大道至简

软件工程实践者的思想

（电子版·第三版）

周爱民（Aimingoo） 著

2017.05.03

内容提要

本书提出了审视软件工程的全新视角和软件工程的体系模型(EHM，软件工程层状模型)。本书用非工程的方式重新解析软件工程现象，全面、细致而深刻地分析了工程中各个环节的由来、价值及其内在关系，综合论述开发、工程二者的现状。全书语言轻快，可读性强，薄且有味。.

本书是在“思想方法学”这一软件工程尚未涉足过的领域中的实习之作。作者亲历国内软件工程的英雄时代、泡沫时代，从失败中醒觉而创建独特的思考方法，对软件开发、工程中的现状深刻反思。本书是第一本讨论软件工程思想本源的书籍，也是第一本从工程实践出发溯源而论的佳作。

序

2004 年 11 月初爱民(Aimingoo)第一次把他的书稿给我，我翻看了一下，第一反应讲的是感想。这不错，在技术界就是需要有真正实践经验的专家把他的思考和心得与我们分享。Aimingoo 在 Delphi 领域颇有名气，其技术钻研的深度直达系统核心层，从其著作《Delphi 源代码分析》可见一斑。不过接下来第二反应就是太薄了，能不能加厚啊。比如说这些感悟都是有其来源的，可以把实际案例啊，背景故事啊都加上。不然太薄了，出版社没有办法出版啊。——国家对于出版的书号是有严格控制的，所以书号是有成本的。一本讲技术高端的图书销量肯定是有限的，以现实情况而言，如果很薄定价就只能比较低，成本无法回收。而且内容只是心得，没有案例，读起来也很硬，对读者的要求也很高，销量可能就更少了。

爱民听完我的意见，还是坚持这本书就是这样的风格。出厚书违背了他的本意，要不然怎么叫“大道至简”。书稿在 2005 年 3 月杀青后，我从 7 月开始在《程序员》上陆续选择其中的三章发表，看看读者的反馈。不过限于篇幅，删掉了一些内容，不能完整体现出作者系统思考的脉络，也比较遗憾。

2005 年 11 月爱民跟我讨论到即使没有出版社愿意出版印刷，也要

把他的作品用电子版问世，并邀我作序。我十分感慨，在这个浮躁功利的社会，难得还有这样的朋友。

现在，我又仔细从头到尾读了一遍。很多作者写书是为厚而厚，大部分内容都是水分，作者原创经验精华只有很少，甚至没有。而这本书是作者从事十年开发工作的总结，虽然不厚，却闪烁着独立思考的光芒。

世界“虽变化万端，而理为一贯。”作者在软件开发一线浸淫近十年，回头思考何为开发的本源？这些理论、方法的本质为何？粗粗一看，这些道理稀松平常，专家教授无数著作早就谈过，还用作者来写吗？其实不然，理论都是从实践而来，但我们学习软件开发的时候，是先掌握这些专家总结的果实，而不是探求本源，所谓“知其然而不知其所以然”。这些道理看似都知道，但却没有真正体会上身，在实践中最重要的去应用这些道理，而不是方法。

大多数人看书都希望学到一些招数、方法，能尽快在工作中用上，这是不错。但要想真正达到更高境界，就必须明白背后的道理。真正的专家是从根上解决问题的，所以大物理学家杨振宁在北京大学针对本科生讲物理学，讲得深入浅出，大受欢迎，就是因为杨先生可以从历史本源来剖析物理定律公式。

只有招数，不明道理，碰到变化的情况，就束手无策了。而在软件开发中，每个团队、每个项目都不是尽然相同的。明白道理，才能知变通之道。

这本小书不是一本教你项目管理、软件工程或者编程技巧的书籍，他是一本闪烁思考光芒的技术散文集。我衷心祝愿这本书的读者，能把这本书当作一位朋友的思考，一位朋友的总结，来参照自身，

这样就会有收获，有想法了。

我也和爱民建议，这本书的很多主题还可以展开，无论是批评，还是讨论，只要有兴趣的朋友，可以给爱民，我或者《程序员》杂志社写信，我们诚恳邀请各位来共同思考，共同把实践经验与大家分享，这样意义也就更大了。

期望大家的参与，谢谢。

蒋涛　　2005.11 月
jiangtao@csdn.net

关于本电子版

决定发布这本书的电子版（第二版）[1]，这与它的姊妹篇《大道至易——实践者的思想》是有些关系的。

几个月前，图灵出版的编辑给我发来邮件，问我是否愿意出版新书《大道至易》的电子版，以向读者提供在手机、平板上的付费阅读。但是，当时《大道至易》上市不久，难道纸质版的销售还没展开就要出电子版本吗？再加上潜在的盗版威胁，我一时间很有些犹豫。

然而，我最终决定应该发布《大道至易》的电子版[2]。我在给图灵编辑的信中写到：

> 原因很简单，为什么七年前我因为《大道至简》过薄不能出版就可以发布电子版，而今天《大道至易》能出版便不发布电子版了呢？我想，无论是哪种方式出版，作者的心态应放在"让读者可读"上面，而不是放在"怕读者盗版而不做什么事"上面。
>
> 所以我很坚定地支持《大道至易》电子版的发布。

---

[1] 本书（第三版）与第二版在内容上是一致的，只是版式和发布方式有所区别。

[2] 在 2012 年《大道至易》一书与纸质书同时发行了电子版。在 2017 年，又与图灵出版社协商，采用完全开放共享的协议形式，以电子版的方式发布了《大道至易（第二版）》。

由此，我也想到了这本七年前发布的小书《大道至简》。事实上它在随后的几年中，已经由电子工业出版了纸质版本，算上电子版已有四版之多，最新也是最终的版本是“典藏版”，出版于 2012 年 8 月。而在这么长的时间里，电子版一直未有更新，甚至连勘误也没有做过。作为出书者，我是有愧于读者的。

因此我决定发布《大道至简》的电子版（第二版），其目的在于对原书做完整的修订，并发布平板与阅读器适用的版本。在陈述它的具体修改之前，我必须说明的是：限于版权，我无法提供纸质版本中才有的新章节，只能基于原电子版做修订并添加我个人已公开的相关文章。

电子版（第二版）对所有章节的内容进行了文字校订，与其纸质版本的最终内容保持了一致。所有漫画采用原电子版本（在此感谢漫画作者邵荣国、明明夫妇），而并未附录“愚公移山”故事，以及全套四格漫画——在这一点上有些遗憾，因为这是书中一条相当重要的暗线。

电子版本未收录纸质版本（参考典藏版）如下章节：

- 第 6 章，谁是解结的人（讨论团队与管理）
- 第 8 章，你看得到工具的本质吗（讨论工具与工匠思维）
- 第 10 章，具体工程（讨论具体工程的思想与基本方法）
- 第 4 章第 3 节，沟通的三层障碍
- 第 9 章第 5 节，审视 AP 和 XP
- 第 11 章第 7 节，细解“法”与“式”

其中的第 10 章的前三节主要取材于我的博客文章《杀不死的人狼》，因此本电子书收录了后者并做了适当的编排。

本电子版为最终版本。谢谢与他一路行来的朋友们，我期待与读者们的交流，并由衷地感谢大家的关爱。

希望您喜欢这本小书。

周爱民　　2012 年 12 月 15 日
aiming@gmail.com

电子版发布前言

我终于决定发布这本书的电子版了。

完成这本书的时候，我已经在这个行业里做了十年了。我对自己这十年来的经历做过两次回顾。第一次是关于技术的，这造就了那本《Delphi 源代码分析》，是讨论开发的技术和方法细节的。第二次就催生了这本《大道至简》，讨论的则是工程、管理中的思想。

我其实是很希望这本书能放在读者的书架案头，而不是放在电脑的某一个目录中。因为它应当是一本可以阅读和品味的书，而不是在电脑中备查的技术资料。

然而，我在决定担任这家公司的软件架构师的同时，我就意识到，我没有足够的精力来运作这本书。——我的意思是如果要把他做成纸质的书的话，我没有足够的精力。

出版商是要寻求利润的。——于此，我一早就知道。但我从来不知道：到底一本书簿一点或者厚一点，哪个会让出版商更有利润。

我只想写一本“阐明软件工程的思想核心”的书。这本书要很容易就读明白，还要很容易就想通，还要很容易就知道：工程其实很简单，

只是大家把它做复杂了。

然而问题是：我把它写得太简单了，以至于只写了 110 页，就没有必要再写下去了。

我当然可以把一本书写得很复杂，或者很厚。这很容易，就如做 Coder 一样：把代码写烂或者写乱都很容易，要想写得简洁却远非易事。

代码写得太简洁，老板会认为你在偷懒；书写得太薄，出版社就不愿意出。我看来是忘掉了侯捷先生说过的“买书如买纸”，以书的厚薄来论价值的故事。

忘掉了就忘掉吧。好的一面是现在书变成了电子版，大家终于可以读到它了。不好的呢？我想大概不要钱的东西很难得到珍视吧：如果下载这本书只是因为收集，而不是阅读，那会是让我感到比讨论“买书如买纸”这样的事更为难过的。

好吧。希望你能象对待纸质书籍那样来阅读这本《大道至简》。放心，我并不介意你把它打印出来放在床头。

周爱民　　2005 年 10 月 14 日
aiming@gmail.com

目录

第 1 章 编程的精义

> “虽我之死，有子存焉；子又生孙，孙又生子；子又有子，子又有孙。子子孙孙，无穷匮也。而山不加增，何苦而不平？”
>
> ——《愚公移山》，《列子 · 汤问篇》

第 1 节 编程的精义

仅仅就编程序来说，实在是一件很简单的事，甚至可以说是一种劳力活。两千年前的寓言，已经成就了一位工程名家：愚公。这位名家的身上，浓缩了项目组织者、团队经理、编程人员、技术分析师等众多角色的优秀素质。他的出现，远远早于计算机发展的历史，甚至早于一些西方国家的文明史。

从《汤问篇》中所述的愚公移山这一事件里，我们看到了原始需求的产生：

> “惩山北之塞，出入之迂”

也看到了项目沟通的基本方式：

> “聚室而谋曰”

然后，我们还看到愚公确定了这个项目的目标：

> “毕力平险，指通豫南，达于汉阴”

并通过研讨，择定了一个井然有序的、可以实现的技术方案：

> “扣石垦壤，箕畚运于渤海之尾”。

在这个项目中，动用了三名技术人员和一名工程管理人员：

> “（愚公）率子孙荷担者三夫”

并获得了一名力量较弱，但满富工作激情的外协：

> “邻人京城氏之孀妻，有遗男，始龀，跳往助之”。

基本上，这已经描述了“愚公移山”整个工程的概况。接下来，我们应该注意到愚公作为编程人员的基本素质。在与“河曲智叟”的对答中，他叙述了整个工程的编程实现：

- “虽我之死，有子存焉”，这里描述了可能存在的分支结构，即“IF”条件判断；
- “子又生孙，孙又生子；.....子子孙孙，无穷匮也”，这里描述了完成这个工程所必需的循环结构。

作为优秀的程序分析师，愚公论述了这个循环的可行性：由于“山不加增”，所以条件“山平”必将成立（“何苦而不平”），所以这不会是一个死循环。

在愚公的论述中，我们看到了编程的根本：顺序、分支和循环。庞大若“愚公移山”这样的工程，都是可以通过这样简单的编程来实现的。这，就是编程的精义了。

第 2 节 能不能学会写程序的问题

我经常会被人问到“（我）能不能学会写程序”这样的问题。

这个问题由来已久。上溯十余年，程序员还是很少有人从事的职业。听说的人少，真正了解的人也不多。而当一个程序软件被装在计算机里并开始运行时，人们便开始惊讶于程序员的厉害。所以“能不能学会写程序”甚至成了一些人对自己的智力考评，所以便有人向我这样发问。

愚公都能明白的编程精义，那些向我发问的智叟们又怎么会不明白呢？

所以除了先天智障或后天懒惰者，都是可以学会写程序的。如果你能确信，自己知道在早上起床后：

- 如果天冷则先穿衣服后洗漱;
- 如果天热则可反之;
- 日复一日直到死亡。

那么你就可以开始编程了。甚至，如果你认为以下条件成立：

- 如果有类似于生病、不能行动，以及意外的紧急事件，则当日可以略过。

那么你就可以开始向程序设计师发展了。

因为你已经具备了一项常人不具备的基本素质：折中。

第 3 节 程序 = 算法 + 结构

编程作为一种行为时，我们只需要知道其逻辑方法就可以了。所谓编程实际上就是把一件事情交给计算机去做，你认为这件事该如何做，就用“程序语言”的形式描述给计算机。如果你原本就不明白如何去做，那么你也不要期望计算机去理解你想要做什么。

所以编程的第一要务是先把事情分析清楚，把事件先后的逻辑关系和依赖关系搞清楚，然后再去写代码实现。一接到任务就开始 Coding 的程序员，通常就是加班最多的程序员。

记住：积极工作和勤于思考都要占时间。

第一个完成关于编程本质思考的人，提出了一个公式“程序=算法+结构”[③]。这个公式的精彩之处，在于它没有任何的地方提及代码。甚

[③] 提出这个公式（Algorithms + Data Structures = Programs）的人是被称做“Pascal 语言之父”的瑞士计算

至可以说，在这个公式里，代码是不存在的。

存在的只是思想。

算法是对一个程序的逻辑实现的描述，而结构是逻辑实现所依附的数据实体。只要开发人员将这个程序的算法设计出来，并把结构描述出来，那么程序就定型了。剩下的事，简而言之，就是劳力活。

在计算机专业所学的课程中，同时讲述算法和结构的只有“数据结构”。现在，请你放下手边这本书，再去读读被你扔到不知哪个角落的《数据结构》。请仔细看看，你将发现，在所有的算法描述中，有且仅有顺序、分支和循环这三种执行逻辑。简单如顺序表，复杂如树、图，它们的算法都是用这三种执行逻辑来描述的。

机科学家尼古拉斯•沃思（Niklaus Wirth）。这个公式是他的一本书的名字，他因为提出“结构化程序设计”的概念而获得 1984 年的图灵奖。正是他的学生菲力浦•凯恩（Philippe Kahn）创建了 Borland 公司，才有了后来的 Turbo Pascal 和 Delphi。

第 4 节 语言

当你熟悉了一门语言之后，你会发现，编程语言只有喜欢与不喜欢的问题，没有会不会的问题。任何一门语言，你都可以在两周内掌握并开始熟练编程。因为任何一门语言，它们的底层函数库都是那样地相似，它们的 API 都是那样地依赖于操作系统。A 语言里有的，B 语言里基本也都有。

通常，语言的差别主要表现在适用范围上。一些语言适合做数值处理，小数点后可以精确到原子级，而小数点前则可以表达到宇宙之无穷；另一些语言则适合做图形处理，它的底层函数库可以比其他语言快十倍或数十倍；还有一些语言则适合做网页，要用它来做一个通讯簿软件都将是史无前例的挑战。

成天讨论这门语言好，或者那门语言坏的人，甚至是可悲的。既悲其一叶障目，更悲其大愚若智的自得心态。

第 5 节 在没有工程的时代

在没有工程的时代，上面所说的就是一个程序员的全部。他们掌握了一门语言，懂得了一些生活中最常见的逻辑，他们用程序的方式思考和学习了一些算法，并根据前人的经验，把这些算法运行在一些数据结构之上。最后，我们就看到了他们写的程序。

在没有工程的时代，出现了非常非常多的人物。其中，有算法大师，有游戏大师，有语言大师，有挣钱的大师.....

唯独，没有工程大师。嗯，可以理解嘛，那是没有工程的时代。好蛮荒，好远古的。

第 2 章 是懒人造就了方法

> “僰道有蜀王兵蘭，亦有神作大滩江中。其崖嶄峻不可破，（冰）乃积薪烧之。”
>
> ——《华阳国志》

第 1 节 是懒人造就了方法

战国时期的李冰凿了一座山。

《史记》中说李冰在成都做太守的时候凿出了离堆。一种说法是他将都江堰附近的玉垒山凿了一个叫宝瓶口的大口子，而凿的石头就堆成了离堆。另一说法，则是李冰的确凿了一座崖，但是在沫水，亦即是今天的大渡河。

在哪里凿的山，是史学家都说不清楚的事。但的确凿了一座山，而且方法是“（因）其崖嶄峻不可破，（冰）乃积薪烧之”。

我们已经看到事物的进化了。《列子•汤问篇》里的愚公要“碎石击壤”，而李冰就已经懂得“积薪烧之”了。

会有人说愚公是“碎石”，但史书中并没有说他究竟是零敲碎打呢，还是用火来烧爆掉的。但想想在那个时代，如果有人懂得了烧石头这个方法，哪有不立即载文志之，永世传承的。

再说了，愚公嘛。愚者怎么会呢？这还需要分析吗？需要吗？

所以愚公会凿，而李冰会烧。那李冰又是为什么会用“烧”这种方法来碎石的呢？如果李冰也像愚公那样日复一日地督促着他的团队凿石开山，那他一定没有时间来学习、寻找或者观察；当然也不会发现“烧”这种方法可以加快工程进度，使得一大座山在短时间内就被哗啦哗啦地“碎”掉了。

要知道李冰的团队可是成百上千人，要修堰筑坝，要“凿离堆”，当然还要吃喝拉撒睡。所以李冰如果忙起来的话，他必然是“受命以来，

夙夜忧叹”，必然食难下咽，睡无安枕。反之，李冰一定是个闲人，可以闲到没事去看火能不能把石头烧爆。

在这么大的工程里，如果有一个人会闲到看火烧石头，那他一定很懒。那么多事堆着不去做，去看烧石头，你说他不是懒是什么？

正是一个懒人造就了“烧石头”这个“碎石”的方法。愚公太勤快了，勤快到今天

可以比昨天多凿一倍的石头。或许在愚公的项目计划案的首页就写着朱批大字：“吾今胜昨倍许，明胜今倍许，而山不加增，何苦而不快。”但是越勤快，愚公将越没有机会找到更快的方法。

人的精力终归是有极限的。提出新的“方法”，解决的将是影响做事成效的根本问题。愚公可以多吃点饭，多加点班，但突破不了人精力的极限。

记住，在两千年前的某一天，闲极无聊的李冰下厨给夫人炒了一个小菜，他突然发现垒灶的鹅卵石被烧得爆裂开来，遇水尤甚。从此《史记》上就记下了“蜀守冰，凿离堆”，而《华阳国志》上则记下了他做这件事的方法：积薪烧之。

第 2 节 一百万行代码是可以写在一个文件里的

早期的程序，都是将代码打在穿孔纸带上，让计算机去读的。要让计算机读，纸带当然是连续的，这无须多讲。其实我没有那样写过程序，其中的苦楚我也不知道。

后来有了汇编语言，可以写一些代码了。这时的代码是先写在文本文件里，然后交给编译器去编译，再由链接器去链接，这样就出来

了程序。

第一个写汇编的人，写的可能是有名的“Hello World”程序[④]，那个程序写在一个文件里就行了。后来就成了习惯，大家都把代码写到一个文件里。在早期的汇编语言里，GOTO 语句用得非常非常频繁，将一条语句 GOTO 到另一个文本文件里去，既不现实也不方便。所以大家习以为常，便统统地把代码写到一个文件里[⑤]。

再后来出现了高级语言，什么 C 呀，Pascal 呀之类的。既然大家已经形成习惯了，所以很自然地会把一个程序写到一个文件里。无论这个程序有多大，多少行代码，写到一个文件里多方便呀。

直到如今，语言发展得更高级了。可是程序员的习惯还是难改，一旦得了机会，他们总还是喜欢把代码写到一个文件里的。

好了，有人说我是想当然耳。嗯，这当然是有实据的。记得 Delphi 1.0 发布的时候，全世界一片叫好声。连“不支持双字节”这样的大问题，都不影响它在华语地区的推广。然而不久，爆出了一个大 BUG！什么大 BUG 呢？Delphi 1.0 的编译器居然不支持超过 64KB 的源代码文件！

这被 Fans 们一通好骂。直到我用 Delphi 2.0 时，一个从 Visual Basic 阵营转过来的程序员还跑过来问我这件事。好在 Delphi 2.0 改掉了这个 BUG，这让我很有面子，好一阵风光。

64KB 的文件是什么概念呢？

---

④ 早期的计算机被用来完成复杂的科学运算，因此不可能真的有人无聊到去写 Hello World。

⑤ 汇编语言是面向特定硬件系统的一种助记符，所以只应该存在指令（而不包括语句）。因此有没有 GOTO，取决于硬件指令系统的设计，而与这种汇编语言没什么关系。

1 行代码大概（平均）是 30 字节，64KB 的源代码是 2184 行，如果代码风格好一点，再多一些空行的话，差不多也就是 3000 行上下。

也就是说，在 Delphi 1.0 的时代（以及其后的很多很多时代），程序员把 3000 行代码写到一个文件里，是司空见惯的事了。如果你不让他这样写，还是会被痛骂的呢。

所以呢，按照这一部分人的逻辑，一百万行代码其实是可以写在一个文件里的。不但可以，而且编译器、编辑器等也都必须予以支持。这才是正统的软件开发。

勤快的愚公创造不了方法。这我已经说过了。对于要把“一百万行代码写到一个文件里”，并且查找一个函数要在编辑器里按 5000 次 PageUp/PageDown 键的勤快人来说，是不能指望他们创造出“单元文件（Unit）”这样的开发方法来的。

然而单元文件毕竟还是出现了。这个世界上，有勤快人就必然有懒人，有懒人也就必然有懒人的懒方法。

有了单元文件，也就很快出现了一个新的概念：模块。把一个大模块分成小模块，再把小模块分成更细的小小模块，一个模块对应于一个单元。于是我们可以开始分工作了，一部分人写这几个单元的代码，另一部分则写那几个。

很好，源代码终于可以被分割开来了。结构化编程的时代终于开始了，新方法从此取代了旧方法。而这一切应当归功于那个在按第 5001 次 PageDown 键时，突然崩溃的程序师。他发自内心地说：“不能让这一切继续下去了，我一定要把下一行代码写到第二个文件

里去。我发誓，我要在编译器里加入一个 Unit 关键字。”[⑥]

第 3 节 你桌上的书是乱的吗？

我曾经在一所电脑培训学校与学生座谈时，被一个学员问道：“为什么我学了一年的编程，却还是不知道怎么写程序呢？”

我想了想，问了这个学员一个问题：“你桌上的书是乱的吗？”他迟疑了一下，不过还是回答：“比较整齐。”我当时便反问：“你既然知道如何把书分类、整整齐齐地放在书桌上，那怎么没想过如何把所学的知识分类、归纳一下，整整齐齐地放在脑子里呢？”

如果一个人学了一年的编程，他的脑袋里还是晕乎乎的，不知道从哪里开始，也不知道如何做程序。那想来只有一个原因：他学了，也把知识学进去了，就是不知道这些知识是干什么的。或者说，他不知道各种知识都可以用来做什么。

其实结构化编程的基本单位是“过程（Procedure）”，而不是上一小节说到的“单元（Unit）”。然而在我看来，过程及其调用是 CPU 指令集所提供的执行逻辑，而不是普通的开发人员在编程实践中所总结和创生的“方法”。

这里要提及 CPU 指令集的产生。产生最初的指令集的方式我已经无可考证，我所知道的是，CISC 指令集与 RISC 指令集之争在 1979 年终于爆发。前者被称为复杂指令集，然而经过 Patterson 等科学家的研究，发现 80%的 CISC 指令只有在 20%的时间内才会用到；更进一步的研究发现，在最常用的 10 条指令中，包含的流程控制只有

[⑥] Turbo Pascal 3.0 中才开始有了 Uses 和 Unit 关键字。在 ANSI Pascal 标准里并没有它。

“条件分支（IF...THEN...）”[7]、“跳转（JUMP）”和“调用返回（CALL/RET）”.....

