Chapter # 5 Testing

Testing is a very important practice and must be done before any real code is written thus proporting the so called Test driven development(TDD). Unit testing and integration testing are integral to determing the quality of applications.

## **Testing in Visual Studio 2012**

Let’s see how testing is supported by Visual Studio 2012 and its associated products, Visual Studio Team Foundation Server (TFS) and Microsoft Test Manager (MTM).

If your specialty is testing the whole application, you’ll be more interested in the support provided by MTM. If you’re a developer, you might take more interest in the support for automated testing in Visual Studio 2012. However, continuous development demands a closer relationship between these two roles, and some teams dispense with the distinction altogether. Therefore, the Visual Studio 2012 tools are designed to integrate the different styles of testing, and they support a broad spectrum of testing practices, from the more traditional approaches through continuous development.

## **Automated Testing with Visual Studio 2012**

Automated testing includes all types of tests that are defined by writing or generating program code. You create automated tests in Visual Studio 2012, where you initially run them for debugging.

When the test—and the application code that it tests—are correct, you check in the test, along with the application code. From the source code repository, it’s picked up by the build service and run regularly according to your team’s build definitions.

**Unit and Integration Tests** Unit testing is one of the most effective ways of maintaining a bug-free codebase through successive changes in an application.

A unit test is a method that tests a method, class or larger component of your application in isolation from other parts, external systems and resources. In practice, developers often write integration tests—that is, tests written in a similar way to unit tests but which might depend on external databases, Web sites or other resources. Either way, these tests use the same tools and infrastructure.

In Visual Studio 2012, you can write tests that use any of several test frameworks, such as NUnit, xUnit and the default VSTest. When you have coded tests in any of these frameworks, you simply open the Test Explorer window and choose Run All. The test results are summarized in the window.

Background testing is an option that efficiently runs your tests in the background every time you build your solution. The tests affected by your changes are performed first. This means that as you work, you can constantly see which tests are passing or failing.

**Isolate Units by Using Fakes** True unit testing means disconnecting the unit under test from the code on which it’s dependent. This has a number of advantages. If your unit is being developed or updated at the same time as other units on which it’s dependent, you can test it without waiting for the others to be complete. If you restructure the application to use this unit in a different way, or in a different application, the tests go with it and don’t need to change.

Visual Studio 2012 provides two mechanisms, collectively called fakes, for disconnecting a unit from its dependencies. Calls from your unit to methods outside its boundary can be handled by small pieces of code that you provide. For example, you can define a shim that intercepts calls to any external method such as DateTime.Now. Because it always receives the same response from the shim, your unit will demonstrate the same behavior every time it’s invoked. You can also define stubs, which provide placeholder implementations of methods in assemblies that haven’t been loaded.

**Performance and Load Tests** Visual Studio 2012 Ultimate provides specific test facilities for performance and stress testing. An application can be instrumented and driven so as to measure its performance under specified loads. Web applications can be driven with multiple requests, simulating many users.

**Coded UI Tests** Coded UI tests let you run your application and generate code that drives its UI. Visual Studio 2012 includes specialized tools for creating and editing coded UI tests, and you can also edit and add to the code yourself. For example, you might create a simple procedure to buy something at a Web site and then edit the code to add a loop that buys many items.

Coded UI tests are particularly useful where there’s validation or other logic in the UI—in a Web page, for example. You can use them either as unit tests for the UI or as integration tests for the whole application.

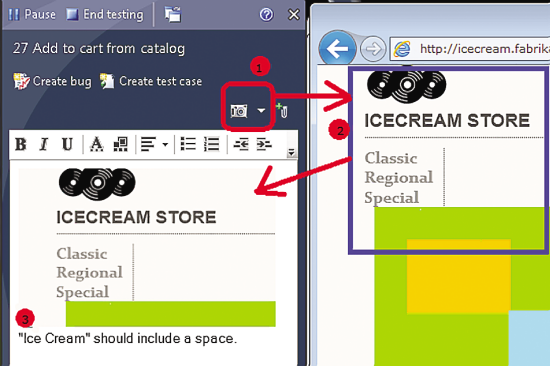
## **Manual Testing with Microsoft Test Manager**

Manual testing can be planned or exploratory. You perform manual tests with the help of MTM. Tests are normally performed on versions of your application that have been built from checked-in code.

Typically, manual tests are linked to user stories (or product backlog items or other requirements), and the results of the tests are displayed in reports on the project dashboard. This means that every­one can quickly see which stories have been successfully implemented.

**Exploratory Testing** Exploratory testing simply means running the application to try it out. However, why do you need MTM to help you run it?

MTM can record your actions, comments and screenshots while you work. If you decide to create a bug report, all this information is automatically added to it, making it unnecessary for you to add a precise description of how to reproduce the bug. **Figure 3** shows an example of the MTM exploratory testing window alongside a Web application that’s being tested.



