**Task 1**

**What is test driven development**

Test driven development (TDD) is a process carried out in a software development process in which tests drive the development of software code. This is enacted by via the process of creating a unit test, then developing and refactoring the code it tests so that the test passes before moving onto the next test and code.

Test driven development was made popular thanks to it being a practice in the agile software development methodology, specifically extreme programming. The practice itself has been around for many decades, for example it was used in NASA’s project Mercury in the early 1960s [1]. Many companies which develop software use the agile methodology and as a result test driven development.

TDD is used within various settings outside of software. However, this report will attempt to explain and evaluate it within the context of software development. TDD’s impact on code quality, code confidence, development management, and development time of software development will be looked at.

**Test driven development cycle**

Test driven development makes use of the test-driven development cycle. Each time an additional piece of code is to be developed this cycle is performed, consisting of 5 steps.

1. Add a test

The test is written to cover the overall requirements of the function it will be testing.

1. Run all tests and check if the new test fails

This ensures that no other code has been affected since the last development iteration, so that focus can remain on the development of the new function which will be tested. It also checks if the new test builds correctly and does not always pass.

1. Develop the code to cause the test to pass

This involves creating the code to make the new test pass, and therefore create a working function. At this stage it does not entirely matter if the developed code is not done in the most efficient or professional way as it will be improved later in step 5 of the cycle.

1. Run all tests

This step is there to ensure the newly developed code is working. If the new test still fails, then the programmer must return to step 3. Running all the tests checks that the newly developed code has not in some way affected other code as well, ensuring integrity of the software.

1. Refactor the code

This step involves refactoring the developed code, making it more efficient, maintainable, and clear. Step including things such as removing duplication, using variable naming conventions, or managing memory.

In general, it is thought that the size of the developed functions should be kept small to provide clarity and reduce debugging effort.

**Code Quality**

As a result of creating the tests of functions first before designing them test driven development forces the programmer(s) to think through the requirements of their code from an interface level before writing it. This is good programming practice as it enforces clarity and modularity of code. It also ensures that the programmers are aware of the specification of code before they can start writing it meaning they are correctly informed on its purpose.

Having refactoring as part of the TDD process ensures quality of code. Refactoring improves efficiency, maintainability, and code clarity.

Bhat and Nagappan [2] demonstrated the results of analysis of two software development case studies on projects of similar size, one using TDD and the other non-TDD processes. The results showed that the overall quality of the code for the project using TDD was 2.6-4.2 times that of the non-TDD project.

The analysis described above however does not show in which areas this improvement was seen. Henderson-Sellers, B [4] analysed three case projects of a similar size, one using TDD. Within the analysis it is shown that the project which used TDD resulted in a lack of cohesion. Cohesion within code is necessary for quality as it represents good design. Due to the consistency of the measurements within the report it can be determined that this is not a coincidence for just that specific project. Regardless of whether this is the case, the analysis indicates that TDD does not help to produce cohesive code, and perhaps does the opposite.

**Code Confidence**

In test-driven development, usually, all functions or sections of code are covered by at least one test. Tests are also designed so that they cover all paths of a function. As a result, code coverage of software using TDD is usually high. This results in code confidence since the results of tests are a strong indication as to the state of the code. Results also allow for code defects to be found easily, saving time and effort. Within the Bhat and Nagappan [2] analysis, block coverage of the TDD project was 79-88%, higher than that of the non-TDD developed project.

Depending on the software being developed, TDD may not be as impactful in terms of code coverage as in the above example. For instance, it is difficult to apply unit testing to software in the realm of user interfaces, networking, or database code.

**Development Management**

The unit testing approach of TDD breaks down the code into small individual chunks. These tests outline the functionality of the function that they test. In many ways this is seen as a viable method of documentation of the system. As a result, a good overview of the system is seen for managers and developers wishing to learn the system.

Lui and Chan [3] present the results of TDD on software developed in China. The conference paper showed task estimation and process tracking to have improved because of TDD.

While aiding management, TDD is also very management and resource heavy. The requirement of testing first requires a rigid structure opposed on developers. Setting up running of the tests, if using continuous integration requires resources and development on the continuous integration servers to facilitate this.

**Time**

Since TDD requires more code to be written, such as tests and potentially mock classes for testing, it would make sense that development time would increase. However, because of better program understanding thanks to the development strategy programmers can use existing methods and understand the code base quicker.

Effects of TTD on time are seen in the analysis between two similar sized software projects one using TDD and the other without, Bhat and Nagappan [2]. Project managers of the project using TDD estimated that it improved development time by 15-35%. It cannot however be concluded that time improvements will always be the case. Factors such as unit testing unfamiliarity by developers or the need to setup an infrastructure for the process may affect development time negatively.

