Desarrollo en Java basado en pruebas (Parte I)

Pruebas Unitarias

Competencias

- Entender importancia de tener pruebas unitarias
- Entender como cambia el flujo de desarrollo implementando pruebas
- Entender que las pruebas unitarias deben ser implementadas con la misma calidad que el código de producción

Motivación

Es común encontrarse en la situación en donde se deben realizar varios cambios en el sistema, pero ¿Cómo validar estos cambios y comprobar que están buenos? No se debe esperar a que ocurra una fallo en producción para realizar una verificación de las nuevas funcionalidades. Las pruebas unitarias verifican que las unidades individuales de código funcionen como se espera. Además, ayudan a comprender el diseño del código en el que se está trabajando.

Cuando se trabaja en un equipo distribuido, con varios integrantes realizando cambios, las pruebas unitarias toman un rol importante, ayudando a mantener la calidad del código y la integración entre las funcionalidades de los miembros del equipo. De esta forma los integrantes verifican su código y apoyan a la agilidad del equipo aumentando la velocidad de su desarrollo.

Ciclo

Cuando se adoptan las pruebas unitarias, se modifican un poco los procesos por los cuales pasan las características añadidas al código, el siguiente flujo (Imagen 1) se aplica para cada cambio.

1. En primer lugar, el proceso comienza descargando el código desde su respectivo repositorio (lugar donde se almacena el código fuente).
2. Se realizan los cambios, se agregan nuevas funcionalidades.
3. Se deben escribir las pruebas unitarias y ser ejecutadas.
4. Se corrigen las pruebas fallidas y se ejecutan nuevamente, al ser exitosas se sigue con el flujo.
5. De forma optativa, se puede hacer una revisión de código por parte de otro miembro del equipo, para validar los cambios.
6. Se confirman y se suben los cambios al repositorio.

> A continuación el diagrama del ciclo de las pruebas unitarias

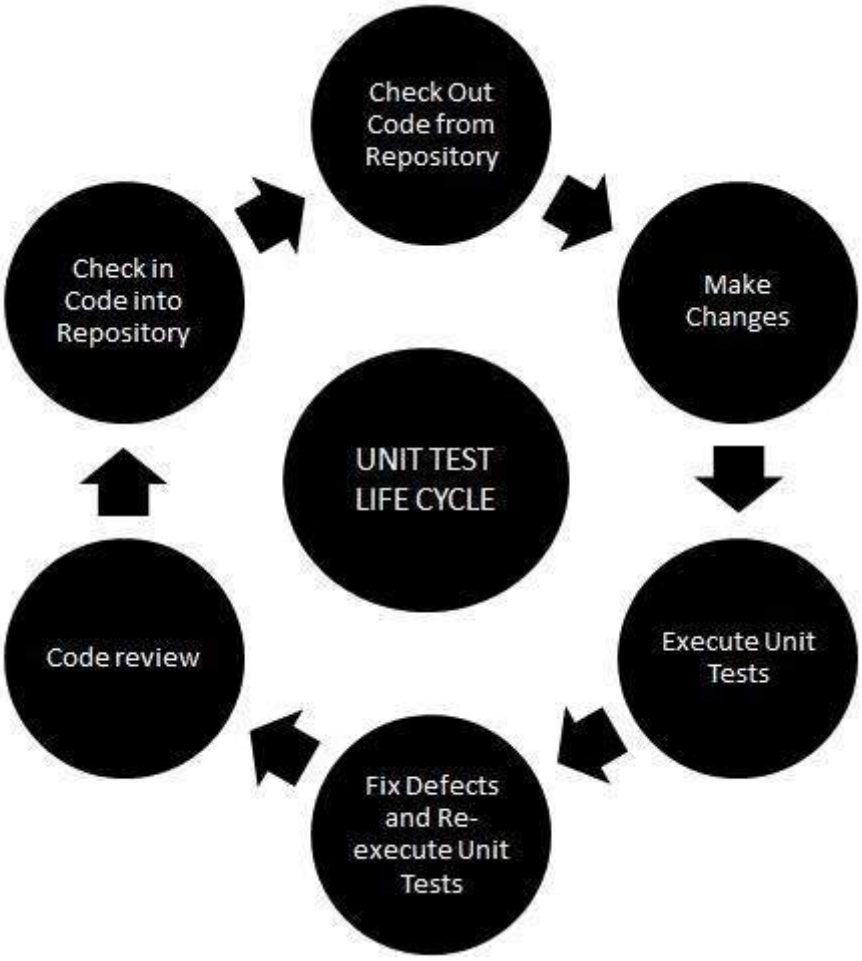

Imagen 1: Ciclo Unit Test.

Ventajas de realizar pruebas unitarias

- Se escriben pequeñas pruebas a la vez, obliga a que el código sea más modular (de lo contrario, sería difícil probarlo).
- Las pruebas sirven de documentación.
- El código es más fácil de mantener y refactorizar.
- Las pruebas unitarias son valiosas como red de seguridad cuando se necesita cambiar el código para agregar nuevas funciones o para corregir un error existente.
- Los errores son atrapados casi inmediatamente.
- Hace más eficiente la colaboración, los miembros del equipo pueden editar el código de cada uno con confianza porque las pruebas verifican si los cambios realizados hacen que el código se comporte de manera inesperada.

Desventajas de realizar pruebas unitarias

- Las pruebas deben mantenerse.
- Inicialmente, ralentiza el desarrollo.
- Las pruebas unitarias deben adoptarse por todo el equipo.
- Un reto puede ser intimidante y no es fácil aprender al principio.
- Difícil de aplicar al código heredado existente.

Las dificultades que vienen con las pruebas unitarias están en como se vende el concepto a una organización. Se rechazan las pruebas unitarias porque el desarrollo con pruebas unitarias lleva más tiempo y, por lo tanto, retrasa la entrega. En algunos casos, eso puede ser cierto, pero no hay forma de saber si esos retrasos son una consecuencia real de las pruebas de unitarias o por un diseño deficiente o requisitos incompletos. Pero al lograr un buen diseño y con requisitos bien definidos.

Las pruebas unitarias entregan muchas ventajas, y a continuación se detallan algunas.

Facilita realizar los cambios

Como punto de partida las pruebas unitarias hacen que el desarrollo sea mayormente ágil. Cuando se agregan nuevas características a un sistema, a veces necesita refactorizar el diseño y el código antiguo. Sin embargo, cambiar el código ya probado generalmente es arriesgado. Pero si ya se tienen las pruebas unitarias implementadas, entonces se puede proceder a refactorizar con confianza. Las pruebas unitarias colaboran con la agilidad porque se basa en pruebas que le permiten realizar cambios con mayor facilidad, haciendo fácil la refactorización.

Calidad del código

Además de mejorar la calidad del código las pruebas unitarias identifican los defectos que pueden surgir antes de que el código pase a las pruebas de integración. Escribir pruebas antes de escribir la lógica de negocios, permite simplificar el problema ayudando a exponer los casos de borde para escribir un mejor código.

