Pruebas unitarias y TDD (Parte II)

Tests Dobles

Competencias

- 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.

¿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 estos 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.

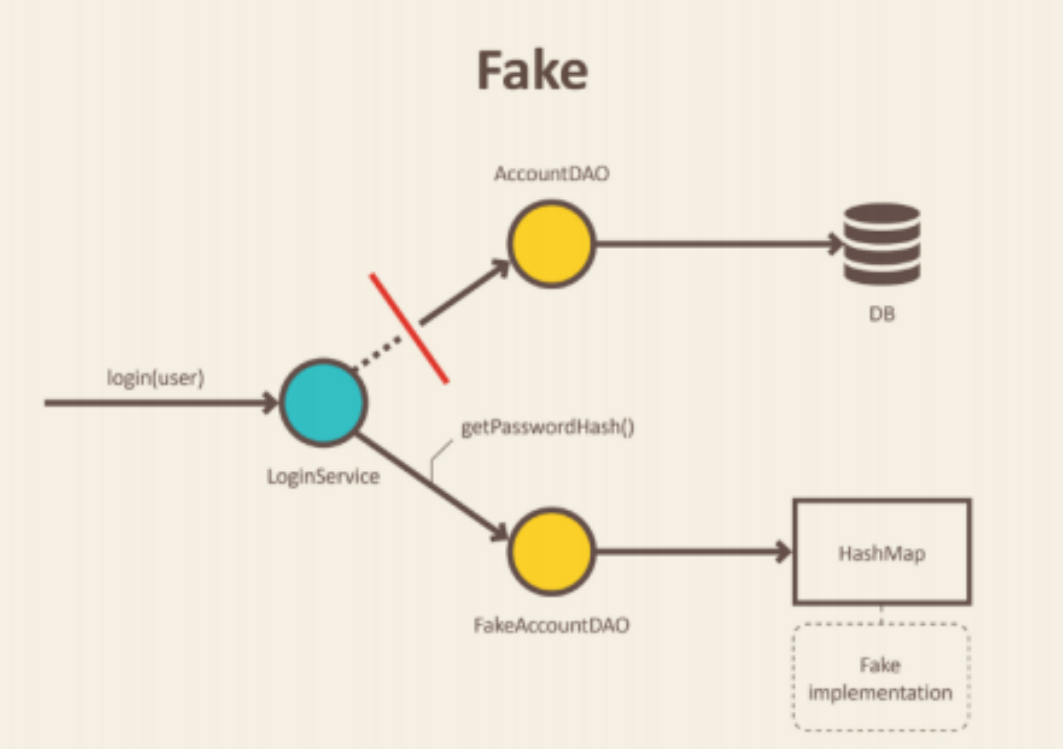

Imagen 1. Fake Diagram.
Fuente: Desafio Latam.

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.

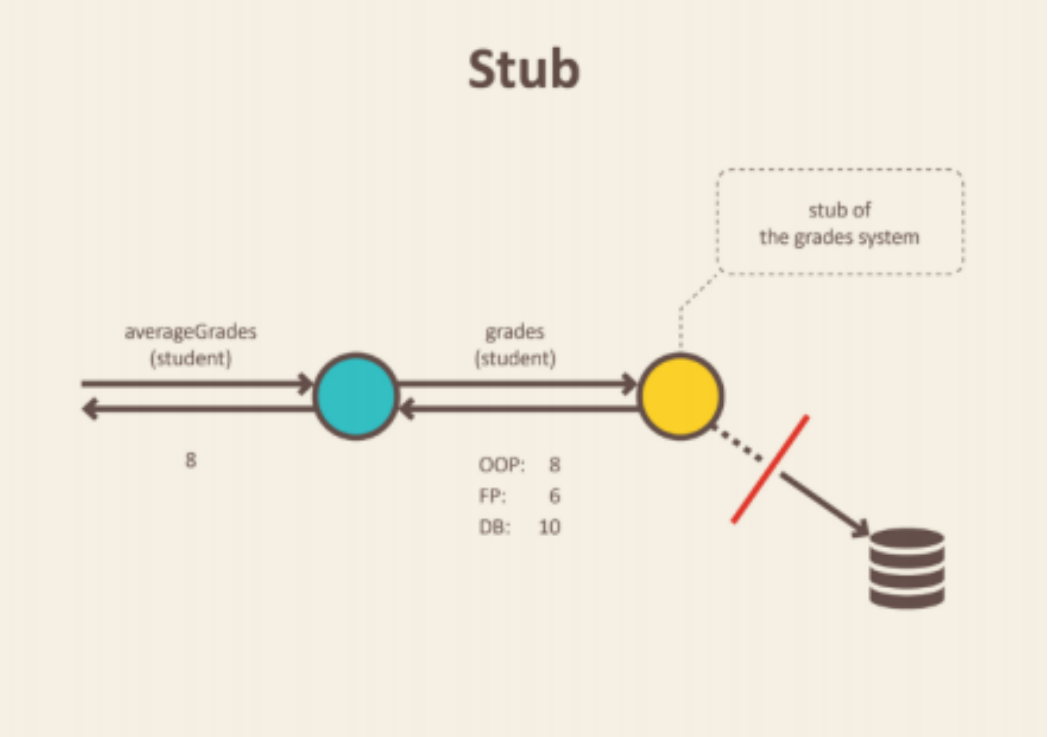

Imagen 2. Stub Diagram.
Fuente: Desafio Latam.

