Test Driven Development – Case Study

Advanced Software Engineering

Barry O'Connor - N0813926

Contents

What is Test Driven Development?

Test Driven Development (TDD) is an approach to development, formalised by Kent Beck, which utilises an incremental process of writing tests, producing code to meet those tests and refactoring before repeating the process. There are 5 key steps involved in the process and these can be grouped into three phases, the following list is colour coded to match with the phases in Figure 1.

1. Create a minimal test
2. This test should fail when the tests are run
3. Make a small change to the code to allow the tests to pass
4. The tests should now pass when run
5. Refactor to remove duplication

The process relies on progress being made through the use of small steps and tests are written, tested and completed with code one at a time, rather than creating a full suite of tests before coding such as Test First Development. Each test is designed to use the language of the envisioned final functionality as if the developer was a user in the future trying to use the code and it is intended that tests initially either fail as a test or fail to compile in cases where the test uses an object or method of a class which does not exist yet. Coding can only happen when a failing test is present and once this happens, the developer writes the least amount of code to allow the test to compile or pass. Once the failing test passes, code can be refactored to remove duplication or improve code quality and once this is completed, the whole process repeats. This results in a body of code that has been tested at every step and for which there exists a full set of tests that can be quickly run to perform regression testing or form the basis for any future development on the code.

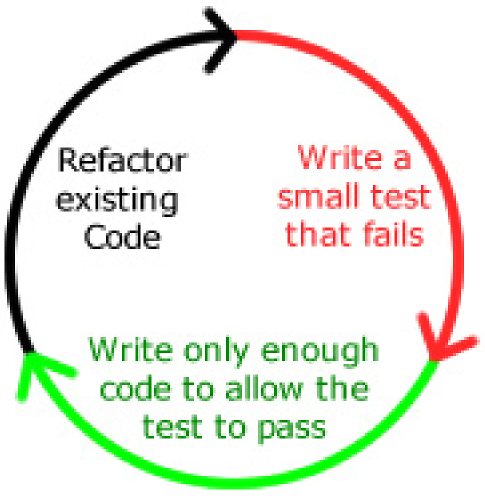
Refactor existing Code
Write a small test that fails
Write only enough code to allow the test to pass

Figure 1: The three phases of Test Driven Development

How to start a project using Test Driven Development

As with any testing, there is a need to have some definition of complete for the final code, so it can be useful to have a set of acceptance conditions or goals that the final code must pass to be considered complete. These can be at quite a high level and for commercial software projects, would generally be made up of the requirements for the project which would be created using use cases or user stories. It is important to remember that these are not tests themselves and each of these conditions may eventually be made of multiple tests, they are simply there to provide an overview of the features the software must have and to help define when development has been completed. To give an example, if the software was a basic calculator, it could be considered complete code if it can perform multiplication, addition, division, subtraction and can reset the calculator between calculations. This gives us a clear set of conditions that we can work towards and when we have code that performs all of these it can be considered complete. Under each, we can make a note of any relevant specifics that need to be handled such as division by zero for the calculator example.

It does not matter in this example which of the conditions we begin with but to reset the calculator there would need to be some value in the calculator in the first place and it would make sense to include any tests for division by zero after the code works as these are connected. As such, it might be logical to start with any of the other three. In this example, if addition was chosen, the first test could be written and that test might simply be to check that 1 + 2 = 3. At this point, we have the starting test and can begin the TDD process by starting the Red or Testing phase.

Red Phase

The red phase is the first phase of the TDD process and is primarily concerned with the first two steps in the list of five. These steps suggest writing a test that should cover a small increment of functionality and should ultimately fail. For this process, compilation issues such as classes, functions and variables not existing can be considered to be a failure. It is important to write tests that use the code as if the developer was the end-user and to write tests that use the terminology of this finalised version of the code even though it does not currently exist. This will help with the process of building the code and so using the terminology, even at the very start of the project will ensure that the correct function and variable names are being used and that the developer does not have to repeatedly go back and change names within the code.

It should be mentioned that additional tests which automatically pass, such as testing working functionality with larger amounts of data can be repeatedly implemented at this phase once the initial phase has completed. In the calculator example, this might simply involve adding several numbers together rather than two and since the initial test has passed after the first cycle, this should automatically pass provided no underlying issues are present. These type of tests can be added as needed and the red phase will simply cycle within itself until a failing test for new functionality is created.

