UML : DIAGRAMME DE CLASSE

Introduction

Ce module a pour objectif la Conception d'un Système d'Information.
Nous allons découvrir au travers de ce module, la modélisation UML (Unified Modeling Language).

UML est un langage unifié de modélisation.
Il permet de décrire sous forme de diagrammes lisible les expressions du besoin orientées métiers.

Il est composé de 14 diagrammes :

- 7 diagrammes de structure (comme le diagramme de classe)
- 7 diagrammes comportementaux (comme le diagramme de cas d'utilisation, diagramme d'activité, diagramme de séquence)

Objectif

Le but est de découvrir le diagramme de classe.

UML : DIAGRAMME DE CLASSE

1. Le Diagramme de Classe

Le Diagramme de Classe est le diagramme le plus utilisé en UML.

Son but est de décrire un système d'un point de vue statique, en mettant l'accent sur la relation entre les classes du système, sans s'occuper des interactions et du cycle de vie des objets.

Il explicite les attributs, méthodes et associations des différents objets constitutifs de notre système.

En cela, Il est central à la modélisation du système, et il est nécessaire d'avoir une idée claire des données qui vont être exploitées par ce dernier.

UML : DIAGRAMME DE CLASSE

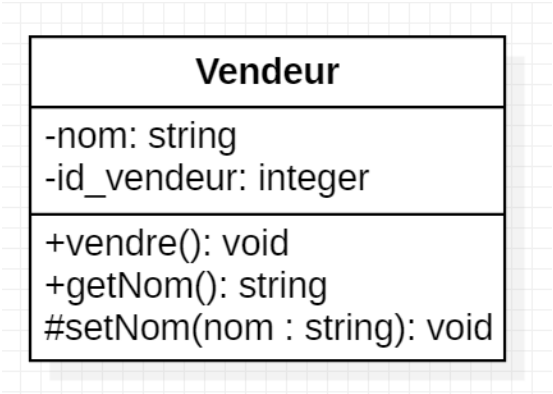
2. Les Classes

Les Classes sont les principaux éléments constituant le Diagramme de Classe.

Une Classe est un plan de construction pour instancier (ou construire) un objet au sein du système. Ce plan a la responsabilité de décrire les données qui caractérisent l'objet, ainsi que les fonctions qui permettent à l'objet de les manipuler.

Cette structure est symbolisée par un rectangle possédant 3 compartiments.

Une Classe doit posséder un nom unique au sein du système (écrit dans le premier compartiments), des attributs (écrits dans le deuxième compartiments), des méthodes (écrites dans le troisième compartiment), et une visibilité.



1. Attributs :

Les attributs sont les variables de la Classe qui représentent les données qu'une instance de la Classe peut contenir.

Un attribut possède un nom, un type, et une visibilité.

2. Méthodes :

Les méthodes sont les fonctions permettant à une Classe de fournir des services et fonctionnalités.

Elles possèdent un nom, d'éventuel paramètres typés, un type de retour, et une visibilité.

Enfin, une méthode possède une signature, constituée du nom de la méthode, de ses paramètres et de son type de retour. La signature permet d'identifier de manière unique la méthode au sein d'une Classe.

UML : DIAGRAMME DE CLASSE

3. L'Encapsulation

Comme il l'a été précisé, les attributs et les méthodes d'une Classe possèdent une visibilité.

C'est ce qu'on appelle l'Encapsulation.

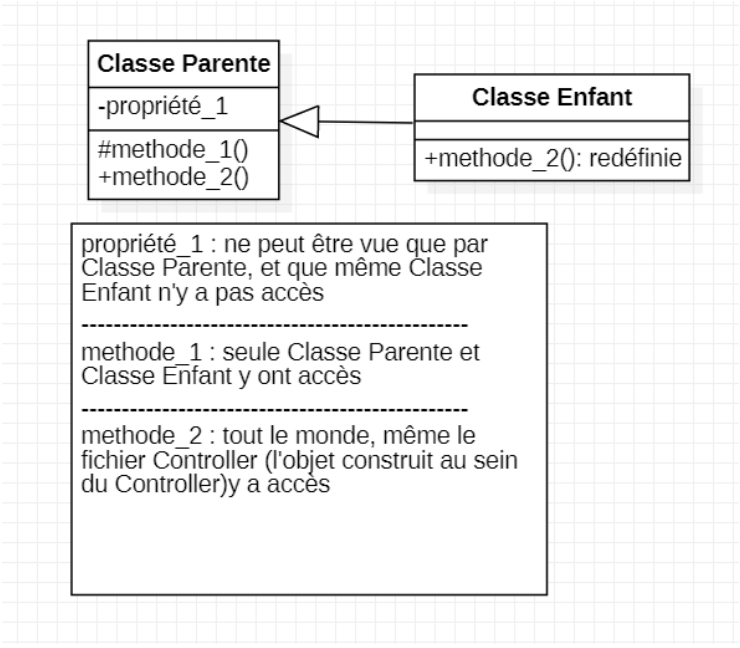
Le rôle de l'Encapsulation est de définir quel élément du système accès aux attributs et fonctions de la Classe.

