**Test unitario**

Es un proceso de desarrollo de software en el que las partes testeables mas pequeñas de una aplicación, llamada unidades, se examinan de forma individual e independiente para comprobar su correcto funcionamiento. O sea que hagan lo que tengan que hacer

Estas unidades normalmente son funciones o métodos de clase.

Se realiza en la parte de desarrollo y si se realiza correctamente nos puede ayudar a:

* Corregir errores en etapas tempranas de desarrollo y ahorrar costos

Si nuestro código no es testeado en etapas tempranas, es posible que arrastre el error a futuras etapas, lo que hará que al detectarse el error sea necesario tocar mas partes del sistema lo que lo haría mas costoso.

* Entender el codebase
* Realizar cambios sin miedo. Los desarrolladores saben que están respaldados si rompen algo porque los tests van a fallar.
* Reutilización de código

Tenemos un input que va a ser procesado y devuelve un output, un valor esperado. En nuestras pruebas unitarias vamos a tener un valor esperado, que no siempre puede dar ese resultado. Conocemos el input por lo que podemos determinar el output, si ese output no coincide con lo que esperamos entonces hay un error. Si coincide el código funciona bien. Ejemplo en una suma de intput 2 y 3.

**Test doubles o dobles de test**

Son objetos que reemplazan a otros objetos para testea su funcionalidad y lograr que los test sean independientes

Existen 5 tipos de test double:

Stubs: Son objeto que simulan lo que retorna la función que vamos a probar

**Mocks:** Mock testing, también conocido como prueba simulada o prueba de simulación, es una técnica de pruebas de software que se utiliza para simular el comportamiento de un componente o sistema en un entorno controlado. En lugar de probar el componente o sistema real, se crean objetos de prueba simulados que emulan el comportamiento del componente o sistema real. Estos objetos simulados, también conocidos como mocks, se utilizan para probar el código del componente o sistema bajo diferentes condiciones y escenarios. Dummys: son objetos que no hacen nada, pero permiten satisfacer las necesidades de una función

Fakes: simulan distintos objetos mediante una implementación distinta como por ejemplo un sistema de autenticación.

Spies: son stubs pero mantienen información sobre el estado del mismo

**TDD**

Es una practica de programación que consiste en escribir los tests primero y luego escribir el código que se quiere probar. De esta manera se puede definir claramente los objetivos que se buscan al realizar el código y además aseguramos que no exista una función que no esté debidamente acompañada de un test.

**Junit 5**

Es framework para escribir pruebas unitarias de nuestro código y ejecutarlas en la Java Virtual Machine. Utiliza programación funcional y lambda e incluye varios estilos diferentes de pruebas, configuraciones, anotaciones, ciclo de vida, etc.

Incluye pruebas unitarias repetibles, que persisten en el tiempo. Permite realizar varias pruebas con distintos parámetros.

Arquitectura de Junit 5

Junit5 tiene un arquitectura robuta con diferentes componentes:

JUnit Plataform: es el core, es la librería principal, es el framework de ejecución, nunca como desarrolladores vamos a interactuar con Junit Plataform. Es el motor, nos permite ejecutar pruebas con Junit Jupiter y Junit Vintage o incluso de otro proveedor que no sea ninguno de estos.

Junit Jupiter es la api por la cual nosotros interactuamos y escribimos nuestras pruebas.

Interactuamos de forma directa con esta api,

Junit Vintage: es un componente que nos permite integrar la versión 4 con la 5. Es decir si tenemos un proyecto que ya usaba Junit4, con Junit Vintage lo podemos integrar a 5 y siguen funcionales.

**Anotaciones**

**@Test**

Se utiliza para marcar un método como un método de prueba, cuando se ejecuta un conjunto de pruebas Junit, el marco de pruebas ejecuta automáticamente todos los métodos de prueba que están marcados con la anotación @Test.

**@DisplayName**

La anotación "@DisplayName" es una anotación de JUnit que se utiliza para proporcionar un nombre legible para una clase de prueba, un método de prueba o un grupo de pruebas.

**@Nested**

La anotación "@Nested" es una característica introducida en JUnit 5 que permite anidar clases de prueba dentro de otras clases de prueba. Esta anotación se utiliza para agrupar pruebas relacionadas en una sola clase de prueba y mejorar la organización y legibilidad del código.

**@Tag**

**Assertions.*assertEquals*(esperado,actual);** compara dos string, el primero es el esperado y el segundo es el real

***assertTrue*(actual.equals("Adrian"));** compara si actual es igual a lo pasado entre comillas

**assertFalse(cuenta.getSaldo)<=0)**verifica si el saldo no es menor a cero, o sea se espera false como resultado. El saldo es menor o igual a cero? False. Si da true falla.

