COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI ICADE CIHS

Practice 2 – Software Architechture

Javier Escobar Serrano
Grupo A

Paradigmas y Técnicas de la programación

Índice

Introducción

Esta es la segunda práctica de la asignatura. Es una continuación de la primera.
Se trata de extender la estructura de la primera práctica utilizando los contenidos del segundo tema.

Objetivo:

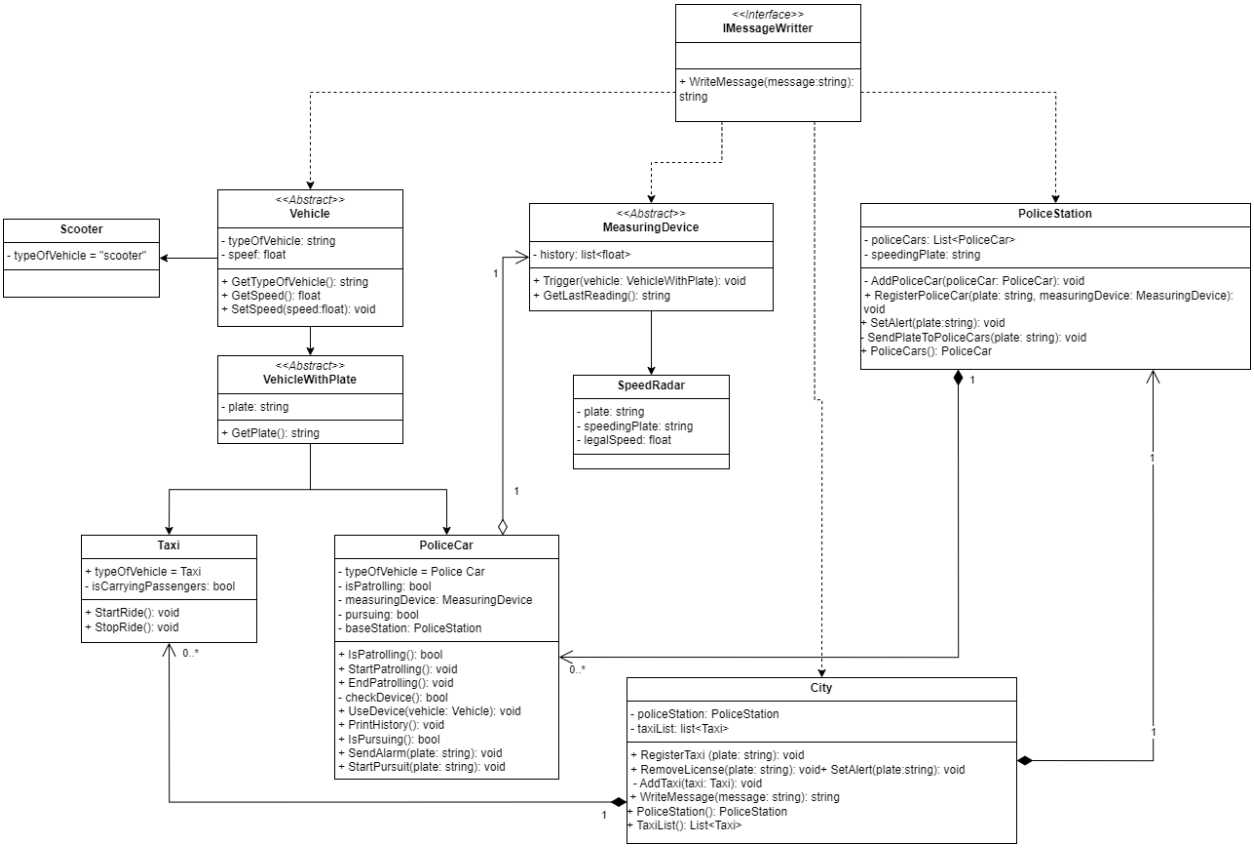
Representar la arquitectura de software de la Práctica 1 mediante un diagrama de clases UML, también se deben crear las historias de usuario y escalar esta arquitectura para añadir las nuevas funcionalidades en el diagrama UML y en el código de C#.

