Part Two

第二篇

中程时期

本书为了帮助读者建立清楚的观念，将软件开发过程划分成四个阶段，然而在实际运作上真的很难这样做，各个阶段彼此都有重复的部分。在本篇中，我不设章节段落，因为这一部分的内容用各项法则来说明比较清楚。因此请不要以为我在每个阶段所用的法则仅适用于该阶段。事实上，我所提出的法则都是观念，适用范围是整个的软件开发过程，好的开发团队应该随时随地注意每一个法则是否被实践。而在起始阶段所提的各项法则，如果你有做到的话，你会发现在孕育阶段更容易实践它们，因为你已经练习过这些方法。

在起始阶段的主要目的是让团队凝聚并建立对团体与产品的目标，而蕴育阶段则是充满对目标的期望、不确定和挣扎，所有的事情都是一片混沌，并且充满对可能失败的恐惧。这时候，毅力是最重要的。暂且不管你的团队是多成熟，暂且不管你经历过多少次这种场面，或是产品出炉时会有多伟大，孕育阶段总是可怕而乱糟糟的，你随时都可能遭遇意外打击或是想象不到的失败，但绝对要坚持下去。

你很可能会有那种心情：但愿有条路让我逃离这一切，每个项目都会有这种时候。每件事情都无法确定，于是组

员的心情开始恶劣，冷嘲热讽、消极抵抗都来了，几个月下来不见天日的辛勤工作，使得组员好似得了坏血病。这场灾难似乎永无休止，可怕啊！

深呼吸一下，进入孕育阶段吧！

中程时期

法则 27

Be like the doctors
用医生的方法

当病人已经药石罔效时，医生通常会对病情有所保留，避免病人太过悲观或恐惧，并且尽量鼓励病人保持希望，最好能让病人有个期望完成的目标。软件开发在这方面也很像医学，它不是完全能用科学来解决一切。只是很不幸，到目前为止，一般人还不了解软件开发是一门艺术——是必须具有技术专长的艺术。

现在，你得学着用医生的方法。医生绝对不会斩钉截铁地断言什么医疗行为一定会有什么样的结果，反而是以一种自在且充满信心的口吻说：“试试看吧，一切都还没有确定呢。”反观软件项目经理人，常常在还不确定是否可行时，就率尔对团队保证产品的特色、时间等等。更糟的是，医生往往是在整体系统大都健全的情况下，对其中的一部分功能做医疗；而软件开发则常常得面对全新的、从来没有运作过的系统，不确定性更高。

当然，任何情况都是有可能的，即使是再简单的医疗行为，都可能有风险。

另外一件应该向医生学习的事情是，即使是再小再简

单的医疗行为，都带着几分风险，所以医生会说：“当然，任何情况都是有可能的，治愈率再高我都不能跟你说百分之百没问题。”如果这样说都还不能让组员明白任何未来的事都有不确定性、都有风险，那我也只好束手无策。

我们必须记住，面对软件开发的不确定性，总会有方法可以试试看。就像医生会试着让病人了解任何医疗行为都有风险一般，管理者也应该让组员了解，任何项目都有不确定性和风险。

“一切都还没有确定呢。”

法则 28

Remember the triangle: features, resources, times

软件开发金三角：特色、资源和时间

作为一位软件开发的领导人，你得集中注意力在三件事情：资源（人和钱）、特色（产品与其品质）和时间。这三件事是软件开发的核心，其他的都是外围。资源、特色和时间是三角形的三个边，任何一边的变化，都会影响到另外一边或两边。所以如果时间落后了，去看你的三角形，看看对特色和资源的影响；当有人谈到可以增加什么功能特色时，你得立刻谈起时间和资源，以显得你思虑周详反应敏捷。所以，管理者的第一要务是把这个三角形放在心里，随时利用这个模式来思考问题，你会发现答案都在这个三角形内。“好吧，我的时间落后了，我得利用另外两个边来弥补，或是三边都得调整，我可以少一点特色，我也可以加一点资源，或是修改一下时间。”由于人、时间和特色都是你最关心的，所以你对这个三角形有具体的概念，很快地，你就会发现这个三角形对你的思考帮助非常大。

软件开发的金三角

还有一点要记得，时间落后时，你只有四种选择：增加时间、减少特色、增加资源或三者同时进行。但人是不可以随便增加的。

不可加派人手？

你是否听说过，加派人手是个错误的决策？在佛莱德·布鲁克有关软件开发的经典著作《人力资源之钥》（The Mythical Man-Month）中，用了很大的篇幅来阐述他的论点：在软件开发项目中不当地增加人手，反而会使工作进行更慢。布鲁克的观点被证明是非常正确的，但我们应该了解的是，布鲁克是在写一本书，而不是写一部适用的律法。

然而，传统的教育使得人们用太过单纯的理解方式，使得布鲁克真正的睿智多少被曲解——绝对不要在进行到一半的开发团队增加人手，重点是“进行中的团队”。虽然布鲁克十之八九是对的，但却成了太多管理者的借口，理直气壮地决定不加派人手。是的，增加人手是很难管理（人员应该在项目开始前就规划妥当），通常也不是个好办法，只有在非不得已时才考虑，但不是被禁止的。

法则 29

Don't know what you don't know

不懂别装懂

对于不懂的事情，千万别装懂，或是看起来似乎懂，也不要接受别人不懂却装懂。不懂却装懂一定会造成团队管理上的困扰，这一点几乎没有例外，你若犯了这种错误，一定会尝到它的苦果。在每个阶段的每一个人，一定有某些重要的事情是他不知道的，这应该是被允许的。去掩饰你不懂的事情只会造成别人在认知上的偏差，结果反而导致不知道那些事情可以相信，那些不能相信。如果你勇于承认自己不懂的事情，就不会掉进这个泥淖。

人们会觉得对于重要的事情，我如果不知道就很丢脸，这是天性。而在软件开发过程中，太多东西是大家不知道的，因此，管理者或开发人员就很容易有这种不懂装懂的倾向。好的开发团队应该有一张清单，上面列着我们目前不知道的事情，这样才比较容易掌握到底什么事情会不确定。抗拒“我都知道”的骄傲天性，是需要团队士气和心理上的勇气的。对于管理者而言，称许承认不懂的勇敢组员更加困难，尤其是这个事实已经被文饰过的时候。虚假的文饰是面对未知事情的错误防卫心理。

虚假的文饰是面对未知事情的错误防卫心理。

我非常建议管理者重视组员了解自己什么地方不知道，而不是被迫或自愿去掩饰。要让你的团队明白他们从未检视过自己的未知，但要从现在开始练习承认自己不懂的事情；一般来说，在团队成立的初期或转变阶段会比较容易做到。我们没有时间拐弯抹角。最后，大家会明白成功比较重要，牺牲一点点面子或是受到幻灭的打击又有什么关系呢？

你坚持组员必须面对不确定性，承认自己不知道，最后会将无知变成知识。领导者的任务是让大家全都明白不确定性是绝对的事实，必须强迫大家去面对它、适应它。为了大家好，团队应该学会在不确定的环境中生存，并且兴旺起来。

既然要写在书里，就让我们说这是正式的规定：当有一件事是不确定时，就直接说明这个事实，即使不确定的事情是何时能够推出新版。不必担心，没有项目经理会因

为承认自己不知道的事情而被撤换。

知之为知之，不知为不知，是知也

不必说大家都晓得，我的属下会因为知道自己不懂什么而获得我的加分，因为知道自己不懂，才会去求知。我宁愿知道什么不足，这样我才知道什么事情可能会绊倒我，我的同事和属下最好也明白这些。

软件开发项目的目标并不是事前做好正确的规划，而是每天都得在事情从未知到已知的时候，做出正确的抉择。如果你明明不知道某件事却假装知道，你就无法在事情从未知到已知的时候得到正确的信息，也就可能会做出错误的决策。当信息证明你错了，你一定觉得非常难过。于是你会更加害怕信息，而别人就以为你在抗拒事实，最后你将陷入恶性循环。

只有当你知道不确定性在那里时，你才有可能解决它；那些没有被发现的确定事情，会把你绊倒。相形之下，承认你不知道是比被击倒要好得多了。

当你不知道事情该如何完成，当你内心有个声音在质问你、在困惑你——面对它吧！不必害怕承认了自己不懂就显得很笨，你一定会因此而学到某件事情的。在你内心不断萦回的声音，事实上是团队还没浮现的怀疑，而你收到了无形的信号。也许当你告诉别人时，得到的回答是：“对啊，我也在怀疑同样的事情。”当然，

偶尔别人会认为你是不理性的空穴来风，没关系，那是你在不确定的环境中必须付出的小代价（如果命中率实在太低，也许你得看看心理医生，但至少不会令你延误就医吧）。

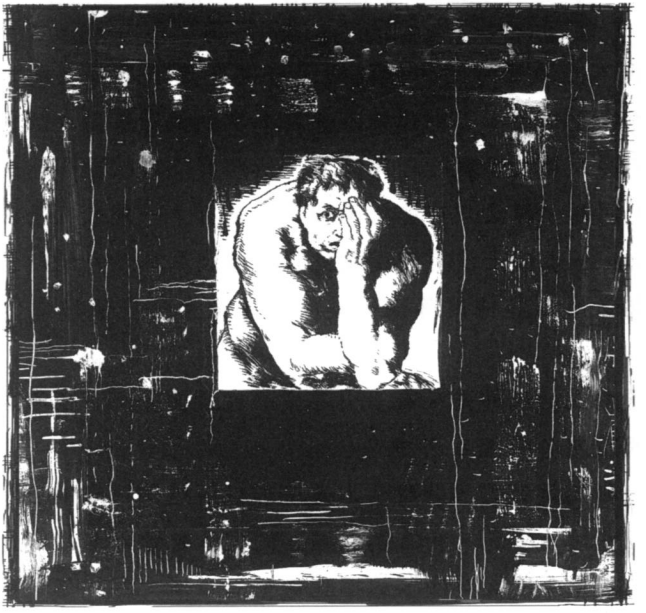
表里一致

表里一致是一种平衡与调和。你是否说一套做一套？或是想一套说一套？或是意念和行为不一致？我们的言谈真的是我们想说的话吗？我们是否让别人牵着鼻子走？我们的行为和我们的感受与意愿是否一致？当我们说要做，是不是要去做呢？我们是真的同意，还是假装一下让别人以为我们同意？

一致性是诚实与伪善的区别所在。当我在做某一件事情的时候，我是因为这件事本身值得做，还是因为它对别人似乎有点价值？我是诚实说明自己的看法，还是比较倾向保护自己？当我认为某件事情是个好主意时，我是否会去做呢？我是真的相信，还是让别人以为我相信？

一致性也是实情之所在。惟一比说出实情更难的是了解实情。当周围所有的人都认定某件事情是对

的，但我实在不能证明，只有直觉地怀疑那是错的，而团体思考模式尚未建立时，我能不能说出我的看法，然后去发掘真相呢？



明人勿处暗室

法则 30

Don't go dark

建立适当的检查点

假设在你接近检查点日期时询问开发人员：“情况如何？跟得上进度吗？”

他回答：“最近六个星期都进行得非常顺利，但是今天的进度呢，是六个月前该做完的事。”

不，落后不是一天造成的，落后不是到了项目快结束时才出现的，也不是到检查点前一天才发生的，它每天都在发生，每小时都会发生。每一次你泡一壶香浓的咖啡，每一次你回个电子邮件、组装一部计算机、抓一只难解的虫，都是落后发生的时候。

为了对付进度落后的问题，你必须把一个大型的开发工作切分成细细的小段，每一小段都是一个检查点，每一个检查点都必须能够看出来有没有延误。检查点的间隔周期应该多长呢？如果太长，检查点的效果不够；如果太短，工作可能无法分割得这么细。软件不像堆积木那样简单，程序之间的关系是立体的、动态的，倒是很像庖丁解牛。在我的小组中，我们为此来来回回争论不休：五天？十天？三周？根据我的经验，三周足以使情况失去控制。

每家公司适合的检查周期并不一定。我们的做法是，让组员和组员之间彼此协议，一方应在期限之内做出该做的东西交给另一方，否则对方无法及时开始，如此就

形成了适当的工作切割点，而且一有延误，我们立刻会知道。比方说，我们知道这个星期的进度落后了一天，一天可不是小事，这是非常必要知道的，这总比进度落后了半年我们才知道要好得多了，万一到了那种落后程度时，恐怕连计算落后多久都来不及了，全世界最糟糕的事情莫过于此。