Nous pouvons définir les niveaux de visibilité suivant :

- **Public :** symbolisé par un **+** , un attribut ou une méthode publique est visible et peut être utilisé par n'importe quel élément du système.
- **Private :** symbolisé par un **-** , un attribut ou une méthode privée est accessible uniquement au sein de la Classe elle-même.
- **Protected :** symbolisé par un **#** , un attribut ou une méthode protégée est accessible uniquement par la Classe elle-même, ainsi que par ses Classes Enfants.

Dans certains langages, il existe un quatrième niveau de visibilité : **Paquetage**, symbolisé par un **~**. A ce niveau, les éléments encapsulés ne sont accessibles que par les Classes du même paquetage.

Par convention, une bonne pratique est de mettre les Attributs en **private**. On y accédera alors en définissant au sein de la Classe des **Getters** et des **Setters**, des méthodes prévues pour retourner la valeur des attributs et les modifier.

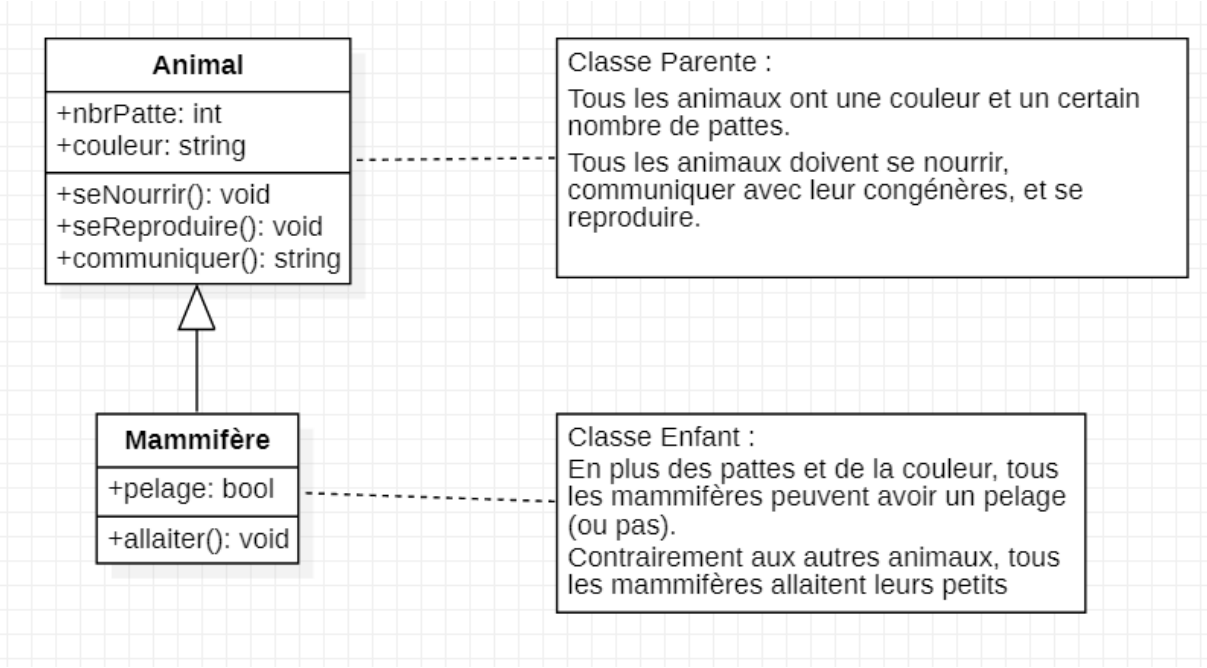


4. L'Héritage : le verbe « être »

L'Héritage est une relation entre deux Classes, définissant une Classe Parente (ou superclasse) et une Classe Enfant (ou sous-classe).

