Unit Testing Succinctly

by Marc Clifton

Syncfusion | Technology Resource Portal

Unit Testing Succinctly

By
Marc Clifton

Foreword by Daniel Jebaraj

Important licensing information. Please read.

Technical Reviewer: Praveen Ramesh, director of development, Syncfusion, Inc.

Copy Editor: Mary Brunnemer

Acquisitions Coordinator: Jessica Rightmer, senior marketing strategist, Syncfusion, Inc.

Proofreader: Graham High, content producer, Syncfusion, Inc.

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese.”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Marc Clifton is a consultant with more than 30 years of experience in programming, starting from 8-bit assembly language, and currently working with C#, F#, Ruby, and more. He has authored 149 articles on CodeProject at the time of this writing. He also provides beginning to advanced programming instruction and is a technology mentor through the Albany Chamber of Commerce.

When he is not coding or writing articles, he enjoys playing Texas Hold'em poker, cooking, having philosophical conversations with friends, and working on social issues.

Introduction

The purpose of this book is not to evangelize that unit testing will save you time and money in your development process and that it will improve the quality of your product, but rather to:

- Provide the necessary information for you to intelligently balance unit testing with the rest of your development effort and associated costs.
- Provide guidance on how to develop meaningful unit tests.
- Consider which technologies you want to utilize as a unit test engine.
- Consider how unit tests affect the coding style and architecture of your application.

Code Examples

All code examples have been tested in both Visual Studio 2008 Pro with .NET 3.5 and Visual Studio 2012 Pro with .NET 4.5.

Screenshots

Due to changes in the Test Explorer window of Visual Studio 2012, the test result screenshots are from Visual Studio 2008, which has a layout more suitable for conveying both the test result and the test result message. The exceptions to this are screenshots from other unit test applications, such as NUnit.

Expectations of the Reader

The reader is expected to have knowledge of C# and the .NET Framework. Some of the examples require knowledge of lambda expressions and LINQ.

Organization of this Book

The first part of this book discusses the purpose, philosophy, and practice of unit testing:

“How Does Unit Testing Work?" describes how a unit test engine works by creating a simple console-based unit test engine.

“Common Unit Test Tools” is a brief overview of unit testing engines.

Writing unit tests with Visual Studio’s unit test engine and NUnit are then covered in detail in the following chapters:

- “Testing Basics”
- “Unit Testing with Visual Studio”
- “Unit Testing with NUnit”

The book concludes with a discussion of advanced unit testing, dealing with code coverage and techniques for accessing internal fields and methods, followed by a chapter on other uses for unit testing.

Chapter 1 Why Unit Test?

The usual mantra we hear regarding any software methodology is that it improves usability and quality, reduces development and testing time, and brings the product to market faster and with fewer bugs. These are lofty goals, but I have yet to see a methodology deliver the Grail of software development.

Ultimately, the primary reason to write unit tests is to prove correctness, and this happens only if you write unit tests *well.* Unit tests by themselves will not directly improve the usability or quality of your product. You can still make a mess of the application whether it's proven correct or not—and it certainly is not guaranteed to reduce development and testing time (more on this later) or bring your product to market sooner.

So, let's be clear and real from the get-go: unit testing can be used to verify correctness, and any *side effect* that occurs with regard to your development process must be balanced with the effort of writing and maintaining useful unit tests.

Measuring Correctness

Well-written unit tests will give you a *measurable* degree of confidence that the myriad of methods that comprise your application will behave correctly. The simplest way to objectively make this measurement is a coverage test: What percentage of the methods in your application have unit tests written against them? While this question does not directly address whether a method should be considered a unit (discussed later), or whether the tests are meaningful, it is nonetheless a measurement that you can take at any time and can be used as a benchmark for the correctness of your application.

Unit testing is an iterative process—there will always be bugs that are missed with unit testing. However, the number of bugs reported over time and the number of unresolved versus resolved issues provides meaningful information regarding your application's health. While it is impossible to say, "With unit testing, the number of bugs has been reduced by 50 percent," it is possible to measure how many bugs your application has because of incomplete unit test coverage. As you write unit tests to verify the issue and the fix, you can also measure how many unit tests you have written against reported bugs as compared to the total number of unit tests.

All of these benchmarks bring a degree of objectivity to your development process. Therefore, one of the benefits of unit testing is that it provides everyone, from developers to managers, with objective information that can be fed back into the development process to improve that process.

Repetition, Repetition, Repetition

Another benefit is repeatability, otherwise known as regression testing. As an application matures, we want to ensure that existing, *working* code is not broken. By writing unit tests against methods as they are written and adding unit tests for bugs as they are reported, all of these can be retested automatically when new code is added or existing code is changed. Unit tests become a significant time-reduction tool when it comes to testing whether an application still behaves correctly after a minor or significant code change. While unit testing does not replace usability testing, performance testing, load testing, and so forth, it definitely helps to eliminate the time wasted on the common question: “This worked before; why doesn’t it now?”

Code Coverage

It’s easy to test that a method is doing the right thing when all the processes in the method execute linearly. However, once you add an **`if`** statement or a **`switch`** statement, you are creating *cyclomatic complexity*, which is a fancy way of saying that your code now has multiple paths of execution. The most useful unit tests are those that test *every single code branch* that occurs in your method. Writing these kinds of unit tests can be painstaking but are well worth the effort, as they guarantee that at least every code branch has been executed, which is not something that may occur during acceptance testing, usability testing, or other testing that the quality assurance department (if you have one) performs.

Chapter 2 What is Unit Testing?

Unit testing is all about proving correctness. To prove that something is working correctly, you first have to understand what both a *unit* and a *test* actually are before you can explore what is provable within the capabilities of unit testing.

What is a Unit?

In the context of unit testing, a unit has several characteristics.

Pure Units

A pure unit is the easiest and most ideal method for writing a unit test. A pure unit has several characteristics that facilitate easy testing.

A Unit Should (Ideally) Not Call Other Methods

With regard to unit testing, a unit should first and foremost be a method that does something without calling any other methods. Examples of these pure units can be found in the **`String`** and **`Math`** classes—most of the operations performed do not rely on any other method. For example, the following code (taken from something the author has written)

```
public void SelectedMasters()
{
  string currentEntity = dgvModel.DataMember;
  string navToEntity = cbMasterTables.SelectedItem.ToString();
  DataGridViewSelectedRowCollection selectedRows = dgvModel.SelectedRows;

  StringBuilder qualifier = BuildQualifier(selectedRows);
  UpdateGrid(navToEntity);
  SetRowFilter(navToEntity, qualifier.ToString());
  ShowNavigateToMaster(navToEntity, qualifier.ToString());
}
```

should not be considered a unit for three reasons:

- Rather than taking parameters, it obtains the values involved in the computation from user interface objects, specifically a DataGridView and a ComboBox.
- It makes several calls to other methods that potentially *are* units.
- One of the methods appears to update the display, entangling a computation with a visualization.

The first reason points out a subtle issue—properties should be considered method calls. In fact, they are in the underlying implementation. If your method is using properties of other classes, this is a kind of method call and should be considered carefully when writing a suitable unit.

Realistically, this is not always possible. Often enough, a call to the framework or some other API is required for the unit to successfully do its work. However, these calls should be inspected to determine whether the method could be improved to make a purer unit, for example, by extracting the calls into a higher method and passing the results of the calls as a parameter to the unit.

A Unit Should Do Only One Thing

A corollary to “a unit should not call other methods” is that a unit is a method that *does one thing* and one thing only. Often other methods are called in order to do *more than one thing*—a valuable skill to know when something actually consists of several subtasks—even if it can be described as a high-level task, which makes it sound like a single task!

The following code might look like a reasonable unit that does one thing: it inserts a name into the database.

```
public int Insert(Person person)
{
  DbProviderFactory factory = SqlClientFactory.Instance;

  using (DbConnection connection = factory.CreateConnection())
  {
    connection.ConnectionString = "Server=localhost; Database=myDataBase;
        Trusted_Connection=True;";
    connection.Open();

    using (DbCommand command = connection.CreateCommand())
    {
      command.CommandText = "insert into PERSON (ID, NAME) values (@Id, @Name)";
      command.CommandType = CommandType.Text;

      DbParameter id = command.CreateParameter();
      id.ParameterName = "@Id";
      id.DbType = DbType.Int32;
      id.Value = person.Id;

      DbParameter name = command.CreateParameter();
      name.ParameterName = "@Name";
      name.DbType = DbType.String;
      name.Size = 50;
      name.Value = person.Name;

      command.Parameters.AddRange(new DbParameter[] { id, name });
```

```
      int rowsAffected = command.ExecuteNonQuery();

      return rowsAffected;
    }
  }
}
```

However, this code is actually doing several things:

- Obtaining a **`SqlClient`** factory provider instance.
- Instantiating a connection and opening it.
- Instantiating a command and initializing the command.
- Creating and adding two parameters to the command.
- Executing the command and returning the number of rows affected.

There are a variety of issues with this code that disqualify it from being a unit and make it difficult to reduce into basic units. A better way to write this code might look like this:

```
public int RefactoredInsert(Person person)
{
  DbProviderFactory factory = SqlClientFactory.Instance;
  using (DbConnection conn = OpenConnection(factory, "Server=localhost; Database=myDataBase;
     Trusted_Connection=True;"))
  {
    using (DbCommand cmd = CreateTextCommand(conn, "insert into PERSON (ID, NAME) values (@Id, @Name)"))
    {
      AddParameter(cmd, "@Id", person.Id);
      AddParameter(cmd, "@Name", 50, person.Name);
      int rowsAffected = cmd.ExecuteNonQuery();

      return rowsAffected;
    }
  }
}

protected DbConnection OpenConnection(DbProviderFactory factory, string connectString)
{
  DbConnection conn = factory.CreateConnection();
  conn.ConnectionString = connectString;
  conn.Open();

  return conn;
}

protected DbCommand CreateTextCommand(DbConnection conn, string cmdText)
{
  DbCommand cmd = conn.CreateCommand();
  cmd.CommandText = cmdText;
  cmd.CommandType = CommandType.Text;

  return cmd;
}
```

```
protected void AddParameter(DbCommand cmd, string paramName, int paramValue)
{
  DbParameter param = cmd.CreateParameter();
  param.ParameterName = paramName;
  param.DbType = DbType.Int32;
  param.Value = paramValue;
  cmd.Parameters.Add(param);
}

protected void AddParameter(DbCommand cmd, string paramName, int size, string paramValue)
{
  DbParameter param = cmd.CreateParameter();
  param.ParameterName = paramName;
  param.DbType = DbType.String;
  param.Size = size;
  param.Value = paramValue;
  cmd.Parameters.Add(param);
}
```

Notice how, in addition to looking cleaner, the methods **`OpenConnection`**, **`CreateTextCommand`**, and **`AddParameter`** are more suitable to unit testing (ignoring the fact that they are protected methods). These methods do only one thing and, as units, can be tested to ensure that they do that one thing correctly. From this, there becomes little point to testing the **`RefactoredInsert`** method, as it relies entirely on other functions that have unit tests. At best, one might want to write some exception handling test cases, and possibly some validation on the fields in the **`Person`** table.

Provably Correct Code

What if the higher-level method does something more than just call other methods for which there are unit tests, say, some sort of additional computation? In that case, the code performing the computation should be moved to its own method, tests should be written for it, and again the higher-level method can rely on the correctness of the code it calls. This is the process of constructing provably correct code. The correctness of higher-level methods improves when all they do is call lower-level methods that have proofs (unit tests) of correctness.

A Unit Should Not (Ideally) Have Multiple Code Paths

Cyclomatic complexity is the bane of unit testing and application testing in general, as it increases the difficulty of testing all the code paths. Ideally, a unit will not have any **`if`** or **`switch`** statements. The body of those statements should be regarded as the units (assuming they meet the other criteria of a unit) and to be made testable, should be extracted into their own methods.

Here is another example taken from the author's MyXaml project (part of the parser):

```
if (tagName=="*")
{
  foreach (XmlNode node in topElement.ChildNodes)
  {
    if (!(node is XmlComment))
    {
```

```
      objectNode = node;
      break;
    }
  }

  foreach (XmlAttribute attr in objectNode.Attributes)
  {
    if (attr.LocalName == "Name")
    {
      nameAttr = attr;
      break;
    }
  }
}
else
{
  ... etc ...
}
```

Here we have multiple code paths involving **`if`**, **`else`**, and **`foreach`** statements, which:

- Create setup complexity, as many conditions must be met to execute the inner code.
- Create testing complexity, as the code paths require different setups to ensure that each code path is tested.

Obviously, conditional branching, loops, case statements, etc. cannot be avoided, but it may be worthwhile to consider refactoring the code so that the internals of the conditions and loops are separate methods that can be independently tested. Then the tests for the higher-level method can simply ensure that the states (represented by conditions, loops, switches, etc.) are properly handled, independent of the computations that they perform.

Dependent Units

Methods that have dependencies on other classes, data, and state information are more complex to test because those dependencies translate into requirements for instantiated objects, existence of data, and predetermined state.

Preconditions

In its simplest form, dependent units have preconditions that must be met. Unit test engines provide mechanisms to instantiate test dependencies, both for individual tests and for all tests within a test group, or “fixture.”

Actual or Simulated Services

Complicated dependent units require services such as database connections to be instantiated or simulated. In the earlier code example, the **`Insert`** method cannot be unit tested without the ability to connect to an actual database. This code becomes more testable if the database interaction can be simulated, typically through the use of interfaces or base classes (abstract or not).

The refactored methods in the **`Insert`** code described earlier are a good example because **`DbProviderFactory`** is an abstract base class, so one can easily create a class deriving from **`DbProviderFactory`** to simulate the database connection.

Handling External Exceptions

Dependent units, because they are making calls to other APIs or methods, are also more fragile—they may need to explicitly handle errors potentially generated by the methods that they call. In the earlier code sample, the **`Insert`** method's code could be wrapped in a try-catch block, because it is certainly possible that the database connection may not exist. The exception handler might return **`0`** for the number of rows affected, reporting the error through some other mechanism. In such a scenario, the unit tests must be capable of simulating this exception to ensure that all code paths are executed correctly, including **`catch`** and **`finally`** blocks.

What is a Test?

A test provides a useful assertion of the correctness of the unit. Tests that assert the correctness of a unit typically exercise the unit in two ways:

- Testing how the unit behaves under normal conditions.
- Testing how the unit behaves under abnormal conditions.

Normal Conditions Testing

Testing how the unit behaves under normal conditions is by far the easiest test to write. After all, when we write a function, we are either writing it to satisfy an explicit or implicit requirement. The implementation reflects an understanding of that requirement, which in part encompasses what we expect as inputs to the function and how we expect the function to behave with those inputs. Therefore, we are testing the result of the function given expected inputs, whether the result of the function is a return value or a state change. Furthermore, if the unit is dependent on other functions or services, we are also expecting them to behave correctly and are writing a test with that implied assumption.

Abnormal Conditions Testing

Testing how the unit behaves under abnormal conditions is much more difficult. It requires determining what an abnormal condition is, which is usually not obvious by inspecting the code. This is made more complicated when testing a dependent unit—a unit that is expecting another function or service to behave correctly. In addition, we don't know how another programmer or user might exercise the unit.

Unit Tests and Other Testing Practices

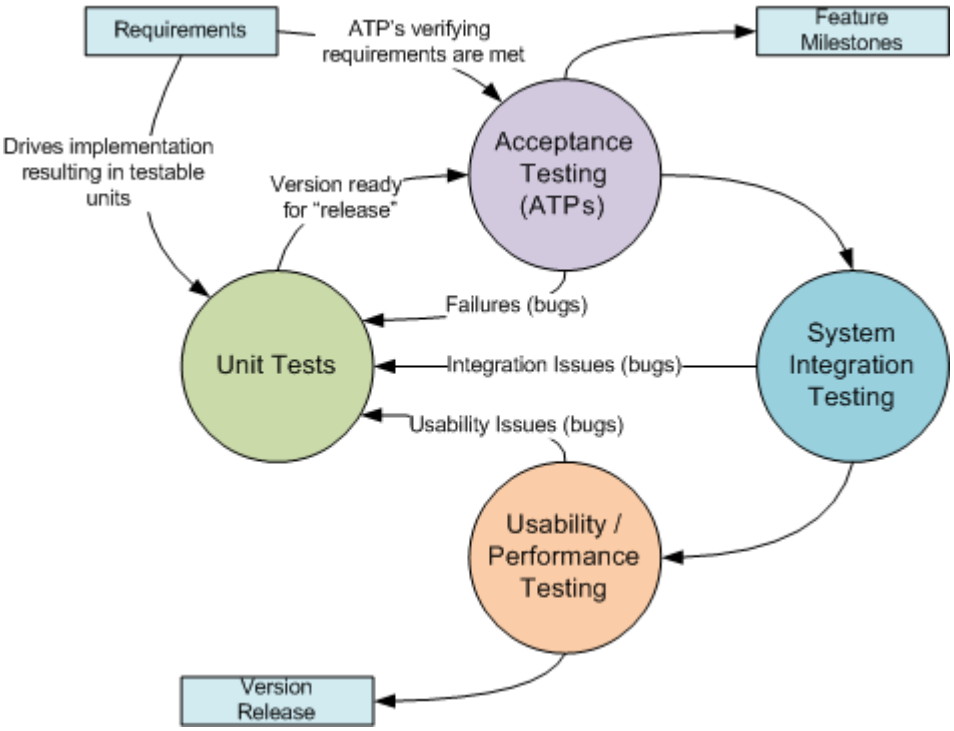


Figure 1: Unit Testing as Part of a Comprehensive Test Approach

Unit testing does not replace other testing practices; it should complement other testing practices, providing additional documentation support and confidence. Figure 1 illustrates one concept of the "application development flow"—how other testing integrates with unit testing. Note that the customer can be involved in any stage, though usually at the acceptance test procedure (ATP), system integration, and usability stages.

Compare this with the V-model of the software development and testing process. While it is related to the waterfall model of software development (which, ultimately, all other software development models are either a subset or an extension of), the V-model provides a good picture of what kind of testing is required for each layer of the software development process:

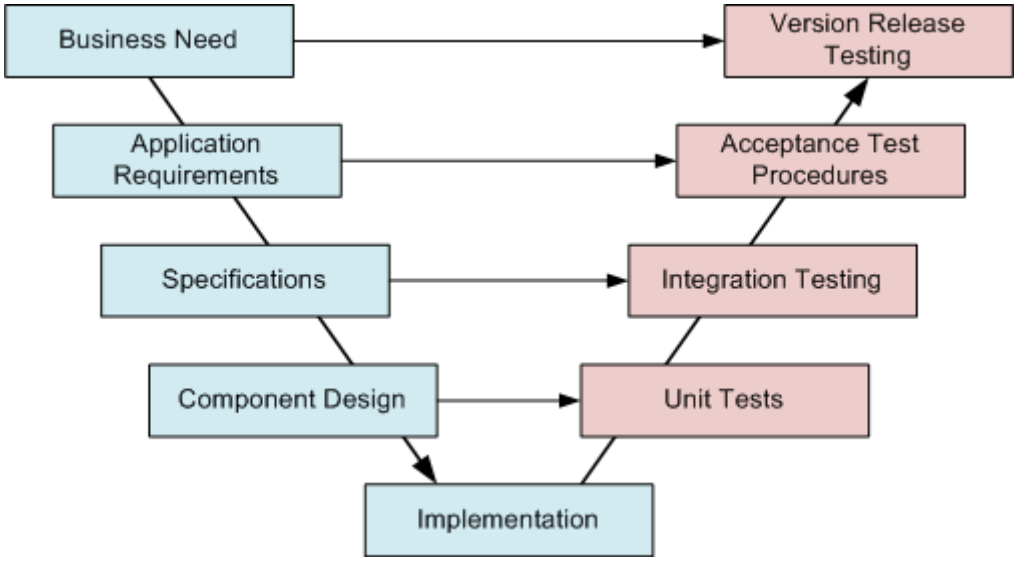


Figure 2: The V-Model of Testing

Furthermore, when a test point fails in some other test practice, a specific piece of code can usually be identified as being responsible for the failure. When that is the case, it becomes possible to treat that piece of code as a unit and write a unit test to first create the failure and, when the code has been changed, to verify the fix.

Acceptance Test Procedures

An acceptance test procedure (ATP) is often used as a contractual requirement to prove that certain functionality has been implemented. ATPs are often associated with milestones, and milestones are often associated with payments or further project funding. An ATP differs from a unit test because the ATP demonstrates that the functionality with respect to the whole line-item requirement has been implemented. For example, a unit test can determine whether the computation is correct. However, the ATP might validate that the user elements are provided in the user interface and that the user interface displays the result of the computation as specified by the requirement. These requirements are not covered by the unit test.

Automated User Interface Testing

An ATP might initially be written as a series of user interface (UI) interactions to verify that the requirements have been met. Regression testing of the application as it continues to evolve is applicable to unit testing as well as acceptance testing. Automated user interface testing is another tool completely separate from unit testing that saves time and manpower, while reducing testing errors. As with ATPs, unit tests in no way replace the value of automated user interface tests.

Usability and User Experience Testing

Unit tests, ATPs, and automated UI tests do not in any way replace usability testing—putting the application in front of users and getting their "user experience" feedback. Usability testing should not be about finding computational defects (bugs), and therefore is completely outside of the purview of unit tests.

Performance and Load Testing

Some unit test tools provide a means for measuring the performance of a method. For example, Visual Studio's test engine reports on execution time, and NUnit has attributes that can be used to verify that a method executes within an allotted time.

Ideally, a unit test tool for .NET languages should explicitly implement performance testing to compensate for just-in-time (JIT) code compilation the first time the code is executed.

