Should I test before I code?

What is test driven development and why you should you it

Aigars Tumanis, tumanis@kth.se

April 30, 2019

1 Introduction

Test driven development (TDD) has increased in popularity over the last few years. The process of creating unit tests before the actual code that these tests are supposed to validate is originally a pretty old technique (from mid-1990s), which was "rediscovered" in 2003 by Kent Beck[1]. A lot of programmers today try it out, fail, then go back to their usual programming method of shoehorning the tests into the code they already have. Often times this leads to fragile tests that will fail as soon as the code is changed, in addition to introducing confirmation bias. In the worst case scenario, the tests are skipped altogether due to time constraints or the fact that "the code seems to work". In this essay, I aim to show that by sticking to the TDD, you can produce clean and testable code that is robust and easy to read. Additionally, I will present a paper that shows the deficiencies of the method and how it could be improved by combining it with Model Based Development.

2 Test Driven Development

2.1 What it is

Test driven development is an incremental software development process where you write and run a set of tests before you run the actual code. Initially, the tests will fail, but by gradually extending the code the tests will begin to pass[2].

Robert C. Martin, the godfather of TDD, describes the process with three rules:

1. You are not allowed to write any production code unless it is to make a failing tests pass.

2. You are not allowed to write any more of a unit test than is sufficient to fail.

[1] https://hackernoon.com/introduction-to-test-driven-development-tdd-61a13bc92d92
[2] https://www.testingexcellence.com/pros-cons-test-driven-development/

3. You are not allowed to write any more production code than is sufficient to pass the one failing unit test.

Following these steps is meant to lead you to well designed code that is easily reusable and maintainable. But it also sounds hard and time consuming, doesn't it? It sounds like it will slow you down and brake your flow! So why should you use TDD? In short, because it is the simplest way to achieve good quality, testable and readable code while also getting good test coverage[3].

2.2 TDD Phases

The TDD process consists of a cycle based on three phases; **Red**, **Green** and **Refactor**. These phases aim to guide the developer towards the correct way of doing TDD and following the three aforementioned rules.

Red Phase In the red phase, you write tests for a behavior that you want to implement. You are not allowed to write, or even think about, any production code yet. The point here is to think about what your system is supposed to do and write a test for that function. The phase is called red, as running the test will cause the said test to fail, and thus result in the color that is associated with the test failure. While this may be disheartening, it has a lot of implication for how the code you produce will be.

As an example, take a look at the following test. It was created created in Mocha, a JavaScript testing framework, for a social number validator.

```
it('should return day and month error if day and month are wrong',
    function() {
        assert.deepEqual(app.validateDOB("41","53","89"),
        ['day', 'month']);
    }
);
```

All you need to know about the framework is that `it{...}` marks the test and `assert` validates the result. Nevertheless, after looking at this test you know that the method will be called `validateDOB()`. It will take three parameters and return a list of errors. The code for this test should be written accordingly.

The important thing to remember is that the red phase is where you design how your code will be used by clients. You make important decisions about how the code will be used and you should base it on what you need at that moment, not what you think you will need in the future.

Green Phase During the green phase, you (finally) write the production code. Important thing here it not to over engineer. You don't need to implement all of the algorithms - just enough for the test to pass. In short terms – you need to create a solution that

[3] http://butunclebob.com/ArticleS.UncleBob.TheThreeRulesOfTdd

makes the test pass. You must not worry about performance or best practices, as long as you make the red turn into green. Additionally, TDD method suggests that you keep a to-do list, that should be used for writing down steps to complete a feature, as well as any doubts or problems you discover during the process. Ideally, when nearing the end of the project, the to-do list should be empty. Once the test passes, you can continue to the next step.

Refactor Phase The last step is the refactor phase, where you are allowed to change code to make it better. How you make it better is up to you, but one rule is that you have to remove code duplication. Often times this alone is enough to make your code cleaner and more readable, but you are not limited to that, as long as your tests keep passing. Once the last step is completed and you are satisfied with your code, you can restart the loop by creating the next test.

