Tool Evaluation of Microsoft's Unit Testing Framework

Himanshu Arora
Department of Computer Science, North Carolina State University
{harora} @ncsu.edu

Abstract

Unit testing is a crucial part of any software Development which follows test driven model (TDD). This paper focuses on evaluating Microsoft's Unit Testing Framework (MSTest) which comes integrated with Visual Studio, on open source software NuPack.

Based on our evaluation, we will compare the usability and performance metrics of such unit testing tools which are integrated with an integrated development environment (IDE) and those which are not.

The results from this paper may also help developers or organizations decide which set of tools or frameworks are best suited for them to carry on Test Driven Development.

1. Introduction

In a software development process, white box testing is a verification technique used by software engineers to examine whether their code works as expected. It takes into account the internal mechanism of the system [5].

Unit testing is a type of white box testing where individual units of software are verified. It is an important part of software development as it ensures that the code is solid before it gets integrated with other code. According to research conducted [11] about 65% of bugs can be caught by unit testing alone.

There are plenty of tools and frameworks available that promote Test Driven Development (TDD) among developers like NUnit, csUnit, MSTest etc. More or less, function wise, these set of tools and framework are quite comprehensive depending on what kind of environment we are working in. For a small scale developer it sounds good to go with free and open source testing frameworks. Large scale organizations have plenty of options to choose their testing frameworks as budget is not the main concern.

Keeping this in mind, the focus is on applying these testing tools on open source software and surveying how these developers or organizations can choose which set of such tools or frameworks are suited best for them to carry on Test Driven Development.

The Visual Studio Unit Testing Framework describes suite of unit testing tools integrated into Visual Studio 2005 and later. This unit test framework is tightly integrated into Microsoft's Visual Studio to provide developers an ease of covering their code using Test Driven Development from within the integrated IDE. The framework can also be loaded from Dynamic Link Libraries and tests can also be run using command line if a developer wants to have a different choice of IDE.

NUnit on the other hand, is a free unit testing framework which is not seamlessly integrated with an IDE. It loads test assembly in different application domain and keeps a watch on targeted assembly for any changed events.

The paper highlights the current practices in unit testing and test driven model in the beginning and then gives brief idea of how TDD is done using unit testing frameworks. Then the paper explains the tools and requirements for the experimental setup which is discussed later.

The experiment is done in the other half of the project where the performance metrics of integrated unit testing frameworks will be discussed and then at the end it concludes with the analysis reports comparing the aforementioned unit testing frameworks.

2. Current Practices

2.1 Unit Testing

The term unit test is a result of Extreme Programming. Most of the IT projects developed today have to target some deadline date and have developers who need to test their own code before the app goes for regression testing. These unit tests can be valuable assets to the whole project paving the way to a bug free and quality code [7]

There can be two ways to write unit tests, either you write it before any code implementation or you can write it after you write some set of code for a new feature to be implemented. If a developer writes a unit test after the code implementation then there are high chances that he may be assuming couple of things that he implemented in the application code. However, if a developer wants to change the functionality of an application, his old test cases may not get cleared by

the new code. In that case he may have to write the unit tests again.

Unit testing is sometimes ignored or poorly performed as the developer writing the code as well as test cases can take lot of time and hence the development cycle can be very much expensive [5]. For this, they are written before implementation of any features. The software architect or analyst can also write the unit test cases prior to the development cycle. This also makes sure that we have well defined deliverables and hence can speed up the development time.

For a developer also, he can set his goals specific to these test cases and can check his progress against them for a day or a week [12]

2.2 Test Driven Development

Test Driven Development is a software engineering process that follows a small development cycle. Before directly start writing code, developer or a tester first writes an automated test so that the given set of feature fails in a program. Then a developer writes code to make sure that all the test cases pass through it and finally refactors the new code following the standards and protocols.

