Practical .NET Unit Testing

by Jason Young
Jason@YTechie.com
http://www.ytechie.com

WHY WRITE UNIT TESTS?

INTRODUCTION

When Ford (the automaker) finishes building an engine, it moves off the assembly line into a special testing area. In this area, the engine is connected to a special test harness that connects a gas and oil line. For the output, a measuring device is connected to the drive shaft that comes directly out of the engine. A starting motor is connected and the engine starts and revs up. Within about 5 minutes the computer is able to analyze the torque curve, gas usage, and oil usage.

What I've just described is an automated unit test:

- **Test Initialization:** Clean gas and oil connections are made, the starter motor is activated.
- **Test Inputs:** Gas & Oil
- **Exercise the Item Under Test:** Start and run the engine
- **Test Output Verification (Assertions)**: The torque curve of the engine, the amount of gas and oil used, and the engine exhaust.
- **Test Teardown**: The engine is disconnected from the inputs and outputs
- **Pass or Fail**: The engine output values are compared against the expected values to determine if the engine meets specifications.

Developers that have been writing software using a modular, testable design, understand the benefits of automated unit testing. Even if they are unable to quantify it, the effects on the entire development cycle are obvious to them.

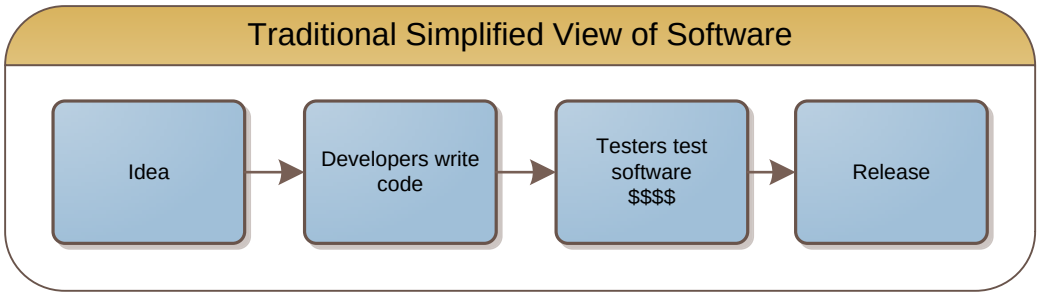
When I learned about the automated unit testing process using NUnit, I did not fully appreciate its usefulness. An early experience however, changed my mind. Years ago, I saw an opportunity to use NUnit to test a static function that parsed relative time strings ("now", "now - 1 day", "yesterday") into a DateTime object. The original parsing code was ported from VB, and had already been in use for some time by our end users.

I started feverishly writing tests, trying to think of every odd combination of input strings that the method would receive. I wrote a test for the case where the user forgot a space after the number they were entering, another test for the case where the user chained multiple operations, another test to check for varying case. When I finished, I had a fairly exhaustive suite of over 30 unit tests. I was shocked to learn that the code had failed on over 10 of those cases, and the original VB version suffered from a majority of same issues. In one hour, I had found an extensive list of bugs that **hadn't been reported by end users**. It is possible that they may not have run into them, but more realistically, they probably just learned to work around them. Because of this, I believe, the users' **overall impression of the system was lowered**, if only by a little bit. At this point, I was sold on unit testing.

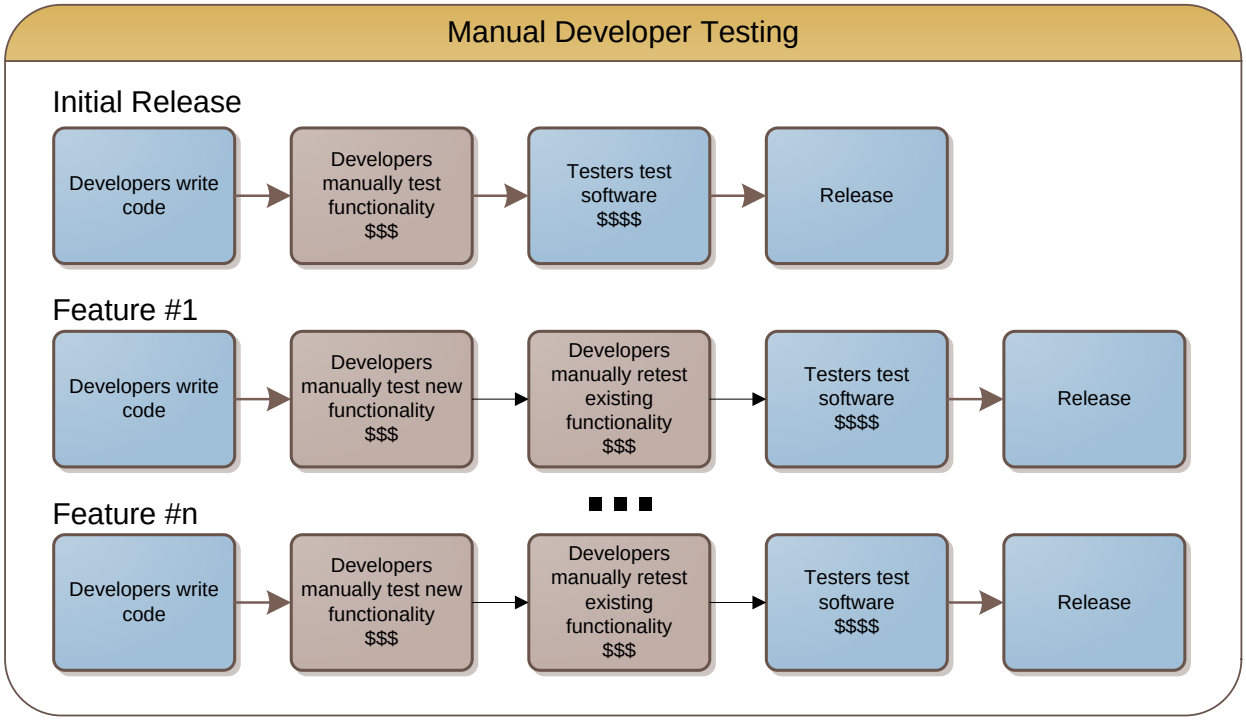
UNIT TESTING & MANAGERS

When the concept of unit testing is presented to a manager or a client managing a project, their reaction is often formed from a naive understanding of the process. They assume that it has about the same ROI as traditional system testing.

