Pruebas unitarias y TDD (Parte I)

Pruebas Unitarias

Competencias

- Comprender la importancia de las pruebas unitarias para su uso en diversos códigos de Java.
- Comprender cómo cambia el flujo de desarrollo de un código al implementar pruebas unitarias.

Introducción

Es común encontrarse en situaciones donde se deben realizar cambios al sistema, pero ¿cómo validamos estos cambios y comprobamos que estén buenos sin la necesidad de acudir a recursos externos? La respuesta es simple y se llama “Prueba o Test Unitario”.

Este proceso evita la espera de verificaciones externas y nos permite comprobar funcionalidades nuevas que podemos adherir a nuestro código. Por lo tanto, si ocurre un fallo en producción no debemos esperar hasta final para realizar verificaciones, sino que podemos corregir problemas a tiempo. Las nuevas funcionalidades son parte del día a día en los/las desarrolladores/as y debemos entender muy bien estos conceptos para mejorar nuestro código.

Test Unitarios

Las pruebas o test unitarios vienen a verificar las unidades individuales del código para que cumplan con el funcionamiento esperado. Además, ayudan a comprender el diseño del código en el que se está trabajando. Cuando se trabaja en un equipo de varios integrantes que realizan cambios casi a diario en el código de la aplicación, las pruebas unitarias toman un rol importante, ya que mantienen la calidad del código y la integración de las funcionalidades nuevas para utilizarlas ahora y no después. Para esto, comenzaremos definiendo conceptos importantes como lo son los ciclos de vida en los test unitarios.

Ciclo de vida en Test Unitarios

Cuando se adoptan las pruebas unitarias se modifican los procesos por los cuales pasan características añadidas al código. Por ejemplo, cuando un/a integrante del grupo de trabajo efectúa cambios en la aplicación, las pruebas unitarias deberían seguir corriendo frente a estos cambios. Por lo tanto, cada vez que trabajemos sobre la aplicación, debemos hacer una actualización sobre los últimos cambios y, en base a estos, comenzar a realizar pruebas.

A continuación, te mostraremos el diagrama del ciclo de las pruebas unitarias (Imagen 1) que se aplica en cada cambio:

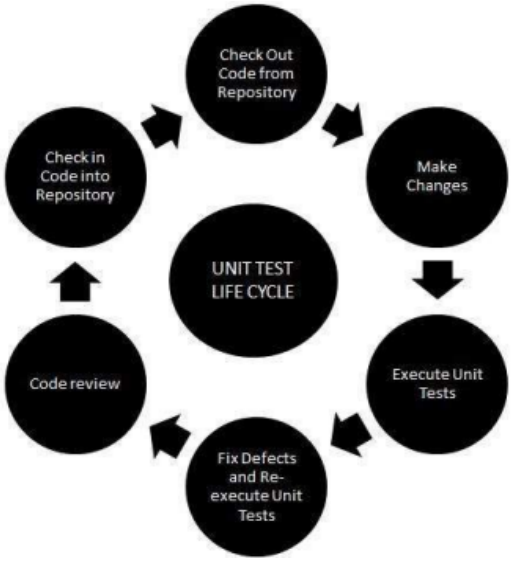

Imagen 1. Ciclo de vida en Test Unitario.
Fuente: Desafío Latam.

1. El proceso comienza descargando el código desde su respectivo repositorio o lugar donde se almacena el código fuente (“Check Out Code from Repository”).

2. Se realizan los cambios y se agregan nuevas funcionalidades (“Make Changes”).

3. Se escriben las pruebas unitarias y son ejecutadas (“Execute Unit Tests”).

4. Se corrigen las pruebas fallidas y se ejecutan nuevamente, al ser exitosas se sigue con el flujo (“Fix Defects and Re-execute Unit Tests”).

5. De forma optativa, se puede hacer una revisión de código por parte de otro miembro del equipo para validar los cambios (“Code Review”).

