设计已死？

作者：**Martin Fowler** 译者：**Daimler Huang**

Planned and Evolutionary Design

我将在这篇文章中说明软件开发的两种设计方式是如何完成的。或许最常见的是演进式设计。它的本质是系统的设计随着软件开发的过程增长。设计 (design) 是撰写程序代码过程的一部份，随着程序代码的发展，设计也跟着调整。

在常见的使用中，演进式设计实在是彻底的失败。设计的结果其实是一堆为了某些特殊条件而巧妙安排的决定所组成，每个条件都会让程序代码更难修改。从很多方面来看，你可能会批评这样根本就没有设计可言，无疑地这样的方式常会导致很差劲的设计。根据 Kent的陈述，所谓的设计 (design) 是要能够让你可以长期很简单地修改软件。当设计 (design) 不如预期时，你应该能够做有效的更改。一段时间之后，设计变得越来越糟，你也体会到这个软件混乱的程度。这样的情形不仅使得软件本身难以修改，也容易产生难以追踪和彻底解决的 bug。随着计画的进行，bug 的数量呈指数地成长而必须花更多成本去解决，这就是 "code and fix" 的恶梦。

Planned Design 的做法正好相反，并且含有取自其它工程的概念。如果你打算做一间狗屋，你只需要找齐木料以及在心中有一个大略的形象。但是如果你想要建一栋摩天大楼，照同样的做法，恐怕还不到一半的高度大楼就垮了。于是你先在一间像我太太在波士顿市区那样的办公室里完成工程图。她在设计图中确定所有的细节，一部份使用数学分析，但是大部分都是使用建筑规范。所谓的建筑规范就是根据成功的经验 (有些是数学分析) 制定出如何设计结构体的法则。当设计图完成，她们公司就可以将设计图交给另一个施工的公司按图施工。

Planned Design 将同样的方式应用在软件开发。Designer 先定出重要的部份，程序代码不是由他们来撰写，因为软件并不是他们 "建造[译注3]" 的，他们只负责设计。所以 designer 可以利用像 UML 这样的技术，不需要太注重撰写程序代码的细节问题，而在一个比较属于抽象的层次上工作。一旦设计的部份完成了，他们就可以将它交给另一个团队 (或甚至是另一家公司) 去 "建造"。因为 designer 朝着大方向思考，所以他们能够避免因为策略方面不断的更改而导致软件的失序。Programmer 就可以依循设计好的方向 (如果有遵循设计) 写出好的系统。

Planned design 方法从七○年代出现，而且很多人都用过它了。在很多方面它比 code and fix 渐进式设计要来的好，但是它也有一些缺点存在。第一个缺点是当你在撰写程序代码时，你不可能同时把所有必须处理的问题都想清楚。所以将无可避免的遇到一些让人对原先设计产生质疑的问题。可是如果 designer 在完成工作之后就转移到其它项目，那怎么办？Programmer 开始迁就设计来写程序，于是软件开始趋于混乱。就算找到 designer，花时间整理设计，变更设计图，然后修改程序代码。但是必须面临更短的时程以及更大的压力来修改问题，又是混乱的开端。

此外，通常还有软件开发文化方面的问题。Designer 因为专精的技术和丰富的经验而成为一位 designer。然而，他们忙于从事设计而没有时间写程序代码。但是，开发软件的

工具发展迅速，当你不再撰写程序代码时，你不只是错失了技术潮流所发生的改变，同时也失去了对于那些实际撰写程序代码的人的尊敬。
建造者 (builder[译注3]) 和设计者之间这种微妙的关系在建筑界也看得到，只是在软件界更加凸显而已。之所以会如此强烈是因为一个关键性的差异。在建筑界，设计师和工程师的技术有清楚的分野；在软件界就比较分不清楚了[译注2]。任何在高度注重 design 的环境工作的 programmer 都必须具备良好的技术，他的能力足够对 designer 的设计提出质疑，尤其是当 designer 对于新的发展工具或平台越来越不熟悉的状况下。
现在这些问题也许可以获得解决。也许我们可以处理人与人之间的互动问题。也许我们可以加强 designer 的技术来处理绝大部份的问题，并且订出一个依照准则去做就足够改变设计图的流程。但是仍然有另外一个问题：变更需求。变更需求是软件项目中最让我感到头痛的问题了。
处理变更需求的方式之一是做有弹性的设计，于是当需求有所更改，你就可以轻易的变更设计。然而，这是需要先见之明去猜测将来你可能会做怎样的变更。一项预留处理易变性质的设计可能对于将来的需求变更有所帮助，但是对于意外的变化却没有帮助 (甚至有害)。所以你必须对于需求有足够的了解以隔离易变的部份。照我的观察，这是非常困难的。
部份有关需求的问题应该归咎于对需求的了解不够清楚，所以有人专注于研究需求处理，希望得到适切的需求以避免后来对设计的修改。但是即使朝这个方向去做一样无法对症下药。很多无法预料的变更起因于瞬息万变的商场，你只有更加小心处理需求问题来应付无法避免的情况。
这么说来，planned design 听起来像是不可能的任务。这种做法当然是一种很大的挑战。但是，跟演进式设计 (evolutionary design) 普遍以 code and fix 方式实作比较起来，我不觉得 planned design 会比较差。事实上，我也比较喜欢 planned design。因为我了解 planned design 的缺点，而且正在寻找更好的方法。