Dans cette relation, la Classe Enfant va récupérer les attributs et méthodes de sa Classe Parente, et par transitivité, des attributs et méthodes de ses « Grands-Parents ».

Le symbole de l'Héritage est une flèche à tête blanche, allant de la Classe Enfant vers la Classe Parent



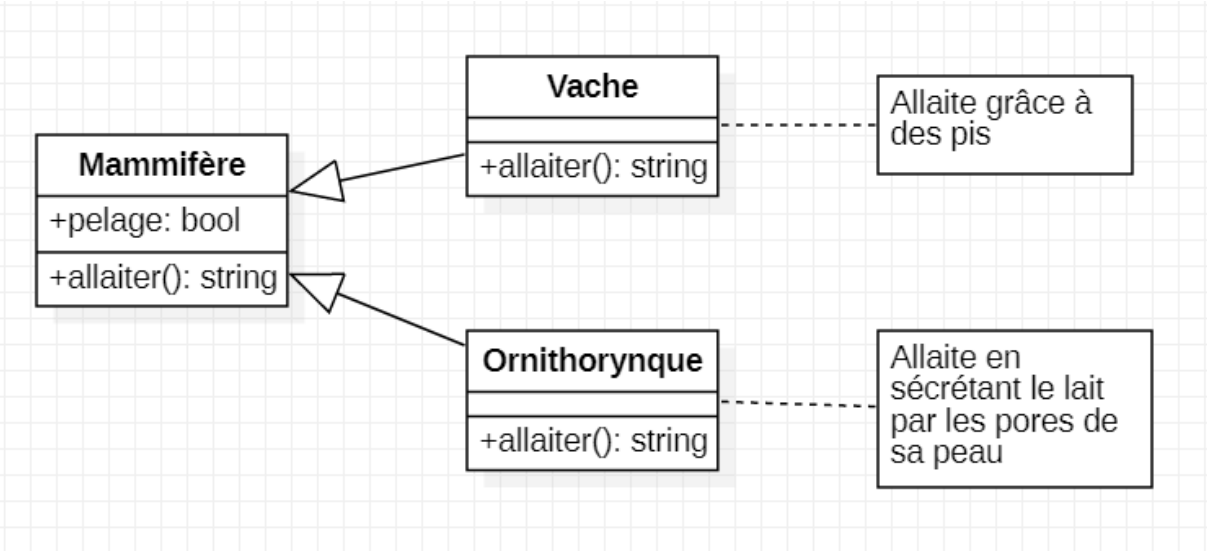
Du point de vue du sens, de la signification, l'Héritage traduit le verbe « être ». Un mammifère **EST** un animal. Cependant, bien qu'un arbre possède un certain nombre de patte égale à 0, une couleur marron et vert, se nourrit, communique avec les autres arbres, et se reproduit, un arbre **N'EST PAS** un animal.

Au moment de réfléchir à la structure du système, il peut parfois être tentant d'abuser du système d'Héritage pour permettre à une Classe d'utiliser les données et services d'une autre Classe. Il faut se souvenir que cela est autorisée par les bonnes pratiques uniquement si l'application du verbe **Être** entre la Classe Enfant et la Classe Parent a du sens.

5. Le Polymorphisme

Le Polymorphisme décrit le fait qu'une Classe peut avoir des formes différentes au travers des objets instanciés par ses Classes Enfants.

Ces objets posséderont les mêmes méthodes que la Classe Parent, mais leur auront défini des comportements différents.

