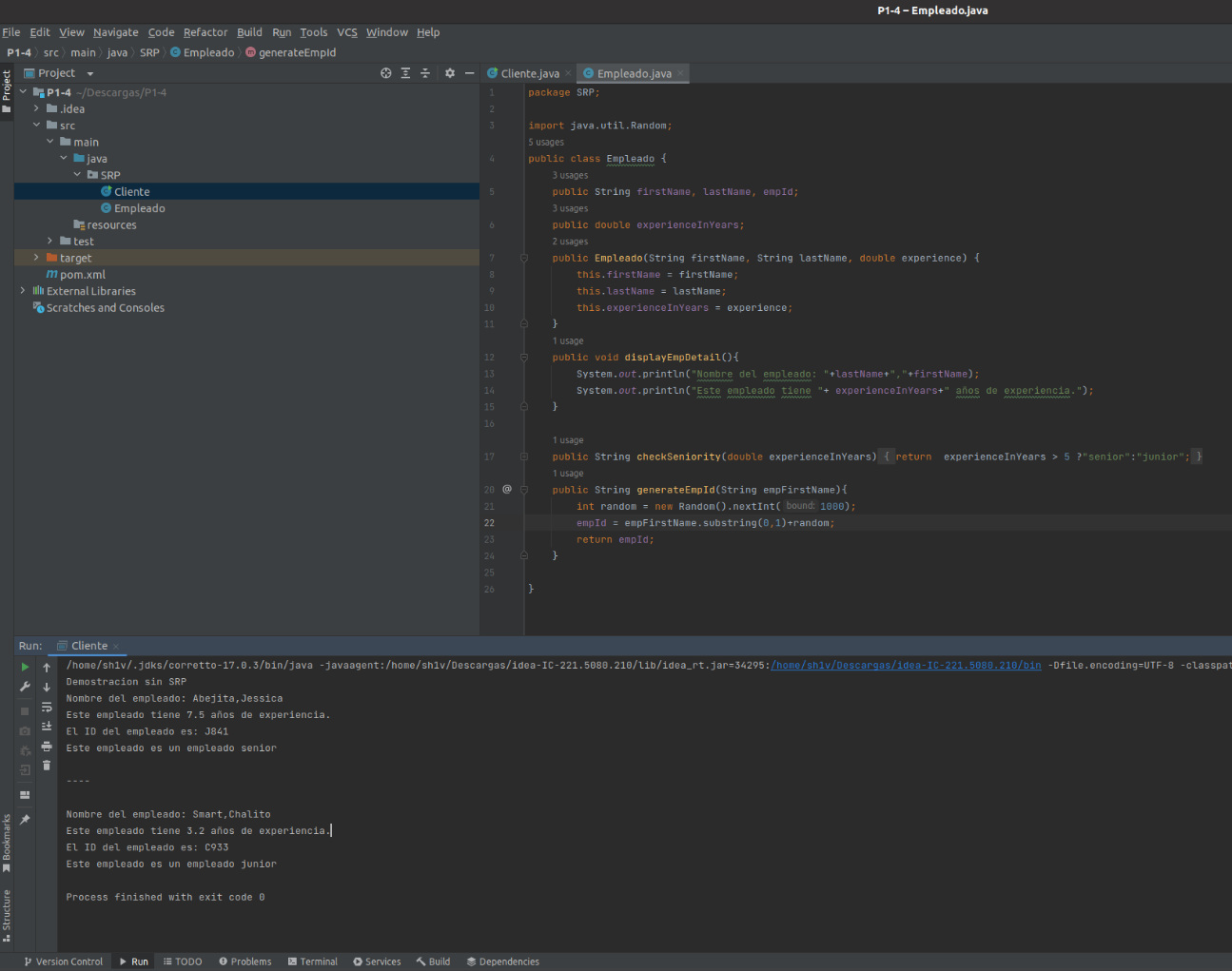
**Práctica Calificada N° 3 – Desarrollo de Software**

