**ESCUELA SUPERIOR POLITECICA DEL LITORAL**

**Taller Refactoring**

**Grupo 1**

**Integrantes:**

**- Alcivar Castro José Emanuel**

**-Richard David Cevallos Apolo**

**-Ricardo Mitridates Zevallos Mosquera**

**-Guillermo Isaac Teran Mendoza**

**-Bryan Segovia**

**Link Repositorio:**

**- https://github.com/Jose-Alcivar-C/TallerRefactoringG1**

**Indice**

Data Class …………………………………………………………………………………………………. 3

Consecuencias …………………………………………………………………………………. 3

Técnicas usadas ……………………………………………………………………………….. 3

Código inicial ……………………………………………………………………………………. 3

Código refactorizado ……………………………………………………………………….. 4

Innapropiate Intimacy ………………….……………………………………………………………. 5

Consecuencias …………………………………………………………………………………. 5

Técnicas usadas ……………………………………………………………………………….. 5

Código inicial ……………………………………………………………………………………. 5

Código refactorizado ……………………………………………………………………….. 5

Speculative Generality ………………….……………………………………………………………. 6

Consecuencias …………………………………………………………………………………. 6

Técnicas usadas ……………………………………………………………………………….. 6

Código inicial ……………………………………………………………………………………. 6

Código refactorizado ……………………………………………………………………….. 6

Duplicate code ………………….………………………………………………………………….……. 7

Consecuencias …………………………………………………………………………………. 7

Técnicas usadas ……………………………………………………………………………….. 7

Código inicial ……………………………………………………………………………………. 7

Código refactorizado ……………………………………………………………………….. 7

Feature Envy …..………………….………………………………………………………………….……. 8

Consecuencias …………………………………………………………………………………. 8

Técnicas usadas ……………………………………………………………………………….. 8

Código inicial ……………………………………………………………………………………. 8

Código refactorizado ……………………………………………………………………….. 9

**Data class**

La clase “InformacionAdicionalProfesor” solamente contiene atributos que perfectamente pueden pertenecer directamente a la clase “Profesor”. En esta clase no hay métodos que contengan acciones adicionales, solamente contiene campos que pertenecen a la clase “Profesor”, podemos eliminar la clase “InformacionAdicionalProfesor” y mover esos atributos a la clase “Profesor”. Además, podemos ver que en ambas clases estos atributos están públicos (no están encapsulados), lo cual no debería ser así.

