Desarrollo en Java basado en pruebas (Parte II)

Test Dobles

Competencias

- Conocer Mocks, Stubs y Fakes.
- Diferenciar dobles de test.
- Identificar cuando usar cada tipo de test .
- Crear Mocks utilizando Mockito.
- Uso de métodos estáticos de Mockito para simular métodos y setear comportamiento.

Motivación

Imaginemos que se necesita probar una parte del programa que interactúa con una pasarela de pagos, el sistema debe comunicarse con ese servicio y necesitamos probarlo. Usar la pasarela de pagos con datos ficticios cada vez que se ejecute una prueba puede ser lento, si se produce algún error existirá la duda sobre si falló la pasarela o tu código. Además es probable que la plataforma contra la que poder ejecutar las pruebas no esté disponible. Es por esto que resulta conveniente ejecutar en aislamiento el sistema bajo prueba.

¿Qué son los dobles de prueba?

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?

En ambiente de pruebas se simulan componentes para no utilizar los que funcionan en un ambiente de producción. Esta propuesta que parece sencilla de explicar, puede resultar bastante complicada en el escenario expuesto cuando se hace uso de servicios de terceros, por lo tanto se necesita de un mecanismo que permita contar con dobles o impostores de estos servicios que no se deben estar llamando durante las pruebas. Aqui entran los dobles de test para facilitar la simulación de estos componentes o servicios.

- Dummy: Son dobles de prueba que se pasan allí donde son necesarios. Son simplemente relleno.
- Fake: Son implementaciones de componentes, ejemplo una base de datos en memoria.
- Stubs: Conjunto de respuestas enlatadas que se ofrecerán como resultado de una serie de llamadas a nuestro doble de prueba.
- Mocks: Dobles de prueba que son capaces de analizar como se relacionan los distintos componentes. Nos ayudan a testear el paso de mensajes entre objetos.

Cuando se programa, el objetivo es que el código sea lo más expresivo y mantenible posible. Una forma de conseguirlo es aplicando buenas prácticas (Clean Code), para que sea expresivo y fácil de leer. Con este sentido en vez de llamar a todos los objetos simulados Mocks, se revisarán las diferencias entre cada uno de los dobles de pruebas.

Según la clasificación de Fowler, se pueden tener a varios tipos de dobles de prueba:

Clasificación dobles de test

Dummy

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

Fake

Son implementaciones de componentes de la aplicación que funcionan y son operativas, pero que sólo implementan lo mínimo necesario de características para poder pasar las pruebas. No son adecuados para ser desplegados en producción. Simplifican la versión del código de producción.

Un ejemplo de este, puede ser una implementación en memoria de un repositorio. Esto permite realizar pruebas de integración de servicios sin iniciar una base de datos y realizar solicitudes que consumen mucho tiempo. Además de las pruebas, la implementación falsa puede ser útil para la creación de prototipos. Podemos implementar y ejecutar nuestro sistema rápidamente con la base de datos en memoria, aplazando las decisiones sobre el implementación de la base de datos.

> Diagrama de Fake

En este diagrama mostrado en la Imagen 1 está el ejemplo de un acceso directo, que puede ser una implementación en memoria del Objeto de acceso a datos o Repositorio.

Esta implementación falsa no comprometerá la base de datos, pero usará una colección simple para almacenar datos (HashMap). Esto permite realizar pruebas de integración de servicios sin iniciar una base de datos.

