Capítulo 3: Migrando a Microsoft Fakes

Traducción por Juan María Laó Ramos

* 1. 

Twitter: @juanlao

Linkedin: [es.linkedin.com/in/juanlao/](http://es.linkedin.com/in/juanlao/" \o "http://es.linkedin.com/in/juanlao/" \t "_parent)

Blog: <http://speakingin.net/>

Esperamos que algunos de los que estáis leyendo esta guía tengáis alguna experiencia con Microsoft Moles [(http://research.microsoft.com/en-us/projects/moles/)](http://research.microsoft.com/en-us/projects/moles/) o con cualquier otro framework de aislamiento comercial u open source. En este capítulo, veremos algunos de los pasos y problemas que nos podemos encontrar cuando migremos a Microsoft Fakes. Recordad que podéis usar Microsoft Fakes con otros frameworks y podéis migrar a vuestro propio ritmo.

## Migrando de Moles a Microsoft Fakes

Moles es el proyecto de Microsoft Research en el que se basa Microsoft Fakes, por lo tanto, tiene muchas similitudes en la sintaxis. Sin embargo, no hay un mecanismo automático para convertir un proyecto que use Moles a otro que use Microsoft Fakes. Con la publicación de Microsoft Fakes, Moles se ha quedado atrasado y es recomendable que los proyectos basados en Moles se migren a Microsoft Fakes cuando sea posible.

La razón principal de esta recomendación, a parte del soporte, es que cuando instalamos Visual Studio 2012 se instala .NET 4.5, y esto es un problema para Moles porque Moles intenta generar stubs o moles para tipos que sólo existen en .NET 4.5. El propio Moles está escrito en .NET 4 y esto genera errores. El único “workaround” es usar filtros en el archivo .moles para evitar la carga de esos tipos (y los tipos dependientes). Este proceso está detallado en el sitio de Microsoft Research ([http://research.microsoft.com/en-us/projects/moles/molesdev11.aspx)](http://research.microsoft.com/en-us/projects/moles/molesdev11.aspx). Como es un proceso potencialmente complejo y tendiente a errores, os recomendamos que no intentéis ejecutar Moles y Fakes a la vez.

Cualquier migración de Moles a Fakes requiere cambios en el código; sin embargo, dado que las bases de Microsoft Fakes es Moles, dichos cambios no son muchos.

|  |
| --- |
| Microsoft Fakes no reemplaza a PEX([http://research.microsoft.com/en-us/projects/pex/)](http://research.microsoft.com/en-us/projects/pex/) y no ofrece la generación automática de test unitarios que si ofrece PEX |

## Cambiando referencias

El primer paso en cualquier migración de Moles a Fakes es eliminar las referencias a los assemblies y borrar los archivos .moles. El siguiente paso es generar un nuevo assembly de Fake. Esto añadirá las referencias necesarias y los archivos .fakes al proyecto de test.

## La forma de hacer fakes

La principal diferencia entre Moles y Shims es la forma en la que se define la operación de fake. En Moles, se coloca un atributo **HostType** en el método de test:

* 1. [TestMethod] [HostType("Moles")] public void TestMethod()
  2. {
  3. //...
  4. }

En Microsoft Fakes, se hace con una sentencia **using** que contiene un objeto **ShimsContext:**

* 1. [TestMethod] public void FakesTestMethod() {
  2. using (ShimsContext.Create())
  3. {
  4. //...
  5. }

Este cambio tiene que hacerse en todos los test basados en moles que vayan a hacer uso de los Shims en Microsoft Fakes. Recordad que no es necesario el **using** si usamos Stubs.

## Definiendo comportamientos

En los test, la forma básica en la que se definen los comportamientos de los Stubs o Shims no ha cambiado entre Moles y Fakes. Sin embargo, sí ha cambiado el nombre de las clases que los definen. Por ejemplo, en Moles, el comportamiento de una llamada a la propiedad Now de DateTime se declararía así:

* 1. MDateTime.NowGet = () =>
  2. { return new DateTime(1, 1, 1);
  3. };
  4. Con Microsoft Fakes sería así:
  5. System.Fakes.ShimDateTime.NowGet = () =>
  6. { return new DateTime(1, 1, 1);
  7. };

Como sólo es un cambio de namespaces, bastaría con una operación de reemplazar con cualquier editor de texto

## Moles y Microsoft SharePoint

Microsoft Research publicó unas librerías de behaviors para ayudar al testing de SharePoint [(http://research.microsoft.com/en-us/projects/pex/pexsharepointbehaviors.pdf)](http://research.microsoft.com/en-us/projects/pex/pexsharepointbehaviors.pdf). Esta librería es un mock de la API de SharePoint. Cuando migremos a Microsoft Fakes, podemos usar los emuladores de SharePoint, que son una versión de los comportamientos de Moles SharePoint. Podéis leer más sobre estos emuladores en el blog Introducing SharePoint Emulators [(http://blogs.msdn.com/b/visualstudioalm/archive/2012/11/26/introducinghttp://blogs.msdn.com/b/visualstudioalm/archive/2012/11/26/introducing-sharepoint-emulators.aspxsharepoint-emulators.aspx)](http://blogs.msdn.com/b/visualstudioalm/archive/2012/11/26/introducing-sharepoint-emulators.aspx) y están disponibles en Nuget.

