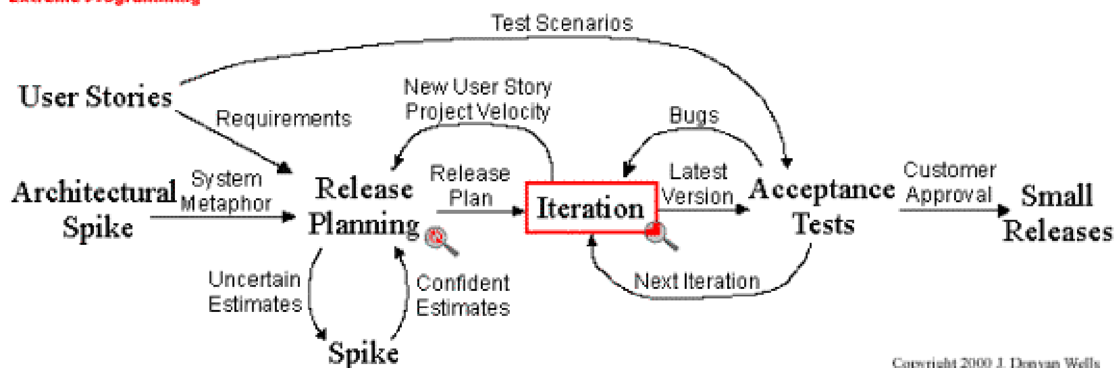
火龙果•整理
uml.org.cn

活 用 X P

---林星 (iamlinx@21cn.com)
2003 年 8 月

Extreme Programming Project



Housor: Philosophaster
BeiHang College of Software
2003-12

XP 作为敏捷方法的一种，拥有很多优秀的实践，用好这些实践，在软件组织中能够起到很好的效果。问题在于，要用好这些实践并不简单，本系列文章的目标就是围绕着 XP 的实践，讨论隐藏在实践内部的敏捷性实质，研究如何灵活的应用 XP 的实践，从而达到改进软件过程的目的。

火龙果•整理 uml.org.cn

软件开发虽然有多个环节，但是我们不能只强调某些环节，任何一个环节出问题最终都会影响产品的质量。因此我们在软件开发中应该考虑整个过程，并且重视人这个因素。

(一) 发挥过程和人的力量

质检员的工作

在以前的工厂作业流程中，产品在生产出来之后，都需要经过质检员的检查。在质检员身边，有两个筐，一个筐上写着合格，一个筐上写着不合格。对于合格的产品，贴上合格证，进入销售的环节，对于不合格的产品，禁止出厂。很久以来，我们一直采用在产品的最终阶段进行质量检验的方式，这种方式用来避免有质量缺陷的产品出厂，但是既没有办法提高产品的质量，也没有办法降低差错率。这种质检方法的基本思想是，产品出现废品是正常的，只要能够找出废品，产品的质量就不受影响。

那我们看看软件开发的工艺流程。当软件经历了需求、分析、设计、编码之后，质检员同样需要检验软件是否满足质量要求。各种各样的测试人员共同担任了质检员的角色。而质检员的工序也不简单。黑盒测试、白盒测试、集成测试、用户测试、并行测试。和工厂不同的是，出现问题的软件并不能够简单的扔到不合格的产品堆中。另一个不同，软件产品一定会出现质量的问题。既然不能简单的抛弃产品，那么只好把产品退回到生产线上。于是，需求人员、分析人员、设计人员、编码人员开始对软件进行调整，力图使软件能够出厂。这个反复的过程往往要持续上一段时间。幸运的软件可以顺利出厂（交付），否则，可能会遭到项目失败的命运。

很明显，我们发现这种做法不够聪明。把问题堆积起来，直到最后才来集中解决，这种做法的代价是非常高昂的。软件开发的特性，决定了越是后期的变更，成本越高。那么，我们应该如何调整我们的做法，来降低成本，提高质量呢？

精益原则

火龙果•整理
uml.org.cn

软件开发总是从其它学科中借鉴管理思路。最早的软件工程从土木工程中借鉴经验，但是后来人们发现建筑和软件开发有很大的差异性。故而新的软件开发方式开始兴起，其中就包括了 XP 方法论。同样的，新的软件开发方式仍然在理论上寻找立足点，这一次众人的焦点落在了现代管理理念上。土木工程管理的一个很大的问题就在于忽视了人的作用，而现代的管理理念将人的作用提到了一个新的高度，这和新兴的软件开发思想是相同的。而对软件开发思路影响最大的，应该算是丰田公司提出的精益生产（Lean Production）的概念。

二战后的美国，以福特公司为首的汽车制造公司在大肆提倡规模制造（Mass Prodution）的同时，东方的日本，丰田英二等人在考察了美国的制造思路之后，认为美国的制造方式不适合日本，提出了自己的精益制造（Lean Production）的思路，精益制造成就了一代霸主－丰田公司，丰田的制造方式被人称为 TPS（Toyota Production System）。丰田公司的丰田英二和大野耐一等人进行了一系列的探索和实验，根据日本的国情，提出了一系列改进生产的方法：及时制生产、全面质量管理、并行工程，逐步创立了独特的多品种、小批量、高质量、低消耗的生产方式。这些方法经过 30 多年的实践，形成了完整的"丰田生产方式"，帮助汽车工业的后来者日本超过了汽车强国美国，产量达到 1300 万辆，占到世界汽车总量的 30％以上。

回顾这段历史，和软件开发的历史何其相似。大规模制造理论认为，一定程度的浪费，一定程度的废品是正常的，允许的。而在软件开发中，浪费、成本居高不下也同样成为阻止软件开发迈向工程化的一大障碍。像 XP 这样的敏捷方法从精益制造的思路中吸取了很多的优秀思想，例如，不断改进质量，设计决策应该交给最贴近生产的人员，根据客户的需求来推动生产。虽然我们一直在强调软件开发和制造行业截然不同，但是，处于变革的十字路口的软件开发行业，总是不断的从其它的行业中寻找可借鉴的理论。这种借鉴来的思路就被称为精益编程（Lean Programming）。精益编程的思路包括：

- 消除浪费。 任何不能够为最终的产品增加用户认可的价值的东西都是浪费。无用的需求是浪费，无用的设计是浪费，超出了功能范围，不能够被马上利用的代码也是浪费，工件在不同的开发组之间无意义的流转，也是浪费。
- 强化学习，鼓励改进。软件开发是一个不断发现问题，不断解决问题的过程。而学习能力的强化，能够令软件开发工作不断的获得改进。
- 延迟决策。软件开发如果工作在一个不确定的环境中，变化性会对软件开发本身造成伤害。延迟决策，当环境变得逐渐清晰之后，才有充足的理由来进行决策。而对软件设计而言，如何构建一个可支持变化的系统则成为关键的问题。

火龙果•整理 uml.org.cn

- 尽快交付。自从互联网流行之后，速度成为了商业中的至关重要的因素，从而直接影响了快速软件开发的成熟。软件阶段性交付的周期越快，软件的风险就越容易识别，用户的需求就越清洗，软件的质量就越高。
- 谁做决策。谁做决策？是高高在上的高级经理，还是贴近代码的编码人员。决策取决于准确的信息，但是掌握这些信息的权威者往往就是做实际工作的编码人员，将编码人员的建议、决定和实践纳入到决策的范畴来，是成功决策的关键。

精益编程代表了一种思想，很多的 Agile 方法都从各自的理论基础出发，支持了这种思想。而在我们的讨论中，讨论的重点就是放在 XP 上。XP 方法论中最有价值的是他的思想。我们研究、学习 XP，不能够光了解他的实践活动，还需要时刻注意 XP 的价值观，并仔细的思考，在实践活动的背后，到底隐藏着什么样的思想。就好像我们在阅读设计模式一书的时候，书中给出的是各种各样的关于面向对象的设计方法，但是书中仍然有一条主线贯穿其中，那就是面向对象的编程原则。

过程

前一段时间，书店中很畅销的书大多数都和 6s 相关。6s 是全面质量管理理论的发展。其中一个很重要，和软件开发非常类似的思路是，过程的每一个步骤，都会对产品最后的质量产生影响，要提高质量，降低成本，提升客户的满意度，最关键的是要对过程进行研究和分析，发现对产品影响较大的步骤，并制定改进的措施。

> 一家专门提供外卖的公司，常常被客户投诉说送货的时间太慢了。于是他们加强了送货的力量，包括使用更好的工具，雇佣更多的送货人员。但是成本增加了，客户的投诉依然不断。问题出在了哪里？在对整个流程进行了量化评估之后，他们发现，送货的时间对整个的时间影响很小，而更多的时间是花费在了制作外卖的过程中。所以，原先投入对送货流程改进的投资，算是白费了。

做任何一件事情，都需要经历一个过程。从外卖店接到客户的订货电话开始，一个过程就已经启动了。记录客户的地址、地址特征、菜名，给厨房下单，分配外送人员，将地址信息传递给外送人员，送货，寻找目的地，交付外卖并收款，后续过程忽略。一个似乎平常的生活活动，其背后都包含了复杂的过程。对软件开发而言也是一样的。从客户提出软件的构想，一直到客户真正开始使用软件，其间的过程非常的复杂，充满了各种各样的不可预测的因素。

送外卖的过程，每一步都会对最终的质量（客户满意度）产生影响。对客户来说，从打电话到收到外卖的时间，外卖的好吃程度，这些都是属于满意度的组成成分。接到电话后，可能客户的口音较重，记录

员听错了地址，导致后续过程全部白费，除非客户等的不耐烦，打电话来重新记录一次地址。下单给厨房之后，可能厨房的电风扇会将单子吹到了地上，客户的要求就被忽略了。记录员把客户的地址描述信息写的很潦草，送货人员可能看不懂，这时候他需要打电话回来问，这就担搁了送货时间。送货人员可能对客户所在不熟悉，找到地址花费了很多的时间。好不容易送到了客户手上，客户已经等的不耐烦了，更糟的是，由于时间太长，外卖已经凉了。客户决定，下一次更换一家外卖店。虽然每一个环节出错的概率都不是很大，但是各个环节组合起来之后，出错的概率就大的惊人。在分析了这样一个流程之后，我们的感慨往往是，居然能够送到，真是不容易！软件开发的过程难道不是这样吗？每一个环节都有可能出问题，需求未必就代表了客户的需要，设计不能够很好的代表需求，反而对编码增加了一些不稳定的因素，由于进度较紧，编码的工作也比较马虎。这样的过程，我们能够开发出客户满意的软件，那么只有一个解释，以前客户接触的软件开发人员，比我们还要烂。

好吧，我们如何改善这一情况呢？对了，对过程进行改进。既然记录员可能会和客户之间出现错配的情况，那我们就要求记录员在听完客户的要求之后，重复一遍。既然，菜单可能会遗失，我们就在厨房中专门设计一个位置，按先进先出的顺序排列这些菜单，而且保证菜单不会被遗失。既然送货员可能会看不懂记录员的字，那么就让送货员和记录员花费一些时间沟通，要么送货员能够习惯记录员的字，要么记录员写出送货员能够理解的字。既然送货员可能未必认识路，那么就对送货员划片，有专门送 A 区的，有专门送 B 区的，每个人熟悉的范围减小了，熟悉的程度自然就上升了。好吧，有了这样的思想，我们也准备对软件过程进行改进了。不过，并不是现在，本文的剩余部分将会围绕着这一点来进行。

过程中的人

除了过程的重要性，我们还需要考虑人的因素，过程是要依靠人去推动的，没有人，过程就没有任何意义。对软件开发更是如此，开发过程的每一个环节都需要人的参与。从来没有一个方法论象 XP 这样充分的强调人的作用。因此，在 XP 的全过程中，人的因素是始终处于首位的。而 XP 的实践也是根据人的优点和弱点进行精心的设计。我们这里做一些简单的讨论：

计划游戏：我们常常挂在嘴边的一句话是计划赶不上变化。计划，往往都是很多软件组织的一块心病。所有人都知道计划的重要性，可是计划又是最难做的。没有计划，软件过程无从遵循；有了计划，软件过程又常常偏离计划。在变化越来越频繁的现在，计划更是难上加难。对待捉摸不定的计划，XP 的态度是：与其在一开始就费时耗力地制定一堆不切实际的计划，倒不如花费少量的精力做一个简单的计划，然后在随后的软件过程中不断的进行调整。

火龙果•整理
uml.org.cn

这就好像我们骑自行车，设定一个 500 米外的目标，然后我们把车把固定住，选取好起点，并预先制定好角度和标准路线，然后骑着车子，严格的按照原定路线前进。车子能到终点吗？可能性不大，设定好的标准路线上可能会有障碍物，这是你原先没有想到的；由于无法调节方向来保持平衡，车子可能会摔倒。

车子，应该这样骑。看看远处的目标，估算距离和时间，得出一个粗糙的速度，然后我们就上路了。在前进的过程中，我们不断的目测目标，察看时间，并调整速度和方向。目标越来越接近，我们的调整越来越熟练，最后，我们成功的抵达的目标点。

传统的计划方法和第一种骑车方法一样不切实际，花费大量的时间思考几个月后发生的事情是很难的。只有根据变化不断的调整，才是正确的态度。

注意，不把时间花费在计划上，并不等于不重视计划。计划仍然是软件开发的核心。你必须有一个当前的迭代计划，当前的周计划，甚至当前的计划。只有保证每一个小计划的严谨性，才能够保证整个项目计划的成功。