TDD has been used since quite a long time and currently five step development practices is followed in industry while coding a software application [10, 7]. The steps are -

1) *Adding a test case:* In a TDD, before writing any code to implement a feature, test cases are proposed and written. Since a developer or a tester is aware of features that have not been implemented, the test case he writes must fail. For this, it is very important that the person writing test cases should understand feature's specification and requirements which may also be depicted in the form of use cases and user stories, covering all the requirements and conditions for exception.
This is the benefit of writing unit tests before writing code. It makes developers focus on the requirements. Also, writing code in this manner makes code more cohesive and less coupled [7]

2*) Running all unit tests and analyzing failing cases:* If any of the test cases that a developer has written in step 1 fails, he could ascertain that either he needs to write some code to pass that test or fix the already written code. Also he needs to make sure that the test case fails for some known reason. Whole of this process gives assurance that he is testing the correct feature set and it will only pass his proposed test cases. A special test case [2] would also help developer to reproduce any bug from his previously written code

3) *Write code*: Once a developer knows which test cases are failing, he needs to write code that will clear all these tests. However the code he writes may not be completely flawless at this stage as he may refine it during later stages. Also, developer needs to make sure that whatever code he is writing should not add any other unpredicted functionality. For this purpose he may prioritize his unit tests based on what changes have been made to the code [8].

4) *Running and clearing all automated tests:* After writing code, developer needs to make sure that all the test cases that were written before, even if they have nothing to do with the current implementation, should pass. This is an indication that now he can work on improving the code structure without compromising the program quality.

5) *Code Refactoring*: Since now developer is sure that the code which he has written clears all the tests; he may restructure his code for better readability or improving the performance. Advantage of above written unit tests is that whatever changes he may make to the code now, he can be ascertain that this won't affect the existing functionality of the build [7], as the test cases written earlier defines the requirement and specifications of the application.

3. Tools and requirements

To run the experiment, minimum hardware requirements is the requirements of Visual Studio integrated with MSTest, which are –

- **Operating System:** Windows XP SP 2+ or Windows 7
- **Platform:** Microsoft Visual Studio 2005 , 2008 or 2010 Team Test (VSTT)
- **Software:** NUnit, MSTest, , NuPack which is the Open Source software(OSS) for which we'll create unit tests.
- **PC** with 2GB DDR RAM, Intel Core 2 Duo/Pentium 4 1.6+ GHz, 10GB of hard disk space.

Each of the software can be easily downloaded from the internet for free except Microsoft Visual Studio for which you need to get a License. However to try out the experiment, you can download a trial version. If you are a student you can download the express edition of Microsoft Visual Studio to try out NUnit, though it doesn't come with integrated unit test framework.

4. OSS to analyze

NuPack is a free, open source developer focused package management system for the .NET platform

intent on simplifying the process of incorporating third party libraries into a .NET application during development.

There are number of useful third party libraries made in .NET which are unknown to developers and they cannot locate them even within the Visual Studio by just adding a reference. Usually those references list the COM or .NET libraries which are by Microsoft but don't show the third party libraries which are available on the system or present online.

So when a developer wants to add these libraries to his project, he first needs to download the library from the internet and then include it in his project by adding a reference to it.

Nupack makes whole of this process easy for a developer to find and include third party package in his project within Visual Studio or a console.

NuPack has huge file base inside about 10 projects which compiles into several dynamic linked libraries. For this project we'd be using the Core library of NuPack which is defined in NuPack namespace.

We'd also be creating a separate Test project and will write unit tests to verify about 30-35 functions inside the various classes of the Core Library of the NuPack.

5. TDD using frameworks

To show what is Test Driven Development (TDD) and how is it done with unit testing frameworks, let's first create a class and its methods in Visual Studio editor. For this purpose I will take a real life example of ATM machine where user inserts an ATM card (having some card Id) and a pin number.

Let's assume that our aim here is to write a quality code to develop a dynamic link library for authentication process, which will be referenced by a software running inside an ATM machine. This library will take the authentication details and then verify it with the Bank through a secure channel.

