A NEW UNIT TESTING FRAMEWORK FOR ENHANCED SOFTWARE DEVELOPMENT METHODOLOGIES

by
Çetin Yılmaz

Submitted to the Institute of Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science
in
Computer Engineering

Yeditepe University
2011

A NEW UNIT TESTING FRAMEWORK FOR ENHANCED SOFTWARE DEVELOPMENT METHODOLOGIES

APPROVED BY:

Assist. Prof. Dr. Birol Aygün (Supervisor) .....

Assist. Prof. Dr. Onur Demir .....

Assist. Prof. Dr. Sezer Gören Uğurdağ .....

DATE OF APPROVAL: .... /.... / ....

ACKNOWLEDGMENTS

I want to thank to my advisor Birol Aygün for his continuous support and patience during the study. Also I would like to thank to my wife Zeynep Yılmaz and my family for their encouragement.

ABSTRACT

A NEW UNIT TESTING FRAMEWORK FOR ENHANCED SOFTWARE DEVELOPMENT METHODOLOGIES

Unit testing is a method for validating that the source code of a program is working properly. In an ideal situation, each test case is independent from the others. Thus when applied to all parts of the code under development, it makes the whole code more reliable.

Although unit testing is used widely, there is no specialized unit testing tool for web-based applications. We developed a new unit testing framework called YUnit which facilitates mobile & web developer's work, which can measure the performance of source code and allows us to compare with previous test results.

The main goal of YUnit is demonstrating how, much of the work which people do manually during conventional testing, will be done automatically with the advanced unit testing tools, especially in web applications.

A brief survey on unit testing and unit testing frameworks, design, implementation and evaluation of entire YUnit system are presented in this thesis. In addition, suggestions to enhance the system are presented.

ÖZET

ÜST DÜZEY YAZILIM GELİŞTİRME TEKNİKLERİ İÇİN YENİ BİR BİRİM TEST ALTYAPISI

Birim test, bir programın kaynak kodunun düzgün çalışıp çalışmadığını doğrulamak için kullanılan bir yöntemdir. İdeal bir uygulamada, her bir test diğerlerinden bağımsız olarak tasarlanır.

Yaygın kullanım alanları olmasına rağmen belirli bir alanda özelleşmiş bir birim test aracı bulunmamaktadır. Özellikle web ve mobil tabanlı uygulama geliştiricilerin işlerini kolaylaştıran, aynı zamanda yazılan kod parçacıklarının performansını ölçebilen, istenildiğinde daha önceki test sonuçları ile karşılaştırma yapabilen YUnit adında yeni bir alt yapı sağlanmıştır.

YUnit ile amaçlanan, insanların ile yaptıkları çoğu işin, özellikle internet uygulamaları alanında, gelişmiş birim test araçları ile otomatik olarak yapılabileceğini göstermektir.

Bu tezde birim testleri ve araçları hakkında kısa bir araştırma ile birlikte YUnit sisteminin tasarım, uygulama ve değerlendirilmesi sunulmaktadır. Ayrıca, sistemi geliştirmek için önerilere yer verilmektedir.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

Unit testing is generally manual and rarely automated. In manual approach, unit test should be executed step-by-step by following an instructional document. However, the aim of unit testing is to isolate a unit and validate its correctness. Automation is efficient for achieving this, and enables the many benefits listed in this article. In an opposite manner, if not planned correctly, a manual unit test case may execute as an integration test case that involves many software components, and thus make impossible to achieve the goals established for unit testing.

Under the automated approach, to understand the effect of isolation, the source code subjected to the unit test is executed within a framework outside of its natural environment, that is, outside of the application or executing unit for which it was originally created. Testing in an isolated way has the benefit of making unnecessary dependencies between the source code and other parts of product. These dependencies can then be eliminated.

Using an automation framework, the developer codifies criteria into the test to verify the correctness of the unit. During execution of the test cases, the unit test application saves those that fail any criterion. Many applications will also report in a summary these failed test cases. The application may halt subsequent testing, depending upon the severity of a failure.

As a result, unit testing is helpful for programmers to create decoupled and robust source codes. This method increases healthy habits in software development. Refactoring, unit testing, and design patterns often work together so that the most ideal solution may emerge.

1.1. LIMITATIONS OF UNIT TESTING

Unit testing cannot catch every single error in the program, which is theoretically impossible. Intrinsically, it only tests the functionality of the program. Consequently, it cannot catch integration errors, which are outside of the program under test, performance

bottlenecks or any other system issues. Furthermore, it may not be easy to predict all special cases of input the program unit under study may receive in reality. Unit testing is more useful when it is used with other software testing activities.

It is not realistic to test all possible combinations of inputs and program states for any non-trivial part of program. Unit tests may only show the presence of possible errors, it cannot show the absence of errors.

1.2. GOALS OF THE PROJECT

The goals of my project are;

- Designing a new unit framework whose purpose is to check test cases.

- The framework will execute the cases one by one and checking the defined test objective which is an identified set of software features to be measured under specified conditions by comparing actual behavior with the required behavior described in the software documentation. Test cases are constructed to test that all units within assemblies, such as a partially compiled code library for use in deployment, versioning and security, interact correctly, for example, across procedure calls or process activations. This is done after testing individual units.

- Facilitating the mission of independent testers by designing a GUI (Graphical User Interface) based on our framework for white box testers (The test is accurate only if the tester knows what the program is supposed to do. User sees if the program diverges from its planned aim. In white box testing all source code must be readable.)

- Checking Test-Driven Development steps for written procedures [1].

We believe that unit testing is an important part of software development and encourages developers to write tests for their own code.

1.3. TECHNIQUES FOR UNIT TESTING

The basic steps of unit testing follow as;

1. Adding a hook to code by using any attribute.
2. Collecting the test cases from .net pre-compiled code (the interpreters or run-time packages are common or standard components of all platforms).
3. Perform the test planning
   a. Plan the general approach, resources, and schedule
   b. Determine features to be tested
   c. Refine the general approach
4. Acquire the test set
   a. Design the set of tests
   b. Implement the refined plan and design
5. Measure the test unit
   a. Execute the test procedures
   b. Check for termination
   c. Evaluate the test effort and unit
6. Classify the test results, where possible, with respect to the expected result.
7. Report the results.

1.4. THE BENEFITS OF UNIT TESTING

It is very difficult to write reliable software. In practice, it is not possible to write a complex piece of software having no bugs [2]. There are many obstacles to writing software with no bugs, especially since the definition of a bug may vary from case to case. Experienced programmers know that, a small mistake or missing a very simple point can create a major failure of the software.

Looking into future gives no sign of an easier job in writing software. With the development of software technology, more complex software will emerge. While the technology solves some of our problems, it will not give us a tool to overcome all obstacles

in the software writing process. However, we still have some possibilities to improve the process and enhance quality standards.

1.5. WHY UNIT TEST?

As unit test examines each unit of a system separately, it enables handling errors and helps in ensuring the correctness of the system. Here the unit refers to a class, package, interface, even a class hierarchy as used in object-oriented programming.

Unit tests give us the chance to control the correctness of a unit. A package of tests is subdivided into small components for easy testing. To reach a true understanding of a system and its functionality, a well-designed pack of unit tests should be used.

The complete set of unit tests can be a powerful tool that can be applied at any time by the programmers. To explain in a different fashion, it will be easier to fix bugs and reorganize the software as any changes which break the code will spontaneously make the tests to fail. Programmers will get feedback instantly as they run the tests. However, the structure of the tests may have to be changed as well.

1.6. AD-HOC TESTING

Adding new properties to software requires a phased approach. Whatever methodology is selected by the programmer, in most cases the developer will start with the requirements, then deal with the design, write the code, make debugging and will check the code into source code controlling system. On this step, properties are established. However, how does the programmer know that the code is working correctly?

The programmers and developers in small companies spend their all-time in the day to test the properties of the software, run the built solution and test the functionality with various inputs and their expected outputs. By finishing the testing, the software developer will be ensured about the correctness and the expected results. Still every developer ever being in process of coding to a software program has feelings of uncertainty.

For manual ad hoc testing, allocating developer time is costly. Instead of this, writing test software is more effective than ad-hoc testing and running for developer time. When developer tests every piece of component in the program, this kind of testing reveals whether the code influenced other properties and components or not. Furthermore, it is not ensured that the program functions properly during its lifespan even as new properties are added to the program.

1.7. MANUAL UNIT TESTING

Most of the time manual unit testing includes a debugging session within an IDE, using breakpoints and step-through debugging approach.

Benefits of unit testing;

- Current developer spots problems with ease.
- Solving problems and fixing bugs take a shorter time, as the code is fresh in the programmer's memory.

Drawbacks of unit testing;

- To perform the testing, the developer has to build any initial state manually. This is a difficult process which usually needs a lot of time, especially when the needed state is complex.
- As the developer do not have the chance to test all possible input/output combinations, errors may emerge despite all the time and efforts spent.
- The results of these tests are not traceable in any way.
- Tracing the code coverage is not possible in this testing type. This situation causes untested code.

- Integration of someone's testing with the testing done by the rest of team members is impossible. This situation causes the negative effects between someone's codes with other team members' codes.

- Since the programmers should apply this type of testing manually, it is impossible to construct a test package and make tests repeatable.

- During testing time, any external components that have been used should be cleaned up manually. These external components may be database records, mainframe records, temporary files or other artifacts.

1.8. AUTOMATED UNIT TESTS

Automated unit tests involve writing tests dynamically for the developer's code. The complete collection of these tests is called a suite. The developer can apply the suite of tests to the code in various ways, manual or automated.

Benefits of automated unit testing;

- Let's the developer construct a set of tests to apply them on the code at any time.

- Maintains syntax of the tests alive in mind by supplying understandable documentation for the code in the form of instances.

- Enables the easy way to ensure that new modifications in the code did not break the functionality of the code that has been already written.

- The complete set of tests can be applied to the code without any interference of users. This provides the property that testing can be applied during an external run, such as building process.

- Since the feedback is automatic and automated unit tests enable covering, modifications in the code are easier to do. When the testing does not succeed, modifications in any code which cause the code to be broken down will be known.

Drawbacks of automated unit testing;

- Automated unit tests are expensive in the startup.

- Although the advantages of automated unit tests can be encountered when a single programmer implements them, to provide full usability, the complete code base must contain automated tests. Further, all members of the software team should use the automated unit testing. In this way, full coverage of code is ensured for testing.

- Automated unit tests can be created and run only with help of some tools.

Unit tests are the lines of code that work on other pieces of code. Unit tests simply have two features: point unit tests work on small units such as methods. The second feature is end-to-end testing that works on a number of units, even the full program.

To implement a property, a programmer or developer must couple the writing of code with unit testing. This guarantees the proper functioning of the property. This is in practice writing code that invokes a test of that entry and exit point of the property and ensures the proper functioning of the code as expected. These tests are also known as pre-condition and post-condition tests [3].

When the property and unit testing are finished, the source code, containing the unit testing code, must be checked by the programmer to the source versioning control system. The programmer must be ensured about checking unit testing code into versioning control system. This situation enables integration of unit testing with the build. Moreover, the other members of the software team can also control the unit testing on their local machines and execute them at will.

As a rule, first a developer must have the last versions of unit tests from versioning control system on his or her local machine and execute all the tests, and then he/she can check his/her code segment into the source control repository. The developer checks in new code segment, after all unit tests pass. This order guarantees that the main code in the versioning control system maintains its state at all check times and can be reconstructed in the system at any time.

To obtain the full advantage of unit testing, the test pack must be executed as part of build and clean process. This refers to automated unit tests. The clean and build through the unit testing must be scheduled and well organized to execute automatically at least twice a day. If there are unit tests and they are not executed as part of a scheduled clean and build, then the developer will not get the full advantages of the testing. By executing the tests as parts of scheduled built solutions, the developer tests early and often. This guarantees the maintenance of the main code and repository. Sometimes the developer may write an unsuccessful code and check it into the main system by accident. The daily clean and build must be the main guarantee against these bugs. There are plenty of cheap tools to implement build solutions and test systems. How to implement a process that is built and cleaned is another major topic and resources on the websites can help the user in the beginning.

Handling unit tests is a very important in comparison with not writing at all, and this must not be neglected. In my experience writing and handling unit tests adds roughly 10-30% to the time it takes to complete a feature. End-to-end tests are likely to be the most time wasting type of tests to write. Then again, when writing and handling end-to-end tests one usually makes a small framework as the most used code in the tests is factored out. Hence writing end-to-end tests seems more effortless after the first few test cases have been written. Such a framework can be built easily by using YUnit, described later in this thesis.

