第 3 章 测试与项目的协调

在本章中，我会探讨令测试适应于项目具体环境的重要概念。在第 6 章中，我将探讨如何在正确的范围内确保你测试了正确的质量特性。在第 4 章中，我会向你提供一些如何选择正确的测试策略的意见。这些都对测试成功至关重要。

然而，测试未能成功地适应项目，是和测试了系统的错误属性，或者使用了错误的测试策略一样危险的，所以在这一章中，我会讨论项目需要如你这样技术熟练的测试专家提供怎样的服务。

3.1 为什么要进行测试

我们为什么在这里？我不是以哲学的角度而是从实践的角度来提出这个问题的。

当我做项目时，我指望得到报酬。有些人可能会付钱给你来测试他们的系统，或者你正在学习测试以便在此领域中找到一个工作。因为测试本身并不能制造财富，有些组织为了进行测试需要专门花钱。测试并不是为它本身而发生的，也不是智力游戏或者服务于某种高贵的目的。测试发生是因为它要服务于需求，作为项目或者操作的一部分提供利益。

不同的组织会从你的工作中谋求不同的需求和利益，诸如：

- 提高质量声誉
- 降低发布后的维护费用
- 更顺畅的发布周期
- 增加自信度
- 确保合法性
- 减少任务失败甚至伤亡事故的风险

这些需求和利益都可以反过来重新描述成为一个风险，例如，一个充满缺陷的客户系统的发布会造成发布后维护费用增加的风险。

3.2 测试的视角

你可能在想：“这些事情看上去没有一件一个测试者应该做的。我寻找缺陷，并试着在我的测试参数中找出哪些可以工作，哪些不能工作”。这些实际上是测试的主要战术，但是大多数的高层经理和执行主管并不会过多考虑这类事务。

高层经理和执行主管想的更多的是战略性的、长期的问题。所以如果你想要和高层经

理或者执行主管沟通那些关于测试的事，则必须从战略上进行沟通。

当谈论测试如何帮助他们管理风险时，你要从战略上来谈论测试。可以谈论测试如何降低组织必须承担未来那些不可预知开销的可能性。这个视角看起来有点像保险，但是并没有实际上的确定性或者包揽损失的承诺。[1]

为什么你应该关心与高层经理和执行主管们进行沟通呢？因为他们有权审批经费让我们的工作能够得以进行！

3.3 在上下文环境中测试

很多时候，优秀的测试团队以一种可信而易懂的方式，把对质量的及时准确评估展现给项目组和经理们，由此实现自己的价值。要做到这样，测试子项目必须和开发或者管理项目的整体相适应。否则我们不可能产生及时准确的信息（信息的展现也不太可信或者易懂），因为我们不能理解也不能让我们的工作和正在进行的项目保持一致。

3.4 常见的测试阶段和目标

将测试活动和项目相适应的一个常见方法是，将整个测试工作分散为一系列阶段或者层次。这些层次常常组织为一个序列，按照系统已经准备好在哪些部分进行测试来划分。故而每个阶段服务于一个高层次的测试目标。

第一个阶段或者层次经常被称为单元、构件或者子系统测试。在这里，可测试已经创建出来的系统部分，当独立的系统片段被集成起来之前，可在这些片段测试中寻找缺陷。

在第二个测试阶段或者测试层次中，通常这一测试层次被称为集成测试（或者不太经常地，被称为群测试），可测试一组交互操作或者互相通信的单元、子系统或者构件，在这些系统片段结合到一起时，可在成对或者成组的关系和界面的测试中寻找缺陷。

第三个阶段或者层次经常被称为系统测试，可测试整个系统，在整个系统总体或者特定的行为、功能或者反应测试中寻找缺陷。

在第四个常被称为可接受性或者引导（Pilot）测试的测试阶段或者层次中，通常不再是寻找缺陷。其目标是展示产品是否已经准备好被部署或者发布。

在这些层次中，重要的并不是名称或者所显示的东西，而是要确保尽可能早地做了足够数量的测试，以预防缺陷在以后浮现而对项目产生更大的危害。测试计划中入口和出口准则（Entry and Exit Criteria）的适当定义能确保阶段或层次之间的顺畅转换。测试计划也

