**Test Double (lenguaje de patrones)**

***Cómo funciona***

Cuando los productores de una película quieren filmar algo que potencialmente es arriesgado o peligroso para que lo realice el actor principal, contratan a un "doble de riesgo" para que tome el lugar del actor en la escena. El doble de riesgo es una persona altamente entrenada que es capaz de cumplir con los requisitos específicos de la escena. Es posible que el doble de riesgo no pueda actuar, pero sabe cómo caer desde grandes alturas, chocar un auto o hacer lo que sea que requiera la escena. Qué tan parecido necesita ser el doble de riesgo al actor depende de la naturaleza de la escena. Por lo general, las cosas se pueden organizar de manera que alguien que se parezca vagamente al actor en estatura pueda tomar su lugar.

Para fines de prueba, podemos reemplazar al verdadero DOC (¡no al SUT!) con nuestro equivalente del "doble de riesgo": el Test Double. Durante la fase de configuración de la prueba de nuestro Test de Cuatro Fases (página 358), reemplazamos el verdadero DOC con nuestro Test Double. Dependiendo del tipo de prueba que estemos ejecutando, podemos codificar el comportamiento del Test Double o configurarlo durante la fase de configuración. Cuando el SUT interactúa con el Test Double, no se dará cuenta de que no está hablando con el verdadero objeto, pero habremos logrado nuestro objetivo de hacer posibles pruebas imposibles.

Independientemente de qué variación de Test Double elijamos usar, debemos tener en cuenta que no necesitamos implementar toda la interfaz del DOC. En cambio, proporcionamos solo la funcionalidad necesaria para nuestra prueba específica. Incluso podemos construir diferentes Test Doubles para diferentes pruebas que involucren al mismo DOC.

***Cuándo usarlo-patrones***

Podríamos querer usar algún tipo de Test Double durante nuestras pruebas en las siguientes circunstancias:

- Si tenemos un Requisito no probado (ver Errores de producción en la página 268) porque ni el SUT ni sus DOC proporcionan un punto de observación para la salida indirecta del SUT que necesitamos verificar usando Verificación de comportamiento (página 468).

- Si tenemos Código no probado (ver Errores de producción) y un DOC no proporciona el punto de control para permitirnos ejercitar el SUT con las entradas indirectas necesarias.

- Si tenemos Pruebas lentas (página 253) y queremos poder ejecutar nuestras pruebas más rápidamente y, por lo tanto, con más frecuencia.

Cada uno de estos escenarios puede ser abordado de alguna manera mediante el uso de un Test Double. Por supuesto, debemos tener cuidado al usar Test Doubles porque estamos probando el SUT en una configuración diferente a la que se utilizará en producción. Por esta razón, realmente deberíamos tener al menos una prueba que verifique que el SUT funcione sin un Test Double. Necesitamos tener cuidado de no reemplazar las partes del SUT que estamos tratando de verificar porque esa práctica puede resultar en pruebas que prueban el software incorrecto. Además, el uso excesivo de Test Doubles puede resultar en Pruebas frágiles (página 239) como resultado de un Software sobre especificado.

Los Test Doubles vienen en varios sabores principales, como se resume en la Figura 23.1. Las variaciones de implementación de estos patrones se describen con más detalle en los documentos de escritura correspondientes.

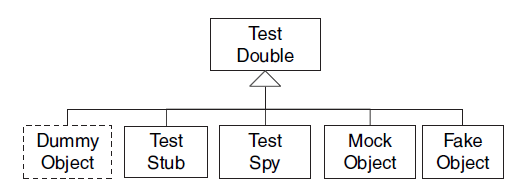
da objeto realiza un seguimiento de su prototipo y delega las solicitudes a él que no sabe cómo manejar.

Figura 23.1 Tipos de Test Doubles. Los Dummy Objects son realmente una alternativa a los patrones de valor. Los Test Stubs se utilizan para verificar entradas indirectas; los Test Spies y los Mock Objects se utilizan para verificar salidas indirectas. Los objetos Fake proporcionan una implementación alternativa.

Estas variaciones se clasifican según cómo/por qué usamos el Test Double. Trataremos las variaciones en torno a cómo construimos los Test Doubles en la sección de "Implementación".

**Variación: Test Stub**

Utilizamos un Test Stub (página 529) para reemplazar un componente real en el que depende el SUT para que la prueba tenga un punto de control para las entradas indirectas del SUT. Su inclusión permite que la prueba fuerce al SUT por caminos que de otra manera no ejecutaría. Podemos clasificar aún más los Test Stubs según el tipo de entradas indirectas que se utilizan para inyectar en el SUT. Un Respondedor (ver Test Stub) inyecta valores válidos, mientras que un Saboteador (ver Test Stub) inyecta errores o excepciones.

Algunas personas utilizan el término "test stub" para referirse a una implementación temporal que se utiliza solo hasta que el objeto o procedimiento real esté disponible. Prefiero llamar a este uso un Temporary Test Stub (ver Test Stub) para evitar confusiones.

**Variación: Test Spy**

Podemos usar una versión más capaz de un Test Stub, el Test Spy (página 538), como punto de observación para las salidas indirectas del SUT. Al igual que un Test Stub, un Test Spy puede necesitar proporcionar valores al SUT en respuesta a llamadas de método. Sin embargo, el Test Spy también captura las salidas indirectas del SUT mientras se ejerce y las guarda para su posterior verificación por parte de la prueba. Por lo tanto, en muchos aspectos, el Test Spy es "solo un" Test Stub con alguna capacidad de grabación. Mientras que un Test Spy se utiliza para el mismo propósito fundamental que un Mock Object (página 544), el estilo de prueba que escribimos usando un Test Spy se parece mucho más a una prueba escrita con un Test Stub.

**Variación: Mock Object**

Podemos usar un Mock Object como punto de observación para verificar las salidas indirectas del SUT mientras se ejerce. Típicamente, el Mock Object también incluye la funcionalidad de un Test Stub en el sentido de que debe devolver valores al SUT si aún no ha fallado las pruebas, pero el énfasis está en la verificación de las salidas indirectas. Por lo tanto, un Mock Object es mucho más que simplemente un Test Stub más afirmaciones: se utiliza de una manera fundamentalmente diferente.

**Variación: Fake Object**

Usamos un Fake Object (página 551) para reemplazar la funcionalidad de un DOC real en una prueba por razones que no sean la verificación de las entradas y salidas indirectas del SUT. Típicamente, un Fake Object implementa la misma funcionalidad que el DOC real, pero de una manera mucho más simple. Mientras que un Fake Object generalmente se construye específicamente para pruebas, la prueba no lo utiliza ni como punto de control ni como punto de observación.

