Analyse et Conception Objet de Logiciels

UML
Unified Modeling Language
Partie 1

UML (Unified Modeling Language)

UML : Langage de modélisation à objets

Notation graphique basée sur un ensemble de diagrammes

Permet de visualiser, spécifier, documenter les différentes parties d'un logiciel

UML n'est pas une méthode de développement, mais peut être utilisé dans tout le cycle de vie du logiciel

Permet de modéliser un système à différents niveaux de granularité

Outils pour passer automatiquement de la modélisation au programme (génération de code ou de squelettes de code)

Historique

Programmation objet

- 1967 : notion de classe pour implémenter des types abstraits (langage Simula)
- 1976 : principaux concepts de la programmation objet encapsulation, agrégation, héritage (Smalltalk)
- 1983 : C++
- 1986 : Eiffel
- 1995 : Java, Ada95
- 1998 : Python
- 2000 : C#

Méthodes de développement

Années 1970 : première méthodes fonctionnelles

Conception « top-down », basée sur le principe « diviser pour mieux régner »

Années 1980 : Approches systémiques, avec modélisation des données et des traitements (Merise)

1900-1995 : Méthodes objet (Booch, OMT, OOSE...)

1997 : UML 1.1

2005 : UML 2.0

Concepts « objets »

Idée principale des approches objet
Centraliser les données et les traitements dans une même unité, appelée *objet*

Objet, caractérisé par

- une identité
- un état (défini par la valeur de ses attributs)
- un comportement (défini par ses méthodes)

Classe : abstraction qui représente un ensemble d'objets de même nature

Un objet est une « instance » de sa classe

Concepts « objets »

Encapsulation : masquer certains détails d'implémentation (constituants *privés* d'un objet)

Agrégation : définir des objets composites, constitués d'objets plus simples

Extension : définir une classe *dérivée* à partir d'une classe existante.

- ajout d'attributs ou de méthodes
- définition de hiérarchies de classes

Héritage : une classe dérivée hérite des attributs et méthodes de sa classe mère

- évite la duplication de constituants
- encourage la réutilisation

UML : aspects fonctionnels

But : représenter les exigences fonctionnelles du système (fonctionnalités du logiciel)

Acteur

- Quelqu'un ou quelque chose qui interagit avec le système
- Peut être un utilisateur, un paramètre de l'environnement (grandeur physique), ou un autre système

Cas d'utilisation

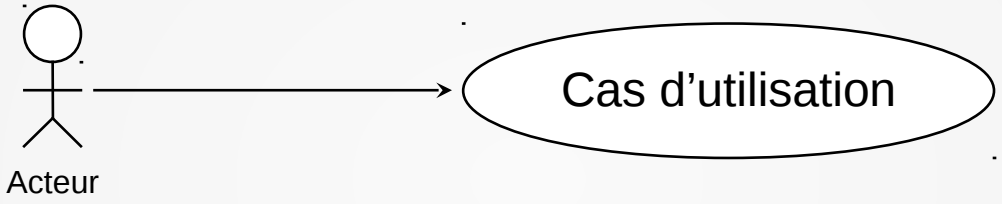
- Manière d'utiliser le système logiciel (représente une fonctionnalité)

→ **Diagramme de cas d'utilisation**

Diagramme de cas d'utilisation

Relation entre un acteur et un cas d'utilisation

- lorsqu'un acteur peut utiliser une fonctionnalité, il « déclenche » le cas d'utilisation



Spécialisation d'un acteur

- Toute instance de l'acteur enfant est une instance de l'acteur parent
- tout ce que peut réaliser l'acteur parent, l'acteur enfant peut le réaliser

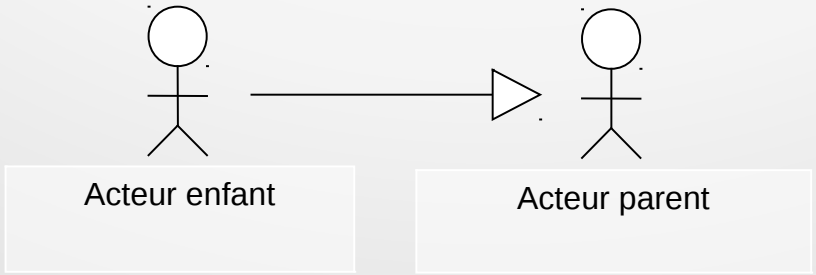


Diagramme de cas d’utilisation

Inclusion entre deux cas d’utilisation

- Pour réaliser le cas d’utilisation de base, il faut réaliser le cas d’utilisation inclus
- Permet de factoriser des comportements communs

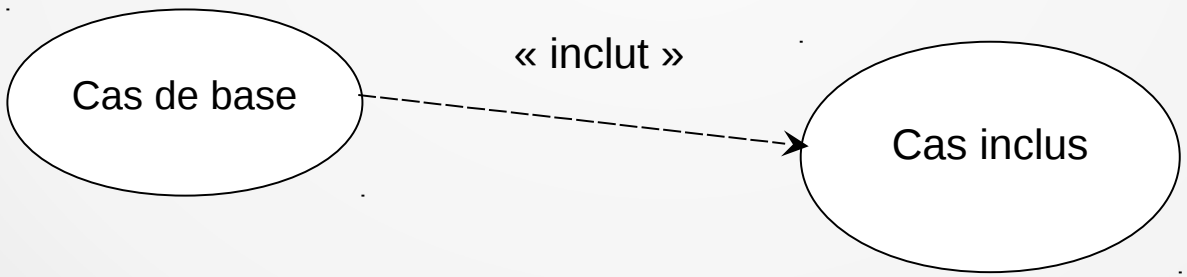


Diagramme de cas d’utilisation

Relation d’extension entre deux cas d’utilisation

- Lors de l’exécution du cas d’utilisation A, le cas d’utilisation B peut être exécuté
- Peut être utilisée pour traiter des cas particuliers ou des fonctions optionnelles

