Unit Testing and TDD

In the history of software development, there has always been one adversary that has plauged the heroic software developer. I'm talking about the arch enemey known as the "bug". The bug has reaked havoc in the careers of every programmer since software was developed on paper punch cards. They strike without warning. They feed on the time and budget of every project. They comprimise the integrity of your data. They inspire angry support calls. These elusive creatures can pop up at any moment. Just as fast as they appear, they have the ability to hide deep within your source code, as if they never existed.

Have you heard the phrase "It worked on my machine"? If not, you are lucky. I too have uttered these damning words myself. This is the phrase spoken by unfortunate developers and software testers when trying to reproduce a defect submitted by a customer. It's a difficult task to explain to a paying customer that you are unable to lure these horrific beast out of hiding when you perform the exact steps that they've submitted in an angry email or support call.

Indeed, the software bug can vary in scope from an annoying session time out, straight through to a data trashing disaster. "But how can I defeat such a cunning and clever pest?" you ask? Just like in the real world, there is no silver bullet when it comes to bugs in software development. The main strategies that have been used throughout the years to discover and destroy bugs in a software system include:

* Defensive programming

In my opinion, defensive programming should be a part of every developers skill set. Defensive programming is the practice of designing all of the methods that you write to try and anticipate all possible points of error and then writing code to gracefully handle such errors so that they don't cause problems for the end users of the application. This includes identifying all valid input argument values and writing code to test all inputs to handle any invalid values. I firmly believe that with a little extra effort, we can make null reference exceptions into campfire ghost stories rather than emergency build releases. You too can prevent null reference exceptions.

In data entry applications, it's important to identify all sources of data that enters the system and to write code to protect the data at all cost. After all, I'm sure you've heared the old saying, "Garbage in, Garbage out". I prefer to use the acronym G.I.G.O (OK, so maybe that was a bad joke, but hey, I've got a whole System.Collections.Queue of them…).

It's important to keep in mind that sources of data are not limited to human data entry. Many enterprise software applications communicate with other programs to send and receive data. These sources include systems such as web servies, imaging scanners, message queues, and one of the most common sources, database systems. You can write as much defenseive code as you like, however, if your system shares a database with another application, you don't have control over the quality of data that is produced by these external systems. This is why it's imperative that you check all data, regardless of its origin, for potential issues.

Defensive programming is essentially the practice of writing code to anticipate all possible error conditions for a logical unit of code and writing code to gracefully handle the error. Bad data will always find its way into your application, however, you can handle the error such that the user doesn't know that there was a problem. For instance, a well designed application will handle the error, write a detailed description of the problem to a database, log file, or event log. It's also handy to write code to send an email to your support team to alert them of the issue. If you write code with the user in mind, errors that can be handled by the system should be invisible to the end user. User's panic when they encounter a cryptic unhandled exception web page or message box. As a rule of thumb, try not to interupt the user's workflow if at all possible.

In the event that your application cannot recover from a "fatal" error, it's best to notify the user with an apologetic, aesthetically pleasing, human readable message to let them know that there was a problem encountered. Always be sure to provide instructions to the user as to what they should do next. If at all possible, try to restart the application after all of the relevant error details have been logged for later investigation.

* System testing

System testing is the practice of testing an application, in its entirety. This is often the job of a quality assurance team, however, if your company is small and you don't have a QA team, then this responsibility falls on you, the developer. It's best to create a collection of test cases so that testing is consistent from build to build, however, this approach involves testing the exact same steps each time. Considering this drawback, it's good practice to also incorporate what I like to call "break challenge testing". What is "break challenge testing" you ask? It's when the tester of an application challenges himself to "break" the system by any means necessary (of course I mean by using absurd data inputs, clicking multiple toolbar buttons or menu items, not beating the computer with a blunt object). It's important to think like a user. Often times, we as developers and testers know where the quirks are and we know how to avoid them. An enterprise system shouldn't allow you to compromise the data's integrity or crash the application. So

