Unit test

# What is Unit Test?

Unit testing is, roughly speaking, testing bits of your code in isolation with test code. Unit testing is the process of writing code to test the behavior and functionality of your system.

Note that if your test code writes to a file, opens a database connection or does something over the network, it's more appropriately categorized as an integration test. Integration tests are a good thing, but should not be confused with unit tests. Unit test code should be short, sweet and quick to execute.

Another way to look at unit testing is that you write the tests first.

Sometimes writing unit tests can be painful. When it gets that way, try to find someone to help you, and resist the temptation to "just write the damn code". Unit testing is a lot like washing the dishes. It's not always pleasant, but it keeps your metaphorical kitchen clean, and you really want it to be clean.

# Advantages:

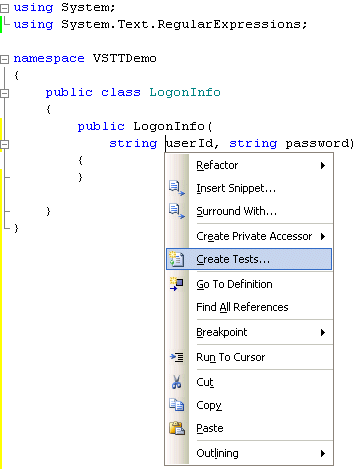
1. Make it easier to change the technical implementation while making sure you don't change the behavior (refactoring). Properly unit tested code can be aggressively refactored/ cleaned up with little chance of breaking anything.
2. Give developers confidence when adding behavior or making fixes.
3. Document your code
4. Indicate areas of your code that are tightly coupled. It's hard to unit test code that's tightly coupled
5. Provide a means to use your API and look for difficulties early on
6. Indicates methods and classes that aren't very cohesive

# Unit Test using VS Test Tool with C#

## Getting Started

To begin, we create a Class Library project named "*VSTSDemo*." By default, the **Create directory for solution** check box is selected. Leaving this option will enable us to create the test project in a separate directory alongside the VSTSDemo project. In contrast, deselecting this option creates a directory structure in which Visual Studio 2005 will place the test project into a subdirectory of the VSTSDemo project. The test project follows the Visual Studio convention of creating additional projects in subdirectories of the solution files path.

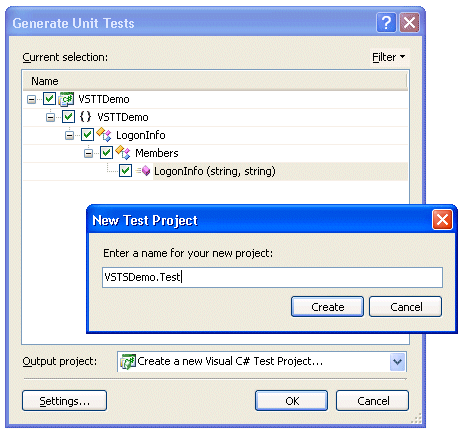
After creating the initial VSTSDemo project, we use the Visual Studio Solution Explorer to rename our Class1.cs file to *LogonInfo.cs*. This will also update the class name to be **LogonInfo**. Next, we modify the constructor to accept two string parameters named **userId** and **password**. Once the constructor signature is declared, we are ready to generate the tests for the constructor.



**Figure 2. The "Create Tests..." menu item from the context menu on the LogonInfo constructor**

## Creating a Test

Before we begin writing any implementation for **LogonInfo**, we follow the TDD practice of first writing a test. TDD isn't required for using Team Test, however, it is best practice we follow through the rest of this article. We do this by right-clicking on the **LogonInfo()** constructor and selecting the **Create Tests...** menu item (shown in Figure 2). This will display a dialog for generating unit tests into a different project (see Figure 3). By default, the Output project setting is for a new Visual Basic project, but C# and C++ test projects are also available. For this article, we will select Visual C# and click the **OK** button, followed by entering a project name of *VSTSDemo.Test* for the project name.



