



**U N I V E R S I D A D
D E L A F R O N T E R A**

FACULTAD DE INGENIERÍA Y CIENCIAS

Departamento de Ciencias de la Computación e Informática

RT-POO-02

Unit Test - Error handler

Profesor

Dr. Samuel Sepúlveda Cuevas

Ayudante

Jesús Tapia Martín

Módulo

ICC490

Temuco, Chile.

Abril, 2024

Índice

1	Introducción	3
2	Pruebas Unitarias	4
2.1	Importancia de las pruebas unitarias	4
2.2	Flexibilidad de las pruebas unitarias	4
2.3	Pruebas limpias	5
2.4	Un «Assert» por test	5
2.4.1	Ejemplo	5
2.5	Un concepto por prueba	6
2.5.1	Ejemplo	7
3	Manejo de Excepciones	9
3.1	Utilizar excepciones en lugar de retornos	9
3.1.1	Ejemplo	9
3.2	Escribir el Bloque Try-Catch Primero	11
3.3	Proporcionar contexto con excepciones	11
3.4	No retornar null	11
3.4.1	Ejemplo	11
3.5	No pasar Null como argumento	13
3.5.1	Ejemplo	14
4	Ejercicio Resuelto - Joaquín Faundez [3]	16
5	Desafío - Joaquín Faúndez - Jesús Tapia [9]	18
5.1	Instrucciones	18
5.2	Consejos Útiles	18
6	Conclusión	20
	Bibliografía	21

1 Introduccción

En el desarrollo de software es común encontrarse con una estrecha relación entre dos herramientas fundamentales: «**Pruebas Unitarias**» y el «**Manejo de Excepciones**».

Las «**Pruebas Unitarias**» permiten verificar si el comportamiento de un módulo del código funciona correctamente. En situaciones excepcionales en donde ocurren errores inesperados y los test fallan, se recurre al «**Manejo de Excepciones**» para controlar y gestionar estas circunstancias, permitiendo que el software continúe su flujo normal de ejecución sin interrupciones.

Al combinar el uso de estas herramientas, se cuenta ya con bases sólidas que aseguran que el código sea robusto. Estas herramientas permiten detectar y corregir errores potenciales antes de que afecten a los usuarios finales, lo que contribuye significativamente al proceso de desarrollo del software.

Cuando se trabaja con pruebas unitarias y el manejo de excepciones, el uso de buenas prácticas toma un papel crucial para sacar todo el potencial, fiabilidad de las pruebas y del manejo de situaciones excepcionales.

Como objetivo de este reporte se abordará:

- Buenas prácticas
- Pruebas Limpias
- Diseño de un buen caso de pruebas
- Prevención de errores.
- Identificación de situaciones excepcionales.
- Relación entre pruebas unitarias y manejo de excepciones.

Todo el código presente en los ejemplos del reporte, se encuentra disponible en el siguiente repositorio Github <https://github.com/JesusTapiaMartin/RT-P00-01.git> [4]

2 Pruebas Unitarias

Las pruebas unitarias son casos automatizados que verifican el comportamiento de módulos pequeños del código.

Tienen como función principal el detectar errores y defectos en el código de manera temprana en el ciclo de desarrollo, lo que permite una corrección oportuna reduciendo el costo de reparación.

Dentro del desarrollo guiado por pruebas (**Test Driven Development, TDD**) aparecen tres leyes a seguir:

1. No escribirás código de producción sin antes escribir un test que falle.
2. No escribirás más de un test unitario suficiente para fallar, y no compilar es fallar.
3. No escribirás más código del necesario para hacer pasar el test.

2.1. Importancia de las pruebas unitarias

La ausencia de pruebas en un proyecto limita la capacidad para garantizar que los cambios que se realizan funcionen como se espera. Por lo tanto, el código de prueba y el código de producción tienen la misma importancia en el desarrollo de nuestro software.

Por esto es que la calidad de las pruebas unitarias representará la facilidad con la que se añaden cambios futuros.

«El código de prueba es tan importante como el código de producción» [1, página 124]

2.2. 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, página 124]

2.3. 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, página 123]

