Test de Unidad

- Testeo de la mínima unidad de ejecución.
- En OOP, la mínima unidad es un método
- Objetivo: aislar cada parte de un programa y mostrar que funciona correctamente.
- Cada test confirma que un método produce el output esperado ante un input conocido.
- Es como un contrato escrito de lo que esa unidad tiene que satisfacer.

Estrategias:

- Particiones Equivalentes
    - Considerar el "dominio" de la funcionalidad a testear
        - Si un elemento pasa el test, otros del mismo conjunto pasarán
    - Considerando el complemento del dominio
        - No hace lo que no debe hacer
- Valores de Borde
    - Considerando las Particiones Equivalentes
        - Se testean escenarios con valores en los límites de las particiones

Test Double

- Crear un objeto que es una maqueta (polimórfica) del objeto o módulo requerido
- Utilizar la maqueta según se necesite

Se generan diferentes patrones que se aplican a cada caso

Test Stub

- Cascarón vacío. Sirve para que el SUT envíe los mensajes esperados

Es un objeto utilizado en pruebas unitarias que proporciona respuestas predefinidas a llamadas realizadas por el código bajo prueba. Los stubs se configuran para simular el comportamiento de las dependencias reales y pueden devolver valores específicos o generar excepciones controladas. Los stubs se utilizan para aislar el código bajo prueba y verificar cómo responde a diferentes resultados esperados de las dependencias.

Test Spy

- Test Stub + registro de outputs

Es un objeto utilizado en pruebas unitarias que registra información sobre las llamadas realizadas por el código bajo prueba a sus dependencias. A diferencia de los stubs, los spies no definen expectativas de llamadas y no generan errores si las llamadas esperadas no ocurren. En lugar de eso, se utilizan para verificar el comportamiento posteriormente, inspeccionando las llamadas registradas. Los spies son útiles para verificar qué llamadas se realizaron y con qué parámetros, permitiendo analizar el comportamiento del código bajo prueba.

Mock Object

- test Stub + verification of outputs

Es un objeto utilizado en pruebas unitarias que se configura con expectativas de llamadas específicas. Los mock objects se utilizan para verificar las interacciones entre el código bajo prueba y sus dependencias. Se definen expectativas de llamadas y se comprueba si se han cumplido durante la prueba. Si alguna expectativa no se cumple, se genera un error de prueba.

Los mock objects son más rigurosos que los stubs y se centran en verificar las interacciones correctas entre objetos.

Fake Object

- imitación. Se comporta como el módulo real (protocolos, tiempos de respuesta, etc)

Es una implementación simplificada de un objeto real utilizado en pruebas unitarias. A diferencia de los stubs, los objetos falsos no se configuran para proporcionar respuestas predefinidas o simular interacciones específicas. En cambio, suelen ser versiones más simples y controladas de las dependencias reales. Por ejemplo, se pueden crear versiones en memoria de una base de datos o un servicio web para acelerar las pruebas y eliminar la dependencia de componentes externos.

- Implementar clases según sea necesario
    - Objetos que no están disponibles para probar
- Que ocurre con objetos que están disponibles
    - Respaldar con test cases
- Implementación:
    - Test Stub: simple y barato
    - Fake object: demanda análisis, threading, requiere mantenimiento

Test de Aceptación

- Por cada funcionalidad esperada.
- Escritos desde la perspectiva del cliente

Test Driven Development

- Combina:
    - Test First Development: escribir el test antes del código que haga pasar el test
    - Refactoring
- Objetivo:
    - pensar en el diseño y qué se espera de cada requerimiento antes de escribir código
    - escribir código limpio que funcione (como técnica de programación)
- Filosofía:
    - Vuelco completo al desarrollo de software tradicional. En vez de escribir el código primero y luego los tests, se escriben los tests primero antes que el código.
    - Se escriben tests funcionales para capturar use cases que se validan automáticamente
    - Se escriben tests de unidad para enfocarse en pequeñas partes a la vez y aislar los errores
    - No agregar funcionalidad hasta que no haya un test que no pasa porque esa funcionalidad no existe.
    - Una vez escrito el test, se codifica lo necesario para que todo el test pase.

    - Pequeños pasos: un test, un poco de código
    - Una vez que los tests pasan, se refactoriza para asegurar que se mantenga una buena calidad en el código.
- Reglas:
    - Diseñar incrementalmente:
        - teniendo código que funciona como feedback para ayudar en las decisiones entre iteraciones.
    - Los programadores escriben sus propios tests:
        - no es efectivo tener que esperar a otro que los escriba por ellos.
    - El diseño debe consistir de componentes altamente cohesivos y desacoplados entre si:
        - mejora evolución y mantenimiento del sistema.
- Hay herramientas que automatizan TDD.

TDD

✔ Menor chance de Sobrediseño

✔ Proceso de elicitación del domino

✔ Arquitectura surge incrementalmente

✔ Refactoring mantiene código mínimo y limpio

✔ Requerimientos se convierten en test cases

✔ Último “Release” como entregable

✔ Adaptabilidad al cambio

⚠️ Refactoring mantiene código mínimo y limpio

⚠️ Mantener objetivo a largo plazo

⚠️ Cambios en DB puede ser costosos

⚠️⚠️ Interacción expertos del dominio

⚠️