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.

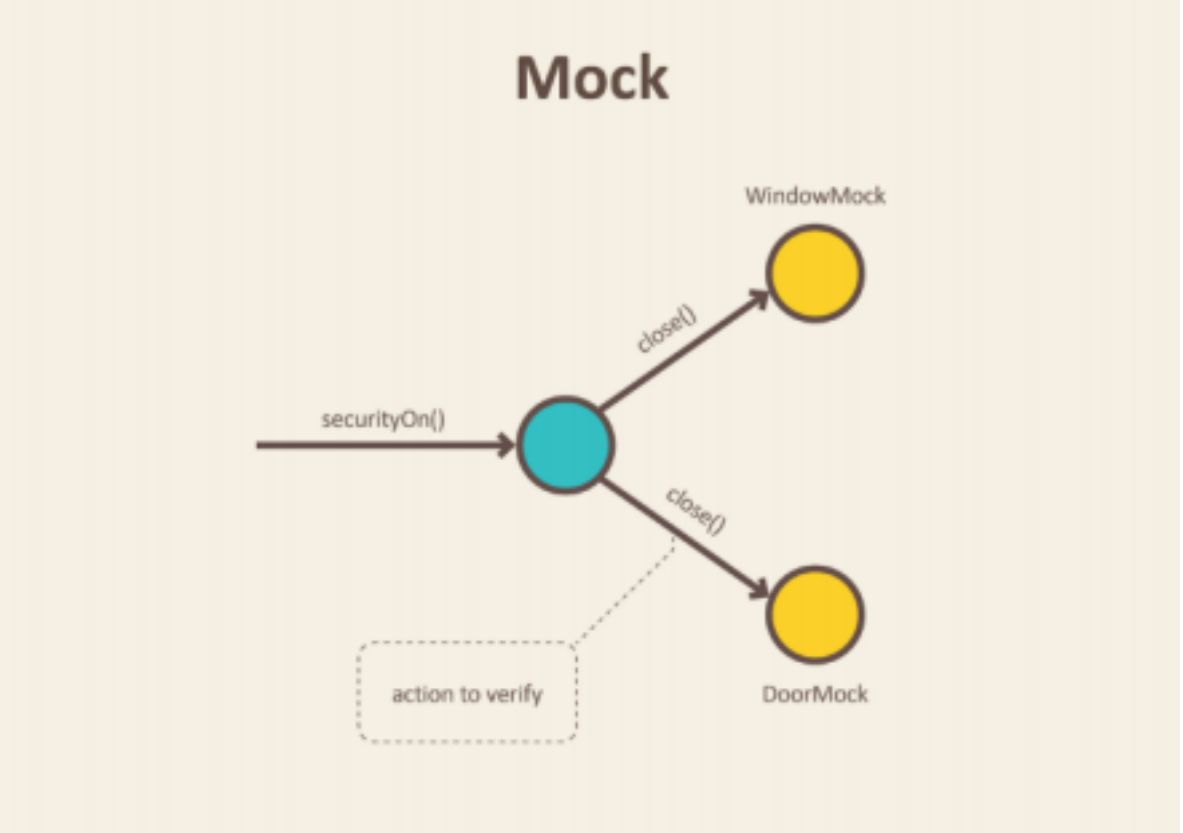

Imagen 3. Mock Diagram
Fuente: Desafio Latam

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, rodeado 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.

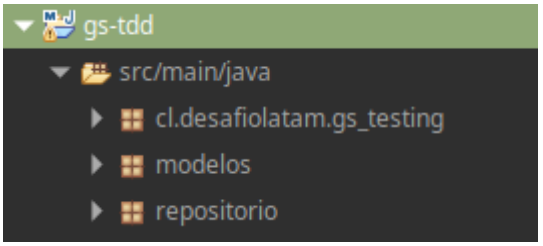

Imagen 4. Creación de package
Fuente: Desafio Latam

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.

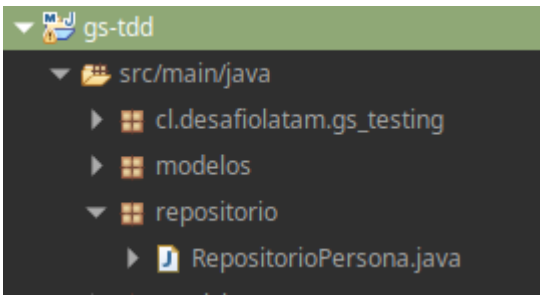

Imagen 5. Creación clase RepositorioPersona
Fuente: Desafio Latam

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:

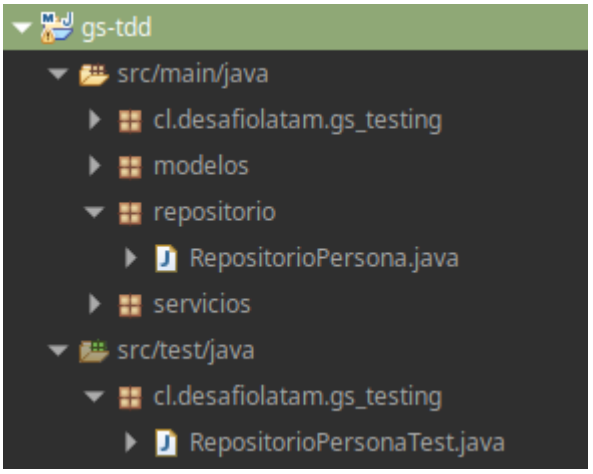

Imagen 6. Creación las clase RepositorioPersonaTest
Fuente: Desafio Latam

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;
Activar la compatibilidad con lectores de pantalla
Buscar y reemplazar

Buscar
2 de 27
Contexto:
package repositorio;
import modelos
Reemplazar por
Coincidencia de mayúsculas y minúsculas
Coincidencia con expresiones regulares Ayuda
Ignorar diacríticos latinos (por ejemplo, ä = a, E = É)
ReemplazarReemplazar todos AnteriorSiguiente

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:

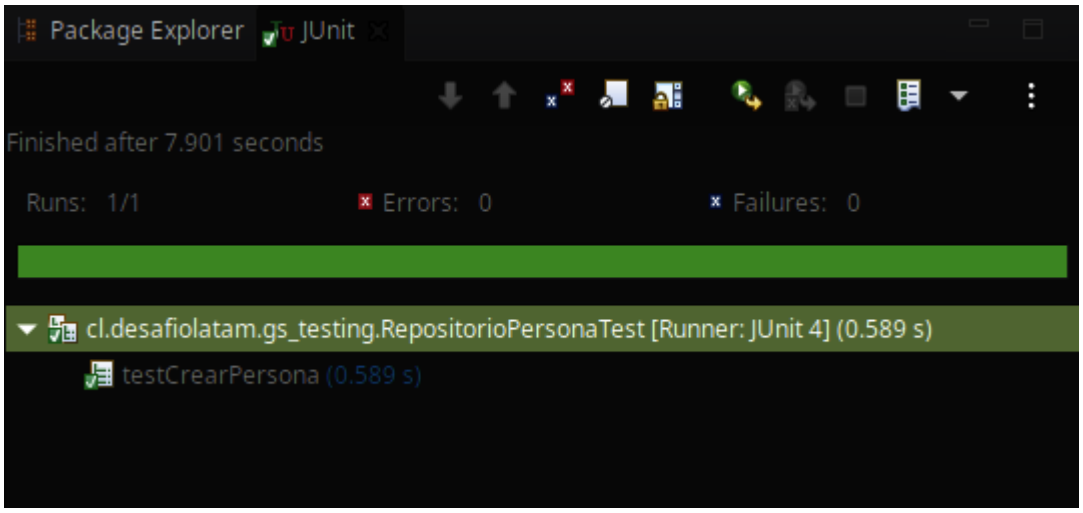

Imagen 7. Test exitoso
Fuente: Desafio Latam

```
[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);
  }
}
```

Ejercicio Propuesto (1)

En base al ejercicio guiado “Mockito” que hemos visto previamente, se pide realizar tests sobre los siguientes métodos ocupados en la clase `RepositorioPersona`.

- eliminarPersona()
- listarPersona()

Verifique que se cumpla cada requerimiento y sus respectivos parámetros.

Test Driven Development

Competencias

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

Introducción

Para los/las desarrolladores/as no es fácil dominar las TDD, inclusive si aprenden toda la teoría y trabajan con las mejores prácticas, ya que se requiere de tiempo y de mucha práctica para dominarla.

Este es un viaje largo que podría no terminar, debido a las adaptaciones aplicadas constantemente en la programación. Siempre hay nuevas formas de llegar a ser más competente y más rápido con los códigos. Sin embargo, a pesar de que el costo es alto, los beneficios son mayores. Las personas que pasan el tiempo suficiente practicando TDD son defensores y propulsores de esta práctica para desarrollar software por sus grandes beneficios.

Para aprender TDD se debe poner manos a la obra, “codear” y tener una base sólida, tanto en la teoría como en la práctica.

¿Por qué TDD?

La respuesta corta a esta pregunta es que TDD **es la forma más sencilla para lograr un código de buena calidad y de buena cobertura de prueba**.

¿Qué es exactamente el desarrollo guiado por pruebas?

TDD es un procedimiento que escribe las pruebas antes de la implementación real. Es un cambio en el enfoque tradicional.

Empezar un nuevo desarrollo usando TDD trae múltiples beneficios, muchos más que el enfoque tradicional de escribir pruebas al final, ya que el usuario debe comprender cuáles son las características de una pieza de código, cuál es su finalidad y cuál es su implementación dentro de la producción. **Con las funcionalidades claras, se entiende la problemática y se aborda de la mejor manera**.

El desarrollo de TDD en etapas avanzadas puede generar cambios significativos en el código y esto podría provocar comportamientos inesperados en la aplicación. Con las pruebas escritas previamente, si buscamos realizar cambios, el/la desarrollador/a puede encargarse de mantener las pruebas en verde (exitosas), eliminando el miedo al error. Los detalles de TDD se verán a continuación:

Reglas del Juego

1. No está permitido escribir ningún código de producción, a menos que sea para hacer una prueba fallida.
2. No está permitido escribir más de una prueba de unidad para fallar. Los fallos de compilación son exactamente eso, fallos.
3. No está permitido escribir más código de producción del que sea suficiente para pasar la prueba de la unidad.

