第1章 专业主义

“噢，笑吧，科廷，老伙计。这是上帝，或者也可以说是命运或自然，跟我们开的一个玩笑。不过，不管这家伙是谁或是什么，他真幽默！哈哈！”

——霍华德，《碧血金沙》

这么说，你确实是想成为一名专业的软件工程师，对吧？你希望能昂首挺胸向世界宣告“我是专业人士”，希望人们充满敬意地注视着你，对你礼遇有加。希望母亲们会指着你告诉自己的孩子要成为像你这样的人。这些都是你想要的，

对吧？

1.1 清楚你要什么

“专业主义”有很深的含义，它不但象征着荣誉与骄傲，而且明确意味着责任与义务。这两者密切相关，因为从你无法负责的事情上不可能获得荣誉与骄傲。

做个非专业人士可轻松多了。非专业人士不需要为自己所做的工作负责，他们大可把责任推给雇主。如果非专业人士把事情搞砸了，收拾摊子的往往是雇主；而专业人士如果犯了错，只好自己收拾残局。

如果你不小心放过了某个模块里的一个 bug，以致公司损失了 1 万美元，结果将会怎样呢？非专业人士会耸耸肩说：“状况总是难免的嘛。”然后像没事儿人一样继续写下一个模块。而专业人士会自己为公司的那 1 万美元买单[①]！

哇，自掏腰包？那可真让人心疼唉！但专业人士就必须这么做。实际上，专业主义的精髓就在于将公司利益视同个人利益。看到了吧，“专业主义”就意味着担当责任。

1.2 担当责任

想必你读过前面的引言了，对吧？如果没有，赶紧翻回去读一遍，因为本书将要讲的内容，都在其营造的语境里展开。

我曾因不负责任尝尽了苦头，所以明白尽职尽责的重要意义。

那是 1979 年，当时我是一家叫 Teradyne 的公司的“负责工程师”，所负责的软件控制着一个测量电话线路质量的小型机系统和微机系统，该系统的中央小型机通过带宽为 300 波特的拨号电话线与几十台控制测量硬件的外围微机连接在一起，程序是用汇编语言编写的。

我们的客户是各大电话公司的客服经理，他们每个人都负责 10 万条甚至更多

① 但愿他上了不错的错漏保险！

的电话线路。我的系统负责帮助这些服务区经理抢在客户之前发现各种线路故障并及时修复。这可以减少客户投诉率，以免对此做监测的公共设施委员会相应下调电话公司收取的服务费。简单地说，这些系统极其重要。

每天晚上，这些系统都会运行“夜间例行程序”，即中央小型机会通知外围微机对所控制的电话线路进行检测；每天早上，中央计算机就能获取故障线路清单及其故障特征。根据这些报告，各服务区经理会安排人员修复故障，这样就不会有客户投诉了。

一次，我对几十个客户推出了一版新发布。“推出”这词可真是形象啊。我把软件写在磁带上，就把这些带子“推出”给客户了。客户载入这些磁带，然后重启系统。

这一新发布修复了几个小故障，还增加了客户要求的一项新功能。之前我们曾承诺会在截止日期之前提供那项新功能。我连夜赶工，总算在约定日期前交付了磁带。

两天后，我接到现场服务经理 Tom 的电话，他告诉我已经有好几个客户投诉“夜间例行程序”没能执行完成，他们没收到任何报告。我不由心头一沉：为了按时交付软件，我没测试“例程”。我测试了系统的其他大部分功能，但测试“例程”要费好几个小时，而当时我又必须交付软件。因为故障修复部分都不涉及“例程”部分的编码，所以我也没担心会有什么不妥。

收不到夜间报告，问题可就大了。修理工们会一时无事可忙但随后又要超负荷工作，而且，有些电话客户也可能会在这期间发现故障并投诉。要是弄丢一晚的数据，某一服务区经理肯定会打电话臭骂 Tom。

我启动实验室系统，加载新软件，然后开始对“夜间例行程序”进行测试。几小时后，运行中断。“例程”运行失败！如果我在匆忙交付软件前对此进行测试，就不会发生服务区丢失数据的事了，服务区经理们这时也不会炮轰 Tom 了。

