



Universitat
Oberta
de Catalunya

M4.256 - Desarrollo front-end avanzado

PEC 6

Aníbal Santos Gómez

- **Ejercicio 1 – Estudio sobre code smells**

¿Qué es un Code Smell?

Un code smell, también conocido como mala práctica, en el desarrollo de software, se refiere a cualquier síntoma en el código fuente de un programa que posiblemente indique un problema más profundo.

Los code smells usualmente no son bugs – no son técnicamente incorrectos y en realidad no impiden que el programa funcione correctamente.

Pero indican deficiencias en el diseño software que puede ralentizar el desarrollo o aumentan el riesgo de errores o fallos en el futuro.

Categorización de los Code Smells

- **Bloaters:** son código, métodos y clases que han aumentado a proporciones tan gigantescas que son difíciles de trabajar.
 - Long method: un método que contiene muchas líneas de código o hace demasiadas cosas.
 - Large Class: una clase que tiene muchos campos, métodos o líneas de código.
 - Primitive Obsession: consiste en utilizar datos primitivos en vez de objetos más simples para tareas más simples.
 - Long Parameter List: un método recibe más de tres o cuatro parámetros. Lo ideal es que reciba máximo uno o dos.
 - Data Clumps: diferentes partes del código contienen idénticos grupos de variables.
- **Object Orientation Abusers:** aplicación incorrecta de los principios de la programación orientada a objetos.
 - Switch Statements: tenemos un switch con complicación lógica o una gran cantidad de if, else if.
 - Temporary Field: los campos temporales obtienen su valor bajo determinadas circunstancias, pero fuera de estos casos están vacíos y sin un valor por defecto.
 - Refused Bequest: se da cuando una subclase utiliza parcialmente algunos métodos de las clases de las que extiende.

- Alternative Classes, different interfaces: dos clases realizan funciones idénticas pero tienen diferentes nombres de métodos.
- **Change Preventer:** significan que cambiar al cambiar algo en un lugar del código, hay que hacer muchos cambios en otros lugares también.
 - Divergent Change: cambiar muchos métodos no relacionados cuando hacemos cambios en una clase.
 - Shotgun Surgery: Hacer una modificación requiere que se hagan muchos pequeños cambios en muchas clases diferentes.
 - Parallel Inheritance Hierarchies: cuando creamos una subclase para una clase, nos encontramos con la necesidad de crear una subclase para otra clase distinta.
- **Dispensables:** es algo inútil cuya ausencia hace el código más limpio, eficiente y fácil de entender.
 - Comments: cuando nuestro código no se explica por si mismo y tenemos muchos comentarios explicativos.
 - Duplicate Code: dos fragmentos de código parecen casi idénticos.
 - Lazy Class: una clase fue diseñada para ser totalmente funcional, pero después del refactoring se ha vuelto muy pequeña.
 - Data Class: cuando una clase solo es contenedora de datos y no tiene ninguna funcionalidad.
 - Dead Code: una variable, un parámetro, un campo, un método o una clase ya no se utilizan.
 - Speculative Generality: clase, un método, un campo o un parámetro no utilizados.
- **Couplers:** acoplamiento excesivo entre clases.
 - Feature Envy: un método accede a los datos de otro objeto más que a sus propios datos.
 - Inappropriate Intimacy: una clase utiliza los campos y métodos internos de otra clase.
 - Message Chains: se da cuando un cliente solicita un objeto y este otro más y así sucesivamente. Un cambio en las relaciones hace que tengamos que modificar el cliente.

- Middle Man: se da cuando la única acción de una clase es delegar trabajo en otras clases.
- Incomplete Library Class: El autor de una librería no ha proporcionado las características que necesitamos.

A continuación se han seleccionado **5 code smells**:

1. Long Method:

- a. Nombre: Long Method**
- b. Problema:** Un método que contiene demasiadas líneas de código y puede abstraerse o simplificarse. Normalmente cuando un método empieza a crecer puede ser poco mantenible y pierde el principio de que una función debe de hacer una sola cosa (clean code).
- c. Técnica de refactoring:** para mejorar la mantenibilidad del código podemos utilizar la técnica de **extraer en un método**, partes del método a refactorizar que no deberían de pertenecer al mismo.
- d. Código ilustrativo.**



The screenshot shows a code editor window with a dark theme. At the top left, there are three circular icons: red, yellow, and green. The code editor displays a file with the following content:

```
1  /**
2   * * Long Method
3   */
4
5  const ComputePrice = (): number => {
6    const items = Array(10)
7      .fill(0)
8      .map(() => ({ price: Math.random() * 10 }));
9
10  const itemsPriceSum = items.reduce(
11    (priceSum, item) => priceSum + item.price,
12    0
13  );
14
15  const finalPrice = itemsPriceSum * 0.8;
16
17  return finalPrice;
18};
19
```

2. Primitive Obsession:

- a. **Nombre:** Primitive Obsession
- b. **Problema:** Se da en situaciones en las cuales utilizamos muchos tipos de datos primitivos sencillos, constantes que codifican información, etc.
- c. **Técnica de refactoring:** podemos utilizar varias técnicas, pero cuando los datos se complican, podemos **reemplazar los tipos de datos por clases**.
- d. **Código ilustrativo**

```
1 class Person {  
2     constructor(  
3         private name: string,  
4         private age: number,  
5         private homeCityName: string,  
6         private homeStreetName: string,  
7         private homeZipCode: number  
8     ) {}  
9  
10    get description() {  
11        return `${this.name}, ${this.age} years old, lives in ${this.address}`;  
12    }  
13  
14    get address() {  
15        return `${this.homeStreetName} in ${this.homeCityName} (${this.homeZipCode})`;  
16    }  
17}  
18
```

