TESTING UNITARIO

Utilizando Angular – Conceptos Básicos

Conociendo las pruebas unitarias

¿qué es una prueba unitaria?

Para comprender que es una prueba unitaria es importante definir qué entendemos por "**unidad**":

*Una **unidad** es simplemente una porción de código o una funcionalidad que realiza una acción específica, de la cual podemos probar los resultados.*

En el caso de Angular, la unidad es la función.

Entonces:

*Una **prueba unitaria** es una prueba para asegurarnos que la porción de código o funcionalidad hace lo que debería hacer, independientemente del resto de la aplicación.*

En Angular, el objetivo de las **pruebas unitarias** es probar el comportamiento de cada uno de las funciones del componente o servicio de manera aislada.

Se trata del primer nivel de pruebas que debe superar el código.

Conociendo las pruebas unitarias

otros niveles de prueba

Además de las pruebas unitarias existen:

- **Pruebas de Integración**: Tienen como objetivo verificar la correcta integración de todas las unidades probadas en la prueba unitaria (**integración interna**) y la correcta integración con otros sistemas involucrados (**integración externa**). Normalmente se realizan tras las pruebas unitarias.
- **Pruebas de Sistema**: Su objetivo es verificar el **comportamiento del sistema** en su **totalidad**. Para ello, las pruebas de Sistema se afrontan desde una doble perspectiva para **cubrir tanto los requisitos Funcionales como los No Funcionales** del sistema bajo prueba. Normalmente, tras las pruebas de integración.
- **Pruebas de Aceptación**: Están **orientadas a la aceptación final** del producto por parte del usuario. El objetivo principal de este nivel de pruebas, es **dar confianza al usuario** de que el sistema funcionará de acuerdo a sus expectativas, cumpliendo los requisitos funcionales y operativos esperados. Normalmente, tras las pruebas de sistema.

Conociendo las pruebas unitarias

ventajas

- ✓ **Aseguran la calidad del código entregado:** Es la mejor forma de detectar errores tempranamente en el desarrollo. Los errores son más fáciles de localizar con este tipo de pruebas.

- ✓ **Fomentan el cambio y la refactorización:** Permiten realizar pruebas sobre los cambios de forma rápida, por ello si consideramos que nuestro código es mejorable podemos cambiarlo con mayores garantías de que no se introducen errores.

- ✓ **Reducen los problemas y tiempos dedicados a la integración:** Permiten llegar a la fase de integración con un grado alto de seguridad de que el código está funcionando correctamente, incluso las dependencias (se pueden simular).

- ✓ **Documentan el código:** A través de las pruebas podemos comprender mejor qué hace una función y qué se espera de ella.

Conociendo las pruebas unitarias

limitaciones

- Nos ayudan a detectar defectos, pero no nos garantizan la ausencia de éstos. Constituye uno de los fundamentos del testing, las pruebas muestran la existencia de defectos, no la ausencia de los mismos.
- Al estar ejercitando trozos de código o funcionalidades independientes de forma aislada, no servirán para descubrir problemas que afecten a todo el sistema en su conjunto. Por eso son más efectivas si se combinan con otros niveles de prueba.

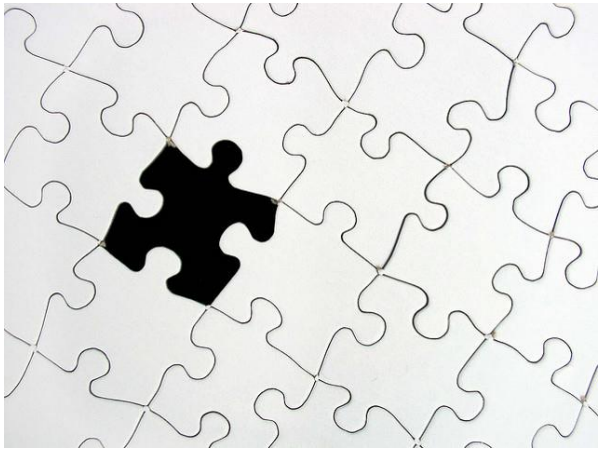
Conociendo las pruebas unitarias

principios FIRST

Las pruebas unitarias deben cumplir las siguientes cualidades:

- **Fast - Rápidas:** La ejecución de las pruebas debe ser rápida, para que no suponga esfuerzo ejecutarlas siempre que se necesiten. Cuanto más frecuentemente se ejecuten, mas eficaces son para detectar problemas. Por este motivo lo mejor es **automatizarlas**, e intentar que requieran la mínima intervención manual para su ejecución.

- **Independent - Independientes:** Para facilitar la tarea de localizar el origen del error es muy importante que los tests sean **independientes** los unos de los otros. Para lograr esto deberemos evitar a toda costa que las salidas de unos tests se utilicen como entradas de otros. Es más, no deberíamos preocuparnos del orden en el cual se vayan a ejecutar los tests ya que cada ejecución puede tener una ordenación distinta sin que eso afecte al resultado.



Conociendo las pruebas unitarias

principios FIRST

- **Repeteable - Repetibles:** No se deben crear pruebas unitarias que solo se puedan ejecutar una única vez, ya sea por motivos del entorno o dependencias. Todos los miembros del equipo de desarrollo deben poder probar que cualquier cambio que hayan hecho no ha afectado al software que están construyendo entre todos. Las principales dependencias suelen ser de la configuración de usuario, la versión de la BBDD, o el sistema operativo. Una solución es utilizar el servidor de integración continua como referente, y en caso de discordancia en el resultado de los tests, determinar la causa para solucionarla cuanto antes.

- **Self-validating – Auto-evaluables**: Las pruebas deben ofrecer un resultado claro que indique si ha pasado o ha fallado. No debería ser necesario realizar absolutamente ninguna operación por parte del programador, tal como comparar valores, leer un fichero de resultados, etc. De ese modo, las herramientas de pruebas pueden ofrecer un resultado global absoluto para el conjunto de pruebas.

Conociendo las pruebas unitarias

principios FIRST

- **Timely – Oportunas**: Las pruebas deberían escribirse *justo cuando se necesitan*, es decir, a la vez que se esté construyendo el software. Es mala idea dejar estas pruebas para el final, cuando el software se haya implementado completamente, porque una vez que se tiene el código funcionando se utiliza como excusa para no escribirla.

Arrange-Act-Assert

Definición

Es una modelo de organizar las pruebas unitarias, el cual se ha convertido en casi un estándar. Este modelo sugiere dividir el código de prueba en **tres secciones: Arrange (organizar), Act (actuar), Assert (afirmar).** Cada sección se debe responsabilizar de lo relacionado a su nombre.

Principalmente se organiza el código, dejando una línea de espacio en blanco entre una sección y la siguiente.

Arrange-Act-Assert

La triple A

Arrange (organizar): Sección donde se inician las variables, elementos, componentes u otro. Los que serán utilizados en el test.

Act (actuar): Sección donde se realizará el llamado y ejecución de las funciones para ser probadas.

Assert (validar): Sección donde se comprobará si el test fue aprobado o no.

NTT DATA

JASMINE Y KARMA

Desarrollo de pruebas unitarias en angular

Conociendo Jasmine y Karma

¿Qué es Jasmine y Karma?

Jasmine *es un behavior-driven devepment framework, el cual es utilizado en programación para hacer pruebas unitarias de aplicaciones JavaScript. Esto permite validar que las funciones desarrolladas, entreguen respuestas o resultados según lo que se esperaba.*

Karma *es un test runner creado por el equipo de angular, el cual nos permite automatizar algunas tareas de los frames de tests, por ejemplo Jasmine.*

¿Por qué utilizarlos?

