**TDD**

**定义**

TDD（测试驱动开发）是一种不同于传统软件开发流程的新型的开发方法。它要求在编写某个功能的代码之前先编写测试代码，然后只编写使测试通过的功能代码，通过测试来推动整个开发的进行。这有助于编写简洁可用和高质量的代码，并加速开发过程。[1]

**测试是如何驱动开发过程的**

测试驱动开发起源于XP法中提倡的测试优先实践[2]。测试优先实践重视单元测试，强调程序员除了编写代码，还应该编写单元测试代码。在开发的顺序上，它改变了以往先编写代码，再编写测试的过程，而采用先编写测试，再编写代码来满足测试的方法。在XP1998年出现之前，几乎没有关于让小的增量式的自动化单元测试驱动软件开发的设计过程概念。尽管缺乏公开的文档，许多开发人员可能非正式地使用了测试的第一种方法。肯特·贝克声称“他在还小的时候就在一本关于编程的书上学到了测试优先。它说，你写的程序就是获取输入然后输出结果，直到得到你期望的结果。”[3] 也许是最知名的敏捷方法，XP常常与其他敏捷方法如Scrum组合。XP建议使用TDD作为开发高质量的软件的一个组成部分。具有高度纪律性的TDD和简单的，轻量级的敏捷过程自然引起了一个有趣的冲突。潜在的TDD使用者经常表示关注编写和维护单元测试的时间和成本。虽然Beck承认没有必要为所有代码进行自动化单元测试，但他坚持认为，没有TDD，XP不能工作因为它使整个开发过程紧密联系在一起。[4]这种方法在实际中能够起到非常好的效果，使得测试工作不仅仅是单纯的测试，而成为设计的一部分。为什么这么说呢？

在编写程序之前，每个人都会先进行设计的工作。可能有些人的设计比较正式，绘制模型，编写文档。有些人的设计只是存在于脑海之中。且不论设计是精细还是粗糙，你都为随后的编码活动制定了一个标准。这个标准的明确程度和你的设计的细致程度有关。但应该承认，这个标准是不够细化的。因为你的设计不可能精细到代码级的程度。而标准不够明确的则会产生一些问题，例如，在编写代码的过程中，你还可能会发现原先的设计出现问题，从而中途改变代码的编写思路。这将会导致成果难以检验，进度难以度量。

既然以设计为导向的标准不够明确，不够具体。那什么样的标准才是合适的呢？只能是代码。因为代码是最明确、最具体的。所以测试优先的本质其实是目标管理。编写测试代码其实是在制定一个小目标。这个小目标非常的明确，它规定了你需要设计的类、方法。以及方法需要满足的结果。这些目标制定完成之后，你才开始编写代码来达成该目标。测试的目标要比设计的目标粒度更小，但是成本上却更为经济。其原因有四：

1. 细粒度的设计需要花费大量的成本，虽然有工具提供了代码自动生成的功能，但结果往往难以令人满意。所以，设计如果要做到和测试相同的粒度，成本不菲，如果粒度不够细，指导的意义又不够。
2. 减轻了测试的工作量。无论是否进行设计工作，测试工作都是不可避免的，先进行单元测试，可以减少后续的测试工作量。
3. 采用测试优先的过程中，设计的粒度较大。因为测试可以实现一部分的设计工作。这样，设计上可以节省一些工作量。例如，你不再需要将类图细化到每个方法。
4. 在编写测试代码上花费的成本，会在回归测试上得到回报。自动化测试的最大好处就是避免代码出现回归。两相权衡，编写测试的代价其实不高。

你也许会说，我既不进行如此精细的设计，也不事先编写测试代码，这样的成本不是最低吗？请注意，我的前提是在讨论高质量的软件设计。在一些规模较小或是开发人员能力极强的项目中，确实可以如此办理。但是对于强调质量的大项目，这种处于混沌状态的开发思路是不可取的。

测试优先是软件开发中一种细粒度的目标管理方法，通过明确的目标，推动软件开发的进行。在业界中，采用测试作为评价软件标准的做法是非常常见的。例如，sun公司就专门设计了测试软件，对各个实现J2EE规范的产品进行测试。使用测试作为规范的最大好处就是明确、具体。同时，平均来说，80%的专业的开发者认为TDD是一个有效的方法并且78%的人相信该方法能够提高开发者的生产率[5]。