3. Middle Man:

- a. **Nombre:** Middle Man
- b. **Problema:** se da cuando la única función de una clase es sólo delegar.
- c. **Técnica de refactoring:** para refactorizar nuestro código **eliminaremos el intermediario.**
- d. **Código ilustrativo**



```
1 type Student = {
2   id: number;
3   name: string;
4   semester: number;
5 };
6
7 let Collection: Array<Student>;
8
9 class Model {
10   public insertOne(data: Student): boolean {
11     try {
12       Collection.push(data);
13       return true;
14     } catch (error) {
15       return false;
16     }
17   }
18 }
19
20 class Controller {
21   saveOne(id: number, name: string, semester: number) {
22     const model = new Model();
23     const tmp: Student = {
24       id,
25       name,
26       semester,
27     };
28     model.insertOne(tmp);
29   }
30 }
31
```

4. Data Cumpls:

- a. **Nombre:** Data Clumps
- b. **Problema:** diferentes partes del código contienen grupos idénticos de variables. Estos grupos deben convertirse en sus propios bloques, objetos o clases.
- c. **Técnica de refactoring:** extraer a una clase o a un objeto manejable.
- d. **Código ilustrativo.**



The screenshot shows a code editor window with a dark theme. At the top left, there are three colored circular icons: red, yellow, and green. The code itself is as follows:

```
1  /**
2   * * Before smell
3   */
4
5  let a1 = [18.2, 34.2, 19, 2];
6  let a2 = ['anibal', 'juan', 'maria'];
7
8 /**
9  * * After Smell
10 */
11
12 let a3 = ['123 Calle falsa', 'Salamanca', 'ES', '37008'];
```

5. Temporary Field:

- a. **Nombre:** Temporary Field
- b. **Problema:** estos campos temporales obtienen sus valores y solo son necesarios para determinadas circunstancias, fuera de esta situación están vacíos.
- c. **Técnica de refactoring:** podemos abstraer estos parámetros temporales a un método o clase.
- d. **Código ilustrativo**



```
1 function nameToObject(name) {  
2     const fullName = name.split(' ');  
3     const firstName = fullName[0];  
4     const lastName = fullName[1];  
5  
6     return {  
7         firstName: `${firstName}`,  
8         lastName: `${lastName}`,  
9     };  
10 }  
11
```

- **Ejercicio 2 – Documentación sobre técnicas de refactoring**

Así como la lista categorizada de code smells anterior, también tenemos una lista categorizada sobre técnicas de refactorización:

- **Composing Methods.**

- **Extracción de método.**
- **Método inline.**
- **Extracción de variable.**
- **Inline Temp.**
- **Reemplazar temp por consulta.**
- **Dividir variable temporal.**
- **Eliminar asignaciones a parámetros.**
- **Sustituir método por objeto de método.**
- **Sustituir algoritmo**

- **Mover características entre objetos.**

- **Método de desplazamiento.**
- **Mover campo.**
- **Extraer clase.**
- **Clase inline.**
- **Ocultar delegado.**
- **Eliminar el intermediario.**
- **Introducir método extranjero.**
- **Introducir extensión local.**

- **Organizar Datos.**

- **Cambiar el valor por la referencia.**
- **Cambiar la referencia por el valor.**
- **Duplicar datos observados.**
- **Campo autoencapsulado.**
- **Sustituir valor de datos por un objeto.**
- **Reemplazar array por un objeto.**
- **Cambiar asociación unidireccional a bidireccional.**
- **Cambiar asociación bidireccional a unidireccional.**
- **Encapsular campo.**
- **Encapsular colección.**
- **Sustituir número mágico por constante simbólica.**
- **Sustituir código de tipo por clase.**
- **Sustituir código de tipo por subclases.**
- **Reemplazar código de tipo por estado/estrategia.**
- **Reemplazar subclase por campos.**