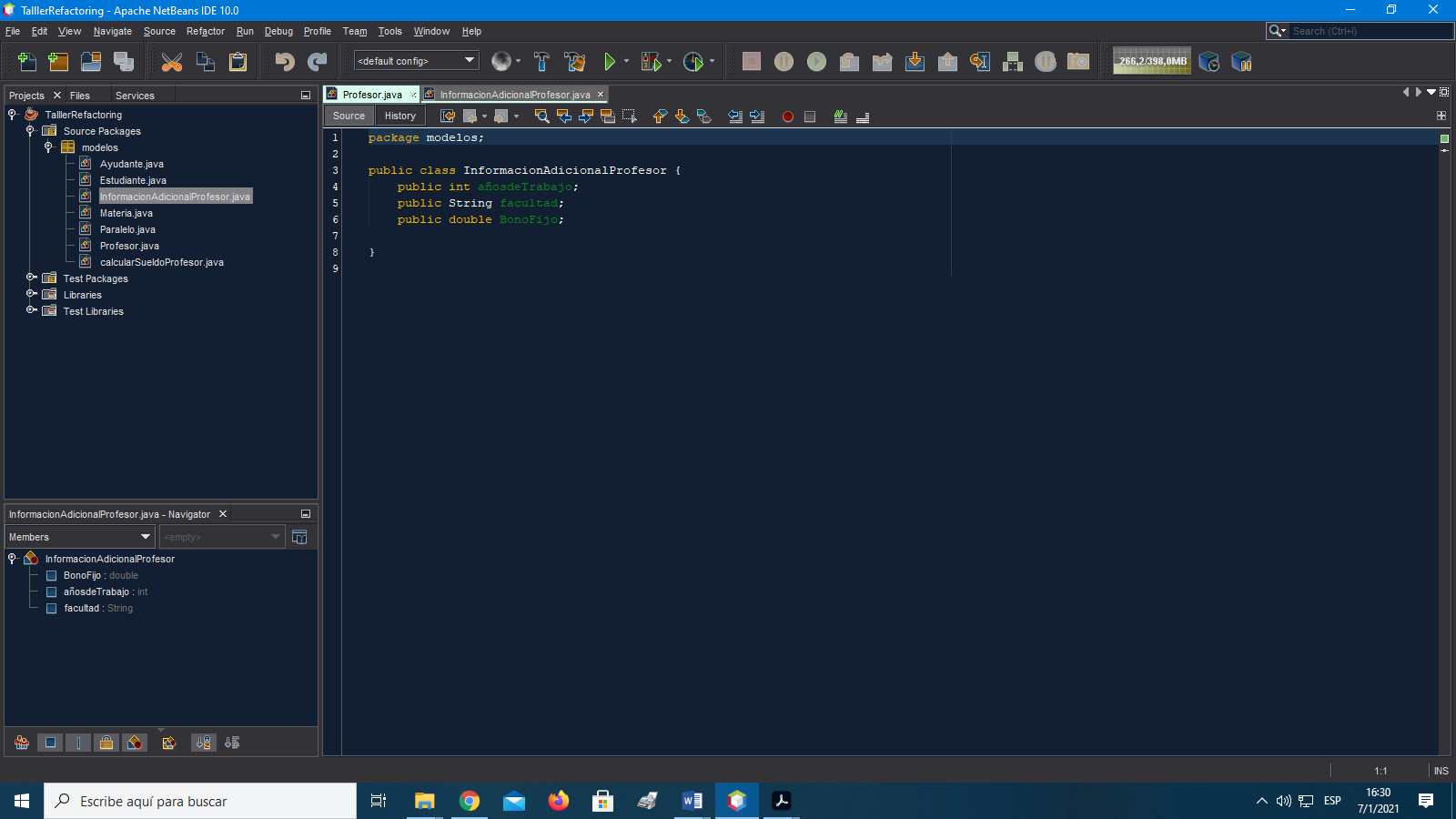
**Consecuencias**

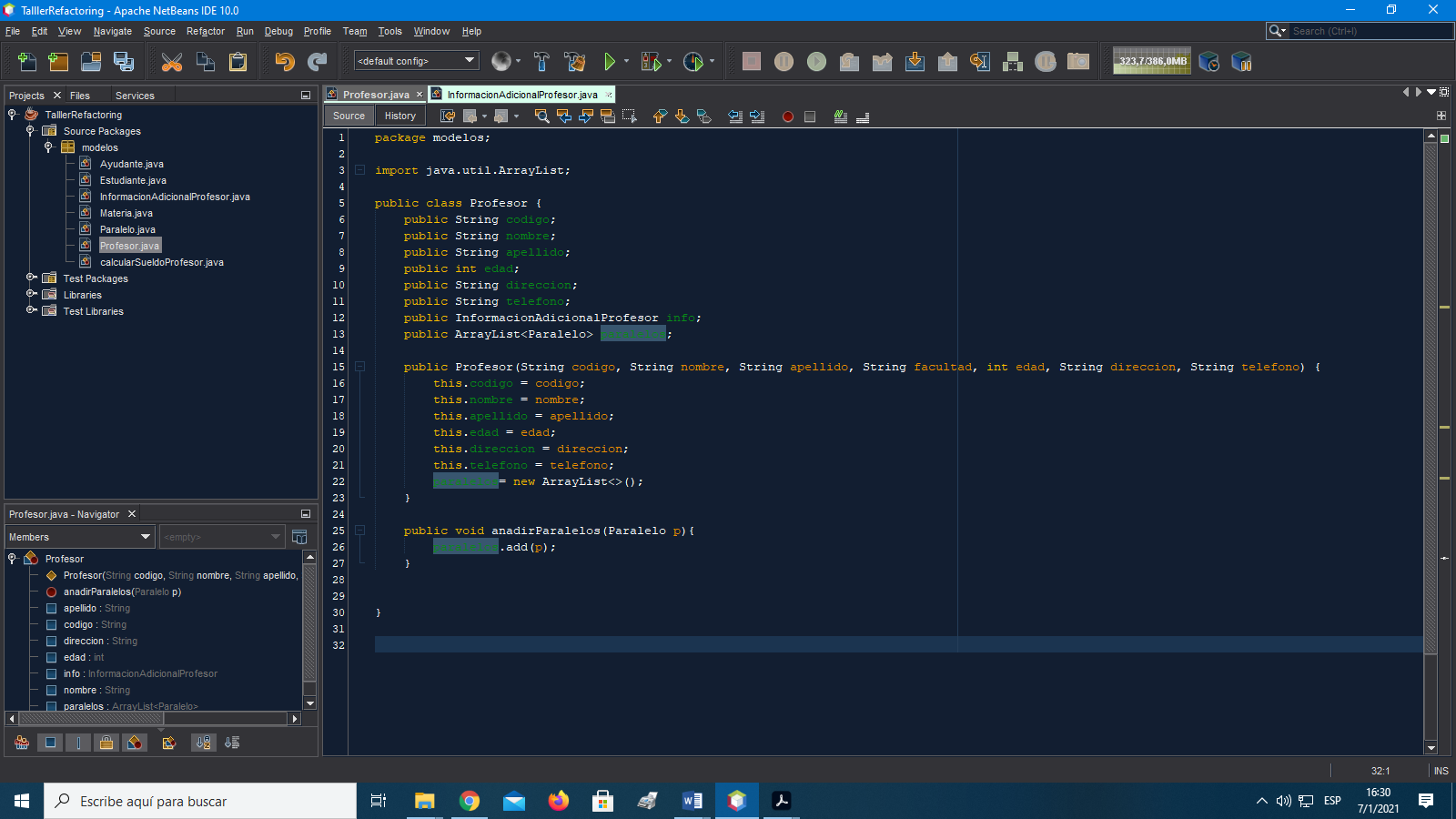
Sería necesario crear más métodos adicionales puesto que los atributos que se usarán se encuentran repartidos entre dos clases distintas, incluso corremos el riesgo de duplicar código en el programa.