**assertNotEquals(cuenta2, cuenta);** verifica si los dos objetos no son iguales. Si son iguales da error.

**assertEquals(cuenta2, cuenta);** verifica si lo dos objetos son iguales. Si no son iguales da error, los objetos son distintos siempre, son distintas referencias por mas que tengan los mismos datos. En la clase de los objetos podemos definir el método equals que será usado por el método en test para determinar si son iguales los objetos.

**assertNotNull(cuenta.getSaldo());** valida que no sea nulo

**Exception exception = assertThrows(DineroInsuficienteException.class, ()->{**

**cuenta.debito(new BigDecimal(1200));** valida si la excepción lanzada en el método cuenta.debito es igual a la excepción DinseroInsuficienteException.

**String actual = exception.getMessage();  
String esperado = "Dinero insuficiente";  
assertEquals(esperado,actual);** valida el mensaje que vino en la excepción

**assertAll(()->{assertEquals("Santander", cuentaOrigen.getBanco().getNombre());},  
 ()->{assertEquals("Santanderrrr", cuentaOrigen.getBanco().getNombre());});**

permite que si falla mas de un assert dentro del método, se muestren los errores de todos los que fallaron y no solamente de uno, permite que se ejecuten todos.

**assertEquals(1100, cuenta.getSaldo().intValue(),"error al calcular");** le manda un mensaje en caso de error

**Ciclo de vida**

Es un proceso en el cual se crea una instancia, en nuestro caso Cuenta test, o sea la referencia. Se administra de alguna forma y luego se termina una instancia, o sea al final del proceso se destruye.

Quien inicializa la clase CuentaTest es el motor de JUnit, JUnit Plataform.

**Eventos:** tenemos la oportunidad de ejecutar determinado código en determinado momento de la ejecución, puede ser antes o después de algún método, un HOOK o algo así.

Instancia de la clase prueba: va a tener métodos que son parte del ciclo de vida, en cada método se creará una instancia nueva de la clase, el orden en el que se van a ejecutar no se puede saber, es un orden aleatorio designado por JUnit.

Se puede ordenar pero no es una buena practica hacerlo para crear dependencia entre los tests ni guardar un estado mientras ocurre la ejecución. Los métodos deben ser independientes porque si no puede cambiar el comportamiento de nuestros tests. No es recomendable dar un estado fuera de los métodos del tests, teniendo en cuenta esto no se deberían ejecutar en un orden establecido.

BeforeAll y AfterAll se ejecutan una sola vez para todos los métodos, se ejecutan incluso antes de ser creada la instancia por debajo CuentaTest, luego se invoca el beforeEache, luego se ejcuta el método test y luego el afterEache.

**Estos métodos se ejecutan antes de creada la instancia porque son métodos estáticos**, por lo que no pertenecen a una instancia en si, si no a la clase.

Anotaciones ciclo de vida

Tendremos diferentes hook

Aquí tendremos distintas anotaciones, la primera es @BeforeAll y la ultima es @AfterAll por lo que hacen es envolver la ejecución de nuestros métodos.

Cada vez que se ejecute un método podemos implementar algo, por ejemplo antes con @BeforeEach o después con @AfterEach.

@TestInstance(TestInstance.Lifecycle.PER\_CLASS) creará una sola instancia para toda la ejecución. Lo cual no es una buena practica

**Condicionales**: son condiciones que se agregan a los métodos para que se ejecuten si las mismas se cumplen o que se ejecuten en determinado contexto dependiendo de sus características.

**@EnabledOnOs(OS.WINDOWS)** se ejecutará solo en el sistema operativo establecido. Si no se cumple con que sea Windows, entonces este método se deshabilita.

**@DisabledOnOs(OS.WINDOWS)** si el sistema operativo es de windows entonces lo deshabilitamos.

**@EnabledOnJre(JRE.JAVA\_8)**

**@EnabledIfSystemProperty(named= “user.name”, matches=”adrir”)** propiedades del sistema

**@DisabledIfSystemProperty(named=”os.arch”,matches=”\*.32.\*”)**

**boolean esDev= "DEV".equals(System.getProperty("ENV"));  
assumeTrue(esDev);**  si es false, entonces automáticamente queda deshabilitada la prueba y no da error, si es true entonces ejecuta lo que hay debajo

assumingThat(esDev, () ->{ // si es true,se ejecuta lo que hay dentro, si no no  
String esperado = "Adrian";  
String actual = cuenta.getPersona();  
Assertions.assertEquals(esperado, actual);  
assertTrue(actual.equals("Adrian"));});  
  
//pueden haber metodos por fuera del assumingThat que se van a ejecutar igual

@Nested nos permite anidar clases dentro de la clase de test, dentro de estas clases podemos agregar BeforeEach o AfterEach, pero no BeforeAll ni AfterAll

**@RepeatedTest(5)** nos permite repetir la cantidad de veces que queramos el método. Sirve cuando pasamos distintos tipos de parámetros.