* Regression testing
* User Acceptance testing

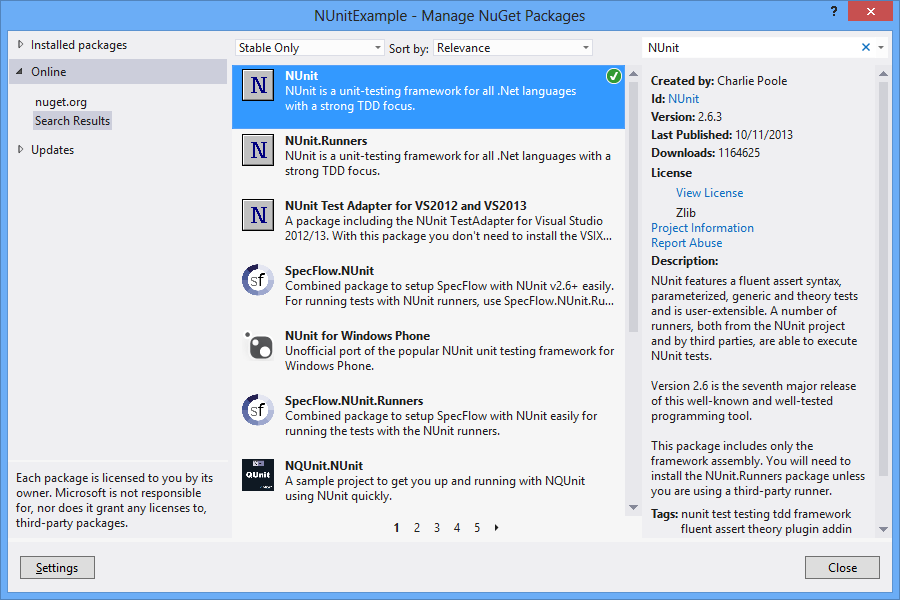
# Unit testing frameworks

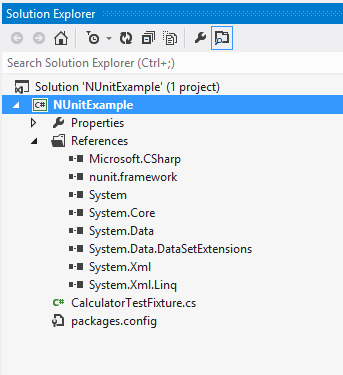
## NUnit

NUnit is an open source unit testing framework that is based on the JUnit Java unit testing framework. NUnit is very popular with open source advocates. The framework is a complete unit testing tool which includes the following:

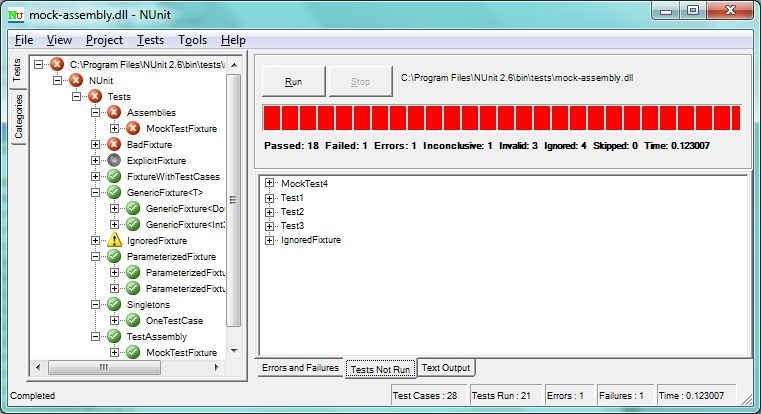
* Easy installation via NuGet

Instructions on how to install go here.





