Prueba de funcionalidad de librerias de JS y test libre

C. Daniela Michel Morales Santos, Universidad Autónoma de Guerrero, Chilpancingo, Guerrero, México, danie.morasantos@gmail.com

MC. Jorge Vázquez Galarce, Universidad Autónoma de Guerrero, Chilpancingo, Guerrero, México, 13216@uagro.mx

DR. Valentin Alvarez Hilario, Universidad Autónoma de Guerrero, Chilpancingo, Guerrero, México, 13701@uagro.mx

DR. Edgardo Solís Carmona, Universidad Autónoma de Guerrero, Chilpancingo, Guerrero, México, 09302@uagro.mx

RESUMEN

Dentro del desarrollo de la ingeniería de software se encuentra la etapa de codificación o programación, donde se ven culminados nuestros proyectos dentro de esta rama de la computación, así mismo dentro de este proceso está la revisión, afinación y testeo del código fuente, situación que es desconocida y no esta difundida en programadores nuevos. Desde la perspectiva de los programadores se debe asegurar que el proyecto funcione correctamente, y es mejor revisar y hacer correcciones (si las hay) antes de liberar los códigos. Y así evitar hacer correcciones a futuro, lo cual podría llevar un pérdida de tiempo y dinero. Debido a lo anterior se llevan a cabo pruebas unitarias donde se testea nuestro código y se detectan errores que puedan existir en los métodos, procedimientos, unidades etc. Lo que permite corregir de manera inmediata y eso nos ahorra tiempo, existen framework que nos ayudan a hacer los testeos a nuestro código, debemos elegir en base a nuestro conocimiento cual utilizar ya que son diferentes y realizan funciones distintas, en este proyecto se compararon dos librerías una llamada assert.js y la segunda chai.js. Ambas tienen similitudes en los métodos de aserciones, y en cuanto a la ejecución de los programas son diferentes, con node.js no es necesario correr las librerías desde la consola, solamente hacerlo de manera tradicional f5, con chai.js es más mecánico, así como definir un punto de entrada para iniciar y finalizar el programa. (S O M M E R V I L L E, 2011)

ABSTRACT

Within the development of software engineering is the coding or programming stage, where the works within this branch of computing are completed, likewise within this process is the review, refinement and testing of the source code, a situation that it is unknown and not widespread among new programmers. From the perspective of the programmers they must ensure that the project works correctly, and it is better to do them before releasing the codes. And thus avoid making corrections, or carrying out rework which translate into costs and loss of time. Due to the above, unit tests are carried out where our code is tested and errors that may exist in the modules, procedures, units, etc. are detected. Which allows us to correct immediately and that saves us time, there are frameworks that help us to test our code, we must choose based on knowledge which one to use since they are different and perform different functions, in this project two libraries were compared one called assert.js and the second chai.js. Both have similarities in the assertion methods, and in how the programs are executed they are different, with node.js it is not necessary to run the libraries from the console, just do it in the traditional way, with chai.js it is more mechanical, as well as defining an entry point to start and end the program.

PALABRAS RESERVADAS nodejs,

chai, Testeo, assert, npm,

KEYWORDS

nodejs, chai, Test, assert, npm,

INTRODUCCIÓN

Al codificar programas en algún lenguaje de programación es normal que encontremos errores en ellos, para ello existen pruebas automatizadas que nos ayudan a leer nuestro código línea a línea para así corregir errores que estos tengan, esta actividad se llama testeo. Estos testeos son de gran utilidad en nuestros proyectos. “En la década de 1990, hubo una transición de las pruebas a un proceso más completo llamado garantía de calidad, que cubre todo el ciclo de desarrollo de software y afecta los procesos de planificación, diseño, creación y ejecución de casos de prueba, soporte para casos existentes y pruebas”. (IBM, s.f.). Un tipo de pruebas a utilizar son las pruebas unitarias que tienen la característica de ejecutarse cuantas veces se desee, produciendo un código de mejor calidad, gracias a que tiene un proceso de afinación y depuración. Uno de los beneficios de las pruebas unitarias es que permiten generar una documentación de cada elemento llamado “función, módulo, procedimiento para complementar los entregables. El objetivo de la investigación es realizar las pruebas de testeo de dos librerías” y verificar si cumplen con las características que prometen sus desarrolladores. Específicamente se probaron las aserciones de nodejs y chai. Las librerías de testing están en constante actualización y son utilizadas por los desarrolladores de software, ya que por sí mismas representan una herramienta para la fase de depuración y prueba de software. Por lo que poner a prueba dos librerías de diferente marca representa un reto que permite averiguar sus características y comparar en condiciones reales o de producción de software.

El testing son pruebas que nos ayudan a mantener un control de calidad al momento de desarrollar software, lo cual evalúa la escritura y el análisis, al momento de detectar errores se envía un mensaje de los defectos que tiene nuestro desarrollo de manera temprana facilitando su resolución para así mismo entregar un producto de calidad y rendimiento.

La ventaja de hacer el testing es la depuración que recibe el código que permite corregir de manera inmediata los errores y la detección temprana de errores. Sin embargo, las pruebas unitarias pueden fallar, así que no siempre son fiables, puede arrojar falsas alarmas, estos fallos se deben a que tiene salidas inesperadas como, cambio de librerías, problemas de red, mala escritura etc.

Otro aspecto importante son las métricas asociadas al testing, que permite realizar estimaciones para futuros procesos de testeo para identificar áreas de riesgo más frecuentes y sea necesario realizar más pruebas, las métricas nos ayudan a identificar la complejidad de los componentes y de esa manera utilizar las mejores técnicas para resolverlo.

