**Tutorial NUnit 2.51**

Introducción

* 1. **¿Qué es NUnit?**

NUnit es un *framework* para pruebas unitarias en los lenguajes .Net. Inicialmente fue porteado de JUnit; el último lanzamiento de NUnit es la versión 2.6 al día 17 de Agosto del 2012, y es el séptimo lanzamiento de esta herramienta basada en xUnit para Microsoft .NET. Está escrito completamente en C# y ha sido completamente rediseñado para aprovechar muchas de las nuevas *features* de lenguaje .Net, por ejemplo: atributos personalizados y otras capacidades de reflexión.

* 1. **¿Qué es una prueba de software?**

De acuerdo a (Wikipedia, 2012), una prueba es una investigación conducida para proveer a los interesados (en este caso el equipo de desarrollo) con información sobre la calidad del producto o servicio bajo prueba. Entre las técnicas de pruebas, se incluye el proceso de ejecutar un programa o aplicación con objetivo de encontrar errores o defectos de software.

* 1. **¿Qué es una prueba unitaria?**

Se refiere a las pruebas que verifican la funcionalidad de una sección específica de código, usualmente al nivel de función. En un ambiente orientado a objetos, esto es usualmente al nivel de clase, y las pruebas unitarias mínimas incluyen los constructores y destructores de objetos. Cabe hacer notar que una prueba unitaria, para ser efectiva, debe hacerse de manera aislada, esto implica que no debe de depender ni verse afectado por ninguna otra pieza de código más que la que se encuentra bajo prueba.

En el contexto de pruebas unitarias escritas por un programador con una herramienta como NUnit, una prueba unitaria es una pieza código que ejercita una pequeña y específica área de funcionalidad del código siendo probado. Usualmente una prueba unitaria ejercita un método particular en un contexto particular. Por ejemplo, tal vez se agregue un valor grande al final de una lista, o se borre un patrón de caracteres de un *string* y luego confirmar que estos hayan sido borrados. (Hunt & thomas, 2003)

Las pruebas unitarias se realizan para probar que una pieza de código hace lo que un programador cree que debería hacer. De otra manera, cualquier resultado correcto sería coincidencia, cuando sucede esto, se le llama: “*programación por coincidencia*” (Hunt & Thomas, 1999), ya que, sin pruebas, si los finales resultados son correctos, entonces es mera coincidencia, pues no habría *pruebas* previas que demuestren que el resultado final iba a ser correcto.

* 1. **¿Qué es una prueba de integración?**

Es cualquier tipo de prueba de software que busca verificar las interfaces entre componentes, contra el diseño de software. Los componentes de software pueden ser integrados de una manera iterativa o todos juntos. El objetivo de las pruebas de integración es el de encontrar defectos en la interacción, por medio de sus interfaces, de los componentes o módulos integrados.

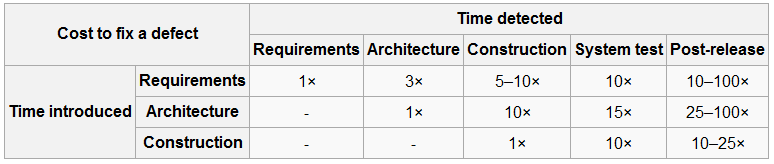
**1.5. ¿Por qué hacer pruebas?**

Razones para hacer pruebas:

- En un estudio conducido por la NIST en el 2002, encontró que los defectos de software cuestan $59.5 billones de dólares a los estados unidos de América. Más del 33% de este costo se pudo haber evitado y se hubieran hecho mejores pruebas de software

- Se sabe que conforme más temprano se encuentre un error, más barato es arreglarlo. Como se puede observar en la tabla a continuación, un defecto que se presente en la construcción de software, que es encontrado después del lanzamiento del software, puede costar %2500 de lo que hubiera costado arreglarlo en construcción.

1 Costo de arreglar un defecto, acorde al momento que se presenta.



**1.6. ¿Por qué hacer pruebas automatizadas?**

En (Matyas, Glover, & Duvall, 2007) nos explican algunas razones:

* Debido al corto tiempo entre la codificación y ejecución de pruebas sobre el código resultante, las pruebas unitarias son una forma eficiente de *debugging*
* Si vamos a construir sistemas que son verdaderamente confiables, tenemos que asegurar esa confiabilidad al nivel de objeto, lo que únicamente se puede lograr con pruebas unitarias [automatizadas]. Las pruebas deben efectivamente ejercitar el uso del objeto; y también, las pruebas deben ejecutarse frecuentemente (con cada cambio al código).
* Debido a que los objetos en un sistema y sus subsistemas de software deben comunicarse entre sí, las pruebas deben poder ejecutarse en cualquier momento y cada vez que cambia algo en cualquier parte del sistema. Esto sólo se puede lograr si las pruebas se ejecutan automáticamente.
* ¡Porque es divertido!

**NUnit**

**2.1 ¿Cómo funciona NUnit?**

Las pruebas de NUnit pueden ser escritas en cualquier lenguaje de .NET que tenga soporte para la clase *Attribute*. Las pruebas NUnit se definen por medio de atributos, aserciones y posiblemente con la ayuda de una variable de contexto de pruebas. Todos estos conceptos serán explicados en secciones posteriores.

Los [*Atributos*](http://nunit.org/index.php?p=attributes&r=2.6.1) indican qué clases son de pruebas, e igualmente identifican los métodos de prueba, así mismo, los atributos también pueden modificar el comportamiento de una prueba en varias maneras.

Las [Aserciones](http://nunit.org/index.php?p=assertions&r=2.6.1) prueban un valor resultante contra una o más [Condiciones](http://nunit.org/index.php?p=constraintModel&r=2.6.1), si el resultado cumple con todas las condiciones, la prueba se considera exitosa, de lo contrario se considera fallida.

El código de usuario en una prueba, puede adquirir información sobre la prueba y el ambiente en que ejecuta, por medio de la variable de [Contexto de Pruebas](http://nunit.org/index.php?p=testContext&r=2.6.1).

**2.2 Un ejemplo básico de NUnit**

Suponiendo que tenemos esta clase Account:

namespace Bank

{

    using System;

    public class Account

    {

        public void Deposit(decimal amount)

        {

            balance += amount;

        }

        public void Withdraw(decimal amount)

        {

            balance -= amount;

        }

        public void TransferFunds(Account destination, decimal amount)

        {

            throw new NotImplementedException();

        }

        public decimal Balance

        {

            get;

            private set;

        }

}

Una prueba unitaria para esta clase sería así:

namespace Bank.UnitTest

{

    using System;

    using NUnit.Framework;

    [TestFixture] // El atributo TestFixture identifica a esta clase como una clase de pruebas

    public class AccountTest

    {

        [Test] // El atributo Test identifica a este método como una prueba unitaria

        public void TransferFunds()

        {

            Account source = new Account();

            source.Deposit(200m);

            Account destination = new Account();

            destination.Deposit(150m);

            source.TransferFunds(destination, 100m);

            // Las aserciones se realizan por medio de la clase estática Assert

            // En este caso se trata de una aserción de igualdad por medio del método

            // estático AreEqual([valor esperado], [valor obtenido])

            Assert.AreEqual(250m, destination.Balance);

            // Otra forma de expresar la aserción es utilizando la interfaz

            // "fluida" de NUnit, donde el código de la aserción es

            // bastante legible.

            Assert.That(source.Balance, Is.EqualTo(100m));

        }

    }

}

Si ejecutásemos el caso de prueba codificado anteriormente, este iba a fallar marcando error. A continuación se te explicará cómo utilizar NUnit en tu ambiente de desarrollo con Visual Studio 2010.

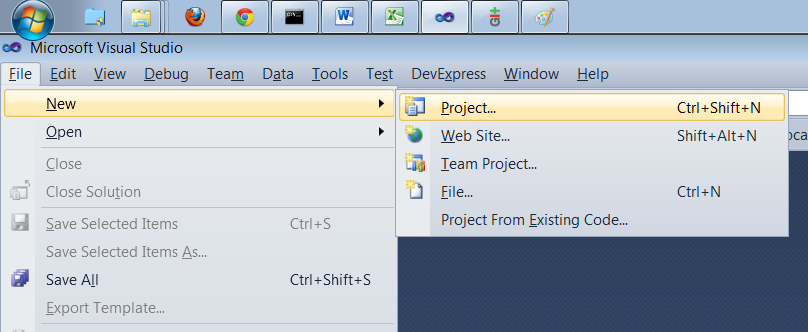
Todo el código de este tutorial lo puedes encontrar en: [GitHub](http://github.com/marcel-valdez/tutorial_nunit)

**2.3 Cómo preparar nuestro ambiente para utilizar NUnit**

Existen múltiples formas de preparar tu ambiente de desarrollo Visual Studio 2010 para utilizar NUnit, pero aquí se te describirá una forma sencilla y práctica de hacerlo.

