测试驱动开发之是与非

撰文 / 莫映

什么是测试驱动开发

众所周知，测试驱动开发（**Test-Driven Development**，以下简称 **TDD**），是敏捷方法中的一项重要实践。一般认为，它是由极限编程之父 **Kent Beck** 所创立的，并且在其经典之作《**Test-Driven Development By Example**》一书中有过详尽的阐述。不过，一如许多软件技术与方法的出现，**TDD** 也可算作是众人智慧的结晶了。这其中，不乏早期与 **Kent Beck** 一起活跃于 **Smalltalk** 社区，而今声名显赫的业界大牛们，如：**Erich Gamma**，**Ward Cunningham**，以及 **Martin Fowler** 等人。有这么多前辈高人作后援，想必 **TDD** 的价值是勿庸置疑的。

典型的 **TDD** 包含如下几个步骤：

1．根据需要快速编写一个测试用例，此时甚至可以是连编译都无法通过的；

2．编写尽可能少的功能代码，以让刚才的测试用例通过；

3．根据需要逐步补充测试用例，此时的测试用例依然是没有通过的；

4．修改功能代码以让新增的测试用例通过，同时也要让原来通过，而今又失败了的测试用例重新通过；

5．对上述功能代码进行重构（有时甚至也包括测试代码），以消除重复；

上述几个步骤的不断往复，便形成了以 **TDD** 方式进行软件开发的基本特征，那便是颇具节奏感的：“**Red**，**Green**，**Refactor**，……，**Red**，**Green**，**Refactor**，……”。笔者将之戏称为“**TDD** 韵律操”，其中的 **Red** 和 **Green** 分别代表了测试用例的失败与通过，这一点想必用惯 **JUnit** 工具进行单元测试的读者一定是谙熟于心的 **J**

TDD 的韵律操向我们喻示了：测试代码的编写，始终是先于被测代码的。由此，**TDD** 也常被人们称为测试先行编程（**Test-First Programming**）。**TDD** 以其测试先行的鲜明特征，往往给人以一种“极端”的印象。被人们亲切的称为“**Uncle Bob**”的敏捷大牛 **Robert C. Martin** 曾经对 **TDD** 的这种“极端”有过进一步的阐释，他将 **TDD** 实践总结为三条原则：[1]

1．You are not allowed to write any production code unless it is to make a failing unit test pass.

2．You are not allowed to write any more of a unit test than is sufficient to fail; and compilation failures are failures.

3．You are not allowed to write any more production code than is sufficient to pass the one failing unit test.

[1] http://www.butunclebob.com/ArticleS.UncleBob.TheThreeRulesOfTdd

三个“**not allowed to write any**”句式层叠递进，无怪乎令初识 **TDD** 者望而却步。毕竟，与传统的软件开发方法相比，**TDD** 有着很大的不同。

从前述 **TDD** 的实践步骤，以及 **Uncle Bob** 总结的三条原则中，我们不难看出，所谓的 **TDD**，其重心不在 **Test**，而在于 **Development**：**TDD** 并非是一种测试技术，而是一种富有新意的软件开发方法。它强调的是，利用测试代码将我们对软件功能需求的意图表达出来，并交替的编写测试代码与功能代码，使这种意图在有可运行的测试代码作为反馈的前提下，逐步得到丰富与精化。这一过程始终贯穿整个开发，也可以说是一种以意图来驱动的开发方法。因此，这种编程方法还有另一个别名，叫做意图编程（**intentional programming**）。[2]

测试驱动开发的优点

为系统改进提供了有效保障

许多人对 **TDD** 的第一印象也许就在于它的“**T**”。是的，至少从传统视角来看，**TDD** 在很大程度上促进和强化了自动回归测试的实践。也许大家都有过类似的体验，当面对一个没有任何测试代码的遗留系统时，如果要想修改其中的程序结构，或是新增代码，需要谨小慎微才行，因为这是一件风险极大的事情。尤其是大规模的改动，稍有不慎便会破坏原有代码的正确性，从而引入 **bug**。而这种 **bug**，也许只能通过后期的集成测试才能被发现。按照经典软件工程的理论，这种 **bug** 的修改成本，往往都是代价高昂的。

始终以 **TDD** 方式进行软件开发的结果便是形成一整套可运行的测试用例集。这使得我们对软件所做的后续修改有了很好的保障。因为每一项功能都有与之对应的测试用例来确保其正确性，所以任何对软件既有功能的修改，或是新功能的增加，都可以通过运行完整的测试用例集获得反馈，使你在第一时间得知改动后的软件质量状况。如此一来，你在修改系统时，也就可以底气十足了。

