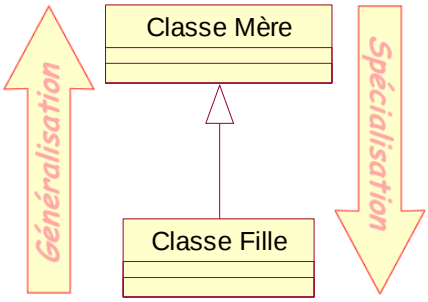
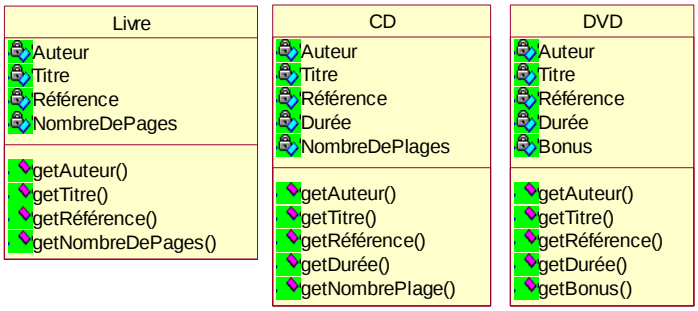
Le mécanisme d'héritage permet de mettre en relation un certain nombre de classes ayant des caractéristiques communes (attributs et comportements) en respectant une certaine filiation.

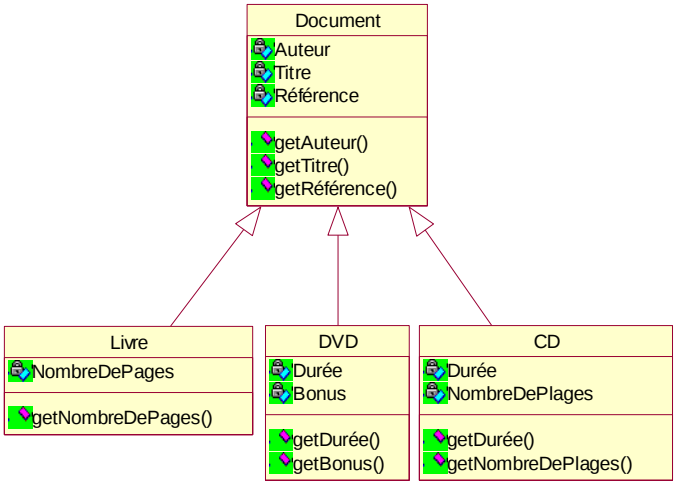


Généralisation

Imaginons que nous devions fabriquer un logiciel qui permet de gérer une bibliothèque. Cette bibliothèque comporte plusieurs types de documents ; des livres, des CDs, ou des DVDs. Une première étude nous amène à mettre en œuvre les classes suivantes.

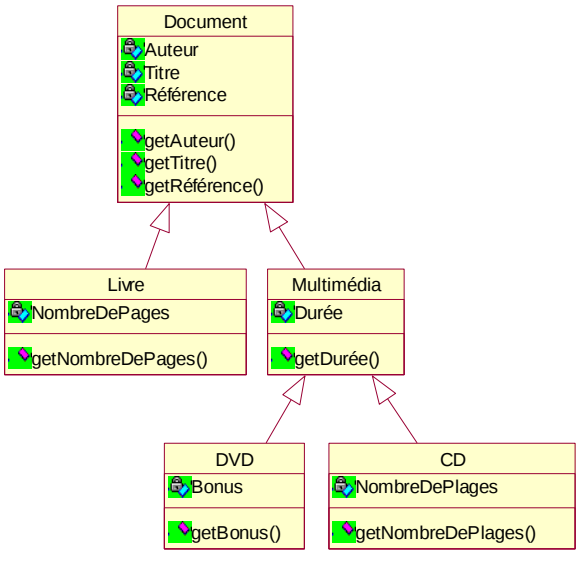


Nous remarquons que dans les trois types de documents, un certain nombre de caractéristiques se retrouvent systématiquement. Afin d'éviter la répétition des éléments constituant chacune des classes, il est préférable de factoriser toutes ces caractéristiques communes pour en faire une nouvelle classe plus généraliste. En effet, nous pouvons dire que, d'une façon générale, et quelque soit le type de document, il comporte au moins un titre, un auteur, etc. il semble allait de soi, que le nom de cette nouvelle classe générale s'appelle justement « *Document* ». Il faut ensuite proposer une relation entres les classes afin de montrer la filiation. Par exemple, il faut bien préciser qu'un « *Livre* » est aussi un « *Document* ».