La razón más común para usar un Fake Object es que el DOC real aún no está disponible, es demasiado lento o no se puede usar en el entorno de prueba debido a efectos secundarios perjudiciales. La sección lateral "Pruebas más rápidas sin fixtures compartidos" (página 319) describe cómo encapsulamos todo el acceso a la base de datos detrás de una interfaz de capa de persistencia y luego reemplazamos la base de datos con tablas hash en memoria y logramos que nuestras pruebas se ejecuten 50 veces más rápido. El Capítulo 6, Estrategia de Automatización de Pruebas, y el Capítulo 11, Uso de Test Doubles, proporcionan una descripción general de las diversas técnicas disponibles para facilitar la prueba de nuestro SUT.

**Notas de Implementación**

Varias consideraciones deben tenerse en cuenta cuando estamos construyendo el Test Double (Figura 23.2):

• Si el Test Double debe ser específico para una sola prueba o reutilizable en muchas pruebas

• Si el Test Double debe existir en el código o generarse sobre la marcha

• Cómo indicamos al SUT que use el Test Double (instalación)

Los primeros y últimos puntos se abordan aquí. La discusión sobre la generación de Test Doubles se deja para la sección sobre Test Doubles Configurables.

Dado que las técnicas para construir Test Doubles son prácticamente independientes de su comportamiento (por ejemplo, se aplican tanto a los Test Stubs como a los Mock Objects), he optado por separar las descripciones de las diversas formas en que podemos construir Test Doubles Codificados en Duro y Test Doubles Configurables (página 558) en patrones separados.

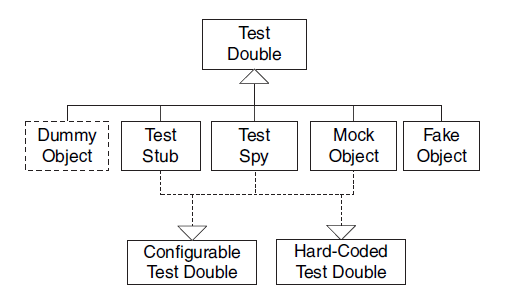


Figura 23.2 Tipos de Test Doubles con opciones de implementación. Solo los Test Stubs, Test Spies y Mock Objects necesitan ser codificados en duro o configurados por la prueba. Los Dummy Objects no tienen implementación; los Fake Objects están instalados, pero no son controlados por la prueba.

**Variación: Test Doubles no Configurables**

Ni los Dummy Objects ni los Fake Objects necesitan ser configurados, cada uno por su propia razón. Los Dummies nunca deben ser utilizados por el receptor, por lo que no necesitan una implementación "real". En contraste, los Fake Objects necesitan una implementación "real" pero una que sea mucho más simple o "ligera" que el objeto que reemplazan. Por lo tanto, ni la prueba ni el automatizador de pruebas necesitarán configurar respuestas o expectativas "predefinidas"; simplemente instalamos el Test Double y permitimos que el SUT lo use como si fuera real.

**Variación: Test Double Codificado en Duro**

Cuando planeamos usar un Test Double específico en una sola prueba, a menudo es más simple codificar en duro el Test Double para devolver valores específicos (para Test Stubs) o esperar llamadas de método específicas (Mock Objects). Los Test Doubles codificados en duro suelen ser construidos a mano por el automatizador de pruebas. Vienen en varias formas, incluido el Self Shunt (ver Test Double Codificado en Duro), donde la Clase de Caso de Prueba actúa como el Test Double; el Test Double Interno Anónimo (ver Test Double Codificado en Duro), donde se utilizan características del lenguaje para crear el Test Double dentro del Método de Prueba; y el Test Double implementado como una Clase de Test Double separada (ver Test Double Codificado en Duro). Cada una de estas opciones se discute con más detalle en el Test Double Codificado en Duro.

**Variación: Test Double Configurable**

Cuando queremos utilizar la misma implementación de Test Double en muchas pruebas, típicamente preferiremos utilizar un Test Double Configurable. Aunque el automatizador de pruebas puede construir manualmente estos objetos, muchos miembros de la familia xUnit tienen kits de herramientas reutilizables disponibles para generar Test Doubles Configurables.

**Instalación del Test Double**

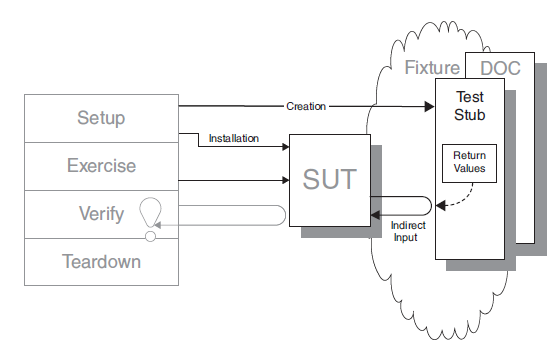
Antes de que podamos ejecutar el SUT, debemos indicarle que use el Test Double en lugar del objeto que el Test Double reemplaza. Podemos utilizar cualquiera de los patrones de dependencia sustituibles para instalar el Test Double durante la fase de configuración de la prueba de nuestro Test de Cuatro Fases. Los Test Doubles Configurables necesitan ser configurados antes de que ejecutemos el SUT, y típicamente realizamos esta configuración antes de instalarlos.

**Ejemplo: Test Double**

Debido a que hay una amplia variedad de razones para usar las variaciones de Test Doubles, es difícil proporcionar un ejemplo único que caracterice la motivación detrás de cada estilo. Por favor, consulte los ejemplos en cada uno de los patrones más detallados mencionados anteriormente.

**Test Stub**

¿Cómo podemos verificar la lógica de manera independiente cuando depende de entradas indirectas de otros componentes de software?

Reemplazamos un objeto real con un objeto específico de prueba que alimenta las entradas indirectas deseadas en el sistema bajo prueba.

En muchas circunstancias, el entorno o contexto en el que opera el SUT influye mucho en el comportamiento del SUT. Para obtener un control adecuado sobre las entradas indirectas del SUT, es posible que tengamos que reemplazar parte del contexto con algo que podamos controlar, es decir, un Test Stub.

**Cómo funciona**

Primero, definimos una implementación específica de prueba de una interfaz en la que depende el SUT. Esta implementación está configurada para responder a las llamadas del SUT con los valores (o excepciones) que ejercitarán el Código No Probado (ver Bugs de Producción en la página 268) dentro del SUT. Antes de ejercitar el SUT, instalamos el Test Stub para que el SUT lo utilice en lugar de la implementación real. Cuando el SUT llama al Test Stub durante la ejecución de la prueba, este devuelve los valores previamente definidos. La prueba puede luego verificar el resultado esperado de manera normal.

**Cuándo usarlo**

Un indicador clave para usar un Test Stub es tener Código No Probado causado por nuestra incapacidad para controlar las entradas indirectas del SUT. Podemos usar un Test Stub como un punto de control que nos permite controlar el comportamiento del SUT con varias entradas indirectas y no necesitamos verificar las salidas indirectas. También podemos usar un Test Stub para inyectar valores que nos permitan superar un punto particular en el software donde el SUT llama a un software que no está disponible en nuestro entorno de prueba.