## Migrando de frameworks comerciales y open source

## Introducción

Hay un gran número de proyectos open source, como RhinoMocks, Moq, etc, que ofrecen tecnologías equivalentes para hacer stubs como en Microsoft Fakes. Sin embargo, no ofrecen una tecnología similar a los shims de Microsoft Fakes. Sólo los productos comerciales ofrecen la posibilidad de mockear objetos de clases privadas y selladas.

Además de Microsoft Fakes, este tipo de mocking también lo ofrece Telerik en JustMock([http://www.telerik.com/products/mocking.aspx )](http://www.telerik.com/products/mocking.aspx) y Typemock en el producto que tienen llamado Isolator [(http://www.typemock.com/isolator-product-page)](http://www.typemock.com/isolator-product-page). En ambos casos se ofrece la misma funcionalidad de stubs que Microsoft Fakes, permitiendo mockear interfaces, etc., en una versión gratuita y “recortada”. También tienen productos Premium que ofrecen herramientas como los shims para mockear objetos que no son “mockeables”.

## Diferencias entre estos productos y Microsoft Fakes

## Creando assemblies “fake”

La principal diferencia entre estos productos y Microsoft Fakes es el proceso que un desarrollador tiene que hacer para generar el shim.

En Microsoft Fakes, simplemente hacemos clic derecho en la referencia del assembly que queremos mockear y seleccionamos “**Add Fake Assembly”**. Esto generará un nuevo assembly que debemos referenciar para crear objetos shims.

En los productos de Telerik y Typemock, no es necesaria esta pre-generación, se encarga el propio framework en tiempo de ejecución.

## Usando Microsoft Fakes

Para crear tests unitarios tanto Telerik como Typemock usan expresiones lambda para definir el comportamiento, igual que Microsoft Fakes. El formato es un poco diferente en cada uno, pero la intención que se expresa es siempre la misma.

## Telerik JustMock

* 1. En este ejemplo vemos cómo mockear una llamada a DateTime.Now usando JustMock de Telerik:
  2. [TestClass] public class MsCorlibFixture
  3. { static MsCorlibFixture()
  4. {
  5. Mock.Replace(() => DateTime.Now).In<MsCorlibFixture>
  6. (x => x.ShouldAssertCustomValueForDateTime());
  7. }
  9. [TestMethod] public void DateTimeTest()
  10. {
  11. Mock.Arrange(() => DateTime.Now).Returns(new DateTime(2016, 2, 29));
  12. // Act
  13. int result = MyCode.DoSomethingSpecialOnALeapYear();
  15. // Assert
  16. Assert.AreEqual(100, result);
  17. }
  18. }

## Typemock isolator

Ahora vamos a ver cómo se mockea la misma llamada a DateTime.Now usando Typemock Isolator

* 1. [TestMethod, Isolated] public void DateTimeTest()
  2. {
  3. // Arrange
  4. Isolate.WhenCalled(() => DateTime.Now).WillReturn(new DateTime(2016, 2, 29));
  6. // Act
  7. int result = MyCode.DoSomethingSpecialOnALeapYear();
  9. // Assert
  10. Assert.AreEqual(100, result);
  11. }

## Microsoft Fakes

Ahora vamos a ver cómo se mockea esa misma llamada con Microsoft Fakes

* 1. [TestMethod] public void DateTimeTes()
  2. { using (ShimsContext.Create())
  3. {
  4. // Arrange:
  5. System.Fakes.ShimDateTime.NowGet = () => { return new DateTime(2016, 2, 29); };
  7. // Act
  8. int result = MyCode.DoSomethingSpecialOnALeapYear();
  10. // Assert
  11. Assert.AreEqual(100, result);
  12. }
  13. }

## Migrando desde Moq

En esta sección veremos cómo migrar algunas de las características más usadas de Moq, Stubs a Microsoft Fakes.

La principal diferencia entre Moq y Stubs es que Moq usa interfaces genéricas y clases abstractas que tienen que ser “stubbeadas” y Stubs usa la generación de código para implementar clases que se derivan de las interfaces y de las clases abstractas.

Las clases stub generadas por Microsoft Fakes ofrecen miembros extra que son llamados por las propiedades y métodos que están siendo stubbeados para ofrecer valores de retorno o para ejecutar cualquier código que sea necesario.

Una de las pequeñas diferencias que veremos en el código de los test unitarios es que cuando se accede a los objetos no tendremos que usar el miembro Object de la instancia Mock<T>, ya que el stub de Microsoft Fakes implementa directamente la interfaz o deriva de la clase abstracta que ha sido “stubbeada”.

En el resto de esta sección veremos algunos ejemplos de código, veremos los escenarios que ofrece Moq y luego veremos cómo cambiarlo para que funcione con Stubs.

