Pruebas de Software, Tests o TDD *Testing Driven Development*

TDD Cómo y porqué How and Why?

Disciplina del Desarrollo Dirigido por Pruebas o TDD Discipline of Development Directed by Testing or TDD

Las excusas abundan, es simplemente demasiado fácil ceder y dejar de hacerlo.

Excuses abound, it is simply too easy to give in and not do it.

Desde los clásicos

"en mi empresa no me lo permiten"

" in my company they don't allow me"

y *"mi jefe dice que me pagan por escribir funcionalidad, no pruebas"*

"my boss says they pay me to write functionality, not proof, not TDD"

hasta los no tan frecuentes even the not so frequent

"las pruebas son trabajo de QA, no mio"

o *"esta porquería es una mi3rd@, ¡es imposible de probar!"*.

No me compensa tanto esfuerzo, he perdido muuuucho tiempo, voy más rápido si no hago Test.

Índice de contenido

¿Qué es TDD?

Es simple de definir, pero su definición parece ir en contra del sentido común. Es sencilla de explicar, pero difícil de llevar a cabo.	It is simple to define, but its definition seems to go against common sense. It is simple to explain, but difficult to carry out.

Y una vez que superas la **resistencia** intelectual inicial y lo entiendes, es difícil de dominar.

If you win the initial intellectual resistance and understand it, you will generalize it and you will not be able to think another way.

Definición

Es una disciplina que promueve el desarrollo de software

- con altos niveles de **calidad**,
- simplicidad de diseño y productividad del programador,
- mediante la utilización de una amplia gama de tipos de **pruebas automáticas** a lo largo de todo el ciclo de vida del software.
- El principio fundamental es que las pruebas **se escriben antes** que el software de producción
- y estas constituyen la especificación objetiva del mismo.

La segunda parte de la definición viene con el primer **"qué me estás contando"** para muchos:

Las pruebas se deben escribir **antes** que el ***software*** **mismo.** The tests/TDD must be written before the software itself.

La primera impresión de muchos es "¿eh?, ¿y cómo demonios escribo una prueba para software que **todavía no existe**?".

Ejemplo: **Imagina que estás haciendo una calculadora.**

En tu lista de **Requistos** de usuario tienes:

R5: Realizar la operación de suma.
R5.1. Solo sumaremos dos números enteros. Nunca más de dos. No sumamos cifras decimales.

Recuerda que

primero debemos hacer el Test

y luego haremos el código.

En el proceso de reflexión para hacer el test ... Debes preguntarte

- 1 - ¿Qué **entradas** necesita?
- 2 - ¿Qué **salidas** me devuelve?

–

Empieza con lo más sencillo. Luego pasarás a cosas más **complejas**: **Elige** por ejemplo dos valores de **entrada**. Y para esas entradas calcula el **resultado**. **Sumar** 2 +3 El Resultado debe ser 5 Y normalmente tenemos un código que funciona y da 5. a=2 a+3 resultaría devolvería 5	Código para lo mismo, pero más **complejo** a=2 b=3 a+b resultaría 5

Y etc, seguiríamos haciendo código **más** **complejo**. And ... we would continue to make more complex code.

Cada vez que vemos el **resultado** que **esperamos** aparecer en la pantalla, aumenta nuestra **autoconfianza**, lo que nos motiva a seguir aprendiendo, a seguir programando.

Este podría tal vez ser el **ejemplo** más **básico** de TDD.

Sin embargo,

una vez que tomamos mayor confianza en nuestro dominio del lenguaje o la programación misma,

comenzamos a escribir **cantidades** cada vez **mayores** de **código** **entre una** **comprobación**

y la siguiente del **resultado**.

Como "sabemos" -en realidad, **creemos**- que nuestro código "esta bien", comenzamos a "**optimizar el tiempo**" escribiendo más y más código de una vez.

Al poco tiempo, **nos olvidamos** de estas primeras experiencias, incluso tachándolas como "cosas de **novatos**".

