

CICLO FORMATIVO DE GRADO SUPERIOR DE DESARROLLO DE APLICACIONES WEB

0487- ENTORNOS DE DESARROLLO UNIDAD DE PROGRAMACIÓN 05 – HERRAMIENTAS PARA LA PRUEBA DE SOFTWARE

IES de Alicante (Comunidad Valenciana)

Curso académico 2024-2025



Docente: Arturo Albero Gabriel – 53241833F

UP05	Herramientas para la prueba de software
Resultado de aprendizaje 3	Todos los CE del RA3
Competencias profesionales y para la empleabilidad	a), m), r), s)
Objetivos de ciclo	a), ñ), t)
Actividades formativas	
Actividad de introducción	La fase de pruebas
Actividades no evaluables	<p>Trazas de software</p> <p>Uso del debugger en IntelliJ</p> <p>Tipos de pruebas</p> <p>Pruebas de unidad con Junit.</p> <p>Pruebas de integración y dobles de pruebas con Mockito</p> <p>Pruebas de aceptación y validación con Gherkin y cucumber</p> <p>Documentación de pruebas</p>
Actividades evaluables	<p>Cooperativo: Diseño de pruebas para el proyecto</p> <p>Individual: Ejercicios de trazas</p> <p>Prueba objetiva: Test de 20 preguntas</p> <p>Prueba objetiva: Diseño de pruebas</p> <p>Prueba objetiva: Ejercicio de trazas</p>
Actividades de refuerzo	Cuaderno de ejercicios y tutoriales
Actividades de consolidación	Problemas intercalados en los documentos
Actividades de recuperación	Repetición de las actividades evaluables
Actividades ampliación	<p>Pruebas unitarias en Python y C#</p> <p>Pruebas de seguridad</p>
Materiales	Apuntes web y pdf; Aules; Mockito; Junit5; Cucumber; IntelliJ IDEA; Java 21 LTS; Office 365; Maven

ÍNDICE

ACTIVIDAD DE INTRODUCCIÓN	Pág 1
La fase de pruebas	
ACTIVIDADES DE EXPOSICIÓN y CONSOLIDACIÓN	
La traza de un programa	Pág. 2
Depuración de software: El Debugger	Pág. 11
Tipos de pruebas de software	Pág. 16
Herramientas para la prueba de software – JUNIT	Pág. 27
Pruebas de integración – Mockito	Pág. 38
Pruebas de aceptación – Cucumber	Pág. 48
Reto cooperativo	Pág. 55
Reto individual	Pág. 58
Modelo de examen	Pág. 61
ACTIVIDADES DE REFUERZO – Cuaderno de ejercicios	Pág. 68
ACTIVIDADES DE AMPLIACIÓN	
Pruebas en Python	Pág. 73
Pruebas de Seguridad	Pág. 83

La fase de pruebas

Antes de poner un software en producción, es esencial comprobar que funciona correctamente. Las pruebas permiten detectar errores, asegurar que el sistema cumple con los requisitos y garantizar que sea fiable y estable. No basta con “parece que funciona”: hay que probarlo de forma rigurosa.

En esta unidad vamos a conocer herramientas que nos ayudan a probar el software de manera sistemática y eficaz, concretamente:

- Herramientas para ejecutar pruebas manuales y automatizadas
- Herramientas de análisis estático y dinámico del código
- Herramientas para medir la cobertura de pruebas
- Entornos de prueba y técnicas de validación

Actividad

Reflexiona sobre la siguiente pregunta:

¿Por qué crees que encontrar errores durante las pruebas es mejor que encontrarlos cuando el software ya está en uso?

Después, debate en clase los siguientes puntos:

- ¿Qué tipo de errores crees que son más difíciles de detectar sin herramientas de prueba?
- ¿Qué ventajas ofrece automatizar las pruebas?
- ¿Cómo afecta la calidad de las pruebas a la calidad del software?
- ¿Crees que los desarrolladores deben ser también responsables de probar su código?

¿Por qué?

La traza de un programa

- [La traza de un programa](#)
 - [Definición de Traza de un programa](#)
 - [Tabla de seguimiento de variables](#)
 - [Ejemplo de aplicación](#)
 - [Pruebas de caja blanca](#)
 - [Pruebas de camino básico](#)
 - [Pruebas condicionales](#)
 - [Pruebas de comprobación de bucles](#)

Definición de Traza de un programa

La traza de un programa es un registro detallado de la ejecución de un programa, que muestra los valores de las variables, los resultados de las operaciones y el flujo de control en cada paso. En ella implementamos las técnicas de depuración de caja blanca de manera holística.

Es una herramienta útil para depurar y entender cómo funciona un programa, especialmente cuando se busca identificar errores o comportamientos inesperados.

Tabla de seguimiento de variables

Una de las técnicas más efectivas es crear una tabla donde se registren los valores de las variables en cada paso del algoritmo. Supongamos el siguiente algoritmo en lenguaje natural:

1. Iniciar con $a = 5$ y $b = 3$.
2. Sumar a y b , guardar el resultado en c .
3. Multiplicar c por 2, guardar el resultado en d .
4. Mostrar el valor de d .

Dado el anterior algoritmo, realizamos la siguiente tabla donde hacemos un seguimiento del valor de las variables.

Paso	Operación	a	b	c	d
1	Iniciar $a = 5$, $b = 3$	5	3	-	-
2	$c = a + b$ ($5 + 3$)	5	3	8	-

Paso	Operación	a	b	c	d
3	$d = c * 2 (8 * 2)$	5	3	8	16
4	Mostrar d	5	3	8	16

Usando Entornos de Desarrollo Integrado, esta información la podemos consultar a través del depurador (debugger). Podemos sustituir la tabla por anotaciones (con la misma información) si el problema es simple, para reducir la aparatosidad. Las anotaciones son otra forma de expresar la traza.

Además del valor de las variables, también podemos evaluar el camino que se toma en los nodos de decisión mediante el uso de tablas de verdad.

Ejemplo de aplicación

Vamos a crear un **ejemplo sencillo** que combine un algoritmo, su traza y una tabla de verdad para evaluar el camino que se toma en los nodos de decisión y el número de veces que se repite un bucle. Supongamos el siguiente algoritmo:

1. Iniciar con $\backslash(x = 2 \backslash)$ y $\backslash(y = 3 \backslash)$.
2. Si $\backslash(x > y \backslash)$, mostrar "x es mayor que y".
3. Si $\backslash(x < y \backslash)$, mostrar "x es menor que y" y repetir el siguiente paso 2 veces:
 - Incrementar $\backslash(x \backslash)$ en 1.
4. Mostrar el valor final de $\backslash(x \backslash)$.

Realizamos la traza siguiendo paso a paso el algoritmo y anotar los valores de las variables y las decisiones tomadas.

1. **Paso 1**: $\backslash(x = 2 \backslash)$, $\backslash(y = 3 \backslash)$.
2. **Paso 2**: Evaluar $\backslash(x > y \backslash)$:
 - $\backslash(2 > 3 \backslash)$ es **Falso**, no se ejecuta esta condición.
3. **Paso 3**: Evaluar $\backslash(x < y \backslash)$:
 - $\backslash(2 < 3 \backslash)$ es **Verdadero**, se muestra "x es menor que y".
 - Se entra en el bucle que se repite 2 veces:
 - **Primera iteración**:
 - Incrementar $\backslash(x \backslash)$ en 1: $\backslash(x = 3 \backslash)$.
 - **Segunda iteración**:
 - Incrementar $\backslash(x \backslash)$ en 1: $\backslash(x = 4 \backslash)$.
4. **Paso 4**: Mostrar el valor final de $\backslash(x \backslash)$, que es **4**.

A continuación, creamos una tabla de verdad para las condiciones $x > y$ y $x < y$, y vemos qué camino se toma en cada caso.

x	y	$x > y$	$x < y$	Camino tomado
2	3	Falso	Verdadero	Mostrar "x es menor que y", bucle
3	3	Falso	Falso	No se cumple ninguna condición
4	3	Verdadero	Falso	Mostrar "x es mayor que y"

Podemos emplear las trazas para cualquier algoritmo que diseñemos y nos serán especialmente útiles para detectar errores o comportamientos anómalos.

Actividad: Realiza un algoritmo que, dada una fecha de entrada, determine si es válida y, en caso de que sea válida, qué signo del zodiaco tendría un bebé nacido ese día. Para calcular si es válida una fecha, debes tener en cuenta los números de día y mes introducidos, contemplando si se trata de un año bisiesto o no.

Un año es bisiesto si es divisible entre 4, con las excepciones de los años que son divisibles entre 100, que solo serán bisiestos si también son divisibles entre 400 (1900 no es bisiesto, pero 2000 y 2004 sí lo son).

Las fechas para determinar el signo del zodiaco son las siguientes:

1. **Aries:** 21 marzo - 19 abril.
2. **Tauro:** 20 abril - 20 mayo.
3. **Géminis:** 21 mayo - 20 junio.
4. **Cáncer:** 21 junio - 22 julio.
5. **Leo:** 23 julio - 22 agosto.
6. **Virgo:** 23 agosto - 22 septiembre.
7. **Libra:** 23 septiembre - 22 octubre.
8. **Escorpio:** 23 octubre - 21 noviembre.
9. **Sagitario:** 22 noviembre - 21 diciembre.
10. **Capricornio:** 22 diciembre - 19 enero.
11. **Acuario:** 20 enero - 18 febrero.
12. **Piscis:** 19 febrero - 20 marzo.

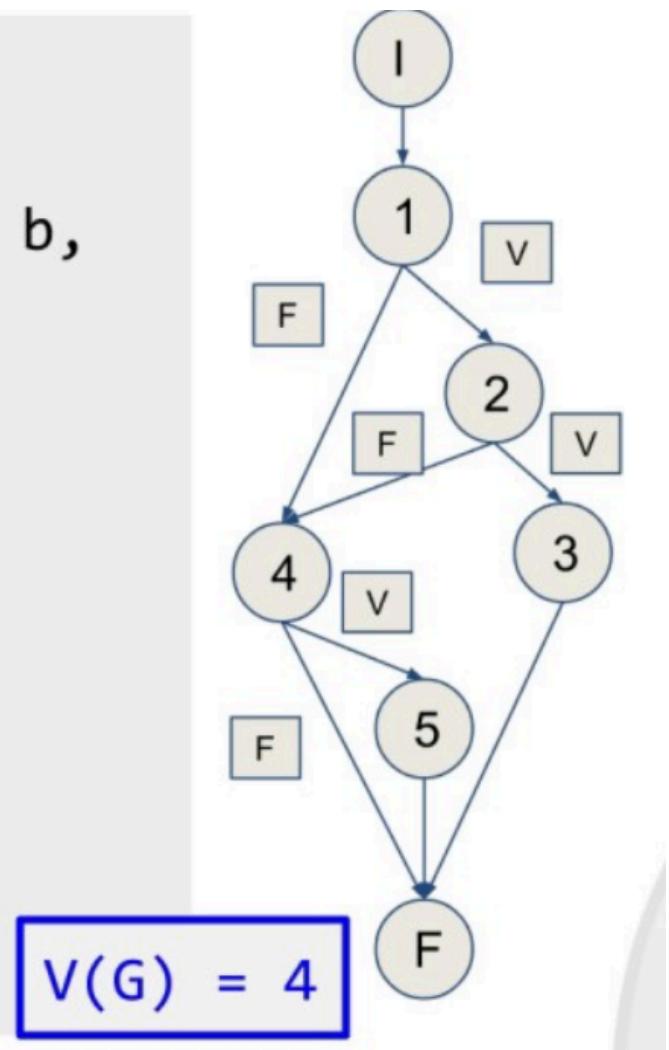
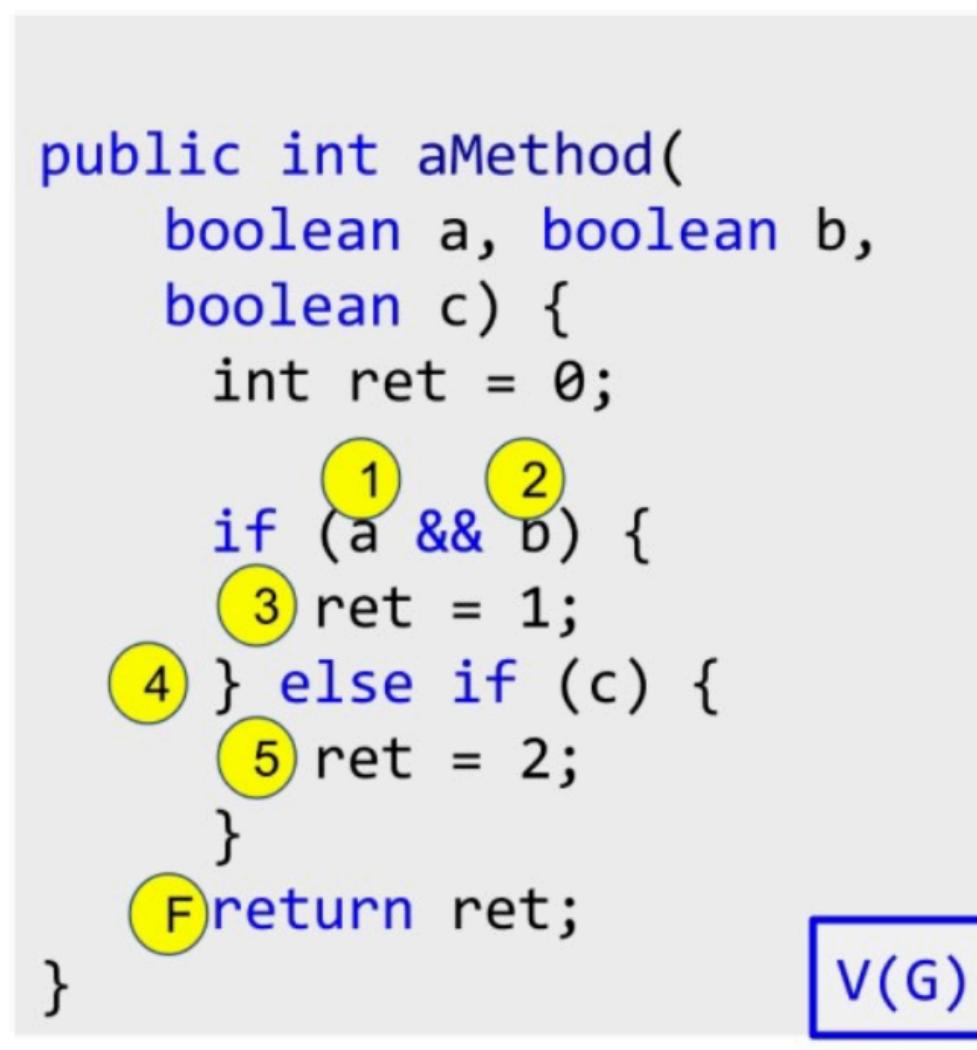
Calcula las trazas para los valores 29 de febrero de 2000, 29 de febrero de 1999, 31 de abril de 2024, 7 de abril de 1787, 1 de enero de 1900 y 13 de marzo de 2032.

Pruebas de caja blanca

Las pruebas de caja blanca son, como veremos más adelante, un tipo de pruebas de unidad que se encargan de comprobar el funcionamiento interno del código. Funcionan codo a codo con las trazas y, en combinación, son la base para el uso adecuado de un depurador.

Pruebas de camino básico

Las pruebas de camino básico tienen como fin determinar el camino que puede seguir la ejecución de un código. Para ello, se organiza cada operación atómica (de un solo paso) del código en un diagrama de árbol, como se ve en la siguiente imagen:



Se llama complejidad ciclomática, representada con $V(G)$, al número de caminos distintos que puede tomar el programa. Los caminos sirven para poder diseñar pruebas de unidad que verifiquen el funcionamiento de cada uno de ellos.

Actividad:

Diseña pruebas de camino básico para el siguiente algoritmo:

```

if (num1 > 10)
{
    if (num2 > 10)
        System.out.println("Ambos son mayores");
    else
        System.out.println("El primero es mayor");
} else {
    if (num2 > 10)
        System.out.println("El segundo es mayor");
    else
        System.out.println("Ninguno es mayor");
}

```

Pruebas condicionales

Las pruebas condicionales analizan el camino que puede seguir la ejecución de un código centrándose en las condiciones del mismo. Para evaluar dichos caminos, la herramienta que se emplea son las **tablas de verdad**.

