Xavier Ledesma Pons

1)

Los tipos de prueba se pueden clasificar según dos perspectivas, la **finalidad de la prueba** o **quien ejecuta la prueba**.

Empezando por la perspectiva de la finalidad de la prueba encontramos dos grupos:

1. **Test funcionales:** se comprueban la función del software para el cual ha sido diseñado.

2. **Test no funcionales:** se comprueban otros elementos del software que forman parte de la funcionalidad principal pero que son importantes para el software.

   Podemos encontrar 3 tipos:

   a. **Test de estrés:** a partir de forzar una misma situación muchas veces, se busca sobrecargar el software y producir un fallo.

   b. **Test de carga:** se mide la capacidad máxima de concurrencia que puede tener el software sin que este falle.

   c. **Test de rendimiento:** comprueba que la respuesta del software ante una situación sea acepta o la descrita por el cliente.

Si, por el contrario, nos centramos en quien realiza la prueba nos encontramos con dos grupos:

1. **Test que realiza el cliente:** son pruebas que normalmente se realizan mediante una metodología **scrum** de desarrollo, en la cual el cliente cada cierto tiempo recibe una versión de software con las implementaciones de ese sprint y evalúa si se cumplen los criterios descritos para ese sprint.

2. **Test que realiza el desarrollador:** son pruebas que se realizan durante el desarrollo del software por parte del equipo de desarrolladores.

   a. **Test unitario:** son pruebas en las que el desarrollador verifica su propio código mediante la definición de pruebas de bajo nivel y que normalmente verifican que una parte unitaria del código se comporta como es esperado.

   b. **Test de integración:** son pruebas en las cuales el desarrollador integra diferentes partes del software y verifica que su interactuación es correcta.

c. **Test de sistemas:** son pruebas que se realizan por el equipo de desarrollo una vez se ha verificado que las pruebas unitarias y de integración son correctas. Se busca replicar un entorno similar al de producción para verificar que todo el conjunto de sistema funciona como es esperado.

2)

La prueba **double** es un termino que hace referencia al reemplazo de objetos de producción con el propósito de realizar una verificación.

Estas pruebas son muy importantes a la hora de desarrollar porque podemos forzar situaciones extrañas, como por ejemplo errores y observar como se comparta nuestro software.

Tal como se explica en el articulo, existen diversos tipos de **double**:

- Objetos **dummy** que se utilizan para rellenar estructuras y verificar el código.

- Objetos **fake**, como puede ser el caso de **InMemoryDatabase**, que nunca se usarían para producción, pero si para engañar a la prueba y hacer creer que tenemos una base de datos con valores predefinidos por el desarrollador.

- **Stubs**, que permiten forzar las repuestas que son necesarias para la prueba.

- Spies,que son stubs pero que recuerdan donde han sido llamados.

- **Mocks** que están programados previamente y que contienen cierta información que queremos usar para realizar la prueba y que se asemejan a la realidad.

3)

- **Dashboard**: se deberían realizar pruebas unitarias para verificar que se llama a **ngOnInit** y que una vez se ha realizado la llamada se llama a la función **getHeroes** del **heroService** y que se consigue rellenar la lista **heroes** con elementos. Por otro lado, se deberían realizar **UI tests** para verificar que existe una cabecera **<h2>** con el texto **"Top Heroes"** y que por cada **hero** de la lista **heroes** se crea un elemento **<a>** con el enlace correcto y se visualiza el nombre del **hero**. Aprovechando esta prueba visual, se comprobaría también que existe un elemento visual llamado **app-hero-search** (la barra de búsqueda).

- **Hero-detail**: se deberían realizar pruebas unitarias para verificar que se llama a **ngOnInit** y que una vez se ha realizado la llamada se llama a la función **getHero** del **heroService** con la id del **hero** que queremos buscar. Por otro lado, se deberían realizar **UI tests** para verificar que si existe algún **hero** se visualiza un **<h2>** con su nombre y una ** ** con su id. Aparte verificaríamos que se visualizan los **labels** correspondientes y que cuando hacemos click en un botón se ejecutan las funciones correspondientes. En este punto, en el aparto de **unit test**, verificaríamos también que, si llamamos a **goBack** y a **save**, se realizan las acciones pertinentes sobre el servicio **heroService** y sobre la navegación.

- **Hero-search:** se deberían realizar pruebas unitarias para verificar que se llama a **ngOnInit** y que una vez se ha realizado la llamada y se llama a la función **search**, se dispara el observable y se hace la petición correspondiente a **heroService**. En cuanto a **UI tests**, se verificaría que por cada **hero** del observable **heroes** se crea un elemento **<a>** con el nombre del **hero** y la URL correcta.

- **Heroes**: se deberían realizar pruebas unitarias para verificar que se las llamadas a todas las funciones realizan la acción pertinente sobre el servicio **heroService** (empezando por **ngOnInit**). En cuanto a **UI tests**, se verificaría que aparecen los botones que toca, que cuando se selecciona uno se ejecuta la acción correcta y que por cada **hero** de **heroes** se crea un elemento **<a>** con el nombre del **hero** y un botón para borrarlo.

- **Messages**: en esta clase no realizaría ninguna prueba unitaria, pero si **UI tests**. Verificaría que existe el elemento **<h2>** con el titulo **Messages**, un botón que en ser seleccionado llama al servicio **messageService** y **x <divs>** por cada mensaje con el texto del mensaje recibido.

- **Hero-service**: en esta clase solo realizaría pruebas unitarias sobre cada método del servicio. Estas pruebas las realizaría utilizando las pruebas **double**, usando **mocks** para simular los datos y **objetos fake** para simular las respuestas. Por otro lado, realizaría **pruebas de integración** de este servicio con el **backend** real, para verificar que el comportamiento entre el servicio y el **backend** es el esperado.

- **Message-service**: en esta clase solo realizaría pruebas unitarias sobre los métodos add y clear, verificando que se añaden los elementos y se borran correctamente de la lista.

Por último, si esto fuera un proyecto real que usara **scrum**, al final de cada iteración, si las **pruebas unitarias**, los **UI tests** y las **pruebas de integración** pasaran, antes de realizar pruebas de aceptación por el cliente, realizara **pruebas E2E** para plantear un escenario de producción y hacer que el cliente cuando realizará las posteriores pruebas de aceptación encontrara el mínimo numero de errores posibles.