XP 对计划的态度是：你不需要把计划做的多么精密，但是你必须去做计划。计划赶不上变化，这句话说的一点都没错，我们不需要逃避变化，花大力气进行精确的计划既浪费，又没有意义。但是这并不是说不做计划。凡事预则立，我们需要简单明了的计划，然后在软件开发的过程中，不断的修正并完善计划。

学习变化：XP 最适合那些需求容易发生变化的过程，在现实中，我们发现这种情况实在是太多了。可能软件的目标市场发生了变化，要求软件同步变化；可能客户对需求没有充分的了解，导致需求的变化；可能客户的组织或业务流程发生了改变，要求软件变化。种种的可能性表示，在一个一成不变的环境下开发软件已经称为一种奢望。在这样一个残酷的环境中，我们不得不去学习变化。

变化包括两个方面：软件过程如何适应变化，以及软件设计如何适应变化。传统的软件过程往往要求上游的软件阶段确定之后，才能够进行下一个软件阶段。但是变化的需要要求软件过程在多个软件阶段之间切换。

由于变化的残酷性，XP 建议开发人员必须建立变化的意识。你必须去改变你的心态，接受变化，变化是合理的，一成不变的东西压根就不存在。

这里插一句题外话。强烈建议在 XP 的项目中使用面向对象技术。虽然面向对象并没有对软件过程或是软件管理提出任何的要求。但是，一个使用面向对象的团队（注意，是使用面向对象技术，而不是使用面

火龙果•整理 uml.org.cn

向对象语言，这么说是因为有着大量的开发人员使用面向对象的编程语言编码面向过程式的代码），其管理过程也会随之变化。用户参与、迭代开发、重用、使用框架。这些都是在使用了面向对象技术之后自然而然出现的变化。使用面向对象技术，能够和 XP 方法进行更加紧密的衔接。

除了上面讨论的两个简单的思路，本文的其它部分都会针对 XP 中过程和人两方面的因素进行讨论。

本文的定位

本文不是一篇介绍 XP 基本知识的文章，这方面的资料已经很多了，要想全面的了解 XP，人民邮电的一套 XP 系列丛书是非常好的一个开始。而本书的定位是讨论在实际的软件开发中，如何灵活的应用 XP，如何遵循 XP 的思想，但又根据实际情况进行折衷。虽然本文没有介绍任何的 XP 基础知识，但是仍然适合 XP 的初学者阅读，刚接触 XP 的人往往都有各种各样的困惑，而从国外翻译过来的注解却未必适合国内的环境，因此阅读本文能够从实践的角度更深的理解 XP 的思想。

和其它的方法论一样，XP 不是万能的。一个软件组织能否从 XP 中获益，不是取决于 XP，而是取决于这个软件组织自身。正如我们在一开始就强调的，学习 XP，关键在于学习思想。软件组织应该根据自身的情况，活学、活用 XP，而不是人云亦云。XP 可不是制作一堆卡片。切记，切记。

文章没有全面的介绍 XP 的所有实践。因为作者并不是 XP 的绝对拥护者，我们以一种客观的态度审视 XP，我们介绍的内容，是在采用了 XP 的实践或是吸收了 XP 实践中的思想之后的经验；我们没有介绍的部分，是因为环境原因无法实践或是不对其表示赞同（但并不是不赞同）。

其实本文介绍的很多知识并不是 XP 的专利，其它的敏捷方法也都提到了这些优点，例如自适应软件方法。所以，更准确的描述是本文如何从 XP 中学习先进的软件开发理念。

****

火龙果•整理 uml.org.cn

螺旋、迭代、增量，不同的名词代表了同样的含义－分阶段开发软件。众多的方法学都采用了这种思路设计软件过程。但是在实践中，更多时候，分阶段开发软件带来的是痛苦。看来，我们常常被书中优美的叙述所迷惑，却没有真正想过实施中的难题。那么，如何管理分阶段的软件开发呢？如何应对现实中的难题呢？

(二)考核和评估之别

在绩效管理中，有两个名词：考核和评估，分别表示了绩效考核和绩效评估两种绩效管理方式。这两者有什么区别呢？

我们说考核是一种制度，而评估是一个过程。怎么理解呢？很多的公司都有绩效考核的制度，这个制度一般是在年底的时候，对员工今年的工作做一个评定。考核是一个点。但是评估不一样，评估是针对某一段时间中员工工作中的不足之处，需要改进之处进行评价。不论是考核还是评估，它们两者虽然都是为了达到评价并改进员工行为的目的而设计的，但是做法是不同的。考核针对过去的事情进行评定，容易实现，但是效果不佳，因为时间一长，大家可能忘记了以前的事情，而要公平的对过去一年的表现做一个评定也不是一件容易的事，评估则不同，评估是不断进行的，针对刚刚发生的事情做出评价，并找到改进方法。就好像我们在第一章中举的外卖店的例子，不断地对过程进行分析和改进，这就是一种评估。评估的效果不错，但难以实现。

软件开发中的考核和评估

这一思路在软件过程中，直接表现为里程碑和迭代的思路。我们可以想想，里程碑是不是一种制度。在需求结束的时候，我们需要需求规约文档，风险列表等等一系列的文档，在设计结束的时候，我们也需要另一些文档。这种处理方式就是考核的思路。但是很多时候，这种考核起到的作用是有局限性的：

工件的设计原本是为了辅助生成最终的代码，但是往往会演变成为了通过里程碑而设计；

- 里程碑的设计不能够完全捕获所有的问题，部分的风险被隐藏了；
- 难以对工作量和工作价值进行评估；
- 里程碑揭露问题的时间要远远落后于问题出现的时间；

火龙果•整理
uml.org.cn

这里对里程碑的方式做一些分析。我们对问题的理解往往是逐步深入的。在项目一开始的时候，业务和技术上都存在问题，存在不确定性和风险，这时候往往是最需要评估和验证的。但是里程碑方式往往要求必须深入的分析需求，很多的问题并没有得以解决，而是被悄悄的有意或无意的掩盖了。需求毕竟不是软件，它是一个不同人具有不同理解的模型，这时候，项目中各个角色对它的理解都不相同，但是这并不影响他们做出一致的决定－通过需求里程碑。问题到了设计阶段依然存在，这时候需求阶段隐藏的一些问题开始出现，导致我们不得不补充一些工作量。但是所有的问题也没有得到解决，依然存在未知的风险。那么风险到了什么时候才会暴露出来呢？最乐观的情况是在编码时期发现，最悲观的情况是在交付期发现。我们把这种过程称为固化考核过程。

问题在哪里？除了软件本身，模型也好、文档也罢，都不能够代替最后的代码。在精益原则中，我们说，必须消除浪费。当我们在开发工件的时候，我们的浪费行为已经或多或少的出现了。

与固化考核过程相对的，我们认为存在另一种动态评估过程。里程碑或是检查点并不是不重要。但是我们需要转换思路，来将里程碑的实践做的更好一些。我们上面提到说里程碑方式最大问题就在于一定要等到问题都积累起来了才解决问题，而这个时候往往已经错过了解决问题的最佳时机。而动态评估过程的含义就是在过程进行中不断的发现并解决问题，而不是等到问题累积到一定程度才批量解决。过程随着环境的变化不断的调整，以适应变化性和不确定性的需要。而里程碑实践重在提供一个复审的机会，能够从一个较高的层次上来评价软件。

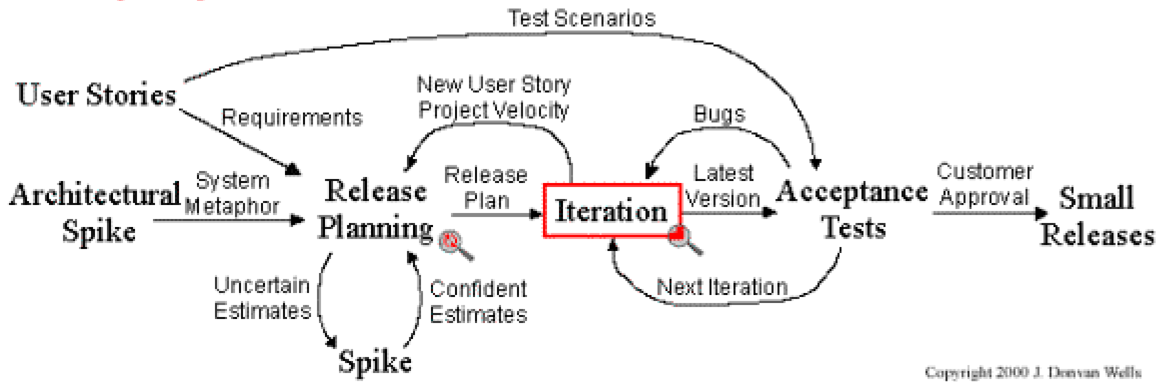
这种过程就是分阶段开发软件的思路，我们也可以称呼它为迭代、螺旋、增量，都没有关系。关键在于，我们需要不断的发现导致客户不满意的问题，发现改进接电话的方法，发现改进做菜的方法，发现更快送货的方法。

实现策略

动态评估过程有一些基本的实现思路，第一个基本思路是尽可能早的发现所有的问题，如何发现呢？进行一次探险式的过程。这个过程周期不能够太长，太长的周期容易失控，而且项目初期人员未必能够全部到位；但这个周期也不能够太短，太短的周期无法发现足够数量的风险，无法为后续的过程提供丰富的数据。

火龙果 • 整理
uml.org.cn

Extreme Programming Project



有时候，我们运用原型法来实现这个 Mini 过程。原型法包括了需求原型和技术原型，分别用于解决业务风险和技术风险。一个典型的需求原型是建立一个界面原型，来帮助客户理解未来的软件，避免抽象的思考。我看过很多界面原型的做法，有使用 HTML 的，有使用画图软件的，有使用规范的 XML 的。但是不管如何，界面原型能够帮助用户直观的理解需求。技术原型的主要目标是解决技术风险，任何一个项目都可能存在这样或那样的技术风险。对待风险的基本态度是尽早的评估风险并制定解决方案，而不是束之高阁。技术风险的解决方案视具体情况而定，但是，值得注意的是，一个项目中，技术风险不能够过多。如果确实存在这种情况，想办法找到有经验的导师或培训师，这要比自己摸索节省许多的成本。

XP 对探险式过程的评估主要包括两个方面，spike solution 和迭代。spike solution 其实就是我们在上面提到了的技术原型。它的目的是让不明确的评估成为明确的评估（参见 XP 的过程图中的 Spike）。只有评估准确了，计划才能够准确。因此它是计划和迭代的输入项。

至于迭代，它是 XP 中的重要概念。迭代选取了用户需要的功能（称为用户故事），并进行设计、开发、测试，迭代不断重复，并解决各种各样的问题。在通过用户的测试和认可之后，最终产生了一个可以运行的版本。这个版本的产生，标志着一组迭代周期的完成。第一个小版本正是我们所强调的探险式的过程。它的成功和教训，足以让你了解项目的各种知识，包括客户的复杂组织关系，投资方的准确意图，找出所有的涉众，发现用例，令团队成员和客户达成初步的共识，验证技术方案，建立一个初步的软件架构，或是针对现有的架构进行初步的映射，程序员需要额外的培训，测试力量似乎不足够，部署环境的风险需要提前解决。只有你按照完整的生命周期真正的去做这项工作，这些问题才会在一开始都暴露出来，否则，其中的很多问题会在后续的阶段中给你制造大麻烦。

第二个基本思路是增量开发。增量和迭代有什么区别呢？Alistair Cockburn 在 Surviving Object-Oriented Projects 一书中将增量描述为修正或改进开发过程，形象的说法是逐步的完成软件开发。注意到，XP 的过程图中的小版本正是一个增量。XP 认为，一个增量应该是可以发布的。做到这一点固然很好，但是并不是所有的项目都能够达成这一目标。例如，第一次的增量目标可能主要是定义一个架构，这个架构并不包含用户需要的功能，但是它是项目开发的基础。架构中可能包括业务实体基础结构、数据操纵基础架构等一系列的框架。但是对于 XP 来说，在用户无法发现价值的框架上花费大量的时间是不值得的，XP 提倡的做法是根据需求的发展来逐步完善架构，而不是在项目一开始就花费精力开发架构。很难评价哪一种说法正确，我比较倾向于前期花费时间进行架构设计，但是实践中确实发生过设计过于复杂导致高昂成本的情况。在花费了大量的时间开发了一个属性处理框架之后，我发现其实简单属性就能够处理大部分的情况，毫无疑问，前期的设计投入打了漂。因此，重要的是权衡前期的投入时间。理想的情况是，你已经拥有了一个可重用的框架（或是架构），这样，你可以将项目的需求映射到框架上，而不是在项目一开始的时候花时间来开发框架。如果你没有框架，在项目一开始的时候，花费一定的时间来开发架构，但是这个时间不宜过长，你完全可以在后续的增量中对架构进行改进，所以不用急于一时。而且，单纯的框架（架构）开发是没有办法进行用户接受测试的，你的测试不得不推迟到第二次增量。这个理由也促使我们尽可能的缩短框架设计的周期。

而迭代则是改进或修正软件质量。这也是第三个基本思路。我们注意看 XP 过程图中的迭代，多次的迭代才构成一次的增量（小版本），每一次的迭代都是对上一次迭代的改进，其中可能是修正了设计错误，或是需求缺陷。值得注意的是，迭代中可能会出现新的需求变更（新需求或需求改变），并令项目人员对项目的进展速度更加的了解（Project Velocity），这些将会反过来影响计划的修正。这体现了我们在上一章所讲述的 XP 对待计划的态度。