1 关于如何和高层经理和执行主管沟通让他们产生关注，进一步的观点可以在我的测试管理的著作：《管理测试过程》（Managing the Testing Process）和《关键测试过程》（Critical Testing Process）中找到，另外也可以参考我们公司的文库网页（Library）（www.rexblackconsulting.com），“投资测试”（Investing in Testing）文章系列和我的 Keynote 报告“增加价值的 4 个测试方法（Four Ways Testing Add Value）”和“投资测试（Investing in Testing）”。

要参与到对有效和高效的测试至关重要的各个责任小组之间的交接中。[1]

3.5 贯穿组织的测试

当我提到在责任小组之间交接时，已经暗示了一个小组并不必履行所有的测试任务，让我来解释一下怎样做，以及为何你可能想要这样做。

从下往上读一下图 3-1，当你在图中由左向右看时，项目时间线的后半部分（省略项目的需求、设计和实现）表示了时间的推移。

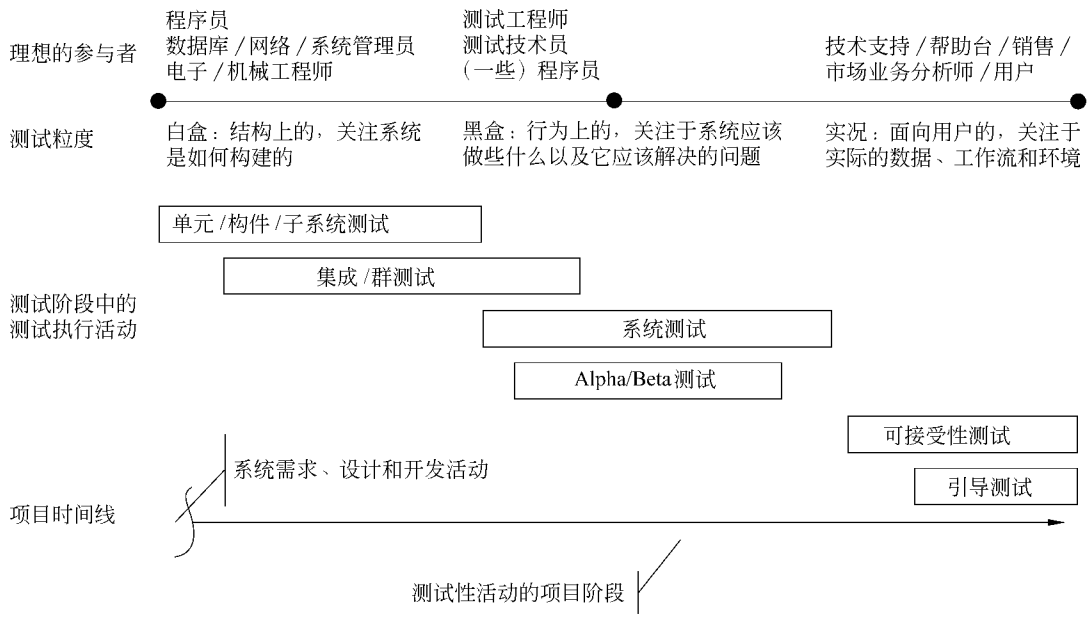

图 3-1　测试执行阶段和参与者

在上述项目时间线中，你将看到前面提到过的按时间顺序显示的测试各阶段。管理项目的不同方法允许这些阶段之间不同规模的彼此覆盖，在一些情况下有些阶段可以完全缺失，而在有些情况下则可以出现更多的阶段。然而，此图显示了一个常见情形下的项目，恰好以一个严谨的顺序包括了所有的测试阶段。

我已在测试阶段的上面，显示了一个测试粒度的说明性谱系。关于**测试粒度**（test granularity），我指的是通过特定的测试技术所关注的细节级别。白盒测试关注于实现细节：代码、数据结构、类和其他的系统底层元素。这是测试技术中粒度最为精细的类别。黑盒测试关注于质量风险、需求和高层次设计。这是相对粒度较粗的类别。实况测试（Live testing）关注于系统的典型使用，它是粒度最粗的测试。在本书的稍后章节中，你会学到每一测试