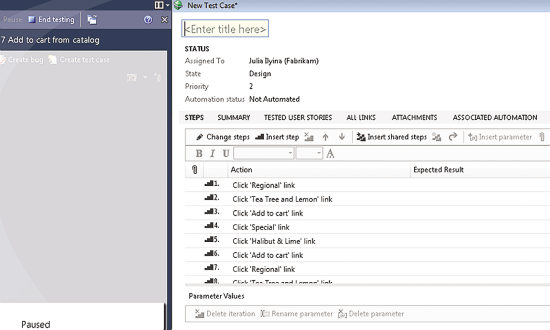
**Figure 3 Recording a Screenshot and Making Notes in the Exploratory Testing Window**

MTM can also instrument the application itself, both in the client and the server, and record event data that can be used to debug the application. This data is automatically attached to your bug report.

When the bug is fixed, you’ll want to repeat the steps you took in the exploration to verify the fix. To help with this, you can generate a test case from the exploratory session, in which you include the relevant steps.

**Planned Testing with Test Cases**

Test cases are manual tests that you define as a series of steps the tester should perform. **Figure 4** shows the steps defined in a test case.



**Figure 4 Defining Steps and Expected Results in a Test Case**

Test cases provide a great way to clarify what the users need. At the start of the sprint, when you’re discussing stories or requirements with the users and other stakeholders, you can use the steps as a precise example of what the users will be able to do by the end of the sprint. Each test case is just one instance of the requirement, and so each requirement is usually associated with more than one test case. For example, if the requirement is to be able to buy ice cream, one test case will detail the steps to buy a particular flavor. You would create another test case to describe buying a mixture of flavors. The guiding principle of the discussion with stakeholders should be: “When you can successfully perform these test cases, then we’ll consider the story to be implemented.”

In TFS, both the stories or requirements and the test cases are represented by work items. You can link them together so the progress of a requirement can be tracked by the results of the tests.

When you run a test case, the steps are displayed at the side of the screen. You check off each step while you run the application. At the end, you check off whether the test has passed or failed.

Just as with exploratory testing, your actions, comments, screenshots and application data are recorded so you can create a detailed bug report very quickly.

A great advantage of using steps is that they help anyone repeat the test reliably, even if they aren’t familiar with the application. When a test is repeated, you can be confident that whether it passes or fails, it isn’t simply because the test was run differently from the last time.

You can also generate a planned test case from an exploratory session. Doing this helps ensure that you always run the test using the same actions.

## **Automating Manual Tests**

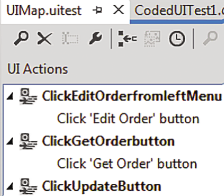
Automating a substantial portion of your manual tests is essential to minimize the time taken by testing in the DevOps cycle. Visual Studio 2012 supports this automation in several ways.

**Record/Playback** You can rerun a test case semiautomatically. On the second and subsequent times you run a test, MTM replays the keystrokes and gestures that you used on the first run. All you have to do is verify that the results you see conform to the expectations detailed in the steps.

Playback makes manual testing quicker and more reliable. It also makes it possible to distribute the testing load among colleagues who might not be completely familiar with the application.

Even if you don’t fully automate a test, rapid and reliable playback helps reduce the DevOps cycle time. This feature does not require Visual Studio 2012 to be installed and does not involve writing code.

**Generating Coded UI Tests** You can generate a fully automated coded UI test from a recorded manual test case run. The generated code performs the same actions as the manual test. By using a special editor in Visual Studio 2012, you can also extend the test to verify the results and generalize it to repeat the test for different input data.**Figure 5** shows the special editor.



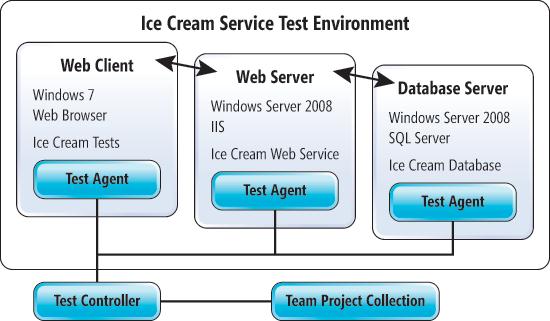
**Figure 5 Editing UI Actions in Visual Studio 2012**

**Linking Test Cases to Test Methods** You can link a test case to any test method, even if it hasn’t been generated from your test run. The result of running the test will be reported as if you had run the manual steps. Typically you would link the test case to an integration test that performs the same actions as the manual test but drives the business logic directly, rather than using the UI.

This approach has the benefit that changes in the UI layout don’t invalidate the test. It’s also useful when the development team has already created a suitable integration test.

## **Lab Management**

When you test an application, the first thing you need is a machine to run it on. In fact, most applications today need several machines. For a realistic test environment, you might, for example, need to install a Web server, a database server and a client browser all on separate computers.**Figure 6** illustrates such an environment.



**Figure 6 A Sample Lab Environment for Testing a Sales Web Site**

In addition to the basic installation, you’ll also want to install agents that can collect the event data that was mentioned earlier in the “Exploratory Testing” section.

In MTM a feature named Lab Center makes all of this straightforward. Lab Center lets you define lab environments. A lab environment is a set of machines that will be used as a group for test purposes.

