Chapitre 2

Programmation orientée objet

2.1 Classes et instances

2.1.1 Classes et instances

La Programmation Orientée Objet (POO) est un paradigme de programmation reposant sur le principe que chaque entité informatique est une **instance** (on pourrait aussi parler de réalisation ou d'avatar) d'une **classe** d'objets. On pourrait voir une classe comme une usine (de vélos, de vêtements, etc) et les instances comme les produits de cette usine (donc les vélos, les vêtements, etc).

2.1.2 Attributs et méthodes d'instance

Chaque objet ou instance possède des :

Attributs : il s'agit des caractéristiques de l'objet ;

Méthodes : ce sont des fonctions permettant de modifier ses attributs.

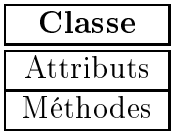
Exemple : considérons une usine de vaisseaux spatiaux, il s'agit de la classe. Si on prend un vaisseau spatial en particulier – le Normandy par exemple –, il s'agit d'une instance.

Chaque vaisseau possède des caractéristiques : longueur, poids, accélération maximale, vitesse maximale, position, vitesse, etc. Ce sont les attributs de l'objet et donc de la classe.

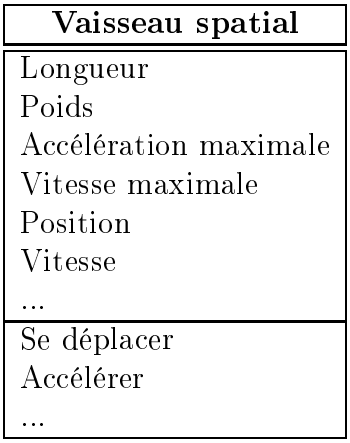
Il est possible de modifier ces caractéristiques (attributs) grâce à des actions telles que se déplacer et accélérer qui changent respectivement la position et la vitesse.

Si les caractéristiques d'un objet correspondent aux attributs d'une classe, ce sont les actions que l'on peut effectuer avec cet objet qui correspondent aux méthodes.

Les attributs et méthodes peuvent être représentés sous la forme d'un schéma UML (Unified Modeling Language) ayant la structure suivante :



Exemple : reprenons nos vaisseaux spatiaux et leurs attributs et méthodes.



2.1.3 Attributs de classe

Il est possible de définir des attributs de classe. Ceux-ci ne sont pas propres à des instances mais à la classe elle-même.

Exemple : on peut considérer l'attribut « compteur de vaisseaux » pour notre classe vaisseaux spatiaux. Celui-ci permettrait de compter le nombre d'instances créées par la classe, autrement dit, le nombre de vaisseaux spatiaux.

2.2 Encapsulation

2.2.1 Public et privé

Dans la plupart des langages dédiés à la POO, on peut créer des attributs et des méthodes publiques et privés. Cette différenciation empêche l'utilisateur d'avoir accès aux attributs et méthodes privés et ainsi limite les risques d'actes de malveillance ou d'erreurs provoquées par ceux-ci.

En Python, ces différenciations n'existent pas. Tout est publique. Il est cependant possible de créer une protection des attributs et méthodes que l'on souhaiterait privés grâce à une syntaxe que nous verrons en TP.

2.2.2 Méthodes spéciales

Pour accéder à et ou modifier des attributs privés d'une classe, ils existent des méthodes dîtes spéciales qui permettent d'utiliser lesdits attributs tout en les protégeant. On peut en distinguer trois catégories.

Les constructeurs (« constructors » en anglais) qui construisent l'instance et initialisent les attributs.

Les accesseurs (« getters » en anglais) qui permettent d'accéder à la valeur d'un attribut.

Les mutateurs (« setters » en anglais) qui permettent de modifier la valeur d'un attribut.

