第一章 高质量软件开发之道

本章讲述高质量软件开发的道理。

为了深入理解软件质量的概念，本章阐述了十个重要的软件质量因素，即正确性、健壮性、可靠性、性能、易用性、清晰性、安全性、可扩展性、兼容性和可移植性。并介绍了消除软件缺陷的基本方法。

人们开发软件产品的目的是赚钱。为了获得更多的利润，人们希望软件开发工作“做得好、做得快并且少花钱”，所以软件质量并不是人们唯一关心的东西。本章论述了“质量、生产率、成本”之间的关系，并给出了能够“提高质量、提高生产率并且降低成本”的软件开发方法。

1.1 软件质量基本概念

1.1.1 如何理解软件的质量

什么是质量？

词典的定义是：（1）典型的或本质的特征；（2）事物固有的或区别于其它事物的特征或本质；（3）优良或出色的程度。

CMM 对质量的定义是：（1）一个系统、组件或过程符合特定需求的程度；（2）一个系统、组件或过程符合客户或用户的要求或期望的程度。

上述定义很抽象，软件开发人员看了会一脸迷惘。软件的质量不容易说清楚，但我们今天非得把它搞个水落石出不可。

就以健康作类比吧。早先人们以为长得结实、饭量大就是健康，这显然是不科学的。现代人总是通过考察多方面的生理因素来判断是否健康，如测量身高、体重、心跳、血压、血液、体温等等。如果上述因素都合格，那么表明这人是健康的。如果某个因素不合格，则表明人在某个方面不健康，医生会对症下药。同理，我们也可以通过考察软件的质量属性来评价软件的质量，并给出提高软件质量的方法。

一提起软件的质量属性，人们首先想到的是“正确性”。“正确性”的确很重要，但“运行正确”的软件就是高质量的软件吗？不见得。

这个软件也许运行速度很低，并且浪费内存，甚至代码写得一塌糊涂，除了开发者本人谁也看不懂也不会使用。可见正确性只是反映软件质量的一个因素而已。

不贪污的官就是好官吗？不见得。

时下老百姓对一些腐败的地方政府深痛恶绝，对“官”不再有质量期望。只要当官的不贪污，哪怕毫无政绩，也算是“好官”。也有一些精明的老百姓打出旗号：“宁要贪污犯，不要大笨蛋”。

相比之下，搞软件开发是够幸福的了。因为我们能通过努力，由自己来把握软件的

质量。让我们好好珍惜权利，不要轻易放弃提高软件质量的机会。

软件的质量属性很多，如正确性、精确性，健壮性、可靠性、容错性、性能、易用性、安全性、可扩展性、可复用性、兼容性、可移植性、可测试性、可维护性、灵活性等等。还可以列出十几个，新词可谓层出不穷。

上述这些质量属性之间“你中有我，我中有他”，非常缠绵。如果开发人员每天要面对那么多的质量属性咬文嚼字，不久就会迂腐得象孔乙己。我们有必要对质量属性作些分类和整合。质量属性可分为两大类：“功能性”与“非功能性”，后者有时也称为“能力”（Capability）。

从实用角度出发，本章将重点论述“十大”质量属性，如表 1-1 所示。

其中功能性质量属性有 3 个：正确性、健壮性和可靠性。

非功能性质量属性有 7 个：性能、易用性、清晰性、安全性、可扩展性、兼容性和可移植性。

“十大”软件质量属性	
功能性	正确性（Correctness）
	健壮性（Robustness）
	可靠性（Reliability）
非功能性	性能（Performance）
	易用性（Usability）
	清晰性（Clarity）
	安全性（Security）
	可扩展性（Extendibility）
	兼容性（Compatibility）
	可移植性（Portability）

表 1-1 “十大”软件质量属性

为什么碰巧是“十大”呢？

不为什么，只是方便记忆而已（如同国际国内经常评“十大”那样）。

为什么“十大”里面不包括可测试性、可维护性、灵活性呢？它们不也是很重要吗？

它们是很重要，但不是软件产品的卖点，所以挤不进“十大”。我认为如果做好了前述“十大”质量属性，软件将会自然而然地具有良好的可测试性、可维护性。人们很少纯粹地去提高可测试性和可维护性，勿要颠倒因果。至于灵活性，它有益处也有坏处。该灵活的地方已经被其它属性覆盖，而不该灵活的地方就不要追求灵活性。

根据经验，如果你想一股脑地把任何事情都做好，结果通常是什么都做不好，做事总是要分主次的。什么是重要的质量属性，应当视产品的特征而定，请读者不要受本书观点的限制。最简单的判别方式就是考察该质量属性是否被用户关注（即卖点）。

1.2.2 提高软件质量的基本方法

质量的死对头是缺陷，缺陷是混在产品中的人们不喜欢、不想要的东西。缺陷越多质量越低，缺陷越少质量越高。

Bug 是缺陷的形象比喻，人们喜欢说 Bug 是因为可以把 Bug 当作“替罪羊”。软件的缺陷明明是人造成的，有了 Bug 这词后就可以把责任推给 Bug——“都是 Bug 搞的鬼”。唉，当一只 Bug 真是太冤枉了。

软件存在缺陷吗？是的，有以下典故为证。

编程大师说：“任何一个程序，无论它多么小，总存在着错误。”

初学者不相信大师的话，他问：“如果有个程序小得只执行一个简单的功能，那会怎么样？”

“这样的程序没有意义，”大师说，“但如果这样的程序存在的话，操作系统最后将失效，产生错误。”

但初学者不满足，他问：“如果操作系统不失效，那会怎么样？”

“没有不失效的操作系统，”大师说，“但如果这样的操作系统存在的话，硬件最后将失效，产生错误。”

初学者仍不满足，再问：“如果硬件也不失效，那会怎么样？”

大师长叹一声道：“没有不失效的硬件。但如果这样的硬件存在的话，用户就会想让那个程序做一件不同的事，这件事也是错误。”

没有错误的程序世间难求。[摘自《编程之道》]

错误是严重的缺陷。医生犯的错误最终会被埋葬在地下，从此一了百了。但软件的错误不会自动消失，它会一直骚扰用户。据统计，对于大多数的软件产品而言，用于测试与改错的工作量将占整个软件开发周期的 30%，这是巨大的浪费。如果不懂得如何有效地提高软件质量，项目会付出很高的代价，你（开发人员）不仅没有功劳，也没人欣赏你的苦劳，你拥有最多的将只是疲劳。

怎样才能提高软件的质量呢？

先听听中国郎中治病的故事吧。

在古代中国，有一家三兄弟全是郎中。其中有一人是名医，人们问他：“你们兄弟三人谁的医术最高？”

他回答说：“我常用猛药给病危者医治，偶尔有些病危者被我救活，于是我的医术远近闻名。我二哥通常在人们刚刚生病的时候马上就治愈他们，临近村庄的人都知道他的医术。我大哥深知人们生病的原因，所以能够防止家里人生病，他的医术只有我们家里才知道。”

提高软件质量的基本手段是消除软件缺陷。与上述三个郎中治病很相似，消除软件缺陷也有三种基本方式：

（1） 在开发过程中有效地防止工作成果产生缺陷，将高质量内建于开发过程之中。这无疑是最佳方式，但是要求开发人员必须懂得正确地做事情（台阶比较高）。我

们学习“高质量编程”的目的就是要在干活的时候一次性编写出高质量的程序，而不是当程序出错后再去修补。

（2） 当工作成果刚刚产生时马上进行质量检查，及时找出并消除工作成果中的缺陷。这种方式效果比较好，人们一般都能学会。最常用的方法是技术评审、测试和质量保证（详见本章 1.5 节），这些方法已经被企业广泛采用，并取得了成效。

（3） 当软件交付给用户后，用着用着出错了，赶紧请开发者来补救，这种方式的代价最高。可笑的是，当软件系统在用户那里出故障了，那些现场补救成功的人倒成了英雄，好心用户甚至还寄来感谢信。

1.2.3 “零缺陷”理念

质量的最高境界是什么？是尽善尽美，即“零缺陷”。

“零缺陷”理念来源于国际上一些著名的硬件厂商。尽管软件的开发与硬件生产有很大的区别，但我们仍可以借鉴，从中得到启迪。

人在做一件事情时，由于存在很多不确定的因素，一般不可能 100% 地达到目标。假设平常人做事能完成目标的 80%。如果某个人的目标是 100 分，那么他最终成绩可达 80 分；如果某个人的目标只是 60 分，那么他最终成绩只有 48 分。我们在考场上身经百战，很清楚那些只想混及格的学生通常都不会及格。即使学习好的学生也常有失误，因而捶胸顿足。

做一个项目通常需要多个人的协作。假设某系统的总质量是十个开发人员的工作质量之积，记最高值为 1.0，最低值为 0。如果每个人的质量目标是 0.95，那么十个人的累积质量不会超过 0.598。如果每个人的质量目标是 0.9，那么十个人的累积质量不会超过 0.35。只有每个人都做到 1.0，系统总质量才会是 1.0。只要其中一人的工作质量是 0，那么系统总质量也成了 0。因系统之中的一个缺陷而导致机毁人亡的事件已不罕见。

