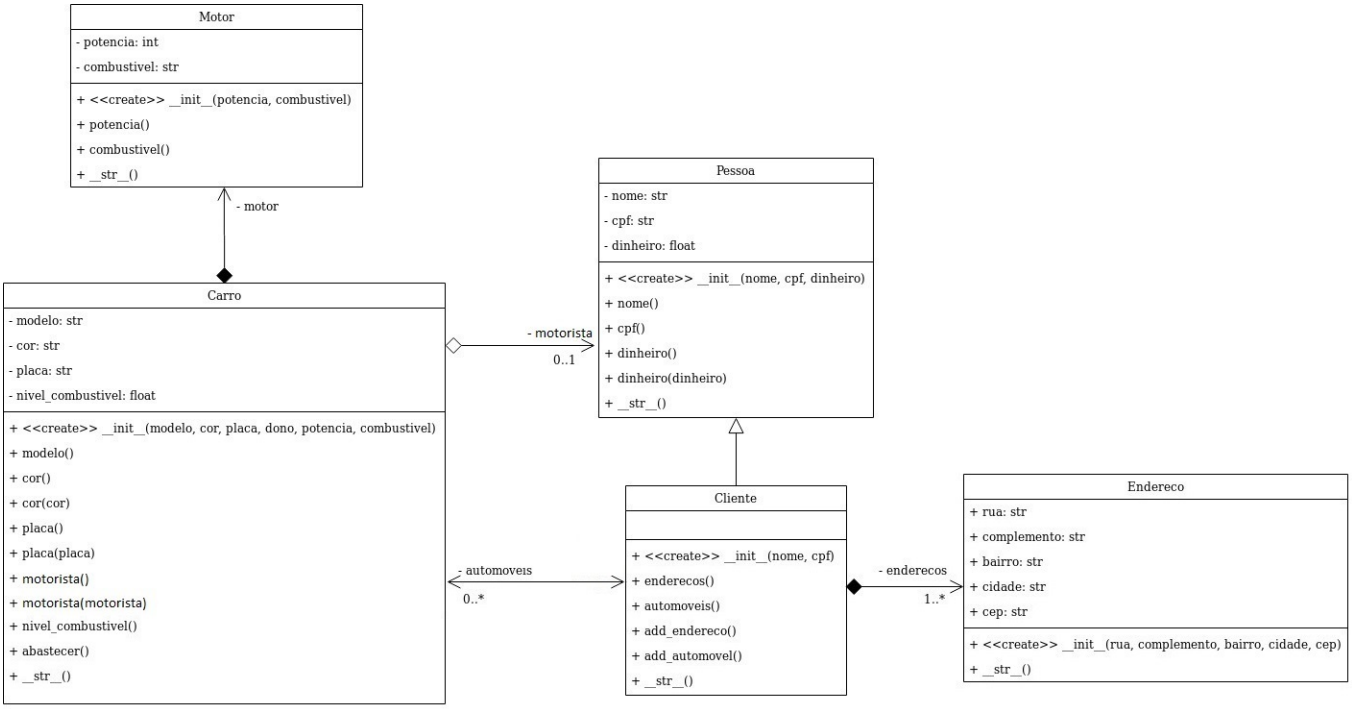
Agregação e Composição

Em um sistema desenvolvido usando o paradigma de Orientação a Objetos, é comum que uma classe possua como atributos referências a objetos de outras classes. Essas relações são chamadas de `Associações`. Dois casos especiais de associação são a `Agregação` e a `Composição`.

Um problema-exemplo

Vamos analisar um problema simplificado: um motorista de aplicativo precisa controlar os dados do seu veículo e o uso e gastos com combustível. Um Carro tem o seu Motor, e para este problema, podemos considerar que o carro pode conter um motorista. O motorista é uma Pessoa que possui dinheiro para poder abastecer o seu Carro. Existem também os Clientes que possuem seus endereços. É necessário que os Clientes mantenham uma lista dos carros que já utilizaram.

Diagrama de classes



Implementando as classes do exemplo

Primeiro vamos declarar uma classe `Pessoa` com os atributos `__nome`, `__cpf` e `__dinheiro`.

> Para fins didáticos, vamos escrever o código do método especial do python `__str__()`, que define o modo que um objeto dessa classe é convertido em um objeto do tipo string. Vamos também imprimir uma mensagem no construtor dessa classe.