Puede parecer que la regla 3 implica la regla 1, por lo tanto:

- Escribe solo lo suficiente de una prueba unitaria para fallar.
- Escribe solo el código de producción suficiente para hacer que la prueba unitaria que falló, pase.

Estas reglas son útiles como lista de verificación cuando se está desarrollando, por lo que simplemente repiten el orden, una y otra vez, para mantenerse en el ciclo de TDD. Son reglas simples, pero las personas que se acercan a TDD a menudo violan uno o más de ellos.

Estas reglas definen la mecánica de TDD, pero definitivamente no son todo lo que necesitas saber. De hecho, el proceso de usar TDD a menudo se describe como un ciclo rojo-verde-refactor.

Veamos de qué se trata, se basa en la repetición de un proceso bastante claro:

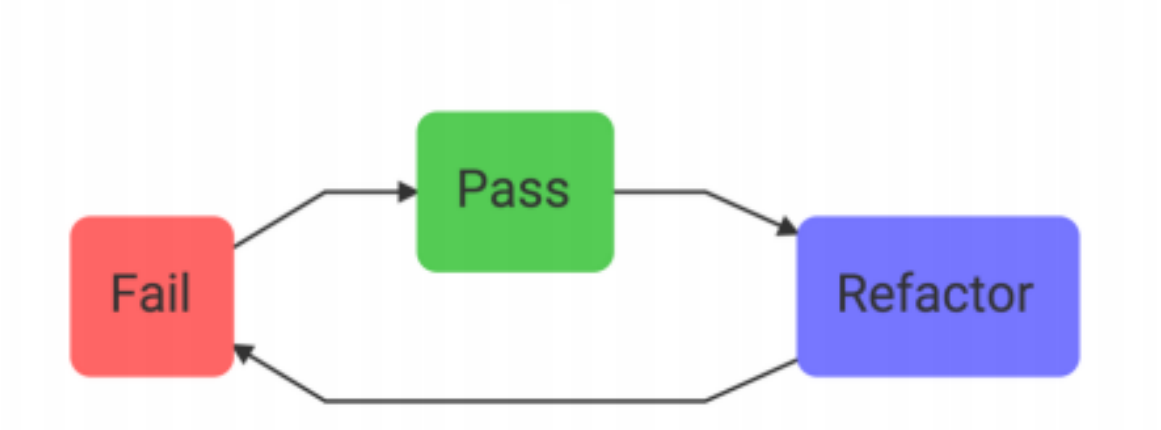

Imagen 8. Diagrama de TDD.
Fuente: Desafio Latam.

Este ciclo de desarrollo se basa en el primer concepto de prueba de la programación externa, donde se fomenta el diseño simple con un alto nivel de confianza. El procedimiento consiste en pocos pasos que se repiten una y otra vez, y otra vez.

La técnica de refactorización rojo-verde es la base de TDD. Es un juego de ping pong en el que estamos cambiando las pruebas y el código de implementación a gran velocidad. En donde se debe fallar, luego tener éxito y, finalmente, mejorar.

Fase Roja

En esta fase se debe concentrar en escribir una interfaz limpia para futuros usuarios. Aquí diseñas la forma en que los clientes utilizarán tu código. Se debe escribir una prueba sobre un comportamiento que está a punto de implementarse.

Para hacerlo se escribe un fragmento de código como si ya estuviera implementado. Sin embargo, la implementación no existe, por lo que es necesario realizarla. Por lo tanto, si en esta fase se está pensando en cómo implementar el código o cómo se va a escribir el código de producción, se está haciendo mal, ya que se debe escribir una prueba para que luego se pueda escribir el código de producción.

Es importante destacar que aquí no se escribe una prueba para probar el código implementado, si no que se comete un error a propósito para que cuando se escriban los métodos adicionales del programa, no se deba escribir un montón de métodos o clases que "quizás" se necesiten más adelante. Lo importante es concentrarse en la función que se está escribiendo y en lo que realmente se necesita.

En esta etapa se deben tomar decisiones sobre cómo se utilizará el código, basándose en lo realmente necesario y no en lo que se cree que pueda ser necesario. Escribir algo que la característica aún no requiere es ingeniería excesiva. Entonces, ¿qué pasa con la abstracción del código? En la fase de refactorización se abordan todas las mejoras y buenas prácticas.

Fase Verde

Esta puede ser la fase más sencilla del ciclo, ya que se escribe el código de producción. Si eres un programador lo haces todo el tiempo. Y en este punto es posible otro gran error: en lugar de escribir suficiente código para pasar la prueba unitaria fallida, se escriben más algoritmos y métodos y mientras haces esto probablemente estés pensando en cuál es la mejor implementación con el mejor rendimiento, pero de ninguna manera se debe pensar en la mejor implementación, mucho menos escribir código extra.

¿Por qué no se puede escribir todo el código que se tiene en mente?

Por dos razones:

1. Una tarea sencilla es menos propensa a errores, y en esta fase se minimizan.
2. No se debe mezclar el código que se está probando con el código que no está escrito.

