TDD con Java: caso práctico sencillo

Para entender cómo utilizar TDD con Java, vamos a ver un ejemplo concreto. Un divisor de números enteros.

Este caso es muy simple para entender la dinámica de TDD. En la práctica, es posible que el programador detecte varios casos a la vez y cree varios tests desde el principio. Aún así, siempre tenemos que asegurarnos de que todos los casos están cubiertos.

Para saber más sobre TDD: https://descubriendoagile.wordpress.com/2016/03/22/tdd/

Divisor de enteros

1. Crearemos un primer test ya que sabemos que cualquier número dividido por sí mismo es 1.

```
public class DivisorTest {
	@Test
	public void dividirUnNumeroPorSiMismo() {
		assertEquals(1, Divisor.divisorDe(4, 4));
	}
}
```

---

2. Programaremos el código mínimo que hace pasar este test. En este caso será un método que devuelve siempre 1.

```
 public class Divisor {
	public static int divisorDe(int dividendo, int divisor) {
		return 1;
	}

}
```

---

3. Añadimos el segundo caso en el cual son números distintos. El test, obviamente fallará ya que sigue devolviendo siempre 1.

```
public class DivisorTest {
	@Test
	public void dividirUnNumeroPorSiMismo() {
		assertEquals(1, Divisor.divisorDe(4, 4));
	}

	@Test
	public void dividirDosNumeros() {
			assertEquals(5, Divisor.divisorDe(20, 4));
	}

}
```

4. Modificaremos el código para que devuelva el cociente de la operación. Esto hace pasar los 2 tests.

```
public class Divisor {

	public static int divisorDe(int dividendo, int divisor) {
		return dividendo / divisor;
	}
}
```

---

5. No se puede dividir un número por 0 y nos parece importante tenerlo en cuenta. Nos creamos un test que espere la excepción aritmética. Ejecutamos los tests y vemos que pasan, es decir, se captura la excepción como queríamos.

```
public class DivisorTest {

	@Test
	public void dividirUnNumeroPorSiMismo() {
		assertEquals(1, Divisor.divisorDe(4, 4));
	}

	@Test
	public void dividirDosNumeros() {
		assertEquals(5, Divisor.divisorDe(20, 4));
	}

	@Test(expected = ArithmeticException.class)
	public void dividirPorCero() {
		Divisor.divisorDe(4, 0);
	}

}
```

Este proceso se puede usar y, de hecho, encaja perfectamente, con la definición de **requerimientos en forma de historias de usuario** que se realiza en las **metodologías ágiles.**

En una historia nosotros definiremos los **criterios de aceptación** (ACs – acceptance criteria) **que verifican la historia.** Para poder utilizar los ACs en los tests, tenemos que asegurarnos de que los ACs sean lo más atómicos posibles.

Nosotros podemos tomar esos ACs y transformarlos en tests, que fallarán inicialmente. Después, programaremos la solución que hará finalmente pasar a los tests. Por último, **refactorizaremos** el código para asegurarnos que cumple con los **principios de calidad y código limpio y evitaremos duplicidades.** Obviamente, después de refactorizar, relanzamos los tests para asegurarnos de que no hemos roto nada. Esto se puede hacer con cada uno de los ACs definidos.

TDD con Historias de Usuario

Caso práctico sobre la web de reservas de vuelos. En él detectamos varias historias de usuarios y desglosamos varios de los criterios de aceptación de una de las historias.

1. La historia que tomaremos será:

Hacer check-in online
COMO cliente con una reserva ya pagada
QUIERO poder hacer el check-in online
PARA poder acceder a mi vuelo sin problemas

2. Tomaremos un caso de aceptación para explicar TDD.

El caso será:

ESCENARIO 1: el nombre del cliente es obligatorio en el check-in

DADO un cliente en la ventana de check-in online
CUANDO deja el campo de nombre en blanco
Y pulsa el botón de confirmación
ENTONCES se muestra un mensaje de error indicando que el campo nombre es obligatorio

3. Crearíamos un test en el que probaríamos.

Test 1:

– acceder a checkin
– nombre en blanco
– pulsar el botón de confirmación
– comprobar que vemos el mensaje de alerta

En principio **el test fallaría**, ya que no tendríamos nada programado. Programaremos la menor cantidad de código posible para hacer pasar el test. Programaríamos el campo de *Nombre* y un botón de confirmación. Añadiríamos un evento al botón que hará mostrar un mensaje de alerta indicando que el nombre es obligatorio.

4. Probamos a darle al botón sin rellenar el campo Nombre y veríamos como se muestra el mensaje. El test pasaría.

Sin embargo, ¿qué ocurre cuando introducimos algo en el campo *Nombre* en esta situación? Si pulsásemos el botón veríamos también el mensaje de alerta y no es lo que queríamos en nuestros requerimientos posteriores.

Si siguiésemos con todos los criterios de aceptación, acabaríamos llegando al siguiente:

ESCENARIO 8: avanzo a la siguiente página cuando los campos obligatorios están rellenos y los confirmo

DADO un cliente en la ventana de check-in online
CUANDO relleno todos los datos obligatorios
Y pulsa el botón de confirmación
ENTONCES avanzo a la siguiente página …

En este punto tendríamos ya los tests que crean mensajes de alertas para cada campo obligatorio, pero seguiríamos con el problema de mostrar un mensaje de alerta cuando los campos se han rellenado correctamente.

El test que fallaría para este criterio sería:

Test 8:
– acceder a checkin
– relleno todos los campos obligatorios (incluido el Nombre)
– pulsar el botón de confirmación
– comprobar que pasamos a la siguiente página

En este caso añadiríamos el mínimo código correspondiente para hacer pasar el test, completando así esta parte de la funcionalidad. Después, probablemente, tendríamos que limpiar y **refactorizar** el código, para evitar redundancias, "ifs" eternos, etc.

Una vez que tenemos todos los tests, con una cobertura lo mayor posible, nos garantizaremos que si alguien toca el código y lo rompe, el fallo será rápidamente detectado por nuestros tests ya existentes.

La creación de código y tests usando TDD puede parecer tediosa cuando creamos nuevas funcionalidades desde cero, pero el valor que aporta es inmenso. Además, para cambios incrementales posteriores, el número de tests que crees será proporcional a la funcionalidad que añadas. Por ejemplo, si el desarrollo consiste en arreglar un bug, el procedimiento sería el siguiente:

1. crear un test para reproducir el bug
2. arreglar el código
3. ejecutar el test para asegurarnos que funciona y que no volverá a pasar más.