How the Testers are Involved in TDD, BDD & ATDD Techniques

Overview of TDD, BDD and ATDD techniques:

TDD, BDD & ATDD are the terms which have revolutionized the tester's world in Agile and have gained momentum too. Change in the mindset of testers also requires learning new skills and more importantly, changing the attitude, and the way of working.

Unlike working in isolation, testers need to collaborate and work together with the programmers which means

- Sharing the test cases
- Participating in writing the acceptance criteria of the stories
- Providing continuous feedback to the stakeholders
- Collaborating to resolve the defects on time.
- Provide suggestions and input for improving the quality of the deliverables
- Automation

Before I jump into the discussion of a tester's involvement in these practices, let's first try to understand the concepts behind these terms:

Test Driven Development

Consider the traditional approach of software development where the code is written first and then tested. Test-driven development or TDD is an approach which is the exact REVERSE of traditional development. In this approach, testing is done first, and then, the code is written.

Confused?!!

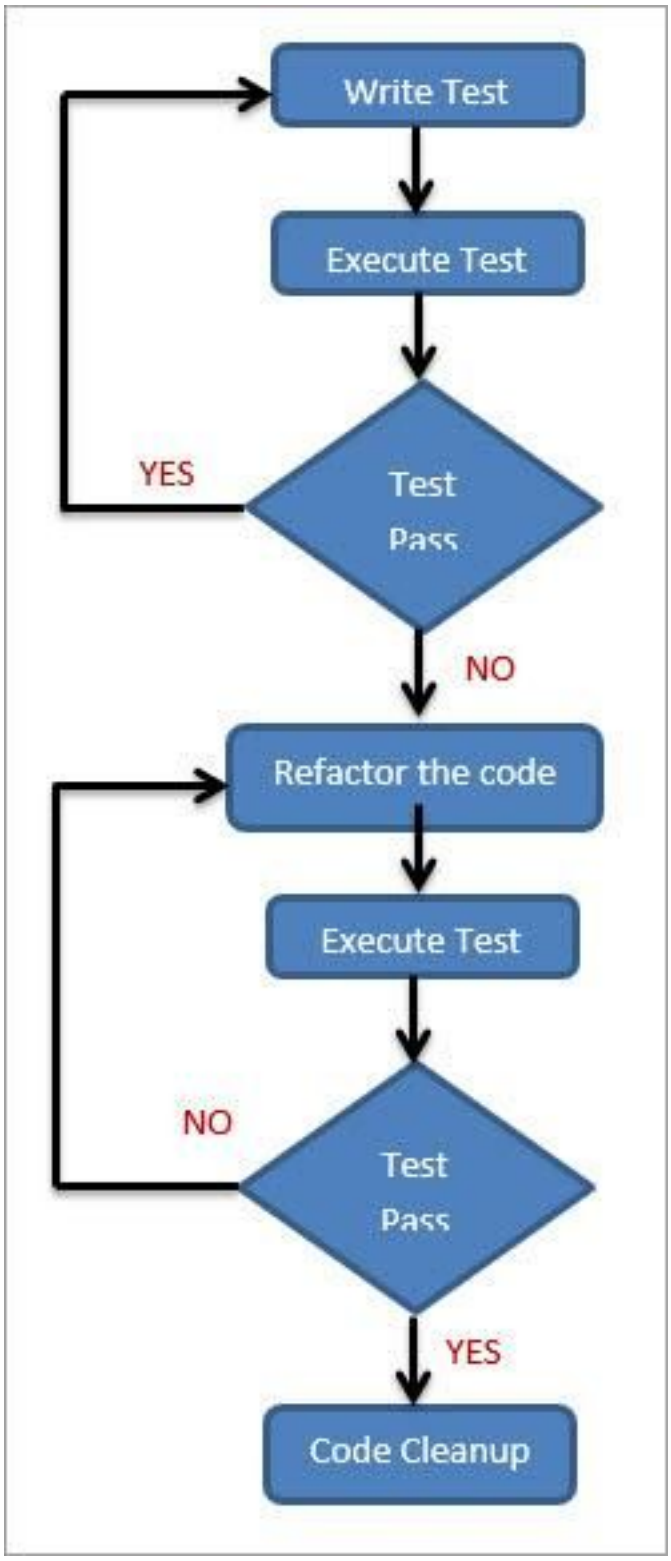
How can we test a piece of software which is yet to be developed?

