



**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-01

Pruebas Unitarias y Manejo de Excepciones

Profesor

Dr. Samuel Sepúlveda Cuevas

Ayudante

Jesús Tapia Martín

Módulo

ICC490

Temuco, Chile.
Abril, 2024

Indice

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	6
3	Manejo de Excepciones	8
3.1	Utilizar excepciones en lugar de retornos	8
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	Desafío [3]	15
5	Conclusión	17
	Bibliografía	18

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, contamos ya con bases sólidas que aseguran que nuestro 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 y fiabilidad de nuestras pruebas y del manejo de situaciones excepcionales, por lo que como objetivo de este informe se abordará:

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

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 y reduce 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 de prueba.

2.4.1 Ejemplo

Para ver la diferencia entre tener un enfoque de prueba con múltiples afirmaciones y la estrategia de tener pruebas separadas con distintos conceptos, consideremos un ejemplo donde se obtendrá una lista de números pares menores o iguales a un número dado.

- En el código con mala práctica, tenemos un único método con múltiples «assert». Esto solo resulta en un difícil entendimiento e identificación si la prueba falla.
- En el código con buena práctica, cada prueba se enfoca en probar una única operación de la Calculadora, lo que hace que las pruebas sean más fáciles de entender y depurar, además de identificar cual falla.



```

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));
    }
}

```

Figure 1: Mala y Buena práctica

2.5 Un concepto por prueba

Cada test debe evaluar un único concepto, por lo que no se deberían tener una prueba que evalúe múltiples módulos, ya que esto solo dificultaría el seguimiento de qué sección del test evalúa cada aspecto. Por lo tanto, en lugar de tener una única prueba que evalúe varios conceptos, es recomendado dividirla en pruebas independientes.

2.5.1 Ejemplo

Para ver la diferencia un test con varios conceptos dentro y el enfoque de un test para cada concepto, consideremos un ejemplo de una clase «Calculadora» la cual

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, esto solo genera un test que está sobrecargado con lógica y «assert» de diferentes escenarios. Esto solo dificulta el seguimiento de qué aspecto específico de la funcionalidad se está probando. Además de que cualquier error o falla en la prueba puede ser difícil de corregir debido a la falta de modularidad.
- 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.

Esto hace que las pruebas sean fáciles de entender y mantener, además de poder corregir e identificar de manera precisa cualquier error.



```
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);
    }
}
```

Figure 2: Mala práctica



```

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);
    }
}

```

Figure 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 ver la diferencia entre el manejo de errores por medio de retornos y excepciones, consideremos un ejemplo donde se dividirá dos números. En el primer enfoque se usará retornos para indicar un error, mientras que en el segundo usaremos excepciones.

- En el código con mala práctica, antes de realizar una división se verifica si el divisor es igual a 0, si lo es se imprime un mensaje de error y se le asigna un (-1) para indicar el error. Al imprimir se verifica que el resultado sea diferente de (-1).

Acá no se lanza ninguna excepción cuando el divisor es 0, con esto nunca se detendrá la ejecución del programa dando la posibilidad de que el programa continúe cuando ocurre un error. Además de la necesidad de verificar el resultado después de cada llamada al método.

- 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 de manera adecuada dentro del try-catch.

Con este enfoque se separa de forma clara 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);
        }
    }
}

```

Figure 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() + "\"");
        }
    }
}

```

Figure 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 en un método se retorna `null`, surgirá el problema de realizar verificaciones adicionales en el código que llama a este. Este proceso resulta altamente propenso a errores y a complicar la lógica del programa.

En lugar de devolver `null`, es preferible considerar otras opciones como lanzar una excepción o devolver un objeto especial que indique una condición especial, como una lista vacía o un valor predeterminado.

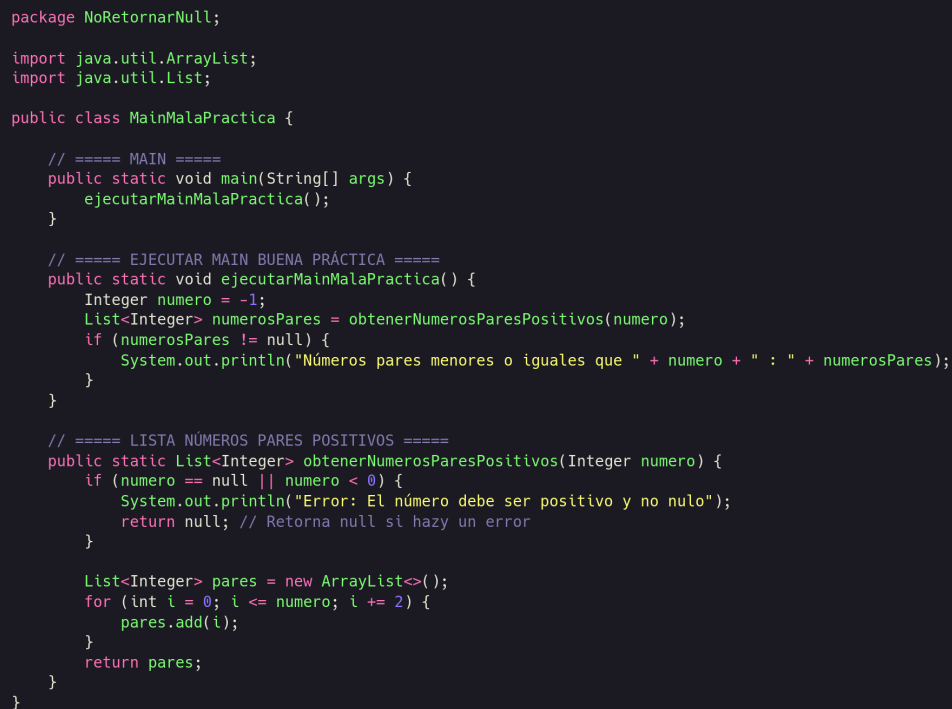
3.4.1 Ejemplo

Para ver la diferencia entre retornar `null` y usar un enfoque sin retornar `null`, consideremos un ejemplo donde se obtendrá una lista de números pares menores o iguales a un número dado.

