Parametrized Unit Testing Tool for. Net Framework

Mostafa Sami[1], Hala Abed El-Gali[1], Doaa Sami[2]
Computer Science Department
[1]Helwan University
[2]Misr University
Cairo, Egypt

ABSTRACT: *Unit testing has been widely recognized as an important and valuable means of improving software reliability, as it exposes bugs early in the software development life cycle. However, manual unit testing is often tedious and insufficient. Testing tools can be used to enable economical use of resources by reducing manual effort. Recently the use of parameters in unit testing has emerged as a very promising and effective methodology to allow the separation of two testing concerns or tasks: the specification of external, black-box behavior (i.e., assertions or specifications) by developers and the generation and selection of internal, white-box test inputs (i.e., high-code-covering test inputs) by tools. The Unit Testing Tool produced in this research is based on a parameterized test method that takes parameters, calls the code under test, and states assertions.*

Keywords: Testing, Unit Testing, Parameterized Unit Testing

Received: 2 March 2012, Revised 24 April 2012, Accepted 29 April 2012

1. Introduction

The essence of software testing is the comparison of the actual execution of a piece of software against that piece of software's expected behaviour [18]. As such, any attempt at automating the whole process of unit testing involves the mechanical generation of test cases that will exercise the software unit, the execution of these test cases, and an automated mechanism for determining whether the software behaved as expected. However, to be of any practical use to the software development professional, he must also be able to measure the thoroughness of the automatically generated test suite [2, 7, 14].

A unit test is simply a method without parameters that performs a sequence of method calls that exercise the code under test and asserts properties of the code's expected behaviour [19]. Unit tests are a key component of software engineering. The Extreme Programming discipline, for instance, leverages them to permit easy code changes. Being of such importance, many companies now provide tools, frameworks, and services around unit tests and each tool dedicated to only programs written in a special programming language [3, 8, 11].

So in this paper we present a parameterized unit testing tool that has the Standard unit testing features such as test, fixture, setup, teardown, ignore, expected exception, etc [15], Easy to use graphical user interface, Recipes for combining several test assemblies into one test suite, Search capabilities across tests, output, and statistics, Statistics per test to create performance base line, Categories to group tests for execution, Works with any. NET language (C#, VB.NET, Managed C++, etc.).

2. The Parameterized Testings in the Tool

Parameterized testing is sometimes also referred to as data-driven testing. The Unit Tool supports parameterization of tests in several ways:

- Simple Parameterization of a single test.
- Parameterization using a static method or property.
- Parameterization using an XML file .
- Parameterization using a database table.

Which option will be chosen depends on the circumstances. The order in which they are listed here starts with the simplest case and ends with the most powerful scenario. Parameterization can be used to test algorithms, APIs, and similar items. All types for parameterization are currently located in the parameterized Unit Testing Tool [1, 17].

2.1 Simple Parameterization

Most tests can be written without the need of parameterizing them. In some cases however, the user would like to be able to test an algorithm that takes a number of inputs and produces some number of outputs so there is a connection between the traditional test and parameterized test as figure 1.

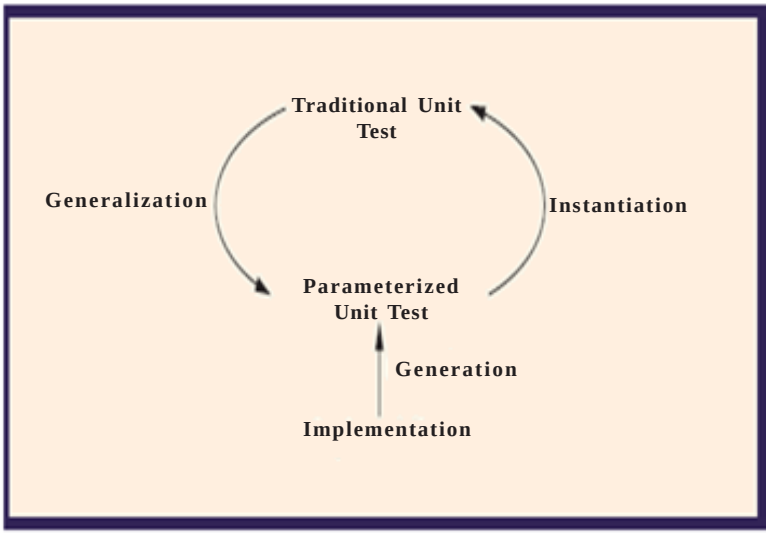

Figure 1. Connections between traditional and parameterized unit tests

As a very simple example, let's use the calculation of a discount as a percentage of the invoice amount:

Invoice Amount	Discount Percentage
100	0
200	5
500	10
1000	15

This simple case is certainly not very thrilling, but gets the idea. Without parameterization the user would have to write four tests, one for each invoice amount [16].

```
[Test]
public void DiscountRange1() {
DiscountCalculator calc = new DiscountCalculator();
 Assert.Equals(0, calc.PercentageFor(100));
}
[Test]
public void DiscountRange2() {
DiscountCalculator calc = new DiscountCalculator();
  Assert.Equals(5, calc.PercentageFor(200));
}
[Test]
public void DiscountRange3() {
DiscountCalculator calc = new DiscountCalculator();
  Assert.Equals(10, calc.PercentageFor(500));
}
[Test]
public void DiscountRange4() {
DiscountCalculator calc = new DiscountCalculator();
   Assert.Equals(15, calc.PercentageFor(1000));
}
```

Instead, it would be nice to refactor the obvious duplication within the tests and write a far simpler test. Parameterized tests allow the user to do exactly that.

With parameterization, the user adds parameters to a test and tells The Unit testing Tool where to read the parameter values from. To support this, by the attribute DataRow, this takes any number of parameters. The user can use the DataRowAttribute to decorate parameterized tests. Each attribute corresponds to a single execution of the test [1].

In essence the user has to give the test parameters and tell the parameterized Unit Testing Tool where to get the parameter values from. For that, introduced the attribute DataRow, which takes any number of parameters. The user can use the DataRowAttribute to decorate parameterized tests [12].

Taking the example from above, the user can write a single test with four DataRow attributes as follows:

```
[Test]
[DataRow (100, 0)]
[DataRow (200, 5)]
[DataRow (500, 10)]
[DataRow (1000, 15)]
public void DiscountRange(int invoiceAmount, int expectedDiscount) {
DiscountCalculator calc = new DiscountCalculator();
  Assert.Equals(expectedDiscount, calc.PercentageFor(invoiceAmount));
}
```

Basically, refactor the four discrete tests into something easier to understand and maintain.

2.2 Specifying an ExpectedException

What if for some data rows the user would expect an exception to be thrown? Well, The Unit Testing Tool supports this as well,

through a named property to the DataRow attribute, ExpectedException.
Here is a simple example of how an expected exception can be specified:

```
[Test]
[DataRow(2, 1, 2)]
[DataRow(6, 2, 3)]
[DataRow(4, 0, 0, ExpectedException = typeof(DivideByZeroException))]
public void DivisionTest(int numerator, int denominator, int quotient) {
  Assert.Equals(quotient, numerator / denominator);
}
```

Note: more than one data row can have an expected exception. Also, the expected exception can be different for each of those data rows [10, 21].

2.3 Parameterization with Static Method or Property

The DataRow approach is useful if the user only want to use a set of parameters once. However, in some cases the user may want to use the same set of data for more than one test. In this case the user can use the DataSourceAttribute and specify a type as a parameter for the attribute. That type is used as the data provider for the parameterization. It needs to implement a static method or a static property that returns an array of data rows.

```
[Test]
[DataSource(typeof(FixtureWithStaticDataProvider))]
public void Squares(int oper, int result) {
  Assert.Equals(result, oper * oper);
}
```

This test requires a class with the name Fixture with Static DataProvider to be implemented elsewhere.

At runtime, the parameterized Unit Testing Tool will search the data provider class for a static method that returns an array of DataRow objects. It will invoke the first one it can find and use the returned array of DataRow objects as the parameter sets for the parameterized test method.

Note: the data provider class and the test fixture containing the test can be the same.

Again, if the user expects an exception to be thrown for one or more of the data rows, the user can assign the expected exception type to the named property ExpectedException of the DataRow attribute. This can be seen on the third data row above.

2.4 Parameterization with XML-File

Suppose that the user would like the parameters to be read from an XML file. The user can do this with the DataSource attribute as well.

And again, the user also needs to account for the possibility that one or more of the data rows expects an exception.

Note: as with the other ways of specifying an expected exception, the user can specify any type. This includes exceptions that the user has implemented by the user.

2.5 Parameterization using a Database Table

The fourth option to provide sets of parameter values to a parameterized test is specifying a .NET data provider, a connection string, and a database table name[1].

This is just a standard connection string which the parameterized Unit Testing Tool passes on to the managed ADO.NET data provider. The first parameter is the Invariant Name of the .NET data provider. The factory for the data provider must be registered in the machine.config file. By default .NET has factories registered for SQL, Oracle, OLE DB, and ODBC.