**Figure 3. Generate unit tests dialog**

The generated test project contains four files related to testing.

|  |  |
| --- | --- |
| **File Name** | **Purpose** |
| AuthoringTest.txt | Provides notes about authoring tests including instructions on adding additional tests to the project. |
| LogonInfoTest.cs | Includes generated test for testing the **LogonInfo()** constructor along with methods for test initialization and cleanup. |
| ManualTest1.mht | Provides a template to fill in with instructions for manual testing. |
| UnitTest1.cs | An empty unit test class skeleton in which to place additional unit tests. |

Because we aren't going to be doing any manual testing in this project, and because we already have a unit test file to work with, we will delete ManualTest1.mht and UnitTest1.cs.

In addition to some default files, the generated test project contains references to both the Microsoft.VisualStudio.QualityTools.UnitTestFramework and the VSTSDemo project, which the unit tests are executing against. The former is the testing framework assembly the test engine depends on when executing the unit tests. The latter is a project reference to the target assembly we are testing.

By default, the generated test method is a placeholder with the following implementation:

**Listing 1. Generated test method, ConstructorTest(), inside VSTSDemo.Test.LogonInfoTest**

/// <summary>

///This is a test class for VSTTDemo.LogonInfo and is intended

///to contain all VSTTDemo.LogonInfo Unit Tests

///</summary>

[TestClass()]

public class LogonInfoTest

{

// ...

/// <summary>

///A test case for LogonInfo (string, string)

///</summary>

[TestMethod()]

public void ConstructorTest()

{

string userId = null; // TODO: Initialize to an appropriate value

string password = null; // TODO: Initialize to an appropriate value

LogonInfo target = new LogonInfo(userId, password);

// TODO: Implement code to verify target

Assert.Inconclusive(

"TODO: Implement code to verify target");

}

}

The exact generated code will vary depending on the method type and signature that the test targets for testing. For example, the wizard will generate reflection code for testing private member functions. In this particular case, we have code specific for public constructor testing.

There are two important attributes related to testing with Team Test. First, the designation of a method as a test is through the **TestMethodAttribute** attribute. In addition, the class containing the test method has a **TestClassAttribute** attribute. Both of these attributes are found in the Microsoft.VisualStudio.QualityTools.UnitTesting.Framework namespace. Team Test uses reflection to search a test assembly and find all the **TestClass** decorated classes, and then find the corresponding **TestMethodAttribute** decorated methods to determine what to execute. One other important criteria, validated by the execution engine but not the compiler, is that the test method signature be an instance method that takes no parameters. The name is irrelevant because reflection searches for the **TestMethodAttribute**.

The test method (**ConstructorTest()**) instantiates the target **LogonInfo** class before asserting that the test is inconclusive (**Assert.Inconclusive()**). When the test is run, the **Assert.Inconclusive()** provides an indication that it is likely to be missing the correct implementation. In our case, we update the **ConstructorTest()** method so that it checks the initialization of user ID and password as shown below.

**Listing 2. Updated ConstructorTest() implementation**

/// <summary>

///A test case for LogonInfo (string, string)

///</summary>

[TestMethod()]

public void ConstructorTest()

{

string userId = "IMontoya";

string password = "P@ssw0rd";

LogonInfo logonInfo = new LogonInfo(userId, password);

Assert.AreEqual<string>(userId, logonInfo.UserId,

"The UserId was not correctly initialized.");

Assert.AreEqual<string>(password, logonInfo.Password,

"The Password was not correctly initialized.");

}

Note that the checks are done using the **Assert.AreEqual<T>()** method. The **Assert** method also supports an **AreEqual()**, without generics, but the generic version is almost always preferred because it will verify at compile time that the types match—a common error in unit testing frameworks available before generics were supported in the CLR.

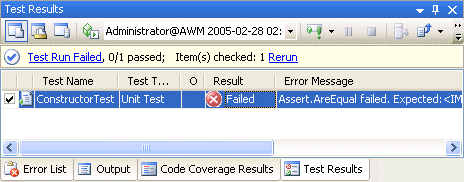
Because instance fields or properties for UserId and Password have not yet been created, we need to go back and add these to the **LogonInfo** class as well, in order for the VSTTDemo.Test project to compile.

