

Instituto Tecnológico de Costa Rica

Curso:

CE4101: Especificación y Diseño de Software

Investigación #1:

Unit Testing

Profesor:

Daniel Eduardo Madriz Huertas

Integrantes:

Carlos Adrián Araya Ramírez – 2018319701

Michael Shakime Richards Sparks – 2018170667

Periodo:

I semestre 2022

Repositorio de GitHub para los ejemplos:

https://github.com/FlowsyCurls/2022_ESPE_UnitTesting

Año:

2022

Índice

Unit Testing	3
Unit testing para Java con el framework: JUnit 5	4
Funcionalidades	5
Tipos de pruebas	5
Ejemplo de uso.....	7
Unit testing para Javascript con el framework: Jest.....	19
Características	19
Funcionalidades	19
Tipos de pruebas	20
Ejemplo de uso.....	22
Conclusiones	30
Bibliografía	31

Unit Testing

Las pruebas unitarias son pruebas automatizadas de unidades individuales o grupos de unidades relacionadas. Se consideran una práctica común en la que los desarrolladores escriben casos de prueba junto con el código normal y es un procedimiento más de los que se llevan a cabo dentro de la metodología ágil de trabajo.

Los frameworks de automatización como JUnit para Java han popularizado este enfoque, lo que permite la ejecución frecuente y automática de conjuntos de pruebas unitarias. [1] A pesar de las evaluaciones de las pruebas unitarias en la práctica, los investigadores de ingeniería de software ven potencial de mejora e investigan técnicas avanzadas como la generación automatizada de pruebas unitarias. Una de las principales razones es debido a que la mayoría de los códigos no triviales son difíciles de probar de forma aislada. Es difícil evitar escribir conjuntos de pruebas que sean complejos, incompletos y difíciles de mantener e interpretar [2].

Actualmente, las pruebas unitarias son indispensables, prácticamente uno de los temas más discutidos a nivel industrial en el desarrollo de software, puesto que brindan beneficios a la hora de revelar errores de implementación, aunque a su vez requieren más tiempo de desarrollado inicial para una característica determinada y un mayor nivel de habilidad de los miembros en los equipos de trabajo.

La correcta implementación de pruebas unitarias contempla una estructura siguiendo las tres A's del Unit Testing: *Arrange* (Preparar), *Act* (Actuar), *Assert* (Afirmar). Dentro de las ventajas de esto se encuentran [3]:

- Una mejora en la calidad final del código, al realizarse pruebas de manera continua con un formato concreto, el código producto será nítido, comprensible y de calidad.
- Las pruebas unitarias bien implementadas sirven como documentación del proyecto.
- La posibilidad de realizar cientos de pruebas en poco tiempo. Así como verificar la lógica del código.
- Garantizan un seguimiento en la funcionalidad del código ya probado. Al efectuar cambios o refactorizaciones de código más adelante es fácil saber si un método específico provoca un error, pues se escriben casos de prueba para todas las funciones y métodos.

En las empresas estas pruebas se utilizan como complemento para afianzar la calidad del software mediante la ejecución de un programa y el descubrimiento de errores. Sin embargo, la calidad del

software sigue siendo un problema, las masas aún argumentan la necesidad de impulsar aún más la automatización en las pruebas, incluso para generar automáticamente pruebas unitarias [6].

Las pruebas unitarias tienen ciertas limitaciones, más aún dentro del procedimiento de la metodología agile, donde es seguro que habrá impactos en el diseño inicial una vez abordada la implementación. A los desarrolladores de software a menudo les resulta difícil realizar pruebas sistemáticas y automatizadas debido a razones como las incertidumbres inherentes al modelo [4]. Por otra parte, para que sean funcionales estas pruebas requieren de un ajuste continuo, para estructuras de datos y algoritmos que funcionan como caja negra, las pruebas unitarias son esenciales, no obstante, al tratar con algoritmos que tienden a ser alterados o ajustados, utilizar pruebas unitarias puede generar una gran inversión de tiempo prácticamente injustificada [5].

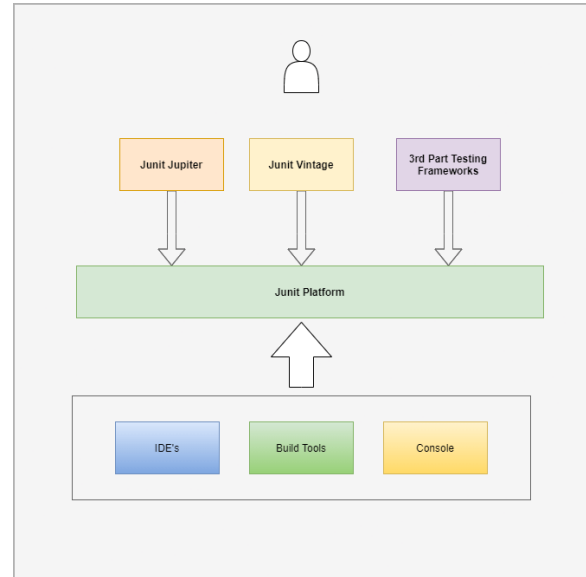
Adicionalmente, no todos los problemas de software se pueden resolver con un enfoque TDD (Test Driven Development), por tanto, su uso en todos los casos no es lo más práctico. Ahora bien, se pueden utilizar, acudiendo al uso de testeo MOCK o STUB, que permiten que se escriban pruebas unitarias para todo, simplifican la estructura de prueba y evitan contaminar el código de dominio con la infraestructura de prueba, es posible, pero eso involucra complejidad adicional y por consiguiente mayor inversión de tiempo y recursos [3].

Unit testing para Java con el framework: JUnit 5

JUnit se trata de un Framework Open Source para la automatización de las pruebas en los proyectos de software en el lenguaje de programación Java. Este framework provee al usuario de herramientas, clases y métodos que le facilitan la tarea de realizar pruebas en su sistema y así asegurar su consistencia y funcionalidad.



Desde la versión JUnit 5 no existe una única biblioteca, se encuentra división en tres subproyectos: JUnit Platform, JUnit Jupiter y JUnit Vintage. Platform es la base que permite el lanzamiento de los frameworks de prueba en la JVM, Jupiter es la combinación del nuevo modelo de programación y el modelo de extensión para escribir pruebas en JUnit 5, y Vintage, proporciona un TestEngine para ejecutar pruebas basadas en JUnit 3 y JUnit 4 en la plataforma, para sistemas con versiones antiguas. [7]



Funcionalidades

Por cada clase del software que se desea probar existe una clase de pruebas que contiene al menos un método de prueba por cada método de la clase. Es necesario tener en cuenta que el nombre de la clase de prueba debe tener la siguiente estructura: "Test". El método de prueba invoca lo que se va a probar y luego se utilizan aserciones que son como afirmaciones para verificar resultar esperados. Adicionalmente, hay distintas formas para compartir información entre las pruebas, por ejemplo, los escenarios de las pruebas y también se pueden crear suites de pruebas para organizar y ejecutar pruebas fácilmente.

La ejecución en JUnit de un conjunto de pruebas se hace de manera gráfica y textual. Si todas las pruebas pasan, el resultado en la ventana de ejecución será de color verde y si alguna falla será de color rojo. [8]

JUnit también ofrece la posibilidad de realizar las pruebas de regresión, que se efectúan cuando una parte del código ha sido modificada y es necesario comprobar su correcto funcionamiento. [9]

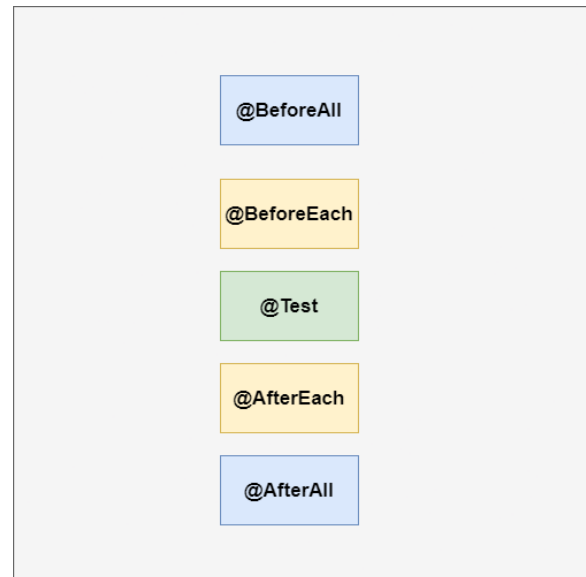
Tipos de pruebas

Las clases de prueba analizadas por JUnit Jupiter presentan una estructura que permite cuatro posibles tipos de métodos:

- **Método setUp:** Asigna valores iniciales a variables antes de la ejecución de cada prueba. Para que se inicialicen al principio una vez, el método se debe llamar "setUpClass"
- **Método tearDown:** Es llamado después de cada test y puede servir para liberar recursos o similar. Para sólo se llame al final de la ejecución de todos los test, se debe llamar "tearDownClass"
- **Métodos Test:** Contienen las pruebas concretas a realizar.
- **Métodos auxiliares:** Ayudan en la evaluación de las pruebas a los métodos de testeo. Están destinadas a manejar subproblemas u operaciones específicas.

