Testing I
Profesor: Daniel Tejerina

Módulo 5: Unit Testing

Sesión 13: Introducción al Debugging
Noviembre 16 de 2021

Debugging o Depuración:
Proceso de encontrar, analizar yremover las causas de fallos en el software.
Acción realizada por un desarrollador para encontrar un error y eliminarlo.

- Depuración por fuerza bruta: Es la más común, pero la menos eficiente. Consiste en cargar el programa con instrucciones de salida.

- Backtracking: Utilizado en pequeños programas. Se recorre el código hacia atrás hasta llegar a la causa del problema.

- Eliminación de causas. Se identifican y se aislan los datos que pueden ser las causas posibles del error.

Otras herramientas de los IDE:
- Los IDE permiten capturar errores típicos sin necesidad de compilar.
- Compiladores con depuración
- Ayudas dinámicas para la depuración
- Generadores automárticos de casos de prueba
- Herramientas de correlación de referencias cruzadas.

Breakpoint
Es un punto de interrupción en el código para detener la ejecución del programa en líneas específicas y analizar la situación del mismo.

Debug en Chrome
F11: Ejecución línea por línea
F8: Recorrer de un breakpoint a otro

Opciones de debug:

- Continue/Resume
Continúa con la ejecución del código hasta el siguiente breakpoint

- Step over
Ejecuta la siguiente línea de código; si se trata de una función, la ejecuta y retorna el resultado.

- Step into
Ejecuta la siguiente línea de código; si se trata de una función, entra a ejecutarla línea por línea.

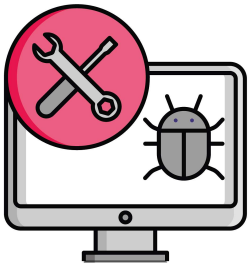
- Step out
Devuelve el debugger a la línea donde se llamó la función.

- Restart
Reinicia el debugger.

- Stop
Frena el debug.

Debug en Visual Studio Code
- Run > Start debugging (F5)
- Si es la primera vez que se va a depurar la aplicación, se requiere ir al menú "Add configuration…" y agregar la configuración de Chrome. En el archivo launch.json que se crea, configurar en "file" la dirección del archivo html de entrada.

Debugging vs Testing

Debugging	Testing
	
Proceso deductivo para corregir los errores encontrados durante las pruebas.	Proceso donde se comprueba que el sistema o componente funcione de acuerdo a lo esperado. Tiene como objetivo la búsqueda de errores.
Permite dar solución a la falla del código.	Permite identificar las fallas del código implementado.
Es la investigación y detección del error.	Es la visualización de errores.
Realizado generalmente por el programador o el desarrollador, a excepción del código generado por el tester como parte de los scripts de prueba automáticos.	Realizado generalmente por el tester.
No se puede realizar sin el conocimiento de diseño adecuado.	No es necesario tener conocimientos de diseño en el proceso de prueba.
Lo realiza únicamente un interno. Generalmente, un externo no puede depurar debido a que no debería tener acceso al código.	Puede ser realizado tanto por personas internas como externas.
Realizado en forma manual.	Puede ser manual o automatizado.
Basado en estrategias de debugging, como depuración por fuerza bruta, bug tracking, eliminación de causas, depuración automatizada utilizando herramientas y la mirada de un colega.	Basado en diferentes niveles de prueba, es decir, pruebas de componentes, pruebas de integración, pruebas del sistema, pruebas de aceptación.
No es una etapa del ciclo de vida del desarrollo de software, ocurre como consecuencia de las pruebas.	Es una etapa del ciclo de vida del desarrollo de software (SDLC).
Busca hacer coincidir el síntoma con la causa, lo que conduce a la corrección del error.	Compuesto por la validación y verificación del software.
Comienza con la ejecución del código debido a un caso de prueba fallido.	Iniciado antes de tener el código (testing estático) o después de que se escribe el código (testing dinámico).

