



T5: TDD

¿Qué es TDD?

Un test que usa métodos así:

Diseñar $\Delta \rightarrow$ test

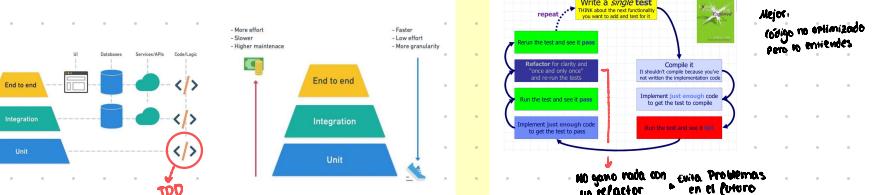
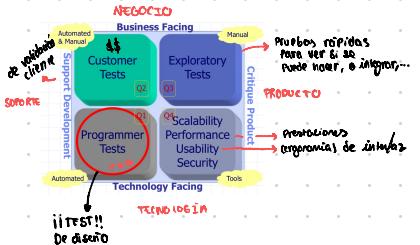
- Toma la idea desde el punto de verano **codigo!!**
- Toma la idea de los **clases de testing** → aislamiento
- Produce **desplazamiento** y **↑ coherencia**
 -claro relacionado
 -interact.
 -entendible por separado

Los pruebors conducen al código

- ↳ código no se puede construir de goteo
- ↳ idea q la base del código esté organizadamente a medida q se crean ejemplos de lo q **debe** hacer

El código se escribe desde fuera hacia dentro

- ↳ test → **GUI** hace realmente el trabajo
- ↳ test es parte **desde documentación**



¿Por qué TDD?

• **codificación Testing [aburrido]**

- Fomenta mantener una cantidad exhaustiva de test reutilizables
- Mejor q "debug-later programming"

TEST UNITARIOS

↳ desde el principio más q de verificación

• **programador = desarrollador**

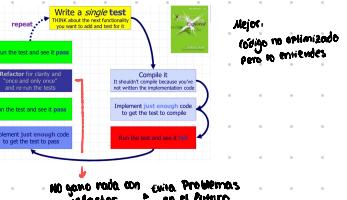
• TEST UNITARIOS → F(x) intento de una clase

- ↳ ¿qué es un **unitario**? Trío más pequeño q una/mi solo procedimiento autónomo de una f(x). Agro tan pequeño q puede ser desarrollado por 1 persona.
- ↳ Caja negra para testear obj. Clases → testear aislamiento

¿Cómo escribir un test?

- TDD promueve **pequeños pasos**: termino más pronto + poco obstáculos → evitar
- TDD: **pequeños pasos** → **pequeña exploración del código**

Tareas del TDD



Unit Testing Framework

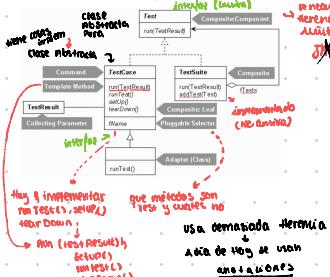
• **Test cobran fuerza autoritativas**

• **JUnit es el rey de estos que son**

JUnit, XUnit,...



TAREA JUnit



Ciclo TDD

1. Escribe un test (rojo)

Debes pillar

Mejor, código no optimizado pero lo entiendes

2. Ejecútalo (verde)

- Debe pillar los test
- Puedes garantizar

3. Refactor: Hazlo bien

- cambia el diseño
- elimina errores, duplicados,...

Proceso de TDD

Características y Beneficios

- Una vez pasa el test → se vuelve a pasar con **toda** cambio
- No se toleran "broken test"
- Ejecutos secundarios → detectan inmediatamente
- Los Asistentes → recomienda periódicamente
- Aislamiento → un test NO debería afectar a otro
- Test automatizados → red de seguridad para refactoring

- ¿Qué se prueba? → Todo lo q se tiene q romper (sentido común)

- 1) Emisión suave, pequeño

- 2) Cada conjunto de test demuestra q los componentes q los conectan?
- 3) Hacer uno test

Test Unitarios Automatizados

- test unitarios → juntarlos en **Test** → no debiera haber
- Repetirlo lo q el código debe tener (ejemplos, no ejemplos)
- Lo ejemplos de como q es el código (documentación)
- Juzgar los resultados BIEN?

