Evaluating the Efficacy of Test-Driven Development: Industrial Case Studies

Thirumalesh Bhat
Center for Software Excellence
One Microsoft Way
Redmond, WA 98052
thirub@microsoft.com

Nachiappan Nagappan
Microsoft Research
One Microsoft Way
Redmond, WA 98052
nachin@microsoft.com

ABSTRACT

This paper discusses software development using the Test Driven Development (TDD) methodology in two different environments (Windows and MSN divisions) at Microsoft. In both these case studies we measure the various context, product and outcome measures to compare and evaluate the efficacy of TDD. We observed a significant increase in quality of the code (greater than two times) for projects developed using TDD compared to similar projects developed in the same organization in a non-TDD fashion. The projects also took at least 15% extra upfront time for writing the tests. Additionally, the unit tests have served as auto documentation for the code when libraries/APIs had to be used as well as for code maintenance.

Categories and Subject Descriptors

D.2.8 **[Software Engineering]:** Metrics - *Performance measures*, *Process metrics*, *Product metrics.*

General Terms: Measurement

Keywords: Test-Driven Development, software quality.

1. INTRODUCTION

The Extreme Programming (XP) software development methodology [3] is a test-centric approach to software development. With XP, developers follow the test-driven development (TDD) [4] practice, incrementally writing unit and acceptance test cases throughout the software development cycle. The writing of these test cases is based upon the implementation of user stories. These test cases are generally written using one of the xUnit (e.g. NUnit for C#, JUnit for Java) automated testing tools.

In this paper we investigate the utility of the TDD process as a stand alone component for software development and do not investigate or assess the efficacy of XP. In industry, TDD has often been discussed as a technique that has lead to producing higher quality code. Unfortunately there has been limited industrial evidence of this increase in quality (if any). Furthermore, there has been speculation over the increase in time required to write the unit-tests. Overall this leads to developers and managers being divided about

the utility of TDD.

In Microsoft several teams have employed TDD successfully in their development practices. We investigate two teams from two different business units in Microsoft (Windows and MSN divisions) and present their various context factors, product factors and result outcomes. The main objective of this paper is to compare the differences in quality and overall development time between projects using TDD and non-TDD methodologies.

This paper is organized as follows. Section 2 discusses the TDD process and Section 3 presents the related work. Section 4 presents our case studies and Section 5 the threats to validity. Section 6 discusses the conclusions and future work.

2. TEST-DRIVEN DEVELOPMENT

TDD is an "opportunistic" [6] software development practice that has been used sporadically for decades [8]; an early reference of its use is the NASA Project Mercury in the early 1960's [12]. The practice has gained added visibility recently as one of the practices of XP. As shown in Figure 1, TDD software engineers develop production code through rapid iterations of the following:

- Writing a (very) small number of automated unit test case(s);
- Running the new unit test case(s) to ensure they fail (since there is no code to run yet);
- Implementing code which should allow the new unit test cases to pass;
- Re-running the new unit test cases to ensure they now pass with the new code;
- Refactoring the implementation or test code, as necessary, and
- Periodically (preferably once a day or more) re-running all the test cases in the code base to ensure the new code does not break any previously-running test cases.

In these iterations, test cases will almost always fail initially (because test code is written prior to implementation code) but all test cases will ultimately pass. Since all of the test cases must successfully pass before new code is added to the code base, there can be some level of confidence that the new code did not introduce a fault or mask a fault in the current code base. Most often, TDD developers automate their test cases using a framework of the xUnit family.

Some possible benefits of TDD discussed previously [16] [7] are as follows:

- **Efficiency and Feedback:** The fine granularity of the test-then-code cycle gives continuous feedback to the developer.

- **Low-Level design:** The tests provide a specification of the low level design decision in terms of the classes, methods and interfaces created.

- **Reducing Defect Injection**. Often with debugging and software maintenance, working code is "patched" to alter its properties and specifications, and designs are neither examined nor updated. Unfortunately, such fixes and "small" code changes may be nearly 40 times more error prone than new development [11]. By continuously running these automated test cases, one can find out whether a change breaks the existing system.

- **Test Assets:** TDD makes programmers write code that is automatically testable. Such automated unit test cases written with TDD are valuable assets to the project in terms of regression testing [16].

3. RELATED WORK

We discuss the empirical studies on TDD reported to date in terms of the environment in which they have been performed. We classify the previous work in this area broadly as industrial and academic case studies.

