Test Driven Development

¡Comencemos!

¿Qué aprenderás?

- 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.

¡Vamos con todo!

¿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:

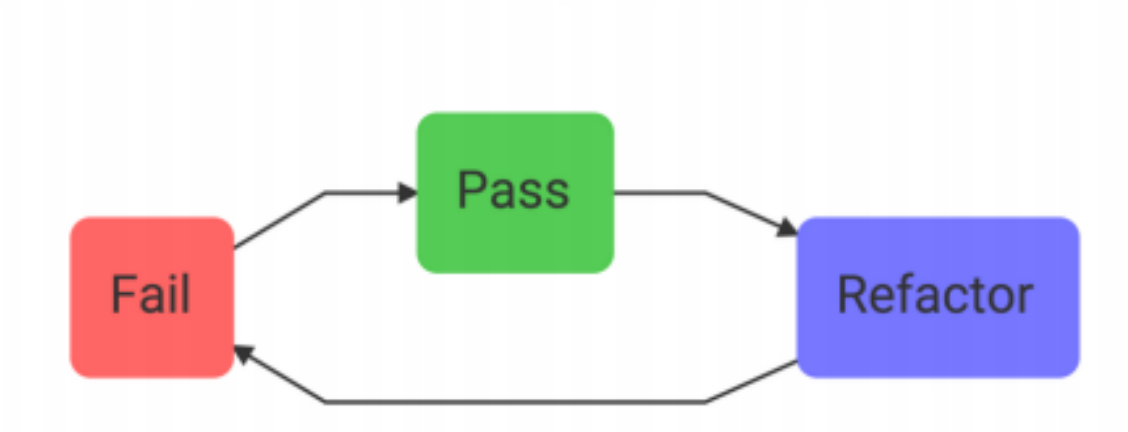

Imagen 1. Diagrama de TDD.
Fuente: Desafio Latam.

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 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.

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.

Pruebas - Fase Roja

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

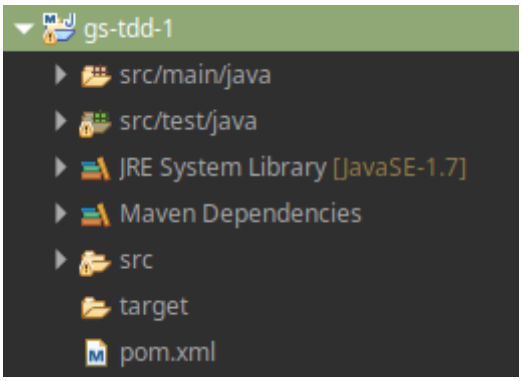

Imagen 2. Estructura de proyecto “gs-tdd-1”.
Fuente: Desafio Latam.

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:

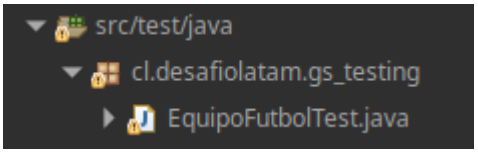

Imagen 3. Carpeta test proyecto “gs-tdd-1”.
Fuente: Desafio Latam.

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:

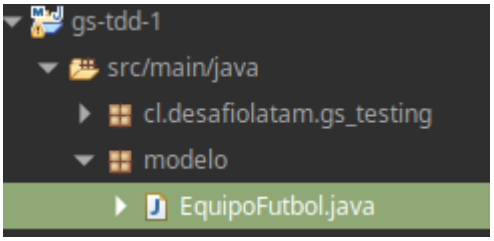

Imagen 4. Creación clase EquipoFutbol.
Fuente: Desafio Latam.

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:

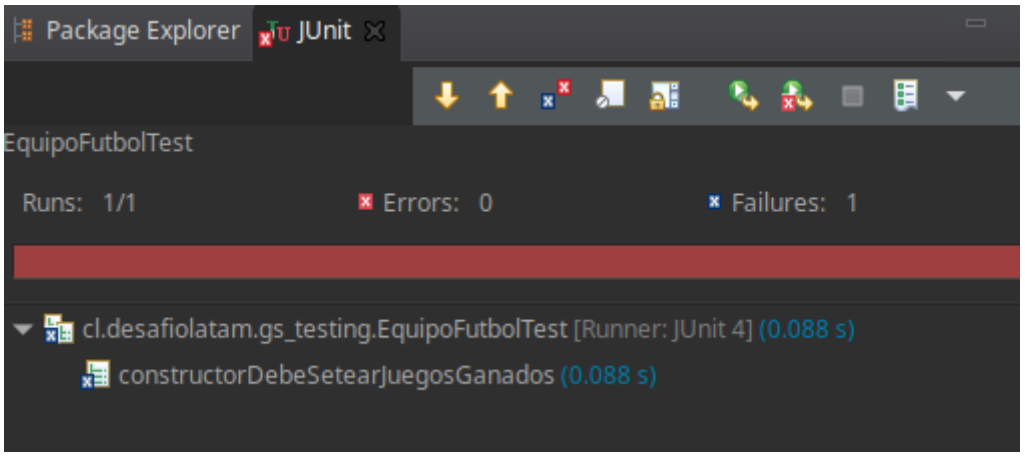

Imagen 5. Falla a propósito del test.
Fuente: Desafio Latam.

```
[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.

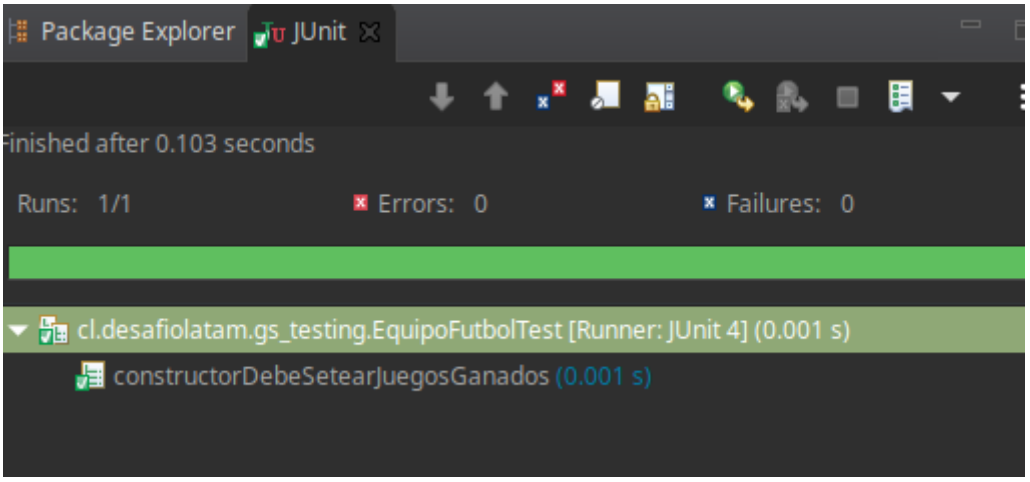

Imagen 6. Test que pasó con éxito
Fuente: Desafio Latam

```
[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:

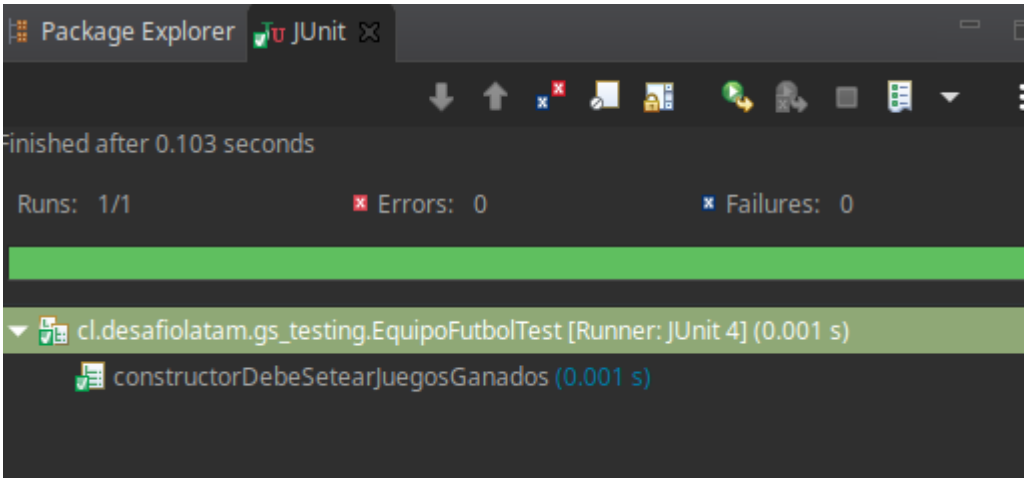

Imagen 7. Test refactorizado que pasó con éxito
Fuente: Desafio Latam

```
[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.