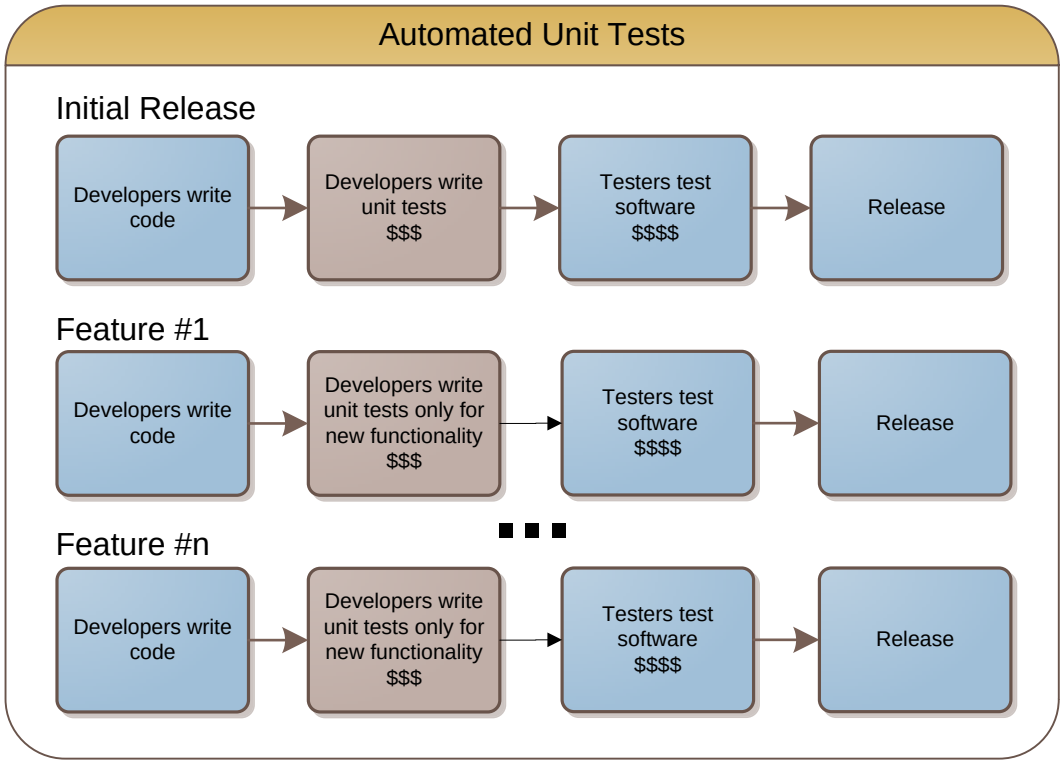
Let's take a look at an oversimplification of the traditional view of the software process:



The problem is, this diagram isn't entirely accurate. As part of the development process, developers are running frequent, manual tests to confirm that what they have written works as expected. If we're truthful in our diagram, our development process will look like this:



Replacing the manual developer tests with automated unit tests, results in a process that now looks like this:



WHAT UNIT TESTS REALLY DO

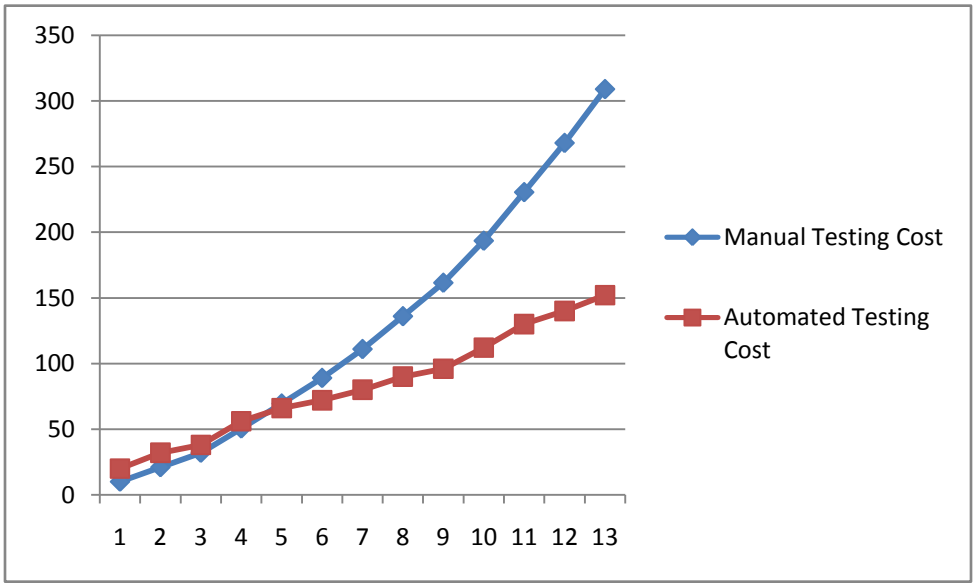
Unit tests are absolutely critical to writing complex, reliable software. Try to avoid comparing unit tests with traditional software testing. **They are NOT the same thing**, and **they have dramatically different purposes**. The idea behind automated unit testing is to **replace manual developer testing**. Manual developer tests are **time consuming** and **inconsistent**. If you were to pick any two developers and watch how they perform their manual testing, you would most likely see two different strategies.

In the manual/automated testing diagrams above, there is a key difference. Though both exhibit a similar return on investment for the first iteration, the automated approach avoids the investment required to manually retest previously completed code.

If we define ***Manual*** as the amount of time to do manual testing for the first iteration, and ***Automated*** as the amount of time to write and run automated unit testing, the ratio of ***Automated*** to ***Manual*** can vary depending on a number of factors including developer experience, problem complexity, and the problem being solved. For a developer that is relatively new to testing, it may take them twice as long to write and run unit tests as it would to manually test the application. In my experience, for a new developer testing, this ratio is often 2:1.

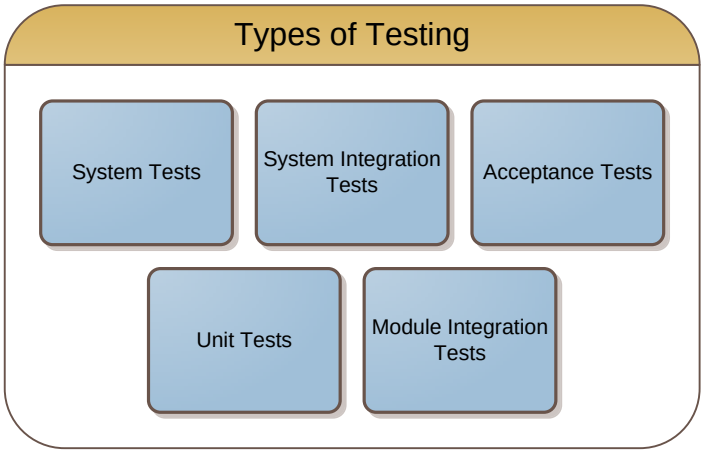
As the proficiency in writing tests and designing for testability increases, the ratio of ***Automated*** to ***Manual*** begins to decrease, and in some cases may even reach or drop below 1.0. Even assuming that there is a high initial cost with writing automated unit tests, it's easy to show that that cost is offset in future iterations. For example, assume the ***Automated***:***Manual*** ratio is 2:1 and that developer regression testing will take half the time that it took to originally develop the manual tests. Using these assumptions along with some pseudo-random data, we come up with the following sample:

Iteration/ Feature #	New Manual Hours	Manual Regression Hours	New Automated Hours	Accumulated Manual Hours	Accumulated Automated Hours
1	10	0	20	10	20
2	6	5	12	21	32
3	3	8	6	32	38
4	9	9.5	18	50.5	56
5	5	14	10	69.5	66
6	3	16.5	6	89	72
7	4	18	8	111	80
8	5	20	10	136	90
9	3	22.5	6	161.5	96
10	8	24	16	193.5	112
11	9	28	18	230.5	130
12	5	32.5	10	268	140
13	6	35	12	309	152