The Enabling Practices of XP

XP 因为许多原因而备受争议，其中之一就是它主张演进式设计 (evolutionary design) 而不是 planned design。我们也知道，演进式设计可能因为特定的设计或是软件开发趋于混乱而行不通。
想了解这些争论的核心，就是软件研发异动曲线。曲线的变化说明，随着项目的进行，变更所需要的成本呈现指数的增加。这样的曲线常以一句话来表示：在分析阶段花一块钱所作的变更，发行之后要花数千元来补救。讽刺的是大部分的计画仍然没有分析过程而以非标准的方式进行，但是这种成本上的指数关系还是存在着。这种指数曲线意味着演进式设计可能行不通，它同时也说明着为什么 planned design 要小心翼翼地规划，因为任何的错误还是会面对同样的问题。
XP 的基本假设是它可以将这种指数曲线拉平，这样演进式设计就行得通了。XP 使曲线更平缓并能运用这种优势。这是因为 XP 实作技巧之间的耦合效果：换句话说，不使用那些能够拉平软件开发曲线的实作技巧来工作，这条曲线也不会趋向平缓。这也是争论的来源，因为评论家不了解这其间的关系。通常这些批评是根据评论家自身的经验，他们并没有实行那些有效的实作技巧，当他们看到结果不如预期，对于 XP 的印象也就是这样了。

这些有效的实作技巧有几个部份，主要是 Testing 和 Continuous Integration。如果没有 testing 提供保障，其它的 XP 实作技巧都不可行。Continuous Integration 可以保持团队成员信息同步，所以当你有改变的部份，不必担心与其它成员资料整合会有问题。同时运用这些实作技巧能够大大影响开发曲线。这让我再次想起在ThoughtWorks 导入 testing 和 continuous integration 之后，明显的改善了研发成果。改善的程度好到令人怀疑是不是像 XP 所主张的，必须要用到所有的实作技巧才能大幅改善效率。[译注4]
Refactoring 具有类似的成效。那些曾经采用 XP 建议的原则来对程序代码进行 refactoring 的人发现，这么做要比无章法或是特殊方式的 restructuring 明显的更有效率。那也曾经是 Kent 指导我适当的 refactor 得到的难忘经验，也因为这么一次巨大的转变促使我以这个主题写了一本书。
Jim Highsmith 写了一篇很棒的文章 "summary of XP"，他把 planned design 和 refactoring 放在天秤的两端。大部份传统的做法假设构想不变，所以 planned design 占优势。而当你的成本越来越不允许变更，你就越倾向于采用 refactoring。Planned design 并不是完全消失，只是将这两种做法互相搭配运用取得平衡。对我来说，在设计进行 refactoring 之前，总觉得这个设计不够健全。
Continuous integration、testing 和 refactoring 这些有效的实作方法让 evolutionary design 看似很有道理。但是我们尚未找出其间的平衡点。我相信，不论外界对 XP 存有什么印象，XP 不仅仅是 testing、coding 和 refactoring。在 coding 之前还有 design 的必要。部份的 design 在 coding 之前准备，大部份的 design 则发生在实作每一项详列的功能之前。总之，在 up-front design 和 refactoring 之间可以找到新的平衡。

The Value of Simplicity

XP 大声疾呼的两个口号是 "Do The Simplest Thing that Could Possibly Work"(只做最简单可以正常运作的设计) 和 "You Aren't Going to Need It"(就是 YAGNI - 你将不会需要它)。两项都是XP实务中简单设计的表现形式。
YAGNI 一词时常被讨论，它的意思是现在不要为了将来可能用到的功能加入任何程序代码。表面上听起来好象很简单，问题则出在像 framework、重用组件、和弹性化设计，这些东西本来就很复杂。你事先付出额外的成本去打造它们，希望稍后将这些花费都赚回来。这个事先弹性设计的想法被认为是软件设计有效率的关键部份。
但XP的建议是，在处理第一个问题时不要因为可能需要某项功能，就建造出弹性的组件组及框架出来。让整体结构随着需要成长。假如我今天想要一个可以处理加法但是不用乘法的 Money 类别，我就只在 Money 类别中建造加法的功能。就算我确定下一个阶段也需要乘法的运算，而且我知道很简单，也花不了多少时间，我还是会留到下一阶段再去做它。
其中一个理由是效益。如果我要花时间在明天才需要的功能，那就表示我没有将精神放在这个阶段应该完成的事情上。发表计画详列目前要完成的事项，现在做以后才需要的事情违背开发人员和顾客之间的协议。这种做法有让现阶段的目标无法达成的可能。而且这个阶段的 stroies[译注5] 是否具有风险，或是需不需要做额外的工作，都是由顾客来决定的 - 还是可能不包括乘法功能。

