">
        <part name="Body" element="s0:int" />
    </message>
```

4）接口部分

```
<portType name="Service1Soap">
    <operation name="GetPassWord">
        <input message="s0:GetPassWordSoapIn" />
        <output message="s0:GetPassWordSoapOut" />
    </operation>
    <operation name="Sum">
        <input message="s0:SumSoapIn" />
        <output message="s0:SumSoapOut" />
    </operation>
</portType>
<portType name="Service1HttpGet">
    <operation name="Sum">
        <input message="s0:SumHttpGetIn" />
        <output message="s0:SumHttpGetOut" />
    </operation>
</portType>
<portType name="Service1HttpPost">
    <operation name="Sum">
        <input message="s0:SumHttpPostIn" />
        <output message="s0:SumHttpPostOut" />
    </operation>
</portType>
<binding name="Service1Soap" type="s0:Service1Soap">
    <soap:binding
            transport="http://schemas.xmlsoap.org/soap/http"
            style="document" />
    <operation name="GetPassWord">
        <soap:operation
                soapAction="http://tempuri.org/GetPassWord"
                style="document" />
        <input>
          <soap:body use="literal" />
          <soap:header d5p1:required="true"
                  message="s0:GetPassWordPassWord"
                  part="PassWord" use="literal"
                  xmlns:d5p1="http://schemas.xmlsoap.org/wsdl/" />
        </input>
```

```
        <output>
            <soap:body use="literal" />
        </output>
    </operation>
    <operation name="Sum">
        <soap:operation soapAction="http://tempuri.org/Sum" style="document" />
        <input>
            <soap:body use="literal" />
        </input>
        <output>
            <soap:body use="literal" />
        </output>
    </operation>
</binding>
<binding name="Service1HttpGet" type="s0:Service1HttpGet">
    <http:binding verb="GET" />
    <operation name="Sum">
        <http:operation location="/Sum" />
        <input>
            <http:urlEncoded />
        </input>
        <output>
            <mime:mimeXml part="Body" />
        </output>
    </operation>
</binding>
<binding name="Service1HttpPost"
              type="s0:Service1HttpPost">
    <http:binding verb="POST" />
    <operation name="Sum">
        <http:operation location="/Sum" />
        <input>
            <mime:content type="application/x-www-form-urlencoded" />
        </input>
        <output>
            <mime:mimeXml part="Body" />
        </output>
    </operation>
</binding>
<service name="Service1">
    <port name="Service1Soap" binding="s0:Service1Soap">
        <soap:address
            location="http://localhost/WebSoapHeader/Service1.asmx" />
    </port>
    <port name="Service1HttpGet"
```