Hay una enorme lista de anotaciones para identificar, ejecutar y admitir muchas funciones para los métodos de prueba, con el fin de simplificarle el trabajo al programador. Las anotaciones son palabras reservadas que se colocan delante de los métodos de prueba, que indican a las librerías JUnit instrucciones concretas a realizar, algunas de las más relevantes son [10]:

- **@Test:** Indica a Junit que se trata de un método de prueba. En versiones anteriores de JUnit los métodos debían tener un nombre con la siguiente estructura: "Test". Con esta notación colocada delante de los métodos podemos elegir el nombre libremente.
- **@BeforeAll:** Indica que el método de prueba se ejecutará antes de que cualquiera de los métodos @Test se ejecute dentro de la clase Test.
- **@BeforeEach:** Indica que el método de prueba se ejecutará antes que cada método @Test en la clase Test.
- **@AfterEach:** Indica que el método de prueba se ejecutará después de cada método @Test en la clase Test.
- **@AfterAll:** Indica que el método de prueba se ejecutará después de que todos los métodos @Test se ejecuten en la clase Test.



- **@DisplayName:** Permite dar un nombre más amistoso tanto a clases como a métodos, así las herramientas que nos muestran los tests lo hacen con el nombre de la anotación en vez de con el propio nombre de la clase o el método.

Anteriormente se puntuó la utilización de aserciones que se manejan para verificar resultados en las pruebas, estas herramientas se encuentran incluidas en la clase Assert, en la documentación de JUnit se pueden encontrar de forma detallada todas las funciones, dentro de las más utilizadas se encuentran [12]:

- **assertArrayEquals:** verifica que los arreglos esperado y real sean iguales.
- **assertTrue and assertFalse:** Sirven para la verificación de una condición, si esta es verdadera o falsa, respectivamente.
- **fail:** Fuerza a la prueba a fallar y mostrar un mensaje determinado, sirve para indicar métodos con un desarrollo incompleto.

Ejemplo de uso

El código que se desea probar es de una aplicación para la administración de contactos. Esta aplicación consta de dos clases principales: **Contact.java**, que contiene la información del contacto (nombre, apellido y teléfono) y **ContactManager.java**, que es la clase principal que administra la información de agenda y las operaciones sobre esta, ya sea agregar, eliminar, etc. El código fuente de ambas clases se muestra en las siguientes figuras:

```

1  import java.util.Collection;
2  import java.util.Map;
3  import java.util.concurrent.ConcurrentHashMap;
4
5  public class ContactManager {
6
7      Map<String, Contact> contactList = new ConcurrentHashMap<String, Contact>();
8
9      public void addContact(String firstName, String lastName, String phoneNumber) {
10         Contact contact = new Contact(firstName, lastName, phoneNumber);
11         validateContact(contact);
12         checkIfContactAlreadyExist(contact);
13         contactList.put(generateKey(contact), contact);
14     }
15
16     public Collection<Contact> getAllContacts() {
17         return contactList.values();
18     }
19
20     private void checkIfContactAlreadyExist(Contact contact) {
21         if (contactList.containsKey(generateKey(contact)))
22             throw new RuntimeException("Contact Already Exists");
23     }
24
25     private void validateContact(Contact contact) {
26         contact.validateFirstName();
27         contact.validateLastName();
28         contact.validatePhoneNumber();
29     }
30
31     private String generateKey(Contact contact) {
32         return String.format("%s-%s", contact.getFirstName(), contact.getLastName());
33     }
34 }

```

```

1 public class Contact {
2     private String firstName;
3     private String lastName;
4     private String phoneNumber;
5
6     public Contact(String firstName, String lastName, String phoneNumber) {
7         this.firstName = firstName;
8         this.lastName = lastName;
9         this.phoneNumber = phoneNumber;
10    }
11
12    public String getFirstName() {
13        return firstName;
14    }
15
16    public void setFirstName(String firstName) {
17        this.firstName = firstName;
18    }
19
20    public String getLastName() {
21        return lastName;
22    }
23
24    public void setLastName(String lastName) {
25        this.lastName = lastName;
26    }
27
28    public String getPhoneNumber() {
29        return phoneNumber;
30    }
31
32    public void setPhoneNumber(String phoneNumber) {
33        this.phoneNumber = phoneNumber;
34    }
35
36    public void validateFirstName() {
37        if (this.firstName == null)
38            throw new RuntimeException("First Name Cannot be null");
39    }
40
41    public void validateLastName() {
42        if (this.lastName == null)
43            throw new RuntimeException("Last Name Cannot be null");
44    }
45
46    public void validatePhoneNumber() {
47        if (this.phoneNumber.length() != 10) {
48            throw new RuntimeException("Phone Number Should be 10 Digits Long");
49        }
50        if (!this.phoneNumber.matches("\\d+")) {
51            throw new RuntimeException("Phone Number Contain only digits");
52        }
53        if (!this.phoneNumber.startsWith("0")) {
54            throw new RuntimeException("Phone Number Should Start with 0");
55        }
56    }
57 }

```

Ahora bien, para la iniciar las pruebas, se crea una clase tipo test *ContactManagerTest.java*

Testing

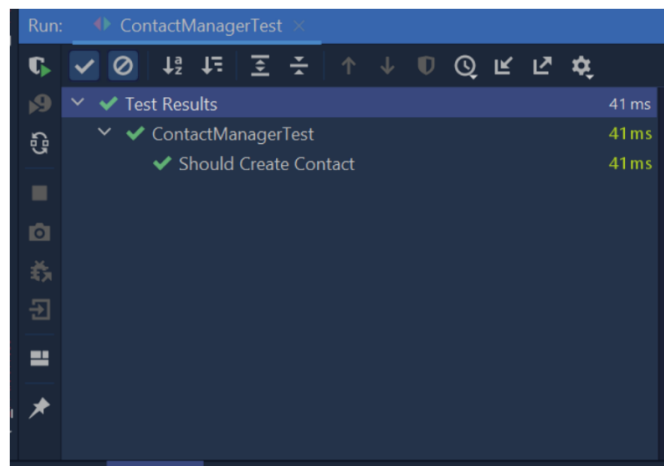
```

1 import org.junit.jupiter.api.DisplayName;
2 import org.junit.jupiter.api.Test;
3
4 import static org.junit.jupiter.api.Assertions.*;
5
6 public class ContactManagerTest {
7
8     @Test
9     @DisplayName("Should Create Contact")
10    public void shouldCreateContact() {
11        ContactManager contactManager = new ContactManager();
12        contactManager.addContact("John", "Doe", "0123456789");
13        assertFalse(contactManager.getAllContacts().isEmpty());
14        assertEquals(1, contactManager.getAllContacts().size());
15    }
16 }

```


Dentro de esta clase irán todas las pruebas referentes a la agenda. Por tanto, como primera prueba se pretende agregar un contacto. No olvidar la anotación `@Test` para indicar a JUnit que el método de prueba. Se utiliza la anotación `@DisplayName` para dar un nombre conveniente a la prueba que se esté perpetrando, en este caso “Should Create Contact”, pues esto es lo que hace.

Para evaluar el correcto funcionamiento primero se crea una instancia de la clase `ContactManager` y luego llama al método `addContact()` con la información correspondiente como parámetro, la idea de esto es mas adelante comprobar si se agrega correctamente, en el código se muestran dos posibles formas haciendo uso de los métodos `assert` mencionados arriba. La primera opción es llamar al método `getAllContacts()` que retorna una lista con los contactos del administrador, si se cumple que la agenda no está vacía, se evidencia que se agregó el contacto, de igual forma la segunda opción sería evaluar la igualdad entre el tamaño de la lista con 1, si ambos valores son iguales se cumple la funcionalidad.



JUnit también permite realizar pruebas que no deberían salir bien para poder validar excepciones, en el caso de este código existen ciertas validaciones a tomar en cuenta con la información ingresada por el usuario en cada contacto.

