Yantra Technologies

www.yantra-technologies.com

Programmation Orientée Objet
Introduction

Facile

Normal

Difficile

Professionnel

Expert

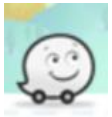
https://wiki.waze.com/wiki/Your_Rank_and_Points

david.palermo@yantra-technologies.com

Sommaire

1 - Présentation

1.1 - Les styles de programmation



- **Fonctionnel ou Applicatif** : évaluation d 'expressions comme une formule, et d'utiliser le résultat pour autre chose *Lisp, Caml*, *APL* (récursivité)
- **Procédural ou Impératif** : exécution d 'instructions étape par étape *Fortran, C, Pascal, Cobol* (itératif)
- **Logique :** répondre à une question par des recherches sur un ensemble, en utilisant des axiomes, des demandes et des règles de déduction *Prolog*
- **Objet** : *Simula, Smalltalk, C++, Java*, *ADA 95*
  - ensemble de composants autonomes (objets) qui disposent de moyens d 'interaction,
  - utilisation de classes,
  - échange de message.

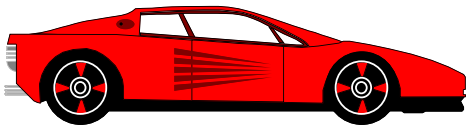
1.1 - L'approche Objet et Langage UML

L'Approche Objet est une **démarche** qui consiste à utiliser **l'objet** pour **modéliser** le monde réel.

- Une démarche => Une méthodologie pour décrire comment s'organiser le système

- Une modélisation => Une Notation **UML**

1.2 - Qu'est-ce qu' un Objet ? (1)



propriétés

Ferrari
rouge
En_stationnement

comportements

rouler
se_garer

identité

ma_ferrari
305 XV 13

1.1,1 - Méthodes Objets

En 1994, plus de 50 méthodes OO

- Fusion, Shlaer-Mellor, ROOM, Classe-Relation, Wirfs-Brock, Coad-Yourdon, MOSES, Syntropy, BOOM, OOSD, OSA, BON, Catalysis,COMMA, HOOD, Ooram, DOORS...

Les méta modèles se ressemblent de plus en plus

Les notations graphiques sont toutes différentes

L'industrie a besoin de standards

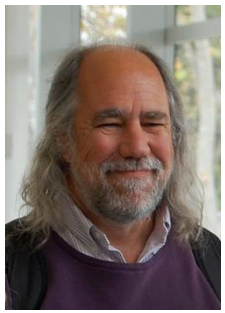
1.1.2 - Unification des méthodes

La pratique des méthodes a permis de faire le tri entre les différents concepts

Jim Rumbaugh, Grady Booch (1993) et plus tard ***Ivar Jacobson*** (1994) décident d'unifier leurs travaux:

- Methode OMT(Object Modeling Technique)
- Methode Booch
- Methode OOSE (Object Oriented Software Engineering)

1.1.3 - Les créateurs de la notation UML



Grady Booch

Méthode de Grady Booch

La méthode proposée par G. Booch est une méthode de conception, définie à l'origine pour une programmation Ada, puis généralisée à d'autres langages. Sans préciser un ordre strict dans l'enchaînement des opérations



James Rumbaugh

Méthode OMT

La méthode OMT (Object Modeling Technique) permet de couvrir l'ensemble des processus d'analyse et de conception en utilisant le même formalisme. L'analyse repose sur les trois points de vue: statique, dynamique, fonctionnel, donnant lieu à trois sous-modèles.



Ivar Jacobson

Méthode OOSE

Object Oriented Software Engineering (OOSE) est un langage de modélisation objet crée par Ivar Jacobson. OOSE est une méthode pour l'analyse initiale des usages de logiciels, basée sur les « cas d'utilisation » et le cycle de vie des logiciels.

1.2 - Qu'est-ce qu 'un Objet ? (2)

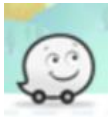
un **objet** représente un concept du monde réel possédant un **état** auquel peuvent être associés des **propriétés** et des **comportements**.

- L 'état d'un objet comprend toutes les propriétés d'un objet (habituellement statiques) plus les valeurs courantes de celles-ci (habituellement dynamiques).
- Une **propriété** d'un objet est une caractéristique inhérente et distincte qui contribue à l 'unicité de l 'objet.
- Le **comportement** d'un objet c'est comment l'objet agit et réagit en termes de changement d'état et de passage de message.

UML : Unified Modeling Language

- **UML permet d'exprimer et d'élaborer des modèles objet, indépendamment de tout langage de programmation.** Il a été pensé pour servir de support à une analyse basée sur les concepts objet.
- UML est un **langage formel**, défini par un **métamodèle**.
- Le métamodèle d'UML décrit de manière très précise tous les éléments de modélisation et la sémantique de ces éléments (leur définition et le sens de leur utilisation).
  UML normalise les concepts objet.
- UML est avant tout **un support de communication performant**, qui facilite la représentation et la compréhension de solutions objet

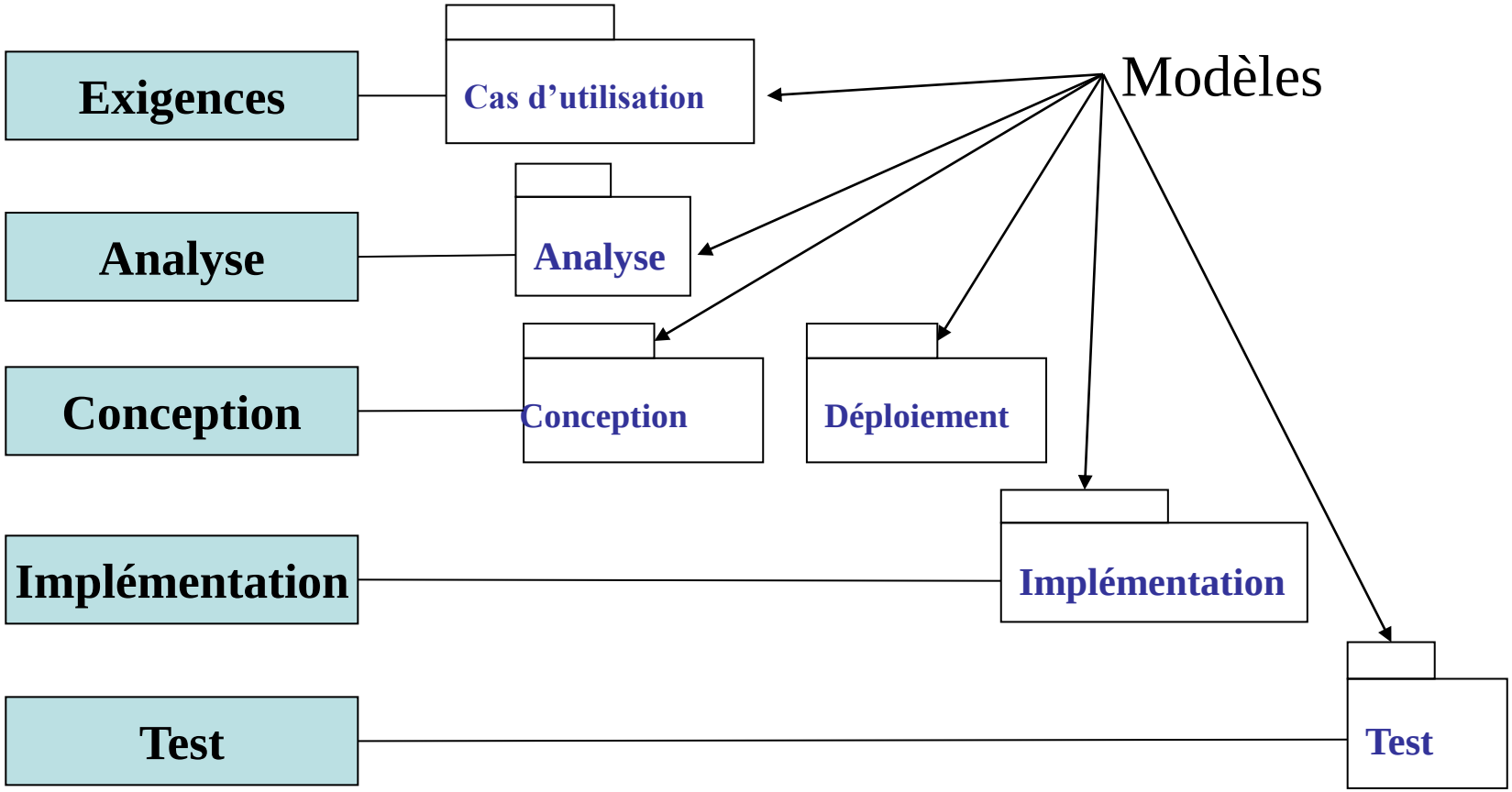
1.5 - UML



- Les points forts d'UML
  - UML est un langage formel et normalisé
  - UML est un support de communication performant