1　测试计划在我的书《管理测试过程》（Managing the Testing Process）和《关键测试过程》（Critical Testing Process），以及 Martin Pol、Ruud Teunissen 和 Erik van Veeneandaal 的《软件测试：TMap 方法指南》（Software Testing: A Guide to the TMap Approach）中有详细的讨论。

类别的更多内容。

当你从单元测试转移到系统测试到可接受性测试，图 3-1 显示了每阶段使用的主要测试技术类从细粒度逐渐转变为粗粒度。

最后，在测试粒度谱系的上面，你可以看到理想的参与者。单元测试和其他较为精细粒度的测试类别，像早期的集成测试，一般要涉及深入地理解系统是如何构建的。所以构造系统的人通常要拥有足够的能力来做这些测试。当你进入后期集成测试和系统测试等粗粒度的测试后，参与者需要使用特定的测试技能，就像你在本书中学到的那样。最后，当测试需要准确理解系统是怎样被使用的时候，那些最接近于用户的人们，或者用户本身，就是理想的测试者。

图 3-1 只是测试阶段序列的一个例子。重要的是要基于你想达到的目标来选择正确的测试阶段。对于每个测试阶段，测试经理应该定义适合的入口和出口准则。这些准则应该在项目进展过程中一直被遵循。经过仔细的考虑和涉众的双方协议，尤其是需求变化时，才可以放弃这些准则。不过，假若它们是在进度压力恐慌下被放弃的，这就是有问题的。适当的入口和出口准则是将每个测试阶段和严谨的测试工作适应到项目中的重点所在。

我提到了管理项目的不同方法。这些方法有时候也被称作系统生命周期模型。现在让我们来考察一些这样的模型，并看看它们如何影响你的测试工作。

3.6 V 模型

所有系统生命周期的祖先是**瀑布模型**（waterfall model），这个模型中你从一个高层次的需求定义开始，然后进入较低的细节层次，建立一个高层次的设计，而后进入较低层次做低层次设计，再进入更低层次做实现，再后面是测试。图 3-2 显示了该模型的一个侧重于测试的改良版，它被称为 V 模型。

在此处，瀑布模型中的测试工作是拐向上方的，展现了诸如单元测试的早期测试级别是关注于低层次细节的，但诸如系统测试这样的晚期测试级别是由高层次来驱动的。

图 3-2 中横着的双向箭头显示了测试分析、设计和实现任务。请注意和每个测试层次相对应的项目产物是怎样来驱动这些任务的。例如，需求归约（作为系统行为的重要描述符）是对系统测试层次的输入，这主要是一个黑盒级别的行为测试。反向指到需求、设计和实现过程的箭头指出在测试开始之前，测试分析、设计和实现经常找到可以在项目早期被消除的缺陷。这是本书的一个重要主题，我们将在后面的章节中再进行讨论。

虽然这个模型是直观的，而且在很多情形下对测试者、开发者和管理者来说是熟悉的，但是它有一些弱点。比如，它通常是由进度和经费驱动的，即当计划有误或者经费短缺时，通常是测试被取消，因为大多数测试都是发生在最后。对大项目来说，提前 6 个月、12 个月，甚至 18 个月，乃至更长时间来做好完美的计划的可能性非常低。虽说这样，这个模型还是比无序混乱要强。

有些简单的步骤可以避免 V 模型项目在结束时变成测试的灾难。首先，在项目早期就开始测试计划、分析、设计和开发，这样你可以利用早期缺陷检测和消除。其次，确保包括了多个周期的测试执行阶段，允许有时间去修复缺陷并重新测试新的版本。最后，宁可

