

- 3.2 -

Modélisation structurelle (I)

Exemples du cours

Une partie inspirés de

- Object-Oriented Software Engineering: Practical Software Development using UML and Java - Timothy C. Lethbridge, Robert Laganière
- UML Distilled: A Brief Guide to the Standard Object Modeling Language, 3rd ed. - Martin Fowler
- UML 2 - de l'apprentissage à la pratique - Laurent Audibert, Ellipses, 2009
- UML 2 par la pratique - Pascal Roques, Eyrolles, 2009

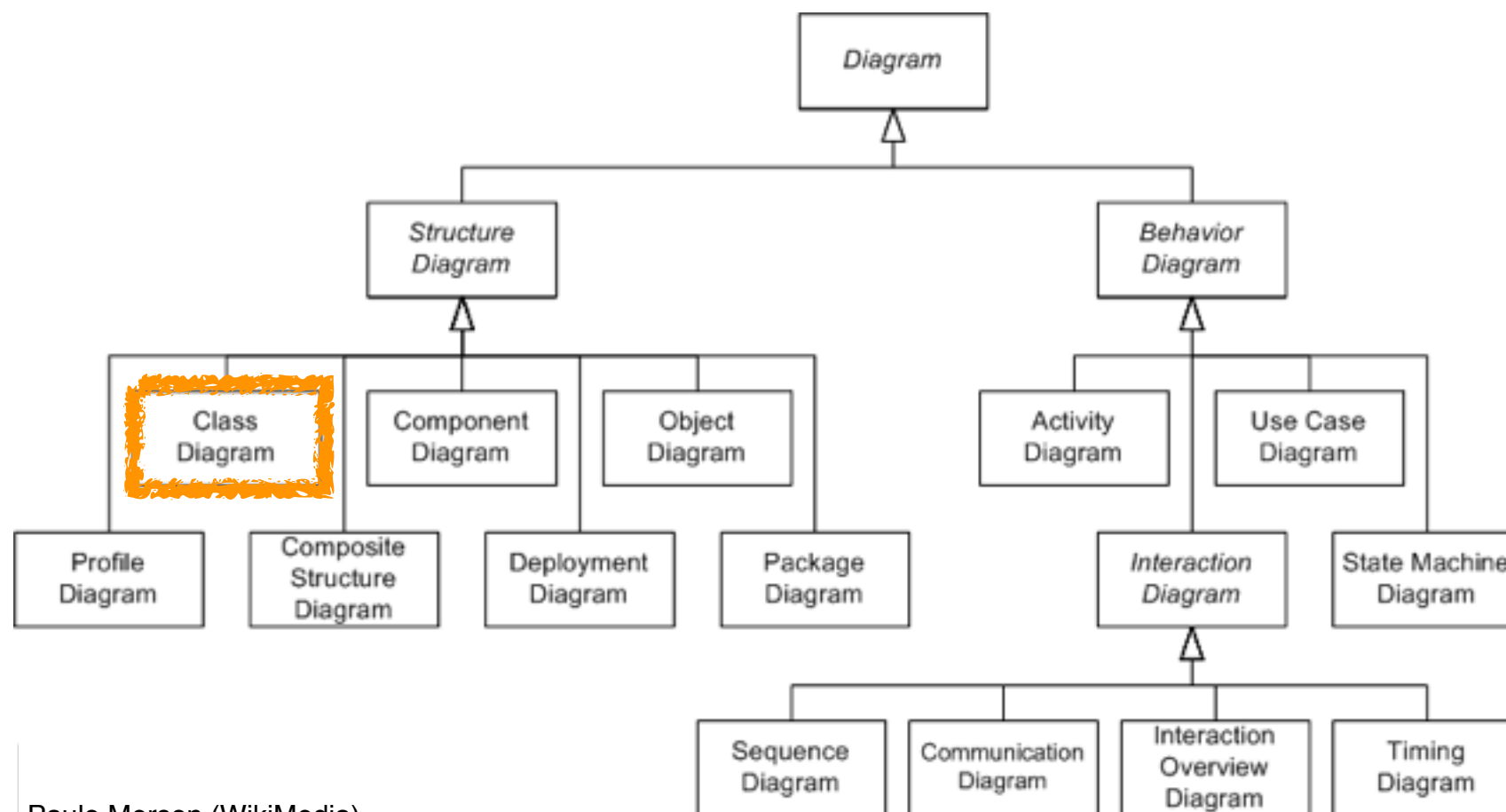
Perspectives

- *externe*
 - ▶ modéliser le contexte (l'environnement)
- *interaction*
 - ▶ modéliser les interactions avec l'environnement et entre les composants
- *structurelle*
 - ▶ modéliser l'architecture et les données
- *comportementale*
 - ▶ modéliser la dynamique

Modèles structurels

- organisation d'un système
 - ▶ composants
 - ▶ relations entre composants

Modélisation avec des diagrammes de classe



Paulo Merson (WikiMedia)

Diagrammes de classe

- utilisés pour développer un modèle orienté-objet du système
 - ▶ modéliser les classes
 - ▶ modéliser les relations entre classes

Éléments des diagrammes de classe

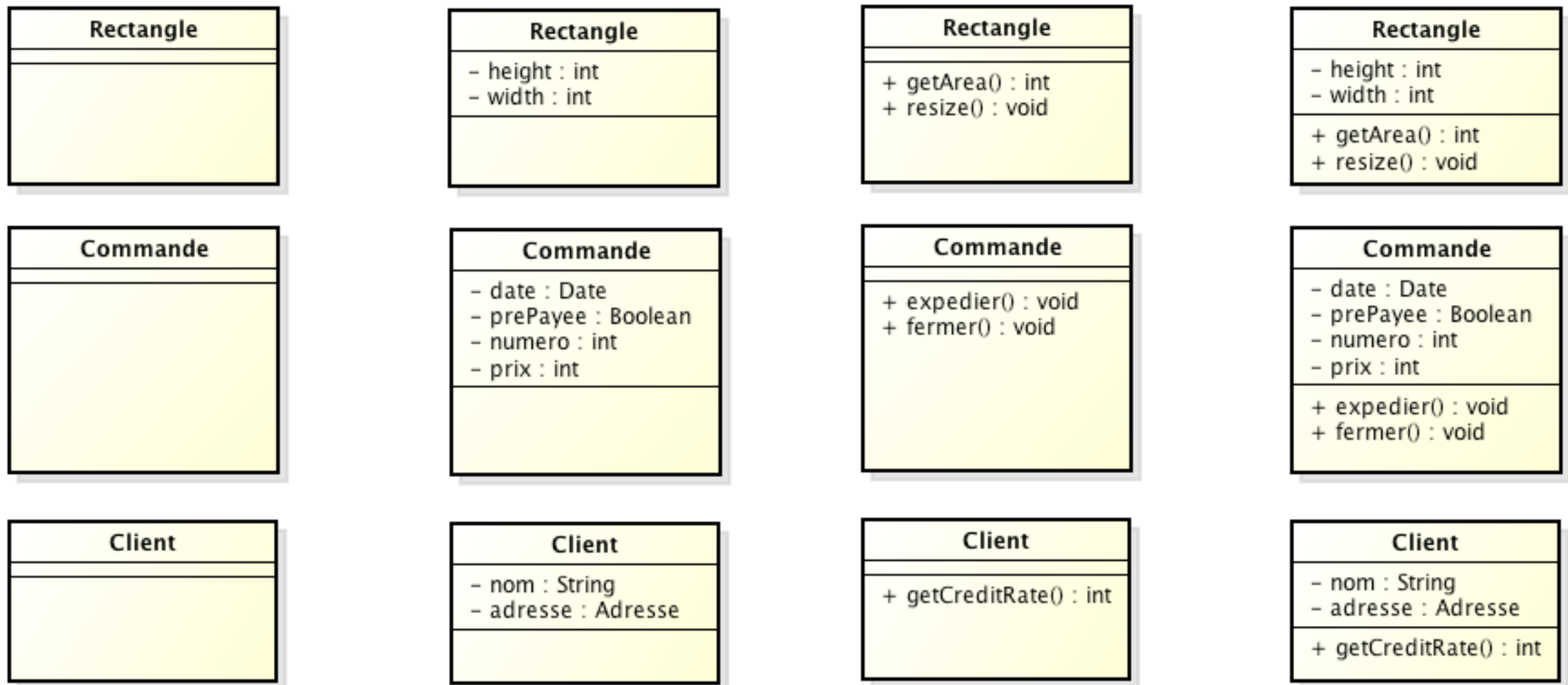
- **Classes**
 - ▶ les types de données disponibles
- **Attributs**
 - ▶ données se trouvant dans les classes et leurs instances
- **Opérations**
 - ▶ les fonctions exécutées par les classes et leurs instances

Éléments des diagrammes de classe

- **Associations**
 - ▶ les liens entre les instances des classes
- **Généralisations**
 - ▶ groupant les classes en hiérarchies d'héritage

Les classes

- Représentées à l'aide d'une boîte comprenant le nom de la classe



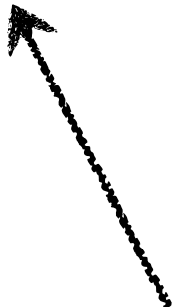
- Le diagramme peut aussi montrer les *attributs* et les *opérations*

Les attributs

- Visibilité
- Type
- Cardinalité
- Valeur par défaut
- Autres propriétés

{readOnly}, {union},
{subsets prop-name},
{redefines prop-name},
{ordered}, {unique},
prop-constraint

$\pm$ / **nom** : type [multiplicity]= val {prop}



➔ attributs dérivés /

➔ attributs de classe **nom**

Niveaux de visibilité

- + *public* : visible pour tous
- # *protected* : visible pour toutes les sous-classes
- *private* : visible pour la classe
- ~ *package* : visible seulement dans le package

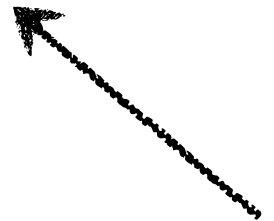
Les opérations

Une opération représente un service pouvant être demandé à n'importe quelle instance de la classe

- modifier l'état de l'objet
- modifier les liens avec les autres objets

Les opération

{*redefines* prop-name}, {*query*},
{*ordered*}, {*unique*}, oper-constraint



$\pm$ **name**(param: pType...): returnType {prop}

Syntaxe des paramètres

direction nom : type [multiplicity]=val

- Direction = *in*, *out*, *inout*, *return*

Exemples

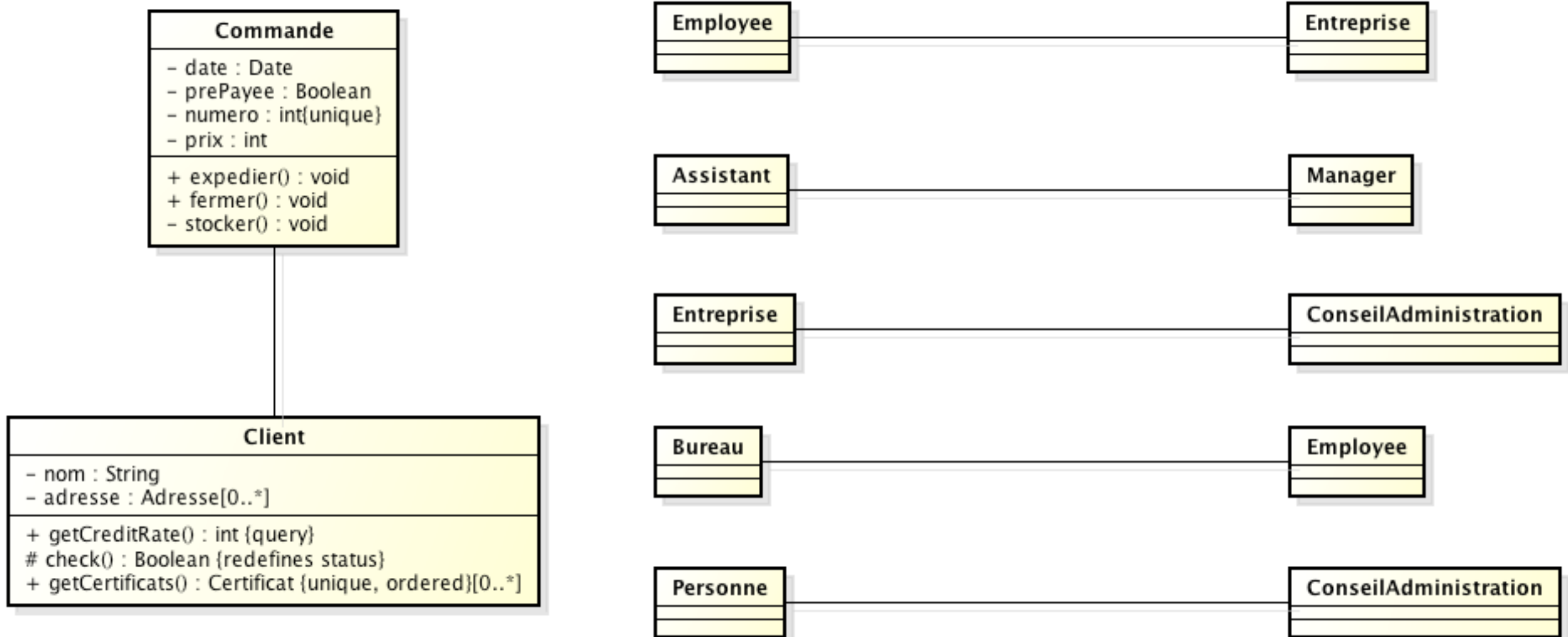
Rectangle
- height : int - width : int - / perimetre : int - <u>nbRectangles</u> : int
+ getArea() : int + resize() : void

Client
- nom : String - adresse : Adresse[0..*]
+ getCreditRate() : int {query} # check() : Boolean {redefines status} + getCertificats() : Certificat {unique, ordered}[0..*]

Commande
- date : Date - prePayee : Boolean - numero : int{unique} - prix : int
+ expedier() : void + fermer() : void - stocker() : void

Associations

Une association est utilisée afin de montrer comment deux classes sont liées entre elles



Associations

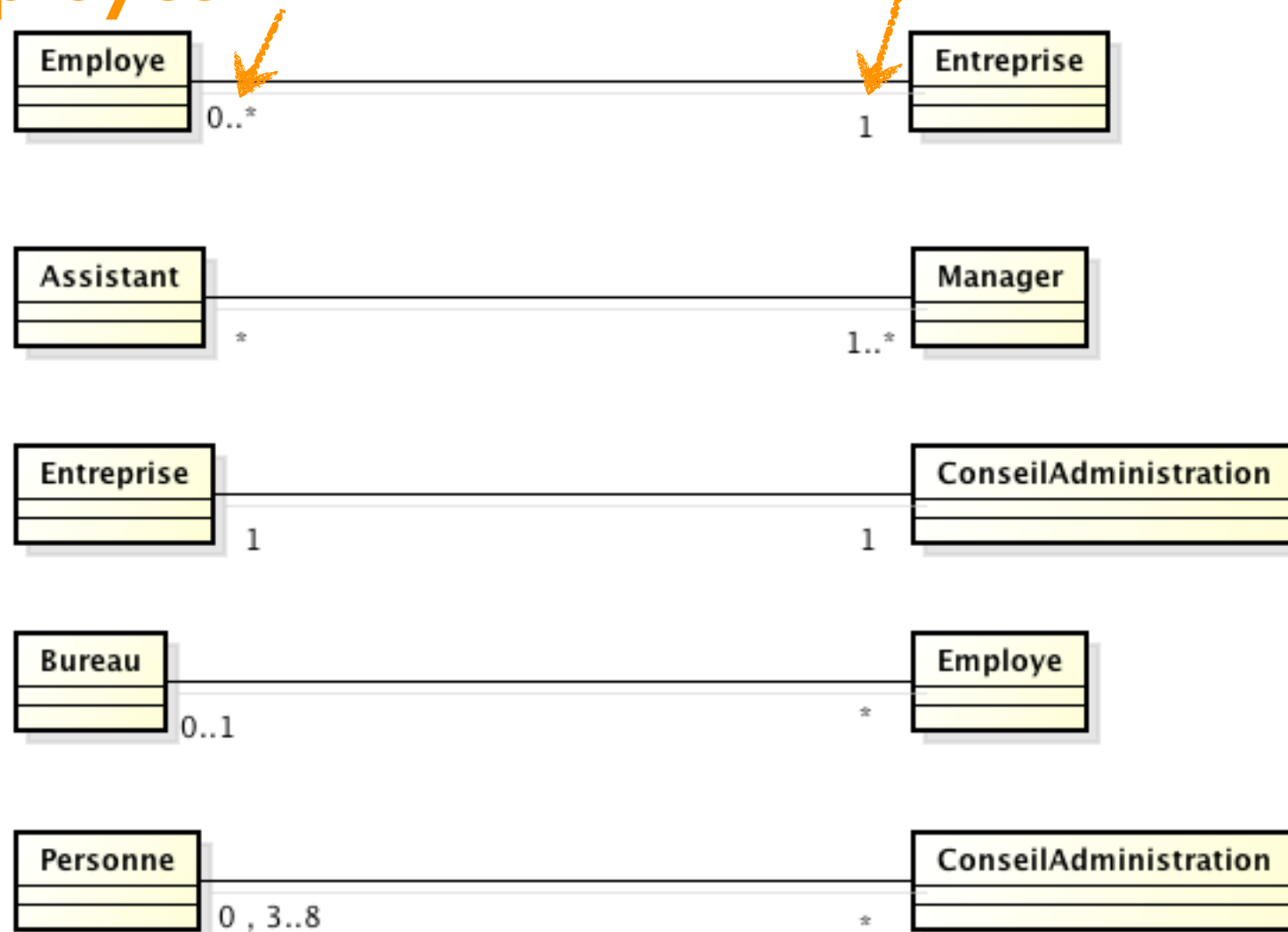
- Concepts
 - ▶ multiplicité
 - ▶ terminaison
 - ▶ navigabilité
 - ▶ arité

