# Section 1 – Test Driven Development

## Introduction

Test Driven Development (TDD) is an Agile software development practice where automated unit tests are written incrementally during the development of the software before the source code is fully implemented (Beck, 2003). It requires test cases to be defined from the requirements of the software and implemented in an automating unit testing program before the relevant area of functionality is developed. The software is written in response to the test cases, as opposed to the test cases being developed after the software is fully implemented. TDD is one of the most frequently used practices in Agile software development. It began gaining popularity when it was defined as a core part of Extreme Programming (XP), however it has now gained independence and become popularised throughout many other paradigms (Beck, 2000).

TDD emphasises a specific cycle to be followed throughout the software development process. This is an iterative process where first, the software engineer considers a specific piece of functionality that needs to be developed within the program, and the inputs and outputs that would be expected. A test case is then defined and implemented, which should assess for that specific functionality. The tests are all executed, with the expectation that the new test should fail. The area of functionality has not been developed yet, and this proves that the new code that is being considered is needed, as well as assessing whether the new test case is flawed or not. Code that passes this test case should then be written within the program. This code should only concern the newly added test case, and not have any extra functionality. It is not necessary for the code to be efficient or elegant, it should pass the test in the simplest way possible. All test cases should be executed again, with the expectation that they all pass. If the new test case does not pass, the newly implemented code should be revised until it does. The new code should not break any pre-existing test cases. Finally, the code should be fully refactored to ensure that it is efficient, elegant, readable, maintainable, and meets any defined coding standards for the project. The test suite should be executed after each refactor, to ensure any functionality is not broken. The test suits may also be refactored in this stage, to ensure that all test cases are maintainable and only test for their specified functionality. This is one iteration of TDD, which is repeated for each new piece of functionality until the program is complete. Each iteration should be small and incremental (Beck, 2003).

## Critical Evaluation

TDD has a variety of benefits for the software development process. A meta-analysis assessing the effectiveness of TDD on internal and external quality, as well as productivity, found that 76% of studies analysing the TDD process found a statistically significant increase in internal software quality, and 88% found a meaningful increase in external software quality (Bissi, et al., 2016). This suggests that TDD is able to produce higher quality, bug-free code (Tosun, et al., 2018). This could be due to the idea that code is written only to pass a single failing test case, and not any further. This means that unexpected functionality cannot be added, and code quality for individual sections is thoroughly considered through refactoring. Additionally, TDD leads to more modularised and extensible code as it encourages the software to be developed in small, independent units to pass a single test case.

Furthermore, while TDD requires a higher amount of code due to the parallel development of a unit test suite, it has been suggested that the total time of implementation could be shorter. The large number of tests and focus on the functionality passing all required unit tests, means bugs within the program code can be limited and reduced. The nature of TDD assists in identifying and fixing any bugs that arise early in the development cycle, which can prevent them from becoming larger issues in later stages of software development (Muller & Padberg, 2003).

Additionally, a series of laboratory experiments with over 200 participants found that TDD offers better modularisation of program code, easier reuse and testing of developed software products, and increased thoroughness of unit tests than alternatives such as Test Last Development (TLD) (Madeyski, 2010). This can be corroborated through a comparative review of various testing paradigms, which found that TDD has many benefits over classical approaches such as higher quality code and high test coverage, however some iterations over TDD such as Behaviour Driven Development (BDD) have advantages that address some issues with TDD, such as better communication between developers, testers, and product owners (Moe, 2019).

However, it has been found that TDD may lead to decreased productivity within industrial scenarios. 44% of studies identified lower productivity in TDD than in TLD (Bissi, et al., 2016). This may be due to the fact the TDD encourages deep consideration for individual components and places many restrictions on how software can be developed. This means that small changes within the code will need to undergo the full, iterative process before implementation, which can drastically reduce speed and efficiency, especially within large scale projects (George & Williams, 2004). Additionally, a study comparing TLD to TLL found that the majority of the developers assessed stated they preferred TLD to TDD, due to less effort required and a lower learning curve (Munir, et al., 2014). This could contribute to decreased productivity as developers within industrial settings may not enjoy implemented TDD.