并没有法律规定迭代需要和增量一起使用，但很明显，结合这两种方式是一种有效的做法。增量的目标是让项目得以向前推进（这就像是修路的时候，路的长度变长了），而迭代的目标是令软件的质量更优（就像是在一段路上架设路基、铺上水泥，建设路面设施）。这让我们想起了什么，不错，重构的两顶帽子。一顶帽子是为软件增加新功能，一顶帽子是改进软件的质量。非常的相似，只不过一个是过程级别的，一个是程序级别的。这里有一个基本的假设，不要同时增加功能和改进质量。团队也好，个人也好，一次只完成一个目标效率是最高的。

思考

火龙果•整理 uml.org.cn

和传统的先定义问题，然后再解决问题的做法不同，XP 偏重于逐步的精化问题。软件开发中的问题定义和数学中不同，它往往是模糊的，动态的，需要在解决问题的过程中不断的调整解题的思路。对 XP 来说，这种解题思路，体现了其反馈的价值观－尽快获得客户对软件的反馈。

****

在了解了分阶段开发软件的基本思路之后，紧接着就需要考虑实施的问题。分阶段开发最难的，并不是在过程的控制上，而是在软件设计能力上。

(三)实践迭代

应用迭代的问题

有一则故事说的是一个人肚子疼，去看医生，医生给他开了眼药，理由是眼神不好，吃错了东西，所以才会肚子疼。软件开发中出现的问题往往不是单纯的问题，头疼医头，脚疼医脚的做法未必适合于软件开发。

应用迭代并不是一件简单的事情，懂得了迭代和增量的概念，并不等于你能够用好它们。为什么这么说呢？很多的软件组织尝试着运用迭代开发，但是结果却不尽人意，于是将问题怪罪在迭代的方法不切实际上。软件工程中有句著名的话－"没有银弹"。迭代和增量也不是什么银弹。要想做好迭代，缺乏优秀的软件设计思想和高明的软件设计师的支持是不行的。在 XP 中，非常强调各项实践的互为补充。在我看来，迭代能够顺利实行的思路需要重构、测试优先、持续集成等的直接支持。而这些实践，体现了软件设计和软件过程中的关系。

迭代实践出现问题往往是在项目的中期。这个时候，软件的主体已经形成，代码的增长速度也处于一个快速增长的情况。这种状态下的软件开发对变化的需求是最没有抵抗力的，尤其是那些设计本身存在问题的软件。软件开发到这个阶段，代码往往比较混乱，缺乏一条主线或是基础的架构。这时候，需求的变化，或是新增的需求导致的成本直线上升，项目进度立刻变得难以预期，开发人员的士气受到影响。

迭代之外的解决方法

在这个时候，软件组织要做的，并不是在迭代这个问题上深究下去，而是应当从软件设计入手，找到一种能够适应变化的软件设计思路或方法。例如，你是否应该考虑在面向对象领域做一些研究呢？面向对

火龙果•整理 uml.org.cn

象的思路很注重将变化的内容和不变的内容相区分，以便支持未来的变化和应对不确定性。然后你再来考虑相应的成本。

做好迭代有几个值得注意的地方：

代码设计优化

软件开发的能力并不体现为代码量的多少，而是体现为代码实现的功能，代码的可扩展性、可理解性上。所以对代码进行不断的改进，对设计进行不断的改进（具体的次数根据需要而定），使软件的结构比较稳定，并能够支持变化。这是迭代的一个前提。否则，每一次的迭代都花费大量的精力来对原先的设计进行修改，对代码进行优化，这样的迭代效率是不高的，也可以视为一种浪费。坚持不断改进软件质量的做法其实是将软件的集中维护、改进的成本分摊到整个过程中，这种思路，和全面质量管理的思路是非常类似的。XP 中的重构实践有一个修饰词，称为无情。这充分表现了 XP 的异类，但是应该承认，只有设计和代码的质量上去了，才能够为后续的迭代过程打下一个基础，更何况，XP 所处的往往是一个不确定的、变化多端的环境。正是因为这种环境对软件开发有着很大的影响，因此代码质量也被高度的重视。不同的行业，不同的项目，需要根据自己的特征进行调整，但是，只有保证代码的优美性，才能够顺利地达成迭代的目标。

代码设计优化同时必须保持简单的原则，不在一开始进行大量的设计投入。我曾坚信，软件编码之前，严格的软件设计是不可或缺的。但是慢慢的，我发现这种思路未必是正确的。在总结了一些开发经验之后，我发现，很多的时间其实是浪费在了设计上。

在一个软件的设计中，对界面结构有着很强的要求，而 Eclipse 的设计思路正当其时。因此，我兴奋的将 Eclipse 的设计思路注入到界面设计上来，在花费了大量的时间进行设计和实现之后，发现并不能很好的满足需要。更为糟糕的是，由于设计的复杂性，导致调试和变更的难度都加大，而团队的其它成员，也表示难以理解这种思路。最后的这个设计废弃了，但是损失已经是造成了，复杂的设计和实现，足足花费了一个星期的开发时间。

重构和审查

除了第一次的迭代，后续的迭代过程都是建立在前一次迭代的基础上。因此，每一次迭代中积累下来的问题最终都会反应在后续的迭代过程中。要想保证迭代顺利的进行，对代码进行重构和审查是少不了的工作。其中最重要的工作莫过于消除重复代码，重复代码是造成代码杂乱的罪魁祸首。消除重复代码的工作可不仅仅只是找出公函这么简单，其间涉及到重构、面向对象设计、设计模式、框架等众多的知识。

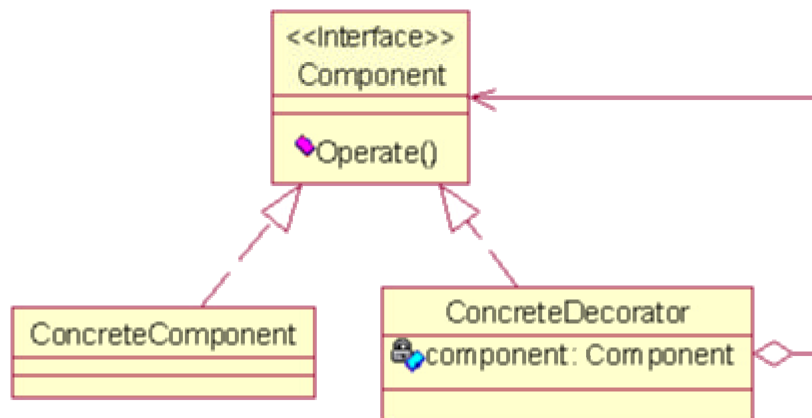
这些知识的介绍并不是本文的重点，但是我们必须知道，只有严格的控制好代码的质量，软件的质量和软件过程的质量才有保证。

推迟设计决策

精益编程告诉我们，尽可能推迟决策。在一个变化的环境中，早期的决策往往缺乏足够的事实支持和实践证明。即便是再高明的软件设计师，难免会犯错误，这是非常正常的，那么，既然目前的决定是有着很大风险的，那为什么我们还要急于做出决定呢？在看待设计这个问题上，一种比较好的做法是，尽量避免高难度、高浪费的设计，以满足现有的需要作为实现的目标。未来的需求等到确定的时候再进行调整。

推迟决策其实是软件设计的一大能力，为什么我们会推荐使用面向对象技术呢？因为面向对象技术具有很强的推迟决策的能力，先将目前确定的问题纳入面向对象设计，并为未来的不确定性留下扩展。推迟决策并不是一个简单的问题，它需要很强的面向对象的设计思维能力。

设计模式中有很多这方面的例子，其中的装饰模式具有很强的代表性。



在设计刚开始的时候，没有人知道 ConcreteComponent 最后的发展会是什么样。很明显，这是一个处于不确定环境中的设计，我们唯一能够确定的，只有 Component 这个类体系一定会拥有 Operate 这个方法，所以，我们设计了一个接口 Component 来约束类体系，要求所有的子类都拥有 Operate 方法。另一个目的是为客户端调用提供了统一的接口，这样，客户端对服务端信息的了解到了最小的程度，只需要知道 Operate 这个方法，并选择适当的类就可以了。还可以对这个模型做进一步的改进，令耦合程度进一步降低。

火龙果•整理 uml.org.cn

在统一了接口之后，我们就可以根据需要来实现现有的功能，我们实现了一个 ConcreteComponent 类，它实现了 Component 接口，并实现了核心的功能。如果在未来，需求的变化，要求我们增加额外的行为，我们就使用 ConcreteDecorator 类来为 ConcreteComponent 添加新的功能：

```
public class ConcreteDecorator implement Component
{
	private Component component;
	public void Operate()
	{
		//额外的行为
		component.Operate;
	}
}
```

先找出共通点，然后实现共通点，并把不确定的信息设计为扩展，这就是推迟决策的设计思路。但是，应该指出的是，上面这个例子的设计，仍然有很多的限制，例如，增加的需求（也就是某个 ConcreteDecorator）中可能拥有新的接口，例如需要一个 AnotherOperate 方法，这时候，原先的扩展性设计就又变得难以满足需要了。在软件设计中，针对接口设计的灵活性和扩展性虽然比以往的设计增强的许多，但它并不是万能的，而且取决于设计师对需求的理解能力和设计水平。此外，推迟设计决策要求我们学习抽象的思维，识别和区分软件中变化和不变的部分。

注重接口，而不是注重实现

Martin Fowler 把软件设计分为三个层面：概念（conceptual）层面、规约（Specification）层面、实现（Implementation）层面。软件的设计应该尽可能地站在概念、规约层面上进行，而不是过分关注实现层面。之所以有时候我们发现在迭代的过程中，软件难以承受这种变化，那么，很大的可能是规约层面和实现层面出了问题。我们在前面一节讨论重构和审查的时候说，消除重复代码是一项复杂的工作，针对规约设计就是其中最有效，但也是最难的一种方法。

我们可以把规约层面想象为软件的接口或是抽象类，或是具体类的公有方法，而把实现层面想象为实现类、实现细节。那么，我们的原则应该尽可能设计稳定的规约层面，并为客户（可能是真正的客户，大部分情况下是使用你的代码的客户程序员）提供一个优秀的、简单的界面（接口）。社会发展到现在的水平，任何一个人都不会花费过多的时间来研究你的代码，如果你的代码不能够为他人提供便利性，那

么最后被淘汰的一定就是你的代码。Java 语言的成功，很大程度上就在于他在保证其强大功能的同时，还提供了一个简单、易用、清晰的规约界面。

在软件设计中，重视规约层面的设计是很普遍的。为什么我们提倡三层架构的软件设计？最重要的是因为他为软件结构合理性贡献巨大，远远超过了他的其它价值。在现代的软件设计中，数据库、界面、业务建模其实是三种差异较大的技术，这就导致了三者的变化度是不同的。根据区分不同变化度的原则，我们知道，必须对三种技术进行区分。而这正是三层架构的主要思路。从这个思路扩展出去，我们还可以根据变化度的需要，将三层架构演变为四层架构、甚至多层架构。而多个层次之间，正是通过优秀的规约界面来达到最松散的耦合的。

在精益编程中，为了避免浪费，要求每位程序员提高代码的规约层面的稳定性是非常有必要的。一个系统中，设计优良的规约界面能够拥有比较好的抗变化能力，能够较好的适应迭代过程。

回归

版本 2 的软件出现了版本 1 中不存在的行为，称为回归。回归是软件开发中的主要问题。在对现有功能修改的同时影响原有的行为，这是造成 bug 的主要原因。在迭代的过程中，必须避免回归行为的出现。而避免回归问题的主要解决方法是构建自动化的测试，实现回归测试。

成功构建回归测试的关键仍然在于是否能够设计出优秀的规约界面，并针对规约界面进行测试。这样，不但设计具有抗变化性，测试同样具有抗变化性。而唯一可能改变的就只有实现了。在回归测试的帮助下，代码的变化是不足为惧的。我们把有关测试的详细讨论放在测试一节中。

组织规则

在后续的章节中，我们会详细的讨论 XP 中的一项非常有特点的组织规则－结对编程。这里我们需要知道，不同的团队有着不同的组织，其迭代过程也需要应用不同的组织规则。例如，组织的规模，小规模的组织可以应用更快的迭代周期，如一周，在一个迭代周期中，团队可以集中力量来开发一个需求，强调重构和测试，避免过多的前期设计。对于大的组织来说，可以考虑迭代周期更长一些，更注重前期设计，并将开发人员和测试人员的迭代周期交错开来。团队的组织构成也是影响迭代过程的主要原因。团队是否都是由相同水平的人构成，每个人的专长是否能够互补，团队是否存在沟通问题。

火龙果•整理 uml.org.cn

(四)需求和故事

如何分析需求，如何记录需求，如何将需求映射为设计，这些永远是需求分析中最为重要的问题。XP 提倡以一种简单实用的态度来对待需求，而在软件开发的历史中，需求分析从来都是最需要严谨对待的工作流程。究竟谁是对的？

故事

每个人都喜欢听故事，这也许是从小就养成的习惯。如果能够把需求分析工作变成听故事的过程，那该有多好。需求分析人员写出一个个优美的故事，开发人员边看故事，边实现故事。也许这就是 XP 的设计思路所在。用户故事，XP 把需求变成了一个个故事，摒弃了枯燥无味的需求稳定。文档的作用是传递信息，如果失去这个意义，再优秀的文档也没有任何用处。但是，完整细致、厚达数十页的需求文档是否真的能够达到沟通的目标呢？对于大多数而言，恐怕看到文档的厚度就已经心生惧意了吧。好吧，我们通过很多的辅助手段，可以强制要求开发人员都投入大量的精力来研究、学习复杂的需求文档。但是这厚厚的需求文档真的能够完整的记录所有的需求吗？更糟糕的是，需求是会发生变化的，到时候如何维护这份需求文档呢？回想精益原则，我们可以判定，这种处理需求的方式一定会产生大量的浪费。将需求做的尽善尽美需要成本，项目组的人员熟悉需求需要成本，维护文档需求成本，解决不一致的问题也需要成本。那么，我们可以针对这几点做一个分析：