Green Phase

The green phase is the second phase within TDD and is concerned with steps three and four in the list of five. At this point, it suggests writing only enough code to allow the failing test that was previously created to compile and pass this phase will remain current until all tests are passed. It is important to remember that the goal is always to take small testable steps rather than creating perfect, beautiful code and sometimes this may mean taking steps that feel ugly, uncomfortable or frustrating especially to those with programming experience. By taking such small steps, over-engineering the solution is prevented meaning that time is only spent on creating code that creates the required functionality.

Refactor phase

The refactor phase is concerned with the final item in the list of 5 which is the refactoring of the existing codebase. This is the point where the developer can consider code quality and make improvements to the codebase to improve readability, remove duplication and generalise the code so that it is not reliant on hardcoded values. Refactoring in this way should always try to be as minimal as needed, focusing on the quality of the code rather than trying to introduce new features.

It is also important to remember that refactoring can allow changes to the code but not to the result and any changes made must still allow the tests to pass so some refactoring of the tests may be needed to accommodate changes to things like function names. Once refactoring has been completed on the code and tests, the cycle is complete and the red phase begins again.

Generally Accepted Benefits and Drawbacks of TDD

To critically evaluate the process, it is vital to understand the generally accepted drawbacks and benefits of the TDD approach. The following lists have been compiled from several websites [GeeksForGeeks (2020), Infoworld (Shronek, K., 2018) and AgilePainRelief (Levison, M., 2008)] with pages focusing on this topic and, while not academic, give an overview of the generally held perceptions developers have regarding TDD. This will form the basis for discussion on these points in the conclusion section of this case study.

Generally Accepted Benefits

- **Only code needed for a feature is written:**
  Since each cycle of TDD encourages the developer to only write code that allows a test to pass or some refactoring, the codebase should only consist of the needed code.

- **Leads to modular code**:
  developing one function or feature at a time and creating unit tests for that function or feature results in more modularity.

- **Easier to maintain:**
  since a test suite already exists, even drastic changes can be tested and any issues that arise in future changes can be fixed using the existing tests. These can be run frequently during an ongoing project to ensure everything works.

- **High Test Coverage:**
  Each feature has a test and so this offers a higher test coverage for the code.

- **Helps in documenting software:**
  Since the code is developed to meet needs, the test code only covers the essentials and less time is spent on embellishing.

- **Easier to Test:**
  Mistakes are caught early by frequent testing and any existing code has a matching set of tests that can be run automatically at very little cost in terms of time.

Generally Accepted Drawbacks

- **The whole team must use TDD:**
  There must be a general acceptance of the practice and everyone within the team must use the practice otherwise issues will arise.

- **The process can be slow initially:**
  TDD can be slow initially and will incur more overhead throughout the project than other

testing methodologies.

- **Tests must be maintained alongside code:**
  Tests must be maintained alongside code and should updates break the tests, these must be fixed alongside the codebase, meaning additional maintenance.

- **TDD can take some time to learn:**
  Since TDD can take a while to learn, a developer new to the concept may suffer decreased productivity until they have learned the process.

Case Study

When researching the testing environments open to Haskell development, Tasty seemed to be popular and was derived from other libraries such as HUnit and QuickCheck and would allow for both to exist side by side which seemed relevant for this case study. Tasty was selected and so the implementation of tests will look slightly different but grouping and labelling will be used to make it clear what the tests are doing.

Before

Thoughts Before Starting the Case Study

I deliberately wrote the first part of this document before starting to code as I wanted to approach the coursework genuinely and see how my thoughts changed during the process. I have had several interviews asking for Functional Programming skills (although GoLang and ErLang seem to be the languages of choice) and so I thought that it might be interesting to put down my thoughts before starting and at the end so that a comparison can be made to see if my thoughts changed due to the experience.

