Universidad Rey Juan Carlos
Escuela Técnica Superior de Ingeniería Informática
Grado en Ingeniería Informática (Tercer curso)
Ampliación de Ingeniería del Software
Autor: Gustavo Andrés Marrero Tovar

Práctica 1 AIS: Pruebas y calidad de Software

Móstoles, Marzo de 2021

Test unitarios (WebServiceTest)

Antes que nada tuve que implementar test unitarios para comprobar la correcta creación y eliminación de libros usando **BookService**, así como la notificación de esas acciones.

Para ello he creado la clase de prueba **BookServiceTest**, la cual tiene 4 atributos, un tipo **Book** (book), un tipo **BookRepository** (repository), un tipo **NotificationService** (notification) y un tipo **BookService** (service).

A continuación, he creado un método para inicializar esos cuatro atributos **(setUp())** utilizando mocks, salvo para service, pues es del tipo que queremos probar. Service se inicializa utilizando un **BookRepository** y un **NotificationService** mockeados, los mismos que representan los atributos de la clase de prueba. Este método se vale de la anotación **@BeforeEach** para ejecutarse antes de todos los métodos de prueba.

A partir de ahora se explican los métodos de prueba, los cuales he dividido en cuatro diferentes, cada uno con su anotación **@Test.**

1) **whenSaveBook_LaunchNotification:** Este método verifica si se lanza una notificación al guardar un libro utilizando **BookService**. Además, verifica que el contenido de esa notificación es el correcto. Para ejecutarse necesita mockear el comportamiento del repositorio al guardar (método save) y el comportamiento de un libro al solicitar su título (método getTitle).

2) **whenSaveBook_SaveInRepository:** Este método verifica si al guardar un libro utilizando BookService, este se guarda realmente en el repositorio. Para ello, se mockea el comportamiento de los métodos getId() y getTitle() de un libro, para que nos devuelvan un Long y un String. También hay que mockear los métodos save y existsById del repositorio, para que nos devuelva lo deseado en base a los valores del tipo Book. Se ejercita el SUT y al final se comprueba si el valor retornado al llamar a **BookService.exist()** es el mismo que el que devuelve el repositorio mockeado.

3) **whenErraseBook_LaunchNotification:** Este método verifica que al eliminar un libro, se lanza una notificación, para ello solo mockea el método getId de un libro, ejercita el SUT y verifica que el valor del getId mockeado es el mismo de la notificación.

4) **whenErraseBook_ErraseInRepository:** Por último, este método comprueba que al borrar un libro del BookService, este deja de estar en el repositorio. Para ello mockea los métodos getId del tipo Book (que devuelve un Long), como el existsById del tipo Repository (el cual tendrá que devolver FALSE, dado que el libro ya no tendría que estar disponible). Luego de eso se ejercita el SUT y se verifica que no existe el libro con esa Id en el BookService.

Test de la ApiREST (ApiRestTest)

Para la **ApiREST** utilicé sólo dos métodos, sin incluir el setUp del puerto en el que se trabaja, los cuales revisan que un libro se puede recuperar usando un **GET** si este libro se ha creado, y si al borrar un libro, éste se vuelve inaccesible, por lo que al intentar acceder a este utilizando un **GET** debería mostrar el código de estado **404** (not found).

1) **whenAddBook_ThenIsRetrievable:** Este método verifica que al crear un libro, se puede recuperar utilizando un **GET,** para ello guarda como variable una respuesta, que contiene una petición en la cual se postea un libro (con su respectivo título y descripción) en la api. Asimismo, se guarda en una variable de tipo entero el valor identificativo de ese libro, que es el que determina la dirección en la que es guardado **(idBook).** En la parte del when se procede a realizar la petición **GET** con la dirección que venga dada por el id del libro y por último se comprueba que el elemento se ha recuperado correctamente. que su id coincide y que su título también.

2) **whenDeleteBook_ThenIsNotRetrievable:** Este es el caso opuesto al anterior, se quiere verificar que si una vez que se ha borrado un libro, este deja de ser accesible, para ello, necesitamos primero crear un libro, que posteriormente se borrará. La creación de ese libro sigue un proceso análogo al del método anterior, la diferencia es que ahora, en vez de realizar una petición **GET**, borraremos el libro con el método **delete**. Una vez hecho esto, se verifica que se ha borrado (el código de estado 200 indica si el recurso se ha borrado correctamente) y procedemos a realizar lo que pide el enunciado; comprobar que ya no se puede acceder al libro. Para ello intentamos realizar un **GET** del libro en cuestión, y la aserción en este caso debe ser que no se puede encontrar el libro, lo que significa que el código de estado debe ser **404.** Esto es lo que comprueba el test. Si se cumple, el test pasa. Adicionalmente, para confirmar que el test estaba bien hecho hemos variado las condiciones, cambiando el libro que se borraba o el código de estado que debía aparecer.

