UNIT TESTING

WORKSHOP

Contenido

[Definición de Unit Testing, ¿Qué es? ¿Y para qué sirve? 3](#_Toc461702486)

[Detalle del proyecto a agregar. 4](#_Toc461702487)

[Agregar el primer archivo de pruebas 5](#_Toc461702488)

[Decoradores 6](#_Toc461702489)

[Estructura (AAA) 6](#_Toc461702490)

[Best Practices 7](#_Toc461702491)

[TDD (Test-Driven Develpoment) 9](#_Toc461702492)

[¿Qué es TDD? 9](#_Toc461702493)

[Pasos TFD 9](#_Toc461702494)

[Práctica: Test Simples 10](#_Toc461702495)

[Implementación de Mock/Mocking Frameworks 10](#_Toc461702496)

[¿Por qué es útil? 10](#_Toc461702497)

[¿Qué es un mocking framework? 11](#_Toc461702498)

[Top 5 Mock Framework. 11](#_Toc461702499)

[Summary: 11](#_Toc461702500)

[Entonces, ¿Qué significa Mock? 11](#_Toc461702501)

[Contexto de pruebas (Context Pattern) 12](#_Toc461702502)

[Testing Context in action 13](#_Toc461702503)

[Práctica: Test Complejos 14](#_Toc461702504)

[Code Coverage 14](#_Toc461702505)

[Interpretar los resultados 15](#_Toc461702506)

[Reporte en bloques o líneas 15](#_Toc461702507)

[Utilizar los resultados de code coverage 16](#_Toc461702508)

[Excluyendo elementos del Code Coverage Result 16](#_Toc461702509)

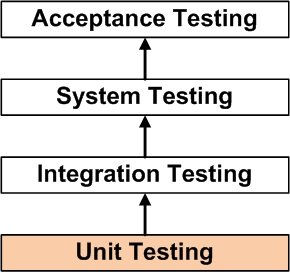
[Referencias 17](#_Toc461702510)

# Definición de Unit Testing, ¿Qué es? ¿Y para qué sirve?

Un unit test es un método que prueba una unidad de código. Al hablar de una unidad de código nos referimos a un requerimiento. Muchos desarrolladores tienen su propio concepto de lo que es una prueba unitaria; sin embargo, la gran mayoría coincide en que una prueba unitaria tiene las siguientes características:

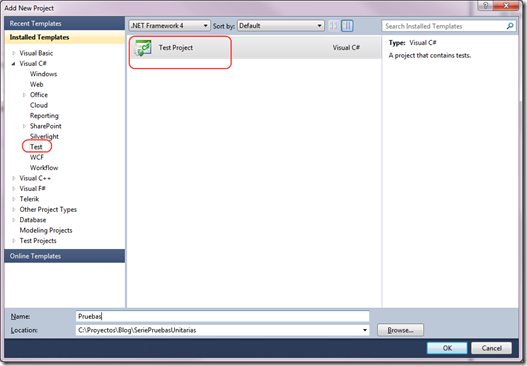
* Prueba solamente pequeñas cantidades de código: Solamente prueba el código del requerimiento específico.
* Se aísla de otro código y de otros desarrolladores: El unit test prueba exclusivamente el código relacionado con el requerimiento y no interfiere con el trabajo hecho por otros desarrolladores.
* Solamente se prueban los endpoints públicos: Esto principalmente porque los disparadores de los métodos privados son métodos públicos por lo tanto se abarca el código de los métodos privados dentro de las pruebas.
* Los resultados son automatizados: Cuando ejecutamos las pruebas lo podemos hacer de forma individual o de forma grupal. Estas pruebas las hace el motor de prueba y los resultados de los mismos deben de ser precisos con respecto a cada prueba unitaria desarrollada
* Repetible y predecible: No importa el orden y las veces que se repita la prueba, el resultado siempre debe de ser el mismo.
* Son rápidos de desarrollar: Contrariamente a lo que piensan los desarrolladores –> que el desarrollo de pruebas unitarias quita tiempo – los unit test por lo general deben de ser simples y rápidos de desarrollar. Difícilmente una prueba unitaria deba de tomar más de cinco minutos en su desarrollo.