This can be reinforced by a study performed to assess how TDD is used by both novice and experienced developers within the industry, which found that many developers do not perform the refactoring stage as often as is required. The unit tests are not kept up to date, and many developers design a model of the source code in their mind and build unit tests based on that (Romano, et al., 2017). This suggests that when TDD is applied in professional settings, it is often not followed properly and may lead to lower quality code and testing suites.

Personally, I have had many interactions with both TDD and TLD throughout my academic and personal lives. Through my personal experiences, I have found that TDD can lead to a better understanding of the core parts of the program, and how they are expected to behave. This has lead to more focused, and higher quality code that was able to be completed in a more structured and efficient manner than when utilising TLD, which is in line with the findings of many research papers that have been discussed. I found that a university project involving refactoring code was made much simpler and easier to undertake with a suite of predetermined unit tests, as I was able to determine the precise requirements of the system and how it needed to behave. I could then modularly implement functionality for each unit test and complete the assignment with efficiency. This contrasts with a university project concerning the development of a system utilising TLD. I found that this project was much less structured, and requirements needed to be adjusted, added, or removed throughout the course of development. Development of this system was more problematic, as many bugs and issues arose throughout the development process.

# Section 2 – Applying TDD

For this project, I fully implemented a Net Income calculator, factoring in student loan, national insurance, and income tax based on the inputted income. I applied the full TDD process throughout the implementation and will describe four iterations of the process within this section.

## Iteration 1

At the beginning of the implementation of the program, I started with the development of the income tax calculation function. I decided to begin here as it was the most complex and would provide insight into the other areas of functionality.

The first item to implement was if no income tax needed to be paid, in the case of the inputted income falling below the personal allowance threshold of £12,570. The first part of TDD is to implement a unit test. Therefore, a stub function declaration was written, that was assigned to be undefined, and a unit test was implemented that would check whether the output of the function is “0”, based on input data of “10,000”, which falls below the personal allowance threshold. The test suite was executed, and the newly defined unit test failed, suggesting that the functionality needed to be developed.

The function stub was then expanded, implementing an “if” statement that checked whether the inputted income was below “12,570”. If it was, the program would output “0”, which is the expected output of the unit test. As no other functionality should be developed, the function was set to output a “1” if the income was above the threshold. The test suite was then executed again, and the new test case had passed. This suggests that this area of functionality had been successfully developed without any areas of concern. Then, refactoring of the code took place, where the personal allowance threshold was moved into a global variable to avoid “magic numbers” and facilitate for any future changes to the value.

Following the refactoring of the code, the unit test suite was executed once again to ensure that the refactoring did not introduce any new issues. The test suite was successful, and therefore I could move on to the next iteration for the next piece of functionality for this function, which was the basic rate income tax, followed by the higher rate income tax. The full TDD process was followed for both of these, and no significant issues arose.

## Iteration 2

A significant area in the process of developing the income tax calculation function was the implementation of the higher rate calculation. First, the accurate test case was identified and implemented in the testing suite. If the income is higher than £150,000, additional income tax applies. The amount used within the test case was “175,000”, and an external website was used to generate testing data for the expected result of this value, of “63,710”. The test case was executed, and it failed, and therefore the functionality could now be developed.

Code was reused from the implementation of the basic tax rate and higher tax rate, in order to calculate 20% of the first set of income minus the personal allowance, and then 40% on the second. Then a calculation was applied to calculate 45% of the remaining income and return the sum of each calculation. Upon running the test suite, it was found that the expected results did not match the output, and therefor this calculation was incorrect. This means that this section of the implementation needed to be refactored for it to pass the test.

