UNIVERSIDAD
DE LA FRONTERA

FACULTAD DE INGENIERÍA Y CIENCIAS

Departamento de Ciencias de la Computación e Informática

RT-POO-03

Pruebas Unitarias y Manejo de Excepciones

Profesor

Dr. Samuel Sepúlveda Cuevas

Ayudante

Jesús Tapia Martin

Código

ICC490

Temuco, Chile.
Noviembre, 2025

Índice

1. Pruebas Unitarias

Las *Pruebas Unitarias* son procedimientos automatizados que validan que cada parte del código funcione como se espera. Su objetivo es garantizar un software libre de errores facilitando correcciones rápidas y reduciendo costos de mantenimiento **[3]**.

1.1 Importancia de las pruebas unitarias

La ausencia de pruebas en un proyecto dificulta garantizar que los cambios en el software funcionen correctamente. Por ello, el código de prueba debe ser considerado tan importante como el código de producción, o incluso más dentro del proceso de desarrollo.

Un buen conjunto de pruebas influye directamente en la facilidad de realizar modificaciones y reducir el riesgo de errores, asegurando que el software sea más estable y mantenible a largo plazo.

«El código de prueba es tan importante como el código de producción» **[1]**

> **Demostración: Desastre de Ariane 5 - No uso de pruebas**
> El desastre del Ariane 5 (4 de junio de 1996) resultó en la destrucción del cohete en menos de 40 segundos después de su lanzamiento. La explosión fue causada por un problema en el sistema de referencia inercial, donde un valor de 64 bits se intentó almacenar en una variable de 16 bits, provocando un desbordamiento y una interpretación incorrecta de los datos. Todo esto provocó una corrección errónea de los motores, causado la directa destrucción del cohete.
>
> Este fallo se debió, en parte, al reciclaje del código del Ariane 4 que contenía elementos innecesarios para el nuevo diseño del Ariane 5.

1.2 Un concepto por prueba

Cada prueba unitaria debe evaluar una única *responsabilidad* en el código, evitando evaluar múltiples módulos o responsabilidades. Por lo tanto, es recomendable dividir en pruebas independientes y pequeñas.

Esto permite identificar qué parte del código falla o pasa la prueba, haciendo el proceso de depuración sencillo y eficiente, ya que se sabe exactamente qué módulo está funcionando correctamente o cuál necesita ajustes.

Definición: assert

Los métodos *assert* se utilizan en las pruebas unitarias para verificar que una condición o un resultado obtenido del código, coincida con el valor que se espera.
Si es necesario se puede agregar más de un *assert* por prueba, siempre y cuando estén relacionadas con el mismo concepto a probar.

1.2.1. Ejemplo

Para ver la diferencia entre un test con varios conceptos y el enfoque de un test para una única responsabilidad, se considerará el ejemplo de una clase **«Calculadora»** con métodos para realizar las diferentes operaciones.

- En el código con **mala práctica**, se intenta cubrir múltiples casos en una sola prueba, resultando un test ambiguo sobrecargado con lógica y *asserts*, lo cuál dificulta el entendimiento y la identificación de qué prueba falla si hay error.

```
public class CalculadoraTest {
    @Test
    public void testCalculadoraMalaPractica() {
        Calculadora calc = new Calculadora();

        assertEquals(5, calc.sumar(2, 3));
        assertEquals(2, calc.dividir(6, 3));

        try {
            calc.dividir(6, 0);
        } catch (ArithmeticException e) {
            assertEquals("Error: Divisor es 0", e.getMessage());
        }
    }
}
```

Listing 1: Mala práctica: Varios conceptos en una sola prueba

- En contraste, el **código con buena práctica** contiene pruebas que verifican una única responsabilidad, identificando el módulo que falla en caso de error.

```
public class CalculadoraTest {
    @Test
    public void testSuma() {
```

```
        Calculadora calc    = new Calculadora();
        int resultado       = calc.sumar(2, 3);
        assertEquals(5, resultado);
    }

    @Test
    public void testResta() {
        Calculadora calc    = new Calculadora();
        int resultado       = calc.restar(5, 3);
        assertEquals(2, resultado);
    }

    @Test
    public void testMultiplicar() {
        Calculadora calc    = new Calculadora();
        int resultado       = calc.multiplicar(2, 3);
        assertEquals(6, resultado);
    }

    @Test
    public void testDividir() {
        Calculadora calc    = new Calculadora();
        int resultado       = calc.dividir(6, 3);
        assertEquals(2, resultado);
    }
}
```

Listing 2: Buena práctica: Un concepto por prueba

1.3 Pruebas limpias

Una prueba es legible cuando es sencilla y concisa en su contenido, logrando comunicar claramente qué se está probando y qué se espera como resultado, con una cantidad mínima de expresiones en el código. Las pruebas limpias se destacan por su facilidad de comprensión y mantenimiento.

