- 2 -

Modélisation structurelle (classes)

UNIVERSITÉ DE LORRAINE

Analyse et Conception de Logiciels

Horatiu CIRSTEA

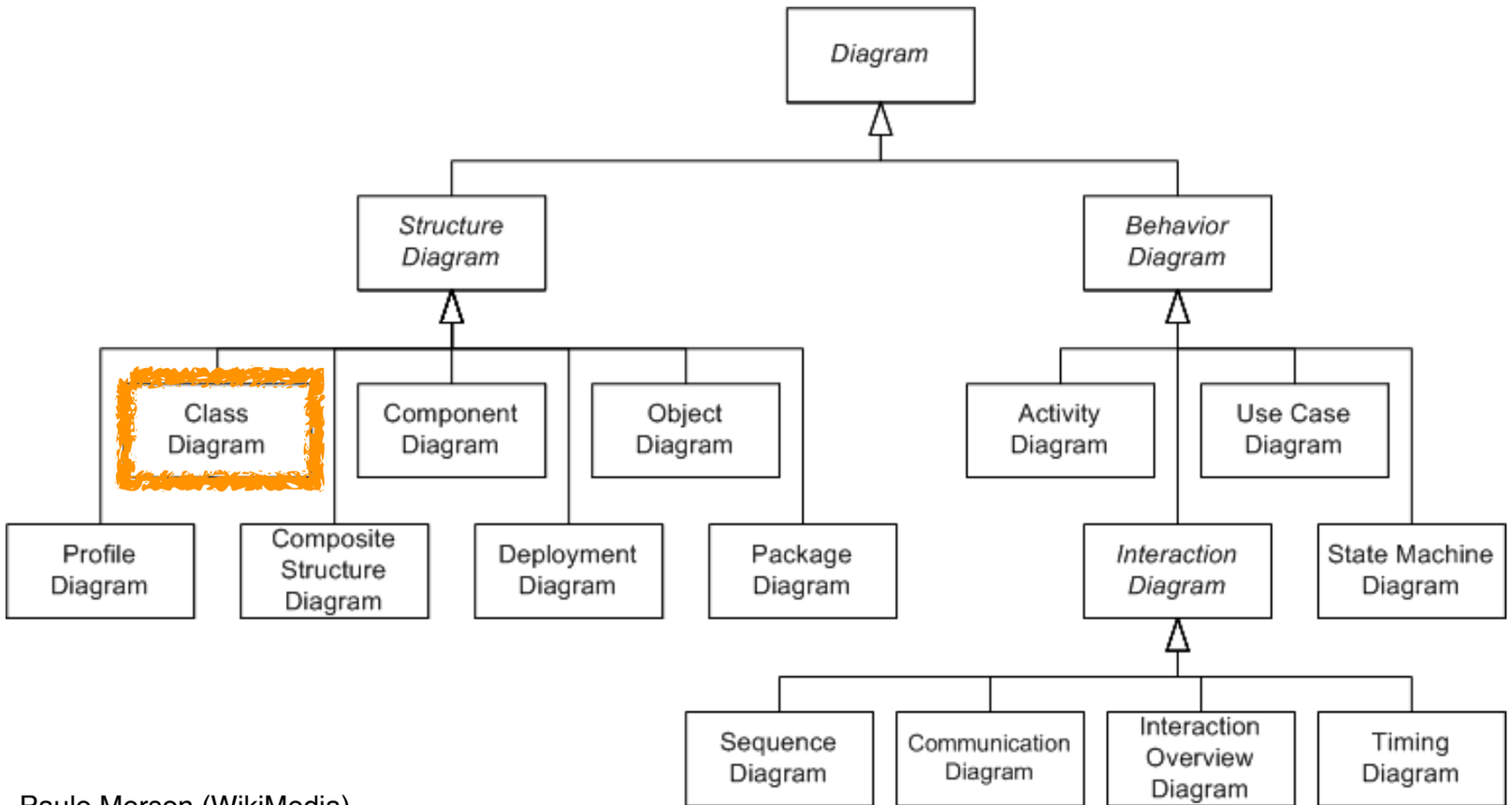
Perspectives

- ***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

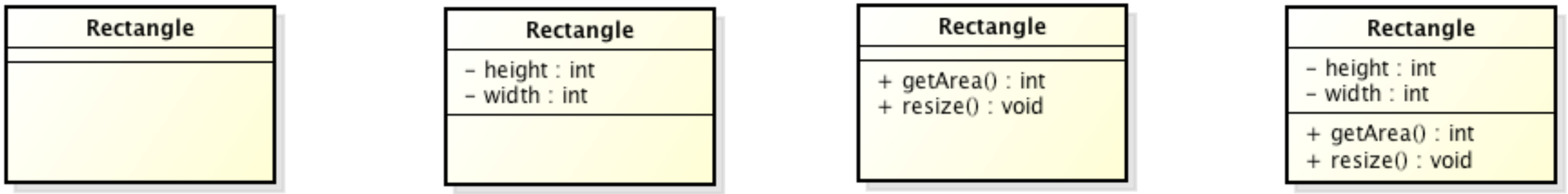
- **Classes**
  - les types de données disponibles