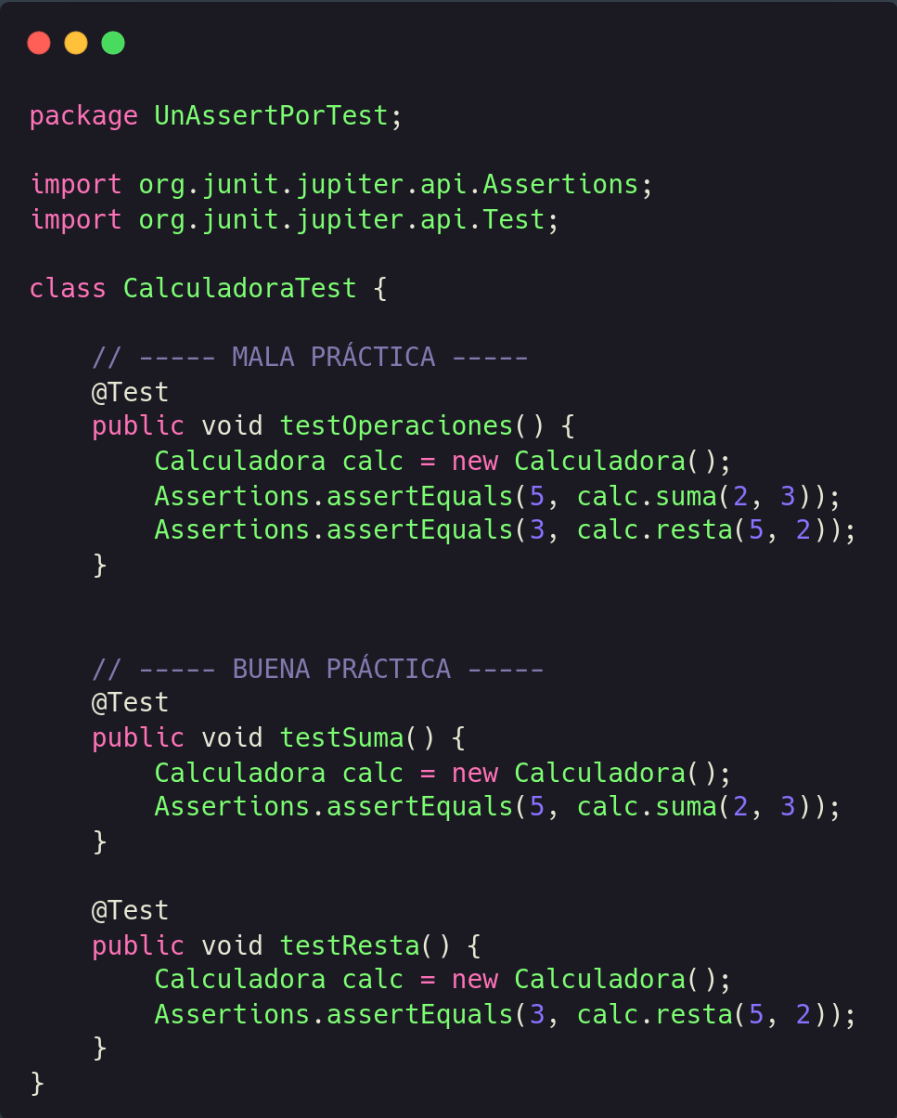
2.4. Un «Assert» por test

Cada prueba debería contener únicamente una declaración **«assert»**. En muchas ocasiones es posible dividir una prueba que tiene muchos «asserts» en varias pruebas. Con esto se facilita la comprensión de lo que se está evaluando en cada test. Sin embargo, si es necesario se puede agregar más de un «assert» por prueba, siempre y cuando estén relacionadas con el mismo concepto a probar.

2.4.1. Ejemplo

Para entender la diferencia entre un enfoque de prueba con múltiples **asserts** y la estrategia de pruebas separadas con distintos conceptos, consideraremos un ejemplo en el que se obtiene una lista de números pares menores o iguales a un número dado.

- En el caso del código con mala práctica, se encuentra un único método con múltiples **assert**. Esto dificulta el entendimiento y la identificación de qué prueba falla en caso de un error.
- Por otro lado, en el código con buena práctica, cada prueba se centra en verificar una única operación de la calculadora. Esto hace que las pruebas sean más fáciles de entender, depurar e identificar cuál falla en caso de error.



```

package UnAssertPorTest;

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

class CalculadoraTest {

    // ----- MALA PRÁCTICA -----
    @Test
    public void testOperaciones() {
        Calculadora calc = new Calculadora();
        Assertions.assertEquals(5, calc.suma(2, 3));
        Assertions.assertEquals(3, calc.resta(5, 2));
    }

    // ----- BUENA PRÁCTICA -----
    @Test
    public void testSuma() {
        Calculadora calc = new Calculadora();
        Assertions.assertEquals(5, calc.suma(2, 3));
    }

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