- El nombre ni apellido pueden ser nulos.
- El número de teléfono debe tener exactamente 10 dígitos de largo, contener solo dígitos y empezar con 0.

```
19     @Test
20     @DisplayName("Should Not Create Contact When First Name is Null")
21     public void shouldThrowRuntimeExceptionWhenFirstNameIsNull() {
22         ContactManager contactManager = new ContactManager();
23         Assertions.assertThrows(RuntimeException.class, () -> {
24             contactManager.addContact(null, "Doe", "0123456789");
25         });
26     }
```

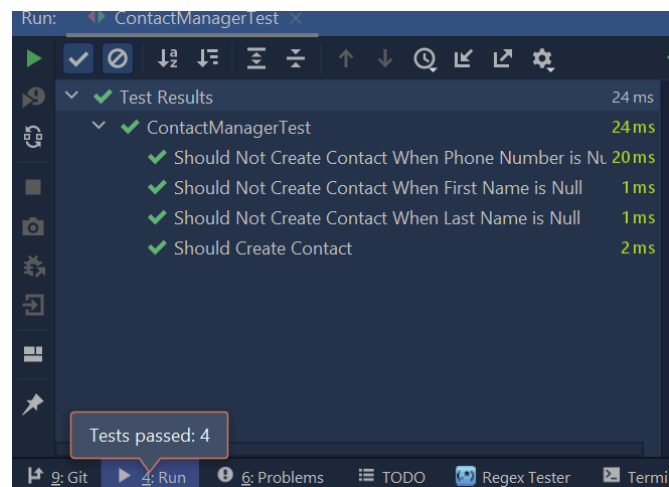
Excepciones de prueba usando assertThrows()

En este caso la prueba es ver cómo se comporta la aplicación cuando la validación del nombre no se cumple. El método `assertThrows()` permite tomar el tipo de excepción como primer parámetro y el ejecutable que arroja la excepción como segundo parámetro.

Se puede realizar de igual manera para los otros parámetros:

```
28     @Test
29     @DisplayName("Should Not Create Contact When Last Name is Null")
30     public void shouldThrowRuntimeExceptionWhenLastNameIsNull() {
31         ContactManager contactManager = new ContactManager();
32         Assertions.assertThrows(RuntimeException.class, () -> {
33             contactManager.addContact("John", null, "0123456789");
34         });
35     }
36
37     @Test
38     @DisplayName("Should Not Create Contact When Phone Number is Null")
39     public void shouldThrowRuntimeExceptionWhenPhoneNumberIsNull() {
40         ContactManager contactManager = new ContactManager();
41         Assertions.assertThrows(RuntimeException.class, () -> {
42             contactManager.addContact("John", "Doe", null);
43         });
44     }
```

Al ejecutar todos los resultados deberían estar en verde para comprobar un correcto funcionamiento, no tendría sentido que alguno de los contactos se haya creado aún cuando se incumple la validación en alguno de los parámetros.

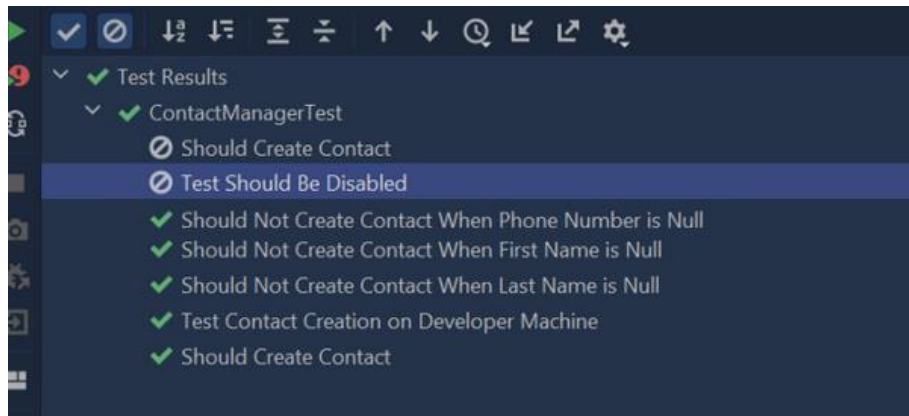


Deshabilitar pruebas

Es posible deshabilitar pruebas utilizando la anotación `@Disabled`.

```
1  @Test
2  @DisplayName("Test Should Be Disabled")
3  @Disabled
4  public void shouldBeDisabled() {
5      throw new RuntimeException("Test Should Not be executed");
6  }
```

Cuando se ejecute la clase de prueba, aquellos métodos en particular que se encuentren deshabilitados saldrán en gris y no se verificarán.



Fases del ciclo de ejecución de las pruebas

Cada Prueba pasa por diferentes fases como parte de su ejecución, cada fase está representada por las anotaciones citadas anteriormente. Las anotaciones **@BeforeAll**, **@BeforeEach**, son utilizadas principalmente para la configuración del entorno, acciones de inicialización. Mientras que las anotaciones **@AfterAll**, **@AfterEach**, son utilizadas para limpieza de datos o el entorno.

Por ejemplo, para poder realizar un testeo, en cada prueba es necesario tener una instancia de la clase `ContactManager` a la cual se le aplica el método, por tanto, para evitar repetir código es recomendable mantener una instancia estática de la clase que se ejecute antes de cada prueba y se almacene en el entorno, para esto se usa la anotación `@BeforeEach`.

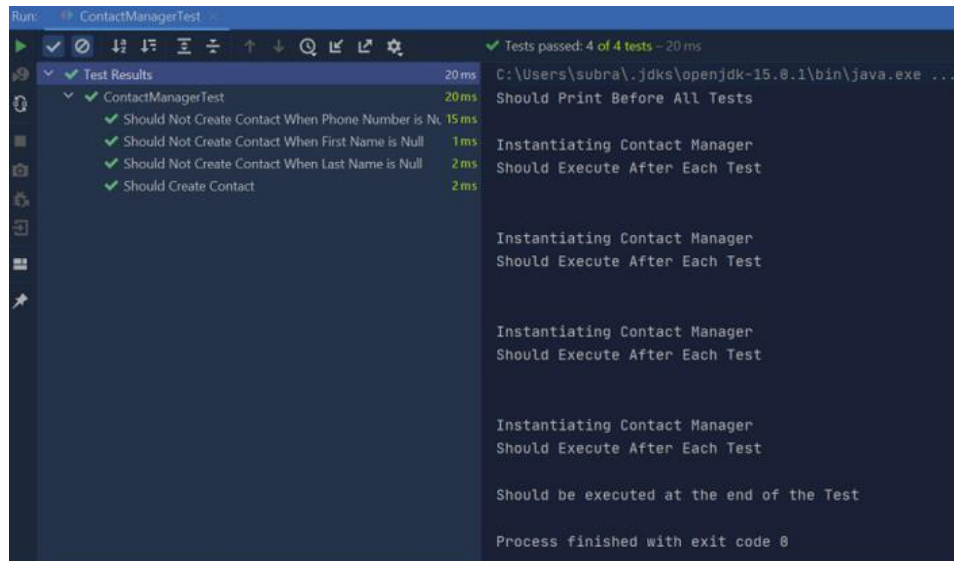
Las anotaciones `@AfterAll`, `@AfterEach` y `@BeforeAll` no tiene una utilidad específica, no obstante, para evidenciar su utilidad y funcionamiento simplemente se imprime un texto que indica lo que se está haciendo.

```

1  import org.junit.jupiter.api.Assertions;
2  import org.junit.jupiter.api.BeforeAll;
3  import org.junit.jupiter.api.DisplayName;
4  import org.junit.jupiter.api.Test;
5
6  import static org.junit.jupiter.api.Assertions.assertEquals;
7  import static org.junit.jupiter.api.Assertions.assertFalse;
8
9  public class ContactManagerTest {
10
11     private static ContactManager contactManager;
12
13     @BeforeAll
14     public static void setup() {
15         System.out.println("Instantiating Contact Manager before the Test Execution");
16         contactManager = new ContactManager();
17     }
18
19     @Test
20     @DisplayName("Should Create Contact")
21     public void shouldCreateContact() {
22         contactManager.addContact("John", "Doe", "0123456789");
23         assertFalse(contactManager.getAllContacts().isEmpty());
24         assertEquals(1, contactManager.getAllContacts().size());
25     }
26
27     @Test
28     @DisplayName("Should Not Create Contact When First Name is Null")
29     public void shouldThrowRuntimeExceptionWhenFirstNameIsNull() {
30         ContactManager contactManager = new ContactManager();
31         Assertions.assertThrows(RuntimeException.class, () -> {
32             contactManager.addContact(null, "Doe", "0123456789");
33         });
34     }
35
36     @Test
37     @DisplayName("Should Not Create Contact When Last Name is Null")
38     public void shouldThrowRuntimeExceptionWhenLastNameIsNull() {
39         ContactManager contactManager = new ContactManager();
40         Assertions.assertThrows(RuntimeException.class, () -> {
41             contactManager.addContact("John", null, "0123456789");
42         });
43     }
44
45     @Test
46     @DisplayName("Should Not Create Contact When Phone Number is Null")
47     public void shouldThrowRuntimeExceptionWhenPhoneNumberIsNull() {
48         ContactManager contactManager = new ContactManager();
49         Assertions.assertThrows(RuntimeException.class, () -> {
50             contactManager.addContact("John", "Doe", null);
51         });
52     }
53 }

```

En la ejecución se evidencia la funcionalidad de las anotaciones con las salidas de la siguiente imagen, el método llamado **tearDown()** se ejecuta al final de cada prueba, indicando con el texto “Should Execute After Each Test”. Asimismo el método llamado **tearDownAll()** se ejecuta al final de todas las pruebas, indicando con el texto “Should be executed at the end of the Test”. El método **setUp()** con se ejecuta antes de cada una de las pruebas. Y el método **setUpAll()** se ejecuta antes de todas las pruebas, indicando con el texto “Should Print Before All Tests”.