Most load tests (and the related performance tests) are not suitable for unit tests. Certain forms of load tests can be done with unit testing as well, at least to the limitations of the hardware and operating system, such as:

- Simulating memory constraints.
- Simulating resource constraints.

However, these kinds of tests ideally require the support of the framework or OS API to simulate these kinds of loads for the application being tested. Forcing the entire OS to consume a large amount of memory, resources, or both, affects all the applications, including the unit test application. This is not a desirable approach.

Other types of load testing, such as simulating multiple instances of running an operation simultaneously, are not candidates for unit testing. For example, testing the performance of a web service with a load of one million transactions per minute is probably not possible using a single machine. While this kind of test can be easily written as a unit, the actual test would involve a suite of test machines. And in the end, you've only tested a very narrow behavior of the web service under very specific network conditions, which in no way actually represent the real world.

For this reason, performance and load testing have limited application with unit testing.

Chapter 3 Proving Correctness

The phrase "proving correctness" is normally used in the context of the veracity of a computation, but with regard to unit testing, proving correctness actually has three broad categories, only the second of which relates to computations themselves:

- Verifying that inputs to a computation are correct (method contract).
- Verifying that a method call results in the desired computational result (called the computational aspect), broken down into four typical processes:
    - Data transformation
    - Data reduction
    - State change
    - State correctness
- External error handling and recovery.

There are many aspects of an application in which unit testing usually cannot be applied to proving correctness. These include most user interface features such as layout and usability. In many cases, unit testing is not the appropriate technology for testing requirements and application behavior regarding performance, load, and so forth.

How Unit Tests Prove Correctness

Proving correctness involves:

- Verifying the contract.
- Verifying computational results.
- Verifying data transformation results.
- Verifying external errors are handled correctly.

Let’s look at some examples of each of these categories, their strengths, weaknesses, and problems that we might encounter with our code.

Prove Contract is Implemented

The most basic form of unit testing is to verify that the developer has written a method that clearly states the “contract” between the caller and the method being called. This usually takes the form of verifying that bad inputs to a method result in an exception being thrown. For example, a "divide by” method might throw an **`ArgumentOutOfRangeException`** if the denominator is 0:

```
public static int Divide(int numerator, int denominator)
```

```
{
  if (denominator == 0)
  {
    throw new ArgumentOutOfRangeException("Denominator cannot be 0.");
  }

  return numerator / denominator;
}

[TestMethod]
[ExpectedException(typeof(ArgumentOutOfRangeException))]
public void BadParameterTest()
{
  Divide(5, 0);
}
```

However, verifying that a method implements contract tests is one of the weakest unit tests one can write.

Prove Computational Results

A stronger unit test involves verifying that the computation is correct. It is useful to categorize your methods into one of the three forms of computation:

- Data reduction
- Data transformation
- State change

These determine the kinds of unit tests you might want to write for a particular method.

Data Reduction

The **`Divide`** method in the previous sample can be considered a form of data reduction. It takes two values and returns one value. To illustrate:

```
[TestMethod]
public void VerifyDivisionTest()
{
  Assert.IsTrue(Divide(6, 2) == 3, "6/2 should equal 3!");
}
```

This is illustrative of testing a method that reduces the inputs, usually, to one resulting output. This is the simplest form of useful unit testing.

Data Transformation

Data transformation unit tests tend to operate on sets of values. For example, the following is a test for a method that converts Cartesian coordinates to polar coordinates.

```
public static double[] ConvertToPolarCoordinates(double x, double y)
{
  double dist = Math.Sqrt(x * x + y * y);
  double angle = Math.Atan2(y, x);

  return new double[] { dist, angle };
}

[TestMethod]
public void ConvertToPolarCoordinatesTest()
{
  double[] pcoord = ConvertToPolarCoordinates(3, 4);
  Assert.IsTrue(pcoord[0] == 5, "Expected distance to equal 5");
  Assert.IsTrue(pcoord[1] == 0.92729521800161219, "Expected angle to be 53.130
      degrees");
}
```

This test verifies the correctness of the mathematical transformation.

List Transformations

List transformations should be separated into two tests:

- Verify that the core transformation is correct.
- Verify that the list operation is correct.

For example, from the perspective of unit testing, the following sample is poorly written because it incorporates both the data reduction and the data transformation:

```
public struct Name
{
  public string FirstName { get; set; }
  public string LastName { get; set; }
}

public List<string> ConcatNames(List<Name> names)
{
  List<string> concatenatedNames = new List<string>();

  foreach (Name name in names)
  {
    concatenatedNames.Add(name.LastName + ", " + name.FirstName);
  }

  return concatenatedNames;
}

[TestMethod]
public void NameConcatenationTest()
```

```
{
  List<Name> names = new List<Name>()
  {
    new Name() { FirstName="John", LastName="Travolta"},
    new Name() {FirstName="Allen", LastName="Nancy"}
  };

  List<string> newNames = ConcatNames(names);

  Assert.IsTrue(newNames[0] == "Travolta, John");
  Assert.IsTrue(newNames[1] == "Nancy, Allen");
}
```

This code is better unit tested by separating the data reduction from the data transformation:

```
public string Concat(Name name)
{
  return name.LastName + ", " + name.FirstName;
}

[TestMethod]
public void ContactNameTest()
{
  Name name = new Name() { FirstName="John", LastName="Travolta"};
  string concatenatedName = Concat(name);
  Assert.IsTrue(concatenatedName == "Travolta, John");
}
```

Lambda Expressions and Unit Tests

The Language-Integrated Query (LINQ) syntax is closely coupled with lambda expressions, which results in an easy-to-read syntax that makes life difficult for unit testing. For example, this code:

```
public List<string> ConcatNamesWithLinq(List<Name> names)
{
  return names.Select(t => t.LastName + ", " + t.FirstName).ToList();
}
```

is significantly more elegant than the previous examples, but it does not lend itself well to unit testing the actual “unit,” that is, the data reduction from a name structure to a single comma-delimited string expressed in the lambda function **`t => t.LastName + ", " + t.FirstName`**. To separate the unit from the list operation requires:

```
public List<string> ConcatNamesWithLinq(List<Name> names)
{
  return names.Select(t => Concat(t)).ToList();
}
```

We can see that unit testing can often require refactoring of the code to separate the units from other transformations.

State Change

Most languages are “stateful,” and classes often manage state. The state of a class, represented by its properties, is often a useful thing to test. Consider this class representing the concept of a connection:

```
public class AlreadyConnectedToServiceException : ApplicationException
{
  public AlreadyConnectedToServiceException(string msg) : base(msg) { }
}

public class ServiceConnection
{
  public bool Connected { get; protected set; }

  public void Connect()
  {
    if (Connected)
    {
      throw new AlreadyConnectedToServiceException("Only one connection at a time is
permitted.");
    }

    // Connect to the service.
    Connected = true;
  }

  public void Disconnect()
  {
    // Disconnect from the service.
    Connected = false;
  }
}
```

We can write unit tests to verify the various permitted and unpermitted states of the object:

```
[TestClass]
public class ServiceConnectionFixture
{
  [TestMethod]
  public void TestInitialState()
  {
    ServiceConnection conn = new ServiceConnection();
    Assert.IsFalse(conn.Connected);
  }
```

```
  [TestMethod]
  public void TestConnectedState()
  {
    ServiceConnection conn = new ServiceConnection();
    conn.Connect();
    Assert.IsTrue(conn.Connected);
  }

  [TestMethod]
  public void TestDisconnectedState()
  {
    ServiceConnection conn = new ServiceConnection();
    conn.Connect();
    conn.Disconnect();
    Assert.IsFalse(conn.Connected);
  }

  [TestMethod]
  [ExpectedException(typeof(AlreadyConnectedToServiceException))]
  public void TestAlreadyConnectedException()
  {
    ServiceConnection conn = new ServiceConnection();
    conn.Connect();
    conn.Connect();
  }
}
```

Here, each test verifies the correctness of the state of the object:

- When it is initialized.
- When instructed to connect to the service.
- When instructed to disconnect from the service.
- When more than one simultaneous connection is attempted.

State verification often reveals bugs in state management. Also see the following "Mocking Classes" for further improvements to the preceding example code.

Prove a Method Correctly Handles an External Exception

External error handling and recovery is often more important than testing whether your own code generates exceptions at the correct times. There are several reasons for this:

- You have no control over a physically separate dependency, whether it's a web service, database, or other separate server.
- You have no proof of the correctness of someone else's code, typically a third-party library.

- Third-party services and software may throw an exception because of a problem that your code is creating but not detecting (and would not necessarily be easy to detect). An example of this is, when deleting records in a database, the database throws an exception because of records in other tables referencing the records your program is deleting, thereby violating a foreign key constraint.

These kinds of exceptions are difficult to test because they require creating at least some error that would be typically generated by the service that you do not control. One way to do this is to "mock" the service; however, this is only possible if the external object is implemented with an interface, an abstract class, or virtual methods.

Mocking Classes

For example, the earlier code for the "ServiceConnection" class is not mockable. If you want to test its state management, you must physically create a connection to the service (whatever that is) that may or may not be available when running the unit tests. A better implementation might look like this:

```
public class MockableServiceConnection
{
  public bool Connected { get; protected set; }

  protected virtual void ConnectToService()
  {
    // Connect to the service.
  }

  protected virtual void DisconnectFromService()
  {
    // Disconnect from the service.
  }

  public void Connect()
  {
    if (Connected)
    {
      throw new AlreadyConnectedToServiceException("Only one connection at a time is
      permitted.");
    }

    ConnectToService();
    Connected = true;
  }

  public void Disconnect()
  {
    DisconnectFromService();
    Connected = false;
  }
}
```

Notice how this minor refactoring now allows you to write a mock class:

```
public class ServiceConnectionMock : MockableServiceConnection
{
  protected override void ConnectToService()
  {
    // Do nothing.
  }

  protected override void DisconnectFromService()
  {
    // Do nothing.
  }
}
```

which allows you to write a unit test that tests the state management regardless of the availability of the service. As this illustrates, even simple architectural or implementation changes can greatly improve the testability of a class.

Prove a Bug is Re-creatable

Your first line of defense in proving that the problem has been corrected is, ironically, proving that the problem exists. Earlier we saw an example of writing a test that proved that the Divide method checks for a denominator value of **`0`**. Let's say a bug report is filed because a user crashed the program when entering **`0`** for the denominator value.

Negative Testing

The first order of business is to create a test that exercises this condition:

```
[TestMethod]
[ExpectedException(typeof(DivideByZeroException))]
public void BadParameterTest()
{
  Divide(5, 0);
}
```

This test *passes* because we are proving that the bug exists by verifying that when the denominator is **`0`**, a **`DivideByZeroException`** is raised. These kinds of tests are considered "negative tests," as they *pass* when an error occurs. Negative testing is as important as positive testing (discussed next) because it verifies the existence of a problem before it is corrected.

Prove a Bug is Fixed

Obviously, we want to prove that a bug has been fixed. This is a "positive" test.

Positive Testing

We can now introduce a new test, one that will test that the code itself detects the error by throwing an **`ArgumentOutOfRangeException`**.

```
[TestMethod]
[ExpectedException(typeof(ArgumentOutOfRangeException))]
public void BadParameterTest()
{
  Divide(5, 0);
}
```

If we can write this test *before* fixing the problem, we will see that the test fails. Finally, after fixing the problem, our positive test passes, and the negative test now fails.

While this is a trivial example, it demonstrates two concepts:

- Negative tests—proving that something is repeatedly not working—are important in understanding the problem and the solution.
- Positive tests—proving that the problem has been fixed—are important not only to verify the solution, but also for repeating the test whenever a change is made. Unit testing plays an important role when it comes to regression testing.

Lastly, proving that a bug exists is not always easy. However, as a general rule of thumb, unit tests that require too much setup and mocking are an indicator that the code being tested is not isolated enough from external dependencies and might be a candidate for refactoring.

Prove Nothing Broke When Changing Code

It should be obvious that regression testing is a measurably useful outcome of unit testing. As code undergoes changes, bugs will be introduced that will be revealed if you have good code coverage in your unit tests. This effectively saves considerable time in debugging and more importantly, saves time and money when the programmer discovers the bug rather than the user.

Prove Requirements Are Met

Application development typically starts with a high-level set of requirements, usually oriented around the user interface, workflow, and computations. Ideally, the team reduces the *visible* set of requirements down to a set of programmatic requirements, which are *invisible* to the user, by their very nature.

The difference manifests in how the program is tested. Integration testing is typically at the *visible* level, while unit testing is at the finer grain of *invisible*, programmatic correctness testing. It is important to keep in mind that unit tests are not intended to replace integration testing; however, just as with high-level application requirements, there are low-level programmatic requirements that can be defined. Because of these programmatic requirements, it is important to write unit tests.

Let's take a Round method. The .NET Math.Round method will round up a number whose fractional component is greater than 0.5, but will round down when the fractional component is 0.5 or less. Let's say that is not the behavior we desire (for whatever reason), and we want to round up when the fractional component is 0.5 or greater. This is a computational requirement that should be able to be derived from a higher-level integration requirement, resulting in the following method and test:

```
public static int RoundUpHalf(double n)
{
  if (n < 0) throw new ArgumentOutOfRangeException("Value must be >= 0.");

  int ret = (int)n;
  double fraction = n - ret;

  if (fraction >= 0.5)
  {
    ++ret;
  }

  return ret;
}

[TestMethod]
public void RoundUpTest()
{
  int result1 = RoundUpHalf(1.5);
  int result2 = RoundUpHalf(1.499999);
  Assert.IsTrue(result1 == 2, "Expected 2.");
  Assert.IsTrue(result2 == 1, "Expected 1.");
}
```

A separate test for the exception should also be written.

Taking application-level requirements that are verified with integration testing and reducing them to lower-level computational requirements is an important part of the overall unit testing strategy as it defines clear computational requirements that the application must meet. If difficulty is encountered with this process, try to convert the application requirements into one of the three computational categories: data reduction, data transformation, and state change.

Chapter 4 Strategies for Implementing Unit Tests

Testing approaches depend on where you are in the project and your "budget," in terms of time, money, manpower, need, etc. Ideally, unit testing is budgeted into the development process, but realistically, we often encounter existing or legacy programs that have little or no code coverage but must be upgraded or maintained. The worst scenario is a product that is currently being developed but exhibits an increased number of failures during its development, again with little or no code coverage. As a product manager, either at the beginning of a development effort or as a result of being handed an existing application, it is important to develop a reasonable unit testing strategy. Remember that unit tests should provide measurable benefits to your project to offset the liability of their development, maintenance, and their own testing. Furthermore, the strategy that you adopt for your unit testing can affect the architecture of your application. While this is almost always a good thing, it may introduce unnecessary overhead for your needs.

Starting From Requirements

If you are starting a sufficiently complex application from a clean slate, and all that is in your hands is a set of requirements, consider the following guidance.

Prioritizing Computational Requirements

Prioritize the application's computational requirements to determine where the complexity lies. Complexity can be determined by discovering the number of states that a particular computation must accommodate, or it can be the result of a large set of input data required to perform the computation, or it could simply be algorithmically complex, such as doing failure case analysis on a satellite's redundancy ring. Also consider where code is likely to change in the future as the result of unknown changing requirements. While that sounds like it requires clairvoyance, a skilled software architect can categorize code into general purpose (solving a common problem), and domain specific (solving a specific requirement problem). The latter becomes a candidate for future change.

While writing unit tests for trivial functions is easy, fast, and gratifying in the number of test cases that the program churns through, they are the least cost-effective tests—they take time to write and, because they will most likely be written correctly to begin with and they most likely will not change over time, they are the least useful as the application's code base grows. Instead, focus your unit testing strategy on the code that is domain specific and complex.

Select an Architecture

One of the benefits of starting a project from a set of requirements is that you get to create the architecture (or select a third-party architecture) as part of the development process. Third-party frameworks that allow you to leverage architectures such as inversion of control (and the related concept of dependency injection), as well as formal architectures such as Model-View-Controller (MVC) and Model-View-ViewModel (MVVM) facilitate unit testing for the simple reason that a modular architecture is typically easier to unit test. These architectures separate out:

- The presentation (view).
- The model (responsible for persistence and data representation).
- The controller (where the computations should be occurring).

While some aspects of the model might be candidates for unit testing, most of the unit tests will likely be written against methods in the controller or view model, which is where the computations on the model or view are implemented.

Maintenance Phase

Unit testing can be of benefit even if you are involved in the maintenance of an application, one that either requires adding new features to an existing application or simply fixing bugs of a legacy application. There are several approaches one can take to an existing application and questions underlying those approaches that can determine the cost-effectiveness of unit testing:

- Do you write unit tests only for new features and bug fixes? Is the feature or bug fix something that will benefit from regression testing, or is it a one-time, isolated issue that is easier tested during integration testing?
- Do you start writing unit tests against existing features? If so, how do you prioritize which features to test first?
- Does the existing code base work well with unit testing or does the code first need refactoring to isolate code units?
- What setups or teardowns are needed for the feature or bug testing?
- What dependencies can be discovered about the code changes that may result in side effects in other code, and should the unit tests be broadened to test the behavior of dependent code?

Walking into the maintenance phase of a legacy application that lacks unit testing is not trivial—the planning, consideration, and investigation into the code may often require more resources than simply fixing the bug. However, the judicious use of unit testing can be cost-effective, and while this is not always easy to determine, it is worth the exercise, if for no other reason than to get a deeper understanding of the code base.

Determine Your Process

There are three strategies one can take with regard to the unit test process: “Test-Driven Development,” “Code First,” and, though it may seem antithetical to the theme of this book, the “No Unit Test” process.

Test-Driven Development

One camp is “Test-Driven Development,” summarized by the following workflow:

Given a computational requirement (see earlier section), first, write a stub for the method.

- If dependencies on other objects that are not yet implemented are required (objects passed in as parameters to the method or returned by the method), implement those as empty interfaces.
- If properties are missing, implement stubs for properties that are needed to verify the results.
- Write any setup or teardown test requirements.
- Write the tests. The reasons for writing any stubs *before* writing the test are: first, to take advantage of IntelliSense when writing the test; second, to establish that the code still compiles; and third, to ensure that the method being tested, its parameters, interfaces, and properties have all synchronized with regard to naming.
- Run the tests, verifying that they fail.
- Code the implementation.
- Run the tests, verifying that they succeed.

In practice, this is harder than it looks. It’s easy to fall prey to writing tests that are not cost-effective, and often, one discovers that the method being tested is not a sufficiently fine-grained unit to actually be a good candidate for a test. Perhaps the method is doing too much, requiring too much setup or teardown, or has dependencies on too many other objects that all must be initialized to a known state. These are all things that are more easily discovered when writing the code, not the test.

One advantage to a test-driven approach is that the process instills the discipline of unit testing and writing the unit tests first. It’s easy to determine if the developer is following the process. With practice, one can become facile at also making the process cost-effective.

Another advantage to a test-driven approach is that, by its nature, it enforces a kind of architecture. It would be absurd but doable to write a unit test that initializes a form, puts values into a control, and then calls a method that is expected to perform some computation on the values, as this code would require (actually found here):

```
private void btnCalculate_Click(object sender, System.EventArgs e)
{
  double Principal, AnnualRate, InterestEarned;
  double FutureValue, RatePerPeriod;
  int NumberOfPeriods, CompoundType;
```

```
  Principal = Double.Parse(txtPrincipal.Text);
  AnnualRate = Double.Parse(txtInterest.Text) / 100;

  if (rdoMonthly.Checked)
    CompoundType = 12;
  else if (rdoQuarterly.Checked)
    CompoundType = 4;
  else if (rdoSemiannually.Checked)
    CompoundType = 2;
  else
    CompoundType = 1;

  NumberOfPeriods = Int32.Parse(txtPeriods.Text);
  double i = AnnualRate / CompoundType;
  int n = CompoundType * NumberOfPeriods;

  RatePerPeriod = AnnualRate / NumberOfPeriods;
  FutureValue = Principal * Math.Pow(1 + i, n);
  InterestEarned = FutureValue - Principal;

  txtInterestEarned.Text = InterestEarned.ToString("C");
  txtAmountEarned.Text = FutureValue.ToString("C");
}
```

The preceding code is untestable as it is entangled with the event handler and the user interface. Rather, one could write the compound interest calculation method:

```
public enum CompoundType
{
  Annually = 1,
  SemiAnnually = 2,
  Quarterly = 4,
  Monthly = 12
}

private double CompoundInterestCalculation(
  double principal,
  double annualRate,
  CompoundType compoundType,
  int periods)
{
  double annualRateDecimal = annualRate / 100.0;
  double i = annualRateDecimal / (int)compoundType;
  int n = (int)compoundType * periods;
  double ratePerPeriod = annualRateDecimal / periods;
  double futureValue = principal * Math.Pow(1 + i, n);
  double interestEaned = futureValue - principal;
```

```
  return interestEaned;
}
```

which would then allow for a simple test to be written:

```
[TestMethod]
public void CompoundInterestTest()
{
  double interest = CompoundInterestCalculation(2500, 7.55, CompoundType.Monthly, 4);
  Assert.AreEqual(878.21, interest, 0.01);
}
```

Furthermore, by using parameterized testing, it would be straightforward to test each compound type, a range of years, and different interest and principal amounts.

The test-driven approach actually *facilitates* a more formalized development process by the discovery of actual testable units and isolating them from boundary-crossing dependencies.

Code First, Test Second

Coding first is more natural if only because that's the usual way applications are developed. The requirement and its implementation may also seem easy enough at first sight so that writing several unit tests seems like a poor use of time. Other factors such as deadlines can force a project into a "just get the code written so we can ship" development process.

The problem with the code-first approach is that it is easy to write code that requires the kind of test we saw earlier. Code first requires an active discipline to test the code that has been written. This discipline is incredibly difficult to achieve, especially as there is always the next new feature to implement.

