As humans, we're prone to making mistakes. Inherently, we devise techniques and procedures (including machines) which thereby automate the process and lower the risk of errors. Test Driven Development is one such technique used by the Software Developers to continuously test as they code.

Wikipedia defines Test-driven development (TDD) as a software development process that relies on the repetition of a very short development cycle: first the developer writes an (initially failing) automated test case that defines a desired improvement or new function, then produces the minimum amount of code to pass that test and finally refactors the new code to acceptable standards.

It is basically a method of repeating your tests on the source code to ensure that the base functionality is still working as the developer continues to evolve and advance the code to improve the feature. If something goes wrong and the tests fail, the developer will only have a few lines of code to check and identify the problem source and so it automatically becomes a cost saver.

Moreover, since these tests are written into the code itself, they remain and are utilized by being run each time code changes are made. This helps ensure that the basic and critical functionality still works as the codebase continues to expand with new features and does not bring in any surprises at critical moments.

However, this paper is not about TDD; rather it talks about how it is very different from Traditional Testing and how it can become a collaboration front between the DEV and QA teams. We'll evaluate and focus more on these in the upcoming sections.

TDD vs. QA Myths

TDD is primarily a development technique with the objective of ensuring that the basic functionality of a unit in the source code continues to work as you build more complex functions on it. However, often people confuse it as being a replacement of Software Testing. TDD is only part of your overall testing efforts. At best it ensures that the basic functionality was not adversely affected while building extended functionality or new functionality and acts as a self-check and control mechanism for the developers to give them some confidence about their own code. What is not taken into account is that there are several phases and types of testing such as integration testing, scenario based testing, system testing, compatibility testing, ad-hoc or exploratory testing and even release testing which are not related to the TDD we have seen thus far.

Moreover, 'Testing' itself can be done in several ways including static means such as code reviews, code inspections, etc., which is not just about bug finding (which is the primary objective of TDD).

TDD and QA – Extension/Collaboration

The core idea behind TDD is to have the developer "write the test first" and then write the code to pass the test. The technique forces the developer to think of the conditions which would fail the code before even starting to code. This is easier said than done as the age old premise distinguishing developers from testers comes into play. It has always been felt that developers and testers hold two different mindsets and cannot be replaced by each other. Although, I am not a very big advocate of this hypothesis, having several years of experience in the testing domain, I too somewhat feel that after working in one of these domains for a while, you do get attuned to and develop a way of thinking which becomes hard to change after so many years.

To take this idea further, ever wondered about having QA write the test cases for new functionalities just as they normally do; which would be used by the DEV team while developing the code using TDD methodology. Now, this is not something very prevalent across the various SDLC models. Even

in Agile, although DEV and QA teams work very closely together, QA maintains their own test scripts that are not used by DEV before delivering their code for testing. In some teams this practice is changing, with BVTs and smoke tests handed off to the developers to be run before a build is deployed for testing. This certainly helps improve overall quality of the testable build and saves time for everyone in the development life cycle.

However, let's just think about the various benefits this approach could possibly bring in. Developers who do not have a testers mindset will be able to leverage the test cases created by QA (whether requirements or system based tests) and keep them in mind while developing the functionality so as to minimize the churn that happens so often due to issues observed in the functionality developed.

To demonstrate this with the help of a live example, please refer to the case study below that I have put together from one of the recent projects where I was able to persuade the cross-functional team to experiment with this 'Extended' TDD approach.

Case Study

The case study is based on a project which was completely development driven. Although it was based on an Agile model, the QA team always felt that their requests and suggestions were not given enough importance. Moreover, there weren't enough checks in place to ensure that the code coming in to QA was stable.

<u>The Current Process:</u>

1. No running list of requirements (There may be one liners' in defect tracking tool for tracking, but comprehensive details were missing)

2. QA was unable to propose any changes to the requirements based on analysis

<u>Suggestions proposed:</u>

- If proper requirements (User-Stories) were in place, QA can review them and prepare test cases around user stories

- DEV team to follow a TDD approach for building new features using the test cases created by QA team based on User Stories and Requirements

<u>Expected Benefits:</u>

- The slips when fixed by Dev team would have a very minimal chance of failure

- The Dev and QA Teams would be coordinating closely and working as one single unit to accomplish the task of meeting requirements

- This would help in defect prevention as compared to defect detection

- It would also help to ensure that release deadlines are largely met.

The methodology that we ultimately followed for this was to have user stories released to QA for release 2 while development of release 1 was still in process. Before the start of release 2, the test cases would have been created and provided to the dev team. During the course of release 2, dev team would use the test cases created by QA for their TDD and send the features to QA for testing once complete.

The proposed approach was implemented and monitored for a period of 3 months. It was observed that the number of defects being reported by QA team via test case execution declined by a

considerable percentage. The releases also started occurring on schedule and the churn rate for new features also decreased significantly. An additional benefit that was realized due to this approach was the fact that QA got more time to spend on exploratory testing to uncover issues which require time and user mindset to track. This in turn helped improve the stability of application which was the overall objective of using the extended TDD approach.