- La ejecución de pruebas unitarias no necesariamente tiene que ser automatizada, aunque ya hemos visto que **es lo optimo**, ya que van a repetirse con bastante frecuencia.
- Aunque es posible automatizar las pruebas unitarias generando nuestras propias clases y funciones, conviene disponer de una herramienta para **facilitar esta labor**.
- Jasmine es un framework Open Source que se ha convertido en el **estándar para la automatización** de las pruebas de funciones JavaScript.

Conociendo Jasmine y Karma

otras ventajas…

- ✓ Es **gratuito**, open source y puede ser utilizado en desarrollos comerciales.
- ✓ Actualmente se encuentra bien **mantenido** y en **constante mejora**.
- ✓ Es **fácil de usar y configurar** por lo que prácticamente no existe curva de aprendizaje.

salvo el cambio de mentalidad en los programadores para usar técnicas que faciliten el uso de unidades de prueba.

- ✓ Viene preinstalado al momento de crear un proyecto basado en angular.
- ✓ Incluye **formas de ver los resultados,** que pueden ser en modo texto (consola), reporte html (generado por karma).
- ✓ Permite su inclusión automática en procesos de **integración continua**.

Conociendo Jasmine y Karma

¿cómo funciona Jasmine?

estructura de un caso de prueba

- **Preparación**
  - Crear los accesorios necesarios para realizar el test
- **Ejecución**
  - Ejercitar la clase o componente que queremos probar
- **Verificación**
  - Comprobar que la ejecución ha sido correcta
- **Limpieza**
  - Eliminar los accesorios utilizados

Conociendo Jasmine y Karma

conceptos básicos

Tipos de funciones

La clase de Jasmine para realizar el Test tiene los siguientes tipos de funciones:

Preparación

- **describe:** Para agrupar los test relacionados a una misma función o componente (suit de pruebas).
- **beforeEach**: Para crear los accesorios necesarios antes de la ejecución de cada test, por ejemplo asignar valores iniciales.
- **beforeAll**: Para crear los accesorios utilizados por varias pruebas, que van a ser creados solo una vez: antes de empezar los test.

Ejecución

Verificación

- **it**: Es la declaración de una prueba unitaria, la cual pertenece a una suit de pruebas (describe).

Limpieza

- **afterEach**: Es llamado después de cada test y puede servir para liberar recursos o similar.
- **afterAll**: Igual que antes, si queremos que sólo se llame al final de la ejecución de todos los test.

Conociendo Jasmine y Karma

conceptos básicos

Tipos de verificaciones

Las siguientes funciones se pueden utilizar para verificar el tipo de resultado esperado con el test. Estas se llaman desde el **expect**:

- **toBe:** Función que espera una aseveración. (true, false, null, etc).
- **toBeTruthy:** El resultado siempre se esperará a que exista.
- **toBeFalsy:** El resultado siempre se esperará a que no exista.
- **toContain:** Generalmente utilizado para verificar si contiene algún valor o texto.
- **toEqual:** Espera siempre que la comparación sea igual al objeto, valor, texto, etc.

Conociendo Jasmine y Karma

conceptos básicos

Ejecución de pruebas

Para ejecutar las pruebas se debe utilizar el comando **ng test,** en la terminal dentro del proyecto. Esto ejecutará con karma el set de pruebas que se fueron definidas, entregando la información en la terminal y además se abrirá un navegador con los resultados.