Una tabla de verdad es una herramienta utilizada en lógica proposicional para representar todos los posibles valores de verdad de una expresión lógica, en función de los valores de verdad de sus variables. Podemos etiquetar las condiciones que vayamos encontrando y realizar una tabla que nos señale los caminos posibles. Asimismo, también podemos anotar las veces que se repite un bucle en función de los valores de entrada.

```

public boolean esAnyoBisiesto(int anyo)
{
    boolean esAnyoBisiesto = false;
    if(anyo % 4 == 0)
    {
        esAnyoBisiesto = true;

        if(anyo % 100 == 0)
        {
            esAnyoBisiesto = false;

            if(anyo % 400 == 0)
            {
                esAnyoBisiesto = true;
            }
        }
    }
    return esAnyoBisiesto;
}

```

A partir del código anterior, etiquetamos las condiciones para crear una tabla de verdad:

- C1 = anyo % 4 == 0;
- C2 = anyo % 100 == 0;
- C3 = anyo % 400 == 0;

Y creamos la siguiente tabla:

N	C1	C2	C3	esAnyoBisiesto
1	true	true	true	true
2	true	true	false	false
3	true	false	true	true
4	true	false	false	true
5	false	true	true	false
6	false	true	false	false
7	false	false	true	false

N	C1	C2	C3	esAnyoBisiesto
8	false	false	false	false

Si nos fijamos bien, los casos 3 y 4 llevan al mismo resultado independientemente de C3 y algo similar ocurre con los casos del 5 al 8, que dependen de C1, por lo que podemos simplificar la tabla de la siguiente forma:

N	C1	C2	C3	esAnyoBisiesto
1	true	true	true	true
2	true	true	false	false
3	true	false	*	true
4	false	*	*	false

Actividad:

Utiliza el código del ejercicio del apartado anterior para realizar las pruebas condicionales.

Pruebas de comprobación de bucles

Esta prueba evalúa los posibles caminos para los bucles. Para cada bucle con n iteraciones, debemos verificar si:

- El bucle nunca se ejecuta.
- El bucle se ejecuta solo una vez.
- El bucle se ejecuta dos veces.
- El bucle se ejecuta m veces, siendo $m < n$.
- El bucle se ejecuta n y $n-1$ veces.

Si hay algún bucle anidado, debemos comenzar explorando los bucles internos y luego pasar a los externos.

Por ejemplo, observemos el siguiente código que verifica si un número dado (introducido previamente por el usuario) es primo o no:

```

boolean result = true;
if (number == 0 || number == 1)
    result = false;
int i = 2;
while (i <= number / 2 && result)
{
    if (number % i == 0)
        result = false;
    else
        i++;
}

```

Se espera que el bucle se ejecute como máximo hasta $N = \text{number} / 2 - 1$ veces. Según el enfoque de prueba de bucles, debemos diseñar casos de prueba en los que:

- **El bucle nunca se ejecuta.** Por ejemplo, si el número es 2, automáticamente es primo y no se realiza ninguna iteración.
También podríamos probar los casos de 0 y 1, que están cubiertos por la primera cláusula *if*.
- **El bucle se ejecuta una vez.** Esto se puede lograr con *number* = 3.
- **El bucle se ejecuta dos veces.** Por ejemplo, con *number* = 9.
- **El bucle se ejecuta m veces, donde $m < N$.** Por ejemplo, con *number* = 25, el bucle se ejecuta 4 veces.
- **El bucle se ejecuta N veces y/o $N-1$ veces.** Para alcanzar N iteraciones, solo necesitamos un número primo, como 23.

Para iterar $N-1$ veces, necesitamos un número compuesto que no se descubra hasta la última iteración.

En este caso, podríamos usar *number* = 4, aunque es un caso de prueba bastante simple.

Podemos construir la siguiente tabla de casos de prueba:

ID	Nombre	Datos	Resultado esperado	Resultado real
U0	BasicCases	1	false	
U1	Niterations	2	true	
U2	Onelteration	3	true	
U3	Twolterations	9	false	
U4	MIterations	25	false	

ID	Nombre	Datos	Resultado esperado	Resultado real
U5	N-1Iterations	4	false	
U6	NIterations	23	true	

Actividad:

El siguiente fragmento de código verifica si los dígitos de un número están en orden ascendente:

```
boolean result = true;
while (number >= 10 && result)
{
    int lastDigit = number % 10;
    number /= 10;
    int newLastDigit = number % 10;
    if (lastDigit < newLastDigit)
        result = false;
}
```

Se te pide diseñar una tabla de casos de prueba considerando todas las posibles iteraciones del bucle, siguiendo el ejemplo anterior.

Información extraída de [esta página web](#).

- [Depuración de Software: El Debugger](#)
- [Uso del debugger de IntelliJ](#)

Depuración de Software: El Debugger

Actividad: Para comprender el funcionamiento del Debugger de IntelliJ, realiza una memoria por escrito de las actividades de este apartado. Usa capturas de pantalla para copiar los resultados obtenidos.

Podemos hacer trazas en programas añadiendo salidas por consola de variables o datos. Esto puede ser efectivo para hacernos una idea de lo que está sucediendo en el código, pero no es muy práctico para hacer un análisis pormenorizado del comportamiento de nuestro programa. En muchos programas es más adecuado usar un **debugger**, que generalmente está integrado en cualquier IDE.

Un **debugger** es una herramienta para inspeccionar y analizar el comportamiento de un programa mientras se ejecuta. De esta forma, nos permite identificar y corregir errores en el código (bugs).

Para controlar el flujo de un programa, indicamos puntos de ruptura (breakpoints), en los cuales se detiene el programa. También podemos realizar ejecuciones paso a paso.

En todo momento, podemos controlar el estado del programa gracias a esta herramienta de depuración.

Uso del debugger de IntelliJ

Vamos a comprobar el funcionamiento del Debugger con este código en IntelliJ. Crea un proyecto y copia este código:

```

1 import java.util.ArrayList;
2 import java.util.HashMap;
3 import java.util.List;
4 import java.util.Map;
5
6 public class Main {
7     public static void main(String[] args) {
8         List<Integer> primos = new ArrayList<>();
9         Map<Integer, Integer> divisibles = new HashMap<>();
10
11        for(int i = 0; i< 50; i++){
12            boolean esPrimo = true;
13            int divisores = 0;
14            for(int j = 2; j<i; j++){
15                if(i % j == 0){
16                    esPrimo = false;
17                    divisores++;
18                }
19            }
20            if (esPrimo){
21                primos.add(i);
22            }else{
23                divisibles.put(i, divisores);
24            }
25        }
26        for(Integer i : primos){
27            System.out.println("El número " + i + " es primo");
28        }
29        for(Integer i : divisibles.keySet()){
30            int divisores = divisibles.get(i);
31            String msg = "El número " + i + " no es primo y tiene " + divisores;
32            msg += (divisores>1)? " divisores.":" divisor.";
33            System.out.println(msg);
34        }
35    }
36}

```

Lo primero que debemos hacer es añadir un punto de ruptura. Para ello hacemos click en la línea 11. Se añade un punto rojo, que es el punto de ruptura. Al hacer click en la ejecución de Debug, al llegar a ese punto el programa se detendrá y nos permitirá inspeccionar las variables.

Los puntos de ruptura pueden ser condicionales o incondicionales. Para añadir una condición a un punto de ruptura, haz click derecho. En la línea 20, añade la condición `esPrimo` para que el programa solo se detenga cuando la variable `esPrimo` sea cierta.

Actividad:

Comprueba el funcionamiento de los dos puntos de ruptura. Añade un punto más, siendo uno incondicional y el otro condicional. Despues bórralos haciendo click.

Cuando activas la depuración, la información se muestra en la pantalla inferior (donde está normalmente la consola):



En este menú podemos ver los puntos de ruptura, realizar acciones de ejecución paso a paso (que veremos más adelante) y comprobar el valor de las variables.

Actividad:

Inspecciona el valor de las variables en el punto de ruptura de la línea 11 cuando `i` vale `12`. ¿Qué información tenemos que no aparece en nuestro código?

Una vez establecido un punto de ruptura, al parar podemos continuar la ejecución del programa paso a paso de las siguientes formas:

- **Step Over:** Ejecuta la línea actual y pasa a la siguiente, sin entrar en los detalles de las funciones llamadas.
- **Step Into:** Entra en el código de las funciones llamadas para inspeccionarlas.
- **Step Out:** Sale de la función actual y regresa al punto donde fue llamada.

Además, contamos con la **Pila de Llamadas** (Call Stack):

- Muestra la secuencia de llamadas a funciones que llevaron al punto actual de ejecución.
- Te ayuda a entender cómo llegaste a un punto específico en el código.

Todas estas opciones están en el menú del debugger. Para comprobar cómo funcionan, vamos a modificar el código de la siguiente forma, añadiendo funciones:

```

1 import java.util.ArrayList;
2 import java.util.HashMap;
3 import java.util.List;
4 import java.util.Map;
5
6 public class Main {
7     public static void main(String[] args) {
8         List<Integer> primos = new ArrayList<>();
9         Map<Integer, Integer> divisibles = new HashMap<>();
10
11        for(int i = 0; i< 50; i++){
12            boolean esPrimo = true;
13            int divisores = 0;
14            for(int j = 2; j<i; j++){
15                if(i % j == 0){
16                    esPrimo = false;
17                    divisores++;
18                }
19            }
20            if (esPrimo){
21                primos.add(i);
22            }else{
23                divisibles.put(i, divisores);
24            }
25        }
26        visualizar(primos);
27        visualizar(divisibles);
28    }
29    private static void visualizar(List<Integer> primos){
30        for(Integer i : primos){
31            System.out.println("El número " + i + " es primo");
32        }
33    }
34    private static void visualizar (Map<Integer, Integer> divisibles){
35        for(Integer i : divisibles.keySet()){
36            int divisores = divisibles.get(i);
37            String msg = "El número " + i + " no es primo y tiene " + divisores;
38            msg += (divisores>1)? " divisores.:" divisor.";
39            System.out.println(msg);
40        }
41    }
42}

```

Añade un punto de ruptura en la línea 26 y otro en la línea 38.

Actividad:

Prueba las funciones de Step Over, Step Into (desde la línea 26) y Step Out (desde la línea 38). Observa la pila de llamadas y también los valores.

- Prueba primero el *Step Over* desde la línea 26, avanzando 3 pasos. Despues, vuelve a ejecutar el debug y prueba el *Step Into* tambien con tres pasos. ¿Qué diferencia hay?
- Despues prueba el *Step Out* desde la línea 38. Describe qué sucede.

Tipos de pruebas de Software

En este documento vamos a introducirnos en una de las fases más importantes del desarrollo de software, la fase de pruebas. Introduciremos los diferentes tipos de pruebas que se pueden realizar y más adelante profundizaremos en ellas.

- [Tipos de pruebas de Software](#)
 - [Las pruebas de software dentro del desarrollo de un sistema](#)
 - [Objetivos y metodologías de testeo](#)
 - [Etapas del testeo](#)
 - [Pruebas de unidad](#)
 - **Características clave**
 - [Pruebas de integración](#)
 - **Características clave**
 - **Tipos de pruebas de integración**
 - [Pruebas de CI/CD](#)
 - **Características clave**
 - **Flujo de trabajo**
 - [Pruebas de aceptación](#)
 - **Características clave**
 - **Tipos de pruebas de aceptación**
 - [Pruebas de validación](#)
 - **Características clave**
 - **Tipos de pruebas de validación**
 - [Pruebas exploratorias](#)
 - [Estándares de pruebas de software](#)
 - **ISO 25010 - Modelo de Calidad del Software**
 - **ISTQB - Certificación en Testing de Software**
 - **Niveles de Pruebas**
 - **Tipos de Testing**
 - **Principios del Testing**
 - [Documentación de Pruebas](#)
 - [Factores de importancia para la documentación de las pruebas](#)
 - [Elementos clave de la documentación de pruebas](#)

Las pruebas de software dentro del desarrollo de un sistema

El desarrollo de software sigue una serie de fases para garantizar que el producto final sea funcional, eficiente y libre de errores. Una de las fases fundamentales es la **prueba y depuración del software**, donde se verifica que el código cumple con los requisitos y funciona correctamente en diferentes escenarios.

Las pruebas de software permiten detectar errores antes de que el sistema llegue a producción, reduciendo los costos y riesgos asociados a fallos en entornos reales. Para ello, existen diversas metodologías y enfoques que buscan garantizar la calidad del software en diferentes niveles.

Las pruebas de software son una parte fundamental del desarrollo, ya que permiten garantizar la calidad, seguridad y funcionalidad de una aplicación antes de que llegue a los usuarios finales. Un error no detectado a tiempo puede provocar fallos graves, pérdidas económicas y problemas de reputación para una empresa. Es por ello que el proceso de testeo es clave en cualquier proyecto de software.

Objetivos y metodologías de testeo

El objetivo principal del testeo de software es **detectar y corregir errores** para asegurar la calidad del producto final. Además, el testeo permite verificar que el software cumple con los requisitos funcionales y no funcionales establecidos en la fase de diseño.

Para alcanzar estos objetivos, se emplean diferentes metodologías de prueba:

- **Pruebas de caja blanca:** Se basan en el conocimiento interno del código y su estructura. Se analizan las rutas de ejecución, estructuras de control y dependencias internas del programa.
- **Pruebas de caja negra:** Se enfocan en evaluar la funcionalidad del software sin conocer su implementación interna. Se prueban los casos de entrada y salida para validar que el sistema responde correctamente.
- **Pruebas manuales y pruebas automatizadas:** Las pruebas pueden realizarse de forma manual, con la intervención de testers que evalúan el sistema, o de forma automatizada, usando herramientas como JUnit para ejecutar test de manera programada.

Etapas del testeo

El proceso de testeo se realiza en varias etapas, desde las pruebas iniciales en el código hasta la validación final del producto:

1. **Pruebas de unidad:** Se prueban componentes individuales del código, como funciones o clases, para verificar que funcionan de manera aislada.
2. **Pruebas de integración:** Se verifica la interacción entre módulos o componentes del software para asegurar su correcto funcionamiento en conjunto.
3. **Pruebas de sistema:** Se prueba el software en su totalidad, evaluando su comportamiento en diferentes escenarios y condiciones.
4. **Pruebas de aceptación y validación:** Se comprueba que el software cumple con los requisitos del cliente y está listo para su despliegue.

Pruebas de unidad

Las **pruebas de unidad** son el nivel más básico de testeo y se centran en evaluar el correcto funcionamiento de componentes individuales del código, como funciones, métodos o clases. Estas pruebas se realizan de manera aislada, sin depender de otros módulos del sistema, lo que permite detectar errores en partes específicas del código antes de integrarlas con el resto del software.

Características clave

- Se enfocan en probar unidades pequeñas e independientes.
- Se ejecutan de forma rápida y frecuente.
- Permiten identificar errores en etapas tempranas del desarrollo.
- Son generalmente automatizadas con herramientas de testing.

En **Java**, la herramienta más utilizada para realizar pruebas de unidad es **JUnit**, que permite definir pruebas automatizadas y ejecutarlas repetidamente. Un ejemplo básico de test con JUnit sería:

```

import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.api.Test;

public class CalculadoraTest {
    @Test
    public void testSuma() {
        Calculadora calc = new Calculadora();
        assertEquals(10, calc.sumar(5, 5));
    }
}

```

En este caso, el test verifica que el método `sumar()` de la clase `Calculadora` devuelve el valor esperado. Si el resultado no coincide, la prueba fallará, indicando un error en la implementación.

Pruebas de integración

Las **pruebas de integración** verifican la interacción entre diferentes módulos o componentes del software. Aunque cada unidad puede haber pasado sus pruebas individuales, es fundamental asegurarse de que, al trabajar juntas, se comuniquen correctamente y no generen errores inesperados.