## Código de ejemplo

En el resto de la sección, haremos referencia a un código de ejemplo. Usaremos una clase muy simple de cuenta de banco como la que vemos en el siguiente trozo de código

* 1. public enum TransactionType { Credit, Debit };
  2. public interface ITransaction
  3. {
  4. decimal Amount { get; }
  5. TransactionType TransactionType { get; }
  6. } public interface ITransactionManager
  7. {
  8. int GetAccountTransactionCount(DateTime date);
  9. ITransaction GetTransaction(DateTime date, int transactionNumber);
  10. void PostTransaction(decimal amount, TransactionType transactionType);
  11. event TransactionEventHandler TransactionEvent;
  12. }
  13. public delegate void TransactionEventHandler(DateTime date, decimal amount, TransactionType tran sactionType);

## Cambiando el Setup con Returns

En Moq, el método Setup se usa para crear un stub que responderá al conjunto de parámetros de entrada con una salida que le indiquemos. Por ejemplo, si queremos que el objeto ITransactionManager que se pasa al código a testear devuelva un valor concreto cuando se le pase una fecha al método GetAccountTransactionCount, deberemos usar el siguiente código:

* 1. DateTime testDate = new DateTime(2012, 1, 1);
  2. Mock<ITransactionManager> mockTM = new Mock<ITransactionManager>(); mockTM.Setup(tm => tm.GetAccountTransactionCount(testDate)).Returns(8);

Esto mismo se consigue con Stubs de la siguiente manera:

* 1. DateTime testDate = new DateTime(2012, 1, 1);
  2. StubITransactionManager stubTM = new StubITransactionManager();
  3. stubTM.GetAccountTransactionCountDateTime = (date) => (date == testDate) ? 8 : default(int);

La clase Stub que crea Fakes ofrece un miembro llamado GetAccountTransactionCountDateTime, que podemos asignar a una expresión Lambda que nos devolverá el valor que queremos. Fijaos que esta expresión lambda comprueba el parámetro de entrada de la misma manera que haría Moq. Si se le pasa un valor diferente al indicado, devolverá el valor por defecto del tipo.

Moq también nos permite llamar al método Setup varias veces para devolver valores diferentes para diferentes entradas. Aquí tenéis un ejemplo:

* 1. // txn1 and txn2 previously set up as mock transactions
  2. mockTM.Setup(tm => tm.GetTransaction(testDate, 0)).Returns(txn1.Object); mockTM.Setup(tm => tm.GetTransaction(testDate, 1)).Returns(txn2.Object);

Esto lo podemos conseguir con una expresión lambda algo más compleja como la siguiente:

* 1. // txn1 and txn2 previously set up as stub transactions
  2. ITransaction[] stubTransactions = new ITransaction[] { txn1, txn2 }; stubTM.GetTransactionDateTimeInt32 = (date, index) => (index >= 0 || index < stubTransactions.Le ngth) ? stubTransactions[index] : null;

Estamos usando un array para poder corresponder los valores de entrada, de manera que la expresión lambda las busca. En este caso estamos ignorando el parámetro de fecha.

Algunas veces, los valores que queremos no se conocen con antelación y usar una colección nos permite inicializar el diccionario dinámicamente, incluso después de que el stub se halla pasado al código que queremos testear. Un escenario en el que necesitamos hacer esto es cuando queremos centralizar la creación del código bajo test en un código de inicialización, pero cada test se ejecuta con diferentes conjuntos de valores. Podríamos hacer algo como esto en Stubs:

* 1. private StubITransactionManager stubTM = new StubITransactionManager(); private List<ITransaction> transactions = new List<ITransaction>(); private DateTime testDate = new DateTime(2012, 1, 1); private Account cut;
  3. [TestInitialize] public void InitializeTest()
  4. { this.stubTM.GetTransactionDateTimeInt32 = (date, index) => (date == testDate && index >= 0 || index < this. transactions.Count) ? this.transactions[index] : null;
  5. this.stubTM.GetAccountTransactionCountDateTime = (date) => (date == testDate) ? this.transactions.C ount : default(int); this.cut = new Account(stubTM);
  6. }
  8. [TestMethod] public void StubCreditsSumToPositiveBalance()
  9. {
  10. // Arrange
  11. this.AddTransaction(10m, TransactionType.Credit);
  12. this.AddTransaction(20m, TransactionType.Credit);
  14. // Act
  15. decimal result = this.cut.CalculateBalance(this.testDate);
  17. // Assert
  18. Assert.AreEqual<decimal>(30m, result);
  19. }
  21. [TestMethod] public void StubDebitsAndCreditsSum()
  22. {
  23. // Arrange
  24. this.AddTransaction(10m, TransactionType.Credit);
  25. this.AddTransaction(20m, TransactionType.Debit);
  27. // Act
  28. decimal result = this.cut.CalculateBalance(this.testDate);
  30. // Assert
  31. Assert.AreEqual<decimal>(-10m, result);
  32. } private void AddTransaction(decimal amount, TransactionType transactionType)
  33. { this.transactions.Add(new StubITransaction
  34. {
  35. AmountGet = () => amount,
  36. TransactionTypeGet = () => transactionType
  37. });
  38. }

Por supuesto, Moq también es capaz de tratar la configuración de métodos con varios parámetros. Pero en Stub es un poco más complejo crear expresiones lambda que comprueban cada uno de los parámetros.