Haskell

I have not been looking forward to this case study due to the use of Haskell. Initially, I found the language to be indecipherable and this is not greatly helped by online tutorials which revolve heavily around one-letter identifiers. I am sure this is perfectly understandable once you learn the language but having parameters identified by random letters in most tutorials and having to count to understand which parameter is being referred to makes the learning process harder than it needs to be. Hopefully, if I bear this in mind in my code and refactor it to be readable, this will make things more understandable. As I mentioned before I am seeing Functional Programming as being standard in fast-paced online businesses, although I have not seen anyone require Haskell yet so even if I do not gain a lot of love for Haskell by the end, I hope that the ideas behind the language might be useful in picking up GoLang or ErLang in the future.

TDD / Property Testing

Test First development was not a style I felt comfortable using since I am naturally uncomfortable about making concrete decisions on uncertain things and was uncomfortable with the idea of having to create all of the tests before code. Knowing that I could not possibly predict every outcome meant tests felt like educated guesses and assumptions and I did not ever feel like the code was bulletproof. As a perfect example of this, I still regularly have issues entering my Surname into many websites, where the developer simply has not ever thought of a surname containing an apostrophe.

My natural coding style would probably resemble a reversed TDD with bigger steps between each phase in that I would tend to write a small piece of code, test it and then refactor but I would also have a bigger set of testing at the end. I will admit that the interim tests generally would be manual so It will be interesting to see if I adjust to TDD quickly because of this similarity and whether it offers noticeable benefits or a smoother as I complete the coursework.

Property testing seems like it could be useful as it can test hundreds or thousands of combinations without needing any input from the developer and so would suggest that it could be used to spot issues that the developer could or would not foresee. My initial thoughts are that this is a good tool to have in your toolbox but I am uncertain how much extra work would be needed to implement this

Definition of Done

To understand a rough plan for the testing, I have created a set of bullet points to outline the conditions that need to be catered for in order for the software to be complete. As mentioned previously, these may represent multiple tests and this list may be expanded but it seems a good idea to have a rough framework plan in place at this point. The main points are taken directly from the coursework and marking grid in an attempt to emulate requirements and any additional ideas or edge cases and boundaries that I think are useful during the process will be added as a second-level item purely to differentiate.

- Create an empty dictionary.
    - If we need to verify at multiple points perhaps a helper function needed?
- Insert an entry.
- Lookup the item by specifying a key.
- Produce a list of all entries (probably in order traversal).
    - Maybe need to specify if an item is a leaf
- Remove an entry by specifying the key.
    - Special cases for removing a root rather than a leaf
- Remove all entries that satisfy a specified predicate function.
- The BST does not need to self-balance (so no action needed).
- Implement polymorphic operations.
- Use Unit testing and Property Testing.

During

Test Driven Development – example cycles

For the implementation, I used the Tasty library which allowed me to use HUnit and QuickCheck together and made a little more sense in terms of how to layout tests and group them without having to create functions for each.

First Cycle

Red Phase

To begin the process, I decided that the most fundamental step, to begin with, was the first and so a test was created to compare the result of the create function with an empty BST.

```
testCase "create_compare_with_leaf" $ assertEqual "create and compare to leaf"
 Leaf (create)
```

Due to there being no code, this failed compilation and so matched the requirement to have a failing test.

Green Phase

To make the code compile a new module called BST was created and the following minimal code was implemented. Since the function needed to return a BST, the simplest solution was to have the function return a Leaf since this was the only option open to us.

```
data BST = Node Int String BST BST
    | Leaf

create :: BST
create = Leaf
```

Creating the data type and the function removed some of the compilation errors but a new compilation error occurred, this error was due to comparing our custom data type without deriving it from the Eq class which allows for operations such as this. This was remedied by changing the definition to the following. This action allowed the test to pass.

```
data BST = Node Int String BST BST
    | Leaf
    deriving (Eq)
```

Refactor phase

At this point, no changes need to be made since the names for the data type (BST, Node & Leaf) are all clear and the function itself is very small. The definition is not polymorphic and will only work for a string and an integer so this may need to be addressed but since there is no test to check this yet, I just made a note to do this and moved onto the next round.

Second Cycle

Red Phase

With the first function simply returning an empty BST there were no further tests needed and so I moved onto the next set of functionality which involves the lookup function. At this point, the only tree available to us would be a manually defined one or the empty one created by the create function so we may as well use the create function. It will be necessary to check if the supplied BST is empty since this will never match any node so the following test was written which had compilation errors.

```
testCase "lookup_in_emptyBST" $ assertEqual "lookup value 22 in empty BST" ""
(BST.find 22 create)
```

Green Phase

To make the code compile a new function was created and the simplest way to make this pass was to have the function return the value that is being tested for.

```
lookup :: Int -> BST -> String
lookup a Leaf = ""
```

This worked and the test passed the first time.

