L'objectif de cet exercice est d'appliquer les concepts de la Programmation Orientée Objet en utilisant le design pattern Factory. Vous devrez créer quatre classes : `Vehicule`, `Voiture` et `Moto`

> Ce design pattern sert à centraliser pléthore d'instanciation de classe différente grâce à une seule méthode.

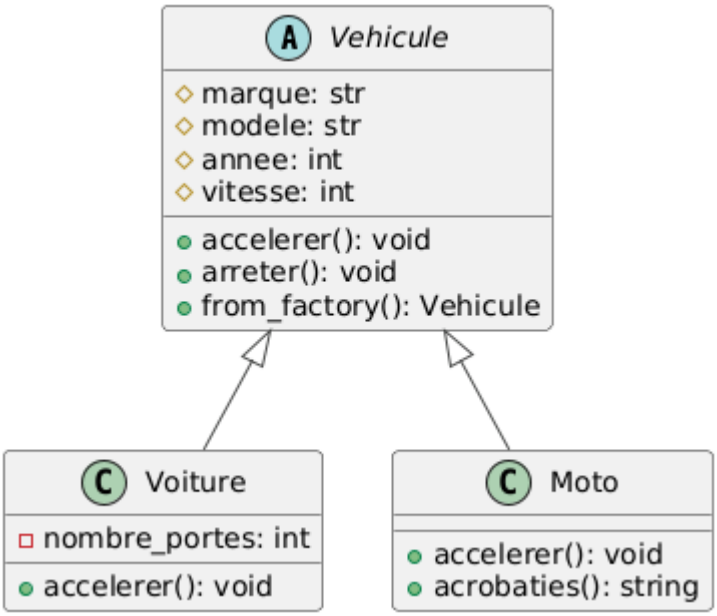
Pour ce faire, on va implémenter petit à petit les concepts plutôt que faire tout d'un coup et se rendre compte que ça ne fonctionne pas. Cela nous fera perdre plus de temps que ce que l'on pourrait en gagner. **Ceci est une méthode agile et permet d'avoir constamment une application fonctionnelle.**

> [!Info] Le typage Le typage est apparus avec python 3.0 pour les variables/attributs et en python 3.5 pour les retours de fonctions/méthodes.
>
> Il est fortement recommandé de l'utiliser, mais ne sert qu'à titre informatif.

> [!TIP] Documentation Comme dans tout travail, pensez à faire de la documentation.

Représentation UML

- © représente une classe
- Ⓐ représente une classe abstraite
- □ représente un attribut privé
- ♢ représente un attribut protected
- ○ représente une méthode public



> [!TIP] Visibilité d'attributs Quand on entend attribut protected ou private, on pense getter et setter

Création des classes

On va tout d'abord créer les 3 classes les plus basiques possibles au format PascalCase, c'est à dire avec la première lettre de chaque mot collé et en majuscule : `VehiculeFactory`

```
class Vehicule:
 def __init__(self):
  pass
```

```
class Voiture:
 def __init__(self):
  pass
```

```
class Moto:
 def __init__(self):
  pass
```

Héritage

Maintenant que nos classes sont créées, nous allons pouvoir leur instaurer de l'héritage qui aura pour but de réduire la duplication de code, et donc de faciliter la maintenance.

Il a aussi un autre gros avantage, c'est que grâce à l'héritage, on peut faire de la généralisation de traitement. Exemple : faire une boucle sur tous les véhicules pour tous leur faire faire la même méthode.

```
class Voiture(Vehicule):
 def __init__(self):
  pass
```

```
class Moto(Vehicule):
 def __init__(self):
  pass
```

Abstraction

L'abstraction va de concert avec l'héritage afin de réduire la duplication de code, mais il a aussi une autre fonctionnalité majeure. **Une classe abstraite ne peut pas être instancié.** Le but de cela est de **ne pas**

pouvoir créer des objets amalgame qui n'ont pas de forme ou de représentation fixe comme une voiture.

