Clase 8 - Test JUnit

Status	Librerías
Assign	

Para testear el código usamos Junit. Este nos brinda distintas etiqueta para generar los tests

- `@test`: etiqueta necesaria antes de cada método de test para que Junit lo reconozca como un test y lo ejecute.
- `@ParameterizedTest`: permite correr el test con múltiples argumentos. Puede tomar los parámetros de diferentes fuentas, como un método, unos valores o un archivo.
- `@Disable`: Deshabilita un test para que no se ejecute, este test será ignorado.
- `@BeforeEach`: ejecuta el método antes de la ejecución de cada test.
- `@BeforeAll`: ejecuta el método antes de la ejecución de todos los test de la clase.
- `@AfterEach`: ejecuta el método luego de la ejecución de cada test
- `@AfterAll`: ejecuta el método luego de la ejecución de todos los test de la clase.
- `@Tag`: permite lanzar conjuntos de test en función de las etiquetas que especifiquemos. Anotaciones de ciclo de vida. Sirven para establecer los fixtures. Pueden ser de método o de clase.

Dentro de los métodos podemos usar assertions para corroborar los valores de salida de los métodos.

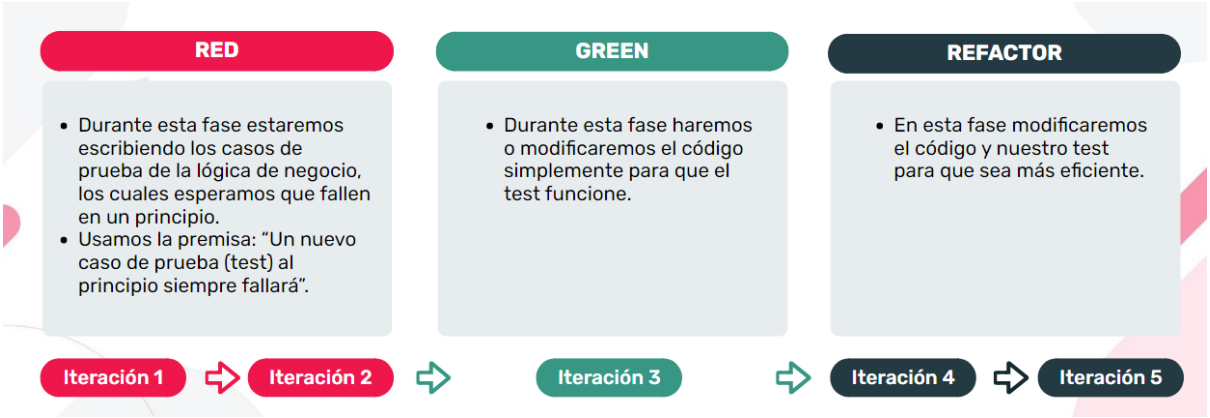
```
assertEquals(4, 4);
assertNotEquals(3, 4);
assertTrue(true);
assertFalse(false);
assertNull(null);
assertNotNull("Hello");
assertNotSame(originalObject, otherObject);
```

```
assertTrue(persona1.esMayor());
assertFalse(personaMenor.esMayor());
```

Tipos de pruebas

TDD:

Por último, veremos de manera muy general que es TDD (Test Driven Development). Cabe aclarar que esto lo verán en profundidad en Testing I.TDD significa que usamos las pruebas (tests) para orientar o dirigir la forma en la que escribimos nuestro código. Normalmente, el flujo de trabajo se describe como Red, Green, Refactor.



Unitarios

Los test o pruebas unitarias tiene por objetivo tomar una pequeña parte de software, **aislandola** del resto del código, para determinar si se comporta/funciona tal como esperamos.

Cada unidad se prueba por separado antes de ser integrada en los componentes para probar las interfaces entre las unidades.

Cabe aclarar que, **cualquier dependencia del módulo bajo prueba debe sustituirse por un mock o un stub, para acotar la prueba específicamente a esa unidad de código**.