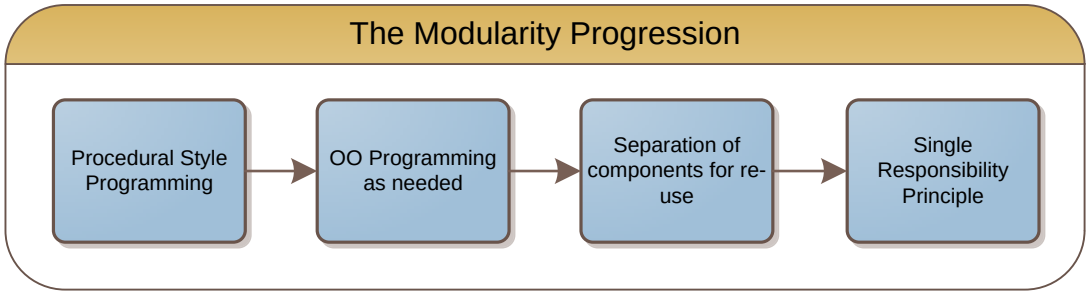
Initially, it's clear that the cost of automated testing is higher using the aforementioned assumptions. However, after the 5th iteration, the cost of manual testing starts to get out of control because **the amount of time testing is proportional to the size of the project**. In contrast, automated testing **incrementally** adds time to your testing, simplifies planning, and ensures consistent quality.

TYPES OF TESTING



In the big picture of testing, the intent of a unit test is to test the smallest pieces of logic in your software. The friendly structure of unit tests often makes them suitable for other purposes which may include (but is not limited to) module integration tests, acceptance tests, and performance tests. Because of this, we need to pay careful attention to what the actual purpose of our test is, so that we don't make the test difficult to write or run. It may be necessary to organize the unit tests by the purpose that they serve, so that they can be easily run at different times. For example, integration tests that use a unit testing framework may be separated into their own project.

Next, we need to understand the progression of software as an industry, as well as the typical progression of an individual developer:



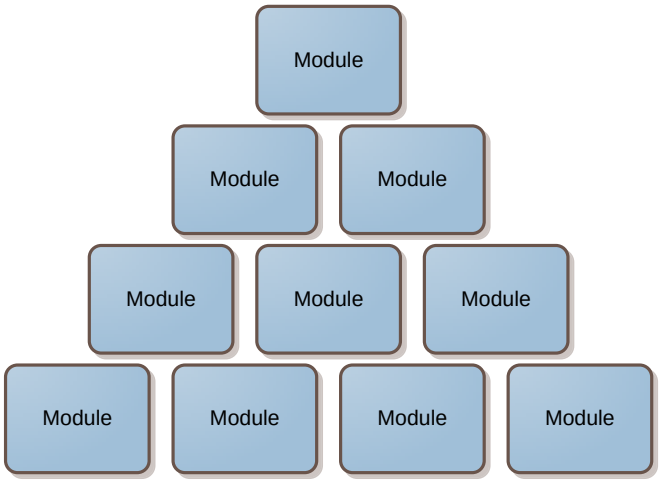
Initially, developers think in terms of a process, or procedure. They have a set of steps that the thing needs to get done. Once they start to think in code, they understand that this process is painful. As time progresses, developers start to structure their code in such a way as to increase its maintainability. Maintainability is a critical component of refactoring, bug fixes, feature changes, and new features.

Consider the automotive analogy presented earlier. Imagine if Ford could not, or chose not to test an engine this way? How much extra would it cost to run all of the engine tests when it was already assembled into a car? Even if they're already running a suite of tests on the car after it's built, imagine for a moment 1% of engines were bad. Once a tester put the car through its paces, he would **hopefully** realize that the car isn't performing as it should. The problem is that there is now a high cost of retooling. There would first have to be an analysis done on the car to confirm that the engine is in fact the problem. What if it's actually the transmission, or the cable connecting the gas pedal to the engine? What if the tester didn't even notice that the engine was sluggish because the cars computer was able to compensate for minor problems?

When we start testing components instead of a product, we get a tremendous amount of power.

Smaller parts can be tested, assembly workers or robots inspect elements as they're built. **The later a defect is discovered, the higher the cost to fix it**.

Software development is no longer a string of instructions for the computer to execute (i.e., no longer procedural). Today, the software we create is a hierarchy of modular components (i.e., object oriented) . Each layer in this structure becomes the foundation for the layer above. The lower the module is, the more important it becomes that it can be trusted. Just as a pyramid would suffer if just a single block from the base would not be able to hold the weight of the blocks above, higher level code cannot function properly if the lower level functions could not be trusted completely.



The truth is, **most developers actually do test their code**. However, instead of using a testing framework to verify their components, they run the actual product to determine the quality of the layers below. It's akin to making sure a pyramid is built properly by making sure the top block is correctly positioned. Though of some value, this form of testing is manual, prone to human mistakes, and non-exhaustive.

Now that we are imagining our code modules as a hierarchy of layers, we can begin to understand how difficult it is to test lower modules by examining the behavior of the higher modules. Using a top-down methodology, the various code paths are hidden from us. We're effectively throwing away our knowledge of the internal structure in favor of trying to discover it through the user interface. Employing the automaker analogy, we are trying to test that the engine is performing maximally by driving around (i.e., by manipulating the car through its user interface). In many projects, this type of testing may be necessary. **However, in the real world of tight budgets and deadlines, we can use unit testing to remove some of the guesswork from the process**.

UNIT TEST MECHANICS

There are a wide variety of unit testing tools available for .NET. Fortunately, the structure of unit tests is similar within most of the frameworks. NUnit has historically been one of the most popular frameworks, but it now faces competition because unit testing functionality is now integrated with many editions of Visual Studio.

There are 2 main parts of a unit testing system, the **testing framework** and a **test runner**. The testing framework is a way of defining tests. This is typically done by using standard classes and methods, but using special attributes to add unit testing meta data.

TESTING FRAMEWORK

Examine the following Visual Studio testing example of a class that has been decorated with testing attributes:

```
[TestClass]
public class String_Tester
{
	[TestMethod]
	public void Format_String_Single_Parameter_Verify_Substitution()
	{
		var result = string.Format("result: {0}", "test");
		Assert.AreEqual("result: test", result);
	}
}
```

To designate the class as a unit testing class, the "TestClass" attribute is added. The "TestMethod" attribute is added to a method that takes no parameters and returns void. This designates a method as one that will be executed as a test.