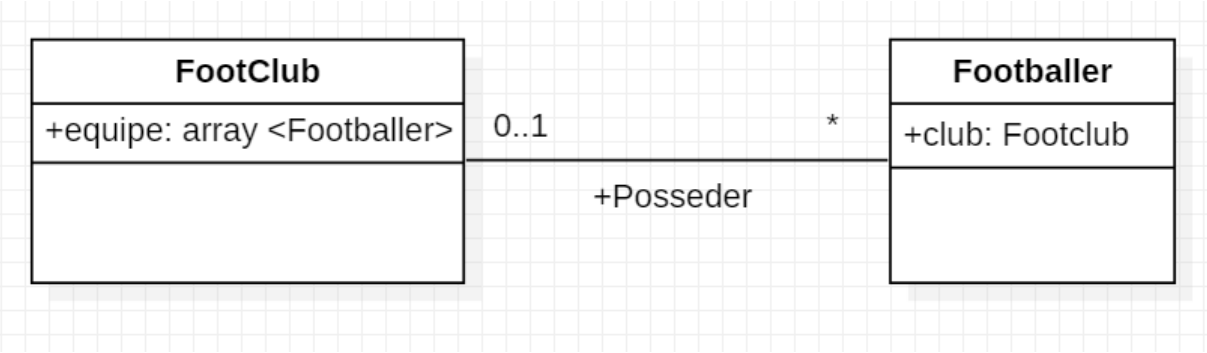


6. Les Associations : le verbe « avoir »

Comme dans le monde réel, des objets peuvent avoir un lien entre eux. Par exemple, il y a un lien entre une équipe de foot et les membres qui la constituent.

En Diagramme de Classe ce lien se traduit par une Association, traduisant le verbe « Avoir ». Par cette relation de possession, une Classe peut avoir accès aux attributs et méthodes d'une autre Classe, en utilisant une instance de cette dernière.

Une Association est symbolisé par un trait reliant les Classes entre elles. Ce trait possède ou non un symbole selon le type d'Association. Un Classe « possédant » une autre, doit y faire référence au sein de ses attributs. Enfin, une Association doit être qualifier par un nom unique et des cardinalités.



1. Les Cardinalités

Les Cardinalités représentent le nombre d'association minimum et maximum qu'une instance d'une Classe peut réaliser avec les instances d'une autre Classe.

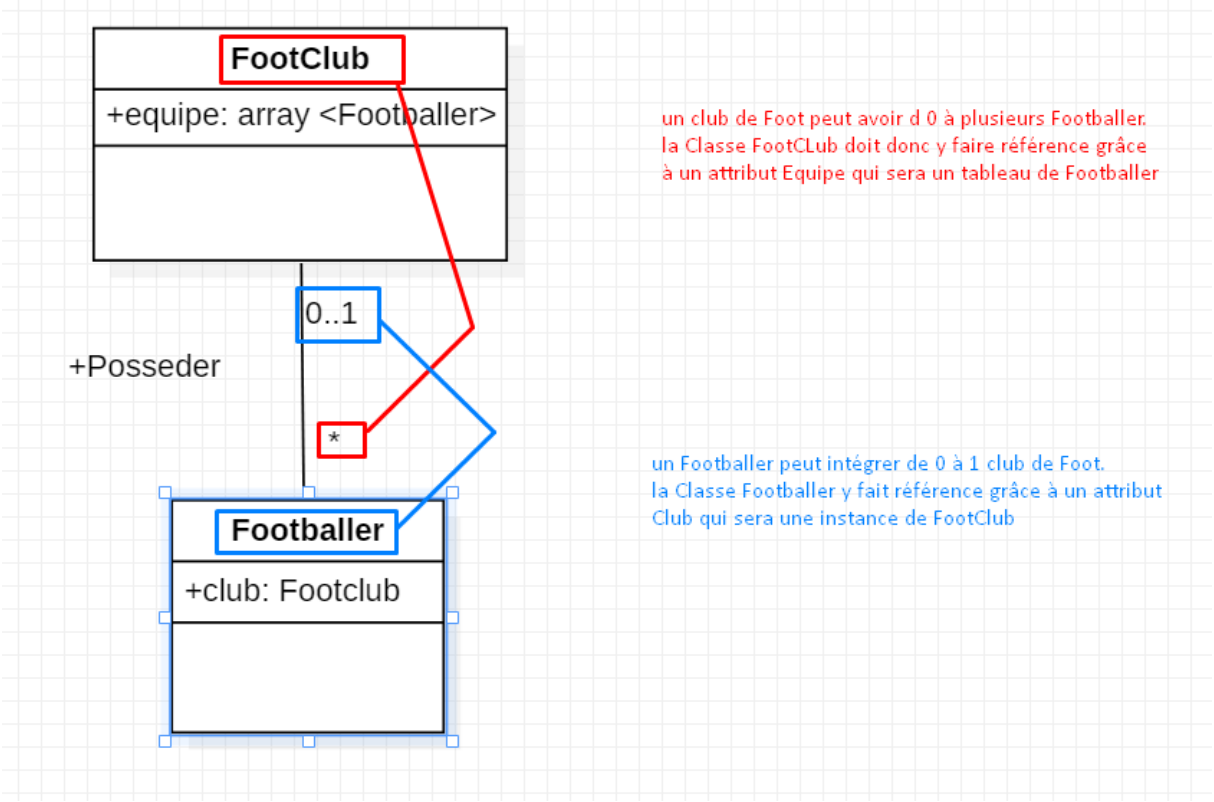
Au sein du couple indiquant la cardinalité, le chiffre de gauche indique le minimum, et le chiffre de droit indique le maximum. Ces 2 chiffres sont séparés par deux fois le point.

