Fall 2017

Best Practices for Test Driven Development

Timothy Tacker
Governors State University

BEST PRACTICES FOR TEST-DRIVEN DEVELOPMENT

By

Timothy W. Tacker
B.S., Illinois State University, 2000

THESIS

Submitted in partial fulfillment of the requirements

For the Degree of Master of Science,
With a Major in Computer Science

Governors State University
University Park, IL 60484

2017

Abstract

In his award-winning book, Test-driven Development By Example, Kent Beck wrote, "Clean code that works...is the goal of Test-driven Development (TDD)." TDD is a style of software development that first begins with the creation of tests and then makes use short, iterative development cycles until all test requirements are fulfilled. In order to provide the reader with sufficient background to understand the concepts discussed, this thesis begins by presenting a detailed description of this style of development. TDD is then contrasted with other popular styles, with a focus toward highlighting the many benefits this style offers over the others. This thesis then offers the reader a series of concrete and practical best practices that can be used in conjunction with TDD. It is the hope of the author that these lessons learned will aid those considering the adoption of this style of development avoid a number of pitfalls.

For my grandfather Dale. My education and career have been guided by an interest in technology for which he is directly responsible.

Acknowledgements

This thesis would not exist had it not been for the efforts of my parents to provide me with the resources, opportunity, environment, motivation, and encouragement to pursue higher education. I am extremely grateful for the many sacrifices they made.

I also owe a debt of gratitude to the members of my committee. None were required to volunteer; however, the dedication of their time and energy is very much appreciated.

Contents

Table of Figures

Keywords

Keywords: Test-driven Development, TDD, agile software development, software testing, software requirements

1. Introduction

1.1 Background

This thesis was written and submitted in partial fulfillment of the requirements for the degree of Master of Science, with a Major in Computer Science, at Governors State University, in University Park, Illinois. The author, Timothy W. Tacker, previously earned the degree of Bachelor of Science, with a Major in Applied Computer Science and a minor in Business Administration, at Illinois State University, in Normal, Illinois. The author has worked in the industry for more than twenty-three years, including eight years teaching at the university level and most recently successfully leading a security test organization for mission-critical systems within a Fortune 500 company.

1.2 Topic

The topic of best practices for TDD is explored in this thesis. This includes best practices for both the implementation of TDD and for the continued use of TDD following implementation. This paper restates advice from a variety of expert sources to which the author adds their own context, analysis, and observations.

1.3 Importance

TDD is rapidly being adopted throughout the software development industry. This trend is linked with the adoption of agile software development practices. As agile practices continue to be adopted, the adoption of TDD is also expected to grow in parallel; and therefore, the topic of TDD only becomes more relevant every day.

1.4 Need

This research is necessary because implementation and usage of TDD can be extremely difficult. While a variety of sources on the topic do exist, it will be useful to

collect the lessons already learned in a single location and synthesize them. In addition to aggregation of these ideas, this thesis will help present opposing opinions and mediate between them, exploring ideas such as the ultimate purpose of TDD.

1.5 Objective

The objective of this thesis is to provide a useful reference for those entities that desire to adopt and use TDD. It is the hope of the author that the research collected here will serve as a guide that will assist in avoiding many pitfalls already discovered. It is the aim of the author to also provide information for those entities merely evaluating the use of TDD. After achieving an understanding of the information contained in this paper, the reader will be in a better position to determine some of what is required for the adoption of TDD and make an informed decision regarding its use.

1.6 Structure

This thesis consists primarily of literature review. First, the history of TDD will be explained. Second, a description of TDD will be provided, discussing both the purpose and process of TDD. Third, some of the benefits of TDD will be presented. Fourth, for balance, some of the limitations of TDD will be presented. After this context has been established, best practices for TDD will finally be listed. The paper will end with a summary of findings in the conclusion.

2. Literature Review

2.1 History

The practice of test elaboration prior to the start of programming did not originate with TDD; however, TDD combined this idea with developer testing. (Agile Alliance, n.d.) American software engineer Kent Beck is often credited as the “reviver” of TDD.

Beck is the developer of the lightweight software development methodology known as extreme programming (XP). (Copeland, 2001) Beck is also one of the original seventeen signatories of the Manifesto for Agile Software Development, which popularized the now common practice of Agile Software Development. (Beck, et al., 2001)

Beck reports that he first tried TDD after reading the original description in what he refers to as an “ancient book” about programming. After describing TDD to some older programmers, Beck discovered that they found it obvious and wondered how else one might program. For this reason, Beck indicates his role was merely “rediscovering” TDD, rather than inventing it. (Beck, Kent Beck's Answer to Why does Kent Beck refer to the "rediscovery" of test-driven development? What's the history of test-driven development before Kent Beck's rediscovery?, 2012)

In the 1976 book, Software Reliability, Glenford Myers indicated that developers should not test their own code. Twenty-two years later, in 1988, it was stated in an article on Extreme Programming that the test is usually written first. By 2003, Kent Beck had published his award-winning book, Test Driven Development: By Example. (Agile Alliance, n.d.) In a span of 27 years, best practices had changed from developers never testing their own code to the expectation that developers should always create the tests to validate their work before starting that work. See Figure 1 below for a timeline that highlights these key events in this evolution of practice.

Figure 1. Publication Timeline. These publications mark changes in best practice.

2.2 Description

2.2.1 Purpose. There is some debate about the ultimate purpose of TDD. Nonetheless, there does appear to be a consensus answer to this question. Both the majority opinion and other views will be discussed in this thesis. In general, TDD is an advanced technique that uses unit tests to affect software design. (Palermo, 2006) As an advanced technique, TDD is not currently used for all, or even a majority of, software development projects; however, TDD is rapidly gaining acceptance in the industry. This trend continues to become more apparent with the adoption of various agile software development methodologies that naturally include the use of TDD.

