

Universidad de Sevilla

Escuela Técnica Superior de Ingeniería Informática

Entregable 1: Introducción

Informe de conocimiento previo de testing de un WIS



Grado en Ingeniería Informática – Ingeniería del Software

Diseño y Pruebas 2

Curso 2021 – 2022

Grupo: E6.02

Repositorio: <https://github.com/INoelia/Acme-Toolkits>

Nombre: Cruz Duárez, Sara

Email: sarcrudua@alum.us.es

Nombre: Delgado Sánchez, José María

Email: josedelsan9@alum.us.es

Nombre: López Durán, Noelia

Email: noelopdur@alum.us.es

Nombre: Molina Arregui, Rosa María

Email: rosmolarr@alum.us.es


Nombre: Nadal García, Ricardo

Email: ricnadgar@alum.us.es

Nombre: Varela Soult, Carlos

Email: carvarsou@alum.us.es

Fecha: Sevilla, Febrero 23, 2022.

	<p>Diseño y Pruebas 2</p> <p>Entregable 1: Introducción</p>
---	---

Índice

1. Resumen ejecutivo	3
2. Tabla de revisiones	3
3. Introducción	3
4. Contenido	4
4.1 Tipos de testing	4
4.2 Conceptos básicos sobre el testing unitario	4
4.3 Buenas prácticas	4
4.4 Aplicación de JUnit en pruebas de servicio	5
4.5 Vocabulario en las pruebas de controlador	5
4.6 Aplicación de Mockito en pruebas de controlador	6
5. Conclusión	6
6. Bibliografía	7



1. Resumen ejecutivo

Una vez sabemos cómo desarrollar un WIS basado en la arquitectura MVC y Spring como framework, debemos profundizar en el cumplimiento de los requisitos de verificación y validación para conocer en todo momento si lo desarrollado está correctamente implementado de acuerdo a las necesidades del cliente. De manera paralela a la construcción del WIS, debemos probar sus funcionalidades e identificar sus defectos para corregirlos.

El testing, por consiguiente, se divide en varios tipos dependiendo de su granularidad. Nuestro conocimiento previo es, mayoritariamente, sobre testing enfocado a pruebas unitarias, aunque también tenemos lecciones breves sobre testing end-to-end y pruebas de aceptación con Spring.

2. Tabla de revisiones


Fecha	Nº de revisión	Descripción
28/02/2022	1	Durante una reunión, todos los miembros del grupo leímos el documento, cambiamos aquello en lo que no estábamos de acuerdo y dimos por finalizado el documento.

3. Introducción

En el presente informe se expone detalladamente información de interés para el estudio del testing unitario. En el orden indicado a continuación, se explican los distintos tipos de pruebas que existen y el enfoque de las mismas, se describen conceptos a tener en cuenta sobre los tests unitarios para familiarizarse con el vocabulario, se especifican un conjunto de buenas prácticas para minimizar riesgos, se argumenta sobre la implementación de pruebas de servicio haciendo uso de la biblioteca de JUnit en Spring y se introduce un breve vocabulario relacionado con las pruebas de controlador, además de la aplicación de Mockito como framework para la creación de estas pruebas de manera automatizada.

En síntesis, a continuación tenemos los apartados a detallar:

- Tipos de testing
- Conceptos básicos sobre el testing unitario
- Buenas prácticas
- Aplicación de JUnit en pruebas de servicio
- Vocabulario en las pruebas de controlador
- Aplicación de Mockito en pruebas de controlador

	<p>Diseño y Pruebas 2 Entregable 1: Introducción</p>
---	--

4. Contenido

4.1 Tipos de testing

Podemos diferenciar entre varios tipos diferentes de testing, de menor a mayor grado de complejidad, explicados brevemente a continuación:

- Tests unitarios: Sirven para comprobar que una unidad de código o “test unit” funciona correctamente, es decir, se hacen pruebas sobre métodos o funciones de las clases de la aplicación para ver su comportamiento.
- Tests de integración: Prueba la comunicación entre varias clases o módulos, incluso la interacción del sistema con el entorno de producción.
- Tests end-to-end: Como su nombre sugiere, se encarga de probar el software completo desde la simulación de la experiencia del usuario para validarlo.
- Tests de aceptación: Luego de resultar exitosos los casos de prueba anteriores, el cliente prueba el software y verifica que cumple con sus expectativas. Estas pruebas son de tipo caja negra, pues el cliente solo atiende a los resultados y no se preocupa por el comportamiento interno.
- Tests exploratorios: Sirven para la optimización de los casos de prueba anteriores a través del aprendizaje continuo del tester sobre el sistema que se trabaja.


4.2 Conceptos básicos sobre el testing unitario

Como se ha indicado anteriormente, nuestro conocimiento actual está enfocado en las pruebas unitarias. Luego, revisaremos algunos conceptos generales sobre testing así como otros necesarios para desarrollar testing unitario.

- SUT: System under test, es aquella clase Java cuyas funcionalidades queremos probar.
- Bug: Error en el código que a simple vista puede ser imperceptible de detectar.
- Failure: Manifestación del bug, nos indica que existe un bug.
- Debugging: Proceso de búsqueda de bugs para su corrección.
- Fixture: Conjunto de objetos que permiten realizar testing.
- Test case: Método de la clase de test que prueba una unidad de test.
- Test class: Clase que contiene todos los casos de test.
- Test suite: Conjunto de clases de test.

4.3 Buenas prácticas

- Principio FIRST: Un test unitario debe cumplir las siguientes siglas.
 - ◆ **Fast**: Las pruebas deben ejecutarse rápidamente.
 - ◆ **Isolated**: Deben ser independientes entre ellas.

