Test-Driven Development, Acceptance Test-Driven Development, and Behaviour-Driven Development

[Note: This is an excerpt from Agile Testing Foundations: An ISTQB Foundation Level Agile Tester Guide*, by Rex Black, Marie Walsh, Gerry Coleman, Bertrand Cornanguer, Istvan Forgacs, Kari Kakkonen, and Jan Sabak, published July 2017. Kari Kakkonen wrote this selection. The authors are all members of the ISTQB Working Group that wrote the ISTQB Agile Tester Foundation syllabus.]*

The traditional way of developing code is to write the code first, and then test it. Some of the major challenges of this approach are that testing is generally conducted late in the process and it is difficult to achieve adequate test coverage. Test-first practices can help solve these challenges. In this environment, tests are designed first, in a collaboration between business stakeholders, testers, and developers. Their knowledge of what will be tested helps developers write code that fulfils the tests. A test-first approach allows the team to focus on and clarify the expressed needs through a discussion of how to test the resulting code. Developers can use these tests to guide their development. Developers, testers, and business stakeholders can use these tests to verify the code once it is developed.[1]

A number of test-first practices have been created for Agile projects, as mentioned in section 2.1 of this book. They tend to be called X Driven Development, where X stands for the driving force for the development. In Test Driven Development (TDD), the driving force is testing. In Acceptance Test-Driven Development (ATDD), it is the acceptance tests that will verify the implemented user story. In Behaviour-Driven Development (BDD), it is the behaviour of the software that the user will experience. Common to all these approaches is that the tests are written before the code is developed, i.e., they are test-first approaches. The approaches are usually better known by their acronyms. This subsection describes these test-first approaches and information on how to apply them is contained in section 3.3.

Test-Driven Development was the first of these approaches to appear. It was introduced as one of the practices within Extreme Programming (XP) back in 1990s[2] . It has been practiced for two decades and has been adopted by many software developers, in Agile and traditional projects. However, it is also a good example of an Agile practice that is not used in all projects. One limitation

1 This concept is not unique or new to Agile development. Boris Beizer, in his book *Software Testing Techniques*, talks about the value of a test-first approach to software development.

2 You can find the initial description in Kent Beck's *Test-driven Development: By Example.*

with TDD is that if the developer misunderstands what the software is to do, the unit tests will also include the same misunderstandings, giving passing results even though the software is not working properly. There is some controversy over whether TDD delivers the benefits it promises. Some, such as Jim Coplien, even suggest that unit testing is mostly waste[3].

TDD is mostly for unit testing by developers. Agile teams soon came up with the question: What if we could have a way to get the benefits of test-first development for acceptance tests and higher level testing in general? And thus Acceptance Test-Driven Development was born. (There are also other names for similar higher-level test-first methods; for example, Specification by Example (SBE) from Gojko Adzic.)[4]. Later, Dan North wanted to emphasize the behaviours from a business perspective, leading him to give his technique the name Behaviour-Driven Development[5]. ATDD and BDD are in practice very similar concepts.

Let's look at these three test-first techniques, TDD, ATDD, and BDD, more closely in the following subsections.

Test-Driven Development

Test-Driven Development is a method whereby unit tests are created, in small incremental steps, and, in the same small incremental steps, the code is created to meet those tests. Metaphorically, think of how a bush's shape can be made orderly and in conformance with desired behaviour by the use of a lattice-like frame (i.e. a trellis). These unit tests allow software developers to verify whether their code behaves according to their design, both as they develop the unit and after making any changes to the unit. This provides a level of confidence that leads many developers to stay with TDD once they have become accustomed to the process. However, other developers find the process too tedious or cumbersome, and instead create and run their unit tests after coding.

TDD involves first writing a test of the expected low-level functionality, and only then writing the code that will exercise that test. When the test passes, it is time to move to the next piece of low-level functionality. As you grow the number of tests and code base incrementally, you also need to refactor the code frequently. If you don't refactor, you may end up with 'spaghetti' code that is neither maintainable nor understandable. Some people would rather do lots of architectural design first, and you can do that, as well, to some extent, to avoid too much refactoring. However, the strength of TDD is that you only develop the minimum code that is required to pass the existing unit tests, and only then move on to the next piece of test and code. This way you avoid unnecessary

---

3 You can find Coplien's article, "Why Most Unit Testing is Waste," at http://rbcs-us.com/resources/articles/why-most-unit-testing-is-waste/

4 One widely-read source on the topic is Adzic's *Bridging the communication gap: Specification by Example and Agile Acceptance Testing.*

5 A good discussion on this topic is Chelimsky's *The RSpec Book: Behavior Driven Development with Rspec, Cucumber, and Friends.*

code. Your code will be lean, fast and maintainable. Debugging will be very easy as you have only a small code increment to look at[6].

