Watch: 🤹 **Refactor Extract Method ayudándonos del IDE y problemas relacionados**

# 🤹 Refactor Extract Method ayudándonos del IDE y problemas relacionados

Para empezar a refactorizar el código que vimos en el video anterior vamos a ayudarnos de algunas de las herramientas y utilidades que nos ofrece nuestro IDE de confianza, en este caso utilizaremos IntelliJ, por lo que os recomendamos encarecidamente que le echeis un ojo al curso de IntelliJ IDEA 👌

Una vez decididos a refactorizar nuestro código, hay dos cuestiones sumamente importantes que debemos considerar desde el inicio: comprender qué es lo que hace el código y disponer de una cobertura de tests que nos indiquen si hemos roto algo al tratar de mejorarlo. En este caso, y como vemos aquí, contamos con varios tests que ya de por sí nos dejan entrever que hay muchas casuísticas a validar dentro de ese método, con lo que ir a ciegas hubiera sido todo un peligro 🤯

Clase `Pair`:

```
public final class Pair<FirstType, SecondType> {
    final private FirstType first;
    final private SecondType second;

    public Pair(FirstType first, SecondType second) {
        this.first  = first;
        this.second = second;
    }

    public FirstType first() {
        return first;
    }

    public SecondType second() {
        return second;
    }
}
```

Para seguir poniéndonos en situación, esta clase `Pair` que veremos en el método no es sino un agrupador que contiene dos tipos genéricos, en el caso que nos atañe se viene utilizando para recoger pares de notas con sus respectivos pesos en la evaluación, así como para la asociación de profesores con su catalogación como benevolentes o no 👩‍🏫 En otros lenguajes menos tipados lo que tendríamos en su lugar sería simplemente arrays con pares de valores `[["Rafa", true],["Javi", false]]`

Una vez ubicados ¿Por dónde empezamos? Es un fallo habitual querer ponernos a refactorizarlo todo de una vez, y es que esto, además de aumentar el riesgo de equivocarnos, no nos permitiría seguir una cierta

metodología de refactoring, que es justamente lo que queremos lograr 🙌 Por eso iremos pasito a pasito y empezaremos extrayendo métodos

Clase `StudentGradeCalculator`:

```
public float calculateGrades(final List<Pair<Integer, Float>>
examsGrades, final boolean hasReachedMinimumClasses) {
    if (!examsGrades.isEmpty()) {
        boolean hasToIncreaseOneExtraPoint = false;

        for (Map.Entry<Integer, List<Pair<String, Boolean>>>
yearlyTeachers : allYearsTeachers.entrySet()) {
            if (!(yearToCalculate != yearlyTeachers.getKey())) {
                List<Pair<String, Boolean>> teachers =
yearlyTeachers.getValue();

                for (Pair<String, Boolean> teacher : teachers) {
                    if (teacher.second() != true) {
                        continue;
                    }
                    hasToIncreaseOneExtraPoint = true;
                }
            } else {
                continue;
            }
        }
    // more...
     if (hasToIncreaseOneExtraPoint) {
        return Float.min(10f, gradesSum + 1);
    }
    // more..
    }
```

Una de las cosas a considerar a la hora al analizar el código es ¿Tiene sentido que esté aquí o debería estar fuera? En este ejemplo vemos que la lógica para determinar si hay que añadir un punto extra a la nota final se está entrelazando innecesariamente, a pesar de guardar una relación con el caso de uso, por lo que tendría mucho sentido llevárnoslo a un método a parte 🧹

Al movernos a la definición de esa variable `hasToIncreaseOneExtraPoint` y hacer *Cmd+clic* sobre ella podemos ver que solo se lee en un punto, pero más importante aún: sólo se

modifica una vez. De editarse en varios puntos, habría que considerar si debemos extraer dos métodos distintos o unar de algún modo la lógica donde se modifica, pero en este caso podremos sencillamente extraer esta lógica a un método fuera 👌

Para extraerlo, bastaría con seleccionar la porción de texto en cuestión, hacer click derecho para abrir el menú de opciones y buscar el bloque *Refactor > Extract Mehod...*. Si queremos hacerlo algo más pro 😎 podríamos utilizar el atajo de teclado *Ctrl + T* que nos abrirá el menú de 'Refactor This', el cual, como deduciréis nos mostrará diferentes opciones para refactorizar esa porción de código seleccionada. También podemos utilizar el atajo *Ctrl + Option + M* que nos ejecutará directamente la opción de extraer a un método

Sea como sea, esto nos llevará al menú de configuración donde el IDE ya nos estará deduciendo el tipo de retorno del método, los parámetros de entrada si los necesitara y la visibilidad (recordad que los detalles de implementación del servicio siempre deberían tener la menor visibilidad posible). Al retornar un booleano, el propio IDE nos añade el prefijo 'is' de forma bastante semántica

Clase `StudentGradeCalculator` (Refactor paso 1):

```
public float calculateGrades(final List<Pair<Integer, Float>>
examsGrades, final boolean hasReachedMinimumClasses) {
    if (!examsGrades.isEmpty()) {
        boolean hasToIncreaseOneExtraPoint =
hasToIncreaseOneExtraPoint();
    // more...
     if (hasToIncreaseOneExtraPoint) {
        return Float.min(10f, gradesSum + 1);
    }
     // more...
    }
}
```

Ahora nuestro método ya resulta algo más sencillo de leer, reduciendo todo el código anterior a una única llamada al método para asignarle el valor de retorno a la variable, y si ejecutamos los test veremos que siguen pasando correctamente

Algo que no debemos pasar por alto al llevar a cabo este tipo de técnicas es preguntarnos ¿Este nuevo método está realizando algún **side effect** además de devolver un valor? Y es que resulta muy peligroso el hecho de modificar el estado de la clase sin que **tengamos constancia explícita de qué está pasando**, estaríamos ejecutando el método pensando que va a producir un determinado efecto cuando en realidad está haciendo mucho más por detrás.

En su lugar, deberíamos intentar que las **modificaciones de estado** queden siempre **fuera del** código sobre el que vamos a aplicar el **extract method**. Un truquillo útil en lenguajes tipados es el hecho de usar el tipo de retorno void/null para especificar justamente métodos que van a producir side effects 🤷

Clase `StudentGradeCalculator` (Refactor paso 2):

```
public float calculateGrades(final List<Pair<Integer, Float>>
examsGrades, final boolean hasReachedMinimumClasses) {
    if (!examsGrades.isEmpty()) {
    // more...
     if (hasToIncreaseOneExtraPoint()) {
        return Float.min(10f, gradesSum + 1);
    }
     // more...
    }
}
```

Finalmente daremos un pasito más en la mejora del código gracias al *inline*, que consiste en sustituir las llamadas a una variable por el texto de su derecha (su asignación). En este caso, como vimos al principio, sólo había un punto en el que se estuviera consultando el valor de esa variable, y es precisamente donde estaremos llamando al método que acabamos de extraer. Este tipo de métodos también se conocen como 'Query method' porque justamente son llamados para consultar un valor 🤔

# ¿Alguna Duda?

Si tienes cualquier duda sobre el contenido del video o te apetece compartir cualquier sugerencia recuerda que puedes en abrir una nueva discusión justo aquí abajo 👇👇👇

Después de estos pasitos, y viendo lo que aún tenemos en este StudentGradeCalculator ¿Qué próxima extracción crees que podríamos llevar a cabo?

¡Nos vemos en el siguiente video: 💿 Jugando a ser puristas: Contiene trazas de programación funcional!