JUnit

¡Comencemos!

¿Qué aprenderás?

- Conocer las anotaciones de JUnit para saber cuándo aplicarlas.
- Desarrollar pruebas unitarias en un proyecto Java usando características de JUnit.

Introducción

En su versión más reciente, JUnit ha proveído de características que ayudan a integrar pruebas mediante Java y otros tipos de bibliotecas. Escribir pruebas unitarias con JUnit convierte la tarea en una experiencia agradable. Existen varias alternativas para hacer pruebas sobre Java, pero la comunidad de JUnit es la más extensa. En la página oficial encontrás documentación y ejemplos: https://junit.org/junit4.

Es importante destacar que JUnit es parte fundamental en el día a día de varios/as programadores/as, ya que nos sirve para comprobar códigos sin alterar significativamente el proceso final de aplicación. Es por esto, que a continuación veremos los detalles más importantes del cómo aplicar e implementar JUnit en Eclipse.

Antes de empezar a escribir las pruebas

Anotaciones

Aún 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 que sirven para añadir metadatos al código y están disponibles para la aplicación en tiempo de ejecución o de compilación.

¿Por qué usarlas?

Porque son una alternativa para escribir las configuraciones en XML, además, es muy sencillo aprender a usarlas.

A continuación, describiremos las anotaciones que importamos 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 métodos de prueba.

Desde `org.mockito.Mockito` se importa el método estático `mock`, el cual crea un objeto simulado dada una clase o una interface.

Loggers

Es un objeto que se usa para registrar mensajes de 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 múltiples 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, sin embargo, destacaremos 2 principalmente:

- 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 obtienen de los registros varían. A veces es posible que se requieran muchos detalles respecto al problema o solo información sencilla. Para ello los logs tienen niveles como `SEVERE`, que indica que algo grave falló, `WARNING`, que indica un potencial problema, e `INFO`, que indica información general, entre otros.

Afirmaciones

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

```
import static org.junit..Assert.assertEquals;
assertEquals("OK", respuestaEsperadaQueDebeSerOK);
```

Dos de las afirmaciones más 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.

Implementación JUnit

Paso 1: Crear un nuevo proyecto mediante File -> New -> Project.

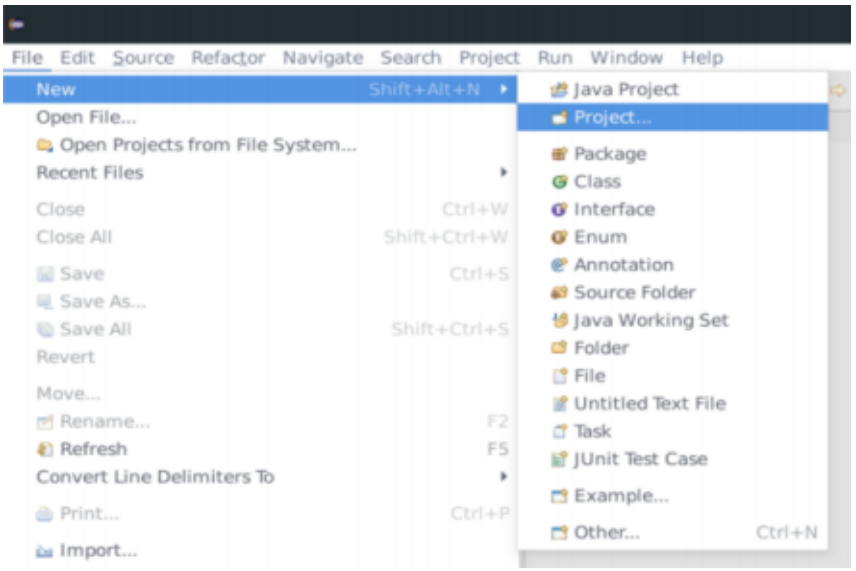

Imagen 1. Creando un nuevo proyecto.
Fuente: Desafio Latam.