Características clave

- Se enfocan en la interacción entre componentes.
- Pueden incluir bases de datos, API externas y otros módulos del sistema.
- Se realizan después de las pruebas de unidad.
- Ayudan a detectar problemas de compatibilidad y dependencias.

Tipos de pruebas de integración

- **Big Bang:** Se integran todos los módulos a la vez y se prueba el sistema completo.
- **Top-down:** Se prueban primero los módulos de alto nivel y luego los de menor nivel.
- **Bottom-up:** Se prueban primero los módulos de bajo nivel y luego los de mayor nivel.
- **Sandwich (híbrido):** Mezcla de top-down y bottom-up.

En Java, herramientas como **Mockito** permiten simular dependencias y realizar pruebas de integración sin necesidad de depender de servicios externos.

Pruebas de CI/CD

Las **pruebas en entornos de Integración Continua (CI) y Entrega Continua (CD)** permiten garantizar que los cambios en el código se integren y desplieguen sin introducir errores.

Características clave

- Permiten detectar fallos automáticamente en cada cambio del código.
- Se integran con herramientas de desarrollo como [GitHub Actions](#), [Jenkins](#) o [GitLab CI/CD](#).
- Se ejecutan en servidores automatizados.
- Mejoran la estabilidad del software y reducen riesgos en la entrega.

Flujo de trabajo

1. Un desarrollador sube cambios al repositorio de código (Git).
2. Se ejecutan pruebas de unidad y de integración automáticamente (CI).
3. Si las pruebas pasan, el software se despliega en un entorno de prueba o producción (CD).

Pruebas de aceptación

Las **pruebas de aceptación** verifican que el software cumple con los requisitos funcionales establecidos por el cliente o los usuarios finales. Son una parte crucial antes del lanzamiento del producto.

Características clave

- Se centran en la experiencia del usuario y en la funcionalidad esperada.
- Pueden ser realizadas manualmente o automatizadas.
- Incluyen escenarios de uso reales.
- Generalmente son ejecutadas por un equipo de QA o por el cliente.

Tipos de pruebas de aceptación

- **Pruebas Alfa:** Se realizan en un entorno de desarrollo, con usuarios internos o testers.
- **Pruebas Beta:** Se realizan con usuarios reales en un entorno de producción limitado.

Herramientas como **Cucumber** permiten escribir pruebas en lenguaje natural usando la metodología **BDD (Behavior-Driven Development)**.

Cucumber emplea una sintaxis especial llamada Gherkin. Gherkin está traducido a 70 idiomas o sabores. Algunos son serios, otros son más en código de broma. [Aquí tienes](#) cómo traducir las palabras clave a diferentes lenguajes.

Actividad: Lee el siguiente ejemplo en Gherkin para cucumber. Sus palabras clave están en inglés. Tradúcelas al castellano siguiendo la sintaxis facilitada en la web. Tradúcelas también a catalán, a inglés en jerga pirata y a inglés en jerga de texas profundo. Parece de broma, y en parte lo es, pero todas estas traducciones son posibles.

Feature: Login de usuario

Scenario: Usuario ingresa credenciales válidas

Given el usuario está en la página de login

When ingresa su usuario "**juan**" y contraseña "**1234**"

Then debe ver la página de inicio

Este tipo de pruebas facilita la validación con clientes y usuarios no técnicos.

Pruebas de validación

Las **pruebas de validación** aseguran que el software cumple con todas las especificaciones y normativas antes de su despliegue. Son un paso esencial para garantizar la calidad del producto final.

Características clave

- Verifican que el software cumpla con los requisitos técnicos y legales.
- Aseguran la estabilidad y seguridad del sistema.
- Se realizan antes del lanzamiento oficial del software.

Tipos de pruebas de validación

- **Pruebas de seguridad:** Evalúan vulnerabilidades y riesgos.
- **Pruebas de rendimiento:** Analizan la eficiencia del sistema bajo carga.
- **Pruebas de compatibilidad:** Verifican el funcionamiento en diferentes dispositivos y entornos.
- **Pruebas de usabilidad:** Evalúan la experiencia del usuario.

Para pruebas de seguridad, por ejemplo, se pueden usar herramientas como **OWASP ZAP** (Zed Attack Proxy) , que permite analizar vulnerabilidades en aplicaciones web.

Aquí tienes el **manual para empezar a trabajar con ZAP**.

Actividad: Busca herramientas para pruebas de rendimiento, de compatibilidad y usabilidad. Agrega el enlace a la página web principal de la herramienta, al getting-started si lo tiene y a algún videotutorial (en Youtube u otras plataformas) que te parezca interesante, para tener guardado el recurso para cuando te sea útil.

Pruebas exploratorias

Hasta ahora, las pruebas que hemos visto siguen un plan estructurado. Sin embargo, en muchas ocasiones los testers utilizan un enfoque más libre e intuitivo llamado **pruebas exploratorias**.

Las pruebas exploratorias son un tipo de prueba donde el tester **investiga** la aplicación sin seguir un conjunto de pasos predefinidos, buscando errores de manera creativa. Se basan en la experiencia y la intuición del tester. Sus principales características son:

- No requieren casos de prueba predefinidos.
- Permiten descubrir errores inesperados.
- Son útiles cuando no hay suficiente documentación.
- Son complementarias a las pruebas automatizadas.

Supongamos que estamos probando una aplicación de banca en línea. Un tester exploratorio podría intentar:

- Ingresar caracteres extraños en el campo de usuario (!@#\$%^&*).
- Intentar iniciar sesión con un usuario incorrecto varias veces seguidas para ver si bloquea la cuenta.
- Modificar manualmente la URL después de iniciar sesión (www.banco.com/admin).

En el ámbito académico, los profesores de programación son expertos *destruyendo* los programas de los alumnos a base de pruebas exploratorias. Son una forma excelente de simular el comportamiento caótico de miles de clientes interactuando con tu aplicación.

Actividad:

Realiza una sesión de pruebas exploratorias sobre una aplicación web de tu elección. Registra los errores que encuentres y clasifícalos según su gravedad.

Estándares de pruebas de software

Existen dos estándares importantes relacionados con las pruebas de software, el **ISO 25010** y el **ISTQB**.

ISO 25010 - Modelo de Calidad del Software

La norma **ISO/IEC 25010** define un modelo de calidad que evalúa el software en base a **8 características principales**:

- Adecuación funcional:** ¿El software cumple con sus funciones correctamente?
- Eficiencia de rendimiento:** ¿Responde rápido y usa bien los recursos?
- Compatibilidad:** ¿Funciona en diferentes entornos y sistemas?
- Usabilidad:** ¿Es fácil de usar para los usuarios finales?
- Fiabilidad:** ¿Es estable y maneja bien los errores?
- Seguridad:** ¿Protege los datos y evita accesos no autorizados?
- Mantenibilidad:** ¿Es fácil de modificar y mejorar?
- Portabilidad:** ¿Puede ejecutarse en distintos entornos sin problemas?

Las pruebas de software ayudan a garantizar estas características, dependiendo del tipo de prueba que se aplique.

Actividad:

Relaciona cada una de las 8 características de ISO 25010 con los tipos de pruebas de software vistos en clase.

ISTQB - Certificación en Testing de Software

El **ISTQB (International Software Testing Qualifications Board)** es un organismo internacional que define estándares y promueve buenas prácticas en el ámbito de las pruebas de software. Su principal objetivo es certificar a profesionales en el área del testing, ofreciendo una estructura de certificación que cubre distintos niveles de conocimiento y habilidad. A través de esta certificación, ISTQB busca mejorar la calidad del software y la profesionalización de los testers a nivel global.

Niveles de Pruebas

Dentro del marco del ISTQB, se definen **diferentes niveles de pruebas** para garantizar la calidad del software en cada fase del ciclo de vida del desarrollo, que son las que ya conocemos:

- **Pruebas de Unidad**
- **Pruebas de Integración**
- **Pruebas de Sistema**
- **Pruebas de Aceptación**

Tipos de Testing

En el ISTQB, se reconocen varios **tipos de testing**, cada uno con un enfoque particular para evaluar distintas características del software:

Tipo de test	Descripción
Testing Funcional	Se centra en evaluar la funcionalidad del software y comprobar que el sistema haga lo que se espera según los requisitos definidos. Este tipo de pruebas está orientado a verificar la correcta ejecución de las funciones del sistema
Testing No Funcional	Examina características que no están directamente relacionadas con la funcionalidad, como el rendimiento, la seguridad, la accesibilidad o la usabilidad del sistema. Aquí se incluyen las pruebas de carga, de estrés y de seguridad.
Testing Estructural	Se basa en la estructura interna del software, como el código fuente, para asegurar que los componentes estén correctamente implementados. A menudo, se utiliza la técnica de <i>white-box testing</i> (pruebas de caja blanca), en la que se tiene acceso al código y la estructura interna.
Testing de Regresión	Se realizan para asegurar que las nuevas modificaciones o mejoras del software no hayan introducido errores en las funcionalidades previamente implementadas. Este tipo de pruebas son esenciales en cada ciclo de desarrollo y mantenimiento.

Principios del Testing

El ISTQB también establece una serie de **principios fundamentales del testing** que guían las mejores prácticas en la ejecución de pruebas de software:

- **No es posible probar todo:** Dado que no se puede probar exhaustivamente todo el sistema, es importante definir un enfoque de testing basado en riesgos y prioridades, seleccionando los escenarios más relevantes.
- **Las pruebas tempranas ahorrarán costos:** Detectar errores en etapas tempranas del desarrollo reduce significativamente los costos, ya que se evitan fallos mayores que podrían surgir más adelante en el ciclo de vida del software.
- **Las pruebas exhaustivas son imposibles:** En la práctica, nunca es posible probar todos los posibles escenarios. Por lo tanto, el testing debe enfocarse en las áreas más críticas y de mayor impacto.
- **El testing demuestra la presencia de defectos, no la ausencia de ellos:** Las pruebas de software no garantizan que el sistema esté libre de defectos, sino que buscan identificar posibles errores en el software.

Actividad

Contesta a las siguientes preguntas tipo test:

Pregunta 1: ¿Cuál de estas pruebas verifica la funcionalidad sin conocer el código interno?

- A) Caja blanca
- B) Caja negra
- C) Prueba de integración
- D) Prueba de carga

Pregunta 2: ¿Cuál de estas herramientas se usa para pruebas unitarias en Java?

- A) JUnit
- B) Selenium
- C) Jenkins
- D) Postman

Pregunta 3: ¿Qué tipo de prueba se usa en entornos de CI/CD?

- A) Pruebas manuales
- B) Pruebas exploratorias
- C) Pruebas automatizadas
- D) Pruebas alfa

Documentación de Pruebas

La **documentación de pruebas** es el registro organizado y sistemático de todo el proceso de prueba de software. No es solo un trámite, sino una parte fundamental para garantizar la **calidad**, la **trazabilidad** y el **mantenimiento** de cualquier aplicación. Permite que cualquier miembro del equipo, o incluso futuros equipos, comprenda qué se probó, cómo se probó, por qué se probó y cuáles fueron los resultados.

Factores de importancia para la documentación de las pruebas

- **Trazabilidad:** Conecta los requisitos del software con los casos de prueba, asegurando que cada funcionalidad esperada sea validada.
- **Reusabilidad:** Facilita la ejecución repetida de pruebas (pruebas de regresión) a lo largo del ciclo de vida del desarrollo, ahorrando tiempo y recursos.
- **Análisis de Fallos:** Proporciona un registro detallado para depurar errores y entender por qué un defecto se manifestó.
- **Conocimiento Compartido:** Actúa como una base de conocimiento para el equipo, reduciendo la dependencia de individuos y facilitando la incorporación de nuevos miembros.

- **Mejora Continua:** Permite analizar el proceso de prueba, identificar cuellos de botella y optimizar futuras estrategias de testeo.
- **Cumplimiento Normativo:** En muchos sectores (ej. médico, financiero), la documentación de pruebas es un requisito legal o de auditoría.

Elementos clave de la documentación de pruebas

Aunque puede variar según el proyecto, la documentación de pruebas comúnmente incluye:

- **Plan de Pruebas:** Describe el alcance, los objetivos, la estrategia, el cronograma, los recursos y los roles del equipo de pruebas.
- **Casos de Prueba:** Especifican los pasos detallados para ejecutar una prueba, los datos de entrada, los resultados esperados y el criterio de éxito/falla.
- **Guiones/Scripts de Prueba:** Para pruebas automatizadas, es el código y los pasos para ejecutar la automatización.
- **Informes de Defectos (Bugs):** Documentan los errores encontrados, incluyendo pasos para reproducirlos, gravedad, prioridad y estado.
- **Informes de Resumen de Pruebas:** Presentan un resumen ejecutivo de los resultados de la prueba, métricas clave (cobertura, defectos encontrados), riesgos y recomendaciones.

Actividad:

Imagina que trabajas como tester en una empresa que ha desarrollado una aplicación para reservar hoteles. Debes diseñar la documentación de pruebas necesaria, que incluya:

1. **Pruebas unitarias** (ej.: verificar que el cálculo del precio total funcione correctamente).
2. **Pruebas de integración** (ej.: validar que la conexión con la pasarela de pagos funcione bien).
3. **Pruebas de aceptación** (ej.: asegurar que un usuario pueda buscar un hotel y completar una reserva sin problemas).
4. **Pruebas de seguridad** (ej.: evitar que los usuarios puedan modificar los precios desde el frontend).

Debate los diferentes puntos en clase y crea un documento con los casos de prueba para esta aplicación y especifica qué herramientas usarías en cada prueba.

Herramientas para la prueba de software

En este bloque, vamos a repasar diferentes herramientas para la prueba de software. Primero veremos qué son las anotaciones, un recurso que emplean todos los frameworks para funcionar. Después, veremos las herramientas JUnit y Mockito.

- Herramientas para la prueba de software
 - ¿Qué son los **Test de Unidad**?
 - Beneficios de los Test de Unidad
 - Ejemplo sencillo de test de unidad
 - Implementación de Test de Unidad en Java usando IntelliJ y JUnit
 - **Paso 1: Configurar el proyecto en IntelliJ con JUnit**
 - **Paso 2: Escribir un Test de Unidad en Java usando JUnit**
 - **Paso 3: Ejecutar el Test en IntelliJ**
 - Agregando más tests
 - Diseño de Test de unidad
 - Anotaciones en Java
 - Test de unidad con JUNIT
 - **1. Configurar JUnit en IntelliJ**
 - **2. Crear la clase a probar**
 - **3. Escribir pruebas con JUnit**
 - Principales anotaciones de JUnit:

¿Qué son los Test de Unidad?

Los **tests de unidad** son pequeñas pruebas automatizadas que se escriben para verificar que una porción específica de código (una "unidad", generalmente una función o método) funcione correctamente. Su propósito es comprobar que una unidad de código aislada haga lo que se espera que haga sin depender de otras partes del sistema.

Beneficios de los Test de Unidad

1. **Mejora la calidad del código:** Detectan errores en las primeras etapas del desarrollo.
2. **Facilitan el mantenimiento:** Al cambiar el código, los tests ayudan a asegurarse de que no se rompa el comportamiento existente.
3. **Permiten refactorizar con confianza:** Si los tests pasan después de refactorizar, es probable que el código siga funcionando.

Ejemplo sencillo de test de unidad

Supongamos que tenemos una clase **Calculadora** en Java, con un método que suma dos números:

```
public class Calculadora {  
    public static int sumar(Integer a, Integer b) {  
        return a + b;  
    }  
}
```

Queremos escribir un test de unidad para verificar que el método `sumar()` funciona correctamente.

Implementación de Test de Unidad en Java usando IntelliJ y JUnit

Paso 1: Configurar el proyecto en IntelliJ con JUnit

1. **Crear un nuevo proyecto en IntelliJ** utilizando Maven.
2. **Agregar dependencia de JUnit (si no está configurado automáticamente):**

Asegúrate de que el archivo `pom.xml` tenga la siguiente dependencia para JUnit:

```
<dependencies>  
    <dependency>  
        <groupId>org.junit.jupiter</groupId>  
        <artifactId>junit-jupiter-api</artifactId>  
        <version>5.7.0</version>  
        <scope>test</scope>  
    </dependency>  
    <dependency>  
        <groupId>org.junit.jupiter</groupId>  
        <artifactId>junit-jupiter-engine</artifactId>  
        <version>5.7.0</version>  
        <scope>test</scope>  
    </dependency>  
</dependencies>
```

3. **Crear la clase de test:**

- En el árbol de directorios de IntelliJ, crea un nuevo paquete llamado `test` (`src/test/java`).
- Crea una nueva clase de prueba (ej. `CalculadoraTest`) en ese paquete.