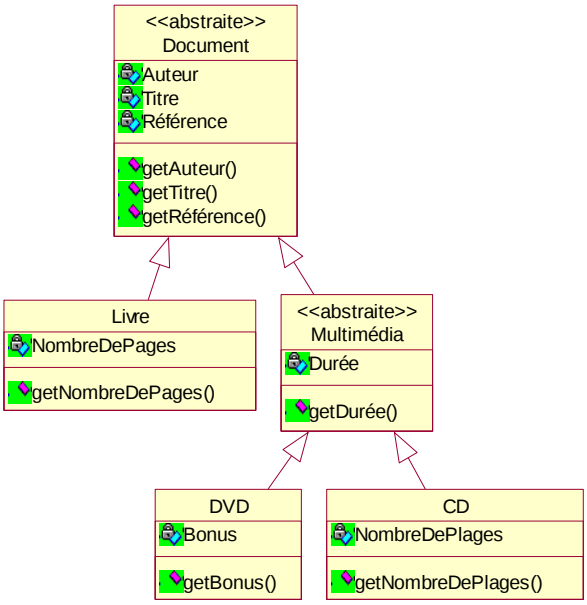
La généralisation se représente par une flèche qui part de la classe fille vers la classe mère. Par exemple, Un « *Livre* » possède, certes un nombre de page, mais en suivant la flèche indiquée par la relation de généralisation, elle comporte également un nom d'auteur, un titre, une référence. En fait, la classe « *Livre* » hérite de tout ce que possède la classe « *Document* », les attributs comme les méthodes. Finalement, cela correspond exactement à ce que nous avions avant sans l'héritage, sauf qu'avec cette technique, nous évitons toutes les duplications. Toutes les caractéristiques communes n'apparaissent plus dans chacune des classes filles, alors qu'elles sont bien présentes implicitement.

Dans cet exemple, nous avons un seul niveau d'héritage, mais il est bien entendu possible d'avoir une hiérarchie beaucoup plus développée. D'ailleurs, si nous regardons de plus prêt, nous remarquons que nous pouvons appliquer une nouvelle fois la généralisation en factorisant la « *durée* » du support « *CD* » et du support « *DVD* ». En fait, il s'agit dans les deux cas d'un support commun appelé « *Multimédia* ».



Classes abstraites

Lorsque nous allons mettre en œuvre notre programme de gestion de bibliothèque, nous allons avoir finalement un certain nombre d'objets relatifs à chacun de ces documents. Toutefois, si nous regardons de plus près, nous n'aurons, par exemple, aucun objet relatif à la classe « *Document* ». En effet, les objets qui intéressent le bibliothécaire, sont les livres, les CDs et les DVDs. Déclarer un objet de la classe « *Document* » n'aurait pas de sens. Il s'agit d'une abstraction. Cette classe n'existe que pour les classes filles qui elles correspondent à quelque chose de concret. Nous pouvons d'ailleurs appliquer le même raisonnement à la classe « *Multimédia* ». Il est d'ailleurs de notre devoir d'empêcher que ces classes puissent fournir des objets. De telles classes sont appelées « *abstraites* ». Du coup, les classes classiques sont appelées classes « *concrètes* ». Nous ne pouvons déclarer des objets que sur des classes « *concrètes* ».

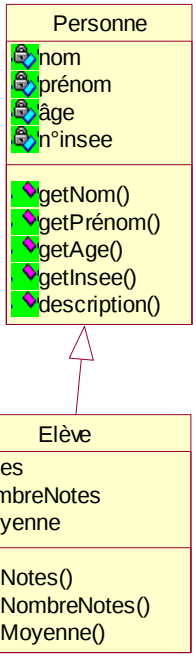


> ***Remarque***
> Attention, il ne faut pas penser que la généralisation donne systématiquement des classes abstraites. La plupart du temps, les classes sont très souvent concrètes.

Spécialisation

Dans la conception des classes, nous pouvons avoir une démarche inverse de la généralisation, c'est-à-dire, cette fois-ci, de partir plutôt de la classe mère pour aboutir ensuite aux classes filles.

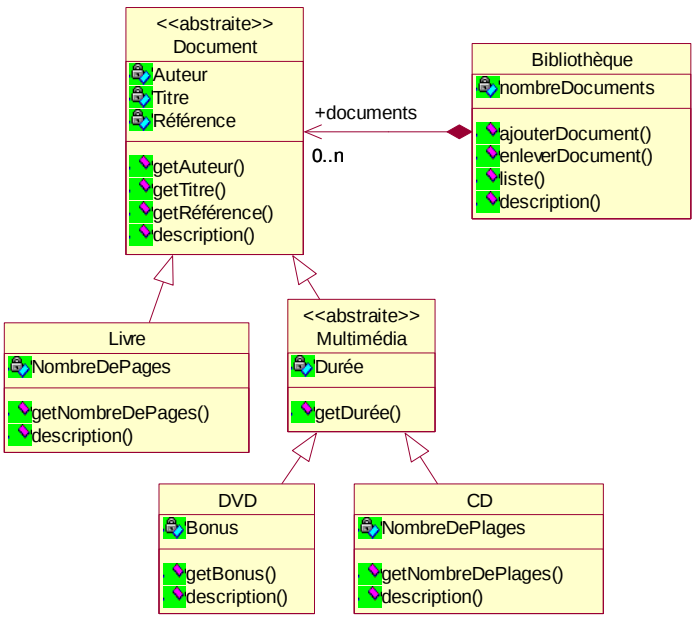
Le concept d'héritage constitue l'un des fondements de la programmation orientée objet. En particulier, il est à la base des possibilités de réutilisation des composants logiciels (en l'occurrence de classes). En effet, il vous autorise à définir une nouvelle classe, dite « *dérivée* », à partir d'une classe existante dite « *de base* ». La classe dérivée héritera donc des potentialités de la classe de base, tout en lui ajoutant de nouvelles sans remettre en question la classe de base. Il ne sera pas utile de la recompiler, ni même de disposer du programme source correspondant (exception faite de sa déclaration).