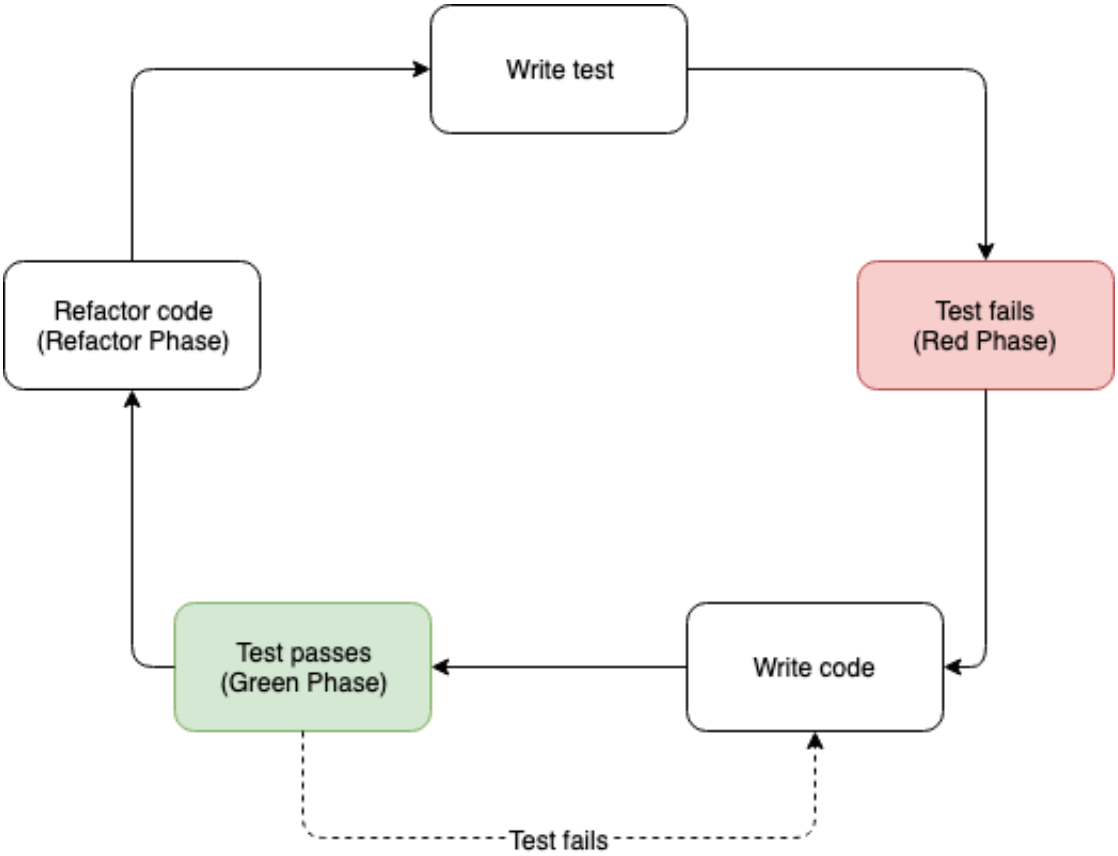

Figure 1: TDD cycle with Red, Green and Refactor phases. The red phase includes writing tests and failing them, the green phase includes writing code until the test passes and the refactor phase leads to cleaner code with no duplicated code.

2.3 Does it work?

There are a lot of misconceptions about test driven development. One of the main concerns is that it requires more time than "regular" programming. In his article about test driven development[4], Andrea Koutifaris suggests that what actually takes a long time is to master the process. Once that is done, the process actually makes you code as simple as possible and may even save time. This sentiment is also resonated by a lot of proponents of the method. By having code that you know works at any moment in time, you save a lot of time on not needing to debug and fix broken functions and tests. Every time you add a new feature, you only need to run the tests to make sure that it doesn't break the rest of the application. To exemplify this, the TDD process has been tested on companies like IBM, Microsoft and Springer. It was found that the *"pre-release defect density of products decreased by between 40% and 90%"*[5].

Another major concern is that you cannot actually write tests until you know how your code looks, but as previously mentioned, studies have confirmed that this is not true. The fact that while developing tests, you really have to think about what you are going to do, encourages you to plan out you application as opposed to the usual way of just typing out code and hoping for the best.

Another point is that TDD is not just about "writing a lot of tests" - if you just start by creating several tests and then try to implement the code for them, you are doing it wrong. The TDD process is set up the way it is for a reason - write one test, write code for that test. Overengineering will only lead to trouble. That said, TDD is not an exact science or a set of strict rules, and more of a suggestions, so you can adapt it the way you see fit, as long as you adhere to the phases mentioned above.

3 Model Based Test Driven Development

While TDD provides a lot of positives, there are also some drawbacks to the method. There are shortages and deficiencies that discourage developers from using it, especially development of large-scale software intensive systems. In a paper by Alireza Sadeghi and Seyed Hosseinabadi[6] the writers propose the idea that since TDD is a low-level approach that focuses primarily on pieces of code and test cases, and that manually writing tests for each piece of code is time consuming, the approach can hinder the flow of creating large-scale applications. In order for it to be applicable to those kinds of systems, they propose to merge Model Based Testing with TDD.