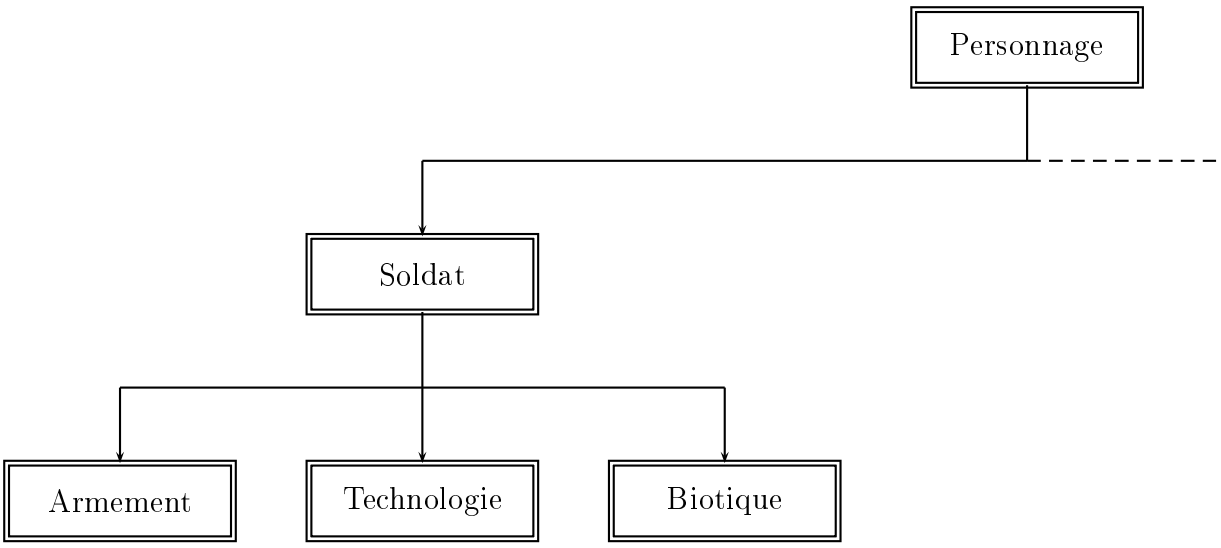
2.3 Héritage

2.3.1 Héritage simple

En POO, il est possible de créer une sous-classe d'une classe. On parle aussi de classes **parentes** et **enfants**. Le grand avantage est que les classes enfants héritent des attributs et méthodes de la classe parente en plus de posséder leurs propres attributs et méthodes.

Exemple : considérons le jeu Mass Effect, il s'agit d'un jeu vidéo de science fiction dans lequel on incarne un soldat humain voyageant à travers la galaxie et interagissant avec d'autres espèces aliens (en très très résumé).

Tous les personnages, qu'ils soient jouables ou non, possèdent des caractéristiques et des capacités communes, un nom, une apparence, la faculté de se déplacer, etc. Le personnage principal jouable par le joueur possède aussi ces caractéristiques mais il en a beaucoup plus. Lors de sa création, le joueur peut choisir parmi plusieurs catégories de soldats ; chacune ayant ses caractéristiques et capacités propres. Pour simplifier, nous allons en considérer trois : l'expert en armement, l'expert en technologie et l'expert en pouvoirs psychiques dits biotiques. Cela se représente sous la forme de l'arborescence ci-dessous.



On pourrait très bien commencer par construire une classe `Personnage` regroupant toutes les caractéristiques communes aux personnages du jeu : nom, capacité à se déplacer, etc. Puis une sous-classe de `Personnage` : `Soldat` qui contiendrait les spécificités liées aux soldats (meilleures capacités physiques, armements, etc). Enfin, des sous-classes de `Soldat` en fonction de l'expertise : `ExpertArmement`, `Ingenieur` et `Biotique`.

Chacune de ces classes hériterait des attributs et méthodes de sa classe parente. Ce qui en faciliterait la conception et permettrait de se concentrer sur les spécificités de chacune.

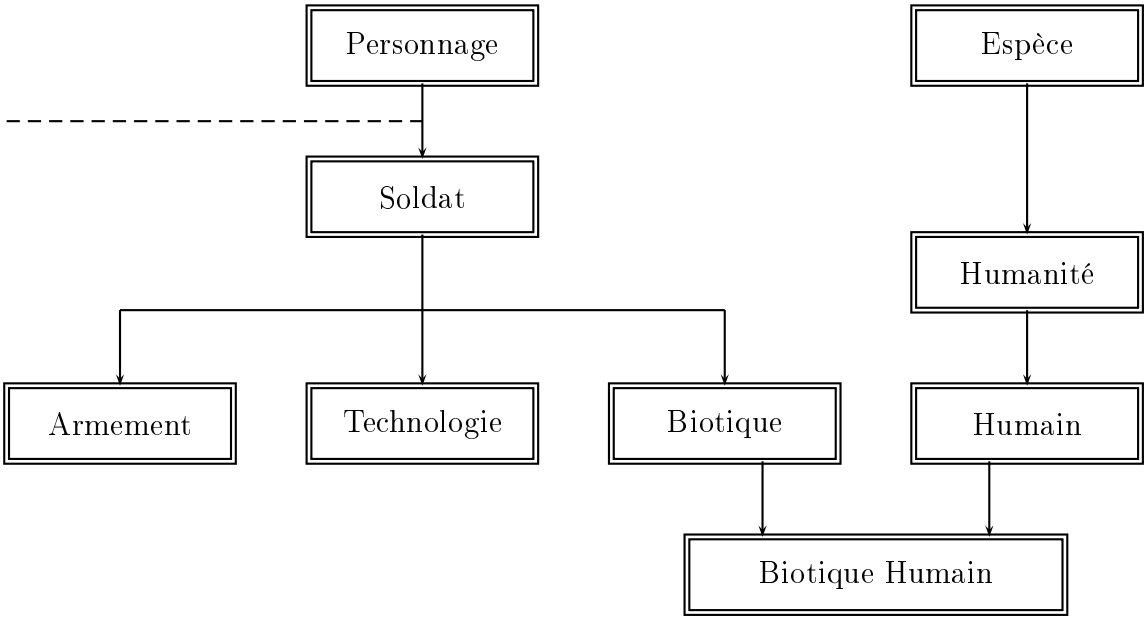
Finalement, la classe personnage servirait aussi à construire tous les autres personnages, comme les médecins, les techniciens, les pilotes, etc.

