# Pruebas unitarias y TDD

## Tests dobles

{desafío} latam_

***Implementar una suite de pruebas unitarias en lenguaje Java utilizando JUnit para asegurar el buen funcionamiento de una pieza de software***

- Unidad 1: Flujo, ciclos y métodos
- Unidad 2: Arreglos y archivos
- Unidad 3: Programación orientada a objetos
- Unidad 4: Pruebas unitarias y TDD

***Te encuentras aquí***

{desafío} latam_

# ¿Qué aprenderás en esta sesión?

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

# ¿Qué entendemos por test dobles?

# /* Test dobles */

# ¿Cuándo usar los dobles de prueba?

- Cuando hablamos de pruebas, debemos entender que estas simulan componentes para no utilizar los que funcionan en producción.

- 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**.

# Buenas prácticas o “Clean Code”

## *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.

# Buenas prácticas o “Clean Code”

## *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.

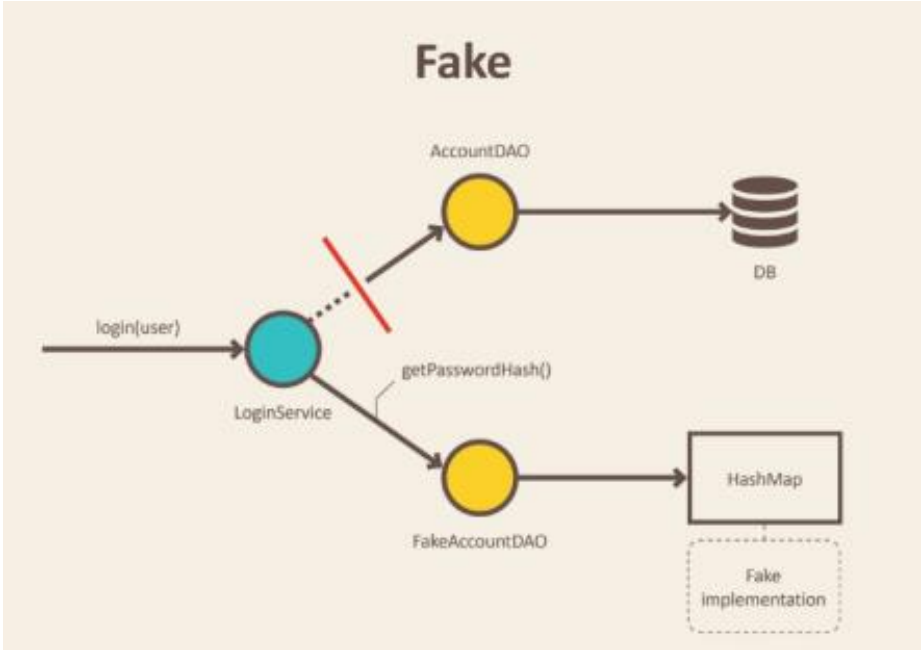
# Buenas prácticas o “Clean Code”

## *Fake*

Ejemplo:

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.



# Buenas prácticas o “Clean Code”

***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.

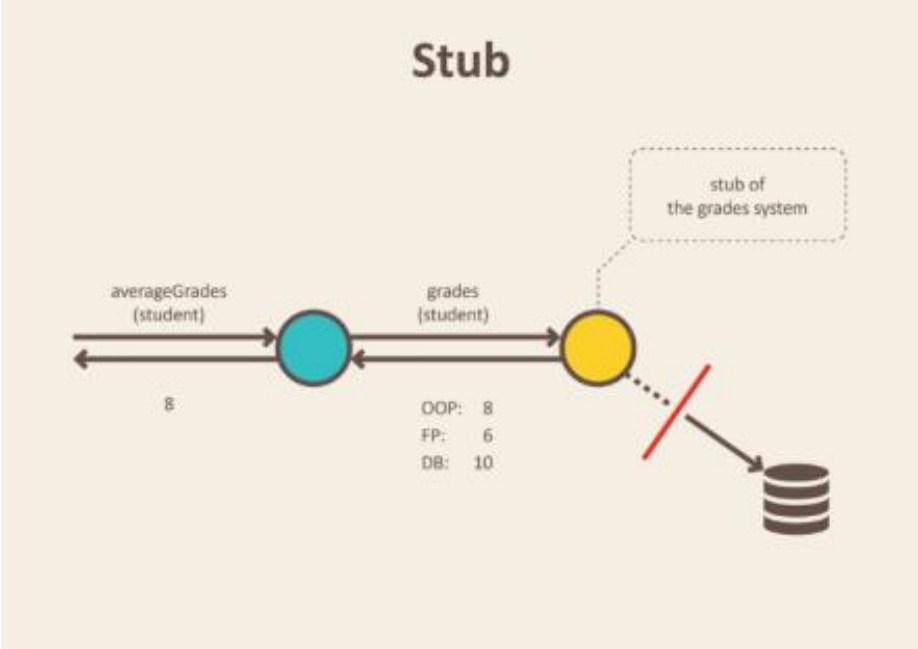
- En este tipo de dobles únicamente se hace énfasis al estado que tienen estos objetos y nunca a su comportamiento o relación con otras entidades.