Paso 2: Con esto se abrirá una ventana para asistir la creación del nuevo proyecto. Buscar la carpeta llamada Maven y hacer clic sobre aquella que diga `Maven Project.`

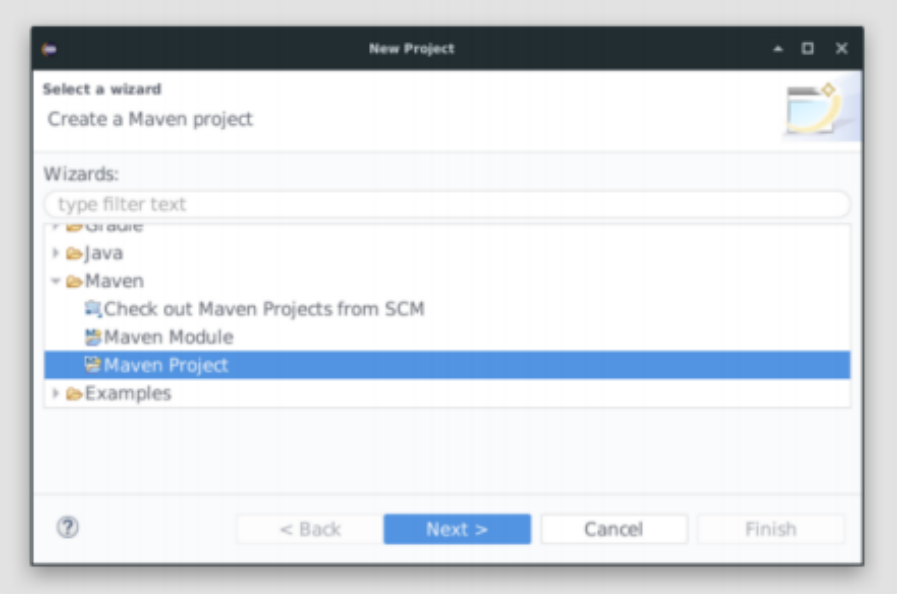

Imagen 2. Creando un proyecto Maven.
Fuente: Desafio Latam.

Paso 3: Seleccionar la ubicación del espacio de trabajo.

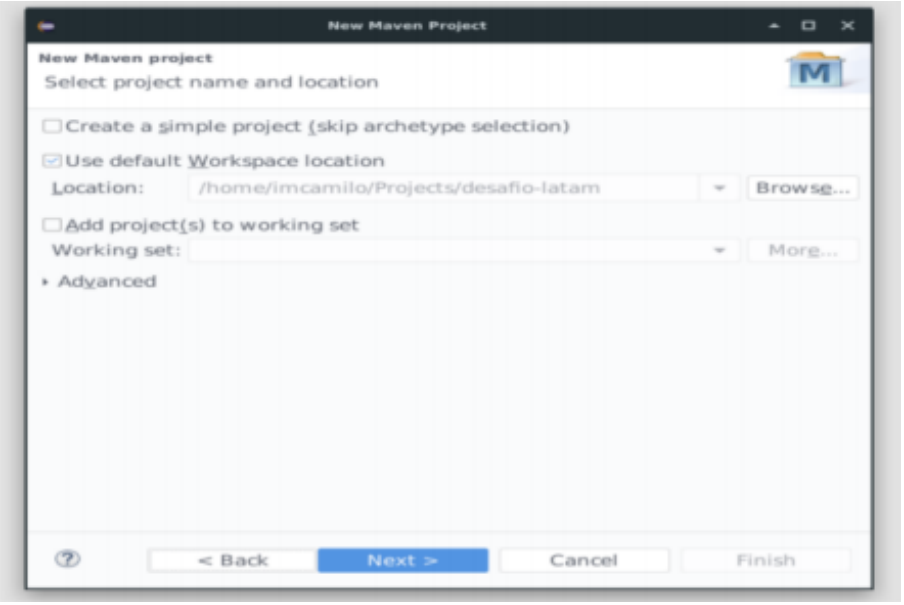

Imagen 3. Select Project Name and Location.
Fuente: Desafio Latam.

Paso 4: Una vez seleccionado el espacio de trabajo, seleccionar un template (arquetipo) para el nuevo proyecto. El template **maven-archetype-quickstart** contiene lo necesario para iniciar un nuevo proyecto Maven, como se muestra a continuación:

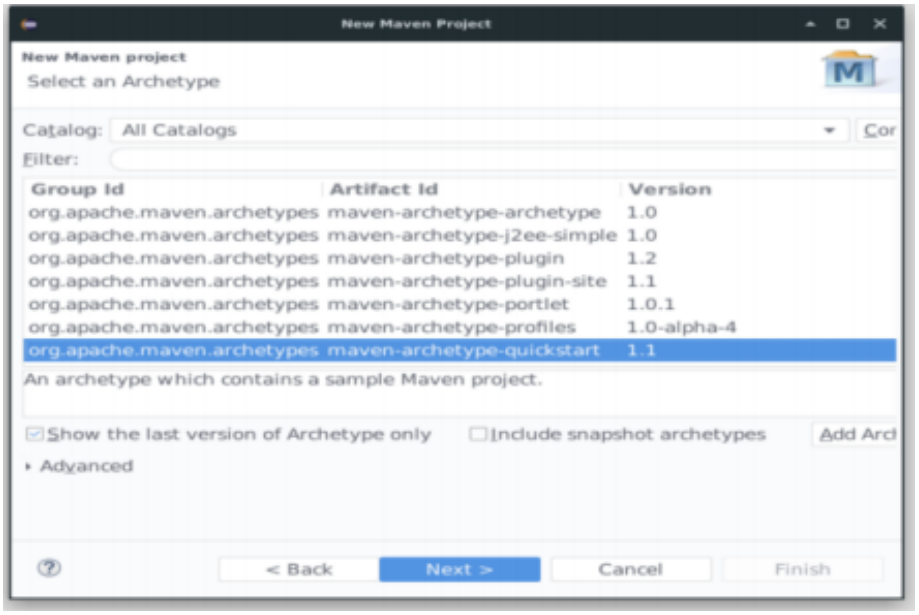

Imagen 4. Select Archetype.
Fuente: Desafio Latam.

Paso 5: Luego, definimos los parámetros del template:

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

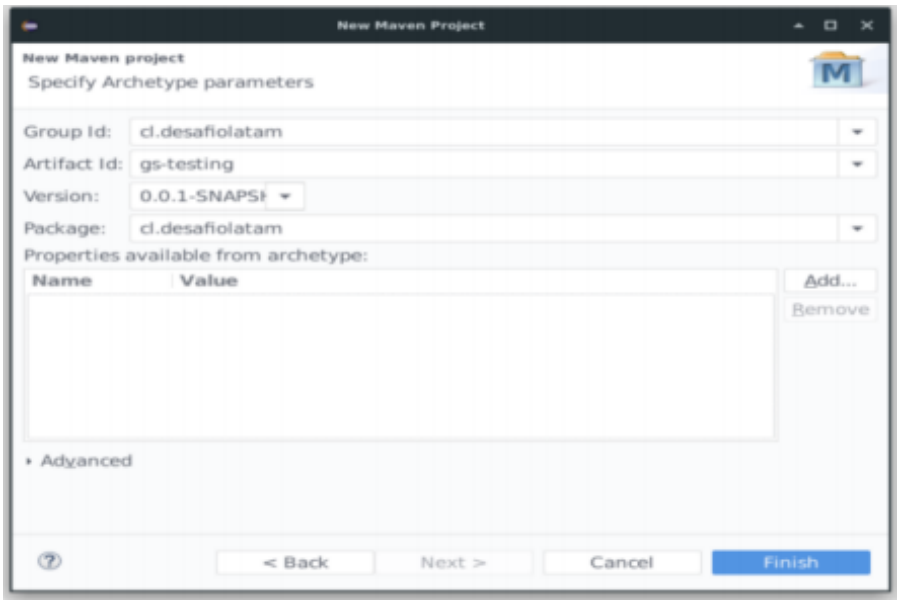

Imagen 5. Specify Archetype Parameters.
Fuente: Desafio Latam.

Paso 6: Modificar el archivo pom.xml para configurar `maven.compiler.source` y `maven.compiler.target`, 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>
```

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

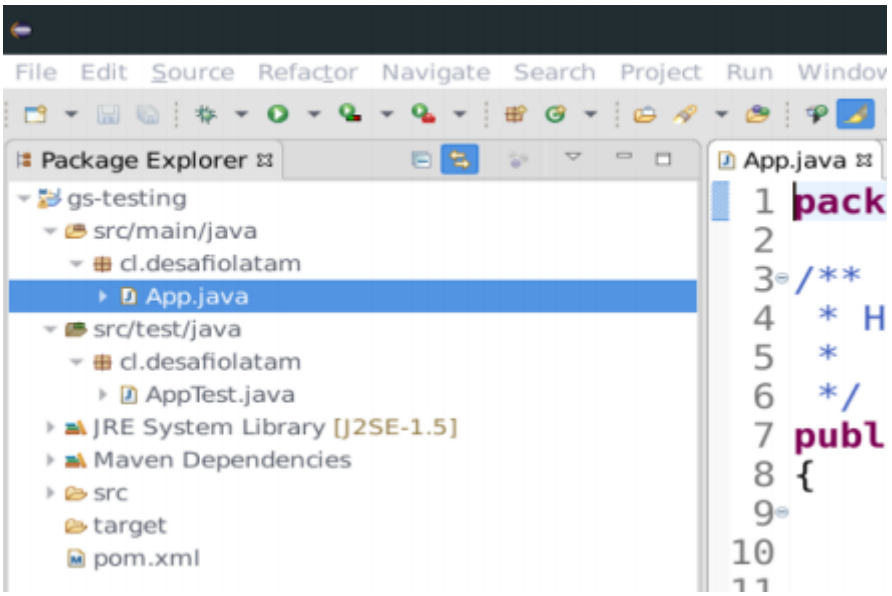

Imagen 6. Getting Started.
Fuente: Desafio Latam.

Iniciando JUnit

Para empezar a utilizar JUnit debemos agregarlo al proyecto como dependencia adicional mediante sistemas de compilación como Gradle o Maven. La dependencia que viene por defecto al crear un proyecto Java es:

```
<dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
</dependency>
```

Sin embargo, 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>
```

En base al proyecto creado previamente, vamos a realizar nuestras primeras pruebas unitarias en Java mediante JUnit.

Ejercicio guiado: Pruebas

Para comenzar a trabajar e implementar todo lo que hemos visto previamente, haremos uso de nuestro conocimiento y crearemos un nuevo proyecto para poner en práctica estos conceptos. Para esto, deberás seguir los siguientes pasos:

Paso 1: Agregar un nuevo package llamado `modelos` dentro del proyecto `gs-testing`.

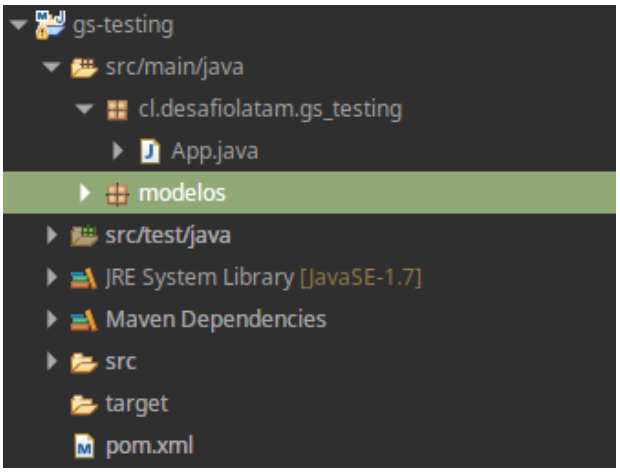

Imagen 7. Creación package modelo.
Fuente: Desafio Latam.

Paso 2: Crear la clase `Persona` dentro del proyecto `gs-testing` y ubicarla en la carpeta `src/main`. Este será el objeto que contendrá nuestros datos como el Rut y el nombre. Además, debemos generar el constructor y los getters y setters correspondientes.

```
package modelos;

public class Persona {
 private String rut;
 private String nombre;
public Persona(String rut, String nombre) {
    super();
    this.rut = rut;
    this.nombre = nombre;
}
public String getRut() {
    return rut;
}
public void setRut(String rut) {
    this.rut = rut;
}
public String getNombre() {
    return nombre;
}
public void setNombre(String nombre) {
    this.nombre = nombre;
}
}
```

Paso 3: Ahora, crear la clase llamada `ServicioPersona` en un nuevo package llamado `servicios`.

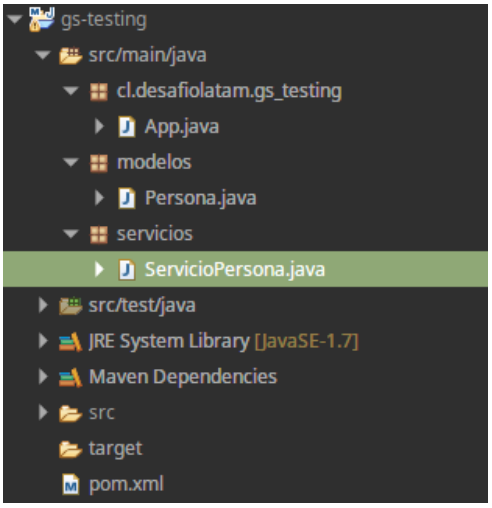

Imagen 8. Creación package servicios con su clase ServicioPersona
Fuente: Desafio Latam

Paso 4: Al interior de la clase `ServicioPersona`, crear el método `crearPersona()`, 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.

```
import 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";
      }
   }
}
```