In addition to handling the assignment of the machines (so you don’t accidentally use a machine that’s running someone else’s tests), Lab Center also installs the necessary test agents. Lab Center provides a console where you can quickly log in to any of the machines in an environment.

Lab Center is also good at creating and managing virtual machines (VMs). You can create a virtual environment, install the relevant platform software and then store a library copy of it that you can use whenever you want to test your application. All you have to do is reinstantiate a clean copy of the environment and install the new versions of the application’s components. You can also automate this deployment process.

Using Lab Center—and, in particular, taking advantage of its facility with VMs—can significantly decrease lab setup time compared to more traditional approaches to maintaining a lab. Lab Center is a substantial contributor to reducing the time spent in each DevOps cycle.

## **Automated Testing on TFS**

Automated tests are initially performed in Visual Studio 2012 on a developer’s computer. After the code has been checked in to the source repository, there are a number of ways in which tests on the integrated code can be run by the build service.

**Periodic Builds** The build service compiles the code and runs the tests. You can create build definitions to specify which tests should be run, and you can specify when they should run. For example, you might run a core set of tests on a continuous basis and run a more extensive set every night.

Build results can be viewed in Visual Studio 2012 and are also available from your project’s TFS Web service. E-mails can notify you of failures.

**Lab Deployment** As we previously described, you can assign a group of lab machines to a test by using Lab Center. By defining a lab build, you can automate this process. When your build is triggered—for example, when code is checked in, or at a particular time of day—the build starts by compiling all the application and test code. If this is successful, a lab environment is assigned, and if it’s a virtual environment, it can be set back to a fresh state. Your application components are then deployed to the correct machines, and the tests are installed on the designated client machine from where they drive the application.

The tests can be automated tests of any kind, but typically you use this type of build to perform large integration tests or tests of the whole application.

If your tests are linked to test cases, the results will be recorded against the related user stories or requirements and displayed in the project progress reports.

## **Reports**

TFS provides a number of charts and tables that show the progress of a project. You can view them either individually or in project dashboards. Among the reports are several related to testing.

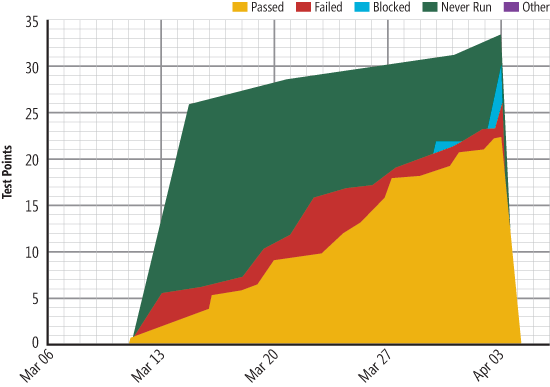
For example, the User Story Test Status report, illustrated in **Figure 7**, shows the list of stories you’re working on in the current sprint. In addition to the development work performed for each story, the chart shows the success or failure of its associated tests. If failures were discovered while running the tests, the resulting bug reports are also linked to the requirements.

**Figure 7 User Story Test Status Report**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Work Progress | Test Status | | | | | | | | | | | | | | | | |
| Title | % Hours Completed |  | Hours  Remaining | Test  Points | Test Results | Bugs | | | | | | | | | | | | |
| Customers can buy ice cream. | 60% |  |  |  |  | 16 | 3 | 67% | 33% | 3 | | | | | | | | |
| Customers can select flavor from catalog. | 60% |  |  |  |  | 10 | 3 | 67% | 33% |  | | | | | | | | |
| Vendor can vary flavor catalog. | 25% |  |  |  |  |  |  |  |  | 15 | 2 | 100% |  | | | | | |
| Vendor can set different prices for flavors. | 39% |  |  |  |  |  |  | 23 | 2 | 100% | 4 | | | | | | | |
| Customers can set favorite flavors. | 100% | 0 | 1 | 100% | 4 | | | | | | | | | | | | | |
| Users can choose U.K. spelling of favourite flavour. | 100% | 0 | 6 | 50% | 33% |  |  |  | | | | | | | | | | |
| Customers can select different types of cones. | 75% |  |  |  | 5 | 22 | 63% | 26% |  |  | 2 | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

The results on the chart come from both the most recently run manual tests and the automated test runs.

The Test Plan Progress report, illustrated in **Figure 8**, shows how many test cases were created for the current sprint and how many have been run.



**Figure 8 Test Plan Progress Report for a Sprint**

Some teams like to create test cases at the start of each sprint, as a target for the team. All the tests should be green at the end of the sprint.

## **Visual Studio 2012 and Continuous Development**

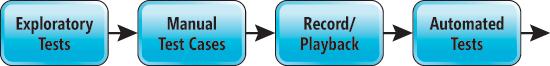
You now have some familiarity with the range of the Visual Studio 2012 testing features, from unit tests to whole-application manual tests.