- En el código con mala práctica, si el número dado es «`null`» o negativo, se

imprime un mensaje de error y se retorna null. Esto obliga a realizar múltiples verificaciones para manejar el caso en que se retorne «null», lo que puede conducir a una lógica de programa más complicada y propensa a errores.

- En el código con buena práctica, si el número dado es negativo, se lanza una excepción «IllegalArgumentException». Esto permite indicar el error y manejar la excepción de manera adecuada. No se devuelve null, lo que elimina la necesidad de verificaciones adicionales en el código, por lo que se simplifica la lógica del programa junto a su tamaño.



```
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;
    }
}
```

Figure 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);
        }
    }
}

```

Figure 7: Buena práctica

3.5 No pasar Null como argumento

Pasar `null` como argumento es una fuente común de errores en tiempo de ejecución, como la excepción de `NullPointerException`, que interrumpe la ejecución del programa y ser difícil de rastrear y corregir.

Evitar pasar «null» como argumento simplifica el desarrollo de nuestro código, ya que se elimina la necesidad asociada al tratamiento de valores «null»

En la mayoría de los lenguajes de programación, no hay una forma efectiva de manejar adecuadamente un `null` que se pase accidentalmente como argumento. Por lo tanto, es mejor evitar pasar `null` como argumento.

3.5.1 Ejemplo

Para ver la diferencia entre pasar null como argumento y manejar correctamente que no se pase un null, consideremos un ejemplo donde se imprimirá el largo de una cadena.

- En el código con mala práctica, no se considera ni verifica que la cadena es nula antes de obtener la longitud. Esto nos llevará a una excepción de «NullPointerException»
- 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, dependiendo de los requisitos del programa. Esto evita la excepción de «NullPointerException» un código más robusto.



```
package NoPasarNullArgumento;

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

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

Figure 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());
        }
    }
}

```

Figure 9: Buena práctica

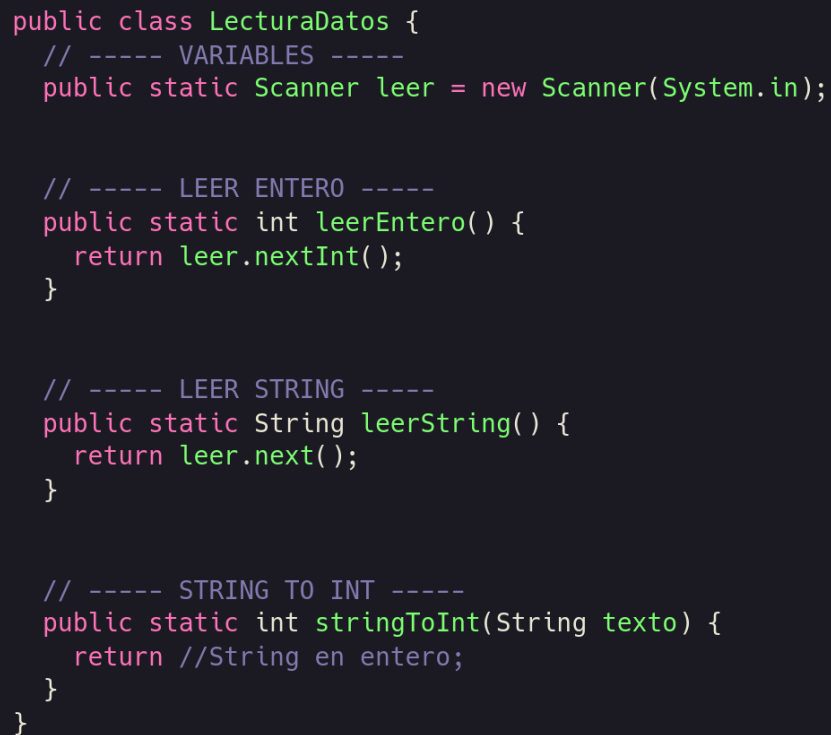
4 Desafío [3]

Elaborar 3 métodos para realizar la lectura de la información ingresada por el usuario utilizando la clase «Scanner». Estos métodos deben ser sensibles a posibles excepciones en el ingreso del tipo de dato. En caso de presentar excepciones, se debe mostrar por consola un mensaje con la excepción capturada

1. leerEntero: Solicita al usuario que ingrese un número entero. Debe retornar la lectura del valor ingresado por el usuario. En caso de ser necesario, debe manejar la excepción y retornar “0”.
2. leerString: Solicita al usuario que ingrese un carácter de tipo «String». Debe retornar la lectura del «String» ingresado por el usuario. En caso de ser necesario, debe manejar la excepción y retornar “0”.
3. stringToInt: Toma como parámetro un dato de tipo «String» e intenta realizar la conversión a un número entero. Debe retornar el string ingresado en formato de entero. En caso de ser necesario debe manejar la excepción y retornar “0”.

Todo el desarrollo debe ser almacenado en un repositorio en GitHub, que tenga

una rama de desarrollo donde se harán cambios y se trabajará la actividad, y una rama main donde se encontrará la version final.



```
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;  
    }  
}
```

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

5 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 vió 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 las 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 JUnit 5 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).

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

Todo el código presente en los ejemplos del informe se encuentra disponible en un repositorio de Github [4]

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/>