DP2 2021-2022

D01- Introducción

Informe de testeo en arquitecturas de un WIS

URL Github: https://github.com/migrivros/Acme-One

Miembros:

- Miguel Ángel Gómez Gómez (miggomgom1@alum.us.es)
- Dámaris Gómez Serrano (damgomser@alum.us.es)
- Mariano Martín Avecilla (marmarave@alum.us.es)
- Iván Moreno Granado (ivamorgra@alum.us.es)
- Miguel Ángel Rivas Rosado (migrivros@alum.us.es)
- Rafael Sanabria Espárrago (rafasana9@gmail.com)

GRUPO E3.04

Versión 1.0

22-02-2022

Tabla de contenidos

Historial de versiones

Fecha	Versión	Descripción de los cambios	Sprint
22/02/2022	V1.0	• Creación del documento • Introducción • Resumen ejecutivo • Vocabulario empleado en el diseño de pruebas • Tests unitarios en arquitecturas de capas • Tipos de pruebas • Estructura de una prueba • Principios • Buenas prácticas • Testeo sobre una arquitectura WIS por capas • Conclusiones	1

Introducción

Este documento presenta los conocimientos previos del equipo de trabajo sobre el diseño y pruebas de un sistema software. Ha sido redactado por el equipo de trabajo pensando en los conocimientos adquiridos previamente en la asignatura de Diseño y Pruebas I, que ha sido superada por todos y cada uno de los miembros del equipo.

Tras un debate previo se han decidido incluir no todos los conocimientos que impartían asignaturas previas a esta sino solo los que verdaderamente el equipo de trabajo ha aprendido e interiorizado.

A continuación, se muestra un resumen ejecutivo del informe, el contenido del mismo en el que se especifica todo lo aprendido anteriormente, las conclusiones y la bibliografía.

Resumen ejecutivo

El objetivo de este documento es detallar lo aprendido anteriormente en el proceso de diseño y creación de pruebas sobre un sistema software.

Para ello se proporciona a continuación los conocimientos teóricos, desde un punto de vista de mayor a menor abstracción, de todos los recursos en los que nos apoyamos para realizar desde un punto de vista práctico la realización de las pruebas.

Contenido

Conocimiento previo sobre el diseño de pruebas

Vocabulario empleado en el diseño de pruebas

Un bug es un defecto de en el software que se manifiesta en ejecución de una prueba. Un fallo es el resultado de ese bug. Por otro lado, el proceso de Debugging es el encargado de depurar un error con la finalidad de eliminarlo. El SUT es el sistema bajo testeo y el fixture es el sistema que realiza las pruebas.

Tests unitarios en arquitectura en capas

En el caso de las pruebas unitarias, siendo las estudiadas anteriormente, el sujeto bajo prueba es una unidad (unit). La unidad en software son los métodos o funciones. Su granularidad es la más baja de todos los tipos de pruebas, esto quiere decir que el alcance de la prueba es más estrecho. La principal característica de los tests unitarios es su rápida ejecución

Tipos de pruebas

Podemos identificar, según el resultado que se espera de una prueba, dos tipos de prueba:

- Prueba unitaria positiva: Comprueba el "camino feliz". Esto quiere decir que el comportamiento de la prueba es el normal, cuando se hace un buen uso del sistema.
- Prueba unitaria negativa: Comprueba el comportamiento anormal cuando se hace un mal uso del sistema. Entre otras comprobaciones, verifica que se maneja los errores

Según el alcance (granularidad) de la prueba, podemos diferenciar cuatro tipos de pruebas:

- Pruebas unitarias, explicadas anteriormente
- Pruebas de integración, se asegura de que todos los módulos de la aplicación estén bien integrados
- Pruebas End-to-end, permiten comprobar posibles fallos en la interfaz de usuario
- Pruebas de aceptación, donde el cliente es parte fundamental, siendo el encargado de verificar las diferentes funcionalidades e interfaces.

Según el conocimiento de la implementación del software, se pueden distinguir dos tipos de pruebas:

- Pruebas de caja blanca: El encargado de diseñar la prueba tiene conocimiento sobre la implementación llevada a cabo de la funcionalidad a testear.
- Pruebas de caja negra: El encargado de diseñar la prueba no tiene conocimiento sobre la implementación llevada a cabo de la funcionalidad a testear.

Estructura de una prueba

La estructura de una prueba viene dada en 3 pasos:

- Arrange/Fixture: Se encarga de configurar los datos que se utilizará en la prueba
- Act: Se encarga de ejecutar la prueba
- Assert: Confirma y verifica que se devuelven los resultados esperados

Principios

- DRY (Don't Repeat Yourself): Este principio fomenta la reutilización del código frente a la duplicación del mismo. Recomienda extraer funcionalidad a métodos auxiliares.
- DAMP: Fomenta la legibilidad sobre la singularidad. Para ello, propone crear un código que sea lo suficientemente descriptivo con el fin de entenderlo lo más rápido posible. Puede dar lugar a redundancia de código, pero hace que las pruebas sean más fáciles de comprender que sean correctas.

Buenas prácticas para las pruebas

Para lograr un buen testeo de nuestro sistema, se recomienda realizar las siguientes prácticas:

- Parametrización, siempre que sea posible, las pruebas unitarias.
- Mantener las pruebas unitarias enfocadas e independientes
- Mantener claro la causa y el efecto de cada test

Testeo sobre una arquitectura WIS por capas

En la asignatura DP-1 aprendimos a hacer uso de la arquitectura por capas, utilizando el patrón Modelo-Vista-Controlador. Para ello, nos enseñaron también a hacer los tests unitarios de controladores y servicios.