Proceso de pruebas consiste en integrase en un proceso definido y controlado cuya gestión se realiza por quienes están a cargo, en la cual debe estar documentado con sus respectivas pruebas ya que será una guía para el personal que están dentro del equipo de trabajo.

PLANTEAMIENTO DEL PROBLEMA

Cuando se programa la mayoría de las veces se tiene la idea de que el código que escribimos funciona bien y a la primera, lo cual es erróneo, y es posible que pueda presentar problemas al momento de poner en operación el programa. Para solucionar esta problemática se propone probar algunas funciones de las librerías de pruebas unitarias, en especial los métodos denominados assert de “node.js” y de “chai.js”, ambas librerías para el lenguaje de programación JavaScript. Y generar un comparativo de los resultados.

MÉTODO

Para el desarrollo de la investigación se hizo en primer término una recopilación documental e informativa sobre el testeo en general, y librerías de pruebas unitarias para JavaScript. En un segundo plano se seleccionaron los métodos o funciones de las librerías seleccionadas node.js y chai, esto en base a la similitud que guardan y verificar la forma de utilizarlas, los resultados que arrojan y finalmente se hicieron las pruebas e integraron los resultados para su análisis y verificar si la información documental coincide en la aplicación práctica es acertada.

Las librerías de node.js y chai poseen librerías y métodos en común, para ello se hicieron una comparación de ambas librerías al tener similitudes con algunos métodos y/o funciones se verifico si estos métodos cumplen con lo que prometen y

determinar si es útil incluirlas para el desarrollo de proyectos de programación. Para realizar esta investigación de recopilo información a través de libros de ingeniería de software, de testeo, en páginas de internet donde realzan el funcionamiento de cada uno de los métodos de pruebas unitarias, referencias de videos, una vez recapitulado toda la información necesaria se procedió a realizar las pruebas unitarias, se diseñó una plantilla para documentar dicho proceso y comparar el funcionamiento de cada uno de ellos en especial los assert de node.js y chai.js.

Nombre del método:
Sintaxis o signatura:
Definición de los parámetros:
Funcionalidad del método:
Valor devuelto o esperado:
Ejemplo de código:

Plantilla 1.0 permite documentar los métodos que se consideran en la evaluación:

DESARROLLO

1. Pruebas simples

El desarrollo de software conlleva una serie de pasos o etapas para la producción del software donde es posible que existan errores, estos debido a la intervención humana, los que interesan desde esta perspectiva son los que se presentan una vez que la codificación se ha hecho. Una forma de mitigar esta problemática es llevar a cabo pruebas del software o también llamadas test. Luego entonces las pruebas son una revisión final de las especificaciones, diseño y codificación.

El testing sirve para evaluar la calidad del desarrollo e identificar los defectos y problemas que este pude tener, algunas características que tienen estas pruebas es que son dinámicos, observables, de fácil control, y tienen un conjunto finito de casos de pruebas. El realizar testeos para un programa es ejecutarlo de manera controlada que nos permite ver el resultado que este arroja, si en uno de los casos detecta un error esto quiere decir que nuestro programa es incorrecto lo cual nos permitirá corregir nuestro programa de manera inmediata y nos ahorrará tiempo en el futuro.

1.1 Técnicas de prueba

Existen dos metodologías o estrategias básicas para el desarrollo del testing: las pruebas funcionales y las no funcionales, mismas que se detallas a continuación:

1.1.1 Pruebas funcionales

Se basan en las funcionalidades de un sistema que se describen en la especificación de requisitos, (es decir lo que ha de hacer el sistema), pueden o no estar documentadas, pero se requiere un nivel de experiencia elevado para interpretar estas pruebas. La funcionalidad representa la capacidad el producto de software para proporcionar funciones que satisfacen las necesidades declaradas e implícitas, cuando el producto se usa en condiciones específicas.

Testing estructural o caja blanca cae dentro de las pruebas funcionales, este tipo de testing se basa en que *hace y no lo que se supone que debe hacer* una vez ya teniendo las bases fundamentales nos daremos cuenta que usa las sentencias de tipo *if, case, while etc.* Este tipo de testing estructural se basa en testear en cuando todo el programa esté finalizado, y en muchas ocasiones no es posible poder generar los errores que este pueda llegar a tener un si se ha hecho algún cambio y pude ser necesario volver a calcular todos los casos, este tipo de testing no se puede encontrar algún error si no se ejecuta cada línea de código.

1.1.2 Pruebas no funcionales

Este tipo de pruebas tiene el comportamiento externo del software (como funciona el sistema) ya que se utilizan técnicas de diseño como por ejemplo “la caja negra”.

Testing basado en modelos o caja negra: Al ejecutar este tipo de prueba se basa en que no considera ningún tipo de detalle del como fue implementado el software si no partiendo de los documentos de requerimiento de los casos de prueba, es decir que este método de prueba se diseña a partir de la especificación de los requerimientos y de la selección de los datos de entrada. Por esta razón se lo llama a veces conducido por los datos. Consiste en la ejecución de tests con un conjunto de datos de entrada que ejercitan exhaustivamente al software bajo prueba. Debido a la imposibilidad de probar con las infinitas posibles variantes, se elaboran conjuntos de datos representativos de la totalidad, denominados clases de equivalencia

1.2 Niveles de prueba

Las pruebas o test se realizan en diferente nivel o profundidad, en base a dos criterios el objeto de las pruebas y el objetivo que se persigue con ellas. De acuerdo con el objeto de la prueba se dividen en tres niveles: prueba de unidad, de integración y de sistema. (S O M M E R V I L L E, 2011)