Aprendiendo TDD

Nos conseguimos una **copia** de

- JUnit, Nunit, (jazmin, expect, Karma para javascritp)
- o el **framework** de moda para nuestro **lenguaje** de elección y comenzamos a seguir el **tutorial** que seguramente encontraremos en el sitio de este último. Los más afortunados probablemente tendrán **integrada** la funcionalidad directamente en su IDE.
-

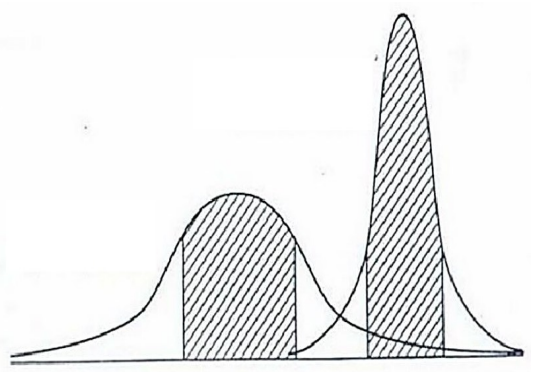
A partir de aquí, estamos en la parte **sencilla** de nuestra curva de aprendizaje.

En los próximos días comenzarás a producir **grandes cantidades de pruebas** y no tardarás en sentirte **cómodo o cómoda** con el proceso.

Esto es lo más lejos que la mayoría llegamos en la **curva** y es aquí justamente donde **comienzan** los **problemas iniciales.**

Conforme comenzamos a intentar escribir pruebas para **proyectos** más **complejos** nos topamos con varios **obstáculos** en el camino:

- Las pruebas se tornan **difíciles** de escribir, por lo que sentimos una **desaceleración** importante.
- Corren **lentamente**, lo que nos volvemos **Reacios** a ejecutarlas frecuentemente.
- Son **frágiles**, por lo que **cambios** aparentemente **sin importancia** en el código provocan que un **montón** de **pruebas fallen**.
- **Mantenerlas** en forma y funcionando se vuelve **complejo** y consume **tiempo**.



Finalmente nos damos por vencido/a y **abandonamos** completamente nuestras mejores intenciones y **pensamos** "Simplemente no vale la pena".

- Estamos en la parte más pronunciada de nuestra curva de aprendizaje. Tal vez estamos produciendo muchas pruebas y logrando lo que buscamos. Sin embargo el **esfuerzo** para escribir/mantener estas mismas **parece desproporcionado**.

- Sin embargo, como cualquier otra **habilidad que valga la pena adquirir,**
  - si en lugar de rendirnos **seguimos adelante**,
  - eventualmente aprenderemos a cruzar a la parte de nuestra gráfica donde la pendiente de la curva se **invierte y baja**
  - y comenzamos a escribir pruebas más efectivas con un **menor esfuerzo / gracias a la perseverancia.**

Aprender a escribir bien y de mantener las pruebas lleva tiempo y práctica.

(Learn to write well and keep the tests It takes time and practice.)

Las reglas de TDD

Robert C. Martin, es una de las autoridades en TDD.

Robert C. Martin, **is one of the authorities in TDD.**

En varias ocasiones ha **descrito** el **proceso** en base a **tres** simples **reglas:**

Three simple rules:

1. No está permitido escribir ningún código de producción **sin tene**r una **prueba** que **falle.** (a test that fails)
2. No está permitido escribir más código de prueba que el necesario para **fallar** (y no compilar es fallar).
3. No está permitido escribir más código de producción que el necesario para **pasar** su prueba unitaria. (pass de test)

Esto significa que **antes** de poder escribir cualquier código, **debemos pensar en una prueba apropiada para é**l.

This means that before we can write any code, we must think of an appropriate test for it.