```

Figura 1: Mala y Buena práctica

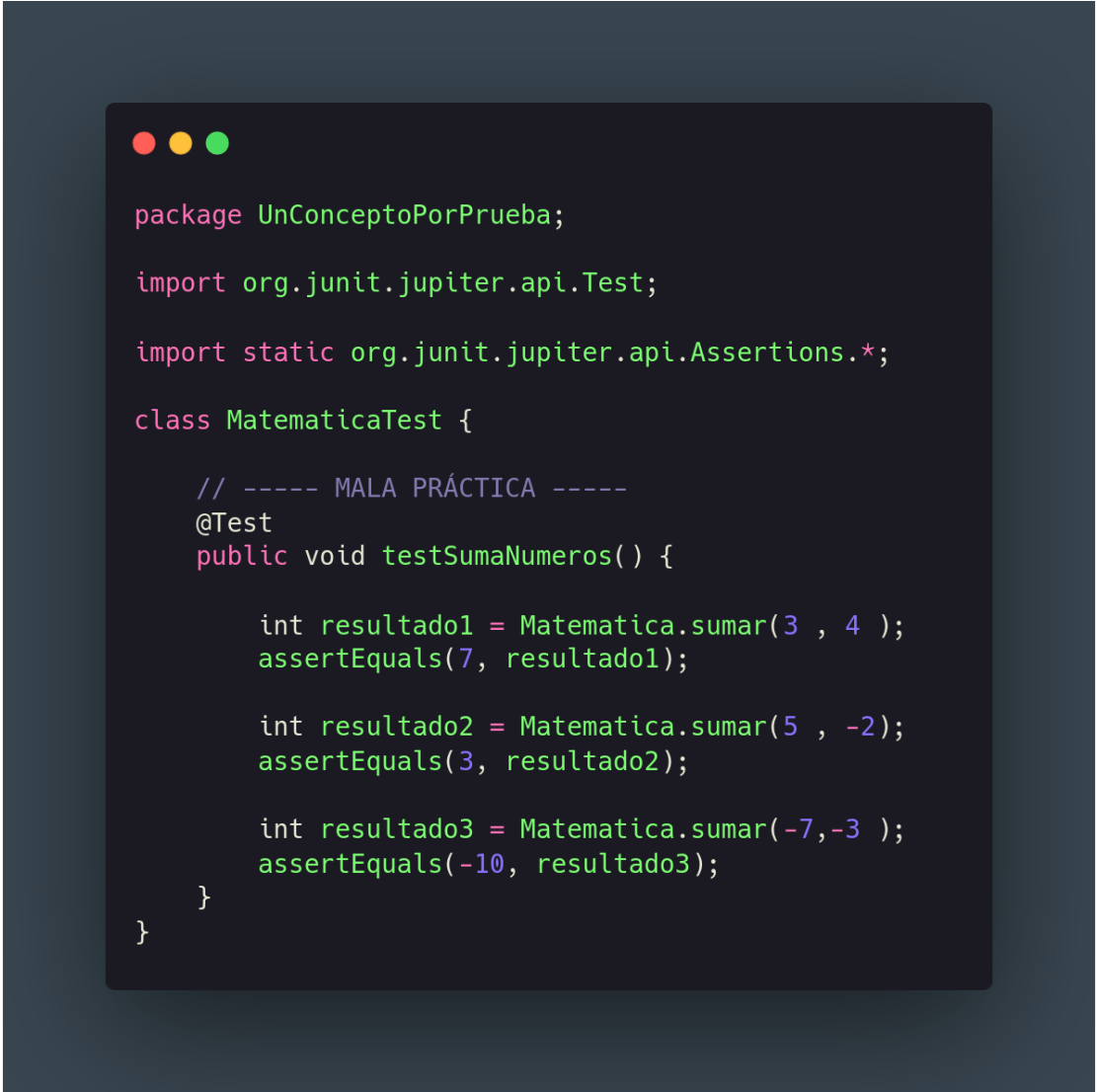
2.5. Un concepto por prueba

Cada test debería evaluar un único concepto, evitando tener pruebas que evalúen múltiples módulos. Por lo tanto, en lugar de tener una única prueba que abarque varios conceptos, es recomendable dividirla en pruebas independientes.

2.5.1. Ejemplo

Para ver la diferencia un test con varios conceptos y el enfoque de un test para cada concepto, se considerará el ejemplo de una clase «**Calculadora**» que contiene un método para realizar sumas entre dos números, incluyendo positivos, negativos y combinaciones de ambos.

- En el código con mala práctica, se intenta cubrir múltiples casos en una sola prueba, resultando un test sobrecargado con lógica y assert. Debido a esto, se dificulta el seguimiento de qué aspecto específico se está probando, además de que cualquier error en la prueba será difícil de corregir debido a la falta de modularidad.