Encontrar errores en diseño

Los problemas se encuentran en una etapa temprana, dado que los desarrolladores que prueban un código individual antes de la integración realizan las pruebas unitarias, los problemas se pueden encontrar muy pronto y se pueden resolver en cualquier momento sin afectar las otras partes del código. Esto incluye errores en la implementación del programador y fallas o partes faltantes de la especificación de la unidad.

Proporciona documentación

Las pruebas unitarias pueden incluso ser usadas como documentación del sistema. Para los desarrolladores que buscan entender qué característica proporciona cada unidad y cómo usarla pueden ver las pruebas, para entender la funcionalidad que estas poseen. Además si las pruebas son claras y expresan sus intenciones de forma concisa, el código será fácil de entender, por lo tanto sirven de documentación.

Gestores de ciclo de vida de la aplicación

Son sistemas de automatización de construcción, que tienen el objetivo de simplificar los procesos de build (limpiar, compilar y generar ejecutables del proyecto a partir del código fuente). Se abordará Apache Maven, Maven es una herramienta de gestión y comprensión de proyectos de software. Basándose en el concepto de un modelo de objeto de proyecto (POM), Maven puede gestionar la compilación, los informes y la documentación de un proyecto a partir de una información central.

El archivo pom.xml es el núcleo de la configuración de un proyecto en Maven. Es un archivo de configuración único que contiene la mayoría de la información necesaria para construir un proyecto de la forma que desee. El POM es enorme y puede ser desalentador en su complejidad, pero no es necesario entender todas las complejidades para usarlo de manera efectiva.

JUnit

Competencias

- Introducción a Maven.
- Desarrollar pruebas usando características de JUnit.
- Conocer las anotaciones de JUnit, para saber cuándo aplicarlas.
- Desarrollar pruebas usando fixtures, para organizar y reutilizar código.

Motivación

En su versión más reciente, JUnit provee características para escribir pruebas expresivas, que representen sus intenciones de forma clara. Además posee muy buena integración con Java y otras librerias. Escribir pruebas con JUnit convierte la tarea de escribir pruebas unitarias en una experiencia agradable. Existen alternativas para hacer pruebas sobre Java, pero la comunidad de JUnit es la más extensa. Se puede encontrar la documentación y ejemplos en la pagina oficial (https://junit.org/junit5).

Creación nuevo proyecto

Eclipse

Se puede crear un nuevo proyecto Java Maven con Eclipse, como en la Imagen 2.

Primer es ir a **File** > **New** > **Project...**

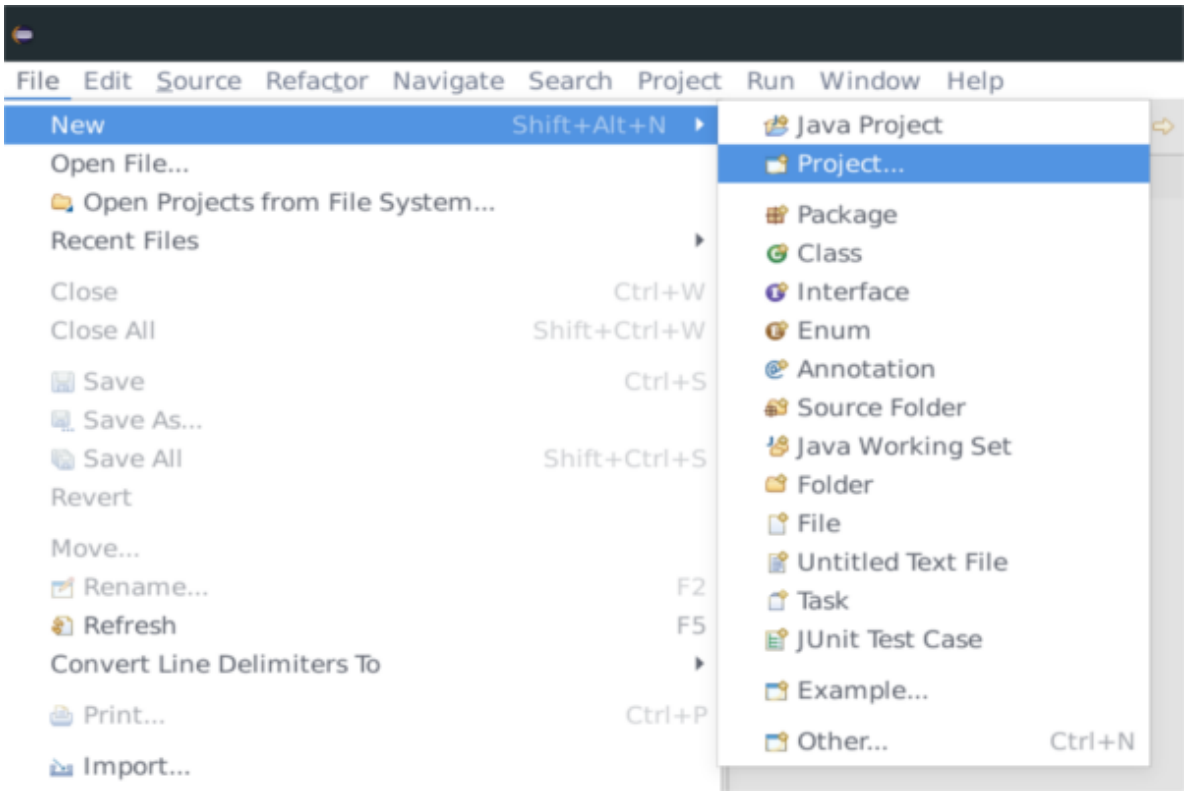

Imagen 2: Creando un nuevo proyecto.

Con esto se abrirá una ventana para asistir la creación del nuevo proyecto, se debe buscar la carpeta Maven y seleccionar Maven Project como se muestra en la Imagen 3.