The DevOps cycle views development as just one half of a process that also incorporates feedback from operations. Depending on the context, the DevOps cycle will be short or long. If you’re developing a nuclear power station, hopefully you’ll go around the loop very slowly. If you’re running a Web application, you might go around it every few days. Slow and fast cycles are equally valid and appropriate for different types of systems. Both incorporate the need for testing, to different degrees. If the fast cycle of continuous development is appropriate for your project, it’s important to reduce the time it takes to perform every action in the loop. You’ll probably also make less distinction between the roles of developer and tester than in projects that work at a more measured pace.

The tools available in Visual Studio 2012 can substantially reduce the amount of time it takes to test your application. Here are some points to remember:

* Clean lab environments can be set up quickly and automatically, especially by using VMs. Take advantage of the Lab Center.
* Recording actions during exploratory and planned testing lets you create bug reports quickly and reliably, reducing the chance that a bug will disappear when someone tries to reproduce it.
* You can implement a gradual transition from manual tests to automated tests. The automated tests handle the bulk of regression testing while manual tests focus on new and updated stories.
  + From exploratory tests, you can generate repeatable manual test cases.
  + You can record test runs and play them back rapidly and reliably.
  + You can generate coded UI tests from test runs.Alternatively, you can link separately coded integration tests to test cases.

**Figure 9**shows the progression from exploratory tests to automated tests.



**Figure 9 Test Progression**

* Test cases can help you describe your user stories precisely. You can work with your stakeholders to write the steps to reduce any misunderstandings about what the stories do.
* Tests are linked to requirements (or user stories or product backlog items) so you can see comprehensive reports on how far the users’ needs have been implemented—not simply in terms of the work done on them, but in terms of whether they’re successful. The tests provide the goal for the development team in each sprint.
* Test cases and requirements can be linked to storyboards in PowerPoint, requirements documents or Unified Modeling Language models. When you make changes in the storyboard, you can trace the necessary changes to the tests.
* A test plan links together test cases, code, requirements, test environments and test settings, and the team project. If the team spends a few months working on something else before returning to enhance a product, it’s easy to reconstruct everything required for testing.

Test Driven Development

## Introduction

Although developers have been unit testing their code for years, it was typically performed after the code was designed and written. As a great number of developers can attest, writing tests after the fact is difficult to do and often gets omitted when time runs out. Test-driven development (TDD) attempts to resolve this problem and produce higher quality, well-tested code by putting the cart before the horse and writing the tests before we write the code. One of the core practices of Extreme Programming (XP), TDD is acquiring a strong following in the Java community, but very little has been written about doing it in .NET.

## What Are Unit Tests?

According to Ron Jeffries, Unit Tests are "programs written to run in batches and test classes. Each typically sends a class a fixed message and verifies it returns the predicted answer." In practical terms this means that you write programs that test the public interfaces of all of the classes in your application. This is not requirements testing or acceptance testing. Rather it is testing to ensure the methods you write are doing what you expect them to do. This can be very challenging to do well. First of all, you have to decide what tools you will use to build your tests. In the past we had large testing engines with complicated scripting languages that were great for dedicated QA teams, but weren't very good for unit testing. What journeyman programmers need is a toolkit that lets them develop tests using the same language and IDE that they are using to develop the application. Most modern Unit Testing frameworks are derived from the framework created by Kent Beck for the first XP project, the Chrysler C3 Project. It was written in Smalltalk and still exists today, although it has gone through many revisions. Later, Kent and Erich Gamma (of Patterns fame) ported it to Java and called it jUnit. Since then, it has been ported to many different languages, including C++, VB, Python, Perl and more.

## The NUnit Testing Framwork

NUnit 2.0 is a radical departure from its ancestors. Those systems provided base classes from which you derived your test classes. There simply was no other way to do it. Unfortunately, they also imposed certain restrictions on the development of test code because many languages (like Java and C#) only allow single inheritance. This meant that refactoring test code was difficult without introducing complicated inheritance hierarchies. .NET introduced a new concept to programming that solves this problem: attributes. Attributes allow you to add metadata to your code. They typically don't affect the running code itself, but instead provide extra information about the code you write. Attributes are most often used to document your code, but they can also be used to provide information about a .NET assembly to a program that has never seen the assembly before. This is exactly how NUnit 2.0 works. The Test Runner application scans your compiled code looking for attributes that tell it which classes and methods are tests. It then uses reflection to execute those methods. You don't have to derive your test classes from a common base class. You just have to use the right attributes. NUNit provides a variety of attributes that you use when creating unit tests. They are used to define test fixtures, test methods, setup and teardown methods. There are also attributes for indicating expected exceptions or to cause a test to be skipped.

### TestFixture Attribute

The TestFixture attribute is used to indicate that a class contains test methods. When you attach this attribute to a class in your project, the Test Runner application will scan it for test methods. The following code illustrates the usage of this attribute. (All of the code in this article is in C#, but NUnit will work with any .NET language, including VB.NET. See the NUnit documentation for additional information.)

namespace UnitTestingExamples

{

using System;

using NUnit.Framework;

[TestFixture]

public class SomeTests

{

}

}

The only restrictions on classes that use the TestFixture attribute are that they must have a public default constructor (or no constructor which is the same thing).