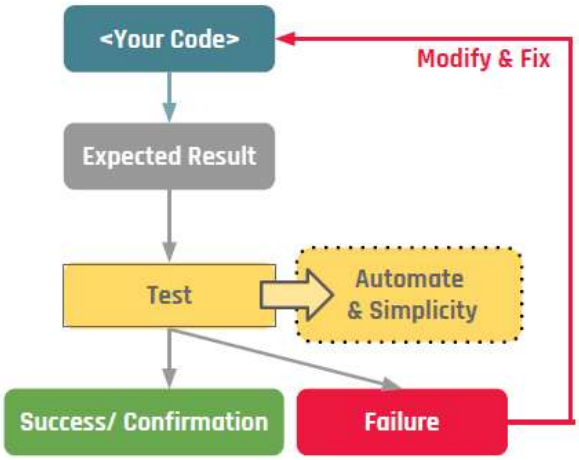
Sesión 14: Introducción a la prueba de componente
Noviembre 18 de 2021

Prueba de componente o prueba unitaria (Unit test)

Es la prueba de los componentes individuales de software. Son pequeños test creados específicamente para cubrir todos los requisitos del código y verificar sus resultados. Para generar estos test se utilizan técnicas de caja blanca.

Generalmente se trata de pruebas automatizadas escritas y ejecutadas por desarrolladores de software para garantizar que una sección de una aplicación —conocida como la "unidad"— cumpla con su diseño y se comporte según lo previsto. Una unidad puede ser una línea de código, un método o una clase.

Para la realización de Unit Tests se pueden requerir objetos simulados, virtualización de servicios, arneses, stubs y controladores.



Unit Test cuando no se utiliza TDD

En las metodologías ágiles, la prueba de regresión de componente automatizada garantiza que los cambios no han dañado los componentes existentes.

¿Qué se requiere para automatizar los Unit Test?

- Un Test Runner (Ej: Mocha): Herramienta que ejecuta los test y muestra los resultados en forma de reporte.
- Una Assertion Library (Ej: Chai): Herramienta que se utiliza para validar la lógica de prueba, las condiciones y resultados esperados.
- JEST es un framework que incluye ambas herramientas.

Frameworks más utilizados
- JUnit 5: Para Java, de uso gratuito.
- NUnit: Para lenguakes .NET, open-source.

- JMockit. Herramienta para Unit Test de código abierto. Permite hacer mocks de API.
- EMMA. Herramienta de código abierto para Java.
- PHPUnit. Herramienta de Unit Test para PHP.
- JEST. Para JavaScript.

Unit Tests en JavaScript

Para probar la ejecución de un bloque de código en node:

```
function suma(a, b) {
	return a + b;
}

console.assert(suma(1,2) == 3, “ERROR. El resultado es incorrecto”);
```

Para incluir un framework externo se requiere inicializar el proyecto:

```
npm init
```

Después de especificar las opciones solicitadas por la consola, se genera el archivo package.json, con las configuraciones básicas del proyecto.

```
{
	"name": "dividir",
	"version": "1.0.0",
	"description": "División de dos número",
	"main": "dividir.js",
	"scripts": {
		"test": "jest"
	},
	"keywords": [
		"test",
		"jest"
	],
	"author": "Mauricio Pineda",
	"license": "ISC"
}
```

En scripts se especifica el comando que se utilizará para ejecutar los tests (“jest”).

Para instalar jest:

```
npm install --save-dev jest
```

Al correr este comando se crea la carpeta node_modules y se agrega jest como dependencia.

Crear un caso de prueba con Jest:

- Exportar el módulo dividir:

```
module.exports dividir;
```

- Crear la carpeta __test__ en la raíz del proyecto

- Crear el archivo dividir.test.js e importar el módulo dividir

- Agregar el código del test:

```
const { expect } = require("@jest/globals"); // Esta línea se agrega automáticamente
const dividir = require("../dividir.js");

test("División entre dos números", () => {
        expect(dividir(4,2)).toBe(2);
})
```

- Ejecutar con el comando npm test. (o npm run test)

Sesión 15:

Sincrónico

Sesión 16. Prueba de componente.

