Capítulo 5: Técnicas avanzadas

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/>

El tema central de este capítulo no es estrictamente el testing unitario, sino algunos escenarios en los que podemos aprovechar más de lo que ofrece Fakes. Siguiendo los principios establecidos del Test-Driven Development suele ser la mejor opción cuando empezamos un nuevo proyecto. Sin embargo, cuando tratamos con código heredado (legacy code) que no fue diseñado para que fuese testable, se presentan muchas situaciones que tenemos que solventar. En muchos casos, el equipo de desarrollo original no está ya disponible, y los detalles de la implementación, como la documentación, no está accesible.

En estas circunstancias, suele ser necesario entender el comportamiento de la implementación. El framework de testing unitario que ofrece Visual Studio, junto a Microsoft Fakes, son un conjunto de herramientas perfectas para esto. Además, la habilidad inherente de ejecutar los tests de manera selectiva para obtener información adicional sobre lo que está ocurriendo en diferentes condiciones, aumenta el proceso de aprendizaje. Como también veremos, el código resultante generará muchos artefactos, normalmente conocidos como “Emuladores”, que pueden ser muy útiles para un testing unitario “convencional” y se podrán usar en desarrollos posteriores, lo que incluirá un refactoring para mejorar la testabilidad.

El código de referencia es el simulador de tráfico que usamos en el Ejercicio 4. Tiene un número de elementos que están altamente acoplados. Vamos a trabajar sobre algunos de los retos que nos ofrece esta aplicación.

## Tratando con servicios Windows Communication Foundation (WCF)

Muchas aplicaciones se implementan como varios procesos que se comunican a través de servicios. Podemos configurar un entorno para el sistema completo usando Microsoft Test Manager y la característica de Lab Management, pero esto no es recomendable para tareas de testing unitario.

Si en el lado del cliente tenemos una instancia de la interfaz (contrato) que se obtiene de una clase factoría, es un problema sencillo el hacer que la factoría devuelva un objeto que queramos. Sin embargo, si el cliente WCF se instancia internamente, no hay forma de reemplazar las llamadas:

* 1. private void UpdateRoadwork()
  2. { var client = new RoadworkServiceReference.RoadworkServiceClient();
  3. var locations = new List<RoadworkServiceReference.Block>();
  5. // Initialization of locations removed for clarity…
  6. var impediments = client.RetrieveCurrent(locations.ToArray());
  7. }

Si queremos testear este código, o cualquier otro código que llame al método *UpdateRoadword()*, tendremos que tratar con esta situación. La mejor opción sería refactorizar para tratarlo bien, pero hay ocasiones en las que eso no es posible o deseable.

## Rompiendo la dependencia

La solución más simple es hacer un Shim del cliente WCF y ofrecer nuestra propia implementación. Esto no requiere ejecutar una instancia del servicio:

* 1. using (ShimsContext.Create())
  2. {
  3. RoadworkServiceReference.Fakes.ShimRoadworkServiceClient.Constructor = (real) => { };
  4. var intercept = new
  5. FakesDelegates.Func<RoadworkServiceReference.RoadworkServiceClient,
  6. RoadworkServiceReference.Block[], RoadworkServiceReference.Impediment[]>((instance, blocks) =>
  7. {
  8. // Body of Shim removed for brevity…
  9. });
  10. RoadworkServiceReference.Fakes.ShimRoadworkServiceClient.AllInstances.RetrieveCurrentBlockAr ray = intercept;
  11. }

Es importante ver que además de crear un shim en la operación especificada, también se ha creado uno para el constructor. Esto es debido a que un constructor real de WCF lanzará una excepción debido a la falta de configuración del servidor.

Esta aproximación puede tener algunas limitaciones. Primero, la lógica requerida para ofrecer una implementación simple puede que no lo sea tanto, y segundo, esta aproximación enmascarará cualquier problema de serialización que pueda ocurrir, como pasar una clase derivada que no sea reconocida como un tipo válido (known type).

## Mejorando la situación

Si el assembly con la implementación actual del servicio es parte de la solución, hay una solución que solventará ambas limitaciones. Crearemos una instancia actual y usaremos la lógica real, pero evitaremos los aspectos del servicio actual.

Como el cliente y el servidor no comparten la implementación de los tipos, tendremos que transformar los parámetros y la salida. Usando un *DataContractSerializer*, también veremos cualquier problema de serialización:

* 1. var services = new Dictionary<RoadworkServiceReference.RoadworkServiceClient,
  2. RoadworkService.RoadworkService>();
  3. using (ShimsContext.Create())
  4. {
  5. RoadworkServiceReference.Fakes.ShimRoadworkServiceClient.Constructor = real =>
  6. {
  7. services.Add(real, new RoadworkService.RoadworkService());
  8. };
  10. var intercept = new FakesDelegates.Func<RoadworkServiceReference.RoadworkServiceClient, RoadworkServiceReference.Block[], RoadworkServiceReference.Impediment[]>(
  11. (instance, blocks) =>
  12. {
  13. // =============================================================================
  14. // The following (commented out) code uses explicit transforms, see docs for
  15. // reasons this may rapidly become difficult, and other potential issues..
  16. // =============================================================================
  17. // var realBlocks = new List<Models.Block>();
  18. // foreach (RoadworkServiceReference.Block item in blocks)
  19. // {
  20. // var realBlock = Transform(item);
  21. // realBlocks.Add(realBlock);
  22. // }
  23. Models.Block[] dataContractTransform =
  24. DataContractTransform<RoadworkServiceReference.Block[],
  25. Models.Block[]>(blocks);
  26. var realBlocks = new List<Models.Block>(dataContractTransform);
  27. var service = services[instance];
  28. var results = service.RetrieveCurrent(realBlocks);
  29. var impediments = new List<RoadworkServiceReference.Impediment>();
  30. foreach (var result in results)
  31. {
  32. var clientImpediment = new RoadworkServiceReference.Impediment();
  33. clientImpediment.location = Transform(result.Location);
  34. impediments.Add(clientImpediment);
  35. }
  36. return impediments.ToArray();
  37. });

  40. RoadworkServiceReference.Fakes.ShimRoadworkServiceClient.AllInstances.RetrieveCurrentBlockAr ray = intercept;
  41. }

La implementación completa está disponible en el código del Hands-on Lab en:

Exercise 4\Traffic.AdvancedTechniques\Examples\BreakingServiceBoudnaryTechniques.cs

## Tratando con cálculos no deterministas

En este ejemplo, el simulador hace cálculos basados en el intervalo de tiempo que hay entre llamadas sucesivas. Como esto no se puede controlar de una manera acotada, usaremos un shim para interceptar los valores que se le pasan al código, y haremos que nuestros test los usen para las comparaciones.

## Operaciones basadas en Timer

Tratar con elementos de código que se invocan dependiendo del tiempo suele presentar algunos retos:

Si el tiempo es muy rápido, puede que sea imposible saber exactamente cuántas llamadas se hacen.

Si el tiempo es muy lento, el tiempo de test necesario para invocar las llamadas puede ser excesivo.

Para afrontar ambos escenarios, podemos generar un shim sobre el timer y permitir la invocación manual del código:

* 1. TimerCallback applicationCallback = null; object state = null;
  2. TimeSpan interval = TimeSpan.Zero;
  3. System.Threading.Fakes.ShimTimer.ConstructorTimerCallbackObjectTimeSpanTimeSpan = (timer, callba ck, arg3, arg4, arg5) =>
  4. { applicationCallback = callback; state = arg3; interval = arg5;
  5. };
  6. //Shim del timer para capturar los parámetros.
  7. const int IterationCount = 10; for (int i = 1; i <= IterationCount; ++i)
  8. {
  9. applicationCallback(state);
  10. Thread.Sleep(interval);
  11. }

Invocación el código deseado de manera determinista

## Datos no repetibles

Otra situación que se suele dar es cuando la lógica se basa en una distribución creada por un generador de números aleatorios. Esto hace imposible saber exactamente qué datos se generarán. La forma más simple es hacer un Shim de la clase Random, y ofrecer unos valores deterministas.

En este ejemplo vamos a asegurarnos que los coches son todos con la orientación oeste, en lugar de que sea algo aleatorio en el código:

* 1. System.Fakes.ShimRandom.Constructor = (real) => { };
  2. System.Fakes.ShimRandom.AllInstances.NextDouble = this.NextDouble;
  3. System.Fakes.ShimRandom.AllInstances.NextInt32Int32 = this.NextInt32Int32;
  4. private int NextInt32Int32(Random random, int i, int arg3)
  5. {
  6. return (i + arg3) / 2;
  7. }
  8. private double NextDouble(Random random)
  9. {
  10. return 0.5;
  11. }

La implementación completa está disponible en el Hands-on Lab:

Exercise 4\Traffic.AdvancedTechniques\Examples\NonDeterministicBehaviorTechniques.cs

Como estamos tratando con generadores de número aleatorios, hay un detalle que se suele pasar por alto: ¡*Varias instancias con la misma semilla generará la misma secuencia de números!* Si intentáis realizar operaciones con conjuntos independientes de números aleatorios y estáis usando varias instancias de Random para conseguirlo, debéis aseguraros de que tienen la misma semilla. Esto lo veremos en la próxima sección.

## Recopilación de casos de uso y más información analítica.

En este ejemplo, el código que se testea realiza un número de operaciones matemáticas, sin embargo no se encuentra un rango y combinaciones posibles como valore de entrada. Usaremos test unitarios para recopilar los valores que se van a presentar con varias condiciones.

## Validar detalles de implementación

Aquí nos enfrentamos a un problema potencial cuando trabajamos con generación de número aleatorios. Cada vez que se crea una instancia de la clase Random, se usa una semilla; diferentes instancias con la misma semilla generarán la misma secuencia de números.

Nos queremos asegurar de que cada instancia de Random nos devuelve una secuencia única para evitar que varias instancias se queden en un estado de bloqueo. Además, como queremos que este test recopile datos sin tener que alterar el comportamiento del código que se testea, debemos asegurarnos de que el generador de números aleatorios continúa trabajando.