Both types of tests - end-to-end and point tests are needed for the best covering. End-to-end tests encounter various kinds of problems as they work on a number of features that are involved in the implementing a component. Point unit tests are also necessary for the most significant features because they can ensure proper functioning of a feature fully.

1.9. BENEFITS OF AN AUTOMATED UNIT TEST SUITE

The automation provided by unit test packages creates many important advantages over other types of testing approaches.

The first advantage of the unit tests is that the problems are spotted in the early stages of the development cycle. The cost of fixing problems found in the early stages is less than fixing the problems found in the later stages of the development cycle. Automation of unit test package enables developers to find problems earlier, before the encounter with the customers and sometimes even earlier than the software reaches the QA team. By the time the programmer checks the code into source control many of the problems are spotted and handled.

The second advantage of the automated unit test package is that it scans your program in two dimensions; time and space. It ensures that the program functions properly now and in the future; it is the time dimension. The unit tests created for different properties ensures that your new program did not break them, which points out the second dimension that the unit test package watches over your program; the space dimension.

The third advantage is that the programmers will not hesitate to change the current code. As time passes by, the programmers normally become reluctant to change the code, as it gets riskier to do it because of the possible harm that may be caused while changing the code. A change in any part may affect other features of the program.

The fear of changing the code may be harmful. The overall quality of the program will decrease while the programmer fixes the bugs and adds new features. The original structure of the software may not allow the developer to add new features that may or may not fit. Considering the changing world we live in, it is impossible to foresee all future needs and also impossible to construct an architecture that will create the basis for features for these needs. In such a world the only way to keep creating new features and maintain the overall quality of the code is through refactoring. If occasional refactoring is not done, the code will lack quality, grow more and get tangled until every class knowing of every

other class. To work and clean on an existing code, refactoring is a scary process if you do not have automated unit tests.

Fourth advantage relates to the fact that the development process gets more flexible. Many times, the developer may find himself in a situation in which to fix a problem and to deploy it quickly is essential. A bug may escape from attention, which in turn will cause an important feature not to work properly. Imagine a situation where the customers cannot purchase software, the users cannot use the program and your boss wants you to solve the problem immediately. The problem about a quick fix is that it may have side-effects to the whole of the software. In such a situation, running the unit tests may come in handy as they should point out undesirable side-effects. A unit test suite saves us from applying and publishing hot fixes. Even if the developer is in need of a hot-fix, a unit test will help to improve a less problematic solution with lesser side-effects.

The fifth advantage is about your project's "truck factor" (number of people on your team who have to be hit with a truck before the project is in serious trouble) [4]. Truck factor is the minimum number of programmers, that if hit by a truck would crash the project. The higher the truck factor, the less risky your project is. If only one developer has the knowledge to carry operations on a project, then your truck factor is 1. This is a good example of a risky project.

An extensive unit test package increases truck factor because it makes it simpler for a programmer to gain understanding of a piece of code which he is not familiar with in the first place. With the right guidance from unit tests, the developer may work on a code created by others. It will make a smaller damage to lose a developer if you are applying unit tests as a safety net.

The sixth advantage of an automated unit test package is the reduced need for manual testing. Eliminating all of the manual testing is virtually impossible since humans have the talent to discover bugs with complex natures. It may take too much time to create a unit test for complex cases which is not a cost-effective way to deal with the problem. Assigning hard to find bugs to the QA team while the mundane testing will be covered by unit testing seems optimal.

Above listed benefits of unit testing on software development will become obvious and repeatable, as it is in a real engineering discipline. The artistic expression will still remain in the design and coding phases. After the coding is done, building process will shape and test the software similar to any product produced in an assembly line. This way, the ad-hoc nature in software development which is the cause for many problems, will be lessened.

1.10. UNIT TESTING IN THE REAL WORLD

In spite of the all the advantages that automated unit tests provide us, it has to be said that they are not the best way to avoid the view bugs in our software products for good. Feeling trust on unit tests as blind as "if unit tests pass, code is ready for production" is a recipe for shipping bugs. Even if we have an extensive unit test package that is executed often, we can still encounter bugs and an amount of manual testing by developers is necessary.

There were two types of bugs in the code; the original bug in the main source code and the other ones in the test code. Generally when this happens, it is according to particular situations in the source code being tested. The possibility for the two bugs occurring together is more seldom than for the first bug occurring alone.

The production area is different from the test area. For example, in a web application, there was a different version of a database driver in the production environment than in the test environment. A bug in a query that caused sensible results in the test environment led to a syntax error in the production environment. The certain consequence is that there must be compatibility between the test area and the real production area as closely as possible.

Whenever a bug leaks into the system, we turn back to unit tests and add a particular test case to ensure that the bug should be fixed, not avoided. In this kind of way the relation gets closer and closer on bugs and it gets harder and harder for the bugs to slip through.

Some developers may think that writing unit tests is a waste of time and requires effort. Often, this challenge occurs according to the fact that the developers think unit tests as like micro unit tests, e.g., unit tests that work on a get/set property of a class. There is no need to consume time to write test codes that test non-significant source code. It is cheaper to write a test code that tests a single proper functioning that would fail if the feature did not work as expected. With .NET it is very convenient to write a complete test code that tests get/set properties of a given class using reflection that it is really convenient to write it once and have it do the non-significant tests. Another typical challenge is that writing more developed unit tests needs ability, plenty of time and can be boring. To this I can only say that the unit test must be counted as a complete part of a property and writing well prepared unit tests is like writing any kind of program: it needs ability and expertise.

When interfaces in the software program are modified, unit tests that work on the source code will be broken down. This situation cannot be solved easily except by persuading the user that the advantages of the unit tests overweigh the additional effort in development and maintenance of the unit tests.

2. NOMENCLATURE OF UNIT TESTING

General unit testing terminology and existing frameworks are analyzed in this section.

2.1. DEFINITION OF UNIT TEST

A unit test examines a method or process in requirement environments. It can be performed as a “black box” test or a “white box” test as described below [5].

2.2. TYPES OF UNIT TEST

These test methods differs from others. Using what kind of unit testing can be determined from complexity of source code.

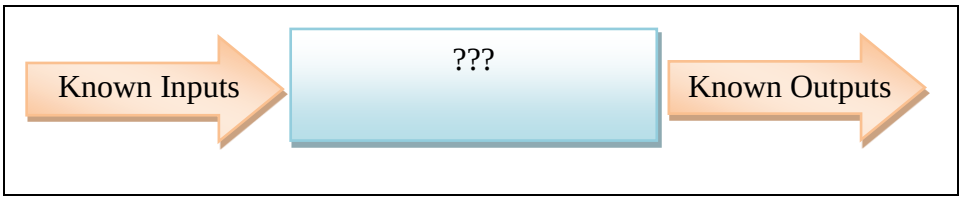

Figure 2.1. Black Box Testing [6]

2.2.1. Black Box Testing

Black box test verifies the obtained outputs with the given inputs without knowing the internal process. Because of this also it is known as functional testing; there is no information about:

- How error handling is applied.
- How the given inputs are processed in the code-paths.
- How the given inputs modified before execution.
- What dependencies are used in the box?

If you use black box testing, it limits the skills required of the tester directly, because if you do not know the business logic of the process, you cannot examine the all code pathways. Furthermore black box testing verifies the outputs when given good inputs.

Naturally in black box testing testers only access the public methods or properties of the classes.

2.2.2. White Box Testing

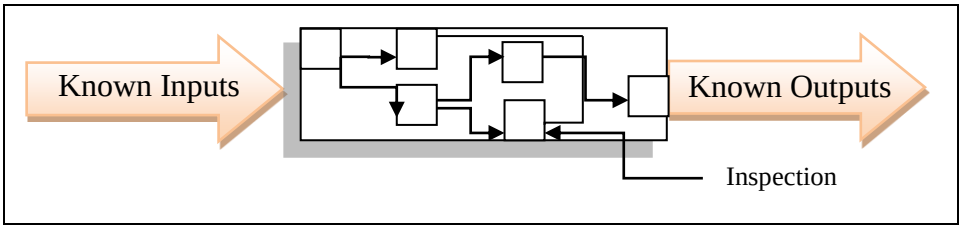

Figure 2.2. White Box Testing [6]

In white box testing we know the necessary information about the process pathways. In this method we give correct inputs and incorrect inputs to the test code but incorrect inputs handled from error handling. These are the advantages of the white box testing:

- We know the error handling mechanism.
- We can write test codes that examine the source code.
- We know the all dependencies of the business logic and the source code.
- Internal methods can be researchable.

In a good white box test we must ensure that all possible pathways are exercised, but achieving this is very difficult.

Researching the internal business logic or methods in the box of unit test is another advantage of the white box testing. This is useful internal information about the process and input state even if the output is verified.

2.3. WHAT IS TEST HARNESS?

A unit test includes an important object called "test fixture" or "test harness".

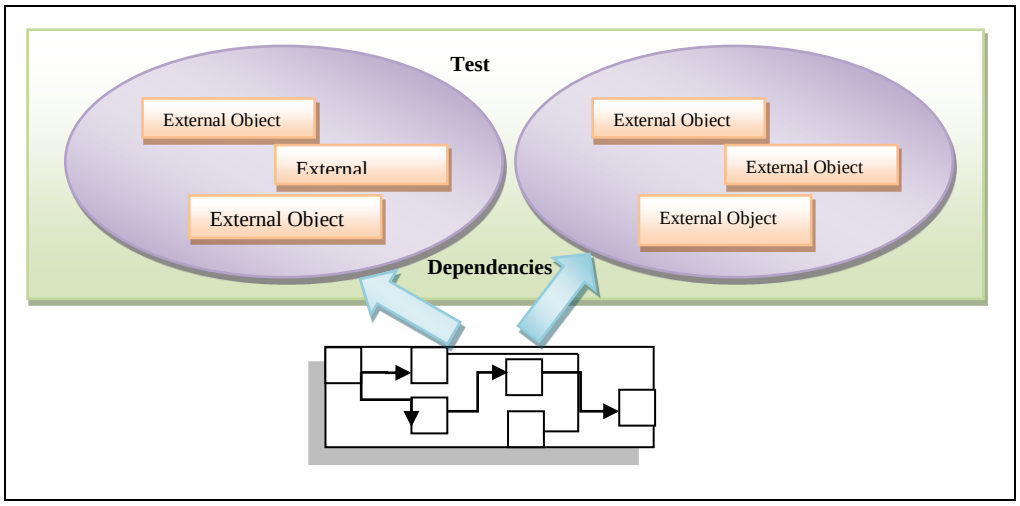

Figure 2.3. Test Harness [6]

A unit test fixture includes teardown and setup sections that the test case requires. This is required for initializing the given inputs like setting up a DB connection or other dependent objects. Real world examples may require more complicated teardown or setup processes and they are difficult to handle.

Generally the test fixture consists of two sets of teardown and setup activities:

- Required setup and teardown for suite of unit test.
- Required setup and teardown for running unit test.

Having multi-point setup and teardown for a suite of test cases is not good for performance reasons. The setup and teardown for each method or process is known to be efficient compared to single teardown and setup for a collection of objects.

2.4. MOCK OBJECTS IN UNIT TEST

We are using the mock objects to simulate complicated environment functionality and it simplifies to creating test fixtures. We need to mock object (mock objects are simulated objects that mimic the behavior of real objects in controlled ways) to simulate resources like network connections needed by the process unit test [7]. Another advantage of using mock object is performance gain because in production environment taking a new instance of an object is very painful and it decreases the performance of the test fixtures.

2.5. PERFORMANCE & VALIDATION TESTING

Writing quality code requires very strong and reliable rules when developing code. Performance requirements and validation of input controls varies according to business requirements and project specifications.

For example, in our project inserting a new customer should be less than 200 milliseconds. In a graphic processor program applying a new filter should require less memory than applied image size.

The Unit testing framework should help us to automate this process.

2.6. EXISTING FRAMEWORKS

There are several unit testing frameworks that are written for .Net Framework, See the complete list for these tools in appendix. csUnit, NUnit and Visual Studio Unit Testing Framework are the best known. These are commonly used for generic testing purposes. Despite numerous features, they are not useful for specific purposes. For example, there is no performance measurement option in these tools.