El desarrollador es el encargado de realizar un completo y detallado UT de sus componentes, para poder reducir y mitigar futuros errores en las etapas posteriores, principalmente en la Integración donde su componente va a ser implementado junto con otras funcionalidades:

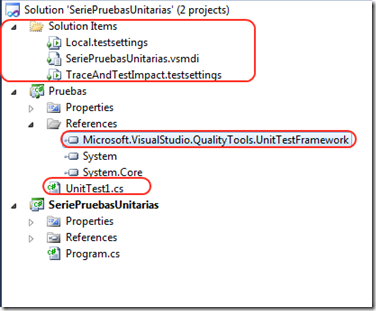


# Detalle del proyecto a agregar.

El primer paso es agregar un proyecto a la solución del tipo Test –> vamos a utilizar MSTest para hacer las pruebas unitarias en este ejemplo [Observación: Yo utilizo Visual Studio Ultimate por lo tanto algunas opciones de testing pueden no aparecer en otras versiones menores o pueden estar en otro orden].

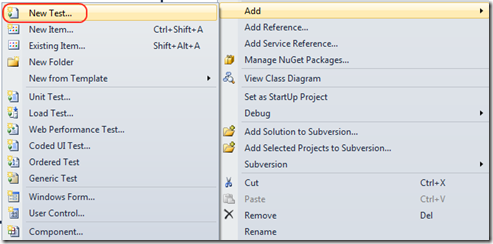


Como vemos en la siguiente figura, se va a crear un proyecto de testing que tiene una referencia directa al componente – dll – en donde están todas las librerías para hacer pruebas unitarias con el MSTest. Además, se crea un archivo inicial de testing y un grupo de tres “Solution Items” desde donde vamos a poder configurar y administrar las funcionalidades de nuestros tests.

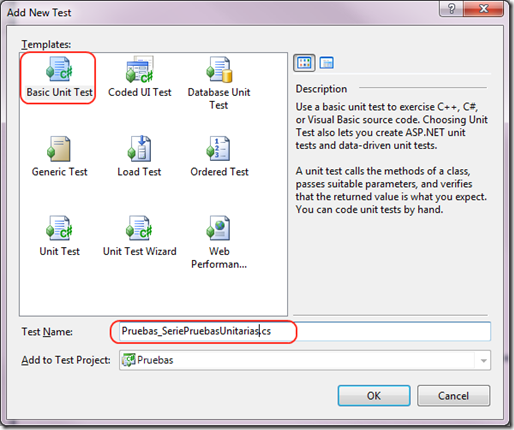


# Agregar el primer archivo de pruebas

Ahora procedemos a borrar el archivo UnitTest1.cs y vamos a crear uno desde cero. Luego de borrar el archivo, procedemos a agregar una referencia al proyecto que queremos probar. Luego damos botón derecho sobre el proyecto de testing y seleccionamos agregar nuevo test.



Ahora procedemos a seleccionar el tipo de unit test que vamos a crear, en nuestro caso una prueba de unidad básica.



# Decoradores

Para que el MSTest pueda identificar las clases y métodos de pruebas involucrados en los test, se deben completar mediante el uso de decoradores (Decorators). Estos decoradores definen cada uno de los componentes dentro de un archivo de pruebas y pueden cambiar en función del programa que se use para realizar los test (MSTest / NUnit / Etc). Los principales son:

* [**TestClass**]: Se decora sobre el nombre de la clase y define que la clase se utilizará para los unit test.
* [**TestInitialize**]: Se decora sobre el nombre de la clase que inicializar objetos, configuraciones, etc. Que serán usados en las pruebas posteriores.
* [**TestMethod**]: Se decora sobre cada uno de los métodos a ejecutarse dentro de la clase y serán evaluados como resultados al momento de correr.
* [**ExcludeFromCoverage**]: Este decorador, se utiliza para excluir del coverage algún método que no se desea ser tenido en cuenta (en detalle en el apartado correspondiente).

# Estructura (AAA)

* **Arrange**: Realiza toda configuración necesaria para ejecutar el test. Incluye, cualquier inicialización de dependencias, mocks y data necesaria para que la ejecución.
* **Act**: Invoca el código a probar.
* **Assert**: Especifica el “PASS CRITERIA” para el test en particular, el cual fallará si estas especificaciones fallan.

Este pattern es de fácil lectura y consistente. Se refiere a una manera fácil de identificar cada parte del test y crear test para otros escenarios a partir de la prueba principal.