```
            binding="s0:Service1HttpGet">
          <http:address
              location="http://localhost/WebSoapHeader/Service1.asmx" />
    </port>
    <port name="Service1HttpPost"
            binding="s0:Service1HttpPost">
          <http:address
              location="http://localhost/WebSoapHeader/Service1.asmx" />
  </port>
</service>
</definitions>
```

显然，WSDL 作为契约足够确切而且无歧义。

五、面向服务的架构

1，面向服务架构的思考方式

面向服务的架构（SOA）是一种设计方式，它指导业务服务在其生命周期（从构思开始，直到停止使用）包括创建和使用的方方面。面向服务的架构也是一种定义和提供 IT 基础设施的方式，它允许不同的应用相互交换数据、参与业务流程，而不必关心它们各自背后使用的是何种操作系统，或者采用什么编程语言。

SOA 可以被看成一种构建 IT 系统的方案，它把业务服务（也就是一个组织向顾客、客户、合作伙伴直接或间接提供的服务）作为协调 IT 系统与业务需求的关键组织原则。相反，早期构建 IT 系统的方案，大多数受限于特定的使用环境来处理业务问题，结果造成 IT 系统依赖于具体执行环境的特点和性能。

SOA 的概念本身并不新，新颖之处在于它能够混合搭配各种执行环境，这样，IT 部门就可以为新的或者现有的应用选择最佳的执行环境，并且采用一致的架构方式把它们结合起来，这与原先基于一种执行一种环境和技术的 SOA 是有区别的。

关于“接口与实现分离”的思想，是早已在 J2EE、COM 中实现了并得到充分检验了的。但是，SOA 主要是通过文本文件来实现服务与其执行环境的分离，而且分离的更明确、也更完全。SOA 的这种能力是新增的，它主要借鉴于 Web 的概念和技术。传统的接口事先并没有考虑到这种“松散的”分离，因为这会影响它的效率。然而在许多情况下，易于获得互操作性的能力要比效率更重要。我们需要注意到，互操作性是业界一直在努力争取的，但至今仍然没有彻底实现的目标。

使用基于 WSDL 这样的 XML 文本方式描述接口和契约，就可以建立标准化的特性描述，任何语言和技术都可以利用某些符合标准的生成软件，基于软件来自动生成相应语言特征的映射层代理，而不需要人工来完成，这就比利用与技术相关的方式来描述 Interface 更具通用性。

SOA 是否成功，并不在于 Web 服务给 IT 软件的形式带来什么样的变化，而是依赖于处理问题的方法发生转变。Web 服务中接口与执行环境的进一步分离，促进了工作职责的分离，企业可以根据服务描述对业务运营问题的实现进行考虑，以及进行 IT 的投资。而依赖于具体产品的接口和服务描述的方案则不能这么做。这样，服务描述所定义的特征与功能就可以通过任何技术来实现。

这样的风格，只有转变了对 IT 的思考方式的企业才能实现这一点。当然，这种风格的转变并不是说完全不顾领域的特点，而是考虑在本领域中业务操作以更加快速而灵活的方式来

实现。同时，在支持 SOA 的环境中，企业可能不仅仅要考虑不同执行环境中的服务，也要知道如何使用来自各个厂商的组件来组装系统。

SOA 的真正优点体现在部署以后的各个阶段中，一旦实现了用可重用服务来组合新业务，就可以实现降低成本，以实现最大的投资回报率（ROI），但实现这一点是需要时间的，并且需要在服务开发上作相当大的投资。

重用通用的业务服务（比如客户姓名查找、邮政编码验证以及信用检查等）的好处是容易理解的。在出现 SOA 以前，这些通用功能是在可重用代码库或者类库中完成的。但是在 SOA 中，这些通用的功能以及常见的系统功能（比如安全检查、事务协调、审核）是通过服务来实现的。

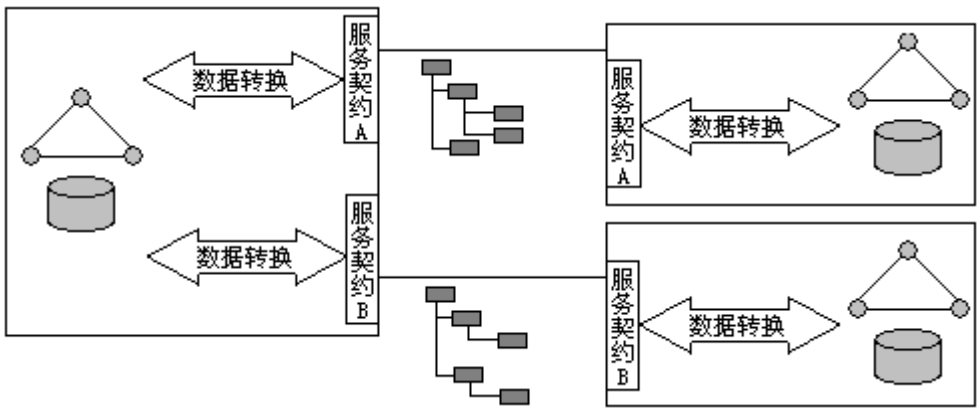
成功的 SOA 项目应该保证可重用的软件服务于实际的业务流程完全一致，业务流程与它的软件实现良好的配合与协调，令业务的运营流程可以根据外界的环境的变化作出快速变化，从而使组织能够适应发展的需要。

使用服务，不仅减少了需要部署的代码量，而且集中化的代码部署和管理还减轻了管理、维护和支持方面的负担。但是要注意到的是，访问服务的效率是需要经过评估的，因为通常使用服务比使用通用代码库要消耗更多的计算和网络资源。

2，协调不同的服务领域间的异构数据模型

服务领域，指的是一组特定行业（比如金融、销售、市场）相关的服务集合。某一服务领域中的所有服务应该采用一个通用的数据模型进行通讯。

在使用来自不同服务领域的服务的时候，应该理解不同领域模型间的语义与结构之间的差异。下图的服务请求者使用了来自不同服务领域的服务（服务 A 和服务 B），两个服务都定义了“ship data”，但是各自具有不同的含义。



3，面向服务的集成（SOI）

SOI 是在 SOA 的环境下使用 web 服务进行的集成，SOI 是战略性的、系统的使用 Web 服务来解决集成与互操作的问题。如果希望在某一个集成架构上做巨大的投资，那么 SOI 将是一个不错的选择。

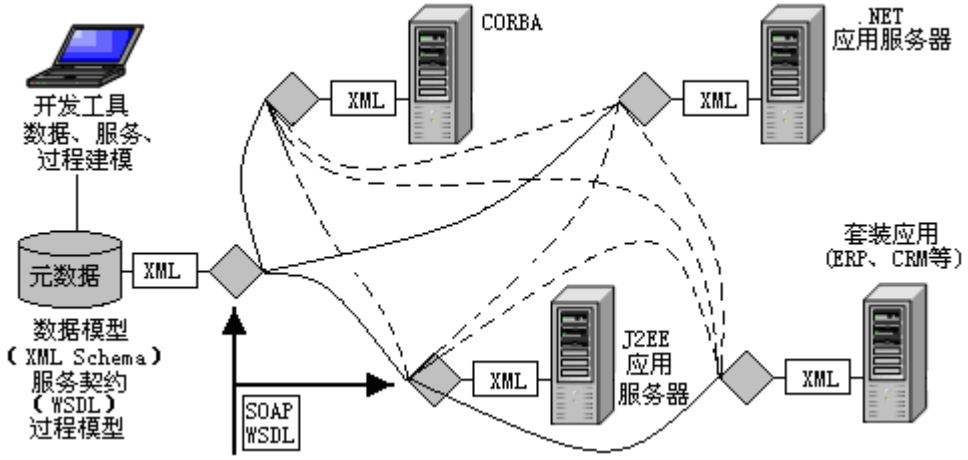
实现 SOI，应该第一个项目集成之前就开始，在启动 SOI 阶段：

- 定义 SOA 框架、过程、准则、模型和工具等；
- 对服务领域进行形式化建模。虽然这些模型不必要做到毫无遗漏，但应该是别处组织用到一些关键数据类型的、服务契约和过程等。
- 定义一个服务分类层次，以便各个集成项目可以对服务进行意志的分类和编目，以促进将来的重用；
- 如果 Web 服务平台为完成相似项目提供了多种选择，应该选择一种标准化的契约，比如 WSDL。

在这个框架下，SOI 项目通常包括：

- 改进现有数据模型，以适应当前集成项目；
- 根据服务契约对传统系统进行包装，创建当前集成系统所需要的新服务；
- 定义用于“进行不同数据模型的映射”的数据转换，以便数据能够跨越不同的数据领域边界；
- 为 Web 服务平台配置执行环境，以支持并实施企业级的服务质量（如事务需求、有保证的消息传递、恢复策略等）。

下图显示了 SOI 是如何随着时间的推移而逐步发展的。



最近的经验表明，解决“集成所面临的无数难题”的一个比较好的办法，是为已有的和新的系统增加一个抽象层，在 Web 服务中是 XML 层。而在这里，WSDL 在服务契约的定义上担任了关键角色。数据被转换为各个系统都能理解的 XML 形式，在发送时，本地数据被转换为 XML 形式，而接收消息时，XML 被转换为本地数据格式。

在开始阶段，用建模工具创建服务领域的初始数据、服务和过程模型，然后把他们保存在一个元数据仓库中。各个集成项目都是使用和改进同样的项目开始，所以使用诸如“客户”、“账户”的定义是一致的，尽管他们在各个传统系统的内不会有不同。IT 系统运行的时候，通过服务注册库（比如 UDDI）查找服务，然后直接调用服务。

这就形成了集中而又分散的 SOI 架构。

集中：所有的集成项目都访问相同数据。

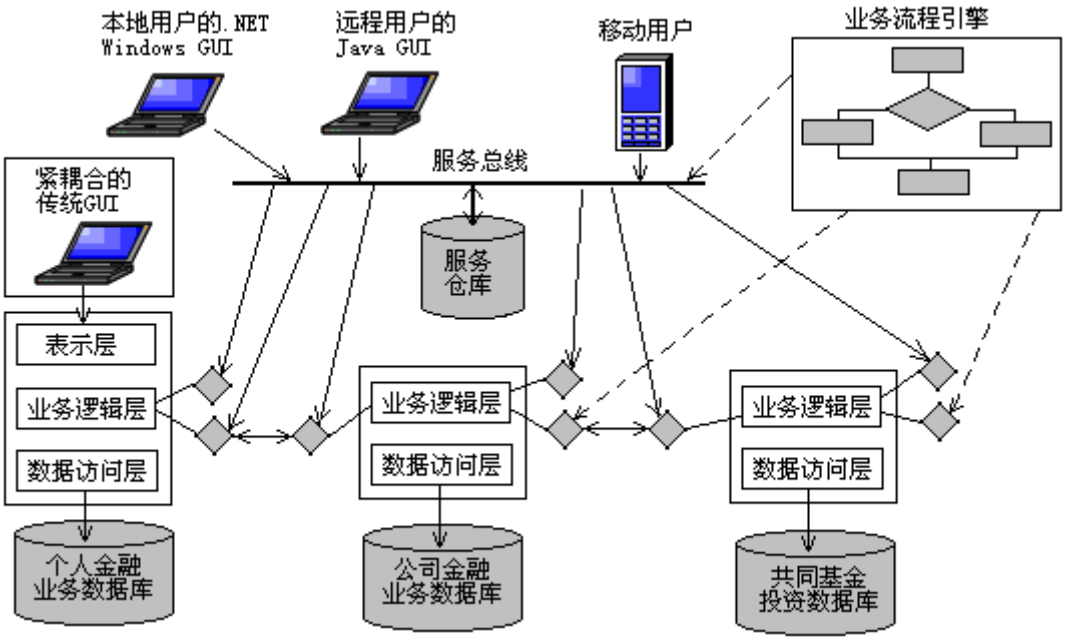
分散：任何系统都直接使用另一个系统提供的服务，而不需要通过集中控制来访问它。

Web 服务与 SOA 的组合，能够提供一种快速集成方案，它关注共享数据与可重用的服务，而不是专有的集成产品，因此能够更快、更轻松的确保 IT 投入与企业战略规划保持一致。

