Los Mocks no son Stubs

Notas del Traductor:

Algunas palabras como mock o stub tienen una traducción muy dificil porque a pesar de que existen traducciones directas en el diccionario, éstas no parecen reflejar la semántica que el autor les dá en el texto original. En el caso de mock lo traduciremos como imitación. Stub se traducirá como stub. Mockist se traducirá como fingidor en lugar de mezclar los idiomas y escribir algo como mockero. Sobre todo porque mock no suena nada bien en castellano.

Martin Fowler

El término Objetos de Imitación (Mock Objects) se ha convertido en popular para describir objetos especiales que imitan a objetos reales para hacer pruebas. La mayoría de los entornos de programación ya tienen frameworks que facilitan la creación de objetos de imitación. Lo que a menudo es inadvertido sin embargo, es que los objetos de imitación son una forma especial de objeto para tests, que proporciona un estilo diferente de escribir tests. En este artículo explicaré cómo trabajan los objetos de imitación, cómo fomentan la escritura de tests basados en la verificación del comportamiento y cómo la comunidad que hay alrededor de ellos los usa para desarrollar un estilo diferente de hacer pruebas.

La primera vez que me encontré el término “objeto de imitación” fue en la comunidad XP hace años. Desde entonces me he metido con los objetos de imitación cada vez más. En parte porque muchos de los desarrolladores líderes de los objetos de imitación han sido colegas míos en ThoughtWorks en varias ocasiones. Y en parte, porque los veo cada vez más en la literatura de tests influenciada por la XP.

Pero a menudo veo los objetos de imitación descritos de manera póbre. En particular los veo confundidos con los stubs; una ayuda común para probar entornos. Entiendo esta confusión; yo también los ví similares durante un tiempo pero mis conversaciones con los desarrolladores de imitaciones han permitido que algo de conocimiento sobre estos objetos penetre mi testaruda cabeza.

La diferencia es en realidad dos diferencias por separado. Por un lado hay diferencia en como se verifican los resultados de los tests: la distinción es, verificación del estado contra verificación del comportamiento. Por otro lado, hay toda una filosofía referente a la forma en que pruebas y diseño se relacionan, lo que denomino aquí como los enfoques clásico y emulador del Desarrollo Guiado por los Tests (TDD).

(En una versión anterior de éste ensayo, me dí cuenta de que había diferencia, pero las combiné en una. Desde entonces mi comprensión ha mejorado y como resultado es hora de actualizar éste ensayo. Si no has leído el ensayo anterior puedes ignorar mis penas, he escrito éste ensayo como si el anterior no existiese. Si el viejo ensayo te es familiar puedes encontrar útil ver que he dividido la dicotomia del testeo basado en estado y testeo basado en interaccion con la dicotomia de la verificacion estado/comportamiento y la clasica/emuladora. También he ajustado mi nomenclatura para ajustarme a la de Gerard Meszaros en el libro Patrones xUnit.)

Tests Normales

Empezaré ilustrando los dos estilos con un ejemplo simple (el ejemplo está en Java pero los principios tienen sentido en cualquier lenguaje orientado a objetos). Queremos tomar un objeto pedido y rellenarlo desde el objeto almacén. El pedido es muy simple, sólo tiene un producto y una cantidad. El almacén mantiene un inventario de los diferentes productos. Cuando le indicamos al pedido que se rellene desde el almacén hay dos posibles respuestas. Si hay suficiente producto en el almacén como para rellenar el pedido, éste se rellena y la cantidad sustraida del almacén se resta del inventario. Si no hay suficiente cantidad de producto en el almacén, entonces el pedido no se rellena y en el almacén no cambia nada.

Estos dos comportamientos implican unos cuantos tests que se ajustan bastante a los casos JUnit convencionales.

```
public class PedidoStateTester extends TestCase {

  private static String TALISKER = "Talisker";
  private static String HIGHLAND_PARK = "Highland Park";
  private Almacen almacen = new AlmacenImpl();

  protected void setUp() throws Exception {
    almacen.anade(TALISKER, 50);
    almacen.anade(HIGHLAND_PARK, 25);
  }
  public void testPedidoSeRellenaSiHaySuficienteEnElAlmacen() {
    Pedido pedido = new Pedido(TALISKER, 50);
    pedido.fill(almacen);
    assertTrue(pedido.estaRelleno());
    assertEquals(0, almacen.getInventory(TALISKER));
  }
  public void testPedidoNoQuitaCantidadSiNoHaySuficienteEnAlmacen() {
    Pedido pedido = new Pedido(TALISKER, 51);
    pedido.rellenar(almacen);
    assertFalse(pedido.estaRelleno());
    assertEquals(50, almacen.getInventory(TALISKER));
  }
```

Los tests xUnit siguen una secuencia típica de cuatro fases: configuración, ejecución, verificación, y finalización. En éste caso la fase de configuración se hace en el método setUp (configurando el almacén) y parcialmente en el própio test (configurando el pedido). La llamada a pedido.rellenar es la fase de ejecución. Aquí es donde le pinchamos al objeto para que haga lo que queremos probar. Las sentencias de aserto son entonces la fase de verificación, comprobando que la ejecución del método se llevó a cabo correctamente. En éste caso no hay fase de finalización explícita, el recolector de basura lo hace por nosotros implícitamente.