Paso 5: Crear el método `actualizarPersona()` que 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 el mensaje "Actualizada".

```
import 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";
     }
   }
}
```

Paso 6: Agregar el método `listarPersonas()` que retorna `personasDB`, el cual es el mapa que se utiliza como almacén de datos.

```
import 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;
  }
}
```

Paso 7: Agregar el método `eliminarPersona()` 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 "Eliminada".

```
import 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";
    }
  }
}
```

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

Paso 8: Crear la clase `ServicioPersonaTest` dentro de la carpeta `src/test` del proyecto, como se muestra a continuación:

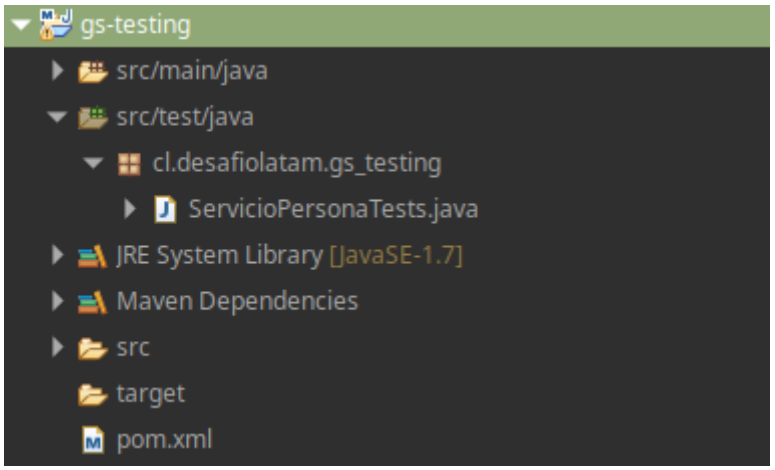

Imagen 9. Creación Test Unitarios en la carpeta src/test/java.
Fuente: Desafio Latam.

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

```
package cl.desafiolatam.gs_testing;
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. Un método contiene un log con el nombre de la prueba y tienen la anotación `@DisplayName` que indica el nombre.

Paso 9: Crear el método llamado `testCrearPersona` para el método `crearPersona()`, además de sus respectivas anotaciones. Lo siguiente es crear un objeto de tipo Persona, pasándole datos a través del constructor que será la persona a guardar, por ejemplo, Juanito. Se crea una variable `respuestaServicio,` la cual almacenará el valor de la respuesta del servicio `crearPersona` y recibe como parámetro a "Juanito", retornando un String. Finalmente, se usa `assertEquals` para comprobar que lo esperado fue "creado" y las respuestas del servicio son iguales, pasando la prueba.

```
import cl.desafiolatam.gs_testing;
import org.junit.jupiter.api.*;
import java.util.logging.Logger;
import static org.junit.jupiter.api.Assertions.assertEquals;
import servicios.ServicioPersona;
import modelos.Persona;
import org.junit.Test;

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

  @Test
  public 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 prueba testCrearPersona, se debe aplicar el botón Run As - > Junit Test. La salida de ese comando es:

```
Feb 12, 2021 1:54:17 PM cl.desafiolatam.gs_testing.ServicioPersonaTest
testCrearPersona
INFO: info test crear persona
```

Además, nos aparecerá una pantalla con una barra de color verde. Esto demuestra que el test ha salido con éxito.

