Práctica 2: Arquitectura Software

María González Gómez

Paradigma y técnicas de programación

Índice

1. Introducción

En esta práctica hemos seguido con el código base de la práctica 1 y hemos implementado los cambios pedidos, asegurándonos en todo momento de que se mantenían presentes los principios SOLID.

2. Principios SOLID

2.1. Descripción

- **S:** Principio de Responsabilidad Única
  - Ejemplo sin: Una clase 'UserCreator' crea usuarios y valida correos electrónicos.
  - Ejemplo con: Creamos dos clases, 'UserCreator' que crea usuarios y por otro lado la clase 'EmailValidator', que como su nombre índica, es la indicada de la validación de los email de cada usuario.

- **O:** Principio Abierto/Cerrado
  - Ejemplo sin: La clase ‘AreaCalculator‘ depende de ‘Rectangle‘ y no puede manejar otras figuras.
  - Ejemplo con: Creamos una interfaz llamada Shape, dejando a cada figura el cálculo propio de su área. Así, se generaliza la clase ‘AreaCalculator‘ de tal forma que está abierta a cualquier figura, sin necesidad de tener que modificar su código.
- **L:** Principio de Sustitución de Liskov
  - Ejemplo sin: Una clase ‘Animal‘ tiene un método ‘makeSound‘. Ahora creamos dos clases ‘Dog‘ y ‘Fish‘ que heredan de esta y el método, pero ‘Fish‘ no hace uso de ello.
  - Ejemplo con: Eliminamos el método ‘makeSound‘ de la clase ‘Animal‘ de tal forma que esta generalizada para cualquier animal. A continuación, creamos otra clase llamada ‘AnimalWithSound‘ que hereda de la clase ‘Animal‘ y además tienen el método ‘makeSound‘. Así la clase ‘Fish‘ herederará directamente de la clase general ‘Animal‘ y ‘Dog‘ de ‘AnimalWithSound‘.
- **I:** Principio de Segregación de Interfaces
  - Ejemplo sin: Una interfaz ‘IWorker‘ tiene los métodos ‘work‘ y ‘eat‘. Si creamos dos clases: ‘Humano‘ y ‘Robot‘ que hereden de esta, observamos que la clase ‘Robot‘ depende del método ‘eat‘, pero no lo implementa.
  - Ejemplo con: Se divide la interfaz ‘IWorker‘ en ‘IWorker‘ e ‘IEater‘, así el robot y el humano siguen implementando la interfaz ‘IWorker‘ que ahora solo tiene el método ‘work‘, y el humano es el único que usa la interfaz ‘IEater‘, de tal forma que el ‘Robot‘ no depende de algo que no usa.
- **D:** Principio de Inversión de Dependencias
  - Ejemplo sin: Tenemos un ‘Switch‘ que depende directamente de una ‘LightBulb‘, por lo que si quisieramos añadir funcionalidades a ‘Switch‘, dependeríamos de los requisitos de ‘LightBulb‘.
  - Ejemplo con: Se usa una interfaz para separar las dependencias, permitiendo que ‘Switch‘ funcione con cualquier dispositivo.

2.2. Aplicación en el código

- **S:** Principio de Responsabilidad Única

  Cada clase realiza unicamente sus tareas correspondientes, por ejemplo en vez de hacer que la PoliceStation además de registrar los coches de policia los cree con sus atributos y métodos, tenemos una clase PoliceCar que se encarga de la creación de estos y la definición de sus métodos.

  Por otro lado la ciudad unicamente se encarga de registrar y retirar licencias. El radar se encarga de las mediciones de velocidad y su almacenamiento, y los vehículos realizan su acción indicada.

- **O:** Principio Abierto/Cerrado

  Esto se cumple, como por ejemplo con la interfaz usada. Esta, tiene un método WriteMessage que permite a las clases que lo usen y devolvuevan un mensaje personalizado a sus tareas, y no tenemos necesidad de tener que cambiar el código en la interfaz.

- **L:** Principio de Sustitución de Liskov

  En un principio nuestra clase Vehicle es la general de la que heredan todos los vehículos un tipo, velocidada y matrícula. Sin embargo, al crear la clase Scooter, nos surge un problema, ya que este no tiene matrícula. Para solucionarlo, generalizamos la clase Vehicle, eliminando el atributo de matrícula y sus métodos, y creamos una clase VehicleWithPlate que hereda de Vehicle y además tiene el atríbuto "platez sus métodos correspondientes.