Durante la configuración hay dos tipos de objeto que estamos poniendo juntos. El Pedido es la clase que estamos probando, para que Pedido.rellenar funcione necesitamos una instancia de Almacén. En esta situación Pedido es el objeto en que nos estamos concentrando en probar. A la gente orientada a los tests le gustan los términos cómo objeto bajo tests o sistema bajo test para nombrar algo asi. Cualquiera de los dos es igualmente feo de decir pero cómo están ampliamente aceptados me tapo la nariz y los uso. Siguiendo a Meszaros usare Sistema Bajo Test o su abreviación en inglés SUT.

Entonces para éste test uso el SUT (Pedido) y un colaborador (Almacén). Necesito un almacén por dos razones: Una es hacer funcionar el comportamiento que estamos probando (porque Pedido.rellenar llama a métodos del almacén) y en segundo lugar lo necesito para la verificación (porque uno de los posibles resultados de Pedido.rellenar es un cambio potencial en el estado del almacén). Según vayamos explorando este asunto más adelante verás que hacemos mucha distinción entre el SUT y los colaboradores (En una versión anterior de éste artículo me refiero al SUT como el objeto primario y a los colaboradores como objetos secundarios).

Éste estilo de tests usa **verificación del estado**: lo cual significa que determinamos si la ejecución ha funcionado correctamente examinando el estado del SUT y sus colaboradores después de la ejecución. Tal como veremos, los objetos de imitación habilitan un enfoque diferente de verificación.

Tests con Objetos de Imitación

Ahora haré lo mismo pero usaré objetos de imitación. Para este código estoy usando la libreria jMock para definir imitaciones. jMock es una libreria de objetos de imitación para Java. Hay más librerías de imitaciones por ahí, pero jMock está al día y está escrita por los que originaron esta técnica, lo cual, la convierte en una buena librería para empezar.

```
public class TesterDeInteraccionConPedido extends MockObjectTestCase {
  private static String TALISKER = "Talisker";

  public void testHacerPedidoQuitaDelInventarioSiEstaEnStock() {
    //setup - data
    Pedido pedido = new Pedido(TALISKER, 50);
    Mock almacenMock = new Mock(Almacen.class);

    //setup - expectations
    almacenMock.expects(once()).method("tieneInventario")
      .with(eq(TALISKER),eq(50))
      .will(returnValue(true));
    almacenMock.expects(once()).method("quitar")
      .with(eq(TALISKER), eq(50))
      .after("hasInventory");

    //exercise
    pedido.rellenar((Almacen) almacenMock.proxy());

    //verify
    almacenMock.verify();
    assertTrue(pedido.estaRelleno());
  }

  public void testHacerPedidoNoQuitaSiNoHaySuficienteEnStock() {
    Pedido pedido = new Pedido(TALISKER, 51);
    Mock almacen = mock(Almacen.class);

    almacen.expects(once()).method("tieneInventario")
      .withAnyArguments()
      .will(returnValue(false));

    pedido.rellenar((Almacen) almacen.proxy());

    assertFalse(pedido.estaRelleno());
  }
```

Concéntrate en *testHacerPedidoQuitaDelInventarioSiEstaEnStock* primero porque he hecho unos cuantos atajos en el último test.
Para empezar, la fase de configuración (setup) es muy diferente. De entrada está dividida en dos partes: datos y expectativas. Los datos configuran los objetos en los que estamos interesados, en ese sentido es similar a la configuración tradicional. La diferencia está en los objetos que son creados. El SUT es el mismo, un pedido. Sin embargo, el colaborador no es el objeto almacén, en su lugar hay un almacén de imitación, técnicamente una instancia de la clase Mock.
La segunda parte de la configuración crea expectativas en el objeto de imitación. Las expectativas indican qué métodos deben llamarse en las imitaciones cuando el SUT se pone en marcha.
Una vez que todas las expectativas están definidas pongo a correr el SUT. Después de la ejecución hago verificación, la cual tiene dos aspectos. Aserto contra el SUT como antes, sin embargo también verifico las imitaciones comprobando que fueron llamadas de manera acorde a sus expectativas.
La diferencia clave aquí es cómo verificamos que el pedido lo hizo correctamente en su interacción con el almacén. Con verificación del estado hacemos esto comprobando contra el estado del almacén. Las imitaciones usan verificación del comportamiento, donde lo que hacemos es comprobar si el pedido hizo las llamadas correctas en el almacén, en lugar de verificar el estado. Hacemos esto diciéndole a la imitación durante la etapa de configuración, qué debe esperar y pidiéndole que se verifique durante la verificación. Sólo el pedido es chequeado mediante asertos y si el método no cambia el estado

del pedido, no hay nada que verificar en abosluto.
En el segundo test hago varias cosas diferentes. Primero creo la imitación de manera diferente usando el método *mock* de *MockObjectTestCase* en vez de el constructor. Este es un método que hay en jMock y que me conviene porque no necesito llamar explícitamente a verificar más adelante; cualquier imitación creada con este método es automáticamente verificada al final del test. Podría haber hecho esto en el primero también, pero quería mostrar la verificación más explícitamente para enseñar cómo se hacen tests con imitaciones.

