Introducción

Definición: práctica de diseño de software orientada a objetos.

TDD como práctica de metodología:

Incluye tres sub-prácticas:

- Automatización: las pruebas del programa deben ser hechas en código, y con la simple corrida del código de pruebas debemos saber si lo que estamos probando funciona bien o mal.
- Test-First: las pruebas se escriben antes del propio código a probar.
- Refactorización posterior: para mantener la calidad del diseño, se cambia el diseño sin cambiar la funcionalidad, manteniendo las pruebas como reaseguro.

Ventajas:

- Automatización:
  - Independización del factor humano, con su carga de subjetividad y variabilidad en el tiempo.
  - Repetición de las mismas pruebas, con un costo ínfimo comparado con las pruebas realizadas por una persona
- Test-first:
  - Da confianza de que el código que uno escribe siempre funciona.
  - Permite especificar el comportamiento sin restringirse a una única implementación
- Refactor:
  - Facilita el mantenimiento de un buen diseño.

Reglas:

"Nunca escribas una nueva funcionalidad sin una prueba que falle antes."

"Si no puedes escribir una prueba para lo que estás por codear, entonces no deberías estar pensando en codearlo."

La aparición de los frameworks y el corrimiento de TDD a UTDD

Problema: TDD se vio como sinónimo de pruebas unitarias automatizadas realizadas antes de escribir el código.

Causas de esta visión:

1. Primer causa: proviene del propio nombre de TDD que incluye la palabra "test". Además, las herramientas que surgieron de este método obligaban a empezar las pruebas con "test" o requerían que derivasen de la clase "TestCase".
2. Segunda causa: en todos los ejemplos que plantean los creadores, se trabaja sobre pequeñas porciones de código e incluso ellos mismos hablan de pruebas unitarias.

Ventajas de UTDD:

- Las pruebas en código sirven como documentación.

- ❖ Las pruebas en código indican con menor ambigüedad lo que las clases y métodos deben hacer.
- ❖ Las pruebas escritas con anterioridad ayudan a entender mejor la clase que se está creando.
- ❖ Las pruebas escritas con anterioridad suelen incluir más casos de pruebas negativas.

Desventajas de UTDD:

- ❖ Tiende a basar todo el desarrollo en la programación de pequeñas unidades, sin una visión del conjunto.
- ❖ Muchos critican que la arquitectura evolucione sola sin hacer nada de diseño previo.
- ❖ No permite probar interfaces ni comportamiento esperado por el cliente.
- ❖ Los cambios de diseño medianos y grandes suelen exigir cambios en las pruebas unitarias.

Los primeros intentos de pruebas de integración

Problema de la integración: qué desarrollar primero y cómo probar las interacciones con módulos aún no implementados.
Solución: la idea más simple es la de construir módulos ficticios o stubs.

Objetos ficticios:

- ❖ Dummy object (objeto ficticio): aquellos objetos que se generan para probar una funcionalidad, pero no se utilizan en la prueba.
  - ➢ Ejemplo: cuando un método necesita un objeto como parámetro, pero éste no se usa en la prueba.
- ❖ Test Stub (muñón): son los que reemplazan a objetos reales del sistema, generalmente para generar entradas de datos o impulsar funcionalidades del objeto que está siendo probado.
  - ➢ Ejemplo: objetos que invocan mensajes sobre el objeto sobetido a prueba.
- ❖ Test Spy (espía): se usan para verificar los mensajes que envía el objeto que se está probando, una vez corrida la prueba.
- ❖ Mock Object (objeto de imitación): son objetos que reemplazan a objetos reales para observar los mensajes enviados a otros objetos.
- ❖ Fake Objetct (objeto falso): reemplazan objetos con una implementación alternativa.

Objetivos:

- ❖ Disminuir las dependencias.
- ❖ Mantener las pruebas de integración como pruebas unitarias.

El punto de vista de los requerimientos del comportamiento:

Otras prácticas de diseño:

- ATDD: obtiene el producto a partir de pruebas de aceptación.
- BTDD: TDD bien hecho.
- STDD: utiliza pruebas como ejemplos.
- NTDD: comportamiento definido en como se envian mensajes los objetos.

ATDD (Acceptance Test Driven Development):

Definición: es una práctica de diseño de software que obtiene el producto a partir de las pruebas de aceptación. Se basa en TDD, pero en vez de escribir pruebas unitarias (que escriben y usan los programadores), escriben pruebas de aceptación de usuario, en conjunto con ellos.

Idea: tomar cada requerimiento, en forma de user story, construir varias pruebas de aceptación de usuario y a partir de ellas construir las pruebas automáticas de aceptación, para luego escribir el código.

User story: requerimientos simples y no representan el cómo, sino solamente quién necesita qué y por qué.

Ventajas:

- A los clientes no les interesa el código, sino que el producto satisfaga sus necesidades con la mejor calidad posible, ATDD permite saber cuando una necesidad se ha satisfecho: sólo hay que ver que la prueba de aceptación basada en esa user story funque.
- Ayuda a eliminar posibles ambigüedades entre qué quiere el cliente y qué entiende el desarrollador.

La premisa fundamental es que TDD se concibió para diseñar y probar código, pero los clientes no están interesados en el código, sino en que el sistema satisfaga sus necesidades con la mejor calidad posible. ATDD permite conocer mejor cuándo se ha satisfecho un requerimiento, tanto para el desarrollador como para el cliente: simplemente hay que preguntar si todas las pruebas de aceptación de las user story están funcionando.