这种经济效益上的限制是因为我们有可能出错。就算是我们已经确定这个功能应该如何运作，都有可能出错 - 尤其是这时候我们还没有取得详细需求。提前做一件错误的事情比提前做一件对的事情更浪费时间。而且XP专家们通常相信我们比较有可能会做错而不是做对(我心有戚戚)。

第二个支持 simple design 的理由是复杂的设计违反光线行进的原理。复杂的设计比简单的设计还要令人难懂。所以随着渐增的系统复杂度，更加难以对系统做任何修改。如此，若系统必须加入更复杂的设计时，成本势必增加。

现在很多人发现这样的建议是无意义的，其实他们那样想是对的。因为你所想象一般的研发并没有被 XP 有效的技巧所取代。然而，当规划式设计和渐进式设计之间的平衡点有了变化 (也只有当这样的变化发生时)，YAGNI 就会变成好的技巧。

所以结论是，除非到了往后的阶段有所需要，否则你不会浪费精神去增加新的功能。即使不会多花成本，你也不会这样做，因为就算现在加入这些功能并不增加成本，但是却会增加将来做修改时的成本。总之，你可以在套用 XP 时明智的遵守这样的方法，或是采取一种能降低成本的类似的方法。

What on Earth is Simplicity Anyway

因此，我们希望程序代码能够越简单越好，这听起来没什么好争论的，毕竟有谁想要复杂呢？但问题来了，究竟 "什么才叫简单呢？"

在 XPE 一书中，Kent 对简单系统订了四个评量标准，依序是 (最重要排最前面)：

通过所有测试。

呈现所有的意图。

避免重复。

最少数量的类别或方法。

通过所有测试是一项很普通的评量标准，避免重复也很明确，尽管有些研发人员需要别人的指点才能做到。比较麻烦的是 "呈现所有的意图"这一项，这到底指的是什么呢？

这句话的本意就是简单明了的程序代码。XP 对程序代码的易读性有很高的标准。虽然在 XP 当中，"巧妙的程序代码 (clever code)" 这个字眼经常被滥用，不过意图清楚的程序代码，对其他人来说真的是一种巧妙。Josh Kerievsky 在 XP 2000 论文中举了一个很好的例子，检视在 XP 领域可能是大家最熟知的 JUnit 的程序代码。JUnit 使用 decorators 在 test cases 中加入非必要的功能，像是同步机制及批次设定等，将这些程序代码抽出成为 decorator，的确让一般的程序代码看起来清楚许多。

但是你必须扪心自问，这样做之后的程序代码够简单吗？我觉得是，因为我了解 Decorator 这个 patterns。但是对于不了解的人来说还是相当复杂的。类似的情况，JUnit 使用 pluggable method，一种大部分的人刚开始接触时都不会觉得简单的技巧。所以，也许我们可以说 JUnit 对有经验的人来说是比较简单的，新手反而会觉得它很复杂。

XP 的 "Once and Only Once" 以及 Pragmatic Programmer(书名) 的 DRY(Don't Repeat Yourself) 都专注在去除重复的程序代码。这些良好的建议都有很显著而且惊人的效果。只要依照这个方式，项目就可以一路顺利的运作。但是它也不能解决所有问题，简单化还是不容易达成。

最近我参与一个可能是过度设计的项目，系统经过 refactor 之后去除部份弹性的设计。但是就像其中一位开发者所说的 "重构过度设计的系统要比重构没有设计的要来的容易多

了" 做一个比你所需要简单一点的设计是最好的，但是，稍微复杂一点点也不是什么严重的事情。
我听过最好的建议来自 Bob 大叔 (Robert Martin)。他的建议是不要太在意什么是最简单的设计。毕竟后来你可以，应该，也会再重构。愿意在最后重构，比知道如何做简单的设计重要得多。

Does Refactoring Violate YAGNI?