Refactor phase

At this point, the function works but there is duplication because the return value and test value are the same hardcoded empty string and the function is only likely to work for this particular test case. Refactoring to remove the value required an alternate way to handle returning a value or no value since we will eventually need to return both and in Haskell, a Maybe type can be used for this purpose so the code was amended and the first parameter renamed to something more informative as follows.

```
lookup :: Int -> BST -> Maybe String
lookup a Leaf = Nothing
```

The test needed to be updated to replace “” with Nothing to complete this refactoring.

Sixth Cycle

Red Phase

At this point there is a full lookup function and a set of tests that have all passed, however, the function and the underlying BST class are not parametrically polymorphic and so it is a good time to look at converting these over. To allow for this, another test is created to test that the function can work with a Char key and an Integer item and a suitable BST is pre-prepared.

```
parametricBST :: BST
parametricBST = Node 'c' 66 (Node 'a' 44 Leaf Leaf) (Node 'z' 1 Leaf Leaf)

testCase "lookup_polymorphic" $ assertEqual "lookup 'a' which is > root value"
 (Just 43) (BST.lookup 'a' parametricBST)
```

Green Phase

In an attempt to make the code compile the BST data type and function were changed to the following:

```
data BST a = Node a a (BST a) (BST a)
    | Leaf
    deriving (Eq)
```

however, this did not work as the BST has both a key and item values that need to be polymorphic and so this had to be changed to allow for both items to be independent of each other so I changed the data type to read:

```
data BST a b = Node a b (BST a b) (BST a b)
    | Leaf
    deriving (Eq)
```

The existing function declarations were amended to match the new format BST and the tests passed.

```
create :: BST a b

lookup ::(Ord a) => a -> BST a b -> Maybe b
```

Refactor phase

While this worked, I found it confusing to read and understand what k and v were and I opted to use the names of the actual elements throughout the code. This change should also help in making other functions more readable as I can reuse terms like key, item, leftChild, rightChild. Replacing a and b with "key" and "item" makes the code more readable

```
data BST key item = Node key item (BST key item) (BST key item)
  | Leaf deriving (Eq)
```

```
create :: BST key item

lookup ::(Ord key) => key -> BST key item -> Maybe item
```

Twelfth Cycle

Red Phase

At this point, things have moved onto looking at the ability to convert the tree to a list of String which will give a list back and more importantly give the ability to check the order of inserted items to make sure that the previous insert function is not just randomly inserting items. The eleventh Cycle focused on testing an empty list that was returned for an empty BST so now the focus is on testing for non-empty BST. The following test and list based on the nonEmptyBST tree were added to the tests file.

```
nonEmptyBSTAsList :: [String]
nonEmptyBSTAsList = ["22 - \"Mary\"", "11 - \"John\"", "33 - \"Peter\""]

testCase "display_non_empty" $ assertEqual "display a non-
empty BST" nonEmptyBSTAsList (BST.treeAsList nonEmptyBST)
```

Green Phase

The simplest way to make this test work was to understand that a string would be enclosed in \" in Haskell and this would allow us to simply build the string for the current key and item and add this to the list then recursively do the same for the left and right child which results in an Inorder Traversal, other traversal methods can be made by changing the order of the current, left and right node but this gives an order similar to the way the tree would be drawn and so made sense for comparison.

```haskell
treeAsList :: (Show key, Show item, Eq key, Eq item) => BST key item -
> [String]
treeAsList Leaf = []
treeAsList (Node key item leftChild rightChild) = [show key ++ " - " ++ show i
tem] ++ (treeAsList leftChild) ++ (treeAsList rightChild)
```

Refactor phase

The above code does work and allows the refactor phase to begin, however, it does not show any indicator for a leaf node and to be honest the included \" is ugly and will result in strings like "22 - \"Mary\"(*)" if we implement the asterisk to denote a leaf, this is awkward to type so I felt that it needed to just be a clean string.