In [ ]:

```
class Pessoa:

    def __init__(self, nome: str, cpf: str, dinheiro: float):
        self.__nome = nome
        self.__cpf = cpf
        self.__dinheiro = dinheiro
        print(self, 'criado.')

    @property
    def nome(self):
        return self.__nome

    @property
    def cpf(self):
        return self.__cpf

    @property
    def dinheiro(self):
        return round(self.__dinheiro, 2)

    @dinheiro.setter
    def dinheiro(self, dinheiro):
        self.__dinheiro = dinheiro

    def __str__(self):
        return 'Pessoa de nome ' + self.nome
```

Agora, vamos declarar uma classe `Motor`, que representa o motor de um carro. O método `__str__()` também é definido nesta classe e uma mensagem é impressa no construtor.

In [ ]:

```
class Motor:

    def __init__(self, potencia: int, combustivel: str):
        self.__potencia = potencia
        self.__combustivel = combustivel
        print(self, 'criado.')

    @property
    def potencia(self):
        return self.__potencia

    @property
    def combustivel(self):
        return self.__combustivel

    def __str__(self):
        return 'Motor de potencia ' + str(self.potencia)
```

Finalmente, vamos declarar uma classe `Carro` que se relacionará com as duas classes anteriores.

> A saber, todo carro contém um motorista, especialmente quando estiver em movimento ;-) O motorista é um objeto da classe `Pessoa`. O Carro também contém o seu motor, que é um objeto da classe `Motor`.

A classe `Carro` tem um método `abastecer` que acessa ambos os objetos. O método `__str__()` também é definido e uma mensagem de instanciação é impressa no construtor.

In [ ]:

```
class Carro:

    def __init__(self, modelo: str, cor: str, placa: str, motorista: Pessoa, potencia: in
        self.__modelo = modelo
        self.__cor = cor
        self.__placa = placa
        self.__motorista = motorista
        self.__motor = Motor(potencia, combustivel)
        self.__nivel_combustivel = 0
        print(self, 'criado.')

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

    @property
    def cor(self):
        return self.__cor

    @cor.setter
    def cor(self, cor: str):
        self.__cor = cor

    @property
    def placa(self):
        return self.__placa

    @placa.setter
    def placa(self, placa: str):
        self.__placa = placa

    @property
    def potencia(self):
        return self.__motor.potencia

    @property
    def motorista(self):
        return self.__motorista

    @motorista.setter
    def motorista(self, motorista: Pessoa):
        self.__motorista = motorista

    @property
    def nivel_combustivel(self):
        return self.__nivel_combustivel

    def abastecer(self, litros: int):
        total = 0
        if self.__motor.combustivel == 'gasolina':
            total = litros * 4.13
        elif self.__motor.combustivel == 'alcool':
            total = litros * 3.49
        if total < self.motorista.dinheiro:
            self.motorista.dinheiro -= total
            self.__nivel_combustivel += litros

    def __str__(self):
        return 'Carro de modelo ' + self.modelo
```

E agora vamos instanciar alguns objetos destas classes.

In [ ]:

```
guilherme = Pessoa('Guilherme', '123.456.789-0', 10000)
jean = Pessoa('Jean', '987.654.321-0', 100)

carro1 = Carro('Gol', 'Vermelho', 'ABC-1234', guilherme, 75, 'gasolina')
carro2 = Carro('Fox', 'Prata', 'ABC-9999', jean, 104, 'alcool')
carro3 = Carro('Fusca', 'Azul', 'ABC-0000', jean, 65, 'gasolina')
```

> Note que embora pareça que apenas 5 objetos foram instanciados, o construtor da classe `Carro` instanciou um novo objeto da classe `Motor` para cada carro.

E aqui podemos fazer algumas manipulações com os objetos instanciados. Por exemplo, abastecer os carros e imprimir algumas informações.

In [ ]:

```
print('Guilherme: R$', guilherme.dinheiro, 'Jean: R$', jean.dinheiro)
carro1.abastecer(10)
carro2.abastecer(5)
carro3.abastecer(20)
carro2.abastecer(8)
print('Guilherme: R$', guilherme.dinheiro, 'Jean R$:', jean.dinheiro)
print(carro1.motorista.nome, carro1.modelo, carro1.potencia, carro1.nivel_combustivel)
print(carro2.motorista.nome, carro2.modelo, carro2.potencia, carro2.nivel_combustivel)
print(carro3.motorista.nome, carro3.modelo, carro3.potencia, carro3.nivel_combustivel)
```

E agora vejamos o que acontece ao destruírmos o objeto que representa o fusca.