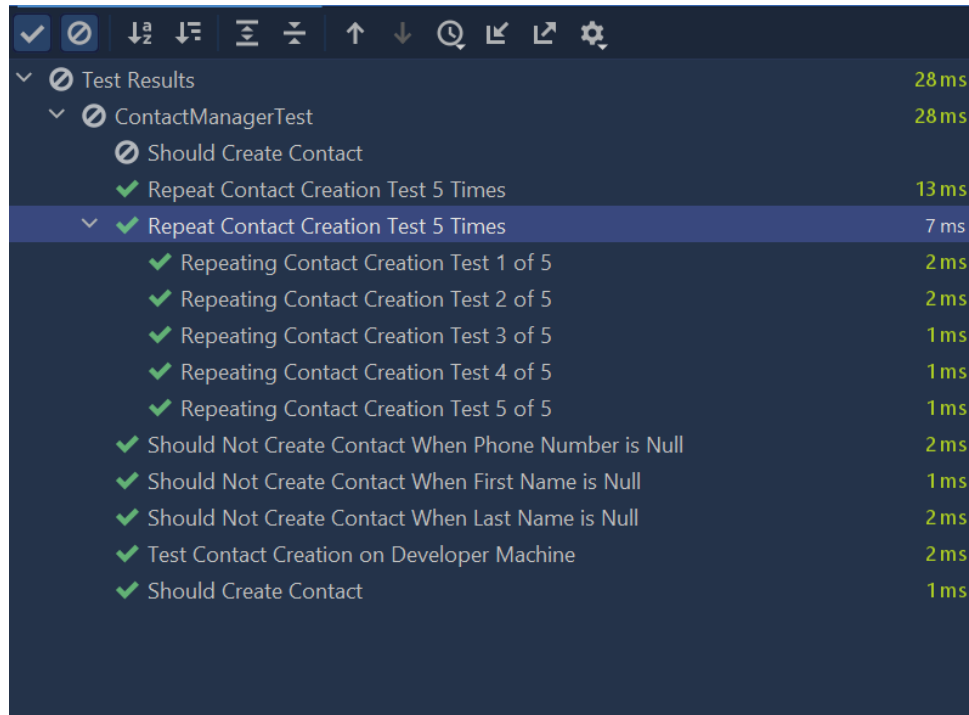
Pruebas repetidas

Cuando se desea probar funciones que incorporan aleatoriedad de valores en su implementación es recomendable ejecutar varias veces la prueba para cubrir más posibles casos y asegurarse de una correcta funcionalidad.

La anotación **@RepeatedTest** se usa para indicar que el método correspondiente es un método de plantilla de prueba que debe repetirse una cantidad específica de veces con un nombre para mostrar configurable. [7]

```
@DisplayName("Repeat Contact Creation Test 5 Times")
@RepeatedTest(value = 5,
    name = "Repeating Contact Creation Test {currentRepetition} of {totalRepetitions}")
public void shouldTestContactCreationRepeatedly() {
    contactManager.addContact("John", "Doe", "0123456789");
    assertFalse(contactManager.getAllContacts().isEmpty());
    assertEquals(1, contactManager.getAllContacts().size());
}
```

Cuando ejecuta la prueba anterior, puede ver que se ejecuta 5 veces la primera prueba:



✓	Test Results	28 ms
✓	ContactManagerTest	28 ms
✓	Should Create Contact	
✓	Repeat Contact Creation Test 5 Times	13 ms
✓	Repeat Contact Creation Test 5 Times	7 ms
✓	Repeating Contact Creation Test 1 of 5	2 ms
✓	Repeating Contact Creation Test 2 of 5	2 ms
✓	Repeating Contact Creation Test 3 of 5	1 ms
✓	Repeating Contact Creation Test 4 of 5	1 ms
✓	Repeating Contact Creation Test 5 of 5	1 ms
✓	Should Not Create Contact When Phone Number is Null	2 ms
✓	Should Not Create Contact When First Name is Null	1 ms
✓	Should Not Create Contact When Last Name is Null	2 ms
✓	Test Contact Creation on Developer Machine	2 ms
✓	Should Create Contact	1 ms

Pruebas parametrizadas

La anotación **@ParameterizedTest** se usa para indicar que el método anotado es un método de prueba parametrizado. [7]

Para mostrar un ejemplo, se utilizará la prueba mostrada a continuación, que verifica el formato de un numero ingresado con respecto a las validaciones dichas antes.

```
@Test
@DisplayName("Phone Number should start with 0")
public void shouldTestPhoneNumberFormat() {
    contactManager.addContact("John", "Doe", "0123456789");
    assertFalse(contactManager.getAllContacts().isEmpty());
    assertEquals(1, contactManager.getAllContacts().size());
}
```

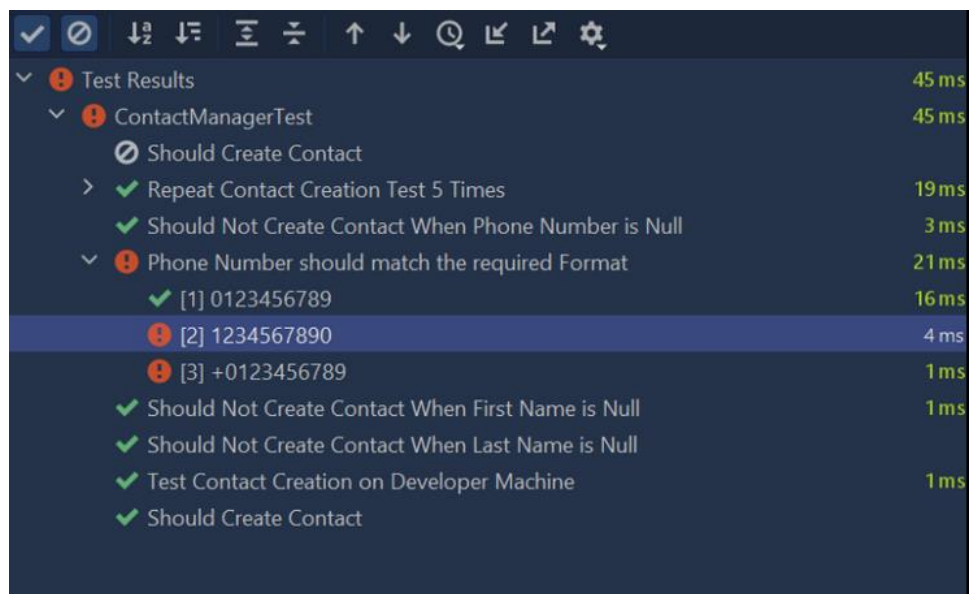
Las pruebas parametrizadas se pueden ejecutar varias veces para distintos valores, existen distintas formas de pasar estos valores:

Utilizando la anotación **@ValueSource** que permite proporcionar un conjunto de literales de cadena, largos, dobles y flotantes como parámetro para nuestra prueba. Un ejemplo de la

sintaxis para un parámetro de entrada de tipo string, sería `@ValueSource` (`strings = {"string1","string2","string3"}`). Significa que se va a realizar la prueba tres veces para esos tres valores distintos de strings. En la siguiente imagen se muestra la variación de la prueba de chequeo del número con la anotación `@ValueSource`.

```
@DisplayName("Phone Number should match the required Format")
@ParameterizedTest
@ValueSource(strings = {"0123456789", "1234567890", "+0123456789"})
public void shouldTestPhoneNumberFormat(String phoneNumber) {
    contactManager.addContact("John", "Doe", phoneNumber);
    assertFalse(contactManager.getAllContacts().isEmpty());
    assertEquals(1, contactManager.getAllContacts().size());
}
```

Se presentan tres números con formato distinto, y se sabe que de la prueba se esperan tres intentos y solamente el primero debería revelar un testeo correcto, como se muestra a continuación.



