# ✅ Test, Software que funciona

El código se escribe para resolver un problema. El test es su garantía.

> *"Nunca pidas permiso para refactorizar. Nunca pidas permiso para escribir pruebas. Haces estas cosas porque SABES que son la mejor manera de ir rápido."*
>
> -- ✍ **Robert C. Martin**

## 😉 Las excusas

- ✕ *Las pruebas son inútiles.*
- ✕ *Requieren demasiado esfuerzo.*
- ✕ *No dudo de mi código.*
- ✕ *Nadie me las pide.*
- ✕ *Nadie las valora.*

## 😉 Los motivos

- ✔ **Las pruebas reducen errores.**
- ✔ **Son menos costosas cuanto más pronto se incluyan.**
- ✔ **Si tu código es bueno, al incluir tests será aún mejor.**
- ✔ **Las pruebas te permiten dormir tranquilamente.**
- ✔ **El valor del trabajo bien hecho empieza por uno mismo.**

> *"Los geeks son gente que aman tanto algo que le importan todos sus detalles."*
>
> ✍ Marissa Mayer

# 🔀 Tipos de Pruebas

Hay una prueba para cada situación.

> "*Escribe tests. No demasiados. Principalmente de integración.*"
>
> ✍ Kent C. Dodds

Si empiezo con esta frase es para introducir la idea de que no hay un sólo tipo de test. Y dejar caer que no hay que volverse locos probado código. Es mejor **empezar poco a poco**, pero empezar. Si ya has empezado, entonces **avanzar un poco más**.

En cualquier caso te vendrá bien un repaso de conceptos básicos y nomenclatura.

## Tipos de Pruebas

### 🤘 Manuales -> Programadas

- ✕ Dependemos de las personas
- ✔ Se pueden configurar y lanzar automáticamente

### ⚖ Técnicas -> Funcionales

- ◯ Se puede comprobar el rendimiento, la seguridad, usabilidad...
- ✔ La función del software, su utilidad.

### 🔎 Unitarias -> De integración -> De inicio a fin

- ✔ **unitarias**: Pruebas de caja blanca que verifican una función, una clase o un componente.
- ✔ **de integración**: Pruebas de caja blanca que verifican que varios componentes funcionan bien juntos.
- ✔ **de inicio a fin**: Pruebas de caja negra que replican el comportamiento de un usuario ante un sistema completo.

> Otras: de regresión, de humo, de aceptación...

### ⌚ Después -> Durante -> Antes

- ✔ **Después** o mucho después *legacy*. Es costoso, pero imprescindible para un *refactoring* y muy habitual en un *end to end*
- ✔ **Durante** es aburrido pero necesario para las pruebas de integración, vas probando lo que vas programando.
- ✔ **Antes** El conocido como *TDD* para pruebas unitarias o *BDD* para las de integración. Menos costoso, más divertido y con mucho mejor diseño resultante.

# Filosofía y patrones

## Qué hay que saber para programar tests.

### 1 Mantra

- **El código de prueba no es como el código de producción:** diséñalo para que sea simple, corto, sin abstracciones, agradable de leer. Uno debe mirar una prueba y obtener la intención al instante.

### 2 Siglas y conceptos

- **SUT**: *System (Subject) Under Test*. Lo que se está probando.
- **DOCs**: *Depended On Components*. Lo que se necesita para que funcione el SUT.

### 3 Secciones: Arrange, Act & Assert (AAA Pattern)

- **Arrange**: Prepara y organiza lo que necesitas.
- **Act**: Ejecuta el código y obtén una respuesta.
- **Assert**: Verifica que la respuesta es la esperada.

### 4 Cuestiones: Given, Should, Actual, Expected.

- **Given**: 📃 Texto. Condiciones de la prueba. *(Arrange)*
- **Should**: 📃 Texto. Funcionalidad esperada.
- **Actual**: 🎰 Variable. El resultado obtenido. *(Act)*
- **Expected**: 💰 Variable. La respuesta esperada. *(Assert)*

### 5 Test Doubles: Simuladores para no depender de las dependencias DOC.

- **Dummy**: Datos requeridos para que el SUT funcione, pero que no se usan durante la prueba. *(Carga previa de una base de datos)*
- **Stub**: Un objeto que cumpliendo una interfaz de un DOC tiene una respuesta constante y predeterminada. *(Responder como lo haría un llamada http)*
- **Fake**: Un objeto que realiza una funcionalidad coherente pero simplificada de un DOC. *(Simular una base de datos en memoria)*
- **Spy**: Cuenta las llamadas a una función o método. *(Comprobar que se ejecuta una acción un determinado número de veces)*
- **Mock**: Monitoriza el uso de un objeto y las llamadas a una función junto con sus argumentos. *(Simular un envío de correo completo)*

## 6 Comprobaciones: igualdad, existencia, comparación, pertenencia, excepciones y negación

- **igualdad**: El valor actual es igual al esperado.
- **existencia**: El valor actual existe.
- **comparación**: El valor actual es mayor o menor que el esperado.
- **pertenencia**: El valor actual contiene o está contenido en el esperado.
- **excepciones**: Se espera que una excepción sea lanzada.
- **negación**: Niega cualquiera de los anteriores.