Para llevar a cabo un correcto test unitario se debe seguir un proceso conocido como 3A:

- `Arrange`: En este paso se definen los **requisitos** que debe cumplir el código.
- `Act:` Aquí **se ejecuta el test** que dará lugar a los resultados que debemos analizar.
- `Assert`: Se comprueban si los resultados obtenidos **son los esperados**. Si es así, se valida y se continúa. Caso contrario, se corrige el error hasta que desaparezca.

Ventajas

- Facilitar los cambios en el código: Al detectar el error rápidamente es más fácil cambiarlo y volver a probar.
- Proveen documentación: Ayudan a comprender qué hace el código y cuál fue la intención al desarrollarlo.
- Encontrar bugs: Probando componentes individuales antes de la integración. Esto genera que no impacten en otra parte del código.
- Mejoran el diseño y la calidad del código: Invitan al desarrollador a pensar en el diseño antes de escribirlo (Test Driven Development - TDD).

Principio FIRST

Son las cinco características que deben tener para ser considerados tests con calidad.

- **F**: Fast ⇒ Es posible tener miles de tests en nuestro proyecto y deben ser r**ápidos de correr**.
- **I**: Isolated/Independent ⇒ Un método de test debe cumplir con los «3A» (arrange, act, assert). Además, no debe ser necesario que sean corridos en un determinado orden para funcionar, es decir, deben ser **independientes unos de los otros.**
- **R**: Repetable ⇒ Resultados determinísticos. No deben depender de datos del ambiente mientras están corriendo —por ejemplo, la hora del sistema—.

- **S**: Self-validating ⇒ **No debe ser requerida una inspección manual para validar los resultados.**
- **T**: Thorough ⇒ Deben **cubrir cada escenario** de un caso de uso y no solo buscar una cobertura del 100%. Probar mutaciones, edge cases, excepciones, errores, entre otros.

De Integración

Las unidades individuales se **integran** para formar componentes más grandes, por ejemplo, **dos unidades que ya fueron probadas se combinan en un componente integrado y se prueba la interfaz entre ellas**. Esto nos permite cubrir un área mayor de código, del que a veces no tenemos control.

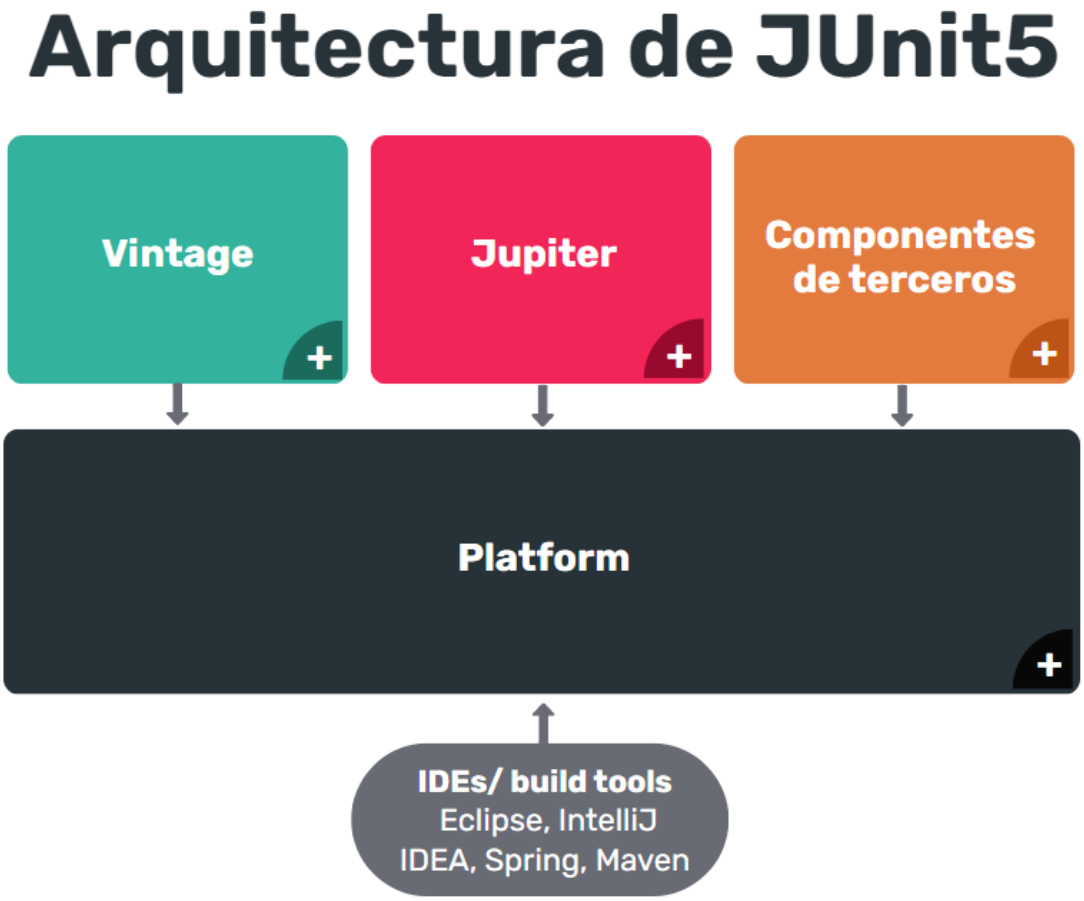
Por lo tanto, podemos concluir que este tipo de test tiene por objetivo **validar la interacción** entre los módulos de software.