Y en este punto surgen algunas preguntas: ¿qué pasa con el código limpio?, ¿qué pasa con el rendimiento?, ¿qué pasa si escribir código me hace descubrir un problema?, ¿qué pasa con las dudas? La optimización del rendimiento en esta fase es una optimización prematura, ya que en este punto debes actuar como un/a desarrollador/a que tiene una tarea simple, escribir una solución sencilla que convierte la prueba fallida en una prueba exitosa para que el rojo alarmante en el detalle de la prueba se convierta en un verde aprobado.

Además, se le permite a el/la desarrollador/a romper las buenas prácticas e incluso duplicar el código, considerando que la fase de refactorización se debe utilizar para limpiar el código.

Fase Refactorización

En la fase de refactorización se debe modificar el código siempre y cuando se mantengan todas las pruebas en verde para que sea mejor. Lo que es “mejor” depende de cada desarrollador, pero existe una tarea obligatoria: **se debe eliminar el código duplicado**.

Kent Becks sugiere en su libro que eliminar todo código duplicado es todo lo que necesitas hacer. Ahora juegan su rol los desarrolladores exigentes que refactorizan el código para llevarlo a un buen nivel. En la fase roja se expresa claramente la intención de las pruebas para los usuarios (código de producción), pero en la fase refactorización se demuestran las habilidades del desarrollador a los demás desarrolladores que leerán el código.

Ejercicios guiado: Fases

Se tiene el caso donde se quiere controlar una liga de fútbol femenino, y una de las partes del programa necesita comparar dos equipos para ver quién va primero. Si se quiere comparar cada equipo, se debe recordar la cantidad de partidos que ha ganado y el mecanismo de comparación usará estos datos.

Entonces, una clase de `EquipoFutbol` necesita un campo en el que se pueda guardar la información y debería ser accesible de alguna manera. Se necesitan pruebas en la comparación para ver que los equipos con más victorias ocupen el primer lugar, y a la vez comprobar qué sucede cuando dos equipos tienen el mismo número de victorias.

Pruebas - Fase Roja

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

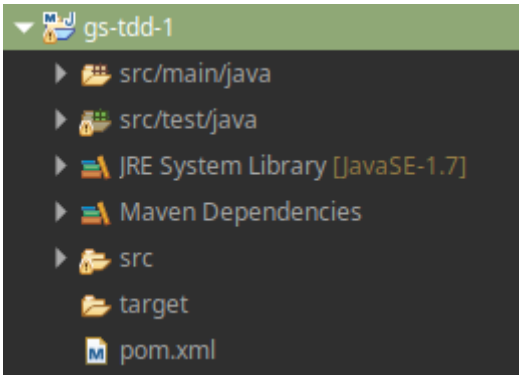

Imagen 9. Estructura de proyecto “gs-tdd-1”.
Fuente: Desafio Latam.

Paso 2: Agregar las dependencias en el archivo **pom.xml**:

```
<dependencies>
   <dependency>
     <groupId>org.junit.jupiter</groupId>
     <artifactId>junit-jupiter</artifactId>
     <version>5.4.2</version>
     <scope>test</scope>
</dependency>
   <dependency>
     <groupId>org.mockito</groupId>
     <artifactId>mockito-core</artifactId>
     <version>2.28.2</version>
     <scope>test</scope>
  </dependency>
</dependencies>
```

Paso 3: Para comparar dos equipos, cada uno de ellos debe recordar su número de victorias, y para mantener la simplicidad se va a permitir diseñar una clase `EquipoFutbol` que toma el número de partidos como un parámetro del constructor. Lo primero es escribir la prueba y hacer que falle, esta clase de prueba llamada `EquipoFutbolTest` debe estar alojada en:

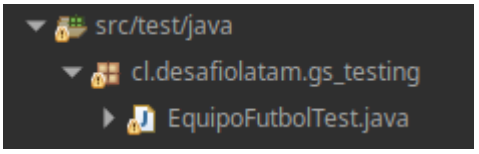

Imagen 10. Carpeta test proyecto “gs-tdd-1”.
Fuente: Desafio Latam.

Paso 4: Para asegurar que el constructor funcione, la clase `EquipoFutbolTest` debe contener una prueba que llama a la clase `EquipoFutbol` y revisar si el constructor recibe el número de partidos ganados.

```
package cl.desafiolatam;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertEquals;

  public class EquipoFutbolTest {
  @Test
  public void constructorDebeSetearJuegosGanados() {
  EquipoFutbol team = new EquipoFutbol(3);
  assertEquals(3, team.getJuegosGanados());
 }
}
```

EquipoFutbol no existe, por lo tanto, si estás usando un IDE debe resaltar ese error; ocurrirá lo mismo con el método `getJuegosGanados` . Se debe escribir la clase `EquipoFutbol` como su método `getJuegosGanados`. Siempre y cuando se escriba acorde a las reglas.

Paso 5: Crear la clase `EquipoFutbol` y su respectivo método, pero sin agregar lógica de negocios. La estructura de directorios queda así:

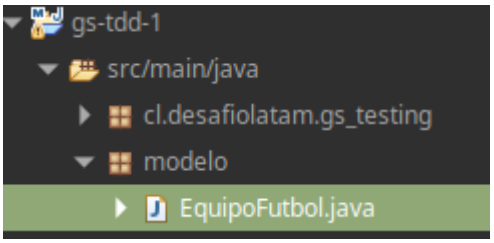

Imagen 11. Creación clase EquipoFutbol.
Fuente: Desafio Latam.

