Pruebas unitarias y TDD

Test Driven Development

{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?

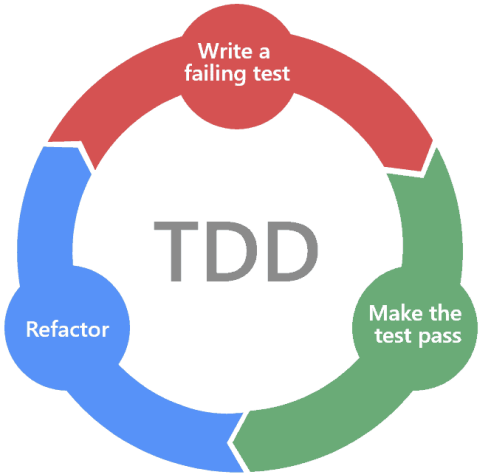
- *Comprender las fases de TDD para ser escritas usando características JUnit.*
- *Desarrollar funcionalidades siguiendo la metodología de TDD para aplicarlas en Java.*

¿Qué es TDD?

/* Test Driven Development */

¿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?

- Es un procedimiento que escribe las pruebas antes de la implementación real.
- 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.

Reglas del juego

1. No está permitido escribir ningún código de producción, a menos que sea para hacer una prueba fallida.

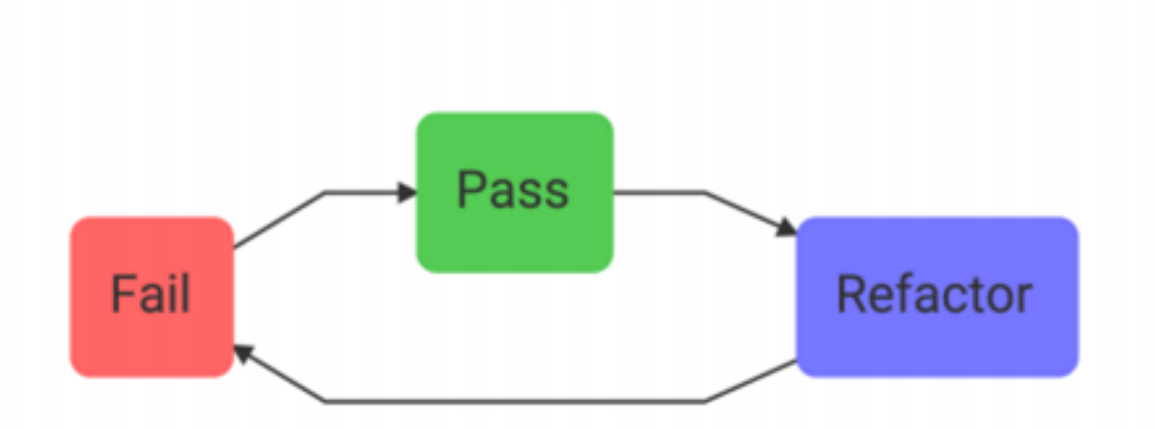
1. No está permitido escribir más de una prueba de unidad para fallar. Los fallos de compilación son exactamente eso, fallos.

1. No está permitido escribir más código de producción del que sea suficiente para pasar la prueba de la unidad.

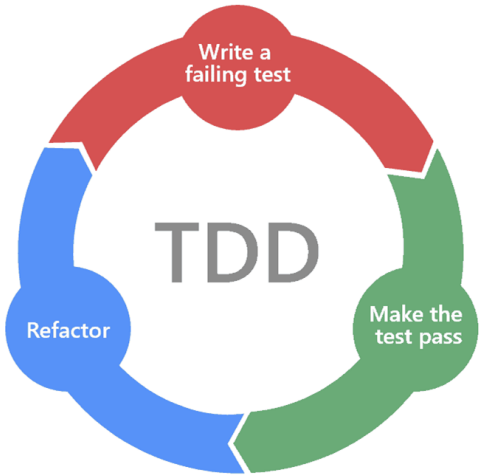
Reglas del juego

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:

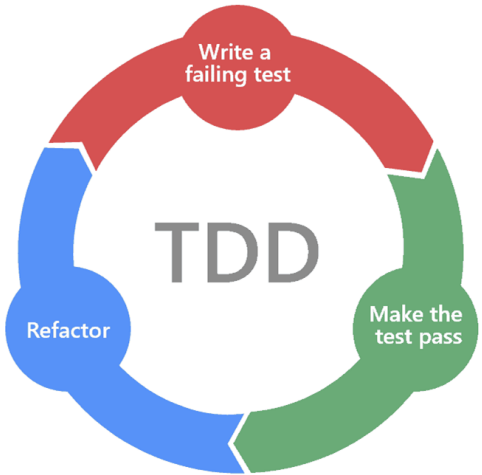


Fase roja



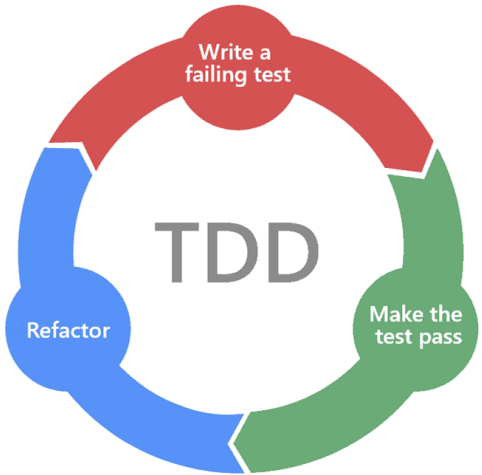
- Se debe concentrar en escribir una interfaz limpia para futuros usuarios.
- Se diseña la forma en que los clientes utilizarán tu código.
- Se debe escribir una prueba sobre un comportamiento que está a punto de implementarse.
- Se deben tomar decisiones sobre cómo se utilizará el código, basándose en lo realmente necesario y no en lo que se cree que pueda ser necesario.

Fase verde



- Se escribe el código de producción.
- 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.
- 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 refactor



- Se debe modificar el código siempre y cuando se mantengan todas las pruebas en verde para que sea mejor.
- Existe una tarea obligatoria: se debe eliminar el código duplicado.
- Se demuestran las habilidades del desarrollador a los demás desarrolladores que leerán el código.

Ejercicio guiado

Fases

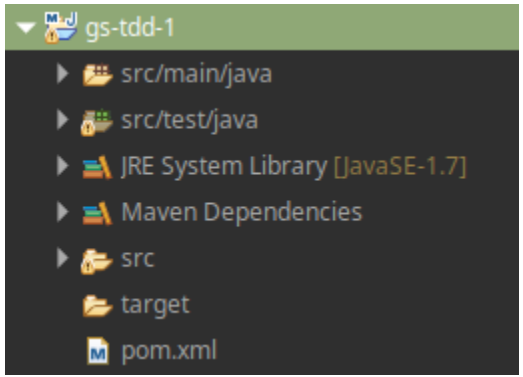
Se tiene el caso donde se quiere controlar una liga de fútbol femenino, y una de las partes del programa necesita comparar dos equipos para ver quién va primero. Si se quiere comparar cada equipo, se debe recordar la cantidad de partidos que ha ganado y el mecanismo de comparación usará estos datos.

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.

Fase Roja

Paso 1

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



Fase Roja

Paso 1

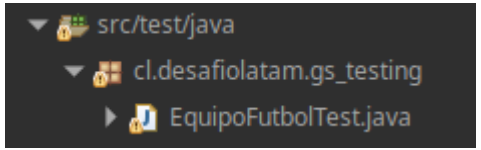
Agregar las dependencias en el archivo pom.xml

```
<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>
```

Fase Roja

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:



Fase Roja

Paso 4

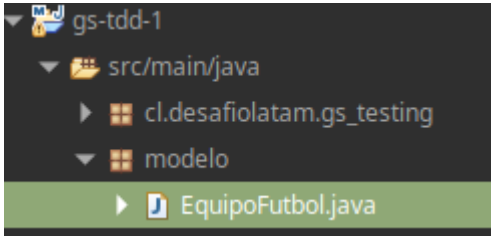
Para asegurar que el constructor funcione, la clase EquipoFutbolTest debe contener una prueba que llama a la clase EquipoFutbol y revisar si el constructor recibe el número de partidos ganados

```
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());
 }
}
```

Fase Roja

Paso 5

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



Fase Roja

Paso 6

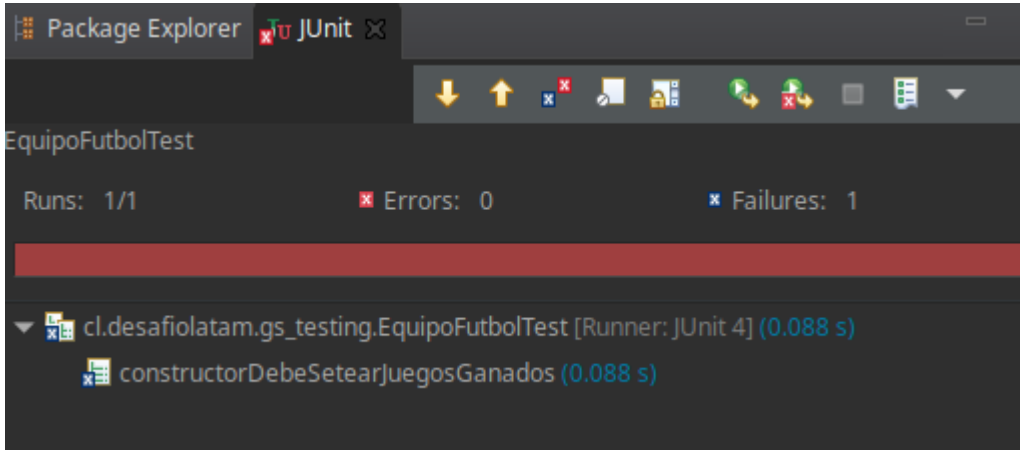
La clase EquipoFutbol solo contendrá lo necesario para que la prueba compile.

```
package cl.desafiolatam;
public class EquipoFutbol {
     public EquipoFutbol(int juegosGanados) {
     }
     public int getJuegosGanados() {
          return 0;
          }
}
```

