MODELISATION DES STRUCTURES ELEMENTAIRES

1 Les classes

Ce sont les briques de base les plus importantes d'un système O.O..

Classe = description d'un ensemble d'objets partageant les mêmes attributs, méthodes, relations et sémantique. Une classe peut implémenter une ou plusieurs interfaces.

Les classes sont utilisées pour capturer le vocabulaire du système que l'on va modéliser. On identifie des éléments importants du système et les relations qui les associent.

Les classes peuvent représenter :

- **Des éléments logiciels**
- **Des éléments matériels**
- **Des objets purement conceptuels**

Une classe = une abstraction d'un ensemble d'objets présentant les mêmes caractéristiques et qui appartiennent au vocabulaire du système.Un objet devient une instance de sa classe correspondante.

Objet = Etat + Comportement + Identité

• Etat : ensemble des valeurs instantanées de tous les attributs d'un objet

• Comportement : regroupe l'ensemble des compétences d'un objet

• Identité : caractérise l'existence d'un objet propre

Une classe est composée:

- **d'attributs: il s'agit de données, dont les valeurs représentent l'état de l'objet . Un attribut = une propriété de la chose modélisée. Elle est commune à tous les objets de cette classe.**

Deux instanciations de classes pourront avoir tous leurs attributs égaux sans pour autant être un seul et même objet, c'est la différence entre *état* et *identité*.

Une classe peut ne pas avoir d'attribut.

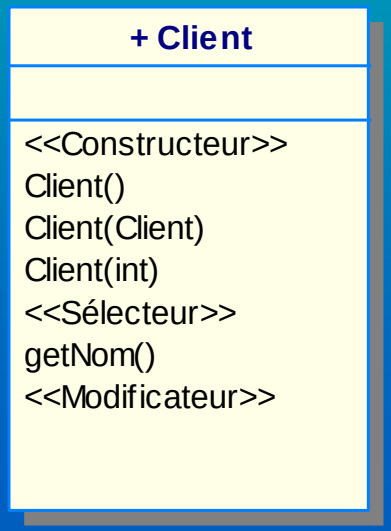
- **d'opérations : il s'agit d' opérations applicables aux objets.**

Une opération= une implémentation d'un service qui peut être demandé à tous les objets d'une même classe en vue de déclencher un comportement.

Il s'agit donc d'une abstraction de ce que peut réaliser un objet d'une classe.

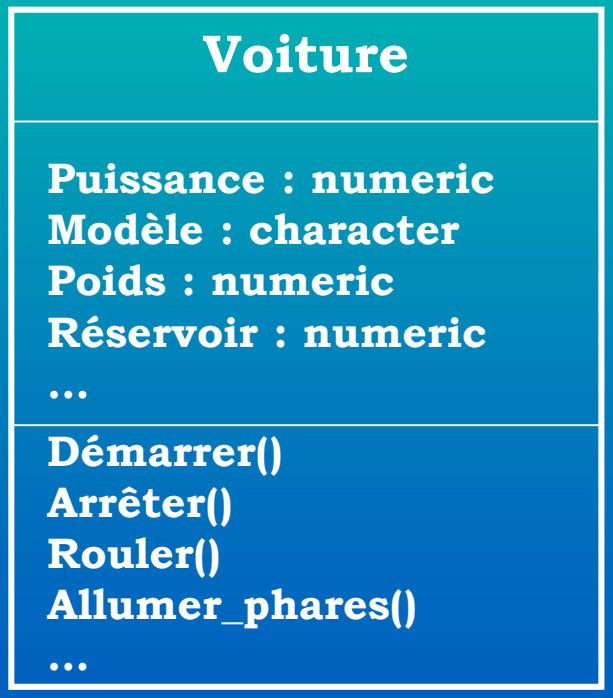
Il existe 5 catégories principales d'opérations :

- **les constructeurs qui créent des objets**
- **les destructeurs qui détruisent des objets**
- **les sélecteurs qui renvoient tout ou une partie de l'état d'un objet**
- **les modificateurs qui changent tout ou une partie de l'état d'un objet**
- **les itérateurs qui visitent l'état d'un objet ou le contenu d'une structure de données qui contient plusieurs objets.**



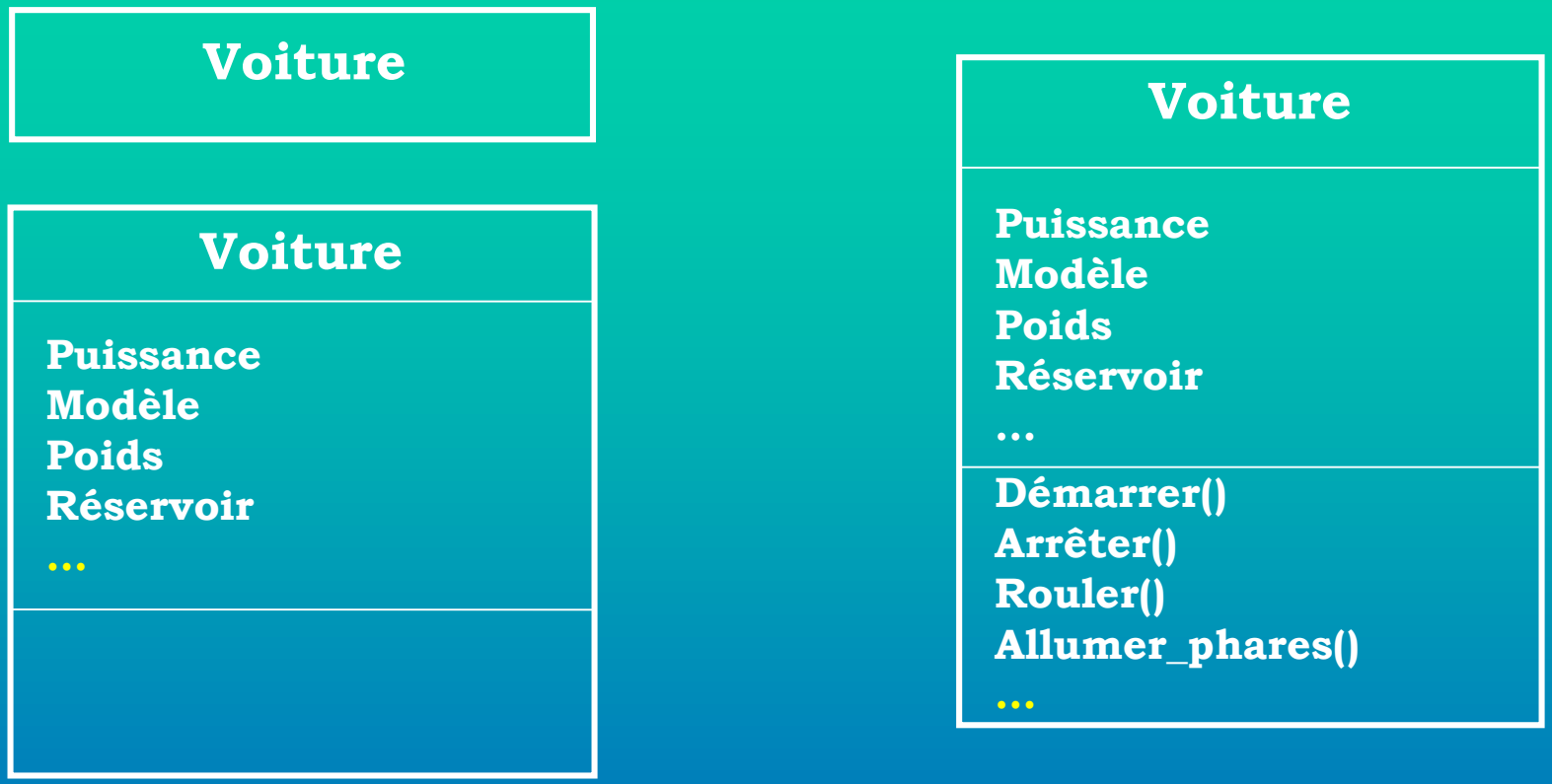
Une classe se représente, avec UML, sous forme d'un rectangle divisé principalement en trois sections :

- le nom donné à la classe, ce nom contrairement aux objets n'est pas souligné.
- les attributs.
- les méthodes



Le nom de la classe doit évoquer le concept décrit par la classe. Il commence par une majuscule. Quand le nom est composé, chaque mot commence par une minuscule et les espaces blancs sont éliminés

Suivant le niveau de détails exigé (en fonction du développement du projet), on peut trouver les notations graphiques suivantes :



Un compartiment vide (d'attributs ou d'opérations) ne signifie pas forcément qu'il n'y a pas d'attributs ou d'opérations; mais que l'on a choisi de ne pas les montrer à ce stade de la modélisation.

On peut également n'en montrer qu'une partie (l'indiquer par …)

Il n’est pas toujours évident, dans une liste d’informations à modéliser, de bien faire la différence entre un objet (classe) et un attribut.

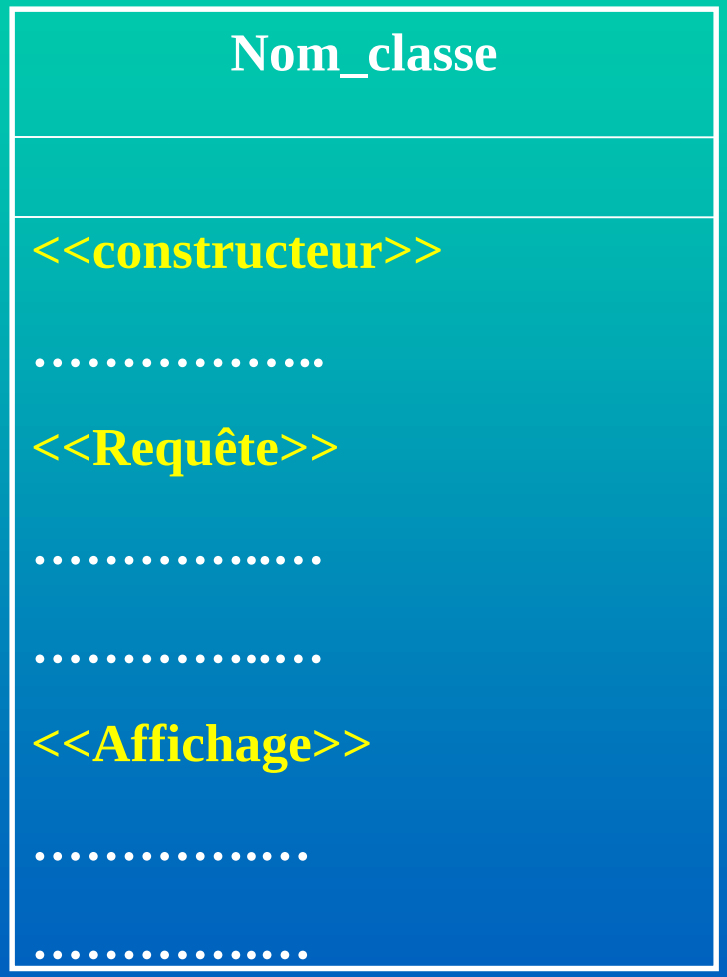
Quand à propos d’un élément :

- **on ne peut demander que sa valeur, il s’agit d’un attribut.**

Exemple : année de naissance

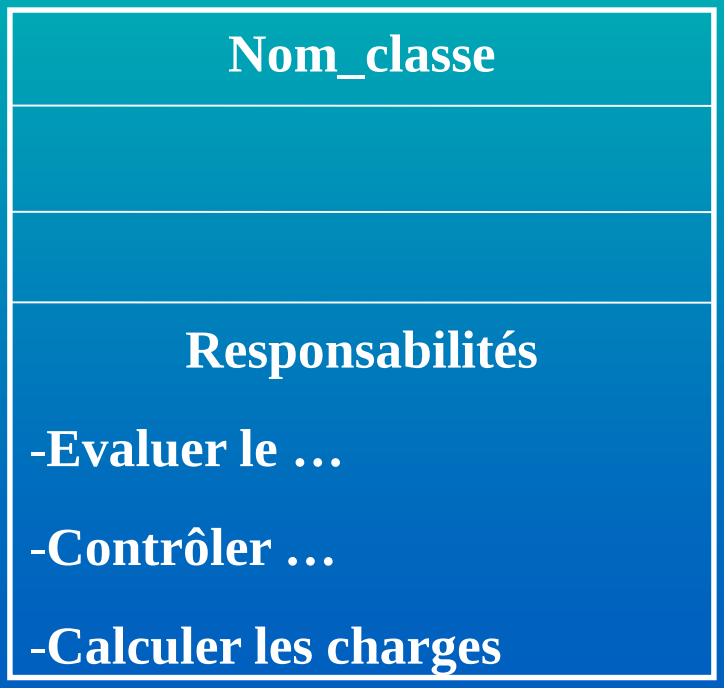
- **on peut lui appliquer plusieurs questions, il s’agit d’un objet qui possède lui-même des attributs. Exemple : Date de naissance → jour, mois et année !**

On peut également organiser les longues listes d'opérations et/ou d'attributs en les regroupant par catégorie et en faisant précéder chaque groupe d'un préfixe (un stéréotype).



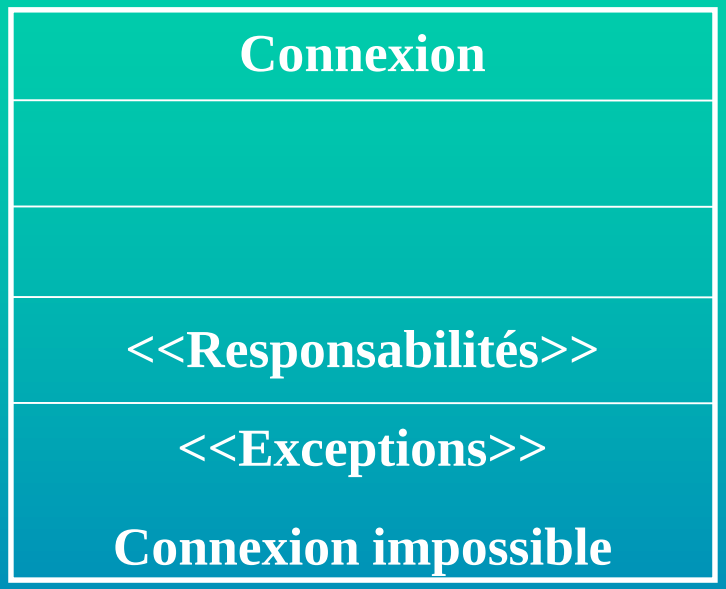
Responsabilité : une responsabilité est un contrat ou une obligation qu'une classe doit respecter. Les attributs et les opérations ne sont que les fonctionnalités qui permettent l'exécution des responsabilités d'une classe.

Généralement, la modélisation d'une classe commence par la définition des responsabilités, et au fur et à mesure du développement du modèle, on traduit ces responsabilités en attributs/opérations.



Le compartiment des responsabilités énumère l'ensemble des tâches devant être assurées par la classe !

Exceptions : Un compartiment supplémentaire peut être ajouté. Il énumère les situations exceptionnelles et les anomalies devant être gérées par la classe.



Les compartiments Responsabilités et Exceptions disparaissent quand on arrive à la phase de génération de code. Ils seront transformés en un ensemble d'attributs et de méthodes.

Modélisation des classes

On utilise les classes pour modéliser les abstractions qui découlent :

- **du problème que l'on essaye de résoudre (Client, Facture, …)**

- **de la technologie utilisée pour résoudre le problème (contraintes informatiques). Exemple : La classe Transaction.**

Chacune des abstractions font partie du vocabulaire du système et forment les éléments les plus importants pour les utilisateurs et les développeurs.

Pour modéliser le vocabulaire, il faut :

- **Identifier les éléments dont les utilisateurs ou programmeurs se servent pour décrire les problèmes ou la solution**
- **Identifier les responsabilités pour chaque abstraction. Veiller à la répartition des responsabilités entre les différentes classes. Pour bien faire, une « bonne classe » doit se contenter de correctement prendre en charge 1, 2 ou 3 responsabilités max.**
- **Définir les attributs et opérations nécessaires à la réalisation des responsabilités.**

Les classes se rassemblent souvent dans des groupes reliés d'un point de vue sémantique et conceptuel dans des packages.

2 Les relations

En général, les classes ne sont pas isolées. Elles collaborent avec d'autres classes. Elles établissent des relations entre elles.

3 relations importantes :

- **les dépendances : relations d'utilisation**
- **les généralisations : relations Mère/Fille**
- **les associations : relations structurelles entre instances**

2.1 La dépendance

Une dépendance permet de mettre en évidence qu'un élément en utilise un autre (relation à sens unique).

Elle établit qu'un changement de spécification de l' élément cible peut affecter l'élément source qui l'utilise.

On peut associer un nom à une dépendance.

La dépendance la plus courante est une classe utilisée comme type de paramètre d'une méthode d'une autre classe.

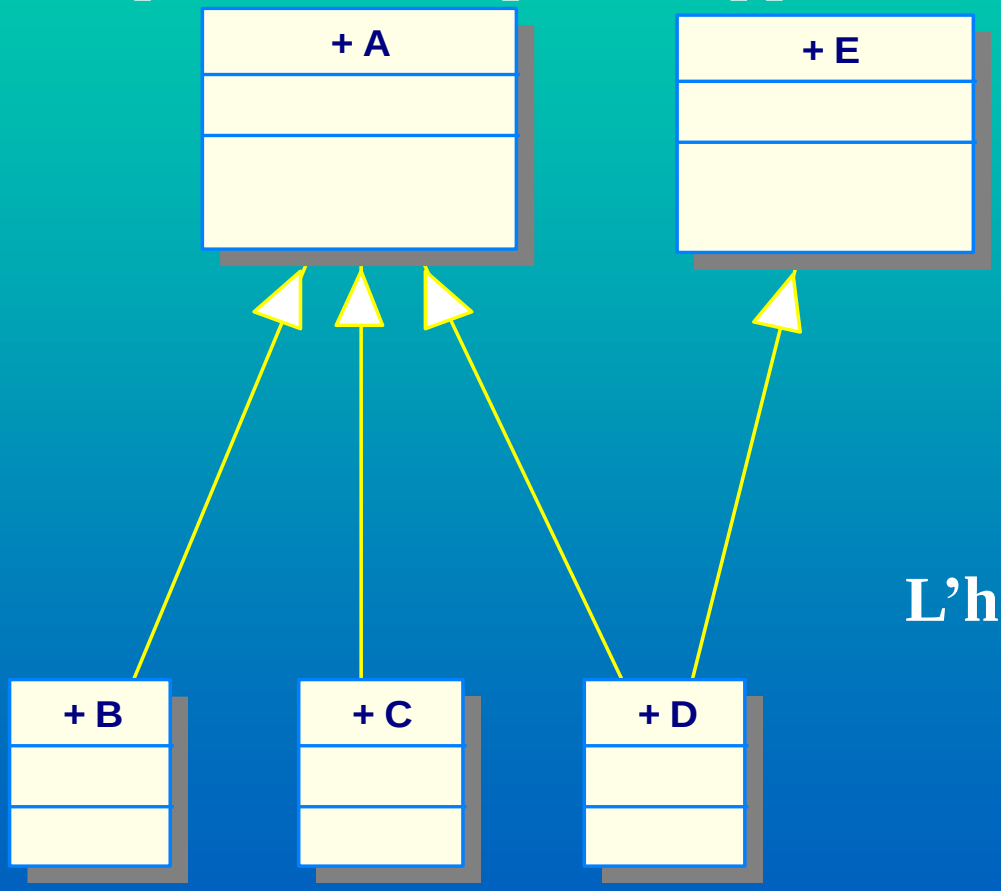
La dépendance est souvent stéréotypée pour mieux expliciter le lien sémantique entre les éléments du modèle :

- **<<friend>> : on accorde une visibilité spéciale à la classe source dans la cible**
- **<<dérive >> : la source est calculée à partir de la cible**
- **<<appel>> : appel d'une opération de la cible à partir de la source**
- **<<instantiate>> : une opération de la source crée un instance de la cible**
- **<<send>> : une opération envoie un signal vers un élément cible**
- **<<trace>> : la cible est une version précédente de la source**
- **<<refine>>**
- **<<copie>> et <<create>>**

2.2 La généralisation (Hiérarchie de classes)

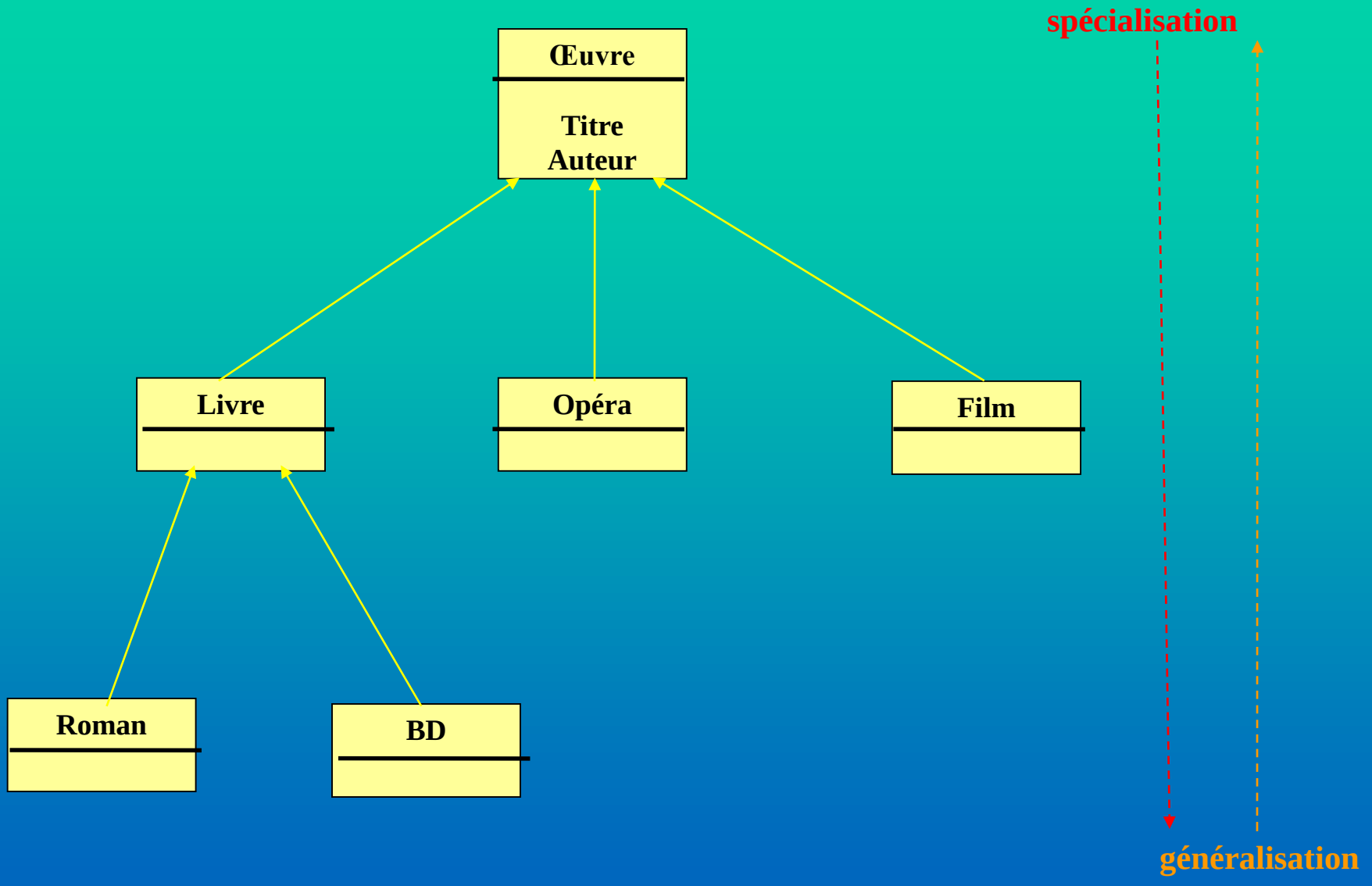
C’est une relation « … est une sorte de … ».

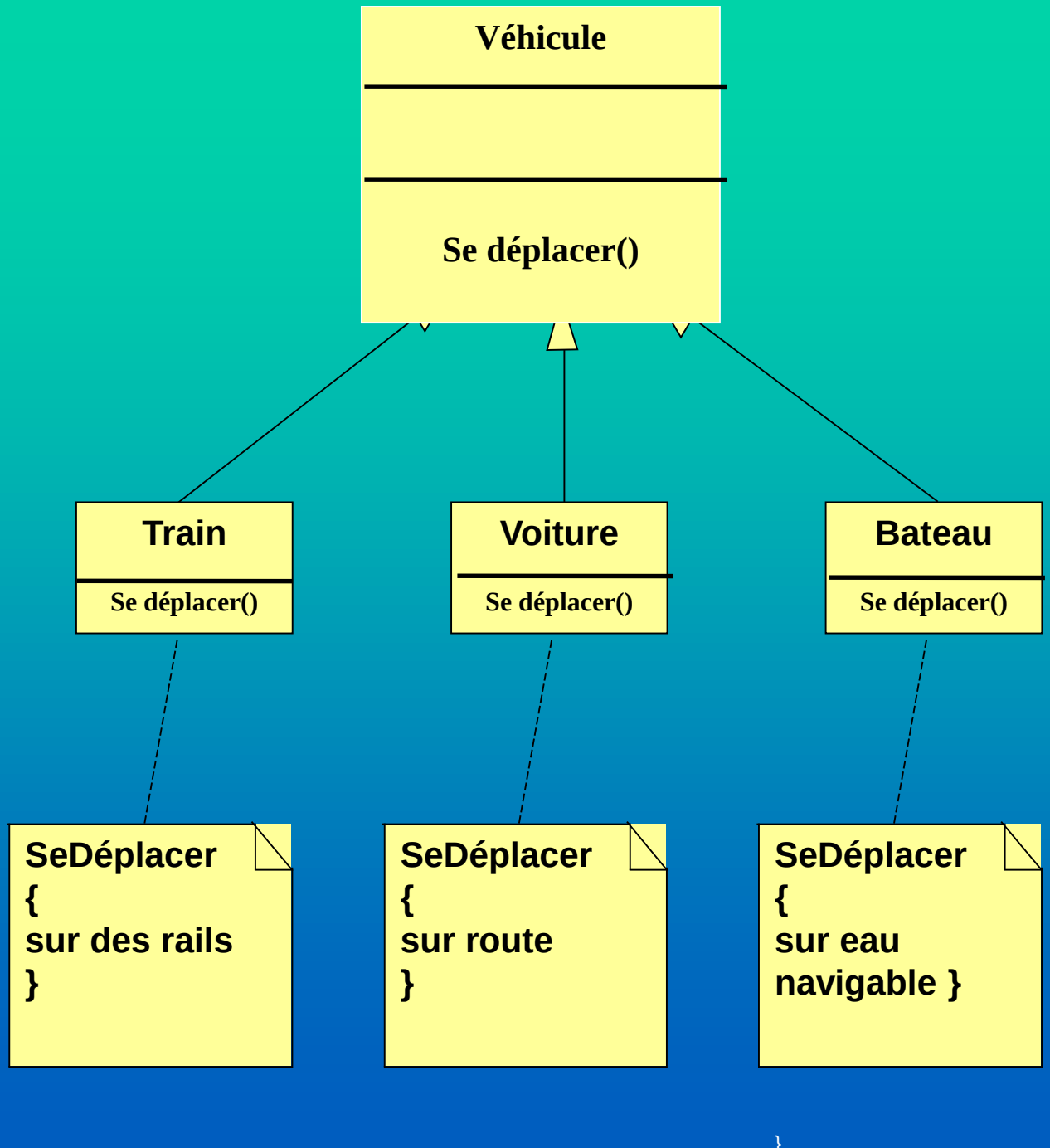
La généralisation implique que des objets de l’enfant puissent être utilisés partout où le parent apparaît mais pas l’inverse.