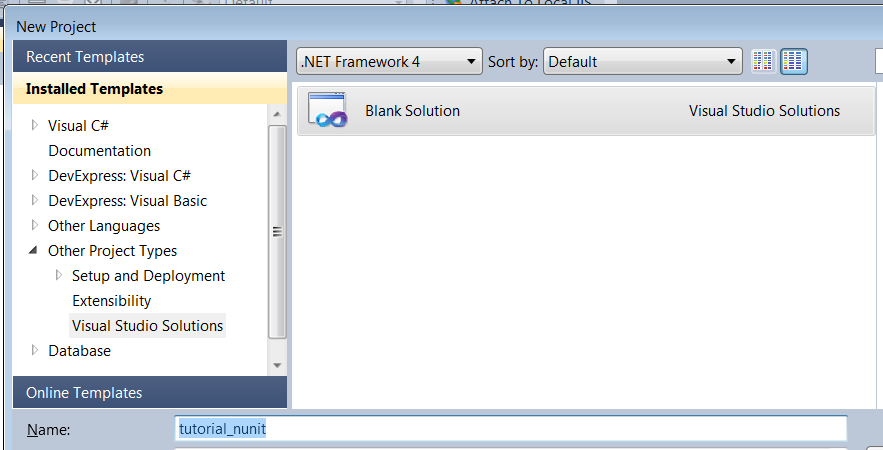
**Paso 1.** Inicia Visual Studio 2010

**Paso 2.** Crea una solución vacía de Visual Studio 2010.

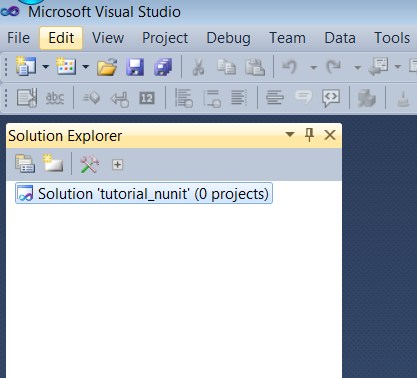


Subpaso 1 Entra a la pantalla de nuevo proyecto.

Subpaso 2 Escoge crear una solución en blanco llamada tutorial\_nunit

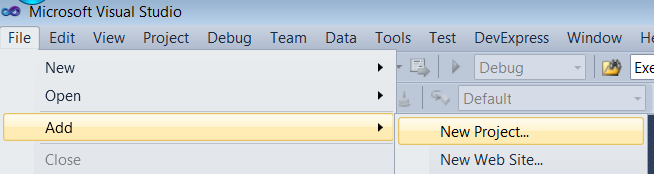


Subpaso 3 Te debería presentar esta estructura de proyecto.

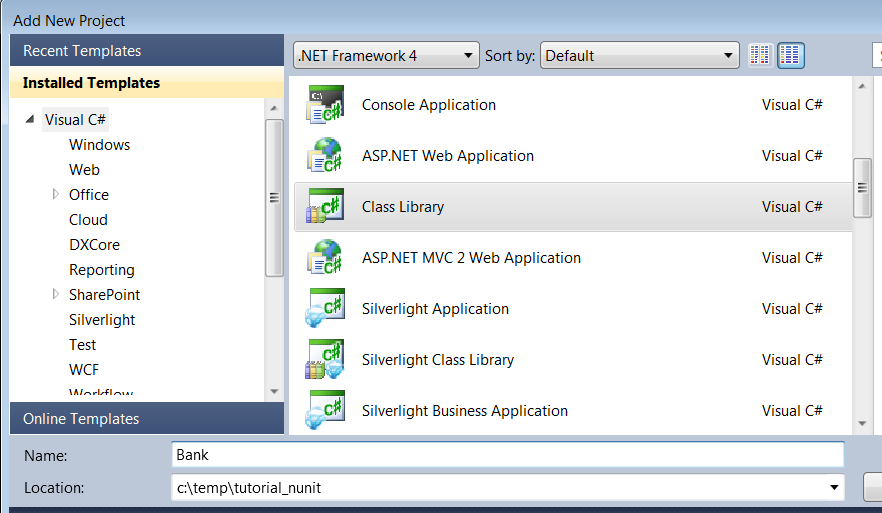


**Paso 3.** Agrega un proyecto C# del tipo librería, llamado: Bank

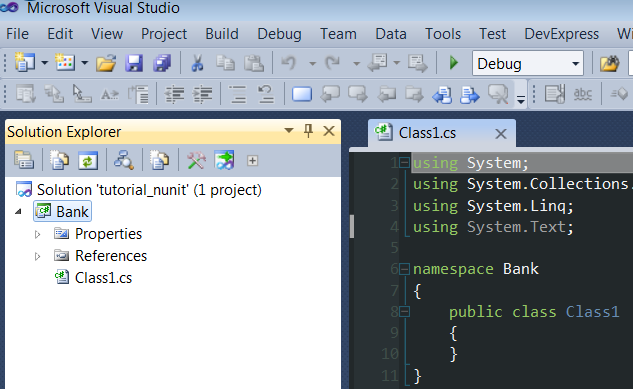
Subpaso 4 Entra a la pantalla para agregar un proyecto a la solución.



Subpaso 5 Agrega el proyecto del tipo librería de clases con el nombre Bank

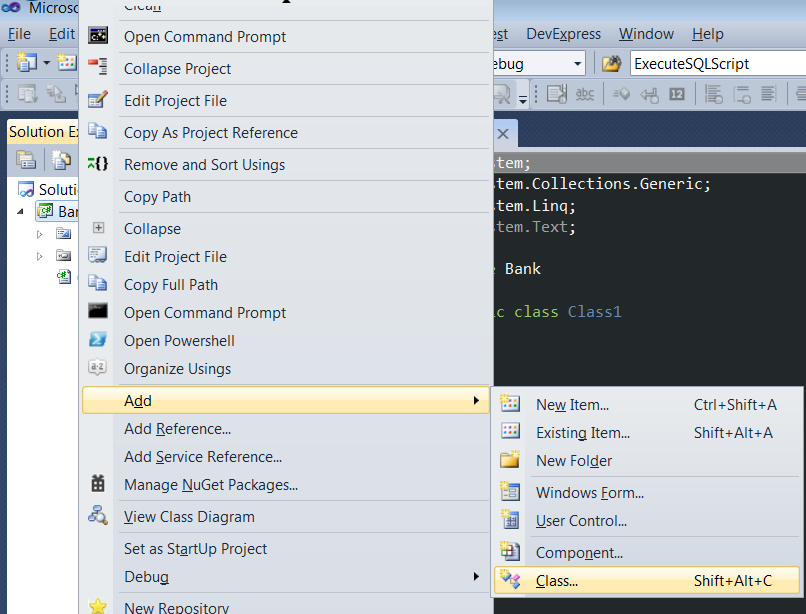


Subpaso 6 Pantalla resultante de la creación del proyecto.

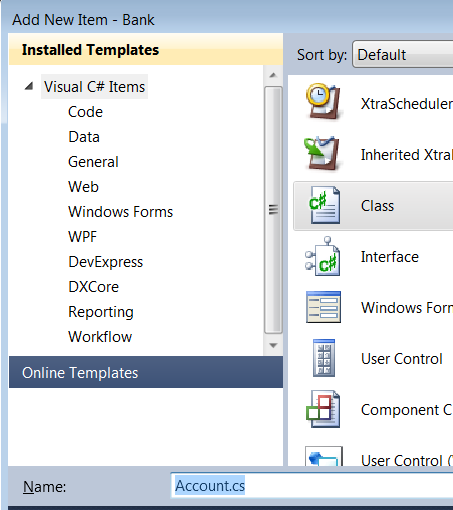


**Paso 4.** Agrega una clase llamada **Account** al proyecto.

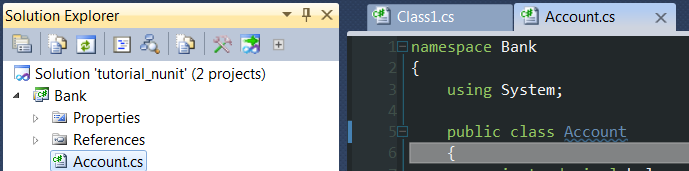
Subpaso 7 Entra al menú de creación de clases.