上述比喻虽然严厉了一些，但从严要求只有好处没有坏处。如果不严以律已，人的堕落就很快。如果没有“零缺陷”的质量理念，也许缺陷就会成堆。

从理念到行动还是有一定距离的，企业在开发产品时应当根据自身实力和用户的期望值来设定可以实现的质量目标。过低的质量目标会毁坏企业的声誉，而过高的质量目标也有可能导致成本过高而拖累企业（请参见本章 1.3 节）。

1.2 细说软件质量属性

1.2.1 正确性

正确性是指软件按照需求正确执行任务的能力。这里“正确性”的语义涵盖了“精确性”。正确性无疑是第一重要的软件质量属性。如果软件运行不正确，将会给用户造成不便甚至造成损失。技术评审和测试的第一关都是检查工作成果的正确性。

正确性说起来容易做起来难。因为从“需求开发”到“系统设计”再到“编程”，任

何一个环节出现差错都会降低正确性。机器不会主动欺骗人，软件运行出错通常都是人造成的，所以不要找~~接口~~借口埋怨机器有毛病。开发任何软件，开发者都要为“正确”两字竭尽全力。

1.2.2 健壮性

健壮性是指在异常情况下，软件能够正常运行的能力。正确性与健壮性的区别是：前者描述软件在需求范围之内的行为，而后者描述软件在需求范围之外的行为。可是正常情况与异常情况并不容易区分，开发者往往把异常情况错当成正常情况而不作处理，结果降低了健壮性。用户才不管正确性与健壮性的区别，反正软件出了差错都是开发方的错。所以提高软件的健壮性也是开发者的义务。

健壮性有两层含义：一是容错能力，二是恢复能力。

容错是指发生异常情况时系统不出错误的能力，对于应用于航空航天、武器、金融等领域的这类高风险系统，容错性设计非常重要。

容错是非常健壮的意思，比如 Unix 的容错能力很强，很难搞死它。而恢复则是指软件发生错误后（不论死活）重新运行时，能否恢复到没有发生错误前的状态的能力。

从语义上理解，恢复不及容错那么健壮。例如某人挨了坏蛋一顿拳脚，特别健壮的人一点事都没有，表示有容错能力；比较健壮人，虽然被打到在地，过了一会还能爬起来，除了皮肉之痛外倒也不用去医院，表示恢复能力比较强；而虚弱的人可能在短期恢复不过来，得在病床上躺很久。

恢复能力是很有价值的。Microsoft 公司早期的窗口系统如 Windows 3.x 和 Windows 9x，动不动就死机，其容错性的确比较差。但它们的恢复能力还不错，机器重新启动后一般都能正常运行，看在这个份上，人们也愿意将就着用。

1.2.3 可靠性

可靠性是指在一定的环境下，在给定的时间内，系统不发生故障的概率。可靠性本来是硬件领域的术语。比如某个电子设备在刚开始工作时挺好的，但由于器件在工作中其物理性质会发生变化（如发热），慢慢地系统的功能或性能就会失常。所以一个从设计到生产完全正确的硬件系统，在工作中未必就是可靠的。

软件在运行时不会发生物理性质的变化，人们常以为如果软件的某个功能是正确的，那么它一辈子都是正确的。可是我们无法对软件进行彻底地测试，无法根除软件中潜在的错误。平时软件运行得好好的，说不准哪一天就不正常了，如有千年等一回的“千年虫”问题，司空见惯的“内存泄露”问题、“误差累积”问题等等。因此把可靠性引入软件领域是有意义的。

软件可靠性分析通常采用统计技术，遗憾的是目前可供第一线开发人员使用的成果很少见，大多数文章限于理论研究。我曾买了一本关于软件可靠性的著作，此书充满了数学公式。我实在难以看懂，更不知道怎样应用。请宽恕我的愚昧，我把此“天书”给

“供养”起来，没敢用笔画一处记号。

口语中的可靠性含义宽泛，几乎把正确性、健壮性全部囊括。只要人们发现系统有毛病，便归结为可靠性差。从专业上讲，这种说法是不对的，可是我们并不能要求所有的人都准确地把握质量属性的含义。

1.2.4 性能

性能通常是指软件的“时间-空间”效率，而不仅是指软件的运行速度。人们总希望软件的运行速度高些，并且占用资源少些。旧社会地主就是这么对待长工的：干活要快点，吃得要少点。

程序员可以通过优化数据结构、算法和代码来提高软件的性能。算法复杂度分析是很好的方法，可以达到“未卜先知”的功效。

性能优化的关键工作是找出限制性能的“瓶颈”，不要在无关痛痒的地方瞎忙乎。在大学里当教师，光靠使劲讲课或者埋头做实验，职称是升不快的。有些人找到了突破口，一年之内“造”它几十篇文章，争取破格升副教授、教授。在学术上走捷径，这类“学者”的质量真让人担忧。

1.2.5 易用性

易用性是指用户使用软件的容易程度。现代人的生活节奏快，干啥事都想图个方便。所以把易用性作为重要的质量属性对待无可非议。

导致软件易用性差的根本原因是开发人员犯了“错位”的毛病：他以为只要自己用起来方便，用户也一定会满意。俗话说“王婆卖瓜，自卖自夸”。当开发人员向用户展示软件时，常会得意地讲：“这个软件非常好用，我操作给你看，……是很好用吧！”

软件的易用性要让用户来评价。如果用户觉得软件很难用，开发人员不要有逆反心里：到哪里找来的笨蛋！

其实不是用户笨，是自己开发的软件太笨了。当用户真的感到软件很好用时，一股温暖的感觉油然而生，于是就用“界面友好”、“方便易用”等词来评价易用性。

1.2.6 清晰性

清晰意味着工作成果易读、易理解，这个质量属性表达了人们一种质朴的愿望：让我化钱买它或者用它，总得让我看明白它是什么东西。我小时候的一个伙伴在读中学时，就因搞不明白电荷为什么要分“正”和“负”，觉得很烦恼，便早早地辍学当了工人。

开发人员只有在自己思路清晰的时候才可能写出让别人易读、易理解的程序和文档。可理解的东西通常是简洁的。一个原始问题可能很复杂，但高水平的人就能够把软件系统设计得很简洁。如果软件系统臃肿不堪，它迟早会出问题。所以简洁是人们对工作“精益求精”的结果，而不是潦草应付的结果。

在生活中，与简洁对立的是“罗里罗嗦”。废话大师有句名言：“如果我令你过于轻松地明白了，那你一定是误解了我的意思。”中国小说中最“婆婆妈妈”的男人是唐僧。有一项民意调查：如果世上只有唐僧、孙悟空、猪八戒和沙僧这四类男人，你要嫁给哪一类？请列出优先级。调查结果表明，现代女性毫不例外地把唐僧摆在老末。

很多人在读研究生时有种奇怪的体会：如果把文章写得很简洁，让人很容易理解，投稿往往中不了；只有加上一些玄乎的东西，把本来简单的弄成复杂的，才会增加投稿的命中率。虽然靠这种做法能把文凭混到手，可千万不要把此“经验”应用到产品的开发中！

1.2.7 安全性

这里安全性是指信息安全，英文是 Security 而不是 Safety。安全性是指防止系统被非法入侵的能力，既属于技术问题又属于管理问题。信息安全是一门比较深奥的学问，其发展是建立在正义与邪恶的斗争之上。这世界似乎不存在绝对安全的系统，连美国军方的系统都频频被黑客入侵。如今全球黑客泛滥，真是“道高一尺，魔高一丈”啊。

对于大多数软件产品而言，杜绝非法入侵既不可能也没有必要。因为开发商和客户愿意为提高安全性而投入的资金是有限的，他们要考虑值不值得。究竟什么样的安全性是令人满意的呢？

一般地，如果黑客为非法入侵花费的代价（考虑时间、费用、风险等因素）高于得到的好处，那么这样的系统可以认为是安全的。

1.2.8 可扩展性

可扩展性反映软件适应“变化”的能力。在软件开发过程中，“变化”是司空见惯的事情，如需求、设计的变化，算法的改进，程序的变化等等。

由于软件是“软”的，是否它天生就容易修改以适应“变化”？

关键要看软件的规模和复杂性。

如果软件规模很小，问题很简单，那么修改起来的确比较容易，这时就无所谓“可扩展性”了。要是软件的代码只有 100 行，那么“软件工程”也就用不着了。

如果软件规模很大，问题很复杂，倘若软件的可扩展性不好，那么该软件就象用卡片造成的房子，抽出或者塞进去一张卡片都有可能使房子倒塌。可扩展性是系统设计阶段重点考虑的质量属性。

1.2.9 兼容性

兼容性是指两个或两个以上的软件相互交换信息的能力。由于软件不是在“真空”里应用的，它需要具备与其它软件交互信息的能力。例如两个字处理软件的文件格式兼容，那么它们都可以操作对方的文件，这种能力对用户很有好处。国内金山公司开发的

字处理软件 WPS 就可以操作 Word 文件。

开发某领域的新软件，应尽量与已有的流行的软件相兼容，否则难以被市场接受。

1.2.10 可移植性

可移植性是指软件运行于不同软硬件环境的能力。编程语言越低级，其程序越难移植，反之则容易。例如 C 程序比汇编程序的可移植性好。而 Java 程序则号称“一次编程，到处运行”，具有 100%的可移植性。