La segunda diferencia en el segundo test es que he rebajado las condiciones en cuanto a las condiciones de la expectativa usando *withAnyArguments* (con argumentos cualquiera). La razón para esto es que el primer test comprueba que el número es pasado al almacén, de modo que el segundo test no necesita repetir ese elemento. Si la lógica de negocio del pedido necesita ser cambiada posteriormente, entonces sólo un test fallará, facilitando el esfuerzo de migrar los tests. Al mismo tiempo resulta que podría haber dejado *withAnyArguments* totalmente fuera, porque esto es lo que sucede por defecto.

Usando EasyMock

Hay un buen número de librerias de objetos de imitación. Una que uso bastante es EasyMock, tanto en su versión Java como en .Net. EasyMock también habilita verificación, pero tiene unas cuantas diferencias en cuanto a estilo con jMock las cuales vale la pena discutir. Aquí están los tradicionales tests otra vez:

```
public class TesterEasyPedido extends TestCase {

 private static String TALISKER = "Talisker";

 private MockControl almacenControl;
 private Almacen almacenMock;

 public void setUp() {
  almacenControl = MockControl.createControl(Almacen.class);
  almacenMock = (Almacen) almacenControl.getMock();
 }

 public void testHacerPedidoQuitaDelInventarioSiHayStock() {

  //setup - data
  Pedido pedido = new Pedido(TALISKER, 50);

  //setup - expectations
  almacenMock.tieneInventario(TALISKER, 50);
  almacenControl.setReturnValue(true);
  almacenMock.quitar(TALISKER, 50);
  almacenControl.replay();

  //exercise
  pedido.rellenar(almacenMock);

  //verify
  almacenControl.verify();

  assertTrue(pedido.estaRelleno());
 }

 public void testHacerPedidoNoQuitaSiNoHaySuficienteEnStock() {
  Pedido pedido = new Pedido(TALISKER, 51);
```

```
    almacenMock.tieneInventario(TALISKER, 51);
    almacenControl.setReturnValue(false);
    almacenControl.replay();

    pedido.rellenar((Almacen) almacenMock);

    assertFalse(pedido.estaRelleno());
    almacenControl.verify();
  }
}
```

EasyMock usa la metáfora de record/replay (grabar/ejecutar) para definir las expectativas. Para cada objeto que deseas imitar creas un control y objeto de imitación, que residen ambos en la misma variable. La imitación satisface la interfaz del objeto secundario y el control te da funcionalidad adicional. Para indicar una expectativa llamas al método con los argumentos que esperas, en la imitación. Sigues con una llamada al control si quieres devolver un valor. Una vez que has acabado de definir las expectativas llamas a replay en el control(,) en cuyo punto la imitación termina de grabar las expectativas y está lista para responder al objeto primario. Una vez hecho llamas a verificar en el control.

Parece que mientras la gente a menudo se queda desconcertada con la metáfora de record/replay, se acostumbran bastante rápido a ella. Es una ventaja sobre las restricciones de jMock en que estás haciendo llamadas reales a la imitación en lugar de especificando nombres de métodos mediante cadenas. Esto significa que consigues utilizar el autocompletado de código en tu IDE y que cualquier refactorización de nómbres de método actualizará automáticamente los tests.
Los desarrolladores de jMock están trabajando en una nueva versión que usará otras ténicas para permitirte usar llamadas a métodos reales.

La diferencia entre Imitaciones y Stubs

Cuando se introdujeron por primera vez, mucha gente confundió fácilmente objetos de imitación con la noción común de usar stubs. Desde entonces parece que la gente tiene mejor compresión de las diferencias (y espero que la anterior versión de este ensayo ayudase). Sin embargo para comprender complétamente la manera en que la gente usa imitaciones es imporante entender imitaciones y otros tipos de dobles (¿Dobles?, No te preocupes si este es un término nuevo para ti, espera unos cuantos párrafos y todo estará claro.)
Cuando estas haciendo tests como estos, te estas centrando en un elemento del software cada vez, de aquí el común nombre de pruebas unitarias (unit testing). El problema es que para construir una simple unidad de trabajo, a menudo necesitas otras unidades, de aquí la necesidad de algún tipo de almacén en nuestro ejemplo.
En los dos estilos de tests que he enseñado arriba, el primer caso usa un objeto almacén real y el segundo usa un almacén de imitación. Usar imitaciones es una forma de no usar un almacén real en el test pero hay otras formas de objetos irreales para tests como estos.
El vocabulario para hablar de esto se vuelve farragoso pronto, todo tipo de palabras son usadas: stub, mock, fake (falso), dummy. Para éste artículo voy a seguir el vocabulario del libro de Gerard Meszaros. No es lo que todo el mundo usa pero pienso que es un buen vocabulario y puesto que este es mi ensayo puedo elegir las palabras que uso.
Meszaros usa el término Doble de Test como el genérico para cualquier tipo de objeto que se hace pasar por otro real para propósitos de pruebas. El nombre viene de la noción del doble de los actores en las películas. (Uno de sus propósitos fue evitar usar cualquier nombre que fuese ámpliamente usado)
Meszaros entonces definió cuatro tipos de doble en particular:

* Objetos dummy: se pasan como argumento pero nunca se usan realmente. Normalmente se usan sólo para rellenar listas de parámetros.
* Objetos fake: tienen implementaciones que realmente funcionan pero normalmente toman algún atajo o cortocircuito que les hace inapropiados para producción (como base de datos en memoria por ejemplo)
* Los stubs (stubs) proporcionan respuestas enlatadas a llamadas hechas durante los tests normalmente sin responder en absoluto a cualquier otra cosa fuera de aquello para lo que han sido programados. Los stubs pueden también grabar información sobre las llamadas tal como una pasarela de email que recuerda qué mensajes envió o quizás sólo cómo se enviaron los mensajes.
* Las imitaciones (mocks) son de lo que estamos hablando aquí: objetos preprogramados con expectativas que conforman la especificación de cómo se espera que se reciban las llamadas.

De estos tipos de dobles, sólo las imitaciones insisten en la verificación de comportamiento. Los otros dobles pueden, y normalmente sí que usan verificación del estado. Las imitaciones realmente se comportan como los otros dobles durante la fase de ejecución, cuando necesitan hacer creer al SUT que está hablando con su colaborador real, pero difieren en las fases de configuración y verificación.
Para explorar los dobles de test un poco más, necesitamos extender nuestro ejemplo. Mucha gente sólo usa el doble de test ,si con el objeto real, es dificultoso trabajar. Un caso más común de doble de test sería que dijésemos que queremos enviar un email en caso que rellenar el pedido falle. El problema es que no queremos enviar emails reales a nuestros clientes durante los tests. En su lugar creamos un doble de test para nuestro sistema de email, uno que podamos controlar y manipular.

Aquí podemos empezar a ver la diferencia entre imitaciones y stubs. Si estuviéramos escribiendo un test para este sistema de correo, podríamos escribir un stub simple tal que así:

```
public interface MailService {
  public void send (Message msg);
}

public class MailServiceStub implements MailService {
  private List<Message> messages = new ArrayList<Message>();
  public void send (Message msg) {
    messages.add(msg);
  }
  public int numberSent() {
    return messages.size();
  }
}
```

Podemos usar verificación del estado en el stub así.

```
class PedidoStateTester...
  public void testPedidoEnviaEmailSiNoSeRellena(
) {

    Pedido pedido = new Pedido(TALISKER, 51);
    MailServiceStub mailer = new MailServiceStub();
    pedido.setMailer(mailer);
    pedido.rellenar(almacen);

    assertEquals(1, mailer.numberSent());
  }
```

Por supuesto que esto es un test muy simple, sólo se prueba que un mensaje se ha enviado. No hemos comprobado si fue enviado por la persona adecuada, o con el contenido adecuado pero ilustra la cuestión.

Usando imitaciones este test hubiera parecido bastante diferente.

```
class PedidoInteractionTester...
  public void testPedidoEnviaEmailSiRellenarFalla() {

    Pedido pedido = new Pedido(TALISKER, 51);
    Mock almacen = mock(Almacen.class);
    Mock mailer = mock(MailService.class);
    pedido.setMailer((MailService) mailer.proxy());


    mailer.expects(once()).method("send");
    almacen.expects(once()).method("tieneInventario")
      .withAnyArguments()
      .will(returnValue(false));

    pedido.rellenar((Almacen) almacen.proxy());

  }
}
```

En ambos casos estoy usando test doble en lugar del servicio de email real. La diferencia está en que el stub usa verificación

del estado mientras que la imitación usa verificación del comportamiento.
Para poder usar la verificación del estado en el stub necesito construir algunos métodos extra como soporte. El resultado es que el stub implementa MailService pero además añade métodos extra para el test.
Los objetos de imitación siempre usan verificación del comportamiento, mientras que un stub puede ir en ambas direcciones. Meszaros se refiere a un stub que usa verificación de comportamiento, como Test Espía. La diferencia está en cómo corre y verifica exáctamente el doble, y eso lo dejaré para que lo explores por tu cuenta.