	<p>Diseño y Pruebas 2</p> <p>Entregable 1: Introducción</p>
---	---

- ◆ **Repeatable:** El paso del tiempo no afecta al resultado de las pruebas.
- ◆ **Self-validating:** Cada prueba debe autovalidarse a sí misma.
- ◆ **Timely:** Deben desarrollarse pruebas en paralelo a la implementación de la lógica de negocio.

→ Principio DRY:

- ◆ **Don't Repeat Yourself:** Persigue la reutilización del código sobre su duplicación para mantener una alta cohesión y bajo acoplamiento entre las pruebas.

→ Parametrizar tests unitarios: Debemos repasar la misma unidad de código con diferentes valores para incrementar la cobertura de los datos y reutilizar código.

→ Assertions fluidas: Spring incluye una librería llamada AssertJ que nos permite esquematizar la estructura de las assertions para identificar fácilmente lo que queremos comprobar.

→ Enfocar tests unitarios: Es recomendable que cada unidad de test se enfoque en realizar una única tarea para garantizar facilidad en el entendimiento de aquello que se prueba, así como evitar efectos colaterales en el caso de múltiples escenarios de prueba.

→ Mantener clara la causa y el efecto: Es imprescindible que cada assert se encuentre próximo a aquellos objetos sobre los que trabaja, es decir, debemos garantizar que el programador obtenga una lectura estructurada y coherente de los datos para minimizar el coste de mantenimiento.

4.4 Aplicación de JUnit en pruebas de servicio

JUnit es un framework Java para implementar tests en clases Java y está basado en anotaciones. Para utilizar JUnit, lo importamos con `"import org.junit.jupiter.api.Test"` y para cada test escribimos la anotación `@Test`, de tal manera que indicamos que el método referenciado contiene un test. Además, existen más anotaciones como `@Before` o `@After` para indicar en qué orden queremos que se ejecuten los tests.

En general, la estructura del test sigue el patrón Arrange-Act-Assert:

- Arrange: Sinónimo de "fixture". Se inicializan instancias de objetos.
- Act: Se ejecutan funciones sobre el sujeto bajo prueba.
- Assert: Verificamos si se devuelven los resultados esperados.

4.5 Vocabulario en las pruebas de controlador

La simulación de las clases se hacen mediante dobles de test, es decir, objetos que



Diseño y Pruebas 2

Entregable 1: Introducción

se hacen pasar por el objeto real en un test. Utilizándolos conseguimos una versión más rápida y simple, así como evitamos efectos secundarios y configuraciones complicadas.


- Dummy: Son objetos que llaman a otros métodos pero no se utilizan.
- Seam: Doble de test utilizado para aislar comportamientos del SUT. Los tipos más comunes de seam son stubs, mocks y fakes.
- Stub: Parecido a un dummy pues sus métodos no hacen nada. No obstante, devuelven respuestas enlatadas necesarias para ejecutar nuestro test bajo ciertas condiciones. Verificamos el estado del SUT para ver si pasa la prueba.
- Spy: Stub que permite verificar el uso que se hace de un objeto real, como el número de veces que se ejecuta un método o los valores que se le pasan.
- Mock: Stub en el que sus métodos esperan recibir unos valores y devuelven una respuesta. Se utiliza para comprobar interacciones entre objetos, sin embargo, el uso excesivo de mocks así como la desincronización de los mismos con las implementaciones reales pueden no garantizar que el SUT esté funcionando correctamente.
- Fake: Implementación ligera de una API que se comporta como la real pero no está lista para producción. Se utiliza cuando la API real no se puede usar para las pruebas por su lentitud en el acceso a la red.

4.6 Aplicación de Mockito en pruebas de controlador

Mockito es una biblioteca Java que permite simular el comportamiento de una clase controlador de forma dinámica. De esta forma nos aislamos de las dependencias con otras clases y sólo testamos la funcionalidad concreta que queremos. Para utilizar Mockito, lo importamos con `import org.mockito.Mockito`, especificamos la clase sobre la que simulamos con `@WebMvcTest(value=ClaseController.class)`, instanciamos los servicios con `@MockBean` y para cada prueba anotamos `@WithMockUser`.

5. Conclusión

Hemos observado que existen diferentes tipos de testing en una escala de granularidad, donde cada uno tiene una importancia esencial en el producto software que se desarrolla. Desde el enfoque del testing unitario, a pesar de que sea el menos complejo de todos, no quiere decir que sea menos importante, pues cada módulo de la aplicación se prueba

	<p>Diseño y Pruebas 2</p> <p>Entregable 1: Introducción</p>
---	---

individualmente, es decir, cada método de la aplicación debe tener asociado una prueba que lo respalde y verifique si cumple con aquello que quiere realizar.

Además, mediante el uso de métricas podemos conocer la cobertura de nuestros tests unitarios para la mejora continua de los mismos en el caso de que, por ejemplo, ciertas ramas de un loop no sean recorridas en ningún momento, por lo que no sabemos si funciona adecuadamente cierta parte de nuestro código.

Por otra parte, este enfoque como cualquier otro debe estar respaldado con una documentación del software para que el cliente pueda saber en todo momento cómo se comporta y cómo se utiliza adecuadamente la aplicación sobre la que está trabajando.

En definitiva, las empresas desarrolladores de software deben garantizar que el producto vendido a sus clientes sea de calidad y esté libre de bugs. Por tanto, una detección temprana de errores a nivel de código va a minimizar el gasto en etapas futuras como la de mantenimiento.

6. Bibliografía

<https://blog.softtek.com/es/testing-unitario>

<https://junit.org/junit4/cookbook.html>

T7 y T11 de la asignatura de DP1