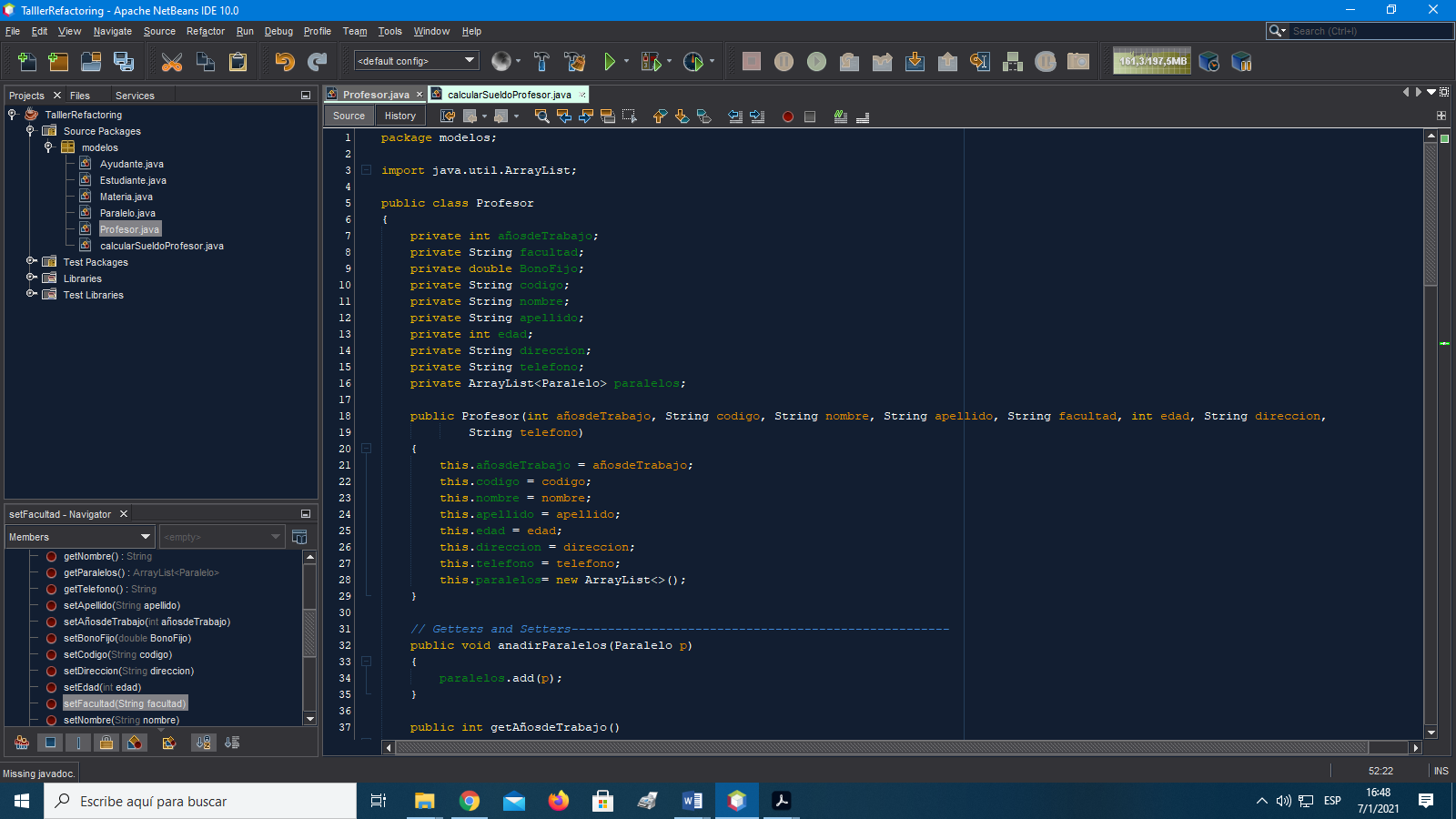
**Técnicas usadas**

- Move Fields

- Encapsulate Fields

** Código inicial**

****

**Código refactorizado**

**Innapropiate Intimacy**

La clase “calcularSueldoProfesor” contiene un método que accede directamente a los atributos de otra clase, además de eso, podemos notar que este método no tendría que estar en esta clase, debido a que perfectamente podría estar en la clase “Profesor” y tomar de ahí los atributos, esto nos permite eliminar la clase “calcularSueldoProfesor”, puesto que está de más.