下面我们通过一个例子来展示面向服务的集成所带来的优点。

假定有三种比较典型的金融业数据库的应用，它们分别用于支持个人金融业务、公司金融业务以及共同基金投资等操作。用传统的三层架构来开发应用，将区分表示层、业务逻辑层和数据库逻辑三个层次。

在系统发展的历程中，这种传统的三层架构可以被重用为一个面向服务的应用，也就是在业务逻辑层创建服务，利用服务总线把上述应用于其它应用集成，如下图所示。



面向服务的一个优点是，如果业务逻辑层支持服务的话，就更容易实现表示逻辑与业务逻辑的分离，而不论客户端是 GUI 设备还是移动设备。表示逻辑置于一个特定的设备上（比如移动设备），而通过服务总线与业务逻辑通信，而不必为每种客户端构造紧耦合的表示逻辑层。

使用在业务逻辑层定义的 Web 服务，与其它集成技术相比，更有利于数据交换，因为 XML 可以独立的定义数据类型与结构，它体现了各种软件的公共标准。

在业务逻辑层进行面向服务入口点的开发，使业务流程管理引擎可以启动一个或者多个服务的自动执行流程。

服务总线实际上是一个公共的业务逻辑层，它通过“服务仓库”来存放和获取服务描述，如果一个新的应用要使用已有的服务，可以通过查询服务仓库获得服务描述（以 WSDL 表达），然后通过服务描述生成与该服务交互的 SOAP 消息。

6.3 SOA 与业务流程管理

一、业务流程管理的基本概念

业务流程（business process）是一种现实世界中的活动，它由一系列逻辑上相关的任务组成。如果根据恰当的顺序和正确的业务规则来执行这些任务，就可以产生业务效果。

我们在“需求抽取与业务建模”一章中定义的需求过程，就是一个典型的业务流程。

业务流程管理（Business Process Management，BPM）关注的是组织如何识别、建模、开发、部署和管理业务流程（其中也包括 IT 系统与人交互的过程）。

BPM 的主要目标与优点如下：

- **减少业务需求与 IT 系统的失配**：通过允许业务用户对业务流程进行建模，然后由 IT 部门提供执行和控制这些业务流程的基础设施。
- **提高员工的生产力，降低运营成本**：通过把业务流程自动化和流畅化。
- **提高组织的机动性和灵活性**：通过把业务逻辑与其它业务规则显式分离，并且用一种“易于随业务需求的变化而修改”的形式来表示业务流程。这样，组织将更具机动性，能够针对市场的变化做出更快的响应，并快速取得竞争优势。
- **降低开发成本**：通过使用一种高层的、图形化的编程语言，令业务分析师和开发人

员可以在特定的问题领域中快速构建和更新 IT 系统。

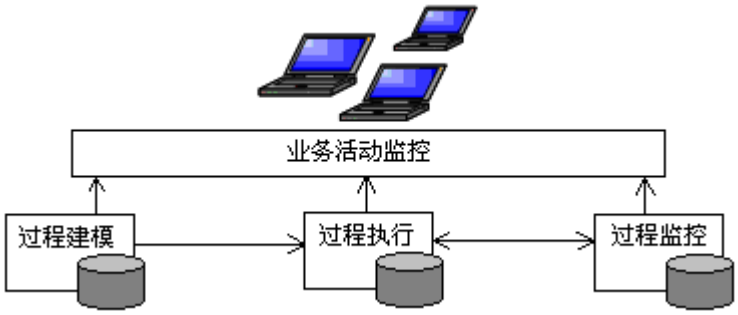
业务流程自动化，就是把组织中原来需要由人来处理的活动变成企业级的、高度自动化的系统。业务流程自动化一般会涉及到业务流程的跟踪，也就是文档、信息或者任务在参与者之间传递，以确保活动遵守业务规则。

业务流程管理（BPM）一个显著的特点，就是把业务流程逻辑从其它规则中分离出来，这与早期业务流程深嵌在代码中的情况形成明显的对比。

二、业务流程管理系统

所谓业务流程管理系统（BPMS）提供实现一个或者多个核心 BPM 功能的技术。

业务流程管理系统的名称可能是多种多样的，比如工作流、过程自动化、过程集成、B2B、服务合成、编制和编排等，它提供了一种流程建模工具，允许用图来定义流程。下图显示了 BPMS 的基本组件。

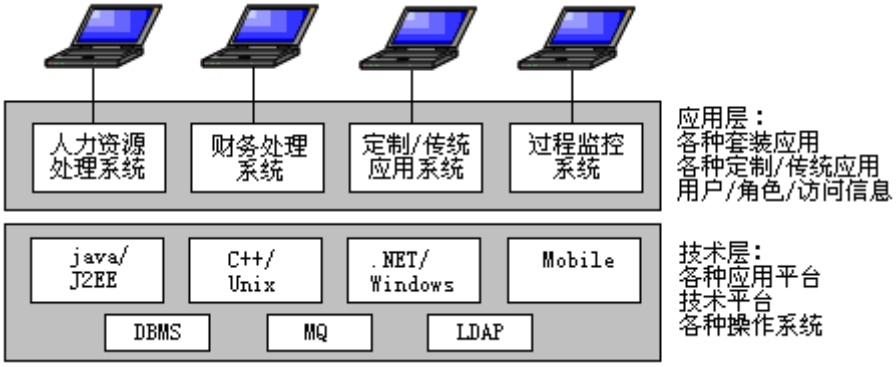


三、组合 BPM、SOA 与 Web 服务

1，组合 BPM、SOA 与 Web 服务的原因

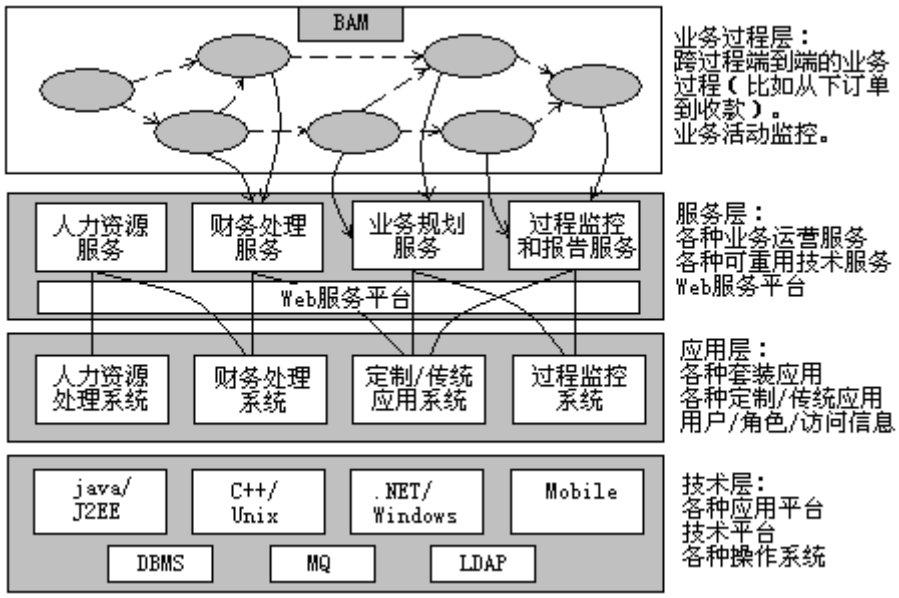
通常，过程流由多个独立的任务组成，其中每个任务都是一个独立的 Web 服务。过程流会根据一项任务的执行结果，来选择不同的分支继续执行，比如，如果“验证账户信息”是无效信息的话，将转入“修正账户信息”过程。

在企业信息化的历史变迁中，大多数组织都存在不同历史时期的应用和技术共存的情况，由于技术平台与数据模型的差异，要在这些硬功之间共享信息是很困难的，如下图所示。



今天的业务流程往往需要根据市场与用户特征发生变迁，但是，由于早期技术过程流嵌入到具体技术中，修改业务流程是比较困难的。但是 SOA 架构要求通过抽象的契约来暴露任务，这就有助于灵活的组合业务流程。

转向一种面向服务的架构和 Web 服务，需要加入一个服务层。服务层包含针对特定业务领域的特定服务、适应于各个业务领域的数据模型以及 Web 服务平台，这样就可以实现一种与下层应用技术无关的方式来定义和使用业务服务。再上一层是业务流程层，如下图所示。



BPM 作为一个层，有助于简化把解决不同问题的 Web 服务组合起来的任务，这种任务在过去是非常困难的。如果把服务看成是 IT 系统与业务功能（比如处理订单）的对应，那么，BPM 层就可以看成把多个服务联合在一起完成一定功能的（比如验证订单）的过程流。通过有应用层代码组成的过程流，业务流程将更易于针对新的应用特性与功能需求，作出变更和更新。

服务层和业务流程层提供了理想的平台，这是因为：

- 业务运营服务提供了粗粒度的业务功能（一个业务功能提供了对应于业务流程中的一个任务）。
- 业务运营服务的服务契约，为访问服务提供了良好定义的、无歧义的接口，因此业务流程无需了解下层应用技术平台的细节。
- 服务层的服务注册库与服务发现设施确保了业务流程层可以根据需要动态找到并访问服务。
- 服务数据模型是根据服务业务领域定义的，而且是独立于特定应用数据模型的。
- 由于 XML 与下层应用所用的数据格式无关，任务在与其它任务交换数据或者调用服务的时候，均采用 XML 作为规范格式。
- 服务层的安全模型提供了单点登录和基于角色的访问控制，这确保了任务可以获得使用服务的权限，并且令业务流程层免于处理各种本来是下层技术平台提供的安全接口。
- 服务层管理模型可以生成有关服务使用状况的运行时统计数据，供业务流程层的 BAM 工具使用。

在过去，没有提供基于 SOA 的 Web 服务的服务层，因为业务流程需要直接访问下层业务应用，所以 BPM 不但复杂，而且是脆弱的。

说它复杂，是因为业务流程必须通过各个应用定义接口（API、消息队列、数据库表）直接访问应用，这需要使用者必须精通这些应用接口。这往往需要在业务流程中增加步骤，以弥补不符合业务需要的应用接口定义。

说它脆弱，是因为流程直接与特定的应用接口紧耦合，这就可能为了一个升级版本，导致所有访问它的流程出问题。紧耦合也为修改应用增加了困难。例如把一个已有的流程更换为别的厂商提供的新应用，那么原来的流程必须经过修改才可以访问新的应用，这往往可能导致系统使用上的困难。

2，定义原子与合成服务

为了进一步了解基于 BPM、SOA 及 Web 服务解决方案在设计与实现上的细节，我们可以把服务分为两种：

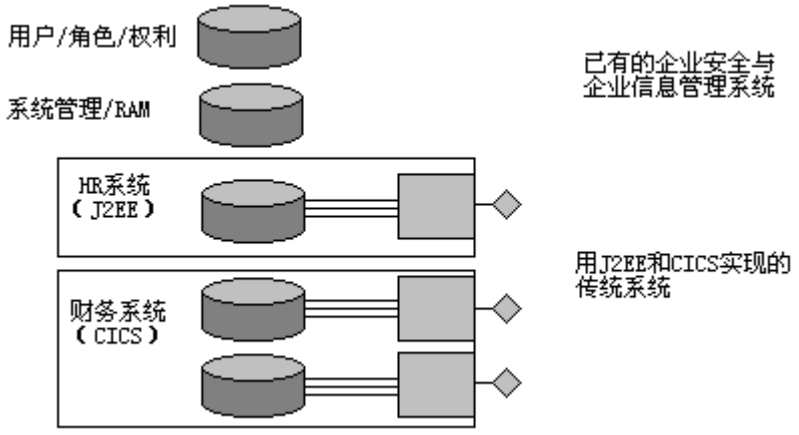
原子服务（atomic service）：不可在分解为更细粒度的服务。

合成服务（composite service）：由其它服务组合而成的服务。

建立 SOA 架构体系，需要有一种形式化的方法，使我们的设计过程规范、清晰、有条理，而且每个阶段的关注点也比较集中。下面我们通过一个例子，讨论一个原有的传统系统，是如何发展成新开发的业务运营服务，最终为内部和外部用户提供业务服务的细节。通过这个例子，说明每个步骤需要考虑的问题以及方法上的重点。

初始系统：

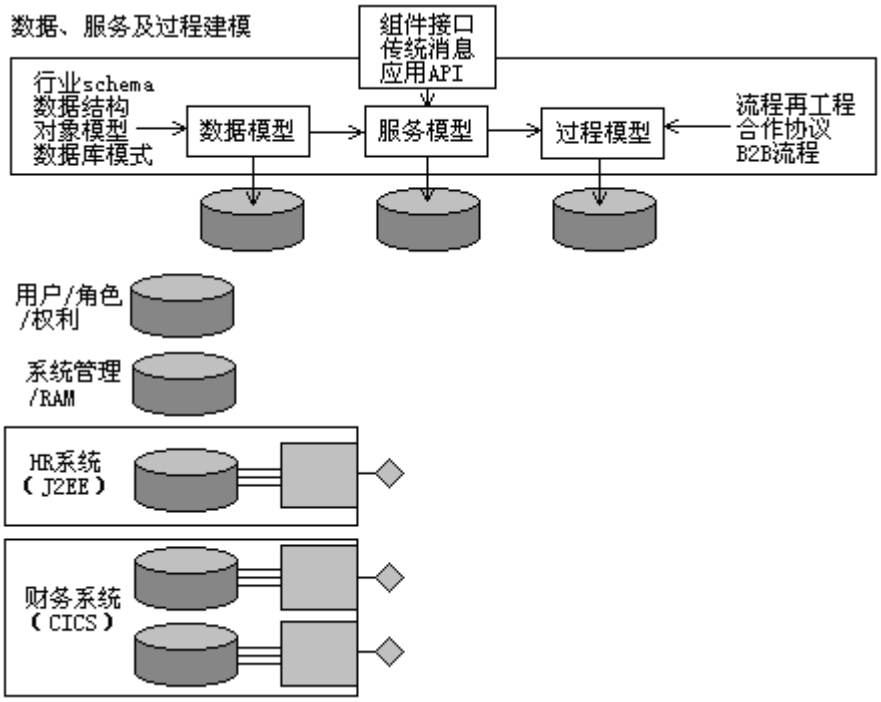
