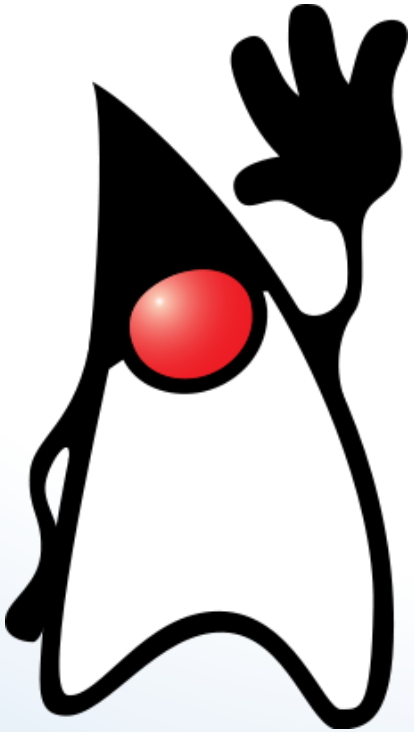
.UBAfiuba
FACULTAD DE INGENIERÍA

Pruebas de Software

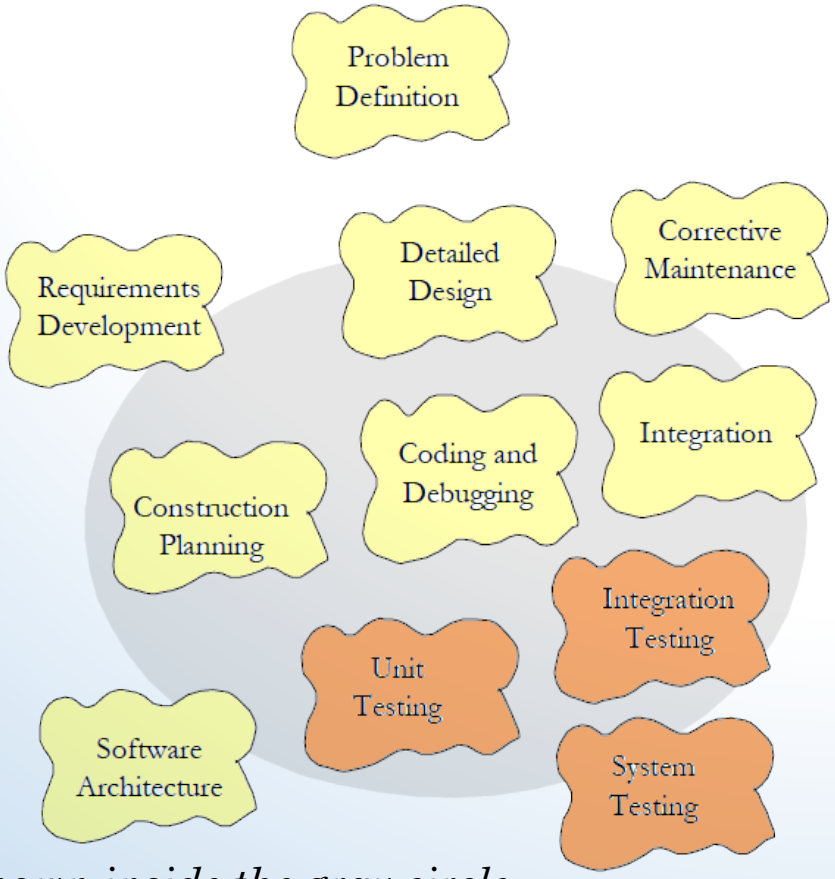


Cátedra: Corsi - Essaya - Maraggi

¿Qué son las pruebas de software?

Las pruebas son una **disciplina característica del desarrollo de software**.

Son una actividad de **control de la calidad** (QC = *Quality Control*)



Construction activities are shown inside the gray circle. Construction focuses on coding and debugging but also includes some detailed design, unit testing, integration testing and other activities.

Construction is also sometimes known as “coding” or “programming.” “Coding” isn’t really the best word because it implies the mechanical translation of a preexisting design into a computer language; construction is not at all mechanical and involves substantial creativity and judgment. . . . I use “programming” interchangeably with “construction.”

McConnell, S. (2004). *Code Complete, 2da. ed.*

Pero NO son las pruebas las que confieren calidad al producto:

> "Testing by itself does not improve software quality. Test results are an indicator of quality, but in and of themselves, they don't improve it. Trying to improve software quality by increasing the amount of testing is like trying to lose weight by weighing yourself more often. What you eat before you step onto the scale determines how much you will weigh, and the software development techniques you use determine how many errors testing will find. If you want to lose weight, don't buy a new scale; change your diet. If you want to improve your software, don't test more; develop better."

McConnell, S. (2004). *Code Complete, 2da. ed.*

Las pruebas solo tienen el objetivo de detectar errores; y cuando los detecten, se deberá mejorar la calidad del producto.

> **Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence.**

Dijkstra, E. *The Humble Programmer*. ACM Turing Lecture 1972

Hay prácticas de **aseguramiento de la calidad** (QA = *Quality Assurance*) para mejorar el proceso de desarrollo con vistas a que el producto final tenga mejor calidad: DevOps, CI/CD (integración continua/entrega continua), etc.

Pruebas de verificación (*pruebas técnicas*)

vs.

Pruebas de validación (*pruebas de usuarios*)

Verificar: controlar que hayamos construido el producto tal como pretendimos construirlo.

Validar: controlar que hayamos construido el producto que nuestro cliente quería.

Supongamos que un cliente nos pide que desarrollemos un juego de TaTeTi para celulares. Luego de hablar con él sobre colores, plataformas, etc., el equipo se pone a trabajar. Al cabo de unos días, se reúne con el cliente y le muestra el producto: impecable en diseño, se ejecuta en las plataformas pedidas, juega contra el usuario… Sin embargo, nuestro cliente no está satisfecho y afirma: "Esto no es lo que pedí: yo preferiría jugar en red contra otro jugador".

¿Qué fue lo que falló? Evidentemente, él tenía una expectativa que no fue cubierta: el juego en red. También hay algo que en el sistema que él no deseaba: la autonomía de la aplicación, que le permite jugar contra el usuario. Ocurre que las pruebas que el equipo ejecutó estaban centradas en la **verificación**: entendimos algo, diseñamos una solución y verificamos que el programa hace lo que nos propusimos. Lo que falló fue la **validación**: lo que construimos no es lo que quiere el usuario.

Fuente: Adaptado de Fontela, C. (2018). *Pruebas de software*

Pruebas funcionales vs. Pruebas de atributos de calidad

(Pruebas no funcionales o de rendimiento)

- Si intento colocar una ficha en una celda ocupada, el programa debe impedirlo.
- Si quiero jugar dos veces seguidas sin esperar que juegue el adversario, el programa debe impedirlo.
- Si coloco una cruz en un casillero, esa cruz debe permanecer allí hasta el final del juego.
- Si al colocar una cruz o círculo quedan tres cruces o círculos en línea, el usuario que jugó acaba de ganar y el juego debe terminar.