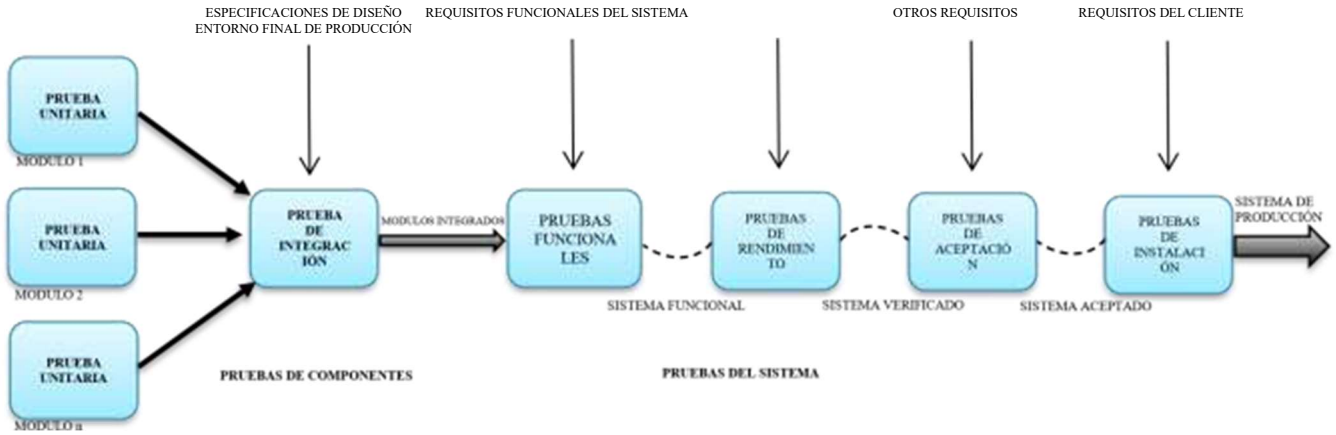


Figura 1. Procesos del desarrollo del software. (Sanchez Alonso, Silicia Urbán, & Rogríguez Garcia, Mayo 2012)

De acuerdo con el objetivo que se busca se dividen en pruebas de aceptación, de instalación, alfa y beta, de conformidad, y de regresión.

1.3 Pruebas de unidad

Esta prueba evalúa módulos concretos de un software donde convive con otros módulos, funciones, métodos es decir unidades de programación y cumple el objetivo de verificar que el funcionamiento aislado de dicho modulo cumple con la o las funcionalidades para el cual fue diseñado y construido. El objetivo de las pruebas unitarias es comprobar que los componentes individuales que integran el software funcionan bien.

Las pruebas de unitarias se llevan a cabo cuando los desarrolladores de software consideran que el código está listo para ser evaluado. Hoy en día las pruebas de los módulos de programación se planean y diseña por anticipado incluso antes de que sea escrito el programa e incluso esta actividad la lleva a cabo el mismo programador. (Alzate, 2021)

2. Biblioteca de pruebas

nodejs y chaijs son framework que nos permiten realizar pruebas unitarias esto quiere decir que nos ayudan a los procesos de validación del funcionamiento para nuestro proyecto o código fuente, ya que se encargaran de comprobar cada módulo que este tenga y validara que esté funcionando de manera adecuada. Existen diferentes tipos de framework o librerías cada una de ellas funciona de diferente manera, para ello es muy importante investigar cual es el adecuado para nuestro desarrollo. (johanmedia, 2022)

ChaiJS

Es una librería de testing de JavaScript rico en funciones que se ejecuta en Node.js estas pruebas son vitales para asegurar el buen funcionamiento de nuestro código o proyecto, chai.js tiene diferentes formas para crear estas pruebas y son **expect, should y assert**, cada uno de estas interfaces trabajan de diferente manera, la que veremos a continuación de assert, ya que nos ayuda a las afirmaciones que tenga nuestro código y verificar que este en declaración correcta.

NodeJS

Es un entorno de ejecución para JavaScript, que esta deseñado para crear aplicaciones web, pero nodejs está diseñado para hacer ejecuciones runtime sin necesidad de alguna biblioteca, en la mayoría de los casos de pruebas unitarias se necesita una llamada existente para correr nuestro programa**.**

2.1 Métodos seleccionados a probar

Los métodos seleccionados de ambas librerías se enlistan a continuación:

CHAI

Equal(): Afirma la igualdad no estricta

*notEqual():*Afirma la desigualdad no estricta

typeOf(): firma un valor de tipo carácter según lo determinado

lengthOf(): Afirma que un objeto tiene el mismo número de letras con el valor esperado.

*isUndefined():*Afirma que el valor no está definido.*

*isObject():*Afirma que el valor es un objeto de tipo 'Objeto'.*

*isTrue():*Afirma que un valor es verdadero.* *notMatch():* Afirma que el valor no coincide con la expresión regular*

NodeJS *deepEqual():*Comprueba si dos valores son iguales *deepStrictEqual():*Comprueba si dos valores son iguales, utilizando el operador de igualdad estricta (===)

*fail():*arroja un error de aserción *ifError():*Lanza un error especificado si el error especificado se evalúa como verdadero *doesNotMatch():* Espera que la cadena de entrada no coincida con la expresión regular.