**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.

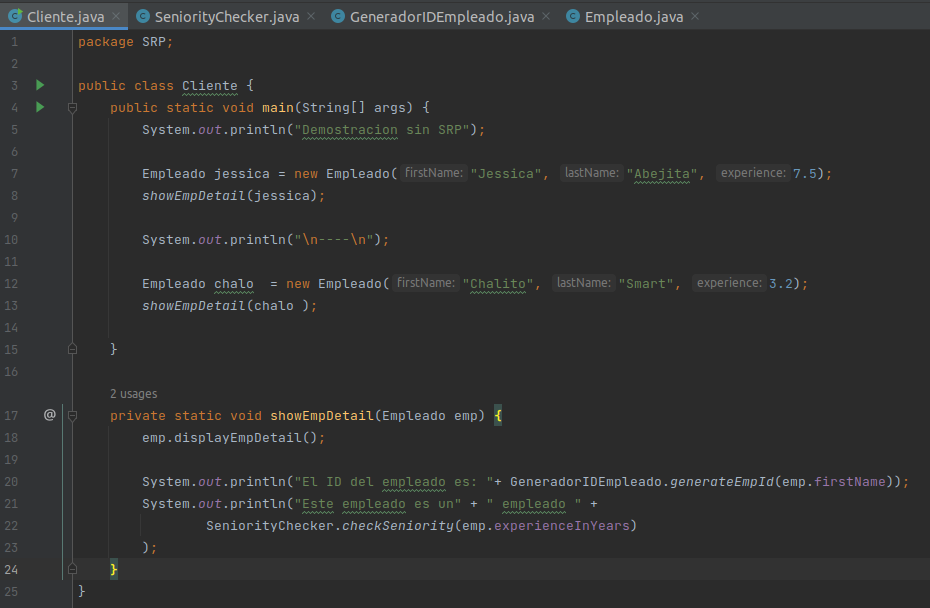
**Pregunta 2** ¿Cuál es el problema con este diseño y las razones posibles del problema?

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.

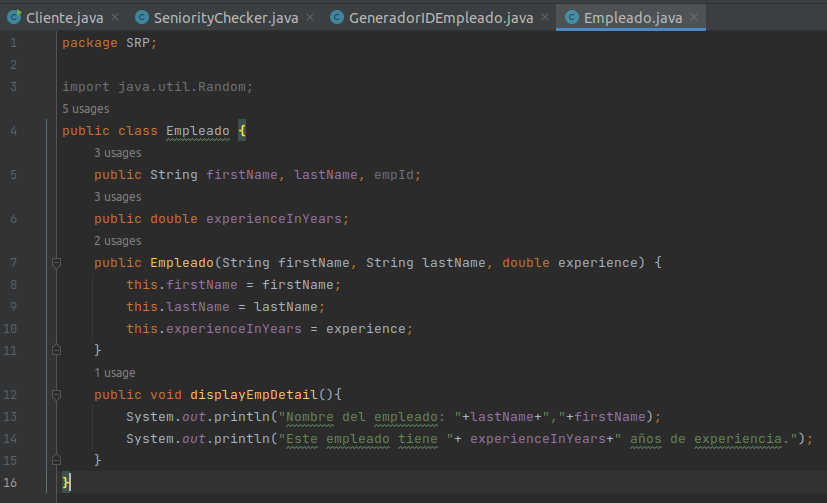
Dentro de la clase Cliente también nos encontramos con algo “innecesario”. Se está declarando un método (**showEmpDetail**) encargado de llamar a varios métodos de la clase Empleado. Esto podría obviarse y realizar el llamado a dichos métodos desde el propio método **main**.