全世界最糟糕的事情，是你迷失了，迷失在软件开发的金字塔中。在孕育阶段你很可能有这种完全被搞迷糊的经验："我不知道该做什么才对，我不知道现在是离目标更近或更远，我只知道现在的状况糟透了，我所做的每一项措施似乎只有把事情搞砸，我只能确定每一件事情都搞砸了……"这种可怕的感觉是来自纵容黑箱工作——让工作在没有人检查的状况下进行。把你的灯打开吧，让你的组员在透明的环境工作，一天完成一天份量的工作。

迷失在软件中

我不必浪费时间向你说明开发进度落后是多么严重的问题，你完全明白，否则你就不会在本书中找寻答案。随便找个软件开发人员或项目经理问问看，他们有没有经历过进度严重落后的问题，答案只可能

有两种：一是有，一是表面上没有但事实上有。表面上没有但进度延误是因为原本估计的进度表过于宽松，所以能补偿落后的进度，或是大幅降低目标作为代价。

更恼人的问题是，有多少项目经理或开发人员感觉自己迷失在软件开发的金字塔中？

大概凡是软件开发从业人员都能理解这个问题的涵义，迷失是极可怕的感觉。空有满坑满谷的理论帮你计算项目进度的指标，却完全没有办法让你摆脱郁闷，每一个人都觉得闷闷的，你开始失去敏锐的判断力，即使是有点疯狂的档案结构方式也被认真考虑，焦虑、畏惧、烦恼、沮丧，偶尔又觉得挺有希望的。你根本已经不知道自己在干什么，事情做得怎么样，问题到底在那里，你的每一个动作都是大失败，你完全束手无策。

于是，你开始有一种信心溃散的直觉，自己一定是能力不足才会搞成这样。接着，每一个人都会发现你的恐惧，对你的信心也因此而消失；组员已经不信任你，你还想如何领导他们？

这些，正是每个项目经理人的锥心之痛。

迷失在软件中

现在，基于检查点的重要性，你告诉开发人员：“我们要每周做一次进度检查。”有人反映这样的管理太细了。为了让这些人能够真正接受这样的管理方式，你得向他们说明，还有其他的人等着这份程序完成才能开始做事——文件要撰写、品保人员要评核，而且还有其他的开发人员也需要得知本阶段的执行结果，才能撰写或测试相关的程序。团队成员彼此之间的工作时程有上下游和回馈的关系，彼此互相依赖彼此的进度，于是形成了实施检查点的动机，而不是靠管理者的铁腕来强制。

有些产品的功能特色需要长达数个月或数年的时间才能完成，但是时程的掌握是与任务大小无关的，因为任何的进度延误都是一点一滴累积出来。也就是说，工作的切割必须够细，万一其中一小段发生延误，可以在短期内立即赶上。事实上，一个星期的检查周期已经够长了，想想看，如果要在下一个星期内补赶进度，你是否做得到。因此，即使管得太罗嗦，让组员讨厌你，但为了整体进度能够如期完成还是不得不如此；而且如果大家都把时程当作是最重要的，那么大家都会觉得时程表让人有成就感，而比较不介意检查的麻烦。

此外，同事之间在进度上的互相依赖和承诺，也是顺

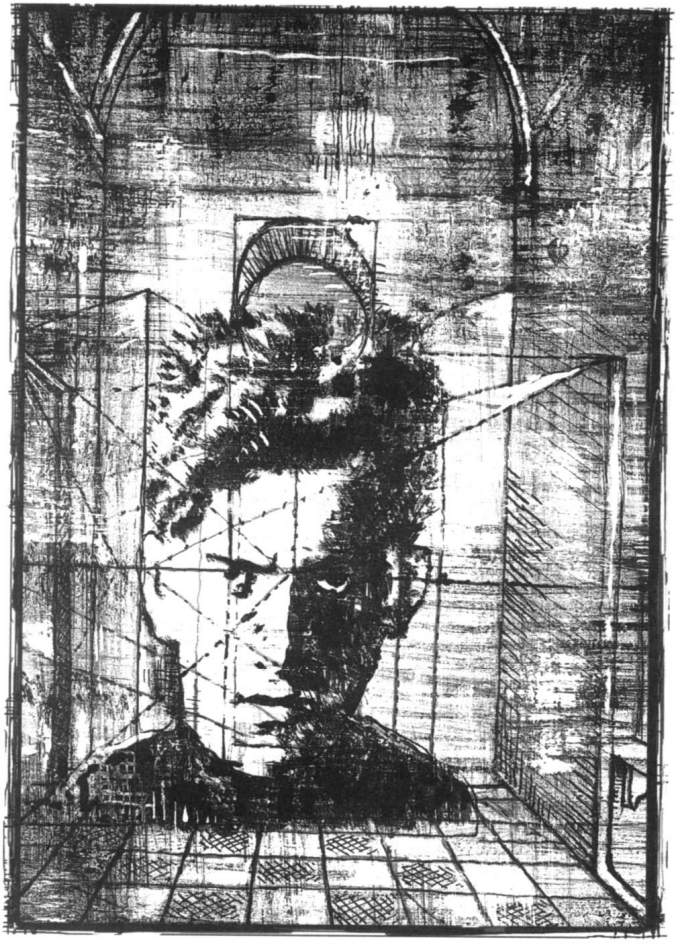
利执行检查点的要素之一。就像产品设计的目标是让每位成员都参与，每个人都贡献出自己最好的想法，检查点的实施也是同样的道理，每个人都实践自己的承诺，这个制度就能成功。比方说，开发人员玛丽答应文件撰稿人乔伊星期五给他写好的程序，于是玛丽个人的信用就成了工作的一部分，这时候玛丽会自动自发地如期完成；另一方面，如果管理者比尔命令开发人员哈瓦德负责的程序必须在周五前完成，那么哈瓦德心中对威权的反感成了工作的一部分，而比尔对自己管理者形象的不安全感，也同样成了工作的一部分。这又是创造智能财产过程中有趣的现象：每个人的心理状态对成果都有直接影响，没有人例外。

我在前文所提的表里一致，对于软件开发的影响就是进度问题，每个人都重视自己的承诺和信用，彼此没有隐瞒，而检查点的实施就会很顺利。对于那成千上万的小段工作，就靠彼此的承诺和信用来维系。这和整体目标的大小没有关系，倒和组员是否表里一致、勇于面对不确定性很有关系。

法则 31

Beware of a guy in a room

留心没有检查点的组员



留心没有检查点的组员

法则31是法则30的特例，但有必要独立说明。

曾经有一次（其实是三次，但我实在惭愧得不敢把三个故事都说出来），我把最困难的工作分配给团队里最最优秀的人才威廉，大家都知道他是最有经验、最有创造力、技术能力最强的人，我们拥有他是上帝的恩宠。没有人怀疑他的天份与判断力。

当威廉开始他最具挑战性的工作时，他回房关上门，此时陪伴他的只有水族箱、重金属音乐与莫扎特的旋律、泡泡糖和星际争霸战的海报，以及6箱果汁和可乐，一切就绪后，他开始工作了。其他的组员都对他充满信心，非常庆幸我们拥有威廉这样的天才可以扛下这么艰巨的任务。在门外我们听到他快速敲击键盘的声音日夜不停，心中在微笑，虽然没有人知道在关上的门内是多么伟大的程序，但我们都觉得有高手威廉辛勤地为公司的命运打拼，真是大家的福气。

他的前额出现了斗大的汗珠，显然压力很大。

（而这是前所未有的乐观情况，我们的目标很清楚，我们的安全控管做得很好，我们的团队是最优秀的人所组成，虽然我们偶尔来杯即溶咖啡，但我们都深信自己在做一项伟大的计划，别人的批评我们充耳不闻。）

渐渐地，我们开始看出威廉遇到了困难。他渐渐不回家了，鱼也不喂了，音乐也停止了，斗大的汗珠出现在他的额头，泡泡糖愈吹愈小，威廉显然是……进度落后了。

整个工作只有威廉的部分没有检查点，他只要在完成时交出作品就行了，当其他的人都已经如期完成份内的工作，我们不得不敲敲他的门：“威廉，做得怎样了？”里面是一阵迟疑，然后听到他说：“还不错。”“什么时候可以完成呢？”经过更长的停顿，我们几乎听到泪水流下的声音，威廉哽咽着回答：“就快了。”

我们知道他说的“快”，其实还早得很。现在全公司的人都只能呆呆站着干等，一点也帮不上忙。而我，身为主管和项目负责人，面临了困难的抉择：我可以开除威廉，但考虑到他是惟一能够解救我们的人，惟一真正了解当前困难之所在的人，所以我不加思索地决定，让威廉继续完成他的工作，他毕竟是团对中最优秀的人才，也是团队中惟一能够掌握这个问题里里外外的人。

我惟一能做的，只有替他买更多的可乐。

另一个比较好的方法是给他加点压力，让他搞清楚自己身系团队的危急存亡，所有组员和他们的家庭都指望他不凡的脑袋和飞快的手指，我们得催逼他做出个像样的东西出来。

但我稍微三思，就觉得这也不是个好办法，对生产力没有帮助。我没有办法摆脱这个困境了，我只能往好的方面想：威廉会完成他的工作的，只要他没死或离职。经过这次教训，他大概再也不敢为我工作了。现在我惟一能做的事只有帮他买更多的可乐，其他什么办法都没有！

当然，如果你有一位大天才关起房门埋头苦干，你或许会有更具独特创意的结果，一种是健康型，一种是病态型。病态型的闭关修炼是大天才无法在任何一关做出成品，却能在期限的最后五分钟交出奇迹似的杰作；健康型的闭关修炼是大天才免除一切世俗的干扰，全心致力于工作，由于他的工作创意程度太高，完全得靠他的心理、情绪等

内在的心智力量，这时候团队合作对他来说没有什么帮助，反而有打扰之虞。

像这种天才型的特殊程序设计师，为了清静而把自己关在房间里很长的时间，而没有人知道他的工作进展，也许是有发挥创造力的必要，但这对于如期推出产品而言是冒着极大的风险，无论他多么优秀，领导者都不该把重要的工作交托给他，除非让他与其他的开发人员一起接受定期检查工作进度，而且对时限绝不宽贷。可惜有些天赋才智恣意奔驰的人无法忍受这样的限制。在软件业中，这种人确实存在，他可以做一些无法想象的创举（在实验室里?），掀起一波技术的飞跃，但他绝对不会出现在矢志如期推出产品的开发团队中。

不论是健康的或病态的闭关修炼者，允许一位开发人员关在房间里没有人知道他的工作进度，结果通常会是开发团队的致命伤。所以，一定得竭尽所能地避免这种情况，绝对不要允许“没有检查点的组员”，否则你难逃失败。

法则 32

If you build it, it will ship

软件要经常建构，就能顺利推出

(软件通常是由许多许多程序组成的；程序写好后，经过编译器产生执行码，这个动作叫 compile，通常如果是小而独立的程序，可选直接选择编译成可执行档，但是以软件包或C++之类的大型软件而言，程序数目都在数百个以上，通常每个程序都是编译成对象文件，即object code，然后众多不同的程序依一定的结构关系组合成一个软件，这个组合的动作叫作建构，build，产生的结果是真正可执行的完整软件。每一套软件的建构方式多多少少都有些不同，原则上，被呼叫的程序或函数库要先build，主程序最后build。——译者注)

将程序建构成软件才能推出，这个自然，总不能卖给顾客一个不可执行的原始程序代码吧！但我的意思是要能够经常且定期地建构软件，并不是到了期限前一天才建构那么一次。你必须随时都让整个软件的现状都能被大家看见才行。

(在很早以前曾经有过一个案例，各个程序都按计划中的时间表进行撰写，个别程序的测试也完全正确，但是到了最后却怎样也无法组合，各个程序就是无法搭配。因为各个程序之间都有相互传递资料或先

后连结的关系，或是其中有一个程序内有个无伤大雅的小瑕疵未被注意，但却在别的程序上造成大问题。在写程序的过程中随时要注意我的程序能否跟其他所有的程序相互配合，最好的办法就是常常 build，这样才能做完整的测试。完整测试是 build最基本的目的，所以作者就不刻意强调，但定期建构软件还有更重要的益处。——译者注）