*isNotString():*Afirma que un valor no es una cadena de caracteres. *assert.ok():* Se declara una propiedad si se evalúa que es cierto manda un mensaje de si pasa*

assert.throws(): La función debe regresar true para indicar todas las validaciones internas pasadas de lo contrario, fallará con un AssertionError* *assert.notEqual():*Lanza un mensaje de sigualdad.*

2.2 Plantillas de los métodos assert de Node.JS:

Nombre del método: deepEqual() Comprueba si dos valores son iguales **Sintaxis o signatura** assert.deepEqual(actual, expected[, message]) **Definición de los parámetros:** *actual <any> expected <any> message <string> | <Error> <NoError>*

Funcionalidad del método: Comprueba si dos valores son iguales

Valor devuelto o esperado: se comprueba si los valores de la igualdad son reales de ser así se manda un mensaje que en la funcionalidad de método no presenta error, de lo contrario presentara un mensaje de error de aserción **Ejemplo:**

```
const assert = require('assert').strict;      try
{      assert.deepEqual({ a: '7' }, { a: '7'
});
   console.log("No hay error")
} catch(error) {
   console.log("Error: ", error)
```

PLANTILLA 2.0 Nos muestra un ejemplo de los parámetros que se espera del método *deepEqual* con assert Node.JS.

Nombre del método: deepStrictEqual()
Sintaxis o signatura assert.deepStrictEqual(*valor1, valor2*, *message*); **Definición de los parámetros:**

Valor1 <any> Valor2
<any> message <string> | <Error>
<NoError>

Funcionalidad del método: Comprueba si dos valores son iguales, utilizando el operador de igualdad estricta (===)
Valor devuelto o esperado: se comprueba si dos objetos o valores son iguales usando el operador == , de lo contario tendrá un error de aserción y finalizara el programa.
Ejemplo:

```
const assert = require('assert').strict; try
{  assert.deepStrictEqual({ a: 1 }, { a: '1'
});
} catch(error) {
console.log("Error: ", error) }
```

PLANTILLA 2.1 Nos muestra un ejemplo de los parámetros que se espera del método *deepStrictEqual* con assert Node.JS.

Nombre del método: doesNotMatch()
Sintaxis o signatura assert.doesNotMatch(string, regexp[, message]) **Definición de los parámetros:**

String <any> regexp
<any> message <string> | <Error>
<NoError>

Funcionalidad del método: esta función devuelve un error de aserción del tipo de objeto.
Valor devuelto o esperado: espera que la entrada de cadena no coincida con la expresión regular. Si la condición es verdadera, no producirá una salida; de lo contrario, se generará un error de afirmación.
Ejemplo:

```
const assert = require('assert').strict;  try
{
   assert.doesNotMatch('hará el recorrido', /fail/);
console.log("No se reprodujo error")
} catch(error) {
   console.log("Error:", error) }
```

PLANTILLA 2.2 Nos muestra un ejemplo de los parámetros que se espera del método *doesNotMatch* con assert Node.JS.

Nombre del método: ifError()-
Sintaxis o signatura assert.ifError(value) **Definición de los parámetros**: value <any>
message <string> | <Error> <NoError>
Funcionalidad del método: Lanza un error especificado de tipo objeto, si se evalúa como verdadero o falso. Valor devuelto o esperado: Devuelve un valor verdadero o falso.
Ejemplo:

```
const assert = require('assert').strict; try
{
assert.ifError(null);
   console.log("No se reprodujo error")
} catch(error) {
   console.log("Error:", error) }
```

PLANTILLA 2.3 Nos muestra un ejemplo de los parámetros que se espera del método *ifError* con assert Node.JS.

Nombre del método: fail()-arroja un error de aserción

Sintaxis o signatura: assert.fail([message]) **Definición de los parámetros:**
message <string> | <Error> <NoError>
Funcionalidad del método: Este parámetro contiene el mensaje de error de cadena o tipo de error.
Valor devuelto o esperado: evalúa la cadena, para mandar un mensaje de error, ya sea predeterminado o de forma maual.
Ejemplo:

```
const assert = require('assert').strict;   try {
   assert.fail(); } catch(error)
{
   console.log("Error:", error) }
```

PLANTILLA 2.4 Nos muestra un ejemplo de los parámetros que se espera del método *fail* con assert Node.JS.

2.3 Plantillas de los métodos assert de Chai.JS

Nombre del método: assert.equal();
Sin taxis o signatura: assert.equal(actual, expected, [message]) **Definición de los parámetros:**
Actual <any> Expected
<any>
message <string> | passing
Funcionalidad del método: esta función compara si las funciones son iguales
Valor devuelto o esperado: se espera que la función asignada sea igual que el resultando, de ser igual manda un mensaje de que ha pasado **Ejemplo:**

```
const assert = require('chai').assert; const
main = require('../main'); const
helloworld = main.HelloWorld();
describe('Main Suite', function(){
   it('Es como se esperaba hello world',function (){
assert.equal(helloworld,'hello world');     })  })
```

PLANTILLA 3.0 Nos muestra un ejemplo de los parámetros que se espera del método *assert.equal* con asert Chai.JS.

Nombre del método: assert.notEqual();
Sintaxis o signatura: assert.notEqual(actual, expected, [message]) **Definición de los parámetros:**
Actual <any> Expected
<any>
message <string> | passing
Funcionalidad del método: compara si las funciones son iguales manda un mense de error
Valor devuelto o esperado: se espera que la función asignada diferente, que los valores no se **Ejemplo:** const assert = require('chai').assert; const main = require('../main'); var textos =

```
'hello world';
const helloWorldText = main.HelloWorld();  describe('Menu',
function(){
   it('Los textos no son iguales', function(){        assert.notEqual(helloWorldText,
textos)
   });
})
```

PLANTILLA 3.1 Nos muestra un ejemplo de los parámetros que se espera del método *assert.notEqual* con assert chai.js.