Paso 2: Escribir un Test de Unidad en Java usando JUnit

Vamos a escribir un test para la clase `Calculadora` :

1. Crea la clase `CalculadoraTest` y usa las anotaciones de **JUnit** para definir tus pruebas:

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertEquals;

public class CalculadoraTest {

    @Test
    public void testSumar() {
        int resultado = Calculadora.sumar(2, 3);
        assertEquals(5, resultado, "La suma de 2 y 3 debe ser 5");
    }
}
```

- **@Test**: Esta anotación indica que el método `testSumar` es una prueba que se debe ejecutar.
- **assertEquals(esperado, actual)**: Verifica si el valor que devuelve el método (en este caso `5`) es igual al valor esperado (también `5`). Si no coincide, el test fallará.

Paso 3: Ejecutar el Test en IntelliJ

1. Haz clic derecho sobre la clase de prueba `CalculadoraTest` y selecciona "Run 'CalculadoraTest'".
2. Si todo está bien, IntelliJ te mostrará que el test ha pasado exitosamente.

Agregando más tests

Para asegurarnos de que la clase **Calculadora** se comporta correctamente en diferentes escenarios, podemos agregar más tests. Por ejemplo, un test para sumar números negativos:

```
@Test
public void testSumarNumerosNegativos() {
    int resultado = Calculadora.sumar(-1, -5);
    assertEquals(-6, resultado, "La suma de -1 y -5 debe ser -6");
}
```

Cada vez que agregues un test, simplemente ejecútalo de nuevo para verificar que todos los casos pasen.

Diseño de Test de unidad

Cuando creamos los test de unidad, debemos diseñarlos para comprobar valores tanto acertados, como errados o fuera de ámbito. Es tan importante comprobar que hace bien una operación como que el programa sabe enfrentarse a entradas no esperadas. En general, cuando diseñamos un test, creamos una tabla como la que sigue:

Identificador	Nombre	Entrada de datos	Resultado esperado	Resultado obtenido
U1	testSumarNumerosNegativos	Calculadora.sumar(-1, -5)	-6	?
U2	testSumarConStrings	Calculadora.sumar(-1, "paco")	error	?

Una buena estrategia es:

- Buscar casos comunes.
- Si procede, buscar casos en el límite de operaciones.
- Comprobar cómo reacciona un método ante entradas inesperadas.

Actividad

Números primos

Diseña test de unidad para el siguiente método, tanto en una tabla como en JUNIT.

```
public class Calculadora {  
    public static boolean esPrimo(int numero) {  
        if (numero <= 1) return false;  
        for (int i = 2; i * i <= numero; i++) {  
            if (numero % i == 0) return false;  
        }  
        return true;  
    }  
}
```

Actividad

Sumador de palabras

Diseña test de unidad para el siguiente método, tanto en una tabla como en JUNIT. El método coge dos palabras y suma el valor de todos los caracteres. El valor de un carácter es su número en

```

public class Calculadora {
    public static int sumaPalabras(String palabra1, String palabra2) {
        int suma = 0;
        for (char c : palabra1.toCharArray()) {
            suma += (int) c;
        }
        for (char c : palabra2.toCharArray()) {
            suma += (int) c;
        }
        return suma;
    }
}

```

Anotaciones en Java

Para poder funcionar correctamente, los diferentes frameworks de Java y otros lenguajes de programación emplean anotaciones. Las **anotaciones** en Java son una forma especial de **metadatos** que se pueden agregar al código para proporcionar información adicional a los compiladores, herramientas o frameworks. Se identifican con el símbolo @ seguido del nombre de la anotación.

```

@Override
public String toString() {
    return "Ejemplo de anotación";
}

```

En este caso, `@Override` indica que el método `toString()` sobrescribe un método de la superclase.

Las anotaciones **no cambian la lógica del código**, pero **permiten que herramientas externas lo procesen de manera especial**. Las anotaciones tienen varios propósitos, como:

- **Reducir código repetitivo**: permiten automatizar configuraciones.
- **Asegurar buenas prácticas**: por ejemplo, `@Override` previene errores en la herencia.
- **Facilitar pruebas y frameworks**: como veremos en JUnit, Mockito y, posteriormente, Cucumber.

Test de unidad con JUNIT

JUnit es el framework más utilizado para realizar **pruebas unitarias en Java**. En este tutorial, aprenderás a configurarlo y a escribir pruebas paso a paso.

Actividad

Realiza una memoria con capturas de pantalla de este tutorial

1. Configurar JUnit en IntelliJ

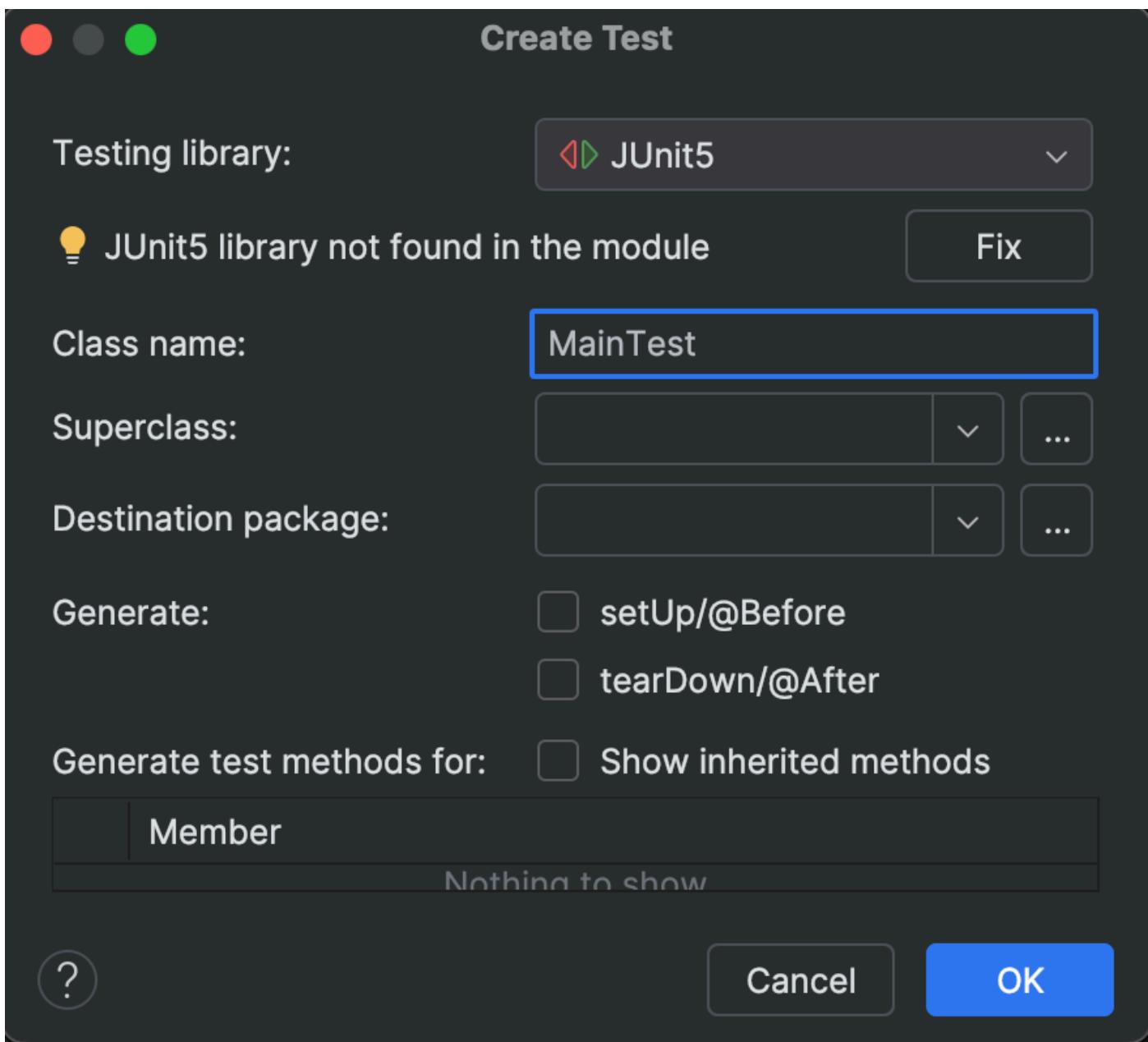
1. **Abre IntelliJ IDEA.**
2. **Crea un nuevo proyecto** con soporte para **Java**. Usa Maven (aunque también es compatible con Gradle)
3. **Agrega las dependencias de JUnit** en tu archivo de configuración.

**Edita el archivo `pom.xml` y agrega dentro de la etiqueta `<dependencies></dependencies>` :

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter</artifactId>
    <version>LATEST</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter</artifactId>
    <version>LATEST</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter</artifactId>
    <version>LATEST</version>
    <scope>test</scope>
</dependency>
```

Si no estuviera creada la etiqueta `dependencies`, añádela. También puedes dejar que sea el propio editor el que añada la dependencia de jUnit de la siguiente forma:

- Crea un archivo de código fuente cualquiera (por ejemplo, `Main.java`) en la carpeta `src/main/java`.
- Haz click derecho en el nombre de la clase y elige la opción **Generate**. A continuación, elige la opción **Test**.



- En esta pantalla te avisará de que JUnit5 no se encuentra en el módulo. Haz click en Fix. Se quedará marcado. Entonces haz click en OK.
- Revisa el archivo `pom.xml` para comprobar que se han añadido las dependencias correctamente (cosa que habrá pasado con un 99% de probabilidades).

Dentro de la etiqueta `<version></version>` puedes colocar el número de la versión que quieras usar o emplear alguna palabra clave, como `LATEST` que en este caso descargará la versión más reciente de la dependencia. En un entorno experimental, esto es útil. Sin embargo, para una producción real sería mejor escoger una versión y no cambiarla. En este caso, al tratarse de unos apuntes, nos quedamos con la versión `LATEST`.

2. Crear la clase a probar

Vamos a probar una **clase simple de calculadora**. Crea un nuevo archivo `Calculadora.java` y añade el siguiente código:

```
public class Calculadora {  
    public int sumar(int a, int b) {  
        return a + b;  
    }  
  
    public int restar(int a, int b) {  
        return a - b;  
    }  
  
    public int multiplicar(int a, int b) {  
        return a * b;  
    }  
  
    public int dividir(int a, int b) {  
        if (b == 0) {  
            throw new ArithmeticException("No se puede dividir por cero");  
        }  
        return a / b;  
    }  
}
```

3. Escribir pruebas con JUnit

1. **Crea un nuevo archivo de prueba** en `src/test/java` o haz click derecho en el nombre de la clase, selecciona Generate y luego Test.
2. **Nombra la clase** como `CalculadoraTest.java`. Si usas la opción de Generate, puedes decirle a IntelliJ que además te genere las cabeceras necesarias para testear todos los métodos, así como las opciones de **setup** y **teardown**. La opción de setup `@BeforeEach` ejecuta dicho código antes de cada prueba y la función `@AfterEach` al acabar cada prueba. En este caso, emplearemos solo la de `setup`, pero la de `teardown` puede ser interesante trabajando con propiedades estáticas.
3. **Escribe las pruebas unitarias**. Puedes copiar el siguiente código:

```

public class CalculadoraTest {
    private Calculadora calculadora;

    @BeforeEach
    void setUp() {
        calculadora = new Calculadora();
    }

    @Test
    void testSuma() {
        int resultado = calculadora.sumar(5, 3);
        assertEquals(8, resultado, "La suma de 5 + 3 debe ser 8");
    }

    @Test
    void testResta() {
        int resultado = calculadora.restar(10, 4);
        assertEquals(6, resultado, "La resta de 10 - 4 debe ser 6");
    }

    @Test
    void testMultiplicacion() {
        int resultado = calculadora.multiplicar(3, 7);
        assertEquals(21, resultado, "La multiplicación de 3 x 7 debe ser 21");
    }

    @Test
    void testDivision() {
        int resultado = calculadora.dividir(10, 2);
        assertEquals(5, resultado, "La división de 10 / 2 debe ser 5");
    }

    @Test
    void testDivisionPorCero() {
        Exception exception = assertThrows(ArithmeticException.class, () -> calculadora.di
        assertEquals("No se puede dividir por cero", exception.getMessage());
    }
}

```

Al copiarlo, verás que te aparecen partes en rojo. Esto es porque no se han importado los módulos. Para resolver ese problema, haz click en cada uno de los elementos en rojo, dale a show context actions y dale a importar la clase (esta opción también aparece mantener estático el

ratón encima del elemento en rojo). Repite el proceso hasta que todas las importaciones estén correctas.

El código anterior se compone de los siguientes elementos:

- **@BeforeEach** : Se ejecuta antes de cada prueba para inicializar la calculadora.
- **@Test** : Marca un método como una prueba de JUnit.
- **assertEquals(valor Esperado, valor Real, mensaje)** : Verifica si el resultado es correcto.
- **assertThrows(ArithmetiException.class, () -> calculadora.dividir(10, 0))** : Verifica que la excepción se lance correctamente.

Para ejecutar las pruebas, realiza las siguientes acciones:

1. Haz clic derecho en `CalculadoraTest.java` y selecciona **Run 'CalculadoraTest'**.
2. Verifica que **todas las pruebas pasen** con una marca verde.
3. Si alguna falla, revisa el mensaje de error y corrige la implementación.

En este caso, no debería fallar ningún test. Prueba qué ocurre cuando falla cambiando algún `valor Esperado`.

Para las pruebas unitarias, el método `assertEquals(_valorEsperado, _valorObtenido)` es suficiente para la mayoría de situaciones, pero se dispone también de especializaciones como `assertFalse`, etc. El propio IntelliJ te propondrá esas opciones cuando sea pertinente.

Principales anotaciones de JUnit:

Anotación	Descripción
<code>@Test</code>	Indica que el método es una prueba.
<code>@BeforeEach</code>	Se ejecuta antes de cada prueba.
<code>@AfterEach</code>	Se ejecuta después de cada prueba.
<code>@BeforeAll</code>	Se ejecuta una vez antes de todas las pruebas.
<code>@AfterAll</code>	Se ejecuta una vez después de todas las pruebas.

En otros lenguajes de programación, encontramos las siguientes librerías para test de unidad similares a JUnit:

- **PHPUnit** para PHP.
- **xUnit o NUnit** para C#.
- **Jest o Mocha** en JavaScript).

- **unittest** o **pytest** para python.

Pruebas de integración

- Pruebas de integración
 - Dobles de prueba con Mockito
 - Principales anotaciones de Mockito:

Dobles de prueba con Mockito

Mockito es una de las bibliotecas más utilizadas para la creación de pruebas unitarias en Java. Permite simular objetos y comportamientos para probar métodos sin depender de implementaciones reales. Vamos a continuar el proyecto anterior y añadirle soporte para Mockito.