**Pregunta 3** Modifica la clase Empleado

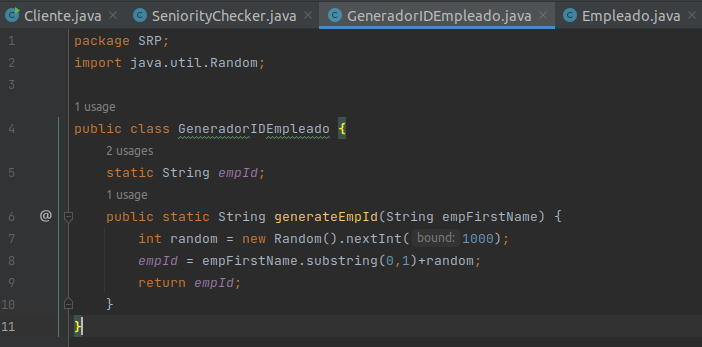
Clase **Cliente**:



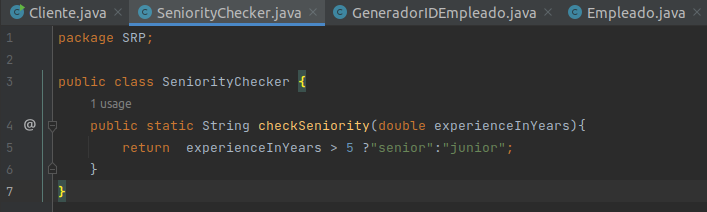
Clase **Empleado**:



Clase **GeneradorIDEmpleado**:



Clase **SeniorityChecker**:

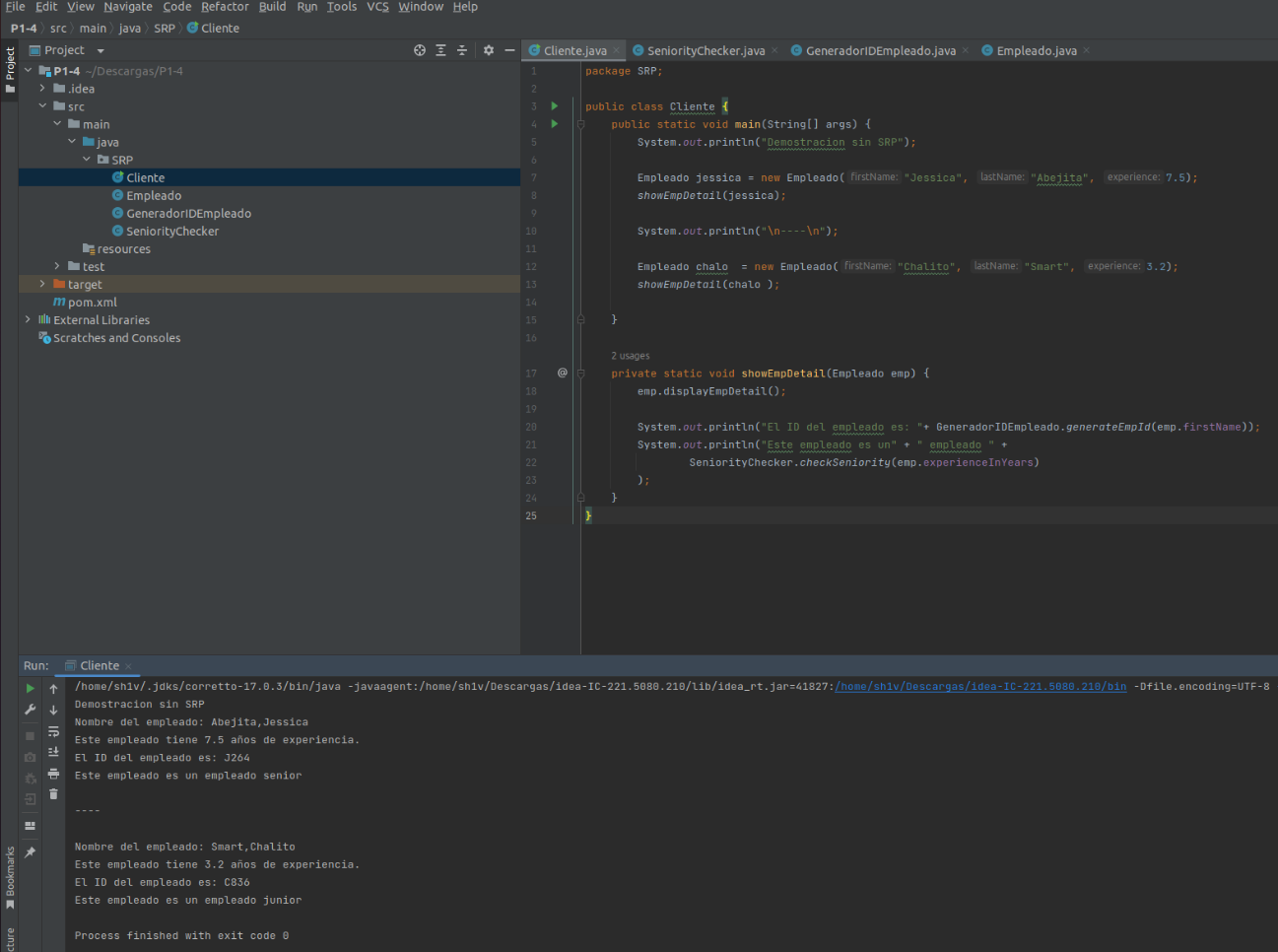


Dentro de la clase Empleado, se removieron los métodos generateEmpId y checkSeniority (con el fin de cumplir con el principio de responsabilidad única).