Yes!! That's test-driven development or TDD.

TDD works in small increments where:

- The test is written first
- The test is executed – which will fail (for obvious reasons)
- The code is written (or refactored) just to make the test case pass
- The test is executed again
- If the test passes, move on to the next test ELSE re-write / modify the code to make the test pass

Let me try to explain it through a flowchart:


Write Test
Execute Test
Test Pass
YES
NO
Refactor the code
Execute Test
Test Pass
NO
YES
Code Cleanup

Now, we have to understand the fact that TDD does not talk about the testing that testers do. Rather this approach actually talks about the testing which the programmers do.

Yes!! You guessed it right!! It's the unit testing.

The test which is written first is not the test that the testers write, but it is the unit test which the programmers write. So, I would rephrase the steps as:

- The UNIT test is written first
- The UNIT test is executed – which will fail (for obvious reasons)
- The code is written (or refactored) just to make the test pass
- The UNIT test is executed again
- If the test passes, move on to next test ELSE re-write / modify the code to make the test pass

Now, the question that arises here is – if TDD is a programmer's job, what is the tester's role in this approach?

Well, having said that TDD is a programmer's job, it does not mean that the testers are not involved in it. Testers can collaborate by sharing the test scenarios consisting of:

- Boundary value cases
- Equivalence class test cases
- Critical business cases
- Cases of the error-prone functionalities
- Securing level cases

What I mean to say is – testers should participate in defining the unit test scenarios and collaborate with the programmers to implement the same. Testers should provide their feedback on the test results.

If we want to implement TDD, testers need to upgrade their skill sets. They need to be more technical and focus on improving their analytical and logical skills.

Testers should also put in an effort to understand the technical jargon that the programmers use, and if possible, must have a bird's eye view of the code. In a similar fashion, the programmers have to step into the tester's shoes and try to come up with more sophisticated scenarios that will make the unit testing more robust and solid.

Both programmers and testers have to step into each other shoes, unlike the old traditional approach where the programmers did not give much weight to the unit tests because they relied on the testers for finding bugs and defects, and the testers did not want to indulge themselves into learning the technical stuff because they think their job ends after finding the defects.

Behavior Driven Development

Behavior Driven Development or BDD is an extension to Test Driven Development.

BDD, as the name suggests, illustrates the methods of developing a feature based on its behavior. The behavior is basically explained in terms of examples in a very simple language which can be understood by everyone in the team who is responsible for the development.

Some of the key features of BDD are as follows:

- The language used to define the behavior is very simple and in a single format in which it can be understood by everyone – both technical and non-technical people involved in the implementation
- Gives a platform that enables the three amigos (programmer, tester, and PO/ BA) to collaborate and understand the requirement. Determines a common template to document it

- This technique/approach discusses the final behavior of the system or how the system should behave and it does NOT talk about how the system should be designed or implemented
- Emphasizes on both the aspects of quality:
    - Meet the requirement
    - Fit for use

Why BDD?

Well, we know that fixing a defect/bug at the later stage of any development cycle is quite costly. Fixing of the production defects not only impacts the code but also the design and its requirements. When we do the RCA (Root Cause Analysis) of any defect, most of the time, we conclude that the defect actually boils down to miss-understood requirements.

This basically happens because everybody has different aptitudes to understand the requirements. Hence, technical and non-technical people may interpret the same requirement differently, which may lead to a faulty delivery. Therefore, it is critical that everybody in the development team understand the SAME requirement and interpret it in the SAME way.

We need to make sure that the entire development efforts are directed and focused towards meeting the requirements. In order to avoid any kind of a “requirement – miss” defect, the entire development team has to align them to understand the requirement which is fit for use.