Test de la interfaz web (Web_IntefaceTest)

Para el desarrollo de los test de la interfaz web he utilizado **Selenium**. La forma en que estos se realizan es muy similar al ejemplo visto en clase de la web de anuncios, las aserciones se parecen bastante y los elementos de la web también.

En este test he utilizado dos métodos de prueba, uno para añadir un libro nuevo y verificar si se ha creado, y el segundo para borrar un libro y comprobar que ya no existe. Estuve tentado a crear un método que añadiera un libro que pudiesen aprovechar ambos métodos para reutilizar un poco el código, pues para ambos métodos se necesita crear un libro, solo que el segundo lo borraría inmediatamente después de crearlo. Pero por cuestiones de modularización y claridad he decidido mantener ambos test como bloques independientes de otros métodos. Dicho esto, procedo con la descripción del test:

Para empezar se requiere un puerto (para no tener que estarlo especificando todo el rato), que lleva la anotación **@LocalServerPort** y un **WebDriverManager,** los cuales los he incluido como elementos internos de la clase de prueba.
Al iniciar, se realizan tres métodos auxiliares:

- **setupClass():** Que realiza la configuración automática del driver de Firefox, para no tener que estar descargando el controlador ni tener que manejarlo con dificultad.
- **setupTest():** que inicializa el driver para Firefox cada vez que utilizamos un método de prueba (en un principio iba a ser un driver de Chrome, pero al ejecutar me daba un error que indicaba que la versión no estaba soportada, por lo que lo he cambiado a Firefox).
- **teardown():** Que cierra los navegadores abiertos por cada test al finalizar la prueba.

Asimismo, también están los dos métodos de prueba en sí, los cuales se explican detalladamente a continuación:

1) **whenAddBook_ThenBookIsCreated():** Primero que nada se usa el driver para acceder a la web creada, luego este método inicializa dos strings que se usarán como nombre y descripción de un libro que vamos a crear. Como se quiere añadir un nuevo libro se busca el botón **New book** utilizando xpath y pasando como argumento el valor **("//button")**, pues es el único botón de la pantalla principal. Al acceder, la siguiente página tiene dos campos de texto, identificados con los nombres **"title"** y **"description",** que se rellenan con los Strings que se han inicializado previamente utilizando el método **sendKeys(),** antes de hacer click en el botón con **id = Save.** Hecho esto, el libro ya se ha creado. Ahora se llega a otra pantalla que tiene tres botones. Lo que se quiere es volver al menú principal, así que utilizando xpath, se hace click en el botón que contenga el texto **"All Books".** Me costó algo de trabajo encontrar una forma de acceder al botón, pues no tenía nombre ni id y tuve que empaparme un poco sobre el uso de xpath. Una vez de regreso a la pantalla principal se realiza la prueba en sí, la aserción, la cual comprueba si alguno de los enlaces que tiene la página principal concuerda con el String que identifica al nombre del libro que hemos creado, pues los libros están almacenados como una lista de enlaces. Si es así, el test pasa.

2) **whenDeleteBook_ThenBookNoLongerExists():** La primera parte de este test es muy similar al test anterior, pues hay que crear un libro. Una vez creado el procedimiento cambia un poco.
Hay dos opciones en este punto, se puede retornar a la página principal para luego borrar el libro, o por otro lado, ya que al crear el libro nos encontramos precisamente en la página en la cual lo podemos borrar, se borra directamente sin antes volver al menú de inicio. Por mantener corto el código he decidido decantarme por esta segunda opción.

Dicho esto, una vez que se ha creado el libro, se busca dentro de la web el botón que contiene el texto **"Remove"** y se le hace click. Esto nos llevará a la página de inicio, en la cual realizaremos la aserción directamente. La aserción comprueba que al intentar encontrar un libro cuyo título coincida con el String utilizado para crear el libro que se ha borrado, se lanzará una excepción del tipo **NoSuchElementException,** pues no se puede encontrar un título si se ha borrado el libro asociado a este. Si esto ocurre, el test pasa.

