Tests dobles

¡Comencemos!

¿Qué aprenderás?

- Comprender qué son los dobles en test para utilizarlos en Java.
- Usar Mocks utilizando Mockito para simular métodos.

Introducción

Para introducir este capítulo, imaginemos que necesitamos probar una aplicación que interactúe con una pasarela de pagos. Podríamos usar datos ficticios cada vez que se ejecute una prueba y esto podría ser demasiado lento, ya que de producirse algún error existirá la duda sobre si falló la pasarela o el código generado. Inclusive, es probable que la plataforma contra la cual buscas ejecutar las pruebas no esté disponible. Es por esto que nos resulta conveniente generar estas pruebas a medida que la aplicación va tomando forma y no cuando esté finalizada.

A continuación, veremos algunos métodos con los cuales puedes abordar de manera sencilla la problemática.

¡Vamos con todo!

¿Qué son los “dobles de prueba” o “Test Dobles”?

Es un término genérico para cualquier tipo de objeto de simulación utilizado (en lugar de un objeto real) para propósitos de prueba.

¿Cuándo usar los dobles de prueba?

Cuando hablamos de pruebas, debemos entender que estas simulan componentes para no utilizar los que funcionan en producción. Esta propuesta que parece sencilla de explicar, puede resultar bastante complicada cuando se hace uso de servicios de terceros. Por lo tanto, necesitamos un mecanismo que permita contar con “dobles” o “impostores” de estos servicios y evitar así estar llamándolos durante las pruebas. Aquí entran en juego los “dobles de test” que facilitan la simulación de estos componentes o servicios.

Uno de los objetivos de la programación es que el código sea lo más fácil de leer y duradero en el tiempo. Una forma de conseguirlo es aplicando buenas prácticas o “Clean Code”. Según la clasificación de Fowler, se pueden obtener varios tipos de dobles de prueba, como por ejemplo:

Dummy

Son dobles de prueba que se pasan donde sean necesarios para completar la signatura de los métodos empleados, pero no intervienen directamente en la funcionalidad que se está probando. Son generalmente de relleno.

Fake

Son implementaciones de componentes funcionales y operativos de la aplicación, pero que buscan el mínimo de características para pasar las pruebas. No son adecuados para ser desplegados en producción, pero simplifican la versión del código.

Un ejemplo de esto es la implementación en memoria de un repositorio. Esto nos permite realizar pruebas de integración en servicios sin iniciar una base de datos y evitando así las solicitudes que consumen mucho tiempo. Esto puede ser útil en la creación de prototipos, ya que nos permite tomar decisiones sobre la base de datos para más adelante.

Un ejemplo de lo anterior se realiza en el diagrama mostrado en la “Imagen 1”, donde hay un acceso directo de una implementación en memoria a los datos o Repositorio. Esta implementación falsa no comprometería la base de datos, pero usará una colección simple para almacenar datos (HashMap), permitiendo así realizar pruebas de integración de servicios sin iniciar una base de datos.