Testeo Clásico y Mock

Ahora estoy en el punto donde puedo explorar la segunda dicotomía: aquella entre TDD clásico y TDD fingido. El gran asunto aquí es cuándo usar una imitación (u otro doble).
El estilo TDD clásico consiste en usar objetos reales si es posible y un doble si es muy costoso usar lo real. Así una persona que practica el estilo TDD clásico usaría un almacén real y un doble para el servicio de email. El tipo de doble no importa mucho realmente.
Un practicante del estilo TDD fingido, sin embargo, siempre usará una imitación para cualquier objeto con un comportamiento interesante. En este caso para ambos, el almacén y el servicio de email.
Aunque los diferentes frameworks de imitaciones fueron diseñados con el estilo fingido en mente, muchos clásicos los encuentran útiles para crear dobles.
Una importante rama del estilo fingido es la del Desarrollo Guiado por el Comportamiento (Behavior Driven Development, BDD). BDD fue desarrollado originalmente por mi colega Dan North como una técnica para ayudar a la gente a aprender mejor el Desarrollo Guiado por los Tests (Test Drive Development) concentrándose en la manera en que TDD es una técnica de diseño. Ésto llevó a cambiar el nombre de tests por el de comportamientos para explorar mejor dónde ayuda TDD a pensar sobre qué necesita hacer un objeto. BDD toma un enfoque fingido pero lo espande tanto de su nomenclatura como con su deseo de integrar el análisis dentro de dicha técnica. No entraré más en esto aquí porque la única relevancia que tiene en éste artículo es que BDD es otra variación de TDD que tiende a usar tests con imitaciones. Lo dejo para que sigas el enlace con más información.

Eligiendo entre las diferentes opciones

En éste artículo he explicado un par de diferencias: verificación del estado o del comportamiento / estilo clásico o estilo fingido de TDD. ¿Cuáles son los argumentos que tener en mente cuando debemos elegir entre ellos? Empezaré con verificación del estado versus verificación de comportamiento.
Lo primero a considerar es el contexto. ¿Estamos pensando en una colaboración simple como pedido y almacén, ó en una engorrosa como entre pedido y servicio de email?
Si la colaboración es sencilla entonces la elección es simple. Si soy un practicante de TDD clásico, no uso imitación ni stub ni ningún tipo de doble. Uso un objeto real y verificación del estado. Si soy un practicante TDD fingido uso una imitación y verificación del comportamiento. Sin decisiones para nada.
Si es una colaboración complicada entonces no hay decisión si soy un fingidor; simplemente uso imitaciones y verificación del comportamiento. Si soy un practicante TDD clásico entonces sí que tengo una elección, pero no es un gran problema cuál hacer. Normalmente los clásicos deciden caso por caso tomando el camino más fácil para cada situación.
Tal como vemos, verificación del estado versus verificación del comportamiento no son una gran decisión. La auténtica cuestión está entre TDD clásico y TDD fingido. Resulta que las características de verificación de estado y comportamiento sí que afectan a la discusión, y en eso es en lo que voy a concentrar la mayor parte de mi energía.
Pero antes que lo haga déjame lanzar un caso en el límite. A veces te ves en cosas en las que realmente es duro usar verificación del estado, incluso si no son colaboraciones complejas. Un gran ejemplo de esto es una caché. Todo el tema de la caché es que no puedes decir si se encontraron datos en ellla (ella) o no basándote en su estado; este es el caso en que la verificación del comportamiento sería la opción más sabia incluso para un practicante TDD clásico estricto. Estoy seguro de que hay otras excepciones en ambos sentidos.
Según vamos indagando en la decisión clásico/fingido, hay montones de factores que considerar así que los he partido en grupos a groso modo.

Guiándo TDD

Los objetos imitación vinieron de la comunidad XP (Programación eXtrema), y una de las principales características de XP es su énfasis en el Desarrollo Guiado por los Tests; donde el diseño de un sistema evoluciona mediante la iteración, guiado por la escritura de tests.
Así pues, no es una sorpresa que los fingidores hablen particularmente del efecto de los tests con imitaciones en el diseño. En particular abogan por el estilo llamado desarrollo guiado por la necesidad. Con este estilo empiezas desarrollando una historia escribiendo el primer test de tu sistema desde las afueras, haciendo de la interfaz de algun (algun) objeto tu SUT. Pensando en las expectativas de los colaboradores, exploras la interacción entre el SUT y sus vecinos; efectivamente diseñando la interfaz exterior del SUT.
Una vez que tienes el primer test corriendo, las expectativas en la imitación proporcionan una especificación para el

siguiente paso y un punto de entrada para los tests. Transformas la expectativa en un test para el colaborador y repites el proceso trabajando en el sistema SUT a SUT. Este estilo también se conoce como outside-in (fuera-dentro), lo cual en inglés es un nombre muy descriptivo para ello. Funciona bien con sistemas de varias capas. Empiezas programando la UI (interfaz de usuario) usando capas de imitación por debajo. Después escribes tests para la capa de más abajo, saltando gradualmente a cada capa del sistema en cada paso. Este es un enfoque muy estructurado y controlado, uno que mucha gente piensa que es útil para guiar a principiantes hacia OO (orientación a objetos) y TDD.
El TDD clásico no provee la misma guía exactamente. Puedes seguir un enfoque paso a paso similar, usando métodos stub en lugar de imitaciones. Para ello, cuando quiera que necesites algo de un colaborador, simplemente escribes literalmente en el código (hard-coded) la respuesta que el test requiere para que el SUT funcione. Entonces, una vez que estas en verde (el test pasa) reemplazas ese código literal con el código apropiado.
Pero el TDD clásico puede hacer otras cosas también. Un estilo común es middle-out (medio-fuera). En éste estilo tomas una característica y decides qué necesitas en el dominio para que dicha característica funcione. Obtienes los objetos de dominio para hacer lo que necesitas y una vez que están funcionando construyes la capa de UI encima. Haciendo esto podría ser que nunca tuvieras que falsificar nada. A mucha gente le gusta porque centra su atención en el modelo de dominio antes que nada, lo cual ayuda a que la lógica de dominio se mantenga separada de la UI.
Debo enfatizar en que tanto fingidores como clásicos hacen esto de una cosa a cada paso. Hay una escuela de pensamiento que construye aplicaciones capa a capa, sin empezar una capa hasta que otra está completa, sin embargo tanto fingidores como clásicos tienden a tener experiencia con metodología ágil y prefieren iteraciones pequeñitas de manera que trabajan función a función en lugar de capa a capa.