Un maximum de plusieurs est noté *.

Dans le cas où le minimum et le maximum sont de 1, il est possible d'abrégé la notation en juste 1.

Dans le cas où le minimum est 0 et le maximum est *, il est possible d'abrégé la notation en juste *.

UML : DIAGRAMME DE CLASSE



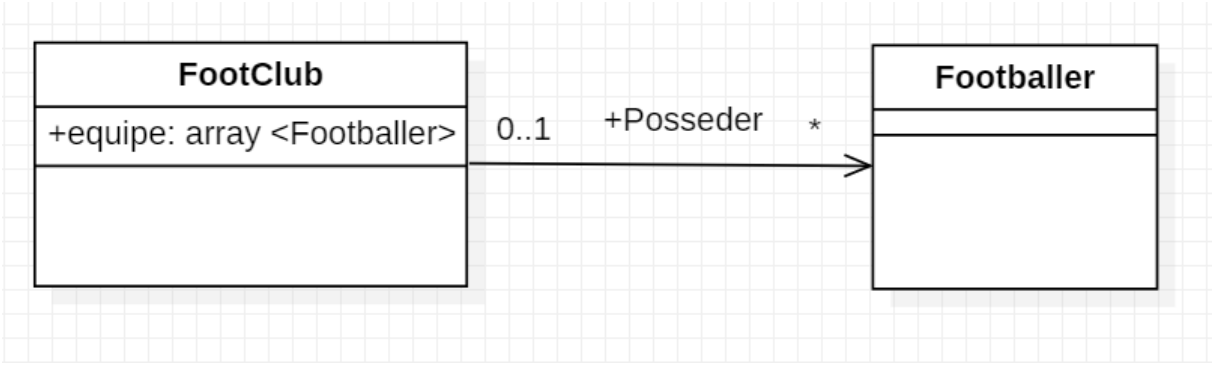
2. Association Directe

Dans l'exemple précédent, l'Association utilisée est une Association Bidirectionnelle. Cela signifie que chaque Classe impliquée dans l'Association possède l'autre.

Il faut éviter autant que possible d'utiliser ce type d'Association, car cela est complexe à coder.

A la place, il vaut mieux définir une Association Directe.

Dans une Association Directe, la relation de possession se fait uniquement dans un sens. Une telle Association est symbolisée par une flèche en direction de la Classe possédée.



3. Agrégation

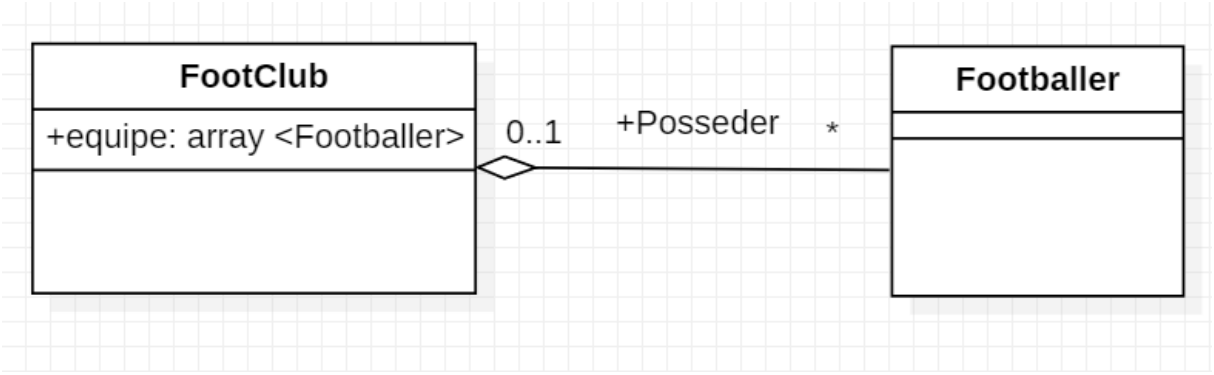
Une Agrégation est une relation dite de Composition Faible.