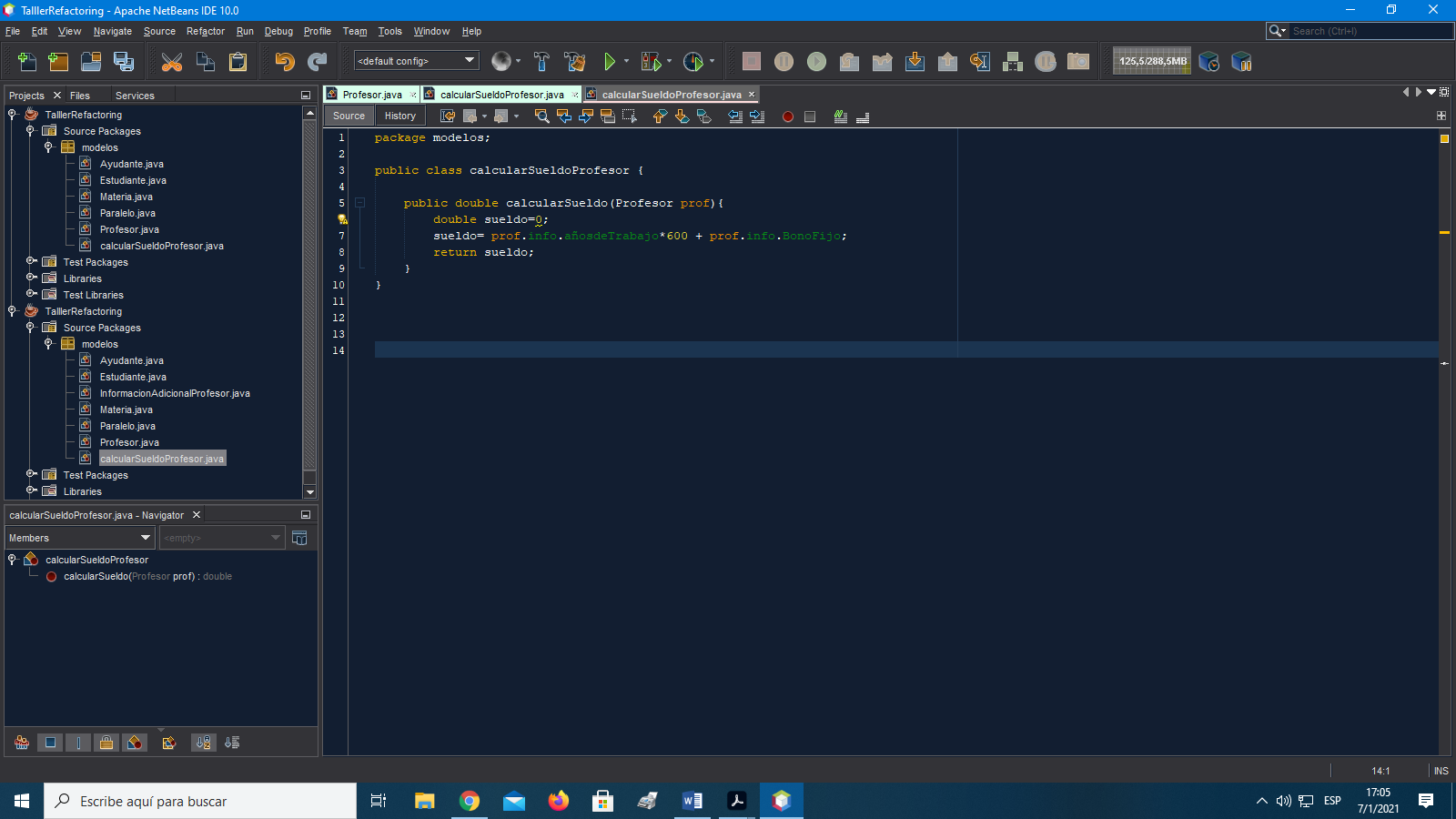
**Consecuencias**

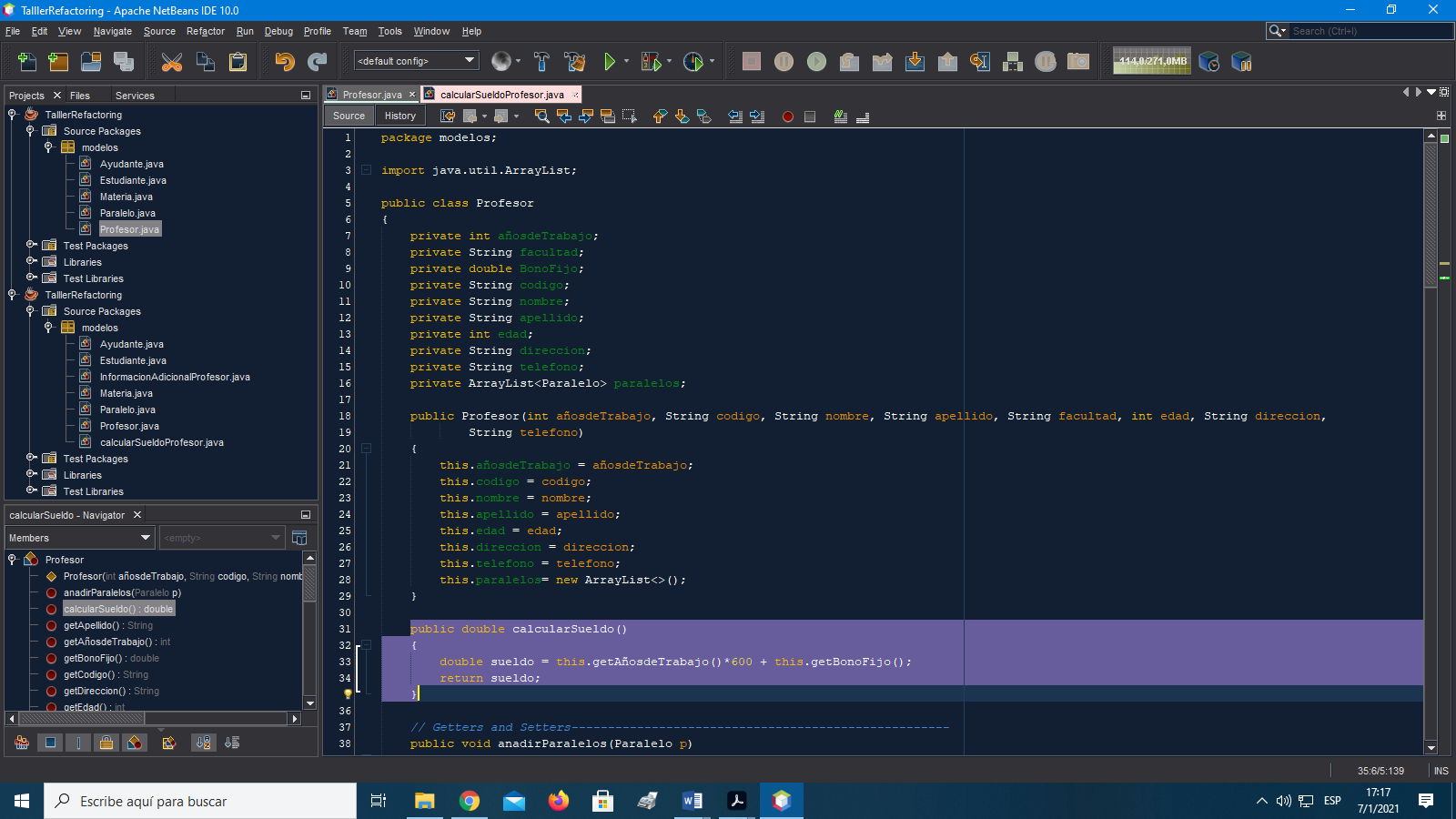
El código se vuelve más complejo de mantener, incluso se complica el control sobre los datos de cada clase debido a que se accede directamente a ellos.