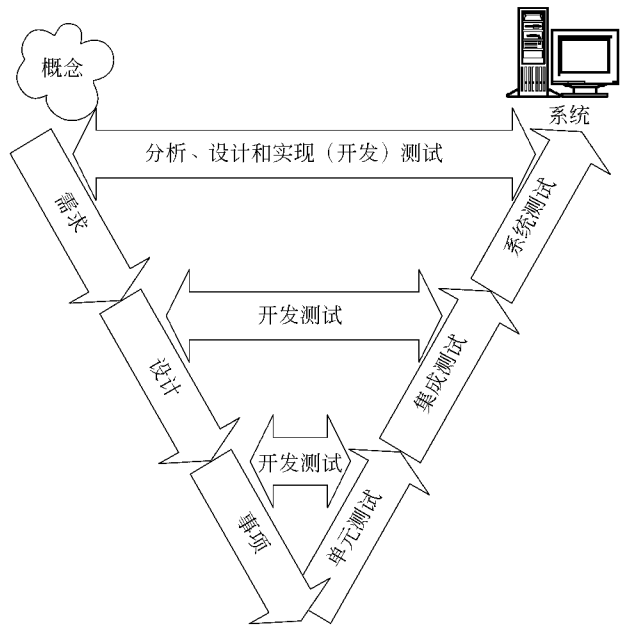

图 3-2　系统开发的 V 模型

考虑把功能推迟到稍后的发布版本，而不要推迟测试阶段开始时间，或者在系统准备好之间就进入某个测试阶段。在系统测试期间有好运气让一切都良好地一起正常工作的可能性是很低的，“系统测试期间的项目大灾难”现象是有大量明文记载的。[1]

图 3-3 显示了一个运行良好的 V 模型项目的项目进度表视图。被选出的和测试相关的开发任务用灰色显示。测试任务用黑色显示。

ID	任务名	开始日期	结束日期	延续时间	2007 十月	2007 十一月	2007 十二月	2008 一月	2008 二月	2008 三月	2008 四月
1	捕获产品需求，撰写项目计划	10/01/2007	10/26/2007	20 天							
2	分析质量风险，撰写测试计划	10/01/2007	10/26/2007	20 天							
3	产品架构和设计规约	10/29/2007	11/23/2007	20 天							
4	测试系统架构和设计规约	10/29/2007	11/23/2007	20 天							
5	开发产品	11/26/2007	01/18/2008	40 天							
6	开发测试系统	11/26/2007	01/18/2008	40 天							
7	调试产品单元	12/02/2007	01/25/2008	40 天							
8	调试测试系统	12/02/2007	01/25/2008	40 天							
9	单元测试执行	12/05/2007	01/29/2008	40 天							
10	集成测试执行	12/05/2007	01/29/2008	40 天							
11	系统测试执行	01/30/2008	03/25/2008	40 天							
12	可接受性执行	03/26/2008	04/08/2008	10 天							

图 3-3　一个 V 模型项目的 Gantt 图表视图

1　我注意到的第一个提出此事的是 Fred Brooks 的经典著作《人月神话》（The Mythical Man-Month），那本书是每个软件和系统工程师的必读著作。

3.7 演化式和增量式模型

为了克服 V 模型的某些挑战，许多项目采用了演化式和增量式模型，如图 3-4 所示。在这种情况下，通过清晰定义的增量来对系统进行分析、设计、开发和测试。在第一个增量被测试之后的任何时刻，项目组都可以至少交付所计划的功能中的一部分。

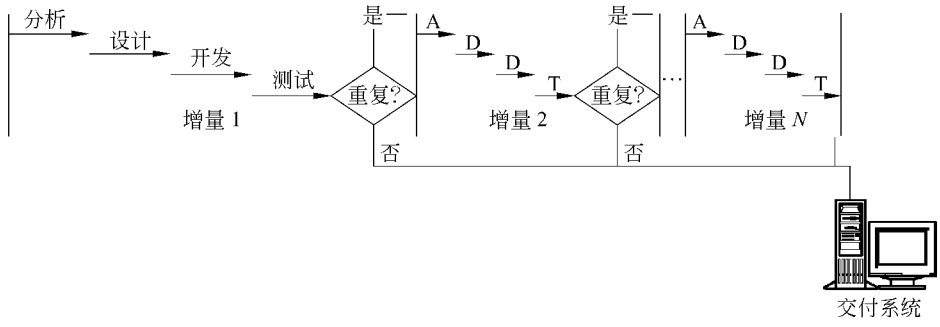

图 3-4　一个增量式开发生命周期

虽然**演化式模型**（evolutionary model）和**增量式模型**（incremental model）这两个术语常常可以互换地使用，然而纯粹主义者会说，当特征集可以预先被定义的话，就是遵循了增量式模型。而当特征集是随时间演化的时候，就是在遵循演化式模型。