# Buenas prácticas o “Clean Code”

## *Stub*

Ejemplo:

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.



# Buenas prácticas o “Clean Code”

## *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.

# Buenas prácticas o “Clean Code”

## *Mock*

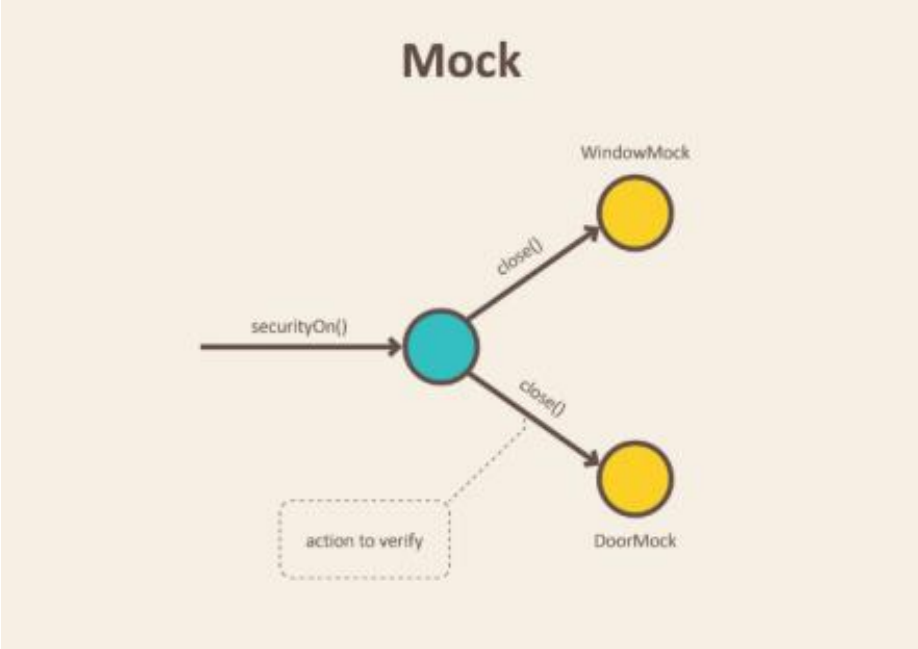
- Nos ayuda a probar la comunicación entre objetos.
- Las pruebas deben ser expresivas y transmitir la intención de forma clara a la hora de crear pruebas, ya que no pueden depender de otros servicios o bases de datos externas.
- Los dobles de prueba son herramientas muy útiles en la gestión del estado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

# Buenas prácticas o “Clean Code”

## *Mock*

Ejemplo:

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.



# Buenas prácticas o “Clean Code”

## *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

## *Paso 1*

Crear un nuevo proyecto del tipo Maven y lo llamamos “gs-tdd”

# Mockito

## *Paso 2*

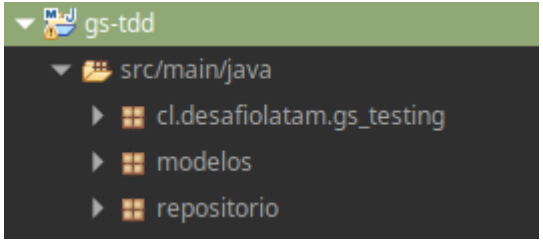
Para comenzar a trabajar con Maven, debemos ir al archivo pom.xml en la raíz del proyecto y agregar la dependencia dentro del tag dependencies.

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

# Mockito

## *Paso 3*

Crear un paquete llamado modelos y otro paquete llamado repositorio.



# Mockito

***Paso 4***

Crear la clase Persona que contiene 2 atributos:
Rut y nombre.