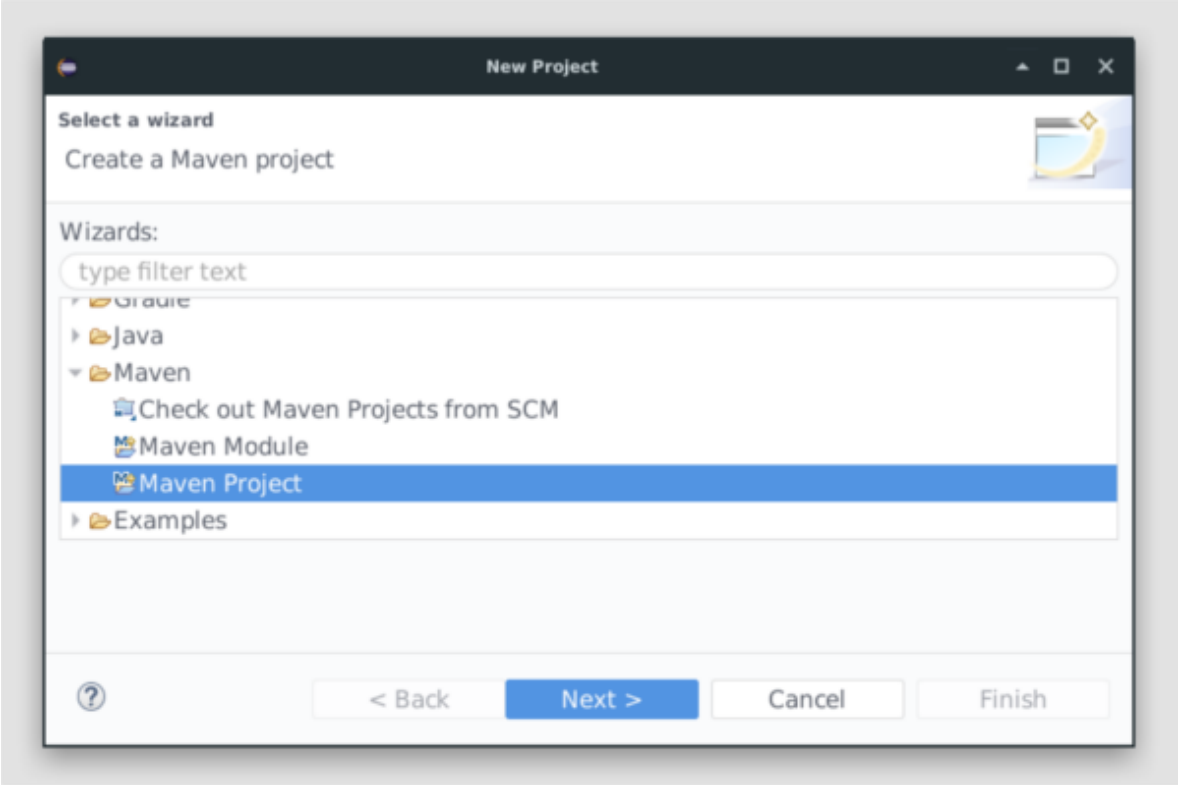

Imagen 3: Create a Maven Project.

Lo siguiente, se muestra en la Imagen 4, que corresponde al paso de seleccionar la ubicación del espacio de trabajo.

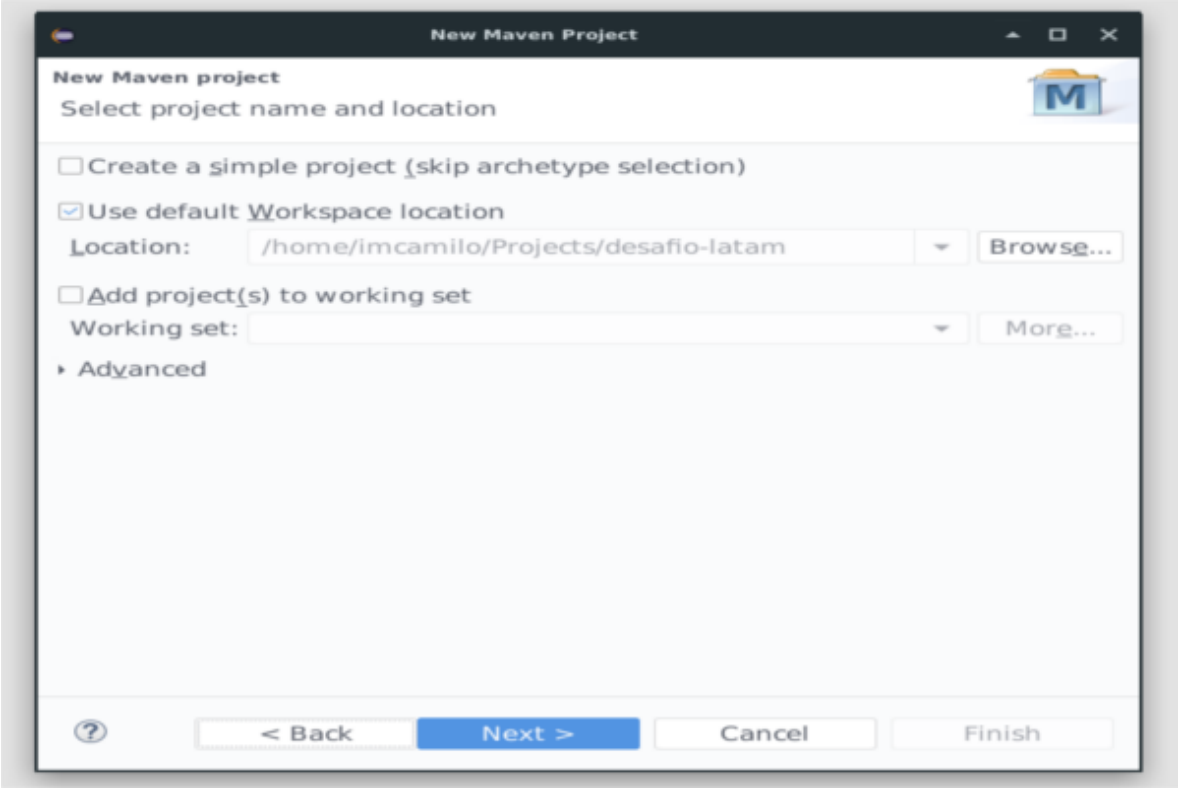

Imagen 4: Select Project Name and Location.

Una vez seleccionado el espacio de trabajo, se puede seleccionar un template (arquetipo) para le nuevo proyecto, el template **maven-archetype-quickstart** contiene lo necesario para iniciar un nuevo proyecto Maven, como se muestra en la Imagen 5.