Otra forma de pasar valores parametrizados es utilizando la anotación `@MethodSource` que proporciona acceso a los valores devueltos por los métodos de fábrica de la clase en la que se declara esta anotación o por los métodos de fábrica estáticos en clases externas a las que hace referencia el nombre de método completo. [7]

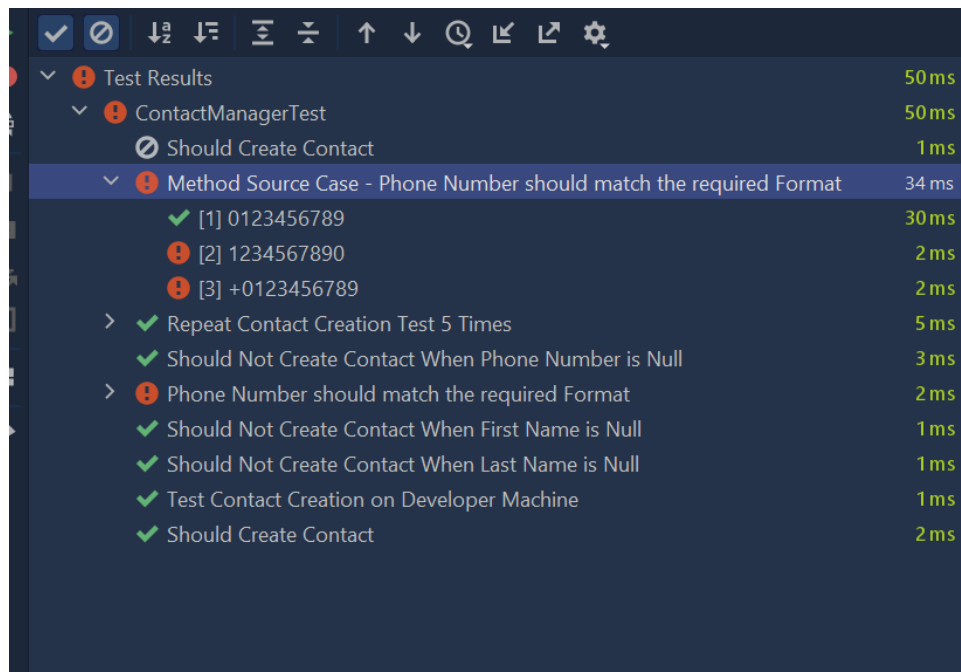

```

@DisplayName("Method Source Case - Phone Number should match the required Format")
@ParameterizedTest
@MethodSource("phoneNumberList")
public void shouldTestPhoneNumberFormatUsingMethodSource(String phoneNumber) {
    contactManager.addContact("John", "Doe", phoneNumber);
    assertFalse(contactManager.getAllContacts().isEmpty());
    assertEquals(1, contactManager.getAllContacts().size());
}

private List<String> phoneNumberList() {
    return Arrays.asList("0123456789", "1234567890", "+0123456789");
}

```

En esta prueba se declara un método de tipo auxiliar *phoneNumberList()* para la creación de una lista con los números a analizar. Para poder proporcionar la entrada a la prueba parametrizada, hay que referir en la anotación el nombre del método auxiliar que se creó. Seguidamente se muestra la salida de la prueba, que es igual a la que se obtuvo con el método anterior.



También existen las anotaciones `@CsvSource` que permite al método de prueba leer valores separados por comas de uno o más registros CSV, y `@CsvFileSource` que permite referir a un archivo CSV como tal con la sintaxis `@CsvFileSource(resources = "/data.csv")`. Ambos casos se muestran a continuación, donde el contenido del data.csv es el mismo que el de los ejemplos anteriores:


```

@DisplayName("CSV Source Case - Phone Number should match the required Format")
@ParameterizedTest
@CsvSource({"1,0123456789", "2,1234567890", "3,+0123456789"})
public void shouldTestPhoneNumberFormatUsingCSVSource(String phoneNumber) {
    contactManager.addContact("John", "Doe", phoneNumber);
    assertFalse(contactManager.getAllContacts().isEmpty());
    assertEquals(1, contactManager.getAllContacts().size());
}

@DisplayName("CSV File Source Case - Phone Number should match the required Format")
@ParameterizedTest
@CsvFileSource(resources = "/data.csv")
public void shouldTestPhoneNumberFormatUsingCSVFileSource(String phoneNumber) {
    contactManager.addContact("John", "Doe", phoneNumber);
    assertFalse(contactManager.getAllContacts().isEmpty());
    assertEquals(1, contactManager.getAllContacts().size());
}

```

Al igual que para los métodos de ValueSource y MethodSource, la salida es la misma porque los valores son equivalentes. Es solo para ejemplificar las diferentes formas de parametrización un método de pruebas.

Test Name	Duration
Test Results	50 ms
ContactManagerTest	50 ms
Should Create Contact	1 ms
Method Source Case - Phone Number should match the required Format	34 ms
[1] 0123456789	30 ms
[2] 1234567890	2 ms
[3] +0123456789	2 ms
Repeat Contact Creation Test 5 Times	5 ms
Should Not Create Contact When Phone Number is Null	3 ms
Phone Number should match the required Format	2 ms
Should Not Create Contact When First Name is Null	1 ms
Should Not Create Contact When Last Name is Null	1 ms
Test Contact Creation on Developer Machine	1 ms
Should Create Contact	2 ms

Nested

Es normal tener pruebas parecidas, la anotación @Nested se usa para indicar que la clase anotada es una clase de prueba anidada y no estática (es decir, una clase interna) que puede compartir la configuración y el estado con una instancia de su clase adjunta. La clase envolvente puede ser una clase de prueba de nivel superior u otra clase de prueba @Nested, y la anidación puede ser

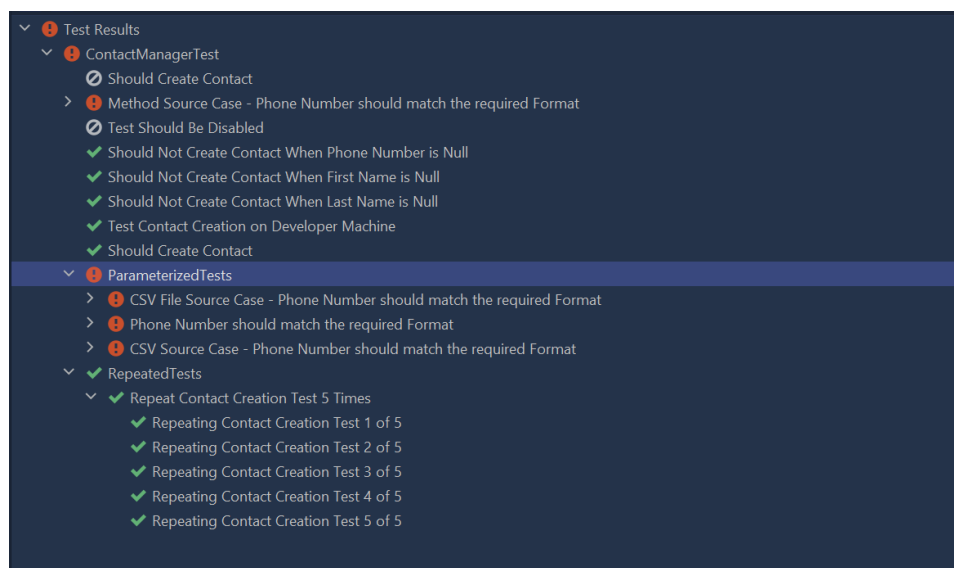
arbitrariamente profunda. [7] La idea es tomar un grupo de pruebas parecidas y organizarlas en categorías, se crea una sola clase con esta anotación. A continuación, se muestra el resultado agrupando todas las pruebas parametrizadas implementadas dentro de una misma clase:

```
@Nested
class ParameterizedTests {
    @DisplayName("Phone Number should match the required Format")
    @ParameterizedTest
    @ValueSource(strings = {"0123456789", "1234567890", "+0123456789"})
    public void shouldTestPhoneNumberFormatUsingValueSource(String phoneNumber) {
        contactManager.addContact("John", "Doe", phoneNumber);
        assertFalse(contactManager.getAllContacts().isEmpty());
        assertEquals(1, contactManager.getAllContacts().size());
    }

    @DisplayName("CSV Source Case - Phone Number should match the required Format")
    @ParameterizedTest
    @CsvSource({"0123456789", "1234567890", "+0123456789"})
    public void shouldTestPhoneNumberFormatUsingCSVSource(String phoneNumber) {
        contactManager.addContact("John", "Doe", phoneNumber);
        assertFalse(contactManager.getAllContacts().isEmpty());
        assertEquals(1, contactManager.getAllContacts().size());
    }

    @DisplayName("CSV File Source Case - Phone Number should match the required Format")
    @ParameterizedTest
    @CsvFileSource(resources = "/data.csv")
    public void shouldTestPhoneNumberFormatUsingCSVFileSource(String phoneNumber) {
        contactManager.addContact("John", "Doe", phoneNumber);
        assertFalse(contactManager.getAllContacts().isEmpty());
        assertEquals(1, contactManager.getAllContacts().size());
    }
}
```

La ejecución de la clase de prueba funciona de igual, solo que la nueva clase con anotación `@Nested` se muestra en un nodo distinto.