Nombre del método: assert.typeOf()
Sintaxis o signatura: *assert.typeOf(value, name, [message])* **Definición de los parámetros:**
Actual <any> Value
<any> message <string> |
passing
Funcionalidad del método: compara si las funciones si es una cadena de caracteres
Valor devuelto o esperado: se espera que se lea la función y que afirme si es una cadena de caracteres **Ejemplo:** const assert = require('chai').assert; const main = require('../main'); var textos =

```
'hello world';
const helloWorldText = main.HelloWorld();
describe('Menu', function(){    it('Es
un string',function(){
assert.typeOf(textos, 'string')
});
})
```

PLANTILLA 3.2 Nos muestra un ejemplo de los parámetros que se espera del método *assert.typeOf* con assert chai.js.

Nombre del método: assert.lenghOf()
Sintaxis o signatura: *assert.lengthOf(object, length, [message])* **Definición de los parámetros:** *Objet <any>*
Length <number>
message <string> | passing
Funcionalidad del método: devuelve el número de caracteres de una cadena
Valor devuelto o esperado: esta función lee una cadena y compara el número real con el valor esperado
Ejemplo:

```
const assert = require('chai').assert; const
main = require('../main'); const
helloworld = main.HelloWorld();
describe('Main Suite', function(){
it('La frase tiene una longitud de 11', function(){
assert.lengthOf(helloWorldText,11)
   });
})
```

PLANTILLA 3.3 Nos muestra un ejemplo de los parámetros que se espera del método *assert.notEqual* con assert chai.js.

Nombre del método: assert.isNotString()
Sintaxis o signatura: *assert.isNotString(value, [message])*
Definición de los parámetros: *Value <any> message <string> | passing*
Funcionalidad del método: devuelve el número de caracteres de una cadena
Valor devuelto o esperado: esta función lee una cadena y compara el número real con el valor esperado
Ejemplo:

```
const assert = require('chai').assert;
const main = require('../main'); const
helloworld = main.HelloWorld();
var teaOrder = 4;
describe('Main Suite', function(){ it('No es
un estring', function(){
assert.isNotString(teaOrder, textos);
   });
})
```

PLANTILLA 3.4 Nos muestra un ejemplo de los parámetros que se espera del método *assert.isNotString* con assert chai.js.

3. Comparativo de bibliotecas de JavaScript

Existen diferentes frameworks para realizar pruebas unitarias en diferentes lenguajes de programación, los más utilizados son los que llevan a cabo la ejecución y comprobación de las pruebas de forma automatizada. Para las pruebas es necesario definir los casos de prueba y las colecciones de prueba.

Los casos de prueba: son los métodos que ejecutan los métodos o funciones de una librería o clase, la cual es el objeto de la prueba. Los casos de prueba se pueden estructurar en colecciones o conjuntos lo que permite ejecutar las pruebas asociadas de manera fraccionada o todo en un conjunto. De acuerdo con el avance del desarrollo de la aplicación o programa, se crean un conjunto de tests, con un grado mayor de complejidad y buscando la reducción de errores.

Las colecciones de pruebas: son un conjunto de casos de prueba sobre clases funcionalmente relacionadas. Estas colecciones se pueden estructurar a fin de organizar y dar un orden a las mismas. (IBM, s.f.)

3.1 Desarrollo de las pruebas o testing

Node.js es un código abierto de JavaScript, diseñado para generar páginas web de forma optimizada, y tiene diferentes módulos en especial tiene el node:assert o aserciones, este proporciona un conjunto de funciones de aserción para realizar afirmaciones, verificaciones o comprobaciones. Los assert tiene el modo de afirmación estricto y elegante, en los métodos no estrictos se comportan como sus correspondientes métodos estrictos.

Para este caso se utilizó la suite de aserciones llamadas assert de node.js y chai.js. El proceso de testeo se realizó de la siguiente manera: se definió el conjunto de funciones a evaluar de ambas librerías, se definieron los casos de prueba y finalmente se realizó la ejecución de cada función. (Haminton, 2022)

Para el caso de los assert node.js se probaron de manera individual cada método, esto con programas de JavaScript independientes, en cuanto a chai.js se probaron mediante un *case*:

```
const assert =
require('assert').strict;

// Function call
try {
    assert.deepStrictEqual({ a: 1
}, { a: '1' });
} catch(error) {
    console.log("Error: ", error)
}
```

```
const assert = require('chai').assert;

const main = require('../main');
var textos = 'hello world';
var teaOrder = 4;

const helloWorldText = main.HelloWorld();

describe('Menu', function(){
it('Es un string',function(){
assert.typeOf(textos, 'string')
});
it('Ambos textos coinciden', function(){
assert.equal(helloWorldText, textos)
});

})
```

Figura 2. Fragmentos de código con el que se probó los métodos node.js y chai.js.

3.2 Casos de prueba

Se seleccionaron cinco métodos de cada librería de acuerdo con su utilidad y similitud. Estos se muestran en la tabla siguiente, donde se describe de forma breve.

NodeJS 18 LT Assert	Descripción del método	Chai v4 3.7 Assert	Descripción del método
deepEqual()	Comprueba si dos valores son iguales	Equal()	Afirma la igualdad no descrita
deepStrictEqual()	Comprueba si dos valores son iguales	notEqual()	Afirma la desigualdad no estricta
doesNotMatch()	Espera que la cadena de entrada no coincida con la expresión regular.	typeOf()	Firma un valor de tipo carácter según lo determinado
Fail()	arroja un error de aserción	LengOf()	Afirma que un objeto tiene el mismo número de letras con el valor esperado.

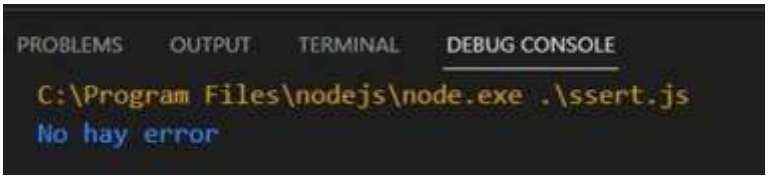
ifError()	Lanza un error especificado si el error especificado se evalúa como verdadero	IsnotString()	Afirma que un valor no es una cadena de caracteres.