RISC（精简指令集计算机）由此成为一种可能的发展方向，进而演变成一场长达 RISC（精简指令集计算机）由此成为一种可能的发展方向，进而演变成一场长达 20 年的指令战争。而动摇了 CISC 指令集地位的方法，就是分类统计。

正如 CISC 指令集搅乱了一代程序设计师的思路一样，大量的知识和资讯搅乱了上面向我提问的那位学员的思想。他应该尝试一下分类，把既有的知识像桌子上的书一样整理一下，最常用的放在手边，而最不常用的放在书柜里。如果这样，我想他应该在九个月前就开始写第一个软件产品了。

你桌上的书还是乱的吗？

第 4 节 我的第一次思考：程序=算法+结构+方法

我对程序的本质的第一次思考发生在几年前。那时我与 Soul（王昊）有一次网上对话。

Soul 是 DelphiBBS 现任的总版主，是我很敬重的一位程序员。那时我们正在做 DelphiBBS 的一个“B 计划 II”，也就是出第二本书。他当时在写一篇有关“面向对象（OOP）”的文章，而我正在写《Delphi 源代码分析》——在这本书的初期版本里，有“面向对象”的内容。

[7] 在 X86 系统中，循环是用条件分支（或循环指令）来实现的，而且条件分支指令并不是 IF...THEN...，这里用这两个关键字，仅用于说明问题。

我们的对话摘要如下[8]：

> Soul：我在写书讨论“面向对象的局限性”。
> 我： 嗯。这个倒与我的意见一致。哈哈哈。
> 我： “绝对可以用面向过程的方法来实现任意复杂的系统。要知道，航天飞机也是在面向过程时代上的天。但是，为了使一切变得不是那么复杂，还是出现了‘面向对象程序设计’的方法。”
> 我： ——我那本书里，在“面向对象”一部分之前的引文中，就是这样写的。
>
> Soul：现在的程序是按照冯•诺伊曼的第一种方案做的，本来就是顺序的，而不是同步的。
> CPU 怎么说都是一条指令一条指令执行的。
> 我： 面向过程是对“流程”、“结构”和“编程方法”的高度概括。而面向对象本身只解决了“结构”和“编程方法”的问题，而并没有对“流程”加以改造。
> Soul：确实如此。确实如此。
>
> 我： 对流程进一步概括的，是“事件驱动”程序模型。但这个模型不是 OO（面向对象）提出的，而是 Windows 的消息系统内置的。所以，现在很多人迷惑于“对象”和“事件”，试图通过 OO 来解决一切的想法原本就是很可笑的。
> Soul：我先停下来，和你讨论这个问题，顺便补充到书里去。我：如果要了解事件驱动的本质，就应该追溯到 Windows 内核。这样就涉及线程、进程和窗体消息系统这些与 OO 无关的内容。所以，整个的 RAD（快速应用程序开发）编程模型是 OO 与 OS（操作系统）一起构建的。现在很多的开发人员只知其 OO 的外表，而看不到 OS 的内核，所以也就总是难以提高。
> Soul：OO 里面，我觉得事件的概念是很牵强的，因为真正的对象之间是相互作用的，就好像作用力和反作用力，不会有个“顺序”的延时。
> 我： 应该留意到，整个的“事件”模型都是以“记录”和“消息”的方式来传递的。也就是说，事件模型停留在“面向过程”编程时代使用的数据结构的层面上。因此，也就不难明白，使用或不使用 OO 都能写 Windows 程序。
> 我： 因为流程还是在“面向过程”时代。

[8] 这段对话的确很长。如果你不是非常有经验的程序员，那么不能完整地阅读和理解这段文字是很正常的。部分读者甚至可以跳过这段引文，直接阅读后面的结论。

Soul：所以所谓的面向对象的事件还是“顺序”的。所以我们经常要考虑一个事件发生后对其他过程的影响，所以面向对象在现在而言还是牵强的。

我：　如果你深入 OS 来看 SHE（结构化异常处理），来看 Messages（窗体消息），就知道这些东西原本就不是为了 OO 而准备的。面向对象封装了它们，却无法改造它们的流程和内核。因为 OO 的抽象层面并不是这个。

我：　事件的连续性并不是由某种编程方法或者程序逻辑结构所决定的。正如你前面所说的，那是 CPU 决定的事。

Soul：比如条件选择，其实也可以用一种对象来实现，而事实却没有。这个是因为 CPU 的特性和面向对象太麻烦。

我：　可能，将 CPU 做成面向对象的可能还是比较难以想象和理解。所以 MS 才启动.NET Framework。我不认为.NET 在面向对象方法上有什么超越，也不认为它的 FCL 库会有什么奇特的地方—除了它们足够庞大。但是我认为，如果有一天 OS 也是用.NET Framework 来编写的，OS 一级的消息系统、异常机制、线程机制等都是.NET 的，都是面向对象的。那么，在这个基础上，将“事件驱动”并入 OO 层面的模型，才有可能。

Soul：所以我发觉面向对象的思维第一不可能彻底，第二只能用在总体分析层上。在很多时候，实质上我们只是把一个顺序的流程折叠成对象。

我：　倒也不是不可能彻底。有绝对 OO 的模型，这样的模型我见过。哈哈，但说实在的，我觉得小应用用“绝对 OO”的方式来编写，有失“应用”的本意。我们做东西只是要“用”，而不是研究它用的是什么模型。所以，“Hello World”也用 OO 方式实现，原本就只是出现在教科书中的 Sample 罢了。哈哈。

Soul：还有不可能用彻底的面向对象方法来表达世界。因为这个世界不是面向对象的，是关系网络图，面向对象只是树，只能片面地表达世界。所以很多时候面向对象去解决问题会非常痛苦。所以编程退到数据结构更合理，哈哈。

我：　如果内存是“层状存取”的，那么我们的“数据结构”就可以基于多层来形成“多层数据结构”体系。如果内存是“树状存取”的，那么我们当然可以用“树”的方式来存取—可惜我们只有顺序存取的内存。

我：　程序＝数据＋算法

—这个是面向过程时代的事。

程序＝数据＋算法＋方法

—在 OO 时代，我们看到了事件驱动和模型驱动，所以出现了“方法”问题。

Soul：我的经验是：总体结构→面向对象，关系→数据结构，实现→算法。

> Soul：看来我们对面向对象的认识还是比较一致的。

我第一次提到我对程序的理解是“程序=数据+算法+方法”，就是在这一次与 Soul 的交谈之中。这次交谈中的思考仍有些不成熟的地方，例如我完全忽略了在面向过程时代的“方法”问题。实际上面向过程开发也是有相关的“方法”的。

所谓“面向过程开发”，其实是对“结构化程序设计”在代码阶段的一个习惯性的说法。而我忽略了这个阶段的“方法”的根本原因，是即使没有任何“方法”的存在，只需要“单元（Unit）”和“模块（Module）”的概念，在面向过程时代，一样可以做出任意大型的程序。在那个时代，“方法”问题并不会像鼻子一样凸显在每一个程序员的面前。

在面向过程开发中，“过程（Procedure）”是由 CPU 提供的，“单元（Unit）”则是编译器提供的（机制）。程序员无须（至少不是必须）再创造什么“方法”，就可以进行愚公式的开发工作了。

如果不出现面向对象的话，这样伟大的工程可能还要再干一百年.....

而与“面向对象”是否出现完全无关的一个东西，却因为“过程”和“单元”的出现而出现了。这就是“工程（Engineering）”。

第3章 团队缺乏的不只是管理

> “言人三为众，虽难尽继，取其功尤高者一人继之，於名为众矣。”
>
> ——《汉书 · 高惠高后文功臣表序》颜师古注

第 1 节 三个人的团队

《汉书》中说“言人三为众”[⑨]，是指三个人就算得上是“众”了。这里的“众”应该理解成一个群体，亦或者说是一个团队。

团队至少是以三个人为规模的。这有其合理性。为什么呢？首先一个人算不得团队，那是个体。两个人则互相支撑，古文中“从”字是二人互立，就是这个意思。然而二人互立并不算团队，因为没有监督。三个人便可以构成团队，这样便有了团队的一些基本特性：主从、监督和责任。

一个人的开发行为可以成功，这取决于个人努力。大家熟知的 KV100、KV200 系列反病毒软件，最早就是王江民先生一个人做出来的。二人小组如果能相互支撑，那也是可以获得成功的，同样作为反病毒软件的 AV95 在 1995 到 1997 年成功占据反病毒软件市场之一隅，就是周辉和刘杰先生两个人的作品。

然而到了三个人的时候呢，就得选个领导了。三国曹魏时的孟康在为《汉书》作注时曾写道：“取其功尤高者一人继之，于名为众。”意思就是功高者代替群体受功。古人的受功当然包括封侯晋爵，因此这便似乎成了惯例而推广开来，功劳大的、能力强的便成了团队中的领导角色。

但是在上面的例子中，目的并非要选领导，而是要表彰功绩。项目

[⑨] 片面地理解成“三人为众”是不对的。“三”在这里是虚词，指的是很多人的意思。然而，古人以“三”来泛指很多人或者群体，则是很值得玩味的事。

结束会议上，总经理说：“M 项目完成得很好，小 S 的进步尤其大，他独立完成了全部核心代码的编写，因此月奖加三倍”。看起来奖励丰厚，然而这并不代表下一个项目该让小 S 来做项目经理。

同样，三板斧定了瓦岗寨的程咬金，功高技强，但不是将帅之才。

做管理起码要能承担责任，这是最基本的素质。

春秋时晋国最高司法长官李离，因为听信属下而判案失误，把一个不该处死的人错判了死刑。随后“自拘于廷，请死于君”。晋文公打算追究他属下的过错而免掉他的罪，而李离说：

> “臣居官为长，不与吏让位；受禄为多，不与下分利。今过听杀人，傅其罪下吏，非所闻也。”

随后李离就拔剑自杀了。

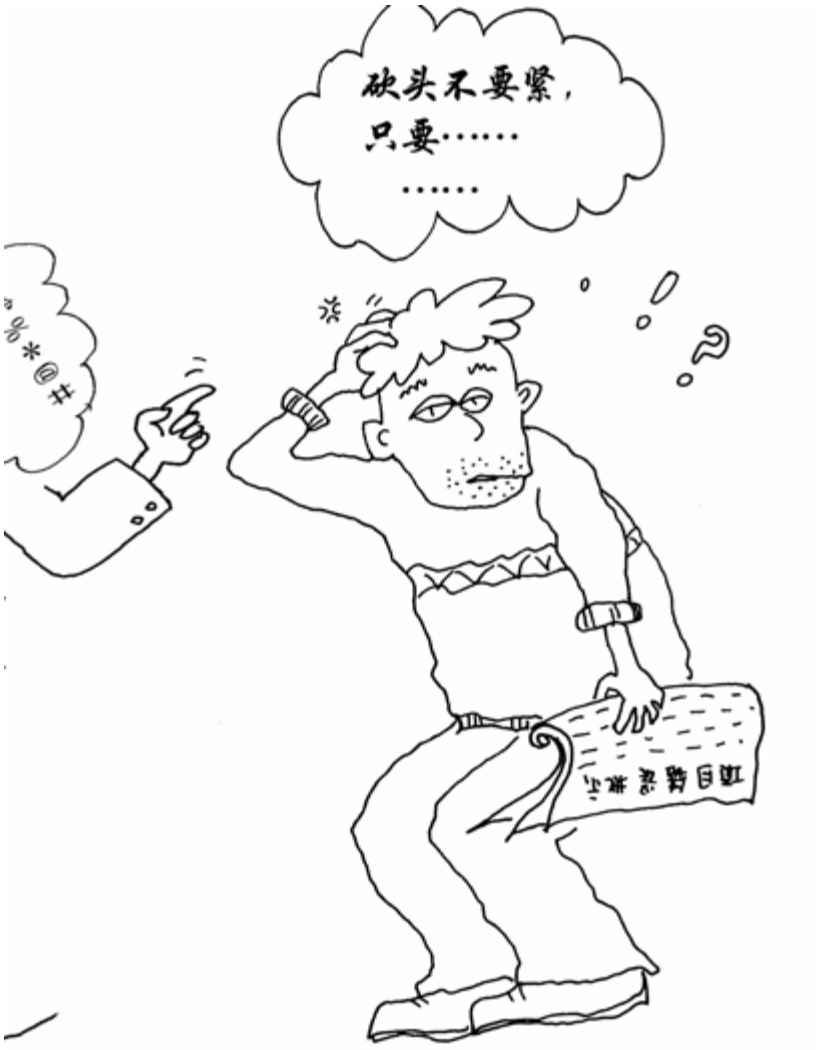
同样的道理，你的项目经理职位又没有让给别人做，你拿的经理级工资又没有分给别人，那项目失败了，你为什么要把责任推到别人头上呢？

三人团队中的那个领导，不是要程咬金一样的牛人，而是要李离一样的死士。不过，项目完成不了，切脑袋的事倒不必做，递交辞呈的那点勇气总是要有的。

第 2 节 做项目 = 死亡游戏?

项目做不成就要掉脑袋，就好比枕着锒刀做程序；如果项目失败就要交辞呈，那可能就从来不会有项目经理。

为什么这么说呢？



从管理的角度来看，项目失败与否与项目经理的经验直接相关。我曾经听过一个来自澳大利亚的讲师说软件工程。她说到项目的成功是由两个方面来评估的：

- 项目完成质量；
- 项目完成时间。

由于项目的完成时间是在项目前期对项目工期的设定，因此我问这

个讲师：什么方法能保证预期的工期是正确的，或者说项目是可以按时完成的。

讲师的回答很有意思：经验丰富的工程师能尽可能接近地预估工期，但没有办法保证（预估的）工期是绝对合理的[10]。

那么进一步的推论是，由于没有绝对合理的工期，所以项目的完成时间可能总是被进度变更所修正，所以项目也就总是不能“成功”。

看来外来的和尚（包括尼姑）也未必能比本地的更会念经。在这一点上，来自澳大利亚的讲师，与来自北极的爱斯基摩人（如果他们也念经的话）如出一辙。

项目工期的问题不能解决，就不能保证项目成功。只有经验更加丰富，才更有可能逼近“合理的工期”。因此在这之前，项目经理面临的就是失败。这个失败可能不是由项目经理本身的能力所决定，或者也不是由团队成员的工作所决定，而是在一开始，那份给客户的项目协议就签错了。

因此，项目经理是需要时间来成熟的。他需要机会来承受错误，而不是一开始就享受成功。

第 3 节 做 ISO 质量体系的教训

Y 公司终于在 2001 年发现管理跟不上了，于是开始引进 ISO 质量认证体系，希望通过这个体系来规范管理行为，提高产品质量和对外的竞争力。

[10] 软件工程中有专门的学科来研究项目的工期问题。学者们试图寻找公式来计算项目的复杂度，从而计算出所需的工时和人月。然而在实践中，这被认为是不可行的。

他们做得非常认真，把全公司的人员都调动起来了。他们与每一个员工一起论证质量手册的编订；按照标准软件工程模型进行开发流程的重组；每一份流程相关的材料都约定了格式，并进行归档说明；每一个环节都设定了质量监督员来考核和回顾.....

接下来，他们开始实施。

三个月后，他们发现了一个问题：所有环节的质量监督员是同一个人。这个人没有工程经验，于是他提出来的问题总是得不到工程负责人的确认——很显然，没有工程负责人愿意说自己这里存在问题：有问题就要改，就有可能中断或者重新来过。再者，这位质量监督员也没有管理权限，于是他即使确认了问题，也没有权利要求立即整改，工程负责人随时可以以进度为由搁置这份监督报告。

再两三个月后，他们发现一切如旧，好像工作并没有因为《质量手册》的存在而发生什么变化，在手册上被改造的流程因为人力资源不充分而没有办法运作起来；绝大多数应该书写的材料因为时间不够而被“候补”。

改变最大的是综合部，这里多了一个虚设的机构用于分管 ISO 质量，综合部的经理也变成了分管质量的副总，但又没有因此而多拿一分钱。改变最少的是开发部，其表现为每个人的显示器顶上放了一本质量手册，用来挡住窗口射进来的阳光，以及落向显示器的灰尘。

两年之后，我们一群人来回顾这一次失败时，很多人都说是“体制的问题”，说是原有的公司转型到新的公司，不适合新的公司的管理体制及对管理的要求。

其实这并不十分正确。体制的内涵是分两个方面的，其一是“体”，即“体系”；其二是“制”，即“制度”。“ISO 质量体系”所产生的那份手册只是“制度”。在这份制度的背后，所要求的是对旧有“体系”的改变——旧的公司转型到新的公司，不是搬来一本“管理制度”给每个员

工读一遍就可以了的。

在这一转型期，第一要务是解决“体”——也就是“组织机构建设”的问题。如果把这个问题缩小到开发部门的工程环节，那就是“如何组织开发团队”。只有有了确定的团队模式，才能寻求相应的管理制度，并且才能把这样的制度实施在团队之上。

汉朝的刘向在《新序•杂事二》中记录了一个故事，说是魏文侯出游，见路人把羊皮统子毛向内皮朝外地反穿着，还背着一篓喂牲口的草。文侯奇怪地问他为什么。这个人答道：我爱惜这件皮衣，怕毛被磨掉了。文侯叹道：你难道不知道，如果皮被磨尽了，毛不也就掉光了吗？

皮之不存，毛将焉附。没有确定的组织机构，又如何能指望做出来的管理制度“合用”呢？

Y 公司在 1999 年至 2001 年一直保持着从 K 公司移植过来的组织机构模式和管理模式，两年的组织机构建设的时间被白白浪费了。本来，做 ISO 体系是最后一次弥补组织机构建设的机会，然而在管理者还没有意识到“皮之不存”的时候，Y 公司就连“毛”也都掉光了。

第 4 节 谁动摇了你的制度？

组织模式确定的同时，相应的制度也随之建立。很少有组织几年之后才来补制度的。

然而制度究竟决定了什么呢？我们先来看看，如果员工在工作中出了纰漏，那么：

- 没有制度，你没有办法和依据来惩戒员工，因此是管理者的过失；

- 有了制度而没有惩戒他，是执行者和监督者的过失；
- 一而再、再而三地犯错，又一而再、再而三地被惩戒，那就是教而不改，就真正是员工的品性和素质问题了。

因此，先做制度总是好的。至少在选择做伏剑自刎的李离之前，你还有机会把黑锅扔到出问题的员工头上。

对于一个已经规范管理、体制健全的公司，不容许员工犯错是对的。即使是一次犯错，立即开除也说得过去。但还得是有前提的，这至少包括：

- 员工已经接受过相关的培训，至少是员工规范和技术技能的学习；
- 在该员工之前，相同或相关的错误没有被枉纵。

第一条是人性化的体现。中国人常说不知者不为过：员工不知道，管理者也没有给他知道的条件，那怎么能说是他的过错呢？如果是因为不知道而出了问题，那管理者首先应该自省才是。

第二条则是公平性的体现。不管是针对谁，制度都是一样的，没有情面可讲的，常说的“特殊情况特殊处理”在制度面前行不通。规矩一旦被破坏就形同虚设，反而被员工当做笑柄，用来类比其他的制度。如此一来，整个制度也就离崩溃不远了。反过来，在已经被破坏了的制度面前，若再做杀鸡儆猴状，那猴子是被吓着了，不平声、怨愤声也就跟着出来了。

因此最好的方法是赶紧修订制度，而不是修理人。

所以，在更加普遍的情况中，动摇了制度的人往往不是犯错的员工，而是管理者自己。如果在制度面前，既做得到“人性化”，又做得到“公平性”，那么当管理者的便可以多做几次黑脸的包龙图，而脖子上的脑袋便可以比李离顶得长久一些。

第 5 节 “那我们就开始开发吧”

现在，公司的组织机构和制度建设已经完成了[11]，在这个组织机构里，我们已经有了一个或者多个团队。接下来，我们要真正地开始团队建设了。

这是任务。因为正在读这本书的你，和我一样，是要拉齐七八杆枪，开始做工程的了。而在这一切开始之前、再之前的时间里，我还想知道一件事：你知道如何做工程吗？

我们先来回顾一下。

前一章说的是编程。嗯，那实在是简单的、愚公式的工作罢了。我们先不管愚公们的水平如何，以及够不够勤快，反正，他们会编程就是了。

上一章呢，说的是一部分懒人“创造”或“寻找”到一些编程的方法。这些懒人们可能来源于做得太老的、或者太累的愚公，或者是.....一些看着愚公们着急，又被闲出了毛病的人。反正他们找了一些方法出来，而我们的愚公们也已经学会了这些方法。

现在，有了会（比较快速地）编程的愚公，而且有了公司，我们完成了组织机构建设，我们还找到了一名（或好多名）项目经理，他们一不怕死，二不怕苦.....对了，更为可喜的是我们还有了开发部：对内，我们制定了一套规章制度；对外，我们还拿到了项目单子。

“那我们就开始开发吧”——你就这样跟我说。

---

[11] 这里说制度建设的“完成”是指告一段落，或者说是阶段性的结束。完成，并不等于完善。而完美，则更是无可企及。

很久以前，很久很久以前，人们都是这样做的。拿到项目单子，然后“那我们就开始开发吧”。这样的事出现得很自然，因为积极的愚公们总是有挖山不止的欲望。所以他们一看到项目单子，第一个反应就是：那我们就开始开发吧。

做了这么多年项目，我现在一听到这句话，就哆嗦。

第 6 节 组织的学问：角色

现在先考察一下你的公司，在整个系统里面，有没有这样的人：他既不归任何人管理，也不管理任何人。如果有，那么就早早把他开掉好了。

这样的人在组织机构中是一个盲点，或者空洞。按照我的观点来看，他在组织中不担任任何的角色，既然“不是角色”，那么当然要开掉。

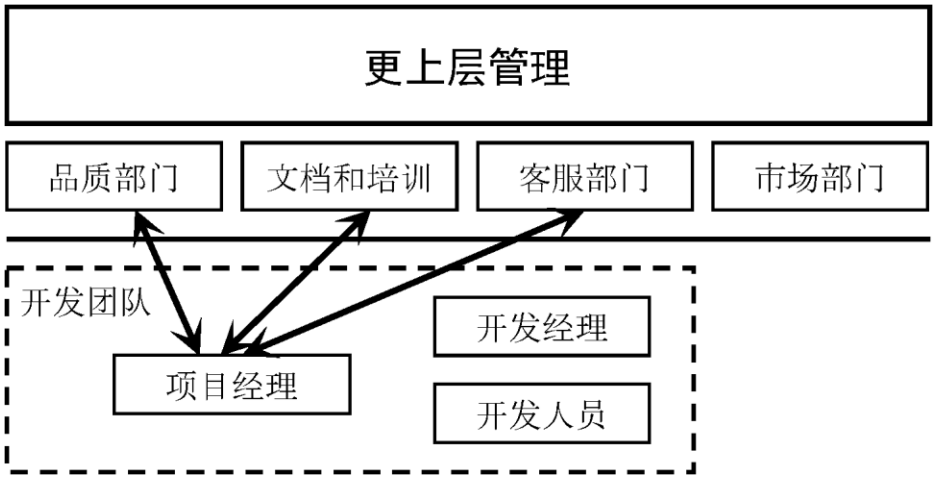
但是，在你做这件事之前，确切地说是在任何错误被归咎于员工之前，管理者应该先想想是不是自己的问题。

是的。你可能很快就发现问题出在了管理者那里。因为管理者没有确定组织机构模式，或者没有为组织中的成员进行角色定位和分工。如果这样，出现“既不能令，又不受命”的人就是必然的了。

同样的道理，在工程开始“做”之前就得先把“角色”确定好——可能部分角色是与既已存在的组织机构相关的，例如“部门经理”和“开发人员”；而有些就需要临时授命。

对于一个项目来说，第一个授命的人当然是“项目经理”。但接下来的事件就要复杂得多了。按照微软的惯例，授命项目经理的同时，会有“产品经理”、“开发经理”、“市场经理”及“文档化和培训负责人”。

这当然不表明至少需要 5 个人才能构成团队，在大量角色从项目团队中抽取与剥离后，我们可以得到一个精减过的团队模型[12]（在后面我会把它叫“R 模型”[13]）：



在保障这样一个组织机构模式的过程中，以下几点内容是需要注意的：

- 如果项目针对直接客户，而且没有产品化的可能性（或必要性），那么可以将与市场（以及市场部门）相关的问题和角色先放在一边。
- 已经存在于开发团队中的成员，不适合在品质部门中兼任角色。
- （在这个模型里）项目经理应致力于减少团队中开发角色与其他部门的沟通，必要时开发经理应该站在开发人员之前进行部门间的交互。
- 品质部门、文档和培训部门以及客服部门应该主要由专职人员构成，尽管