A screenshot of a code editor with a dark background. At the top left, there are three colored window control buttons (red, yellow, green). The code is written in Java and shows a package declaration, imports for JUnit, and a class named 'MatematicaTest'. Inside the class, there is a single test method 'testSumaNumeros()' annotated with '@Test'. This method contains three separate assertions, each testing a different sum: 3+4=7, 5+(-2)=3, and -7+(-3)=-10. A comment '// ----- MALA PRÁCTICA -----' is placed above the first assertion to indicate that this is a bad practice because multiple cases are tested in a single method.

```
package UnConceptoPorPrueba;

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

class MatematicaTest {

    // ----- MALA PRÁCTICA -----
    @Test
    public void testSumaNumeros() {

        int resultado1 = Matematica.sumar(3 , 4 );
        assertEquals(7, resultado1);

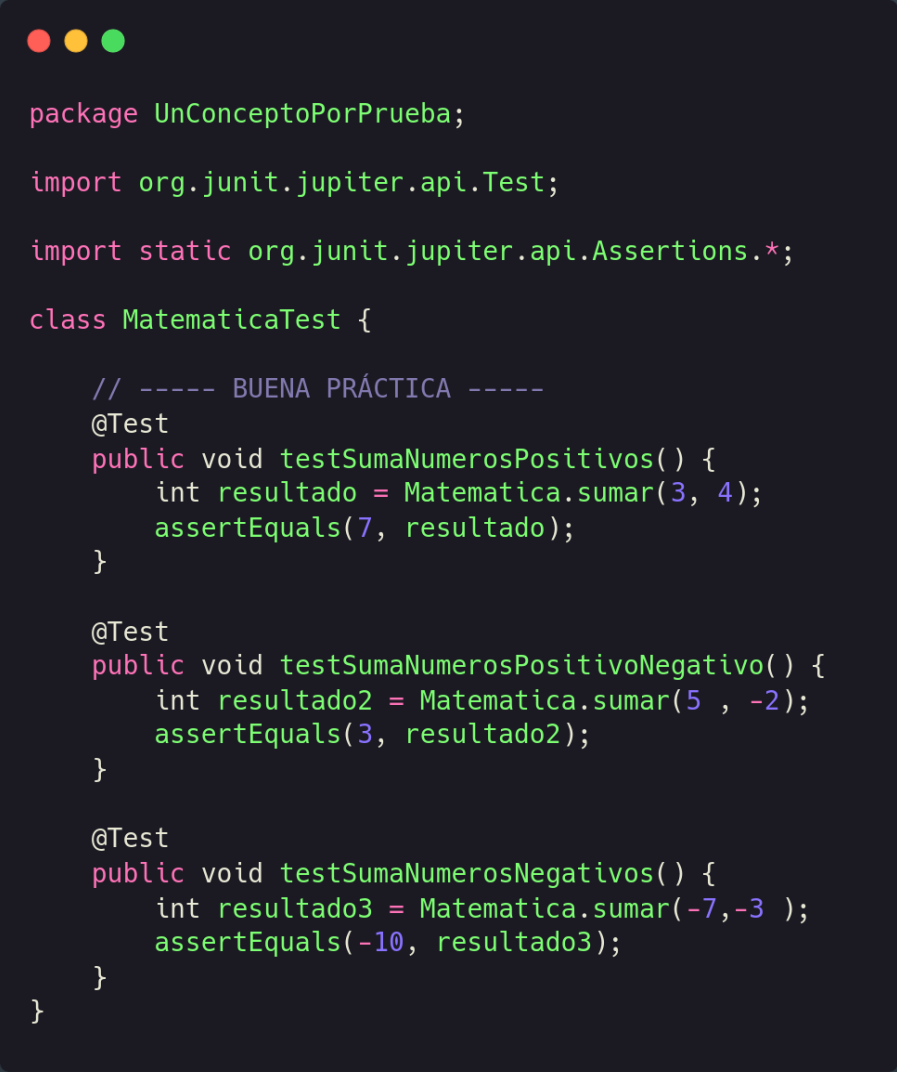
        int resultado2 = Matematica.sumar(5 , -2);
        assertEquals(3, resultado2);

        int resultado3 = Matematica.sumar(-7,-3 );
        assertEquals(-10, resultado3);

    }
}
```

Figura 2: Mala práctica

- En el código con buena práctica, cada prueba se enfoca en probar un solo concepto específico de la suma, ya sea para números positivos, negativos o una combinación de ambos.

A screenshot of a code editor with a dark background and light-colored text. The code is in Java and represents a unit test class. It includes package declarations, imports for JUnit, and a class named 'MatematicaTest' containing three test methods. The methods are annotated with '@Test' and use 'assertEquals' to verify the results of a 'sumar' method. The first method tests positive numbers (3 + 4 = 7), the second tests a positive and a negative number (5 + -2 = 3), and the third tests two negative numbers (-7 + -3 = -10). A comment '// ----- BUENA PRÁCTICA -----' is placed above the first test method.

```
package UnConceptoPorPrueba;

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

class MatematicaTest {

    // ----- BUENA PRÁCTICA -----
    @Test
    public void testSumaNumerosPositivos() {
        int resultado = Matematica.sumar(3, 4);
        assertEquals(7, resultado);
    }

    @Test
    public void testSumaNumerosPositivoNegativo() {
        int resultado2 = Matematica.sumar(5, -2);
        assertEquals(3, resultado2);
    }

    @Test
    public void testSumaNumerosNegativos() {
        int resultado3 = Matematica.sumar(-7, -3);
        assertEquals(-10, resultado3);
    }
}
```

Figura 3: Buena práctica

3 Manejo de Excepciones

3.1. Utilizar excepciones en lugar de retornos

En los métodos la utilización de retornos en lugar de excepciones solo llevará a una combinación confusa entre la lógica del método y la del manejo de errores. Lo que dificultará la comprensión y el mantenimiento del código, al quedar ambas responsabilidades entrelazadas entre sí.

En base a esto, es preferible que los métodos no manejen explícitamente los errores, sino que tengan la capacidad de lanzar excepciones cuando encuentren un error con un **try-catch**, lo que permitirá que el método maneje la excepción solo cuando sea necesario, sin tener que verificar cada vez que se invoca el método.

3.1.1. Ejemplo

Para entender la diferencia entre el manejo de errores mediante retornos y excepciones, se considera un ejemplo en el que se divide dos números. En el primer enfoque, se emplean retornos para indicar un error, mientras que en el segundo se utilizan excepciones.

- En el caso del código con mala práctica, antes de realizar la división se verifica si el divisor es igual a 0. En caso afirmativo, se imprime un mensaje de error y se le asigna un valor **(-1)** para indicar el error. Luego, se verifica que el resultado sea diferente de **(-1)**.

En este caso, no se lanzan excepciones cuando el divisor es 0, lo que significa que la ejecución del programa no se detendrá en caso de error. Además, es necesario verificar el resultado después de cada llamada al método.

- En contraste, en el código con buena práctica, el método lanza una excepción **«ArithmeticException»** cuando el divisor es igual a 0. Esta excepción es capturada y manejada adecuadamente dentro de un bloque **try-catch**.