- 需求的文档是否要尽善尽美？需求文档的最大目标是将信息从业务人员传递给开发人员（当然也会存在其它的目的，例如作为合同的组成部分）。那么，文档是否完美和能否实现沟通效果并没有直接的关系。
- 开发人员怎么才能够快速理解需求？文档的制作融入了制作者的思想，因此他人理解总是需要一定的时间的。解决问题的思路有两个：一是提供标准通用的做法；二是简化文档，简单的东西总是要容易理解，但简单的东西并不等同于制作容易。
- 维护文档需要成本。不管如何，维护成本始终是无法避免的，关键在于，能否降低这部分的成本呢？维护成本和文档数量、复杂度成正比，因此文档的数量要尽可能的少、复杂度要降低。此外，减少维护的次数也是关键的因素之一，在讨论精益原则的时候我们说尽可能推迟决策就是这个意思。

火龙果•整理 uml.org.cn

针对以上的几点，XP 提出了自己的实现思路－用户故事。用户故事简单，每个人都会写，每个人也都能理解，改变起来也很容易。但用户故事只是对系统功能的一个简单的描述，他并不能提供所有的需求内容，因此，在 XP 中，用户故事的实践需要现场客户的支持。用户故事之所以简单，是因为它只是开发人员和客户之间的一种契约，更详细的信息需要通过现场客户来获得支持。

从 XP 的观点来看，用户故事有这么几点作用：

- 客户自己描述、编写需求。对于任何一个需求来说，最理想的状态都是开发人员教授客户编写需求。最差的情况是开发人员代替客户编写需求。毫无疑问的，XP 要求的就是最优秀的做法。客户要能够自己开发需求，前提条件是编写需求的技巧应该足够简单，能够很容易掌握，或是经过培训很容易掌握。用户故事就是这样一种简单的机制。
- 用户的观点。优秀的需求应该是站在用户的角度来思考问题，是用户能够利用系统完成什么，而不是系统自己完成。用户故事很好的达成了这一原则。因为用户故事是用户站在自己立场上编写，表现了用户对系统的期望和看法。
- 重视全局，而不是细节。需求有精度上的差别，软件开发初期最关键的，是建立一个高阶的需求概况，而不是立刻深入细节。对于 XP 来说，最主要的细节需求获取的方法是经过现场客户。现场客户随时提供对需求细节的指导。因此，用户故事的重点在于，尽可能全面的发现需求，以及，维持一个简单的需求列表。
- 评估的依据。用户编写的需求为软件的估算提供了依据。虽然这个依据是比较粗的，但随着项目的发展，开发速度的估算会越来越精确。在需求初期就进行适当的估算，其目的是让用户能够有一个比较直观的成本概念。这为用户制定需求实现的先后次序提供了指导。
- 用户自己的统筹安排。制定用户故事就像是上商场购物，虽然每件物品都是有用的，但是最后购买的次序和数量则要取决于钱包的厚度。在每一个用户故事具有了成本（即上一条中的估算）之后，用户就能够权衡实际成本和需要，并排定需求的座次。
- 迭代计划的输入。用户对用户故事的选择直接影响到迭代计划的制定，在第一个版本中，用户希望能够实现哪一些的需求（通过选择用户故事），经过估算，这些需求是不是能够在这个版本中实现，计划需要多长的时间。这些都是需求对迭代计划的影响。

故事的弊端

火龙果•整理 uml.org.cn

在收到国外汇款时，业务人员需要记录汇款的相关信息，如果汇款指定的收款人帐户为本行帐户，进行入帐处理，如果收款人帐户属于同城同业（本地的其他银行），则通过同城同业转汇给收款人（后续如何处理？），如果收款人帐户属于异地同业（异地的其他银行），则通过银行的帐户行将汇款转汇至异地，并支付帐户行转汇的费用（后续如何处理？）。

以上是一个银行的国际结算业务中款业务的例子。简短的叙述和非正式的形式体现了 XP 强调的简单原则。故事帮助开发人员和用户理顺流程的关系。在上述例子中，我们看到开发双方对流程仍然存在一定的疑虑（即括号中有问号的部分），但是这并不影响到用户故事的创作，因为这个版本的用户故事还会变化多次。但从这个简单的例子上来看，我们发现故事的形式仍然存在着一些不足：

故事的形式更容易被人接受，但是也有不规则的缺点。任意描述需求虽然节约了培训的成本，但是却造成了不一致性。不同的人对故事有着不同的理解，对需求也就有了不同的理解。需求故事虽然看起来很简单，但是要讲好一个需求故事绝对不是一件容易的事情。需求规约过于形式化和正式化，导致了需求规约难以使用，但是完全不要形式也不是一个好的做法。在形式和可用性之间保持平衡，是讲好需求故事的关键。

需求故事虽然容易阅读，但是却很难写得好。如何控制需求的描写精度，如何分解需求，如何组织，如何确定边界。但是 XP 并不关心这个问题，只要能够起到沟通的效果，怎么做都行。这种态度是否正确我们暂不去评价。但在实践中，由于缺乏系统的指导，一个新手往往需要花费很长的时间才能够学会故事的写法。

对于 XP 来说，需求的开发只有先后次序之分。而先后次序的制定由客户来负责。但是在实践中，识别出先后次序并不仅仅是客户的责任，开发人员同样需要提供需求优先级和风险的建议。这里有几点需求优先级的建议：

- 需求中包含了主要的设计，或是包含了大量的业务实体和典型的业务规则。对于这样具有系统代表性的需求，应该赋予较高的优先级。
- 需求中存在重大的技术风险，例如需求中包括了 OCR 开发包，而开发团队原先并没有相关的经验。
- 需求中包含了重要的业务流程，代表了本软件的主要设计思路。
- 需求具有代表性，并且难以估算，急需对需求进行试验性的评估，以获得较为精确的速度值。
- 需求的时间紧迫。

火龙果•整理
uml.org.cn

采用用例技术

用例技术保持了需求的简单原则，用例和形式和用户故事非常的相似，但是用例具有自己的格式，虽然这个格式也是可以任意定义的。用例的重点是表示系统的行为。我们看看上面的例子如何用用例来表示：

主要角色：业务人员
层次：业务流程级别
前置条件：收到汇款
基本流程：
1 业务人员选择汇入汇款业务。
2 业务人员输入必要的汇款相关信息。
3 业务人员将汇款转入收款人帐户。
  3.1 如果收款人为本银行帐户，直接入帐。
  3.2 如果收款人为同城同业（本地的其他银行），则通过同城同业转汇给收款人（后续如何处理？）
  3.3 如果收款人帐户属于异地同业（异地的其他银行），则通过银行的帐户行将汇款转汇至异地，并支付帐户行转汇的费用（后续如何处理？）。
备选流程
暂缺

可以看到，用例表示的内容和用户故事并没有太大的差别，但用例比较强调格式。虽然不同的团队有不同的格式，但是在同一个团队中，尽可能使用相同和相似的格式（不同的用例可能需要不同的用例格式）基本流程中的每一个步骤都代表了业务人员和系统一次交互，流程非常的简单，但是已经覆盖了一个成功的流程。我们看到，流程的每一步都高度抽象的原因是该用例的层次是业务流程级别的。（业务流程级别也仅仅是一种约定，并不是标准）。利用层次的概念对用例进行精度的划分。在上面的例子中，低精度的用例主要的目标是把握系统的全貌。在 RUP 中，这种用例也被称为业务用例（Business Use Case）。在原先的用户故事中，对分支情况描述比较含糊，但采用了用例的这种描述形式，分支情况就一目了然了，和前面一样，分支情况的表述也有很多种的形式。

用例技术从提出到现在，已经有了大量的经验积累。在 XP 项目中采用用例技术并不是什么新鲜事。但在 XP 中应用用例也必须遵循 XP 的原则，以及精益编程的思路。所幸的是，这些思路是非常自然的，使用用例技术是完全可以实现的。本文并不打算详细的描述用例技术，如果要深入了解用例技术，有几本书是非常值得一看的（见附录）。

先把握系统的全貌：在做需求的时候，常常出现的一种情况是需求分析人员花费了很多的心思来精华、完善某个用例。对 XP 来说，这种做法并不推荐，而根据精益原则，这种行为存在浪费的可能性。我们对

火龙果•整理 uml.org.cn

软件、对项目的认识是不断深入的。因此，我们在项目一开始就深入到需求、故事、或用例的细节，分析人员的能力可能很强，能够正确的捕捉到用户的实际需要。但是一个星期之后我们对需求的认识就有可能发生变化，也许是原先对用例范围的界定出现了问题，也许从另一个角度分析用例效果会更好，也许原先处理用例的思路不正确。不管如何，需求变化的可能性是非常大。用例越详细，发生变化的可能性就越大。这时候，原先花在精化用例上的时间就被浪费了。

因此，不要在一开始就精化需求，一开始的工作重点应该是放在尽可能全面的收集用例，了解整体的业务流程，分析主体业务流程等工作上。在获得了系统的全貌之后，你会发现你原先对系统的认识是不充分的，用例需要根据新的思路进行重新排列，用例的优先级需要调整，在 UML 图中，往往有一张系统的用例概览图，这张图所表示的就是系统行为的一个概述。

寻找优先级高的用例进行精化：我们在上文提到了需求优先级的判断，用例的优先级判断和需求的优先级判断相似。在讨论迭代的时候我们说过，前几次迭代的主要目的是要识别出项目风险。因此，寻找有代表性、优先级高的用例进行精化，能够帮助开发人员更快的理解领域知识，构建起初步的领域模型。

继续上面国际结算的例子，在完成总的用例图之后，我们发现，银行的业务非常的复杂，如果缺少领域专家，要在短时间内领会领域逻辑是非常困难的，同时，我们发现，汇款的业务在日常业务中所占的百分比是非常高的，而汇款业务涉及到了大多数的领域知识，而业务流程却相对简单。因此，我们决定，先把汇款的用例作为一个突破口，在完成了这个用例之后，我们的开发人员就会对业务领域有着比较深入的认识，也就能够进行更复杂的工作了：

主要角色：业务人员
层次：业务流程级别
前置条件：收到汇款
基本流程：
1 业务人员选择汇入汇款业务。
2 业务人员输入必要的汇款相关信息。
3 业务人员将汇款转入收款人帐户。
 3.1 如果收款人为本银行帐户，直接入帐。
 3.2 如果收款人为同城同业（本地的其他银行），则通过同城同业转汇给收款人（后续如何处理？）
 3.3 如果收款人帐户属于异地同业（异地的其他银行），则通过银行的帐户行将汇款转汇至异地，并支付帐户行转汇的费用（后续如何处理？）。

备选流程
2．A 在任何时候，业务人员都可以应客户的要求对向汇款银行进行查询。

火龙果•整理 uml.org.cn

2．A1 在收到汇款银行的查询答复之后，记录答复信息。
2．B 在任何时候，业务人员收到汇款银行要求退回汇款的授权。
2．B1 如果汇款未被提走，根据要求将汇款退回汇款银行。
2．B2 如果汇款已被提走，通知汇款银行无法处理，用例结束。

注意到，在这个例子中我们对用例优先级的判定条件和上文的稍有不同，我们选择有代表性，但又相对简单的用例作为高优先级的用例。这样做是因为对业务领域比较陌生，一开始实现复杂的需求有很大的难度。所以，虽然我们提供了一些制定用例优先级的思路，但是实践的时候仍需要根据实际情况权衡。

迭代精化：用例的编写过程是一个对业务领域不断熟悉的过程。随着调研的深入，不断有新的问题显露出来，需要补充或修改原先的用例。这里有两种情况，一种是在同一个增量内，在对用例 B 精化的时候，发现用例 A 中忽略了一种情况，这时候我们就需要补充用例 A。例如，我们在精化其它用例的时候，发现汇款用例中忽略了报表的需求，这样我们的工作又必须回到汇款用例上。这样的情况是非常普遍的，这就要求我们不要过分的修饰用例，不要把精力花在用例格式上，这样只会造成浪费。

第二种情况是在不同的增量中，这时候用例往往会加入新的需求、新的情境。我们如何去控制不同增量期间的迭代呢？一般来说，有两种方法，一种是对原有的用例进行增补，增补的部分用不同的颜色或标记。另一种方法是为用例建立版本，不同版本的用例对应于不同的增量周期。这样，对应对 N 个增量周期就有了 n 个不同版本的用例（n=N）。不管是哪一种情况，都要求我们采用迭代的思路来处理用例。

形式不是最重要的：在团队中强制要求统一的用例书写格式是有意义的，但有的时候，这个意义并没有想象中的那么大。可以约定条件的编写形式、也可以约定层次的划分。但是过分的强制形式就没有什么意义了。

****

火龙果•整理 uml.org.cn

(五)测试管理

无论从那一点上来看，要保证软件的质量，测试工作是少不了的。而测试往往又是经常被忽略的。对于敏捷方法，精益编程而言，如何保证测试的有效性？如何减小测试的成本？是测试中首要考虑的两个问题。