[12] 我非常不情愿给出一个模型来让读者跟随，但如果没有这样的一个模型，我想接下来的讲述可能会令很多人如坠雾里。明确的组织机构，既是团队的关键，也是我们思考问题的基础。

[13] 我试图找一个单词来表现这个模型的简单和粗糙。我得到的一个建议是 Rough（粗糙的）。然而我更愿意溯源到这个单词在古英语中的形态（Ruh），希望我这样一再强调，能让你真正地注意到：“R 模型”是一个原始而且粗糙的东西。

开发人员可以（或者经常会）参与文档、培训和客服工作，但这也通常是他们最不能胜任的角色。

- 这是中小型规模的公司和团队的参考组织机构模型，对大型团队并不适用。

在这个模型中，我们仍然看到了一个至少由三个人构成的团队。其中，在开发经理和开发人员之间，既存在主从关系，也存在协作关系。而项目经理，则在团队中处于领导者、组织者和团队保障者的地位。

如果非要精简到两个角色的团队模式，那么通常是开发经理兼任项目经理。因此这位开发经理一定要能清楚地区分这种双重角色的身份：在任何时候，明确自己是在进行“团队内协作”，还是“团队管理（和组织）”，还是在与“团队外交流”。

如果这个开发经理总是混淆自己的角色，那么，我建议，换人吧。

第 7 节 跟随蚂蚁。但不要栽进蚂蚁洞里。

团队真的需要管理吗？这经常是“经营”开发团队的管理者最容易给错答案的问题。这些管理者兢兢业业，试图细化每一个管理环节，将自己的意愿贯彻到……嗯，CPU 里去。

实际上，开发团队并不需要管理。或者说，在你还没有弄清楚状况之前，不要去管它。

温伯格（Gerald M. Weinberg）在“给软件开发经理的建议”中提到了这样一个问题：开发经理如何面对一个并非由他亲自雇佣的成员组成的团队。温伯格的回答是：

- 与成员面谈，让他们签约受雇于你；
- 或者，解聘他们；

- 再或者，放弃这个职位。

温伯格的意思是“没办法管就不管”。温伯格当然可以有更多的选择，他总可以找到适合自己管的公司。然而目前，你可能是唯一的人选。或者你原本就期待这个角色很久了，当然不能像温伯格一样放弃。

你得找办法来解决团队问题。“签约”这样的事，在大多数环境下是行不通的。要知道，既然他们与公司的签约保证不了工作的质量，同样与你的这份签约也保证不了。协议并不能建立管理者与被管理者的信任，而只是确保了这种关系。

但是你应该相信我，在你接手这个团队之前，上一任经理也确保了这种关系。然而团队失败了，否则不会换做是你。

所以告诉团队成员“现在轮到我管理你们了”，根本就是一句废话。或者在你来之前，他们就已经知道你要来了。

小的时候，我就喜欢观察蚂蚁。我喜欢看它们成群结队地搬着东西穿过小路，或者水沟。我尝试用木棍导引它们改变行动的路线，然而不久之后，它们就会翻过那根木棍，按照既定的路线行进。

禀性难移，要改变一个人都难，何况是改变一个团队的既定习惯。

如果有一群开发人员像蚂蚁一样辛勒地工作着，那么，请先不要打扰他们。你应该跟随他们，看看他们是如何做的。发现规律，分析这个规律的价值，最后再尝试改变他们（的一些负面价值的规律）。

所以你要紧紧地跟随他们——除了一个地方。那地方是你去不得的，那就是蚂蚁洞。

显然，你不是开发者，你是管理人员。所以尽管你也是团队中的角

色，但千万记得离蚂蚁洞远点。你在洞外张望，可以发现问题；你在洞内，就只有做“循规蹈矩”的蚂蚁。

管理者是那个可以在洞外放木棍的人。

第 8 节“什么是增值税发票？”

现在你已经足够细致地观察了你的团队，知道这个团队存在问题，你也知道这必须被改变。当然，你也知道这种改变并不是放一根木棍那么简单。

你已经确定了组织结构，确定了组织中的角色，还有了一个团队（五个？或者十个？）的人。所以作为项目经理，你需要先分工。

在分工之前，那个团队只能算是一个没有组织与合作的群体，所以英文中群是 Group，而开发团队是 Team。

被优先考虑的是弹性分工。每一个人都被要求做一颗革命的螺丝钉，哪里需要哪里拧。所以弹性分工总是被放在企业节省人力资本的第一要务上。然而我们真的会做弹性分工吗？

蚂蚁的分工模式之一就是弹性分工。一只蚂蚁搬着食物往回走时，碰到下一只蚂蚁，会把食物交给它，自己再回头；碰到上游的蚂蚁时，将食物接过来，再交给下一只蚂蚁。蚂蚁要在哪个位置换手不一定，唯一固定的是起始点和目的地。

确定被“弹性分工”的员工需要可以快速地转换到新的角色。但首先要考察的并不是他是否“有能力”胜任这个角色：能力可以通过学习来增强，而角色的转换，则首先是思想的转换。

1997 年，P&J 的公司打算全面拓展市场，我随他一起到了成都。当

时我是开发部的三个主力开发人员之一，因此在原定计划里，我是到成都组建西南区开发部（或技术中心）。然而在两周之后，P&J 发现总公司的运作存在问题，因此他必须回郑州。P&J 决定将成都市场的问题全权交给我，换而言之，我必须出任成都办事处经理。

我对市场一窍不通，也不懂得公司的经营与管理。但很明显，做办事处经理不是做技术，这与我（当时的）个人意愿是相悖的。于是我拒而不受。理由也很充分：我不会做市场。

P&J 用了两天的时间来说服我，直到在临回郑州的前一晚我仍未能接受这个任命。

这时他告诉我：即使是做开发，也是需要了解市场的，你必须知道用户想要什么，你必须理解你的客户。因此你如果想要做一个好的开发人员，你应该正视这次机会。

我沉默了许久。我想明白了两件事：从公司的角度上，我需要接受这个职务；从个人的角度上，我需要接受这个职务。于是，我问了我的第一个问题：“什么是增值税发票？”

P&J 笑了。接下来我们开始讨论经营问题。第二天 P&J 飞回郑州。五个月后我升任西南区总经理，一年后，西南区做到全国六个分区市场中业绩第二。此后我辞职回到郑州，再一次从开发人员做起。

“什么是增值税发票？”意味着从技术到经营的角色转变。这个问题本身带来的并不是能力的提升，但如果我提不出这个问题，我将没有可能理解经营与市场。

尽管弹性分工非常有效，然而真正做弹性分工却并非易事。蚂蚁的角色转换是本能的，而 P&J 却不得不花两天时间来说服我。因而更

应当留意团队成员“自激”式的角色转换，知道他是不是真的想（而不是需要）转变一下角色，这样起码可以省去你两天的工夫。

然而能提问“什么是增值税发票”的愚公毕竟不是太多，大多数时候他们在“箕畚运于渤海之尾”，如果实在闲得厉害，他们可能会去发明翅膀，而不是思考“什么是增值税发票？”

更好的选择是明确分工，而不是弹性分工。你应该明白，重要角色的更替通常是极具风险的，例如项目经理或者开发经理；频繁的开发人员的调度也会直接影响到工程的质量和进度。

如果所有人都在思考“什么是增值税发票”，那么你的组织机构将立即溃散。

因此，明确分工是你的管理职责。做管理≠做伯乐。

第 4 章 流于形式的沟通

> “足下求速化之术，不于其人，乃以访愈，是所谓借听于聋，求道于盲。”
>
> ——唐 · 韩愈《答陈生书》

第 1 节 客户不会用 C，难道就会用 UML 吗？

我们总是要先接触客户的。是的，如果不这样，我们将无法确知要做什么。作为开发人员，你可能更希望客户能学习或者精通 C 语言。这样客户就知道开发人员正在做什么，以及他们有多么地勤劳。或者，这样的客户还能以 C 语言的方式告诉开发人员他们究竟想要什么。

然而，要求客户学习 C 语言明显是自杀式的行为。在客户（的代表）学会用 C 语言来向开发人员描述他们的需求之前，可能就已经被老板开掉了。因此没有客户会笨到愿意用 C 语言来描述他们的需求。

C 语言是程序员与计算机交流的语言，而不是他与客户交流的语言。程序员面对的是计算机，但计算机不是客户。

因此，在前面提到的 R 模型中，开发人员最好不要直接面对客户。项目经理有这样一种优势：他可以不用了解 C 语言，也可以用一种非 C 的语言来与客户交流（比如说汉语）。

或者你更愿意开发人员尽早进入状态，那么，可以让开发人员以需求调研的身份出现在客户面前。但是，请注意这个人员的角色将变成“需求调研者”，如果他不能适应这种转变，那就别让他去——那会是灾难的开始。

要深入项目需求阶段的项目经理或者调研人员，被要求深谙项目所涉及的业务。但这往往是做不到的，因为我们是软件公司，而不是做这些（客户的）业务的公司。这时惯常的做法是聘请行业咨询公司（或者个人）来介入需求阶段，以协助了解和分析需求。

这些咨询专家总是很喜欢把事情搞得很复杂，所以他们会说这一切的过程有个专用名词，“嗯……这叫需求建模。”他们很专业地说。

现在你应该发现了差距。比如我们的项目经理，以及那个被调来充当调研角色的程序员，他们就不会什么“需求建模”。

接下来咨询公司会与我们的客户一起做业务建模，然后再做业务到需求的映射，再抽取需求并完成需求建模。他们做业务建模的时候，可能使用一些客户业务范畴内的符号和标识；而在做需求建模时，则需要使用一些软件行业中（的设计和分析人员）习惯的符号和标识。

这些符号和标识也有个专用名称，“嗯……这个叫建模语言（ML）。”他们无时无刻不在向你展现他们的专业（这已经是他们还存在的唯一原因了）。

如果他们更加专业，他们会告诉你他们用的是 UML。向你介绍这个名词的时候，他们的眼镜或者眼睛里通常将会大放异彩。

UML 是模型世界里的世界语[14]。

到现在为止，你应该看到，咨询公司除了把问题搞得更加复杂之外，他们仍然需要面对最直接的问题：如何与客户交流？他们的解决之道是建模语言。

有什么差别吗？

程序员不能要求客户会 C，难道需求分析师们就能要求客户会 UML

[14] 现实的情况未必如此。但 UML 这个名词起码显示了它本源性的期望：Unified Modeling Language（统一建模语言）。

吗？！

1. 第 x 节 项目文档真的可以用甲骨文来写

台湾的项目管理顾问独孤木先生[15]曾经在《UML, OOAD and RUP》系列文章中讨论到 UML 在实际应用中的问题。其中的两个问题是：

- “大部分的使用者，以及客户的信息人员，其实并没有足够的能力来确认这些文件‘User Case’的正确性与完整性。”
- “除了客户不了解 UML、OOAD 跟 RUP 以外，另外一个更糟糕的现象就是 project team 里面的人也不懂。”

这实在是一件很有趣的事。

看来在一些情况下，在项目中使用 UML 只是完全不懂的老板，以及什么都懂的博士的主意，而真实的场景中去做事的那些客户与项目成员，其实是未见得就能用好 UML 的。

仅以 UML 的 User Case 来说，它由“用例图”和“用例规约”组成。规约跟我们写的需求说明书差不多，不过更加细节罢了，而且还有一套相应的方法论来阐述如何去实作。图则很简单，就是几个图形符号来描述系统边界和角色关系。

显然甲骨文也能描述范围与关系。例如甲骨文中的“家”这个字，就是上有房子下有猪[16]，这个边界就定义得很好；在古文中，“三”通常是泛指，跟 UML 图中的线条上标注的那个“*”是同义的；而甲骨文中“众”这个字，就是“日”字的下面立有三个人，也就是用“在同一个

[15] http://singlelog.pixnet.net/blog

[16] 至于“家”这个字表述的是“内有猪”还是“下有猪”的问题，不是我们要争论的。有些考古学家根据甲骨文的象形来认为古人与家猪是杂居的，但我想那时的猪可能还比较野性，因此这种可能性还是小些。

日头下做事的很多人”来表示“众”，这个关系也描述得很确切。

所以只要你运用得法，甲骨文一样可以用来画用例图和写用例规约。同样地，只要约定一套“语法”，你同样可以用甲骨文来做活动图、类图、构件图……以及与这些图相关的规约。相比来说，古巴比伦人使用的楔形文字“象形性”差一些，因此我不建议用它来画用例图。

既然甲骨文可以用来做为一种模型语言（同时它也是一种文字和口头的语言），那么，如果你的项目面对的对象是商周文化的考古学家，以及你的项目组都由精通这种语言的成员构成，这时你就可以用甲骨文来做项目文档，以及画各种模型图例。

你要明白，要让考古学家看懂用例图，难度远大于看懂甲骨文。与其要求他们学一种语言，不如使用他们那个世界的通用语（当然，前提是你的项目组也懂得这种语言）。

在韩愈的《答陈生书》中，他因自己不会“速化之术”，所以说陈生是“求道于盲”。然而他用了一个不恰当的比喻：因为盲人并非不知道路如何走，只是他不能像常人一样描述他所知道的路。因此“问道于盲”是没有错误的，真正的错误是你睁着眼睛问。

我们需要在正常人与盲人之间建立一种沟通的方式，既然盲人不能睁开眼睛，那么你就闭上眼睛好了。

UML 图在一些客户眼里无异于盲人的世界，如果需要向他们做需求调研，你只能使用一种这些客户能够理解和接受的方式，例如表格、流程图以及……更深入的交谈。

你要确认你的沟通方式是否有效，而不是去追求这种方式是不是 UML，以及你用 UML 表达得是否正确。客户是因为他认为你理解了

他们的需求，而在“需求确认书”上签字，而不是因为你的 UML 图画得是否精准。

现在来思考：为什么非要让客户看 UML 图呢？如果有能够满足“极限编程”所要求的“现场客户”[17]，那当然可以不画用例图；相反，如果客户雇了一个专家组来评审需求，那么你就老老实实地画用例图好了。

需要留意的是，专家组还要有一种方式与客户沟通，这有可能不是UML。当然，客户愿意增加沟通成本，那是他们的事。

一旦源头确定，接下来，你就可以约定在项目组中要使用的沟通方式。愚公——这个伟大的项目经理——所使用的“聚室而谋曰”，就是很好的沟通方式。当然，如果客户精通 UML，那么我想愚公采用的项目沟通方式将会是“聚室而论 UML”。我想一定会这样，因为愚公是很懂得沟通的、伟大的项目经理。

第 2 节 最简沟通

在 D 项目中，我向我的项目组员提出在需求阶段与客户的沟通计划。这个计划只有三条：

- 在一个月中，只能跟客户进行三次联系；
- 三次联系中，最多只能有一次面谈的机会；
- 一个月后，提交全部的需求调研报告、需求分析和关于该项目的远景规划。

[17] “现场客户（On-site Customer）”是极限编程（Extreme Programming，简称 XP）的特征之一。指的是要求客户可以在程序员开发的第一现场，能随时向程序员确认完成功能的有效性，以及修正需求或者先前的需求描述。这通常要求程序员有良好的沟通能力和较专业的客户知识。但事实可能正好相反：程序员并不具备这种能力。这样的事实，正是前面“R 模型”所基于的一个前提。

D 项目并不大，所以从主观上来讲，客户（代表）并不会为这个项目投入太多的精力。重要的是，我们在前期交涉中已经发现：这个客户代表为大量其他的项目和工作所困扰，他不会有时间来处理我们的问题。因此，减少沟通和保障沟通质量就显得尤其必要。

在大多数的项目中，都存在着这样的问题。真正能满足极限编程所提出的“现场客户”的情形并不经常出现。即使能将程序员送到客户现场中去，沟通问题仍然是不可避免的。

因此，在 D 项目中我提出了“最简沟通”。

我们开始在网络上查看相关软件系统的特征，以抽取客户所关注的内容；了解该客户的公司、经营理念、组织结构形式及工作模式；了解同类公司的成功经验和优秀的管理模式，以及客户的竞争对手在做什么和在关心什么.....

最后，我们开始综合以下两个方面的因素：

- 客户在公司层面的外在表现、内部机制和运营管理手段；
- 客户在项目中已经明确的需求和可能发生的需求，以及客户围绕其公司行为（和方向）所提出的需求。

这样就了解了客户项目中所有会产生需求的信息点。

我们开始设计提问，每一个提问涵盖尽可能多的信息点，尽可能地具有发散性，以便形成更多的推论和假设。

我们把这些做成项目概要，用 E-mail 提交给客户，并在第二天电话回访他。他以口头形式回复了这封 E-mail，这让我们尽可能多地得到了项目在方向上的修正。

我们确定了项目的实际目标，以及远期的方向。接下来就是设计需求条目。

客户已经先期提供了一些关于项目的文档、报表和工作数据。因此基于这些数据的需求分析，将是下一个沟通前所进行的最艰苦的工作。项目组员被要求：

- 分析用户的每一个表格，以构建基础数据库；
- 分析每一条数据的含义以确定它的上下限，以及数据间的相关性；
- 从工作文档中去了解客户的组织机构及其相互关系，同时确定每一类使用该系统的角色；
- 从报表中了解客户关注的数据信息，以及被他们所忽略掉的数据信息。

我们从数百条需求条目中，整理出系统结构和模块，需求条目被映射到各个模块。我们很快就画出了模块间的相互关系图，并通过这个图分析了数据的交叉关系，设计了相应的数据索引，并增加了一些新的关系性数据。

对用户角色、原始数据和系统结构进行了梳理之后，我们花了很短的时间实现了第一个系统模型。当然，很多的功能项目都只是简单地 show a dialog.....但我们优化了每一个操作流程，以保证不同的用户（角色）在使用时都尽可能流畅。

这一次的沟通我们使用了面对面的模式。我们很庆幸地得到了与这个系统的每一类用户（角色）接触的机会。而正好我们有一个模型，我们便让他们来操作并提出意见。这一次我们终于有了一份详尽的调研报告。

接下来的分析设计顺理成章。我们在一个月后完成了这个项目的需求分析报告，以及在这个分析上的一些框架型的设计。此外，还有

一个被用户接受的原始模型。

尽管第三次的沟通中还发现了一些问题，但我们终于有了一个好的开端。

应该清楚的是，保障每一次沟通的有效性都是最重要的事。沟通不是打电话或者请客户吃饭那么简单。你得到的每一次沟通机会，都是向客户了解更深层次需求的机会，因此最好在见到客户之前，你就已经设计了所有的问题和提问方式。

吃饭并不是有效的沟通。大多数时候，那将以醉酒收场。

第 3 节 为不存在的角色留下沟通的渠道

大多数人也许不知道，中国的“五千年文明史”实际上仅有三千年“有史可查”。因为我国古代编年史的上限，只能追溯到《史记•十二诸

侯年表》中记载的西周共和元年，亦即公元前 841 年。事实上，在司马迁在写史记时，他面临的许多文献材料就比较模糊，且相互间不一致，只好弃之而不用。因而，这导致“（夏商周）三代”的年代无法考证确实，司马迁也只能为其后可考证的年代写出“（十二诸侯国）年表”。

项目的中断和中止，与历史产生断层的内因是一致的。——我发现很多项目（尤其是产品计划）在负责人员离开后，就自然而然地死掉了。我把这一切的原因归咎于“没有 History”。

在先人写“谱牒”（简、册）的时候，想必是没有考虑过有后人要读的，或者更为远古的先人可能根本没想过要留下他们的生活和部落的记录，再加上有像秦始皇这样的人在前面放火烧东西，所以司马迁拿不到夏商周三代的确切史料，也是情理之中的事了。

远古的先人不知道司马迁这一号角色的存在，司马迁也没有办法跟古人约定一下要留点记录给他写《史记》。

我们做项目的时候，如果也不留下历史记录（History），那么以后别人来看这个项目，也会是两眼一抹黑，要么就像司马迁一样“存而不论”，项目便就此中止；要么就像“夏商周断代工程”一样，花大量的人力物力来攻关。

维护旧项目比做新项目更难，许多人深有同感。然而这些“有同感”的人又何曾想过，自己在做“新项目”的时候，要为“项目维护”这种还不存在的角色，留下一个沟通、对话的渠道呢？

我把项目的 History 作为跟这种“不存在的角色”沟通的一种方式。History 的丰富和准确为项目的后继开发、维护提供了可能。

历史记录（History）与注释（Comment）不是一回事。代码中的注释是为阅读代码而留备的，而 History 是为整个项目而记录的。一些参考的记录内容有：

- **需求阶段**：与谁联系、联系方式、过程、结果以及由此引发的需求或变更；
- **设计阶段**：如何进行设计、最初的构架、各个阶段的框架变化、因需求变更导致项目结构上的变化（有助于了解构架的可扩充性）；
- **开发阶段**：每一种技术选型的过程，每一种开发技巧的细节和相关文档，摘引的每一段代码、算法、开发包、组件库的出处和评测，程序单元的测试框架，每一个设计和构架变更所导致的影响；
- **测试阶段**：还记得测试用例和测试报告吗？那是最好的 History 之一。

当然，另一件最重要的事，是记得在每一笔记录后写下时间和你的名字。你的每一笔记录都是打算留给一些根本不了解这个项目的人看的，之所以要记下你的名字，是便于那些人能够再找到你并溯源到问题的源头。——当然，这得赶在你和古人一样“与天地共存”之前。

我们知道，大多数的工具都有历史记录的功能。在开发工具和测试工具中尤其如此。此外，版本管理工具也留下了每个阶段的印迹。然而，我建议不要过于信任它们。如果不明确要求组员写下详细的 History[18]，那么他们可能在每一次版本签入时都只写下两个字的备注：“完成”。

[18] 大多数的软件公司已经意识到版本管理的重要性。然而项目各个阶段的文档、代码及其他输入/输出都是具有版本问题的。单一的版本管理软件并不能胜任这些。因此，除了出于记叙开发历史的目的，我 也建议用编写 History 这样的方法，来弥补 ClearCase、SourceSafe、CVS 等这些软件的不足。

第 4 节 流于形式的沟通

在很多时候，我所听到的沟通，都是一种形式。例如与客户吃饭或者打回访电话。

其实沟通是具有目的性的，如果在没有明确目的的情况下与客户沟通，那将是浪费客户和自己的时间。这种目的，可以是了解项目的讯息、挖掘潜在的项目.....最末了，才是交流感情。

然而大多数情况下，它仅仅被看成是交流感情。这便成了形式，且往往是客户所讨厌的一种形式。

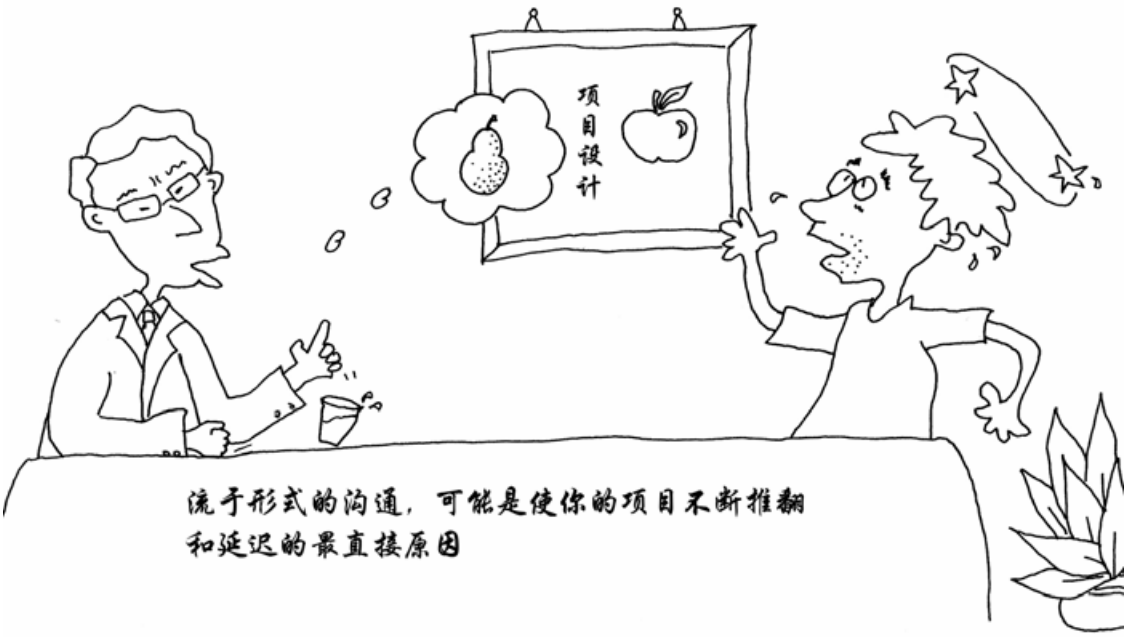
沟通问题不仅仅存在于你跟客户的交流之中，还存在于项目的各个角色之间。设计人员看不懂项目的分析报告，或者开发人员看不懂设计人员的方案，又或者测试人员看不懂开发的结果，等等，都是沟通问题。

