Entornos de Desarrollo – UD3

DAM1

|  |
| --- |
|  |
| Entornos de Desarrollo |
| UD3 – Diseño y Realización de Pruebas  Actividad 1  Pruebas unitarias Java con JUnit. |

Contenidos

**Contenidos 1 Objetivos. 1 Material. 1 Actividad. 2**

Requerimientos. 2 Creamos la clase con atributos y métodos. 2 Diseño de pruebas de Account. 3 Creamos la clase de pruebas para Account. 4 Validar excepciones. 5 Test controlados por datos. 7 Tipos de test. 7 Pruebas de rendimiento. 8

Objetivos.

Crear y ejecutar una batería de pruebas unitarias de un proyecto Java con JUnit. Las pruebas comprobarán los resultados de los métodos, excepciones generadas y por último crearemos una prueba controlada por datos.

Material.

**● IntelliJ Idea**

● **JUnit 5**. Al utilizar IntelliJ deberemos instalar el plugin JUnit y en cada proyecto deberemos importar las librerías necesarias en el IDE: Project Structure → Libraries y buscamos:

org.junit.jupiter:junit-jupiter.engine:5.9.3 User Guide JUnit 5 ● Para realizar las pruebas sin un IDE necesitaremos configurar Maven o Gradle (el build tool que estemos utilizando) para poder usar JUnit. Por ejemplo: <dependency>

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter-params</artifactId>

<version>5.x.x</version>

<scope>test</scope>

</dependency>

1

Entornos de Desarrollo – UD3 DAM1

*Cambiaremos la versión por la que estemos utilizando.*

Actividad.

Vamos a ver un ejemplo de cómo crear pruebas unitarias para nuestros proyectos; crearemos una clase con una funcionalidad sencilla y a continuación vamos a crear un proyecto de pruebas para dicha clase.

**Requerimientos.**

Proyecto bancario para gestión de cuentas personales.

La cuenta del banco se identificará mediante el nombre del cliente No se aceptan cuentas anónimas, el nombre no podrá estar vacío. Las operaciones permitidas serán:

- Crédito, añade un importe al balance de la cuenta.

- Débito, sustrae un importe al balance de la cuenta, si hay saldo suficiente.

**Creamos la clase con atributos y métodos.**

1. Creamos un proyecto de tipo Java llamado *Bank*.

2. Creamos la clase *Account*.

3. Creamos dos propiedades una para el nombre del titular, *CustomerName*, y otra para saldo de la cuenta, *Balance*.

public class Account {

public String CustomerName;

public double Balance;

}

4. Creamos los getter y setter para las dos propiedades usando el generador de código (Alt+Ins). La segunda será tendrá el set privado para evitar que se manipule el saldo desde fuera del objeto.

5. Creamos, haciendo uso del generador, un constructor que reciba el parámetro *customerName* e inicialice el saldo a cero.

6. Añadimos al constructor una validación del parámetro *customerName* para lanzar una excepción IllegalArgumentException si el parámetro es nulo. if customerName == null {

throw new IllegalArgumentException("El nombre no puede ser nulo"); }

7. Creamos el método *Credit* que permita incrementar el saldo:

public void Credit(double amount) {

if amount <= 0 {

throw new IllegalArgumentException("La cantidad debe ser mayor que cero"); }

Balance += amount;

}

8. Creamos el método *Debit* que permita retirar saldo de la cuenta: public void Debit(double amount) {

if amount <= 0 {

2

Entornos de Desarrollo – UD3

DAM1

throw new IllegalArgumentException("La cantidad debe ser mayor que cero");

}

if Balance - amount >= 0 {

Balance += amount;

} else {

throw new IllegalArgumentException("Saldo insuficiente");

}

}

**Diseño de pruebas de *Account*.**

9. Análisis de los métodos de Account: constructor, Credit y Debit, mediante la prueba de camino básico.

● Dibujar Grafo de flujo (por ejemplo con draw.io)

● Obtener complejidad ciclomática

● Obtener los caminos independientes.