JUnit

JUnit es el framework open source de testing para Java más utilizado. Nos permite escribir y ejecutar tests automatizados.

Es soportado por todas las IDEs (Eclipse, IntelliJ IDEA), build tools (Maven, Gradle) y por frameworks como Spring. Conozcamos su arquitectura.

Arquitectura JUnit54



`Vintage`: Contiene el motor de JUnit 3 y 4 para correr tests escritos en estas versiones (Old test).

`Jupiter`: Es el componente que implementa el nuevo modelo de programación y extensión: API para escribir tests y motor para ejecutarlos (New tests).

`Componentes de terceros`: La idea es que otros frameworks puedan ejecutar sus propios casos de prueba realizando una extensión de la plataforma.

`Platform`: Es un componente que actúa de ejecutor genérico de los tests. La platform-launcher es usada por las IDEs y los build tools.

Configurar la librería

Para configurar la librería de JUnit en nuestro entorno de desarrollo (IntelliJ IDEA) debemos
seguir los siguientes pasos:

1. **Nos paramos sobre la clase que queremos testear, hacemos alt + Enter y hacemos**
   click en Create Test.

2. Luego el IDE nos guiará automáticamente para **agregar la librería de JUnit de la**
   siguiente manera.
3. Hacemos click sobre el **botón OK y se descargará la librería en la ruta recomendada**.
4. **Como último paso debemos agregarla a lo que se denomina classpath. La manera más fácil es agregar el paquete, como vemos en la imagen, y en la recomendación del IDE aparece cómo agregarla**.

Testeo Parametrizado y Test Suite

Para construir test parametrizados, JUnit utiliza un custom runner que es Parametrized. Nos permite definir los parámetros de varias ejecuciones de un solo test.

```
 import org.junit.Assert;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;

import java.util.Arrays;

@RunWith(Parameterized.class)
public class MultiplicarTest {
   @Parameterized.Parameters
   public static Iterable data(){
       return Arrays.asList(new Object[][]{
               {4,2,2},{6,3,2},{5,5,1},{10,5,2}
       });
   }
```

```java
 private int multiplierOne;
   private int expected;
   private int multiplierTwo;

   public MultiplicarTest(int expected, int multiplierOne, int multiplierTwo) {
       this.multiplierOne = multiplierOne;
       this.expected = expected;
       this.multiplierTwo = multiplierTwo;
   }

   @Test
   public void debeMultiplicarElResultado(){
       Assert.assertEquals(expected,multiplierOne*multiplierTwo);
   }
}
```

```
@RunWith(Parameterized.class)
```

Con esa línea indicamos que vamos a utilizar el runner de Parameterized, que se encargará de ejecutar el test las veces necesarias dependiendo del número de parámetros configurado.