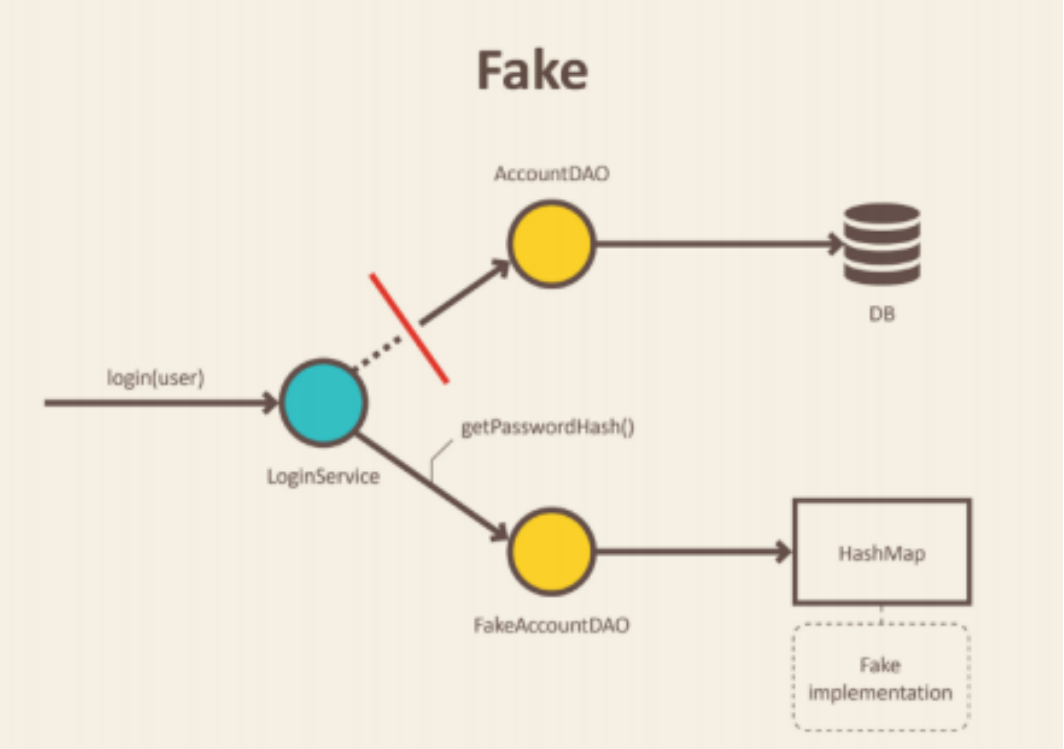

Imagen 1. Fake Diagram.
Fuente: Desafio Latam.

Stub

Es un conjunto de respuestas empaquetadas que se ofrecerán como resultado de una serie de llamadas a nuestro doble de prueba. Puede entenderse como un objeto que contiene datos predefinidos y lo utiliza para responder llamadas durante las pruebas.

Se utiliza para no involucrar objetos que responderían con datos reales o tendrían efectos secundarios no deseados. Sería, por ejemplo, el resultado de una consulta a base de datos que puede realizar un repositorio o un mapper. Es importante comentar que en este tipo de dobles únicamente se hace énfasis al estado que tienen estos objetos y nunca a su comportamiento o relación con otras entidades.

Un ejemplo de lo anterior es el diagrama mostrado en la "Imagen 2", donde un objeto necesita tomar algunos datos desde la base de datos, sin embargo, para responder a una llamada del método `averageGrades`, en lugar del objeto real, se usa un código auxiliar y se define qué datos deberían retornar.

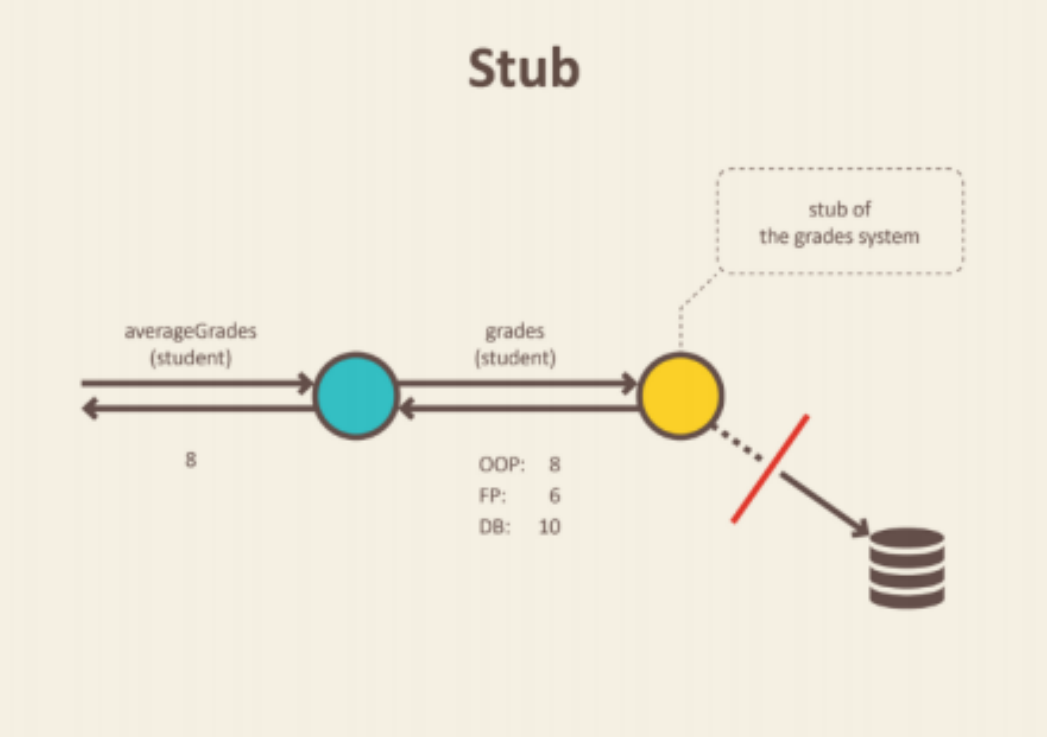

Imagen 2. Stub Diagram.
Fuente: Desafio Latam.