软件设计时应该将“设备相关程序”与“设备无关程序”分开，将“功能模块”与“用户界面”分开。这样可以提高可移植性。

1.3 人们关注的不仅是质量

企业开发产品的目的是赚钱，为了使利润极大化，人们希望软件开发工作“做得好、做得快并且少花钱”。用软件工程的术语来讲，即“提高质量、提高生产率并且降低成本”。古代哲学家曾为“鱼和熊掌不可得兼”的问题费尽心思，我们现在却梦想鱼、熊掌、美酒三者兼得，现代人的欲望真是无止境啊。

让我们先谈谈质量、生产率和成本之间的关系。

1.3.1 质量、生产率和成本之间的关系

质量无疑是客户最关心的问题。客户即使不图物美价廉，也要求货真价实。软件开发商必须满足客户对质量的要求（不论是写在合同上的还是约定俗成的），否则做不成买卖。现在就连做盗版光盘生意的人也讲究质量，如果盘片不好，是可以退货的。高质量既是软件开发人员的技术追求，又是职业道德的要求。

在关注质量的同时，软件开发商又期望生产率能高些并且成本能低些。老板和员工们谁不想用更少的时间赚更多的钱！

质量与生产率之间存在相辅相成的关系。高生产率必须以质量合格为前提。如果质量不合格，软件产品要么卖不出去，要么卖出去了再赔偿客户的损失。这种情况下“高生产率”变得毫无意义。别看开发商和客户双方的代表能在餐桌上谈笑风生，一旦出了质量问题，那就不会很亲热了。从短期效益看，追求高质量可能会延长软件开发时间，一定程度上降低了生产率。从长期效益看，追求高质量将使软件开发过程更加成熟和规范化。日积月累，当开发过程成熟到一定地步后，必将大大降低软件的测试和改错的代价，缩短产品的开发周期，实质上是提高了生产率，同时又获得了很好的信誉。所以质量与生产率之间不存在根本的对立。

提高质量与生产率需要一个过程，企业不可操之过切。一般地，**软件过程能力比较低的企业（例如低于 CMM 2 级），应该将质量放在第一位，生产率放在第二位。只有这**

样才可能持久地提高质量与生产率。（“能力成熟度模型 CMM”将在后面解释）

如果一个企业的软件过程能力低于 CMM 2 级，表明其开发能力与管理能力还很薄弱。就其目前的实力而言，无论下多大的决心去做，都不可能一开始就把质量与生产率改善得一样好。并不是我们刻意贬低生产率的“地位”，是公司的现实情况要求在质量与生产率之间分个“轻重缓急”。由于人们天生就有“急功近利”的倾向，如果公司领导人认可“生产率第一、质量第二”，那么员工们做着做着必定会回到混乱的局面。这样的教训实在是太多了！老话说得好：磨刀不误砍柴功。用它类比上述理念最合适不过了。

俗话说“一分价钱一分货”，人们买东西的时候大多认可“质量越好价格就越高”。除了垄断性的产品外，一般来说成本是影响价格的主要因素。

对于软件开发而言，质量与成本之间有什么关系？高质量必然会导致高成本吗？

经验表明，如果软件的“高质量”是“修补”出来的，毫无疑问会导致低生产率和高成本。如果能研制出某些好方法，将高质量与高生产率内建于开发过程之中，那么就能自然地能降低开发成本，这是软件过程改进的目标。

要提醒大家的是，大公司与小公司对成本的关注程度是不尽相似的。

首先谈一下“市场价”（Marketing Price）与“成本价”（Cost Price）的概念。在某个领域，当市场上只出现尚未形成竞争格局的一个或几个产品时，产品价格基本上是由厂商自己制定，称为“市场价”。由于缺乏竞争，无论成本多高，总能获得高额利润。电影《大腕》里那个搞房地产的精神病人说“不求最好，但求最贵”，真是实话实说。

当产品之间形成竞争时，就会出现“杀价”现象。由于各家产品的功能、质量旗鼓相当，竞争实质上是在拼成本。谁的成本低，谁就有利可图。这时的产品价格叫做“成本价”。

中国的彩电业是一个活生生的例子。若干年前彩电价格极高，彩电远离百姓人家，一部分人即使买得起也买不到。如今连超市里都充斥着各种品牌的彩电，价格战打得呜呼哀哉，把厂商逼到“微利”的地步。现在工薪阶层人士很少有买不起彩电的。商场里 TOTO 品牌的马桶价格为 2000～3000 元，比同体积的国产纯平彩电还贵，并且利润高得多。唉，我们坐在这样的马桶上真的要为民族工业忧心忡忡哪。

由于“市场价”与“成本价”的差价十分悬殊，IT 行业的大公司都想吃“市场价”这块肥肉。大公司的资金雄厚，销售力量强，只要能抢先推出产品，就不愁卖不出去。怎样才能达到目的呢？通常有两种方式。

一种方式是从别处购买快要成形的产品，改头换面，贴上大公司的标签就可以上市销售。所以 IT 行业的“公司收购”特别盛行。如果 Cisco 公司的网络产品全部让原班人马来开发，它很难能够那么快就发展成为网络业的霸主！

另一种方式是自行开发新产品，让公司的研发队伍加班加点地干活。这么辛苦是值得的，产品成功会让员工们有很大的成就感。

无论通过哪种方式抢先推出产品，前提条件都要求产品的质量合格。如果产品因质量不合格而被市场拒绝，那么损失的不仅仅是成本，更惨重的是失去机会和信誉。象 Intel 这样了不起的公司也会吃败仗。每当 Intel 公司的 CPU 芯片出现缺陷时，就会骂声一片。Intel 公司不得不大量回收芯片并向用户道歉，此时竞争对手如 AMD 公司就会乘虚而入，

抢走象 IBM、Compaq 这些大客户的部分定单。

在信息高度发达的社会里，你能想得到的产品别人也能想得到。只有少数大公司能够享受到“市场价”的利益，但是好景不会太长。大多数公司在大部分时间里开发的是“成本价”的产品。所以树立“降低开发成本”的理念仍然十分重要。

1.3.2 软件过程改进基本概念

在 20 世纪 70、80 年代，软件工程的研究重点是需求分析、系统设计、编程、测试、维护等领域的方法、技术和工具，我们称之为经典软件工程。

现代的软件技术、软件开发工具比十年前好不知多少，而且几乎所有的开发人员都学习过分析、设计、编程、测试等技能，可是如今绝大多数软件项目依然面临着质量低下、进度延误、费用超支这些老问题。

人们逐渐意识到，由于机构管理软件过程的能力比较弱，常常导致项目处于混乱状态，过程的混乱使得新技术、新工具的优势难以体现。经典的软件工程不是不好，而是不够用。从 20 世纪 90 年代至今，软件过程改进成为软件工程学科的一个主流研究方向，其中 CMM 和 CMMI 是该领域举世瞩目的重大成果。

让我们首先解释什么是过程。过程就是人们使用相应的方法、规程、技术、工具等将原始材料（输入）转化成用户需要的产品。过程的 3 个基本要素是：人、方法与规程、技术与工具，如图 1-1 所示。可以把过程比喻为 3 条腿的桌子，要使桌子平稳，这 3 条腿必须协调好。

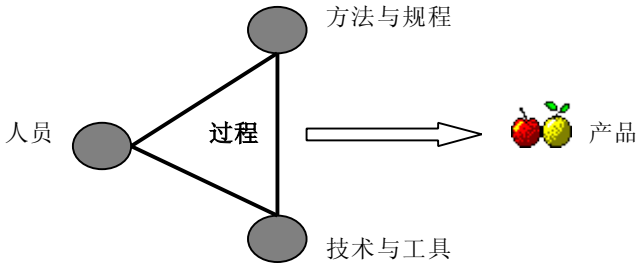

图 1-1 过程的要素

从图 1-1 可知，过程与产品存在因果关系。即好的过程才能得到好的产品，而差的过程只会得到差的产品。这个道理很朴实，但是很多人并未理解或者理解了却不实行。毕竟我们销售的是产品，而非过程。人们常常只把眼光盯在产品上，而忘了过程的重要性。例如，领导对员工们下达命令时总强调：“我不管你们如何做，反正时间一到，你们就得交付产品。”其实这是一句因果关系颠倒了的话，却在业界普遍存在。

在过程混乱的企业里，一批人马累死累活地做完产品后，马上又因质量问题被折腾得焦头烂额。这种现象反反复复地发生，让人疲惫不堪。怎么办？长痛不如短痛，应该下决心、舍得花精力与金钱去改进软件过程能力。

CMM（Capability Maturity Model）是用于衡量软件过程能力的事实上的标准，同时

也是目前软件过程改进最好的参考标准。CMM是由美国卡内基-梅隆大学（Carnegie-Mellon）软件工程研究所（Software Engineering Institute, SEI）研制的，其发展简史如下：