L’héritage multiple est autorisé

- **l'héritage est un mécanisme de transmission des propriétés d'une classe vers une sous-classe**
- **une classe peut être spécialisée en d'autres classes afin d'y ajouter des caractéristiques spécifiques ou d'en adapter certaines**
- **plusieurs classes peuvent être généralisées en une classe qui les factorise afin de regrouper les caractéristiques communes d'un ensemble de classes**
- **la spécialisation et la généralisation permettent de construire des hiérarchies de classes ; l'héritage peut être simple ou multiple.**
- **l'héritage évite la duplication et encourage la réutilisation**
- **le polymorphisme représente la faculté d'une méthode à pourvoir s'appliquer à des objets de classes différentes**
- **le polymorphisme augmente la généricité du code**


spécialisation
Œuvre
Titre
Auteur
Livre
Opéra
Film
Roman
BD
généralisation


Véhicule
Se déplacer()
Train
Se déplacer()
Voiture
Se déplacer()
Bateau
Se déplacer()
SeDéplacer
{
sur des rails
}
SeDéplacer
{
sur route
}
SeDéplacer
{
sur eau
navigable }

La relation d'héritage est une relation à sens unique. La fille hérite de sa mère qui ne connaît rien de sa fille. Les classes reliées (Mère/Fille) représentent des niveaux d'importance ou d'abstraction différents.

Modéliser des relations d'héritage par généralisation

- **Chercher les responsabilités, les opérations et les attributs qui sont communs à 2 classes ou plus dans un ensemble de classes données**
- **Regrouper ces responsabilités, ces opérations et attributs communs dans une classe plus générale (la créer si nécessaire)**
- **Préciser que les classes plus spécifiques héritent des classes plus générales (tracer la relation de généralisation).**

Cette approche est basée sur le fait que les feuilles (classes filles) appartiennent au monde réel tandis que les niveaux supérieurs sont des abstractions permettant un regroupement de propriétés, opérations, contraintes).

Une superclasse est une abstraction de ses sous-classes.

Modéliser des relations d'héritage par spécialisation

La spécialisation est une technique efficace pour l'extension cohérente d'un ensemble de classes déjà définies afin de prendre en considération des nouveaux besoins. Elle est à la base de la programmation par extension et de la réutilisation. Les sous-classes étendent les fonctions existantes.

On peut spécialiser par extension de propriétés (ajout de propriétés spécifiques) ou par restriction du domaine de valeurs d'un attribut (un carré est un rectangle dont la longueur = la largeur)

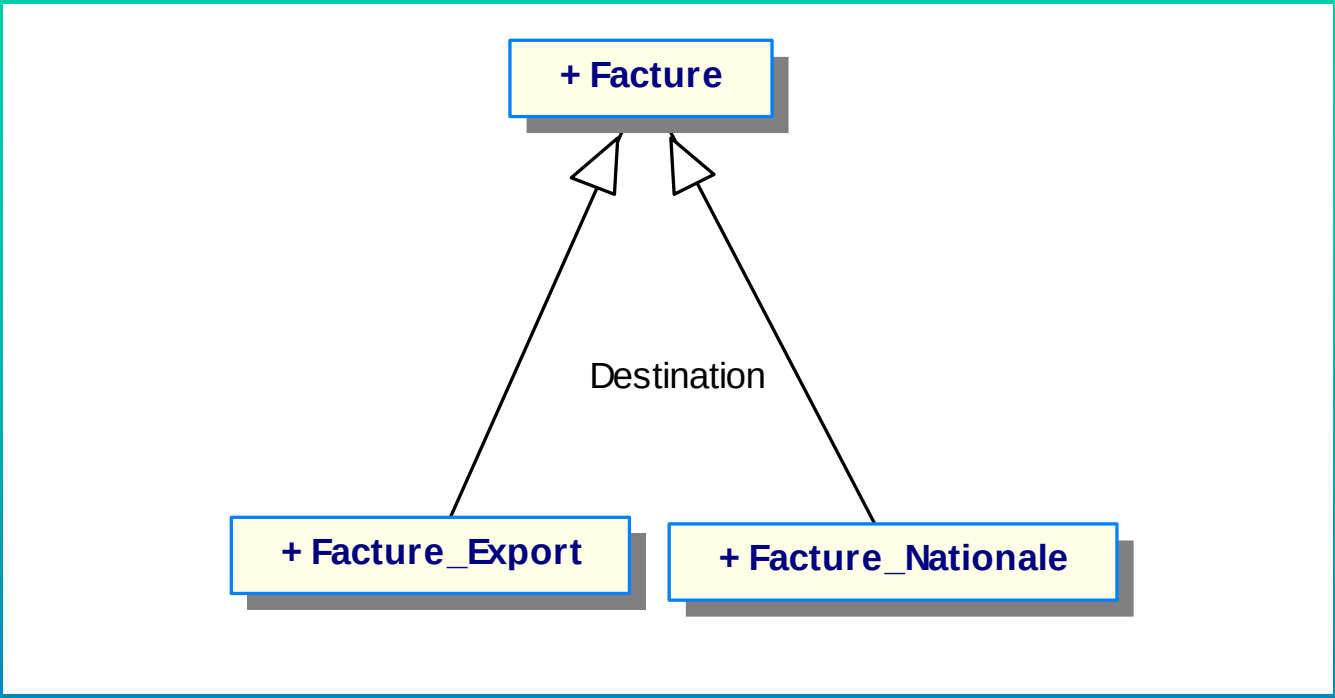
La sous-classe définit une portée plus spécifique de l'attribut

Au niveau de la modélisation UML, on parle en fait de spécialisation et de généralisation pour évoquer des hiérarchies de classes.

L'héritage, à proprement parlé, est une technique de programmation permettant la mise en œuvre des hiérarchies de classes.

L'héritage est souvent opposé à la délégation qui permet également d'appliquer le mécanisme de réutilisation (voir plus loin).

Le **discriminant** (absent par défaut) est le critère selon lequel la spécialisation est faite. C'est un attribut de la spécialisation dont les valeurs partitionnent la super classe en sous-classes.



Lorsque le nombre de sous-classes définies est très grand, on peut se dispenser de toutes les montrer. Utiliser la notation **...**
C'est différent d'utiliser la contrainte {incomplète} qui précise que toutes les sous-classes ne sont pas définies. La spécialisation pourra être étendue.

2.3 L'association

C'est une relation structurelle qui précise que les objets d'un élément sont reliés aux objets d'un autre élément.

Une association exprime une connexion sémantique bidirectionnelle (par défaut) entre deux classes. Elle est instanciable dans un diagramme d'objets ou de collaboration sous forme de liens entre objets issus de classes associées.

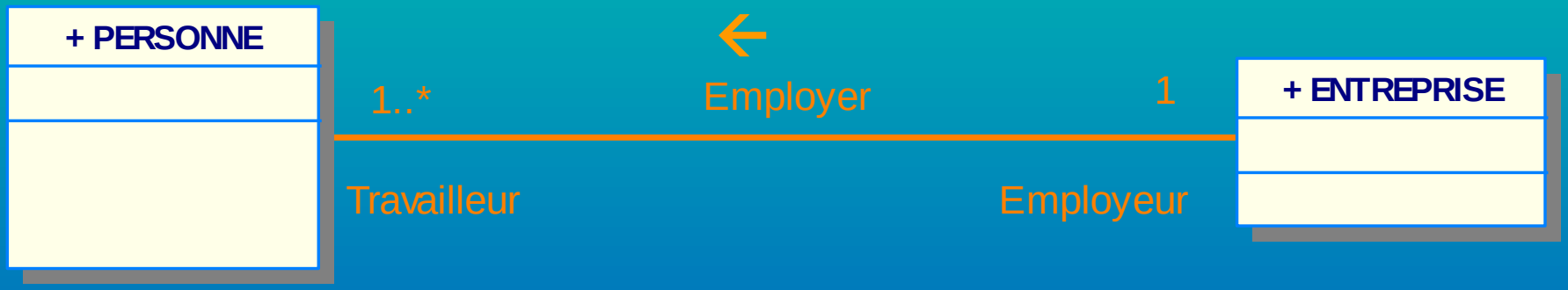
Une association est simplement représentée par une ligne pleine entre classes. On peut décorer cette ligne pleine.

Une simple association n'est pas transitive !

Une relation peut être binaire ou « n-aire »

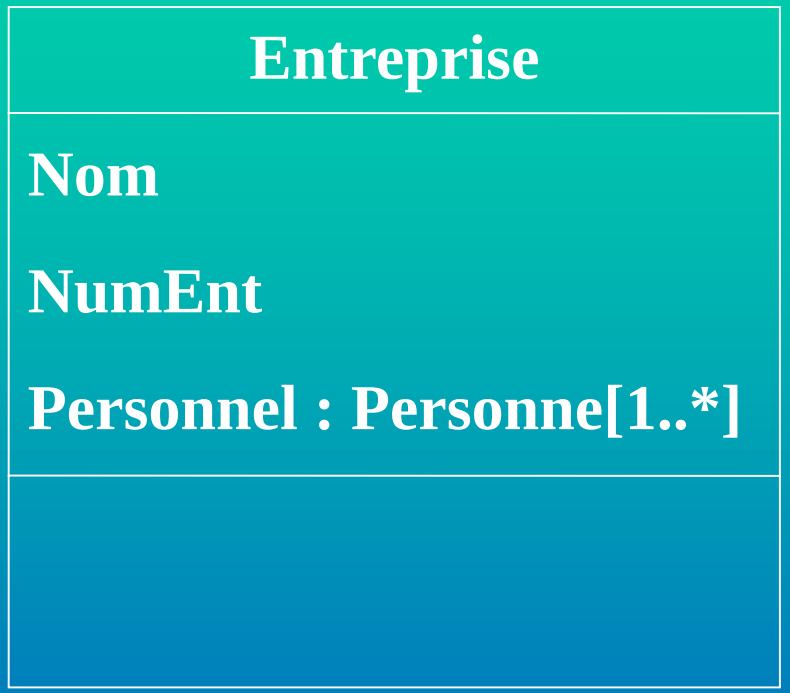
On peut ajouter :

- **un nom et une direction**
- **des rôles aux extrémités (rôles joués par les classes)**
- **des multiplicités qui définissent combien d'objets peuvent être reliés par l'intermédiaire d'une instance d'association**



L'association est complétée par un verbe à l'infinitif avec une précision du sens de lecture si ambiguïté . Indiquer → ou ← à coté du verbe utilisé.

Il est possible de représenter une association sous forme d'un attribut dans la classe. C'est un choix de modélisation.

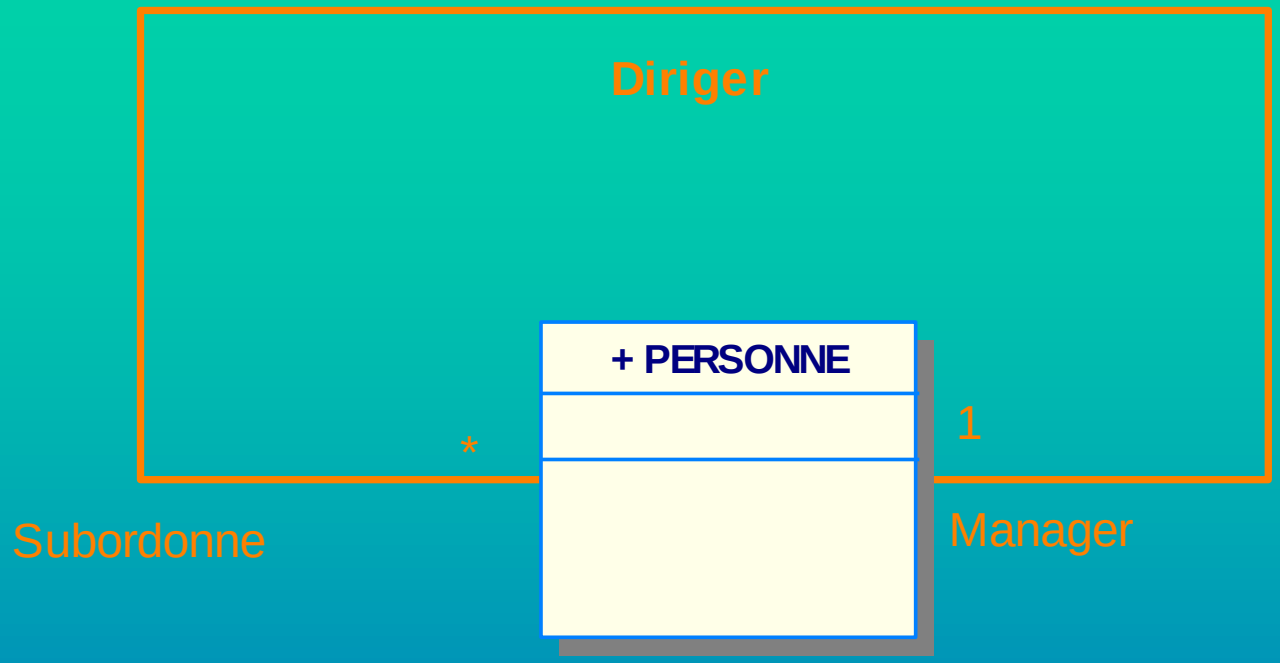


L'association n'est pas modélisée explicitement. Personne n'apparaît pas comme une classe mais comme un type connu. A éviter !!!

La multiplicité définit combien d'objets peuvent être reliés par l'intermédiaire d'une instance d'association :

- **1**
- *** équivaut à 0..***
- **0..1**
- **0..***
- **1..***
- **n**
- **n..m où n et m représentent une valeur constante exacte**
- **Liste de la forme : 0..1, 10..20, 50..***

Une association peut être réflexive : elle relie des instances de la même classe.



Une association réflexive peut être symétrique/asymétrique et transitive/non transitive.

Diriger est une association asymétrique et transitive

Avoir le même âge est une association symétrique et transitive

Etre amie de est une association symétrique et non transitive

L’association réflexive permet de structurer les objets d’une même classe.

Une association asymétrique permet de créer une hiérarchie entre les objets. Dans le cas contraire, elle partitionne les instances de la même classe.

Quand l’association est à la fois symétrique et transitive, les objets sont regroupés en groupes d’équivalence.

Une association simple entre deux classes représente une relation structurelle entre classes de même niveau. Aucune n'est plus importante que l'autre. L'association exprime une relation à couplage faible. Chaque classe reste relativement indépendante.

L'agrégation est un cas particulier d'association qui exprime un couplage plus fort entre classes. Elle est purement conceptuelle.

Elle indique qu'une des classes joue un rôle plus important que l'autre dans l'association. C'est une connexion bidirectionnelle dissymétrique permettant de mettre en évidence un TOUT et ses parties.

Elle représentée graphiquement par un losange vide dessiné du coté du TOUT.

L’agrégation ne change pas la sémantique de la navigabilité d’une association entre le tout et ses parties.

L’agrégation ne modifie pas non plus la sémantique de la durée de vie du tout et de ses parties. Ils peuvent exister l’un sans l’autre !

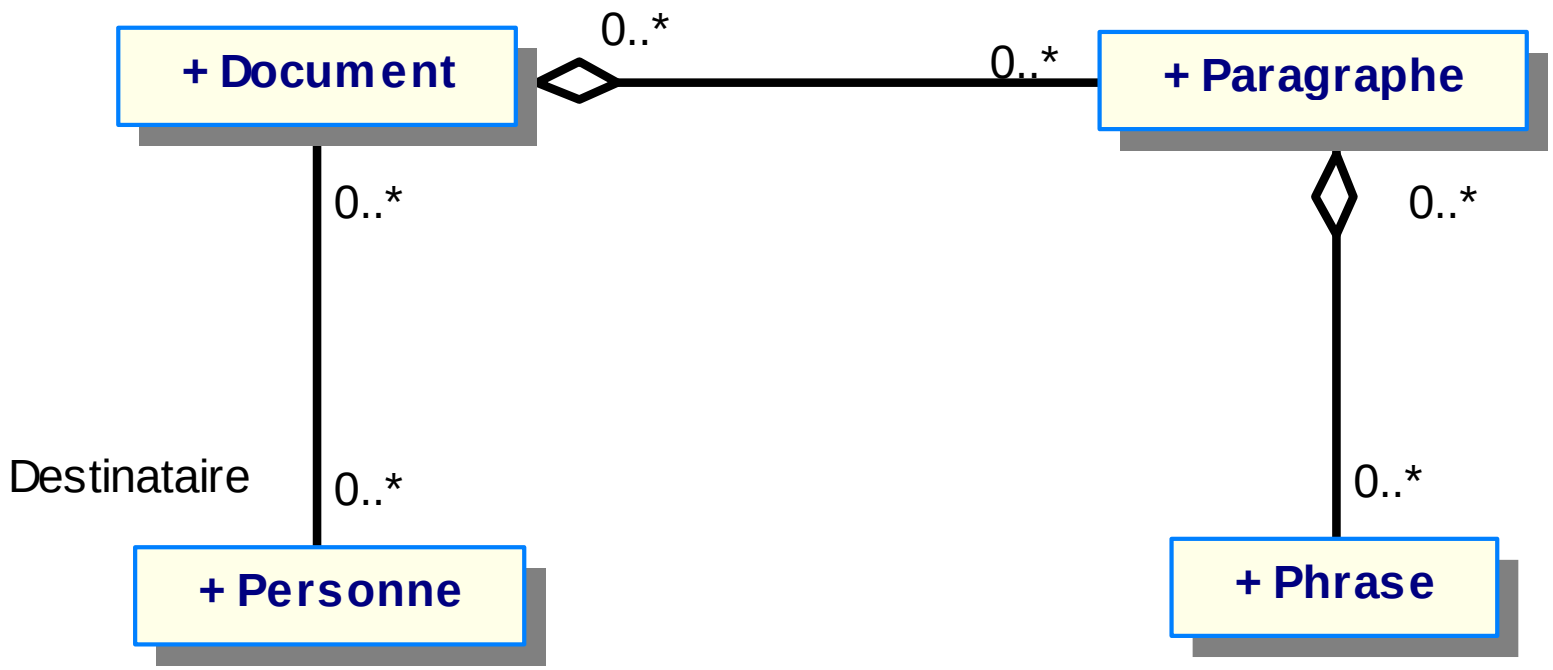
L’agrégation est une relation transitive, non symétrique et réflexive à couplage plus fort que la simple association.

L’agrégation permet de modéliser une contrainte d’intégrité dont l’agrégat (classe coté losange) est le garant.

Une agrégation permet de représenter (modéliser), par exemple :

- **la propagation des valeurs d’attributs d’une classe vers une autre classe**
- **une action sur une classe qui implique une action sur une autre classe**
- **une subordination des objets d’une classe à ceux d’une autre.**

La notion d’agrégation ne suppose aucune forme de réalisation particulière.



Propagation de valeurs du tout vers les parties

On peut également utiliser la délégation d'opérations : Document.Copier() peut être déléguée à l'opération Paragraphe.Copier() qui sera déléguée à Phrase.Copier().

Distinction Agrégation / Association :

Les questions à se poser sont :

- **Y a t'il une asymétrie intrinsèque dans l'association qui se manifeste par un lien de subordination entre les instances des deux classes (agrégation) ?**
- **Au contraire, les objets sont-ils plutôt indépendants les uns des autres (association) ?**
- **Y a t'il des opérations du tout qui s'appliquent automatiquement aux parties (Document.Copier()) ?**
- **Peut-on dire « X est partie de Y » ?**
- **Y-a-t-il des valeurs d'attributs du tout qui se propagent vers les parties (Ex. : Couleur de fond du Document qui se propage aux paragraphes qui le composent)?**

Dans l’exemple précédent, on voit bien que Personne ne fait pas partie de Document (pourtant un attribut de Document peut se propager vers Personne).

La composition (agrégation composite) est un cas particulier de l'agrégation.

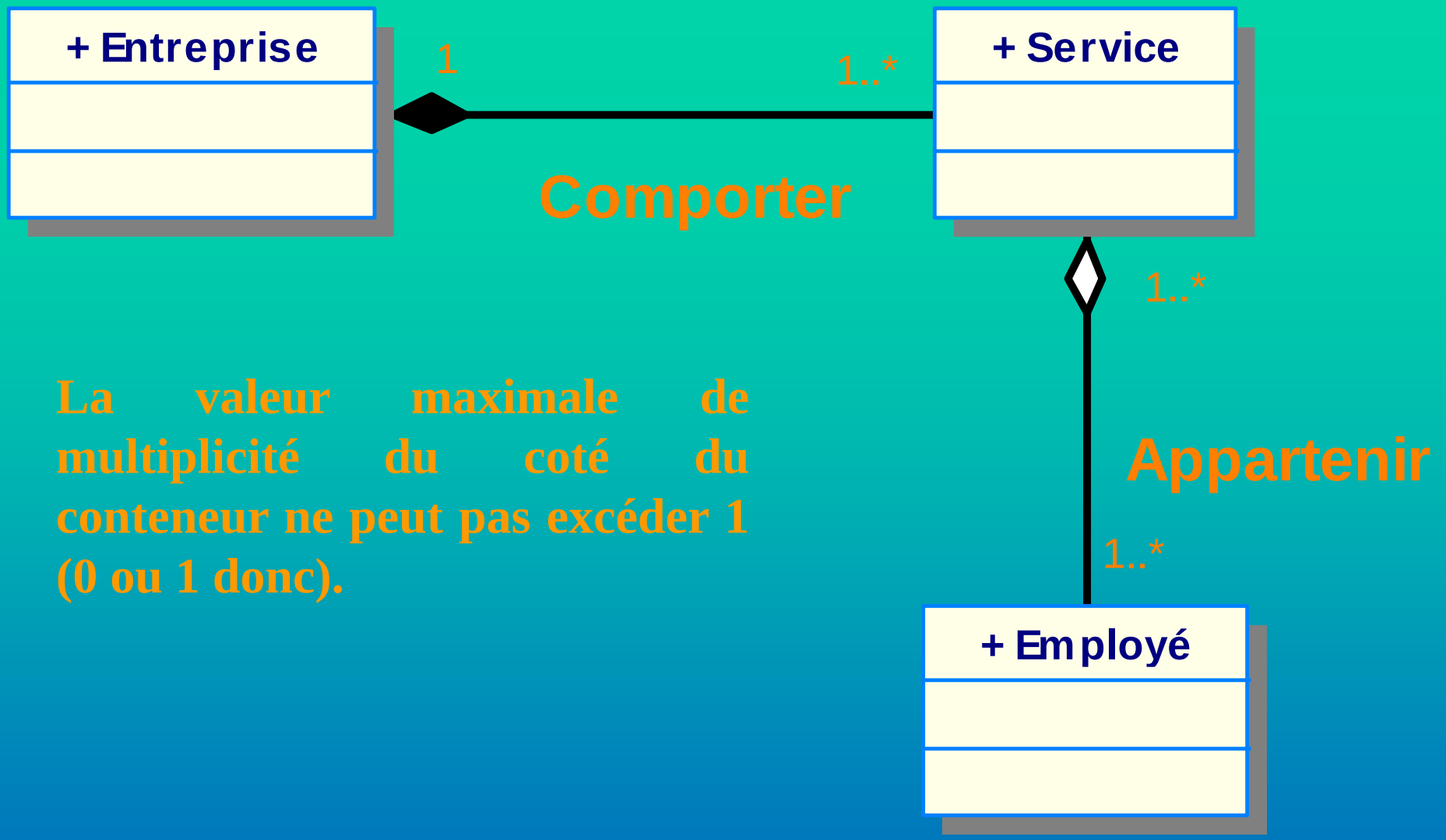
La composition ajoute des règles sémantiques :

- **mise en évidence d'une notion de propriété forte**
- **mise en avant d'une coïncidence des cycles de vie.**

Les parties, dont la multiplicité n'est pas fixée (*), peuvent êtres créées après le composite mais vivent et meurent avec lui. Elles peuvent être supprimées avant la mort du composite mais ne peuvent pas leur survivre.

Dans une composition, un objet ne peut faire partie que d'un seul composite à la fois.