Mock

Son objetos que registran las llamadas que reciben. En la afirmación de una prueba se puede verificar que se realizaron todas las llamadas a métodos y acciones esperadas. Se usa Mock cuando no se quiere invocar el código de producción o cuando no existe una manera fácil de verificar que se ejecutó el código deseado. No hay un valor de retorno ni una forma fácil de verificar el cambio de estado del sistema.

Un ejemplo puede ser una funcionalidad que llame al servicio de envío de correo electrónico o un servicio cualquiera que interactúe con una base de datos. Con esto buscamos verificar los resultados de la funcionalidad cuando se ejerzan las pruebas. La idea es que estas sean capaces de analizar cómo se relacionan los distintos componentes, permitiendo verificar si un método concreto ha sido invocado o no, qué parámetros han recibido o cuántas veces lo hemos ejercido.

Por otra parte, Mock nos ayuda a probar la comunicación entre objetos. Las pruebas deben ser expresivas y transmitir la intención de forma clara a la hora de crear pruebas, ya que no pueden depender de otros servicios o bases de datos externas. Los dobles de prueba son herramientas muy útiles en la gestión del estado si se saben usar.

En el siguiente diagrama de la "Imagen 3", veremos que después de la ejecución del método `securityOn`, las ventanas y puertas simularon todas las interacciones. Esto permite verificar que los objetos de puertas y ventanas detonaron sus métodos para cerrarse.

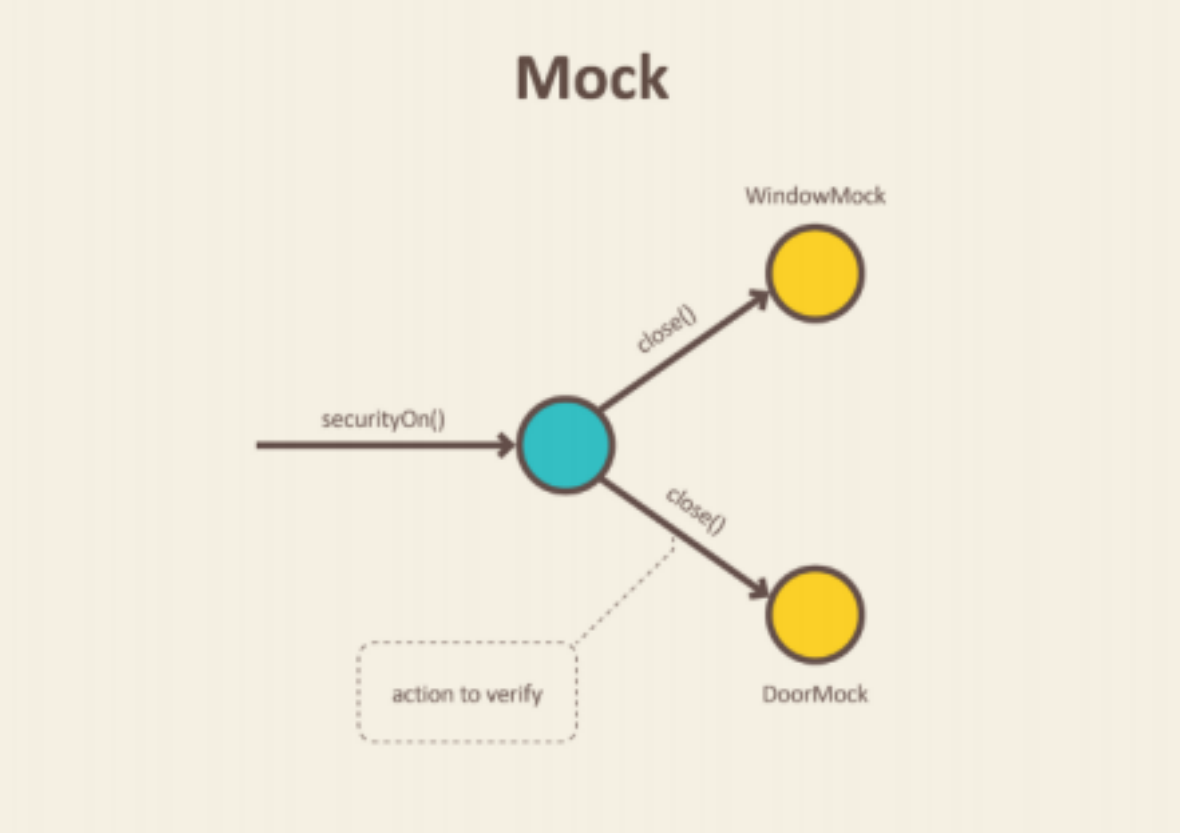

Imagen 3. Mock Diagram
Fuente: Desafio Latam

Mockito

En este caso nos centraremos en Mocks utilizando Mockito, el cual permite escribir pruebas expresivas ofreciendo una API simple. Además es una de las bibliotecas para Java más populares en GitHub, rodeada de una gran comunidad.