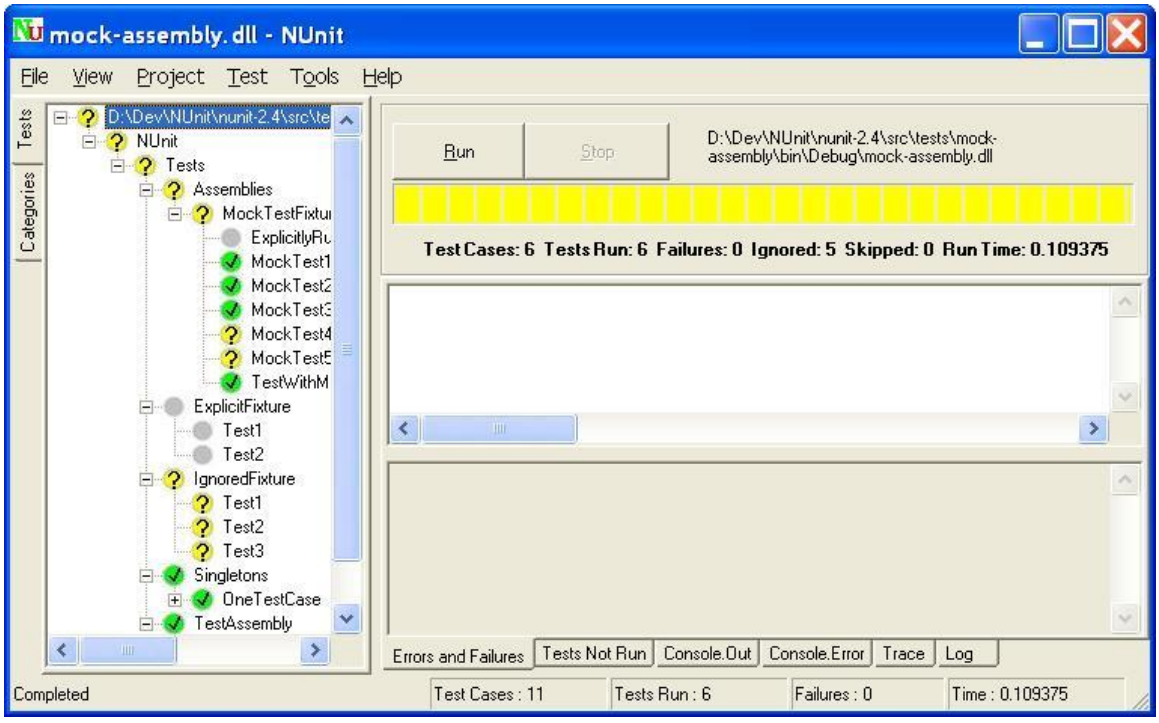
To reduce the amount of common code that is required to setup and tear down the test, there are other special methods within a test class. The *TestInitialize* attribute decorates a method that will be called before each unit test. This method helps define the initial state that must exist before a unit test is written. To execute code after each unit test is run, *TestCleanup* attribute can be used. Below is a table defining the most common attributes:

MS Test	NUnit	Purpose
TestClass	TestFixture	Designates a class as one containing unit tests
TestMethod	Test	Defines a single unit test
TestInitialize	SetUp	Run before each test is run
TestCleanup	TearDown	Run after each test is run
ClassInitialize	TestFixtureSetUp	Run one time before all of the unit tests
ClassCleanUp	TestFixtureCleanUp	Run one time after all of the unit tests

TEST RUNNER

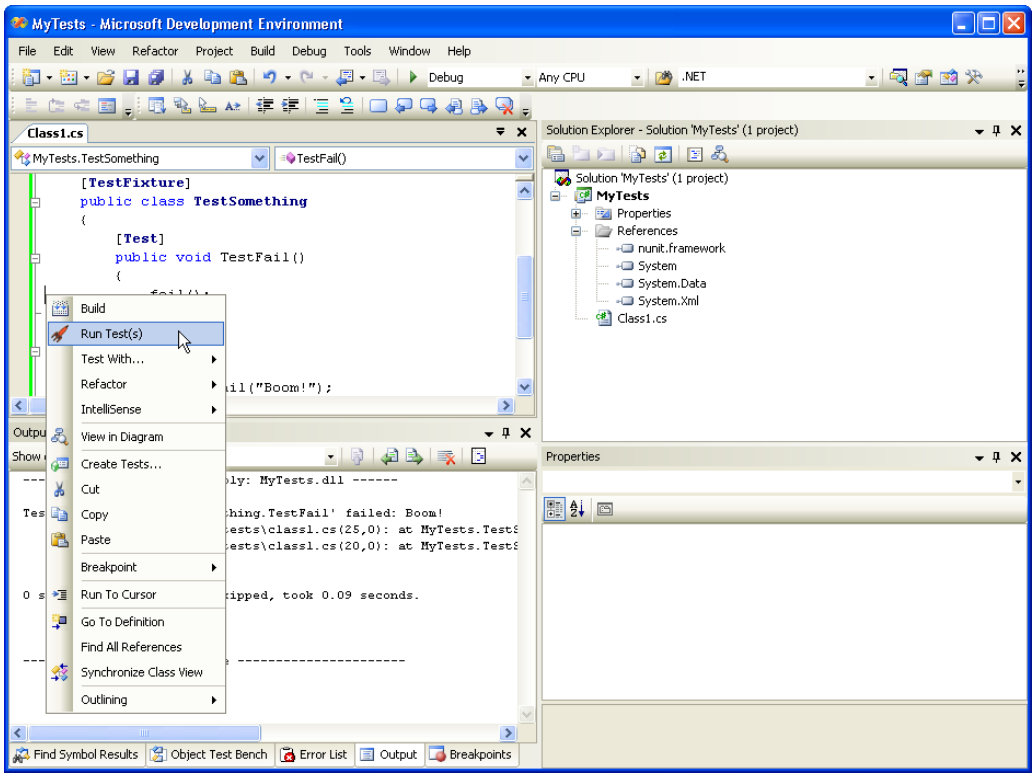
Having test classes and methods defined does not provide any value without a system for running those tests. Historically, the job of running unit tests has been the responsibility of a dedicated test application. Even today, NUnit provides a dedicated application that makes it easy to run your tests. The advantage of using this approach

is that it is simple and has no other required application dependencies. You simply reference your unit testing assembly and execute your tests.



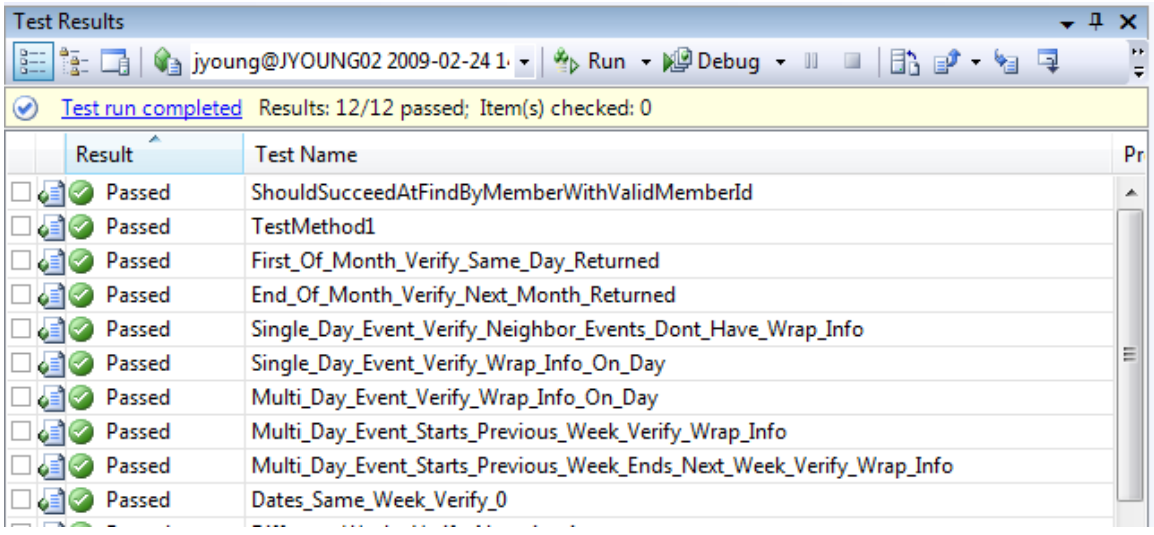
Frequently switching between a test-runner application and your development environment can be inefficient. Fortunately, there are a couple of solutions that integrate directly into your development environment. One such product is called TestDriven.NET, which is a Visual Studio add-in that supports all of the popular testing frameworks.