Cuando tenemos varios parámetros de entrada en varios Setups, lo más simple suele ser usar un diccionario que usa como key un Tuple<T,R>. Aquí tenéis un ejemplo:

* 1. private StubITransactionManager stubTM = new StubITransactionManager(); private Dictionary<Tuple<DateTime, int>, ITransaction> transactions = new Dictionary<Tuple<DateT ime, int>, ITransaction>(); private DateTime testDate = new DateTime(2012, 1, 1); private Account cut;
  2. [TestInitialize] public void InitializeTest()
  3. { this.stubTM.GetTransactionDateTimeInt32 =
  4. (date, index) =>
  5. {
  6. ITransaction txn;
  7. if (!this.transactions.TryGetValue(new Tuple<DateTime, int>(date, index), out txn))
  8. { txn = null;
  9. }
  10. return txn;
  11. }; stubTM.GetAccountTransactionCountDateTime = (date) => this.cut = new Account(stubTM);
  12. }
  14. [TestMethod] public void StubCreditsSumToPositiveBalance()
  15. {
  16. // Arrange
  17. this.AddTransaction(testDate, 0, 10m, TransactionType.Credit);
  18. this.AddTransaction(testDate, 1, 20m, TransactionType.Credit);
  19. // Act
  20. decimal result = this.cut.CalculateBalance(this.testDate);
  22. // Assert
  23. Assert.AreEqual<decimal>(30m, result);
  24. }
  26. [TestMethod] public void StubDebitsAndCreditsSum()
  27. {
  28. // Arrange this.AddTransaction(testDate, 0, 10m, TransactionType.Credit); this.AddTransaction(testDate, 1, 20m, TransactionType.Debit);
  29. // Act decimal result = this.cut.CalculateBalance(this.testDate);
  31. // Assert
  32. Assert.AreEqual<decimal>(-10m, result);
  33. } private void AddTransaction(DateTime date, int index, decimal amount, TransactionType transactio nType)
  34. {
  35. ITransaction txn = new StubITransaction
  36. {
  37. AmountGet = () => amount,
  38. TransactionTypeGet = () => transactionType };
  39. this.transactions.Add(new Tuple<DateTime, int>(date, index), txn); }

Moq también nos ofrece varias formas de enlazar varios valores de entrada en una sola llamada al Setup. Por ejemplo:

* 1. mockTM.Setup(tm => tm.GetTransaction(It.IsAny<DateTime>(), It.IsAny<int>()))
  2. .Returns(txn.Object);

En este caso, un Stub podría simplemente ignorar el parámetro apropiado en la lambda. En el caso anterior podría ser algo así:

* 1. stubTM.GetTransactionDateTimeInt32 = (date, index) => txn1;

## Migrando los Callbacks

Los Callbacks nos permiten registrar un método que se ejecutará cuando ocurra otra acción. Tanto Moq como Stubs nos permiten especificar métodos de callback en el propio test unitario.

Si por ejemplo, queremos llamar a un método como este en nuestra clase de test:

* 1. bool callBackCalled = false;
  3. public void CallBackMethod(decimal param)
  4. {
  5. callBackCalled = true; }

En Moq, usaremos el método .Setup como en el ejemplo anterior. Sin embargo, en lugar de la llamada al .Returns llamaremos al .Callback para indicar cuál es el método que se tiene que ejecutar, pasando los parámetros que hagan falta de manera similar a como lo haríamos en el método Returns:

* 1. [TestMethod]
  2. public void MoqCallback()
  3. {
  4. // arrange
  5. Mock<ITransactionManager> mockTM = new Mock<ITransactionManager>();
  6. mockTM.Setup(tm => tm.PostTransaction(It.IsAny<decimal>(), It.IsAny<TransactionType>())).Callback<decimal, TransactionType> ((amount, transType) => CallBackMethod(amount));
  7. Account cut = new Account(mockTM.Object);
  9. // act
  10. cut.AddCredit(9.99m);
  12. // assert
  13. Assert.AreEqual(true, callBackCalled); }

Con Stubs, el callback se declara como un delegado:

* 1. [TestMethod]
  2. public void StubCallback()
  3. {
  4. // arrange
  5. StubITransactionManager stubTM = new StubITransactionManager(); stubTM.PostTransactionDecimalTransactionType = (amount, transType) => CallBackMethod(amount);
  6. Account cut = new Account(stubTM);
  8. // act
  9. cut.AddCredit(9.99m);
  11. // assert
  12. Assert.AreEqual(true, callBackCalled); }

## Migrando los verify

El Verify se usa en Moq para verificar comportamientos, para asegurarse de que el código que se está testeando ha hecho ciertas llamadas con ciertos parámetros o que se han hecho ciertas llamadas un cierto número de veces.