During runtime the parameterized Unit Testing Tool executes the test once for each data row and reports separately on the outcome.

Note: accessing a database, if local, is an expensive operation. Some databases work in-memory thus at least avoiding the cross-process and/or cross-machine communication. Whether the user chooses a database table as a feed for his parameterized test requires careful considerations and trade-offs.

3. Categories

The Unit Testing Tool supports categorization of tests. Basically this means the user can assign categories to tests and test fixtures, and then use the categorization for instance for selecting tests [2, 3, 20].

If the user doesn't need this feature right now he can safely ignore it. This is one of the design principles The Tool try to follow wherever possible.

If the user assigns one or more category to a test or a test fixture, the user should be aware of the following rules:

• If a test has no categories assigned then the default setup/teardown method will be executed. The default setup/teardown method is the one that has no categories assigned to it. If no such default setup/teardown method exists, no setup/teardown will get executed.

• If a test has one or more categories assigned then the setup/teardown for that category or those categories will be executed. If a categorized setup/teardown method doesn't exist, the default method will be executed if it exists [6].

• If more than one default setup/teardown method exists, or if more than one categorized setup/teardown method for the same category exists, this is considered to be an error and the test(s) will fail. This test is performed per test fixture. The latter means that in a hierarchy of test fixtures a base class can have a setup/teardown method and a derived class can have a setup/teardown method, either default or categorized.

• If a test fixture is derived from a base class that is itself a test fixture, setup/teardown methods from the base class will not be considered for the derived test fixture. Also, even if a method in the base class is declared virtual and marked as SetUp/TearDown, it will not be considered by The Unit Testing Tool's runtime. If the user need to execute setup/teardown code in the base class, the user need to call the base class method from your code, e.g. base.MySpecialSetupMethod ().

Once the user has defined categories for his tests he can then select categories in the graphical user interface. When the user then save his settings as a recipe, the category selection will be save along with it. After that the user can supply the recipe to UnitCmd, e.g. for inclusion in his automated build. The category selector is part of The Unit Testing Tool's runtime environment regardless of the front end.

3.1 Category Hierarchy

Both TestFixture and Test support categories. As the Test methods are always contained in a TestFixture, and as both can be assigned to one or more categories, there are more scenarios to be explored [9].

Let's take the following two test fixtures:

```
[Fixture (Categories = " DB " )]
  public class Fixture10 {
    [Test (Categories  = " mySQL " )]
    public void Test1 () {
    }
    [Test (Categories = " MSFT " )]
    public void Test2 () {
    }
  }
  [Fixture (Categories = " DATA " )]
```

```
public class Fixture 2 {
    [Test (Categories = " mySQL " )]
    public void Test3 () {
    }
    [Test (Categories = "MSFT " )]
    public void Test4 () {
    }
    [Test (Categories = " Oracle " )]
    public void Test5 () {
    }
  }
```

Included Categories	Excluded Categories	Executed Tests
DB	-	Test1, Test2
DATA	-	Test3, Test4, Test5
DB	MSFT	Test1
DB	MySQL	Test2
DATA	MSFT	Test3, Test5
DATA	mySQL, Oracle	Test4
-	MSFT	Test1, Test3
-	MySQL	Test2, Test4, Test5
-	-	All tests

Table 1. Different Set of Tests Depend on the Test's Category

There are now several scenarios possible, and depending on which categories the user include or exclude, a different set of tests will be executed as seen in Table1.

The easiest way to include or exclude categories is through the graphical user interface of either UnitRunner or the addin. Alternatively the user can edit the recipe directly, which is an XML file and also includes the settings for the Category Selector [5].

3.1.1 Test Fixtures

A test fixture is basically a class that contains tests.

The tool supports two different techniques for decorating test fixtures within an assembly:

- Attribute based
- Name based

For the first approach, the tool's runtime inspects all publicly visible types within an assembly. All classes that have the attribute TestFixtureAttribute will be available for execution. The following is an example of such a class:

```
[TestFixture]
  public class FacadeTests {
    [Test]
    public void WithEmptyID() {
      ...
```

```
}
    [Test]
    public void SystemIDReturnsAdminObject ()
{
      ...
    }
  }
```

The second option is to follow a certain name convention. The tool finds all classes with a prefix "*Test* ". This search is case insensitive, so it will also find all classes beginning with "*test* ", "*TEST* ", "*tESt* ", "*TesT* ", etc.

3.1.2 Important Rules

The tool searches only publicly visible types. In C# this means that the user must declare the test fixture as public. The tool will not find test fixtures that are internal or private.