Tabla 1. Métodos seleccionados a probar con pruebas unitarias.

3.3 Plantilla para pruebas de assert de Node.JS

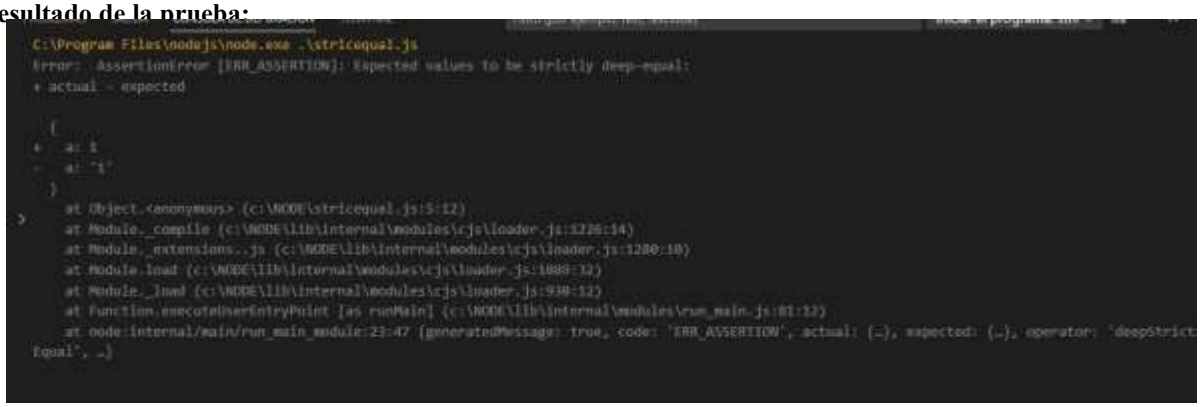
Las colecciones de pruebas y su ejecución se muestran en las figuras siguientes, mismas que se definieron en una plantilla. Primero se muestran la realizadas para node.js y finalmente las chai.js.

Nombre de la prueba: "PRUEBA DE FUNCIONALIDAD BASICA"
Nombre del método: *deepEqual()*
Valor de entrada: { a: '7' }{ a: '7' }
Valor esperado: 'No hay error' '7'
Resultado de la prueba:



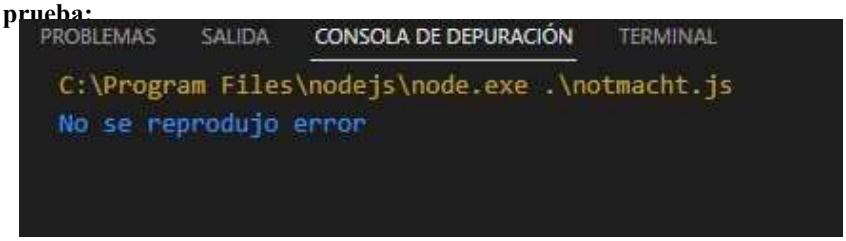
Plantilla 4.0 Nos muestra los resultados ya ejecutados de las librerías que se eligieron para hacer las pruebas con assert *deepEqual* node.js.

Nombre de la prueba: "PRUEBA DE FUNCIONALIDAD BASICA"
Nombre del método: deepStrictEqual()
Valor de entrada: { a: '1' }{ a: '1' }
Valor esperado: Error de desigualdad
Resultado de la prueba:



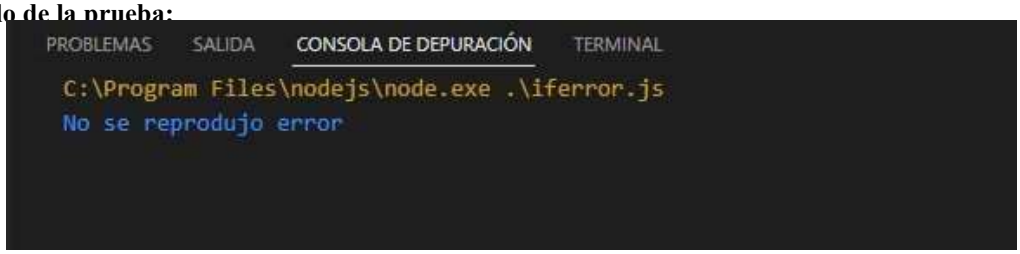
Plantilla 4.1 Nos muestra los resultados ya ejecutados de las librerías que se eligieron para hacer las pruebas con assert *deepStripEqual* node.js.

Nombre de la prueba: "PRUEBA DE FUNCIONALIDAD BASICA"
Nombre del método: doesNotMatch()
Valor de entrada: cadena 'I will try to pass'
Valor esperado: No hay error
Resultado de la prueba:



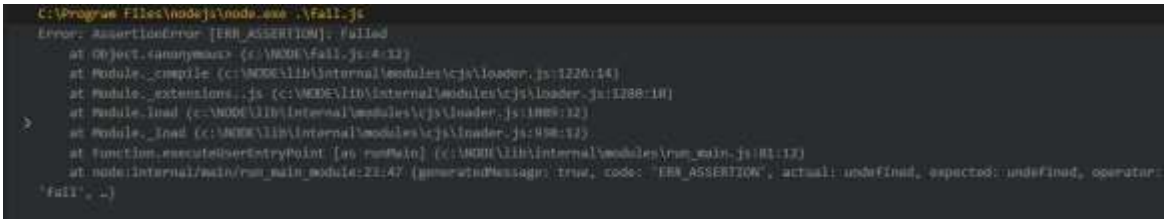
Plantilla 4.2 Nos muestra los reultados ya ejecutados de las librerías que se eligieron para hacer las pruebas con assert *doesNotMatch* node.js.