Para isso, apenas para fins didáticos, vamos primeiro definir uma função usando o pacote `gc` que será responsável por listar todos os objetos instanciados de uma determinada classe. Vamos chamar essa função para as classes `Pessoa`, `Motor` e `Carro` antes e depois de destruir o objeto `carro3`, que no nosso caso está representando o fusca. Também utilizaremos o comando `del` que permite remover um determinado objeto da memória.

> Mais informações sobre gc.get_objects() e del():
> https://docs.python.org/3/library/gc.html#gc.get_objects (https://docs.python.org/3/library/gc.html#gc.get_objects)
> https://docs.python.org/3/tutorial/datastructures.html#the-del-statement (https://docs.python.org/3/tutorial/datastructures.html#the-del-statement)

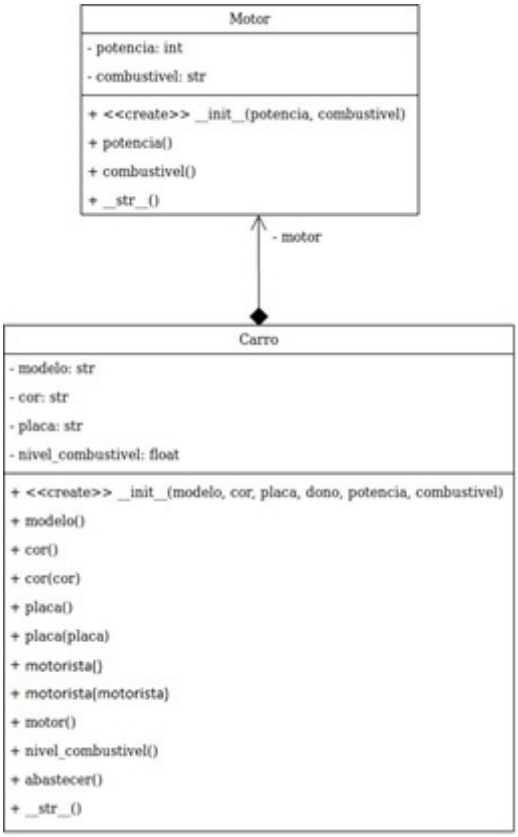
In [ ]:

```
import gc

def lista_instancias(Classe):
    for obj in gc.get_objects():
        if isinstance(obj, Classe):
            print(id(obj), obj)

print('Instâncias:')
lista_instancias(Pessoa)
lista_instancias(Motor)
lista_instancias(Carro)

print('------------')
print('Destruindo o carro', carro3.modelo)
del(carro3)
print('------------')
print('Instâncias:')
lista_instancias(Pessoa)
lista_instancias(Motor)
lista_instancias(Carro)
```

Note como é forte a relação entre as classes `Carro` e `Motor`:

Ao destruírmos o objeto `carro3`, referente ao Fusca, o objeto motor de potência 65 deste carro também foi automaticamente destruído. O mesmo não aconteceu com objeto da classe `Pessoa` que é o motorista do carro.

O motor é instanciado **diretamente no construtor** da classe `Carro`, de modo que o motor instanciado está diretamente relacionado com o respectivo objeto carro. Se um carro for destruído, **o motor relacionado à ele também será**. Note que o motor de um carro **não pode mudar** e que para construir um objeto da classe `Carro` é preciso passar como parâmetros as informações do motor (`potencia` e `combustível`), mas não um objeto da classe `Motor`. Um motor está relacionado com **um único** carro.