Este enfoque separa claramente la lógica del manejo de errores de la lógica principal del método, además de permitir describir y especificar la causa de la excepción.

```

package UtilizarExcepcionesNoRetornos;

public class DivisionMalaPractica {

    // ===== MÉTODO MAIN =====
    public static void main(String[] args) {
        realizarDivision(10, 0);
    }

    // ===== MÉTODO REALIZAR DIVISIÓN =====
    public static void realizarDivision(int dividendo, int divisor)
    {
        int resultado;
        if (divisor == 0) {
            System.out.println(" Error, divisor igual a 0");
            resultado = -1;
        } else {
            resultado = dividendo / divisor;
        }

        if (resultado != -1) {
            System.out.println(" La división es: " + resultado);
        }
    }
}

```

Figura 4: Mala práctica

```

package UtilizarExcepcionesNoRetornos;

public class DivisionBuenaPractica {
    // ===== MAIN =====
    public static void main(String[] args) {
        realizarDivision(10, 0);
    }

    // ===== REALIZAR DIVISIÓN =====
    public static void realizarDivision(int dividendo, int divisor) {
        try {
            int resultado = dividendo / divisor;
            System.out.println("La división es: " + resultado);
        } catch (ArithmeticException e) {
            System.out.println("Error, división por cero : " + "\"" + e.getMessage() + "\"");
        }
    }
}

```

Figura 5: Buena práctica

3.2. Escribir el Bloque Try-Catch Primero

Cuando se está escribiendo un método que puede lanzar excepciones, es una buena práctica comenzar con un bloque `try-catch`.

Con esto se anticipa y planifica el manejo de excepciones desde el inicio, pensando en cómo manejar y cómo se quiere que el método se comporte en caso de que ocurra un error. Finalmente, así se garantiza que el programa siempre se mantenga en un estado consistente y manejable, incluso en situaciones de error.

3.3. Proporcionar contexto con excepciones

Al desarrollar el bloque `catch` en un método, este debe ser capaz de brindar el contexto suficiente para identificar la causa del error. Por lo que las excepciones capturadas deben tener información detallada sobre qué salió mal, como mensajes de error o datos relevantes sobre el estado del programa en la excepción. Esto ayuda con la depuración y resolución de problemas para tomar medidas correctivas adecuadas.

3.4. No retornar null

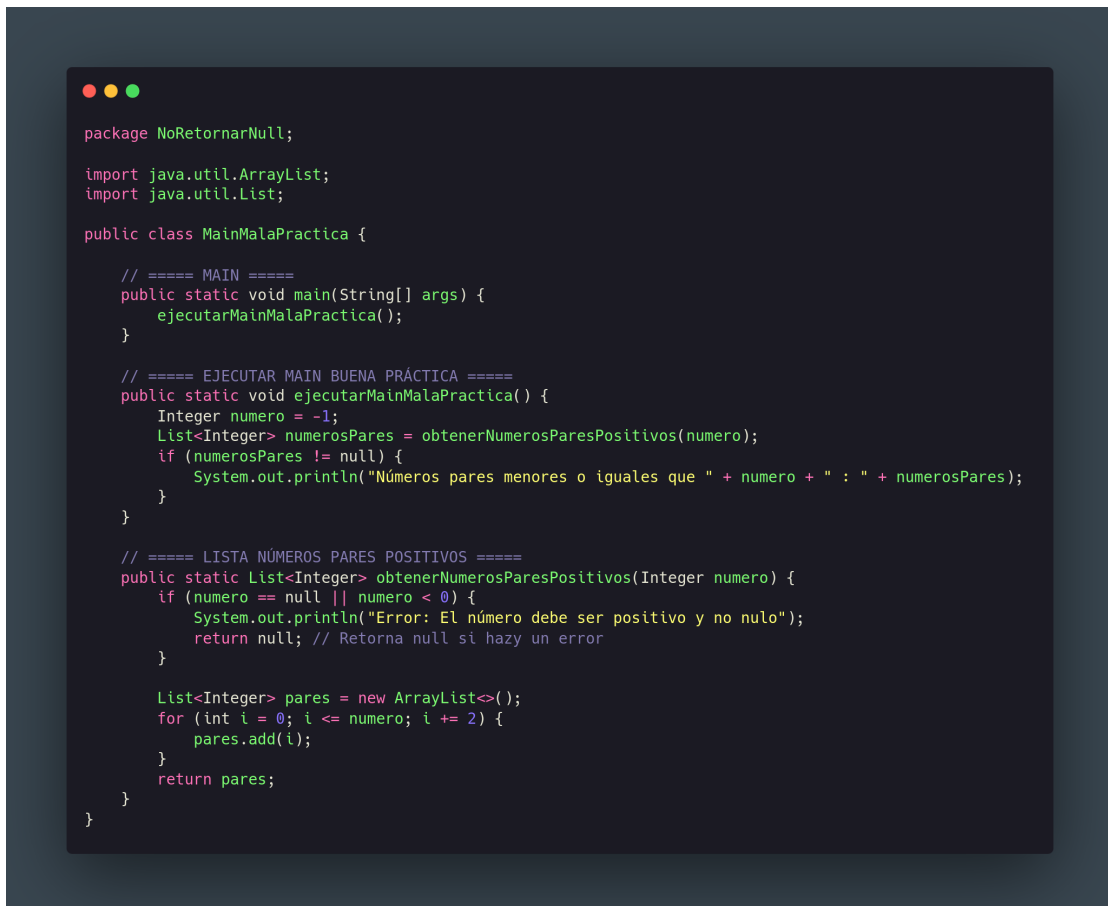
Cuando un método devuelve `null`, puede generar complejidades adicionales en el código cliente al requerir verificaciones adicionales para evitar errores de puntero nulo. Este enfoque aumenta la complejidad y la propensión a errores en la lógica del programa.

