Tema 3 - Diseño y Realización de Pruebas

Entornos de Desarrollo

Diseño de pruebas

La etapa de pruebas de software (software testing) es un proceso más dentro del desarrollo de software. Sirve para mantener un control de calidad sobre el producto que creamos, y aunque requiera tiempo al igual que programar, a la larga el software mal programado acaba saliendo demasiado caro.

Casos de prueba

Un caso de prueba (test case) es una situación, contexto o escenario bajo el que se comprueba una funcionalidad de un programa para ver si se comporta de la forma en que se espera.

Por ejemplo, un videojuego de plataformas; cuando el personaje cae al agua el jugador pierde una vida y el nivel se reinicia.

- **Objeto de la prueba**: asegurarnos que cuando el personaje cae al agua, ocurre lo esperado
- **Caso prueba**: llevas al personaje a un punto de la pantalla donde pueda caer al agua

En programación, el caso de prueba puede estar planteado por los parámetros que recibe un método o el estado de los datos del programa a la hora de ejecutar el método.

Otro ejemplo, int Math.abs(int num) devuelve un int con el valor absoluto del parámetro num. Si num > 0, devuelve num, si num < 0, devuelve num.

- **Objeto de la prueba**: asegurarnos que Math.abs() devuelve el valor absoluto
- **Caso de prueba**: si calculo el valor absoluto de -7 debe dar 7

En otras palabras, un caso de prueba es una pregunta que se le hace al programa para saber si el programa contesta (reacciona) correctamente (tal y como se quiere que reaccione).

Tipos de pruebas según su enfoque

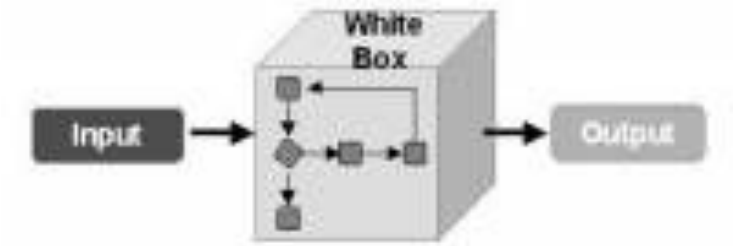
Pruebas de Caja Negra

Son las pruebas que se centran en evaluar el valor de las salidas de un sistemas a partir de unas entradas concretas, sin tener en cuenta el funcionamiento interno del sistema. Se centran en el qué hace un sistema y no en el cómo lo hace. El programador no necesita saber cómo funciona el código, sino evaluar solamente la salidas.

Este tipo de test se puede aplicar a cualquier nivel de testeo de software: pruebas unitarias, de integración, de aceptación, etc.

Pruebas de Caja Blanca

Estos tipos de pruebas se centran en analizar cada uno de los posibles caminos en el flujo de ejecución de un programa antes unos valores de entrada concretos. Es decir, si ante unos valores de entrada (parámetros) de un método, el flujo del programa ejecuta los if, o los else, o entra en un bucle, o sale de él. Este tipo de funcionamiento se puede comprobar también con el depurador.



Las pruebas de Caja Blanca también se pueden aplicar a las pruebas unitarias, pruebas de integración o de sistema.

Tipos de pruebas según su alcance

Pruebas Unitarias

Una prueba unitaria es un tipo de prueba enfocada en verificar una sección específica dentro del código de un programa. En programación orientada a objetos se realiza en cada clase y se centra en los métodos de cada clase.

Las pruebas unitarias son código (Java, C#, etc) escrito normalmente al mismo tiempo que se va desarrollando el programa (enfoque Caja Blanca). Un método creado en el programa, puede tener distintas pruebas unitarias. Por si misma no asegura el funcionamiento completo del programa, pero sin embargo no indican que los bloques de software que vamos creando funcionan independientemente de los otros.

Son los tests más importantes para el desarrollo de software y por supuesto para el desarrollo guiado por tests (TDD).

Para realizar pruebas unitarias tenemos herramientas parecidas para cada lenguaje o plataforma:

- JUnit para Java
- PHPUnit para PHP
- CPPUnit para C++
- NUnit para .NET
- CUnit para C
- PyUnit para Python

Pruebas de Integración

Las pruebas de integración es una fase en el proceso de pruebas de software, en la que se combinan los distintos módulo de software de un programa y se comprueba que trabajan de forma conjunta. Se realizan después de la fase de pruebas unitarias y se centran principalmente en probar la comunicación entre los componentes y sus comunicaciones, ya sea hardware o software.