UML 的确是解决沟通问题的最佳手段之一。然而如果项目一开始就不能用它，那么强求的结果必然是痛苦的——要让 UML 在一个没有经过相关培训的团队及其各个角色之间用起来，几乎是不可能的事。即使用得起来，也存在经验问题。千万不要指望仅仅一个项目，就能让你的组员深刻地理解 UML 的思想。

也不要指望在每个项目中都能用它，如果你的客户能理解并支持使用 UML，那么这个项目就会有一个良好的 UML 使用环境。否则，开发环节中资料的不一致性，将会使得项目难以收场。

使用与不使用 UML，其根本的问题在于沟通方式的选择。只要是行之有效的、能在各个项目角色间通用的，就是好的沟通方式。

在每一次回顾项目时都应该注意：流于形式的沟通，极有可能是你的项目被不断推翻和不断延迟的最直接原因。

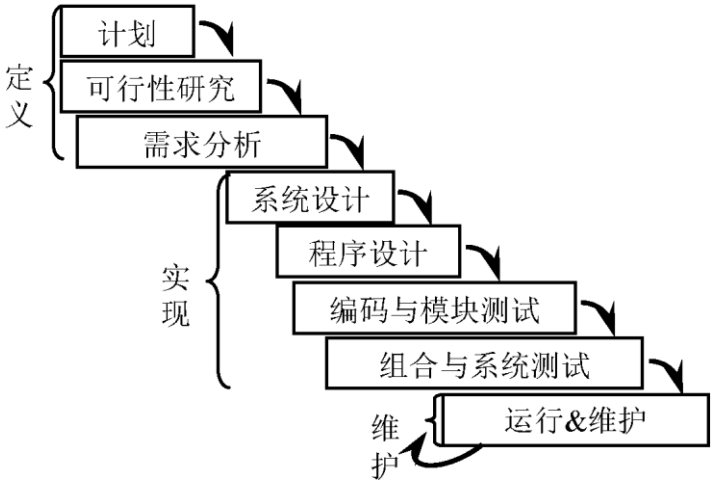


第 5 章 失败的过程也是过程

> “虚有其表耳。”
>
> ——《明皇实录》

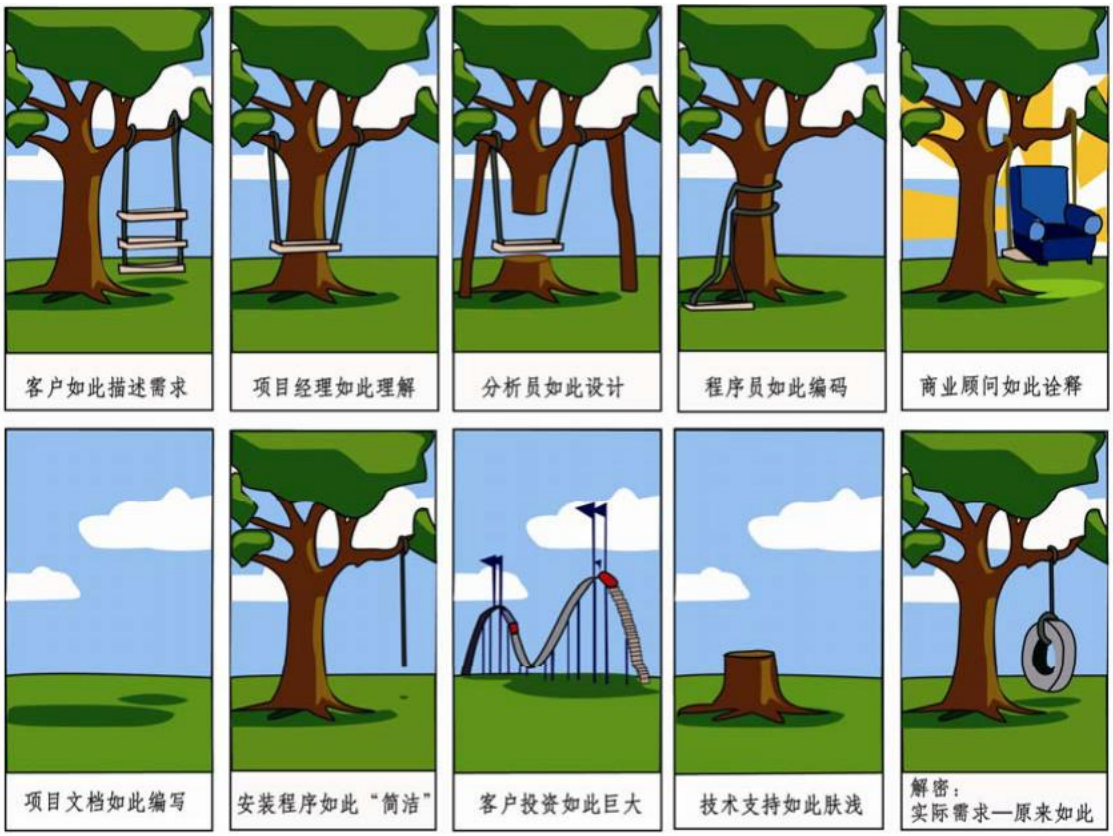
第 1 节 做过程不是做工程

软件工程这个概念被提出的时候大概是在 20 个世纪 60 年代末。它作为成熟的概念的标志是软件工程的瀑布模型的提出。瀑布模型将软件开发的过程分成需求、分析、设计、开发和测试五个主要阶段，其主要环节关系表现为如下的这样一种形态，如下图所示：



在瀑布模型之后，很多人开始研究过程模型的问题。很多从实际工程中提炼出来的过程模型都是值得称道的，例如 RAD（Rapid Application Development，快速应用开发）模型、螺旋模型和现在常被提及的 RUP（Rational Unified Process，Rational 统一开发过程）模型。

模型就是“样子”。人家拿出一个东西来说：这是模型。其言下之意就是要你按照这个样子来做。过程被描述为可重复的模型，实施的结果却可能演变成为相当尴尬的局面，如图：



我们看到，按照模型所描述的这个“样子”，做完过程的每一个阶段，并不等于做完了工程。或者说，工程并不是这样就可以做成功的。

换而言之，无论是用 RAD 模型还是 RUP 模型来做工程，即使是亦步亦趋，也做不好工程。

如果工程可以那样做成的话，只需要有瀑布模型就足够了。因此“做过程”并不是做工程的精义。

也不是目的。

第 2 节 做过场

为什么会存在这样的问题呢？

四川有句地方话叫“做过场”，也有说成“走过场”的。“过场”是舞台术语，意思是角色从舞台一端出场，再走到另一端进场的一个过程。过场角色一般没有唱腔或道白，即便是有，也是没有什么实质内容的。

前面那张图就是一个过场。尽管那是一张用来描述“沟通问题”的经典图例，然而你应该注意到，每一个角色都把自己的环节当成一个“过场”。如同演戏一样，从 A 做到 Z，就一切都完成了。当然，按照 RUP 的思想，是要从 A 到 Z 一遍又一遍地做下去的。然而，如果每一遍（或者用 RUP 的那个术语“迭代”）都只是“过场”的话，项目将只是一场无休止的演出而已。

最终呢？观众受不了，就交钱走人；演员受不了，就下台散伙。戏目和项目的结局，竟如此的相似。

第 3 节 实现，才是目的

很多人把问题的本质给忘掉了。从最开始，从我们编程开始，我们的目的就是实现一个东西。无论这个东西是小到称手的一个工具，还是大到千万的一个工程，我们的目标，都是要“实现”它。

工程只是一种实现的途径。最初做开发的前辈们，不用什么工程或者过程，也一样编出了程序，也一样解决了问题，也一样实现了目的。而现如今，我们讲工程了，讲过程了，讲方法了，却什么都再也做不出来了。

不奇怪么？

工程被当成了借口，掩盖了我们做事的真正目的：“实现”。因此，我们在一个项目中常常听到说“工程要这样做”，或者“工程要那样做”，而绝少听到“项目要求这样做”或者“客户的本意是那样的”。

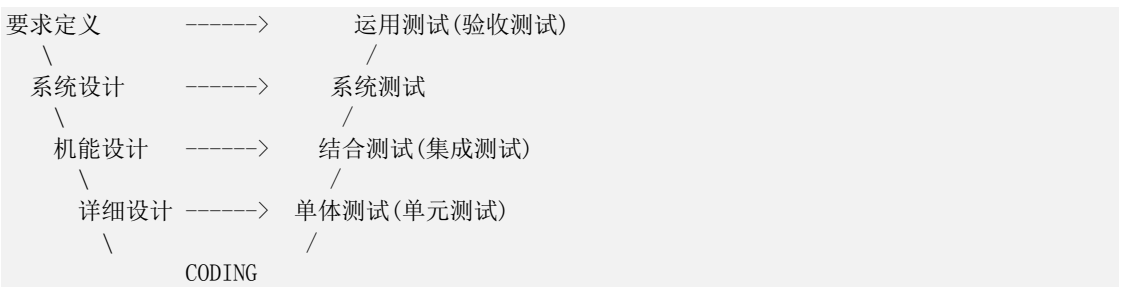
这样的结果是：我们做完了工程（的每一个过程），却没有完成项目（的每一个“实现目标”）。

为工程而工程的人，都迷失在项目中了。就像开发人员迷失在一个技术的细节上一样。专注于 RUP 或者 RAD 之间区别的人，可以把每一个过程的流程图都画出来，却也被这每一个流程给捆绑得死死的，再也没有挣扎一下的力气。

第 4 节 过程不是死模型

试着跳出大师们的身影，再仔细地看一下那些所谓的“经典”过程，不过是瀑布模型的一再变形。瀑布模型描述了开发的主要环节，于是一群人把这些环节拧来扭去或者反复叠加，就成了 RAD、螺旋、RUP，以及未知的、还没有被扭出来或者堆叠出来的 X、Y、Z。

2002 年前后，我在 CSDN 论坛中看到一个漂洋过海东渡扶桑的取经人，领受了被称为“日本 IT 工业发展史的活字典”的某个牛人的指点，贴出了一个足以令人振聋发聩的“V 字型模型”。



我看后就很不以为然。

为什么呢？你看，你把 V 模型拉直了，还不就是瀑布模型吗？

```
要求定义
  \
 系统设计
   \
  机能设计
    \
     详细设计
       \
       CODING
         \
          测试
```

然而如果仅仅是这样看问题，那还只不过是看到了皮表上的东西。

《韩非子•外储说左上》记载了“买椟还珠”的故事：

> “楚人有卖其珠于郑者，为木兰之柜，熏以桂椒，缀以珠玉，饰以玫瑰，辑以羽翠。郑人买其椟而还其珠。”

郑人就只看到了事物的表面，而忽略了实质的东西。如果我们只是把 V 模型当成折弯了的瀑布，那便是犯了相同的错误。

因此，我们应该去思考由瀑布模型到 V 模型这种变化的真实意图。

V 模型在每一个环节中都强调了测试（并提供了测试的依据），同时又在每一个环节都做到了对实现者和测试者的分离。由于测试者相对于实现者的关系是监督、考察和评审，因此测试者相当于在不断地做回顾和确认。

这有什么意义呢？你把它放在一个工程外包的背景里去考虑就明白了。由于日本近年来老龄化严重，因此劳动力短缺导致的劳工输入和项目外包，直接影响了它的组织管理模式。无论是在本国雇佣外来劳工，还是将项目直接外包给国外的开发团队，项目成果的阶段

性考察都是他们的第一要务，这直接决定了何时、如何，以及由谁来进入下一个环节。

因此，V 模型变得比其他模型更为实用。模型的左端是接受外包任务的团队或者公司，而右端则是日本软件企业中有丰富经验的工程人员。这样既节省人力，又可以保障工程质量。事实上，即使左端的外包任务是由多个团队同时承接，右边的工程人员也不需要更多的投入。

相对于瀑布模型，V 模型有改变了什么吗？是的。源于实际的需要，它把测试（和评审）阶段抽取出来，于是就成了 V 模型。

那么，如果需要，为什么不能把瀑布模型变成 W 模型，或者 M 模型呢？为什么就一定要跟随那个以迭代为基础的 RUP 模型呢？

更进一步想，如果瀑布可以变成 V、W 和 M，为什么它不能更关注于其中某个具体的环节（例如实现和设计环节）？如果在这些环节上引入 RUP 的思想，哈哈，你看看，是不是出现了勋章模型和烟斗模型？

然而你要知道，过程模型是在既有工程中总结出来的。也就是说，在某个模型有名字之前它就已经存在了，就已经被一些团队或者公司创生并使用了。

那么，为什么我们不是创生那些新的工程方法和软件过程理论的团队或者公司呢？

第 5 节“刻鹄类鹜”与“画虎类狗”

东汉时期，伏波将军马援在南征交趾期间写过一封著名的家书，是

教导两个“喜讥议，而通轻侠客”的侄儿的，希望他们学习敦厚谨慎的龙伯高，不要仿效豪侠仗义的杜季良。

龙伯高时任山都长，杜季良时任越骑司马，都是马援所“爱之重之”的人。然而马援告诫两个侄儿说：

> “效伯高不得，犹为谨敕之士，所谓刻鹄不成尚类鹜者也。效季良不得，陷为天下轻薄子，所谓画虎不成反类狗者也。”

于是，伯高、季良因马援家书而名留史册，“刻鹄类鹜”和“画虎类犬”就此成为典故。这意思便是说，雕刻天鹅不成，总还可以像只鸭子（鹜）；若画虎不成反而像条狗，那就事与愿违，贻人笑柄了。

后来的后来，近代作家朱湘就引用了这个典故，写了一篇《画虎》并收入《中书集》。《画虎》中写道：

> “一班胆小如鼠的老前辈便是这样警劝后生：学老杜罢，千万不要学李太白。因为老杜学不成，你至少还有个架子；学不成李的时候，你简直一无所有了。”

《画虎》这篇文章真的是好，真是建议大家读读。其中画师与画匠之论，精妙深彻，痛指骨质。而后论及日本，“国家中的画匠”几个字便打发了。

大师的眼光果然独到。

马援说刻成鸭子比画成狗好，其真实的意思是说学龙伯高不成，可得“谨敕”；学杜季良不成，则会流于“轻薄”。马援比较的是二者在骨子里所得所失的东西，而不是如朱湘所说的那种“架子上的像与不像”。

同样，以得失而论，在瀑布模型与 RUP 模型之间，学习前者而不成，

可思过程的本质；学习后者而不成，可得文字的架子。——用不好 RUP 的人，总会说自己终归还有一堆文档模板可以抄，便是这个缘故。

过程理论中，如果懂得了所谓的模型原本都演化自那个简单的瀑布，那么文档是按 XP 写还是按 RUP 写，也就可以应时、应需、因地制宜、择善而从了。本质的东西若能理解得透，架子还不是随手搬来就可以用的吗？

越是简单的东西，往往越是接近于本质。

RUP 中，真正精髓的东西，既不是那个 R（Rational），那是人家的招牌；也不是那个 U（Unified），那是人家的广告。而是那个 P（Process），那才是实实在在的东西。你要明白，如果瀑布模型理论是 Rational 提出的，他们一样会把它叫 RUP。

朱湘说：“画不成的老虎，真像狗；刻不成的鸿鹄，真像鹜吗？不然，不然。成功了便是虎同鹄，不成功时便都是怪物。”

马援说：“学龙伯高，即使达不到他的水平，总还能成为一个谨慎的人；而学杜季良，如果学不到家，便会沦为轻薄浪子。”

你到底是选择架子，还是骨子？

第 6 节 工程不是做的，是组织的

我们总是在说“做工程”，好像工程就是面包馒头一样，有个模子，拿来照着一堆面按上一按，放在笼屉上蒸上一蒸，就可以“做”出来了。

经历过工程的人都知道，我们没有那个模子，而工程中的人员也不

是那一堆面。所以我们当然不能“做”工程，而是要“组织”工程。项目经理的工作，就是要去

组织这个工程中的各个角色，使得分工明确，步调一致，共同地完成这个项目。

第 6 章 从编程到工程

> “得其精而忘其粗，在其内而忘其外；见其所见，不见其所不见，视其所视，而遗其所不视。”
>
> ——《列子 · 说符》

第 1 节 语言只是工具

我曾经是非常执著的开发人员。我有连续几天几夜 Coding 的经历，也曾经为了一个技术问题耗上三四个星期而导致项目一再延迟，还曾经为了一个实现细节与项目相关的人员逐一争论。

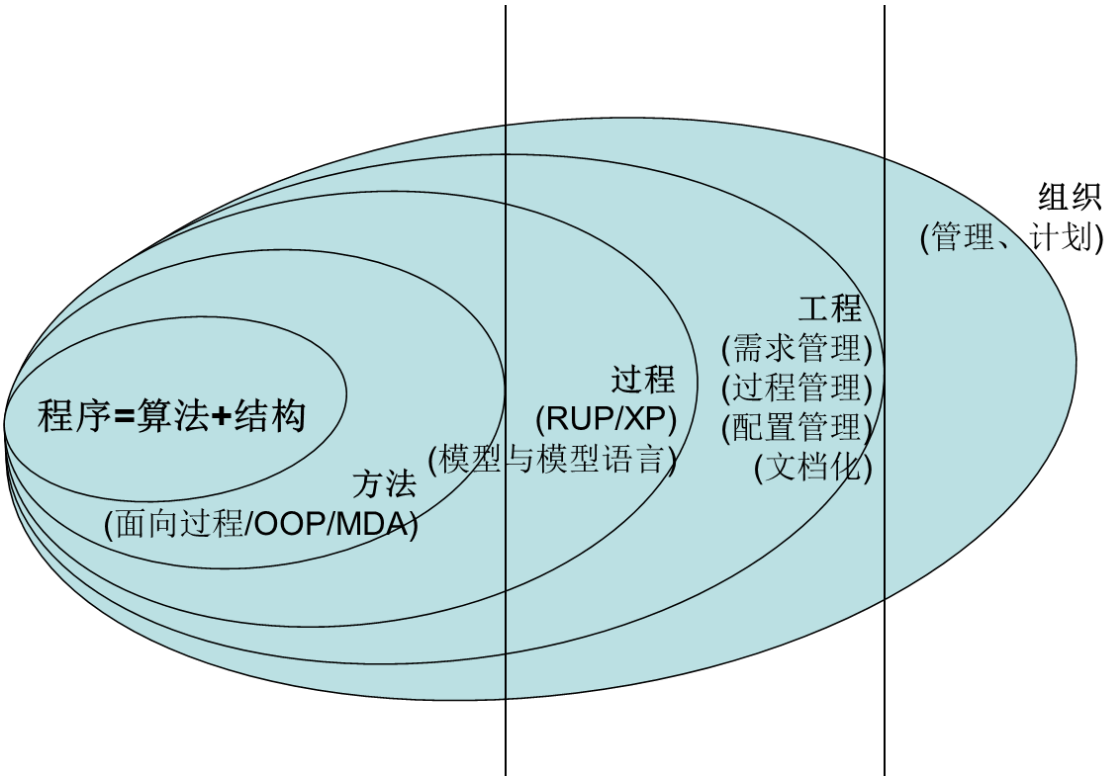
我也曾经像大多数的开发人员一样热衷于争论语言之间孰优孰劣。我在“Delphi 大富翁论坛”上写过一个简介，其中个人特长是“擅长 Turbo Pascal、Delphi、TASM 系列语言，痛恨 C/C++。（凡见有价值之 C 代码，先读通，后写成 Pascal/Delphi，最后骂一句：C 写得真笨！）”。我至今保留这段文字，因为那的确是真实的经历。

如今我已经不再专注于语言，正如我在第一章中写到的一样：成天讨论这门语言好，或者那门语言坏的人，甚至是可悲的。

然而就在我写这段文字之前的一年，我还在写《Delphi 源代码分析》，我还在无休止地深入语言的细节，深入操作系统的细节，以及深入.....开发的细节。

就在 2004 年 3 月间，我接受一个朋友的邀请，去北京做一个名为“Delphi & Delphi .NET 源码分析”的专题培训。我用了近两周的时间，做了 50 页的幻灯，全面讲述 Delphi 和 Delphi .NET。然而在临行前的一晚，我辗转反侧，一直在思考一个问题：我究竟做了些什么？或者说，我究竟想告诉学员些什么？

凌晨 5 点，我在幻灯的末页后面插入了一张幻灯，标题是“语言只是工具”，而幻灯的内容是这样一张图：



这是与培训完全无关的一张幻灯。然而，这是我自 1997 年接触到管理，以及 1998 年接触到工程以来，第一次正视“软件工程”这四个字。我第一次看清楚代码、方法、过程、工程与组织的关系！

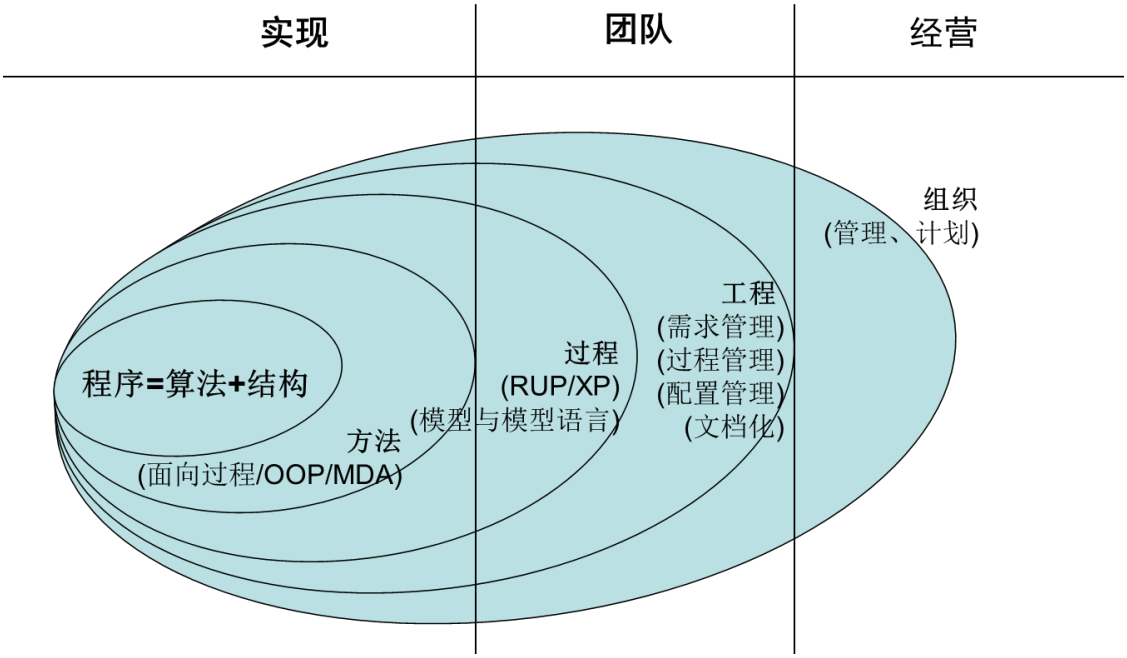
对于一个程序员，或者以程序员自命的人来说，看清楚这一切的第一步，竟是一句“语言只是工具”！

猿之于为人，“学会制作和使用工具”是最重要的标志。因而我不知道“语言只是工具”这句话，究竟是对语言的膜拜，还是漠视。

然而从那一刻开始，我才真正地知道工程。

第 2 节 关注点

在前面的模型图中，每一条纵向的细线用于定义一个关注点[19]。我在另一次培训中为这些关注点加上了标注：



这被我命名为软件工程层状模型（EHM，Engineering Hierarchy Model）。EHM 模型不描述工程元素间的关系，甚至在试图割裂这些元素，以使得工程角色定位及各自的视角更加清晰明确。