Si necesitamos un punto de observación que nos permita verificar las salidas indirectas del SUT, debemos considerar el uso de un Mock Object (ver página 544) o un Test Spy (ver página 538). Por supuesto, debemos tener una forma de instalar un Test Double (ver página 522) en el SUT para poder utilizar cualquier forma de Test Double.

*Variación: Responder*

Un Test Stub que se utiliza para inyectar entradas indirectas válidas en el SUT para que pueda llevar a cabo sus funciones se llama Responder. Los Responders se utilizan comúnmente en las pruebas de "camino feliz" cuando el componente real es incontrolable, aún no está disponible o no se puede utilizar en el entorno de desarrollo. Las pruebas serán invariablemente Pruebas de Éxito Simple (ver Método de Prueba en la página 348).

*Variación: Saboteur*

Un Test Stub que se utiliza para inyectar entradas indirectas inválidas en el SUT se llama a menudo Saboteur porque su propósito es descarrilar lo que el SUT está intentando hacer para que podamos ver cómo se las arregla el SUT bajo estas circunstancias. El "descarrilamiento" puede ser causado por el retorno de valores u objetos inesperados, o puede resultar de la generación de una excepción o la ocurrencia de un error en tiempo de ejecución. Cada prueba puede ser una Prueba de Éxito Simple o una Prueba de Excepción Esperada (ver Método de Prueba), dependiendo de cómo se espera que se comporte el SUT en respuesta a la entrada indirecta.

*Variación: Test Stub Temporal*

Un Test Stub Temporal reemplaza a un DOC que aún no está disponible. Este tipo de Test Stub típicamente consiste en una cáscara vacía de una clase real con declaraciones de retorno codificadas. Tan pronto como el DOC real esté disponible, se reemplaza el Test Stub Temporal. El desarrollo guiado por pruebas a menudo nos exige crear Test Stubs Temporales a medida que escribimos código desde afuera hacia adentro; estas cáscaras evolucionan hacia las clases reales a medida que agregamos código a ellas. En el desarrollo impulsado por la necesidad, tendemos a usar Mock Objects porque queremos verificar que el SUT llame a los métodos correctos en el Test Stub Temporal; además, típicamente seguimos utilizando el Mock Object incluso después de que el DOC real esté disponible.

*Variación: Test Stub Procedural*

Un Test Stub Procedural es un Test Stub escrito en un lenguaje de programación procedural. Es particularmente desafiante de crear en lenguajes de programación procedural que no admiten variables de procedimiento (también conocidas como punteros de función). En la mayoría de los casos, debemos colocar ganchos if testing then en el código de producción (una forma de Lógica de Prueba en Producción; ver página 217).

*Variación: Recorte de Cadena de Entidades*

El Recorte de Cadena de Entidades (ver Test Stub en la página 529) es un caso especial de un Responder que se utiliza para reemplazar una red compleja de objetos con un solo Test Stub que finge ser la red de objetos. Su inclusión puede hacer que la configuración del fixture sea mucho más rápida (especialmente cuando los objetos normalmente deberían persistirse en una base de datos) y puede hacer que las pruebas sean mucho más fáciles de entender.

**Notas de Implementación**

Debemos tener cuidado al usar Test Stubs porque estamos probando el SUT en una configuración diferente a la que se usará en producción. Realmente deberíamos tener al menos una prueba que verifique que el SUT funcione sin un Test Stub. Un error común cometido por los automatizadores de pruebas que son nuevos en los stubs es reemplazar una parte del SUT que están tratando de probar. Por esta razón, es importante ser muy claro sobre qué está desempeñando el papel de SUT y qué está desempeñando el papel de configuración de prueba. Además, tenga en cuenta que el uso excesivo de Test Stubs puede resultar en un Software Sobre-especificado (ver Prueba Frágil en la página 239).

Los Test Stubs pueden construirse de varias maneras diferentes dependiendo de nuestras necesidades específicas y las herramientas que tengamos a mano.

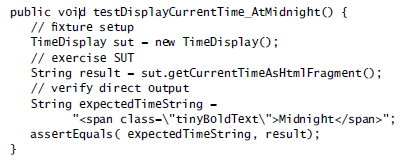
*Variación: Test Stub Codificado*

Un Test Stub Codificado tiene sus respuestas codificadas dentro de su lógica de programa. Estos Test Stubs tienden a ser creados a medida para una sola prueba o un número muy pequeño de pruebas. Consulte Test Double Codificado (página 568) para obtener más información.

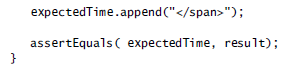
Variación: Test Stub Configurable

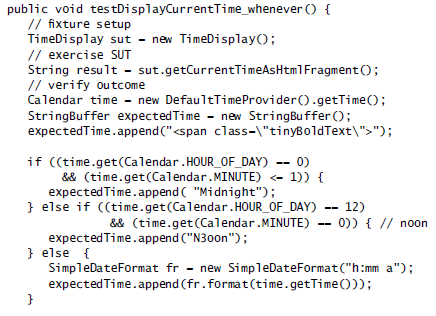
Cuando queremos evitar construir un Test Stub Codificado diferente para cada prueba, podemos usar un Test Stub Configurable (ver Test Double Configurable en la página 558). Una prueba configura el Test Stub Configurable como parte de su fase de configuración de fixture. Muchos miembros de la familia xUnit ofrecen herramientas con las cuales generar Test Doubles Configurables (página 558), incluyendo Test Stubs Configurables.

**Ejemplo Motivador**

La siguiente prueba verifica la funcionalidad básica de un componente que formatea una cadena HTML que contiene la hora actual. Desafortunadamente, depende del reloj del sistema real, ¡así que rara vez pasa!

Podríamos intentar abordar este problema haciendo que la prueba calcule los resultados esperados en función de la hora del sistema actual de la siguiente manera:

Podríamos intentar abordar este problema haciendo que la prueba calcule los resultados esperados en función de la hora del sistema actual de la siguiente manera:



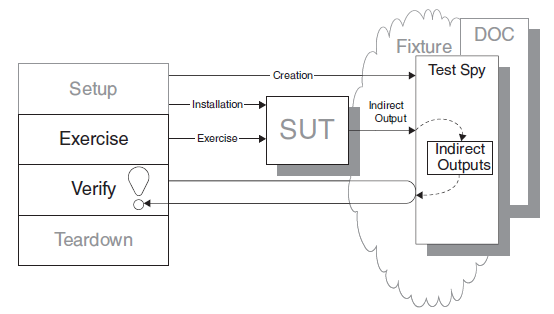
Esta prueba flexible (ver Lógica de Prueba Condicional en la página 200) introduce dos problemas. Primero, algunas condiciones de prueba nunca se ejercitan. (¿Quieres venir a trabajar a la medianoche para ejecutar las pruebas y demostrar que el software funciona a medianoche?) En segundo lugar, la prueba necesita duplicar gran parte de la lógica en el SUT para calcular los resultados esperados. ¿Cómo probamos que la lógica es realmente correcta?