- ✧ CMM 1.0 于 1991 年制定。
- ✧ CMM 1.1 于 1993 发布，该版本应用最广泛。
- ✧ CMM 2.0 草案于 1997 年制定（未广泛应用）。
- ✧ 到 2000 年，CMM 演化成为 CMMI（Capability Maturity Model Integration），CMM 2.0 成为 CMMI 1.0 的主要组成部分。
- ✧ CMMI-SE/SW 1.1（CMMI for System Engineering and Software Engineering）于 2002 年 1 月正式推出。

CMM 将软件过程能力分为 5 个级别，最低为 1 级，最高为 5 级。目前国内只有几家 IT 企业达到了 CMM 2 级或 CMM 3 级。鉴于 CMM 已经被美国、印度软件业广为采纳，并且取得了卓著成效，近两年来国内兴起了 CMM 热潮。CMM 受欢迎的程度远远超过了 ISO 同类标准。

国内IT企业采用CMM的目的大体有两种：

（1）主要想提高企业的软件过程能力，但并不关心CMM评估。

（2）既要提高企业的软件过程能力，又想通过CMM评估来提升企业的威望与知名度。

出于第一种考虑的企业占绝大多数，它们主要是一些中小型IT企业。出于第二种考虑的一般是实力雄厚的大型IT企业。无论是哪类IT企业，它们在实施CMM时遇到的共性问题是“费用高、难度大、见效慢”。

企业做一次比较完整的CMM 2-3级咨询和评估大约要花费60~100万元。然而CMM咨询师只能起到“参谋”的作用，解决实际问题还得靠自己。企业要组建软件工程过程小组（Software Engineering Process Group, SEPG）专门从事CMM研究与推广工作，SEPG的成本并不比咨询费低。如果企业再购买一些昂贵的软件工程工具（例如Rational的产品），那么总成本会更高。

即使企业舍得花钱，也不意味着就能够容易地提高软件过程能力。目前国内通过CMM 2-3级评估的企业屈指可数，而这些企业的实际能力也没有宣传的那么好。因为参加CMM评估的项目都是精心准备的，个别项目或者事业部通过了CMM评估并不意味着整个企业达到了那个水平，这里面的水分相当大。

曾经有一段时间，IT人士经常争论“CMM好不好”、“值不值得推广CMM”等话题。现在业界关注的焦点则是“企业如何以比较低的代价有效地提高软件过程能力”，攻克这个难题必将产生巨大的经济效益和社会效益。

一般地，为了真正提高软件过程能力，企业至少要做三件最重要的事情：

（1）首先制定适合于本企业的软件过程规范。

（2）对员工们进行培训，指导他们依据规范来开发产品。

（3）购买或者开发一些软件工程和项目管理工具，提高员工们的工作效率 。

本书作者及其合作者根据上述需求，研制了一套“软件过程改进解决方案'（Software Process Improvement Solution, SPIS）。SPIS的主要组成部分有：

- ✧ 基于CMMI 3级的软件过程改进方法与规范。
- ✧ 一系列培训教材，包括软件工程、项目管理、高质量编程等。本书《高质量C++/C编程指南》即为其中之一。

✧ 基于 Web 的项目管理工具，包括项目计划、项目监控、质量管理、配置管理、需求管理等功能，命名为 Future。

1.4 高质量软件开发的基本方法

1.4.1 建立软件过程规范

人们意识到，若想顺利开发出高质量的软件产品，必须有条理地组织技术开发活动和项目管理活动。我们把这些活动的组织形式称为过程模型。软件企业应当根据产品的特征，建立一整套在企业范围内通用的软件过程模型及规范，并形成制度。这样开发人员与管理人员就可以依照过程规范有条不紊地开展工作。

我们曾与国内很多研发人员和各级经理交流过，大家都对软件开发的混乱局面表示了不满和无奈。尽管“土匪游击队”的开发模式到处可见，但是没有人真的喜欢混乱。“规范化”是区别“正规军”和“土匪游击队”的根本标志。大家无不渴望以规范化的方式开发产品。这是现状、是需求、也是希望。

对软件开发模型的研究兴起于 60 年代末 70 年代初，典型成果是 1970 年提出的瀑布模型。人们研制了很多的软件开发模型，常见的有“瀑布模型”、“喷泉模型”、“增量模型”，“快速原型模型”、“螺旋模型”、“迭代模型”等。

这么多软件开发模型，企业应该如何选择并应用呢？

企业在选择软件开发模型时，不要太在乎学术上的“先进”与“落后”，正如有才华的人并不一定要出自名牌大学或拥有高学历那样。关键是看该模型能否有效地帮助企业顺利地开发出软件产品，并且要考虑员工们使用起来是否方便。简而言之，就是考察模型是否“实用、好用”。

最早出现的软件开发模型是瀑布模型。它太理想化、太单纯，看起来已经落后于现代的软件开发模式。如今瀑布模型几乎被学术界抛弃，偶而被人提起，都属于被贬对象，未被留一丝惋惜。说它如何如何地差，为的是说明新模型是怎样怎样地好。

然而企业界不同于学术界，我认为瀑布模型对企业太有价值了，我要为它声辩，恢复它应有的名誉。瀑布模型的精髓是“线性顺序”地开发软件。我们应该认识到“线性化”是人们最容易掌握并能熟练应用的思想方法。当人们碰到一个复杂的“非线性”问题时，总是千方百计地将其分解或转化为一系列简单的线性问题，然后逐个解决。一个软件系统的整体可能是复杂的，而细分后的子程序总是简单的，可以用“线性化”的方式来实现，否则干活就太累了。

让我们引用 Albert Einstein 的话作为信条——“任何事物都应该尽可能地简洁”，“线性”是一种简洁，简洁就是美。当我们领会了“线性”的精神，就不要再呆板地套用“线性”的外表，而应该用活它。例如增量模型实质就是分段的线性模型。螺旋模型则是迭代的弯曲了的线性模型。在其它模型中大都能够找到“线性”的影子。

瀑布模型是如此的简洁，所有的软件开发人员天生就能学会（如果学不会，那他就别干软件这一行了）。所以瀑布模型特别适合于企业，请大家别轻易贬低它。

软件开发模型只关注技术开发活动，并不考虑项目管理，这对开发产品而言是不够的，所以开发模型只是软件过程模型的一部分。奇怪的是，我迄今为止尚未找到论述软件过程模型的软件工程书籍。我就自己创作了一个基于 CMMI 3 级的软件过程模型，称为“精简并行过程”（Simplified Parallel Process, SPP）。