- El tiempo de respuesta del juego no puede ser superior a 1 segundo.
- El programa debe correr en las dos plataformas que soporten la mayor cantidad de dispositivos en uso en América Latina.
- Si se produce un error en el programa, el dispositivo móvil debe seguir funcionando para todas las demás aplicaciones.

Pruebas de atributos de calidad

- **Pruebas de compatibilidad (*Compatibility testing*)**: chequean las diferentes configuraciones de hardware o de red y de software que debe soportar el producto.
- **Pruebas de rendimiento (*Performance testing*)**: evalúan el rendimiento en el uso habitual.
- **Pruebas de resistencia o de estrés (*Stress testing*)**: comprueban el comportamiento del sistema ante situaciones donde se demanden cantidades extremas de recursos.
- **Pruebas de recuperación (*Recovery testing*)**: chequean que el programa se recupere de fallas.
- **Pruebas de conformidad (*Compliance testing*)**: determina si un software cumple con un conjunto definido de estándares internos o externos antes de su lanzamiento.
- **Pruebas de seguridad (*Security testing*)**: comprueban que solo los usuarios autorizados accedan a las funcionalidades y que el programa soporte ataques.
- **Pruebas de instalación (*Install testing*)**: verifican que el sistema puede ser instalado satisfactoriamente en el equipo del cliente.
- **Pruebas de usabilidad (*Usability testing*)**: evalúan la facilidad de uso.
- **Pruebas de localización (*Localization testing*)**: verifican el comportamiento de un producto de acuerdo con los entornos locales o culturales específicos.

Pruebas de verificación (*pruebas técnicas*)

Las **pruebas de verificación** son pruebas que los propios desarrolladores ejecutan para ver que están logrando que el programa funcione como ellos pretenden.

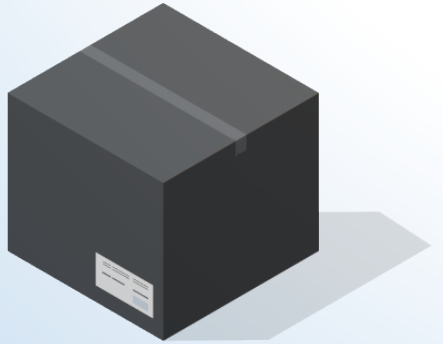
- Las **pruebas unitarias (*Unit testing*)** verifican pequeñas porciones de código. Por ejemplo, verifican alguna responsabilidad única de un método.
- Las **pruebas de integración (*Integration testing*)** prueban que varias porciones de código, trabajando en conjunto, hacen lo que pretendíamos. Por ejemplo, se trata de pruebas que involucran varios métodos, clases o incluso subsistemas enteros.
- Las **pruebas de regresión (*Regression testing*)** garantizan que la aplicación siga funcionando correctamente después de producirse algún cambio en el código.
- Las **pruebas de punta a punta (*E2E o End-to-End testing*)** verifican funcionalidades que atraviesan todas las capas de la aplicación, p. ej. desde la pantalla a la base de datos.
- Las **pruebas de sistema (*System testing*)** evalúan cómo los diferentes componentes interactúan juntos en la aplicación completa e integrada.

Pruebas de verificación (*pruebas técnicas*)

Hay ocasiones en que debemos probar código que necesita de otros objetos, métodos o funciones para poder funcionar: en esas situaciones se utilizan **objetos ficticios** o dobles de prueba (*stubs*, *mocks*, *fakes*, etc.), que ayudan a aislar el código a probar.

Las pruebas de verificación podrían ser de caja negra o de caja blanca:

- Decimos que una prueba es de **caja negra** cuando la ejecutamos sin mirar el código que estamos probando.
- Cuando, en cambio, analizamos el código durante la prueba, decimos que es una prueba de **caja blanca**.



Black box - we do not know anything

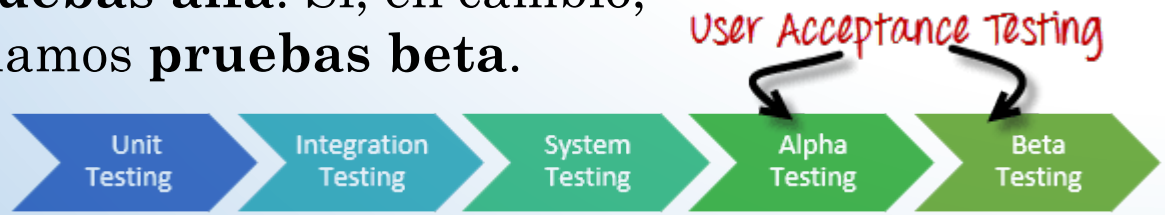
White box - we know everything

En general, se prefiere probar el funcionamiento y no verificar la calidad del código.

Pruebas de validación (*pruebas de usuarios*)

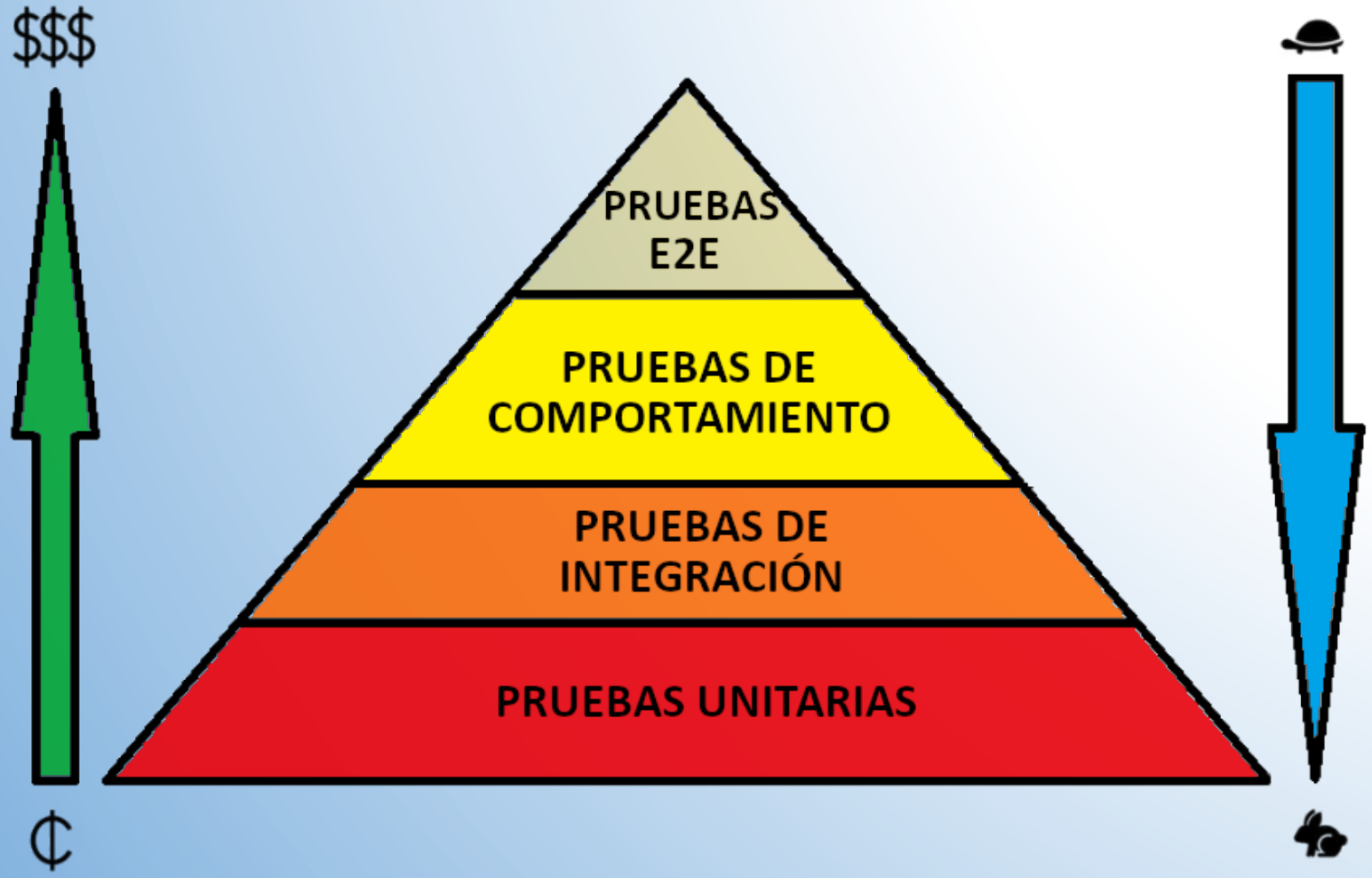
Para validar que hayamos construido el producto que nuestros clientes querían, se usan **pruebas de aceptación de usuarios (*User Acceptance Tests*)**. En general, se suele trabajar con pruebas de aceptación diseñadas por usuarios – o al menos en conjunto con usuarios, o en el caso extremo, diseñadas por el equipo de desarrollo y validadas por usuarios – pero **ejecutadas por el equipo de desarrollo** en un entorno lo más parecido posible al que va a utilizar el usuario.