- **Attributs**
  - données se trouvant dans les classes et leurs instances
- **Opérations**
  - les fonctions exécutées par les classes et leurs instances

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},
{ordered}, {unique},
prop-constraint
```

```
±/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
```

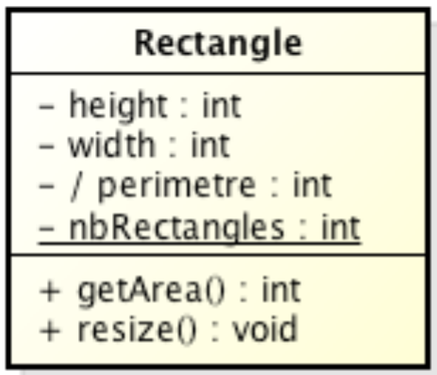
```
± name(param: pType…): returnType {prop}
```

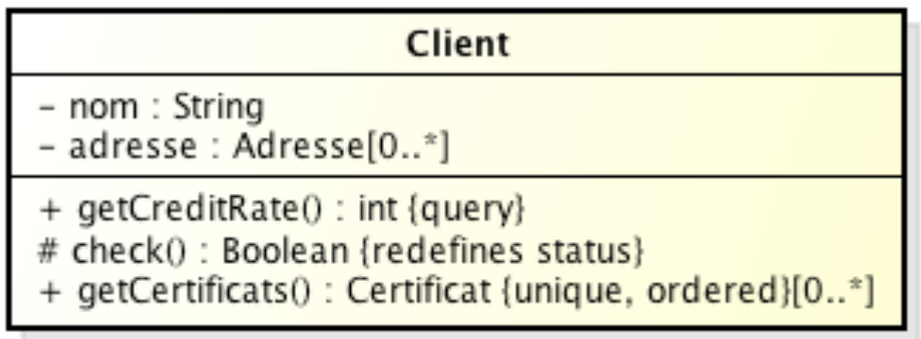
Syntaxe des paramètres

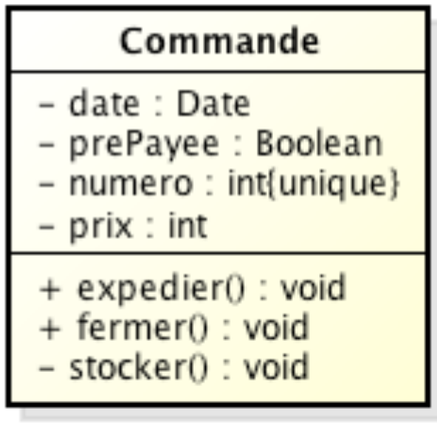
```
direction nom : type [multiplicity]=val
```

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

Exemples