It also requires intelligence, if you will, to avoid writing entangled, boundary-crossing code, and the discipline to do so. Who hasn't clicked on a button in the Visual Studio designer and coded the event's computation right there in the stub that Visual Studio creates for you? It's easy and because the tool is directing you in that direction, the naive programmer will think this is the right way of coding.

This approach requires careful consideration of the skills and discipline of your team, and requires closer monitoring of the team, especially during high-stress periods when disciplined approaches tend to break down. Granted, a test-driven discipline may also be thrown out as deadlines loom, but that tends to be a conscious decision to make an exception, whereas it can easily become the rule in a code first approach.

No Unit Tests

Just because you don't have unit tests doesn't mean you are throwing out testing. It may simply be that the testing emphasizes acceptance test procedures or integration testing.

Balancing Testing Strategies

A cost-effective unit testing process requires a balance between Test-Driven Development, Code First, Test Second, and “Test Some Other Way” strategies. The cost-effectiveness of unit testing should always be considered, as well as factors such as the experience of the developers on the team. As a manager, you may not want to hear that a test-driven approach is a good idea if your team is fairly green and you need the process to instill discipline and approach.

Chapter 5 Look before You Leap: The Cost of Unit Testing

The previous chapters have touched upon a variety of concerns and benefits of unit testing. This chapter is a more formalized look at the cost and benefits of unit testing.

Unit Test Code vs. Code Being Tested

Your unit test code is a separate entity from the code being tested, yet it shares many of the same issues required by your production code:

- Planning
- Development
- Testing (yes, unit tests must be tested)

In addition, unit tests may also:

- Have a larger code base than the production code.
- Need to be synchronized when production code changes.
- Tend to enforce architectural directions and implementation patterns.

Unit Test Code Base May Be Larger Than Production Code

When determining whether the tests can be written against a single method, one should consider:

- Does it validate the contract?
- Does the computation work correctly?
- Is the internal state of the object set correctly?
- Does it return the object to a “sane” state if an exception occurs?
- Are all code paths tested?
- What setup or teardown requirements does the method have?

One should realize that the line count of code to test even a simple method could be considerably larger than the line count of the method itself.

Maintaining Unit Tests

Changing the production code can often invalidate unit tests. Code changes roughly fall into two categories:

- New code or changes to existing code that enhance the user experience.
- Significant restructuring to support requirements that the existing architecture does not support.

The former usually carries little or no maintenance requirements on existing unit tests. The latter, however, often requires considerable rework of unit tests, depending on the complexity of the change:

- Refactoring concrete class parameters to interfaces or abstract classes.
- Refactoring the class hierarchy.
- Replacing a third-party technology with another.
- Refactoring the code to be asynchronous or support tasks.
- Others:
    - Example: Changing from a concrete database class such as SqlConnection to IDbConnection, so that the code supports different databases and requires reworking the unit tests that call methods that were dependent upon concrete classes for their parameters.
    - Example: Modifying a model to utilize a standard serialization format, such as XML, rather than a custom serialization methodology.
    - Example: Changing from an in-house ORM to a third-party ORM such as Entity Framework may require considerable changes to the setup or teardowns of unit tests.

Does Unit Testing Enforce an Architecture Paradigm?

As mentioned previously, unit testing, especially in a test-driven process, enforces certain minimal architecture and implementation paradigms. To further support the ease of setting up or tearing down of some areas of the code, unit testing may also benefit from more complex architecture considerations, such as inversion of control.

Unit Test Performance

At a minimum, most classes should facilitate mocking of any object. This can significantly improve the performance of the tests—for example, testing a method that performs foreign key integrity checks (rather than relying on the database to report errors later) shouldn't require a complex setup or teardown of the test scenario in the database itself. Furthermore, it shouldn't require the method to actually query the database. These are all performance hits to the test and add dependencies on a live, authenticated connection with the database, and therefore may not handle another workstation running exactly the same test at the same time. Instead, by mocking the database connection, the unit test can easily set up the scenario in memory and pass the connection object as an interface.

However, simply mocking a class is not necessarily the best practice either—it might be better to refactor the code so that all the information the method needs is obtained separately, separating out the acquisition of the data from the computation of the data. Now, the computation can be performed without mocking the object that is responsible for acquiring the data, which further simplifies the test setup.

Mitigating Costs

There are a couple of cost-mitigating strategies that should be considered.

Correct Inputs

The most effective way of reducing the cost of unit testing is to avoid having to write the test. While this sounds obvious, how is this achieved? The answer is to ensure that the data being passed to the method is correct—in other words—correct input, correct output (the converse of "garbage in, garbage out"). Yes, you probably still want to test the computation itself, but if you can guarantee that the contract is met by the caller, there is no particular need to test the method to see if it handles incorrect parameters (violations of the contract).

This is a little bit of a slippery slope because you have no idea how the method might be called in the future—in fact, you may want the method to still validate its contract, but *in the context* in which it is currently used, if you can guarantee that the contract is always met, then there is no real point in writing tests against the contract.

How do you ensure correct inputs? For values that come from a user interface, appropriately filtering and controlling the user's interaction to pre-filter the values is one approach. A more sophisticated approach is to define specialized types rather than relying on general purpose types. Consider the Divide method described earlier:

```
public static int Divide(int numerator, int denominator)
{
  if (denominator == 0)
  {
    throw new ArgumentOutOfRangeException("Denominator cannot be 0.");
  }
```

```
  return numerator / denominator;
}
```

If the denominator was a specialized type that guaranteed a non-zero value:

```
public class NonZeroDouble
{
  protected int val;

  public int Value
  {
    get { return val; }
    set
    {
      if (value == 0)
      {
        throw new ArgumentOutOfRangeException("Value cannot be 0.");
      }

      val = value;
    }
  }
}
```

the Divide method would never need to test for this case:

```
/// <summary>
/// An example of using type specificity to avoid a contract test.
/// </summary>
public static int Divide(int numerator, NonZeroDouble denominator)
{
  return numerator / denominator.Value;
}
```

When one considers that this improves the type specificity of the application and establishes (hopefully) reusable types, one realizes how this avoids having to write a slew of unit tests because code often utilizes types that are too general.

Avoiding Third-party Exceptions

Ask yourself—should my method be responsible for handling exceptions from third parties, such as web services, databases, network connections, etc.? It can be argued that the answer is “no.” Granted, this requires some further up-front work—the third-party (or even framework) API needs a wrapper that handles the exception and an architecture in which the internal state of the application can be rolled back when an exception occurs and should probably be implemented. However, these are probably worthwhile improvements to the application anyway.

Avoid Writing the Same Tests for Each Method

The earlier examples—correct inputs, specialized type systems, avoiding third-party exceptions—all push the problem to more general purpose and possibly reusable code. This helps to avoid writing the same or similar contract validation, exception-handling unit tests, and allows you to focus instead on tests that validate what the method should be doing under normal conditions, that being the computation itself.

Cost Benefits

As mentioned previously, there are definite cost benefits to unit testing.

Coding to the Requirement

One of the obvious benefits is the process in formalizing the internal code requirements from external usability/process requirements. As one goes through this exercise, direction with the overall architecture is typically a side benefit. More concretely, developing a suite of tests that a specific requirement is met from the *unit* perspective (rather than the *integration test* perspective) is objective proof that the code implements the requirement.

Reduces Downstream Errors

Regression testing is another (often measurable) benefit. As the code base grows, verifying that existing code still works as intended saves considerable manual testing time and avoids the "oops, we didn't test for that" scenario. Furthermore, when an error is reported, it can be corrected immediately, often saving other members of the team the considerable headache of wondering why something that they were relying on is suddenly not working correctly.

Test Cases Provide a Form of Documentation

Unit tests verify not just that a method handles itself correctly when given bad inputs or third-party exceptions (as described earlier, try to reduce these kinds of tests), but also how the method is expected to behave under normal conditions. This provides valuable documentation to developers, especially new team members—via the unit test they can easily glean the setup requirements and the use cases. If your project undergoes a significant architectural refactoring, the new unit tests can be used to guide developers in reworking their dependent code.

Enforcing an Architecture Paradigm Improves the Architecture

As described earlier, a more robust architecture through the use of interfaces, inversion of control, specialized types, etc.—all of which facilitate unit testing—*also* improve the robustness of the application. Requirements change, even during development, and a well-thought-out architecture can handle those changes considerably better than an application that has no or little architectural consideration.

Junior Programmers

Rather than handing a junior programmer a high-level requirement to be implemented at the skill level of the programmer, you can instead guarantee a higher level of code and success (and provide a teaching experience) by having the junior programmer code the implementation *against the test* rather than the requirement. This eliminates a lot of bad practices or guesswork that a junior programmer ends up implementing (we've all been there) and reduces the rework a more senior developer needs to do in the future.

Code Reviews

There are several kinds of code reviews. Unit tests can reduce the amount of time spent reviewing code for architectural issues because they tend to enforce architecture. Furthermore, unit tests validate the computation and can also be used to validate all code paths for a given method. This makes code reviews almost unnecessary—the unit test becomes a self-review of the code.

Converting Requirements to Tests

An interesting side effect of converting external usability or process requirements to formalized code tests (and their supporting architecture) is that:

- Problems with the requirements are often discovered.
- Architectural requirements are brought to light.
- Assumptions and other gaps in the requirements are identified.

These discoveries, as a result of the unit test process, identify issues earlier in the development process, which usually helps to reduce confusion, rework, and therefore, reduces cost.

Chapter 6 How Does Unit Testing Work?

A unit test engine in a reflective language (such as any .NET language) has three parts:

- Loading assemblies that contain the tests.
- Using reflection to find the test methods.
- Invoking the methods and validating the results.

This chapter provides code examples of how this works, putting together a simple unit test engine. If you're not interested in the under-the-hood investigation of unit test engines, feel free to skip this chapter.

The code here assumes that we are writing a test engine against the Visual Studio unit test attributes defined in the Microsoft.VisualStudio.QualityTools.UnitTestFramework assembly. Other unit test engines may use other attributes for the same purposes.

Loading Assemblies

Architecturally, your unit tests should either reside in a separate assembly from the code being tested, or at a minimum, should only be included in the assembly if it is compiled in "Debug" mode. The benefit of putting the unit tests in a separate assembly is that you can also unit test the non-debug, optimized production version of the code.

That said, the first step is to load the assembly:

```
static bool LoadAssembly(string assemblyFilename, out Assembly assy, out string issue)
{
  bool ok = true;
  issue = String.Empty;
  assy = null;

  try
  {
    assy = Assembly.LoadFile(assemblyFilename);
  }
  catch (Exception ex)
  {
    issue = "Error loading assembly: " + ex.Message;
    ok = false;
  }

  return ok;
}
```

Note that professional unit test engines load assemblies into a separate application domain so that the assembly can be unloaded or reloaded without restarting the unit test engine. This also allows the unit test assembly and dependent assemblies to be recompiled without shutting down the unit test engine first.

Using Reflection to Find Unit Test Methods

The next step is to reflect over the assembly to identify the classes that are designated as a "test fixture," and within those classes, to identify the test methods. A basic set of four methods support the minimum unit test engine requirements, the discovery of test fixtures, test methods, and exception handling attributes:

```
/// <summary>
/// Returns a list of classes in the provided assembly that have a "TestClass" attribute.
/// </summary>
static IEnumerable<Type> GetTestFixtures(Assembly assy)
{
  return assy.GetTypes().Where(t => t.GetCustomAttributes(typeof(TestClassAttribute), false).Length == 1);
}

/// <summary>
/// Returns a list of methods in the test fixture that are decorated with the "TestMethod" attribute.
/// </summary>
static IEnumerable<MethodInfo> GetTestMethods(Type testFixture)
{
  return testFixture.GetMethods().Where(m => m.GetCustomAttributes(
      typeof(TestMethodAttribute), false).Length == 1);
}

/// <summary>
/// Returns a list of specific attributes that may be decorating the method.
/// </summary>
static IEnumerable<AttrType> GetMethodAttributes<AttrType>(MethodInfo method)
{
  return method.GetCustomAttributes(typeof(AttrType), false).Cast<AttrType>();
}

/// <summary>
/// Returns true if the method is decorated with an "ExpectedException" attribute while exception type is
the expected exception.
/// </summary>
static bool IsExpectedException(MethodInfo method, Exception expectedException)
{
  Type expectedExceptionType = expectedException.GetType();

  return GetMethodAttributes<ExpectedExceptionAttribute>(method).
            Where(attr=>attr.ExceptionType == expectedExceptionType).Count() != 0;
}
```

Invoking Methods

Once this information has been compiled, the engine invokes the test methods within a try-catch block (we don't want the unit test engine itself crashing):

```
static void RunTests(Type testFixture, Action<string> result)
{
  IEnumerable<MethodInfo> testMethods = GetTestMethods(testFixture);

  if (testMethods.Count() == 0)
  {
    // Don't do anything if there are no test methods.
    return;
  }

  object inst = Activator.CreateInstance(testFixture);

  foreach (MethodInfo mi in testMethods)
  {
    bool pass = false;

    try
    {
      // Test methods do not have parameters.
      mi.Invoke(inst, null);
      pass = true;
    }
    catch (Exception ex)
    {
      pass = IsExpectedException(mi, ex.InnerException);
    }
    finally
    {
      result(testFixture.Name + "." + mi.Name + ": " + (pass ? "Pass" : "Fail"));
    }
  }
}
```

Finally, we can put this code together into a simple console application that takes the unit test assembly as parameter results in a usable, but simple engine:

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Reflection;
using System.Text;
```

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace SimpleUnitTestEngine
{
  class Program
  {
    static void Main(string[] args)
    {
      string issue;

      if (!VerifyArgs(args, out issue))
      {
        Console.WriteLine(issue);
        return;
      }

      Assembly assy;

      if (!LoadAssembly(args[0], out assy, out issue))
      {
        Console.WriteLine(issue);
        return;
      }

      IEnumerable<Type> testFixtures = GetTestFixtures(assy);

      foreach (Type testFixture in testFixtures)
      {
        RunTests(testFixture, t => Console.WriteLine(t));
      }
    }

    static bool VerifyArgs(string[] args, out string issue)
    {
      bool ok = true;
      issue = String.Empty;

      if (args.Length != 1)
      {
        issue = "Usage: SimpleUnitTestEngine <assembly filename>";
        ok = false;
      }
      else
      {
        string assemblyFilename = args[0];

        if (!File.Exists(assemblyFilename))
        {
          issue = "The filename '" + args[0] + "' does not exist.";
```

```
          ok = false;
        }
      }

      return ok;
    }

... the rest of the code ...
```

The result of running this simple test engine is displayed in a console window, for example:

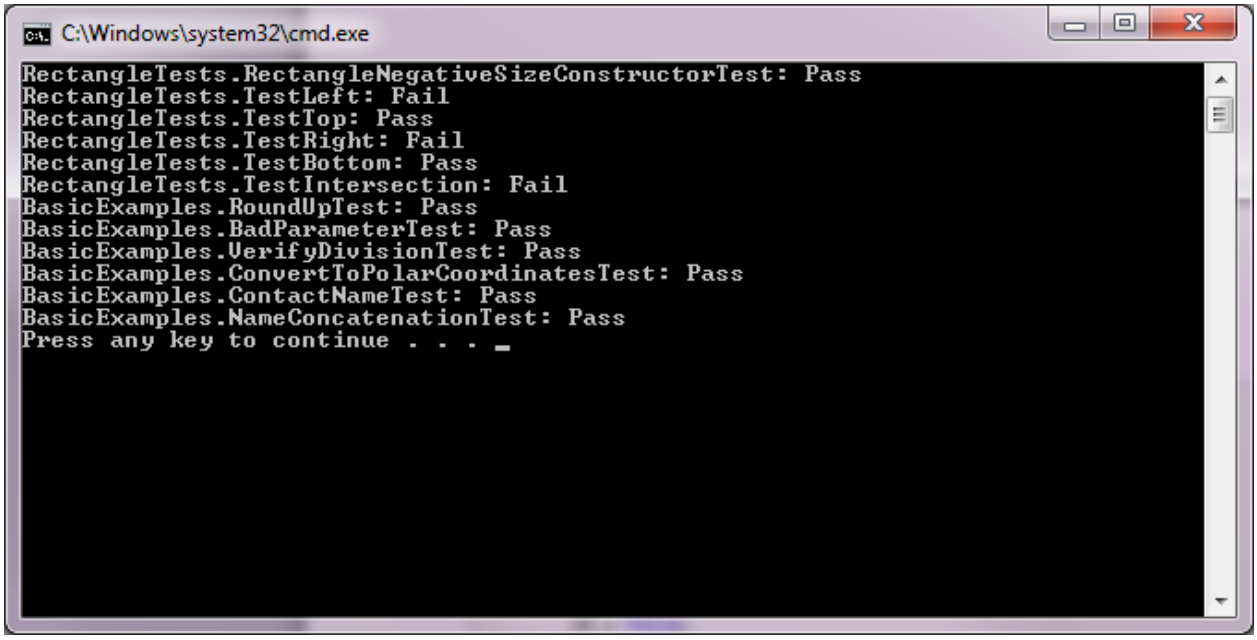


Figure 3: Our Simple Test Engine Console Results

Chapter 7 Common Unit Test Tools

NUnit

NUnit (http://www.nunit.org/) was originally ported from JUnit as an open-source unit test engine providing a rich suite of test fixture, method, and variable attributes, as well as test assertions. The documentation for all versions of NUnit can be found at http://www.nunit.org/index.php?p=documentation. NUnit is still being maintained. The latest stable release at the time of this writing is version 2.6.2, released on October 22, 2012.

CSUnit

CSUnit (http://www.csunit.org/) is a lighter-weight unit test engine. Note that it does not appear to be maintained, as the last release was in March 2009. It offers a minimal but functional set of attributes to use for defining test fixtures and test methods.

Visual Studio Test Project

Visual Studio provides the ability to create test projects directly in the IDE. One of the issues the author discovered is that the user interface has changed between VS 2008 and VS 2012:

Visual Studio 2008 Test Results UI

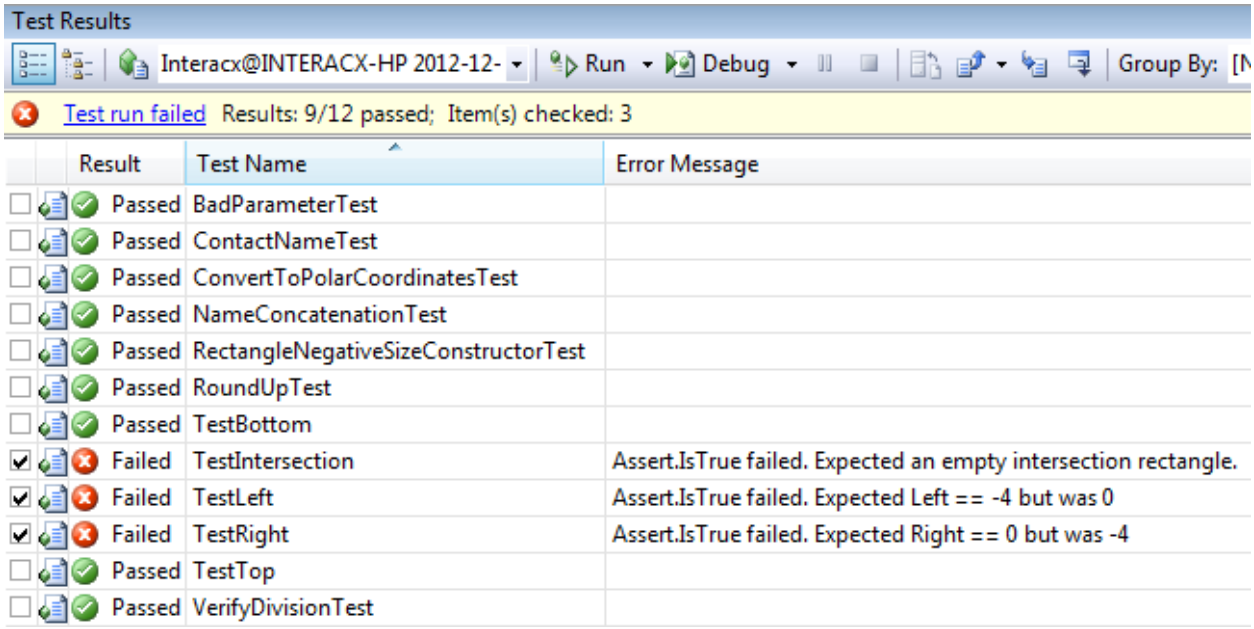


Figure 4: VS2008 Test Results UI

In VS2008, the test run executes very quickly and displays an easy-to-read list of test results and, for failed tests, the error message providing information as to why the test failed.

Also, when the test project is the active project, running the tests (or debugging them) is the same as with any other application—you can run them with Ctrl+F5 (run) or F5 (debug).