I'll first create a basic skeleton structure defining the members and properties of a class and initializing the instance variables of the class inside the constructor .[12]

```
namespace Authentication
{
    class UserAuth
    {
          public string CardId
        {
            get;
            set;
        }
        public string Pin
        {
            get;
            set;
        }

        private void UserAuth(string cardId,
string pin)
        {
            this.CardId = cardId;
            this.Pin = pin;
            //Connect with Bank here or call
explicitly
        }

    }
}
```

Figure 1: Sample class for user authentication

Now as a first step of TDD, without further implementing the code, we'll cover the code by writing unit tests. To do this using unit testing frameworks we first write a class for our test case and then write functions in them to test a particular method using assertions.

Assertion is a statement used by developers inside a program to indicate what they think is true. Whole Test Driven development is based on these asserts inside the code. When a developer writes a comment inside a program, the runtime execution environment knows nothing about them. However these assert statements provide run time checks of their assumptions that would have otherwise put in the comments. Basically, assert statements act as a test code which is integrated directly into the code implementation.

Here our aim is to write a unit test as per the requirements which should not be cleared by the current code. Following code covers such test case checking if the pin is non-numeric.

```
 [TestMethod]
[ExpectedException(typeof(FormatException),
    "Pin number is not numeric.")]
        public void NonNumericPinTest()
        {
            UserAuth target = new
UserAuth("WAC32332", "FG23");
        }
```

Figure 2: Writing test case that fails on non-numeric pin number.

Now on Test project execution we get a failing test-

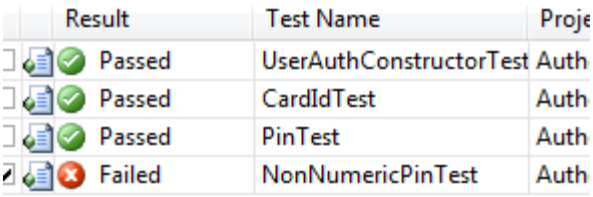

Figure 3: Test results pane showing failing tests inside Visual Studio.

This means that I need to rewrite my code in order to comply with the test case. Following is the rectified code for the Constructor now –

```
public UserAuth(string cardId, string pin)
        {
            this.CardId = cardId;
            int num;
            if (int.TryParse(pin, out num))
                this.Pin = pin;
            else
                throw new FormatException();
        }
```

Figure 4: Fixing the code after the test case failed

The last step now clears all the unit tests and we are free to refactor the code we have written or writing few more test cases as per the standards mentioned above.

2.4 TDD with NUnit

NUnit is a free and open source Unit Testing framework used to test .Net applications. It has been ported from JUnit which is another popular testing framework for Java and written in C# to take full advantage of other .Net languages.

Unlike JUnit which provides solid integration with Eclipse IDE, NUnit doesn't have such seamless integration with Visual Studio [1]. For a developer to write unit tests using NUnit the learning curve may be bit steeper.

Since the template for the test cases are not generated automatically in NUnit, it means that for a developer who is totally unaware about unit tests or TDD, has very less scope of exploring things unless he has got some training on using these frameworks. Also, just fresh out of the box NUnit doesn't comes integrated with any IDE. We need to run a separate NUnit app and then load the assemblies where we have written code for Unit tests [12].

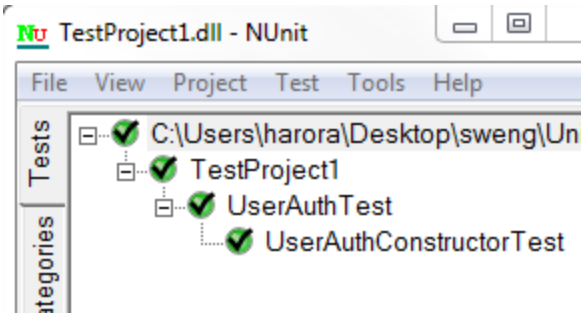

Figure 5: NUnit interface showing successful test cases

The NUnit app itself doesn't provide any interface to create unit tests very fast, like the way we did for Microsoft's framework inside Visual Studio. On the other hand, we have plenty of options integrated with Visual Studio to create or debug unit tests, create tests' list, collaborate with team etc.