Descripción:

Para unificar las diferentes arquitecturas desarrolladas por los alumnos en la Práctica 1, se va a partir de la solución proporcionada por el profesor. También será necesario añadir nuevas clases en el diagrama UML para cumplir con los nuevos requisitos. Posteriormente implementar la nueva arquitectura al código de C# con un criterio crítico sobre la implementación.

Resultados

Diagrama UML:



Repositorio:

https://github.com/JES0406/tecnicas-y-paradigmas.git

Preguntas:

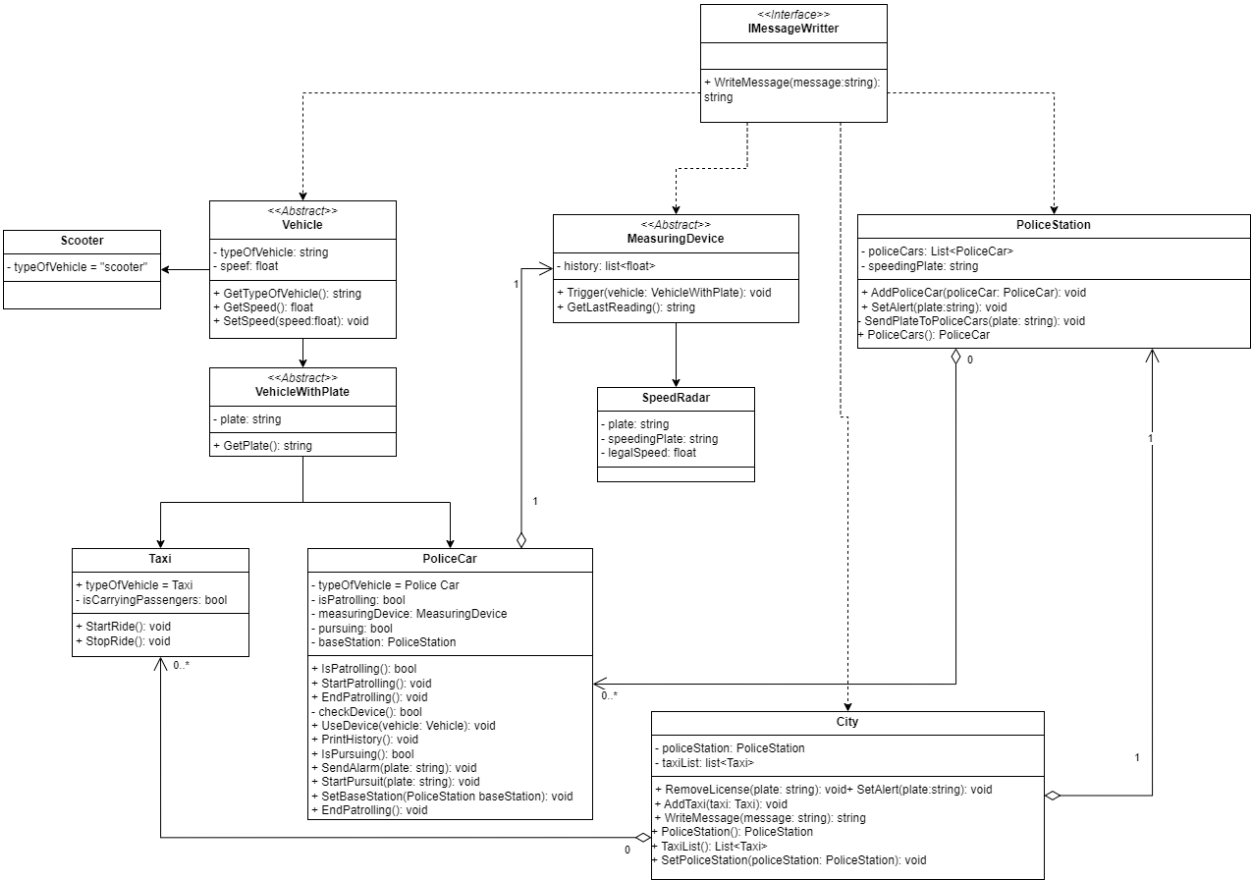
a) ¿Explicar brevemente si se están aplicando los principios SOLID dentro del código y el porqué de esa implementación?
   a. Single Responsibility Principle.
      Se trata de un principio para simplificar la lectura del código.
      Sin embargo, en city, measuringDevice, vehicle y PoliceStation no se respeta del todo al utilizarlo como controlador de lógica y escritor de mensajes. Esto es para no complicar en exceso el diagrama con conexiones innecesarias, así simplificando la estructura.
   b. Open/Closed Principle.
      Se trata de un principio para garantizar la extensión de las clases en futuras implementaciones.