BDD: TDD mejorado

Idea:

- ❖ En vez de hacer pruebas de clases y métodos, lo más lógico es probar porciones de comportamiento esperados del sistema.
- ❖ En vez de pensar en términos de pruebas, deberíamos pensar en términos de especificaciones o comportamiento.
- ❖ Escribir las pruebas desde el punto de vista del consumidor y no del producto.

Prácticas:

- ❖ Cambiar todos los métodos test por should o must.
- ❖ Escribir una clase de prueba por requerimiento y un método por aserción.

- Escribir los nombres de los métodos para que se puedan leer como oraciones.
- Hablar en el lenguaje del cliente ayuda mantener la alta abstracción, sin caer en detalles de implementación.

Críticas:

- Un cambio de nombre a TDD, cuidando un poco el lenguaje.
- TDD bien hecho.

STDD: ejemplos como pruebas y pruebas como ejemplos

Idea: usar ejemplos como partes de las especificaciones

- los desarrolladores construyen su código y escriben pruebas unitarias, para las cuales deben basarse en ejemplos de entradas y salidas.
- los testes desarrollan casos de pruebas, que contienen a su vez ejemplos.
- a veces los desarrolladores y los testers le solicitan a los analistas que les den ejemplos concretos para aclarar ideas y otras veces son los mismos analistas los que proveen escenarios, que no son más que requerimientos en la forma de ejemplos.

Ventajas de los requerimientos con ejemplos:

- sirven como herramienta de comunicación
- evita las largas descripciones y reglas de prosa, propensas a interpretaciones diversas.
- son más sencillos de acordar con los clientes
- sirven como pruebas de aceptación

Desventajas:

- A veces ocurre que al plantear los requerimientos como ejemplos se pierde la visión global del proyecto.
- No todo requerimiento puede llevarse a un ejemplo. Ejemplo: una aplicación que debe generar un número al azar, los ejemplos que podamos escribir nunca van a servir como prueba de aceptación.
- No resulta fácil escribir ejemplos para pruebas de estrés o de desempeño.

Roles del enfoque tradicional:

- analistas: hace de traductor entre los clientes y los desarrolladores y testers
- tester: valida el producto contra las especificaciones. Recibe especificaciones, elabora casos de prueba en base a escenario y ejecuta casos de prueba.
- desarrollador: diseñador y constructor del producto
- cliente o usuario: solo habla con el analista.

Roles en el enfoque STDD:

- Analista: deja de ser un traductor para ser un facilitador de intercambio de conocimiento

- Cliente: tiene un rol más activo, como autor principal de pruebas de aceptación con ejemplos, asistido por el resto de los interesados

Taller:

- intercambiar ideas entre los participantes
- se reduce la brecha entre lo que entiende uno y otro participante

El foco de diseño orientado a objetos: NDD

Ley de Demeter: pretende que los objetos revelen lo mínimo posible de su estado interno, dando a conocer sólo su comportamiento.

Idea: el comportamiento de los objetos debe estar definido en términos de cómo envía mensajes a otros objetos, además de los resultados devueltos en respuesta a mensajes recibidos.

Propósito: obtener un buen diseño orientado a objetos y una separación clara de responsabilidades, sobre la base de:

- mejorar el código en términos de dominio
- preservar el encapsulamiento
- reducir dependencias
- clarificar las interacciones entre clases

Pretende definir de antemano qué mensajes emitirá el objeto sometido a prueba cuando reciba un mensaje en particular.

Utiliza Mock Objects: porque nos interesa la comunicación entre objetos y no su estado o implementación.

Ventajas:

- Interfaces bien angostas.

Desventajas:

- Sensible a refactorizaciones, ya que al renombrar o sacar métodos, las pruebas dejan de funcionar ya que los Mock Objects trabajan buscando métodos por nombre.

Pruebas de interacción y de TDD

Pruebas e interfaz de usuarios

Problemas a la hora de hacer pruebas:

- la interfaz de usuario es cambiante, por lo que las pruebas se desactualizan con mucha frecuencia.
- Suelen ser muy lentas de ejecutar
- Hay cuestiones que las puede probar satisfactoriamente un usuario, por ejemplo, la ubicación de los botones, los colores, la consistencia de la aplicación en su totalidad.

Pero por qué es importante hacerlas:

- El valor de la aplicación se muestra a través de la interfaz de usuario.
- La mayoría de los usuarios reales sólo consideran que la aplicación les sirve cuando la ven a través de su interfaz de usuario.

Cómo se prueban: casi todas las herramientas permiten grabar una prueba realizada a mano para después ejecutarla de forma repetitiva.

Limitaciones de las pruebas de interacción:

Problemas que suelen presentar las pruebas grabadas:

- Sensibilidad al comportamiento: los cambios en el comportamiento generan cambios importantes en la interfaz de usuario, que hacen que las pruebas de interacción dejen de funcionar.
- Sensibilidad a la interfaz: aún pequeños cambios a la interfaz provocan que las pruebas dejen de correr y deban ser cambiadas.
- Sensibilidad a los datos: si se cambian los datos con los que corren las pruebas, los resultados arrojados por las mismas van a cambiar, lo que hace que haya que generar datos especiales para probar.
- Sensibilidad al contexto: las pruebas pueden ser sensibles a cambios en el dispositivo externos a la aplicación.

Consejos:

- Tratar de no probar al mismo tiempo la lógica del negocio con la lógica de la interacción.
- Evitar hacer estas pruebas automatizadas si la lógica del negocio es muy cambiante.
- Volver a generar las pruebas cada tanto.

Tipos de pruebas y automatización

Que automatizar