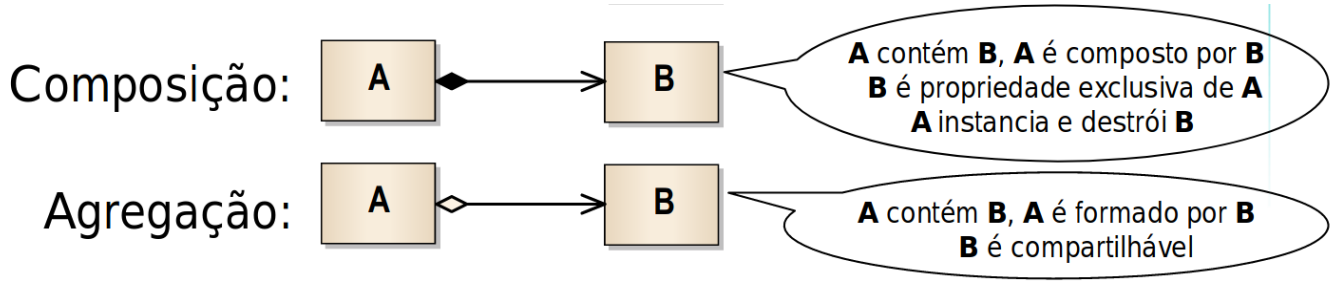


A relação entre as classes `Carro` e `Pessoa` é mais fraca: objetos da classe `Pessoa` são instanciados "**fora**" da classe `Carro` e um desses objetos é passado ao construtor desta classe como sendo o `motorista` do carro. Se um carro for destruído, o seu motorista continua existindo. É possível alterar o motorista de um carro e cada pessoa pode estar relacionada com diversos carros.

Em cada um desses casos temos um tipo específico de associação:

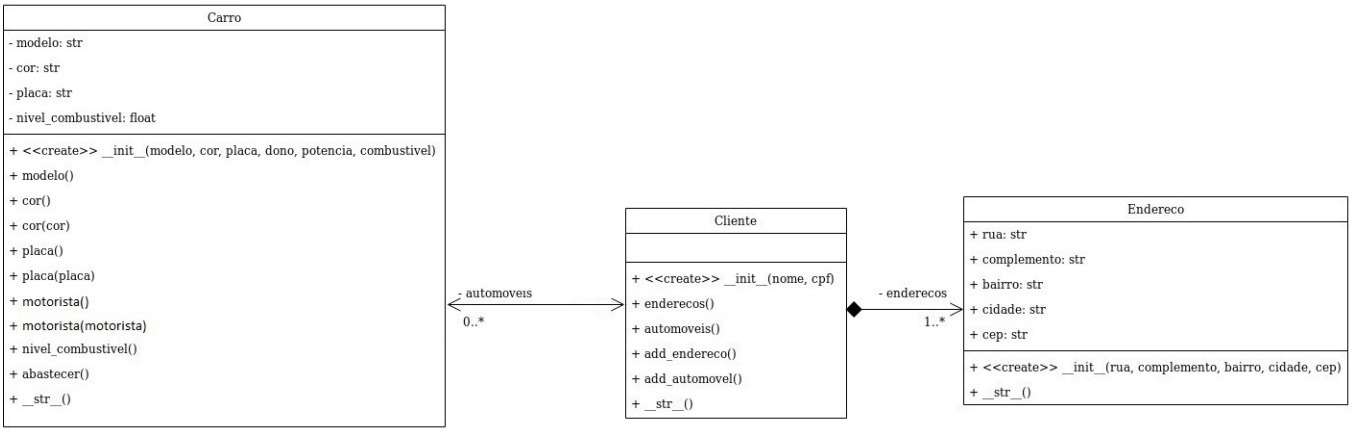
- **Composição:** *a parte não existe sem o objeto*. Todo carro tem um motor, e esse motor só existe dentro de um único carro. No contexto deste sistema, podemos pensar que um motor não pode existir sem que esteja dentro de um carro. Se um carro se move, o motor move-se junto. Se um carro é destruído, o seu motor também é destruído. A forma mais usual de tratar composição é instanciar o objeto "parte" diretamente **dentro** da classe que a contém, garantindo que sua existência dependa da existência do objeto "todo".
- **Agregação:** *a parte pode ser compartilhada com outros objetos*. Esse é o caso do nosso exemplo `Pessoa`, pois o atributo `motorista` da classe `Carro` faz referência a um objeto do tipo `Pessoa` que pode estar agregado a vários outros objetos (a mesma pessoa pode ser motorista de vários carros). Além disso, mesmo que um carro seja destruído, a pessoa (seu motorista) continua existindo e continua podendo ser motorista de outros carros. Assim, o objeto da classe `Pessoa` deve ser instanciado **fora** da classe `Carro`.

Nota-se, portanto, que a **composição** é um tipo de relação mais forte que a **agregação**.



E o Cliente?

Considere o caso dos clientes que utilizam o carro pelo aplicativo.



> Vamos definir uma classe `Cliente` que estende a classe `Pessoa`, adicionado os atributos `enderecos` e `automoveis`.