使用测试代码建立目标，编写代码完成测试目标，再制定下一个目标，如此循环，构成了测试驱动开发的工作流程。

**测试的分类**

**单元测试**

单元测试是典型的对代码逻辑的黑盒测试。在测试驱动方法中，不太强调白盒测试（绝大多数的白盒测试都是通过评审进行的）。这样做的好处是关注接口胜于关注实现，这是一种分析复杂软件的有效办法。

单元测试是开发人员的职责。一般来说，测试的编码最好由不同人来负责，避免出现盲点，以提高测试的有效性。但是单元测试的粒度很小，如果进行分工，沟通的成本会相当高。此外，采用测试优先的实践，对测试进行适当的培训，也能够有效的降低单个人的盲点范围。

单元测试可以加入到小组日构建中，也可以不加入。如果不加入，那么需要有一种机制来管理单元测试活动。

**集成测试**

集成测试的粒度和测试的范围要比单元测试大。就拿数据库测试来说，现在需要对一个业务对象进行测试，它需要用到持久化机制。在单元测试中，我们将不涉及数据库而单独对业务对象进行测试；但是在集成测试中，我们需要将数据库的数据一致性也纳入进来，所以测试包括数据库数据的建立，测试业务方法，使数据库恢复原状。

集成测试应该是日构建的重要组成部分，即日构建标准中的测试标准。最好将集成测试交给QA部门负责。QA部门的精力可以放在使用或编写一些工具（Cactus就是典型的集成测试工具），建立标准的测试数据，安排测试计划等活动上。

**接受测试**

有时候很难区分集成测试和接受测试。接受测试的关注点是用户。用户通过将数据输入系统来观察系统的输出。所以，了解用户的需要并将用户的需要转换为接受测试是接受测试中最关键的工作。接受测试处于测试过程的最后环节，是判断软件是否满足用户需求的试金石。毫不例外。接受测试也应该是自动化的。例如，HttpUnit就是一个自动化接受测试的工具。另外，很多的专业测试工具提供的自动化的脚本测试工具也属于这个范畴。

**TDD的优势**

TDD的基本思路就是通过测试来推动整个开发的进行。而测试驱动开发技术并不只是单纯的测试工作。

需求向来就是软件开发过程中感觉最不好明确描述、易变的东西。这里说的需求不只是指用户的需求，还包括对代码的使用需求。很多开发人员最害怕的就是后期还要修改某个类或者函数的接口进行修改或者扩展，为什么会发生这样的事情就是因为这部分代码的使用需求没有很好的描述。测试驱动开发就是通过编写测试用例，先考虑代码的使用需求（包括功能、过程、接口等），而且这个描述是无二义的，可执行验证的。

通过编写这部分代码的测试用例，对其功能的分解、使用过程、接口都进行了设计。而且这种从使用角度对代码的设计通常更符合后期开发的需求。可测试的要求，对代码的内聚性的提高和复用都非常有益。因此测试驱动开发也是一种代码设计的过程。

开发人员通常对编写文档非常厌烦，但要使用、理解别人的代码时通常又希望能有文档进行指导。而测试驱动开发过程中产生的测试用例代码就是对代码的最好的解释。

快乐工作的基础就是对自己有信心，对自己的工作成果有信心。当前很多开发人员却经常在担心：“代码是否正确？”“辛苦编写的代码还有没有严重bug？”“修改的新代码对其他部分有没有影响？”。这种担心甚至导致某些代码应该修改却不敢修改的地步。测试驱动开发提供的测试集就可以作为你信心的来源。

当然测试驱动开发最重要的功能还在于保障代码的正确性，能够迅速发现、定位bug。而迅速发现、定位bug是很多开发人员的梦想。针对关键代码的测试集，以及不断完善的测试用例，为迅速发现、定位bug提供了条件。

**原理**

测试驱动开发的基本思想就是在开发功能代码之前，先编写测试代码。也就是说在明确要开发某个功能后，首先思考如何对这个功能进行测试，并完成测试代码的编写，然后编写相关的代码满足这些测试用例。然后循环进行添加其他功能，直到完全部功能的开发。