Los tests de integración prueban que el sistema completo funciona como se espera. Por ejemplo, una interfaz gráfica y la aplicación con la que trabaja, una aplicación de acceso a datos y el módulo que trabaja un gestor de bases de datos, etc.

Pruebas de Usabilidad

Las pruebas de usabilidad consisten en seleccionar a un grupo de usuarios de una aplicación y solicitarles que lleven a cabo las tareas para las cuales fue diseñada.

Posteriormente los desarrolladores involucrados toman nota de la interacción, particularmente de los errores y dificultades con las que se encuentren los usuarios. Este tipo de pruebas no se implementan con software, sino con encuestas de uso o herramientas similares.

No es necesario que se trate de una aplicación completamente terminada, pudiendo tratarse de un prototipo o de las distintas etapas de avance de un proyecto.

JUnit

JUnit

Es un framework para Java enfocado en la realización de pruebas unitarias (unit testing). Consiste en unas librerías JAR que debemos añadir a un proyecto en Java. http://www.junit.org

Eclipse viene con JUnit integrado, por lo que no es necesario descargar ningún software adicional.

JUnit5 ofrece herramientas para poder ejecutar tests de las versiones anteriores (JUnit4 y JUnit3), por lo que JUnit5 puede ejecutar tests escritos en las versiones anteriores.

La API de JUnit5, en la que encontraremos los métodos para implementar nuestros casos de prueba:

https://junit.org/junit5/docs/current/api/

Ejemplo de clase de pruebas

```
public class ClaseTest {
    @BeforeAll
    public static void setUpBeforeClass() throws Exception {
        //Codigo que se ejecuta antes de cualquier prueba
    }

    @BeforeEach
    public void setUp() throws Exception {
        //Codigo que se ejecuta antes de cada prueba
    }

    @Test
    public void test1() {
        //Caso de prueba
    }
//Continúa…
```

```java
    @Test
    public void test2() {
        //Caso de prueba
    }

    @Test
    public void test3() {
        //Caso de prueba
    }

    @AfterEach
    public void tearDown() throws Exception {
        //Metodo que se ejecuta después de cada prueba
    }

    @AfterAll
    public static void tearDownAfterClass() throws Exception {
        //Codigo que se ejecuta después de todas las pruebas
    }
}
```

¿Qué cosas deben probarse?

Los tests unitarios se usan para evitar el miedo a que algo se rompa. Cualquier método que no tengamos claro si funciona de forma correcta o no, es un método susceptible de ser probado de forma unitaria.

Los métodos getters y setters normalmente son tan sencillos que no suelen albergar dudas, pero si hay alguna razón para que no funcionen, entonces también se deben probar.

En resumen, se debe probar todas las cosas hasta que todas nos inspiren confianza, basándonos en nuestro criterio.

Una buena aproximación es realizar una prueba afirmativa y otra negativa para cada requisito. Una prueba para evaluar que un requisito funciona adecuadamente, y otra para evaluar lo contrario.

En caso de querer comprobar todos los requisitos de un programa se deben al menos plantear dos casos de prueba para cada requisito o funcionalidad: una prueba afirmativa y una prueba negativa. Representan un caso en el que se debe obtener un resultado concreto y un caso en el que no se debe dar un resultado concreto.

Casos de prueba vs Métodos de prueba

Un **método de prueba** es simplemente un fragmento de código que realiza una comprobación sobre una funcionalidad de un programa. Para que un método sea un método de prueba debe estar precedido por la anotación @Test.

Un **caso de prueba** (test case), como hemos dicho es una situación o caso (un estado concreto del programa) bajo el que se prueba una funcionalidad del programa.

Nosotros vamos a intentar que cada método de prueba plantee únicamente un caso de prueba distinto, ya que un programa puede funcionar de formas distintas dependiendo del contexto.

Los métodos de prueba se agrupan atendiendo a su finalidad en clases de prueba. Estas clases se suelen emplazar en un package distinto al resto de las clases, por ejemplo: src/tests

Si tenemos una clase *Matematicas* con distintos métodos:

```
public class Matematicas{
    public static int suma(int num1, int num2){
        return num1 + num2;
    }
 }
```

Y creamos una clase *MatematicasTest* con métodos de prueba para probar los métodos.

```
class MatematicasTest {
    @Test
    void testSuma() {
        int actual= Metodos.suma(7,0);
        int esperado = 7;
        assertEquals(esperado, actual);
    }
 }
```

Por convenio a cada método de prueba se le suele llamar con un nombre que identifique lo que se está probando, seguido o a continuación de la palabra Test para reconocerlo rápidamente como un caso de prueba.