Sin embargo estas capacidades de comprobación de comportamiento son limitadas. Por ejemplo, no se puede comprobar que los métodos se ejecutan en cierto orden. Los Stubs en Microsoft Fakes no están pensados para usarlos de esa manera; sin embargo, pueden hacer verificaciones de comportamiento si es necesario. En el siguiente ejemplo, queremos testear que se ha lanzado una transacción al balance de apertura cuando se abre una cuenta. Esto puede testearse de otras maneras, pero este ejemplo muestra cómo testear un comportamiento. En Moq, el test podría ser algo parecido a esto:

* 1. [TestMethod]
  2. public void MoqAccountOpenPostsInitialBalanceCreditTransaction()
  3. {
  4. // Arrange
  5. Mock<ITransactionManager> mockTM = new Mock<ITransactionManager>();
  6. Account cut = new Account(mockTM.Object);
  8. // Act cut.Open(10m);
  10. // Assert
  11. mockTM.Verify(tm => tm.PostTransaction(10m, TransactionType.Credit), Times.Once()); }

Usando Stubs en Microsoft Fakes esto necesita un poco más de código en la expresión lambda para grabar las llamadas:

* 1. [TestMethod]
  2. public void StubAccountOpenPostsInitialBalanceCreditTransaction()
  3. {
  4. // Arrange int callCount = 0;
  5. StubITransactionManager stubTM = new StubITransactionManager {
  6. PostTransactionDecimalTransactionType = (amount, type) =>
  7. {
  8. if (amount == 10m && type == TransactionType.Credit)
  9. {
  10. callCount++;
  11. }
  12. }
  13. };
  15. Account cut = new Account(stubTM);
  17. // Act cut.Open(10m);
  19. // Assert
  20. Assert.AreEqual<int>(1, callCount); }

Con Moq, el desarrollador tiene que llamar al método de verificación adecuado, dependiendo del elemento que quiera verificar:

.Verify – para los métodos

.VerifyGet – para los Get de las propiedades

.VerifySet – para los Set de las propiedades

Como Stubs no ofrece métodos de verificación, tendremos que crearnos los nuestros, de manera que no hay diferencia entre verificaciones de métodos o propiedades; es todo código personalizado.

Obviamente, algunas veces hacen falta verificaciones más complejas. Por ejemplo, pueden ser necesarias diferentes combinaciones de parámetros. Esto se puede realizar con la técnica del Diccionario de Tuplas, similar al que vimos en la sección de Setup, para contar el número de llamadas por cada combinación de parámetros.

## Migrando los eventos

En arquitecturas guiadas por eventos, es muy importante ser capaz de lanzar eventos en los tests. Esto se puede hacer tanto en Moq como en Stubs con una sintaxis muy parecida.

En ambos casos, tendremos que registrar un delegado que será llamado cuando se lance el evento. En este ejemplo, sólo setearemos un flag booleano para indicar que el evento se ha lanzado. Sin embargo, las técnicas de verificación, como vimos antes, pueden usarse para comprobar cualquier parámetro que se halla pasado.

En la sección de “act”, lanzamos el evento que queremos testear pasando los parámetros adecuados. Aquí es donde la sintaxis se diferencia. Con Stubs el evento se lanza con una llamada a un método:

* 1. [TestMethod] public void StubsraiseEvent()
  2. {
  3. // arrange
  4. bool delegateCalled = false;
  5. DateTime testDate = new DateTime(2012, 1, 1);
  6. StubITransactionManager stubTM = new StubITransactionManager();
  7. stubTM.TransactionEventEvent = (date, amount, transType) => { delegateCalled = true; };
  9. // act
  10. // Raise passing the custom arguments expected by the event delegate
  11. stubTM.TransactionEventEvent(testDate, 9.99m, TransactionType.Credit);
  12. // assert
  13. Assert.AreEqual(true, delegateCalled); }

Mientras que en Moq tenemos que usar una expresión lambda para pasar los parámetros:

* 1. [TestMethod] public void MoqRaiseEvent()
  2. {
  3. // arrange
  4. bool delegateCalled = false;
  5. DateTime testDate = new DateTime(2012, 1, 1);
  6. Mock<ITransactionManager> mockTM = new Mock<ITransactionManager>();
  7. mockTM.Object.TransactionEvent += delegate { delegateCalled = true; };
  8. // act
  9. // Raise passing the custom arguments expected by the event delegate
  10. mockTM.Raise(tm => tm.TransactionEvent += null, testDate, 9.99m, TransactionType.Credit);
  12. // assert
  13. Assert.AreEqual(true, delegateCalled); }

## Fakes recursivos

Cuando tenemos un árbol de objetos complejo que tiene que ser “fakeado”, se tarda mucho tiempo en setear todas las propiedades y suele ser innecesario. En muchos casos lo único necesario es que el objeto que se ha fakeado no lance excepciones del tipo NullReferenceException.

Moq ofrece una manera de conseguir que esto no ocurra, seteando todas las referencias/propiedades en el árbol de objetos. Esto se hace usando las opciones de DefaultValue.Mock para los objetos mockeados y el método SetupAllProperties para las propiedades. De esta manera, el test no lanzará ningún NullReferenceException. Se devolverá el valor por defecto de cualquier objeto que haya que devolver. Por ejemplo, los enteros devolverán un 0, y los strings devolverán un String.Empty. Si hace falta algún otro valor, tendremos que indicarlo de manera explícita:

* 1. Mock<ITransaction> mockTr = new Mock<ITransaction>() { DefaultValue = DefaultValue.Mock }; mockTr.SetupAllProperties();

Con Stubs, se usa una sintaxis muy similar. Simplemente seteando la propiedad InstanceBehavior al comportamiento deseado cuando se acceda a cualquier propiedad o método:

* 1. StubITransaction stubTr = new StubITransaction(); stubTr.InstanceBehavior = StubBehaviors.DefaultValue;

## Ejemplo adicional

Por motivos ilustrativos, vamos a ver otro ejemplo para terminar de migrar de Moq a Microsoft Fakes. Este ejemplo está basado en el post [*Mocking HttpWebRequest using Microsoft Fakes*,](http://hamidshahid.blogspot.co.uk/2013/01/mocking-httpwebrequest-using-microsoft.html) tenemos un objeto WebServiceClient que usa un HttpWebReques, que nos gustaría testear:

* 1. public class WebServiceClient
  2. {
  3. /// <summary>
  4. /// Calls a web service with the given URL
  5. /// </summary>
  6. /// <param name="url">The web service's URL</param>
  7. /// <returns>True if the services responds with an OK status code (200). False Otherwise</re turns> public bool CallWebService(string url)
  8. { var request = CreateWebRequest(url);
  9. var isValid = true;
  10. try { var response = request.GetResponse() as HttpWebResponse;
  11. isValid = HttpStatusCode.OK == response.StatusCode;
  12. } catch (Exception ex)
  13. { isValid = false;
  14. } return isValid;
  15. }
  17. /// <summary>
  18. /// Creates an HttpWebRequest object
  19. /// </summary>
  20. /// <param name="url">The URL to be called.</param>
  21. /// <returns>An HttpWebRequest.</returns> private static HttpWebRequest CreateWebRequest(string url)
  22. { var request = WebRequest.Create(url) as HttpWebRequest;
  23. request.ContentType = "text/xml;charset=\"utf-8\"";
  24. request.Method = "GET";
  25. request.Timeout = 1000;
  26. request.Credentials = CredentialCache.DefaultNetworkCredentials;
  27. return request;
  28. }
  29. }

Para usar Moq necesitamos crear un objeto CustomWebRequestCreate que implemente la interfaz IWebRequestCreate. Esto nos permite mockear el HttpWebResponse usando RegisterPrefix:

* 1. /// <summary>
  2. /// A custom implementation of IWebRequestCreate for Web Requests.
  4. /// </summary>
  5. /// <summary>A web request creator for unit testing</summary> public class CustomWebRequestCreate : IWebRequestCreate
  6. {
  7. /// <summary>
  8. /// The web request. /// </summary> private static WebRequest nextRequest;
  10. /// <summary>
  11. /// Internally held lock object for multi-threading support.
  12. /// </summary> private static object lockObject = new object();
  14. /// <summary>
  15. /// Gets or sets the next request object.
  16. /// </summary> public static WebRequest NextRequest
  17. { get { return nextRequest;
  18. } set { lock (lockObject)
  19. { nextRequest = value;
  20. }
  21. }
  22. }
  24. /// <summary>
  25. /// Creates a Mock Http Web request
  26. /// </summary>
  27. /// <param name="httpStatusCode"></param>
  28. /// <returns>The mocked HttpRequest object</returns>
  29. public static HttpWebRequest CreateMockHttpWebRequestWithGivenResponseCode(HttpStatusCode httpStatusCode)
  30. { var response = new Mock<HttpWebResponse>(MockBehavior.Loose);
  31. response.Setup(c => c.StatusCode).Returns(httpStatusCode);
  33. var request = new Mock<HttpWebRequest>();
  34. request.Setup(s => s.GetResponse()).Returns(response.Object);
  35. NextRequest = request.Object;
  36. return request.Object;
  37. }
  39. /// <summary>
  40. /// Creates the new instance of the CustomWebRequest.
  41. /// </summary>
  42. /// <param name="uri">The given Uri</param>
  43. /// <returns>An instantiated web request object requesting from the given Uri.</returns>
  44. public WebRequest Create(Uri uri)
  45. { return nextRequest;
  46. }
  47. }

Nuestro test en Moq sería algo así:

* 1. [TestMethod] public void TestThatServiceReturnsAForbiddenStatuscode()
  2. {
  3. // Arrange var url = "http://testService"; var expectedResult = false;
  4. WebRequest.RegisterPrefix(url, new CustomWebRequestCreate());
  5. CustomWebRequestCreate.CreateMockHttpWebRequestWithGivenResponseCode(HttpStatusCode.Forbidden);
  6. var client = new WebServiceClient();
  8. //Act bool actualresult = client.CallWebService(url);
  10. //Assert
  11. Assert.AreEqual(expectedResult, actualresult); }

Con Microsoft Fakes, podemos falisificar el objeto HttpWebRequest sin tener que implementar la interfaz IWebRequestCreate. El test sería algo así:

* 1. [TestMethod] public void TestThatServiceReturnsAForbiddenStatuscode()
  2. { using (ShimsContext.Create())
  3. {
  4. // Arrange
  5. var requestShim = new ShimHttpWebRequest();
  6. ShimWebRequest.CreateString = (uri) => requestShim.Instance;
  7. requestShim.GetResponse = () => { return new ShimHttpWebResponse() { StatusCodeGet = () =>
  8. { return HttpStatusCode.Forbidden; } }; }; var client = new WebServiceClient(); var url = "testService"; var expectedResult = false;
  10. // Act
  11. bool actualresult = client.CallWebService(url);
  13. // Assert
  14. Assert.AreEqual(expectedResult, actualresult);
  15. }
  16. }