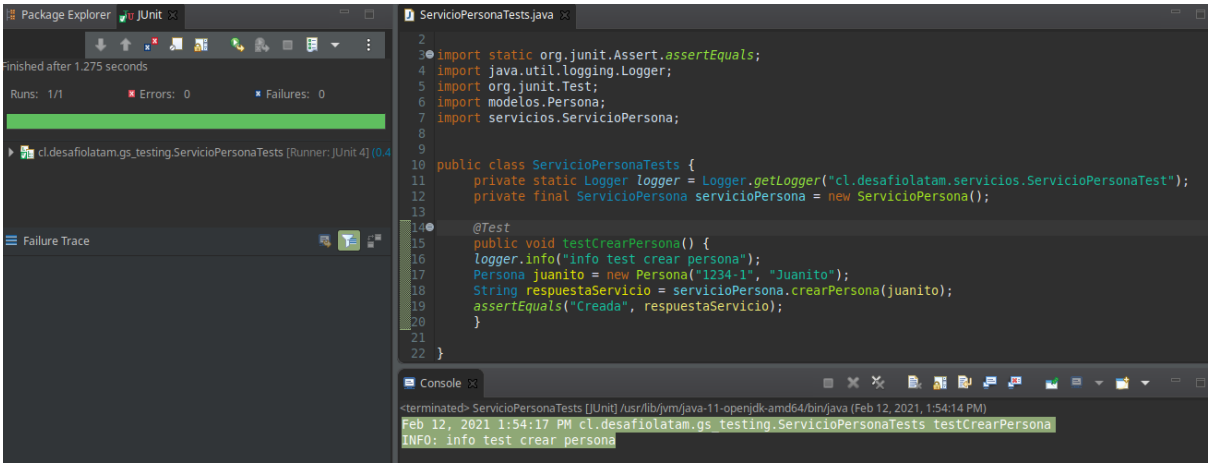

Imagen 10. Visión panorámica de un Test en Eclipse.
Fuente: Desafio Latam.

Paso 10: Crear el 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" en este caso. Se crea una variable `respuestaServicio` la cual almacenará el valor de la respuesta de `actualizarPersona`, esta recibe como parámetro a Pepe y retorna un String. Finalmente, se usa `assertEquals` para comprobar que lo esperado "Se actualizó" y la respuesta del servicio son iguales, pasando la prueba.

```
import cl.desafiolatam.gs-testing;
import org.junit.jupiter.api.*;
import java.util.logging.Logger;
import static org.junit.Assert.assertEquals;
import servicios.ServicioPersona;
import modelos.Persona;
import org.junit.Test;

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

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

```
assertEquals("Se actualizo", respuestaServicio);
 }
}
```

En consola aparecerá lo siguiente:

```
Feb 12, 2021 1:59:04 PM cl.desafiolatam.gs_testing.ServicioPersonaTest
testActualizarPersona
INFO: info actualizar persona
```

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”.

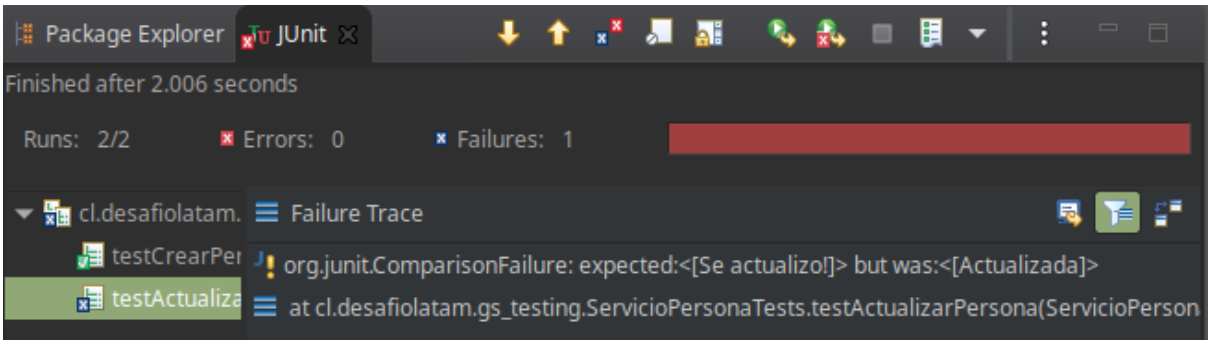

Imagen 11. Falla en el test.
Fuente: Desafio Latam.

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 aserción. Se debe corregir la prueba para comprobar si la salida es correcta.

```
import cl.desafiolatam.gs-testing;
import org.junit.jupiter.api.*;
import java.util.logging.Logger;
import static org.junit.Assert.assertEquals;
import servicios.ServicioPersona;
import modelos.Persona;
import org.junit.Test;
public class ServicioPersonaTest {
  private static Logger logger =
  Logger.getLogger("cl.desafiolatam.servicios.ServicioPersonaTest");
private final ServicioPersona servicioPersona = new ServicioPersona();

@Test

public 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 Maven test con testActualizarPersona modificada, resulta exitosa.

```
Feb 12, 2021 2:04:31 PM cl.desafiolatam.gs_testing.ServicioPersonaTest
testCrearPersona
INFO: info test crear persona
Feb 12, 2021 2:04:32 PM cl.desafiolatam.gs_testing.ServicioPersonaTest
testActualizarPersona
INFO: info actualizar persona
```

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, daremos 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, 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
}
```

Descripción Paso 2: Test para el método listarPersona()

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

  @Test

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