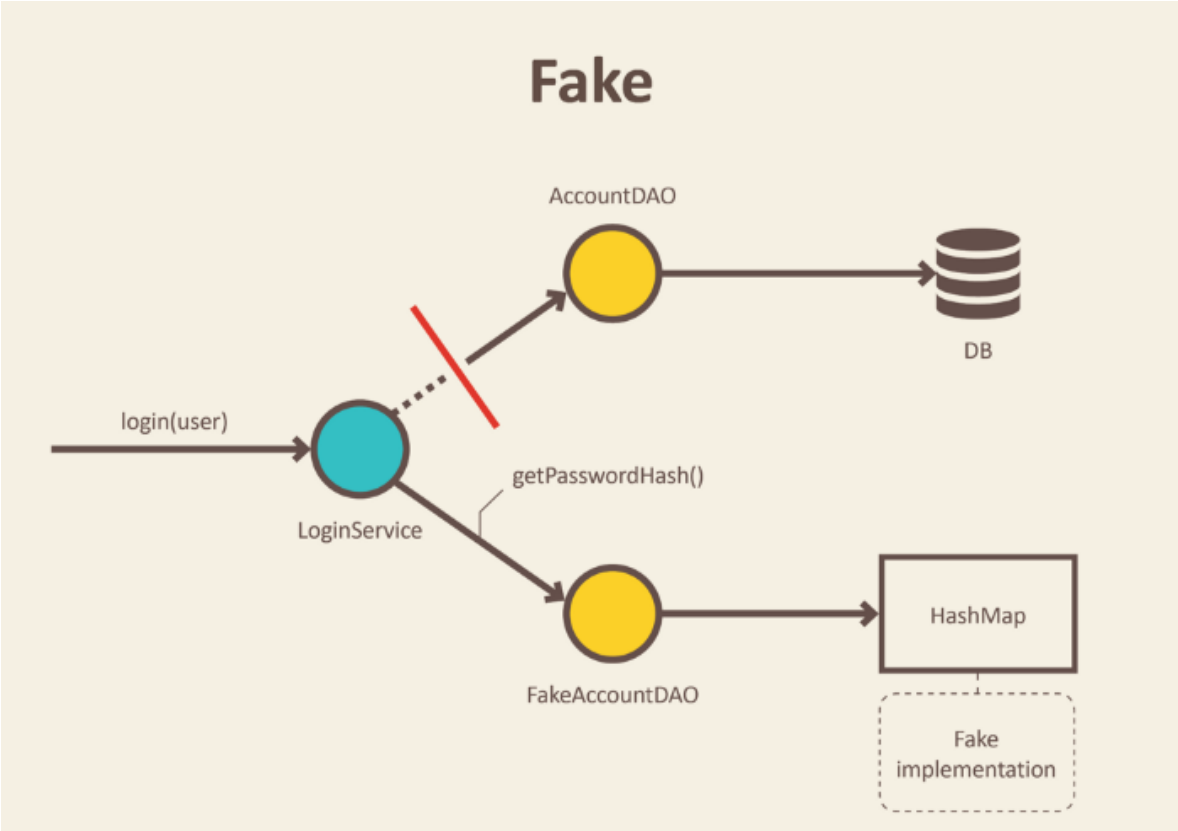

Imagen 1: Fake Diagram.

Stub

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. Serían 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.

> Diagrama Stub

En este diagrama (ver Imagen 2) está el ejemplo de un objeto que necesita tomar algunos datos de la base de datos 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.

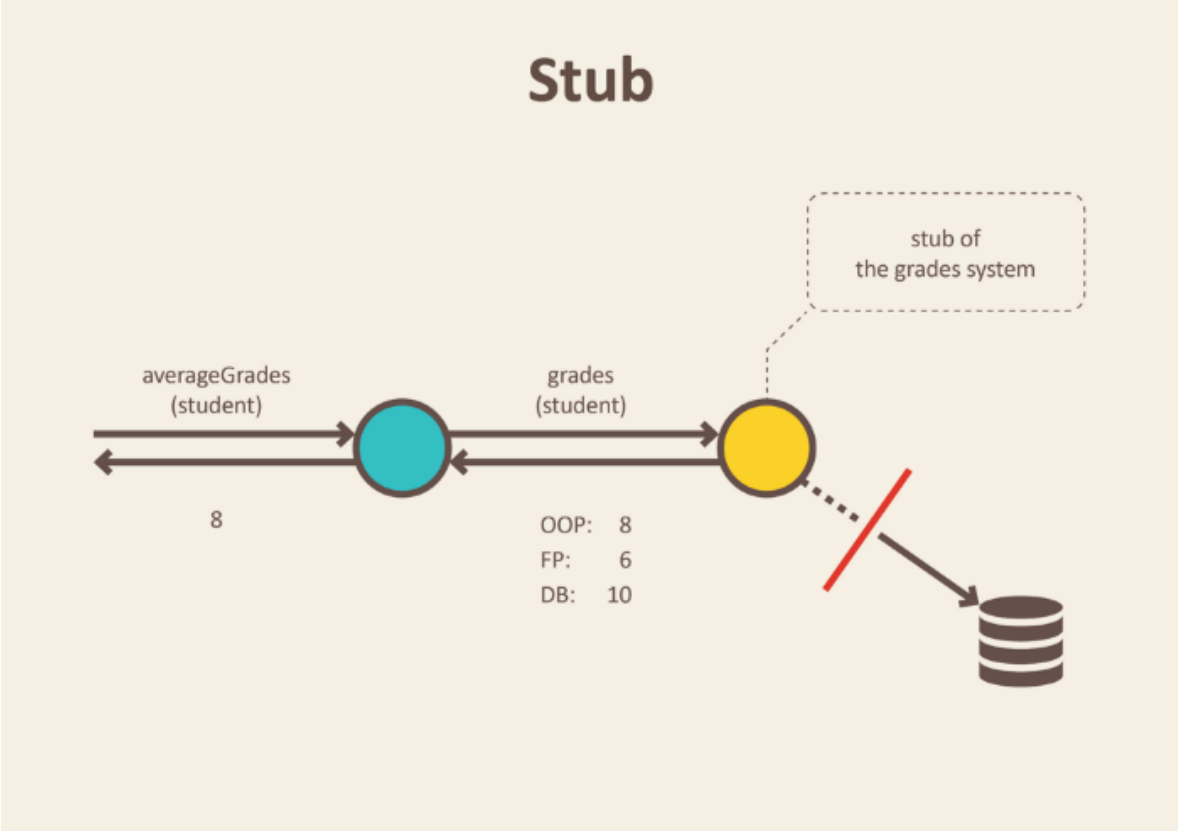

Imagen 2: Stub Diagram .