En nuestros métodos a probar deseamos obtener los resultados esperados. Usualmente, el resultado puede ser el determinado en este escenario, el cual queda expuesto al ejecutarse según:

* Valores devuelto desde el método
* Valores expuestos a través de campos de un objeto.
* Valores devuelto a través de otros métodos o propiedades del objeto.
* Valores que provienen desde afuera del objeto, por ejemplo static o shared.

Las pruebas que cumplen con estos tipos, se denominan, pruebas basadas en estado (state-based test). Dependiendo del test framework utilizado se puede encontrar Assert API’s que permitirán comparar estos valores devueltos con los esperados.

Por ejemplo, para el MSTest Framework, **Assert.AreEqual** compara el valor devuelto con el esperado por el método invocado dentro del TestMethod.

|  |
| --- |
| [TestMethod] |

|  |
| --- |
| public void GetCount\_ItemCountIsZero\_NoNewMessages() |

|  |
| --- |
| { |

|  |
| --- |
| //Arrange |

|  |
| --- |
| Mailbox mailbox = new Mailbox(); |

|  |
| --- |
|  |

|  |
| --- |
| //Act |

|  |
| --- |
| var result = mailbox.GetCount(0); |

|  |
| --- |
|  |

|  |
| --- |
| //Assert |

|  |
| --- |
| Assert.AreEqual("No new messages.", result); |

|  |
| --- |
| } |

Adicionalmente a los state-based tests, existen los interaction-based test, los cuales son realizados cuando por ejemplo se necesita devolver/comparar/procesar respuestas como resultados de algún 3rd-party service, 3rd-party dependecy, files, database, system objects y/o cualquier otro que no sea propia de nuestra solución.

Para estos tipos de pruebas se implementan los denominados Mocking Framework, estos se configuran para realizar un Fake e Insolation de los métodos invocados. Estos frameworks pueden encontrarse una amplia variedad en internet según cada una de las necesidades.

# Best Practices

1. [**Always Write Isolated Test Cases**](http://www.sw-engineering-candies.com/blog-1/unit-testing-best-practices#TOC-Always-Write-Isolated-Test-Cases)

El orden de ejecución han de ser independientes entre test-cases. Esto nos dará la chance de reordenar los test-cases en paquetes de pruebas (short-, long-running) y en caso de ser necesario re-testear en test cases mas simples.

1. [**Test One Thing Only in One Test Case**](http://www.sw-engineering-candies.com/blog-1/unit-testing-best-practices#TOC-Test-One-Thing-Only-in-One-Test-Case)

Cada test deberá focalizarse solamente en un aspecto de un método aislado y/o clase. Para las pruebas de integración (integration-tests) el objetivo podrá ser difícil de conseguir, pero también en este punto tratar de hacer foco en aspectos simples ayuda. Esto significa que usualmente deberá haber UN ASSERT por Test Case. Más de un Assert siempre son útiles cuando NO se necesita centrarse en un aspecto o funcionalidad a testear.

1. [**Use a Single Assert Method per Test Case**](http://www.sw-engineering-candies.com/blog-1/unit-testing-best-practices#TOC-Use-a-Single-Assert-Method-per-Test-Case)

No realice tests a muchas y/o deferentes cosas al mismo tiempo. Si solamente usa UN Assert, es todo mucho más fácil. Esto aumenta el número de test cases y cada single test es mejor de mantener. Si uno o más test fallan, se podrá encontrar la causa del fallo de una manera fácil y directa.

1. [**Use a Naming Convention for Test Cases**](http://www.sw-engineering-candies.com/blog-1/unit-testing-best-practices#TOC-Use-a-Naming-Convention-for-Test-Cases)

Un limpio y ordenado naming-convention como: *<Method-Name Under Test>\_<Scenario>\_<Expected-Outcome>* ayuda a prevenir comentarios en exceso y mejora la mantenibilidad del mismo, además en caso que el test falle sabrá rápidamente que funcionalidad se ha sido.

1. [**Use the Arrange-Act-Assert Style or Given-When-Then Style**](http://www.sw-engineering-candies.com/blog-1/unit-testing-best-practices#TOC-Use-the-Arrange-Act-Assert-Style-or-Given-When-Then-Style-)

Use secciones separadas con el patrón (AAA), para ayudar a mejorar la mantenibilidad y use comentarios para marcar cada sección. Como alternativa se podrá utilizar el estilo (“GIVEN”, “WHEN” y “THEN”).