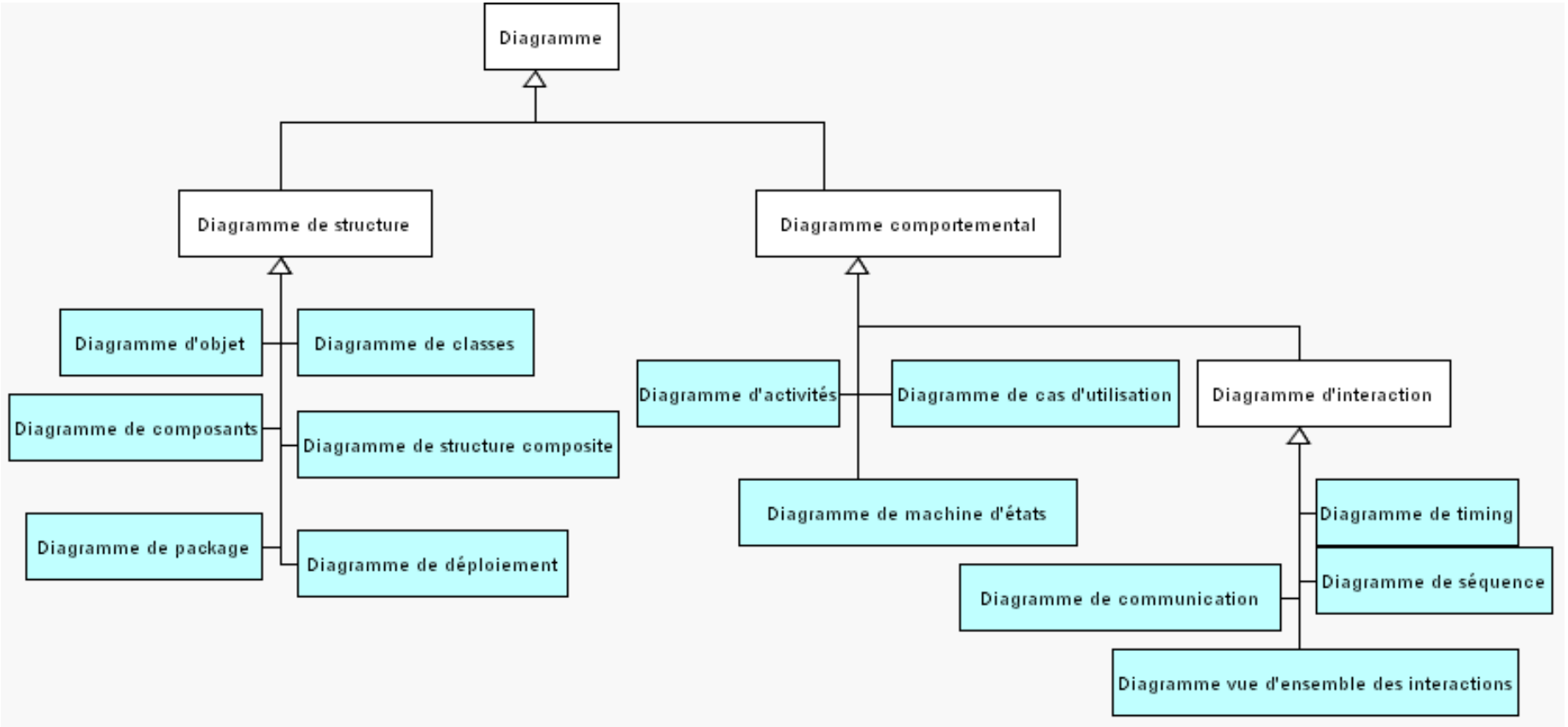
- Les points faibles d'UML
  - La mise en pratique d'UML nécessite un apprentissage et passe par une période d'adaptation.
  - Le processus (non couvert par UML) est une autre clé de la réussite d'un projet.

Modélisation UML



1.7 - Les diagrammes UML 2.0

Modélisation UML 2.0



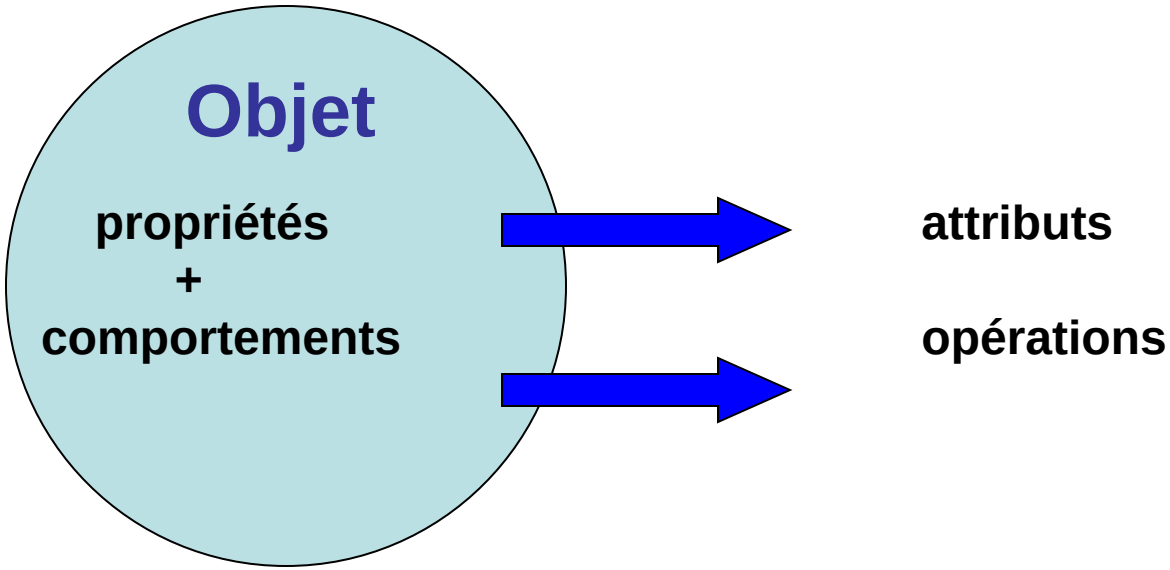
1.8 - L'approche orientée objet

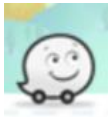
L'approche orientée objet considère le logiciel comme une collection d'objets dissociés définis par des **propriétés**. (propriété : **attribut** ou une **opération)**

⇨ Un objet comprend à la fois une structure de données et une collection d'opérations (son comportement).

⇨ Un certain nombre de caractéristiques pour qu'une approche soit dite orientée objet il faut : l'identité, la classification, le polymorphisme et l'héritage.

Le modèle objet

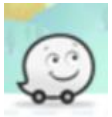




3 concepts pour faire un langage objet :

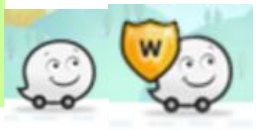
- **Encapsulation :** combiner des données et un comportement dans un emballage unique,
- **Héritage :** chien et chat sont des mammifères, ils héritent du comportement du mammifère.
- **Polymorphisme :** Cercle et rectangle sont des formes géométriques, chacun doit calculer sa surface.

1.10 - Les avantages de la POO



- **Facilite la programmation modulaire :**
  - composants réutilisables,
  - un composant offre des services et en utilise d ’autres,
  - il expose ses services au travers d ’une interface.
- **Facilite l ’abstraction :**
  - elle sépare la définition de son implémentation,
  - elle extrait un modèle commun à plusieurs composants,
  - le modèle commun est partagé par le mécanisme d ’héritage.
- **Facilite la spécialisation :**
  - elle traite des cas particuliers,
  - le mécanisme de dérivation rend les cas particuliers transparents.

2 - Introduction



- Présentation de la notion d ’Objet
- Le principe de l’encapsulation
- La Classe
- L’Héritage
- Agrégation & Composition
- Polymorphisme
- Les classes abstraites
- Relations d’association

2 - Introduction -> Présentation de la notion d'Objet

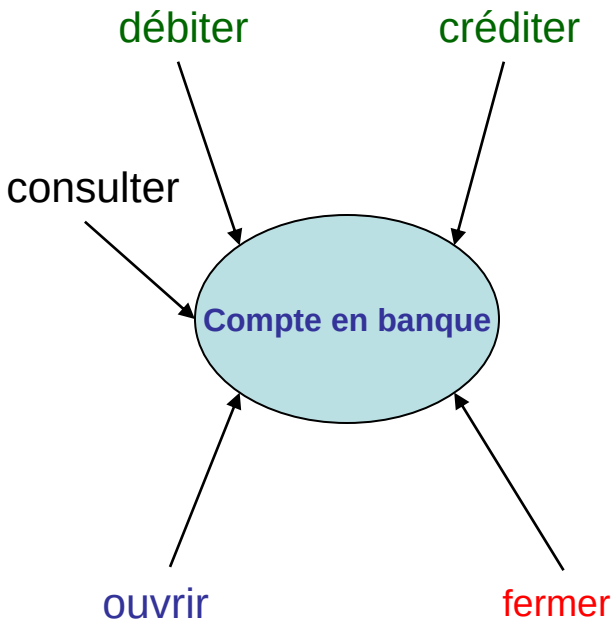
Un Objet peut :

- changer d 'état,
- se comporter de façon discernable,
- être manipulé par diverses formes de stimuli,
- être en relation avec d 'autres objets.

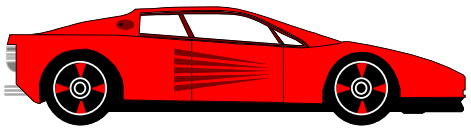
Chaque objet a sa propre **identité** et donc une existence indépendante des autres.

Types d'opération sur le comportement

- **Modificateur** : une opération qui altère l 'état d 'un objet,
- **Sélecteur** : une opération qui accède à l 'état d 'un objet
- **Itérateur** : une opération qui permet d 'avoir accès à toutes les parties d 'un objet dans un ordre défini,
- **Constructeur** : un opération qui crée un objet et initialise son état,
- **Destructeur** : une opération qui détruit un objet.



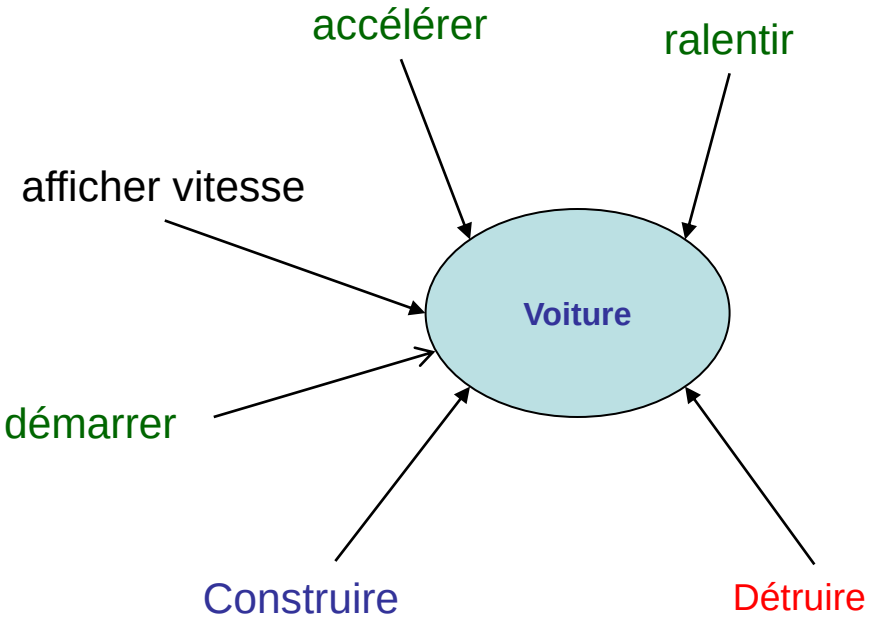
2 - Introduction -> Présentation de la notion d'Objet