Firstly, I refactored the test to include the "(*)" and then refactored the treeAsList function to include an if statement which would identify a node with two leaf children and I added "(*)" onto the end of the string for that node.

```haskell
treeAsList :: (Show key, Show item, Eq key, Eq item) => BST key item -
> [String]
treeAsList Leaf = []
treeAsList (Node key item leftChild rightChild) =
  if leftChild == Leaf && rightChild == Leaf then
    [show key ++ " - " ++ show item ++ "(*)"] ++ (treeAsList leftChild) ++ (tr
eeAsList rightChild)
  else
    [show key ++ " - " ++ show item] ++ (treeAsList leftChild) ++ (treeAsList
rightChild)
```

This passed the amended test however, I still preferred the look of a string without the included escaped quotes so I decided to use a filter to remove this and since that would get messy, I decided to implement a helper function which would take the item as input, show the item so that it would be converted to a string and then apply the function to remove the \" and return the string. This resulted in the following:

```haskell
noQuotes :: (Show a) => a -> String
noQuotes input = filter (`notElem` "\"\'") (Prelude.show input)

treeAsList :: (Show key, Show item, Eq key, Eq item) => BST key item -
> [String]
treeAsList Leaf = []
treeAsList (Node key item leftChild rightChild) =
  if leftChild == Leaf && rightChild == Leaf then
    [noQuotes key ++ " - " ++ noQuotes item ++ "(*)"] ++ (treeAsList leftChild
) ++ (treeAsList rightChild)
  else
    [noQuotes key ++ " - " ++ noQuotes item] ++ (treeAsList leftChild) ++ (tre
eAsList rightChild)
```

Conclusion

Functional Programming / Haskell

I found the experience difficult since I am not from a mathematical background, coming from a languages background instead so a lot of the terminology involved was out of my depth and I struggled with even understanding the language at a basic level. It was extremely frustrating because I could understand the ideas behind the language was asking me to do, so I understood immutability, pure functions, recursion, guards, use of let and where but the basics like “How do I create a variable” took me far too long to understand and I am honestly not sure I completely do. As I mentioned previously, a lot of the online tutorials are terrible at explaining the language and have very heavy use of single-letter identifiers and mathematically heavy explanations. This resulted in maybe 4 or 5 times the amount of research I would normally need.

I found basic functional programming fairly quick once I had understood the code a little better. It was very alarming at first to find that stub functions containing essentially what I considered pseudocode simply worked the first time which I would not expect from any language. This did initially make me doubt the code since there is little chance of getting it right the first time in a new language but as time went on, I started to accept this and it was a little enjoyable. The perfect example is the higher-order deleteIf function where I must have spent 24 hours looking for help in how to do it and read up on folding, zipping, fmap, functors and more. None of these helped and in the end, I just sat down and wrote a rough outline of how it should work and this ended up being about 90% correct. I had not thought to re-apply the function to the resulting tree from the deletion and so was missing cases where the root was replaced by another node that also matched but once I included this, it all just worked.

My BST implementation in C++ works out at around 380 lines of code (including line spaces) while the Haskell implementation is around 140. In terms of timescale, the original C++ BST implementation took around 2 days for me to complete while the Haskell implementation took around 3.5 to 4 days however, the Haskell implementation was in a completely new language and with a completely different paradigm at play so I would estimate that 2 of those days were lost on getting to grips with the language and that timeframe also includes a full suite of unit tests. If that is taken into consideration, then Haskell was probably a lot faster to implement and given the same experience level as C++, I would expect that those figures would be reduced to a matter of hours.

Functional programming seems to be weaker in terms of UI than some languages and reminded me a little of C++ which is also great at computation but fairly bad at creating beautiful user interfaces. I would imagine that functional programming would work very well as a sort of middleware, where it represents a business logic layer between a UI and a database for example. It does seem to have a focus on functions that operate on data which would back up the idea that some sort of processing or business logic would be a good use.

