架构设计中的方法学

架构设计是一种权衡（trade-off）。一个问题总是有多种的解决方案。而我们要确定唯一的架构设计的解决方案，就意味着我们要在不同的矛盾体之间做出一个权衡。我们在设计的过程总是可以看到很多的矛盾体：开放和整合，一致性和特殊化，稳定性和延展性等等。任何一对矛盾体都源于我们对软件的不同期望。可是，要满足我们希望软件稳定运行的要求，就必然会影响我们对软件易于扩展的期望。我们希望软件简单明了，却增加了我们设计的复杂度。没有一个软件能够满足所有的要求，因为这些要求之间带有天生的互斥性。而我们评价架构设计的好坏的依据，就只能是根据不同要求的轻重缓急，在其间做出权衡的合理性。

目标

我们希望一个好的架构能够：

重用：为了避免重复劳动，为了降低成本，我们希望能够重用之前的代码、之前的设计。重用是我们不断追求的目标之一，但事实上，做到这一点可没有那么容易。在现实中，人们已经在架构重用上做了很多的工作，工作的成果称为框架（Framework），比如说 Windows 的窗口机制、J2EE 平台等。但是在企业商业建模方面，有效的框架还非常的少。

透明：有些时候，我们为了提高效率，把实现的细节隐藏起来，仅把客户需求的接口呈现给客户。这样，具体的实现对客户来说就是透明的。一个具体的例子是我们使用 JSP 的 tag 技术来代替 JSP 的嵌入代码，因为我们的 HTML 界面人员更熟悉 tag 的方式。

延展：我们对延展的渴求源于需求的易变。因此我们需要架构具有一定的延展性，以适应未来可能的变化。可是，如上所说，延展性和稳定性，延展性和简单性都是矛盾的。因此我们需要权衡我们的投入/产出比。以设计出具有适当和延展性的架构。

简明：一个复杂的架构不论是测试还是维护都是困难的。我们希望架构能够在满足目的的情况下尽可能的简单明了。但是简单明了的含义究竟是什么好像并没有一个明确的定义。使用模式能够使设计变得简单，但这是建立在我熟悉设计模式的基础上。对于一个并不懂设计模式的人，他会认为这个架构很复杂。对于这种情况，我只能对他说，去看看设计模式。

高效：不论是什么系统，我们都希望架构是高效的。这一点对于一些特定的系统来说尤其重要。例如实时系统、高访问量的网站。这些值的是技术上的高效，有时候我们指的高效是效益上的高效。例如，一个只有几十到一百访问量的信息系统，是不是有必要使用 EJB 技术，这就需要我们综合的评估效益了。

安全：安全并不是我们文章讨论的重点，却是架构的一个很重要的方面。

规则

为了达到上述的目的，我们通常需要对架构设计制定一些简单的规则：

功能分解

顾名思义，就是把功能分解开来。为什么呢？我们之所以很难达到重用目标就是因为我们编写的程序经常处于一种好像是重复的功能，但又有轻微差别的状态中。我们很多时候就会经不住诱惑，用拷贝粘贴再做少量修改的方式完成一个功能。这种行为在 XP 中是坚决不被允许的。XP 提倡"Onceandonlyonce"，目的就是为了杜绝这种拷贝修改的现象。为了做到这一点，我们通常要把功能分解到细粒度。很多的设计思想都提倡小类，为的就是这个目的。

所以，我们的程序中的类和方法的数目就会大大增长，而每个类和方法的平均代码却会大大的下降。可是，我们怎么知道这个度应该要如何把握呢，关于这个疑问，并没有明确的答案，要看个人的功力和具体的要求，但是一般来说，我们可以用一个简单的动词短语来命

名类或方法的，那就会是比较好的分类方法。

我们使用功能分解的规则，有助于提高重用性，因为我们每个类和方法的精度都提高了。这是符合大自然的原则的，我们研究自然的主要的一个方向就是将物质分解。我们的思路同样可以应用在软件开发上。除了重用性，功能分解还能实现透明的目标，因为我们使用了功能分解的规则之后，每个类都有自己的单独功能，这样，我们对一个类的研究就可以集中在这个类本身，而不用牵涉到过多的类。