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 usan Mocks 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 que persiste en interactúa con una base de datos. Lo que se busca es verificar los resultados de la funcionalidad que se ejerce en la prueba. Verificando que se haya llamado al servicio bajo prueba. Son capaces de analizar como se relacionan los distintos componentes, permitiendo verificar si un método concreto ha sido invocado o no, qué parámetros ha recibido o cuantas veces lo hemos ejercitado.

Aunque también pueden devolver una respuesta con un estado determinado, su foco se centra más en el análisis del comportamiento. Nos ayudan a probar la comunicación entre objetos. Las pruebas deben ser expresivas y transmitir la intención de forma clara y concisa. A la hora de crear pruebas que no dependan de otros servicios o bases de datos externa, los dobles de prueba pueden ser herramientas muy útiles. Su uso para la gestión del estado con los Stubs, como del comportamiento con los Mocks, son una herramienta más que no se debe olvidar en la caja de herramientas de desarrollador.

> Diagrama Mock

En el siguiente diagrama 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. (ver Imagen 3)

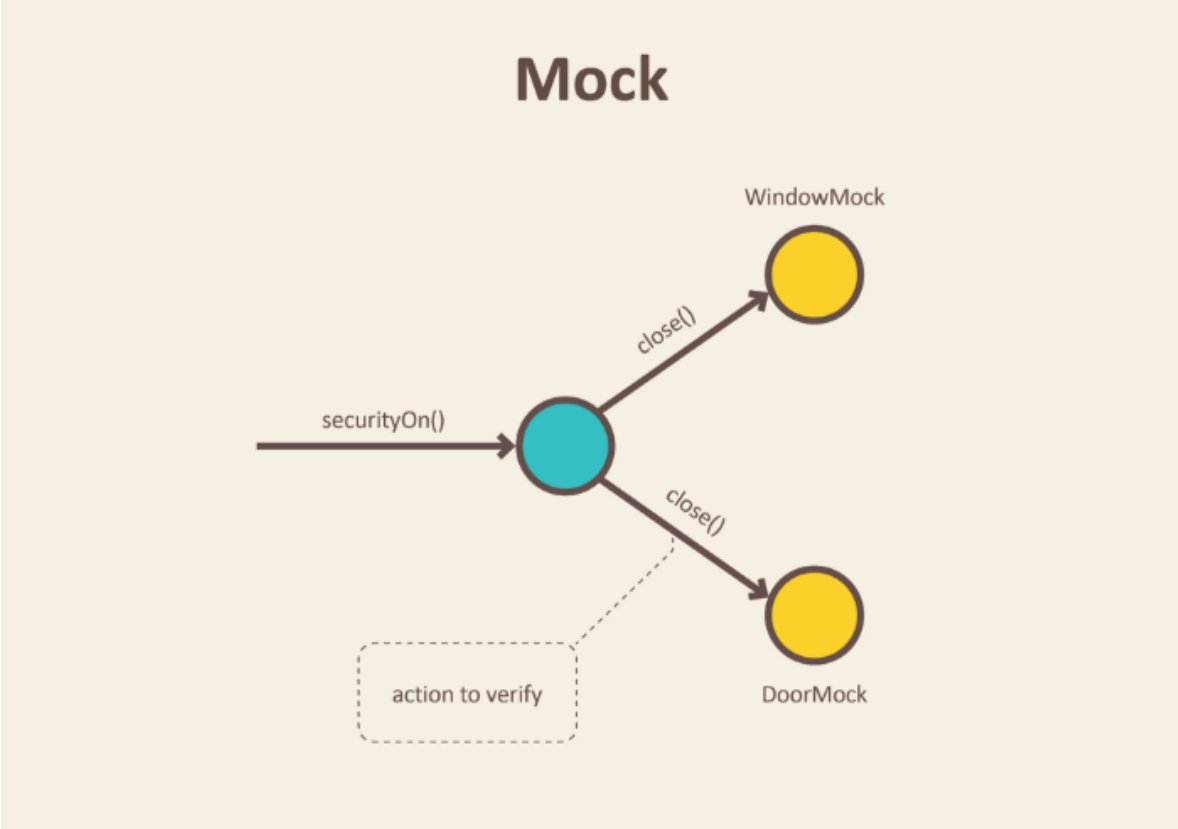

Imagen 3: Mock Diagram .

Mockito

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

Añadir dependencia

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

Para trabajar con Maven se debe 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>
```

Escribir código a probar

Se crea la clase RepositorioPersona dentro de la carpeta src/main, para que persista la información, con el objetivo que simule la interacción con una base de datos, se crea en la ruta:

`src.main.java.cl.desafiolatam.repositorios.RepositorioPersona.java`

Y los directorios quedarían así:

```
gs-testing
├── pom.xml
└── src
    ├── main
    │   └── java
    │       └── cl
    │           └── desafiolatam
    │               ├── modelos
    │               │   └── Persona.java
    │               ├── repositorios
    │               │   └── RepositorioPersona.java
    │               └── servicios
    │                   └── ServicioPersona.java
    │
    └── test
        └── java
```

```
            └── cl
                └── desafiolatam
                    └── servicios
                        └── ServicioPersonaTest.java
```

RepositorioPesona contiene los métodos crear, actualizar, listar y eliminar una persona, se importa el modelo persona desde la carpeta modelos. Esta clase será utilizada por otros servicios dentro del sistema, el código del repositorio es el siguiente.

