

Test Driven Development

Test Driven Development, or TDD to its friends, is an approach to software development which attempts to put the cart before the horse. What are the pluses and minuses of the approach and does it make sense in our pressured software world? Read on to see if you agree with my thoughts on this.... and please email me if you don't!

So what is TDD?

Basically it is the idea that we write a set of executable tests for a "smallish" unit of software and then attempt to develop our unit by adding functionality to it and repeatedly running the tests until they all succeed.

So the concept is that the specification of our software is embedded in executable tests. If these tests are passed then the software conforms to the requirements and we are done! This is rather nicer than the use of (often) sloppy English to specify functionality. When you can compile it there is nowhere to hide!

However, if you feel a little uncomfortable with this you probably have good reason.. There was a pretty famous Computer Scientist, Edsgar Dijkstra, who stated that "testing can show the presence of bugs but not their absence". Given this statement, what advantages does TDD bring to the table? The tests will, of necessity, be incomplete and for them to be of value, they will have to be very large, certainly several times the size of the component under test. It is also possible that the Tests will become a drain on the support activities because they must be synchronised with any changes in the component.

Does this mean that TDD is a waste of time or a fake? Not at all! Its benefits, which are significant, lie in other areas. Below I have attempted to bring these together.

TDD in Context

We should look at the alternative to TDD. The alternative is to produce a piece of code as a result of Requirements Analysis and Design. Initially it may have nothing more than an interface behind which the code will be provided to deliver the logic. Following that, if we are lucky, a set of unit tests will be written to test the component.

The first clue that we have a problem here is that the majority of tests will initially fail and this failure is usually down to a misunderstanding of how the component should work! This is not reassuring when we consider that these tests are written by the developer of the component to be tested. If they do not understand its use then who does? The tests then get modified to fit the component and on we go. The tests being made to fit the component often invalidate the tests as we have a chicken and egg situation. This cannot bring a warm feeling to any professional software developer.

Let's consider a TDD approach. Here we incrementally develop tests and the code to be verified against the tests.... in fact a microcosm of iterative development. The tests are taken as *specification* for the development unit as well as tests which have to be passed. The iteration causes (allows) more thought to be applied to the purpose of the component and how it should work. The benefits obtained here exactly mirror the benefits obtained by iterative and incremental development. Small steps are being taken, with plenty of feedback, in meaningful, executable, terms. This is not a paper exercise, and providing that the tests are kept synchronised with the test-component and its various versions, we have a major asset to the project.

The tests themselves are usually written in the same language as the component being tested and themselves are a major advantage during regression testing and maintenance activities.

We need to focus a little on the actual tests being written. Some tests represent an instance of correct use of the component. Tests which represent the same instance of use test nothing more than whether the component gets fatigued from running tests.

Other tests are written to verify the behaviour of the component under fault or exceptional conditions. The total test-span is important. Sadly the total number of test-instances is usually massive and cannot be approached. At this point some software-intelligence is needed to identify where testing should be focussed.

TDD as described will not satisfy Dijkstra. There still has to be attention to correctness of method, validity of input, well defined outputs and so on. These latter issues are there for any design activity and have to be resolved. TDD brings its own benefits on top of this by providing an opportunity to incrementally develop against an executable and unambiguous specification.

TDD in an XP environment brings even more benefits because in addition to the specification view of tests we also get peer pressure to provide clean and effective code. TDD without XP is missing a lot of the synergy available (almost) for free.

Anything Else?

Well yes there is and it is another very important synergy. When we talk about TDD we usually mean that one of the standard Test environments will be used, perhaps JUnit or NUnit. These very useful tools provide us with standard environments for developing, running and monitoring our tests and are so successful that any modern programmer should be completely at home in them.

Now it is pretty easy to write and run unit tests, at least the initial set. Lately on my courses I have been pushing the implicit requirement that economic unit testing of components should be possible. That means, roughly, that the cost of testing a component should always be so low that we can afford to develop and perform thorough tests. That in itself is a benefit. However, there is another benefit, and to my mind it is as significant as any of the others. The "economic" testing requirement essentially means that the component under test must easily and economically be able to run in two environments, the production environment and the test environment. This causes / forces the developer to pay attention to design areas that were often skimped, perhaps because they look a little abstract. Examples are cohesion, coupling, interface,

data hiding and so on, which are all techniques which when applied properly make economical testing possible. And as a spin-off benefit we get much better structured programs.

A lot of the code that I write gets shown to audiences. Consequently it tends to be that much better structured, commented and so on. Recently I decided to make an example of a piece of code that runs (economically) under JUnit as well as in its production environment. It was not until I had redesigned the original interfaces from the bottom up that I succeeded in this task. The result was an inherently better structured piece of work. I should mention that in over 30 training classes where the original code was presented, no one had ever commented on the structure/interfaces of this version of the code!

The point here is that what look well-structured in the abstract, can turn out not to be so when a practical application is attempted! And this is what I think is one of the major spin-off benefits of TDD, which forces ALL code to be runnable in the two environments. Once you add the "economic" requirement, you should see much superior interface design surfacing.

Just to round off this whole thing a short story from the 1960s! I worked with a mathematician in a software development group. This experienced guy was given a new project and made a complete pain of himself because, instead of getting on with the coding, he spent his time creating or attempting to create test data and coming up with difficult questions about how the product should actually work. At the time everyone thought he was eccentric, but now

In Summary

TDD leads us to active specification through demonstration. It is most successful when combined with traditional problem-avoiding techniques. The tests developed under TDD are a key project asset and must be maintained and delivered in parallel with the components which they test. The requirement to be able to execute components under more than one environment leads to better structuring and design of these components and turns what was an abstract goal into something where we see the tangible benefits of good design.

References:

eXtreme .NET Dr. Neil Roodyn Introduction to TDD.	ISBN 0-321-30363-6 Ambler http://www.agiledata.org/essays/tdd.html
Test Driven Development: By Example	Beck: ISBN: 0321146530

About the Reviewer

George Brooke is a Technical Trainer / Consultant as well as being a PRINCE2 Approved Trainer. As Head of Software Development at Siemens Nixdorf he managed solutions development for the Financial, Retail, Hotel and Energy markets for UK/European departments and customers worldwide, from concept through to delivery and roll-out. He runs a training and consultancy company based in Cambridge UK focusing on OO design with UML, programming and implementation techniques and IT project management .