Associations

- la *multiplicité* est indiquée à chaque extrémité d'une association

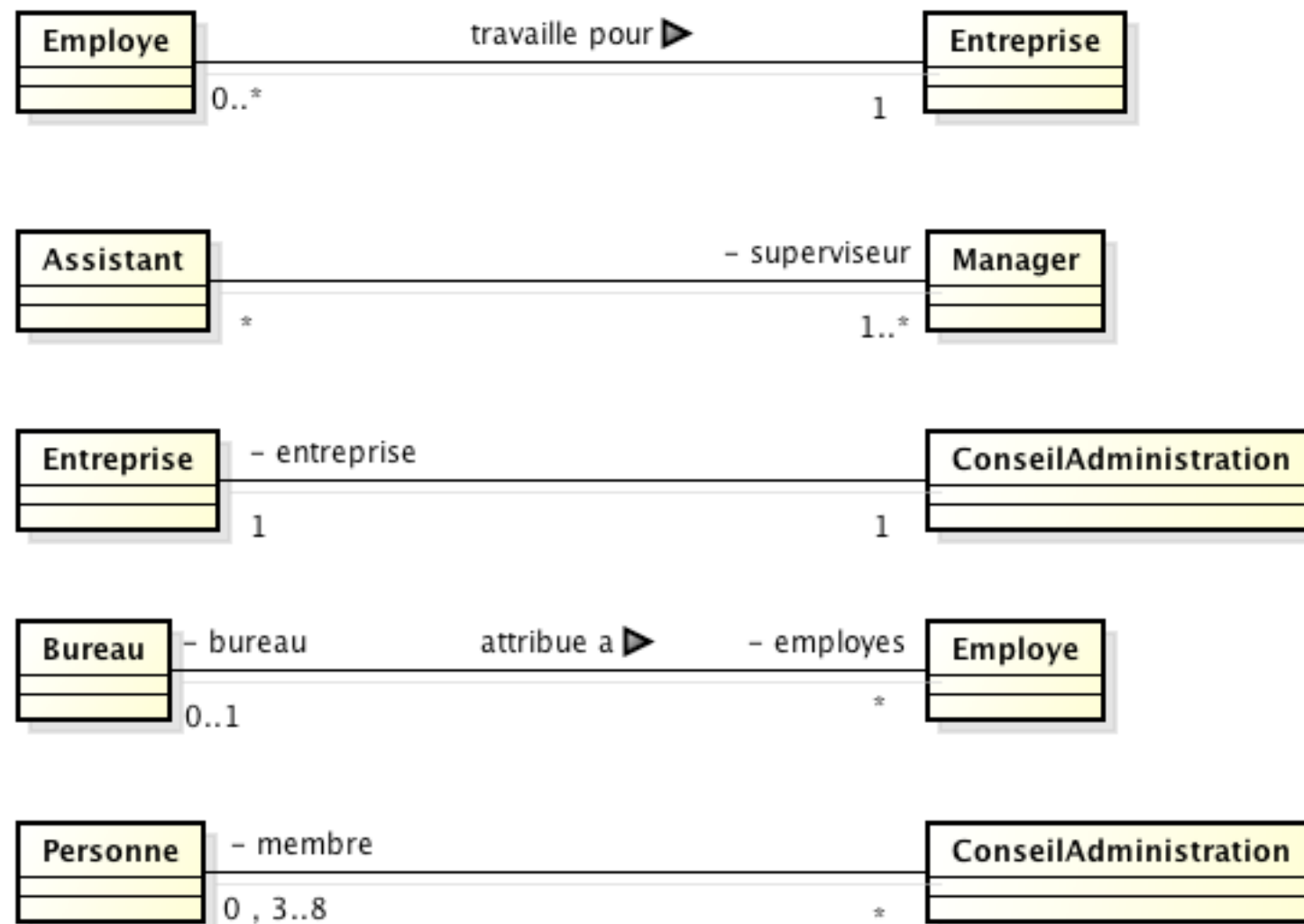
une **Entreprise** emploie 0 ou plusieurs **Employés**

un **Employé** travaille dans 1 entreprise



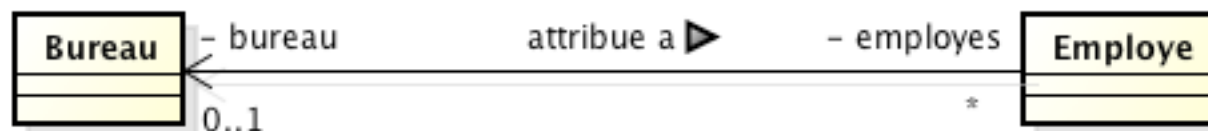
Associations

- Une association peut être étiquetée
 - ▶ nom association
 - ▶ terminaison d'association



Navigabilité

- possibilité de traverser une association
 - navigable, non-navigable, non-spécifié



Analyser et valider les associations

- Une compagnie a plusieurs employés
- Un employé ne peut travailler que pour une seule compagnie
 - ▶ et les employés occupant un double emploi?!
- Une compagnie peut n'avoir aucun employé
- On peut être employé seulement si on travaille pour une compagnie



Analyser et valider les associations

- Un(e) assistant(e) peut travailler pour plusieurs superviseurs
- Un superviseur peut avoir plusieurs assistants
- Certains superviseurs peuvent n'avoir aucun assistant
- Est-il possible qu'un(e) assistant(e) puisse se retrouver sans superviseur?



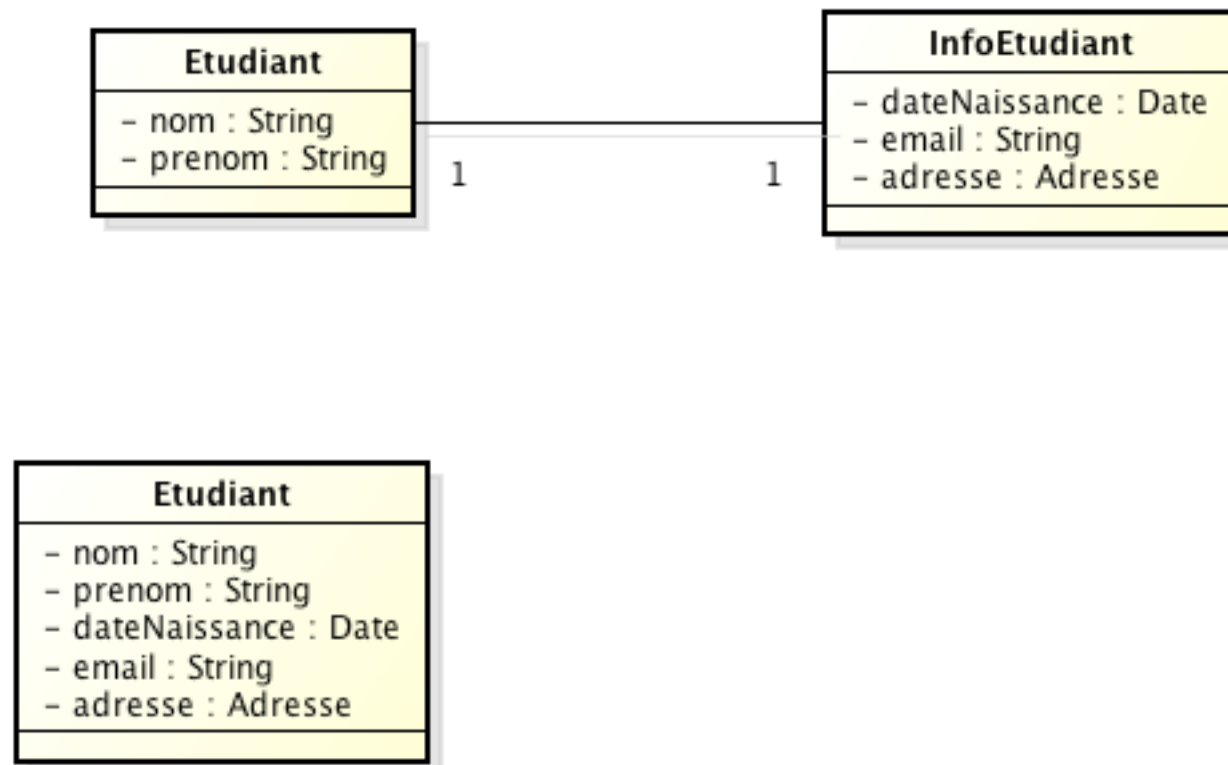
Analyser et valider les associations

- A chaque entreprise est associé un conseil d'administration
- Un conseil d'administration gère une seule entreprise
- Une entreprise doit avoir un conseil d'administration
- Un conseil d'administration est toujours attaché à une et une seule entreprise



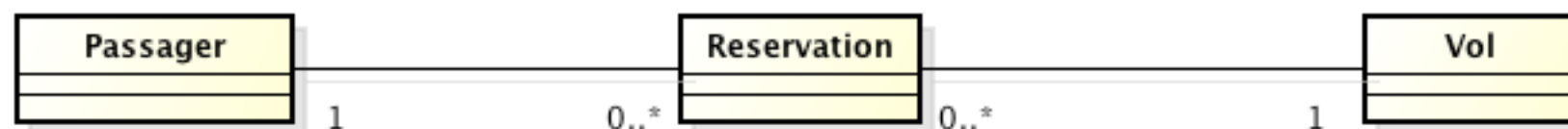
Analyser et valider les associations

- Attention aux associations une-à-une injustifiées



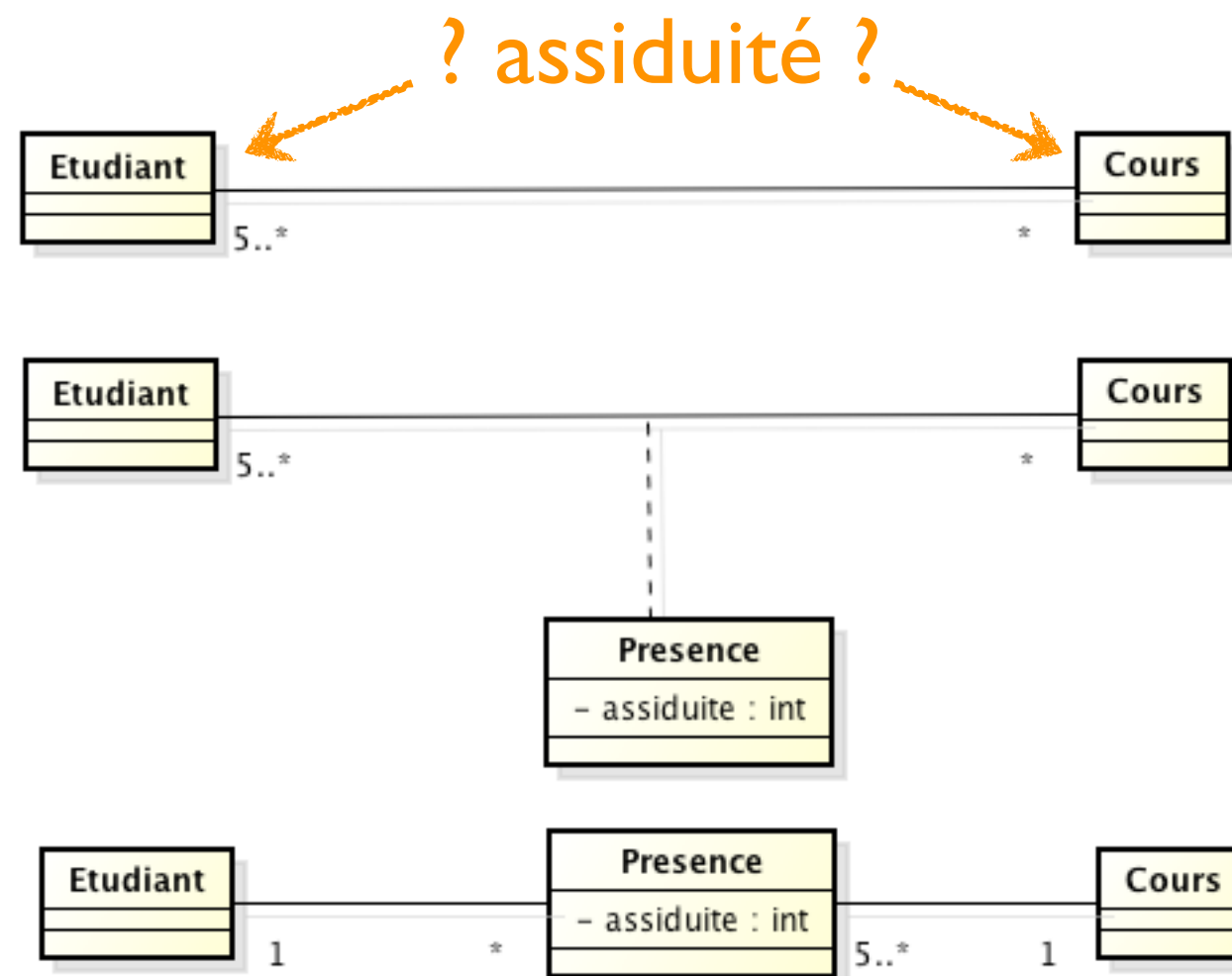
Analyser et valider les associations

- Une réservation est pour un passager
 - ▶ pas de réservations sans passager
 - ▶ une réservation ne devrait jamais compter plus d'un passager
- Un passager peut effectuer plusieurs réservations
 - ▶ un passager peut aussi n'avoir fait aucune réservation



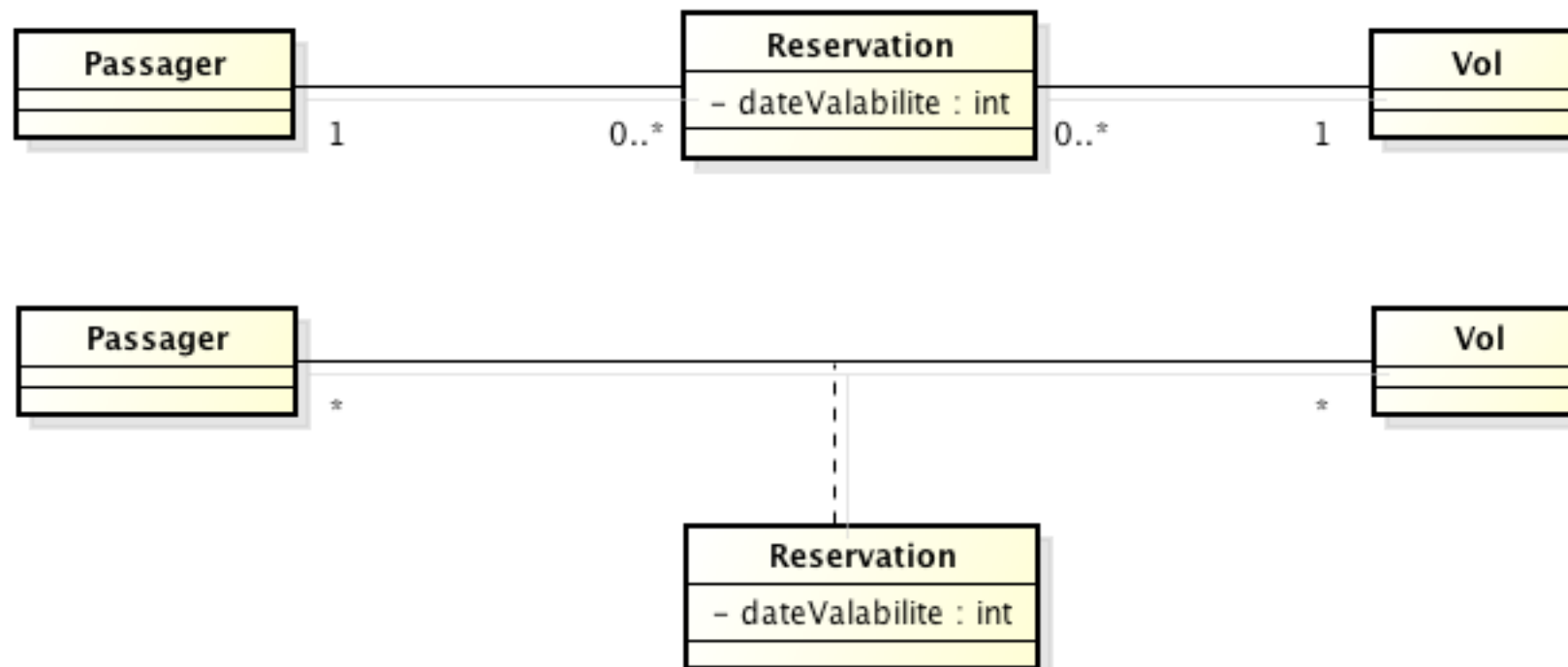
Classes d'association

- Il arrive qu'un attribut ne puisse être attaché à aucune des deux classes d'une association
➔ il peut être attaché à l'association



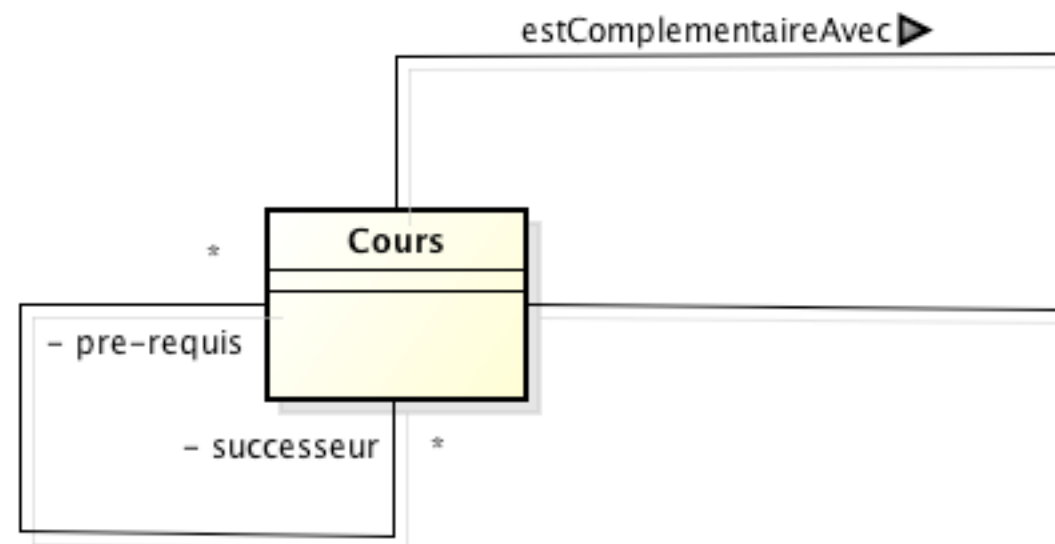
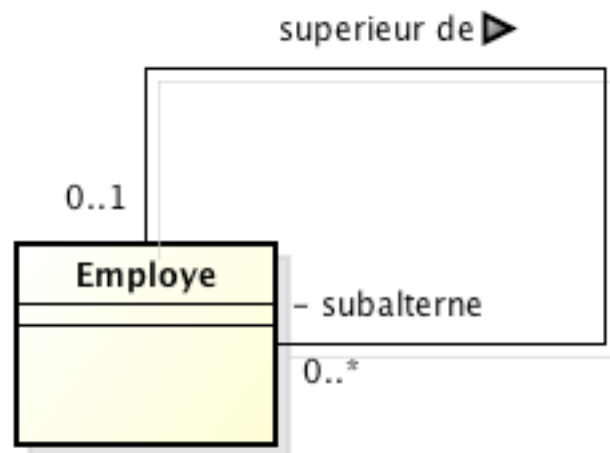
Classes d'association

- Il arrive qu'un attribut ne puisse être attaché à aucune des deux classes d'une association
➡ il peut être attaché à l'association