假定，下图所示的初始系统中包括一个传统的人力资源管理系统（HR）、一个传统的财务系统（CICS）、一个传统的企业安全系统（用于管理用户/角色权利信息，例如 LDAP）以及一个已有的企业信息管理系统。



HR 系统是用 J2EE 实现的，它包含一个应用数据库和一个应用对象模型，它还提供一个 EJB 接口。财务系统运行于大型机上，它是针对大型机数据库 CICS 事务实现的，客户端可以通过 WebSphere MQ 访问它。

第一步，定义数据、服务与过程模型：

要用一种面向服务的架构（SOA）来提供可重用的服务，第一步就是定义业务领域的数据、服务与过程模型，如下图所示。



服务层数据模型：定义将在服务间交换，以及将提供给服务请求者使用的业务层数据。

服务层数据模型包括数据定义（XML Schema）、数据验证规则（XML Schema 约束以及 XPath），数据转换规则（XSLT）。在理想情况下，应该根据现有的行业架构来创建服务层数据模型，但实际上，还需要参考现有的数据结构、对象模型以及数据库模式才能得出服务层数据模型。因此，数据建模者必须对这些底层的数据定义进行仔细的抽象，才能创建出一个真正与应用和技术无关的服务层数据模型。

服务模型：定义了业务运营服务的服务契约（例如 WSDL），该服务契约定义了：

- 服务的输入和输出参数（根据服务层数据模型定义的文档类型）。
- 服务的安全概要（例如权力、访问控制列表、保密及不可否认性等）。
- 服务质量（优先级、有保证的递送、事务特征及恢复语义等）。
- 服务水平协议（例如响应时间、可用率等）。

服务分为原子服务与合成服务两种。与服务层数据模型类似，在理想情况下，应该根据现有的行业服务定义来创建服务模型，但实际上，可能要根据现有的组件接口（例如 COM/DCOM 类型库、CORBA IDL、Java 对象等）、传统消息格式以及现有应用的 API 才能得出比较现实的服务模型。

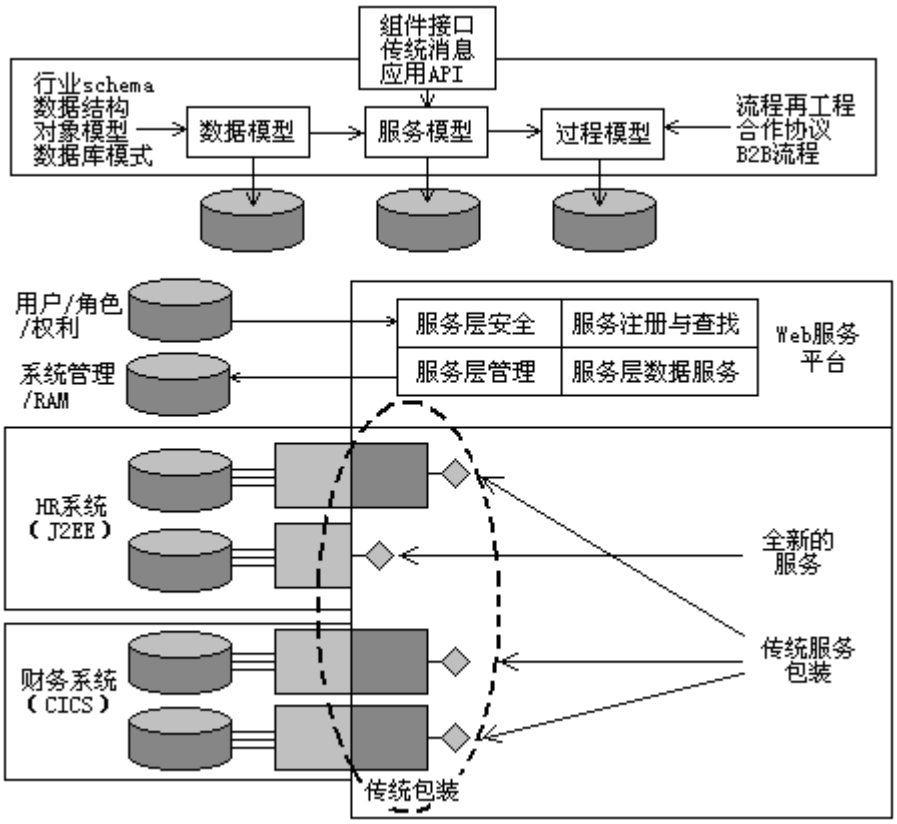
过程模型：定义了后面要讨论的采用业务流程执行语言（WS-BPEL）的方案实现的业务流程。各个过程定义包括：过程任务、任务间的控制流、任务间的数据流一基于过程相关的其它业务规则，例如，过程或者任务的前提条件、过程或者任务的最后期限、给用户分配任务的规则、进行路由警报的规则、进行自动调整问题的规则等。过程模型需要根据现有的过程定义，经过再工程后的过程定义、过程交互的合作协议、行业过程定义等创建的。

在理想情况下，应该先创建服务层数据模型，然后在它的基础上进行服务模型的建立，最后再根据服务模型来定义过程模型。不过很多情况下，这种定义是以迭代的方式进行的。

第二步，定义原子服务：

下图展示了如何根据数据模型来定义原子服务。在这个例子中，有三个服务是通过为传统系统提供传统服务包装实现的。传统服务包装的作用是为传统系统提供了一个 Web 服务接口（SOAP 与 WSDL），由它负责接收输入的 SOAP 消息，并把它转换成传统系统能够理解的

格式，然后把请求传递给传统系统（例如调用 EJB 或者往 WebSphere MQ 队列中加一个消息）。在这个例子中，有一个服务是通过实现一个新的 J2EE 组件，并把它发布为 Web 服务实现的。

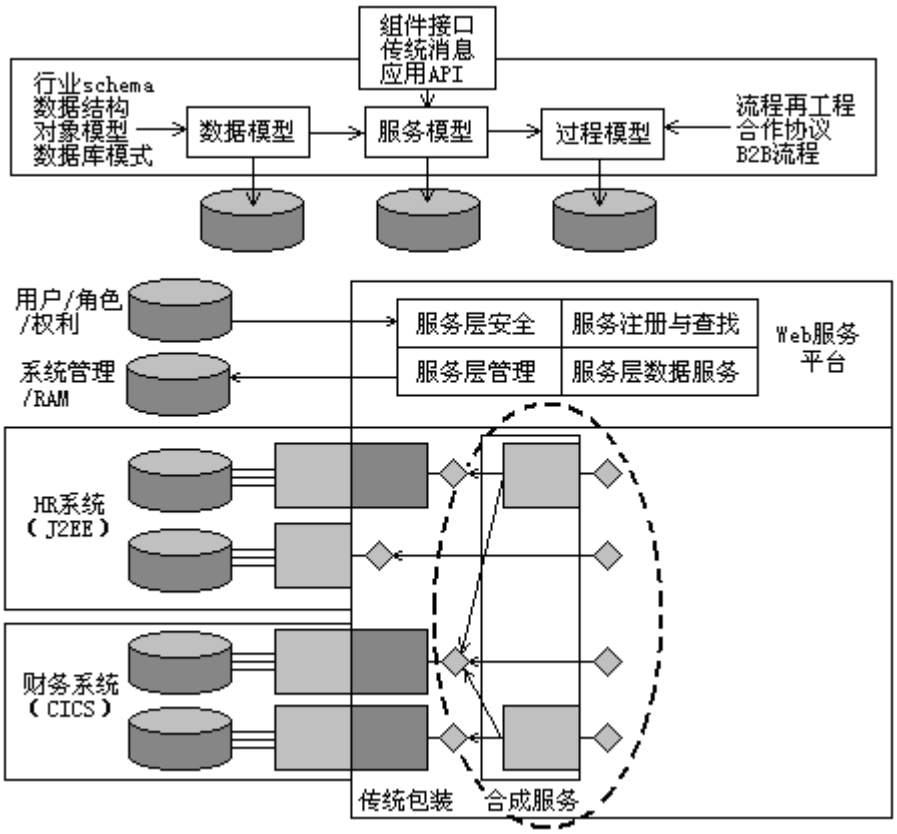


上图还展示了 Web 服务平台，它提供了用于定义、注册、保护和管理原子服务的核心设施。这个 Web 服务平台利用了已有的企业安全系统，包括支持 SAML、XKMS、XACML、WS-Security 等。还考虑了目录服务，这种服务类似于 LDAP 或者 ADS，其中包括用户/角色/权利等信息。

这个 Web 服务平台还利用了已有的支持 WSDM 的企业信息管理系统，WSDM 有例如 Hp Openview 或者 IBM Tivoli 系统，用于监控和管理服务。

第三步，定义合成服务：

下图展示了如何根据原子服务来定义合成服务服务的层次。



合成服务指的是那些由其它服务组合而成的 Web 服务，合成服务与其它的 Web 服务类似，因为它们都有 WSDL 契约，并且都是通过 SOAP 调用的。

我们可以通过直接编程的方式来创建合成服务，例如把一个 EJB 发布为用到其它 Web 服务的 Web 服务。也可以通过使用 Web 服务编制（Web Service Orchestration）以及 WS-BPEL 来创建合成服务，在这种情况下，开发者一般使用 Web 服务编制产品来定义合成服务，这种产品提供了用于合成 Web 服务的图形用户界面，可以通过它生成相应的 WS-BPEL 流程定义，以及一个执行 WS-BPEL 流程定义的运行时引擎。利用 WS-BPEL 进行 Web 服务的合成具有简单和灵活的优点，而且修改合成服务不需要引入新的代码，不过需要对效率进行评价。

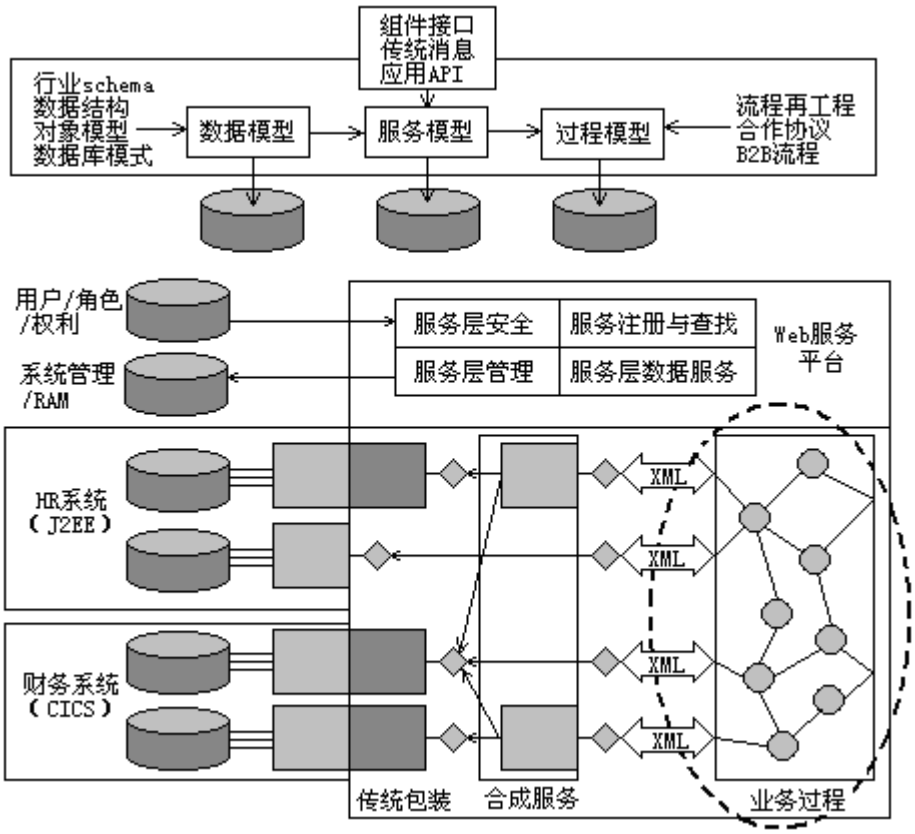
有时候，采用直接编程的方式也不失为一些好方法，特别是对于那些比较简单或者特别关注效率的合成服务尤其如此。

Web 服务平台为支持创建合成服务提供了一下设施：

- 在设计时和运行时发现已有的服务；
- 注册新的合成服务；
- 安全的访问已有服务；
- 保护新的合成服务；
- 利用 WS-BPEL 编制已有服务；
- 在访问已有的服务时应用数据验证规则；
- 在收到已有服务发来的数据之后、或者把数据发送给已有服务之前，进行必要的数据转换，这种转换包括数据的聚合、过滤与分割。

第四步，定义业务流程：

业务流程实现了复杂的、多步的业务功能，它通常涉及到多个参与者，包括内部用户、外部用户与合作伙伴等。业务流程的运行时语义是由 WS-BPEL 来定义的，过程引擎负责执行过程任务并确保根据 WS-BPEL 脚本实施有关的业务规则。



业务流程的各个任务，要不是由 Web 服务来完成，要不就是由用户来完成。如果任务是由 Web 服务来完成的，那么过程引擎要负责找到并调用相应的 Web 服务，如果任务是由用户来完成的，那么引擎就负责把任务传递给一个被授权的用户。

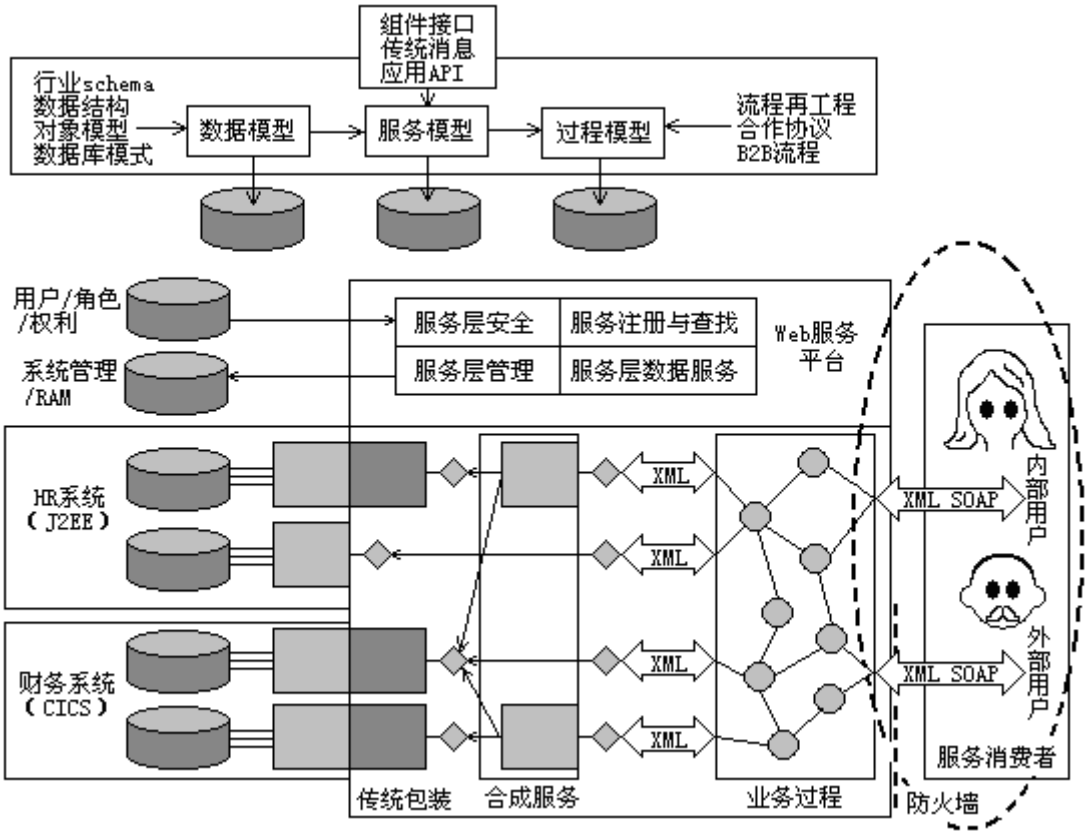
业务流程还可以定义如何处理事务，发生错误的时候如何恢复语义（例如是否请求 ACID 事务或者补偿事务）。业务流程本身也可以是一个 Web 服务，这样任何服务请求者（包括其它业务流程）都可以启动它。

Web 服务平台为创建和执行业务流程而提供的设施包括：

- 在设计时与运行时发现已有的服务；
- 把新的业务流程注册为一个业务流程；
- 安全的访问已有服务；
- 保护新的业务流程；
- 用 WS-BPEL 编制已有的服务；
- 允许业务流程在访问已有服务的时候应用数据验证规则；
- 允许业务流程对任务间传递的数据执行数据转换（例如数据的聚合、过滤和分割）；
- 生成关于服务使用状况的运行时统计数据，供业务流程层的 BAM 工具使用。

第五步 研究服务消费者访问和使用 Web 服务的方法

作为 SOA 架构的应用考虑，我们必须研究服务消费者访问和使用 Web 服务的方法，使内部和外部用户可以方便的使用业务服务。消费者使用 Web 服务如下图所示。



只要是被授权的服务请求者（包括 IT 系统、最终用户应用以及其它的 Web 服务），应该可以随时随地访问任何服务，这些服务既包括原子服务，又包括合成服务，当然也包括作为 Web 服务发布的业务流程，而并不能仅仅规定用户只能使用最终组合成功的业务流程。

有时，组织对其外部的服务请求者（比如客户或者合作伙伴）之提供少数可供使用的服务，而且对外部请求者必须比组织内部服务请求更严格的安全要求。