我打电话给 Tom，说我能重现问题了。Tom 告诉我其他大部分客户也已经打电话抱怨了，并问我什么时候能解决问题。我说我也没把握，但正在努力。同时我告诉他应该建议客户倒回去使用旧版软件。Tom 发火了，说那对客户来说无疑

是个双重打击，因为客户不仅为此丢失了一整个晚上的数据，而且还无法使用事先承诺的新功能。

故障排查非常困难，每次测试就要好几个小时。第一次修复失败了。第二次也没能成功。我试了好几次，等我发现问题所在时，好几天已过去了。这期间，Tom 每隔几小时就打电话问我问题什么时候能解决，他也没少让我听那些服务区经理对他喋喋不休的抱怨，并一再告诉我让那些客户重新起用旧软件令他多么尴尬。

最后，我终于找出了缺陷所在，重新交付修复了问题的新程序，一切恢复正常。Tom 也平静下来，不再提这段插曲，毕竟，他不是我的上司。事后，我的老板过来对我说：“ 你最好别再犯同样的错误。” 我只能默默地点点头。

经过反省，我意识到未对“ 例程” 进行测试就交付软件是不负责任的。为了如期交付产品，我忽略了测试环节，整个过程中只考虑要如何保全自己的颜面，却没顾及客户和雇主的声誉。我本该早点儿担起责任，告诉 Tom 测试还未完成、自己不能按时交付产品。那么做绝非易事，Tom 一定会不高兴，但客户不会丢失数据，客服经理也不会打电话来轰炸。

1.3 首先，不行损害之事

那么，我们该如何承担责任呢？的确有一些原则可供参考。援引“ 希波克拉底誓言” 或许显得有点夸张，但没有比这更好的引据了。的确，作为一名有追求有抱负的专业人士，他的首要职责与目标难道不正是尽其所能行有益之事吗？

软件开发人员能做出什么坏事呢？从纯软件角度看，他可以破坏软件的功能与架构。我们会探讨如何避免带来这些破坏。

1.3.1 不要破坏软件功能

显然，我们希望软件可以运行。没错，我们中的大部分人今天之所以是程序员，是因为我们曾带来了可用的软件，而且希望能再度体验那种成功创作的喜悦。

但希望软件有用的不单单是我们，客户和雇主也希望它们能用。是啊，他们出钱，让我们去开发那些能按照他们意愿运行的软件。

开发的软件有 bug 会损害软件的功能。因此，要做得专业，就不能留下 bug。

“ 等等！”你肯定会说，“ 可是那是不可能的呀。软件开发太复杂了，怎么可能会没 bug 呢！”

当然，你说的没错。软件开发太复杂了，不可能没什么 bug。但很不幸，这并不能为你开脱。人体太复杂了，不可能尽知其全部，但医生仍誓言不伤害病人。如果他们都不拿“ 人体的复杂性” 作托辞，我们又怎么能开脱自己的责任呢？

“ 你的意思是我们要追求完美喽？” 你可能会这样抬杠吧？

不，我其实是想告诉你，要对自己的不完美负责。代码中难免会出现 bug，但这并不意味着你不用对它们负责；没人能写出完美的软件，但这并不表示你不用对不完美负责。

所谓专业人士，就是能对自己犯下的错误负责的人，哪怕那些错误实际上在所难免。所以，雄心勃勃的专业人士们，你们要练习的第一件事就是“ 道歉” 。道歉是必要的，但还不够。你不能一而再、再而三地犯相同的错误。职业经验多了之后，你的失误率应该快速减少，甚至渐近于零。失误率永远不可能等于零，但你有责任让它无限接近零。

1. 让 QA 找不出任何问题

因此，发布软件时，你应该确保 QA 找不出任何问题。故意发送明知有缺陷的代码，这种做法是极其不专业的。什么样的代码是有缺陷的呢？那些你没把握的代码都是！

有些家伙会把 QA 当作抓虫机器看待。他们把自己没有全盘检查过的代码发送过去，想等 QA 找出 bug 再反馈回来。没错，有些公司确实按照所发现的 bug 数来奖励测试人员，揪出的 bug 越多，奖金越多。