测试过程

要做好测试可不是一件容易的事情。测试工作和软件开发密切相关，却又自成体系。测试并不是一个单独的阶段或活动，测试本身就是一个过程，具有自己的生命周期，从测试计划开始，到测试用例的制定，测试的结构设计，测试代码的编写。测试的生命周期和软件开发生命周期拧在一起，相互影响。当然，我们还是那句老话，罗马不是一天建成的。对我们来说，还是从简单的开始。

在我们谈及精益编程理论的时候，曾经讨论过全面质量管理的概念：生产过程的每一个环节都需要为质量负责，而不是把质量问题留给最后的质检员。这对于软件开发有着很好的借鉴。软件开发中最头疼的就是质量问题，因为人的行为过于不确定了。在经过漫长的软件开发周期之后，软件渐渐成型，但是缺陷也慢慢增多，试图在最后的关头解决长期积累的问题并不是一个好的做法。软件开发到了这种时候，发现和修改缺陷需要付出很大的代价。

我们说，最后关头的测试并不是不重要，但是，软件质量问题应该在整个软件过程中予以重视。

测试的最小单位

测试问题的很重要的思路在于测试的管理上，如何管理一个项目中所有的测试，以及它们相关的文档，相关的代码，如何定义测试人员的职责，如何协调测试人员和开发人员之间的关系？

XP 的测试优先和自动化测试实践是一个非常优秀的实践，我们也曾不止一次的提到该实践，但是对 XP 强调的单元测试，很多人都有一些误解：

- XP 中提供的例子过于简单，无法和生产环境相结合。XP 中的单元测试只是为测试提供了一个具体的操作思路，但是它并不能取代其它的测试原理。如何进行测试，如何组织测试，如何管理测试，这些都要由不同的软件组织自己来进行定义。

- 测试代码本身不能够适应变化。黑盒测试的理想状况是外部行为不因为内部行为的改变而改变。当需求或是设计发生变化的时候，一段代码的内部行为需要改变，但是外部行为却不需要变化，这样，针对外部接口进行的单元测试同样不需要改变，但是这个规则一旦被违反，我们就需要付出同时改变测试代码的双重代价了。因此，测试代码的设计本身就是很讲究的。

单元测试（有时候也称为类测试）是代码级别的测试，是测试的最小单位。XP 非常看重这个最小单位。我们观察测试优先框架 XUnit，发现它使用组合模式将大量的最小单位的单元测试组织起来，形成完整的测试网。所以，XP 的思路非常的简单：最小单位的测试能够做好，全系统的测试就能够做好。这个思路未必就正确，但是注重最小单位的测试的思路是绝对正确的。每个部件都正确，最后的软件未必正确，但任何一个部件不正确，最后的软件一定是不正确的。

测试优先

测试优先和单元测试在 XP 中属于同一个实践，但是它们仍然是由区别的。测试优先强调行为，在写代码之前写测试，单元测试主要指的是测试的范围或级别。我们说，测试优先实践真正关心的，并不是测试是否要先于代码，关键在于你是否能够编写出适合于测试的代码，是否能够从测试的角度来考虑设计，考虑代码。

从另外的一个角度上说，坚持测试优先的实践，可以让你从一个外部接口和客户端的角度来考虑问题，这样可以保证软件系统各个模块之间能够较好的连接在一起，而开发人员的思考方式，也会逐步地从单纯的考虑实现，转移到对软件结构的思考上来。这才是测试优先的真正思路。而坚持先写测试，只不过是帮助你转变思维习惯的一种措施而已。对于一些优秀的程序员来说，只要能达成目的，是否测试优先，倒并不是最关键的了。

其实做测试是一件很难的事情，因为很多时候，我们不能够完全的模拟出测试环境，或者是完全模拟出测试环境的代价太高。软件开发总是在一个固定的时间和成本的前提下进行，因此我们必须尽可能用小的成本来达成我们的关键目标。很多关于测试的书中都提到诸如磁盘出错之类的错误是很难进行测试的，但实际上，还有很多很多的内容是难以进行测试的。例如，一个业务逻辑，它使用到了 14 个业务实体和其它的一些配合的类，如何测试它？使用 Mock Object 方法，建立测试 Fixture 的代价将会很高，此外，如果实体类是可以控制的（例如，该实体类可以使用程序来初始化数据，而不是从数据库中获取数据），这个测试的成本还可以接受，如果不是（例如，第三方提供的技术），这个成本将会更高。类似的情况还有很多，但是为什么会出现这些问题呢？其中一个很大的原因就是我们并没有真正的把测试作为软件开发的一个重要的组成部分，

火龙果•整理 uml.org.cn

坚持测试优先的思考方式，可以大幅度的降低测试成本。现代的软件开发往往都依赖于特定的中间件或是开发平台，如果这些第三方产品没有提供一个强大的测试机制的话，要对最终的产品进行全面的测试往往是很难的。例如，在 J2EE 提供的 Jsp/Serverlet 环境，模拟 Http 的输入和输出是一件很难的事情。如果在软件设计阶段不考虑测试，那么最后的测试将会是寸步难行的。但是实际上，如果在软件设计时考虑到测试的困难程度，并将业务代码和环境控制代码区分开发，使之彼此之间没有过大的耦合。这样，测试工作就可以针对独立的业务代码进行，而这个成本就会低很多。

```
public class UserLog
{
        public Service()
                {
                //难以进行测试的代码
                //需要测试的业务代码
        }
}
```

注意到，在上面的示例类中，提供服务的代码分为两个部分，一部分是框架提供的、难以进行测试模拟的代码，这类的代码有很多，例如对 HttpRequest 的处理，模拟 http 的数据是比较复杂的。这就增大了测试的难度。而这部分的处理往往是平台提供的功能，不需要进行测试。第二部分是关键的业务代码，是测试的核心。那么，一方面构建测试环境难度较大，另一方面又需要对业务代码进行测试。因此我们自然就想到将待测的业务代码分离出来：

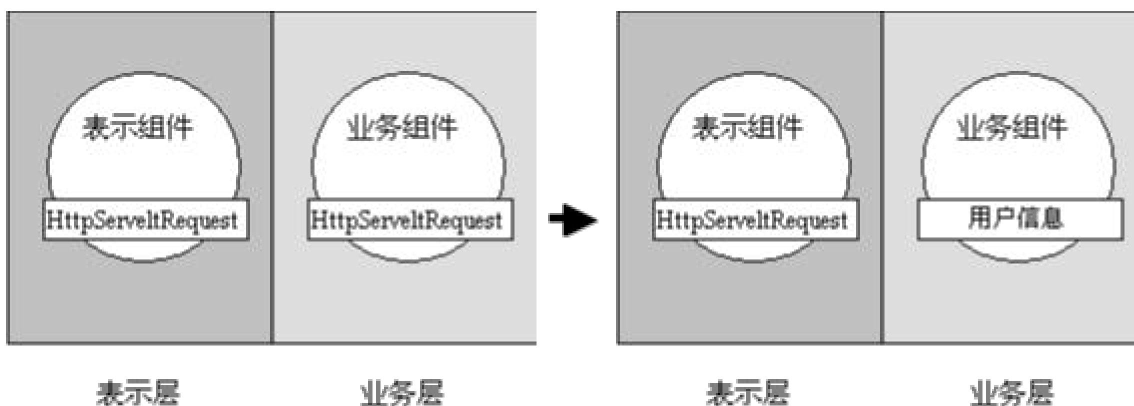
```
public class UserLog
{
   public static void Write(String name)
   {
        //写入用户信息;
   }
}

public class UserLogAdapter
{
public Service()
    {
        //难以进行测试的代码
        UserLog.Write(Name);
```

火龙果•整理 uml.org.cn

```
    }
}
```

这样，测试就可以针对 UserLog 进行，由于不需要复杂的测试环境，对 UserLog 进行测试的成本是很低的。在 J2EE 核心模式一书中，提到了一种向业务层隐藏特定表示层细节的重构思路：

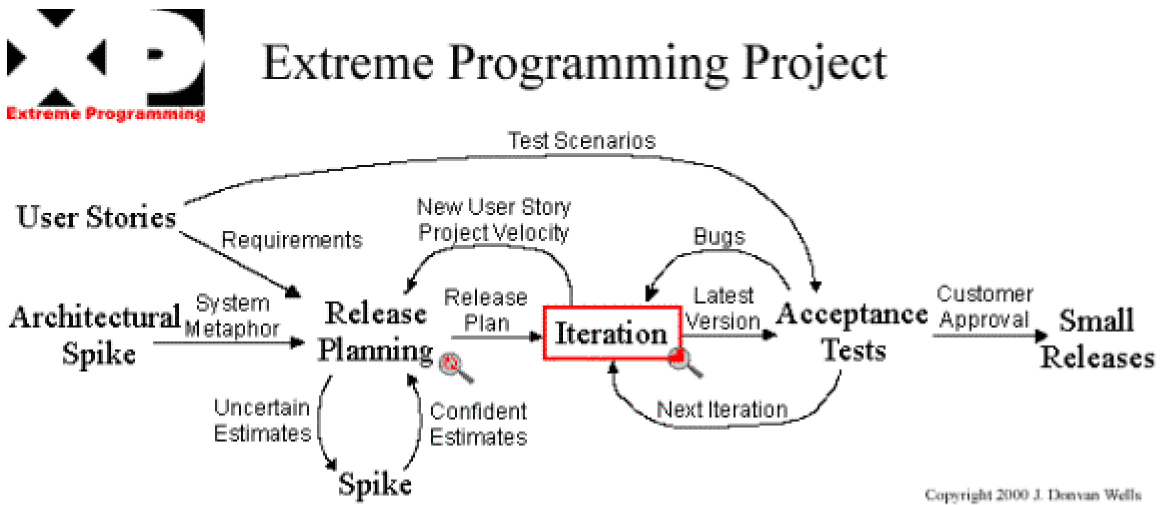


虽然，这种重构方法的出发思路是避免界面层次的细节暴露给业务层，但是从另一个角度来说，也提高了业务层组件的可测试性。毕竟，构建一个用户信息，要比构建一个 HttpServeltRequest 要容易的多。

因此，最合理的引入测试的阶段是在需求阶段。需求阶段的测试工作的重点是如何定义测试计划，如何定义接受测试并获得客户的认可，在需求阶段结束的时候，必须保证所有的需求都是可测试的，都拥有测试用例，需求阶段另一个重要的测试任务是准备构建测试沙盒，建立一个测试环境，以及这个软件项目所需要的测试数据；在设计阶段，测试工作的重点则在于如何定义各个模块的详细测试内容，最好的方式是实现测试代码，并构建测试框架，对于一些比较复杂的项目，甚至还需要编写一些测试工具。实践中我们发现，在 XUnit 的基础上扩展出一个测试框架是一种简单但又实用的方法。XUnit 的重点是对自动化测试提供了一个通用的框架，捕获异常，记录错误和失败，并利用组合模式对 Test Case 和 Test Suite 进行管理。实际上，还有很多工作是可以在 XUnit 框架上继续开展的，例如，软件开发中是不是存在较为通用的测试用例？这样，你就可以定义一些抽象的测试用例，并以此作为测试框架的基础。再比如，我们希望每天晚上在进行日集成的时候，测试结果能够通过短信直接发送到负责人的手机上，那么我们可以在框架中嵌入这部分的功能。这些都属于对测试框架的积累。对一个软件组织来说，很有必要花费时间对测试框架进行积累。这可以简化测试的工作量，并提升软件的质量。

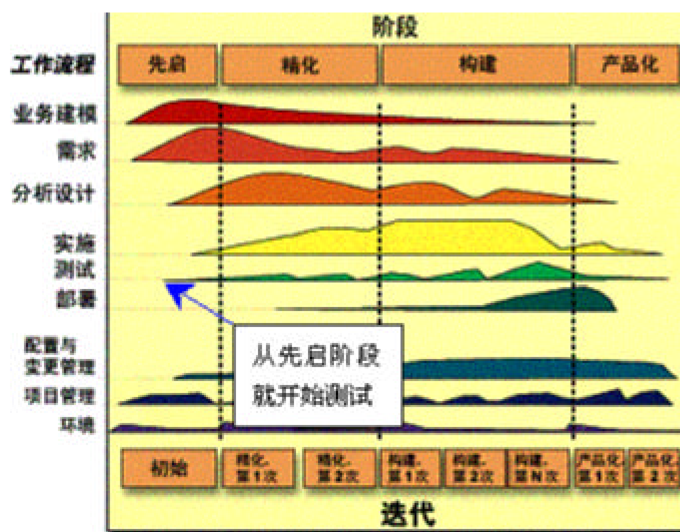
测试过程

我们一开始说，测试有其自己的过程，虽然 XP 并没有花费太多的笔墨来描述自己的测试过程，但经过细心的观察，我们可以发现，在 XP 中同样存在着一个测试过程：



这个过程是从用户故事（或者是我们在上一章中推荐的用例）开始的，用户故事不但为版本计划提供了需求，而且为接受测试提供了测试场景。而对于客户参与的接受测试来说，它为每一次的迭代提供了反馈，包括 bug 的反馈和下次迭代信息的反馈。只有客户认可了接受测试，软件才能够发布小版本。这是 XP 过程最高层次的测试过程。