**Notas de Refactorización**

Podemos lograr una verificación adecuada de los inputs indirectos tomando control del tiempo. Para hacerlo, usamos la refactorización de Reemplazar Dependencia con Test Double (página 522) para reemplazar el reloj del sistema real (representado aquí por TimeProvider) con un Reloj Virtual [VCTP]. Luego lo implementamos como un Test Stub que es configurado por la prueba con el tiempo que queremos usar como input indirecto para el SUT.

Luego varios ejemplos.

**Test Spy**

¿Cómo implementamos la Verificación de Comportamiento? ¿Cómo podemos verificar la lógica de forma independiente cuando tiene salidas indirectas hacia otros componentes de software? Utilizamos un Test Double para capturar las llamadas de salida indirectas realizadas a otro componente por el SUT para su posterior verificación por la prueba.

En muchas circunstancias, el entorno o contexto en el que opera el SUT influye mucho en el comportamiento del mismo. Para obtener una visibilidad adecuada de las salidas indirectas del SUT, es posible que tengamos que reemplazar parte del contexto con algo que podamos usar para capturar estas salidas del SUT. El uso de un Test Spy es una forma simple e intuitiva de implementar la Verificación de Comportamiento (ver página 468) a través de un punto de observación que expone las salidas indirectas del SUT para que puedan ser verificadas.

**Cómo Funciona**

Antes de ejercitar el SUT, instalamos un Test Spy como un sustituto de un DOC utilizado por el SUT. El Test Spy está diseñado para actuar como un punto de observación registrando las llamadas de métodos realizadas por el SUT mientras se ejercita. Durante la fase de verificación de resultados, la prueba compara los valores reales pasados al Test Spy por el SUT con los valores esperados por la prueba.

**Cuándo Usarlo**

Una indicación clave para usar un Test Spy es tener un Requerimiento No Probado (ver Errores en Producción en la página 268) causado por la imposibilidad de observar los efectos secundarios de invocar métodos en el SUT. Los Test Spies son una forma natural e intuitiva de ampliar las pruebas existentes para cubrir estas salidas indirectas porque las llamadas a los Métodos de Aserción (ver página 362) son invocadas por la prueba después de que el SUT ha sido ejercitado, al igual que en pruebas "normales". El Test Spy simplemente actúa como el punto de observación que brinda al Método de Prueba (ver página 348) acceso a los valores registrados durante la ejecución del SUT.

Deberíamos usar un Test Spy en las siguientes circunstancias:

- Estamos verificando las salidas indirectas del SUT y no podemos predecir los valores de todos los atributos de las interacciones con el SUT de antemano.

- Queremos que las aserciones sean visibles en la prueba y no creemos que la forma en que se establecen las expectativas del Mock Object (ver página 544) sea suficientemente reveladora de la intención.

- Nuestra prueba requiere igualdad específica de la prueba (por lo que no podemos usar la definición estándar de igualdad implementada en el SUT) y estamos utilizando herramientas que generan el Mock Object, pero no nos dan control sobre los Métodos de Aserción que se llaman.

- Una aserción fallida no puede reportarse efectivamente al Corredor de Pruebas (ver página 377). Esto podría ocurrir si el SUT se está ejecutando dentro de un contenedor que captura todas las excepciones y dificulta informar los resultados o si la lógica del SUT se ejecuta en un hilo o proceso diferente al de la prueba que lo invoca. (Ambos casos realmente requieren refactorización para permitirnos probar directamente la lógica del SUT, pero ese es el tema de otro capítulo).

- Nos gustaría tener acceso a todas las llamadas salientes del SUT antes de hacer alguna aserción sobre ellas.

Si ninguno de estos criterios se aplica, es posible que queramos considerar el uso de un Mock Object. Si estamos intentando abordar Código No Probado (ver Errores en Producción) controlando las entradas indirectas del SUT, un simple Test Stub (ver página 529) puede ser todo lo que necesitamos. A diferencia de un Mock Object, un Test Spy no falla la prueba en la primera desviación del comportamiento esperado. Por lo tanto, nuestras pruebas podrán incluir información diagnóstica más detallada en el Mensaje de Aserción (ver página 370) basada en la información recopilada después de que un Mock Object habría fallado la prueba. En el momento del fallo de la prueba, sin embargo, solo está disponible la información dentro del Método de Prueba mismo para ser utilizada en las llamadas a los Métodos de Aserción. Si necesitamos incluir información que solo es accesible mientras se está ejercitando el SUT, debemos capturarla explícitamente dentro de nuestro Test Spy o debemos usar un Mock Object. Por supuesto, no podremos usar ningún Test Double (ver página 522) a menos que el SUT implemente alguna forma de dependencia substituible.

**Notas de Implementación**

El propio Test Spy puede ser construido como un Doble de Prueba Codificado (ver página 568) o como un Doble de Prueba Configurable (ver página 558). Dado que ejemplos detallados aparecen en la discusión de esos patrones, aquí se proporciona solo un resumen rápido. Asimismo, podemos usar cualquiera de los patrones de dependencia substituible para instalar el Test Spy antes de ejercitar el SUT.

La característica clave en cómo una prueba utiliza un Test Spy está relacionada con el hecho de que las aserciones se realizan desde dentro del Método de Prueba. Por lo tanto, la prueba debe recuperar las salidas indirectas capturadas por el Test Spy antes de poder hacer sus aserciones, lo cual se puede hacer de varias maneras.

*Variación: Interfaz de Recuperación*

Podemos definir el Test Spy como una clase separada con una Interfaz de Recuperación que expone la información registrada. El Método de Prueba instala el Test Spy en lugar del DOC normal como parte de la fase de configuración de la prueba. Después de que la prueba ha ejercitado el SUT, utiliza la Interfaz de Recuperación para recuperar las salidas indirectas reales del SUT del Test Spy y luego llama a los Métodos de Aserción con esas salidas como argumentos.

*Variación: Autoshunt*

Podemos fusionar el Test Spy y la Clase de Prueba en un solo objeto llamado Autoshunt. El Método de Prueba se instala a sí mismo, el Objeto de Prueba, como el DOC en el SUT. Cada vez que el SUT delega en el DOC, en realidad está llamando a métodos en el Objeto de Prueba, que implementa los métodos guardando los valores reales en variables de instancia que pueden ser accedidas por el Método de Prueba. Los métodos también podrían hacer aserciones en los métodos del Test Spy, en cuyo caso el Autoshunt es una variación de un Mock Object en lugar de un Test Spy. En lenguajes con tipos estáticos, la Clase de Prueba debe implementar la interfaz saliente (el punto de observación) en la que el SUT depende para que la Clase de Prueba sea compatible con los variables que se utilizan para contener el DOC.