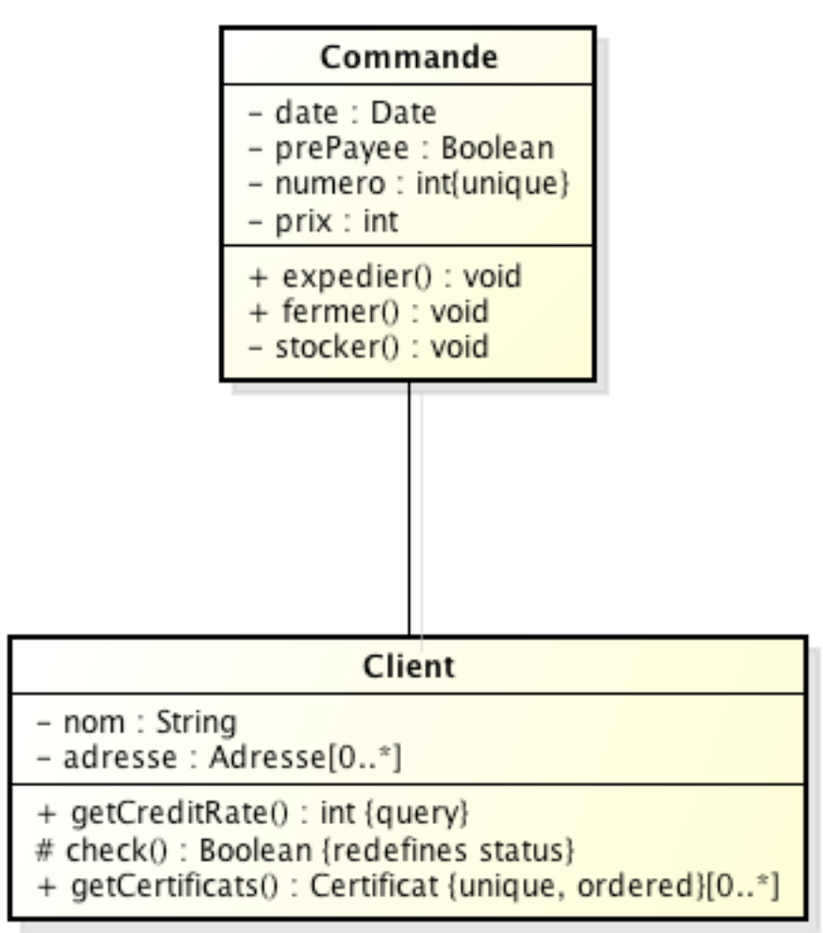

Rectangle
- height : int
- width : int
- / perimetre : int
- nbRectangles : int
+ getArea() : int
+ resize() : void

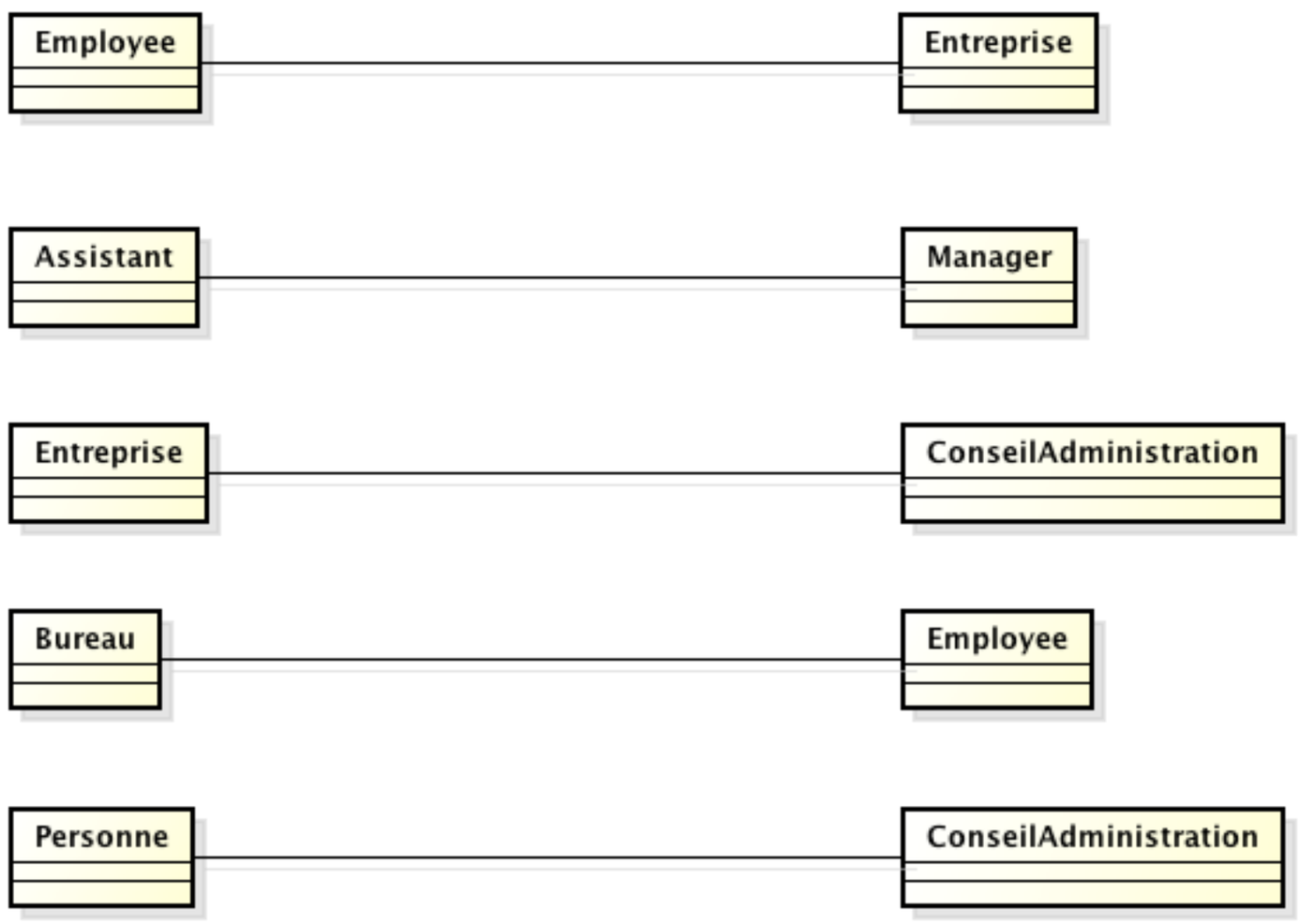

Client
- nom : String
- adresse : Adresse[0..*]
+ getCreditRate() : int {query}
# check() : Boolean {redefines status}
+ getCertificats() : Certificat {unique, ordered}[0..*]