Using an integrated tool can have several advantages. First, there is less context switching for the developer, because all functionality is available without switching between applications. Second, options such as running specific tests or tests in a specific class become simple. The tool is able to deduce the tests to run based on the source of the context menu, as well as the cursor position within a test class. If the cursor is on a class or between methods, all of the tests in that class will be executed. If the cursor is on a specific test, only that test will be executed. Additionally, the context menu in the solution explorer has options for running tests within a certain file, project, or an entire solution.



Starting with Visual Studio 2005, a test runner was built into the product. Unfortunately, it was only available in the high-end Team System edition. Starting in Visual Studio 2008, however, unit testing is now available in any edition above Standard. Having unit testing functionality built directly in, and supported by Microsoft, helps significantly lower the barrier to entry.

Visual Studio, like TestDriven.NET, provides several context menu options for executing your unit tests. There is a new pane that lists the unit tests, shows their current progress, and shows the results of the tests.



UNIT TEST STRUCTURE

The overall structure of a unit test is extremely simple. The first part of a test performs an action, and the second part of the test checks the result in some way. The action performed can be anything; a simple call to a static method, creating an object and changing its state, etc.

To verify the results of the action being performed meets our expectations, "*Assert*" style verifications are used. Unit testing frameworks include a wide variety of assertion checks that make verifying our expectations easy:

MS Test	NUnit	NUnit - new syntax	Purpose
Assert.AreEqual(x,y)	Assert.AreEqual(x,y)	Assert.That(x, Is.EqualTo(y))	Verify that x and y are equivalent
Assert.AreNotEqual(x,y)	Assert.AreNotEqual(x,y)	Assert.That(x, Is.Not.EqualTo(y))	Verify that x and y are not equivalent
Assert.AreSame(x,y)	Assert.AreSame(x,y)	Assert.That(x, Is.SameAs(y))	Verify that x and y are the same object
Assert.AreNotSame(x,y)	Assert.AreNotSame(x,y)	Assert.That(x, Is.Not.SameAs(y))	Verify that x and y are not the same object
Assert.IsTrue(x)	Assert.IsTrue(x)	Assert.That(x, Is.True)	Verify that x is true
Assert.IsFalse(x)	Assert.IsFalse(x)	Assert.That(x, Is.False)	Verify that x is false
Assert.IsNull(x)	Assert.IsNull(x)	Assert.That(x, Is.Null)	Verify that x is NULL
Assert.IsNotNull(x)	Assert.IsNotNull(x)	Assert.That(x, Is.Not.Null)	Verify that x is not NULL

OTHER TEST ATTRIBUTES

Some of the more advanced unit testing frameworks provide additional attributes that can be used to simplify tests and provide more functionality.

Attribute Name	Purpose
Explcit	Causes a test to be ignored unless it is explicitly chosen to run.
Ignore	Used to temporarily skip the running of a test.
Values	Provides a list of input parameters that will be combined to create a set of data to execute the test against.

COMMON UNIT TESTING STRATEGIES

There are various strategies for unit testing with your application. Unit testing often starts as a supplementary practice, only being used when it is convenient. Once a developer's skill progresses, it becomes beneficial to define a unit testing strategy. Guiding principles will help you write tests that offer the highest return on investment, and will ensure that unit testing is an efficient part of your overall development process.

TESTS SHOULD BE INDEPENDENT, ISOLATED, AND FAST

There are a few rules that must be followed for a unit test to provide value:

Tests should be independent - The set of tests you run, as well as their order should not affect the outcome of the unit tests.

Tests should be isolated - Tests should be focused on testing the smallest unit of functionality possible. They should not cross boundaries such as talking to the database, the file system, etc. External dependencies should be replaced with fake/mock versions so that the test result can be accurately gauged. Isolation also ensures that another developer can run the same test and get the same result. In order to ensure isolation, each test should only test one thing, and should have only one reason to fail.

Tests should be fast - If your unit tests are slow, they are less likely to be run. Slow unit tests can also slow down a continuous integration process, delaying the crucial feedback to the development team. The immediate feedback of a good unit test is invaluable.

WHAT IS REFACTORING?

In order to understand testing strategies it is necessary to first understand refactoring which is defined, by Wikipedia (http://en.wikipedia.org/wiki/Code_refactoring) as "the process of changing a computer program's internal structure without modifying its external functional behavior or existing functionality. This is usually done to improve external or internal non-functional properties of the software, such as code readability, simplify code structure, change code to adhere to a given programming paradigm, improve maintainability, improve extensibility, or increase execution performance."

Simplifying the code structure and its readability are obviously related. One of the primary goals of refactoring is to make your code as blatantly obvious as possible. One of the most common ways of doing this is extracting complex code into other, well-named methods. When you do this, your code should read like a book, and may even be somewhat readable by non-developers.

Example (before):

```
public void ShowLoginGreeting()
{
	Console.WriteLine("Welcome " + GetUserName());
	Console.WriteLine("Running Processes:")

	foreach(var process in Process.GetProcesses())
	{
		Console.WriteLine("Process Name: " + process.Name);
	}
	Console.WriteLine("Last Login: " + GetLastLogin();
}
```

Example (after):

```
public void ShowLoginGreeting()
{
	ShowWelcomeMessage();
	ShowRunningProcesses();
	ShowLastLogin();
}

private void ShowWelcomeMessage()
{
	Console.WriteLine("Welcome " + GetUserName());
}

private void ShowRunningProcesses()
{
	Console.WriteLine("Running Processes:")
	foreach(var process in Process.GetProcesses())
	{
		Console.WriteLine("Process Name: " + process.Name);
	}
}

private void ShowLastLogin()
{
	Console.WriteLine("Last Login: " + GetLastLogin());
}
```

Notice that refactoring does not necessarily shorten your code. However, it should make individual sections of your code easier to read and maintain.

TEST DRIVEN DEVELOPMENT

One of the simplest and most talked about strategies is known as **Test Driven Development** or **TDD**. It's comprised of only three steps: **red**, **green**, **refactor**.