Elle définit une relation dans laquelle une instance d'une Classe est composée d'un ensemble d'instance d'une autre Classe. Par exemple : une équipe est composée de footballers, une entreprise est composée de salariés, une session est composée de stagiaires.

Contrairement à l'Association Directe, la relation d'Agrégation traduit une forme de hiérarchie : une Classe plus importante et plus grande englobe une plus petite.

Cette Composition est Faible, dans le sens où la suppression des objets d'une Classe n'entraîne pas la suppression des objets de l'autre Classe. Ainsi, l'Agrégation est compatible avec toutes les combinaisons de Cardinalité.

Une Agrégation est symbolisée par un losange blanc du côté de la Classe composée par l'autre Classe.

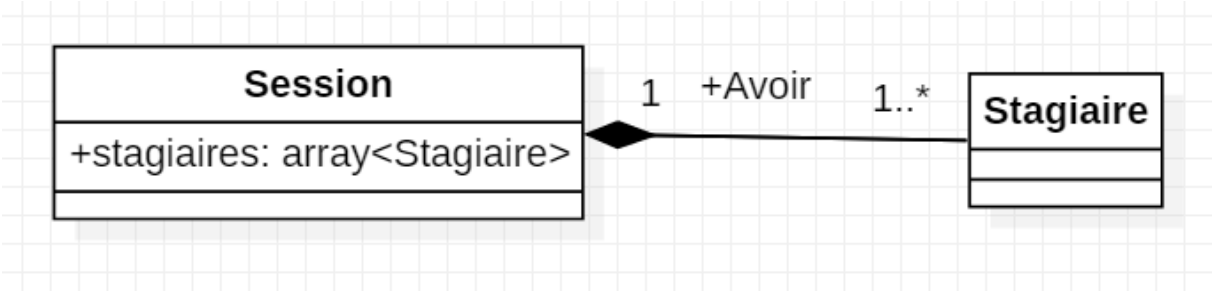


4. Composition

Une Composition est une relation dite de Composition Forte.

Elle définit le même rapport entre deux Classes que l'Agrégation, à la différence que la suppression de la Classe composé entraîne la suppression en cascade de la Classe composante. Ainsi cette Association n'est compatible qu'avec une cardinalité ayant un maximum de 1 du côté du composé.

Une Composition est symbolisée par un losange noir du côté de la Classe composée par l'autre Classe.