根据实际情况决定不同类间的耦合度

虽然我们总是希望类间的耦合度比较低，但是我们必须客观的评价耦合度。系统之间不可能总是松耦合的，那样肯定什么也做不了。而我们决定耦合的程度的依据何在呢？简单的说，就是根据需求的稳定性，来决定耦合的程度。对于稳定性高的需求，不容易发生变化的需求，我们完全可以把各类设计成紧耦合的（我们虽然讨论类之间的耦合度，但其实功能块、模块、包之间的耦合度也是一样的），因为这样可以提高效率，而且我们还可以使用一些更好的技术来提高效率或简化代码，例如 Java 中的内部类技术。可是，如果需求极有可能变化，我们就需要充分的考虑类之间的耦合问题，我们可以想出各种各样的办法来降低耦合程度，但是归纳起来，不外乎增加抽象的层次来隔离不同的类，这个抽象层次可以是具体的类，也可以是接口，或是一组的类（例如 Beans）。我们可以借用 Java 中的一句话来概括降低耦合度的思想："针对接口编程，而不是针对实现编程。

设计不同的耦合度有利于实现透明和延展。对于类的客户（调用者）来说，他不需要知道过多的细节（实现），他只关心他感兴趣的（接口）。这样，目标类对客户来说就是一个黑盒子。如果接口是稳定的，那么，实现再怎么扩展，对客户来说也不会有很大的影响。以前那种牵一发而动全身的问题完全可以缓解甚至避免。

其实，我们仔细的观察 GOF 的 23 种设计模式，没有一种模式的思路不是从增加抽象层次入手来解决问题的。同样，我们去观察 Java 源码的时候，我们也可以发现，Java 源码中存在着大量的抽象层次，初看之下，它们什么都不干，但是它们对系统的设计起着重大的作用。

够用就好

我们在上一章中就谈过敏捷方法很看重刚好够用的问题，现在我们结合架构设计来看：在同样都能够满足需要的情况下，一项复杂的设计和一项简单的设计，哪一个更好。从敏捷的观点来看，一定是后者。因为目前的需求只有 10 项，而你的设计能够满足 100 项的需求，只能说这是种浪费。你在设计时完全没有考虑成本问题，不考虑成本问题，你就是对开发组织的不负责，对客户的不负责。

应用模式

这篇文章的写作思路很多来源于对模式的研究。因此，文章中到处都可以看到模式思想的影子。模式是一种整理、传播思想的非常优秀的途径，我们可以通过模式的方式学习他人的经验。一个好的模式代表了某个问题研究的成果，因此我们把模式应用在架构设计上，能够大大增强架构的稳定性。

抽象

架构的本质在于其抽象性。它包括两个方面的抽象：业务抽象和技术抽象。架构是现实世界的一个模型，所以我们首先需要对现实世界有一个很深的了解，然后我们还要能够熟练的应用技术来实现现实世界到模型的映射。因此，我们在对业务或技术理解不够深入的情况

下，就很难设计出好的架构。当然，这时候我们发现一个问题：怎样才能算是理解足够深入呢。我认为这没有一个绝对的准则。

一次，一位朋友问我：他现在做的系统有很大的变化，原先设计的工作流架构不能满足现在的要求。他很希望能够设计出足够好的工作流架构，以适应不同的变化。但是他发现这样做无异于重新开发一个 lotusnotes。我听了他的疑问之后觉得有两点问题：

首先，他的开发团队中并没有工作流领域的专家。他的客户虽然了解自己的工作流程，但是缺乏足够的理论知识把工作流提到抽象的地步。显然，他本身虽然有技术方面的才能，但就工作流业务本身，他也没有足够的经验。所以，设计出象 notes 那样的系统的前提条件并不存在。

其次，开发一个工作流系统的目的是什么。原先的工作流系统运作的不好，其原因是有变化发生。因此才有改进工作流系统的动机出现。可是，毕竟 notes 是为了满足世界上所有的工作流系统而开发的，他目前的应用肯定达不到这个层次。

因此，虽然做不到最优的业务抽象，但是我们完全可以在特定目的下，特定范围内做到最优的业务抽象。比如说，我们工作流可能的变化是工组流路径的变化。我们就完全可以把工作流的路径做一个抽象，设计一个可以动态改变路径的工作流架构。

有些时候，我们虽然在技术上和业务上都有所欠缺，没有办法设计出好的架构。但是我们完全可以借鉴他人的经验，看看类似的问题别人是如何解决的。这就是我们前面提到的模式。我们不要把模式看成是一个硬性的解决方法，它只是一种解决问题的思路。MartinFowler 曾说："模式和业务组件的区别就在于模式会引发你的思考。"