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





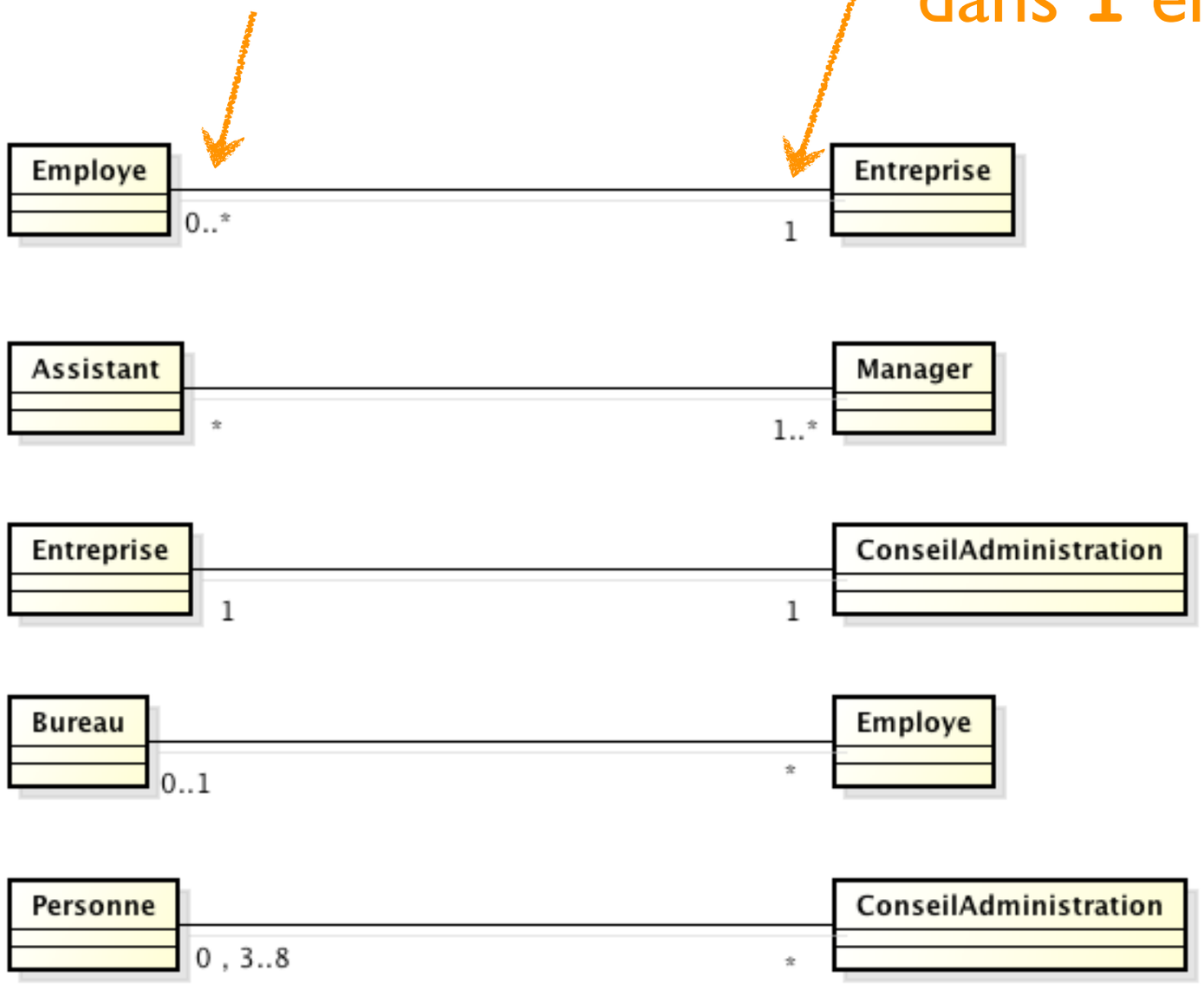
Associations

- Concepts
  - multiplicité
  - terminaison
  - navigabilité
  - arité

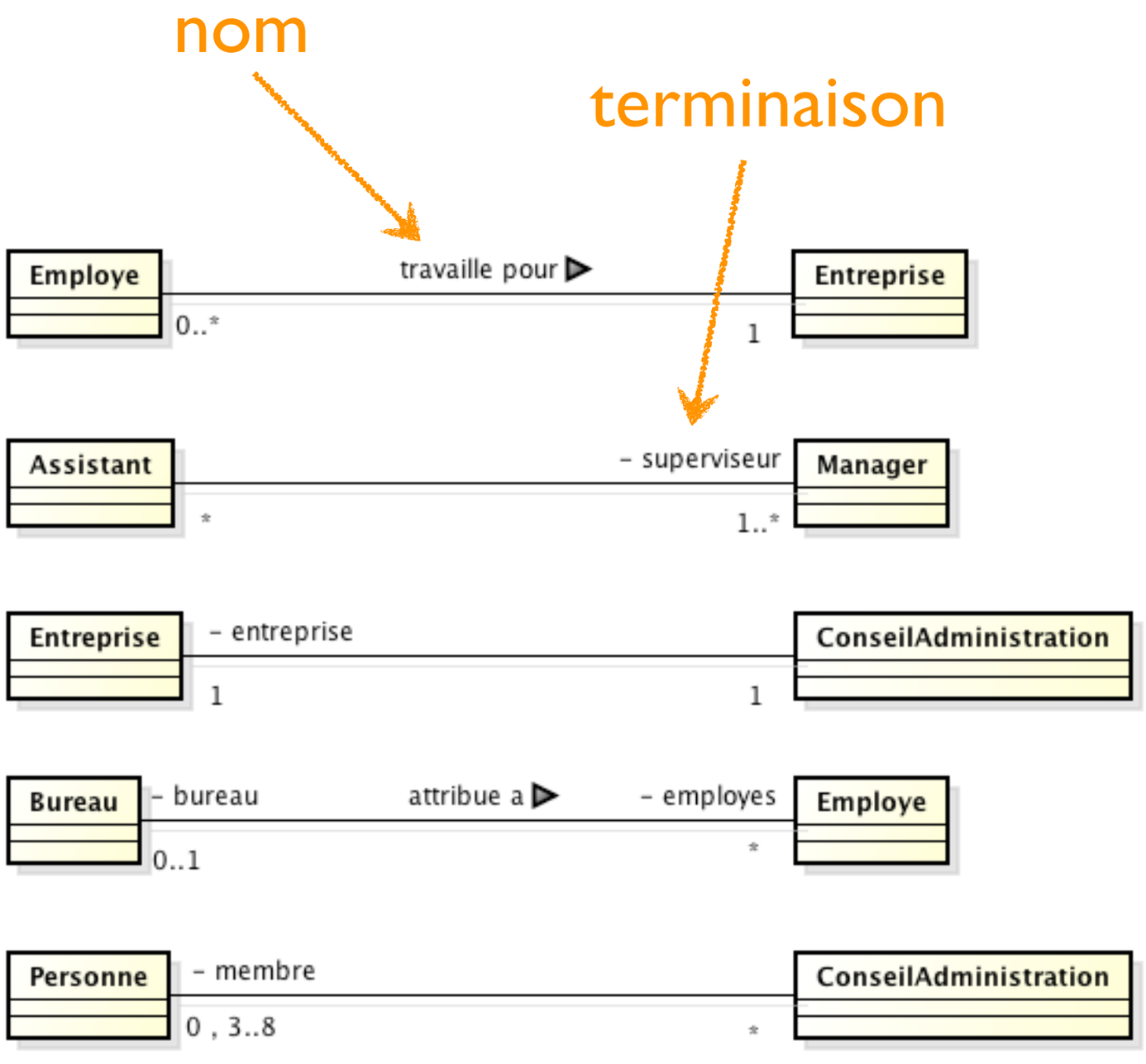
Associations - *multiplicité*

