Actividad Autónoma - Programación II

Unidad 2: Principios Avanzados de Programación Orientada a Objetos (Python)

Tema: Vehículos Autónomos - Integración de Principios OOP y Patrones de Diseño

- Nombre: Maria Jose Vimos lluay
- Fecha: 30/04/2025
- Carrera: Ingenieria en Ciencia de Datos e Inteligencia Artificial
- Periodo Academico: 2025-2S
- Semestre: Segundo Semestre "A"

🎯 Objetivo

Diseñar una solución orientada a objetos en Python, aplicando los principios de OOP, patrones de diseño y sobrecarga de operadores.

In [6]:
```python
from abc import ABC, abstractmethod

#CREACION DE LA CLASE ABSTRACTA
class Vehiculo(ABC):
    def __init__ (self, identificadorunico, mode,velocidad_max, capacidad):
        self.__identificadorunico = identificadorunico
        self.__modelo = mode
        self.__velocidad_maxima = velocidad_max
        self.__capacidad_carga = capacidad
        self.__velo_actual = 0
        self._Strategy = None

    # Implementacion de getters y setters
    @property
    def identificador(self):
        return self.__identificadorunico

    @property
    def modelo(self):
        return self.__modelo

    @property
    def velocidad_maxima(self):
        return self.__velocidad_maxima

    @property
    def capacidad_carga(self):
        return self.__capacidad_carga

    @property
    def velo_actual(self):
```

```
        return self.__velo_actual

    #Metodos de Abstraccion
    @abstractmethod
    def acelerar(self):
        pass

    @abstractmethod
    def frenar(self):
        pass

    @abstractmethod
    def estado(self):
        pass

    # Strategy Pattern
    def set_strategy(self, strategy):
        self._Strategy = strategy

    def inicializar_strategy(self):
        if self._Strategy:
            return self._Strategy.ejecutar()
        else:
            return "No se asigna aun la estrategia"

    #Sobrecarga de operadores:
    def __eq__(self, otro):
        return self.modelo == otro.modelo and self.velocidad_maxima == otro.velo


  # HERENCIA : Crear subclases específicas (Automóvil, Camión, Motocicleta que s

class Camion(Vehiculo):
    def __init__(self, identificadorunico, modelo, velocidad_maxima, capacidad_c
       super().__init__(identificadorunico, modelo,velocidad_maxima, capacidad_c
       self._remolque = remolque

    @property
    def remolque(self):
        return self._remolque


    @remolque.setter
    def remolque(self, Si_remolque):
        self._remolque = Si_remolque

    #Nos va ayudar aumentar la velocidad del camion
    def acelerar(self):
        self._Vehiculo__velo_actual +=20

    #Va a disminuir la velocidad actual del camion

    def frenar(self):
        self._Vehiculo__velo_actual -=10

    def estado(self):
        estado_remolque = "Con Remolque" if self._remolque else "Sin Remolque"
        return f" Camion {self.modelo} ({self.identificador}) : Velocidad Actual

    #Engancha un remolque si no lo tiene y si ya tiene uno le avisa
```

```
    def engrana_remolques(self):
        if not self._remolque :
            self._remolque = True
            print("Remolque recien engranado")
        else:
            print("Este camion ya tiene un remolque")

  # CREACIONDE LA CLASE AUTOMOVIL

class Automovil (Vehiculo):
    def __init__(self, identificadorunico, modelo, velo_max, capacidad_carga, pa
        super().__init__(identificadorunico, modelo, velo_max, capacidad_carga)
        self._pasajeros= pasajeros

    @property
    def pasajeros(self):
        return self._pasajeros

    @pasajeros.setter
    def pasajeros(self, cantidad):
        self._pasajeros = cantidad

    def acelerar(self):
        #estrategia de emergencia
        emergencia = isinstance(self._Strategy, ConducciondeEmergencia)

        incremento = 10
        limite = self.velocidad_maxima
        if emergencia:
            limite +=30 #Nos permite superar la velocidad a 30


        if self._Vehiculo__velo_actual + incremento <= limite:
            self._Vehiculo__velo_actual += incremento
        else:
            self._Vehiculo__velo_actual = limite

    def frenar(self):
        if self._Vehiculo__velo_actual - 5 >= 0:
            self._Vehiculo__velo_actual -=5
        else :
            self._Vehiculo__velo_actual = 0

    def estado(self):
        return f"Automovil {self.modelo} con su {self.identificador} : Velocidad
               f"Pasajeros : {self._pasajeros} y una Capacidad de Carga de : {se

# CREACION DE LA CLASE MOTOCICLETA S
class Motocicleta(Vehiculo):

    def __init__(self, identificadorunico, modelo, velocidad_max, capacidad , ti
        super().__init__(identificadorunico, modelo, velocidad_max, capacidad)
        self._tipo_motor = tipo_motor

    @property
    def tipo_motor(self):
        return self._tipo_motor

    @tipo_motor.setter
    def tipo_motor(self, tipo):
```

```
        self._tipo_motor = tipo


    def acelerar(self):
        if self.velo_actual + 10 <= self.velocidad_maxima:
            self._Vehiculo__velo_actual += 10
        else:
            self._Vehiculo__velo_actual = self.velocidad_maxima


    def frenar(self):
        if self.velo_actual -5 >= 0:
            self._Vehiculo__velo_actual -= 5
        else:
            self._Vehiculo__velo_actual = 0


    def estado(self):
        return f"La Motocicleta es {self.modelo} y su identificador es {self.ide

  # Permite cambiar dinámicamente la estrategia de conduccion de Vehiculo
class Estrategiadeconduccion(ABC):
    @abstractmethod
    def ejecutar(self):
        pass

class ConduccionEconomica(Estrategiadeconduccion):
    def ejecutar(self):
        return "Se Activo la conduccion ECONOMICA"

class ConduccionDeportiva(Estrategiadeconduccion):
    def ejecutar(self):
        return "Se activo la conduccion DEPORTIVA"

class ConduccionOffRoad(Estrategiadeconduccion):
    def ejecutar(self):
        return "Se activo la conduccion OFF-ROAD"

 # Decorator Pattern: Agregar nuevas funcionalidades a los vehiculos
class Decorador(Vehiculo):
    def __init__(self, vehiculo):
        self._vehiculo = vehiculo

    def acelerar(self):
        self._vehiculo.acelerar()

    def frenar(self):
        self._vehiculo.frenar()

    def estado(self):
        return self._vehiculo.estado()

    @property
    def identificador(self):
        return self._vehiculo.identificador

    @property
    def modelo (self):
        return self._vehiculo.modelo
```

