Advanced Software Engineering Assignment 2

N0786072

Stephen Anderson

Contents

**No table of contents entries found.**

# Task 1

Test driven development (TDD), in a nutshell, is where tests are designed and created before writing code designed to perform functions. It is generally done as an iterative cycle with short stints of programming alternated with running tests and designing new ones where appropriate. (D. Fucci, 2017) The first stint of testing starts with a developer writing a test – usually a unit test – that addresses the desired functionality. This is followed up with writing code that passes the test with time to refactor the function if required. After a test has been passed, and the code cleaned up, another test is written and the cycle continues.

Unit tests are the probably most common – where tests are designed to test a specific function, normally with knowledge of the function itself using example based matching. There are other types of tests such as property based tests where instead of the tester giving an explicit result that the function should be providing, logic is used in conjunction with automatically generating a large input set.

Doing this can help give a better understanding of the function itself, and where issues could occur (Steinhauser, 2018). They also allow for the tests to potentially show different problems every time they are run as the input data is freshly generated every time.

Both unit tests and property-based tests are frequently used in a white box situation where the function or algorithm they are testing is known and the tests are designed to specifically target aspects of the code. Even in TDD where the tests are created before the code is written, generally an understanding of the eventual function will be known and so edge cases and boundary cases can be planned. Edge cases are generally to do with extreme possible values supported by the various data types involved in a function. Boundary cases are where overlaps in logic can occur, for example ‘if statements’ that rely on being above/below a threshold.

The paper written by (A. Cauevic, 2012) presents findings about the quality of tests and test cases in TDD compared to more traditional test last development (TLD) – where code is written to meet requirements and then it is tested after the code is finished to determine if there are any issues with the code.

Overall the findings from the study show that there is not a huge difference between the number of tests between TDD and TLD, and the quality of those tests. It did however show that the developers using TDD had more positive test cases than negative ones, at a slightly higher ratio than TLD, which is indicative of what is referred to as a “positive test bias.” Positive tests are tests that are designed around expected functionality and normal operation, whereas negative tests are specifically designed to see how the code reacts to abnormal situations and unexpected scenarios. The impact that this bias has on TDD could be more detrimental than against test last development due to not necessarily building in functionality to deal with unexpected scenarios.

Personally, I can say that I have noticed the positive test bias even just from this piece of coursework because I find it significantly harder to design tests that are focussed around scenarios that will break or create unexpected outcomes of my code. On the other hand, it can be easier to create tests that specifically address requirements and functionality for the final product.

The work presented by (Mäkinen S., 2014) looks at analysing the effectiveness of TDD and its impact as a whole on the product. What their study shows is consistent with the findings of the paper previously mentioned. There was not a huge disparity between TDD and TLD in terms of the amount of time taken because whilst the coding process of TDD is quicker, the time taken to create tests and write up test reports for each of them balances out this advantage. It was also found that the code base was a similar size, even though the individual function complexity of TDD was slightly better, because what space was saved with code reduction was then used by writing tests to cover significantly more of the code base.

In fact, in my coursework my test file is larger than the file that contains my implementation of a binary search tree.

In the paper by (Dogša, 2011), the effectiveness of TDD is discussed. What was discovered was that TDD generally has a better (lower) failure rate in its tests when compared to TLD. Along with this the project that was being created using TDD generally had an easier to maintain code base and the code quality was on the whole, higher. However, this was offset by the decrease in productivity of the developers because of the increased emphasis and effort put into writing and utilising tests. It also required significant amounts of time put into preparation of the testing unit/framework to ensure that everything is working even before code has been written. The conclusions they present are that TDD can create more modular and abstracted code that is easier to maintain due to only the bare minimum amount of code being written.

What I have found is that setting up a new testing framework and getting used to its workings and methods can be a slow process, especially if it is significantly different from previously used frameworks. It also heavily relies on the tests themselves being correct and functional, I know from experience that just because a test passes it doesn’t mean that the test itself is valid and that the functionality is correct.

(A. Causevic, 2011) looks at some of the factors that are preventing widespread adoption of TDD in the software development industry. The research that they conducted was looking at identifying what the limiting factors actually were, as well as why they had the effects they did. Some of the factors that were identified were: increased development time, how long it takes for a set of requirements to be implemented; lack of knowledge or experience with TDD, including the divide between experienced and novice TDD developers; insufficient design, TDD is usually implemented with a minimal amount of up-front design and frequent refactoring which is different to most existing architectures; lack in testing skills, developers being unable to quickly and efficiently write effective tests. TDD has its advantages but as can be seen, it also has its drawbacks like most methodologies and techniques. To be able to widely adopt TDD and to measure its impact further, these issues need to be planned out and addressed where possible.