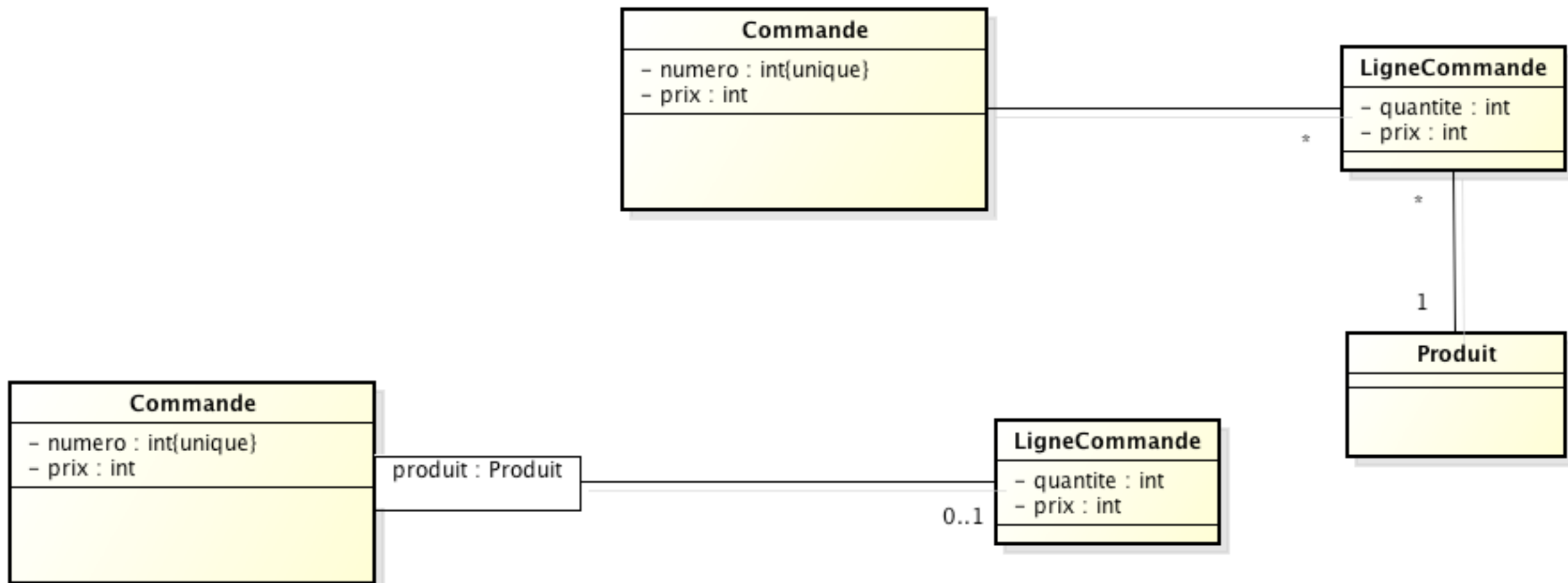
Association réflexive

- Une classe peut être associée à elle-même



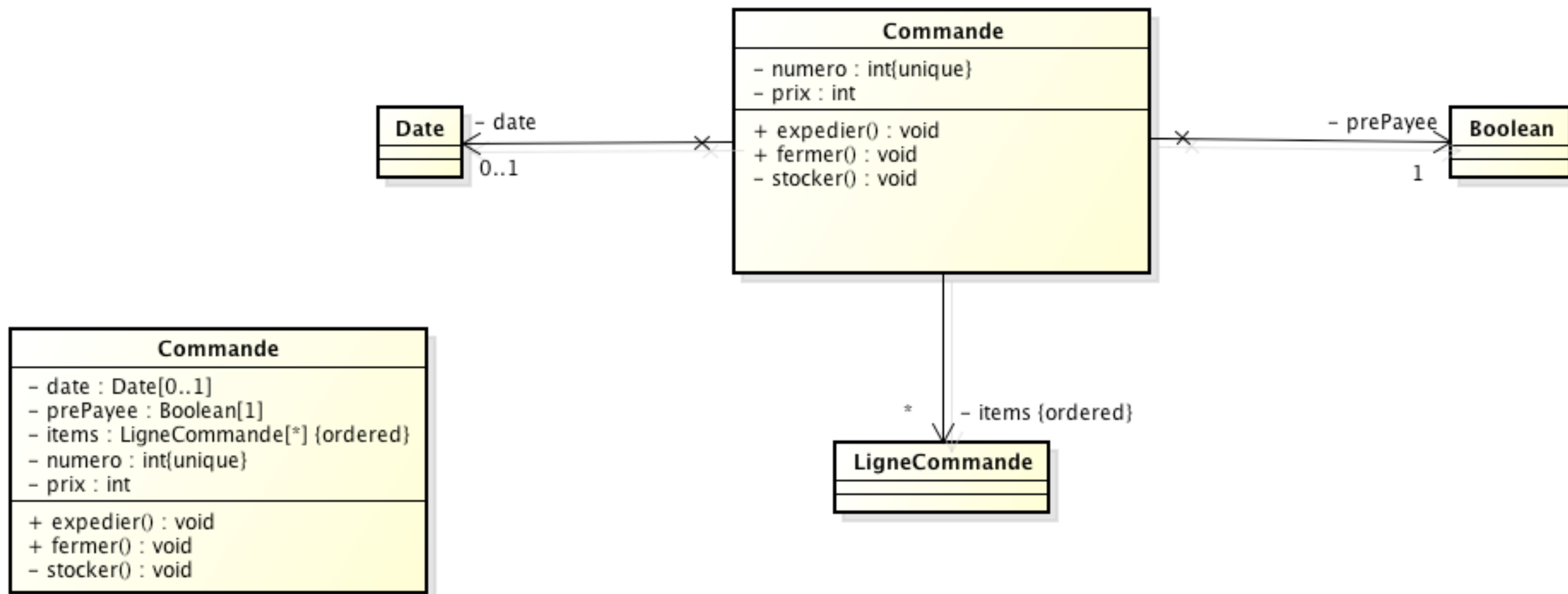
Association qualifiée

- le « **qualifier** » est utilisé pour identifier des objets reliés
- correspond à un tableau associatif/dictionnaire



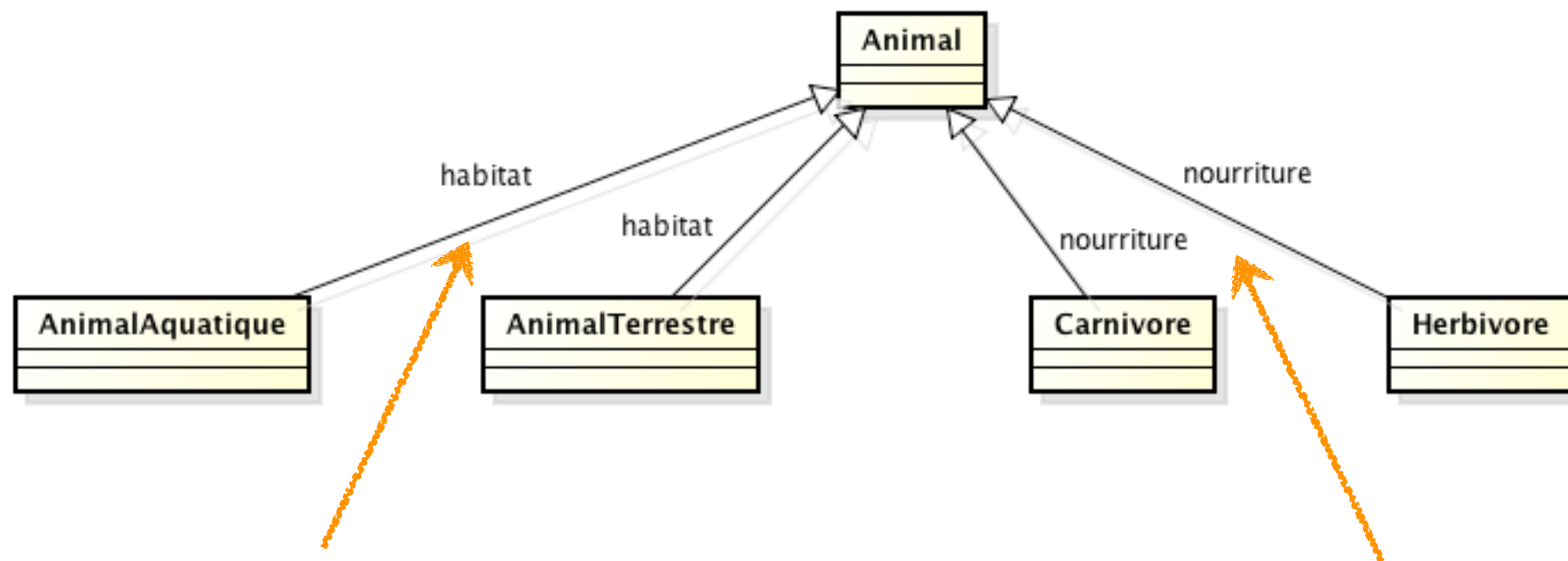
Propriétés

- attributs et associations
 - comment choisir ?



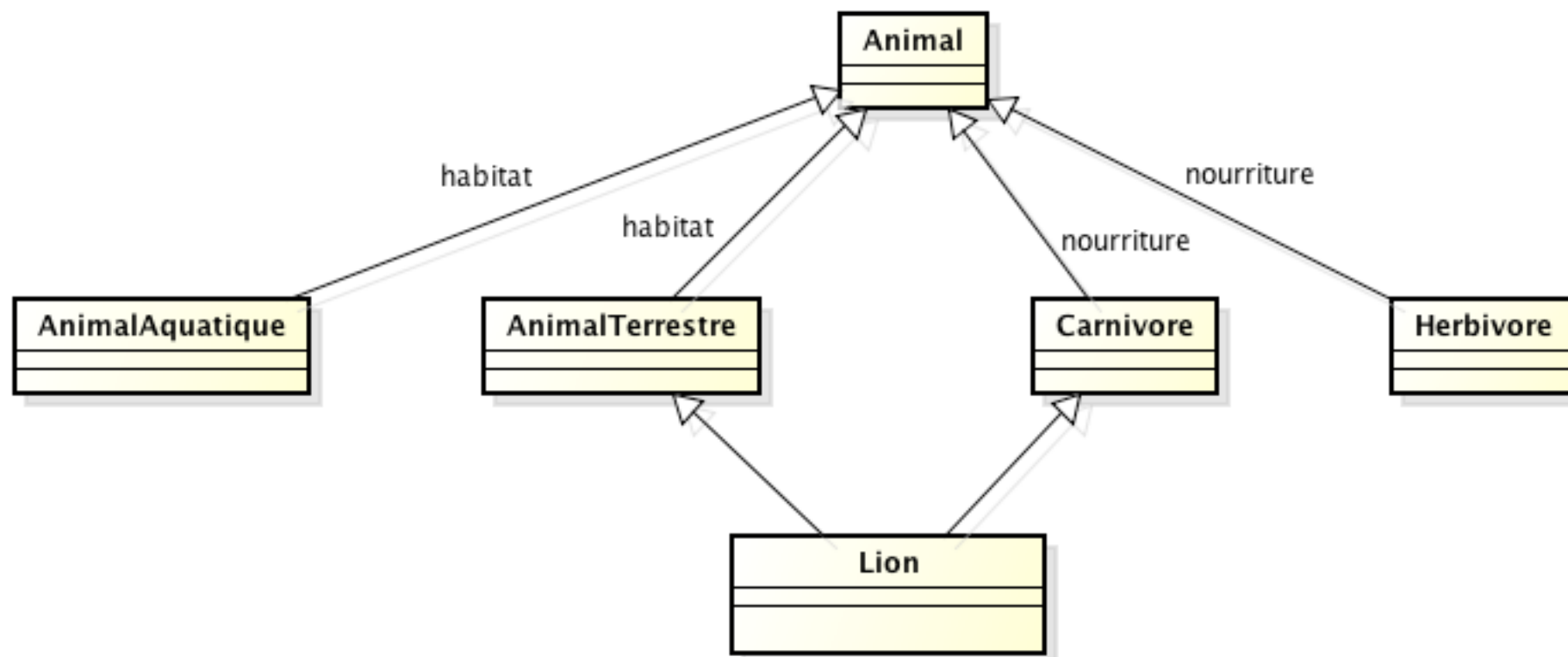
Généralisation

- Une super-classe se spécialise en sous-classes



le **discriminant** (**discriminator/generalization set name**)
est une étiquette décrivant le critère suivant lequel se
base la spécialisation

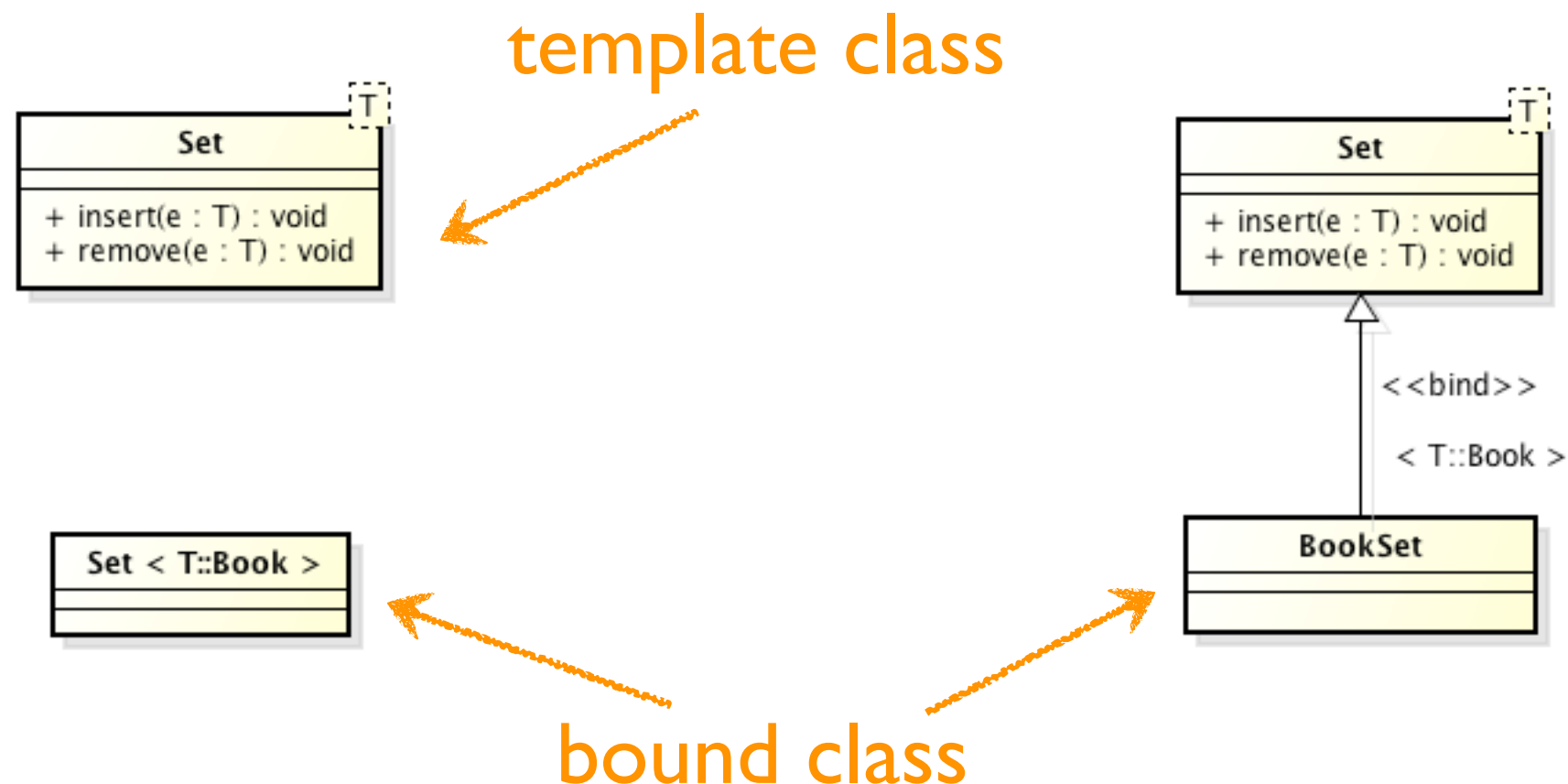
Héritage multiple



- accepter implicitement mais pas défini explicitement

Classes paramétrisées

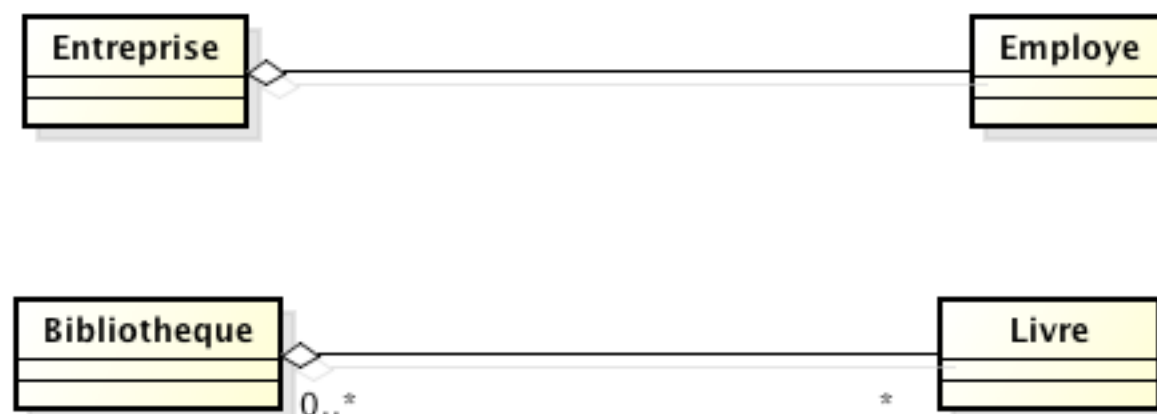
- définition d'une classes en fonction d'un paramètre de type



Agrégation

Une agrégation est une forme spéciale représentant une relation '*partie-tout*'.

- le '*tout*' est souvent appelé l'**ensemble** ou l'**agrégat**
- abréviation d'une association « **est une partie de** »



A quel moment faut-il utiliser l'agrégation?

En général, une *association* peut être représentée comme une *agrégation* si:

- il y a une relation de type *est-une-partie-de*
- il y a une relation de type *est-composé-de*

Lorsque quelque chose contrôle l'agrégat, il contrôle aussi ses parties.