- **I:** Principio de Segregación de Interfaces

  Se cumple ya que nuestro código contiene una única interfaz llamada IMessageWriter con un único método: WriteMessage, el cual usan todas las clases que heredan de ella.

- **D:** Principio de Inversión de Dependencias

  Se cumple ya que no ninguna clase de alto nivel como Vehicle depende de clases de bajo nivel como Scooter.

3. Caso Hipotético

Ahora queremos que el policía pueda tener diferentes aparatos de medida, que pueden ser un medidor de velocidad (Radar) o un medidor de alcohol (Alcoholímetro).

Al coche de policía solo se le puede asignar un único medidor, y en el coche de policía únicamente va a haber un método para activar el aparato de medida.

¿Qué principio SOLID incumpliríamos y cómo lo solucionarías?

En primer lugar, crearíamos un atributo llamado breathalyzer con la misma estructura que el de SpeedRadar: *private Breathalyzer? breathalyzer*, de tal forma que el argumento puede tomar el valor de null en caso de no ser asignado a un coche. También, añadimos un método UseBreathalyzer, que llamará a algún método de dicha clase en caso de tener el breathalyzer como atributo.

Ahora pasaríamos a la creación de la clase Breathalyzer que tendría 2 atributos privados: *float legalAlcoholValue* y una *List<float> AlcoholHistory* que iria almacenando los valores del alcoholímetro. En cuanto a los métodos tendría un *public float GetLegalAlcoholValue*, que devolvería el valor legal del alcoholímetro, un *public string GetLastReading* que devolvería el historial de valores, un *public void TriggerDevice* que obtendría la placa del vehículo, su valor de alcohol y lo almacenaría en el historial.

Como se puede apreciar la clase Breathalyzer comparte gran parte de su estructura con la clase Radar, aquí estamos incumpliendo el principio de Sustitución de Liskov.

Solución: Crear una clase llamada MeasuringDevice de la que hereden el radar y el alcoholímetro. Esta tendrá como atributos: *float legalValue*, *List<float> History*, y como métodos: *public string GetLastReading*, *public string MessageWriter*, *public float GetLegalValue* y *public void TriggerDevice*.

De esta forma evitamos la duplicidad de código y mantenemos los principios SOLID presentes en toda la estructura del código. Además el código del PoliceCar no cambia más allá de las 2 actualizaciones necesarias.

4. Diagrama UML

Los diagramas UML son comunmente usados en el desarrollo de software y sistemas ya que permiten representar de manera gráfica la estructura y comportamiento del sistema, facilitando la comprensión de este. Además, son un gran medio de comunicación entre miembros del equipo, ya que realizar cambios en el es algo sencillo y rápido, al contrario que en un proyecto de código ya elaborado.

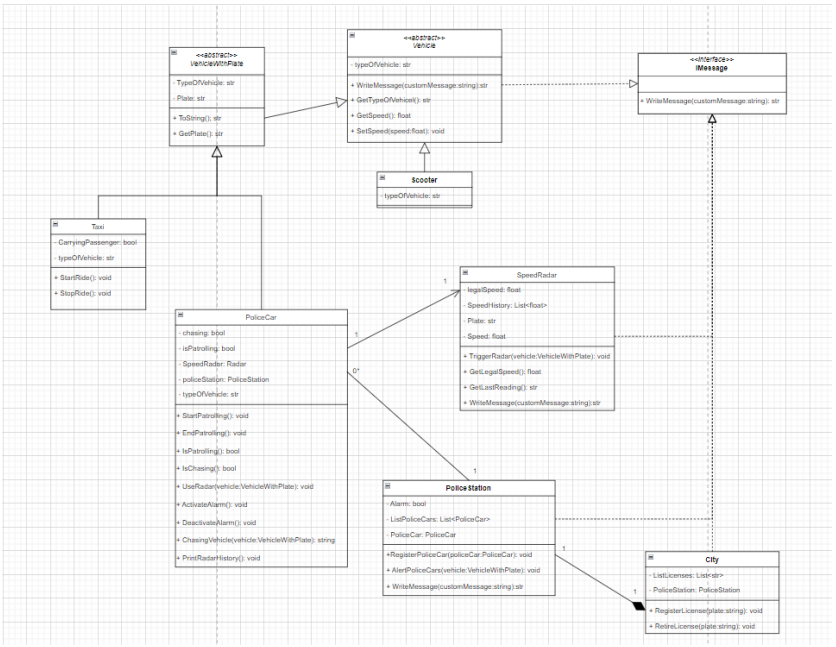

Figura 1: Diagrama UML del sistema.