1. Proves unitàries JUnit

JUnit és un *framework* de Java per automatitzar les proves unitàries d'aplicacions desenvolupades amb aquest llenguatge de programació. Amb JUnit es poden executar classes Java de manera controlada. D'aquesta manera, es podrà avaluar si el funcionament de cada un dels mètodes de la classe es comporta com s'espera.

La forma de treballar de JUnit consisteix a validar el resultat que s'obté d'executar una classe Java amb el o els valors que s'hagin definit com a retorns esperats per a l'execució d'aquesta classe. Si coincideixen, JUnit indicarà que el mètode ha superat el test; en cas contrari, retornarà una fallada.

JUnit és utilitzat pels programadors per tal de validar les principals funcionalitats del codi, i és molt útil per efectuar **proves de regressió** quan una part del codi ha estat modificat i es vol comprovar que la resta de codi segueix funcionant correctament.

1.1. Manual de referència

A continuació, es mostren algunes indicacions i referències per a l'ús de JUnit, com ara les anotacions i les declaracions.

1.1.1. Anotacions

1. `@Test public void metode`

   Aquest mètode serà executat com a test, per exemple:

   ```
   @Test
   public void nomMetode() {
       // Codificació del test
       System.out.println("Execució del test");
   }
   ```

2. `@Before public void mètode`

   Aquesta anotació s'executa prèviament a iniciar cada mètode `@Test`. Es pot observar el codi a continuació:

   ```
   @Before
   public void setUp() {
       // Codificació.
       // Exemple: inicialització de variables, crida al
       // constructor, lectura d'un paràmetre d'usuari,...
       System.out.println("Execució a l'inici de cada test");
   }
   ```

3. `@After public void mètode`

   `@After` és executat en finalitzar cada `@Test`.

```
@After
public void after() {
    // Codificació.
    // Exemple: alliberació de memòria, reiniciar paràmetres per
    // defecte,...
    System.out.println("Execució al finalitzar cada test");
}
```

4. `@BeforeClass public void mètode`

   S'executa una única vegada, abans que es comencin a executar els tests. Aquesta funció pot ser útil quan es vol efectuar la connexió a una base de dades.

5. `@AfterClass public void mètode`

   S'executa una única vegada, un cop s'han finalitzat els tests. Aquesta funció pot ser útil quan es vol alliberar la connexió a la base de dades.

6. `@Ignore`

   Ignora el test. Se sol utilitzar quan s'està modificant el codi i no es vol que s'executin determinades proves, ja que aquestes no han estat adaptades a les noves modificacions.

```
@Ignore
@Test
public void nomMetode() {
    // Codificació del test
    System.out.println("Aquest missatge no s'hauria de
    visualitzar");
}
```

7. `@Test (expected = Exception.class)`

   Provoca un error si el test no retorna l'excepció que s'especificarà.

```
@Test(expected = ArithmeticException.class)
public void dividirPorCero() {
    // Codificació del test
    System.out.println("Divisió del test");
}
```

8. `@Test (timeout=100)`

   El test falla si l'execució del mètode dura més de 100 ms.

1.1.2. Declaracions

A continuació, s'indiquen les principals declaracions de JUnit:

- `fail (string)` Deixa que el mètode falli. Podria ser utilitzat per verificar que una certa part del codi no s'assoleix.
- `assertTrue (true)` / `assertTrue(false)`. Sempre serà vertader o fals. Es pot utilitzar per predefinir un resultat de la prova, si la prova no s'ha implementat encara.
- `assertTrue([missatge], boolean condició)`. Comprova que la condició booleana és verdadera.
- `assertEquals([String missatge], valor_esperat, valor_actual)`. Valida que dos valors són iguals.
- `assertsEquals([String message], valor_esperat, valor_actual, tolerància)`. Verifica que els valors flotant o doble coincideixen. La tolerància és el nombre de decimals que han de coincidir.
- `assertNull([missatge], objecte)`. Verifica que l'objecte és nul.
- `assertNotNull([missatge], objecte)`. Verifica que l'objecte no és nul.
- `assertSame([String], valor_esperat, valor_actual)`. Verifica que les dues variables referencien el mateix objecte.
- `assertNotSame([String], valor_esperat,valor_actual)`. Verifica que les dues variables no referencien el mateix objecte.

1.2. Exemple

Suposem que hem definit la classe `Enter` com es mostra a continuació:

```
public class Enter {
     int valor;

     public void setValor(int v) {
         valor = v;
     }

     public int getValor() {
         return valor;
     }

     public int suma(Enter e) {
         valor += e.getValor();
         return valor;
     }
}
```

Volem testejar que els mètodes són correctes, per fer-ho utilitzarem JUnit.

- **Crear el fitxer de verificació**
  - El primer que caldrà fer serà crear el fitxer de verificació, per fer-ho des de l'Eclipse anirem a `File > New > JUnit Test Case`.
  - Posarem un nom a la classe i indicarem quina és la classe que volem testejar (*Class Under Test*), també seleccionarem que creï el mètode `setUp()` i clicarem a `Next`. Aquest mètode és el que s'executarà abans de realitzar qualsevol test i l'aprofitarem per crear i inicialitzar el l'objecte `Enter`.
  - A continuació es podrà seleccionar quins mètodes es voldran verificar.
  - Ja tenim la nova classe creada, ara només caldrà implementar els mètodes.

```
public class EnterTest {

    private Enter enter;

    @Before
    public void setUp() throws Exception {
        enter = new Enter();
        enter.setValor(5);
    }
    @Test
    public void testSetValor() {
        enter.setValor(8);
        assertEquals(enter.getValor(), 8);
    }
    @Test
    public void testGetValor() {
        assertEquals(enter.getValor(), 5);
    }
    @Test
    public void testSuma() {
        Enter e = new Enter();
        e.setValor(4);
        enter.suma(e);
        assertEquals(enter.getValor(), 9);
    }
}
```

- **Verificar els mètodes**
  - Per verificar els mètodes només caldrà anar a `Run > Run As > JUnit Test` i obtindrem un resum de si els mètodes retornen els resultats esperats.