3.1 INDUSTRIAL CASE STUDIES

An year-long empirical study performed at IBM [16] using professional programmers found that the TDD practice helps programmers produce higher quality code. The IBM team produced a thorough suite of automated test cases after UML design. The code developed using the TDD practice showed, during functional verification and regression tests, approximately 40% fewer defects than a baseline prior product developed in a more traditional fashion. The productivity of the team was not impacted by the additional focus on producing automated test cases.
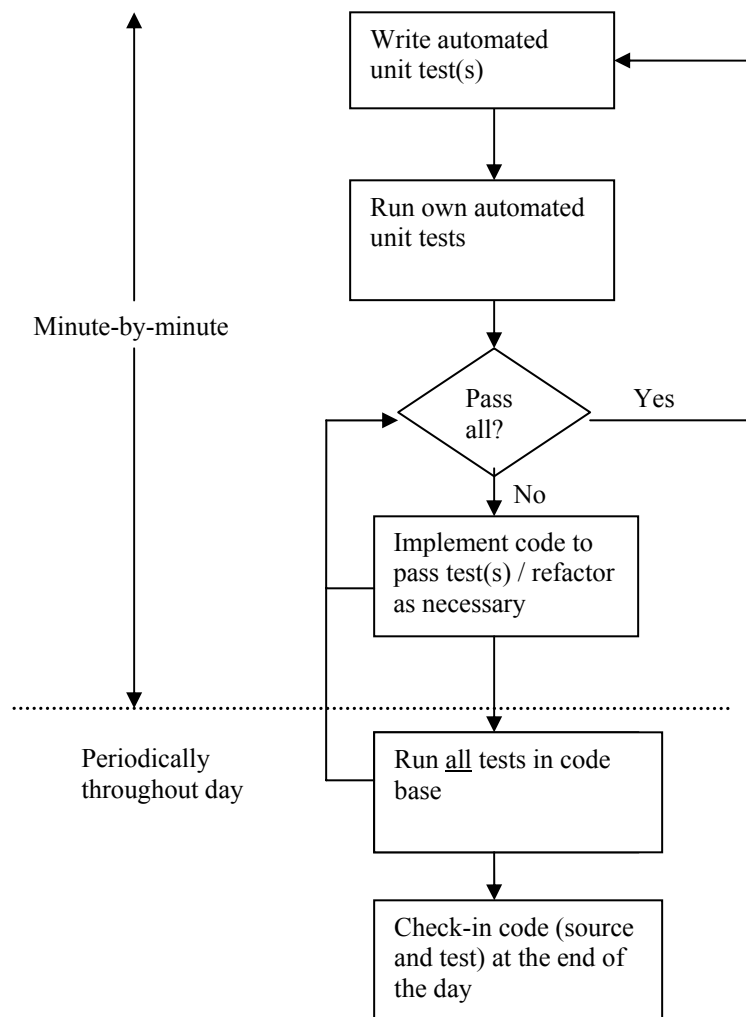

Figure 1: Test-Driven Development

A structured experiment involving 24 professional programmers from John Deere, Rolemodel Software and Ericsson [10] was performed to investigate the efficacy of TDD. One group developed a small Java program using TDD while the other control group used a waterfall-like approach. Experimental results indicated that TDD programmers produce higher quality code because they passed 18% more functional black-box test cases. However, the TDD programmers took 16% more time. However, the programmers in the control group often did not write the required automated test cases after completing their code.

3.2 ACADEMIC CASE STUDIES

Müller and Hagner found that a TDD-variant did not help to produce a higher quality system [13]. In this study, the programmers had to write their complete set of automated unit test cases before writing any production code rather than in the highly iterative manner of TDD. Erdogmus et al. [7] performed a controlled investigation regarding Test-first and Test-last programming using 24 undergraduate computer science students. They observed that TDD improved programmer productivity but did not, on average achieve better quality. Their conclusions also brought out a very valid point that the effectiveness of the Test-first technique depends on the ability to encourage programmers to back up their code with test assets. Müller and Tichy [14] investigated XP in a university context using 11 students. From a testing perspective they observed that in the final review of the course 87% stated that the execution of the test cases strengthened their confidence in their code.