> ***«Tener pruebas sucias es equivalente, si no peor, que no tener pruebas»***
> **[1]**

1.4 Flexibilidad de las pruebas unitarias

Sin pruebas en el software, cada modificación podrá representar la introducción a un posible error, independientemente de la flexibilidad y calidad del diseño.

> ***«Son las pruebas unitarias las que mantienen nuestro código flexible, mantenible y reutilizable»*** **[1]**

Ejemplo:
En una aplicación de gestión de inventarios, las pruebas unitarias aseguran que el cálculo del stock sea correcto, ya que si en un futuro se cambia la lógica para agregar descuentos por cantidad o precios variables, las pruebas previamente escritas validarán que el cálculo original del stock no se vea afectado.

1.5 Integrar pruebas unitarias desde el principio

Las pruebas unitarias deben ser implementadas desde las etapas tempranas del desarrollo del software, en lugar de esperar a que el proyecto crezca o se vuelva demasiado complejo. Con esto se obtiene un código robusto desde un inicio al minimizar los riesgos de problemas durante fases posteriores del proyecto.

Ejemplo:
En el desarrollo de una API para gestionar usuarios, se implementan pruebas unitarias desde el inicio para verificar que la creación y eliminación de usuarios funcionan correctamente. Con esto se asegura que al añadir funciones más complejas como la actualización de perfiles o la gestión de permisos, se detecten rápidamente posibles errores introducidos por los nuevos cambios.

1.6 Testing continuo

El testing debe ser un proceso constante a lo largo del ciclo de vida del desarrollo. Se recomienda crear y ejecutar pruebas unitarias cada vez que se realicen cambios importantes para garantizar que las modificaciones no introduzcan nuevos errores. La integración de herramientas de integración continua (CI) automatiza este proceso, ayudando a detectar fallos antes de desplegar cualquier actualización.

2.1 Definición

El *Manejo de Excepciones* gestiona errores inesperados para que el software siga su flujo de ejecución normal sin interrupciones.

2.2 Utilizar excepciones en lugar de retornos

Utilizar *retornos* para manejar errores mezcla la *lógica principal* con el *manejo de errores*, haciendo el código ambiguo y díficil de mantener.

En base a esto, es preferible lanzar excepciones usando bloques *try-catch*, separando la lógica principal del manejo de errores para evitar verificaciones innecesarias en cada invocación.

2.2.1. Ejemplo

Para entender la diferencia entre retornos y excepciones en el manejo de errores, se abordará un ejemplo de división entre dos números.

- En el **código con mala práctica**, se verifica si el divisor es 0 para devolver un valor arbitrario en lugar de lanzar una excepción. Esto permite que el programa continúe incluso con un error, obligando a realizar verificaciones adicionales tras cada llamada al método.

```
public class Calculadora {
    public int dividir(int a, int b) {
        if (b == 0) {
            System.out.println("Error: división por cero");
            return -1;
        }
        if (a == 0) {
            System.out.println("Advertencia: Dividendo es 0");
            return 0;
        }
        return a / b;
    }

    public static void main(String[] args) {
        Calculadora c   = new Calculadora();
        int result      = c.dividir(0, 10);
```

```
        if (result == -1) {
            System.out.println("Error: División no permitida");
        } else {
            System.out.println("Resultado: " + result);
        }
    }
}
```

Listing 3: Mala práctica - Utilizar excepciones en lugar de retornos

- En contraste, el **código con buena práctica** lanza una excepción *ArithmeticException* cuando el divisor es 0, capturando y manejando la excepción con un bloque *try-catch*.

  Con esto se separa la lógica del manejo de errores de la lógica principal, permitiendo describir la causa del error de forma clara y precisa.

```
public class Calculadora {
    public int dividir(int a, int b) throws ArithmeticException {
        if (b == 0) {
            throw new ArithmeticException("Error: División por 0");
        }
        return a / b;
    }

    public static void main(String[] args) {
        Calculadora c = new Calculadora();

        try {
            int result = c.dividir(0, 10);
            System.out.println("Resultado: " + result);
        } catch (ArithmeticException e) {
            System.err.println(e.getMessage());
        }
    }
}
```

Listing 4: Buena práctica - Utilizar excepciones en lugar de retornos

2.3 Escribir el Bloque Try-Catch Primero

Al escribir un método propenso a excepciones, abordarlo directamente con un bloque *try-catch* anticipa cómo se manejarán los errores desde el principio.