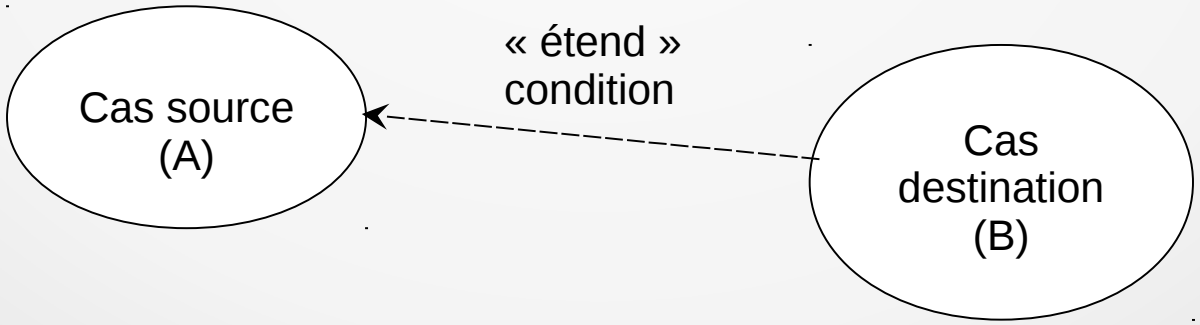


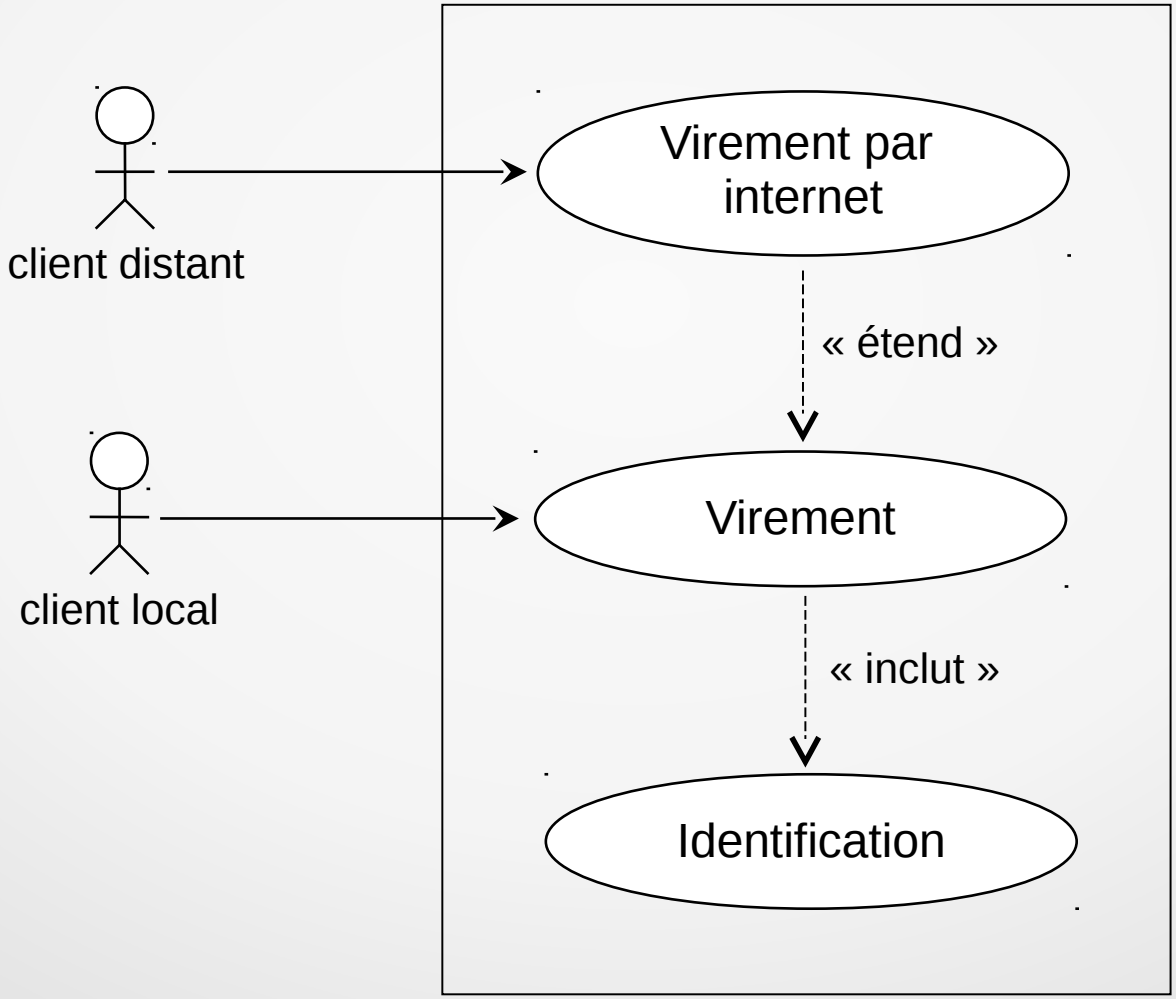
Diagramme de cas d’utilisation

Généralisation entre deux cas d’utilisation

- Le cas A généralise le cas B, ou le cas B spécialise le cas A
- L’exécution du cas B est une façon d’exécuter le cas plus général A

Exemple de diagramme de cas d'utilisation

Système de virement d'une banque



UML : aspects statiques

Principaux diagrammes : **diagramme de classes** et **diagramme d'objets**

Permet de représenter les différentes entités qui composent le système en termes de **classes** et de **relations** entre ces classes

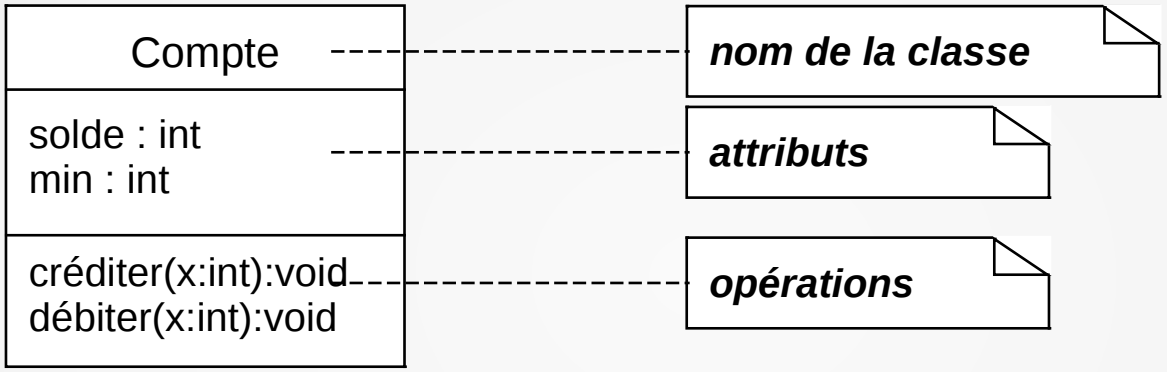
Classe : abstraction qui représente un ensemble d'**objets**

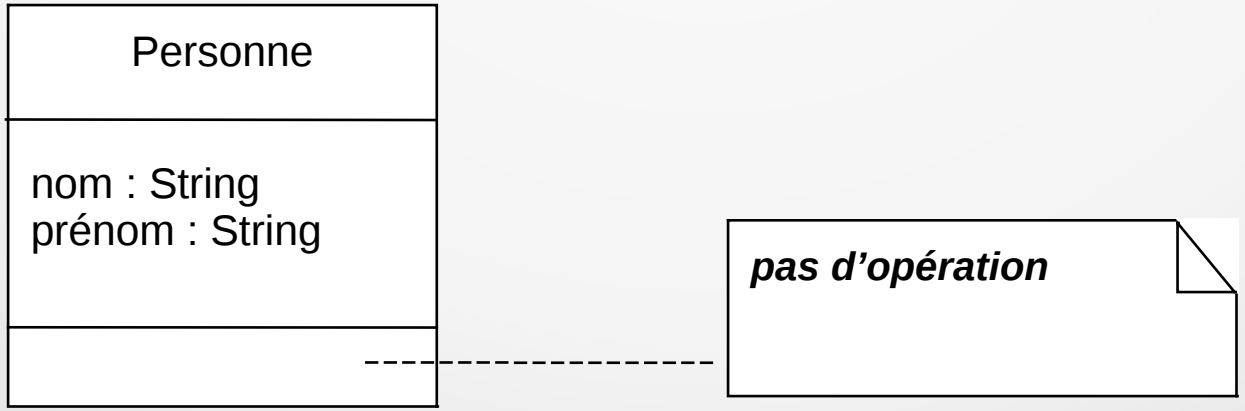
Une classe comporte

- un nom,
- des attributs,
- des opérations.

Exemples de classes

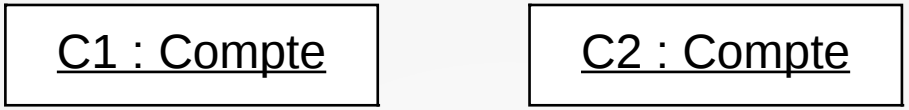
Classes Compte et Personne





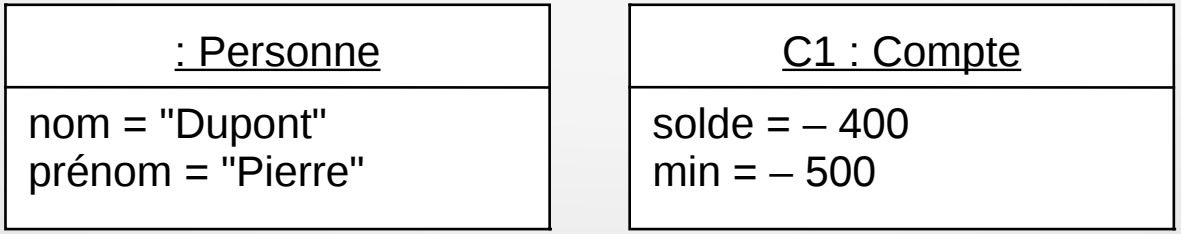
Exemples d’objets

Deux instances de la classe Compte



Trois instances de la classe Personne

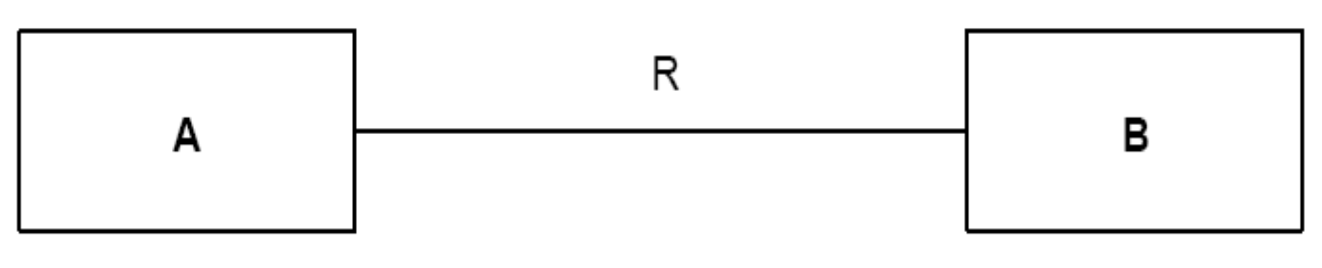
Instances avec valeurs d’attributs



Relations entre classes

Association : relation (au sens mathématique) entre *n* classes

Association binaire entre deux classes A et B : relation (au sens mathématique) entre les deux classes, c'est-à-dire un *sous-ensemble R* de $A \times B$



Si $(a, b) \in R$, on a un **lien** entre les objets *a* de la classe *A* et *b* de la classe *B*.

Remarque : pour une relation *R* donnée, il existe **au plus un lien** entre deux objets *a* et *b*.