3.3 CONTRIBUTIONS

Our work builds up on the prior empirical work in this area. Carver et al. [5] discuss in detail about using students as subjects in empirical studies. Academic case studies provide a meaningful ground for researchers to try out their ideas before replicating the studies in industry. From an industry perspective there have been limited case studies performed on exploring the efficacy of TDD. With respect to the published industrial case studies, George et al. [10] investigated TDD using professional programmers working on a trivial program and not real commercial software systems. The IBM study [16] compared the results of running functional verification tests as quality measures and not defects. This though is an approximation the actual quality measures in terms of defect density provides a more accurate view of the quality of the system.

We address these issues by performing our case study using professional programmers on real software systems in different environments. We selected the projects to be as dissimilar as possible with respect to their domain, size, program managers expertise, language of programming, and development time. This helps us in an empirical perspective to observe if similar outcomes can be obtained using different contexts.

Our main contributions are highlighted below,

- *TDD evaluation using commercial software in two different divisions at Microsoft.* This addresses threats to validity from a development environment point of view, i.e. the results are similar in two different environments: Windows an operating system and MSN a primarily web based system. Also other than the earlier IBM study [16] there have been hardly any studies on investigating TDD from a commercial software development viewpoint.

- *Accurate quality measurements with comparable projects in terms of defect density to measure quality.* Earlier studies have all used other quality measures like functional verification tests, black box tests to measure quality. We mine the defect databases to obtain an accurate measure of software quality.

- *Quantification of increase in development time due to adoption of TDD.* With TDD there is usually an increase in the actual development time of the system. We attempt to quantify the increase in development time using project management estimates from the relevant managers.

- *Contribute to strengthening the existing empirical body of knowledge regarding investigations of TDD.* Our study contributes towards strengthening the existing results from academia and industry on the use of the TDD process in software development. Also most of the published studies have been performed on Java using the Junit testing framework. Our studies involve development using C++ - CppUnit framework and the C# - NUnit framework combinations. This serves to indicate the generalization of results across programming languages. (A detailed code example for NUnit is given in appendix A)

4. CASE STUDIES

In this section we discuss two case studies performed at Microsoft Corporation using TDD. These studies were performed in two different divisions inside Microsoft, Windows and MSN respectively. For the sake of convenience we call the projects, A (Windows division) and B (MSN division) respectively. For selecting comparable projects in order to compute the difference (increase/decrease) in quality we select the projects as shown in Figure 2. Both the projects are developed by managers with similar levels of responsibilities (Alice and Bob) and reporting to the same higher level manager (Tom). This ensures that we compare projects with respect to the same usage domain and not make unequal comparisons (for example between a children's game software and space shuttle software). Also these projects were analyzed in retrospect after completion and the developers did not know during development that their work was going to be assessed. This minimizes any influence there might have been on the developer's performance.
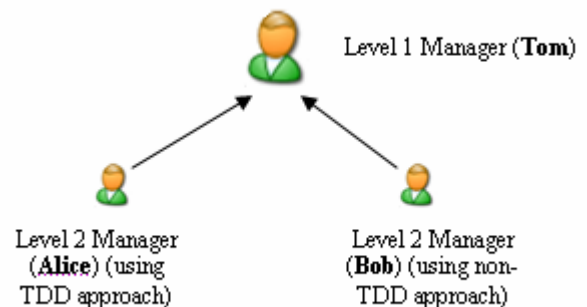

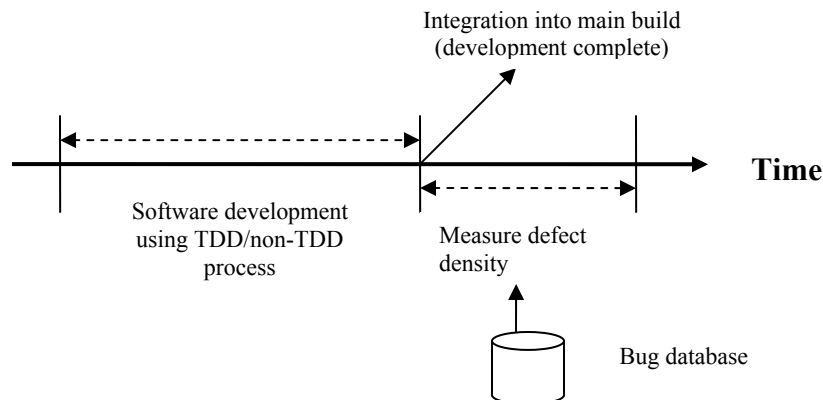**Figure 2: Comparable teams' organization**