当然，你可能会希望特征是以一系列的增量的形式来出现的，其中的第一个增量就是核心功能。每个后续的增量都应该用来添加更多的功能，而这些功能不如前面已经添加的功能重要。通过这样的优先排序方法，团队所交付的永远都是最重要的功能。

从形式上来说，这些方法从轻量的**极限编程**（Extreme Programming）到**快速应用开发方法**（Rapid Application Development），各自有所不同。这些模型并不能解决我们所遇到的所有测试问题。交付一个带有缺陷的特性的系统的倾向依然是无法避免的，虽然这些有问题的功能应该只能是那些最不重要的特性。另外，对于使用在演化中的敏捷开发方法的项目来说，测试的角色也一样是在演化中。[1]

3.8 螺旋模型

当项目组无法精确地提前对系统进行规约描述，但又必须最终交付正确的功能集的时候，螺旋模型就有用武之地了。你可以通过原型化来逐步发现所需的功能。开发者先建立一个最初的原型，然后你开始对它进行测试。在这个时候，测试、重新设计以及原型化可能会一直持续到功能集被最终确定为止。在那个时候，就可以开始运行更加传统的测试集。这种情形被描述在图 3-5 中。

1　比如，可以参考这本书里的经验和观点：Lisa Crispin 和 Tip House 的《测试极端编程》（Testing Extreme Programming）。

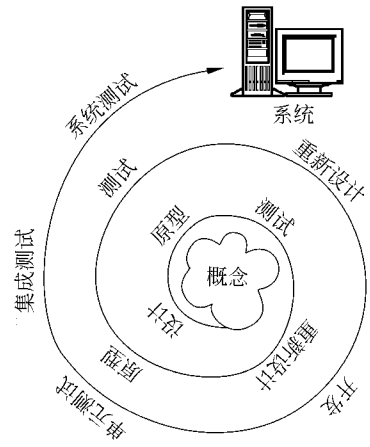

图 3-5　一个系统开发的螺旋模型

这个模型和瀑布模型相比有很大不同。在瀑布模型中，一旦开始进行测试，你就很快接近项目完成了。在螺旋模型中，项目中的第一个原型可能会很快地进入测试阶段，但是你离完成还很远。

你必须要小心，不要把早期的原型作为已完成的产品交付给用户。有时候，来自管理方的压力会逼着你这样去做。在这样的情况下，你的风险分析应当可以帮助你发现重要的测试空白。

3.9 遗憾的常用模型：编码和修正

编码和修正（Code and Fix）是一个讽刺的称呼，指对所要创造的系统并无真正想法，就开始开发系统。你一开始就准备去创建最终系统，而不是先创建一个可能被放弃的原型系统做早期测试。如果这样的方法还存在任何过程的话，那么它可能看起来会是这样的：

1．程序员编写代码和调试代码。通常没有单元测试，或者至少并不能担得起测试的名义。

2．一旦编译器制造出可执行程序，代码就被认为是准备好测试和发布给一个非正式的测试团队，这个团队通常是编程新手或者想成为程序员的人，他们不能找到一个程序员的工作，也没有作为测试者的专业技能。

3．即使是业余的测试，测试者也常能找到许多缺陷。程序员有时匆忙地在测试环境中现场修正这些缺陷，接着这些修正没有被登记到代码库中。于是测试者多次地找到相同的缺陷。有时客户能找到一些测试者已经找到过且程序员已经修正的缺陷。

4．所有前面的步骤同时在重复，很少存在协调工作。

这个过程通常会持续到经费、时间或者项目团队的耐心被耗尽。这个时候，系统就被发布了，或者系统被取消了。有时，公司在经历了一段时间的这种磨难后，业务就垮台了。

上述情形常在典型的创业中出现，我有时称这种创业为“车库中的两个小子”。苹果公司的创立者——Steve Jobs 和 Steve Wozniak，就是在硅谷的某个车库中创立苹果公司的。同