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

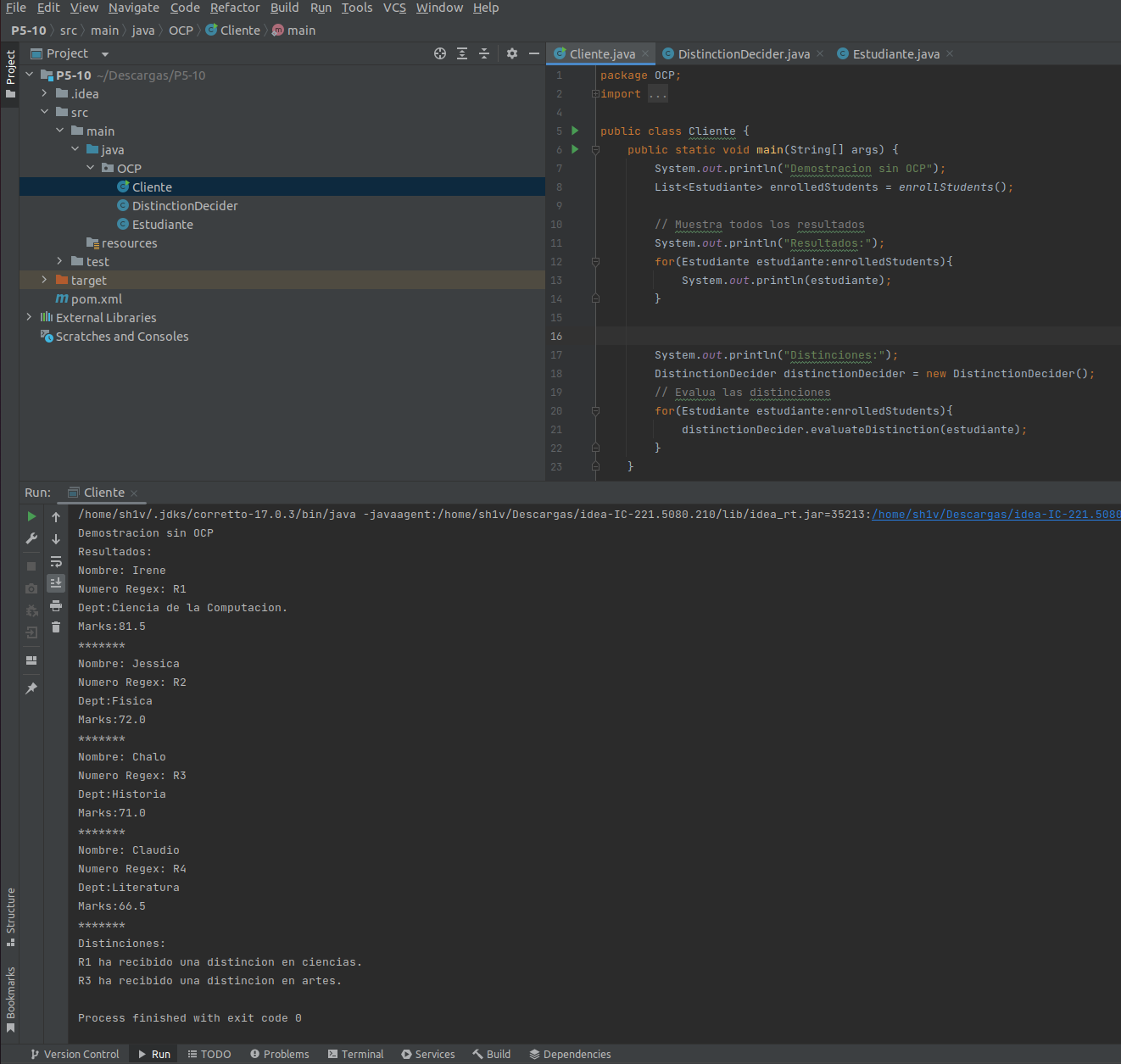
}

// …

}

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.

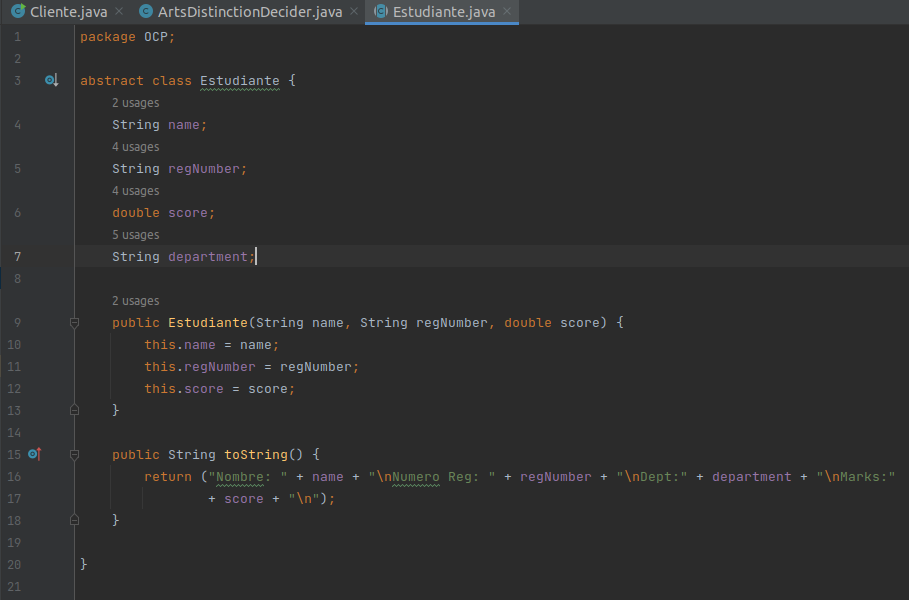
Finalmente, se instancia un objeto de la clase **DistinctionDecider**, llamado distintionDecider. La clase en mención cuenta con un método **evaluateDistinction**, el cual dependiendo del estudiante que le hayamos pasado como parámetro, nos imprimirá en pantalla si dicho estudiante ha recibido una mención 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.

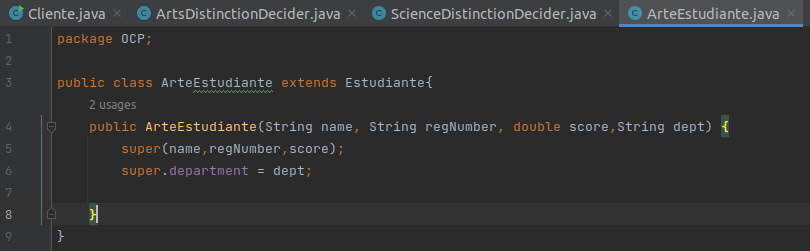
**Pregunta 7** ¿Cuál es el problema con este diseño y las razones posibles del problema?