Tests de servicios

Los tests de servicios nos permite verificar si la capa lógica de negocio cumple con los requisitos establecidos y funciona correctamente. Estos tests pueden ser transacciones, esto significa que la operación debe tener éxito o fracasar como unidad completa, nunca parcialmente. Estos tests deben ser anotados como @Transactional.

Además, vimos cómo tratar la cobertura, mediante varias estrategias, dependiendo de la estructura del código, diferenciando bien secuencias, condicionales o bucles, o el tipo de datos que se usa (rangos, opcionales o colecciones).

Para ello, se ha estudiado la creación de tests parametrizados unitarios, permitiendo así la reusabilidad de código para hacer el testing.

Por último, se ha visto cómo manejar el flujo de asserts mediante el uso de AssertJ (ya que estaba importado automáticamente con spring-boot-startrer-test).

```
@Test
public void testdeleteIslandById() {
    int beforecount = IslandService.islandCount();

    IslandStatusService.deleteIslandStatus(newstatus.getId());
    IslandService.deleteIsland(newisland);

    int aftercount = IslandService.islandCount();

    assertEquals(aftercount, beforecount-1);
}
```

Para esta prueba de servicio de ejemplo de lo que se hizo en DP1, se ha creado un @BeforeEach (arrange) en el que se crea una nueva isla, se salva, se crea una IslandStatus (representa el estado de la isla en la partida, en un turno) y se le coloca una carta. Por último se guarda el estado de la isla (IslandStatus).
En el act lo que hacemos es en primer lugar, contar el número de islas que hay. Después, borramos el estado de la isla y la isla en concreto que pertenece a ese estado y por último contamos de nuevo el número de islas.
El assert comprueba que ahora el número de islas que había antes menos 1 es igual al número de islas que hay después de borrar una de ellas.

Tests de controladores

Estos tests nos permiten verificar que el controlador funciona correctamente. A diferencia de los tests de servicios, en este caso comprobamos si el controlador devuelve la ruta que deseamos, si se pasa bien el atributo del modelo a la vista y el estado de la respuesta al cliente.

Los tests de controladores deben ser pruebas solitarias. Esto quiere decir que todos los colaboradores del SUT deben ser dobles, no objetos reales. Con ello, se consigue un aislamiento del test permitiendo que éste sea independiente de la base de datos. Una prueba doble es un seam (lugar donde puedes alterar comportamientos en tu programa sin llegar a editarlos) que aíslan comportamientos del SUT de los comportamientos de otros colaboradores.

Por tanto, nosotros aprendimos las grandes diferencias entre hacer una prueba de servicio, donde se accedía a dicho elemento de manera directa (pruebas sociables), estableciendo una comunicación real con otros colaboradores (otros servicios) frente a una prueba de controlador, que viene dada por colaboradores (servicios) fictios, y, por tanto, se debe detallar en el mismo arrange qué es lo que devuelve cada método que se encuentra en cada uno de los colaboradores.

Para ello hemos trabajado con Mocks, mediante el framework de Mockito, que permitía probar las interacciones entre objetos y verificar el comportamiento del SUT para decidir si pasa o no la prueba. Es útil su uso cuando no hay cambios de estado en el SUT.

```
@Test
@WithMockUser(value="spring", authorities=("admin"))
void usersListPageableTest() throws Exception{
    mockMvc.perform(get("/admin/page/users?page=1")).andExpect(status().isOk())
        .andExpect(model().attributeExists("users"))
        .andExpect(model().attributeExists("paginas"))
        .andExpect(view().name("authorities/usersList"));
}
```

En este ejemplo de test con escenario positivo que se hizo en el proyecto de la asignatura DP1 se pone a prueba el controlador de paginación que envía a la vista el listado de usuarios por páginas. El arrange de la prueba hace uso de la creación de una “PageRequest” para la posterior llamada al método “findByUsernamePageable” para obtener los usuarios en paginación, ya que en el mismo controlador se hace uso de ello. Para ello, hemos comprobado, accediendo como administrador a la primera página, que el controlador llama a la vista llamada “usersList” y que envía como atributos a dicha vista las páginas y los usuarios (assert).

Conclusiones

En general, en el documento se detalla el conocimiento obtenido en asignaturas anteriores, como Diseño y Pruebas 1, sobre el proceso de diseño de las pruebas sobre un sistema de información web.

En general, se ha trabajado sobre el diseño de dos componentes esenciales en el modelo de arquitecturas de capas: controladores y servicios. Hemos aprendido las similitudes y diferencias en el diseño de las pruebas en general para ambos componentes con la ayuda de diferentes frameworks (como Mockito) y bibliotecas (Junit), su impacto en la aplicación y el comportamiento del sistema en cada prueba que realizamos.

Se ha enfocado profundamente en el aprendizaje de pruebas unitarias, sus características y objetivos principales con el objetivo de lograr la máxima optimización del código. Además, se ha estudiado previamente la comprobación del manejo de errores, ya que era un factor fundamental en nuestro proyecto.

Bibliografía

- Diapositivas de los temas 8 y 11 de la asignatura DP-1.
- Apuntes propios de los miembros del grupo.