Universidad de las Fuerzas Armadas ESPE

Departamento de Ciencias de la Computación - DCCO

Carrera de Ingeniería de Software

Pruebas de Software - NRC 22431

Tarea 2: Test-Driven Development

Luis Ajejandro Andrade Encalada

*I*ng. Luis Alberto Castillo Salinas

05 de Julio 2025

Índice

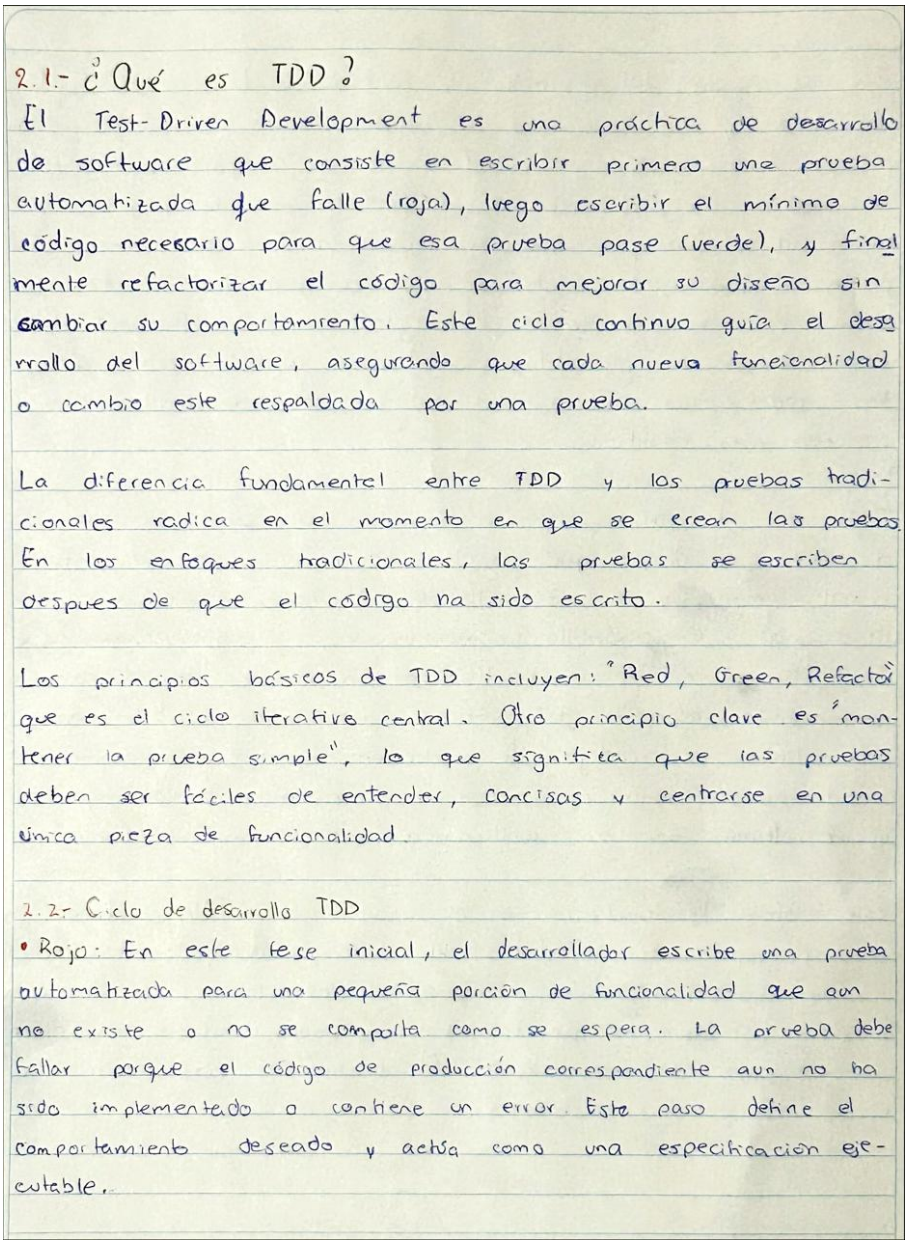
Introducción

EN el mundo del desarrollo de Software, la calidad, la robustez y la capacidad de mantenimiento de código son pilares fundamentales para el éxito de cualquier proyecto. A medida que los sistemas se vuelven mas complejos, la necesidad de enfoques que garanticen la fiabilidad desde las primeras etapas del ciclo de vida del desarrollo se hace cada vez mas relevante. El Test-Driven Development (TDD) emerge como una metodologia agil que aborda estos aspectos, redefiniendo la manera en que los desarrolladores prodeen nuevo software.

TDD no es simplemente una técnica de prueba; es una disciplina de diseño que impulsa la creación de código funcional, limpio. y bien estructurado Al invertir el orden tradicional de desarrollo (primero código, luego pruebas), TDD fuerza a los desarrolladores a pensar en el comportamiento esperado del Software antes de escribir una sola linea de código de producción. Este cambio de mentalidad promueve una comprension mas profunda de los requisitos y, en última instancia conduce a sistemas mas resilientes.

Este informe profundizará en el concepto de TDD, explorando sus principios fundamentales, el ciclo de desarrollo iterativo que lo caracteriza, sus ventajas y desventajas. y las herramientas comunes que facilitan su implementación en diversos lenguajes de programación. El objetivo es proporcionar una visión integral de este enfoque.