这个主题最近出现在 XP 讨论区上，当我们审视设计在 XP 扮演的角色时，我觉得很值得提出来讨论。
基本上这个问题起因于重构需要耗费时间却没有增加新的功能。而 YAGNI 的观点是假设你为了眼前的需要做设计而不是未来，这样算是互相抵触吗？
YAGNI 的观点是不要增加一些现阶段不需要的复杂功能，这也是简单设计这项技巧的部份精神。重构也是为了满足尽可能保持系统的简单性这个需要，所以当你觉得可以让系统变得更简单的时候，就进行重构。
简单设计不但利用了 XP 的实务技巧，本身也是其中一项有用的实务技巧。唯有伴随着测试，持续整合，及重构的运用，才能有效地做出简单设计。同时，让研发异动曲线保持平缓的基础也就是保持设计的简单。任何不必要的复杂都会让系统变得难于调整，除非这个复杂性是你为了所预测的弹性而加入的。不过，人们的预测通常都不太准确，所以最好还是努力地保持简单性。
不管怎样，人们不太可能第一次就能够获得最简单的东西，因此你需要重构来帮助你更接近这个目标。

Patterns and XP

JUnit 的例子让我不得不想到 patterns。XP 和 patterns 之间的关系很微妙，也常常被问起。Joshua Kerievsky 认为 patterns 在 XP 被过分轻视，而且他所提出的理由也相当令人信服，我不想再重提。不过值得一提的是，很多人都认为 patterns 似乎与 XP 是有冲突的。
争论的本质在于 patterns 常被过度滥用。世上有太多传奇性的 programmer，第一次读到四人帮以 32 行程序代码阐述 16 种 patterns 这样的事情还记忆犹新[译注6]。我还记得有一晚与 Kent 喝着醇酒一起讨论一篇文章 "Not Design patterns: 23 cheap tricks (不要用设计模式－23 个简单的诀窍)"。我们认为那不过是以 if 条件式来取代 strategy 这个 pattern 罢了。这样的笑话有个重点，patterns 被滥用了。但并不表示 patterns 是不足取的，问题在于你要怎么运用它。
其中一项论点是简单设计的力量自然会将项目导向 patterns。很多重构的例子明确地这么做，或者甚至不用重构，你只要遵从简单设计的规则就会发现 patterns，即使你还不知道 patterns 是什么。这样的说法也许是真的，不过它真的是最好的方式吗？当然如果你先对于 patterns 有个大略的了解，或者手边有一本书可以参考，会比自己发明新的 patterns 要好些。当我觉得一个 pattern 快浮现的时候，我必定会去翻翻 GOF 的书。对我来说，有效的设计告诉我们 pattern 值得付出代价去学习－那就是它特有的技术。

同样地就像 Joshua 所建议的，我们需要更熟悉于如何逐步地运用 patterns。就这一点而言，XP 只是与一般使用 patterns 的方式不同而已，并没有抹煞它的价值。
但是从讨论区一些文章看来，我觉得很多人明显地认为 XP 并不鼓励使用 patterns，尽管 XP 大部分的提倡者也都是之前 patterns 运动的领导者。因为他们看到了不同于 patterns 的观点吗？或是他们已经将 patterns 融入思考而不必再去理解它？我不知道其它人的答案是什么，但是对我来说，patterns 仍然是非常重要的。XP 也许是开发的一种流程，但 patterns 可是设计知识的骨干，不管是哪种流程这些知识都是很有用的。不同的流程使用 patterns 的方式也就不同，XP 强调等到需要时才使用 patterns 以及透过简单的实作逐步导入 patterns。所以 patterns 仍然是一种必须获得的关键知识。
我对于采用 XP 的人使用 patterns 的建议：
花点时间学习 patterns。
留意使用 patterns 的时机 (但是别太早)。
留意如何先以最简单的方式使用 patterns，然后再慢慢增加复杂度。
如果用了一种 pattern 却觉得没有多大帮助－不用怕，再次把它去掉。
我认为XP应该要更加强调学习 patterns。我不确定它要怎么和 XP 的实务技巧搭配，不过相信 Kent 会想出办法来的。
Growing an Architecture
软件架构是指什么呢？对我来说，架构这个字眼代表系统核心组件的概念，也就是难以改变的部份，剩下的都必须建造在这基础上。
那么当你使用演进式设计时，架构又扮演着什么样的角色呢？XP 的批评者再一次地声称 XP 忽视架构，因为 XP 使用的方法是尽快地写程序，然后相信重构会解决所有设计的问题。很有趣地，他们说得没错，这有可能是 XP 的缺点。无疑地，最积极的 XP 专家－像 Kent Beck, Ron Jeffries, 及 Bob Martin－尽其所能地避免预先结构性的设计。在你知道真的要用到数据库之前，不要加入数据库，先用档案来代替，在之后的阶段再用重构加入数据库。
我常被认为是一个胆小的 XP 专家，这点我不同意。我认为一个概括性的初始架构有它的用处在。像是一开始要怎么将应用分层，如何与数据库互动 (如果你需要的话)，要使用哪种方式去处理网站服务器。
基本上，我认为这些就是近年来我们所研究的 patterns。尤其当你对 patterns 的认识越深，你就会越熟悉要怎么去善用它们。不过，关键性的差异是在于这些初期架构的决定是可以更改的，只要团队认为他们早期的判断有误时，就应该要有勇气去修正它们。有人跟我讲了一个项目的故事，就在项目快要发表时，决定了不再需要 EJB，并且要将它们从系统中移除。这是一个相当大规模的重构，不过最后还是完成了。这些有效的实务技巧不仅让事情变得可能，而且很值得去做。
如果以不同的方式来做这件事呢？如果你决定不采用 EJB，将来会难以加入吗？你是否要在试过各种方式却发现依然欠缺什么，然后才使用 EJB？这是一个牵涉很多因素的问题。不使用复杂的组件当然可以增加系统的简单度，而且可以让事情进展比较快，但有时候从系统中抽掉某个部份会比加入它要容易多了。
所以我建议从评估架构可能的样子开始。假如你看到将会有多个使用者使用到大量的资料，一开始就直接使用数据库。若你看到很复杂的商业逻辑，就套用 domain model。你