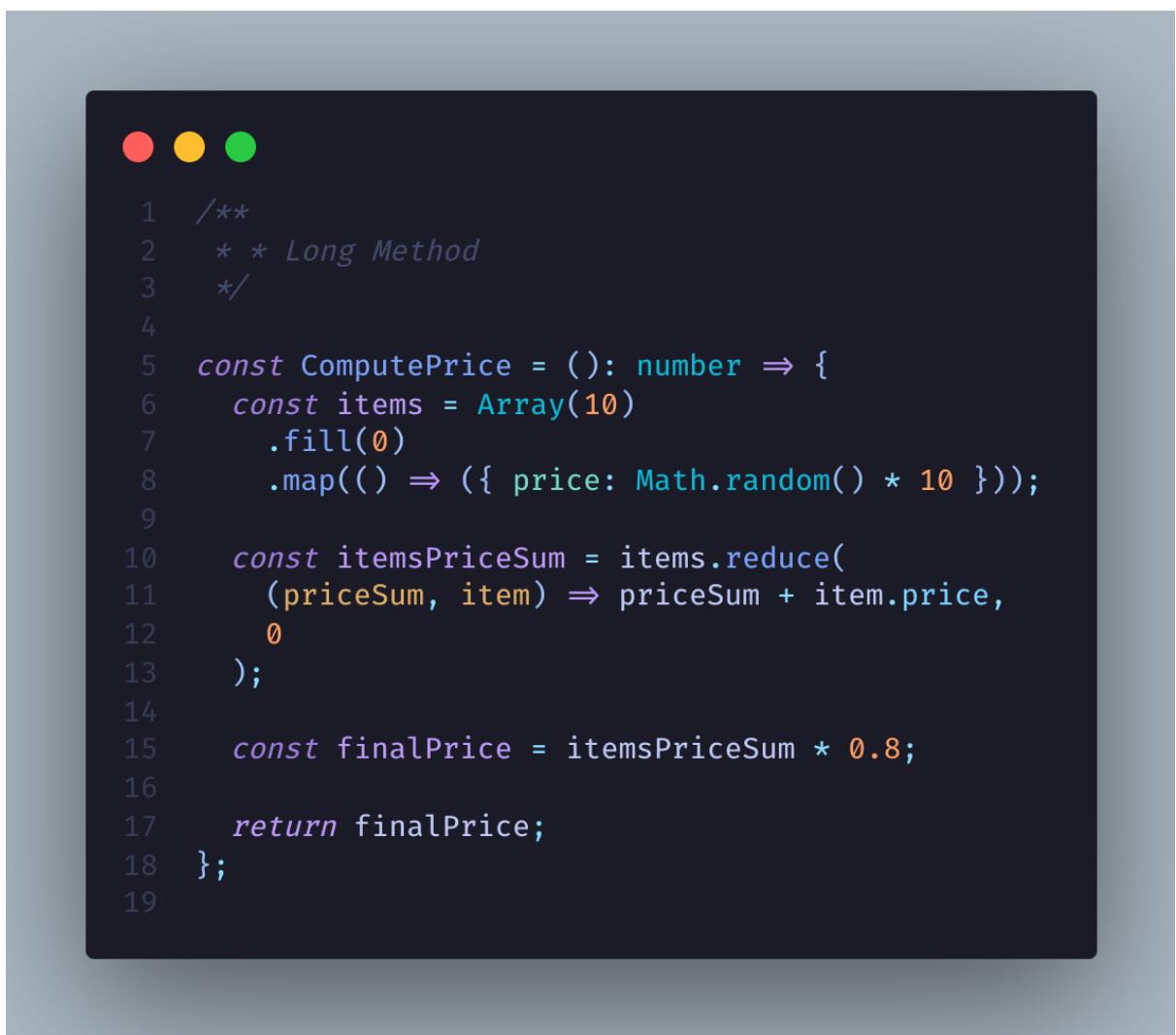
- Simplificar condicionales.
 - Consolidar expresiones condicionales.
 - Consolidar fragmentos condicionales duplicados.
 - Descomponer condicional.
 - Reemplazar condicional por Polimorfismo.
 - Eliminar el indicador de control.
 - Reemplazar condicional anidada con cláusulas de guarda.
 - Introducir objeto nulo.
 - Introducir la aserción.
- Simplificar llamadas a métodos.
 - Añadir parámetro
 - Eliminar parámetro
 - Cambiar el nombre del método
 - Separar la consulta del modificador
 - Parametrizar método
 - Introducir objeto parámetro
 - Conservar todo el objeto
 - Eliminar el método de parametrización
 - Reemplazar parámetro con métodos explícitos
 - Reemplazar parámetro con llamada a método
 - Ocultar método
 - Reemplazar constructor por método de fábrica
 - Reemplazar código de error por excepción
 - Reemplazar excepción por prueba
- Generalización.
 - Pull Up Field
 - Pull Up Method
 - Pull Up Constructor Body
 - Push Down Field
 - Push Down Method
 - Extraer subclase
 - Extraer superclase
 - Interfaz extract
 - Colapsar jerarquía
 - Método de plantilla de formulario
 - Reemplazar herencia por delegación
 - Reemplazar delegación por herencia

A continuación se exponen **5 técnicas** de las anteriormente listadas para refactorizar los **code smells del Ejercicio 1**

1. Extraer método.

- a. **Nombre:** Extraer método
- b. **Cuándo usarla:** podemos utilizar esta técnica cuando tenemos un método largo con mucha funcionalidad reagrupable en otros pequeños métodos reutilizables que hagan nuestro código más mantenable.
- c. **Código Ilustrativo (antes y después).**

Antes:



The screenshot shows a code editor window with a dark theme. At the top left, there are three circular icons: red, yellow, and green. The code editor displays a single file with line numbers from 1 to 19. The code is a long function named 'ComputePrice' that creates an array of 10 items, each with a random price between 0 and 10, and then calculates the total sum and applies an 80% discount.

```
1  /**
2   * * Long Method
3   */
4
5  const ComputePrice = (): number => {
6    const items = Array(10)
7      .fill(0)
8      .map(() => ({ price: Math.random() * 10 }));
9
10  const itemsPriceSum = items.reduce(
11    (priceSum, item) => priceSum + item.price,
12    0
13  );
14
15  const finalPrice = itemsPriceSum * 0.8;
16
17  return finalPrice;
18};
19
```

Después:

```
● ● ●
1  interface Item {
2    price: number;
3  }
4
5  const generateItems = (): Item[] =>
6    Array(10)
7      .fill(0)
8      .map(() => ({ price: Math.random() * 10 }));
9
10 const sumItemPrices = (items: Item[]) =>
11   items.reduce((priceSum, item) => priceSum + item.price, 0);
12
13 const applyTwentyPercentDiscount = (price: number) => price * 0.8;
14
15 const cleanComputePrice = (): number => {
16   const items = generateItems();
17
18   return applyTwentyPercentDiscount(sumItemPrices(items));
19 };
20
```

2. Extraer clase.

- a. **Nombre:** Extraer clase
- b. **Cuándo usarla:** podemos utilizar esta técnica cuando una clase está realmente realizando el trabajo que podrían hacer dos clases.
- c. **Código Ilustrativo (antes y después).**

Antes

```
1 class Person {
2   constructor(
3     private name: string,
4     private age: number,
5     private homeCityName: string,
6     private homeStreetName: string,
7     private homeZipCode: number
8   ) {}
9
10  get description() {
11    return `${this.name}, ${this.age} years old, lives in ${this.address}`;
12  }
13
14  get address() {
15    return `${this.homeStreetName} in ${this.homeCityName} (${this.homeZipCode})`;
16  }
17}
18
```

Después

```
1 class City {
2   constructor(public name: string, public zipCode: number) {}
3
4   toString() {
5     return `${this.name} (${this.zipCode})`;
6   }
7 }
8
9 class Address {
10   constructor(public city: City, public streetName: string) {}
11
12   toString() {
13     return `${this.streetName} in ${this.city}`;
14   }
15 }
16
17 class CleanPerson {
18   constructor(
19     public name: string,
20     public age: number,
21     public address: Address
22   ) {}
23
24   get description() {
25     return `${this.name}, ${this.age} years old, lives in ${this.address}`;
26   }
27 }
28
```

3. Eliminar intermediario (middle man).

- a. **Nombre:** Eliminar middle man
- b. **Cuándo usarla:** podemos usar esta técnica cuando una clase tiene métodos que simplemente delegan en otros objetos.
- c. **Código Ilustrativo (antes y después).**