Desarrollo



2.1.- ¿Qué es TDD?

El Test-Driven Development es una práctica de desarrollo de software que consiste en escribir primero una prueba automatizada que falle (roja), luego escribir el mínimo de código necesario para que esa prueba pase (verde), y finalmente refactorizar el código para mejorar su diseño sin cambiar su comportamiento. Este ciclo continuo guía el desarrollo del software, asegurando que cada nueva funcionalidad o cambio este respaldada por una prueba.

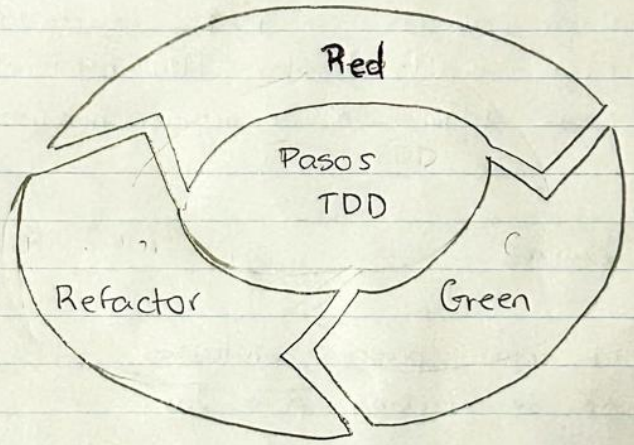
La diferencia fundamental entre TDD y los pruebas tradicionales radica en el momento en que se crean las pruebas. En los enfoques tradicionales, las pruebas se escriben despues de que el código ha sido escrito.

Los principios básicos de TDD incluyen: "Red, Green, Refactor" que es el ciclo iterativo central. Otro principio clave es "mantener la prueba simple", lo que significa que las pruebas deben ser fáciles de entender, concisas y centrarse en una única pieza de funcionalidad.

2.2- Ciclo de desarrollo TDD

- Rojo: En este fase inicial, el desarrollador escribe una prueba automatizada para una pequeña porción de funcionalidad que aun no existe o no se comporta como se espera. La prueba debe fallar porque el código de producción correspondiente aun no ha sido implementado o contiene un error. Este paso define el comportamiento deseado y actúa como una especificación ejecutable.