Esta práctica de mejor estructura a los unit tests.

1. [**Avoid the Expected Exception Tests**](http://www.sw-engineering-candies.com/blog-1/unit-testing-best-practices#TOC-Avoid-the-Expected-Exception-Tests)

Quizás @Test(expected = ArithmeticException.class) no testea lo que enuncia. Hay veces que se lanzan excepciones (Exceptions) en algún lugar del código donde no era esperado, esto podría ser tomado como correcto en el test y este no fallar, cuando la excepción pudo haber sido una casuística no contemplada.

1. [**Structure All Test Cases**](http://www.sw-engineering-candies.com/blog-1/unit-testing-best-practices#TOC-Structure-All-Test-Cases)

Como Short-/Long-Running, Integration-/Unit-test, pero no utilice el test manager para controlar el orden de ejecución. Cuando se tiene cientos de test-cases en muchos casos no se desea esperar demasiado tiempo hasta que finalicen todas las pruebas. Especialmente las integration-test pueden llegar a ser lentas por demás, entonces es preferible separarlas de las unit-test.

1. [**Use Descriptive Messages in Assert Methods**](http://www.sw-engineering-candies.com/blog-1/unit-testing-best-practices#TOC-Use-Descriptive-Messages-in-Assert-Methods)

Describa el POR QUE? y no el QUE?, o sea no use Assert.assertEquals(a, b, “must be equal”). Esto ayuda a prevenir demasiados comentarios en los test-cases y aumenta su mentenibilidad.

1. [**Measure Code Coverage to Find Missing Test Cases**](http://www.sw-engineering-candies.com/blog-1/unit-testing-best-practices#TOC-Measure-Code-Coverage-to-Find-Missing-Test-Cases)

El mejor indicador para saber que no está siendo testeado, pero no asegura que el código funciona como se espera.

1. [**Don't Forget to Refactor the Test Code**](http://www.sw-engineering-candies.com/blog-1/unit-testing-best-practices#TOC-Don-t-Forget-to-Refactor-the-Test-Code)

También mantenga el test-code (una vez realizada la refactorización del código a testear).

1. [**Limit Use of Mocks**](http://www.sw-engineering-candies.com/blog-1/unit-testing-best-practices#TOC-Limit-Use-of-Mocks)

En algunos casos es absolutamente necesario, pero con un buen diseño debería ser suficiente.

1. [**Use Parameterized Tests**](http://www.sw-engineering-candies.com/blog-1/unit-testing-best-practices#TOC-Use-Parameterized-Tests)

Estos pueden ayudar a evitar la duplicidad (o multiplicidad) del código. Lo ideal sería que business provea la test-data a utilizar

# TDD (Test-Driven Develpoment)

## ¿Qué es TDD?

Es una evolucionada técnica de desarrollo, la cual combina la metodología “test-first” en la cual primero se escriben la/s pruebas antes de escribir y lograr el código final que será testeado y re-factorizado si es necesario. Cuál es el objetivo principal de TDD?: Es lograr la ESPECIFICACION y NO la VALIDACION. En otras palabas, es una forma de pensar y lograr los requerimientos o diseños propuestos, antes de escribir el código funcional. O sea se piensa/prueba en base a diseños. Para poder realizar TDD se implican técnicas “agile requirements” y “agile design”.

## Pasos TFD

Los pasos para realizar TFD (Test-First Develoment) se pueden ver en el siguiente diagrama de actividad:



Se puede ver que el primer es rápido: ADD A TEST. Básicamente es hacer que el la prueba falle por el código que se testea. Se pueden tener varias pruebas que cubren los diferentes aspectos del requerimiento, se pueden correr todas o focalizarse solo en algunas según la importancia que se le haya dado (score) a esa tarea.

Si la prueba falla, el paso siguiente es UPDATE CODE. Esta es la parte de Refactorización del método que se está probando, se deberá completar el mismo hasta que la prueba pase con las próximas ejecuciones y tratando de cubrir lo solicitado en el requerimiento/diseño.

Finalmente Ejecutar el set completo de pruebas. Si estas fallan habrá que actualizar el código a probar y re-testear. Si estas pruebas pasan OK, entonces como paso final habrá que verificar que no haya duplicación de pruebas de las distintas tareas del requerimiento.