**Conclusion**

From this report test-driven development most certainly has its upsides. However, only certain projects in certain contexts can benefit from these upsides. The report has shown for example that if the infrastructure to support this development methodology process is not setup then it becomes a drain on development or if a certain coding method is used then this cannot take advantage of the testing features of TDD.

Test driven development is used so widely across software development because it is an effective method. This is why it is part of the popular Agile development process.

**References**

[1] G. Larman and V.R. Basili, "Iterative and Incremental Development: A Brief History", IEEE Computer 36(6), IEEE Computer Soc., Los Alamitos, CA, USA, 2003, pp. 47-56.

[2] Bhat, T. and Nagappan, N. Evaluating the efficacy of testdriven development: industrial case studies. In ISESE '06, Rio de Janeiro, Brazil, 2006.

[3] Lui, K. M. and Chan, K.C.C. Test Driven Development and Software Process Improvement in China. In XP 2004, Garmisch-Partenkirchen, Germany, 2004.

[4] Henderson-Sellers, B., Object-Oriented Metrics: Measures of Complexity, Prentice Hall, Upper Saddle River, NJ, USA, 1996.

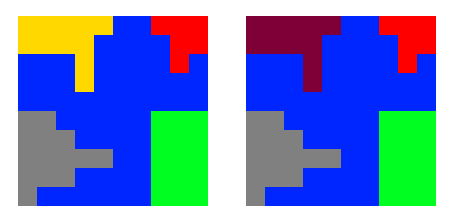
**Bibliography**

<https://arxiv.org/ftp/arxiv/papers/1711/1711.05082.pdf>

<https://www.ripublication.com/ijaer17/ijaerv12n18_81.pdf>

**The Algorithm**

The algorithm developed was a flood fill algorithm done in the functional programming language Haskell. The algorithm resembles the functionality of a paint bucket operation in paint programs. This works by changing a 2d map of colours. A location on the map is picked and the original colour of this location changes to the newly selected colour, this change then propagates in all directions as far as the neighbouring original colour locations are connected to the location chosen. The figure below shows the effect of the algorithm, in which the top left corner location was flood filled with the colour purple resulting in the picture on the right.



**Developing Using Haskell**

Developing a functional algorithm using Haskell for the first time was an experience. I found the structure and style of the language very different to languages I have previously coded in. As a result, I had to overcome quite a large learning curve. Remembering all the functions of the various operators was quite challenging. This was like the creation of the 2D array of chars with position variables. As a result of overcoming these challenges I felt a great sense of accomplishment.

**Test Driven Development**

The algorithm was developed using test driven development. Two separate testing files were created for this purpose, a basic unit testing file and a property unit testing file. Property based unit tests facilitated most of the requirements for testing, however, the function ‘createInputArray’ could not be tested in this manner and therefore the basic unit testing file was created to do this. Essentially, because of the random parameter input aspect of property tests, the result of the ‘createInputArray’ function could not be verified in this manner. Additional tests were however added to this file.

Using test driven development forced me to plan out the system much more thoroughly than I would have if it not using it. This was because I did not want to develop a unit test and function for it if it were to be removed later. As a result, I had a clearer picture of the algorithm and how it would come together.

Test driven development meant that when using functions that had been previously coded for creation of the algorithm, I could be certain that they would work. As a result, finding bugs was much quicker as the scope of the location of the bug was restricted to a specific area. There were however a few times in which I found out I needed to modify the functionality of the previously coded and tested functions when using them in other functions. This meant I needed to change their unit tests as well, slowing development time.

The basic unit tests were created using the HUnit unit testing library. I found this library extremely straightforward to use, and as a result, use of it was one of the easier learning curves to overcome during development.

**Property Based Unit Tests**

Property-based unit tests were created using the quickCheck unit testing library. Learning about the functionality of the library was relatively straightforward and as a result I enjoyed the experience.

The property-based unit testing gave me a sense of confidence in the fact that the functions I had developed worked correctly. This was because of its ability to cover all range of parameter inputs, therefore presumably covering all execution paths. To target all parameter ranges of these functions’ generator classes were created. Two of these were created, one for random 2d colour maps, and one for random positions. Instance overloading of the Arbitrary class was required for this, which was easy thanks to online tutorials.

**Future Use of Test-Driven Development**

Due to unfamiliarity with the language and the testing frameworks development time was not particularly fast. However, now that I am more familiar with the language and tools, use would be much easier. While I have partially enjoyed the experience of using test driven development, I do find it to be rather laborious and not as effective as it ought to be. I could see myself using it but perhaps only within certain contexts, such development of large-scale applications.