**Técnicas usadas**

- Encapsulate Fields

- Move Method

**Código inicial**

**Código Refactorizado**

**Speculative Generality**

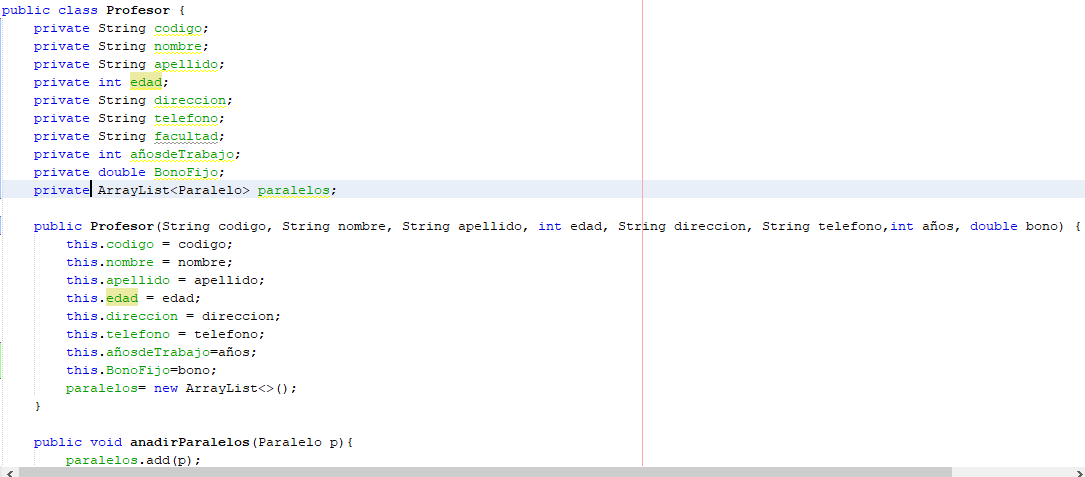
La clase profesor tiene muchos campos que no son utilizados. Por lo tanto, se tiene el code smell “Speculative Generality”. Si se refactoriza esta parte se tendrá un código más delgado y un soporte más sencillo.

**Consecuencias**

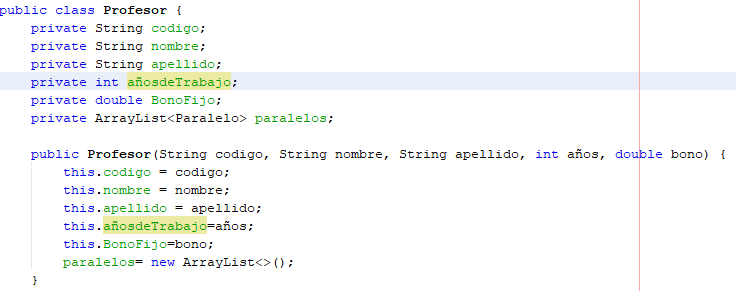
El código se vuelve largo y difícil de entender con tantos campos en una sola clase.

**Técnicas usadas**

Para refactorizar esto simplemente se eliminan estos campos que no se van a usar. En este caso solo se dejaran los campos de código, nombre y apellido para identificar al profesor y los años de trabajo y el bonofijo porque se usan en el método calcularSueldo().

**Código inicial**

**Código Refactorizado**



**Duplicate Code**

La clase estudiante tiene 2 métodos llamados CalcularNotaInicial y CalcularNotaFinal, los cuales tienen código duplicado.

**Consecuencias**

La duplicidad de código puede llevar a problemas a la hora de modificarlo, debido a que puede dejarse alguna duplicación sin modificar y por lo tanto introduciendo bugs que se pueden evitar.

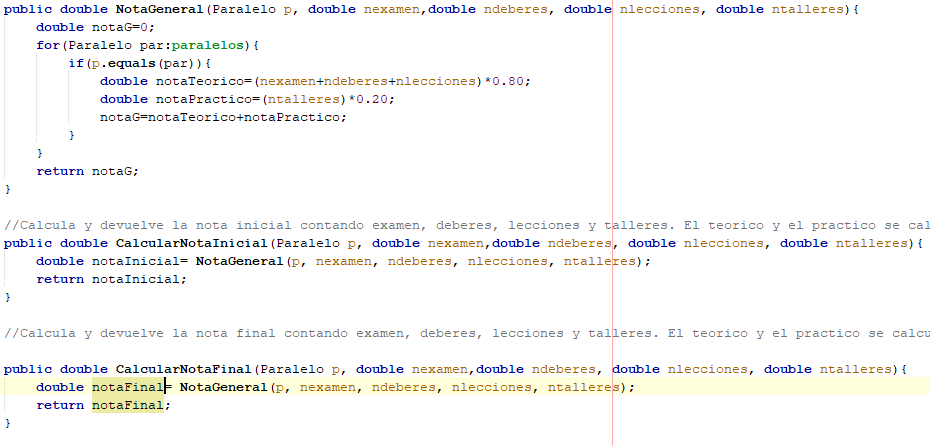
**Técnicas usadas**

Para arreglar esto, se usó la técnica de refactorización "Extract method", la cual extrae un fragmento de código en común que se puede agrupar en un solo método para facilitar la modificación de esa funcionalidad.