### Test Attribute

The Test attribute is used to indicate that a method within a test fixture should be run by the Test Runner application. The method must be public, return void, and take no parameters or it will not be shown in the Test Runner GUI and will not be run when the Test Fixture is run. The following code illustrates the use of this attribute:

namespace UnitTestingExamples

{

using System;

using NUnit.Framework;

[TestFixture]

public class SomeTests

{

[Test]

public void TestOne()

{

*//* *Do something...*

}

}

}

### SetUp & Teardown Attributes

Sometimes when you are putting together Unit Tests, you have to do a number of things, before or after each test. You could create a private method and call it from each and every test method, or you could just use the Setup and Teardown attributes. These attributes indicate that a method should be executed before (SetUp) or after (Teardown) every test method in the Test Fixture. The most common use for these attributes is when you need to create dependent objects (e.g., database connections, etc.). This example shows the usage of these attributes:

namespace UnitTestingExamples

{

using System;

using NUnit.Framework;

[TestFixture]

public class SomeTests

{

private int \_someValue;

[SetUp]

public void Setup()

{

\_someValue = 5;

}

[TearDown]

public void TearDown()

{

\_someValue = 0;

}

[Test]

public void TestOne()

{

*//* *Do something...*

}

}

}

### ExpectedException Attribute

It is also not uncommon to have a situation where you actually want to ensure that an exception occurs. You could, of course, create a big try..catch statement to set a boolean, but that is a bit hack-ish. Instead, you should use theExpectedException attribute, as shown in the following example:

namespace UnitTestingExamples

{

using System;

using NUnit.Framework;

[TestFixture]

public class SomeTests

{

[Test]

[ExpectedException(typeof(InvalidOperationException))]

public void TestOne()

{

*//* *Do something that throws an InvalidOperationException*

}

}

}

When this code runs, the test will pass only if an exception of type InvalidOperationException is thrown. You can stack these attributes up if you need to expect more than one kinds of exception, but you probably should it when possible. A test should test only one thing. Also, be aware that this attribute is not aware of inheritance. In other words, if in the example above the code had thrown an exception that derived from InvalidOperationException, the test would have failed. You must be very explicit when you use this attribute.

### Ignore Attribute

You probably won't use this attribute very often, but when you need it, you'll be glad it's there. If you need to indicate that a test should not be run, use the Ignore attribute as follows:

namespace UnitTestingExamples

{

using System;

using NUnit.Framework;

[TestFixture]

public class SomeTests

{

[Test]

[Ignore("We're skipping this one for now.")]

public void TestOne()

{

*//* *Do something...*

}

}

}

If you feel the need to temporarily comment out a test, use this instead. It lets you keep the test in your arsenal and it will continually remind you in the test runner output.

### The NUnit Assertion Class

In addition to the attributes used to identify the tests in your code, NUnit also provides you a very important class you need to know about. The Assertion class provides a variety of static methods you can use in your test methods to actually test that what has happened is what you wanted to happen. The following sample shows what I mean:

namespace UnitTestingExamples

{

using System;

using NUnit.Framework;

[TestFixture]

public class SomeTests

{

[Test]

public void TestOne()

{

int i = 4;

Assertion.AssertEquals( 4, i );

}

}

}

(I know that isn't the most relevant bit of code, but it shows what I mean.)

### Running Your Tests

Now that we have covered the basics of the code, lets talk about how to run your tests. It is really quite simple. NUnit comes with two different Test Runner applications: a Windows GUI app and a console XML app. Which one you use is a matter of personal preference. To use the GUI app, just run the application and tell it where your test assembly resides. The test assembly is the class library (or executable) that contains the Test Fixtures. The app will then show you a graphical view of each class and test that is in that assembly. To run the entire suite of tests, simple click theRun button. If you want to run only one Test Fixture or even just a single test, you can double-click it in the tree. The following screenshot shows the GUI app:

![](data:None;base64,)

There are situations, particularly when you want to have an automated build script run your tests, when the GUI app isn't appropriate. In these automated builds, you typically want to have the output posted to a website or another place where it can be publicly reviewed by the development team, management or the customer. The NUnit 2.0 console application takes the assembly as a command-line argument and produces XML output. You can then use XSLT or CSS to convert the XML into HTML or any other format. For more information about the console application, check out the NUnit documentation.

## Doing Test-Driven Development

So now you know how to write unit tests, right? Unfortunately just like programming, knowing the syntax isn't enough. You need to have a toolbox of skills and techniques before you can build professional software systems. Here are a few techniques that you can use to get you started. Remember, however, that these tools are just a start. To really improve your unit testing skills, you must practice, practice, practice. If you are unfamiliar with TDD, what I'm about to say may sound a little strange to you. A lot of people have spent a lot of time telling us that we should carefully design our classes, code them up and then test them. What I'm going to suggest is a completely different approach. Instead of designing a module, then coding it and then testing it, you turn the process around and do the testing first. To put it another way, you don't write a single line of production code until you have a test that fails. The typical programming sequence is something like this:

* Write a test.
* Run the test. It fails to compile because the code you're trying to test doesn't even exist yet! (This is the same thing as failing.)
* Write a bare-bones stub to make the test compile.
* Run the test. It should fail. (If it doesn't, then the test wasn't very good.)
* Implement the code to make the test pass.
* Run the test. It should pass. (If it doesn't, back up one step and try again.)
* Start over with a new test!