Even though we don't yet have a valid implementation, let's proceed to running the tests. If we follow the TDD approach, we shouldn't be writing any production code until we have tests demonstrating that we need such code. We violated this principle in order to get the project structure established, but once it has been set up, it is easy to follow the TDD approach throughout.

## Running Tests

To run all tests within the project simply requires running the test project. To enable this action, we need to right-click on the VSTSDemo.Test project within the solution explorer and click **Set as StartUp Project**. Next, use the **Debug->Start (F5)** or **Debug->Start Without Debugging (Ctrl+F5)** menu items to begin running the tests.

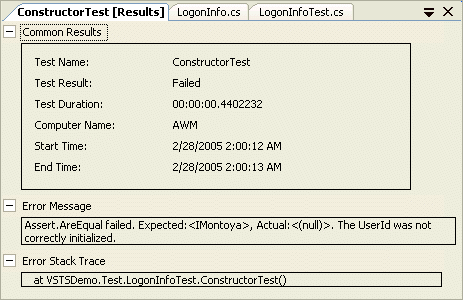
The Test Results window will appear listing all the tests within the project. Because our project only includes one test, only one will appear. Initially, the test will be in a state of pending, but once the test completes it will have a result of **Failed** as we would expect (see Figure 4).



**Figure 4. Test Results window after executing all tests**

Figure 4 shows the **Test Results** windows. This particular screenshot displays the Error Message in addition to the default columns. Columns can be added or removed by right-clicking the menu on the column headers and selecting the **Add/Remove Columns...** menu item.

To view additional details on the test, we can double click on it to open up the "ConstructorTest [Results]" window shown in Figure 5.



**Figure 5. Detailed test ConstructorTest [Results] windows**

In addition, we can right-click on individual tests and select the **Open Test** menu item to jump into the test code. Because we already know that the issue is in the LogonInfo constructor's implementation, we will go there and provide the code that initializes the **UserId** and **Password** fields using the passed in parameters. Re-run the tests to verify that they are now passing.

## Checking for Exceptions

The next step to creating our **LogonInfo** class is to provide some validation on the user ID and password. Unfortunately, the **UserId** and **Password** fields are public, which means that they don't provide any encapsulation that ensures they remain valid. Before we convert them to properties and provide any validation, however, let's write some tests to verify that whatever we implement will be correct.

We begin with a test that prevents null or an empty string from being assigned to the user ID. The expectation will be that if null values are passed to the constructor, then an **ArgumentException** will be thrown. The test code is shown in Listing 3.

**Listing 3. Testing for an exception using ExpectedExceptionAttribute**

[TestMethod]

[ExpectedException(typeof(ArgumentException),

"A userId of null was inappropriately allowed.")]

public void NullUserIdInConstructor()

{

LogonInfo logonInfo = new LogonInfo(null, "P@ss0word");

}

[TestMethod]

[ExpectedException(typeof(ArgumentException),

"A empty userId was inappropriately allowed.")]

public void EmptyUserIdInConstructor()

{

LogonInfo logonInfo = new LogonInfo("", "P@ss0word");

}

Note that there is no try catch block with an explicit test for the **ArgumentException**. Rather, both tests include an additional attribute, **ExpectedException**, that takes a type parameter and optionally, the error message to be displayed if the exception is not thrown. When the unit tests execute, the framework will explicitly watch for an **ArgumentException** to be thrown, and if the method does not throw such an exception, the test will fail. Running the tests will demonstrate that we do no yet make any validation checks on user ID; therefore, the tests will fail because the expected exception will not be thrown.

Given failing tests, it is time to switch back to the production code and update it with the code that will provide the functionality the tests are checking for. In this case we convert the **UserId** field into a property and then provide the validation checks (Listing 4).

**Listing 4. Validating UserId inside the LogonInfo class**

public class LogonInfo

{

public LogonInfo(string userId, string password)

{

this.UserId = userId;

this.Password = password;

}

private string \_UserId;

public string UserId

{

get { return \_UserId; }

private set

{

if (value == null || value.Trim() == string.Empty)

{

throw new ArgumentException(

"Parameter userId may not be null or blank.");

}

\_UserId = value;

}

}

// ...