The primary goal of TDD may not be testing software but, rather, assisting the developer and customer arrive at unambiguous requirements in the form of tests. (Vorontsov & Newkirk, 2004) The creation of accurate requirements is one of the most difficult aspects of software development. Using a test to formalize a piece of functionality, subsequent implementation in a fashion that causes the test to pass, and an ongoing repetition of this process is the foundation of TDD. (Erdogmus, 2005) This foundation fosters what is known as a "test first mentality" and represents a primary differentiator between TDD and other styles of software development.

Kent Beck states that the purpose of TDD is clean code that works. (Beck, Test-driven Development: By Example, 2003) While it is true that one view is that TDD is a technique for programming in which writing clean code that works is the primary goal, another view is that specification, rather than validation, is the ultimate goal of TDD. The specification view suggests that TDD assists in thinking through requirements or design before functional code is written. (Ambler, n.d.) In the specification view, the purpose of

TDD is more closely related to initial design quality, rather than quality assurance. In truth, TDD can serve both purposes simultaneously. See Figure 2 below for a Venn diagram that illustrates the idea that there is overlap between these proposed purposes.

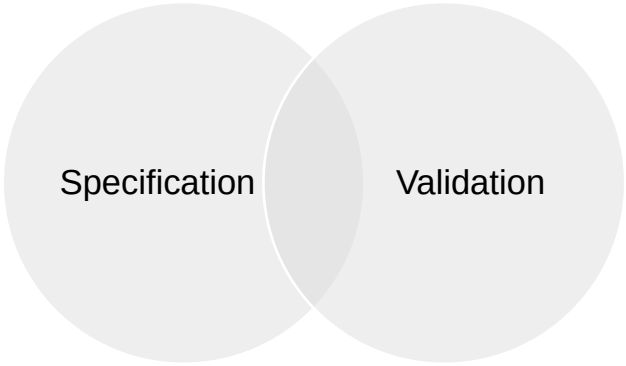

Figure 2. Specification and Validation. These are two views on the purpose of TDD.

2.2.2 Process. There seems to be widespread agreement about the high-level process that constitutes TDD. “Red, Green, Refactor” is the commonly repeated motto of TDD. “Red” involves the creation of a test that will initially fail. “Green” involves the test passing after the implementation work is done. “Refactor” involves code changes to improve the design and remove duplication. Each new unit of code requires a repetition of this cycle. (Palermo, 2006) See Figure 3 below for a flowchart that shows the iterative “Red, Green, Refactor” cycle in a graphical fashion.

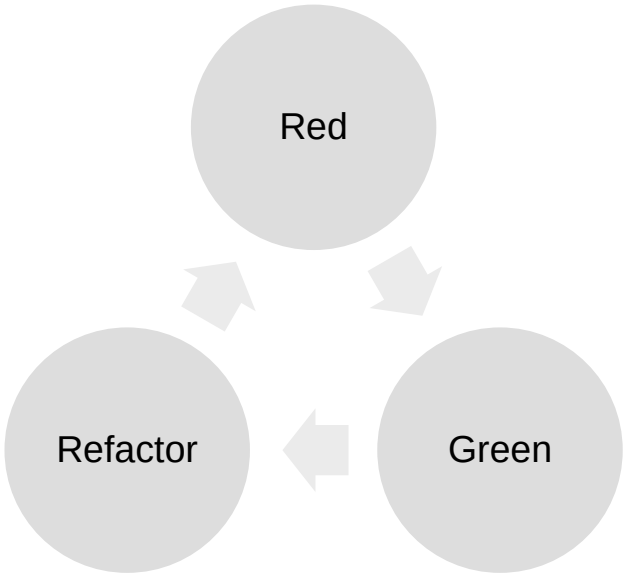

Figure 3. Red, Green, Refactor. This is the motto of TDD.

At a lower level, TDD is a style of programming where the activities of coding, testing, and design are tightly interwoven. First, a single test is written that covers one program aspect. Second, the test should be executed, and fail, because the feature has not yet been implemented. Third, just enough code should be written to make the test pass. Fourth, the code should be refactored for simplicity purposes. Last, the process is repeated; and over time, tests are accumulated. (Agile Alliance, n.d.) These accumulated tests are maintained and continue to be executed during future development to guard against regressions. This model is an alternative to other design approaches, such as the traditional waterfall model, which traditionally progresses through the phases of requirements, design, implementation, verification, and maintenance. See Figure 4 below for a depiction of the prototypical waterfall model that includes the phases mentioned here.

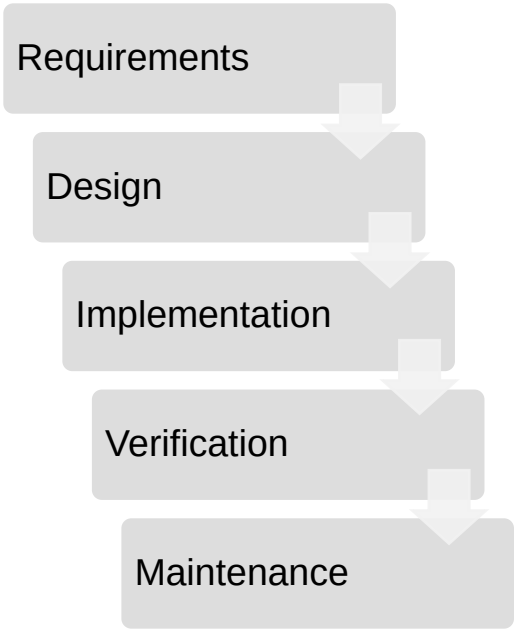

Figure 4. Waterfall Model. A traditional approach to software development.