样地，Bill Gates 和一小组程序员也是以同样卑微的起源开创了微软公司。

苹果和微软随着时间成长了，尽管，对“车库中的两个小子”可行的东西，并不见得对十多个甚至几百个人组成的项目团队也可行。特别当项目团队在地理上是分散分布的。当我在一个稍大一点的组织中遭遇了编码-修正方式时，通常是组织内存在一种混乱认识，认为他们有能力在预算范围内及时地交付保证质量的产品，而整个组织对这种混乱认识的负面效应的无知所导致的。

尽管，有时人们把软件工程实体看成某种竭力要逃脱的过程囚笼，或者是一种无厘头的恶作剧。在任何一种情况下，“我们是不同的”、“我们比这更有创造力”、“我们有 XYZ 技术”、“我们是用户驱动的”，以及其他理由常被作为借口来继续采用一个公司因为成长而已经不再适用的模型。

3.10 测试维护性发布

新的开发工作是魅力四射的，但是维护性发布却要为此买单。在我的经验中，和观众谈论维护性的发布时，几乎每个人至少在一个这样的项目中工作过。许多软件工程师没有做过别的，他们的工作只是在已有的代码中增添新特性或者修正缺陷。

到目前为止，我们所做的许多考虑都可以应用于新特性，但已有特性的回归测试却是维护性发布的最大风险，对于缺陷修正的情况尤其如此。在第 4 章，我们会看到多种可以用来管理回归风险的策略。最被广泛地鼓吹的策略，就是完全自动化整个测试，只是其中一种方式。不太频繁地发布，而对每个发布版有着更广的测试覆盖率，以及智能地选择你的测试子集都是可选的方式。

另外一个测试需要考虑的情形是，一个组织努力把值得做一个主发布版本的特性放到一个较短的维护性发布周期中，这将造成我在 V 模式开发下讨论过的同样的测试挤压。把百分之百的开发时间都用于创造新特性是对项目管理者的一个强烈的诱惑，但这从来不现实。谁来修复积压的缺陷？谁来修复被引入的新缺陷？开发人员实际上花了百分之百的时间来编码吗？

最后，开发新测试的时间经常很少。这导致了过分地依赖于回归测试集，造成新功能没有被充分地测试，进而导致回归测试变得过时。当对实现进行相对简单的变更会对系统有影响深远的严重测试后果时，这个问题特别明显。

3.11 系统集成

比起内部开发或者增强软件，越来越多的项目涉及更多定制的或者商业成品（Commercial Off-The-Shelf）子系统。这可能会降低总体的风险，但每个集成的子系统会给系统质量随之带来显著增加的风险。图 3-6 显示了这是如何发生的。

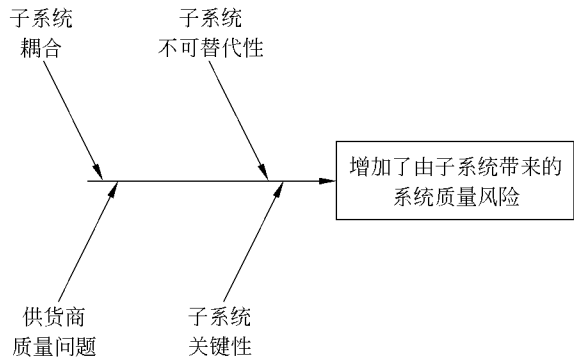

图 3-6　在系统集成中的风险来源

增加风险的一个因素是耦合，当子系统失败时，它与系统（或者系统的结果）之间会存在强大的相互影响。当很难找到相似的可用子系统时，不可替换性是另外一个增加风险的因素。如果子系统达到造成质量问题的程度，你就要被它们困住了。还有一个增加风险的因素是关键性，即系统的一些关键功能或者功能集由于子系统不能正确地工作而不可用。增加风险的因素还有供货商的质量问题和缓慢的缺陷修复周期时间。如果供货商给你的很有可能是一个坏的子系统，整个系统的质量的风险层次会变得更高。

“好吧”，你可能会说，“我看到了系统质量的风险来源，但是我如何去消除这些风险呢？” 我已经发现并使用过不同的可选方法。

其中一个选择是集成、跟踪和管理供货商对其子系统的测试，以此作为系统的整体的、分布式的测试工作的一部分。这涉及预先做好规划，并且对供货商有足够的支配能力，或者供货商能够一直考虑把自己的测试队伍和测试工作从属于或者包含于你的测试组和测试工作中去，当我使用这个方法时，它工作得很好。