No obstante, cuando un sistema debe salir a producción, se suele poner a disposición de usuarios reales para que ellos mismos ejecuten las pruebas. Si las pruebas las hacemos en un entorno controlado por el equipo de desarrollo, se las llama **pruebas alfa**. Si, en cambio, el producto lo prueba el cliente en su entorno, las llamamos **pruebas beta**.



Todas estas pruebas que hemos mencionado suponen que ejecutamos el sistema completo, con sus conexiones con otros sistemas, con su interfaz de usuario, sus medios de almacenamiento de datos persistentes, etc. Hay ocasiones en que deseamos probar solamente *comportamiento*, en el sentido de que la lógica de la aplicación es correcta, pero sin interfaz de usuario. Estas pruebas más limitadas se denominan **pruebas de comportamiento**.

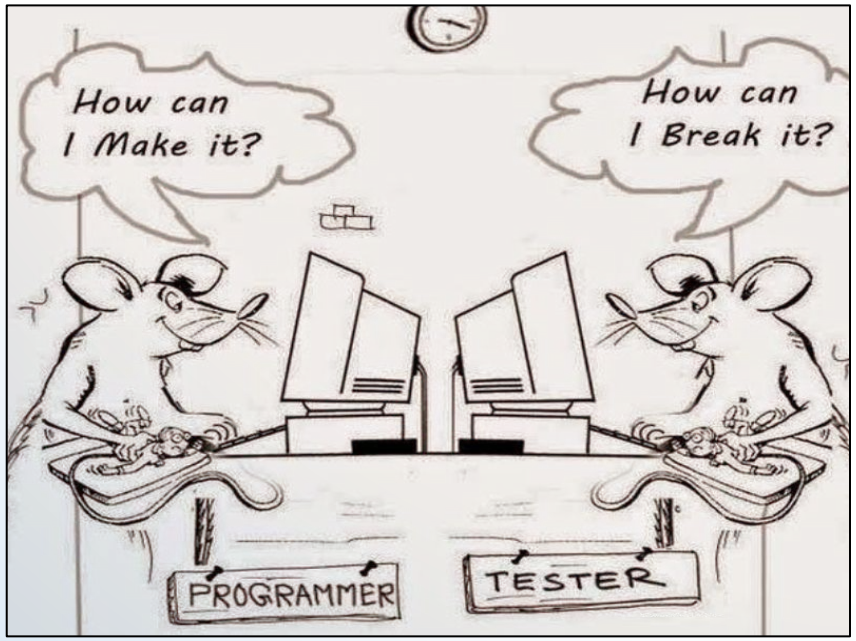
La Pirámide de Pruebas



Roles: Programadores y Testers

Visión tradicional

Separa roles y responsabilidades en dos equipos distintos dentro del desarrollo: el equipo de *programadores* y el equipo de *testers*. Se basa en la noción administrativa de *control cruzado por contraposición de intereses*, que hace que quien ejecuta un trabajo no puede ser el mismo que lo controla.



Visión ágil

La responsabilidad por entregar un producto de calidad y sin errores pasó a ser de todo el equipo de desarrollo (*programadores* y *testers*). Scrum incluso llega a plantear que no debe haber roles diferentes dentro del equipo: todos son parte del *Scrum Delivery Team*, y de afuera no hay distinción entre *programadores* y *testers*.

Algunas herramientas para automatizar pruebas

Pruebas unitarias y de integración

Se destacan los *frameworks* **xUnit**, llamados así genéricamente porque los primeros en surgir se denominaban **SUnit**, **JUnit**, **NUnit**, etc. Más allá del "unit" que aparece en su nombre, suelen ser usados para todo tipo de pruebas técnicas, unitarias o de integración. Hay otros *frameworks* que se han cuidado de no usar la palabra "unit" como parte del nombre: **TestNG**, por ejemplo, o **Mockito** (usado para crear *dobles de prueba*).

Pruebas de comportamiento

Hay herramientas basadas en tablas, como **FIT** y **FitNesse**, otras basadas en texto con formato especial, como **RSpec**, **JBehave**, **Cucumber** y **Behave**, y las hay también de texto más libre, como **Concordion**.

Pruebas de aceptación

Existen herramientas que graban la ejecución y permiten reproducirla luego simulando a un usuario humano, como **Selenium IDE**, y otras más sofisticadas, que permiten programar esa interacción con algunas lenguajes de programación, como **Selenium** Webdriver.

Cobertura

Llamamos **cobertura** al grado en que los casos de pruebas de un programa llegan a recorrer dicho programa al ejecutar las pruebas.

Se usa como una medida, aunque no la única, de la **calidad de las pruebas**.

Se ha definido una gran cantidad de **métricas de cobertura** y en todas ellas se expresa el **porcentaje de cobertura**: cobertura de sentencias, de ramas o de decisiones, de condiciones, de trayectorias, de invocaciones, etc.

Si bien lograr un 100% parece ser a priori una buena meta, no es así teniendo en cuenta los costos frente a los beneficios que podemos obtener. Habitualmente, un **objetivo razonable es llegar a un 80% o 90%** en métricas tales como las de sentencias, ramas y combinada de ramas y condiciones.

