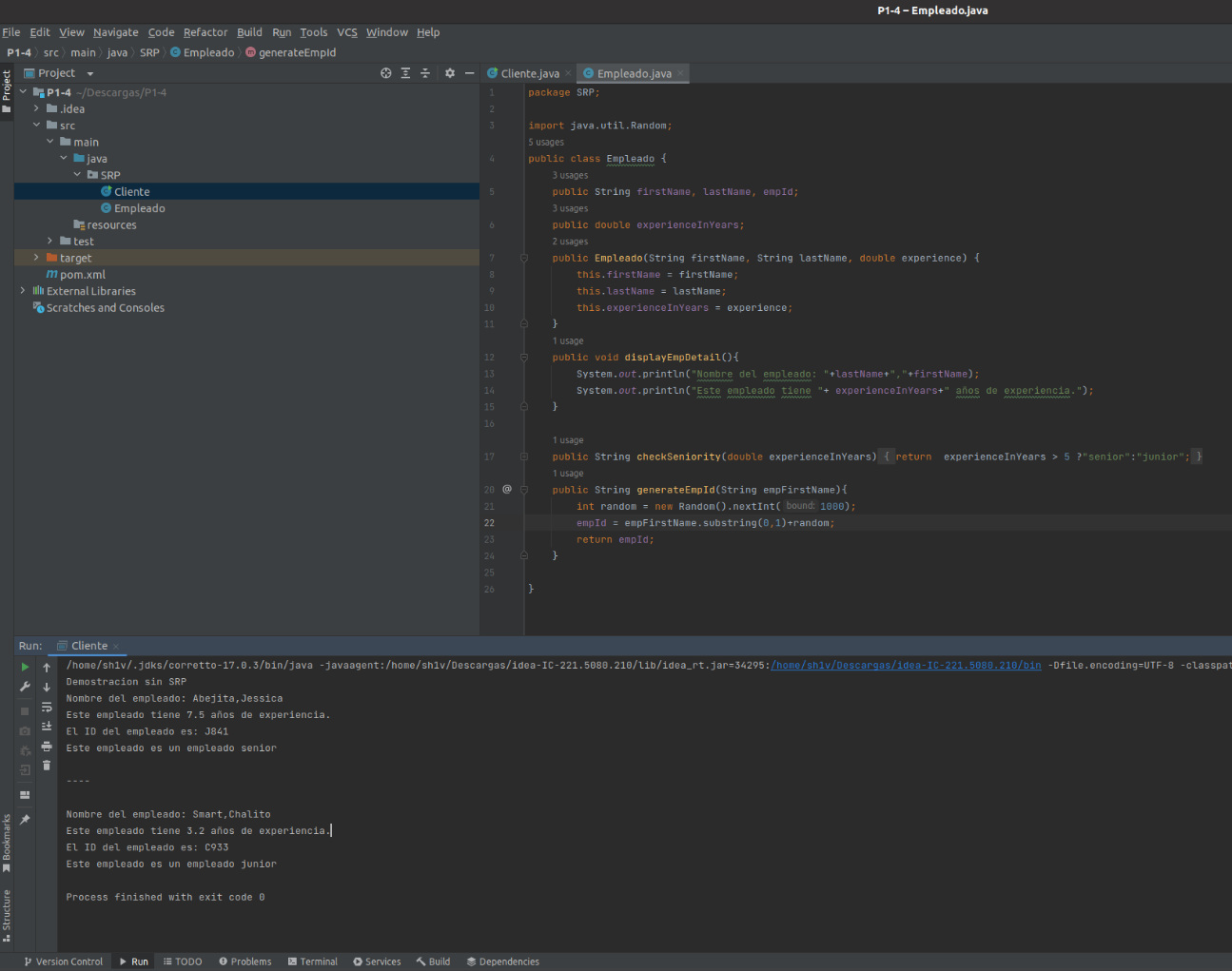
**Práctica Calificada N° 3 – Desarrollo de Software**