When implementing any new feature, one should first ask if the current design is the best one that allows implementation of the functionality. If it is not, the design should be refactored to make it as easy as possible to add the new feature; thus, the quality of the design is always improving, and it becomes easier to work with. Traditional development is completely turned around in TDD. (Ambler, n.d.) This philosophy requires less initial design planning and may represent what could be considered "just in time design." The thinking here is that too much initial planning may result in bad assumptions, or decisions that don't accurately predict changing circumstances or requirements into account, and may result in a significant amount of wasted work on the wrong design. With TDD, if the current requirements don't mandate a specific design consideration, then that design simply should not be considered until a later time at which it may be needed.

Test code is not written after functional code in TDD. Test code is written first and only in very small steps. A TDD approach requires programmers to demand the existence of a test that fails before writing even a single line of the function that the test

verifies. (Ambler, n.d.) This does require significant discipline, and it may even be considered stubborn by those who don't fully understand TDD; however, the creation of appropriate tests before writing code is an essential component of the TDD approach. See Figure 5 below for a model that demonstrates the proper sequence of test creation, code creation, and testing in TDD.

Figure 5. Create Test, Write Code, Pass Test. This is the proper sequence in TDD.

2.3. Benefits

2.3.1 Guaranteed requirements compliance. Software is typically created for specific purposes that must be well understood to ensure those purposes are well met. One of the biggest problems in software development is when customer requirements are misunderstood by programmers. Using tests as requirements eliminates the need for human interpretation regarding success, and successful execution of such tests guarantees that the work meets requirements and is done. (Vorontsov & Newkirk, 2004) Translation of initial requirements into later tests is often a difficult and error-prone process. Skipping this process represents the elimination of a large step in traditional software development, which results in significant time savings and avoids an opportunity for errors that invite defects to creep into the process.

With TDD, progress is made even when a test fails because knowledge has been gained that there is a problem that requires resolution. Determining the success versus failure of the testing is also made very clear. This has the effect of increasing confidence

that the solution accurately meets requirements, works, and that it is acceptable to proceed. (Ambler, n.d.)

While the need for acceptance testing is not eliminated, reliance upon it can be significantly reduced. In some cases, customer involvement, or collaboration, in the creation of initial tests may serve as the only acceptance testing needed. Developers can move on to other tests following a test that passes and refactoring, as the code is clearly finished following these events. (Palermo, 2006)

Assuming the correct initial tests are created, there is very little chance that the code will later be rejected and sent back to development, as the tests are the requirements; and if the tests have passed, the requirements have been met. As a result, the need to cleanly maintain multiple branches of atomic code that can be safely rolled back independently, without affecting subsequent development, is eliminated. This results in a lower overall administrative burden and allows for greater productivity. See Figure 6 below for an example of how TDD allows multiple, independent feature branches of code to be merged into a single branch of code that has been verified to work properly.

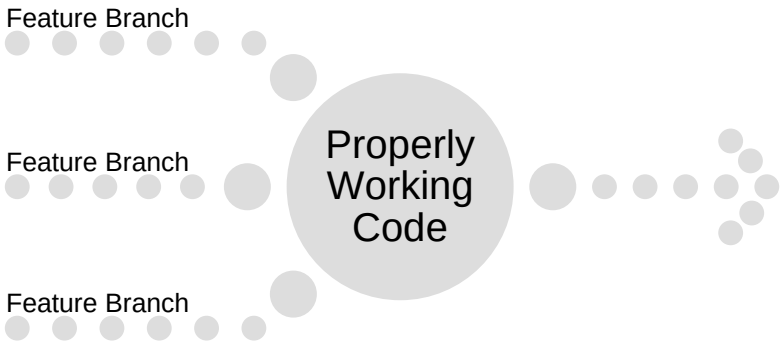

Figure 6. Feature Branches. TDD reduces the need to maintain feature branches.

2.3.2 Better facilitation of testing. Outside of TDD, while most code is designed for functionality and performance, it not designed specifically for testability. It is easier for quality assurance to test code developed using TDD, as the code is written with testing in mind and the process generates a base of pre-existing tests. This allows quality assurance to become more proactive, rather than reactive. (Vorontsov & Newkirk, 2004) Being already provided with essentially all functional testing that is needed, quality assurance is freed to focus on more exploratory and usability testing. The net result is not only more testing but also creates an opportunity for smarter, risk based testing above and beyond the baseline functional testing.

Tests developed for use with TDD can be maintained, updated, and reused beyond initial development to check for breakage of existing functionality as additional changes are made to the software, giving a greater level of confidence that no regressions have been introduced. (Vorontsov & Newkirk, 2004) In other words, in addition to serving as the initial functional testing, the same base of test cases is reused for regression testing each time additional development is done in the future. Constant feedback that each component is still working is provided by the suite of unit tests. (Palermo, 2006) This inspires confidence not only in programmers to make bold changes that are often necessary but also the confidence of quality professionals in the work of those programmers.

Traditional testing recommends, but does not guarantee, that every single line of code is tested; however, 100% testing coverage achievement is a side effect of TDD. (Ambler, n.d.) The reason for this is that in TDD it is not acceptable to add code without first creating a test for any code that is created. In any organization that uses code

coverage as a quality metric, TDD will consistently allow the achievement or perfection in this measure; but this does not guarantee that the code will be free of bugs; and therefore, the importance of other quality metrics should be emphasized with the adoption of TDD. See the limitation section of this thesis for additional details.

2.3.3 Better overall design. Testing usually addresses requirements compliance, functionality, usability, and performance; but it rarely reaches evaluation of underlying design. Reduction of the scope of tasks to be performed, improved understanding of requirements, and better decomposition are encouraged by creating tests prior to implementation and proceeding with development one test at a time. (Erdogmus, 2005) Improved code design qualities and better technical quality in the form of cohesion and coupling metrics are reported by veteran practitioners of TDD. (Agile Alliance, n.d.) If true, then TDD not only allows for more testing and better testing but also results in the side effect of encouraging better underlying design. Additional research is needed regarding this quality of TDD.