3.1 Model Based Testing

Model Based Testing (MBT) is a software testing technique that describes how a system behaves in response to an action. This is checked against a description of a system's

[4]https://medium.freecodecamp.org/test-driven-development-what-it-is-and-what-it-is-not-41fa6bca02a2

[5]https://medium.com/javascript-scene/5-common-misconceptions-about-tdd-unit-tests-863d5beb3ce9

[6]2012, A.Sadeghi, S. Hossenabadi, MBTDD: Model Based Test Driven Development

behavior, called a model. A model can be described in terms of input sequences, conditions, output or flow of data[7]. An MBT process usually consists of modelling the behavior of a system, specifying test criteria, verifying the behavioral model, generating and running test cases and analyzing test results.

3.2 Model Based Test Driven Development

While the main focus of TDD is on the low-level software programming and code that ignores more abstract structures such as architecture, the main strengths of MDD is exactly the opposite - the abstraction, platform independence and the ability of automation. In theory the merging of these would therefore strengthen both methods. The paper suggests that this can be accomplished by extending the TDD-model with an additional Modelling step, which is added first. During this a reference model is created. This abstract model is then converted into a concrete test case. Finally software code is written to fulfill this requirement. This process is depicted in figure 2.

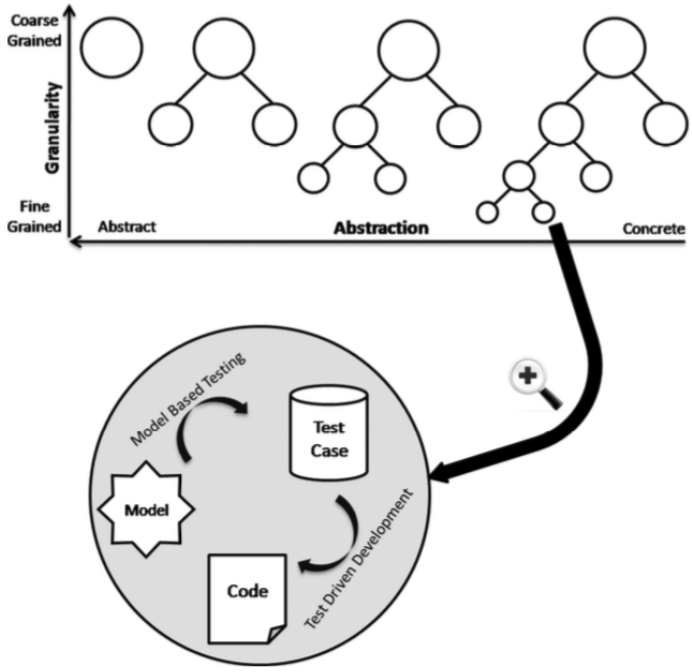

Figure 2: MBTDD Structure as depicted in the paper. The modelling phase is a top-down traversal of the tree, while test and code creation is traversed bottom-up.

The model is presented in the form of a tree and the traversal direction is dependent on where in the process you are. During the Red phase, the tree is traversed top-down, starting with the coarse grained tests and ending in the fine-grained ones. When writing

[7] https://www.guru99.com/model-based-testing-tutorial.html

code for the tests, the tree is traversed bottom-up. This means that the fine-grained tests must be passed before the corresponding course-grained test cases are written.

3.3 MBTDD Structure

The first steps towards constructing the framework is meta-model definition. The meta-model, or a model of a model, helps in building the components of frameworks easily, rapidly and correctly. MBTDD meta-model consists of the test setup, pre- and post-condition and test inputs. The test setup entails the initialization and preparation of the test environment, but also the test teardown, that happens after the tests have been completed. Preconditions are a set of conditions that must be in place for a test to run. Similarly, postconditions are condition that should be met after the execution of tests. Test inputs are all input data and parameters that are used during the tests.

The meta-model is the part that separates TDD from MBTDD. The meta-model leads to a creation of high-level abstractions that in turn consist of concrete models and components. To keep track of this, every requirement is assigned an identifier that cascades to from coarse- to fine-grained elements, including test cases and the code that satisfies them.

3.4 MBDTT in practice

In order to test the model, Sadeghi and Hossenbadi used a case study of a company that implemented a Human Resource Management System, which resulted in a web application that exploits a 3-tier architecture. The first layer is the Presentation layer, which handles the UI. The middle layer is the Business layer, that handles the logic of the application, while the third layer is the Persistence layer, that handles the management of the persistent data, such as storing and retrieving data from a database.

Modeling During the modeling phase, the desired behavior of the system was drafted with different failures and success scenarios. An example of these were users providing invalid information, creating new personnel records and retrieving data about a specific employee. These were drafted as platform independent and different models were created for different layers.

In order to transition to writing test cases, the platform was specified and the models were translated for that platform.