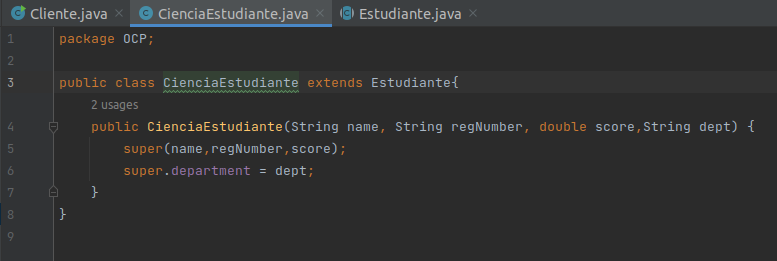
Basándonos en el diseño actual de la clase Estudiante, si nosotros quisiésemos, por ejemplo, añadir nuevos métodos específicos para los estudiantes del departamento de Física, tendríamos que modificar la clase; y si ahora necesitásemos añadir métodos para los estudiantes de Inglés, tendríamos que volver 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 número 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.

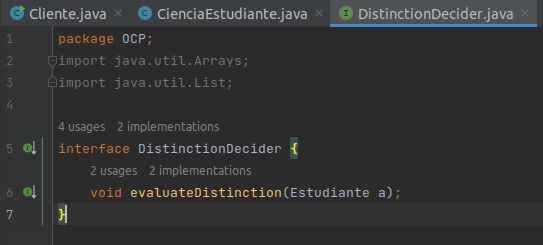
A continuación, se muestran imágenes de las modificaciones en el archivo **Estudiante.java** y la creación de **CienciaEstudiante.java** y **ArteEstudiante.java**.







**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).

**Pregunta 10** Realiza una demostración completa que sigue a OCP. Explica tus resultados

Estudiante.java

ArteEstudiante.java

CienciaEstudiante.java

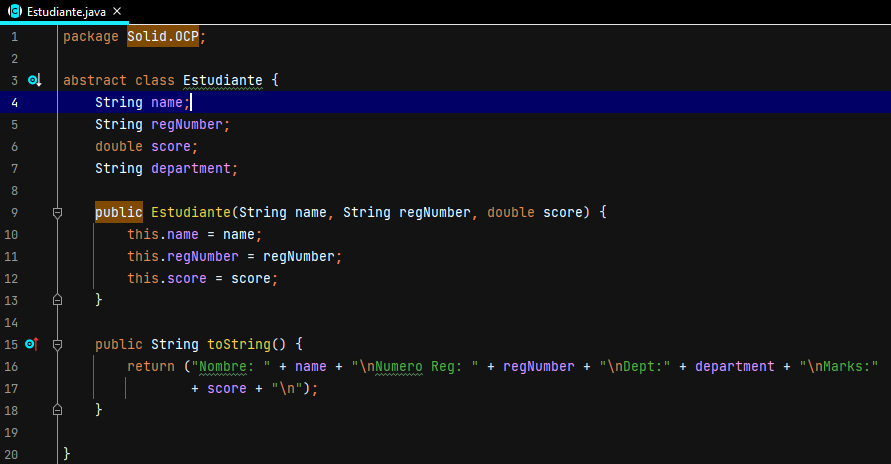
DistinctionDecider.java

ScienceDistinctionDecider.java

ArtsDistinctionDecider.java

Cliente.java

La clase Estudiante crea objetos de la clase estudiante



Las clases ArteEstudiante y CienciaEstudiante extienden de la clase abstracta Estudiante. Aquí se puede ver un posible uso de OCP, pues simplemente se crearía nuevas clases que extiendan de Estudiante

A screenshot of a computer

Description automatically generated with medium confidence

Text

Description automatically generated

Se crea la interfaz DistinctionDecider con la firma del método evaluateDistinction que va a recibir como parámetro un objeto de la clase Estudiante

