**Nombre**: Ricardo Millanao Burgos

En todos los pantallazos, la parte izquierda representa al código sin refactorizar, y el derecho al refactorizado.

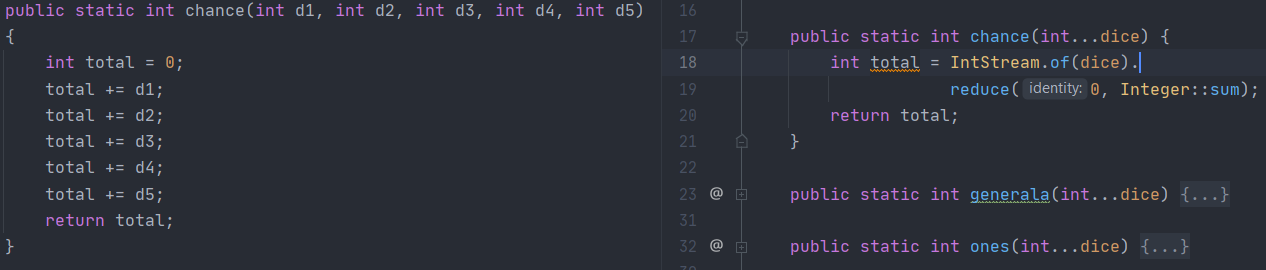
En todos los métodos que tenían parámetros se los cambio para que fuesen dinámicos y así ampliar las opciones de refactoring, por lo que no se repite como un refactoring especifico

# **Constructor**

**Modificador de acceso**: Por el momento nada indica el modificador de acceso “protected” en la variable “dice”, por lo que se cambia a “private”

**Constructor con cuerpo grande:** Las líneas de códigos para la asignación, se pueden ahorrar cambiando los parámetros a dinámicos, y así, solo se asignarían como un arreglo.