Subpaso 8 Crea la clase llamada Account



Subpaso 9 Pantalla resultante de la creación de la clase Account



**Paso 5.** Agrega el código en: <https://gist.github.com/3441171> a la clase **Account**.

namespace Bank

{

    using System;

    public class Account

    {

        public void Deposit(decimal amount)

        {

            balance += amount;

        }

        public void Withdraw(decimal amount)

        {

            balance -= amount;

        }

        public void TransferFunds(Account destination, decimal amount)

        {

            throw new NotImplementedException();

        }

        public decimal Balance

        {

            get;

            private set;

        }

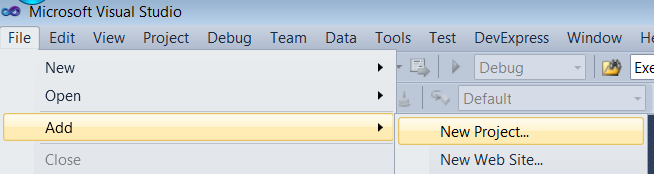
    }

}

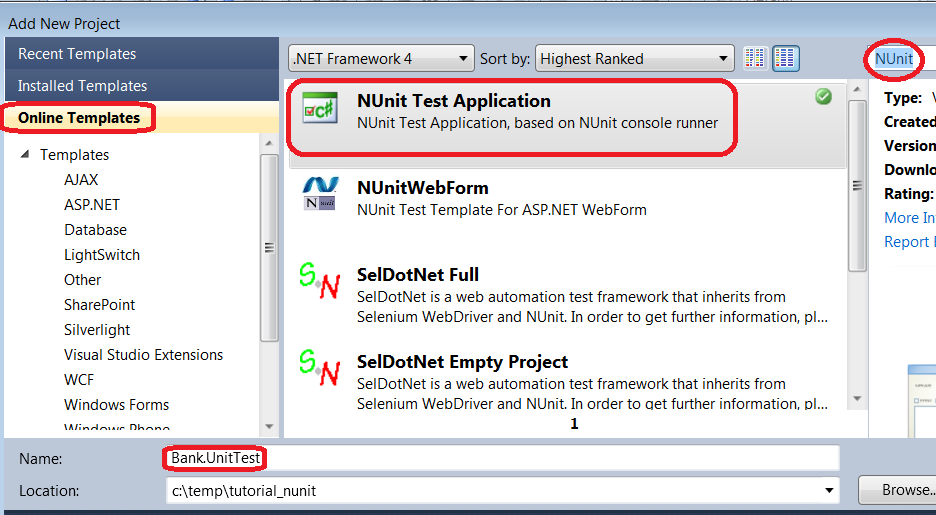
**Paso 6.** Agrega un proyecto de pruebas NUnit para ser ejecutado como programa de consola.

La forma más sencilla y práctica para hacer esto, es utilizando una de las plantillas NUnit en línea, que contienen un proyecto pre configurado para ser ejecutado como un programa de consola que ejercitará las pruebas unitarias. Hacer esto, es de lo más sencillo.

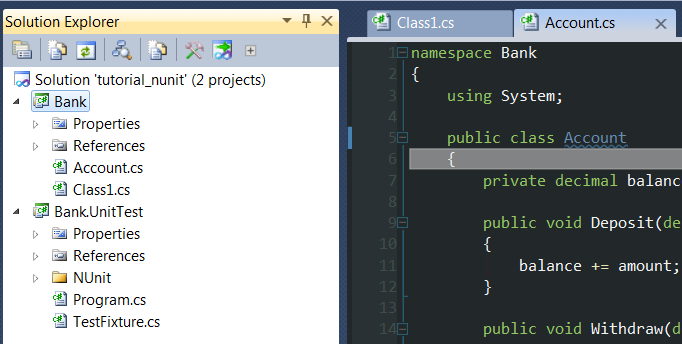
Subpaso 10 Entra a la pantalla para agregar un proyecto a la solución.



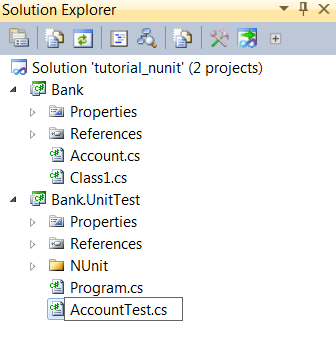
Subpaso 11 En la categoría de "Online Templates" haz una búsqueda por NUnit, y selecciona NUnit Test Application y ponle de nombre de proyecto: Bank.UnitTest



Subpaso 12 Estructura resultante de la creación del proyecto de pruebas.



**Paso 7.** Renombra la clase TestFixture a AccountTest



**Paso 8.** Agrega el código de: <https://gist.github.com/3441375> a la prueba unitaria.

﻿namespace Bank.UnitTest

 {

     using System;

     using NUnit.Framework;

     [TestFixture] // El atributo TestFixture identifica a esta clase como una clase de pruebas

     public class AccountTest

     {

         [Test] // El atributo Test identifica a este método como una prueba unitaria

         public void TransferFunds()

         {

             Account source = new Account();

             source.Deposit(200m);

             Account destination = new Account();

             destination.Deposit(150m);

             source.TransferFunds(destination, 100m);

             // Las aserciones se realizan por medio de la clase estática Assert

             // En este caso se trata de una aserción de igualdad por medio del método

             // estático AreEqual([valor esperado], [valor obtenido])

             Assert.AreEqual(250m, destination.Balance);

             Assert.AreEqual(100m, source.Balance);

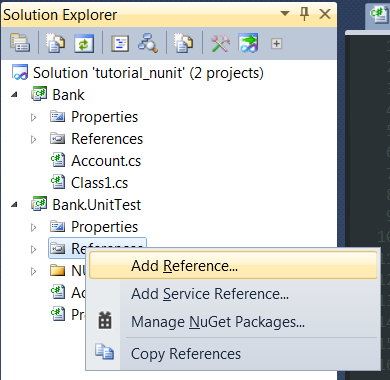
         }

     }

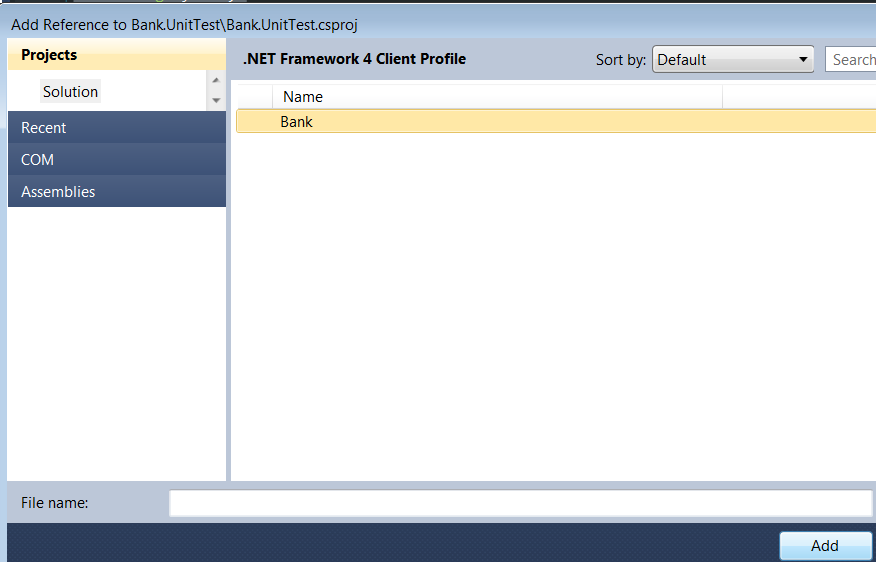
 }

**Paso 9.** La prueba unitaria aún no funciona, y el proyecto de pruebas no compila, esto se debe a que el proyecto de pruebas necesita una referencia al proyecto C# de la clase de librerías. Debes agregar una referencia al proyecto de librerías.