Configuración del Test Unitario

Con el TDD clásico tienes que crear no sólo el SUT sino tambien (también) todos los colaboradores que necesita en respuesta al test. Mientras que el ejemplo tenía sólo unos cuantos objetos, los tests reales a menudo involucran una amplia variedad de objetos secundarios. Normalmente estos objetos son creados y desechados con cada ejecución de los tests.
Los tests con imitación sin embargo, sólo necesitan crear el SUT e imitaciones para los vecinos de al lado. Esto puede evitar parte del trabajo que supone construir test unitarios complejos (Por lo menos en teoría. Me he enfrentado a casos de configuraciones de imitaciones bastante complejas pero ello podría ser por no haber usado las herramientas bien).
En la práctica, los que escriben tests clásicos tienden a reusar las configuraciones de los tests tanto como es posible. La manera más simple de hacerlo es poniendo la configuración del test unitario en el método setup (configuración) de xUnit. Hay configuraciones más complicadas que necesitan ser usadas por varias clases, así que en este caso creas clases generadas con las unidades especiales. Normalmente llamo a esto Objetos Madre, basándome en la nomenclatura usada en un proyecto XP temprano de ThoughtWorks. Usar madres es esencial en grandes desarrollos de tests clásicos pero las madres son código adicional que necesita ser mantenido y cualquier cambio en ellos puede tener un efecto significativo en cadena sobre los tests. Puebe haber también una penalización en el rendimiento al configurar las unidades, aunque no he oído que esto sea un problema serio cuando se hace adecuadamente. La mayoría de los objetos de test son computacionalmente baratos de crear, al no doblarse.
Al final resulta que he oído a ambos estilos acusar al otro de suponer demasiado trabajo. Los fingidores dicen que crear las configuraciones de los tests son mucho esfuerzo pero los clásicos dicen que estas se reusan mientras que las imitaciones se crean con cada test.

Aislamiento del Test

Si introduces un fallo en un sistema con tests con imitaciones, causará problemas sólo en aquellos tests cuyo SUT contiene el fallo. Con el enfoque clásico, sin embargo, cualquier test de objetos cliente puede fallar tambien(también), lo que conlleva a fallos donde el objeto que tiene el problema, se usa como colaborador del otro objeto que se está probando. El resultado es que un objeto que se use mucho causa un efecto en cadena que hace fallar todos los tests del sistema que tienen relación.
Los que escriben tests con imitaciones consideran esto un problema de gran magnitud; supone un montón de depuración para encontrar la raíz del problema y solventarlo. Sin embargo los clásicos no expresan esto como una fuente de problemas. Normalmente el culpable es relativamente fácil de señalar, con mirar cuáles tests fallan los desarrolladores pueden decir qué fracasos se derivan de la raíz. Además, si estas escribiendo tests regularmente (tal como debes) entonces sabrás que el daño fue causado por lo último que editaste, así que no es difícil encontrar la falta.
Un factor que puede ser significante aquí es la granularidad de los tests. Ya que los tests clásicos emplean múltiples objetos reales, a menudo encuentras un test que usa una granja de objetos él sólo, en lugar de tener simplemente uno. Si la granja

contiene muchos objetos entonces puede ser mucho más duro encontrar la causa real del fallo. Lo que está sucediendo es que los tests tienen una granularidad tosca.
Es bastante probable que los tests con imitaciones no sufran de este problema, porque la convención es la de imitar a cualquier objeto más allá del primario, lo cual deja claro la necesidad de refinar la granularidad de los tests de cara a los colaboradores. Dicho esto, también es verdad que usar tests de gruesa granularidad no es necesariamente un fallo de los tests clásicos como técnica, sino que más bien el fallo está en no escribir los tests clásicos adecuadamente.
Una buena regla básica es asegurarse de separar los tests de cada clase y hacerlos con una granularidad fina. Mientras que las granjas son a veces razonables, deben limitarse a unos pocos objetos; no más de media docena. Además, si te encuentras depurando un problema debido a tests de escasa granularidad, debes depurar en la manera guiada por el test (test driven), creando tests de granularidad fina al tiempo que depuras.
En esencia, los tests xUnit clásicos no son sólo tests unitarios sino también tests de mini-integración. Resulta que a mucha gente le gusta el hecho de que un test cliente pueda capturar errores que el test printipal de un objeto podría no percibir, particularmente probando áreas en las que las clases interaccionan. Los tests con imitaciones pierden esta cualidad. Hay que añadir que también se corre el riesgo de que las expectativas de los tests con imitaciones sean incorrectas, resultando en tests unitarios que están en verde pero que enmascaran errores inherentes.
Es en este punto donde hago énfasis en que sea cual sea el estilo que uses, debes combinarlo con tests de aceptación de granularidad más gruesa para operar sobre el sistema como un todo. He trabajado en proyectos en los que tardé en usar tests de aceptación y me arrepentí de ello.