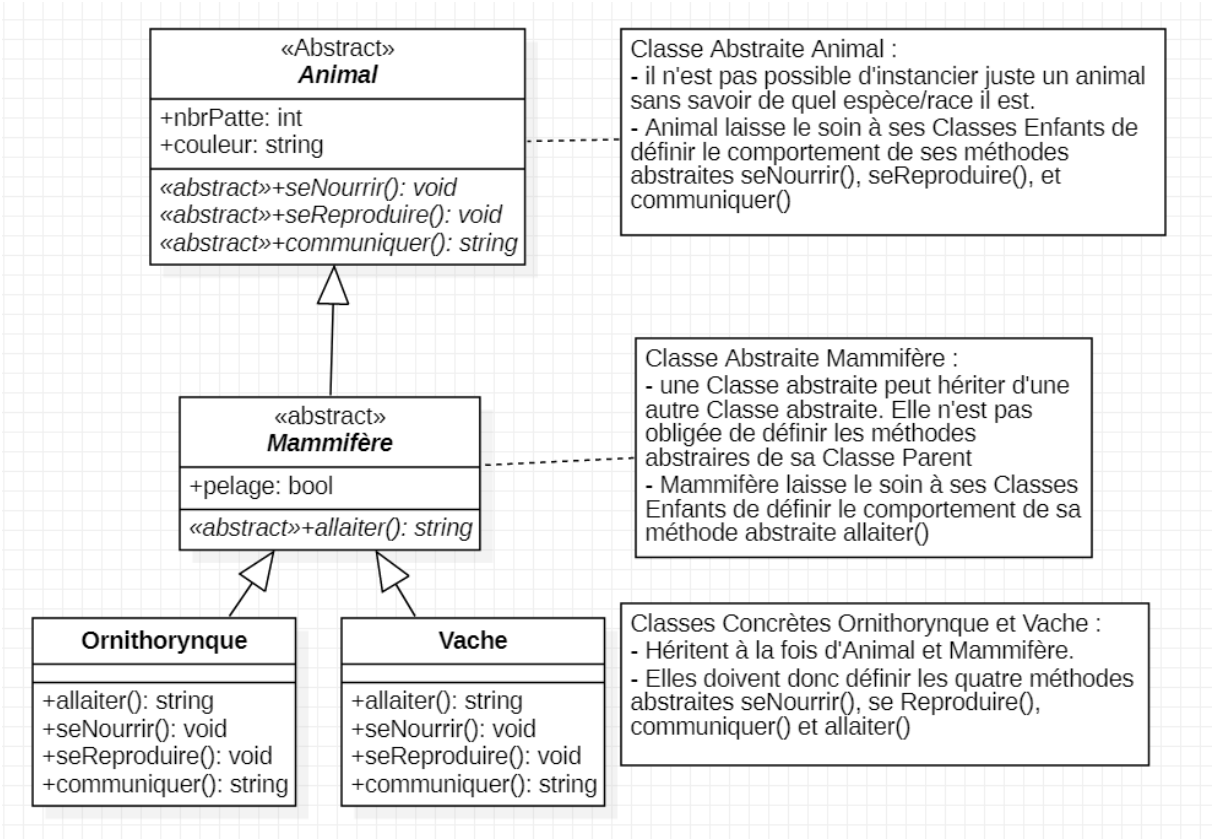
7. Classe Abstraite

Une Classe Abstraite est une Classe qui possède au moins une Méthode Abstraite, c'est à dire une méthode qui n'est pas définie (sauf par sa signature : nom de méthode, paramètre, et typage).

Une Classe Abstraite ne peut pas instancier d'objet. Pour cela on doit passer par une ou des Classes qui héritent de la Classe Abstraite. C'est comme un plan de construction incomplet, que chaque développeur peut combler selon ses besoins.

Sur le Diagramme de Classe, le nom d'une Classe Abstraite est en italique - ou précédé de la notation <Abstraite>

Une méthode Abstraite est représentée en italique.



8. Interface

Les Interfaces ressemblent aux Classes Abstraites, à la seule différence qu'elles ne possèdent uniquement que la signature de leurs méthodes (nom, paramètre, typage), ainsi que des Constantes.

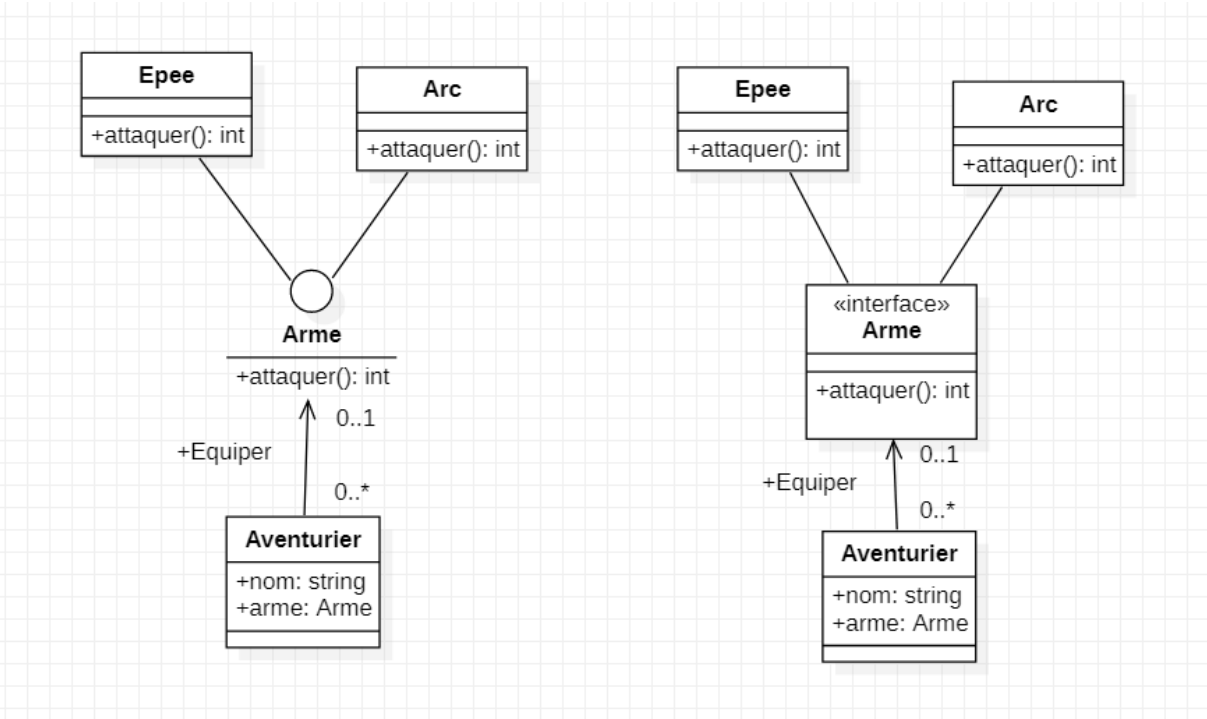
Ainsi, une classe implémentant une Interface doit OBLIGATOIREMENT définir chaque méthode de l'Interface.