The tests need to be automated, usually in a test framework such as Junit, Cppunit, or any of the other xUnit family of frameworks (where *x* stands for any programming language). We'll discuss these unit testing frameworks further in section 3.4. Without automation, repeating the tests all the time is not viable, which makes refactoring and other code changes more likely to result in undetected regression. These automated tests also should be part of a continuous integration framework, so you'll know that your whole system works as you include more new code. The process is highly iterative:

1. You write a new test and expect it to fail.
2. You write just enough code to pass that test and keep running the test and changing or adding code until the test passes.
3. You refactor code for maintainability and run the new test and earlier test, again repeating the actions if the tests don't pass.

Graphically, this process is shown in Figure 3.1.

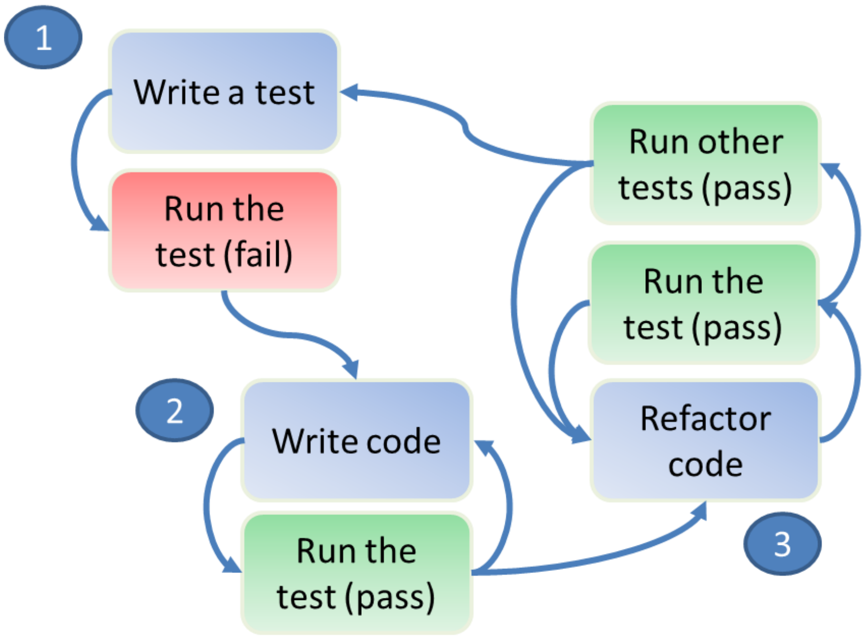

Figure 3.1: Process of creating tests and code with TDD

The tests you create in this way are unit tests for the code. They can even work as a replacement of some of the technical documentation or technical design. Typically, you would try to achieve at least

[6] Lehtonen, T. et al (2014) *Sulautettujen järjestelmien ketterä käsikirja (Agile handbook for embedded systems).* Available from http://trc.utu.fi/embedded/kasikirja [13 July, 2016].

100% decision (branch) coverage with unit tests. You can also apply TDD to integration tests and systems tests, although this is seldom done in practice.

Example

Susan is a developer of an e-commerce store software. Currently she wants to create a postal address input field and avoid mistakes with the input field. She writes a sequence of unit tests that exercise the input field utilizing equivalence partitioning and boundary value analysis technique. As she does this, she also adds equivalent input field validation codes to her code. She works in TDD-fashion.

Therefore, she only scripts her first unit test to begin with.

"Try a regular value for the postal address field". Test should fail, as the field is not coded yet.

She runs the test and it fails, giving a different expected value than was scripted.

Test fails, not surprisingly.

Then she creates the postal address field and adds the equivalent validation check to the code.

"Regular value is allowed"

She then runs the same test again.

Now the test passes, as the field is there and the validation check also works.

Then she scripts the next test, and adds some more validation code and so on. In the end, there is a well-validated postal address input field in the user interface.

Acceptance Test-Driven Development

When doing Acceptance Test Driven development, as part of an Agile practice, tests are created in an iterative way, starting before, and continuing during the implementation of a user story (see more about user stories in section 1.2).

User stories need to include acceptance criteria and those in turn can be turned into (drafts of) acceptance tests. All this happens through business representatives, developers and testers working together in a specification workshop[7]. The workshop is not only about creating acceptance tests. The real goal is to collaboratively understand what the software should and should not do. ATDD strives to encourage and improve the communication between the business, the developers, and the testers. Acceptance tests are a by-product, or, if you will, an exposition of this discussion. The (drafts of) acceptance tests should next be detailed enough that code can be exercised against them. (In other words, they tend to be more toward the concrete test end of the logical test-concrete test spectrum.) They could be (and usually are) also automated for the ease of use. The

[7] Adzic, G. *Bridging the communication gap: Specification by Example and Agile Acceptance Testing.*

acceptance tests are continuously refined with additional details about how to interpret the user story and how to show whether it satisfies the expressed user needs or not. ATDD tests are higher-level tests than unit tests, but still quite small tests, each exercising one or more acceptance criteria of a user story. They belong to quadrant two of the Agile Testing Quadrants, which we'll discuss later in this section. The ATDD process is described in detail in section 3.3.