事实上，无论是对于内部还是外部请求者，组织必须发布业务流程和服务使用的案例，这些简单又包括所有要素的案例，对于服务和业务的良好应用，都是非常必要的。

四、编制与编排规范

Web 服务正逐渐成为系统架构和实现组织内外的业务流程与业务协作的基础，从上面的讨论可以看出，实现的关键是需要一种规范、统一、功能强大的服务与业务流程的编排语言，以及与之对应的产品。目前已经存在两种 Web 服务合成语言：

- **业务流程执行语言（Business Process Execution Language，WS-BPEL）**：这是由 BEA、IBM、Microsoft 以及 Siebel 制定的，而后被提交给了 OASIS WS-BPEL 技术委员会的标准。
- **Web 服务编排语言（Web Service Choreography Description Language，WS-CDL）**：这是由 W3C Web 服务编排工作组制定的。该规范是基于一个由 Sun、Intalio、BEA、SAP 等制定的规范制定的。

上述两种语言的目标都是：以一种面向过程的方式，把多个 Web 服务粘合起来。

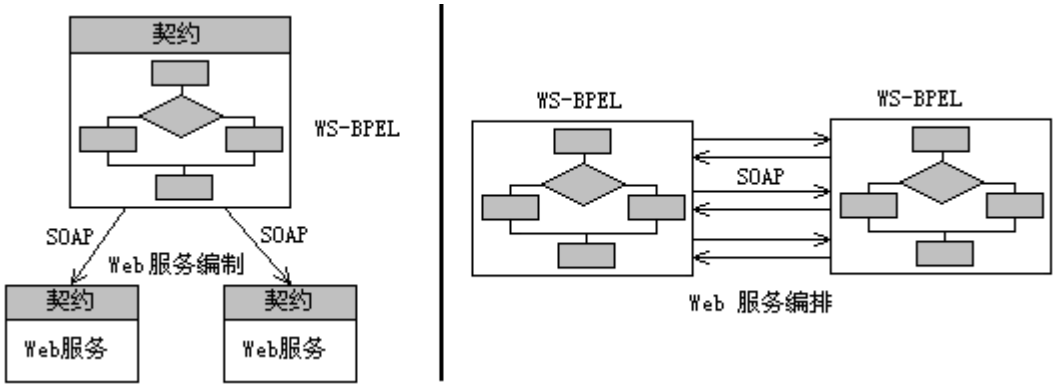
1， Web 服务编制与编排的比较

编制（orchestration）和**编排**（choreography）是用于描述合成 Web 服务的两种术语，虽然它们有共同之处，但还是有区别的。

Web 服务编制（Web Service Orchestration，WSO）：指的是为业务流程进行的服务合成。它主要用于重用已有服务的内部过程。

Web 服务编排（Web Service choreography，WSC）：指的是为业务协作（business collaborations）而进行的业务合成。它主要定义多个方面如何在一个更大的事务中，通过与交易伙伴及外部机构（比如供应商与客户）交换信息，进行对等（peer-to-peer）协作。

两种合成方式的区别如下图所示。



WSO 关注于以一种说明性的方式、而不是编程的方式来创建合成服务。它定义了组成编制的服务，以及这些服务的执行顺序（顺序、并发、条件分支）。因此，可以把编制视为一种简单的流程，这种流程本身也是一个 Web 服务。WSO 流通常包括分支控制点、并行处理选择、用户响应步骤以及各种预定义的步骤（例如转换、适配器、电子邮件及 Web 服务等）。

WSC 关注于定义多方如何在一个更大的业务事件中进行合作，它通过各方描述自己如何与其它 Web 服务进行公共消息交换来定义业务交互，而不是像 WSO 那样描述一方是如何执行某个具体业务流程的。

这种业务协作定义，需要涉及各方面指定的可观察到的消息行为，而不暴露内部实现细节，这样做的好处有两个：

- 组织不愿意把内部业务流程与数据管理暴露给业务伙伴。
- 公开和私有的流程分离后，内部流程实现上的变化不会影响公开的业务协议。

业务流程执行语言（WS-BPEL）关注于 WSO 和各个 Web 服务的合成，也可以用于驱动跨企业边界的 WSC 形式的交互。而 Web 服务编排语言（WS-CDL）关注于 WSC 和企业之间关系的定义。从另一个方面来说，CDL 不太适合于 WSO 和进行 Web 服务的合成。CDL 可以把 Web 服务端点作为企业之间协作的入口点，但它更关注于定义企业之间的关系。

CDL 定义了自己与 WS-BPEL 的关系，但是 WS-BPEL 并没有定义自己与 CDL 的关系，许多 WS-BPEL 的支持者认为，这是一种支持各种交互的唯一需要的语言，但还是有很多人认为一些补充的语言还是需要的，因为一种语言不可能处理所有的事情，尤其是多方协作的细节定义，多种语言的并存和协调恐怕是未来 SOA 的一种特征。

2，业务流程执行语言（WS-BPEL）

Web 服务业务流程执行语言（WS-BPEL）是一种面向过程的服务合成语言，它是面向服务的，并且依赖于 WSDL。一个 WS-BPEL 过程可以发布为一个 WSDL 定义的服务，可以像其它 Web 服务一样被调用。

另外，WS-BPEL 希望一个 Web 服务合成所包含的全部外部 Web 服务，这都是用 WSDL 服务契约定义的，这令 WS-BPEL 流程可以调用其它的 WS-BPEL 流程，甚至可以递归的调用自己，这就具有很好的灵活性。

WS-BPEL 被称为是对 WSFL 和 XLANG 中优秀成分的组合。

IBM 的 WSFL（Web Service Flow Language，Web 服务流语言）是一种图结构语言，它

主要依赖于控制链（control links）的概念。而微软的 XLANG（Web Service for Process Design，用于业务流程设计的 Web 服务）是一种块结构化语言，它的基本控制流结构有顺序（sequence）、选择（switch）、循环（while）、并行（all）等。WS-BPEL 允许混合块结构与图结构的过程模型，但这样一来也可能会引起到底使用什么方式的困惑，所以，WS-BPEL 的开发者如果缺乏经验，很可能会制造出一种最差的组合。

WS-BPEL 定义了一组创建 Web 服务合成的**基本任务**：

- **Invoke 任务**：允许业务流程在某一个 Web 服务提供的 portType 上调用单向的或者请求/响应操作。
- **Receive 任务**：允许业务流程停下来等待消息到来。
- **Reply 任务**：允许业务流程对收到的消息发送一个回复消息。
- **Wait 任务**：通知过程等待一段时间。
- **Assign 任务**：把数据从一处复制到另一处。
- **Throw 任务**：表明发生了某个错误。
- **Terminate 任务**：中止整个编制实例。

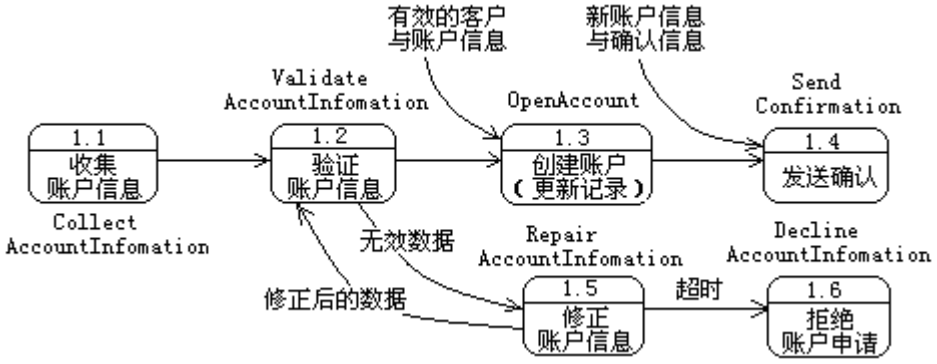
另外还定义了一组**结构化任务**，以便于把原子任务组合为更复杂的过程：

- **Sequence 任务**：定义一个有序的任务序列。
- **Switch 任务**：根据条件选择某个分支。
- **Pick 任务**：停下并等待某一适当的消息到来，或者等到超时继续前进，只要多个触发器中的一个发生，就执行相应的活动，任务便结束了。
- **While 任务**：定义选环执行，直到满足某一条件的一组任务。
- **Flow 任务**：表明一组应该并行执行的步骤，可以通过建立连接来定义一个特定过程的执行序列。

由于 XML 描述的标准化，生成 WS-BPEL 就可以应用相应的图形界面产品方便的构造过程，自动地完成 XML 文档的编制，而不需要自己写任何具体的 XML 语句。生成文档的目的也与 WSDL 一样，是希望借助于某些工具软件自动生成业务流程的映射层代理，所以，WS-BPEL 可以作为 WSDL 的一部分存在。

尽管实际开发中并不需要直接书写 XML 文档，但是借助一个简单的、有代表性的例子，研究清楚 WS-BPEL 的 XML 格式应用是有意义的，因为这样可以使我们对问题的理解更深入也更直观。

下面的例子描述了如下图所示的过程，这个例子展示了 WSDL 如何与 WS-BPEL 关联，并共同实现利用 Web 服务完成业务流程自动化的目标的。由于 WSDL 版本还在变化，在下面的讨论中，我们将关注于构建逻辑，可以把描述看成一种伪码。



第一步 定义消息（message）：

首先对客户信息记录进行定义，在过程流的开始步骤将会用到它。记录结构的定义，既可以放在 WSDL 文档中，也可以放在相关联的文件中，然后在 WSDL 文档中通过 include 或者 import 元素来引用该文件，消息类型的定义，一般建议使用 XML Schema。