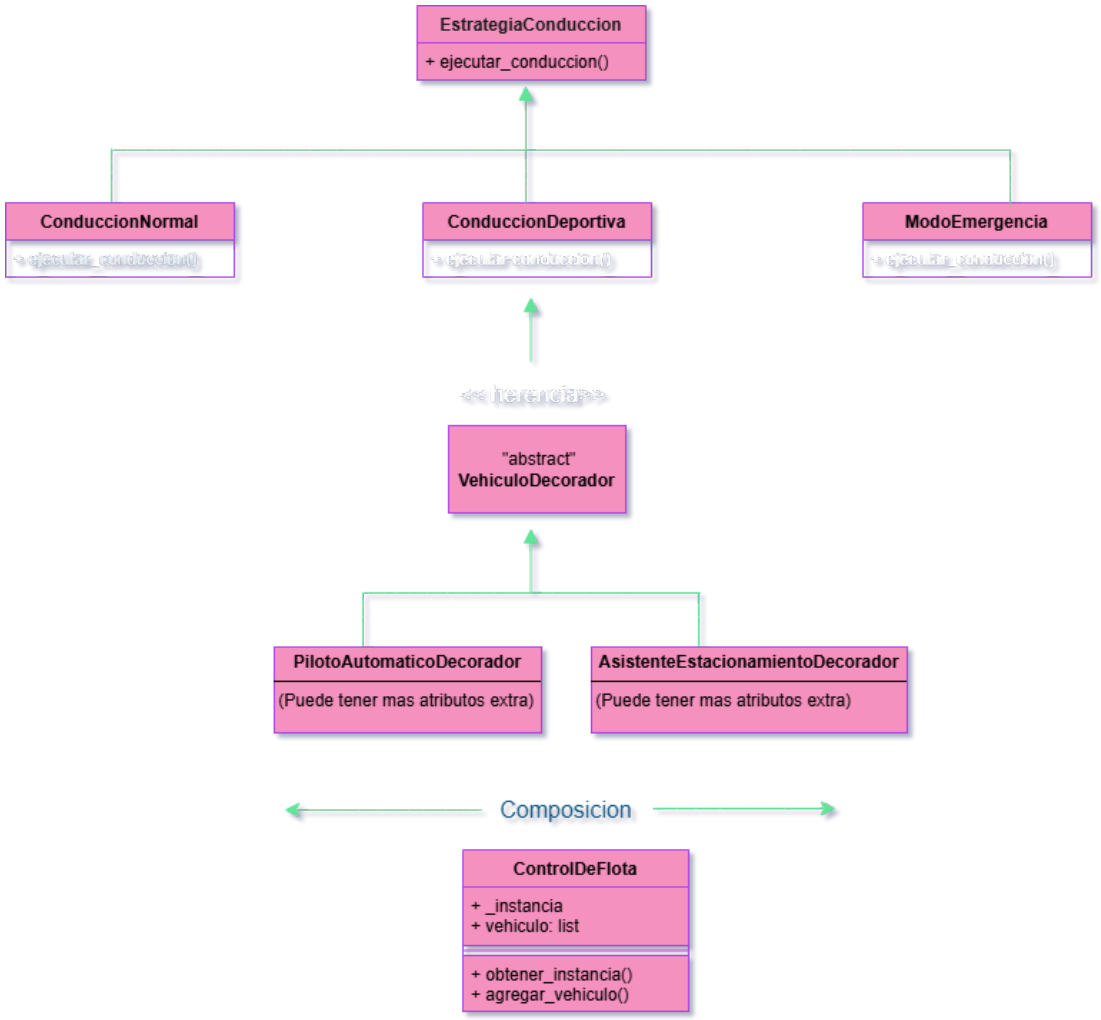
```
    @property
    def velocidad_maxima(self):
        return self._velocidad_maxima

    @property
    def capacidad_carga(self):
        return self._vehiculo.capacidad_carga

    @property
    def velo_actual(self):
        return self._vehiculo.velo_actual

class PilotoAuto(Decorador):
    def estado(self):
        return self._vehiculo.estado() +"Piloto Automatico"

class Asistente(Decorador):
    def estado(self):
        return self._vehiculo.estado() +"Asistente de Estacionamiento"
 #Singleton Pattern: Implementar un único ControlDeFlota que maneja toda la flot
class Control:
    _instancia = None

    def __new__(cls):
        if cls._instancia is None:
            cls._instancia = super(Control, cls).__new__(cls)
            cls._instancia._flota = []
        return cls._instancia

    def agregar_vehiculo(self, vehiculo):
        self._flota.append(vehiculo)

    def mostrarlaflota(self):
        for mostrar in self._flota:
            print(mostrar.estado())

    def __add__(self, vehiculo):
        self.agregar_vehiculo(vehiculo)
        return self

  # RETO ADICIONAL
class ConducciondeEmergencia(Estrategiadeconduccion):
    def ejecutar(self):
        return "MODO DE EMERGENCIA ACTIVADOO"


#EJEMPLO PARA PRUEBA DEL CODIGO
if __name__ =="__main__":
    #Creacion de los vehiculos
    carro1 = Automovil("B00", "Roll-Royce", 200, 600)
    moto1 = Motocicleta("A11", "Harley-Davidson", 120, 160)
    camion1 = Camion("D11", "Volvo", 200, 5000)

    print("-" * 60)
    print("-> CAMBIO A CONDUCCION DEPORTIVA")
    carro1.set_strategy(ConduccionDeportiva())
    print(carro1.inicializar_strategy())
    carro1.acelerar()
    carro1.frenar()
    print(carro1.estado())
```

```
#PERMITE CAMBIAR A LA ESTRATEGIA DE EMERGENCIA
print("-"* 60)
print("-> SE CAMBIO A CONDUCCION DE EMERGENCIA")
carro1.set_strategy(ConducciondeEmergencia())
print(carro1.inicializar_strategy())
for _ in range(15):
    carro1.acelerar()
print(carro1.estado())


#NOS VA A PERMITIR PROBAR EL REMOLQUE DEL CAMION
print("-" * 60)
print("-> PRUEBA DE REMOLQUES")