Si todo sale Ok, entonces se prosigue al requerimiento siguiente.

TDD va en oposición respecto al método tradicional de desarrollo. Cuando primero se implementa la nueva funcionalidad, la primera pregunta que debe hacerse uno es si el diseño planteado/existente es el mejor diseño posible, el cual me da todas las herramientas necesarias para poder implementar esa nueva funcionalidad. Si esto es así, entonces se deberá proceder con TFD. Si no, habría que re-factorizar el diseño en las secciones que afecten a la nueva funcionalidad, lo cual me permita agregar y adaptarla de la más rápida y fácil posible. Como resultado siempre se estaría mejorar la calidad del diseño y por lo tanto facilitando el trabajo a futuro.

A diferencia del método tradicional, donde primero se codifica todo lo funcional y luego se realizan las pruebas completas, si ya está todo implementado aún se deberá escribir los test adicionales.

Un desarrollador que adopte la metodología TDD, siempre rechazará codificar la nueva funcionalidad primero, antes de tener la primer prueba codificada; aun así si esta fallara (que seguro lo hará) por no tener código presente en la funcionalidad testeada.

## Resumen:

TDD = Refactoring + TFD.

TDD es una técnica para diseñar software que se centra en tres pilares fundamentales:

* La implementación de las funciones justas que el cliente necesita y no más.
* La minimización del número de defectos que llegan al software en fase de producción.
* La producción de software modular, altamente reutilizable y preparado para el cambio.

La esencia de TDD es sencilla pero ponerla en práctica correctamente es cuestión de entrenamiento, como tantas otras cosas.

El algoritmo TDD sólo tiene tres pasos:

1. Escribir la especificación del requisito (el ejemplo, el test).
2. Implementar el código según dicho ejemplo.
3. Re-factorizar para eliminar duplicidad y hacer mejoras.

# Práctica: Test Simples

{Completar con los diversos ejemplos y proyectos}

* Sumatoria de números
  + Resultado Correcto
  + Resultado Incorrecto
* Manejo de exceptions (Assert.Fail) y como personalizar el error devuelto.

# Implementación de Mock/Mocking Frameworks

## ¿Por qué es útil?

Cuando uno planea realizar tests a gran escala, es muy difícil de manejar y aumentar la velocidad cuando se requieren múltiples interacciones con servicios, databases, files, etc. El Unit Testing para ese tipo de grandes aplicaciones pueden ser realizadas de forma manual, pero llevaría mucho tiempo realizarlas, implicando esto pérdidas de tiempo de trabajo/dinero. Entonces, a razón de estos conceptos, surge la función de Mock o Mocking Frameworks convirtiéndose en la opción ideal para estos casos.

## ¿Qué es un mocking framework?

Mocking Framework es usado para crear objetos que servirán de reemplazo, como Stubs, Fakes y Mocks. Son usados para aislar cada dependencia y ayuda a los desarrolladores a concretar test más performantes en un conciso, rápido y una óptima manera.

Crear manualmente Mock Object es complicado y lleva tiempo, entonces para aumentar la productividad se puede recurrir a la generación automática de mocks object usando el Mocking Framework. Luego el developer podría crear su propio unit test usando: NUnit, MbUnit, MSTest, xUnit, etc unit test framework que luego pueden testearse en el Mocking Framework.

## Top 5 Mock Framework.

1. **NSubstitute:** <http://nsubstitute.github.io/>
2. **Rhino Mocks:** <http://www.hibernatingrhinos.com/oss/rhino-mocks>
3. **Moq:** <https://code.google.com/p/moq/downloads/list>
4. **FakeItEasy:** <https://github.com/FakeItEasy/FakeItEasy/>
5. **NMock3:** <http://nmock3.codeplex.com/releases/view/108516>

### Summary:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **NSubstitute** | **Rhino Mocks** | **Moq** | **FakeItEasy** | **NMock3** |
| **Free Availability** | mocking framework | mocking framework | mocking framework | mocking framework | mocking framework |
| **License With** | BSD License | BSD License | BSD-2-Clause License |  | Apache License 2.0 |
| **Latest Version** | NSubstitute 1.7.2 | Rhino Mocks 4.0.0-alpha3 | Moq 4.0.10827 | FakeItEasy 1.22.0 | NMock3 v3.5.44 |
| **Source Code Availability** | mocking framework | mocking framework | mocking framework | mocking framework | mocking framework |