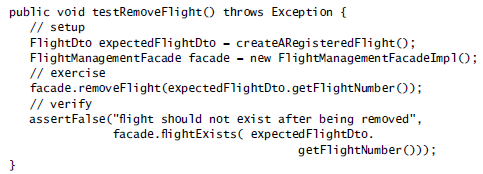
*Variación: Doble de Prueba Interno*

Una forma popular de implementar el Test Spy como un Doble de Prueba Codificado es codificarlo como una clase interna anónima o un bloque de cierre dentro del Método de Prueba y tener esta clase o bloque guardar los valores reales en variables de instancia o locales que son accesibles por el Método de Prueba. Esta variación es realmente otra forma de implementar un Autoshunt (ver Doble de Prueba Codificado).

*Variación: Registro de Salida Indirecta*

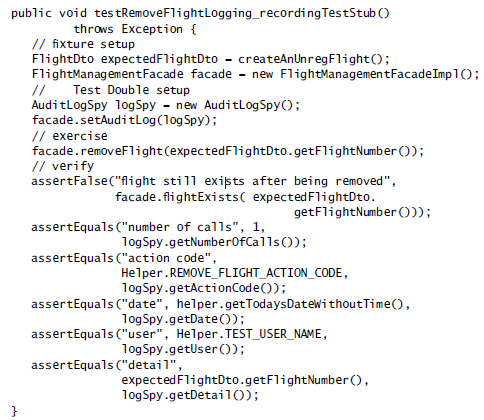
Otra posibilidad es que el Test Spy almacene los parámetros reales en un lugar conocido donde el Método de Prueba pueda acceder a ellos. Por ejemplo, el Test Spy podría guardar esos valores en un archivo o en un objeto de Registro.

**Ejemplo Motivador**

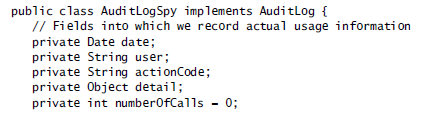
La siguiente prueba verifica la funcionalidad básica de eliminar un vuelo pero no verifica las salidas indirectas del SUT, es decir, el hecho de que se espera que el SUT registre cada vez que se elimina un vuelo junto con la fecha/hora y el nombre de usuario del solicitante.

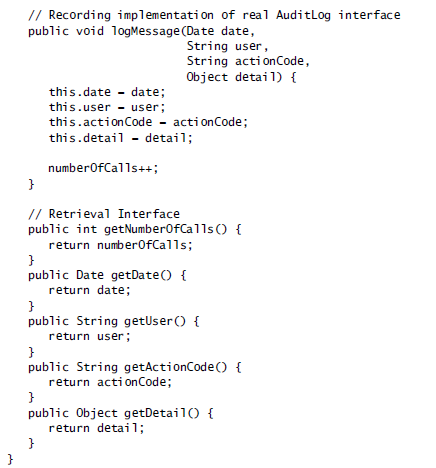
**Notas de Refactorización**

Podemos agregar la verificación de salidas indirectas a pruebas existentes usando una refactorización de Reemplazo de Dependencia con Doble de Prueba (ver página 522). Implica agregar código a la lógica de configuración de la prueba para crear el Test Spy, configurar el Test Spy con cualquier valor que necesite devolver e instalarlo. Al final de la prueba, agregamos aserciones comparando los nombres de métodos esperados y los argumentos de las salidas indirectas con los valores reales recuperados del Test Spy usando la Interfaz de Recuperación.

**Ejemplo: Test Spy**

En esta versión mejorada de la prueba, logSpy es nuestro Test Spy. La instrucción `facade.setAuditLog(logSpy)` instala el Test Spy usando el patrón de Inyección de Dependencia mediante el método Setter (ver Inyección de Dependencia en la página 678). Los métodos `getDate`, `getActionCode`, y así sucesivamente son la Interfaz de Recuperación utilizada para acceder a los argumentos reales de la llamada al registro.

Esta prueba depende de la siguiente definición del Test Spy:

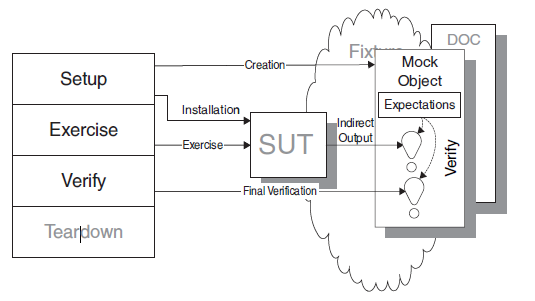


Por supuesto, podríamos haber implementado la Interfaz de Recuperación haciendo que los varios campos de nuestro espía sean públicos y así evitar la necesidad de métodos de acceso. Consulte los ejemplos en Doble de Prueba Codificado para ver otras opciones de implementación.

**Objeto Mock**

¿Cómo implementamos la Verificación de Comportamiento para las salidas indirectas del SUT?

¿Cómo podemos verificar la lógica de manera independiente cuando depende de entradas indirectas de otros componentes de software?

Reemplazamos un objeto en el que el SUT depende con un objeto específico de prueba que verifica que está siendo utilizado correctamente por el SUT.

En muchas circunstancias, el entorno o contexto en el que opera el SUT influye mucho en su comportamiento. En otros casos, debemos "mirar dentro" del SUT para determinar si se ha producido el comportamiento esperado.

Un Objeto Mock es una forma poderosa de implementar la Verificación de Comportamiento (ver página 468) evitando la Duplicación de Código de Prueba (ver página 213) entre pruebas similares. Funciona delegando la tarea de verificar las salidas indirectas del SUT completamente a un Doble de Prueba (ver página 522).

**Cómo Funciona**

Primero, definimos un Objeto Mock que implementa la misma interfaz que un objeto en el que el SUT depende. Luego, durante la prueba, configuramos el Objeto Mock con los valores con los que debe responder al SUT y las llamadas de método (completas con argumentos esperados) que espera del SUT. Antes de ejercitar el SUT, instalamos el Objeto Mock para que el SUT lo utilice en lugar de la implementación real. Cuando se llama durante la ejecución del SUT, el Objeto Mock compara los argumentos reales recibidos con los argumentos esperados usando Aserciones de Igualdad (ver Método de Aserción en la página 362) y falla la prueba si no coinciden. ¡La prueba no necesita hacer ninguna afirmación en absoluto!

**Cuándo Usarlo**

Podemos usar un Objeto Mock como un punto de observación cuando necesitamos hacer Verificación de Comportamiento para evitar tener un Requerimiento No Probado (ver Errores de Producción en la página 268) causado por nuestra incapacidad para observar los efectos secundarios de invocar métodos en el SUT. Este patrón se usa comúnmente durante pruebas endoscópicas [ET] o desarrollo impulsado por la necesidad [MRNO]. Aunque no necesitamos usar un Objeto Mock cuando estamos haciendo Verificación de Estado (ver página 462), podríamos usar un Test Stub (ver página 529) u Objeto Falso (ver página 551). Tenga en cuenta que los conductores de prueba han encontrado otros usos para los kits de herramientas de Objeto Mock, pero muchos de estos son realmente ejemplos de uso de un Test Stub en lugar de un Objeto Mock.