SonarQube y evaluación de vulnerabilidades y Code Smells

Luego de iniciar y cargar el proyecto en el sonarQube, se realizó el escaneo correspondiente dando como resultado **4 vulnerabilidades y 21 code smells.**

Vulnerabilidades

Con respecto a las vulnerabilidades, la que he encontrado es la misma repetida 4 veces, dos en cada controlador (ApiREST y web). SonarQube indica que **las peticiones HTTP no deben manejar entidades de una base de datos**, en este caso, refiriéndose a los libros, pues si esto ocurre, nada evita que se pueda acceder a atributos privados utilizando peticiones web. Esto significa que **los métodos que utilicen mapping no pueden recibir libros como argumento.**

La solución sugerida es utilizar clases intermedias para proteger a la clase **Book** original. Así lo he solucionado, he creado una clase **BookDTO**, como lo ha sugerido SonarQube, que es una especie de libro falso cuyos atributos concuerdan con el tipo Book, salvo por el id, el cual no posee. También le he añadido getters y setters.

Lo siguiente ha sido cambiar los métodos del controlador de la ApiREST y del controlador web que recibían como argumento un libro; estos son: **createBook, updateBook (apiREST), newBookProcess y editBookProcess (web)**. Ahora los métodos recibirán un **BookBTO** como argumento e iniciarán dentro de sí los valores del libro real usando sus propios setters y los getters del **BookBTO.**

No fue fácil dar con la solución, tuve que investigar por mi cuenta sobre conceptos como POJO y DTO (que son objetos que sirven para enviar información entre capas en Java para intentar evitar la desencapsulación), ver algunos vídeos y tutoriales, además de por supuesto, analizar con calma y tiempo la información obtenida con SonarQube, que fue determinante a la hora de pensar en una solución:

https://www.arquitecturajava.com/pojos-vo-dto-y-javabeans/

https://es.stackoverflow.com/questions/334383/en-java-cual-es-la-diferencia-entre-pojo-javabeans-business-object-value-ob

https://www.generacodice.com/es/articolo/79997/Difference-between-DTO%2C-VO%2C-POJO%2C-JavaBeans#:~:text=DTO%3A%20%22%20Objetos%20de%20transferencia%20de,no%20es%20un%20objeto%20especial.

Code Smells

De los 21 code smells **11** recomendaban la **eliminación del modificador de acceso "public" en las clases y los métodos de prueba**, lo cual ha sido muy fácil de corregir, simplemente eliminando dichos modificadores y dejando el acceso por defecto.

El siguiente code smell nos indicaba que se podía cambiar el **string "books"** que se repetía 4 veces en **BookServiceTest** por una constante de tipo string que a la que se hace referencia en cada aparición de dicho string. A pesar de no formar parte de los test he decidido hacer caso al SonarQube pues es un smell muy fácil de corregir. He añadido la constante **private final String BOOKS = "books"** en el código e instanciado la constante en cada aparición del string "books".

Otros **6** code smells aparecían por el uso de **Thread.sleep()** en el código, los he eliminado y cambiado por métodos que utilizan wait para que esperen a que se cargue cada parte de la página: por ejemplo, en lugar de usar **Thread.sleep(1000)** luego de hacer click en el botón de **"New Book"** he creado una variable wait y le he indicado al driver que espere hasta que se cargue la cabecera de la página. Los seis code smells tienen una solución similar a esta:

```
WebDriverWait wait = new WebDriverWait (driver, 30);
wait.until(presenceOfElementLocated(By.xpath("//h2[contains(text(),'NewBook')]")));
```

A continuación procedí a corregir **2** Code Smells en la clase **BookServiceTest**, que recomiendan el uso de las aserciones siguientes:

- **assertThat(service.exist(10L)).isTrue();**
  en lugar de:
  assertThat(service.exist(10L)).isEqualTo(true);

- **assertThat(service.exist(30L)).isFalse();**
  en lugar de:
  assertThat(service.exist(30L)).isEqualTo(false);