● Definir las clases de equivalencia para cada método de *Account*.

Diagrama

El contenido generado por IA puede ser incorrecto.

Diagrama

El contenido generado por IA puede ser incorrecto.

Imagen que contiene interior, tabla, alambre, colgando

El contenido generado por IA puede ser incorrecto.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Método** | **Campo** | **Condición de**  **entrada** | **Clase de equivalencia** | **Válida** | **Id** |
| Constructor | Nombre | Lógica | Nulo | No | NV1 |
| Valor | String válido | Sí | V1 |
| Credit | amount | Lógica | Nulo | No | NV1 |
| Lógica | no número | No | NV2 |
| Rango | Negativo | No | NV3 |
| Rango | Positivo | Si | V1 |

Debit amount Lógica Nulo o no número No NV1 Rango Negativo No NV2

Rango Positivo Sí V1

● Definimos los casos de prueba para cada método de *Account*.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Método?** | **Descripción /**  **Requisito** | **Entrada** | **Res. esperado** | **Res.** |
| Constructor | Objeto creado ok /  Nombre correcto **V1** | “Joe Brown” | nombre = “Joe Brown” | Ok / |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Constructor | Objeto no ok /  Nombre nulo | Nombre nulo | Lanza excepción IllegalArgumentException "El nombre no puede ser nulo" | mal |
| Credit | Error Importe negativo | amount = -1 | Lanza excepción  IllegalArgumentException |  |
| Credit | Operación crédito  correcta / Balance correcto después de la operación | amount = 1 | balance = 1 |  |
| Debit | Error / Importe negativo | amount = -1 | Lanza excepción  IllegalArgumentException |  |
| Debit | Incorreto / Balance suficiente | credit amount = 10 debit amount = 15 | Balance = 5 |  |
| Debit | Correcto  Amount > cantidad | Credit = 12  Debit = 8 | Balance = 4 |  |

Interfaz de usuario gráfica, Aplicación

El contenido generado por IA puede ser incorrecto.

Debit Operación débito correcta / Balance correcto

credit amount = 10 debit amount = 5

balance = 5

Credit / Debit

Valores límite

**Creamos la clase de pruebas para *Account*.**

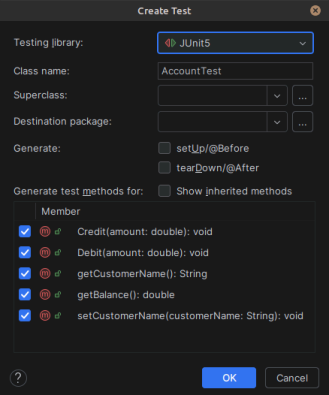
10. Ahora que tenemos la clase *Account* creada vamos a crear un proyecto de pruebas que nos permita llevar a cabo pruebas unitarias automatizadas de dicha clase.

*Observamos que existen distintos tipos de proyectos de pruebas como NUnit o xUnit además del tipo que hemos elegido. Podremos elegir la plataforma que más se ajuste a nuestro proyecto o a la manera de hacer de nuestra organización. Para ello deberíamos revisar las opciones que ofrece cada una y ver cuál es más conveniente para nuestro caso. Como nosotros estamos trabajando con Java elegimos un tipo de proyecto que se ajusta a nuestro caso.*

11. Creamos la clase para los test de *Account.* Con la clase abierta en el menú Code –> Generate.. seleccionamos Test… dejamos el nombre *AccountTest* y seleccionamos crear los test para todos los métodos.

4

Entornos de Desarrollo – UD3 DAM1

12. La clase *AccountTest* se habrá creado por defecto con los test de los métodos de *Account*. Creamos un nuevo método *TestConstructor* para **verificar que el constructor de *Account* funciona correctamente** cuando introducimos un nombre de cliente. Hacemos uso de la clase *Assert* para validar que los valores esperados coinciden con los valores obtenidos por la clase.