Table 2.1. List of .NET Unit Testing Frameworks [8]

Name	Remarks
csUnit	includes GUI, command line, VS2005 plug-in; supports C#, VB.NET, Managed C++, J#, other .NET languages; integrated with ReSharper
NUnit	includes GUI, command line, integrates into Visual Studio with ReSharper
NaturalSpec	Domain specific language for writing specifications in a natural language. Based on NUnit.
Visual T#	Visual T# is a unit testing framework and development environment integrated with Visual Studio. It includes T#, a programming language designed specifically to naturally express unit test intentions, and tools for compiling, running and maintaining them.
Visual Studio	The Visual Studio Unit Testing Framework is included with Visual Studio Team System 2005 and later editions, integrated with Visual Studio IDE. It is not included with Visual Studio Standard Edition or Visual Studio Express editions.
MSTest	A command-line tool for executing Visual Studio created unit tests outside of the Visual Studio IDE - not really a testing framework as it is a part of the Visual Studio Unit Testing Framework.
MbUnit	Extensible, model-based nUnit compatible framework.
QuickUnit.net	Implement unit tests without coding. Minimalist approach to test driven development.
xUnit.net	
Specter	Behavior Driven Development with an easy and readable syntax for writing specifications. Includes command line, optional integration with NUnit
NUnitAsp	Based on NUnit

Table 2.1. List of .NET Unit Testing Frameworks [8] (Continue)

NMate	Nunit Integration and Code Generation Add in for Microsoft Visual Studio
Roaster	NUnit based framework and tools for the .NET Compact Framework
Pex	Microsoft Research project providing White box testing for .NET, using the Z3 constraint solver to generate unit test input (rather than Fuzzing).

Nowadays, when we are developing an enterprise application, we want to check or measure some outputs periodically, I think a unit testing tool can automate this process to help us.

2.7. WEAKNESSES OF UNIT TEST FRAMEWORKS

As a result of my investigations I found several weaknesses of existing unit test frameworks.

Comparison table of Unit Testing Framework capabilities in Table 2.2 can be helpful for understanding the differences.

Table 2.2. Unit Testing Framework Capabilities

Feature	**YUnit**	**NUnit**	**MS Unit Testing**
Can usable mock objects	Yes	Yes	Yes
Usable with .Net:			
v1.0	Yes	No	Yes
v1.1	Yes	Yes	Yes
v2.0	Yes	Yes	Yes
v3.0	Yes	Yes	Yes
v3.5	Yes	Yes	Yes
v4.0	Yes	No	Yes
FixtureSetUp	Yes	Yes	Yes
FixtureTearDown	Yes	Yes	Yes
TestSetUp	Yes	Yes	Yes

Table 2.2. Unit Testing Framework Capabilities (Continue)

TestTearDown	Yes	Yes	Yes
Categorized test methods	Yes	Yes	No
Checking timeout of test method	Yes	Yes	No
MemoryLimit Feature	Yes	No	No
Iterate Feature	Yes	Yes	No
ObjectiveExecution Feature	Yes	No	No
Prior Feature	Yes	No	No
TestOrder Feature	Yes	Yes	No
URLExist Feature	Yes	No	No
URLHasContent Feature	Yes	No	No
FileExist Feature	Yes	No	No
FileHasContent Feature	Yes	No	No
Managing references from a configuration file	Yes	No	No
RecordSQL Feature	Yes	No	Yes
CheckSQL Feature	Yes	No	No
RegionCulture Feature	Yes	Yes	No
FrameworkVersion Feature	Yes	No	No
ValidateHTML Feature	Yes	No	No
ValidateCSS Feature	Yes	No	No
ValidateXHTML Feature	Yes	No	No
ValidateXML Feature	Yes	No	No
ValidateWML Feature	Yes	No	No
Assert.AreEqual assertion	Yes	No	Yes
Assert.IsTrue assertion	Yes	Yes	Yes
Assert.IsFalse assertion	Yes	No	Yes
Assert.IsNull assertion	Yes	Yes	Yes
Assert.IsNotNull assertion	Yes	No	Yes
Assert.Fail assertion	Yes	No	No
Checking ontology of parameters	Yes	No	No

2.7.1. Monitoring the Performance Outputs

Method implementers choose and implement from key-value pair list which uses 0-n search (with initial low and high values of 0 and n) [9]. This is an inefficient way to choose correct value. Despite this the most unit tests do not monitor the performance.

2.7.2. Measuring the Resource Utilization

Generally classes have several child classes. For this reason the each parent class should have a Dispose method to free up the used resources instead of waiting to run garbage collector. Unit testing does not deal with cleaning up the unreferenced memory.

2.7.3. Ranking Priorities

Changing the unit test does not make it obvious that code that ends up depending on this requirement also needs to be refactored. There really is not anything that can be done about this except to recognize this dependency and write unit tests for the "higher" objects that specifically test this requirement. Forward only test cases are difficult to monitor because a process that has higher priority should be completed before the test cases which have low level priority.

2.7.4. Focus Point

A Unit testing tool, which is written for general purposes, is not helpful for specific usage. For example, when we are writing a web application, we need some specific method and process in Windows or mobile applications.

Accordingly, I would expect that the unit testing tool which I am using for web applications should include some helpful features to automate the process which I am doing manual.

3. IMPLEMENTATION REVIEW

In the beginning of the project in order to produce quick solutions, I need a small and easy project; I researched a lot of frameworks, open source and commercial ones, and some parts of YUnit are affected from the Mark Clifton's article. [6] Especially parser functions like assembly parser and test fixture runner.

My project YUNIT covers some interesting topics:

- Loading a managed assembly
- Identifying the namespaces in the assemblies
- Identifying the classes in the namespaces
- Identifying the methods in the classes
- Identifying attributes for classes and methods
- Invoking methods using reflection
- The difference between using "MethodInfo" reflection and delegate reflection with regards to capturing exceptions
- Creating custom attributes with attribute parameters
- Creating a notification event

3.1. UNIT TEST ATTRIBUTE AND ASSERTION DEFINITIONS

A .NET assembly includes necessary definitions for implementing a unit test class. The attributes that are associated with a unit test class and its methods must are defined. In a similar manner, the assertions to be performed are defined. This assembly is the only assembly that needs to be referenced by a unit test assembly. Below are the definitions of some terms used in unit test frameworks.

- "TestFixture" attribute is applied to a class, describes the class that is going to be used a unit test.

- “Test” attribute is applied to a method, describes the method that is going to be invoked from a unit test.

- “SetUp” attribute is applied to a method, describes applied method that is going to be invoked before test method executed.

- “TearDown” attribute is applied to a method, describes applied method that is going to be invoked after test method executed.

- “ExpectedException” attribute is applied to a method, describes thrown exception which may be expected in program flow.

- “Ignore” attribute is applied to a method, describes thrown exception that should be ignored in program flow.

3.2. UNIT TEST CORE ENGINE

This component consists of two pieces:

- General assembly parsing functions, which extract out the classes and methods in an assembly and their attributes
- Unit test components, building test fixtures and running the tests.

A high level block diagram of the test apparatus can be illustrated as:

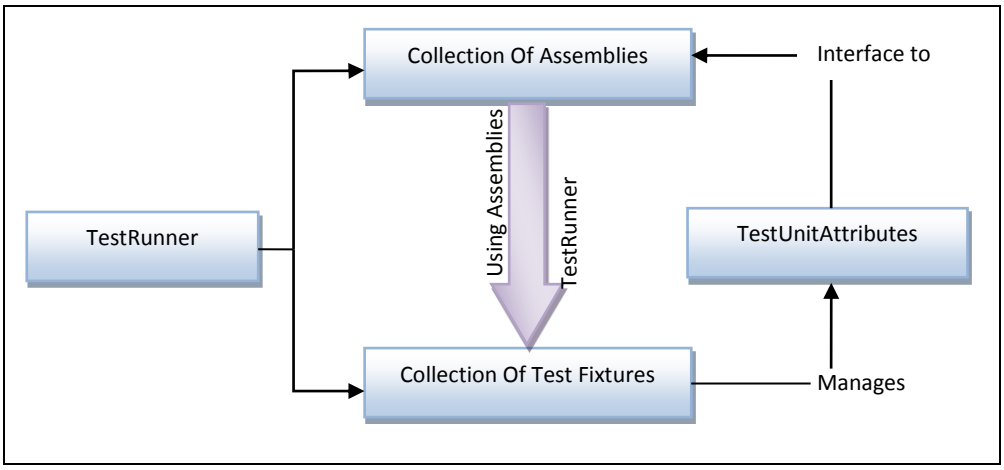

Figure 3.1. High level block diagram of the test apparatus [6]

Generally, I have tried to detach the assembly information from the unit test equipment's. In the YUNIT, collection of test fixtures and collection of assemblies are maintained by the "TestRunner" class. Using information in the assemblies, it creates the test fixtures. Test fixtures contain the information about the fixture like methods. Methods are associated with "TestFixtureAttribute" attribute; every attribute is associated with a class.

3.2.1. Running the Tests

A test fixture is representing by the "TestFixture" class. It handles the associated test fixture attribute, the setup and teardown methods to run for each test, and a list of tests to run. The main goal of the test fixture is to examine the tests; any exceptions are compared to expected exceptions. When failed assertions generate an exception, handled mechanism is stopped as well. The result state is represented by the test attribute and the notification event is triggered.

3.2.2. Invoking the Test Methods

"MethodItem" class runs the tests; we can write an event of the same type and fire the event. Handling the exception is easy; it is thrown to the test fixture which handles it.

3.3. UNIT TEST WINDOWS APPLICATION

The Window Forms application consists of three sections:

- A tree view showing all the unit test classes, their methods, and the specific test results.
- A progress bar providing the user with feedback as to the progress of the test cases.
- A report of test executions; represents the count of passed, ignored, and failed tests.

A tree component shows the test result in the assembly; failed tests have higher priority than ignored, passed tests have less priority than ignored tests.

As you can see GUI of unit testing tools in Figure 3.2, Figure 3.3 and Figure 3.4 are quiet simple and easy to use.