La anotación Parameters indica cuál es el método que nos va a devolver el conjunto de parámetros a utilizar por el runner.

Lo que necesitamos es un constructor que permita ser inicializado con los objetos que tenemos en cada elemento de la colección. Finalmente, se ejecutará el test utilizando los datos que hemos recogido en el constructor.

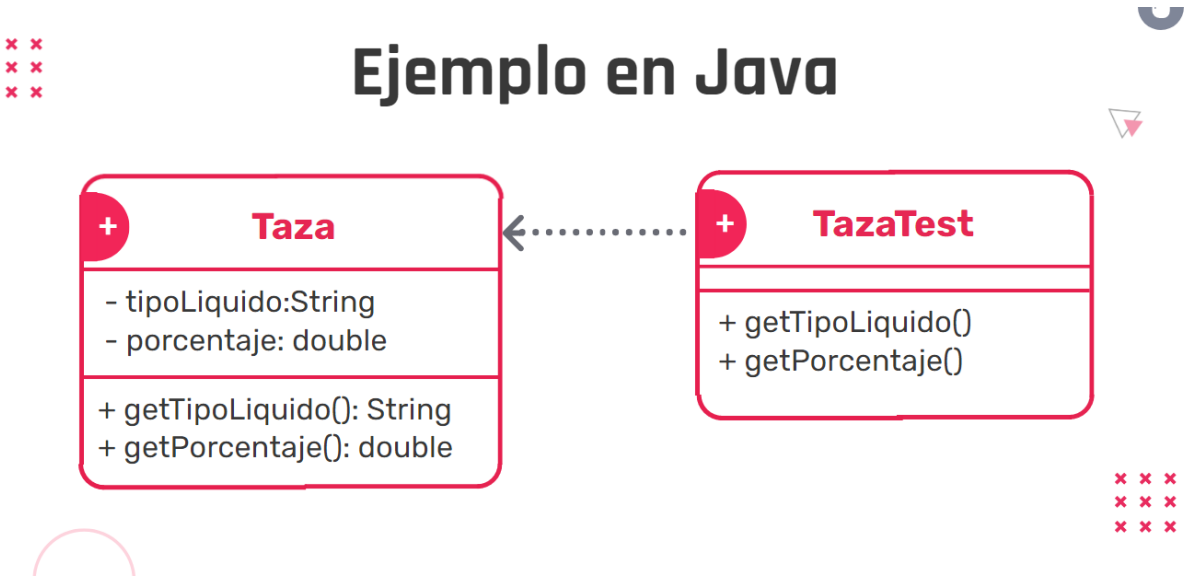
Test Suite

JUnit test suite nos permite agrupar y ejecutar los test en grupo. Las suites de prueba se pueden crear y ejecutar con:

`RunWith`

`SuiteClasses`

Ejemplo




```
public class Taza {
   private String tipoLiquido;
   private double porcentaje;
   public Taza(String tipoLiquido, double porcentaje) {
       this.tipoLiquido = tipoLiquido;
       this.porcentaje = porcentaje;
   }
   public String getTipoLiquido() {
       return tipoLiquido;
   }
   public void setTipoLiquido(String tipoLiquido)
       this.tipoLiquido = tipoLiquido;
   }
   public double getPorcentaje() {
       return porcentaje;
   }
   public void setPorcentaje(double porcentaje) {
       this.porcentaje = porcentaje;
   }
 }
```

```
import static org.junit.jupiter.api.Assertions.*;
 class TazaTest {
    @Test
    void getTipoLiquido() {
       Taza taza = new Taza("Jugo de Naranja", 70.5);
       assertEquals("Jugo de Naranja", taza.getTipoLiquido());
    }
    @Test
    void getPorcentaje() {
       Taza taza = new Taza("Jugo de Naranja", 70.5);
```

```
        assertEquals(70.5, taza.getPorcentaje());
    }
}
```