En lugar de retornar `null`, es más recomendable considerar otras alternativas, como lanzar una excepción adecuada para manejar situaciones excepcionales, o devolver un objeto especial que represente una condición específica, como una lista vacía o un valor predeterminado, según el contexto y los requisitos del programa.

3.4.1. Ejemplo

Para ilustrar la diferencia entre retornar `null` y adoptar un enfoque que evite retornar `null`, se considera un ejemplo donde se genera una lista de números pares menores o iguales a un número dado.

- En el caso del código con mala práctica, si el número dado es nulo o negativo, se emite un mensaje de error y se devuelve null. Esto conlleva la necesidad de realizar múltiples verificaciones adicionales para manejar el caso en que se retorne null, lo que puede aumentar la complejidad y la propensión a errores en la lógica del programa.
- Por otro lado, en el código con buena práctica si el número dado es negativo se lanza una excepción del tipo «**IllegalArgumentException**». Esto permite señalar claramente el error y manejar la excepción de manera apropiada. En este enfoque no se retorna **null**, lo que elimina la necesidad de verificaciones adicionales en el código y simplifica tanto la lógica como la estructura del programa.



```
package NoRetornarNull;

import java.util.ArrayList;
import java.util.List;

public class MainMalaPractica {

    // ===== MAIN =====
    public static void main(String[] args) {
        ejecutarMainMalaPractica();
    }

    // ===== EJECUTAR MAIN BUENA PRÁCTICA =====
    public static void ejecutarMainMalaPractica() {
        Integer numero = -1;
        List<Integer> numerosPares = obtenerNumerosParesPositivos(numero);
        if (numerosPares != null) {
            System.out.println("Números pares menores o iguales que " + numero + " : " + numerosPares);
        }
    }

    // ===== LISTA NÚMEROS PARES POSITIVOS =====
    public static List<Integer> obtenerNumerosParesPositivos(Integer numero) {
        if (numero == null || numero < 0) {
            System.out.println("Error: El número debe ser positivo y no nulo");
            return null; // Retorna null si hay un error
        }

        List<Integer> pares = new ArrayList<>();
        for (int i = 0; i <= numero; i += 2) {
            pares.add(i);
        }
        return pares;
    }
}
```

Figura 6: Mala práctica

```

package NoRetornarNull;

import java.util.ArrayList;
import java.util.List;

public class MainBuenaPractica {

    // ===== MAIN =====
    public static void main(String[] args) {
        ejecutarMainBuenaPractica();
    }

    // ===== EJECUTAR MAIN BUENA PRÁCTICA =====
    public static void ejecutarMainBuenaPractica() {
        int numero = -1;

        try {
            List<Integer> numerosPares = obtenerNumerosParesPositivos(numero);
            System.out.println("Números pares menores o iguales que " + numero + " : " + numerosPares);
        } catch (IllegalArgumentException e) {
            System.out.println("Error:aaa " + e.getMessage());
        }
    }

    public static List<Integer> obtenerNumerosParesPositivos(int numero) {
        if (numero <= 0) {
            throw new IllegalArgumentException("El naaaaúmero debe ser positivo");
        }

        try {
            List<Integer> pares = new ArrayList<>();
            for (int i = 0; i <= numero; i += 2) {
                pares.add(i);
            }
            return pares;
        } catch (Exception e) {
            throw new RuntimeException("Error: ", e);
        }
    }
}

```

Figura 7: Buena práctica

3.5. No pasar Null como argumento

Pasar `null` como argumento representa una fuente común de errores en tiempo de ejecución, como las excepciones de «`NullPointerException`», que interrumpen el flujo normal de ejecución del programa y pueden resultar difíciles de rastrear y corregir.

Evitar el paso de «`null`» como argumento simplifica el desarrollo del código, al eliminar la necesidad asociada al tratamiento de valores `null`.