且不说这么做是否会大幅增加公司成本，严重损害软件，是否会破坏计划并让企业对开发小组的信心打折扣，也不去评判这么做是否等同于懒惰失职，把自

己没把握的代码发送给 QA 这么做本身就是不专业的。这违背了“不行损害之事”的原则。

QA 会发现 bug 吗？可能会吧，所以，准备好道歉吧，然后反思那些 bug 是怎么逃过你的注意的，想办法防止它再次出现。

每次 QA 找出问题时，更糟糕的是用户找出问题时，你都该震惊羞愧，并决心以此为戒。

2. 要确信代码正常运行

你怎么知道代码能否常运行呢？很简单，测试！一遍遍地测，翻来覆去、颠来倒去地测，使出浑身解数来测！

你或许会担心这么狂测代码会占用很多时间，毕竟，你还要赶进度，要在截止日期前完工。如果不停地花时间做测试，你就没时间写别的代码了。言之有理！所以要实行自动化测试。写一些随时都能运行的单元测试，然后尽可能多地执行这些测试。

要用这些自动化单元测试去测多少代码呢？还要说吗？全部！全部都要测！

我是在建议进行百分百测试覆盖吗？不，我不是在建议，我是在要求！你写的每一行代码都要测试。完毕！

这是不是不切实际？当然不是。你写代码是因为想执行它，如果你希望代码可以执行，那你就该知道它是否可行。而要知道它是否可行，就一定要对它进行测试。

我是开源项目 FitNesse 的主要贡献者和代码提交者。在写作本书的时候，FitNesse 的代码有 6 万多行。在这 6 万行代码中有 2000 多个单元测试，共 2.6 万多行。Emma 的报告显示，这 2000 多个测试对代码的覆盖率约为 90%。

为什么只有 90%呢？因为 Emma 会忽略一些执行的代码。我确信实际的覆盖率会比 90%高许多。能达到 100%吗？不，达不到，100%只是个理想值。

但是有些代码不是很难测试吗？是的，但之所以很难测试，是因为设计时就没考虑如何测试。唯一的解决办法就是要设计易于测试的代码，最好是先写测试，

再写要测的代码。

这一方法叫做测试驱动开发（TDD），我们在随后的章节里会继续谈到。

3. 自动化 QA

FitNesse 的整个 QA 流程即是执行单元测试和验收测试。如果这些测试通过了，我就会发布软件。这意味着我的 QA 流程大概需要 3 分钟，只要我想要，可以随时执行完整的测试流程。

没错，FitNesse 即使有 bug 也不是什么人命关天的事，也不会有人为此损失几百万美元。值得一提的是 FitNesse 用户上万，但它的 bug 列表却很短。

当然，也不排除有些系统因其任务极其关键特殊，不能只靠简短的自动化测试来判断软件是否已经足够高质量，是否可以投入使用。而且，作为开发人员，你需要有个相对迅捷可靠的机制，以此判断所写的代码可否正常工作，并且不会干扰系统的其他部分。因此，你的自动化测试至少要能够让你知道，你的系统很有可能通过 QA 的测试。

1.3.2 不要破坏结构

成熟的专业开发人员知道，聪明人不会为了发布新功能而破坏结构。结构良好的代码更灵活。以牺牲结构为代价，得不偿失，将来必追悔莫及。

所有软件项目的根本指导原则是，软件要易于修改。如果违背这条原则搭建僵化的结构，就破坏了构筑整个行业的经济模型。

简言之，你必须保证，不需太高代价就可以。

不幸的是，实在是已有太多的项目因结构糟糕而深陷失败的泥潭。那些曾经只要几天就能完成的任务现在需要耗费几周甚至几个月的时间。急于重新树立威望的管理层于是聘来更多的开发人员来加快项目进度，但这些开发人员只会进一步破坏结构，乱上添乱。

描述如何创建灵活可维护的结构的软件设计原则和模式[①]已经有许多了。专

① [PPP2001]

业的软件开发人员会牢记这些原则和模式，并在开发软件时认真遵循。但是其中有一条实在是没几个软件开发人员会认真照做，那就是，如果你希望自己的软件灵活可变，那就应该时常修改它！