Cette technique permet donc de développer de nouveaux outils en se fondant sur un certain acquis, ce qui justifie le terme d'héritage. Comme nous venons de le voir, plusieurs classes peuvent être dérivées de la même classe de base. En outre, l'héritage, n'est pas limité à un seul niveau : une classe dérivée peut devenir à son tour classe de base pour une autre classe. Nous voyons apparaître la notion d'héritage comme outil de spécialisation croissante.

Par exemple, imaginons que nous disposions déjà d'une classe concrète « *Personne* ». Nous pouvons alors spécialiser cette classe afin d'obtenir une classe « *Elève* » qui reste bien entendu une personne mais qui possède en plus un certain nombre de spécificités comme, par exemple, la gestion des notes. Dans cet exemple, vous remarquez d'ailleurs que les deux classes sont des classes concrètes.

Utilisation de l'héritage et intérêt



Une fois que la hiérarchie est constituée, il est possible d'accéder à n'importe qu'elle classe représentant la parenté. Ainsi, d'après le diagramme UML, nous avons la classe « *Bibliothèque* » qui est composée d'un l'ensemble de documents. Vous remarquez que nous connectons l'agrégation sur la classe ancêtre « *Document* » uniquement. Toutefois, cela sous-entend que la classe « *Bibliothèque* » est composée en fait de n'importe quel document, c'est-à-dire n'importe quelle classe faisant partie de la hiérarchie. Ainsi, le premier document de la bibliothèque sera peut-être un CD, et le deuxième un livre.

Alors que la composition indique qu'une classe possède plusieurs éléments, l'héritage indique plutôt que nous utilisons une classe parmi toutes celles proposées par la hiérarchie.

Nous pourrons donc déclarer des objets par rapport à des classes concrètes faisant parti de cet héritage. La bibliothèque sera alors capable d'enregistrer chacun d'entre eux sans aucune difficulté, du moment bien entendu, qu'ils fassent effectivement partie de cette hiérarchie. C'est un mécanisme extrêmement performant et qui a fortement séduit le monde des développeurs. Le concept est facile à comprendre et par ailleurs la mise en œuvre est relativement simple. Le plus génial dans ce système, c'est qu'il est même possible de rajouter une classe à cet héritage sans qu'il soit nécessaire de tout reconstruire. En effet, Il n'est pas nécessaire de recompiler le code relatif à la classe « *Bibliothèque* », puisqu'elle est capable d'appréhender n'importe qu'elle classe de la hiérarchie des documents, même si une nouvelle classe est composée après coup.

Le polymorphisme

Le terme polymorphisme indique qu'une entité peut apparaître suivant plusieurs formes. Dans le cas de notre hiérarchie de documents nous remarquons qu'il existe une méthode qui porte le même nom. Il s'agit de « *description()* ». Elle apparaît sur toutes les classes faisant partie de la hiérarchie. Normalement, le principe même de l'héritage, c'est que lorsque une méthode est décrite sur une classe parente, elle est automatiquement hérité par les classes enfants. Si malgré tout, nous spécifions le même nom de la méthode sur les enfants, cela signifie que le comportement sera différent. En effet, la description d'un CD est différente de celle d'un livre.

Nous sommes là en présence d'un polymorphisme. La description d'un document dans le sens général n'est pas suffisante pour la description d'un livre. Cette technique s'appelle une redéfinition, c'est-à-dire que dans la classe dérivée, nous allons redéfinir une méthode qui porte le même nom avec une signature identique (polymorphisme) que la classe de base.

ATTENTION

Il ne faut pas mélanger la redéfinition et la sur-définition.

1. Une ***sur-définition*** (ou surcharge) permet d'utiliser plusieurs méthodes qui portent le même nom au sein d'une même classe, avec une signature différente, pour que le système puisse s'y retrouver.
2. Une ***redéfinition*** permet de fournir une nouvelle définition d'une méthode d'une classe ascendante et ainsi de substituer la description qui en été faite. Nous avons également le même nom que la méthode parente mais surtout avec une signature rigoureusement identique.

Alors que ce mécanisme de polymorphisme parait simple et intuitif, il en sera autrement lorsque nous aborderons ce sujet dans le langage de programmation.

Héritage multiple et classes paramétrables

Nous pouvons même envisager qu'une classe puisse hériter de plusieurs caractéristiques. Ainsi une classe peut être issue de plusieurs classes, c'est ce qui s'appelle l'héritage multiple.

Ainsi, le cheval est à la fois un quadrupède et un herbivore.

Pour finir, il est bien évidemment possible de proposer l'héritage sur des classes paramétrables.