Hay muchas **herramientas de cobertura** disponibles, que en general se pueden vincular a los entornos de desarrollo. Entre ellas, destacan *Cobertura*, *Emma*, *JaCoCo*, *EclEmma*, *Clover*, *SimpleCov*, y muchas más.

Pruebas automatizadas: quién las desarrolla

Pruebas unitarias
Son pruebas de programador, así que sin duda las deben escribir los programadores y ejecutarlas ellos mismos (o un proceso automatizado de integración).

Pruebas de integración
Son pruebas de programador: conviene que las escriban los programadores y que las ejecuten sobre servidores de integración.

Pruebas de comportamiento
Están a mitad de camino entre pruebas de programadores y pruebas de *testers*. Por lo tanto, cualquiera de los dos roles podría desarrollarlas. En algunos enfoques metodológicos como BDD, SBE o ATDD se espera que estas pruebas salgan de talleres multidisciplinarios.

Pruebas de aceptación
En principio, se pretende que las escriban usuarios o analistas de negocio. Si esto no se consiguiera, las podrían escribir los *testers* o los desarrolladores, pero es imperativo validarlas con los usuarios para no construir algo diferente a lo que ellos desean.

Pruebas de software: cuándo se llevan a cabo

- El primer ciclo de vida del desarrollo de software suponía que las pruebas se ejecutaban íntegramente **como última etapa del desarrollo** de un proyecto, justo antes de la puesta en producción. En la práctica, esto nunca funcionó del todo así: como a medida que se terminaba de desarrollar una funcionalidad ya se la podía probar, se adelantaba trabajo probando antes.
- En los años 1990, con el auge de los procesos iterativos, se incorporó la idea de probar **al final de cada iteración**.
- En los años 2000, los métodos ágiles abogaron por la **prueba más o menos continua**, automatizando todo lo posible y haciendo que las pruebas guiaran el proceso de desarrollo.
- Hoy en día, con el surgimiento de ciclos de flujo continuo, como el de entrega continua (CD), **las pruebas son continuas** a lo largo de todo el desarrollo.

Pruebas unitarias (y técnicas en general): cómo se diseñan

Analicemos la construcción de **pruebas de caja negra**. La clave aquí es seleccionar los casos de prueba con la mayor probabilidad de encontrar errores.

Lo que podríamos hacer es **establecer precondiciones y postcondiciones**. Con las precondiciones podemos obtener casos de excepción y con las postcondiciones las posibles salidas. Cada postcondición debería al menos definir una prueba.

Ejemplo: supongamos que lo que debemos probar es una función que retorna el factorial de un entero *n* pasado como argumento, que no sea mayor que 30.

Se prueban valores de los casos principales, casos borde y excepcionales.

Si n = ...	Deberíamos obtener
-2	Una excepción
0	1
1	1
5	120
30	265252859812191058636308480000000
100	Una excepción

TDD (*Test-Driven Development* o desarrollo guiado por pruebas)

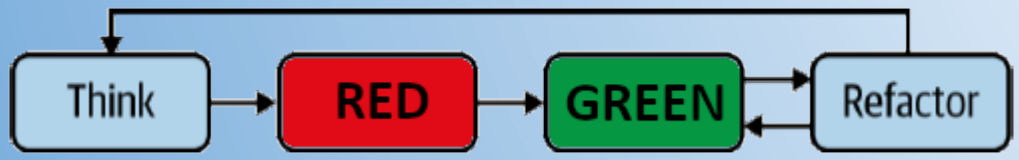
Fue la primera práctica basada en automatización de pruebas bien soportada por herramientas. Fue presentada por Kent Beck en el marco de *Extreme Programming* y enseguida surgieron *frameworks* para llevarla a la práctica.

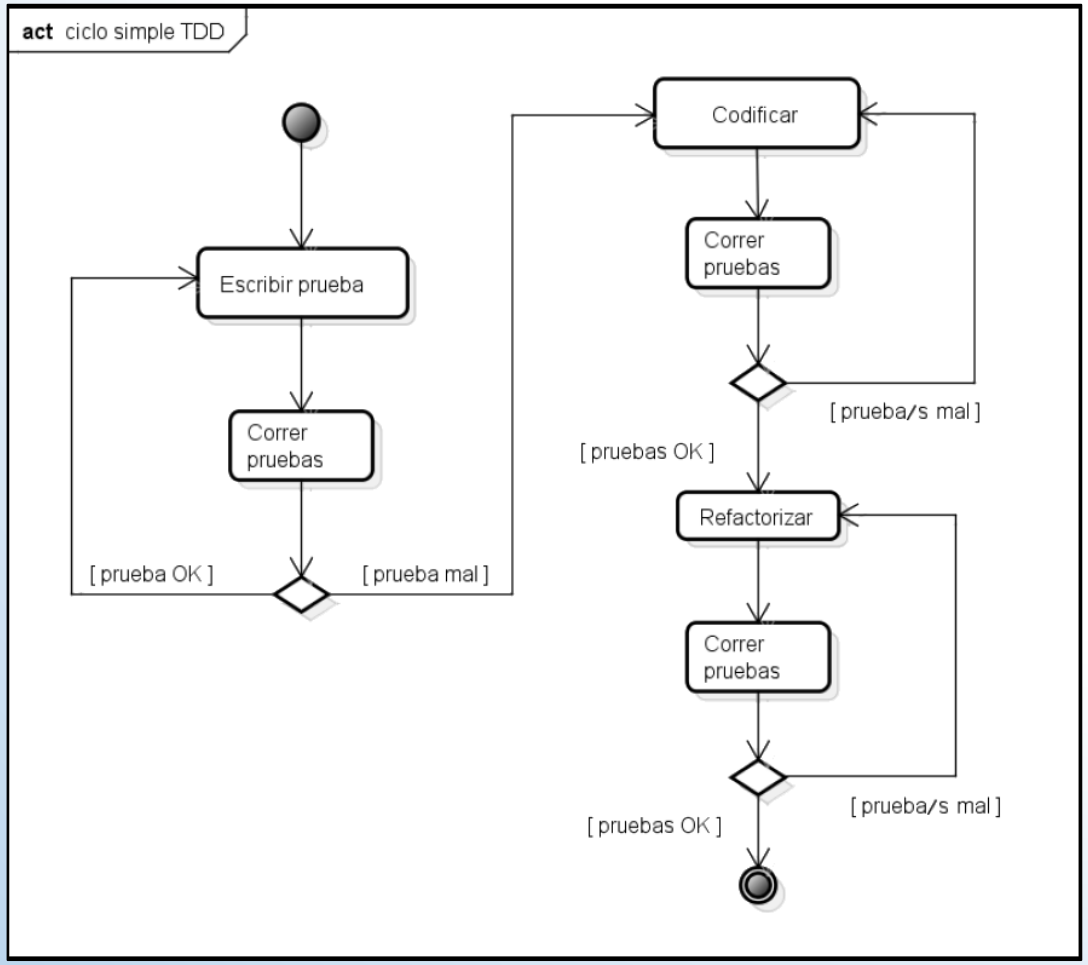
Básicamente, TDD incluye tres subprácticas:

- **Automatización**: las pruebas del programa deben ser hechas en código (generalmente siguiendo las reglas "*arrange, act, assert*" o "*given, when, then*"), y con la sola ejecución del código de pruebas debemos saber si lo que estamos probando funciona bien o mal.
- **Test-First**: las pruebas se escriben antes del propio código a probar.
- **Refactorización posterior**: para mantener la calidad del código, se lo cambia sin cambiar la funcionalidad, manteniendo las pruebas como reaseguro.

