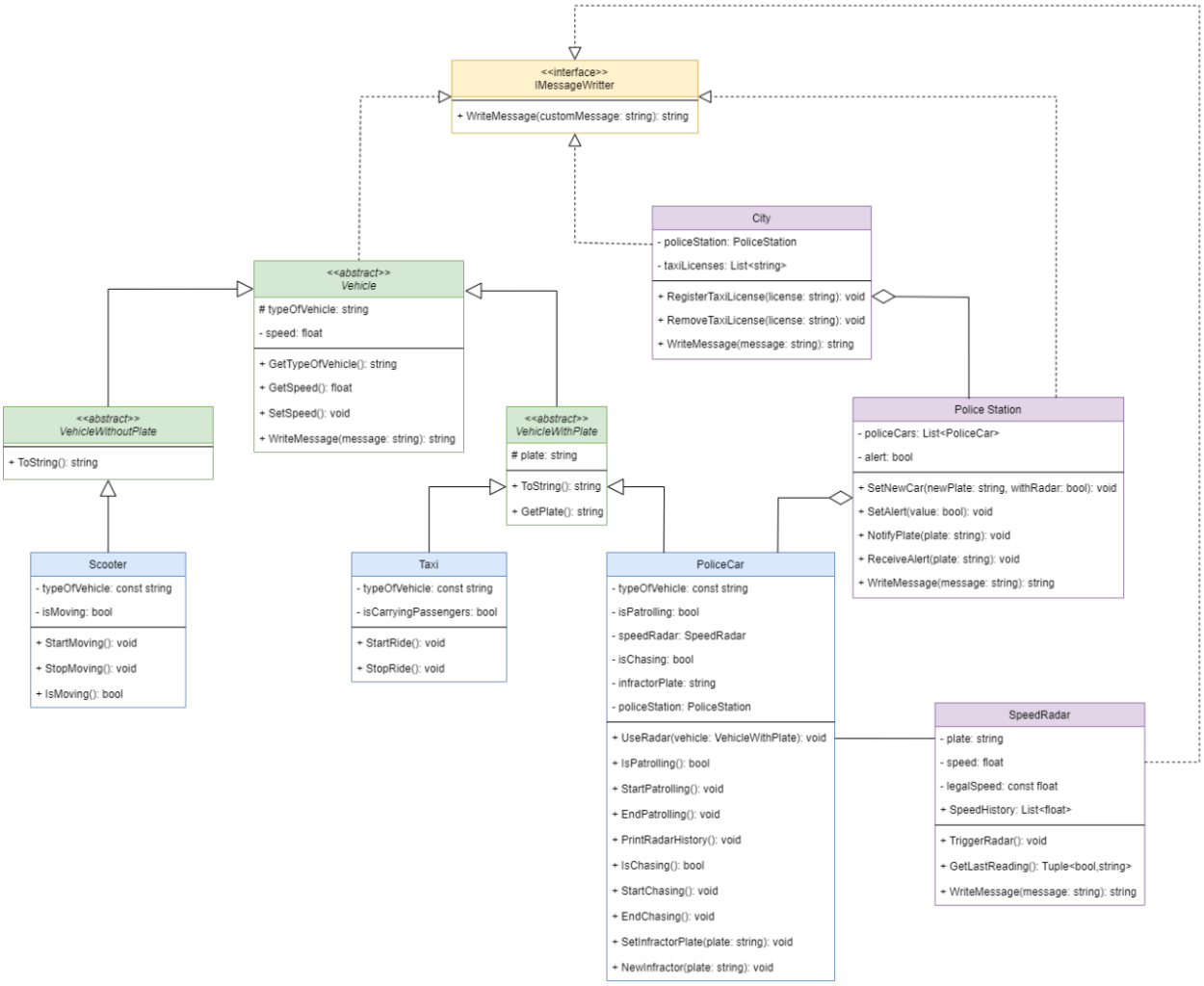
Diagrama UML



Repositorio GitHub: https://github.com/202215473/PyTP_practicas.git

Principios SOLID

Ya hemos visto en el diagrama UML el esquema principal del programa. Veamos cada uno de los principios SOLID y su cumplimiento dentro del código.