6. Se confirman y se suben los cambios al repositorio (“Check in Code into Repository”).

Ventajas de realizar pruebas unitarias

- Se escriben pequeñas pruebas, lo que obliga a que el código sea 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 entre miembros de equipo, ya que se puede editar el código de cada uno con confianza. Las pruebas unitarias 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 que puede ser intimidante y no es fácil de aprender al comienzo.
- Difícil de aplicar al código heredado existente.

Las dificultades que vienen con las pruebas unitarias están en el cómo se vende el concepto a una organización. A veces, se rechazan las pruebas unitarias porque el desarrollo conlleva más tiempo y retrasa la entrega. En algunos casos esto puede ser cierto, pero no hay forma de saber si esos retrasos son una consecuencia real de las pruebas unitarias o por un diseño deficiente.

Sin embargo, al lograr un buen diseño y con requisitos bien definidos, las pruebas unitarias entregan muchos beneficios, a continuación, detallaremos algunos:

1. **Facilita realizar 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 procede a refactorizar con confianza. Las pruebas unitarias colaboran con la agilidad porque se basa en pruebas que permiten realizar cambios con mayor facilidad al refactorizar.

2. **Mejora la 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.

3. **Encuentra errores en el diseño:** Los problemas se encuentran en una etapa temprana, dado que los/las desarrolladores/as que prueban un código individual antes de la integración realizan estas pruebas unitarias. Los errores encontrados se resuelven en cualquier momento sin afectar las otras partes del código, y puede incluir errores en la implementación del programador y fallas o partes faltantes de la especificación de la unidad.

4. **Proporciona documentación:** Las pruebas unitarias pueden incluso ser usadas como documentación del sistema. Para los desarrolladores que buscan entender qué característica proporciona y cómo se utiliza cada unidad, las pruebas muestran la funcionalidad de esta. Por otra parte, si las pruebas son claras y expresan sus intenciones de forma concisa, el código será fácil de entender.

Gestores en el ciclo de vida

La gestión del ciclo de vida brinda un marco para el desarrollo de software y permite gestionar sistemas a lo largo del tiempo. Por otra parte, el desarrollo del ciclo de vida en aplicaciones involucra a personas, herramientas y procesos que gestionan una aplicación desde que se diseña hasta que deja de estar disponible.

Los gestores son sistemas automatizados que buscan simplificar los procesos de construcción (limpiar, compilar y generar ejecutables del proyecto a partir del código fuente). En esta unidad abordaremos Apache Maven como gestor.

Maven es una herramienta de gestión y comprensión de proyectos de software que se basa en el concepto “POM” o “Project Object Model”. Permite gestionar todo aquello que está relacionado a la compilación, los informes y la documentación a partir de una información central.

El archivo `pom.xml` es el núcleo de configuración para un proyecto en Maven. El POM es enorme y puede ser desalentador en cuanto a su complejidad de código, sin embargo, no es necesario entender todo, si no que aquello que utilizaremos.

Ejercicio guiado: Ciclo de vida

A continuación, ordena del 1 al 6 las etapas del ciclo de vida de las pruebas unitarias.

- Rellenar columna “Posición en el ciclo de vida” según el lugar correspondiente.

Conceptos	**Posición en el ciclo de vida**
Se confirman y se suben los cambios al repositorio (“Check in Code into Repository”).	
Se corrigen las pruebas fallidas y se ejecutan nuevamente, al ser exitosas se sigue con el flujo (“Fix Defects and Re-execute Unit Tests”).	
Se escriben las pruebas unitarias y son ejecutadas (“Execute Unit Tests”).	
Se puede hacer una revisión de código por parte de otro miembro del equipo para validar los cambios (“Code Review”).	
Se descarga el código desde su respectivo repositorio o lugar donde se almacena el código fuente (“Check Out Code from Repository”).	
Se realizan los cambios y se agregan nuevas funcionalidades (“Make Changes”).	

Tabla 1. Ciclo de vida en Test Unitario.
Fuente: Desafío Latam.