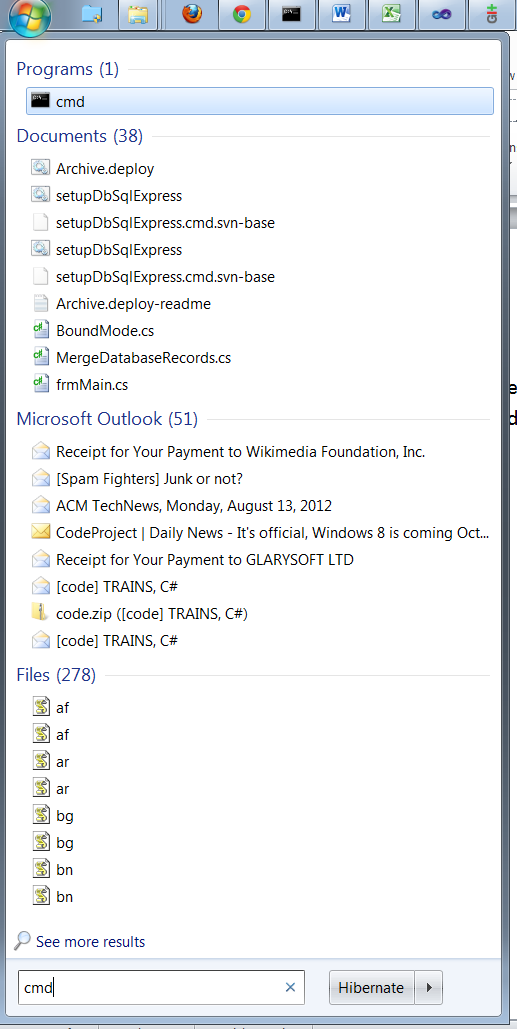
Subpaso 13 Entrar a la pantalla de manejo de referencias.



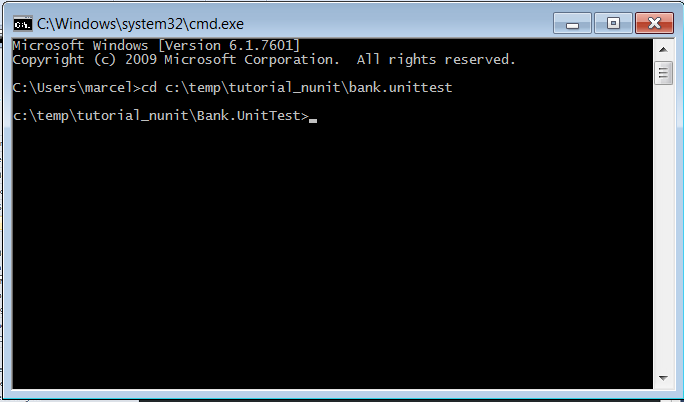
Subpaso 14 Agregas la referencia al proyecto Bank.



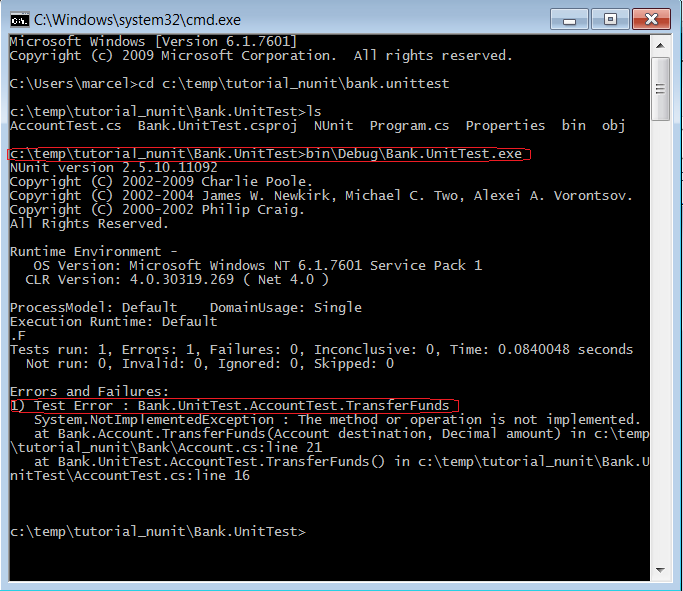
**Paso 10.** El proyecto de pruebas unitarias ya funciona correctamente, para ejecutarlo necesitas abrir la línea de comandos. En Windows 7, esto lo logras presionando el botón de inicio, y escribiendo cmd [enter]



Subpaso 15 En la línea de comandos, entras al directorio del proyecto de pruebas.



Subpaso 16 Ejecutas las pruebas unitarias con la instrucción: bin\debug\bank.unittest.exe



Como puedes observar, al ejecutar el programa Bank.UnitTest.exe, este invoca NUnit quién a su vez, invoca la prueba unitaria que definimos anteriormente, dándonos detalles sobre el error encontrado.

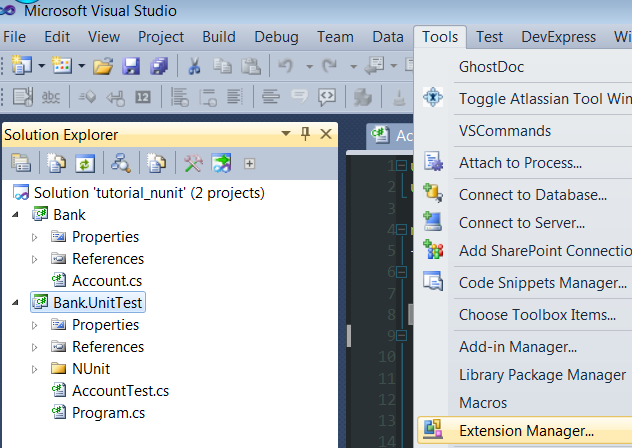
**2.3.1 Cómo instalar Visual NUnit**

Ejecutar las pruebas unitarias con la línea de comandos es una forma efectiva de hacerlo, ya que funcionará prácticamente en cualquier PC-Windows que encuentres, pues solamente dependes de la línea de comandos y la aplicación compilada, pero existen formas más cómodas de ejecutar las pruebas unitarias, esto normalmente se hace por medio de lo que se le llaman “Test Runners”, de estos hay varios, entre ellos están: [TestDriven.net](http://testdriven.net)®™, [CodeRush](http://devexpress.com/coderush)® de DevExpress™, [ReSharper](http://www.jetbrains.com/resharper)® de JetBrains™, todos ellos son de paga para uso profesional.

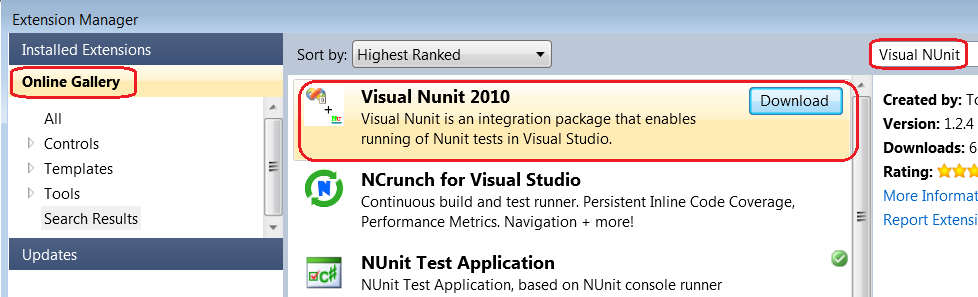
Existe otra opción gratuita y simple, se trata de una extensión de visual studio para ejecutar pruebas de NUnit: Visual NUnit. Para instalarlo, sigue las instrucciones a continuación:

**Paso 1.** Instala la extensión de Visual Studio.

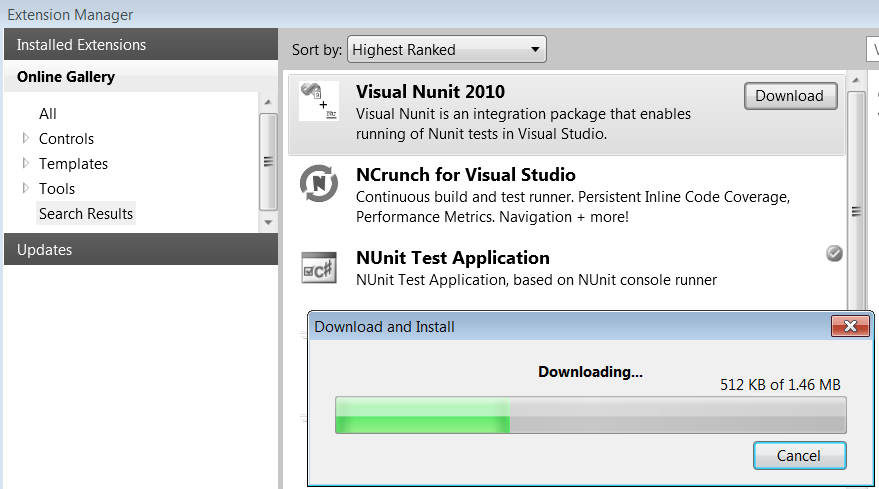
Subpaso 17 Entra al manejador de extensiones.