- Verde: Una vez que la prueba roja ha sido escrita, el desarrollador escribe la cantidad mínima de código de producción necesaria para que esa prueba pase. El objetivo en esta fase no es escribir código perfecto, sino simplemente satisfacer los requisitos de la prueba y lograr que pase.
- Refactor: Con la prueba pasada, el desarrollador puede refactorizar el código. Esto implica mejorar la estructura interna, la legibilidad, la eficiencia y el diseño general de código, sin alterar su comportamiento externo. La suite de pruebas existentes actúa como una red de seguridad, asegurando que los cambios realizados durante la refactorización no introduzcan regresiones.



2.3.- Ventajas y Desventajas del uso de TDD

Ventajas:

- Mayor calidad del Código: TDD asegura que cada componente este debidamente probado y funcione según lo esperado.
- Mejor diseño del código: TDD fomenta un diseño de código más modular, cohesionado y con bajo acoplamiento. Al pensar en como probar el código, los desarrolladores tienden a crear componentes mas pequeños, independientes y faciles de testear.

ESTILO

- Reducción de regresiones: Ejecutar pruebas con frecuencia garantiza que los cambios no rompan las funcionalidades existentes.

Desventajas

- Mantenimiento de pruebas: A medida que el Software crece, la suite de pruebas también lo hace. Mantener las pruebas actualizadas puede convertirse en un desafío si no se gestionan adecuadamente.
- Tiempo inicial adicional: Al principio, escribir pruebas antes del código puede parecer que consume más tiempo. Aunque esta inversión inicial se recupera a largo plazo, puede ser percibida como un obstáculo en proyectos con plazos ajustados.
- Curva de aprendizaje inicial: Adoptar TDD requiere un cambio de mentalidad y la adquisición de nuevas habilidades para los desarrolladores.

2.4: Herramientas comunes

Java:

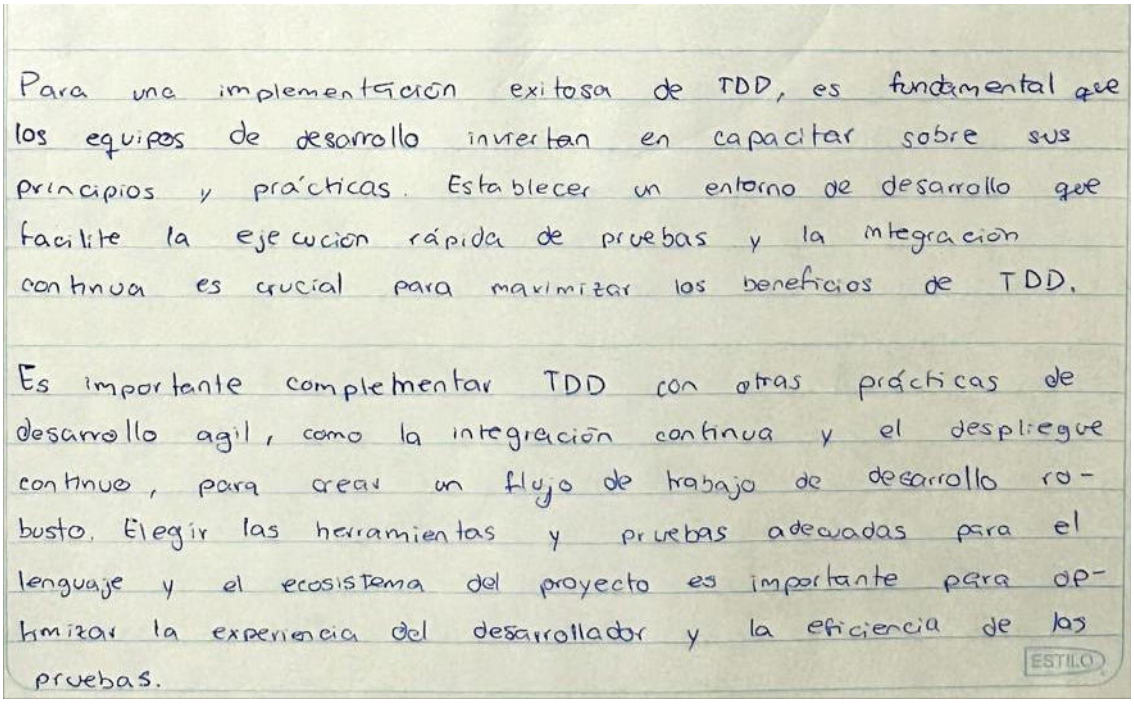
- JUnit: Framework para pruebas unitarias.
- Mockito: Framework de mocking para Java.