Ejemplo de creación de métodos de prueba con JUnit5

Características de un caso de prueba

A la hora de implementar pruebas con JUnit debemos seguir los siguientes principios. **Cada caso de prueba debe**:

1. Probar una sola cosa
2. Tener un propósito claro
3. Estar escrito de la forma más clara posible
4. Ser lo más pequeño posible
5. Ser independiente: no debe depender de otros casos de prueba
6. Poder ser repetido las veces necesarias

Anotaciones

Las siguientes anotaciones de JUnit 5 se emplean en la cabecera de cada método de test:

Anotación	Descripción
@Test	Indica que el método es un test
@DisplayName	Indica un nombre para el *test class* o el *test method*
@Tag	Define etiquetas para filtrar por tests
@BeforeEach	Se aplica a un método para indicar que se ejecute antes de cada método de prueba. (JUnit4 @Before)
@AfterEach	Se aplica a un método para indicar que se ejecuta después de cada método de prueba. (JUnit4 @After)
@BeforeAll	Se aplica a un método `static` para indicar que se ejecuta antes que todos lo métodos de prueba de la clase. (JUnit4 @BeforeClass)
@AfterAll	Se aplica a un método `static` para indicar que se ejecuta antes que todos lo métodos de prueba de la clase. (JUnit4 @AfterClass)
@Disable	Ese aplica a un método de prueba para evitar esa prueba (JUnit4 @Ignore)

Clase Assertions: Afirmaciones

La clase Assertions contiene los métodos principales de las librerías de JUnit, y los usamos para realizar las comprobaciones en los método de prueba. Son métodos estáticos que no devuelven nada, pero hacen que JUnit nos diga si una prueba falla o no.

Se usan para afirmar una condición como resultado del funcionamiento esperado de una parte del código. Si no se cumple lo que se afirma, la prueba falla y JUnit nos lo indica.

Se recomienda realizar una sola afirmación por cada caso de prueba.

- Clase Assertions JUnit5
- Clase Assert JUnit4

Metodo	Descripción
assertTrue(boolean valor)	Falla si valor no es `true`
assertFalse(boolean valor)	Falla si valor no es `false`
assertFalse(boolean valor, String mensaje)	Falla si valor no es `false` y muestra el mensaje
assertEquals(int esperado, int actual)	Falla si `esperado` es distinto de `actual`
assertEquals(int esperado, int actual, String mensaje)	Falla si `esperado` es distinto de `actual`
assertEquals(double esperado, double actual, double delta)	Falla si la diferencia entre `esperado` y `actual` es mayor a `delta`
assertNull(Object obj)	Falla si `obj` es distinto de `null`
assertNotNull(Object obj)	Falla si `obj` es `null`
assertEquals(Object esperado, Object actual)	Falla si los objetos son distintos, evaluando su método `equals()`
assertNotEquals(Object esperado, Object actual)	Falla si los objetos no son distintos, evaluando su método `equals()`
assertSame(Object esperado, Object actual)	Falla si las referencias de los objetos sin distintas

Contexto de una clase de prueba: Fixtures

Preparar el contexto de un test o grupo de tests se conoce en la jerga de JUnit como test fixture. Para ello podemos usar ciertas etiquetas antes de algunos métodos para indicar el momento y el número de veces que se ejecutan esos métodos.

Se usan para ejecutar código antes o después, de todos o cada uno, de los tests de una clase de pruebas. De este modo, evitan duplicar código, sobre todo a la hora de inicializar objetos en cada método de test.

No es necesario utilizar todas y muchas veces no necesitaremos ninguna, pero nos pueden servir para construir objetos, vaciar listas.

Una vez por cada test: @BeforeEach y @AfterEach

Actúan sobre un método para indicar que se ejecutará antes o después de ejecutar cada caso de prueba. Se usa para establecer el contexto del test.

```
public class ClaseTest{

//Una vez antes de cada test
@BeforeEach
void setUp(){
   //Me aseguro que antes de cada test solo haya un vehiculo en la lista
   listaVehiculos.clear();
   listaVehiculos.add(new Vehiculo("ASD-123");
}

//Una vez después de cada test
@AfterEach
void tearDown(){
   listaVehiculos.clear();
}
...
  //Resto de métodos de prueba
```

Una vez por cada clase de pruebas: @BeforeAll y @AfterAll

Se realiza una sola vez para toda la clase: al iniciar la clase o al terminarla. Se suele usar para abrir recursos: creación de instancias, conexiones, apertura de ficheros, etc.