Unit testing para Javascript con el framework: Jest

Cuando se trata de frameworks de pruebas unitarias para Javascript, Jest es sin duda un serio competidor para el puesto #1 [13].

Inicialmente, Jest fue creado por Facebook para probar aplicaciones hechas con React, de hecho, es uno de los frameworks más utilizados para probar componentes de React. Desde que Jest apareció ha ganado tanta popularidad que hoy en día es utilizado para probar aplicaciones tanto de front-end como de back-end en Javascript [13].

La característica más relevante de Jest, es su enfoque en la simplicidad, Jest ofrece una experiencia completa y sin complicaciones a la hora de realizar pruebas en JavaScript y es utilizado por grandes empresas como Twitter, Instagram, Pinterest, Airbnb y muchas otras [14].

Características

Tomado de [14].

- **zero config:** Jest tiene como objetivo funcionar de forma inmediata, sin configuración, en la mayoría de los proyectos de JavaScript.
- **Snapshots:** Permite hacer pruebas que realicen un seguimiento de objetos grandes con facilidad. Las snapshots viven junto con sus pruebas o se integran en línea.
- **Isolated:** Las pruebas se paralelizan ejecutándolas en sus propios procesos para maximizar el rendimiento.
- **Great api:** Jest tiene todo el conjunto de herramientas en un solo lugar. Bien documentado, bien mantenido, bien bueno. Esta api se encuentra en la página oficial de Jest <https://jestjs.io/docs/api>.
- **Fast and safe:** Al garantizar que sus pruebas tengan un estado global único, Jest puede ejecutar pruebas en paralelo de manera confiable. Para agilizar las cosas, Jest ejecuta primero las pruebas fallidas y reorganiza las ejecuciones en función de la duración de los archivos de prueba.

Funcionalidades

Tomado de [13] y [14].

- **Code coverage:** Usando la bandera `--coverage` es posible recolectar la información de código de todo el proyecto y ejecutar las pruebas existentes. No se necesita configuración adicional. Jest puede recopilar información de código de proyectos completos, incluidos archivos no testeados.
- **Great Exceptions:** Las pruebas fallan; cuando lo hacen, Jest proporciona un contexto enriquecido del por qué, podemos utilizar el parámetro `--verbose` a la hora de realizar los tests para que Jest nos especifique al máximo por qué algo falla.
- **Describe blocks:** Los bloques de descripción se utilizan para organizar casos de prueba en grupos lógicos de pruebas. Por ejemplo, si queremos agrupar todas las pruebas de una clase específica. Además, se pueden anidar nuevos bloques de descripción en un bloque de descripción existente.

Tipos de pruebas

- **Globals:** En los archivos de prueba, Jest coloca algunos métodos y objetos útiles en el entorno global. No se requiere importar nada para usarlos.
 - Algunos de estos métodos globales disponibles en la api [15] son:
 - ***describe(name, fn)***: crea un bloque que agrupa varias pruebas relacionadas.
 - ***beforeEach(fn, timeout)***: Ejecuta una función antes de que se ejecute cada una de las pruebas de este archivo.
 - ***afterEach(fn, timeout)***: Ejecuta una función después de que se completa cada una de las pruebas en el archivo.
 - ***test(name, fn, timeout)***: Este es el método que ejecuta una prueba.
- **Expect:** Cuando se están escribiendo pruebas, a menudo se necesita verificar que los valores cumplan con ciertas condiciones. *expect* le da acceso a una serie de "**matchers**" que permiten validar diferentes cosas. Los **matchers** se utilizan para crear aserciones en combinación con la palabra clave *expect*. Es decir, cuando se quiere comparar el resultado de una prueba con un valor que se espera que devuelva la función [13].
 - Algunos de los métodos matchers disponibles en la api [15] son:
 - ***.toBe(value)***: se utiliza para comparar valores primitivos o para verificar la identidad referencial de instancias de objetos.

- ***.toEqual(value)***: se utiliza para comparar recursivamente todas las propiedades de instancias de objetos (también conocido como "deep equality" o igualdad profunda).
- ***toBeInstanceOf(Class)***: es utilizado para comprobar que un objeto es una instancia de una clase.
- ***.toBeNull()***: es lo mismo que utilizar *.toBe(null)* pero el mensaje de error es mejor, por lo que es mejor utilizarlo para checkear cuando algo es nulo.
- **Mock functions**: Las funciones mock también se conocen como "espías", porque le permiten espiar el comportamiento de una función a la que otro código llama indirectamente, en lugar de solo probar la salida. Puede crear una función mock con la instrucción *jest.fn()*. Si no se proporciona ninguna implementación, la función mock devolverá undefined cuando se invoque.
 - En la siguiente figura se muestra un ejemplo del uso de un Mock.

```
1  const mockFn = jest.fn();
2  mockFn();
3  expect(mockFn).toHaveBeenCalled();
```

- En la línea 1 se crea el mock asignándolo a la variable mockFn.
- En la línea 2 se hace una llamada al mock (como no se proporcionó una implementación entonces esta función debería devolver undefined).
- En la línea 3 se realiza una prueba que pasará solo si la instrucción fue llamada anteriormente, y fallará si no lo fue, en este caso el resultado de la prueba dará un resultado correcto porque el mock fue llamado en la línea 2.
- **The jest object**: El objeto Jest está dentro del alcance de cada archivo de prueba. Los métodos en el objeto Jest ayudan a crear mocks y permiten controlar el comportamiento general de Jest.
 - Algunos ejemplos de los métodos del objeto jest según el api [15] son:
 - ***jest.spyOn(object,methodName)***: Un spy tiene un comportamiento ligeramente diferente al mock, pero sigue siendo comparable. Los spy crean una función mock similar a *jest.fn()* pero también rastrean las llamadas a los métodos de un objeto. Devuelve una función mock de Jest.
 - ***jest.isMockFunction(fn)***: *Determina si la función dada es una función mock.*

- ***jest.setTimeout(timeout)***: Establece el intervalo de tiempo de espera predeterminado para las pruebas en milisegundos. Esto solo afecta al archivo de prueba desde el que se llama a esta función.
- ***jest.retryTimes()***: Ejecuta pruebas fallidas n veces hasta que pasan o hasta que se agota el número máximo de reintentos. Los reintentos no funcionarán si se llama a `jest.retryTimes()` en `beforeEach` o en un bloque de prueba.

Ejemplo de uso

Para el ejemplo de pruebas unitarias con Jest se creó el archivo `index.js` el cual posee un objeto llamado `function_examples` con dos funciones llamadas `play_fizz_buzz(numbers, callback)` y `play_escudo_o_corona()`. El objetivo de este objeto es definir funciones a las que se le realizaran pruebas unitarias con el framework Jest. En la siguiente figura se observa la instancia de `function_examples`:

```
1  const function_examples = {
2    // Se definen algunas funciones de ejemplo para realizar pruebas unitarias.
3
4  >   play_fizz_buzz(numbers, callback) { ...
25     },
26
27 >   play_escudo_o_corona(){ ...
36     }
37   };
38
39   module.exports = function_examples;
```

La función **`play_fizz_buzz(numbers, callback)`** consiste en simular el juego `fizz_buzz` el cual consiste en contar secuencialmente desde el número 1 en adelante reemplazando algunos números si cumplen con una multiplicidad establecida y los que no la cumplen se cuentan como el número en sí. Los números que se reemplazan según su multiplicidad son los siguientes:

- múltiplos de 3 se reemplazan por la palabra “fizz”
- múltiplos de 5 se reemplazan por la palabra “buzz”
- múltiplos de 15 se reemplazan por la palabra “fizzbuzz”