En la clase city, se tiene un problema ya que si cambia la manera en la que los taxis se comportan, hay que modificar el comportamiento de la clase.
Pasa algo similar entre city y policeStation y policeStation y policeCar.
Para reparar esto, hay que cambiar el comportamiento de registerTaxi y registerPoliceCar.
Ahora estas reciben el objeto creado en vez de crearlo y simplemente lo añaden a sus respectivas listas.
Además, al cambiar esto, se pueden crear instancias de estos objetos sin que dependan de las capas superiores(city, PoliceStation, etc).
A su vez, en vez de mandar una policeStation en el constructor de un policeCar, creamos un método para que insertarlo de manera correcta.

c. Liskov Substitution Principle.
   Este principio trata sobre la herencia entre clases.
   En esta práctica no se rompe este principio.
d. Interface Segregation Principle.
   Este principio se centra en el uso correcto de las interfaces.
   En esta practica se rompe al implementar el IMessageWriter en todas las clases en las que se hace pues su función no es la de escribir en pantalla.
e. Dependency Inversion Principle.
   Parecido a la Open/Closed principle este principio trata de las dependencias con otras clases.
   En este caso, se rompe por los mismos motivos que en el principio mencionado.
   La solución es la misma.
   Sin embargo, tenemos ciertas abstracciones que hemos evitado para simplificar la estructura del código sacrificando abstracción.

b) 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?
   En mi caso, yo ya había dado con este caso y lo he arreglado, pero de no hacerlo se estarían incumpliendo el método Open/closed y el Dependency Inversión.
   Para arreglarlo he creado una clase abstracta MeasuringDevice y he usado un detector para tratar con el caso de no tener ningún dispositivo.

Nuevo UML:


<<Interface>>
IMessageWritter
+ WriteMessage(message:string): string
<<Abstract>>
Vehicle
- typeOfVehicle: string
- speef: float
+ GetTypeOfVehicle(): string
+ GetSpeed(): float
+ SetSpeed(speed:float): void
Scooter
- typeOfVehicle = "scooter"
<<Abstract>>
VehicleWithPlate
- plate: string
+ GetPlate(): string
<<Abstract>>
MeasuringDevice
- history: list<float>
+ Trigger(vehicle: VehicleWithPlate): void
+ GetLastReading(): string
SpeedRadar
- plate: string
- speedingPlate: string
- legalSpeed: float
PoliceStation
- policeCars: List<PoliceCar>
- speedingPlate: string
+ AddPoliceCar(policeCar: PoliceCar): void
+ SetAlert(plate:string): void
- SendPlateToPoliceCars(plate: string): void
+ PoliceCars(): PoliceCar
Taxi
+ typeOfVehicle = Taxi
- isCarryingPassengers: bool
+ StartRide(): void
+ StopRide(): void
PoliceCar
- typeOfVehicle = Police Car
- isPatrolling: bool
- measuringDevice: MeasuringDevice
- pursuing: bool
- baseStation: PoliceStation
+ IsPatrolling(): bool
+ StartPatrolling(): void
+ EndPatrolling(): void
- checkDevice(): bool
+ UseDevice(vehicle: Vehicle): void
+ PrintHistory(): void
+ IsPursuing(): bool
+ SendAlarm(plate: string): void
+ StartPursuit(plate: string): void
+ SetBaseStation(PoliceStation baseStation): void
+ EndPatrolling(): void
City
- policeStation: PoliceStation
- taxiList: list<Taxi>
+ RemoveLicense(plate: string): void+ SetAlert(plate:string): void
+ AddTaxi(taxi: Taxi): void
+ WriteMessage(message: string): string
+ PoliceStation(): PoliceStation
+ TaxiList(): List<Taxi>
+ SetPoliceStation(policeStation: PoliceStation): void
1
1
0
1
0..*
0..*
0
1