Antes:

```
● ● ●

1 type Student = {
2   id: number;
3   name: string;
4   semester: number;
5 };
6
7 let Collection: Array<Student>;
8
9 class Model {
10   public insertOne(data: Student): boolean {
11     try {
12       Collection.push(data);
13       return true;
14     } catch (error) {
15       return false;
16     }
17   }
18 }
19
20 class Controller {
21   saveOne(id: number, name: string, semester: number) {
22     const model = new Model();
23     const tmp: Student = {
24       id,
25       name,
26       semester,
27     };
28     model.insertOne(tmp);
29   }
30 }
31
```

Después:

```
● ● ●
1 type Student2 = {
2   id: number;
3   name: string;
4   semester: number;
5 };
6
7 let Collection1: Array<Student2>;
8
9 class Model2 {
10   public insertOne(data: Student2): boolean {
11     try {
12       Collection.push(data);
13       return true;
14     } catch (error) {
15       return false;
16     }
17   }
18 }
19
```

4. Conservar todo el objeto.

- a. **Nombre:** Conservar todo el objeto
- b. **Cuándo usarla:** cuando tengamos datos dispersos o que no sean claramente semánticos, podemos transformarlos a un objeto para que sean más coherentes con la información que queremos manejar.
- c. **Código Ilustrativo (antes y después).**

Antes



```
1  /**
2  * * Before smell
3  */
4
5  let a1 = [18.2, 34.2, 19, 2];
6  let a2 = ['anibal', 'juan', 'maria'];
7
8 /**
9 * * After Smell
10 */
11
12 let a3 = ['123 Calle falsa', 'Salamanca', 'ES', '37008'];
```

Después

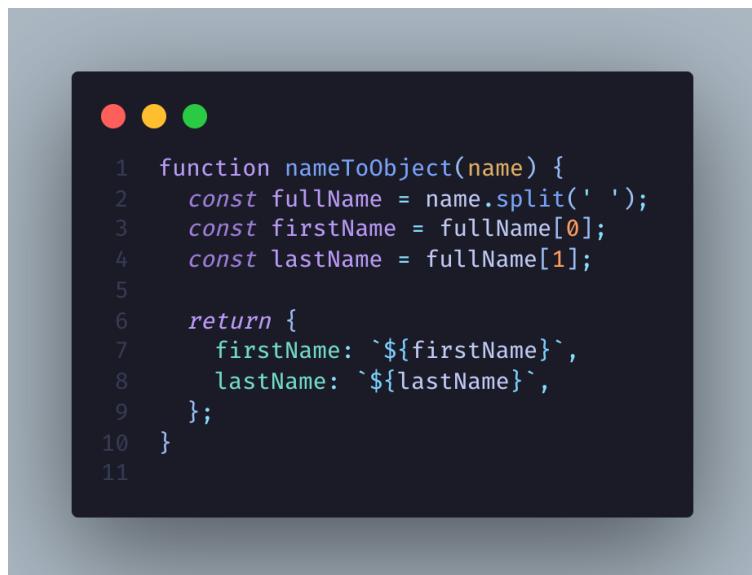


```
1 let address1 = {
2   address: '123 Calle falsa',
3   city: 'Salamanca',
4   country: 'ES',
5   zip: '37008',
6 };
7
```

5. Sustituir el método por el objeto del método.

- a. **Nombre:** Sustituir el método por el objeto del método
- b. **Cuándo usarla:** cuando tenemos un método en el que las variables están entrelazadas para poder realizar una extracción del método, podemos sustituir directamente estas variables.
- c. **Código Ilustrativo (antes y después).**

Antes



```
function nameToObject(name) {  
  const fullName = name.split(' ');\n  const firstName = fullName[0];\n  const lastName = fullName[1];\n\n  return {\n    firstName: `${firstName}`,  
    lastName: `${lastName}`,  
  };\n}  
11
```

Después



```
function nameToObject(name) {  
  const fullName = name.split(' ');\n  return {\n    firstName: fullName[0],  
    lastName: fullName[1],  
  };\n}  
8
```

- **Ejercicio 3 – Refactorizando nuestro proyecto**

Ejemplos de refactorización para **tourist-activities-app**.

1. Ejemplo 1. Código duplicado

- a. **Componente(s)**: LoginService, UserService, ActivityService.
- b. **Método**: extraer método login y eliminar la duplicidad.
- c. **Explicación**: tenemos en ambos servicios un método que es idéntico, por lo que vamos a extraerlo y utilizarlo en ambos, en este caso, como afecta a la funcionalidad de login de usuario, tiene sentido conservarlo en el servicio de user que es el encargado de toda la funcionalidad del usuario. Por lo tanto, podemos eliminar el servicio de loginService.

Además, aprovechando que tenemos más código duplicado en los servicios, vamos a refactorizar extrayendo a una clase común el log de errores que tenemos en todos los servicios.

- d. **Capturas antes**.

loginService:

```
@Injectable({
  providedIn: 'root'
})
export class LoginService {
  private usersUrl = 'api/users'; // URL to web api

  constructor(private http: HttpClient) {}

  login({ email, password }: Credentials): Observable<any> {
    return this.http.get<User[]>(this.usersUrl).pipe(
      map((users) => {
        const user = users.find(x => x.profile.email === email && x.profile.password === password);
        if (user === undefined){
          return user;
        }
        else{
          throw throwError('Invalid username or password');
        }
      })
    );
  }
}
```

UserService:

```
@Injectable({
  providedIn: 'root',
})
export class UserService extends CommonService {
  private usersUrl = 'api/users'; // URL to web api

  constructor(private http: HttpClient) {
    super();
  }

  login({ email, password }: Credentials): Observable<any> {
    return this.http.get<User[]>(this.usersUrl).pipe(
      map((users) => {
        const user = users.find(
          (x) => x.profile.email === email && x.profile.password === password
        );
        if (user === undefined) {
          return user;
        } else {
          throw throwError('Invalid username or password');
        }
      })
    );
  }
}
```