La implementación de esta función en javascript se muestra en la siguiente figura, donde el parámetro `numbers` es una lista con los números que se jugará y `callback` es una función que se utilizará más adelante para realizar pruebas con funciones Mock.

```
play_fizz_buzz(numbers, callback) {  
  // Fizzbuzz es un juego en el que se hace una cuenta secuencial de numeros  
  // reemplazando unicamente los siguientes multiplos por palabras:  
  // ---Multiplos de 3: fizz  
  // ---Multiplos de 5: buzz  
  // ---Multiplos de 15: fizzbuzz  
  // Por ejemplo los primeros 10 numeros: [0,1,2,fizz,4,buzz,fizz,7,8,fizz]  
  let result = []  
  for (number of numbers) {  
    if (number % 15 === 0) {  
      result.push('fizzbuzz')  
    } else if (number % 3 === 0) {  
      result.push('fizz')  
    } else if (number % 5 === 0) {  
      result.push('buzz')  
    } else {  
      result.push(number)  
    }  
  }  
  callback(result.join(', '))  
  return result.join(', ')  
},
```

La función ***play_escudo_o_corona()*** simplemente simula el lanzamiento aleatorio de una moneda y retorna “escudo” o “corona” dependiendo de un número generado aleatoriamente, la posibilidad de que salga escudo o corona es de 50%. La implementación de esta función se muestra en la siguiente figura:

```

play_escudo_o_corona(){
  // Esta funcion simula el juego de lanzar una moneda, se genera un numero aleatorio
  // que solo puede ser 1 o 2, en el caso de que el resultado sea 1 retorna escudo y
  // de lo contrario retorna corona.
  n = Math.floor(Math.random() * 2) + 1

  if(n==1){
    return 'escudo'
  }
  return 'corona'
}

```

Finalmente se utiliza la instrucción de la siguiente figura para exportar el objeto con las funciones de ejemplo al archivo donde se realizarán las pruebas.

```

41  module.exports = function_examples;

```

Una vez implementadas las funciones a las que se le realizaran las pruebas se creó el archivo *index.test.js* el cual incluye las pruebas del framework Jest. Primero se procede a importar las funciones de ejemplo mediante la instrucción que se muestra en la siguiente figura:

```

1  const function_examples = require('./index');

```

Una vez hecho esto, se realiza el primer bloque de descripción que incluye pruebas de **expect** y **Mock** para la función *play_fizz_buzz(numbers, callback)*. Antes de empezar con el bloque de descripción se instancia una función Mock que se enviará como segundo parámetro (parámetro callback) a la función *play_fizz_buzz(numbers, callback)*, la instancia de la función Mock se muestra en la siguiente figura, esta función Mock retorna el valor que se le pasa por parámetro.

```

3  const mockFn = jest.fn(x => x);

```

Una vez instanciada la función Mock se define el bloque de descripción que incluye 4 pruebas de **expect** con distintos parámetros para la función *play_fizz_buzz(numbers, callback)*, para estas pruebas de **expect** se utilizó el método **matcher** *.toBe()* y se pasó la función *mockFn* al parámetro *callable* de la función *play_fizz_buzz(numbers, callback)*. En la siguiente figura se muestra el primer bloque de pruebas **expect**:


```

9   describe("FizzBuzz", () => {
10     test('[3] debería dar como resultado "fizz"', () => {
11       expect(function_examples.play_fizz_buzz([3],mockFn)).toBe('fizz');
12     });
13     test('[5] debería dar como resultado "buzz"', () => {
14       expect(function_examples.play_fizz_buzz([5],mockFn)).toBe('buzz');
15     });
16     test('[15] debería dar como resultado "fizzbuzz"', () => {
17       expect(function_examples.play_fizz_buzz([15],mockFn)).toBe('fizzbuzz');
18     });
19     test('[1,2,3] debería dar como resultado "1, 2, fizz"', () => {
20       expect(function_examples.play_fizz_buzz([1,2,3],mockFn)).toBe('1, 2, fizz');
21     });
22   });
23 });

```

Ejecutando el comando `npm test` en la CLI se obtienen los siguientes resultados del test:

```

PS C:\Users\carlo\OneDrive\Escritorio\2022_ESPE_UnitTesting\Unit_Test_Javascript\jest> npm test

PASS ./index.test.js
  FizzBuzz
    ✓ [3] debería dar como resultado "fizz" (2 ms)
    ✓ [5] debería dar como resultado "buzz" (1 ms)
    ✓ [15] debería dar como resultado "fizzbuzz" (1 ms)
    ✓ [1,2,3] debería dar como resultado "1, 2, fizz"

Test Suites: 1 passed, 1 total
Tests:       4 passed, 4 total
Snapshots:   0 total
Time:        0.391 s, estimated 1 s
Ran all test suites.

```

En la figura anterior podemos observar cómo los 4 tests pasaron correctamente y también se puede observar el tiempo que tomó llevar a cabo las pruebas.

El siguiente bloque de descripción de pruebas contiene 5 pruebas de tipo Mock con el objetivo de ver el comportamiento de las llamadas que se realizan a una función, en este caso será función `mockFn` definida anteriormente. En la siguiente figura se observa la implementación de este bloque:

```
describe("Mock Functions", () => {
  test('Pruebas mock', () => {
    // La funcion mock ha sido llamada
    expect(mockFn).toHaveBeenCalled();
    // La funcion mock ha sido llamada 4 veces
    expect(mockFn.mock.calls.length).toBe(4);
    // El primer argumento de la primera llamada debe ser fizz
    expect(mockFn.mock.calls[0][0]).toBe('fizz');
    // El primer argumento de la segunda llamada debe ser buzz
    expect(mockFn.mock.calls[1][0]).toBe('buzz');
    // Lo que devuelve la primera llamada debe ser fizz
    expect(mockFn.mock.results[0].value).toBe('fizz');
  });
});
```

Como se observa en los comentarios del código de la figura anterior, las pruebas de tipo Mock permiten acceder a la información de las llamadas a una función tipo Mock. A continuación, la explicación de cada prueba:

Nota: la función *play_fizz_buzz(numbers, callback)* llama a la función *callback* al final de su ejecución y le pasa por parámetro el resultado final.

- ***expect(mockFn).toHaveBeenCalled():*** En este caso la prueba pasará satisfactoriamente porque la función *mockFn* fue llamada en varias ocasiones por la función *play_fizz_buzz(numbers, callback)* en las pruebas realizadas en el primer bloque de descripción.
- ***expect(mockFn.mock.calls.length).toBe(4):*** En este caso la prueba pasará satisfactoriamente porque la función *mockFn* fue llamada exactamente 4 veces por la función *play_fizz_buzz(numbers, callback)* en las pruebas realizadas en el primer bloque de descripción.
- ***expect(mockFn.mock.calls[0][0]).toBe('fizz'):*** En este caso la prueba pasará satisfactoriamente porque el primer argumento que se le pasó a *mockFn* la primera vez que se llamó fue “fizz”.

- ***expect(mockFn.mock.calls[0][0]).toBe('fizz')***: En este caso la prueba pasará satisfactoriamente porque el primer argumento que se le pasó a mockFn la primera vez que se llamó fue “fizz”.
- ***expect(mockFn.mock.calls[1][0]).toBe('buzz')***: En este caso la prueba pasará satisfactoriamente porque el primer argumento que se le pasó a mockFn la segunda vez que se llamó fue “buzz”.
- ***expect(mockFn.mock.results[0].value).toBe('fizz')***: En este caso la prueba pasará satisfactoriamente porque el valor que retorna la función mockFn la primera vez que se llama es “fizz”, recordando que mockFn devuelve lo que le pasan por parametro.

Los resultados de la ejecución de pruebas con Jest para el segundo bloque de descripción se muestran en la siguiente figura:

```
PS C:\Users\carlo\OneDrive\Escritorio\2022_ESPE_UnitTesting\Unit_Test_Javascript\jest> npm test

> jest@1.0.0 test
  ✓ [15] debería dar como resultado "fizzbuzz" (1 ms)
  ✓ [1,2,3] debería dar como resultado "1, 2, fizz"
  Mock Functions
    ✓ Pruebas mock (1 ms)

Test Suites: 1 passed, 1 total
Tests:       5 passed, 5 total
Snapshots:   0 total
Time:        0.546 s, estimated 1 s
Ran all test suites.
```

Otra prueba realizada con el framework Jest fue una prueba tipo Jest Object llamada SpyOn la cual hace lo mismo que una función Mock pero rastrea un objeto en específico y las llamadas sus métodos.