A true understanding of the desired result and how to test it is required in TDD before a developer can create production code, so this forces critical analysis and design. (Palermo, 2006) Developers may not quickly skim over complex requirements without full understanding, as they will then be unable to create the required test cases if they do so.

While external documentation is obviously important, developers are often also encouraged to document their code with comments and even to write "self-documenting" code. External documentation frequently goes out of date, but the same is not true for unit tests, which simultaneously act as documentation in TDD. (Palermo, 2006) These unit

tests require maintenance and are not permitted to go out of date. If they were to go out of date, they would no longer product the correct results and would start to report failures during test execution, signaling that either a regression has been introduced or that the test case needs to be updated.

2.3.4 Reduced need for debugging. Traditional development practices allow programmers to create expansive, non-atomic additions to code, which aren't simple or narrowly focused on individual requirements. This would potentially be acceptable if no mistakes were ever made; however, this expectation is obviously unrealistic. When mistakes are eventually made, significant time must be spent debugging and locating the problem area.

TDD forces developers to work in small steps. Instead of waiting until many changes have been made to the code, which may result in uncertainty as to which change caused the problem, smaller steps make it easier to determine when mistakes are made during development. This reduces the need to rely on a debugger, which increases the speed of development. (Vorontsov & Newkirk, 2004) Debugging and rework effort is reduced because a small testing scope permits quicker turnaround and more frequent regression testing. (Erdogmus, 2005) Developers can be trained to make only small, focused, atomic commits to code; but TDD forces the issue and requires what is already widely regarded as a best practice. It is not possible to practice TDD and ignore this way of working. Code additions always map to specific tests.

TDD offers confidence not only that new code additions are correct but also that old code is not broken by new additions. Regressions also typically take a significant amount of time to debug; and in many cases, such issues are even more complex to

resolve. Each time the tests are executed, it is verified that all previous bugs are fixed and remain fixed. This provides protection against regressions and therefore also a reduction in time needed to debug code. (Palermo, 2006) The overall time saved debugging is well worth the discipline.

2.3.5 Reduced rate of defects. While TDD involves an increase in initial development effort required, significant reductions in defect rates are reported by many teams using TDD. During a project's final phase, these same teams report a reduction in effort and indicate the initial overheads as more than offset. (Agile Alliance, n.d.) This is likely a result of the additional testing enabled by TDD and the focused quality of that testing. Verification is also pushed from the end of the project to several times throughout the project, where the work done is fresh in mind and where is takes less time to diagnose issues. In other words, the same quality can be achieved with less effort using TDD, which also means better quality can be achieved with the same amount of effort.

2.3.6 Improved productivity. In addition to all the other benefits of TDD, the practice also results in improved developer productivity. The test-first methodology results in a larger number of programmer tests and improved productivity. (Erdogmus, 2005) This argument is made even more convincing when one considers the creation of test cases part of production and the increase in output ratio of properly working versus defective code.

The smaller steps that are taken in TDD are far more productive than coding in larger steps. It is a less complex task to find and fix defects when less code has been written. It becomes more attractive to take such smaller steps when the compiler and regression test suite can be executed quickly. (Ambler, n.d.) Developers spend less time

resisting this practice, and more time practicing it, when it is thus made so attractive to them.

Developers are free to refactor and make design changes at any time, because test execution provides confidence that the software is still working. This tends to result in a better, loosely coupled design that is easy to maintain. (Palermo, 2006) Because less time and effort is spent on the administrative overhead of required maintenance, more of each can be spent on adding functionality and other core development.

2.4. Limitations

2.4.1 Increased initial effort. Development in TDD can feel slow at the start, because more time and effort are required at the beginning with TDD than in other styles of software development. (Hill, 2015) Developers are required to refrain from jumping straight in to do development; and instead, they are forced to spend time carefully thinking through and defining the work, considering how it will be tested, and writing the test cases that ultimately fulfill these purposes. Objectively, this means that developers using TDD spend more time and energy planning and start work on functional code later.

```
int addfive(int num)
{
   return(num + 5);
}

void test_addfive(void)
{
   CU_ASSERT(addfive(10) == 15);
}
```

Figure 7. Test Case Size. Example size of test case.

To illustrate increased initial effort, shown in Figure 7 above are two example functions written in the C programming language. The first function, addfive, represents

the functional code to be tested. For the purpose of presenting a simple example, the addfive function does nothing more than add five to any integer provided and returns the result. The addfive function required only four lines of code to implement, including the function header and braces on their own lines. The test_addfive function is the code required to test the addfive function. It is written using the CUnit testing framework and checks to ensure that when provided with the integer 10, the addfive function correctly returns 15. The test_addfive function required an additional four lines of code to implement, again including the function header and braces on their own lines. The number of lines of test code that needed to be written in this case is exactly equal to the functional code itself. Because test code is written before functional code in TDD, after four lines of code the TDD developer will have completed only the test case, where the traditional programmer will have completed the entire functional code itself.

2.4.2 Quality not guaranteed. It is a mistake to assume that adoption of TDD will always automatically result in better quality. While TDD may result in more consistent quality, it is not guaranteed to consistently achieve better quality. TDD reduces the influence of developer skill on quality. Variation is reduced, and minimum achievable quality is improved by the execution of a larger number of tests; however, this quality is not unique to TDD. (Erdogmus, 2005) Other styles of software development can also benefit from an increase in the number of tests executed. In other words, enhanced quality is only a potential side effect of TDD, which results from the larger number of test cases that are typically executed in a TDD environment.