Exemple de diagramme de classes

Association entre la classe Compte et la classe Personne

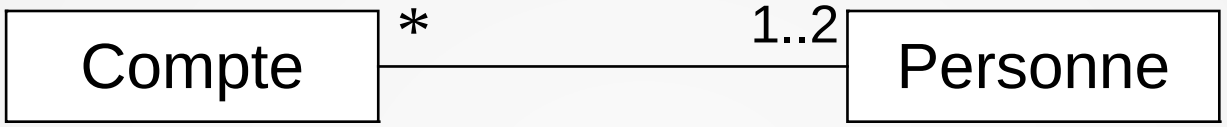


Diagramme de classes

Multiplicités :

- « * » : une personne peut posséder un nombre quelconque de comptes
- « 1..2 » : un compte est associé à une ou deux personnes

Exemple de diagramme d’objets

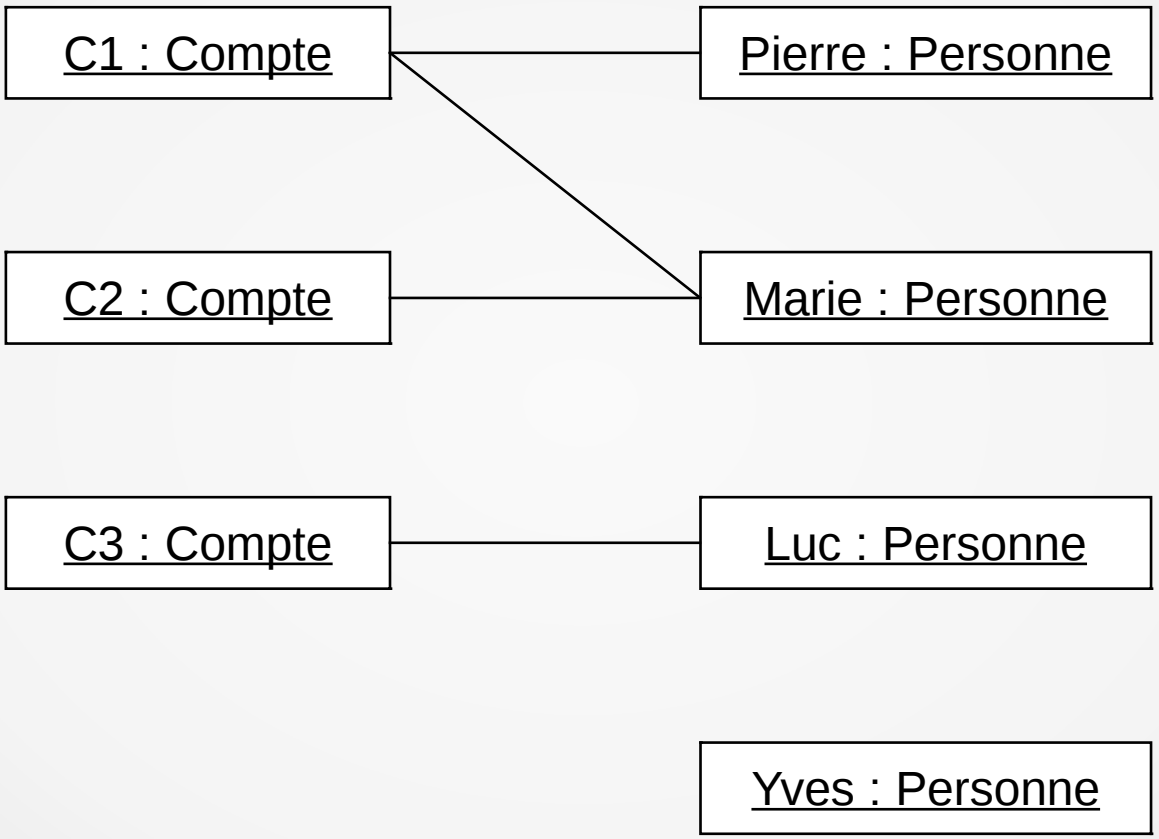
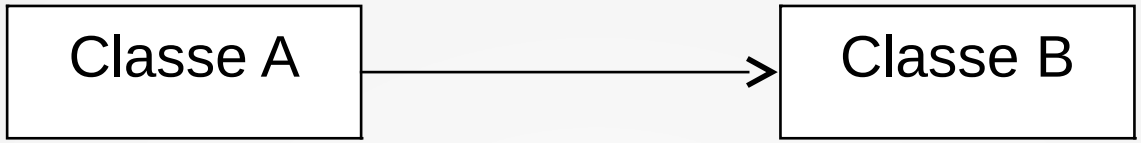


Diagramme d’objets, cohérent avec le diagramme de classes précédent

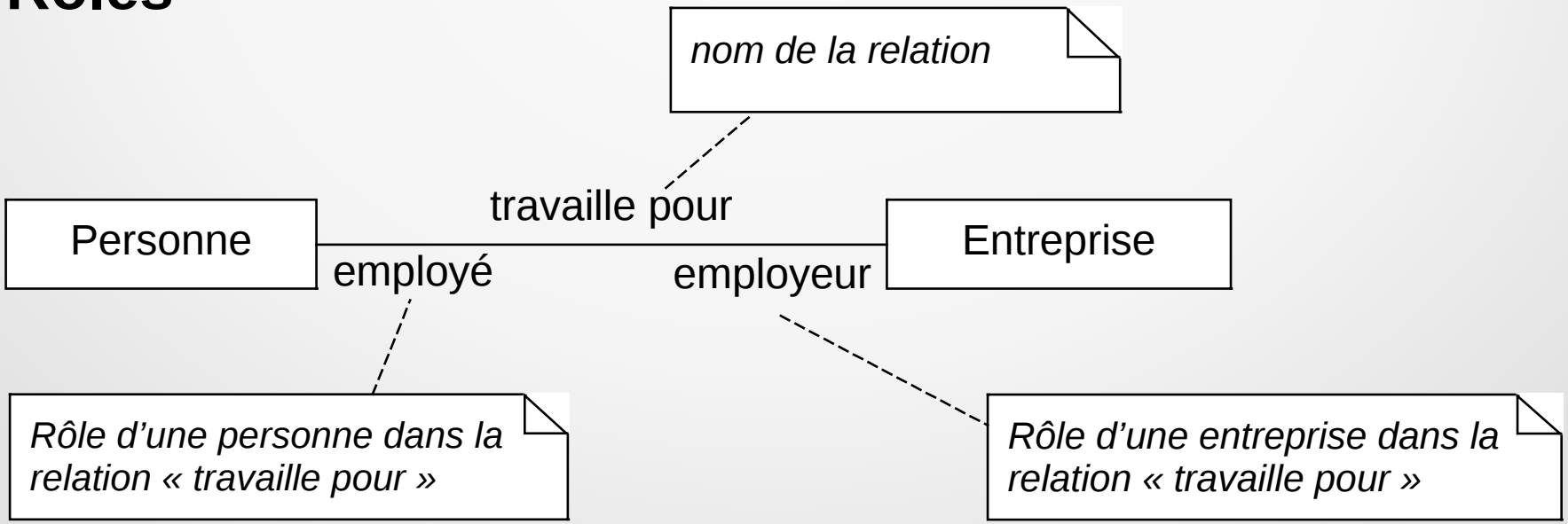
Navigabilité et rôles

Navigabilité



A partir d'un objet de A, on peut accéder aux objets associés de la classe B, mais pas l'inverse.

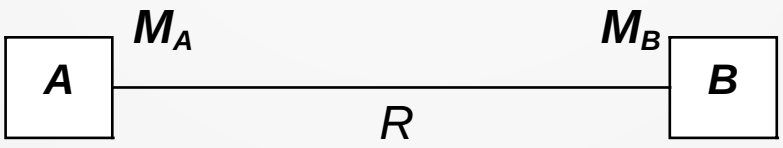
Rôles



Multiplicité : sous-ensemble de $\mathbb{N}$

syntaxe	sémantique
*	$\mathbb{N}$
m..n	$\{ x \in \mathbb{N} ; m \leq x \leq n \}$
m..*	$\{ x \in \mathbb{N} ; m \leq x \}$
m, n, p	$\{m, n, p\}$
M_1, M_2	$M_1 \cup M_2$

Relation *R* :

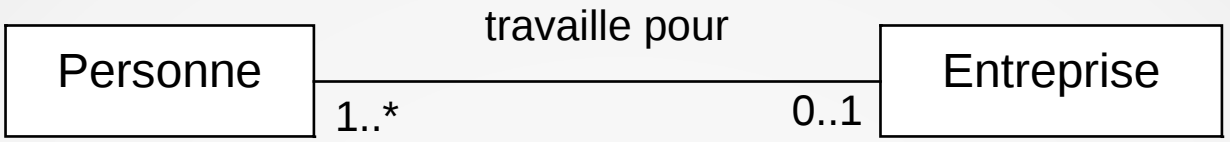


Pour chaque objet *b* de la classe *B*, le nombre d'objets de la classe *A* liés à *b* appartient à M_A : $\forall b \in B$, Card $\{(a, b) \in R ; a \in A\} \in M_A$.