**Pregunta 1** Muestra la salida y explica los resultados en función de los métodos entregados



Observamos que tenemos dos clases: Cliente y Empleado. Al correr el código, el flujo del programa inicia en el método **main** de la clase Cliente, el cual primero imprime un mensaje en pantalla y, a continuación, instancia un objeto de la clase **Empleado** (el constructor recibe como parámetros el nombre, apellido y años de experiencia del empleado). Después, se llama al método **showEmpDetail** de la clase Cliente. Dicho método recibe como parámetro al objeto previamente instanciado de la clase Empleado y, además, será el encargado de llamar al resto de métodos de esta clase. Primero llama al método **displayEmpDetail** (método encargado de imprimir en pantalla el nombre y años de experiencia del empleado) y luego, al imprimir en pantalla el ID y rango del empleado, hace un llamado a los métodos **checkSeniority** y **generateEmpId**, los cuales reciben como parámetros un número y una cadena de texto respectivamente (no necesariamente los atributos previamente definidos en el constructor de la clase Empleado). Luego, se imprime en pantalla un espacio en blanco y se procede a instanciar un nuevo objeto de la clase Empleado. Una vez más, se llama al método **showEmpDetail** y el proceso anteriormente descrito se repite.

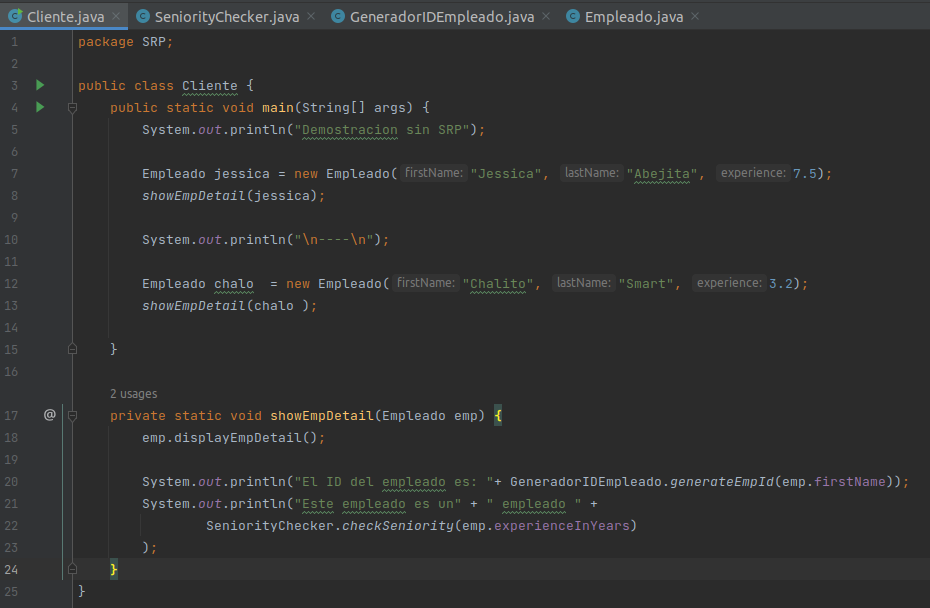
**Pregunta 2** ¿Cuál es el problema con este diseño y las razones posibles del problema?

El gran problema del diseño actual, es que no cumple con el **principio de responsabilidad única** de SOLID. Tenemos a la clase Empleado que se encuentra realizando más de una función; tenemos 3 métodos dentro de esta clase y cada uno de ellos tiene un propósito distinto. Al parecer la razón detrás de esto es que el desarrollador buscó contener en una sola clase todas las funcionalidades que requería su aplicación (con el fin de que todo el código se encuentre solo en dos clases); mas olvido que tomando dicha decisión podría pasar factura, por ejemplo, en temas de **escalabilidad**. Si en un futuro el código de dicha aplicación creciese, y se hiciese necesario el que los métodos de la actual clase Empleado puedan exportar sus resultados a diversos formatos (no solo imprimir mensajes en consola), se tendría que realizar varias modificaciones a la clase; esto a todas luces no es óptimo, por lo que lo recomendable sería refactorizar la clase ahora que el código no es grande.

