🧪 Limpieza de pruebas

Los tests son código. Deben ser lo más explícitos y claros posible.

> *"La duplicidad es más barata que la mala abstracción"*
>
> -- ✍️ Sandi Metz

Prestar atención al código de las pruebas

El código de las pruebas debe ser muy sencillo

Las pruebas son código, y el código ha de estar limpio.

Podríamos permitirles *ciertas licencias*. Pero otras rotundamente no.

Malos olores en las pruebas

Comentarios

aclarar en código la intención de la prueba.
Excepción: historia de usuario.

Datos mágicos

¿Por qué hay cadenas y números desperdigados por las pruebas?

Datos absurdos

¿Qué ganas aporreando el teclado y generando *sdfasd@yjtj.cf* ?

Anidamientos sin fin

¿Te gusta esto?

```
describe('',()=>{context('',()=>it('', ()=>{yourTestingCode(yourParam);}))})
```

Extrae *callbacks* a funciones con nombre.

Licencias para manchar

DRY, WET y DAMP

Seco, mojado y húmedo.

DRY

El principio **DRY (Don´t Repeat Yourself)** asegura que **"La duplicidad es el principal enemigo de un sistema bien diseñado**". Pero...

- **las pruebas deben ser fácilmente entendidas** y modificadas por cualquiera.
- **el código de una prueba debe aislarse** de las demás;
- **las pruebas no tienen pruebas**.

WET

No es práctico caer en el **WET (Write Everything Twice)**.

Incluso afecta a la moral del equipo verse **repitiendo, copiando y pegando, siempre el mismo código**.

DAMP

DAMP (Descriptive And Meaningful Phrases).

> el código es más entendible cuanto mejor nombrado esté.

Los frameworks y los programadores lo cumplimos al rellenar las cadenas de cada `describe('')` y de cada `it('')`

Recomendaciones para el código de pruebas

- Muchas pruebas pequeñas.
- Un fichero, módulo, para cada prueba.
- Textos *super mega ultra hyper* descriptivos.
- Extrae los datos a variables o constantes del módulo.
- Extrae los *callbacks* a funciones locales con nombre **AAA**.
- Permítete pequeños ficheros de utilidad comunes a las pruebas.
- Pero **sin abstracciones complejas**, sólo configuraciones o funciones.

Ejemplo con Cypress

En el laboratorio.

Intención, comentarios y otras historias

```
// FEATURE:     the app should have a well formed html
// As a:        user
// I want to:   view a recognizable web page
// In order to: feel safe using it
```

Textos significativos y descriptivos

```
describe(`GIVEN: the proton tasks web app`, () => {
  arrangeTest();
  context(`WHEN: I visit the url ${Cypress.env(baseUrl)} `, () => {
    actVisit();
    it(`THEN: should have charset UTF-8`, assertCharset);
    it(`AND THEN: should have _Proton Tasks_ on Title`, assertTitle);
    it(`AND THEN: should have a header`, assertHeader);
    it(`AND THEN: should have an h1 on the header with text _Proton Tasks_`, assertH1ContainText);
  });
});
```

Variables y funciones

```
let sutUrl;
let expectedTitle;
let selectorHeader;
let selectorH1;
```

```
function arrangeTest() {
  ignoreParcelError();
  sutUrl = Cypress.env('baseUrl');
  expectedTitle = 'Proton Tasks';
  selectorHeader = 'header';
  selectorH1 = 'header > h1';
}
function actVisit() {
  before(() => cy.visit(sutUrl));
}
function assertCharset() {
  cy.document().should('have.property', 'charset').and('eq', 'UTF-8');
}
```

Configuraciones, datos, utilidades y ejecución

Configuración

`cypress.json` y `cypress.env.json`

Pensado para configuraciones, no para datos. Se leen con `Cypress.env(nombrePropiedad)`.

```
{
  "chromeWebSecurity": false,
  "baseUrl": "https://labsademy.github.io/ProtonTasks/",
  "env": {},
  "video": true
}
```

Y otra para desarrollo

```
{
  "chromeWebSecurity": false,
  "baseUrl": "http://localhost:1234",
  "env": {},
  "video": false
}
```

Datos enviados y esperados

`fixtures`.

Es clave **usar este tipo de convenios para no sobrecargar el cerebro** del siguiente lector.

Comandos

> Compartir o reutilizar lógica y utilidades. *Cypress* nos ofrece su propia solución y le llama `commands`.

```
Cypress.Commands.add('typeText', (selector, text) => cy.get(selector).type(text));
```

> El problema es que es un mecanismo dinámico; lioso para ofrecer *intellisense*.

```
cy.typeText(selector, text);
```

La alternativa es usar un función de toda la vida y exportarla.

```
export const typeText = (selector, text) => cy.get(selector).type(text);
```

Y luego importarla y usarla como si tal cosa.

```
import { typeText } from '../../support/actions';
typeText(selector, text);
```

Útil para **preparar y limpiar escenarios antes y después de los tests**. Por ejemplo, haciendo *login*, *logout*, creando o borrando datos...

Scripts

Presencial: ejecutar las pruebas mientras desarrolla, o antes, si es un *TDD practitioner*.

Desatendida: en modo consola, **integración continua** con videos, fotos y reportes.

```
  "scripts": {
    "start": "cypress open",
    "test": "cypress run"
  }
```

Reporters

```
yarn add -D mocha mochawesome
```

```
  "video": true,
  "screenshotOnRunFailure": true,
  "trashAssetsBeforeRuns": true,
  "reporter": "mochawesome",
  "reporterOptions": {
    "charts": false,
    "html": true,
    "json": true,
    "reportDir": "cypress/reports",
    "reportFilename": "report",
    "overwrite": false
  }
```

> En definitiva, las pruebas son código y merecen un respeto. *Cypress* te ofrece lo necesario para que sean fáciles de entender y mantener.