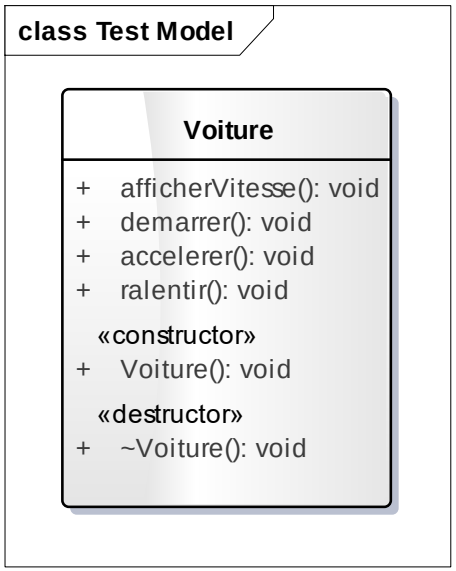


propriétés
Ferrari
rouge
en_stationnement

comportements
rouler
se_garer

identité
ma_ferrari
305 XV 13





2 - Introduction -> Présentation de la notion d'Obje
: Exemple Python

```
def Exemple_01_python():
    print("Exemple 1 : python")
    maFerrari = Voiture("305 XV 13","Ferrari","rouge")
    maFerrari.demarrer();
    maFerrari.afficherVitesse();
    maFerrari.accelerer();
    maFerrari.afficherVitesse();
    maFerrari.ralentir();
    maFerrari.afficherVitesse();
    maFerrari.arreter();
    maFerrari.afficherVitesse();
    del maFerrari;

Exemple_01_python()
```

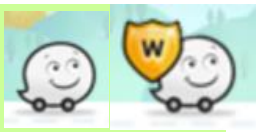
L' Encapsulation

C 'est le processus qui consiste à cacher tous les détails d 'un objet. L'objet peut être vu :

- de **l'intérieur** pour le concepteur : détail de la structure et du comportement, données et méthodes privées,
- de **l'extérieur** pour l 'utilisateur : interface « publique » qui décrite l 'utilisation de l 'objet.

Permet de rendre indépendante la spécification de l 'objet et son implémentation (*la vue externe doit être la plus indépendante possible de la vue interne*).

2 - Introduction -> Notion de classe



On appelle **classe** la structure d'un objet, c'est-à-dire la déclaration de l'ensemble des entités qui composeront un objet. Un objet est donc « issu » d'une classe, c'est le produit qui sort d'un moule.

On dit qu'un objet est une **instanciation** d'une classe, c'est la raison pour laquelle on pourra parler indifféremment d'**objet** ou d'**instance**

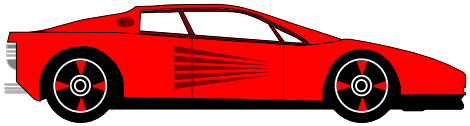
Une classe est composée de deux parties :

- **Les attributs** (parfois appelés *données membres*) : il s'agit des données représentant l'état de l'objet
- **Les méthodes** (parfois appelées *fonctions membres*): il s'agit des opérations applicables aux objets

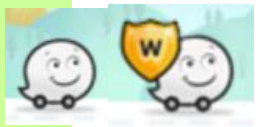
Exemple : Si on définit la classe *Voiture*, les objets *Peugeot 406*, *Renault 18* seront des instanciations de cette classe.

Une classe

C ’est un ensemble d’objets ayant les mêmes propriétés et un comportement commun.



2 - Introduction -> La Classe



- Une **classe** est une construction du langage de programmation :
  - décrit les propriétés communes à des objets

    Class Point { }
  - un objet est une *instance* de classe

- Un **objet** est une entité en mémoire :
  - il a un état, un comportement, une identité.

    Point* a = new Point(3,5); (C++)

    Point a = new Point(3,5); (java)

Un objet sans classe n'existe pas

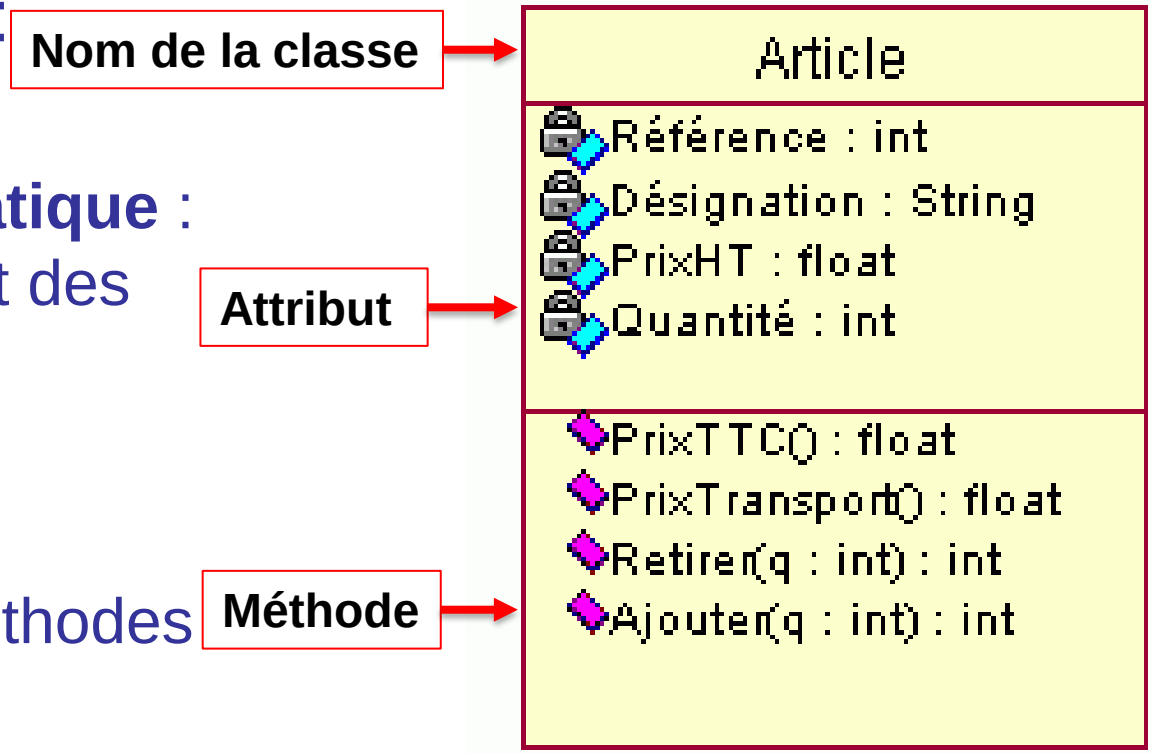
2 - Introduction -> La Classe

La classe possède :

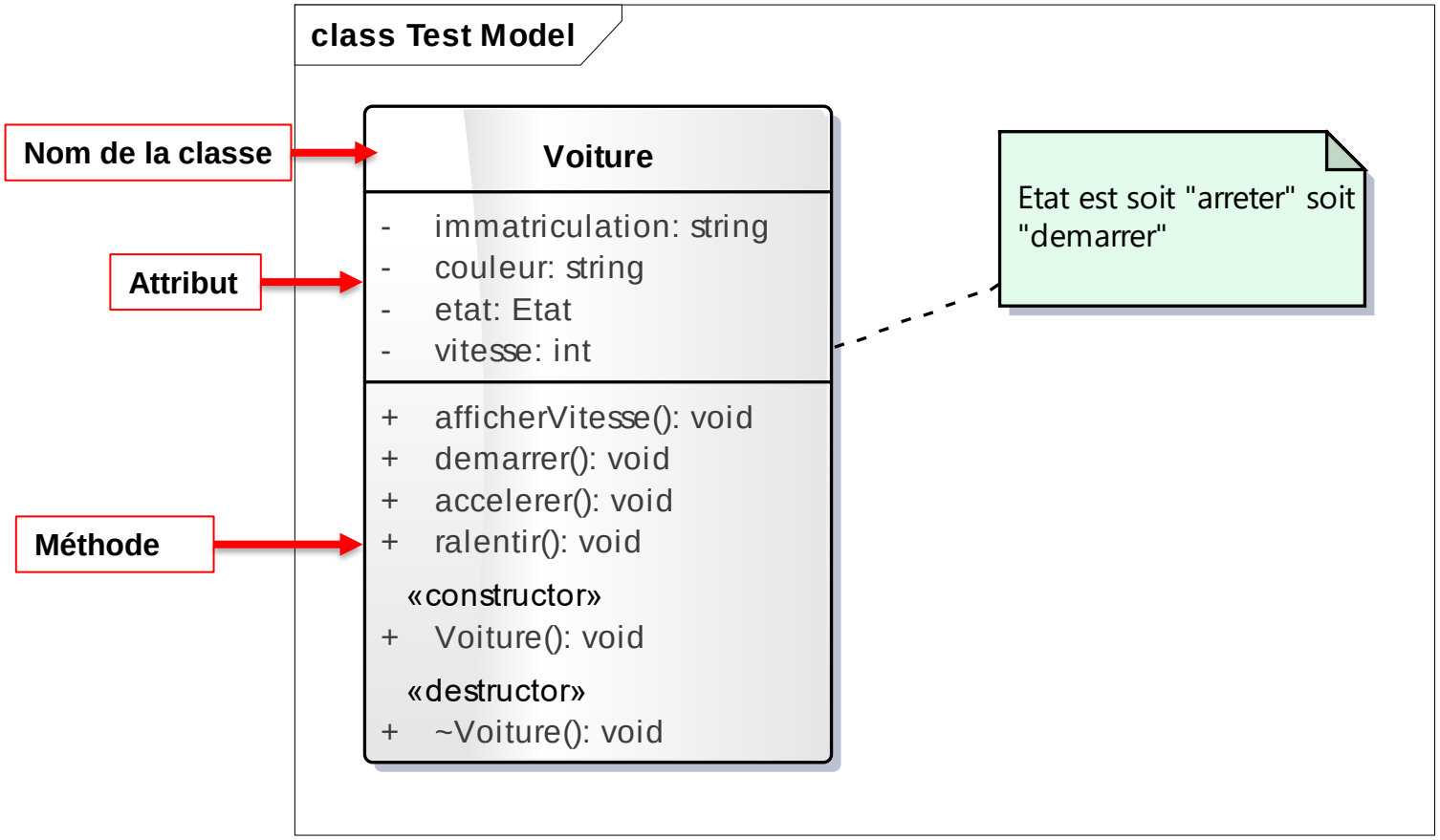
- un nom unique,
- une composante **statique** : les données qui sont des champs (attributs)

  ⇒ Etat
- une composante **dynamique** : les méthodes (opérations)