The user's test fixture doesn't have to have a special base class. Also, they don't have to implement a special interface. If there is however code that the user's tests share, the user may want to consider a common base class for some test fixtures.

3.1.3 Recipe

A key concept of The Unit Testing Tool is the recipe. A recipe is basically almost like a project in the favorite IDE. Basically a recipe consists of one or more test assemblies, that is assemblies that contain tests.

Furthermore the recipe can contain additional information, e.g. about the categories to be included/excluded in a test run [4, 9].

The Unit Testing Tool command line is able to read the recipe and execute the tests according to the information in the recipe. From a certain perspective the UnitRunner (the native application) becomes the editor for recipes.

3.1.4 Selectors

The Unit Testing Tool uses the concept of selectors to determine the tests to run within a recipe. A recipe consists of one or more test assemblies. Each of them containing zero or more tests [4, 5].

A selector is basically a piece of logic that determines whether or not a test is included in a test run. An example could be a category selector. It includes/excludes tests based on the selected categories and the categories assigned to the test.

Currently The Unit Testing Tool supports CategorySelector and CheckedTestsSelector as the only selector types.

The rule is that a test fixture or test is run, if and only if all selectors vote to include the fixture of the test in the test run. This decision is made in real time. This means for example that during a test run, the user can check or uncheck a test or a test fixture, and so long as it has been looked at yet, it will then either be included or excluded from the test run.

Depending on the user choices for the different selectors, categories and check status, the number of included test will be displayed in the progress bar. Some selections and/or check statuses will lead to no test being included in a tests run. So, to back out of any of the selections, both the test hierarchy tab and the category tab provide a " *Reset* " button to set the according selector to a state where all tests and test fixtures are included in the next test run.

3.2 Options Dialog Feature

3.2.1 Start Behaviour

The first option defines what The Unit Testing Tool should load upon startup[13]. There are three possibilities:

- load nothing
- load most recently used recipe
- load most recently used assembly

When closing The Unit Testing Tool the most recently used recipe and assembly are stored. Selecting "*load most recently used recipe*" or "*load most recently used assembly will load that recipe/assembly*".

Note: The user can build a new version of the assembly (or any assembly in the recipe) and The Unit Testing Tool will then load the updated assembly.

3.2.2 Shut down Behaviour
When exiting the user of the Unit Testing Tool can configure that is prompts to save a recipe if it contains unsaved changes.

3.2.3 Output Tab Behaviour
There is one option for controlling the output tab behavior. The user can choose to get line numbers displayed.

3.2.4 Test Hierarchy Tree Options
There are two options to configure the behavior of the test hierarchy tree:

- Automatically expand nodes with comments, etc.
- Automatically expand the entire tree upon loading a recipe

The first option will automatically expand any node in those cases where a test is skipped, or a failure or error occurs. The additionally visible elements in the tree contain more information about the error or the reason for skipping a test.

The second option tells The Unit Testing Tool whether to fully expand the entire tree when loading a recipe.

3.2.5 Colors
The default setting is to display success as "*Lime*" and failure or error as *" Red "*. However, people with red-green blindness sometimes have difficulties to distinguish between those two colors and hence The Unit Testing Tool offers the option to choose different colors [5, 13].

An extra button resets the selection to the "*Default Colors*" (Lime/Red).

4. Conclusions

We presented the concept of parameterized unit tests, a generalization of established closed unit tests. Parameterization allows the separation of two concerns: The specification of the behavior of the system, and the test cases to cover a particular implementation.

The tool introduced in this paper is a parameterized unit testing framework for the .NET Framework. It is designed to work with any .NET compliant language. It has specifically been tested with C#, Visual Basic .NET, Managed C++, and J#.

The Tool follows the concepts of other parameterized unit testing frameworks in the XUnit family. Along with the standard features, the tool offers abilities that are uncommon in other parameterized unit testing frameworks for .NET:

Categories to group included, excluded tests

ExpectedException working with concrete instances rather than type only

A tab for simple performance base lining

A very rich set of assertions, continuously expanded

Rich set of attributes for implementing tests

Parameterized testing, data-driven testing

Search abilities, saving time when test suites have thousands of tests.