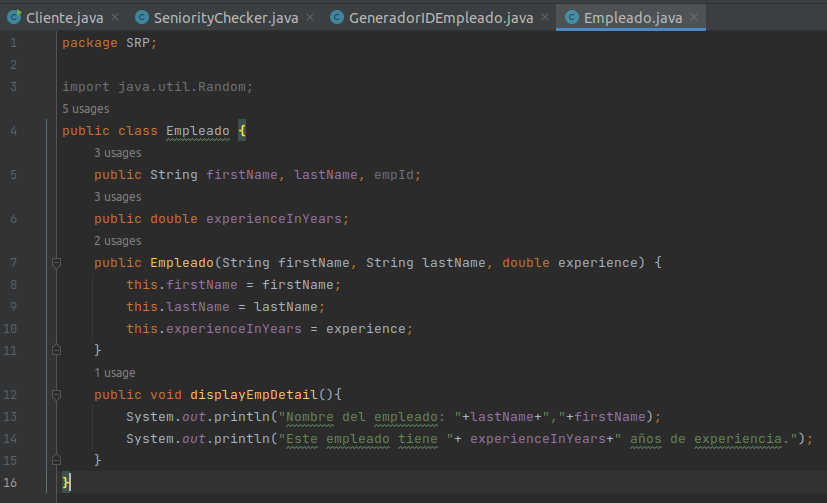
Dentro de la clase Cliente también nos encontramos con algo “innecesario”. Se está declarando un método (**showEmpDetail**) encargado de llamar a varios métodos de la clase Empleado. Esto podría obviarse y realizar el llamado a dichos métodos desde el propio método **main**.

**Pregunta 3** Modifica la clase Empleado

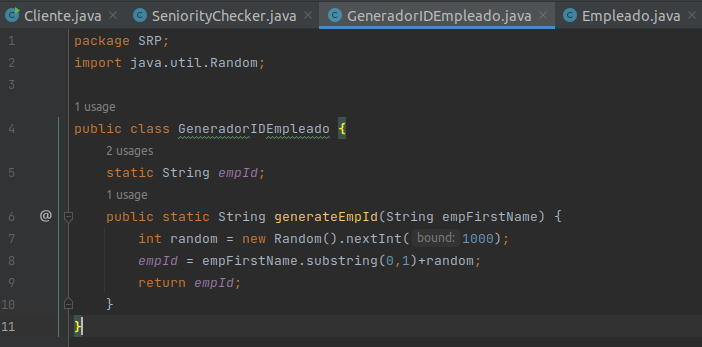
Clase **Cliente**:



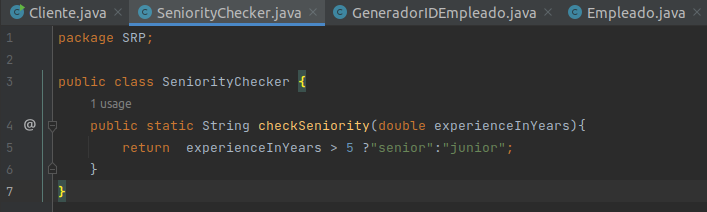
Clase **Empleado**:



Clase **GeneradorIDEmpleado**:



Clase **SeniorityChecker**:

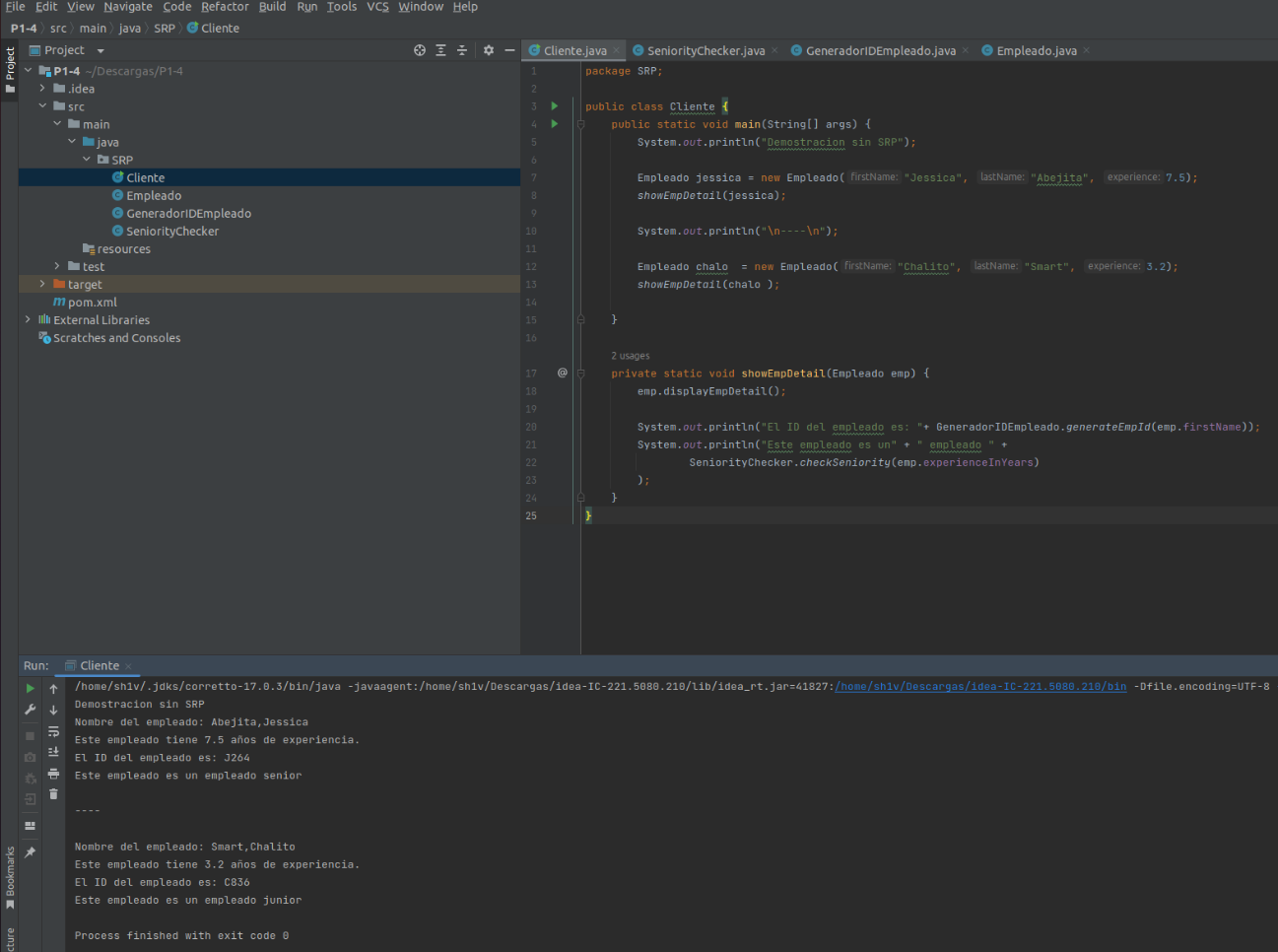


Dentro de la clase Empleado, se removieron los métodos generateEmpId y checkSeniority (con el fin de cumplir con el principio de responsabilidad única).

Se crearon las clases **GeneradorIDEmpleado** y **SeniorityChecker**, las cuales tienen un único método (generateEmpId y checkSeniority respectivamente). Dichos métodos fueron declarados como estáticos con el fin que puedan ser llamados sin necesidad de instanciar un objeto de la clase a la que pertenecen.

En el método showEmpDetail de la clase Cliente, se cambió el llamado a los antiguos métodos de la clase Empleado por los nuevos métodos de las clases recién creadas.

**Pregunta 4** Realiza una demostración completa que sigue a SRP. Explica tus resultados



Como se puede observar, los resultados no han varíado; pero lo que si ha cambiado es el diseño de este código, el cual ahora cumple con el principio de responsabilidad única. Cada una de las clases presentes solo tienen un único método, el cual a su vez solo cumplen una función y no más.

La clase Empleado tiene un método **displayEmpDetail**, el cúal solo se encarga de mostrarnos en pantalla el nombre del Empleado y la cantidad de años de experiencia que este tiene.

La clase GeneradorIDEmpleado solo cuenta con el método **generateEmpId**, el cual se encarga de crear un código aleatorio; código que luego es mostrado en pantalla cuando el presente método es llamado.

La clase SeniorityChecker tiene como único método a **checkSeniority**, encargada de, en base al parámetro que recibe, retornar una cadena indicando un rango. Esto último será también mostrado en pantalla cuando dicho método sea llamado.

Por último, la clase Cliente tiene los métodos **main** y **showEmpDetail** (este último no es realmente necesario y solo es usado para facilitar la comprensión del código). Dentro del método main es donde se hacen los llamados al resto de clases-métodos. De esta forma, se evidencia que cada clase tiene un solo propósito, por lo que queda demostrado el cumplimiento de SRP. Es importante recordar que, tal y como fue mencionado en una pregunta anterior, el diseñar nuestro código de esta forma nos puede ayudar a que nuestro software sea escalable. De ser necesario modificar alguna característica o realizar un pequeño cambio, solo será necesario afectar a un método perteneciente a una clase en específico; y evitar, por el contrario, tener que cambiar completamente una gran clase o incluso tener que refactorizar todo el software.