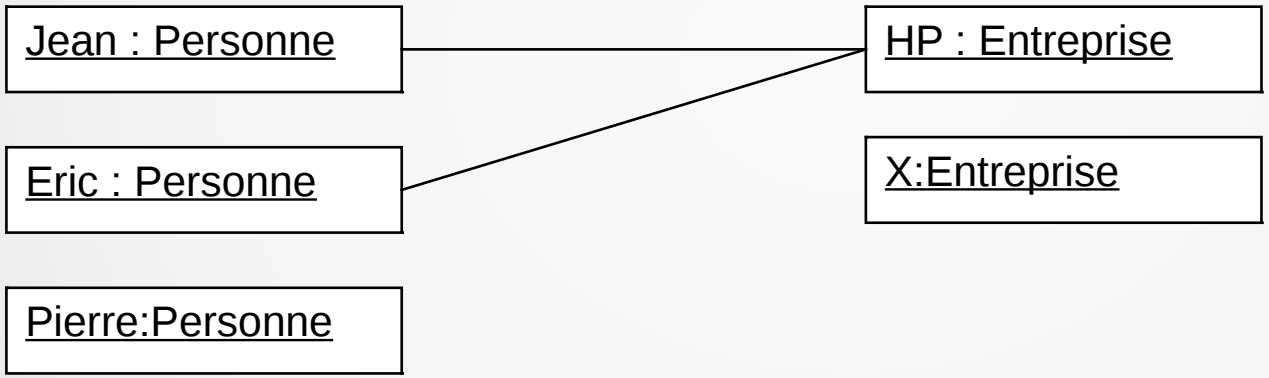
Pour chaque objet a de la classe A, le nombre d'objets de la classe B liés à a appartient à M_B : $\forall a \in A$, Card $\{(a, b) \in R ; b \in B\} \in M_B$.

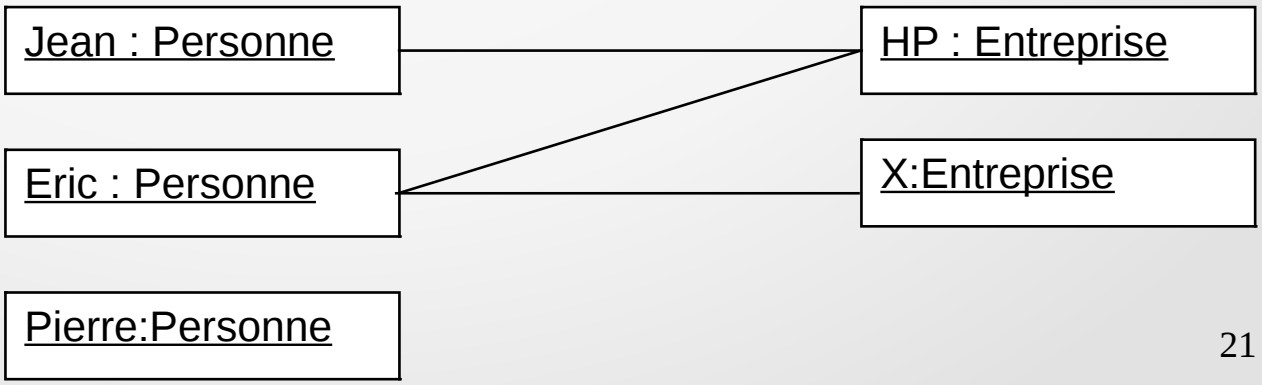
Cohérence des multiplicités

Soit l'association suivante :



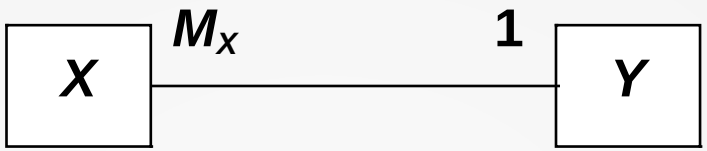
Les diagrammes d'objets suivants sont-ils cohérents ?





Associations particulières

Application de *X* vers *Y*



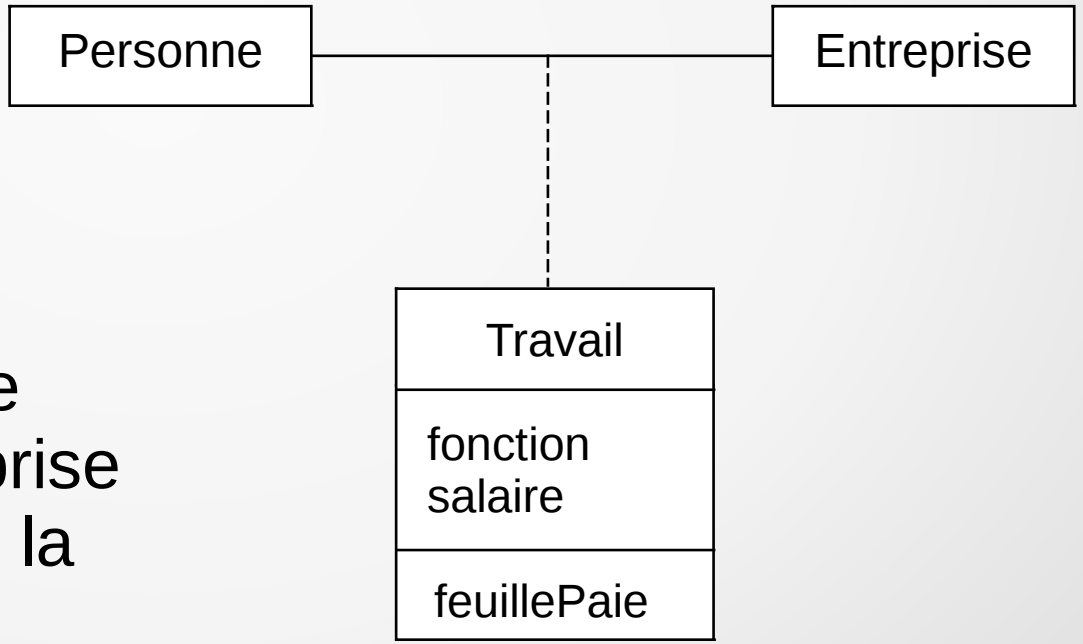
Fonction partielle de *X* vers *Y*

Isomorphisme entre *X* et *Y*

Classes associatives

On peut associer à chaque lien d'une association un objet d'une certaine classe, appelée *classe associative*, ou *classe-association.*

Exemple :



A chaque lien entre une personne et une entreprise est associé un objet de la classe Travail.