Notas de la clase

Date: @August 17, 2021

Topic: Testing

Notes

- `@BeforeEach` se ejecuta este método antes de ejecutarse el testeo, aqui conviene instanciar todos los objetos que voy a necesitar para correr mis tests.
- `assertEquals(expected, actual)` : le paso el valor real y el esperado y dará true si coinciden y false si no.

  Ej: `assertEquals(personaService.getPersonas().size, 2);`
- `Period.between` te saca la diferencia entre fecha y fecha
- `.getYears` retorna los años de cada fecha

📌 **RESUMEN: estos test se usan para corroborar que los resultados (las salidas) del código sean las esperadas.**

Código en clase

Consigna

Crear una aplicación con una persona que se va a guardar en una colección, solamente si la persona es mayor de edad (18 años) y el nombre contiene 5 letras o más. Por ejemplo, ‘’Javier’’ debería ser guardado.

Crear un test que instancie 10 personas:

- 5 de ellas deben tener un nombre menor a 5 letras

- Las otras 5 personas deben tener un nombre de 5 letras o más.
- Solo 2 de las personas que tengan nombres mayores a 4 letras deben ser mayores de edad.
- Hacer un assertion corroborando que solo 2 personas están en la lista.

Código

```
//CLASE TEST
class ImplementacionPersonaTest{
  ImplementacionPersona personaService;

  Persona persona;
  Persona persona1;
  Persona persona2;
  Persona persona3;
  Persona persona4;
  Persona persona5;
  Persona persona6;
  Persona persona7;
  Persona persona8;
  Persona persona9;

  @BeforeEach
  void doBefore(){
    //aca voy a instanciar a las personas en las variables definidas antes con las pro
piedades indicadas en la consigna
    personaService = new ImplementacionPersona();
    persona = new Persona("Mar", "Opra");
    persona.setEdad(LocalDate.of(1980, 4, 12));
    //crear 9 mas...

  }

  @Test
  void getEsMayorDe18(){
    personaService.guardarPersona(persona);
    //repetir para las otras 9
    assertEquals(personaService.getPersonas().size, 2);
  }
}
```

```
public class ImplementacionPersona{
  private List<Persona> listaPersonas = new ArrayList<>();

  public void guardarPersona(Persona persona){
    if(persona.esMayorDeEdad() && person.getNombre().length() > 4){
      listaPersonas.add(persona);
    }
  }
```

```
  //Getter de la listaPersonas
}
```

```
public class Persona{
  private String nombre;
  private String apellido;
  LocalDate date;

  public Persona(String nombre, String apellido){
    this.nombre = nombre;
    this.apellido= apellido;
  }

  public String getFullName(){
    return nombre + " " + apellido;
  }

  public boolean esMayorDeEdad(){
    return Period.between(this.date, LocalDate.now()).getYears >= 18
  }


}
```

Local Date

The of(int, int, int)

method of

LocalDate

class in Java is used to create an instance of

LocalDate

from the input year, month and day of the month. In this method, all the three parameters are passed in the form of integer.

Syntax:

```
public static LocalDate of(int year,
                           int month,
                           int dayOfMonth)
```

Parameters: This method accepts three parameters:

- **year** – It is of integer type and represents the year. It varies from MIN_YEAR to MAX_YEAR.
- **month** – It is of integer type and represents the month of the year. It varies from 1(JANUARY) to 12(DECEMBER).
- **dayOfMonth** – It is of integer type and represents the day of the month. It varies from 1 to 31.

Return Value: This method returns the **localdate**.