BDD approach allows the development team to do so by:-

- Defining a standard approach to define the requirement in simple English
- Provision of defining examples that explains the requirements

- Provide an interface/platform which enables the technical (programmers/testers) and non-technical (PO/ BA/ Customer) to collaborate and come together and be on the same page to understand and implement the requirements

How to Implement BDD?

There are many tools available in the market for implementing BDD. I am naming a few here:

- Cucumber
- Fitnesse
- Concordion
- JBehave
- Spec Flow

Example:

Let's try to understand BDD with an example. For my case, I am using Gherkin (Cucumber):

Consider a simple case where we want to allow only authenticated users to enter into XYZ site.

The Gherkin file (feature file) would be as follows:

Feature: Test registration portal

Scenario Outline: Valid user logged in

Given: Customer opens the registration portal

When: user enters the username as "<user>" & password as "<password>"

Then: the customer should be able to view the form.

Examples:

|user |password|

|user1 |pwd1|

|user2 |pwd2|

We can see how a particular requirement is documented using simple English. The three amigos can work together to design the feature files and specific test data can be documented in the example section. BDD provides a medium to bring programmers, testers, and business to one table and establish a common understanding of the features to be implemented.

BDD approach saves effort & costs and checks if there are any defects post the production deployment as the collaboration of the customers and developers was throughout the development cycle of the feature.

Development teams can utilize these feature files and convert them into executable programs to test a particular feature.

How?

Well, you need to learn Cucumber / Fitnesse for that!!

Acceptance Test Driven Development

Acceptance Test Driven Development or ATDD is a technique where the entire team collaborates to define the acceptance criteria of an epic/story before the implementation

actually begins. These acceptance tests are supported by proper examples and other necessary information.

Most of the time, BDD and ATDD are used interchangeably. The ATDD approach can also be implemented using the Given-When-Then format, just like how we write features in the BDD approach.

Let's quickly try to summarize the differences between the 3 approaches:

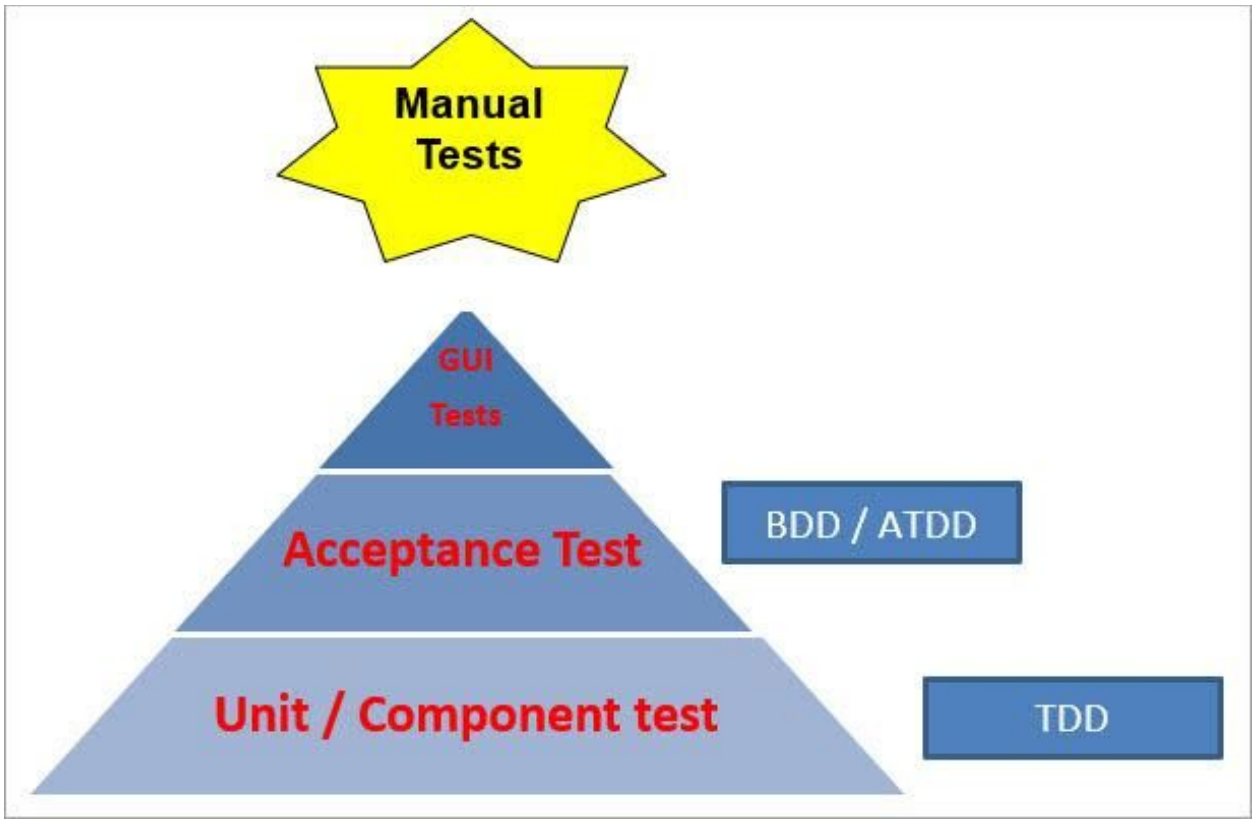
- TDD is more technical and is written in the same language in which the feature is implemented. If the feature is implemented in Java, we write JUnit test cases. Whereas BDD & ATDD is written in simple English language
- The TDD approach focuses on the implementation of a feature. Whereas BDD focuses on the behavior of the feature, and ATDD focuses on capturing the requirements
- To implement TDD we need to have technical knowledge. Whereas BDD & ATDD do not require any technical knowledge. The beauty of BDD / ATDD lies in this fact that both technical, as well as non-technical people, can participate in developing the feature
- Since TDD is evolved, it gives scope for good design and focuses on the "Meeting Requirement" aspect of quality; whereas BDD / ATDD focus on the 2nd aspect of quality which is "Fit for use"