**Código inicial**



**Código Refactorizado**



**Feature Envy**

Los getters y setters de la clase “Ayudante” delegan sus funciones a los de la clase “Estudiante”, en otras palabras, la clase “Ayudante” accede más a atributos de otra clase que a los suyos propios, la clase que delega tiene funciones similares a la clase “Estudiante”, así que podemos reemplazar esto con una herencia.

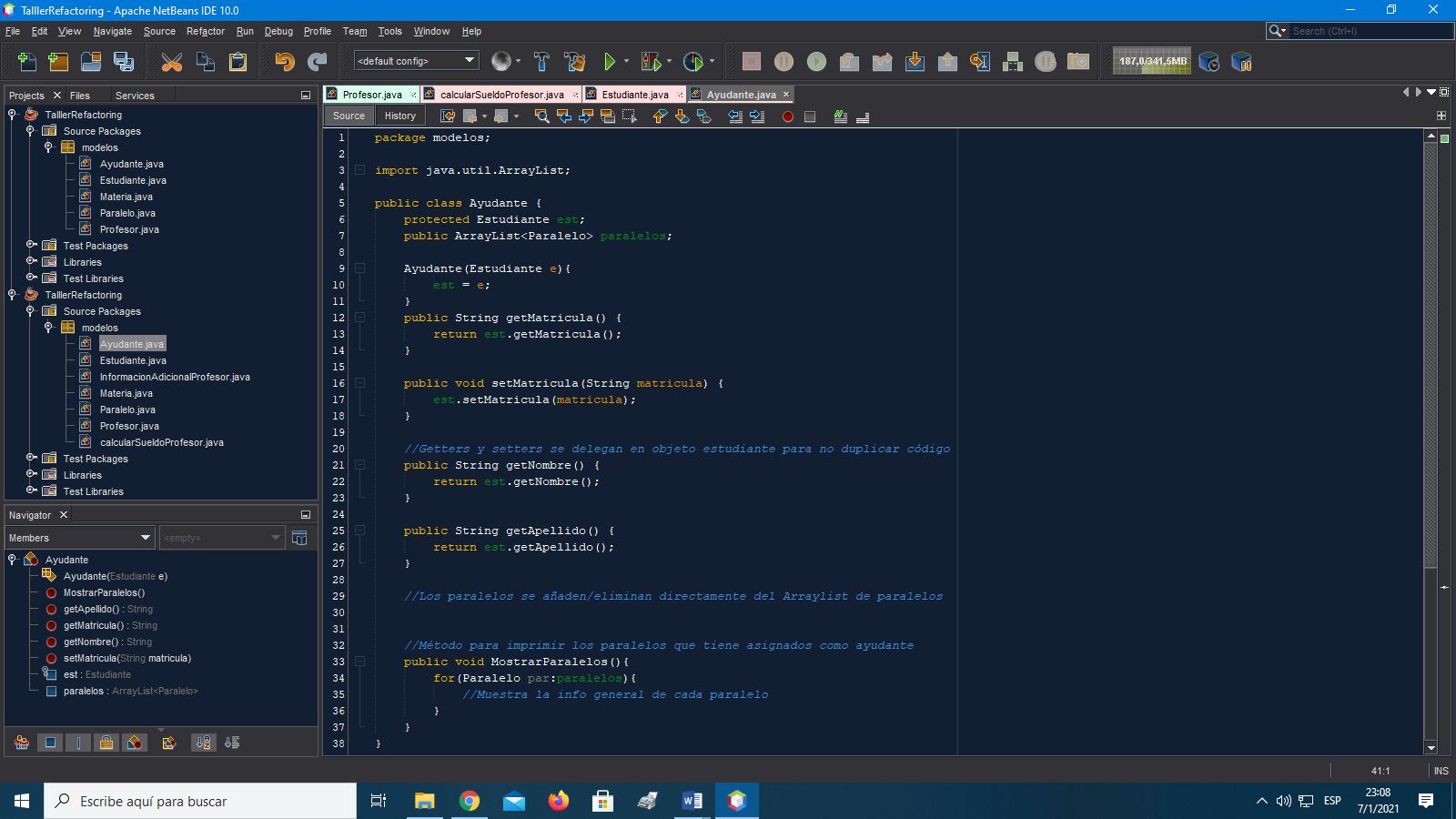
**Consecuencias**

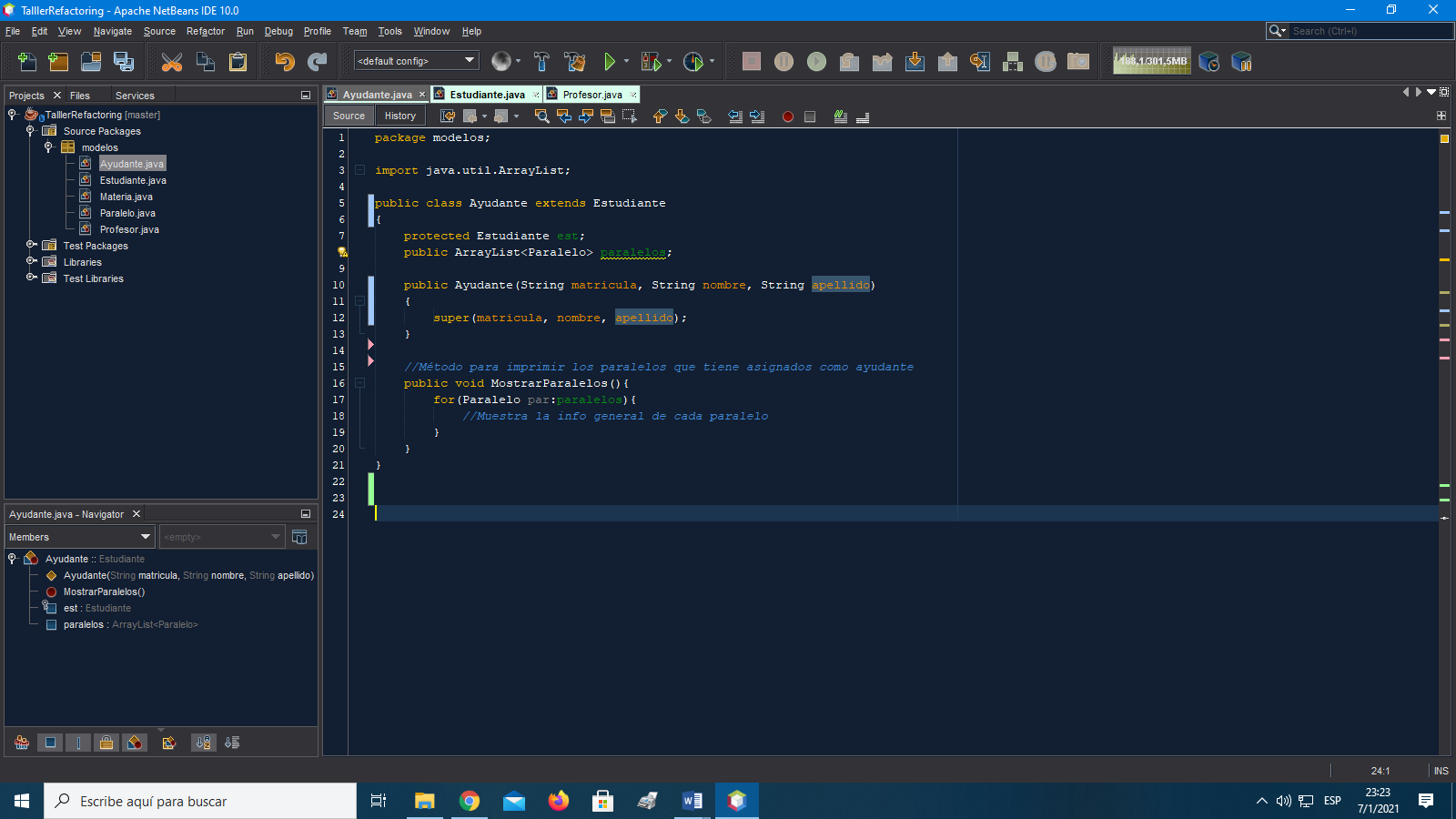
El código en este estado se vuelve más difícil de mantener y comprender por quien lo lea, además se puede ver que su función es las misma que la de otra clase.

**Técnicas usadas**

- Replace Delegation with Inheritance

**Código inicial**



**Código refactorizado**