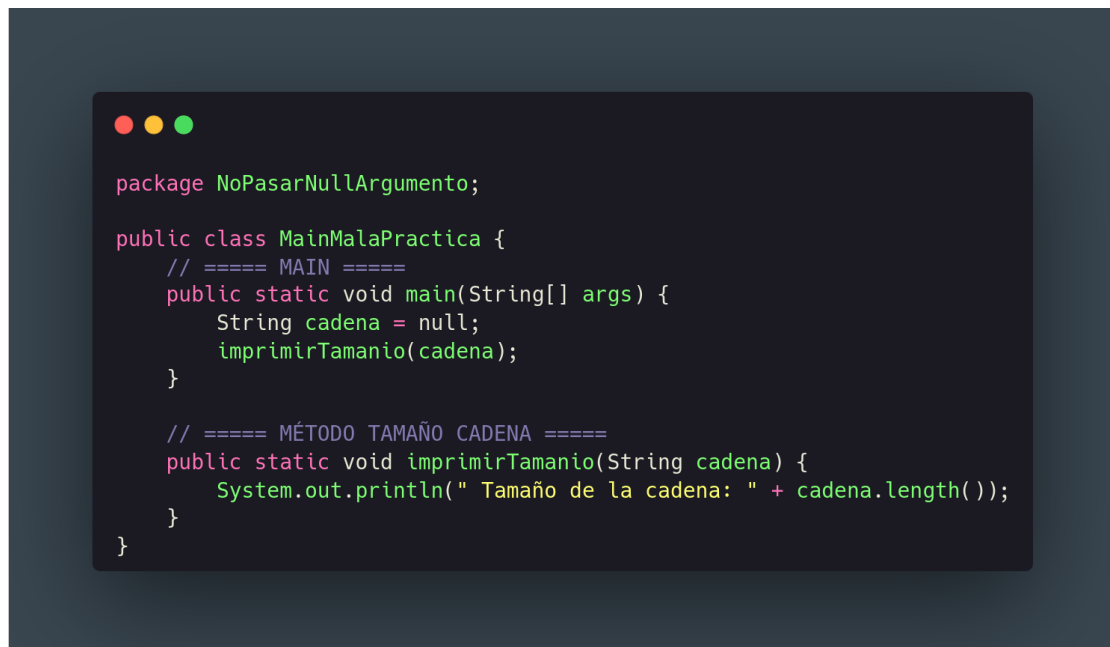
En la mayoría de los lenguajes de programación, no existe una forma efectiva de manejar adecuadamente un `null` que se pase accidentalmente como argumento.

Por lo tanto, es preferible evitar pasar `null` como argumento siempre que sea posible.

3.5.1. Ejemplo

Para ver la diferencia entre pasar `null` como argumento y manejar adecuadamente la posibilidad de que no se pase un `null`, se considerará un ejemplo donde se imprime la longitud de una cadena.

- En el caso del código con mala práctica no se considera ni verifica si la cadena es nula antes de intentar obtener su longitud. Esto conduce a una excepción de «**NullPointerException**» cuando se ejecuta el programa.
- Por otro lado, en el código con buena práctica se verifica si la cadena es nula antes de intentar obtener su longitud. Si la cadena es «**null**», se maneja devolviendo un valor predeterminado o lanzando una excepción específica, según los requisitos del programa. Este enfoque evita la excepción de «**NullPointerException**» lo que resulta en un código más robusto y seguro.


A screenshot of a code editor with a dark background and light-colored text. The code is in Java and shows a package declaration, a class definition, and two methods. The first method, `main`, sets a `String` variable `cadena` to `null` and calls `imprimirTamano(cadena)`. The second method, `imprimirTamano`, prints the length of the `cadena` parameter using `cadena.length()`. The code is as follows:

```
package NoPasarNullArgumento;

public class MainMalaPractica {
    // ===== MAIN =====
    public static void main(String[] args) {
        String cadena = null;
        imprimirTamano(cadena);
    }

    // ===== MÉTODO TAMAÑO CADENA =====
    public static void imprimirTamano(String cadena) {
        System.out.println(" Tamaño de la cadena: " + cadena.length());
    }
}
```

Figura 8: Mala práctica



```
package NoPasarNullArgumento;

public class MainBuenaPractica {

    // ===== MAIN =====
    public static void main(String[] args) {
        String cadena = null;
        imprimirTamano(cadena);
    }


    // ===== MÉTODO TAMAÑO CADENA =====
    public static void imprimirTamano(String cadena) {
        try {
            if (cadena == null) {
                throw new IllegalArgumentException("La cadena no puede ser null");
            }
            System.out.println("Tamaño de la cadena: " + cadena.length());
        } catch (IllegalArgumentException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

Figura 9: Buena práctica

4 Ejercicio Resuelto - Joaquín Faundez [3]

Elaborar tres métodos para realizar la lectura de la información ingresada por el usuario, considerando que los métodos deberán ser sensibles a excepciones en el ingreso. En caso excepciones, se debe mostrar por consola un mensaje con la excepción capturada

1. leerEntero: Solicita al usuario que ingrese un número entero. Retorna la lectura del valor ingresado.
2. leerString: Solicita al usuario que ingrese un carácter de tipo «String». Retorna la lectura del «String» ingresado.
3. stringToInt: Toma como parámetro un dato de tipo «String» e intenta realizar la conversión a un número entero. Retornar el string ingresado en formato de entero.