La diferencia con las anotaciones anteriores, es que @BeforeAll y @AfterAll actúan sobre métodos static.

Test Fixtures en JUnit5

```
public class ClaseTest

    static FileWriter escritor;
    static GestorVehiculos gestor;

    //Se ejecuta al inicio de la clase
    @BeforeAll
    static void setUpBeforeClass() throws Exception {
        gestor = new GestorVehiculo()
    }

    //Se ejecuta al final de la clase
    @AfterAll
    static void tearDownAfterClass() throws IOException{
        escritor.close();
    }
    ...
     //Resto de métodos de prueba
```

Ejecutar varios test: Test Suite

Una Suite de pruebas (test suite) es la agrupación de diversos tests (casos de prueba o clases de prueba) en una sola unidad con la intención de poder probarlas todas al mismo tiempo.

En Junit5 existen distintas formas de hacer esto en base a las anotaciones

Test Runners

Un Test Runner es una clase encargada de lanzar los tests. No necesita tener código pero si algunas anotaciones:

```
@RunWith(JUnitPlatform.class)
public class JUnit5Runner{
  }
```

Ejecutar varias clases de tests

Si tengo varias clases de tests y quiero ejecutarlas de forma conjunta puedo seleccionar un paquete concreto. Esto se hace con la etiqueta @SelectPackages en la clase test runner que haya creado:

```
@RunWith(JUnitPlatform.class)
        @SelectPackages("tests")
        public class JUnit5Runner{
}
```

Ejecutar un paquete de tests

También es posible seleccionar las clases de pruebas, concretando cuáles quiero ejecutar. Para ello se usa la etiqueta @SelectClasses:

```
@RunWith(JUnitPlatform.class)
    @SelectClasses({MetodosTest.class, ProgramaTest.class})
    public class JUnit5Runner{
}
```

Etiquetar los tests

Con la etiqueta @Tag puedo etiquetas los tests bajo categorías. De este modo puedo ejecutar los tests de una categoría u otra.

```
@Test
@Tag("desarrollo")
void testCaseA(TestInfo testInfo) {
...
}
@Test
@Tag("desarrollo")
@Tag("produccion")
void testCaseB(TestInfo testInfo) {
...
 }
```

Ahora desde una clase runner es posible ejecutar los tests con etiquetas concretas:

```
@RunWith(JUnitPlatform.class)
@SelectPackages("tests")
@IncludeTags("produccion")
public class JUnit5Runner{
}

// o excluirlos
@RunWith(JUnitPlatform.class)
@SelectPackages("tests")
@ExcludeTags("produccion")
public class JUnit5Runner{
}

// o ejecutar tests de varias categorias
@RunWith(JUnitPlatform.class)
@SelectPackages("tests")
@IncludeTags({"produccion","desarrollo"})
public class JUnit5Runner{
 }
```

Desarrollo de Software basado en las pruebas (o dirigido por pruebas)

Desarrollo dirigido por pruebas

El desarrollo dirigido por pruebas (Test Driven Development, TDD) es una técnica de diseño e implementación de software que entra dentro de la metodología XP (eXtreme Programming). TDD se centra en tres pilares fundamentales:

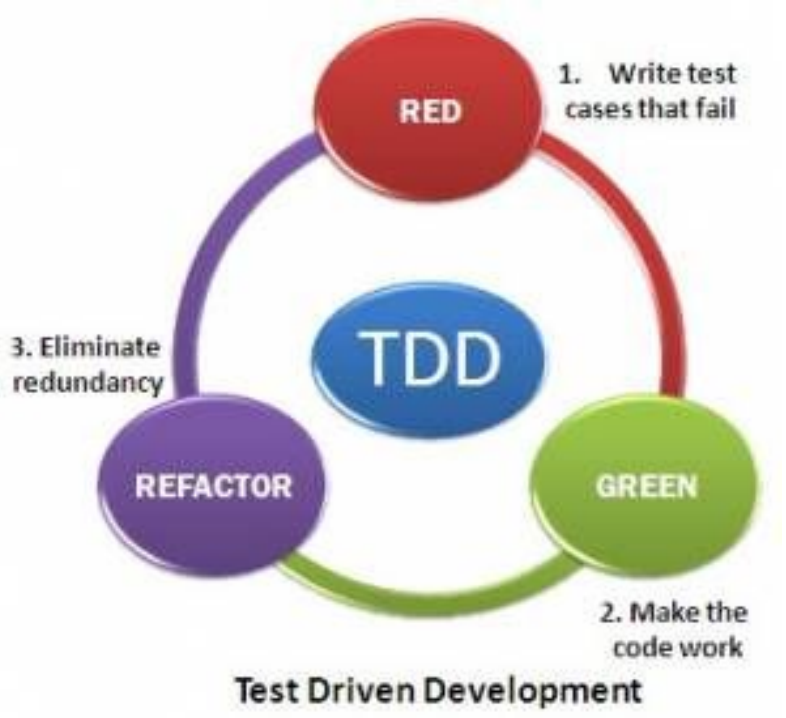
- La implementación de las funciones justas que el cliente necesita, y no más. Se debe evitar desarrollar funcionalidades que nunca serán usadas.
- Reducción al mínimo del número de defectos que llegan al software en fase de producción.
- La producción del software modular, altamente reutilizable y preparado para el cambio.