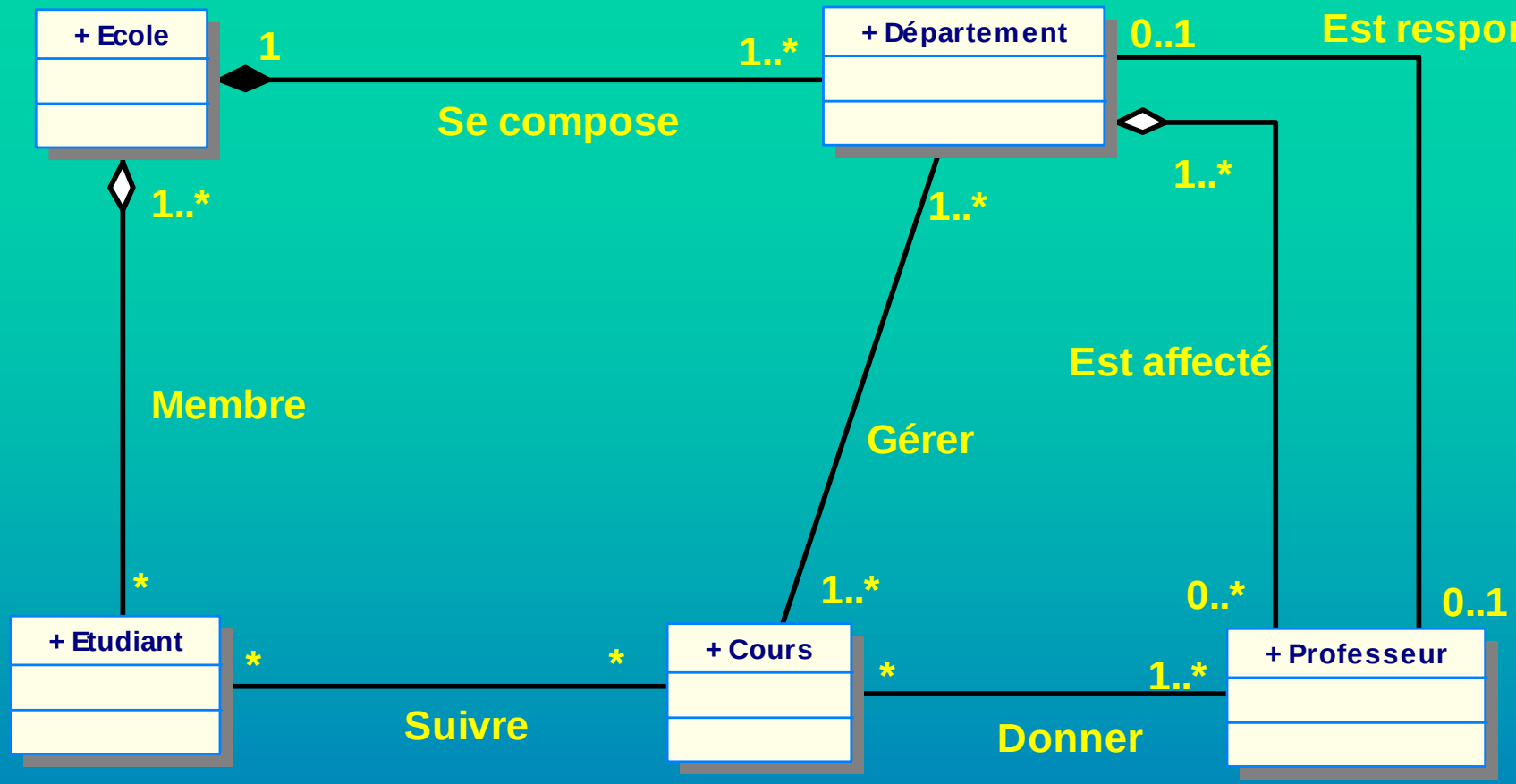
La composition est la relation qui lie une classe à ses attributs. Elle peut être vue comme une collaboration entre un tout et ses parties.



La valeur maximale de multiplicité du coté du conteneur ne peut pas excéder 1 (0 ou 1 donc).

Le composite est également responsable de la mise à disposition de ses parties. Il doit gérer la création et la destruction de ses parties.

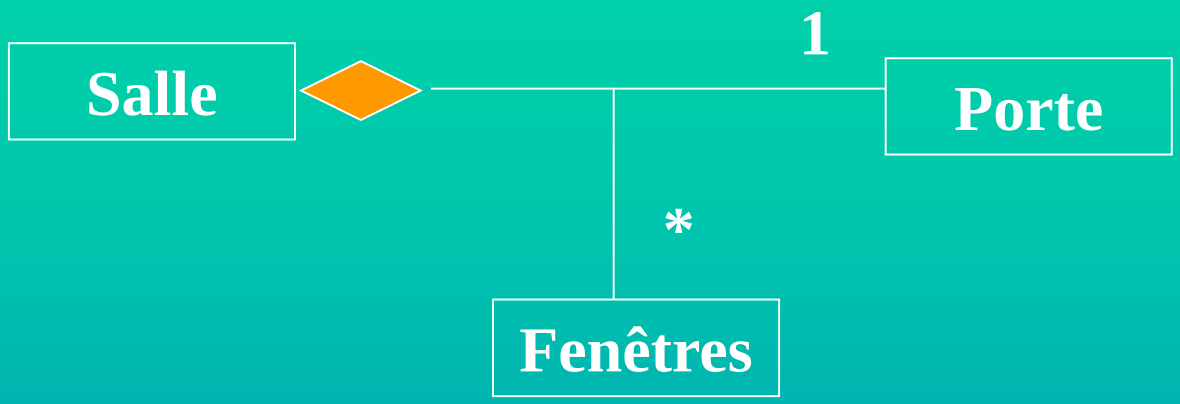
(→ transformation d'une composotion au niveau du MLD !)

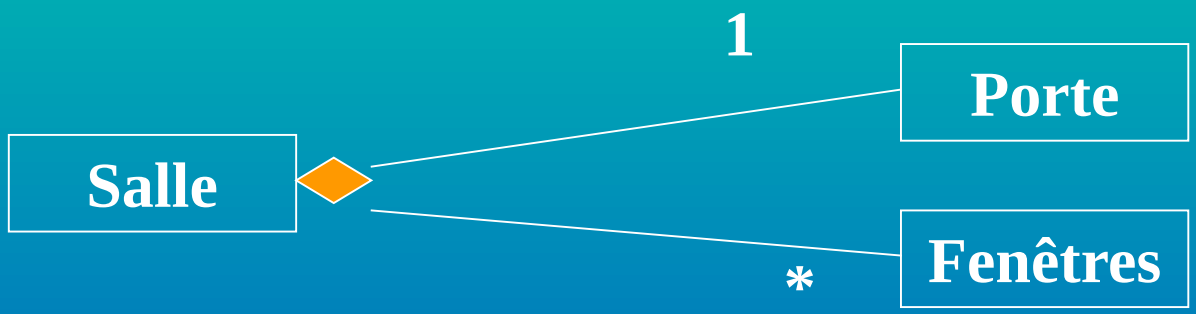


• On utilise deux agrégations pour visualiser qu'une Ecole contient des Etudiants, et un Département des Professeurs.

• Entre Ecole et Département, il s'agit d'une composition (inclusion physique). A un département ne peut correspondre qu'une seule Ecole.

La relation de composition étant structurelle, le langage UML autorise les représentation suivantes :



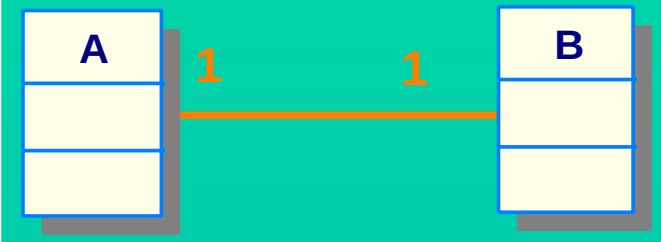


Les valeurs choisies pour les multiplicités expriment des contraintes qui influencent la création et la destruction des objets.

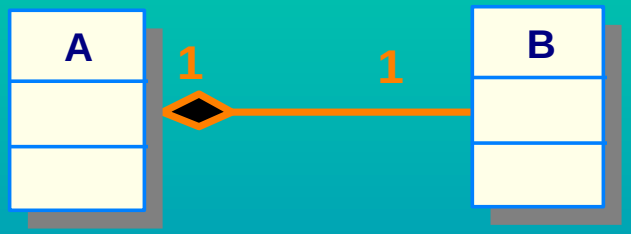
On peut imposer de respecter les multiplicités :

- **soit de façon statique : les multiplicités doivent être vérifiées à tout moment (régime transitoire comme régime normal)**

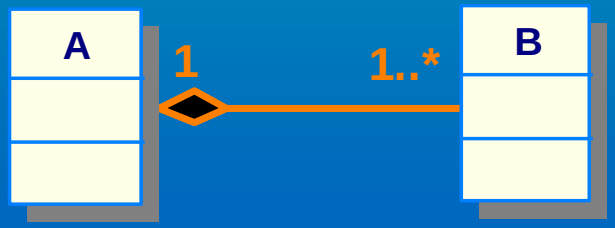
- **soit de façon dynamique : les multiplicités peuvent ne pas être respectées durant des phases transitoires.**



La création d'un objet A requiert un objet B et vice versa. La destruction d'une instance de B ou A est possible si une nouvelle instance remplace l'objet A ou B avec une instance de la classe opposée (multiplicité vérifiée en régime établi).



La création d'un objet A requiert un objet B et vice versa. La destruction d'une instance de A requiert la destruction de l'objet B composé. La destruction d'un objet B est possible si une nouvelle instance remplace l'objet B avec une instance de type A (multiplicité vérifiée en régime établi).



La création d'un objet A requiert un objet B et vice versa. La destruction d'une instance de A requiert la destruction de tous les objets B composés. La destruction d'un objet B est possible si une nouvelle instance remplace l'objet B avec une instance de type A (multiplicité vérifiée en régime établi) ou s'il ne s'agit pas de la dernière instance de B associée à un objet A donné.

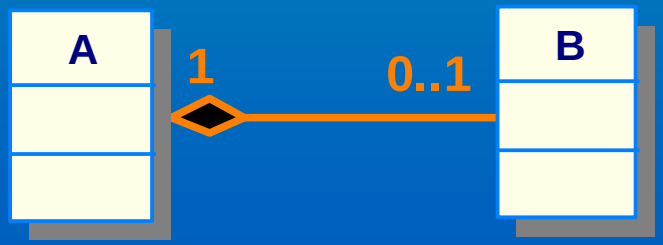
La création d'un objet A requiert un objet B.

La destruction d'une instance de B est possible si une nouvelle instance remplace l'objet B (mult. Vérifiée en régime établi) ou si l'objet B n'est pas associé à un objet A donné.

La création d'un objet A requiert un objet B.

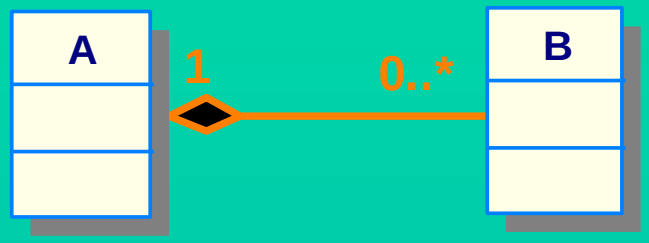
La destruction d'un objet A implique celle de l'objet B composé.

La destruction d'une instance de B est possible si une nouvelle instance remplace l'objet B (mult. Vérifiée en régime établi) ou si l'objet B n'est pas associé à un objet A donné.



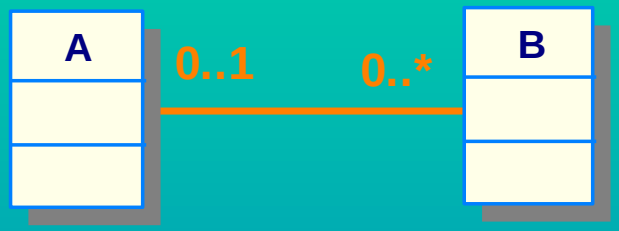
La création d'un objet B requiert un objet A.

La destruction d'une instance de A est possible et implique la destruction de l'objet B associé s'il existe, à moins qu'une nouvelle instance remplace cet objet A dans sa relation avec un objet B



La création d'un objet B requiert un objet A

La destruction d'un objet A implique la destruction de tous les objets B associés à A



Aucune condition

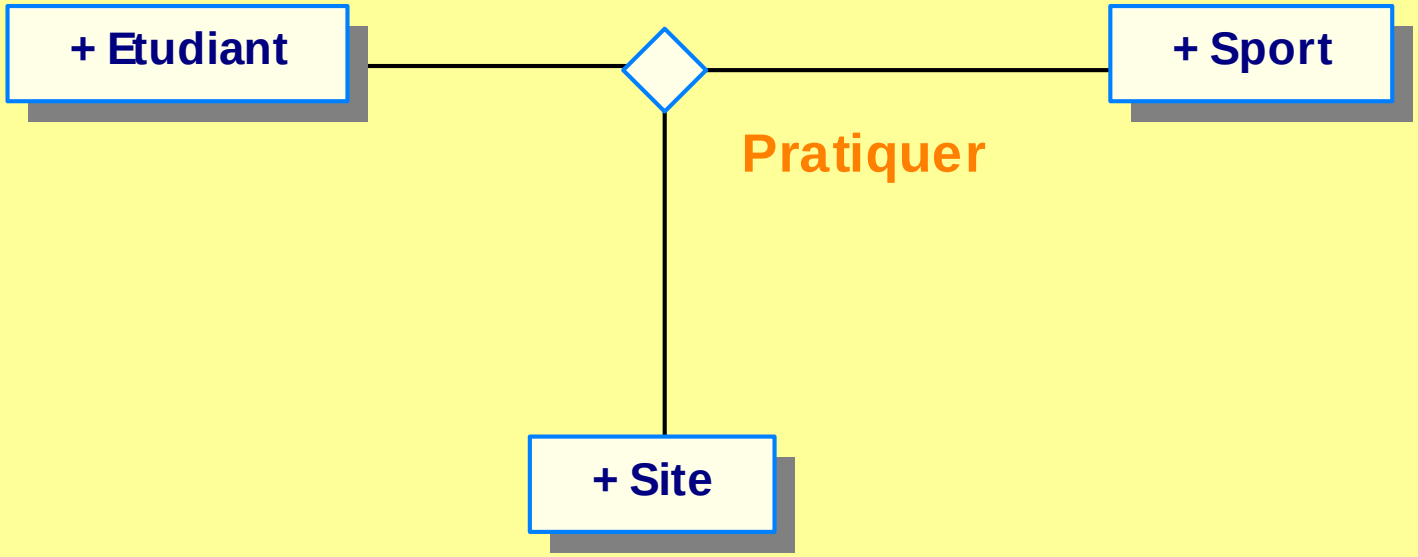
La destruction d'un objet A implique la destruction de tous les objets B associés à A

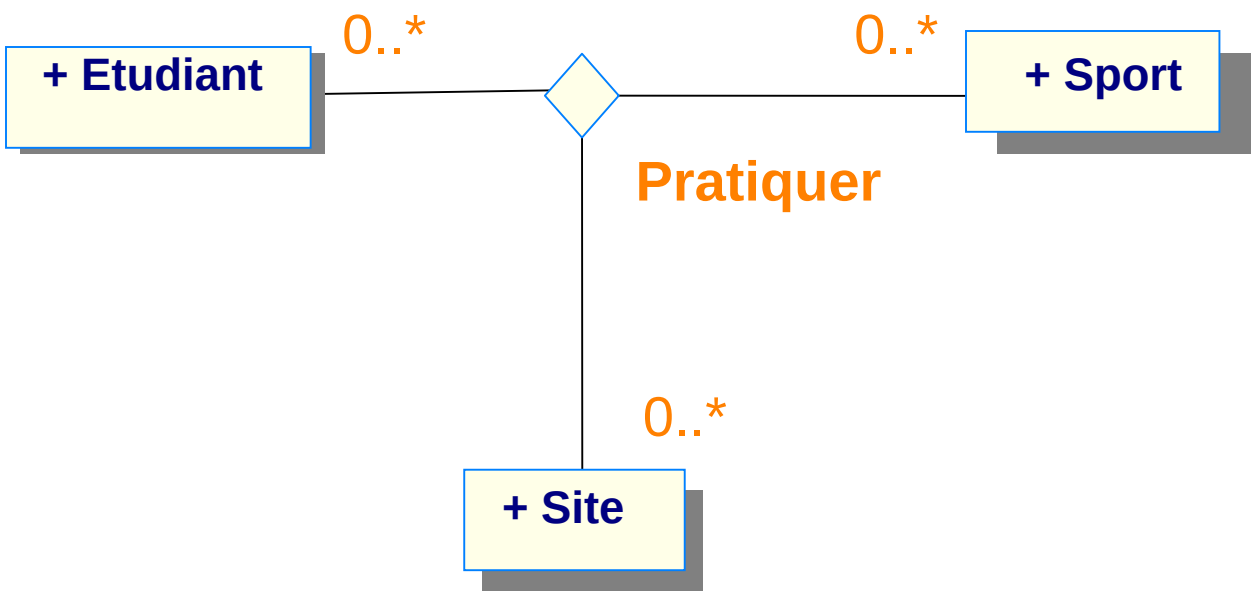
Aucune condition

Associations N-Aire

Généralement les associations sont binaires(elles relient deux classes).

Des arités supérieures (relations ternaires par ex) peuvent exister et se représentent graphiquement par un losange qui connecte les différentes branches de l ’association.




+ Etudiant
0..*
0..*
+ Sport
Pratiquer
0..*
+ Site

La signification des multiplicités change dans le cas d'associations n-aires. Dans l'exemple précédent, pour une paire d'instances (Etudiant x Site), il peut y avoir de 0 à plusieurs Sports

<u>Exercice</u>

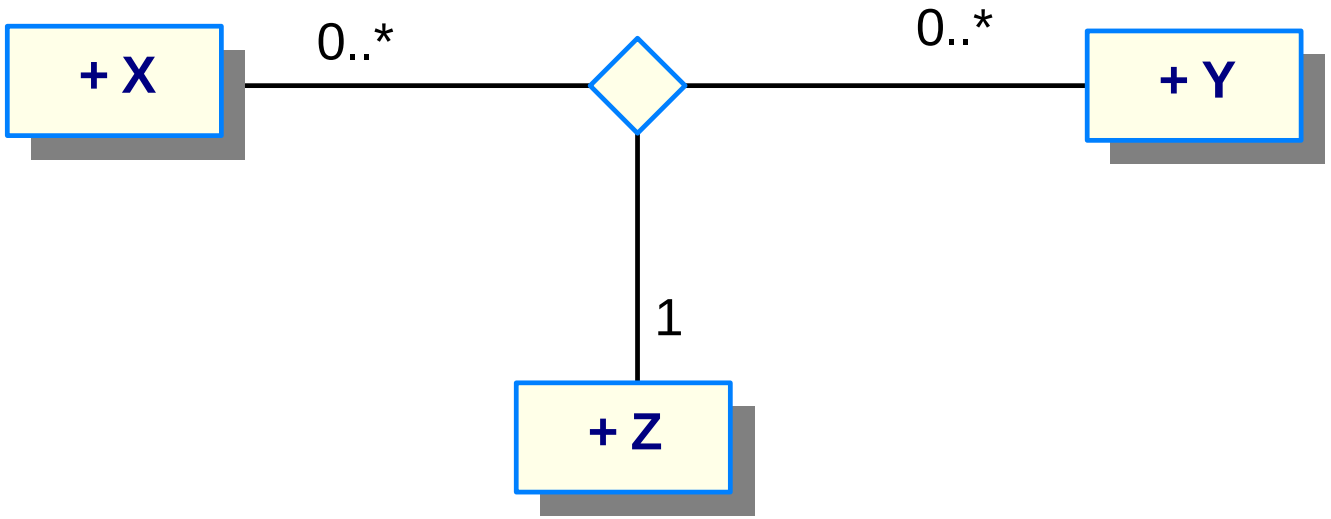
Exprimer :

• Sur un même site, un étudiant ne peut pratiquer qu'un seul sport, mais il peut pratiquer différents sports sur d'autres sites

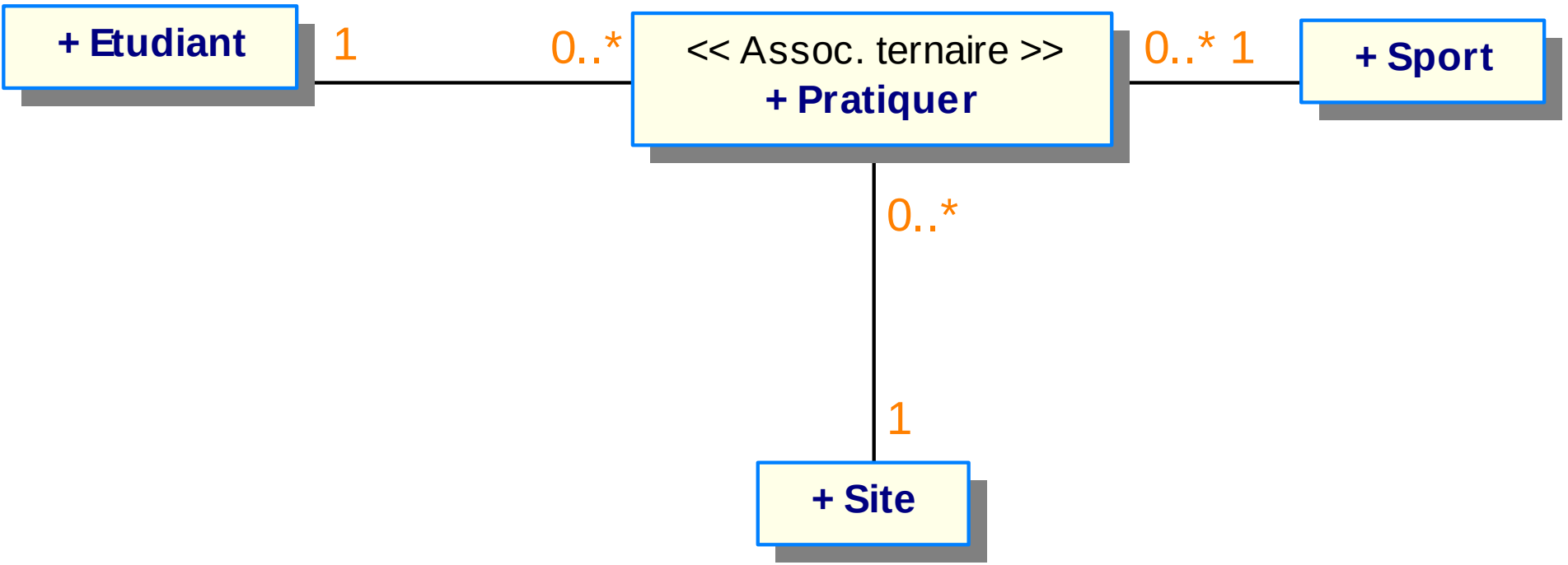
• Un étudiant ne peut jamais pratiquer qu'un seul sport

En fait, quand sur un diagramme de classes une relation n-aire reste modélisée, c'est qu 'elle est non décomposable par une CIF. UML ne permet de modéliser explicitement une CIF.

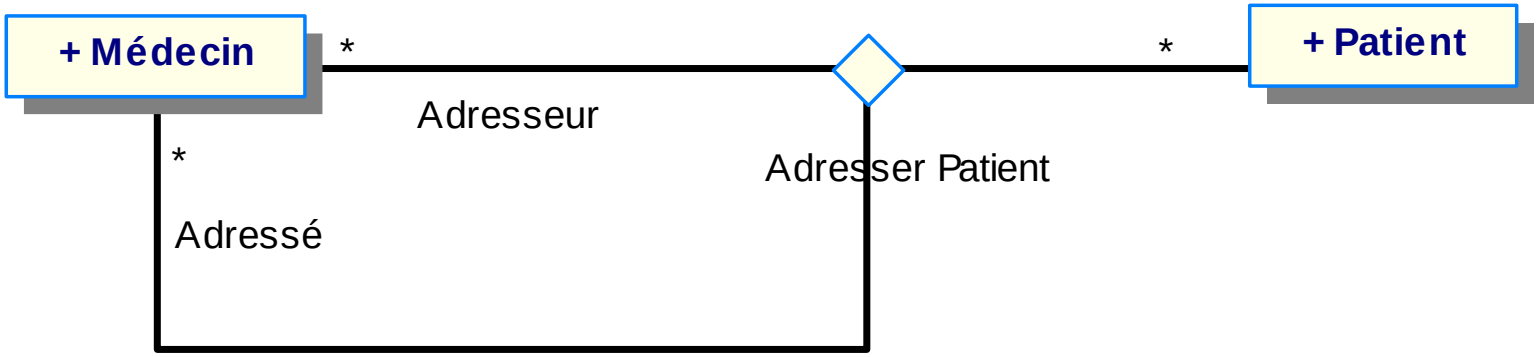
CIF implicite non décomposable : à un couple d'instances X-Y correspond 1 et 1 seule instance de Z.



On peut transformer une association n-aire en changeant l ’association en classe et en ajoutant une contrainte qui exprime que les multiples branches de l ’association s ’instancient simultanément (par un stéréotype par exemple).



Association d’instances d’une même classe :

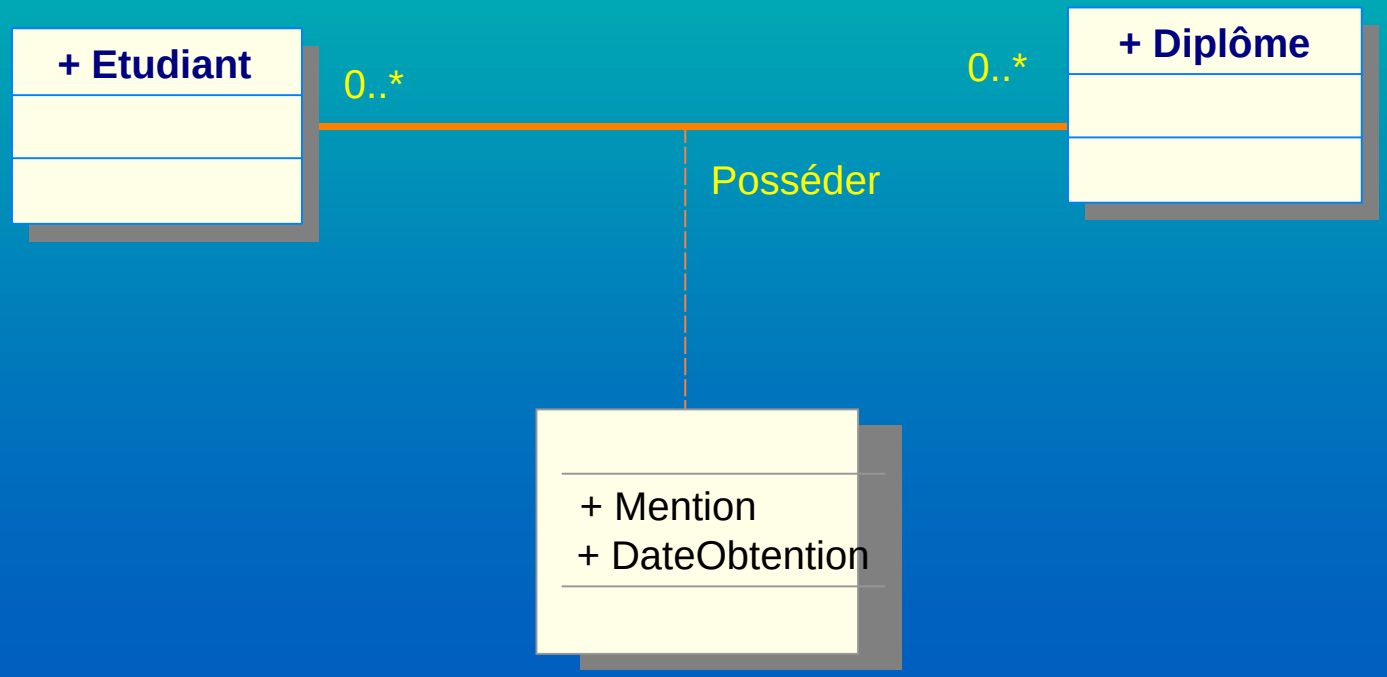


2.4 Les classes d'association

Certaines propriétés ne caractérisent pas des objets mais bien des associations entre objets.