  ⇒ Comportement



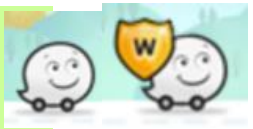
2 - Introduction -> La Classe



2 - Introduction -> La Classe : Exemple Python

```python
class Voiture:
    def __init__(self,immatriculation,typevoiture,couleur):
        print("Voiture : construction  ")
        self.immatriculation=immatriculation
        self.type=typevoiture
        self.couleur=couleur
        self.etat="arreter"
        self.vitesse=0
        pass
    def demarrer(self):
        print("Voiture : construction  ")
        self.etat="demarrer"
        pass
    def afficherVitesse(self):
        print("Vitesse ",self.vitesse)
        pass
    def accelerer(self):
        print("Voiture : accelerer  ")
        if ( self.etat == "demarrer"):
            self.vitesse+=1
        pass
    def ralentir(self):
        print("Voiture : ralentir  ")
        if ( self.etat == "demarrer" and self.vitesse > 0):
            self.vitesse-=1
        pass
    def arreter(self):
        print("Voiture : arreter  ")
        self.etat="arreter"
        self.vitesse=0
        pass
    def __del__(self):
        print("Voiture : destruction ")
    pass
```

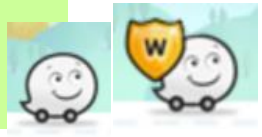
2 - Introduction -> La Classe : Exemple Python



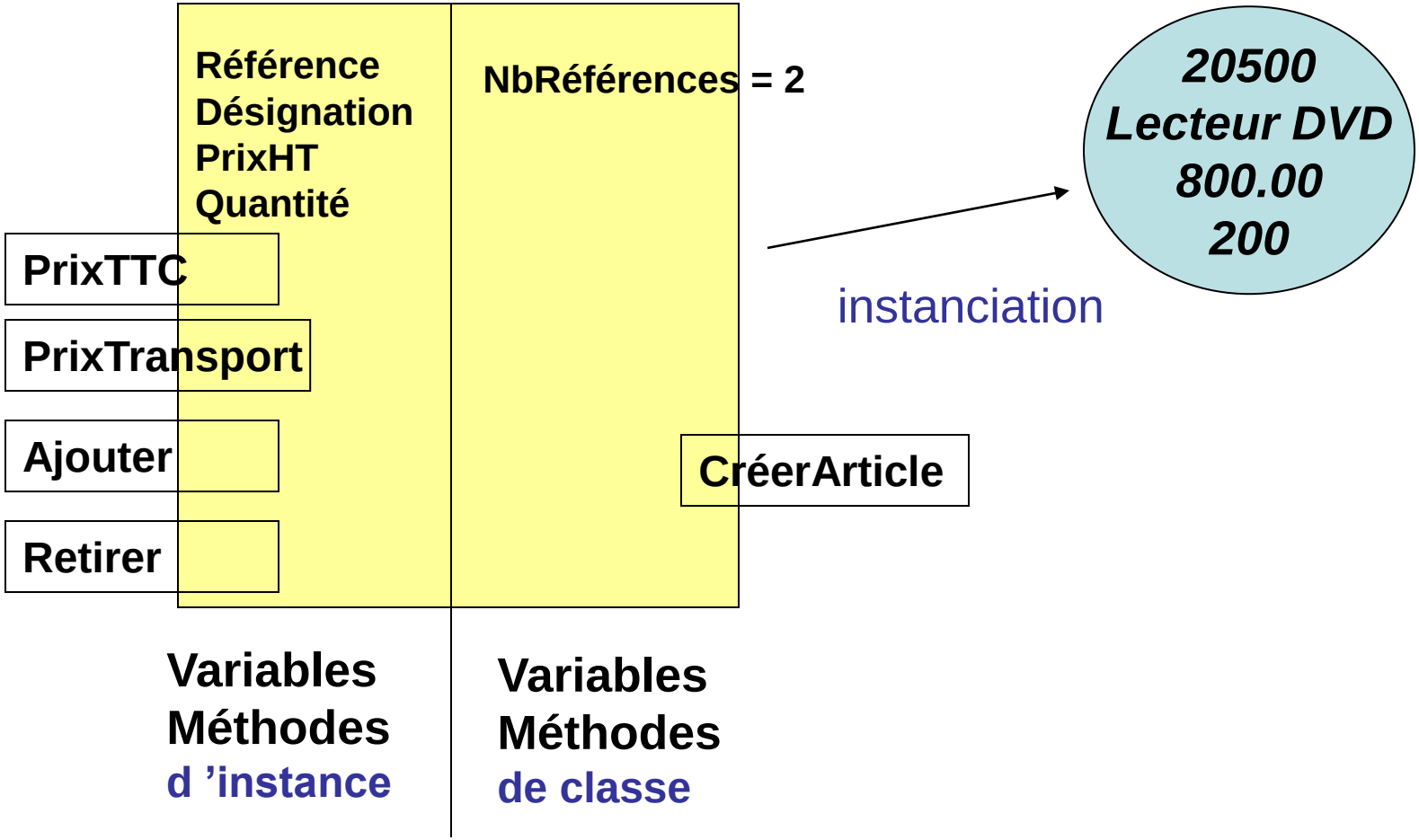
```
def Exemple_01_python():
    print("Exemple 1 : python")
    maFerrari = Voiture("305 XV 13","Ferrari","rouge")
    maFerrari.demarrer();
    maFerrari.afficherVitesse();
    maFerrari.accelerer();
    maFerrari.afficherVitesse();
    maFerrari.ralentir();
    maFerrari.afficherVitesse();
    maFerrari.arreter();
    maFerrari.afficherVitesse();
    del maFerrari;

Exemple_01_python()
```

2 - Introduction -> La classe : Cycle de Vie et mort des objets

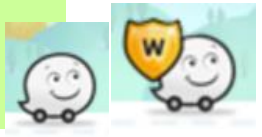
1. Naissance d'un objet *(constructeur)*
   - Allouer de la mémoire
   - Initialiser cette mémoire
2. Vie d'un objet
   - Utilisation des méthodes en modification, sélection et itération
3. Mort d'un objet *(destructeur)*
   - Libérer la mémoire allouée dynamiquement
   - Rendre les ressources systèmes
   - autres……

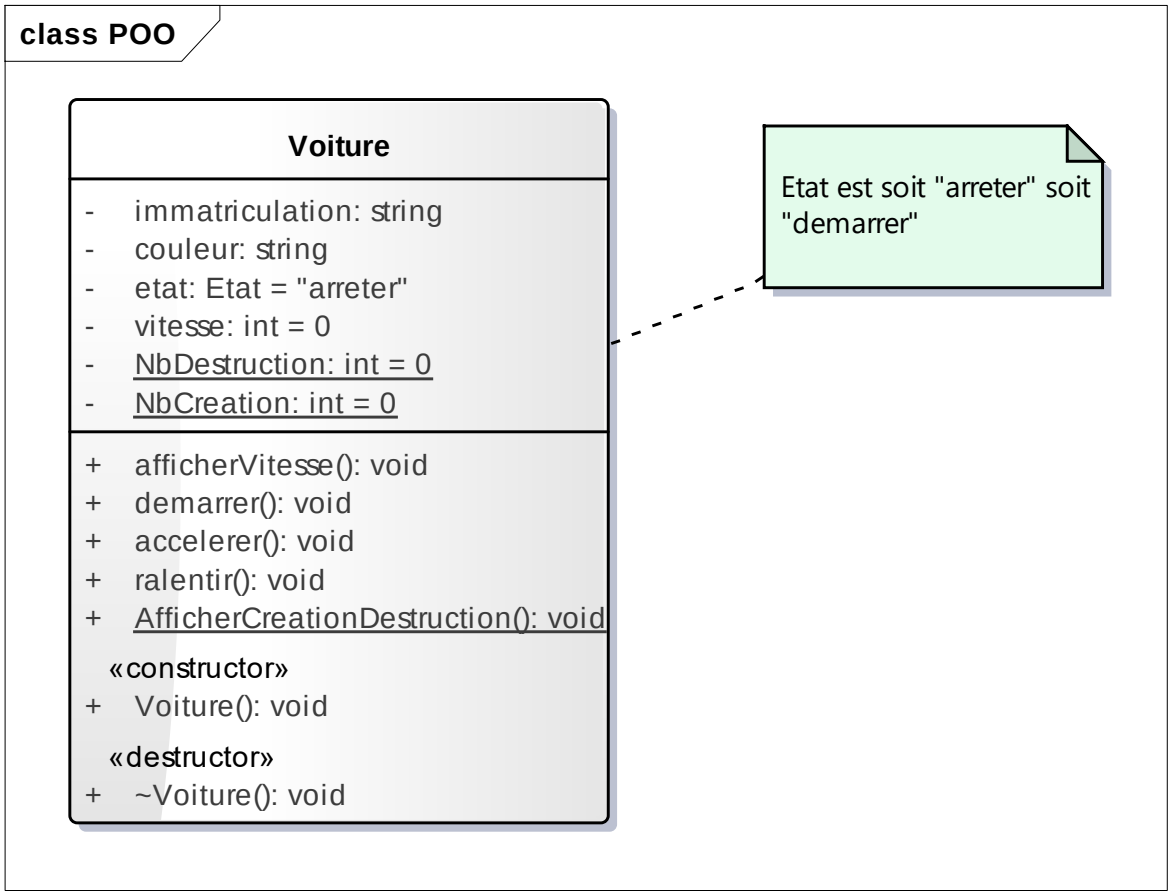


Méthodes, variables d'instance et de classe



2.4- 2 - Introduction -> La Classe





2 - Introduction -> La Classe : Exemple Python

```
class Voiture:
    NbCreation=0
    NbDestruction=0
    @classmethod
    def AfficherCreationDestruction(cls):
        print("Nombre de voiture creee    : ",cls.NbCreation)
        print("Nombre de voiture detruite : ",cls.NbDestruction)
    pass
    vitesse=0
    etat="arreter"
    def __init__(self,immatriculation,typevoiture,couleur):
        print("Voiture : construction  ")
        self.immatriculation=immatriculation
        self.type=typevoiture
        self.couleur=couleur
        self.__class__.NbCreation+=1
        pass
    def demarrer(self):
        print("Voiture : construction  ")
        self.etat="demarrer"
        pass
    def afficherVitesse(self):
        print("Vitesse ",self.vitesse)
        pass
    def accelerer(self):
        print("Voiture : accelerer  ")
        if ( self.etat == "demarrer"):
            self.vitesse+=1
        pass
```