**@ParametrizedTest  
@ValueSource(strings={"100","200","300","400"})** le pasamos parámetros a la prueba

**@CsvFileSources(resource = “/data.csv”)** le indica que los parámetros los debe levantar de el archivo data.csv en el package resources

**@MethodSource(“montoList”)** llama a un método en la misma clase de nombre montoList que devuelve una lista de string con los parámetros

**@Tag(“pruebasDeAssum”)** sirven para ejecutar pruebas de forma selectiva, para que solo se ejecuten las pruebas con determinada etiqueta

**@BeforeEach**void initMetodoTest(**TestInfo testInfo, TestReporter testReporter** System.out.println("Iniciando el metodo");  
 this.cuenta = new Cuenta("Adrian", new BigDecimal("1002.545"));  
 **System.out.println("ejecutando " + testInfo.getDisplayName() + " " + testInfo.getTestMethod());**  
} en el beforeEach podemos determinar la info del test de cada uno

Timeout, se puede controlar el tiempo y si la prueba demora mas de determinado tiempo la podemos hacer fallar, o sea que controlamos el tiempo de ejecución de la prueba, puede ser usada para pruebas muy pesadas.

**@TimeOut(5)** 5 segundos de espera, si demora mas entonces la falla

**@TimeOut(value = 500, unit = TimeUnit.MILLISECONDS) le determinamos el tipo de tiempo**

**assertTimeout(Duration.ofSeconds(5), ()->{TimeUnit.MILLISECONDS.sleep(5500);});** también es una forma de controlar el tiempo de ejecución y que falle si pasa

Si falla al menos un assertion dentro de un método, el método va a fallar, es todo o nada.

Tambien es posible esperar un tipo de error o un tipo de excepción que devuelve un método, son solamente un valor

**Mockito**

Es un framework de pruebas unitarias que nos permite crear objetos simulados(mock) en un entorno controlado y determinado. Le podemos dar un comportamiento deseado a un mock.

Se encarga de dar un comportamiento deseado a un objeto real pero simulando este objeto, no estaremos tabajando con una instancia real de repositorio ni restemplate ni servicio.

De esta forma podemos tener varios contextos de pruebas con varios datos para un mismo objeto simulado.

Cuando trabajamos con mockito debemos preparar el ambiente de prueba, la clase tendrá dependencias de clases exteriores.

Por ejemplo si estamos trabajando con lo que devuelve una lista y queremos probar lo que sucede cuando esa lista es vacía, no sería una buena practica, además de que sería muy trabajoso estar modificando la lista para que nos envié una lista vacía, o incluso puede suceder que ni siquiera tengamos a acceso al control de lo que nos devuelve esa lista para poder ponerle que sea vacía. En este caso es cuando necesitamos utilizar un simulador mock.

**IExamenRepositorio repositorio = Mockito.mock(IExamenRepositorio.class);** esto va a crear una instancia al vuelo, una instancia simulada de IExamenRepositorio

**when(repositorio.findAll()).thenReturn(datos**); lo que hace esto es determinar **que cuando** se llame al método findAll se retorne la lista de datos definida como array dentro del método de prueba y no en el objeto verdadero porque repositorio ya no es el objeto verdadero si no que es el mock definido en el método.

Podemos tener una clase denominada Datos para el manejo de los datos que queramos pasar al objeto mock y probar con ellos, serán datos de ejemplos justamente para las pruebas, sirven para no tener que declarar una y otra vez los datos que queremos probar dentro del método.

Asi como tenemos **when** también tenemos **verify** que sirve para validar si efectivamente se llamó a algún método y si valida que si fue llamado, entonces la prueba es correcta, si no da error.

**verify(repositorio).findAll();** verificamos que del objeto mock repositorio se llamó al findAll, si no se llama va a fallar. Sirve para verificar si hubo interaccion con algún método u objeto mock

**@InjectMocks** realiza una inyección de dependencia

MockitoAnnotations.openMocks(this); /dentro del beforeEache habilita las anotaciones de Mockito