Ces propriétés ne peuvent en principe donc pas appartenir à une classe puisqu'elles ne décrivent pas un objet du monde réel. Elles ne peuvent pas exister en dehors d'une instance d'association.

Elles forment donc une classe d'association.



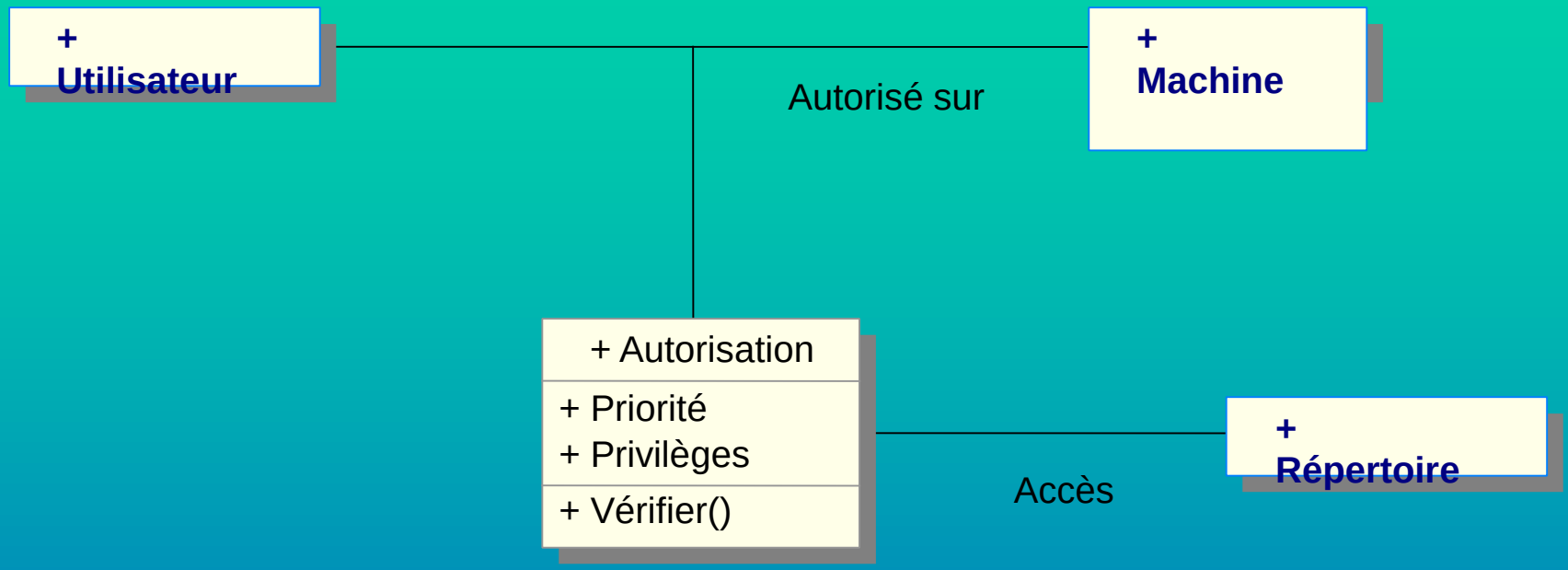
Les propriétés Mention et DateObtention sont en dépendance fonctionnelle à la fois de la classe Etudiant et de la classe Diplôme.

Elles prennent un sens uniquement en présence d'une instance de la classe Etudiant ET d'une instance de la classe Diplôme. Elles caractérisent leur association.

Associer ces propriétés à la classe Diplôme, signifierait qu'une instance spécifique a toujours été obtenue à la même date et avec la même mention quel que soit l'étudiant qui a obtenu ce type de diplôme.

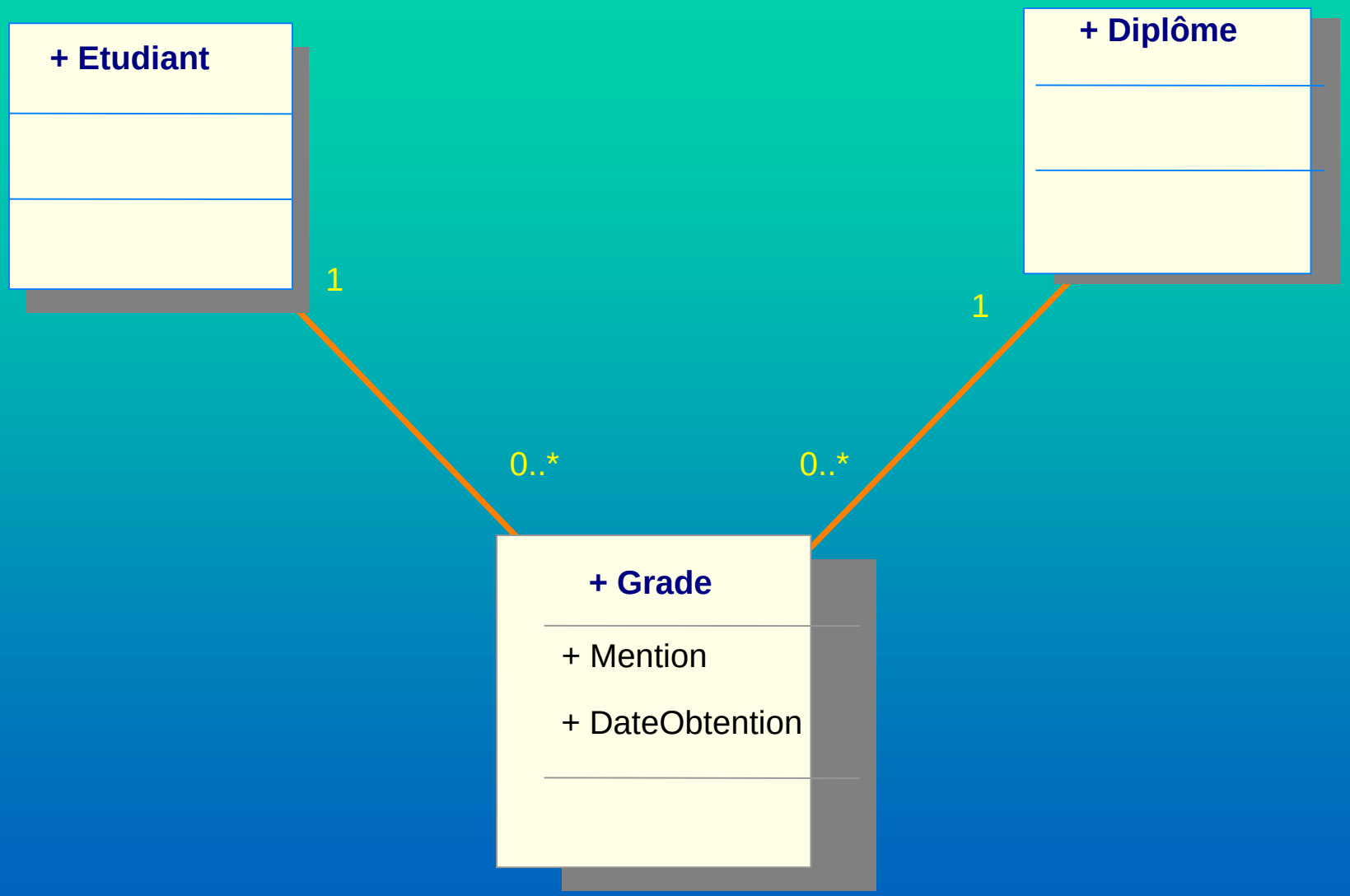
De même, les associer à la classe Etudiant, signifierait qu'un étudiant bien précis a obtenu tous ses diplômes à la même date et avec la même mention.

Une classe-association possède donc à la fois les caractéristiques d ’une classe et d ’une association. Elle peut donc participer à d ’autres relations dans le modèle.

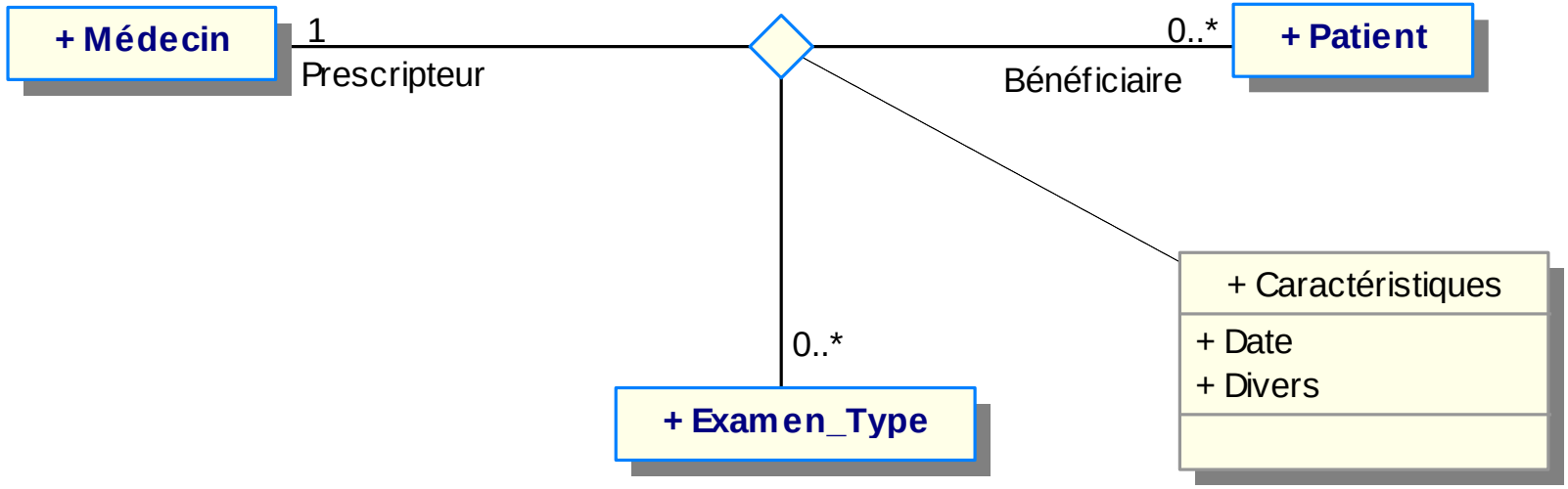


Une association qui contient des attributs sans participer à des relations avec d ’autres classes est appelée association attribuée. La classe-association ne porte alors pas de nom.

En conception, une classe-association peut-être remplacée par une classe intermédiaire.

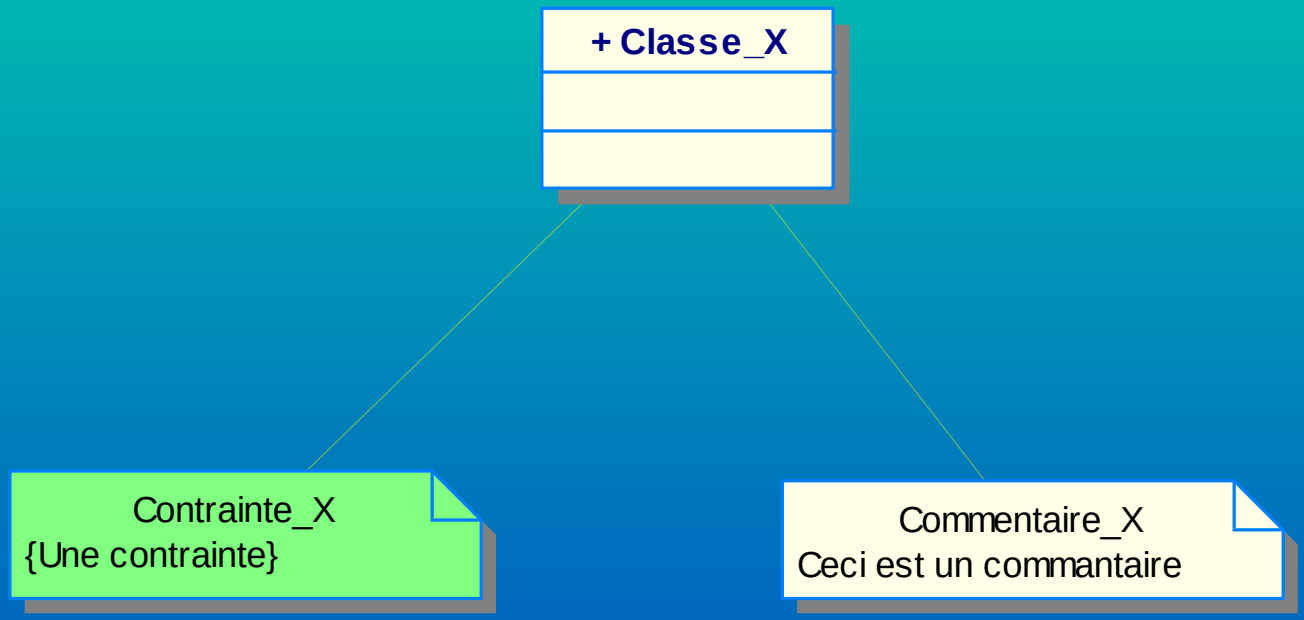


Classe association et association ternaire :



3 Les notes

- Ce sont les décorations les plus importantes d'UML.
- Elles consistent en symboles graphiques utilisés pour représenter les contraintes et les commentaires rattachés à 1 ou plusieurs éléments. Elles ajoutent de l'information à un modèle.
- Une note (qui n 'est pas une contrainte) n'a aucune conséquence sémantique.



Le contenu d'un commentaire peut être quelconque (texte, URL, icône, …)

• **Une note peut contenir une contrainte ({…….}) qui représente une obligation ou une responsabilité. Elle modifie alors la sémantique.**

• **Une note peut aussi être utilisée pour indiquer l'évolution de la modélisation.**

• **En UML, il faut d'abord employer la notation de base de chaque élément et n'ajouter des décorations que si elles sont vraiment nécessaires (informations spécifiques et importantes pour le modèle).**

4 Les stéréotypes

Ils étendent le vocabulaire UML en permettant la création de nouvelles sortes de briques de base spécifiques à un problème donné.

Il existe des stéréotypes standard.

Un stéréotype est représenté par un nom mis entre les caractères << et >> et placé au dessus du nom de l'élément stéréotypé (qui peut également être représenté par une nouvelle icône).

Un stéréotype est plutôt un métatype car il crée l'équivalent d'une nouvelle classe dans le modèle d'UML.

Créer un stéréotype c'est créer une nouvelle brique UML (on étend UML) possédant des propriétés, une sémantique et une notation spécifiques.

Si on crée le stéréotype <<Travail>> propre à une application, il ne s'agit pas de créer une classe Travail avec la notation UML courante qui correspond à un ensemble d'instances (objets) mais bien de créer une sorte de type de classe.

<< Travail >>
+ Terrassement

<< Travail >>
+ Fondation

On peut représenter un stéréotype par une simple icône, par une forme géométrique quelconque, par une forme géométrique + icône.

Le concept de stéréotype est le mécanisme d ’extensibilité le plus puissant d ’UML.

Un stéréotype introduit une nouvelle classe dans le métamodèle (et pas le modèle) par dérivation d ’un classificataeur existant (la classe par ex.).

Les utilisteurs d ’UML peuvent ainsi ajouter de nouvelles classes d ’éléments de modélisation au métamodèle en plus du noyau prédéfini d ’UML

5 Les étiquettes

Une étiquette étend les propriétés d'un élément UML. On enrichit sa spécification avec de nouvelles informations.

Elle est représentée par une chaîne de caractères entre accolades placée au dessous du nom de l'élément auquel elle est associée.

Une étiquette permet d'associer de nouvelles propriétés à un type d'objet. Exemples : Status, Version, Auteur, Langage cible, …

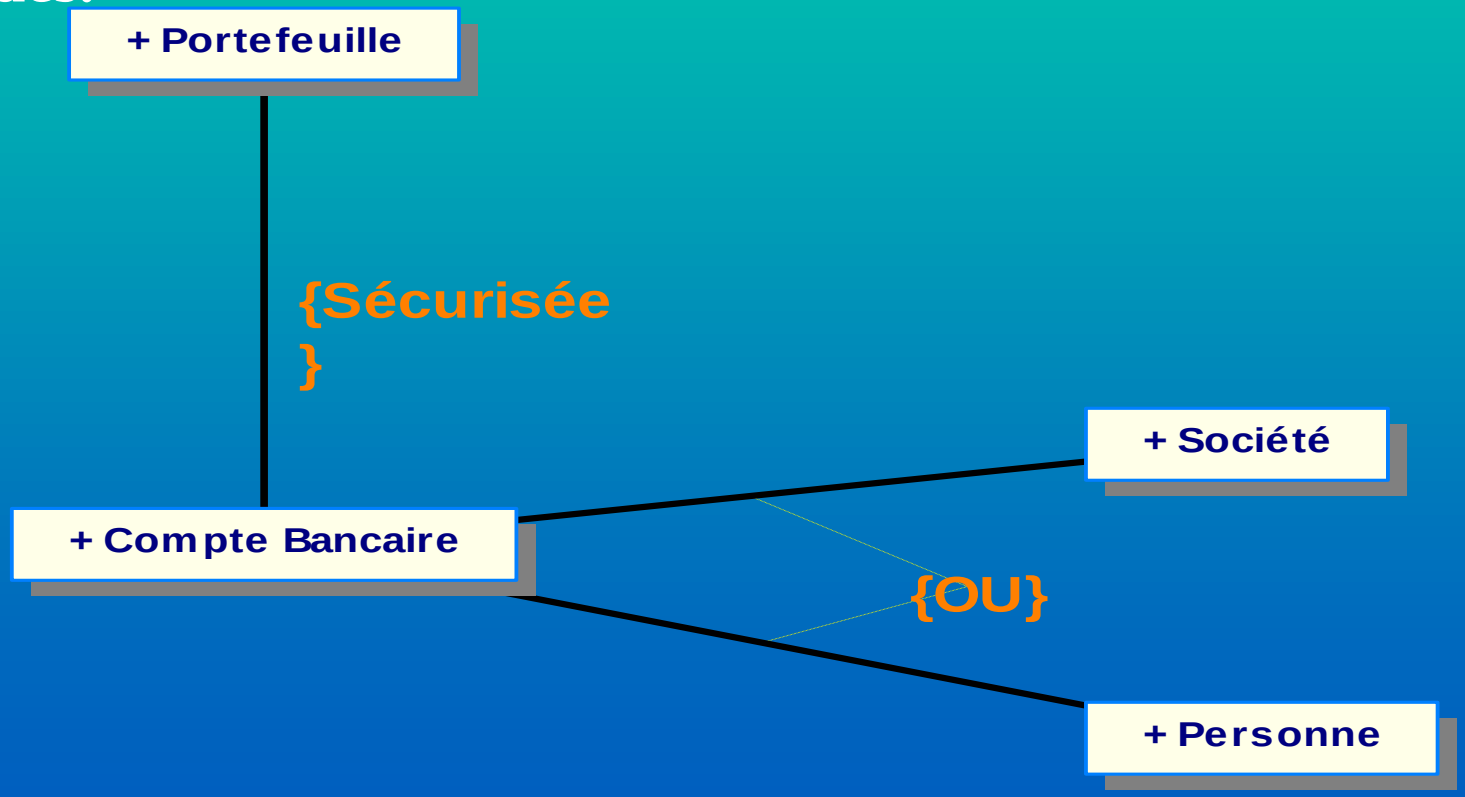
Il ne s'agit pas d'un attribut dont chaque instance aura sa propre valeur mais plutôt d'une métadonnée dont la valeur s'applique au type d'objet (la classe par ex.) et pas à ses instances (auquel cas on utilise des attributs)

6 Les contraintes

Elles permettent d'ajouter de nouvelles règles sémantiques ou de modifier celles qui existent.

Il existe des contraintes standard.

Elles sont représentées par une chaîne de caractères entre accolades.



Les diagrammes de classe

1 Principes

Ce sont les diagrammes les plus courants dans la modélisation O.O.

Ils sont constitués d'un ensemble de classes, d'interfaces, de collaborations ainsi que de leurs relations (dépendances, généralisations et associations).

On peut également ajouter des notes et des contraintes. Ils peuvent contenir des packages ou des sous-systèmes.

Ils permettent de :

- modéliser la vue de conception statique d'un système
- modéliser des collaborations
- modéliser des schémas

La vue de conception statique d'un système consiste à modéliser les services que doit fournir le système à ses utilisateurs.

Modéliser le vocabulaire consiste à préciser les abstractions du système ainsi que leurs responsabilités.

Une collaboration est un ensemble de classes, d'interfaces qui travaillent ensemble pour fournir un comportement coopératif plus grand que la somme de tous ses éléments.

Un schéma consiste en la description de la structure conceptuelle de la Base de données (relationnelles ou O.O.) relative au système modélisé. Il fait référence aux informations persistantes du système.

Généralement, un diagramme de classes ne modélise qu'une partie des éléments et relations qui composent la vue de conception du système. Il ne présente souvent qu'une seule collaboration à la fois.

Pour modéliser une collaboration, il faut :

- **Identifier le mécanisme que l'on souhaite modéliser : un comportement de la partie du système modélisé et qui résulte de l'interaction de classes, d'interfaces**

- **Identifier, pour chaque mécanisme, les classes, interfaces et autres collaborations qui participent à cette collaboration ainsi que les relations qui existent entre ces éléments.**

En fonction :

- **du degré d'avancement de l'analyse (étude préalable ou détaillée)**
- **de la destination de ces diagrammes (utilisateurs, programmeurs)**
- **du message que l'on veut faire passer**
- **du niveau d'abstraction désiré**

les diagrammes de classes modélisés peuvent être plus ou moins détaillés et/ou complets.

En principe, un modèle est destiné à être utilisé pour générer du code. Le but final est le logiciel et pas les diagrammes.

UML est indépendant de tout langage. Il est néanmoins très orienté vers des langages comme C++, Java, Visual Basic, …

Lors de la transformation d'un modèle UML en code, il faut que le modèle et l'implémentation correspondent au mieux.

Comme le langage UML est plus riche d'un point de vue sémantique que tous les langages de programmation O.O., il y aura forcément une perte d'informations du point de vue sémantique lors du passage au code. C'est pourquoi, on a besoin en plus du code de garder les modèles pour une meilleure compréhension du système analysé.

Dans la pratique, on peut modéliser en utilisant des stéréotypes indiquant le langage cible et des propriétés d'attributs et d'opérations orientés vers ce langage. On parle alors d'un modèle de conception légèrement différent du modèle d'analyse initial indépendant du langage.

2 Modélisation d'un schéma logique de B.D.

Un diagramme de classes UML est un SURENSEMBLE d'un diagramme Entités-Relations.

- **Diagramme E-R : centré uniquement sur les données**

- **Diagramme de classes UML : permettent également la modélisation de comportements.**

Pour modéliser un schéma, il faut :

- **Identifier les classes du modèle qui sont persistantes (dont l'état doit dépasser la durée de vie de leurs applications).**
- **Créer un diagramme de classes qui contient ces classes et les marquer comme persistantes (étiquette standard)**
- **Définir correctement le détail des attributs, les associations et les cardinalités. Ces caractéristiques ont une influence sur le modèle physique de la base de données.**
- **Essayer de simplifier la structure logique du modèle. Eviter : les associations cycliques, les associations 1-1 et les associations n-aires. Si cela est nécessaire, on peut créer des abstractions intermédiaires pour simplifier cette structure.**
- **Définir correctement le comportement des classes à propos des opérations qui sont importantes pour l'accès aux données et l'intégrité des données.**

• En général la logique de métier est encapsulée dans une couche au-dessus des classes persistantes.

• Utiliser des outils pour transformer la structure logique définie en structure physique.

3 Trucs et astuces

Un diagramme de classes UML doit :

• Mettre l'accent sur un aspect de la vue de conception statique d'un système.

• Ne contenir que les éléments spécifiques à cet aspect

• Fournir les détails voulus en fonction du niveau d'abstraction choisi

• Se limiter aux décorations nécessaires.