Aclaración: TDD denomina “pruebas” a las pruebas unitarias.

El ciclo de TDD





Fuente: Adaptado de Fontela, C. (2018). *Pruebas de software*

Ventajas de TDD

- Las pruebas en código sirven como **documentación** del uso esperado de lo que se está probando, sin ambigüedades.
- Las pruebas escritas con anterioridad **ayudan a entender mejor** lo que se está por desarrollar.
- Las pruebas escritas con anterioridad suelen incluir más casos de **pruebas negativas** que las que escribimos a posteriori.
- Escribir las pruebas antes del código a probar minimiza el condicionamiento del autor por lo ya construido. También da **más confianza al programador** sobre el hecho de que el código que escribe siempre funciona.
- Escribir las pruebas antes del código a probar permite especificar el comportamiento sin restringirse a una única implementación.
- La automatización permite independizarse del factor humano y **facilita la repetición** de las mismas pruebas a un costo menor.
- La refactorización constante **facilita el mantenimiento** de un buen diseño a pesar de los cambios que, en caso de no hacerla, lo degradarían.

Fuente: Adaptado de Fontela, C. (2018). *Pruebas de software*

Lectura para reflexionar...

CARGO CULT

Test-Driven Debaclement

"Oh yeah, TDD." Alisa scowls. "I tried that once. What a disaster."

"What happened?" you ask. TDD has worked well for you, but some things *have* been a challenge to figure out. Maybe you can share some tips.

"Okay, so TDD is about codifying your spec with tests, right?" Alisa explains, with a hint of condescension. "You start out by figuring out what you want your code to do, then write all the tests so that the code is fully specified. Then you write code until the tests pass."

"But that's just stupid!" she rants. "Forcing a spec up front requires you to make decisions before you fully understand the problem. Now you have all these tests that are hard to change, so you're invested in the solution and won't look for better options. Even if you do decide to change, all those tests lock in your implementation so it's nearly impossible to change without redoing all your work. It's ridiculous!"

"I…I don't think that's TDD," you stammer. "That sounds awful. You're supposed to work in small steps, not write all the tests up front. And you're supposed to test behavior, not implementation."

"No, you're wrong," Alisa says firmly. "TDD is test-first development. Write the tests, then write the code. And it sucks. I know, I've tried it."

Fuente: Shore, J. et al. (2022). *The Art of Agile Development, 2da. ed.*

Ejercicio Nro. 4

Usando TDD, desarrolle la clase TaTeTi para que pueda ser usada desde Main.java:

*Main.java

```java
package tateti;

public class Main {
    public static final java.util.Scanner teclado = new java.util.Scanner(System.in);
    public static final java.io.PrintStream pantalla = new java.io.PrintStream(System.out);

    public static void main(String[] args) {

        TaTeTi juego = new TaTeTi(3);
        String result;

        do {
            pantalla.print("Fila: ");
            int fila = Integer.parseInt(teclado.nextLine());

            pantalla.print("Columna: ");
            int columna = Integer.parseInt(teclado.nextLine());

            result = juego.jugar(fila, columna);

            pantalla.println('\n' + juego.generarTablero());

        } while(result.equals("Aun no hay un ganador!"));

        pantalla.println(result);
    }
}
```

JUnit

Java

https://github.com/dcorsi/algo3/tree/main/tateti

Solución

1) Crear el proyecto `tateti` (*Maven Project*) con las clases `Main` y `TaTeTi` (esta vacía).
2) Configurar *JUnit* en el proyecto anterior:
3) Crear la prueba `TaTeTiTest`:

1. **File → New → JUnit Test Case**
2. En la pantalla **JUnit Test Case**:
   - Tildar **New JUnit 4 test**
   - Name: `TaTeTiTest`
   - Class under test: click en **Browse...** y seleccionar la clase `TaTeTi`
   - Click en **Finish**
3. Se genera automáticamente una función de prueba que siempre falla:

```
@Test
public void test() {
    fail("Not yet implemented");
}
```

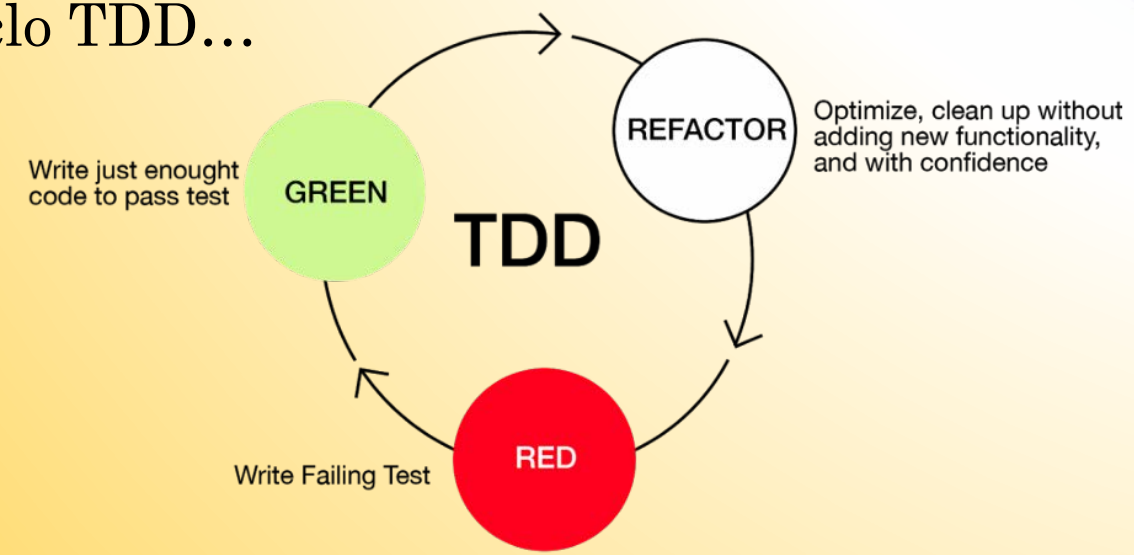
1. Abrir el archivo `pom.xml`
2. Seleccionar la pestaña **Dependencies**
3. En el panel de la izquierda, click en **Add...**
4. En la pantalla **Select Dependency**:
   - Group Id: `junit`
   - Artifact Id: `junit`
   - Version: `4.13.2`
   - Scope: `test`
   - Click en **OK**
5. En la solapa **Overview**:
   - **Properties → Create...**
     - Name: `maven.compiler.source`
     - Value: `18` (o la versión correspondiente al jdk)
   - **Properties → Create...**
     - Name: `maven.compiler.target`
     - Value: `18` (o la versión correspondiente al jdk)
6. **File → Save**