从这个模型中可以看到，在“程序”与“方法”层面，是关注于“（具体的）实现”的；而在“过程”和“工程”层面，更首要考虑的是团队问题。从角色的角度上来说：开发经理思考项目的实施方案和管理具体的开发行为，而项目经理则保障团队的稳定性和一致性。

[19] 我画出的的确是线而不是点，“关注点”只是一个概念。如果你非要去发现一个“点”，那么你可以用几何的目光，关注于弧线与直线的切点。然而，这样的结果将是你彻底地忽视了“关注点”的本质含义。

第 3 节 程序

EHM 模型图中，在最内层的环里，是“程序＝算法＋结构”。这是编程的本源定义，也是原始的状态。与代码相关的任何工作，最终仍旧会落足于这样的一条规则。

编程的精义在于此。从有开发行为开始，它就存在了。愚公在数千年前就在用类同的行为做编程实践，而几十万年前的智人，也在循环与分支所构成的逻辑中打转。

第 4 节 方法

推动这种逻辑向前发展的，是“方法”和“方法论”的出现。长期的编程实践，自然的归演与总结，必须沉淀为某种（软件开发）方法，于是“过程”出现了，于是“对象”出现了，于是相关的方法论也就出现了。

这是实践的成果。方法不是某个人或者某个组织创造的。瓜熟而蒂落，实践积累达到一定的程度，微软不提出某个方法，IBM 也会提出这个方法。即便他们都不提出，可能你自己已经在使用这个方法了。

方法并不神秘，因为它就是你今天正在做的、从事的和实现的。正如“模式”是一种方法，而模式就是你昨天书写代码的那个行为。只不过，GoF 归纳、抽取、提升了这些行为的内在规律。

你看不到你做事的行为，也就不能理解“模式”作为一种方法的价值。所以大师们众口一词：模式需要一定的编程经验才能理解。

同理，理解过程也需要编程经验，理解对象也需要编程经验，理解 MDA（模型驱动架构）与 SOA（面向服务的体系结构）还是需要编

程经验。

——这可能就发生在你去回顾你的上一行代码编写的经过，或者上一个项目失败的经历的那一瞬息。经验来源于回顾、理解与分析，而不是你将要写的下一行代码。

有人在寺院扫了一辈子的落叶而得道，也有人因为一句话而得道。

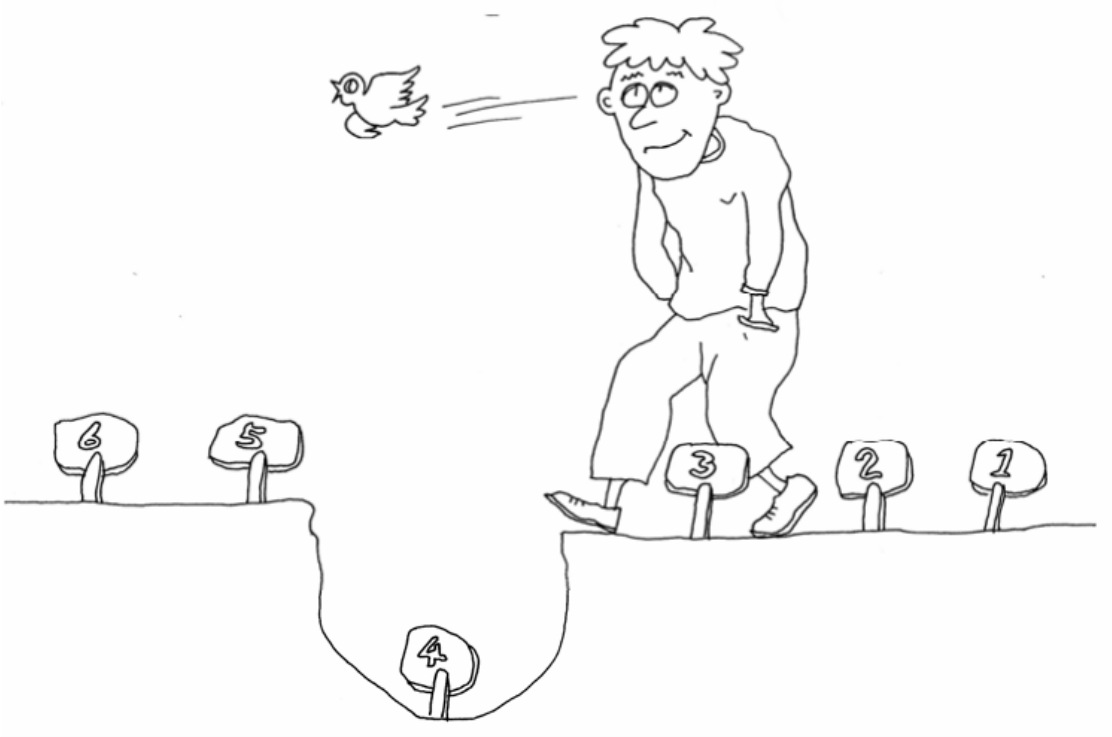
GoF 因为无数次的代码回顾而得道。

第 5 节 过程

过程伴生工程而出现。过程解决的是工程中角色间的关系问题。

过程说的是很多的人（团队）如何组织在一起进行开发的问题。它首先把工程中的环节分解出来。这样，有了环节，就有了角色；有了角色，就有了沟通。

因此过程中的问题，就是角色、沟通和环节的问题。



哪些环节重要，取决于具体的编程行为，也就是具体的项目。

例如产品开发的周期可以一再拖延，因为对产品来说，更重要的是品质和技术壁垒。因此你可以看到暴雪公司（Blizzard）的游戏总是一再跳票，而它从来都是将大幅的玩家测试和开发人员的个性特征放在第一位。与此相同，DOOM 与 QUAKE 系列的灵魂就是在游戏引擎的开发和设计上。

如果用这样的模式去做项目，可能软件公司没死掉，工程需求方也被拖死掉了——你有看见客户因为你对技术的远景描述而憧憬吗？不，你只会看到他们因为项目的一再延迟而懊恼、沮丧，或.....暴怒。

憧憬这种事情，只会发生在那些铁杆玩家的身上。分不清玩家与客户的区别，对项目经理来说，是可怕的。这将意味着他不能清楚地

知道哪个环节更加重要。

角色的确定，以及角色间的沟通问题，在项目过程中也同样重要。工程组织是否合理，相互的协作是否紧密，是这个项目能否成功的保障。

“合作无间”通常是流于书面报告中的措辞。真正的“无间”，应当是沟通的结果。然而 UML，则可能是你与客户，以及项目经理与开发人员被“离间”的第一因素。

第 6 节 工程

最狭义的工程，是描述“做什么”和“做到什么”。也就是说，是对目标的描述和成果的检测。至于这个工程目标的实现，是“过程”和“方法”的事；而有效、快速地实现“过程”和“方法”所需的，就是“工具”。

这种软件工程体系层次（Software Engineering Architectural Layers）被描述成一张图，我很有幸地在第一次画它的时候，将它画成了一堆牛屎。因此我从此都把它叫“牛屎图”，如图[20]：

过程伴随工程而出现，解决的是工程中“步调一致”的协作问题。那么工程是因为什么而出现的呢？

很显然，软件规模的不断增大是根本的原因。所以你会看到在几年前，开发一个小工具可以不讲工程；或者现在在你的 Word 中，为了将半角替换成全角字符而写的那个宏，也不需要工程。

[20] 在 2001 年进行一次公司的内部培训时，我在幻灯片中画下这张图。半年后，我去北京读研时又看到了它，不过是画在《软件工程——实践者的研究方法》这本经典巨著中，第四层的“实现对象”是叫做“质量焦点”，名字则是“软件工程层次图”。其实所谓“经典”也是对既有知识的总结。大师们所知的，与你所思考的未必就有天壤之别。

接下来，即使软件规模增大，如果有一个牛人中的超牛人，愿意用 20 年来写一个任意庞大和复杂的操作系统，他也是能做到的。然而现实中不会有软件公司给他这样的机会。

项目的“复杂”可能要求不同知识领域的角色参与，而“庞大”则要求更多的（人力、技术与管理）资源。“团队”作为开发行为的模式，是软件规模和复杂度渐次累积的结果。

团队必将越来越庞大，因为（与工程对应的）软件必将越来越复杂。没有团队意识的软件公司在高度过程化、通晓方法理论[21]、拥有大量工具的集团军面前，必将一触即溃[22]。

第 7 节 组织

工程理论其实是包含组织学的。然而我在上面的那张图中，将组织与工程分离开来，并在二者之间画下了一道纵向的线，如图：

[21] 《三十六计》更多的时候是被当成方法论来读的。其根源在于“计谋”本身只是方法，而不是战略。

[22] 如今这样的战役正在国外软件与国内软件之间进行着。而战局，并不是民族热情或者政府保护可以扭转的。

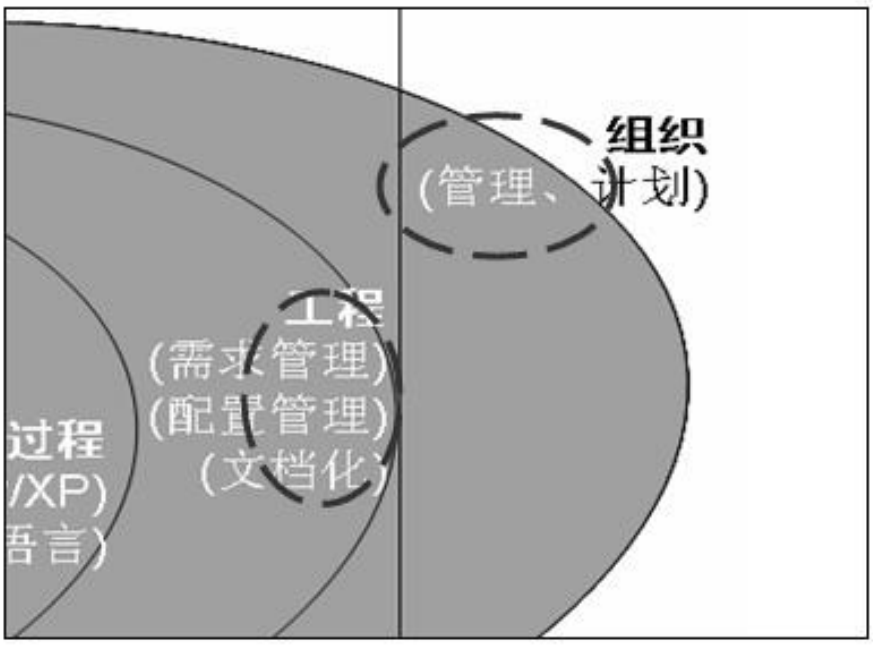


图 7-4 软件工程层状模型（局部）

即使如图上所表现的那样，工程关心的是“需求”、“配置”和“文档”等这样一些要素，那么这样的工程还是停留在技术层面的：关注的还是工程的实现细节，而非目标。从角色的角度来看，这是项目经理和技术经理所共同关注的那一部分。

作为那个解结的人，你应该知道这个图例上的需求管理决定了你的阶段性目标，配置管理决定了你检测的方法，相关的文档化工作是沟通的渠道，以及协调团队矛盾的手段。“需求”、“配置”和“文档”等不仅仅是你每日一成不变的、模式化的工作步骤，而且也是解结的工具与方法，或者是你实施管理思想的道具。

此外，你（项目经理）还必须关注于人力资源、项目资金及多个项目之间的协调等。这些与工程本身并没有直接关系，而是“组织”方

面的内容。

所以在工程环节中，“文档管理”和“配置管理”等中的那个词汇“管理”，是管理的具体技术和方法；而在“组织”这个环节中的这个“管理”，才是真正的管理学上的用词。

我在这张图上，试图从这个角度来说明：作为项目经理，你必须有一部分的工作是非技术性的。甚至，你可能绝大部分的工作是非技术性的——因为与技术相关的管理技能（需求、配置、过程管理等）可以由开发经理来做，或者公司对于这一方面有较统一且成熟的规范，因而无须投入过多的精力。

你必须更关注于对这个（或这些）工程的组织与计划。站在“组织者”这个角色上，你现在要考虑的内容可能会是：

- 为项目的各个阶段建立计划，并逐渐地细化计划的内容，以及确立项目过程中每一个环节、每一个计划阶段的优先级和复杂度；
- 确立项目或者产品阶段目标，成果的准确描述、定位，以及整个项目的质量目标及其评核办法；
- 对团队中的不同角色展开培训，以指导并协调角色间的工作，从而消除因为工作习惯的差异带来的影响；
- 为每一个人准备他所需要的资源，这不单单是把一套 Shareware 变成正式版或者把 512MB 内存变成 2GB，还包括准确地评估他的工作量，以及决定是否为他增加一个（能协同工作的）副手；
- 决定在哪些环节上反复审核和回顾，而在哪些环节上采用较为宽松的方式以加快进度；
- 习惯于开会、组织更短而有效的会议，以及建立激励机制，当然也不要忘记让每一个成员意识到这一项目的风险；
- 不要乐观。

即使你做好这一切，可能项目的结果仍然不够理想。但是你应该知道，好的项目经理并不是不犯错误的人，而是以尽可能少的失败来获得成功的那个人。

无论是你的团队成员，还是你的老板，对重复的错误以及可预料的错误都是不会宽容的。——在一个团队中，失去了组员的信任比失去老板的信任更可怕。

所以回顾每一个项目，或者项目中的每一个阶段，以及与每一个团队成员交流的细节，是你的日常工作。

第 8 节 BOSS

很多人以为 BOSS 是给自己发钱的那个人，这其实是错误的。发钱的决策通常是由以下三个角色来做出的：

- 部门 / 团队经理：你的直接上司，他是雇佣你的人，是他用薪金的多少来衡量你的价值，或者反之。
- 绩效经理：如果你的公司有这个角色的话，那么他总是盯着你的错误以决定你薪水的扣除比例。
- 财务经理：有钱？没钱？没钱？有钱？.....

BOSS 并不决定你的薪水[23]。

BOSS 在公司中解决的是“经营”问题。这其实是在比“组织”更靠外侧的一层——在前面的图例中并没有给出，这也意味着“经营者”与“工程”基本没有关系。

[23] 顺便告诉你一个秘密，给予你奖励的决定通常是你的上司，而不是绩效经理作出的。

在一个更大规模的组织机构里，你可以更直接地观察到“经营者”与“组织者”之间的差异。例如公司的大小股东是“经营者”，董事会通常是解决经营问题的地方；而总经理、执行经理及各个部门经理则是各级的“组织者”，经理办公会则是解决组织问题的地方。

你应该清楚，真正的 BOSS 是经营者[24]。这有助于你明确你被雇来的原因，你的工作是面向哪一个层面的，以及你或者你的上司有没有权限来决定一个项目是否应该立项，或者中止。

BOSS（经营者）决定了一个方向，组织者保证决策与这个方向是同步的，而工程是在这样的一个方向、决策的构架下的一个具体行为。

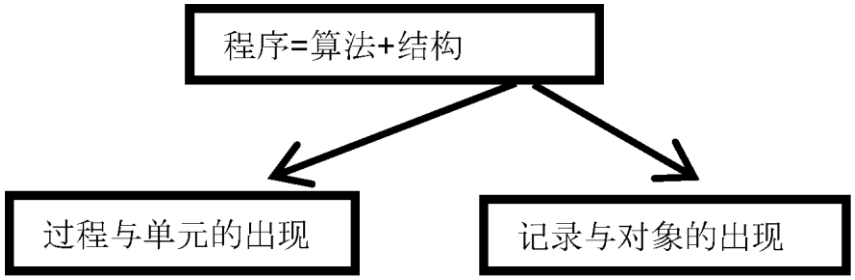
工程中没有 BOSS。

第 9 节 上帝之手

从最初的简单编程开始，到现在工程团队的组织开发，实现（一个软件）都是最终的目的。所以可以这样说：实现，是软件开发的本质需求。

我们看到，正是出于实现的需要，我们才设计了一些数据结构或逻辑结构来映射物理模型。因此类似于过程、单元、记录（结构）、对象等的出现，其实都是出于编程实现的需要，如图所示：

[24] 不过，可能你仅受雇于你的上司，你习惯于把他叫作 BOOOOSS 则是另外一回事。



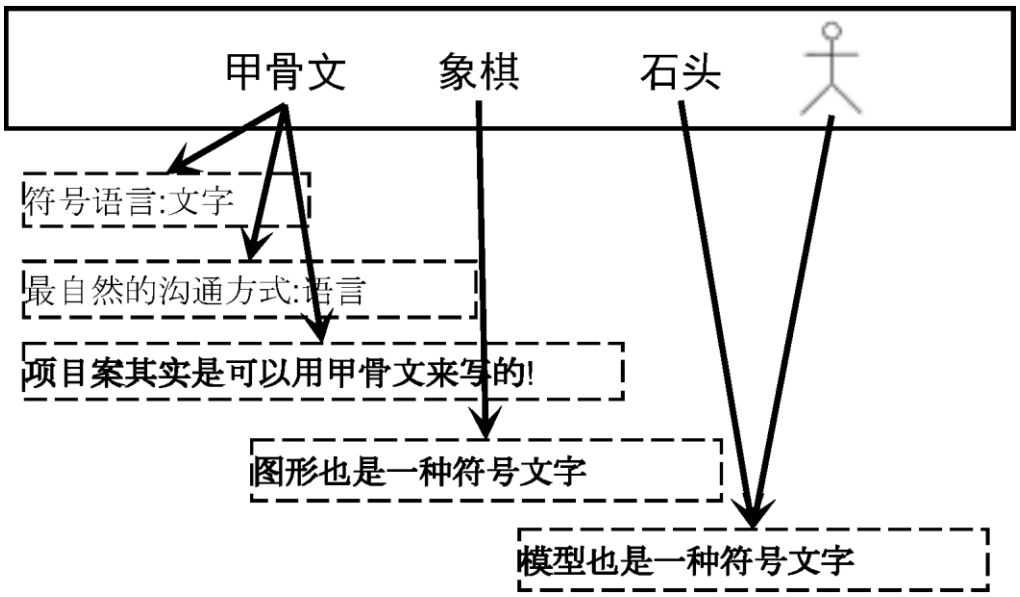
而后，基于某种数据结构的编程实践（的不断积累），决定了软件开发方法理论的产生。

从这一点可以看出：方法，是对既有行为的归纳总结。

因而实现方法总是最先出现的，而后才有分析和设计方法。可以看到：面向对象分析（OOA）、设计（OOD）与编程（OOP）的出现顺序，与它们在工程过程中的实作顺序正好相反，而与编程实践行为的顺序则正好相同。

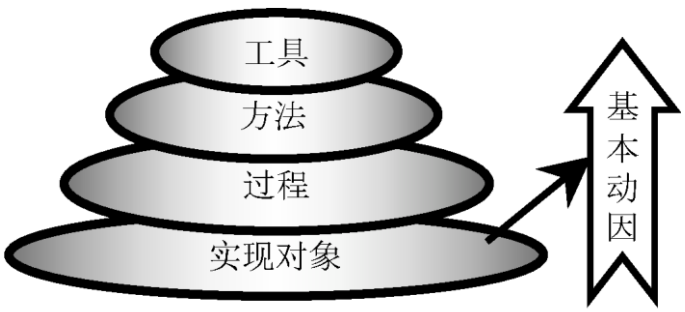
为了实现更大规模的软件系统而有了团队组织模式，而团队的协作决定了过程模型的产生，在过程环节中的沟通问题导致了（模型化）语言的出现。

如同编程工具中的编译器和集成开发环境（IDE）一样，开发中的编程语言、过程中的建模语言都只是一种工具。如下图所示，他们之间并无本质不同：



工具的产生仍旧是出于“（软件）实现”的需要。不可能从软件开发实践中产生出轮子和指南针，因为那不是“软件开发的本质需求”可以推动的。

软件工程的体系中，“实现”作为软件开发的本质需求和基本动因，如同上帝之手在推动这几十年来的软件工程理论体系的形成。



软件工程

第 7 章 现实中的软件工程

> “王不如远交而近攻，得寸，则王之寸；得尺，亦王之尺也。”
> ——《战国策．秦策》

第 1 节 大公司手中的算盘

从最早仅仅关注于软件开发工具到现在，软件行业中的巨头们已经在层出不穷的思想中涅槃了一回又一回。

Rational 被 IBM 购并的真实原因在于 IBM 需要构建一个完整的软件工程体系。有了 Rational 的 IBM 会变成如下表这个样子：

表 1 IBM 的软件工程（2004）

IBM的软件工程（2004）		
	理论体系	实　现
工具	Language	Rational Suite、WebSphere、Eclipse
方法	OOA/D/P	IBM软件开发平台(SDP)
过程	RUP	RUP2000、RUP2003

IBM 得到 Rational 的最大好处，是在软件工程方面快速地拥有了一套成熟的理论体系和实作工具。对于 IBM 来说，Rational 在 UML 方面有着非常丰富的实践经验，还有着 RUP 作为理论框架的创立者和领导者的地位，这些对 IBM 在确立大型软件工程应用方案提供商的行业形象上，都是极大的支持。

在语言方面，IBM 注意到 JAVA 平台中立的语言特性，以及它在大型应用工程方面的成功表现。作为扼制 Microsoft 的平台优势的唯一途径，IBM 在语言方面选择支持 JAVA 是明智的。

出于同样的理由，IBM 亲近开源软件界，并很快得到开源软件领域

的头羊地位。这使得 IBM 从没有语言优势立即变成了“可以忽略语言劣势”。开源界给了 IBM 一种对抗的背景和实力，而 IBM 只需要做到把握这种力量，就可以在潮流中稳如磐石。

把握力量总之比创造力量来得经济。

同样，Borland 也从开发工具厂商的位置跳出来，希望构建类似的软件工程体系。所以现在你会看到这样的一个 Borland：

表 2 Borland 的软件工程（2004）

Borland的软件工程（2004）		
	理论体系	实　现
工具	Language	Togther、StarTeam、Delphi、CBX、JB...
方法	OOA/D/P	Galileo、PrimeTime
过程	ALM	Borland ALM Solution

对于 Borland 来说，在对软件开发语言（C、Java、Delphi）的把握方面是优势，所以 Borland 一直保持在语言上的中立，以寻求一种在不同平台上的开发者社群的支持最大化。Borland 积极地推动 UML 的标准化，一方面可以使得 Borland 有机会在模型语言标准的制定上有机会制造影响，另一方面也可以快速地与 IBM（以及 Rational）构成对抗 Microsoft 的战线。

作为工具开发商，Borland 快速地拥有了实现 ALM（Application Lifecycle Management）所需的绝大多数软件产品。然而 Borland 也很快意识到，（当前的）ALM 是一个产品体系，而不是一个理论体系：Borland 没有在 ALM 作为工程理论方面的任何优势。于是 Borland 开始购并与实现 ALM 体系相关的公司，其中，收购过程改

进咨询公司 TeraQuest，并组建流程优化实务部，以及收购 TogetherSoft 为开发工具来强化模型构建能力，都是相当大的一些举措。通过这些努力，Borland 快速地补全了 ALM 作为一个工程体系在理论方面的不足。

对于 IBM 来说，RUP 和 UML 是优势，所以 IBM 用来削弱 Borland 在开发语言上优势的最佳手段，就是支持开源的 Eclipse，以及用 UML 的标准化来确立其规范制定者的地位。然而你会惊异地发现，Borland 一方面在支持 UML 的标准化，另一方面还在支持着 Eclipse 的开发并协助其快速成为一个完整的、具有商业品质的开发平台。

这似乎是极其怪异的战略：帮对手磨剑。

如果 Borland 只为一个对手磨剑，那它可能是一个傻子。但问题是，Borland 几乎为它所有既已成为或者终将成为对手的人磨剑：Kylix 是 Linux 平台上的产品，C++Builder、C# Builder、CBX、Delphi 是 Win32 和.NET 平台上的产品，JBuilder 则是 SUN 平台上的产品——一切正如 Borland 自己说的那样，它是“（语言、平台和技术）中立的软件厂商”。