在上文中，我们提到引入测试最好的时机是在需求分析阶段。因为测试生命周期的起源活动一测试计划和测试用例都需要需求的支持。我们再参考 RUP 的过程：



我们看到，RUP 建议在先启阶段就开始测试活动。在开发过程的前期就进行测试活动，其目的是为了提高软件的可测试性。软件设计如果没能够考虑软件的可测试性，那么测试的成本就会升高，软件质量随之下降。有时候，单元测试或是组件测试是很难进行的。因此，我们需要专门针对类或组件的可测试性进行测试。例如，对于一个实现企业流程的组件，之间涉及到大量的状态、事件、分支选择等等因素。对这样的组件进行组件测试的代价是非常高的。如果能够在组件设计的时候，能够考虑到测试性，例如，将组件拆分为粒度更小的子组件，或是在组件中内嵌供测试使用的方法，能够直接操纵组件的状态。在设计时充分考虑可测试性，是降低测试成本的关键。而设计测试的源泉，正是先启阶段中对需求的分析。对流程组件测试的依据，正是源于项目涉众对流程的需求。

测试的一些实践问题

严格按照先维护测试，再维护代码的顺序要实现变更。在实践中，测试优先常常发生的一个问题是，设计变更影响到测试代码的时候，开发人员往往会绕过测试代码，直接修改代码。

在刚刚接触测试优先思路的时候，我严格按照先写测试的做法编写代码，但是当代码需要修改时，有时候只是一些非常小的修改，这时候我仍然保持原有的习惯，直接对代码进行了修改，在完成代码的修改之后，我突然意识到测试代码需要修改，于是我又修改了代码，由于只是一个小修改，我认为没有必要再运行测试了。这件事情很快被我遗忘了，但隐患就此埋下。到了两天后的集成测试时，测试程序捕捉到了这段代码的错误，经过调试，发现当时认为简单的修改忽略了一种极端的情况。定位错误，调试代码，并通过测试的时间远远超过了当初贪图省事节省的时间。所幸的是，代码在下一个检查点（集成测试）被发现出来。

完善测试网。在我学习并实践测试优先的时候，我所处的团队正处于项目的中期，已经有大量的没有实现测试的代码被创建出来，当时我采取的思路是，新编写的代码必须遵循新的测试方法，旧有的代码保持现状。这样做可以节省一定的成本，但是很快我们发现，投入力量把现有的代码加上测试是绝对值得的。加上测试的代码能够迅速回应变化，仅仅这一点，就值得我们重建测试网。此外，由于需要构建测试，我们还发现了原有代码中一些接口定义不合理或是不规范的地方。

而在另一些一开始就采用测试优先思路的项目中，往往遇到的问题是，随着项目的进展，后期的测试代码越来越优秀。这时候，我们需不需要对原有的测试代码进行改进呢？答案是肯定的，你一定会从中获益的，对于自动化测试来说，修改测试代码并重新运行测试的代价并没有你想象中的那么大。

完美的测试是不存在的，但是测试可以越来越完美。我们在文章一开始就提到了全面质量管理（TQM）的思路，TQM 认为，产品生产的每个过程都会对最后的产品质量产生影响，每个人都需要对质量负责。对于软件开发也是一样，开发过程的任何步骤都会对软件质量产生影响，要提高软件质量，并不是加强测试力量就能够做到的，需要在整个过程中保证软件的质量。构建测试并不断改进测试的行为贯穿于整个开发过程，为质量提供了基础的保证。

自动化测试

自动化测试是 XP 测试活动的另一个优秀思路。在我们讨论迭代的时候，曾经简单讨论过回归和自动化测试。只有测试实现了自动化，回归测试才能实现，重构才能够贯彻，而迭代也才能够进行。所以 XP 一直强调它的实践就像是拼图，只有全部实现才能够完全展现其魅力。单单从这个角度，我们就能够体会到这句话的含义了。

对于一个自动化测试系统而言，有几个部分是特别重要的：

数据准备：对于一个简单的 TestCase 而言，数据准备的工作在 Setup 中就完成了处理（参见 JUnit），但是现实开发过程中的测试数据通常比较复杂，因此有必要准备单独的数据提供类。对于一个完整的企业应用系统而言，往往包含数千的测试用例，而相应的测试数据量也极为庞大，这时候，我们还需要有专门的机制来生成和管理测试数据。

测试数据和特定的项目有关，因此不存在一个标准的建立测试数据的规范。所以我们在 XUnit 框架中看到，框架仅仅只是把建立数据这个活动给抽象出来，并未做额外的处理。但对于自动化测试而言，为各个单元测试建立独立的测试数据是很有必要的。测试数据的独立性是测试用例独立性的前提。测试数据大部分采用脚本的形式建立，包括输入数据和输出数据两个部分。例如，对于一个业务实体，就可以使用一个脚本来对它的属性赋值。脚本文件的形式有很多，例如配置文件、数据库数据脚本等。

验证：验证是将待测试的方法返回的结果值和预定的结果值进行比较，以判断该方法是否成功执行。结果值总是和输入值相匹配，因此，我们经常将结果值和输入值放在同样的脚本中处理。比较通用的验证方式是采用断言机制，此外，还包括错误记录、浏览测试结果，产生测试报告等功能。

桩：桩（Stub）是自动化测试中常用的一种技巧。在 OO 设计中，类和类之间往往都有关系，我们如何对一个依赖于其它类的类进行单独的测试呢？很多的软件设计中都存在难以模拟错误的现象。例如对磁

盘出错、网络协议出错的情况就难以模拟。测试桩的思路就是为了解决这些问题，一个桩并不是真正的对象，但是能够提供待测对象感兴趣的数据或状态，这样，待测试对象就能够顺利的使用依赖对象，或是模拟事件。

****

(六)强化沟通

结对编程是本系列文章讨论的最后一个主题，也是备受争议的一个主题。为什么一个人的工作要两个人来完成，这对于老板来说简直就是犯罪。和前面的主题类似的，我们要学习和应用一项实践，关键的还是要把握其实质。

沟通为王

沟通问题是一个项目成功最重要的因素之一。一个项目可能并没有什么正式的软件过程，但是只要团队成员能够进行有效的沟通，项目成功的可能性就很大，但是如果项目中缺乏有效的沟通渠道，再优秀，再严谨的软件过程也没有用。优秀的软件方法学，总是会在沟通渠道的建立，推动有效沟通上花费大量的精力。我们分析 RUP、XP 等方法学，都会看到很多这样的实践。沟通对一个项目而言是重要的，对一个软件组织而言就更重要了。从长期来看，内部能够进行有效沟通的组织能够得到很好的发展，但是反过来，内部沟通不畅的组织将会出现很多的问题。

在软件开发过程存在的一个很大的问题就是沟通不畅的问题。事实上，这个问题并不仅仅在一个开发过程中存在，在整个软件组织内都将长期的存在，并成为阻碍软件组织发展的一大障碍。这样的说法可能过于理论化，但是我们只要想想，如果现在的项目中，一个主力程序员离开的话，是否会给项目，甚至组织带来重大的影响，就能够理解这段话的含义了。造成这种现象的主要问题是程序是分散在各个程序员手中的。各个代码块就像是程序员们的私有财产一样，神圣不可侵犯。

更为糟糕的是，任何一个程序员都不愿意阅读他人的代码，比起理解别人的代码，程序员们宁可自己重新编写代码，这导致另一个严重的问题？？软件组织中大部分的工作都是重复的，以至于程序员天天忙

火龙果•整理 uml.org.cn

于开发代码，却难以把精力放在更有价值的地方（关于什么是更有价值的地方，我们在下文会详细的描述）。

在一些项目中，我们经常看到这样一种开发环境：每个程序员都拥有个人的隔离空间，彼此之间不进行交流，甚至有时候他们整天不说一句话。在和项目中的一位主力程序员进行沟通之后，我们发现了他们的真实想法：

项目非常紧张，团队成员之间的关系非常的微妙，主力程序员必须要保持自己的主力地位，对他们来说，必须努力写出优秀的代码，同时，你还需要承担项目进度的压力，并提防着其它的程序员。将程序掌握在手中是自己安全感的来源。压力如此之大，他们不得不每天工作 12 个小时以上。程序开发就如同噩梦一样。

虽然未必所有的团队都如此不良的开发人文环境，但是或多或少都存在一些不好的环境因素。可以肯定的说，没有多少人愿意在这样一个开发环境中工作。这些环境因素都影响了沟通问题的形成。

XP 的四大价值观中的一项就是沟通。XP 中的沟通范围很广，有开发人员和客户之间的沟通（我们在需求和故事一章中也提到了沟通问题），有程序员和设计师之间的沟通，有程序员和测试人员之间的沟通。但是本文的重点集中在开发团队内部，即，如何改进开发团队内部的沟通质量。

改进沟通的实践－结对编程

XP 方法论非常强调营造一种轻松的开发氛围，重视人的价值胜于重视过程。沟通是 XP 的一大价值观。XP 中大量的实践是围绕沟通这个价值观设计的。例如，用户故事，现场客户，代码集体所有权等等，但是我们这里要强调的，是结对编程这一实践。本文中不对结对编程做介绍，这方面的资料有很多，没有必要在这里浪费笔墨。本文要讨论的，是我们如何在项目的角度上考虑结对编程。

结对编程是一种非常有效的改善沟通的方法。一对编程人员是协作过程中最基本的沟通单元。在经典的 XP 方法中，结对编程指的是两个程序员在同一时间、同一机器前，主动的共同的解决统一问题。也许经理们听到这句话的第一个反应就是："这不可能，我花了两倍的钱，却只做一个人的事情！"事实上，结对编程运用得当的话，是能够提高工作效率的，不但体现在进度上，还体现在代码质量、以及项目风险上。

个人编程

个人编程往往会遇到各种各样的问题。在软件开发中，编写代码往往只占构建过程中很小一部分的时间，

火龙果•整理 uml.org.cn

很多的时间花在调试代码、改进代码结构，以及针对需求或是设计的变更修改代码。想必很多人都有这样的经历，在一些关键的技术问题上卡壳，而单人进行研究不但费时费力，而且很容易导致士气的低落。

在另一些时候，程序员往往需要在不同的设计选择之间进行权衡，而一个人做出技术决策往往造成内心的不安，这时候就希望能够有另一个同伴支持你做出决定。

说代码是最严谨的工件是一点错也没有，任何一个微小的错误，例如缺少分号，都会造成程序运行的错误。虽然编译器能够检查大部分的错误，可是仍然会有一些深藏其中的，时不时出来捣乱的小错误。一个人的眼睛往往容易错过一些错误，但是两个人同时进行编码，这种出错的概率将会大幅度的下降。

为了修正代码缺陷而进行的调试工作往往会占用大量的人月，如果代码缺陷到了测试团队的手中才被发现，修改缺陷的代价会很高，而如果代码缺陷一直持续到客户手中才被发现，这个代价更是惊人。而通过对开发人员配对，可以减少缺陷的数量。根据一些数据显示，结对编程可以让缺陷的数量减少 15%。相对于在软件过程后期改正缺陷所付出的高昂代价，采用结对编程还是值得的。

以上讨论的是个人编程中遇到的一些问题，这些是很小的问题，但是都会对开发人员的情绪、进度产生影响。而在一个团队环境中，这些问题还会扩大，升级为团队问题。

团队编程

虽然软件组织规定了软件编码规范，但是编码规范不可能约定的过细，过细的编码规范不具备可操作性。因此不同人写出的代码仍然相差很大，优秀的代码和拙劣的代码同时存在，每个人都熟悉各自的代码，但却不愿意碰别人的代码。各种各样风格的代码逐渐产生的代码的混乱。这会产生很多问题。首先，软件组织内部复用的目标难以实现，如果人人都不愿意看别人的代码，你又如何建立一个内部复用的框架呢？现存的代码无法进行控制，旧项目的维护成本不断上升，团队积累也成为一句空话，

其次，代码复审的难度加大。代码复审是非常重要的工作，但是代码的混乱将会加大代码复审的难度，因为复审小组的成员不得不花费时间来了解代码的风格，并做出指导。更糟糕的是，代码复审小组的成员往往都是软件组织中的重要成员，他们的时间都代表了高昂的成本。也许没有人仔细计算过这样的成本，但是这些成本累积起来，也会是一个令人吃惊的数字。

火龙果•整理
uml.org.cn

再次，项目风险和组织风险都随之增大。这种在以项目开发为结算单位的软件开发组织中尤为明显，因项目开发人员离开而导致项目源代码难以维护的情况非常的普遍。对于已经完工的项目而言，这使得项目维护成本上升，对于尚未完工的项目而言，这会打乱现有的项目进度，导致项目进度的延后。

最后，也是致命的一个问题，内部沟通难以有效的进行。软件开发不是一个单独的活动。优秀的程序员组成的团队未必就是一个优秀的团队。究其原因，大部分都是因为沟通不善造成的原因。组织内部的知识很难形成流动，开发人员之间难以共享知识，而新成员也无法从经验丰富的老员工那里学习。

沟通不畅最终会积累形成组织软件设计平均水平无法提高的问题。软件设计属于脑力劳动，但是个人的知识覆盖程度和思考能力都有限，个人的设计往往都是有缺陷的，而雇佣大师级的开发人员的成本是相当高昂的，并不是所有的软件组织都能够像 IBM 或是微软那样雇佣大量的优秀人才。因此面对有限的人力资源（数量和质量两方面），关键的问题就在于如何让有限的资源发挥最大的作用。