}

The property implementation uses the C# 2.0 functionality in which the setter and getter accessibility do not match. The set implementation is marked as private while the get implementation is public. As a result, the user ID cannot be modified (except through reflection) from outside the LogonInfo class.

Once the validation has been added, we are able to re-run the tests and verify that the implementation was correct. We run all three tests to verify that the refactoring of the **UserId** field into property did not introduce any unexpected errors. The true value of unit testing is really revealed over time as code is modified. A suite of unit tests verifies that we didn't break the code in the process of maintaining and improving it.

## Loading Test Data from a Database

For the next modification to our **LogonInfo** class, we will provide a change password method. This method will take a parameter for the old password and a parameter for the new password value. In addition, we will validate that the password conforms to certain complexity requirements. Specifically, we will ensure that the password matches the default Windows Active directory requirements, that it includes characters from three of the following four categories:

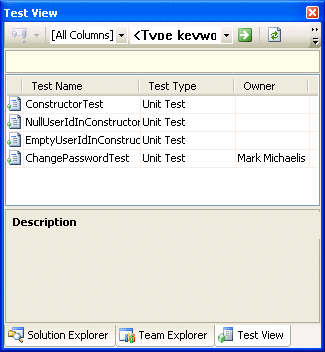
* Uppercase letters
* Lowercase letters
* Punctuation
* Numbers

In addition, we will check that the password is a minimum of 6 characters and a maximum of 255 characters.

As before, we would like to write our tests for the password complexity requirements before we write the implementation. Obviously, however, we will need to provide a large set of test values against which we can verify the implementation. Rather than creating an individual test for each test case, or even creating a loop that iterates through a number of test cases, we will instead create a data driven test that pulls the required data out of the database.

## Test View Windows

To begin, we define a new test called **ChangePasswordTest()**. After defining, the test method opens up the **Test View** window from the **Test->View and Author Tests** menu item as shown in Figure 6.



**Figure 6. Test view window**

The test view window can be used for running specific tests and browsing specific properties of a test. By adding additional columns (right-click on the header column and select **Add/Remove Columns...**) we can sort and view tests according to preference. Some of the columns are pulled from attributes decorating a test. For example, adding **OwnerAttribute** will display the test owner within the Owner column. Other metadata attributes like **DescriptionAttribute** are also supported. Each of these is found in the Microsoft.VisualStudio.QualityTools.UnitTesting.Framework namespace. If no explicit property exists, then we can use the free form **TestPropertyAttribute** to add name-value pairs to particular test methods.

Properties that do not have corresponding columns can be viewed from the properties window for a test (select a test and click **Properties** from the right-click context menu). This includes properties for specifying the Data Connection String and table name from which to load the test data. Obviously, in order to specify valid values we will need to have a database connection.

## Adding a Test Database

From the Server Explorer windows we can use the **Create new SQL Server Database** menu item. The caution that accompanies this method, however, is that if we want the tests to be executed on other computers, we need to be sure to create the database on a server that is accessible from the other computers that may be executing the test—a build machine for example.

Another option is simply to add a database file. Using **Project->Add new item...** allows the insertion of a SQL Database file into the project. This way the test data will follow the test project. The drawback is that we will not want to do this if the database is going to get large. Rather, we will want to provide some global data source.

For the data in this project we created a local project database file named *VSTSDemo.mdf*. To add the test data to the file, we use the **Tools->Connect to Database** menu item and specify the VSTSDemo.mdf file. Then, from the Server Explorer window we can add a new table called **LogonInfoTest** using the designer. Listing 5 shows the table definition.