**Pregunta 5** ¿Por qué no es correcto colocar displayResult() y evaluateDistinction() en la misma clase, como la siguiente:

class Estudiante {

// ….

public void displayResult() {

// codigo

}

public void evaluateDistinction() {

// codigo

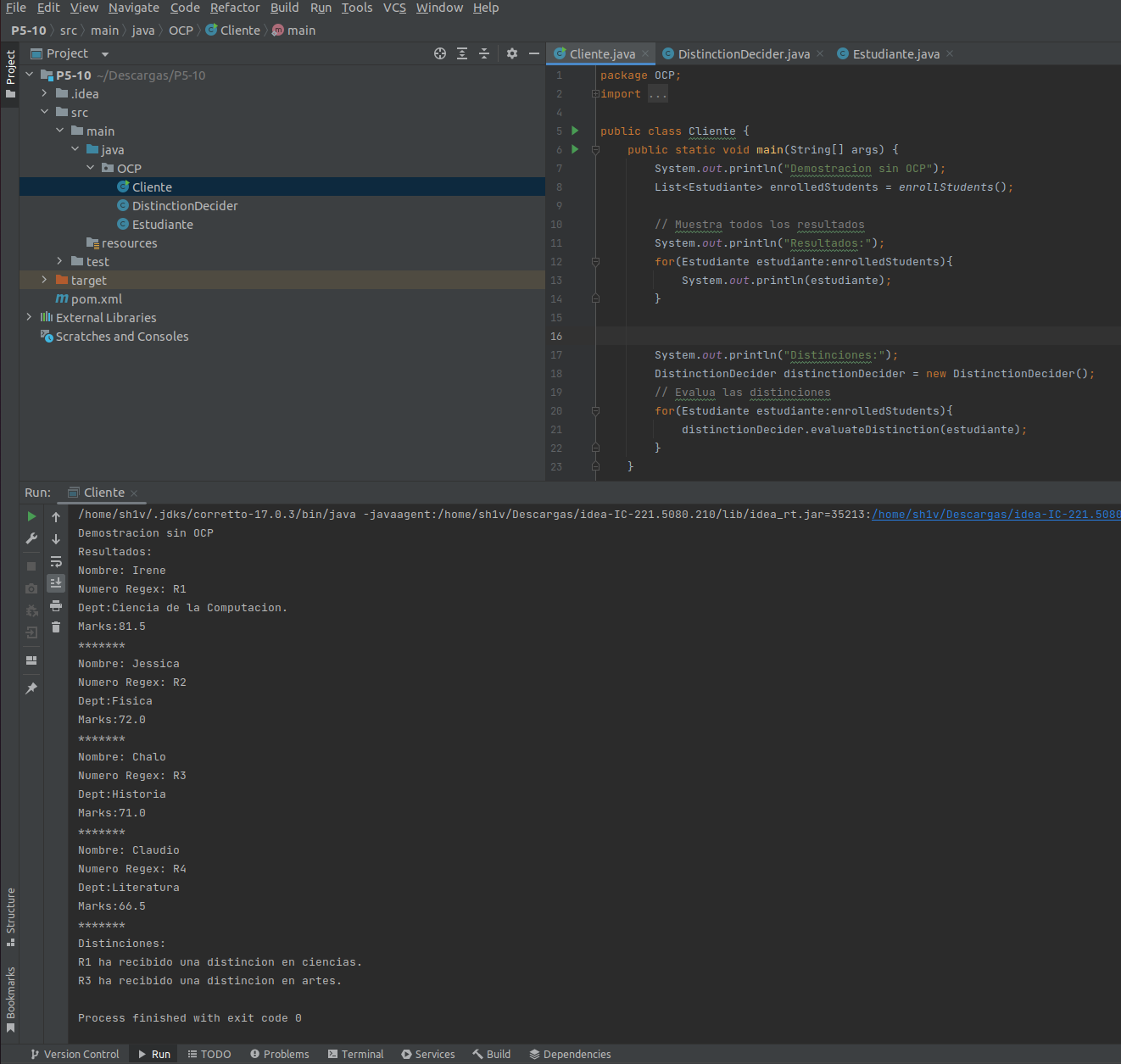
}

// …

}

En base a lo aprendido sobre el principio de responsabilidad única; al nosotros agregar dos métodos con funcionalidades distintas a la clase Empleado (clase que ya cuenta con el método toString), nos encontraríamos en un escenario donde estaríamos quebrantando dicho principio. Además, con el fin de aplicar a nuestro diseño el principio de Abierto/Cerrado, debemos considerar la importancia de previamente cumplir con SRP. Tener un código abierto a extensión y su vez cerrado a modificación, se basa en que nuestras clases tengan una única responsabilidad; así podremos añadir nuevas funcionalidades sin tener que estar afectando al código ya existente.