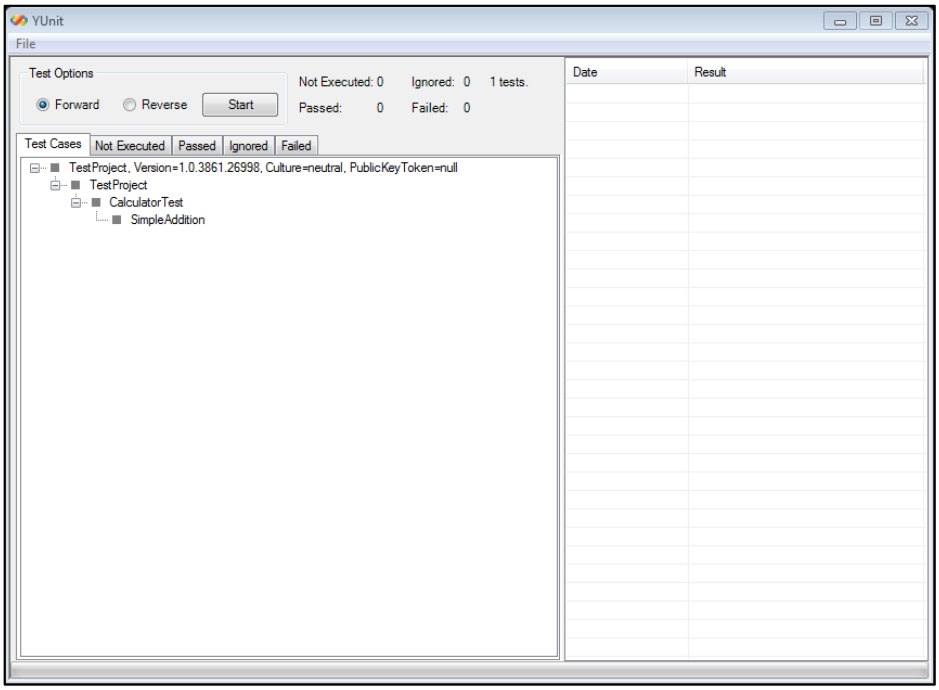

Figure 3.2. YUnit GUI: Our project

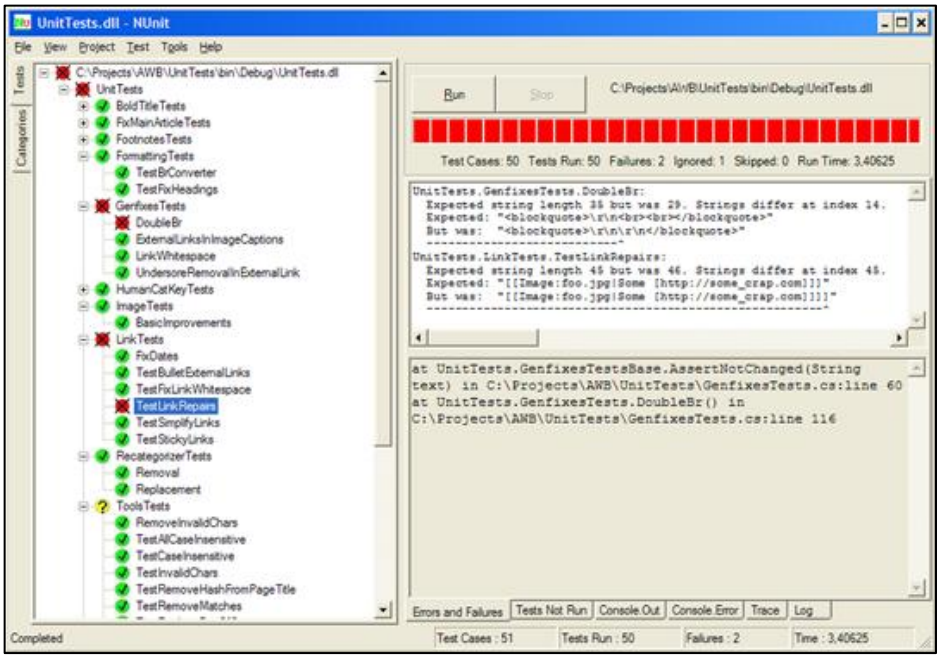

Figure 3.3. Nunit GUI

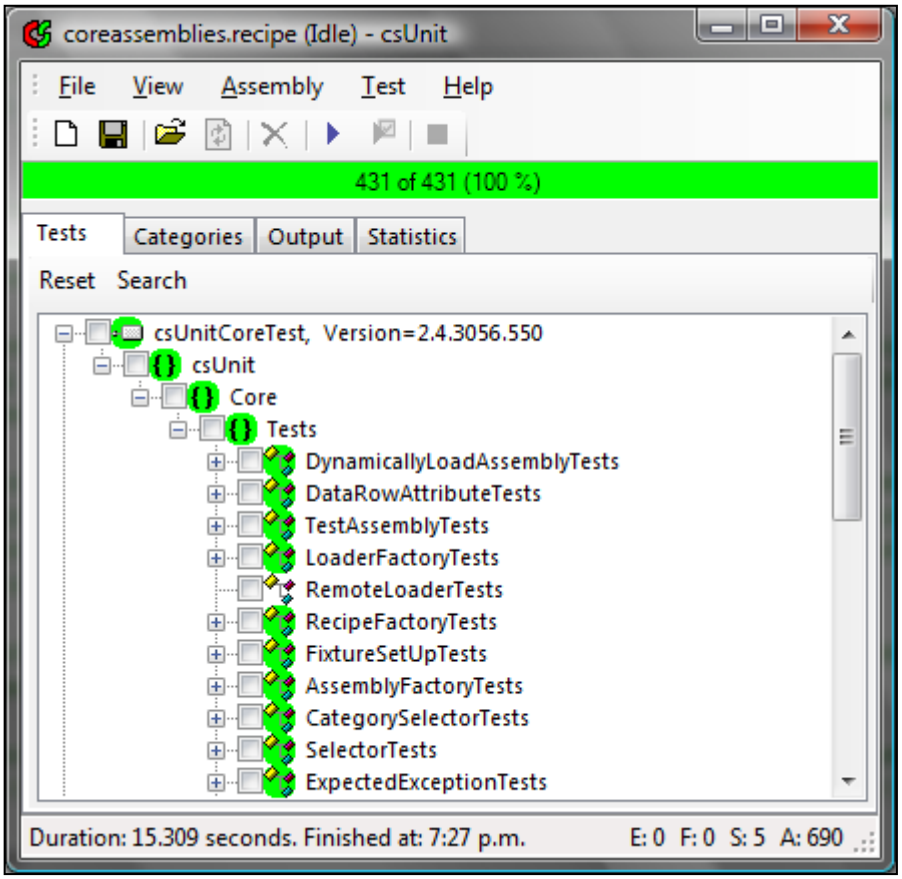

Figure 3.4. CsUnit GUI

3.3.1. The Test Notification Event

This event is used as a notification that a test has completed. This event updates the progress bar in the same GUI, changes the state of image which is associated in the tree view, and filters the image state of the tree.

In the tree view's image list, the TestAttribute.State enumeration is calculatedly designed to track the status of tests with icons in different colors:

- Gray icon → Untested,
- Green icon → Passed,
- Yellow icon → Ignored,
- Red icon → Failed.

The number of test results like passed-ignored-failed is calculated by this same index. Synchronizing the GUI to reflect these test counts is very simple.

3.3.2. Populating the Tree View

Populating the tree view consists of iterating through different collections:

- The assemblies,
- The namespaces in each assembly,
- The classes in each namespace,
- The methods in each class.

4. OVERCOMING THE WEAKNESSES

The chained unit tests generate a unit process. When an examination is finished then next one is started.

- A test fixture should be created as a process
- Test cases have their own order priority
- When a test case failure and not rune is shown as a output
- Test cases run forward and reverse.

4.1. SMART FEATURES

These features were developed to overcome the ranking priorities weaknesses which mentioned in section 2.7.3.

4.1.1. TestOrder Attribute

Test classes are ordered by the unit test creator who is generally the programmer. The previous unit test typically informs the each following test. Remaining unit tests not run when a previous unit test fails. This is displayed with a blue icon for each test not run.

"Out of order" means that a part of code is executed before another part of code. As a result the number of combinations that have to be analyzed is reduced, because if the code was run in its normal order, it may require many combinations. Thus, if a test can be run out of order we may not have to deal with total combinations because; we know that the ordered units of the process already succeeded! The reverse run process (running test cases in reverse order) is only in out of order. So, in a successful unit test I have to test two processes, reverse run and forward run.

Positive number can be given as order priority by using of this attribute. Test cases would be tested in given order. For example:

Algorithm 4.1. Sample test codes of using TestOrder attribute

```
[Test]
[TestOrder(1)]
public void FirstTest()
{
  string expected = "Test 1";
  string target = testString;
  Assert.IsTrue(target==expected, "Result: " + target + ", should
be: " + expected);
}

[Test]
[TestOrder(2)]
public void SecondTest()
{
  string expected = "Test 2";
  string target = testString;
  Assert.IsTrue(target==expected, "Result: " + target + ", should
be: " + expected);
}

[Test]
[TestOrder(3)]
public void ThirdTest()
{
  string expected = "Test 3";
  string target = testString;
  Assert.IsTrue(target==expected, "Result: " + target + ", should
be: " + expected);
}
```

In this example, test cases will be executed in order “FirstTest”, “SecondTest” and “ThirdTest”.

4.1.2. Prior Attribute

Some cases, we need to ensure that another test fixture is succeeded before current test case run. We can overcome this situation by using “Prior” attribute.

Algorithm 4.2. Sample usage of Prior attribute

```
[Test, TestOrder(4), Prior("RequiredConstructor")]
public void CustomInitialization()
```

In order to initialize a “CustomInitialization”, we shall require that “RequiredConstructor” be constructed first! Therefore, it is necessary to run the unit test before the constructor one. This is not describing how this attribute should be used, because this process is run in a forward direction. This attribute only guarantees that needed test cases exist and run successfully.

Algorithm 4.3. Sample test codes of using Prior attribute

```
[Test]
public void CheckList()
{
  int expected = 50;
  int target = stringList.Length;
  Assert.IsTrue(target==expected, "Result: " + target.ToString() +
 ", should be: " + expected.ToString());
}

[Test]
[Prior("CheckList")]
public void AssignationPerformance()
{
  int n=rnd.Next(0, 50);
  string target = stringList[n];
  Assert.IsTrue(target.StartsWith("Test"), "Result: " + target + "
, should be started with 'Test'");
}
```

According to this example; “AssignationPerformance” will be executed only if “CheckList” has been successfully already.

4.1.3. Category Attribute

In a real world we need to classify test methods for managing and executing them easily. For example, in a project that we have both data methods and business methods, if you want to examine only database operations, you will comment out or delete the other test methods. This is unnecessary for the test routine. To achieve this, we can classify the

test methods with “Category” attribute and execute test methods or exclude some other methods before executing the test fixture.

Algorithm 4.4. Sample test codes of using Category attribute

```
[TestFixture]
[Category("DATA")]
public class CategoryTest
{
    [Test]
    [Category("DATA")]
    public void DataTest()
    {
        Assert.IsTrue(true, "Pass.");
    }

    [Test]
    [Category("BLL")]
    public void BLLTest()
    {
        Assert.IsTrue(true, "Pass.");
    }

    [Test]
    public void SimpleTest()
    {
        Assert.IsTrue(true, "Pass.");
    }
}
```

In this kind of definition, only “Data” category test methods should be executed in a text fixture class. In below example;

Algorithm 4.5. Sample usage of Category attribute

```
[Category(“Data”, true)]
public class TestClass
```

All test methods should be executed except specific test methods with marked with “Data” category.

This useful attribute provides us to filter the test methods before executing each of them; the excluded methods will marked as ignored in the result pane.

4.2. PERFORMANCE FEATURES

These features were developed to overcome monitoring the performance outputs and measuring the resource utilization weaknesses which mentioned in section 2.7.1 and 2.7.2.

4.2.1. ObjectiveExecution Attribute

The unit tests are called in memory by using of delegates in an assembly. Therefore additional processes are loaded adding over the text fixture, because of the wrapper function calls. In an obvious manner, this has involuntary side-effects, specifically when the unit test itself executes in memory and/or time consuming code not part of the actual code under test. The optimum method to handle this situation is to implement a two stage test order; the first stage does the preparing and the second stage calling the method(s) under test.

Memory allocation of a method cannot be tracked in .net environments in a thread. There is no way doing this calling the GC.Collect() function; does not guarantee a correct value because the garbage collection runs on an internal CLR thread.

There are occasions in which are used to validate the processing time, because it varies so much depending on the machine. Processing time can be calculated for various purposes, such as:

- Detecting a highly inefficient algorithm;
- Assisting in detecting Quality of Service (QoS) problems;
- Meeting some baseline performance in a controlled environment.

Performance of a method varies from environment to environment, calculating the operation per second is very helpful, and some low-level code that has a major impact in the performance of an algorithm may be discovered this way.

The ObjectiveExecution attribute can be applied to any unit test to validate performance:

Algorithm 4.6. Sample test codes of using ObjectiveExecution attribute

```
[Test]
[ObjectiveExecution(15000)]
public void AssignationPerformance()
{
  int n=rnd.Next(0, 50);
  string target = stringList[n];
  Assert.IsTrue(target.StartsWith("Test"), "Result: " + target + "
, should be started with 'Test'");
}
```

This test code verifies that, minimum 15.000 operations are performed in a second.

4.2.2. Iterate Attribute

In a multi-tasking operating system, measuring the performance of function repeated a certain number of times are important. The Iterate attribute is used to iterate the test run for the given number of times. Additionally a wait can be inserted between two executions:

Algorithm 4.7. Sample test codes of using Iterate attribute

```
[Test]
[Iterate(50)]
public void ThirdTest()
{
  string expected = "Test 3";
  string target = testString;
  Assert.IsTrue(target==expected, "Result: " + target + ", should
be: " + expected);
}
```

The above code will run 50 times.

4.2.3. MemoryLimit Attribute

In garbage collecting environment there is no way to measure memory allocation directly. The .net environment does not provide us any way to monitor memory utilization because of this difficult situation.

Memory is classified as allocated or unallocated. Programmer needs to free allocations in real world.

But if you are in an environment that uses that uses garbage collection, you do not need free allocations. Memory is classified as referenced or unreferenced.

This situation is not same as allocated-unallocated one. In a memory, the garbage collector qualifies the memory in three ways: unreferenced-unallocated, unreferenced -allocated, referenced -allocated.

It is almost impossible to calculate memory usage for a time period in unreferenced-allocated part of physical memory. This part is allocated but waiting to be recovered by the garbage collector.