Nombre de la prueba: "PRUEBA DE FUNCIONALIDAD BASICA"
Nombre del método: ifError()
Valor de entrada: (null);
Valor esperado: No hay error
Resultado de la prueba:



Plantilla 4.3 Nos muestra los resultados ya ejecutados de las librerías que se eligieron para hacer las pruebas con assert *ifError* node.js.

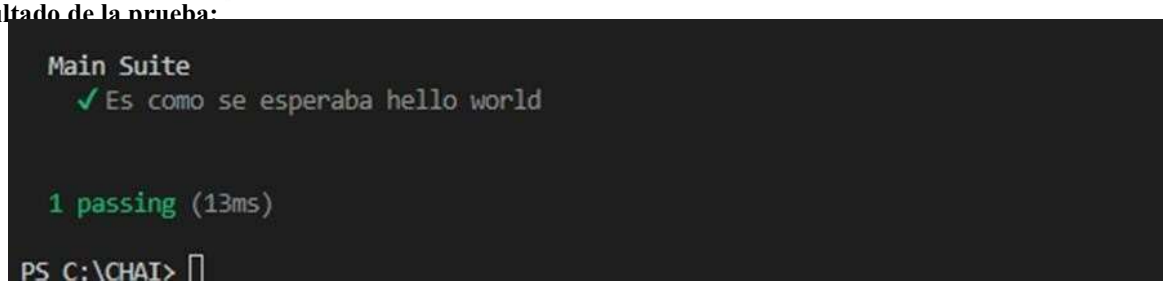
Nombre de la prueba: "PRUEBA DE FUNCIONALIDAD BASICA"
Nombre del método: fail();
Valor de entrada: 'null'
Valor esperado: Error de assersión
Resultado de la prueba:



Plantilla 4.4 Nos muestra los resultados ya ejecutados de las librerías que se eligieron para hacer las pruebas con assert *fail* node.js.

3.4 Plantilla para pruebas de assert de Chai

Nombre de la prueba: "PRUEBA DE FUNCIONALIDAD BASICA"
Nombre del método: *assert.equal*
Valor de entrada: var textos = ‹hello world›;
Valor esperado: Paso la prueba de funcionalidad
Resultado de la prueba:



Plantilla 5.0 Nos muestra los resultados ya ejecutados de las librerías que se eligieron para hacer las pruebas con assert *equal* chai.js.

Nombre de la prueba: "PRUEBA DE FUNCIONALIDAD BASICA"
Nombre del método: *assert.notEqual*
Valor de entrada: var textos = ‹hello world›;
Valor esperado: error de aserción
Resultado de la prueba:

Plantilla 5.1 Nos muestra los resultados ya ejecutados de las librerías que se eligieron para hacer las pruebas con assert *notEqual* chai.js.

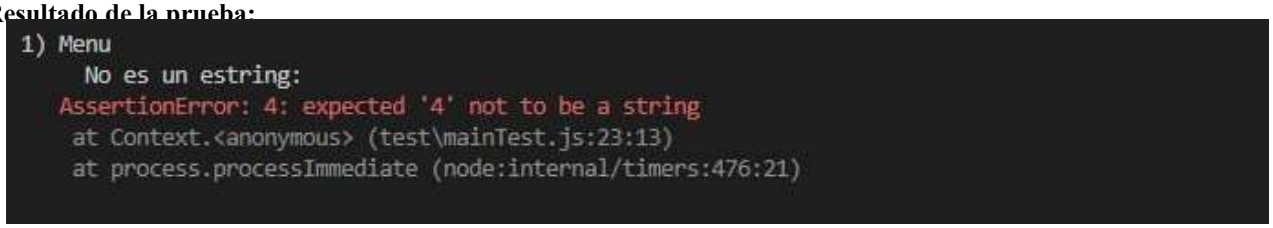
Nombre de la prueba: "PRUEBA DE FUNCIONALIDAD BASICA"
Nombre del método: *assert.typeOf()*
Valor de entrada: var textos = 'hello world';
Valor esperado: Paso la prueba de funcionalidad
Resultado de la prueba:

Plantilla 5.2 Nos muestra los resultados ya ejecutados de las librerías que se eligieron para hacer las pruebas con assert *typeOf* chai.js

Nombre de la prueba: "PRUEBA DE FUNCIONALIDAD BASICA"
Nombre del método: assert.lenghOf()
Valor de entrada: var textos = 'hello world';
Valor esperado: Tiene el mismo número de caracteres
Resultado de la prueba:

Plantilla 5.3 Nos muestra los resultados ya ejecutados de las librerías que se eligieron para hacer las pruebas con assert *lenghOf* chai.js

Nombre de la prueba: "PRUEBA DE FUNCIONALIDAD BASICA"
Nombre del método: assert.isNotString()
Valor de entrada: var textos = 'hello world'; var teaOrder = 4;
Valor esperado: No es una cadena de caracteres
Resultado de la prueba:



Plantilla 5.4 Nos muestra los resultados ya ejecutados de las librerías que se eligieron para hacer las pruebas con assert *isNorString* chai.js

RESULTADOS

De acuerdo con lo realizado durante esta investigación se obtuvieron los resultados que se resumen en las tablas 2 y 3 donde se muestra la cantidad de eventos que se llevaron a cabo y el comportamiento de las funciones probadas. En todos los casos las pruebas demuestran que cumplen con el resultado esperado de acuerdo con la documentación del desarrollador. La forma de presentar los resultados de los de la librería chai.js, son más comprensibles sobre todo a usuarios nuevos, a diferencia de las de node.js donde si dan el resultado correcto pero se tiene que hacer más trabajo para el formateo o presentación. **Resultados cuantitativos de las pruebas**