What I found when doing this project is that whilst code may pass one test, it can sometimes require significant refactoring to pass others. There were in fact multiple occasions where the functions that I had written, needed to be altered so heavily it was actually easier to delete the code that had been written and start again but this time with the new test in mind as well as the old one.

Some of the further details of the impact on quality of TDD can be seen in the work by (Wilson Bissi, 2016). What the study looks into is previous research and the results from each of them in an attempt to objectively evaluate the quality of code that is created when using it. The research that they performed showed that there were significant enough results to suggest that the external quality of the software was either on-par or marginally better than standard TLD, however this itself can likely be put down to different teams and projects than solely because of the methodology. On the other hand, it was noted that the internal quality was better almost across the board.

# Task 2

First off, I decided upon implementing a binary search tree with a key and value to be used as a dictionary. This then allowed me to have an outline for the main functionality that I was going to aim to have. This included the ability to create an empty data structure, being able to insert a key/item object, having a way of returning a value by using the key, the ability to delete an entry using its key, and also a way to display the different entries in order – using their unique integer key going from smallest to largest.

Now that the main functionality is determined, some tests can be designed. I decided to use the property-based testing framework called quickTest. This is a framework that I had very little prior experience with, being briefly used in some of the lab work. However, it still allowed me to create a rough plan for the tests that I should need:

* Create an empty dictionary
* Create a dictionary and populate it with random key-item pairs
* The created dictionary should store the items in numerical order
* The dictionary can insert many entries as part of a list
* If an item to be inserted has a key that already exists then the existing entry has its value overwritten
* Can only find items in the dictionary when given a key
  + (doesn’t return a random value when given a key for an item that doesn’t exist)
* Can find every single item in a dictionary when given a key for each
* Can print the entries of the tree to the console in numerical order
* Can delete an item when given a key
* Can delete all items in the dictionary in order from the leaves back to the root (the furthest nodes to the closest)
* Can delete every item in the dictionary in a random order

I went through this list in order and defined every test first with a stub definition making sure that when each test was tried it returned false and failed before any other code was written. Then the tests themselves were defined properly.

For example, I needed a test that would attempt to create an empty dictionary. I started with a name for the test, using a standard naming convention in quickCheck, and settled on prop\_bst\_create\_empty which has to return Bool due to the way the framework functions. By default, it was defined as just return false but this is not going to tell me anything whether the functionality is in place or not. The way I designed this test was to create a variable with the constructor named to make the most sense to me, newEmptyBST. After this function call the test will return True which means that the framework has reached the end of the property without crashing and so the test is a success.

Trying to run the test at this stage causes errors during compilation as there is not yet a function definition to create a new empty dictionary. To address this, I needed to define the function which of course requires a stub definition of the data structure as well. Once these are created but without any real functionality the test can be run, but it passes because the function doesn’t do anything which also means that it doesn’t crash. This is a potential drawback that can come from TDD as it requires tests and functionality to be appropriate which is not always the case and the methodology can require a minimum level of functionality and architecture before testing is even possible.

Okay, so the first one didn’t work quite as planned so let’s move to adding an item. This also has a similar problem where even if there is no functionality there is no real way to test that the item has been correctly added to the dictionary without another function. This requires making two functions for just one test which is not ideal but there is not a good way to avoid it other than to make some specific example based unit tests.

The property based test ends up using a randomly generated integer key and a randomly generated string value. This can then be used to add an item, let’s call this function addItem, to an empty dictionary created using newEmptyBST before being searched for using a function like lookupKey which returns just the value of the entry that matches the given key. If the found value matches the generated value provided by the framework then this test is considered passing, currently it fails.

Starting with addItem there are a couple of different scenarios but for this test the only one that can occur is adding an item to an empty tree which means that all the function needs to do is return a tree with a root that has the given key, value and two leaves for each of the two branches.

Fortunately, the lookup function was provided in the lab work and so only needs some minor tweaking to make it fit the correct data structure. It is also at this point that I run through it myself just to make sure that the logic looks sound which it does. Finally, a test can be run and it passes.

Now that some of the basic manipulation functions have been created I can go back to looking at a way to add multiple random pairs. Due to the way that Haskell works, I did not find it possible to easily loop through every pair and insert them individually and I realised that I probably needed to create another function to be able to add an entire list of pairs as they are easier to work with than slicing a list.