2 - Introduction -> La Classe : Exemple Python

```
    def ralentir(self):
        print("Voiture : ralentir  ")
        if ( self.etat == "demarrer" and self.vitesse > 0):
            self.vitesse-=1
        pass
    def arreter(self):
        print("Voiture : arreter  ")
        self.etat="arreter"
        self.vitesse=0
        pass
    def __del__(self):
        print("Voiture : destruction ")
        self.__class__.NbDestruction+=1


def Exemple_01_python():
    print("Exemple 1 : python")
    maFerrari = Voiture("305 XV 13","Ferrari","rouge")
    maFerrari.demarrer();
    maFerrari.afficherVitesse();
    maFerrari.accelerer();
    maFerrari.afficherVitesse();
    maFerrari.ralentir();
    maFerrari.afficherVitesse();
    maFerrari.arreter();
    maFerrari.afficherVitesse();
    del maFerrari;
    Voiture.AfficherCreationDestruction()
    maFerrari2 = Voiture("305 XV 13","Ferrari","rouge")
    Voiture.AfficherCreationDestruction()


Exemple_01_python()
```

2 - Introduction -> La Classe : encapsulation des attributs & méthodes

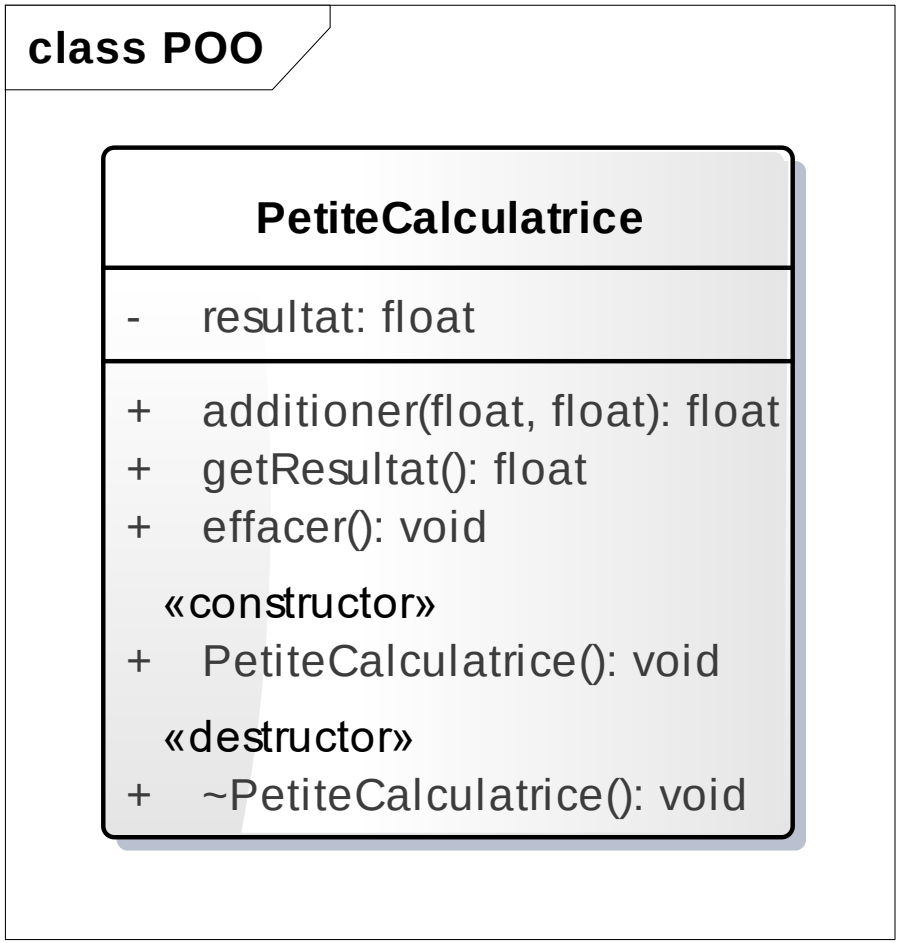
En POO les attributs et méthodes d'une classe peuvent être visible depuis les instances de toutes les classes d'une application.

En POO il faudrait que :

- Les attributs ne soient lus ou modifiés que par l'intermédiaire de méthodes prenant en charge les vérifications et effets de bord éventuels.

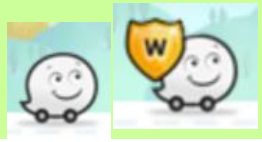
- les méthodes "utilitaires" ne soient pas visibles, seules les fonctionnalités de l'objet, destinées à être utilisées par d'autres objets soient visibles.

2 - Introduction -> La Classe : encapsulation des attributs & méthodes



2 - Introduction -> La Classe : encapsulation des attributs & méthodes - Exemple : Python

```python
class PetiteCalculatrice:
    resultat=0.0
    def additionner(self,nb1,nb2):
        self.resultat=nb1+nb2
        return self.resultat
    def effacer(self):
        self.resultat=0.0
        pass
    def getResultat(self):
        return self.resultat

cal = PetiteCalculatrice()
print(cal.additionner(5.,2.))
```



La relation d'association

- exprime une dépendance sémantique entre des classes
- une association est bidirectionnelle
- une association a une cardinalité
  - 0 ou 1
  - un pour un
  - un pour n
  - n pour n

2 - Introduction -> Les Relations d'association

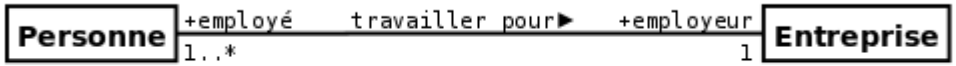


Figure 3.5 : Exemple d'association binaire.

Figure 3.7 : Navigabilité.

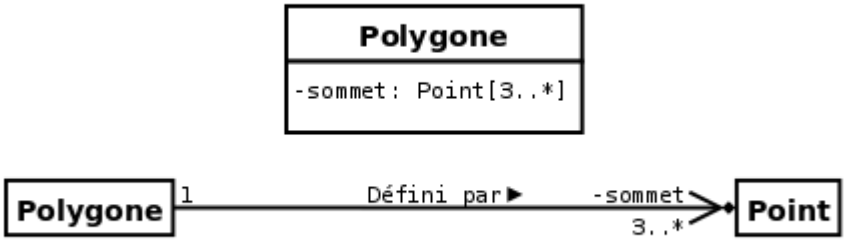
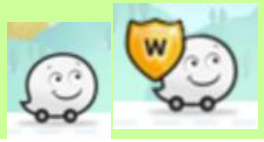


Figure 3.9 : Deux modélisations équivalentes.

https://laurent-audibert.developpez.com/Cours-UML/?page=diagramme-classes#L3-3-3

2 - Introduction -> Les Relations d'association



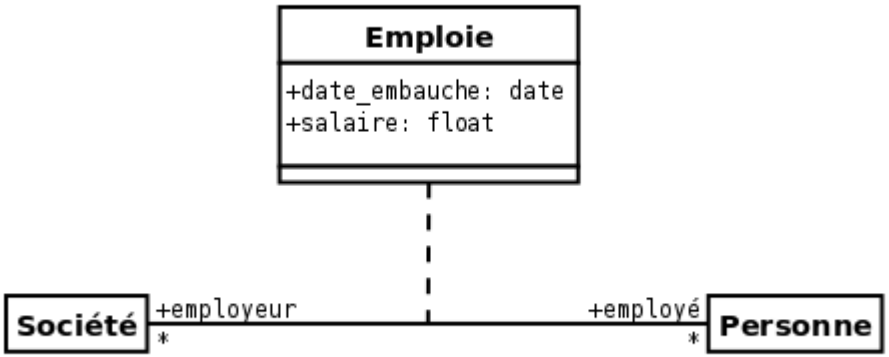


Figure 3.11 : Exemple de classe-association.

https://laurent-audibert.developpez.com/Cours-UML/?page=diagramme-classes#L3-3-3

2 - Introduction -> Les Relations d'association

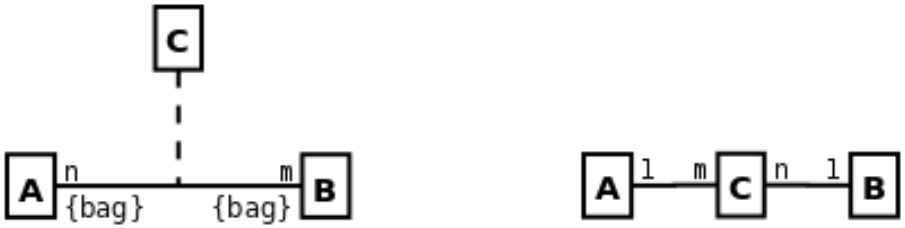
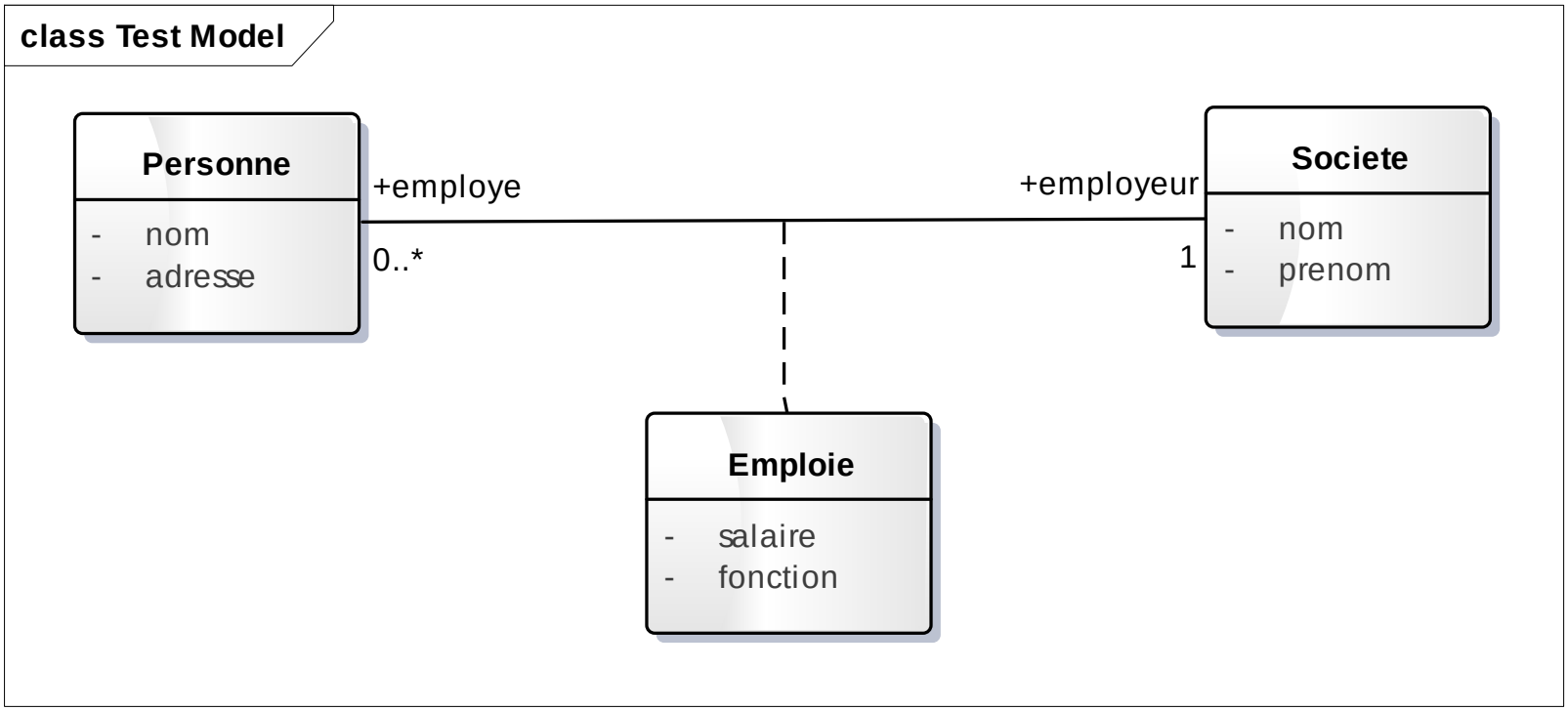


Figure 3.14 : Deux modélisations modélisant la même information.

