测试驱动开发

现在的很多公司，包括 Google 和我现在的公司 Coverity，都喜欢一种“测试驱动的开发”（test-driven development）。它的原理是，在写程序的时候同时写上自动化的“单元测试”（unit test）。在代码修改之后，这些测试可以批量的被运行，这样就可以避免不应该出现的错误。

这不是一个坏主意。我在 Kent 的编译器课程上也使用了很多测试。它们在编译器的开发中是不可缺少的。编译器是一种极其精密的程序，微小的改动都可能带来重大的错误。所以编译器的项目一般都含有大量的测试。

然而测试的构建，应该是在程序主体已经成形的情况下才能进行。如果程序属于创造性的设计，主体并未成形，过早的加入测试反而会大幅度的降低开发效率。所以当我给 Google 开发 Python 静态分析的时候，我几乎没有使用任何测试。虽然组里的成员催我写测试，但是我却知道那只会降低我的开发效率，因为这个程序在几个星期的过程中，被我推翻重来了好几次。要是我一开头就写上测试，这些测试就会碍手碍脚，阻碍我大幅度的修改代码。

测试的另一个副作用是，它让很多人对测试有一种盲目的依赖心理。改了程序之后，把测试跑一遍没出错，就以为自己的代码是正确的。可是测试其实并不能保证代码的正确，即使完全“覆盖”了也是一样。覆盖只是说你的代码被测试碰到过了，可是它在什么条件下碰到的却没法判断。如果实际的条件跟测试时的条件不同，那么实际运行中仍然会出问题。测试的条件往往是“组合爆炸”的数量级，所以你不可能测试所有的情况。唯一能可靠的方法是使用严密的“逻辑推理”，证明它的正确。

当然我并不是让你用 ACL2 或者 Coq 这样的定理证明软件。虽然它们的逻辑非常严密，但是用它们来证明复杂的软件系统，需要顶尖的程序员和大量的时间。即使如此，由于理论的限制，程序的正确性有可能根本无法证明。所以我这里说的“逻辑推理”，只是局部的，人力的，基本的逻辑推理。

很多人写程序只是凭现象来判断，而不能精密的分析程序的逻辑，所以他们修改程序经常“治标不治本”。如果程序出问题了，他们的办法是看看哪里错了，也不怎么理解，就改一下让它不再出错，最多再把所有测试跑一遍。或者再加上一些新的测试，以保证这个地方下次不再出问题。

这种做法的结果是，程序里出现大量的“特殊情况”和“创可贴”。把一个“虫子”按下去，另一个虫子又冒出来。忙活来忙活去，最后仍然不能让程序满足“所有情况”。其实能够“满足所有情况”的程序，往往比能够“满足特殊情况”的程序简单很多。这是一个很奇怪的事情：能做的事越多，代码量却越少。也许这就叫做程序的“美”，它跟数学的“美”其实是一回事。

美的程序不可能从修修补补中来。它必须完美的把握住事物的本质，否则就会有许许多多无法修补的特例。其实程序员跟画家差不多，画家如果一天到头蹲在家里，肯定什么好东西也画不出来。程序员也一样，蹲在家里面对电脑，其实很难写出什么好的代码。你必须出去观察事物，寻找“灵感”，而不只是写代码。在修改代码的时候，你必须用“心灵之眼”看见代码背后所表达的事物。这也是为什么很多高明的程序员不怎么用调试器（debugger）的原因。他们只是用眼睛看着代码，然后闭上眼，脑海里浮现出其中信息的流动，所以他们经常一动手就能改到正确的地方。