**Escuela Superior Politécnica del Litoral**

**Tarea02-SOLID**

Diseño de Software

Grupo #1

Henry Quiñonez Chávez

Victor Garcia

Sergio Basurto

**PAO I 2022**

Incumplimiento de S.O.L.I.D.

Repositorio 1:

S: Principio de responsabilidad única

#1

Texto

Descripción generada automáticamenteInterfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

La clase AccountService y Console tienen el mismo método, además AccountService tiene un método de cambio de formato de numero, donde esta clase estaría teniendo 2 responsabilidades, de ser un servicio de cuenta y la otra de tener métodos de una consola, como imprimir una linea (printLine) y cambiar el formato de un numero (formatNumber)

Una posible solución, sería que solo la clase Console tenga los 2 metodos *formatNumber* y *printLine.*

Diagrama

Descripción generada automáticamente con confianza media

#2

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

La clase *Transaction*, tiene 2 responsabilidades, la primera de crear su constructor y la otra de fijas retornar valores de fecha, por lo que se considera que los 2 ultimos métodos de *Transaction,* deberia de ser de la clase *Clock.*

Diagrama

Descripción generada automáticamente

O: principio abierto/cerradoInterfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Para retornar el valor de pago de cualquier empleado tiene 2 clases, en conde intera para saber que empleado es, en caso de tener N mas empleados, deberia crear N métodos donde intere que tipo de empleado es, para ello se debe de crear una clase hija *Employee*  y que Manager y Engineer hereden de *Employee.*

Posible solución:

Diagrama

Descripción generada automáticamente

L: Principio de sustitución de Liskov

Texto

Descripción generada automáticamente con confianza media

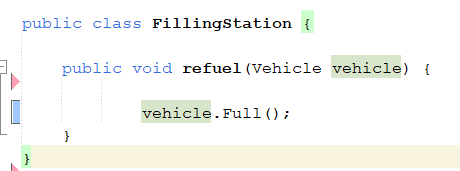
La clase padre *Vehicule* tiene 2 metodos abstractos, y estas sin heredadas por el sus hijas *ElectricCar* y *PetrolCar,* cuando a ElectricCar llama al método fillUpWithFuel(), lanza un exception. Así mismo el método batteryLevel de PetrolCar, lanza una exception.

Una posible solución es crear un método abstract llamado Full() cada clase hija va a modifica de la clase padre.

Posible solución:

Diagrama

Descripción generada automáticamente



I: Principio de segregación de interfaz

Interfaz de usuario gráfica, Aplicación, Word

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

No todos los animales pueden correr, volar, ladrar, lo que se puede hacer es crear mas interfaces que hereden de Animal, y cada clase de animal haga su correspondiente ajuste, y solo use las interfaces necesarias y así cada clase evitaría de tener métodos que no va a usar.

Diagrama

Descripción generada automáticamente

D: principio de inversión de dependencias

La BDD de envio, puede cambiar con el tiempo, para ello es necesario emplear una abstracion de envio de información.

Diagrama

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Repositorio 2: https://github.com/Valkyries12/escuelita-capgemini

Principio: S: Single Responsibility Principle

Violación: Dentro de la clase AlumnoController.java hay una función que no está acorde con todas las funcionalidades dentro de la clase que es la de listarAlumnos, por lo que procedemos a separla en una clase aparte llamada InfoAlumnos, y los demás métodos pasan a otra clase llamada adminAlumnos.

Texto

Descripción generada automáticamente

Solución:

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Diagrama

Descripción generada automáticamente

Principio: O: Open/Closed Principle

Violación: en la clase Alumno y en la clase Profesor tienen los mismos atributos, por lo que procedemos a hacer una clase padre llamada Usuario en donde, si en algún momento deciden añadir algún otro método lo puedan extender sin modificar el codigo.

Texto

Descripción generada automáticamente Texto

Descripción generada automáticamente

Solución:

Texto

Descripción generada automáticamente

Diagrama

Descripción generada automáticamente

Principio: L: Liskov Substitution Principle

Violación: al momento de crear una interfaz de usuario puede haber una confusión ya que los métodos de las dos clases hijas se usan específicamente para los estudios, y al momento de añadir alguna otra persona como conserje o secretaria no los compartiría y surge un error, por lo que pasamos a crear una interfaz persona con la que cada uno de ellos puede tener acceso a los atributos.