Este enfoque facilita visualizar cómo será el comportamiento del programa ante fallos, mejorando la estructura y reduciendo la probabilidad de que los errores pasen desapercibidos, manteniendo el sistema manejable y consistente en situaciones excepcionales.

> **Analogía:**
> Esta práctica es similar al diseño de un edificio, el cual incorpora medidas de seguridad desde el inicio para prevenir que este colapse. De igual manera, escribir el bloque **«try-catch»** desde el comienzo es igual a planificar esos refuerzos antes de que aparezcan los problemas, garantizando que el programa maneje los errores de forma controlada, incluso en situaciones inesperadas.

2.4 Proporcionar contexto con excepciones

El bloque *catch* debe proporcionar suficiente contexto para identificar la causa del error, incluyendo mensajes de error o información sobre el estado del programa en el momento de la excepción, facilitando la depuración y corrección de problemas.

2.4.1. Ejemplo

Para entender el valor de entregar contexto y detalles mediante una excepción, se abordara el siguiente ejemplo:

- En el **código con mala práctica**, el bloque *catch* imprime un mensaje genérico que carece de detalles sobre la naturaleza del error dificultando su identificación.

```
try {
    // Código que puede lanzar una excepción
} catch (IOException e) {
    // Mensaje genérico sin proporcionar información relevante
    System.err.println("Se ha producido un error");
}
```

Listing 5: Mala práctica - No proporcionar contexto en las excepciones

- En contraste, el **código con buena práctica** presenta un bloque *catch* que sí proporciona detalles sobre el error, incluyendo el tipo de excepción, un mensaje específico e información

relevante como la ruta del archivo. Esto facilita la depuración y permite tomar medidas correctivas de manera eficiente.

```java
try {
    // Código que puede lanzar una excepción
} catch (IOException e) {
    System.err.println("Error al intentar leer el archivo: " + e.getMessage());
    System.err.println("Ruta del archivo: " + filePath);
    e.printStackTrace();
}
```

Listing 6: Buena práctica - Proporcionar contexto en las excepciones

2.5 No retornar null

Retornar *null* en un método generará complejidades adicionales en el código cliente, ya que se requerirán múltiples verificaciones para evitar errores como *NullPointerException*. Esto resulta en un código menos mantenible y altamente propenso a convertirse en *Código Spaghetti*, dificultando su comprensión y manejo.

En lugar de retornar *null*, se recomienda lanzar una excepción adecuada o devolver un objeto especial que represente una condición específica, como una lista vacía o un valor predeterminado.

2.5.1. Ejemplo

Para entender la diferencia entre retornar *null* y adoptar un enfoque que evite retornar *null*, se abordará un ejemplo donde se genere una lista de números pares menores o iguales a un número dado.

- En el **código con mala práctica**, si el número es nulo o negativo se emite un mensaje de error y se retorna *null*, lo que obliga a realizar verificaciones adicionales para evitar excepciones como *NullPointerException*, aumentando así la complejidad y la propensión a errores en la lógica del programa.

```java
public List<Integer> generarPares(int numero) {
    if (numero < 0) {
        System.out.println("Error: El número es negativo");
        return null;
    }
    if (numero > 1000) {
        System.out.println("Advertencia: Número es grande");
```

```
        return null;
    }
    if (numero == 0) {
        System.out.println("Número es 0");
        return null;
    }
    List<Integer> pares = new ArrayList<>();
    for (int i = 0; i <= numero; i += 2) {
        pares.add(i);
    }
    return pares;
}
```

Listing 7: Mala práctica - Retornar null - Método para generar pares

```
public static void main(String[] args) {
    List<Integer> pares = generarPares(-5);
    if (pares != null) {
        System.out.println("Generado con éxito: " + pares);
    } else {
        System.out.println("Error: La lista de pares es null");
    }

    pares = generarPares(1001);
    if (pares != null) {
        for (Integer par : pares) {
            System.out.println(par);
        }
    } else {
        System.out.println("Error: La lista es null");
    }

    pares = generarPares(0);
    if (pares == null) {
        System.out.println("Error: La lista de pares es null");
    } else {
        System.out.println("Lista generada correctamente");
    }
}
```