Through research, it was found that a more accurate method of calculating income tax would be to first calculate the total taxable income based on the personal allowance, and calculating income tax on that, rather than deducting the personal allowance at the time of calculation. This meant a different function needed to be added to calculate this value and allow for reuse throughout the program, following the methods of functional programming. The program was therefore reverted to a state in which all unit tests were succeeding, in order to focus on one area of functionality. A stub function as added for the calculation of taxable income. A new unit test was then defined for no taxable income. The input value was “10,000” and the expected value was “0”. The unit test failed when executed, and so this area of functionality was implemented using an if statement. If the user’s income was below the personal allowance threshold, they would have no taxable income and the function would return “0”. Otherwise, it was set to return “1”. Upon testing, this piece of functionality was successful, and no opportunities for refactoring were identified.

Therefore, next, a unit test was implemented in case of having any taxable income. An input value of “50,000” was used, and an expected value of “37,430” was also used. If the income was above the personal allowance threshold, the personal allowance would be deducted from it. This would result in the total taxable income. The testing suite was executing, and no errors were detected. Now the income tax function would need to be refactored in order to utilise this function. After each line of refactoring the calculations performed by the function, the testing suite was executed to ensure there were no issues that resulted in the function no longer working. Finally, the failing unit test for the additional tax rate was added back in, and the testing suite was executed to confirm it was still failing. The section of code to perform calculations was then refactored, and the testing suite was executed once more, which resulted in all of the tests succeeding. Opportunities for refactoring were then identified, for moving the values for the basic and higher rate caps into global variables.

After this, the final section of implementation for the income tax calculator was implemented using the full TDD process, creating a personal allowance calculator function for calculating the personal allowance of incomes greater than £100,000. Unit tests were developed and executed prior to and after each implementation, and opportunities for refactoring were identified and carried out. This completed the income tax calculator section of the implementation.

## Iteration 3

The development of the national insurance contributions calculator consisted of three key implementations: the contributions of income below £9,564, between £9,564 and £50,268, and above £50,268. The full TDD process was utilised throughout the iterations, and no major problems were found until the second iteration for this function.

The unit test had been identified and implemented, using a value of “25000” for the input data and a value of “1852” for the expected output. This was generated from an external website to calculate national insurance contributions. The testing suite was executed, and the newly defined test failed, as expected. The functionality could now be implemented. The stub “else” statement from the previous functionality was extended, and a calculation was added for calculating and outputting 12% of the income. Upon running the test suite, it was found that the test still failed, as the output value was much higher than expected. Through analysis of the issue, it was found that the national insurance calculations had been misunderstood, and national insurance did not need to be paid on the entire income, only the income above £9,564. This was then amended, calculating 12% of the given income above the threshold. Upon running the unit tests once more, all of the tests passed. A refactoring opportunity was identified in moving more “magic numbers” into global variables for this new section of implementation.

The final part of this function was then implemented following TDD without any major issues.

## Iteration 4

The final main area of implementation was the student loans contribution calculator function. This had two areas of functionality: no contributions if the income was below £27,288, and 9% of gross earnings above this value if it was above. The development of this area went largely without issues, as the more complex functionality had been developed prior.

After having implemented the first area of functionality, for checking if no student loan needed to be paid, a unit test was defined for the second area. An input value of “50000” was used, with an expected output of “2044”, gained from external calculators. The testing suite was executed, and the new unit test failed. Therefore, the functionality could be implemented. An “else” statement was added to calculate 9% of the threshold subtracted from the income, and the testing suite was executed again. All tests succeeded; therefore, this functionality had been successfully implemented. Refactoring opportunities were identified in the testing suite during this iteration. There was a lot of redundant, repeating code in new definitions for each test case, which decreased readability. Additionally, the list for running all test cases had become very long, and it was difficult to easily change which tests were being run. Therefore, all unit tests were grouped into lists depending on which function they were testing, and the master list was updated to contain these lists. The testing suite was executed once again, and all tests passed successfully.