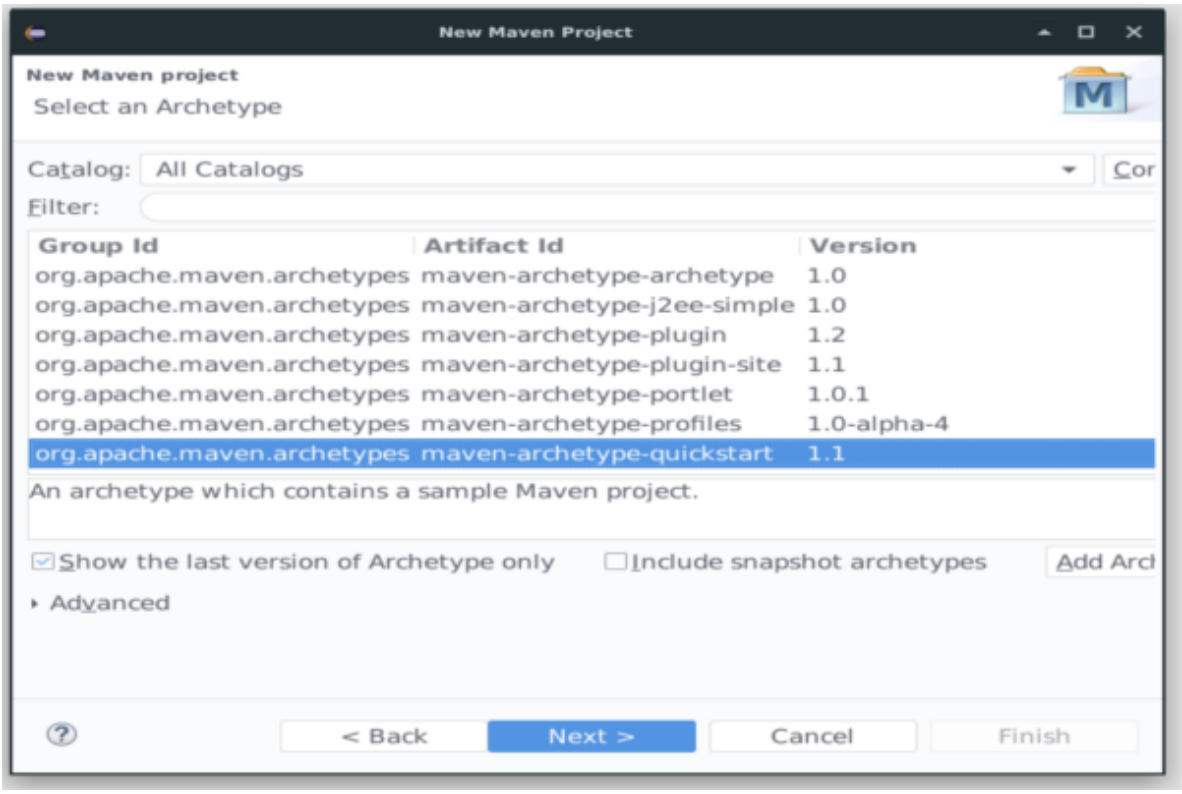

Imagen 5: Select and Archetype.

Luego se definen los parámetros del template (Imagen 6):

- **Group Id** el que contendrá el dominio de la organización, comunmente se usa como `nombredeusuario.github.com`.
- **Artifact Id** será el nombre del proyecto.

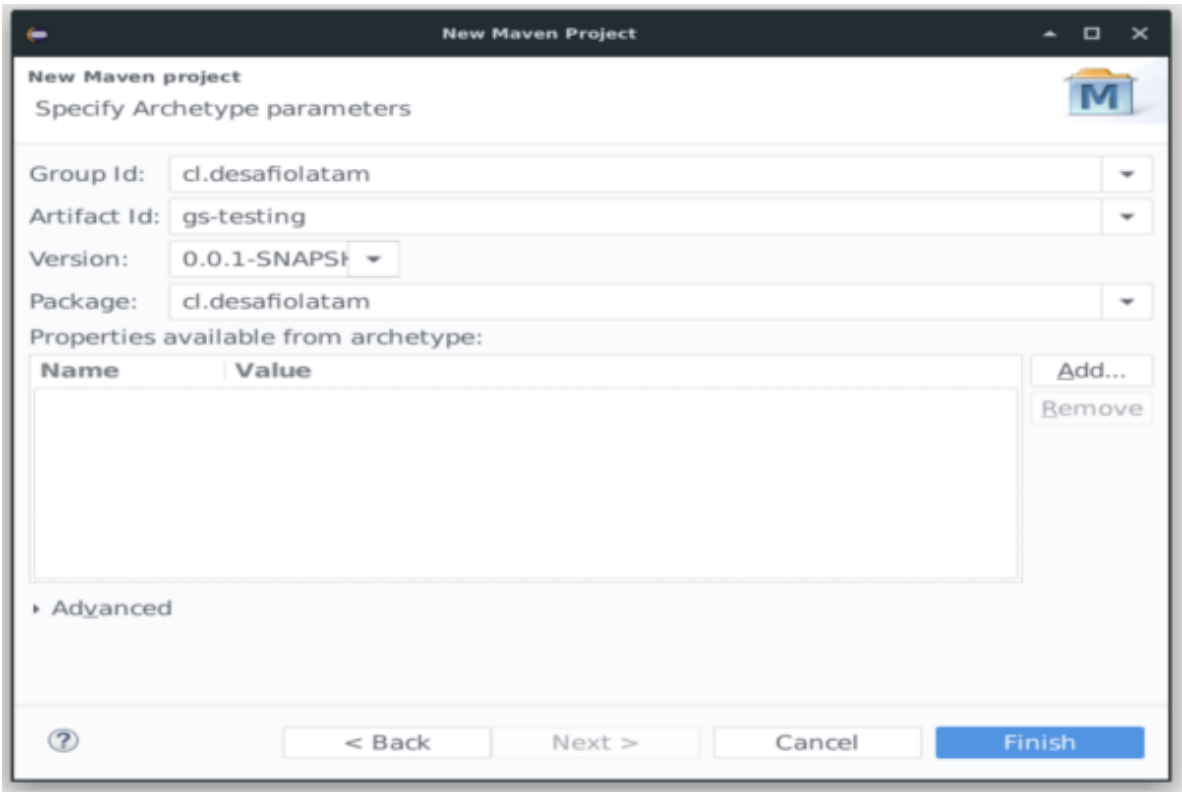

Imagen 6: Specify Archetype Parameters.

Finalmente el proyecto ya está generado y se puede navegar a través de lar carpetas.

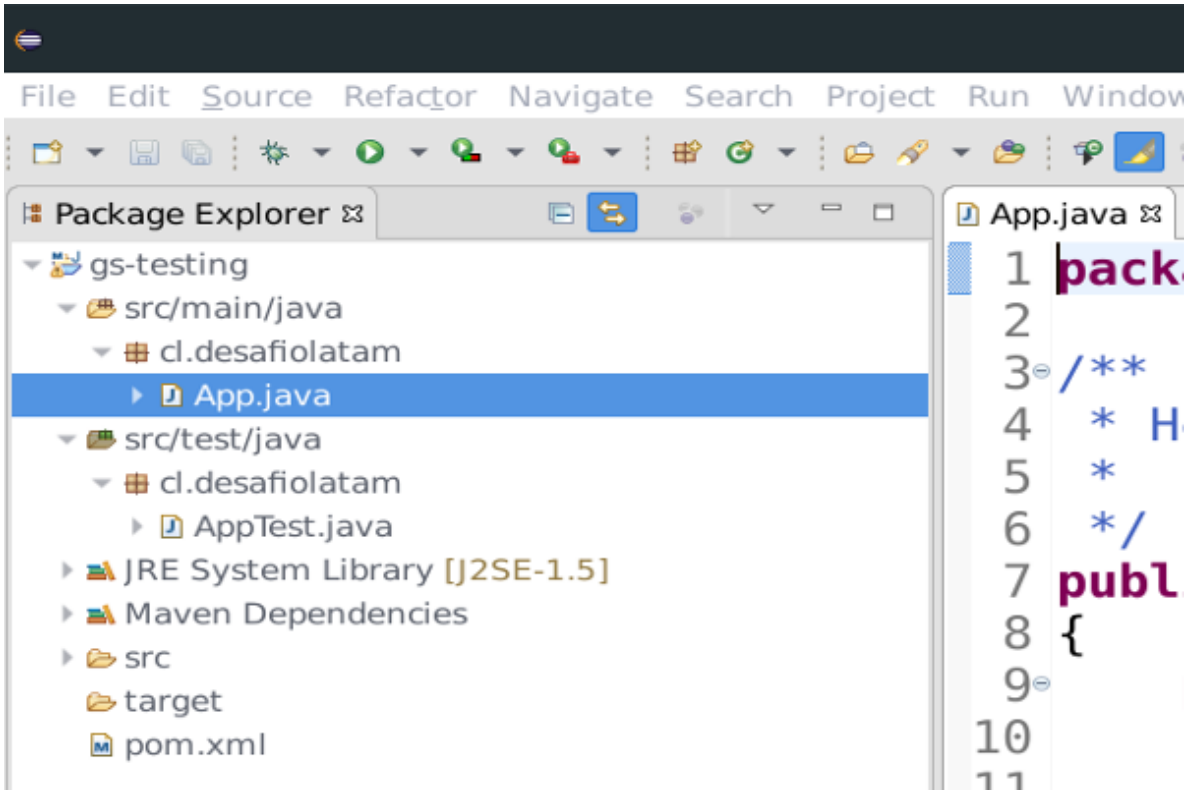

Imagen 7: Getting Started.

