📈 Mejores resultados y mejor diseño

Hacer las pruebas antes mejora el código de después.

> *"TDD te hace escribir código más desacoplado, lo cual mejora el diseño del sistema."*
>
> -- ✍️ Robert C. Martin (Uncle Bob)

Aprender algo es costoso, incluirlo en tu rutina lo es aún más.

Tenemos que **visualizar el objetivo** para motivarnos.

E ir paso a paso para no desmotivarnos.

Test first

Supongamos que nos piden que el sistema sea capaz de obtener un balance a partir de transacciones anteriores.

```
FEATURE: a BankClient account
As_a: high level service
I_want_to: have a class where deposit money
In_order_to: accumulate several amounts of money for MUCH later
```

Pues empezamos por especificar un método llamado `calculateBalance` que reciba un array y devuelva un valor.

```
describe('GIVEN: a calculate balance function', () => {
  test('WHEN i have a transactions array THEN it calculates the balance', () => {
    const input = [{ _id: 1, amount: 12 }];
    const sut = new BankClient();
    const actual = sut.calculateBalance(input);
    const expected = 12;
    expect(actual).toEqual(expected);
  });
});
```

Con esto podemos empezar, obviamente habría que incluir más casos.

Better implementation

La implementación en la clase `BankClient` tiene algo implícitamente bueno: Se ha creado un método, se ha nombrado según el uso esperado y aprovechando el código previo, como la propiedad `this.balance`.

```
calculateBalance(transactions) {
  this.balance = transactions.reduce(
    (runningBalance, transaction) => runningBalance + transaction.amount,
    this.balance
  );
  return this.balance;
}
```

> Refactorizar al gusto: for, forEach...

Dependencias

Funciones que almacenen y lean transacciones, un efecto colateral no relacionado con la lógica bancaria:

¡Vamos a declara su *in* dependencia!

```
describe('GIVEN: a BankClient class with load logic', () => {
  test('WHEN  i load the transactions THEN it calls the specific function', () => {
    const getAllTransactions = () => [];
    const sut = new BankClient(getAllTransactions);
    const calculateBalanceSpy = jest.spyOn(sut, 'getAllTransactions');
    sut.load();
    expect(calculateBalanceSpy).toHaveBeenCalled();
  });
});
```

El sistema bancario usará una función llamada `getAllTransactions` dentro del método `load`.

Inyectables

```
export class BankClient {
  constructor(getAllTransactions) {
    this.getAllTransactions = getAllTransactions;
    this.balance = 0;
  }
  load() {
    const transactions = this.getAllTransactions();
  }
  ...
}
```

> La responsabilidad de la carga no es mía.

Mejoras paso a paso

Incorporando funcionalidad: calcular el balance tras la carga de las transacciones.

```
test('WHEN load transactions THEN call calculateBalance function', () => {
  const fakePreviousTransactions = [{ _id: 1, amount: 12 }];
  const getAllTransactions = () => fakePreviousTransactions;
  const sut = new BankClient(getAllTransactions);
  const calculateBalanceSpy = jest.spyOn(sut, 'calculateBalance');
  sut.load();
  expect(calculateBalanceSpy).toHaveBeenCalledWith(fakePreviousTransactions);
});
```

Pero además, con el argumento de entrada que corresponda con la salida producida por `getAllTransactions()`.

Y para cumplirlo hay que hacer muy poquito.

```
load() {
  const transactions = this.getAllTransactions();
  this.calculateBalance(transactions);
}
```

Asíncrono

El mundo asíncrono es independiente de realizar los tests antes o después. Se trata de exigir a la función `load` que sea asíncrona.

```
describe('GIVEN: a BankClient class with load logic', () => {
  test('WHEN load async transactions THEN waits for data and calculates balance', async () => {
    const fakePreviousTransactions = [{ _id: 1, amount: 12 }];
    const getAllTransactions = () => fakePreviousTransactions;
    const sut = new BankClient(getAllTransactions);
    const calculateBalanceSpy = jest.spyOn(sut, 'calculateBalance');
    await sut.load();
    expect(calculateBalanceSpy).toHaveBeenCalledWith(fakePreviousTransactions);
  });
});
```

```
async load() {
  const transactions = await this.getAllTransactions();
  this.calculateBalance(transactions);
}
```

No hemos hecho nada, simplemente obligar a que la implementación use la sintaxis asíncrona.

Promesas

Los dobles de las funciones asíncronas tiene la dificultad de... las funciones asíncronas.

Ejemplo de promesas simulando un retardo con `setTimeout`.

```
describe('GIVEN: a BankClient system with a previous saved transaction of 12', () => {
  let sut;
  beforeEach(async () => {
    const fakePreviousTransactions = [{ _id: 1, amount: 12 }];
    const loadPromise = new Promise(resolve => {
      setTimeout(() => resolve(fakePreviousTransactions), 1000);
    });
    const getAllTransactions = async () => loadPromise;
    const saveTransaction = function resolveAfter(transaction) {
      return new Promise(resolve => {
        setTimeout(() => {resolve({ ...transaction, _id: 1 });}, 1000);
      });
    };
    sut = new BankClient(getAllTransactions, saveTransaction);
    await sut.load();
  });
  test('WHEN: i make a deposit of 10 THEN returns a balance of 22', async () => {
    const input = 10; const expected = 22;
    const actual = await sut.deposit(input);
    expect(actual).toEqual(expected);
  });
});
```

Refactored

Recomendación: funciones con nombre **AAA**

```
beforeEach(async () => {
  const { getAllTransactions, saveTransaction } = arrangeDependencies();
  sut = new BankClient(getAllTransactions, saveTransaction);
  await sut.load();
});
function arrangeDependencies() {
  const loadPromise = new Promise(resolve => {
    setTimeout(() => resolve(fakePreviousTransactions), 1000);
  });
  const getAllTransactions = async () => loadPromise;
  const saveTransaction = function (transaction) {
    return new Promise(resolve => {
      setTimeout(() => {resolve({ ...transaction, _id: 1 });}, 1000);
    });
  };
  return { getAllTransactions, saveTransaction };
}
```

Sin API

En una capa de lógica de negocio no importa cómo o dónde se guardan las transacciones.

Basta con **estar preparados para trabajar de forma asíncrona** con las funciones que decidan inyectarnos...

Tenemos el código preparado para usar técnicas de **inversión de control**.

> El código está desacoplado y es muy sencillo mantenerlo. Este es el objetivo del software bien diseñado. Para conseguirlo merece la pena el esfuerzo invertido.