Esto quiere decir que **no puedes**

escribir el método limpiarNombre (...)

sin antes haber realizado su test (en el paquete/carpeta/componente de **test** correspondiente)

testLimpiarNombre o limpiarNombreTest

before taking the test, even before that doing the class

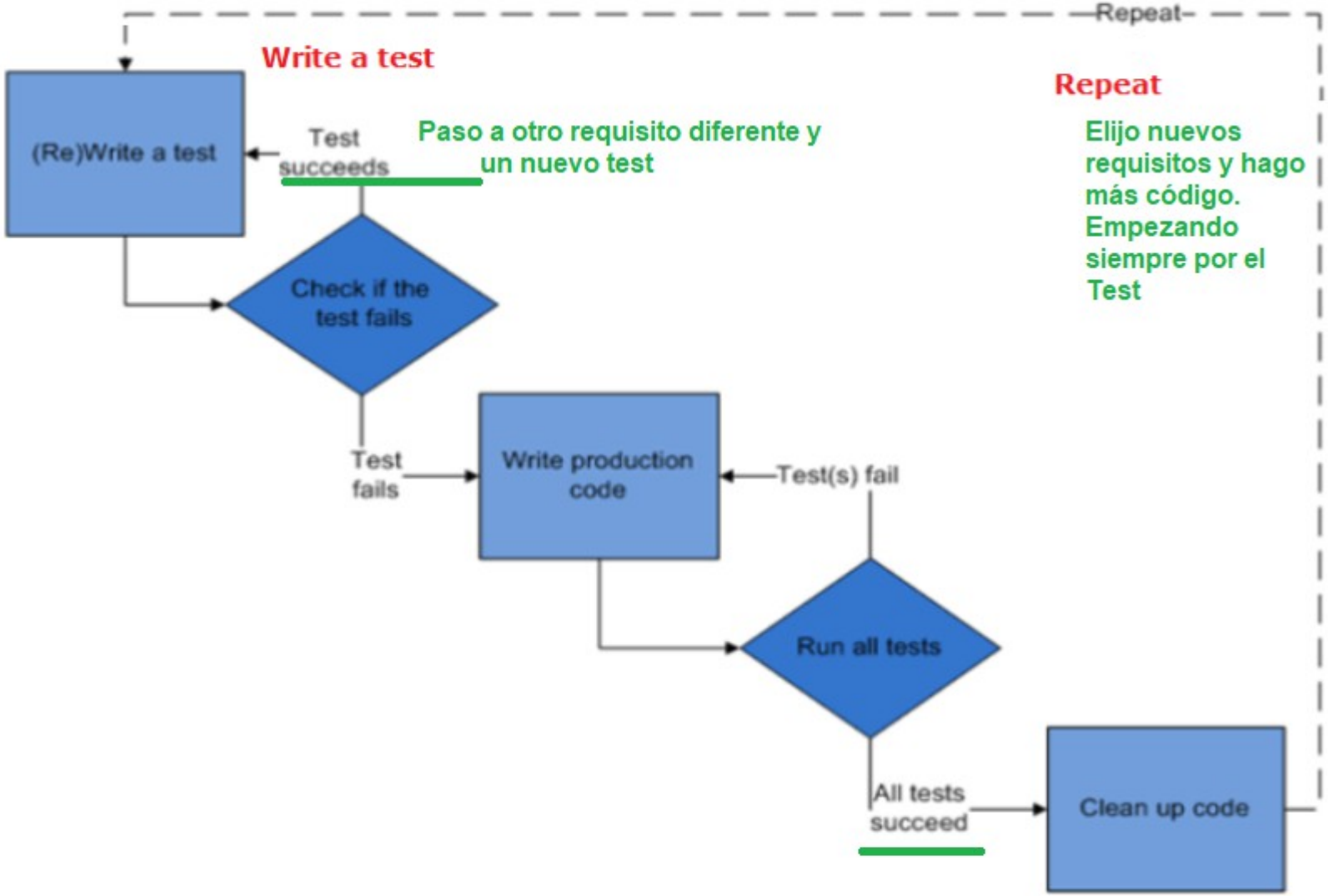
Pero por la regla número **dos**, i tampoco podemos escribir mucho de dicha prueba ! En realidad, debemos detenernos en el momento en que la prueba **falla** al compilar o falla un assert y comenzar a escribir código de producción.

Pero por la regla número **tres**, tan pronto como la prueba **pasa** (o compila, según el caso), debemos **dejar** de escribir código y continuar escribiendo la **prueba** o pasar a la siguiente prueba.