Acceptance test-driven development makes it possible to

- test the code quickly at the user story level, system level, or acceptance level
- verify that acceptance criteria are met
- find higher level defects early in user story development
- automate regression test sets, or at least create automatable elements for such sets
- improve collaboration between all stakeholders

ATDD as a methodology doesn't require test automation, but it is usually coupled with a powerful test automation framework that can take input from specification workshops (with the help of the test automation expert). Some acceptance tests need to access databases, some APIs, and some the user interface, so usually a framework is also needed from a technical point of view. Frameworks can then use multiple test execution tools.

Example

The project team wants to understand how the registration to a web site will work. The user story is just "As a new user I want register on the web site so I can use it." They set up a workshop and quickly find several questions they want answers for:

- In which order should the process go?
- What information should be received from the user and in which format?
- How should the user be informed about the progress and status of registration?

They take the approach of talking about example answers to the questions (simplified):

- Process example: A user called John Davis could register first his user ID JohnD, then give a password, then give more required information, and then voluntary information.
- Information example: We want to capture, at least, first name (John), surname (Davis), user ID (JohnD), password (#¤2"dd), email (JohnD@email.com), and country (UK). Voluntary information would be the rest of the address and phone number. The format should be plain text in text fields, with copy-paste and auto-fill allowed. Length limits should be enforced on fields; for example, a maximum of 50 letters for surname.
- Progress example: John should expect to see the registration steps on a progress bar on the top window at all times.

Talking through examples with real data, the team end up adding acceptance criteria to user stories, formatting them as such:

- User ID and password must be created first

And also a similar set of acceptance tests:

- If user John Davis tries to register user ID John and, having checked the user ID list it is already taken, he must try again with JohnD, which is then created. Then he is asked for his password.

All this happens during the same meeting, which is the specification workshop.

BEHAVIOUR-DRIVEN DEVELOPMENT

Behaviour-Driven Development (BDD) starts from a point of view that behaviours of software are easier for stakeholders to understand when taking part in test creation than the specific tests themselves[8] . The idea is, together with all team members, business representatives, and possibly even customers, to define the behaviours of the software. This definition can happen in a specification workshop, very much like ATDD. Tests are then based on the expected behaviours, and the developer runs these tests all the time as she develops the code.

BDD promotes immediate test automation even more than ATDD. One of the leading ideas is to use very clear English (or another spoken language), so that both business users participating in a workshop and a test execution tool can understand the behaviour and the test that verifies that behaviour. Some test automation frameworks are adapted (such as Robot Framework) and some even created (such as Cucumber and JBehave) to work specifically with BDD methodology.

Participants of the specification workshop are instructed to think how a system will behave. One format is the *given/when/then* syntax, part of the Gherkin format of the Cucumber tool family:

- *Given* some initial context,
- *When* an event occurs,
- *Then* ensure some outcomes.

The format should also include data that can be used in a test.

Example

A successful system login scenario should work as follows when described in *Given/When/Then* format:

Given that the system is in the main page or login page
When user IDs (Steve, Stan, Debbie) *And* passwords (45KE3oo%&, DF44&aa, 23##a&SK) are inputted to the user ID *And* password fields
Then the system will give the message "Login Accepted" *And* move to landing page.

Behaviour-Driven Development creates business level or acceptance test level tests that can be used by a developer as the team's acceptance tests or even as part of unit tests. Such tests would be in addition to other unit tests the developer creates as she codes the program, aiming for good coverage (for example, 100% decision coverage). The test automation framework will give the

[8] Chelimsky, D. et al (2010) *The RSpec Book: Behavior Driven Development with Rspec, Cucumber, and Friends.*

drafts of tests for developers and testers directly. The drafts can then be refined to executable test scripts.

In both BDD and ATDD, the automated BDD or ATDD tests are usually included in the continuous integration framework. This is done to:

- expand the smoke test (or regression test) set to include a business perspective as well as the technical low-level unit test perspective (for example, provided by TDD)
- to verify the outcomes and behaviours don't unexpectedly change after a change to the code
- find defects as close as possible to the moment of introduction, and to fix it upon discovery

Possible downsides of running the tests in continuous integration include:

- building a new executable might take too long with all those tests included (but you can of course only choose part of the unit tests, ATDD, or BDD tests to run each time)
- maintaining tests might become time-consuming

[We hope you've found this excerpt of Agile Testing Foundations *interesting. For more, pick up a copy of the book, which is available now from RBCS and other online bookstores.]*