Some questions which may be asked are:

- How much memory remains referenced when a process runs?
- A function uses a reasonable amount of memory?
- Can garbage collector clean the used memory in an efficient way?
- A function properly disposes of unmanaged objects?

The garbage collector does not include true memory utilization functions. We only handle unmanaged resource by wrapping into managed resource or not. To achieve this situation, in example monitoring memory allocation we need a simple memory helper class. The MemoryLimit is used to specify the maximum amount of memory that a function is allowed to allocate on the process heap (not the GC pool) without failing.

Algorithm 4.8. Sample test codes of using MemoryLimit attribute

```
[Test]
[MemoryLimit(1000)]
public void MemoryTest()
{
  ImageLoader image=new ImageLoader();
  image.LoadImage("7tepe.jpg");
  image.Dispose();
}
```

Above code verifies that after the class has been loaded and disposed of, less than 1000Kb of memory remains allocated.

4.2.4. TimeOut Attribute

Validating the processing time is dubious because processing time varies so much depending on the machine, what it is doing and other technologies with which the unit tests are interfacing. However, this does not mean that testing the processing time of a function is without merit when used appropriately. Below optimal formula for calculating CPU time is;

Algorithm 4.9. Calculation of real execution time

```
Real secs = ExecutionTimeTicks/Timer.TicksPerSecond.
```

It is possible to measure performance metrics of a method or assembly with a unit test. We need to set a timer before execute the test method and measure after it finished, therefore we can abstract two time metrics and obtain the explicit value of the method performance. This is post execution assertion of the unit test engine, if the expected value of the performance is less then obtained value, processing test method will be failed manually otherwise this test method passes the assertion.

Algorithm 4.10. Sample test codes of using TimeOut attribute

```
[Test]
[TimeOut(100)]
public void TestMedium()
{
    Assert.IsTrue(true, "Pass.");
}
```

According the above example this test method should be completed in 100 milliseconds, otherwise test method will fail.

4.3. HELPFUL FEATURES FOR WEB APPLICATIONS

These features were developed to overcome absence of focus point in unit testing weaknesses which mentioned in section 2.7.4.

4.3.1. URLExist Attribute

While developing web applications that content is automatically generated from a data source especially if you are using URL rewrite methods, MVP or MVC methods you need to check a specific URL after business process finished.

For example, in a shopping site admin menu, after adding a new product to a category, we need to check product page will be generated properly from the browser. Technically, if we can see the page that means, http response of the URL returned “200 OK” code. According to this idea, we can check the specific given URL returned 200 or not. If web request response code is 200 then, assertion will pass otherwise fail.

Algorithm 4.11. Sample test codes of using URLExists attribute

```
[Test]
[URLExists("http://www.google.com.tr")]
public void CheckGoogle()
{
    Assert.IsTrue(true, "Pass.");
}
```

This is another post processing assertion attribute, in this example; Google URL if returns the “200 OK” code, assertion will pass otherwise it fails.

4.3.2. URLHasContent Attribute

Working with application whose content is automatically generated we need to check some content if generated runtime or not. For example, in a news portal when we add a new content, we want to check that; if content is properly generated in live time.

To overcome this bottleneck, we need a post process attribute, after a test method finished we should check the specific content is exist in the URL source code. If content is in the URL source; test method will pass otherwise will fail.

Algorithm 4.12. Sample test codes of using URLHasContent attribute

```
[URLHasContent("http://www.google.com.tr", "Google")]
public void CheckMark()
{
    Assert.IsTrue(true, "Pass.");
}
```

In this method if the source code of Google has “Google” this method will be resulted as pass otherwise will be failed.

4.3.3. FileExist Attribute

Especially; while working with document management system, sometimes we need to check a specified file or folder physically exist. For example, when upload a new report

to Sales Category; we want to ensure that, file physically uploaded to document management server. To perform this, after file operation was completed, we should check the output file is really existed on the target path. This is post process validation in a unit testing.

Algorithm 4.13. Sample test codes of using FileExists attribute

```
[Test]
[FileExists(@"C:\Windows\System32\drivers\etc\hosts")]
public void CheckHostFile()
{
    Assert.IsTrue(true, "Pass.");
}
```

After executing a process in this test method, if the file "C:\Windows\System32\drivers\etc\hosts" is physically exist test method will pass otherwise it will fail.

4.3.4. FileHasContent Attribute

In many applications we need to save some content into a file; for example, archiving previous year sales report. To ensure that report is saved properly, we should search or filter any content in the output file. To achieve this, we can check a keyword or specific content in the output file after the operation completed. This is another post process operation of the unit testing.

Algorithm 4.14. Sample test codes of using FileHasContent attribute

```
[Test]
[FileHasContent(@"C:\Windows\System32\drivers\etc\hosts", "127.0.0
.1")]
public void CheckLocalhost()
{
    Assert.IsTrue(true, "Pass.");
}
```

If "127.0.0.1" is in the "C:\Windows\System32\drivers\etc\hosts", test method will pass; otherwise will fail.

4.4. DATA FEATURES

4.4.1. RecordSQL Attribute

While developing a database oriented application we need to test some operations with realistic data. For example, “UpdateCustomer” method in a business module, requires real properties like Name, Surname, and Email etc.

Using real time data in a test method increases to quality of the test method.

Algorithm 4.15. Sample test codes of using RecordSQL attribute

```
[Test]
[RecordSQL("SELECT * FROM Course WHERE CourseID = 1045")]
public void SuccessRecordSQL()
{
    Assert.IsTrue(Record["Title"].Equals("Calculus"), "Pass.");
}
```

To achieve this, we need a test fixture class that inherited from ITestData interface, before executing the method, query the given record SQL into a DataRecord which can be used in the running test method. This solution is preprocess attribute of the unit testing and utilizes the using real time data in test method for any assertions.

4.4.2. CheckSQL Attribute

This is another useful attribute for database oriented applications. Sometimes we need to check some data in database or not. For example, after deleting passive customers in database, we should check any active customer still exists on the same data source.

Algorithm 4.16. Sample test codes of using CheckSQL attribute

```
[Test]
[CheckSQL("SELECT * FROM Course")]
public void SuccessCheckSQL()
{
    Assert.IsTrue(true, "Pass.");
}
```

This example demonstrates that, after dozen of business processes completed, if the specific SQL returned any record. This test method will pass otherwise it fails. This is another post process attribute of unit testing.

4.5. VALIDATION FEATURES

4.5.1. RegionCulture Attribute

In real world, we are developing applications with multi nationality feature. Nowadays, it is being an obligation in projects. During the application development we need to distinguish some rules for specific culture. For example, in Arabian culture, content texts should be in right aligned format. Same is true in test methods. Some procedure should be executed only specific culture.

Algorithm 4.17. Sample test codes of using RegionCulture attribute.

```
[Test]
[RegionCulture(RegionNames.tr_TR)]
public void TurkishTest()
{
    Assert.IsTrue(true, "Pass.");
}
```

According to above example; this test method should be tested only if the user culture is “tr-TR”, otherwise it will be ignored.

This is preprocess attribute of unit testing, before triggering the test method there is a simple check for the user's system settings. If the user's settings same with the specific culture, test method will be executed from unit test engine.

4.5.2. FrameworkVersion Attribute

Today current version of the Microsoft .Net Framework is 4.0, if the application was started previous version of the framework and updated to the newer version, we need to write some code to specific framework version. For example, reading any data from an application configuration file, provided with "ConfiguratonManager" class but in version 1.1 it has been provided with "System.Configuration" class. According to framework version that installed on client user machine, we need to change code of configuration reader procedure.

Algorithm 4.18. Sample test codes of using FrameworkVersion attribute

```
[Test]
[FrameworkVersion(NETFrameworkVersions.v4_0)]
public void WorkOnlyFramework40()
{
    Assert.IsTrue(true, "Pass.");
}
```

In this example this test method only executed if the framework version of user equals with specific version of .Net Framework, otherwise it will be ignored.

4.5.3. ValidateHTML Attribute

While working with dynamic content, we need to check our project if is valid or not for W3 standards before uploading if to deployment server. To automate this process we should check a URL validation after a test method finished. This is another post process attribute of unit testing.

Algorithm 4.19. Sample test codes of using ValidateHTML attribute.

```
[Test]
[ValidateHTML("http://www.google.com.tr/")]
public void GoogleTest()
{
    Assert.IsTrue(true, "Pass.");
}
```

After “GoogleTest” is executed, the HTML source code of “http://www.google.com.tr” is valid according to HTML 4.0 specification that are listed in http://www.w3.org/TR/1999/REC-html401-19991224/, this method will pass otherwise it will fail.

4.5.4. ValidateCSS Attribute

This is another helpful attribute for web designer, when we are developing visual design part of an application; we need to write Cascading Style Sheets that are generated in runtime. For the performance reason and to increase quality of code that have been written; output file should be valid for W3 standards according to specifications are listed in http://www.w3.org/TR/2008/REC-CSS2-20080411/.

Algorithm 4.20. Sample test codes of using ValidateCSS attribute

```
[ValidateCSS("http://voxelit.com/test.css")]
public void TestCSS()
{
    Assert.IsTrue(true, "Pass.");
}
```

After a test method is executed, we need to check source code of specific URL that is valid for CSS2.1 specifications, if it so test method will pass, otherwise will fail.

4.5.5. ValidateXHTML Attribute

Nowadays; designing web projects with XHTML has become standard especially in web 2.0. For the reason of structure of XHTML, we need XSD (XML Schema Definition)

file to validation. We define any constraint and rules into XSD file, and assign it to a XHTML file. By default XSD definition is in the first line of the XHTML file.

If any XHTML file is used this Schema Definition, all elements and attributes must be performed the rules that are defined in http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd.

Algorithm 4.21. Sample test codes of using ValidateHTML attribute.

```
[ValidateXHTML("http://voxelit.com/test.html")]
public void TestPage()
{
    Assert.IsTrue(true, "Pass.");
}
```

This is post process attribute in unit testing too, according to above example this test method will pass if source code of “http://voxelit.com/test.html” is valid for own XSD rules, otherwise it fails. Validation procedures are written in “System.XML” namespace.

4.5.6. ValidateXML Attribute

XML (Extensible Markup Language) is standard language of communication in WWW. For example, if you integrate your application with another application or system, output data and input values should be written in XML format. XML is very strong and easy for this action but if there is an error on the syntax, it does not work, it must be well-formed.

We need to check our XML file is valid or not before deploy it. Also C# language provides as useful parsing functions:

Algorithm 4.22. Sample source code of loading a XML file

```
XmlDocument xmlDoc = new XmlDocument();
xmlDoc.Load(url);
```

To automate this process we can use this simple code block; as it seems in below example, if “http://voxelit.com/test.xml” file has any syntax error, this test method will fail, but if not so, it will pass.

Algorithm 4.23. Sample test codes of using ValidateXML attribute

```
[Test]
[ValidateXML("http://voxelit.com/test.xml")]
public void TestXML()
{
    Assert.IsTrue(true, "Pass.");
}
```

4.5.7. ValidateWML Attribute

WML (Wireless Markup Language) is a XML based Markup Language and implemented with mobile devices such as mobile phones. If your project has dynamic content, it should be valid for WML standards which are declared in its own XSD file, before deploying the server.

I added new post process attribute to unit testing to automate this process, this all the same with ValidateXML attribute.

Algorithm 4.24. Sample test codes of using ValidateWML attribute

```
[Test]
[ValidateWML(@"D:\My Dropbox\Thesis\YUnit\YUnitGUI\test.wml")]
public void TestWML()
{
    Assert.IsTrue(true, "Pass.");
}
```

In this example, if source code of http://localhost/DemoApp/Test.wml file is valid according to own XSD like http://www.wapforum.org/DTD/wml_1.1.xml, test method is going to return pass, otherwise it fails.

4.6. ADDITIONAL ASSERTION METHODS

While writing unit test, we use assertion classes and methods for deciding individual condition is true or not. By default we have “Assert.IsTrue” method; this method will fail when condition is false; otherwise test method will pass.

Increasing the assertion possibilities is helpful for writing test methods. These are some new assertion methods;

- Assert.AreEqual(parameters[]): if any parameter is different in parameter list, test method is going to fail, otherwise it will pass.

Algorithm 4.25. Sample test codes of using Assert.AreEqual method

```
[Test]
public void AreEqualSuccess()
{
    Assert.AreEqual("Pass.", "X", "X", "X");
}

[Test]
public void AreEqualFail()
{
    Assert.AreEqual("Pass.", "X", "X", "Y");
}
```

- Assert.IsTrue(): if given condition is true than, test method will pass, otherwise it fails.