Como último paso, se debe modificar el archivo pom.xml para configurar `maven.compiler.source` y `maven.compiler.source` , dentro del tag `properties` , a continuación el detalle:

```
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
  </properties>
```

Usar comandos mvn

También se se puede crear un nuevo proyecto Java con Maven, para ello se debe ejecutar el siguiente comando por consola, esto genera la estructura de carpetas en base a un artifact que no es mas que un template.

```
mvn archetype:generate -DgroupId=cl.desafiolatam -DartifactId=gs-testing -
DarchetypeArtifactId=maven-archetype-quickstart -DarchetypeVersion=1.4 -
DinteractiveMode=false
```

El proyecto generado contiene esta estructura de carpetas.

```
gs-testing
├── pom.xml
└── src
    ├── main
    │   └── java
    │       └── cl
    │           └── desafiolatam
    │               └── App.java
    └── test
        └── java
            └── cl
                └── desafiolatam
                    └── AppTest.java
```

Se pueden borrar las clases Java generadas en main y en test (App.java, AppTest.java) para que crear las propias.

Implementación JUnit

Para empezar a utilizarla debemos agregarlo al proyecto como dependencia adicional, en sistemas de compilación como Gradle o Maven, la dependencia viene por defecto al crear un proyecto Java.

Añadir dependencia

Se usa la versión mas reciente de JUnit, que solo consiste en un librería adicional en donde el mismo Maven se encarga de gestionar y agregar al proyecto, y que corresponde a esta dependencia:

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter</artifactId>
    <version>5.4.2</version>
    <scope>test</scope>
</dependency>
```

Se debe ir al archivo `pom.xml` en la raíz del proyecto, se borra la dependencia de JUnit que viene por defecto, y se agrega la nueva dentro del tag **dependencies**. El archivo `pom.xml` sería el siguiente.

```
<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>cl.desafiolatam</groupId>
  <artifactId>gs-testing</artifactId>
  <version>1.0-SNAPSHOT</version>

  <name>gs-testing</name>
  <!-- FIXME change it to the project's website -->
  <url>http://www.example.com</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter</artifactId>
      <version>5.4.2</version>
      <scope>test</scope>
    </dependency>
  </dependencies>

  <!--resto del archivo-->

</project>
```

Escribir código a probar

Para empezar ya tenemos creado el proyecto llamado gs-testing.

Lo siguiente es que se debe crear la clase Persona dentro de la carpeta src/main del proyecto, en la siguiente ruta.

`src.main.java.cl.desafiolatam.modelos.Persona.java`

Luego, se debe agregar en la carpeta modelos, para que el código se divida por responsabilidades.

```
gs-testing
├── pom.xml
└── src
    ├── main
    │   └── java
    │       └── cl
    │           └── desafiolatam
    │               └── modelos
    │                   └── Persona.java
    └── test
        └── java
            └── cl
                └── desafiolatam
```

Este será el objeto que contendrá nuestros datos, Persona tendrá las propiedades rut y nombre, además de su constructor, getters y setters.

```java
package cl.desafiolatam.modelos;

public class Persona {

    private String rut;
    private String nombre;

    //constructor, getters y setters

}
```

Ahora se crea la clase llamada ServicioPersona en la carpeta servicios, la ruta es la siguiente:

`src.main.java.cl.desafiolatam.servicios.ServicioPersona.java`

Las clases del proyecto quedarían así:

```
gs-testing
├── pom.xml
└── src
    ├── main
    │   └── java
    │       └── cl
    │           └── desafiolatam
    │               ├── modelos
    │               │   └── Persona.java
    │               └── servicios
    │                   └── ServicioPersona.java
    └── test
        └── java
            └── cl
                └── desafiolatam
```

Se crea el método crearPersona dentro de la clase ServicioPersona, este recibe un objeto de tipo Persona el cual se encargará de guardarlos en el mapa llamado personasDB, este mapa consiste en un tipo de datos clave-valor que se usará para simular una fuente de datos. El método guardará las personas con el rut como su clave y el nombre como valor, verificando que persona sea distinto de nulo.

Si la persona es insertada correctamente este devuelve un mensaje de “Creada”.

```
package cl.desafiolatam.servicios;

import cl.desafiolatam.modelos.Persona;

import java.util.HashMap;
import java.util.Map;

public class ServicioPersona {

    private Map<String,String> personasDB = new HashMap<>();

    public String crearPersona(Persona persona) {
        if (persona != null) {
            personasDB.put(persona.getRut(), persona.getNombre());
            return "Creada";
        } else {
            return "No creada";
        }
    }

}
```

Se crea el método actualizarPersona realiza una tarea similar a crearPersona, validando que el dato de entrada sea distinto de nulo y actualizando el valor de personasDB. Si la persona es actualizada correctamente, este devuelve un mensaje de “Actualizada”.

```
package cl.desafiolatam.servicios;

import cl.desafiolatam.modelos.Persona;

import java.util.HashMap;
import java.util.Map;

public class ServicioPersona {

    //resto de la clase

    public String actualizarPersona(Persona persona) {
        if (persona != null) {
            personasDB.put(persona.getRut(), persona.getNombre());
            return "Actualizada";
        } else {
            return "No actualizada";
        }
    }

}
```

Se agrega el método listarPersonas que retorna personasDB, el cual es el mapa que se utiliza como almacén de datos.

```
package cl.desafiolatam.servicios;

import cl.desafiolatam.modelos.Persona;

import java.util.HashMap;
import java.util.Map;

public class ServicioPersona {

    //resto de la clase

    public Map<String, String> listarPersonas() {
        return personasDB;
    }

}
```

Como último paso se agrega el método eliminarPersona el que recibe un parámetro de tipo Persona, valida si es nulo, y procede a eliminar la persona que contenga la clave dentro del mapa personasDB, retornando finalmente “Eliminada”.

```
package cl.desafiolatam.servicios;

import cl.desafiolatam.modelos.Persona;

import java.util.HashMap;
import java.util.Map;

public class ServicioPersona {

    //resto de la clase

    public String eliminarPersona(Persona persona) {
        if (persona != null) {
            personasDB.remove(persona.getRut());
            return "Eliminada";
        } else {
            return "No eliminada";
        }
    }
}
```

Clase ServicioPersona completada.