Finally, following the TDD process, functionality was implemented for calculating the total deductions from the initial income value, and then for calculating the net income based on the deductions. Unit tests were developed and executed prior to the implementation of each area of functionality, and refactoring opportunities were addressed after successful implementation. This completed the development of the net income calculator program.

# Section 3 – Reflection

The use of TDD during the development of this application went well and was a positive overall experience. TDD provided a clear structure for the development of the program, allowing the process of development to proceed smoothly without any major challenges. The code at the end was modularised, readable, relatively high-quality, and focused on the specific required functionality. During the development process, it was difficult to become distracted with other possible functionality, due to the nature of TDD. However, some issues were encountered with using TDD. As this rigorous approach of incrementally developing unit tests, and then the functionality for only that unit test was relatively new to me, I sometimes found difficulty in following the TDD process closely. I sometimes found myself accidentally developing functionality beyond what had been defined in the unit test suite, and therefore I needed to backtrack and regain focus. This may have slowed down the development of the program and is in line with some critique of TDD following the critical evaluation, where in some settings TLD is preferred due to the lower effort required. However, I believe that the benefits outweigh the negatives, and with practice and future use, the negatives of TDD can be minimised or mitigated. I found it much more intuitive than TLD, and I finished with a much higher quality testing suite than in previous projects that utilised TLD.

Functional programming was a new paradigm to become used to during the course of development, and therefore I found it difficult to shift away from the Object-Oriented Programming style of thinking. This led to some impure functions where global variables needed to be used, to eliminate “magic numbers”. Despite this, I believe I was successful in utilising functional programming techniques, such as functions performing calculations without adjusting any global variables, which I was inclined to do at the beginning. As I continued, I became more used to the idea of functional programming and found it easier to follow the rules of the paradigm. I found that this resulted in much simpler, more readable code where it was clear to see what exactly was happening, whereas in languages such as C++ I often find that the code can become convoluted when using global attributes within classes. The idea of a function doing one, specific task, with the output of the task being immediately obvious was a major benefit.

Finally, I found the automated testing framework being used, HUnit, to be very easy to use and useful. The framework allowed for test cases to be implemented with ease. Automated testing allowed for the quick testing of the success of each developed piece of functionality and was an integral part of the TDD process. It allowed me to develop and run unit tests quickly and efficiently, which would have been much more difficult when manually testing each unit. I had no major difficulties with using automated testing as a concept or utilising the HUnit testing framework.

In conclusion, I found the experience of using automated testing, test first development, and functional programming to be a highly enjoyable process. Most difficulties I encountered were due to my lack of practical familiarity with the concepts, and I will continue to use develop my knowledge, understanding, and skills within these domains in future projects.

# References

Beck, K., 2000. *Extreme Programming Explained: Embrace Change.* Reading: Addison-Wesley.

Beck, K., 2003. *Test-Driven Development By Example.* s.l.:Addison-Wesley.

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

George, B. & Williams, L., 2004. A structured experiment of test-driven development. *Information and Software Technology,* 46(5), pp. 337-342.

Madeyski, L., 2010. *Test-Driven Development.* 1 ed. Heidelberg: Springer Berlin.

Moe, M. M., 2019. Comparative Study of Test-Driven Development (TDD), Behavior-Driven Development (BDD) and Acceptance Test–Driven Development (ATDD). *International Journal of Trend in Scientific Research and Development,* 3(4), pp. 1-3.

Muller, M. M. & Padberg, F., 2003. *About the Return on Investment of Test-driven Development.* Portland, Association for Computing Machinery.

Munir, H., Wnuk, K., Petersen, K. & Moayyed, M., 2014. *An experimental evaluation of test driven development vs. test-last development with industry professionals.* London, Association for Computing Machinery.

Romano, S. et al., 2017. Findings from a multi-method study on test-driven development. *Information and Software Technology,* pp. 64-7789.

Tosun, A., Ahmed, M., Turhan, B. & Juritso, N., 2018. *On the effectiveness of unit tests in test-driven development.* Sweden, Association for Computing Machinery.