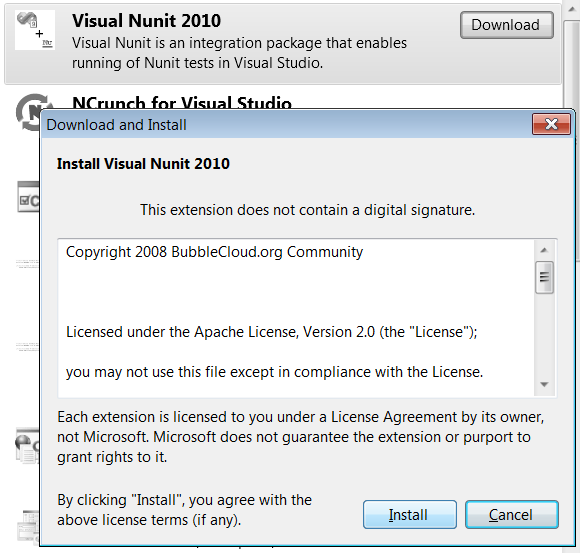
Subpaso 18 Haz una búsqueda en la Galería en línea por Visual NUnit



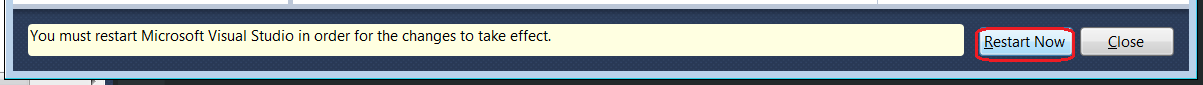
Subpaso 19 Presiona 'Download' para iniciar la instalación.



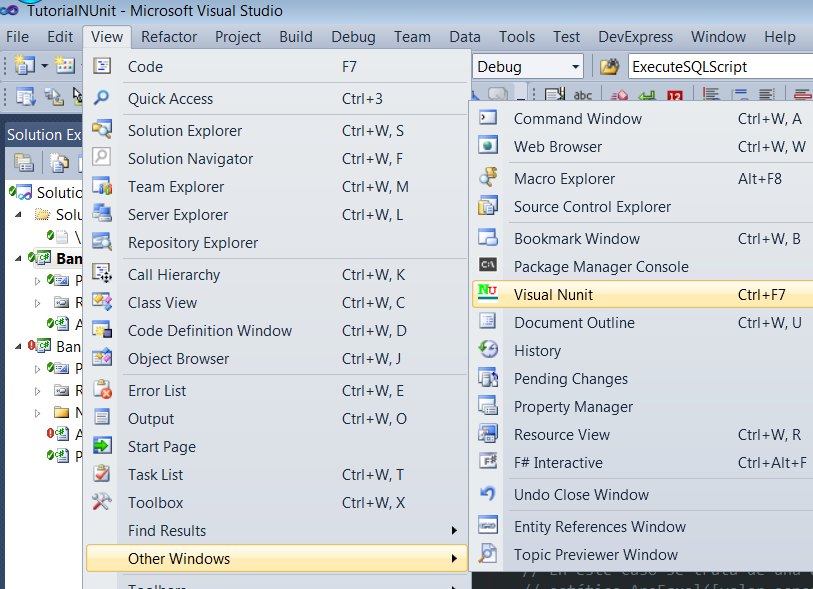
Subpaso 20 Al finalizar la instalación, continua presionando 'Install'



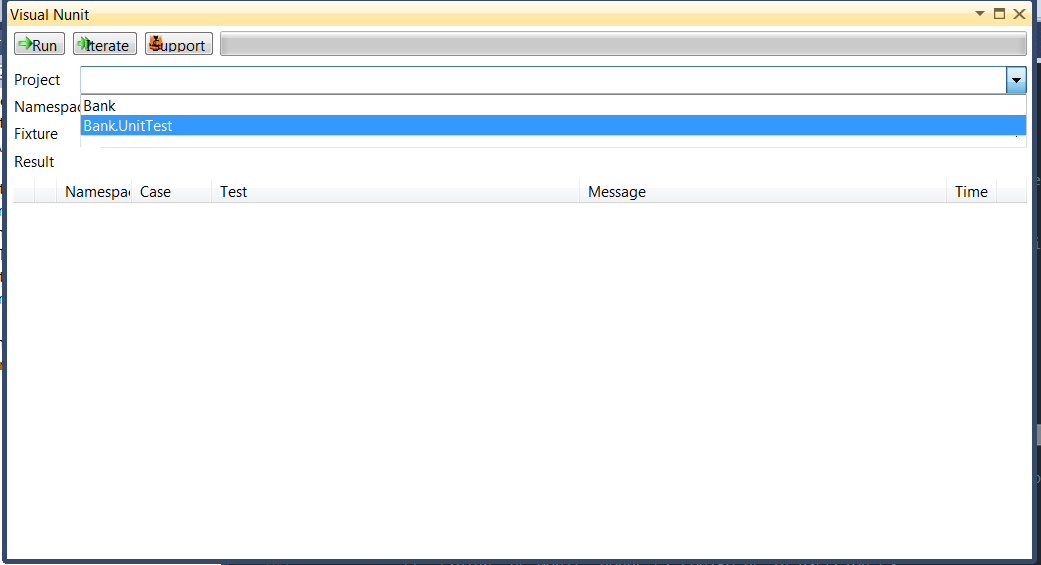
Subpaso 21 En la sección inferior del manejador de extensiones de dará la opción de reiniciar VS 2010, hazlo.



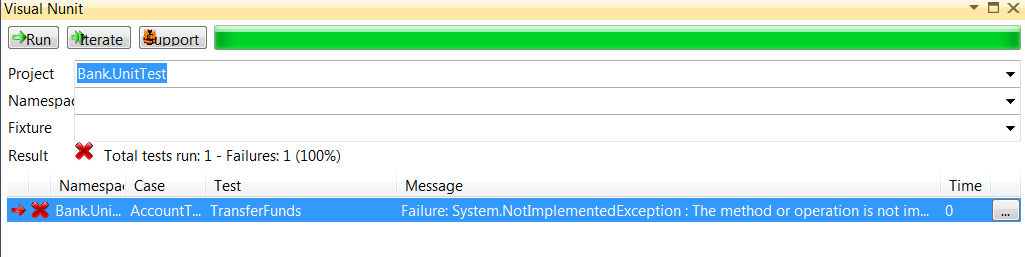
Subpaso 22 Después de reiniciar VS 2010, abre la ventana de Visual NUnit o presiona Ctrl + F7



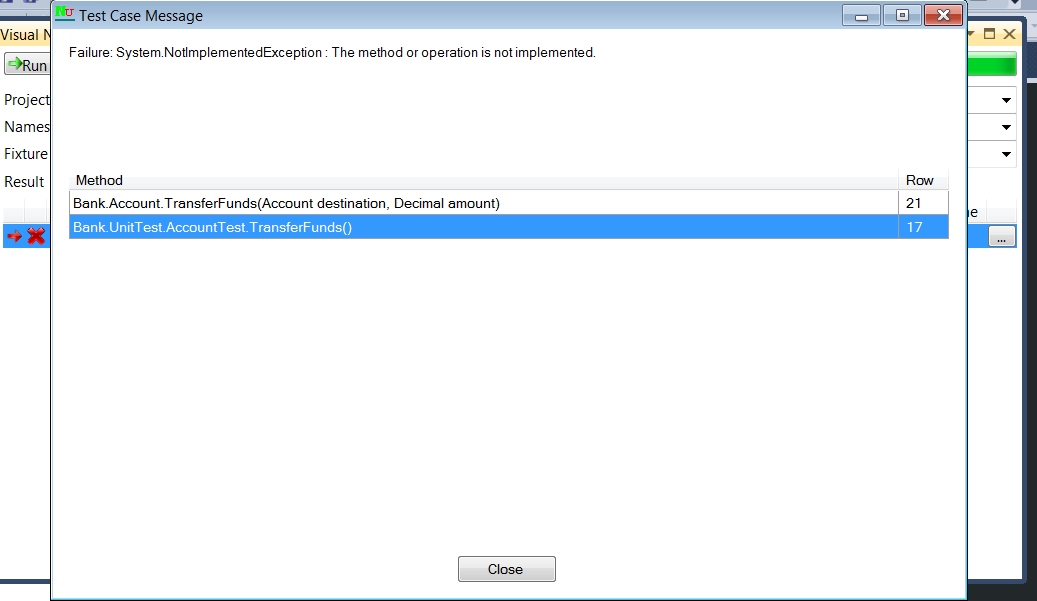
Subpaso 23 Esto abrirá la pantalla de Visual NUnit, donde debes elegir el proyecto de Bank.UnitTest.