ActivityService y UserService:

```
/**
 * Handle Http operation that failed.
 * Let the app continue.
 * @param operation - name of the operation that failed
 * @param result - optional value to return as the observable result
 */
private handleError<T>(operation = 'operation', result?: T) {
  return (error: any): Observable<T> => {

    // TODO: send the error to remote logging infrastructure
    console.error(error); // log to console instead

    // Let the app keep running by returning an empty result.
    return of(result as T);
  };
}
```

e. Capturas después.

```
@Injectable({
  providedIn: 'root',
})
export class UserService extends CommonService {
  private usersUrl = 'api/users'; // URL to web api

  constructor(private http: HttpClient) {
    super();
  }

  login({ email, password }: Credentials): Observable<any> {
    return this.http.get<User[]>(this.usersUrl).pipe(
      map((users) => {
        const user = users.find(
          (x) => x.profile.email === email && x.profile.password === password
        );
        if (user === undefined) {
          return user;
        } else {
          throw throwError('Invalid username or password');
        }
      })
    );
  }
}
```

```
import { Injectable } from '@angular/core';
import { HttpHeaders } from '@angular/common/http';
import { Observable, of } from 'rxjs';

@Injectable({
  providedIn: 'root',
})
export class CommonService {
  httpOptions = {
    headers: new HttpHeaders({ 'Content-Type': 'application/json' }),
  };
  handleError<T>(operation = 'operation', result?: T) {
    return (error: any): Observable<T> => {
      // TODO: send the error to remote logging infrastructure
      console.error(error); // log to console instead

      // Let the app keep running by returning an empty result.
      return of(result as T);
    };
  }
}
```

```
import { UserService } from 'src/app/profile/services/user.service';

@Injectable()
export class LoginEffects {
  login$ = createEffect(() =>
    this.actions$.pipe(
      ofType(LoginActions.login),
      exhaustMap(({ credentials }) =>
        this.userService.login(credentials).pipe(
          map((user) => LoginActions.loginSuccess({ credentials })),
          catchError((error) =>
            of(LoginActions.loginFailure({ payload: error }))
          )
        )
      )
    );
}
```

2. Ejemplo 2. Método Largo

- a. **Componente(s):** activity-detail.component.ts
- b. **Método:** Extraer el método para eliminar carga en **loadFormInstance()**
- c. **Explicación:** tenemos un método excesivamente largo en nuestro componente de activity detail, por lo que vamos a separarlo en un método más manejable.
- d. **Capturas antes.**

```

public loadFormInstance(): void {
    this.rForm = new FormGroup({
        name: new FormControl(this.activity?.name),
        category: new FormControl(this.activity?.category),
        subcategory: new FormControl(this.activity?.subcategory),
        description: new FormControl(this.activity?.description),
        language: new FormControl(this.activity?.language),
        date: new FormControl(this.activity?.date),
        price: new FormControl(this.activity?.price),
        miniumCapacity: new FormControl(this.activity?.miniumCapacity),
        limitCapacity: new FormControl(this.activity?.limitCapacity),
        peopleRegistered: new FormControl(this.activity?.peopleRegistered),
        state: new FormControl(this.activity?.state),
        myActivitySignUpVisible: new FormControl(false),
        saveFavoritesVisible: new FormControl(false),
        deleteFavoritesVisible: new FormControl(false),
        myActivityDeleteVisible: new FormControl(false),
    });
    // Si hay un usuario logado y tiene el perfil de turista
    // se muestran los botones de sign up y save favorites
    const idLoggedUser = this.userState$.user?.id;

    if (
        this.userState$.user !== null &&
        this.userState$.user?.profile.type === userTypes.Tourist.toString()
    ) {

        // Se obtienen la lista de actividades favoritas del usuario de la memoria local
        this.idActivitiesUserFavorites = this.userState$.user?.profile.favorites;

        // Si la información de las actividades se visualiza desde la opción "My activities" (@Input() eFilterType)
        if (this.eFilterType === FilterType.myActivitiesFilter.toString()) {
            // Se habilita el botón de eliminar el registro a la actividad
            this.rForm.controls.myActivityDeleteVisible.setValue(true);
        }

        // Si la información de las actividades se visualiza desde la opción "Favorites"
        // (@Input() eFilterType)
        else if (this.eFilterType === FilterType.favoritesFilter.toString()) {
            // Se habilita el botón de eliminar la selección favorita de la actividad
            this.rForm.controls.deleteFavoritesVisible.setValue(true);
        }
        // En caso de visualizarse desde la opción de actividades sin filtro
        // si el usuario logado tiene un perfil Tourist
        else {

            // Se muestra el botón de sign Up y favorites en caso de que no estén cubiertas
            // las plazas disponibles
            // y que el usuario no esté ya apuntado en la actividad
            if (
                this.activity.peopleRegistered < this.activity.limitCapacity &&
                !this.activity.signUpUsers.includes(idLoggedUser)
            ) {
                this.rForm.controls.myActivitySignUpVisible.setValue(true);
            }
            // Si el usuario no tiene la actividad como favorita
            // se muestra el botón de añadir a favoritos
            const foundIndex = this.idActivitiesUserFavorites.findIndex(
                (x) => x === this.activity.id
            );
            if (foundIndex === -1) {
                this.rForm.controls.saveFavoritesVisible.setValue(true);
            }
        }
    }
}