Solución Tabla 1:

Conceptos	Posición en el ciclo
Se confirman y se suben los cambios al repositorio (“Check in Code into Repository”).	6°
Se corrigen las pruebas fallidas y se ejecutan nuevamente, al ser exitosas se sigue con el flujo (“Fix Defects and Re-execute Unit Tests”).	4°
Se escriben las pruebas unitarias y son ejecutadas (“Execute Unit Tests”).	3°
De forma optativa, se realiza revisión de código por parte de otro miembro del equipo para validar los cambios (“Code Review”).	5°
Se descarga el código desde su respectivo repositorio o lugar donde se almacena el código fuente (“Check Out Code from Repository”).	1°
Se realizan los cambios y se agregan nuevas funcionalidades (“Make Changes”).	2°

Tabla 2. Solución tabla 1.
Fuente: Desafío Latam.

Ejercicio propuesto (1)

- Formar grupos de dos o tres personas.
- Elegir un contenido específico del capítulo que les haya llamado la atención o que consideren importante.
- Con el contenido seleccionado y la ayuda de internet, profundizar y complementar la información respondiendo a las siguientes preguntas:
    - ¿Qué es?, ¿para qué nos sirve? y ¿cuál es su importancia?
- Con las respuestas deben realizar un podcast (grabación de audio), donde exista una conversación entre el “entrevistador” y el “experto en contenido”.

JUnit

Competencias

- 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 a 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 método 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.

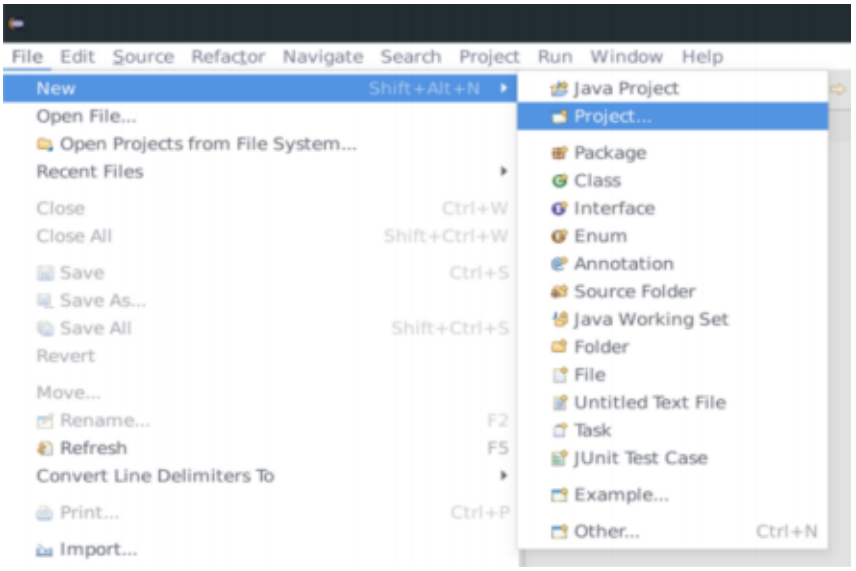

Imagen 2. Creando un nuevo proyecto.
Fuente: Desafio Latam.