- **Pour définir les classes du domaine :**
  - **avoir recours à un expert du domaine**
  - **trouver une liste de classes candidates**
  - **éliminer les classes redondantes**
  - **donner un nom clair et précis, pas trop long, aux classes**
  - **attention aux classes qui ont un nom de rôle (ClientPrincipal, ClientSecondaire → une seule classe Client !!!)**
  - **niveau de granularité. Eviter des attributs décomposables (plusieurs valeurs) → il s'agit alors probablement d'une classe non modélisée ! Ex. : auteur, attribut de la classe Livre …**
  - **modéliser une relation par une association et pas par un attribut. Ex. : pas d'attribut Auteur dans la classe Livre → association entre Livre et Auteur**

- **éviter les associations n-aire. Souvent association ternaire → association binaire qualifiée**
- **éviter d'augmenter inutilement le nombre d'associations entre classes (chemins redondants !). Multiplier les chemins diminuent la réutilisabilité de l'application !**
- **les attributs ne doivent pas être dérivés d'autres attributs**
- **les attributs sont trouvés tout au long du cycle de développement**
- **simplifier le diagramme en utilisant l'héritage (généralisation/spécialisation)**
- **tester les chemins d'accès aux classes, sont-ils suffisants ou au contraire trop nombreux ?**
- **Itérer et affiner le modèle. Un diagramme n'est jamais bon du premier coup !**

- **Ne pas ajouter les tableaux ou listes d'objets dans les classes. Cette information est visible dans les multiplicités associées aux relations entre les classes.**
- **Une bonne façon de représenter un tableau d'instances d'une classe est d'utiliser l'agrégation ou la composition**
- **Minimiser le nombre de responsabilités d'une classe !**

Classes et relations avancées

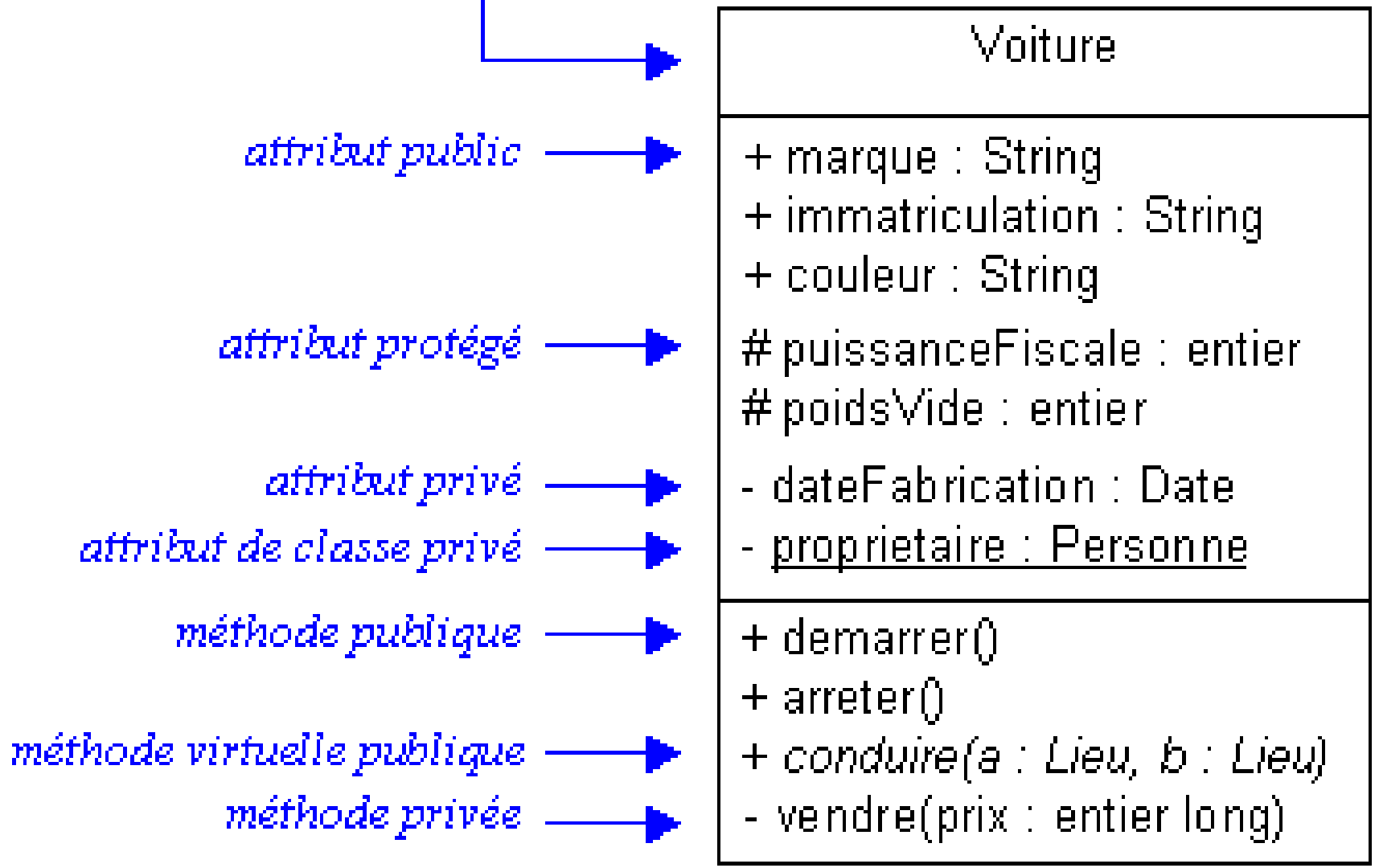
1 Classes avancées

1.1 Visibilité

La visibilité des attributs et opérations d'un classificateur (classe, interface, composant, …) indique si d'autres classificateurs peuvent utiliser ces caractéristiques :

- **public** : notée par le symbole **+**, tout classificateur extérieur peut utiliser la caractéristique sur laquelle est appliquée la visibilité. Valeur par défaut
- **protected** : notée par le symbole **#**, seuls les classificateurs qui héritent du classificateur en cours peuvent utiliser la caractéristique
- **private** : le classificateur et lui seul peut utiliser la caractéristique. Elle est notée par le symbole **-**.

classe détaillée : *les attributs sont typés, les prototypes des méthodes sont spécifiés et les niveaux de protection des membres sont renseignés.*



La visibilité des attributs d'une classe permet de définir le niveau d'ENCAPSULATION désiré pour les données de chaque instance de la classe.

1.2 Portée

La portée d'une caractéristique d'un classificateur indique si cette caractéristique existe pour chaque instance du classificateur :

- **instance : chaque instance du classificateur possède sa propre valeur pour la caractéristique. Valeur par défaut.**
- **classificateur : la caractéristique n'a qu'une seule valeur commune à toutes les instances du classificateur.**

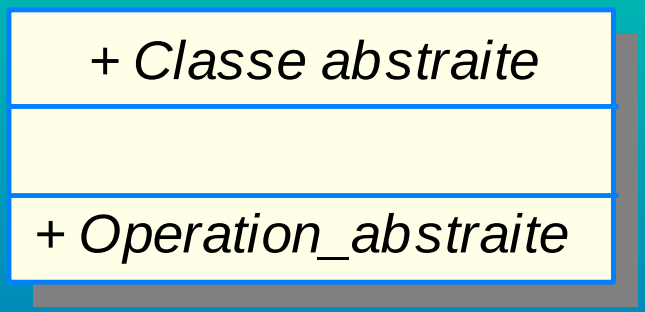
Une caractéristique de portée classificateur est notée avec son nom souligné.

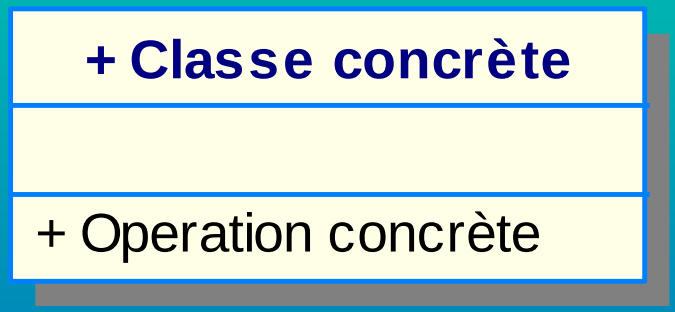
La portée classificateur correspond à la portée « static » en java et C++.

1.3 Classificateurs abstraits, terminaux, racines et polymorphes

Une classe abstraite est une classe qui ne peut pas avoir d'instance directe. Elle correspond aux classes abstract Java.

Une classe abstraite a son nom écrit en italique. Elle possède en général des opérations également définies abstraites (nom en italique).





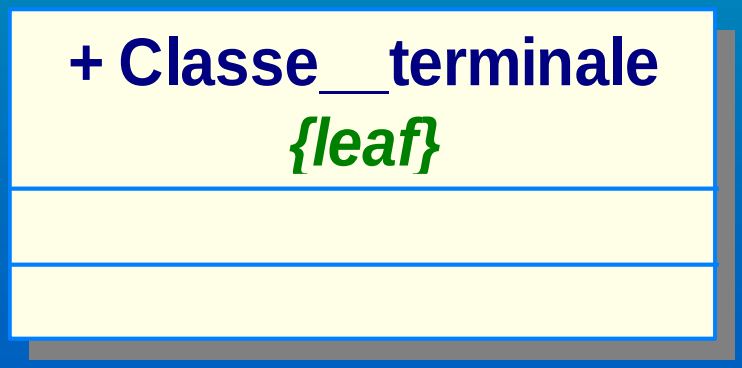
Une classe concrète peut par contre avoir des instances directes.

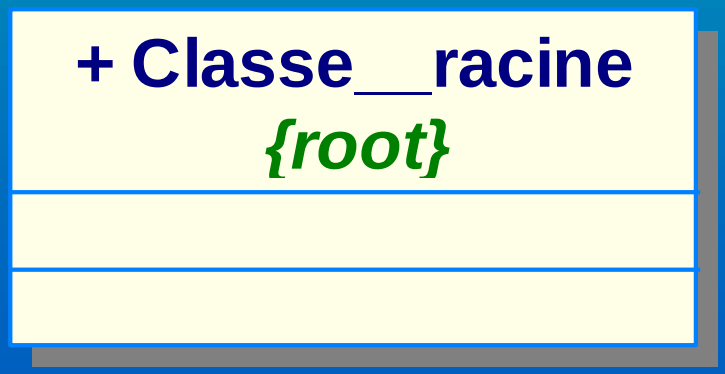
On utilise des classes abstraites pour déclarer des comportements qui seront définis et implémentés dans des classes filles. On interdit de pouvoir utiliser des instances de la classe mère.

On peut également définir une classe comme étant **terminale.** Elle ne peut plus alors avoir de classes filles. Elle est notée avec la propriété **leaf** inscrite sous le nom de la classe.

Une opération concrète peut également être déclarée terminale dans une classe non terminale. On est ainsi certain qu'elle ne pourra plus être redéfinie dans une classe fille. Elle est aussi notée {leaf}. Elle correspond à la propriété **final** en java.

De la même façon, on peut déclarer une classe comme ne pouvant pas avoir de classe mère. On parle de classe racine.





Dans une hiérarchie de classe, on peut trouver des opérations ayant exactement la même signature à des endroits différents de la hiérarchie.

Si on définit une implémentation d'une opération dans une classe fille, alors qu'il existait déjà une implémentation de cette opération dans une classe mère; alors le comportement de cette opération pour une instance de la classe fille est celui de la nouvelle implémentation.

A l'exécution, au moment de l'appel de la fonction, une correspondance avec la fonction adéquate se fait en fonction du type d'instance sur laquelle la fonction est appelée.

• class forme // forme est une classe concrète

{ void dessiner(){}

    void effacer(){} }

• class cercle extends forme

{ void dessiner(){System.out.println("Je dessine un cercle");}

   void effacer(){System.out.println("J'efface un cercle");} }

• class carre extends forme

{ void dessiner(){System.out.println("Je dessine un carre");}

   void effacer(){System.out.println("J'efface un carre");} }

• class ligne extends forme

{ void dessiner(){System.out.println("Je dessine une ligne");}

   void effacer(){System.out.println("J'efface une ligne");} }

```
• abstract class  forme                    // forme est une classe abstraite
{  abstract void dessiner();
    abstract void effacer(); }

• class cercle extends forme
{ void dessiner(){System.out.println("Je dessine un cercle");}
   void effacer(){System.out.println("J'efface un cercle");} }

• class carre extends forme
{ void dessiner(){System.out.println("Je dessine un carre");}
   void effacer(){System.out.println("J'efface un carre");} }

• class ligne extends forme
{ void dessiner(){System.out.println("Je dessine une ligne");}
   void effacer(){System.out.println("J'efface une ligne");}}
```

```
• interface  forme            // forme est une interface
{  void dessiner();
    void effacer();}

• class cercle  implements forme
{public  void dessiner(){System.out.println("Je dessine un cercle");}
  public  void effacer()  {System.out.println("J'efface un cercle");} }

• class carre implements forme
{ public void dessiner(){System.out.println("Je dessine un carre");}
   public void effacer(){System.out.println("J'efface un carre");} }

• class ligne implements forme
{ public void dessiner(){System.out.println("Je dessine une ligne");}
   public void effacer(){System.out.println("J'efface une ligne");}}
```

```
public class Formes
{   public static void main(String arg[])
  {  carre c=new carre(); ligne l=new ligne(); cercle ci=new cercle();
     c.dessiner();
     l.dessiner();
    ci.dessiner();
    forme T[] = new forme[5];
    T[0]=new carre();    T[1]=new ligne();
    T[2]= new cercle();
     for(int i=0;i<3;i++) T[i].dessiner();
     for(int i=0;i<3;i++) T[i].effacer();
      /*  forme F=new forme();  Strictement interdit !!!! Si forme est une classe
      abstraite ou une interface. Autorisé si forme est une classe concrète.
      Mais alors F.dessiner() ??????    */
}}
```

Polymorphisme d'opérations

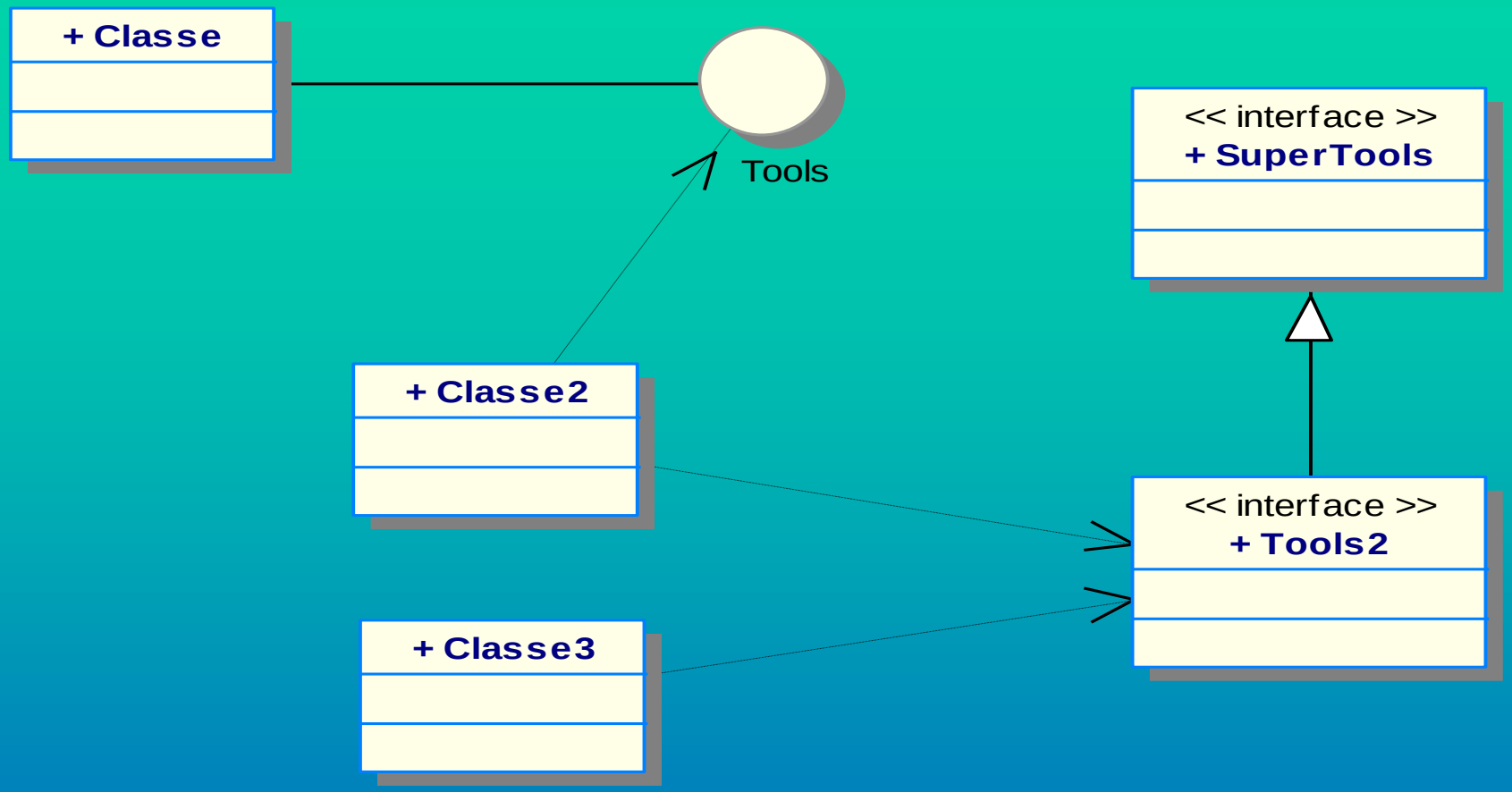
Une interface décrit le comportement visible d'une classe, d'un composant, d'un paquetage ou d'un autre classificateur.

Ce comportement visible est défini par une liste d'opérations ayant une visibilité publique.

On représente les interfaces au moyen d'un cercle relié par un trait à l'élément qui fournit les services décrits par l'interface.

On peut également représenter une interface au moyen d'une classe stéréotypée par le mot <<interface>>. La classe qui réalise alors l'interface est relié à celle-ci par une flèche en pointillé.

Une classe peut réalisé plusieurs interfaces et une interface peut être réalisée par plusieurs classes.



La notation au moyen des classes <<interface>> permet également de représenter des relations de généralisation entre interfaces.

1.4 Multiplicité

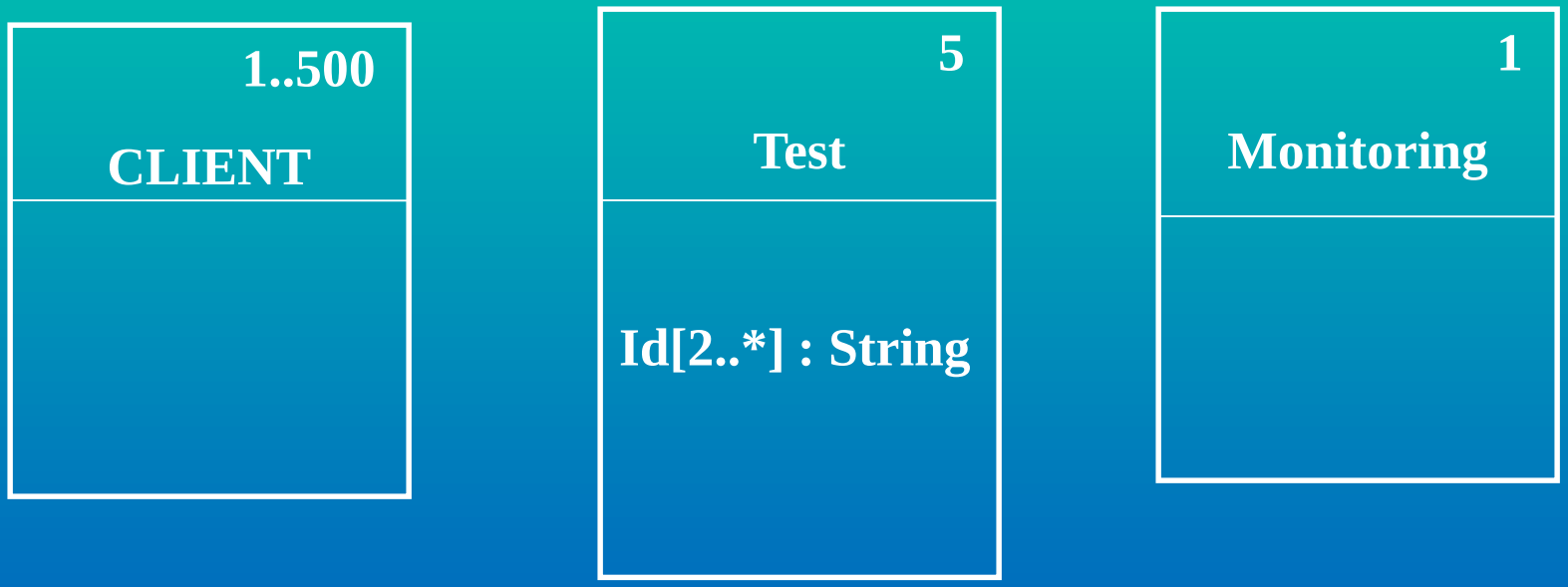
Une classe est supposée être associée à un nombre quelconque d'instances, sauf :

- **une classe abstraite ne peut pas être associée à une instance**
- **une classe peut parfois avoir un nombre réduit d'instances :**
  - **0 : la classe est alors une classe utilitaire qui met des attributs et opérations (portée classificateur) à disposition d'autres classes. Cfr méthodes statiques d'une classe publique en Java**
  - **1 : classe singleton**
  - **n : nombre précis défini**
  - *** : nombre quelconque (valeur par défaut)**

Le nombre d'instances d'une classe est sa multiplicité

Cette multiplicité s'indique par un nombre situé en haut à droite de l'icône de classe.

La multiplicité peut également s'appliquer aux attributs et se note par une valeur entre crochets à coté du nom.



1.5 Attributs

Au niveau d ’abstraction le plus élevé, la modélisation des caractéristiques structurelles d ’une classe (ses attributs), on peut se contenter de préciser le nom de chaque attribut.

A des niveaux de détails plus importants, on peut préciser la visibilité, la portée, la multiplicité.

On peut également préciser le type , la valeur initiale et la variabilité.

Syntaxe générale :

[Visibilité] nom [multiplicité] [:type]

[=Valeur initiale] [{Chaine_propriété}]

{Chaine_propriété} peut prendre les valeurs suivantes :

- **{changeable} : pas de restrictions sur la modification de la valeur de l ’attribut**

- **{addOnly} : pour les attributs dont la multiplicité est supérieure à 1, on peut ajouter des valeurs supplémentaires. Mais une fois créée, une valeur ne peut être ni enlevée ni modifiée.**

- **{frozen} : l ’attribut est en fait une constante qui doit être inialisée (cfr const en C++ et final en Java).**

1.6 Opérations

Comme pour les attributs, plusieurs niveaux de détails peuvent être utilisés pour les définir.

UML fait la différence entre une opération et une méthode.

Une opération correspond au service qui peut être demandé à n ’importe quel objet de la classe pour déclencher un comportement.

Une méthode est l ’implémentation d ’une opération. Chaque opération non abstraite doit avoir une méthode qui fournit un algorithme exécutable.

Syntaxe générale :

[Visibilité] nom [(liste de paramètres)]

[:type de retour][{chaine_propriété}]

{Chaine_propriété} peut prendre les valeurs suivantes :

- **{leaf} : opération terminale**
- **{isQuery} : l ’opération ne modifie pas le système. Simple requête sans effet secondaire.**
- **{sequential, gurded, concurrent} : cfr objets actifs, processus, threads**

Chaque paramètre passé peut suivre la syntaxe suivante :

[direction] nom : type [=valeur par défaut]

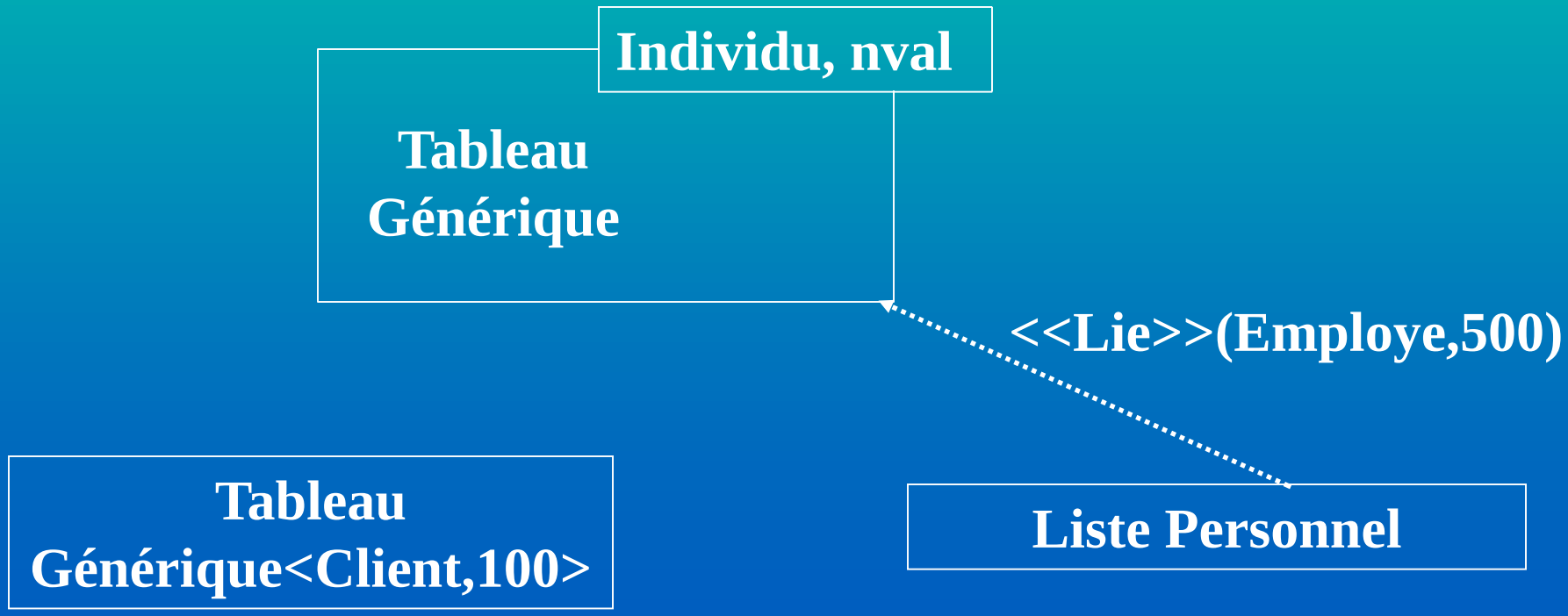
- **in : paramètre en entrée. Ne peut être modifié**
- **out : Paramètre en sortie. Peut être modifié**
- **inout : paramètre d ’entrée pouvant être modifié**

- **Display()**
- **+ Display():void**
- **+Display(in Titre : string) : void {leaf}**

1.7 Les classes paramétrables

Ces classes, aussi appelées classes templates, permettent de créer des patrons de classes (classes génériques).

Un patron de classe ne peut pas être instancié tel quel. Il faut lui associer les paramères nécessaires (types et/ou valeurs).



Le paramètre générique est placé dans un rectangle pointillé assiocié à la classe tandis que le paramètre effectif est :

- **soit placé entre < et > associé à la classe qui instancie le patron**

- **soit défini comme argument d'une relation de dépendance <<Lie>>**

Les classe paramétrables sont seulement utilisées en etude détaillée pour incorporer par exemple des composants réutilisables.

2 Relations avancées

Dépendances, généralisations et associations sont les 3 briques de base relationnelles les plus importantes d ’UML.

2.1 Dépendance

Une dépendance ordinaire et sans décoration est souvent suffisante.

On peut cependant utiliser des stéréotypes pour indiquer différentes nuances.

- <<derive>> : la source peut être calculée à partir de la cible

- <<friend>> : visibilité spéciale de la source dans la cible (friend en C++)

- <<instanceof>> : l ’objet source est une instance du classificateur cible

- <<instantiate>> : les opérations sur la classe source créent des instances de la classe cible

- <<refine>> : le degré d ’abstraction de la source est plus fin que la cible

- <<use>> : pour indiquer explicitement que la relation de dépendance est une relation d ’utilisation de la partie publique de la cible

- <<trace>> : la cible est une version précédente de la source.