Composition

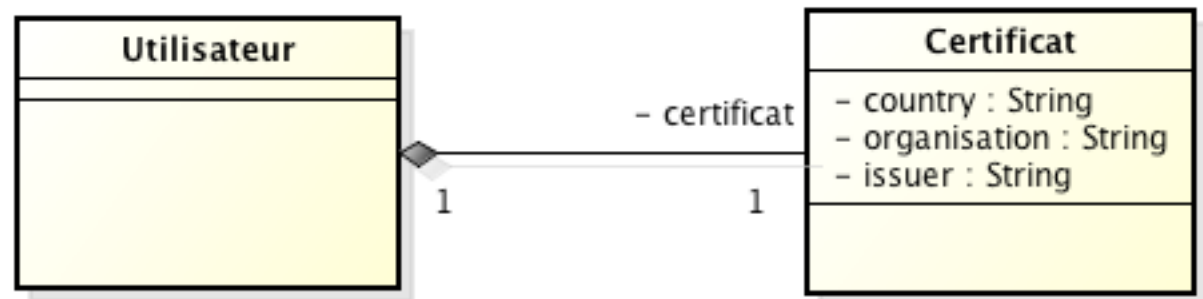
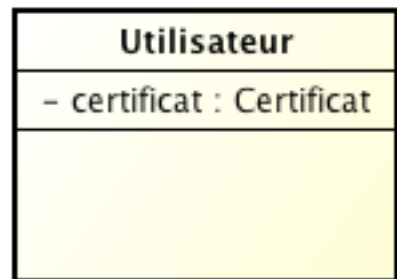
Une composition est une forme forte d'agrégation

- si l'agrégat est détruit, alors ses parties le sont aussi

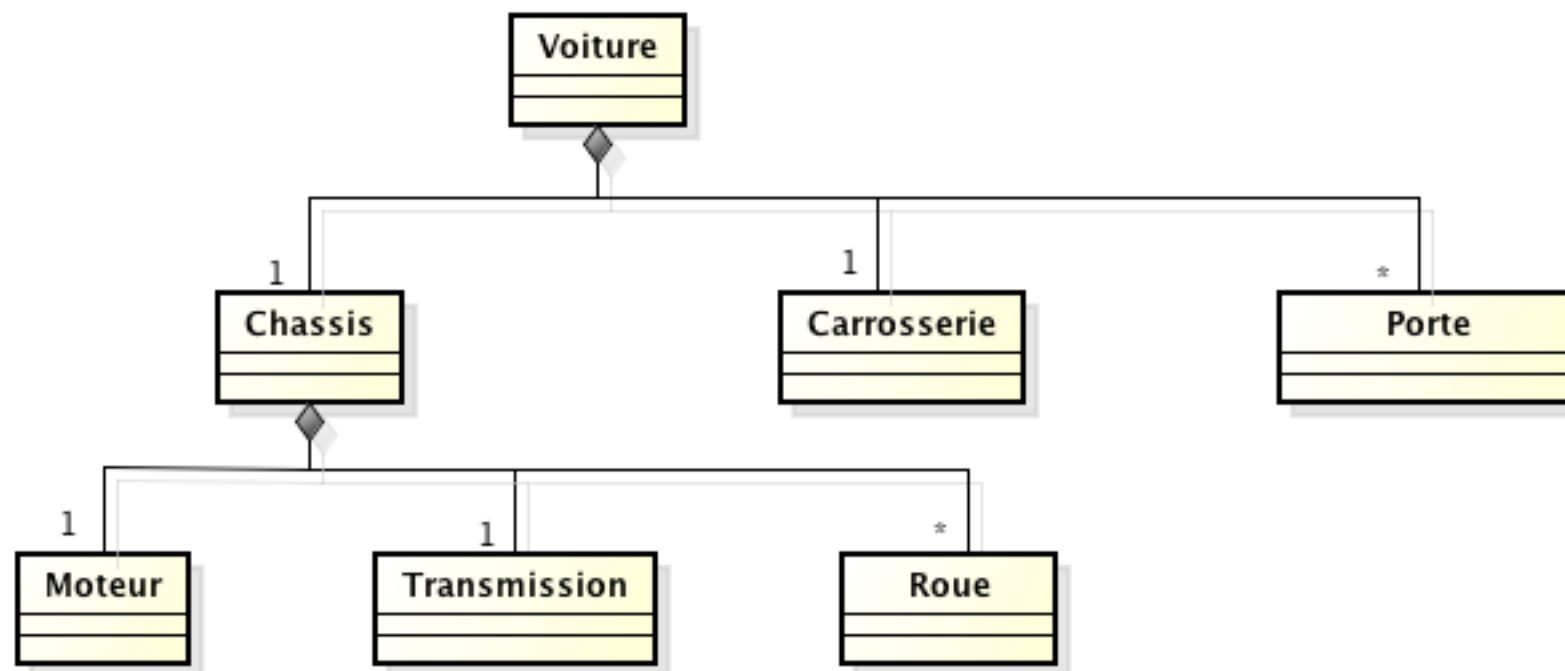


Composition

Modélisations alternatives:



Hiérarchie d'agrégations



Propagation

- La *propagation* est un mécanisme statuant que lorsqu'une opération est effectuée sur le tout elle doit aussi s'appliquer à ses parties
- La *propagation* permet aussi aux propriétés des parties de se propager vers le tout

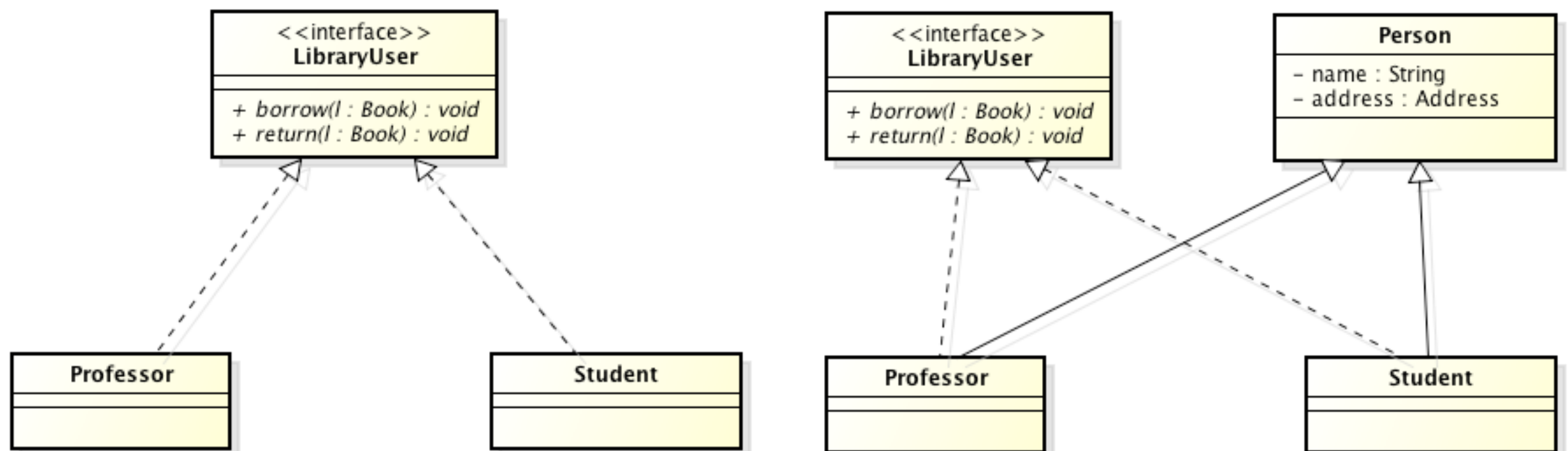
Propagation

- La **propagation** est à **l'agrégation** ce que **l'héritage** est à la **généralisation**.
- La différence majeure est que:
 - ➔ *L'héritage* est un mécanisme *implicite*
 - ➔ La *propagation* doit être *programmée*

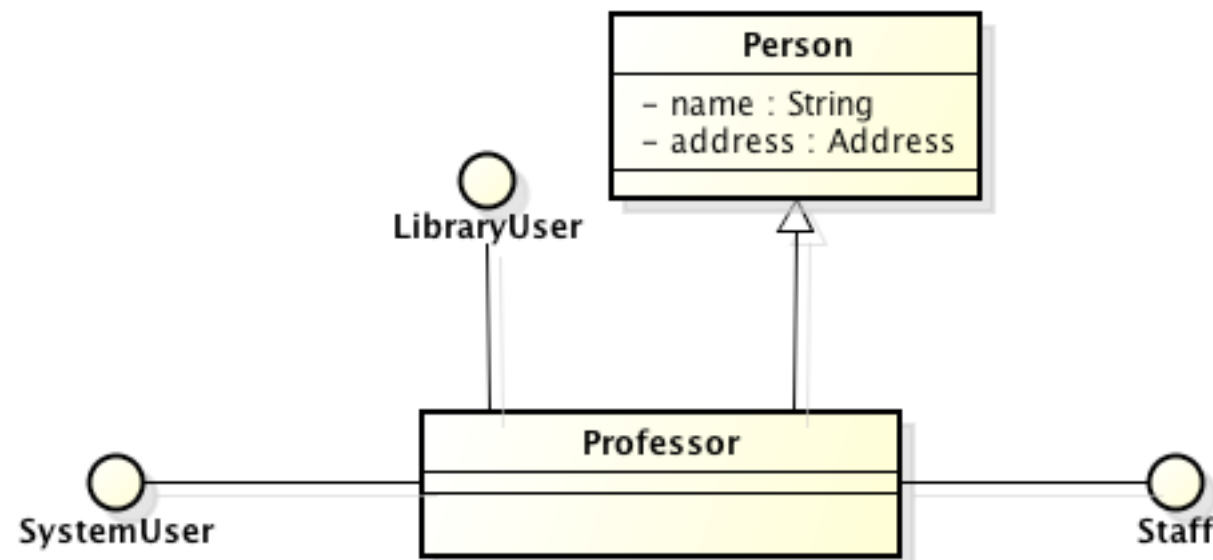
Les interfaces

Une *interface* décrit une *portion du comportement visible* d'un ensemble d'objets.

- similaire à une classe ne contenant que des *opérations abstraites*

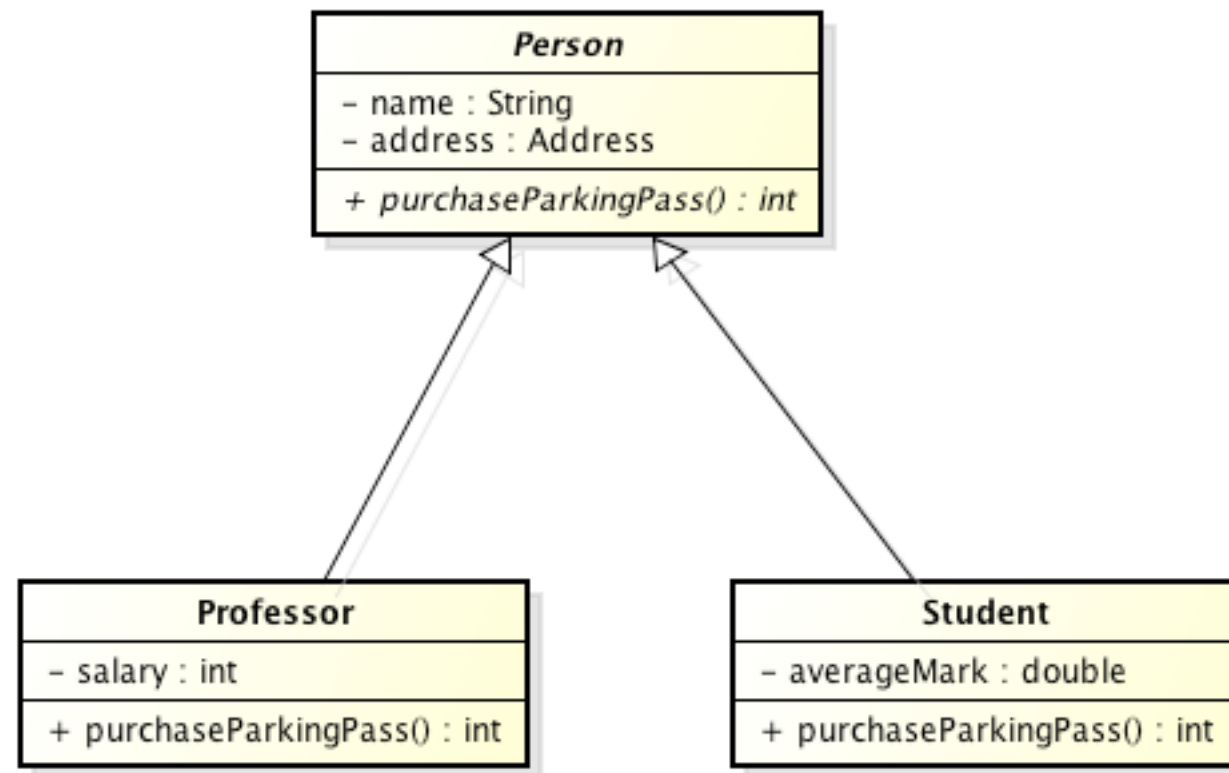


Socket notation

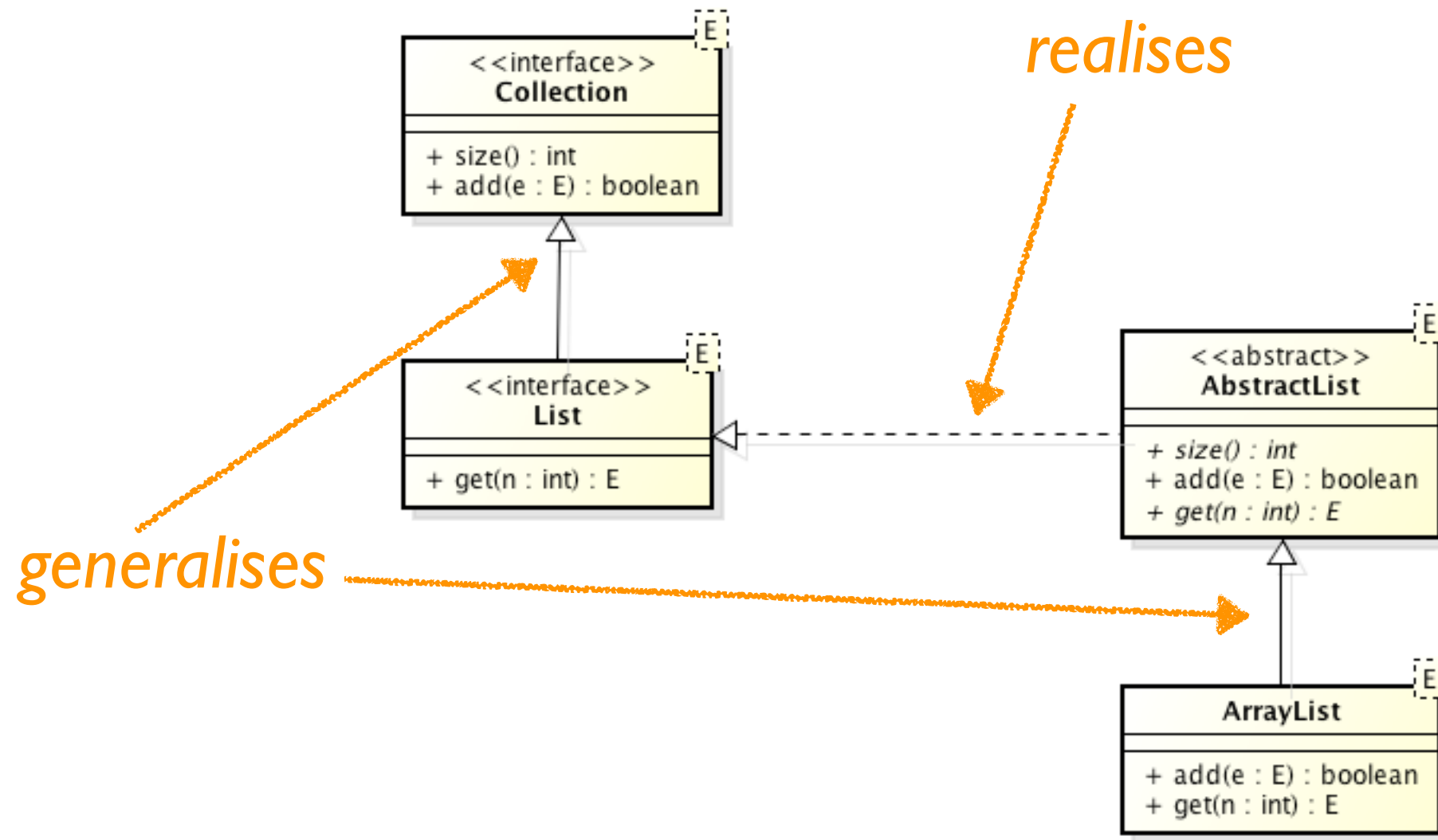


Classes abstraites

- classes avec des opérations abstraites (c.a.d. sans implantation)



Interfaces et classes abstraites



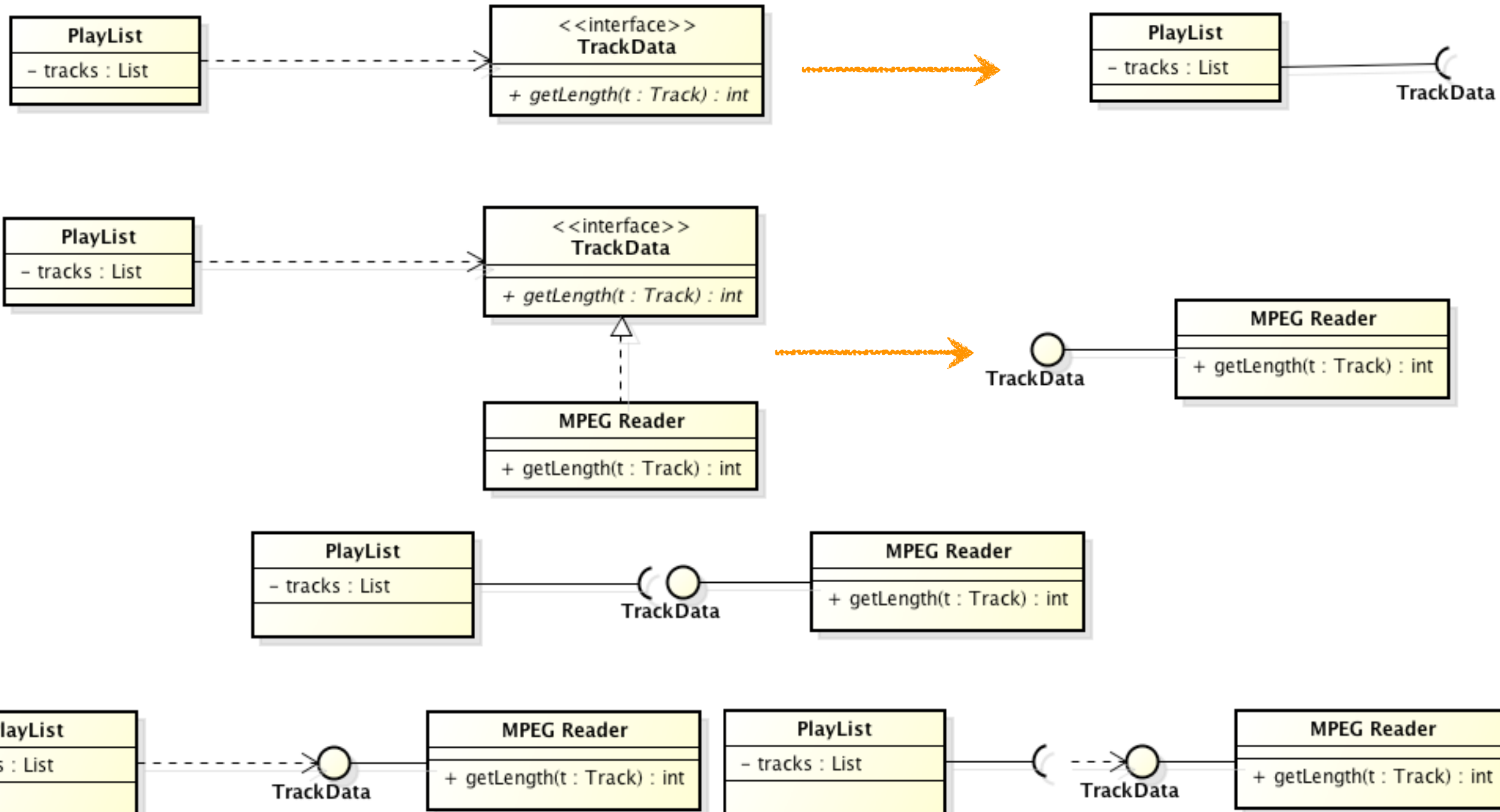
Dépendances

- relations (unidirectionnelles) exprimant une dépendance sémantique entre deux classes
 - ▶ la modification de la cible peut impliquer une modification de la source



utiliser une
interface

Ball and socket



Développement des diagrammes de classes

- *Modèle exploratoire du domaine:*
 - ▶ développé durant l'analyse de domaine dans le but d'en apprendre un peu plus sur le domaine
- *Modèle du domaine appartenant au système:*
 - ▶ modéliser aspects domaine faisant partie du système
- *Modèle du système:*
 - ▶ toutes les classes, y compris les classes d'architecture et des interfaces utilisateurs
 - ▶ développé durant la conception du système

Domaine-système vs système

Modèle du domaine-système

- souvent moins de la moitié des classes du système
- devrait être indépendant:
 - des interfaces utilisateur
 - des classes d'architecture

Modèle du système

- les classes des interfaces utilisateur
- les classes de l'architecture
- les autres classes utilitaires

Séquence d'activités

- Identifier un premier ensemble de *classes candidates*
- Ajouter les *associations* et *attributs*
- Trouver les *généralisations*
- Lister les *responsabilités* principales des classes
- Décider quelles seront les *opérations*

Séquence d'activités

- Effectuer quelques itérations jusqu'à satisfaction:
 - ▶ Ajouter/retirer des classes, associations, attributs, généralisations, opérations, responsabilités...
 - ▶ Identifier les interfaces
 - ▶ Appliquer les design patterns appropriés