Subpaso 24 Ahora presiona Run, y esto ejecutará todas las pruebas unitarias del proyecto Bank.UnitTest, si eliges Namespace y/o Fixture, puedes filtrar qué pruebas deseas ejecutar; en nuestro caso, solamente tenemos una prueba.



Subpaso 25 Si haces doble clic sobre la prueba que falló, puedes entrar a ver los detalles del error, donde te informará la línea de código donde ocurrió el error.



**2.3.2 Haciendo que pase el caso de prueba unitario.**

Ahora, corrige el método Bank.Account.TransferFunds(Account, decimal) para que pase la prueba unitaria, con el siguiente código:

        public void TransferFunds(Account destination, decimal amount)

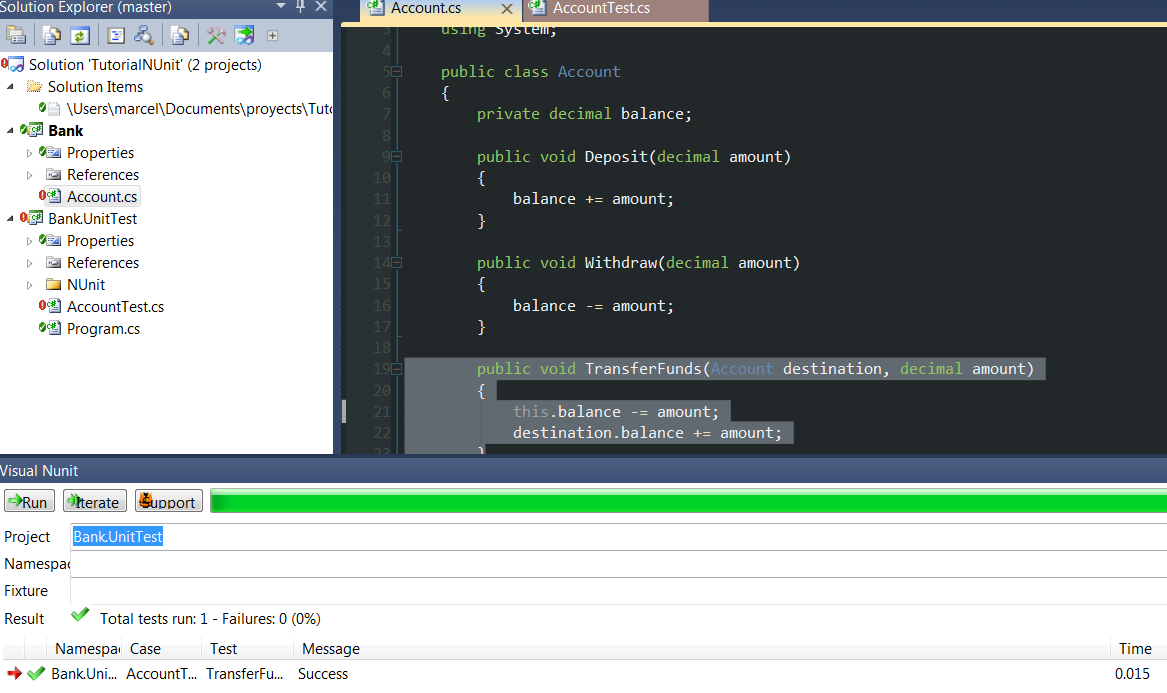
        {

            this.balance -= amount;

            destination.balance += amount;

        }

Ejecuta de nuevo las pruebas en el proyecto Bank.UnitTest:



Ahora es exitosa la ejecución de las pruebas unitarias, como se muestra en la imagen anterior. Claro que el código anterior tiene errores, pues sabemos que si el objeto **Account** no tiene suficientes fondos para la transferencia, entonces no se le debería permitir realizar la transferencia. Agreguemos ese caso de prueba.

**2.3.3 Agregando otro caso de prueba**

**Paso 1.** Agrega el siguiente código a la clase de pruebas unitarias AccountTest:

        [Test]

        public void TransferFundsWithInsufficientFunds()

        {

            Account source = new Account();

            source.Deposit(100m);

            Account destination = new Account();

            destination.Deposit(50m);

            TestDelegate invalidTransfer = () => source.TransferFunds(destination, 101m);

            Assert.Throws<InsufficientFundsException>(invalidTransfer);

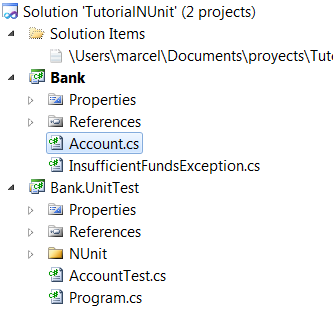
            Assert.That(source.Balance, Is.EqualTo(100m));

            Assert.That(destination.Balance, Is.EqualTo(50m));

        }

Como podrás darte cuenta, aún no existe la clase InsufficientFundsException, ni tampoco tiene el comportamiento deseado aún el método Account.TransferFunds, pero aun así, primero escribimos la prueba para el comportamiento deseado, luego haremos la implementación del comportamiento . A este tipo de desarrollo de código se le llama [Test Driven Development](http://en.wikipedia.org/wiki/Test-driven_development).

**Paso 2.** Lo primero que haremos, será agregar la excepción InsufficientFundsException, para esto debes crear una nueva clase en el proyecto Bank. Ya vimos como se crean las clases en Visual Studio 2010, ahora crea una nueva clase que se llame InsufficientFundsException en el proyecto **Bank**,de tal manera que te quede la siguiente estructura de archivos:



La clase InsufficientFundsException simplemente hereda de Exception, e implementa su constructor, no requiere pruebas:

namespace Bank

{

    using System;

    public class InsufficientFundsException : Exception

    {

        public InsufficientFundsException(string message)

            : base(message)

        {

        }

    }

}

**Paso 3.** Ahora debería compilar la prueba unitaria, pero igual sigue fallando, pues aún no se implementa el comportamiento en el método Account.TransferFunds, el método TransferFunds debe tener esta implementación:

        public void TransferFunds(Account destination, decimal amount)

        {

            if(this.Balance < amount)

            {

                throw new InsufficientFundsException("A ${0:0.00} transfer was requested, but the source account did not have enough balance.");

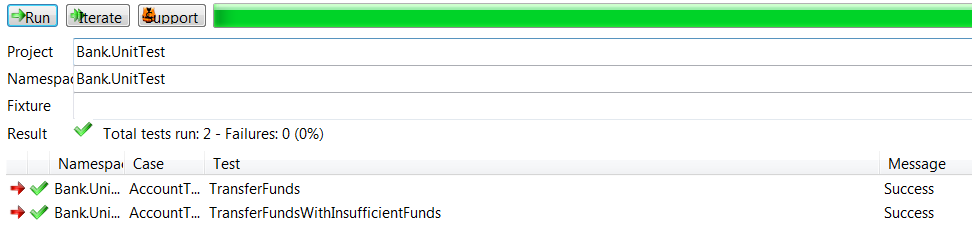
            }

            this.Balance -= amount;

            destination.Balance += amount;

        }

**Paso 4.** De nuevo ejecuta las pruebas unitarias del proyecto Bank.UnitTest, y te debería de marcar todo en verde, la prueba ha pasado:

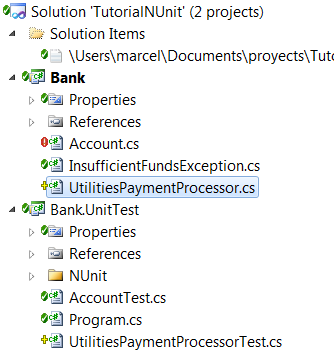


**2.4** Cómo hacer una prueba unitaria con NSubstitute

Digamos, que el banco en cuestión, el dueño del proyecto Bank, quiere agregar la capacidad de pagar y agregar facturas de servicios como agua, luz, cable, etcétera, directamente en el banco, por medio del código de la factura.