```
<xs:element name="openA:CustomerInfo" type="openB:CustomerInfo" />
    <xs:complexType name="openA:tCustomerInfo">
        <xs:sequence>
            <xs:element name="AccountNumber" type="xs:double"/>
            <xs:element name="CustomerName" type="xs:string"/>
            <xs:element name="CustomerAddress" type="xs:string"/>
            <xs:element name="CustomerType" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>
```

上面定义的是一个消息类型，在具体的消息中需要与之建立关联。对于导入的数据类型与结构，一般用命名空间（namespaces）来限定其元素名与属性名.

下面，为模式数据类型 CudtomerInfo 与 WSDL 消息类型 CustomerMessage 建立了关联。

```
<message name="CustomerMessage">
    <part name="CustomerInfo" type="openA:tCustomerInfo"/>
</message>
```

要使用 CustomerInfo 数据类型，需要定义命名空间，比如：

```
xmlns:openA="http://www.bank.com/xsd/AccountManagement"
```

第二步 为消息定义接口类型：

然后，需要为消息定义 portType（从 WSDL 1.2 开始，portType 改名为 Interface），它包括操作名称以及输入/输出消息，例如：

```
<portType name="CollectAccountInfo">
  <operation name="CheckInfo">
        <input message="openA:CustomerInfo"/>
        <output message="openA:validInfo"/>
        <fault name="InvalidData"
              message="openA:CustomerDataError"/>
  </operation>
</portType>
```

绑定（bindings）属于 WSDL 的物理部分，这里省略了它们，因为采用何种通信协议，并不影响到 WSDL 的逻辑部分或者 WS-BPEL 的定义。

第三步 定义合作链接：

在 WSDL porTypes 的后面，需要添加称之为合作链接（partner links）的结构，它用于连接 portType 与 WS-BPEL 流程定义，例如：

```
<Plnk:partnerLinkType name="CheckCustomer">
    <Plnk:role name="AccountManagement">
```

```
            <Plnk:portType name="openA:CollectAccountInfo"/>
        </Plnk:role>
    </Plnk:partnerLinkType>
    <Plnk:partnerLinkType name="ValidateAccount">
        <Plnk:role name="ValidRequest">
            <Plnk:portType name="openA:ApproveApplication"/>
        </Plnk:role>
        <Plnk:role name="InvalidRequest">
            <Plnk:portType name="openA:RejectApplication"/>
        </Plnk:role>
    </Plnk:partnerLinkType>
```

这些合作伙伴链接把 WSDL 操作（）与 WS-BPEL 过程中的步骤（）相关联，并且用于衔接“服务以及接口的描述”和“使用这些服务的过程流的描述”。

第四步 定义 WS-BPEL 过程：

在扩展的 WSDL 中，WS-BPEL 部分是通过一个过程名称（proccess name）来标识的，例如：

```
<proccess name="OpenAccount"
        targetNamespace="http://www.bank.com/wsdl/accountManagement"
        xmlns="http://www.bank.com/2008/01/schemas/AccountService.xsd"
        xmls:openA="http://www.bank.com/wsdl/OpenAccount"
        abstractProcess="no">
```

一个抽象流程是是不可直接执行的，如果要用 WS-BPEL 来编排而不是编制，就需要设定 abstractProcess="yes"。这部分定义所需要的命名空间就在这部分定义，给出过程名称以后，还要引用该过程合作者链接（partner links），以指出该过程所要使用的 WSDL 相应的合作者链接，例如：

```
<partnerLinks>
    <partnerLink name="CollectAccountInfomation"
            partnerLinkType="openA:GetAccountInfo"/>
    <partnerLink name="ValidateAccountInfomation"
            partnerLinkType="openA:CheckAccountInfo"/>
    <partnerLink name="OpenAccount"
            partnerLinkType="openA:OpenAccountOK"/>
    <partnerLink name="SendConfirmation"
            partnerLinkType="openA:ConfirmOpen"/>
    <partnerLink name="RepairAccountInfomation"
            partnerLinkType="openA:RepairInfomation"/>
    <partnerLink name="DeclineAccountInfomation"
            partnerLinkType="openA:DeclineApplication"/>
</partnerLinks>
```

这里列出了过程流中的六个步骤所需的六个合作链接，包括：

CollectAccountInfomation：收集账户信息；
ValidateAccountInfomation：有效帐户信息；
OpenAccount：新建帐户；
SendConfirmation：发送确认信息；
RepairAccountInfomation：修改帐户信息；
DeclineAccountInfomation：降级帐户信息
在合作链接列表之后是变量名称的声明，也就是过程所需的 XML Schema、XML Simple 以及 WSDL 消息定义，例如：