2.4.3 Traditional testing not replaced. TDD is primarily focused on unit testing. While TDD provides a method to ensure effective unit testing, it does not replace

traditional testing. (Ambler, n.d.) Following adoption of TDD, the need for other types of testing will remain. For example, if the customer was not heavily involved in the creation of initial test cases, and even in some cases if they were, the customer may still require extensive acceptance testing. Integration and system-level testing beyond what TDD provides is also necessary in most cases.

```
int addten(int num)
{
   return(num+10);
}

void test_addten(void)
{
   CU_ASSERT(addten(20) == 30);
}

void calculate(int option)
{
   int option1 = 50;
   float option2 = 50.5;

   if(option == 1) /* Integer */
   {
      printf("Answer: %i", addten(option1));
   }
   else if(option == 2) /* Float */
   {
      printf("Answer: %f", addten(option2));
   }
}
```

Figure 8. Integration Issue. An integration issue TDD won't catch.

In Figure 8 above, three C language functions are presented. The first, addten, represents a hypothetical new implementation of a function that previously existed in a system. This function expects an integer number, adds ten to it, and returns the result as an integer. In accordance with TDD practices, before this function was created, a function to test the addten function was created. The test function is named test_addten and uses

the CUnit testing framework to check that when the integer twenty is provided to the addten function, the integer thirty is correctly returned. Unfortunately, when the addten function was reimplemented, it was not realized that, in some cases, the existing calculate function may call the addten function with a floating-point number, instead of an integer as expected. The unit test provided by the test_addten function will indicate that the addten function passes; however, due to the narrow focus on only the code being implemented in the new addten function, it was missed that using second option of the existing calculate function will now produce undesirable behavior. The calculate function could easily have been implemented as a remote procedure located in a different part of the system that was a black block to the TDD developer. TDD will not catch these types of bugs in distributed systems; and for this reason alone, integration and system-level testing are still needed.

2.4.4 Not appropriate in all circumstances. Mechanically determining that the goals of software have been met is not always possible with TDD. Some programming tasks cannot be driven only by tests. Concurrency and security issues are two examples of this limitation. Reliable duplication of subtle concurrency issues can't be guaranteed by running code. Also, while TDD may be able to discover defects in software, it can't make the human judgements that are needed about the methods that were used to secure that software. (Beck, Test-driven Development: By Example, 2003)

```
int percent = 90;

int addpercent(int increase)
{
   if((percent+increase) <= 100)
   {
      percent = percent + increase;
      return(increase);
   }
   else
   {
      return(0); /* Refuse */
   }
}

void test_addpercent(void)
{
   CU_ASSERT(addpercent(10) == 10);
   CU_ASSERT(addpercent(20) == 0);
}
```

Figure 9. Potential Concurrency Issue. Function not thread safe.

Presented in Figure 9 above is an addpercent function written in C. The purpose of this function is to increase a global percent variable by a requested amount, and then return the amount increased if successful; however, the function should also refuse any increase that would make percent greater than one-hundred. The CUnit testing framework has been used to create a test_addpercent function with two tests. The first test ensures that ten percent can be successfully added to the initial ninety percent and that the ten percent increase will be reported back as returned by the addpercent function. The second test ensures that if a twenty percent increase is requested beyond the initial ninety percent, the addpercent function correctly refuses the increase and reports back a zero increase. The addpercent function passes the unit testing demanded in accordance with TDD; however, if the function is used in a multithreaded application, it's possible that it

will be called simultaneously by two different threads, allowing for a scenario that the testing did not cover and inadvertently allowing percent to be increased beyond one-hundred. Because we can not determine the order in which multiple threads execute, TDD does not guarantee that a valid result will always be produced if the addpercent function is used in a multithreaded application.

```
int checkpin(int pin)
{
   if(pin == 1234)
   {
      return(1); /* Success */
   }
   else
   {
      return(0); /* Failure */
   }
}

void test_checkpin(void)
{
   CU_ASSERT(checkpin(1234) == 1);
   CU_ASSERT(checkpin(9876) == 0);
}
```

Figure 10. Bad Security. Functional but insecure.

Shown in Figure 10 above is a simple C language function called checkpin. The purpose of this function is to check if the PIN provided equals 1234. If yes, the function should return the integer one; and if not, the function should return the integer zero. Using the CUnit testing framework, two simple tests have been implemented in the test_checkpin function to verify that this function works as designed; and according to these TDD tests, the function passes. Nonetheless, the checkpin function demonstrates highly questionable security practices. For example, it is not a good secure coding practice to hard code the PIN unencrypted and directly in the source code. One can also raise several other security concerns with this code, such as if the code requires the PIN

to be long or complex enough. The unit testing demanded by TDD in this case merely verified that the function worked as designed—not that the design decisions made will ultimately result in a secure system.

In addition, because TDD emphasizes unit testing, instead of integration and system-level testing, it can be a significant challenge to use TDD with large systems, developed by geographically distributed teams. It is not impossible to use TDD in these circumstances; however, it is critical that developers account for the failure of TDD to rigorously address the communication issues involved. (Sangwan & LaPlante, 2006)

2.4.5 More refactoring required. Major refactoring is often required with TDD. This is because, instead of thinking ahead, TDD developers are expected to focus only on implementation of the simplest design that meets current requirements—not future needs. (Hill, 2015) In most other styles of development, consideration of future needs often goes into planning and development, which typically reduces the need to refactor during later development. “You Aren’t Gonna Need It” is a principle in both extreme programming and TDD; and while this may sometimes be true, it is also true that “You Will Indeed Sometimes Need It.” Adoption is TDD is a conscious decision to take this risk and acknowledgement that more factoring will be required in some cases.

```
float total(float price, float tip)
{
   float taxrate = 0.08;
   return(price + (price * taxrate) + tip);
}

float total(float price, float tip, int exempt)
{
   float taxrate = 0.08;
   if(exempt == 0) /* Not Tax Exempt */
   {
      return(price + (price * taxrate) + tip);
   }
   else if(exempt == 1) /* Tax Exempt */
   {
      return(price + tip);
   }
}
```

Figure 11. Refactoring. Redesign to accommodate tax exemption.