Il est a noté qu'il est possible pour une Interface d'hériter (extends) d'une ou plusieurs autre Interface.

Représentation des Interface en Diagramme de Classe (2 manière) :

- Sous forme de Classe avec le stéréotype Interface (noté avant le nom entre double chevron <<Interface>>)

- Sous forme de Cercle (outils Interface) : on utilisera alors l'outils Interface Realization (Réalisation d'Interface) pour l'associer aux Classes qui l'implémentent



9. Classes Abstraites ou Interface ?

Pour comprendre l'intérêt des Interfaces face aux Classes Abstraites, il faut voir les choses ainsi :

- Une Classe Abstraite est un plan de construction avec des blancs pouvant être remplis à sa guise par un Développeur qui crée une Sous-Classe qui en hérite.

- Une Interface est un Contrat de Service, comme une Norme, qui oblige chaque Développeur à implémenter chaque méthode définie par cette norme, avec la même signature de méthode. Cela garantissant une homogénéité au sein de l'écosystème de développement.

- Une autre différence avec les Classes Abstraites est qu'une Sous-Classe ne peut hérité que d'UNE et UNE SEULE Classe Parent, MAIS elle peut implémenter PLUSIEURS Interface.

10. Pour aller plus loin

Pour aller plus loin dans la manière de structurer un système en POO, il est fortement recommandé de s'intéresser aux principes SOLID et aux Design Patterns.

Les principes SOLID sont un ensemble de 5 règles permettant d'aboutir à une structure propre, facile à maintenir, facile à modifier, et évitant les effets de bords. On peut les résumer ainsi :

S pour **Single Responsibility** (Responsabilité) : une Classe, une fonction ou une méthode ne doit s'occuper de faire qu'une chose et n'avoir qu'une seule et unique raison d'être modifiée.

O pour **Open to Extend / Closed to Modify** (Ouvert à l'extension / Fermée à la modification) : une Classe testée et approuvée doit être fermée à la modification directe, mais ouverte à l'extension pour étendre ce qu'elle peut faire.

L pour **Liskov Substitution** (Substitution de Liskov) : une Classe Parent doit pouvoir être remplacée par ses Classes Enfants sans que cela n'affecte le code.

I pour **Interface Segregation** (Ségrégation d'Interface) : il est préférable que les Interfaces soient spécifiques et spécialisées, quitte à la multiplier, plutôt que généralistes en intégrant de trop nombreuses méthodes.

D pour **Dependency Inversion** (Inversion des Dépendances) : il faut dépendre des abstractions (Classe Abstraite, Interface), et non des implémentations concrètes (Classe Concrète).

La playlist **Principes SOLID et Design Patterns** de la chaîne youtube **Cours-en-ligne** est une bonne porte d'entrée commencer.

De leur côté, les Design Patterns sont un ensemble de 23 solutions, se reposant sur les principes SOLID, pour répondre à des problèmes spécifiques de code. Par exemple, on peut citer le Design Pattern Singleton qui permet de s'assurer qu'une Classe n'est instanciée qu'une et une seule fois, ou bien le Design Pattern Adapter qui permet à deux Classes incompatibles de collaborer.

Un très bon livre, très pédagogue, a été écrit à ce sujet : **Design Pattern, Tête la Première**, et est devenu le livre de chevet de nombreux développeur.

Le site **Refactoring Guru** fournit aussi une excellente ressource pour les découvrir.