# **Chance()**



**Suma repetitiva**: Repetir sumas variable a variable se puede ahorrar con una expresión lambda que sume los términos.

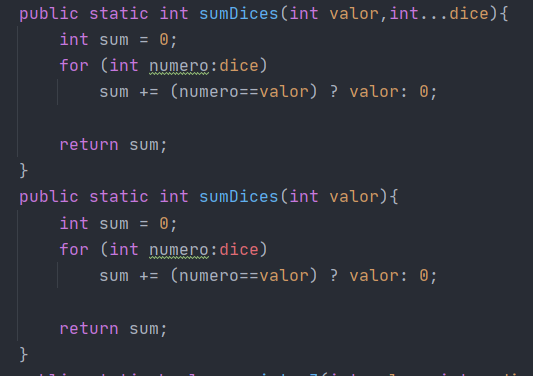
# **Generala()**

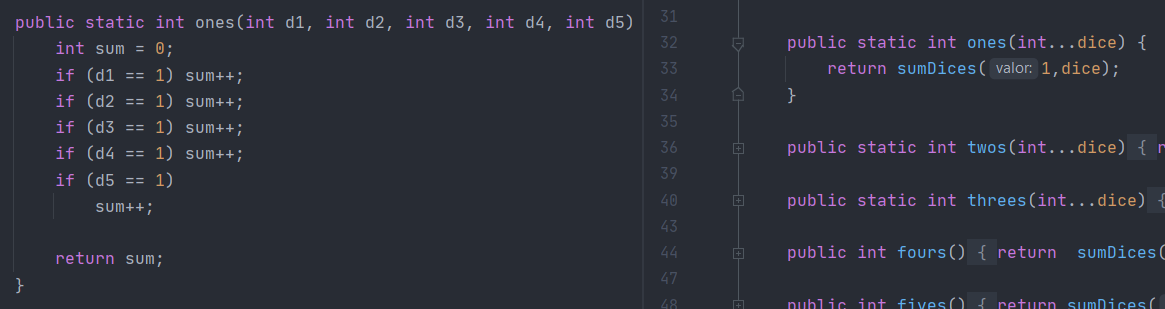
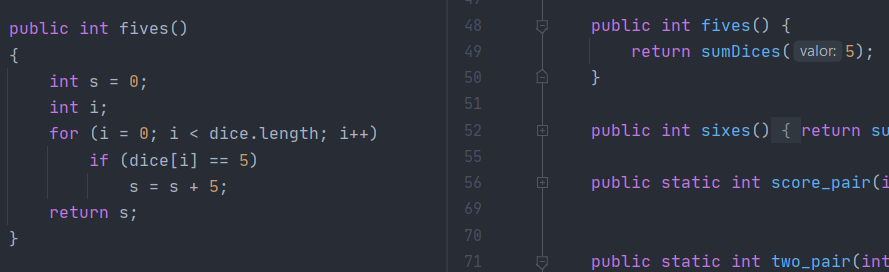
**Complejidad innecesaria:** Al recorrer con un doble For e insertar complejidad que no es necesario provoca una perdida de tiempo al entender el código. Esto se puede mejorar obteniendo un numero cualquiera de “dice”, luego multiplicarlo por el largo del arreglo “dice” y para compararlo con el total, se reutiliza el método chance(), esto debería ser True si es que todos los números son iguales.

# **Ones()Twos()Threes()Fours(), Fives(), Sixes():**

Se agrupan estos métodos, ya que su refactorización es similar en todos. Para esta refactorización se utiliza dos métodos con casi el mismo cuerpo, pero se adapta según su sobrecarga de parámetros, en palabras simples, hacen lo mismo.

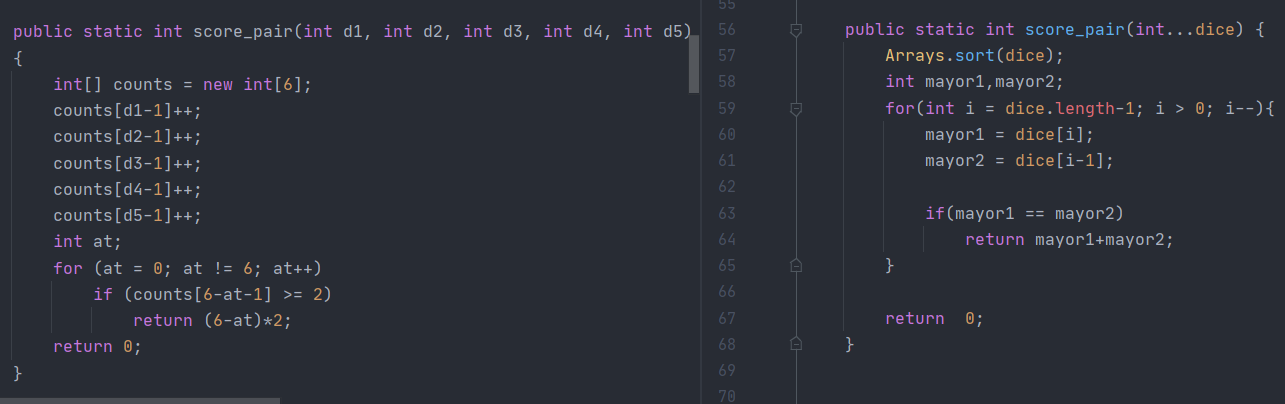
Estos métodos son:





**Sumas repetitivas y Código duplicado:** Estos métodos consistían en obtener una suma especifica a partir de un parámetro dependiendo el caso. Como era repetitivo se hicieron dos métodos ya hablados anteriormente: sumDices(). En este método se realiza una suma iterativa según sea el parámetro de cada metro en especial.

# **Score\_pair()**



**Complejidad:** El método funciona a partir de una estructura compleja, donde es claro que se esfuerza por acortar el código en vez de su legibilidad.

Para eso en la refactorización se va por etapa. Partiendo por ordenar el arreglo, iterarlo de mayor a menor para que sea más fácil su comparación y obtener el resultado.