While you are doing step #5, you create your code using a process called Coding by Intention. When you practice Coding by Intention you write your code top-down instead of bottom up. Instead of thinking, "I'm going to need this class with these methods," you just write the code that you want... before the class you need actually exists. If you try to compile your code, it fails because the compiler can't find the missing class. This is a good thing, because as I said above, failing to compile counts as a failing test. What you are doing here is expressing the intention of the code that you are writing. Not only does this help produce well-tested code, it also results in code that is easier to read, easier to debug and has a better design. In traditional software development, tests were thought to verify that an existing bit of code was written correctly. When you do TDD, however, your tests are used to define the behavior of a class before you write it. I won't suggest that this is easier than the old ways, but in my experience it is vastly better. If you have read about Extreme Programming, then this is primarity a review. However, if this is new to you, here is a sample. Suppose the application that I'm writing has to allow the user to make a deposit in a bank account. Before creating aBankAccount class, I create a class in my testing library called BankAccountTests. The first thing I need my bank account class to do is be able to take a deposit and show the correct balance. So I write the following code:

namespace UnitTestingExamples.Tests

{

using System;

using NUnit.Framework;

[TestFixture]

public class BankAccountTests

{

[Test]

public void TestDeposit()

{

BankAccount account = new BankAccount();

account.Deposit( 125.0 );

account.Deposit( 25.0 );

Assertion.AssertEquals( 150.0, account.Balance );

}

}

}

Once this is written, I compile my code. It fails of course because the BankAccount class doesn't exist. This illustrates the primary principle of Test-Driven Development: don't write any code unless you have a test that fails. Remember, when your test code won't compile, that counts as a failing test. Now I create my BankAccount class and I write just enough code to make the tests compile:

namespace UnitTestingExamples.Library

{

using System;

public class BankAccount

{

public void Deposit( double amount )

{

}

public double Balance

{

get { return 0.0; }

}

}

}

This time everything compiles just fine, so I go ahead and run the test. My test fails with the message "TestDeposit: expected: <150> but was <0>". So the next thing we do it write just enough code to make this test pass:

namespace UnitTestingExamples.Library

{

using System;

public class BankAccount

{

private double \_balance = 0.0;

public void Deposit( double amount )

{

\_balance += amount;

}

public double Balance

{

get { return \_balance; }

}

}

}

This time our tests pass.

### Using Mock Objects - DotNetMock

One of the biggest challenges you will face when writing units tests is to make sure that each test is only testing one thing. It is very common to have a situation where the method you are testing uses other objects to do its work. If you write a test for this method you end up testing not only the code in that method, but also code in the other classes. This is a problem. Instead we use mock objects to ensure that we are only testing the code we intend to test. A mock object emulates a real class and helps test expectations about how that class is used. Most importantly, mock objects are:

* Easy to make
* Easy to set up
* Fast
* Deterministic (produce predictable results)
* Allow easy verification the correct calls were made, perhaps in the right order

The following example shows a typical mock object usage scenario. Notice that the test code is clean, easy to understand and not dependent on anything except the code being tested.

namespace UnitTestingExamples.Tests

{

using DotNetMock;

using System;

[TestFixture]

public class ModelTests

{

[Test]

public void TestSave()

{

MockDatabase db = new MockDatabase();

db.SetExpectedUpdates(2);

ModelClass model = new ModelClass();

model.Save( db );

db.Verify();

}

}

}

As you can see, the MockDatabase was easy to setup and allowed us to confirm that the Save method made certain calls on it. Also notice that the mock object prevents us from having to worry about real databases. We know that when the ModelClass saves itself, it should call the database's Update method twice. So we tell the MockDatabaseto expect two updates calls, call Save and the confirm that what we expected really happened. Because theMockDatabase doesn't really connect to a database, we don't have to worry about keeping "test data" around. We only test that the Save code causes two updates.

"When Mock Objects are used, only the unit test and the target domain code are real." -- Endo-Testing: Unit Testing with Mock Objects by Tim Mackinnon, Steve Freeman and Philip Craig.

### Testing the Business Layer

Testing the business layer is where most developers feel comfortable talking about unit testing. Well designed business layer classes are loosely coupled and highly cohesive. In practical terms, coupling described the level of dependence that classes have with one another. If a class is loosely coupled, then making a change to one class should not have an impact on another class. On the other hand, a highly cohesive class is a class that does the one thing is was designed to do and nothing else. If you create your business layer class library so that the classes are loosely coupled and highly cohesive, then creating useful unit tests is easy. You should be able to create a single test class for each business class. You should be able to test each of its public methods with a minimal amount of effort. By the way, if you are having a difficult time creating unit tests for your business layer classes, then you probably need to do some significant refactoring. Of course, if you have been writing your tests first, you shouldn't have this problem.