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

un **Employé** travaille
dans 1 entreprise

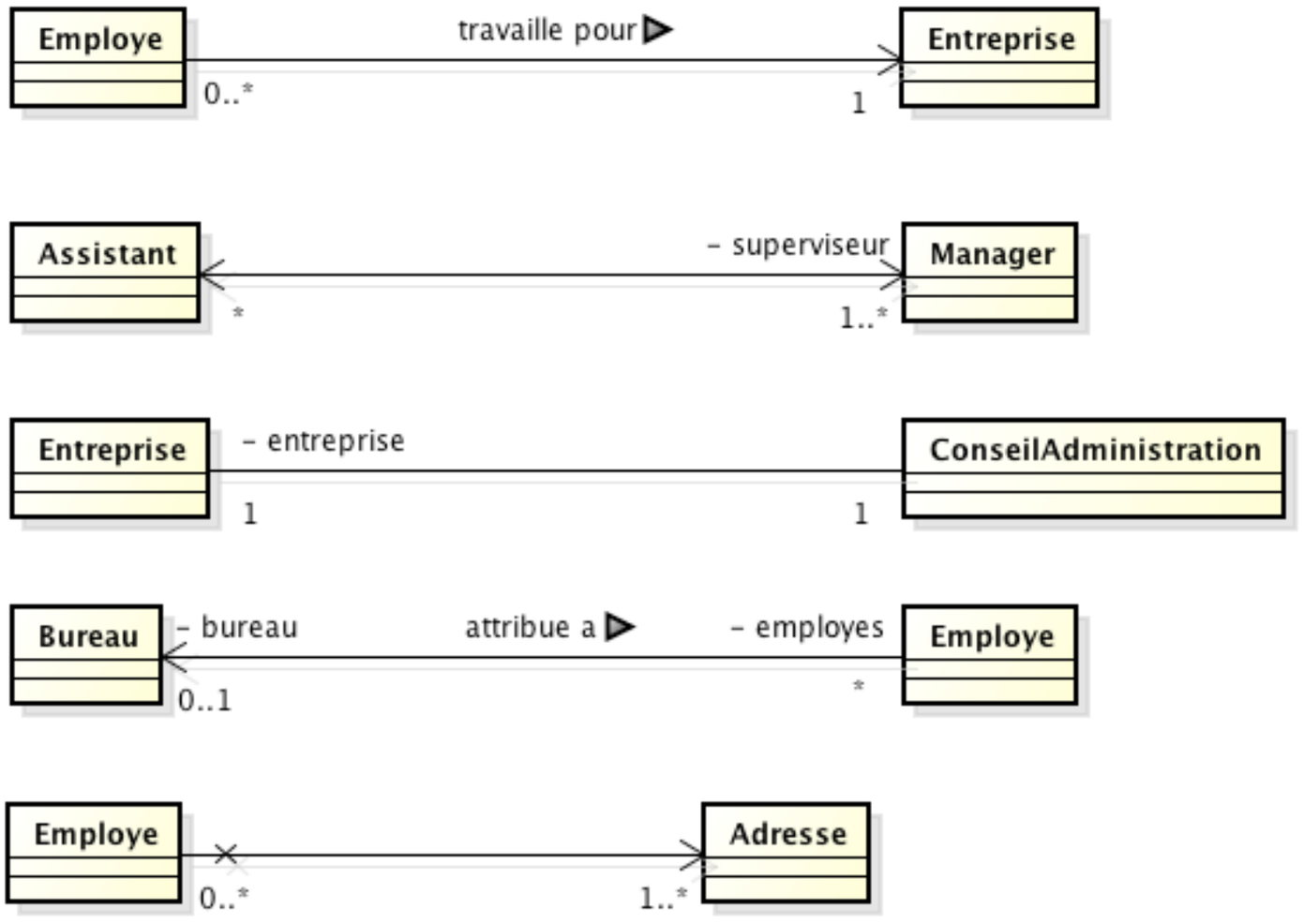


Associations - *terminaison*

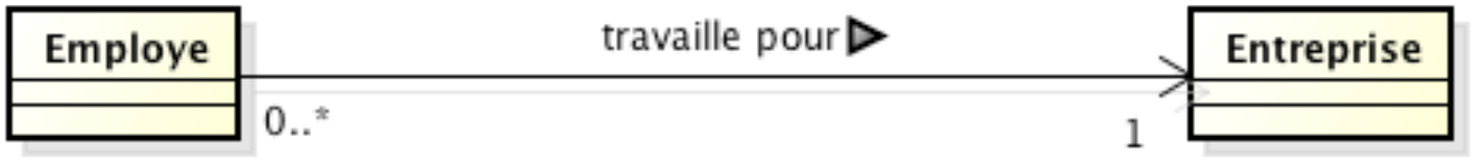


Associations - *navigabilité*

- navigable, non-navigable, non-spécifié

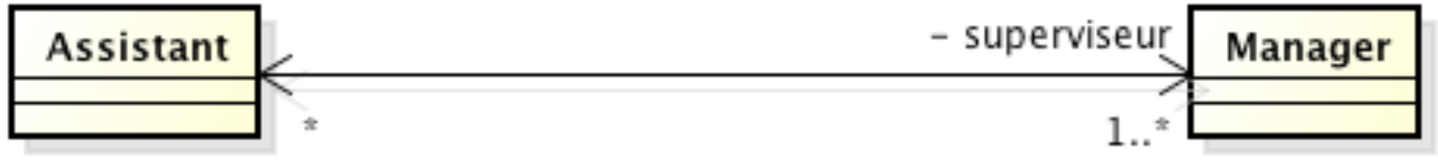