NOTA

|  |
| --- |
| Nota: El constructor sin parámetros usa Environment.TickCount, con lo que varias instancias creadas en un tiempo pequeño podrían tener la misma semilla. |

* 1. System.Fakes.ShimRandom.Constructor = delegate(Random random)
  2. {
  3. ShimsContext.ExecuteWithoutShims(delegate
  4. { var constructor = typeof(Random).GetConstructor(new Type[] { });
  5. constructor.Invoke(random, new object[] { });
  6. });
  7. };
  9. System.Fakes.ShimRandom.ConstructorInt32 = delegate(Random random, int i) {
  10. ShimsContext.ExecuteWithoutShims(delegate
  11. { var constructor = typeof(Random).GetConstructor(new[] { typeof(int) });
  12. constructor.Invoke(random, new object[] { i });
  13. }); if (this.values.Contains(i))
  14. {
  15. passed = false;
  16. Assert.Fail("Multiple Random instances Created with identical seed Value={0}", i);
  17. }
  18. this.values.Add(i);
  19. };

La implementación completa está en el Hands-on Lab:

Excercise 4\Traffic.AdvancedTechnices\Examples\DataGatheringTechniques.cs

## Analizando el estado interno

En este ejemplo comprobaremos algunas operaciones del motor de simulador de rutas. El objetivo es comprobar que se han considerado todas las rutas válidas cuando seleccionamos la mejor ruta para un coche. La lógica para esto está contenida en las clases *ShortestTime* y *ShortestDistance.* Desafortunadamente, la lista de rutas válidas está en una variable local:

* 1. List<Route> consideredRoutes = new List<Route>();
  2. MethodInfo mi = typeof(ShortestTime).GetMethod("SelectBestRoute", BindingFlags.Instance | Bindin gFlags.NonPublic);
  3. System.Collections.Generic.Fakes.ShimList<Route>.AllInstances.AddT0 =
  4. (collection, route) =>
  5. ShimsContext.ExecuteWithoutShims(() =>
  6. { if (this.IsArmed)
  7. { consideredRoutes.Add(route);
  8. } collection.Add(route);
  9. });
  11. // TODO: We can Shim the protected method, but without using reflection, there is no way to invo ke it from within the shim
  12. // FYI: ExecuteWithoutShims disables ALL Shims, thereby breaking the capture of "consideredRoute s", but setting the individual shim to null works. FakesDelegates.Func<ShortestTime, Car, Route> shim = null;
  13. shim = (time, car) =>
  14. {
  15. Route route = null;
  16. IsArmed = true;
  17. ShimShortestTime.AllInstances.SelectBestRouteCar = null;
  18. var result = mi.Invoke(time, new object[] { car });
  19. ShimShortestTime.AllInstances.SelectBestRouteCar = shim;
  20. route = (Route)result;
  21. IsArmed = false;
  22. Assert.IsTrue(consideredRoutes.Count > 0, String.Format("Failed to Find Any Considered Route s from {0} to {1}", car.Routing.StartTrip.Name, car.Routing.EndTrip.Name));
  23. return route; };
  24. ShimShortestTime.AllInstances.SelectBestRouteCar = shim;

}

La implementación completa del código está en el Hands-on Lab:

Excersie 4\Traffic.AdvancedTechniques\Examples\DataGatheringTechniques.cs

## Evitando la duplicación de estructuras de testing

En muchos ejemplos, todo el trabajo necesario para hacer stubs o shims ha sido realizado desde dentro del test unitario. Esto puede llevar a muchas duplicaciones debido a que diferentes elementos pueden requerir una infraestructura similar o idéntica.

Una solución común puede ser crear clases que combinan el shim con la funcionalidad asociada y simplemente la activan en un ShimsContext. Aunque sea útil para reducir duplicaciones, puede que no sirva de mucho en escenarios más complejos en los que o las relaciones entre las clases están muy acopladas o si queremos llamar a implementaciones reales para alguna o todas las funcionalidades.

Una solución más sensata es crear un Doppelgänger. Para conseguirlo, extenderemos el concepto de emulador para que la nueva clase tenga la instancia actual de la clase real, junto con los shims necesarios y funciones de ayuda e incluso alguna conversión entre los dos. Como Doppelgänger es largo y difícil de decir, simplemente nos referiremos a él como “clase Testable” y será fácilmente identificable ya que tendrá el mismo nombre que la clase real pero con el prefijo “Testable”, por ejemplo TestableCar para Car.

Tenemos un conjunto de clases “Testable” que se corresponden con las clases del assembly que queremos testear. Os pedimos que miréis estas clases. Nos permiten, por ejemplo, hacer un shim de una llamada a un servicio y hacer un shim de la inicialización de una ruta, evitando los largos tiempos de inicialización en una TestableCity, o hacer un shim del constructor Random para conseguir el control de un TestableCar La implementación completa está disponible en el Hands-on Lab:

Exercise 4\Traffic.AdvancedTechniques\Examples\AvoidingDuplicationTechniques.cs

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.