- Validar con **BICEPS**

- Si no sabes lo q "bien" significa, como lo vas a probar?
- Requisitos, Anexos o no anexos?

- Tomar una decisión

BICEPS

- **Boundary conditions**

- ! CORRECT:
 - Contar邚os → código responde requisitos
 - Descripción →
 - Rango →
 - Reference → depende de un criterio? Ej: menor
 - Existence →
 - Cardinality → Absoluto (0 ó más) o relativo (1 ó no más de n)

Revisar Invaciones

- Ej: revisar lista ()
- inserir (2)
- borrar (2)
- lista (2)
- lista == 2

Cross-checks: multiplicación: Probar multiplicando a lo largo y a lo alto para ver condiciones de errores y振奋 invitados, excepciones, fallas, etc.

Objetos q preparan a los otros objetos

Salto de items, red...

salto de items, red...

T5: TDD

Escribir TEST = Entender Código = Identif Fallos Mejor

Recomendaciones

Usa el Compilador

- Arregla los errores y Advertencias

assert/check por test

- Si un test falla → sbt en g exactamente
- Si tienes 2 assert → no sbt cual es el q ha fallado

KISS (Keep It Simple Stupid)

- lo más simple para q pase el test
- DRY - Don't Repeat Yourself
- OACO → Only And Once only (Duplicados)

fake it (Till you Make it)

- El test no te sale → return 0
- poco a poco vas mejorando la salida

Ejemplo implementación PyUnit:

- First version
- return

- Second version
- return 0 if args < 2 else len(args) - 1

- Third version
- return 0 if args[0] == 'sum' & len(args) > 1 else len(args) - 1

Triangulate

Abstact solo cuando tengas 2 o más ejemplos

```
import unittest
class TestClass(unittest.TestCase):
    def test_plus(self):
        self.assertEqual(1, plus(1,1))

def plus(augend, addend):
    return 0

import unittest
class TestClass(unittest.TestCase):
    def test_plus(self):
        self.assertEqual(1, plus(1,1))
        self.assertEqual(1, plus(1,1))
```

Implementación Obviamente

- Take it o Triangulación
- Si sabes q escribir y es rápido → hazlo

Después del Primer Ciclo tienes...

- 1) Lista de test con los q necesitas trabajar
- 2) Historia contada sobre como se va a ver una operación
- 3) Los test compilan con "Codes" return 4
- 4) Los test corren, incluso con guardados de código
- 5) Gradualmente generalizar el código, cts vs variables 4 ≈ 96%
- 6) Añade los items a una TD-DO-List

Refactorizar

- No puede cambiar la semántica del programa
- "Observación equivalente"
 - ↳ todos los test q pasaban ANTES del refactor → tienen q pasar DESPUÉS
 - ↳ debes ver cuantos fallos con los test al refactorizar

Niveles de Escala

- ↳ 2 Buclez son Σ Δ = y merge
- ↳ 2 Rama de condicionales son Δ Δ = y elimina una
- ↳ 2 Métodos Δ Δ = y elimina
- ↳ 2 Clases Δ Δ = y elimina

Resumen

- ① TDD no reemplaza el Sist stand. de Testing
 - Define una forma probada y efectiva de hacer test unitarios
 - Test son ejemplos de como funciona
 - Ningún código debe ir a producción si no tiene un test
- ② TDD no es nuevo
- ③ TDD = menos θ de debuggando → entender código = tener errores fáciles
- ④ TDD niega el miedo
 - ↳ miedo → ∇PC + Basta q le hagan muchos test
- ⑤ TDD crea → conjunto de "Test programadores"
 - ↳ test automatizados → El por el API
 - ↳ exhaustivo: Necesitar uno y otra vez
- ⑥ TDD Permite Refactor sin Miedo
- ⑦ (con cuidado) se pueden crear test de validación User Acceptance Test programar