To create the test that works for this I first started with looking at creating the empty dictionary and then adding the list of items to it. To then verify if this works I needed to use the lookupKey function again but this time indexing the first item from the head (front) of the generated list and comparing the found value against the second indexed item from the same pair. When trying to run this test, I actually encountered issues when trying to access the head of the list if it was empty which led to automatically returning True if it was so as not to interrupt the flow of the framework. There are ways to stop the generated list being empty but I did not have much luck getting them to work.

Going back to implementing this, I started with a function that takes in a list and a BST object before recursively calling itself with the tail of the given list and adding the head using addItem and an empty BST object. Unfortunately, this is actually incorrect but because of the inadequate test I originally had which only checked that the last item inserted could be found I did not realise that this was the case. What was happening was it was only adding the last item as it was overwriting every other item with a brand new empty tree. This was eventually fixed but demonstrates some of the issues that can occur if an inexperienced developer designs tests that don’t work properly.

Because I did not realise this had happened that testing phase was used for instead designing a test that would let me later create a dictionary directly from a list instead of adding items as a list using an existing dictionary, even if it’s empty.

It took me a long time going through each coding phase and testing phase in turn before I started failing tests that I was confident should have been passing but they were failing for reasons unrelated to the functionality I had been adding. Because some of my logic had been fundamentally flawed I needed to go through and basically delete the functions and start over from scratch. Both with the test and the function.

Another example of test maybe not being complete would be in the ability to delete a node. Deleting the node itself works but I struggled to work out how to move the leftmost/minimum node instead of just copying its values, which occurs whenever the entry to be removed has two children that are branches. This leads to a duplicated node that the test does not pick up but that I am confident will show up if the tree is printed out.

# Task 4

There were definitely some positives that I found from doing TDD. It helped keep me focussed on adding functionality and getting a specific function working instead of doing what I often do which is implementing half of a function, getting distracted with ideas in my head and moving onto a completely different function. Although it more just delayed the process than stopped it altogether as some of the functionality added may not be necessary. I also think that it helped me to notice when mistakes were made slightly sooner than otherwise which means that I don’t need to then later refactor as much code to get functionality to work properly.

Unfortunately, I did not have much experience with Haskell and it took me a while to adjust to a functional programming language, especially with some of the syntax being very different to C-style object oriented languages. I also had next to no experience using quickCheck which took some time to get used to as it was also the first time I had tried property-based testing. Changing my way of thinking was quite difficult and also took a while. This combination of inexperience led to the mistakes that I made along the way, and why it took me quite a while to sometimes see them.

As can be seen, I did not finish implementing all of the functionality that I wanted. This is mostly because I did not know how to craft the tests to properly test the functions. For my own curiosity I did actually implement the ability to print out the dictionary in order and it seemed to work when printed in the console but I couldn’t work out how to test this functionality using property-based testing.

For small personal projects I think I may use TDD but using unit tests instead of property-based testing more as a form of sanity check than for serious development. I am okay at designing tests to check specific aspects of a function but I am not as good at designing a test that checks functionality and make a function to match it. I also highly doubt that I will use Haskell, especially with quickCheck, unless the project I am doing is much more conducive to a functional language than an object oriented one due to the low level nature and efficiency available.

In a team, with a mix of experienced and inexperienced TDD developers then it may work better in a workplace but otherwise it puts too much pressure on being able to design tests that are useful with nobody to double check your work.

# References

A. Cauevic, S. P. a. D. S., 2012. Quality of Testing in Test Driven Development. *2012 Eighth International Conference on the Quality of Information and Communications ,* pp. 266-271.

A. Causevic, D. S. a. S. P., 2011. Factors Limiting Industrial Adoption of Test Driven Development: A Systematic Review. *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation,* pp. 337-346.

D. Fucci, H. E. B. T. M. O. a. N. J., 2017. A Dissection of the Test-Driven Development Process: Does It Really Matter to Test-First or to Test-Last?. *IEEE Transactions on Software Engineering,* 43(7), pp. 597-614.

Dogša, T. B. D., 2011. The effectiveness of test-driven development: an industrial case study. *Software Qual J,* pp. 643-661.

Mäkinen S., M. J., 2014. Effects of Test-Driven Development: A Comparative Analysis of Empirical Studies. *International Conference on Software Quality: Model-Based Approaches for Advanced Software and Systems Engineering,* pp. 155-169.

Steinhauser, J., 2018. *Intro to Property-Based Testing.* [Online]   
Available at: https://dev.to/jdsteinhauser/intro-to-property-based-testing-2cj8  
[Accessed 25th May 2021].

Wilson Bissi, A. G. S. S. N. M. C. F. P. E., 2016. The effects of test driven development on internal quality, external quality and productivity: A systematic review. *Information and Software Technology,* Volume Volume 74, pp. Pages 45-54.