- **Single Responsibility Principle (SRP)**

Cada clase debe tener un único propósito (función, tarea, responsabilidad)

En nuestro caso esto se cumple, pues no hay ninguna clase que tenga más de un propósito. Por ejemplo, *PoliceCar* puede medir la velocidad de los coches que se encuentra gracias a que crea una instancia de *SpeedRadar*. El coche de policía no mide velocidad directamente, solo patrulla y, si es necesario, persigue a los coches que infringen la ley (esto último forma parte de la tarea de patrullar).

- Open/Close Principle (OCP)

Las entidades de software deben estar abiertas a extensión, pero cerradas a modificación.

Por ejemplo, tenemos la clase *Vehicle*, que tiene dos hijos: *VehicleWithPlate* y *VehicleWithoutPlate*. Estos, a su vez, extienden su aplicación ya que tienen distintos hijos (tipos de vehículos, *Taxi*, *PoliceCar*, *Scooter*). Los hijos parten de las clases de las que provienen y amplían su funcionalidad para implementar lo que se necesita en cada caso. *PoliceCar* y *Taxi* hacen cosas distintas, pero heredan *typeOfVehicle* de su "abuelo" y plate de su "padre", junto con los correspondientes métodos.

- Liskov Substitution Principle (LSP)

Los objetos de una clase derivada deben poder sustituir a los objetos de la clase base sin alterar el correcto funcionamiento del programa

En otras palabras, si A es hijo de B, puedo perfectamente meter un objeto A cuando se espere un objeto B. Esto se aplica en la clase *SpeedRadar*: cuando medimos la velocidad de un vehículo, esperamos obtener un objeto *VehicleWithPlate*. Sin embargo, podemos pasarle un *Taxi* sin problemas ya que *Taxi* es hijo de *VehicleWithPlate*.

- Interface Segregation Principle (ISP)

Los clientes no deben estar obligados a depender de interfaces que no utilizan

En vez de tener una interfaz grande con todos los métodos de todas las clases, es preferible tener varias interfaces más pequeñas y específicas. En nuestro caso este principio se mantiene, aunque sólo tenemos una gran interfaz, pero esto es porque su función es la de mostrar mensajes por pantalla, por lo que todas las clases usan el único método que tiene la interfaz.

- Dependency Inversion Principle (DIP)

Las dependencias deben ser de abstracciones y no de clases concretas

Esto se cumple en nuestro caso: todas las clases son abstractas excepto las clases hijo o aquellas que no dependen de otra. Por ejemplo, *City*, *PoliceStation* o *SpeedRadar* no son clases abstractas; como tampoco lo son *Taxi*, *PoliceCar* o *Scooter*. Sin embargo, *Vehicle*, *VehicleWithPlate* y *VehicleWithoutPlate* sí que lo son, ya que no van a ser instanciadas en ningún momento.

Pregunta propuesta

Ahora queremos que el policía pueda tener diferentes aparatos de medida, que pueden ser un medidor de velocidad (Radar) o un medidos 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. Con la arquitectura actual, ¿Qué principio SOLID incumpliríamos y cómo lo solucionarías?

Esto no lo podemos implementar directamente con el código que tenemos, pues estaríamos incumpliendo la L (LSP) de los principios SOLID. Cuando el coche de policía quiera usar un objeto de tipo radar, si resulta que le hemos metido un alcoholímetro, el programa fallará.

Una solución puede ser crear una clase padre llamada *MeasurementTool*, por ejemplo, que tenga *SpeedRadar* y *Breathalyser* como hijos. De esta forma, en el coche de policía se espera un objeto de tipo *MeasurementTool* en todos los métodos para que, independientemente del aparato de medida que posea el vehículo, el programa no presente errores.