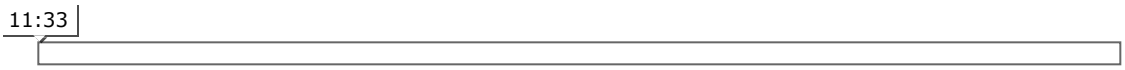
Watch: 🤝 **Extract Class Refactoring: Encapsulando responsabilidades**



🤝 Extract Class Refactoring: Encapsulando responsabilidades

Ya hemos visto los pros y contras del Template Method en este caso de refactor, así que pasamos a ver la opción que nosotros consideramos más beneficiosa para refactorizar nuestra large class

💬 Añadiendo semántica

Recordemos que partíamos de este escenario de clase demasiado larga

Clase `Teachers`:

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

El primer pasito 👣 que vamos a dar será **extraer** a una clase de tipo **colección tipada** a parte todos los datos de los profesores que teníamos en StudentGradeCalculator. Con esto, no sólo ganamos en semántica, sino que damos pie a poder ubicar toda la lógica referente a los profesores en esta clase

💉 Inyección de dependencia y empujando la lógica

Clase `StudentGradeCalculator` (inyectando Teachers):

```
export class StudentGradeCalculator {
    constructor(yearToCalculate, teachers) {
        this.yearToCalculate = yearToCalculate;
        this.teachers        = teachers;
    }
    // more...
}
```

Al haber extraído esta clase, será ahora necesario pasársela por constructor a StudentGradeCalculator, para asigarle estos datos al campo de clase. Por supuesto, este cambio nos obligará a modificar todas las instanciaciones que tuvieramos previas para añadir este nuevo parámetro 👷

Clase `Teachers`:

```
export class Teachers {
    // more...
    isThereAnyBenevolent(yearToCalculate) {
        let isThereAnyBenevolent = false;

        for (let [year, teachers] of
Object.entries(this.allYearsTeachers)) {
            if (!(yearToCalculate != year)) {
                for (let teacher of teachers) {
                    if (teacher[1] != true) {
                        continue;
                    }
                    isThereAnyBenevolent = true;
                }
            } else {
                continue;
            }
        }
        return isThereAnyBenevolent;
    }
}
```

Una vez que contamos con la clase `Teachers`, resulta mucho más sencillo ver qué lógica le pertenecerá, como es el caso de la comprobación que realizábamos para saber si había algún profesor benevolente en el año dado, por lo que será algo que sacaremos de StudentGradeCalculator. Como vemos en el ejemplo, la semántica dentro de este código también ha cambiado al ubicarla dentro de su propio modelo

Con esto además abordamos otro code smell conocido como **Feature Envy**: Nuestro servicio está solicitando datos al modelo para realizar una serie de cómputos que realmente pertenecen al propio modelo,

recordemos que la orientación a objetos persigue una alta cohesión y un bajo acoplamiento

Si bien hemos llegado a un buen punto en nuestro refactor, cabe pensar que al final, nuestra clase Teachers no deja de ser un conjunto de datos, por lo que presumiblemente serán introducidos por algún mecanismo de Entrada/Salida. Si queréis profundizar más sobre cómo introducir el uso de repositorios os recomendamos revisar los cursos de Principios SOLID y de Arquitectura Hexagonal

Como veníamos comentando, podemos concluir que este refactoring aboga por el principio de composición sobre herencia, aumentando la trazabilidad y testabilidad del código y ganando bastante en declaraciones explícitas

¿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 la próxima lección video: 🐒 Primitive Obsession Code Smell!