El Desarrollo guiado por pruebas plantea las pruebas antes de implementar el código, de modo que el código se debe centrar única y exclusivamente en pasar las pruebas. A partir de unos requisitos para un proyecto se diseñan los casos de prueba. Los requisitos son funcionalidades que debe realizar mi programa, por lo que se puede de antemano diseñar los tests que comprueban dicha funcionalidad y posteriormente el código que superará cada prueba.

El algoritmo TDD

La esencia de TDD es sencilla pero ponerla en práctica correctamente es cuestión de entrenamiento, como tantas otras cosas. El algoritmo TDD sólo tiene tres pasos:

1. Escribir la prueba para el requisito (el ejemplo, el test).
2. Implementar el código según dicho ejemplo.
3. Refactorizar para eliminar duplicidad y hacer mejoras.



Escribir la especificación primero

Una vez que tenemos claro cuál es el requisito o funcionalidad, lo expresamos en forma de código. ¿Cómo escribimos un test para un código que todavía no existe? ¿Acaso no es posible escribir una especificación antes de implementarla? Un test no es inicialmente un test sino un ejemplo o especificación.

Para poder escribir los tests, tenemos que pensar primero en cómo queremos que sea la API del programa, es decir, qué métodos queremos que tenga el programa y cómo funcionarán. Pero sólo una parte pequeña, un comportamiento del programa bien definido y sólo uno (método). Tenemos que hacer el esfuerzo de imaginar cómo sería el código del programa si ya estuviera implementado y cómo comprobaremos que, efectivamente, hace lo que le pedimos que haga.

No tenemos que diseñar todas las especificaciones antes de implementar cada una, sino que vamos una a una siguiendo los tres pasos del algoritmo TDD. El hecho de tener que usar una funcionalidad antes de haberla escrito le da un giro de 180 grados al código resultante. No vamos a empezar por fastidiarnos a nosotros mismos sino que nos cuidaremos de diseñar lo que nos sea más cómodo, más claro, siempre que cumpla con el requisito objetivo.

Implementar el código que hace funcionar el test

Teniendo el ejemplo escrito, codificamos lo mínimo necesario para que se cumpla, para que el test pase.

Típicamente, el mínimo código es el de menor número de caracteres porque mínimo quiere decir el que menos tiempo nos llevó escribirlo.

No importa que el código parezca feo o chapucero, eso lo vamos a enmendar en el siguiente paso y en las siguientes iteraciones.

Refactorizar

Refactorizar es modificar el diseño sin alterar su comportamiento, a ser posible, sin alterar su API pública (los métodos).

Objetos Mock
(o Simulados)

Objetos Mock

Los objetos Mock son objetos que imitan el comportamiento de un tipo real de objetos. Esto permite especificar qué valores pueden devolver sus métodos en un caso concreto. Se usan para probar el comportamiento de otras operaciones que requieran el uso de ese tipo de objetos, de la misma forma que se usan muñecos con aspecto humano (test dummies) para simular el comportamiento de un vehículo ante un choque.

Existen diversos frameworks para crear objetos simulados en Java:

- Mockito
- EasyMock
- JMock

Nosotros nos centraremos en Mockito debido a su simplicidad y facilidad de uso. Podemos descargar un fichero JAR con sus librerías desde el repositorio de Maven:

https://mvnrepository.com/artifact/org.mockito/mockito-all

Herramientas de depuración

Una vez que advertimos que nuestro código no pasa las pruebas, podemos utilizar el depurador (Modo Debug) del IDE que empleemos para revisar el estado de nuestro programa mientras lo vamos ejecutando paso a paso y corregir los posibles errores que pueda contener. Es recomendable establecer breakpoints en lugares cercanos al módulo o parte del código que queremos evaluar e investigar.