Simulation d’une classe associative

Exemple

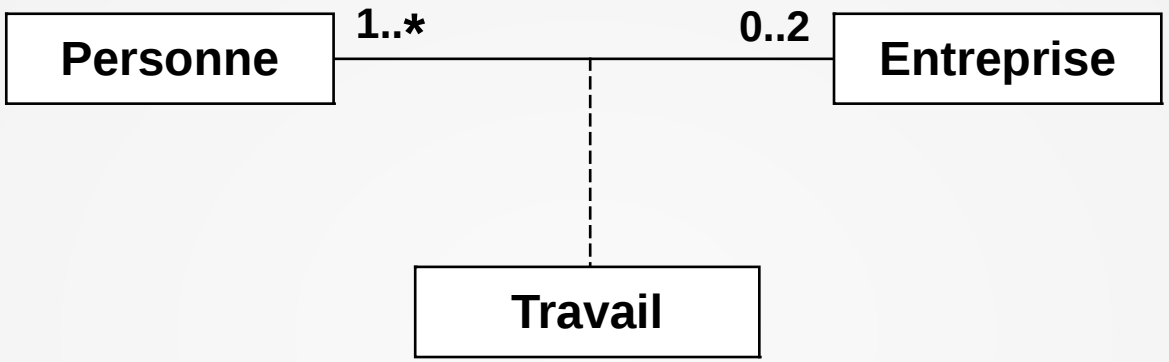


Diagramme de classes D_1

Simulation du diagramme D_1

Diagramme de classes D_2

Simulation d'une classe associative

Pour les diagrammes d'objets

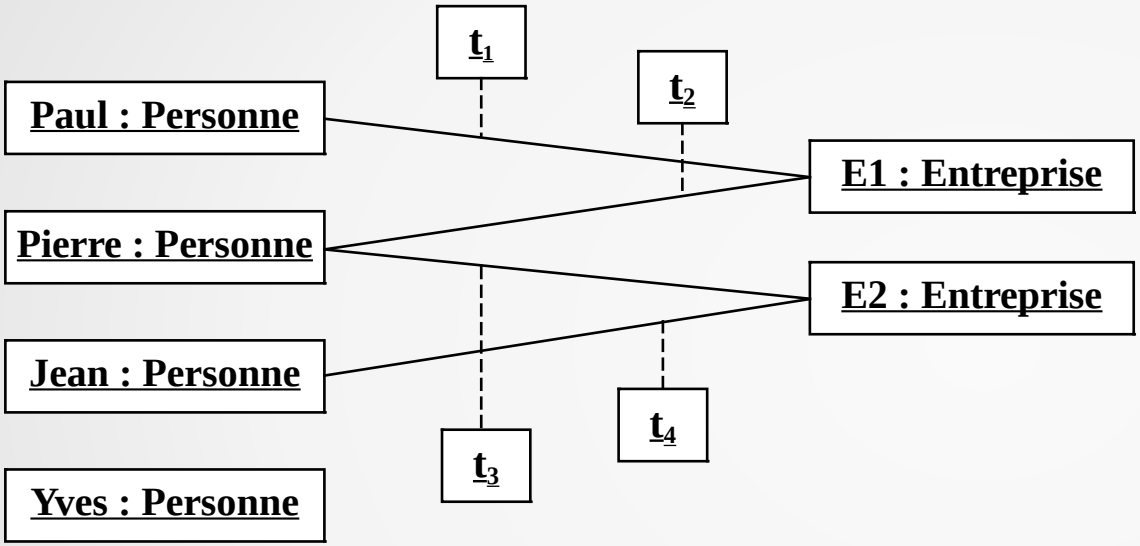


Diagramme d'objets correspondant au diagramme de classes D_1

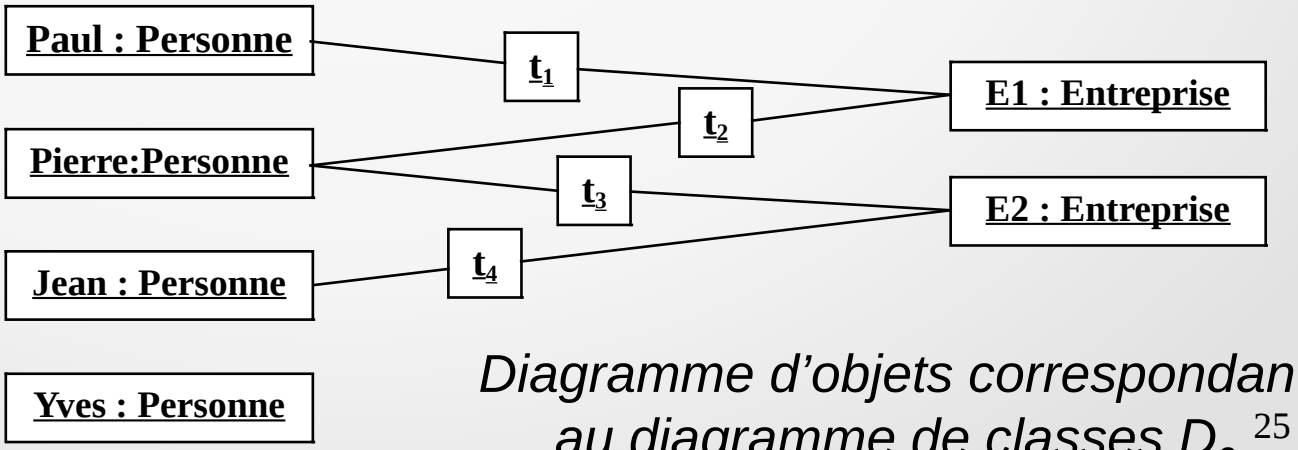
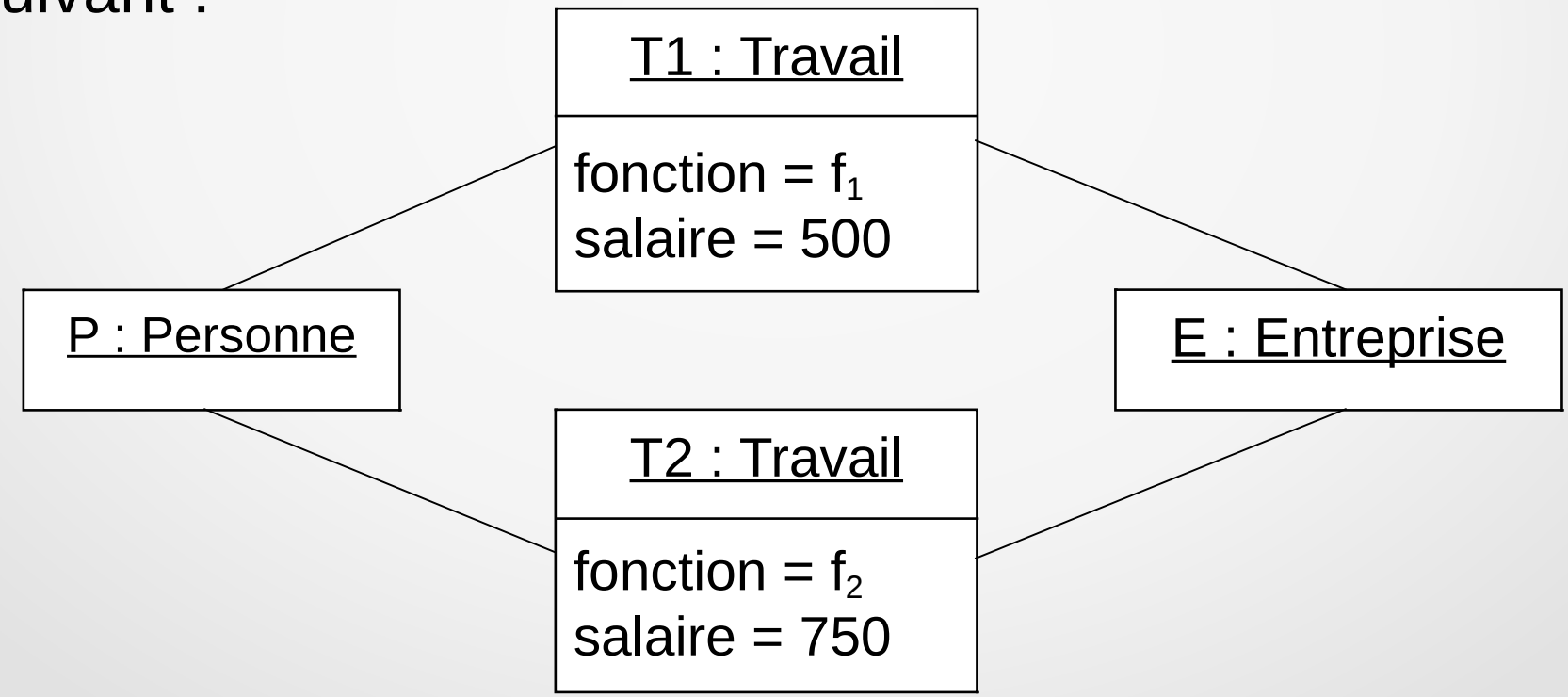


Diagramme d'objets correspondant au diagramme de classes D_2

« Simulation » et non « équivalence »

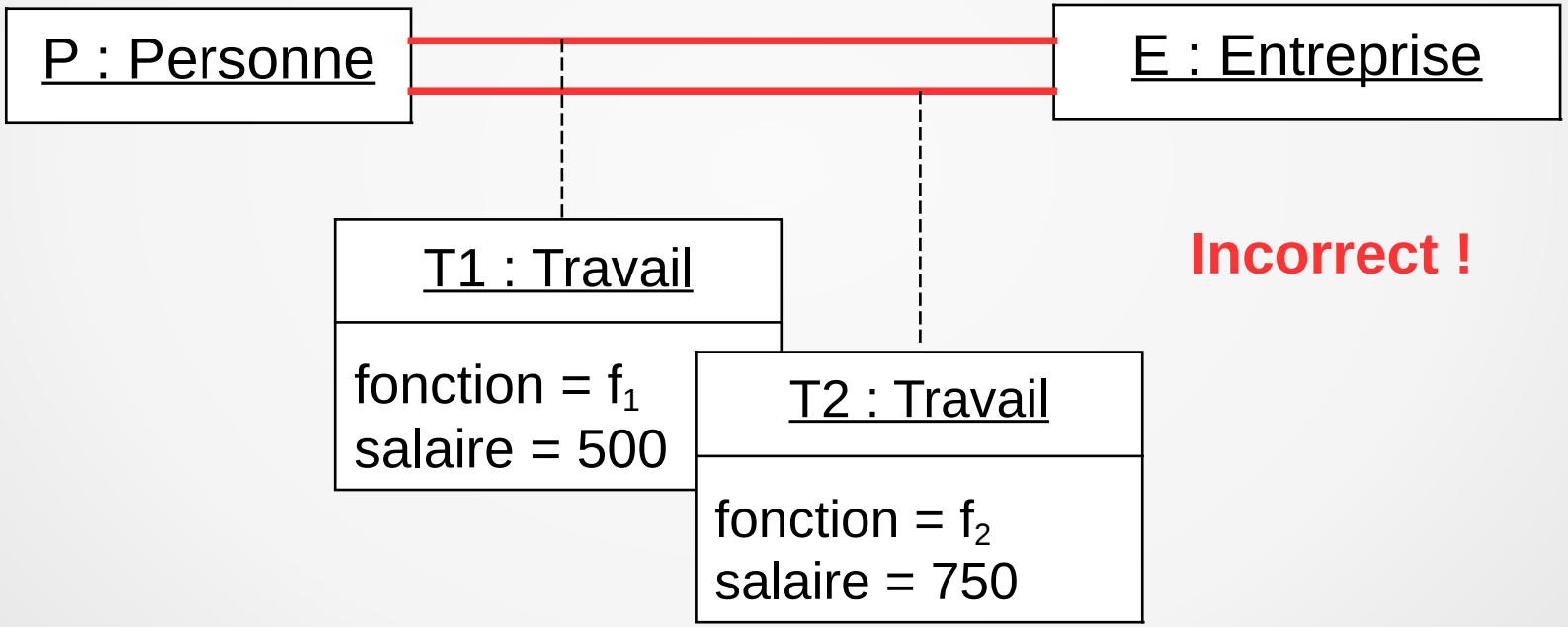
Remarque : il n'y a pas d'équivalence entre les deux diagrammes de classes D_1 et D_2.