## Referencias

Para ver más ejemplos de Moq visitad: <http://code.google.com/p/moq/wiki/QuickStart>

## Migrando desde RhinoMocks

En esta sección veremos cómo migrar alguna de las características más usadas en la API de RhinoMocks a Microsoft Fakes. RhinoMocks es uno de muchos proyectos open source que ofrecen formas de stubbear el código bajo test. Rhinomocks usa una API para crear stubs de interfaces y clases abstractas a través de reflexión. Microsoft Fakes lo que hace es generar código para interfaces y clases abstractas. La sintaxis usada para RhinoMocks en este documento corresponde a la versión 3.5 o superior. Para los desarrolladores que quieran usar Microsoft Fakes, tan sólo tienen que hacer click derecho en el assembly para el que se quieren crear stubs y seleccionar la opción “Add Fake Assembly” del menú.

Sólo veremos ejemplos de migración para las APIs que más se usan en RhinoMocks. El conjunto de APIs de RhinoMocks es muy amplio y no está en el alcance de este documento. Para ver detalles que no se ven aquí, ved los documentos de RhinoMocks o de Microsoft Fakes para obtener ayuda en la migración.

## Código de ejemplo

La mayoría de ejemplos que vamos a ver se basan en la siguiente interfaz y clase para testear:

* 1. public interface IDetermineTempWithWindChill
  2. { double WhatisCurrentTemp(double airTemp, double airSpeed); double CalcSpecialCurrentTemp(double airTemp, double airSpeed, double aboveSeaLevel);
  3. } public interface IClassUnderTest
  4. { double WhatIsTheTempToday(double currentAirTemp, double currentWindSpeed, double currentFeetAboveSeaLevel);
  5. } public class ClassUnderTest : IClassUnderTest
  6. { private readonly IDetermineTempWithWindChill \_determineTempWithWindChill;
  7. public ClassUnderTest(IDetermineTempWithWindChill determineTempWithWindChill) {
  8. \_determineTempWithWindChill = determineTempWithWindChill;
  9. } public double WhatIsTheTempToday(double currentAirTemp, double currentWindSpeed, double currentFeetAboveSeaLevel)
  10. { return currentFeetAboveSeaLevel >= 5000.0
  11. ? \_determineTempWithWindChill.WhatisCurrentTemp (currentAirTemp, currentWindSpeed) \* 0.1
  12. : \_determineTempWithWindChill.WhatisCurrentTemp
  13. (currentAirTemp, currentWindSpeed);
  14. }
  15. }

## Migrando lo setups y returns

RhinoMocks usa dos tipos de mockeo: el estricto y el dinámico. El mockeo estricto (Code 31) consiste en que el desarrollador tiene que definir las salidas de todos los métodos que se van a llamar en el test. Si no se ha definido una salida para algún método, se lanzará una excepción para ese método cuando sea llamado. Por defecto, Microsoft Fakes usa el valor por defecto para el mocking. El desarrollador puede sobrescribir este comportamiento

definiendo el InstanceBehavior (Code 32) en la interfaz stubbeada o en la clase abstracta. Si no se ha configurado un Stub con un valor de salida y se le llama, lanzará una excepción cuando se ejecute el test:

* 1. IDetermineTempWithWindChill determineTempWithWindChill =
  2. MockRepository.GenerateStrictMock<IDetermineTempWithWindChill>(); Code 1 – RhinoMock Strict Mocking
  3. stubIDetermineTempWithWindChill.InstanceBehavior = StubBehaviors.NotImplemented; Code 2 – InstanceBehavior NotImplemented sample

Usando el **stubIDetermineTempWithWindChill.InstanceBehavior =** StubBehaviors.NotImplemented; si el desarrollador no define un valor de salida para el método *WhatIsCurrentTemp* se lanzará una excepción cuando se ejecute el test. Para la mayoría de ejemplos, usaremos el mocking dinámico.

Los stubs de RhinoMocks (setup y returns) son llamadas simples a la API, como vemos en el siguiente código:

* 1. [TestMethod] public void TestSetupAndReturn()
  2. {
  3. //Arrange double airTemp = 35; double airSpeed = 5.0;
  4. IDetermineTempWithWindChill determineTempWithWindChill = MockRepository.GenerateStub<IDetermineTempWithWindChill>();
  5. determineTempWithWindChill.Stub(x => x.WhatisCurrentTemp(airTemp, airSpeed)).Return(airTemp \* airSpeed);
  6. IClassUnderTest classUnderTest = new ClassUnderTest(determineTempWithWindChill);
  7. //Act
  8. var results = classUnderTest.WhatIsTheTempToday(airTemp, airSpeed, 2000.0);
  10. //Assert
  11. Assert.AreEqual(airTemp \* airSpeed, results); }

En Microsoft Fakes el stub del setup y del return es un poco más evolucionado con expresiones lambda, como vemos aquí:

* 1. [TestMethod] public void TestSetupAndReturn()
  2. {
  3. //Arrange double airTemp = 35; double airSpeed = 5.0;
  4. StubIDetermineTempWithWindChill stubIDetermineTempWithWindChill = new StubIDetermineTempWithWindChill();
  5. stubIDetermineTempWithWindChill.WhatisCurrentTempDoubleDouble = (x, y) => x \* y;
  6. IClassUnderTest classUnderTest = new ClassUnderTest(stubIDetermineTempWithWindChill);
  7. //Act
  8. var results = classUnderTest.WhatIsTheTempToday(airTemp, airSpeed, 2000.0);
  10. //Assert
  11. Assert.AreEqual(airTemp \* airSpeed, results); }

## Migrando las sentencias Expect y AssertWasCalled

RhinoMocks usa las sentencias Expect, AssertWasCalled o AssertWasNotCalled para verificar algoritmos o llamadas a métodos. La API de RhinoMocks ofrece varias opciones para administrar el nivel de detalle de cuantas veces se llamó a un método. Durante el testing, si alguna de las sentencias de verificación no se corresponde con las llamadas esperadas, se lanza una excepción:

* 1. determineTempWithWindChill.Expect(x => x.WhatisCurrentTemp(airTemp,airSpeed)).Repeat.Times(2);
  2. determineTempWithWindChill.AssertWasCalled(x => x.WhatisCurrentTemp(airTemp, airSpeed),options => options.Repeat.Times(1));
  3. determineTempWithWindChill.AssertWasNotCalled(x => x.CalcSpecialCurrentTemp(airTemp,airSpeed, aboveSeaLevel));

Con Microsoft Fakes, se usan expresiones lambda para realizar las mismas verificaciones. El siguiente código muestra una verificación para ver que *WhatIsCurrentTemp* y *CalcSpecialCurrentTemp* se han llamado una vez. En el caso de *CalcSpecialCurrentTemp* se lanzará una excepción si no se llama al método. Con un poco más de trabajo en la expresión lambda, Microsoft Fakes ofrece el mismo nivel de opciones para verificar algoritmos y llamadas a métodos que RhinoMocks:

* 1. [TestMethod] public void TestSetupAndReturn()
  2. {
  3. //Arrange int WhatIsCurrentTempCalled = 0; int CalcSpecialCurrentTempCalled = 0;
  4. double airTemp = 35; double airSpeed = 5.0;
  6. StubIDetermineTempWithWindChill stubIDetermineTempWithWindChill = new StubIDetermineTempWithWindChill();
  7. stubIDetermineTempWithWindChill.WhatisCurrentTempDoubleDouble = (x, y) => {
  8. WhatIsCurrentTempCalled++; return x \* y;
  9. }; stubIDetermineTempWithWindChill.CalcSpecialCurrentTempDoubleDoubleDouble = (x, y, z) => {
  10. CalcSpecialCurrentTempCalled++;
  11. throw new Exception("CalcSpecialCurrentTemp should not have been " + "called");
  12. };
  14. IClassUnderTest classUnderTest = new ClassUnderTest(stubIDetermineTempWithWindChill);
  15. //Act var results = classUnderTest.WhatIsTheTempToday(airTemp, airSpeed, 2000.0);
  17. //Assert
  18. Assert.AreEqual(airTemp \* airSpeed, results);
  19. Assert.AreEqual(1, WhatIsCurrentTempCalled);
  20. Assert.AreEqual(0, CalcSpecialCurrentTempCalled); }

## Conclusión

Cualquier migración de test de Telerik, RhinoMocks o Typemock requerirá un gran esfuerzo. Las migraciones basadas en Buscar y Reemplazar no funcionarán bien ya que las definiciones de los comportamientos son muy diferentes. Los detalles para cada uno de esos frameworks están fuera del alcance de este documento. Para ver cómo migrar de estos productos a Microsoft Fakes, será mejor leer la documentación de cada producto.

La información contenida en este documento representa la visión Microsoft Corporation sobre los asuntos analizados a la fecha de publicación. Dado que Microsoft debe responder a las condiciones cambiantes del mercado, no debe interpretarse como un compromiso por parte de Microsoft, y Microsoft no puede garantizar la exactitud de la información presentada después de la fecha de publicación.

Este documento es sólo para fines informativos. MICROSOFT NO OFRECE NINGUNA GARANTÍA, EXPRESA, IMPLÍCITA O LEGAL, EN CUANTO A LA INFORMACIÓN CONTENIDA EN ESTE DOCUMENTO.

Microsoft publica este documento bajo los términos de la licencia Creative Commons Attribution 3.0 License. Todos los demás derechos están reservados.

*© 2013 Microsoft Corporation.*

Microsoft, Active Directory, Excel, Internet Explorer, SQL Server, Visual Studio, and Windows son marcas comerciales del grupo de compañías de Microsoft.

Todas las demás marcas son propiedad de sus respectivos dueños

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This document is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, AS TO THE INFORMATION IN THIS DOCUMENT.

Microsoft grants you a license to this document under the terms of the Creative Commons Attribution 3.0 License**.** All other rights are reserved.

*© 2013 Microsoft Corporation.*

Microsoft, Active Directory, Excel, Internet Explorer, SQL Server, Visual Studio, and Windows are trademarks of the Microsoft group of companies.

All other trademarks are property of their respective owners.