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.`

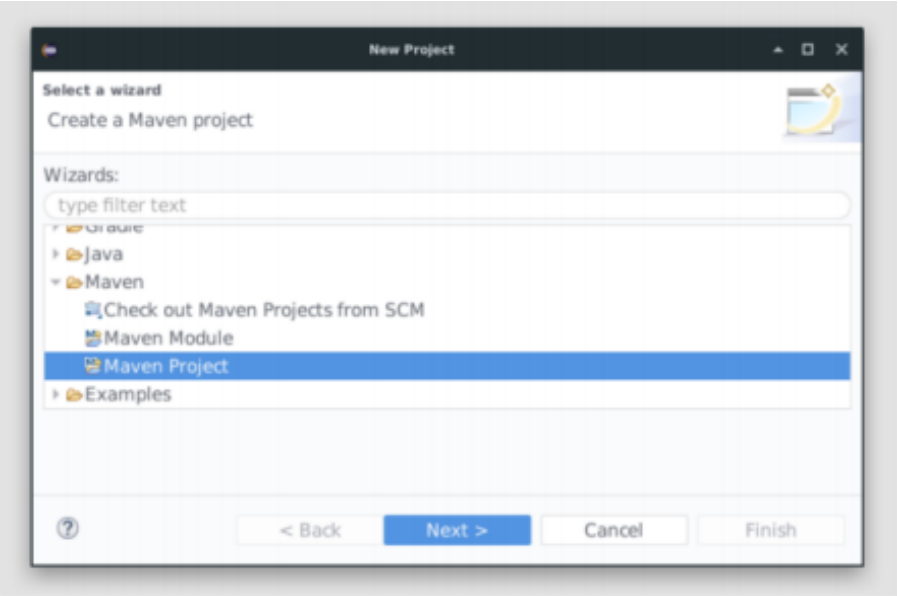

Imagen 3. Creando un proyecto Maven.
Fuente: Desafio Latam.

Paso 3: Seleccionar la ubicación del espacio de trabajo.

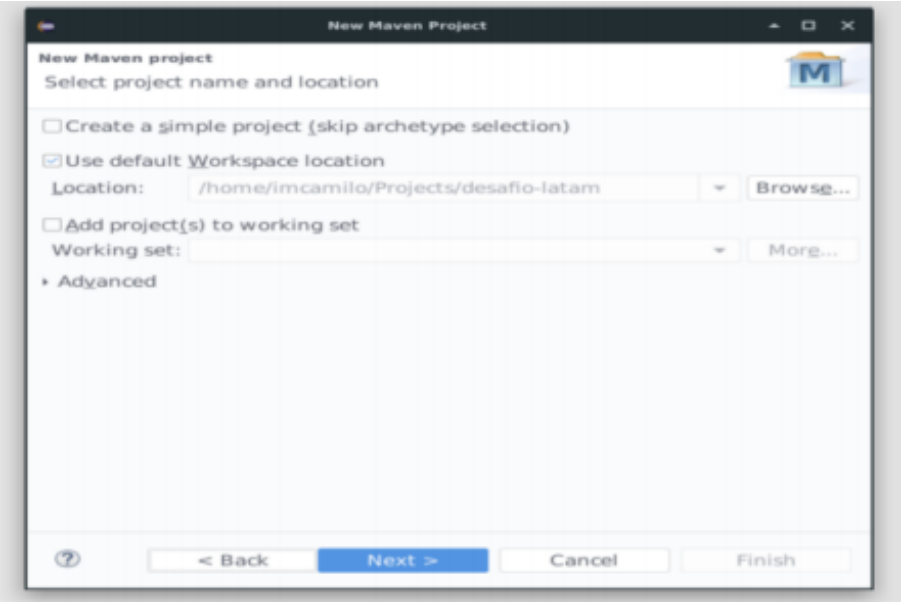

Imagen 4. Select Project Name and Location.
Fuente: Desafio Latam.

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:

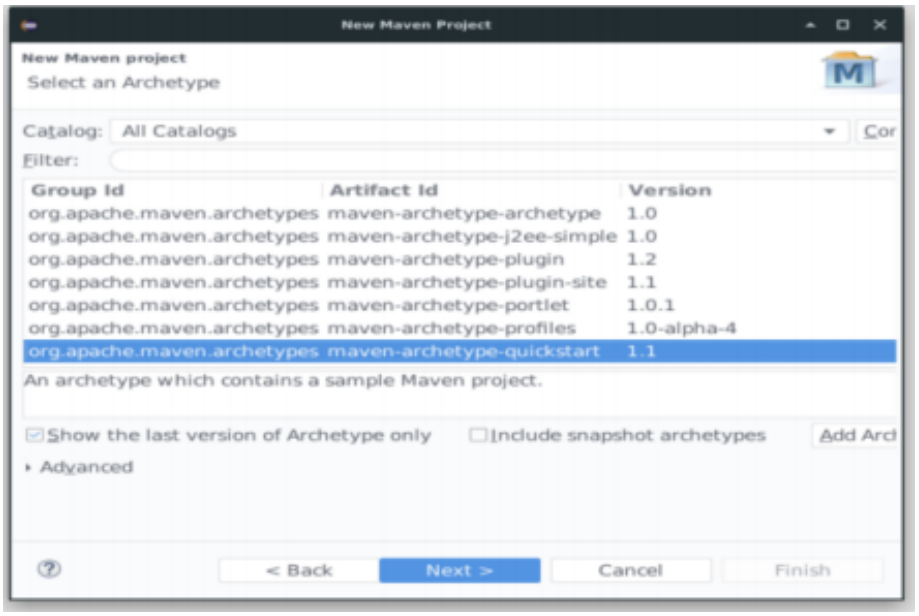

Imagen 5. Select Archetype.
Fuente: Desafio Latam.

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.

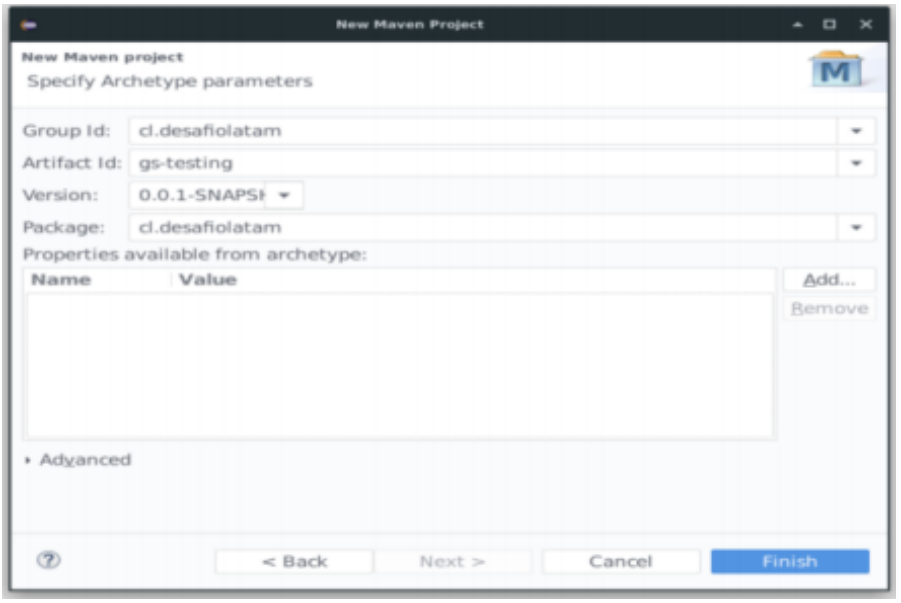

Imagen 6. Specify Archetype Parameters.
Fuente: Desafio Latam.

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.