Ejercicio guiado: Mockito

La forma recomendada para agregar Mockito al proyecto es añadir la dependencia de la biblioteca "mockito-core" utilizando tu sistema de compilación favorito.

Paso 1: Crear un nuevo proyecto del tipo Maven y lo llamamos “gs-tdd”.

Paso 2: Para comenzar a trabajar con Maven, debemos ir al archivo pom.xml en la raíz del proyecto y agregar la dependencia dentro del tag `dependencies`.

```
<dependencies>
   <!--resto de dependencias-->
    <dependency>
       <groupId>org.mockito</groupId>
       <artifactId>mockito-core</artifactId>
       <version>2.28.2</version>
       <scope>test</scope>
    </dependency>
</dependencies>
```

Paso 3: Crear un paquete llamado modelos y otro paquete llamado repositorio.

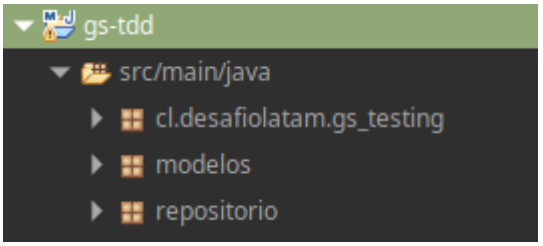

Imagen 4. Creación de package
Fuente: Desafio Latam

Paso 4: Crear la clase `Persona` que contiene 2 atributos: Rut y nombre. Generar los getter and setter correspondientes, su constructor y el método `toString()`.

```
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 5: Crear la clase RepositorioPersona dentro de la carpeta src/main, con el objetivo que simule una interacción con una base de datos.

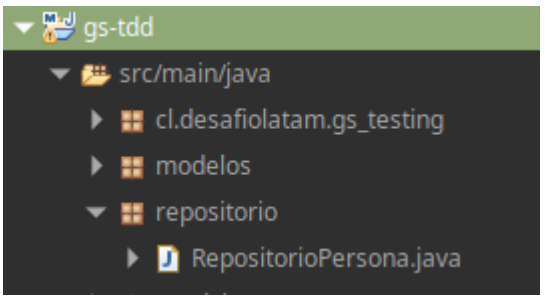

Imagen 5. Creación clase RepositorioPersona
Fuente: Desafio Latam

Paso 6: La clase `RepositorioPersona` contiene los métodos “crear”, “actualizar”, “listar” y “eliminar” una persona que se importan desde la clase Persona en la carpeta modelos. Esta clase será utilizada por otros servicios dentro del sistema. El código del repositorio queda así:

```
package repositorios;
import modelos.Persona;
import java.util.HashMap;
import java.util.Map;