Captura de pantalla de computadora

Descripción generada automáticamente

Solución:

Texto

Descripción generada automáticamente

Diagrama

Descripción generada automáticamente

Principio: I: Interface Segregation Principle

Violación: Como el principio lo indica, buscamos segregar las clases, en este caso volvemos a tomar de nuevo la clase AlumnoController.java para este principio ya que a cada una de las clases podemos hacerlas una interfaz manejadas por un controller.

Texto

Descripción generada automáticamente

Solución:

Texto

Descripción generada automáticamente

Diagrama

Descripción generada automáticamente

Principio: D: Inversion Principle

Violación: en las clases de profesor y alumno incumple este otro principio ya que al estar estas dos clases separadas obliga a modificar a cada una de ellas ya que en este caso están dependiendo de los módulos de bajo nivel por lo que lo solucionamos con una interfaz persona

Captura de pantalla de computadora

Descripción generada automáticamente

Solución:

Texto

Descripción generada automáticamente

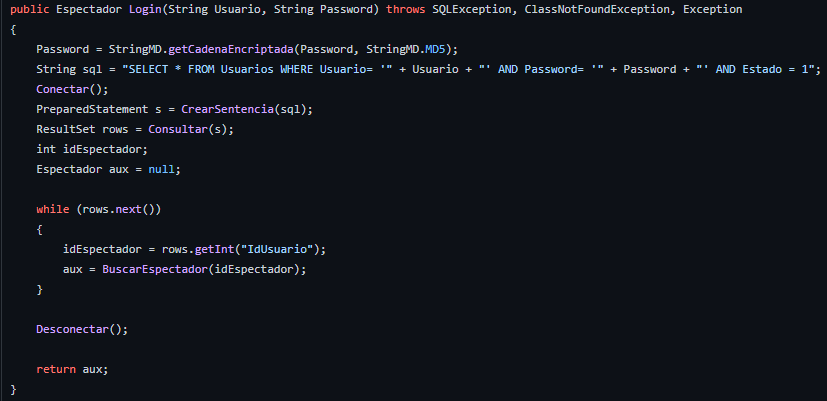
Texto

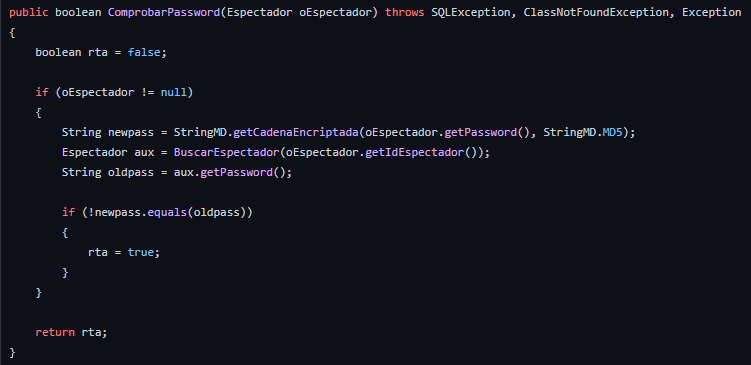
Descripción generada automáticamente

Repositorio 3: <https://github.com/wzalazar/Cine>

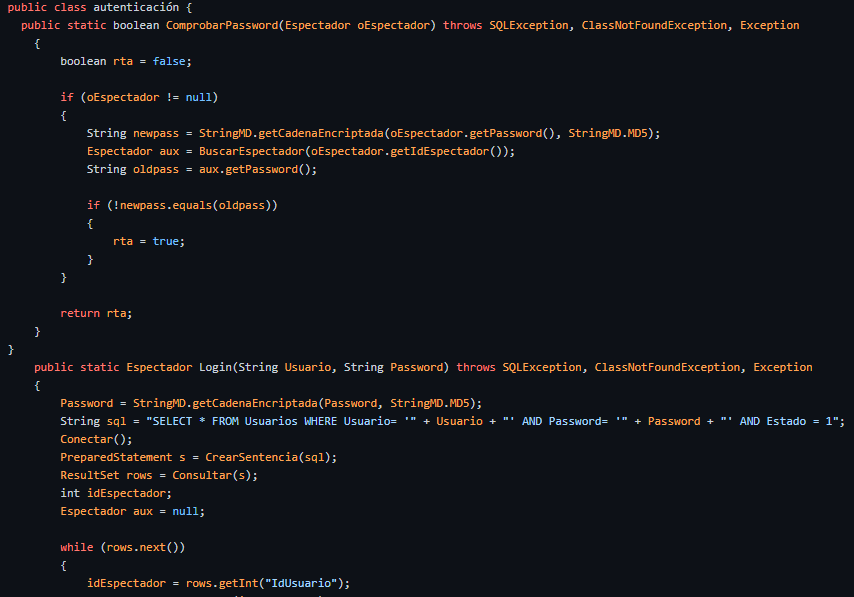
Principio: S: Single Responsibility Principle