camion1.engrana_remolques()
camion1.acelerar()
print(camion1.estado())


#PROBAR LOS DECORADORES
print("-" * 60)
print("-> DECORADORES")
auto_decorador = PilotoAuto(carro1)
print(auto_decorador.estado())

moto_decorador = Asistente(moto1)
print(moto_decorador.estado())


#Probar el control de Flota
print("-" * 60)
print("-> CONTROL DE FLOTA")
flota = Control()
flota += carro1
flota += moto1
flota += camion1

print("Flota de los vehiculos Registrados: ")
flota.mostrarlaflota()
print("-" * 60)
```

```
------------------------------------------------------------
-> CAMBIO A CONDUCCION DEPORTIVA
Se activo la conduccion DEPORTIVA
Automovil Roll-Royce con su B00 : Velocidad Actual : 5 KM/H Pasajeros : 6 y una C
apacidad de Carga de : 600 Kg
------------------------------------------------------------
-> SE CAMBIO A CONDUCCION DE EMERGENCIA
MODO DE EMERGENCIA ACTIVADOO
Automovil Roll-Royce con su B00 : Velocidad Actual : 155 KM/H Pasajeros : 6 y una
Capacidad de Carga de : 600 Kg
------------------------------------------------------------
-> PRUEBA DE REMOLQUES
Remolque recien engranado
 Camion Volvo (D11) : Velocidad Actual 20 KM/H, Con Remolque, capacidad de carga
5000 Kg
------------------------------------------------------------
-> DECORADORES
Automovil Roll-Royce con su B00 : Velocidad Actual : 155 KM/H Pasajeros : 6 y una
Capacidad de Carga de : 600 KgPiloto Automatico
La Motocicleta es Harley-Davidson y su identificador es A11 : Su velocidad Actual
0 KM/H ,Tipo de Motor: Combustion, Capacidad de Carga: 160 KgAsistente de Estacio
namiento
------------------------------------------------------------
-> CONTROL DE FLOTA
Flota de los vehiculos Registrados:
Automovil Roll-Royce con su B00 : Velocidad Actual : 155 KM/H Pasajeros : 6 y una
Capacidad de Carga de : 600 Kg
La Motocicleta es Harley-Davidson y su identificador es A11 : Su velocidad Actual
0 KM/H ,Tipo de Motor: Combustion, Capacidad de Carga: 160 Kg
 Camion Volvo (D11) : Velocidad Actual 20 KM/H, Con Remolque, capacidad de carga
5000 Kg
------------------------------------------------------------
```