## 7 Consejos generales

- **incorpora herramientas**: Puedes empezar de cero, pero hay muchas ayudas.
- **evita arreglos globales**: Cada prueba deber ser autónoma e independiente.
- **datos realistas en los fakes**: Nada de *foo bar baz asdf*
- **usa etiquetas o códigos**: Útil para buscar resultados o pre filtrar pruebas.
- **public black box**: Prueba los métodos públicos.
- **evita los mocks**: Mejor usa *Stubs* y *Spies*.
- **haz alguna prueba**: Esto no va de todo o nada.

# Comportamiento

BDD Desarrollo por comportamiento.

> *"Si a ti no te apetece probar tu software, lo normal es que a tus usuarios tampoco."*
>
> ✍ Programador desconocido

Cuando empezamos con las pruebas, a veces lo más difícil es determinar qué es lo que vamos a probar. Aquí es dónde el *Behavior Driven Development* nos ayuda. Al final se trata de satisfacer a los usuarios, y eso se hace **comprobando el comportamiento** de nuestro software.

## Pruebas funcionales

Definir la funcionalidad esperada.

### Escenarios

Situación en la que se encuentra el sistema.

### Usuario final

Comportamiento y resultado obtenido por el usuario.

## No sólo e2e

Este esquema mental es muy utilizado en pruebas de alto nivel... Pero

### También integración

### También unitarias

### También TDD

## Historias de usuarios

Este esquema mental se puede usar en cualquier nivel de abstracción para captar la

### Funcionalidad...

### que resuelve un problema ✅

Aunque es original y fundamental en *Agile* lo podemos aplicar pues nos va a ayudar a **documentar y estructurar** las pruebas.

Hablando de estructura; vamos a encontrarnos repetidamente un patrón de tres fases. Puede que tenga representación formal en alguna de estas variantes:

## 1 Rol; 2 Feature; 3 Reason

### 1 Quién; 2 Qué; 3 Por qué

#### 1 **As a []; 2 I want to []; 3 In order to []**

Pero siempre responde a las mismas dudas. Quién usa el software? ¿Qué hace con el? ¿Qué espera de su uso?

Estos son algunos ejemplos formales a distintos niveles de abstracción.

```
FEATURE: have web store with products
As a: visitor
I want to: view, navigate and purchase
In order to: get information and buy from home
```

```
FEATURE: Get the lowest price
As a: customer
I want to: get the lowest price
In order to: choose the cheaper product
```

```
FEATURE: Get the lowest number from array
As a: array
I want to: compare all numbers
In order to: return the lowest
```

## Documentar pruebas

Otra forma de especificar este comportamiento es documentando las pruebas con textos muy *human frindly*

### 1 Given

#### **Dado un escenario**

### 2 When

#### **Cuando realizo una acción**

### 3 Then

#### **Entonces compruebo una expectativa**

La documentación funcional es uno de los valores fundamentales de las pruebas.

## Estructurar pruebas

Por último reaparece la regla de tres al organizar el código dentro de una prueba. Es decir, al estructurar la prueba también seguiremos un esquema repetible que facilite su escritura y comprensión.

En este caso el acrónimo propuesto es el de la *tripe A*

1️⃣ Arrange

**Preparar el escenario**

2️⃣ Act

**Actuar para ejercitar el SUT**

3️⃣ Assert

**Comprobar el resultado**

## Implementar pruebas

Claro que hablar es barato, al final hay que escribir código. Obviamente cada framework de pruebas hacen sus propias lectura de los conceptos anteriores. Pero casi todos acaban usando alguna de estas funciones para los tres actos de la prueba.

1️⃣ **given()**, describe(), beforeAll()...

2️⃣ **when()**, before(), context()...

3️⃣ **then()**, it(), assert()...

## Limpieza de las pruebas

Las pruebas son código, y por tanto deben estar limpias. Pero su objetivo fundamental es explicar claramente un comportamiento. Así que se les permitirá ciertas licencias.

⚠️ No es código de producción.

🆗 Permiten cierta humedad...

🚫 Pero no mal olor

Por ejemplo, nunca permitiremos estos *Malos olores:*

🤢 Comentarios

🤢 Datos mágicos

😡 Datos absurdos

😡 Anidamientos profundos

Pero sí que podremos admitir duplicidades que en código de producción podrían resultar molestas.

## Licencias para humedades

🐫 DRY

**Don´t Repeat Yourself**

🐊 WET

**Write Everything Twice**

🍪 DAMP

**Descriptive And Meaningful Phrases**

## Recomendaciones finales

- Muchas pruebas pequeñas.
- Un fichero, módulo, por prueba.
- Textos *super mega ultra hyper* descriptivos.
- Datos en variables.
- Extrae *callbacks* complejos a funciones.
- Algunos ficheros de utilidad comunes.
- Pero **sin abstracciones complejas.**