6. Experimental setup

Most of the core features of all the unit testing frameworks are same. The most common of these features are test declaration, test execution and pass/fail notices on assertions.

The paper aims at comparing the usability and performance of certain set of features. By usability, we mean how developer-friendly is the integrated unit testing framework when compared to the third party unit testing framework which has to be run externally.

For our experiment, we'll be using the core functions of the open source project NuPack which resides in the Nupack.Core Library.

To do the investigation and analysis of both the frameworks, we will be testing exhaustive set of test cases. In the Nupack's core library we'll be unit testing functions of numerous classes which are written in the implementation of NuPack's core library.

Since we were intending to do TDD, we needed to prepare the environment and write same test cases for both the frameworks. These test cases can either be written separately in different projects one for each framework or can also be written inside one wrapper class which checks for the performances of both the frameworks inside the same test case.

The analysis to measure the performance was done by comparing the time of execution of the assertions of both the frameworks. MSTest comes with an UI interface which displays the start time and the end time of the test execution. However, since NUnit

doesn't come with any such interface, we needed to write a wrapper class *TestPerformance* which compares the time of execution of assertions using the same metrics.

The *TestPerformance* Class was written in the same namespace as of the Open source project and each test case written inside it compares the time taken by the assertion of both the frameworks. The class can also be extended to check other metrics if required in future.

7. Analysis

The analysis of the experiment was done starting from the downloading and installing the framework, setting the environment, reading documentation, writing first sample test case and its execution by developers who were acquainted with basic knowledge of running Visual Studio and had very little information about unit testing.

This was done primarily to compare the usability metrics of these frameworks, one of which is integrated inside an IDE.

To compare the performance metric, as described in the experimental setup, we created a separate project with a wrapper class TestPerformance and wrote about 20 Test case functions, each of which uses methods from both the frameworks and compare the time writing it on a Console Window.

Following are the detailed explanation of the factors that were used to compare both the frameworks.

Since for both the frameworks we used Microsoft Visual Studio as an IDE to write and edit code, we assume that Visual Studio 2008 or later is installed on the system where the test needs to be executed.

7.1 Download and Installation.

Now since the Visual Studio is installed on the system, we need to download and install NUnit binaries which come as a Microsoft installer package.

The binaries and documentation of NUnit can be downloaded from their website. It comes as an installer package or can also be downloaded in a standalone folder which is archived inside a file.

The installation of NUnit is quite simple process but still the developer remains clueless what he needs to do next. In the Visual Studio the menu bar gives enough indication about creating test cases which we will discuss later in the same section.

7.2 Setting up an Environment

For Visual Studio, a developer doesn't need to setup an environment or include any of the testing libraries inside the project as everything is built in and integrated. However for NUnit, a developer will require to read the documentation or search internet to begin.

To get started with NUnit, we'll have to manually create a test project to contain the NUnit tests. Once we create a project, it requires certain amount of non-trivial work to finally start writing a test case.

Before we start writing a test case, we should include NUnit's linked libraries to the project. To include them, we need to add references to the Core library of NUnit i.e. Nunit.Core and the whole NUnit framework which is in NUnit.framework dll.

MSTest's tight integration with the Visual studio IDE does help here. A developer can directly include a new test project in the solution and all the required libraries of MSTest will automatically be included in the project.

Here the developers were aware of the basic practices of software engineering and knew that they need to create a separate test project to write unit tests as we won't be shipping the product with the unit test code written inside them.

7.3 Writing First Unit Test

Now since the environment is all set for both the frameworks, we need to write unit tests.