要想证明软件易于修改，唯一办法就是做些实际的修改。如果发现这些改动并不像你预想的那样简单，你便应该改进设计，使后续修改变简单。

该在什么时候做这些简单的小修改呢？随时！关注哪个模块，就对它做点简单的修改来改进结构。每次通读代码的时候，也可以不时调整一下结构。

这一策略有时也叫“无情重构”，我把它叫作“童子军训练守则”：对每个模块，每检入一次代码，就要让它比上次检出时变得更为简洁。每次读代码，都别忘了进行点滴的改善。

这完全与大多数人对软件的理解相反。他们认为对可工作软件不断地做一系列修改是危险的。错！让软件保持固定不变才是危险的！如果一直不重构代码，等到最后不得不重构时，你就会发现代码已经“僵化了”。

为什么大多数开发人员不敢不断修改他的代码呢？因为他们害怕会改坏代码！为什么会有这样的担心呢？因为他们没做过测试。

话题又回到测试上来了。如果你有一套覆盖了全部代码的自动化测试，如果那套测试可以随时快速执行，那么你根本不会害怕修改代码。怎样才能证明你不怕修改代码呢？那就是，你一直在改。

专业开发人员对自己的代码和测试极有把握，他们会极其疯狂随意地做各种修改。他们敢于随心所欲修改类的名称。在通读代码时，如果发现一个冗长的方法，他们肯定会将它拆分，重新组织。他们还会把 switch 语句改为多态结构，或者将继承层次重构成一条“命令链”。简单地说，他们对待代码，就如同雕塑家对待泥巴那样，要对它进行不断的变形与塑造。

1.4 职业道德

职业发展是你自己的事。雇主没有义务确保你在职场能够立于不败之地，也

没义务培训你，送你参加各种会议或给你买各种书籍充电。这些都是你自己的事。将自己的职业发展寄希望于雇主的软件开发人员将会很惨。

有些雇主愿意为员工买各种书籍或送员工参加各种培训课程和会议。那样挺不错的，说明他们待你不薄。但可千万别就此认为这些是雇主该做的。如果他们不为你做这些，你就该自己想办法去做。

另外，雇主也没义务给你留学习时间。有些雇主会这么做，有些甚至要求你这么做。但是还是那句话，他们待你不薄，你应该适当表示感激。因为这些优待不是你理所当然就该享有的。

雇主出了钱，你必须付出时间和精力。为了说明问题，就用一周工作 40 小时的美国标准来做参照吧。这 40 小时应该用来解决雇主的问题，而不是你自己的问题。

你应该计划每周工作 60 小时。前 40 小时是给雇主的，后 20 小时是给自己的。在这剩余的 20 小时里，你应该看书、练习、学习，或者做其他能提升职业能力的事情。

你肯定会说：“那我的家庭该怎么办？还有我的生活呢？难道我就该为雇主牺牲这些吗？”

在此，我不是说要占用你全部的业余时间。我是指每周额外增加 20 小时，也就是大约每天 3 小时。如果你在午饭时间看看书，在通勤路上听听播客，花 90 分钟学一门新的语言，那么你就都能兼顾到了。

做个简单的计算吧。一周有 168 小时，给你的雇主 40 小时，为自己的职业发展留 20 小时，剩下的 108 小时再留 56 小时给睡眠，那么还剩 52 小时可做其他的事呢。

或许你不愿那么勤勉。没问题。只是那样的话你也不能自视为专业人士了，因为所谓“术业有专攻”那也是需要投入时间去追求的。

或许你会觉得工作就该在上班时完成，不该再带回家中。赞成！那 20 小时你不用为雇主工作。相反，你该为自己的职业发展工作。

有时这两者并不矛盾，而是一致的。有时你为雇主做的工作让你个人的职业发展受益匪浅，这种情况下，在那 20 小时里花点时间为雇主工作也是合理的。但别忘了，那 20 小时是为你自己的。它们将会让你成为更有价值的专业人士。

或许你会觉得这样做只会让人精力枯竭。恰恰相反，这样做其实能让你免于枯竭匮乏。假设你是因为热爱软件而成为软件开发者，渴望成为专业开发者的动力也正是来自对软件的热情，那么在那 20 小时里，就应该做能够激发、强化你的热情的事。那 20 小时应该充满乐趣！