Figure 3: Process timeline for development and defect density measurement

Upon selection of these projects we measure the quality in terms of defect density. To better distinguish between defect and failure we provide the following definition. "When software is being developed, a person makes an error that results in a physical fault (or defect) in a software element. When this element is executed, traversal of the fault/defect may put the element (or system) into an erroneous state. When this erroneous state results in an externally visible anomaly, we say that a failure has occurred" [1]. We use the above stated definition of defect, and normalize it per thousand lines of code (KLOC) for deriving our quality measure defect density. As with most large organizations Microsoft uses a bug tracking system that stores various context and time sensitive information. Figure 3 describes the process of mining the defects from the bug database after a specific time in the software development process (i.e. integration of the system into the main build).

We do not discuss or assess the efficacy of XP or any other agile processes as it beyond the scope of this paper. One of the teams used agile methodologies for development whereas the other team used TDD as a stand alone practice. For each of the projects we discuss specific context, product and outcome measures that provide detailed information about the specifics of the projects [15]. Context factors are critical for understanding the relevant environments of the projects and are useful while replicating studies and making comparisons. The product measures are derived from the product itself and the outcome measures are used to compare the results of evaluating the efficacy of one technique over another, in our case defect density and increased development time.

4.1 CASE STUDY A

This case study was performed in the Windows networking team. The unit testing framework and the design architecture for the networking library is shown in Figure 4.

The networking common library and the unit test code are written in C++. The unittest code inherits from CUnitTest class from unittest.h. They may inherit from the relevant classes from the networking library or may use the class under test as a container

class. The unittest class and library provide logging and command line output functionality. The unit test framework enables the developer to figure out the status of the test run from the command line. The log files are used for deeper analysis of passing/failing tests.
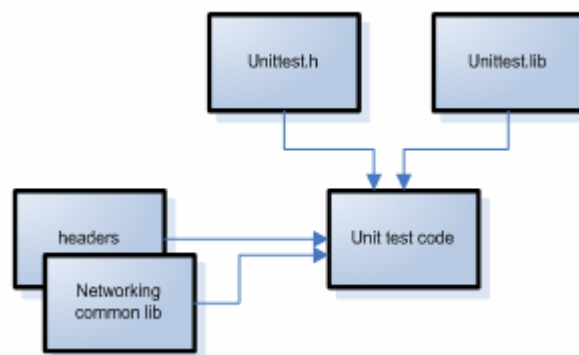

Figure 4: TDD in Networking

The networking common library was written by the tool development team as a re-usable set of modules for different projects within the networking tools team. This library provides seventeen different classes that provide functionality commonly used in network tools. Some examples are generic packet manipulation routines, generic command line manipulation, memory allocation/tracking routines, random number generator and timer classes. This library is used by more than fifty internal applications.

The context factors for project A are presented in Table 1. The context factors indicate the size of the development team involved (6 developers – not including testers/program managers) and their experience levels. Project A consisted of an experienced set of people in terms of the program manager's expertise, domain and language expertise. The project was developed using C/C++ with all developers located in Redmond, USA.

Table 1: Project A – Context Factors

Metric description		Value
Team Size (only developers)		6
Team Location		Redmond, WA
	> 10 years	0
Experience Level	6-10 years	5
	< 5 years	1
Domain expertise (Low, Medium, High)		High
Language expertise (Low, Medium, High)		High
Programming Language		C/C++
Program Manager's expertise (Low, Medium, High)		High
Team location		Collocated

The overall product metrics for the developed system are shown in Table 2. The product measures are derived from the product itself and broadly discuss the size and test effort involved in the project. Project A though not large is a non-trivial project that has a 79% block coverage based on the unit-test effort alone. This coverage is exclusive of the coverage attained by the test teams. The overall development time also for the two projects in presented in man-months.

Table 2: Project A - Product Measures

Metric Description	Value
Source LOC	6 KLOC
Test LOC	4 KLOC
Test LOC/ Source LOC	0.66
% Block coverage (unit-tests)	79%
Development time (in man-months)	24
Legacy code (Yes/No)	No
Comparable project - Metric Description	**Value**
Source LOC	4.5 KLOC
Development time (in man-months)	12
Team Size	2

From the outcome measures perspective as shown in Table 3 we observe that there is greater defect density (greater than two and a half times) for the project that did not employ TDD. Though estimates from managers put the increase in development time for the TDD project was in the order of 25-35%. From our first case study, project A we can hence observe a significant improvement in quality of the system developed using TDD. Alternatively there is a 25-35% increase in the overall development time for the team employing TDD.