Borland 走在钢丝绳的中间，对它的考验是平衡的艺术和技术[25]：如

[25] 在这个小节完成的一年之后，Borland 公司在软件开发界中发生了里程碑式的大事：2006 年，Borland 试图一次出售全部的 IDE 产品线。这件事的起因与结局，正好如实地反映了这里的两个观点。观点之一，在 Borland 试图从工具厂商走向软件工程方案供应商的过程中，作为“开发工具”的语言及其 IDE，只是其庞大战略中的一个可以或缺的环节。因为在工程领域中，方法、过程以及领域知识，是相对更为重要的成功关键。因此，同样是工具，Borland 不会出手 Together，却可以放弃 IDE——这绝不是用“是否有赢利”这种简单的法子去衡量他们的价值。观点之二，是 Borland 在情势发生变化时，SUN、IBM 与 Microsoft 等都不能及时地援手。而这，正是因为 Borland 过于复杂的产品线，和“中立态度”的平台策略，使得没有一家公司（在期望的价格上）拿下整单交易。于是，Borland 不得不将 IDE 产品线变成了 CodeGear 子公司。而最终，Borland 也被以 7500 万美金卖出，开发工具史上的一段传奇最终黯然落幕了。

果它倒下，钢丝绳两端之任一，对它都不及援手；然而如果它存在，那么它无论向哪边迈出一步，都将给另一方以最大的压力。

敌人的敌人就是自己的朋友，聪明的战略家总是能看到这一点。然而 Borland 却力图使这个敌我都分不清的战场呈现出一种古怪的格局：一方面 Microsoft 是 Borland 的股东之一，另一方面 Borland 在做 SUN、IBM 以及 Linux 平台上的软件提供商。

与 Borland 和 IBM 通过购并来达到目的的方式并不相同，Microsoft 有足够的力量全方位出击，因此你看到的体系会是下面的这个样子：

表 3 Microsoft 的软件工程（2004）

Microsoft的软件工程（2004）		
	理论体系	实　现
工具	Language	VS.NET、DSL、.NET Framework
方法	OOA/D/P	需求方法、模型方法、测试方法...
过程	MSF	MSF Process Model v.3.1

Microsoft 在工具、方法和过程方面都有具体的实现。而 IBM 在方法和过程层面上大都停留在理论阶段，Borland 在这些方面虽有丰富的产品实现，却又相对缺乏理论基础。

Microsoft 并不仅停留于此。从.NET Framework 提出开始，Microsoft 就试图在开发语言和基础框架上实现大统一，希望能达到类似于 UML 在建模语言中的地位。因此出现了通用的语言体系：CLR+CTS，以及其具体的实现：.NET CLR + IAsm。.NET 上的代码要求最终被实现成中间代码，可以反汇编到 IAsm，这意味着任何其他公司在开发语言层面上的优势丧失殆尽，所以开发者们看到 C#、

JScript.NET 和 VB.NET 的同期实现的“壮举”。

而 Mono 的出现，对于 Microsoft 来说是绝对的福音。Microsoft 把.NET Framework 中的 C#、公共语言架构（CLI）及通用类型系统（CTS）等做成 ECMA 标准，最期望看到的就是类似 Mono 这样的第三方产品的出现。事实上，Mono 做了 Microsoft 从来都想做而不敢做的事——解决了 Microsoft 产品的跨平台问题，进而削弱了 SUN 这样的语言的跨平台优势。Microsoft 一方面不想放弃自己的平台优势，另一方面又为 SUN 的跨平台优势所制肘。而 Mono 的出现，以及它适度的影响力，正好成为 Microsoft 平衡这种微妙的、相对优劣形势的棋子。

接下来 Microsoft 开始向建模语言发难。领域专用语言（Domain-Specific Language，DSL）的提出绝非偶然，那是在硝烟未尽的战场上重新燃起的战火。

软件业界如今的局面，不是一些人（例如程序员或者评论家们）争争吵吵的结果，而是大公司们相互制衡的结果。Borland 与 IBM、IBM 与 SUN，以及 SUN 与 Apple 都在做着相同的事，又都有各自的算盘。他们一面打压对手的优势，一面又借助对手和同盟的力量来弱化自己的劣势或者补充实力。

跳出到局外来看，并不是说 Microsoft 是他们的共同对手，而只是因为 Microsoft 站在了风头浪尖，便成了众矢之的。所有人面对的并不是 Microsoft 的这个名字，而只是这个地位，无论谁成就了这个地位，都将承受相同的风险与压力。

当然也包括机会。

大公司们在标准、理论、语言上的争来夺去，未必全然出于“软件实

现”的考虑。对统一理论、统一工具、统一过程的企图，其最终目的是在整个软件工程体系中的全面胜出。

算盘上的绝大多数人，只是用于计算胜负的一枚算子。

第 2 节 思考项目成本的经理

因而，除了软件本质力量的推动之外，商业因素也推动着软件工程体系的发展。大公司们的争夺战的最终结果，已经开始把软件工程，从原始的“自身演进”状态，逐渐推进到“他激发展”的状态上了。

这种他激发展可能会影响到软件工程发展的速度，而不会影响到各个工程层面上的“关注点”。因而对于工程体系的描述来说，EHM 应当是一种基本模式。

但是，这也是一种理想模式。正是在为 EHM 图标注关注点时，如下的问题引起了我的思考：

- 项目的管理到底是组织管理还是成本管理？
- 项目的计划到底是组织规划还是成本计划？

简单地说：项目管理要不要考虑成本问题？

现在，让我们从一个细节跳出来，来看看我们的角色。这个细节就是：如何完成今天的工作。正如前面所说，如果你是一个软件公司里的项目经理，你今天的工作可能是写一份项目计划案，或者听测试部的报告，又或者是安排会议来听取和分析一个新的产品需求。

然而，我要说的是：这是细节。

细节就是你使用的 Project 2003，或者你正在公司内部署和推广的

ClearCase。如果它们正好是你今天要完成的工作，或者是你明天要用来工作的工具，那么，作为项目经理的你，现在就要立即跳出来。

理想状况下，“软件工程=过程+方法+工具”。然而工程成功的真正关键，并不在于你把你的团队“组织”得有多好。即使在团队中他们都表现得有条不紊，你一样会面临失败。

蚂蚁的团队总是被本能地组织得非常好。然而如果一个蚂蚁的群体中有了流行疾病，蚂蚁在死去，而新生蚂蚁不能跟上其死亡的速度，那么很快，这个团队就溃散了。

这是因为蚂蚁用于维护团队运作的“资本”在流失。如果资本没有了，就没了运作，团队的存在就没有了必要性和可能性。

项目就死亡了。

埋头于画甘特图的项目经理犯下了与挖山不止的愚公类同的错误：忽略了成本。

如果愚公真的可以成功，那么可能是 300 年之后。然而如果一个工程要 300 年才能做成，那么在做成之前，客户就选择了放弃。

如果有机会，项目经理可以选择向另一家公司购买一个产品来卖给客户，从“为客户开发”变成“为客户定制”，以及“为客户服务”。这样在没有任何开发成本的前提下完成了工程。与另一个同样极端的例子相比，你会发现它与第五章中那个“做过场”的项目全然不同。后者是做完了工程（的全部过程），却没有做成工程。而现在这个项目经理却做成了工程，但是在许多的过程环节上，他根本就没有开始。

然而现在，除了跃跃欲试的技术经理之外，没有人会不满意这个结

果。技术经理最常说的话是：我们可以开发出来；开发人员最常说的话是：我可以开发出来；愚公最常说的话是：何苦而不平？

还记得那句话吗？——不要栽进蚂蚁洞里！

愚公如果停下来，思考的问题可能是碎石的“方法”。而项目经理从细节中跳出来，思考的问题就应当是完成工程的“方法”。评价这个方法的好坏的标准只有一个：节约成本。

Y 公司由 K 公司过渡而来的时候带来了一个市场前景非常看好的产品。而存在的问题则是两方面的，一是扩大市场占有，二是继续的技术投入。

于是，Y 公司请来了专家 D。他做过公司，也在无数家公司做过，是一个在行业中摸爬滚打了多年的顾问型专家。D 先生的项目计划可能是无可挑剔的，但其投资规模决定了它无法实施；D 先生在一些产品计划上的思考上也是切近市场的，然而他没有办法为团队争取到两名以上的开发人员；D 先生在部门管理上的方法也是适当的，然而他却无法训练部门人员以使他们与自己保持一致的步调和方向（组织和管理一个松散的团队比照顾一群蚂蚁难得多）。

于是在 Y 公司建立到倒掉的四年时间里，D 先生三进三出，营销计划一再被否决，而产品的再研发计划也数度搁置。很快，这个并不生动的故事被终结于我跟他的最后一次会谈：三年之后，产品彻底从市场中退出。

——思考成本，这是 D 先生给我的教训：

- 不计成本的项目计划不会得到经营者的支持；
- 毫无目的地消耗成本是项目中的慢性毒药；

- 最致命的风险是成本的枯竭[26]。

第 3 节 审视 AOP

我读到的第一篇关于 AOP（面向方面编程）的文章，居然说它是“新一代的 Java 语言”。OH，正如文章的标题所表现的那样，作者大概是在学习如何向方格子里填写“错误”：其结果当然是每一个格子都是“错误”——如果他像小学生一样勤奋的话。

AOP 不是语言。AOP 首先是方法论，这就像 OOP 是“面向对象的编程方法”这种方法论一样。而 Delphi / C++才是语言，是 OOP 这个方法论的一个实现工具。

很好，有了这个基础，我们再来讨论相似性的问题。我们提到过开发方法是基于一种数据结构的编程实践的结果。很显然，OOP 所基于的数据结构是对象（Object），而 AOP 所基于的数据结构就是切面（Aspect）[27]。落足到开发工具的实现上，Delphi 将 Object 表现为一组有（继承）关系的“记录”[28]。相对应地，Java 则用类（Class）来实现 Aspect。

Aspect 在定义时没有确定的对象模块，Aspect 本身只是对一个“对

---

[26] 我经常注意到的成本因素包括时间、人力、资金和客户成本。而大多数情况下，人们不会把客户的数量及耐心当做（客户）成本来计算。而在我的项目规划中，这是成本。

[27] 人们在争论 Aspect 到底应该译成“切面”还是“方面”这件事上花了很多工夫。其实，就如同讨论前面的“关注点”究竟是“点”还是“线”的问题一样，他们陷入了细节。如果这些细节被作为问题持续下去，那么可能有一天台海战争将不是发生在军队之间，而是在程序员之间：到底是“物件”，还是“对象”？

[28] 在 C 中，这个名词是“结构（Struct）”。很多人不会承认“对象是有继承关系的记录”这样的观点。是的，所有的教科书上都不会这样写。但是从数据结构本身及数据结构在语言中的实现来看，对象终究是记录。记录是平板化的内存存储体系中所能表达的最复杂的单一数据体。

象模块群体”的观察视角，因此它更易于表现成接口——只有描述而没有实现。

在 Object 一层的抽象上，Object 关注于“有继承关系的考察对象”的个体特征；而在 Aspect 一层的抽象上，Aspect 关注于“有相似需求的考察对象”的群体特性。其相似性在群体中表现得越广泛，则 AOP 的优势也就越明显。例如在 Delphi 的 VCL 框架中，以下两个需求就可以用 AOP 的思想来实现：

- 使 Delphi 中全部对象具有多线程特性（即线程安全）；
- 实现助手工具类以观察、控制 Delphi 对象的运行期特性或表现。

到现在为止，我们弄清楚了 AOP 作为“思想、方法、工具”的一些基本知识，以及它的应用范围：至少你要明白，它是用来考察对象（而不是设计对象）的思想方法。

所以接下来 AOP 的三个概念我们就明白了：

- 指示（Advice）/拦截器（Interceptor）：考察这些对象以“达到什么样的目的”（即需求）；
- 引导（Introduction）：在目标上实现这些需求时，目标所需要表现出来的公共特性，引导特性可能需要配合编译器来实现；
- 元数据（Metadata）：如果需要，为既有对象实体再补充一些参考数据。

确切地说，切分点（Pointcut）并不是 AOP 编程方法所需要的概念，而是 AOP 作为一个框架去实现时所需要的一个工具：一组辨识 Aspects 和 Objects 的索引。

现在你已经会用 Aspect 的思想来编程了，而无论它是用 Java 来实现的，还是用 C#、Delphi，乃至于 FORTRAN 或 COBOL 来实现的。其实如同这里所表现的那样，学习任何一种新的编程方法，你需要

做的仅仅只是回到工程最核心的那个环节：程序=算法+结构+方法。

第 4 节 审视 MDA/MDD

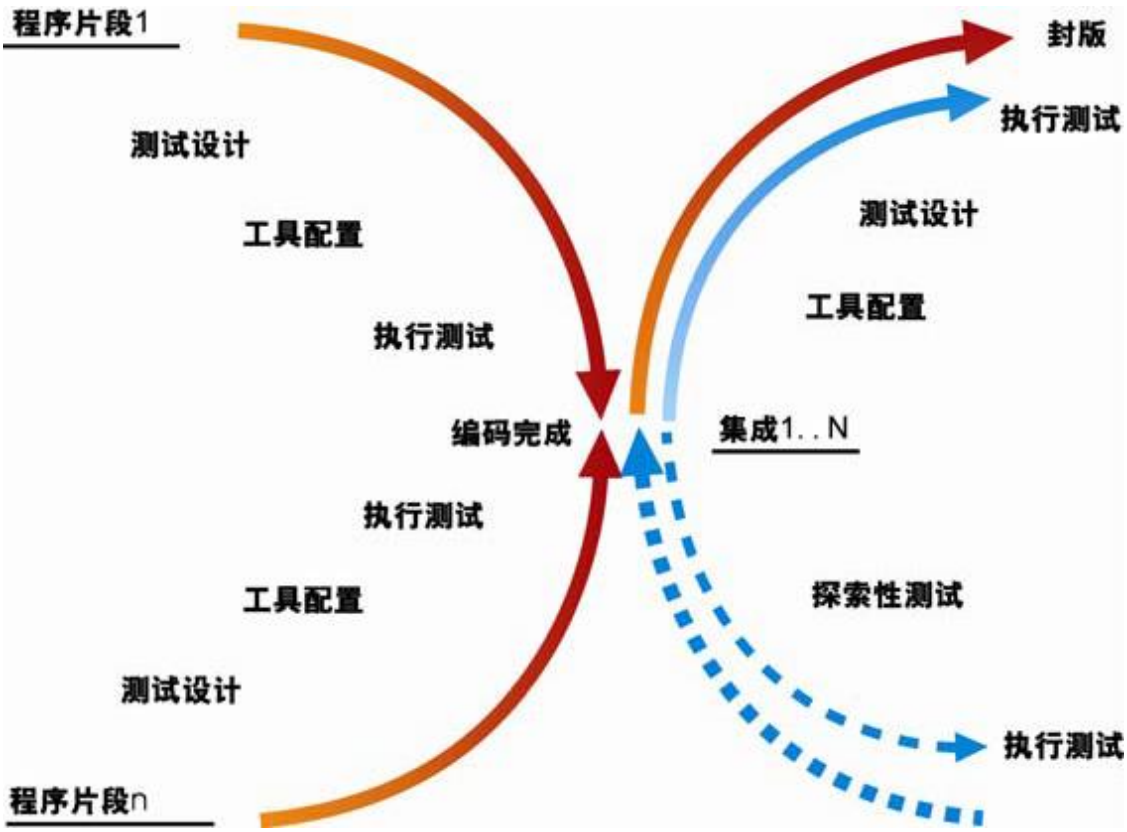
MDA（Model Driven Architecture）也是一个方法论层面上的名词。它讨论的是“创建出机器可读和高度抽象的模型”的方法。受 MDA 影响的开发活动被称为 MDD（Model Driven Development）。

与 MDD 在同一个层面上的概念是：

- TDD（Test Driven Development）；
- FDD（Feature Driven Development）；
- BDD（Business Driven Development）；
- R-TDD（Rapid-Template Driven Development）；
- CDD（Contract Driven Development）；
- RDD（Requirements Driven Development）；
- …

我不厌其烦地罗列这些名词，只想告诉读者一个事实：什么都可以“驱动开发”。不同的方案提供商基于自己的产品构架和当前的理论倾向，随时都在准备改变他们“驱动开发”的方式。在这种形势下的“XDD”或“XDA”，已经成为他们促销产品的保留用词。

回到软件工程的过程环节中来吧，你会看到，“以什么驱动开发”只是一个“以哪个环节为中心（或导引）”的问题。所以你会看到 TDD 中的 X 模型（也可参考 V 模型）是这样的：



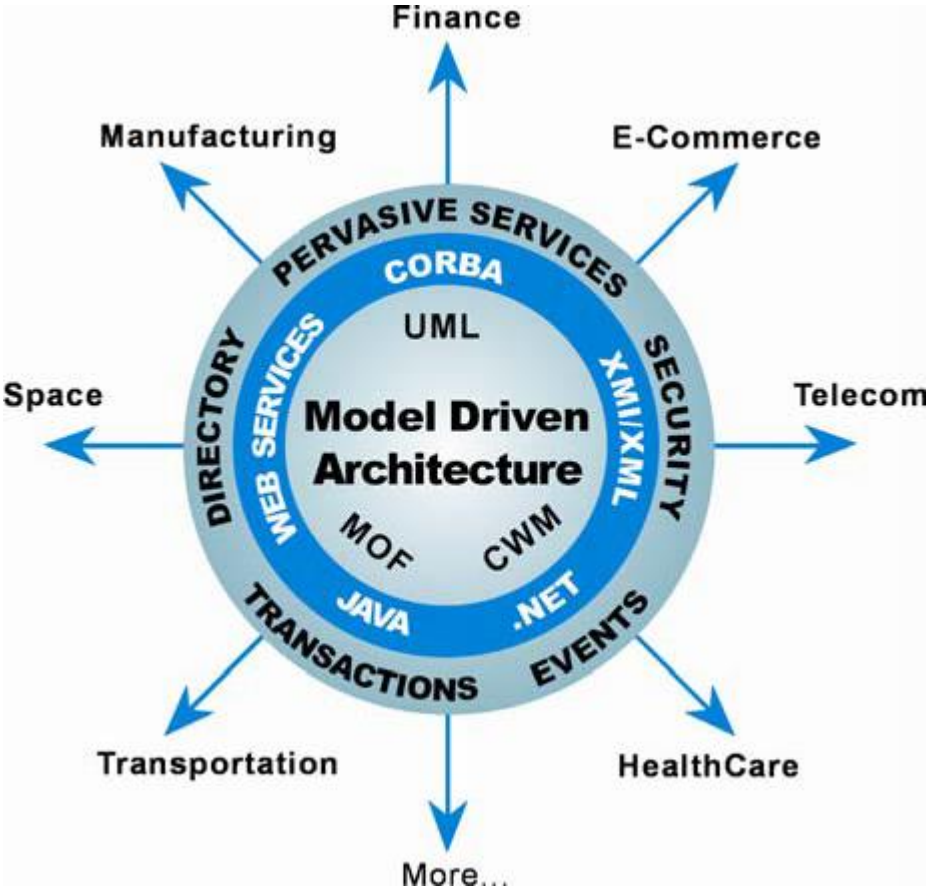
如果你仍旧不能明白为什么会有这么多被神秘力量所“驱动着的开发”，那么你就干脆去厨房找个平底锅烧点热油，然后敲下一个鸡蛋，很快，你就会体悟到“以蛋黄驱动开发”的真谛了。

抛开实现的技术细节不论，在工程中，“以什么驱动开发”其实是一个过程问题。而你应该明白，过程的选择（或制订）取决于你的工程需要，以及它在相关应用领域的适用性、过程工具的充备性和这个过程理论的完善程度，而不是大公司们的鼓吹。

过程模型决定了工程的实施步骤和组织方式。但是 Object Management Group（OMG）尽管对 MDA 提出了一套完备的技术和方法体系，工程实施者却无法在这个体系中找到一个可以适用的软

件过程模型——MDA 不讨论过程。

也就是说，MDA 架构作为一个新的软件开发方法的架构，即使在技术研究、底层协议和软件实现方面经过了持续地完善而渐至成熟，然而如果没有同样成熟的软件过程理论支持，那么它在工程中的实用价值也就有限，如图：



仔细审视一下这个 MDA，如果你现在就决定将下一个工程项目建立在这个构架的基础上，或者用 MDD 的方式来开发 BIOS，那么你离精神病就不远了。

第 8 章 具体工程[29]

> “齐人就赵学瑟，因之先调，胶柱而归。三年不成一曲，齐人怪之。”
>
> ——邯郸淳《笑林 · 胶柱鼓瑟》

[29] 全文主要来自我的博客文章《杀不死的人狼——我读〈人月神话〉》(2007.03)。本书纸质版中的“具体工程”一章也是引述这篇文章，并进一步明确地提出和阐述了“具体工程”的思想、方法、原则与逻辑。本电子版只收录了文章内容，而并未展开讨论。

第 1 节 我读《人月神话》

我对《人月神话》的内容结构做过一些分析，并大概地统计了一下该书在第 18 章里列举的观点，其中：

表 4 对《人月神话》内容结构的分析

章	现象	答案	本质	章	现象	答案	本质
1	3			9	7	7	2
2	10	1	1	10	7	4	1
3	3	3		11	21	6	2
4	3	4	1	12	15	3	1
5	3	2		13	13	4	
6	3	5	1	14	13	4	1
7	5	14	2	15	9	5	1
8	10		1	统计	62%	31%	7%

上表中分出了三类：现象、答案和本质。通常我们总是能给出“答案”，但未见得触及“本质”。例如街口的乞丐向我伸出手来，我给了他十元钞票，我给出了解决了他伸手（这个问题）的答案，但没并有触及他伸手的本质：饥饿；更未能触及整个事件的本质：贫穷（或者懒惰）。另外，在标注成“现象”的项中，有一部分是包括某种现象的成因的。我最开始分析时，曾定义“原因”这个分类，但后来发现原因其实也是一种现象，所以我合并了它们。

对“现象－答案－本质”的分析存在主观的成分，因此你可以重做这

个实验。但我建议你谨慎使用“本质”这个标签。至于其它两种，即使你混淆了，也不是至关重要的——尤其是现在，很多 Brooks 先生曾经给出的答案已经变成了思考同类问题的现实现象[30]。

你可以在工程中应用这些既有的答案。但是无论这些“解／答案”看起来如何合理，如果脱离它本质上讨论的对象，那么就可能不是正确的解。而另一方面，如果作者的逻辑足够清晰，那么他提出的“解／答案”必然是围绕着某些本质的东西。在上面的列表的分析过程中，我只看到这样的几点本质：

表 5《人月神话》一书中的几点本质

本质含义	原文	注
项目在定义阶段就发生了错误	2.6我们围绕成本核算的估计技术，混淆了工作量和项目进展。人月是危险和带有欺骗性的神话，因为它暗示人员数量和时间是可以相互替换的。	
概念不完整＝定义不明确＝无法实施	4.1 “概念完整性是系统设计中最重要的考虑因素”	注1
形式化会带来精确的定义	6.3出于精确性的考虑，我们需要形式化的设计定义，同样，我们需要记叙性定义来加深理解。	
组织是交流（沟通）的结果	7.1巴比伦塔项目的失败是因为缺乏交流，以及交流的结果——组织。	
组织的目标：减少必要的交流和协作量	7.16 团队组织的目标是为了减少必要的交流和协作量。	
小型程序与大型程序不同	8.2 构建独立小型程序的数据不适用于编程系统项目。	

[30] 但这些在我看来，还只是“现象”。Brooks 的持续思考也是现象，所述的言论也是现象。我们既不能因为其过程，也不能因为其结果而坚信这些观点：在决定全盘接受之前，至少要看清楚盘子里的东西。

(续表)

本质含义	原文	注
私利性是本质问题	9.6 在大型的团队中，各个小组倾向于不断地局部优化，以满足自己的目标，而较少考虑队用户的整体影响。这种方向性的问题是大型项目的主要危险。	注2
数据表现形式是编程的根本	9.16 更普遍的是，战略上突破常来自于数据或表的重新表达。数据的表现形式是编程的根本。	
项目经理的基本职责	10.9 项目经理的基本职责是使每个人都向着相同的方向前进。	
产品交付的关键是质量的保障程度	11.7 “开发人员交付的是用户满意程度，而不仅仅是实际的产品。”（Cosgrove）	注3

(续表)