Fuente: https://algo3.super.site/entorno-de-desarrollo/tutorial-junit-4-en-eclipse

4) Ejecutar la prueba:

- En **Package Explorer** seleccionar el proyecto
- **Run → Run As → JUnit test**
- Aparece la pestaña **JUnit** con el resultado de las pruebas

5) Borrar la prueba e iniciar el ciclo TDD…



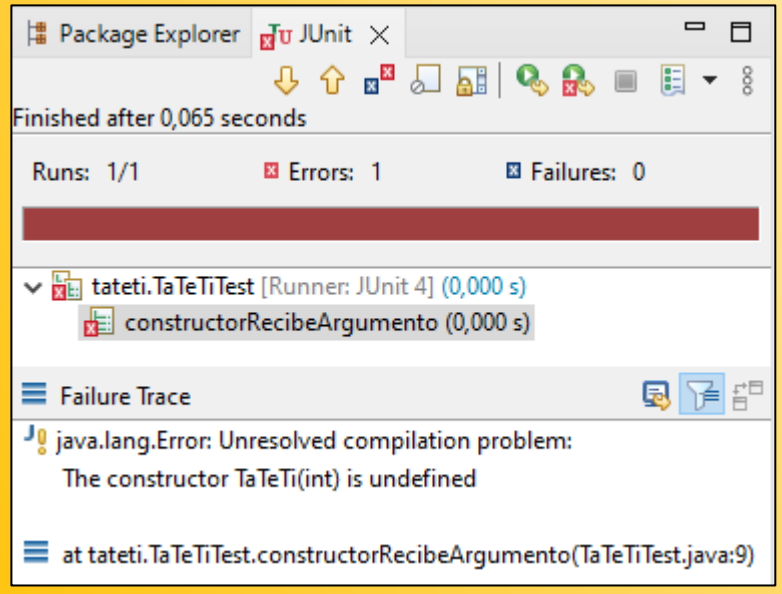
JUnit

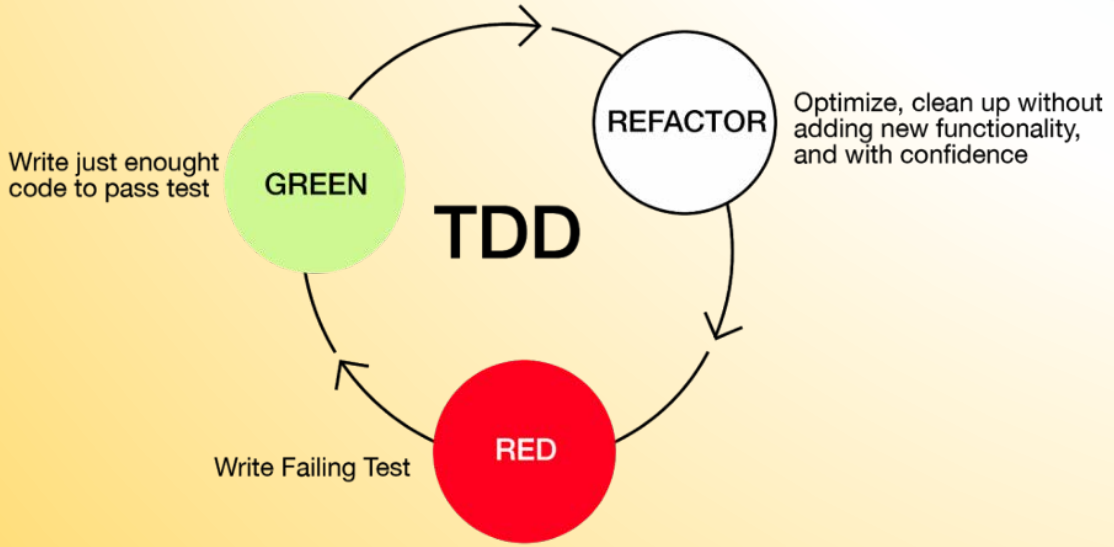
Java

6) Escribir la primera prueba que falle:

```java
package tateti;

import org.junit.Test;

public class TaTeTiTest {

    @Test
    public void constructorRecibeArgumento() {
        TaTeTi tateti = new TaTeTi(3);
    }
}
```

7) Ejecutar la prueba:





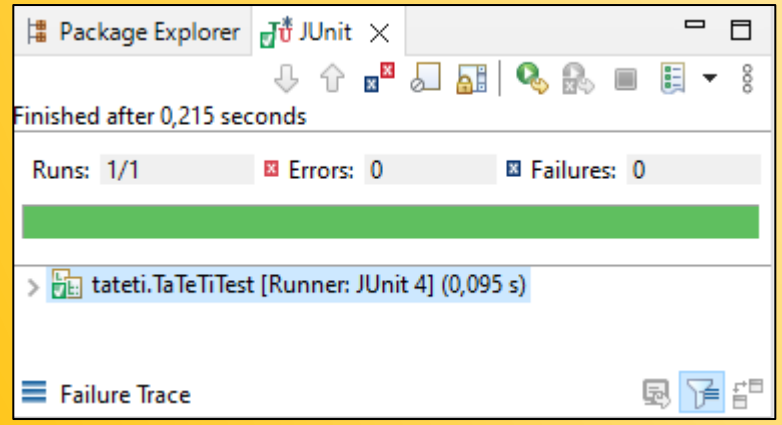
JUnit

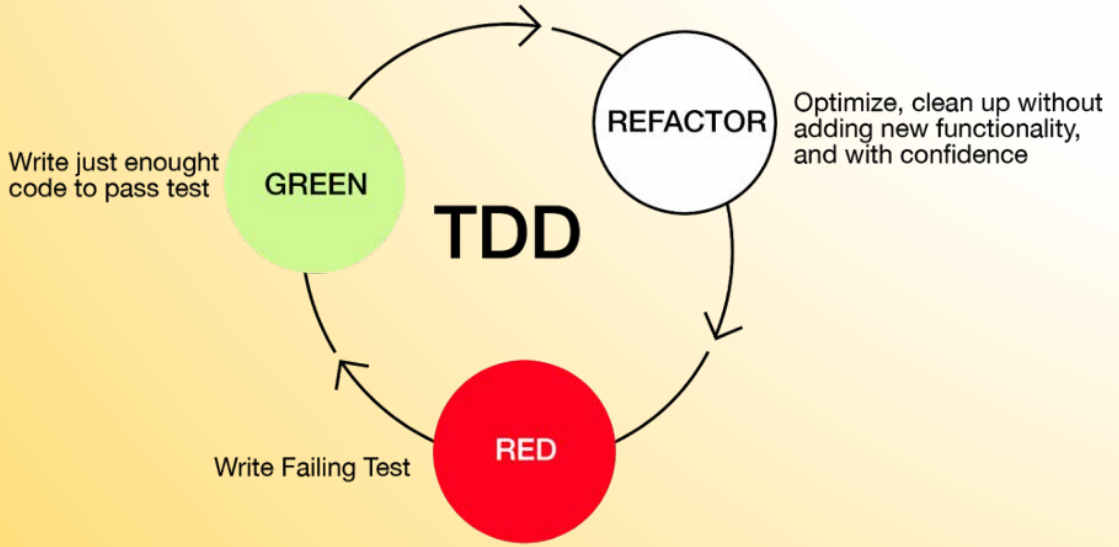
Java

8) Agregar el constructor en la clase `TaTeTi`:

```
1  package tateti;
2
3  public class TaTeTi {
4
5      public TaTeTi(int tamanoTablero) {
6      }
7  }
8
```