1.4.1 了解你的领域

你知道什么是 N-S（Nassi-Schneiderman）图表吗？如果不知道，那为什么不了解一下呢？你知道“米利型”（Mealy）和“摩尔型”（Moore）这两种状态机的差别吗？你应该知道的。你能不需查阅算法手册就可写出一个快速排序程序吗？你知道“变换分析”（Transform Analysis）这个术语的意思吗?你知道如何用数据流图进行功能分解吗？你知道“临时传递数据”（Tramp Data）的意思吗？你听说过“耦合性”（Conascence）吗？什么是 Parnas 表呢？

近 50 年来，各种观点、实践、技术、工具与术语在我们这一领域层出不穷。你对这些了解多少呢？如果想成为一名专业开发者，那你就得对其中的相当一大部分有所了解，而且要不断扩展这一知识面。

为什么要了解这些呢？这一行业发展迅速，许多旧见解似乎也已经过时了，不是吗？前半句似乎是显而易见的。确实，行业正迅猛发展，而有趣的是，从多个方面来看，这种进展都只是很浅层的。没错，我们不再需要为拿到编译结果苦等上 24 小时，我们也已经可以写出吉字节级别的系统，我们置身覆盖全球的网络之中，各种信息唾手可得。但另一方面，我们还是跟 50 年前一样，写着各种 if 和 while 语句。所以，改变说多也多，说少也少。

旧见解过时了这种说法明显是不对的。过去 50 年中产生的理念，已经过时的其实很少。有一部分理论确实在慢慢淡出，比如说“瀑布式开发”的理论确实不再流行了。但这并不表示我们不需要了解它，不需要知道他的长处和短处。

总的来说，那些在过去 50 年中来之不易的理念，绝大部分在今天仍像过去一样富有价值，甚至宝贵了。

别忘了桑塔亚纳的诅咒：“ 不能铭记过去的人，注定重蹈先人的覆辙。”

下面列出了每个专业软件开发人员必须精通的事项。

- 设计模式。必须能描述 GOF 书中的全部 24 种模式，同时还要有 POSA 书中的多数模式的实战经验。
- 设计原则。必须了解 SOLID 原则，而且要深刻理解组件设计原则。
- 方法。必须理解 XP、Scrum、精益、看板、瀑布、结构化分析及结构化设计等。
- 实践。必须掌握测试驱动开发、面向对象设计、结构化编程、持续集成和结对编程。
- 工件。必须了解如何使用 UML 图、DFD 图、 结构图、Petri 网络图、状态迁移图表、流程图和决策表。

1.4.2 坚持学习

软件行业的飞速改变，意味着软件开发人员必须坚持广泛学习才不至于落伍。不写代码的架构师必然遭殃，他们很快会发现自己跟不上时代了；不学习新语言的程序员同样会遭殃，他们只能眼睁睁看着软件业径直向前，把自己抛在后面；学不会新原则和技术的开发人员必将沦落，他们身边的人都日益卓越。

你会找那些已经不看医学期刊的医生看病吗？你会聘请那些不了解最新税法和判例的税务律师吗？雇主们干嘛要聘用那些不能与时俱进的开发人员呢？

读书，看相关文章，关注博客和微博，参加技术大会，访问用户群，多参与读书与学习小组。不懂就学，不要畏难。如果你是.NET 程序员，就去学学 Java；如果你是 Java 程序员，就去学学 Ruby；如果你是 C 语言程序员，就去学学 Lisp；如果你真想练练脑子，就去学学 Prolog 和 Forth 吧！

1.4.3 练习

业精于勤。真正的专业人士往往勤学苦干，以求得自身技能的纯熟精炼。只完成日常工作是不足以称为练习的，那只能算是种执行性质的操作，而不是练习。练习，指的是在日常工作之余专门练习技能，以期自我提升。

对软件开发人员来说，有什么可以用以操练的呢？乍一听，这概念显得荒唐。但是再仔细想一会儿，想想音乐家是如何掌握演练技能的。他们靠的不是表演，而是练习。他们又是如何练习的呢？首先，表演之前，都需要经历过特别的训练，音阶、练习曲、不断演奏等。他们一遍又一遍地训练自己的手指和意识，保持技巧纯熟。

那么软件开发者该怎样来不断训练自己呢？本书会用一整章的篇幅来谈论各种练习技巧，所以在此先不赘述了。简单说，我常用的一个技巧是重复做一些简单的练习，如“保龄球游戏”或“素数筛选”，我把这些练习叫作“卡塔”（kata）[①]。卡塔有很多类型。