Algorithm 4.26. Sample test codes of using Assert.IsTrue method

```
[Test]
public void IsTrueSuccess()
{
    Assert.IsTrue(true, "Pass.");
}

[Test]
public void IsTrueFail()
{
    Assert.IsTrue(false, "Pass.");
}
```

- Assert.IsFalse(): This is opposite of the "Assert.IsTrue" method.

Algorithm 4.27. Sample test codes of using Assert.IsFalse method.

```
[Test]
public void IsFalseSuccess()
{
    Assert.IsFalse(false, "Pass.");
}

[Test]
public void IsFalseFail()
{
    Assert.IsFalse(true, "Pass.");
}
```

- Assert.IsNull(): If given parameter is null then, test method will pass otherwise it fails.

Algorithm 4.28. Sample test codes of using Assert.IsNull method

```
[Test]
public void IsNullSuccess()
{
    Assert.IsNull(null, "Pass.");
}

[Test]
public void IsNullFail()
{
    Assert.IsNull(false, "Pass.");
}
```

- Assert.IsNotNull(). This is opposite of the "Assert.IsNull" method.

Algorithm 4.29. Sample test codes of using Assert.IsNotNull method

```
[Test]
public void IsNotNullSuccess()
{
    Assert.IsNotNull(false, "Pass.");
}

[Test]
public void IsNotNullFail()
{
    Assert.IsNotNull(null, "Pass.");
}
```

- Assert.Fail(): In some cases, while writing test methods, we need to change test results manually, When this method called, test method will fail with "forced manually" message.

Algorithm 4.30. Sample test codes of using Assert.Fail method

```
[Test]
public void FailTest()
{
    Assert.Fail("Manually failed.");
    Assert.IsTrue(true, "Pass.");
}
```

4.7. STORING CONFIGURATIONS IN A FILE

In an application we need to store some constants like connection strings, user defined settings, etc. in a configuration file. Microsoft .NET Framework provides us a settings file called “app.config” to do this action. Generally we use this file to save specific information about .Net assembly file.

We can use this kind of file for unit test assembly, for this purpose I add a new feature called assembly configuration file. Now we can add constants to a file called “app.config“. For example, if you want to use “RecordSQL” attribute, you should store database connection string into this file and you can change the settings when you want, without changing a single line of code.

4.8. ONTOLOGY OF PARAMETERS

In computer science and information science, ontology is a formal representation of the knowledge by a set of concepts within a domain and the relationships between those concepts. It is used to reason about the properties of that domain, and may be used to describe the domain. In unit testing we can use ontology for constraint purpose [10]. We can define some rules to parameters for example, checking the given URL is in valid format or not. If all parameters are given in expected format, test process will start, otherwise it will fail.

Algorithm 4.31. Sample test codes of using Ontology of Parameters

```
[Test]
[FileExists("foo")]
public void IncorrectFormatForFileExists()
{
    Assert.IsTrue(true, "Pass.");
}
```

In this example, test method will be failed because “FileExists” attribute accepts file path in a correct format but “foo” value is not a valid file path.

4.8.1. Setting attributes with RDF syntax.

The Web Ontology Language (OWL) is a family of knowledge representation languages for authoring ontologies. The languages are characterized by formal semantics and RDF/XML-based serializations for the Semantic Web. OWL is endorsed by the World Wide Web Consortium (W3C) and has attracted academic, medical and commercial interest [11].

According to Holger’s article, these standards provide a technical infrastructure for software developers who have little guidance on how to build real-world semantic applications [12].

In the light of this information, I add a feature called RDF for importing all parameter values through a single attribute or file. Using RDF syntax or file for passing attribute values to test methods is very helpful.

Algorithm 4.32. Passing value of TimeOut attribute with RDF syntax

```
[Test]
[RDF("<rdf:RDF xmlns:rdf=\"http://www.w3.org/1999/02/22-rdf-
syntax-
ns#\" xmlns:ex=\"http://www.example.org/\"><rdf:Description rdf:ab
out=\"http://www.example.org/\"><ex:TimeOut>10</ex:TimeOut></rdf:D
escription></rdf:RDF>")]
public void InternalRDFTest()
{
    Assert.IsTrue(true, "Pass.");
}
```

Algorithm 4.33. Passing value of TimeOut attribute with .RDF file

```
[Test]
[RDF(@"D:\My Dropbox\Thesis\YUnit\YUnitGUI\test.rdf")]
public void ExternalRDFTest()
{
    Assert.IsTrue(true, "Pass.");
}
```

In Algorithm 4.32., sample the value of “TimeOut” attribute was given via RDF syntax otherwise in Algorithm 4.33., sample attribute parameters are given via .rdf file format. This is a sample of setting test parameters with an external file.

In the example below, full content of test.rdf file can be viewed.

Algorithm 4.34. Full content of RDF file

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-
ns#" xmlns:ex="http://www.example.org/">
  <rdf:Description rdf:about="http://www.example.org/">
    <ex:ObjectiveExecution>150</ex:ObjectiveExecution>
    <ex:Iterate>10,2</ex:Iterate>
    <ex:MemoryLimit>1000</ex:MemoryLimit>
    <ex:TestOrder>2</ex:TestOrder>
    <ex:Prior>TestTimeout2</ex:Prior>
    <ex:Ignore>reason</ex:Ignore>
    <ex:Category>TEST</ex:Category>
    <ex:TimeOut>1000</ex:TimeOut>
    <ex:ComparePrevious>true</ex:ComparePrevious>
    <ex:URLExists>http://www.g.com</ex:URLExists>
    <ex:URLHasContent>http://www.google.com,xx</ex:URLHasContent>
    <ex:FileExists>c:\xx.txt</ex:FileExists>
    <ex:FileHasContent>C:\Windows\System32\drivers\etc\hosts,conte
nt</ex:FileHasContent>
    <ex:RegionCulture>en-US</ex:RegionCulture>
    <ex:FrameworkVersion>3.0</ex:FrameworkVersion>
    <ex:ValidateHTML>http://google.com</ex:ValidateHTML>
    <ex:ValidateCSS>http://google.com</ex:ValidateCSS>
    <ex:ValidateXHTML>http://google.com</ex:ValidateXHTML>
    <ex:ValidateXML>http://google.com</ex:ValidateXML>
    <ex:ValidateWML>D:\My Dropbox\Thesis\YUnit\YUnitGUI\test-
invalid.wml</ex:ValidateWML>
    <ex:CheckSQL>SELECT * FROM Course</ex:CheckSQL>
    <ex:RecordSQL>SELECT * FROM Course WHERE CourseID = 1045</ex:R
ecordSQL>
  </rdf:Description>
</rdf:RDF>
```

All parameters can be set by using of .rdf file, according to this feature, no need to build source code of test project before examining the methods with different parameters, just change and save the .rdf file before new execution.

4.8.2. Exporting attributes to RDF format.

In the near future, programs which use semantic definitions will be even more widespread. In keeping write to this idea exporting attributes to a file is being more elegant for integrating with other programs. In YUnit we can do this by using “Export as RDF” button in “Test Detail” window.

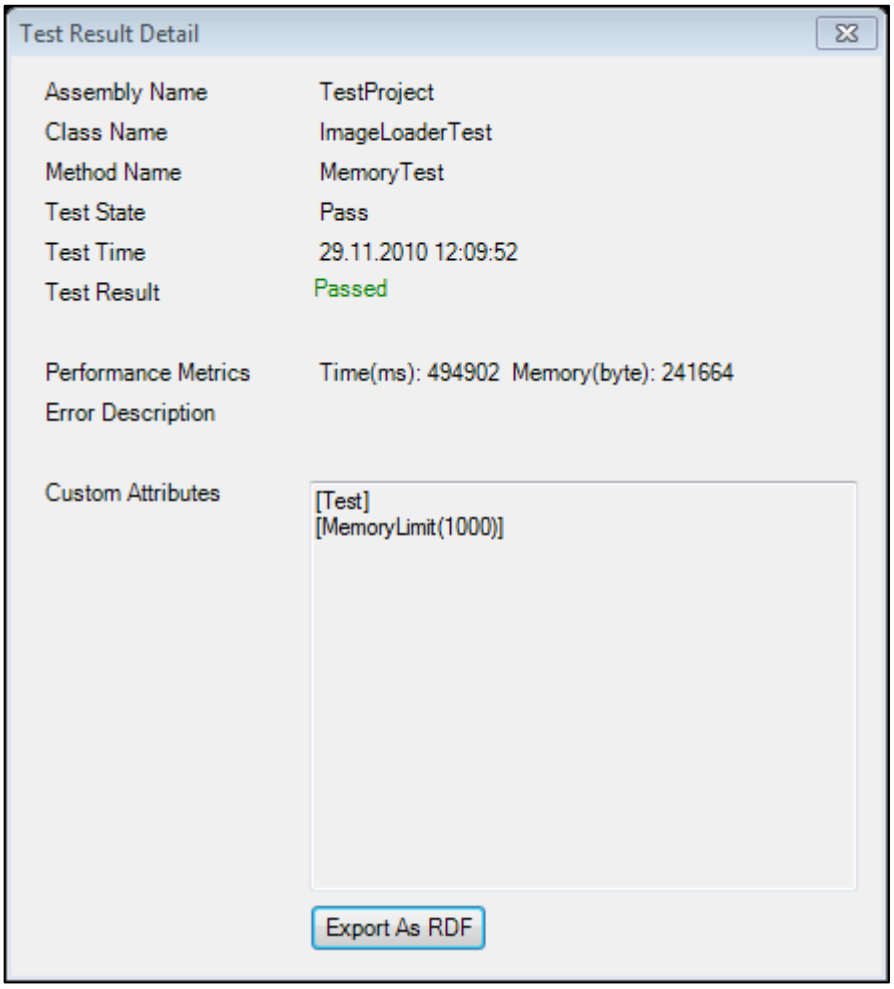

Figure 4.1. Screenshot for showing "Export as RDF" button

When that button is pressed, a folder browsing dialog will appear. After choosing a folder, individual test result is going to be exported. This is an example of an exported test result.

Algorithm 4.35. Example test result which exported in .rdf output format

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-
ns#" xmlns:ex="http://www.example.org/">
  <rdf:Description rdf:about="http://www.example.org/">
    <ex:AssemblyName>TestProject</ex:AssemblyName>
    <ex:ClassName>ImageLoaderTest</ex:ClassName>
    <ex:MethodName>MemoryTest</ex:MethodName>
    <ex:TestState>Pass</ex:TestState>
    <ex:TestResult></ex:TestResult>
    <ex:TestDate>29.11.2010 12:09:52</ex:TestDate>
    <ex:ErrorMessage></ex:ErrorMessage>
    <ex:MemoryUsage>241664</ex:MemoryUsage>
    <ex:ExecutionTime>494902</ex:ExecutionTime>
    <ex:TestTimeAsSecond>0,2005405</ex:TestTimeAsSecond>
    <ex:MemoryLimit>1000</ex:MemoryLimit>
  </rdf:Description>
</rdf:RDF>
```

4.8.3. ComparePrevious Attribute

During the development process comparing test results can be helpful for reaching the expected results. Especially, while working with performance metrics, we need to check current values with previous ones to show risk map.

A Risk Map allows you to instantly see the biggest risks facing a project. It is a graphical tool and as such is a great mechanism to show the team the risks [13].

For this purpose, we can do this by using “ComparePrevious” attribute.

Algorithm 4.36. Sample test codes of using ComparePrevious attribute

```
[Test]
[ComparePrevious(true)]
public void TestTimeout()
{
    Assert.IsTrue(true, "Pass.");
}
```

According to this example, if previous test result of timeout value is less than the current timeout value than, this test method is going to be failed, and otherwise it succeeds.

This attribute can be used with other performance metrics like “MemoryLimit”, “TimeOut” etc. Comparing test results with each other accelerates the development process to obtain expected goals.

4.9. TEST RESULT KNOWLEDGE BASE

Having a knowledge base is a great advantage when retest is required. Test results give us important clues to write successful codes. For this purpose, I added a knowledge database to store test case results. We can query or report it to view process results when any code changes.