Two versions of a hypothetical total function, written in C, are shown in Figure 11 above. For a simple example, floating-point numbers have been used for currency, even though this is not a recommended best practice for actual production code. Also, the tests that would have been created are not shown, as they are not relevant to the point illustrated here. They are, however, shown in a later example to illustrate a different point. The purpose of the total function is to add tax and tip to a price to calculate a total. The first version of the total function was created using a TDD process. The developer was aware that there could be potential cases where tax should not be added due to a tax exemption; however, it was not an immediate need when the first version was created, so this requirement was ignored. As a result, the total function eventually needed to be refactored into the second version to accommodate instances of tax exemption. There is a significant difference between the two versions with the function essentially needing a complete rewrite. In accordance with the TDD philosophy, the developer also chose to

ignore other possibilities, such as different tax rates for different types of customers; and this may very well result in a need to refactor the total function into a third version in the future. If these features had been incorporated before they were needed, these additional rounds of refactoring would not have been necessary.

2.4.6 Extensive test maintenance. Constant reconfiguration of the test suite is required to achieve maximum value in TDD. This translates to an ongoing investment of time and energy dedicated to test suite maintenance. This burden is increased as changes in design become more frequent, as they often do in TDD. (Hill, 2015) In other styles of development, this effort can be redirected toward the creation of functional code. This can be a challenge for developers that feel their primary responsibility is the creation of functional code, rather than the creation of test code, and is one of the reasons that implementation of TDD requires changes in organizational attitudes and culture.

```
void test_total(void)
{
   CU_ASSERT(total(10.00, 2.00) == 12.80);
}

void test_total(void)
{
   CU_ASSERT(total(10.00, 2.00, 0) = 12.80);
   CU_ASSERT(total(10.00, 2.00, 1) = 12.00);
}
```

Figure 12. Test Case Maintenance. Needed after refactoring.

Figure 12 above illustrates two versions of the test_total function, intended to test the two versions of the total function referenced earlier in this thesis. Both are written in C and make use of the CUnit testing framework. The first version of the test_total function did not pass a third argument to the total function, as the original version of that function did not include a third parameter to indicate tax exemption. When the total

function was refactored to include this third argument, it was also necessary to maintain the associated test function. As can be seen in this example, this did not only require the addition of a third argument but, rather, also an entirely separate, second test within the test_total function. Where there was only one line of code in the test_total function previously, now there are two. This representing a doubling of the size of the test function. Granted, it is a small addition here; however, that is because the example is simple. Given a more complex system, the test case modifications will be similarly complex, resulting in an ongoing need for extensive text case maintenance.

One can imagine that the developer may have also chosen to include additional tests, beyond those shown in this example, for added confidence. For example, perhaps in addition to the two tests in the second version of the test_total function, the developer chose to also include tests for a variety of prices, tip amounts, or even no tip. As tests accumulate over time; and the test suite grows in size, there may arise a need to come back to this function and remove some of the additional tests, to help speed execution time of the test suite. This is another type of ongoing test maintenance that will be required even when not triggered by tested functions being refactored.

2.5. Best Practices

2.5.1 Avoid partial adoption. In some environments, there is an attempt for one part of a team to start using TDD while other parts of the same team are still using other methodologies. This is understandable, as it is often difficult to implement such a sweeping change in culture and process wholesale. Nonetheless, partial adoption should always be avoided. Everyone on the same team should use TDD. (Agile Alliance, n.d.)

Adoption of TDD is an all or none proposition, and significant effort may be required to make such a transition.

2.5.2 Create tests before writing code. In TDD, code should never be written before an automated test case has been created. This test should initially fail, as the code that it is intended to test should not yet exist. If there are no such test in existence, there is no requirement, and nothing need be implemented. Following this rule avoids the creation of unnecessary or untested code. (Vorontsov & Newkirk, 2004) Failing to follow this rule may result in the creation of code for which no test case is ever written. This will undermine the complete code coverage during testing that TDD offers. Worse, without such a test, there will be no advance agreement regarding when the work has been completed or is correctly implemented.

```
void test_addtwenty(void)
{
   CU_ASSERT(addtwenty(40) == 60);
}
```

Figure 13. Test Code First. Written before functional code.

Figure 13 above shows a function, called test_addtwenty, which is written in C and implemented using the CUnit testing framework. As can be seen from the code, this function is intended to test a hypothetical function named addtwenty and ensure that when passed the integer forty it correctly returns sixty. It should be noted that the addtwenty function is not shown because it has not yet been created. If this test_addtwenty test case is executed now, it should fail due to the absence of any function named addtwenty. This is the first step in TDD—create a test case that fails. The addtwenty function can be considered correctly and completely implemented once this test_addtwenty function eventually indicates a pass. There are various techniques that can

be used to make compliance with this rule more likely. For example, pair programming can assist in the avoidance of such slips. (Ambler, n.d.) It is more difficult to break the rule when this type of external accountability is introduced.

2.5.3 Create a test list. Brainstorming is the starting part when introducing new functionality with TDD. The developer should write down a list of tests when starting a new feature or task. The scope of activity is defined by this test list; and by describing the requirements unambiguously, the test list serves as the best criteria for determining completion. (Vorontsov & Newkirk, 2004) The first draft of the test list need not be perfect, as it is only intended as an aid in helping the developer to think through which unit tests need to be created and in what order.

Don't write too many tests at once. (Agile Alliance, n.d.) Additional tests may be added to the test list as the developer realizes they are needed. At this point in the process, it is perfectly acceptable for the list to remain a simple list of potential tests, without specific details about those tests. See Figure 14 below for an example of a username validation test list, showing simple candidate tests that might ultimately be used to validate such a hypothetical feature.