@Test

public void TestConstructor(){

String custName = "Joe Brown";

double balance = 0.0;

Account account = new Account(custName);

Assert.*assertArrayEquals*(custName.toCharArray(), account.CustomerName.toCharArray());

Assert.*assertEquals*(balance, account.Balance,0.0);

//Assertions.assertArrayEquals(custName.toCharArray(), account.CustomerName.toCharArray()); //Assertions.assertEquals(balance, account.Balance, 0.0);

// En lugar de varios assert separados es más claro y ordenado un assertAll con varios assert Assertions.*assertAll*("Account creation",

() -> Assertions.*assertArrayEquals*(custName.toCharArray(),account.getCustomerName().toCharArray()), () -> Assertions.*assertEquals*(balance,account.getBalance())

);

}

13. Ejecutamos la clase *AccountTest* y vemos que la prueba se ejecuta de manera exitosa.

5

Entornos de Desarrollo – UD3 DAM1

**Validar excepciones.**

14. Creamos ahora un **método de prueba para comprobar el tipo de excepción** lanzada al proporcionar un nombre nulo al constructor.. Haremos uso del método *assertThrows* para comprobar si recibimos la excepción y es del tipo esperado y el método *assertTrue* para comprobar el mensaje obtenido. Si ambos son iguales a lo esperado el caso de prueba será evaluado como correcto y pasaremos el test.

@Test

public void TestConstructor\_NullName() {

Exception exception =

Assertions.*assertThrows*(IllegalArgumentException.class, () -> { Account account = new Account(null);

});

String expectedMessage = "El nombre no puede ser nulo"; String actualMessage = exception.getMessage();

Assertions.*assertTrue*(actualMessage.contains(expectedMessage)); }

15. Creamos las **pruebas para los métodos *Credit* y *Debit***. Al crear la clase el IDE automáticamente nos ha creado los métodos de test, pero estos métodos están vacíos y los nombres son iguales a los de la clase original. Vamos a darles nuevos nombres *testCredit* y *testDebit* y añadir el código para los test. @Test

void testCredit(){

String custName = "Joe Brown";

double credit = 10.00;

Account account = new Account(custName);

account.Credit(credit);

Assertions.*assertEquals*(credit, account.Balance);

}

@Test

void testDebit() {

String custName = "Joe Brown";

double credit = 10.00;

double debit = 5.00;

Account account = new Account(custName);

account.Credit(credit);

account.Debit(debit);

Assertions.*assertEquals*(credit-debit, account.Balance); }

6

Entornos de Desarrollo – UD3

DAM1

16. **Ejecutamos las pruebas** y vemos que el *testDebit* ha fallado, revisamos el mensaje de error y descubrimos que tenemos un bug en el método *debit* de nuestra clase (probablemente de aplicar sin cuidado un copy / paste al hacer nuestro código). Corregimos la clase *Account* y probamos ejecutamos de nuevo los tests.

//Código incorrecto para una retirada de efectivo Balance += amount;

//Código correcto para una retirada de efectivo Balance -= amount;

17. Vamos a crear dos nuevos **métodos de test para comprobar la excepción** lanzada cuando el signo de los importes pasados como argumento a los métodos *credit y debit* no son correctos.

@Test

public void Test\_Credit\_Negative()

{

String custName = "Joe Brown";

double credit = -100.00;

Account account = new Account(custName);

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

account.Credit(credit);

});

String expectedMessage = "La cantidad debe ser mayor que cero";

String actualMessage = exception.getMessage();

Assertions.*assertTrue*(actualMessage.contains(expectedMessage));

}

@Test

public void Test\_Debit\_Negative()

{

String custName = "Joe Brown";

double debit = -100.00;

Account account = new Account(custName);

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

account.Debit(debit);

});

String expectedMessage = "La cantidad debe ser mayor que cero";

String actualMessage = exception.getMessage();