Identifier les classes

- Lors du développement du modèle du domaine, les classes sont découvertes
- Lorsque le reste du système est élaboré, de nouvelles classes sont identifiées
 - ▶ classes requises pour obtenir un système fonctionnel
- La réutilisation doit rester une priorité
 - ▶ frameworks
 - ▶ systèmes similaires

Découvrir les classes

- Examiner la description des exigences
- *Extraire* les **noms** et en faire des classes candidates
- *Éliminer* les **noms** qui:
 - ▶ sont redondants
 - ▶ représentent des instances
 - ▶ sont vagues ou trop généraux
 - ▶ sont inutiles dans le contexte de l'application

Identifier attributs et associations

- Commencer avec les classes considérées *centrales* dans le système
- Déterminer les *informations* que chacune de ces classes devrait contenir et les *relations* avec les autres classes
- Ajouter graduellement les autres classes
- Éviter d'ajouter trop d'attributs ou d'associations
 - ▶ *Un système manipulant peu d'information est toujours plus facile à maîtriser*

Identifier des associations

- Une association devrait exister si une classe
 - ▶ possède
 - ▶ contrôle
 - ▶ est connecté à
 - ▶ est relié à
 - ▶ est une partie de
 - ▶ est constituée de parties de
 - ▶ est membre de
 - ▶ a comme membres

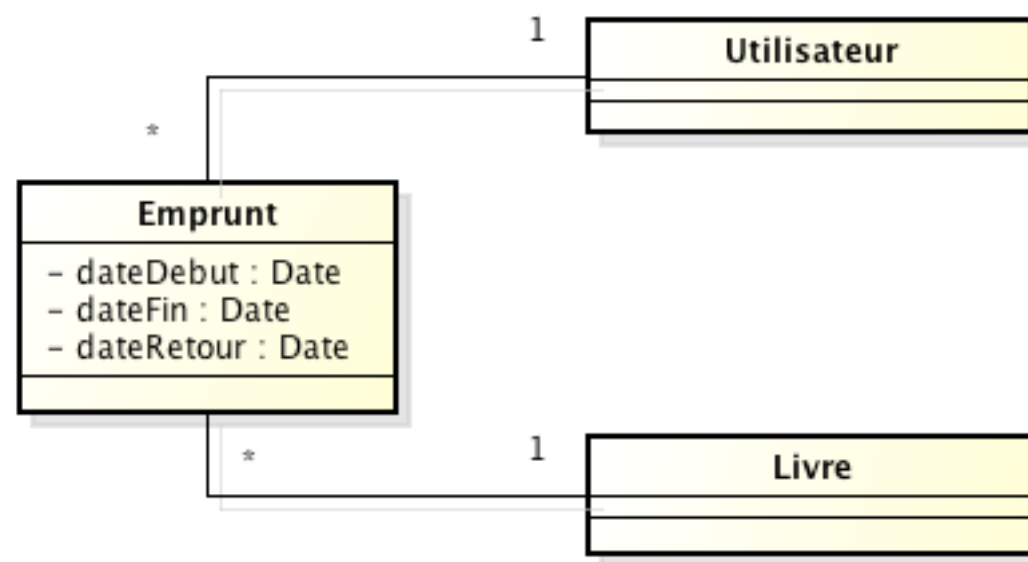
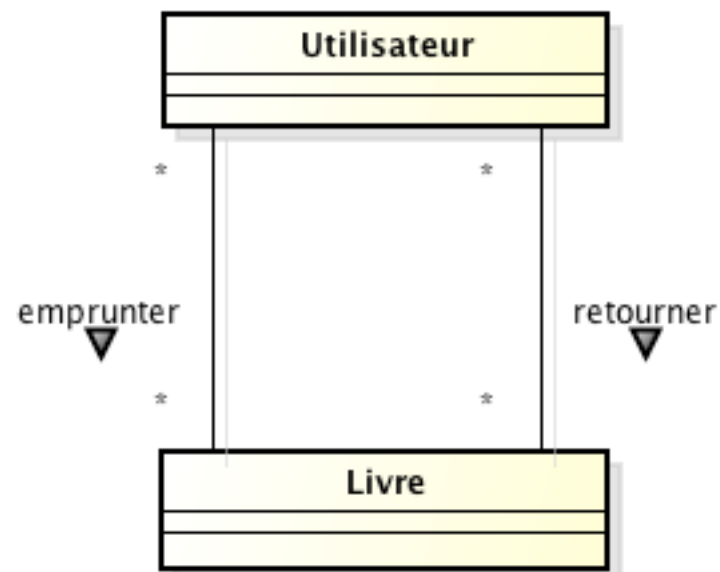
une autre classe dans le modèle

Identifier des associations

- Spécifier ensuite la multiplicité à chaque extrémité de l'association
- Étiqueter clairement cette association

Actions versus associations

- **erreur fréquente** → représenter une action comme si elle était une association

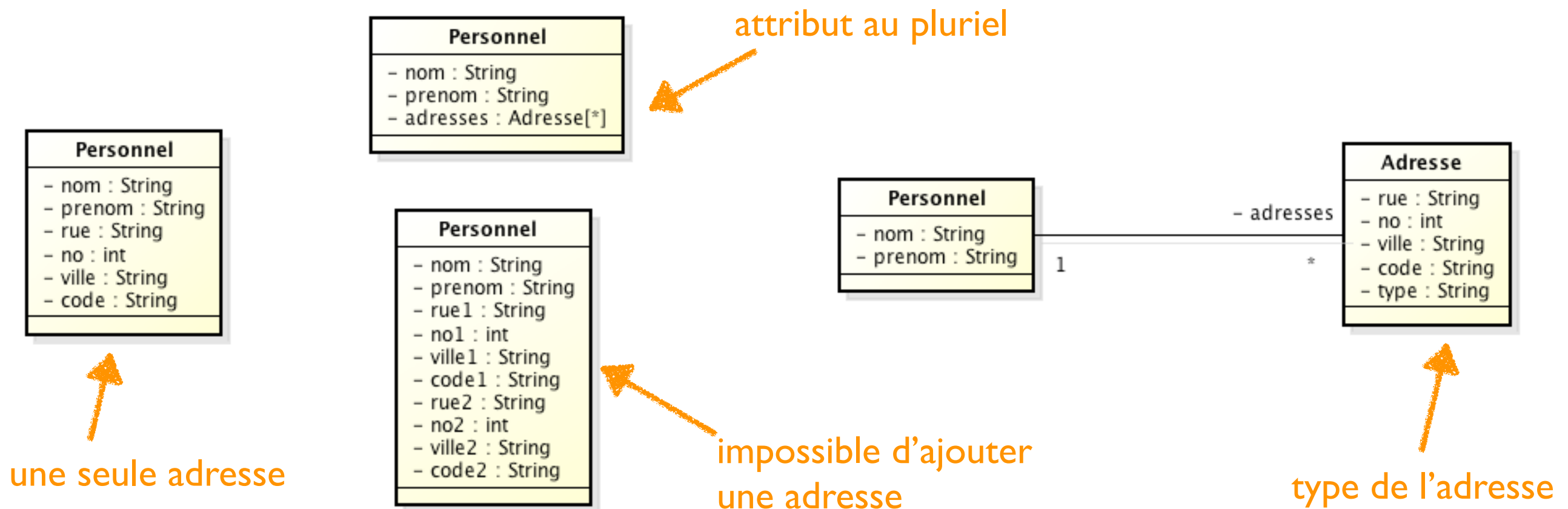


Identifier les attributs

- Déterminer l'information qui doit être maintenue dans chacune des classes
- Plusieurs noms ayant été rejetés comme classes deviennent alors des attributs
- Un attribut contient en général une valeur simple
 - ▶ ex: chaîne de caractères, nombre

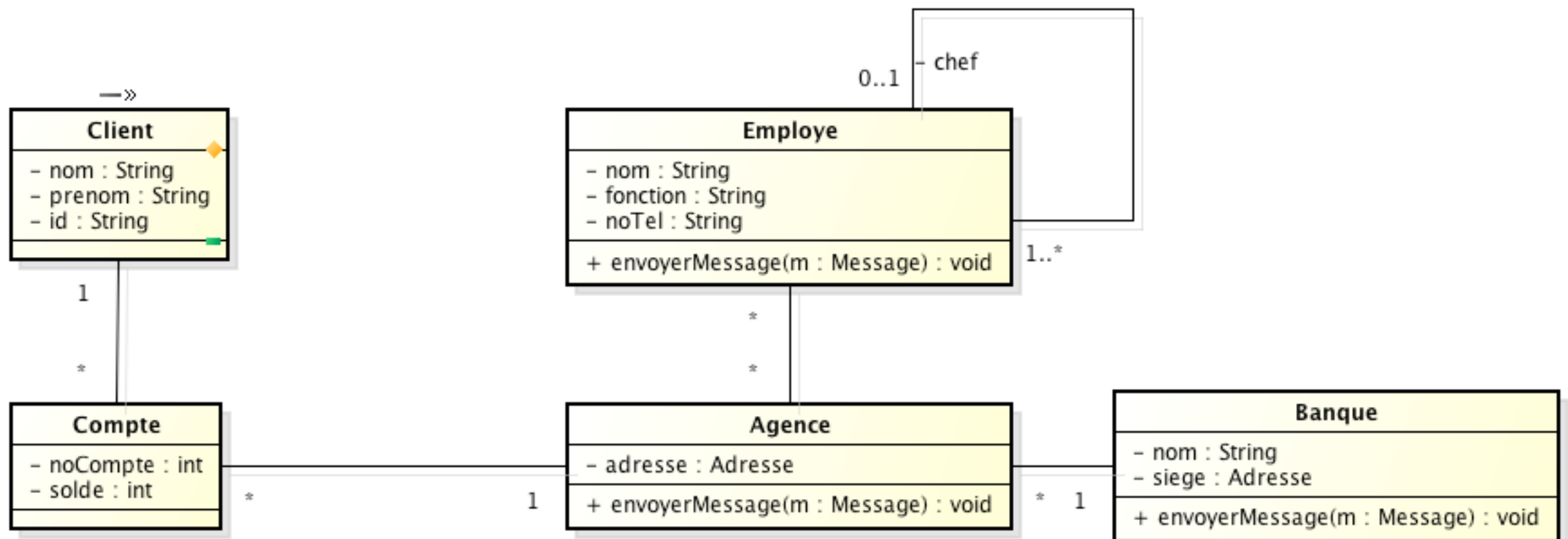
Identifier les attributs

- Pas de duplication des attributs
- Si un sous-ensemble des attributs forme un groupe cohérent, alors une nouvelle classe devrait peut être introduite



Exemple

« les **clients** avec des **comptes** ouverts dans les **agences** d'une **banque** doivent pouvoir contacter un **employé** de l'agence (ou son **chef**) par **courrier** ou **téléphone** » »



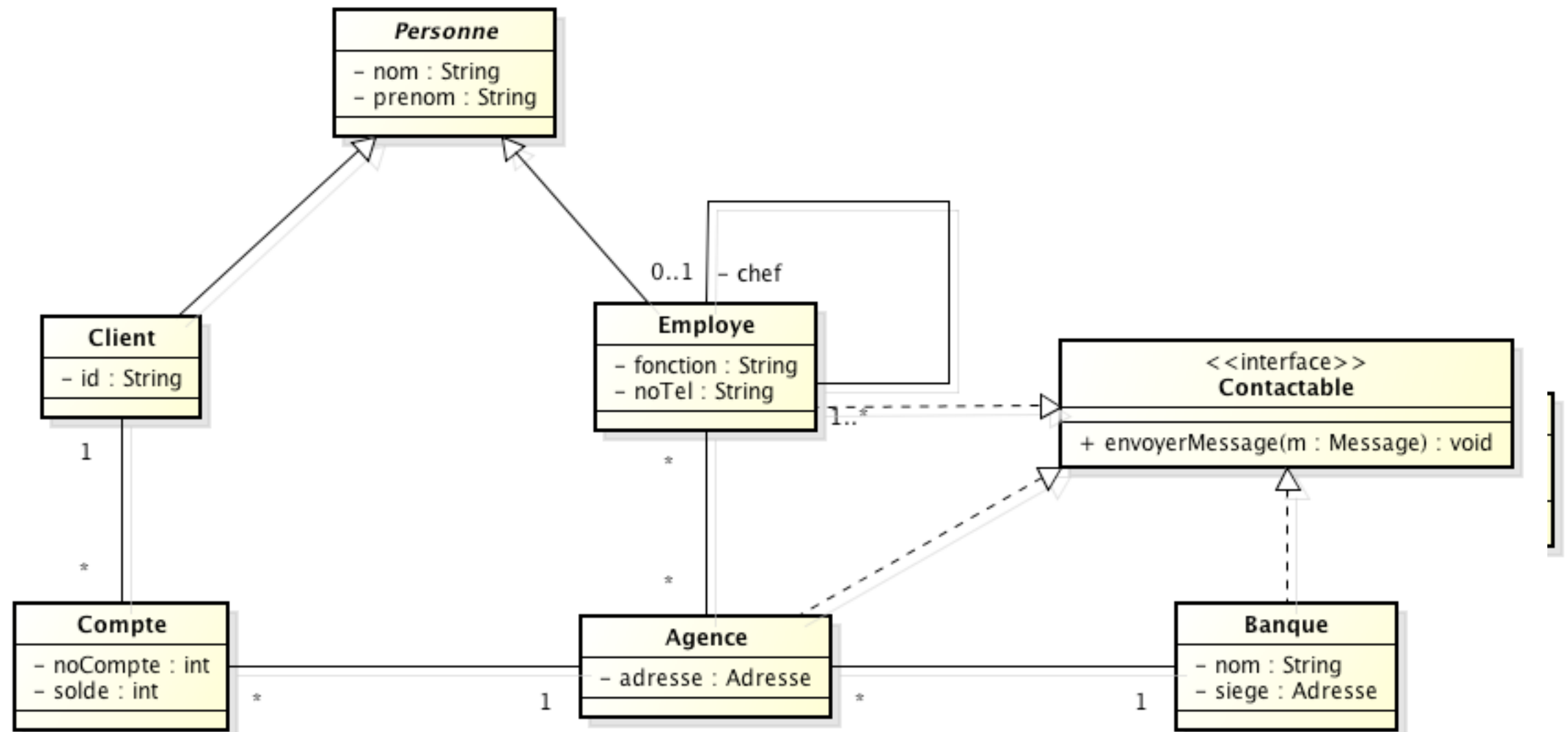
Identifier les interfaces et les généralisations

- De bas en haut
 - ▶ grouper ensemble les classes similaires
- De haut en bas
 - ▶ spécialiser les classes plus générales

Identifier les interfaces et les généralisations

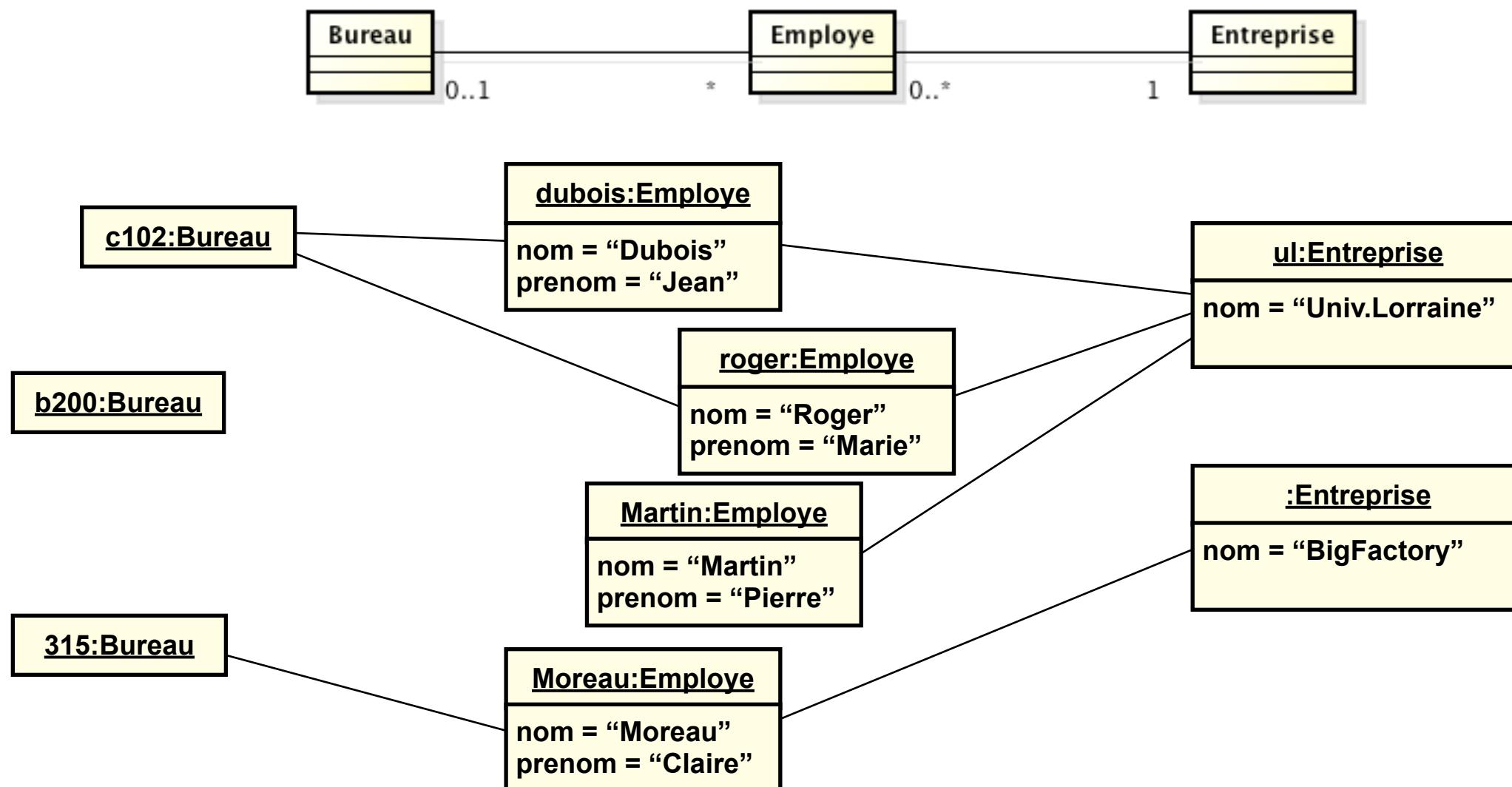
- Créer une interface, plutôt qu'une super-classe si
 - ▶ deux classes partageant certaines opérations sont considérées similaires
 - ▶ une classe à généraliser possède déjà sa propre super-classe
 - ▶ différentes implémentations de la même classe pourraient être disponibles

Exemple



Diagrammes d'instance

- Un lien est l'instance d'une association
 - ➡ comme un objet est l'instance d'une classe



Associations versus généralisations

Une *associations* décrit la relation qui existera entre deux instances en cours d'exécution.