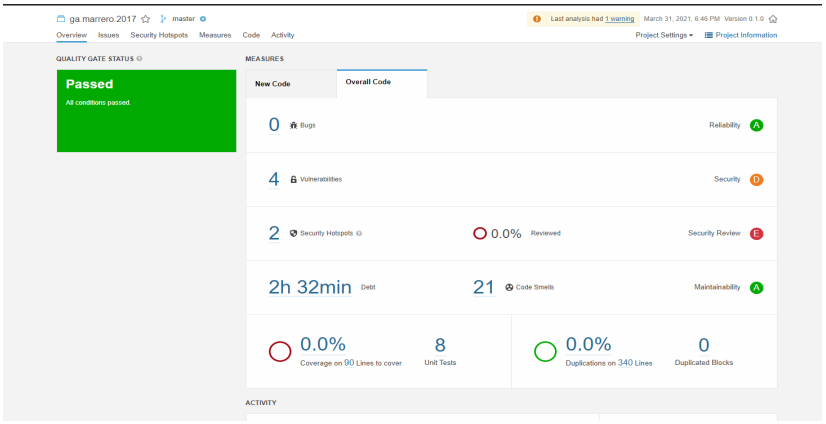
  Simplemente he cambiado el código de las dos aserciones y problema resuelto.

El último code smell corresponde a la siguiente aserción del test de la interfaz web:

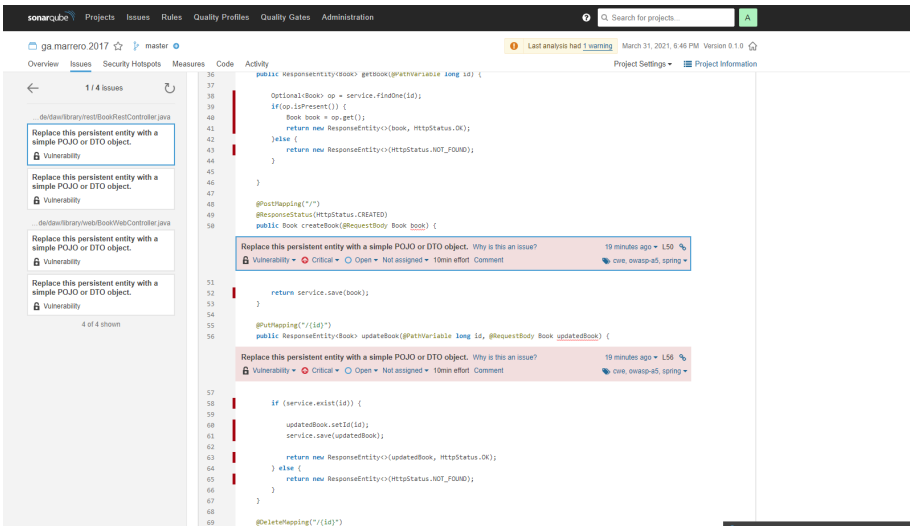
```
assertThrows(NoSuchElementException.class, () -> {
        driver.findElement(By.linkText(newBookName));
});
```

SonarQube indica que hay dos métodos que pueden lanzar la excepción, tanto linkText() como findElement() y recomienda solo incluir uno de estos métodos dentro del test donde se comprueba el lanzamiento de la excepción. Dado que los métodos están anidados y no se me ha ocurrido una manera eficaz de separarlos (no se puede aislar el findElement del linkText con variables intermedias) y tomando en cuenta que es el método findElement el que lanzará la excepción, he marcado este smell como un falso positivo.

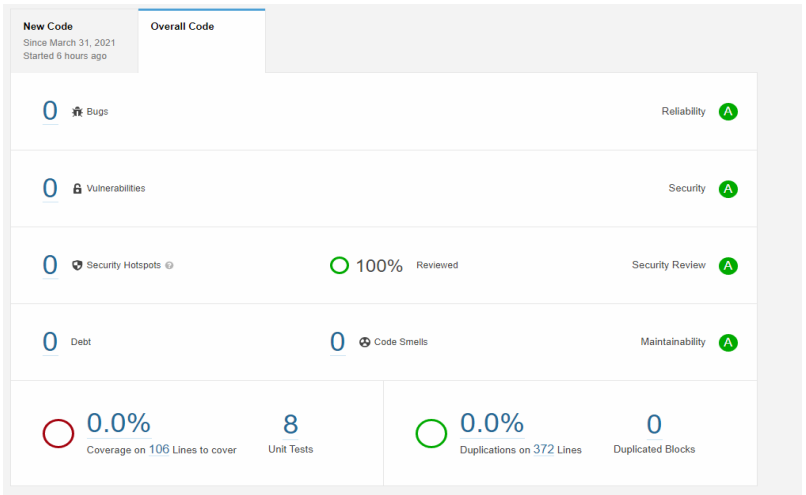
Anexos



Img 1: Pantalla SonarQube previa a la corrección



Img 2: Vulnerabilidades detectadas



Img 3: Pantalla SonarQube posterior a la corrección