Violación: Dentro de la clase de DatosEspectadores tenemos médotos que están realizando funcionalidades que están relacionadas a funciones que tienen responsabilidades de otra funcionalidad que debe ser implementada, cómo podria ser autenticación

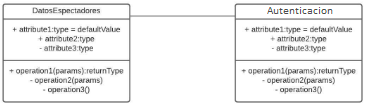




Solución: Para que la clase DatosEspectadores no tenga el trabajo de autenticar, se debería crear otra clase autenticacion y asignarle esos métodos, dado que la clase está orientada a añadir, eliminar, modificar, listar. No autenticar, así protegeremos los datos del usuario también.

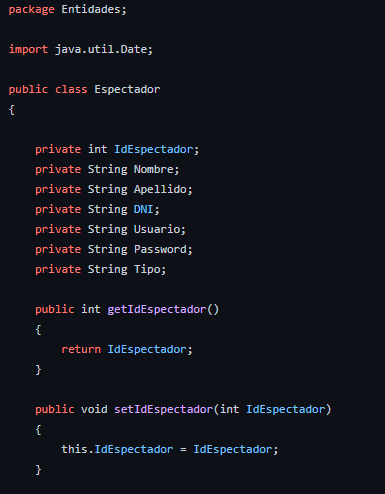


Uml:

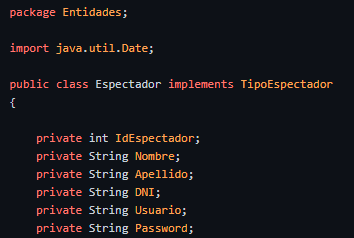


Principio: O: Open/Closed Principle

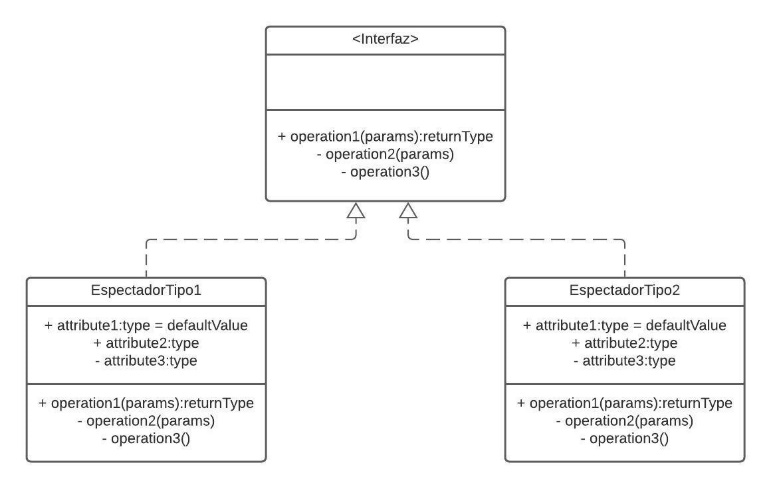
Violación: Cuando el programa quiera crecer y tenga nuevos tipos de espectadores con algún atributo especial, será más más complicado escalarlo, ya que tendriamos que hacer validaciones o agregando características más complejas.



Solución: Debemos realizar una interfaz nueva, donde cuando exista un nuevo tipo del atributo este adopte de esta los métodos especiales que pueda tener un espectador de acorde a su tipo con lo métodos se tengan que implementar a futuro.



Uml: Los diferentes tipos de espectadores vendrán de una interfaz.



Principio: L: Liskov Substitution Principle

Violación: El repositorio al parecer no tiene problema de Liskov, pero a futuro con los tipos de espectadores se puede tener, dado que tendremos que tener una forma de identificarlos.