> O atributo `enderecos` é uma lista de objetos da classe `Endereco`, definida abaixo:

In [ ]:

```
class Endereco:

    def __init__(self, rua: str = "", complemento: str = "", bairro: str = "", cidade: st
        self.rua = rua
        self.complemento = complemento
        self.bairro = bairro
        self.cidade = cidade
        self.cep = cep
        print(self, 'criado.')

    def __str__(self):
        return 'Endereco: ' + self.rua + ', ' + self.complemento
```

> Ou seja, estamos considerando que um cliente pode ter vários endereços.

E agora a classe `Cliente`, onde o atributo `automoveis` é uma lista de carros que ele já utilizou. São definidos métodos para adicionar novo endereço e novo automóvel ao cliente.

> Aqui, para fins de simplificação, não vamos definir os métodos para remover ou alterar endereços ou automóveis

In [ ]:

```
class Cliente(Pessoa):

    def __init__(self, nome, cpf):
        super().__init__(nome, cpf, 0)
        self.__enderecos = []
        self.__automoveis = []

    @property
    def enderecos(self):
        return self.__enderecos

    @property
    def automoveis(self):
        return self.__automoveis

    def add_endereco(self, rua: str = "", complemento: str = "", bairro: str = "", cidade
        self.__enderecos.append(Endereco(rua, complemento, bairro, cidade, cep))

    def add_automovel(self, carro: Carro):
        self.__automoveis.append(carro)

    def __str__(self):
        return 'Cliente ' + self.nome
```

Note as diferenças entre os atributos enderecos e automoveis

- um novo endereço é instanciado **dentro** do método `add_endereco` e pertence unicamente a este objeto da classe `Cliente` ;
- para adicionar um novo endereço, é preciso chamar o método `add_endereco` passando como parâmetros **as informações** do endereço;
- para adicionar um novo automóvel, é preciso chamar o método `add_automovel` passando como parâmetro **o objeto** da classe `Carro` , que deve ter sido anteriormente instanciado **fora** do método `add_automovel` .

> Neste caso, a relação entre as classes `Cliente` e `Endereço` é uma relação de **composição**, enquanto a relação entre as classes `Cliente` e `Carro` é somente uma relação de **associação** bidirecional, pois nem há uma relação de todo-parte nesse caso.

Observe que, ao destruir um cliente, todos os seus endereços também são destruídos, o que não ocorre com os seus automóveis, que continuam a existir. Para testar isso vamos primeiro instanciar alguns objetos e imprimir algumas informações:

In [ ]:

```
cliente = Cliente('Fulano', '123.456.789-0')
cliente.add_endereco('Rua 1', 'n 42', 'Bairro 1', 'Cidade 1', '01234-567')
cliente.add_endereco('Rua 2', 'n 01, apto 0', 'Bairro 2', 'Cidade 2', '98765-123')

motorista = Pessoa('Jean', '987.654.321-0', 1000)

carro4 = Carro('Uno', 'Branco', 'XYZ-1234', motorista, 80, 'gasolina')
carro5 = Carro('Ka', 'Prata', 'XYZ-9999', motorista, 100, 'alcool')

cliente.add_automovel(carro4)
cliente.add_automovel(carro5)

print('Id do Cliente:', id(cliente))

def lista_enderecos(cliente):
    print('------------')
    print('Endereços do', cliente.nome)
    for end in cliente.enderecos:
        print(end.rua, end.complemento, end.bairro, end.cidade, end.cep)

def lista_automoveis(cliente):
    print('------------')
    print('Automóveis do', cliente.nome)
    for carro in cliente.automoveis:
        print(carro.modelo, carro.cor, carro.placa)

lista_enderecos(cliente)
lista_automoveis(cliente)
```

O que acontece quando destruimos o cliente?

E agora vamos destruir o objeto `cliente`, mostrando todas as instâncias que temos antes e depois da sua destruição. Note o que acontece com os objetos das classes `Carro` e `Endereco`.

In [ ]:

```
print('Instâncias:')
lista_instancias(Cliente)
lista_instancias(Endereco)
lista_instancias(Carro)

print('------------')
#print('Destruindo o cliente', cliente.nome)

del(cliente)

print('------------')
print('Instâncias:')
lista_instancias(Cliente)
lista_instancias(Endereco)
lista_instancias(Carro)
```

Objetos da classe Endereco e Carro têm comportamentos diferentes após a

exclusão do cliente

Note que os objetos da classe `Endereco` foram destruídos, enquanto os objetos da classe `Carro` foram mantidos.

Isso mostra na prática uma importante diferença da **composição** para outros tipos de associação.