Método	Librería		Numero de pruebas realizado	Valores de entrada	Valor esperado	Resultado de la prueba
	NodeJs	ChaiJs				
deepEqual()	√	√	10	{ a: '7' } { a: '7' }	No hay error.	Satisfactorio

deepStrictEqual()	√	x	5	{ a: '1' } { a: '1' }	Error de aserción.	Satisfactorio
doesNotMatch()	√	x	5	cadena 'I will try to pass'	No ha error.	Satisfactorio
Fail()	√	x	10	'null'	Error de aserción.	Satisfactorio
ifError()	√	x	5	(null)	No hay Error.	Satisfactorio
Equal()	√	√	5	'null'	Error de aserción.	Satisfactorio
notEqual()	x	√	5	var textos = 'hello world';	No paso la prueba de funcionalidad.	Satisfactorio
typeOf()	x	√	5	var textos = 'hello world';	Paso la prueba de funcionalidad.	Satisfactorio
lenghOf()	x	√	5	var textos = 'hello world';	Paso la prueba de funcionalidad.	Satisfactorio
IsNotString()	x	√	5	var textos = 'hello world'; var teaOrder = 4;	No paso la prueba de funcionalidad.	Satisfactorio

Tabla 2. Resultados de los métodos evaluados.

Resultados cualitativos de las pruebas

En la tabla 3 se muestran los resultados de las características que se buscó evaluar de node.js y chai.js

Característica que se evaluó	NodeJS	Chai	Observación
Facilidad de implementación de la librería	√	√	Para ejecutar el programa NodeJS está disponible para ejecutar desde consola y mpn, chai solo por mpn

Claridad en los nombres de los métodos	x	√	Chai es más claro con el nombre de sus métodos
Claridad en la documentación de los métodos	√	x	Los errores de aserción con más claros y específicos con NodeJS
Diversidad en el paso de argumentos a los métodos	√	√	Ambos presentan con diferentes métodos para hacer pruebas
Legibilidad en la presentación de resultados	√	x	NodeJS te detecta en línea de aserción, en cuanto a chai solo muestra si el método el funcional o no

Tabla 3. Resumen de características evaluadas

DISCUSIÓN

Para realizar las pruebas unitarias se usaron las librerías node.js y chai.js, se compararon estas librerías debido a que las aserciones de ambas tienen un gran parecido. Para utilizar node.js, se instala una librería, para la ejecución de los programas se utiliza el comando npm o se lleva a cabo de manera manual, en ambos casos se ejecutan los programas sin problema, dando respuesta de las aserciones propuestas en los códigos, en el caso de Chai se deben instalar dos librerías adicionales para poder mostrar los errores de aserción y se requiere una clase especial main donde deben residir las funciones para lograr ejecutar las pruebas. Chai solo está disponible por npm de alguna otra manera no se podrá ejecutar nuestro programa, node.js y chai.js al ser tan parecidos, el tipo de sintaxis para evaluar los errores son diferentes ya que ambos tienen diferente manera de escritura, lo importante de todo es que ambas librerías funcionan para crear pruebas unitarias.

CONCLUSIONES

Las pruebas unitarias tienen una gran funcionalidad al momento de programar, para ello debemos tener en claro ¿que son las pruebas unitarias? Y ¿para qué sirven? Una vez visto esto nos daremos cuenta de que las pruebas unitarias nos muestra los errores que tiene nuestro código para la corrección inmediata, en este proyecto se usaron dos librerías las cuales son node.js y Chai JS que son framework para JavaScript que fueron diseñados para detección de errores, así mandando mensajes de error de aserción, estas dos librerías son las que nos mostraron los errores de aserción que tenía nuestro código, node.js es fácil usar sus librerías de aserciones, solo es necesario ejecutar el programa y con un solo botón nos muestra si nuestro código está funcionando bien o tiene errores, en cambio chai.js únicamente está diseñado para ser ejecutado a través de npm y tener una clase main para realizar las pruebas, por lo que si eres principiante se recomienda documentarse sobre el funcionamiento de las librerías. Estas dos librerías node.js y chai.js se compararon porque en las aserciones son muy parecidas y son muy utilizadas por desarrolladores. Y estas permiten depurar y mejorar los desarrollos antes de ser puestos en producción y lograr que lleguen con el menor número de errores funcionales.

REFERENCIAS

Alzate, E. B. (2021). IMPLEMENTACIÓN DE PRUEBAS UNITARIAS. 15.

Haminton, T. (29 de Octubre de 2022). *Guru99*. Obtenido de https://www.guru99.com/software-testingintroductionimportance.html

IBM. (s.f.). *What is software testing?* Obtenido de https://www.ibm.com/topics/software-testing johanmedia.
(19 de Diciembre de 2022). *PRUEBAS con CHAI y MOCHA js*. Obtenido de https://www.youtube.com/watch?v=-sn3H0V3PuY&t=742s

S O M M E R V I L L E. (2011). INGENIERÍA DE SOFTWARE. En Addison-Wesley, *INGENIERÍA DE SOFTWARE* (pág. 792). MEXICO: PEARSON.

Sanchez Alonso, S., Silicia Urbán, M. A., & Rogríguez Garcia, D. (Mayo 2012). INGENIARIA DEL SOFTWARE. En S. Sanchez Alonso, M. A. Silicia Urbán, & D. Rogríguez Garcia. Madrid España: Alfaomega Grupo Editor, S.A de C.V.,México.