Noviembre 23 de 2021.

Prueba y cobertura de sentencia
Cobertura de sentencia: Porcentaje de sentencias ejecutables que han sido practicadas por un juego de pruebas. Se escriben casos de prueba suficientes para que cada sentencia en el programa se ejecute (al menos) una vez. No asegura que se haya probado toda la lógica de decisión.

Cobertura (%) = (Sentencias ejecutadas / Sentencias ejecutables en el objeto de prueba) * 100

Prueba y cobertura de decisión
Es aquella prueba en la que se escriben casos de prueba suficientes para que cada decisión en el programa se ejecute una vez con resultado verdadero y otra con el falso. Una cobertura del 100% de decisión garantiza una cobertura del 100% de sentencia, pero no al revés.

Cobertura (%) = (Resultados de decisión ejecutados / Resultados de decisión en el objeto de prueba) * 100

TDD: Tests Driven Development (Desarrollo guiado por pruebas)

TDD es una práctica de desarrollo de software que consiste en escribir en primer lugar las pruebas unitarias, luego el código fuente que pase la prueba satisfactoriamente y, por último, refactorizar el código escrito. De esta manera, su lógica sigue el camino inverso al desarrollo tradicional, en el que habitualmente se codifica y, luego, se verifica el software.

TDD propone pensar y comprender primero el problema en su totalidad, antes de plantear la solución.

Ventajas del TDD:

- Se gana visibilidad al redactar primero los criterios sobre la totalidad del problema a solucionar. Luego procedemos a escribir el código.

- Facilita la tarea de resolver un problema a la vez (como plantean los marcos ágiles).

- Permite iterar una vez que se tiene un código base funcional.

- Libera la "presión" de escribir un código prolijo y performante al primer intento, dado que se prioriza la funcionalidad.

- Ayuda a trabajar en la precisión del código necesario (ni más, ni menos que lo que se requiere).

TDD vs BDD

Proceso del TDD (Desarrollo guiado por pruebas):

1. Se añade una prueba que capture el concepto del programador sobre el funcionamiento deseado de un pequeño fragmento de código.

2. Se ejecuta la prueba, que debería fallar, ya que el código no existe.

3. Se escribe el código y se ejecuta la prueba en un bucle cerrado hasta que la prueba pase.

4. Se refactoriza el código después de que la prueba haya sido exitosa, y se vuelve a ejecutar la prueba para asegurarse de que sigue pasando contra el código refactorizado.

5. Se repite este proceso para el siguiente pequeño fragmento de código, ejecutando las pruebas anteriores así como las pruebas añadidas.

Proceso del BDD (Desarrollo guiado por el comportamiento):

1. Se busca un lenguaje común, llamado lenguaje natural, para unir las especificaciones técnicas y los requisitos del cliente / negocio (historias de usuario), generalmente se utiliza Gherkin.

2. Se definen los criterios de aceptación de cada user story. Pueden utilizarse marcos de desarrollo guiados por el comportamiento (frameworks como Cucumber, Jbehave, Specflow) para definir criterios de aceptación basados en el formato dado - cuando - entonces (*Given - When - Then*).
- *Dado* un contexto inicial,
- *cuando* se produce un evento,
- *entonces*, se aseguran algunos resultados.

3. Se escribe el código del software de acuerdo a los criterios de aceptación estructurados.

4. Se genera el código para los casos de prueba, es decir, se implementa el comportamiento para cada línea en lenguaje natural.

5. Se ejecutan los casos de prueba y se refactoriza

Sincrónico

Para node, una suite son todos los suites que están dentro de un mismo archivo.
Para probar la entrada de datos, por ej. Selenium, (cypress?)