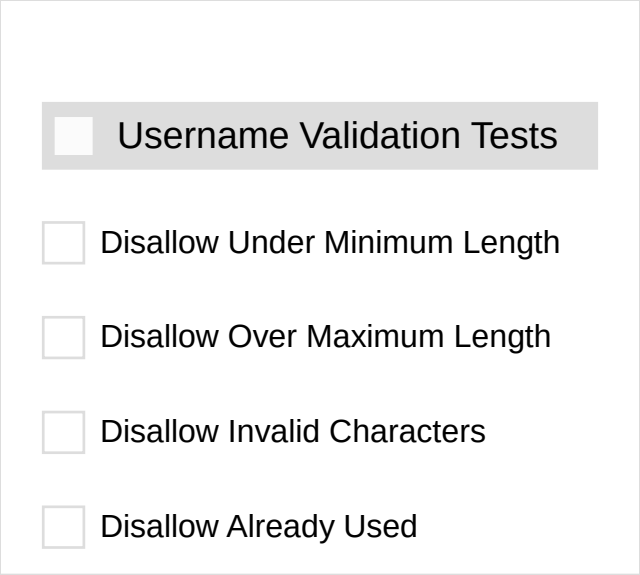

Figure 14. Username Validation Test List. This is an example of a test list.

The next step is to determine which tests from the list to implement and in what order. One strategy is to implement tests from the list in the order that they provide useful feedback from the problem being solved. (Vorontsov & Newkirk, 2004) Those tests that provide more useful feedback should be implemented first; and those that provide less useful feedback should be implemented later, or not at all. If a test does not provide useful feedback, then it is not a good choice anyway and should be eliminated from the test list. This should not be considered a failure to be avoided. It is better to error on the side of caution and include all test cases that may potentially be needed during the brainstorming phase.

Another strategy is to implement tests from the list in order of simplicity. For example, with username validation, it may be simple to determine if a username is less than the minimum length; but it may be more difficult to determine if invalid characters have been included. In this scenario, with a strategy of implementing the simplest tests

first, checking for minimum length should therefore be done before checking for invalid characters in the username.

2.5.4 Automate testing. Software testing can be either manual or automated. In manual testing, testers must interactively execute test cases and check for expected results. In automated testing, test code is created that is intended to exercise the functional code being tested and automatically report a pass or failure. In TDD, tests must be automated. (Vorontsov & Newkirk, 2004) Test automation ensures consistent, reliable results and speeds the process of test execution, which is vital to ensure that the tests continue to be run with each additional feature developed on an ongoing basis. Throughout this thesis, examples of automated tests have been provided, implemented using the CUnit testing framework. A variety of other automated testing frameworks are also available; however, a comprehensive list is too large to include here.

2.**5.5 Design tests well.** Tests should not be too large or general. (Agile Alliance, n.d.) Unit tests should test only one single thing; so, if there are any problems, it's obvious where to look. This means unit tests should be very limited in scope. (Palermo, 2006) A test that is too large may be an indicator of scope creep. Large tests are also difficult to maintain and update. If multiple different aspects require testing, create multiple, dedicated test cases focused on each of those aspects instead. The individual test cases can be executed together as part of a test suite; however, each will be easier to maintain and give less ambiguous results regarding the function being tested. (Agile Alliance, n.d.) In Figure 15 below, the test_validate function includes tests for multiple aspects of the validate function. As compared to the second and third example, instead of focusing on a single test, this function attempts to encapsulate multiple tests. Should one

of the included tests fail, it may be unclear as to exactly which test has failed. Objectively, as can be seen in the example, this also results in a larger function which will be more difficult to maintain; and the complexity almost certainly be worse in the actual production code of a complex system.

```
void test_validate(void)
{
   /* Disallow Under Minimum Length */
   CU_ASSERT(validate("A") == 0);

   /* Disallow Over Maximum Length */
   CU_ASSERT(validate("ABCDEFGHIJKL") == 0);
}

void test_validate_minlength(void)
{
   CU_ASSERT(validate("A") == 0);
}

void test_validate_maxlength(void)
{
   CU_ASSERT(validate("ABCDEFGHIJKL") == 0);
}
```

Figure 15. Test Case Scope. Lesser scope is better.

Units tests should clearly reveal their intention. In other words, it should be easy for another developer to arrive at an understanding of what is expected in production code simply by looking at the unit test. (Palermo, 2006) This is essentially the idea that code should always be self-documenting, and it applies not only to functional code but also automated test cases. Because test cases also serve as requirements in TDD, self-documenting code may be more important in test cases than in functional code. Tests should not be viewed as temporary, ad hoc frameworks that are to be quickly hacked together and discarded. They are themselves solutions that require careful design consideration and ongoing maintenance. In Figure 15 above, note that the first example,

with greater scope requires comments, while the second and third examples are self-documenting and do not. The function of these tests can be derived not only from the function names but is also easier to determine as there is less code within each.

2.5.6 Keep code simple. TDD practitioners should strive to write code satisfies the requirements but no less and no more. Failing to write enough code will result in requirements not being met, but too much code adds complexity and creates a maintenance burden. (Vorontsov & Newkirk, 2004) Resulting code should be as simple as possible. Among other factors, this means the creation of the smallest number of classes and methods needed to pass testing. Achieving such simplicity can be difficult, but it results in resilient code that is easy to modify. (Vorontsov & Newkirk, 2004) An addition of complexity to remove duplicate code from a solution is acceptable because, rather than merely anticipating a possible future need, it addresses the actual, current need to remove duplicate code. This results in complexity only where complexity is truly needed in the solution. (Vorontsov & Newkirk, 2004)

```
int addfifty(int num)
{
   return(num + 50);
}

int timestwo(int num)
{
   return(num * 2);
}

int calculate(int num)
{
   int x = 0;

   x = addfifty(num);
   x = timestwo(x);
   return(x);
}

int calculate(int num)
{
   return((num + 50) * 2);
}
```

Figure 16. Code Simplicity. Keep functions minimized.