你能想象几百位工匠蒙着眼睛盖一栋大楼吗？不知道自己在盖大楼的那一部分，不知道这栋大楼现在盖到第几层，只顾盖自己的部分，这种大楼怎会不垮呢？因此，经常建构软件是非常必要的，这个法则的重点不仅是要在整个开发过程中经常地、定期地建构软件，而且要尽可能建构出现阶段最完整又最正确的软件，更重要的是建构出来的结果要放在公开的地方，让每个人都能看到。

软件也许不必每天都做建构，但是一定要经常且定期做，而且不只是程序要做建构，安装程序和线上求助的部分也要包括在内，然后将建构出来的结果放在公开的地方，让品保人员可以评估每天的软件状况，也可以观察出它发展的情形，或是陷于停滞。规律建构软件是一项最可靠的指标，表示团队的运作是否正常，软件是否能够完成。

丑媳妇也要经常见公婆

软件的建构需要时间，程序也很复杂，值得你为它仔细研拟最适当的策略。在微软内部有很多开发团队采取每日建构的策略，并有专门的小组（build master）负责这项工作。建构程序是很重要的，必须确定软件能够建构得起

来，没有失败或中断。在每一只程序置入（check-in）开发环境时，会产生这一只程序需要的建构程序（通常是配合适当选项的compile），然后整合在整体的建构程序中。于是其中任何一段建构程序发生问题的话，很容易找到由谁来负责，所以组员的责任心和荣誉感会促使自己一定要做出至少能够合格的程序，而每天的软件建构就会非常顺利。当然你可以做得更严格，以我的经验，一天建构一次是最有效率的方式（请参考下一个法则）。

（有些软件公司每次的建构都是把所有的程序都compile一遍，然后build起来，有些则是将修改过的程序compile，再build上已有的软件。由于全部build一次所需的时间可能太长，所以大部分的公司都采取第二种方式。说得更口语一些，就是把其中的一块挖下来改一改再嵌回去。当这段程序被认为 OK了，就做置入的动作，check-in；每隔一定的时间，就有专人把这些被改过的程序compile和build。——译者注）

另一方面，项目经理应该是拟定最佳化的建构策略，而不是绝对性的每隔几天做一次建构。不一定要每天做，应该是采取对软件最适合又最小的间隔，重点是在能够建构出软件的前题下，用最短的时间间隔建构出软件。

老实说，我并不知道软件“应该”采取多长的建构周期最好，但我非常相信大部分的软件公司在这方面都做得不够。对某些开发团队而言，每天建构一次也许稍多，但我很确定，对任何开发团队而言，每周建构一次绝对是最起码的。

项目经理一定要时常看见软件的整体状况，才能明了现在正在做什么。即使你的工作分配和检查点设置得非常好，各个组员之间的信任与承诺关系也非常理想，但若是无法建构软件，则你对进度的掌握将仅限于想象和猜测，而不是根据事实，所以你还是不能掌握软件开发的真正状况。

每一次你建构软件，你就能算出错虫数目有多少，你就能掌握软件开发的进度，你就能看见功能日益成形。

再者，为了让软件能够建构出来，程序必须维持在一定的品质水准，太夸张的错误要立刻被发现，而且不可能将软件倒回去重来。因此，每隔固定时间建构软件，

才能够维持软件有一定的秩序和品质，这一点要让大家都知道。

更重要的是，定期建构软件就像是团队的心跳一样，每天五点（比方说），大家都看到今天整个团队的工作成果，每天都看得到今天的进步，这对团队士气有很大的鼓舞。如果有一天软件建构失败，或是为了某种原因而不做建构，马上就会引起大家的注意和紧张，一定要找出原因来改正。如果建构经常失败，这就是整个项目失败的前兆。

但请不要搞混了，我所谓的软件建构，不是程序设计师在自己的PC上做的小规模的建构，而是指正式的、公开的建构，将大大小小的程序结合成软件，而且是每个人都能看见、能执行的软件。品保人员可以对它做初步的测试（sniff test），看看它的基本功能大致上有没有错误。如果你每天都做软件建构，你就很有把握它至少不会当得惨兮兮，而且大家可以看见它朝着目标一天天地迈进，功能特色一天天地成形；而且每个人都测试着相同的软件，对于软件的现状也就会有相同的认知。

经常建构软件还有其他的好处：

• 经常而公开地建构软件可以看出组员彼此之间真正

的信赖程度。只要有任何一个环节没有密合，软件就建构不起来，检视建构的过程就像是检视组员之间或是团队对外的关系，在任何地方发生问题都很容易找到。

- 建构出来的软件可以显露出在设计时没有考虑到的问题，例如执行效能、对象大小等等，这种问题万一发现得太晚就来不及修改了。
- 软件的建构会很自然地让组员的脚步一致。一般而言，建构软件最常见的问题是版本协调，最严重的是各人有各人的版本，谁都不知道别人在做什么版本，有了公开建构的软件，大家的版本就可以同步了。
- 建构软件可以促使组员面对他想忽略的问题。团队恒等于软件，所以目前的软件状态就是目前的团队状态。当我遇到产品有问题时，我追问组员：“我们软件建构的情况如何？”答案很多种，意思只有一个：“我们花很多时间才能成功建构一次，有时候长达两个星期，我们希望时间能够缩短，但是因为某些某些原因，我们没有办法照计划来建构。”

是的，要频繁而规律地建构软件当然会有很多困难，

但是解决这些困难却能够带来健康的团队和顺利的开发。开发人员必须得自律，自己仔细检查所有的功能都完善了才把程序置入，程序不仅要在逻辑上执行正确，还得注意体裁（程序段落清楚）和资源运用（内存用完了要释回），以及执行效能，如果个别的程序组件都很健全，就比较容易建构软件，并且能够大大减少回溯检查的时间消耗。

在开发过程中，很容易对软件的状况产生错觉，但是如果你每天都建构一次软件，你看到的将会是具体的事实。

法则 33

Get a known state and stay there

掌握实际情况

这又是一个很简单又很有用的道理，引伸的意义是“每天都要有可以推出的产品”。

你必须对软件的实际状况知道得非常清楚，特别是要推出的时候，你必须知道它是什么模样，架构、特色、效能特性等等。在你将它当成产品推出时，你必须知道所有能够得知的情况。

如果你询问开发人员软件的状况，
他可能答对，但那是碰巧。

现在，假设有人要求你对刚推出的产品加进某一个特色，你能够立刻掌握这个特色对软件的一切意义，那你就会胸有成竹，因为你知道该怎么做才能加入这个特色，会不会对现有架构产生重大的影响或冲突等等。你可以召集一些开发人员和品保人员、文件人员，讨论一下就可以确定大家对这项特色的了解，大家对自己的角色和任务范围都非常清楚，然后你就可以说：“来吧，开始行动！”

也许几天以后，也许一星期左右，那几位组员手上拿着磁盘片来找你，告诉你新的特色做好了，只要安装这个

就可以把新的特色加入软件。并不困难，是不是？

这是因为你能够充分掌握情况的缘故。你要知道，推出一个新版本就好像无数次不停地加上新的特色，然后推出。这是最能让你理解组织软件开发活动的方式：将一群各种角色的人组织成一组，负责一项功能特色；关键就在让软件保持在你能充分掌握，又能够推出的状态。千万不要让软件变成一堆凌乱的单元，千万不要让软件成为你不清楚的状况，千万要抓紧对它的掌握，不要放松。

想要掌握软件的状况，一定要有品保人员帮助你，你需要有人专职负责检查软件的状况。如果你去询问开发人员，他的答案也许是对的，但那只不过是碰巧猜对。开发人员虽然是直接创造软件的人，但他们却无法知道软件全面性的真实面貌，这还是得靠品保人员来评估衡量，让专门的第三者来向你报告软件的状况。

而且你每天都得亲自做点小的测试，一部分是自动的，一部分是手动的；每周或每两周要把产品整个测过一遍，确保你的软件随时都在可以推出的状态。

什么叫做“掌握软件的状况”呢？就是在某一特定的时间点，对于每一个产品组件的一切状态都有精确的信息。因为有品保人员将各个组件都测试过，所以你能确知信息

是正确的。在你手上必须有一张清单，上面清楚列出各个测试通过和没通过的功能、错虫数目、错虫发现率和清除率，和一些其他重要的数据。

我们要注重数据管理，像变动率（churn）就是很好的数据，你可以看出有多少的程序代码增减，当然还有许多其他的数据可以参考，都挺有用的。但重点是，你决定要看那几项数据，就得每天去测量它们。

预定的目标能够确实完成，就可以推出。

然而，并不是由品保人员去评估决定软件是否可以推出。由于开发是团队全体的责任，软件是否可以推出在每个人的心里都有数，这应该是不会有争议的——原本设定的目标能够确实完成，就可以推出。当你终于在每一项预定功能上面打勾表示完成时，你会拥有那种真正的成就感（还有你的报酬），难怪你愿意为这个勾勾去努力。

但先决条件是你能掌握软件的状况，而且维持有关软件的最新信息。你必须善用你的品保人员来做到这一点。

时时刻刻对软件的状况清楚掌握其实是相当困难的，你必须有一群非常优秀的品保人员。很多软件公司只有很少或没有真正的品保人员，所以永远无法掌握软件的状况，事实上，一个现代化的软件开发团队是不能没有品保的。

记录里程碑

这一段主要是对法则33的补充说明。我在此特别要针对这些软件开发的观念作详尽的说明，这些都是我多年来不断摸索、思考、创见、旁征博引以及求来的，是我与多位极聪明的同事，犯过不少大错所换取到的。

你想在软件产业中占有一席之地吗？你想有点不凡的进步，不是吗？所以你必须做个“里程碑”（milestone）的记号，表示你现在到达了某个目标，但你还要前进到下一个里程碑。

里程碑一定要有一种衡量的标准，否则很难达到，所以不要设定一个无法精确定义的目标。你的每一个目标，不论大小，都应该有个专属的档案来做记录，你的资源投入一定不能模糊随便。

法则 34

Use ZD milestones
零缺点里程碑

在发展中的软件本质上是无形的、捉摸不定的。一部分存在于开发人员的脑袋，一部分是在媒体里边看不见的字节，一部分以纸张的形式散见于各个计划中的文字，更有一部分是完全不知道的潜意识，随时都在变化，要到适当的时机才会被激发出来。而团体的潜意识，不论是创造性或是病态性，在无形中都表现在组员的日常活动和每天所做的决定中。正如前文所述，软件恒等于团队，所以团队的一切状况都表现在软件之中，软件就是团队活生生的投影。

零缺点不代表软件中没有错虫，也不表示没有遗漏的功能。

如果想把软件的开发活动管理得当，就得让软件定期“整装待发”，这时候大家对软件所付出一切有形无形的努力就可见分晓，同时也可以分析那看不见的潜意识究竟是创造性或病态性。当以各种形式分散在各处的软件组件建构成一整体之后，你可以得到完整的软件状况信息，看看下回应该改进什么。然而，即便是经常建构

软件的好处那么多，这件事背后所需的一切——包括产品的设计、发展程序和团队内心潜藏的各种动力，实在是件复杂无比的工作。

软件推出就是最后一个里程碑。

暂且不管多么复杂，为了让开发工作能够顺利进行，你至少要让组员把手上的工作划出一个成形的句点，并且要有勇气面对一切麻烦的问题，同时也让组员加强对软件现状的了解，让组员更有信心、更有能力预测自己付出什么程度的努力会有什么样的结果。一开始大家都会很痛苦，但慢慢能够培养出这些能力，对这些场面应付自如，开发工作也会顺利起来（请参看法则3）。

很多微软的开发团队使用“零缺点里程碑”（Zero Defects milestone）的方式来了解软件现状，简称“ZD”里程碑。所谓零缺点，并不代表软件中没有错虫，也不表示没有遗漏的功能，而是指团队的成品达到了事先规划的品质水准，也经过测试验证，就是零缺点里程碑。在Visual C++的团队中，我们每一次推出产品通常会预先规

划三到四个里程碑，一个产品周期正常是一年左右。

当然，最后一个、也是最高的里程碑就是软件推出，但是其他的里程碑绝对是同等重要。

我们在法则33中倡导，软件应该随时保持可以推出的状态，而且每天如此、永远如此，这是个理想，但事实上即使没有进度落后的现象，有时候也会做不到这一点，因为你偶尔会需要倒回头一些，去处理前面的问题（regressive things），像是将前面为了换新功能而暂时替代的东西要移除掉（好像在道路修筑过程中要另辟一条暂时的便道才能维持畅通，确定修好了之后便道就得拆除）。而零缺点里程碑，就像是经过较长的时间后的某一点，要确保软件的状态必须达到预期的品质水准，好似定期大扫除或大检修一样，不要把小缺点久留。