SPP 模型如图 1-2 所示。“精简并行过程”的含义是：

（1）对 CMMI 3 级以内的关键过程域以及关键实践作了“精简”处理；

（2）项目管理过程、技术开发过程和支撑过程“并行”开展。

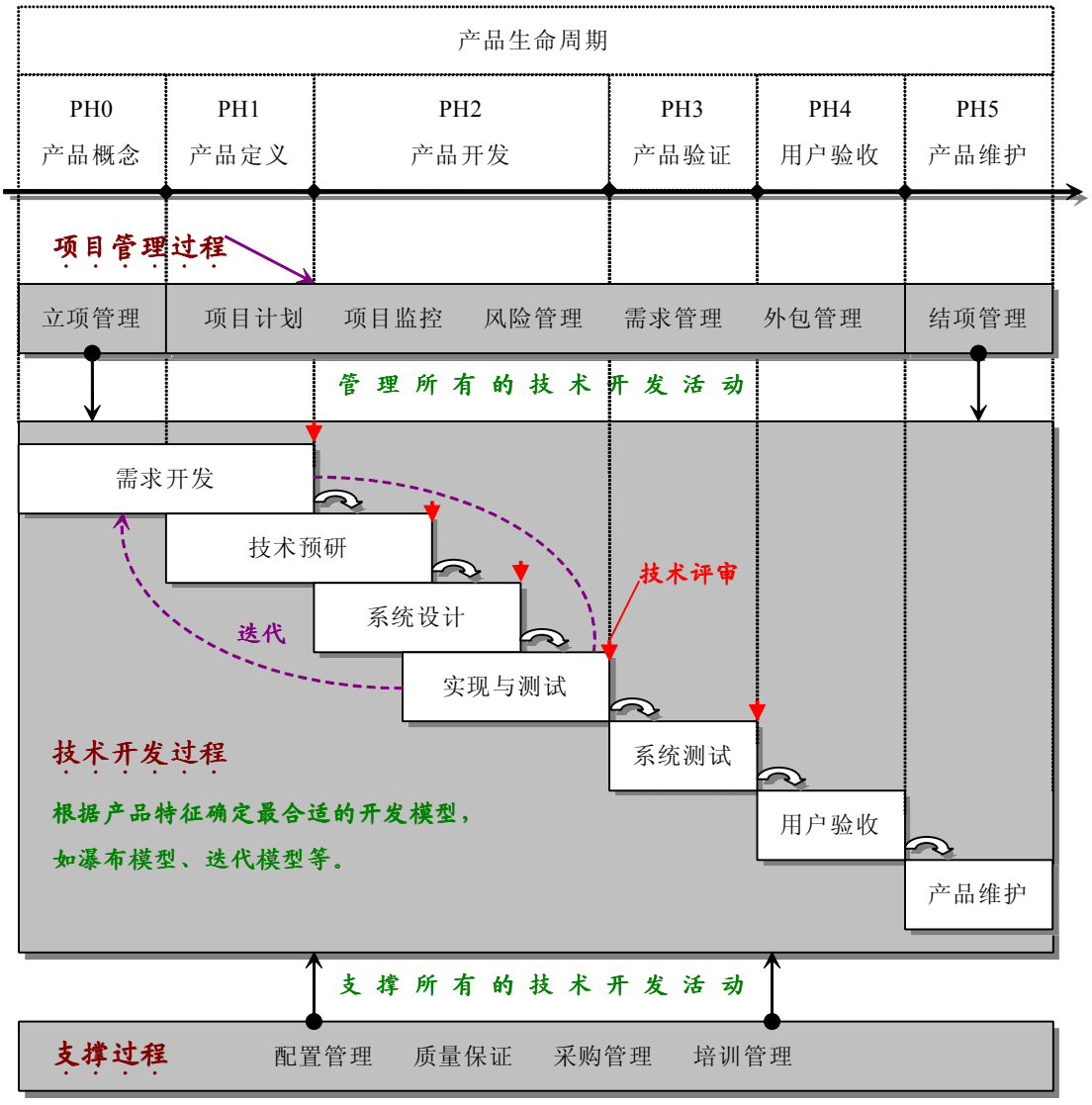

图 1-2 精简并行过程（SPP）模型

SPP 模型把产品生命周期划分为 6 个阶段：

- ✧ 产品概念阶段，记为 PH0。
- ✧ 产品定义阶段，记为 PH1。
- ✧ 产品开发阶段，记为 PH2。

✧ 产品验证阶段，记为 PH3。

✧ 用户验收阶段，记为 PH4。

✧ 产品维护阶段，记为 PH5。

在 SPP 模型中，一个项目从 PH0 到 PH5 共经历 19 个关键过程域（Key Process Area, KPA），它们被划分为三大类过程，如表 1-2 所示。其中项目管理过程含 7 个关键过程域，技术开发过程含 8 个过程域，支撑过程含 4 个过程域。

过程类别	● 项目管理过程	● 技术开发过程	● 支撑过程
关键过程域	✧ 立项管理 ✧ 结项管理 ✧ 项目计划 ✧ 项目监控 ✧ 风险管理 ✧ 外包管理 ✧ 需求管理	✧ 需求开发 ✧ 技术预研 ✧ 系统设计 ✧ 实现与测试 ✧ 系统测试 ✧ 用户验收 ✧ 产品维护 ✧ 技术评审	✧ 配置管理 ✧ 质量保证 ✧ 采购管理 ✧ 培训管理

表 1-2 SPP 过程域分类

SPP 模型的主要优点有：

（1）模型直观。SPP 模型是三层结构，上层是项目管理过程的集合，中层是技术开发过程的集合，下层是支撑过程的集合。这种模型很直观，高级经理、项目经理、开发人员、质量保证员等人根据 SPP 模型很容易知道自己“应该在什么时候做什么事情，以及按照什么规范去做事情”。SPP 模型有助于使各个过程的活动有条不紊地开展。

（2）方便于用户裁剪 SPP 模型。项目管理过程和支撑过程对绝大多数软件产品开发而言都是适用的。需求开发、技术预研、系统设计、编程、测试、技术评审、维护都是技术开发过程中必不可少的环节，用户可以根据产品的特征确定最合适的开发模型（例如瀑布模型、快速原型模型、迭代模型等）。

（3）方便于用户扩充 SPP 模型。如果产品同时涉及软件硬件开发的话，可将产品生命周期、软件开发过程和硬件开发过程集成一起。

1.4.2 复用

复用就是指“利用现成的东西”，文人称之为“拿来主义”。被复用的对象可以是有形的物体，也可以是无形的知识成果。复用不是人类懒惰的表现而是智慧的表现。因为人类总是在继承了前人的成果，不断加以利用、改进或创新后才会进步。所以每当我们欢度国庆时，要清楚祖国远不止 50 来岁，我们今天享用到的财富还有历史上几千年中国人民的贡献。进步只是应该的，没有进步则就可耻了。

复用的有利于提高质量、提高生产率和降低成本。由经验可知，通常在一个新系统中，大部分的内容是成熟的，只有小部分内容是创新的。一般地可以相信成熟的东西总

是比较可靠的（即具有高质量），而大量成熟的工作可以通过复用来快速实现（即具有高生产率）。勤劳并且聪明的人们应该把大部分的时间用在小比例的创新工作上，而把小部分的时间用在大比例的成熟工作中，这样才能把工作做得又快又好。

把复用的思想用于软件开发，称为软件复用。技术开发活动与管理活动中的任何成果都可以被复用，如思想方法、经验、程序、文档等等。据统计，世上已有 1000 亿多行程序，无数功能被重写了成千上万次，真是浪费哪。面向对象（Object Oriented）学者的口头禅就是“请不要再发明相同的车轮子了” 。

将具有一定集成度并可以重复使用的软件组成单元称为软构件(Software Component)。软件复用可以表述为：构造新的软件系统可以不必每次从零做起，直接使用已有的软构件，即可组装或加以合理修改后成为新的系统。

复用方法合理化并简化了软件开发过程，减少了总的开发工作量与维护代价，既降低了软件的成本又提高了生产率。另一方面，由于软构件是经过反复使用验证的，自身具有较高的质量。因此由软构件组成的新系统也具有较高的质量。

软件复用不仅要使自己拿来方便，还要让别人拿去方便，是“拿来拿去主义”。这想法挺好，但现实中执行得并不如意。企业的业务各色各样，谁也不能坐等着天上掉下可以被大规模复用的东西，一般要靠“日积月累”才能建设可以被复用的软件库。从理论上讲这项工作没有不可逾越的技术障碍。真正的障碍是它“耗时费钱”，前期投入较多，缺乏近期效益。大部分的公司都注重近期效益，不是它天生目光短浅，而是为了生存必须这么做。有些处境艰难的公司下一顿饭还不知道在何处着落，更别提软件复用这样的“长久之计”了。所以软件复用对大多数公司来说不是“最高优先级”。

我到公司工作的最初安排是从事电信领域“可复用软件工厂”的开发。领导在招聘时跟我讲，这个想法已经有数年了，一直没有落实，希望我能做好。待我正式上班时，马上就改成做其它短期的研发项目了。要知道我所在的公司人力与财力相当充足，非国内普通中小型 IT 企业所能比。即便我们有如此好的条件，软件复用也只是挂在嘴上，没有实际行动。

所以我建议：随时随地尽可能地复用你所能复用的东西，不要等待公司下达复用的行政命令，因为你很难等到那一天，即使等到了也没有多少意义。

1.4.3 分而治之

分而治之是指把一个复杂的问题分解成若干个简单的问题，然后逐个解决。这种朴素的思想来源于人们生活与工作的经验，完全适合于技术领域。

分而治之说起来容易，做好却难，最糟糕的现象是“分是分了”却“治不了”。

软件的分而治之不可以“硬分硬治”。不像为了吃一个西瓜或是一只鸡，挥刀斩成 n 块，再把每块塞进嘴里粉碎搅拌，然后交由胃肠来消化吸收，象征复杂问题的西瓜或是鸡也就此消失了。

软件的“分而治之”应该着重考虑：复杂问题分解后，每个问题能否用程序实现？所有程序最终能否集成为一个软件系统并有效解决原始的复杂问题？图 1-3 表示了软件的“分而治之”策略。软件的模块化设计就是分而治之的具体表现。

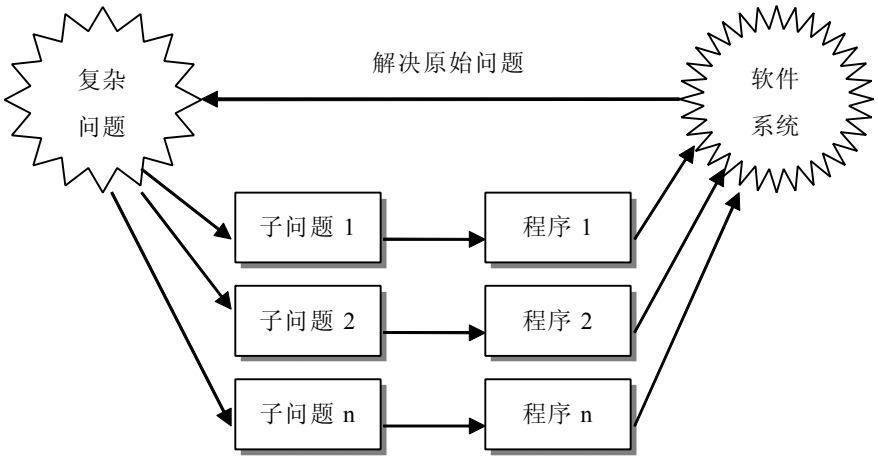

图 1-3 软件的分而治之策略

1.4.4 优化与折衷

软件的优化是指优化软件的各个质量属性，如提高运行速度，提高对内存资源的利用率，使用户界面更加友好，使三维图形的真实感更强等等。想做好优化工作，首先要让开发人员都有正确的认识：**优化工作不是可有可无的事情，而是必须要做的事情。**