Entonces, dadas estas características, utilizando [TDD](http://en.wikipedia.org/wiki/Test-driven_development) primero escribiremos las pruebas para cada funcionalidad.

**Paso 1.** Lo primero que debes hacer es crear un nuevo conjunto de casos de prueba para la funcionalidad de pago de servicios, para esto, se deben crear una clase, a la que llamaremos: **UtilitiesPaymentProcesorTest** en el proyecto **Bank.UnitTest**, y la clase que implementará esta funcionalidad **UtilitiesPaymentProcessor** en el proyecto **Bank**. Al crear las clases, te debería quedar esta estructura de archivos:



**Paso 2.** Abre el archivo **UtilitiesPaymentProcessorTest.cs**, y agregamos los atributos de NUnit para que la considere una clase de pruebas:

namespace Bank.UnitTest

{

    using NUnit.Framework;

    [TestFixture]

    public class UtilitiesPaymentProcessorTest

    {

    }

}

**Paso 3.** Ahora debemos agregar e implementar uno de los casos de prueba, por lo menos debemos empezar con el caso de prueba para agregar facturas por pagar.

namespace Bank.UnitTest

{

    using NUnit.Framework;

    using System.Collections.Generic;

    using System.Reflection;

    [TestFixture]

    public class UtilitiesPaymentProcessorTest

    {

        [Test]

        public void TestIfItCanAddBill()

        {

            // Se inicializan los objetos a utilizar

            var utilitesAccount = new Account();

            var target = new UtilitiesPaymentProcessor(utilitesAccount);

            var fieldInfo = typeof(UtilitiesPaymentProcessor)

                                .GetField("bills", BindingFlags.NonPublic | BindingFlags.Instance);

            string billCode = "bill";

            int debt = 1;

            // Se agrega la factura

            target.AddBill(billCode, debt);

            // Se obtienen las facturas del objeto

            var bills = (IDictionary<string, decimal>)fieldInfo.GetValue(target);

            // Se verifica que efectivamente se haya agregado la factura

            Assert.That(bills.Keys, Contains.Item(billCode));

            Assert.That(bills[billCode], Is.EqualTo(debt));

        }

    }

}

Por el momento, este código ni siquiera compila, pues no existe el constructor ni el método para agregar facturas en **UtilitiesPaymentProcessor**. Agrega estos dos métodos en la clase **UtilitesPaymentProcessor**:

namespace Bank

{

    using System;

    public class UtilitiesPaymentProcessor

    {

        private Account utilitiesAccount;

        public UtilitiesPaymentProcessor(Account utilitiesAccount)

        {

            this.utilitiesAccount = utilitiesAccount;

        }

        public void AddBill(string billCode, decimal debt)

        {

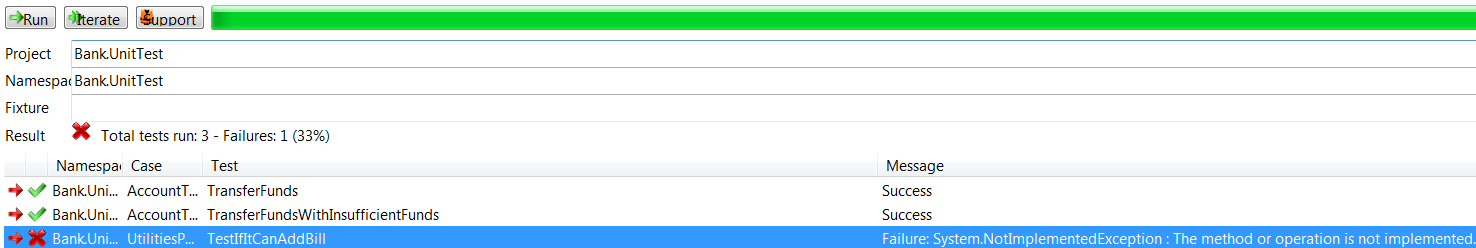
            throw new NotImplementedException();

        }

    }

}

Ahora sí debería compilar el código, pero la prueba no pasa, pues solamente tira una excepción:



Finalmente, implementamos el método **AddBill** en la clase UtilitiesPaymentProcessor:

namespace Bank

{

    using System;

    using System.Collections.Generic;

    public class UtilitiesPaymentProcessor

    {

        private Account utilitiesAccount;

        private IDictionary<string, decimal> bills = new Dictionary<string, decimal>();

        public UtilitiesPaymentProcessor(Account utilitiesAccount)

        {

            this.utilitiesAccount = utilitiesAccount;

        }

        public void AddBill(string billCode, decimal debt)

        {

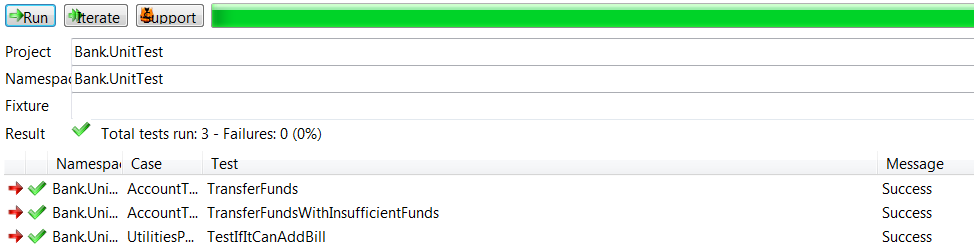
            this.bills.Add(billCode, debt);

        }

    }

}

Como especificamos en la prueba unitaria, se espera que la clase **UtilitiesPaymentProcessor**, tenga un diccionario con las facturas pendientes, que será donde se guarden las facturas y su saldo. Ahora, ejecutemos el caso de prueba para **TestifItCanAddBill**:



Todas las pruebas pasan de nuevo.

**Paso 4.** Pasemos al segundo caso de prueba, pagar la factura. Dado que una prueba unitaria no debe tener dependencias de otras unidades de código (otras clases), pues se convierte en una prueba frágil. No podemos probar código que dependa de los métodos de la clase **Account**, entonces ¿cómo se hace, sin tener que gastar cientos de líneas de código “[boilerplate](http://en.wikipedia.org/wiki/Boilerplate_code)” para ejecutar una simple prueba unitaria? Por medio de [mocks](http://en.wikipedia.org/wiki/Mock_object) y [stubs](http://en.wikipedia.org/wiki/Method_stub), no importa saber cuál es la diferencia entre estos, sino saber cuando se ocupa una forma de quitar dependencias a las pruebas, NUnit viene con una librería para hacer precisamente esto: NSubstitute y puedes descargar las librerías [aquí](http://github.com/downloads/nsubstitute/NSubstitute/NSubstitute-1.4.3.0.zip).

Un detalle sobre mocking, es que requieren que las clases implementen interfaces, o tengan el denominador virtual en su definición de método, por esto agregaremos este denominador:

namespace Bank

{

    using System;

    public class Account

    {

        public virtual void Deposit(decimal amount)

        {

            Balance += amount;

        }

        public virtual void Withdraw(decimal amount)

        {

            Balance -= amount;

        }

        public virtual void TransferFunds(Account destination, decimal amount)

        {

            if(this.Balance < amount)

            {

                throw new InsufficientFundsException("A ${0:0.00} transfer was requested, but the source account did not have enough balance.");

            }

            this.Balance -= amount;

            destination.Balance += amount;

        }

        public virtual decimal Balance

        {

            get;

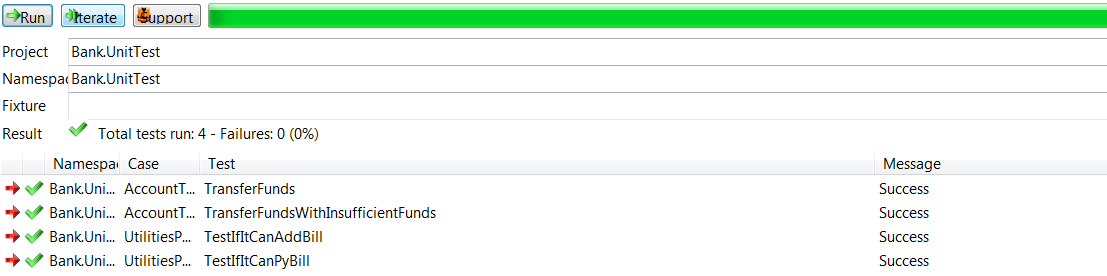
            private set;

        }

    }

}

Ahora ejecuta todas las pruebas unitarias de nuevo, para verificar que este cambio no haya afectado la funcionalidad previamente implementada.