Analyser et validèr les associations



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

Analyser et valider les associations

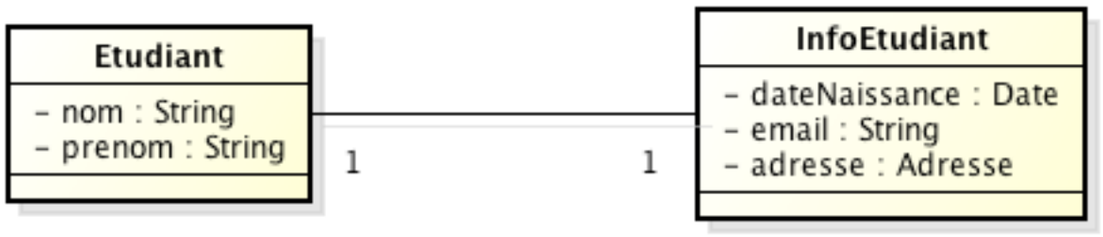


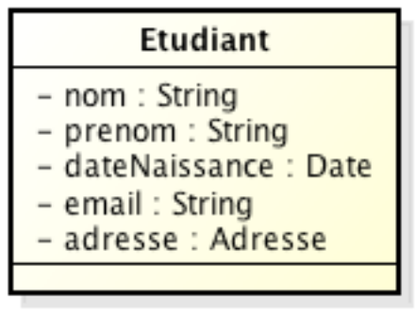
- Un(e) assistant(e) peut travailler pour plusieurs superviseurs
- Un superviseur peut avoir plusieurs assistants