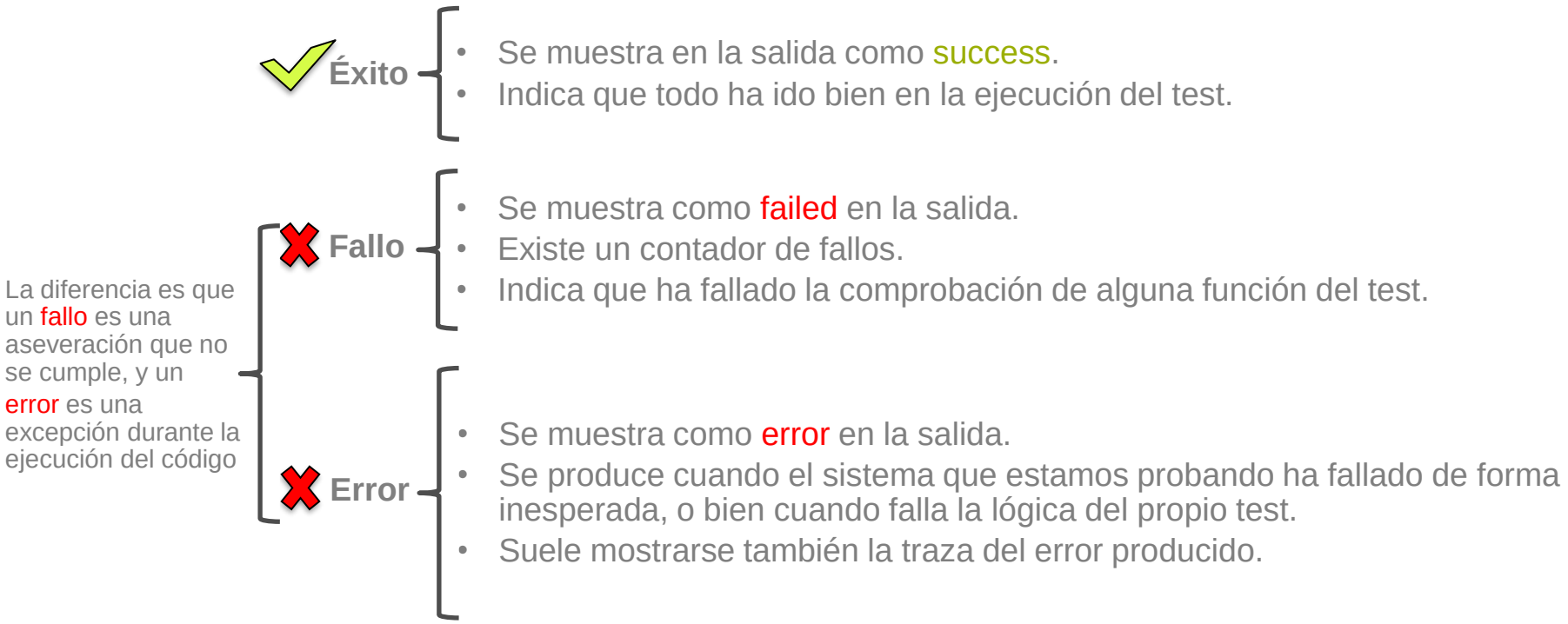
En caso que se desee ejecutar los test unitarios y además conocer la cobertura de código que estos test abarcan, se debe ejecutar el comando **ng test –code-coverage.** El cual generará una carpeta de nombre **coverage** y dentro de esta se podrá ver al abrir el index.html. Además la información será mostrada en la terminal.

Conociendo Jasmine y Karma

conceptos básicos

Resultados

Al ejecutar un test, los resultados que podemos obtener son:



Reglas de Testing

- Probar solamente una funcionalidad a la vez.
- Probar lógica de Negocio, no funciones.
- Las pruebas deben ser repetibles y consistentes

Ejercicio

Crear componente “Calculadora” y hacer la función de sumar con sus test unitarios