本质含义	原文	注
用户需求变化的根源	11.9 软件产品易于掌握的特性和不可见性，导致了它的构建人员（特别容易）面临着永恒的需求变更。	
某些计算机资源不能总是方便的得到	12.4 目标机器的使用需求量是一种特殊曲线：刚开始使用率非常低，突然出现爆发性的增长，接着趋于平缓。	注4
里程碑的性质 / 定义	14.4 里程碑必须是具体的、特定的、可度量的事件，能进行清晰能定义。	
程序＝用户认识＋机器认识	15.1 对于软件编程产品来说，程序向用户所呈现的面貌与提供给机器识别的内容同样重要。	

注 1：这里“精确定义”是本质，形式化只是答案。

注 2：对组织中的个体或组织的局部来说。

注 3：产品问题不是本身的“完成度”的问题，而是用户可感受到的

> 质量问题。
>
> 注 4：书中例举的是“调试环境和目标系统”，但可以引申到例如“目标用户”或者“客户现场”。

我们应该清楚：现象之存在与是否被发现无关。例如苹果从树上掉到地上是现象，你看见这个现象也并不体现你的伟大，你四处大叫“苹果掉地上了”会被人当成疯子。而牛顿没有被人（因此）看成疯子的原因[31]：现象只是引起了他的注意，而探究到“本质”才是关键。

所以上表列出的“62%的现象”只是 Brooks 从四十年前就好心的提醒我们：看啦，快看看这些奇怪的现象，你难道不觉得它们奇怪么？于是，我们开始关注这些，并把它们当成关注的焦点。我只能因此承认 Brooks 是一个醒客，但这并不表明“陈述现象”，就等于告诉了我们什么真理。

接下来我们讨论“31%的答案”。

第 2 节 预言——《人月神话》及其地位

先民们所说的圣人以及通神者，皆因他们多数时候在正确地预言自己的现实。只有当这个“多数时候”变成少数的时候，先民们才会置疑圣人和通神者的能力。

然而事实是：我们现在的很多工程知识，——无论是从书上看到的，还是从实践中体验到的——大多未曾脱离《人月神话》之所言。如今只要论及工程（且不要让人认为是离经叛道），那么所讲述的一定是 Brooks 的这样的经验以及由此推出的观点，或者在不违背这些

[31] 我并不打算讨论这个故事的真实性。

经验和观点上的一些具体的实作方法！如下：

表 6《人月神话》一书中的经验与观点

原　文	基本含义	现　实
规格说明的风格必须清晰、完整和准确。用户常常会单独提到某个定义，所以每条说明都必须重复所有的基本要素，所以所有文字都要相互一致。这往往使手册读起来枯燥乏味，但是精确比生动更加重要。(P46)	重复所有基本要素，以便于单个的特性可能会被抽离出来进行讨论。	RUP
将来的规格说明同时包括形式化和记叙性定义两种方式。(P46)	用形式化来精确定义用记叙性定义来被充说明。	UML
使用实现来作为一种定义的方式有一些优点……（但）可能更加过度地规定了外部功能。(P47)	陈述实现并不是设计中规定外部功能的方法。	UserCase（注1）
对软件系统的体系结构师而言，存在一种更加可爱的方法来分发和强制定义。对于建立模块间接口语法，而非语义时，它特别有用……(P48)	寻求一种描述功能而不涉及实现的方法，来协助架构师陈述它们的设计。	Interface

(续表)

原　文	基本含义	现　实
项目工作手册不是独立的一篇文档，它是对项目必须产出的一系列文档进行组织的一种结构。 项目所有的文档都必须是该结构的一部分……(P54)	项目工作手册应当有组织、有结构地陈述项目各个方面的细节	RUP
笨拙的文字归档工作确保了所有变更会被阅读，这正是工作手册要达到的目的。(P56)	确保变更不会丢失。	需求管理系统或任务管理系统
是因为控制序更加复杂,所以需要更多的人员？或者因为它们被分派了过多的人员，所以要求有更多的模块？是因为复杂程度非常高，还是分配较多的人员，导致花费了更长的时间？没有人可以确定……(P64)	随时关注生产率并控制它。	项目管理软件

(续表)

原　文	基本含义	现　实
但是只有书面计划是精确和可以沟通的。计划中包括了时间、地点、人物做什么、资金……(P75)	以书面化的形式来制定计划，并且确保一些要素的准确性。	项目管理软件
试验性的系统必须被构建然后丢弃……(P77)	做一个原型并准备好扔掉它。	原型过程
目标上的一些变化无可避免，事先为它们做准备总比假设它们不会出现要好得多。不但目标上的变化不可避免，而且设计策略和技术上的变化也不可避免……(P77)	为变化而做出设计。	延长设计和迭代的周期。 风险评估。
流程图是被吹捧得最过分的一种程序文档。事实上，很多程序甚至不需要流程图，很少有程序需要一页纸以上的流程图。 (P107)	编程的结果产生流程图（以供讨论和分析），而不是对照着流程图进行编程。	
试图把信息放在不同的文件中，并努力维持它们之间的同步，是一种非常费力不讨好的事情……(P108)	文档应该与代码同步	代码文档化。
没有银弹(P114)		（注2）

注 1：在用例（UserCase）中不应指示或暗示实现的方法。

注 2：所有曾被认为是银弹的东西都被无情地否定了。

软件工程一些论者全然不顾书中所言是现象，还是本质的推论，或者只是一个由现象归结的、未必正确的答案——尽管这些答案，在大多数时候都恰如预期地出现在你的现实工程中。但如果我们理性地思考这一现象就应该知道，其实并没有预言未来的人。所谓预言，

大多数时候是两种情形导致的假象：

- 他做出了正确的判断；
- 你主观地跟从了他对未来的设定。

后者是危险的。大师们预言了未来也就改变了未来，即便未来未必“应当”如同他所预言的那样。

第 3 节 7%的本质

但如果这种预言的前提不正确，那么未来必然脱离这种影响而回到它应该的状态上去。如同我们看到的另一些事实一样，有很多现象表明，我们正在回归工程本相的道路上摸索前进。我们也发现，在大多数情况下，先哲们的预言在实践中被印证着，只是偶尔“不太灵光”。

对此，我们回顾第一小节，在《人月神话》中的那“31%的答案”的前提——也就是那 7%的本质中，如下两项是明显存疑的（也是我的主要置疑）：

- 目标的本质：是大型工程，是系统项目，而不是程序
- 个体的本质：是私利性的

其实早就有人意识到个体的本质“未必全是私利的”，尊重这些个体就会带来一些效果。例如 AP（Agile Process，敏捷过程）正是因为更加尊重开发人员的个性、能力以及相互间的合作，从而得到了效率的提升。

再进一步地说，既然 Brooks 是设定了“大型工程或系统项目”这样的目标，并给出了一些答案。那么在“小那么一点点的”工程项目中，是不是这些答案就不必须了呢？例如 Brooks 的许多建议，对于某些

目标就并不是很有效——例如你要三个月的时间才能开发一个产品；或者根本无法实施——例如你的团队总共只有 6 个人，连“外科手术式的团队”都组织不起来。

Brooks 的答案对于同样的目标，以及在他所述的“本质”未能发生改变时，还是比较有效（或有实施的可能性）。因此上述一些例外，总是在上述的“7%的本质”被否定或被改变的情况下获得的。因而我们提出的问题是“如何否定或改变”这些难以撼动的本质。然而在我看来，Brooks 早已经在最佳位置上，给出了撬动它们的一个支点：

- Brooks 认为构建“独立小型程序”与“编程系统产品”是不同的问题。

Brooks 讨论的编程系统产品的规模到底有多大呢？我想至少应该是以 IBM 360 为参考的。《人月神话》中引用 Joel Aron（IBM 在马里兰州盖兹堡的系统技术主管）的例子时说，“大型意味着程序员的数目超过 25 人，将近 30,000 行的指令”。按照书中的数据（人均效率 800 指令/人年），这个“大型项目”应该需要 1.5 年才能完成。此外，还需要大约一倍的人工，来负责除开代码之外的测试、管理、文档和沟通等工作。

好的，如果你有一个“50 人，开发一年半”的项目，那么你可以先接受 Brooks 的答案去实践一下：起码你可以有时间来讨论工程问题，也能够组建那样规模的团队。但是，难道只有这样的“大型工程”才算得工程，而“小那么一点点”的就不算吗？

现实是，我们一方面在做着“小那么一点点的”工程项目，另一方面在听着整个业界喧嚣着“为更大规模的工程”而准备的工程理论。我们总在实践 Brooks 的“答案”或者“预言”，而忘却这些答案的前提：

- Brooks 的经验源自对 IBM 360 等大型项目的实践与分析；

- Brooks 所述的工程是要得到编程系统产品；
- Brooks 认为编程系统产品的工作量可能是独立小型程序的 9 倍（在实现大致相同功能的情况下）。

事实上我们现在的软件工程的发展是被驾驭了，而不是被预言了。从本质上来说，Brooks 在《人月神话》中只是讨论了大型工程的实施，以及相应规模下的团队建设。而我们，便按照这样的设定来铺开了整个软件行业的工程化实施。促成这种现状的，并不仅仅是一本书的力量，还在于商业的力量。因为只有在这样铺展开来的行业环境中，才可能有商业机会。——即使那些工程顾问与实施专家从来没有实施过“50 人，开发一年半”这样的项目，只要他们能报出 Brooks 的名字，能谈及某些工具在应对“大型项目”中的成功经验，他们就已经成功了一半了。

为什么“敏捷”之初颇受争议？为什么敏捷对一些中小型的团队显得有效和可实施？为什么当这些争议被摆在眼前的成功平息之后，传统工程的理论家们却不忘恨恨地评上一句：那是一种不能（或难以）应用于大型工程的方法呢？！

因为如果大家都很“敏捷”，都只做比这些大型工程“小那么一点点”的工程，那么传统工程的专家们就失业了。反过来，只有把工程做大，大到“敏捷”失去了意义，而“庞大”变成了实质的时候，传统工程就可以为任何失败找到借口：看啊，Brooks 就说过“没有银弹”嘛。

第 4 节 没有银弹，或人狼杀不死

人狼这个动物很奇怪，皮肉坚实还是自疗系的，所以要么砍它不动，要么杀它不死。这种动物如同习得（传说中的）金钟罩功夫，刀枪不入，水火不怕。也如同金钟罩有罩门一样，人狼对银没有免疫，

因此一颗银弹就能穿透它，进而杀了它。

所以人们总是说一物克一物，大象怕老鼠，总有对付它的法子。但如果有人设定了一个能自圆已说的悖论，那除了否定悖论本身没有意义，也就没有解它的法子了。同样的道理用在“没有银弹”这个观点上，也是成立的。

也就是说，如果我们讨论“有或者没有银弹”，那么应该先反过来看看“人狼”的本质。因为本质是人狼对银不免疫，所以我们才能找到银弹并杀了它。如果人狼根本就杀不死，那么不要说金弹银弹，就是核弹也没用——因为它杀不死。

我们来看看 Brooks 所谓的人狼，也就是“软件活动的根本任务”。首先，Brooks 认为我们并没有足够的精力来放到“软件活动的根本任务”这一目标之上。他的论证过程是：

- 根本任务的目标：抽象软件构成的复杂概念结构；
- 次要任务的目标：表达抽象实体，在一定范围内映射成计算机的执行逻辑；
- 我们大多时候在关注次要目标，例如写程序和开发“写程序用的程序”；
- 我们写再多的程序与再强的“写程序用的程序”都不会触及到根本任务。

进一步的分析来说，我们探索目标的方法，分散了达到目标的力量。我们在通向目标的路线上越是努力，那么我们的力量就被分解得越快。次要目标是达到主要目标所必须的，但次要目标上花费越多的精力，就越无法接近主要目标。既然要经过 A 才能达到 B，而经过了 A 也就没有力量达到 B。那么结论自然是：达不到 B（主要目标）。

这个悖论说的是手法问题。你当然可以超越某种手段，纯理论地推论出：没有了 A 就成了，我们可以由 C 达到 B。由于永远存在 C 至

*（任意）的途径，那么当然存在 C～B 的途径。这样的手段显然无以穷尽，所以 Brooks 并不能据此来说服大众。

于是，Brooks 立即又论述了这个人狼的具体特性：复杂度、一致性、可变性和不可见性。

- **要面对极端的复杂性。**尽管我们可以用模件复用来缓解复杂性，但是软件实体的扩展必须是不同元素实体的添加，这些元素“以非线性递增的方式交互，因此整个软件的复杂度以更大的非线性级数增长”。 所以你创建一个新软件就必然面临更多的（非线性级数增长的）旧软件中不能被复用的元素。所以在复杂性方面，人狼是自疗系的：越做越复杂，不可能变简单。
- **要背上不可丢弃的历史包袱。**由于 Brooks 强调新的软件需要保证跟旧的软件兼容（有点象 MS Vista 兼容 MS DOS），你创生了一个软件也就创生了下一个软件的需求，所有的创生活动产生了需求的自增集合，尽管这种“变体不是必需的”，但它一个不可丢弃的历史包袱。所以在保证一致性这一方面，人狼是自增长的。
- **要接受需求的持续变更。**软件要保证设计一致性才能成功，但从这个软件被设计的那一刻开始，你就必须接受来自它人的、自身的、市场的、自然及社会规律的，以及不同的文化和思想习惯的差异的需求（这意味着每个人的想法都可能被作用在一个软件实体上）。需求是无度和不可控的，所以人狼本身又是变形系的。
- **不可见。**你找不到足够的抽象方法描述软件的不同侧面，也就不能将它们表达为抽象概念上的图形。如果你找到了这样的方法，那么这个“软件”本身就不足够复杂，因此也就不是原本含义上的“根本任务”。所以，它是隐形的，你如果看见了它，要么是看见了诸多复杂的方面中的一面，要么根本就是看错了。

从游戏术语来说，我们要面对的是“自增＋自疗＋变形＋隐身”的终极大 BOSS，而 Brooks 还要求：HI，小子，你得拿个足够简洁的武器（例如小刀？），然后去单挑（独立的解决方案？）。

如果有游戏策划写出这样的脚本，那么他得被玩家活活骂死。但 Brooks 描绘了这样一只“杀不死的人狼”，并开心的说“你们没有银弹”。然而，他不但没有被骂死，还得到了一致的认可，并且整个工程界欢欣雀跃，一致以找出那枚银弹为已任。然而每个在这个方向上努力不已的人都没有看到，Brooks 声明过：构建独立小型程序的数据不适用于编程系统产品。

大师的意思是：因为不能通过“做更多的小型程序”来得到做大型系统的经验与数据，所以无论何时，只要面对大型工程，你的经验值就立即归零（或者极低）。显然，（连白痴都知道）毫无经验值地直接面对终极大 BOSS，结果一定是失败。又由于所有面对这些大 BOSS 的尝试都无可置疑地失败了，因此我们也就不可能有成功。

显然这是一个法宝：如果你违背这个逻辑而又获得了成功，那么这种成功可以立即被归结于：你在做一个小型程序。

放心吧，是没有人能杀得死 Brooks 的人狼的，也不可能找得到这样的银弹。因为 Brooks 的人狼原本就是杀不死的，他甚至连“给睡熟的人狼胸口一刀”这样偶然性的机会也没给你留下。任何时候，你杀死了一头看起来有点象是人狼的怪物，Brooks 都可以轻描淡写的说：OH，小子，你看错了，那并不是人狼。

第 5 节 广义工程、狭义工程，与具体工程

现在我们回到一个实际的问题上：工程的本质需求是什么？如果我问一千个人工程的本质，可能会得到一千种答案。因为大家离本质的东西都很远，又从不同的角度去看这本质，故而得到的答案并不相同——而且每一种答案都貌似正确。

但是我问的是“本质需求”。对此，我的答案是：本质需求是“实现（工程的目标）”。

不管工程本质是怎样的，但这个需求如一。我们完不成它就等于失败，至于它是否是 Brooks 所说的“软件活动的根本任务”，与这个具体的工程无关；至于它是不是从人类发展的历史上来看，向着“软件活动的根本任务”的目标迈进了一步，也无与这个具体的工程无关。

简单地说，我们做“具体的工程”的目标并不等同于 Brooks 所说的“软件活动的根本任务”。而回到“这个工程”的视角，我们面临的可能是：

表 7 具体工程中的现实业务

a1:自研发产品	a2:客户投资的项目
b1:短期效益的市场探针	b2:战略方向上的一项长远计划
c1:一个小型而称手的工具	c2:一个大型的可持续平台
……	……

这个对比中，a1 的需求基本可控，a2 就得面对客户或业务规则的变化；b1 不需要背上历史包袱，甚至不用考虑架构一致性，而 b2 就必须做好设计并面临长期用户需求的沉积；c1 可能一两个人就完成任务，c2 就必须组织大型团队.....我们如果要找一个方案来适应这所有的“软件开发活动”，那么它只可能是弹性的。而弹性并自由伸缩的一个方案必然带来学习和运用上的困难，因此你又得投入人力来学习使用，并在实施中不时监控它。——于是，你的人力和时间成本又增加了。

不要去相信“它适合于所有的工程”这种商业推广的鬼话。那绝对是

赤裸裸的欺骗。你要看住你的口袋，因为你可能在消耗你的资源（时间、人力与金钱）去适应一种方法。你：

- 一方面要花销资源去组织、实践和推动这些工程方法
- 另一方面工程的规模可能根本没有资源去推动它们

所以你必然面临失败；如果你：

- 增加资源去推动，那么成本可能大于项目利润
- 你的老板会不乐意而直接中止这个项目

你还是失败。

所以问题出在你启动这个工程的最早阶段：你认清目标并决定用什么方法来驱动这个工程。你的选择涉及到项目的各方面因素，而不是昨天从某个培训中听到的什么奇怪方法。作为合格的项目经理（和工程决策者），你必须要洞悉各种工程方法的应用环境、代价，也必须清楚所在团队或公司的规模与实力，同时还要了解团队的优点与弱点。只有充分评估这些因素，你才可能决策在某个工程中应用的方法，或尝试之。

但是我们总是会遇到无比庞大的项目，这绝不是“只做做小项目”就可以解决得了的问题。然而我认为 Brooks 的假设过于学术，因为我们找不出一个理由来证明：需要一种纯粹的、独立的和并不那么复杂的方法来实施一个工程。对于一个具体的、哪怕是无比庞大的项目来说，这种学术的纯粹和完美也不是必须的。在这个具体的工程项目来说，完成它是必要的，而寻找完美方法，只是在完成这个项目之后进行总结时的一种附带价值。

换而言之，即使我们承认 Brooks 所说的“软件活动的根本任务”需要一种完美的、类似于银弹的解决方案，我们也不得不承认对于具体

的工程项目来说，“表达抽象实体，在一定范围内映射成计算机的执行逻辑”——亦即是“实现”，才是根本任务。

我们看到了分歧。而我认为对这个分歧的合理的解释是：

在广义工程与狭义工程（或言之具体工程）中，主要目标与次要目标正好是倒置的。

对于后者来说，将需求表达为抽象实体，并在“一定范围”内映射成计算机的执行逻辑，正好是这个工程存在的根本价值。如果这项目标不能被达成，那么这个狭义的工程既不可能实施，也不可能为所谓广义工程产生任何价值。

由此，我对“广义工程”的定义是：

- 面向“软件活动的根本任务”；
- 程序实现的过程无助于求解根本任务。

而我对“狭义工程”的定义是：

- 面向“具体工程活动的需求”：实现；
- 将“实现”作为一个过程或结果，则求解“根本任务”只是其探索活动或附加价值。

既然对于狭义工程（哪怕它无比庞大）来说，Brooks 所设的银弹只是面向其次要目标的，那么它也不是必须的元素。因此，我们尽可以选择集束炸弹来解决问题。我们可以从建筑工程中学习监理，可以从制造工业中学习模件，也可以从哲学中学习概念抽象，我们还可以从社会学中去了解集体、组织、规模和控制.....

一切一切，前提是我们要放开对那个“银质子弹”的追寻。这样，面对一个确定的工程对象，我们便可以找到很多问题的解法。但如果

坚持存在一个“绝对正确的解法”，那么任何实际问题都会延伸出枝节，蔓布于这个“绝对正确的解法”的墙篱之外。正是我们对广义工程及其成效的迷信，影响了我们的决策、尝试与实践。

这就是现状[32]。

第 6 节 告别伪命题

总的来说，我事实上并不反对某种具体的方法，而是在努力的学习种种方法。但是我并不认为有什么单一的方法能解决所有问题。每一种解决方案都是有其前提和背景的，精通一种或全部工程工具、方法和过程，都无助于掌握这些前提、背景以及潜在的关系。理解工程的适用性，涉及到工程专家的整体素养和实践经验，也涉及到具体的工程环境、目标和实施者（团队）。而这些课题，却正好是目前的软件工程甚少谈论的。

在银弹的话题上，答案不外是：有、没有和将来一定会有。我的答案与这些都不一样。我认为“有没有”的话题没有意义，不值得讨论。我进一步的观点是，广义工程对首要任务（人狼）和完美方案（银弹）的假设，不应该成为狭义工程所追求的终极与利器。我们要分清狭义工程与广义工程。正是由于他们对本质需求的设定完全不同，因而也有各自不同的主要与次要任务。一切工程活动的历史告诉我们，曾经的经验、失败和成功，以及据此在广义工程中归纳推演的理论，都只是我们在具体的、狭义的工程中的参考，而非无往不利

[32] 广义工程，或人狼之有无与生死的论题，除了给商家制造更多的商机之外，并不会因为它的争争吵吵而给狭义工程带来什么价值。事实上，正是狭义工程给广义工程提供了谈资和论据，才让后者变得越来越丰富。也正是因此让更多人憧憬于那颗银弹的存在，而全然忘却：一颗子弹的威力，原本是出自一个并不成功的丹药实验。

的银弹。

曾经，我们如同走在一片沙漠，我们看不到沙漠的边缘，也不知道如何行走。Brooks 的伟大贡献，在于他用《人月神话》指出了一条道路并教给我们基本的行走方法，让我们从中学会了辨识流沙，懂得了沙暴的征兆；同时“没有银弹”的假设激发了我们的斗志，如同有人在说“你要证明沙漠有边界，你就拿出一束青草来看看”。

于是有了两种喋喋不休于“有没有青草”的人。一种是已经在灵魂上上升到另一个层次，固而无需行走于沙漠。另一种则只是站在沙盘边上，用长杆推动卒子，而鞋子上根本不会沾上一点点沙土。

还有一种人则不时地宣传他们已经找到了青草。他们总能确保在你付费的时候看到青草，转身时才会发现那里并不是沙漠的边缘。

剩下的人一边喃喃着“沙漠边缘的青草”，一边在焦燥的沙海里缓步前行。不同的是喃喃而不自觉者掉进了沙坑，心怀憧憬而盯紧脚步的人走出了沙漠。

走出来，你才会觉得：原来有没有青草，并不那么重要。

第 9 章 是思考还是思想

> “此郎亦管中窥豹，时见一斑。”
>
> ——《晋书 · 王献之传》

第 1 节 软件工程三个要素的价值

思考问题的方法可以是由点及面的，也可以是统揽全局的。换成业界最常用的词汇，就是“自上而下”还是“自下而上”的区别。

“牛屎图”中描述的工具、方法与过程也被称为软件工程的三个要素。在本书中它们被分解开来思考，这并不是要孤立这三个层面——它们实际上是相互作用的。

例如“过程”问题，就既有实施过程的工具，也有相关的过程方法理论。我虽然说方法是“基于一种数据结构的编程实践的结果”，但这实在是一种非常狭义的定义。这个定义在过程的开发环节上是有效的（或者说是对“开发方法”的定义）。然而“需求”、“设计”、“测试”等其他环节也有各自的方法论，即使站在具体环节之外，过程本身也有方法论的问题，这还不包括管理方法等在内。

由于方法在过程环节及过程总体层面上具有贯通性，因此保证“方法（或其行为）”的实施的“工具”也会出现在过程的各个环节和层面上。这样一来，我们得到的软件工程模型将不是经典的、层状的“牛屎图”，而可能像太极图一样由阴阳交汇而生万物。

为了不使读者认为我已经入了道家理论的歧途，这样的一幅图还是交由你自己去画吧。只不过你应该清楚，即使画出了太极图的软件工程模型，你所见到的仍旧是工程的细部环节。就如同以管窥豹一般，斑是斑，豹是豹。