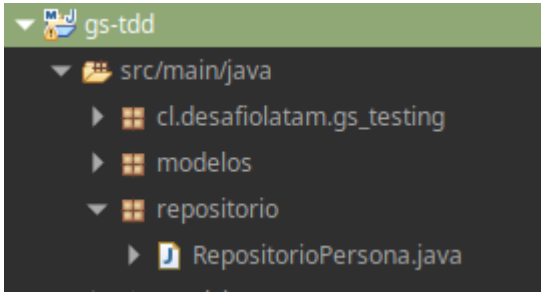
Generar los getter and setter correspondientes, su constructor y el método toString().

```
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;
     }
}
```

# Mockito

## *Paso 5*

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



# Mockito

## *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.

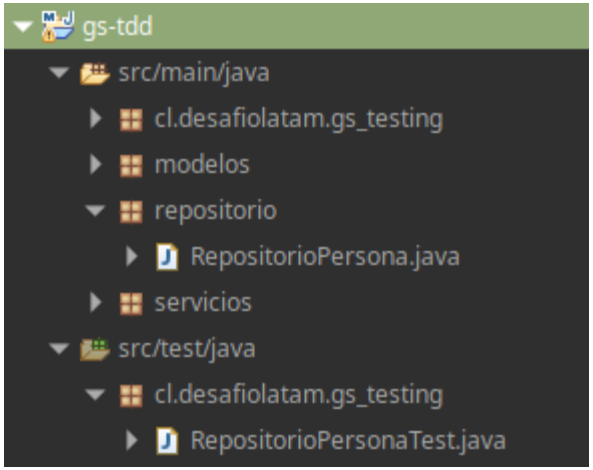
{desafío} latam_

```
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";
     }
}
```

# Mockito

## *Paso 7*

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




{desafío} 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.

# Mockito

## *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);
}
```

# Mockito

## *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");
             }
}
```

# Mockito

## *Paso 9.1*

Habilitar la simulación de los métodos con el método estático “When” importado desde org.mockito.Mockito

```
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");
            }
}
```

# Mockito

## *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());
	}
}
```

# Mockito

## *Paso 9.3*

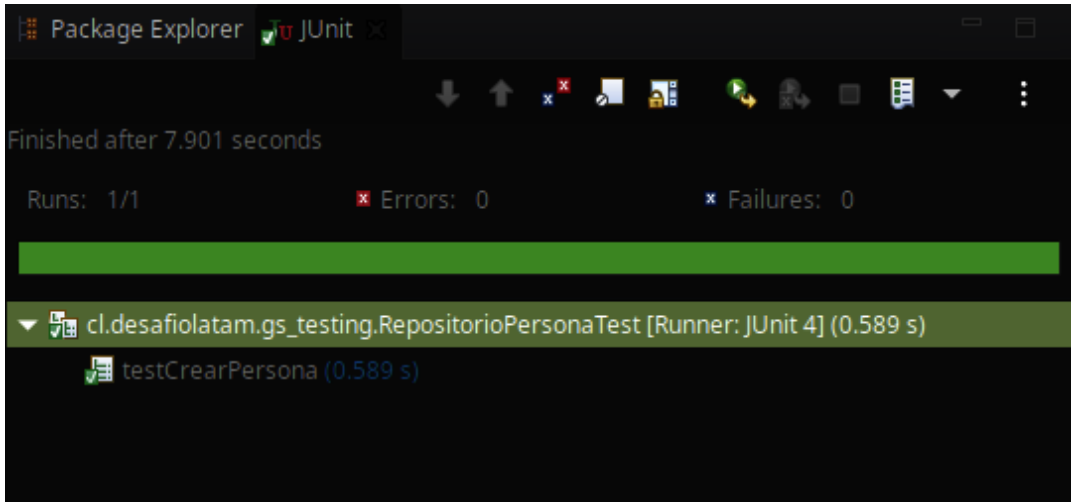
Crear un String llamado crearPersonaRes para almacenar la respuesta del método crearPersona antes simulado, y se usa una afirmación para comprobar si lo esperado es un dato de tipo String "OK".

```
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);

  }
}
```

# Mockito

## *Paso 10*

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




{desafío} latam_

# Mockito

***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);
  }
}
```

# ¿Cuál de los siguientes conceptos pertenecen a dobles de prueba?

1. Dummy

1. False

1. Mock

**Si quisiéramos implementar en memoria un repositorio para realizar pruebas de integración, pero sin iniciar la base de datos, ¿cuál de los siguientes conceptos sirve para implementar este servicio sin comprometer la base de datos?**

1. Stub

1. Mock

1. Fake

# Próxima sesión...

- *Comprender las fases de TDD para ser escritas usando características JUnit.*
- *Desarrollar funcionalidades siguiendo la metodología de TDD para aplicarlas en Java.*

{desafío}
latam_

Academia de
talentos digitales