El framework que se utilizara para este workshop es: **Moq**

## Entonces, ¿Qué significa Mock?

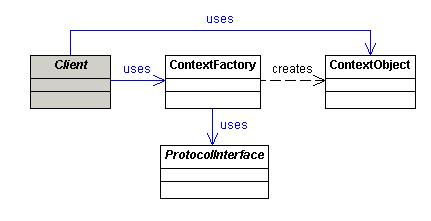
Mock, quiere decir que se está planeando probar diferentes procesos sin el ingreso de data de manera estándar, reemplazando estos con mock/dummy/fake data como entrada y configurando la salida/respuestas, según las necesidades de las pruebas.

Mocking, es mayormente usado en unit testing. Mocking es crear objetos tipo FAKE u objetos que simulan el comportamiento real de los objetos “mockeados”. Con el mocking, se puede probar por separado cada parte del código, no es necesario esperar que otros miembros de desarrollo/proyecto terminen sus componentes, para que se puedan probar determinadas funcionalidades. Hay tres objetos de reemplazo, que son usados para reemplazar las dependencias de la “unidad (unit)” del componente testeado, esos objetos son: Stubs, Fakes y Mocks.

* **Fake**: **La clase que implementa una interfaz pero contiene datos fijos (fixed data) y no posee lógica. Simplemente retorna “buena” o “mala” data, dependiendo de la implementación.**
* **Mock**: **Una clase que implementa una interfaz y permite dinámicamente setear valores de retorno/lanzar excepciones, desde un método en particular y también tiene la capacidad de chequear si algún método en particular ha sido llamado o no.**
* **Stub**: **Similar al Mock, exepto que este no tiene la capacidad de verificar que métodos han sido llamado o no.**

# Contexto de pruebas (Context Pattern)

La idea es crear en tu Proyecto de Prueba una clase privada llamada TestContext (por ejemplo) y poner todos los Mocks/Instancias necesarias para las pruebas a efectuar para clase a testear y creando una instancia de esa clase esta implementará esos Mocks y por lo tanto expondrá todos las propiedades públicas en TestContext.



Si estamos probando la clase WebPageDownloader que necesita dos o más entidades para funcionar (**IUrlPermissioner** y **IUrlRetriever) entonces creamos la clase TestContext como:**

private class TestContext

{

public Mock<IUrlPermissioner> PermissionerMock;

public Mock<IUrlRetriever> UrlRetrieverMock;

public WebPageDownloader Downloader;

}

Vemos que se expondrán los dependencias mockeadas en el context (para este caso se usará el framework Moq, pero se podría utilizar cualquier otro), esto es muy importante porque luego en nuestras pruebas podremos configurar setup, definir valores esperados, lanzar exceptions, etc a esos Mocks que será utilizados dentro de la clase de prueba (test class).

Paso siguiente creamos un método privado (llamado CreateContext()), que crea la instancia del TestContext, inicializará con todos los Mocks necesarios para las pruebas y las instancias de la clase a testear, retornándola (esto evita tener que crear una para cada prueba a realizar):

private TestContext CreateContext()

{

var ctxt = new TestContext()

{

PermissionerMock = new Mock<IUrlPermissioner>(),

UrlRetrieverMock = new Mock<IUrlRetriever>(),

};

ctxt.Downloader = new WebPageDownloader(ctxt.PermissionerMock.Object, ctxt.UrlRetrieverMock.Object);

return ctxt;

}

Como puede verse, se crearon todo los mocks que necesita la clase a testear, luego se instancia pasando las “instancias Mockeadas” mediante el constructor y luego guarda los mocks y clase involucrada en el contexto para que pueda ser reutilizada.

## Testing Context in action

Desde ahora en cada test se llamará al método CreateContext para obtener una instancia de TestContext y que pueda ser utilizada para probar la clase.

Lo bueno de esto es que en el TestContext tenemos todos los Mocks de las dependencias utilizadas, para crear la clase testeada, entonces esta se podrá utilizar sin inconvenientes, mockeando algún/os método específico según cada test a efectuar, también se podría verificar para cada test, si los mocks fueron utilizados de manera correcta, comprobando si fueron invocados desde la clase de test.