In Figure 16 above, two examples of a hypothetical calculate function are presented, written in the C language. Both functions perform exactly the same calculation—take the integer provided, add fifty to it, multiply the result by two, and return the answer. The second example requires a grand total of four lines (Only one line is required if the function header and braces are not counted.) In contrast, the first example of the calculate function requires eight lines and two additional functions, each four lines in length themselves. Both implementations are correct and produce the same results; however, the first is obviously more complex. Developers may be tempted to write code in a more complex fashion in anticipation of future needs or because they have been taught to modularize code whenever possible. Unfortunately, this is the wrong

approach in TDD. TDD encourages code similar to the second example. If the desired result can be achieved more simply, that is what should be done in TDD. If additional complexity, in the form of modularization, is needed for some reason in the future, then then it can, and should, instead be implemented at that future time.

2.5.7 Consider test execution time. Don't forget to run tests frequently. (Agile Alliance, n.d.) Unit tests that take too long to run will not be run often, so ensure they run fast. (Palermo, 2006) To increase the speed of test suite execution, test suites should be separated into multiple components. There should be one suite that includes only the tests for the new functionality being developed and another that contains all tests. The first should be executed more often than the second, and the second in the background or outside of normal working hours. The addition of hardware resources should not be overlooked in the effort to speed execution. (Ambler, n.d.) Exercise of dependencies such as networks, file systems, and databases will cause unit tests to execute slowly; so, it is better to separate of simulate these during testing. (Palermo, 2006)

Test suits should be maintained so that they do not eventually require running times that are too long. (Agile Alliance, n.d.) As tests begin to accumulate, test suites will take longer and longer to execute, and this may serve as a disincentive for those test suites to be executed each time development is competed. Team turnover, or poor maintenance, should not be permitted to result in abandoned test suites that are seldom or never executed. (Agile Alliance, n.d.)

3. Conclusions

This thesis explored the topic of best practices for TDD. This research was necessary due to the difficulty involved in the implementation and use of TDD. This

paper aggregated advice from a variety of expert sources into a single location and provided additional context and insight for that advice.

Providing a useful reference for those wishing to adopt and use TDD was the primary purpose of this research. The creation of a guide to assist in sharing lessons already learned, and the avoidance of pitfalls already known, was the goal of the author. It is hoped that the information presented here will help better, more informed decisions to be made regarding the adoption and use of TDD.

A structured approach was taken in this thesis. First, the author provided background and context for TDD. Next, TDD was described in terms of purpose and detailed process. Following that, both benefits and limitations of TDD were explored. Last, a list of best practices for TDD was presented to the reader. At each stage, the author added his own voice to that of the cited experts.

The benefits of TDD listed in this thesis are as follows:

- Guaranteed requirements compliance.
- Better facilitation of testing.
- Better overall design.
- Reduced need for debugging.
- Reduced rate of defects.
- Improved productivity.

The limitations of TDD highlighted in this thesis are as follows:

- Increased initial effort.
- Quality not guaranteed.
- Traditional testing not replaced.

- Not appropriate in all circumstances.
- More refactoring required.
- Extensive test case maintence.

The best practices for TDD offered in this thesis are as follows:

- Avoid partial adoption.
- Create tests before writing code.
- Create a test list.
- Automate testing.
- Design tests well.
- Keep code simple.
- Consider test execution time.

Readers interested in further study on the topic of TDD should be aware that a variety of sources exist. In addition to several books on the subject, there are also a great deal of websites with extremely useful information. With the expanding popularity of TDD, the author expects the number of such sources to continue to grow.

Researchers seeking related topics to explore will find a wealth of interesting subjects surrounding TDD. More information is needed regarding the impact of TDD on quality. Studies should be completed on the effect that TDD has on developer confidence and if that alone translates into enhanced programmer productivity. An analysis of success rate in TDD adoption would add significantly to this research.

References

Agile Alliance. (n.d.). *What is Test Driven Development (TDD)?* Retrieved from Agile Alliance: https://www.agilealliance.org/glossary/tdd/

Ambler, S. W. (n.d.). *Introduction to Test Driven Development (TDD)*. Retrieved from Agile Data Home Page: http://agiledata.org/essays/tdd.html

Beck, K. (2003). *Test-driven Development: By Example.* Boston: Addison-Wesley.

Beck, K. (2012, May 11). *Kent Beck's Answer to Why does Kent Beck refer to the "rediscovery" of test-driven development? What's the history of test-driven development before Kent Beck's rediscovery?* Retrieved from Quora: https://www.quora.com/Why-does-Kent-Beck-refer-to-the-rediscovery-of-test-driven-development-Whats-the-history-of-test-driven-development-before-Kent-Becks-rediscovery

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., . . . Thomas, D. (2001). *Manifesto for Agile Software Development*. Retrieved from Manifesto for Agile Software Development: http://agilemanifesto.org/

Copeland, L. (2001, December 3). *Extreme Programming*. Retrieved from Computerworld: https://www.computerworld.com/article/2585634/app-development/extreme-programming.html

Erdogmus, H. (2005). On the Effectiveness of Test-first Approach to Programming. *Proceedings of the IEEE Transactions on Software Engineering, 31(1).* National Research Council Canada.

Hill, S. (2015, February 23). *Lean Testing*. Retrieved from The Pros and Cons of Test-Driven Development: https://leantesting.com/test-driven-development/

Palermo, J. (2006, May). *Guidelines for Test-Driven Development*. Retrieved from Microsoft Developer Network: https://msdn.microsoft.com/en-us/library/aa730844(v=vs.80).aspx

Sangwan, R. S., & LaPlante, P. A. (2006). Test-Driven Development in Large Projects. *IT Professional Magazine*, 25-29.

Vorontsov, A., & Newkirk, J. W. (2004). *Test-Driven Development in Microsoft .NET.* Microsoft Press.