```
<variables>
    <variable name="CustomerData"
          messageType="openA:CustomerRecord"/>
    <variable name="ValidData"
          messageType="openA:CustomerRecord"/>
    <variable name="AccountData"
          messageType="openA:AccountRecord"/>
    <variable name="ConfirmstionData"
          messageType="openA:ConfirmstionRecord"/>
    <variable name="FaultData"
          messageType="openA:FaultRecort"/>
</variables>
```

如果过程需要错误处理的话，可以进行如下定义：

```
<faultHandlers>
    <catch faultName="openA:InvalidCustomer" faultVariable="BadData">
        <reply partnerLink="ValidateAccountInformation"
                portTrye="openA:RejectApplication"
                operation="sendRejectionMessage"
                variable="CustomerData"
                faultName="cannotOpenAccount"/>
    </catch>
</faultHandlers>
```

这个错误处理程序用于捕捉在验证客户数据的过程中产生的错误，比如错误的信用数据，或者过去与银行发生的问题。

第五步 定义过程流序列：
最后，定义过程流序列，把操作放到执行关系中去，例如：

```
<sequence>
    <receive partnerLink="CollectAccountInfomation"
        interface="openA:AccountManagement"
        operation="InputData"
        variable="CustomerData"/>
```

```
    <assign>
        <copy>
            <from variable="CustomerData"/>
            <to variable="AccountData"/>
        </copy>
    </assign>
    <invoke partnerLink="ValidateAccountInfomation"
          interface="openA:AccountManagement"
          operation="ValidateData"
          variable="AccountData"/>
    <assign>
        <copy>
            <from variable="AccountData"/>
            <to variable="ValiData"/>
        </copy>
    </assign>
    <receive partnerLink="OpenAccount"
          interface="openA:AccountManagement"
          operation="OpenAccount"
          variable="ValidData"/>
</sequence>
```

这个例子展示了过程流如何先通过收集客户信息功能（CollectAccountInfomation）接受输入的客户数据，然后调用验证帐户信息功能（ValidateAccountInfomation）检查数据。根据验证结果的不同，过程流可以进入批准阶段，也可以进入尝试清除错误阶段。WS-BPEL 为这种分支提供了多种不同的条件动词，如 switch、pick、while 等。

WS-BPEL 关联集（correlation sets）标识在过程流中的多个 Web 服务间共享的数据，补偿处理程序将执行有可能撤销先前结果的补偿程序（比如关闭前面未正确打开的账户等）。

在设计中需要注意的是，除了把 WS-BPEL 语法合并到 WSDL 所带来的麻烦以外，还必须把 WS-BPEL 构词与其它技术（比如安全性、可靠性、事务性等）的过程流构词区分开来，以免在一起使用时发生混淆，这样才能确保整个过程设计的正确工作，当然，这也有赖于 SOA 构词描述的进一步标准化。

3，以编制为中心的服务合成案例

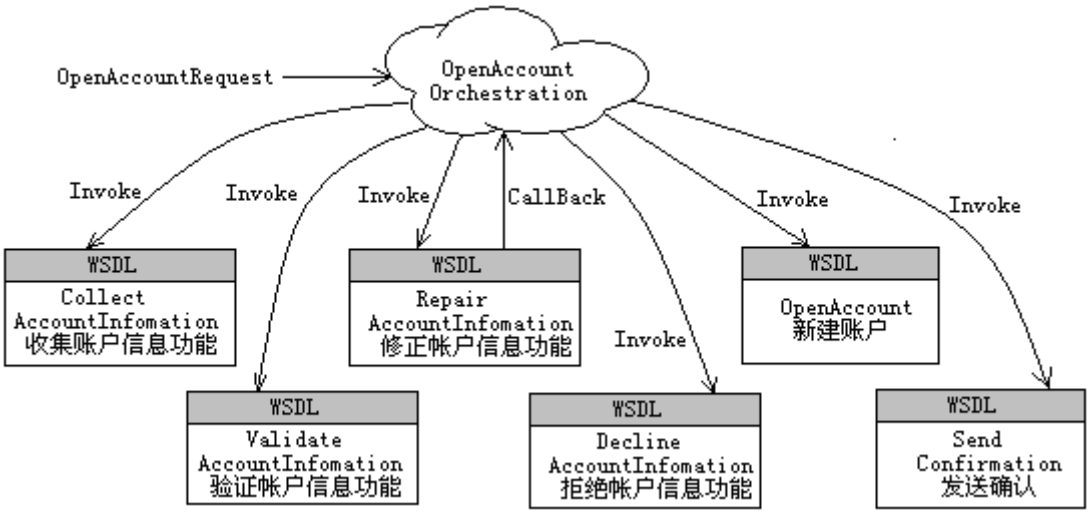
所谓以编制为中心（orchestration-centric）的服务合成，是因为它使用自上而下的方式定义整个过程，其中位于顶层的是一个编制，编制中的各个任务（task），要么是一个 Web 服务，要么是一个被顶层编制调用的子编制（sub-orchestration）。在子编制中的各个任务，要么是一个 Web 服务，要么是一个被子编制调用的编制，以此类推。

下面的例子显示了一个用于“新建账户”（OpenAccount）过程的 WS-BPEL 伪码，这个伪码使用了 WS-BPEL 关键字，但没有采用 XML 格式，而只是为了显示 Web 服务编制的逻辑。

```
receive 'OpenAccountRequest'
```

```
invoke    CollectAccountInfo
invoke ValidateAccountInfo
assign AccountInfoInvalid=ValiDateAccountInfoResponse
while AccountInfoInvalid=true
        invoke RepairAccountInfo
        pick onRepairAccountInfoCB
            invoke ValidateAccountInfo
            assign AccountInfoInvalid=AccountInfoResponse
          otherwise //timeout-assume AccountInfo can't be repaired
            invoke DeclineAccountApplication
        terminate
        end pick
end while
invoke OpenAccount
invoke SendConfirmation
```

下图显示了一个 OpenAccount 过程是如何编制一系列 Web 服务请求执行的。



这里，OpenAccount 过程是作为一个 WS-BPEL 编制实现的，这个 WS-BPEL 编制：

- 在收到 OpenAccountRequest 时启动。
- 调用 OpenAccount、CollectAccountInfomation 等 Web 服务来完成编制中的各个步骤。
- 为 OpenAccount 过程定义业务逻辑，包括控制逻辑（例如控制处理无效账户信息的循环）和数据流。

虽然图中没有显示出循环机条件处理等构词，但实际上它们是存在的（它们是由编制控制点，而不是在各个服务内部处理的）。

4，Web 服务编排描述语言

W3C Web 服务编排工作组（Web Service Choregraphy Working Group）正在制定 Web 服务编排描述语言（WS-CDL）。WS-CDL 是作为 WS-BPEL 等补充 Web 服务技术的补充而提出的，它定义了实现业务编排或者 B2B 场景所需的可执行过程。

常见的 B2B 场景，如 RosettaNet 的合作伙伴信息流程（Partner Information Process，PIP）涉及到一个贸易方向另一个或者多个贸易方提交 XML 文档（例如订单）供其执行。WS-BPEL 可以定义执行流、执行流中的消息及其它活动（比如错误处理程序）。WS-BPEL 的抽象流程

是用于 B2B 交互的，而 WS-CDL 提供了更多的能力，比如不同的业务流程引擎之间彼此如何对话，比方说，一方是 WS-BPEL 引擎，另一方是 RosettaNet 引擎，反之亦然。

WS-CDL 是从“全局的”观点为消息交换定义了公共的排序条件与约束，各个参与者可以利用这个“全局的”定义来构建并测试解决方案。WS-CDL 不是一种可执行的语言，而是一种定义交互模式的说明性语言，参与各方都可以把这个交互模式作为协定。

WS-CDL 文档是一组可以被各方引用的、具名的定义集和，它的根元素是 package 元素，其中包括一个或者多个协作类型定义，package 构词的语法如下：