会怀疑是否偏离简单的特性，这当然不是 YAGNI 的精神。所以你要有所准备，在发现所使用的结构没有帮助时尽快简化你的结构。

UML and XP

在我投身于 XP 领域之后，由于我与 UML 的关系让我遇到一个挥之不去最大的问题：这两者不是不兼容吗？
当然有些不兼容。XP 显然不重视 diagram。虽然台面上大家对 "好用就用" 有共识，但是实际上却是 "实际上采用 XP 的人不画蓝图"。这种印象因为如 Kent 这些人不习惯作图的现象而强化了。事实上我也从来没看过 Kent 主动使用固定的标记法画下软件蓝图。
我觉得这种情形来自两个因素，其一是有人觉得软件蓝图有用，而有人不觉得有用。难就难在觉得蓝图有用的人不是真正必须动手做的人，而必须动手做的人却不觉得有其必要性。事实上我们应该接受有人喜欢用，而有人不喜欢用。
另一种情形是软件蓝图常引人进入繁重的流程中，这些流程耗时费力却不见得有用，甚至还会产生坏处。我认为应该教导人们如何使用蓝图却不落入这样的陷阱，而不是像那些提倡者仅仅消极的说 "必要时才用"。
所以，我对于有效使用蓝图的建议是：
首先别忘了你画这些图的目的，主要的价值在于沟通。有效的沟通意味着选择重要的部份而忽略不重要的部份。这样的选择也是有效运用 UML 的关键。不必把全部的 class 都画出来，画出重要的就好。对于每个 class 也只显示关键的 attribute 和 operation，而不是全部显示出来。也不要为所有的 use case 和步骤画循序图... 除非你已经有完整的想象。有一个使用蓝图的通病就是人们通常希望详细完整的把图表现出来。其实程序代码就是提供完整信息的最佳来源，同时程序代码本身也是保持信息同步最简单的方式。因为图形的巨细靡遗就是一目了然的敌人。
蓝图的用途是在开始撰写程序代码之前探讨设计内容。印象中总是觉得这样的动作在 XP 是不合法的，但并不是这样。很多人都说如果你遇到棘手的问题，就值得先做些设计。但是当你进行设计时：
保持简短。
不要做得太详细(只挑重要的做)。
把结果当作是草图，而不是定案。
最后一点值得深入探讨。当你做预先式设计，无可避免的会发现一些错误，而且是在撰写程序代码的时候才发现。如果你适时变更设计，它就不是问题。麻烦的是如果你认定设计已经定案，没有从 coding 过程学到经验而跟着先前的设计将错就错。
变更设计不代表一定要更改蓝图。画这些蓝图来帮助你了解设计，然后就把图扔开，这么做是非常合理的。这些图能够帮上忙就有它的价值了。它们不必永远存在，最有用的 UML 图形也不会是收藏品。
不少实行 XP 的人使用 CRC 卡，这与 UML 并不冲突。我常常交互运用 CRC 卡和 UML，我也总是依照手上的工作选择最有用的技巧。UML 图形的另一个用途是持续修订的文件。它一般的形式，就是在 case tool 中看到的模型。最初的想法是留着这样的资料有助于建构系统。事实上却常常没什么用。
保持图形的更新太花时间，最后常无法与程序代码同步。