Assertions.*assertTrue*(actualMessage.contains(expectedMessage));

}

18. Creamos otro método test para probar la excepción en caso de saldo insuficiente.

7

Entornos de Desarrollo – UD3 DAM1

@Test

public void Test\_Debit\_InsufficientBalance()

{

String custName = "Joe Brown";

double credit = 10.00;

double debit = 50.00;

Account account = new Account(custName);

Exception exception = Assertions.*assertThrows*(IllegalArgumentException.class, () -> { account.Credit(credit);

account.Debit(debit);

});

String expectedMessage = "Saldo insuficiente";

String actualMessage = exception.getMessage();

Assertions.*assertTrue*(actualMessage.contains(expectedMessage));

}

19. Volvemos a ejecutar la clase de Test y comprobamos que todas las pruebas pasan correctamente.

**~~Test controlados por datos.~~**

~~20. Primero necesitamos instalar la librería de Parametrized.~~ ~~Abrimos el menú~~ ~~Project Structure y en Libraries añadimos una nueva librería~~

com.tngtech.junit.dataprovider:junit-dataprovider-core:2.10

**Tipos de test.**

21. En el menú Run → Edit Configurations… podemos: ver, parametrizar y ejecutar todas las **configuraciones de los test de JUnit**. Documentación oficial. Además podemos:

● Crear nuevas configuraciones de ejecución.

● Guardar la configuración de las pruebas en un fichero para intercambiarlo. ● Podemos definir el tipo de pruebas, tenemos las siguientes:

i. **All in package**: Todos los test del paquete.

ii. **All in directory**: Todos los test del directorio.

iii. **Pattern**: Un conjunto de clases o métodos de test seleccionados en todo el proyecto mediante patrones. Se pueden crear **suites de tests**, es decir varias clases que se ejecutarán juntas. Para crearla hacemos click en el

botón Expand en la derecha y escribimos las clases a incluir en la suite. Se creará una nueva clase con la anotación @Suite.

iv. **Class**: Todos los test de una clase.

v. **Method**: Un método individual.

vi. **Category**: Todos los test de una categoría.

vii. **UniqueId**: Incluye tests y contenedores containers con un ID específico. viii. **Tags**: Clases y métodos etiquetados con la anotación @Tag.

8

Entornos de Desarrollo – UD3

DAM1

**Pruebas de rendimiento.**

22. **Pruebas de rendimiento**. Por ejemplo comprobar el tiempo que tarda un método en ejecutarse.

@Test

void testTiempoEjecucion() {

*assertTimeout*(Duration.*ofMillis*(100), () -> { // Código que debería ejecutarse en menos de 100ms Thread.*sleep*(50);

});

}

23. Realizar las siguientes pruebas (una distinta por cada punto, quitar el sleep(50)): ● La creación de un objeto cuenta

● La creación de 100 cuentas distintas con un bucle for.

● La creación de una cuenta, un crédito y un débito.

● La creación de una cuenta, 100 créditos y un débitos con importes distintos.

24. Al ejecutar los test IntelliJ nos muestra el tiempo de ejecución de cada prueba. Revisar el tiempo de ejecución en cada caso y comprobar si son coherentes, es decir: el tiempo para crear 100 cuentas es aproximadamente 100 veces el de una cuenta.

Entregables

● Código del proyecto:

○ Clase Account y clase de pruebas AccountTest

● Análisis

○ Dibujar Grafo de flujo (por ejemplo con draw.io)

○ Obtener complejidad ciclomática (Nº nodos, aristas, nodos predicado y regiones)

○ Obtener los caminos independientes.

○ Tabla con la definición de los casos de prueba para *Account*.

● Casos de prueba:

○ Método, Id prueba, datos de entrada, salida esperada y resultado de cada caso.

○ Pantallazo(s) de los resultados de las pruebas.

● Pruebas de rendimiento.

○ Casos de prueba.

○ Resultados.

9