- le diagramme d'instances montre les deux instances et le lien qui les unit

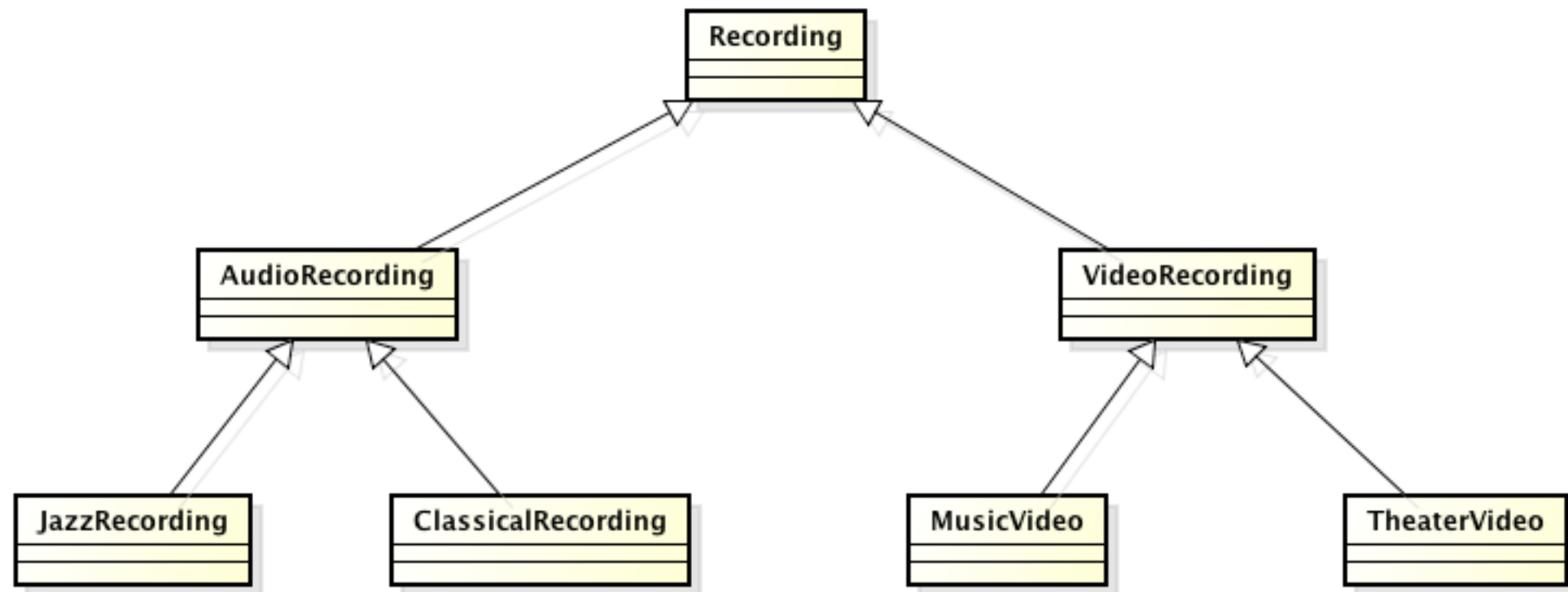
Une *généralisation* décrit une relation entre classes dans un diagramme de classes

- les super-classes n'apparaissent pas dans un diagramme d'instances.
- une instance d'une classe est aussi une instance de sa super-classe

Exemple

- Modéliser une application permettant la gestion d'enregistrements
 - ▶ les enregistrements peuvent être audio ou video
 - ▶ enregistrements audio : jazz, classic, pop, ...
 - ▶ enregistrements video : musique, théâtre, ...

Example

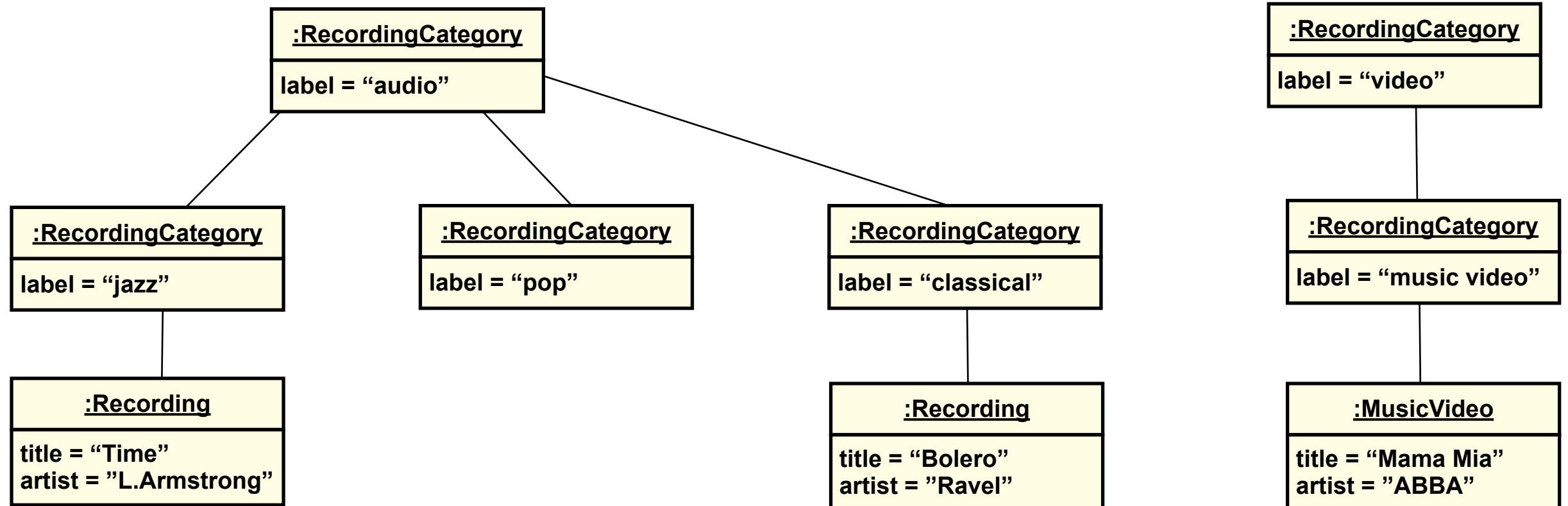
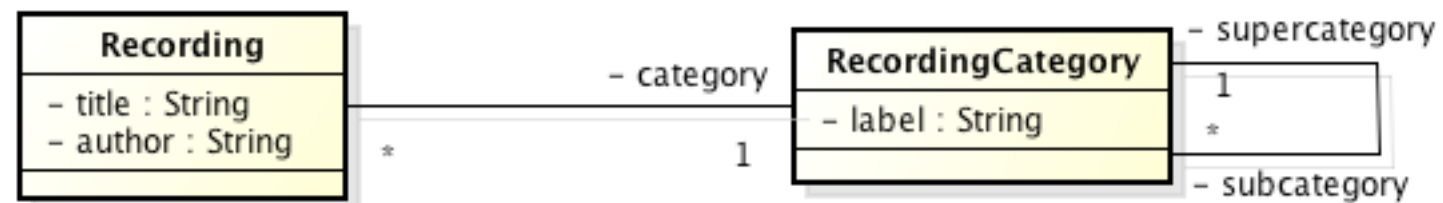


<u>:JazzRecording</u>
title = "Time" artist = "L.Armstrong"

<u>:ClassicalRecording</u>
title = "Bolero" artist = "Ravel"

<u>:MusicVideo</u>
title = "Mama Mia" artist = "ABBA"

Éviter les généralisations inutiles



Allouer des responsabilités

- ▶ Toutes les responsabilités d'une classe devrait être *reliées* entre elles
- ▶ Si une classe a trop de responsabilités, elle devrait être *scindée* en plusieurs classes
- ▶ Si une classe n'a aucune responsabilité, alors celle-ci est probablement *inutile*
- ▶ Lorsqu'une responsabilité ne peut être attribuées à aucune classe, une *nouvelle* classe devrait être introduite

Allouer des responsabilités

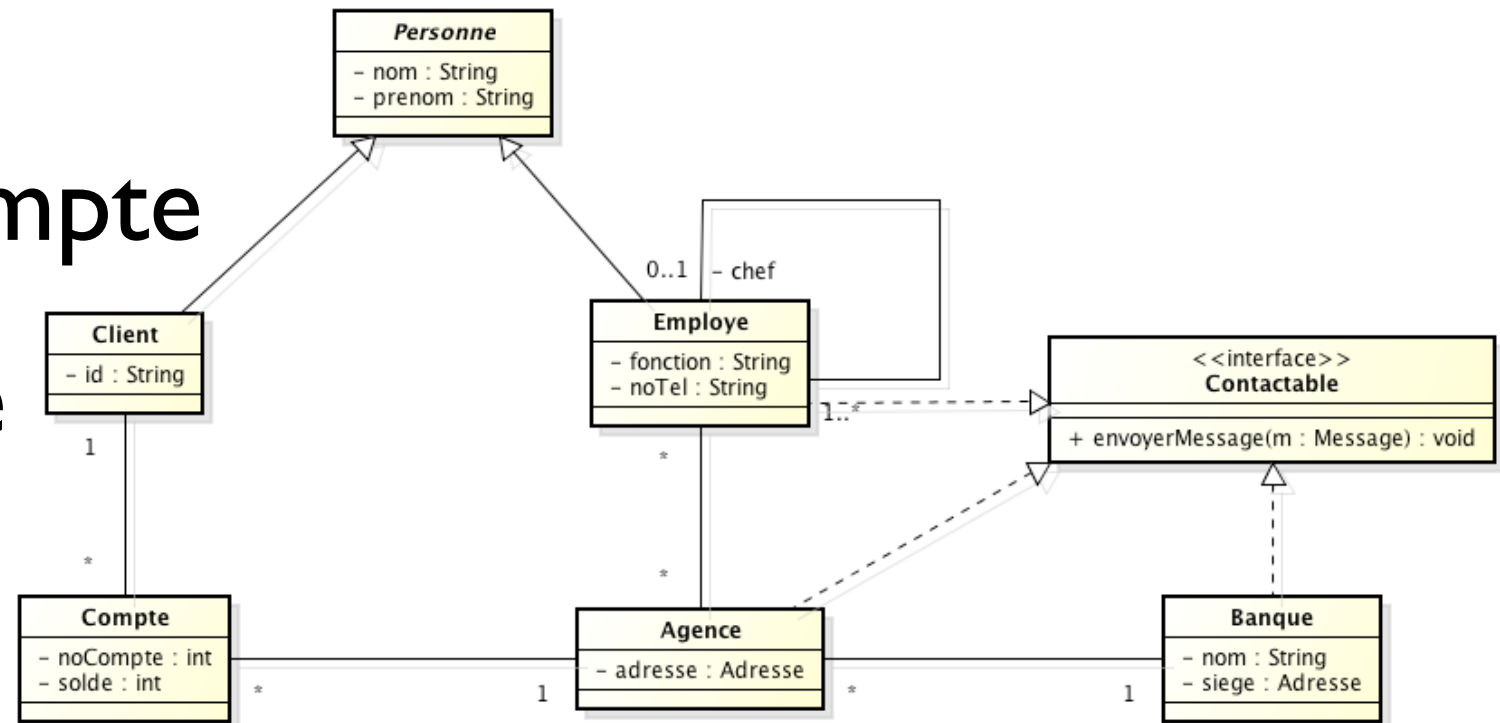
- Afin de déterminer les responsabilités
 - ▶ Effectuer une analyse de cas
 - ▶ Recherche des verbes et des noms décrivant des actions

Catégories de responsabilités

- Fixer et obtenir la valeur d'un attribut
- Créer et initialiser de nouvelles instances
- Sauvegarder et récupérer de l'information persistante
- Détruire des instances
- Ajouter et détruire des liens
- Copier, convertir, transformer, transmettre, afficher
- Calculer des résultats numériques
- Naviguer et rechercher
- Tout autre tâche...

Exemple

- Créer un nouveau compte
- Rechercher un agence
- Créer une agence
- Faire opposition sur un compte
- Envoyer un message à un employé



Identifier les opérations

Les *opérations* servent à réaliser les *responsabilités*

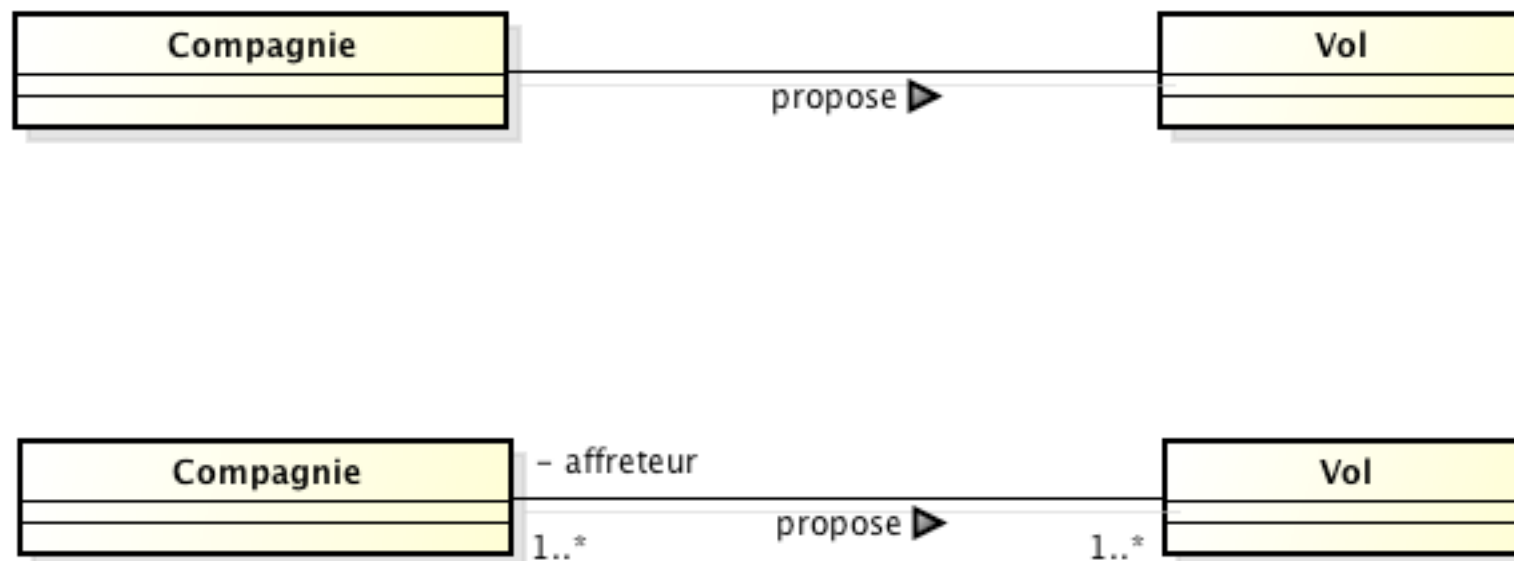
- Il peut y avoir plusieurs opérations pour une responsabilité
- Les opérations de base réalisant ces responsabilités seront déclarées *publiques*
- Une méthode qui collabore à la réalisation d'une responsabilité sera, dans la mesure du possible, déclarée *privées*

Exemple: Vols

1. les vols sont proposés par différentes compagnies
2. la compagnie décide de l'ouverture et la fermeture à la réservation
3. un client peut réserver des vols pour plusieurs passagers
4. une réservation concerne un seul vol et un seul passager
5. une réservation peut être confirmée ou annulée
6. un vol a un aéroport de départ et un d'arrivée
7. un vol a une date/horaire de départ et une autre d'arrivée
8. un vol peut comporter des escales
9. une escale a une heure de départ et une d'arrivée
10. un aéroport dessert une ou plusieurs villes