In our experiment, two developers had never written any unit test to verify the requirements and maintain the integrity of the code. The only thing that they knew was that they can call methods with various inputs to verify if the function gives correct output for each of the input.

With the NUnit, developer had no choice but to search the internet or read the documentation to start writing his first unit test, verifying the function of the open source software used.

On the other hand, Visual Studio gave couple of options to the developer to explore about unit tests, of course with the intention of writing one, and start writing his first test case.

For example, while on the source code file, a developer can right click on any method or even on the class name, and the Visual Studio interface will give him an option to create unit test for that class or method.

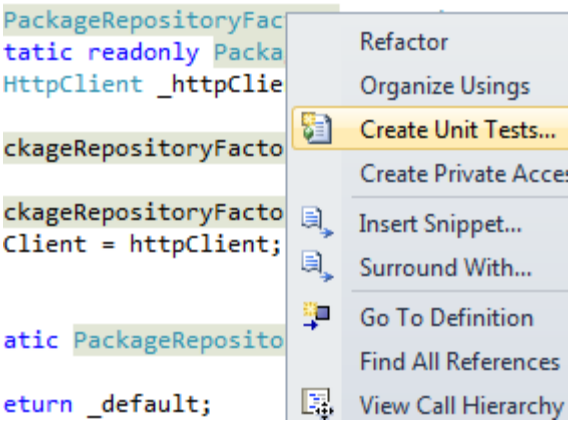

Figure 6: Create Unit test option in the integrated environment of MSTest.

Also, the IDE shows an option on the menu for creating new Tests. This integrated feature is quite useful for instantly creating unit tests after writing a function.

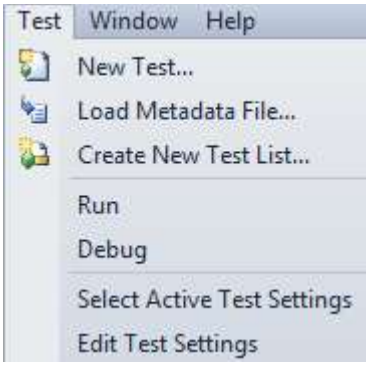

Figure 7: Integrated menu option to create and run unit tests in the integrated environment of MSTest.

In the case of NUnit, once a developer writes class, he has to create a Test project, or if the Test project is already there, he has to manually write separate class to write the test case as there are no integrated features built in inside an IDE.

7.4 Running Test cases

Once we are done with writing test cases, we need to verify the assertions that we had written inside them.

Once again, execution of test cases that were written in MSTest, were very easy to figure out for the developers, like the way they figured out how to write a unit test.

After writing a unit test, again, either right click option or the menu bar gave an option to run those tests. The results of the execution pop out once MSTest starts executing the unit tests.

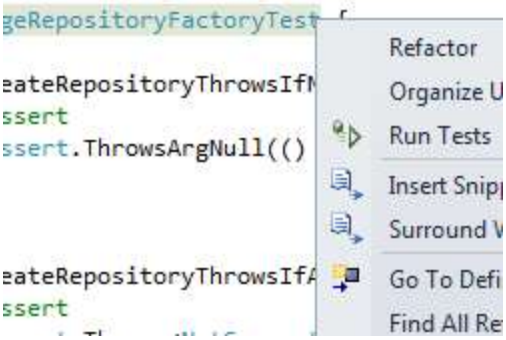

Figure 8: Running Unit tests inside the integrated environment of MSTest.

Now, to execute the tests written for NUnit, once again, developer should be acquainted with the whole process. First he needs to compile the test code to generate a binary file and then load that binary into NUnit and then run the tests for the desired functions.