```
<package
    name="SupplyChain"
    author="Ericn"
    version="1.1"
    targetNamespace=www.iona.com/artix/example
    xmlns="http://www.w3.org/2004/04/ws-chor/cdl">
    importDefinitions*
    infomationType*
    token*
    tokenLocator*
    role*
    relationship*
    participant*
    channelType*
    choreography-Notation*
</package>
```

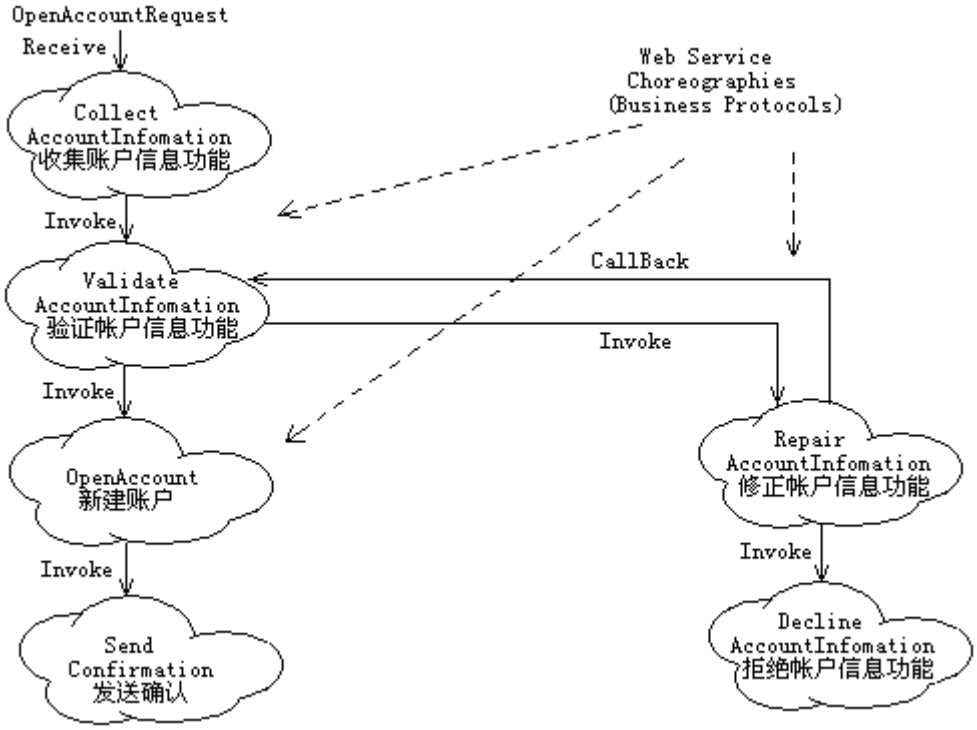
Package 模型定义了协作的参与者、以及各个参与者的角色和相互关系，一旦达成协定，package 就跨越个协作机构的信任边界进行交换，以共享明确的定义。

可以在 package 构词中汇集一些编排定义，其中的 infomationType、token、tokenLocator、role、relationship、participant、channelType 等元素被重用于当前 package 中定义的所有编排。

4，以编排为中心的服务合成案例

以编排为中心（choreography-centric）指的是通过一系列编排，把所有顶层业务流程中的任务连接起来，而没有某个处于控制地位的流程执行引擎。在这里，交互是对等的，而不是像以编制为中心的方法那样用命令来控制。

下图展示了以编排为中心的 OpenAccount 流程



例如，下面这些任务对之间的交互是作为编排定义的：

CollectAccountInfomation 与 ValidateAccountInfomation

ValidateAccountInfomation 与 RepairAccountInfomation

RepairAccountInfomation 与 ValidateAccountInfomation

OpenAccount 与 SendConfirmation

RepairAccountInfomation 与 DeclineAccountInfomation

从某种意义上说，业务流程的编排描述体现了 Web 服务的大粒度要求，对于某些大粒度、分布式的业务重用还是很有意义的。

6.4 SOA 的业务效益与构建

一、SOA 的业务效益

SOA 所描述特征的服务，将具有如下业务效益。

1，增强业务的机动性

增强业务的机动性，是到目前为止 SOA 最重要的业务效益。目前对许多机构而言，对新业务需求与快速响应的业务机动性，是比开发效率还要重要的。业务机动性两个关键要素是速率（velocity）和灵活性（flexibility）。

速率（velocity）：指的是沿着既定的路线快速前进，更快的产品或者服务的上市速度。SOA 显著降低了利用现有服务和 IT 资产组装新业务应用所需的时间，因而提高了速率。

灵活性（flexibility）：根据需要适应 IT 系统的能力。由于不断变化是业务和软件所必须面对的现实，而且也是开销的主要源头，因此，在 IT 可以迅速修改现有系统的情况下，业务可以快速适应新的机遇与竞争威胁。

2，更好的配合业务

当所有的业务都为共同的目标和结果提供支持的话，我们就称之为配合（alignment）。我们可以而且应该把 IT 系统通过 SOA 提供的服务定义为直接支持组织向顾客、客户、公民与合作伙伴等提供的服务。

用面向服务的架构做到业务与 IT 的相互配合，可以改善业务的设计与开发，这是通过业务用户与 IT 技术的要求沟通更加流畅做到的，这也有助于把交流提升到业务层面。

3，改善客户满意度

许多机构都致力于建立一种在不同的服务渠道（面对面、Web 自助服务、移动用户、呼叫中心、ATM 等）一致的用户体验，如果客户从不同的渠道获得自相矛盾的信息，客户满意度就会下降。

以客户为中心的 SOA 致力于确保一致的用户体验，通过创建与任何具体技术和最终设备无关的服务来实现，将更加容易重用于各种服务渠道。

4，降低对厂商的依赖和降低转换成本

传统的 IT 系统中，对厂商技术的依赖发生于各个层面上：

- 应用平台（如 J2EE、.NET 框架、Oracle、 CICS）；
- 套装应用软件（如 SAP、PeopleSoft 等）；
- 中间件技术（如 WebSphere MQ）；
- 特定产品功能（如存储过程、群集缓存）。

我们应该注意到，如果中断与套装应用软件、开发平台、中间件系统的长期关系，是需要付出很大代价的。

SOA 为机构提供了发展空间以适应未来的发展，并显著降低了对厂商技术的依赖。因为以 SOA 为中心的机构是基于服务契约来构建下层 IT 架构的，该服务契约与业务服务层是一致的，并且技术中立、与应用无关和不了解中间件的。这种层次结构更容易替换应用程序、技术和中间件。

5，降低集成成本

SOA 能显著降低集成成本，其原因已经在前面讨论过。

在采用不同的套装应用程序和应用程序的异构环境中，这种成本的降低尤其显著，因为 SOA 提供了一种统一的、一致的技术基础设施，不必为定制集成编写代码，也不用部署和配置许多特定用途的应用程序适配器。

6，提高现有的 IT 资产投资回报率

面向服务的架构能显著提高现有 IT 资产的投资回报率，因为该架构的 IT 资产被重用为服务，确定现有系统的关键业务能力，然后把它们作为构建新服务的基础，这样，SOA 有助于最大化现有 IT 投入的价值，并降低风险。

但是要注意到不是所有的 IT 资产都能够被重用，所以需要一个评估和筛选的过程，这个过程必须特别注意抽象接口的定义，这样的接口应该能够既体现业务功能的本质，又封装了技术细节。

二、如何达成 SOA

1，SOA 业务架构的达成

构建一个合理的 SOA 应采用何种开发方法？从前面的部分可以看出，有业务流程、应用程序和服务。显然，对服务建模是此类方法必须支持的主要任务。另一个重要的方面是确保业务流程和服务之间的链接。

我们必须搞清楚现有的模型（例如面向对象的分析和设计、企业架构框架和业务流程建模技术）对 SOA 设计的作用。我们需要将其它方法元素用于 SOA，例如用于服务标识和聚合的方法和技术、业务跟踪能力、现有资产的集成和重用。我们还需要进行服务的建模，它是在域分解、现有系统分析和目标服务建模之类的技术支持下实现的。

事实上 SOA 并不仅仅是一个 IT 概念，而是偏重于组织、管理以及商业模式。部署 SOA 不仅牵涉到 IT 系统的构建模式，同时也涉及到业务流程架构和业务的管理运作模式，因为

SOA 必须对业务的改变作出迅速反应。达成 SOA 业务需要经过以下几个阶段：

1）行业业务模型和标准的跟踪、学习：要学习行业业务模型和标准，从做咨询顾问开始我们的 IT 项目。

2）融会贯通业务服务化过程：与行业业务专家一起，从划分不同的业务功能域开始，界定项目范围中都包含那些功能域，每个功能域都有哪些过程组成，注意梳理和分解，整理出可重用的业务服务，要注意只有重用才提炼出业务服务、进行业务化处理。

3）完成 SOA 项目是个渐进的过程：在进行整体规划的时候，千万不要与技术层面的 Web 服务、对象、组件混淆起来，和业务人员交流只考虑业务如何组织、流程整合哪些部门和合作伙伴，业务在哪些方面会重用等问题，此时一定要屏蔽实现细节，专注于粗粒度、松耦合的业务。SOA 项目是个渐进的过程，并不需要一蹴而就，这就需要对业务的演进有比较深刻的理解。

4）自上而下形成业务服务的制品：业务服务化的过程是自上而下的，从功能域、业务流程到业务服务。一般来说，业务服务是稳定的、不易变化的、可重用的业务逻辑部分，是行业业务模式的共性所在。而业务流程是善变的、随着业务发展而不断变化的，它是企业真正的核心竞争力的所在。企业业务服务化的过程，就是把变化的部分与稳定的部分区别开来，过程控制变化的部分，业务服务概括稳定的业务模型部分。

2，SOA 需要解决的问题

作为一个具有发展前景的应用系统架构，SOA 尚处在不断发展中，肯定存在许多有待改进的地方。随着标准和实施技术的不断完善，这些问题将迎刃而解，SOA 应用将更加广泛。一般认为，SOA 还是有一定的缺憾的：

1）可靠性(Reliability）

SOA 还没有完全为事务的最高可靠性包括不可否认性(nonrepudiation)、消息一定会被传送且仅传送一次（once-and-only-once delivery）以及事务撤回（rollback）等做好准备，不过等标准和实施技术成熟到可以满足这一需求的程度并不遥远。

2）安全性（Security）

在过去，访问控制只需要登录和验证；而在 SOA 环境中，由于一个应用软件的组件很容易去与属于不同域的其他组件进行对话，所以确保迥然不同又相互连接的系统之间的安全性就复杂得多了。

3）编排 (Orchestration)

统一协调分布式软件组件以便构建有意义的业务流程是最复杂的，但它同时也最适合面向服务类型的集成，原因很显然，建立在 SOA 上面的应用软件被设计成可以按需要拆散、重新组装的服务。作为目前业务流程管理（BPM）解决方案的核心，编排功能使 IT 管理人员能够通过已经部署的套装或自己开发的应用软件的功能，把新的元应用软件（meta-application）连接起来。 事实上，最大的难题不是建立模块化的应用软件，而是改变这些系统表示所处理数据的方法。

4）遗留系统处理（Legacy support）

SOA 中提供集成遗留系统的适配器， 遗留应用适配器屏蔽了许多专用性 API 的复杂性和晦涩性。一个设计良好的适配器的作用好比是一个设计良好的 SOA 服务：它提供了一个抽象层，把应用基础设施的其余部分与各种棘手问题隔离开来。一些厂商就专门把遗留应用软件“语义集成”到基于 XML 的集成构架中。 但是集成遗留系统的工作始终是一种挑战。

5）语义（Semantics）

定义事务和数据的业务含义，一直是 IT 管理人员面临的最棘手的问题。语义关系是设计良好 SOA 架构的核心要素。 就目前而言，没有哪一项技术或软件产品能够真正解决语义问题。为针对特定行业和功能的流程定义并实施功能和数据模型是一项繁重的任务，它最终必

须由业务和 IT 管理人员共同承担。不过，预制组件和经过实践证明的咨询技能可以简化许多难题。

采用 XML 技术也许是一个不错的主意。许多公司越来越认识到制定本行业 XML 标准的重要性。譬如，会计行业已提议用可扩展业务报告语言（XBRL）来描述及审查总账类型的记录。

重要的是学会如何以服务来表示基本的业务流程。改变开发方式需要文化变迁，相比之下，解决技术难题只是一种智力操练。

6）性能（performance）：

这是 SOA 的第六个缺憾吗？批评 SOA 的人士经常会提到性能是阻碍其采用的一个障碍，但技术的标准化总需要在速度方面有一些牺牲。这种怀疑观点通常针对两个方面：SOA 的分布性质和 Web 服务协议的开销。

不可否认，任何分布式系统的执行速度都不如独立式系统，这完全是因为网络的制约作用造成的。当然，有些应用软件无法容忍网络引起的延迟，例如那些对实时性要求很高的应用软件。所以在应用 SOA 架构之前，搞清楚它的适用范围就显得很重要了。

除了上述几点之外，我们认为还有几点也颇值得关注：

7）松耦合和敏捷性要求之间的权衡难题：

服务松耦合设计其实是一把双刃剑，在带来应变敏捷性的同时，也给业务建模和服务划分带来难题。这就是为什么在 SOA 讨论中，业务建模的争论总是最多的原因。

8）跨系统集成难题：

面向服务的架构设计将跨越计算机系统，并且还可能跨越企业边界。我们不得不考虑在使用 Internet 时安全性功能和需求，以及如何链接伙伴的安全域。Internet 协议并不是为可靠性（有保证的提交和提交的顺序）而设计的，但是我们需要确保消息被提交并被处理一次。当这不可能时，请求者必须知道请求并没有被处理。

. **9）SOA 与网格计算（Grid Computing）的关系：**

网格计算（Grid Computing）是利用互联网技术，把分散在不同地理位置的计算机组成一台虚拟超级计算机。每一台参与的计算机就是其中的一个“节点”，所有的计算机就组成了一张节点网——网格。从实质上来说“网格计算”是一种分布式应用，网格中的每一台计算机只是完成工作的一个小部分，虽然单台计算机的运算能力有限，但成千上万台计算机组合起来的计算能力就可以和超级计算机相比了。

网格计算基于因特网，提供了资源整合和共享的平台。十分适合作为 SOA 架构的实施平台。 我们来具体地看一下：

- **SOA 的构建策略**：创建一个面向服务的计算 SOC（service-based computing）环境；可以用类似于 web services 的技术来设计服务：使用 SOAP 通信机制；采用 XML 数据格式；强调服务的重用和互操作；最大化的应用现有资源；希望有一个类似于网格计算环境的基础平台。
- **网格作为平台的基本特点**：网格被视为一个由各种计算资源组成的统一环境，其管理软件将网格整合成一个完整而协调的透明计算整体；网格是一个虚拟的应用服务器；是一个应用实现和数据处理的理想平台；服务在网格中部署和调用执行；商业逻辑和服务调用被当成网格程序一样在平台上运行；网格为 SOC 计算的有效性、快速性、灵活性、伸缩性和计算环境的管理提供便利。

3，SOA 带给企业什么？

作为需要构建 SOA 应用的企业来说，究竟有些什么好处呢？我们来看一下：

- **集成现有系统，不必另起炉灶**：面向服务的架构可以基于现有的系统投资来发展，而不需要彻底重新创建系统。通过使用适当的 SOA 框架并使其用于整个企业，可

以将业务服务构造成现有组件的集合。使用这种新的服务只需要知道它的接口和名称。服务的内部细节以及在组成服务的组件之间传送的数据的复杂性都对外界隐藏了。这种组件的匿名性使组织能够利用现有的投资，从而可以通过合并构建在不同的机器上、运行在不同的操作系统中、用不同的编程语言开发的组件来创建服务。遗留系统可以通过 Web 服务接口来封装和访问。

- **服务设计松耦合，带来多方面优点**：服务是位置透明的，服务不必与特定的系统和特定的网络相连接。服务是协议独立的，服务间的通信框架使得服务重用成为可能。对于业务需求变化，SOA 能够方便组合松耦合的服务，以提供更为优质和快速的响应，允许服务使用者自动发现和连接可用的服务。松耦合系统架构使得服务更容易被应用所集成，或组成其他服务，同时提供了良好的应用开发、运行时服务部属和服务管理能力。提供对服务使用者的验证（authenticatio）、授权（authorization），来加强安全性保障，这一点也优于其他紧耦合架构。
- **统一了业务架构，可扩展性增强**：在所有不同的企业应用程序之间，基础架构的开发和部署将变得更加一致。现有的组件、新开发的组件和从厂商购买的组件可以合并在一个定义良好的 SOA 框架内。这样的组件集合将被作为服务部署在现有的基础构架中，从而使得可以更多地将基础架构作为一种商品化元素来加以考虑，增强了可扩展性。又由于面向服务的敏捷设计，在应对业务变更时，有了更强的“容变性”。
- **加快了开发速度，减少了开发成本**：组织的 Web 服务库将成为采用 SOA 框架的组织的核心资产。使用这些 Web 服务库来构建和部署服务将显著地加快产品的上市速度，因为对现有服务和组件的新的创造性重用缩短了设计、开发、测试和部署产品的时间。 SOA 减少了开发成本，提高了开发人员的工作效率。

市场和制度的不断变化，要求组织具有相当的灵活性，软件架构师提供一种通用的、基于服务的解决方案，将有助于实现组织的灵活性，也更容易实现生产力的提高。尤其在为业务流程管理（BPM）奠定了 SOA 架构基础以后，企业就可以把思想集中于更高层次的问题，比如设计更好的业务流程，而不是考虑实现业务流程的技术细节。

第七章 软件架构设计的其它有关问题

项目成功的关键要素是合理的项目规划，而好的项目规划与软件架构设计具有很强的相关性。

7.1 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庭径。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，或者是用文档来规定约束，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的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中的类而言，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的。但是当一个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。

7.2 进行多维度小组的项目规划

敏捷小组一般不会超过 7～10 人，这种规模的小组可以完成很多事情。但某些项目希望用更大的小组来承担项目的时候，就需要建立更多的比较小的小组，在第九章已经讨论了类似的问题，给出了一些基本概念，我们也提到了在这种情况下，上层实行结构化管理某种意义上说是合理的。下面我们针对这种类型的组织敏捷项目规划的有关问题，做更深入的研究。

规划一个大型的、多小组的项目，可能需要下面的方法：

- 为估计建立共同的基准。

- 更早给用户描述添加细节。
- 进行前瞻规划。
- 在计划中加入馈送缓冲区

根据项目中有多少子小组，以及协调的频繁程度和紧张程度不同，项目可能需要上面这些方法的一部分或者全部，一般还是建议采用上面的顺序进行规划。下面我们详细介绍这些方法。

一、为估计建立共同基准

在项目开始的时候，所有小组应该在一起他们的估计值建立共同的基础，这样一个小组给出的估计值就会和其它小组同样的工作给出的估计值差不多。每个用户描述只要被一个小组估计，那无论是哪个小组来估计，给出的值应该是基本一样的。建立基准有两种方法：

第一种方法，如果小组在过去某个项目一起工作过，他们可以从过去的项目中选择一些用户描述，然后对它进行估计达成一致。他们可以确定 20 来个旧描述，根据现在对这些描述的了解，就没个描述的新估计达成一致。一旦在这些基准描述上达成了一致，各小组就可以把他们各自的用户描述与这些基准描述进行比较来估计他们，这是一种共同基准的类比方法。

第二种方法，使小组在一起估计一些新的用户描述。要选择多种新发布计划的用户描述，这些描述应该覆盖不同的规模，而且选择大多数估计者都与之有关的领域，整个组或者小组代表在一起就这些描述的估计达成一致，然后就可以与第一种方法一样成为基准了。

即使各个小组相互独立，人员也不会流动，我还是建议使用共同基准，因为这有利于进行项目有关的交流。

二、尽早给用户描述添加细节

理论上，敏捷开发小组在开始一次迭代的时候，只有模糊定义的需求，他们在迭代结束之前把这些模糊定义的需求转化为可以工作的意境测试的软件。但是，多小组开发的项目中，在迭代开始之前就对用户描述进行更多的思考常常是适当的也是必要的做法，这些细节可以让不同的小组更容易协调他们的工作。

这种对用户描述更详细的思考，一般是把用户描述（user story）转化为用例（user case）。在建立用例的时候，需要写出简短的用例场景（事件流）。显然，要达到这个目的，更大的组就需要专职的分析员、架构师、用户交互设计师和其他人员，他们在特定的迭代中花一部分时间来准备下一次迭代的工作。一般来说，我不推荐让这些人提前准备整整一次迭代的全部开始工作。他们的主要责任还在当前这次迭代的工作，但也应该包含准备下一次迭代的有关任务。

我发现这些工作最有用的成果，是认定了产品所有者对很可能在下一次迭代中完成的用户描述的满意条件，产品所有者通过用例场景，了解了对这个描述进行高层次可接受性测试的细节，从而明确了满意条件。同时，各个小组对协调方的信息了解的更加详尽，有助于协调一致的工作。

这种处理方式虽然有益，但小组不可能、也不需要在一次迭代以前认定所有用户描述的满意条件，详细的用例场景主要用于对相互协调有关系的那些描述。因为一次迭代到底处理哪些用户描述，在迭代规划会议之前是不可能精确知道的。不过，产品所有者和开发小组可以合理猜测哪些描述最可能在下一次迭代中优先处理，那些描述对小组间的协调最有意义。如果这样的猜测都无法作出，那这样多小组协调的敏捷开发最后陷于混乱，也就是自然而然的事情了。

还要说明，不论是不是多小组开发，在一次迭代开始的时候，首先要做的就是把用户描述转换为用例场景。这里只不过强调的是有些重要的场景要在迭代规划会议之前进行，这样才有利于协调。

三、进行前瞻规划

在发布规划和迭代规划中维持一个滚动的前瞻窗口，可以让大多数具有中等复杂度或者中等频繁度的相互依赖的小组从中受益，下面我们通过一个案例说明这个方法。

案例背景：

假定两个开发小组处理运动员网站 SwimStats，SwimStats 的一部分显示诸如训练成绩、游泳池的地址和方位等静态信息。不过 SwimStats 也必须提供数据库驱动的动态信息，包括过去 15 年所有比赛的成绩和所有运动员所在项目上的个人纪录等。

国家和年龄组纪录存放在国家游泳联合会远程设备上的数据库中。访问这个数据库不想小组当初所希望的那么简单。国家游泳联合会准备在未来 2 年中改换数据库供应商。正是由于这个原因，产品所有者和开发小组都同意要开发一个 API 来访问这个数据库。这样可以让将来改变数据库供应商的工作更容易。

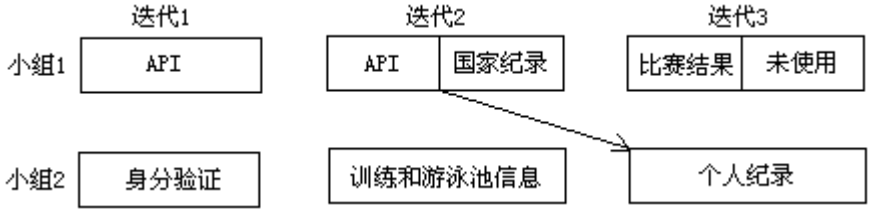
处理办法：

首先，我们列出了初始的用户描述和对它的估计值，如下表所示。

用户描述	描述点
作为 SwimStats，我们希望能够容易的改变数据库供应商	30
作为站点的任意访问者，我在被允许访问敏感内容前，需要进行身份验证	20
作为运动员，我想查看训练活动安排在什么时候	10
作为运动员或者家长，我想知道举行联赛的游泳池在什么地方	10
作为任意访问者，我想按照年龄组或者项目查看国家纪录	10
作为任意访问者，我希望能查看任何比赛的结果	10
作为任意访问者，我希望能看到任何运动员的个人纪录	20

估计的速度是每个小组每次迭代 20 点。这里设定了 2 个小组共同完成这项工作，由于有 110 点的工作，这意味着能够在 3 次迭代中交付所有功能。

但是，开发 API 有 30 点，表中最后 3 个描述需要这个 API 支持，这 3 个描述总共 40 点。必须在 API 完成之后进行。所以，在迭代开始前的规划中，把两个小组的工作安排如下图。

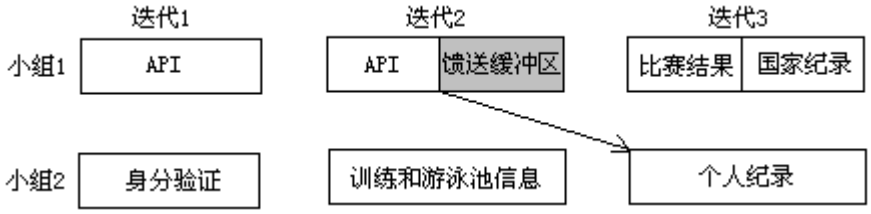


我们曾经建议发布计划只显示最近两次迭代的细节，因为这样做就足以为很多小组遇到的相互依赖提供支持，当然先是迭代的确切次数，取决于小组之间依赖的频度和显著程度。在完成迭代以后，就从发布计划中去掉有关它的细节，然后再前瞻性的安排随后的 2～3 次迭代的期望。由于这总是描绘对新的几次迭代的期望，所以称之为滚动前瞻规划（rolling lookahead plan），又称之为向前窥视。

上图还显示规划了一次小组交接，这比迭代期间任意的提供交接更安全。在第 3 次迭代的时候，第 2 小组知道 API 已经完成，他就可以做出有意义的承诺。

四、在计划中加入馈送缓冲区

但是上面的例子还有一个问题，也就是在第二次迭代的中间必须交付 API。当另一组需要上午 10 点必须使用您的产品的时候，您不可能 9 点 58 分匆匆地把这件工作做完。例如需要书写交接文档、向使用方作详细说明、甚至还要听使用方的一些意见与改进。所以，在工作量的安排上，应该比仅仅开发要多一些，这就需要加上馈送缓冲区。我们需要改变发布规划如下图所示。



1，缓冲的对象

这个例子增加馈送缓冲区并不会增加项目时间，因为时间本来就不满。但是在大多数情况下，增加馈送缓冲区会延长项目时间。这是可以理解的，在多小组并行开发的情况下，协调的成本确实是要增加的。

要确定什么时候要使用馈送缓冲区，首先必须寻找迭代小组之间的依赖，只在一些关键依赖上加入馈送缓冲区。还要看接受方的小组有没有能力快速的把高价值的工作交换近来，如果有，也不一定要加入馈送缓冲区。如果一个小组可以只使用另一个小组的部分交付就可以工作了，也不需要馈送缓冲区。

2，确定馈送缓冲区的大小

理论上缓冲区的大小要用前面我们说过的均方根方法来指导计算。不幸的是，大多数小组间的依赖都是由很少的描述或者功能造成的，您通常没有足够的数据来进行这样的计算，所以可以先简单的用相互依赖的描述规模一定的比例来计算，常用的比例是 50%，然后再根据小组的判断来调整。

如果一次馈送缓冲区比一次迭代都长，就要考虑这个缓冲区设置的是不是合理了。这种长缓冲区通常是计划把一大块功能传递给另一个小组造成的。但如下两个原因告诉我们可能不需要那么长的缓冲区。

首先，一个小组到另一个小组的交接，几乎毫无疑问的应该被分割开来，让功能逐渐交付。

其次，小组一旦发现有另一个小组在跟随他们空转，就应该找到分割工作的方法，或者在迭代期间作出其它的调整，而不是耗费掉一个相当大的馈送缓冲区。

很多情况下，让接受小组扮演产品所有者或客户，往往可以使两个小组都找到对他们有用的增量交付程序。如果一定要用一次 迭代那么长的时间设置馈送缓冲区，就应该质疑并且回顾整个交付计划，看看有什么办法缩短把可交付功能从一个小组传递到另一个小组的链条。

如果只有一个小组，甚至 3～4 个大约 7 个人的小组，就不一定需要做本节所说的事情。但是对于大型的、多开发小组的项目，需要在很多个月以前就宣布和承诺最终期限，而且很多大型项目小组之间具有很强的依赖性，面对这样的状况，多花几个小时来做规划是有意义。这样做可以让您在一开始就更有信心地、更准确地估计目标完成日期，还可以对那些容易避开的进度延误提供一些保护。

7.3 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。

7.4 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，需要扎扎上实实的从基础做起，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。

时代呼唤着优秀的系统架构师，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。