Acoplando Tests a Implementaciones

Cuando escribes tests con imitaciones, estas probando las llamadas salientes del SUT para asegurar que habla adecuadamente a sus proveedores. Un test clásico sólo se preocupa del estado final, no de cómo se derivó en ese estado. Los tests con imitaciones están por tanto más acoplados a la implementación del método. Cambiar la naturaleza de las llamadas a los colaboradores normalmente causa que un test con imitaciones se rompa.
Este acoplamiento lleva a varios problemas. El más importante es el efecto que tiene sobre TDD. En tests con imitaciones, al escribir el test tienes que pensar sobre la implementación del comportamiento; tales practicantes ven esto como una ventaja. Los clásicos, sin embargo, piensan que es importante pensar sólo en lo que sucede en la interfaz externa, y dejar toda consideración de la implementación hasta que has terminado de escribir el test.
El acoplamiento también interfiere en la refactorización, porque los cambios en la implementación hacen mucho más probable que se rompan los tests en comparación con los tests clásicos.
Esto puede empeorarse por la naturaleza de las herramientas para imitaciones. A menudo dichas herramientas especifican las llamadas a los métodos de manera muy precisa, incluso tienen que encajar los parámetros que se le pasan, cuando en verdad pueden no ser relevantes para el test en particular. Uno de los propósitos de la herramienta jMock es ser más flexible en su especificación de las expectativas para permitir que sean menos precisas en áreas donde no importa, con el costo de usar cadenas de caracteres que pueden hacer la refactorización más engorrosa.

Estilo de Diseño

Para mí una de las cosas más fascinantes de estos estilos es cómo influyen en las decisiones de diseño. Como he hablado con ambos tipos de practicantes me he dado cuenta de unas pocas diferencias entre los diseños a los que llevan dichos estilos pero estoy seguro de que apenas estoy rargando (rascando) la superficie.
Ya he mencionado una diferencia en cómo abordar las capas. Los tests con imitaciones soportan el enfoque outside-in (una capa a cada paso, desde la UI hacia abajo) mientras que los desarrolladores que prefieren el modelo de dominio separado de la UI prefieren el estilo clásico.
A un nivel mas pequeño me he dado cuenta de que los tests con imitaciones tienden a no usar métodos que devuelven valores en favor de métodos que actúan recibiendo objetos como parámetros. Toma el ejemplo del comportamiento de recabar información de un grupo de objetos para crear un informe en cadena. Una manera común de hacer esto es que el método que construye el informe pase cadenas a los objetos y recoja las cadenas de salida de los mismos para ensamblarlas. Un test con imitaciones lo que haría muy probablemente sería pasar un objeto depósito a los diferentes objetos y hacer que estos añadiesen las cadenas al depósito, tratándolo como el parámetro de entrada.
Los que escriben tests con imitaciones hablan de evitar “descarrilamientos”; métodos encadenados del estilo *getThis().getThat().getTheOther()*. Evitar los métodos encadenados también se conoce como la Ley de Deméter. Mientras que los métodos encadenados son un mal olor, el problema contrario de los objetos middle-man (de en medio) inflados con reenvío de llamadas a métodos es también un mal sintoma. (Siempre he pensado que me sentiría más cómodo con la Ley de

Deméter si se llamase la Sugerencia de Deméter)
Una de las cuestiones más duras de comprender para la gente en diseño orientado a objetos es el principio "Dí y No preguntes", el cual consiste en decirle a un objeto que haga algo en lugar de destriparle datos para hacer la tarea en el código cliente. Los fingidores dicen que usar tests con imitaciones ayuda a promover que se eluda el confeti de los "getter" que tanto se difunde en el código de hoy día. Los clásicos argumentan que hay muchas otras maneras de hacer esto.
Un conocido problema de la verificación basada en el estado es que puede llevar a crear métodos de consulta sólo para soportar la verificación. Nunca es cómodo añadir métodos a la API de un objeto sólo para hacer pruebas; usar verificación del comportamiento evita éste problema. El contra-argumento a esto es que tales modificaciones son normalmente menores en la práctica.
Los fingidores favorecen interfaces rol y afirman que usando este estilo de tests se estimula más las interfaces rol, porque cada colaboración es imitada por separado y por tanto es más probable que sea convertida en una interfaz rol. Así en mi ejemplo de más arriba, usar un objeto depósito para generar un informe, un fingidor hubiera muy probablemente, inventado un rol particular que tuviese sentido en ese dominio, el cual puede ser implementado por un objeto depósito.
Es importante recordar que esta diferencia en el estilo de diseño es clave en la motivación de la mayoría de los fingidores. Los orígenes de TDD fueron el deseo de conseguir tests de regresión fuertes que soportaran un diseño evolutivo. Por el camino los practicantes descubrieron que escribir los tests primero constituyó una mejora significativa en el proceso de diseño. Los fingidores tienen una idea fuerte de qué tipo de diseño es un buen diseño y han desarrollado librerias de imitaciones principalmente para ayudar a la gente a desarrollar éste estilo de diseño.