Ciclo rojo,verde y refactor Red, green and refactor cycle

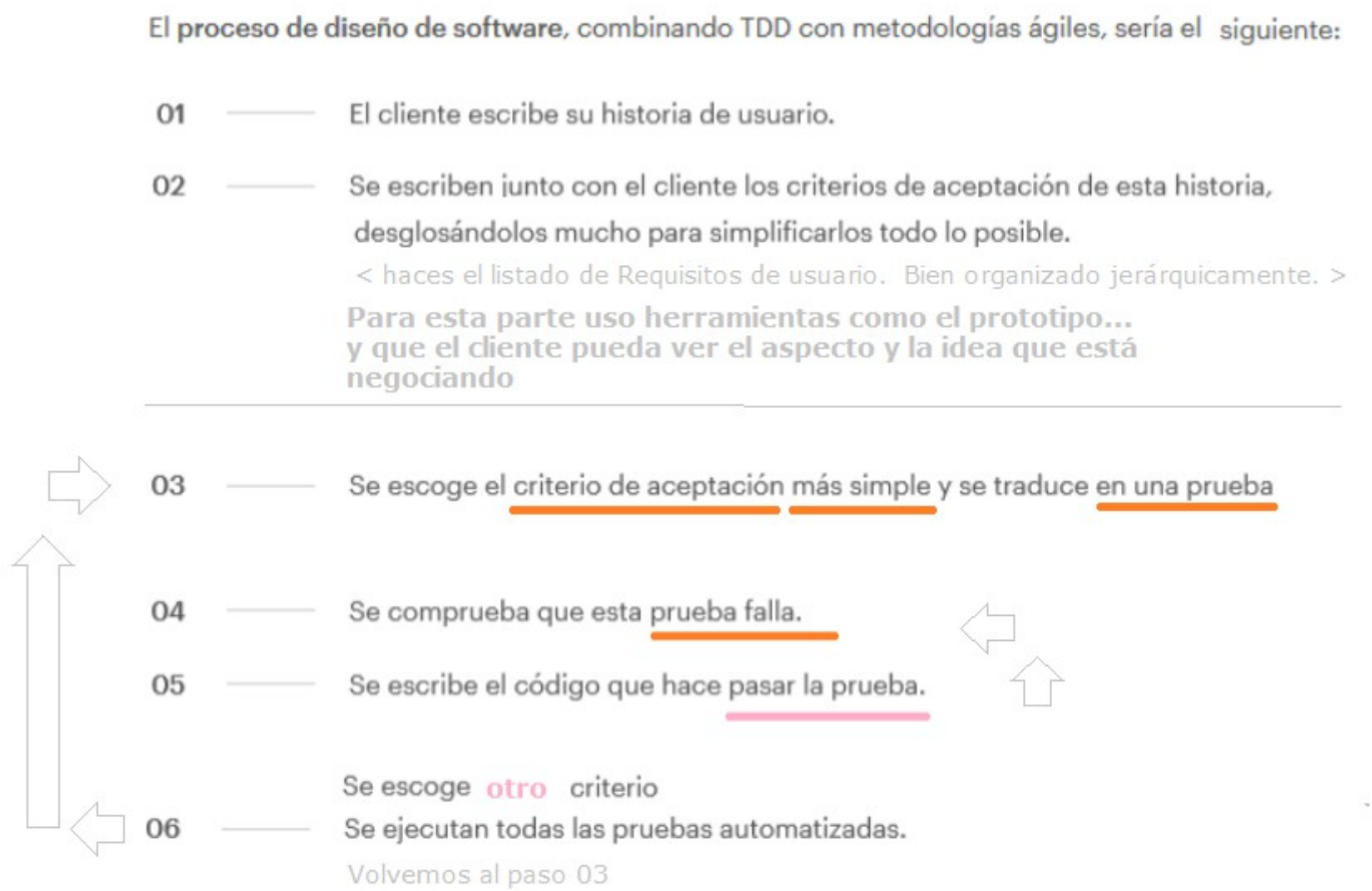
Ciclo **Rojo**: **Hacer que la prueba falle.** the test that fails