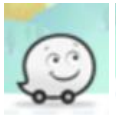
Figure 3.15 : Deux modélisations modélisant la même information.

https://laurent-audibert.developpez.com/Cours-UML/?page=diagramme-classes#L3-3-3

2 - Introduction -> Les Relations d'association



2 - Introduction -> Les Relations d'association : Exemple Python

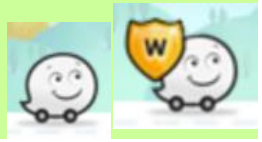


```
class Personne:
    nom=""
    adresse=""
    def __init__(self,nom,adresse):
        self.nom =nom
        self.adresse=adresse
    def getAdresse(self):
        return self.adresse
    def getNom(self):
        return self.nom

class Societe:
    nom=""
    adresse=""
    def __init__(self,nom,adresse):
        self.nom =nom
        self.adresse=adresse
    def getAdresse(self):
        return self.adresse
    def getNom(self):
        return self.nom
```

2 - Introduction -> Les Relations d'association : Exemple Python

```
class Emploi:
    employeur=None
    lienEmployeEmploiSalaire={}
    def __init__ (self,employeur):
        self.employeur=employeur
    def set(self,cle, per,salaire,emploi ):
        self.lienEmployeEmploiSalaire.update({cle:[per,salaire,emploi]})
    def getPersonne(self, cle) :
        return self.lienEmployeEmploiSalaire[cle][0]
    def getSalaire(self, cle) :
        return self.lienEmployeEmploiSalaire[cle][1]
    def getEmploi(self, cle):
        return self.lienEmployeEmploiSalaire[cle][2]
    def nbEmployer(self) :
        return len(self.lienEmployeEmploiSalaire)
    def getSociete(self) :
        return self.employeur


personne =Personne("david","pertuis")
societe=Societe("YNOV","Aix")
emploi =Emploi(societe)
emploi.set("x0124589",personne,"variable","formateur");
print(emploi.getPersonne("x0124589").getNom())
print(emploi.getSalaire("x0124589"))
print(emploi.getEmploi("x0124589"))
print(emploi.nbEmployer())
print(emploi.getSociete().getNom(), " ", emploi.nbEmployer()," employer")
```

2 - Introduction -> L'Agrégation & Composition



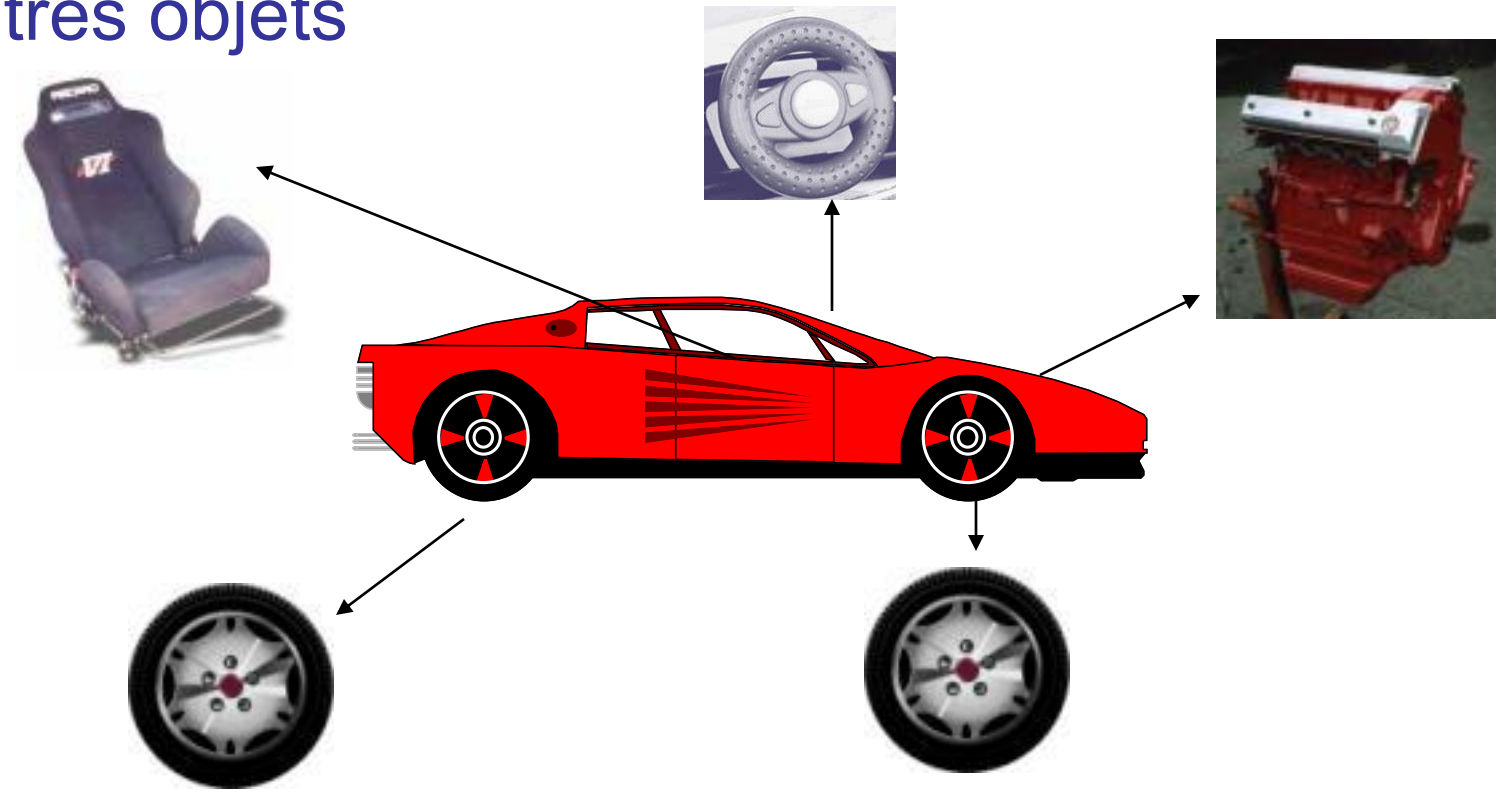
La composition peut être vue comme une relation **"fait partie de"** ("part of"), c'est à dire que si un objet B fait partie d'un objet A alors B ne peut pas exister sans A. Ainsi **si A disparaît alors B également**.

L'agrégation quant à elle est vue comme une relation de type **"a un"** ("has a"), c'est à dire que si un objet A a un objet B alors **B peut vivre sans A**

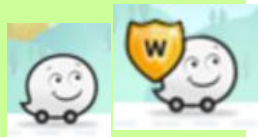
http://www.lechatcode.com/architecture/agregation-et-composition-cest-quoi/

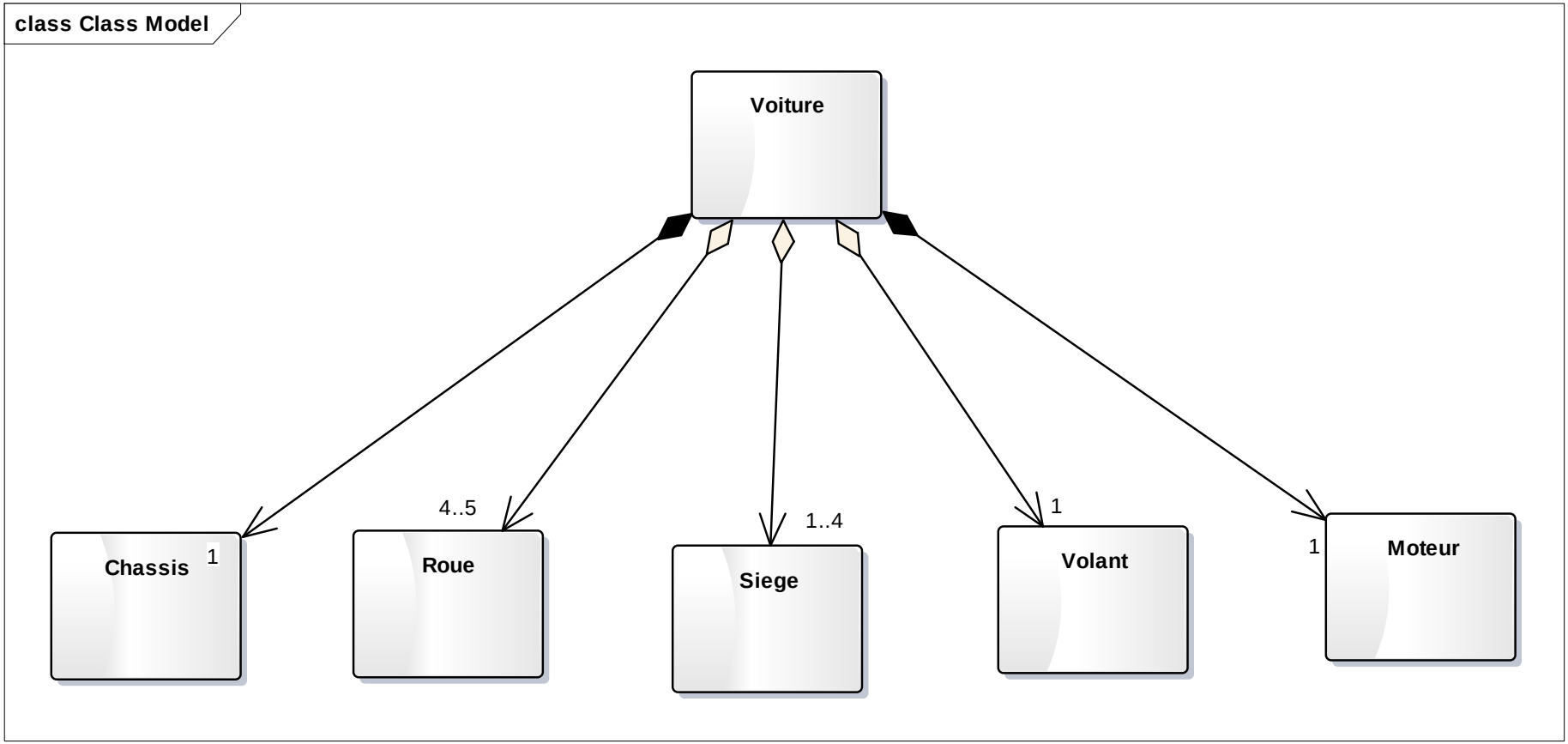
Agrégation

Les objets peuvent contenir des références à d 'autres objets



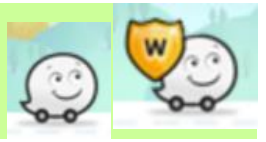
2 - Introduction -> L'Agrégation & Composition : L'Agregation