**Listing 5. LogonInfoTestData SQL script**

CREATE TABLE dbo.LogonInfoTest

(

UserId nchar(256) NOT NULL PRIMARY KEY CLUSTERED,

Password nvarchar(256) NULL,

IsValid bit NOT NULL

) ON [PRIMARY]

GO

After saving the table, we can open it and enter various invalid passwords as shown in the table below.

|  |  |  |
| --- | --- | --- |
| **UserId** | **Password** | **IsValid** |
| Humperdink | P@w0d | false |
| IMontoya | p@ssword | false |
| Inigo.Montoya | P@ssw0rd | false |
| Wesley | password | false |

## Associating the Data with a Test

Once the creation of the table is complete, we need to associate it with the **InvalidPasswords()** test. From the properties windows of the **InvalidPassword** test, we fill in the **Data Connection String** property and **Data Table Name** property. Doing so will update the test with the additional attributes **DataSourceAttribute** and **DataTableNameAttribute**. The final **ChangePasswordTest()** method is shown in Listing 6.

**Listing 6. Test code for data driven test**

enum Column

{

UserId,

Password,

IsValid

}

private TestContext testContextInstance;

/// <summary>

///Gets or sets the test context which provides

///information about and functionality for the

///current test run.

///</summary>

public TestContext TestContext

{

get

{

return testContextInstance;

}

set

{

testContextInstance = value;

}

}

[TestMethod]

[Owner("Mark Michaelis")]

[TestProperty("TestCategory", "Developer"),

DataSource("System.Data.SqlClient",

"Data Source=.\\SQLEXPRESS;AttachDbFilename=\"<*Path to the sample .mdf file*>";Integrated Security=True",

"LogonInfoTest",

DataAccessMethod.Sequential)]

public void ChangePasswordTest()

{

string userId =

(string)TestContext.DataRow[(int)Column.UserId];

string password =

(string)TestContext.DataRow[(int)Column.Password];

bool isValid =

(bool)TestContext.DataRow[(int)Column.IsValid];

LogonInfo logonInfo = new LogonInfo(userId, "P@ssw0rd");

if (!isValid)

{

Exception exception = null;

try

{

logonInfo.ChangePassword(

"P@ssw0rd", password);

}

catch (Exception tempException)

{

exception = tempException;

}

Assert.IsNotNull(exception,

"The expected exception was not thrown.");

Assert.AreEqual<Type>(

typeof(ArgumentException), exception.GetType(),

"The exception type was unexpected.");

}

else

{

logonInfo.ChangePassword(

"P@ssw0rd", password);

Assert.AreEqual<string>(password, logonInfo.Password,

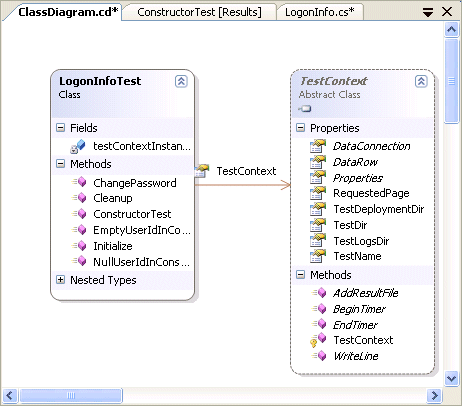
"The password was not changed.");

}

}

The first noteworthy characteristic of Listing 6 is the addition of the **DataSourceAttribute** in which the connection string, table name, and access order are specified. In this listing we use a database file name to identify the database. The advantage of this is that the file will travel with the test project, assuming it is updated to be a relative path.

The second important characteristic is the TestContext.DataRow calls. **TestContext** is a property that the generator provided when we ran the Create Tests wizard. The property is automatically assigned by the test execution engine at runtime so that within a test we can access data associated with the test context, as shown if Figure 7.



**Figure 7. TestContext association**

As Figure 7 shows, the **TestContext** provides data such as the **TestDirectory** and **TestName**, along with methods such as **BeginTimer()** and **EndTimer()**. What is most significant for the **ChangePasswordTest()** method is the **DataRow** property. Because the **ChangePasswordTest()** method is decorated with **DataSourceAttribute**, it will be called once for each record returned from the table specified by the attribute. This enables the test code to use the data within an executing test and have the test repeat for each record inserted into the LogonInfoTest table. If the table contains four records, then the test will be executed four separate times.