Para usar un Objeto Mock, debemos ser capaces de predecir los valores de la mayoría o todos los argumentos de las llamadas de método antes de ejercitar el SUT. No deberíamos usar un Objeto Mock si una aserción fallida no puede informarse efectivamente al Ejecutor de Pruebas (ver página 377). Esto puede ser el caso si el SUT se ejecuta dentro de un contenedor que captura y "come" todas las excepciones. En estas circunstancias, puede ser mejor usar un Test Spy (ver página 538) en su lugar.

Los Objetos Mock (especialmente los creados con herramientas de simulación dinámica) a menudo usan los métodos equals de los varios objetos que se comparan. Si nuestra igualdad específica de prueba difiere de cómo interpretaría la igualdad el SUT, es posible que no podamos usar un Objeto Mock o que nos veamos obligados a agregar un método equals donde no lo necesitábamos. Este olor se llama Contaminación de Igualdad (ver Lógica de Prueba en Producción en la página 217). Algunas implementaciones de Objetos Mock evitan este problema al permitirnos especificar el "comparador" que se utilizará en las Aserciones de Igualdad.

Los Objetos Mock pueden ser "estrictos" o "permisivos" (a veces llamados "buenos"). Un Objeto Mock "estricto" falla la prueba si las llamadas se reciben en un orden diferente al especificado cuando se programó el Objeto Mock. Un Objeto Mock "permisivo" tolera llamadas fuera de orden.

**Notas de Implementación**

Las pruebas escritas usando Objetos Mock se ven diferentes de las pruebas más tradicionales porque todo el comportamiento esperado debe ser especificado antes de que se ejecute el SUT. Esto hace que las pruebas sean más difíciles de escribir y entender para los neófitos en la automatización de pruebas. Este factor puede ser suficiente para que prefiramos escribir nuestras pruebas usando Test Spies.

El estándar de la Prueba de Cuatro Fases (ver página 358) se altera un poco cuando usamos Objetos Mock. En particular, la fase de configuración del fixture de la prueba se desglosa en tres actividades específicas y la fase de verificación de resultados desaparece más o menos, excepto por la posible presencia de una llamada al método de "verificación final" al final de la prueba.

*Configuración del fixture:*

- La prueba construye el Objeto Mock.

- La prueba configura el Objeto Mock. Este paso se omite para Dobles de Prueba Codificados (ver página 568).

- La prueba instala el Objeto Mock en el SUT.

Ejercicio del SUT:

- El SUT llama al Objeto Mock; el Objeto Mock realiza aserciones.

Verificación de resultados:

- La prueba llama al método de "verificación final".

Desmontaje del fixture:

- Sin impacto.

Examinemos estas diferencias un poco más de cerca:

*Construcción*

Como parte de la fase de configuración del fixture de nuestra Prueba de Cuatro Fases, debemos construir el Objeto Mock que usaremos para reemplazar la dependencia substituible. Dependiendo de qué herramientas estén disponibles en nuestro lenguaje de programación, podemos construir la clase del Objeto Mock manualmente, usar un generador de código para crear una clase de Objeto Mock, o usar un Objeto Mock generado dinámicamente.

*Configuración con Valores Esperados*

Debido a que los kits de herramientas de Objetos Mock disponibles en muchos miembros de la familia xUnit típicamente crean Objetos Mock Configurables (ver página 544), necesitamos configurar el Objeto Mock con las llamadas de método esperadas (y sus parámetros), así como los valores que deben ser devueltos por cualquier función. (Algunos marcos de Objetos Mock nos permiten deshabilitar la verificación de las llamadas de método o solo sus parámetros). Típicamente realizamos esta configuración antes de instalar el Doble de Prueba.

Este paso no es necesario cuando estamos usando un Doble de Prueba Codificado como un Doble de Prueba Interno (ver Doble de Prueba Codificado).

*Instalación*

Por supuesto, debemos tener una forma de instalar un Doble de Prueba en el SUT para poder usar un Objeto Mock. Podemos usar cualquier patrón de dependencia substituible que admita el SUT. Un enfoque común en la comunidad de desarrollo dirigido por pruebas es la Inyección de Dependencia (ver página 678); los desarrolladores más tradicionales pueden preferir la Búsqueda de Dependencia (ver página 686).

*Uso*

Cuando el SUT llama a los métodos del Objeto Mock, estos métodos comparan la llamada de método (nombre del método más argumentos) con las expectativas. Si la llamada de método es inesperada o los argumentos son incorrectos, la aserción falla la prueba inmediatamente. Si la llamada se espera pero está fuera de secuencia, un Objeto Mock estricto falla la prueba de inmediato; en cambio, un Objeto Mock permisivo nota que se recibió la llamada y continúa. Las llamadas perdidas se detectan cuando se llama al método de verificación final.

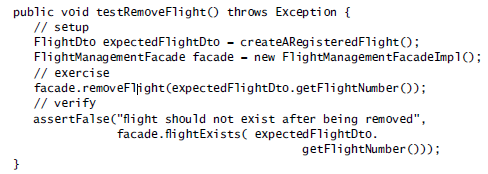
Si la llamada de método tiene algún parámetro saliente o valor de retorno, el Objeto Mock necesita devolver o actualizar algo para permitir que el SUT continúe ejecutando el escenario de prueba. Este comportamiento puede ser codificado o configurado al mismo tiempo que las expectativas. Este comportamiento es el mismo que para los Test Stubs, excepto que típicamente devolvemos valores de camino feliz.

*Verificación Final*

La mayor parte de la verificación de resultados ocurre dentro del Objeto Mock mientras es llamado por el SUT. El Objeto Mock fallará la prueba si los métodos son llamados con argumentos incorrectos o si los métodos son llamados inesperadamente. Pero ¿qué pasa si las llamadas de método esperadas nunca son recibidas por el Objeto Mock? El Objeto Mock puede tener problemas para detectar que la prueba ha terminado y es hora de verificar las expectativas no cumplidas. Por lo tanto, necesitamos asegurarnos de que se llame al método de verificación final. Algunos kits de herramientas de Objetos Mock han encontrado una forma de invocar este método automáticamente al incluir la llamada en el método tearDown. Muchos otros kits de herramientas requieren que recordemos llamar al método de verificación final nosotros mismos.

*Ejemplo motivador*

La siguiente prueba verifica la funcionalidad básica de crear un vuelo. Pero no verifica las salidas indirectas del SUT, es decir, se espera que el SUT registre cada vez que se crea un vuelo junto con la fecha/hora y el nombre de usuario del solicitante.