```

e. Capturas después.

```
displayTouristOptions(): void {
    // Si hay un usuario logado y tiene el perfil de turista
    // se muestran los botones de sign up y save favorites
    const idLoggedUser = this.userState$.user?.id;

    if (
        this.userState$.user != null &&
        this.userState$.user?.profile.type === userTypes.Tourist.toString()
    ) {

        // Se obtienen la lista de actividades favoritas del usuario de la memoria local
        this.idActivitiesUserFavorites = this.userState$.user?.profile.favorites;

        // Si la información de las actividades se visualiza desde la opción "My activities" (@Input() eFilterType)
        if (this.eFilterType === FilterType.myActivitiesFilter.toString()) {
            // Se habilita el botón de eliminar el registro a la actividad
            this.rForm.controls.myActivityDeleteVisible.setValue(true);
        }

        // Si la información de las actividades se visualiza desde la opción "Favorites" (@Input() eFilterType)
        else if (this.eFilterType === FilterType.favoritesFilter.toString()) {

            // Se habilita el botón de eliminar la selección favorita de la actividad
            this.rForm.controls.deleteFavoritesVisible.setValue(true);
        }
        // En caso de visualizarse desde la opción de actividades sin filtro
        // si el usuario logado tiene un perfil Tourist
        else {

            // Se muestra el botón de sign Up y favorites en caso de que no estén cubiertas
            // las plazas disponibles
            // y que el usuario no esté ya apuntado en la actividad
            if (
                this.activity.peopleRegistered < this.activity.limitCapacity &&
                !this.activity.signUpUsers.includes(idLoggedUser)
            ) {
                this.rForm.controls.myActivitySignUpVisible.setValue(true);
            }
            // Si el usuario no tiene la actividad como favorita
            // se muestra el botón de añadir a favoritos
            const foundIndex = this.idActivitiesUserFavorites.findIndex(
                (x) => x === this.activity.id
            );
            if (foundIndex === -1) {
                this.rForm.controls.saveFavoritesVisible.setValue(true);
            }
        }
    }
}
```