Using data driven tests like this makes it very easy to provide additional test data without writing any more code. As soon as an additional test case is needed, all we need to do is add the associated data to the LogonInfoTest table. Although we could have created two separate tests that used separate tables for valid and invalid data, this particular example merges the tests to show a slightly more complex data test example.

## Implementing and Refactoring the Target Method

Now that we have a test, it is time to write the implementation for the test. Using the C# refactoring tools, we can right-click on the **ChangePassword()** method call, select the **GenerateMethodStub** menu item, and then provide an implementation to the generated method. Once we have successfully run all tests using all the test data, we can begin refactoring the code as well. The final implementation for the **LogonInfo** class appears in Listing 7.

**Listing 7. LogonInfo class**

using System;

using System.Text.RegularExpressions;

namespace VSTTDemo

{

public class LogonInfo

{

public LogonInfo(string userId, string password)

{

this.UserId = userId;

this.Password = password;

}

private string \_UserId;

public string UserId

{

get { return \_UserId; }

private set

{

if (value == null || value.Trim() == string.Empty)

{

throw new ArgumentException(

"Parameter userId may not be null or blank.");

}

\_UserId = value;

}

}

private string \_Password;

public string Password

{

get { return \_Password; }

private set

{

string errorMessage;

if (!IsValidPassword(value, out errorMessage))

{

throw new ArgumentException(

errorMessage);

}

\_Password = value;

}

}

public static bool IsValidPassword(string value,

out string errorMessage)

{

const string passwordSizeRegex = "(?=^.{6,255}$)";

const string uppercaseRegex = "(?=.\*[A-Z])";

const string lowercaseRegex = "(?=.\*[a-z])";

const string punctuationRegex = @"(?=.\*\d)";

const string upperlowernumericRegex = "(?=.\*[^A-Za-z0-9])";

bool isValid;

Regex regex = new Regex(

passwordSizeRegex +

"(" + punctuationRegex + uppercaseRegex + lowercaseRegex +

"|" + punctuationRegex + upperlowernumericRegex + lowercaseRegex +

"|" + upperlowernumericRegex + uppercaseRegex + lowercaseRegex +

"|" + punctuationRegex + uppercaseRegex + upperlowernumericRegex +

")^.\*");

if (value == null || value.Trim() == string.Empty)

{

isValid = false;

errorMessage = "Password may not be null or blank.";

}

else

{

if (regex.Match(value).Success)

{

isValid = true;

errorMessage = "";

}

else

{

isValid = false;

errorMessage = "Password does not meet the complexity requirements.";

}

}

return isValid;

}

public void ChangePassword(

string oldPassword, string newPassword)

{

if (oldPassword == Password)

{

Password = newPassword;

}

else

{

throw new ArgumentException(

"The old password was not correct.");

}

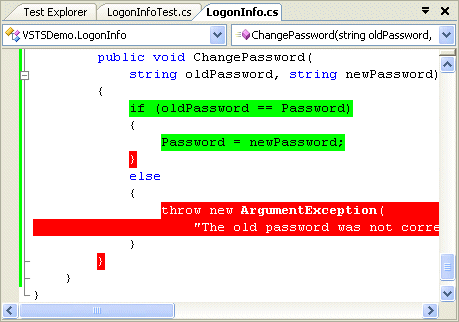
}

}

}

## Code Coverage

One key metric for unit testing is determining how much of the code is tested when the unit tests run. The metric is known as the code coverage and Team Test includes a code coverage tool for detailing both the percentage of code that is executed along with highlighting within the code to show which lines are executed and which lines are not. This feature is shown in Figure 8.



**Figure 8. Code coverage highlighting**

Figure 8 shows the code coverage highlighting after running all the unit tests. The red highlighting demonstrates we have production code that is not being run by any unit tests. This indicates that we wrote some code without following the TDD practice of providing tests before writing the implementation.

## Initializing and Cleaning Up Tests

In general, a test class contains not only the individual test methods, but various methods for initializing and cleaning up tests as well. In fact, the Create Tests wizard added some of these additional methods to the **LogonInfoTest** class when it created our VSTSDemo.Test project (see Listing 8).