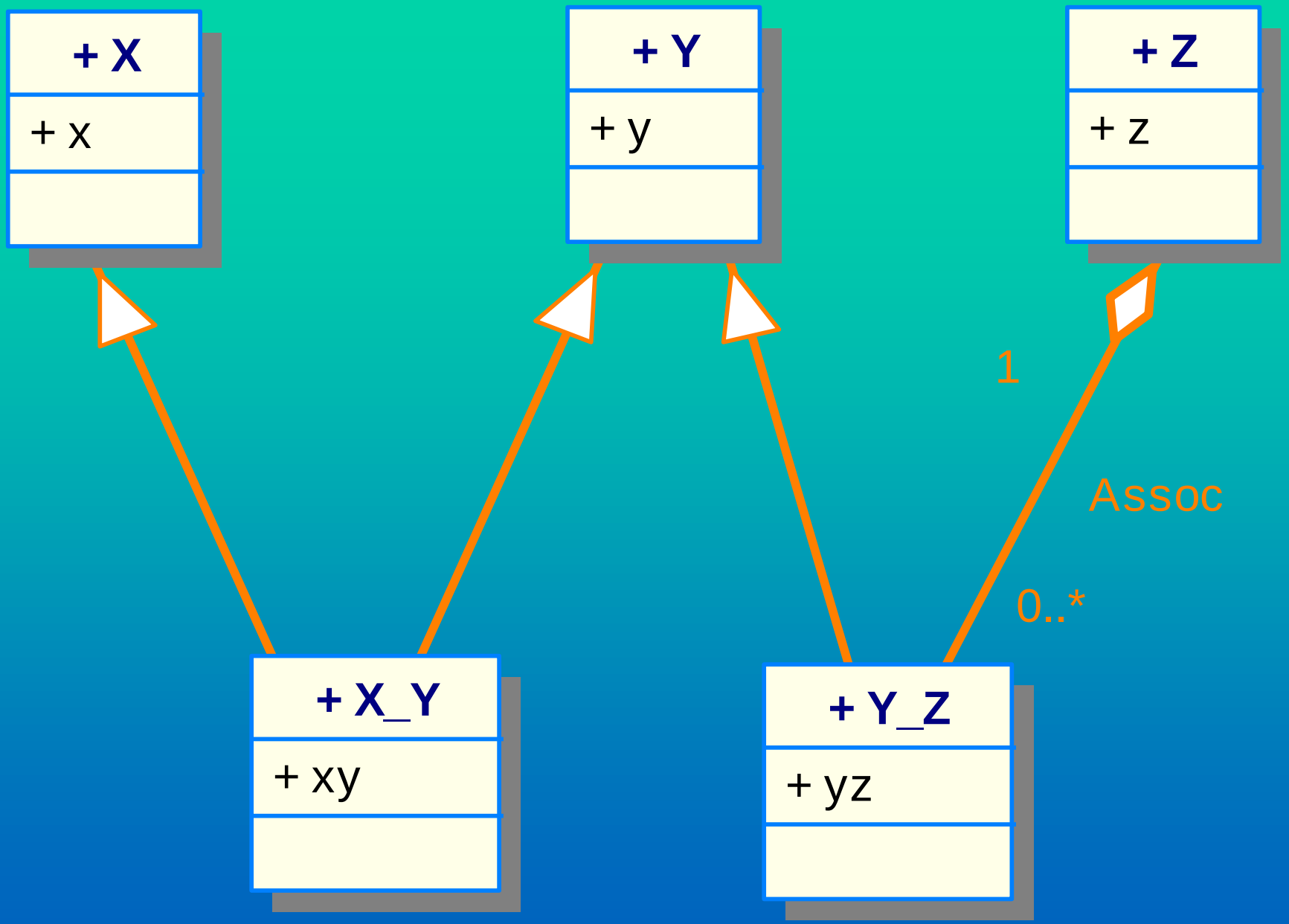
- …….. Etc

2.2 Généralisation

L ’héritage multiple doit être utilisé avec précaution. Il y a un risque qu ’un enfant soit confronté, venant de ses parents, à des structures ou comportements qui interfèrent.

Très souvent l ’héritage multiple peut être remplacé par de la délégation : un enfant hérite d ’un parent et utilise l ’agrégation pour recevoir les structures et les comportements de plusieurs parents subordonnés.

Il y a quand même une perte de la sémantique au niveau de la remplaçabilité avec les parents subordonnés.


+ X
+ x
+ Y
+ y
+ Z
+ z
1
Assoc
0..*
+ X_Y
+ xy
+ Y_Z
+ yz

Classe Java Y_Z générée :

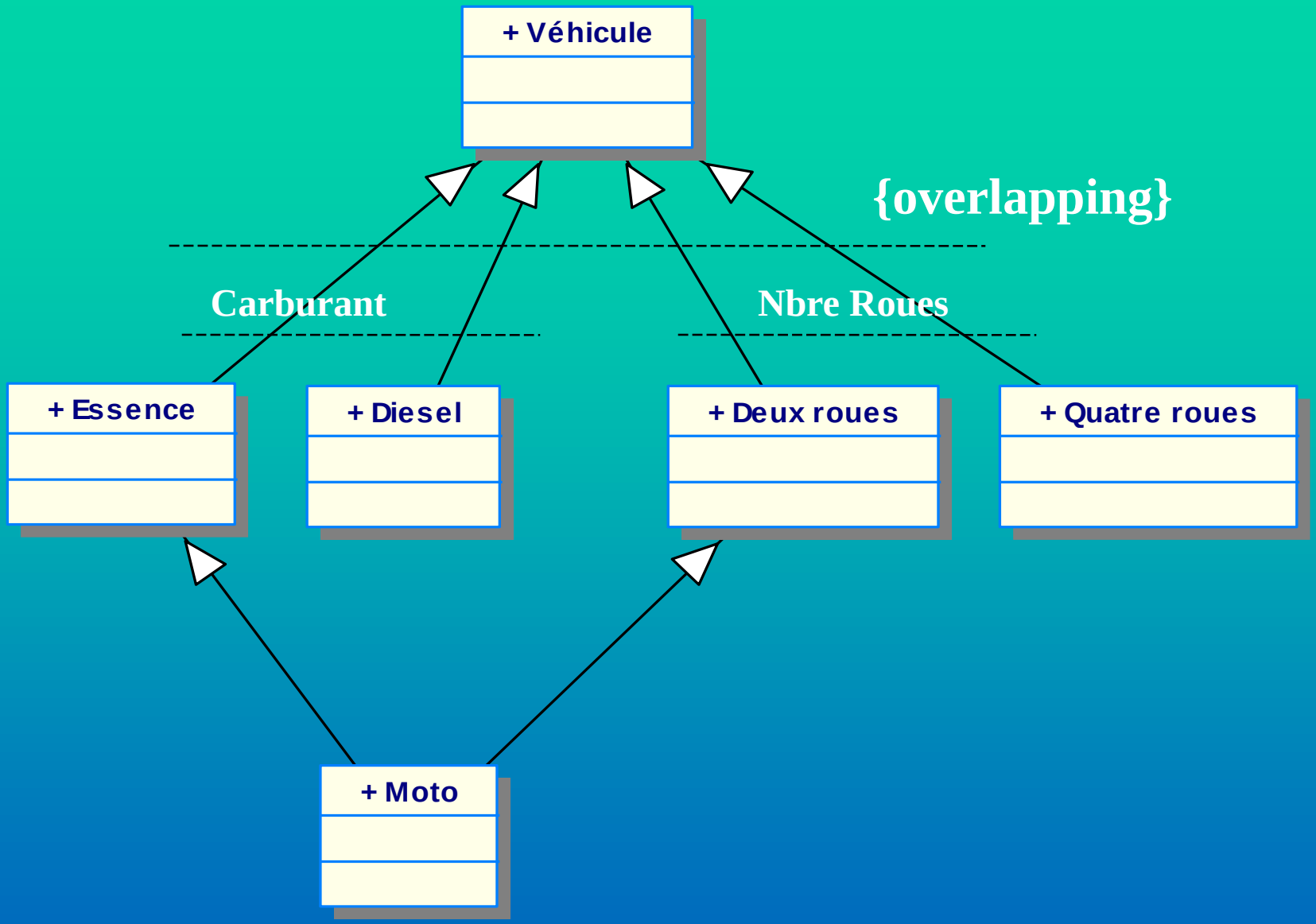
```
public  class Y_Z  extends Y
{    public char yz;

Z Assoc;
}
```

On voit que la classe Y_Z possède les attributs de la classe Z par composition (délégation) et ceux de la classe Y par héritage.

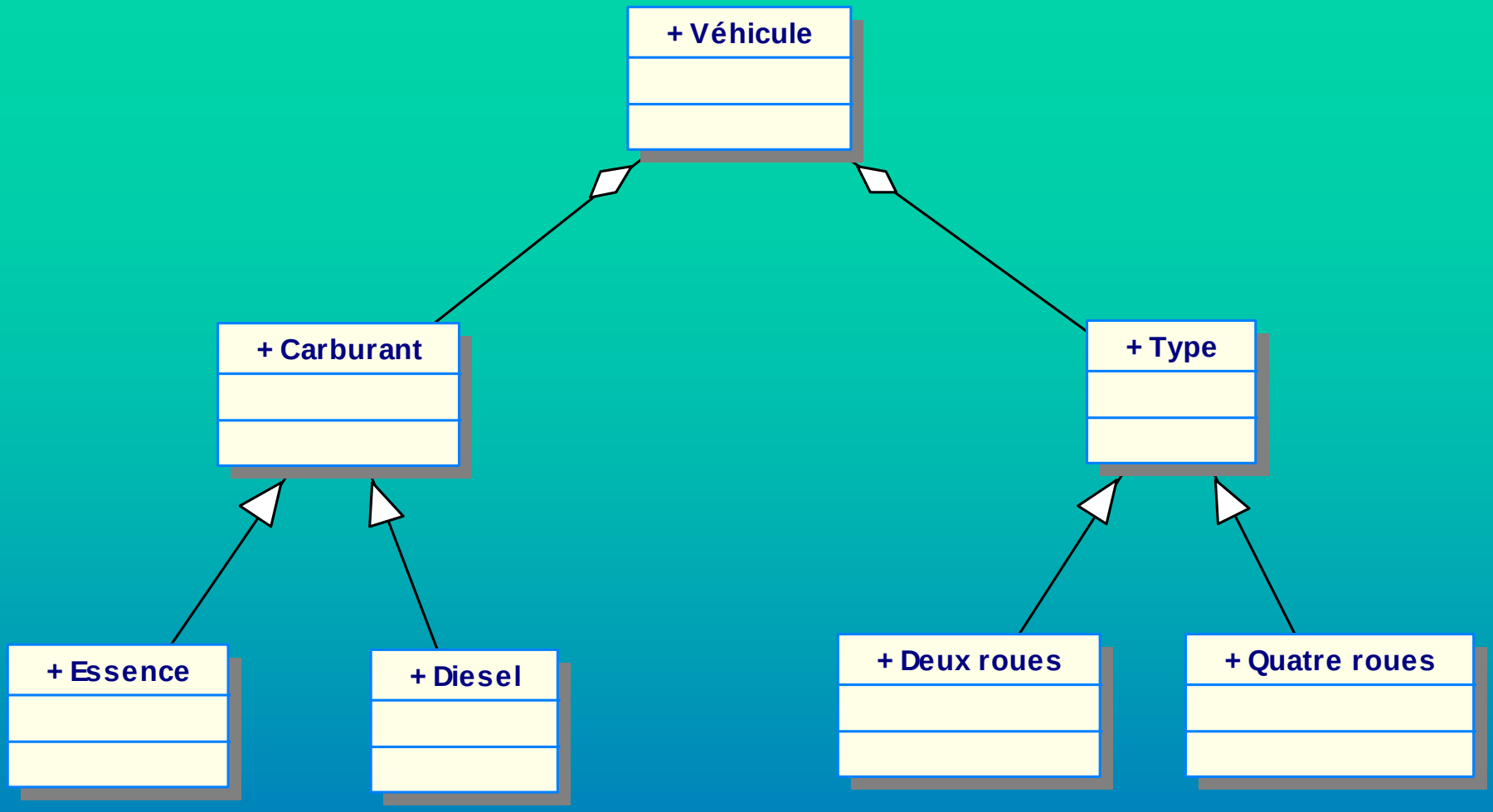
Héritage contre délégation

- l’héritage est une construction rigide mais la propagation des attributs et des opérations vers les sous-classes est automatique.

- la délégation est une construction plus souple basée sur l’agrégation mais la propagation des propriétés doit être réalisée manuellement. Elle permet la mise en oeuvre de la généralisation multiple avec les langages qui ne possèdent que l’héritage simple.

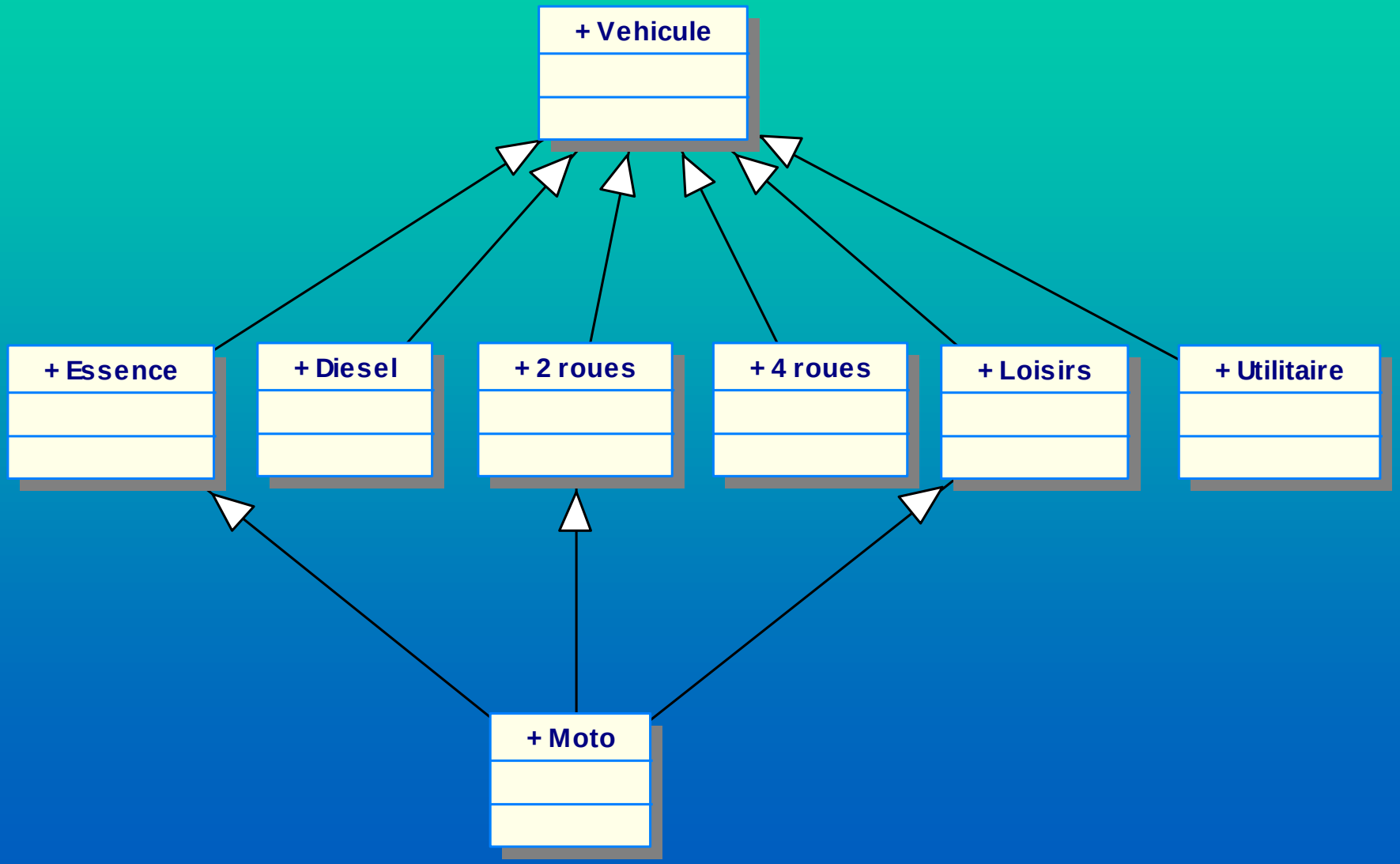


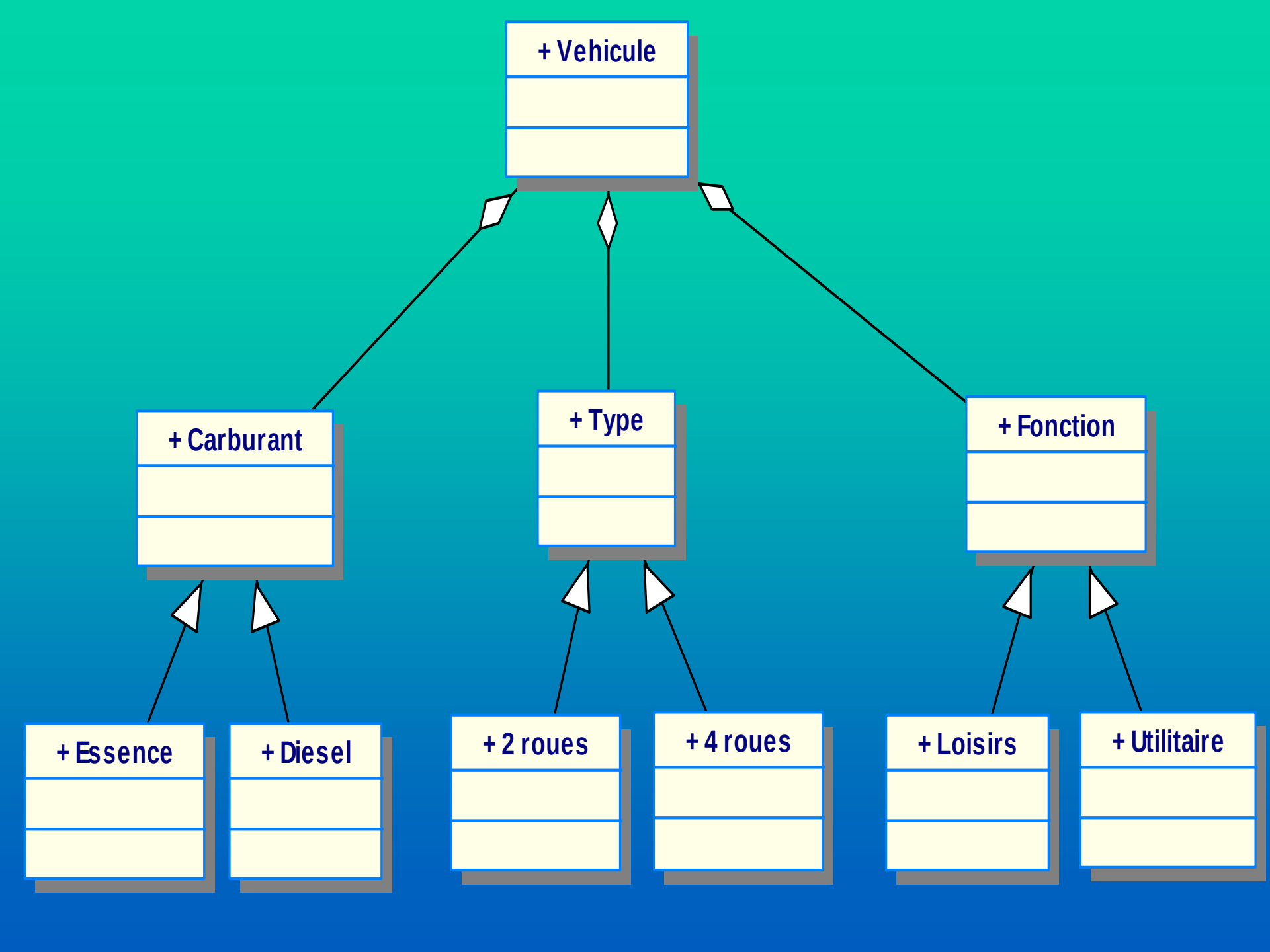
Pour pouvoir manipuler un objet Moto, il faut passer par la création d'une sous-classe Moto. Voir * plus loin.**



On peut créer un objet Moto directement par instanciation de la classe véhicule et agrégation des propriétés qui lui correspondent.

Dans le cadre de la généralisation, une extension des caractéristiques peut amener à devoir redéfinir les sous-classes terminales ce qui n ’est pas le cas de la délégation.




+ Vehicule
+ Carburant
+ Type
+ Fonction
+ Essence
+ Diesel
+ 2 roues
+ 4 roues
+ Loisirs
+ Utilitaire

On peut appliquer 4 contraintes standard lors d'un héritage :

• complete : tous les enfants ont été décrits même si certains ne sont pas représentés sur le diagramme. Les enfants supplémentaires sont interdits.

• incomplete : inverse de complete

•disjoint : les objets d'un parent donné ne peuvent pas avoir plus d'un enfant donné comme type. Inverse de overlapping.

• overlapping : indique que la super-classe possède des instances qui peuvent se recouvrir lors de la réunion de celles de ses sous-classes. * Une instance (indirecte) de véhicule peut être une instance de Essence ou une instance de Deux roues ou encore les deux (soit une instance de Moto). Les discriminants permettent d'exprimer des partitions internes à des branches (ou exclusif).**

2.3 Association

Il existe 4 décorations de base qui s'appliquent aux associations :

- **nom**
- **multiplicité de part et d'autre de l'association**
- **rôle**
- **agrégation**

Pour plus de détails, on peut préciser :

- **navigabilité**
- **qualification**
- **diverses nuances d'agrégation (composition)**

2.3.1 Navigabilité

Par défaut, elle est bidirectionnelle et elle n'est pas dessinée.


```
public  class Client
{//Attributes
  public String Ncli;
  public String Nom;
 //Attributes Association
 Compte Utiliser[];
}
```

```
public  class Compte
{ //Attributes
  public String Numero;
  public String Libelle;
  //Attributes Association
  Client Utiliser[];
}
```

On peut limiter la navigabilité à un seul sens (il doit au moins en rester un !!! ☺)

Impossible de trouver un client à partir d'un compte

```
public  class Compte
{//Attributes
 public String Numero;
 public String Libelle;
 //Attributes Association
}
```

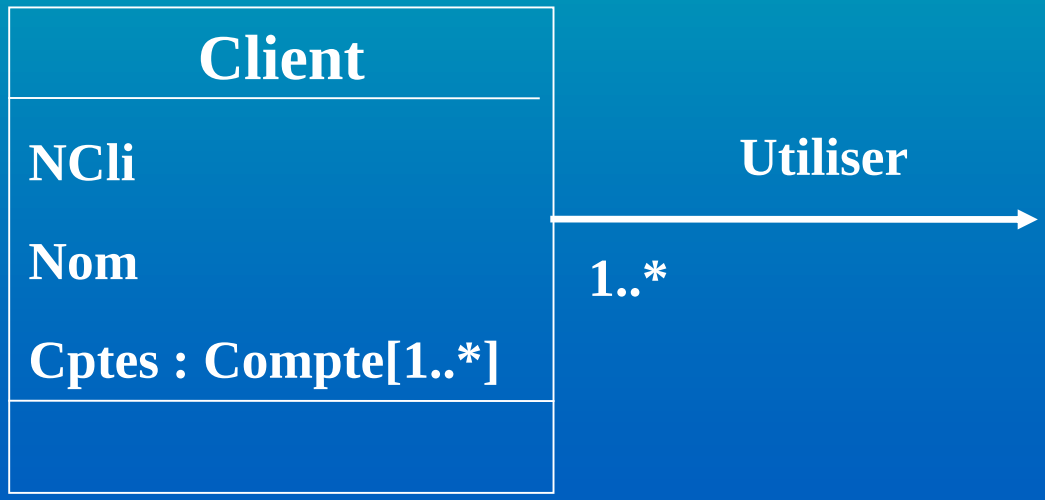
Indiquer un sens ne signifie pas qu'il est totalement impossible de naviguer à contre sens (à partir d'autres associations et classes éventuellement non représentées). Il s'agit plutôt d'indiquer le sens normal d'utilisation de l'association.

La classe Compte n'a plus de propriété d'association

On peut par contre mettre en évidence la non navigabilité vers une classe

On met en évidence la non navigabilité de Compte vers Client !

Par contre pour mettre en évidence la navigabilité, on modélise bien sûr l'association et on ajoute un attribut dans la classe source de l'association



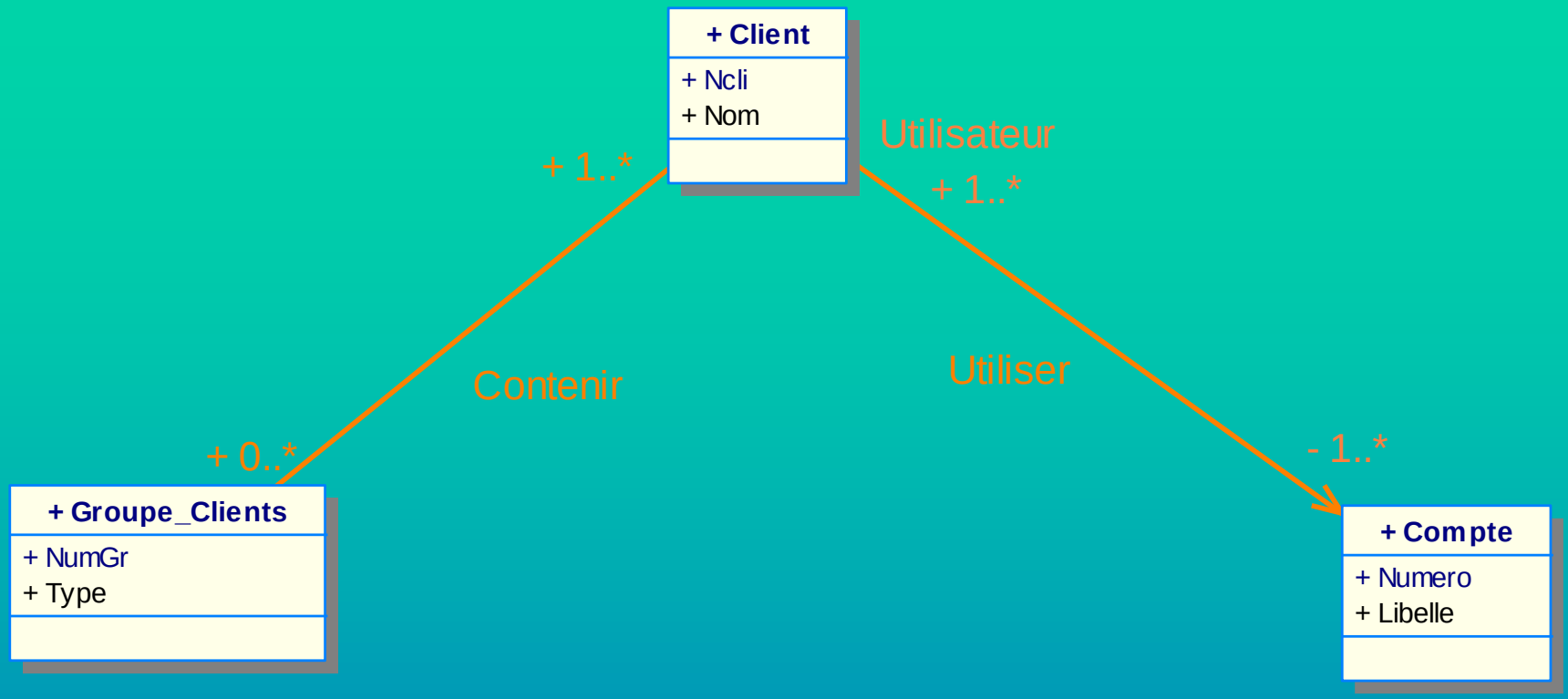
2.3.2 Visibilité

Par défaut, la visibilité d'un rôle est publique.