Actividad

Realiza una memoria de todo el proceso de configuración de Mockito

En el archivo `pom.xml`, añade esto en `:

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>LATEST</version>
</dependency>
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-junit-jupiter</artifactId>
    <version>LATEST</version>
    <scope>test</scope>
</dependency>
```

A continuación, debemos **refrescar** el proyecto para descargar las dependencias. Haz click derecho en el archivo `pom.xml` y ve a Maven->Sync Project.

Supongamos que tenemos una clase `CalculadoraService` que depende de un `Repositorio` para obtener datos. Añadamos la clase a nuestro proyecto en un archivo `CalculadoraService` dentro de `src/main/java`:

```

public class CalculadoraService {
    private final Repositorio repositorio;

    public CalculadoraService(Repositorio repositorio) {
        this.repositorio = repositorio;
    }

    public int sumarValores() {
        return repositorio.obtenerValorA() + repositorio.obtenerValorB();
    }
    //Lo usaremos más tarde en las opciones avanzadas de mockito
    public int sumarValores(int a, int b){
        return a + b;
    }
}

```

El repositorio nos va a salir en rojo, así que creamos una interfaz para representar los métodos que necesitaremos de él en un archivo `Repositorio.java` :

```

public interface Repositorio {
    int obtenerValorA();
    int obtenerValorB();
}

```

Al haberlo creado en el mismo paquete, no es necesario añadir ninguna línea de importación. Si lo hubiéramos creado en otro paquete distinto, deberíamos importar las clases de forma adecuada, empleando las herramientas que nos proporciona IntelliJ, de forma similar a como hicimos al importar la etiqueta `@Test` en el apartado anterior.

Lo siguiente es crear un test usando Mockito. Para ello, generamos los test de `CalculadoraService` igual que en el apartado anterior.

1. En `src/test/java` , crea una clase de prueba:

```

@ExtendWith(MockitoExtension.class)
public class CalculadoraServiceTest {

    @Mock
    private Repositorio repositorio;

    //Esto inserta el Mock anterior como parámetro en el constructor de CalculadoraServ
    @InjectMocks
    private CalculadoraService calculadoraService;

    @Test
    void testSumarValores() {
        when(repositorio.obtenerValorA()).thenReturn(5);
        when(repositorio.obtenerValorB()).thenReturn(3);

        int resultado = calculadoraService.sumarValores();

        assertEquals(8, resultado);
    }
}

```

Volverán a aparecer elementos sin importar. Impórtalos todos haciendo uso de la herramienta de IntelliJ para tal efecto.

- `@Mock` simula un objeto `Repositorio` .
- `@InjectMocks` inyecta el mock en `CalculadoraService` .
- `when(...).thenReturn(...)` define valores falsos que devolverá el mock.
- `assertEquals(8, resultado)` verifica el resultado esperado.

Ejecutar la prueba

- Haz clic derecho en la prueba y selecciona **Run 'CalculadoraServiceTest'**.
- Verifica que la prueba pase correctamente.

Cuando ejecutes el código, funcionará bien pero te saldrá un Warning:

```
Mockito is currently self-attaching to enable the inline-mock-maker. This will no longer
WARNING: A Java agent has been loaded dynamically (/Users/arturoalbero/.m2/repository/n
WARNING: If a serviceability tool is in use, please run with -XX:+EnableDynamicAgentLoa
WARNING: If a serviceability tool is not in use, please run with -Djdk.instrument.trace
WARNING: Dynamic loading of agents will be disallowed by default in a future release
OpenJDK 64-Bit Server VM warning: Sharing is only supported for boot loader classes bec
```

Esto sucede porque Mockito carga un agente de forma dinámica, comportamiento que se pretende restringir en futuras versiones de Java. En este caso, se está usando la versión 21. Para solucionar los WARNING (o futuros errores), debes añadir después de `</dependencies>` el siguiente apartado en tu archivo `pom.xml`:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>3.0.0-M7</version>
      <configuration>
        <argLine>-javaagent:"${settings.localRepository}/net/bytebuddy/byte-bud
      </configuration>
    </plugin>
  </plugins>
</build>
```

Lo que estamos haciendo es añadir Mockito como un agente a nuestro proyecto Maven, tal y como nos decía el *warning*. Refresca el proyecto y ejecuta de nuevo. Te volverá a salir un warning (la última línea), pero de menor importancia:

```
OpenJDK 64-Bit Server VM warning: Sharing is only supported for boot loader classes bec
```

Recuerda que cada vez que hagas un cambio en el archivo `pom.xml`, debes sincronizar el proyecto para que haga efecto.

A continuación, vamos a retocar el código anterior añadiendo todas las funcionalidades de Mockito:

- **Verificación de llamadas** (`verify`)
- **Manejo de excepciones** (`doThrow`)
- **Retornos múltiples** (`thenReturn`)

- **Respuesta personalizada (thenAnswer)**
- **Spies para llamar a métodos reales de la clase si es necesario (@Spy)**

```
@ExtendWith(MockitoExtension.class)
public class CalculadoraServiceTest {

    @Mock
    private Repositorio repositorio;

    //Inyectamos el mock en calculadoraService
    @InjectMocks
    private CalculadoraService calculadoraService;

    //Inyectamos el mock en el espía calculadoraSpy
    @InjectMocks
    @Spy
    private CalculadoraService calculadoraSpy;

    @Test
    void testSumarValores() {
        when(repositorio.obtenerValorA()).thenReturn(5);
        when(repositorio.obtenerValorB()).thenReturn(3);

        int resultado = calculadoraService.sumarValores();

        assertEquals(8, resultado);
        verify(repositorio).obtenerValorA();
        verify(repositorio).obtenerValorB();
    }

    //En este test vamos a probar la obtención de diferentes valores consecutivos.
    @Test
    void testSumarValoresConMultiplesRetornos() {
        when(repositorio.obtenerValorA()).thenReturn(5, 10);
        when(repositorio.obtenerValorB()).thenReturn(3);

        assertEquals(8, calculadoraService.sumarValores());
        assertEquals(13, calculadoraService.sumarValores());
    }

    @Test
    void testSumarValoresConExcepcion() {
        when(repositorio.obtenerValorA()).thenThrow(new RuntimeException("Error al obte

        Exception exception = assertThrows(RuntimeException.class, () -> {
            calculadoraService.sumarValores();
        });
    }
}
```

```

    });

    assertEquals("Error al obtener valor A", exception.getMessage());
}

//En este test verificamos que las operaciones del repositorio se realizan en el orden correcto
@Test
void testVerificacionDeOrden() {
    when(repository.obtenerValorA()).thenReturn(5);
    when(repository.obtenerValorB()).thenReturn(3);

    calculadoraService.sumarValores();

    InOrder inOrder = inOrder(repository);
    inOrder.verify(repository).obtenerValorA();
    inOrder.verify(repository).obtenerValorB();
}

//En este test, hacemos que el espía mantenga funcionalidades del objeto, y mockeamos las indicadas.
@Test
void testUsoDeSpy() {
    doReturn(15).when(calculadoraSpy).sumarValores();

    int resultado = calculadoraSpy.sumarValores();
    assertEquals(15, resultado);
    assertEquals(32, calculadoraService.sumarValores(17, 15));
}
}
}

```

Como siempre, debemos importar las dependencias que faltan para que funcione.

Principales anotaciones de Mockito:

Anotación	Descripción
@Mock	Crea un objeto falso (mock).
@InjectMocks	Inyecta automáticamente los mocks en la clase bajo prueba.
@ExtendWith(MockitoExtension.class)	Habilita Mockito en JUnit 5.

Anotación	Descripción
@Spy	Crea un espía que permite llamar métodos reales y simular comportamientos.
@Captor	Crea un ArgumentCaptor para capturar valores pasados a métodos mockeados.

Actividad: Pruebas en una aplicación bancaria

1. **Configura un nuevo proyecto en IntelliJ IDEA** con soporte para Java y Maven.
2. **Agrega las dependencias necesarias** en el archivo `pom.xml` para JUnit y Mockito.
3. **Crea una clase Banco** con los siguientes métodos:
 - `depositar(String cuenta, double monto)` : Aumenta el saldo de una cuenta.
 - `retirar(String cuenta, double monto)` : Disminuye el saldo de una cuenta si hay fondos suficientes, de lo contrario, lanza una excepción.
 - `consultarSaldo(String cuenta)` : Devuelve el saldo actual de la cuenta.
4. **Define una interfaz RepositorioBanco** que represente el acceso a los datos del banco con los métodos:
 - `obtenerSaldo(String cuenta)` : Devuelve el saldo de una cuenta.
 - `actualizarSaldo(String cuenta, double nuevoSaldo)` : Modifica el saldo de una cuenta.
5. **Implementa BancoService** que use `RepositorioBanco` para interactuar con los datos.
6. **Escribe pruebas unitarias con JUnit** para validar el comportamiento de `Banco`.
7. **Usa Mockito para simular RepositorioBanco** en las pruebas de `BancoService`.
8. **Amplía las pruebas** aplicando:
 - **Verificación de llamadas** (`verify`)
 - **Manejo de excepciones** (`doThrow`)
 - **Retornos múltiples** (`thenReturn`)
 - **Orden de ejecución** (`InOrder`)
 - **Spies para pruebas combinadas** (`@Spy`)

Actividad

Verificación de Interacciones Simples con Mockito

Completa el archivo de prueba `GestorPedidosTest.java` rellenando las líneas comentadas. Tu objetivo es:

1. Declarar e inicializar un mock de `ServicioNotificaciones` .
2. Inyectar este mock en una instancia de `GestorPedidos` .
3. Invocar el método `procesarPedido` en la clase bajo prueba.
4. Verificar que el método `enviarNotificacion` del mock fue llamado exactamente una vez, con los argumentos correctos (`clienteEmail` y el mensaje esperado).

Asegúrate de que tu test compile y pase correctamente.

A continuación, se proporcionan los archivos Java necesarios:

ServicioNotificaciones.java

```
package com.ejercicio.mockito;

public class ServicioNotificaciones {
    public void enviarNotificacion(String destinatario, String mensaje) {
        System.out.println("Enviando notificación a " + destinatario + ":" + mensaje);
    }
}
```

GestorPedidos.java

```
package com.ejercicio.mockito;

public class GestorPedidos {
    private ServicioNotificaciones servicioNotificaciones;

    public GestorPedidos(ServicioNotificaciones servicioNotificaciones) {
        this.servicioNotificaciones = servicioNotificaciones;
    }

    public void procesarPedido(String idPedido, String clienteEmail) {
        System.out.println("Procesando pedido: " + idPedido + " para " + clienteEmail);
        servicioNotificaciones.enviarNotificacion(clienteEmail, "Su pedido " + idPedido)
    }
}
```

GestorPedidosTest.java

```

package com.ejercicio.mockito;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import static org.mockito.Mockito.*;

public class GestorPedidosTest {

    // 1. Declara una variable para el mock de ServicioNotificaciones
    // private ServicioNotificaciones mockServicioNotificaciones;

    // 2. Declara una variable para la instancia de GestorPedidos
    // private GestorPedidos gestorPedidos;

    @BeforeEach
    void setUp() {
        // 3. Inicializa el mock antes de cada test
        // mockServicioNotificaciones = mock(ServicioNotificaciones.class);

        // 4. Inicializa GestorPedidos inyectando el mock
        // gestorPedidos = new GestorPedidos(mockServicioNotificaciones);
    }

    @Test
    void testProcesarPedidoEnvioNotificacion() {
        String idPedido = "PED001";
        String clienteEmail = "cliente@example.com";
        String mensajeEsperado = "Su pedido PED001 ha sido procesado.";

        // 5. Llama al método que quieras probar en GestorPedidos
        // gestorPedidos.procesarPedido(idPedido, clienteEmail);

        // 6. Verifica que el método 'enviarNotificacion' del mock fue llamado
        //     una vez, con los argumentos 'clienteEmail' y 'mensajeEsperado'.
        // verify(mockServicioNotificaciones, times(1)).enviarNotificacion(eq(clienteEm
    }
}

```

Pruebas de aceptación

- Pruebas de aceptación
 - Uso de Cucumber con Maven y Java para las pruebas de aceptación
 - 1. Crear el Proyecto en IntelliJ con Maven
 - 2. Agregar Dependencias de Cucumber
 - 3. Escribir Escenarios de Prueba en Gherkin
 - 4. Implementar la Clase `Calculadora.java`
 - 5. Implementar los Step Definitions
 - 6. Configurar el Runner de Pruebas
 - 7. Ejecutar las Pruebas con Maven
 - Principales anotaciones de Cucumber

Uso de Cucumber con Maven y Java para las pruebas de aceptación

En este documento se describen los pasos para utilizar **Cucumber** en **IntelliJ IDEA** con **Maven**, que nos ayudará a desarrollar una aplicación siguiendo el enfoque **TDD (Desarrollo guiado por Pruebas)**. Cucumber permite escribir pruebas en lenguaje natural **Gherkin** y ejecutarlas en Java.

El **Desarrollo guiado por pruebas de software**, o Test-driven development (TDD) es una práctica de ingeniería de software que involucra otras dos prácticas: Escribir las pruebas primero (Test First Development) y Refactorización (Refactoring). Para escribir las pruebas generalmente se utilizan las pruebas unitarias (unit test en inglés). En primer lugar, se escribe una prueba y se verifica que la nueva prueba falla. A continuación, se implementa el código que hace que la prueba pase satisfactoriamente y seguidamente se refactoriza el código escrito. El propósito del desarrollo guiado por pruebas es lograr un código limpio que funcione. La idea es que los requisitos sean traducidos a pruebas, de este modo, cuando las pruebas pasen se garantizará que el software cumple con los requisitos que se han establecido. El desarrollo guiado por pruebas es una metodología ágil, **centrada en el software y en el desarrollador**.

Con Gherkin, transformamos el desarrollo guiado por pruebas en desarrollo guiado por comportamiento, gracias al uso del lenguaje natural. Siguiendo este método, primero escribiremos las pruebas en **lenguaje natural (Gherkin)**, luego implementaremos la lógica en Java para hacer que las pruebas pasen.

Actividad

Registra los pasos de este tutorial en una memoria.

1. Crear el Proyecto en IntelliJ con Maven

1. Abre **IntelliJ IDEA** y selecciona **New Project**.
2. Elige **Maven** como gestor de dependencias.
3. En **GroupId**, escribe: `com.ejemplo`.
4. En **ArtifactId**, escribe: `calculadora-cucumber`.
5. Haz clic en **Finish**.

El proyecto tendrá esta estructura inicial:

```
calculadora-cucumber
|__ src
|   |__ main
|   |   |__ java
|   |   |   |__ com.ejemplo
|   |   |   |   |__ Calculadora.java
|   |__ test
|   |   |__ java
|   |   |   |__ com.ejemplo
|   |   |   |   |__ RunCucumberTest.java
|   |   |   |   |__ StepDefinitions.java
|   |   |__ resources
|   |   |   |__ features
|   |   |   |   |__ calculadora.feature
|__ pom.xml
|__ .gitignore
```

2. Agregar Dependencias de Cucumber

Edita el archivo `pom.xml` y agrega las siguientes dependencias para **Cucumber y JUnit**:

```

<dependencies>
    <!-- Cucumber Core -->
    <dependency>
        <groupId>io.cucumber</groupId>
        <artifactId>cucumber-java</artifactId>
        <version>7.14.0</version>
    </dependency>

    <!-- Cucumber JUnit -->
    <dependency>
        <groupId>io.cucumber</groupId>
        <artifactId>cucumber-junit</artifactId>
        <version>7.14.0</version>
        <scope>test</scope>
    </dependency>

    <!-- JUnit 5 -->
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter</artifactId>
        <version>5.9.2</version>
        <scope>test</scope>
    </dependency>

    <!-- Plugin para ejecutar pruebas -->
    <dependency>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <version>3.0.0-M7</version>
    </dependency>
</dependencies>

```

Ejecuta en la terminal de IntelliJ:

```
mvn clean install
```

para descargar las dependencias.

3. Escribir Escenarios de Prueba en Gherkin

Creamos el archivo `src/test/resources/features/calculadora.feature` con los escenarios en **Gherkin**:

Feature: Calculadora

Como usuario, quiero realizar operaciones matemáticas básicas para obtener resultados correctos.

Scenario: Sumar dos números

`Given` que tengo una calculadora

`When` sumo 2 y 3

`Then` el resultado debe ser 5

Scenario: Restar dos números

`Given` que tengo una calculadora

`When` resto 5 y 2

`Then` el resultado debe ser 3

- `Given` establece el estado inicial.
- `When` define la acción.
- `Then` verifica el resultado esperado.