RED

The first phase of TDD actually doesn't involve writing any production code. You write a test that defines a particular piece of functionality you're going to implement. The purpose is to verify that your test does indeed fail when the test condition is not met. This simple step arguably has value, and is very controversial. The problem is

that if your actual code does not exist, the test won't even compile. You may end up spending time on adding "non-functionality". It may not be much, but it could add up over time.

GREEN

This is the step where we're really going to add value to our application. In this step, we write just enough code to cause the unit test to succeed in the easiest possible manner. We then run the test to verify that the test does in fact pass. Your code ends up evolving right before your eyes, gaining new functionality each step of the way.

REFACTOR

After each test is added, you need to take a good look at the code and make sure that it's easy to read, easy to maintain, and is quite simply the best way to get the job done. Refactoring in small pieces makes the refactoring process much more manageable.

EVOLVING CODE

When adding code for the purpose of causing a test to succeed, it's important to only solve the immediate problem. Since you're adding features one at a time, you may start to see that the evolution of the code solves the problem in a more elegant way than you had originally anticipated. More importantly, adding features one by one minimizes the number of features that you need to think about at any given time. Unfortunately we're all human, so **we need to take advantage of any chance we get to simplify the problem at hand**.

WHEN SHOULD YOU WRITE UNIT TESTS?

Simplifying development is obviously a huge benefit of Test Driven Development. Another benefit is that we're verifying our features at the lowest level possible. **You have the deepest understanding of your code and its various code paths when you're writing it**. This is an easy time to promise to write tests in the future, only to never follow through. If you put off unit testing until after you've written your code, you may find that you can no longer identify knowledge of edge cases that are contained in your code. Your tests may appear sound, but may not be as logically exhaustive as you would like them to be, leaving gaps.

TESTING IS FOR FUNCTIONALITY, NOT CODE!

When writing unit tests, it's easy to write tests for logic paths that should never get executed. Judgment must be used to determine how important it is for a particular method to handle edge cases.

One of the misconceptions of developers is that they're simply trying to exercise their code. **Unit tests are verifications for what you believe your code should do**. This is one of the reasons that we refactor. Among other things, refactoring is our chance to remove code paths that are not used. If you're writing your unit tests to test the code you've written, you might actually be wasting time verifying code paths that will never actually get executed.

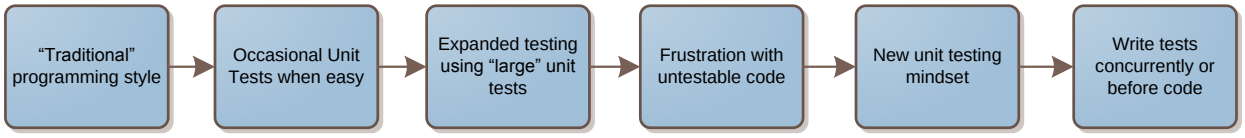
THE CONSTRAINTS OF REALITY

While the strategies of unit testing seem strict, the rules have a clear purpose. As a responsible developer, you must be able to evaluate the situation, and adjust the strategies to fit the constraints of the project. For example, if you're prototyping a new feature that has a lot of unknowns, it may require that you try a number of radically different solutions. In this case, writing unit tests just to throw them away defeats the purpose.

DESIGNING FOR TESTABILITY

The next logical step in the unit testing evolution is a stage where good design and testability converge. It's the place you arrive at when you start to truly decouple your code, and learn that the goal of testability has other advantages.

There is an observable natural progression that developers tend to follow when they're first introduced to the concept of unit testing. They keep on writing code the same as they always have, but quickly realize the limited usefulness of this type of process.



The first obstacle of testing an existing system is that components are tightly coupled. In order to set up the initial state for a test, you'll often run into some barriers:

- A high quantity of classes that need to be created
- Dependencies on external systems
- Complicated logic to wire up classes
- A separate "test" mode that gets used during testing

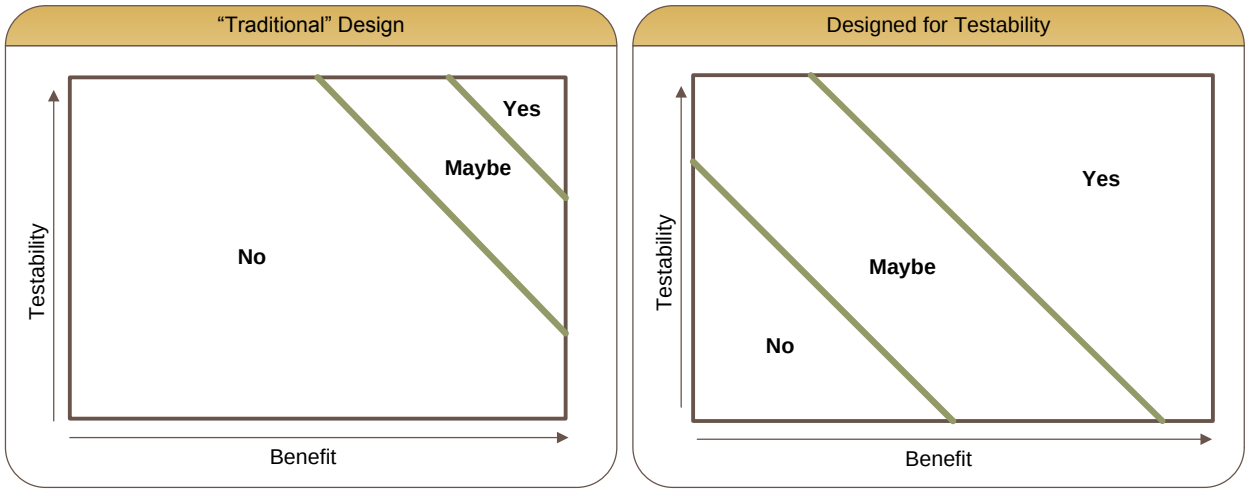
To solve these problems, we start to create clear separations between different classes or groups of classes. At this stage, the costs may often outweigh the benefits of testability. We're gaining a small amount of testability at the cost of a lot of work refactoring the code.

Once a significant suite of tests starts to appear, a limitation is reached again. At this point, it becomes clear that we can continue to use a "divide and conquer" strategy to again decouple our classes.

Single Responsibility Principle

The ideal strategy for building our applications in a modular, and therefore testable manner is the single responsibility principle. The single responsibility principle states that every object should have a single responsibility, and should only have a single reason to change. This is a powerful principle that helps create code that is easier to write and maintain.

Test Driven Development helps drive our design to the point where the majority of the code is testable at a low cost. In the figure below, you'll see a representation of a traditional application design. There is a large region of code that has a low return on investment for unit testing, and therefore is most likely not worth it. When we design for testability, we're minimizing the amount of code that has a low return. The Y-Axis is the degree of testability, or the ease of which a particular piece of logic can be tested. The X-Axis is defined as "benefit", which is comprised of how often the code will be reused, the importance of the code, the complexity of the code, and the risk level associated with the code not working as expected.