En effet, le diagramme D_2 autorise le diagramme d'objets suivant :



« Simulation » et non « équivalence »

Mais le diagramme de classes D_1 interdit le diagramme d'objets correspondant, car il ne peut exister deux liens entre deux objets pour une relation donnée.



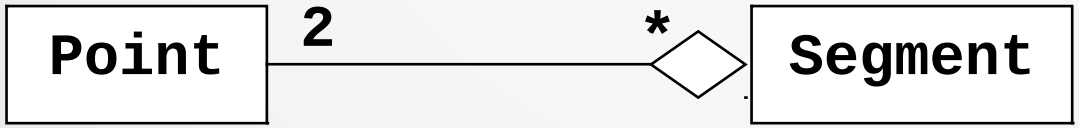
Avec D_1, une personne ne peut pas occuper deux emplois dans la même entreprise

Association binaire particulière : agrégation

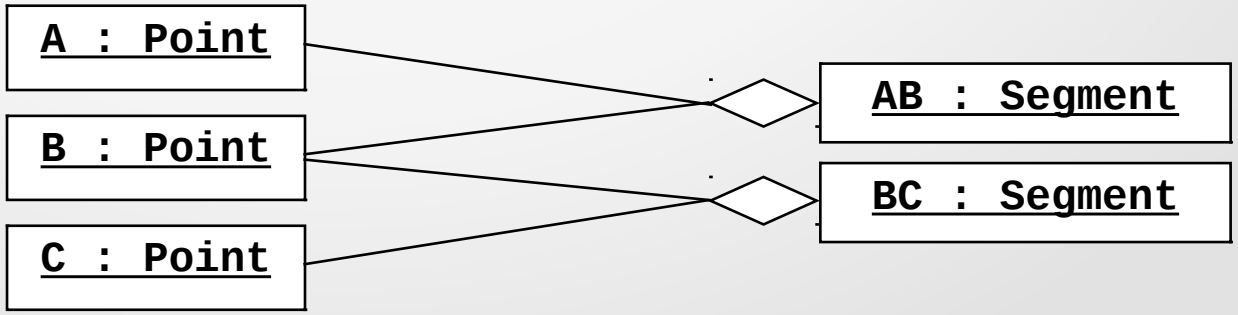
L'agrégation est une association binaire particulière entre un « tout » et une « partie » (cf. relation d'*appartenance*).

Un objet de la classe *B* comporte *n* objets de la classe *A*.

Exemple : spécification d'un segment



Un point peut appartenir à plusieurs segments



Agrégation

Avec une relation d'agrégation, les cycles sont *interdits* dans les *diagrammes d'objets* (cf. relation d'appartenance)

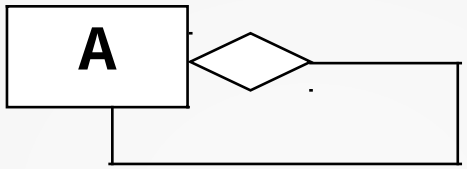
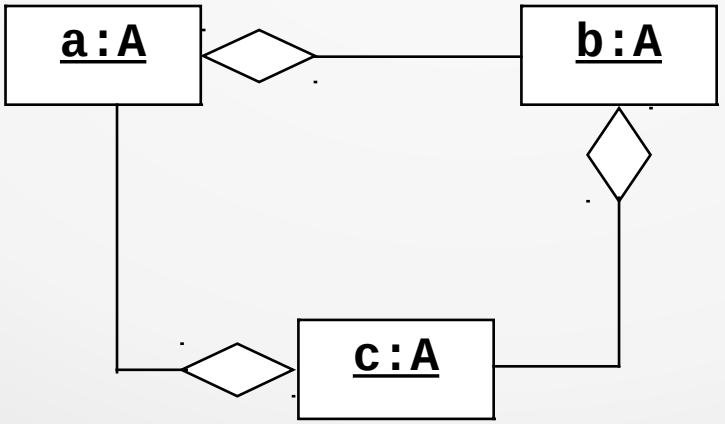


Diagramme de classes correct

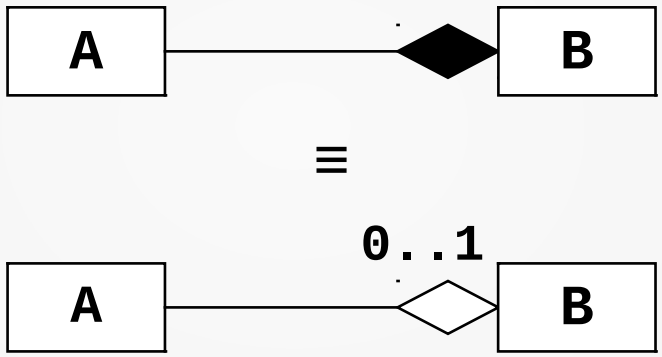


Cycles interdits dans les diagrammes d'objets

Diagramme de d'objets incorrect

Agrégation particulière : composition

La composition est une agrégation particulière, où une partie ne peut pas être partagée entre deux agrégats : la multiplicité du coté de l'agrégat est 0 ou 1.

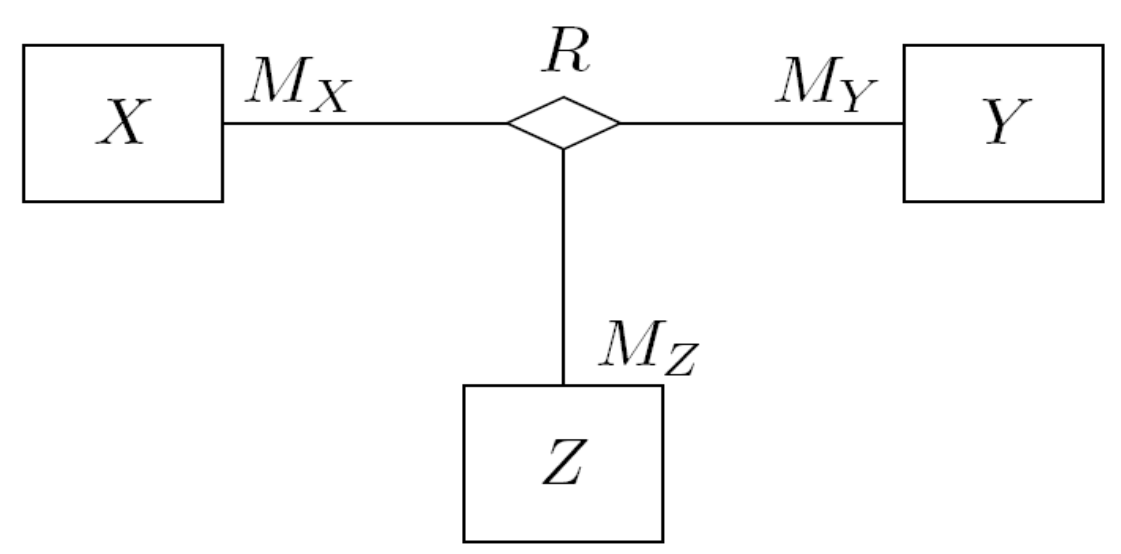


Exemples : une voiture et ses quatre roues, une table et ses pieds...