**Operaciones repetitivas**: Se realizan operaciones repetitivas sumado con la complejidad.

**Nombres pocos descriptivos:** Se puede apreciar que “at” no apunta a nada.

# **Two\_pair()**

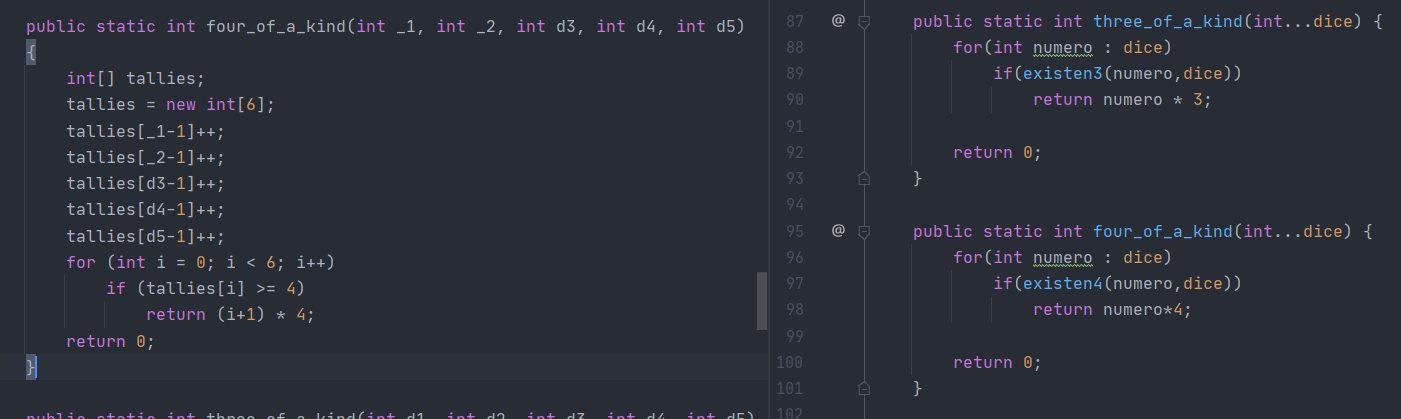
**Complejidad:** El código presenta una complejidad innecesaria, no presenta una estructura clara y poca información ligera para leer a simple vista.

**Operaciones repetitivas:** Aquellas operaciones complejas se hacen repetitivas.

La solución al problema anterior es como en el método anterior:

Ordenar el arreglo, realizar un ciclo que itere, comparar los números con sus sucesores, aquellos números que sean similares sumarlos a un acumulador. Y así que se repita el proceso hasta que no existan más pares que comparar.