Para esta prueba se creó un bloque de descripción bastante sencillo donde simplemente se trackea la función *play_fizz_buzz* para posteriormente analizar la información de sus llamadas, esta vez es información de la función *play_fizz_buzz* y no la función mockFn. En la siguiente figura se muestra la implementación de este bloque de descripción.

```

45 describe("Spy Function", () => {
46
47     test('Pruebas spy', () => {
48
49         const spy = jest.spyOn(function_examples, 'play_fizz_buzz');
50
51         const test_1 = function_examples.play_fizz_buzz([1], mockFn);
52
53         expect(spy).toHaveBeenCalled(); // La funcion fizz_buzz fue llamada.
54         expect(test_1).toBe("1"); // El resultado para la entrada 1 de fizz_buzz debe ser 1.
55
56         spy.mockRestore(); //Restaura el estado del mock creado por el spy.
57     });
58
59 });
60

```

Primero se define spy para que espíe la función *play_fizz_buzz* que se encuentra en el objeto *function_examples* luego se guarda en una constante el resultado de llamar a *play_fizz_buzz* con los parámetros (*numbers=1*, *callback=mockFn*). Posteriormente se realiza las pruebas **expect** mencionadas anteriormente para probar que la función *play_fizz_buzz* fue llamada a través de *spy* y para verificar el resultado de *test_1*.

Al realizar esta prueba se obtuvieron los resultados que se muestran en la siguiente figura:

```

PS C:\Users\carlo\OneDrive\Escritorio\2022_ESPE_UnitTesting\Unit_Test_Javascript\jest> npm test
FizzBuzz
  ✓ [3] debería dar como resultado "fizz" (2 ms)
  ✓ [5] debería dar como resultado "buzz" (1 ms)
  ✓ [15] debería dar como resultado "fizzbuzz" (1 ms)
  ✓ [1,2,3] debería dar como resultado "1, 2, fizz"
Mock Functions
  ✓ Pruebas mock (1 ms)
Spy Function
  ✓ Pruebas spy (2 ms)

Test Suites: 1 passed, 1 total
Tests:       6 passed, 6 total
Snapshots:   0 total
Time:        0.55 s, estimated 1 s
Ran all test suites.

```

Como se puede observar en la figura anterior, las pruebas de *spy* pasaron satisfactoriamente.

Finalmente, la última prueba que se realizó fue la prueba por intentos *retryTimes(n)* la cual realiza *n* cantidad de veces una prueba en busca de un solo resultado satisfactorio, y solo en caso de todos fallen, la prueba fallará. Para esta prueba se utilizó la función *play_escudo_o_corona()* y la implementación de este bloque de descripción se muestra en la siguiente figura:

```

65 describe("RetryTimes escudo o corona", () => {
66
67     jest.retryTimes(3); // Prueba 3 veces la funcion en busqueda de que pase minimo una vez.
68     test('Intento de obtener escudo en 3 turnos de lanzar una moneda', () => {
69         expect(function_examples.play_escudo_o_corona()).toBe('escudo');
70     });
71
72 });

```

En la figura anterior se puede observar que el bloque comienza con la instrucción *jest.retryTimes(3)* la cual indica que la prueba se intentará 3 veces en busca del primer resultado satisfactorio, luego se declara el test como en los casos anteriores con un **expect** que busca que el resultado sea “escudo”. Este test pasará en la mayoría de las ocasiones debido a que la probabilidad de obtener escudo con 3 intentos es muy alta, en la siguiente figura se muestra el resultado del test.

```

> jest --verbose

PASS ./index.test.js
  FizzBuzz
    ✓ [3] deberia dar como resultado "fizz" (3 ms)
    ✓ [5] deberia dar como resultado "buzz" (1 ms)
    ✓ [15] deberia dar como resultado "fizzbuzz"
    ✓ [1,2,3] deberia dar como resultado "1, 2, fizz"
  Mock Functions
    ✓ Pruebas mock (2 ms)
  Spy Function
    ✓ Pruebas spy (1 ms)
  RetryTimes escudo o corona
    ✓ Intento de obtener escudo en 3 turnos de lanzar una moneda (1 ms)

Test Suites: 1 passed, 1 total
Tests:       7 passed, 7 total
Snapshots:   0 total
Time:        0.502 s, estimated 1 s
Ran all test suites.

```

Conclusiones

Las pruebas unitarias son sumamente útiles debido a que permiten tener un control de lo que está bien en un proyecto en tiempo real y esto nos permite una mejor escalabilidad en el proyecto ya que si algo se daña al agregar un nuevo módulo o dependencia las pruebas nos notificaran inmediatamente donde se encuentra el fallo.

Las pruebas unitarias también son favorables para dar mayor legibilidad al código que escribimos ya que si otra persona no comprende muy bien el funcionamiento de una función puede irse a las pruebas unitarias y observar lo que se espera que esta función haga.

JUnit es un framework que admite ejecutar pruebas unitarias de forma automatizada, aunque los resultados deben ser específicos de cada test. Por el conjunto de reglas de diseño permite un desarrollo de código limpio, de mayor calidad, y que su a vez es independiente pues es un código aislado que se ha creado con la misión de comprobar otro código concreto. Además, brinda retroalimentación puntual para cada resultado que alcanza, es rentable y le facilita al usuario la mantenibilidad de su código.

Por otra parte, el set de herramientas que presenta JUnit es compacto, permite al usuario tener control sobre el orden de las pruebas, es rápido de crear y proporciona un trabajo ágil pues posibilita la detección de los errores a tiempo.

El framework Jest es muy útil para realizar pruebas unitarias ya que su principal característica son las pruebas sencillas y rápidas sin necesidad de una configuración inicial compleja y a su capacidad de correr los bloques de descripción de pruebas en paralelo.

Finalmente, los Mocks en Jest son especialmente útiles en las pruebas unitarias porque permiten probar la lógica de las funciones sin preocuparse por sus dependencias.

Bibliografía

[1]"A Survey on Unit Testing Practices and Problems", *ieeexplore.ieee.org*, 2022. [Online]. Available: <https://ieeexplore.ieee.org/document/6982627>.

[2]*Www2.ccs.neu.edu*, 2022. [Online]. Available: <https://www2.ccs.neu.edu/research/demeter/related-work/extreme-programming/MockObjectsFinal.PDF>.

[3]"👉 ¿Qué son las pruebas unitarias y cómo llevar una a cabo?", *Yeeply*, 2022. [Online]. Available: <https://www.yeeply.com/blog/que-son-pruebas-unitarias/#as>.

[4]"Z. Peng, X. Lin, M. Simon and N. Niu, "Unit and regression tests of scientific software: A study on SWMM", *Journal of Computational Science*, vol. 53, p. 101347, 2021. Available: 10.1016/j.jocs.2021.101347.

[5]"¿Desventajas del desarrollo basado en pruebas?", *Iteramos*, 2022. [Online]. Available: <https://www.iteramos.com/pregunta/4523/desventajas-del-desarrollo-basado-en-pruebas>.

[6]"A Survey on Unit Testing Practices and Problems", *ieeexplore.ieee.org*, 2022. [Online]. Available: <https://ieeexplore.ieee.org/document/6982627>.

[7]C. Stefan Bechtold, "JUnit 5 User Guide", *JUnit.org*, 2022. [Online]. Available: <https://junit.org/junit5/docs/current/user-guide/#overview>.

[8]"JUnit · Libro Desarrollo de Software", *Rcasalla.gitbooks.io*, 2022. [Online]. Available: https://rcasalla.gitbooks.io/libro-desarrollo-de-software/content/libro/temas/t_pruebas/prue_junit.html.

[9]"¿Qué es JUnit?", *La paradita*, 2022. [Online]. Available: <https://elrincondeaj.wordpress.com/2012/07/09/junit/>.

[10]"JUnit | Marco de Desarrollo de la Junta de Andalucía", *Juntadeandalucia.es*, 2022. [Online]. Available: <http://www.juntadeandalucia.es/servicios/madeja/contenido/recurso/248>.

[11]"Mastering Software Testing with JUnit 5", García, Boni., 2017. [Online]. Available: <https://search-ebscohost-com.ezproxy.itcr.ac.cr/login.aspx?direct=true&db=e000xww&AN=1626950&lang=es&sit e=ehost-live>.

[13]"Jest Testing: A Helpful, Introductory Tutorial - Testim Blog", AI-driven E2E automation with code-like flexibility for your most resilient tests, 2022. [Online]. Available: <https://www.testim.io/blog/jest-testing-a-helpful-introductory-tutorial/>.

[14]"Jest", *Jestjs.io*, 2022. [Online]. Available: <https://jestjs.io>.

[15]"API Jest", *Jestjs.io*, 2022. [Online]. Available: <https://jestjs.io/docs/api>.