4. Implementar la Clase `Calculadora.java`

Creamos la clase en `src/main/java/com/ejemplo/Calculadora.java` :

```
package com.ejemplo;

public class Calculadora {
    public int sumar(int a, int b) {
        return a + b;
    }

    public int restar(int a, int b) {
        return a - b;
    }
}
```

5. Implementar los Step Definitions

Creamos la clase StepDefinitions.java en
src/test/java/com/ejemplo/StepDefinitions.java :

```
package com.ejemplo;

import io.cucumber.java.en.Given;
import io.cucumber.java.en.When;
import io.cucumber.java.en.Then;
import static org.junit.jupiter.api.Assertions.*;

public class StepDefinitions {
    private Calculadora calculadora;
    private int resultado;

    @Given("que tengo una calculadora")
    public void queTengoUnaCalculadora() {
        calculadora = new Calculadora();
    }

    @When("sumo {int} y {int}")
    public void sumo(int a, int b) {
        resultado = calculadora.sumar(a, b);
    }

    @When("resto {int} y {int}")
    public void resto(int a, int b) {
        resultado = calculadora.restar(a, b);
    }

    @Then("el resultado debe ser {int}")
    public void elResultadoDebeSer(int esperado) {
        assertEquals(esperado, resultado);
    }
}
```

6. Configurar el Runner de Pruebas

Creamos la clase RunCucumberTest.java en
src/test/java/com/ejemplo/RunCucumberTest.java :

```
package com.ejemplo;

import org.junit.platform.suite.api.*;

@Suite
@IncludeEngines("cucumber")
@SelectClasspathResource("features")
@ConfigurationParameter(key = "cucumber.glue", value = "com.ejemplo")
public class RunCucumberTest {
}
```

7. Ejecutar las Pruebas con Maven

Para ejecutar las pruebas, usa el siguiente comando en la terminal:

```
mvn test
```

Si todo está configurado correctamente, deberías ver un resultado similar a este:

```
[INFO] Running com.ejemplo.RunCucumberTest
Feature: Calculadora

  Scenario: Sumar dos números
    Given que tengo una calculadora
    When sumo 2 y 3
    Then el resultado debe ser 5

  Scenario: Restar dos números
    Given que tengo una calculadora
    When resto 5 y 2
    Then el resultado debe ser 3

[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
```

Esto indica que las pruebas han pasado correctamente.

Actividad

Realiza el siguiente tutorial oficial y haz una memoria

- [Tutorial de 10 minutos -se tarda más-](#)

Principales anotaciones de Cucumber

Anotación	Descripción
@Given	Define el contexto inicial.
@When	Describe la acción que ocurre.
@Then	Verifica el resultado esperado.

Reto cooperativo de la Unidad de Programación 05

Herramientas para el desarrollo de software

Formación del equipo

Para esta unidad de programación, la formación de los equipos se hará a través del método *Dinámica de los colores*. Observa la tabla:

Egun ona / Buen día	Egun txarra / Mal día	Egun ona / Buen día	Egun txarra / Mal día
Zehatza Precisa Sistematikoa <i>Sistemática</i> Jakingura duena <i>Curiosa, preguntóna</i> Analitikoa <i>Análitica</i> Zentzuduna <i>Sensata</i> Saiatua Perseverante Metodikoa <i>Metódica</i> Neurtua Controladora Disziplinatua <i>Disciplinada</i> Egonkorra Estable	Ikuspegia itxikoa <i>Estirada, cerrada</i> Zekena Mezquina Erretxina Quisquilloso Aldaezina Inamovible Disoziatua Disociada Hotza Fria Fidekaitza Suspicaz Kritikoa <i>Critica</i> Diplomazia gutxikoa <i>Poco diplomática</i> Hunkibera Susceptible	Ekintzailea <i>Emprendedora</i> Objektiboa <i>Objetiva</i> Tinkoa <i>Decidida</i> Zorrotza Exigente Saiatua Tenaz Helburuetara begira <i>Orientada a objetivos</i> Aktiboa, dinamikoa <i>Enérgica</i> Ordenatua <i>Organizada</i> Erabakitzalea <i>Resolutiva</i> Lehiakorra <i>Competitiva</i>	Mendertzalea <i>Dominante</i> Oldarkorra <i>Agresiva</i> Intolerantea <i>Intolerante</i> Harroa <i>Soberbia</i> Pazientzia gutxikoa <i>Impaciente</i> Begirunerik gabekoa <i>Desconsiderada</i> Zakarra <i>Grosera</i> Eskrupulurik gabea <i>Sin escrúpulos</i> Sasijakintsua <i>"Sabelotodo"</i> Kontrolatzalea <i>Controladora</i>
Egun ona / Buen día	Egun txarra / Mal día	Egun ona / Buen día	Egun txarra / Mal día
Fidagarria <i>Fiable</i> Adeitsua <i>Atenta</i> Elkarbanatzen du <i>Que comparte</i> Leiala <i>Leal</i> Arettatsua <i>Diligente</i> Pazientziaduna <i>Paciente</i> Ulerkorra <i>Comprensiva</i> Lasaila <i>Tranquila</i> Pentsakorra <i>Considerada</i> Diskretoa <i>Discreta</i>	Burugogorra <i>Obstinada</i> Ernaia <i>Cautelosa</i> Zalantzatia <i>Dubitativa</i> Aitzakiz bete <i>Evasiva</i> Adosteko gal ez dena <i>Inconciliable</i> Barnekoia <i>Retraída</i> Uzkurra <i>Reacia</i> Etsigarria <i>Desesperada</i> Sentibera <i>Sensible</i> Isila <i>Reservada</i>	Sinesgarria <i>Convincente</i> Hitz errazekoa <i>Extrovertida</i> Gogotsua <i>Entusiasta</i> Baikorra <i>Optimista</i> Lagunkoia <i>Sociable</i> Dinamikoa <i>Dinámica</i> Irekia <i>Comunicativa</i> Sortzailea <i>Creativa</i> Bat-batekoia <i>Espontánea</i> Burujabea <i>Independiente</i>	Oldarkorra <i>Impulsiva</i> Oso urduria <i>Sobexcitada</i> Frenetikoa, gogorra <i>Agitada</i> Gehiegizkoa <i>Exagerada</i> Diskretoia gutxikoa <i>Indiscreta</i> Nabarmena <i>Extravagante</i> Zaratatsua, ozena <i>Llamativa, ruidosa</i> Azalekoia <i>Superficial</i> Axolagabea <i>Descuidada</i> Ordenik gabea <i>Desorganizada</i>

¿Con qué colores te identificas más? Escoge en la encuesta en FORMS tu primera y tu segunda opción. En base a ella, se crearán los equipos de la clase intentando tener representados todos los colores.

Tarea

Continúa el proyecto que se te ha asignado. Este proyecto fue comenzado por otros equipos, que se encargaron del **Diseño de los diagramas de comportamiento**, del **Diseño de los diagramas de clases y propuesta de traducción a entidad relación** y de la preparación para la **implementación**.

Tu labor es:

- Corregir posibles errores del producto entregado por el equipo anterior entendiendo a la especificación dada. Trabajaremos sobre el proyecto de Java

- Diseñar un documento de planificación de pruebas adecuado al proyecto
- Crear las pruebas de unidad usando JUNIT
- Crear las pruebas de integración usando MOCKITO
- Crear las pruebas de aceptación usando CUCUMBER
- Documentar posibles incidencias en las pruebas

Reto individual

Ejercicio 1: Pruebas de software

Actividad

Diseña un plan de pruebas para el programa que diseñaste en el reto de la unidad 4, acorde al diagrama de la unidad 3, para ello:

- Diseña el plan de pruebas general
- Diseña los casos de uso. Implementa JUNIT
- Diseña pruebas de integración. Implementa Mockito
- Diseña pruebas de aceptación. Implementa Cucumber

Ejercicio 2: Uso del debugger

Dado el siguiente código y usando el Debugger de IntelliJ:

```

public class EjercicioDebug {
    public static void main(String[] args) {
        int[] n = {3, 7, 2, 9, 5};
        int r = 0;
        for (int i = 0; i < n.length; i++) {
            r += procesaNumero(n[i]); // Colocar breakpoint con condición: n[i] > 5
        }
        System.out.println("Resultado: " + r);
        muestraInfoExtra(n);
    }

    static int procesaNumero(int x) {
        int y = 0;
        if (x % 2 == 0) {
            y = x * 2;
        } else {
            y = x * 3;
        }
        System.out.println("Procesando: " + x);
        System.out.println("Temporal: " + y);
        System.out.println("-----");
        return y;
    }

    static void muestraInfoExtra(int[] arr) {
        int s = 0;
        for (int i = 0; i < arr.length; i++) {
            s += arr[i];
        }
        for (int i = 0; i < arr.length; i++) {
            if (arr[i] % 2 == 0) {
                System.out.println(arr[i] + " es par");
            } else {
                System.out.println(arr[i] + " es impar");
            }
        }
        System.out.println("Suma total: " + s);
    }
}

```

Coloca un punto de ruptura en la línea indicada con la condición establecida y realiza las siguientes pruebas:

- En el primer punto de ruptura, ejecuta un step over y continúa hasta la siguiente ruptura.
- La segunda vez que pare, ejecuta un step into y ejecuta step over 3 veces. Después ejecuta step out.

Presenta los resultados en forma de memoria, con capturas de pantalla de cada paso.

Rubrica

	ÍTEM	Criterio Evaluación	PESO
1	Diseño del plan de pruebas	3a	2
1	Pruebas de unidad	3b, 3f, 3g	1
1	Pruebas de integración	3i	1
1	Pruebas de aceptación	3b, 3f, 3g	1
1	Otras pruebas	3a	1
2	Uso del Debugger de intelliJ	3c, 3d	2
2	Presentación de la memoria	3h	2

Se debe entregar un enlace al repositorio y una memoria explicando el proceso punto por punto

Modelo examen

Ejercicio 1: Plan de pruebas

Diseña las siguientes pruebas para una aplicación determinada:

- Diseña un plan de pruebas coherente, teniendo en cuenta
 - Pruebas unitarias: diseña escenarios en los que usarías pruebas tipo JUnit (caja negra, basadas en casos de uso). Es necesario representarlas en forma de tabla y solo es necesario hacer 4, pero que sean relevantes.
 - Pruebas de integración: diseña escenarios en los que usarías pruebas tipo Mockito y GitHub Actions (simulación de componentes y tareas automatizadas en cada cambio).
 - Pruebas de aceptación: diseña escenarios usando pruebas tipo Cucumber, utilizando lenguaje Gherkin para definir los criterios del cliente.
 - Pruebas de seguridad y otros tipos: diseña escenarios donde se ponga a prueba la robustez y protección del sistema ante fallos o ataques.

APLICACIÓN:

Estamos desarrollando una **plataforma de gestión de reservas y pedidos para un hotel**. Los roles principales son: *Recepcionista, Cliente, Gerente, y Limpieza*.

Cada usuario accede desde su propio terminal (dispositivo móvil o PC) utilizando credenciales personalizadas.

La aplicación se conecta con un sistema de reservas externo vía API REST y también accede a una base de datos SQL.

Las funcionalidades están representadas en el siguiente diagrama de casos de uso:

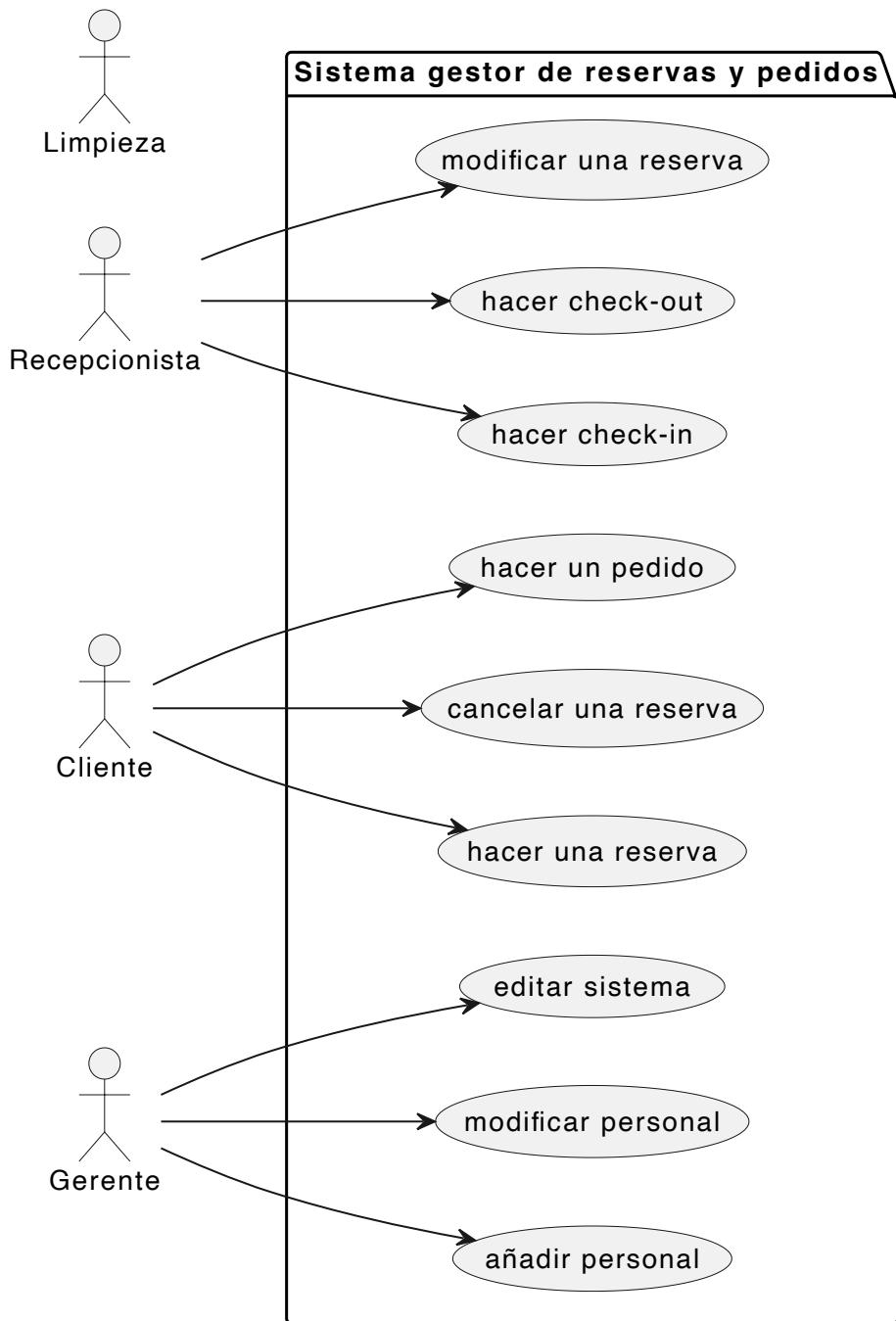


Tabla para las pruebas de unidad:

ID	Nombre de prueba	método	descripción (precondiciones y pasos)	entrada	salida esperada	salida obtenida
1						
2						
3						

ID	Nombre de prueba	método	descripción (precondiciones y pasos)	entrada	salida esperada	salida obtenida
4						

Ejercicio 2: Trazas

Dado el siguiente código:

```

public class EjercicioDebug {
    public static void main(String[] args) {
        int[] n = {3, 7, 2, 9, 5};
        int r = 0;
        for (int i = 0; i < n.length; i++) {
            r += procesaNumero(n[i]); // Colocar breakpoint con condición: n[i] > 5
        }
        System.out.println("Resultado: " + r);
        muestraInfoExtra(n);
    }

    static int procesaNumero(int x) {
        int y = 0;
        if (x % 2 == 0) {
            y = x * 2;
        } else {
            y = x * 3;
        }
        System.out.println("Procesando: " + x);
        System.out.println("Temporal: " + y);
        System.out.println("-----");
        return y;
    }

    static void muestraInfoExtra(int[] arr) {
        int s = 0;
        for (int i = 0; i < arr.length; i++) {
            s += arr[i];
        }
        for (int i = 0; i < arr.length; i++) {
            if (arr[i] % 2 == 0) {
                System.out.println(arr[i] + " es par");
            } else {
                System.out.println(arr[i] + " es impar");
            }
        }
        System.out.println("Suma total: " + s);
    }
}

```

Coloca un punto de ruptura en la línea indicada con la condición establecida y realiza las siguientes pruebas:

- En el primer punto de ruptura, ejecuta un step over y continúa hasta la siguiente ruptura.
- La segunda vez que pare, ejecuta un step into y ejecuta step over 3 veces. Después ejecuta step out.