一个里程碑大约是六星期到两个月，当这个日期到了，除非里程目标已经达成，所有的开发活动都不能跨越这个里程碑往下走。因此，里程目标常常被称为“中间产品”（deliverables），而且是大家都能看得到的。达成里程目标之后，新的里程随即开始。

这个原则说来容易做来难，如果你有充分授权的开发团队，团队中的各个成员可以共同确定里程目标达成，管

理者没什么事要操心。每一次的里程目标都由品保人员事先设计好验证的准则，事先沟通每一个通过验证的品质水准细节，而且大家都同意、有共识。也就是说，这个中间产品的品质水准是由组员协商出来的，而不是管理者命令出来的。每一个中间产品都有事先定义好的验证准则，注明每一项准则应该在什么时候通过，这许许多多的验证准则会由项目经理集合起来，就是里程碑。

采用“零缺点里程碑”最大的好处是，每当有进度延误时，能够立即发现并在最短的时间内补上，也可以确保问题能够防微杜渐，及时处理。如果你每一两个月就有一次里程碑，你就得把进度延误控制在这个里程碑之内。你在一个里程碑内所发生的进度延误总比整个项目少，也比较容易赶上，这比到了最后推出时才发现要好得多了。每一次里程碑都确实达到，也就给你一个确定的信息；若是预定的里程碑日期到了，软件却又再过了n周才达到应有的品质水准，你至少有个立场来催促团队加把劲儿：“我不知道我们最后的推出日期会延误多久，但这次的里程碑已经确定我们落后了n周。”

法则 35

Nobody reaches the ZD mile-stone until everybody does

所有组员一起到达零缺点里程碑

如果有一个工作小组遇到的困难比较多，以致前进的速度比较慢，而其他的小组已经完成份内的工作时，最好让其他人支持比较慢的这一组；团队工作应该平均分配，其中有一组做得特别快或慢都不是很好的现象，毕竟，团队工作是全部都做完才能算完成，只要有一部分尚未完成就等于失败。

有一种很不好的情况是（我真的遇到过这种情况）“落井下石”：比较快的小组发现比较慢的小组在开发关键性技术时遇到困难，可能会使整个里程碑延后时，反而主张在这一部分加入更多的特色。避免这种情况的办法是，领导者必须让整个团队共同担负达到里程碑的责任，除非每一人都达到了里程碑，否则就等于没有人达到里程碑。

法则 36

Every milestone deserves a no-blame postmortem

完成每个里程碑后，心平气和地检讨

每完成一个里程碑后，紧接着应该做一次检讨，这并不需要花很多的时间，却能让团队下一次更好。当我们认为做到了百分之百，是不是真的没有任何遗漏？好的检讨绝不是指责任何人拖累进度。项目经理可以召集一个会议，每一个特色小组中至少有一人参加，或是请大家有任何心得都不要客气，发个电子邮件告诉项目经理，检讨会最后的结果再由项目经理发电子邮件给每一个人分享。尤其是做得特别好的事情，应该在检讨会中强调，大家以后就更知道怎么做会最好。在每个里程碑之后立即做一次检讨，是一个群体学习的最好方法。

在里程碑之后责难某人或某小组拖累大家的进度，是愚蠢的自我伤害。讨论进度延误的目的仅限于：研究下次如何避免，加强大家对延误的警觉心，以及万一延误时最好的补救方法。进度延误的不利后果发生在属于大家的软件事业，不是管理者该去判断或核准的。延误势必增加公司的经营成本，团队成员不会天真到以为自己不受影响，所以管理者不必刻意惩罚，只要让团队明白延误对大家的伤害，下次就会特别小心了。

责难任何人拖累大家的进度是愚蠢的行为，就好像责怪叶子不可以从树上掉下来一样没有意义。

对于个别组员而言，延误进度是不好，但也具有某些教育意义。一般来说，只要是健康的团队，不论成员的素质高低或是延误的情形严重与否，发生延误的组员都会觉得不好意思，而主动试着解决它，他会尝试修正对自己的期望，改善自己的工作效率或工作方法。管理者应该帮助他，责难他是愚蠢的行为，就好像责怪某一片叶子怎么可以从树上掉下来一样没有意义。

如果你在每一次的里程碑之后都做了足够的反省，而且你的组员都吸收了里程碑给他们的教育，你的团队一定会逐渐成熟，最后每一次的里程碑都会准确无误地达成。

法则 37

Stick to both the letter and the spirit of the milestones

把握里程碑的实质意义与精神

里程碑的做法在成本上是比较高的，而且团队得承受比较大的痛苦，但这是惟一能够确保开发成功的方式。团队成员为了协调出里程碑的衡量准则，所付出的时间精力就是里程碑的额外成本；为了能具体标示出里程碑不得不修改特色，团队的理想被打了点折扣，内心多少有点失望心痛。而更多的痛苦是来自于设法使大家都专注在里程碑上，协调大家对里程碑产生一致的价值观，为了里程碑愿意付出额外的时间和努力，即使真正的推出时间还早得很，我们却要劳师动众地练习、经历这个大家都辛苦的中间产品“假推出”。

然而“零缺点里程碑”所带来的好处远超过这一切所付出的代价，每个人所获得的经验和成长也远超过这些痛苦。实施零缺点里程碑可以让软件评核的过程在开发初期就开始，藉此可以带来团队的共识；对事实的了解和包容成了最高的团队价值，因为有太多人对人、群组对群组的信赖和承诺，不是实践就是食言。里程碑内容的定义，就是大家对自己工作的期望。从某个意义来说，里程碑就代表了团队承诺的信赖度。

经过项目初期的几次里程碑之后，在团队中已经培养出对里程碑的使命感，能够准确地判断自己可以实现什么

承诺，在这样的时间限制之下，那些事情可以完成，那些不能。我曾经经历过几次“字面上的里程碑”，团队把里程碑当作形式和教条，而不去实践里程碑的实质意义和精神，这是很糟糕的。以我的经验，凡是健康的团队，都会对“一致性”（请参考法则29）有很强烈的感受力，会极力避免任何欺骗自己或逃避现实的事情，完全自发地希望一切都是真实的，而不是被迫遵守任何传统的教条或规定。健康的团队喜欢里程碑和组员之间有一致性，不要形式化的里程碑，或是过度严格而做不到的里程碑。说实话这是一个成功开发团队的重要品德，我曾经看过组员之间互相挑战对方，是不是在欺骗自己承诺做不到的事；在健康的团队中，组员能够“嗅”得出来彼此是否在说大话。

我曾经听过组员之间彼此挑战，是否在初期的里程碑目标上太过高估自己。

在项目初期的里程碑，把要求放低一点是可以接受的，比较困难的特色不要求在这个里程碑中完成、品质水准不

必那么高、资源可以多用一些，这些都不要紧，可以当作团队成长的代价。反而是忽视已经发生的问题而不予补救，或是对于团队成长的问题自欺欺人，才是致命的大错误。只有在项目初期不断自我检验，团队才能成长，在往后的里程碑中才能学会掌握软件成功的诀窍。

法则 38

Get a handle on "normal"

培养正常的团队运作

在整个发过程中，前前后后的里程碑应该如何定义才适当？有没有一种原则或特征来判断在每一个里程碑中的团队行为是正常或病态？如何看出团队的领导有没有问题？无论如何，一定要先知道什么是正常的团队运作，才能做出诊断、治疗和预防。

诊　断

在软件开发过程中最困难的活动之一便是诊断——准确判断出团队行为和产品状态在不同时间所发生的一切问题。因为创造软件的过程本来就是动态的，充满合宜与冲突、愤怒与喜悦，团队的行为是变幻莫测，基本上不能以组员的心情愉快与否、日程落后与否、或是表面上的行为与事件来判断整个团队行为是否正常。这种变幻莫测的本质，也使得领导人很难预测治疗行动的结果。

曾经有一次，我参加的一个团队因为时间表定得太过自信，而且工作目标也定得高到几乎接近完成，结果眼看着就要延误第一个里程碑（以下简称 M1），此时大家都很紧张，幸而我们还算是个够健康的团队，我们也知道前面的成功不代表后面一定成功，我们愿

意虚心检讨。

诊断有无繁杂

在M1阶段，这个团队深受目标过度膨胀所苦。在M1阶段的开发工作，使我们发现到有一些关键性的功能特色，特别是典范性的功能特色（请参考法则3），显然在设计上有不够周详的地方，没有办法充分支持我们所要传达的信息，这样的设计事实上无法达到我们要改变使用者习惯的目的。于是我们激烈地争论、辩证、思考，几乎使团队分裂。最后我们终于得出结论，一个震憾性的超强新设计方案，大家都会为它而振奋——但也大大增加了全体团队的工作负荷。

原本是改良目标，对软件有益的行动，却造成了一个不良的副作用：只剩下三个星期就到 M1了，组员们还不能确定要不要开发新设计的特色？在旧设计中是不是有些特色要取消？组员们搞不清楚这些特色的优先级，如果要采用新的设计方案，由谁来做、何时做等等问题。旧设计也许不理想，但毕竟是我们好不容易才凝聚起来的共识，就这样被打破再重新设立目标吗？管理者在这样的情况下是否能坚持授权共决的原则（特别是这些功能特色原本是他们主张的），

而且不认为我们是在找麻烦？像这类不确定的问题多得列不完。不难想象，这些问题使我们距离 M1的目标愈来愈远。

互相纠结的功能特色

就算没有工作量的增加，这种提高目标的事情本身就极具争议性。在我们讨论新的设计方案时，显然没有想到它会带来这么多的问题，所以不知不觉地增加了许多特色，这等于是破坏了我们对团队的承诺。不错，以团队的纪律来说我们可能很糟，但是我们确实对新设计方案中的每一项特色都做过彻底的分析，研究它对每个人或每件事的影响，并且取得了新的共识，大家都同意这些新增的特色每一个都很重要，没有包括这些的设计将造成产品的缺陷。我们的结论是牺牲掉其他比较次要的特色，重新配置我们的人力，尽量减少对进度的不利影响。

现在，我们面临了翻案的后果：在决定采用新设计的争论之后，我们回归到开发计划的现实，我们的M1订错了，而且时间上显然岌岌可危。我们集合公司内所有的项目经理和品保人员，对现状做一个彻底的评估，这次讨论的结果，可以说明在软件开发中的一切事情都无法预知，但却是可以被管理得宜的——我喜欢这一点。

争论不休

有一些人认为，我们其实可以稍微放宽一些 M1

的期望，依照旧设计做出M1的目标，在不影响原定目标的前提之下，少做几个特色。毕竟这是第一个里程碑，如果在实质意义上无法达到，至少要在形式上做到。

第二派意见认为，有无达到M1太难定论，一开始目标就定得不够明确，那么即使M1目标的字面意义也会有不同的解释。虽然基本上我们同意M1是应该达成，但实质上我们却用新的设计且违背了原意。

第三派意见认为，我们是因为赶不上M1的时间而狗急跳墙，他们认为这是我们不当地允许在M1目标中出现多余的功能，现在则企图掩饰这些成本。我们在鱼目混珠，模糊焦点，我们在欺骗自己，自以为有团队纪律，自以为表里一致，事实上我们缺乏控制功能特色的能力，而不敢承认无法达成M1的事实。

最后，终于有人指出，我们根本不是在“繁杂”(featuritis)，繁杂是不知不觉、毫无目标地乱加特色，而且是小型特色，但是这些特色对团队共识和产品目标没有关连。在我们新的设计中所增加的特色，本来就是产品该有的，只是很不巧，这些特色都很小，以致在原本的设计中被忽略了，但是这些小特色加起来

却对产品有重大的影响，而如今在 M1的开发阶段发现了它们，以致不得不宣告 M1的延误。然而如果此时没有发现这个问题，最后产品还是会因此而迟延。

就像疾病特别容易侵袭原本不健康的身体，“繁杂”特别会伤害原本就不太健康的团队——也许是市场中的落后者，或是对自身没有清楚认知的团队。在健康的团队中，对于进度的落后自然会产生一种不安，但只有强烈的热忱和信念才会使团队相信这些新增特色的重要性，而且困难终究会被克服。另一方面，太多的不确定性和工作负荷，会削弱团队的免疫系统，开始信心动摇，而不敢肯定这些新增的特色到底是不是产品不可或缺的重心，还是又一个败笔。而此时若是市场或顾客也产生变化，使得团队疯狂地大量新增特色或是重新定义产品，最后必导致失败。