public class RepositorioPersona {
    private Map<String, String> db = new HashMap<>();
    public String crearPersona(Persona persona) {
    db.put(persona.getRut(), persona.getNombre());
   return "OK";
 }
   public String actualizarPersona(Persona persona) {
    db.put(persona.getRut(), persona.getNombre());
    return "OK";
 }
   public Map<String, String> listarPersonas() {
   return db;
  }
   public String eliminarPersona(Persona persona) {
   db.remove(persona.getRut());
   return "OK";
  }
}
```

Paso 7: Crear la clase RepositorioPersonaTest dentro de la carpeta src/test del proyecto en la ruta y directorios que se muestran a continuación:

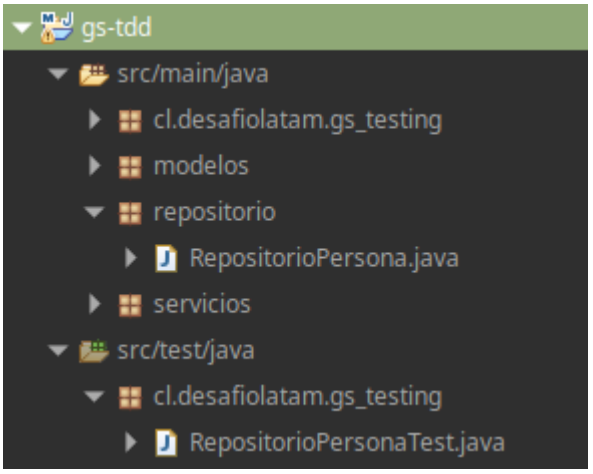

Imagen 6. Creación las clase RepositorioPersonaTest
Fuente: Desafio Latam

Al escribir las pruebas unitarias es probable que aparezcan algunos problemas como que la unidad bajo prueba depende de otros componentes, o que la duración de la configuración para realizar la prueba unitaria es tiempo que excede el alcance del desarrollo. Para evitar lo anterior, se pueden utilizar Mocks en lugar de estos componentes y continuar con la prueba de la unidad.

Pensemos en el servicio que contiene una lógica de negocio de verificaciones y flujos a la base de datos entrante. Cuando el flujo continúa de forma normal, este envía los datos ya procesados hacia el repositorio y los guarda en una base de datos. Sin embargo, en un ambiente de prueba no se puede apuntar al repositorio para guardar los datos, ya que con cada prueba se haría trabajar a la base de datos con lecturas o escrituras. Por lo tanto, se debe simular el repositorio para que cuando las pruebas ejecuten los métodos del servicio, el repositorio devuelva los estados que corresponden al flujo como si fuese el normal.

Paso 8: Crear el objeto simulado de `RepositorioPersona` con el método estático `Mock` . El cual crea un `Mock` dada una clase o una `interface`.

```
package repositorio;
import static org.mockito.Mockito.mock;

public class RepositorioPersonaTest {
  private RepositorioPersona repositorioPersona =
  mock(RepositorioPersona.class);
}
```

Paso 9: En la clase RepositorioPersonaTest, crear el método `testCrearPersona` que tiene su anotación `@Test`. Crear un objeto llamado Pepe de tipo Persona.

```
package repositorio;
import modelos.Persona;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
import static org.mockito.Mockito.mock;

 public class RepositorioPersonaTest {
   private RepositorioPersona repositorioPersona =
   mock(RepositorioPersona.class);

 @Test
 public void testCrearPersona() {
 Persona pepe = new Persona("1-2", "Pepe");
  }
}
```

Paso 9.1: Luego, se habilita la simulación de los métodos con el método estático “When” importado desde org.mockito.Mockito. Se usa cuando se desea que el simulacro devuelva un valor particular al llamar a un método particular. Simplemente colocamos: "Cuando se llama al método x, devuelve y". Con el método “thenReturn” el cual también viene desde org.mockito.Mockito se establece un valor de retorno que se devolverá cuando se llame al método.

```
package repositorio;
import modelos.Persona;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.when;

 public class RepositorioPersonaTest {
   private RepositorioPersona repositorioPersona =
   mock(RepositorioPersona.class);

  @Test
  public void testCrearPersona() {
  Persona pepe = new Persona("1-2", "Pepe");
  when(repositorioPersona.crearPersona(pepe)).thenReturn("OK");
 }
}
```

Paso 9.2: Sin embargo, las simulaciones pueden devolver valores diferentes según los argumentos pasados a un método, para esto se pueden establecer las excepciones que se lanzan cuando se llama al método, usando `thenThrow`, como por ejemplo:

```
package repositorio;
import modelos.Persona;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.when;

  public class RepositorioPersonaTest {

   private RepositorioPersona repositorioPersona =
   mock(RepositorioPersona.class);
```

```
  @Test
  public void testCrearPersona() {
  Persona pepe = new Persona("1-2", "Pepe");
  when(repositorioPersona.crearPersona(null)).thenThrow(new
  NullPointerException());
 }
}
```

Paso 9.3: Finalmente, crear un String llamado `crearPersonaRes` para almacenar la respuesta del método `crearPersona` antes simulado, y se usa una afirmación para comprobar si lo esperado es un dato de tipo String "OK". Se utiliza el método estático `verify` importado desde `org.mockito.Mockito` para verificar que cierto comportamiento ha ocurrido una vez. En este caso se comprueba que `crearPersona` fue ejecutado.