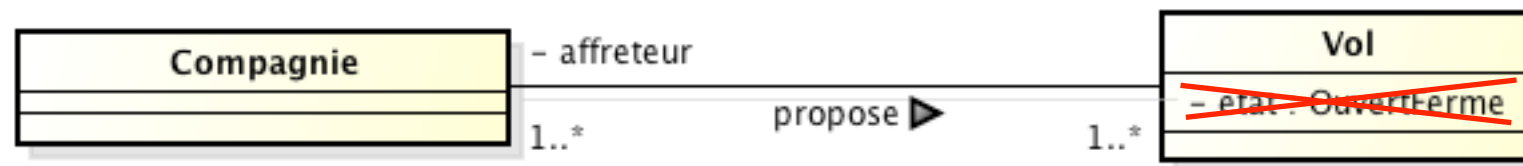
Exemple: Vols

1. les vols sont proposés par différentes compagnies



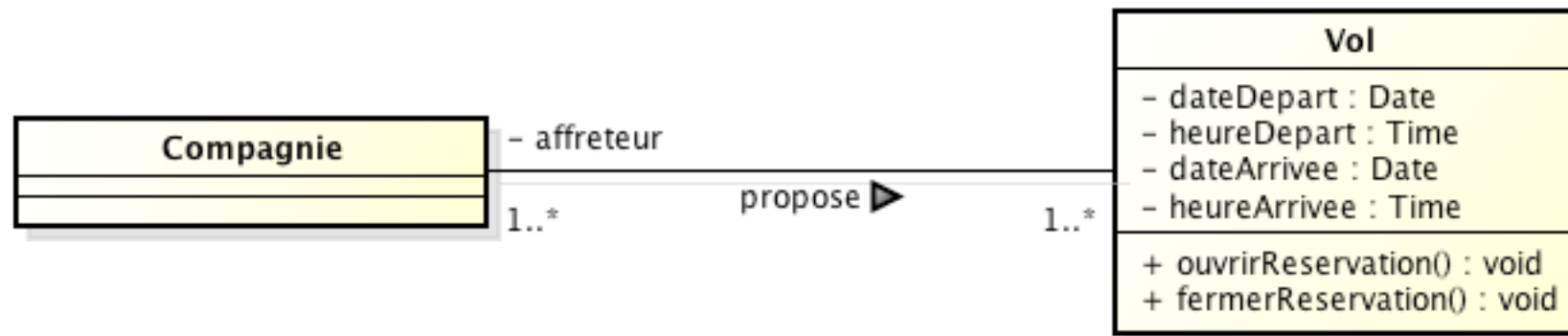
Exemple: Vols

2. la *compagnie* décide de l'**ouverture** et la **fermeture** à la réservation

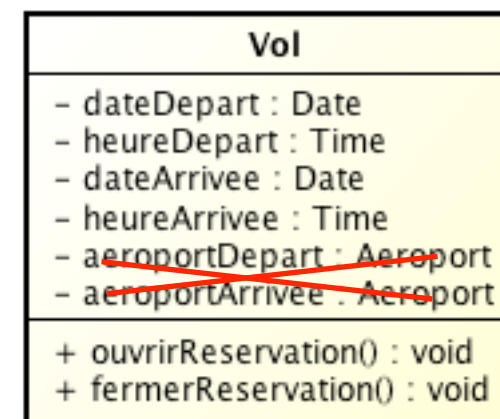


Exemple: Vols

7. un vol a une **date/horaire** de départ et une autre d'arrivée

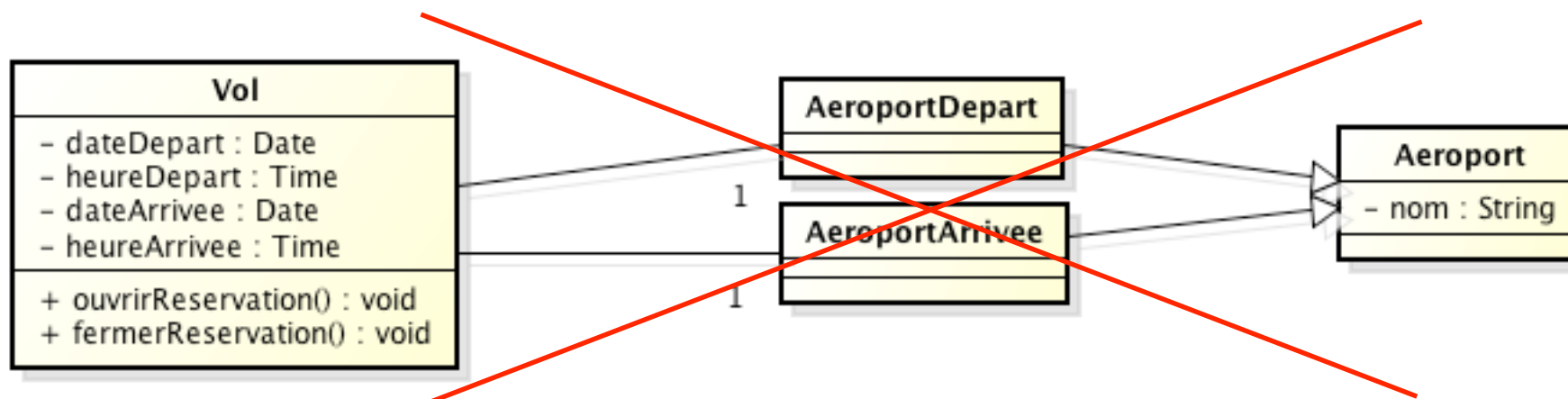
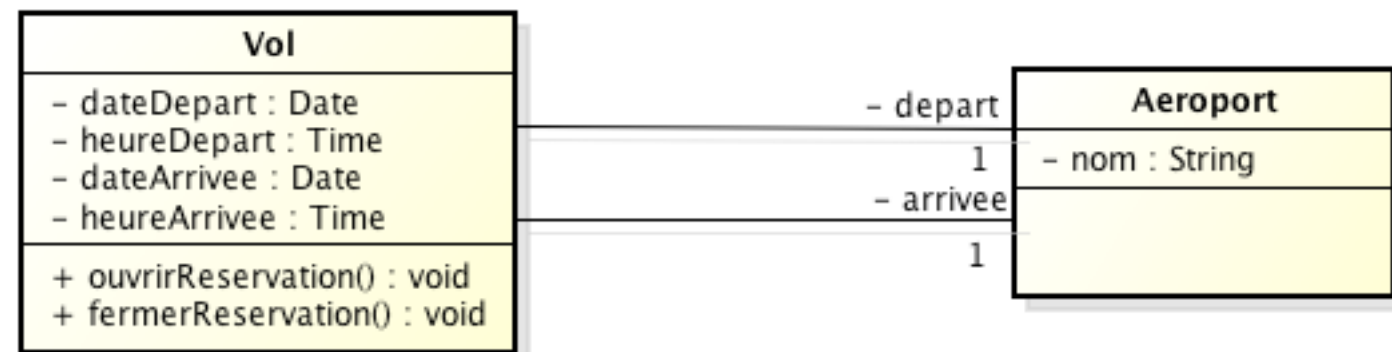
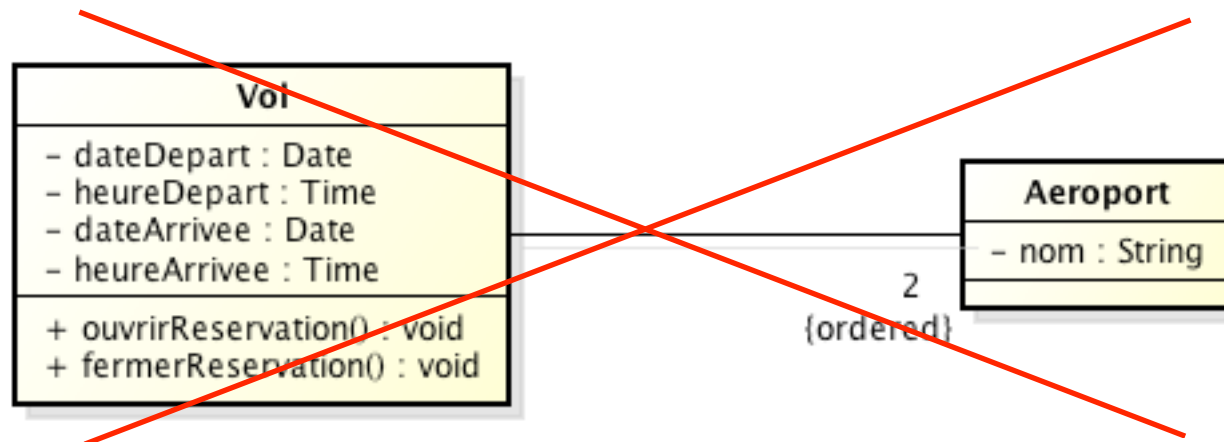


6. un vol a un **aéroport de départ** et un **aéroport d'arrivée**



Exemple: Vols

6. un vol a un **aéroport de départ** et un **aéroport d'arrivée**



Exemple: Vols

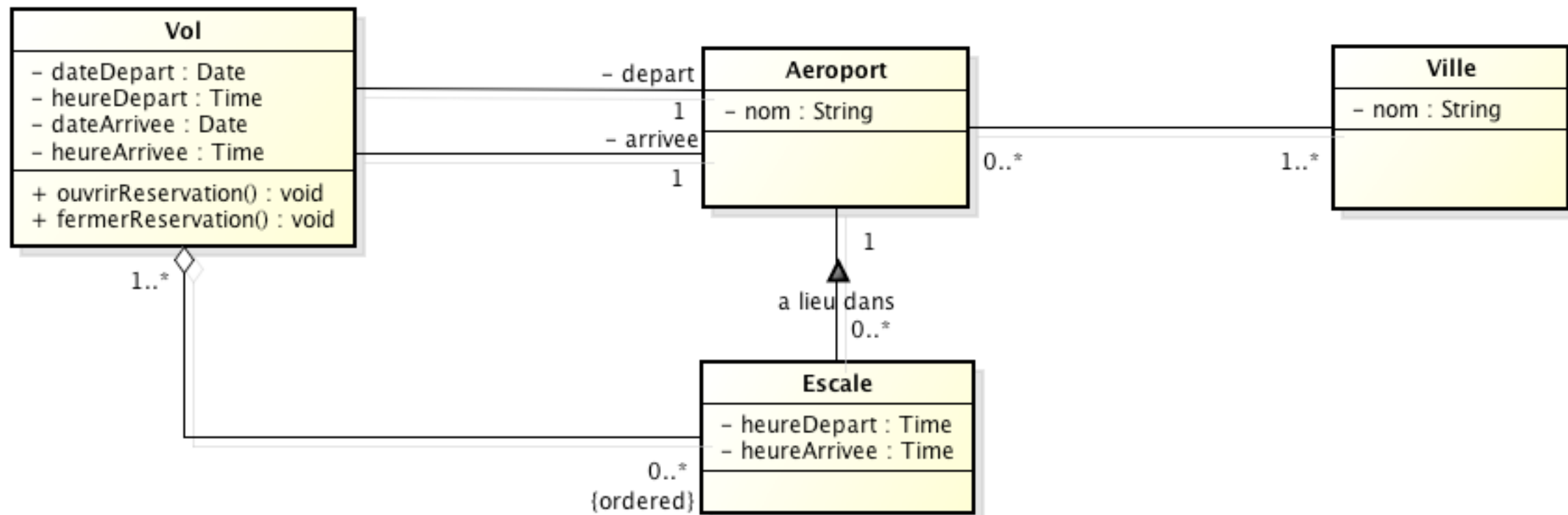
10. un *aéroport* dessert une ou plusieurs **vil**les



Exemple: Vols

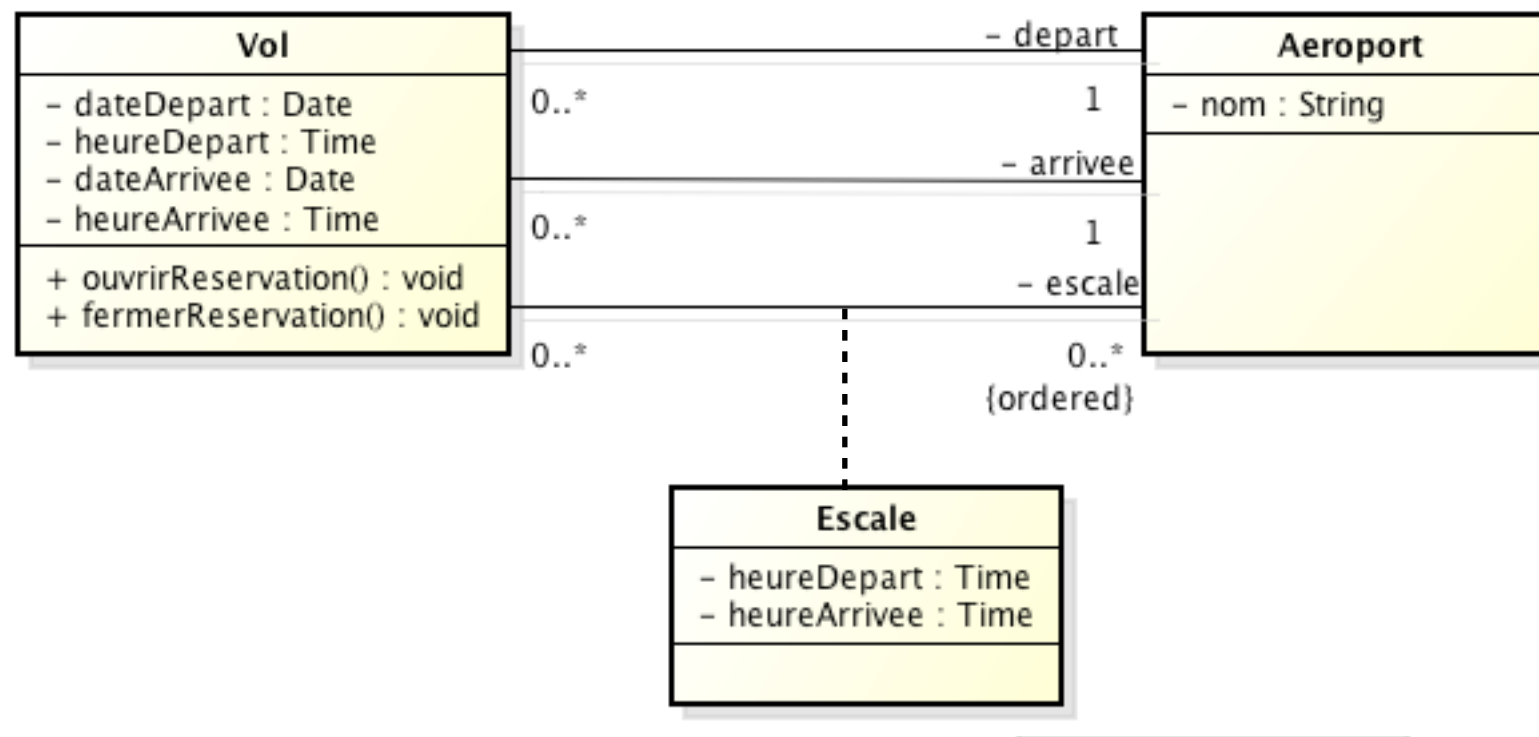
8. un vol peut comporter des **escales**

9. une **escale** a une **heure de départ** et une **heure d'arrivée**



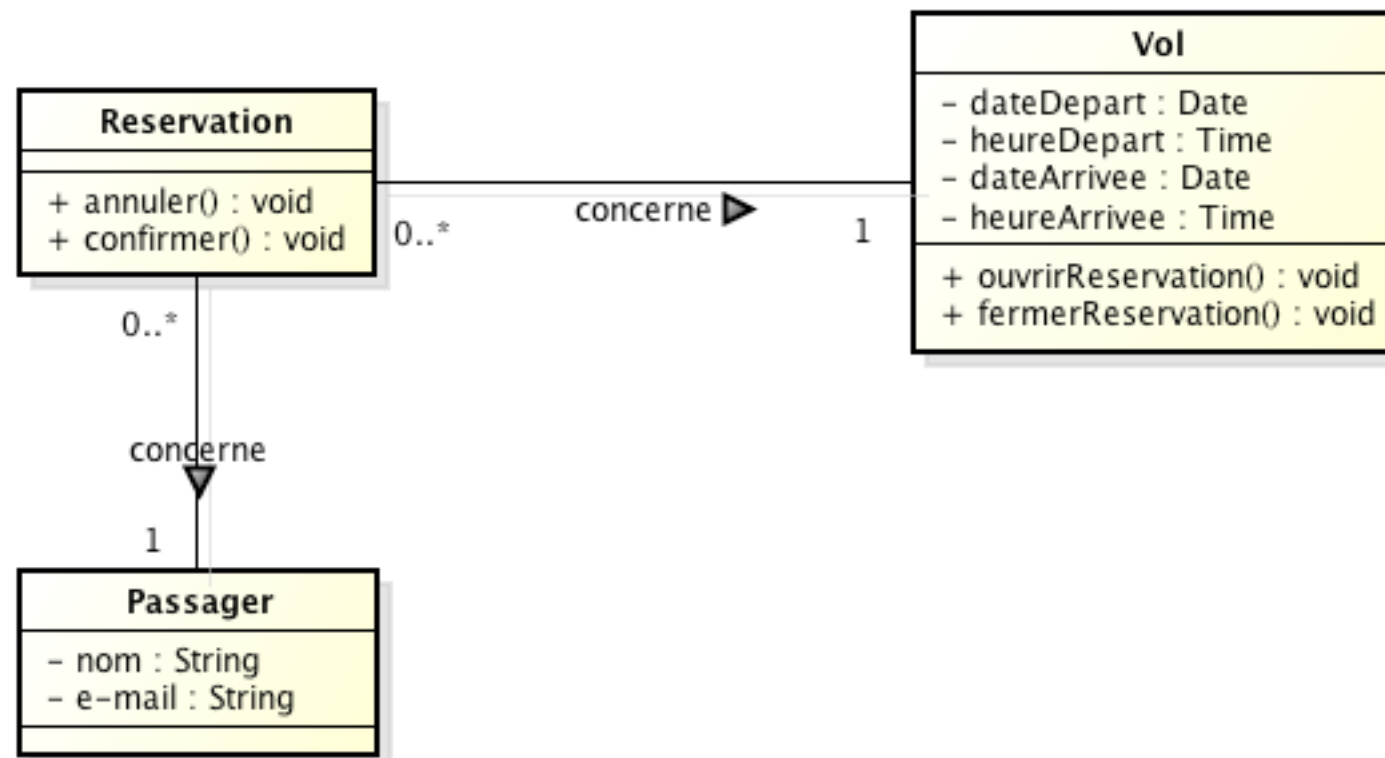
Exemple: Vols

8. un vol peut comporter des **escales**



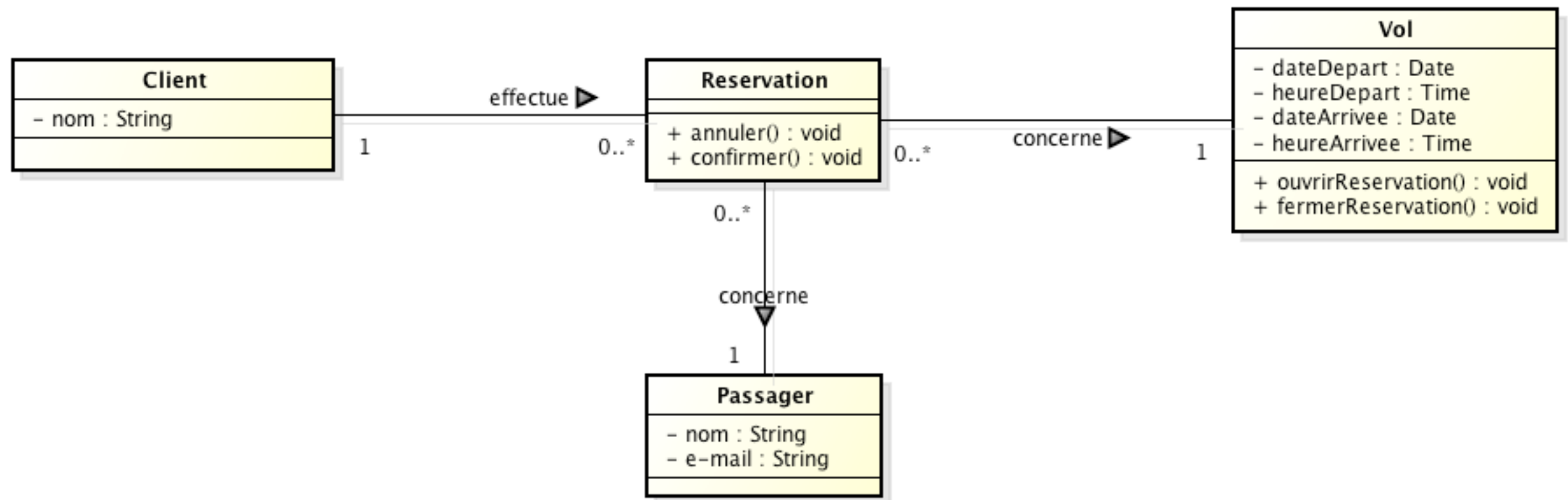
Exemple: Vols

- 4. une réservation concerne un seul vol et un seul passager
- 5. une réservation peut être confirmée ou annulée