Text

Description automatically generated

Las clases ScienceDistinctionDecider y ArtsDistinctionDecider implementan la interfaz DistinctionDecider y su método evaluateDistinction, usan este método para evaluar el departamento al cual corresponde el estudiante

Text

Description automatically generated

Text

Description automatically generated

En la clase Cliente que sigue el OCP se puede observar que se crean diferentes listas para los diferentes tipos de estudiante, esto hace que la lógica del programa sea mas escalable y mas entendible

Text

Description automatically generated

Text

Description automatically generated

Este sería el output siguiendo el principio de Open Close

Text

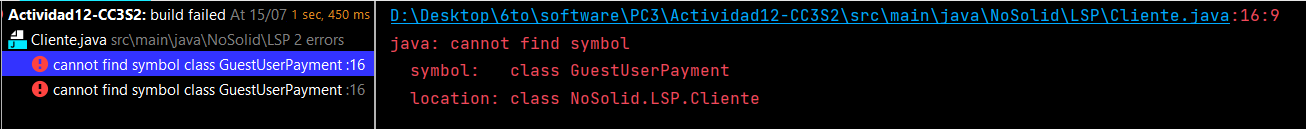
Description automatically generated

**Pregunta 11** ¿Cuáles son las principales ventajas ahora?

Las principales ventajas se muestran, por ejemplo, cuando se desee añadir otro tipo de estudiante (Deportes), en ese caso se debería modificar la clase DistinctionDecider y su método evaluateDistinction lo que conllevaría a no cumplir con el principio de OPEN/CLOSE, puesto que debería estar cerrado a modificación. Por otro lado, si creamos otra clase llamada SportStudent solo deberíamos extenderla de la clase abstracta Student y no modificar la clase ya escrita, lo mismo aplicaría para las clases que implementan la interfaz DistinctionDecider.

**Pregunta 12** Muestra la salida y explica los resultados en función de los métodos entregados

Cuando se corre tal y como fue entregado ocurre un error, pues la clase GuestUserPayment esta vacia



Sin embargo, si comentamos momentáneamente esa parte podemos observar:

Text

Description automatically generatedLa manera como está estructurado este ejemplo sin LSP es:

Existe una Interfaz Payment con las firmas de los métodos previousPaymentInfo y newPayment; esta interfaz es implementada en las clases RegisteredUserPayment y GuestUserPayment que hacen referencia a un tipo de usuario dentro del banco ficticio y también está la clase PaymentHelper que crea una lista de Payment, donde se puede añadir objetos de las clases que implementan esta interfaz (RegisteredUserPayment y GuestUserPayment) y serviría como un cajero o registro de operaciones y por ultimo tenemos la clase Cliente donde se puede ver la demostración sin LSP donde se instancia un objeto de tipo PaymentHelper, se crean dos objetos de la clase RegisteredUserPayment, se añaden estos a helper (PaymentHelper) y se ejecutan los métodos de showPreviousPayments y processNewPayments (ambos de estos métodos iteran sobre los antiguos o actuales pagos de los objetos que usan la interfaz Payment)

**Pregunta 13** Ahora supongamos que tienes un nuevo requisito que dice que necesitas admitir usuarios invitados en el futuro. Puedes procesar la solicitud de pago de un usuario invitado, pero no muestra su último detalle de pago. Entonces, crea la siguiente clase que implementa la interfaz de pago de la siguiente manera:

class GuestUserPayment implements Payment {

String name;

public GuestUserPayment() {

this.name = "guest";

}

@Override

public void previousPaymentInfo(){

throw new UnsupportedOperationException();

}

@Override

public void newPayment(){

System.out.println("Procesando de "+name+ "pago actual request.");

}

}

Una vez completada la clase GuestUserPayment y descomentado la clase Cliente se puede observar que obtenemos una UnsupportedOperationException y esto debido a que el método previousPaymentInfo de la clase GuestUserPayment arroja esa este tipo de Exception

A screenshot of a computer

Description automatically generated with medium confidence

**Pregunta 14** Dentro del método main(), utilizas una instancia de usuario invitado e intentas usar su clase auxiliar de la misma manera, ¿qué tipo de excepción te encuentras?¿Cuál es la solución?