JavaScript:

- Jest: Framework para pruebas unitarias.
- Mocha: Framework de pruebas flexible que requiere de una biblioteca de aserciones separada como Chai.

Python:

- pytest: Conocido por su sintaxis concisa y legible.
- mock: Un módulo de unittest que facilita el mocking y la sustitución de objetos en pruebas.


Para una implementación exitosa de TDD, es fundamental que los equipos de desarrollo inviertan en capacitar sobre sus principios y prácticas. Establecer un entorno de desarrollo que facilite la ejecución rápida de pruebas y la integración continua es crucial para maximizar los beneficios de TDD.

Es importante complementar TDD con otras prácticas de desarrollo agil, como la integración continua y el despliegue continuo, para crear un flujo de trabajo de desarrollo robusto. Elegir las herramientas y pruebas adecuadas para el lenguaje y el ecosistema del proyecto es importante para optimizar la experiencia del desarrollador y la eficiencia de las pruebas.

ESTILO

2.5 Ejemplo práctico

Calculadora de Suma: Se sigue el ciclo TDD paso a paso para crear una función simple que sume dos números. Implementación en Python con unittest.

2.5.1 Definición del problema

Se necesita una función que tome dos números como entrada y devuelva su suma.

2.5.2 Rojo

test_calculadora.py

```python
# test_calculadora.py
import unittest # Importamos el módulo unittest para escribir pruebas

# Importamos la función que vamos a probar (aún no existe)
# from calculadora import sumar

class TestCalculadora(unittest.TestCase): # Creamos una clase de prueba que hereda de unittest.TestCase
    def test_sumar_dos_numeros(self):
        # Esta es nuestra primera prueba unitaria.
        # Intentamos sumar 2 y 3, esperando un resultado de 5.
        # En este punto, 'sumar' no está definida, por lo que esta prueba fallará.
        self.assertEqual(sumar(2, 3), 5)
```

Figure 1: Rojo

```
 ❯❯ /home/bowis/Downloads/Tarea2 : python -m unittest test_calculadora.py
E
======================================================================
ERROR: test_sumar_dos_numeros (test_calculadora.TestCalculadora.test_sumar_dos_numeros)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/bowis/Downloads/Tarea2/test_calculadora.py", line 12, in test_sumar_dos_nume
ros
    self.assertEqual(sumar(2, 3), 5)
                     ^^^^^
NameError: name 'sumar' is not defined. Did you mean: 'super'?

----------------------------------------------------------------------
Ran 1 test in 0.001s

FAILED (errors=1)
```

Figure 2: Error en Rojo

2.5.3 Verde

calculadora.py

```python
# calculadora.py
def sumar(a, b):
    # Esta es la implementación mínima para que la prueba 'test_sumar_dos_numeros' pase.
    # Simplemente devuelve la suma de los dos números.
    return a + b
```

Figure 3: Verde

```python
# test_calculadora.py
import unittest
from calculadora import sumar # ¡Ahora importamos la función!

class TestCalculadora(unittest.TestCase):
    def test_sumar_dos_numeros(self):
        self.assertEqual(sumar(2, 3), 5)
```

Figure 4: Actualizar test

2.5.4 Refactorizar

```python
# calculadora.py
def sumar(a, b):
    """
    Suma dos números y devuelve el resultado.

    Args:
        a (int or float): El primer número.
        b (int or float): El segundo número.

    Returns:
        int or float: La suma de a y b.
    """
    return a + b
```