它们隐含在 CASE tool 或 thick binder，让人忽略它。
所以要希望这种持续修订的文件有用，就从这些看到的问题下手：
只用一些改起来不至于让人觉得痛苦的图。
把图放在显眼的地方。我喜欢画在墙上，鼓励大家一起动手修改。
检讨这些图是不是有人在用，没用的就擦掉。
使用 UML 的最后一个问题是文件的交接，像是不同团队的接手。XP 的想法是文件就像说故事，所以文件的价值由顾客来决定。于是 UML 又派上用场，所提供的图形可以帮助沟通。别忘了程序代码本身就蕴含了所有详细的信息，图形的作用只是提供概观以及标示重要的部份。

On Metaphor

好吧，我也许该坦承 - 我一直没有抓住 metaphor 的精神。它有用，而且在 C3 项目中运用得很好，但是并不表示我知道怎么用它，更不用说要解释怎么用了。
XP 实务技巧中的 Metaphor 是建立在 Ward Cunningham's 为系统命名的做法上。重点是想出一个众所周知的词汇，以这样一个字来比喻整个范畴。这个代表系统的名字会套用在 class 和 method 的命名上。
我以不同领域的观念性模型，利用 UML 和它的前身与领域专家一起建立了一个命名系统。我发现你必须很小心，你要保持最精简的注释，而且要当心别让技术性的问题不知不觉的影响这个模型。但是一旦你完成这个工作，你就可以为各种领域建立一组词汇，这些词汇是大家都能了解并且可用来与研发人员沟通的。这种模型无法与 class 设计完美的吻合，但是足够给整个领域一个通用的代名词。
目前我找不到任何理由说明为何这样的一个字汇无法成为一个比喻，就像 C3 这个成功的例子；我也不觉得以系统为主找到在该专业领域的一个词汇有什么坏处。同时我也不会放弃可以运作自如的为系统命名的技巧。
人们常批评 XP 乃是基于觉得一个系统实在是至少需要一个大概的设计。XP 专家则以 "就是 metaphor 啊！" 来响应。但是我还是没有看到一个对于 metaphor 令人信服的解释。这是 XP 的缺憾，必须由 XP 专家来理出头绪。

Do you wanna be an Architect when you grow up?

近几年来 "software architect (软件设计师)" 越来越热门，这是一个就我个人而言难以接受的名词。我太太是结构工程师，工程师和建筑师之间的关系是... 有趣的。我最喜欢的一句话是：建筑师对三种 B 是好的，灯泡、灌木丛、和鸟。因为建筑师画出这些美丽的图画，但却要工程师保证能做出来。结论是我避免 software architect 一词，毕竟如果连我的太太都不能尊重我的专业，我又怎么能对其他人有所期望呢？
对软件来说，architect 一词可以代表很多事情。(软件界很多词都可以代表很多事。) 这通常符合一句话：我不仅是一个程序员，我还是一个设计师[译注2]。还可以进一步解译成：我现在是一个设计师 - 我对于完成所有程序来说太重要了。然后这个问题就变成，当你要展现技术领导的时候，你是不是该把自己与烦琐的程序撰写分清楚？

这个问题引起众多的不满。我看到人们对于再也无法担任设计角色这样的想法感到生气。我最常听到：在 XP 没有设计师的挥洒空间。
就设计本身的角色来说，我不觉得 XP 不重视经验或好的设计。事实上多位 XP 的提倡者 - Kent Back、Bob Martin、当然还有 Ward Cunningham - 都是我从而学习设计的对象。然而这也代表着他们的角色从大家既有的印象中开始转变成为技术领导者。
我将以一位 ThoughtWorks 的技术领导者 Dave Rice 为例。Dave 参与了很多个研发周期，并且非正式的指导一个 50 人的项目。他担任指导的角色意味着要花很长的时间与程序员为伍。当程序员需要帮助，他就介入，否则就留意着看谁需要协助。他的座位有一个明显的特征，担任一位长期的思考工作者，他可以在任何形式的办公环境适应良好。他曾经与发行部经理 Cara 共享办公室一段时间。而在最后几个月，他更是搬到工程师们工作的开放式空间 (就像 XP 小组喜欢的开放式 "战斗空间")。这么做对他很重要，他可以知道事情的进展，并适时伸出援手。
知道 XP 的人就能够了解我描述的是 XP "教练" 的清楚角色。的确，在 XP 玩的文字游戏中提到领导技术就是在描绘 "教练" 这个角色。其意义是很清楚的：在 XP 技术的领导特质是透过教导程序员和帮助他们做决定而呈现。这是一种需要良好人际管理和技术并重的技巧。Jack Bolles 在 XP2000 说：孤单的大师有点机会了，合作和教导是成功的关键。
在研讨会的晚餐会上，我和 Dave 在谈话时对 XP 有了些对立的意见。当我们讨论到以前的经验，我们的方法有相当的类似。我们都偏好 adaptive，iterative development，也认为测试是重要的。所以我们都对他反对的立场感到疑惑。然而他提到 "最后我要的是程序员照着设计忙于重构"。事情一下子明朗起来。后来 Dave 又对我说 "如果我不信任我的程序员，我何必要用他们呢？"，观念上的隔阂就更加清楚了。在 XP 里头，有经验的研发人员所能做的最重要的一件事就是尽量将所有技术传继给新手。不同于一个决定所有重要事情的建筑师，你有一个能够教导研发人员如何做重大决定的教练。就像 Ward Cunningham 指出，这么做不只是增进了新手的能力，对项目的好处更大于一个孤立无援的超人所能做的。[译注7]