Remarque : on peut représenter les relations parent enfant entre les classes sur les schémas UML comme dans l'arborescence ci-dessus avec des flèches descendant du parent vers l'enfant.

2.3.2 Héritage multiple

Il est possible pour une classe enfant d'hériter de plusieurs classes parentes. Toutefois, nous ne y attarderons pas dans ce cours. Nous en verrons juste l'intérêt dans l'exemple suivant.

Exemple : reprenons le jeu Mass Effect. Chaque espèce galactique possède des spécificités : biologie, apparence, tempérament, planète d'origine, Histoire, etc. On pourrait créer une classe parente abstraite `Espèce` contenant les attributs et méthodes communes à toutes les espèces : nom, planète d'origine, Histoire, etc. Puis on pourrait ensuite créer des sous-classe pour chaque espèce comme `Humain` par exemple. On pourrait alors créer une classe `BiotiqueHumain` héritant de la classe `Humain` et de la classe `Biotique`.



2.4 Polymorphisme

Le **polymorphisme** est le fait que des sous-classe d'une classe parente puissent avoir des méthodes différentes les unes des autres et de la classe parente : elles ont été adaptées au besoin de la classe enfant. L'ajout ou la modification de méthodes d'une classe parente est connue sous le nom de **surcharge**.

Exemple : reprenons nos classes de soldats dans Mass Effect. Chaque soldat est formé au maniement des armes. On peut donc penser à des méthodes pour la classe `Soldat` comme `viser`, `tirer`, etc. Cependant, les trois spécialisations correspondant aux classes `Armement`, `Technologie` et `Biotique` n'ont pas la même maîtrise des armes les unes que les autres : l'expert en armement sera bien meilleur qu'un soldat lambda alors que l'ingénieur et le biotique ayant développé d'autres compétences seront moins bons. On pourra donc tenir compte de cela en surchargeant les méthodes `viser` et `tirer` de la classe `Soldat`.

2.5 Ressources supplémentaires

— Vidéo sur les paradigmes de programmation

— Vidéo sur la POO

2.6 Exercices

Exercice 2.1. Les phrases suivantes décrivent des contextes. Pour chacune d'entre elles, donner la classe associée au contexte, ses attributs et méthodes dans un schéma UML.

1. Un joueur de Quidditch a un nom et possède un balai volant. Il peut monter et descendre du balai et voler.
2. Un poursuiveur (poste particulier au Quidditch) peut avoir le souafle (balle particulière au Quidditch). Il peut le passer ou tenter de tirer au but s'il l'a ; sinon, il peut le recevoir par une passe ou le prendre à un joueur adverse.
3. Un batteur (autre poste particulier au Quidditch) possède une batte. Il peut s'en servir afin de frapper le cognard (autre balle particulière au Quidditch) et l'envoyer ainsi dans une certaine direction.
4. Sachant qu'il y a dans une équipe de Quidditch, un attrapeur, un gardien, deux batteurs et trois poursuiveurs, quels attributs de classe pourrait-on créer pour contrôler cela ?
5. Si vous aimez le Quidditch, vous pouvez aussi réfléchir à des classes pour le gardien et l'attrapeur.

Exercice 2.2. Dans l'exercice précédent, identifier les classes parentes et enfants. Compléter le schéma UML afin de représenter ces dépendances.

Exercice 2.3. Proposer des classes pour les différentes balles du Quidditch, notamment en partant d'une classe abstraite `Balle`.

Exercice 2.4. Proposer un schéma UML avec des relations parent enfant permettant de représenter les Stormtroopers de Star Wars et leurs différentes spécificités : fantassins, pilotes de chasseur, de moto-jets, de TBTT, etc. On pourra aller sur la page Wikipédia des Stormtroopers afin de trouver plus d'informations.