### Testing the User Interface

When you start to write the user interface for your application, a number of different problems arise. Although you can create user interface classes that are loosely coupled with respect to other classes, a user interface class is by definition highly coupled to the user! So how can we create a automated unit test to test this? The answer is that we separate the logic of our user interface from the actual presentation of the view. Various patterns exist in the literature under a variety of different names: Model-View-Controller, Model-View-Presenter, Doc-View, etc. The creators of these patterns recognized that decoupling the logic of what the view does (i.e., controller) from the view is a Good Thing. So how do we use this? The technique I use comes from Michael Feathers' paper [The Humble Dialog Box](http://www.objectmentor.com/resources/articles/TheHumbleDialogBox.pdf). The idea is to make the view class support a simple interface used for getting and setting the values displayed by the view. There is basically no code in the view except for code related to the painting of the screen. The event handlers in the view for each interactive user interface element (e.g., a button) contain nothing more than a pass-thru to a method in the controller. The best way to illustrate this concept is with an example. Assume our application needs a screen that asks the user for their name and social security number. Both fields are required, so we need to make sure that a name is entered and the SSN has the correct format. Since we are writing our unit tests first, we write the following test:

[TestFixture]

public class VitalsControllerTests

{

[Test]

public void TestSuccessful()

{

MockVitalsView view = new MockVitalsView();

VitalsController controller = new VitalsController(view);

view.Name = "Peter Provost";

view.SSN = "123-45-6789";

Assertion.Assert( controller.OnOk() == true );

}

[Test]

public void TestFailed()

{

MockVitalsView view = new MockVitalsView();

VitalsController controller = new VitalsController(view);

view.Name = "";

view.SSN = "123-45-6789";

view.SetExpectedErrorMessage( controller.ERROR\_MESSAGE\_BAD\_NAME );

Assertion.Assert( controller.OnOk() == false );

view.Verify();

view.Name = "Peter Provost";

view.SSN = "";

view.SetExpectedErrorMessage( controller.ERROR\_MESSAGE\_BAD\_SSN );

Assertion.Assert( controller.OnOk() == false );

view.Verify();

}

}

When we build this we receive a lot of compiler errors because we don't have either a MockVitalsView or aVitalsController. So let's write skeletons of those classes. Remember, we only want to write enough to make this code compile.

public class MockVitalsView

{

public string Name

{

get { return null; }

set { }

}

public string SSN

{

get { return null; }

set { }

}

public void SetExpectedErrorMessage( string message )

{

}

public void Verify()

{

throw new NotImplementedException();

}

}

public class VitalsController

{

public const string ERROR\_MESSAGE\_BAD\_SSN = "Bad SSN.";

public const string ERROR\_MESSAGE\_BAD\_NAME = "Bad name.";

public VitalsController( MockVitalsView view )

{

}

public bool OnOk()

{

return false;

}

}

Now our test assembly compiles and when we run the tests, the test runner reports two failures. The first occurs whenTestSuccessful calls controller.OnOk, because the result is false rather than the expected true value. The second failure occurs when TestFailed calls view.Verify. Continuing on with our test-first paradigm, we now need to make these tests pass. It is relatively simple to make TestSuccessful pass, but to make TestFailed pass, we actually have to write some real code, such as:

public class MockVitalsView : MockObject

{

public string Name

{

get { return \_name; }

set { \_name = value; }

}

public string SSN

{

get { return \_ssn; }

set { \_ssn = value; }

}

public string ErrorMessage

{

get { return \_expectedErrorMessage.Actual; }

set { \_expectedErrorMessage.Actual = value; }

}

public void SetExpectedErrorMessage( string message )

{

\_expectedErrorMessage.Expected = message;

}

private string \_name;

private string \_ssn;

private ExpectationString \_expectedErrorMessage =

new ExpectationString("expected error message");

}

public class VitalsController

{

public const string ERROR\_MESSAGE\_BAD\_SSN = "Bad SSN.";

public const string ERROR\_MESSAGE\_BAD\_NAME = "Bad name.";

public VitalsController( MockVitalsView view )

{

\_view = view;

}

public bool OnOk()

{

if( IsValidName() == false )

{

\_view.ErrorMessage = ERROR\_MESSAGE\_BAD\_NAME;

return false;

}

if( IsValidSSN() == false )

{

\_view.ErrorMessage = ERROR\_MESSAGE\_BAD\_SSN;

return false;

}

*//* *All is well, do something...*

return true;

}

private bool IsValidName()

{

return \_view.Name.Length > 0;

}

private bool IsValidSSN()

{

string pattern = @"^\d{3}-\d{2}-\d{4}$";

return Regex.IsMatch( \_view.SSN, pattern );

}

private MockVitalsView \_view;