References

[1] Tillmann, N., de Halleux, J., Tao Xie. (2010). Parameterized unit testing: theory and practice, *In*: ACM/IEEE 32nd International Conference on Software Engineering,, 2, 483 - 484, 2-8.
[2] Narendra Kumar Rao, B., Rama Mohan Reddy, A., Ravi, K. (2011). Level dependencies of individual entities in random unit testing of structured code, *In*: 3rd International ConferenceElectronics Computer Technology (ICECT), 6, 223 - 226, 8-10.
[3] Runeson, P. (2006). A survey of unit testing practices, I*EEE Journals & Magazines*, 23 (4) 22 - 29.
[4] Williams, L., Kudrjavets, G., Nagappan, N. (2009). On the Effectiveness of Unit Test Automation at Microsoft,Software Reliability Engineering, *In*: ISSRE '09. 20th International Symposium, 81 – 89.

[5] Cheng-hui Huang. (2005). A semi-automatic generator for unit testing code files based on JUnit, *In*: IEEE International Conference on Systems, Man and Cybernetics, 10-12, 140 - 145.
[6] Tao Xie, Taneja, K., Kale, S., Marinov, D. (2007). Towards a Framework for Differential Unit Testing of Object-Oriented Programs, Automation of Software Test. AST '07. Second International Workshop, p. 20-26.
[7] Gupta, A., Jalote, P. (2007). Test Inspected Unit or Inspect Unit Tested Code? *In*: ESEM 2007. First International Symposium on Empirical Software Engineering and Measurement, 51 – 60, 20-21.
[8] Bin Xu. (2009). Towards Efficient Collaborative Component-Based Software Unit Testing via Extend E-CARGO Model-Based Activity Dependence Identification, *In:* International Symposium on Intelligent Ubiquitous Computing and Education,, 172 – 175, 15-16.
[9] Vegas, S., Juristo, N., Basili, V. R. (2009). Maturing Software Engineering Knowledge through Classifications: A Case Study on Unit Testing Techniques, *IEEE Transactions on Software Engineering*, 35 (4) 551 – 565.
[10] Na Zhang, Xiaoan Bao, ZuohuaDing. (2009). Unit Testing: Static Analysis and Dynamic Analysis, *In*: Fourth International Conference,Computer Sciences and Convergence Information Technology. ICCIT '09. 232 – 237, 24-26.
[11] Liangliang Kong, Zhaolin Yin. (2006). The Extension of the Unit Testing Tool Junit for Special Testings, Computer and Computational Sciences, IMSCCS '06. First International Multi-Symposiums, 2, 410 – 415, 20-24.
[12] Mouy, P., Marre, B., Williams, N., Le Gall, P. (2008). Generation of All-Paths Unit Test with Function Calls , 1st International Conference, Software Testing, Verification, and Validation, 35, 32 – 41.
[13] Wloka, J., Ryder, B.G., Tip, F. (2009). JUnitMX - A change-aware unit testing tool, *In*: IEEE 31st International Conference on Software Engineering. ICSE 2009, p. 567 – 570.
[14] Runeson, P., Stefik, A., Andrews, A., Gronblom, S., Porres, I., Siebert, S. (2011). A Comparative Analysis of Three Replicated Experiments Comparing Inspection and Unit Testing, *In*: Second International Workshop, Replication in Empirical Software Engineering Research (RESER), 35 – 42.
[15] Tan, R. P., Edwards, S., Evaluating Automated Unit Testing in Sulu, International Conference on Software Testing, Verification, and Validation, 1st , 59, 62 – 71.
[16] Nan Li, Praphamontripong, U., Offutt, J. (2009). An Experimental Comparison of Four Unit Test Criteria: Mutation, Edge-Pair, All-Uses and Prime Path Coverage, ICSTW'09. International Conference on Software Testing, Verification and Validation Workshops, 220 – 229.
[17] Tao Xie, Tillmann, N., de Halleux, J., Schulte, W. (2009). Mutation Analysis of Parameterized Unit Tests, Software Testing, *In*: ICSTW '09. International Conference,Verification and Validation Workshops, 177 – 181.
[18] Thiruvathukal, G. K., Laufer, K., Gonzalez, B. (2006). Unit Testing Considered Useful, *Computing in Science & Engineering Journal*, 8 (6) 76 – 87.
[19] Juristo, N., Moreno, A. M., Vegas, S., Solari, M. (2006). In Search of What We Experimentally Know about Unit Testing, *IEEE*, 23 (6), 72 – 80.
[20] Simons, A. J. H., Thomson, C. D. (2007). Lazy Systematic Unit Testing: JWalk versus JUnit, Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, TAICPART-MUTATION, 14, 138.
[21] Feng Gao, Gannan Yuan, Chun Zheng, Chang Liu. (2011). Research on the Application Technology of Unit Testing Based on Priority, *In:* Fourth International Joint Conference on Computational Sciences and Optimization (CSO), 222, 997 – 1001.