在《分析模式》一书中，MartinFowler 提到了分析和设计的区别。分析并不仅仅只是用用例列出所有的需求，分析还应该深入到表面需求的的背后，以得到关于问题本质的 MentalModel。然后，他引出了概念模型的概念。概念模型就类似于我们在讨论的抽象。MartinFowler 提到了一个有趣的例子，如果要开发一套软件来模拟桌球游戏，那么，用用例来描述各种的需求，可能会导致大量的运动轨迹的出现。如果你没有了解表面现象之后隐藏的运动定律的本质，你可能永远无法开发出这样一个系统。

关于架构和抽象的问题，在后面的文章中有一个测量模式的案例可以很形象的说明这个问题。

架构的一些误解

我们花了一些篇幅来介绍架构的一些知识。现在回到我们的另一个主题上来。对于一个敏捷开发过程，架构意味着什么，我们该如何面对架构。这里我们首先要澄清一些误解：

误解 1：架构设计需要很强的技术能力。从某种程度来说，这句话并没有很大的错误。毕竟，你的能力越强，设计出优秀架构的几率也会上升。但是能力和架构设计之间并没有一个很强的联系。即使是普通的编程人员，他一样有能力设计出能实现目标的架构。

误解 2：架构由专门的设计师来设计，设计出的蓝图交由程序员来实现。我们之所以会认为架构是设计师的工作，是因为我们喜欢把软件开发和建筑工程做类比。但是，这两者实际上是有着很大的区别的。关键之处在于，建筑设计已经有很长的历史，已经发展出完善的理论，可以通过某些理论（如力学原理）来验证设计蓝图。可是，对软件开发而言，验证架构设计的正确性，只能够通过写代码来验证。因此，很多看似完美的架构，往往在实现时会出现问题。

误解 3：在一开始就要设计出完善的架构。这种方式是最传统的前期设计方式。这也是为 XP 所摒弃的一种设计方式。主要的原因是，在一开始设计出完美的架构根本就是在自欺欺人。因为这样做的基本假设就是需求的不变性。但需求是没有不变的（关于需求的细节讨

论，请参看拙作『需求的实践』)。这样做的坏处是，我们一开始就限制了整个的软件的形状。而到实现时，我们虽然发现原来的设计有失误之处，但却不愿意面对现实。这使得软件畸形的生长。原本一些简单的问题，却因为别扭的架构，变得非常的复杂。这种例子我们经常可以看到，例如为兼容前个版本而导致的软件复杂性。而 2000 年问题，TCP/IP 网络的安全性问题也从一个侧面反映了这个问题的严重性。

误解 4：架构蓝图交给程序员之后，架构设计师的任务就完成了。和误解 2 一样，我们借鉴了建筑工程的经验。我们看到建筑设计师把设计好的蓝图交给施工人员，施工人员就会按照图纸建造出和图纸一模一样的大厦。于是，我们也企图在软件开发中使用这种模式。这是非常要命的。软件开发中缺乏一种通用的语言，能够充分的消除设计师和程序员的沟通隔阂。有人说，UML 不可以吗？UML 的设计理念是好的，可以减轻沟通障碍问题。可是要想完全解决这个问题，UML 还做不到。首先，程序员都具有个性化的思维，他会以自己的思维方式去理解设计，因为从设计到实现并不是一项机械的劳动，还是属于一项知识性的劳动（这和施工人员的工作是不同的)。此外，对于程序员来说，他还极有可能按照自己的想法对设计图进行一定的修改，这是非常正常的一项举动。更糟的是，程序员往往都比较自负，他们会潜意识的排斥那些未经过自己认同的设计。

架构设计的过程模式

通常我们认为模式都是用在软件开发、架构设计上的。其实，这只是模式的一个方面。模式的定义告诉我们，模式描述了一个特定环境的解决方法，这个特定环境往往重复出现，制定出一个较好的解决方法有利于我们在未来能有效的解决类似的问题。其实，在管理学上，也存在这种类似的这种思维。称为结构性问题的程序化解决方法。所以呢，我们完全可以把模式的思想用在其它的方面，而目前最佳的运用就是过程模式和组织模式。在我们的文章中，我们仅限于讨论过程模式。

我们讨论的过程仅限于面向对象的软件开发过程。我们称之为 OOSP（object-orientedsoftwareprocess)。因为我们的过程需要面向对象特性的支持。当然，我们的很多做法一样可以用在非 OO 的开发过程中，但是为了达到最佳的效果，我建议您使用 OO 技术。