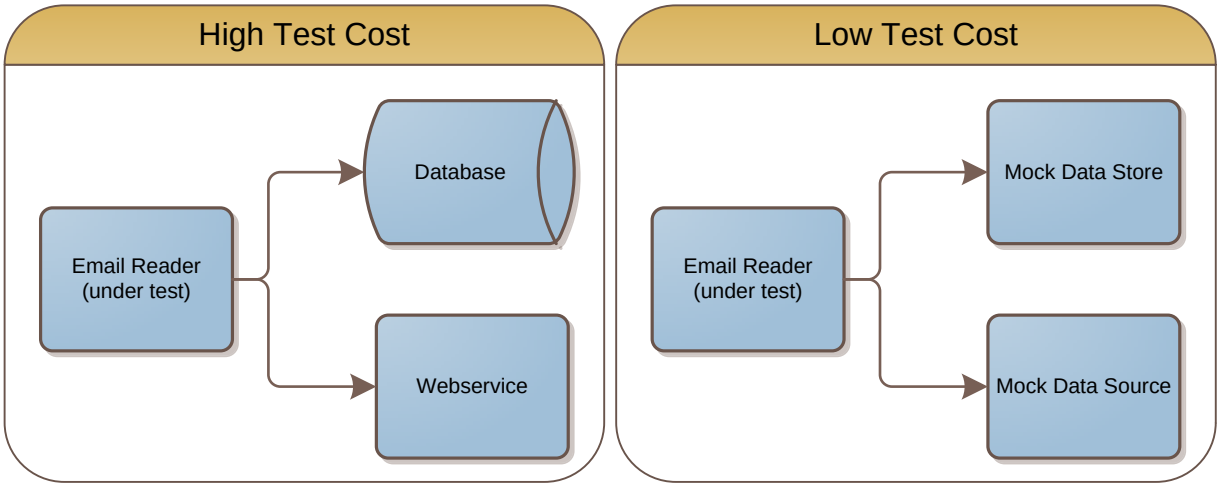


OBJECT MOCKING

Low-level classes that contain isolated logic such as math, collection processing, or algorithms tend to be easy to test. Their interface consists of basic types, and the classes themselves don't have any significant external dependencies. Unfortunately, most code in an application is not so testable by default. For example, a class that uses a database or connects to a web service is particularly difficult to test.

As I mentioned before, your unit tests should be isolated. That means that unit tests should typically not cross boundaries into other systems. Ideally, one level of unit tests should not be testing any functionality outside of the class being tested. Functionality from classes outside of your test isolation can cause overlap in the code being executed, and even worse, can add extra test code that will increase the friction of the testing process.

Let's examine a simple scenario. Imagine an email reader class that calls a web service to retrieve a list of email addresses, and then inserts them into a database. If you were to code this in a traditional fashion, the email reader class would create a new instance of the database connection, and a new instance of the class to query the web service. Testing the class would be nearly impossible. If you do end up testing it, the result is an integration test, since it crosses application boundaries. That can make it slow and means that our test is not isolated. We also run the risk of not leaving the database like we found it.



It's clear that we need to isolate the class being tested, and there are a number of tools that allow us to do this elegantly.

INTERFACES - QUICK OVERVIEW

Interfaces are incredibly important when it comes to being able to separate our code into logical pieces. An interface allows us to define "what" can be used, and not "how" it is being used. A simple example from the .NET framework is the IComparable interface.

```
public static string GetComparisonText(IComparable a, IComparable b)
{
	if (a.CompareTo(b) > 0)
		return "a is bigger";
	if (a.CompareTo(b) < 0)
		return "b is bigger";

	return "same";
}
```

Notice that since we're using the IComparable interface, our method works with any type of data that can be compared. It only cares that "a" provides a way of comparing itself to another value. This method will accept integers, strings, or even new types that a developer creates in the future.

Classes can implement multiple interfaces, and interfaces can even inherit from other interfaces. The .NET framework generics can be applied to interfaces to make them even more flexible and powerful. For example, there is a generic version of IComparable. Applying the generic IComparable<T> interface to our comparison method would force the parameters to be of the same type, yet still work with any IComparable type. Our method signature would then look like this:

```
public static string GetComparisonTextGeneric<T>(IComparable<T> a, T b)
```

If we now apply interfaces to our email reader example, we can effectively isolate the class we're trying to test. We simply create an IDatabase interface, and an IWebservice interface. Within those interfaces, we define the specific functionality that we need to get our job done. The interface may also concisely describe its purpose. For example, our interfaces would probably look like this:

```
public interface IEmailSource
{
	IEnumerable<string> GetEmailAddresses();
}

public interface IEmailDataStore
{
	void SaveEmailAddresses(IEnumerable<string> emailAddresses);
}
```

Programming the class against the interfaces is elegant for number of reasons. First, as I mentioned, we're defining the "what" for retrieving the email addresses. Second, we're actually able to build our class without having a database or a web service in place. This is particularly useful when you're developing in a team where other pieces are being written concurrently. Finally, since we only care about the "how", we can now use mocks that will provide the functionality we need without worrying about anything we're not interested in right now.

If you wanted to manually mock the email source and the email data store, we would need to set the up so that we could define their behavior, and then have them execute that behavior when the methods are called. A first pass would look something like this:

```
public class MockEmailSource : IEmailSource
{
	public IEnumerable<string> EmailAddressesToReturn { get; set; }

	#region IEmailSource Implementation

	public IEnumerable<string> GetEmailAddresses()
	{
		return EmailAddressesToReturn;
	}

	#endregion
}

public class MockEmailDataStore : IEmailDataStore
{
	public IEnumerable<string> SavedEmailAddresses { get; set; }

	#region IEmailDataStore Implementation

	public void SaveEmailAddresses(IEnumerable<string> emailAddresses)
	{
		SavedEmailAddresses = emailAddresses;
	}

	#endregion
}
```

In this case, the mocks look pretty manageable. The problem is, they'll quickly get complicated when we want to add additional behaviors. For example, what if we want to simulate an exception that is raised when reading or saving an email address? We would end up adding another field to store the exception to raise, and then our mock object would raise that exception if it has been supplied. Because of this complexity, manual mocking is typically avoided when possible. Instead, a mocking framework is used.

USING A MOCKING FRAMEWORK

There are a number of mocking frameworks available:

- NMock - http://www.nmock.org/
- moq - http://code.google.com/p/moq/
- Rhino Mocks - http://ayende.com/projects/rhino-mocks.aspx
- TypeMock (not free) - http://www.typemock.com/

A good mocking framework will create mock objects for us. Also, each mock object we create can be configured to behave however we want. This lets us test any scenario we like. Let's take a look at how we can test our EmailReader class using Rhino Mocks. I simply need a reference to the single Rhino.Mocks.dll library, and we're ready to write our test:

TODO: A couple comments on this test method. First, it is making multiple assertions, which we should try to avoid and at least mention. Secondly, consider using the mockSource.AssertWasCalled method over Expect/Verify. This helps solidify the AAA methodology and is also less code.

```
[TestMethod]
public void Read_Email_From_Source_And_Save_To_Destination()
{
	//Create our mocks
	IEmailSource mockSource = MockRepository.GenerateMock<IEmailSource>();
	IEmailDataStore mockDataStore =
MockRepository.GenerateMock<IEmailDataStore>();

	//Set up the fake data that we'll be passing through
	var emailAddresses = new List<string>();

	//Define the mock behavior
	mockSource.Expect(x => x.GetEmailAddresses()).Return(emailAddresses);
	mockDataStore.Expect(x => x.SaveEmailAddresses(emailAddresses));

	var emailReader = new EmailReader(mockSource, mockDataStore);
	emailReader.DoWork();

	//Make sure our expectations were satisfied
	mockSource.VerifyAllExpectations();
	mockDataStore.VerifyAllExpectations();
}
```

The syntax may seem a little strange at first, because it's taking advantage of Lambdas as custom, in-line methods. This allows us to define the expected behavior of our mock objects in a syntax that looks nearly identical to how we would call the method directly. Rhino Mocks also gives us a fluent interface that allows you to configure additional behaviors (such as a return value) at the end of your expectation.