Figure 5: Refactorizado

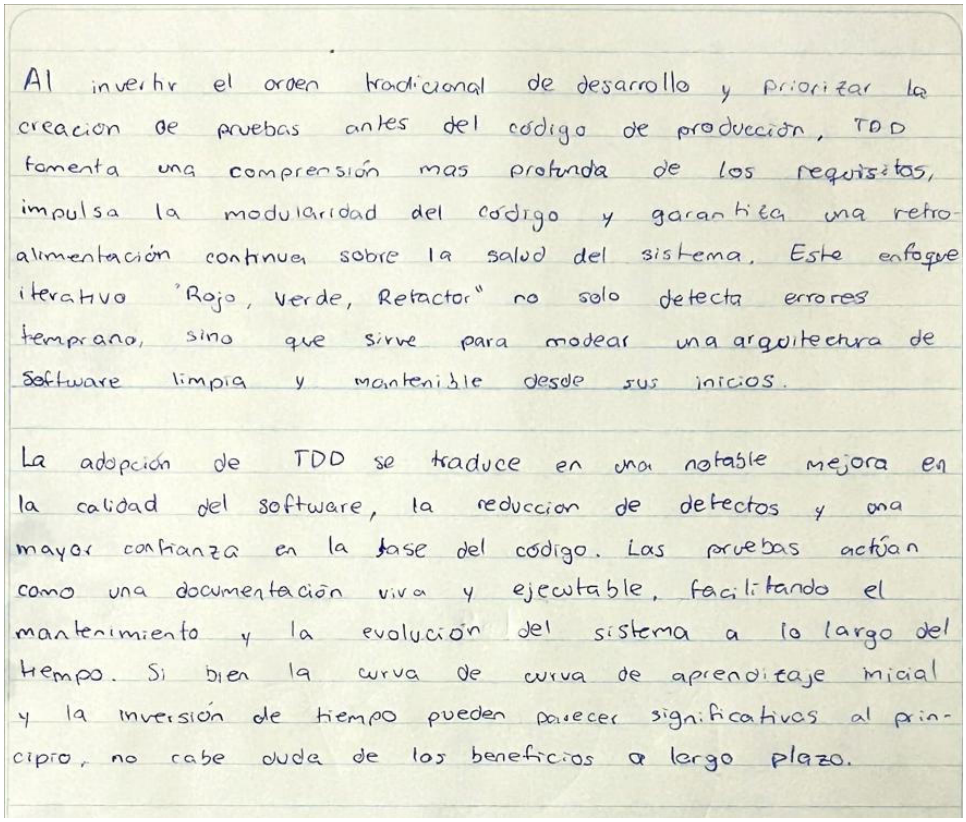
2.5.5 Verificación

```
 ❯❯ /home/bowis/Downloads/Tarea2 : python -m unittest test_calculadora.py
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Figure 6: Verificación

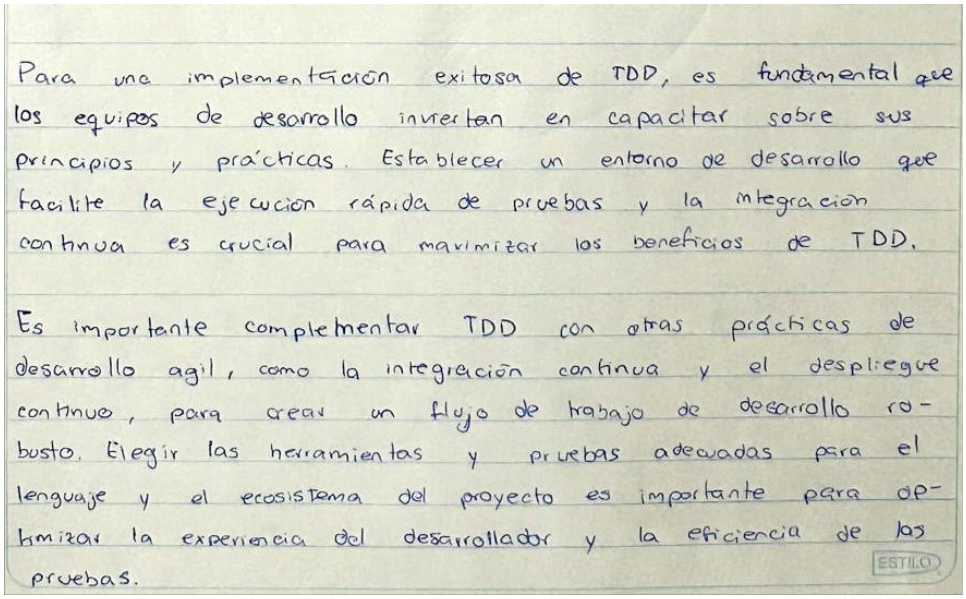
Conclusiones



Al invertir el orden tradicional de desarrollo y priorizar la creación de pruebas antes del código de producción, TDD fomenta una comprensión mas profunda de los requisitos, impulsa la modularidad del código y garantiza una retroalimentación continua sobre la salud del sistema. Este enfoque iterativo "Rojo, Verde, Refactor" no solo detecta errores temprano, sino que sirve para modear una arquitectura de software limpia y mantenible desde sus inicios.

La adopción de TDD se traduce en una notable mejora en la calidad del software, la reduccion de defectos y una mayor confianza en la base del código. Las pruebas actúan como una documentación viva y ejecutable, facilitando el mantenimiento y la evolución del sistema a lo largo del tiempo. Si bien la curva de curva de aprendizaje inicial y la inversión de tiempo pueden parecer significativas al principio, no cabe duda de los beneficios a largo plazo.

Recomendaciones



Para una implementación exitosa de TDD, es fundamental que los equipos de desarrollo inviertan en capacitar sobre sus principios y prácticas. Establecer un entorno de desarrollo que facilite la ejecución rápida de pruebas y la integración continua es crucial para maximizar los beneficios de TDD.

