Watch: 💿 **Jugando a ser puristas: Contiene trazas de programación funcional**

💿 Jugando a ser puristas: Contiene trazas de programación funcional

Seguimos puliendo un poco más nuestro método y ahora veremos cómo hacerlo siendo un poco mas 'puristas' al darle un toque de programación funcional 🎩

🕵️ Identificando el code smell

Clase `StudentGradeCalculator` (estado inicial):

```
public float calculateGrades(final List<Pair<Integer, Float>>
examsGrades, final boolean hasReachedMinimumClasses) {
    if (!examsGrades.isEmpty()) {
        float gradesSum       = 0f;
        int  gradesWeightSum = 0;

        for (Pair<Integer, Float> examGrade : examsGrades) {
            gradesSum += (examGrade.first() * examGrade.second()
/ 100);
            gradesWeightSum += examGrade.first();
        }
    // more...
    }
```

Ponemos ahora el foco en este bucle *for* que vemos cómo está siendo aprovechado para hacer dos operaciones distintas: por un lado hacer el cálculo del sumatorio de notas y por otro el sumatorio del peso de dichas notas. Aunque este código podría aceptarse que es mas óptimo para la máquina, entendemos que para el coste computacional que supone no compensa la pérdida de legibilidad en el código

Clase `StudentGradeCalculator` (desdoblando bucle for):

```
public float calculateGrades(final List<Pair<Integer, Float>>
examsGrades, final boolean hasReachedMinimumClasses) {
    if (!examsGrades.isEmpty()) {
        float gradesSum       = 0f;
        for (Pair<Integer, Float> examGrade : examsGrades) {
            gradesSum += (examGrade.first() * examGrade.second()
/ 100);
        }

        int  gradesWeightSum = 0;
        for (Pair<Integer, Float> examGrade : examsGrades) {
            gradesWeightSum += examGrade.first();
        }
    // more...
    }
```

Ahora que hemos dado este pequeño 'paso atrás', resulta mucho más claro ver cual será nuestro próximo refactor, y es por eso que resulta interesante llevarnos lo más cerca de la inicialización de la variable ese fragmento de código en el cual la acabamos rellenando

Ƛ Extrayendo a funciones lambda

En paso para transformar las iteraciones a funciones lambda lo primero que haremos será extraer ese código a métodos a parte, tal y como vimos en el video anterior, con lo que ya de base estaremos ganando bastante en legibilidad 🔍

Clase `StudentGradeCalculator` (método gradesWeightSum):

```
private int gradesWeightSum(List<Pair<Integer, Float>>
examGrades){
    int gradesWeightSum = 0;
    for (Pair<Integer, Float> examGrade : examsGrades) {
        gradesWeightSum += examGrade.first();
    }
}
```

Para avanzar en nuestro refactor vamos a aplicar lo que se conoce como '*parallel changes*', en lugar de ponernos a cambiar el código directamente dentro de toda la maraña, nos sacamos en primer lugar el código a un bloque a parte donde poder trastear más libremente y con mucha menos carga mental que donde estaba originalmente. Además, vamos a duplicar el método extraído y trabajar en principio sobre la copia, de modo que ahora mismo nuestro código seguiría funcionando perfectamente

Clase `StudentGradeCalculator` (método gradesWeightSum):

```
private int gradesWeightSum(List<Pair<Integer, Float>>
examGrades){
  return
examGrades.stream().map(Pair::first).reduce(Integer::sum).get();
}
```

El método *map*() nos va a permitir convertir un conjunto A en otro conjunto B aplicando una función de transformación, pero a nosotros sólo nos interesa la primera parte de cada `Pair`, con lo que le pasaremos únicamente el elemento 'first' y a continuación ejecutar la operación de acumulación gracias al método *reduce*() que como su nombre indica, va a reducir todos los valores de la lista a uno sólo (en este caso sumándolos)

- Ojo 👀 al usar el método *stream*() junto al *reduce*() puede ser que haya o no un resultado y el propio IDE nos advertirá de que se espera que lo haya, por lo que en este caso estamos forzando a que siempre nos devuelva el valor, aunque esto podría devolvernos un error y sería adecuado gestionarlo además con algún recurso como el `Optional`

Clase `StudentGradeCalculator` (estado final):

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

        int  gradesWeightSum = gradesWeightSum();
    // more...
    }
```

Una vez que hemos sustituído las llamada al método, comprobado que esta solución funciona y los tests siguen pasando en verde podemos eliminar el código anterior y 'guardar partida' antes de seguir refactorizando el código 🎮

¡Recapitulando!:

- Hemos detectado los accesos que se hacen a esas variables dentro del método
- Agrupamos la lógica para ver con más claridad que porción de código podríamos extraer
- Uso de Query Methods con mayor semántica
- Estamos siendo explícitos respecto a lo que retornan los métodos

- Evitamos el paso de argumentos por referencia y la modificación invisible del estado

¿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 👇👇👇

¡Nos vemos en la próxima lección: 📜 Large Class Code Smell!