¿Debo ser un clásico o un fingidor?

Encuentro esta cuestión difícil de responder con confianza. Personalmente siempre he sido un practicante del TDD clásico de la vieja escuela y por tanto no veo razón alguna para cambiar. No veo ningún beneficio atractivo en los TDD con imitaciones, y estoy preocupado por las consecuencias de acoplar los tests a la implementación.
Esto me ha afectado particularmente cuando he observado a un programador fingidor. Realmente me gusta el hecho de que mientras estás escribiendo un tests de concentras en el resultado de su comportamiento y no en cómo está hecho. El fingidor está constantemente pensando en cómo se va a implementar el SUT para escribir las expectativas. Esto hace sentir realmente antinatural para mí.
También sufro de la desventaja de no probar TDD con imitaciones más que en juguetes. Tal como he aprendido del Desarrollo Guiado por Tests (TDD) en sí, a menudo es duro juzgar una técnica sin usarla sériamente. Sí conozco muchos buenos desarrolladores quienes son felices y están contentos con las imitaciones. Así que a pesar de que todavía soy un convencido clásico, más bien presento ambos argumentos tan justamente como puedo para que te puedas forjar tu propia opinión.
Por tanto si los tests con imitaciones te suenan atractivos, te sugiero que lo intentes. Vale particularmente la pena intentarlo si estás teniendo problemas en áreas en las que el TDD con imitaciones está pensado para mejorarlos. Veo dos áreas principales. Una es la de dedicar mucho tiempo a depurar tests que fallan porque no se están rompiendo de manera clara y diciéndote dónde está el problema. (También puedes mejorarlo usando TDD clásico sobre granjas de granularidad fina). La segunda área está donde tus objetos no contienen suficiente comportamiento; los tests con imitaciones puede estimular al equipo de desarrollo a crear objetos con un comportamiento más rico.

Pensamientos Finales

Como el interés en tests unitarios, los frameworks xUnit y el Desarrollo Guiado por los Tests (TDD) ha crecido, cada vez más gente se está metiendo con los objetos de imitación. Mucha gente aprende un poco sobre frameworks de objetos de imitación sin comprender completamente que la división fingidor/clásico les apuntala. Cualquiera que sea la cara de la división por la que te decantes, pienso que es útil comprender esta diferencia. Aunque que no tienes que ser un fingidor para encontrar prácticos los frameworks de imitaciones, es útil comprender el pensamiento que guía muchas de las decisiones de diseño del software.
El propósito de éste artículo fue, y es, señalar estas diferencias y situar el equilibrio entre ellas. Hay más pensamiento de fingidor del que me ha dado tiempo a tratar, particularmente sus consecuencias en el estilo de diseño. Espero que en los próximos pocos años veámos más escrito sobre esto y que profundize nuestro conocimiento de las fascinantes consecuencias de escribir tests antes que el código.

Otras Lecturas

Para una completa revisión de la práctica de los tests xUnit, echa un ojo al próximo libro de Gerard Meszaros (aviso: está en mi serie). También mantiene un sitio web con patrones del libro.

Para encontrar más sobre TDD, el primer lugar a mirar es el libro de Kent.

Para encontrar más sobre el estilo de tests con imitaciones, el primer sitio a mirar es mockobjects.com, donde Steve Freeman y Nat Price abogan por el punto de vista fingidor con ensayos y un blog que vale la pena. En particular lee el excelente artículo de OOPSLA. Para más sobre BDD, una ráma de TDD que es muy fingidora en estilo, empieza con la introducción de Dan North.
También puedes encontrar más sobre éstas técnicas mirando los sitios webs de las herramientas jMock, nMock, EasyMock y en EasyMock de .Net. (Hay mas herramientas para imitaciones por aí, no creas que esta lista está completa)

XP2000 vió el ensayo original sobre objetos de imitación pero está más bien desactualizado ya.

---- Revisiones significativas

02 Enero del 07: Partida la distinción original entre base en el estado y base en la interacción en dos: verificación del estado versus del comportamiento y TDD clásicos versus TDD fingidores. También hice cambios en el vocabulario para adaptarlo al libro de Gerard Meszaros de patrones xunit.

08 Julio de 04: Primera publicación

---- Traducción al Castellano:
Version 1: 31 Julio de 2008 – Carlos Ble Jurado (www.carlosble.com)
Version 2: Revisada y corregida por Eladio Lopez

---- Autor Original:
Martin Fowler (http://martinfowler.com)

--- Texto Original:
http://martinfowler.com/articles/mocksArentStubs.html