Property Testing

The use of QuickCheck, was nigh on impossible for me. Again, I understood what I was meant to do but how to achieve that disappeared in a storm of monads, functors, foldr functions, lifting and type mismatches. I managed to get a generator up and running after at least 50 hours of searching and looking at tutorials but the generator was terrible and simply generates data randomly in a BST shape. I know it is possible to somehow tie this generator into your insert function so that it inserts data via your code, which results in better test data but despite many, many hours of trying, I had to give up. In the end, this means that several of the properties I wanted to test just would not work with the test data but would work if I manually used them. For example, I had intended to test for a valid BST but because the data was random, it would place larger numbers on the left and smaller numbers on the right frequently. While I can see the benefit of property testing and can see that it gave me more confidence in my code being robust, on this occasion, I just could not get the hang of Haskell well enough, hopefully, it will be more intelligible in another language or with a training course. I would like to continue investigating Property Testing and will do so over the summer.

Test Driven Development

The most obvious benefit of TDD is that it is easier to test and this needs little justification. Logically, testing throughout the life of the project will catch more bugs and the focus of TDD on creating failing tests and then coding to allow them to pass makes sure that practically every written line of code is being tested. Having a set of regression tests will make any testing faster since it gives the ability to simply run those tests and get a set of test results for the code in seconds.

I have to admit that I liked the TDD process. Replacing my ad-hoc form of testing with structured tests was not a big change and despite the steps sometimes feeling a little smaller than I wanted them to be, coding and testing at the same time felt very logical and natural and allowed a steady progress throughout. I found myself frequently stopping myself from implementing additional features or functions which occurred to me and this is something I do tend to be bad at so from my own experience I do think that TDD helps a developer stick to the road and not code additional features without justification.

In terms of modularity, this is fairly obvious since each function or feature is tested independently. This focus on testing interfaces for unit testing will enforce modularity since each piece of functionality is tested independently and this will result in the ability to reuse working functions which is the definition of modularity.

Knowing that completed functions were repeatedly passing tests allowed me to rely more heavily on these without having to be overly concerned with their correctness. For the larger functions, testing and coding almost worked outline for line and each test would be a single idea or condition and the resulting code would simply deal with that one piece. This allowed me to consistently take solid steps towards completed functionality and to break down larger ideas into smaller ideas and deal with those individually. This is backed up by Romano and others (2017) who claim that developers had greater faith in their tests when using TDD and would refactor and commit these to a repository fairly infrequently with some pairs only ever doing this one time for each test. However, this effect could be due to other factors and might suggest bad habits rather than faith in the tests.

I have been coding long enough to be able to instinctively break most problems down and be able to identify a rough roadmap in my head and this worked well with TDD. I was able to match tests with my roadmap and at any point was able to add extra tests if I felt they were needed or change previous tests if the code evolved and this felt much more useful. This allowed me to create tests for conditions I had not expected or to change my mind about how to test and code as I went and it felt much more reliable. This spread of testing activity throughout the project, rather than at the start was observed by Romano et al. (2017) and Papis et al. (2020) found a difference in their competence groups, suggesting that experienced developers may find TDD easier to understand and may value the testing more readily than novice developers. This might also go some way to explain the preconception that TDD can take some time to learn since this study showed that novices took longer to start with TDD. Their study also showed that developers would plan a model of the software in their mind in advance and would work towards this in a white-box approach rather than a black-box approach as TDD suggests and this is perhaps backed up by my own experience. It is also worth mentioning that Scanniello et al. (2016) studied students and professionals during a TDD study and suggested that students had a harder time with TDD simply down to a lack of experience with the more complicated concepts within unit testing such as Mocking which may explain the difference in speed of understanding.

At several points, I had to go back and refactor tests to match refactored code and I can see how developers might find this a drawback in that maintenance would involve tests and code, however, I did not find much additional time was taken up by this and I certainly felt that any cost was offset by the knowledge that I could make the changes and be confident that the tests would catch any errors I made which I feel gave me greater confidence to go back and fix sub-optimal code rather than being scared of changing it. Romano et al. (2017) commented that developers in their study showed a strong preference for the green phase of TDD, however, this study was set in an academic setting and perhaps this would suggest junior developers who would be more likely to value the fun aspect of creating code rather than the reassuring aspect of having a solid set of tests to rely on. This study also found that developers did not refactor as often as TDD suggests they should and perhaps this reluctance to refactor and to "save" refactoring until it can include a larger bout of coding lends itself to promoting the thought that TDD takes more effort to maintain.