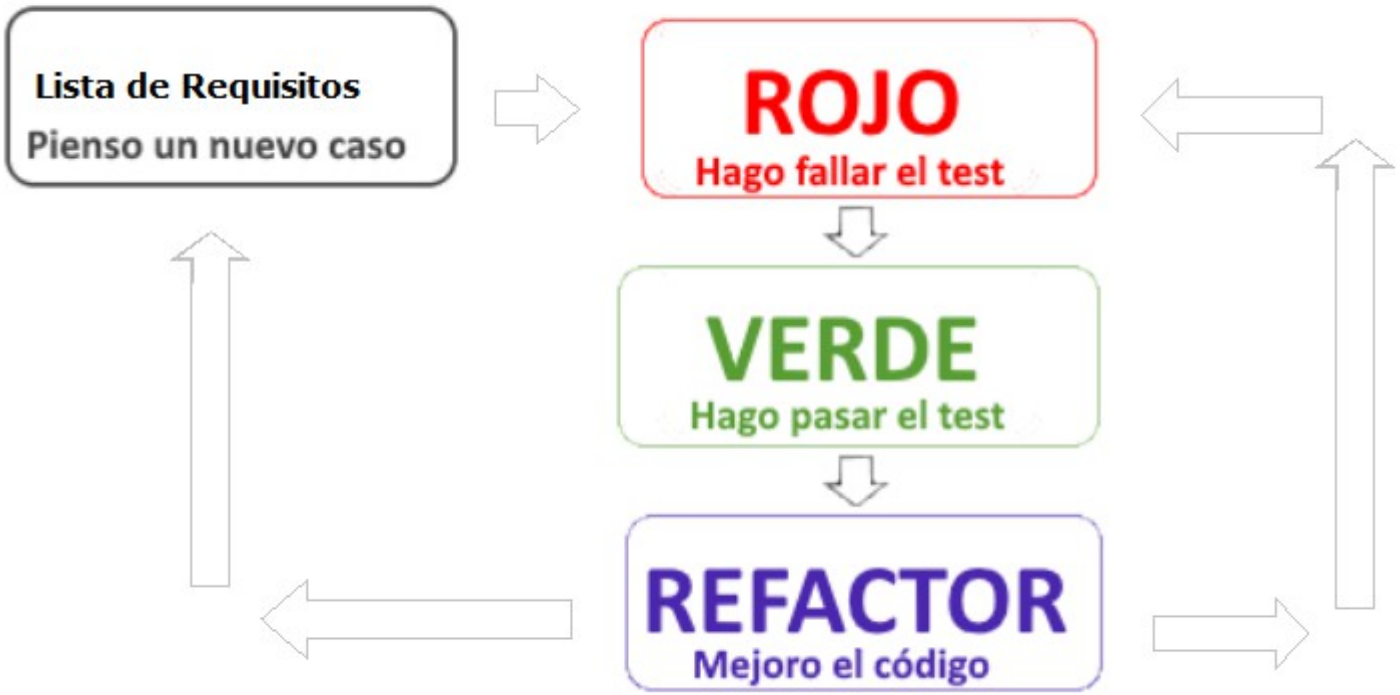
Ciclo **Verde**: Hacer que la prueba pase. The test pass.



Fuente: https://www.paradigmadigital.com/dev/tdd-como-metodologia-de-diseno-de-software/

Proceso de diseño de software, combinando TDD con metodologías ágiles:




Lista de Requisitos
Pienso un nuevo caso
ROJO
Hago fallar el test
VERDE
Hago pasar el test
REFACTOR
Mejoro el código

Tipos


Tipos de Pruebas
1. Integración
2. de Sistema
3.Regresión
Tras un cambio
4. Funcional
Requisitos
5. deCapacidad yRendimiento
Cargas y tiempos de respuesta
6. Integración
Ascendente
Descendente
7. de Seguridad
Accesos autorizados
8. de uso de Recursos
eficiencia
9. Manuales y automáticas
Automáticas
Unitary
Funcional y de Regresión
Integración
Carga y Rendimiento
Aceptación
10. Usuario
Fácil de usar
11. de Aceptación
Está el producto correcto

A parte de test unitarios, ¿qué otros tipos de test existen?