```
package cl.desafiolatam.servicios;

import cl.desafiolatam.modelos.Persona;

import java.util.HashMap;
import java.util.Map;

public class ServicioPersona {

    private Map<String,String> personasDB = new HashMap<>();

    public String crearPersona(Persona persona) {
        if (persona != null) {
            personasDB.put(persona.getRut(), persona.getNombre());
            return "Creada";
        } else {
            return "No creada";
        }
    }

    public String actualizarPersona(Persona persona) {
        if (persona != null) {
            personasDB.put(persona.getRut(), persona.getNombre());
            return "Actualizada";
        } else {
            return "No actualizada";
        }
    }

    public Map<String, String> listarPersonas() {
        return personasDB;
    }

    public String eliminarPersona(Persona persona) {
        if (persona != null) {
            personasDB.remove(persona.getRut());
            return "Eliminada";
        } else {
            return "No eliminada";
        }
    }

}
```

Antes de empezar a escribir las pruebas

Anotaciones

Aun no se han cubierto las anotaciones, pero todo texto que está previo a una clase o un método y que comienza con @ es una anotación.

En JUnit se utilizan anotaciones, las que sirven para añadir metadatos al código y que están disponibles para la aplicación en tiempo de ejecución o de compilación.

¿Por qué usarlas?

Porque se pueden usar como una alternativa a escribir las configuraciones en XML, y porque es muy sencillo aprender a usarlas. A continuación se describen las que se utilizarán.

Estas anotaciones se importan desde `org.junit.jupiter.api`:

- `@Test` Se usa antes de un método e indica que los métodos anotados son métodos de prueba.
- `@DisplayName` Usada para poner un nombre de visualización personalizado para la clase o método de prueba.
- `@BeforeAll` Se utiliza para indicar que el método anotado debe ejecutarse antes de todas las pruebas en la clase de prueba actual.
- `@BeforeEach` Se utiliza para indicar que el método anotado debe ejecutarse antes de cada método de prueba.
- `@AfterEach` Se usa para indicar que el método anotado debe ejecutarse después de cada método de prueba.
- `@AfterAll` Se usa para indicar que el método anotado debe ejecutarse después de todos los método de prueba.

Desde `org.mockito.Mockito` se importa

- Se importa el método estático `mock`, el cual crea un objeto simulado dada una clase o una interfaz.

Loggers

Es un objeto que se usa para registrar mensajes para un sistema específico o componente de aplicación.

Cuando se desarrolla un programa, ya sea para ambiente de pruebas o producción, tener un log donde se estén reportando los eventos o errores, es la clave para detectar posibles fallos en poco tiempo, existen multiples librerías que realizan este trabajo, pero con su propio log Java proporciona la capacidad de capturar los archivos del registro.

¿Por qué usar un log?

Hay varias razones por las que se puede necesitar capturar la actividad de la aplicación.

- Registro de circunstancias inusuales o errores que puedan estar ocurriendo en el programa.
- Obtener la información sobre qué está pasando en la aplicación.

Los detalles que se pueden obtener de los registros pueden variar. A veces, es posible que se requieran muchos detalles con respecto al problema o, a veces, solo información sencilla.

Para ello los logs tienen niveles, como SEVERE que indica que algo grave falló, WARNING indica un potencial problema, INFO indica información general, etc.

Afirmaciones

Las afirmaciones son métodos de utilidad para respaldar las condiciones en las pruebas; estos métodos son accesibles a través de la clase Assertions. Estos métodos se pueden usar directamente: `Assertions.assertEquals("","")`, sin embargo, se leen mejor si se hace referencia a ellos mediante la importación estática, por ejemplo:

```
import static org.junit.jupiter.api.Assertions.assertEquals;

assertEquals("OK", respuestaEsperadaQueDebeSerOK);
```

Dos afirmaciones comunes:

- `assertEquals` recibe dos parámetros, lo esperado y actual para afirmar si son iguales.
- `assertNotNull` recibe un parámetro y se encarga de validar que este no sea nulo.

Escribir pruebas

Como saber si los métodos de la clase ServicioPersona funcionan como deberían, podemos ejecutarlos por nosotros mismos o estar seguros de que el código no tiene errores, pero escribiendo sus pruebas unitarias la verfificacion es segura y evitamos tener efectos secundarios.

Crear la clase ServicioPersonaTest dentro de la carpeta src/test del proyecto, en la ruta

`src.test.java.cl.desafiolatam.servicios.ServicioPersonaTest.java`

Las clases del proyecto deben quedar así:

```
gs-testing
├── pom.xml
└── src
    ├── main
    │   └── java
    │       └── cl
    │           └── desafiolatam
    │               ├── modelos
    │               │   └── Persona.java
    │               └── servicios
    │                   └── ServicioPersona.java
    └── test
        └── java
            └── cl
                └── desafiolatam
                    └── servicios
                        └── ServicioPersonaTest.java
```

Esta sera la clase que contendrá las pruebas de ServicioPersona, inicialmente se instancia la clase ServicioPersona. Además se crea un logger para ir registrando los eventos de forma descriptiva. Para hacer que la prueba sea más legible y expresiva se agrega `@DisplayName`.

```java
package cl.desafiolatam.servicios;

import org.junit.jupiter.api.*;

import java.util.logging.Logger;

@DisplayName("Tests Clase ServicioPersona")
public class ServicioPersonaTest {

    private static Logger logger =
Logger.getLogger("cl.desafiolatam.servicios.ServicioPersonaTest");
    private final ServicioPersona servicioPersona = new ServicioPersona();

}
```

Lo siguiente es escribir las pruebas para los métodos del servicio de personas, tendrán la anotación `@Test` que indica que estos métodos son métodos de prueba. En los métodos se utiliza el método estático `assertEquals`. Los métodos contienen un log con el nombre de la prueba y tienen la anotación `@DisplayName` que indica el nombre.

Método de prueba testCrearPersona para método crearPersona, además de sus respectivas anotaciones, lo siguiente es crear un objeto de tipo Persona, pasándole datos a través del constructor, esta será la persona a guardar (juanito). Se crea una variable respuestaServicio la cual almacenará el valor de la respuesta del servicio crearPersona, crearPersona recibe como parámetro a juanito y retorna un String. Finalmente se usa `assertEquals` para comprobar que lo esperado “Creada”, y la respuesta del servicio son iguales, pasando la prueba.

```java
package cl.desafiolatam.servicios;

import cl.desafiolatam.modelos.Persona;
import org.junit.jupiter.api.*;

import java.util.logging.Logger;

import static org.junit.jupiter.api.Assertions.assertEquals;

@DisplayName("Tests Clase ServicioPersona")
public class ServicioPersonaTest {

    private static Logger logger =
Logger.getLogger("cl.desafiolatam.servicios.ServicioPersonaTest");
    private final ServicioPersona servicioPersona = new ServicioPersona();

    @Test
    @DisplayName("Test testCrearPersona")
    void testCrearPersona() {
        logger.info("info test crear persona");
        Persona juanito = new Persona("1234-1", "Juanito");
        String respuestaServicio = servicioPersona.crearPersona(juanito);
        assertEquals("Creada", respuestaServicio);
    }
}
```

Para ejecutar la preuba testCrearPersona, se debe ejecutar desde una terminal, el comando `mvn test`, el cuál ejecutará las pruebas, la salida de ese comando es:

```
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running cl.desafiolatam.servicios.ServicioPersonaTest
Jul 07, 2019 1:16:19 PM cl.desafiolatam.servicios.ServicioPersonaTest
testCrearPersona
INFO: info test crear persona
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.03 s
- in cl.desafiolatam.servicios.ServicioPersonaTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
-
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
-
[INFO] Total time:  2.151 s
[INFO] Finished at: 2019-07-07T13:16:19-04:00
[INFO] ------------------------------------------------------------------------
-
```

Método de prueba testActualizarPersona para método actualizarPersona, además de sus respectivas anotaciones, lo siguiente es crear un objeto de tipo Persona, pasándole datos a través del constructor, esta será la persona a actualizar (pepe). Se crea una variable respuestaServicio la cual almacenará el valor de la respuesta de actualizarPersona, actualizarPersona recibe como parámetro a pepe y retorna un String. Finalmente se usa `assertEquals` para comprobar que lo esperado “Se actualizo!”, y la respuesta del servicio son iguales.