```java
package cl.desafiolatam.repositorios;

import cl.desafiolatam.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";
    }

}
```

Escribir pruebas

Crear la clase RepositorioPersonaTest dentro de la carpeta src/test del proyecto, en la ruta

`src.test.java.cl.desafiolatam.repositorios.RepositorioPersonaTest.java`

Y los directorios quedarían así:

```
gs-testing
├── pom.xml
└── src
    ├── main
    │   └── java
    │       └── cl
    │           └── desafiolatam
    │               ├── modelos
    │               │   └── Persona.java
    │               ├── repositorios
    │               │   └── RepositorioPersona.java
    │               └── servicios
    │                   └── ServicioPersona.java
    └── test
        └── java
            └── cl
                └── desafiolatam
                    ├── repositorios
                    │   └── RepositorioPersonaTest.java
                    └── servicios
                        └── ServicioPersonaTest.java
```

Al escribir las pruebas unitarias, es probable que aparezca el desafío de que la unidad bajo prueba depende de otros componentes. Y la configuración de otros componentes realizar la prueba unitaria es tiempo que muchas veces no se tiene y excede el alcance del desarrollo. En lugar de eso, se pueden utilizar Mocks en lugar de estos componentes y continuar con la prueba de la unidad.
Pensemos en que el servicio contiene la lógica del negocio, verificaciones y distintos flujos en base a los datos entrantes, cuando su flujo continua de forma normal este envía los datos ya procesados hacia el repositorio, el cual los persiste en una base de datos. Sin embargo en un ambiente de prueba, no se puede apuntar al repositorio para guardar los datos, con cada prueba se haría trabajar a la base de datos con lecturas o escrituras. Esto sería costoso, 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 a como si el flujo fuese normal.

Se crea el objeto simulado de RepositorioPersona con el método estático `mock`. El cual crea un mock dada una clase o una interfaz dada.

```
package cl.desafiolatam.repositorios;

import static org.mockito.Mockito.mock;

public class RepositorioPersonaTest {

    private RepositorioPersona repositorioPersona =
mock(RepositorioPersona.class);

}
```

Uso mock de Mockito

Dentro de RepositorioPersonaTest se crea el método testCrearPersona tiene sus anotaciones, `@Test` y `@DisplayName`. Dentro del método se crea un objeto llamado pepe de tipo Persona.

```
package cl.desafiolatam.repositorios;

import cl.desafiolatam.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
    @DisplayName("given crearPersona mocked method when crearPersona invoked
then mocked value returned")
    public void testCrearPersona() {
        Persona pepe = new Persona("1-2", "Pepe");
    }

}
```

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 cuando se llame a un método particular. Simplemente se coloca: "Cuando se llama al método x, devuelva 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 cl.desafiolatam.repositorios;

import cl.desafiolatam.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
    @DisplayName("given crearPersona mocked method when crearPersona invoked
then mocked value returned")
    public void testCrearPersona() {
        Persona pepe = new Persona("1-2", "Pepe");
        when(repositorioPersona.crearPersona(pepe)).thenReturn("OK");
    }

}
```

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`, por ejemplo:

```
package cl.desafiolatam.repositorios;

import cl.desafiolatam.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
    @DisplayName("given crearPersona mocked method when crearPersona invoked
then mocked value returned")
    public void testCrearPersona() {
        Persona pepe = new Persona("1-2", "Pepe");
```

```
        when(repositorioPersona.crearPersona(null)).thenThrow(new
NullPointerException());
    }

}
```

Finalmente se crea creaPersonaRes 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 cl.desafiolatam.repositorios;

import cl.desafiolatam.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
    @DisplayName("given crearPersona mocked method when crearPersona invoked
then mocked value returned")
    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);
    }

}
```

La salida de `mvn test`, con las pruebas para el repositorio.

```
[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 cl.desafiolatam.repositorios.RepositorioPersonaTest
[INFO] Running cl.desafiolatam.servicios.ServicioPersonaTest
Jul 07, 2019 3:05:05 PM cl.desafiolatam.servicios.ServicioPersonaTest setup
INFO: Inicio clase de prueba
Jul 07, 2019 3:05:05 PM cl.desafiolatam.servicios.ServicioPersonaTest
testEliminarPersona
INFO: info eliminar persona
Jul 07, 2019 3:05:05 PM cl.desafiolatam.servicios.ServicioPersonaTest
testCrearPersona
INFO: info test crear persona
Jul 07, 2019 3:05:05 PM cl.desafiolatam.servicios.ServicioPersonaTest
testActualizarPersona
INFO: info actualizar persona
Jul 07, 2019 3:05:05 PM cl.desafiolatam.servicios.ServicioPersonaTest
testListarPersona
INFO: info listar persona
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.009 s
- in cl.desafiolatam.servicios.ServicioPersonaTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 5, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
-
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
-
[INFO] Total time:  3.056 s
[INFO] Finished at: 2019-07-07T15:05:05-04:00
[INFO] ------------------------------------------------------------------------
-
```

Se detalla que fueron exitosas.

El método de prueba para actualizarPersona, luce igual a testCrearePersona, salvo que se invocan distintos métodos del repositorio.