All these techniques basically talk about the "test-First" approach, unlike the "test-last" approach used in traditional development methodologies.

As the tests are written first, testers do play a very important role. Not only do testers need to have a vast domain and business knowledge, but they also need to possess good technical skills so that they can add value in brainstorming during the 3 amigos discussions.

With the concepts like CI (Continuous Integration) and CD (Continuous Delivery), testers need to collaborate well with the programmers and contribute equally to the development & operations area.

Let me summarize this discussion with the famous Test Pyramid in Agile:



The lowest layer of this pyramid is made up of the unit test layer. This layer is the foundation layer; therefore it's imperial that this layer is rock solid. Most of the tests should be pushed into this layer. This lowest layer focuses only on TDD.

In the Agile world, an emphasis is laid on making this layer of the pyramid more strong and robust and it is emphasized that most of the testing is achieved at this layer.

Tools used in this layer of a pyramid are all the Xunit tools.

The middle layer of the pyramid is the service layer, explaining the service level tests. This layer acts as a bridge between the application user interface and the lower level unit/component. This layer mostly comprises of the APIs that accept requests from the UI and sends back the response. The BDD and TTDD approach is active in this layer of the pyramid.

Tools used in the middle layer of the pyramid are – Fitnesse, Cucumber, and Robot Framework.

The topmost layer of the pyramid is the actual UI, which shows that this layer should contain the least number of tests, or I should say, the testing effort at this particular layer should be minimal. Most of the testing of the feature should have been completed when we reach the top layer of the pyramid.

Tools used in the top layer are – Selenium, QTP, and RFT.

Since we work in small increments in Scrum (called Sprints), manual implementation of all these approaches is not feasible. These approaches require automated intervention. Automation, in this case, is very critical.

In fact, these approaches are actually executed through automation. At the end of every sprint, new features are being added, so it becomes important that the previously implemented feature work as expected; hence, Automation becomes the HERO here.

Conclusion

By the end of the article, you must have learned about How the Testers are Involved in TDD, BDD & ATDD Techniques.