对改善设计有很大助益

传统的瀑布模型强调设计与编码分属于两个不同的阶段，而在典型的 **TDD** 实践中，设计与编码是穿插进行的，这可以使设计在开发过程中逐步完善和改进，从而避免所谓的“过度设计”（**over-engineering**）。运用 **TDD**，我们可以将有风险的设计决策向后推延，直到有足够的需求表明需要这样的设计，这体现了一种增量式设计的思想。

尽管就 **TDD** 是否就是一种设计手段这一问题还存有争议，但唯有动手实践才能检验设计，这是毫无疑问的。**TDD** 使我们能够在编写功能代码的同时，也能够从调用这些功能代码的角度出发，通过编写测试用例来考查功能代码所提供的接口是否合理。这将有利于我们写出更易于调用的功能代码。任何功能代码都是为他人所用的，那么你就得先站在使用者的角度来考虑接口的定义。

另外，**TDD** 也天生保证了功能代码的可测试性（**testable**）。代码中 **Singleton** 的大量运用，对数据库等外部资源的直接依赖，都会阻碍测试代码的编写。为了让代码更加易于测试，我们需要让代码更多的依赖于接口而非具体实现，亦即“针对接口编程”。如此一来，我们就可以利用 **Mock** 技术，将被测功能与外部资源隔离开来，从而完成针对功能逻辑的独立测试。软件可测试性的需求，无形中使我们达成了软件松耦合的目的。

[2] 见 Robot C. Martin 的《敏捷软件开发：原则、模式与实践》一书

为功能代码提供了很好的“文档”

经典的软件工程对文档的要求是非常严格的，如 **ISO9000**、**CMM** 等，编制大量的卷宗是家常便饭。维护这些文档需要投入大量的时间与精力，成本相当可观。而现实中，要维护文档与代码的同步也往往是一件令人头疼的事情。不仅如此，一些文档时常有流于形式的倾向，没有真正起到有效沟通的目的。很多时候，人们会将其作为应付交差的手段，尔后就会束之高阁。

而敏捷方法则主张：能够正确运行的代码要比详实的文档更加重要。在敏捷团队中，文档不再是洋洋洒洒数百页，而是代之以精简的文字和务实的表达，甚至有人将代码本身也看作是“文档”。按照这样的观点，测试代码就是描述功能代码使用方法的一份天然的“文档”。测试用例为我们提供了如何调用被测类的样例代码。利用这份准确可靠，且能直接运行的特殊“文档”，我们可以从中了解到被测代码的使用细节。并且，这类“文档”不会像普通文档那样，存在与实际代码不同步的问题。因为一旦失去了同步，自动或手工运行的测试用例集便会在第一时间告知我们。一句题外话，多数优秀的开源框架，如：**Spring Framework**，其单元测试集往往就是学习框架使用极好的一手资料。

在一定程度上可以代替程序调试的工作

调试往往需要具备较高的技巧，并且通常要解决的往往是由多方因素促成的复合问题，而调试所依赖的运行时环境，也往往是错综复杂的。例如：**web** 应用中的一个配置文件错误，其外在表现往往并非那么直观，而我们需要待到应用部署成功，服务器启动完毕以后，方能对之进行调试。这一过程费时又费力。

而 **TDD** 则遵循着敏捷方法中的“简洁性”（**Simple**）原则[3]，每次只针对一个任务编写测试，并让测试通过。这种持续的小幅迭代前行，能让代码质量得以显著的提升，因而 **bug** 数量也会随之减少。此外，由于测试用例的执行环境与前述调试所依赖的运行时环境相比而言简单了许多，因此要解决的问题也会单一许多，容易许多。即便遇到了 **bug**，我们依然可以通过追加测试用例的方式将问题域缩小到一个相对简单的范围，加以解决。测试运行期间，我们还可以结合调试的手段，此时的问题域已经简单化，因此调试的成本也会大幅减少。追加的测试用例还有一个好处，那就是当同一问题再次出现时，测试用例即刻就会给我们以反馈。这一点也引出了调试的一个问题：调试是手工且不可重复的，而测试用例则是自动可回归的。

可以显著增加开发者的信心并赢得他人的信任

由于任何意图都是以测试，以及令测试得以通过的功能代码来呈现的，因此只要通过了所有测试用例集，那么任何体现于代码背后的意图都是切实可靠的，同时也是优质无误的（当然，这一点还有赖于测试用例集的完善程度）。这便是一切凭事实说话的态度，给人以一种非常踏实的感觉。有了强力的测试用例集作为后援，开发者在交付功能代码时便会自信满满。同时，高质量的成果也会赢得团队内成员，乃至最终客户的信任。