```
package cl.desafiolatam.servicios;

import cl.desafiolatam.modelos.Persona;
import org.junit.jupiter.api.*;

import java.util.logging.Logger;

import static org.junit.jupiter.api.Assertions.assertEquals;

@DisplayName("Tests Clase ServicioPersona")
public class ServicioPersonaTest {

    private static Logger logger =
Logger.getLogger("cl.desafiolatam.servicios.ServicioPersonaTest");
    private final ServicioPersona servicioPersona = new ServicioPersona();

    //resto de la clase

    @Test
    @DisplayName("Test actualizarPersona")
    void testActualizarPersona() {
        logger.info("info actualizar persona");
        Persona pepe = new Persona("1234-1", "Pepe");
        String respuestaServicio = servicioPersona.actualizarPersona(pepe);
        assertEquals("Se actualizo!", respuestaServicio);
    }

}
```

Al ejecutar `mvn test` se observa `AssertionFailedError`, y se detalla que testActualizarPersona falla en la línea 34, donde se espera "Se actualizo!", pero se obtuvo Actualizada.

```
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running cl.desafiolatam.servicios.ServicioPersonaTest
Jul 07, 2019 1:49:08 PM cl.desafiolatam.servicios.ServicioPersonaTest
testCrearPersona
INFO: info test crear persona
Jul 07, 2019 1:49:08 PM cl.desafiolatam.servicios.ServicioPersonaTest
testActualizarPersona
INFO: info actualizar persona
[ERROR] Tests run: 2, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0.052
s <<< FAILURE! - in cl.desafiolatam.servicios.ServicioPersonaTest
[ERROR] testActualizarPersona  Time elapsed: 0.012 s  <<< FAILURE!
org.opentest4j.AssertionFailedError: expected: <Se actualizo!> but was:
<Actualizada>
```

```
    at
cl.desafiolatam.servicios.ServicioPersonaTest.testActualizarPersona(ServicioPe
rsonaTest.java:34)

[INFO]
[INFO] Results:
[INFO]
[ERROR] Failures:
[ERROR]   ServicioPersonaTest.testActualizarPersona:34 expected: <Se
actualizo!> but was: <Actualizada>
[INFO]
[ERROR] Tests run: 2, Failures: 1, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
-
[INFO] BUILD FAILURE
[INFO] ------------------------------------------------------------------------
-
[INFO] Total time:  2.794 s
[INFO] Finished at: 2019-07-07T13:49:08-04:00
```

Esto ocurre porque el método actualizarPersona retorna el String "Actualizada" y se está comparando con otra respuesta, se observa que la prueba detona las características del método, pero falla en la asersión. Se debe corregir la prueba para comprobar si la salida es correcta.

```java
package cl.desafiolatam.servicios;

import cl.desafiolatam.modelos.Persona;
import org.junit.jupiter.api.*;

import java.util.logging.Logger;

import static org.junit.jupiter.api.Assertions.assertEquals;

@DisplayName("Tests Clase ServicioPersona")
public class ServicioPersonaTest {

    private static Logger logger =
Logger.getLogger("cl.desafiolatam.servicios.ServicioPersonaTest");
    private final ServicioPersona servicioPersona = new ServicioPersona();

    //resto de la clase

    @Test
    @DisplayName("Test actualizarPersona")
    void testActualizarPersona() {
        logger.info("info actualizar persona");
        Persona pepe = new Persona("1234-1", "Pepe");
        String respuestaServicio = servicioPersona.actualizarPersona(pepe);
        assertEquals("Actualizada", respuestaServicio);
    }

}
```

La salida de `mvn test` con testActualizarPersona modificado, resultando exitoso.

```
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running cl.desafiolatam.servicios.ServicioPersonaTest
Jul 07, 2019 1:56:01 PM cl.desafiolatam.servicios.ServicioPersonaTest
testCrearPersona
INFO: info test crear persona
Jul 07, 2019 1:56:01 PM cl.desafiolatam.servicios.ServicioPersonaTest
testActualizarPersona
INFO: info actualizar persona
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.038 s
- in cl.desafiolatam.servicios.ServicioPersonaTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
-
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
-
[INFO] Total time:  2.748 s
[INFO] Finished at: 2019-07-07T13:56:01-04:00
[INFO] ------------------------------------------------------------------------
-
```

Método de prueba testEliminarPersona para método eliminarPersona, además de sus respectivas anotaciones, lo siguiente es crear un objeto de tipo Persona, pasándole datos a través del constructor, esta será la persona a eliminar (pepe). Se crea una variable respuestaServicio la cual almacenará el valor de la respuesta de eliminarPersona, crearPersona recibe como parámetro a pepe y retorna un `String`. Finalmente se usa `assertEquals` para comprobar que lo esperado "Eliminada", y la respuesta del servicio son iguales, pasando la prueba.

```java
package cl.desafiolatam.servicios;

import cl.desafiolatam.modelos.Persona;
import org.junit.jupiter.api.*;

import java.util.logging.Logger;

import static org.junit.jupiter.api.Assertions.assertEquals;

@DisplayName("Tests Clase ServicioPersona")
public class ServicioPersonaTest {

    private static Logger logger =
Logger.getLogger("cl.desafiolatam.servicios.ServicioPersonaTest");
    private final ServicioPersona servicioPersona = new ServicioPersona();

    //resto de la clase

    @Test
    @DisplayName("Test eliminarPersona")
    void testEliminarPersona() {
        logger.info("info eliminar persona");
        Persona pepe = new Persona("1234-1", "Pepe");
        String respuestaServicio = servicioPersona.eliminarPersona(pepe);
        assertEquals(respuestaServicio, "Eliminada");
    }

}
```

La salida de `mvn test` una vez escrito la prueba testEliminarPersona, denota 3 pruebas ejecutadas y ninguna fallida.

```
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running cl.desafiolatam.servicios.ServicioPersonaTest
Jul 07, 2019 2:05:39 PM cl.desafiolatam.servicios.ServicioPersonaTest
testEliminarPersona
INFO: info eliminar persona
Jul 07, 2019 2:05:39 PM cl.desafiolatam.servicios.ServicioPersonaTest
testCrearPersona
INFO: info test crear persona
Jul 07, 2019 2:05:39 PM cl.desafiolatam.servicios.ServicioPersonaTest
testActualizarPersona
INFO: info actualizar persona
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.04 s
- in cl.desafiolatam.servicios.ServicioPersonaTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
-
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
-
[INFO] Total time:  2.688 s
[INFO] Finished at: 2019-07-07T14:05:40-04:00
[INFO] ------------------------------------------------------------------------
-
```

Método de prueba testListarPersona para método listarPersona, además de sus respectivas anotaciones, se crea una variable listaPersonas la cual almacenará el valor de la respuesta de listarPersona, listarPersona retorna un `Map<String, String>`, es decir un mapa de claves-valores, se usa `assertNotNull` para comprobar listaPersonas que listaPersonas no sea nula, la cual podría ser una lista vacía, pero nunca nula, pasando la prueba.