Visual Studio 2012 Test Results UI

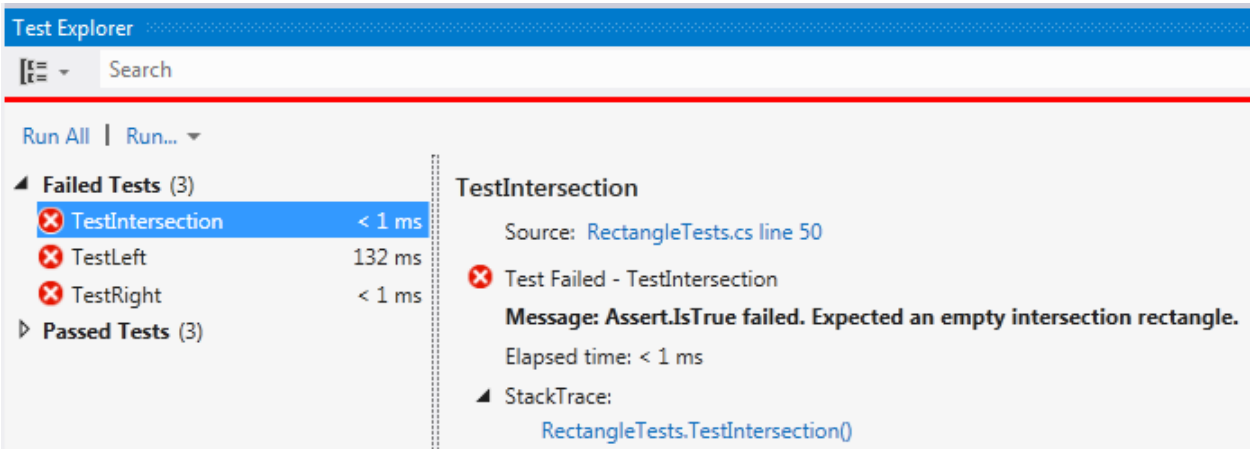


Figure 5: VS2012 Test Results UI

In VS2012, the test runner takes several seconds to initialize. Worse, the messages associated with a test result are obtained *by clicking on the failure.* The additional click requirement is a significant usability issue. Because of the changes in how failures are displayed, Visual Studio 2008 has been used for screenshots throughout this book.

Lastly, the test runner is no longer initiated through the same shortcut keys as a regular application. Instead, the developer must use Ctrl+R, A to run the tests. There is no keyboard shortcut mapped to debugging the unit tests.

Visual Studio and NUnit Integration

Microsoft provides integration tools for NUnit for both Visual Studio 2010 (http://visualstudiogallery.msdn.microsoft.com/c8164c71-0836-4471-80ce-633383031099) and 2012 (http://visualstudiogallery.msdn.microsoft.com/6ab922d0-21c0-4f06-ab5f-4ecd1fe7175d), though at the time of this writing, the Visual Studio 2012 NUnit test adapter is a beta 3-2.

Other Unit Test Tools

There are a few other test engines worth mentioning here.

MSTest

MSTest (http://msdn.microsoft.com/en-us/library/ms182489 is the command-line version of Microsoft's test runner.

MbUnit/Gallio

MbUnit (http://www.mbunit.com/) and Gallio (http://gallio.org/) are closely related. Gallio is a test automation platform allowing you to integrate a variety of test frameworks and reporting tools. This is a sophisticated tool that is worth exploring once you become familiar with unit testing principles and other engines.

Microsoft Test Manager

Microsoft Test Manager (http://msdn.microsoft.com/en-us/library/jj635157.aspx) is a tool for planning, managing, and executing tests, either manually or automatically. Microsoft Test Manager integrates with bug tracking, allows writing notes associated with your tests, and allows you to configure virtual lab machines that can be reset to a known state before tests are run.

FsUnit

FsUnit (https://github.com/dmohl/FsUnit) is a test engine that facilitates working with the F# language.

Integration Testing Frameworks

Unit testing is designed to validate the correctness of computational code units. Integration testing is designed to test the behavior of the user interface.

NBehave

For readers familiar with Ruby, NBehave (http://nbehave.tumblr.com/) is an early prototype of features similar to Cucumber (http://cukes.info/) in that one writes behavioral tests. Tests are written in natural language, for example (both of these examples come from the github NBehave website https://github.com/nbehave/NBehave/wiki/Getting%20Started):

```
Scenario: Login
  Given I am not logged in
  When I log in as Morgan with a password SecretPassw0rd
  Then I should see a message, "Welcome, Morgan!"
```

and methods implement the phrases (which can of course be re-used):

```
[Given("I am not logged in")]
public void LogOut()
{
  . . .
}
```

Keep in mind that this is not unit testing, rather, it facilitates a paradigm called Behavior-Driven Development (http://en.wikipedia.org/wiki/Behavior-driven_development) and is mentioned here to illustrate other forms of testing that complement unit testing.

Chapter 8 Testing Basics

A brief discussion of the fundamentals of testing is relevant to unit testing.

First, it is assumed that two other vital pieces of any software development project are in place:

- A source control system, such as CVS, SVN, Git, Mercurial, etc.
- A bug tracking system, such as Bugzilla, FogBugz, etc.

Microsoft provides integration of these pieces with Visual Source Safe, SharePoint, and other products.

If you do not have *both* of these pieces already in place, stop now. These are vital components of a good solid unit testing process.

So You Have a Bug

If you discover a bug, either through using the application or a failed unit test:

- Report the bug in the bug tracker and assign it to someone.

If the bug is not reported by a unit test but was discovered through using the code (either another developer or a user):

- Write the unit test that verifies how to create the issue (see “negative testing” described earlier).

With your source control system:

- Consider creating a branch specifically for this bug fix.

Then:

- Write the unit test that describes how the method should correctly behave.
- Fix the bug and retest.

With your source control system:

- Check in the changes, referring to the bug number assigned by the bug tracker.
- If you added new unit tests, check these in, again referring to the bug number.
- If you branched the code, decide when you want to merge the changes into the main branch.

Tracking and Reporting

By working with a bug-tracking and version-control system, anyone (from manager to junior programmer) can easily review:

- The changes made to the code base.
- The unit tests to create the problem.
- The unit tests that fix the problem.

This makes life a lot easier for the developer team, the management, and even the end-user if your project is set up such that the end-user is interested in this level of detail. (For example, if software is developed in-house for another department, others might be interested in knowing when a bug is fixed.)

Integration with other technologies such as SharePoint can facilitate the communication between team members, managers, and other departments. Also, using sophisticated source control tools such as Perforce and continuous integration tools such as CruiseControl.NET will further integrate and automate the use of source control with unit testing, integration testing, reporting tools, and so forth.

Chapter 9 Unit Testing with Visual Studio

This chapter will discuss writing a unit test, covering the common attributes and forms of assertion found in unit test classes and methods.

Basic Unit Test Structure

A unit test is comprised of two things:

- A class representing the test fixture.
- Methods in the class representing the unit tests.

Visual Studio will automatically create a stub for a test project, which is where we will start.

Creating a Unit Test Project in Visual Studio

Unit tests are typically placed in a separate project (resulting in a distinct assembly) from your application code. In Visual Studio 2008 or 2012, you can create a unit test project by right-clicking on the solution and selecting **Add** followed by **New Project** from the pop-up menu:

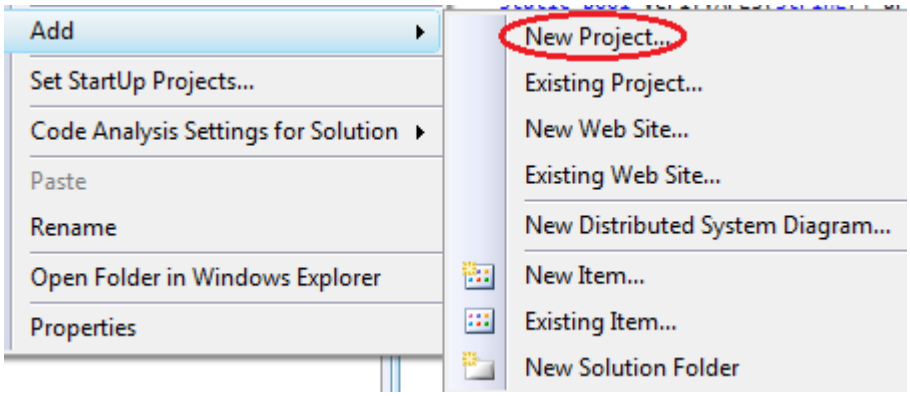


Figure 6: Adding a New Project

From the dialog box that appears, select a **Test** project:

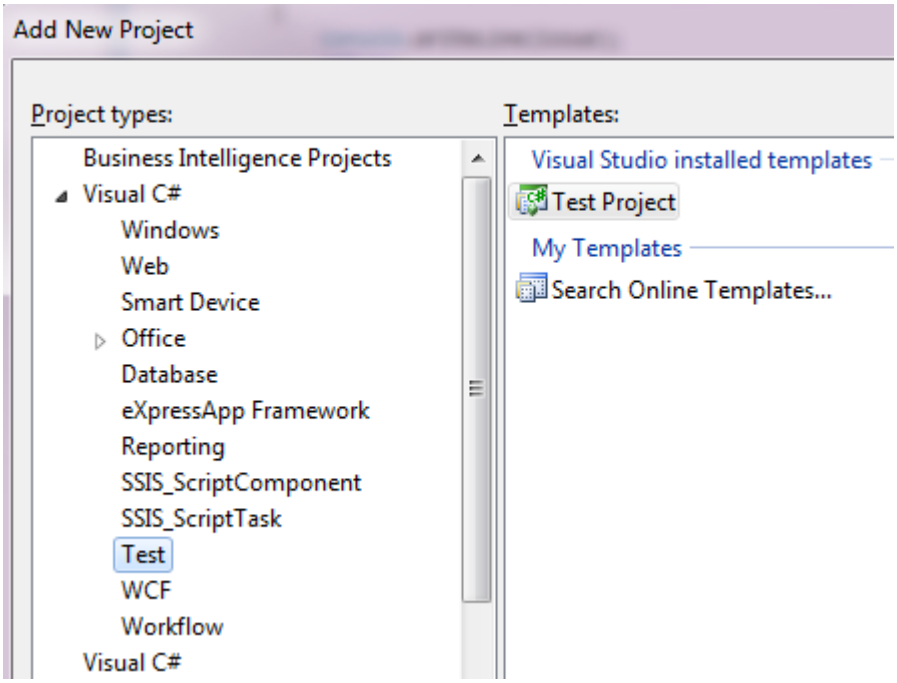


Figure 7: VS2008 New Test Project

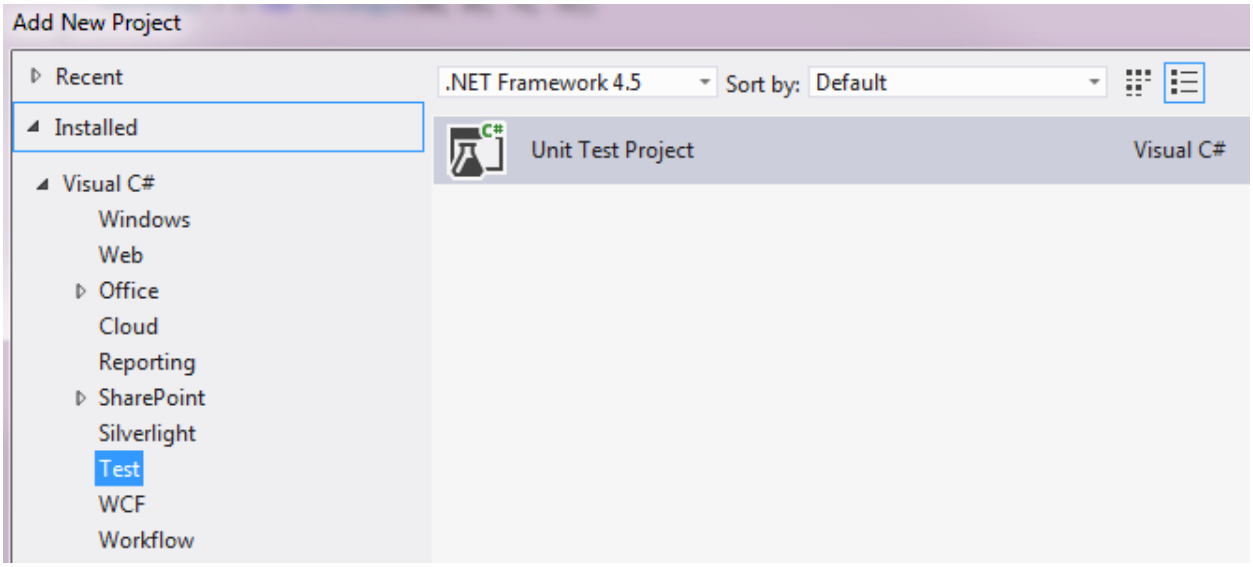


Figure 8: VS2012 New Test Project

Visual Studio 2008 will create a stub file, “UnitTest1.cs” (if you selected the C# language), with a variety of helpful comments in the stub. Visual Studio 2012 creates a much terser stub:

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace VS2012UnitTestProject1
{
```

```
  [TestClass]
  public class UnitTest1
  {
    [TestMethod]
    public void TestMethod1()
    {
    }
  }
}
```

For Visual Studio 2008 Users

Visual Studio 2008 will also create a TestContext class—this no longer exists in VS2012 and we will ignore it—use the previous stub from VS2012 instead.

Also, delete the “ManualTest1.mht” file, otherwise you will be prompted to select test results and enter test notes manually.

Test Fixtures

Notice that the class is decorated with the attribute TestClass. This defines the test fixture—a collection of test methods.

Test Methods

Notice that the method is decorated with the attribute TestMethod. This defines a method, which the test fixture will run.

The Assert Class

The assert class defines the following static methods that can be used to verify a method’s computation:

- AreEqual/AreNotEqual
- AreSame/AreNotSame
- IsTrue/IsFalse
- IsNull/IsNotNull
- IsInstanceOfType/IsNotInstanceOfType

Many of these assertions are overloaded and it is recommended that you review the full documentation that Microsoft provides.

Fundamentals of Making an Assertion

Note that the following examples use VS2008.

Assertions are the backbone of every test. There are a variety of assertions one can make regarding the results of a test. To begin with, we'll write a simple assertion that states "one equals one," in other words, a truism:

```
[TestClass]
public class UnitTest1
{
  [TestMethod]
  public void TestMethod1()
  {
    Assert.IsTrue(1 == 1);
  }
}
```

Run the test, which should result in "Passed":

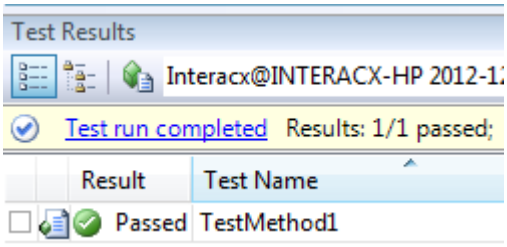


Figure 9: Simple Assertion

AreEqual/AreNotEqual

The AreEqual and AreNotEqual methods compare:

- objects
- doubles
- singles
- strings
- typed data

They take the form of comparing the expected (the first parameter) with the actual (the second parameter) value. With regard to single and double values, "within a certain accuracy" can be specified. Lastly, all the overloads have the option to display a message (optionally formatted) if the assertion fails.

With regard to object equality, this method compares whether the instances are identical:

```
public class AnObject
{
}

[TestMethod]
public void ObjectEqualityTest()
{
  AnObject object1 = new AnObject();
  AnObject object2 = new AnObject();

  Assert.AreNotEqual(object1, object2);
}
```

The preceding test passes, as object1 and object2 are not equal. However, if the class overrides the Equals method, then the equality is based on the comparison made by the Equals method implemented in the class. For example:

```
public class AnObject
{
  public int SomeValue { get; set; }

  public override bool Equals(object obj)
  {
    return SomeValue == ((AnObject)obj).SomeValue;
  }
}

[TestMethod]
public void ObjectEqualityTest()
{
  AnObject object1 = new AnObject() { SomeValue = 1 };
  AnObject object2 = new AnObject() { SomeValue = 1 } ;

  Assert.AreEqual(object1, object2);
}
```

AreSame/AreNotSame

These two methods verify that the *instances* are the same (or not). For example:

```
[TestMethod]
public void SamenessTest()
{
  AnObject object1 = new AnObject() { SomeValue = 1 };
  AnObject object2 = new AnObject() { SomeValue = 1 };
```

```
  Assert.AreNotSame(object1, object2);
}
```

Even though the class AnObject overrides the Equals operator, the preceding test passes because the *instances* of the two objects are not the same.

IsTrue/IsFalse

These two methods allow you to test the truth of a value comparison. From a readability perspective, the IsTrue and IsFalse methods are typically used for *value* comparisons, whereas AreEqual and AreSame are typically used to compare *instances* (objects).

For example:

```
[TestMethod]
public void IsTrueTest()
{
  AnObject object1 = new AnObject() { SomeValue = 1 };

  Assert.IsTrue(object1.SomeValue == 1);
}
```

This verifies that value of a property.

IsNull/IsNotNull

These two tests verify whether an object is null or not:

```
[TestMethod]
public void IsNullTest()
{
  AnObject object1 = null;

  Assert.IsNull(object1);
}
```

IsInstanceOfType/IsNotInstanceOfType

These two methods verify that an object is an instance of a specific type (or not). For example:

```
public class AnotherObject
{
}
```

```
[TestMethod]
public void TypeOfTest()
{
  AnObject object1 = new AnObject();

  Assert.IsNotInstanceOfType(object1, typeof(AnotherObject));
}
```

Inconclusive

The Assert.Inconclusive method can be used to specify that either the test or the functionality behind the test has not yet been implemented and therefore the test is inconclusive.

What Happens When An Assertion Fails?

With regard to Visual Studio unit testing, when an assertion fails, the Assert method throws an AssertFailedException. This exception should never be handled by your test code.

Other Assertion Classes

There are two other assertion classes:

- CollectionAssert
- StringAssert

As their names imply, these assertions operate on collections and strings, respectively.

Collection Assertions

These methods are implemented in the Microsoft.VisualStudio.TestTools.UnitTesting.CollectionAssert class. Note that the collection parameter in these methods expects the collection to implement ICollection (contrast this with NUnit, which expects IEnumerable).

AllItemsAreInstanceOfType

This assertion verifies that objects in a collection are of the same type, which includes derived types of the expected type, as illustrated here:

```
public class A { }
public class B : A { }

[TestClass]
public class CollectionTests
```

```
{
  [TestMethod]
  public void InstancesOfTypeTest()
  {
    List<Point> points = new List<Point>() { new Point(1, 2), new Point(3, 4) };
    List<object> items = new List<object>() { new B(), new A() };

    CollectionAssert.AllItemsAreInstancesOfType(points, typeof(Point));
    CollectionAssert.AllItemsAreInstancesOfType(items, typeof(A));
  }
}
```

AllItemsAreNotNull

This assertion verifies that objects in the collection are not null.

AllItemsAreUnique

This test ensures that objects in a collection are unique. If comparing structures:

```
[TestMethod]
public void AreUniqueTest()
{
  List<Point> points = new List<Point>() { new Point(1, 2), new Point(1, 2) };
  CollectionAssert.AllItemsAreUnique(points);
}
```

the structures are compared by value, not by instance—the preceding test fails. However, even if the class overrides the Equals method:

```
public class AnObject
{
  public int SomeValue { get; set; }

  public override bool Equals(object obj)
  {
    return SomeValue == ((AnObject)obj).SomeValue;
  }
}
```

this test passes:

```
[TestMethod]
public void AreUniqueObjectsTest()
{
  List<object> items = new List<object>()
  {
    new AnObject() { SomeValue = 1 },
```

```
    new AnObject() { SomeValue = 1 }
  };

  CollectionAssert.AllItemsAreUnique(items);
}
```

AreEqual/AreNotEqual

These tests assert that two collections are equal. The methods include overloads that allow you to provide a comparator method. If an object overrides the Equals method, that method will be used to determine equality. For example:

```
[TestMethod]
public void AreEqualTest()
{
List<object> itemList1 = new List<object>()
  {
    new AnObject() { SomeValue = 1 },
    new AnObject() { SomeValue = 2 }
  };

  List<object> itemList2 = new List<object>()
  {
    new AnObject() { SomeValue = 1 },
    new AnObject() { SomeValue = 2 }
  };

  CollectionAssert.AreEqual(itemList1, itemList2);
}
```

These two collections are equal because the class AnObject overrides the Equals method (see previous example).

Note that to pass the assertion, the lists must be of the same length and are considered not equal if the lists are identical except in a different order. Compare this with the AreEquivalent assertion described next.

AreEquivalent/AreNotEquivalent

This assertion compares two lists and considers lists of equal items to be equivalent regardless of order. Unfortunately, there appears to be a bug in the implementation, as this test fails:

```
[TestMethod]
public void AreEqualTest()
{
List<object> itemList1 = new List<object>()
  {
    new AnObject() { SomeValue = 1 },
    new AnObject() { SomeValue = 2 }
```

```
  };
  List<object> itemList2 = new List<object>()
  {
    new AnObject() { SomeValue = 2 },
    new AnObject() { SomeValue = 1 }
  };

  CollectionAssert.AreEquivalent(itemList1, itemList2);
}
```

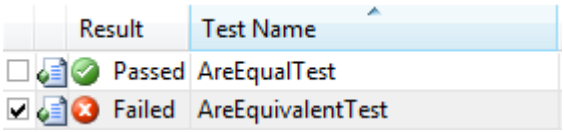


Figure 10: Visual Studio's AreEquivalent Bug

with the error message:

```
CollectionAssert.AreEquivalent failed. The expected collection contains 1
occurrence(s) of <UnitTestExamplesVS2008.AnObject>. The actual collection contains 0
occurrence(s).
```

Whereas NUnit's implementation of this assertion passes:

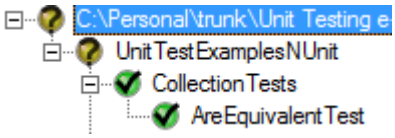


Figure 11: NUnit's AreEquivalent Works Correctly

Contains/DoesNotContain

This assertion verifies that an object is contained in a collection:

```
[TestMethod]
public void ContainsTest()
{
  List<object> itemList = new List<object>()
  {
    new AnObject() { SomeValue = 1 },
    new AnObject() { SomeValue = 2 }
  };

  CollectionAssert.Contains(itemList, new AnObject() { SomeValue = 1 });
}
```

using the Equals method (if overridden) to perform the equality test.

IsSubsetOf/IsNotSubsetOf

This assertion verifies that the first parameter (the subset) is contained in the second parameter's collection (the superset).