在笔者所经历的一个失败项目中，由于代码质量不过关，使系统 **bug** 频频，最终导致后方开发人员与现场实施人员产生了彼此间的不信任，无形的团队内隔阂使沟通受阻，一度令项目几近崩溃的边缘。而扭转这样的局势则花去了大量的成本：不计其数的加班，强化的双向沟通，加之大量的返工。即便如此，客户关系仍然颇为紧张，项目实施倍感被动。

[3] 极限编程中的核心价值观之一，除此以外，其他价值观还包括：沟通、反馈、勇气、尊重

另一个正面教材则来自于笔者所在的产品团队（听起来有些自卖自夸之嫌，因为一时没有找到合适的例子，还请读者见谅）。正因为代码的关键逻辑有严格的测试用例作为保障，并且测试用例的设计也考虑了包括边界条件在内的各种情形，因此每当 **QA** 将集成测试过后的 **bug** 清单提交给我时，我都可以镇定自若的为她指出，这是使用不当造成的问题，而非代码本身的错误。而每当团队进行内部代码复查之时，我也会十分坦然的将自己的代码公之于众，当然，也包括测试代码。

对测试驱动开发的疑虑

许多人对 **TDD** 的最大质疑在于其开发效率上，这种质疑并非没有道理。天下没有免费的午餐，纵然 **TDD** 有百般好处，编写测试用例依然是要花费时间的。一些程序员不愿意写测试用例就是因为担心严格按照 **TDD** 行事会导致效率低下。他们更习惯于迅速切入正题，直截了当的完成功能代码。也有人会稍做妥协，设想在功能代码编写完毕之后再补上测试，或者直接依赖于手工测试，因为他们还担心功能代码编写过程中的不断重构会导致不断同步测试代码的开销。

关于这一点，**Kent Beck** 在评价自己时曾说过一句经典的话：**I am not a great programmer. I am a programmer with great habit**。依笔者看来，这并非谦虚，而是实情。**TDD** 就是要求开发者摒弃旧有的开发习惯，养成新的习惯。所谓习惯成自然，一旦 **TDD** 成为习惯，一切就变得自然，开销也就可以容忍了。**Erich Gamma** 将这种习惯的养成称之为“**test infected**”。意指，习惯了 **TDD** 的开发者们是很难再回到原来的开发模式的，那样会另他们因为失去了严格的契约保障而缺乏安全感。

此外，就手工测试而言，如果测试用例只运行一次，那么自然更划算些。但是在典型的敏捷团队里，测试用例往往是需要频繁运行的，亦即需要具备可回归性。此时，让计算机去完成重复性的工作应该是明智之举。

至于后补测试用例，倒不失为一个折中的办法。不过，依笔者的个人体验，这种设想有时也未必能够兑现。所谓趁热打铁，待到人走茶凉之时，相信可以有很多借口让自己逃避测试用例的编写。

还有一些人担心实践 **TDD** 后的结果，会导致需要维护一套庞大而沉重的测试用例集。这种担心倒不是多余的。在笔者的 **TDD** 实践中，就曾有过这样的经验教训：庞大的 **StrutsTestCase** 自动测试集，每个 **test** 方法都会加载大量的 **struts** 及 **spring** 配置文件，再加上测试代码依赖于诸如数据库、**LDAP** 之类的外部资源，导致完整运行测试用例集所需的时间有数十分钟之多。对于一个讲求快速反馈的敏捷团队而言，这样的开发效率是难以容忍的。为了避免这种情况，需要一些技巧来保障。例如：使用最简单的测试工具，尽可能减少不必要的外部依赖，还有一点很重要——保持简洁的设计，因为繁冗的测试代码往往也就意味着潜藏在功能代码中的坏味道。

最后，教科书上的 **TDD** 范例只是供人参考用的，大可不必对一些以常规视角而言有些极端的做法过于敏感，不妨将之看作是矫枉过正，何况实践中我们也可以对之加以变通。比如，就“**Red**，**Green**，**Refactor**”韵律操的节奏而言，当自己有足够的把握时，自然可以加快步伐，跳过一些自认为多余的测试代码编写工作，而一旦遭遇挫败，此时再放慢步伐也不迟。总之，节奏的把握尽在自己的手中。