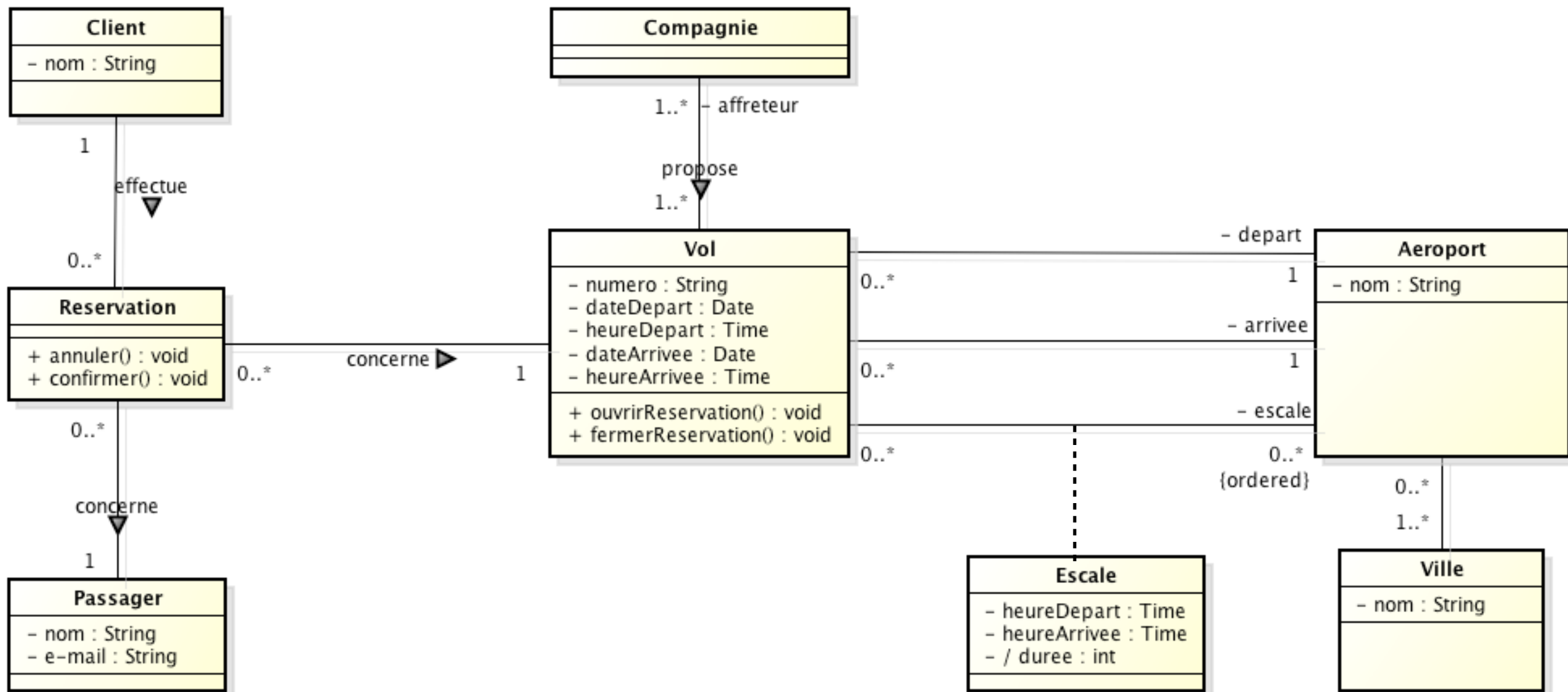


Exemple: Vols

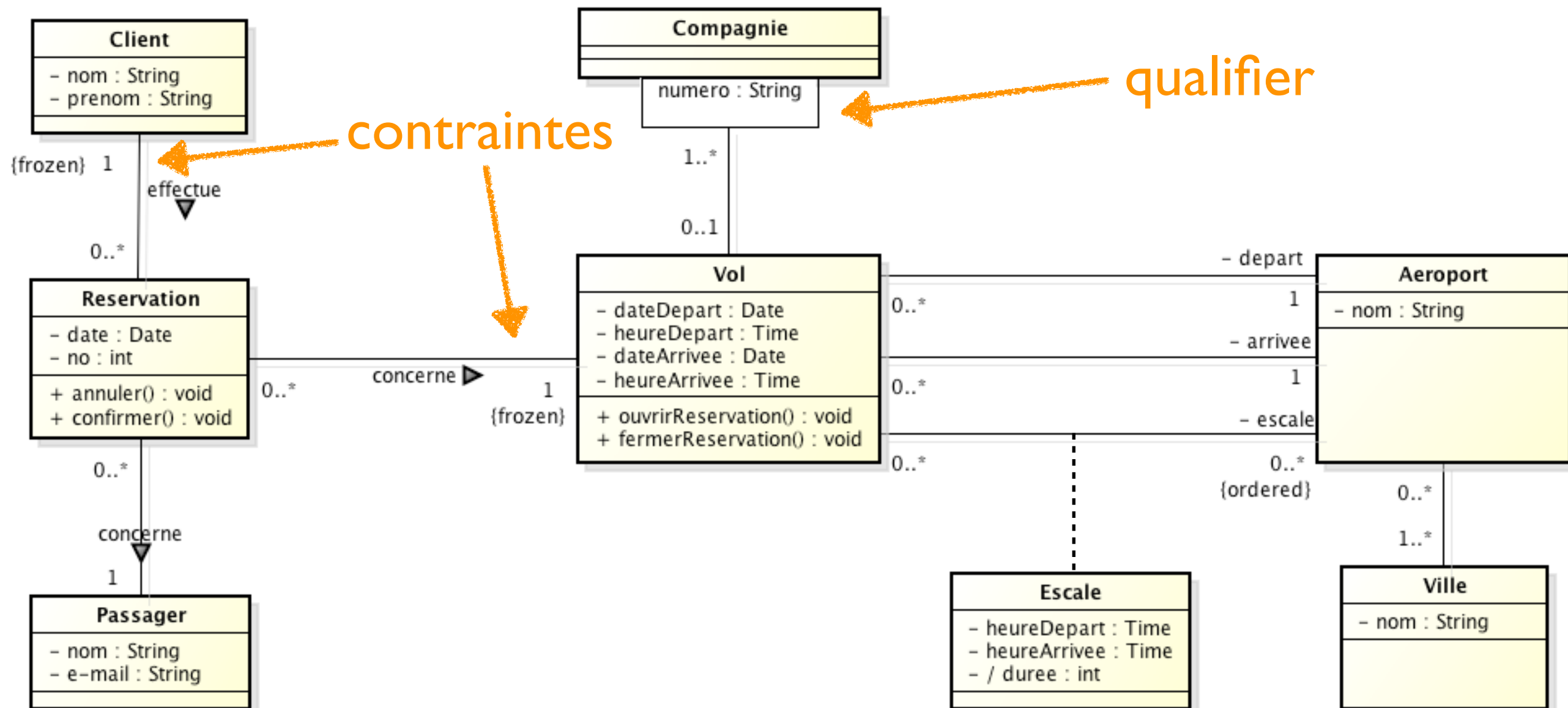
3. un **client** peut réserver des vols pour plusieurs **passagers**
4. une **réservation** concerne un seul vol et un seul **passager**
5. une **réservation** peut être **confirmée** ou **annulée**



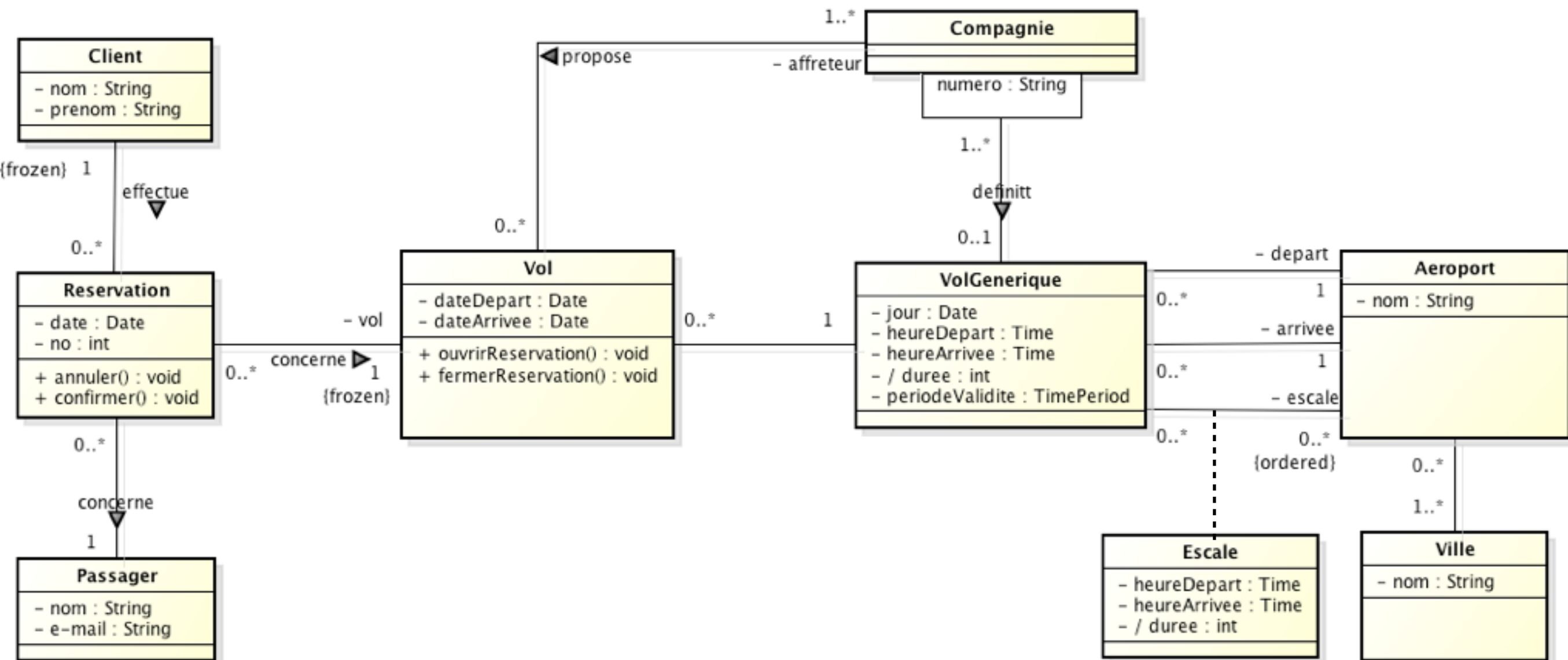
Exemple: Vols



Exemple: Vols



Exemple: Vols



Object Constraint Language (OCL)

- Un langage permettant de décrire des contraintes présentes dans un module
 - ▶ une expression OCL décrit un énoncé logique (une contrainte) à propos du système
 - ▶ une contrainte ne doit pas conduire à des effets secondaires
 - ▶ un énoncé OCL peut spécifier quelles valeurs doivent avoir les attributs d'une classe ou d'une association

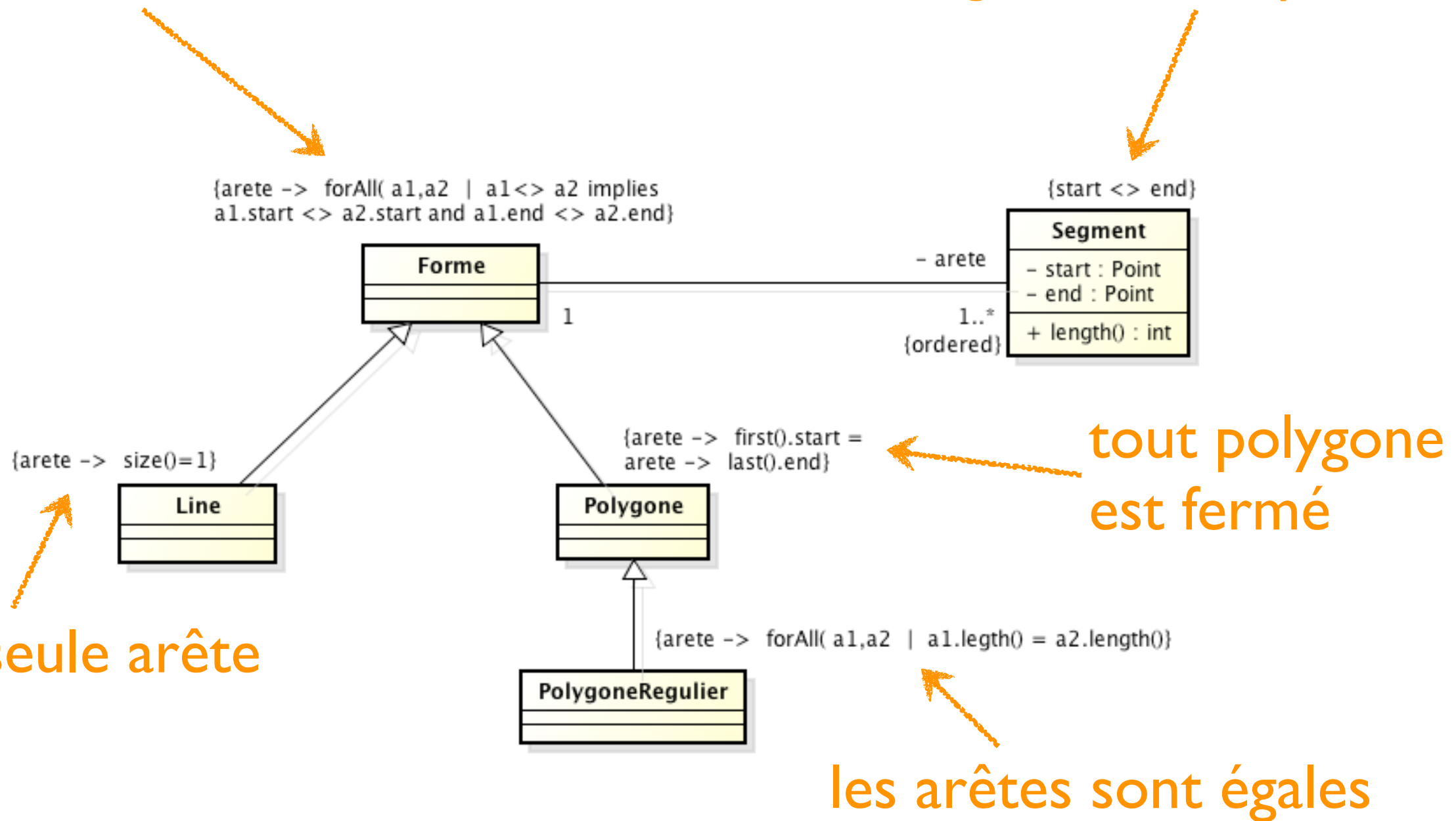
Les énoncés OCL

- Référence à des noms de rôle, d'associations, d'attributs et de résultats d'opérations
- Valeurs logiques étant vrai ou fausse
- Opérateurs logiques tels que non, et, ou, =, >, <, <>
- Chaînes de caractères
- Entiers ou nombres réels
- Opérations arithmétiques *, /, +, -

Exemple

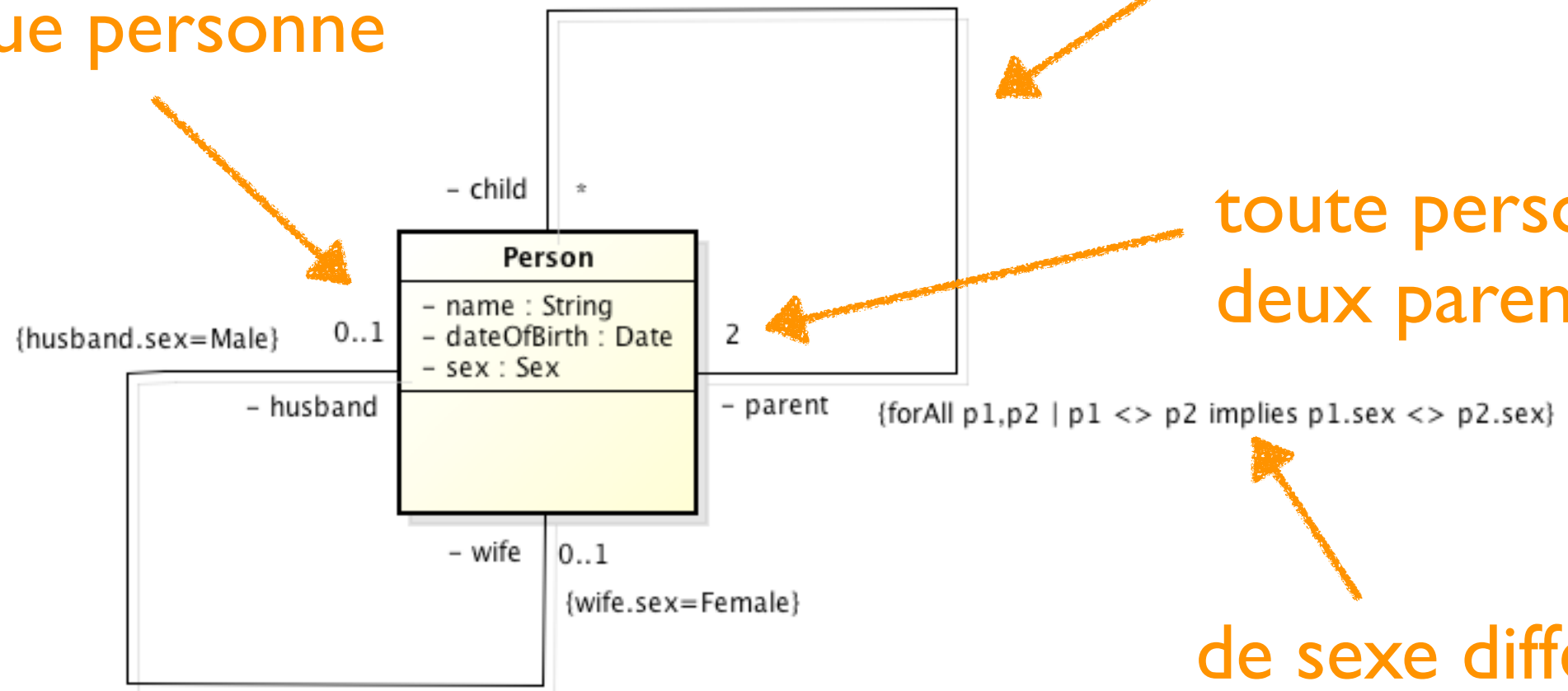
les extrémités des arêtes sont mutuellement différents

un segment n'est pas un point



Exemple: généalogie

maximum un mariage
pour chaque personne



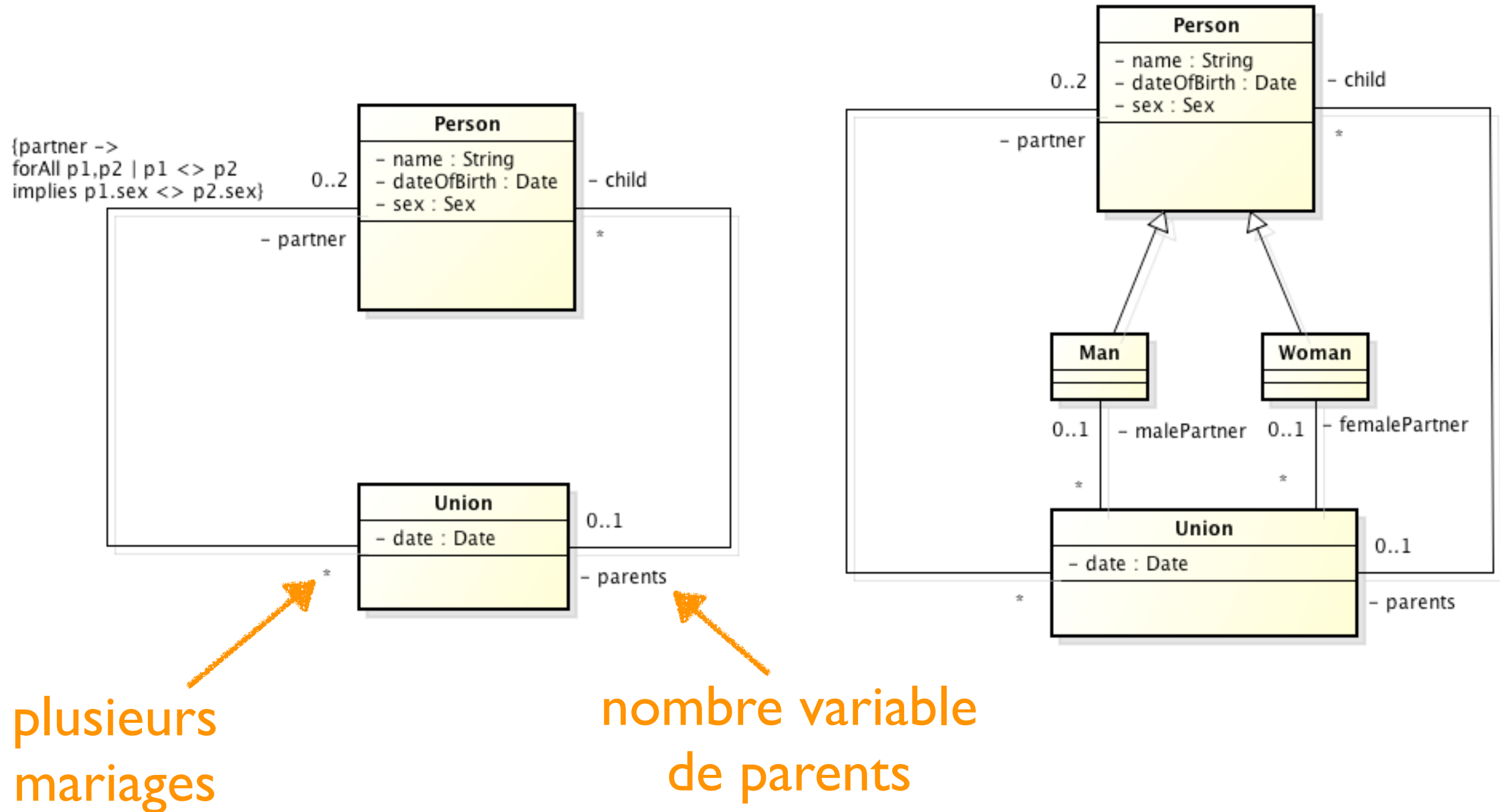
enfants

toute personne a
deux parents

de sexe différent

mariages

Exemple: généalogie



Questions ?