```
[TestMethod]
public void SubsetTest()
{
  string subset = "abc";
  string superset = "1d2c3b4a";

  CollectionAssert.IsSubsetOf(subset.ToCharArray(), superset.ToCharArray());
}
```

Note that the subset test does not test for order or sequence—it simply tests whether the items in the subset list are contained in the superset.

String Assertions

These methods are implemented in the Microsoft.VisualStudio.TestTools.UnitTesting.StringAssert class:

- Contains
- Matches/DoesNotMatch
- StartsWith/EndsWith

These are discussed next.

Contains

The Contains method asserts that the subset (note that this is the second parameter) is contained in the string (the first parameter). For example, this test passes:

```
[TestClass]
public class StringTests
{
  [TestMethod]
  public void ContainsTest()
  {
    string subset = "abc";
    string superset = "123abc456";

    StringAssert.Contains(superset, subset);
  }
}
```

Matches/DoesNotMatch

This method asserts that the string (the first parameter) matches the regex pattern provided in the second parameter.

StartsWith/EndsWith

This method asserts that the string (the first parameter) either starts with or ends with another string (the second parameter).

Exceptions

Exceptions can be tested without writing try-catch blocks around the test method. For example, while you could write this:

```
[TestMethod]
public void CatchingExceptionsTest()
{
  try
  {
    Divide(5, 0);
  }
  catch (ArgumentOutOfRangeException)
  {
    // Silently accept the exception as valid.
  }
}
```

It is much more readable to use the ExpectedException attribute on the test method:

```
[TestMethod]
[ExpectedException(typeof(ArgumentOutOfRangeException))]
public void BadParameterTest()
{
  Divide(5, 0);
}
```

Other Useful Attributes

There are some additional attributes that are useful for running a suite of tests as well as individual tests that improve the reusability and readability of the unit test code base.

Setup/Teardown

Visual Studio's unit test engine provides four additional method attributes:

- ClassInitialize
- ClassCleanup
- TestInitialize
- TestCleanup

These attributes precede and follow the execution of all tests within the fixture (class), as well as before and after each test in the fixture.

Note that the methods decorated with this attribute must be static.

ClassInitialize

If a method is decorated with this attribute, the code in the method is executed prior to running all tests in the fixture. Note that this method requires a TestContext parameter.

This method is useful to allocate resources or instantiate classes that all the tests in the fixture rely on. An important consideration with resources and objects created during the fixture initialization is that these resources and objects should be considered read-only. It is not advisable for tests to change the state of resources and objects on which other tests depend. This includes connections to services such as database and web services whose connection might be put into an invalid state as the result of an error in a test, thus invalidating all of the other tests. Furthermore, the order in which tests run is not guaranteed. Altering the state of a resource and object created in the fixture initialization may result in side effects, depending on the order in which tests are run.

ClassCleanup

A method decorated with this attribute is responsible for de-allocating resources, closing connections, etc., that were created during class initialization. This method will always execute after running the tests within the fixture, regardless of the success or failure of the tests themselves.

TestInitialize

Similar to the ClassInitialize attribute, a method decorated with this attribute will be executed *for each test* prior to running the test. One of the purposes of this attribute is to ensure that resources or objects allocated by the ClassInitialize code are initialized to a known state before running each test.

TestCleanup

Complementing the TestInitialize attribute, methods decorated with TestCleanup will be executed at the completion *of each test.*

Setup and Teardown Flow

The following code demonstrates the flow of fixture and test setup and teardown with relation to the actual tests:

```
[TestClass]
public class SetupTeardownFlow
{
  [ClassInitialize]
  public static void SetupFixture(TestContext context)
  {
    Debug.WriteLine("Fixture Setup.");
  }

  [ClassCleanup]
  public static void TeardownFixture()
  {
    Debug.WriteLine("Fixture Teardown.");
  }

  [TestInitialize]
  public void SetupTest()
  {
    Debug.WriteLine("Test Setup.");
  }

  [TestCleanup]
  public void TeardownTest()
  {
    Debug.WriteLine("Test Teardown.");
  }

  [TestMethod]
  public void TestA()
  {
    Debug.WriteLine("Test A.");
  }

  [TestMethod]
  public void TestB()
  {
    Debug.WriteLine("Test B.");
  }
}
```

Running this fixture results in the following debug output trace:

```
Fixture Setup.
Test Setup.
Test A.
Test Teardown.
Test Setup.
Test B.
Test Teardown.
Fixture Teardown.
```

As is illustrated in the previous example, the fixture is initialized—then for each test, the test setup and teardown code executes, followed by the fixture teardown at the end.

Less Frequently Used Attributes

The following section describes less commonly used attributes.

AssemblyInitialize/AssemblyCleanup

Methods decorated with this attribute must be static and are executed when the assembly is loaded. This begs the question—what if the assembly has more than one test fixture?

```
[TestClass]
public class Fixture1
{
  [AssemblyInitialize]
  public static void AssemblyInit(TestContext context)
  {
    // ... some operation
  }
}

[TestClass]
public class Fixture2
{
  [AssemblyInitialize]
  public static void AssemblyInit(TestContext context)
  {
    // ... some operation
  }
}
```

If you try this, the test engine fails to run any unit tests, reporting:

“UTA013: UnitTestExamplesVS2008.Fixture2: Cannot define more than one method with the AssemblyInitialize attribute inside an assembly.”

Therefore, only one AssemblyInitialize and one AssemblyCleanup method can exist for an assembly, regardless of the number of test fixtures in that assembly. It is therefore recommended that no actual tests are put into the class that defines these methods:

```
[TestClass]
public class AssemblyFixture
{
  [AssemblyInitialize]
  public static void AssemblySetup(TestContext context)
  {
    Debug.WriteLine("Assembly Initialize.");
  }

  [AssemblyCleanup]
  public static void AssemblyTeardown()
  {
    Debug.WriteLine("Assembly Cleanup.");
  }
}
```

resulting in the following execution sequence:

```
Assembly Initialize.
Fixture Setup.
Test Setup.
Test A.
Test Teardown.
Test Setup.
Test B.
Test Teardown.
Fixture Teardown.
Assembly Cleanup.
```

Note the additional assembly initialize and cleanup calls.

Ignore

This method can decorate specific methods or entire fixtures.

Ignore a Test Method

If this attribute decorates a test method:

```
[TestMethod, Ignore]
public void TestA()
```

```
{
  Debug.WriteLine("Test A.");
}
```

the test will not run. Unfortunately, the Visual Studio Test Result pane does not indicate that there are tests currently ignored:

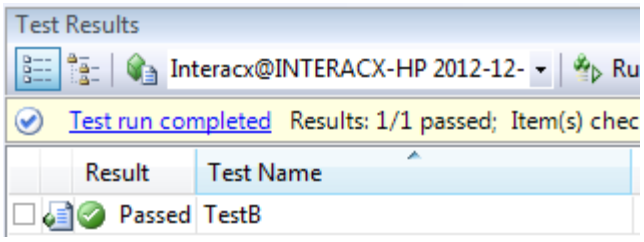


Figure 12: Ignored Tests Are Not Shown

Compare this with NUnit, which clearly shows ignored tests:

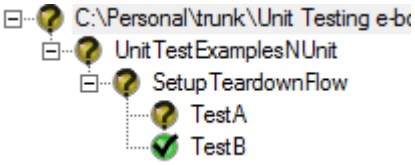


Figure 13: NUnit Shows Ignored Tests

The NUnit display marks the entire test tree as “unknown” when one or more test methods are marked as “Ignore.”

Ignore a Test Fixture

An entire fixture’s methods can be ignored by using the Ignore attribute at the class level:

```
[TestClass, Ignore]
public class SetupTeardownFlow
{
... etc ...
```

Clearing the Test Cache

If you add the Ignore attribute to a method, you may notice that Visual Studio still runs the test. It is necessary to clear the test cache for Visual Studio to pick up the change. One way to do this is to clean the solution and rebuild it.

Owner

Used for reporting purposes, this attribute describes the person responsible for the unit test method.

DeploymentItem

If the unit tests are being run in a separate deployment folder, this attribute can be used to specify files that a test class or test method requires in order to run. You can specify files or folders to copy to the deployment folder and optionally specify the target path relative to the deployment folder.

Description

Used for reporting, this attribute provides a description of the test method. Oddly, this attribute is available only on test methods and is not available on test classes.

HostType

For test methods, this attribute is used to specify the host that the unit test will run in.

Priority

This attribute is not used by the test engine, but could be used, via reflection, by your own test code. The usefulness of this attribute is questionable.

WorkItem

If you are using Team Foundation Server (TFS), you can use this attribute on a test method to specify the work item ID assigned by TFS to the specific unit test.

CssIteration/CssProjectStructure

These two attributes are used in relationship with TeamBuild and TestManagementService and allow you to specify a project iteration to which the test method corresponds.

Parameterized Testing with the DataSource Attribute

Microsoft's unit test engine supports CSV, XML, or database data sources for parameterized testing. This is not exactly true parameterized testing (see how NUnit implements parameterized testing) because the parameters are not passed to the unit test method but must be extracted from the data source and passed to the method under test. However, the ability to load test data into a DataTable from a variety of sources is helpful for driving test automation.

CSV Data Source

A comma-separated-value text file can be used for a data source:

```
Numerator, Denominator, ExpectedResult
10, 5, 2
20,5, 4
33, 3, 11
```

and used in a test method:

```
[TestClass]
public class DataSourceExamples
{
  public TestContext TestContext { get; set; }

  [TestMethod]
  [DataSource("Microsoft.VisualStudio.TestTools.DataSource.CSV", "C:\\temp\\csvData.txt",
"csvData#txt",
     DataAccessMethod.Sequential)]
  public void CsvDataSourceTest()
  {
     int n = Convert.ToInt32(TestContext.DataRow["Numerator"]);
     int d = Convert.ToInt32(TestContext.DataRow["Denominator"]);
     int r = Convert.ToInt32(TestContext.DataRow["ExpectedResult"]);

     Debug.WriteLine("n = " + n + " , d = " + d + " , r = " + r);
  }
}
```

It results in the following output:

```
n = 10 , d = 5 , r = 2
n = 20 , d = 5 , r = 4
n = 33 , d = 3 , r = 11
```

Note that the test result window does not show the parameter runs (contrast this to NUnit):

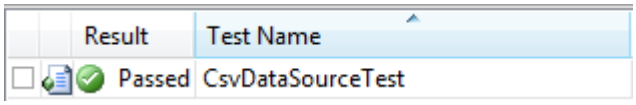


Figure 14: Parameterized Test Results

However, there are obvious advantages to not displaying each test combination, especially for large datasets.

XML Data Source

Given an XML file such as:

```
<Data>
<Row Numerator = "10" Denominator = "5" ExpectedResult = "2"/>
<Row Numerator = "20" Denominator = "5" ExpectedResult = "4"/>
<Row Numerator = "33" Denominator = "3" ExpectedResult = "11"/>
</Data>
```

an example of using an XML data source for a unit test is:

```
[TestMethod]
[DataSource("Microsoft.VisualStudio.TestTools.DataSource.XML", "C:\\temp\\xmlData.xml",
     "Row", DataAccessMethod.Sequential)]
  public void XmlDataSourceTest()
  {
    int n = Convert.ToInt32(TestContext.DataRow["Numerator"]);
    int d = Convert.ToInt32(TestContext.DataRow["Denominator"]);
    int r = Convert.ToInt32(TestContext.DataRow["ExpectedResult"]);

    Debug.WriteLine("n = " + n + " , d = " + d + " , r = " + r);
}
```

Note that other than the data source attribute parameters, the test code is the same.

Database Data Source

A database table can also be used as the data source. Given a table such as:

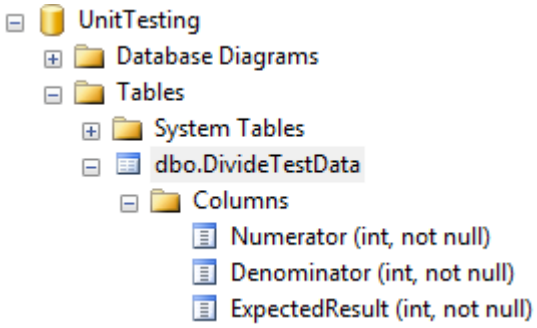
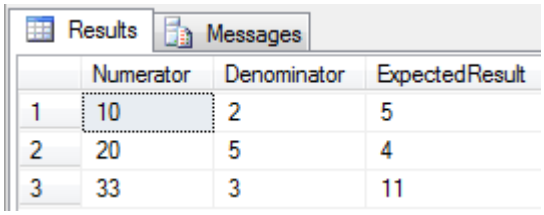


Figure 15: Database Table as a Data Source

and data:



	Numerator	Denominator	ExpectedResult
1	10	2	5
2	20	5	4
3	33	3	11

Figure 16: Database Test Data

An example test method using this data looks like:

```
[TestMethod]
[DataSource("System.Data.SqlClient", "Data Source=INTERACX-HP;Initial
Catalog=UnitTesting;Integrated Security=True", "DivideTestData",
DataAccessMethod.Sequential)]
  public void XmlDataSourceTest()
  {
    int n = Convert.ToInt32(TestContext.DataRow["Numerator"]);
    int d = Convert.ToInt32(TestContext.DataRow["Denominator"]);
    int r = Convert.ToInt32(TestContext.DataRow["ExpectedResult"]);

    Debug.WriteLine("n = " + n + " , d = " + d + " , r = " + r);
}
```

Again, observe that the test method code itself is the same—the only thing we've done here is change the DataSource definition.

TestProperty Attribute

The MSDN documentation for this attribute illustrates declaring a TestProperty name-value pair and then, using reflection, acquiring the name and value. This seems to be an obtuse way of creating parameterized tests. Furthermore, the code, described on Craig Andera's blog, to use the TestProperty attribute to parameterize the test initialization process does not affect the TestContext.Properties collection on Visual Studio 2008 or Visual Studio 2012.

Chapter 10 Unit Testing with NUnit

This chapter discusses testing with NUnit and the attributes and assertions that it supports. NUnit 2.6.2 was used at the time of this writing. The description of the attributes and assertions here should be considered supplemental to the online NUnit documentation; however, in most cases the author has attempted to provide valuable supplemental information.

NUnit Attributes

The following table maps the attributes used for writing tests with NUnit with Visual Studio:

NUnit Attribute	Visual Studio Attribute	Description
TestFixture	TestClass	Defines a test fixture
Test	TestMethod	Defines a test method within the test fixture class
TestFixtureSetUp	ClassInitialize	Specifies the code that runs before all test methods in the test fixture run.
TestFixtureTearDown	ClassCleanup	Specifies the code that runs after all tests in the fixture are complete.
SetUp	TestInitialize	Specifies the code to run prior to running each test.
TearDown	TestCleanup	Specifies the code to run at the completion of each test.
SetUpFixture (see the following)	AssemblyInitialize	Specifies the code to run when the assembly containing all the test fixtures is loaded.
SetUpFixture (see the following)	AssemblyCleanup	Specifies the code to run when the assembly containing all the test fixtures is unloaded.
Ignore	Ignore	Ignores the specific test.
Description (also applies to test fixtures).	Description	A description of the test method. In NUnit, this attribute can also decorate a test fixture.

The following Visual Studio attributes do not correspond to any NUnit attributes:

- Owner
- DeploymentItem
- HostType
- Priority
- WorkItem
- CssIteration
- CssProjectStructure
- TestProperty

The SetUpFixture Attribute

The SetUpFixture attribute, which applies to classes, is different from Visual Studio's AssemblyInitialize and AssemblyCleanup because it applies to fixtures in a given namespace. If all your test fixture classes are in the same namespace, then this attribute does behave similarly to AssemblyInitialize and AssemblyCleanup. Given the code:

```
namespace UnitTestExamplesNUnit
{
  [SetUpFixture]
  public class SetupFixtureForNamespace
  {
    [SetUp]
    public void RunBeforeAllFixtures()
    {
      Console.WriteLine("Before all test fixtures.");
    }

    [TearDown]
    public void RunAfterAllFixtures()
    {
      Console.WriteLine("After all test fixtures.");
    }
  }

  [TestFixture]
  public class SetupTeardownFlow
  {
    [TestFixtureSetUp]
    public void SetupFixture()
    {
      Console.WriteLine("Fixture Setup.");
    }

    [TestFixtureTearDown]
    public void TeardownFixture()
```

```
    {
      Console.WriteLine("Fixture Teardown.");
    }

    [SetUp]
    public void SetupTest()
    {
      Console.WriteLine("Test Setup.");
    }

    [TearDown]
    public void TeardownTest()
    {
      Console.WriteLine("Test Teardown.");
    }

    [Test]
    public void TestA()
    {
      Console.WriteLine("Test A.");
    }

    [Test]
    public void TestB()
    {
      Console.WriteLine("Test B.");
    }
  }
}
```

The resulting output is:

```
Before all test fixtures.
Fixture Setup.
***** UnitTestExamplesNUnit.SetupTeardownFlow.TestA
Test Setup.
Test A.
Test Teardown.
***** UnitTestExamplesNUnit.SetupTeardownFlow.TestB
Test Setup.
Test B.
Test Teardown.
Fixture Teardown.
After all test fixtures.
```

However, adding another namespace:

```
namespace AnotherNamespace
{
```

```
  [SetUpFixture]
  public class AnotherSetupFixtureForNamespace
  {
    [SetUp]
    public void RunBeforeAllFixtures()
    {
      Console.WriteLine("Another before all test fixtures.");
    }

    [TearDown]
    public void RunAfterAllFixtures()
    {
      Console.WriteLine("Another after all test fixtures.");
    }
  }
}
```

Results in the following output:

```
Another before all test fixtures.
Another after all test fixtures.
Before all test fixtures.
Fixture Setup.
***** UnitTestExamplesNUnit.SetupTeardownFlow.TestA
Test Setup.
Test A.
Test Teardown.
***** UnitTestExamplesNUnit.SetupTeardownFlow.TestB
Test Setup.
Test B.
Test Teardown.
Fixture Teardown.
After all test fixtures.
```

The ability to execute code specific to the namespace context has the advantage of being able to organize a suite of tests in different fixtures, but all in the same namespace that require a specific setup and teardown process.

Also see “Assembly Actions” under “User Defined Action Attributes” in the following section.

Additional NUnit Attributes

NUnit has an extensive set of attributes that provide considerable additional functionality to unit testing.

Test Grouping and Control

- Category
- Suite
- Explicit
- Timeout

Category

The Category attribute allows you to group tests and run tests in selected categories. This attribute can be applied to tests in a fixture as well as individual tests within a fixture. Specific categories can be selected from the console runner using the /include and /exclude arguments or from the GUI runner:

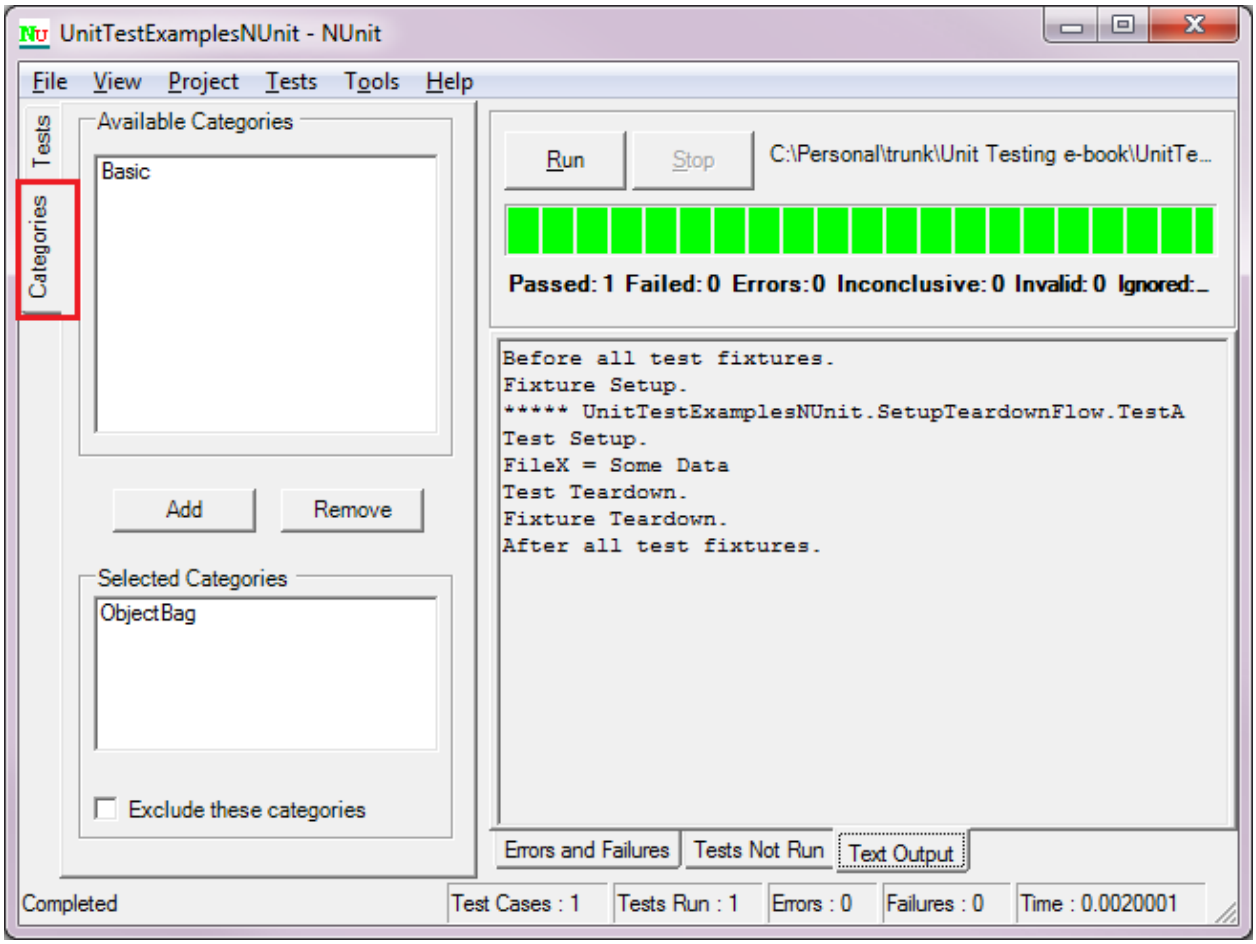


Figure 17: NUnit Categories

The tab for selecting the categories to include or exclude is on the left edge of the GUI runner.

Suite

The Suite attribute provides a means of programmatically specifying the test fixtures that the console runner should test. This option is available only in the console runner. The concept of

running only specific tests or test suites (fixtures) is supported better by the Category attribute described earlier.

Explicit

This attribute designates that a test is run only if the GUI or console runner is explicitly told to run the test by selecting the test. The test will also run if the test is part of a category of tests to run.

Uses for this attribute are to only run long-running tests when explicitly required, when a service is up and running, etc.

Timeout

This attribute can be used to ensure that a test runs only within the specified time in milliseconds. If the test exceeds the specified timeframe, the runner cancels the execution and reports the test as failed.

Compare this attribute with the MaxTime attribute described below.

Also note that, when running unit tests in the debugger, the Timeout attribute is ignored—otherwise, the test that you are debugging could terminate as you are manually stepping through code, inspecting values, etc.

Culture Attributes

- Culture
- SetCulture
- SetUICulture

Culture

The Culture attribute can be used for a fixture to define the cultures for which tests should be run. Tests that are culture specific should then be decorated with this attribute as well, describing the cultures that they test. The test or fixture is skipped if *the current culture* does not match the culture for which the test is written. For example:

```
[TestFixture, SetupData]
[Culture("fr, en")]
public class SetupTeardownFlow
{
  [Test, Category("ObjectBag")]
  [Culture("fr")]
  public void TestA()
  {
    Console.WriteLine("FileX = "+Globals.objectBag["FileX"].ToString());
  }

  [Test, Category("Basic")]
  public void TestB()
```

```
  {
   Console.WriteLine("Test B.");
  }
}
```

In the preceding code, the fixture runs if the current culture is either “fr” or “en”; however, TestA specifies that the test should only be run if the culture is “fr.” Therefore, this fixture will result in:

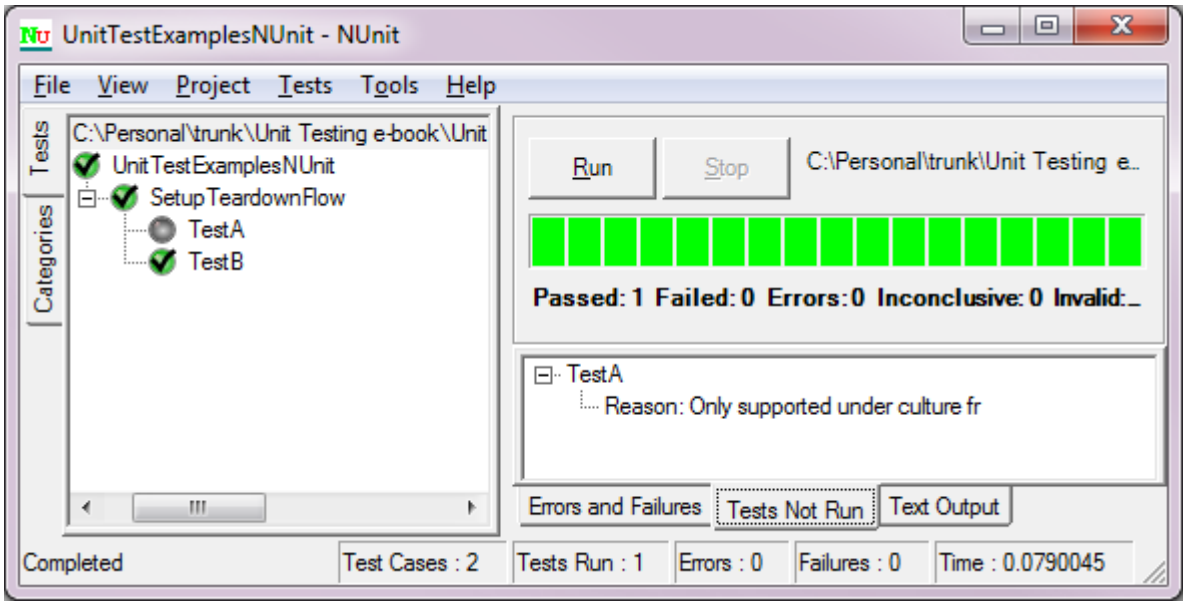


Figure 18: Culture Unit Testing

as “TestA” does not run because the current culture is not “fr.”

Tests such as “TestB” that omit any culture specification always run unless the entire fixture is excluded because the current culture does not match.

SetCulture

This attribute, applied to either the entire fixture or specific tests, sets the current Culture for the duration of the test (or tests in the fixture) and then restores the original value. Unlike the Culture attribute before this, only one culture can be specified, though according to the NUnit documentation, running tests for multiple cultures is planned as a future enhancement. Note that setting the culture in the fixture does not change whether the test using the Culture attribute runs. For example:

```
[TestFixture]
[SetCulture("fr-FR")]
public class CultureTests
{
  [Test]
  [Culture("fr-FR")]
  public void TestA()
```

```
  {
    double value = 1.2;
    Console.WriteLine(value.ToString("0.00", CultureInfo.CurrentCulture));
  }

  [Test]
  public void TestB()
  {
    double value = 1.2;
    Console.WriteLine("French: " + value.ToString("0.00", CultureInfo.CurrentCulture));
  }

  [Test, SetCulture("en-US")]
  public void TestC()
  {
    double value = 1.2;
    Console.WriteLine("English-US: " + value.ToString("0.00",
CultureInfo.CurrentCulture));
  }
}
```

results in TestA not running. Also observe the default behavior of TestB when the culture is set in the fixture and how it is overridden in TestC:

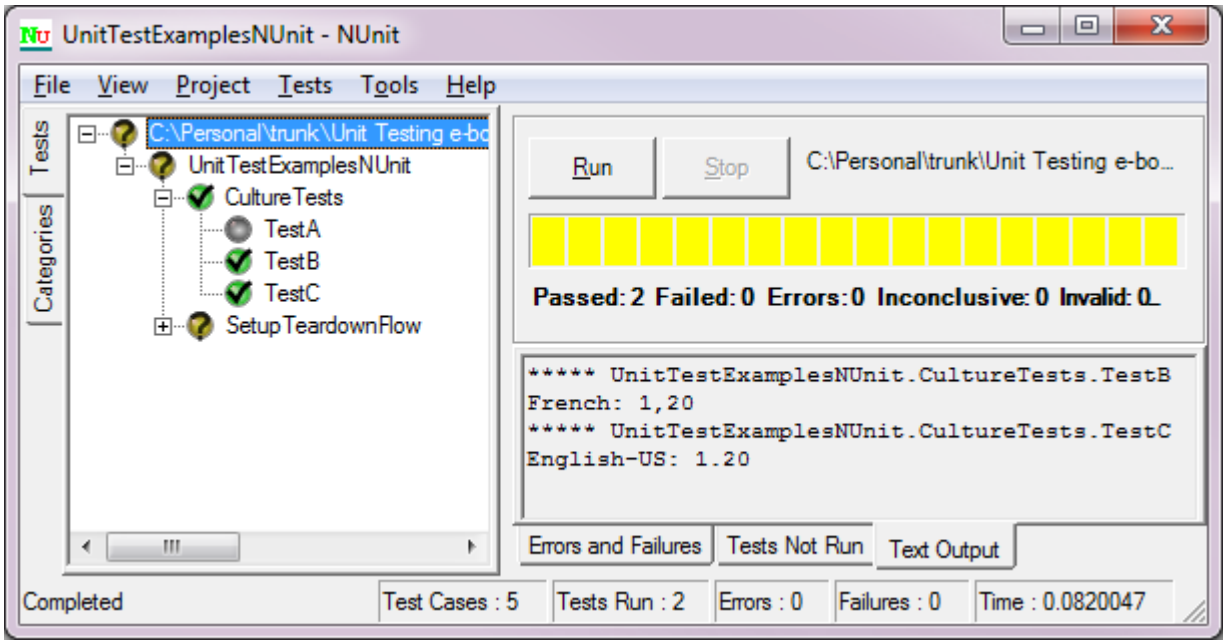


Figure 19: Overriding in TestC

SetUICulture

This attribute is supposed to set the culture for the user interface; however, it apparently does nothing. For example (don't write a unit test like this, this is for illustration purposes only):

```
[TestFixture]
[SetUICulture("fr-FR")]
public class CultureTests
{
  public event EventHandler ValueChanged;

  protected CultureTestForm ctForm;
  protected double val;

  public double Value
  {
    get { return val; }
    set
    {
      val = value;

      if (ValueChanged != null)
      {
        ValueChanged(this, EventArgs.Empty);
      }
    }
  }

  [TestFixtureSetUp]
  public void SetupFixture()
  {
    ctForm = new CultureTestForm();
    ctForm.tbFrench.DataBindings.Add(new Binding("Text", this, "Value"));
  }

  [TestFixtureTearDown]
  public void TeardownFixture()
  {
    ctForm.ShowDialog();
  }

  [Test]
  public void TestA()
  {
    Value = 1.2;
  }
}
```

results in the value “1.2” being displayed in my current culture, being en-US:

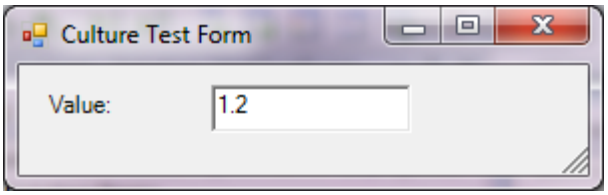


Figure 20: SetUICulture Attribute Does Nothing

If I change the attribute to “SetCulture,” then the user interface displays the value in the correct culture format (you may have to squint to see the difference between “1 point 2” and “1 comma 2”:

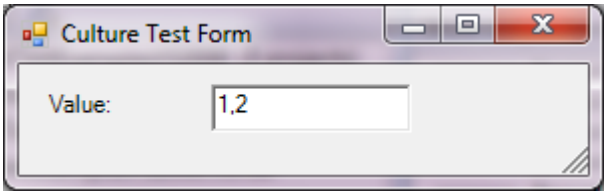


Figure 21: SetCulture Changes the UI Representation

Parameterized Tests

Parameterized tests are ones in which the unit test describes the parameters and their values that will be used to run a test iteratively until all parameter combinations are run.

For example, this test:

```
[TestFixture]
public class ParameterizedTests
{
  [Test]
  public void TestWithParams([Values(1, 2, 3)] int x)
  {
    Console.WriteLine("X = " + x);
  }
}
```

results in the following output (and notice the test graph also displays the test parameters):

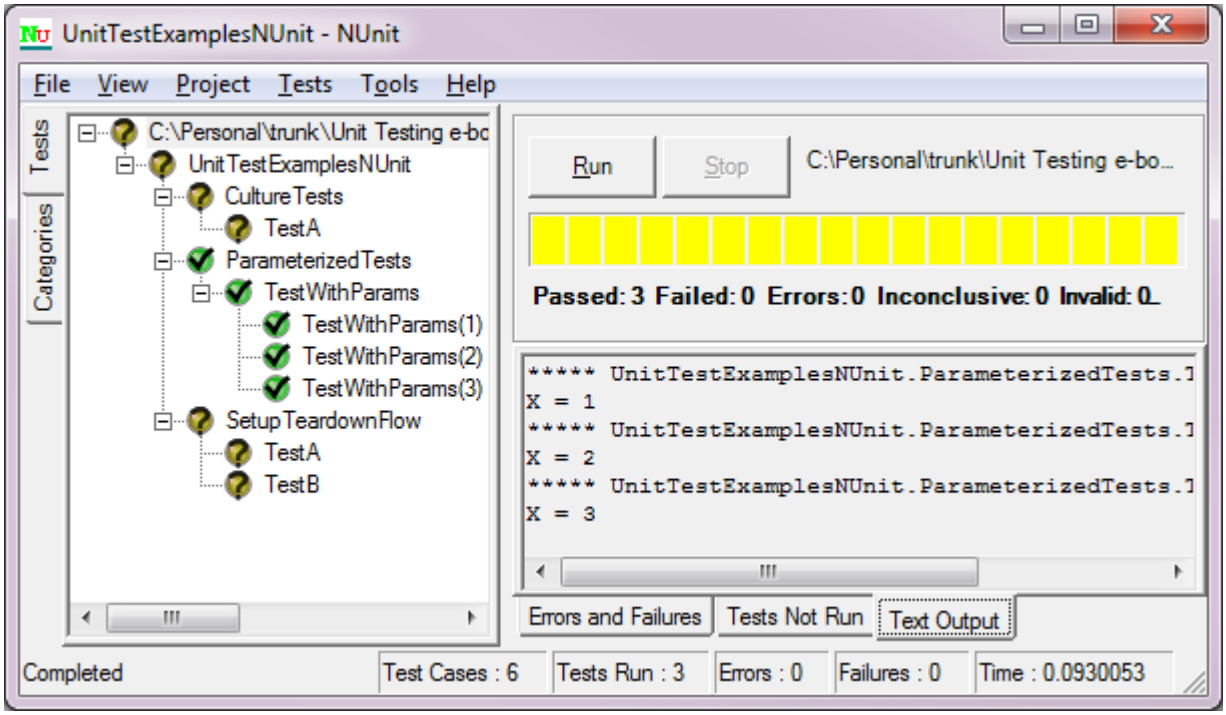


Figure 22: Parameterized Tests

The following attributes are used for parameterized testing:

- Values
- ValuesSource
- Combinatorial
- Pairwise (not implemented)
- Sequential
- Random
- Range
- TestCase
- TestCaseSource

Values

As illustrated in the previous example, the Values attribute is used to specify the values passed into the test method for each parameter. Because these are values assigned in an attribute, they must be of value type: constant expressions, typeof expressions, or an array of attribute types.

Also note that NUnit will by default test every combination of values for each parameter (see the attributes Combinatorial, Pairwise, and Sequential).

ValuesSource

With the ValueSource attribute, you can specify an IEnumerable source for the values of a specific parameter. The source can be a field, property, or method of a separate class or the current class. For example, these are all valid ways of expressing value sources:

```
public class NumeratorList
{
  protected List<int> values;

  public IEnumerable Values {get {return values;}}

  public NumeratorList()
  {
    values = new List<int>() {10, 20, 30};
  }
}

public class DenominatorList
{
  public List<int> values = new List<int>() { 1, 2, 3 };
}

[TestFixture]
public class ValueSourceExamples
{
  List<int> results = new List<int>() { 10, 10, 10};

  [Test, Sequential]
  public void DivideTest(
    [ValueSource(typeof(NumeratorList), "Values")] int n,
    [ValueSource(typeof(DenominatorList), "values")] int d,
    [ValueSource("results")] int expectedResult)
  {
    int r = MyMath.Divide(n, d);
    Assert.AreEqual(expectedResult, r);
  }
}
```

Note the use of the Sequential attribute on the test, which ensures that the source values are used sequentially rather than combinatorially. Therefore the result is:

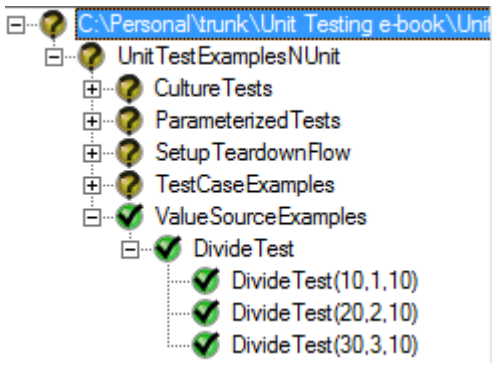


Figure 23: ValueSource Tests

Combinatorial

This attribute is optional, as NUnit will automatically apply test parameters in all possible combinations. For example:

```
[TestFixture]
public class ParameterizedTests
{
  [Test, Combinatorial]
  public void TestWithParams(
     [Values(1, 2, 3)] int x,
     [Values("A", "B")] string s)
  {
    Console.WriteLine("X = " + x + " , " + s);
  }
}
```

results in all combinations of “x” and “s,” whether or not “Combinatorial” is specified:

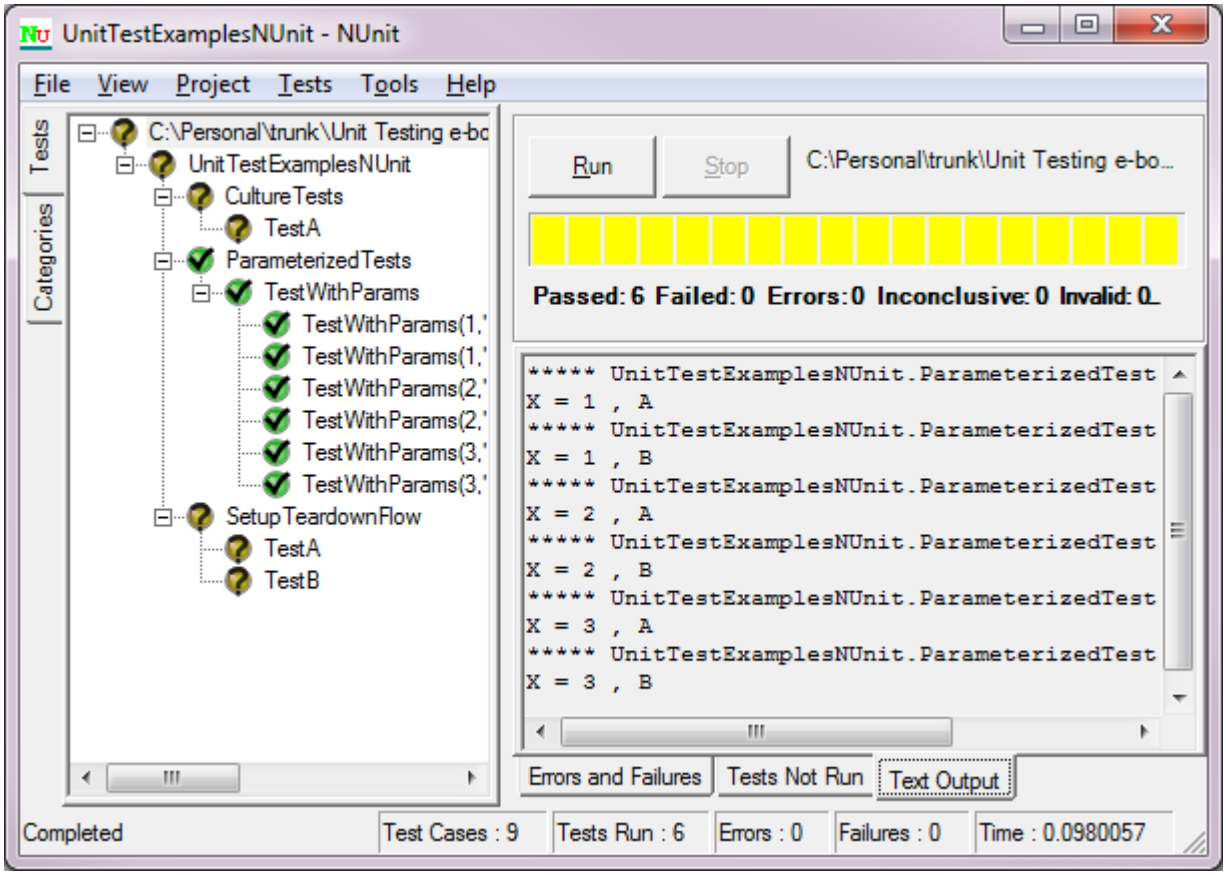


Figure 24: Combinatorial Test

Pairwise

This attribute exists but is not implemented—it is intended to reduce the number of combinations when combinatorial parameters are specified; however, this attribute is ignored.

Sequential

This attribute specifies that tests will be run by sequentially stepping through parameter values. If the number of values is not the same for each parameter, the default value will be used. Note that the default value for a string is null. For example, this code, using the Sequential attribute:

```
[TestFixture]
public class ParameterizedTests
{
  [Test, Sequential]
  public void TestWithParams(
  [Values(1, 2, 3)] int x,
  [Values("A", "B")] string s,
  [Values(10.1, 10.2, 10.3, 10.4)] double v)
  {
    Console.WriteLine("X = " + x + " , s = " + ((s == null) ? "[null]" : s) + " , v = " +
```