```
package cl.desafiolatam.repositorios;

import cl.desafiolatam.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
   @DisplayName("given actualizarPersona mocked method when actualizarPersona
invoked then mocked value returned")
   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);
   }

}
```

Para la prueba del método testEliminarPersona, se pasan parámetros similares, llamando a método eliminarPersona del repositorio.

```
package cl.desafiolatam.repositorios;

import cl.desafiolatam.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
    @DisplayName("given eliminarPersona mocked method when eliminarPersona
invoked then mocked value returned")
    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);
    }

}
```

Finalmente para método de prueba testListarPersona, se establece un mapa llamado mockRespuesta, el cuál es un `HashMap<String, String>`, que se 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. Finalmente se comprueba si el método ocurrió una vez usando verify.

```
package cl.desafiolatam.repositorios;

import cl.desafiolatam.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
    @DisplayName("given listarPersona mocked method when listarPersona invoked
then mocked value returned")
    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();
    }

}
```

Finalmente salida de ejecutar `mvn test`, con las pruebas para el repositorio.

```
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running cl.desafiolatam.repositorios.RepositorioPersonaTest
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.355 s
- in cl.desafiolatam.repositorios.RepositorioPersonaTest
[INFO] Running cl.desafiolatam.servicios.ServicioPersonaTest
Jul 07, 2019 4:19:19 PM cl.desafiolatam.servicios.ServicioPersonaTest setup
INFO: Inicio clase de prueba
Jul 07, 2019 4:19:19 PM cl.desafiolatam.servicios.ServicioPersonaTest
testEliminarPersona
INFO: info eliminar persona
Jul 07, 2019 4:19:19 PM cl.desafiolatam.servicios.ServicioPersonaTest
testCrearPersona
INFO: info test crear persona
Jul 07, 2019 4:19:19 PM cl.desafiolatam.servicios.ServicioPersonaTest
testActualizarPersona
INFO: info actualizar persona
Jul 07, 2019 4:19:19 PM cl.desafiolatam.servicios.ServicioPersonaTest
testListarPersona
INFO: info listar persona
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.02 s
- in cl.desafiolatam.servicios.ServicioPersonaTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 8, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
-
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
-
[INFO] Total time:  3.064 s
[INFO] Finished at: 2019-07-07T16:19:20-04:00
[INFO] ------------------------------------------------------------------------
-
```

Taller

Crear nuevo proyecto Java con Maven ya sea con Eclipse o mediante consola

```
mvn archetype:generate -DgroupId=cl.desafiolatam -DartifactId=testing-practico
-DarchetypeArtifactId=maven-archetype-quickstart -DarchetypeVersion=1.4 -
DinteractiveMode=false
```

La estructura de directorios del proyecto queda así

```
testing-practico
├── pom.xml
└── src
    ├── main
    │   └── java
    │       └── cl
    │           └── desafiolatam
    │               └── App.java
    └── test
        └── java
            └── cl
                └── desafiolatam
                    └── AppTest.java
```

Ahora agregar la dependencia de JUnit 5 al proyecto, eliminando la dependencia de JUnit 4 que viene por defecto. Y se agrega también la dependencia de Mockito.

```
<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>
```

Con las dependencias listas en el proyecto se pueden escribir las pruebas usando las caracterísitcas que ofrecen JUnit y Mockito.

A continuación crear modelo Comentario y el repositorio ComentarioRepositorio, el cuál persiste información de los comentarios pertenecientes a un album.

La estructura de directorios es la siguiente:

```
testing-practico
├── pom.xml
└── src
    ├── main
    │   └── java
    │       └── cl
    │           └── desafiolatam
    │               ├── modelos
    │               │   └── Comentario.java
    │               └── repositorios
    │                   └── ComentarioRepositorio.java
    └── test
        └── java
            └── cl
                └── desafiolatam
```

Modelo Comentario:

```
package cl.desafiolatam.modelos;

import java.util.Date;

public class Comentario {

    private Long id;
    private Long usuarioId;
    private Long albumId;
    private String detalle;
    private Date fecha;

    //constructor, getters y setters

}
```

Clase ComentarioRepositorio, que contiene metodos crear, actualizar y listar, que intervendrían directamente con otros componentes, por lo tanto debe ser simulado en las pruebas.

```
package cl.desafiolatam.repositorios;

import cl.desafiolatam.modelos.Comentario;

import java.util.ArrayList;
import java.util.List;

public class ComentarioRepositorio {

    private List<Comentario> db = new ArrayList<>();

    public String crear(Comentario comentario) {
        if (comentario != null) {
            db.add(comentario);
```

```
        } else {
            throw new NullPointerException();
        }
        return "OK";
    }

    public String actualizar(Comentario comentario) {
        if (comentario != null) {
            db.add(comentario);
        } else {
            throw new NullPointerException();
        }
        return "OK";
    }

    public List<Comentario> listarTodosLosComentarios() {
        return db;
    }

}
```

Para estar seguros de que la clase ComentarioRepositorio se comporta de la manera en la que se espera, se escriben sus respecticas pruebas unitarias.

Se crea la clase de pruebas ComentarioRepositorioTest, la estructura de directorios queda así:

```
testing-practico
├── pom.xml
└── src
    ├── main
    │   └── java
    │       └── cl
    │           └── desafiolatam
    │               ├── modelos
    │               │   └── Comentario.java
    │               └── repositorios
    │                   └── ComentarioRepositorio.java
    └── test
        └── java
            └── cl
                └── desafiolatam
                    └── repositorios
                        └── ComentarioRepositorioTest.java
```

La clase ComentarioRepositorioTest, contiene lo ya visto, sin embargo se inicializa el mock del repositorio y un comentario dentro de una fixture. El método setup, anotado con `@BeforeAll` se encarga de inicializar los objetos que serán compartidos por todas las pruebas.

A continuación la clase ComentarioRepositorioTest:

```
package cl.desafiolatam.repositorios;

import cl.desafiolatam.modelos.Comentario;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;

import java.util.*;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.mockito.Mockito.*;

public class ComentarioRepositorioTest {

    private static ComentarioRepositorio comentarioRepositorio;
    private static Comentario primerComentario;

    @BeforeAll
    static void setup() {
        primerComentario =
                new Comentario(1L,1L,1L,"Agregado en carga inicial",new
Date());
        comentarioRepositorio = mock(ComentarioRepositorio.class);
    }

    @Test
    @DisplayName("given crear mocked method " +
            "when crear invoked " +
            "then mocked value returned")
    public void testCrearComentario() {
        when(comentarioRepositorio.crear(primerComentario)).thenReturn("OK");
        String creado = comentarioRepositorio.crear(primerComentario);
        assertEquals("OK" , creado);
        verify(comentarioRepositorio).crear(primerComentario);
    }

    @Test
    @DisplayName("given actualizar mocked method " +
            "when actualizar invoked " +
            "then mocked value returned")
    public void testActualizarComentario() {
        primerComentario.setDetalle("Actualizado en prueba");

 when(comentarioRepositorio.actualizar(primerComentario)).thenReturn("OK");
        String actualizado =
comentarioRepositorio.actualizar(primerComentario);
        assertEquals("OK" , actualizado);
        verify(comentarioRepositorio).actualizar(primerComentario);
    }


    @Test
    @DisplayName("given listarTodosLosComentarios mocked method " +
            "when listarTodosLosComentarios invoked " +
            "then mocked value returned")
```

```
    public void testListarTodosLosComentarios() {
        List<Comentario> mockRespuesta = new ArrayList<>();

 when(comentarioRepositorio.listarTodosLosComentarios()).thenReturn(mockRespue
sta);
        List<Comentario> listarRes =
comentarioRepositorio.listarTodosLosComentarios();
        assertEquals(mockRespuesta, listarRes);
        verify(comentarioRepositorio).listarTodosLosComentarios();
    }

}
```

Para correr las pruebas, basta con ejecutar `mvn test`. La salida de estas pruebas es:

```
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running cl.desafiolatam.repositorios.ComentarioRepositorioTest
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.562 s
- in cl.desafiolatam.repositorios.ComentarioRepositorioTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
-
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
-
[INFO] Total time:  4.531 s
[INFO] Finished at: 2019-07-08T00:21:00-04:00
[INFO] ------------------------------------------------------------------------
-
```

Test Driven Development

Competencias

- Entender fases de TDD (Verde, Roja y de Refactorización)
- Entender que la cobertura de las pruebas no necesariamente debe ser de 100%
- Escribir pruebas usando características de JUnit
- Desarrollar funcionalidades siguiendo la metodología de TDD

¿Por qué TDD?

La respuesta corta es porque es la forma más sencilla de lograr tanto un código de buena calidad como una buena cobertura de prueba.

¿Qué es exactamente el desarrollo guiado por pruebas?

TDD es un procedimiento en donde se escriben las pruebas antes de la implementación real. Es el cambio de un enfoque tradicional donde las pruebas se realizan después de que se escribe el código.

Empezar un nuevo desarrollo usando TDD trae multiples beneficios por sobre el enfoque tradicional de escribir las pruebas después. A partir de las historias de usuario o requisitos ya definidos, se debe tener claro cuales serán las características de una pieza de código, ahí el primer punto bueno, es que se tiene claro cuál es la finalidad de la pieza de código antes de escribir su implementación para producción. Con las funcionalidades claras, se entiende mejor la problematica y se aborda de mejor manera. Con el desarrollo en etapas avanzadas, puede que tenga que hacer cambios, y hacerlos conlleva la desconfianza de que el código puede comportarse de maneras inesperadas. Con las pruebas ya escritas, al realizar un cambio el desarrollador debe encargarse de mantener las pruebas en verde (exitosas), quitando el miedo al cambio. Los detalles de TDD se verán a continuación

Motivación

No es algo fácil llegar dominar TDD. Incluso cuando ya se aprende toda la teoría y al trabajar con las mejores prácticas. TDD requiere tiempo y mucha práctica.

Es un viaje largo que de hecho, nunca termina. Siempre hay nuevas formas de llegar a ser más competente y más rápido. Sin embargo, a pesar de que el costo es alto, los beneficios son incluso mayores. Las personas que pasan el tiempo suficiente practicando TDD son defensores y propulsores de esta practica para desarrollar software por sus beneficios. Para aprender TDD se deben ensuciar las manos y programar. El objetivo es hacer sentir al desarrollador cómodo con TDD y tener una base sólida, tanto en la teoría como en la práctica, el resto depende del desarrollador y la experiencia que se tendrá construyéndola y aplicándola a diario.

Reglas del Juego

- No se le permite escribir ningún código de producción a menos que sea para hacer una prueba de prueba fallida.
- No está permitido escribir más de una prueba de unidad de lo que es suficiente para fallar; y los fallos de compilación son fallos.
- No se le permite escribir más código de producción del que sea suficiente para pasar la única prueba de la unidad.

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

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

Estas reglas también es útil utilizarlas como lista de verificación cuando se está desarrollando, por lo que simplemente repiten en orden, una y otra vez, para mantenerse en el ciclo de TDD.

Estas reglas son simples, pero las personas que se acercan a TDD a menudo violan uno o más de ellos. Esas 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. A ver de qué se trata. Se basa en la repetición de un proceso bastante claro.

> Diagrama de TDD

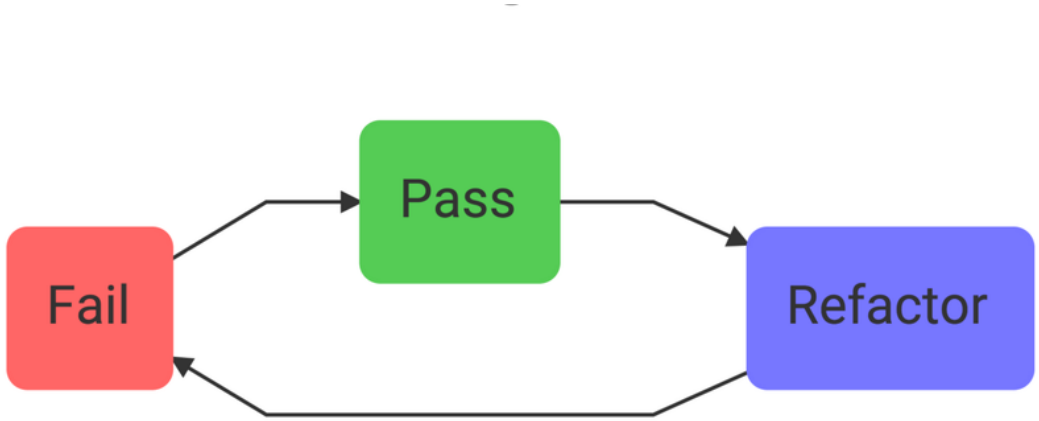

Imagen 4. Diagrama de TDD.

Este ciclo de desarrollo mostrado en la Imagen 4, se basa en el primer concepto de prueba de la programación extrema en donde se fomenta el diseño simple con un alto nivel de confianza. El procedimiento en sí es simple y consiste en unos 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 se tendrá éxito y finalmente mejorar.

Fase Roja

Es en esta fase se debe concentrar en escribir una interfaz limpia para futuros usuarios. Esta es la fase en la que diseñas la forma en que los clientes utilizarán tu código. En la fase roja, se debe escribir una prueba sobre un comportamiento que está a punto de implementar. Para hacerlo se debe escribir una prueba que use un fragmento de código como si ya estuviera implementado, embargo esa implementación no existe, es necesario olvidarse de la implementación. Si en esta fase, se está pensando en cómo implementar el código o cómo va a escribir el código de producción se está haciendo mal. Lo que se hace es escribir una prueba para que luego se pueda escribir el código de producción, la implementación vienen después. En esta fase no se escribe una prueba para probar tu código ya implementado y aquí viene un error que puede cometerse, cuando se escriben métodos adicionales, para desacoplar su programa, bueno lo que se debe hacer es no escribir un montón de métodos o clases que crea que pueda necesitar más adelante. Lo importante es concentrarse en la función que está escribiendo y en lo que realmente se necesita.

En esta fase, debe tomar decisiones sobre cómo se utilizará el código. Basándose en lo que realmente se necesita en este momento y no en lo que se cree que pueda ser necesario. Escribir algo que la característica no requiere es ingeniería excesiva.

¿Qué pasa con la abstraer el 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, porque en esta fase se escribe el código de producción. Si eres un programador, lo haces todo el tiempo. Aquí viene otro gran error: en lugar de escribir suficiente código para pasar la prueba unitaria fallida, se escriben más algoritmos y métodos. Mientras hace esto, probablemente esté pensando en cuál es la mejor implementación y la que tenga un mejor rendimiento. 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: Una tarea sencilla es menos propensa a errores, y en esta fase se minimizan los errores. No se debe mezclar el código que se está probando con el código que no lo está. ¿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 esta fase, debe actuar como un desarrollador que tiene una tarea simple, escribir una solución sencilla que convierta 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 al desarrollador 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 le debe modificar el código, siempre y cuando se estén manteniendo todas las pruebas en verde, para que sea mejor. Que significa mejor depende del desarrollador. Pero existe una tarea obligatoria, se tienes que eliminar el código duplicado, Kent Becks sugiere en su libro que eliminar todo código duplicado es todo lo que necesita hacer. Ahora juegan su rol los desarrolladores exigentes que refactorizan el código para llevarlo a un buen nivel. En la fase roja se está expresando claramente la intención de las pruebas para los usuarios (código de producción), pero en la refactorización se muestran las habilidades del desarrollador a los demás desarrolladores que leerán el código.

Aplicando TDD

Se tiene el caso en donde se quiere controlar una liga de futbol, y una de las partes del programa se necesita comparar dos equipos para ver quien va primero, si se quiere comparar, cada equipo debe recordar la cantidad de partidos que ha ganado y el mecanismo de comparación los utilizará.

Entonces, una clase de EquipoFutbol necesita un campo en el que guardar esta información, y este campo 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 comprobar qué sucede cuando dos equipos tienen el mismo número victorias.

Crear proyecto llamado gs-tdd usando Eclipse o con el siguiente comando:

```
mvn archetype:generate -DgroupId=cl.desafiolatam -DartifactId=gs-tdd -DarchetypeArtifactId=maven-archetype-quickstart -DarchetypeVersion=1.4 -DinteractiveMode=false
```

Se agregan las dependencias en el 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>
```

Pruebas - Fase Roja

Para poder 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:

```
gs-tdd
├── pom.xml
└── src
    └── test
        └── java
            └── cl
                └── desafiolatam
                    └── EquipoFutbolTest.java
```

Se debe asegurar que el constructor funcione, para eso la clase EquipoFutbolTest contiene una prueba que llama a la clase EquipoFutbol y revisa si el constructor recibe el número de partidos ganados.

```java
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.

No hacer mas que lo necesario para que la prueba compile

A continuación crar la clase `EquipoFutbol` y su respectivo método, pero sin agregar lógica de negocios. La estructura de directorios queda así:

```
gs-tdd
├── pom.xml
└── src
    ├── main
    │   └── java
    │       └── cl
    │           └── desafiolatam
    │               └── EquipoFutbol.java
    └── test
        └── java
            └── cl
                └── desafiolatam
                    └── EquipoFutbolTest.java
```

La clase `EquipoFutbol` solo contiene lo necesario para que la prueba compile

```java
package cl.desafiolatam;

public class EquipoFutbol {


    public EquipoFutbol(int juegosGanados) {
    }

    public int getJuegosGanados() {
        return 0;
    }

}
```

Se se ejecuta `mvn test` la prueba falla

```
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running cl.desafiolatam.EquipoFutbolTest
[ERROR] Tests run: 1, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0.023
s <<< FAILURE! - in cl.desafiolatam.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] ------------------------------------------------------------------------
-
```

Y al ejecutar `mvn compile` se compila correctamente

```
[INFO] ------------------------------------------------------------------------
-
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
-
[INFO] Total time:  2.620 s
[INFO] Finished at: 2019-07-08T17:38:21-04:00
[INFO] ------------------------------------------------------------------------
-
```

Escribir el código de producción, en donde irá la lógica de negocios es más sencilla una vez que las pruebas están listas, se puede decir que escribir la prueba fue mas exigente.

La prueba fallida en este punto, es algo bueno ya que la prueba está escrita para verificar como se comporta la clase bajo prueba.

En el caso de que alguna vez rompamos nuestra clase bajo prueba, por lo que la prueba fallará, el mensaje de error dirá exactamente que 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

La refactorización es sencilla esta vez: lo que se debe hacer es 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 `EquipoFubol`.

```
package cl.desafiolatam;

public class EquipoFutbol {

    private int juegosGanados;

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

    public int getJuegosGanados() {
        return juegosGanados;
    }

}
```

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 haya realizado, vuelva a ejecutar la prueba para asegurarse de que nada se ha roto accidentalmente.

La salida de `mvn test`

```
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running cl.desafiolatam.EquipoFutbolTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.018 s
- in cl.desafiolatam.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

En el caso de este ejemplo, algo simple como la clase de EquipoFutbol, no hay 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());
    }
}
```

La salida de `mvn test` sigue siendo exitosa

```
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running cl.desafiolatam.EquipoFutbolTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.027 s
- in cl.desafiolatam.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] ------------------------------------------------------------------------
-
```

Acaba de terminar su primer ciclo de TDD, pasando por fase roja en donde falla la prueba, luego en fase verde se escribe solo el código necesario para pasar la prueba. Finalmente se refactoriza, se deja el código lo mejor posible, ¿Qué es mejor? Eso depende del desarrollador.

Consideraciones

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

Lo que toma tiempo es aprender y dominar TDD, así cómo configurar y usar un entorno de prueba. Cuando está familiarizado con las herramientas de prueba y la técnica TDD, en realidad no requiere más tiempo. Por el contrario, ayuda a mantener 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 va a implementar no está bien diseñado, se encontrará con casos que no consideró. Y esto significa que tendrá que eliminar la prueba y el código que 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 las que pueden cambiar a menudo.

Se puede escribir código con muy pocos errores, no necesita pruebas

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