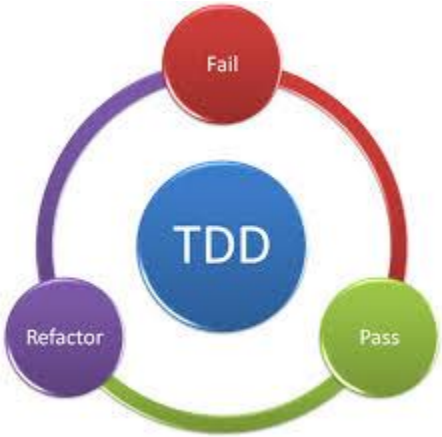
NTT DATA

TDD

Test Driven Development

TDD – Test Driven Development

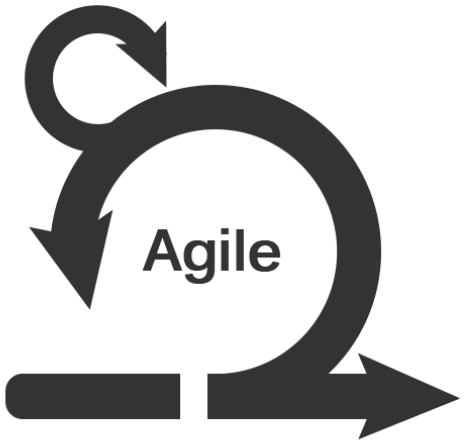
Definición



- **“Desarrollo guiado por pruebas de software**, o **Test-driven development (TDD)** es una práctica de ingeniería de software que involucra otras dos prácticas: *Escribir las pruebas primero (Test First Development)* y *Refactorización (Refactoring)*. Para escribir las pruebas generalmente se utilizan las pruebas unitarias (unit test en inglés). En primer lugar, se escribe una prueba y se verifica que las pruebas fallan. A continuación, se implementa el código que hace que la prueba pase satisfactoriamente y seguidamente se refactoriza el código escrito. El propósito del *desarrollo guiado por pruebas* es lograr un código limpio que funcione. La idea es que los requisitos sean traducidos a pruebas, de este modo, cuando las pruebas pasen se garantizará que el software cumple con los requisitos que se han establecido.” (Wikipedia)

TDD – Test Driven Development

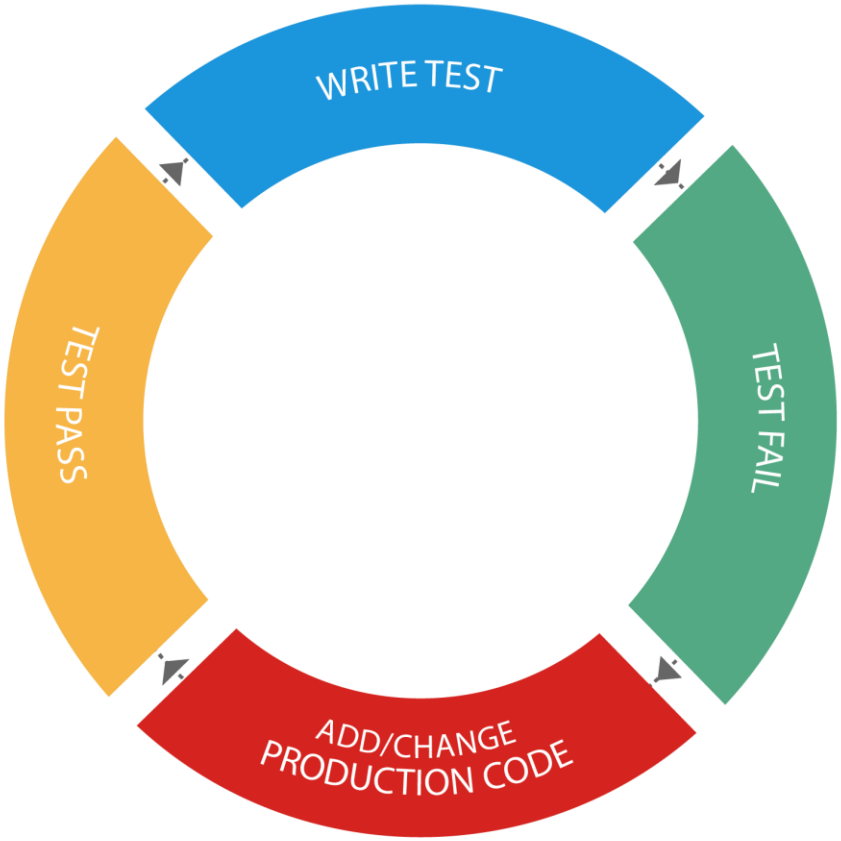
Desarrollo Ágil - Características



- El tiempo es corto
- El foco debe ser el valor generado
- La arquitectura debe de soportar el cambio
- Cualquier cambio en código es bienvenido
- La propiedad del código debe ser colectiva
- Los requerimientos cambian
- Documentación restricta (si lo hay!)