当优化工作成为一种责任时，开发人员才会不断改进软件中的数据结构、算法和程序，从而提高软件质量。

著名的 3D 游戏软件 Quake，能够在 PC 机上实时地绘制高度真实感的复杂场景。Quake 的开发者能把很多成熟的图形技术发挥到极致，例如把 Bresenham 画线、多边形裁剪、树遍历等算法的速度提高近一个数量级。我的博士研究方向是计算机图形学，我第一次看到 Quake 时不仅感到震动，而且深受打击。这个 PC 游戏软件的技术水平已经远胜于我所见识到的国内领先的图形学相关科研成果。这对国内日益盛行的点到完止的学术研究真是莫大的讽刺。

所以当我们开发出来的软件表现出很多不可救药的病症时，不要埋怨机器差。真的都是我们自己没有把工作做好，写不好字不要嫌笔钝。

假设我们经过思想教育后，精神抖擞，随时准备为优化工作干上六天七夜。但愿意做并不意味着就能把事情做好。优化工作的复杂之处是很多目标存在千丝万缕的关系，可谓数不清理还乱。当不能够使所有的目标都得到优化时，就需要“折衷”。

软件中的“折衷”是指协调各个质量属性，实现整体质量的最优。就象某些官僚扮演和事佬的角色：“…为了使整个组织具有最好的战斗力，我们要重用几个人，照顾一些人，在万不得已的情况下委屈一批人”。

软件折衷的重要原则是不能使某一方损失关键的功能，更不可以象“舍鱼而取熊掌”那样抛弃一方。例如 3D 动画软件的瓶颈通常是速度，但如果为了提高速度而在程序中取消光照明计算，那么场景就会丧失真实感，3D 动画也就不再有意义了（如果人类全是色盲的话，计算机图形学将会变得异常简单）。

人都有惰性，如果允许滥用折衷的话，那么一当碰到困难，人们就会用拆东墙补西

墙的方式去折衷，不再下苦功去做有意义的优化。所以我们有必要为折衷制定严正的立场：**在保证其它重要质量属性不差的前提下，使某些质量属性变得更好**。

下面让我们用优化与折衷的思想解决“鱼和熊掌不可得兼”的难题。

问题提出：假设鱼每千克 10 元，熊掌每千克一万元。有个倔脾气的人只有 20 元钱，非得要吃上一公斤美妙的“熊掌烧鱼”，怎么办？

解决方案：化 9 元 9 角 9 分钱买 999 克鱼肉，化 10 元钱买 1 克熊掌肉，可做一道“熊掌戏鱼”菜。剩下的那一分钱还可建立基金，用于推广优化与折衷方法。

1.4.5 技术评审

技术评审（Technical Review, TR）的目的是尽早地发现工作成果中的缺陷，并帮助开发人员及时消除缺陷，从而有效地提高产品的质量。

技术评审最初是由 IBM 公司为了提高软件质量和提高程序员生产率而倡导的。技术评审方法已经被业界广泛采用并收到了很好的效果，它被普遍认为是软件开发的最佳实践之一。技术评审能够在任何开发阶段执行，它可以比测试更早地发现并消除工作成果中的缺陷。技术评审的主要好处有：

- ✧ 通过消除工作成果的缺陷而提高产品的质量。
- ✧ 越早消除缺陷就越能降低开发成本。
- ✧ 开发人员能够及时地得到同行专家的帮助和指导，无疑会加深对工作成果的理解，更好地预防缺陷，一定程度上提高了开发生产率。

技术评审有两种基本类型：

- ✧ 正规技术评审（FTR）。FTR 比较严格，需要举行评审会议，参加评审会议的人员比较多。
- ✧ 非正规技术评审（ITR）。ITR 的形式比较灵活，通常在同伴之间开展，不必举行评审会议，评审人员比较少。

理论上讲，为了确保产品的质量，产品的所有工作成果都应当接受技术评审。现实中，为了节约时间，允许人们有选择地对工作成果进行技术评审。技术评审方式也视工作成果的重要性和复杂性而定。例如，将重要性、复杂性各分“高、中、低”三个等级，重要性－复杂性的组合与技术评审方式的对应关系见表 1-3。

工作成果的重要性－复杂性组合	技术评审方式（FTR, ITR）
高高	FTR
高中	FTR
高低	FTR 或者 ITR 均可
中中	FTR 或者 ITR 均可
中低	ITR
低低	ITR

表 1-3 工作成果重要性－复杂性组合与技术评审方式的对应关系

正规技术评审的一般流程如图 1-4 所示。

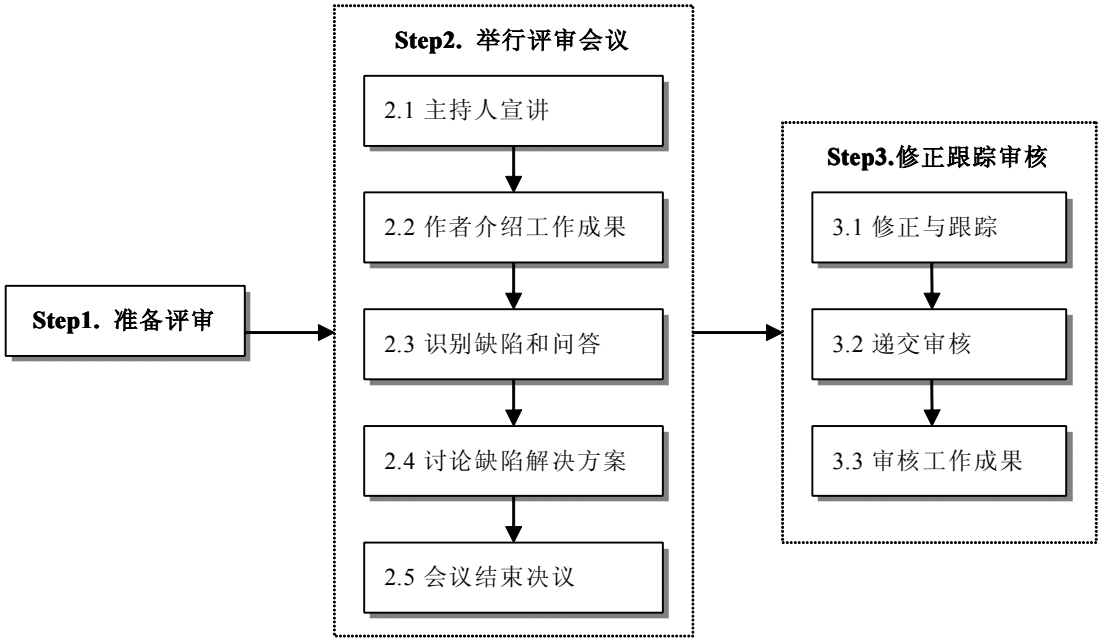

图 1-4 正规技术评审的一般流程

技术评审的注意事项：

- ✧ 评审人员的职责是发现工作成果中的缺陷，并帮助开发人员给出消除缺陷的办法，而不是替开发人员消除缺陷。
- ✧ 技术评审应当“就是论事”，不要打击有失误的开发人员的工作积极性，更不准搞人身攻击（如挖苦、讽刺等）。
- ✧ 在会议评审期间要限制过多的争论，以免浪费他人的时间。

对技术评审的一些建议：

- ✧ 对于重要性和复杂性都很高的工作成果，建议先在项目内部进行“非正规技术评审”，然后再进行“正规技术评审”。
- ✧ 技术评审应当与质量保证有机地结合起来，最好请质量保证人员参加并监督正规技术评审。
- ✧ 技术评审应当与配置管理有机地结合起来，例如规定工作成果成为基准（Baselined）之前必须先通过技术评审。
- ✧ 建议机构采用统一的缺陷跟踪工具，使得技术评审所发现的缺陷能被及时地消除，不被遗漏。

1.4.6 测试

测试是通过运行测试用例来找出软件中的缺陷。测试与技术评审的主要区别是前者要运行软件而后者不必运行软件。

在软件开发过程中，编程和测试是紧密相关的技术活动，缺一不可。理论上讲两者

不分贵贱，同等重要。但在大多数软件企业中，程序员的待遇普遍要高于专职的测试人员。即使不考虑待遇问题，大多数人认为开发工作比测试工作有乐趣、有成就感、有前途。所以计算机专业人员通常会把编程当成一种看家本领，舍得下功夫学习和专研，但极少有人以这种态度对待软件测试。这种意识导致软件测试被过于轻视。不仅学生们在读书时懒得学习测试（目前国内高校似乎没有“软件测试”的课程），就连有数年工作经验的软件开发人员也未必懂得测试。

不少开发人员虽然不敌视测试工作，但有“临时抱佛脚”的坏习惯，往往事到临头才到处找测试资料，向人请教。经常有人向我要文档模板，用来对付测试。

你以为有了文档模板就懂得如何测试了吗？

测试虽然并不深奥，但是学好并不容易。不懂得“有效”测试的项目小组往往面临这样的问题：计划中的时间很快就用完了，即使有迹象表明软件中还有不少缺陷，也只好草草收场，把麻烦留在将来。