Ejemplo de test de una clase, que verifica si esta clase lanza una exception de URI que no está permitida por IUrlPermissioner:

[Test]

[ExpectedException(typeof(SecurityException))]

public void Download\_WhenNotPermitted\_Throws()

{

var ctxt = CreateContext();

var badUrl = "http://www.SomeBadUrl";

// arrange

ctxt.PermissionerMock.Setup(a => a.IsUrlAllowed(badUrl)).Returns(false);

// act

ctxt.Downloader.Download(badUrl);

// assert we don't need to do since we have ExpectedException attribute

}

Como se puede ver en el test anterior, llamamos a CreateContext primero para generar un nevo Context, luego usamos los mocks del context configurados según nuestras necesidades y los valores de llamadas y respuesta, finalmente invocamos el método de la clase que estamos testeando y obtenemos la exception esperada (mediante el decorador ExpectedException).

Ahora un escenario un más complejo, donde necesitamos realizar múltiples Setup de interacciones, mockeadas con nuestra clase y para estos obtener el resultado esperado, validando luego que sea el correcto:

[Test]

public void Download\_WhenAllowed\_ShouldReturnWebpageGotViaRetriever()

{

var ctxt = CreateContext();

var OkUrl = "http://www.SomeOkUrl";

var OkUrlContent = "Some Web Page";

// arrange

ctxt.PermissionerMock.Setup(a => a.IsUrlAllowed(OkUrl)).Returns(true);

ctxt.UrlRetrieverMock.Setup(a => a.Retrieve(OkUrl)).Returns(OkUrlContent);

// act

var result = ctxt.Downloader.Download(OkUrl);

// assert

Assert.AreEqual(OkUrlContent, result);

}

Otro punto positivo en esta prueba es que si más adelante se re-factoriza la clase a probar y/o elimina dependencias, fácilmente cambiando el TestContext y el método CreateContext en consecuencia, no fallará ninguno de los test method por estos cambios.

# Práctica: Test Complejos

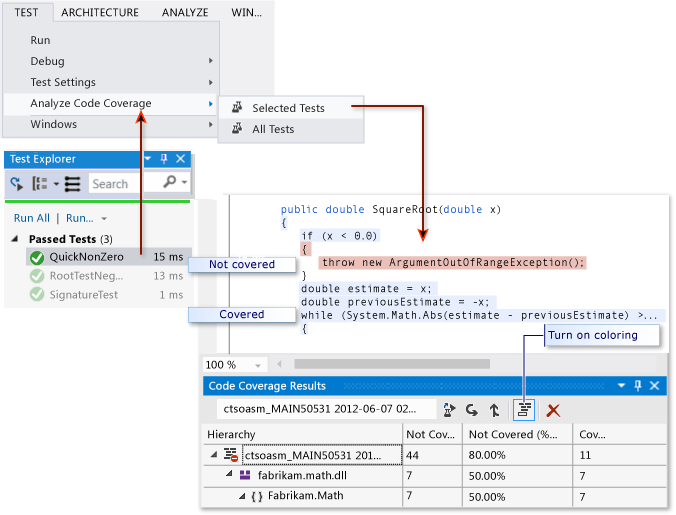
{Completar con los diversos ejemplos y proyectos}

* Instalación de Moq desde visualstudio gallery
* Implementación Moq
* Mocking en context
* Mocking de Variables de Session
* Mocking de llamados a servicios

# Code Coverage

Para determinar qué proporción del código dentro del proyecto es actualmente testeado/cubierto en la prueba, se puede usar el Code Coverage del Visual Studio. Para anticiparse a futuros defectos, los test deberán cubrir la mayor proporción del código a testear.

El code coverage es una opción cuando se ejecutan test utilizando Test Explorer. La tabla de resultado que se muestra, exhibe el porcentaje de código que se cubrirá en cada assembly, clase y método. Adicionalmente el Source Editor, muestra que porciones de código han sido testeadas.



## Interpretar los resultados

Si los resultados/valores devuelto son bajos, se deberá investigar cual/cuales partes/secciones del código no están siendo alcanzadas o cubiertas en las pruebas realizadas; por tal caso el equipo de desarrollo deberá implementar el código necesario para cumplirlas y/o incrementarlas. Usualmente los requerimientos de coverage rondan el 80% como mínimo. En algunas situaciones, un coverage menor podría ser aceptable (usualmente cuando el código generado proviene de un standard template).

