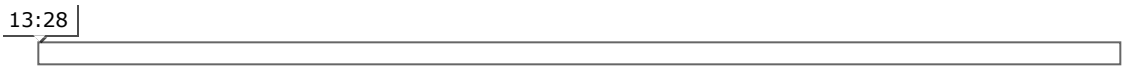
Watch: ✋ **Qué Refactoring NO hacer: Un gran poder conlleva una gran responsabilidad (única)**



✋ Qué Refactoring NO hacer: Un gran poder conlleva una gran responsabilidad (única)

⚠️ ¡Atención! En este video veremos un ejemplo de cómo NO deberíamos hacer un refactoring de large class, para explicar finalmente por qué consideramos que tendríamos que evitarlo 🙏

Clase `StudentGradeCalculator` (Estado inicial):

```
export class StudentGradeCalculator {
    allYearsTeachers = {
        2020: [
            ["Josefina", true],
            ["Edonisio", true],
            ["Edufasio", false],
        ],
        2019: [
            ["Eduarda", false],
            ["Abelardo", false],
            ["Francisca", false],
        ]
    };

    constructor(yearToCalculate) {
        this.yearToCalculate = yearToCalculate;
    }

    calculate(examGrades, hasReachedMinimumClasses) {
        if (examGrades.length !== 0) {
            let hasToIncreaseOneExtraPoint = false;

            for (let [year, teachers] of
Object.entries(this.allYearsTeachers)) {
                if (!(this.yearToCalculate != year)) {
                   for (let teacher of teachers) {
                     // ...
                   }
                  // more...
                }
            }
        }
    }
```

Seguimos con el ejemplo de StudentGradeCalculator en Javascript, en el que habíamos identificado el code smell de large class. Lo primero que encontramos en esta clase es que se está encargando de almacenar el listado de todos los profesores de cada año con su asignación como profesor benevolente o no

Como estamos en 'modo flipado' con el refactoring, lo primero que hemos decidido es establecer distintos niveles de abstracción, dejando nuestro StudentGradeCalculator como clase abstracta bajo la cual cuelguen dos clases concretas, una para cada año 👨‍👧

Clase `StudentGradeCalculator` (Extayendo a subclases):

```
export class StudentGradeCalculator {
  // ...
    thisYearTeachers() {
      throw 'This method should be implemented';
    }
    constructor(yearToCalculate) {
        this.yearToCalculate = yearToCalculate;
    }

    calculate(examGrades, hasReachedMinimumClasses) {
        if (examGrades.length !== 0) {
            let hasToIncreaseOneExtraPoint = false;

            for (let [year, teachers] of
Object.entries(this.allYearsTeachers)) {
                if (!(this.yearToCalculate != year)) {
                   for (let teacher of teachers) {
                     // ...
                   }
                  // more...
            }
        }
    }
```

Lo que estamos haciendo de fondo es, desde la clase principal que contiene la lógica común, identificar las particularidades que contiene para distintas condiciones y extraerlas a subclases. Este patrón se conoce como **Template Method**: Partimos de un metodo padre/plantilla que nos obligará a implementar en los hijos la lógica de ese 'hueco' que deja abierto para rellenar 🧩

En este ejemplo, ese 'hueco' a implementar será el de la lógica que determina si ese año ha habido algún profesor benevolente. En el caso de javascript, la única manera que tenemos de emular un método abstracto es que éste lance un error en la clase padre, forzando a que sea sobrescrito en las subclases

Clase `StudentGrade2019Calculator`:

```
export class StudentGrade2019Calculator extends
StudentGradeCalculator {

    thisYearTeachers() {
      return [
            ["Eduarda", false],
            ["Abelardo", false],
            ["Francisca", false],
        ];
    }
}
```

Ahora los clientes sólo podrán implementar una de las subclases, que tal y como vemos, la única lógica que recogen es la de implementar este *thisYearTeachers*() para retornar el listado de profesores del año en concreto

Clase `StudentGradeCalculator` (Sustituyendo la lógica de las subclases):

```
export class StudentGradeCalculator {

    thisYearTeachers() {
      throw 'This method should be implemented';
    }

    calculate(examGrades, hasReachedMinimumClasses) {
        if (examGrades.length !== 0) {
            let hasToIncreaseOneExtraPoint = false;
            for (let teacher of this.thisYearTeachers()) {
              // ...
            }
          // more...
    }
```

Ahora ya no es necesario iterar sobre los distintos años e ir comprobando si coincide con el año actual, puesto que la implementación de *thisYearTeachers*() ya nos devuelve el listado de profesores correspondiente, por lo que podemos quitarnos de encima tanto el bucle for como el bloque if que gestionaba esa lógica y llamar directamente a nuestro método para comprobar si alguno de ellos es

benevolente (Recordad que ahora habrá que actualizar los tests sustituyendo las instanciaciones de la clase padre por las de las hijas)

Pero es que además, al no necesitar que nos pasen el año actual al instanciar la clase, ya que eso depende de la subclase implementada, podemos ahorrarnos la implementación del constructor y despejar esas líneas 🧹

- **CodelyTV Tip** ☝️: En la inmensa mayoría de los lenguajes, podemos ahorrarnos definir los constructores por defecto que no tienen argumentos de entrada

🤯 Conclusiones: Por qué no hacer este refactoring

El primer inconveniente que vemos a la hora de aplicar esta técnica es que resulta **muy difícil tracear** el código a lo largo de esa jerarquía de clases, el código se vuelve mucho menos explícito

Por otro lado, también nos encontramos con que la **testabilidad del código** se hace mucho **más costosa**, al forzarnos a pasar por tóda la lógica de la clase padre cuando queramos testear la de cualquiera de las subclases

Finalmente, en este caso tendremos que estar pendientes de crear una nueva subclase para nueva condición, puesto que de lo contrario fallará la aplicación cuando queramos ejecutarla. Si además de esto, cada condición/subclase tiene otras lógicas particulares que deben aplicarse en la clase padre, será muy fácil que acabemos generando bugs al tratar de encajar una lógica que no es común

¿Alguna Duda?

Si tienes alguna duda sobre el contenido del video o te gustaría compartir cualquier sugerencia recuerda que puedes en abrir una nueva

discusión justo aquí abajo 👇👇👇

¡Nos vemos en el siguiente video: 🤝 Extract Class Refactoring: Encapsulando responsabilidades!