Pour définir en théorie une classe abstraite en python, il faut soit déclarer un héritage entre notre classe et la classe `abc.ABC`

```
from abc import ABC

class Vehicule(ABC):
```

Soit, il faut déclarer une méthode avec l'annotation `@abstractmethod`

```
from abc import abstractmethod

@abstractmethod
def accelerer(self) -> None:
 pass
```

Sauf qu'en pratique, si la classe n'a pas de méthode abstraite, et peut quand même être instanciée.

> Donc pour rendre une classe abstraite, elle dit absolument avoir au moins une méthode abstraite.

> [!TIP] Les retours de fonctions la partie `-> None` du code au-dessus permet de fixer ce que devra renvoyer la méthode.
>
> Ici, il renvoi un `None`, car dans mon code, je l'utilisais pour de l'affichage. Du coup, la méthode ne doit rien renvoyer.

Attribut

Maintenant que 3 classes et leurs relations sont définis, on va pouvoir définir leurs attributs.

Il faut se demander si l'attribut que l'on veut ajouter pourra se mettre dans Voiture et/ou dans Moto :

- Si on peut la mettre que dans une seule des 2, alors on l'a lui affecte.
- Sinon on l'affecte à véhicule, car elle mutualise l'information.

Je veux ajouter les attributs que l'on écrit au format snake_case: `marque`, `modele`, `annee`, `vitesse` et `nb_portes`

On se pose la question : Ou je mets mes attributs ?

Comme `marque`, `modèle`, `année` et `vitesse` peuvent être dans `Voiture` et `Moto`, alors on les mets dans `Vehicule`.

```
class Vehicule(ABC):
 def __init__(self, marque: str, modele: str, annee: int):
     self._marque: str = marque
     self._modele: str = modele
     self._annee: int = annee
     self._vitesse: int = 0
```

> [!TIP] Les affectation d'attributs On peut décider d'affecter des informations à la création de l'objet en le mettant dans la signature de fonction : `def __init__(self, marque: str, modele: str, annee: int):`, ou alors de les affecter avec une valeur par défaut : `self._vitesse: int = 0`.
>
> Il est aussi possible de laisser le choix de remplir un attribut à la création d'un Objet, mais avec une valeur par défaut, ce qui rend l'attribut optionnel pour créer l'objet : `def __init__(self, marque: str, modele: str, annee: int, vitesse: int = 0):`

Vous vous demandez pourquoi on a mis un '_' devant mes attributs ?

> [!Info] La visibilité des attributs Ceci est une convention utilisé en python est consiste à laisser ou non l'accès à un attribut :
>
> ```
> self.publique  # attribut accessible par tout le monde
> self._proctected  # attribut accessible uniquement par héritage
> self.__private  # attribut accessible uniquement par la classe qui la possède
> ```
>
> Dans d'autre langage, ce n'est pas une convention mais il est instaurer en dur avec des mots clé, comme en java.

Comme il n'y a que la voiture qui possède des portes, je lui affecte l'attribut privé

```
class Voiture(Vehicule):
    def __init__(self, marque: str, modele: str, annee: int, nombre_portes: int):
        super().__init__(marque, modele, annee)
        self.__nombre_portes: int = nombre_portes
```

> [!Info] Le mot clé `super()` C'est la façon pour une classe fille qu'elle a pour utiliser une méthode de sa classe mère.
>
> Ici, elle appelle le `__init__()` de sa classe mère afin qu'elle fasse le traitement de ses attributs à sa place.

Les méthodes

Les méthodes sont les actions que peux faire une classe. Je veux que mes véhicules puissent `accelerer` et `arreter` Je veux aussi que ma moto puisse `acrobaties` comme un 'Wheel-in!'.

```
from abc import ABC, abstractmethod

class Vehicule(ABC):
 ...

 def demarrer(self) -> str:
     return f"{self.marque} {self.modele} démarre."

 @abstractmethod
 def accelerer(self) -> str:
     pass

 def arreter(self) -> str:
     self.vitesse = 0
```

Comme on a déjà défini `demarrer`, `accelerer` et `arreter` dans la classe `Vehicule`, on aura plus besoin de les réécrire, sauf `accelerer` car elle est abstraite.

```
class Voiture:
 ...

 def accelerer(self) -> None:
     self.vitesse += 10
```

Pareil que pour la Voiture, sauf qu'on doit aussi ajouter la méthode `faire_des_acrobaties` qui est spécifique à la Moto

```
class Moto:
 ...

 def accelerer(self) -> None:
     self.vitesse += 30

 def acrobaties(self) -> str:
  print("La Moto fais un wheel-in")
```

L'encapsulation

L'encapsulation permet principalement de protéger les attributs et les méthodes `protected` et `private`. Il a divers autre utilisation mais que je ne vais pas tout détailler.