Contre-exemple : une porte de communication entre deux salles de cours

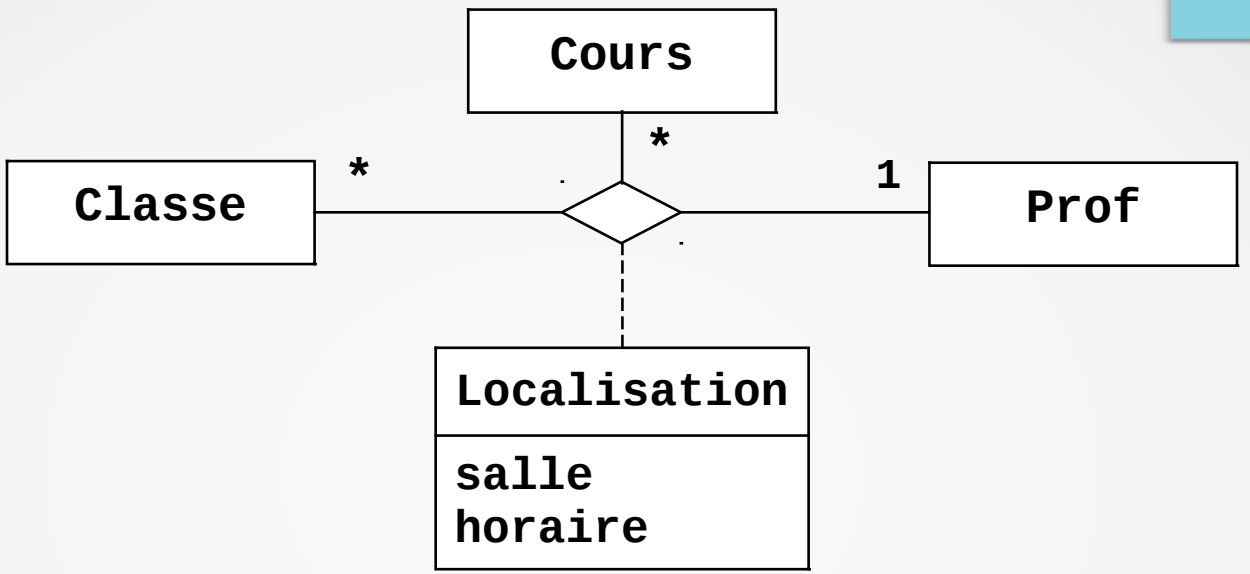
Association *n*-aire

Relation *n*-aire entre *n* classes
Exemple : relation ternaire entre *X*, *Y* et *Z*



- $\forall y_0 \in Y,\ \forall z_0 \in Z,\ \mathrm{Card}\{(x, y_0, z_0) \in R\ ;\ x \in X\} \in M_X$;
- $\forall x_0 \in X,\ \forall z_0 \in Z,\ \mathrm{Card}\{(x_0, y, z_0) \in R\ ;\ y \in Y\} \in M_Y$;
- $\forall x_0 \in X,\ \forall y_0 \in Y,\ \mathrm{Card}\{(x_0, y_0, z) \in R\ ;\ z \in Z\} \in M_Z$;

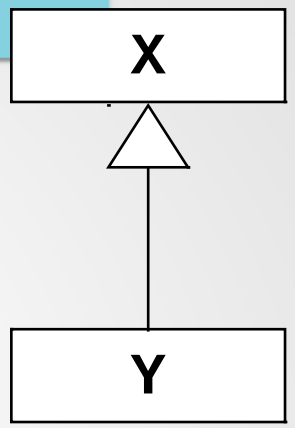
Exemple d’association ternaire



Multiplicités

- Un prof peut faire un même cours à plusieurs classes ;
- dans une classe, un cours est fait par un seul prof ;
- un prof peut faire plusieurs cours dans une même classe (un prof peut donc enseigner plusieurs matières) ;
- à chaque lien entre une classe, un cours et un prof correspond une localisation (une salle et un horaire).

Extension - Héritage



Extension : la classe Y est dérivée de la classe X

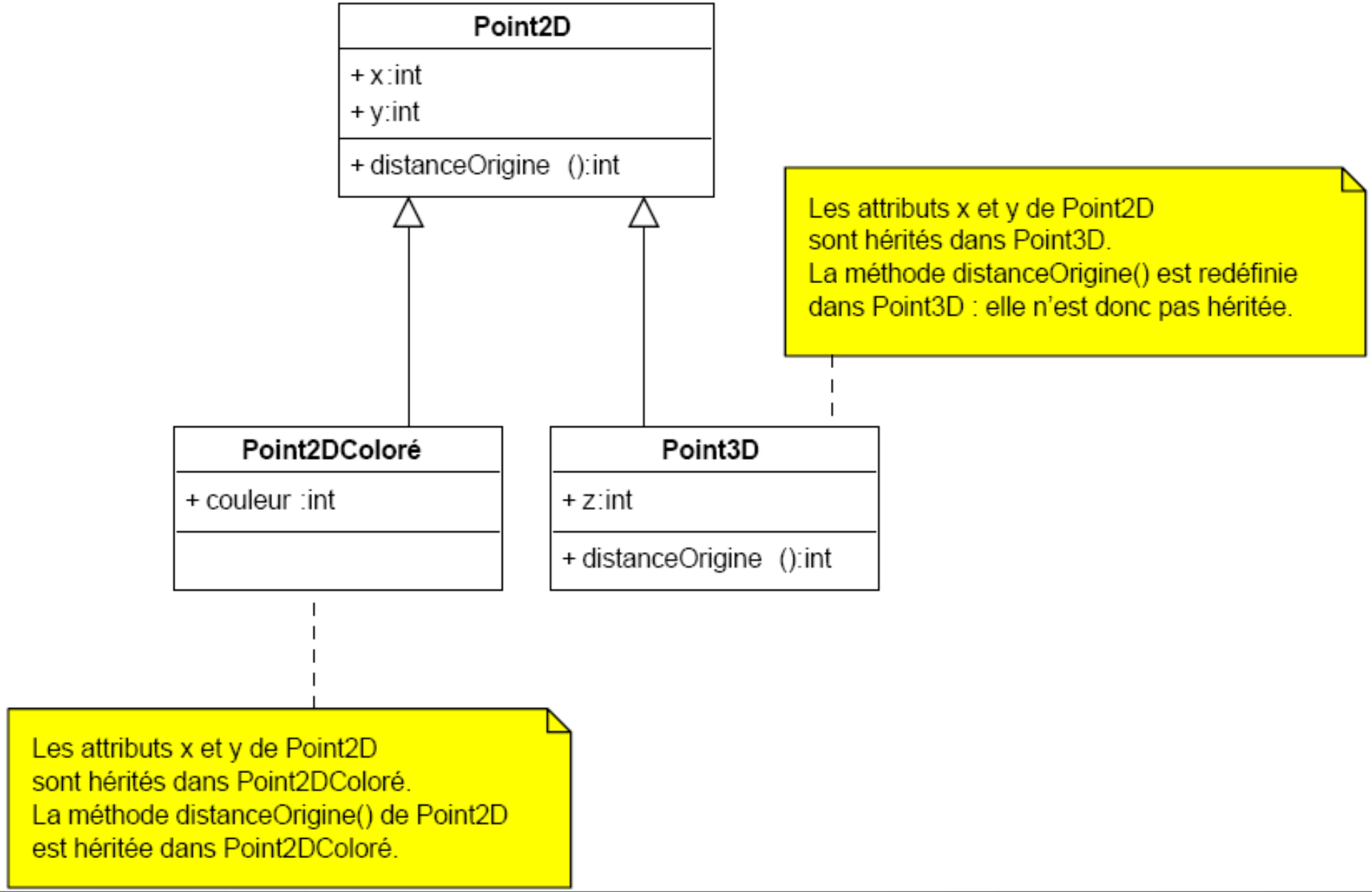
- La classe Y étend la classe X
- Y spécialise X, ou X généralise Y

Propriété d'héritage : une classe fille hérite des attributs, opérations *et associations* de sa classe mère

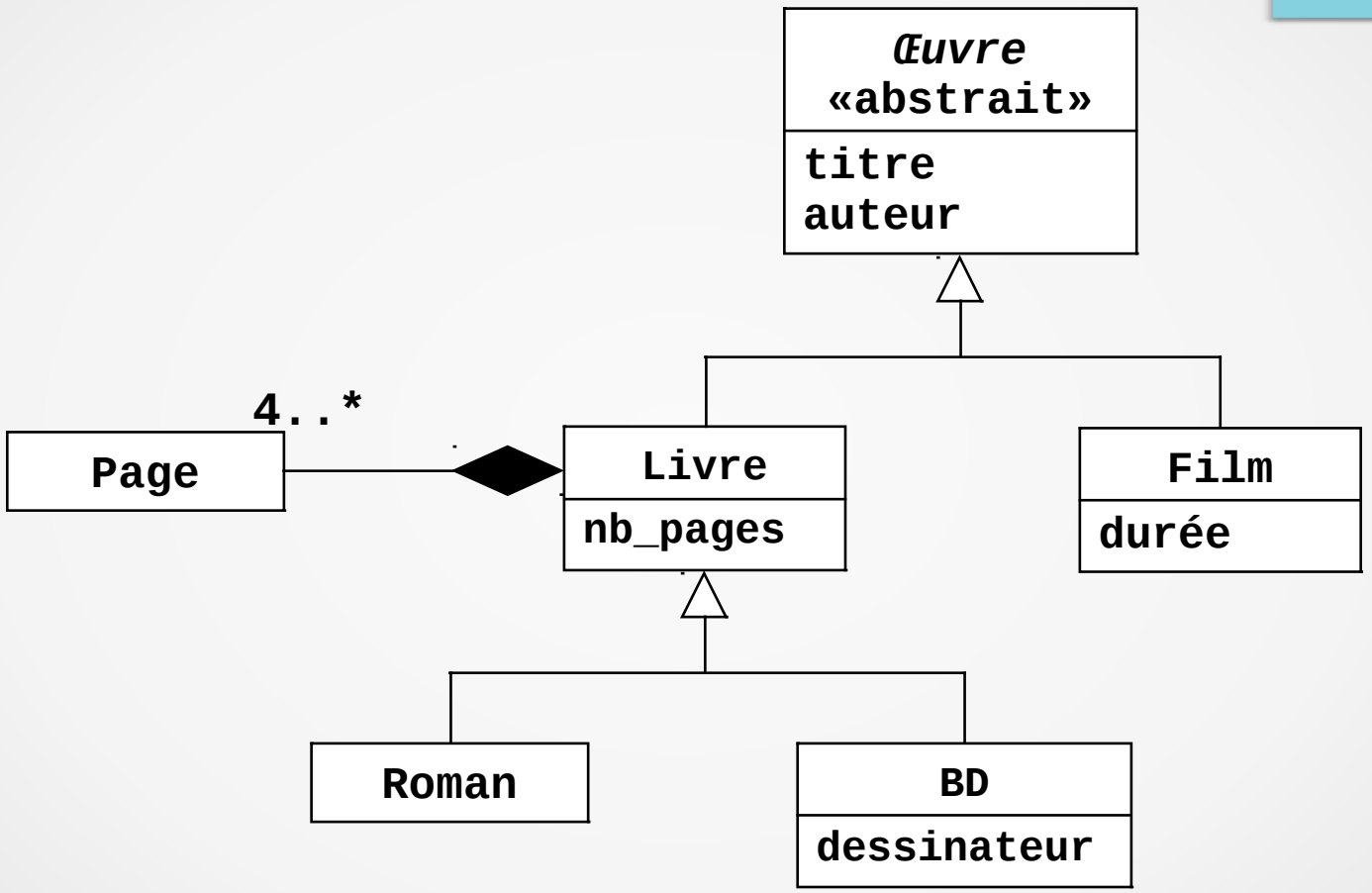
Propriété de substitution : tout objet de la classe fille peut être utilisé à la place d'un objet de sa classe mère