Comme pour les caractéristiques d'une classe, on peut indiquer 3 niveaux de visibilité à chaque extrémité d'une association :

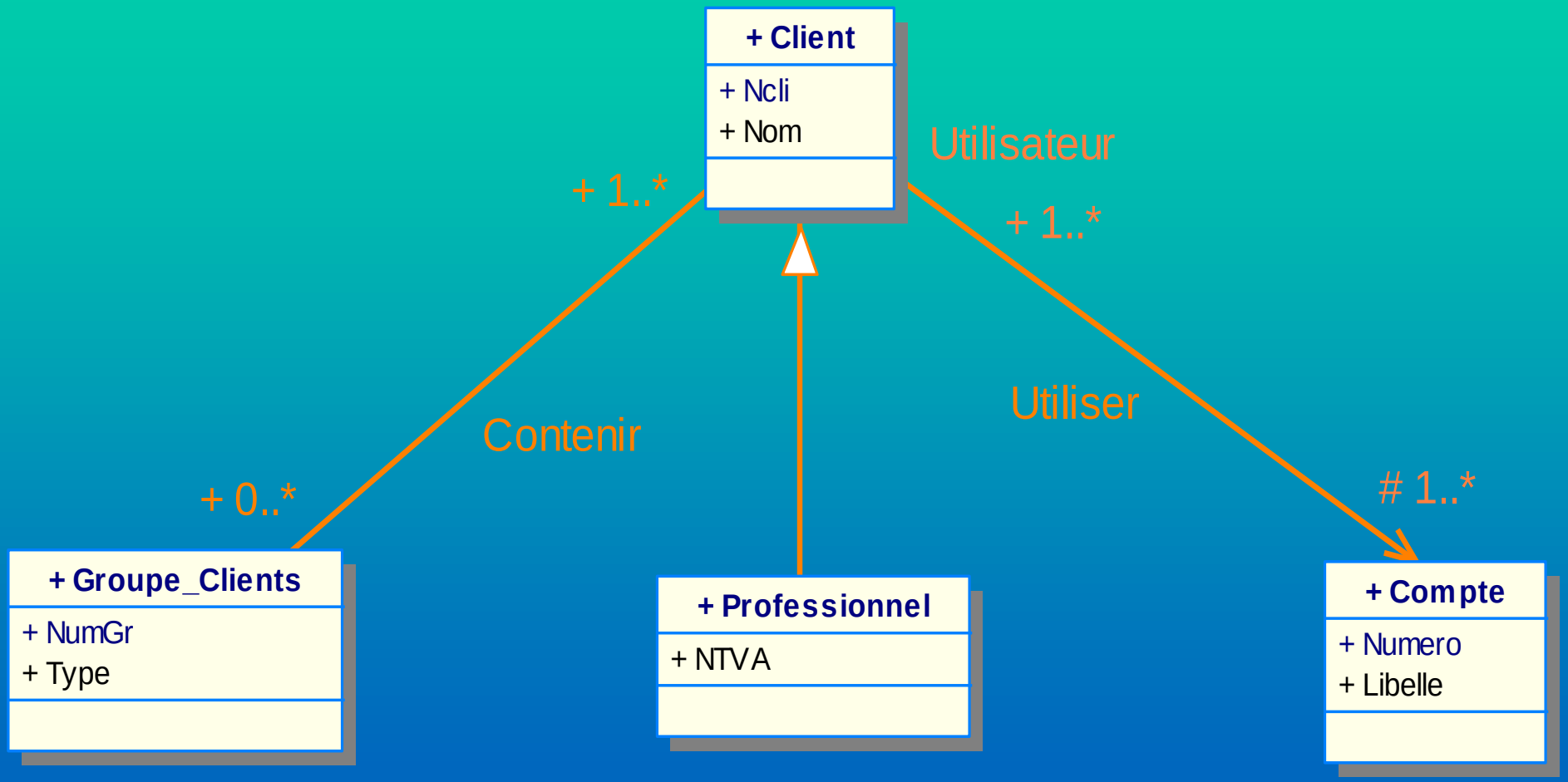
- **publique +**
- **privée –**
- **protégée #**

Si elle est privée, les objets situés à cette extrémité ne sont pas accessibles aux objets extérieurs à l'association.



Pour chaque objet Client, il est possible d'identifier les objets Compte correspondants. Mais un compte ne peut être connu que de l'utilisateur et ne peut donc pas être accessible depuis l'extérieur de l'association (via un objet Groupe_Clients.

La visibilité « protégée » indique que des objets à cette extrémité ne sont pas accessibles aux objets extérieurs à l'association, excepté aux objets enfants de la classe qui se trouve à l'autre extrémité.

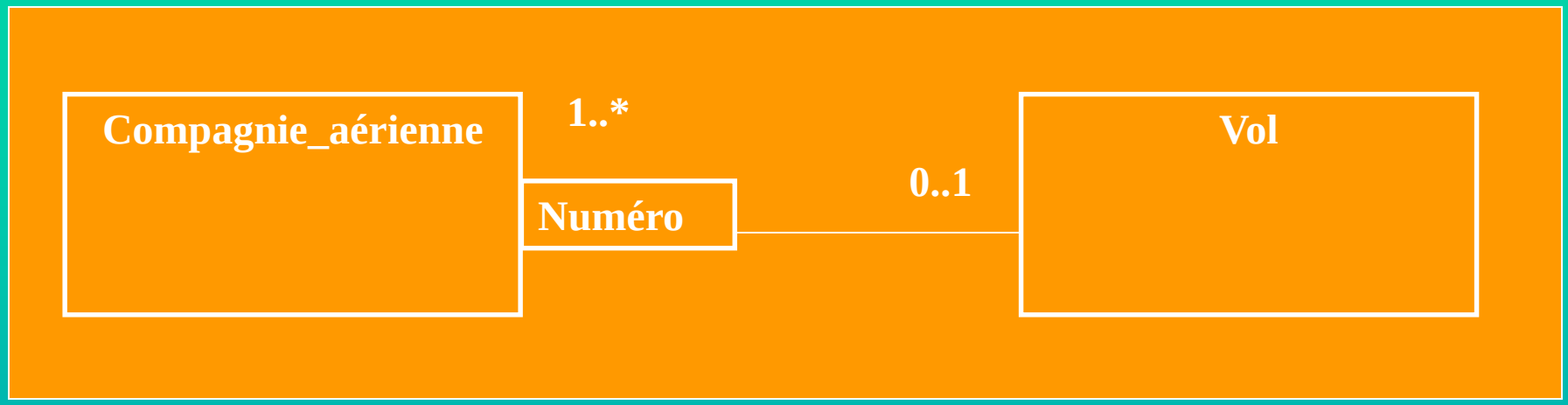


2.3.3 Qualification

La qualification d ’une association consiste à sélectionner un sous-ensemble d ’objets parmi l ’ensemble des objets qui participent à une association.

La restriction de l'association est définie par une clé (le qualificatif), qui permet de sélectionner les objets ciblés. Le qualificatif appartient pleinement à l ’association et non aux classes associées mais est utilisé conjointement avec un objet de la classe source.

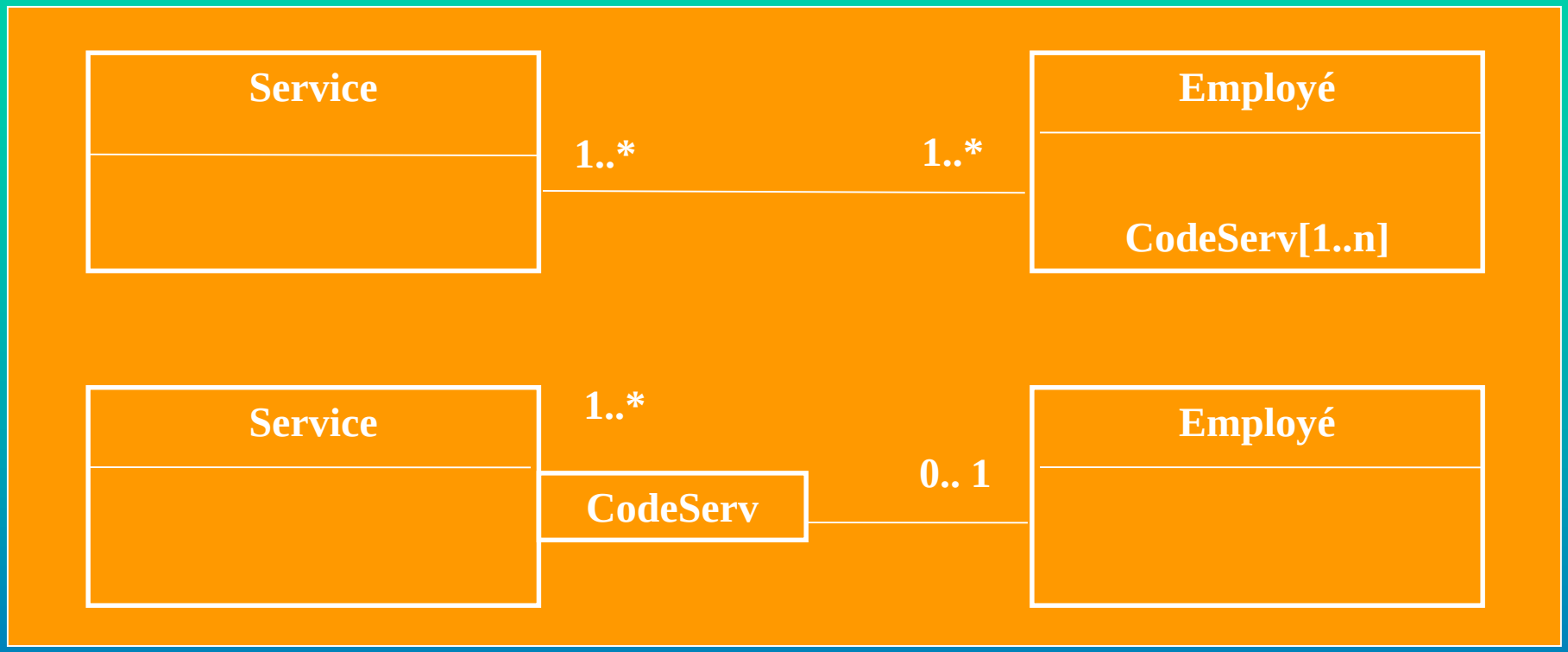
Cela devient :



L’objet source, avec une valeur de l’attribut qualificatif, exploite un objet cible (si la multiplicité de la cible est d’au plus 1) ou un ensemble d’objets (si la multiplicité est supérieure à 1).

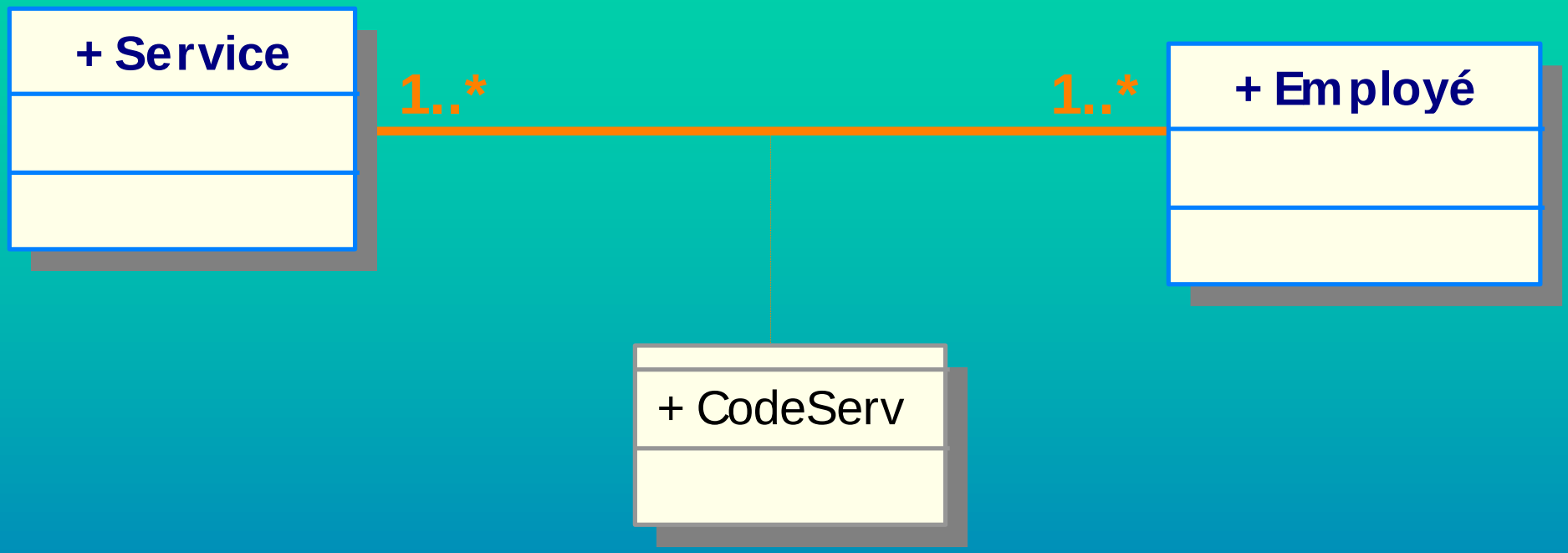
L ’utilisation d ’une association qualifiée est une alternative à l ’utilisation d ’un identifiant d ’objets. Surtout que très souvent, un identifiant est relatif et dépend donc d ’un contexte.

Une association qualifiée permet également d'éviter de déclarer des attributs mutlivalués.



Cette construction facilite la navigation de service vers Employé. On met nettement en évidence le fait qu'un employé est qualifié, au sein d'un service, par un code spécifique.

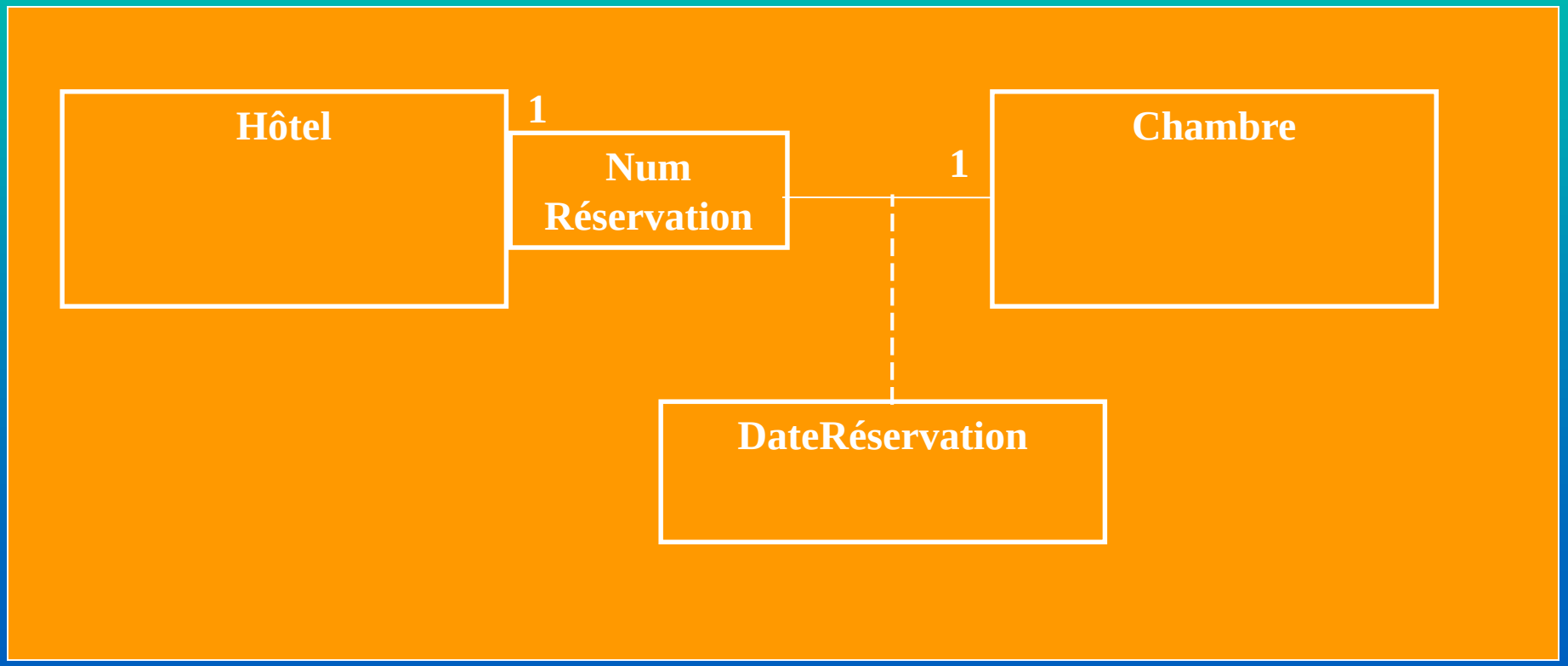
L’association qualifiée peut se transformer en classe-association car le qualificatif est un attribut de l’association.



On perd cependant en précision. Le contexte (« au sein d’un service, un employé est défini de façon unique par un code service ») est perdu. La navigation de l’association ne reflète plus la même sémantique.

Avec une classe-association, un lien entre objets donnés ne peut avoir qu'une valeur pour chaque attribut de la classe-association (les instances d'une association sont des tuples; sans doublons).

Avec une qualification, il peut y avoir plusieurs valeurs de qualification pour une association d'objets donnés.



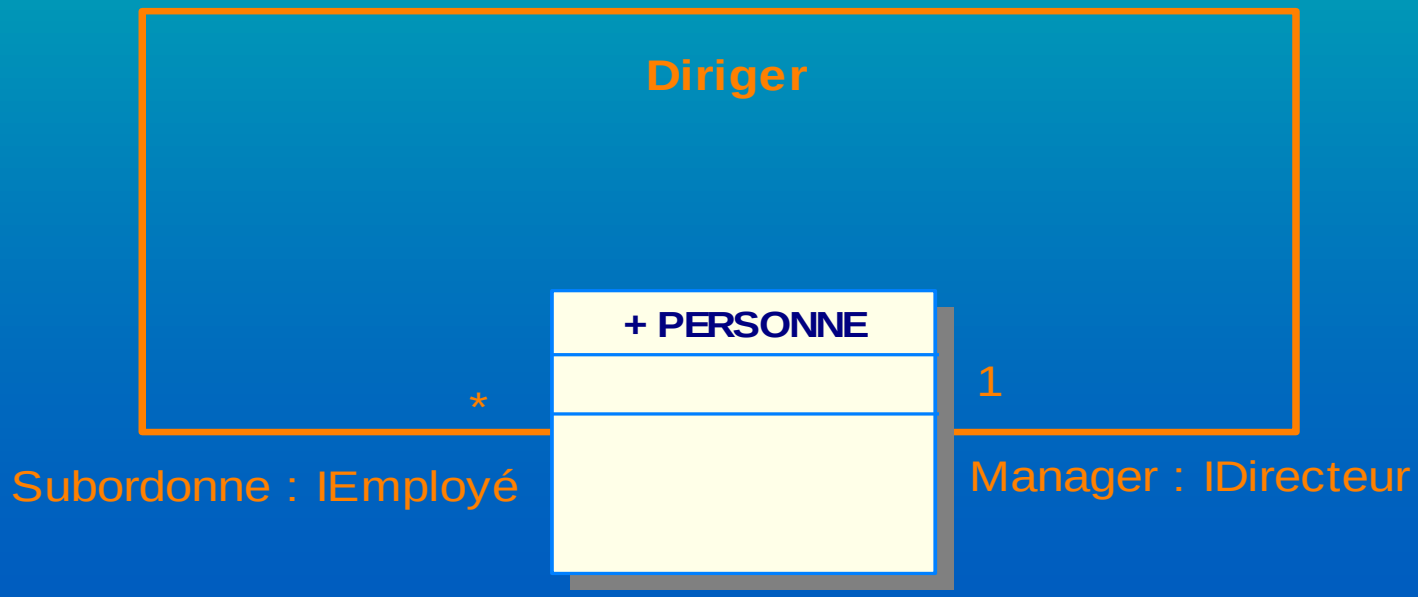
2.3.4 Spécificateur d'interface

Un interface est un ensemble d'opérations qui définissent un service d'une classe ou d'un composant.

Une classe peut réaliser différents interfaces et tous ces interfaces définissent le comportement complet de la classe.

Exemple : une classe Personne peut se comporter comme un Directeur, un Cadre ou un Employé.

On peut caractériser un rôle avec un interface spécifique



2.3.5 Les contraintes

5 contraintes peuvent être appliquées sur une association.

• **implicit** : la relation est conceptuelle et pas réelle. Elle n'est pas manifeste

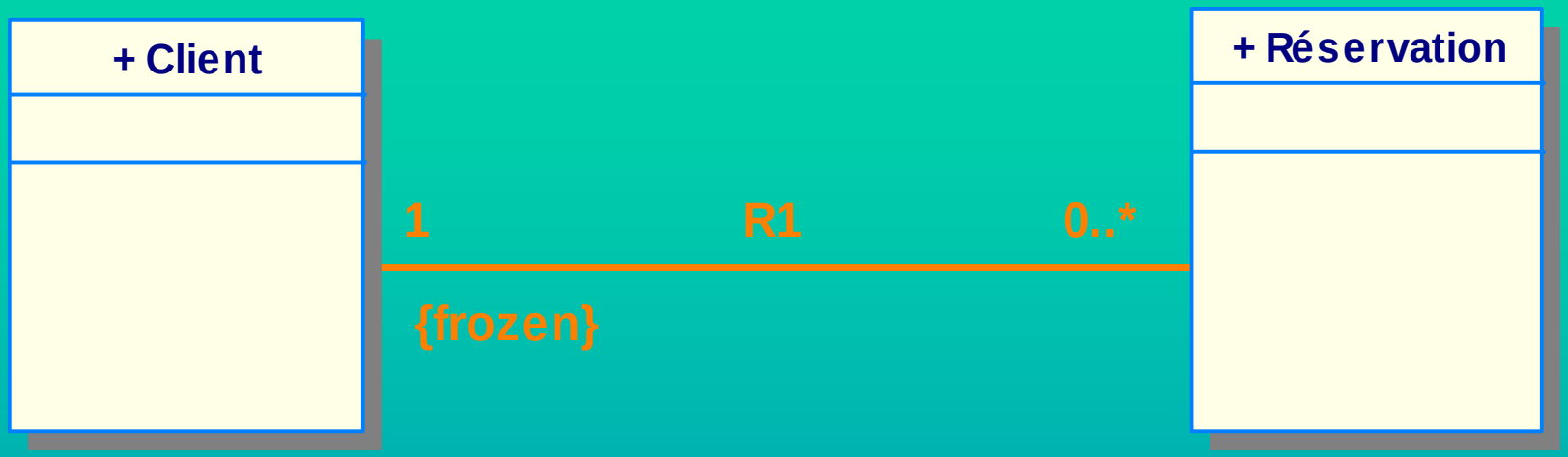
• **ordered** : l'ensemble des objets situées à une des extrémités d'une association est classé selon un ordre explicite.

• **changeable** : les liens entre objets (instances des classes) peuvent être ajoutés, supprimés ou modifiés en toute liberté à tout moment. Quand un lien unissant deux instances de classes a été créé, il peut par la suite être modifié ou supprimé.

• **addOnly** : de nouveaux liens peuvent être ajoutés depuis un objet vers l'extrémité opposée de l'association.