Table 3: Project A- Outcome Measures

Metric Description	Value
Actual defects/KLOC (using TDD)	X
Defects/KLOC of comparable team in org but not using TDD	2.6X
Increase in time taken to code the feature because of TDD (%) [Management estimates]	25-35%

4.2 CASE STUDY B

This case study was conducted on a web services application in the MSN division at Microsoft. The related context factors are shown in Table 4. The project during its development lifecycle involved between 5 and 8 developers. The personnel involved in project B were less experienced than the personnel involved in project A in terms of program managers expertise, domain and language expertise. The development team was collocated in Redmond, USA.

Table 4: Table 1: Project B – Context Factors

Metric description		Value
Team Size (only developers)		5-8
Team Location		Redmond, WA
	> 10 years	1
Experience Level	6-10 years	7
	< 5 years	0
Domain expertise (Low, Medium, High)		Medium
Language expertise (Low, Medium, High)		Medium
Programming Language		C++/C#
Program Manager's expertise (Low, Medium, High)		Medium
Team location		Collocated

The product measures indicate the size of the projects and overall test effort, and the development time in man months. The ratio of the lines of code between source and test is high (0.89). The block coverage is also higher than project A at 88% from the unit testing effort alone.

Table 5: Project B - Product Measures

Metric Description	Value
Source LOC	26 KLOC
Test LOC	23.2 KLOC
Test LOC/ Source LOC	0.89
% Block coverage	88%
Development time (in man-months)	46
Legacy code (Yes/No)	No
Comparable project - Metric Description	**Value**
Source LOC	149 KLOC
Development time (in man-months)	144
Team Size	12

For project B the outcome measures shown in Table 6 indicate that there is a 4.2 times increase in the quality of code developed using TDD compared to a project using a non-TDD approach in the same organizational hierarchy. Additionally the project developed using TDD took 15% more time than the non-TDD project.

Table 6: Project B- Outcome Measures

Metric Description	Value
Actual defects/KLOC (using TDD)	Y
Defects/KLOC of comparable team in org but not using TDD	4.2Y
Increase in time taken to code the feature because of TDD (%) [PM/Dev LEAD/MGR estimates]	15%

5. THREATS TO VALIDITY

The main threats to validity are,

- Developers using TDD might have been more motivated to produce higher quality code as they were trying out a new process and this could have resulted in higher quality code for the TDD projects. To an extent this is alleviated by the fact that these were professional programmers all of whom had to have their tasks completed.

- The projects developed using TDD might have been easier to develop as there can never be an accurate equal comparison between two projects except in a controlled case study. This is alleviated to some degree by the fact that these systems were compared within the same organization (with the same higher level manager). It would be unlikely that there will be a huge discrimination in the complexity of the TDD and non-TDD projects.

- As with all empirical studies, these analyses should be repeated in different environments and in different contexts before generalizing the results. Several teams in Microsoft employ TDD as part of the development process. We intend to collect data from all these projects and build an empirical body of knowledge in this field.

6. CONCLUSIONS AND FUTURE WORK

In this paper we have investigated the utility of TDD from the view point of software quality and productivity in terms of development time. The case study is performed with software professionals in the Windows and MSN teams with different organizational goals, platforms and environments. The results summarized in Figure 5 indicate that while the development of both the systems utilizing TDD took extra time upfront the resulting quality was higher than teams that adopted a non-TDD approach by an order of at least two times.

Drawing general conclusions from empirical studies in software engineering is difficult because any process depends to a large degree on a potentially large number of relevant context variables. For this reason, we cannot assume a priori that the results of a study generalize beyond the specific environment in which it was conducted [2]. Researchers become more confident in a theory when similar findings emerge in different contexts [2]. Towards this end we intend that our case study contributes towards strengthening the existing empirical body of knowledge in this field [7, 9, 13, 14, 16].
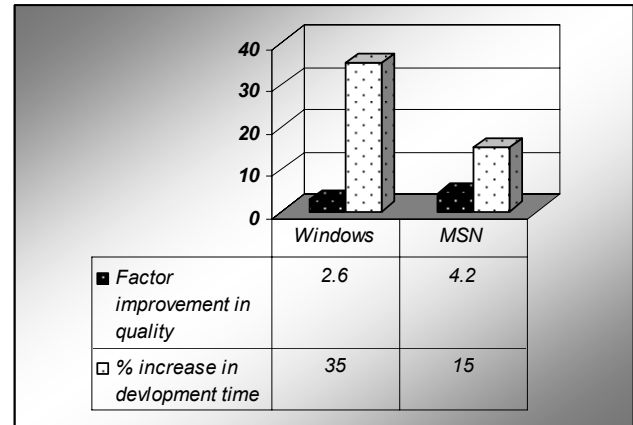