我们在M1所决定的新增特色，都是经过彻底的分析，而且大家都有共识，愿意为了这个理想而承受增加的工作负荷，所有的重要工作同仁都同意我们为了这些特色，势必将进度延后。然而如果没有这些特色，我们就无法改变使用者的习惯，也就失去典范性功能特色的意义。为此我们的进度和人员

都得重新调整，但所有的人都同意，发展这些特色是正确的决定。

M1的本质

上述的结论对M1是有特殊含义的，我们把这种经验称为“M1的本质”（M1-ness），以便与一般的病态性团队行为有所区别。我们不认为这是“繁杂”（featuritis），因为我们确实地、彻底地、不遗余力地分析过这些新增特色，这是与繁杂最大的差别。此外，这次的设计翻案，确实是对原设计的缺失有重要的改正，而这方面的贡献远超过纪律不良的负面影响。

基本上我们要了解，M1正好是把这类问题理清的时机，M1可以让你发现原来设计时忽略或模糊的地方、进度表上的弱点，以及再次凝聚团队共识，对目标会有更清楚的概念。从这次有点惨烈的经验中，我的结论是：从此以后大家对M1都特别注意，修订M1的目标必须有团队共识，而且确实可以达成。

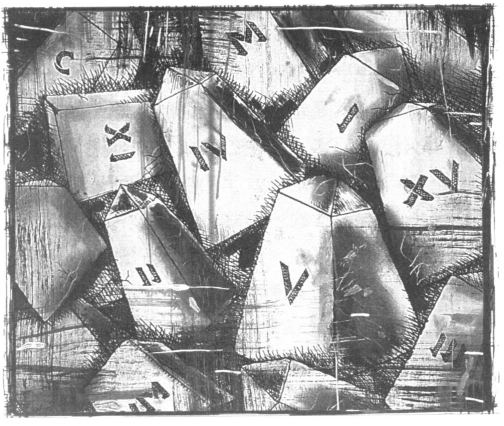
法则 39

A handful of milestones is a handful

里程碑不宜太多，才好掌握

有很多人主张，六星期到三个月是里程碑之间比较理想的时间间隔（interval）。如果你的团队比较小（譬如十到二十人左右），或是目标较小，就可以用比较多的里程碑，因为里程碑对小型团队的额外负担（overhead）会比较少。而小型团队如果经历过比较多的里程碑，会迅速成长和凝聚共同文化（请参考法则7），这是附带的好处。

但是以我的看法，任何团队的里程碑数目如果超过 7 个，就过多了，至少是个不寻常的现象。因为在一般情况下，任何人都无法预测那么远的未来。我的经验是三到四个里程碑，再加上要推出的那一个，应该是最理想的数目（请参考法则18）。



太多的里程碑反而难以掌握

法则 40

Every little milestone has a meaning (story) all its own

每一个里程碑应有专属的宗旨

每一个里程碑，无论大小，都应该有它成立的理由。记得我们刚刚在谈论M1的本质时，提过团队内部发生的激烈争辩吧，我们希望达成M1实质上的精神，不亚于我们希望达成M1形式上的目标。里程碑的宗旨就是设立它的理由，可以被简短的一两句话表达，同时也能描述达到这个里程碑对团队的意义。就理论上来说，任何对里程碑的争论都可以回归到它的宗旨而获得解决。里程碑的宗旨可以清楚地表达它的目标，我故意不称它为里程碑的目标，因为里程碑的宗旨应该包括了团队与它的互动、团队的士气，以及里程碑所隐含的信息，也就是包含了它全面的意义。如果单从M1的目标来看，大部分的人都无法产生去达成它的动机，或是联想到它所代表的报酬， M1的目标是一大串的菜单列，是我们要去做的工作（虽然目标也可能是错的，谁教软件这么不确定呢）。一张清单不能引起人们的热情、专注，以做出伟大的软件为理想。人们只有为了有意义的事情才会愿意牺牲奉献，只有里程碑的宗旨才能让人们愿意付出自己的心灵、精力去换取某种结果。

在Visual C++的开发过程中，有几个不错的里程碑宗旨，值得参考。在典型的第一次里程碑，我们通常会把目标放在“狗食理论”（dogfooding），意思是要使用开发中

的产品，才知道应该如何进一步开发(Dogfooding is using the product under development to further the development effort.)。这是源自行销学的理论，原创者是史提夫·巴摩(teve Ballmer)，由微软将之发扬光大，他的名言是：“狗食工厂应该试吃自己生产的狗食。只有亲身试用自己的产品，才会真正了解自己的产品。”对于像Visual C++这样的程序开发工具，这个概念尤其重要，因为它就是用自己早期的开发工具写出来的。对于我们团队而言，彻底实践狗食理论的意义是，新版本一定要比旧版本更具生产力，即使新版本尚未完工或还有错虫。表面上我们没有特别强调狗食理论：管理者不会每天巡视办公室看看是不是大家都在试用自己开发的软件。然而，全体开发团队都太清楚狗食理论对我们的重要性：如果我们在每一个组件上都发挥狗食理论的精神，我们的产品一定会变得愈来愈好。借着经常使用自己开发的软件，我们找到了所有的错虫和瑕疵，生产力提升了，也更懂得掌握设计的效率。

新版本的试用者通常是精挑细选过的。

另一点是我们在拟定里程碑的宗旨时会特别注意“什么时候将什么产品交给什么人”。为了保护我们自己的荣誉，将新版本交付出去时会特别小心，通常对象都是选择性的，因为在错误的时机交给错误的人会造成团队额外的工作负担。因此，里程碑可以说是我们准备将中间产品交给某些早期客户来试用。

所以，我们第一个里程碑的宗旨是：“所有的产品组件都已经被开发团队试用过，并准备将中间产品交给微软内部的单位试用。”简单的说法就是“内部试吃狗食”。而后面一点的（就算是第三个吧）里程碑，它的宗旨也许是：“所有的特色都已经完成，而且软件的使用者接口也不再更动，准备交给一些外部顾客试用。”

里程碑宗旨的关键点在于：

- 宗旨必须点出里程碑的精神所在。
- 判断这个里程碑有没有达到，是看里程碑的宗旨是否被实现。
- 在开始工作以前，所有的人都已经充分明白并接受这个里程碑的宗旨。

法则 41

Look for the natural milestones

寻找自然出现的里程碑

虽然很难精确地把软件开发过程依时间顺序通通记录下来，但有一些事情是必定会自然发生，而且对整个开发活动就像是分水岭一样，有划分的作用，这就是自然出现的里程碑。请注意我所谓自然出现的里程碑，是指“正常”的，而不是“常见”的，正常的是指开发过程本身没有重大问题。以下是六种自然出现的里程碑：

1. 产品设计趋于稳定。

2. 中间产品被明确定义。

3. 团队真正了解要花多少时间和努力才能完成目标（通常这会发生很多次，而且多半是进度落后的时候）。

4. 产品设计被删减，或是资源增加，或是进度延误，或是三者同时发生。

5. 开发活动停止。

6. 产品进入除错或稳定阶段。

这六个自然的里程碑不仅在整个开发过程中出现，也会在单一的里程碑中出现。稍后我们会用较大的篇幅详细讨论，但现在请大家注意的是，每一个里程碑所意味的行动，跟所有的里程碑加起来所意味的行动，应该是相同的，跟整个技术计划所意味的行动，也应该相同。

健康的团队在每一个里程碑所表现的行为应该是可预见的，称作“里程碑的自然模式”（metamilestone pattern），管理者应该予以重视和培养。如果里程碑的自然模式没有出现，就代表团队出了问题，管理者应该采取某些治疗的行动。到目前为止，我所观察到的自然里程碑可以归纳为六种事件，但各种事件会持续多久则是无法预测的，比较可以确定的是，前一个事件结束时，下一个事件一定要跟着出现，前后事件之间可能会有重叠，或是几乎同时发生；这些事件会依序发生或是同时发生，是取决于团队的沟通频宽，在沟通性强的团队中，信息传递得像病毒一样快，这六个事件就比较可能同时发生。

第一个里程碑：设计趋于稳定

最开始的时候，由许多特色组成的产品设计是非常抽象的，是一种令人又兴奋又迷惑的东西，兴奋是因为自己的想法或创意即将实现，迷惑是因为不太确定它究竟是什么模样。一切的人事物（技术计划、团队共识在里程碑之前已经形成）都显示出有这些特色是要完成的，但真的问起来，又没有人确切知道该由谁、在什么时候、做什么事情，才能完成这些特色。这种情况我称之为“软件梦想综合症”（the software dream syndrome）。在产品的设计阶段，

这种细节不明的现象会不断地重复出现，直到详细的工作分配形成为止。

也许你会听到开发人员说：“喔，是的，我们当然要在第一个里程碑中加入footbar的功能；我正在弄出它大概的模样给你瞧瞧。”

而品保人员可能会说：“Widget的特色确定要在第二个里程碑中开始动工，但我还不知道该由谁来负责开发。”

在传统上，或者说至少是在有制度的公司内，对于程序开发一般都要有几项文件的前置作业，包括需求描述、档案或数据库规格定义、各种程序逻辑图等等，不胜枚举。虽然这些文件有助于将程序要做的事情具体化，容易厘清细节，但是基于以下的理由，我个人反对花时间做这些文件：

- 在设计趋于稳定之前必会频频修改，因此使得文件一再重做，结果保持文件的更新变成了一种道德或服从命令，而不是因为事实上真有这个需要。
- 遵守很快就会过时的文件来做事，势必限制了弹性，错失宝贵的机会，而且对健康的团队来说是负担的加重。
- 文件的作用是在开发工作开始前，把所有的事情都

确定下来，结果也会限制了程序的发展。与软件有关的很多事情是在开发过程中，才会确定该怎么做的，而且这时候才能确定怎么做最好，有时候开发人员会有意想不到的创意，为产品增色。所以，如果将软件中未知的部分完全排除，就等于是宣告软件的结束，而非开始。

在软件开发的过程中，最重要的事情不是做出你承诺要做的事，而是在有限的时间、资源限制下，从各种可能的方式中做出最明智的选择，使结果达到最佳化。

产品设计会藉助组员之间各种形式——电子邮件、规格定义、产品模型、交谊厅的聊天、用白板的小组讨论——的沟通、讨论，而由变动渐渐趋于稳定，当产品设计逐渐确定，下一个阶段就可以准备开始。管理者应该注意是不是所有的争议都已经被共识所平息，检查是不是所有的功能特色都包括在设计中，是不是所有的人对产品设计的认知都很类似，如果答案是以上皆是，就可以进入下一个阶段，否则产品设计还不能算是成熟。

第二个里程碑：中间产品被明确定义

如果能用清楚、简明又可信的方式来表达中间产品，没有含糊、复杂或诡异，就可以说“ 中间产品被明确定义 ”。

在项目正式进入开发阶段时，以及需要检验里程碑是否达成时，我们需要的都是明确的定义：产品究竟该是什么模样、有什么工作要做完、由谁来负责完成、品质的要求到什么程度，这些都需要有明确的估计值，虽然不必要和照片一样的精确程度，但至少要像印象派画作一样的明确。

随着产品设计的确立，品保人员、项目经理、开发人员和文件人员就可以组成特色小组了，特色小组各自将负责开发的特色具体化，更清楚地描述他们的需求和目的。他们开始花大量时间讨论和沟通，因为他们有共同的目标和工作，他们关心的是同一件事，而他们对产品设计的认知也相同，于是，开发的细部工作项目逐渐在组员的心中自然成形，为了将这些工作依时间先后顺序安排，并拟出进度的草案，沟通和协商也许要重新展开。

第三个里程碑：团队真正了解要花多少时间和努力才能完成目标

当然啦，在你完成开发工作之前，你永远无法得知真正所需的时间。当产品设计定案，中间产品被明确定义，组员会有一种幻灭似的失落感，第一、因为产品的外观似乎不如想象中的美丽，第二、真正该做的事永远比想象中的多，第三、团队发现自己需要更多的资源、较小的目标、

更多的时间——或者以上皆是。

在这个时候，无论团队多么经验丰富、领导多么卓越，总是有一股失望的气氛在弥漫着，那是一种抑郁而消沉的感受，好像自己被击垮一样。其实这是个好现象，表示组员们对事实有进一步的认识，往后才会因为明白事实而感到自在。幻灭是成长的开始，“软件梦想综合症”于是消失。在健康的团队中，对事实的认识会引发另一波行动的高潮，在我的团队中，我们把它称作“战斗会议”（war room meeting），我们会连续开好几个小时的会，来检讨我们现阶段的状况，并研究出问题的对策。