Fase Roja

Paso 7

Ejecutamos el Maven Test para que la prueba falle



{desafío} latam_

Fase Verde

Paso 8

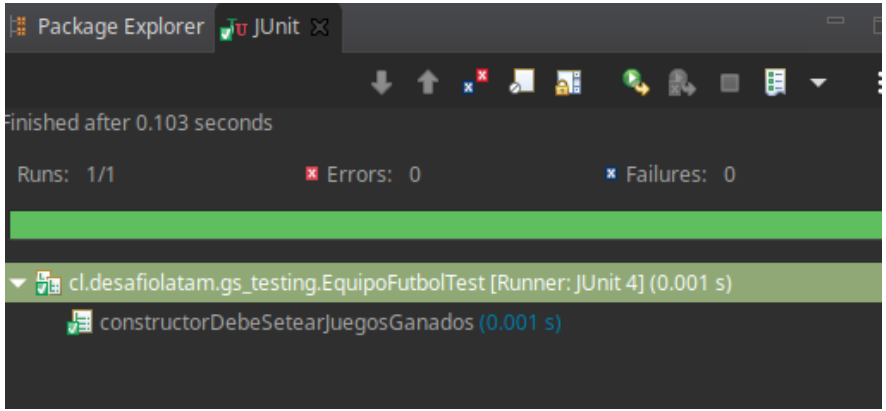
Almacenar el valor pasado como parámetro del constructor a alguna variable interna.

```
package cl.desafiolatam;
public class EquipoFutbol {
     private int juegosGanados;
     public EquipoFutbol(int juegosGanados) {
          this.juegosGanados = juegosGanados;
          }
          public int getJuegosGanados() {
                    return juegosGanados;
          }
}
```

Fase Verde

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.



{desafío} latam_

Fase Refactor

Paso 10

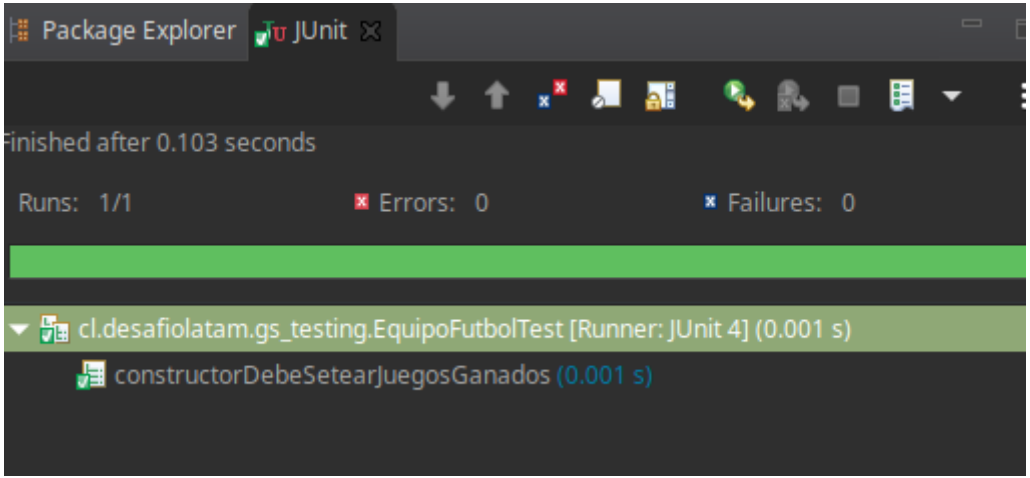
La refactorización será deshacerse del número 3 como parámetro de assertEquals, usando una variable CUATRO_JUEGOS_GANADOS.

```
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());
            }
}
```

Fase Refactor

Paso 11

La salida de mvn test sigue siendo exitosa:



{desafío} latam_

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 comparte 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.

¿Cuál es la fase que se debe concentrar en escribir una interfaz limpia para futuros usuarios?

¿Con TDD se necesita dedicar tiempo al análisis?

Próxima sesión...

- *Guia de ejercicios*

{desafío}
latam_

{desafío}
latam_

Academia de
talentos digitales