The syntax being used is called Arrange, Act, Assert (AAA). In the **arrange** section, you're defining your mocks. In the **act** section, you're defining the behaviors you're expecting and defining. in the **assert** section, we've verifying that our expected behaviors actually occurred.

The previous example allowed us to avoid writing our own implementation of IEmailSource and IEmailDataStore. We now have extra functionality that allows us to test some additional scenarios. Let's say we want to verify that the email reader handles exceptions when reading email addresses. The test is very simple:

```
[TestMethod]
public void Exception_Thrown_From_Email_Source_Verify_Exception_Suppressed()
{
	//Create our mocks
	IEmailSource mockSource = MockRepository.GenerateMock<IEmailSource>();
	IEmailDataStore mockDataStore =
MockRepository.GenerateMock<IEmailDataStore>();

	//Throw an exception when reading the addresses
	mockSource.Expect(x => x.GetEmailAddresses()).Throw(new Exception("Boom!"));

	var emailReader = new EmailReader(mockSource, mockDataStore);
	emailReader.DoWork();

	//Make sure our expectations were satisfied
	mockSource.VerifyAllExpectations();
	mockDataStore.VerifyAllExpectations();
}
```

Even the most basic features in mocking frameworks give you an amazing amount of power. A good mocking framework will also let you raise events, define custom or complex behaviors, and will let you configure the number of times a particular method is called.

STUBS

At times, a distinction is made between Mocks and Stubs. Mocks are configured by setting up expectations, and defining the behaviors in certain situations. A mock will verify that the expectations have actually been satisfied. A Stub however, is an object that can still have configurable behaviors, but stubs do not have expectations to verify.

For example, if we define a mock object and tell it that method "DoStuff" will be called, it MUST be called for the test to pass. If the object is a stub, it's just there to stand in and provide functionality when needed. In short, a stub will never cause a test to fail.

MOCKING THE UNMOCKABLE

There are often times when you want to test a class that has dependencies that cannot be mocked. For example, a class that reads information from the ASP.NET session. For this situations, there are a couple of solutions:

- refactor and Create adapter layer and use an interface
- Mock the dependencies that your dependency uses

^^

TODO: Mocking dependencies of framework elements (email server, SQLite DB server)

THE TEST DRIVEN DESIGN PARADOX

If you get to the point where your tests are driving your design and as a specification, your code starts to become easy to develop and maintain. The irony is that once these positive development habits are formed, the tests start to have a very high likelihood of passing the first time they are run. This may seem to negate the need to write

tests while creating new code or making changes to existing code, but the other advantages such as long-term maintainability and refactoring ability should not be forgotten.

TESTING UNDER PRESSURE

Even with a strict unit testing philosophy, there is always the concern that towards the end of a project, unit testing will dropped with the goal of removing the perceived overhead associated with it. There are a couple of problems with doing so:

- Long-term stability is traded for a small potential in short term productivity - If you have the luxury of never touching the product again, this may be acceptable to you. Usually, you're acquiring technical debt that will only hurt the product in subsequent releases, possibly snowballing to the point of being out of control.
- If unit testing is causing significant overhead, this may indicate a problem with the process. This purpose of this document has been to minimize or eliminate the overhead. Realize that unit testing is simply the way that you write software, and live with that choice.
- Adding code quickly just increases the need to have tests in place. This is the worst time to skip them. The new code or fixes will probably just introduce new bugs which will make the problems much larger than originally estimated.

EXTRACTING DUPLICATE LOGIC

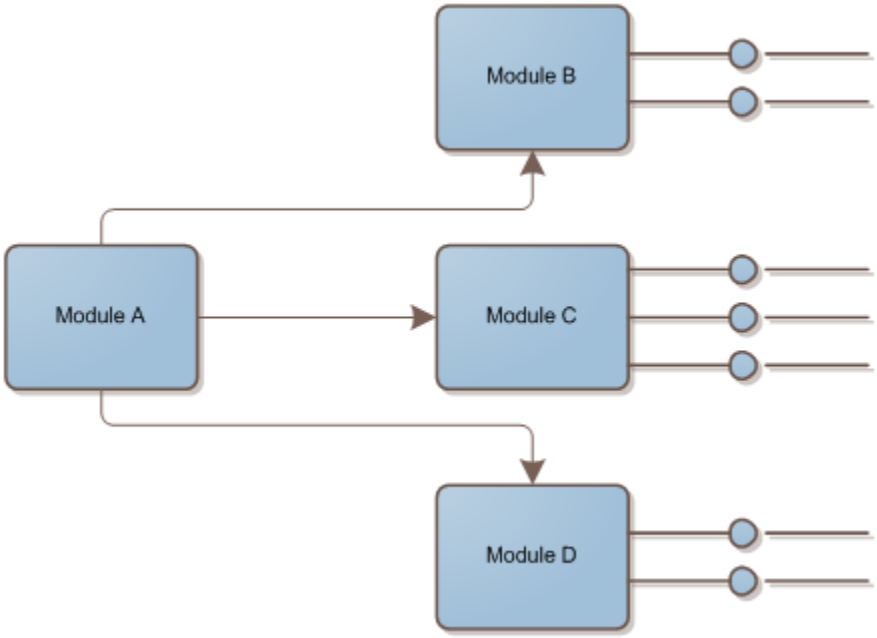
TODO: helps minimize code and increase testability.

MODULAR DESIGN BENEFITS

TODO: A modular design means you'll spend more time working on the important logic, and less time hacking your code.

ADVANCED TECHNIQUES (TODO)

AUTOMATIC TEST CASE REDUCTION



Test Coverage Analysis

Refactoring unit tests to take advantage of the testing framework constructs - for example, TestInitialize, and even test tables.

Calculating ROI based on work needed vs benefit

RESOURCES

Hanselminutes

SOLID Principles - http://www.hanselminutes.com/default.aspx?showID=163

SOLID Principles #2 - http://www.hanselminutes.com/default.aspx?showID=168

Test Driven Development - http://www.hanselminutes.com/default.aspx?showID=164

Unit Test Boundaries - http://haacked.com/archive/2008/07/22/unit-test-boundaries.aspx

Unit Tests and Developers under Pressure: http://www.elilopian.com/2008/12/04/unit-tests-and-developers-under-pressure/

Other TODO

Talk about dependency injection?

Add benefits:

- *Safety net for multiple developers*
- *consistency*
- *confidence*