**Pregunta 6** Muestra la salida y explica los resultados en función de los métodos entregados



El flujo de nuestro programa inicia en el método **main()** de nuestra clase Cliente. Lo primero que hace es mostrar un mensaje en pantalla indicando que se realizará una demostración sin implementar el principio abierto/cerrado. Acto seguido, se declara una lista de objetos de tipo Estudiante, llamado **enrolledStudents**. Esta lista se obtiene al llamar el método estático **enrollStudents** (del tipo lista de estudiantes) de la clase Cliente. Dentro de este método se instancian objetos de la clase Estudiante, proporcionando los parámetros requeridos por el constructor de esta (nombre, número de registro, puntaje y departamento) y, posteriormente, se declara una lista de objetos “estudiante”, la cual será previamente llenada con los objetos que acabamos de instanciar, para finalmente ser retornada por el método actualmente descrito.

Siguiendo con el flujo principal del programa, se mostrará en pantalla los datos de cada uno de los estudiantes generados dentro del método **enrollStudents**. Esto es realizado mediante un **for**, el cual llamará al método **toString** (a partir del uso de un *system.out.printnl*) de cada uno de los objetos creados a partir de la clase Estudiante.

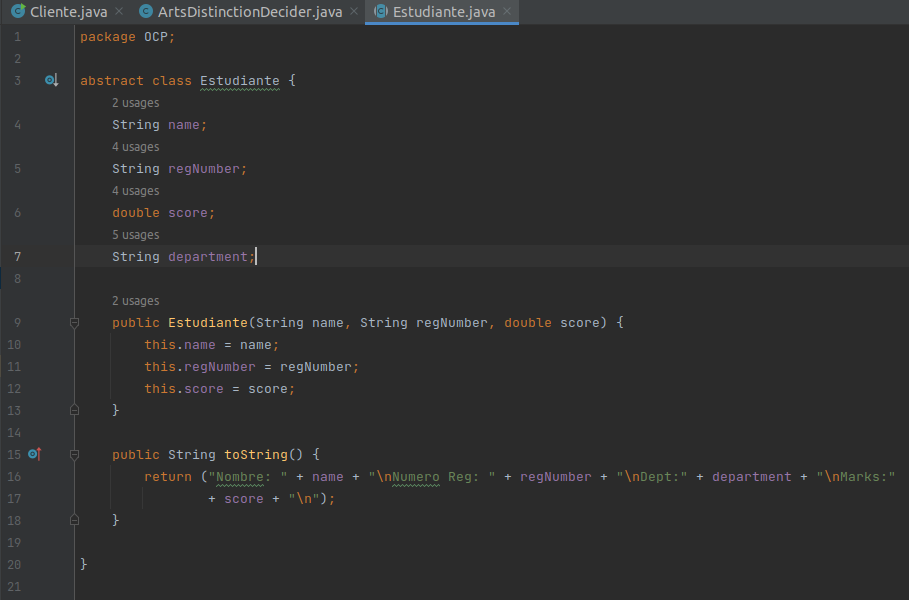
Finalmente, se instancia un objeto de la clase **DistinctionDecider**, llamado distintionDecider. La clase en mención cuenta con un método **evaluateDistinction**, el cual dependiendo del estudiante que le hayamos pasado como parámetro, nos imprimirá en pantalla si dicho estudiante ha recibido una mención en artes o ciencias. De vuelta en el flujo principal de nuestro programa, nuevamente mediante un **for** se iterará a través de los elementos de la lista **enrolledStudents** y, por cada uno de estos elementos, se llamará al método **evaluateDistinction** de distintionDecider, el cual recibirá como parámetro dicho elemento. De esta forma, podremos leer en pantalla el número de registro de los estudiantes que hayan recibido una mención, ya sea en artes o ciencias.