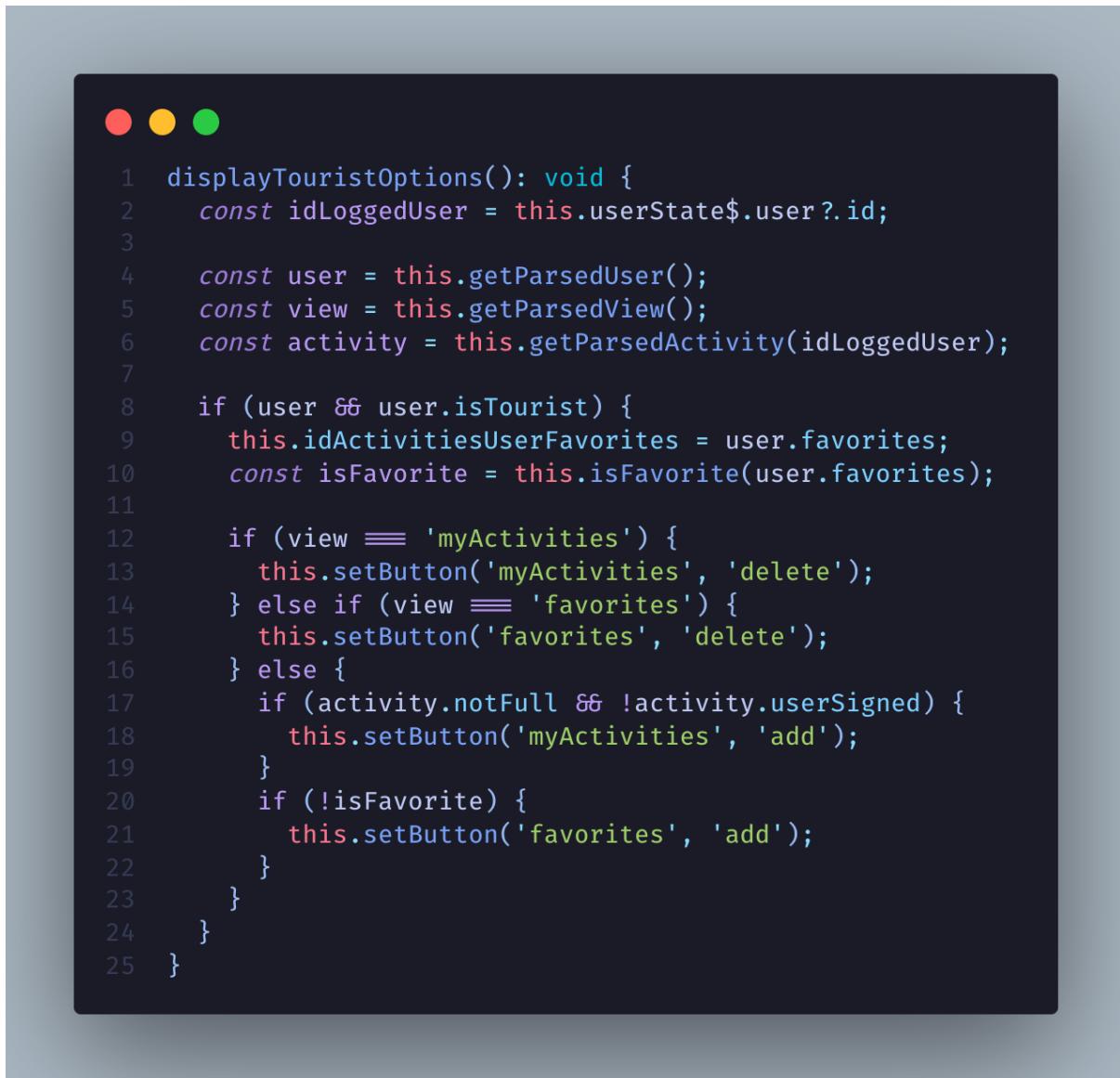
3. Ejemplo 3. Comentarios

- a. **Componente(s):** activity-detail.component.ts
- b. **Método:** aplicaremos extracción de método en **displayTouristOptions()**
- c. **Explicación:** después de refactorizar el componente anterior, nos encontramos con un método que contiene muchos comentarios explicativos de lo que se hace, esto también es un code smell, por lo que haremos una extracción de método y trataremos la lógica como pequeños objetos para poder hacerlo autoexplicativo.
- d. **Capturas antes.**



```
1 displayTouristOptions(): void {
2     // Si hay un usuario logado y tiene el perfil de turista
3     // se muestran los botones de sign up y save favorites
4     const idLoggedUser = this.userState$.user?.id;
5
6     if (
7         this.userState$.user !== null &&
8         this.userState$.user?.profile.type === userTypes.Tourist.toString()
9     ) {
10        // Se obtienen la lista de actividades favoritas del usuario de la memoria local
11        this.idActivitiesUserFavorites = this.userState$.user?.profile.favorites;
12        // Si la información de las actividades se visualiza desde la opción "My activities" (@Input() eFilterType)
13        if (this.eFilterType === FilterType.myActivitiesFilter.toString()) {
14            // Se habilita el botón de eliminar el registro a la actividad
15            this.rForm.controls.myActivityDeleteVisible.setValue(true);
16        }
17        // Si la información de las actividades se visualiza desde la opción "Favorites" (@Input() eFilterType)
18        else if (this.eFilterType === FilterType.favoritesFilter.toString()) {
19            // Se habilita el botón de eliminar la selección favorita de la actividad
20            this.rForm.controls.deleteFavoritesVisible.setValue(true);
21        }
22        // En caso de visualizarse desde la opción de actividades sin filtro
23        // si el usuario logado tiene un perfil Tourist
24    else {
25        // Se muestra el botón de sign Up y favorites en caso de que no estén cubiertas las plazas disponibles
26        // y que el usuario no esté ya apuntado en la actividad
27        if (
28            this.activity.peopleRegistered < this.activity.limitCapacity &&
29            !this.activity.signUpUsers.includes(idLoggedUser)
30        ) {
31            this.rForm.controls.myActivitySignUpVisible.setValue(true);
32        }
33        // Si el usuario no tiene la actividad como favorita
34        // se muestra el botón de añadir a favoritos
35        const foundIndex = this.idActivitiesUserFavorites.findIndex(
36            (x) => x === this.activity.id
37        );
38        if (foundIndex === -1) {
39            this.rForm.controls.saveFavoritesVisible.setValue(true);
40        }
41    }
42}
43}
```

e. Capturas después.



The screenshot shows a mobile application interface with a dark background. At the top, there are three colored dots (red, yellow, green) followed by a navigation bar with icons for back, forward, and search. Below the navigation bar is a large black rectangular area containing white text. The text is a code snippet for a method named `displayTouristOptions()`. The code uses ES6 syntax and includes several conditional statements and function calls. The text is numbered from 1 to 25 on the left side.

```
1  displayTouristOptions(): void {
2      const idLoggedUser = this.userState$.user?.id;
3
4      const user = this.getParsedUser();
5      const view = this.getParsedView();
6      const activity = this.getParsedActivity(idLoggedUser);
7
8      if (user && user.isTourist) {
9          this.idActivitiesUserFavorites = user.favorites;
10         const isFavorite = this.isFavorite(user.favorites);
11
12         if (view === 'myActivities') {
13             this.setButton('myActivities', 'delete');
14         } else if (view === 'favorites') {
15             this.setButton('favorites', 'delete');
16         } else {
17             if (activity.notFull && !activity.userSigned) {
18                 this.setButton('myActivities', 'add');
19             }
20             if (!isFavorite) {
21                 this.setButton('favorites', 'add');
22             }
23         }
24     }
25 }
```

```
1  getParsedUser(): any | null {
2    const isNotNull = this.userState$.user !== null;
3    if (isNotNull) {
4      return {
5        id: this.userState$.user?.id,
6        isTourist:
7          this.userState$.user?.profile.type === userTypes.Tourist.toString(),
8        favorites: this.userState$.user?.profile.favorites,
9      };
10   } else {
11     return null;
12   }
13 }
14 }
15
16 getParsedView(): string {
17   if (this.eFilterType === FilterType.myActivitiesFilter.toString()) {
18     return 'myActivities';
19   } else if (this.eFilterType === FilterType.favoritesFilter.toString()) {
20     return 'favorites';
21   }
22 }
23
24 getParsedActivity(idUser: number): any | null {
25   if (this.activity) {
26     return {
27       notFull: this.activity.peopleRegistered < this.activity.limitCapacity,
28       userSigned: this.activity.signUpUsers.includes(idUser),
29     };
30   } else {
31     return null;
32   }
33 }
34
35 setButton(view: string, action: string): void {
36   if (view === 'myActivities' && action === 'delete') {
37     this.rForm.controls.myActivityDeleteVisible.setValue(true);
38   } else if (view === 'favorites' && action === 'delete') {
39     this.rForm.controls.deleteFavoritesVisible.setValue(true);
40   } else if (view === 'myActivities' && action === 'add') {
41     this.rForm.controls.myActivitySignUpVisible.setValue(true);
42   } else if (view === 'favorites' && 'add') {
43     this.rForm.controls.saveFavoritesVisible.setValue(true);
44   }
45 }
46
47 isFavorite(activitiesId: number[]): boolean {
48   const foundIndex = activitiesId.findIndex((x) => x === this.activity.id);
49   if (foundIndex !== -1) {
50     return true;
51   } else {
52     return false;
53   }
54 }
```

4. Ejemplo 4. Dead code

- a. **Componente(s):** header, footer, (muchos otros componentes)
- b. **Método:** constructor, init, e imports (se importan muchos módulos que luego no se utilizan)
- c. **Explicación:** vamos a eliminar código no utilizado en nuestro proyecto, en este caso por ejemplo tenemos en el componente footer un constructor que no se utiliza, y en el componente cabecera un método init que tampoco. Así como a lo largo de todo el proyecto tenemos gran cantidad de imports de diversos módulos que no estamos utilizando en ningún momento y que por lo tanto suprimimos.
- d. **Capturas antes.**

```
import { Component, OnInit } from '@angular/core'; 103.8K (gzipped: 32.7K)
import {
  ValidatorFn,
  FormBuilder,
  FormControl,
  FormGroup,
  Validators,
} from '@angular/forms'; 63.2K (gzipped: 13.5K)
import { ActivatedRoute, Router } from '@angular/router'; 91.8K (gzipped: 23.2K)
import { AppState } from 'src/app/app.reducers';
import { Store } from '@ngrx/store'; 22.2K (gzipped: 6.1K)
import { LoginState } from '../reducers';
import * as LoginAction from '../actions';|      You, 3 days ago • rf
```

```
export class FooterComponent implements OnInit {
  public currentYear: string;

  constructor() {}
```

```
export class HeaderComponent implements OnInit {  
  
    loginState$: LoginState;  
    userState$: UserState;  
  
    constructor( private route: ActivatedRoute , public router:  
    Router,  
                e  
                private store: Store<AppState>){}  
  
    ngOnInit(): void {  
        this.store.select('login').subscribe(login => this.  
        loginState$ = login);  
        this.store.select('user').subscribe(user => this.userState$  
= user);  
    }  
  
    // Se inicializa el component  
    init(): void {}
```