```
v);
  }
}
```

results in this output:

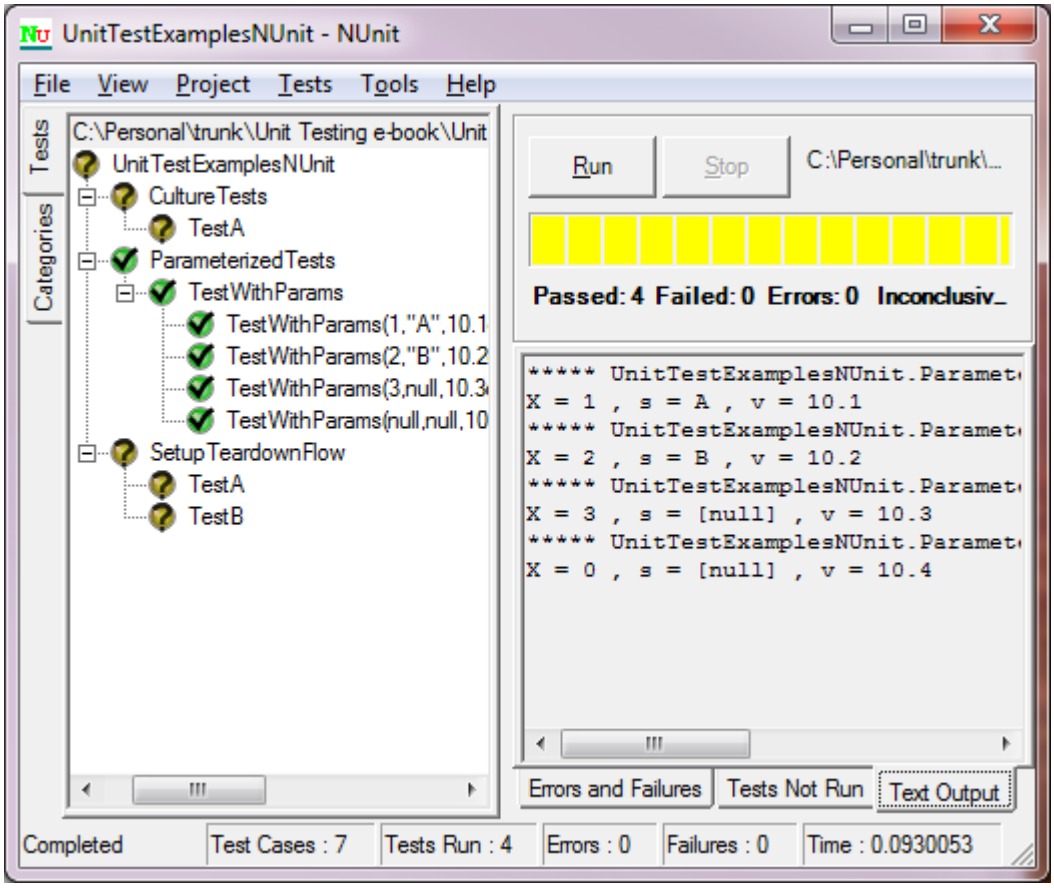


Figure 25: The Sequential Attribute

Note how the string (having only two values) defaults to null for the last two parameterized tests, and the int parameter, having three cases, defaults to 0 for the last case.

Random

The Random attribute can be applied with combinatorial parameterized tests to provide random values rather than specific values for parameters. Random value parameters are always double types. For example:

```
[TestFixture, Platform("Windows7")]
public class ParameterizedTests
{
  [Test]
  public void RandomTest([Random(5)] double x)
  {
```

```
    Console.WriteLine("X = " + x);
  }
}
```

results in five tests with random values:

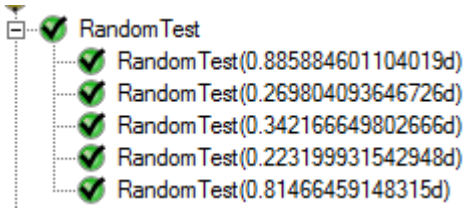


Figure 26: Random Test

A minimum and maximum can also be specified for the random value range.

It is easy to create hundreds, if not thousands, of test cases with combinatorial parameterized tests. Consider using the Sequential attribute to prevent large numbers of combinations from being tested. The Sequential attribute applied to the test can be used in conjunction with the Random attribute on the test's parameters.

Range

The Range attribute specifies a range of values to test. Except for an integer range, other ranges (long, float, double) all require a step to be specified. The step for integer values is optional.

Ranges are combined with other parameterized values, so again, consider using the Sequential attribute to limit the number of test combinations that are generated.

TestCase

Rather than a combinatorial or sequential parameterized test, you can explicitly define the parameter values passed to each test case to run for a particular test. This has the advantage of reducing the number of combinatorial tests as well as targeting specific test scenarios. Furthermore, the TestCase attribute includes the additional characteristic of being able to test the return value (as long as it is a "value" type) to a value specified in the metadata of the attribute. Lastly, additional properties, such as mirroring attributes, are exposed in the TestCase attribute.

A basic test case looks like this (note the lack of the TestFixture attribute):

```
public class TestCaseExamples
{
  public int Divide(int n, int d)
  {
    return n / d;
  }
```

```
  [TestCase(10, 2)]
  [TestCase(20, 10)]
  public void DivideTest(int n, int d)
  {
    int r = Divide(n, d);
    Assert.AreEqual(r, n / d);
  }
}
```

However, the real power of the TestCase attribute is in the ability to apply the test cases to the method directly and inspect the result:

```
public class TestCaseExamples
{
  [TestCase(10, 2, Result=5)]
  [TestCase(20, 10, Result=2)]
  public int Divide(int n, int d)
  {
    return n / d;
  }
}
```

Remember that the values specified in attributes must be value types: a constant expression, a typeof expression, or an array creation expression of an attribute parameter type.

Also beware that the TestCase attribute might encourage test cases to be placed in the same assembly as the production code rather than in a separate test assembly—attributes are not removed in the release build of an assembly. It is recommended that when using the TestCase attribute, you do not apply the attribute directly to the code for which you want the test cases but rather the test fixture itself. So, ideally, the correct code should look something like this:

```
// In the application's assembly:
public static class MyMath
{
  public static int Divide(int n, int d)
  {
    return n / d;
  }
}

// In the unit test assembly:
[TestFixture]
public class TestCaseExamples
{
  [TestCase(10, 2, Result=5)]
  [TestCase(20, 10, Result=2)]
  public int DivideTest(int n, int d)
  {
    return MyMath.Divide(n, d);
```

```
  }
}
```

in which the class MyMath is in the application's assembly and the test fixture is in the unit test assembly. The DivideTest method then becomes a thin wrapper to the application's method.

Other properties that can be assigned to the TestFixture are illustrated in this example:

```
[TestFixture]
public class TestCaseExamples
{
  [TestCase(10, 2,
    Result=5,
    Description = "Normal usage.",
    TestName = "Normal divide test.")]
  [TestCase(5, 0,
    ExpectedException=typeof(ArgumentOutOfRangeException),
    Description = "Divide by 0 test.",
    TestName = "Divide by 0 test.")]
  [TestCase(10, 5,
    Result=2,
    Ignore = true,
    Reason = "Already tested normal usage.")]
  [TestCase(15, 5,
    Result=3,
    IgnoreReason = "Yet another already tested normal usage.")]
  public int DivideTest(int n, int d)
  {
    return MyMath.Divide(n, d);
  }
}
```

There are also alternatives specifying the expected exception:

- ExpectedExceptionName
- ExpectedExceptionMessage
- MatchType

which allows you to specify the full name (matching the value of Type.FullName property) or the exception message, as well as how the exception message should be matched—Contains, Exact, Regex, or StartsWith.

TestCaseSource

The TestCaseSource attribute behaves similarly to the ValueSource attribute described earlier; however, the preferred implementation is to specify a class type that implements IEnumerable. This preferred implementation constrains you to something like this:

```
public class ValuesList : IEnumerable
{
  public List<int[]> values = new List<int[]>()
    {
      new int[] {10, 1, 10},
      new int[] {20, 2, 10},
      new int[] {30, 3, 10} };

  public IEnumerator GetEnumerator()
  {
    return new ValuesListEnum(values);
  }
}

public class ValuesListEnum : IEnumerator
{
  protected List<int[]> values;
  protected int pos = -1;

  public ValuesListEnum(List<int[]> values)
  {
    this.values = values;
  }

  public bool MoveNext()
  {
    ++pos;

    return pos < values.Count;
  }

  public object Current
  {
    get { return values[pos]; }
  }

  public void Reset()
  {
    pos = -1;
  }
}

[TestFixture]
public class TestCaseSourceExamples
{
  [Test, Sequential]
  [TestCaseSource(typeof(ValuesList))]
  public void DivideTest(int n, int d, int expectedResult)
```

```
  {
    int r = MyMath.Divide(n, d);
    Assert.AreEqual(expectedResult, r);
  }
}
```

Note how the test case values are an int[] (integer array), in which each element of the array maps to a parameter in the test function. Also note again that the Sequential attribute is specified for the test, ensuring that the test data is used sequentially rather than combinatorially.

Other NUnit Attributes

There are several other NUnit attributes, described next.

Platform

The Platform attribute can be used to specify the platform for which the test should run. If the platform does not match the current platform, the test is not run and is not even included in the ignored tests or the total number of tests. Review the documentation at http://www.nunit.org/index.php?p=platform&r=2.6.2 to see the list of supported platform specifiers.

Property

You can use this attribute to set a name or value pair on a test or fixture, which is then displayed in the XML output file. Via reflection, you can access these attributes. Custom property attributes can also be created by deriving a custom attribute class from the PropertyAttribute class.

MaxTime

The MaxTime attribute (if compared to the Timeout attribute described earlier) will cause the test to fail if it exceeds the specified time in milliseconds. However, unlike the Timeout attribute, the test will be allowed to finish.

Repeat

The Repeat attribute can only be applied to tests and is ignored for parameterized tests. It instructs the test engine to repeat the test the specified number of times. If any repetition of the test fails, the remaining tests are not run and the engine reports a failure.

RequiredAddin

With this attribute, you can specify additional assemblies (in the AssemblyInfo file) that the unit tests require.

RequiresMTA

This attribute, applied to a test or a fixture, indicates that the test should run in a multithreaded apartment. NUnit will create a new thread for the test if:

- The assembly is not already running on an MTA thread.
- The containing fixture is not already running on an MTA thread.

If a thread is created for the entire fixture, all tests in that fixture run on the new thread.

RequiresSTA

This attribute, applied to a test or a fixture, indicates that the test should run in a single-threaded apartment. NUnit will create a new thread for the test if:

- The assembly is not already running on an STA thread.
- The containing fixture is not already running on an STA thread.

If a thread is created for the entire fixture, all tests in that fixture run on the new thread.

RequiresThread

This attribute, applied to a test or a fixture, indicates that the test should run in a separate thread. NUnit will create a new thread if this attribute is specified in the AssemblyInfo file:

```
[assembly:RequiresThread]
```

If the fixture specifies this attribute, all tests in the fixture run on a newly created thread.

If the test specifies this attribute, the test will run on a newly created thread.

Optionally, the apartment state can be specified as well, which causes either an MTA or STA thread to be created.

For example:

```
[TestFixture, RequiresThread]
public class ThreadExamples
{
  [Test]
  public void TestA()
  {
    Console.WriteLine(Thread.CurrentThread.ManagedThreadId);
  }

  [Test, RequiresThread]
  public void TestB()
  {
```

```
    Console.WriteLine(Thread.CurrentThread.ManagedThreadId);
  }

  [Test]
  public void TestC()
  {
    Console.WriteLine(Thread.CurrentThread.ManagedThreadId);
  }
}
```

Notice the thread IDs when the test runs:

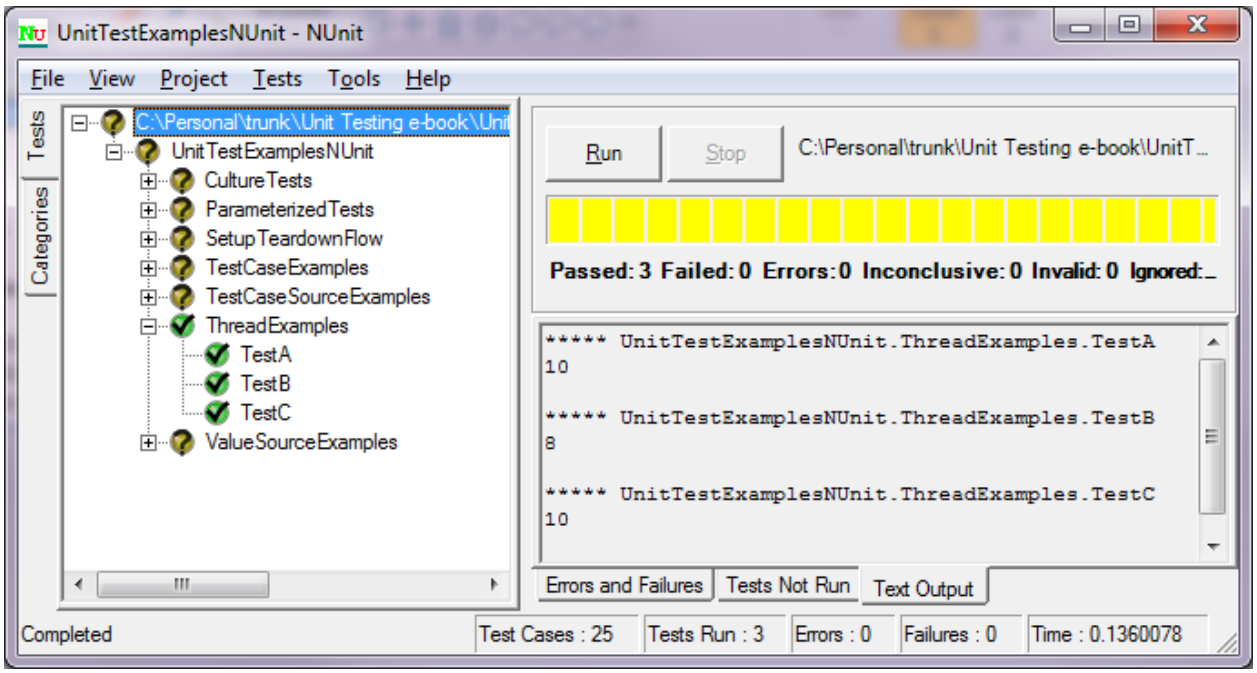


Figure 27: Thread Examples

Test A and Test C run on a thread created by the fixture, whereas Test B, having specified the RequiresThread attribute, runs on a separate thread from that of the other tests in the fixture.

Theory and Datapoint(s)

A theory is a parameterized test that verifies assumptions about the values being passed in. If the assumption fails, the test does not fail. It works with the Datapoint(s) attribute.

These two classes are equivalent. The first illustrates the use of the Datapoint attribute, the second, the Datapoints attribute:

```
public class TheoryExample1
{
  [Datapoint]
  public int a = 10;
```

```
  [Datapoint]
  public int b = 5;
  [Datapoint]
  public int c = 2;
  [Datapoint]
  public int d = 0;

  [Theory]
  public void DivideTest(int n, int d, int expectedResult)
  {
    Assume.That(d > 0);
    Assume.That(n / d == expectedResult);

    int r = MyMath.Divide(n, d);
    Assert.AreEqual(expectedResult, r);
  }
}

public class TheoryExample2
{
  [Datapoints]
  public int[] values = new int[] { 10, 5, 2, 0 };

  [Theory]
  public void DivideTest(int n, int d, int expectedResult)
  {
    Assume.That(d > 0);
    Assume.That(n/d == expectedResult);

    int r = MyMath.Divide(n, d);
    Assert.AreEqual(expectedResult, r);
  }
}
```

Note in the first class, TheoryExample1, discrete data point values are used, whereas in the second class, TheoryExample2, an array is used.

In both cases, NUnit will create all possible combinations of values from the data points. Note how the test method makes certain assumptions:

- The denominator cannot be 0.
- The expected result must equal n/d.

The reason for the second assumption is that the test will fail for a majority of the combinations—for example, 5/10 does not equal 2. As can be seen from the test run, the assumptions do not cause the test to fail:

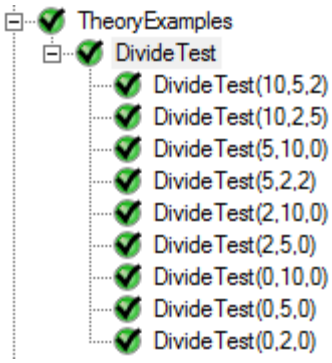


Figure 28: Theories

The combinations of parameter values that do not pass the assumptions are omitted from the test. The typical pattern for a theory test is:

- State the assumptions about the parameters.
- Perform the test.
- Make assertions about the results.

Care should be taken with theory tests because of the exploding number of combinations of arguments.

Also note that unless explicitly specified, NUnit will automatically populate the combinations of parameter values for bool and enum types.

User Defined Action Attributes

Action attributes are user-defined attributes that allow for the composition of unrelated test setup code for each test. Often, putting all the setup code in the fixture or namespace setup violates the single responsibility principle—some tests require certain setups while other tests require other setups. Action attributes allow you to specify exactly the setup that a test requires.

One of the advantages of user-defined action attributes is that it moves the setup and teardown code out of the fixture itself and into a more reusable and maintainable code base.

Defining an Action

A user-defined action is either derived from TestActionAttribute:

```
public class UserDefinedAttribute : TestActionAttribute
{
  public override void BeforeTest(TestDetails testDetails)
  {
    Console.WriteLine("User defined action before test.");
  }
```

```
  public override void AfterTest(TestDetails testDetails)
  {
    Console.WriteLine("User defined action after test.");
  }
}
```

or can be derived from System.Attribute and implement the ITestAction interface:

```
public class AnotherUserDefinedAttribute : Attribute, ITestAction
{
  public ActionTargets Targets { get { return ActionTargets.Default; } }

  public void BeforeTest(TestDetails testDetails)
  {
    Console.WriteLine("Another user defined action before test.");
  }

  public void AfterTest(TestDetails testDetails)
  {
    Console.WriteLine("Another user defined action after test.");
  }
}
```

In the latter case, the property “Targets” must also be defined.

The Action Targets

The action target (either a fixture or suite) or a test method can be defined through the ActionTargets enumeration (defined in NUnit.Framework):

```
// Summary:
//     The different targets a test action attribute can be applied to.
[Flags]
public enum ActionTargets
{
  // Summary:
  //     Default target, which is determined by where the action attribute is attached.
  Default = 0,
  //
  // Summary:
  //     Target an individual test case.
  Test = 1,
  //
  // Summary:
  //     Target a suite of test cases.
  Suite = 2,
```

```
  }
}
```

ActionTargets.Default

Specifying a value of ActionTargets.Default allows the user-defined attribute to be applied to both test fixtures (classes) and tests (methods). However, when applied to a fixture (class), the attribute behaves as if ActionTargets.Suite had been specified—the user-defined action runs once for the entire suite of tests.

ActionTargets.Test

If ActionTargets.Test is returned by the user-defined action, this changes the behavior of when the user-defined action is called. If applied to the fixture (class), then the user-defined action is executed *for each test.* If the user attribute is applied to a specific test, then the user-defined action is executed for that specific test.

ActionTargets.Suite

If an attribute action target is a suite but is used in the context of a test method, no error is reported—the action simply does not run.

Test and Suite Action Targets

ActionTargets are flags, so they can be combined:

```
public ActionTargets Targets
{
  get
  {
    return ActionTargets.Suite | ActionTargets.Test;
  }
}
```

to specify that the user-defined action should execute for both the fixture (the suite of tests) and the individual tests.

The TestDetails Class

This class is instantiated with information about the fixture or test that is about to run (from NUnit.Framework.TestDetails):

```
// Summary:
//     The fixture that the test is a member of, if available.
public object Fixture { get; }
//
// Summary:
//     The full name of the test.
```

```
public string FullName { get; }
//
// Summary:
//     Indicates if the test represents a suite of tests.
public bool IsSuite { get; }
//
// Summary:
//     The method that implements the test, if available.
public MethodInfo Method { get; }
//
// Summary:
//     A string representing the type of test, e.g. "Test Case".
public string Type { get; }
```

These properties can be inspected and specific actions can be taken, such as providing additional console output diagnostic messages.

Assembly Actions

Adhering more to Visual Studio's concept of AssemblyInitialize and AssemblyCleanup, you can use an assembly-based user action. For example, with the previous code, if the following line is added to the AssemblyInfo.cs file:

```
[assembly: UnitTestExamplesNUnit.UserDefined]
```

then the user-defined action will run when the assembly is loaded:

```
User-defined action before test.
Before all test fixtures.
Fixture Setup.
***** UnitTestExamplesNUnit.SetupTeardownFlow.TestA
Test Setup.
Test A.
Test Teardown.
***** UnitTestExamplesNUnit.SetupTeardownFlow.TestB
Test Setup.
Test B.
Test Teardown.
Fixture Teardown.
After all test fixtures.
User defined action after test.
```

If the target action for the user-defined action is Suite, the user-defined action executes before and after all fixture tests, behaving as “before or after all suites.” However, note that if the TargetActions.Test is specified, then the action runs before or after *all tests in all suites.* Therefore, care should be taken with regard to the return value of the Target property and the desired behavior of the user-defined action.

Passing Information to/from Tests from User-Defined Actions

Two questions might be:

- How do I pass information from the fixture to the user-defined action?
- How do I use an object that is instantiated by a user-defined action in the test itself?

Passing Information to a User-Defined Action

This is handled straightforwardly as parameters to the user action. Let’s say you want to pass in a connection string to test the behavior for a particular user role. For example, using this stub:

```
public class InitializeDatabaseConnection : TestActionAttribute
{
  protected string connection;

  public InitializeDatabaseConnection(string myConnection)
  {
    connection = myConnection;
  }

  public override void BeforeTest(TestDetails testDetails)
  {
    Console.WriteLine("Using " + connection);
  }

  public override void AfterTest(TestDetails testDetails)
  {
    Console.WriteLine("Closing connection.");
  }
}
```

and applying it to the fixture:

```
[TestFixture, InitializeDatabaseConnection("Username=Foo, Password=Bar")]
public class SetupTeardownFlow
{
  ... etc ...
```

results in the following test flow:

```
Before all test fixtures.
Fixture Setup.
Using Username=Foo, Password=Bar
***** UnitTestExamplesNUnit.SetupTeardownFlow.TestA
Test Setup.
Test A.
Test Teardown.
***** UnitTestExamplesNUnit.SetupTeardownFlow.TestB
Test Setup.
Test B.
Test Teardown.
Closing connection.
Fixture Teardown.
After all test fixtures.
```

Note that the user-defined action for a suite is executed after the “namespace” setup and after the fixture setup. If the user-defined action needs some information that cannot be passed in using the constructor (attribute constructors are restricted to constants) then an approach similar to that outlined next might be useful. However, keep in mind that the user-defined actions should be as autonomous as possible—entanglement with test states should be avoided.

Passing Information from a User-Defined Action to a Test/Test Suite

This is not possible without some form of boundary-crossing container. An extremely simple implementation is to create an object bag from a key-value dictionary:

```
/// <summary>
/// A wrapper class for a key-value lookup.
/// </summary>
public class ObjectBag : Dictionary<string, object>
{
}

/// <summary>
/// A global container class.
/// </summary>
public static class Globals
{
  public static ObjectBag objectBag = new ObjectBag();
}
```

which allows disconnected objects to easily share data, such as in the user-defined action attribute:

```
public class SetupData : TestActionAttribute
{
  public override void BeforeTest(TestDetails testDetails)
  {
```

```
    Globals.objectBag["FileX"] = "Some Data";
  }
}
```

The data can then be accessed in the test methods:

```
[Test]
public void TestA()
{
  Console.WriteLine("FileX = "+Globals.objectBag["FileX"].ToString());
}
```

More sophisticated approaches exist and the preceding examples are intended to illustrate how to coordinate a user-defined action with the sharing of information. The prior example has numerous problems—type safety, no checks for whether an object already exists in the container, and so forth—issues that a robust container class should handle.

NUnit Core Assertions

NUnit provides the same or similar assertions in the NUnit.Framework.Assert class as those found in Visual Studio's Assert class, as described earlier. NUnit also provides the additional assertions:

IsEmpty/IsNotEmpty

These assertions verify that a string is empty or not, or that a collection is empty or not:

```
[TestFixture]
public class OtherAssertions
{
  [Test]
  public void IsEmptyTest()
  {
    List<object> itemList = new List<object>();
    Assert.IsEmpty(itemList);
  }
}
```

Greater/Less

These two assertions work with numeric values and objects implementing IComparable, asserting that the first value is either greater or less than the second value. The purpose for this assertion is that it improves readability of the test.

GreaterOrEqual/LessOrEqual

These two assertions work with numeric values and objects implementing IComparable, asserting that the first value is either “greater than or equal” or “less than or equal” to the second value. The purpose for this assertion is that it improves readability of the test.

IsAssignableFrom/IsNotAssignableFrom

This pair of methods asserts that an expected type is assignable from an object instance of compatible type. This is similar to the IsInstanceOfType assertion in Visual Studio’s test engine. For example, given:

```
public class A { }
public class B : A { }

[TestFixture]
public class OtherAssertions
{
  [Test]
  public void IsAssignableFrom()
  {
    Assert.IsAssignableFrom(typeof(B), new A());
  }
}
```

This test passes because the object of type A can be assigned an instance of type B.

Throws/Throws<T>/DoesNotThrow

The non-generic form of this assertion validates that an exception is thrown (or not) when invoking the specified method. For example:

```
[Test]
public void DivideByZeroThrownTest()
{
  Assert.Throws(typeof(ArgumentOutOfRangeException), () => MyMath.Divide(5, 0));
}
```

This test passes.

The generic version places the exception type as the generic parameter:

```
[Test]
public void DivideByZeroThrownTest()
{
  Assert.Throws<ArgumentOutOfRangeException>(() => MyMath.Divide(5, 0));
```

```
}
```

Catch/Catch<T>

These two methods assert that a method throws a type of exception or one derived from it. In contrast, the Throws method before this asserts that exactly the specified exception type is thrown. For example, this code passes:

```
[Test]
public void CatchTest()
{
  Assert.Catch<Exception>(() => MyMath.Divide(5, 0));
}
```

Collection Assertions

In the NUnit.Framework.CollectionAssert class, NUnit implements the same assertions as found in Visual Studio's CollectionAssert class, with these additional assertions:

- IsEmpty/IsNotEmpty
- IsOrdered

Note that the collection parameter in these methods expects the collection to implement IEnumerable (contrast this with Visual Studio's test framework, which expects ICollection).

IsEmpty/IsNotEmpty

The CollectionAssert.IsEmpty method is the same as the Assert.IsEmpty method.

IsOrdered

This method verifies that the values in a collection are ordered. For example, because AnObject implements IComparable:

```
public class AnObject : IComparable
{
  public int SomeValue { get; set; }

  public override bool Equals(object obj)
  {
    return SomeValue == ((AnObject)obj).SomeValue;
  }

  public int CompareTo(object obj)
```

```
  {
    return SomeValue.CompareTo(((AnObject)obj).SomeValue);
  }
}
```

this test passes:

```
[Test]
public void OrderedObjectTest()
{
  List<object> itemList = new List<object>()
  {
    new AnObject() { SomeValue = 1 },
    new AnObject() { SomeValue = 2 }
  };

  CollectionAssert.IsOrdered(itemList);
}
```

String Assertions

These assertion methods are members of the NUnit.Framework.StringAssert class and are the same as found in Visual Studio's StringAssert class, with a couple of differences:

- AreEqualIgnoringCase
- IsMatch

which are discussed next.

AreEqualIgnoringCase

The assertion "AreEqualIgnoringCase" performs a comparison of two strings ignoring case.

IsMatch

This method is similar to Visual Studio's StringAssert.Matches method—it asserts that a string matches a regex pattern.

File Assertions

These assertion methods are members of the NUnit.Framework.FileAssert class.

AreEqual/AreNotEqual

These methods assert that two files are identical or not. A byte-by-byte comparison is made of the two files. If the files are empty, they are considered equal. If either or both files do not exist, a FileNotFoundException is thrown.

Directory Assertions

These assertion methods are members of the NUnit.Framework.DirectoryAssert class.

AreEqual/AreNotEqual

These methods assert that the files in the two directories are equal nor not.

IsEmpty/IsNotEmpty

These methods assert that a directory is empty or not.

IsWithin/IsNotWithin

These methods assert that the path contains the specified subdirectory or not.

Other Assertions

That

While one form of this method does take a Boolean parameter as an assertion of truth (behaving exactly like Assert.IsTrue), the predominant form of this method is the ability to compare an actual value using a specific constraint. The NUnit documentation on constraints is quite extensive regarding constraints, so only a short example will be provided here. For example, given this test:

```
[TestFixture]
public class CollectionTests
{
  [Test]
  public void AreEquivalentTest()
  {
    List<object> itemList1 = new List<object>()
    {
        new AnObject() { SomeValue = 1 },
```

```
        new AnObject() { SomeValue = 2 }
    };
    List<object> itemList2 = new List<object>()
    {
        new AnObject() { SomeValue = 2 },
        new AnObject() { SomeValue = 1 }
    };

    CollectionAssert.AreEquivalent(itemList1, itemList2);
  }
}
```

Using the constraint system, the assertion could be rewritten as:

```
[TestFixture]
public class CollectionTests
{
  [Test]
  public void AreEquivalentTest()
  {
    List<object> itemList1 = new List<object>()
    {
        new AnObject() { SomeValue = 1 },
        new AnObject() { SomeValue = 2 }
    };
    List<object> itemList2 = new List<object>()
    {
        new AnObject() { SomeValue = 2 },
        new AnObject() { SomeValue = 1 }
    };

    Assert.That(itemList1, Is.EquivalentTo(itemList2);
  }
}
```

The constraint syntax facilitates the readability of the unit test as well as promotes an extensible architecture for new methods.

IsNan

This method tests whether a double value is “not a number.” A typical case where a value is not a number is division by zero or the square root of a negative value.

Utility Methods

The assert class also provides several utility methods:

Pass

Assert.Pass ends the test immediately by throwing a SuccessException, which results in the test being marked as passing.

Fail

Assert.Fail ends the test immediately by throwing an AssertionException, which results in the test being marked as failing.

Ignore

Assert.Ignore ends the test immediately by throwing an IgnoreException, reporting the test as being ignored.

Inconclusive

Assert.Inconclusive ends the test immediately by throwing an InconclusiveException, reporting the test as being inconclusive. This might be useful when a unit test or fixture determines that a required service is down and therefore the tests against that service cannot proceed.

Chapter 11 Advanced Unit Testing

Cyclometric Complexity

Cyclometric complexity is a measure of the number of independent paths through your code. Code coverage testing is intended to ensure that your tests execute all possible code paths. Code coverage testing is available in Visual Studio's Test, Premium, and Ultimate versions. Code coverage is not part of NUnit. A third-party solution, NCover, is also available.

To illustrate this issue, consider this code, which parses command line parameters:

```
public static class CommandLineParser
{
  /// <summary>
  /// Returns a list of options based on brute force parsing
  /// of command line options. The function returns the
  /// specified options and the option parameter if required.
  /// </summary>
  public static Dictionary<string, string> Parse(string cmdLine)
  {
    Dictionary<string, string> options = new Dictionary<string, string>();
    string[] items = cmdLine.Split(' ');
    int n = 0;

    while (n < items.Length)
    {
      string option = items[n];
      string param = String.Empty;

      if (option[0] != '-')
      {
        throw new ApplicationException("Expected an option.");
      }

      if (option == "-f")
      {
        // Has the parameter been supplied?
        if (items.Length <= n + 1)
        {
          throw new ApplicationException("Filename not specified.");
        }

        param = items[n + 1];

        // Is it a parameter or another option?
        if (param[0] == '-')
```

```
        {
          throw new ApplicationException("Filename not specified.");
        }

        ++n;         // Skip the filename option parameter.
      }

      options[option] = param;
      ++n;
    }

    return options;
  }
}
```

and a couple of simple tests (note that these tests omit the code paths that result in exceptions being thrown):

```
[TestFixture]
public class CodeCoverageTests
{
  [Test]
  public void CommandParserTest()
  {
    Dictionary<string, string> options = CommandLineParser.Parse("-a -b");
    Assert.That(options.Count == 2, "Count expected to be 2");
    Assert.That(options.ContainsKey("-a"), "Expected option '-a'");
    Assert.That(options.ContainsKey("-b"), "Expected option '-b'");
  }

  [Test]
  public void FilenameParsingTest()
  {
    Dictionary<string, string> options = CommandLineParser.Parse("-f foobar");
    Assert.That(options.Count == 1, "Count expected to be 1");
    Assert.That(options.ContainsKey("-f"), "Expected option '-f'");
    Assert.That(options["-f"] == "foobar");
  }
}
```

Now let's look at what a code coverage test might look like, first by writing a poor man's code coverage helper:

```
public static class Coverage
{
  public static List<int> CoveragePoints {get; set;}

  public static void Reset()
```

```
  {
    CoveragePoints = new List<int>();
  }

  [Conditional("DEBUG")]
  public static void Set(int coveragePoint)
  {
    CoveragePoints.Add(coveragePoint);
  }
}
```

We'll also need a simple extension method; you'll see why in a minute:

```
public static class ListExtensions
{
  public static bool HasOrderedItems(this List<int> itemList, int[] items)
  {
    int n = 0;

    foreach (int i in itemList)
    {
      if (i != items[n])
      {
        return false;
      }

      ++n;
    }

  return true;
  }
}
```

Now we can add coverage set-points in our code, which will be compiled into the application when it is compiled in DEBUG mode (the bold red lines where added):

```
public static class CommandLineParser
{
  /// <summary>
  /// Returns a list of options based on brute force parsing
  /// of command line options. The function returns the
  /// specified options and the option parameter if required.
  /// </summary>
  public static Dictionary<string, string> Parse(string cmdLine)
  {
    Dictionary<string, string> options = new Dictionary<string, string>();
    string[] items = cmdLine.Split(' ');
    int n = 0;
```

```
    while (n < items.Length)
    {
      Coverage.Set(1);           // WE ADD THIS COVERAGE SET-POINT

      string option = items[n];
      string param = String.Empty;

      if (option[0] != '-')
      {
        throw new ApplicationException("Expected an option.");
      }

      if (option == "-f")
      {
        Coverage.Set(2);           // WE ADD THIS COVERAGE SET-POINT

        // Has the parameter been supplied?
        if (items.Length <= n + 1)
        {
          throw new ApplicationException("Filename not specified.");
        }

        param = items[n + 1];

        // Is it a parameter or another option?
        if (param[0] == '-')
        {
          throw new ApplicationException("Filename not specified.");
        }

        ++n;         // Skip the filename option parameter.
      }

      options[option] = param;
      ++n;
    }

    return options;
  }
}
```

And now we can write the following test fixture:

```
[TestFixture]
public class CommandParserCoverageTests
{
```

```
  [SetUp]
  public void CoverageSetup()
  {
    Coverage.Reset();
  }

  [Test]
  public void CommandParserTest()
  {
    Dictionary<string, string> options = CommandLineParser.Parse("-a -b");
    Assert.That(Coverage.CoveragePoints.HasOrderedItems(new [] {1, 1}));
  }

  [Test]
  public void FilenameParsingTest()
  {
    Dictionary<string, string> options = CommandLineParser.Parse("-f foobar");
    Assert.That(Coverage.CoveragePoints.HasOrderedItems(new [] { 1, 2 }));
  }
}
```

Note how we are now ignoring the actual results, but are ensuring that the desired code blocks are being executed.

White Box Testing: Inspecting Protected and Private Fields and Methods

Arguably, a unit test should only concern itself with public fields and methods. The counterargument for this is that in order to test the entire implementation, access to protected or private fields, to assert their state, and the ability to unit test protected or private methods is required. Considering that it is probably not desirable to expose most low-level computations, and those are exactly the methods one wants to test, it is most likely that testing at least protected or private methods is necessary. There are several options available.

Exposing Methods and Fields in Test Mode

This example illustrates the concept:

```
public class DoesSomething
{
#if TEST
  public
#else
  private
#endif
  void SomeComputation()
```

```
  {
  }
}
```

While this is doable, it produces ugly source code and runs the serious risk that someone might actually call the method with the symbol TEST defined, only to discover that his or her code breaks in a production build where the TEST symbol is undefined.

Deriving a Test Class

As an alternative, if the methods are protected, consider deriving a test class:

```
public class DoesSomethingElse
{
  protected void SomeComputation() { }
}

public class DoesSomethingElseTesting : DoesSomethingElse
{
  public void TestSomeComputation()
  {
    base.SomeComputation();
  }
}
```

This allows you to instantiate the derived testing class and access a protected method via a publicly exposed method in the subclass.

Reflection

Lastly, one can use reflection for private methods or sealed classes. The following illustrates a private method and executing that method via reflection in a test:

```
public class DoesSomething
{
  private void SomeComputation() { }
}

[TestClass]
public class DoesSomethingTest
{
  [TestMethod]
  public void SomeComputationTest()
  {
    DoesSomething ds = new DoesSomething();
    Type t = ds.GetType();
```

```
    MethodInfo mi = t.GetMethod("SomeComputation", BindingFlags.Instance |
BindingFlags.NonPublic);
    mi.Invoke(ds, null);
  }
}
```

Chapter 12 Unit Testing for Other Purposes

Unit testing is also valuable for other purposes.

As Examples of Usage

One of the side benefits of unit testing is that it creates a large code base exemplifying how to use the code. For example, the code we saw earlier:

```
[Test]
public void FilenameParsingTest()
{
Dictionary<string, string> options = CommandLineParser.Parse("-f foobar");
Assert.That(options.Count == 1, "Count expected to be 1");
Assert.That(options.ContainsKey("-f"), "Expected option '-f'");
Assert.That(options["-f"] == "foobar");
}
```

documents an expected valid use case for the command line parser. Also consider writing unit tests—not only for your own code, but also to provide examples for third-party libraries. (See subsequent examples and the interesting things revealed about the Rectangle struct.)

Black Box Testing

Black box testing assumes that you do not know anything about the internals of the class or service and are verifying its behavior strictly from the publicly exposed interfaces. This is frequently necessary when you don't have the code available. For example, when working with a record management company, we were required to use a web service provided by a government agency to update records. By writing unit tests for the web service, we were able to prove that the documentation provided to us did not result in the expected behavior of the web service.

You might use this technique as well when working with code provided by different departments. For example, the database group might have its own white box unit testing; however, you should also verify that from a black-box perspective, the triggers and constraints have been programmed correctly by inspecting the result of transactions from the functionality that is exposed to you.

Test Your Assumptions

Unit testing can be a simple way to put together some tests regarding our assumptions about an API. Let's take the **`System.Drawing.Rectangle`** structure and test some seemingly reasonable assumptions about the implementation.

Test Constructor Assumptions

There are two **`Rectangle`** constructors: one having **`Point`** and **`Size`** parameters, the other having x, y, width, and height parameters. The documentation makes no indication of whether the size (the width or height) must be positive, so let's write a test to verify that we can construct a rectangle with a negative width or height:

```
[TestMethod]
public void RectangleNegativeSizeConstructorTest()
{
    Rectangle r = new Rectangle(0, 0, -4, -6);
}
```

All we are doing here in this test is verifying that no exceptions are thrown when we construct the rectangle, and indeed, this is the case:

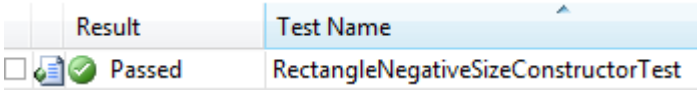


Figure 29: Rectangle Constructor Test

Test Assumptions Regarding Property Values

Now let's test our assumptions about certain properties. The properties **`Top`**, **`Left`**, **`Bottom`**, and **`Right`** are described as (see http://msdn.microsoft.com/en-us/library/system.drawing.rectangle.aspx):

Top: Gets the y-coordinate of the top edge of this Rectangle structure.

Left: Gets the x-coordinate of the left edge of this Rectangle structure.

Bottom: Gets the y-coordinate that is the sum of the Y and Height property values of this Rectangle structure.

Right: Gets the x-coordinate that is the sum of X and Width property values of this Rectangle structure.

So, with the preceding rectangle, with a negative width and height, and therefore having coordinates [(-4, -6), (0, 0)], we would make the following assumptions:

```
[TestMethod]
public void TestLeft()
{
    Rectangle r = new Rectangle(0, 0, -4, -6);
    Assert.IsTrue(r.Left == -4, "Expected Left == -4 but was " + r.Left);
}

[TestMethod]
public void TestTop()
{
    Rectangle r = new Rectangle(0, 0, -4, -6);
    Assert.IsTrue(r.Top == 0, "Expected Top == 0 but was " + r.Top);
}

[TestMethod]
public void TestRight()
{
    Rectangle r = new Rectangle(0, 0, -4, -6);
    Assert.IsTrue(r.Right == 0, "Expected Right == 0 but was " + r.Right);
}

[TestMethod]
public void TestBottom()
{
    Rectangle r = new Rectangle(0, 0, -4, -6);
    Assert.IsTrue(r.Bottom == -6, "Expected Bottom == -6 but was " + r.Bottom);
}
```

However, this is not the case:

Result	Test Name	Error Message
Passed	RectangleNegativeSizeConstru	
Passed	TestBottom	
Failed	TestLeft	Assert.IsTrue failed. Expected Left == -4 but was 0
Failed	TestRight	Assert.IsTrue failed. Expected Right == 0 but was -4
Passed	TestTop	

Figure 30: Testing Assumptions about Rectangle Properties

In fact, the determination of top and bottom appears totally arbitrary as well, as I have run tests on exactly the same rectangle dimensions and observed different results in the **`Top`** and **`Bottom`** property values.

Test Assumptions about Method Results

The MSDN documentation states that the **`Rectangle.Intersect`** method:

- *Returns a third Rectangle structure that represents the intersection of two other Rectangle structures.*
- *If there is no intersection, an empty Rectangle is returned.*

Therefore, we can construct a simple test:

```
[TestMethod]
public void TestIntersection()
{
    Rectangle r1 = new Rectangle(0, 0, 10, 10);
    Rectangle r2 = new Rectangle(10, 10, 5, 5);
    Assert.IsFalse(r1.IntersectsWith(r2), "Expected R1 and R2 not to intersect.");
    Assert.IsTrue(Rectangle.Intersect(r1, r2) == Rectangle.Empty,
                  "Expected an empty intersection rectangle.");
}
```

with the result:

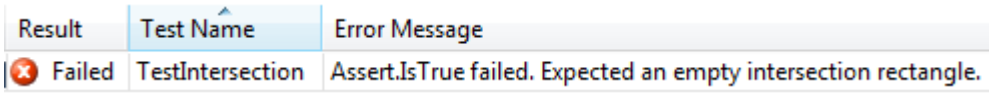


Figure 31: Testing Our Assumptions about Method Returns

This informs us that our expectation, based on the documentation, is incorrect.

In Conclusion

Unit testing is an important tool in the testing process. While integration and usability testing are often more customer-centric (reporting, milestones, verifying high-level requirements), unit testing is the first line of defense for a programmer, his or her team, and the team managers. If used judiciously (remember, you are not aiming to create thousands of pretty green lights), it can be a cost-effective way of verifying the computational correctness of the code and for re-creating bugs and verifying that they have been fixed.

However, good unit testing practices require a disciplined approach, a commitment to the time and effort required to implement and maintain the tests, and, from a coder's perspective, it also requires good programming practices and often enforces architectural decisions. The latter may fly in the face of "just get it done" constraints (which may be quite legitimate), and may potentially impact performance. On the upside, the programming practices and architectures that unit testing forces one to use are often of great benefit to the entire application development process, thus reducing costs and improving maintainability, not because the code is unit tested, but because the code is written better so that it *can* be unit tested.