```java
package cl.desafiolatam.servicios;

import cl.desafiolatam.modelos.Persona;
import org.junit.jupiter.api.*;

import java.util.Map;
import java.util.logging.Logger;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertNotNull;

@DisplayName("Tests Clase ServicioPersona")
public class ServicioPersonaTest {

    private static Logger logger =
Logger.getLogger("cl.desafiolatam.servicios.ServicioPersonaTest");
    private final ServicioPersona servicioPersona = new ServicioPersona();

    //resto de la clase
```

```
    @Test
    @DisplayName("Test ListarPersona")
    void testListarPersona() {
        logger.info("info listar persona");
        Map<String, String> listaPersonas = servicioPersona.listarPersonas();
        assertNotNull(listaPersonas);
    }

}
```

La salida de `mvn test` una vez escrito la prueba testListarPersona, denota 4 pruebas ejecutadas, y ninguna fallida.

```
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running cl.desafiolatam.servicios.ServicioPersonaTest
Jul 07, 2019 2:09:06 PM cl.desafiolatam.servicios.ServicioPersonaTest
testEliminarPersona
INFO: info eliminar persona
Jul 07, 2019 2:09:06 PM cl.desafiolatam.servicios.ServicioPersonaTest
testCrearPersona
INFO: info test crear persona
Jul 07, 2019 2:09:06 PM cl.desafiolatam.servicios.ServicioPersonaTest
testActualizarPersona
INFO: info actualizar persona
Jul 07, 2019 2:09:06 PM cl.desafiolatam.servicios.ServicioPersonaTest
testListarPersona
INFO: info listar persona
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.04 s
- in cl.desafiolatam.servicios.ServicioPersonaTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
-
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
-
[INFO] Total time:  2.813 s
[INFO] Finished at: 2019-07-07T14:09:06-04:00
[INFO] ------------------------------------------------------------------------
-
```

El código completo de la clase de prueba.

```
package cl.desafiolatam.servicios;

import cl.desafiolatam.modelos.Persona;
import org.junit.jupiter.api.*;

import java.util.Map;
import java.util.logging.Logger;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertNotNull;

@DisplayName("Tests Clase ServicioPersona")
public class ServicioPersonaTest {

    private static Logger logger =
Logger.getLogger("cl.desafiolatam.servicios.ServicioPersonaTest");
    private final ServicioPersona servicioPersona = new ServicioPersona();

    @Test
    @DisplayName("Test testCrearPersona")
    void testCrearPersona() {
        logger.info("info test crear persona");
        Persona juanito = new Persona("1234-1", "Juanito");
        String respuestaServicio = servicioPersona.crearPersona(juanito);
        assertEquals("Creada", respuestaServicio);
    }

    @Test
    @DisplayName("Test actualizarPersona")
    void testActualizarPersona() {
        logger.info("info actualizar persona");
        Persona pepe = new Persona("1234-1", "Pepe");
        String respuestaServicio = servicioPersona.actualizarPersona(pepe);
        assertEquals("Actualizada", respuestaServicio);
    }

    @Test
    @DisplayName("Test eliminarPersona")
    void testEliminarPersona() {
        logger.info("info eliminar persona");
        Persona pepe = new Persona("1234-1", "Pepe");
        String respuestaServicio = servicioPersona.eliminarPersona(pepe);
        assertEquals(respuestaServicio, "Eliminada");
    }

    @Test
    @DisplayName("Test ListarPersona")
    void testListarPersona() {
        logger.info("info listar persona");
        Map<String, String> listaPersonas = servicioPersona.listarPersonas();
        assertNotNull(listaPersonas);
    }

}
```

TestFixtures

Si existen pruebas que tienen necesidades parecidas o sus características son iguales, estas características se pueden agrupar en una TestFixture, o en términos más simples escribiendo las tareas en la misma clase con el objetivo de reutilizar código y eliminar código duplicado. De esta forma al estar en la misma clase, se pueden empezar a crear métodos que todas las pruebas puedan consumir. JUnit brinda anotaciones útiles que se pueden usar para reutilizar código, facilitar su desarrollo y claridad para inicializar objetos, a continuación el detalle de algunas que se pueden integrar en su clase de prueba.

`@BeforeAll` Se utiliza para indicar que el método anotado debe ejecutarse antes de todas las pruebas, el cual puede ser utilizado para inicializar objetos, preparación de datos de entrada o simular objetos para la prueba. Además los métodos deben tener un tipo de retorno nulo, no deben ser privados y deben ser estáticos por defecto.

```
package cl.desafiolatam.servicios;

import org.junit.jupiter.api.*;

//imports

@DisplayName("Tests Clase ServicioPersona")
public class ServicioPersonaTest {

    @BeforeAll
    static void setup() {
        logger.info("Inicio clase de prueba");
    }

    //resto de la clase

}
```

`@BeforeEach` se utiliza para indicar que el método anotado debe ejecutarse antes de cada método que esté anotado con `@Test` en la clase de prueba actual, puede utilizarse para inicializar o simular objetos específicos para cada prueba. Los métodos `@BeforeEach` deben tener un tipo de retorno nulo, no deben ser privados y no deben ser estáticos.

```
package cl.desafiolatam.servicios;

import org.junit.jupiter.api.*;

//imports

@DisplayName("Tests Clase ServicioPersona")
public class ServicioPersonaTest {

    @BeforeEach
    void init() {
        logger.info("Inicio metodo de prueba");
    }

    //resto de la clase

}
```

`@AfterEach` se usa para indicar que el método anotado debe ejecutarse después de cada método anotado con `@Test` en la clase de prueba actual. Los métodos `@AfterEach` deben tener un tipo de retorno nulo, no deben ser privados y no deben ser estáticos.

```
package cl.desafiolatam.servicios;

import org.junit.jupiter.api.*;

//imports

@DisplayName("Tests Clase ServicioPersona")
public class ServicioPersonaTest {

    @AfterEach
    void tearDown() {
        logger.info("Metodo de prueba finalizado");
    }

    //resto de la clase

}
```

`@AfterAll` se utiliza para indicar que el método anotado debe ejecutarse después de todas las pruebas en la clase de prueba actual, en donde es idóneo liberar los objetos creados. Los métodos `@AfterAll` deben tener un tipo de retorno nulo, no deben ser privados y deben ser estáticos por defecto

```
package cl.desafiolatam.servicios;

import org.junit.jupiter.api.*;

//imports

@DisplayName("Tests Clase ServicioPersona")
public class ServicioPersonaTest {

    @AfterAll
    static void done() {
        logger.info("Fin clase de prueba");
    }

    //resto de la clase

}
```