Abstract—NUnit is a framework that serves to generate component and module tests for .NET platforms. This framework acts as support in the test-driven development and enhances the quality of codes by quickly isolating errors in the source code. This in effect minimizes additional the cost of subsequent error search. We aim at highlighting the advantages and limitations of NUnit. We will present and exemplify key Asserts.

I. SOFTWARE DEVELOPMENT VS. SOFTWARE TEST

Time and financial constraints often prevent extensive software test in the development of software. However, these tests are vital as they enhance the quality and reliability of the product. As a result, tests are increasingly as vital as actual programming in large software projects. Currently, they generate up to 50% of the total cost in the software development process. Increasing demands on the standards of software may see a further growth of this proportion. Software tests are often perceived as importunate. Projects are often strapped for time. As a result, the development of superior tests costs additional time that does not go into working on the functionality of the software. Instead, the programmer spends the time in defining, implementing and analyzing test cases.

II. WHY CONDUCT TESTING?

The cost of error correction raises exponentially with the increase advancement and complexity of a project increases. As a result, early detection of errors in classes, procedures and components through software tests payoff and enhances the quality of the software. Tests have become an important part of software development as they guarantee a minimum degree of performance and reliability. Focus is on the development of a cost efficient test procedure that detects as many errors, easy to implement, and that can be automated and used even after modification of tested software components (regression tests). The capacity of a test procedure to support regression tests is especially vital given the frequent revision of software, but also considering the need for maintenance, advancement and error correction of a software product.

Studies have shown that component and module tests (Unit Tests) have a more superior cost-benefit ratio as compared to other test procedures. They generate lower follow-up cost of error search and correction, in spite of the high initial cost of component development.

Unit tests are the most effective means of detecting the highest number of errors in codes. They are an alternative to debugging and standard output that delivers ambiguous codes especially for complex software elements or strings. A string with multiple cycles already produces ambiguous results. Debugging is cumbersome where programmes have multiple threads. It requires considerable concentration levels and knowledge about the entire system; which in effect makes it prone to mistakes.

Component tests do not substitute integration, system, or performance tests. This is because their correct operation does not necessarily guarantee an error-free interplay of the components together. The viability of the components must be assessed through integration tests. These utilize a series of co-ordinated stand-alone test to determine the correct interplay of interdependent components in a system. Component tests may also be viewed as a preliminary stage of integration test. Tests that are conducted at system level as vital as they guarantee that the overall system functions as specified. System tests assess both the non-functional elements and the functional characteristics, such as correctness and completeness. Load and performance tests are software tests that monitor and evaluate the performance of a running system.

III. COMPONENT TEST WITH NUnit

NUnit is a framework that facilitates the writing and execution of automated Unit Tests for .NET languages. It is written entirely in C# and provides a graphic user interface. Initially, Philip A. Craig ported the test framework from JUnit. The developers Jim Newkirk, Michael Two, Alexei Vorontson and Charlie Poole then modified it and adapted it to fit .NET specifications. NUnit is Open Source and can thus be changed or adjusted to fit individual specifications. It can be accessed from the zlib/libpng license\(^1\) free of charge. A detailed documentation and download link for the up-to-date NUnit version can be found on the producer page\(^2\).

IV. PROCEDURAL METHOD

Appropriate Unit Tests are developed and implemented alongside the programming or before the implementation of the actual class or function. This allows the developer the chance to check the programmed code immediately and perform any necessary changes. NUnit is a framework that supports the generation, summarizing and execution of tests. It presents a graphic user interface (GUI) that shows the correct and incorrect tests in a clear manner. The tests are

---

\(^1\)http://www.opensource.org/licenses/zlib-license.html
\(^2\)http://www.nunit.org/
classified in individual test categories, thus do not interfere with the product code of the software package. The developer can incorporate asserts in the test objects. The tests abort if these asserts do not strike. Unit tests are White-Box tests. They imply prior knowledge of the internal functioning of the system that is being tested. Thus, the developer can test if all internal operations meet the pre-defined specifications. NUnit assist the developer in designing test drivers, stubs and tests, in the sourcing of test data, and in the execution of test through automation.

V. Test-Driven Development with Unit Tests

The use of unit tests is a vital element in the agile software development. The developer aims at attaining the necessary flexibility to quickly respond to changing consumer needs. The code must be constantly re-written and updated. Automated unit tests should guarantee the successful operation of a program after making changes.