团队中弥漫着一股失望的气氛，那是一种抑郁而消沉的感受，好像自己被击垮一样。

在前三个自然里程碑中，最重要的事情是：解决那些应该理清的未知。然而，当事情一旦被弄清楚，工作通常是只会多不会少（模糊的面孔比较美丽吧），组员可能会被平添的信息吓住，但是健康的团队会面对这种情况，设

法克服恐惧。

第四个里程碑：产品设计被删减，或是资源增加，或是进度延误，或是三者同时发生

在这个阶段中，团队会利用“软件的三角形”（请参考法则28）来解决所遇到的冲突。如果一切顺利，这时候时间应该刚好是整个项目（或是某一个人为的里程碑）的前四分之一。我们很有理由相信，愈是健康的团队，前三个里程碑会走得愈快，并且会有充分的时间来解决三角形告诉他们的冲突，或是重新建立共识，这差不多是该对里程碑计划稍加修正的时候了。如果一切顺利，虽然对里程碑的某些期望可能需要调整——也许是有些特色要取消、也许得增加一些资源等等，里程碑的宗旨应该是到现在还维持不变。

在里程碑到达之前，谁也无法保证
你能如期到达。

健康的团队会在此时展现出弹性和反应能力，以准确地达到这个里程碑的要求。你不会希望里程碑延期，这是

最后的手段；你也不愿意见到资源被无限地要求增加。在这时候，你最需要看见的是组员之间形成了无数的承诺，培养出解决问题的效率，并准备好表现出最优异的里程碑行为模式。

第五个里程碑：开发活动停止

在某一个时间点，所有撰写程序的动作会全部停止，也就是特色全部开发完毕，此时必须禁止一切修改，以便软件进入完成和稳定阶段。开发工作完成后就不必再写或改程序，这听起来好像是废话，但事实上有必要特别标出这一个里程碑，因为原本预估的时间需求可能不对，这个里程碑等于是计算耗用时间的依据。必须确定软件不再需要开发活动，才算到达这个里程碑。

第六个里程碑：产品进入除错或稳定阶段

如果你能估出“净除错能力”（net bug fix capacity），即“错虫清除率”（bug fix rate）减去“错虫发现率”（bug find rate），而且“净除错能力”是正值，那就差不多可以算出第六个里程碑要花多少时间。而且基于本次的经验，对于以后类似的项目，就可以降低“错虫发现率”，并达到更高的软件正确性。这时候你需要找出这些问题的答案：有多少错虫？要多久才能把错虫全部清除？有多少

比例的程序代码需要整个重新检讨（regression rate）？

当这些问题被认真地讨论时，你大概快接近里程碑的尾声，而且所有能发现的错虫全都被清除了。你会发现利用错误等级评估法（bug triage）来解决问题，会比随抓随改的方式要轻松多了。

法则 42

When you slip, don't fall

如果滑了一跤，别就此倒地不起

进度落后（slip）？不要紧，这并不是世界末日，就像感冒发烧一样，令人不舒服，但也表示身体正在自我治疗。

从另一个角度看，进度落后是团队从软件的梦幻中惊醒，开始面对现实。好像一种醒悟，一种“现在，我懂了”的心情。进度落后像是从一个怪异的、层层包裹着你的梦境，当你从第一层梦境中醒来，想道：“好吧，现在我醒了，我刚才是怎么了？三个月来竟然蠢到完全没发现这个明显的错误——延误了footbar？啊！我终于醒了。”

若把进度落后当成道德上的罪孽，
人们就会全力拒绝面对它。

三个月之后，也许是三个星期之后，你又发现了一个明显的延误，你又想道：“哇，这次我真的是醒了，现在，我懂了。”每一次的觉醒都让你进入一个“真实”的世界，但也许有一天你再次从这个“真实”的世界醒来，发现刚才的“真实”其实不是真正的真实。

你可以把这种不断的觉醒当作成长的过程。当然，每

一回觉醒都带给你更多的知识和经验，并让你更接近真实，更丰富你的人生。这个多层梦境的醒悟计数器是每个新项目从零开始，每一次觉醒加1，如果你已经有了n次觉醒的经验，你就是从n开始你的醒悟计数器。

任何项目都有可能碰到进度落后和觉醒，但有些觉醒不一定是跟随进度落后而来，这很有可能，但有点危险，并且不是好事。以下是你对进度落后应有的正确观念：

- 进度落后与道德无关，请切记！这并不是失败或做错事，应该受到责罚，这是创造智能财产过程中无法避免的。千万不要有道德上的罪恶感或良心上的负担，并且要求你的团队也要这么想。因为组员们最怕被威权形象责备，他们会因此而陷入一种很微妙的精神压力中，在讨论时拖拖拉拉不着边际，在恐惧犯错和渴望荣耀之间挣扎，在责难和赞誉之间游移，结果会尽全力摆脱这种压力，会胡思乱想，因此而消耗所有的心力。所以，千万不要让任何人对进度落后有道德上的联想，认为这是不应该或不对的事情。
- 不要隐瞒事实。人在遭遇冲突或危险时，一定但愿没这回事，而有逃避的倾向。在进度落后时，你和

组员们都难免会不想谈它，但这时候特别要克服这种天性，发挥你卓越的领导力，把事情变得不一样。不要瑟缩在你的办公室中。喃喃自语："我不敢去看，进度迟延了，……不，我不相信有这种事……"勇敢地走出去面对问题，你要打败问题，而不是逃避问题。

- 化阻力为助力，利用进度落后来激发效率。对于进度落后的认知是一种醒悟，组员们开始有重新振作的精神，有最新的信息，有最新的看法，创意和想象力也可能有新的境界，领导者必须得把这些新兴的朝气导向最有效用的地方，不要管传统规矩的包袱，进度落后是考验团队弹性的最佳时机。

进度落后不是问题，被进度落后吓倒才是问题。进度落后并不代表产品的难度太高而无法开发。但是如果进度已经落后却未被察觉，那表示组员们不思考、不观察、不讨论，此时组织可说是濒临瓦解了。

善用你的迟延，这是最能看出你领导能力的时候，此时也是组员最脆弱也最需要你的时候，在这个时候组员最能把你的话听进去，此时组员的学习能力最强。如果你在办公室内激动得大喊大叫，指天骂地，那就错失了赢得组

员的心的大好机会。你必须说：“ OK，进度落后了，让我们来看看问题出在那里？……今天下午五点在会议室，我们要检讨每一个细节，问题一定要设法解决！ ”当组员了解到你不是企图卸责或算帐，而是真诚地想解决问题，就会乐意把一切开诚布公地摊开来谈，大家一起研究问题，从各种角度去设法克服问题。“ 进度落后 ”反而变成大家宝贵的成长经验。

进度落后

里程碑达到了吗？其实这个问题很难有完全客观的衡量标准。由于产品设计可能会变，而品质本身就是无形的，所以里程碑的到期日也可能会变，因此，里程碑到达与否，变成了一种人为的判断：当需要回头修改的项目愈来愈少、里程碑的宗旨达成、组员对于下一个里程碑跃跃欲试时，你差不多可以宣布里程碑成功。当然这一刻是很难决定的。对于这个问题，我只能举个反面的例子来说明这个观念：比方说你去问路，人家的回答是：“ 你一直往前走，若是看到尖塔教堂，就表示走过头了。” 如果你已经接近里程碑，但开发活动还在进行，就表示已经走过头了。身为领

导者，你必须时时掌握软件的一切状态，必须知道在每一个阶段应该进行那些活动，又该停止那些活动。

曾经有一次（也许不只是曾经），我们接近M1了，但一切工作都陷入混乱，我们只剩下两星期，而软件的情况却糟得难以想象：错虫数（bugs count）实在太高了，虽然每个程序设计师的错虫率还在合理范围，但加起来的错误率却高得惊人，从我们每天的错虫增加数目显示出软件距离稳定还差得很远。

我们没有办法每天做软件建构（请参考法则32），虽然主要的产品组件多半都能建构成功，但就是有那么几个不成的，表面上都能编译和连结（compile & link），但就是建构不起来。这个问题大大地妨碍了下一步骤工作：“猎犬测试”（sniff test）和程序代码分析。当其中的一个定点问题好不容易被找出来解决，这至少又花掉了一天的时间（请记得，在里程碑的后期，一天就可能耗用是百分之十的资源）。

组员和领导者并没有感到特别紧张，通常在接近里程碑的时候，组员们都知道自己已经尽力，但不知怎地，就是觉得不对劲，里程碑的精神似乎不见了。

事实上，这个问题的主要原因（也是一般常见的状况）是出在沟通和协调不良，起于至微而酿成巨祸。当一只程序置入时，就改变了另一只程序的假设前提，结果另一只程序就遇到了意料之外的状况。A小组依赖B小组的公用程序，A小组改好也测试过了自己的程序，但前提是B小组维持在第n版，然而B小组却改到了第n＋1版，虽然改得很少，但恰好是悠关A小组的部分，B小组误以为不会影响A小组的程序，所以没有告知A小组，结果A小组的程序就建构不起来，因为它应该与B小组的第n版搭配，结果B小组已经置入了第n＋1版的程序。

领导人应该制止这种情况吗？

绝大多数的项目经理一旦发现这种情况，第一个反应是采取补救行动——成立特别小组来接管事务。但是这么做可能会造成一些问题：最严重的是，这种行动的代价甚高，虽然这么做一定会使组员非常注意软件的状态和风险——我把这种活动称作“强烈焦点”（radical focus）——你等于是用领导者的职权来使组员就范，这种敲响警钟的方法不能太常用，否则团队就会发生“狼嚎症状”（cry wolf syndrome），

懒得对老板太认真，老板只是叫得凄厉而已。

如果进度落后的情况太频繁，是很危险的。进度落后变成正常的事，此时运用领导者的威严来提高团队的危机感，这种手段只有在士气低落时才适用，这时团队需要一点刺激，就是“强烈焦点”活动：让小组在同一时间内担负几个小项目（大概不超过两个），但是这几个项目的目标是南辕北辙、毫不相干的。

而当“强烈焦点”展开时，即使是健康的团队也多少会反弹，觉得领导人在“反授权”（disempowerment），于是一切不顺利的牢骚都算在“当权官员”的帐上，把这项措施解释成不尊重团队和团队的决定。防卫心理升起，而且开始情绪不平，甚至可能因此怨恨起来。即使是最优秀的领导者在采取“强烈焦点”措施时都会遭遇一段抗拒期。在违反对方的意愿之下教育他，虽然做得到，但要辛苦很久。

同时，你要和团队中最优秀开发人员的反抗权威心理作战，他们可能不会很好沟通，而直接把你当成笨蛋（请参考法则4）。在另一方面，你也有可能误判士气的低落程度，这时候麻烦就大了，健康的团队

会对你反弹，如果你死不认错就会丧失组员对你的尊敬和忠诚，要不就是你的自大使你愈走愈远；而团队如果不是那么健康，不敢对你谏言，那你绝对会陷入万劫不复的失败。

在你决定采用“强烈焦点”时，以上所谈的故事都是你的成本和风险。你的“固定成本”至少是几个工作天（大概一星期左右），而你必须抓住组员的注意力，保持他们的兴趣，让组员明白你是玩真的，让他们愿意告诉你心里的感受，让他们开始做点实际的事情。在这个起步阶段所需的时间视团队的大小、沟通的意愿与能力而定。

现在我们回到接近第一个里程碑时的原地踏步，领导者要团队有什么样清楚的认识（awareness）？如何做到这一点？然后会引起什么行为？

在我们的案例中，要团队清楚认知的是M1迫在眉睫，而且除非过了M1这一关，否则我们不可能完成产品；我们也希望让团队练习使软件“整装待发”、完工交货；最后，我们希望团队经历一下不成熟的团队合作所带来的可怕后果，而从中学到我们每一个人都有等量的责任，也是全部的责任-就是达成M1