Es importante complementar TDD con otras prácticas de desarrollo agil, como la integración continua y el despliegue continuo, para crear un flujo de trabajo de desarrollo robusto. Elegir las herramientas y pruebas adecuadas para el lenguaje y el ecosistema del proyecto es importante para optimizar la experiencia del desarrollador y la eficiencia de las pruebas.

ESTILO

Referencias

[1] R. S. Pressman y B. Lowe, Software Engineering: A Practitioner's Approach, 9th ed. New York, NY, USA: McGraw-Hill Education, 2019.

[2] I. Sommerville, Software Engineering, 10th ed. Boston, MA, USA: Pearson, 2016.

[3] International Software Testing Qualifications Board (ISTQB). (2025). ISTQB Certified Tester Foundation Level Syllabus. [En línea]. Disponible en: https://www.istqb.org/ (Fecha de consulta: 4 de julio de 2025).

[4] Checkstyle. (2025). Checkstyle Documentation. [En línea]. Disponible en: https://checkstyle.sourceforge.io/ (Fecha de consulta: 4 de julio de 2025).

[5] JUnit. (2025). JUnit 5 Documentation. [En línea]. Disponible en: https://junit.org/junit5/ (Fecha de consulta: 4 de julio de 2025).

[6] Mocha.js. (2025). Mocha Documentation. [En línea]. Disponible en: https://mochajs.org/ (Fecha de consulta: 4 de julio de 2025).

[7] Selenium. (2025). Selenium Documentation. [En línea]. Disponible en: https://www.selenium.dev/documentation/ (Fecha de consulta: 4 de julio de 2025).

[8] Pytest. (2025). pytest Documentation. [En línea]. Disponible en: https://docs.pytest.org/en/stable/ (Fecha de consulta: 4 de julio de 2025).

[9] Pylint. (2025). Pylint Documentation. [En línea]. Disponible en: https://pylint.pycqa.org/en/latest/ (Fecha de consulta: 4 de julio de 2025).

[10] NUnit. (2025). NUnit Documentation. [En línea]. Disponible en: https://nunit.org/ (Fecha de consulta: 4 de julio de 2025).