卡塔的形式往往是一个有待解决的简单编程问题，比如编写计算拆分某个整数的素数因子等。做卡塔的目的不是找出解决方法（你已经知道方法了），而是训练你的手指和大脑。

每天我都会做一两个卡塔，时间往往安排在正式投入工作之前。我可能会选用 Java、Ruby、Clojure 或其他我希望保持纯熟的语言来练习。我会用卡塔来培养某种专门的技能，比如让我的手指习惯点击快捷键或习惯使用某些重构技法等。

不妨早晚都来个 10 分钟的卡塔吧，把它当作热身练习或者静心过程。

1.4.4 合作

学习的第二个最佳方法是与他人合作。专业软件开发人员往往会更加努力地尝试与他人一起编程、一起练习、一起设计、一起计划，这样他们可以从彼此身上学到很多东西，而且能在更短的时间内更高质量地完成更多工作。

并不是让你花全部时间一直和别人共事。独处的时间也很重要。虽然我很喜

---

① kata，这个词目前还没有公认的译法，可以理解为“套路”，或者某种固定的“形”。——译者注

欢和别人一起编程，但是如果不能经常独处，我也一样会发疯。

1.4.5 辅导

俗话说：教学相长。想迅速牢固地掌握某些事实和观念，最好的方法就是与由你负责的人交流这些内容。这样，传道授业的同时，导师也会从中受益。

同样，让新人融入团队的最好办法是和他们坐到一起，向他们传授工作要诀。专业人士会视辅导新人为己任，他们不会放任未经辅导的新手乱打乱撞。

1.4.6 了解业务领域

每位专业软件开发人员都有义务了解自己开发的解决方案所对应的业务领域。如果编写财务系统，你就应该对财务领域有所了解；如果编写旅游应用程序，那么你需要去了解旅游业。你未必需要成为该领域的专家，但你仍需要勤勉，付出相当的努力来认识业务领域。

开始一个新领域的项目时，应当读一两本该领域相关的书，要就该领域的基础架构与基本知识作客户和用户访谈，还应当花时间和业内专家交流，了解他们的原则与价值观念。

最糟糕、最不专业的做法是，简单按照规格说明来编写代码，但却对为什么那些业务需要那样的规格定义不求甚解。相反，你应该对这一领域有所了解，能辨别、质疑规格说明书中的错误。

1.4.7 与雇主/客户保持一致

雇主的问题就是你的问题。你必须弄明白这些问题，并寻求最佳的解决方案。每次开发系统，都应该站在雇主的角度来思考，确保开发的功能真正能满足雇主的需要。

开发人员之间互相认同是容易的，但把一方换成雇主，人们就容易产生“彼”、“此”之分。专业人士会尽全力避免这样的狭隘之见。

1.4.8 谦逊

编程是一种创造性活动。写代码是无中生有的创造过程，我们大胆地从混沌之中创建秩序。我们自信地发布准确无误的指令，稍有差错，机器的错误行为就可能造成无法估量的损失。因此，编程也是极其自负的行为。

专业人士知道自己自负，不会故作谦逊。他们熟知自己的工作，并引以为荣；他们对自己的能力充满自信，并因此勇于承担有把握的风险。专业人士不是胆小鬼。

然而，专业人士也知道自己会摔跟头，自己的风险评估也有出错的时候，自己也有力不从心的时候。这时候，如果他们揽镜自照，会看到那个自负的傻瓜正对着自己笑。

因此，在发现自己成为笑柄时，专业人士会第一个发笑。他从不会嘲讽别人，自作自受时他会接受别人的嘲讽。反之，他则会一笑了之。他不会因别人犯错就对之横加贬损，因为他知道，自己有可能就是下一个犯错的人。

专业人士都清楚自己的自负，也知道上天会注意到这种自负，并加以惩戒。如若果真遭遇挫折，上策就是按照霍华德说的——一笑了之吧！

1.5 参考文献

[PPP2001]: Robert C. Martin, *Principles, Patterns, and Practices of Agile Software Development*, Upper Saddle River, NJ: Prentice Hall, 2002.