里程碑。

在我们的案例中，“强烈焦点”的施行是不合适的，因为团队的问题不在士气低落，而且事后发现，团队的反弹非常强烈。我们应该为真正的紧急情况保留子弹。

那该怎么办？

很明显地，我们不可能让整个投资都押M1这个岌岌可危的一注，我们知道一定不能让问题持续下去。我们现在正处于瓶颈，无法建构软件的不良效应正扩散开来，这一点可以从超高的错虫率可以证明。大家对程序的置入动作更谨慎，必须得仔细检查是否会妨碍到别人的程序，也就是速度慢了很多。错虫数仍然居高不下，为了除虫而做的修改仍然会发生前述版本不同步的问题。当软件无法建构的状态持续太久，会使得更多的程序搭配到别人的旧版本，结果使得软件建构更加成功无望。我们很关心这种情况，组员对危机的感受度似乎无法和工作的进度成正比。

团队合作是相互影响的。为了完成整体的工作，你必须划分出局部的工作。你可能让三人或四人组成一个小组，这一小撮人的影响力就会扩及全部的团队

或工作，相对地，这一小组的失败也会扩散到全体，导致软件严重的伤害。此时你的重心集中放在问题根源的小组，这会比企图整顿全体，结果却头痛医头脚痛医脚地到处乱转要有效得多。

我们经过一番诊断和试疗，终于决定：在我们做到每天都建构成功以前，禁止所有的程序置入。有三个主要的原因迫使我们非这么做不可：第一、没有协调妥当的置入动作，会使很多其他的小组受到影响，最后使得大家都无所适从；第二、负责建构的小组经验不足，因为建构的任务不久前才从开发部门转移到品保部门；第三、有一个小组总是通不过测试，它是问题的根源。

好吧，以上这些症状，告诉我们什么？这个团队的心理状态到底出了什么问题？

看看团队出了什么问题？

在起始阶段我曾经提过的一项理论：从软件能够看出团队的心理状况（psyche）。现在事情不对头了，团队显然无法发挥应有的工作效能，此时你必须问问自己，团队的这些行为是否透露出什么样信息？

首先，我们来看软件始终无法建构的问题，协

调不良的程序置入动作（surprise check-in）代表团队缺乏整体性的自我认知：小组之间缺乏交谈。每天的开发目标和程序撰写都是纯组内的，小组之间的程序开发动作太不协调，彼此相互掣肘。每一个小组单独来看差不多都是朝着M1的目标前进，但每个小组的工作成果却无法组成产品。个别组员的贡献在组内都很不错，但到了组外就完全看不出来。

第二个问题是建构小组的经验不足。品保人员没有做过建构，以前都是由最资深的开发人员负责每天的建构。这一群开发人员在技术上有足够的能力解决任何建构失败的问题，因此开发人员已经习惯对于建构小组期望过高。这一群资深开发人员像看门狗一样为软件建构把关，解决任何问题，甚至指导其他人关于程序开发的知识和技巧。而新接手的品保人员没有办法对软件（和其他开发人员）照顾得那么细微，没有技术能力来立刻解决建构失败的问题（例如debug），而且这也不是品保人员的责任。另一方面，品保人员正试着融入团队文化，过去是开发人员最大，现在开始要在团队中建立品保人员应有的角色及工作模式，品保人员一时还无法坚决地要求开发人员把程

序改到什么地步。

因此，开发人员对于建构小组的“不良”工作表现觉得失望，无形中，开发人员开始认为建构小组都是笨蛋，他们已经习惯有资深高手替自己修缮程序，程序设计设计师（尤其是那些负责核心部分的）已经被宠坏了，忘记去思考自己的程序与整个软件的关系，也不认为自己的程序必须处处配合别的程序才行，每个人都认为我只管写自己的程序，建构成不成功不关我的事。现在资深的保姆不再为大家打点这个清理那个，每天都有高品质的程序是每位开发人员的责任。建构小组的工作是把品质好的程序建构成品质好的软件，维护可预测的建构程序和结果，找出瓶颈所在，并且监督开发人员改正自己的程序。开发人员对于这项责任归属一开始当然是不情不愿的，他们原本对于建构失败的反应是“建构小组今天没把事情做好”，现在，要把他们的观念转变为“是今天的程序不够好而无法建构”。

第三个问题是有一组总是做不好，他们的程序总是无法建构成功或趋向稳定，一直通不过测试。这并不意外，这部分程序长久以来受到忽视，没有足够

的开发和品保人员照料，项目经理根本没有为它付出关爱，也许每天的建构失败就是它的不平和控诉吧！这一部分程序对产品而言其实是挺重要的，只是不知道为什么它老是被当成二等公民。于是我们重新分配资源，拨出足够的人力来把它弄好，我们得补偿过去对它的忽视和差别待遇。从团队的心理状态或产品本身，都可以清楚看出我们过去没有给这一部分程序应有的重视。

总而言之，团队没有协调好，责任没有划分清楚，而且没有相互为对方的程序负责、全体共同对软件负责的成熟心态，管理者也疏于分配资源到适当的地方。等到我们把问题诊断清楚，收回“强烈焦点”的错误措施，开始真正对症下药的治疗行动。

让组员学习彼此互相配合

很明显地，由于团队缺乏共同的任务认知，小组与小组之间既不交谈，也不会朝共同的工作目标努力，因此我们必须充分利用团队工作的扩散效应。也就是说，如果你在某处解决了一个问题，就等于是对全体解决了这个问题。项目经理是团队中理所当然的领导者和灵魂人物，他必须有很好的管理者形象，负

责找资源、保护团队、带领团队迈向成功，他同时也是执行长，在别人眼中他就等于产品的总括，每一个人的事情都是他的事情。

项目经理或上级主管可以命令团队应该维持某种程度的沟通，也可以用鼓励的方式达到更好的沟通效果。然而，若是一开始组成团队就期望很有效率的整体沟通，这野心太大了，失败的可能性会很高：因为团队中的事情太多，组员与组员之间的关系太复杂，而且随着团队人数的规模而呈指数成长。比较好的做法是，让几位项目经理互相讨论他们的工作状况、目标、希望以及长期或短期的技术计划，这种层次高、人数少、没有隐藏的沟通虽然困难，但并非不可能。然后让各项目经理去维持自己小组的沟通。用这种方式，团队会很自然也很快地建立起整体的沟通，虽然你不能命令或创造团队整体的沟通，但你可以起个头、可以培养。

虽然项目经理会有很多位，各自负责不同的产品领域，但其中有三位的专长和职权可说是综览全局：一是三人中最年轻的，但资历比较丰富，曾经领导过多次产品推出工作，另外两位比较年长，虽然从未担

负过如此重任，但也是快速窜红的“新秀”。三人之间的沟通很差，其中一个原因是，最年轻的那位没有给另外两位足够的引领，让他们能更进入情况。

部分原因是这三位项目经理分别辖属于不同的老板。还有一个原因是在M1的早期阶段，此时项目经理们还未能充分掌握自己应该担任的角色，他们会倾向萧规曹随，看看前一位担任此职务的人是怎么做，以为自己照做就够了，或是他们以为有些事已经有前人完成，自己毋需费心，而且三人彼此之间的沟通相当少。这个问题反映在每天的建构上，我们必须将建构失败的问题和三人之间的沟通问题一起分析，深入了解二者的关系。

代罪羔羊

最常导致建构失败的原因是程序置入时，版本发生不协调，结果发生更难建构的连锁效应，另一个无法建构的原因是置入的程序其实尚未完全确定没有错误。项目经理当然不可能不知道本组现在正在修改什么程序，项目经理（或至少是品保人员）应该把不够完美的程序退回去给开发人员修改，并追踪修改的进度。程序置入前的品质没有被重视和把关，当不严

谨的程序进入了建构程序，或是开发程序应有的纪律没有被认真遵守，在心态上就已经开始散漫，开始推诿塞责——“ 都是建构小组没有做好工作 ”。

推诿塞责是团队合作失败的必然现象，尤其在责任感很狭隘、误解或扭曲的环境中，更容易泛滥成灾。对道德标准高的某个人或某个群组，特别容易在推诿塞责中受到伤害。代罪羔羊存在，就表示团队的心态有问题，这是一种病态性的防卫反应，好像只要有别人可以责怪，自己就可以没事，人们直觉地想找个替死鬼，以便证明自己的表现没问题，而忽略了整体的情况正在急速恶化（那不是我的错！）。推诿塞责是将团队的整体问题与自己隔离，把问题简化、窄化的一种企图。

推诿塞责当然是人类原始的、不成熟的恶习之一，但对于团队的心理状况却有短期的改善功效。这也就是为什么推诿塞责的现象这么普遍，而且是在短期合作的团队中。然而，推诿塞责毕竟不是长久之计，短期的好处实在没有什么大用，凡是聪明、有自信心的人都不会采用推诿塞责的方式，只有比较年轻、比较怕受伤害的人才会这样做。

很不幸，推诿塞责是会生根的恶习，它会扩散、传染出去，直到整个团队的工作绩效低落到大家都无法忍受为止。区分“推诿塞责”和“合作无间”最明显的指针是看看团队是否对于被当成代罪羔羊的牺牲者视若无睹。大部分的受害者会觉得自己是被罗织入罪，然后这种压力会造成受害者更强的防卫心理和“太极拳给他打回去”的行为。推诿塞责的目的是逃避检讨问题，姑息问题的存在，找个代罪羔羊替自己承担责任。推诿塞责的结果会像滚雪球般愈来愈严重，太极拳大战会从小组打到管理者，再扩大到管理者的管理者，最后会造成组织的致命伤。

推诿塞责对组织的破坏性可以分成两个层面。那只成为众矢之的的倒霉羔羊会被自己的同仁当成坏蛋，被排斥和被轻视的情况使他不得不学习各种保护自己的方法，当然包括把责任踢回去，或是另外找个人来当代罪羔羊的代罪羔羊。无论团队原来的工作效能如何，绝对会因此而急剧下降。一旦第二只倒霉的羔羊（很可能是宰杀第一只羊的人）也面临被指责的危险，就会开始出现两种对组织具有破坏性的后果：第一，因为把责任推给别人是这么的容易方便，所以

真正的问题就更不会被人分析研究和解决，而且根本没有人会注意到，人们忙着拼命把责任向外推，结果加速了推诿塞责的病情扩大，更快速地侵蚀组织原本就偏低的工作效能；因此，只要有一个人变成倒霉的代罪羔羊，就开始了相互推卸责任的恶性循环。

第二个破坏性的后果更加严重，因为内部互相踢皮球，同事彼此之间关系的平衡遭到破坏，必然要引来高层主管的干涉（讽刺的是，也许高层主管是为了解决推诿塞责而加入战局的），于是局面变成各执一词的两造双方，而高层主管要做出仲裁。判决的结果很可能双方都不满意，因为实在太难找出适当的方法解开这个结，当然会造成彼此的信任丧失，忠诚度降低，对于主管的敬意也打了折扣。

任何领导人都必须对“推诿塞责”的情况有正确的认知，它是一种团队心理的病征，尤其是正在扩散的推诿塞责，更是团队工作效能的严重伤害，长期下来必会危害整个组织。

在我们的案例中，项目经理应该有责任为建构小组设立明确的目标，与他们一起建立团队对建构小组应有的期望：现在不再有资深开发人员为大家修整

程序，将软件建构成功是每一位开发人员的责任，而建构小组的责任则是把不够理想的程序退回去，并协助开发人员找出程序无法建构的问题症结。

到最后我们为了解决这个问题，不得已将 M1的时程延后几天或一星期。延期已经无法避免，但借着延期的动作，让组员更清楚了解到我们的团队其实很脆弱，即使初期曾经信誓旦旦，即使管理者对组员充分信任而不采取强烈措施；让延期在组员心中刻下难以忘怀的教训。现在，我们不得不采取扼阻的行动，我们停止所有的开发动作，直到软件能建构成功为止。而且除非程序经过确实的测试，不准任意置入。

于是我们让开发人员了解，他们的工作是多么重要，万一有任何瑕疵，将会影响整个软件的建构，所以程序在置入前一定得坚持完全没有错误。最后我们加派人手给那个孤儿程序，让它很快地修改妥善，不再妨碍建构。

沟通方式与效果

事情的沟通方式等于暗示了事情的意义。一小段没说什么特别事情的电子邮件，人们也不会对它特别注意；而当面开会，则是最强力的沟通法，表示所

谈的事情非常重要，或意味着事情不是我们这群人加上我们的顶头上司就能决定，得来点组际协调。

有关于程序置入程序的问题，被安排在“特别会议”（special war room meeting）中讨论。以我们公司的文化，特别会议是不同于一般会议那样送个小开会通知，该到的人到齐就行，特别会议意味着：第一、与产品的推出有关；第二、一位以上的资深经理参与；第三、讨论的主题非常重要，而且需要共识；第四、虽然没有形式上的规定，所有的小组领导最好是参加，因为这个决策必定事关重大。通常是由最高项目经理来召开并主持这个会议，但其他的领导人也可以召开这种特别会议。