Things that are difficult to refactor in

我们能用 refactoring 来处理所有设计方面的决定吗？或者，有些问题太普遍而且难以在将来加入设计中？此时，XP 的正统做法是所有需求都可以轻易的在需要的时候增加，所以 YAGNI 总是能够适用。我猜是不是有例外？有一个不错的，被讨论到的例子是软件的国际化。这是不是一种现在应该立即进行，否则以后再加入时会觉得痛苦的事情？
我能轻易的想象一些事情就是这种情形。事实上我们仍然掌握太少的信息。如果你必须陆续加入一些功能，如国际化，而你知道那需要多少工夫。你比较不容易意识到在你真正需要他之前，你要花多少时间加入它，并且要长时间的维护它。你也比较不容易察觉到也许你可能做错了，所以到头来还是需要做些 refactoring。
有一部份能够为 YAGNI 辩护的理由是有些预先做的功能可能最后并不需要，或者它并不如预期的结果。不做这些所省下的力气比用 refactoring 来更改成为符合需要所用的力气要少。
另外一个要想的问题是你是否真的知道怎么做。如果你有很多次做软件国际化的经验，你会知道该用什么模式来作。那样的情形下你应该会把它作好。这时你所加入的预留的结构

可能会比你头一次处理这种问题要好。所以我的想法是，如果你知道怎么做，你就要考虑现在做和将来做，两种情形之间不同的成本。反过来说，如果你没有处理过那样的问题，不仅是你无法正确判断需要的成本，你也比较不可能把事情作好。这种情形，你就要选择将来再做。如果你还是执意做了，而且尝到苦果，可能会比不做的情况更糟。当你的组

员更有经验，你对相关领域有更多认识，你对需求也会更了解。通常到这时你回头看才会发现事情有多简单。提早加入的设计比你想象中要难多了。
这个问题也跟 stories 的顺序密切相关。在 Planning XP 一书中，Kent 和我公开的指出我们的歧见。Kent 偏向于只让商业价值这一个因素影响 stories 的顺序。在短暂的意见不合之后 Ron Jeffries 也同意这种想法。我仍保持怀疑。我相信在商业价值和技术风险之间能找到平衡点。基于这样的理由让我提早为软件国际化做准备以降低风险。但是这种做法也只有当第一阶段发行就需要将软件国际化才能成立。尽快达到一个阶段的发行是非常重要的。任何原来不需要而后来必须增加的复杂性都值得在第一阶段发行之后才开始。发行之后运作中的程序有强大的力量，它抓住顾客的注意，增加信任感并且是一个学习的好机会。就算是在初次发行之后会有更多的事情要做，还是要尽所有努力将第一阶段日期往前推。

So is Design Dead?

没什么原因，只是设计的本质已经改变。XP 的设计追求以下的技巧：
持续保持整洁的程序代码，越简单越好。
重构的技巧，所以当你觉得必要的时候都可以有信心的动手。
具有 patterns 的知识：不只是照它的解法，更要感觉何时可以应用，或是如何导入 patterns。
知道如何将设计说给必要的人了解[译注8]，用程序代码、或是图形、或上述所有的工具：交谈。
以上挑出来的技巧看来都挺吓人，但是要成为一个优秀的设计师本来就很难。XP 也不是要让它变得简单，至少我就不觉得。但是我想 XP 让我们对有效率的设计有全新的看法，因为它让渐进式设计听起来是可行的方式。而且我也很支持演进 - 否则谁知道我会变成什么呢？

Acknowledgements[译注9]

过去这些年我从很多好朋友身上偷学到不少好的想法，很多都已经记不起来。但是我记得从 Joshua Kerievski 那里偷到的好东西。我也记得 Fred George 和 Ron Jeffries 给我很好的建议。我当然也不能忘记 Ward 和 Kent 不断有好的想法。
我也感谢曾经提出问题和指出打字错误的朋友。同时感谢 Craig Jones 提醒我好几个漏掉的 "a"。

Revision History

[译注1] 一种在着手进行程序代码的撰写之前，就先按照既定的程序分析、设计、制图、撰写文件等等耗时费力的工作方式。
[译注2] 在台湾你觉得 "程序设计师"、"软件设计师" 和 "软件工程师" 有什么不同吗？相信大部分的人觉得都是同一种角色。但是从英文字意就比较容易区别 "programmer"、"designer"、"architect"等等不同的角色。这样的语言文化差异性，也