另外一个选择是简单地信任供货商子系统已经被测试过，并作为一个可用的子系统提供给你了。这么描述使得此方法听来有些愚蠢和幼稚。然而，项目团队总是在这样做。我的建议是你做此事的时候要睁大眼睛，理解你所接受的风险，并在进度表上为处理问题分配一些时间。

还有一个选择是决定自己来修正供货商的测试或者质量问题。在某个项目中，我的客户雇佣我为一个供货商这样做。当时也完成得很好。需要再次提醒，你的组织必须有能力坚持允许你介入并修复他们测试过程出问题的地方，并且分配给你足够时间来修复你所找到的问题。然而，你不是有自己的工作要做吗？所以在这样的情况下，理想的方案是找一个测试咨询师来完成这些事情。

最后一个选择是漠视供货商的测试，假设你收到的子系统是未被测试过的，并且重新测试子系统，尤其是当你认为、或者发现自己面临着供货商的测试不胜任的证据时。我曾经在某个项目中不得不这么做，当时一个供货商卖给我的客户一个缺陷严重的 IMAP 邮件服务器。

最后两个选择有严重的政治意味。供货商不可能接受你的断言（它们的测试是不合格的），很可能会想办法来攻击你的可信度。因为有人做出了使用这个供货商的选择（这个选择很可能是昂贵的）此人也很可能站在供货商的一边来攻击你的论断。需要把数据带到讨

论中来。更好的方法是，看看你能否影响合同的商谈，预先把测试证明和你自己的小组在付费之前做可接受性测试的条件包含进来。事实证明，这样做对供货商会有很奇妙的激励作用。

即使是进行了最佳测试的和最高质量的子系统，也可能在你准备使用它们的特定环境中工作得不好。所以，要做好自己对集成的子系统做集成和系统测试的计划。

3.12 硬件开发和软件开发

除了集成或者新软件，有些项目还包括了客户硬件开发或集成。这会牵涉到全新的硬件，或者以一种和成品部件不同的独特方式制造的硬件。

新硬件或者独特硬件造成了一些有趣的测试复杂性。例如，假若要求软件测试人员测试硬件，他们会发现需要不同的技能。又例如，在硬件上测试软件子系统的需要意味着，当硬件和软件都处于开发阶段，提供给测试可用的可能是不充分的建设中的原型。这些问题需要在计划阶段就被解决。

在硬件设计阶段，你可能想和工程师一起工作，看看你是否能让硬件做的更容易被测试。换句话说，看看你是否能使用简单的工具或技术让内部状态（特别是错误状态）可见。并且，看看能否在硬件中建立一些自动化测试工具和日志工具。也许还需要模拟器或者仿真器。在一个 Internet 应用中，我们有一个隐藏的串口设置，以便让我们通过这个端口启用多种重要的行为日志记录。

不要假设在硬件上测试软件就能充分地测试了硬件。不可能！一定要确保已经完成了合格的硬件测试。如果你的硬件供货商做不到的话，有多个独立的测试实验室可以为你做这样的事情。

话虽这样说，涉及新硬件或者特有硬件的项目是很有意思的。不要被它们吓跑。通过仔细考虑过的计划、雇佣合适的测试团队，以及小心地执行测试，出现的问题是能够被一一解决的。随着你的项目进展，可享受一下目睹某些实实在在的东西逐渐成形的过程。

3.13 测试过程

至此我已经谈论了多种软件开发和维护过程。图 3-7 显示了一个简单的过程图表，能帮助你思考测试过程。[1]

以一个测试者的角色，你可能关注于其中一些活动和过程，而非全部。这依赖于你的角色是如何来定义的，以及你提供怎样的服务。然而，为了做到有效和高效，你需要理解测试过程是怎样工作的，以及它如何适应于整体的项目。

1 测试过程、怎样识别何时测试过程工作得好、怎样处理对测试过程的质疑，以及怎样提高测试过程，所有这些主题我在《关键测试过程》（Critical Testing Processes）中都讨论到了。