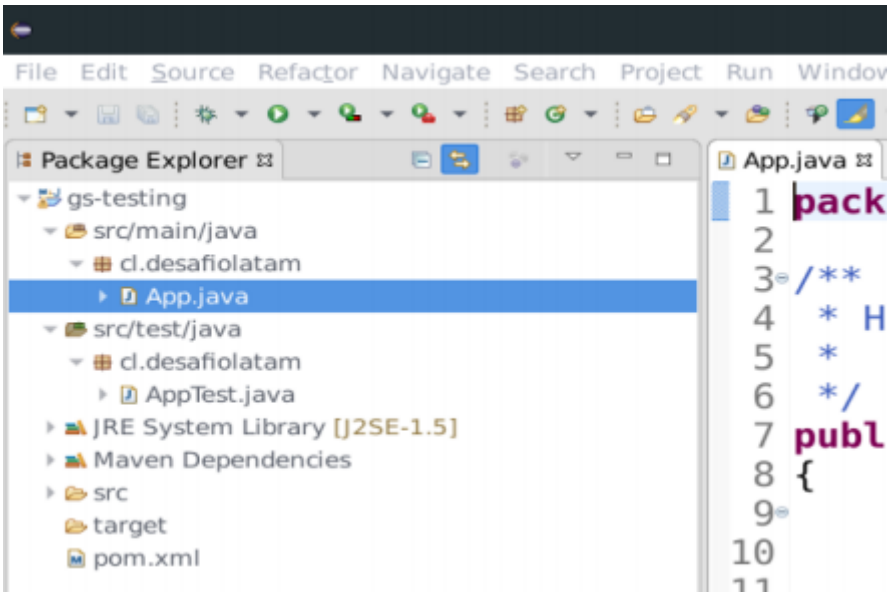

Imagen 7. Getting Started.
Fuente: Desafio Latam.

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`.

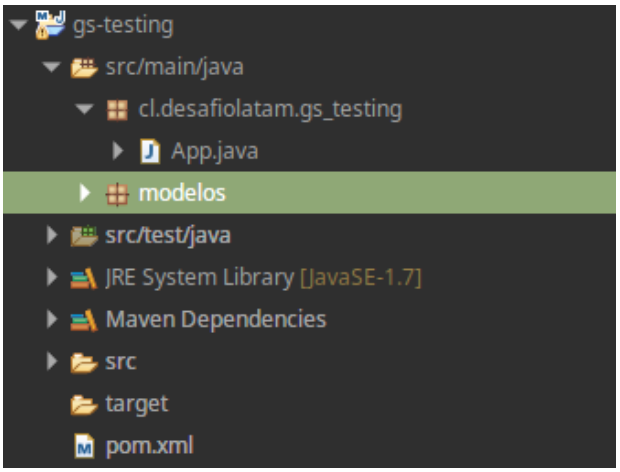

Imagen 8. Creación package modelo.
Fuente: Desafio Latam.

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`.

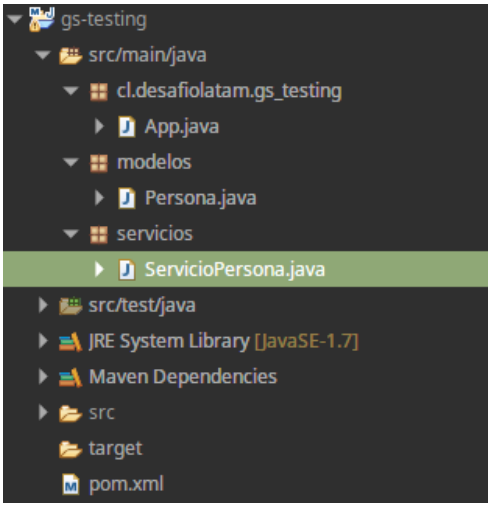

Imagen 9. Creación package servicios con su clase ServicioPersona
Fuente: Desafio Latam

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:

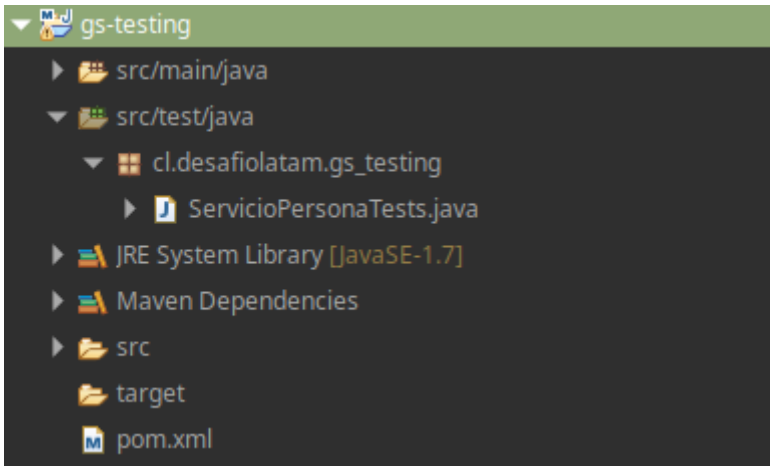

Imagen 10. Creación Test Unitarios en la carpeta src/test/java.
Fuente: Desafio Latam.

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.