* Integration with Visual Studio
* A command line interface that you can use to execute your unit tests. This makes automated test execution when building your solution from the command line a breeze.
* A GUI interface that provides visual cues regarding the status of your testing efforts such as: which test is executing, which tests have passed, which tests have failed, and which tests have yet to be ran.

fdsfafdsafdsafdsaFigure 1-1 NUnit Test Runner GUI

* Several test method attributes used to mark special test methods for purposes such as test setup, teardown, expected exceptions, and more. Here are a few attributes that you will most likely find yourself using in most of your unit test fixture class methods. A comprehensive list can be found here <http://nunit.org/index.php?p=attributes&r=2.6.3>.

The [TestFixture] attribute identifies a class whose purpose is to provide all unit test methods. The unit test methods are marked with the [Test] attribute.

When writing unit tests, often times you must write boiler plate code that is used to prepare dependencies and other related objects that are used in multiple unit test methods. In honor of the DRY principle (don't repeat yourself), NUnit offers the [Setup] attribute which indicates that the specified method should be ran before each test is executed. This allows you to initialize all of the related variables to a predefined state before running each test. This is a perfect place to create stubs and mock objects to be used in your unit tests.

Much like the [Setup] attribute, the [TearDown] attribute designates a method to be used in disposing of any unmanaged resources that aren't handled by the .NET CLR garbage collector. A few examples of such resources include: database connections, web service references, file system objects, etc. This is also a prime candidate for setting all object references to null, deleting any files created as a part of your unit tests. If your test creates any objects in memory, or any other type of resource that should be disposed of, this is the attribute to use.

* A plethora of methods that allow you to assert that your code is executing as expected. You can find all of the assertion methods in the official NUnit documentation at the following URL: <http://nunit.org/index.php?p=assertions&r=2.6.3>.

As you can see, the NUnit unit testing framework is a comprehensive testing suite and best of all, its open source (and free). Below you will find an example NUnit test fixture class.

ICalculator.cs code listing

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace NUnitExample

{

public interface ICalculator : IDisposable

{

int Add(int numberOne, int numberTwo);

int Subtract(int numberOne, int numberTwo);

int Divide(int numberOne, int numberTwo);

int Multiply(int numberOne, int numberTwo);

}

}

Calculator.cs code listing

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace NUnitExample

{

/// <summary>

/// Represents a primative calculator for demonstration purposes

/// </summary>

public class Calculator : ICalculator, IDisposable

{

#region ICalculator Members

/// <summary>

/// Add two integers and return the sum to the caller

/// </summary>

/// <param name="numberOne">a number to add</param>

/// <param name="numberTwo">a number to add</param>

/// <returns>The sum of the two input arguments</returns>

public int Add(int numberOne, int numberTwo)

{

return numberOne + numberTwo;

}

/// <summary>

/// Subtract two integers and return the difference to the caller

/// </summary>

/// <param name="numberOne">a number to subtract</param>

/// <param name="numberTwo">a number to subtract</param>

/// <returns>The difference between the two input arguments</returns>

public int Subtract(int numberOne, int numberTwo)

{

return numberOne - numberTwo;

}

/// <summary>

/// Divide two numbers

/// </summary>

/// <param name="numberOne">A number that should be divided by the seconde argument</param>

/// <param name="numberTwo">A number to use to divide the first argument</param>

/// <returns>The result of the division</returns>

/// <remarks>Generally, you should always check for a DivideByZero exception, however, we are

/// leaving this out to demonstrate how to test for expected exceptions in unit tests.</remarks>

public int Divide(int numberOne, int numberTwo)

{

return numberOne / numberTwo;

}

/// <summary>

/// Multiply two integers and return the product to the caller

/// </summary>

/// <param name="numberOne">a number to multiply</param>

/// <param name="numberTwo">a number to multiply</param>

/// <returns>The product of the multiplication between the two numbers</returns>

public int Multiply(int numberOne, int numberTwo)

{

return numberOne \* numberTwo;

}

#endregion

#region IDisposable Members

/// <summary>

/// if we had any unmanaged resources, we would free the associated resources here.