软件工艺

在参与一家软件组织的代码复审之后，我加入了这一小节的内容。既然是工艺，当然是一些很细微的环节，例如浏览集合的写法、类和方法的命令、注释的规则等等。这些都属于程序员自身修养的部分，但是很多组织恰恰是在这个环节上存在问题。编码的随意性导致了代码可理解性的下降，为团队共享代码设置了障碍，没有人会主动的去看别人的代码。在前面我们说代码的混乱会导致复审的困难，而代码混乱同时产生的另一个影响，就是软件组织的平均软件工艺水平无法提高。虽然每个程序员都希望能够编写优美的代码，但编写优美代码需要一定的毅力和时间，尤其是在项目时间压力大的时候，代码的优美性常常是被忽略的。但是，强制要求代码优美性并不容易实现，需要监督的成本，效果也难以令人满意。

结对编程可以从组织结构上缓解这个问题。程序员大多是骄傲的，如果有一个同伴在身边，那程序员可拉不下脸来编写难看的代码。这是很有意思的现象，但是挺有效的。程序员通过这种方式，可以相互促进，提高编程工艺水平。虽然软件工艺解决的都是一些微小的问题，但是正是这些问题，最终影响到了软件的质量。从代码管理的角度上来说，管理的基本任务都是这些"小问题"。

过程保证

结对编程可以在有效的解决这些问题的同时保证成本最小，这是结对编程之所以成为结对编程而不是三人编程的原因。在硬件设备的运行过程中，单点故障的最好解决方法是双机备份。这一思想运用到团队和过程上就形成了结对编程的基础。我们见过一个软件组织实施结对编程的初衷是为了保证产品的安全

火龙果•整理 uml.org.cn

性，在产品的各个重要部件上都至少配备了两位负责人。一开始他们没有意识到他们朝着结对编程迈出了第一步，后来他们发现这种方法非常的有效，并针对这种方法进行扩展，形成了完整的结对编程体系。

在传统的软件开发中，一般都会在软件过程中建立几个检查点（Check Point），在这个点上，软件的各个部分都需要进行检查，设计是否符合规范，是否满足需求，程序中是否存在缺陷。但是在每个 Check Point 上花费的时间往往是非常可怕的。每个 CP 上花费的工作包括：

- 熟悉他人的设计思路和代码风格
- 将不同的系统整合起来
- 对缺陷进行改进

而结对编程的实践实际上就是将这部分的成本分摊到每一个人天中去。通过两两互配，让组织中所有的人都能能够熟悉软件的各个部分。这个成本在刚开始时确实会比较高，但是随着对结对编程理解的深入，这个成本会慢慢的降低。根据资料显示，结对编程并不是像大多数人想象的那样，会增加 100%成本，这个数字取决于具体的实现形式，但绝对不会到 100%。

(七)实战结对

结对编程的根本思路是改善开发团队内部的沟通质量。在实际情况中，不同的开发团队面临着不同的沟通问题。那么，该如何找到一个共通的指导思路呢，又该如何根据实际的情况进行调整呢？

成本权衡和策略选择

从上一篇文章的讨论中，我们可以了解到，由于现实的因素，做到理想化的结对编程往往会有很大的阻力。这个时候，我们可以根据实际的情况进行调整，选用不同的方式。但我们如何评估这些方式的成本呢？设计结对，测试结对，复审结对等等的变通方式都存在一个问题：就是表面上看起来它们似乎既达到了结对的效果，又节省了成本。但是实际上，这个成本并没有节省，而是转移了。

在项目中，为了令结对编程的思路更容易令人接受，我们采用了变通的做法，在设计和复审的时候结对，编码则由单个开发人员负责。A 和 B 针对某项需求进行了 2 个小时的设计讨论，然后由 B 负责编码。但是 B 在编码的时候发现原先的设计存在

火龙果●整理
uml.org.cn

考虑不周的情况，他决定对设计进行一些修改，这时候，他想通知 A，但是此时 A 不在，于是 B 根据自己的思路调整了设计，并完成了实现。而在复审的时候，B 不得不花上一段时间来向 A 说明设计变更的原因和细节。

注意到，这个过程中，A 和 B 不进行结对编码而节省的时间其实是转移到复审上来了。当然，复审上花费的时间可能要比编码的时间短得多。但是我们还必须看到，如果 B 变更后的设计也存在缺陷，A 和 B 仍然需要花费一定的时间来改进设计和实现，这种情况也是有很大的可能性发生的。

对结对编程的成本进行讨论并不是要下一个定论。对于不同的组织而言，这个成本是不确定的。对于某一些组织而言，理想的结对编程也许非常的合适，但对于另外一些组织来说就未必。重点在于，必须找到一种方法，使得团队之间的沟通能力得以增强。

不同团队进行结对方式设计的时候的标准只有一种，就是如何改进沟通质量。不同的团队有着不同的沟通问题。找到这个沟通问题，才能够对症下药。有这样一个软件组织，他贯彻结对编程的思路很简单，就是为了减小人员流动对业务的影响。经过研究，我们发现这个组织有这么一些特点：产品经历过数代的演化，结构复杂；开发人员仅对自己负责的模块比较了解，全面掌握系统的人极少；产品拥有固定的客户群，客户时常有修改的需求；任何一个开发人员的流失都意味着他负责的模块在一段时间内无人接手；相对于模块无人负责的尴尬境地，增加一个开发人员的成本是可以接受的。在这样的一种情况下，该组织要求任何一个模块都必须有两个开发人员负责。事实上，采用了这一方法之后，人员并没有翻倍，因为维护老产品和开发新产品的工作是并行的，而且，同一个开发人员不仅仅只负责一个模块。虽然人员增多了，但是客户的满意度提高了，而开发力量也同时得到了增强，这个结果还是令人满意的。

可以看到，在这个例子中，结对的方式并不是 XP 中所描述的结对编程，它只是一种组织形式，但是在解决沟通问题上，两者的思路是相类似的。同样的，我们如果希望在自己的组织中应用结对，那么分析自己组织的沟通瓶颈的工作是少不了的。

设计结对

设计结对的含义是某一模块的设计由双人完成，这里的设计并不是大规模的软件设计（对于大规模的前期设计而言，我们更倾向于让团队设计，请参看敏捷架构设计一文），而是在某个特性在编码之前的设计，这种设计的特点是持续的时间很短（只有几个小时或是几十分钟），但是对于整个代码的质量而言非常的重要，因为我们需要保证设计符合架构的原则，以及设计的灵活性，一致性等等，还需要保证设

火龙果•整理 uml.org.cn

计的性能和速度。而某个特性在设计完成并进入编码之后，这部分特性就已经确定下来了。因此这种小规模的设计往往是软件开发中比较重要的细微点。在设计上配置双人，能够有效地提高代码质量。这种结对的思路是把成本花在关键的部件上，但是小规模设计结对的具体表现往往是两个人对某个问题的某种看法，他并不能以代码或是模型的形式来体现，对非编码者一方的约束比较小，而代码实现很可能和设计有所出入，这样，非实现者也难以获得这方面的知识。这种方式如果单独使用，容易演变成一种形式，效果并不是很好。因此，我们需要其它结对方式的配合。

测试结对

这里的测试结对专指单元测试结对。结对的基本思路是 A 和 B 就类轮廓（类结构和公有方法）达成一致后，A 编写测试代码，B 编写代码来满足测试。如果 B 对设计的理解有误，那么代码一定通不过测试，如果 A 对设计的理解有误，B 必须通知 A 重新编写测试。如果对 XP 的单元测试的改变非常熟悉的话，采用这种方式会有不错的效果。首先，测试代码本身就是小规模设计，而且它以一种规范的编码形式反映出来。只要测试代码足够优秀，它可以捕捉很多的设计缺陷。其次，这种方式是测试有限的一种变体，但是其效果要优于单个人的测试优先。因为一个人思考测试和设计难免有考虑不周的地方，但是如果两个人来考虑的话，测试往往能够发现出更多的问题或缺陷。刚开始使用这种方式的时候，可能会有些不习惯，但是熟悉之后就会比较顺利。

复审结对

设计结对和测试结对都是在编码活动开始之前进行结对活动。但是复审结对则是在编码活动完成后进行的。A 在 B 完成代码之后，需要对代码进行复审，复审的内容包括，代码是否体现了设计的意图，代码中是否存在缺陷，代码是否满足需求，代码是否符合一致性原则。一般这种复审都属于同级复审。当然根据我们下文中讨论的组织风格，也可以让有经验的程序员对没有经验的程序员进行指导。复审结对对软件过程的最大意义就在于它形成了一个持续复审的体制，它保留了复审制度的优点，而且可以克服复审制度中的缺点。例如花费时间长，遭至开发人员的反感，不能够进行彻底的复审等等。

这三种方式的结对可以单独实行，也可以配合实行。这三种方式虽然都不是完整的结对编程实践，但是尽可能的获得了结对编程好处，而成本是相对低廉的。刚开始实施结对编程或是没有足够的资源采用结对编程的，可以采用以上的变通方式。

结对编程的组织风格

结对编程并不是抓阄。成员的组织是需要一定的技巧的。基本的操作思路是，先找出沟通的关键性问题，然后针对问题入手，组织人员。举一个例子来说，对于某个项目而言，参与的开发人员经验较少，开发人员对组织的开发模式不熟悉，对开发的目标领域也同样不熟悉。主要的工作任务都压到了经验丰富的高级程序员身上。为了解决这个问题，在项目的头几次的迭代中，强制实行了配对制度，配对的基本思路是老手带新手，配对的实现是老手编写单元测试，要求新手实现，并共同进行代码复审（即采用测试结对和复审结对两种方式）。在完成一个小模块之后，老手就需要更换他的搭档，以保证在前两个迭代完成的时候，新手能够较为独立的进行开发工作。一开始的进度非常的不理想。老手也有着不同程度的怨言，认为这是在耽误时间。但高层管理人员听取了项目负责人的汇报之后，表示支持这种做法。在所有的新手都和老手搭配过后，情况有了很大的变化。系统的开发速度明显加快，团队内已经形成了密切沟通的氛围，老手们能够腾出手进行更复杂的设计和质量控制，更令人惊喜的是，已经有两名新手快要接近老手的水平了，这意味着，下一轮的结对编程过程中，他们将扮演当老手的角色。

这个例子告诉我们：

- 针对重要的沟通问题实行结对编程
- 一次解决一个问题
- 结对编程的形式是针对沟通问题和环境特点而设计的
- 优秀的实践必须坚持才能够有效果

这是一个非常典型的组织风格，可以适用于很多的软件项目。它充分体现了沟通的重要性。更多的组织风格还包括：

- 培训性项目。有时候这种项目也被称为摸索性项目，项目的最大目标就是为了研究和试验某些技术，以便在软件组织内部推广该技术。在这种项目中，知识的探索和研究往往要比成本更重要，因此可以采用结对编程的形式。
- 质量控制。软件开发的过程中，往往会有设计的核心部分。这部分的设计要么关系到软件的整体结构，要么就是代表了客户最为关心的需求。这部分的软件设计的好坏，将会直接影响客户对软件的看法。因此这部分的设计是值得投入双倍的开发力量的。而在很多的项目中，我们发现很多开发人员并没有意识到这一点，对开发人员来说，可能只是一些微不足道的错误，但这些错误有

火龙果•整理
uml.org.cn

时候就会让客户留下非常不好的影响。因此，识别软件中的重要部分并投入更多的开发力量，往往能够令最终的软件质量有很大的提升。

- 轮岗制度。研究表明，即便是知识管理做的再好的组织，仍然是有大量的知识是保存在人脑中的。而为了不形成一个个的信息孤岛，最好的信息流转的方式就是沟通。轮岗制度，或者说是 Cross Training，正是为了解决这一问题而设计的。轮岗制度解决的另一个问题是，保证你的团队中既没有忙的团团转的人，也没有闲的发慌的人。这是管理着重要解决的问题，而轮岗制度可以很好的解决它。

(八)杂说

XP 还拥有其它优秀的实践，本文讨论了 XP 的另外三个实践，并研究如何在项目中灵活的使用它们。

代码集体所有权

XP 提倡代码归属集体所有，这样做的理由是每个人都可以修改代码，而不是等待别人来修改代码。这种做法可以有效避免形成代码之间的鸿沟。但集体代码所有权也它的问题。

我们尝试了由多人共享代码的做法，其目的是为了加强交流，避免出现一段代码只有一个人了解的情况。这种方法一开始工作的很好，但很快我们发现出现了很多的问题，类的定义变得不清晰了，某些类变得臃肿，我们闻到了"Large Class"的味道。更为糟糕的是，这些类的清理和重构相当的困难，因为这些类的客户太多了。正是因为这些类被广泛的使用，因此大家都对其进行修改和扩充，导致了代码的混乱。于是，我们加大了重构的力度，对这些类不断的进行审查和重构，但是新的问题又出现了，很难找到一个平衡点，既能够保持团队的敏捷性，又保证类的高度可用性。更糟糕的是，不同的人对这些类有着不同的了解和期望，导致了这些类的设计风格有些怪异。

在本文中，我们不只一次的强调过，XP 中所有的实践是配合使用的。项目中采用集体代码所有权不是不行，但有前提。在上面的例子中，我们至少犯了几个错误：

火龙果•整理 uml.org.cn

- 没有考虑到团队成员之间配合的默契程度。只有团队成员之间知识程度相近，或是具有相近的思维观，例如都熟悉面向对象，都熟悉设计模式。这样他们才能够共同保证代码的质量。在项目中，我们尝试了一种新的做法，将集体所有权的范围缩小到组，这个组由三到四个资深的开发人员组成，称为核心组，负责构建基础的框架。这个组之间采用高效的共享代码机制。其他的开发人员利用核心组的成果进行工作，他们并不修改核心代码，但可以提供修改意见。
- 忽视了代码质量。不同人修改代码要比单人修改代码更容易导致代码质量的下降。必须考虑这个成本，并找出恢复代码质量的办法。审查是非常有效的手段，FDD（Feature Driven Development 特征驱动开发）方法就非常强调审查在项目中的作用。只要成本能够接受，再多的审查都是不过分的。
- 代码标准化。这里说的代码标准化不仅仅指代码规范。还包括代码是不是具有可读性，代码的结构是否足够的清晰。这些都可能导致团队集体拥有代码时形成混乱。