e. Capturas después.

```
export class FooterComponent implements OnInit {  
    public currentYear: string;
```

```
import { Component, OnInit } from '@angular/core';
import { Router } from '@angular/router';
import { AppState } from 'src/app/app.reducer';
import { Store } from '@ngrx/store';
import { LoginState } from '../../login/reducer';
import * as LoginAction from '../../login/action';
import { UserState } from '../profile/reducer';
s
@Component({
  selector: 'app-',
  templateUrl: 'header.component.html',
  styleUrls: ['./header.component.css'],
})
s
export class HeaderComponent implements OnInit {
  loginState$: LoginState;
  userState$: UserState;

  constructor(public router: Router, private store: Store<AppState>) {}

  ngOnInit(): void {
    this.store.select('login').subscribe((login) => (this.loginState$ = login));
    this.store.select('user').subscribe((user) => (this.userState$ = user));
  }
}
```

5. Ejemplo 5. Long import

- a. **Componente(s):** user.reducet.ts
- b. **Método:** alias import.
- c. **Explicación:** técnicamente no es un code smell como los que hemos estudiado dentro de todas estas listas, pero sí que conviene cambiar la masiva importación de módulos utilizando alias que nos ayuden a leer mejor nuestro código.

d. Capturas antes.

```
import {  
  getLoginUser,  
  getLoginUserSuccess,  
  getLoginUserFailure,  
  formatUserSuccess,  
  createUser,  
  createUserSuccess,  
  createUserFailure,  
  updateUser,  
  updateUserSuccess,  
  updateUserFailure,  
  getFavoriteUserActivitiesStorageSuccess,  
  getFavoriteUserActivitiesStorageFailure,  
  setFavoriteUserActivitiesStorage,  
  setFavoriteUserActivitiesStorageSuccess,  
  setFavoriteUserActivitiesStorageFailure,  
  updateUserEducation,  
  updateUserEducationSuccess,  
  updateUserEducationFailure,  
  deleteUserEducation,  
  deleteUserEducationSuccess,  
  deleteUserEducationFailure,  
  addUserEducation,  
  addUserEducationSuccess,  
  addUserEducationFailure,  
  updateUserLanguage,  
  updateUserLanguageSuccess,  
  updateUserLanguageFailure,  
  deleteUserLanguage,  
  deleteUserLanguageSuccess,  
  deleteUserLanguageFailure,  
  addUserLanguage,  
  addUserLanguageSuccess,  
  addUserLanguageFailure,  
}  
}
```

e. Capturas después.

```
import * as UserAction from './actions';
import { createReducer, on } from '@ngrx/store';
import { User } from '../models/user';

export interface UserState {
  user: User;
  error: string | null;
  pending: boolean;
}

export const initialState: UserState = {
  user: null,
  error: null,
  pending: false,
};

const _userReducer = createReducer(
  initialState,
  on(UserAction.getLoginUser, (state) => ({
    ... state,
    error: null,
    pending: true,
})),
  on(UserAction.getLoginUserSuccess, (state, action) => ({
    ... state,
    user: action.user,
    error: null,
    pending: false,
})),
```

6. Ejemplo 6. Método duplicado

- a. Componente(s): profile.component.ts
- b. Método: onClickDeleteLanguage() y onClickDeleteEducation()
- c. Explicación: tenemos dos métodos que son prácticamente iguales, por lo tanto vamos a generalizar, extrayendo un método y haciendo pull up del método lo convertiremos en uno.

d. Capturas antes.

```
onClickDeleteLanguage(languageId: any): void {
    // Se solicita confirmación de eliminación
    if (confirm('Are you sure to delete this language?')) {
        const languages = this.user.languages;
        const index = this.user.languages.findIndex(
            (language) => language.uid === languageId
        );
        if (index === -1) {
            alert('Error language not found');
            return;
        }
        // Se elimina la lengua de la colección
        languages.splice(index, 1);
        // Se actualiza el usuario
        this.store.dispatch(UserAction.deleteUserLanguage({ user: this.user }));
    }
}

// Se recoge la pulsación sobre el botón de borrar educación
onClickDeleteEducation(educationId: any): void {
    // Se solicita confirmación de eliminación
    if (confirm('Are you sure to delete this education?')) {
        const educations = this.user.educations;
        const index = this.user.educations.findIndex(
            (education) => education.uid === educationId
        );
        if (index === -1) {
            alert('Error education not found');
            return;
        }
        // Se elimina la educación de la colección
        educations.splice(index, 1);
        // Se actualiza el usuario
        this.store.dispatch(UserAction.deleteUserEducation({ user: this.user }));
    }
}
```

e. Capturas después.

```
onDeleteButton({ button, id }: any) {
  if (confirm('Are you sure to delete this education?')) {
    const data =
      button === 'language' ? this.user.languages : this.user.educations;
    const index = data.findIndex((el) => el.uid === id);
    if (index === -1) {
      alert(`Error ${button} not found`);
      return;
    }
    data.splice(index, 1);
    if (button === 'language') {
      this.store.dispatch(UserAction.deleteUserLanguage({ user: this.user }));
    } else {
      this.store.dispatch(
        UserAction.deleteUserEducation({ user: this.user })
      );
    }
  }
}
```