Tal como explique en la pregunta 13 nos encontramos con una Exception de tipo UnsupportedOperationException; la solución sería emular lo que hace la otra clase que implementa la interfaz Payment (RegisteredUserPayment) para el método previousPaymentInfo y sería cambiar

Graphical user interface, text

Description automatically generated

Por un comentario que diga que no se puede mostrar los pagos anteriores pues es un usuario invitado

Graphical user interface

Description automatically generated

**Pregunta 15** Todo lo anterior Lo más importante es que viola el OCP cada vez que modifica una clase existente que usa esta cadena if-else. Entonces, busquemos una mejor solución.

Un caso que viola OCP es en la clase main

Text

Description automatically generated

Se podrían reemplazar RegisteredUserPayment y GuestUserPayment por la interfaz Payment

Text

Description automatically generated

Text

Description automatically generated

**Pregunta 16** En el próximo programa, eliminaremos el método newPayment() de la interfaz de payment. Coloca este método en otra interfaz llamada NewPayment. Como resultado, ahora tienes dos interfaces con las operaciones específicas. Dado que todos los tipos de usuarios pueden generar una nueva solicitud de pago, las clases concretas de RegisteredUserPayment y GuestUserPayment implementan la interfaz NewPayment. Pero muestra el último detalle de pago solo para los usuarios registrados. Entonces, la clase RegisteredUser implementa la interfaz payment. Dado que Payment contiene el método previousPaymentInfo(), tiene sentido elegir un nombre mejor, como PreviousPayment en lugar de Payment. Entonces, ahora verá las siguientes interfaces:

interface PreviousPayment {

void previousPaymentInfo();

}

interface NewPayment {

void newPayment();

}

Ajuste estos nuevos nombres en la clase auxiliar también. En sección del código debes tener los siguientes archivos

PreviousPayment.java

NewPayment.java

RegisteredUserPayment.java

GuestUserPayment.java

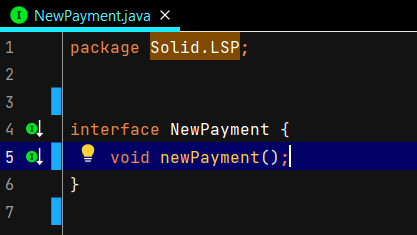
PaymentHelper.java

Cliente.java

Creamos las interfaces PreviousPayment y NewPayment

Text

Description automatically generated



Completamos las clases RegisteredUserPayment y GuestUserPayment implementando las interfaces necesarias

Text

Description automatically generated

A screenshot of a computer

Description automatically generated with medium confidence

A la hora de completar la clase PayementHelper hacemos hincapié en crear las listas y métodos del tipo deseado (Interfaces NewPayment y PreviousPayment) y así poder incluir y cumpli con el principio LSP

Text

Description automatically generated

Debido a que se verifico el uso de interfaces en nuestra clase Cliente se podrá cumplir el LSP

Text

Description automatically generated

**Pregunta 17** ¿Cuáles son los cambios clave?

Los cambios clave fueron separar los métodos previousPaymentInfo newPayment de la interfaz, esto porque la lógica indicaba que los objetos de la clase GuestUserPayment no debian usar el método previousPaymentInfo.

Al separar la interfaz Payment en dos interfaces nuevas (NewPayment y PreviousPayment) doy más independencia, escalabilidad y minimizo los posibles errores en el futuro

**Pregunta 18** Ten que aquí el enfoque clave estaba en el principio LSP, nada más. Podrías refactorizar fácilmente el código del cliente usando algún método estático. Por ejemplo: realiza una modificación donde utilizas un método estático para mostrar todas las solicitudes de pago y utilizar este método siempre que lo necesites.

La posible inclusión del método showEveryPayment que muestra los pagos ya registrados y los que serán registrados

Text

Description automatically generated

Text

Description automatically generated