Agile processes, such as “extreme programming” demand at least one test for each module or component [1]. In addition, they require testing of all procedures. This guarantees both exact localization and quick repair of program errors. Test cases are implemented before the actual functionalities, and thus demand that the developer looks into his code design in early stages. Component tests in the test-driven development are a mixture of White-Box and Black-Box (only their specification is known; not the implementation), the so-called Grey-Box tests. A combination of these two types guarantees optimum test case coverage, as it combines the advantages of both types. A continuous repetition of all tests (regression tests) guarantees remarkable quality enhancement [2].

VI. Structure of the Unit Tests

NUnit consists of a graphic user interface and a console that provide similar functionalities. They provide information about successful and aborted tests.

The window is partitioned in two columns as shown in diagram 1. The left shows the selected test categories and the therein contained tree structure. A green, yellow or red dot appears in front of the tests and the test class once the tests are run. A green dot symbolizes a successful assert, a yellow dot symbolizes tests that have not been run, while a red dot symbolizes aborted tests. To further clarify the results, a balk in the same colour is visible on the right column. Below this balk, possible errors are listed in a window. It shows the names of aborted tests and the expected results of the assertions and the actual results. This facilitates a quick error search and remedy. An additional branch appears on the left column of the graphic interface.

Beside the graphic interface, NUnit offers a console version. The console saves the results in a XML file. This file can be processed with appropriate programs. The command nunit-console <foldername> starts the verification of the file. It supports .dll, .csproj or .nunit (NUnit test) files. Additional commands can be viewed at http://www.nunit.org/.

VII. Composition of a Test Class

Test cases are defined in a particular user attribute (custom attributes) in a .NET source code and selected by use of reflection.

A. Example Class

namespace NUnitTestingExamples
{
    using System;
    using NUnit.Framework;

    [TestFixture]
    public class SuccessTests
    {
        [SetUp] public void Init()
        { /* ... */ }

        [TearDown] public void Dispose()
        { /* ... */ }

        [Test] public void Test1()
        { /* ... */ }

        [Test]
        [ExpectedException(typeof(XYZ)]
        public void Test2()
        {
            // Something that throws an
            // XYZ Exception
        }

        [Ignore("skip this test")]
        public void IgnoredTest()
        { /* ... */ }
    }
}

B. Explanation

- using NUnit.Framework binds the NUnit framework.
• **[TestFixture]** is a collection of test units. This attribute must be inserted before the definition of classes, in order to specify that the category contains tests.
• **[SetUp]** is an initialising intercept that contains the code to be executed before the particular unit test.
• **[TearDown]** is a code intercept that is executed after all tests.
• **[Test]** is a single component test. It is inserted before a procedure to earmark a test.
• **[ExpectedException]** contains exceptions that can be generated from the procedure.
• **[Ignore]** is inserted in the form of a unit test to deactivate it. NUnit ignores all tests with this attribute. A justification may also be provided.

VIII. asserts

The developer can use asserts to assess whether a program or a section of the program meets the specified requirements. These are written in the respective test categories and test the performance of the procedures with various entries or parameters. Asserts are structures as follows: Assert.NameOfAssert (argument1, argument2...). They are often supercharged and contain different lists of parameters. See the NUnit site for a detailed explanation of specific commands [3]. A short summary of the different commands is offered on Table 1.

Example: Assert.AreEqual(object expected, object actual).

Version 2.4 onwards contain new asserts for strings (StringAsserts) and collections (CollectionAsserts). The syntax also varies from normal asserts.

Example: StringAssert.Contains("the weather is nice", "nice") tests the string "the weather is nice" upon the assertion it contains "nice".

IX. Example

The category of person exemplifies the assertions. An object in this category contains a name, a work area and a given income. A person may have employed seniors or juniors. These people are also of the category person and can be inserted using the procedure setEmployee() and setChief(). For the purpose of simplicity, we will assume that only a definite number of juniors (MAX_EMPLOYEES) and a definite number of seniors (MAX_CHIEFS) is viable.

```csharp
namespace TestProgram
public class Person
{
    private string name;
    private string department;
    private double salary;
    private Person[] chiefs;
    private Person[] employees;
    const int MAX_EMPLOYEES = 10;
    const int MAX_CHIEFS = 10;
    public Person(string name)
    {
        department = "";
        salary = 0;
        chiefs = new Person[MAX_CHIEFS];
        employees = new Person[MAX_EMPLOYEES];
    }
    public void setChief(Person chief)
    {
        if (countChiefs() < MAX_CHIEFS)
            chiefs[countChiefs()] = chief;
    }
    public void setEmployee(Person employee)
    {
        if (countEmployees() < MAX_EMPLOYEES)
            employees[countEmployees()] = employee;
    }
    public int countEmployees()
    {
        // return amount of employees in list
    }
    public int countChiefs()
    {
```
The corresponding test files must have the same namespace as the category person and import the NUnit frame through the using directive. The category PersonTest that contains the tests of the category person must be labelled by the custom attribute [TestFixture].

```csharp
using System;
using System.Collections.Generic;
using NUnit.Framework;

namespace TestProgram
{
    [TestFixture]
    public class PersonTest
    {
        Person person;

        [SetUp]
        public void Init()
        {
            person = new Person("John");
            person.Salary = 5000;
            person.Department = "Accountancy";
        }
    }
}
```