## [Reporte en bloques o](javascript:void(0)) líneas

El code coverage es contado en blocks (bloques). Un bloque es una pieza de código con exactamente un punto de entrada y un punto de salida. Si el flow del programa pasa a través de un bloque durante una ejecución de pruebas, ese bloque es contado como “cubierto/alcanzado”. El número de veces que el bloque de código es usado, no infiere sobre los resultados.

También podrías obtener los resultados mostrados en términos de líneas, eligiendo **Add/Remove Columns en el header de la tabla de resultados. Si la ejecución de la prueba alcanza (cubre) todas y cada una de las líneas de un bloque de códigos, esta es contada como una línea. Donde la línea contiene algunos bloques de código que son alcanzados y otros que no, estos serán contados como parciales.**

**Algunos usuarios prefieren contar líneas porque los porcentajes son más aproximados al tamaño del fragmento que se está revisando en el código. Un bloque grande de cálculos podría contar como un único bloque aún si ocupa muchas líneas.**

## Utilizar los resultados de code coverage

La ventana de resultados usualmente muestra el resultado de la ultima ejecución. Este resultado podrúa variar si se modifica la test data o si solamente se ejecuta alguna de las pruebas por vez.

Esta ventana también puede ser utilizada para mostrar los resultados anteriores, o resultados provenientes de otros equipos.

Se puede realizar merge de diferentes resultados de ejecuciones, por ejemplo diferentes resultados por usar diferente test data.

* Para ver resultados anteriores, seleccionarlo desde el dropdown. El menú muestra una lista temporal que se limpia cada vez que se abre una nueva solución.
* Para ver resultados de una sesión anterior, elija “**Import Code Coverage Results”, elija la carpeta dentro de la solución que contiene los diferentes archivos *.coverage*.**

**El resaltado de color que muestra como resultados, podría ser diferente o no coincidir con lo que se muestra, si el código cambió desde que se generó el archivo .coverage.**

* Para hacer los resultados más legibles o interpretables en forma de texto, elija “**Export Code Coverage Results**”. Esto generará un archivo .coveragexml el cual podría ser procesado mediante el uso de otras herramientas. Y poder enviarse por mail a otros usuarios.

## Excluyendo elementos del Code Coverage Result

Puede darse el caso en los que se desea excluir elementos específicos del código del coverage, por ejemplo si el código es generado desde un text template. Agregue el atributo (decorator) **[ExcludeFromCodeCoverage]** el cual podrá implementarse agregando la referencia: **System.Diagnostics.CodeAnalysis** para cualquiera de los siguientes elementos:

* + Clases
  + Struct
  + Method
  + Property
  + Property setter o getter
  + Event

Nótese que excluyendo una clase no excluye a su clase derivada.

using System.Diagnostics.CodeAnalysis;

...

public class ExampleClass1

{

**[ExcludeFromCodeCoverage]**

void ExampleMethod() {...}

**[ExcludeFromCodeCoverage]** // exclude property

int ExampleProperty1

{ get {...} set{...}}

int ExampleProperty2

{

get

{

...

}

**[ExcludeFromCodeCoverage]** // exclude setter

set

{

...

}

}

}

**[ExcludeFromCodeCoverage]**

class ExampleClass2 { ... }

# Referencias

<https://msdn.microsoft.com/es-es/library/jj130731.aspx>

<https://www.typemock.com/unit-test-patterns-for-net/>

<http://stackoverflow.com/questions/15367568/c-sharp-how-to-properly-unit-test-a-class-that-follows-a-decorator-pattern>

<http://www.sw-engineering-candies.com/blog-1/unit-testing-best-practices>

<http://www.coderewind.com/2014/07/top-5-mocking-frameworks-net-developers/>

<https://www.nuget.org/packages/Moq/>

<http://stephenwalther.com/archive/2008/06/12/tdd-introduction-to-moq>

<https://dzone.com/articles/introducing-unit-testing>

<https://msdn.microsoft.com/en-us/library/ff650441.aspx>

<https://msdn.microsoft.com/en-us/library/dd537628.aspx>

<http://agiledata.org/essays/tdd.html>

<http://librosweb.es/libro/tdd/capitulo_2.html>

<http://pjcj.net/testing_and_code_coverage/paper.html>