3.4.1 Evaluation

In order to test the performance of the model, Sadeghi and Hossenbadi chose the following criteria to test, based on the previous research on the topic: *quality of the product* and *the efficiency of the production process.*

Quality The quality of product was defined to be "conformance to explicitly state functional and performance requirements, explicitly documented standards and implicit

characteristics that are expected of all professionally developed software". In order to compensate for the variable amount of code in different areas, the number of defects per kilo lines of code (KLOC) was utilized.

Efficiency The efficiency of the production process was measured by the total effort in man hours.

3.4.2 Case Study

The case study for the paper was based on data from a company that created a Human Resource Management system. Four modules of the system were created by separate teams that consisted for four subteams: requirements engineering, design and development, testing team and deployment teams.

The subsystem was split into five parts:

- Basic information subsystem
- Employee subsystem
- Organization subsystem
- Recruitment subsystem
- Salary calculation subsystem

The MBTDD was used during the development of the **employee subsystem**.

The teams were comprised of the same members that usually work together, but for some of them a new method was introduced. The requirement engineering team gathered the requirements in the form of expected behavior. The test team translated these into the test model, which the developer team used to generate executable tests and finally code to satisfy the test cases. While this was done, the other teams worked in other subsystems in the usual manner.

The study was conducted for one and a half months. A month after the deployment of subsystems the data was gathered and analyzed.

3.4.3 Results

After analyzing the data, the following results were presented in the paper.

Quality of product The paper concludes that the MBTDD process resulted in a significant advantage in the quality of code (fig 3). The product had significantly smaller number of errors compared to the other modules. This number becomes even better when normalized for the amount of code written.

Index	HR module name				
	Basic information	Employees	Organization	Recruitment	Salary calculation
Number of errors and defects	10	6	16	20	14
KLOC	2	6.5	4	5	6.5
Validity	0.20	1.08	0.25	0.25	0.46

Figure 3: Quality of product measurements in total number of errors, KLOC and validity index

Efficiency of process The efficiency of process was calculated as a correlation between the sum of the development and maintenance time and the Function Point indicator, which measures the size of an application based on the functional view of the system. The size is determined by counting the number of inputs, outputs, queries, internal and external files and adjusting that total for the functional complexity of the system[8].

Index	HR module name				
	Basic information	Employees	Organization	Recruitment	Salary calculation
Total Effort (man-hour)	300	665	430	505	790
Function Point	80	145	150	180	180
Efficiency	0.27	0.22	0.35	0.36	0.23

Figure 4: Efficiency of process in total errors, function point and efficiency index

Figure 4 shows that the efficiency of production process of MBTDD decreased compared to the other subsystems. The authors mention that this could be caused by the extra work of writing the test case and assuring the quality. This extra cost will be compensated during the maintenance phase due to the lower amount of defects.

4 Critique

While the MBTDD may look like a good method of increasing the appeal of TDD for larger scale projects, the reduces efficiency may still be a factor for companies being averse to implement it. In order to find a definite conclusion, the research should have been conducted for a longer period of time with teams getting used to the method. This is why we have to take a look at how the TDD method was improved. The main problem

[8]https://www.gartner.com/it-glossary/fp-function-point

with TDD is its lack of focus on the "bigger picture" and its lack of the design phase. By just focusing on passing the tests, the team may miss key design decisions that may set back a project. It is precisely this that the MBTDD is addressing. By providing meta-models, the process describes the *desired behavior* of the developed software. This change has also been expressed in the Behavior Driven Development (BDD).

BDD is an approach that has evolved from TDD and differs from it by being written in a shared, domain unspecific language that promotes communication and interaction between team members[9]. As with test driven development, the test are written ahead of the production code, but the focus is shifted. The key difference between them is that the tests that are written are focused on user-interaction and the desired behavior of the system. This elevates the focus of the tests and promotes thorough planning before the production can begin. An additional advantage of BDD is that the shared language can promote better cooperation between technical and non-technical stakeholders and also easily be incorporated into Agile development process through use of stories. By using behavior driven development, you can have your cake and eat it too.

5 Conclusion

Test Driven Development is an approach that promotes quality of the software product. The empirical results have shown its positive effect on the quality. Due to its deficiencies, however it has been deemed to be unsuitable for enterprise-sized projects. Making the step from the normal development towards test driven can be tough and many developers are not willing to put the time and effort into this. However, by doing this the code quality can be raised significantly, thus reducing the costs of maintenance.

Furthermore, by combining a high level focused development method, like Model Based Development, with Test Driven Development the deficiencies of Test Driven Development can be limited, making it a viable method for larger enterprise-sized projects. The key fact to keep in mind is that none of these methodologies are the answer to all of your problems. By choosing the best parts of different methods and eliminating others, you can produce a method that is right for the needs of your project.

[9] https://blog.testlodge.com/what-is-bdd/