2 - Introduction -> L'Agrégation & Composition : L'Agrégation : Exemple Python

```
class Chassis:
    pass

class Roue:
    pass

class Siege :
    pass

class Volant :
    pass

class Moteur :
    pass

class Voiture:
    def __init__(self):
        self.chassis = Chassis()
        self.roue = [Roue(),Roue(),Roue(),Roue()]
        self.siege = [Siege(),Siege(),Siege(),Siege()]
        self.volant = Volant()
        self.moteur = Moteur ()
    def __del__(self):
        del self.chassis
        del self.moteur
```



L 'Héritage

Par héritage, une classe dérivée possède les attributs et les méthodes de la superclasse.

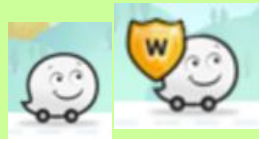
⇒ La classe dérivée possède les attributs et méthodes de la classe de base,

⇒ la classe dérivée peut en ajouter ou en masquer,

⇒ facilite la programmation par raffinement,

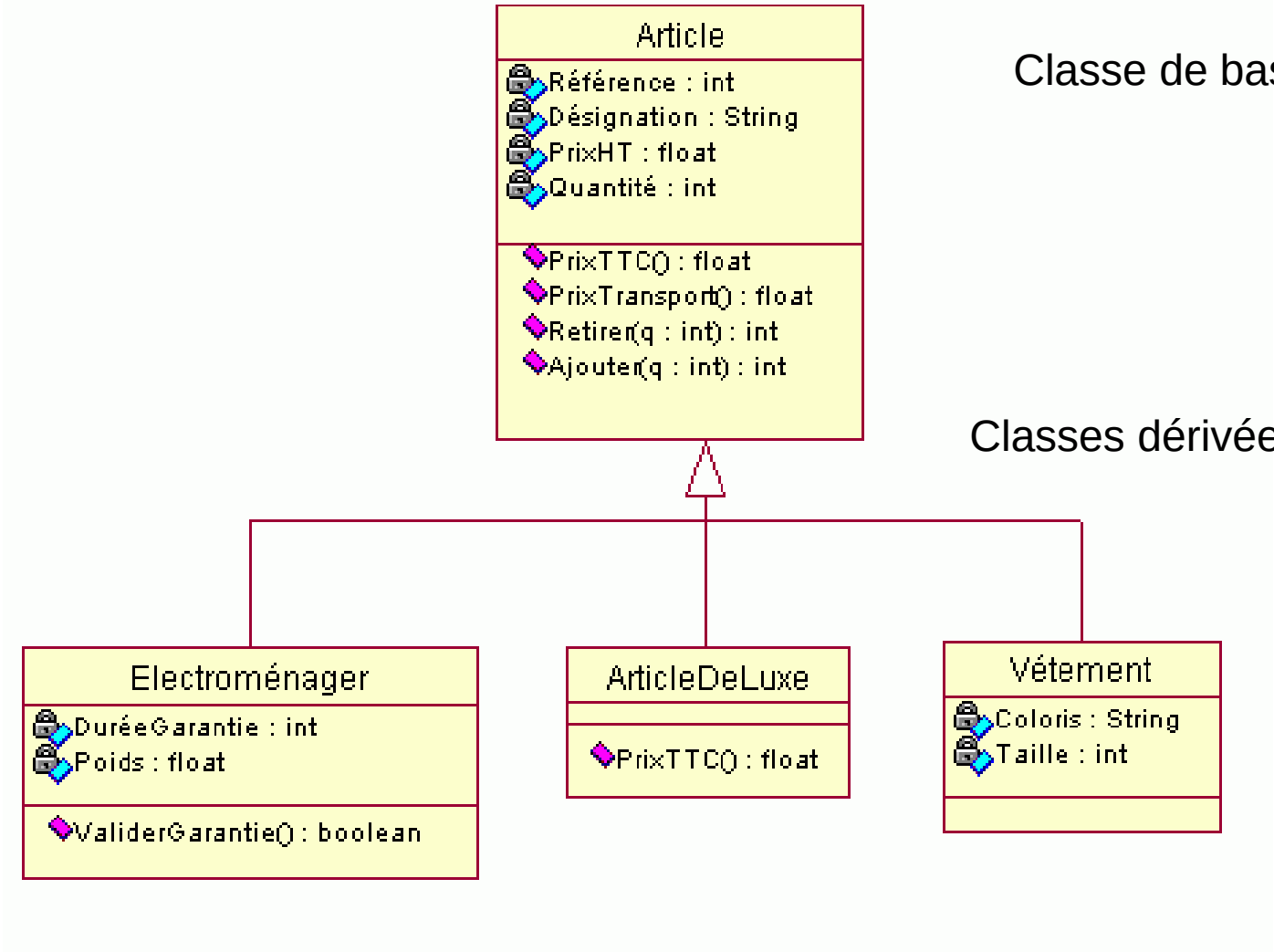
⇒ facilite la prise en compte de la spécialisation.

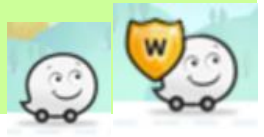
2 - Introduction -> L 'Héritage



GENERALISATION

SPECIALISATION





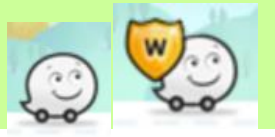
Le Polymorphisme

C ’est un concept selon lequel le même nom peut désigner des méthodes différentes.

La méthode désignée dépend de l’objet auquel on s’adresse.

- Le polymorphisme **Statique:** l’objet est connu à la compilation
- Le polymorphisme **Dynamique:** l’objet n’est connu qu’ultérieurement. L’identité ne sera qualifiée qu’au moment de l’exécution.

2 - Introduction -> Le Polymorphisme



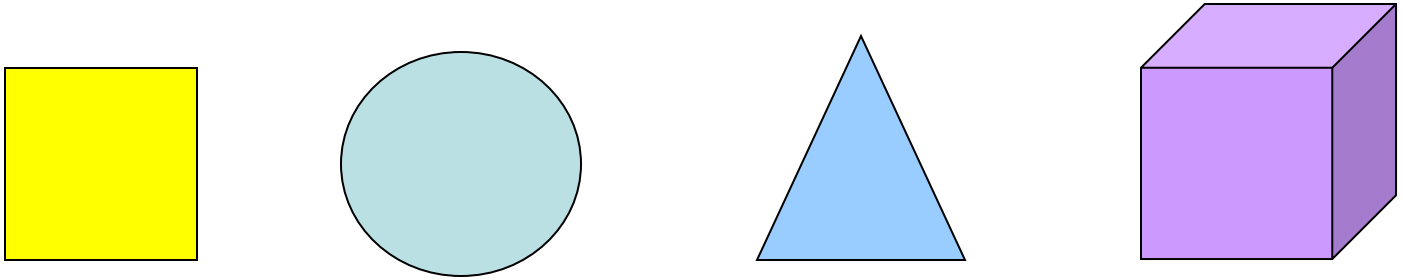
```
class Forme
{ afficher(); }
class Rectangle
{afficher();}
class Cercle
{afficher();}
```

STATIQUE

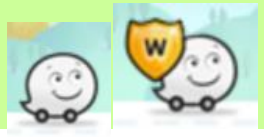
un Rectangle.afficher();
un Cercle.afficher();

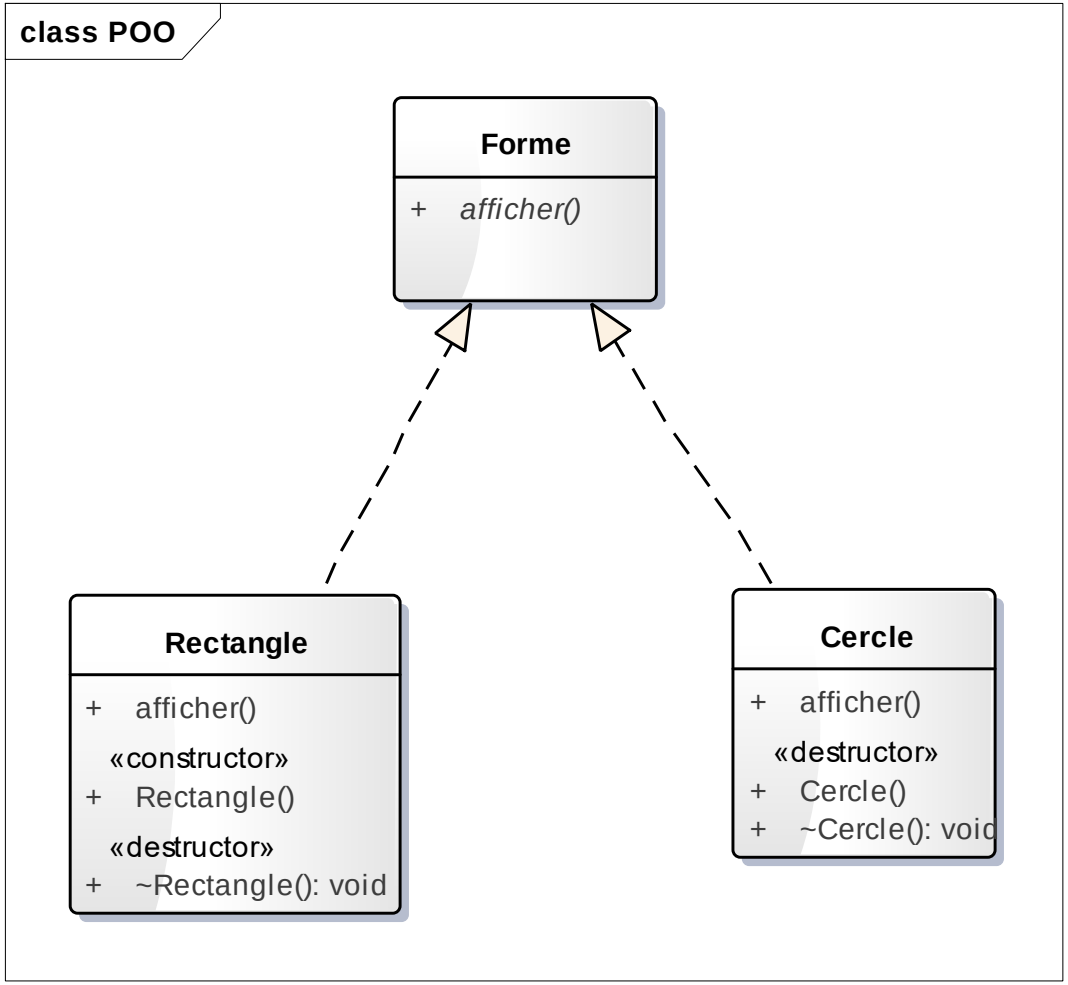
DYNAMIQUE

forme.afficher();