	Windows	MSN
■ Factor improvement in quality	2.6	4.2
□ % increase in devlopment time	35	15

Figure 5: Results summary

In addition to building an empirical body of knowledge on the efficacy of TDD by replicating the studies inside Microsoft we also intend to perform a cost-benefit economic analysis on the utility of TDD. This involves a return of investment analysis to determine the cost-benefit tradeoff between the increase in development time and the resulting improvement in software quality. Such measures we hope will help project managers make meaningful decisions about the utility of deploying TDD in their organization.

ACKNOWLEDGEMENTS

REFERENCES

[1] "IEEE Std 982.2-1988 IEEE guide for the use of IEEE standard dictionary of measures to produce reliable software," 1988.

[2] V. Basili, Shull, F.,Lanubile, F., "Building Knowledge through Families of Experiments", *IEEE Transactions on Software Engineering*, 25(4), pp. 456-472, 1999

[3] K. Beck, *Extreme Programming Explained, Embrace Change*: Addison Wesley, 2000.

[4] K. Beck, *Test Driven Development- by Example*. Boston: Addison-Wesley, 2003.

[5] J. Carver, Jaccheri, L., Morasca, S., and Shull, F., "Issues Using Students in Empirical Studies in Software Engineering Education ", Proceedings of IEEE Metrics, pp. 239-249, 2003.

[6] B. Curtis, "Three Problems Overcome with Behavioral Models of the Software Development Process (Panel)", Proceedings of International Conference on Software Engineering, Pittsburgh, PA, pp. 398-399, 1989.

[7] H. Erdogmus, Morisio, M., Torchiano, M., "On the effectiveness of the Test-First Approach to Programming", *IEEE Transactions in Software Engineering*, 31(3), pp. 226-237, 2005.

[8] D. Gelperin and W. Hetzel, " Software Quality Engineering", Proceedings of Fourth International Conference on Software Testing, Washington D.C., June 1987.

[9] B. George and L. Williams, "An Initial Investigation of Test-Driven Development in Industry", Proceedings of ACM Symposium on Applied Computing, Melbourne, FL, pp. 1135-1139, 2003.

[10] B. George and L. Williams, "A Structured Experiment of Test-Driven Development", *Information and Software Technology (IST)*, 46(5), pp. 337-342, 2003.

[11] W. S. Humphrey, *Managing the Software Process*. Reading, Massachusetts: Addison-Wesley, 1989.

[12] C. Larman and V. Basili, "A History of Iterative and Incremental Development", *IEEE Computer*, 36(6), pp. 47-56, June 2003.

[13] M. M. Müller and O. Hagner, "Experiment about Test-first Programming", Proceedings of Conference on Empirical Assessment in Software Engineering (EASE), 2002.

[14] M. M. Müller and W. F. Tichy, "Case Study: Extreme Programming in a University Environment", Proceedings of 23rd International Conference on Software Engineering, pp. 537-544, May 2001.

[15] L. Williams, Krebs, W., Layman, L., "Extreme Programming Evaluation Framework for Object-Oriented Languages -- Version 1.1," Technical Report, North Carolina State University, NCSU CSC TR-2003-20, 2003.

[16] L. Williams, E. M. Maximilien, and M. Vouk, "Test-Driven Development as a Defect-Reduction Practice", Proceedings of IEEE International Symposium on Software Reliability Engineering, Denver, CO, pp. 34-45, 2003.