> Les méthodes `accelerer` et `arreter` sont de l'encapsulation, car on ne donne qu'une quantitée limité d'action effectuable sur un attribut.

Une fonctionnalité de l'encapsulation est les `getters` et les `setters` :

```
@property
def marque(self) -> str:
    return self._marque

@marque.setter
def marque(self, valeur) -> None:
    self._marque = valeur
```

> [!info] getters / setters Il est très intéressant d'en mettre uniquement à partir du moment où on doit faire des traitements sur l'attribut.
>
> ```
> @nombre_portes.setter
> def nombre_portes(self, valeur) -> None:
>     if valeur > 2:
>         self.__nombre_portes = valeur
>     else:
>         raise ValueError("Le nombre de portes doit être supérieur à 2.")
> ```
>
> Ici, on ne peut pas créer une voiture avec moins de 2 portes.

> [!TIP] Vous pouvez utiliser getters et les setters dans le constructeur `__init__` afin de restreindre aussi à la création les restrictions des attributs, et que l'on ne puisse pas créer de voiture sans porte.
>
> ```
> class Voiture(Vehicule):
>    def __init__(self, marque: str, modele: str, annee: int, nombre_portes:
> int = 2):
>        super().__init__(marque, modele, annee)
>        self.nombre_portes: int = nombre_portes
> ```
>
> on a remplacé le `self.__nombre_portes` par `self.nombre_portes`
>
> > Il est aussi possible de faire pareil pour les méthodes.

Implémentation de la Factory

Maintenant que nos 2 Entités `Moto` et `Voiture` sont créées, on va pouvoir faire notre Factory.

La fabrique ne contiendra qu'une seule méthode statique qui aura pour but de créer le bon objet avec les bons attributs quand on l'appellera.

```
@classmethod
def from_factory(cls, *args) -> "Vehicule":
    if cls == Vehicule:
        raise NotImplementedError("Utilisez une sous-classe de Vehicule.")
    return cls(*args)
```

On peut maintenant l'appeler dans le `main.py` afin de créer des `Vehicules`.

```
from voiture import Voiture
from moto import Moto

# CREATION DES VÉHICULES
print(f"Création d'une voiture")
voiture: Voiture = Voiture.from_factory("Toyota", "Camry", 2022, 4)
print(f"Création d'une moto")
moto: Moto = Moto.from_factory("Harley-Davidson", "Sportster", 2023)
```

Maintenant que nos Véhicules sont créés, on peut utiliser leurs méthodes.

```
# METHODES DES VEHICULES
vehicules = [voiture, moto]
for vehicule in vehicules:
    class_name = vehicule.__class__.__name__
    print(f"\nNouveau véhicule")
    print(vehicule)
    print(f"la {class_name} roule à {vehicule.vitesse} km/h")
    print(f"la {class_name} accelere")
    vehicule.accelerer()
    print(f"la {class_name} roule à {vehicule.vitesse} km/h")
    print(f"la {class_name} s'est arreter")
    vehicule.arreter()
    print(f"la {class_name} roule à {vehicule.vitesse} km/h")
print(moto.acrobaties())
```

En bonus à la fin, on peut même les détruires avec un affichage personnalisé avec la méthode magique `__del__` dans la classe Vehicule :

```
    ...

    # DESTRUCTEUR
    def __del__(self) -> None:
```

```
        class_name = self.__class__.__name__
        print(f"Objet {class_name} est détruit.")
```

Ce qui nous permettra de réaliser ceci.

```
print("\nLibération de l'espace mémoire")
del voiture # Objet Voiture est détruit.
del moto    # Objet Moto est détruit.
```