测试的主要目的是为了发现尽可能多的缺陷。可是这个观念不容易被人接受。

正确理解测试的目的十分重要。如果认为测试的目的是为了说明软件中没有缺陷，那么测试人员就会向这个目标靠拢，因而下意识地选用一些不易暴露错误的测试示例。这样的测试是不真实的。如果为了说明软件有多么好，那么应当制作专门的演示。千万不要将“测试”与“演示”混为一谈。

看来测试并不单单是个技术问题，还是个职业道德问题。

根据测试的目的，可以得出一个推论：成功的测试在于发现了迄今尚未发现的缺陷。所以测试人员的职责是设计这样的测试用例，它能有效地揭示潜伏在软件里的缺陷。

如果测试工作很全面、很认真，但是的确没有发现新的缺陷。那么这样的测试是否毫无价值？

不，那不是测试的过失，应当反过来理解：由于软件的质量实在太好了，以致于这样的测试发现不了缺陷。

所以，如果产品通过了严格的测试，大家不要不吭气，应当好好地宣传一把，把测试的成本捞回一些。

软件测试的常规分类如表 1-4 所示，测试的一般流程如图 1-5 所示。如果对表 1-4 和图 1-5 的内容作深入论述，大约还需要几十页的篇幅，这显然超出了本章的范畴。这里就点到为止吧，请读者参考软件测试的有关文献。

测试的常规分类	
测试阶段	单元测试、集成测试、系统测试、验收测试
测试方式	白盒测试、黑盒测试
测试内容	功能测试、健壮性测试、性能测试、用户界面测试、安全性测试、压力测试、可靠性测试、安装/反安装测试，…

表 1-4 测试的常规分类

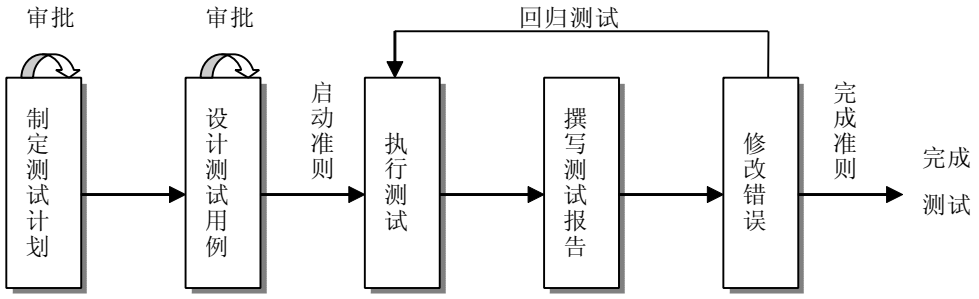

图 1-5 测试的一般流程

测试的经验之谈：

- ✧ 测试能提高软件的质量，但是提高质量不能依赖测试。
- ✧ 测试只能证明缺陷存在，不能证明缺陷不存在。“彻底地测试”难以成为现实，要考虑时间、费用等限制，不允许无休止地测试。我们应当祈祷：软件的缺陷在产品被淘汰之前一直没有机会发作。
- ✧ 测试的主要困难是不知道如何进行有效地测试，也不知道什么时候可以放心地结束测试。
- ✧ 每个开发人员应当测试自己的程序（份内之事），但是不能作为该程序已经通过测试的依据（所以项目需要独立测试人员）。
- ✧ 80-20 原则：80％的缺陷聚集在 20％的模块中，经常出错的模块改错后还会经常出错。

1.4.7 质量保证

质量保证（Quality Assurance, QA）的目的是提供一种有效的人员组织形式和管理方法，通过客观地检查和监控“过程质量”与“产品质量”，从而实现持续地改进质量。质量保证是一种有计划的、贯穿于整个产品生命周期的质量管理方法。

过程质量与产品质量存在某种因果关系，通常“好的过程”产生“好的产品”而“差的过程”将产生“差的产品”。人们销售的是产品而不是过程，用户关心的是最终产品的质量，而软件开发团队既要关心过程质量又要关心产品质量。

质量保证的基本方法是通过有计划地检查“工作过程以及工作成果”是否符合既定的规范，来监控和改进“过程质量”与“产品质量”。如果“工作过程以及工作成果”不符合既定的规范，那么产品的质量肯定有问题。基于这样的推理，质量保证人员即使不是技术专家，他也能够客观地检查和监控产品的质量。这是质量保证方法富有成效的一面。但是“工作过程以及工作成果”符合既定的规范却并不意味着产品的质量一定合格，因为仅靠规范无法识别出产品中可能存在的大量缺陷。这是质量保证方法的不足之处。所以单独的“质量保证”其实并不能“保证质量”。

技术评审与测试关注的是产品质量而不是过程质量，两者的技术强度比质量保证要高得多。技术评审和测试能弥补质量保证的不足，三者是相辅相成的质量管理方法。我

们在实践中不能将质量保证、技术评审和测试混为一谈，也不能把三者孤立起来执行。**建议让质量保证人员参加并监督重要的技术评审和测试工作，把三者有机地结合起来，可提高工作效率，降低成本。**

质量保证小组（Quality Assurance Group, QAG）有如下特点：

- ✧ 质量保证小组在行政上独立于任何项目。这种独立性有助于质量保证小组客观地检查和监控产品的质量。
- ✧ 质量保证小组有一定的权利，可以对质量不合格的工作成果做出处理。这种权利使得质量保证小组的工作不会被轻视，并有助于加强全员的质量意识。需要强调的是，提高产品质量是全员的职责，并非只是质量保证小组的职责。

质量保证过程域的主要活动如图 1-5 所示。

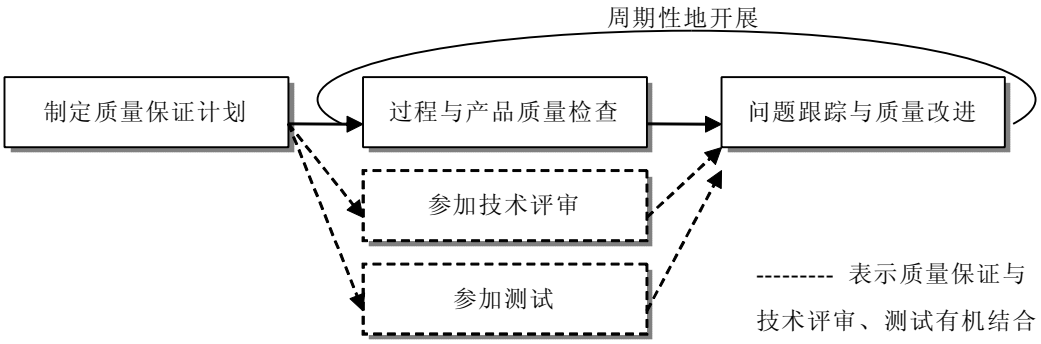

图 1 质量保证过程域示意图

1.4.8 改错

改错是个大悲大喜的过程，一天之内可以让人在悲伤的低谷和喜悦的颠峰之间跌荡起伏。如果改过了成千上万个程序错误，那么少男少女们不必经历失恋的挫折也能变得成熟起来。

我从大三开始真正接受改错的磨练，已记不清楚多少次汗流浃背、湿透板凳。改不了错误时，恨不得撞墙。改了错误时，比女孩子朝我笑笑还开心。

在做本科毕业设计时，一天夜里，一哥们流窜到我的实验室，哈不拢嘴地对我嚷嚷："你知道什么叫茅塞顿开吗？"

我象文盲似地摇摇头。

他说："今天我化了十几个小时没能干掉一个错误，刚才我去了厕所五分钟，一切都解决了。"

他还用那没洗过的手拉我，一定要请我吃"肉夹馍"。那得意劲儿仿佛同时谈了两个女朋友。

软件中的错误通常只有开发者自己才能找出并改掉。如果因畏惧而拖延，会让你终日心情不定，食无味，睡不香。所以长痛不如短痛，要集中精力对付错误。

东北有个林场工人，工作勤奋，一人能干几个人的活。前三十年是伐树劳模，受到周总理的接见。忽有一天醒悟过来，觉得自己太对不起森林，决心补救错误。后三十年

成了植树劳模，受到朱总理的接见。若能以此大勇来改错，正是无往而不胜也。我们软件开发人员应当向这位可敬的林场工人学习。

改错过程很像侦破案件，有些坏事发生了，而仅有的信息就是它的确发生了。我们必须从结果出发，逆向思考。改错的第一步是找出错误的根源，如同医生治病，必须先找出病因才能“对症下药”。

有人问阿凡提：“我肚子痛，应该用什么药？”

阿凡提说：“应该用眼药水，因为你眼睛不好，吃了脏东西才肚子痛。”