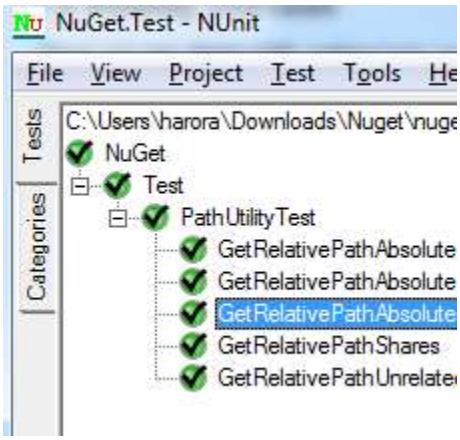

Figure 9: NUnit GUI to execute unit tests

This means that if a developer is creating unit test for any major implemented function, he needs to reload the binary in the NUnit and then select the tests that he wants to run in it.

While in Visual Studio, developer can instantly know where is his test case failing and before writing any further code he can fix his code to prevent any bugs. By doing this developers save lot of their time and have to spend less time in fixing bugs which are usually passed on to the testers.

7.5 Code Coverage

Code coverage is an important metric in unit testing which determine how much code is testing once the unit tests run. Microsoft Visual Studio also includes a code coverage tool. This tool lets developers know about the percentage of code that is executed while performing unit tests and the code is highlighted

within the IDE to show which lines are executed and which are not.

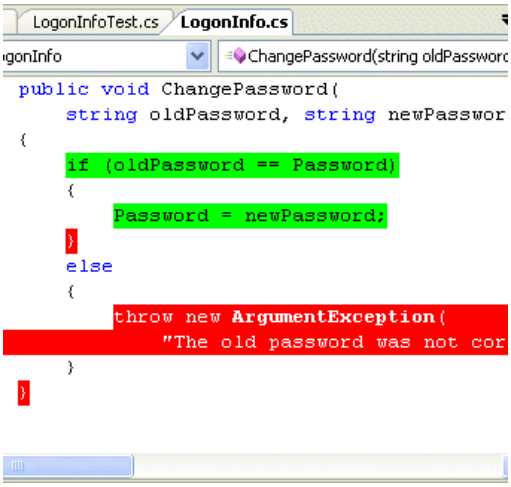

Figure 10: Code coverage tool inside Visual Studio

Though there are many code coverage tool available which can be integrated with NUnit, for example, NCover, SD Test Coverage etc, but they don't get integrated with the Visual Studio and most of them are Licensed.

7.6 Comparing the Performance

As explained in the Experimental Setup Section above, to test the performance of both the frameworks, we created a Test project inside the Visual Studio and included NUnit libraries in it so that inside the one unit test we can compare the performances of both the assert calls.

Though this can also be done using Aspect Oriented Programming, but for our case we wrote a separate wrapper class TestPerformance which calculates the total time taken by the same assert statements of both the frameworks.

Also, this class can be extended to compare other metric if desired in future.

To compare the performance, we wrote about 35 unit test functions inside the wrapper class and calculated overall time taken by each of the framework in executing those unit tests.

The diagnostic tests included comparing the time of execution of assert statements in milliseconds.

After seeing the results, MS Test completely won over NUnit, however there wasn't significant time difference for 35 unit test functions.

Reports of the usability as well as performance are given in the next section.

8. Analysis Report

Based on the above test, we generated a report considering some usability and performance tests of NUnit as well as MSTest in integrated development environment.

Also, as mentioned earlier, the analysis was done based on two developers who had never written any unit tests before. These developers were very well acquainted with Visual Studio, though had never used the integrated unit testing framework. The wrapper for the performance was written to calculate the execution trace for both the frameworks.