请注意这和宣达命令的会议是完全不同的，如果资深项目经理说我们要怎么怎么做，然后大家照章行事，虽然也会讨论，但基本上是主持人所许可的讨论范围。既然是宣达命令，已经没有什么讨论的空间，讨论只是更确立决策或更了解决策的用意。如果与会者中有人比决策者更聪明、更有想象力、对问题有更深入的了解，或是有更好的办法，他会觉得挫折和气愤，其他人则因为不能使用更好的办法而感到遗憾。

在会议的讨论过程中做出决策，比较容易得到高明的决策，执行起来也会有效得多。通常是有人发现重大问题，而且已经想了又想，询问过专家的意见，也许已经有了初步的行动方针，在会议中邀集众人的看法；通常与会者已经在会前对这个问题有某种程度的了解，甚至已经思考过一些解决方案，即使不在事先排定的议程内，也可以提案讨论无妨。通常在这类会议中，提出的意见或建议都是经过深思熟虑，初步的行动方针被补充、被改良，共识在这种过程和气氛中凝聚。与会者有意见的话，应该让别人都了解，然后在大家都能认同这个解决方案的情况之下，齐心解决问题。

法则 43

Don't trade a bad date for an equally bad date

不要因为进度落后而更改最后期限

将来有一天我退休了，真希望能到学术界研究出有关进度落后的数学公式。目前我能确定的是，第一次的进度落后往往是最严重的一次，之后因为经验的积累会使你更能掌握进度，所以，当你第一次遇到进度落后时，千万得有耐心。

进度落后的程度是与计划的不确定性成正比。就好像你在翻阅一本名叫未来的书，你无法预料未来是什么，直到你走到那一步；随着时间过去，本来不确定的事会变成已知，于是整个计划的不确定性会随着时间而降低，进度就愈来愈能准确掌握。理论上，进度落后的程度相对于计划的不确定程度，并不是数学上的趋近于零而永不等于零，而是，如果不确定的程度为零，就不会发生落后。

我相信应该有这么一个算法，用时间轴（X）和不确定性（Y），给定有任两点进度落后的值，就可以投射出理论上的项目完成日期。我相信这种算法，因为我没有一次不用到它，虽然我无法提出精确的公式或证明。但请你睁大眼睛注意，你所经历的进度落后是否一次比一次轻微，如果是，表示你渐入佳境，如果不是，那你离目标渐行渐远。

在你知道自己什么时候能够完成以前，
你往往会经历自己即将落后的煎熬。

但是，千万不要为了减轻进度的延误，而将最后期限向后挪，这种交易不划算，你会因此而信用扫地。一般来说，早在你知道确切的可完成日期之前，你会知道项目已经延误，这时候，整个团队乃至于全公司所有的人，包括外界的新闻探子（如果他们知道的话），都会问你：“既然每个人都知道延迟势难避免，何不干脆将到期日延后？”仅管有庞大的压力逼你延期，但原来正式的到期日仍然是项目正式的最后期限，如果轻言更改，团队成员会因此隐约觉得原来的日期设定不妥当，造成人心浮动。项目的正式到期日并不是不能改变，但不可因为进度赶不上而将它延后。

在进度落后时，最最糟糕的办法是估计落后的时间差，而将到期日向后顺延。这等于是将一件你确定已经做错的事情（进度落后），往后再背负下去（进度还会再次落后）；这个方法似乎是很方便的权宜之计（我们再也不必

去想它），却是最极致的愚蠢。这时候你最确定的事情就是必须思考为什么会发生进度落后。

即使你现在已知的信息，比起你在项目初期设定期限时要丰富许多，但大概依然不足以让你决定新的时程表。不过话又说回来，要说明在什么情况之下可以重设新期限本来就是极困难的事。比较好的一般性原则是，除非你已经很确定各个组件产生多少程度的落后，还欠缺什么样的内容，发生落后的原因已经确实找出来，并且问题已经在克服中，否则绝对不要轻言更改最后期限。当然，要做到这个前提，必须全体团队的合作，正确的领导，以及全心的投入，在能够精算出期限的数学公式（我预备退休后要研究的）发表之前，你实在无法估算出正确的期限。你应该很务实地将目标放在最近一次的里程碑上，并且你必须做好心理准备，往后的里程还是充满不确定的因素。

珍惜你的时间，把它善用在找出发生进度落后的原因上，而不是计算我们该延期多久，你不会因为花时间解决问题而耽误更多的时间，相反地，你会使团队以后走得更快更稳，早一点达到目的。

法则 44

After a slip, hit the next milestone, no matter what

延误了这个里程碑，就一定要如期到达下一个里程碑

我们必须明白，每一次的延误，就是你和团队信心的一次受挫，所以，延误这个里程碑时，最好的补救办法就是无论如何绝不延误下一个里程碑。团队必须挽回对自己的信心和对理想的承诺；因此，下一个任务必须准时完成的意义更重大，团队需要重建信心。

你知道进度落后的情形有多严重，也知道是什么原因造成，你知道该如何改正问题，然后你建立一个比较近程的、保守估算的下一个里程碑，这次绝不能再失误，然后重新发布这个消息。如果你的时程表是以下三周内不做任何事，那也很好，喔，实在太过头了，是不是？我的意思是你一定得定下一个能够完成的里程碑，哪怕这个里程碑在内容上没什么了不起，并且绝不失误地达成这个目标，这个动作必须铿锵有力才行。绝不能再次延误，因为团队需要藉这个机会重建自信。如期达成里程碑，这个目标的成功，会带给团队鼓舞的力量，重新充满活力，相信自己有能力达到对时程的承诺。这种心情是非常重要的，如果团队相信自己能够如期达成里程碑，他们就会尽一切努力去达成，这似乎是人类应有的工作方式。

法则 45

A good slip is a net positive

把延误当作宝贵的学习机会

把进度落后当成一种宝贵的学习经验：你曾经不明白的事情，现在明白了，此时正是分析、了解、思考和消化的最佳时机。这时候学到的心得是最深刻的，也许很难用言语描述。如果你只是用刻板的印象去看待进度落后，把它当作是一件坏事，那这个进度落后的事实不能让你进步，进度落后会成为你深怀恐惧却时常发生的事。

虽然有这么多不确定的因素会造成进度落后，延误几乎是必然发生，甚至已经被视为正常。有很多软件开发工作本质上是具有实验性的，新的平台、新的操作系统、新的程序技术，往往使得每一个新项目都充满不确定性。

延误既然不可避免，为了防止酿成大祸，你必须衡量各种可能造成延误的因素。最理想的情况是，你事先已经知道一个以上的不确定因素，并且让所有的工作人员明白进度本身必须包括这些风险，让大家知道我们如何评估这些不确定因素对进度的影响，如何估算风险所占的时间，这项预估能力是团队的知识与技巧，对未来大有帮助的。你同时必须懂得判断人们是不是在做“对”的事情，进度的落后经常的原因是，人们花太多时间在研究一些比较外围或衍生出来的特色，事实上这些工作对产品的核心信息没有什么帮助。

如果延误是你突然的发现，那就表示沟通的渠道不畅通，你必须加强团队的沟通能力，你必须搬出一大堆详细的信息给团队成员看，让他们对这个问题有具体而真实的体会，延误暴露了团队的弱点，但也提供了毫无保留的检讨机会。你必须确定每个人的每个角色都得到了需要的指点。

延误也是重新评估未来相对于什么目标该有什么资源的好机会，日后对该有的产品功能特色，会更加务实地捡选，对于不利于团队工作效率的特色将尽量不予纳入目标，对于资源与特色相互的作用也能更准确估算。

总之，能够让团队学习的延误，绝对是件好事。

法则 46

See the forest

见树亦见林

如果本项目有六个模块，各有90%的部分已经完成，那么你已经完成了54%。每个模块完成了九成，听起来是个挺不错的成绩，但不能当成整个项目完成了百分之九十，它们之间不是相加的关系。你必须“见树亦见林”。如果任何一个模块完成比率是零，那么整个项目的完成率也是零。

用这种数学运算去计算整个项目的完成率是非常正确的，因为每一个组件对产品都是相等的重要性，假定你的团队是平均分配工作，那么任何一位失败就是团队全体的失败。

必须注意的是，没有一件事情是能够有百分之百把握的，如果是，那很可能发生“骄兵必败”的后果。今天的英雄可能明天会惨遭滑铁卢，我每每对曾经跃升的明星和它坠落的速度震惊不已。你必须设法让组员了解声誉得来如此不易，却是如此脆弱易碎，不堪一击。团队必须时时刻刻惕励自己，不要任意把别人当作笨蛋，不要狂妄自大，不要自我膨胀，不要让自己心中的魔鬼打败自己。

法则 47

The world changes, so should you

世界在变，所以你也得跟着改变

成功的软件开发工作有一项重要的特质，就是能够从每天涌进的新信息中，做出正确的决策。你不必过度拘泥于计划和进度，那是人造的事实，难免有失真的时候。改变是机会之母，如果你对学校课堂中教你的知识不能活用，你会失去许多机会，会很难适应这个迅速变迁的环境，你会把改变或机会当成是障碍，当成计划和进度的干扰，而不是充满潜力的转机。

软件开发也是一个不断改变的有机体，通常一个大得不得了的困难，事实上代表着团队对于开发出好软件的强烈欲望。在你作出直觉反应之前，在你把它当成问题去消灭之前，花点时间挖掘这种心理状态背后的玄机，试着把这股能量导入正确的方向。问题本身可能表达着许多意义，你必须运用你的观察力、想象力、第六感，决定如何将软件开发的过程导引顺畅。绝对记住你是在领导一群人的心，共同进行创造性的工作。如果可预测性愈高，代表创造性愈低，所以，每件事情都得保持弹性。

当然，你不会希望轻率改变，任何改变都不可导致你偏离方向，过度的改变会使人无所适从。弹性不代表随便（randomness），弹性是延展性、适应性，是自然地改变，而随便则是突然地、毫无关连的胡乱改变。如果外界没有

变化，随机的改变一下也许会有点灵光乍现的好处，但维持原状则更加保险。只有在改变能够加强整体效率时才能接受它，改变的目的是让团队的工作最佳化，而不是导入更多的复杂性。

我们这样举例说吧，也许有一天，你想增加一个里程碑，没有明显的理由非得这么做不可，但是有确定的迹象显示这是目前的发展方向所趋，这时候增加里程碑可能增加了进度落后的风险，但也可能增加产品推出后的机会。我特别提出这个例子，是因为我的团队最近经常遇到类似的困扰，虽然我们一直以准时推出产品为荣，但是环境的变化使我们愈来愈觉得必须与众不同，必须有所突破，在原定计划中加入一个新的里程碑，做一点特别的东西。每一个项目都像是新生的孩子，每个项目都是独特的，你必须顺应项目独有的特质，配合它定出最适当的里程碑，并且充分支持它，用它的语言去表达它，而不是像切蛋糕般，切成你想要的样子。

虽然你想做些改变，你未必有勇气实行。

伟大的软件必定只有一个中心思想，至于品质能够实现到什么程度，依赖领导者能否带领团队融合无数个小而重要的改变。如果你能在混乱中辨识出对项目最有意义的改变，并且引导团队去适应它，将它融入团队的精神中，将来就会在产品中表现出这项改变，呈现在顾客眼前。

抗拒变化是失败的策略，你必须学习找出无法抗拒或是对产品有利的改变，拥抱它、适应它，最后这项改变会带给你丰富的收获。这是困难的心路历程，你需要刚铁般的意志，因为你得依赖自己独有的远见，你所看见的没有任何人能看见，你会很寂寞，每每在深夜被怀疑缠绕而不得安眠。虽然你实在很想做些改变，你未必有足够的勇气付诸行动，你等于是在挑战着人们的期望，而且你自己的感觉也会因此而被左右，你不敢确信这些改变是好的。你会面临艰难的抉择，在改变与不改变之间饱受折磨。

然而，不论改变有多大的阻力，只要是必须的改变，你就得排除万难接受它，否则你会被它摧毁。

对我而言，即使只是把这些想法写下来，都令我感到惊慌害怕，但这实在是我在软件开发的过程中，确确实实的经验体会啊！