Refleja el resultado en una traza con tabla de seguimiento de variables y de pasos (como la pila de llamadas del debugger).

Ejercicio 3: Test

1. Las pruebas de caja blanca...
 - a) Sirven para probar el código del programa
 - b) Sirven para probar la entrada y salida del programa
 - c) Sirven para probar la integración de los componentes del sistema
 - d) Son como las pruebas de caja negra, pero las tienes que implementar de forma manual
2. Las pruebas de caja negra...
 - a) Sirven para probar el código del programa
 - b) Sirven para probar la entrada y salida del programa
 - c) Sirven para probar la integración de los componentes del sistema
 - d) Son como las pruebas de caja blanca, pero se generan de forma automática
3. Las pruebas de sistema...
 - a) Prueban los componentes individuales
 - b) Prueban la interacción entre componentes
 - c) Prueban el software en su totalidad
 - d) Prueban que el software cumple todos los requisitos
4. Las pruebas de unidad...
 - a) Prueban los componentes individuales
 - b) Prueban la interacción entre componentes
 - c) Prueban el software en su totalidad
 - d) Prueban que el software cumple todos los requisitos
5. Las pruebas de validación...
 - a) Prueban los componentes individuales
 - b) Prueban la interacción entre componentes
 - c) Prueban el software en su totalidad
 - d) Prueban que el software cumple todos los requisitos
6. Las pruebas de integración...
 - a) Prueban los componentes individuales
 - b) Prueban la interacción entre componentes
 - c) Prueban el software en su totalidad
 - d) Prueban que el software cumple todos los **65** requisitos

7. Un diagrama de casos de uso...

- a) Nos puede ayudar a realizar pruebas de caja blanca
- b) Nos puede ayudar a realizar pruebas de caja negra
- c) Nos puede ayudar a realizar pruebas de integración
- d) No sirve para planificar pruebas

8. Un diagrama de actividad...

- a) Nos puede ayudar a realizar pruebas de caja blanca
- b) Nos puede ayudar a realizar pruebas de caja negra
- c) Nos puede ayudar a realizar pruebas de integración
- d) No sirve para planificar pruebas

9. Las pruebas exploratorias...

- a) Son pruebas con un enfoque riguroso y exhaustivo de prueba de casos reales
- b) Siguen un plan estructurado
- c) Permiten descubrir errores inesperados
- d) Son alternativas a las pruebas automatizadas

10. Acerca de ISTQB, podemos decir...

- a) Que es un framework de software para pruebas
- b) Que es un organismo internacional de estándares de todo tipo
- c) Que reconoce varios tipos de testeo
- d) Que es un organismo internacional de estándares de pruebas de integración continua

11. Qué es la traza de un programa?

- a) La salida del depurador de software
- b) Un sistema consistente en tablas de seguimiento de variables enlazadas
- c) Un registro detallado de la ejecución de un programa
- d) Una prueba que analiza el camino que puede seguir la ejecución de un código

12. La función Step Over de un depurador...

- a) Ejecuta la línea actual y pasa a la siguiente, sin entrar en los detalles de las funciones llamadas
- b) Ejecuta la línea actual y entra en el código de las funciones llamadas para inspeccionarlas
- c) Sale de la función actual y regresa al punto donde fue llamada la función
- d) Muestra la secuencia de llamadas a funciones que llevaron al punto actual de ejecución

Rubrica

	ÍTEM	Criterio Evaluación	PESO
1	Diseño del plan de pruebas	3a	2

	ÍTEM	Criterio Evaluación	PESO
1	Pruebas de unidad	3b, 3f, 3g	1
1	Pruebas de integración	3i	1
1	Pruebas de aceptación	3b, 3f, 3g	1
1	Otras pruebas	3a	1
2	Confección de la traza	3c, 3d	2
2	Test	TODOS	2

Cuaderno de ejercicios de la UP05

Realiza un plan de pruebas de los siguientes enunciados que trabajaste en la unidad de programación 3

Sigue estas instrucciones

- Diseña un plan de pruebas coherente, teniendo en cuenta
 - Pruebas unitarias: diseña escenarios en los que usarías pruebas tipo JUnit (caja negra, basadas en casos de uso). Es necesario representarlas en forma de tabla y solo es necesario hacer 4, pero que sean relevantes.
 - Pruebas de integración: diseña escenarios en los que usarías pruebas tipo Mockito y GitHub Actions (simulación de componentes y tareas automatizadas en cada cambio).
 - Pruebas de aceptación: diseña escenarios usando pruebas tipo Cucumber, utilizando lenguaje Gherkin para definir los criterios del cliente.
 - Pruebas de seguridad y otros tipos: diseña escenarios donde se ponga a prueba la robustez y protección del sistema ante fallos o ataques.

Ejercicio 1: Gestor de Liga de Fútbol

Se ha encargado desarrollar un gestor para una liga de fútbol. La liga debe estar identificada por un nombre y una temporada, definida por el año de inicio y el de finalización. La liga estará compuesta por un máximo de 22 equipos.

Cada equipo debe incluir información como su nombre, el número de partidos ganados, empelados y perdidos. A partir de estos datos, se podrá calcular la puntuación total de cada equipo según el sistema estándar de puntos (3 puntos por victoria, 1 por empate, 0 por derrota). Además, cada equipo debe tener entre 18 y 24 futbolistas.

Cada futbolista debe tener datos que lo identifiquen: nombre, nacionalidad, un número de identificación único, su posición en el campo (portero, defensa, centrocampista o delantero), el número de goles marcados y el número de partidos jugados.

Se requiere que solo exista una instancia de la liga (patrón singleton). Debes implementar la funcionalidad para añadir y eliminar equipos, así como para añadir y eliminar futbolistas de los

equipos. Además, se debe poder acceder a la información completa de cada jugador, equipo o la liga misma.

El sistema debe proporcionar las siguientes funcionalidades:

- Mostrar los equipos en posiciones de descenso (los 4 últimos).
- Mostrar los equipos en posiciones de clasificación para competiciones europeas (los 4 primeros).
- Calcular los goles a favor de cada equipo.
- Identificar al máximo goleador ("pichichi") de la liga.

Ejercicio 2: Gestor de Aeropuertos

Se te ha pedido desarrollar un sistema para gestionar la operativa de un aeropuerto internacional. El aeropuerto debe estar identificado por un nombre, un código IATA de tres letras, y la ciudad donde se encuentra.

Cada aeropuerto gestiona varios vuelos. Un vuelo está identificado por un número único, la aerolínea operadora, el destino y el origen. También debe incluir la hora de salida, la hora de llegada estimada y el estado del vuelo (en hora, retrasado, cancelado).

Cada vuelo tiene una tripulación asignada, formada por al menos un piloto y dos auxiliares de vuelo. Los tripulantes deben incluir información como su nombre, nacionalidad, y su número de identificación único.

Se debe poder:

- Añadir y eliminar vuelos.
- Asignar o eliminar tripulantes de los vuelos.
- Consultar el estado de cualquier vuelo en cualquier momento.
- Filtrar los vuelos por su destino, origen o estado.
- Calcular el tiempo estimado de llegada para los vuelos en base a su hora de salida.

El sistema debe permitir acceder a la información completa de cada vuelo y cada tripulante.

Tabla para las pruebas de unidad:

ID	Nombre de prueba	método	descripción (precondiciones y pasos)	entrada	salida esperada	salida obtenida
1						
2						
3						
4						

Ejercicio de Debug

Dado el siguiente código:

```

public class EjercicioDebug {
    public static void main(String[] args) {
        int[] n = {8, 1, 6, 2, 3};
        int r = 0;
        for (int i = 0; i < n.length; i++) {
            r += procesaNumero(n[i]); // Colocar breakpoint con condición: n[i] > 5
        }
        System.out.println("Resultado: " + r);
        muestraInfoExtra(n);
    }
    static int procesaNumero(int x) {
        int y = 0;
        if (x % 3 == 0) {
            y = (x * 2)/3;
        } else {
            y = x * 3;
        }
        System.out.println("Procesando: " + x);
        System.out.println("Temporal: " + y);
        System.out.println("-----");
        return y;
    }
}

static void muestraInfoExtra(int[] arr) {
    int s = 0;
    for (int i = 0; i < arr.length; i++) {
        s += arr[i];
    }
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] % 3 == 0) {
            System.out.println(arr[i] + " es divisible entre 3");
        } else {
            System.out.println(arr[i] + " no es divisible entre 3");
        }
    }
    System.out.println("Suma total: " + s);
}
}

```

Coloca un punto de ruptura en la línea indicada con la condición establecida y realiza las siguientes pruebas:

- En el primer punto de ruptura, ejecuta un step over y continúa hasta la siguiente ruptura.
- La segunda vez que pare, ejecuta un step into y ejecuta step over 3 veces. Después ejecuta step out.

Refleja el resultado en una traza con tabla de seguimiento de variables y de pasos (como la pila de llamadas del debugger).

Ampliación: Pruebas en Python

- Ampliación: Pruebas en Python
 - Pruebas Unitarias y Dobles de Prueba en Python: Implementación Práctica
 - 1. Pruebas Unitarias con `unittest`
 - Estructura Básica de una Prueba Unitaria
 - 2. Dobles de Prueba con `unittest.mock`
 - `MagicMock` (El análogo a los Mocks de Mockito)
 - `patch` (Inyección de Mocks/Stubs)
 - Patrones Adicionales de `patch`
 - Consideraciones Finales

Pruebas Unitarias y Dobles de Prueba en Python: Implementación Práctica

Para desarrolladores familiarizados con frameworks de prueba como JUnit y bibliotecas de mocking como Mockito, la transición a Python es fluida. El ecosistema de pruebas de Python se centra principalmente en el módulo `unittest` de la biblioteca estándar, que ofrece una funcionalidad robusta para pruebas unitarias, y `unittest.mock`, fundamental para la implementación de dobles de prueba.

1. Pruebas Unitarias con `unittest`

El módulo `unittest` es la piedra angular para las pruebas unitarias en Python. Su diseño se inspira en JUnit y otros frameworks de pruebas de xUnit.

Estructura Básica de una Prueba Unitaria

Cada conjunto de pruebas se organiza en **clases de prueba** que heredan de `unittest.TestCase`. Los métodos dentro de estas clases que comienzan con `test_` son reconocidos automáticamente como **métodos de prueba**.

Ejemplo de Código bajo Prueba (CUT - Code Under Test):

Consideremos un módulo `calculadora.py` que contiene funciones aritméticas básicas:

```
# calculadora.py

class Calculadora:
    def sumar(self, a, b):
        """Suma dos números."""
        return a + b

    def restar(self, a, b):
        """Resta dos números."""
        return a - b

    def dividir(self, a, b):
        """Divide dos números. Lanza ValueError si el divisor es cero."""
        if b == 0:
            raise ValueError("No se puede dividir por cero.")
        return a / b
```

Creando un Archivo de Pruebas:

Por convención, los archivos de prueba suelen nombrarse con el prefijo `test_`. Crearemos `test_calculadora.py`:

```

# test_calculadora.py
import unittest
from calculadora import Calculadora

class TestCalculadora(unittest.TestCase):
    """Clase de pruebas para la clase Calculadora."""

    def setUp(self):
        """
        Método que se ejecuta ANTES de cada método de prueba.
        Inicializa una nueva instancia de Calculadora para cada prueba.
        """
        self.calc = Calculadora()

    def test_sumar(self):
        """Verifica la función sumar con diferentes escenarios."""
        self.assertEqual(self.calc.sumar(5, 3), 8)
        self.assertEqual(self.calc.sumar(-1, 1), 0)
        self.assertEqual(self.calc.sumar(0, 0), 0)

    def test_restar(self):
        """Verifica la función restar."""
        self.assertEqual(self.calc.restar(10, 4), 6)
        self.assertEqual(self.calc.restar(4, 10), -6)

    def test_dividir_numeros_enteros(self):
        """Verifica la función dividir con números enteros."""
        self.assertEqual(self.calc.dividir(10, 2), 5.0)

    def test_dividir_por_cero(self):
        """Verifica que dividir por cero levante un ValueError."""
        with self.assertRaises(ValueError):
            self.calc.dividir(10, 0)

    def tearDown(self):
        """
        Método que se ejecuta DESPUÉS de cada método de prueba.
        Utilizado para limpiar recursos si fuera necesario (ej. cerrar conexiones).
        """
        self.calc = None # Liberar referencia, aunque para este objeto simple no es crí

# Punto de entrada para ejecutar las pruebas

```

```
if __name__ == '__main__':
    unittest.main()
```

Métodos Clave de `unittest.TestCase`:

- **setUp()** : Se ejecuta antes de cada método de prueba. Ideal para inicializar objetos o preparar el entorno. Equivalente a `@BeforeEach` en JUnit 5 o `@Before` en JUnit 4.
- **tearDown()** : Se ejecuta después de cada método de prueba. Ideal para limpiar recursos. Equivalente a `@AfterEach` o `@After`.
- **setUpClass(cls)** : (Método de clase) Se ejecuta una vez antes de todas las pruebas en la clase. Equivalente a `@BeforeAll`.
- **tearDownClass(cls)** : (Método de clase) Se ejecuta una vez después de todas las pruebas en la clase. Equivalente a `@AfterAll`.

Métodos de Aserción (Ejemplos):

Los métodos de asercción de `unittest.TestCase` son cruciales para verificar los resultados de las pruebas.

- `self.assertEqual(a, b)` : $a == b$
- `self.assertNotEqual(a, b)` : $a \neq b$
- `self.assertTrue(x)` : `bool(x)` es `True`
- `self.assertFalse(x)` : `bool(x)` es `False`
- `self.assertEqual(a, b)` : a es b (identidad de objeto)
- `self.assertNotEqual(a, b)` : a no es b
- `self.assertIsNone(x)` : x es `None`
- `self.assertIsNotNone(x)` : x no es `None`
- `self.assertIn(member, container)` : `member` está en `container`
- `self.assertNotIn(member, container)` : `member` no está en `container`
- `self.assertRaises(exception, callable, *args, **kwargs)` : Verifica que `callable(*args, **kwargs)` levante la `exception` especificada. También puede usarse como un gestor de contexto (`with self.assertRaises(Exception): ...`).

Ejecución de Pruebas:

Para ejecutar las pruebas, navega al directorio que contiene `test_calculadora.py` en tu terminal y ejecuta:

```
python -m unittest test_calculadora.py
```

O simplemente:

```
python test_calculadora.py
```

Si tus pruebas están en un directorio y quieres descubrirlas automáticamente, puedes usar:

```
python -m unittest discover
```

Actividad

Realiza las actividades propuestas en el documento de pruebas de unidad de JUnit, traducidas a python.

2. Dobles de Prueba con `unittest.mock`

`unittest.mock` es la herramienta estándar de Python para crear dobles de prueba (Stubs, Mocks, Spies, Fakes). Su funcionalidad es comparable a la de Mockito en Java, permitiendo reemplazar objetos reales con objetos controlados para aislar la unidad bajo prueba y verificar interacciones.

Los elementos clave son `MagicMock` y la función `patch`.

MagicMock (El análogo a los Mocks de Mockito)

`MagicMock` es una clase versátil que puede simular cualquier objeto o método, y registra todas las interacciones.

Conceptos clave:

- **Comportamiento predefinido:** Puedes configurar valores de retorno (`.return_value`), excepciones (`.side_effect`), o incluso implementar lógicas personalizadas.
- **Verificación de interacciones:** Registra llamadas a métodos, argumentos pasados y cuántas veces se llamaron.

patch (Inyección de Mocks/Stubs)

`patch` es una función (o decorador) que reemplaza temporalmente un objeto por un mock durante la ejecución de una prueba. Es tu principal mecanismo para la **inversión de control** en pruebas.