Esta es precisamente una de las principales ventajas de tener pruebas unitarias exhaustivas, porque podemos verificar que cambios hecho al código no provoquen nuevos errores, esto no es factible hacer manualmente siquiera en un programa de tamaño mediano, pero para un proyecto con pruebas unitarias exhaustivas, esto no representa un reto, ni siquiera se considera problema, al contrario, se considera ventaja.

[Test]

public void TestIfItCanPayBill()

{

    // Se inicializan los mocks

    var utilitesAccount = Substitute.For<Account>();

    var payerAccount = Substitute.For<Account>();

    payerAccount.Balance.Returns(20);

    utilitesAccount.Balance.Returns(0);

    // Se inicializa el objeto a probar

    var target = new UtilitiesPaymentProcessor(utilitesAccount);

    string billCode = "bill";

    decimal debt = 10;

    decimal leftOver = 1;

    IDictionary<string, decimal> bills = GetBills(target);

    bills.Add(billCode, debt);

    // Se ejercita el código bajo prueba

    decimal actualLeftOver = target.PayBill(billCode, payerAccount, debt - leftOver);

    // Verificar que el método haya funcionado correctamente

    // Prueba de Caja negra

    Assert.That(bills[billCode], Is.EqualTo(actualLeftOver));

    // Prueba de Caja blanca o transparente

    // Se especifica que se debe transferir fondos del deudor al acreedor.

    payerAccount.Received().TransferFunds(utilitesAccount, debt - leftOver);

    utilitesAccount.DidNotReceive().TransferFunds(Arg.Any<Account>(), Arg.Any<decimal>());

}

Por el momento, no compila la prueba. Se agregó el método GetBills en la clase **UtilitiesPaymentProcessorTest**, para obtener la instancia **bills**del objeto **UtilitiesPaymentProcessor** para evitar duplicación de código, ya que esto mismo ya se hacía en la prueba anterior:

private static IDictionary<string, decimal> GetBills(UtilitiesPaymentProcessor target)

{

    var fieldInfo = typeof(UtilitiesPaymentProcessor)

                        .GetField("bills", BindingFlags.NonPublic | BindingFlags.Instance);

    var bills = (IDictionary<string, decimal>)fieldInfo.GetValue(target);

    return bills;

}

Ahora, se implementa la funcionalidad en el objeto **UtilitiesPaymentProcessor**:

public decimal PayBill(string billCode, Account payer, decimal payment)

{

    payer.TransferFunds(utilitiesAccount, payment);

    this.bills[billCode] -= payment;

    return this.bills[billCode];

}

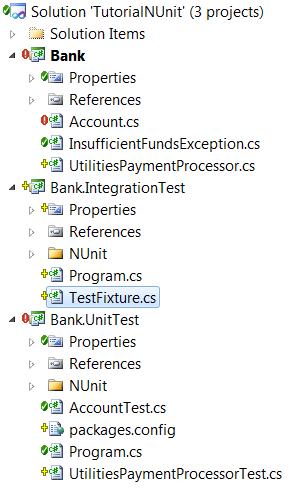
**Ejercicios:**

Tal vez te diste cuenta que el método **PayBill** tiene defectos: ¿Qué pasa si se desea pagar más de lo que se debe? ¿Qué pasa si el billCode no existe? ¿Qué pasa si el pago es por una cantidad negativa o 0? ¿Qué pasa si el parámetro payer es nulo? Escribe estos 4 casos de prueba y la implementación en código que los corrija.

**2.6 Cómo hacer una prueba de integración.**

Como se mencionó al principio, la prueba de integración verificará que la interacción entre los elementos sea correcta, y que provean el resultado deseado, un ejemplo de esto, sería verificar que el pago de facturas se haga correctamente cuando se utilicen las instancias no-mock de los objetos.

Crea otro proyecto de pruebas, igual al de pruebas unitarias, pero llámale **Bank.IntegrationTest**, la estructura resultante debería ser la siguiente:



Ahora modifica el nombre del archivo TestFixture a **UtilitiesPayment\_Account\_Test.cs** e introduce el código:

namespace Bank.IntegrationTest

{

    using NUnit.Framework;

    [TestFixture]

    public class UtilitiesPayment\_Account\_Test

    {

        [Test]

        public void TestAnAccountCanPayABillThroughPaymentProcessor()

        {

            // Arrange

            string billCode = "bill";

            int initialPayerBalance = 100;

            int initialDebt = 50;

            int payment = 40;

            int expectedDebtBalance = initialDebt - payment;

            int expectedPayerBalance = initialPayerBalance - payment;

            var utilitiesAccount = new Account();

            var payerAccount = new Account();

            payerAccount.Deposit(initialPayerBalance);

            var processor = new UtilitiesPaymentProcessor(utilitiesAccount);

            // Act

            processor.AddBill(billCode, initialDebt);

            decimal debtBalance = processor.PayBill(billCode, payerAccount, payment);

            // Assert

            Assert.That(debtBalance, Is.EqualTo(expectedDebtBalance));

            Assert.That(utilitiesAccount.Balance, Is.EqualTo(payment));

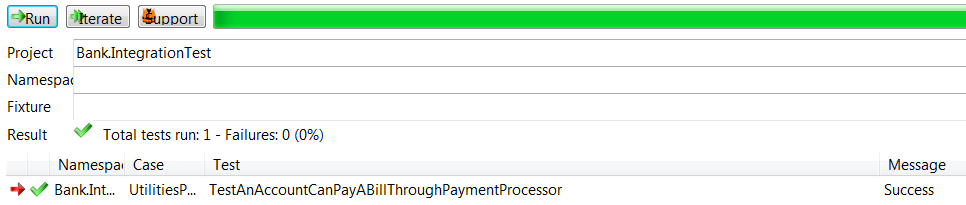
            Assert.That(payerAccount.Balance, Is.EqualTo(expectedPayerBalance));

        }

    }

}

Finalmente, ejecuta esta prueba, y deberías verla pasando adecuadamente:



**3. Conclusión**

Después de implementar pruebas unitarias y sacarles provecho, como una suite de pruebas de regresión, queda sobre entendido que las pruebas automatizadas unitarias y de integración son de gran valor en un proyecto de cualquier tamaño, donde importe la calidad y tiempo de entrega del software.

También observamos que TDD es una técnica de programación que, dada su naturaleza, provoca que creamos programas correctos y simples, pues cada línea de código escrita esta orientada a proveer una funcionalidad que se requiere en el momento, y no a algo que *podría* ocuparse en el futuro.

Ahora estás listo para salir al mundo a crear programas de alta calidad por medio de suites de pruebas unitarias e integración. Se recomienda que leas sobre cómo automatizar las pruebas de aceptación con herramientas como: [White](http://white.codeplex.com/), <Selenium,> <Robot>, [Ranorex](http://www.ranorex.com/) e [Ivonna.](http://ivonna.biz/)

**Bibliografía**

Hunt, A., & Thomas, D. (1999). *The Pragmatic Programmer.* Boston: Addison-Wesley.

Hunt, A., & thomas, D. (2003). *Pragmatic Unit Testing in Java with JUnit.* Cambridge: O'Reilly.

Matyas, S., Glover, A., & Duvall, P. M. (2007). *Continuous Integration: Improving Software Quality and Reducing Risk.* Upper Saddle River: Pearson Education.

Wikipedia, t. f. (17 de August de 2012). *Wikipedia - Software testing*. Recuperado el 17 de August de 2012, de Wikipedia: http://en.wikipedia.org/wiki/Software\_testing