```
package repositorio;
import modelos.Persona;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.mockito.Mockito.*;

  public class RepositorioPersonaTest {
   private RepositorioPersona repositorioPersona =
   mock(RepositorioPersona.class);

  @Test
  public void testCrearPersona() {

  Persona pepe = new Persona("1-2", "Pepe");

  when(repositorioPersona.crearPersona(pepe)).thenReturn("OK");

  String crearPersonaRes = repositorioPersona.crearPersona(pepe);

  assertEquals("OK", crearPersonaRes);

  verify(repositorioPersona).crearPersona(pepe);

  }
}
```

Paso 10: La salida de Maven Test con las pruebas para el repositorio son las siguientes:

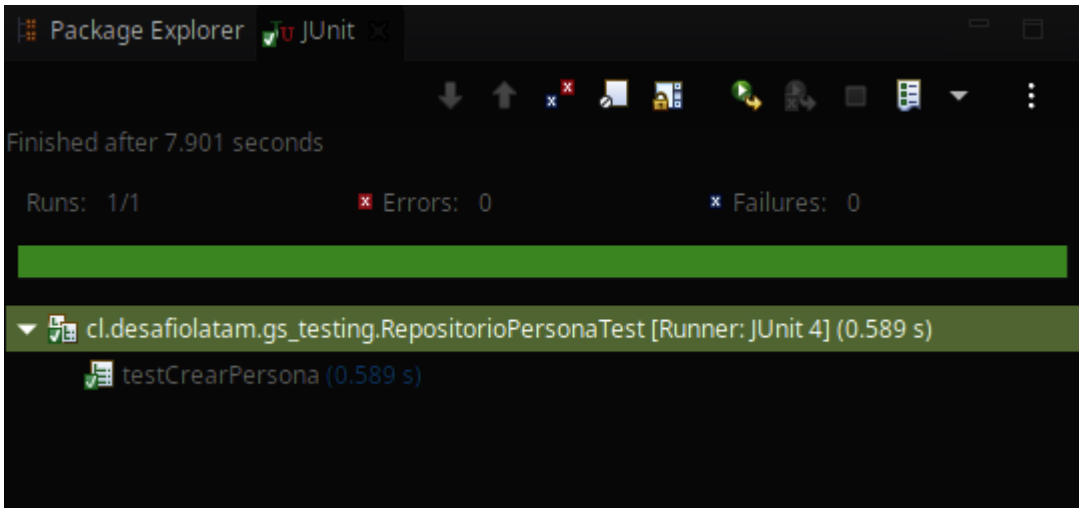

Imagen 7. Test exitoso
Fuente: Desafio Latam

```
[INFO] -------------------------------------------------------
[INFO] T E S T S
[INFO] -------------------------------------------------------
[INFO] c
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
0.317 s - in repositorios.RepositorioPersonaTest
[INFO] Running servicios.ServicioPersonaTest
Jul 07, 2019 3:05:05 PM servicios.ServicioPersonaTest setup INFO: Inicio
clase de prueba
Jul 07, 2019 3:05:05 PM servicios.ServicioPersonaTest
testEliminarPersona
INFO: info eliminar persona
Jul 07, 2019 3:05:05 PM servicios.ServicioPersonaTest testCrearPersona
INFO: info test crear persona
Jul 07, 2019 3:05:05 PM servicios.ServicioPersonaTest
testActualizarPersona
INFO: info actualizar persona
Jul 07, 2019 3:05:05 PM servicios.ServicioPersonaTest testListarPersona
INFO: info listar persona
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
0.009 s - in servicios.ServicioPersonaTest
[INFO] Results:
[INFO] Tests run: 5, Failures: 0, Errors: 0, Skipped: 0
-------------------------------------------------------------------------
-
```

```
[INFO] BUILD SUCCESS
[INFO]
------------------------------------------------------------------------
-
[INFO] Total time: 3.056 s
[INFO] Finished at: 2019-07-07T15:05:05-04:00
[INFO]
------------------------------------------------------------------------
-
```

Paso 11: El método de prueba para `actualizarPersona` luce igual a `testCrearPersona` salvo que se invocan distintos métodos del repositorio.

```
package repositorio;
import modelos.Persona;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.mockito.Mockito.*;

  public class RepositorioPersonaTest {
   private RepositorioPersona repositorioPersona =
   mock(RepositorioPersona.class);
   //resto de la clase

   @Test
   public void testActualizarPersona () {
   Persona juanito = new Persona("1-2", "Juanito");
   when(repositorioPersona.actualizarPersona(juanito)).thenReturn("OK");
String actualizarRes = repositorioPersona.actualizarPersona(juanito);
assertEquals("OK", actualizarRes);
   verify(repositorioPersona).actualizarPersona(juanito);
  }
}
```