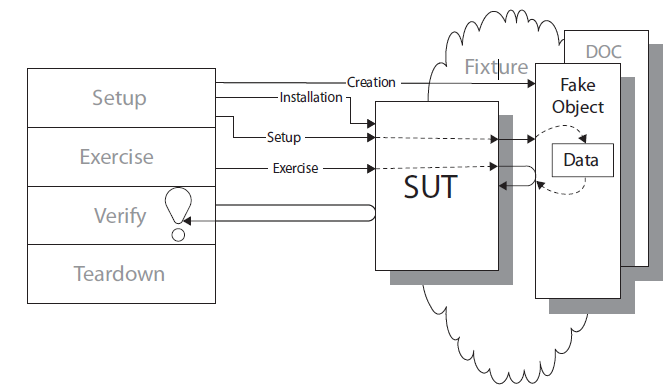
*Notas de refactorización*

La verificación de las salidas indirectas se puede agregar a pruebas existentes utilizando una refactorización de Reemplazo de Dependencia con Doble de Prueba (ver página 522). Esto implica agregar código a la lógica de configuración del fixture de nuestra prueba para crear el Objeto Mock; configurar el Objeto Mock con las llamadas de método esperadas, los argumentos y los valores que deben devolverse; e instalarlo utilizando el mecanismo de dependencia substituible proporcionado por el SUT. Al final de la prueba, agregamos una llamada al método de verificación final si nuestro marco de Objeto Mock lo requiere.

Ejemplos.

**Fake Object**

¿Cómo podemos verificar la lógica de forma independiente cuando los objetos en los que dependemos no pueden ser utilizados? ¿Cómo podemos evitar las pruebas lentas?

Reemplazamos un componente en el que se basa el SUT con una implementación mucho más liviana.

El SUT a menudo depende de otros componentes o sistemas. Aunque las interacciones con estos otros componentes pueden ser necesarias, los efectos secundarios de estas interacciones implementadas por el verdadero DOC pueden ser innecesarios o incluso perjudiciales.

Un Objeto Falso es una implementación mucho más simple y liviana de la funcionalidad proporcionada por el DOC sin los efectos secundarios que elegimos no tener.

**¿Cómo funciona?**

Adquirimos o construimos una implementación muy liviana de la misma funcionalidad proporcionada por un componente en el que el SUT depende e instruimos al SUT para que lo utilice en lugar del verdadero DOC. Esta implementación no necesita tener ninguno de los "ilitys" que el verdadero DOC necesita tener (como escalabilidad); solo necesita proporcionar los servicios equivalentes al SUT para que el SUT no se dé cuenta de que no está utilizando el verdadero DOC.

Un Objeto Falso es una especie de Doble de Prueba que es similar a un Stub de Prueba en muchos aspectos, incluida la necesidad de instalar en el SUT una dependencia substituible. Mientras que un Stub de Prueba actúa como un punto de control para inyectar entradas indirectas en el SUT, sin embargo, el Objeto Falso no lo hace: simplemente proporciona una manera para que las interacciones ocurran de manera autoconsistente. Estas interacciones (es decir, entre el SUT y el Objeto Falso) típicamente serán muchas, y los valores pasados como argumentos de las llamadas de método anteriores a menudo se devolverán como resultados de las llamadas de método posteriores. Contrasta este comportamiento con el de Stub de Prueba y Objetos Falsos, donde las respuestas son o bien codificadas o configuradas por la prueba.

Si bien la prueba normalmente no configura un Objeto Falso, la configuración de la fijación compleja que típicamente implicaría inicializar el estado del DOC también se puede realizar con el Objeto Falso directamente utilizando la Manipulación por Puerta Trasera. Técnicas como Cargador de Datos (ver Manipulación por Puerta Trasera) y Configuración por Puerta Trasera (ver Manipulación por Puerta Trasera) se pueden utilizar con bastante éxito con menos miedo de Software Sobre-Especificado porque simplemente nos vinculan a la interfaz entre el SUT y el Objeto Falso; la interfaz utilizada para configurar el Objeto Falso es una preocupación exclusiva de la prueba.

**Cuándo usarlo**

Deberíamos usar un Objeto Falso siempre que el SUT dependa de otros componentes que no estén disponibles o que dificulten o ralenticen las pruebas (por ejemplo, Pruebas Lentas; ver página 253) y las pruebas necesiten secuencias de comportamiento más complejas de las que valga la pena implementar en un Stub de Prueba u Objeto Falso. También debe ser más fácil crear una implementación liviana que construir y programar Objetos Falsos adecuados, al menos a largo plazo, si construir un Objeto Falso va a valer la pena.

El uso de un Objeto Falso nos ayuda a evitar el Software Sobre-Especificado porque no codificamos las secuencias exactas de llamadas esperadas del DOC dentro de la prueba. El SUT puede variar cuántas veces se llaman los métodos del DOC sin causar que las pruebas fallen.

Si necesitamos controlar las entradas indirectas o verificar las salidas indirectas del SUT, probablemente deberíamos usar un Objeto Falso o Stub de Prueba en su lugar.

A continuación, se describen algunas situaciones específicas donde reemplazamos el componente real con un Objeto Falso.

*Variación: Base de Datos Falsa*

Con el patrón de Base de Datos Falsa, la base de datos real o la capa de persistencia se reemplaza por un Objeto Falso que es funcionalmente equivalente pero que tiene características de rendimiento mucho mejores. Un enfoque que a menudo hemos utilizado implica reemplazar la base de datos con un conjunto de HashTables en memoria que actúan como una forma muy liviana de recuperar objetos que han sido "persistidos" anteriormente en la prueba.

*Variación: Base de Datos en Memoria*

Otro ejemplo de un Objeto Falso es el uso de una base de datos sin disco y de pequeño tamaño en lugar de una base de datos basada en disco con todas las características. Este tipo de Base de Datos en Memoria mejorará la velocidad de las pruebas al menos en un orden de magnitud mientras se renuncia a menos funcionalidad que una Base de Datos Falsa.

*Variación: Servicio Web Falso*

Cuando probamos software que depende de otros componentes que se acceden como servicios web, podemos construir una implementación pequeña codificada o basada en datos que se pueda utilizar en lugar del servicio web real para hacer que nuestras pruebas sean más robustas y evitar tener que crear una instancia de prueba del servicio web real en nuestro entorno de desarrollo.

*Variación: Capa de Servicio Falsa*

Cuando probamos interfaces de usuario, podemos evitar la Sensibilidad a los Datos (ver Prueba Frágil) y la Sensibilidad al Comportamiento (ver Prueba Frágil) de las pruebas reemplazando el componente que implementa la Capa de Servicio [PEAA] (incluida la capa de dominio) de nuestra aplicación con un Objeto Falso que devuelve resultados recordados o basados en datos. Este enfoque nos permite centrarnos en probar la interfaz de usuario sin tener que preocuparnos de que los datos devueltos cambien con el tiempo.

