BETTER SOFTWARE FASTER

Information Sheet 1

Properties of Good Tests

TDD & BDD – Design Through Testing

Info 1.0

The Properties of Good Tests

TDD is an automated testing approach. When we get it right, good tests are extremely valuable tools that guide our development. What constitutes “good” when it comes to these kinds of tests?

Good Tests Are...

Understandable – Our tests should describe what they are testing so that we can understand the goals of our software: Focused on the behaviour of the system, not a specific implementation.

Maintainable – We’d like our tests to act as a defence of our system, breaking when we want them to, but also remaining true as we change our system. So they need to be maintainable so that they are easy to change, without losing their intent.

Repeatable – Our tests should be definitive, so should always pass or fail in the same way for a given version of the software that they are testing. We want our tests to be reliable: to give us the same result every time they run, wherever, or whenever, they are run.

Atomic – Tests should be isolated and focus on a single outcome. We want tests with no side-effects and so able to achieve definitive results whatever else is going on. Independent of other tests and resources, we can run our tests in parallel, or in any order.

Necessary – We don’t need to create “tests for test sake”. Our aim is to guide development from tests, so we should create tests that help us, are necessary, to guide our development choices.

Granular – Tests should be small, simple and focused, and assert a single outcome of the code that they are testing. Giving us a clear Pass/Fail result, and a clear indication of what the problem is when they fail, that do not require interpretation.

Fast – We will use tests as a tool to guide our development, so we will end up with lots of them. That means we need them to be efficient and give us results quickly – our tests need to be fast. Easiest way to do that is to keep them simple and focused.

In Test Driven Development we write the test before writing code. This has a huge impact on our ability to achieve the properties that we value in good tests.

Here we compare the impact of writing the test after we have written the code (Unit Testing) vs writing the test before we write the code (TDD) on our ability to achieve “good” tests.

The Properties of Good Tests

Understandable

Unit Testing

When Tests are written after the code that they test, they are led by a specific implementation and so tend to be less abstract. As a result they are often more difficult to understand.

TDD

Writing the test before the code forces you to think about what you are really trying to achieve. The test then expresses that behavioural desire. This leads you to think more from the perspective of a user of your component, class or method. It leads you to design outside-in, instead of inside-out. This means that the test tends to be a clearer description of the desired outcome.

Maintainable

Unit Testing

When Tests are written after the code that they test, they tend to focus on testing the solution that is there. This is a problem when you come to change the solution – reworking or refactoring the implementation can break the test. This is as a result of the test being tightly-coupled with the code under test. It is always harder to write maintainable tests after the code has already been written.

TDD

Tests written first are forced to be more loosely-coupled with the solution – unless you cheat and design the solution in your head first! This means that the solution can change, but the test can remain valid. The tests are more like "executable specifications" than tests. The tests only change when the specification (aka requirements) change.

Repeatable

Unit Testing

Most tests written after the code are hard to run over and over again, and hard to run in parallel with other tests. In part this is because these tests are often much more complex to set-up and so more tightly-coupled to the specifics of implementation and environment.

Repeatable

TDD

Writing the test before designing the solution makes it clear when something is a pain to setup for the test. This means that we tend to abstract things to make them easier to set-up. This means that we can fake quite a lot of the outside world and so better control the state of the system-under-test.

Atomic

Unit Testing

Tests written after the code are often more complex to set-up. Because we did a lot of work to set things up, we want to take advantage of all of that work. As a result we tend to write much more complex tests that test a lot of different things. So these tests are usually not atomic.

TDD

If a test is easy to set-up and the outside world is fake, then there is less use of shared state and/or resources and so tests tend to be easier to run over and over again and easier to run in parallel with other tests. They are much more atomic.

Necessary

Unit Testing

Tests written after tend to focus on the solution rather than the requirements. This means that we tend to test for what we have already thought of when designing the code. This means that there is a significant danger of missing some necessary test cases.

TDD

If we are strict, and only write code in the SUT as demanded by a test, it means that we must test for every desirable behaviour of the SUT. We can still miss something, we are only human. On the whole though we won't miss a test case as much as miss the need for some behaviour of the system. This means that every test is necessary, but we may forget some necessary tests (and their associated behaviour in the SUT).

Granular

Unit Testing

Tests written after the code are nearly always more complex and, if the code evolves, tend to get even more complex over time! The whole point of Unit Testing is granular tests, but tests written after the fact tend to be broad and complex, both in scope and often set-up too :(

TDD

If we are strict, and only write code in the SUT as demanded by a test it means that we must test at a fine-grained level in terms of desirable behaviours of the SUT. This means that tests are, very granular. It is common for there to be more test code than code in the SUT!

Fast

Unit Testing

When writing the test after the code, there is nothing to drive towards fast and efficient tests. It is not impossible, but it is rare to see such tests be efficient. One of the reasons for this is the poorer granularity. Since the code is doing more and the tests are more complex, the whole combination is significantly slower: often using real system resources, files, databases, queues, and so adding even more to the complexity, and so reducing performance.

TDD

The TDD cycle is VERY interactive. When developing with TDD most developers are rarely more than a few minutes, often seconds, from the last test-run. This applies a pressure to make the process efficient. We want instant feedback, so become intolerant of slow tests. The separation of concerns that "outside-in" development encourages means that we tend to write code that is very loosely-coupled with the surrounding infrastructure and so we can fake that infrastructure. TDD tests don't talk to real files, DBs, WebServers, etc. So they tend to be much faster than test-after Unit tests.

Consequences of "Test-After" Unit Testing

Writing tests after the code is better than not having any tests at all, but it has some serious consequences…

Tests Tend to be coarser-grained – Big complex tests

Tests are often more intermittent – Complexity breeds complexity

Tests tend to be closely coupled with implementation – Over time development pace slows