Listing 8: Mala práctica - Retornar null - Método main

- En contraste, el **código con buena práctica** lanza una excepción *IllegalArgumentException* al recibir un valor no válido en lugar de retornar *null*. Con este enfoque, se está comunicando claramente la causa del error, eliminando la necesidad de comprobaciones adicionales.

```
public static List<Integer> generarPares(int numero) {
    if (numero < 0) {
        throw new IllegalArgumentException("El número no puede ser negativo");
    }
    if (numero > 1000) {
        throw new IllegalArgumentException("El número es demasiado grande");
    }
    List<Integer> pares = new ArrayList<>();
    for (int i = 0; i <= numero; i += 2) {
        pares.add(i);
    }
    return pares;
}
```

Listing 9: Buena práctica - Lanzar excepción en lugar de retornar null - Método para generar pares

```
public static void main(String[] args) {
    try {
        List<Integer> pares = generarPares(-5);
        System.out.println("Generado con éxito: " + pares);
    } catch (IllegalArgumentException e) {  // Agregado "e"
        System.out.println("Error: " + e.getMessage());
    }

    try {
        List<Integer> pares = generarPares(1001);
        for (Integer par : pares) {
            System.out.println(par);
        }
    } catch (IllegalArgumentException e) {
        System.out.println("Error: " + e.getMessage());
    }

    try {
        List<Integer> pares = generarPares(0);
```

```
        System.out.println("Lista generada correctamente: "
                           + pares);
    } catch (IllegalArgumentException e) {
        System.out.println("Error: " + e.getMessage());
    }
}
```

Listing 10: Buena práctica - Lanzar excepción en lugar de retornar null - Metodo main

2.6 No pasar Null como argumento

Pasar *null* como argumento generará una fuente común de errores en tiempo de ejecución, ya que al igual que retornar *null*, se necesitarán verificaciones para manejarlo explícitamente. Además, dado que muchos lenguajes no gestionan bien *null*, es mejor evitarlo siempre que sea posible.

2.6.1. Ejemplo

Para entender la diferencia entre pasar *null* y manejarlo correctamente, se abordará un ejemplo donde se imprime la longitud de una cadena:

- En el **código con mala práctica**, el método calcula la longitud sin verificar si la cadena es *null*, provocando un *NullPointerException* si se pasa una cadena nula.

```
public static void imprimirLongitud(String texto) {
    System.out.println("La longitud del texto es: " + texto.length());
}

public static void main(String[] args) {
    imprimirLongitud(null);
}
```

Listing 11: Mala práctica - Pasar argumento null

- En contraste, el **código con buena práctica** verifica si la cadena es *null* antes de calcular su longitud, lanzando una excepción o devolviendo un valor predeterminado si la cadena es nula.

```
public static void imprimirLongitud(String texto) {
    if (texto == null) {
        throw new IllegalArgumentException("El texto no puede ser null");
```

```
    }
    System.out.println("La longitud del texto es: " + texto.length());
}

public static void main(String[] args) {
    try {
        imprimirLongitud(null);
    } catch (IllegalArgumentException e) {
        System.out.println("Error: " + e.getMessage());
    }
}
```

Listing 12: Buena práctica - No pasar argumento null - Excepción

```
public static void imprimirLongitud(String texto) {
    if (texto == null) {
        texto = ""; // Valor predeterminado
    }
    System.out.println("La longitud del texto es: " + texto.length());
}

public static void main(String[] args) {
    // No lanza excepción, maneja el caso null
    imprimirLongitud(null);
}
```

Listing 13: Buena práctica - No pasar argumento null - Valor predeterminado

3. Bibliografía y Recomendaciones

[1] R. C. Martin, *Clean code: A handbook of agile software craftsmanship*, Prentice Hall, 2009.

[2] R. C. Martin, *Clean Architecture: A Craftsman's Guide to Software Structure and Design*, 1st ed., Prentice Hall, 2017.

[3] L. Vogel, "What is software testing with unit and integration tests,"2021. [Online]. Available: https://www.vogella.com/tutorials/SoftwareTesting/article.html.

[4] L. Vogel, "JUnit 5 tutorial - Learn how to write unit tests,"2021. [Online]. Available: https://www.vogella.com/tutorials/JUnit/article.html.

[5] S. Bechtold, B. Brannen, I. Philipp, and M. Stein, *JUnit 5 User Guide*, 2023. [Online]. Available: https://junit.org/junit5/docs/current/user-guide/.

[6] E. Holzman, "The worst computer bugs in history: The Ariane 5 disaster,"BugSnag Blog, June 5, 2023. [Online]. Available: https://www.bugsnag.com/blog/bug-day-ariane-5-disaster/.

[7] "What are Assertions in Unit Testing, and How Does It Help with Unit Testing?,"WeTest Blog. [Online]. Available: https://www.wetest.net/blog/what-are-assertions-in-unit-testing-and-how-does-it-help-with-unittesting-517.html .