根据软件错误的症状推断出根源并不是件容易的事，因为：

（1）症状和根源可能相隔很远。也就是说，症状可能在某一个程序单元中出现，而根源实际上在很远的另一个地方。高度耦合的程序结构加剧了这种情况。

（2）症状可能在另一个错误被纠正后暂时性消失。

（3）症状可能并不是由某个程序错误直接引发的，如误差累积。

（4）症状可能是由不太容易跟踪的人工错误引起的。

（5）症状可能时隐时现，如内存泄漏。

（6）很难重新产生完全一样的输入条件，难以恢复“错误的现场”。

（7）症状可能分布在许多不同的任务中，难以跟踪。

改错的最大忌讳是“急躁蛮干”。人们常说“急中生智”，我不信。我认为大多数人着急了就会蛮干，早把“智”丢到脑后。不仅人如此，动物也如此。

我们经常看到，蜜蜂或者苍蝇想从玻璃窗中飞出，它们会顶着玻璃折腾几个小时，却不晓得从旁边轻轻松松地飞走。我原以为蜜蜂和苍蝇长得太小，视野有限，以致看不见近在咫尺的逃生之窗，所以只好蛮干。可是有一天夜里，有只麻雀飞进我的房间，它的逃生方式竟然与蜜蜂一模一样。我用灯光照着那扇打开的窗户为其引路，并向它打手势，对它说话，均无济于事。它是到天亮后才飞走的，这一宿我和它都没有休息好。

我们把寻找错误根源的过程称为调试（Debugging）。调试的基本方法是“粗分细找”。对于隐藏得很深的 Bug，我们应该运用归纳、推理、“二分”等方法先“快速、粗略”地确定错误根源的范围，然后再用调试工具仔细地跟踪此范围的源代码。如果没有调试工具，那么只好用“土办法”：在程序中插入打印语句如 printf(…)，观看屏幕的输出。

有些时候，世界上最好的调试工具恐怕是那些有经验的人。我们经常会长时间地追踪某个 Bug，苦恼万分。恰好有高手路过，被他一语“道破天机”。顿时沮丧的阴云就被驱散，你不得不说“I 服了 You”。

修改代码错误时的注意事项：

- ✧ 找到错误时，不要急于修改，先思考一下：修改此代码会不会引发其它问题？如果没有问题，则可以放心修改。如果有问题，那么可能要改动程序结构，而不止一行代码。
- ✧ 有些时候，软件中可能潜伏同一类型的许多错误（例如由不良的编程习惯引起的）。好不容易逮住一个，应当乘胜追击，全部歼灭。
- ✧ 在改错之后一定要马上进行回归测试，以免引入新的错误。有人在马路上捡到钱包后得意忘形，不料自己却被汽车撞倒。改了一个程序错误固然是喜事，但

要防止乐极生悲。更加严格的要求是：不论原有程序是否绝对正确，只要对此程序作过改动（哪怕是微不足道的），都要进行回归测试。

- ✧ 上述事情做完后，应当好好反思：我为什么会犯这样的错误？怎么能够防止下次不犯相似的错误？最好能写下心得体会，与他人共享经验教训。

1.6 关于软件开发的一些常识和思考

1.6.1 有最好的编程语言吗

作者的观点：程序员在最初学习 Basic、Fortran、 Pascal、C、C++等语言时会感觉一个比一个好，不免有喜新厌旧之举。而如今的 Visual Basic、Delphi、Visual C++、Java 等语言各有所长，真的难分优劣。能很好地解决问题的编程语言就是好语言。开发人员应该根据实际情况，选择业界推荐的并且是自己擅长的编程语言来开发软件，才能保证有较好的质量与生产率。

编程是件自由与快乐的事情，不要发誓忠于某某语言而自寻烦恼。

1.6.2 编程是一门艺术吗

作者的观点：水平高到一定程度后，干啥事都能感受到“艺术”，编程也不例外。但在技术行业，人们通常认为“艺术”是随心所欲、不可把握的东西。如果程序员都把编程当成“艺术”来看待，准会把公司老板吓昏过去。

大部分人开发软件是为了满足客户的需求，而不是为了自己享受。本书提倡规范化编程。规范化能够提高质量与生产率，最具实用价值，尽管它在一定程度上压抑了“艺术”。编程艺术是人们对高水平程序创作的一种感受，但只可意会，不可言传，不能成为软件公司的一个指导方针。

1.6.3 编程时应该多使用技巧吗

作者的观点：就软件开发而言，技巧的优点在于能另辟蹊径地解决一些问题，缺点是技巧并不为人熟知。若在程序中使用太多的技巧，可能会留下错误隐患，别人也难以理解程序。鉴于一个局部的优点对整个系统而言是微小的，而一个错误则可能对整个系统是致命的。我建议用自然的方式编程，不要滥用技巧。我们有时的确不知道自己的得意之举究竟是锦上添花，还是画蛇添足。就象蒸出一笼馒头，在上面插一朵鲜花，本想弄点诗情画意，却让人误以为那是一堆热气腾腾的牛粪。

小时候读的《狼三则》故事启示我们，失败的技巧被讽刺为“技俩”。当我们在编程时无法判断是用了技巧还是用了技俩，那就少用。《卖油翁》的故事又告诉我们“熟能生巧”，表明技巧是自然而然产生的，而不是卖弄出来的。卖油翁的绝技是可到中央电视台

表演的，而他老人家却谦虚地说：“没啥没啥，用熟了而已”。

1.6.4 换更快的计算机还是换更快的算法

如果软件运行较慢，是换一台更快的计算机，还是设计一种更快的算法？

作者的观点：如果开发软件的目的是为了学习或是研究，那么应该设计一种更快的算法。如果该软件已经用于商业，则需谨慎考虑。若换一台更快的计算机能解决问题，则是最快的解决方案。改进算法虽然可以从根本上提高软件的运行速度，但可能引入错误以及延误进度。

技术狂毫无疑问会选择后者，因为他们觉得放弃任何可以优化的机会就等于犯罪。类似的争议还有：是买现成的程序，还是彻底由自己开发？技术人员和商业人士常常会有不同的决策。

1.6.5 错误是否应该分等级

微软的一些开发小组将错误分成四个等级[Cusumano, p354-p355]：

- ✧ 一级严重：错误导致软件崩溃。
- ✧ 二级严重：错误导致一个特性不能运行并且没有替代方案。
- ✧ 三级严重：错误导致一个特性不能运行但有替代方案。
- ✧ 四级严重：错误是表面化的或是微小的。

作者的观点：将错误分等级的好处是便于统计分析。但上述分类带有较重的技术倾向，并不是普适的。假设某个财务软件有两个错误：错误 A 使该软件死掉，错误 B 导致工资计算错误。按上述分类，错误 A 属一级严重，错误 B 属二级严重。但事实上 B 要比 A 严重。工资算多了或者算少了，将会使老板或员工遭受经济损失。而错误 A 只是使操作员感到厌烦，并没有造成经济损失。再例如操作手册写错了，按上述分类则属四级严重，但这种错误可能导致机毁人亡，难道还算微小吗？

开发人员应该意识到：所有的错误都是严重的，不存在微不足道的错误。只有这样才能少犯错误。

1.6.6 一些错误的观念

错误观念之一：我们拥有一套讲述如何开发软件的书籍，书中充满了标准与示例，可以帮助我们解决软件开发中遇到的任何问题。

作者的观点：好的参考书无疑能指导我们的工作。充分利用书籍中的方法、技术和技巧，可以有效地解决软件开发中大量常见的问题。但实践者并不能因此依赖于书籍，这是因为：

（1） 在现实中，由于工作条件千差万别，即使是相当成熟的软件工程规范，也常常无法套用。

（2） 软件技术日新月异，没有哪一种标准能长盛不衰。祖传秘方在某些领域很吃香，而在软件领域可能意味着落后。

错误观念之二：我们拥有充足的资源和经费，可以买最好的设备，一定能做出优秀的软件产品。

作者的观点：大公司经常有这样夜郎自大的心态。良好的开发环境只是产出成果的必要条件，而不是充分条件。如果拥有好环境的是一群庸人或者是一群勾心斗角的聪明人，难保他们不干出南辕北辙的事情。

错误观念之三：如果我们落后于计划，可以增加更多的程序员来解决问题。

作者的观点：软件开发不同于传统的农业生产，人多不见得力量大。如果给落后于计划的项目增添新手，可能会更加延误项目。因为：

（1） 新手会产生很多新的错误，给项目添麻烦。

（2） 老手向新手解释工作以及交流思想都要花费时间，使实际开发时间更少。

所以精确地制定项目计划很重要，不在乎计划中的进度看起来有多么快，计划要恰如其分。如果用“大跃进”的方式干活，只会产生倒退的后果。

错误观念之一：只要干活小心点，就能提高软件的质量。

作者的观点：软件开发是一种智力创作活动，世上最小心翼翼、最老实巴脚的程序员未必就能开发出高质量的软件来。程序员必须了解软件质量的方方面面（称为质量属性素），一定要先搞清楚怎样才能提高质量，才可以在进行需求开发、系统设计、编程、测试时将高质量内建其中。

1.7 小结

经典的软件工程书籍厚得象砖头，或让人望而却步，或让人看了心事重重。请宽恕我的幼稚，我试图用白话文的方式来解释软件工程的道理。本章虽然有些罗嗦，但还称不上是又臭又长。

软件工程的观念、方法、和规范都是朴实无华的，平凡之人皆可领会，但只有实实在在地用起来才有价值。我们不可以把软件工程方法看成是诸葛亮的锦囊妙计——在出了问题之后才打开看看，而应该事先预料将要出现的问题，控制每个实践环节，防患于未然。

研究软件工程永远做不到理论家那么潇洒：定理证明了，就完事。