**Listing 8. Final LogonInfoTest class**

using VSTTDemo;

using Microsoft.VisualStudio.QualityTools.UnitTesting.Framework;

using System;

namespace VSTSDemo.Test

{

/// <summary>

///This is a test class for VSTTDemo.LogonInfo and is intended

///to contain all VSTTDemo.LogonInfo Unit Tests

///</summary>

[TestClass()]

public class LogonInfoTest

{

private TestContext testContextInstance;

/// <summary>

///Gets or sets the test context which provides

///information about and functionality for the

///current test run.

///</summary>

public TestContext TestContext

{

get

{

return testContextInstance;

}

set

{

testContextInstance = value;

}

}

/// <summary>

///Initialize() is called once during test execution before

///test methods in this test class are executed.

///</summary>

[TestInitialize()]

public void Initialize()

{

// TODO: Add test initialization code

}

/// <summary>

///Cleanup() is called once during test execution after

///test methods in this class have executed unless

///this test class' Initialize() method throws an exception.

///</summary>

[TestCleanup()]

public void Cleanup()

{

// TODO: Add test cleanup code

}

// ...

[TestMethod]

// ...

public void ChangePasswordTest()

{

// ...

}

}

}

The test setup and cleanup methods are decorated with **TestInitializeAttribute** and **TestCleanupAttribute** respectively. Within each of these methods we can place any additional code that needs to be run before or after every test. This means that before each execution of **ChangePasswordTest()** corresponding to each record within the LogonInfoTest table, both **Initialize()** and **Cleanup()** will be executed. The same is true each time **NullUserIdInConstructor** and **EmptyUserIdInConstructor** execute. Such methods can be used to insert default data in a database and then clean up the data at the end of the test. It would be possible, for example, to begin a transaction within **Initialize()** and then rollback the same transaction during cleanup such that, assuming the test methods used the same connection, the data state would be restored at the end of each test execution. Similarly, test files could be manipulated.

During debugging it is possible that the **TestCleanupAttribute** decorated method will not run simply because the debugger was stopped before the cleanup code executed. For this reason, it is often a good practice to check for cleanup during test setup and execute the cleanup prior to the setup as necessary. Other test attributes available for initialization and cleanup are **AssemblyInitializeAttribute/AssemblyCleanupAttribute** and **ClassInitializeAttribute/ClassCleanupAttribute**. The assembly related attributes run once for an entire assembly while the class related attributes run once each for the loading of a particular test class.

## Best Practices

There are several unit testing best practices to review before we close. Firstly, TDD is an invaluable practice. Of all the development methodologies available, TDD is probably the one that will most radically improve development for many years to come and the investment is minimal. Any QA engineer will tell you that developers can't write successful software without corresponding tests. With TDD, the practice is to write those tests before even writing the implementation and ideally, writing the tests so that they can run as part of an unattended build script. It takes discipline to begin this habit, but once it is established, coding without the TDD approach feels like driving without a seatbelt.

For the tests themselves there are some additional principals that will help with successful testing:

* Avoid creating dependencies between tests such that tests need to run in a particular order. Each test should be autonomous.
* Use test initialization code to verify that test cleanup executed successfully and re-run the cleanup before executing a test if it did not run.
* Write tests before writing the any production code implementation.
* Create one test class corresponding to each class within the production code. This simplifies the test organization and makes it easy to choose where to places each test.
* Use Visual Studio to generate the initial test project. This will significantly reduce the number of steps needed when manually setting up a test project and associating it to the production project.
* Avoid creating other machine dependent tests such as tests dependent on a particular directory path.
* Create mock objects to test interfaces. Mock objects are implemented within a test project to verify that the API matches the required functionality.
* Verify that all tests run successfully before moving on to creating a new test. That way you ensure that you fix code immediately upon breaking it.
* Maximize the number of tests that can be run unattended. Make absolutely certain that there is no reasonable unattended testing solution before relying solely on manual testing.