(Iván)
"Normalmente, las pruebas esas que dices se pueden hacer automatizadas, pero normalmente se hacen en pruebas de sistema y se hacen especialmente para cubrir las pruebas de regresión… Se simula que usuario ingrese datos (no siempre). Por ejemplo, se simula un usuario ingrese datos válidos y al darle clic el usuario pueda ver su perfil

Puedes probar esas cosas con frameworks como selenium o cypress

Se automatizan las pruebas de tal manera que con código se simulen las acciones que haría un usuario final, por ejemplo, en una página web, nada desde adentro del código, porque es caja negra. Entonces lo que se haría es que con código se abre ele browser, se va a página que quieres sestear y así sucesivamente"

Para crear el package con todos los valores por defecto:
`npm init -y`

Sesión 17 – Test Unitario

Noviembre 25 de 2021

Para correr los test autométicamente, se debe editar el package.json:

```
scripts: {
        // ...
        "test": "jest –watchAll"
}
```

Cobertura de pruebas con Jest

- Configuración de Jest para ejecutar el reporte de cobertura: Agregar la sentencia test:coverage en el archivo package.json:

```
scripts: {
        // ...
        "test": "jest",
        "test:coverage": "jest --coverage"
}
```

- Ejecutar el reporte de cobertura:

```
npm run test:coverage
```

Al ejecutar este comando, se presenta el reporte de cobertura en la consola,, pero además se genera la carpeta coverage, en la que se encuentra un archivo index.html con el detalle de cobertura de las pruebas.

Matchers más usados con Jest

.toBe

Para comparar valores primitivos:

```
expect(sumar(1,1)).toBe(2);
```

.toEqual

Para comparar objetos y todas sus propiedades:

expect(datos).toEqual(datos2);

.toBeLessThan, .toBeLessThanOrEqual
expect(restar(5,3)).toBeLessThan(3);

.toBeGreaterThan, .toBeGreaterThanOrEqual
expect(multiplicar(2,5)).toBeGreaterThanOrEqual(10);

.toBeTruthy
expect(isTrue).toBeTruthy();

.toBeFalsy
expect(isFalse).toBeFalsy();

.toBeUndefined
expect(isUndefined).toBeUndefined();

.toBeNull
expect(isNull).toBeNull();

.toContain (array)
expect(arrProvincias()).toContain('Madrid');

.toHaveLength (array)
expect(arrDias()).toHaveLength(7);

.toHaveLength (string)
expect(exp.responseFAIL).toHaveLength(13);

.toMatch
Comprueba que un texto coincida con una expresión regular:
expect(exp.email).toMatch(/^[a-zA-Z0-9._-]+@[a-zA-Z0-9.-]+\.([a-zA-Z]{2,4})+$/);

Otros matchers:
https://jestjs.io/docs/expect

Sincrónico

npm install –save-dev jest
La dependencia no se ejecuta en producción; por eso se instala como una dependencia de desarrollo únicamente.

test.only
Ignora los demás test y corre solamente uno

xtes

Ignora el test actual y corre los demás

npm run test -- -t “Happy path para resta”
El -- implica que el parámetro se pasa al archivo package
También se puede utilizar para correr una sola suite (un solo archivo de test)

Para que los test se ejecuten automáticamente
“test”: “jest --watchAll”

Para que se ejecuten todos los tests, independientemente de los archivos js que se están probando.
Crear un archivo jest.config.json en la raíz del proyecto

```
// jest.config.json
{
	“collectCoverageFrom”: [“*.{js, jsx}”]
}
```

en el package.json:
“test”: “jest --config-jest.config.json”

Al hacerlo se ejecutan los test de los dos archivos.

Configuración para utilizar el DOM:

Agregar en el package.json:
“jest”: {
	“testEnvironment”: “jsdom”
}

Sesión 18.

Sincrónico.

En la planilla de defectos, solo se presenta un resultado esperado general. En la planilla de casos de prueba, se pueden presentar resultados esperados para cada paso de ejecución.

const { init, limpiar, resetear, resolver } = require('../funcionalidad.js');
const fs = require('fs');

document.body.innerHTML = fs.readFileSync('calculadora.html')