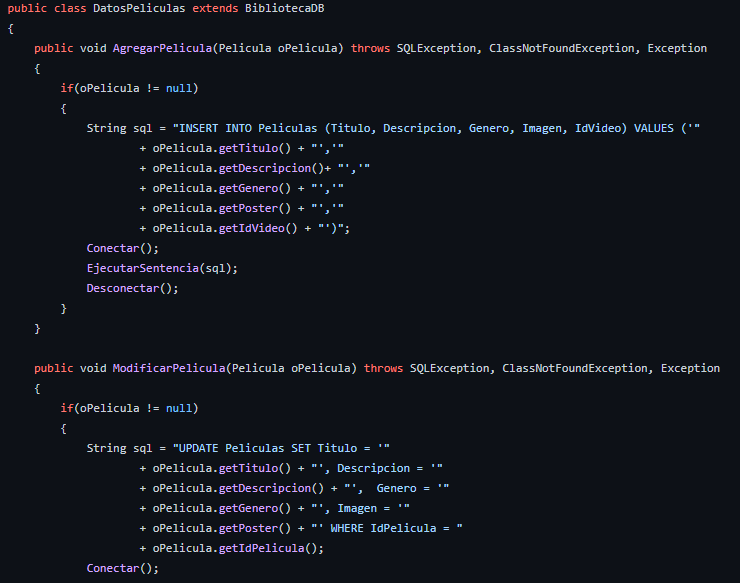
Solución: si se crean clases subclases a futuro podríamos tener una interfaz para evitar futuras dependencias y errores al modificar clases hijas.  
Uml: Una posible solución a este problema futuro, tener una interfaz para los nuevos tipos de espectadores, por ejemplo, aquí hemos creado uno exclusivo que extienda siempre de su clase principal.

Diagram

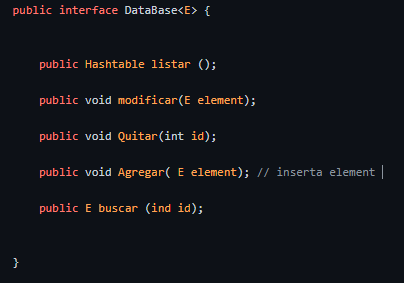
Description automatically generated

Principio: I: Interface Segregation Principle

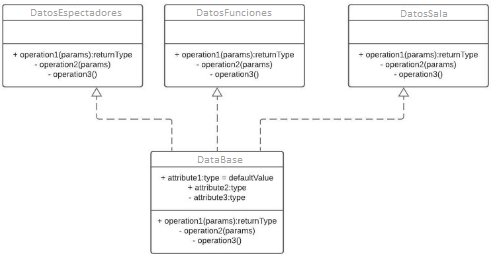
Violación: Algunas clases de las entidades tienen métodos repetidos cuya función es invocar datos desde una base de datos. DatosEspectadores, DatosFunciones, DatosPelículas,DatosSalas.



Solución: Crear una interface llamada Database donde estén los métodos agregar, buscar, modificar, eliminar, enlistar.

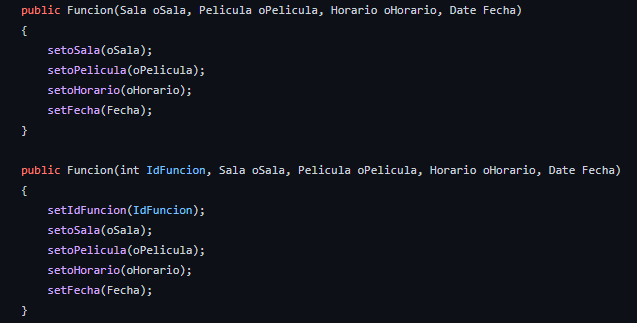


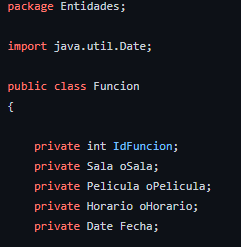
Uml:



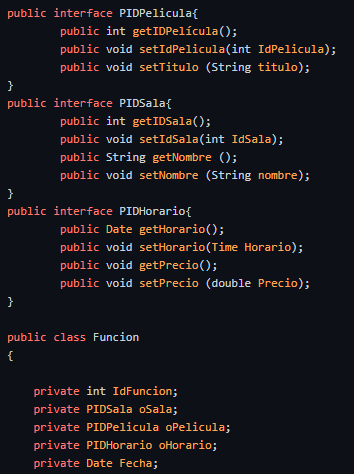
Principio: D: Dependency Inversion Principle

Violación: En el principio de dependencias, tenemos como para crear una función tenemos que utilizar métodos de otras tres clases Horario, Sala, Película lo cual viola el principio de no depender de módulos de un nivel inferior.





Solución: Ahora función dependerá de abstracciones y no de las clases de otros nivel, que pueden alterar su comportamiento.



Uml:

Diagram

Description automatically generated