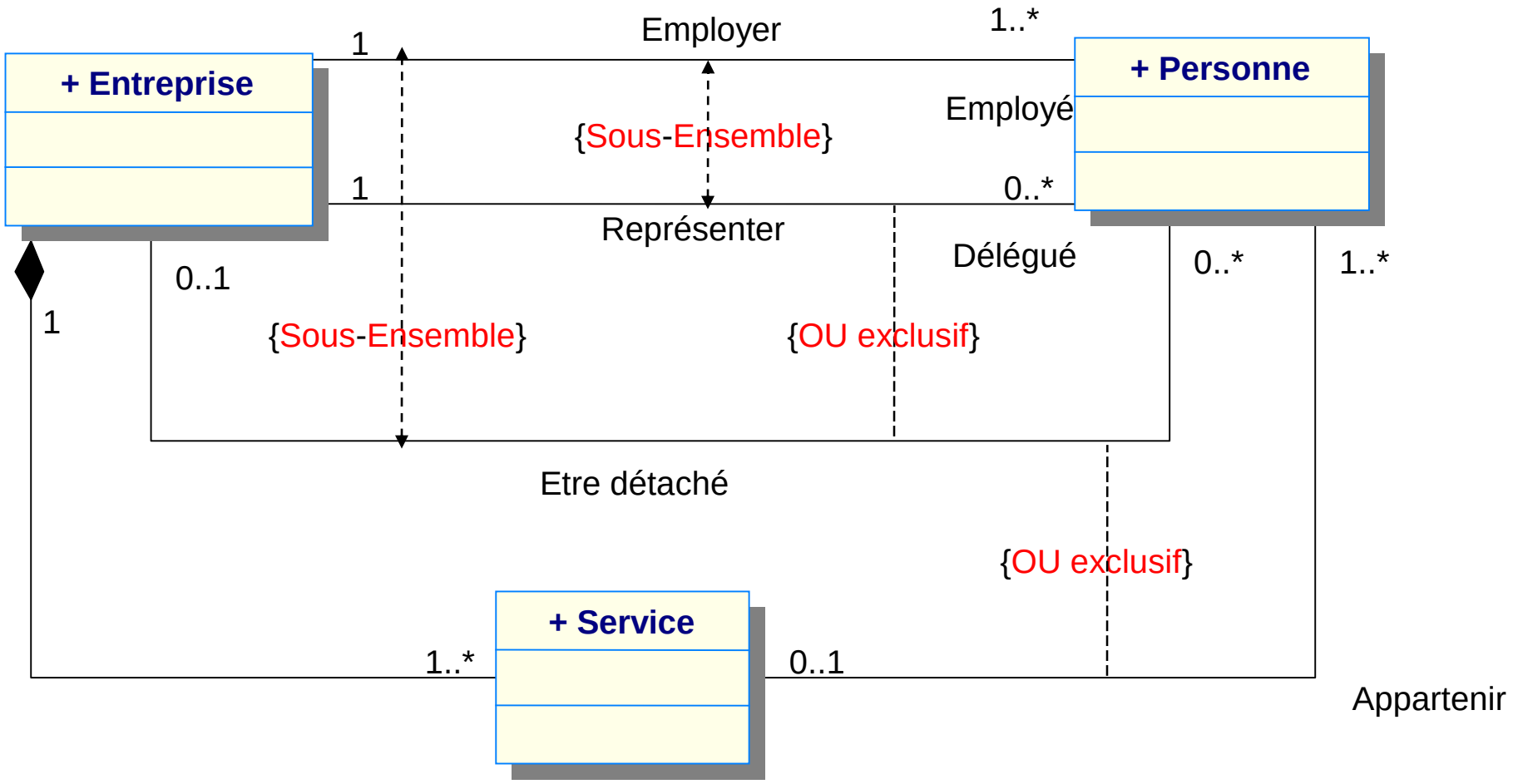
• **frozen** : une fois ajouté, un lien ne peut être ni modifié, ni effacé depuis un objet vers l'autre extrémité de l'association.

Bien que notées comme des contraintes, il s'agit en fait de propriétés qui se rapportent à une des extrémités d'une association.



Le lien qui part d'une instance de Client vers une instance de Réservation ne peut pas être modifié/supprimé. Pour pouvoir ce faire, il faut d'abord supprimer la réservation correspondante.

On peut également définir des contraintes sur un groupe d ’associations.

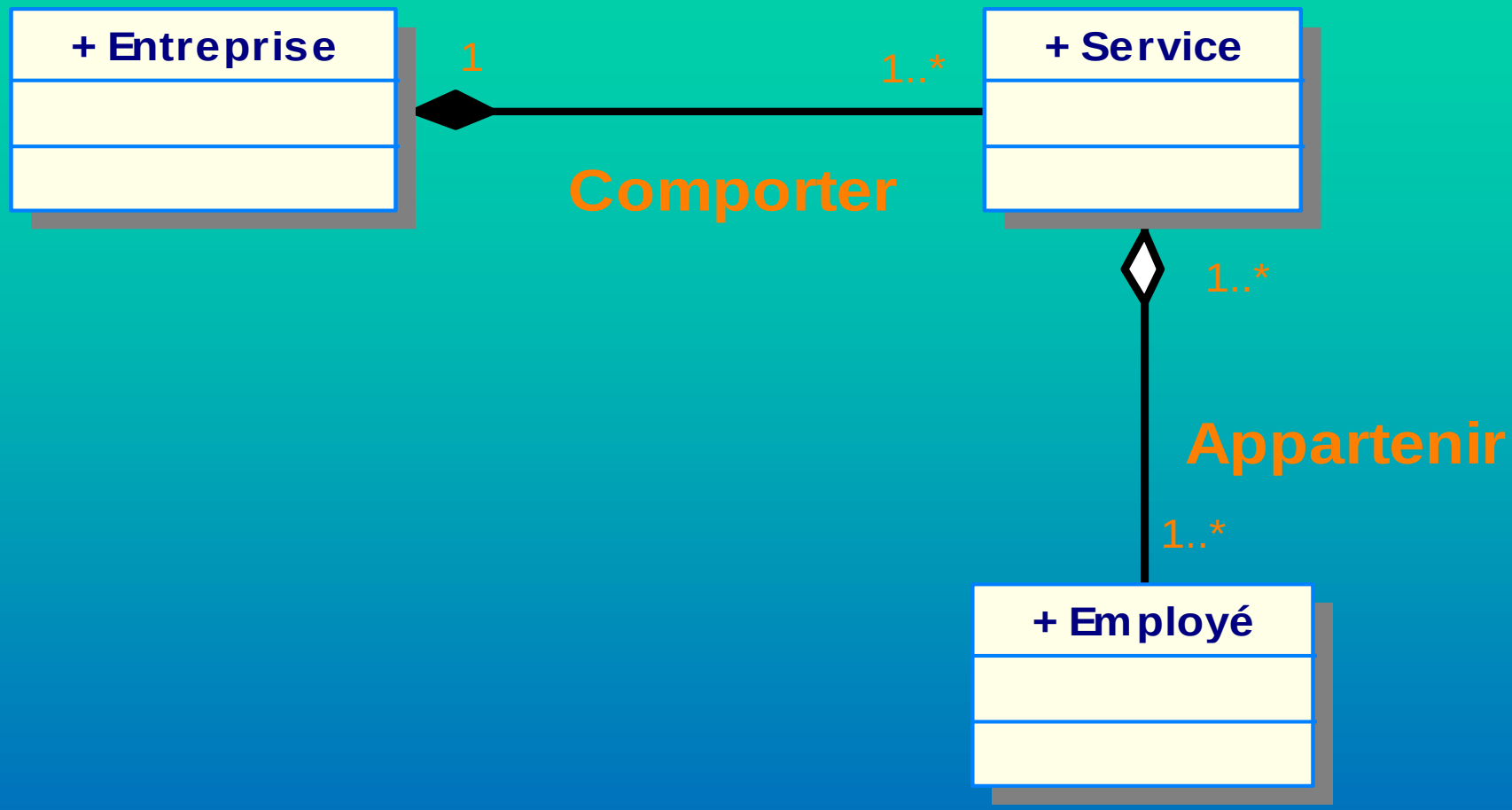


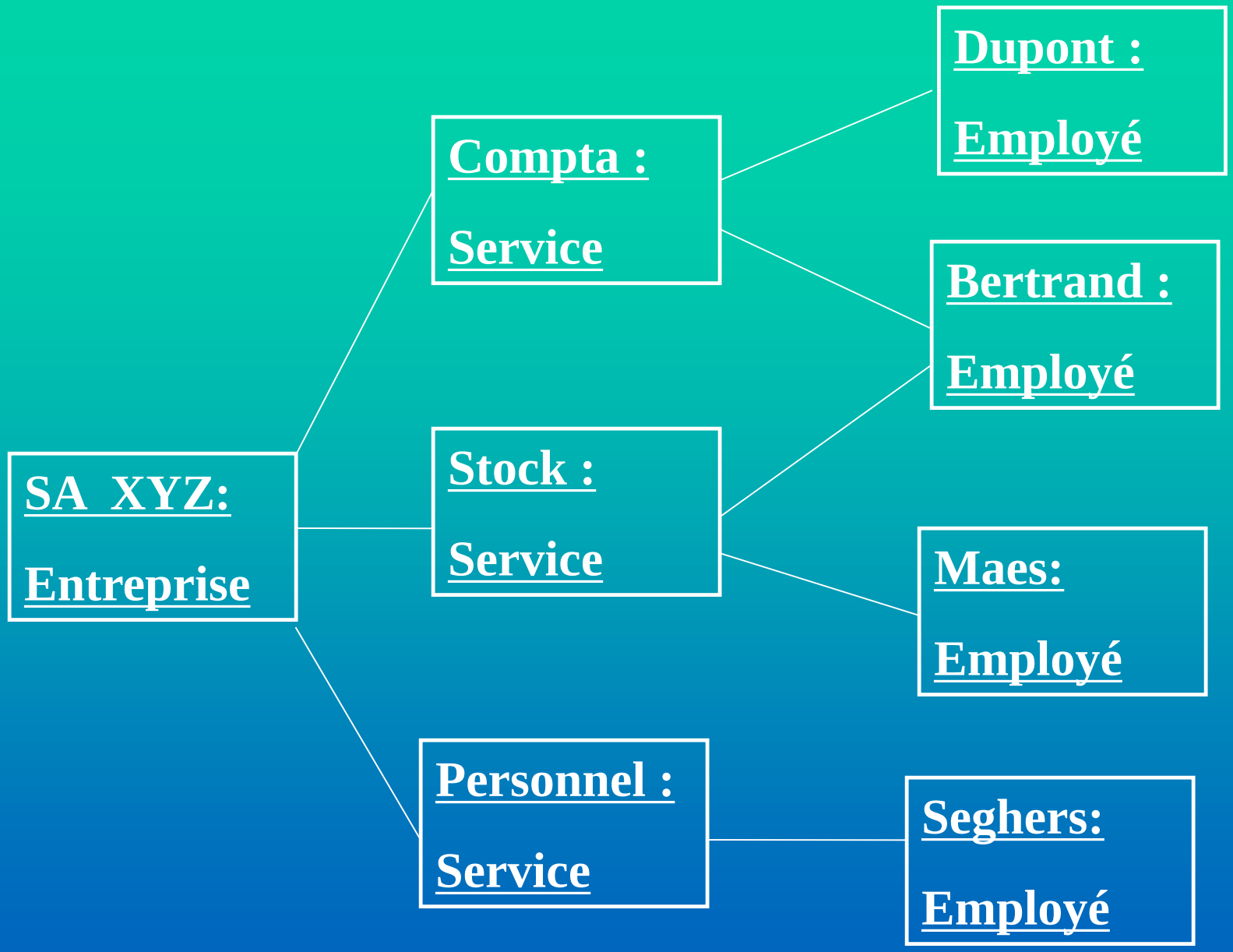
3 Correspondance entre diagrammes de classes et diagrammes d ’objets

Les diagrammes de classes et les diagrammes d ’objets appartiennent à deux vues complémentaires du modèle.

Le diagramme de classes montre une abstraction de la réalité tandis qu ’un diagramme d ’objets représente un cas particulier à un moment donné du système.

Les diagrammes d ’objets peuvent permettre de vérifier la pertinence du diagramme de classes ou de plus facilement modéliser celui-ci (le monde qui nous entoure est constitué d ’objets et pas de classes !).


+ Entreprise
1
1..*
+ Service
Comporter
1..*
Appartenir
1..*
+ Employé

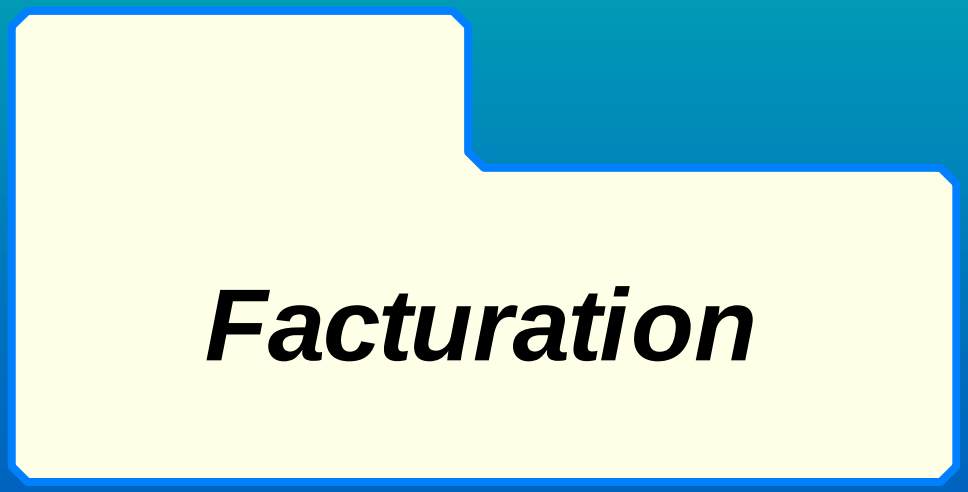

SA XYZ:
Entreprise
Compta :
Service
Stock :
Service
Personnel :
Service
Dupont :
Employé
Bertrand :
Employé
Maes:
Employé
Seghers:
Employé

- **chaque objet est une instance d'une classe et la classe de l'objet ne change pas durant la vie de l'objet**
- **les classes abstraites ne peuvent être instanciées**
- **chaque lien est une instance d'une relation (association simple, agrégation ou composition)**
- **les liens relient les objets et les associations relient les classes**
- **un lien entre deux objets implique obligatoirement une relation entre les classes (ou superclasses) des deux objets**
- **un lien entre deux objets signifie que les deux objets se connaissent et peuvent s'échanger des messages.**
- **Les diagrammes d'objets qui contiennent des objets et des liens sont des instances des diagrammes de classes qui contiennent des classes et des relations.**

4 Les paquetages (packages)

Les paquetages offrent un mécanisme général pour la partition des modèles et le regroupement des éléments de modélisation.

Un paquetage est représenté graphiquement par un dossier.



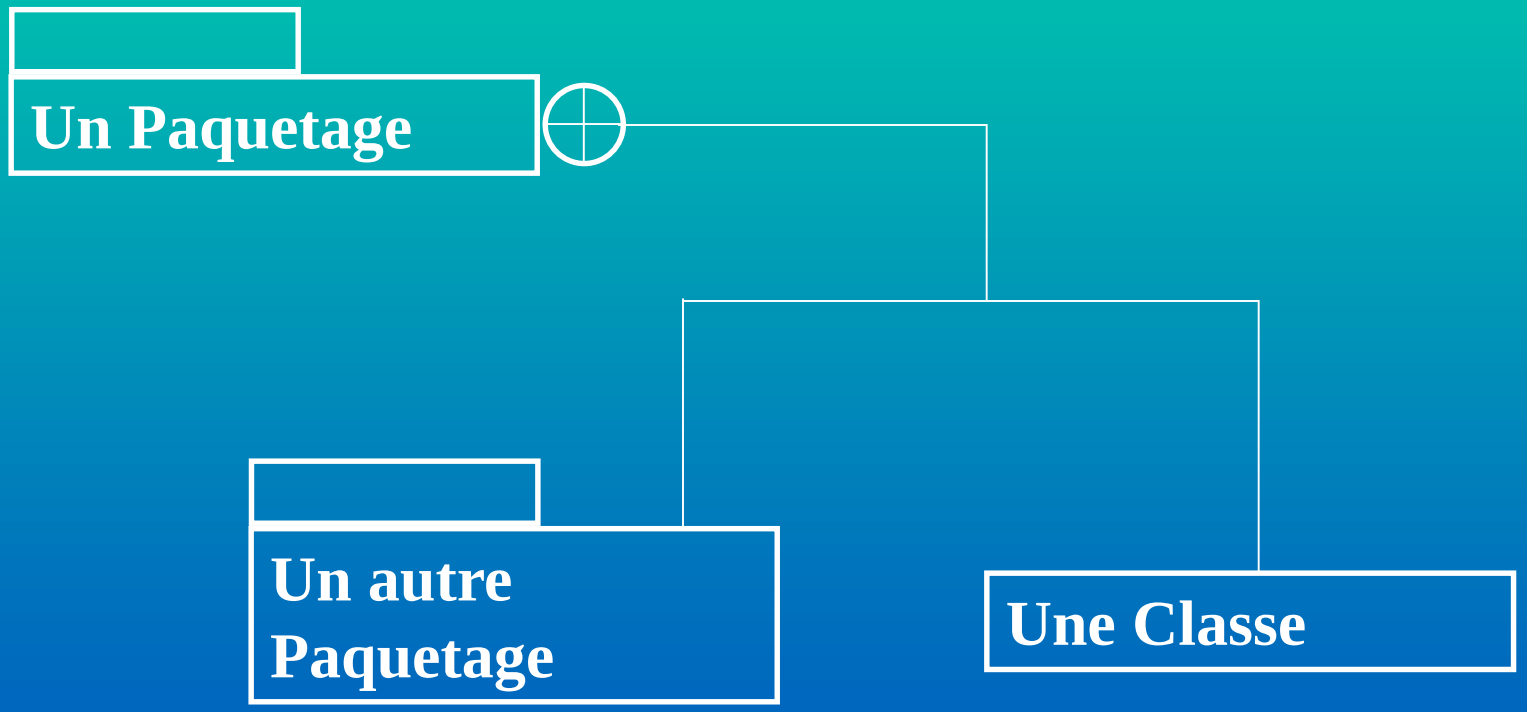
Les paquetages divisent et organisent les modèles comme les répertoires organisent les systèmes de fichiers.

Un élément de modélisation ne peut appartenir qu'à un seul paquetage et l'élément est détruit lors de la destruction du paquetage.

Chaque paquetage correspond à un sous-ensemble du modèle et peut contenir des classes, des objets, des relations, des composants, des cas d'utilisation, des nœuds, d'autres paquetages, …

La décomposition en paquetage est purement logique (et pas fonctionnelle). Il y a un cohérence forte entre les éléments du même paquetage et un couplage faible entre paquetages.

Graphiquement, le contenu d'un paquetage est placé dans le dossier qui représente le paquetage ou à l'extérieur du dossier en utilisant des lignes pour relier les éléments de modélisation du contenu au dossier du paquetage. On utilise alors le symbole ⊕ pour relier les éléments extérieurs au paquetage.



Chaque paquetage définit un espace de nommage.

Deux éléments de modélisation se trouvant dans deux paquetages différents peuvent porter le même nom.

Considérons l'élement Compte (nom simple) modélisé dans deux paquetages différents (noms complets) :

Facturation::Compte

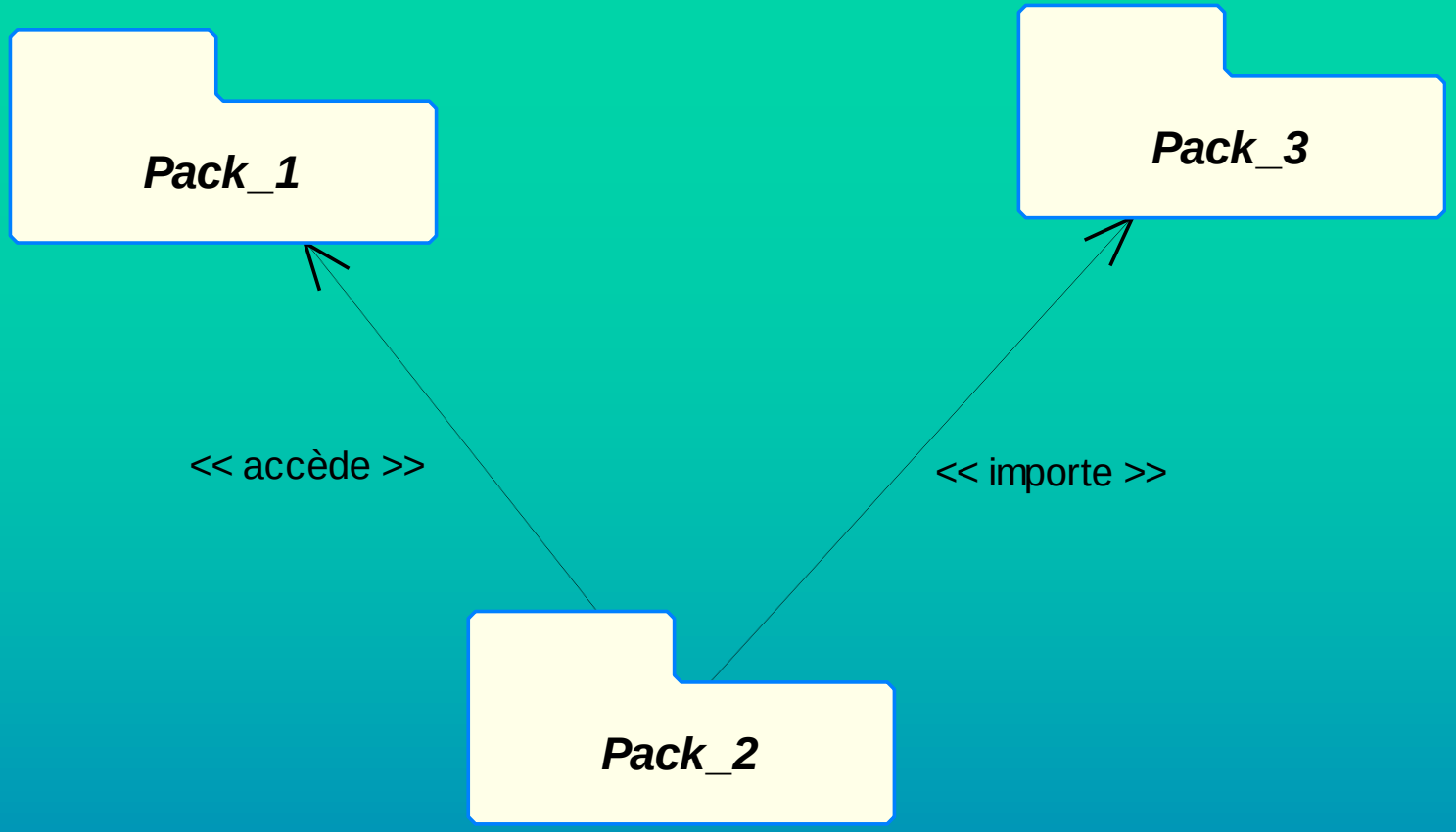
Banque::Compte

Les éléments contenus dans un paquetage emboîté voient les éléments contenus dans leur paquetage ou dans les paquetages englobant mais pas l'inverse.

Pour avoir accès à des éléments qui ne sont pas accessibles par défaut, il faut créer une relation de dépendance entre les paquetages concernés et pour autant que la visibilité de ces éléments soit publique.

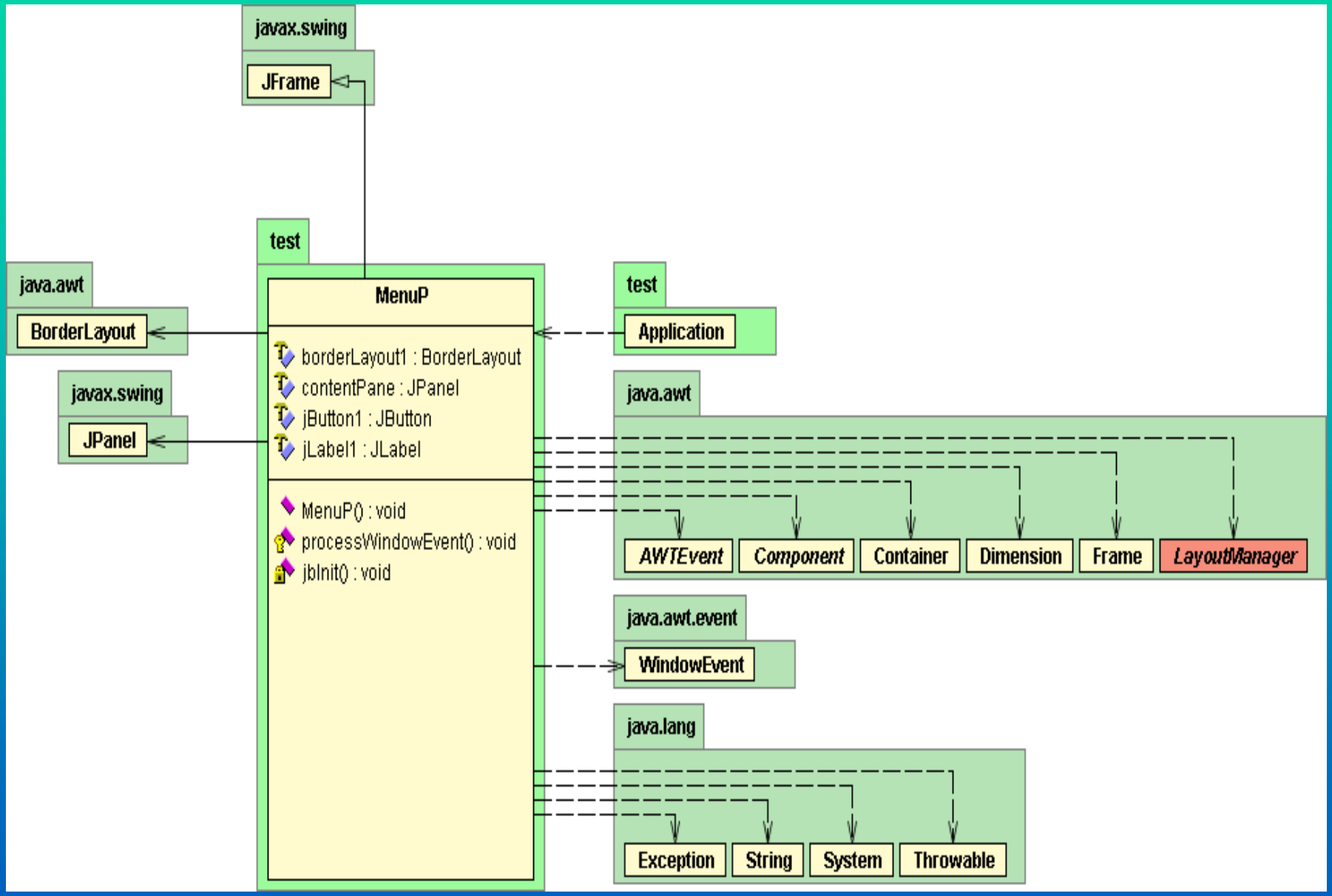
Une relation de dépendance doit exister dès que deux éléments issus de deux paquetages différents sont associés (sauf en cas de dépendance implicite : généralisation et emboîtement).

On peut associer les stéréotypes <<importe>> et <<accède>> à ces dépendances.



<<importe>> ajoute les éléments du paquetage destination à celui de la source de la dépendance. L'utilisation des noms simples peut suffire.

<<accède>> permet de référencer les éléments du paquetage destination en utilisant les noms complets


javax.swing
JFrame
test
java.awt
BorderLayout
MenuP
borderLayout1 : BorderLayout
contentPane : JPanel
jButton1 : JButton
jLabel1 : JLabel
MenuP() : void
processWindowEvent() : void
jbInit() : void
test
Application
javax.swing
JPanel
java.awt
AWTEvent
Component
Container
Dimension
Frame
LayoutManager
java.awt.event
WindowEvent
java.lang
Exception
String
System
Throwable

On peut définir des relations de généralisation entre paquetage. Les éléments publics et protégés sont alors accessibles dans le paquetage de spécialisation.