是我在内文中留下不少原文的原因。在部份字句中，留下原文并不影响阅读的顺畅，但是可以避免文意因为翻译所造成的模糊或扭曲。

[译注3] 在建筑业应该是称呼“施工人员”比较顺耳，而在软件业应该是“programmer”比较恰当。但是这两者都是“builder”。

[译注4] 关于这一点，译者也提供一个实际的经验，也就是翻译这篇文章的过程。我和一位朋友一起翻译这篇文章，而且因为都先看过 XP Distilled 一文，所以决定采用 XP 12 个有效的实务技巧其中的 Continuous Integration、Pair Programming、Small Releases、Refactoring、Coding Standards、Collective Code Ownership 等等技巧。因为 XP Distilled 一文的经验我们首先对于部份翻译名词提出彼此的统一版本，这就是一种 Coding Standards。我们同时都对这篇文章进行翻译，不同的人进行同一项工作进度就不会相同，每翻译几个段落就将进度寄给对方，这就是 Small Releases。当我收到朋友寄来的部份，我就对照两方的差异将文章针对译意的正确性和辞句的通畅做初步的整理和修正，这就是 Continuous Integration。我整理过的部份再寄回给朋友进行检查，如果发现不妥的部份，就进行讨论或修改，这就是 Refactoring。从头到尾的过程中，我们都收到彼此的翻译版本，也拥有整理过的版本，对于每个部份也都清楚了解，符合 Collective Code Ownership 的精神。最后，因为我们刚好是两个人，不巧又可以冠上另一个技巧Pair Programming。所以，我们使用了 XP 一半的方法，对于工作上有帮助的方法。但并不是非得每一项技巧都利用到。我想，更重要的一点是，XP 并不是只能用在软件研发，我们的翻译过程不也借来用了！

[译注5] Story 是一种类似 use case 的描述。

[译注6] 按照原文的意思就是以 32 行的程序代码而能够表现出 16 种 patterns！这样的传奇性功力实在不是经验及见识浅薄的译者所能想象，所以我们只能按照原文翻译，而无法对这句话提出左证。如果读者见过作者提到的例子，很感谢您提供相关资料给我。

[译注7] 这一点也是译者对于台湾软件环境非常忧心的问题，大家都能体会中国人的一种不知道能不能说是重大缺点的民族习性“藏私”。我在不同的机会从文章中以及朋友的口中听到一种想法，要提升团队的整理效率，一种最简单的方法就是将技术落后的成员教育提升到与技术领先的成员相同的水平。这样的想法实在是一种非常有用而且深刻的体认，更是一种感叹。在台湾能够在口头上同意这种做法，而实际上更能够进而实践这种精神的人有多少？？有些人都担心自己的技术被人窥透之后，个人地位恐将遭到取代，更遭的是可能有老板真的会“狡兔死走狗烹”的在榨取经验的传承之后，将有贡献却成本高的成员以各种方式逼走，这实在是让人忧心又灰心的现象啊！译者相信，再困难的技术只要是你能了解，就已经存在了竞争对手，如果不能将身边成员的技术水平同时迅速提升，只有等着被快速超越，更可能被淘汰。译者也实践着 XP 这方面的精神，虽然译者所拥有的技术知识非常粗浅，但是只要同事朋友需要我的协助，我总是知无不言，言无不尽。译者要向流星许愿，希望能够找到越来越多的同类。

[译注8] 有效的沟通也是译者常遭遇的问题之一，有些人无法以不同的方式为事情下批注，有些人对于门外汉还是满口专业术语，有些人不能观察顾虑到听者是不是已经了解所阐述的精髓。这些都不是有效的沟通。我们似乎还不够重视人际沟通这门学问，如何以对方能懂的方式说给对方听，真是不容易呢！译者只是发出感叹，却也亟需要培养沟通能力，才不会让共事的同事摇头。

[译注9] 译者之一的我 (Daimler) 也要搭个顺风车，感谢朋友 (S.H.Chen) 在百忙之中抽空与我完成这篇文章的翻译，这位朋友在软件工程方面的知识给我的帮助非常关键。翻译本身就存在不少争议，应该按照句子逐字翻译，还是要考虑依据本国文字的用语习惯做适当的调整，以求字句的通顺，真的让人难以取舍抉择。我 (Daimler) 是比较倾向于在不扭曲原著作文意的原则下，对句子做适当的重组和调整。但是这个动作本身就存在着非常大的危险，我对于原文是否有足够的认识？我对于本国文字的造诣是否及格？希望这篇文章对于国内读者能有所帮助，至少是效率上的帮助。更期盼来自各方的指教。