Paso 6: La clase `EquipoFutbol` solo contendrá lo necesario para que la prueba compile.

```
package cl.desafiolatam;

public class EquipoFutbol {
  public EquipoFutbol(int juegosGanados) {
 }
  public int getJuegosGanados() {
  return 0;
 }
}
```

Paso 7: Ejecutamos el `Maven Test` para que la prueba falle:

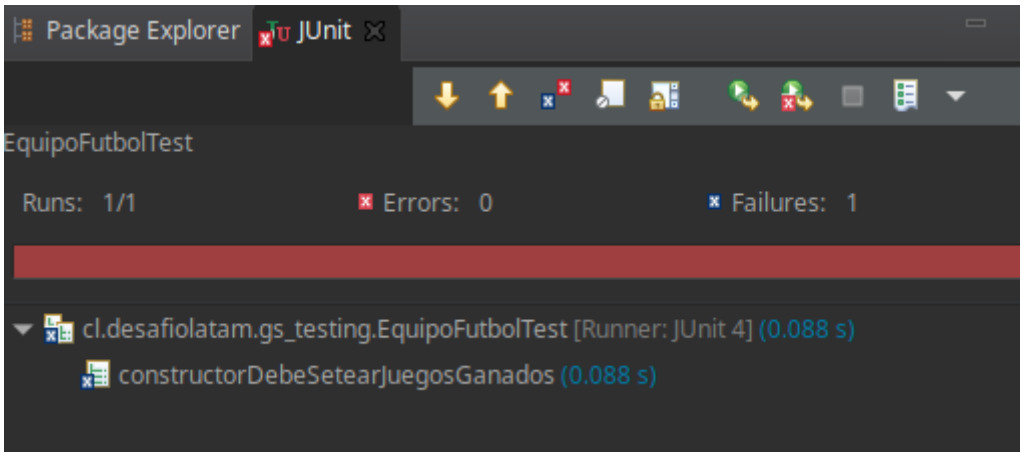

Imagen 12. Falla a propósito del test.
Fuente: Desafio Latam.

```
[INFO] ------------------------------------------------------- [INFO] T
E S T S
[INFO] ------------------------------------------------------- [INFO]
Running EquipoFutbolTest
[ERROR] Tests run: 1, Failures: 1, Errors: 0, Skipped: 0, Time elapsed:
0.023 s <<< FAILURE! - in EquipoFutbolTest
[ERROR] constructorDebeSetearJuegosGanados Time elapsed: 0.004 s <<<
FAILURE!
org.opentest4j.AssertionFailedError: expected: <3> but was: <0> at
constructorDebeSetearJuegosGanados(EquipoFutbolTest.java:15)
[INFO]
[INFO] Results:
```

```
[INFO]
[ERROR] Failures:
[ERROR] EquipoFutbolTest.constructorDebeSetearJuegosGanados:15 expected:
<3> but was: <0>
[INFO]
[ERROR] Tests run: 1, Failures: 1, Errors: 0, Skipped: 0
[INFO]
[INFO]
------------------------------------------------------------------------
-
[INFO] BUILD FAILURE
[INFO]
------------------------------------------------------------------------
-
```

Escribir el código de producción donde irá la lógica de negocios es más sencilla una vez que las pruebas están listas, podríamos decir que escribir la prueba fue más exigente. La prueba fallida en este punto es algo bueno ya que está escrita para verificar cómo se comporta la clase bajo prueba. En el caso de que alguna vez rompamos nuestra clase bajo prueba, la prueba fallará y el mensaje de error dirá exactamente qué es lo que falla y, por lo tanto, se podrá arreglar con facilidad. En este caso la salida de la prueba:

```
EquipoFutbolTest.constructorDebeSetearJuegosGanados:15 expected: <3> but
was: <0>

//En donde se detalla que la prueba llamada
constructorDebeSetearJuegosGanados espera como resultado 3, pero fue 0.
```

Implementación - Fase Verde

Paso 8: La fase verde es sencilla esta vez: Almacenar el valor pasado como parámetro del constructor a alguna variable interna. De tal forma que el método getJuegosGanados entregue el valor que se pasó como parámetro del constructor. A continuación la clase `EquipoFutbol`.

```
package cl.desafiolatam;

 public class EquipoFutbol {
  private int juegosGanados;

 public EquipoFutbol(int juegosGanados) {
  this.juegosGanados = juegosGanados;
 }
 public int getJuegosGanados() {
 return juegosGanados;
 }
}
```

Paso 9: La prueba debe pasar ahora. Sin embargo, todavía queda algo que hacer. Este es el momento de pulir el código, refactorizar. No importa cuán pequeños sean los cambios que hayas realizado, vuelve a ejecutar la prueba para asegurar que nada se ha roto accidentalmente.