L'extension permet de gérer la complexité en organisant les classes sous forme de *hiérarchies* de classes. Elle permet également la *réutilisation*.

Exemple 1


Point2D
+ x:int
+ y:int
+ distanceOrigine ():int
Les attributs x et y de Point2D
sont hérités dans Point3D.
La méthode distanceOrigine() est redéfinie
dans Point3D : elle n'est donc pas héritée.
Point2DColoré
+ couleur :int
Point3D
+ z:int
+ distanceOrigine ():int
Les attributs x et y de Point2D
sont hérités dans Point2DColoré.
La méthode distanceOrigine() de Point2D
est héritée dans Point2DColoré.

Exemple 2



Un livre est composé de pages, donc un roman et une BD sont également composés de pages

Classes abstraites

Classe abstraite : classe qui ne peut pas être instanciée, et peut contenir des opérations abstraites

Opération abstraite : opération non implémentée, à laquelle n'est associé aucun code

Intérêt des classes abstraites :

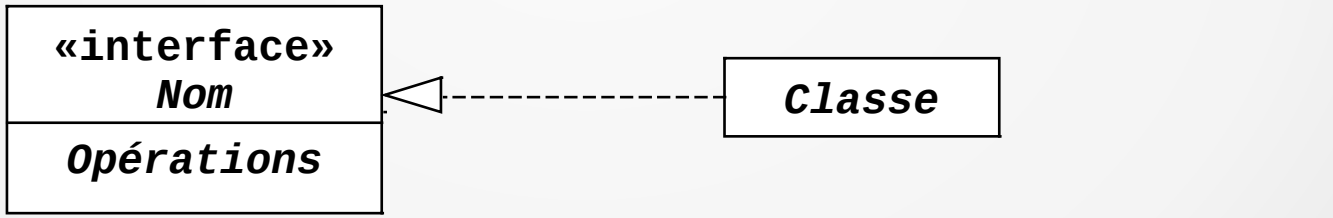
- regrouper des attributs et opérations (abstraites ou concrètes) communs à plusieurs classes dans les hiérarchies de classes (factorisation) ;
- en programmation objet, une classe concrète doit implémenter les opérations abstraites dont elle hérite ;
- Œuvre est une classe abstraite, elle ne peut donc pas être instanciée ;
- Livre est une classe concrète, ce qui permet de l'instancier pour des livres qui ne sont ni des romans, ni des BD.

Interfaces

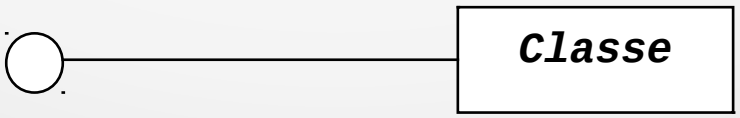
Une interface spécifie un ensemble d'opérations qui constituent un service.

Une interface ne contient que des opérations abstraites.

Une classe implémente une interface lorsqu'elle fournit une implémentation pour toutes ses opérations.

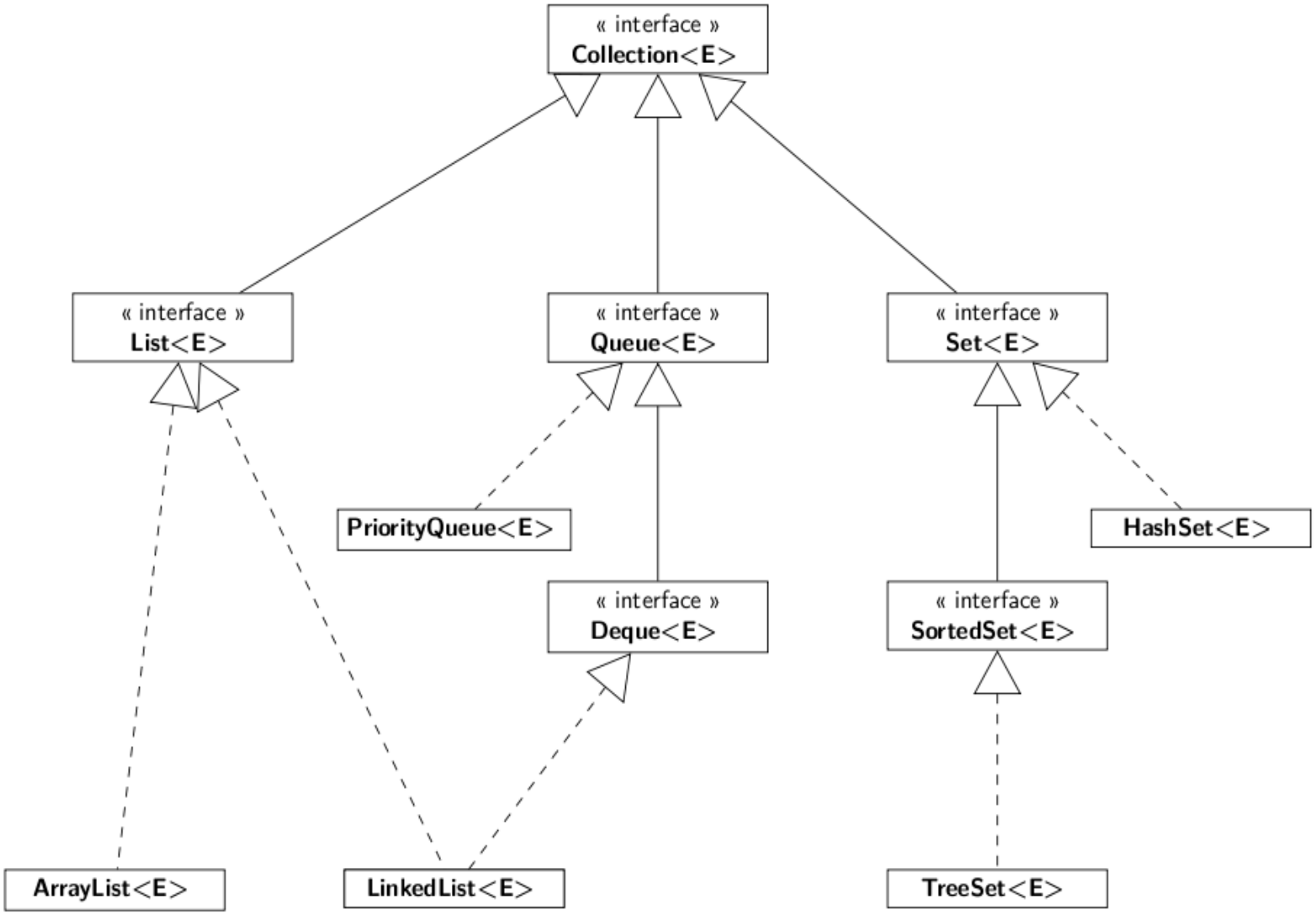


Classe qui implémente l'interface



Nom de l'interface

Interfaces : exemple des collections Java



Interfaces : exemple des collections Java

Intérêt : définir des opérations qui s'appliquent à :

- une collection quelconque (interface Collection),
- un ensemble quelconque (interface Set),
- une liste quelconque (interface List).

Les classes ArrayList et LinkedList implémentent l'interface List.

De même les classes HashSet et TreeSet implémentent l'interface Set.

On peut ensuite très facilement, pour des raisons d'efficacité, changer d'implémentation.