9) Ejecutar la prueba:





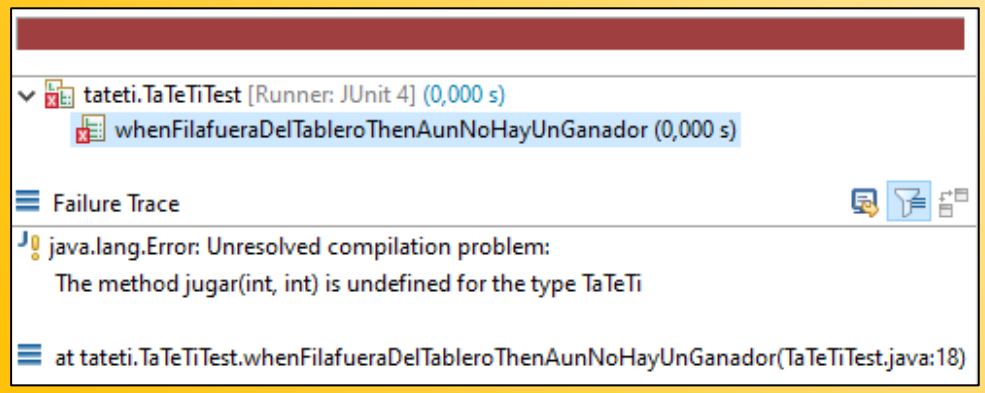
JUnit

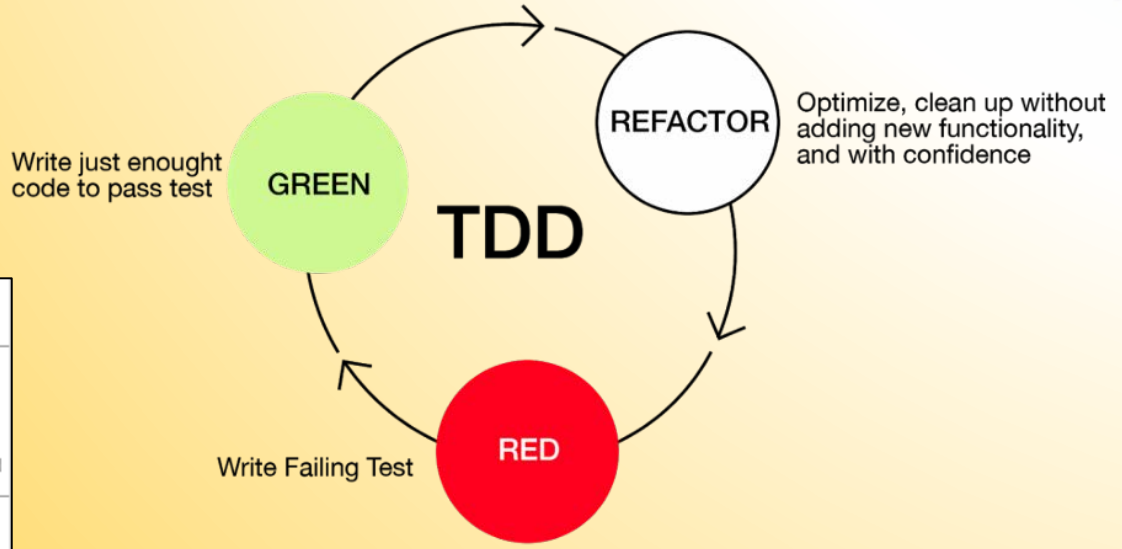
Java

10) Escribir la nueva prueba que falle (refactorizando la clase `TaTeTiTest`):

```java
package tateti;

import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;

public class TaTeTiTest {

    private TaTeTi juego;

    @Before
    public void arrange() {
        juego = new TaTeTi(3);
    }

    @Test
    public void whenFilafueraDelTableroThenAunNoHayUnGanador() {
     assertEquals("Aun no hay un ganador!", juego.jugar(4,1));
    }
}
```

11) Ejecutar la prueba:





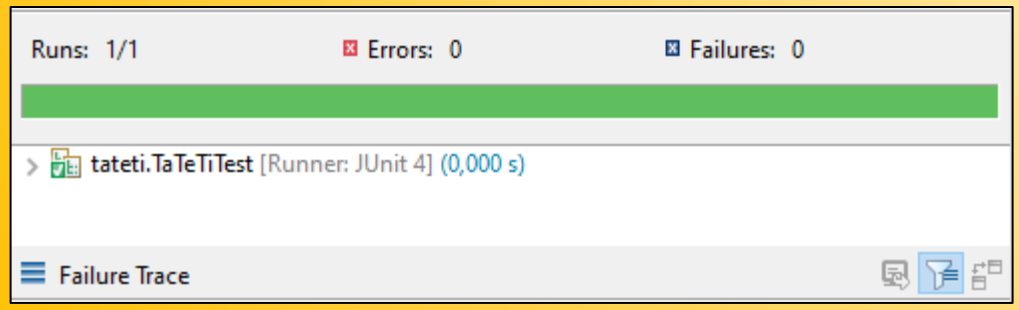
JUnit

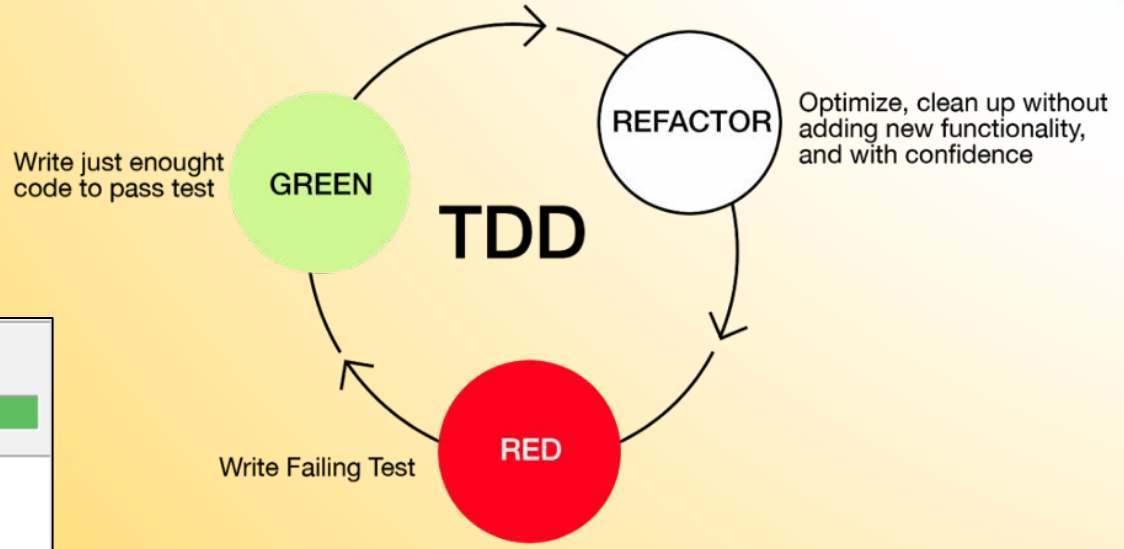
Java

12) Agregar el método `jugar` con la validación de la fila (para esta debe guardarse el tamaño del tablero en un atributo del juego):

```java
package tateti;

public class TaTeTi {

    private final int tamanoTablero;

    public TaTeTi(int tamanoTablero) {
        this.tamanoTablero = tamanoTablero;
    }

    public String jugar(int fila, int columna) {
        if (fila < 1 || fila > tamanoTablero)
        return "Aun no hay un ganador!";
        else return "";
    }
}
```