2 - Introduction -> Le Polymorphisme

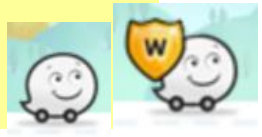




2 - Introduction -> Le Polymorphisme : Exemple Python

```
class Forme :
    def afficher(self):
        pass

class Rectangle(Forme):
    def afficher(self):
        print("je suis un rectangle")

class Cercle(Forme):
    def afficher(self):
        print("je suis un cercle")

def afficherForme(f):
    f.afficher()

cer = Cercle()
rec =Rectangle()
cer.afficher()
rec.afficher()
print("\n")
afficherForme(rec)
afficherForme(cer)
```

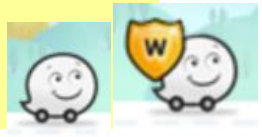
```
je suis un cercle
je suis un rectangle


je suis un rectangle
je suis un cercle
```

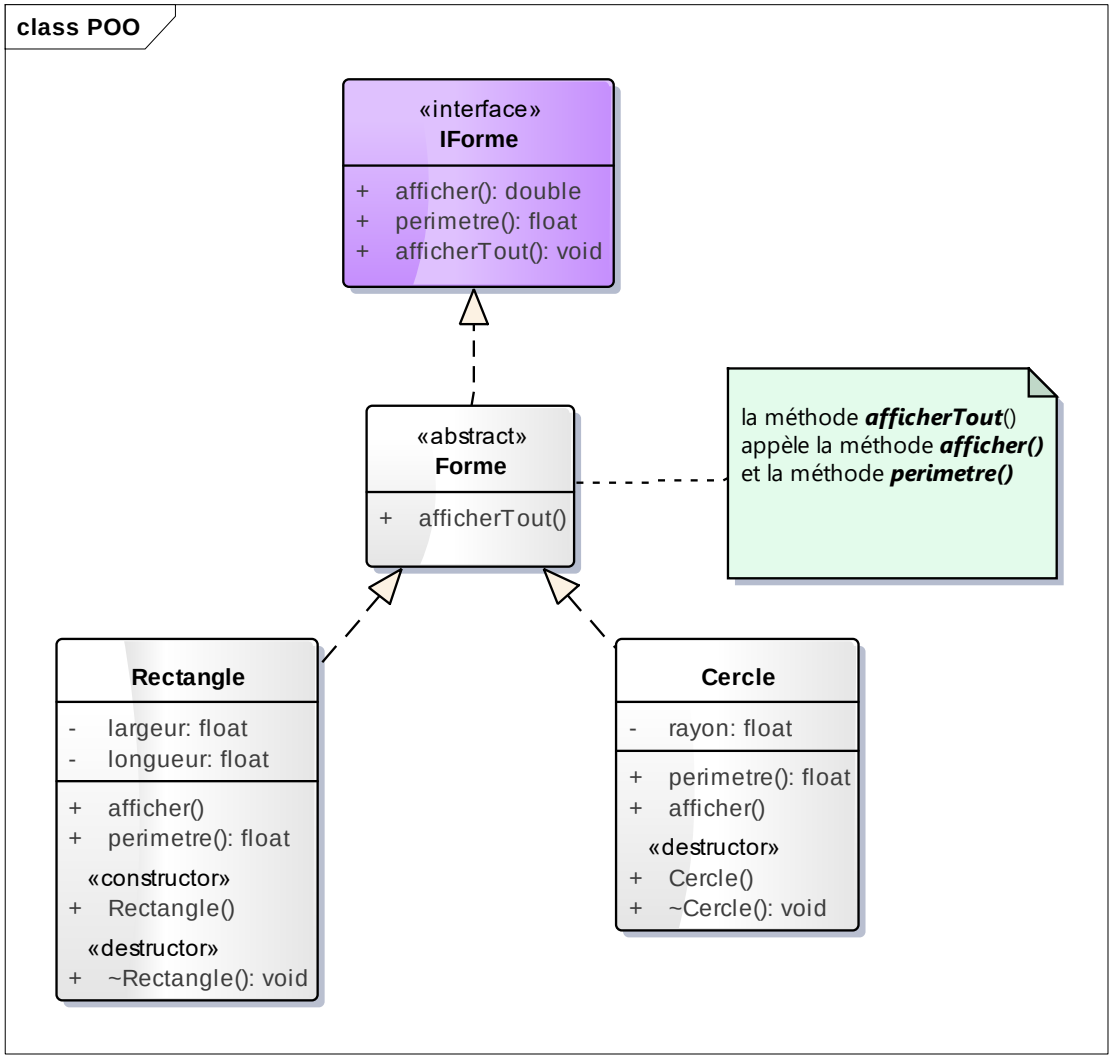


Les Classes Abstraites

- Elles représentent des concepts : *Mammifère*
- Elles peuvent contenir des méthodes abstraites :
  - méthodes sans implémentation (corps),
  - la mise en œuvre pour chaque sous-classe peut être différente (*calculerSurface*)
  - polymorphisme

2 - Introduction -> Les Classes Abstraites



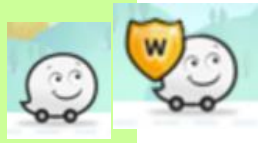


2 - Introduction -> Les Classes Abstraites : Exemple Python

```
class IForme:
    def afficher(self) :
        pass
    def perimetre(self):
        pass
    def afficherTout(self):
        pass

class Forme (IForme):
    def afficherTout(self):
        self.afficher();
        print("le perimetre de la forme est : ",self.perimetre())


class Rectangle(Forme):
    def __init__(self,longeur,largeur):
        self.longeur = longeur
        self.largeur=largeur
    def afficher(self):
        print("je suis un rectangle")
    def perimetre(self):
        return (self.longeur+self.largeur) * 2

class Cercle(Forme):
    def __init__(self,rayon):
        self.rayon = rayon
    def afficher(self):
        print("je suis un cercle")
    def perimetre(self):
        return 2 * 3.1415 * self.rayon


cer = Cercle(10)
rec =Rectangle(5,2)
cer.afficherTout()
rec.afficherTout()
```

```
je suis un cercle
le perimetre de la forme est :  62.830000000000005
je suis un rectangle
le perimetre de la forme est :  14
```

2 - Introduction -> Interface

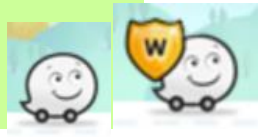


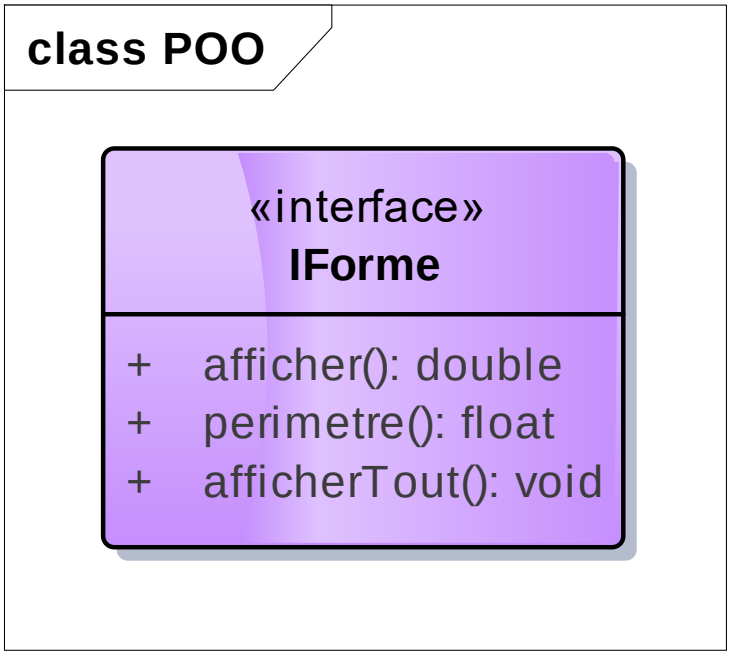
Il s'agit donc de l'ensemble des méthodes accessibles depuis l'extérieur de la classe, par lesquelles on peut modifier l'objet. Pour rappel la différenciation publique/privée ou portée de variable permet :

- d'éviter de manipuler l'objet de façon non voulue, en limitant ses modifications à celles autorisées comme publiques par le concepteur de la classe
- Au concepteur, de modifier l'implémentation interne de ces méthodes de manière transparente.

https://fr.wikipedia.org/wiki/Interface_(programmation_orient%C3%A9e_objet)

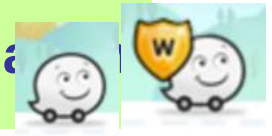
2 - Introduction -> Interface





2 - Introduction -> Interface : Exemple Python

```
class IForme:
    def afficher(self) :
        pass
    def perimetre(self):
        pass
    def afficherTout(self):
        pass
```



Clonage d'un Objet

- Créer un objet *identique* à un autre

Comparaison Objet :

- Comparer 2 objet pour savoir si il sont *identiques*

Assignation Objet

- Copier l'état d'un objet dans un autre afin qu'il soit *identique*

3 - Bibliographies