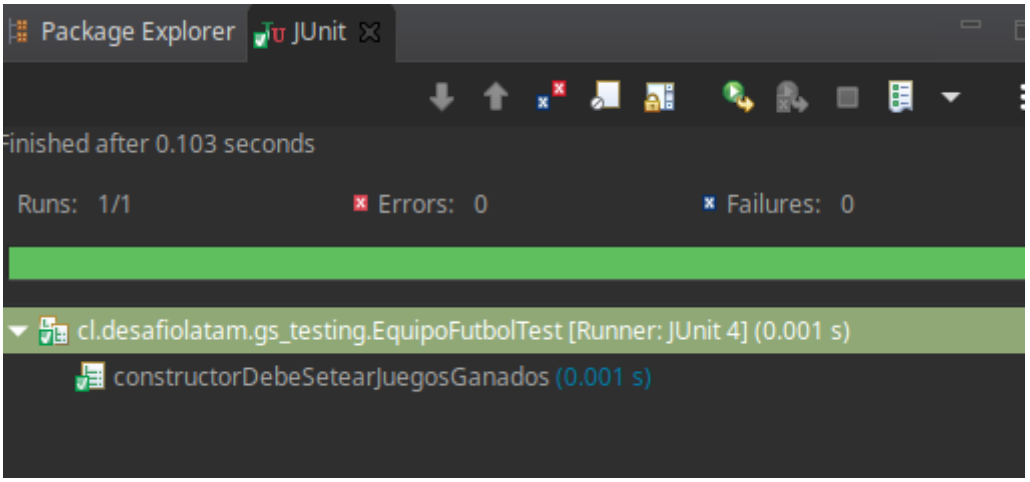

Imagen 13. Test que pasó con éxito
Fuente: Desafio Latam

```
[INFO] -------------------------------------------------------
[INFO] T E S T S
[INFO] -------------------------------------------------------
[INFO] Running EquipoFutbolTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
0.018 s - in EquipoFutbolTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO]
------------------------------------------------------------------------
-
[INFO] BUILD SUCCESS
[INFO]
------------------------------------------------------------------------
-
[INFO] Total time: 2.941 s
[INFO] Finished at: 2019-07-08T17:56:27-04:00
[INFO]
------------------------------------------------------------------------
-
```

Refactorización

Paso 10: En el caso de este ejemplo, algo simple como la clase de EquipoFutbol, no tiene mucho que refactorizar. Sin embargo, la refactorización de la prueba también se debe considerar. La refactorización será deshacerse del número 3 como parámetro de assertEquals, usando una variable CUATRO_JUEGOS_GANADOS.

```
package cl.desafiolatam;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertEquals;

  public class EquipoFutbolTest {
   private static final int CUATRO_JUEGOS_GANADOS = 4;

  @Test
  public void constructorDebeSetearJuegosGanados() {
  EquipoFutbol team = new EquipoFutbol(CUATRO_JUEGOS_GANADOS);
  assertEquals(CUATRO_JUEGOS_GANADOS,team.getJuegosGanados());
 }
}
```

Paso 11: La salida de mvn test sigue siendo exitosa:

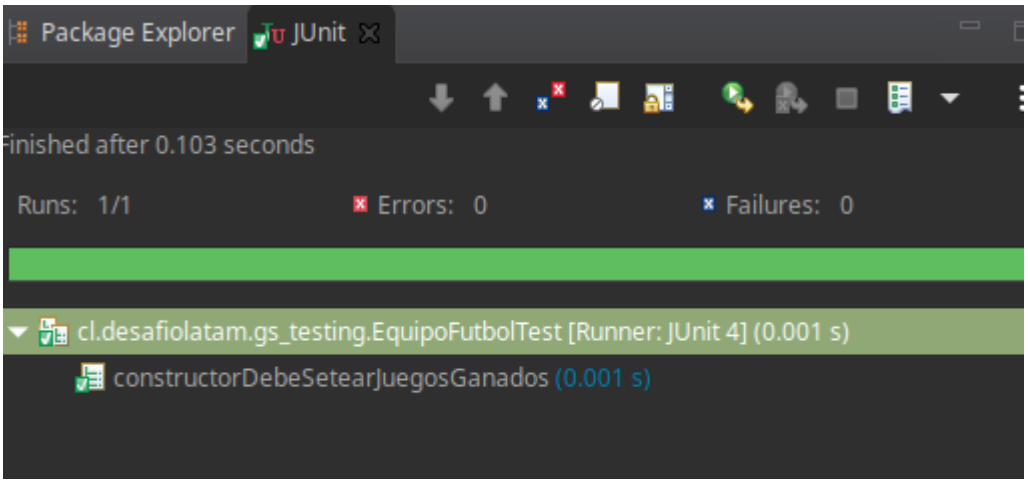

Imagen 15. Test refactorizado que pasó con éxito
Fuente: Desafio Latam

```
[INFO] -------------------------------------------------------
[INFO] T E S T S
[INFO] -------------------------------------------------------
[INFO] Running EquipoFutbolTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
0.027 s - in EquipoFutbolTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO]
------------------------------------------------------------------------
-
[INFO] BUILD SUCCESS
[INFO]
------------------------------------------------------------------------
-
[INFO] Total time: 2.863 s
[INFO] Finished at: 2019-07-08T18:04:42-04:00
[INFO]
------------------------------------------------------------------------
-
```

Acabas de terminar tu primer ciclo de TDD, pasando por fase roja, donde falla la prueba, luego la fase verde, donde se escribe solo el código necesario para pasar la prueba, y finalmente se refactoriza para dejar el código lo mejor posible. ¿Qué es mejor? Eso depende de ti, de el/la desarrollador/a.