Notas de Implementación

Introducir un Objeto Falso implica dos preocupaciones básicas:

• Construcción de la implementación del Objeto Falso

• Instalación del Objeto Falso

Construcción del Objeto Falso

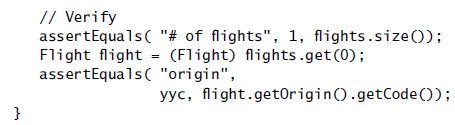
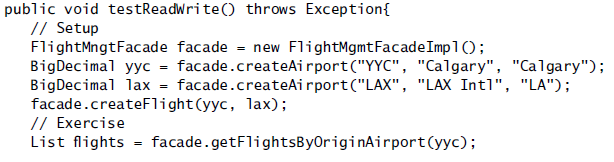
La mayoría de los Objetos Falsos son construidos a mano. A menudo, el Objeto Falso se utiliza para reemplazar una implementación real que sufre problemas de latencia debido a la mensajería real o la E/S de disco con una implementación en memoria mucho más liviana. Con las bibliotecas de clases ricas disponibles en la mayoría de los lenguajes de programación orientados a objetos, generalmente es posible construir una implementación falsa que sea suficiente para satisfacer las necesidades del SUT, al menos para los propósitos de pruebas específicas, con relativamente poco esfuerzo.

Una estrategia popular es comenzar construyendo un Objeto Falso para admitir un conjunto específico de pruebas donde el SUT requiere solo un subconjunto de los servicios del DOC. Si esto resulta exitoso, podemos considerar la posibilidad de expandir el Objeto Falso para manejar pruebas adicionales. Con el tiempo, podemos descubrir que podemos ejecutar todas nuestras pruebas utilizando el Objeto Falso. (Consulte la barra lateral "Pruebas más rápidas sin fijaciones compartidas" en la página 319 para una descripción de cómo engañamos a toda la base de datos con tablas hash y hicimos que nuestras pruebas se ejecutaran 50 veces más rápido).

*Instalación del Objeto Falso*

Por supuesto, debemos tener una forma de instalar el Objeto Falso en el SUT para poder aprovecharlo. Podemos usar cualquier patrón de dependencia sustituible que admita el SUT. Un enfoque común en la comunidad de desarrollo impulsada por pruebas es la Inyección de Dependencias (página 678); los desarrolladores más tradicionales pueden preferir la Búsqueda de Dependencias (página 686). La última técnica también es más apropiada cuando introducimos una Base de Datos Falsa (ver Objeto Falso en la página 551) en un esfuerzo por acelerar la ejecución de las pruebas del cliente; la Inyección de Dependencias no funciona tan bien con este tipo de pruebas.

**Ejemplo Motivador**

En este ejemplo, el SUT necesita leer y escribir registros de una base de datos. La prueba debe configurar el fixture en la base de datos (varias escrituras), el SUT interactúa (lee y escribe) con la base de datos varias veces más, y luego la prueba elimina los registros de la base de datos (varias eliminaciones). Todo este trabajo lleva tiempo, varios segundos por prueba. Esto rápidamente se suma a minutos, y pronto descubrimos que nuestros desarrolladores no están ejecutando las pruebas con tanta frecuencia. Aquí hay un ejemplo de una de estas pruebas:

La prueba llama a createAirport en nuestra Fachada de Servicio [CJ2EEP], que llama, entre otras cosas, a nuestra capa de acceso a datos. Aquí está la implementación real de varios de los métodos que estamos llamando:

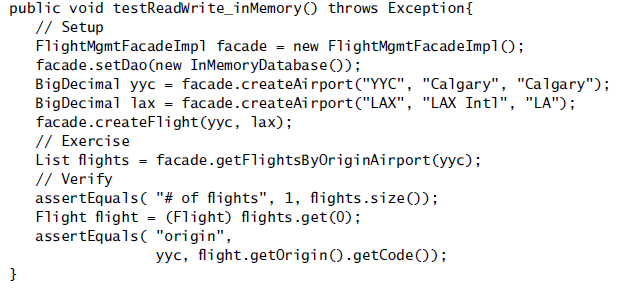
Las llamadas a dataAccess.createAirport, dataAccess.createFlight y TransactionManager.commitTransaction hacen que nuestra prueba se ralentice más. Las llamadas a dataAccess.getAirportByPrimaryKey y dataAccess.getFlightsByOriginAirport son un factor menor pero aún contribuyen a la prueba lenta.

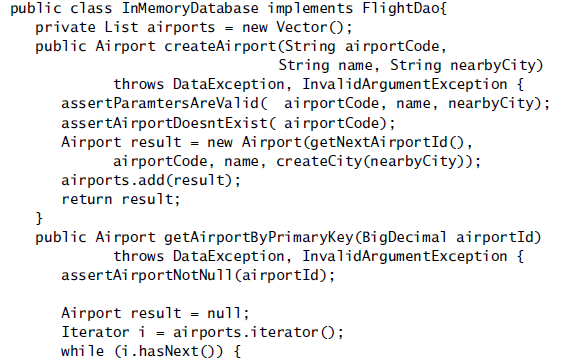
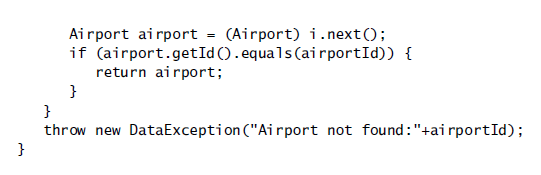
**Notas de Refactorización**

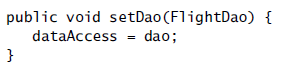
Los pasos para introducir un Objeto Falso son muy similares a los de agregar un Objeto Mock. Si uno no existe, usamos una refactorización de Reemplazar Dependencia con Doble de Prueba (página 522) para introducir una forma de sustituir el Objeto Falso por el DOC, generalmente un campo (atributo) para contener la referencia a él. En lenguajes de tipo estático, podemos tener que hacer una refactorización de Extracción de Interfaz [Fowler] antes de poder introducir la implementación falsa. Luego, usamos esta interfaz como el tipo de variable que contiene la referencia a la dependencia sustituible.

Una diferencia notable es que no necesitamos configurar el Objeto Falso con expectativas o valores de retorno; simplemente configuramos el fixture de la forma normal.\

Ejemplo: Base de Datos Falsa

En este ejemplo, hemos creado un Objeto Falso que reemplaza la base de datos, es decir, una Base de Datos Falsa implementada completamente en memoria utilizando tablas hash. La prueba no cambia mucho, pero la ejecución de la prueba ocurre mucho, mucho más rápido.

Aquí está la implementación de la Base de Datos Falsa:

Ahora todo lo que necesitamos es la implementación del método que instala la Base de Datos Falsa en la fachada para que nuestros desarrolladores estén más que contentos de ejecutar todas las pruebas después de cada cambio de código.