Ejemplo de Código con Dependencia Externa:

Consideremos una clase `GestorDeUsuarios` que interactúa con un `ServicioDeBaseDeDatos` para obtener y guardar usuarios.

```

# gestor_usuarios.py

class ServicioDeBaseDeDatos:
    def obtener_usuario(self, user_id):
        """Simula una consulta a base de datos real."""
        # En una aplicación real, esto interactuaría con una DB.
        print(f"DEBUG: Obteniendo usuario {user_id} de la DB real...")
        if user_id == 101:
            return {"id": 101, "nombre": "Alice", "email": "alice@example.com"}
        return None

    def guardar_usuario(self, user_data):
        """Simula guardar un usuario en la base de datos."""
        print(f"DEBUG: Guardando usuario {user_data['id']} en la DB real...")
        # Lógica real para guardar
        return True

class GestorDeUsuarios:
    def __init__(self):
        self.db_servicio = ServicioDeBaseDeDatos()

    def obtener_info_usuario(self, user_id):
        """Obtiene información de usuario y la formatea."""
        usuario = self.db_servicio.obtener_usuario(user_id)
        if usuario:
            return f"ID: {usuario['id']}, Nombre: {usuario['nombre']}, Email: {usuario['email']}"
        return "Usuario no encontrado."

    def registrar_usuario(self, user_id, nombre, email):
        """Registra un nuevo usuario en la base de datos."""
        nuevo_usuario = {"id": user_id, "nombre": nombre, "email": email}
        if self.db_servicio.guardar_usuario(nuevo_usuario):
            return "Usuario registrado exitosamente."
        return "Fallo al registrar el usuario."

```

Creando Pruebas con `unittest.mock.patch`:

```
# test_gestor_usuarios.py
import unittest
from unittest.mock import patch, MagicMock
from gestor_usuarios import GestorDeUsuarios

class TestGestorDeUsuarios(unittest.TestCase):

    # Usamos @patch para reemplazar ServicioDeBaseDeDatos con un mock
    # El string 'gestor_usuarios.ServicioDeBaseDeDatos' es la ruta al objeto que se va
    @patch('gestor_usuarios.ServicioDeBaseDeDatos')
    def test_obtener_info_usuario_existente(self, MockServicioDeBaseDeDatos):
        # MockServicioDeBaseDeDatos es la clase Mockeada.
        # Cuando GestorDeUsuarios() crea una instancia de ServicioDeBaseDeDatos(),
        # recibirá una instancia de MagicMock. Necesitamos configurar el comportamiento
        mock_instancia_db = MockServicioDeBaseDeDatos.return_value

        # Configuramos el stub para el método obtener_usuario
        mock_instancia_db.obtener_usuario.return_value = {
            "id": 200, "nombre": "Bob", "email": "bob@example.com"
        }

        gestor = GestorDeUsuarios()
        resultado = gestor.obtener_info_usuario(200)

        # Verificaciones (Assertions)
        self.assertEqual(resultado, "ID: 200, Nombre: Bob, Email: bob@example.com")
        # Verificamos que el método obtener_usuario fue llamado con los argumentos correctos
        mock_instancia_db.obtener_usuario.assert_called_once_with(200)
        # Verificamos que no se llamó a guardar_usuario
        mock_instancia_db.guardar_usuario.assert_not_called()

    @patch('gestor_usuarios.ServicioDeBaseDeDatos')
    def test_obtener_info_usuario_no_existe(self, MockServicioDeBaseDeDatos):
        mock_instancia_db = MockServicioDeBaseDeDatos.return_value
        mock_instancia_db.obtener_usuario.return_value = None # Simula que el usuario no existe

        gestor = GestorDeUsuarios()
        resultado = gestor.obtener_info_usuario(999)

        self.assertEqual(resultado, "Usuario no encontrado.")
        mock_instancia_db.obtener_usuario.assert_called_once_with(999)

    @patch('gestor_usuarios.ServicioDeBaseDeDatos')
```

```

def test_registrar_usuario_exitoso(self, MockServicioDeBaseDeDatos):
    mock_instancia_db = MockServicioDeBaseDeDatos.return_value
    mock_instancia_db.guardar_usuario.return_value = True # Simula éxito al guardar

    gestor = GestorDeUsuarios()
    resultado = gestor.registrar_usuario(300, "Charlie", "charlie@example.com")

    self.assertEqual(resultado, "Usuario registrado exitosamente.")
    # Verificamos que guardar_usuario fue llamado con los datos correctos
    mock_instancia_db.guardar_usuario.assert_called_once_with(
        {"id": 300, "nombre": "Charlie", "email": "charlie@example.com"})
    )
    mock_instancia_db.obtener_usuario.assert_not_called() # No debería haber llamad

@patch('gestor_usuarios.ServicioDeBaseDeDatos')
def test_registrar_usuario_fallido(self, MockServicioDeBaseDeDatos):
    mock_instancia_db = MockServicioDeBaseDeDatos.return_value
    mock_instancia_db.guardar_usuario.return_value = False # Simula fallo al guarda

    gestor = GestorDeUsuarios()
    resultado = gestor.registrar_usuario(400, "Diana", "diana@example.com")

    self.assertEqual(resultado, "Fallo al registrar el usuario.")
    mock_instancia_db.guardar_usuario.assert_called_once_with(
        {"id": 400, "nombre": "Diana", "email": "diana@example.com"})
    )

if __name__ == '__main__':
    unittest.main()

```

Puntos Clave sobre patch y MagicMock :

- Ruta de patch :** El argumento a patch es una cadena de texto que indica la ruta *donde el objeto se busca* en el momento de la ejecución. Por ejemplo, si GestorDeUsuarios importa ServicioDeBaseDeDatos como from gestor_usuarios import ServicioDeBaseDeDatos , entonces la ruta para mockear es gestor_usuarios.ServicioDeBaseDeDatos .
- Mock de la Clase vs. Mock de la Instancia:** Cuando usas @patch('modulo.Clase') , MockServicioDeBaseDeDatos (el argumento que recibe el método de prueba) es un mock de la clase ServicioDeBaseDeDatos . Para interactuar con los métodos que serían llamados en una instancia de esa clase (ej. obtener_usuario), debes acceder a MockServicioDeBaseDeDatos.return_value . Esto simula lo que ocurre cuando GestorDeUsuarios hace self.db_servicio = ServicioDeBaseDeDatos() .

3. Configuración de Retornos y Efectos Secundarios:

- `mock_objeto.metodo.return_value = valor` : Configura el valor que devolverá el método mockeado.
- `mock_objeto.metodo.side_effect = Excepcion` : Hace que el método mockeado levante una excepción.
- `mock_objeto.metodo.side_effect = [valor1, valor2, ...]` o
`mock_objeto.metodo.side_effect = funcion` : Permite definir una secuencia de retornos o una función personalizada que se ejecutará.

4. Métodos de Verificación de Mocks (Análogos a `Mockito.verify()`):

- `mock_objeto.metodo.assert_called()` : Verifica que el método fue llamado al menos una vez.
- `mock_objeto.metodo.assert_called_once()` : Verifica que el método fue llamado exactamente una vez.
- `mock_objeto.metodo.assert_called_with(*args, **kwargs)` : Verifica que el método fue llamado con los argumentos especificados.
- `mock_objeto.metodo.assert_called_once_with(*args, **kwargs)` : Verifica que fue llamado exactamente una vez con los argumentos especificados.
- `mock_objeto.metodo.assert_not_called()` : Verifica que el método no fue llamado.

Patrones Adicionales de `patch`

- `@patch.object()` : Permite mockear un atributo o método específico de un objeto ya existente, en lugar de una clase completa o un objeto a nivel de módulo.

```
from unittest.mock import patch
# ... dentro de una clase de prueba
def test_con_patch_object(self):
    gestor = GestorDeUsuarios() # Instancia real
    with patch.object(gestor.db_servicio, 'obtener_usuario') as mock_obtener:
        mock_obtener.return_value = {"id": 1, "nombre": "Juan"}
        resultado = gestor.obtener_info_usuario(1)
        self.assertEqual(resultado, "ID: 1, Nombre: Juan, Email: None") # Email es None
        mock_obtener.assert_called_once_with(1)
```

- `patch como gestor de contexto (with patch(...) as mock_obj:)`: Útil para mockear objetos que no son clases, o para un control más granular del alcance del mock. El mock se revierte automáticamente al salir del bloque `with`.

Consideraciones Finales

- **Alcance de los Mocks:** Los mocks creados con `patch` se activan y desactivan automáticamente dentro del alcance del decorador o del bloque `with`. Esto asegura que las pruebas son aisladas y no interfieren entre sí.
- **Velocidad de Ejecución:** Python `unittest` es generalmente rápido. Los dobles de prueba contribuyen significativamente a mantener esta velocidad al evitar interacciones con sistemas lentos.
- **Mantenimiento:** Escribir pruebas claras y bien estructuradas, junto con un uso juicioso de los dobles de prueba, reduce el acoplamiento y facilita el mantenimiento del código a largo plazo.
- **Pytest:** Aunque `unittest` es el estándar, **Pytest** es un framework de pruebas de terceros muy popular en la comunidad de Python, conocido por su sintaxis más concisa, sus "fixtures" (que son una forma elegante de configurar el entorno de prueba) y su poderosa integración con `unittest.mock`. Si buscas una experiencia aún más simplificada, Pytest es una excelente opción a explorar después de dominar `unittest`.

Actividad

Realiza las actividades propuestas en el documento de pruebas de integración de Mockito, traducidas a python.

Pruebas de Seguridad

En la ingeniería de software contemporánea, la **seguridad** no es un componente adicional, sino un aspecto intrínseco del ciclo de vida del desarrollo de software (SDLC). Las pruebas de seguridad son un conjunto de actividades diseñadas para evaluar la resiliencia de una aplicación frente a ataques maliciosos, identificar vulnerabilidades y asegurar que los controles de seguridad implementados funcionan como se espera.

- Pruebas de Seguridad
 - Importancia de las pruebas de seguridad
 - Tipos de Pruebas de Seguridad
 - Pruebas de Seguridad de Aplicaciones Estáticas (SAST)
 - Pruebas de Seguridad de Aplicaciones Dinámicas (DAST)
 - Análisis de Composición de Software (SCA)
 - Pruebas de Penetración (Pentesting)
 - Estrategias y Consideraciones Clave
 - Recursos para la realización de pruebas de seguridad

Actividad

Realiza un mapa conceptual sobre las pruebas de seguridad y experimenta con alguna de las aplicaciones presentadas al final de este apartado

Importancia de las pruebas de seguridad

La integración de pruebas de seguridad es indispensable por múltiples razones. Permiten la **prevención de brechas de datos**, salvaguardando la información sensible y la privacidad de los usuarios. Aseguran el **cumplimiento normativo** con regulaciones cada vez más estrictas, como GDPR o PCI DSS. Además, protegen la **reputación** de una organización, ya que un incidente de seguridad puede tener consecuencias devastadoras para la confianza del cliente. Finalmente, **reducen costos** significativamente, puesto que corregir vulnerabilidades en las primeras etapas del desarrollo es considerablemente más económico que hacerlo tras un incidente en producción.

Tipos de Pruebas de Seguridad

Las pruebas de seguridad se clasifican principalmente por la metodología y el punto del SDLC en el que se aplican:

Pruebas de Seguridad de Aplicaciones Estáticas (SAST)

El **SAST** analiza el código fuente, el bytecode o los binarios de una aplicación sin ejecutarla. Se realiza en fases tempranas del SDLC, a menudo durante el desarrollo o la integración continua. Su objetivo es identificar vulnerabilidades comunes como inyección SQL, Cross-Site Scripting (XSS) y configuraciones inseguras. La principal ventaja del SAST radica en su capacidad para detectar fallos tempranamente y proporcionar la ubicación precisa de la vulnerabilidad en el código. Sin embargo, puede generar un número considerable de falsos positivos y no detecta vulnerabilidades que solo se manifiestan en tiempo de ejecución.

Pruebas de Seguridad de Aplicaciones Dinámicas (DAST)

El **DAST** evalúa la aplicación mientras está en ejecución, interactuando con ella desde una perspectiva externa, similar a como lo haría un atacante. Este enfoque revela vulnerabilidades que se manifiestan durante la ejecución, como problemas de autenticación, autorización, manejo de sesiones y validación de entradas. Una ventaja clave del DAST es su capacidad para simular ataques reales y detectar problemas relacionados con el entorno de ejecución, como configuraciones incorrectas. No obstante, no proporciona la ubicación exacta del código fuente vulnerable y su cobertura depende de las rutas de la aplicación que se ejerciten.

Análisis de Composición de Software (SCA)

El **SCA** se enfoca en identificar y evaluar las vulnerabilidades de seguridad inherentes en las **bibliotecas y componentes de código abierto** que una aplicación utiliza. Dada la prevalencia del uso de componentes de terceros, este análisis es crucial para detectar dependencias con vulnerabilidades conocidas (CVEs), las cuales podrían introducir puntos débiles significativos en la aplicación final.

Pruebas de Penetración (Pentesting)

El **pentesting** es una simulación controlada de un ataque real al sistema, llevada a cabo por pentesters éticos. Combina herramientas automatizadas con habilidades manuales y un conocimiento profundo de las tácticas de ataque para identificar y explotar vulnerabilidades críticas que las herramientas automáticas podrían pasar por alto. Los resultados se materializan en informes detallados sobre la explotabilidad de las vulnerabilidades y recomendaciones específicas para su mitigación.

Estrategias y Consideraciones Clave

Para una implementación efectiva de las pruebas de seguridad, es fundamental adoptar un enfoque integral. Esto incluye la **seguridad por diseño**, integrando consideraciones de seguridad desde las fases iniciales del diseño arquitectónico. La **revisión manual de código** por parte de expertos en seguridad puede complementar las herramientas automatizadas, especialmente en la lógica de negocio crítica. El uso de guías como el **OWASP Top 10** proporciona una hoja de ruta invaluable para priorizar las pruebas. Además, la **integración de herramientas de seguridad en los pipelines de CI/CD** automatiza la detección de vulnerabilidades, proporcionando una retroalimentación continua y rápida a los equipos de desarrollo.

Recursos para la realización de pruebas de seguridad

Para experimentar de primera mano con las pruebas de seguridad en entornos Java, existen diversas aplicaciones deliberadamente vulnerables. Estas aplicaciones están diseñadas para simular escenarios de vulnerabilidades comunes y permiten practicar la detección y explotación de fallos de seguridad de manera ética y controlada.

- **OWASP WebGoat**: Probablemente el proyecto más conocido para aprender sobre seguridad de aplicaciones web. WebGoat es una aplicación web de Java diseñada por OWASP que contiene una serie de lecciones interactivas. Cada lección presenta una vulnerabilidad específica (como inyección SQL, XSS, control de acceso roto, etc.) y guía al usuario para explotarla y luego comprender cómo mitigarla. Es una excelente plataforma para practicar DAST y revisión manual de código.
- **OWASP Juice Shop**: Aunque no está construida en Java, es una aplicación Node.js/Angular.js, su naturaleza de aplicación web moderna con una amplia gama de vulnerabilidades la convierte en una plataforma de prueba fantástica para DAST y pruebas de penetración, sin importar la tecnología de backend específica. Las técnicas para encontrar vulnerabilidades son agnósticas al lenguaje del servidor en muchos casos.
- **Damn Vulnerable Java Application (DVJA)**: Similar a WebGoat, DVJA es una aplicación web Java creada para ser vulnerable. Ofrece un entorno controlado para que los estudiantes y profesionales de la seguridad puedan probar y experimentar con diferentes tipos de vulnerabilidades de aplicaciones web y Java, incluyendo deserialización insegura y problemas de manejo de archivos.
- **bWAPP (Buggy Web Application)**: Es una aplicación web gratuita y de código abierto que contiene más de 100 vulnerabilidades web. Si bien está escrita principalmente en PHP, muchas de las vulnerabilidades son agnósticas al lenguaje y pueden ser un excelente banco de pruebas para herramientas DAST y principios de pruebas de penetración que son aplicables a cualquier aplicación web, incluidas las Java.

La utilización de estas plataformas en un entorno de laboratorio permite a los alumnos desarrollar habilidades prácticas en la identificación, explotación y mitigación de vulnerabilidades, un conocimiento crítico para cualquier desarrollador que aspire a construir software seguro.