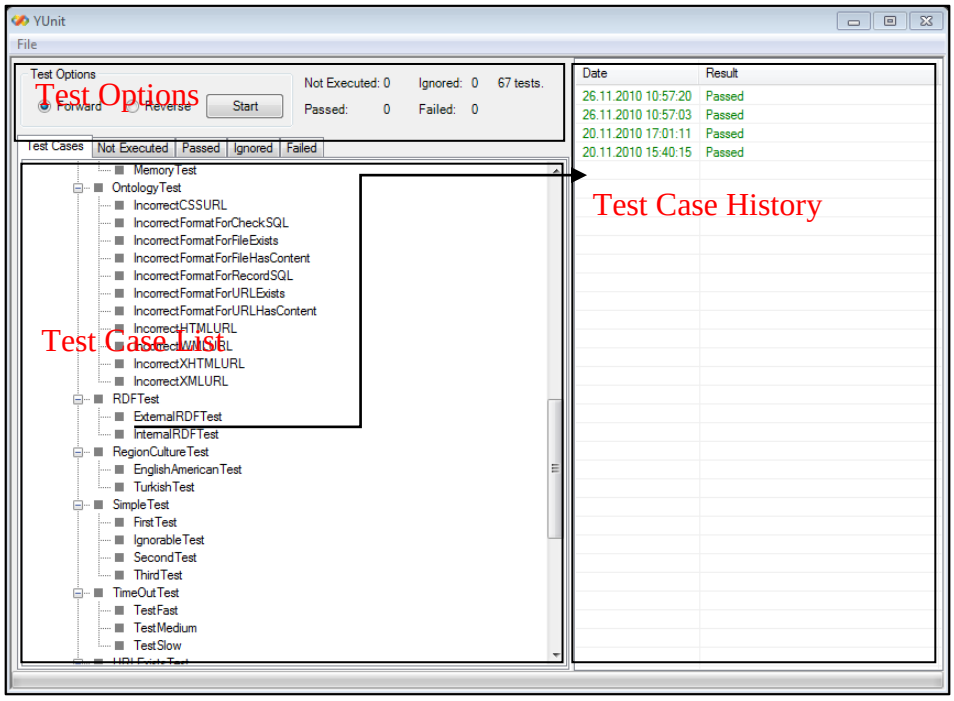

Figure 4.2. The YUnit GUI consists of three basic parts

4.9.1. Test Options

This part has options for testing direction and test case statistics for selected assembly. Forward and Reverse options define the ways of examination process, if “Forward” option is selected, YUnit starts to test from beginning to end. When “Reverse” option selected is starts to test from end to beginning. It changes the order of test methods.

The displayed statistics for an assembly are;

- The number of total test cases
- The number of passed test cases
- The number of not executed test cases
- The number of ignored test cases
- The number of failed test cases

When an assembly is selected for testing from “File > Open” step, test cases will be found and listed to the tree called “Test Case List”. Then just select the direction and click the “Start” button. See Figure 4.2.

4.9.2. Test Case List

When an assembly is selected to test, all test cases in the assembly will be listed to a tree. The gray color icon means that individual case has not been tested yet, the red icon says us the individual case tested but failed because of an error and the yellow icon means the test case is ignored for the process.

Test cases are grouped before listing in the tree view, the first level nodes displays the assembly, the next level displays the project, the next level displays the class and the other level displays the test methods. If you click any test method, the previous result about this test method will be listed in the history area.

When process is completed for the assembly, test results will be grouped in some other tabs.

- Test Cases, displays all test cases.
- Not Executed, displays not executed test cases.
- Passed, displays executed and passed test cases.
- Ignored, displays ignored test cases.
- Failed, displays executed but failed test cases.

Several useful information about test cases can be viewed in this part, such as fixture name, test method name, summary of error description, memory usage summary and CPU usage summary.

4.9.3. Test Case History

When you click on a test result from “Test Case List” pane, the history of this test case results will be displayed in this pane with two columns, date and result. The full details of test result will be displayed on a window, when double clicked on a history.

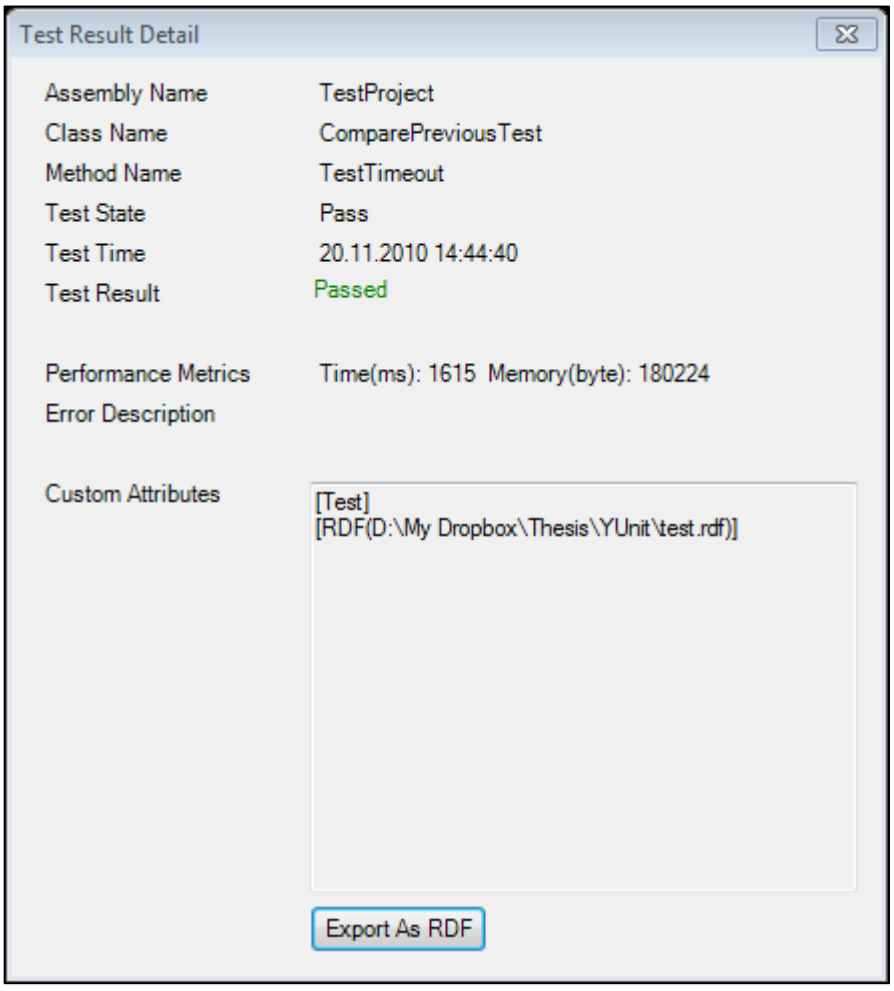

Figure 4.3. Screenshot of Test Result Detail window

This window shows the full detail of a test result, all given inputs and test results can be compared by using of this information.

We can analyze the improvements step by step when any source code is changed by using YUnit GUI.

4.9.4. Querying the Test Results

All test results are stored in a database in YUnit; this information can be used during the development time. Below are the attributes of a test result.

	Column Name	Data Type	Allow Nulls
🔑	ResultId	int	☐
	AssemblyName	varchar(255)	☐
	ClassName	varchar(255)	☐
	MethodName	varchar(255)	☐
	TestState	varchar(255)	☐
	TestResult	varchar(255)	☑
	TestDate	datetime	☐
	ErrorMessage	ntext	☑
	AttributeValues	ntext	☑
	MemoryUsage	int	☑
	TestTime	nvarchar(50)	☑

Figure 4.4. Description of Knowledgebase data model

- “ResultId” field is primary key of the “TestResult” table.
- “AssemblyName” field holds the assembly name of test case.
- “ClassName” field holds the class name of test case.
- “MethodName” field holds the test case name.
- “TestState” field stores the state of the executed test case. Pass, Fail, Ignored etc.
- “TestResult” field holds the summary of test result.
- “TestDate” field stores execution time of test case.
- “ErrorMessage” field holds the detailed error description when test case failed.
- “AttributeValues” field holds the attribute values of inputs before testing process.
- “MemoryUsage” field holds the memory statistics of test case, before and after.
- “TestTime” field stores CPU statistics of test case.

Stored test results can be queried from “File > Search Result” step. This window search the any test case result from the knowledge base. Search criteria are “Assembly Name”, “Class Name”, “Method Name”, “Test State” and “Error Message”. After any filter is given than click the “Start Search” button for the retrieving data. On the result pane, double click on the any result to see detailed view.

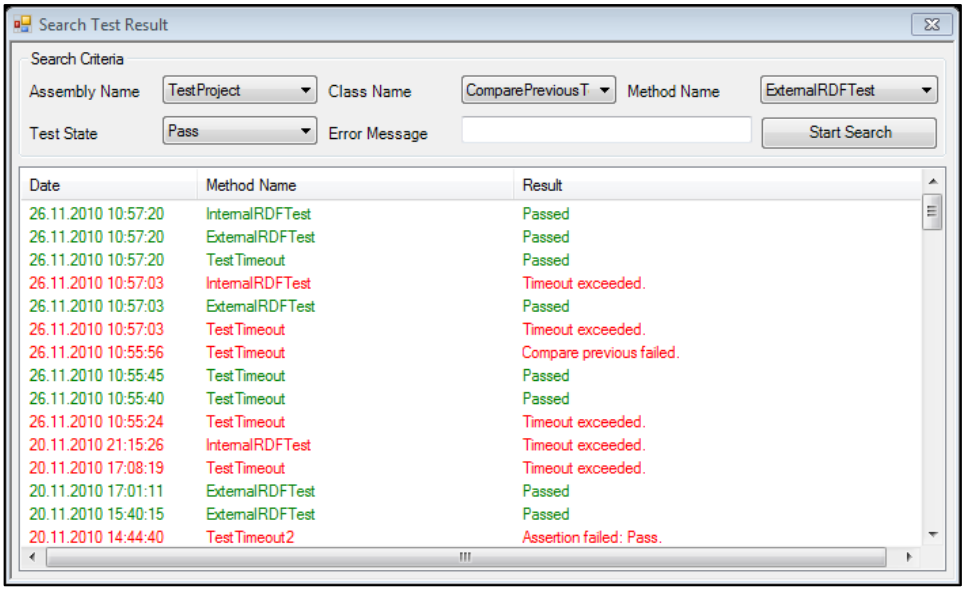

Figure 4.5. Screenshot of Search Test Result window

Reports can give us some critical information about the projects; also YUnit GUI has two reports to analyze test case results. The first one reports the results by test method. We can analyze the distribution of test cases in a test project. If anyone has great portion with clear difference, we think that this test case has a lot of business logics or performance issues. See Figure 4.6.

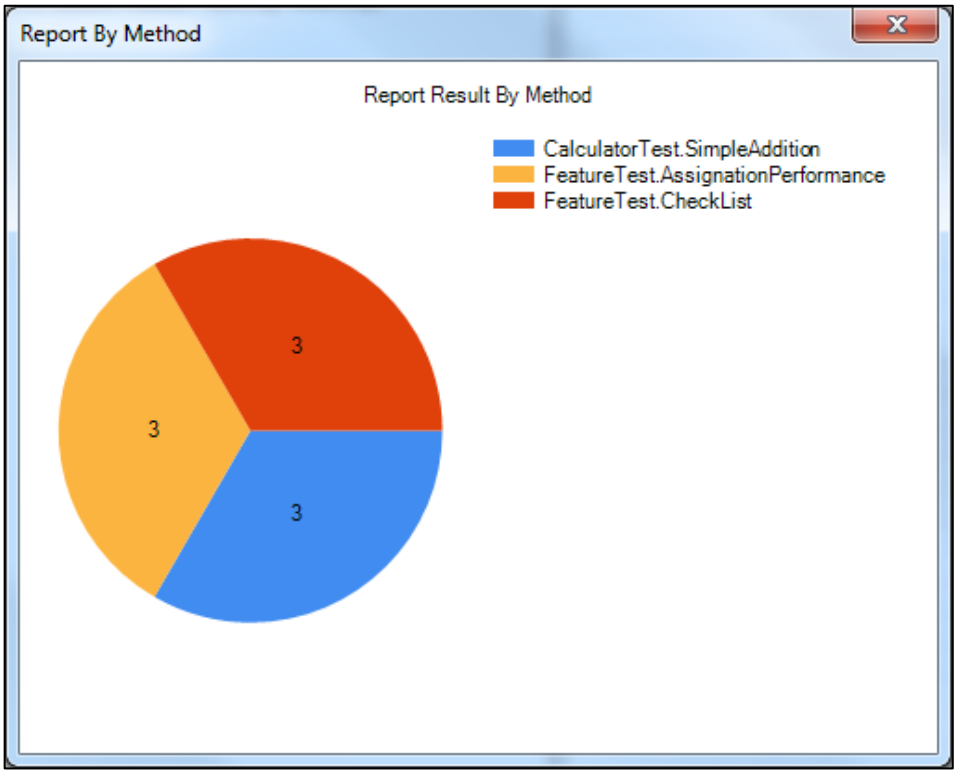

Figure 4.6. Screenshot of Report by Method window

This report is accessible from the “File > Report by Method” menu.

