🕵️ Pruebas con espías y dobles

Pruebas de sistemas legacy complejos.

> *"Los probadores de software siempre van al cielo; Ya han tenido su parte de infierno."*
>
> -- ✍️ Tester anónimo

Hacer pruebas sobre código heredado es costoso y poco atractivo. Pero hay que hacerlo.

Si funciona... no lo toques

Las pruebas automáticas son una inversión rentable

Integración

Se trata de una clase `BankClient` que utiliza a otras para su operativa. Una `Account` para ingresos y gastos y otra `Loan` para créditos. La clase principal es la fachada que utiliza el usuario y dispone de una lógica mínima para manejar entradas y salidas de dinero.

Con lo que sabemos de *Jest* es fácil entender esta prueba; y entendiendo esta prueba es fácil adivinar la funcionalidad del programa. Mira en el código de legacy-integration.spec.js

```
import { BankClient } from '../bank-client';

let sutBankClient;
describe('GIVEN: a new BankClient WHEN: i deposit 20', () => {
  beforeAll(() => {
    const inputCredit = 100;
    const inputAmount = 20;
    arrangeBank(inputCredit);
    actDeposit(inputAmount);
  });
  test('THEN should have a balance of 20', assertBalance);
});
function arrangeBank(input) {
  sutBankClient = new BankClient(input);
}
function actDeposit(input) {
  sutBankClient.deposit(input);
}
function assertBalance() {
  const actual = sutBankClient.getPosition();
  const expected = 20;
  expect(actual).toEqual(expected);
}
```

Este código demuestra que el programa funciona, es una **prueba de integración**.

Ejercitamos a `BankClient` y *sin querer* a sus dependencias `Acount` y `Loan`

Si la prueba pasa es una buena dosis de **confianza**.

Pero si no la pasa... entonces no sabremos gran cosa sobe **el motivo** del fallo.

Unitarias

Las pruebas unitarias son para **descubrir fallos en una unidad manejable** de código.

DOC Depended On Components

Las dependencias incrustadas en el sujeto a probar (*Subject Under Test*), se evita al realizar **TDD** (*Testing Driven Development*).

Pero en código *legacy* hay que esquivar el problema de las dependencias simulando los DOC (*Depended On Components*) con los ***mocks***.

Espías

Lo más sencillo es **sustituir a cada dependencia por un objeto que cumpla** su interfaz. Sin implementación.

La idea es simplemente **comprobar si se llaman o no a los métodos adecuados**.

```
import { Account } from '../account';
import { BankClient } from '../bank-client';
jest.mock('./account');
describe('GIVEN: a new BankClient', () => {
  beforeEach(() => {
    Account.mockClear();
    const inputCredit = 100;
    arrangeBank(inputCredit);
  });
  test('THEN account constructor should be called', assertAccountConstructorIsCalled);
});
function arrangeBank(input) {
  sutBankClient = new BankClient(input);
}
function assertAccountConstructorIsCalled() {
  expect(Account).toHaveBeenCalledTimes(1);
}
```

En la línea `jest.mock('./account')` se instruye a *Jest* para que al *SUT* `BankClient` le dé un doble y no la clase real de su *DOC* `Account` .

La clase doble no hace nada práctico, pero permite saber si nuestro *SUT* hace las llamadas correctas.

Dobles

A veces es necesario **actuar en función de las respuestas** posibles.

El siguiente ejemplo usa un espía para la gestión del crédito, pero necesita **un doble para simular** el balance de la cuenta.

En el ejemplo completo nuestro espía es capaz de informar también de **los argumentos de cada llamada**.

La instrucción clave es `Account.mockImplementation()` que lleva incrustada una factoría de dobles de la clase `Account`.

En esta **factoría** implementamos los mínimos métodos imprescindibles, y simulamos las respuestas que nos convenga para probar el `BankClient`.

No es interesante la clase `Account`. Recuerda, probamos el *SUT* no sus *DOCs*

> Fíjate en el orden. Necesitamos montar al doble antes de que lo instancien.
> Mientras que el espionaje se realiza sobre una instancia ya construida.

Todo esto puede parecer lioso, y aunque al principio lo es, en cuestión de semanas lo integrarás en tus habilidades y te resultará natural. Pero sobre todo valorarás mucho no tener que abusar de estas técnicas. ¿Cómo? Con software bien diseñado ¿Cómo? Con TDD.