```
public class LecturaDatos {  
    // ----- VARIABLES -----  
    public static Scanner leer = new Scanner(System.in);  
  
    // ----- LEER ENTERO -----  
    public static int leerEntero() {  
        return leer.nextInt();  
    }  
  
    // ----- LEER STRING -----  
    public static String leerString() {  
        return leer.next();  
    }  
  
    // ----- STRING TO INT -----  
    public static int stringToInt(String texto) {  
        return //String en entero;  
    }  
}
```

Figura 10: Identificación de errores y manejo de excepciones - Joaquín Faúndez

El código resuelto se encuentra en el siguiente repositorio de GitHub <https://github.com/jfaundez07/Ayudantia04.04.git> [8]

5 Desafío - Joaquín Faúndez - Jesús Tapia

[9]

Desarrollar un programa que gestione el inventario de productos de una tienda. El inventario se debe manejar en una matriz de la forma [fila][columna], donde la columna 0 representa el nombre del producto y la columna 1 representa la cantidad en stock.

El programa debe ser capaz de:

1. Agregar un producto : Añadir producto con su nombre y stock.
2. Vender un producto : Seleccionar un producto específico por el usuario, junto con la cantidad requerida.
3. Mostrar inventario : Mostrar los artículos y las cantidades disponibles.

5.1. Instrucciones

- Implementar un método llamado «`buscarProducto()`» que será utilizado en el método «`venderProducto()`». Además, el programa debe ser capaz de responder ante posibles excepciones en el ingreso de datos y otros escenarios de error.
- Implemente pruebas unitarias para verificar el correcto funcionamiento de los métodos.
- El método «`main()`» deberá únicamente lanzar el menú principal.
- Todo el desarrollo debe ser almacenado en un repositorio en GitHub, con una rama de «Desarrollo» donde se trabajará en la actividad, y una rama «main» donde se encontrará la versión final. El archivo README debe contener los nombres de los integrantes del equipo (en caso de trabajar en grupo) y una breve descripción del funcionamiento de los métodos implementados.

5.2. Consejos Útiles

- Prestar atención a cómo se manejarán los tipos de datos al momento de registrar el stock y al solicitar una cantidad por el usuario (el tipo de dato a

comparar debe ser el mismo).

- Para facilitar la elaboración de pruebas unitarias, desarrollar métodos que tomen como parámetro a la matriz que almacena el stock, en vez de modificar directamente una variable global.
- El método «`agregarProducto()`» solo debe cumplir la tarea de modificar la matriz de stock con el nuevo producto, mas no debe contener validaciones necesarias y/o manejo de excepciones. Esto último debe ser considerado por ejemplo en un método «`menuAgregarProducto()`»

El código resuelto del desafío se encuentra en el siguiente repositorio GitHub <https://github.com/jfaundez07/Actividad11.04.git> [9].

6 Conclusión

En este informe se presentó como la correcta implementación entre «**Pruebas Unitarias**» y «**Manejo de excepciones**» representa una de las mejores aliadas para desarrollar software robusto y resistente a fallos. Con las **pruebas unitarias** se observó cómo estas verifican el comportamiento del código. Por otro lado, con el **manejo de excepciones** se vio como permiten controlar y gestionar aquellas situaciones excepcionales que surgan durante el flujo del programa, dando la seguridad de que el software continúe su ejecución sin interrupciones.

Se revisó cómo al diseñar casos de pruebas con buenas prácticas, considerando tanto las situaciones favorables como las desfavorables, hace que estas puedan cubrir todos los escenarios posibles, dando una cobertura completa del código en su mantención y detección temprana de errores.

Con la buenas prácticas en la prevención de errores, se evidenció como se identifican y corrigen errores potenciales antes de que afecten en el desarrollo.

Es importante destacar que lo visto en este informe solo brinda una introducción a los principios de las «**Pruebas Unitarias**» y el «**Manejo de Excepciones**», pero cada uno de estos corresponde a todo un área dentro de la programación, habiendo equipos dedicados solo a estos.

Las pruebas unitarias con **JUnit5** ofrecen una amplia gama de herramientas que sacan todo el potencial del proceso de testing. Entre estas herramientas se encuentran el Ciclo de Vida de JUnit, test condicionales, clases tests anidadas, pruebas parametrizadas, tagging tests, inyección de dependencias, tests condicionales con Assumptions, y el uso del framework **Mockito** para crear objetos ficticios (mocks).

Finalmente, queda abierta la invitación a profundizar las herramientas que da JUnit, con el fin de enriquecer el testing.

Bibliografía y Recomendaciones

- [1] Martin, R. C. (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design* (1st ed.). Prentice Hall.
- [2] Martin, R. C. (2009). *Clean code: a handbook of agile software craftsmanship* Prentice Hall.
- [3] Faúndez Concha, J. (2024). Identificación de errores y manejo de excepciones. Universidad de La Frontera.
- [4] Tapia Martin, J. (2024). Repositorio GitHub. <https://github.com/JesusTapiaMartin/RT-P00-01.git>
- [5] Lars Vogel. (2021) . What is software testing with unit and integration tests. <https://www.vogella.com/tutorials/SoftwareTesting/article.html>
- [6] Lars Vogel. (2021) . JUnit 5 tutorial - Learn how to write unit tests. <https://www.vogella.com/tutorials/JUnit/article.html>
- [7] Bechtold-Brannen-Brannen-Merdes-Philipp-Rancourt-Stein. JUnit 5 User Guide. <https://junit.org/junit5/docs/current/user-guide/>
- [8] Faúndez Concha, J. (2024). Universidad de La Frontera. <https://github.com/jfaundez07/Ayudantia04.04.git>
- [9] Faúndez Concha, J - Tapia Martin, J. (2024). Universidad de La Frontera. <https://github.com/jfaundez07/Actividad11.04.git>