Ya hemos mencionado algunos tipos de test: unitarios y de carga. Hay varios tipos de test, incluso se podrían añadir alguno más que vuestros sistema requiera:

- **Test unitarios**: prueban una funcionalidad única y se basan en el principio de responsabilidad única (la S de los principios de diseño SOLID)
- **Integración**: prueban la conexión entre componentes, sería el siguiente paso a los test unitarios.
- **Funcionales (o Sistema)**: prueban la integración de todos los componentes que desarrollan una funcionalidad concreta (por ejemplo, la automatización de pruebas con Selenium serían test funcionales).
- **Aceptación de Usuarios**: Pruebas definidas por el Product Owner basadas en ejemplos (BDD con Cucumber). Para más detalle [Link a entrada blog cucumber].
- **Regresión**: Prueban que los test unitarios y funcionales siguen funcionando a lo largo del tiempo (se pueden lanzar tanto de forma manual como en sistemas de Integración Continua).
- **Carga**: Prueban la eficiencia del código.

Test unitarios

Deben cumplir las siguientes **características:**

Principio FIRST

- **F**ast: Rápida ejecución.
- **I**solated: Independiente de otros test.
- **R**epeatable: Se puede repetir en el tiempo.
- **S**elf-Validating: Cada test debe poder validar si es correcto o no a sí mismo.
- **T**imely: ¿Cuándo se deben desarrollar los test? ¿Antes o después de que esté todo implementado? Sabemos que cuesta hacer primero los test y después la implementación (TDD: Test-driven development), pero es lo suyo para centrarnos en lo que realmente se desea implementar.

Además podemos añadir estos puntos (más):

- Sólo pruebas los **métodos públicos** de cada clase o componente.
- **No** se debe hacer uso de las **dependencias** de la clase a probar. Esto quizás es discutible porque en algunos casos donde la dependencias son clases de utilidades y se puede ser menos estricto. Se recomienda siempre aplicar el sentido común.
- Un **test** **no debe implementar ninguna lógica de negocio** (nada de if...else...for...etc)

Los tests unitarios tienen la siguiente **estructura**:

- Preparación de datos de entrada.
- Ejecución del test.
- Comprobación del test (assert). No debe haber más de 1 assert en cada test.

Ejemplo para las reglas de TDD

Se verá mejor con un pequeño ejemplo. Escribimos suficiente de nuestra primera prueba para que **falle**

https://es.slideshare.net/aleherse/qu-es-tdd-y-algunos-consejos-para-iniciados-61455458

PASO 0

Elijo un requisito de usuarios sobre el que trabajar. Voy a hacer la suma de enteros. R5.1 de nuestra lista de Requisitos

PASO 1

Ejemplo java. Tendremos instalado JUnit para java, para puebas, test o TDD-

Crea el paquete de los Test: Importaremos la clase del Código que queremos probar **Calculator**.

```
public class TestCalculator{


@Test
public static int testAdd (){
        Calculator myCalc = new Calculator();
        result= myCalc.add ( 2,  3);
```

** Sabemos que la suma debe ser 5, por lo que **comparamos** con 5. La **comparación** en Junit se hace con **assert**.

```
  Assert.assertEquals( result, 5 );
} }
```

PASO 2

Para probar este test
Ahora hay que hacer la clase de la App
Creamos paquete de la App y ahí Creamos la clase.

```
public class Calculator{
    public static int add (int opera1, int opera2){
      return 100;
}}
```

PASO 3

Si probamos el test como nuestro test compara con 5

Y el método add devuelve 100

Va a Fallar

PASO 4

Ahora **modificamos** el **código**

… Si hago un cambio, inmediatamente después vuelvo a **lanzar** el **Test** para probarlo.

Como el test falla … Modificamos el código a ver si corregimos el error.

```
public class Calculator{
    public static int add (int opera1, int opera2){
      return operator1+operator2
}}
```

PASO 5

Lanzamos el test **a ver si** funciona o si vuelve a fallar.

Si falla

repetimos el paso 4 hasta que logremos que el Test funcione.

PASO 6

Cuando funcione volvemos al paso 0 y

elegimos un nuevo Requisito sobre el que trabajar.