/// </summary>

public void Dispose()

{

Console.WriteLine("Clean as a whistle.");

}

#endregion

}

}

CalculatorTestFixture.cs code listing

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using NUnit.Framework;

namespace NUnitExample

{

[TestFixture]

public class CalculatorTestFixture

{

private ICalculator \_Calculator;

[SetUp]

public void Initialize\_Test()

{

//Create a new instance of the calculator class for each test.

\_Calculator = new Calculator();

}

[Test]

public void When\_Adding\_Two\_Integers\_The\_Method\_Should\_Return\_The\_Sum()

{

var numberOne = 2;

var numberTwo = 8;

Assert.AreEqual(10, \_Calculator.Add(numberOne, numberTwo));

}

[Test]

public void When\_Subtracting\_Two\_Integers\_The\_Method\_Should\_Return\_The\_Difference()

{

var numberOne = 10;

var numberTwo = 2;

Assert.AreEqual(8, \_Calculator.Subtract(numberOne, numberTwo));

}

[Test]

public void When\_Multiplying\_Two\_Integers\_The\_Method\_Should\_Return\_The\_Product()

{

var numberOne = 2;

var numberTwo = 2;

Assert.AreEqual(4, \_Calculator.Multiply(numberOne, numberTwo));

}

[Test]

[ExpectedException("System.DivideByZeroException")]

public void When\_Dividing\_A\_Number\_By\_Zero\_The\_Method\_Should\_Throw\_A\_DivideByZeroException()

{

var numberOne = 0;

var numberTwo = 0;

\_Calculator.Divide(numberOne, numberTwo);

}

[TearDown]

public void TearDown\_Test()

{

//perform any necessary clean up

\_Calculator.Dispose();

}

}

}

## XUnit

## NCrunch

## ContinuousTest

## MSTest

# Unit testing basics

## What makes a good unit test

## What makes a bad unit test

## Unit test fixtures

### Test method attributes

## Class under test

# Dealing with dependencies

## Stubs

## Mock objects

# Mocking frameworks

## Moq

## Rhino Mocks

## MS Fakes

# Design by testing: Test Driven Development

## Test first, code second

## Red, Green, Refactor

## Small iterations and refactoring

# Continuous integration and automated unit tests

# Further reading

**Note** There are some implementations of the layered architecture in which the user interface layer and the presentation layer are combined. This is often found in WinForms applications and some web applications. You could even take this approach with XAML based applications, however, as you will see, the awesome features that were introduced with WPF and XAML made way for a new paradigm for creating unit test friendly applications with rich user interfaces and components that can be built with no event handlers in the code behind. This separation of the user interface from the user interface logic makes XAML based applications a prime candidate for the five layer approach.

Table 1-1. This Is a Table Caption. The Style is Table Caption.

|  |  |
| --- | --- |
| Member Name | Description |
| CanExecute | The method which indicates if the specified command's state is valid for execution. For example, WPF implements a Command to support the Paste command when something is copied to the clipboard. If there is nothing to be pasted, then this method will return false, and any buttons that are bound to the Paste command will appear disabled until there is something on the clipboard to paste. |
| Execute | The method which contains the logic to execute when the command is triggered and the command's CanExecute method returns true. |

## Summary

Design patterns are extremely valuable to any software development team. The repository pattern will allow us to develop our data access code with no need to wait on the database to be created. We'll use the Adapter pattern to create an architecture that will make the software easier to change in order to meet the user's needs. This will also result in a code base that is easier to maintain as technologies and business needs change over time. We've covered the MVVM design pattern, which I consider to be a XAML application's bread and butter. We also learned how to use the ICommand interface to eliminate code-behind event handlers by creating Command classes that are dead simple to unit test. In the following chapters, we will analyze a business problem and use what we've learned to create a complete, multi-platform line of business application using domain driven design, complete with full unit test coverage.