# **Three\_of\_a\_kind() y Four\_of\_a\_kind()**

****

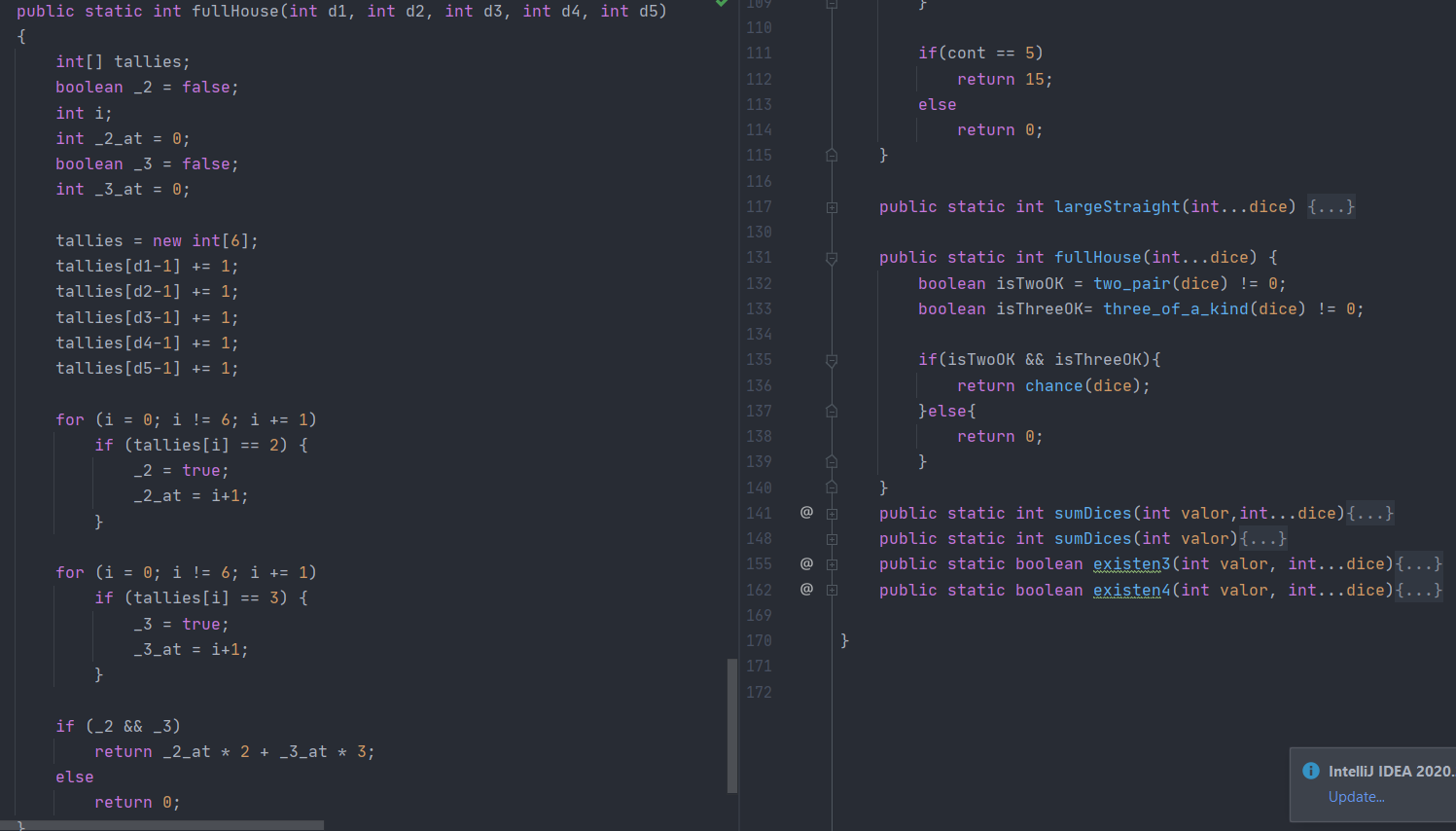
Los errores se repiten al igual que lo anteriores métodos.

Estos métodos comparten algo similar, y es que, se refactorizó en función de dos métodos que buscaban coincidencias. De encontrarse retornaba el numero encontrado, por la cantidad de veces según sea el método. Mientras que en el código no refactorizado realizaba operaciones repetitivas y engorrosas.

# **SmallStraight() y LargeStraight()**

Estos métodos son muy parecidos en comparar números consecutivos. Para aprovechar esto, se utiliza un ciclo For arreglado para que itere ordenamente junto a un arreglo ordenado para que así, si existe coincidencias estas se comparen de una forma óptima. Si existen coincidencias un acumulador las contará, para luego ser evaluado

# **FullHouse()**



La complejidad agresiva en este método hizo replantear todo, para realizarse de nuevo.

Aprovechando reutilizar código se crean dos banderas. Estas banderas luego serán comparadas. Si es que ambas son verdaderas se vuelve a reutilizar el código para la suma, si no son verdaderas retorna 0.

NOTA: Todas las pruebas unitarias han sido realizadas con éxito.