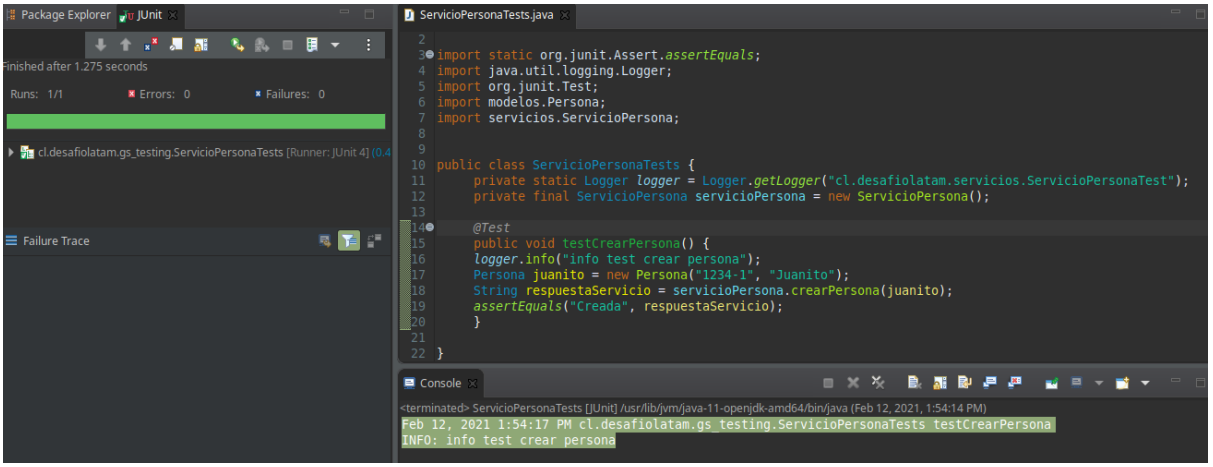

Imagen 11. Visión panorámica de un Test en Eclipse.
Fuente: Desafio Latam.

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”.

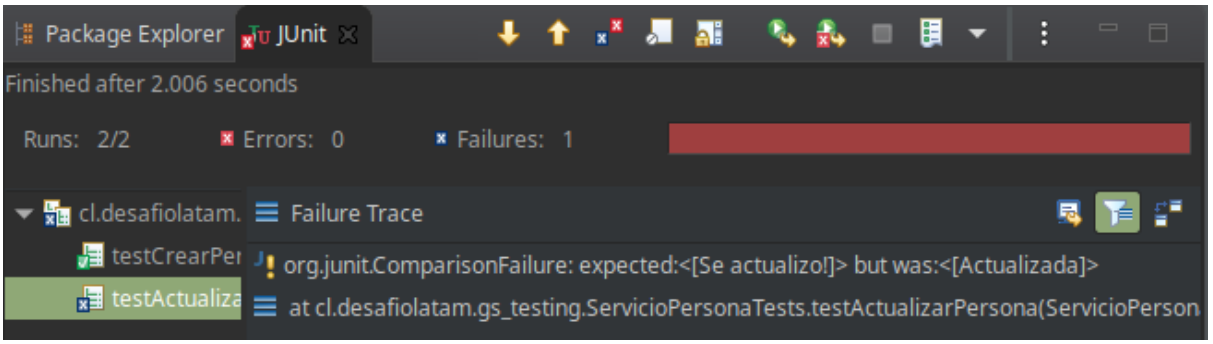

Imagen 12. Falla en el test.
Fuente: Desafio Latam.

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
}
```

Ejercicio Propuesto (2)

En base al ejercicio guiado “Pruebas” que hemos visto previamente, se le pide realizar tests sobre los siguientes métodos ocupados en la clase ServicioPersona. Para ello, se le pedirá hacer las siguientes pruebas unitarias:

- Eliminar Persona Test.
- Listar Persona Test.

Solución Ejercicio Propuesto (2)

Descripción Paso 1: Test para el método eliminarPersona()

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

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

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);
 }
}
```