Metric	MSTest	NUnit
Download/Installation	Easy	Easy
Setting up environment	Easy	Moderate
Avg. Time taken to write First Test case	4 mins.	8 mins.
Avg. Time taken to Run First unit Test Case	< 1 mins	4 mins.
Code Coverage Tool	Integrated	Third party
Documentation/ Internet forums	Excellent	Very good
Avg. Time taken to unit test a class with 5 methods (only assertions)	3 mins.	8 mins.
Avg. time taken to unit test about 35 functions (Performance Test)	9ms	17ms

Figure 11: Analysis report of MSTest and NUnit

On giving a closer look at the time taken to unit test few functions, we can conclude that it can't be the significant measure to compare between the two frameworks as developer may not be running unit test every time the code is compiled and for less number of unit test. Also, both the framework supports running unit tests asynchronously, so developers can simultaneously write code and unit test the compiled code.

9. Conclusion

Test-driven development enforces critical analysis of the requirements and design of software because the developer cannot create the production code without truly understanding what the desired output should be and how to test it.

By writing unit tests, a developer is actually writing a documentation which cannot go out of date unlike inline comments or separate documentation with the code.

Choosing unit testing framework is a very important step of Software Engineering. With integrated unit testing framework, developer tends to better understand the goal of testing as it makes his job easier to follow Test driven development model.

With enhanced set of features of these integrated environments software tends to be better designed, less coupled and easily maintainable [12]

Also, the integrated testing environment saves lot of development time which can not only increase the productivity but can also generate resources for better testing and creating a bug free software.

10. References

[1] Anthony Allowatt , Stephen Edwards, "IDE Support for test-driven development and automated grading in both Java and C++", *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, San Diego, California, October 16-17, 2005

[2] Cyrille Artho, Armin Biere, "Advanced unit testing: how to scale up a unit test framework*", International Conference on Software Engineering, Proceedings of the 2006 international workshop on Automation of software test*, April 2006, pp. 92 – 98

[3] Jian Zhang; Chen Xu; Cheung, S.-C.; , "Automatic generation of database instances for white-box testing ", *Computer Software and Applications Conference (COMPSAC)* , 2001, pp.161-165,

[4] Nikolai Tillmann, Jonathan De Halleux , "Pex: white box test generation for .NET", *Proceedings of the 2nd international conference on Tests and proofs, TAP 2008*, Prato, Italy, April 9-11, 2008, pp. 134 – 153

[5] Antonia Bertolino, "Software Testing Research: Achievements, Challenges, Dreams", *Future of Software Engineering*, May 23-25, 2007 , p.85-103,

[6] André Restivo , Ademar Aguiar, "Towards detecting and solving aspect conflicts and interferences using unit tests", *Proceedings of the 5th workshop on Software engineering properties of languages and aspect technologies*, Vancouver, British Columbia, Canada, March 2007, p.7-es

[7] Laurie Williams, Nachiappan Nagappan, E. Michael Maximilien, Thirumalesh Bhat, "Realizing quality improvement through test driven development: results and experiences of four industrial teams", *Empirical Software Engineering*, Vol. 13, No. 3. (1 June 2008), pp. 289-302.

[8] Amitabh Srivastava , Jay Thiagarajan, "Effectively prioritizing tests in development environment*", Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, Roma, Italy, July 22-24, 2002

[9] Stephen H. Edwards , Manuel A. Pérez-Quiñones, "Experiences using test-driven development with an automated grader", *Journal of Computing Sciences in Colleges*, v.22 n.3, January 2007, p.44-50,

[10] Williams L., Kudrjavets G, Nagappan N, "On the Effectiveness of Unit Test Automation at Microsoft", *Proceedings of the IEEE International Symposium on Software Reliability Engineering (ISSRE)*, IEEE, November 2009, pp.81-89

[11] Boris Beizer, "*Software Testing Techniques, 2nd Edition*", 1990

[12] Himanshu Arora, "Test Driven Development with integrated Microsoft Unit Testing Framework Environment", *Mid-term Paper, NCSU Software Engineering 2010*, Raleigh, North Carolina, USA, October 7 2010, pp. 1-6.

[13] MSDN, A Unit Testing Walkthrough with Visual Studio Team Test, Web, 21 November 2010 http://msdn.microsoft.com/en-us/library/ms379625(VS.80).aspx