一言以蔽之，唯一能够掌握 **TDD** 的方法就是动手实践，在实践中慢慢体会，而实践 **TDD** 首先就是要具备尝试的勇气，这也是极限编程中的核心价值观之一。曾在论坛中见到过人们讨论有关 **TDD** 的一些问题，例如：对私有方法是否要先写测试，是否要为测试代码编写测试，凡此种种。

其实依笔者的愚见，这貌似有些舍本逐末，忘记了 **TDD** 的根本目的。从现在做起，先接受，多实践，相信你一定会从中受益匪浅，说不定就会“**test infected**”**J**

如何有效实践测试驱动开发的一些建议

在没有 TDD 习惯的团队中进行实践

这样的团队多半面对的是遗留系统，此时 **TDD** 一般是很难入手的，因为问题域太复杂，且原有代码多半不具备可测试性，因此很容易让人知难而退。此时，可以从一些相对独立的新加特性入手，使用最简单的 **JUnit** 测试工具，逐步实践，细细体会，待收获成功的喜悦之后再继续扩大战果。被测功能最好不要涉及 **UI** 或数据库，最好是具备明显算法特征的逻辑，因为这样的逻辑非常适合尝试 **TDD**。

此外，从 **bug fix** 入手也不失为一个好方法。在每次遇到 **bug** 时，首先尝试先写测试用例，然后再修改 **bug**。虽然这并非纯粹意义的测试先行，但也是一种很好的实践手段，并且当 **bug** 复现时，这一测试用例更是显现出价值，因为它能在第一时间告知 **failure**。

与结对编程的结合使用

在敏捷方法中，结对编程也是一项颇受争议的实践，有不少人质疑其对开发效率产生的负面影响。不过，恰如其分的运用结对，对于团队内的知识传递与有效沟通将会起到积极的作用。有一种特别的结对编程方式，被称作“乒乓式”结对（**PING-PONG Programming**）。结对时，两位程序员各自有明确的分工，其中一位负责编写测试代码，而另一位则负责编写功能代码，以让测试通过，二者的职责轮换交替，持续往复，就像乒乓球竞技中的攻防转换。

事实上，“乒乓式”结对就是单人 **TDD** 的结对版。**Martin Follower** 在形容 **TDD** 实践时将其中的新功能增加与既有功能的重构比做交替的戴两顶帽子，而同一时刻只能戴其中一顶。其实，编写测试代码与编写功能代码也好比是两顶帽子。测试代码的编写是站在需求的角度，从意图出发；而功能代码的编写则是站在实现的角度，完成意图，同一时刻只有一种角色在起作用。如果不习惯集两种角色于一身，那么就试着做一下分解练习，尝试一下“乒乓式”结对。

与持续集成的结合使用

随时随地的运行 **TDD** 所积累的测试用例集，能够以最敏捷的反馈方式告诉我们，目前代码的质量状况。而持续集成则使这一反馈变得更加制度化，更加可靠。通过每日构建，或者更短周期的自动构建，我们可以定期获得有关系统当前状况的有效反馈，使得因集成而导致的错误能够尽早的暴露。反观持续集成本身，若要另其发挥更大的价值，也必然要积累更多的测试用例才行。否则，若仅是编译通过，则无法找出大量的运行期错误。此时，这一反馈当前代码质量状况的自动化机制的效用也就会大打折扣。

此外，持续集成对测试用例的彼此独立性也提出了要求。我们在编写测试用例时，需要注意测试用例间的彼此依赖，不能因为一个测试的失败而影响了后续测试的执行，这一点也很重要。

善用 Todo List

经典的 **TDD** 实践范例中，人们往往会在开始编写测试代码前先制订一个 **todo list**，其上列举了待实现的需求要点。随着实践的进展，这个 **todo list** 也会逐渐调整：删去已达成的条目，调整原有的条目，又或增加新的条目。这样的 **todo list** 也许是记录在一张普通的纸上，又或是电子的文本文件，不论以何种形式，其主旨在于：有效的对当前任务进行管理和跟踪。**Todo list** 似乎与 **TDD** 本身并无多大的关联性，因此初试 **TDD** 者往往也会忽略其价值。但依笔者的切身体会，**todo list** 对成功实践 **TDD** 是很有裨益的，因为它为 **TDD** 指引了前进的方向。其实，要将一个功能需求降解成若干个 **todo**，也是需要认真思考的，这是一种思维的锻炼。

结语

TDD 是敏捷方法中技巧性较高的一项实践，它代表着一种截然不同的软件开发方法。要习惯 **TDD** 的思维，需要多实践，多体会。一旦掌握了其中的要领，相信一定会另你获益匪浅。祝愿你早日 **test infected J**