The procedure Init() that is labelled with the [SetUp] attribute generates an entity of the category person and allocates a name, income and working area. The attribute [SetUp] guarantees that this procedure is carried out before all other procedures in the test category PersonTest. It therefore offers a concrete object for all other test procedures.

The following example is a simple test that is labelled [Test] attribute. It contains three asserts, that must be correct for successful processing of the test files. The assert AreEqual considers two strings as identical if they possess similar content. AreSame and AreNotSame tests whether the

```csharp
[Test]
public void NameToStringTest()
{
    Assert.AreEqual(person.ToString(), "John");
    Person person2 = new Person("John");
    Assert.AreNotSame(person, person2);
    Assert.AreEqual(person.ToString(), person2.ToString());
}
```

The category person contains two functions getChiefs(int pos) and getEmployees(int pos). These functions deliver the seniors of juniors of the respective persons at the index position pos. In addition, they may activate a IndexOutOfRangeException exception, if one attempts to access a index that is greater than the value of the constants MAX_EMPLOYEES and MAX_CHIEFS. While this exemption is introduced in this example, no provisions are made to absorb it.

```csharp
// throws IndexOutOfRangeException
public Person getEmployee(int pos)
{
    try
    {
        return employees[pos];
    }
    catch (IndexOutOfRangeException ex)
    {
        throw new IndexOutOfRangeException(
            "An Error occured: " + ex);
    }
}
```

The respective test procedures GetEmployeeExceptionTest is designed to anticipate and test this event. The additional attribute [ExpectedException] and the expected exception character label characteristic as a parameter (typeof(IndexOutOfRangeException)).

```csharp
[Test]
[ExpectedException(typeof(IndexOutOfRangeException))]
public void GetEmployeeExceptionTest()
{
    // throws IndexOutOfRangeException
    Assert.AreEqual(person.getEmployee(100).ToString(), "Julian");
}
```

X. ADVANTAGES OF ADOPTING NUnit
- Tests can be conducted and repeated as often as may be necessary (reproducible)
- the Unit Test tool enhances the quality of codes with a short period and at lower development costs
- can be automated
- the defragmenting of the software in smaller modules and components reduces its complexity

XI. LIMITS OF NUnit
- It does not offer the option to test for private members as the TestCase category from which all tests in NUnit as derived lies in a separate test category
- Tests must be written as codes in a file. Modification of these test suites requires programming knowledge.
- NUnit can only analyse the business logic. However, special frameworks that test GUI elements, such as NUnitASP [4] and NUnitForms [5] are available.
- In some occasions, tests programmes may be too complex
- Tests can only highlight errors. They do not notify the inexistence of the same
XII. RATING

NUnit do not offer a universal solution. However, it affords the developer a comfortable suite for current test cases. Its simplicity in the development of test cases in a separate file offers support in the test-driven development. The test-first strategy which dictates that tests are written before the actual implementation of the components forces the developer to deal with the design of the codes. This, in effect enhances the quality of the code. However, Unit test only reveal the existence of errors and not their inexistence. They offer a reasonable add-on but not a complete replacement of other test methods such as integration, system, load and performance tests. The development of a test case is often cheaper than subsequent error search.

A. Expandability of NUnit

NUnit can be expanded to include the capacity to test elements of graphic user interface. They all focus on NUnit and are also Open Source.

- NUnitASP [4] is available for ASP.NET pages. The framework supports queries from web controls, among others from buttons, labels and text boxes.
- NUnitForms is available for Windows forms elements.

B. Alternatives for NUnit

While the CSUnit [6] framework is similar to NUnit, it has fewer assert commands. The provider homepage offers tutorial for the transfer from NUnit to CSUnit. The commercial product TestDriven.NET [7] or the cost free VSNUnit [8] are recommended for the integration of Unit tests in visual studio.

The Team system component and module framework are already integrated in visual studio. This shows that Microsoft also recognises the importance of test-driven development, and employs it as a selling point. All subsequent Visual Studio versions will presumably support test-driven development.

REFERENCES