Consideraciones de TDD

¿Podemos decir que TDD requiere más tiempo que la programación normal?

Lo que toma tiempo es aprender y dominar TDD, así como configurar y usar un entorno de prueba. Cuando se está familiarizado con las herramientas de prueba y la técnica TDD en realidad no se requiere de más tiempo. Por el contrario, mantiene un proyecto lo más simple posible y, por lo tanto, ahorra tiempo.

¿Cuántas pruebas se deben escribir?

La cantidad mínima que le permita escribir todo el código de producción. La cantidad mínima porque cada prueba demora la refactorización (cuando cambia el código de producción, debe corregir todas las pruebas que fallan). Por otro lado, la refactorización es mucho más simple y segura en el código bajo pruebas.

Con TDD no se necesita dedicar tiempo al análisis

Falso. Si lo que vas a implementar no está bien diseñado, te encontrarás con casos que no consideraste. Y esto significa que tendrá que eliminar la prueba y el código de esta prueba.

¿La cobertura de pruebas debe ser del 100%?

Se puede evitar el uso de TDD en algunas partes del proyecto. Por ejemplo, en las vistas porque son las que pueden cambiar a menudo.

Se puede escribir código con pocos errores que no necesitan pruebas

Puede ser verdadero, pero ¿todos los miembros del equipo comparten esto? Los demás miembros modificarán el código y es probable que se rompa. En este caso aplica tener pruebas unitarias para detectar un error de inmediato y no en producción.

Ejercicio Propuesto (2)

En base al ejercicio creado previamente, ahora debemos implementar 2 nuevos test que ayuden a ver los otros juegos que ha tenido el equipo de fútbol femenino. Para esto se deben cubrir lo siguientes requisitos:

- Crear 2 nuevas variables llamadas: juegosPerdidos y juegosEmpatados.
- Generar el getter correspondiente a cada variable y el constructor.
- Luego, en la clase que habíamos ocupado de los test, crearemos 2 nuevos parámetros: CINCO_JUEGOS_EMPATADOS = 5 y TRES_JUEGOS_PERDIDOS = 3
- Finalmente, haremos el test correspondiente para verificar que se cumpla la igualdad.
-

Solución Ejercicio Propuesto (1)

Paso 1: Para la prueba del método `testEliminarPersona`, se pasan parámetros similares llamando al método `eliminarPersona` 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 testEliminarPersona() {
   Persona sam = new Persona("1-4", "Sam");
   when(repositorioPersona.eliminarPersona(sam)).thenReturn("OK");
String eliminarRes = repositorioPersona.eliminarPersona(sam);
   assertEquals("OK", eliminarRes);
   verify(repositorioPersona).eliminarPersona(sam);
  }
}
```

Paso 2: Finalmente, para método de prueba `testListarPersona` se establece un mapa llamado `mockRespuesta`, el cual es un `HashMap<String, String>` que será seteado como el valor que será retornado por parte del repositorio. Con esto tenemos la respuesta esperada y se puede hacer el flujo de llamar al método del repositorio. Se comprueba si el método ocurrió una vez usando `verify`.

```
package repositorio;
import modelos.Persona;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
import java.util.HashMap;
import java.util.Map;
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 testListarPersona() {
  Map<String, String> mockRespuesta = new HashMap<>();
  when(repositorioPersona.listarPersonas()).thenReturn(mockRespuesta);
Map<String,String> listarRes = repositorioPersona.listarPersonas();
assertEquals(mockRespuesta, listarRes);
  verify(repositorioPersona).listarPersonas();
 }
}
```

Solución Ejercicio Propuesto (2)

Paso 1: Crear la clase EquipoFutbol.

```
package modelo;

public class EquipoFutbol {
     private int juegosGanados;
     private int juegosPerdidos;
     private int juegosEmpatados;

    public EquipoFutbol(int juegosGanados, int juegosPerdidos, int
juegosEmpatados) {
          this.juegosGanados = juegosGanados;
          this.juegosPerdidos = juegosPerdidos;
          this.juegosEmpatados = juegosEmpatados;
     }
      public int getJuegosGanados() {
      return juegosGanados;
     }
    public int getJuegosPerdidos() {
       return juegosPerdidos;
    }
    public int getJuegosEmpatados() {
       return juegosEmpatados;
    }

}
```

Paso 2: Crear los test en la clase EquipoFutbolTest.

```
public class EquipoFutbolTest {
     private static final int CUATRO_JUEGOS_GANADOS = 4;
     private static final int CINCO_JUEGOS_EMPATADOS = 5;
     private static final int TRES_JUEGOS_PERDIDOS = 3;

     @Test
      public void constructorDebeSetearJuegosGanados() {
        EquipoFutbol team = new EquipoFutbol(CUATRO_JUEGOS_GANADOS,
TRES_JUEGOS_PERDIDOS, CINCO_JUEGOS_EMPATADOS);
         assertEquals(CUATRO_JUEGOS_GANADOS,team.getJuegosGanados());
         assertEquals(TRES_JUEGOS_PERDIDOS,team.getJuegosPerdidos());
         assertEquals(CINCO_JUEGOS_EMPATADOS,team.getJuegosEmpatados());
     }
}
```