TDD – Test Driven Development

Ciclo de desarrollo en TDD



TDD – Test Driven Development

Ventajas de su utilización

- Mejor calidad
- Diseño enfocado en las necesidades
- Mayor simplicidad en el diseño
- El diseño se va adaptando al entendimiento del problema
- Mayor productividad
- Menos tiempo invertido en correcciones de errores
- Código enfocado en el comportamiento esperado

TDD – Test Driven Development

Dificultades en su utilización

- Interfaz de usuario
- La base de datos
- Falso sentimiento de seguridad
- Perder la visión general (Ver el árbol en lugar del bosque)
- Tiempo de adaptación / cambio de mentalidad

TDD – Test Driven Development

Test doubles - 1/2

Sirven para reemplazar un objeto real que es costoso en tiempo, recursos o es imposible de tener en entorno de prueba.

- **Dummys:** solo sustituyen un objeto real; no serán utilizados para nada. Por ejemplo, son creados para rellenar argumentos de métodos que no serán utilizados en la prueba.
- **Stubs**: Objetos que son creados para sustituir un objeto que va a ofrecer uno o mas inputs de valor al método en prueba.

TDD – Test Driven Development

Test doubles – 2/2

- **Fakes:** Cuando hay dependencias que son mas complejas, por ejemplo cuando un determinado volumen de ítems en una compra van a motivar un descuento de valor, utilizamos los Fakes para poner la lógica necesaria para la prueba y evitar acceder a dependencias externas.
- **Mocks**: Verifican las salidas indirectas del método en prueba. Por ejemplo, ofrecen un sustituto para los objetos que contienen métodos que serán llamados por el método en prueba. Aquí tenemos un cambio de verificación de estados para verificación de comportamientos.

Ejercicio

En el componente “Calculadora” completar las otras tres operaciones básicas con TDD

TDD – Test Driven Development

Ejemplo practico - requerimiento

- **Como dueño de una casa de cambio, necesito obtener una forma de poder calcular el cambio de moneda desde peso chileno (CLP) a USD y EUR. Además, si se quiere enviar dinero, se aplicará un cobro adicional a la venta de USD o EUR (10%).**

CLP	USD	EUR
830	1	--
940	--	1

TDD – Test Driven Development

Ejemplo practico

TDD – Test Driven Development

Ejemplo – requerimientos ajustados

ExpertBank, por lanzamiento a la bolsa requiere implementar un sistema capaz de manejar la información relacionada con las cuentas de los clientes. Para esto se debe considerar:

- Cada cliente tiene su información personal básica (nombre, apellido, rut, correo, teléfono).
- Una cuenta puede ser corriente, de ahorro o vista.
- Los clientes del banco tienen beneficios en cuanto al monto total de la línea de crédito relacionada a la cuenta corriente.
- Puede tener un máximo de 5 cuentas asociadas al rut de cada cliente (hacer funciones de agregar y eliminar cuentas. En el caso de agregar, si supera las 5 cuentas, debe enviar un mensaje de error).
- Debe existir una función que me permita buscar un tipo de cuenta específica (ahorro, corriente o vista) asociada a un cliente dado por rut. En caso de no encontrar, mostrar mensaje asociado.

Supuestos: Los datos ingresado son válidos (rut y tipo de cuentas).

TESTING UNITARIO

Utilizando Angular – Conceptos Básicos