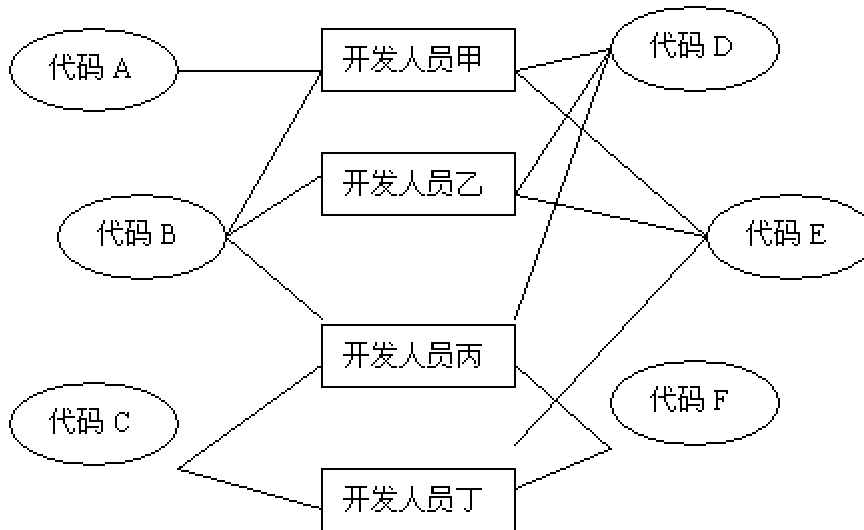
此外，还应该注意到集体代码要求能够频繁的集成代码。代码必须要快速的同步和集成，共享代码往往意味着同一个包，同一个类都有可能被同时修改。这样大大增加了引入 bug 的可能。尽快的同步代码时非常有必要的。

如果一个软件组织不能够解决这些问题，冒然采用集体代码所有权的实践是比较危险的。相反，可以考虑采用个人代码所有权，或是微团队代码所有权。前者说的是个人对个人的代码负责，后者说的是两到四个人对某部分代码负责。

不论是个人代码所有权还是微团队代码所有权，其立足的根本是有明确的开发人员对代码负责，他保证代码的统一设计思路和风格，负责代码的客户端接口，负责维护和改进代码，负责代码的相关文档，负责解释代码的运行机理。个人代码所有权是最清晰的做法，但其坏处和集体所有权正好相反。某个人的代码可能造成进度的瓶颈，任何一个人离开团队都会造成损失。

个人代码所有权很容易理解，但微团队代码所有权就需要特别做解释了。他的组织思路非常类似于我们在结对编程中提倡的组织风格：

火龙果•整理 uml.org.cn



不同的人负责不同的代码，人员之间形成交叉。这样的组织比较灵活，和结对编程有着异曲同工之妙。

持续集成

在 Martin Fowler **的持续集成（在** Agilechina **网站上可以找到该文的中文译本）一文中，对持续继承有着这样的描述：**

在软件开发的领域里有各种各样的"最佳实践"，它们经常被人们谈起，但是似乎很少有真正得到实现的。这些实践最基本、最有价值的就是：都有一个完全自动化的创建、测试过程，让开发团队可以每天多次创建他们的软件。"日创建"也是人们经常讨论的一个观点， *McConnell* ***在他的《快速软件开发》中将日创建作为一个最佳实践来推荐，同时日创建也是微软很出名的一项开发方法。但是，我们更支持*** *XP* ***社群的观点：日创建只是最低要求。一个完全自动化的过程让你可以每天完成多次创建，这是可以做到的，也是完全值得的。***

和本文提到的其它实践一样，持续集成的主要思路是将软件过程末期的软件继承分摊到软件的全过程。虽然没有办法评判两种持续方式的成本，但是持续集成可以获得很多额外的好处。单次集成最要命的地方是除 bug **的过程，尤其是那些隐藏的很深，让人觉得无从下手的** bug**。如果说写代码是一种享受，那修复** bug **的过程绝对是一种煎熬。在这个过程中花费的时间有时候是惊人的，更糟糕的问题是，这部分的时间根本无法估计，这令项目管理者头疼不已。向编码者询问进度时得到的回复永远都是"还差一点儿"。**

持续集成避免了这种尴尬处境，由于间隔的时间很短，集成中出现的问题可以很轻易的发现。即便无法定位错误，最差的情况也可以不把代码集成到软件中。这样，软件的质量就会比较高。此外，持续集成的另一个重要任务是运行自动化测试，保证所有的代码都是经过测试的，没有发生问题的。这里的测试源自于单元测试，在本文的测试一章，可以找到更为详细的讨论。

对一些没有持续集成经验的团队来说，持续集成像是一块吊的很高的饼，看得见却摸不着。要做好持续继承并不容易，但我们可以使用持续集成的思路，来接近持续集成的目标。

持续集成最好的做法是自动化的构建和测试。这要求软件组织拥有很好的配置管理机制，以及丰富的测试脚本编写经验。对于一个企业应用软件来说（抱歉，我只有这方面的经验），软件设计包括很多的因素，要把这些因素都考虑到持续集成的过程中并不是一件容易的事情。因此很多组织都可能缺少这两个因素，但没有关系，我们可以利用半手工的方式来完成持续集成，然后再慢慢的将持续集成的过程自动化。这里提供一个半自动持续集成的思路和改进过程。

首先是定义职责。如果你采用了代码非集体所有权的形式，那么，请明确的指定各个类或包（对于面向过程语言来说是函数和模块）的负责团队（或个人）。同样，你还需要指定数据库模式的负责人。这些代码之间可能会有交叉的地方，但没有关系，只要保证沟通，少量的交叉职责没什么特别的。最好还必须指定一个专门负责持续集成的人，可以让项目中的不同人交替担任该职责。

其次是定义自动化代码。所有的测试都必须写成测试代码的形式，并能够运行。所有的数据库模式定义也必须编写为 DDL 的形式，而不是使用数据库工具。千万别偷这个懒。总的原则是，能够写成代码的都写成代码，只有代码才是可以执行的。我承认，工作量是很大的，但这是必须的。

再次是定义集成的频度。频度的制定取决于团队的规模和沟通质量。对于小的团队而言，一小时一次的集成也是可行的。对于大的项目，可以划分子团队，子团队中采用频繁集成（一小时一次），子团队之间采用日集成（每天集成一次）。有了子团队的集成保证，整个团队的集成一般不会有什么问题。

接下来是使用工具。最需要的是版本控制工具，可以选用正式的，例如 ClearCase。也可以选用简单的，例如 SourceSafe，还可以选用免费的（CVS）。都没有关系，关键在于是否合用。代码和文档的集成都通过这个工具，数据库的集成则通过数据库管理员（就是第一步指定的负责数据库模式的人）。而集成负责人负责协调集成过程，保证集成的成功。

火龙果•整理 uml.org.cn

最后是发现问题，这只是一个开始，你在集成的过程一定会遇到各种各样的问题的。例如集成的时间，数据的相关性，设计的耦合度，测试代码的变化等。没有关系，这里存在一个自适应的过程。一开始的持续集成过程一定是错误百出的，慢慢的就会稳定下来，这时候就是改进的时候了，改进的主要目标始终都是过程自动化。有时候，为了保证集成的质量，我们要求出现错误的人请吃冰淇淋，不要笑，这也是过程的一部分，实践证明，这可是非常有效的，所有人在提交代码之前都会很认真的保证测试成功。

代码标准

代码标准是非常基础的管理常识。但是我们这里并不打算再赘述代码标准的问题。我们将重点讨论开发标准。开发标准包括各种各样的标准。例如过程的标志、文档的标准、设计模型的标准、代码风格的标准、变量命名的标准、大小写标准等等。在 XP 中，其实并不非常强调标准，因为 XP 提倡简单的做法，但有时候标准往往是违背这一准则的，因为它会带来额外的标准化成本。而重量级方法之所以笨重，过分遵循标准正是一大原因。

但在实际的过程中，我认为宁可多投入一些资源在标准制定的执行上。这和国内目前的软件开发实际情况有关系。国外的轻量级方法出现在重量级方法之后，大部分的程序员都经历过强制性的标准化过程。但是国内不同，虽然轻量级方法很好，但是理解或执行不当的话，却常常导致画虎不成反类犬的后果。国内很多软件组织的开发过程仍然处于无序的状态，而开发人员也鲜有标准过程的经验，在这样一种情况下，盲目的推崇敏捷的做法，其实骨子里仍然是一种混沌的状态。

因此，标准的制定和执行总是值得的，虽然会需要一定的成本，但这个成本同它的带来的改善沟通、促进积累等效益比起来不算什么。

但是标准的制定绝对不是要把程序员上洗手间的时间都规范起来，如何保持标准的平衡是敏捷方法的重点。XP 认为代码规范就已经足够了。但我认为至少还有几种标准是需要重视的：

文档标准：这是一个大的话题，最好的文档标准是 UML。一幅图胜过千言万语。当 UML2.0 出世之后，UML 将会越来越强大。因此，使用 UML 来代替部分的文档是必要的。UML 的相关资料有很多，这里不做过多的讨论，但关于 UML 的一句忠告是，不要试图在一开始就利用 UML 的所有类型的图，也不要一开始就利用 UML 所有特性。这种做法，和摆弄文字处理器的特性，与不写文档的做法没啥区别。

设计标准：同样是一个大的话题。设计标准包括如何进行设计，如何表示设计，如何设计架构，如何设计各个层次等等问题。在一个组织中贯彻设计标准是一个长期的过程。但仍然是那句话，做总是比不做的好。

代码风格标准：XP 非常看中代码的可读性，代码可读性好代表了程序员的水平，设计人员的努力。提高代码可读性的最佳实践是重构和复审，这两项实践在本文的其它位置都有详细的讨论。

界面标准：界面标准有时候只是一个很小的问题，但对于现代的软件来说，界面正扮演越来越重要的角色。而在软工领域，优秀界面的关键是一致性。同样的按钮必须有同样的大小、位置和字体。制定一份界面标准是非常关键的。

如何制定标准：标准的制定其实并不需要大量的说明文档，这些文档晦涩难懂，也不会有多少人会看，就算看了也未必会懂。只能是浪费时间。最好的方式是示例和培训。文档标准的说明就编写一份文档范例，并加以简短的说明。再辅以面对面的讲解，其效果要远远超过说明文档。

作者简介

林星，辰讯软件工作室项目管理组资深项目经理，有多年项目实施经验。辰讯软件工作室致力于先进软件思想、软件技术的应用，主要的研究方向在于软件过程思想、Linux 集群技术、OO 技术和软件工厂模式。您可以通过电子邮件 iamlinx@21cn.com 和他联系

参考资料

- Martin Fowler and K. Scott, UML Distilled: A Brief Guide to the Standard Object Modeling Language, Addison-Wesley, 1997. 中文版：《UML 精粹--标准对象建模语言简明指南》，徐家福译，清华大学出版社，2002 年 6 月。
- Martin Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.
- Joshua, Kerievsky Refactoring To Patterns, 2002.
- Gamma, E. Helm, R. Johnson and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995. 中文版：《设计模式：可复用面向对象软件的基础》，李英军等译，机械工业出版社，2000 年 9 月。
- Deepak Alur, John Crupi, Dan Malks, Core J2EE Patterns: Best Practices and Design Strategies, Prentice Hall PTR, 2001. 中文版：《J2EE 核心模式》 牛志奇等译，机械工业出版社，2002 年。

火龙果•整理 uml.org.cn

- Stephen R.Palmer,John M.Felsing, A Practical Guide to Feature-Driven Development, Prentice Hall PTR, 2001. **中文版：《特征驱动开发方法原理与实践》** Stephen R.Palmer, John M.Felsing **熊焕宇译，机械工业出版社** 2003 **年** 4 **月。**
- Ken Auer,Roy Miller, Extreme Programming Applied:Playing to Win, Addison-Wesley, 2001. **中文版：《应用极限编程--积极求胜》， 唐东铭译，人民邮电出版社，**2003 **年** 4 **月。**
- William C. Wake, Extreme Programming Explored, Pearson Education, 2001. **中文版：《探索极限编程》， 郑荣林译，人民邮电出版社，**2002 **年** 6 **月。**
- Kent Beck,Martin Fowler, Planning Extreme Programming, Pearson Education, 2001. **中文版：《规划极限编程》， 曹济译，人民邮电出版社，**2002 **年** 6 **月。**
- Giancarlo Succi, Michele Marchesi, Extreme Programming Examined, Pearson Education, 2001. **中文版：《极限编程研究》， 张辉译，人民邮电出版社，**2002 **年** 6 **月。**
- Kent Beck, Extreme Programming Explained:embrace change, 2001. **中文版：《解析极限编程-拥抱变化》， 唐东铭，人民邮电出版社，**2002 **年** 6 **月。**
- Ron Jeffries,Ann Anderson,Chet Hendrickson, Extreme Programming Installed, Pearson Education, 2001. **中文版：《极限编程实施》， 袁国忠译，人民邮电出版社，**2002 **年** 6 **月。**