我们这里把这个技术的应用领域从代码编写扩展到整个开发过程。应该对整个开发过程的各个阶段进行测试驱动，首先思考如何对这个阶段进行测试、验证、考核，并编写相关的测试文档，然后开始下一步工作，最后再验证相关的工作。

在开发的各个阶段，包括需求分析、概要设计、详细设计、编码过程中都应该考虑相对应的测试工作，完成相关的测试用例的设计、测试方案、测试计划的编写。这里提到的开发阶段只是举例，根据实际的开发活动进行调整。相关的测试文档也不一定是非常详细复杂的文档，或者什么形式，但应该养成测试驱动的习惯。

**原则**

测试隔离。不同代码的测试应该相互隔离。对一块代码的测试只考虑此代码的测试，不要考虑其实现细节（比如它使用了其他类的边界条件）。

一顶帽子。开发人员开发过程中要做不同的工作，比如：编写测试代码、开发功能代码、对代码重构等。做不同的事，承担不同的角色。开发人员完成对应的工作时应该保持注意力集中在当前工作上，而不要过多的考虑其他方面的细节，保证头上只有一顶帽子。避免考虑无关细节过多，无谓地增加复杂度。

测试列表。需要测试的功能点很多。应该在任何阶段想添加功能需求问题时，把相关功能点加到测试列表中，然后继续手头工作。然后不断的完成对应的测试用例、功能代码、重构。一是避免疏漏，也避免干扰当前进行的工作。

测试驱动。这个比较核心。完成某个功能，某个类，首先编写测试代码，考虑其如何使用、如何测试。然后在对其进行设计、编码。

先写断言。测试代码编写时，应该首先编写对功能代码的判断用的断言语句，然后编写相应的辅助语句。

可测试性。功能代码设计、开发时应该具有较强的可测试性。其实遵循比较好的设计原则的代码都具备较好的测试性。比如比较高的内聚性，尽量依赖于接口等。

及时重构。无论是功能代码还是测试代码，对结构不合理，重复的代码等情况，在测试通过后，及时进行重构。关于重构，我会另撰文详细分析。

小步前进。软件开发是个复杂性非常高的工作，开发过程中要考虑很多东西，包括代码的正确性、可扩展性、性能等等，很多问题都是因为复杂性太大导致的。极限编程提出了一个非常好的思路就是小步前进。把所有的规模大、复杂性高的工作，分解成小的任务来完成。对于一个类来说，一个功能一个功能的完成，如果太困难就再分解。每个功能的完成就走测试代码－功能代码－测试－重构的循环。通过分解降低整个系统开发的复杂性。这样的效果非常明显。几个小的功能代码完成后，大的功能代码几乎是不用调试就可以通过。一个个类方法的实现，很快就看到整个类很快就完成啦。本来感觉很多特性需要增加，很快就会看到没有几个啦。你甚至会为这个速度感到震惊。（我理解，是大幅度减少调试、出错的时间产生的这种速度感）

**总结**

测试驱动方法不是一个简单的方法论，它也不会和任何的方法论进行竞争。事实上，无论你的组织采用何种方法或过程，都可以从测试驱动中获利。因为它强调的是质量文化。把测试看作一项核心工作，测试同样需要重构，以及必须的文档。固定测试的目录组织和包组织。例如，一种较好的组织测试的方法是采用和源代码同样的包名，但处于完全不同的目录中。使测试成为日创建的核心步骤。测试是所有人的事情，而不仅是QA的事。

[1] 必应学术，<http://cn.bing.com/academic/profile?id=4478048&v=fos_preview>

[2] David Janzen,Test-Driven Development: Concepts, Taxonomy,and Future Direction,IEEE Computer Society,1-2

[3]. K. Beck, “Aim, Fire,” IEEE Software, Sept./Oct. 2001, pp. 87-89.

[4] K. Beck, Extreme Programming Explained: Embrace Change, Addison-Wesley, 1999.

[5] Boby George, Laurie Williams,An Initial Investigation of Test Driven Development in Industry,4-5