A++

Mejora del sistema de test unitarios

María Victoria Calbet González
Marta Ramírez González
David Romero Esparraga
Jesús Ortiz Calleja
Guillermo Alcalá Gamero
Juan Carlos Utrilla Martín

Contenido

1. El problema de los tests unitarios de Driver-Template

Una de las características que distingue al buen ingeniero de software es su capacidad para evitar duplicar el código. La presencia de código repetido en distintas partes del proyecto provoca que en caso de fallo, éste deba ser corregido en distintos sitios, lo que se traduce en más dinero invertido en solucionar el mismo error.

En esta entrega se nos presenta por primera vez la oportunidad depurar nuestro código gracias al sistema de Driver y Template de los test unitarios.

Hasta ahora, para hacer tests sobre el mismo caso de uso, había que copiar el mismo código, cambiando sólo los parámetros que necesitábamos probar. Con este sistema de Driver y Template, sacamos factor común del código y reutilizamos la lógica del caso de uso y vamos pasando por parámetros los datos referentes a los distintos casos de prueba.

Sin embargo, hay un problema con esta metodología aplicada a JUnit. Hasta ahora, cada caso de uso era un test individual, lo que significa que cada caso de prueba poseía la anotación @Test en su cabecera. Para JUnit, todo método con esta anotación constituye un test individual.

Con el sistema de Driver y Template, el driver se convierte en una matriz de datos donde cada fila se refiere a un caso de uso y la template aglutina la lógica del caso de uso. Tras definir los datos, se llama de forma iterativa a la template para ejecutar el test con los diferentes datos.

Al ser la anotación @Test exclusiva de la template y al ejecutarse de forma iterativa, JUnit reconoce todos los casos de prueba del mismo template como un único test. Si falla uno de los casos, el sistema entiende que ha fallado el método y éste único test se considera como fallo.

Obviando el problema de que un sólo caso de prueba erróneo lleva al sistema a creer que todas la iteraciones han resultado fallidas, encontramos el que a mi parecer es el verdadero problema: Una vez que falla un caso no hay manera de conocer qué error ha provocado el fallo.

En el ámbito de esta entrega supondré que estamos creando un usuario. La creación de un usuario puede salir mal por diferentes motivos: la ausencia de un atributo, un atributo que no cumple las reglas del dominio o la ejecución de un assert en el servicio.

Cada caso de prueba es una prueba distinta pero por restricciones de la tecnología, varios casos de prueba se ejecutarán dentro de un mismo test. Si realizando mantenimiento en el código, introducimos un bug que no permite crear usuarios, los tests sólo nos servirán para advertirnos que la el caso de uso ya no funciona.

Al encontrarme de primera mano con este problema, tuve que modificar la forma en la que se hacen las pruebas en la asignatura con el fin de poder solucionar los errores que presentaban mis tests. Los pasos que seguí son los siguientes:

1. Establecer mensajes de error personalizados:

Esto fue una modificación que ya implementamos en anteriores entregas y que nos servía para saber qué error era el que provocaba el fallo. Los Assert disponen

de dos implementaciones, una donde sólo se evalúa la condición y otra dónde además, puedes añadir una cadena de texto explicativa. Con el fin de mostrar errores personalizados en las vistas, recomendamos que estas cadenas de texto se almacenen en los archivos messages.properties siguiendo siempre un mismo patrón.

El patrón elegido por mi equipo es este: “message.error.[entidad].[descripción]”. Aunque el almacenaje de estos mensajes en los messages.properties no es necesario para que funcione el método, sí que será necesario que todos sigan el mismo patrón.

Un ejemplo sería el siguiente:

```
Assert.isTrue(user.getBirthDate().before(new  Date()),“messa-
ge.error.user.birthDate.future”);
```

2. Añadir un identificador para cada test:

En el driver, tenemos una matriz de objetos que incluye todos los datos necesarios para los tests. Una opción recomendable sería añadir una columna (preferiblemente la primera) que incluyese algún identificador. Por identificador, podemos entender una cadena de texto única que se le asocie a cada test con el fin de que sepamos en todo momento qué test se está ejecutando. Un ejemplo de identificador sería: “testSaveFromCreate1”. No hace falta añadir ninguna referencia a la entidad testada ya que los tests están agrupados por entidades de dominio.

3. Modificar el bloque try-catch del template:

Una vez que tenemos los errores homogéneos, tenemos que capturar estos errores. Para ellos, modificaremos la template de la siguiente forma:

1. Fuera del bloque try-catch, añadimos una variable que almacene el error a capturar. En este ejemplo se llamará `messageError` y será de tipo `String`. Esta variable se puede declarar como nula o se le puede añadir un mensaje genérico.

2. Modificaremos el tipo capturado por el catch para que sea `Throwable`. En este problema, la variable se llamará `oops`, como se hace en los controladores. Esta variable dispone de un método llamado `getMessage()` que permite recuperar el mensaje asociado al assert.

3. Dentro del catch, preguntaremos si el mensaje del assert incluye parte del patrón. En nuestro ejemplo, comprobamos que el error capturado contenga “`message.error`”. La formulación de un patrón fijo para todos los errores personalizados del sistema es fundamental para que este sistema funcione.

4. En el caso en el que se cumpla la condición, `messageError` se actualizará con el mensaje del Assert que haya provocado el fallo.

4. Modificar el método checkException de AbstractTest:

Una vez que se ha ejecutado el bloque try-catch, se llama al método `checkException` de AbstractTest. Este método recibe dos excepciones: La esperada y la capturada. El problema viene cuando no disponemos de excepciones personalizadas. Un fallo `IllegalArgumentException` no aportará ninguna información, ya que todos los errores de los Assert son de este tipo.

Las modificaciones que propongo realizar son las siguientes:

1. Añadir dos parámetros más de entrada: El identificador del test y el mensaje de error personalizados capturado en el bloque try-catch.
2. En el caso en el que las excepciones difieran, es decir, que el test sea erróneo, habría que añadir tanto el identificador como el mensaje personalizado a la nueva excepción que se va a generar.

Una vez que se realizar estos cambios, cada vez que haya un error en cualquiera de los casos de prueba, podremos identificar qué test es el que falla y aportaremos algo de información sobre el propio error en la consola de JUnit.