13) Ejecutar la prueba:

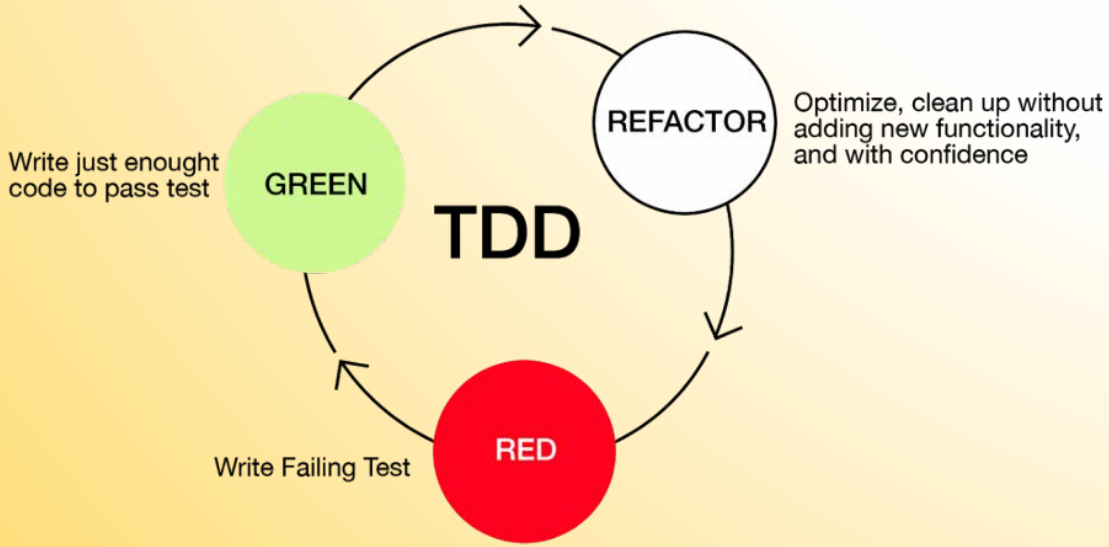




JUnit

Java

14) Repitiendo el ciclo TDD, finalizar el juego…

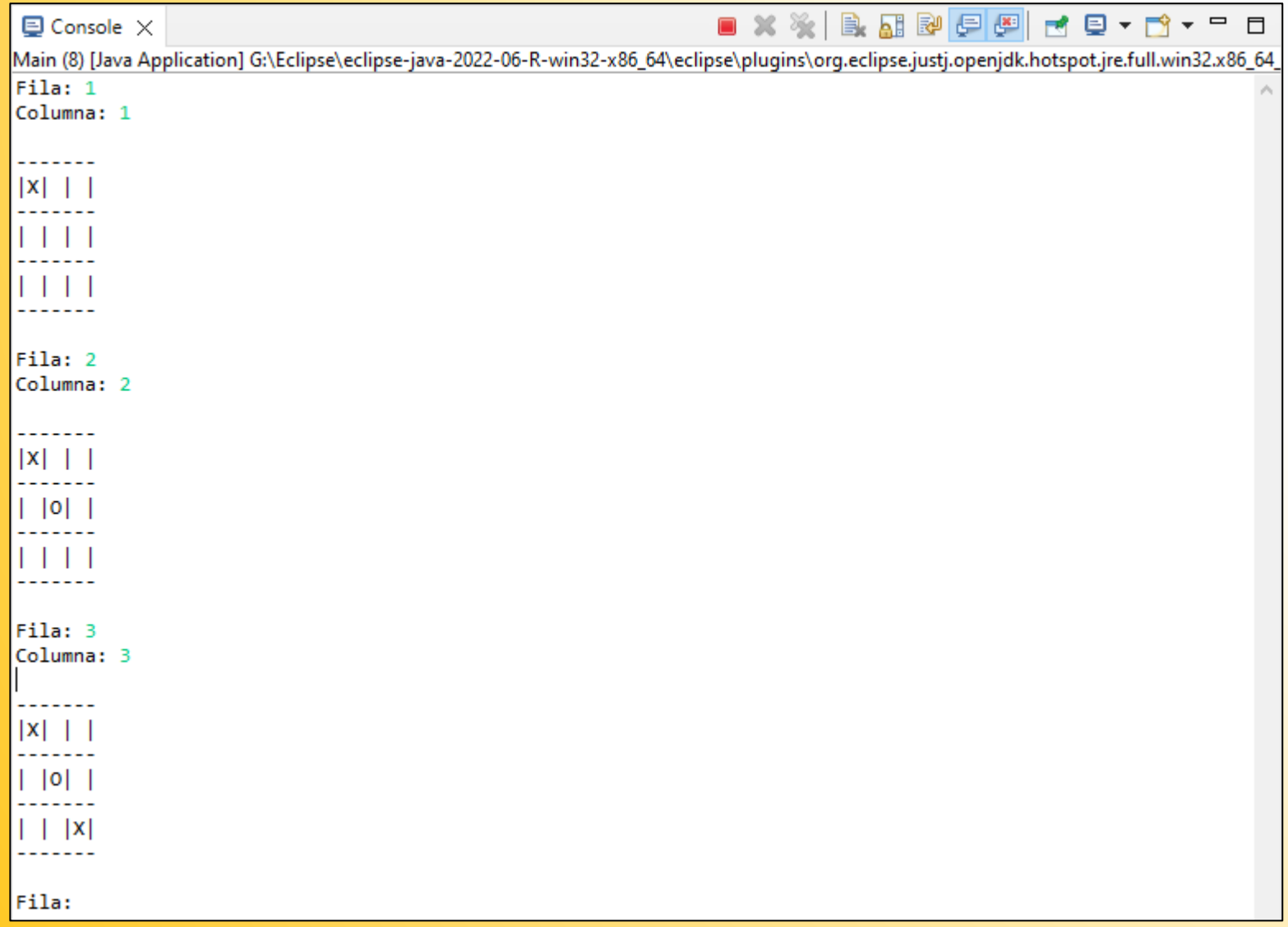


JUnit

Java

75.07 / 95.02 ALGORITMOS Y PROGRAMACIÓN III

.UBA fiuba
FACULTAD DE INGENIERÍA



JUnit

Java