Appendix A

We show in Appendix A an example for the use of NUnit. Most prior studies of TDD have involved Java, hence we present an NUnit example that is used to test a C# program. The program is designed to add, subtract and divide two numbers. The schema and the results of running the NUnit test are show in Figure A.1 and Figure A.2. Figure A.3 present the actual NUnit code to add, multiply and divide two numbers.
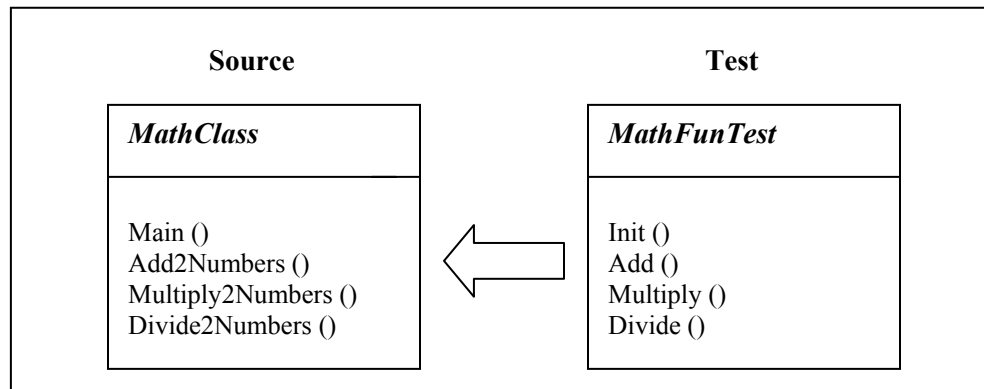

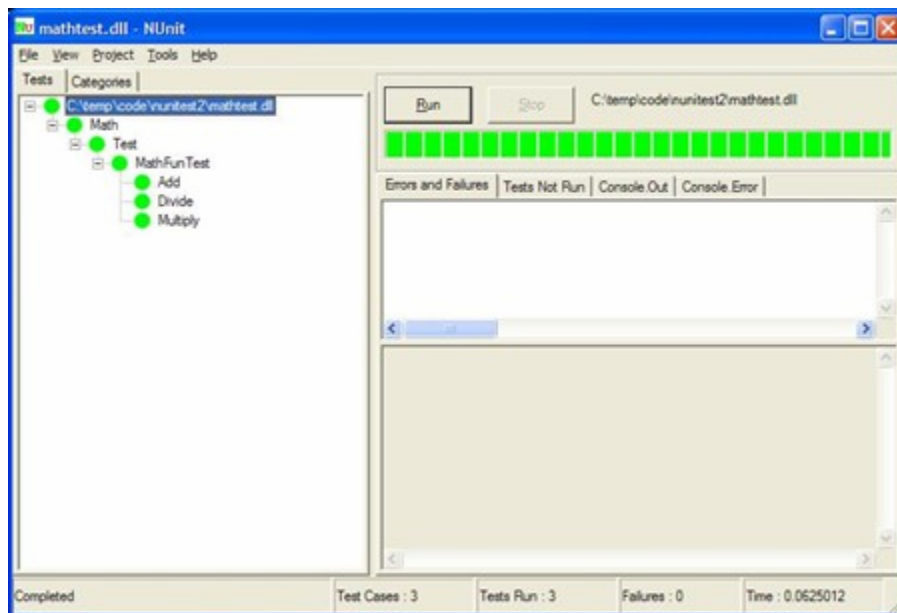**Figure A.1: Source and Test Classes to Add, Multiply and Divide two numbers**

Figure A.2 Screen shot of NUnit after Add, Multiply and Divide tests pass

```csharp
namespace Math.Test
{
        using System;
        using MathFun;
        using NUnit.Framework;

        [TestFixture]
        public class MathFunTest
        {
          protected int fValue1;
          protected int fValue2;
          protected MathClass testClass;

          [SetUp] public void Init()
          {
              fValue1= 5;
              fValue2= 10;
              testClass = new MathClass( );
          }

          ///  test AddTwoNumbers Function

          [Test] public void Add()
          {
              int result= testClass.AddTwoNumbers(fValue1,fValue2);

              // check that the returned result == expected result
              Assert.AreEqual( 15, result, "Expected Pass" );
          }

         /// test MultiplyTwoNumbers Function

          [Test] public void Multiply( )
          {
              int result = testClass.MultiplyTwoNumbers(fValue1,fValue2);

              // check that the returned result == expected result
              Assert.AreEqual( 50, result, "Expected Pass" );
          }

          ///  divide fValue2 by fValue1

          [Test] public void Divide( )
          {

              int result = testClass.DivideTwoNumbers(fValue1,fValue2);

              // check that the returned result == expected result
              Assert.AreEqual( 2, result, "Expected Fail!" );
          }

        }
}
```

Figure A.3: NUnit test code (MathFunTest class) to Add, Multiply and Divide two numbers