若你能把每一个“管见”拼合起来，你得到的才是“豹”，而不是“斑”。所以尽管本书割裂了软件工程的各个要素，并从每个孤立的层面来

审视，但实质上，你应该回归到软件工程的本体上来思考问题，而不是仅关注于每一个局部的要素。

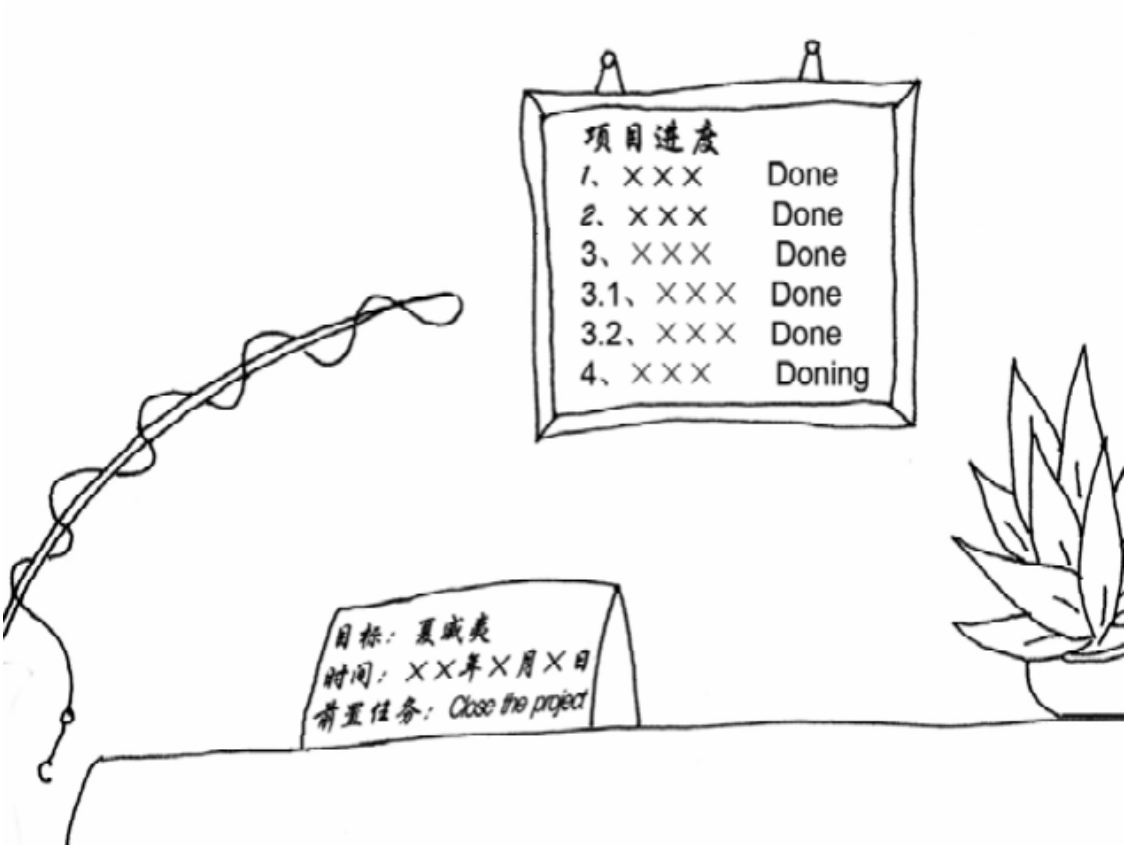
工程的整体问题仍旧是“实现”。

第 2 节 其实 RUP 是一个杂物箱

我或许总是在批评 RUP，但是我不得不承认它是对前人在软件过程思想方面的高度包容。

请注意我用“包容”这个词，而不是按照语言习惯那样用“概括”。因为如果是“高度概括”，那你应该把目光投向瀑布模型，而 RUP 其实就像一个杂物箱一样“包容”了全部的已知理论。

你可以把 RUP 定制成其他任何模型所表述的过程形态——RUP 本身的特质决定了这一点——因而它也如同一个杂物箱一样放满了各种稀奇古怪的东西。你可能从这个杂物箱里面拿出了一把剪刀，或一只苍蝇拍，或者是一根钓竿.....



喂，等等。面对“软件开发”这样的一个需求，钓竿能有什么作用呢？在你扔掉它之前，请转换一下你的思维：钓竿可能带给你的团队以精神上的激励。如果你能意识到这一点，那么它将立即转化为生产力：把钓竿挂在开发部的墙上。

RUP 能不能被用起来，将取决于你刚才那个挑挑拣拣的行为，以及现在你在拿到钓竿后的辨识能力与组织能力。

第 3 节 UML 与甲骨文之间的异同

在你真的打算用甲骨文来写项目文档之前，请先明确 UML 与甲骨文之间的异同。在这本书里，他们都被作为沟通的工具。因此目标是

沟通，而不是“选用工具”这件事本身。更进一步的推论是：即使你因为个人喜好而选择了甲骨文，也不要试图在结绳记事的原始人面前去用它。

UML 与甲骨文都是符号文字，都具有象形含义。然而这并不表明 UML 符号本身能表达多么丰富的含义。如果要像甲骨文一样用几代人、上千册的论著去解释它，那么 UML 图的价值也就只剩下象征性的意义了。

出于沟通的必要，这种语言的象征意义在一个图中应当被表述得足够准确和详细，乃至针对于不同的阅读者来说都能提供充足的信息。然而，一方面 UML 的规范中没有提供一个标准来衡量“怎样的 UML 图是描述充分的”；另一方面，UML 作为一个语言，也无法直接在某个环境或场景中被语法检错和调试。

所以在工程中使用 UML 图，应该有相应的文字来描述它。而且这种描述与图之间的对应关系要持续地维护下去。如果这种关系松散了、断裂了，那么下一个阅读 UML 图的人所面对的，将是无异于甲骨文出土时的困境。

好在做 UML 图的那个工程设计人员（在辞世之前）还有机会为这些古怪符号写下规约。

第 4 节 经营者离开发者很远，反之亦然

使我第一次意识到 EHM 模型反映了角色所关注的不同视角的人，是我的老板。事实上，他是一个完全不懂软件技术的老板。在 EHM 模型中，他所处于的位置在最右端，而开发者在最左端，在二者之间没有相同的关注界面（关注点）。EHM 真实地反映了“老板不懂技

术”的合理性，同样也真实地反映了“开发者转型为老板”的道路将是相当漫长与艰难的。

于是，项目经理这个中间角色就有了一种使命：协调经营者与开发者之间的沟通。例如招来一名开发高手，对于公司的运作并不会有深入的影响（当然如果你招来了 Anders Hejlsberg 就另当别论）。因此，我甚至不需要与 BOSS 讨论这名高手的来历及作用。同样，与一个技术分析人员讨论一个产品的技术价值与市场价值之间的差异，以及市场运作方式与技术实现手段的无关性，也是毫无必要的。

你要理解这种根源：角色的关注层面完全不同。

第 5 节 矛盾：实现目标与保障质量

在需求阶段我们就会面临“目标”的问题。然而（在大多数时候），与此相反的是我们会在项目交付和试用时才碰到客户在质量上的投诉。

需求人员会把所有的责任归咎到开发人员，而开发人员又不停地埋怨需求的不清不楚，或者变更的没完没了。又如果正巧需求和开发都是同一个人或者小组来做的，那么他们便会开始埋怨客户的苛刻及工期的紧张。

总之一件事，没有人会跳出来说：我们原本就错了。然而事实上问题可能真的出在源头：我们把目标定错了。

我们看到，在项目的平衡三角（时间、资源和功能）中讨论的是目标问题，而并不讨论质量问题。也就是说，经典教材中总是关注：如何更快地完成项目，并减少资源占用，以及实现更多的功能。然而，即使平衡了这种关系，项目的结果仍可能产生一个天生的残障。

因为目标可能在平衡中确立，但质量却要在过程中控制。即使在时间、资源和功能三者中取得了平衡，即使客户、项目组和公司同样满意于这个平衡“目标”，它仍然有可能是“不能实施”的。

如果原定的目标（的整体）本身就过大，那么无论如何平衡这三者之间的关系，其结果仍旧保障不了质量。

问题是：又有谁愿意在最初签订协议的时候，就降低或者放弃协议的标的呢？

第 6 节 枝节与细节

刚才说到目标和质量的问题时，提及“平衡时间、资源和功能三者的关系”。这其实是一个实施过程中的细节。或者说，它是一个具体的方法，而不是目的。

所以我们通常所说的细节，其实是对实施方法的一些有限量的描绘。比如“软件工艺”这个概念本身的提出，就是考究“细节问题”的。从这个角度上来说，我并不反对“细节决定成败”这样的观点。但请注意一个前提：这是技术或方法的细部。

我其实在前面的行文中一再地混用了“细节”与“枝节”这两个词。枝节是事实发展的次要的分支，它不涉及行为本身，也不是对行为本身的考量。因此我在前面的文字中说到“跳出细节”，其实本意是“跳出枝节”——细节只是做到何种程度的问题，而并不是关不关注（或做不做）的问题。

大多数情况下，管理人员有责任去审核、评估其他成员的工作成果。这个时候可以讨论“细节决定成败”这样的问题，因为这决定了产品的最终质量，而质量是工程的目标之一。

而在另一些情况下，例如管理人员做事件的决策的时候，就必须要学会忽略枝节问题。

混淆这两个名词的使用，其根本原因在于一大部分读者并不能区分“细节”与“枝节”。从惯于“实作”的程序员一路走来的工程人员，很难分清自己什么时候是在“工作”，而什么时候又是在“决策”。

因此我只好用最笨的方法提示管理者：别管它是细节还是枝节，只要你感到你的脚趾已经沾上了泥淖，就快点回头。

用脚趾去感觉，有时比用头脑去思考来得有效。

第 7 节 灵活的软件工程

并不象现代人想见的那样，古诗词一定是“逐字论平仄”的。变化或者变通，其实是常见之事。因此古词谱中，才常会见到冠以“摊破”、“减字”、“添字”等字的词格。然则古人在词格上的这种变通，是基于“音律”的。

通常说的词律是指词格，这与音律是两回事。词律(格)是平仄，音律则是乐器、音调与歌舞。古词中用来吟唱与歌舞的词牌就不能混用，律不同，调不同，如是之。

> “古人做词的变格，势必依音律而为之，舍周邦彦、东坡、辛弃疾此等人物，轻易变格，是为他人所笑话的。所以，词谱中所录变格既少，且俱为名家所创。”

然而古词的音律（亦即是律谱）已经失传了，也就是说，今天的词是用来读的，不是唱，也不是舞，甚至连吟哦也不是。所以今人总是拿普通话中的 1、2 声作为平声，3、4 声为仄声来填词，并以此论平仄，而全然不想词的格律的根基是“词律”与“音律”这两个部分的

融合。

我曾经参与过一个讨论，叫“古人是如何说话的”。在我看来，古人做文章和说话是两回事，文章中之乎者也，日常交流中还是市井俚语的。因此评论中会说“以俚语入词”。也可见填词作文章与说话毕竟是不同。再者说话也存在方言的问题，因此方言之间平仄音调也当不同。古代的歌妓是要求会“官话”的，这相当于现在“普通话”的地位，她们歌唱起来，也是用的官话。

更进一步的推论是：古代的词律中的平仄是以官话为基础的。然而如今的普通话毕竟不是古时的官话，也就是说，即使我们以普通话的四声为基础在论平仄，在古人看来，也是可笑的：这样做出来的词，依旧不可唱，也不可读。

因此今人做词的标准，是应该重定的了。除了词格(这里仅指字句的格式)和用韵之外，其它的部分是无法遵循的了。在各字的平仄以及句式上，应当以“能通顺”和“能品味”为准，风格上则以古雅为益。

仅此而已。

对于我这样的格律观点，一位网友曾有一句“未蕴而变，自欺也。知律而变，智者之道也”，实为良言。变向不变求。不变者，万变之所源，亦万变之所归。习诗词之法度，若蚕虫之结茧，若无结茧于前，何有破茧于后？故，知律而变，智者之道也。

“知律而变”中的“律”字，若解释作“规律”，那么便是可以用于软件工程中的了。“道”是规律，如果明“道”，而可以变化无穷，这样做软件工程才是活的。就如同今人难于填词一样，不明道，则不明智，不明智则无所以为，因而在软件工程实施中不可避免的盲目与停滞。

“知律”的另一层意思，是在于“知道原理”。明白“为什么要这样”或者“为什么不是那样”。这在软件开发中是常见的问题，大多数人不知究竟地使用着技巧和方法，而一旦出了问题，则归究于这些技巧和方法的不好。而真正的问题在于，这些人(我们通常叫做Copy&Paster)并不知道这些技巧、技术和方法的原理，因而不知道变通，也不知道回避错误。

所以死读一本《软件工程》的人不会做真正的软件工程。

所以我写《大道至简——软件工程实践者的思想》。

前言后语

后语

你现在读到的文字，原本应该是这本书的前言。然而如今，它却变成了后语。

如果读者是客户，那么我就应该是这本书的工程管理人员了。与第一次著书不同，这一次我已经尝试参与整个“做书”的过程。我必须要考虑客户问题，必须要知道作为客户或者读者这个角色的你，在想些什么，或关注些什么。

在这本书的最最最初的时间里，我把书稿给了我的老朋友王寒松。然而我很失望地听到他的意见：这个前言没法读——我想了又想，是的，因为有了玄而又玄的观点及曲折旁引的论述，所以的确不是太容易读。一方面，从我的角度上讲：

- 这个前言总括地讲述了我的观点。

而另一方面，从读者的角度上讲：

- 读者关注的是细节的文字；
- 读者享受的是阅读的快乐。

发现我在“做书”这件事中扮演的是工程管理人员这一角色的同时，我就决定了这篇前言应该改成后语。原因很简单，客户并不关注我

对观点的总括，至少一开始他们不会关注。如果要客户做结论，或者要他们与技术人员讨论结论，那就让他们在看完演示之后，在最最后的阶段去做。

读者的阅读行为决定了我将这篇文字放在这个位置：这篇前言既不是细节的文字，又不能给读者以阅读的快乐，因此它就应该放在后面。至于它是叫前言，还是改名叫后语，那只是形式的问题。

如果你觉得你的项目中还有一个模块不是用户所关注的，那么，En，建议调整一下明天的客户演示，把这一部分放到幻灯最后一页的后面，只有在客户提及时，才拿出来跟他们解说。

否则，如果他们不感兴趣，那么他们将永远看不到这张幻灯。一如这一篇前言。

软件工程

如果做一份软件工程中的经典书目，决不会有人漏掉 Roger S.Pressman 所著的《软件工程》。这本书有第四、五版梅宏教授所译的中译本。然而相信读这本书的人不会太多地注意到它的副标题是“实践者的研究方法”——从根源上说，它是讲述软件工程方法的书，而不是软件工程思想。

这有什么区别呢？拿这本书来作为软件工程活动的参考时，绝大多数的人不能明白类似如下的问题：

- 为什么要这样做呢？
- 我们这里应该这样做，但是接下去呢？
- 这个环节很重要，但是如果不做会有怎样的风险？
- 我们在做这件事的时候，其他的人在做什么？

- 为什么失败了？

相关的问题很多，但总而言之，这本经典教材更多的是在描述“怎样实作”，而绝少讲述“为什么这样做”。以致行为失去了思想的引领，能“完成”工程而不能“做成”工程，便是可以想见的事了。

任何人在实施软件工程的过程中，或者处于工程过程的某一个阶段的时候，都会有自己的思想或思考（哪怕是牢骚），那么为什么没有人写“软件工程思想”这样的书呢？

我与软件工程

我始终认为无论是哪家公司实施软件工程，都将是一部成功者的血泪史。然而不实施软件工程，则将是一部失败者的血泪史。换而言之，做软件工程可能流血流泪，但终究可能成功；而不实施软件工程，那就是抛头颅洒热血的失败了。

我所遇见的却是不打算实施软件工程的公司。在“誓死不做软件工程”这一思想的引领下，一家堪称河南省最具资本实力的软件公司于 2003 年至 2004 年间倒掉了。我在这家公司前后工作了七年。在 2003 年 2 月的时候，我开始请假在家写书，以一个绝对 Coder 的身份完成了《Delphi 源代码分析》。用这一年的写书时间，完成了我对这些年的程序生涯的回顾和反思，我看到了我在做 Develop Manager 和 Project Manager 过程中的得失，也透析了那家公司几年来的成败与沉浮。

我再次与总经理 P&J 对坐的时候，我们又讨论到公司的问题。他依旧固执地认为“最重要的是人的问题”。我看不到他对管理、工程和决策上的任何反思，于是我终于辞职了。

2004 年 3 月，我开始应职于一家新的软件公司。因为规模小，所以实施软件工程的风险也就小。在一次公司内的软件工程培训中，我突然意识到工程实践与工程思想之间的差异与关系，同时也看到《软件工程——实践者的研究方法》一书根本性的不足。时值我那本《Delphi 源代码分析》将近完成之时，于是我匆匆记下当时的想法，并确定了这本新书的名字《大道至简——软件工程实践者的思想》。

大道至简

直到现在[33]，这本书的基本目标仍旧与它最初定名时的目标一样：

- 这是一本小书；
- 只用于读与思考，没有实作。

所谓“小书”，是我不想做成教材或者宏论。思想应该简明，阐释应该清晰，而读者应该更多地去思考，而不是跟随这本书去完成什么。

老子说“道之为物，惟恍惟惚”。道是要体悟的，而不是像做木工活那样是“会与不会”的问题。道是什么呢？“道是本体，是规律，是自然”，简而言之，道是既已存在的事实和影响事物发展的规律。

这里需要说明的是，道并不是人为的规则，而是事物本身特质的规律。因此，本书中所要讲述的重点是这种规律。即使提及一些“实践规则”，也是在对规律讨论之后。读者应该发现这些“人为规则”是那样地遵从于“本质规律”。

[33] 我写书的习惯是先写前言，这相当于大纲。因此所谓“现在”，是指我写下前言的这个时候：2004 年 11 月 01 日凌晨 5 时。

经常听到的一句话是“规矩是人定的”，因此也要“靠人来推翻”。但是（初级的）软件工程实施者经常抱着一些经典的教材亦步亦趋，此谓之曰“知其然而不知其所以然”。无僭越便无建树，无大成者。

画眉深浅入时无

我不是太喜欢写很“入时”的东西。“入时”的往往是新的，因而也就乏有研究。这样的东西流于口头的讨论是可以的，然而著书立说，是要将心得之见或谨严之论呈现给读者，不是把自己想说的话说出来就可以的了。

在写《Delphi 源代码分析》的时候，Delphi8 都已经发布了，Win32 的时代也已近末路。促使我写那本书的原因，在于没有人用 Delphi 来研究操作系统的内核机制，而 Delphi 的源码对这些的实现细节实在是宝藏。绝大多数用过 Delphi 的开发人员，入源代码之宝山而空回，实在令人痛惜。因此那本书能否卖得了钱我是不在乎的，我在乎的是读过那本书的朋友，能从编译器的角度上对 Win32 体系增加多少了解。

从 Delphi7 的时代我就开始接触.NET Framework。2003 年的时候给 BorCon China 做演讲时，我已经对 Borland 在底层上为 Delphi.NET 的实现非常了解了。因此如果以“入时”（以及“适时”）而论，在《Delphi 源代码分析》完成之后，我应该写的书是《Delphi .NET 源代码分析》，来全面讲述 Delphi7、Delphi8 和 Delphi9（Delphi2005）中对.NET 开发的实现。

这个计划被我搁置了。

如今在我看来，语言其实是开发的细枝末节，而在大学时代、在课

桌上令人昏昏欲睡的《软件工程》才是软件开发中的髓质与灵魂。十年的软件开发实践中，其实在很多时间里我都落入了细节陷阱。

“实现”的欲望是从程序员出身的管理者的通病。因此如果你仍然在思考选择什么语言、如何重构，以及在开发部里争论一段代码有没有或应不应该采用某种模式，那么请你暂时沉寂下来，听我说：那是细节。

真正的问题是：你的老板要求你下周二就给客户演示这个系统；而客户并不关注你的实现细节，他关注的是你本月月底是否能 Close Project。

软件工程首先关注的是以客户为对象的、整个工程的成败和质量。根本上说，技术性、重用性，等等，只是保障工程成败与质量的手段而已。

重要的东西往往并不入时[34]。例如你的 ThinkPad 还在工作，仅仅是因为电池还没有用光。

知之、好之、乐之

从读者的角度上来讲，是“知之不如好之，好之不如乐之”的，因此作为作者，则希望自己的作品使人“以之为知，以之为好，以之为乐”。在写《Delphi 源代码分析》时，我的书稿的第一页就写着“知之、好之、乐之”，然而那本书仅能给人以知识，让人“知道”就很不错了，况乎乐哉？

[34] 你当然也可以由此反推出第 7 章的部分内容并不重要。的确，那只是我思考事物的一种方式，我希望你看到本书中讲的思想是如何被实例化的。但对于本书来说，如同我一再强调的那样：这是枝节。

读书给人以痛苦之感是有可能的。如果读《Delphi 源代码分析》不感到痛苦，那是没认真读。然而我毕竟不是想让人（或者想授人）痛苦的，将《Delphi 源代码分析》写到那般地步，非我所愿。

读《大道至简》的话，就用不着这样了。我虽然做不到让读者“以之为乐”，但“以之为好”还是可以的。我希望读者可以轻松地将这本小书读完，然后便可以束之高阁了。毕竟这本书不是理论，也不是方法论，只是思想。

思想已经领悟，文字的、纸质的东西还有什么价值吗？

致谢

感谢 P&J 和 Danny.Chou。他们给了我七年的从业经验，看着一个小公司做大，又从一个大公司做到消亡。这些经历深深地影响了我如今思考问题的方式，以及决策的方法。P&J 的知人善任和用人不疑成就了我的用人观，而以 Danny.Chou 为鉴，则同时形成了我对技术的执著与背离的两种态度（用于不同的思考场景）。

再次感谢 P&J 在 1997 年成功地说服我留任西南区市场经理一职。如果没有那时的转变，我想我至今仍然会困宥于程序员的这一角色。

感谢我的父亲母亲。父亲在我 12 岁之后的教育上是成功的，他留给了我足够的、独立的思考空间，以及面对事物的决策权。我至今仍然记得我的第一封信，是写于 9 岁那年的一封家书，这是我第一次用笔写课本之外的东西。这是母亲的功劳，她成就了我对文学和文字的喜好。没有他们，我不会有今日的观点和表达这些观点的能力。

感谢 Joy。En，我还活着，是因为她无微不至的照顾。感谢上帝把她给了我，成了我的厨师、司机、保姆、听众、苦力工、开心果，

以及，我最爱的妻。

版本历史

第一版：2005.11.06，电子版

原始版本。

第二版：2007.03，电子工业出版社出版

添加了两章三节的内容：

- 第 6 章，谁是解结的人；
- 第 8 章，你看得到工具的本质吗；
- 第 4 章第 3 节，沟通的三层障碍；
- 第 9 章第 5 节，审视 AP 和 XP；
- 第 10 章第 7 节，细解“法”与“式”。

新加入九幅四格漫画和一篇《愚公移山记》。 修改了一些章节的位置。 做了一些细部修改。

第三版：2010.01，点评版，电子工业出版社出版

增加了 6 位专家的 134 条点评内容。

新写了第 10 章“具体工程”。

第 11 章（原第 10 章）的“第 8 节 灵活的软件工程”做了较多的删减。

去掉了一些正文中的插图。

做了一些词句上的勘误与修改。

第四版：2012.07，典藏版，电子工业出版社出版

基于第二、三版内容，但取消了点评版的形式。 添加了“附录三：幕后故事（摘选）”。 添加了“版本历史”。 全书修订，排版做了较大的改动，提高了图片的印刷精度。

该版为纸质版的最终版本。

第五版：2012.12.18，电子版（第二版）

该版为电子版的最终版本。

第六版：2017.05.03，电子版（第三版）

在发布《大道至易（第二版）》时为本书重制了电子版。

内容在本书电子版（第二版）的基础上没有更新。

使用 markdown 重排了全部格式，发布了.epub 和.azw3 版本，并重制了 pdf 版本。