The research pointed to TDD taking more time to implement than other strategies but my own experience did not match with this. This may be a personal thing as I do tend to manually test as I code so perhaps my coding style is slower than that of others who prefer to write all the code and then test. I did not notice a major difference in the amount of time it would take me to do a manual test compared to the amount of time spent on TDD. It took seconds to write a test and even though I made a mental note that I had probably run the tests a few hundred times during the lifetime of the code, this did not feel like it took up a lot of my time.

However, that is not to say that TDD is a silver bullet. All testing is only as good as the imagination of the developer and so can be prone to issues when given to an end-user. Scanniello et al. (2016) suggest that TDD is best when used in unfamiliar territories such as exploring new directions or new features of a language as it allows the developer to take small steps and to assure themselves through the use of testing. This seems very true to me and is certainly the area where I think TDD would offer me the greatest benefit and I found this explanation is used by Kent Beck frequently in various articles.

Interestingly, there is debate on the effectiveness of TDD in the academic sphere and the validity of many previous studies has been called into question. Bissi et al. (2016) examined many previous studies and found some interesting results. In general, TDD created better external and internal software quality in almost all studies, however, the various studies were split into Academic and Industrial categories and this showed that productivity was almost always better in an academic setting but was almost universally worse in an industrial setting. This perhaps suggests that the structured approach is of more benefit to students who are more able to adjust their behaviour and be less likely to stick to known methodologies.

Karac and Burak (2018) also called the previous studies into question, showing that many had inconclusive results and tended towards flawed comparisons with projects which focused more on Waterfall methodologies. They go on to discuss the reasons for the lack of uptake within industry and state that the increase in time taken and drop in productivity coupled with the presence of legacy code or need to work with domain or tool-specific code. This seems likely as it would be extremely difficult to retrofit an existing codebase to work with TDD and so developers would be more likely to consider TDD for new projects.

In an attempt to work out the penetration of TDD on Github projects, Borle et al. (2018) used data mining techniques and a set of rules such as checking logs to see if test files were committed before code in an attempt to find projects which had focused on TDD. The investigation found little in the way of interesting differences in commit frequency, pull requests, issues or number of bug fixing commits generally and in fact comments are made to this end but what is interesting is that the uptake of TDD seems to be very small. The investigation focused solely on Java and so is not representative across all languages but it found that only 0.8% of all GitHub repositories in 2015 were following TDD practices. This might lend credence to the generally held feeling that if the whole team is not using TDD, then nobody is.

Bibliography

BALDASSARRE, M.T., et al., 2021. Studying test-driven development and its retainment over a six-month time span. *The Journal of Systems and Software,* 176.

BISSI, W., SERRA SECA NETO, ADOLFO GUSTAVO and EMER, MARIA CLAUDIA FIGUEIREDO PEREIRA, 2016. The effects of test driven development on internal quality, external quality and productivity: A systematic review. *Information and Software Technology,* 74, 45-54.

GEEKSFORGEEKS, 2020. *Advantages and disadvantages of Test Driven Development (TDD)* [online]. . Available at: https://www.geeksforgeeks.org/advantages-and-disadvantages-of-test-driven-development-tdd/ [Accessed Jun 9, 2021].

KARAC, I. and TURHAN, B., 2018. What Do We (Really) Know about Test-Driven Development? *IEEE Software,* 35 (4), 81-85.

LEVISON, M., 2008. *Advantages of Test-Driven Development | Agile Pain Relief Consulting* [online]. . Available at: https://agilepainrelief.com/blog/advantages-of-tdd.html [Accessed Jun 9, 2021].

PAPIS, B.K., et al., 2020. Experimental evaluation of test-driven development with interns working on a real industrial project. *IEEE Transactions on Software Engineering,* , 1.

ROMANO, S., et al., 2017. Findings from a multi-method study on test-driven development. *Information and Software Technology,* 89, 64-77.

SCANNIELLO, G., et al., 2016. Students' and professionals' perceptions of test-driven development: a focus group study. *PeerJ PrePrints,* .

SKRONEK, K., 2018. *From BDD to TDD, the pros and cons of various agile techniques* [online]. . Available at: https://www.infoworld.com/article/3269039/from-bdd-to-tdd-the-pros-and-cons-of-various-agile-techniques.html [Accessed Jun 9, 2021].