📐 Diagramas UML requeridos

- Diagrama de clases (con herencia, estrategias, decoradores, singleton).
- Diagrama de relaciones (composición, asociación).



📋 Explicación breve

Para poder desarrollar esta actividad autonoma sobre los Vehiculos Autonomos aplique los principios Fundamentales de la programacion orientada a objetos junto a poatrones de diseño . Lo primero que hice fue definir una clase abastracta llamada "Vehiculo" que tiene atributos y metodos para (Automoviles, Motocicletas y Camión) este nos va a permitir poder reutilizar el codigo .

Tambien utilice un interfaz que se llama "EstrategiaConduccion" Nos permite implementar el conocido PATRÓN STRATEGY este nos facilita cambiar el comportamiento de la conduccion como (normal,deportivo o emergencia) durante su ejecucion sin modificar las clases de los vehiculos.

Se aladio funcionalidades como EL PILOTO AUTOMATICO o el ASISTENTE DE ESTACIONAMIENTO sin la necesidad de alterar las clases base para lo que se aplico el patron decorador mediante una clse abastracta "VehiculoDecorador" con sus derivaciones.

Para finalizar implemente el patron SINGLETON en la clase del "ControldeFlota" para asi garantizar una unica instancia de administracion de vehiculos que nos permite tener el contol eficiente de nuestro sistemas.