The main purpose of the test project is ensuring all the tests are successful. From time to time; during the development process, we would like to receive information about the course of the project. Seeing the rate of the test results helps us to calculate remaining time of the project. For this purpose we can use the “Report Result by Test State” report in YUnit. See Figure 4.7.

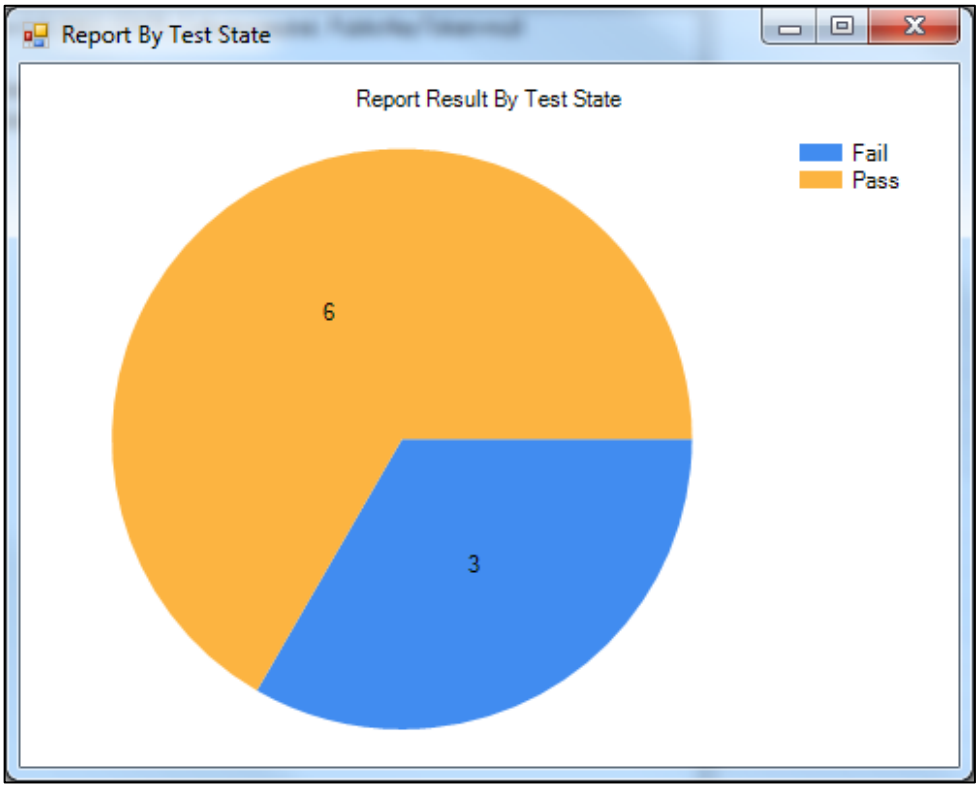

Figure 4.7. Screenshot of Report by Test State window

This report is accessible from the “File > Report by Test State” menu.

You may automate the test process via command line arguments with YUnit, command line tools future provides us to integrate with some other development IDE like Microsoft Visual Studio.

In Visual Studio, to open a project which will be integrated with YUnit, select “Properties” section using right clicks on the individual project. Select the “Build Events” section from the Property tabs. Then enter “YUnitGUI.exe %output_file%” line to “Post build event command lines” area. See Figure 4.8 for an example.

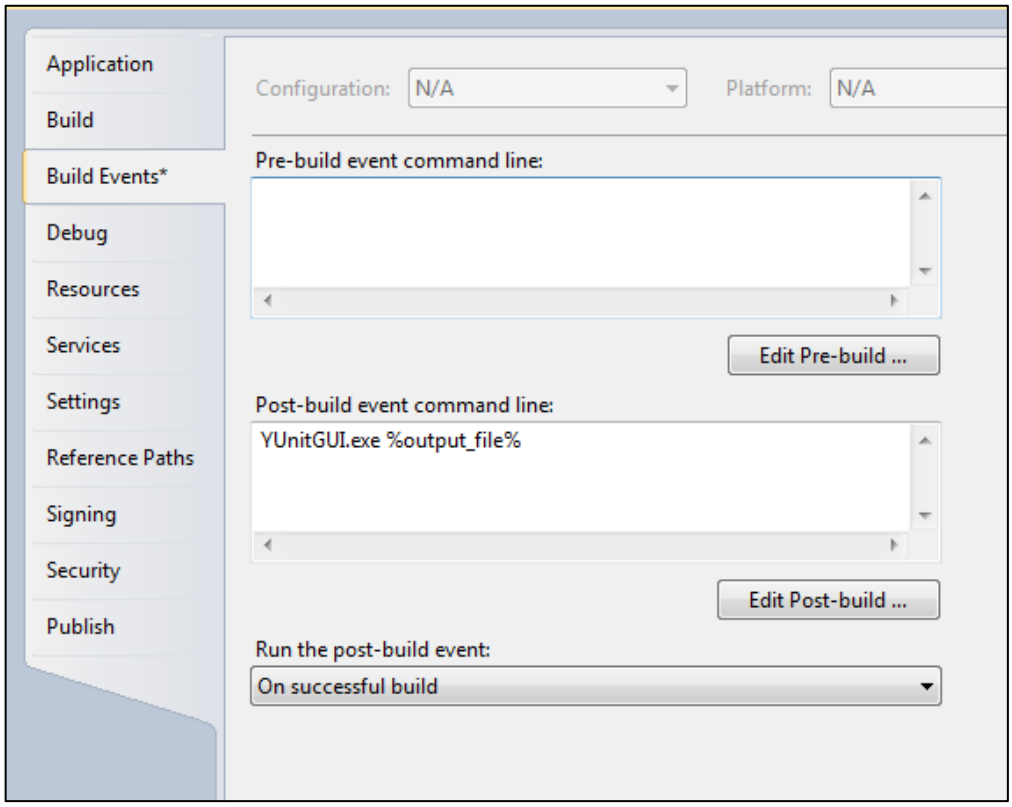

Figure 4.8. Screenshot of setting Post-build event command line

This command line integration feature will be used with a Windows .bat file to start test cases automatically. During the any phases of project development cycle, test results can be analyzed by running this command line file. It is easy to integrate and examine projects with YUnit.

5. FUTURE WORK

Unit testing has a wide range of uses in software development. The main idea is automating some of the required manual work. It can be extended for many purposes. In this study, I expand it for performance measurement and validation for web and mobile applications.

Calculating the tested code ratio is helpful before generating the risk map of the project. It tells us how much of the code or modules have been tested (or retested) before deploying alpha release or functional test.

Generating an error dependency diagram (A dependency diagram is a visual representation of a dependency graph; in the case of a dependency graph without circular dependencies) and visualizing it can be helpful for analyzing the source of errors and preventing the probable bugs [14].

We are using code generators for generating draft code from some sources like tables, views or stored procedures, before starting coding. Similarly we can generate some draft test cases from functions or procedures from precompiled code. It can be useful we want to write an extensive set of test cases.

In this study, I made a database for storing errors with some details like assembly-class-method name, error message and performance values. This information can be used for solving the chronic bugs in the project or may be used to prevent some bugs while building source codes.

6. CONCLUSION

According a report, one hundred and forty billion dollars each year in the U.S. software industry and software manpower going to waste and that programmer on the average make a mistake in one of every ten lines of code [15].

The testing process requires a skilled workforce and sufficient test time often is not included in the software project plan. Using unit testing is a good way to reduce waste in software development process.

Unit testing helps us find errors in our code; it can be seen as a seat belt. It decreases rate of the facing surprised errors and provides handling the errors correctly.

Unit testing supports writing better code. It forces the developer to consider the fact that another developer will use the written code. It should be smaller, clear and modular.

Writing test cases will save us time later. During development we are experience a lot of tricky points. When we need to work on the same process again, we should remember these tricky points and thus save debugging time. If we have written unit test cases, we can simply run them to find errors without knowing the tricky points.

Test cases are a small part of the documentation of the project. Every developer writes small code blocks in separate modules, in fact test cases define how the code will be work. Even developers should read test cases before writing a new line of code in related module.

The ontology of parameters we have used is very broad concept. We have included validation of the parameters before executing SQL and processing URL or file paths. Setting attribute values with RDF syntax or from an external RDF file is very helpful for changing parameter values without re-compiling the source code of test project.

As can be seen, unit testing helps us to write faster, reliable, more readable projects. We can use it for performance testing, memory utilization and stress testing or automating other manual operations.

It can be used in a small portion of the technology for developing web or mobile projects. I customized the unit test framework for web applications. We can validate the source code for popular scripting languages, can automate examining of inputs for test cases and can be used to measure the memory or CPU usage with other smart features.

APPENDIX A: USING YUNIT

It is very easy to do a unit testing project with YUnit. The following steps are all that is required to do such a project.

- First, open a project which will be your testing project.
- Then, add a reference to “UnitTestEngine.dll” in the Project References section.
- Add a new test class to your project that includes at least one test method.
- Just build your solution and open your test project assembly with the YUnit GUI.

YUnit will analyze the given assembly and then start to examine test methods in the project.

REFERENCES

1. Beck, K., *Test-driven Development by Example*, Addison-Wesley, 2002.

2. Patton, R., *Software Testing*, Sams Publishing, 2006.

3. Cook, B., *Preconditions and Post conditions with Use Cases Example*, Cook Enterprise Corporation, 2009.

4. Bowler, M., *Truck Factor*, http://www.agileadvice.com/archives/2005/05/truck_factor.html, 2005.

5. Osherove, R., *The Art of Unit Testing: With Examples in .Net*, Manning, 2009.

6. Clifton, M., *Unit Test Patterns*, http://www.marcclifton.com/Articles/UnitTesting/UnitTestPatterns, 2005.

7. Freeman, S. and N. Pryce, *Growing Object-Oriented Software, Guided by Tests*, Addison-Wesley, 2010.

8. *List of .NET Unit Testing Frameworks*, Wikipedia, http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks#.NET_programming_languages, 2010.

9. Skiena, S., *The Algorithm Design Manual*, Springer, 2010.

10. Allemang, D., and J. Handler, *Semantic Web for the Working Ontologist: Effective Modeling in RDFS and OWL*, Morgan Kaufman, 2008.

11. Lacy, L., *Owl: Representing Information Using the Web Ontology Language*, Trafford, 2005

12. Knublauch, H., *Ontology-Driven Software Development in the Context of the Semantic Web*, Monterey, 2004.

13. Arvaldeco, M., *How to Make a Project Risk Map*, http://hubpages.com/hub/How-to-Make-a-Project-Risk-Map, 2010.

14. *Dependency diagram*, Wikipedia, http://en.wikipedia.org/wiki/Dependency_diagram, 2010.

15. Nurhan, S., *ABD yazılımdan çok zarar ediyor*, http://www.ntvmsnbc.com/id/25135242, 2010.

REFERENCES NOT CITED

Meszaros, G., *xUnit Test Patterns: Refactoring Test Code*, Pearson Education Inc., 2007.

IEEE, *IEEE Standard for Software Unit Testing*, American National Standards Institute, 1986.

Artho, C. and A. Biere, *How to Scale up a Unit Test Framework*, http://fmv.jku.at/papers/ArthoEtAl-AST06.pdf, 2006.

Cheon, Y. and G. T. Leavens, *A Simple and Practical Approach to Unit Testing: The JML and JUnit Way*, http://www.springerlink.com/content/2yjt7jdnduntpgwp, 2002.

Vaaraniemi, S., *The Benefits of Automated Unit Testing*, 2003.

Caprio, G., *Unit Testing Overview*, http://www.code-magazine.com/article.aspx?quickid=0411031, 2004

Beust, C. and H. Suleiman, *Next Generation Java Testing: TestNG and Advanced Concepts*, Addison-Wesley Professional, 2007.