}

Let's briefly review this code before proceeding. The first thing to notice is that we haven't changed the tests at all (which is why I didn't even bother to show them). We did, however, make significant changes to bothMockVitalsView and VitalsController. Let's begin with the MockVitalsView. In our previous example,MockVitalsView didn't derive from any base class. To make our lives easier, we changed it to derive fromDotNetMock.MockObject. The MockObject class gives us a stock implementation of Verify that does all the work for us. It does this by using expectation classes through which we indicate what we expect to happen to our mock object. In this case our tests are expecting specific values for the ErrorMessage property. This property is a string, so we add an ExpectationString member to our mock object. Then we implement the SetExpectedErrorMessagemethod and the ErrorMessage property to use this object. When we call Verify in our test code, the MockObjectbase class will check this expectation and identify anything that doesn't happen as expected. Pretty cool, eh? The other class that changed was our VitalsController class. Because this is where all the working code resides, we expected there to be quite a few changes here. Basically, we implemented the core logic of the view in the OnOk method. We use the accessor methods defined in the view to read the input values, and if an error occurs, we use theErrorMessage property to write out an appropriate message. So we're done, right? Not quite. At this point, all we have is a working test of a controller using a mock view. We don't have anything to show the customer! What we need to do is use this controller with a real implementation of a view. How do we do that? The first thing we need to do is extract an interface from MockVitalsView. A quick look at VitalsController andVitalsControllerTests shows us that the following interface will work.

public interface IVitalsView

{

string Name { get; set; }

string SSN { get; set; }

string ErrorMessage { get; set; }

}

After creating the new interface, we change all references to MockVitalsView to IVitalsView in the controller and we add IVitalsView to the inheritance chain of MockVitalsView. And, of course, after performing this refactoring job we run our tests again. Assuming everything is fine, we can create our new view. For this example I will be creating an ASP.NET page to act as the view, but you could just as easily create a Windows Form. Here is the .ASPX file:

<%@ Page language="c#" Codebehind="VitalsView.aspx.cs"

AutoEventWireup="false"

Inherits="UnitTestingExamples.VitalsView" %>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >

<html>

<head>

<title>VitalsView</title>

<meta name="GENERATOR" Content="Microsoft Visual Studio 7.0">

<meta name="CODE\_LANGUAGE" Content="C#">

<meta name=vs\_defaultClientScript content="JavaScript">

<meta name=vs\_targetSchema

content="http://schemas.microsoft.com/intellisense/ie5">

</head>

<body MS\_POSITIONING="GridLayout">

<form id="VitalsView" method="post" runat="server">

<table border="0">

<tr>

<td>Name:</td>

<td><asp:Textbox runat="server" id=nameTextbox /></td>

</tr>

<tr>

<td>SSN:</td>

<td><asp:Textbox runat="server" id=ssnTextbox /></td>

</tr>

<tr>

<td> </td>

<td><asp:Label runat="server" id=errorMessageLabel /></td>

</tr>

<tr>

<td> </td>

<td><asp:Button runat="server" id=okButton Text="OK" /></td>

</tr>

</table>

</form>

</body>

</html>

And here is the code-behind file:

using System;

using System.Web.UI.WebControls;

using UnitTestingExamples.Library;

namespace UnitTestingExamples

{

/// *<span class="code-SummaryComment"><summary></span>*

/// *Summary description for VitalsView.*

/// *<span class="code-SummaryComment"></summary></span>*

public class VitalsView : System.Web.UI.Page, IVitalsView

{

protected TextBox nameTextbox;

protected TextBox ssnTextbox;

protected Label errorMessageLabel;

protected Button okButton;

private VitalsController \_controller;

private void Page\_Load(object sender, System.EventArgs e)

{

\_controller = new VitalsController(this);

}

private void OkButton\_Click( object sender, System.EventArgs e )

{

if( \_controller.OnOk() == true )

Response.Redirect("ThankYou.aspx");

}

#region IVitalsView Implementation

public string Name

{

get { return nameTextbox.Text; }

set { nameTextbox.Text = value; }

}

public string SSN

{

get { return ssnTextbox.Text; }

set { ssnTextbox.Text = value; }

}

public string ErrorMessage

{

get { return errorMessageLabel.Text; }

set { errorMessageLabel.Text = value; }

}

#endregion

#region Web Form Designer generated code

override protected void OnInit(EventArgs e)

{

*//*

*//* *CODEGEN: This call is required by the ASP.NET Web Form Designer.*

*//*

InitializeComponent();

base.OnInit(e);

}

/// *<span class="code-SummaryComment"><summary></span>*

/// *Required method for Designer support - do not modify*

/// *the contents of this method with the code editor.*

/// *<span class="code-SummaryComment"></summary></span>*

private void InitializeComponent()

{

this.Load += new System.EventHandler(this.Page\_Load);

okButton.Click += new System.EventHandler( this.OkButton\_Click );

}

#endregion

}

}

As you can see, the only code in the view is code that ties the IVitalsView interface to the ASP.NET Web Controls and a couple of lines to create the controller and call its methods. Views like this are easy to implement. Also, because all of the real code is in the controller, we can feel confident that we are rigorously testing our code.

## Conclusion

Test-driven development is a powerful technique that you can use today to improve the quality of your code. It forces you to think carefully about the design of your code, and is ensures that all of your code is tested. Without adequate unit tests, refactoring existing code is next to impossible. Very few people who take the plunge into TDD later decide to stop doing it. Try and you will see that it is a Good Thing