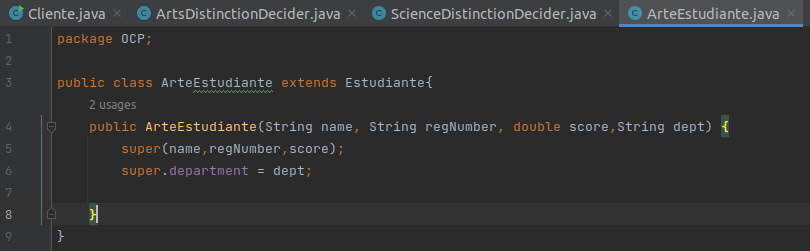
**Pregunta 7** ¿Cuál es el problema con este diseño y las razones posibles del problema?

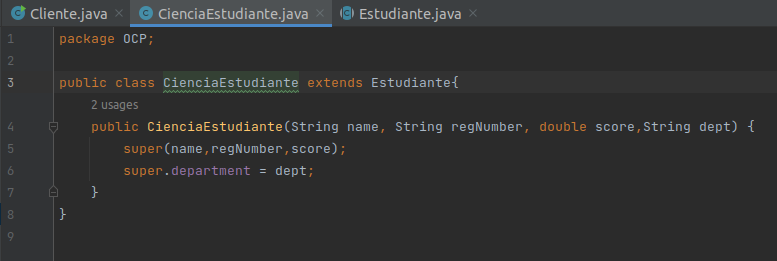
Basándonos en el diseño actual de la clase Estudiante, si nosotros quisiésemos, por ejemplo, añadir nuevos métodos específicos para los estudiantes del departamento de Física, tendríamos que modificar la clase; y si ahora necesitásemos añadir métodos para los estudiantes de Inglés, tendríamos que volver a realizar una modificación en la clase Estudiante. Claramente estamos violando el principio de abierto/cerrado, puesto que si bien nuestro código es abierto a extensión (añadir nuevas funcionalidades-métodos), debería permanecer cerrado a modificación; en otras palabras, no deberíamos tener la necesidad de cambiar el código ya existente de la clase Estudiante (la cual contiene a los estudiantes de 4 departamentos distintos).

De manera similar al caso anterior (respecto a SRP), probablemente el autor del código buscó simplificar el números de clases creadas agrupando a todos los estudiantes por su una característica general (son estudiantes), y no por algo quizás un poco más específico, pero necesario, como el hecho de que pueden pertenecer a cuatro departamentos distintos y que el flujo del programa depende mucho de esto.

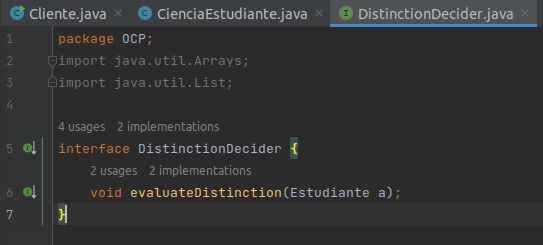
A continuación, se muestran imágenes de las modificaciones en e archivo **Estudiante.java** y la creación de **CienciaEstudiante.java** y **ArteEstudiante.java**.







**Pregunta 8** Debes abordar el método de evaluación para la distinción de una mejor manera. Por lo tanto, crea la interfaz DistinctionDecider que contiene un método llamado evaluateDistinction.



Se puede apreciar la interfaz DistintionDecider que ha sido creada. Se le añadió el método evaluateDistinction según las directrices de la pregunta. Dicho método será del tipo void y recibirá como parámetro un objeto de la clase Estudiante.

**Pregunta 9** Completa el código de ArtsDistinctionDecider y ScienceDistinctionDecider que implementan esta interfaz y sobreescriben el método de evaluateDistinction(...) para especificar los criterios de evaluación según sus necesidades. De esta forma, los criterios de distinción específicos de flujo se envuelven en una unidad independiente.





Se presentan capturas las clases **ArtsDistinctionDecider.java** y **ScienceDistinctionDecider.java**. A cada una de ellas se les implementa un método **evaluateDistinction**, previamente declarado en la interfaz DistinctionDecider. Debido que evaluateDistinction acepta un objeto “Estudiante” como parámetro, también puede aceptar a sus objetos hijos como CienciaEstudiante y ArteEstudiante (polimorfismo).