- Certains superviseurs peuvent n'avoir aucun assistant
- Est-il possible qu'un(e) assistant(e) puisse se retrouver sans superviseur?

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

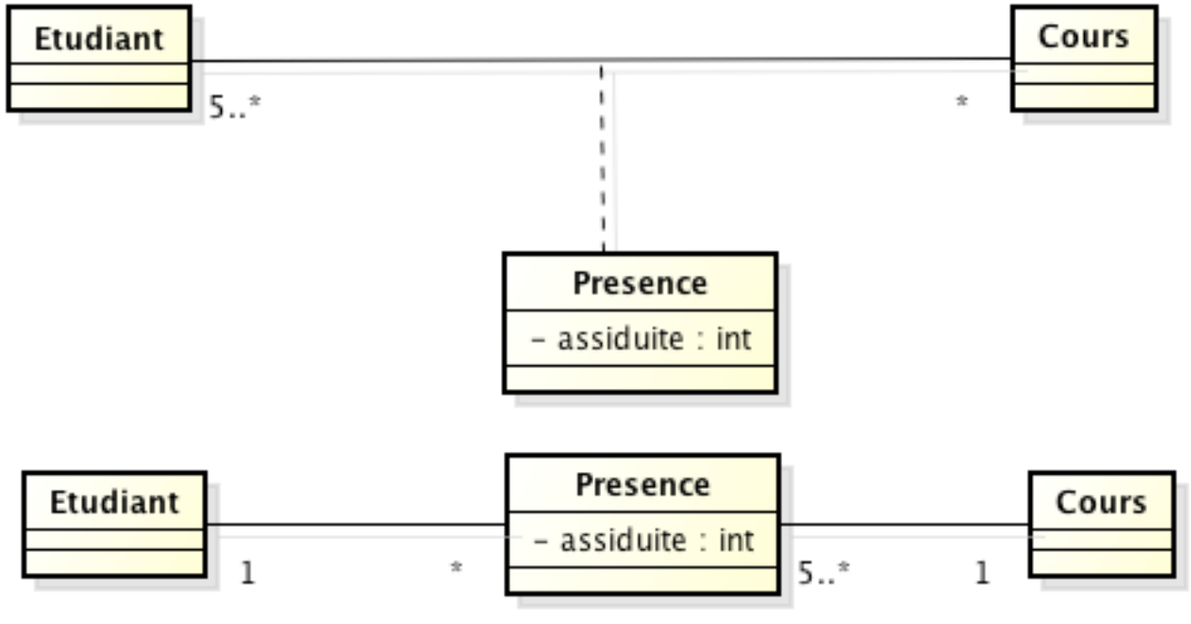
Analyser et valider les associations

- 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

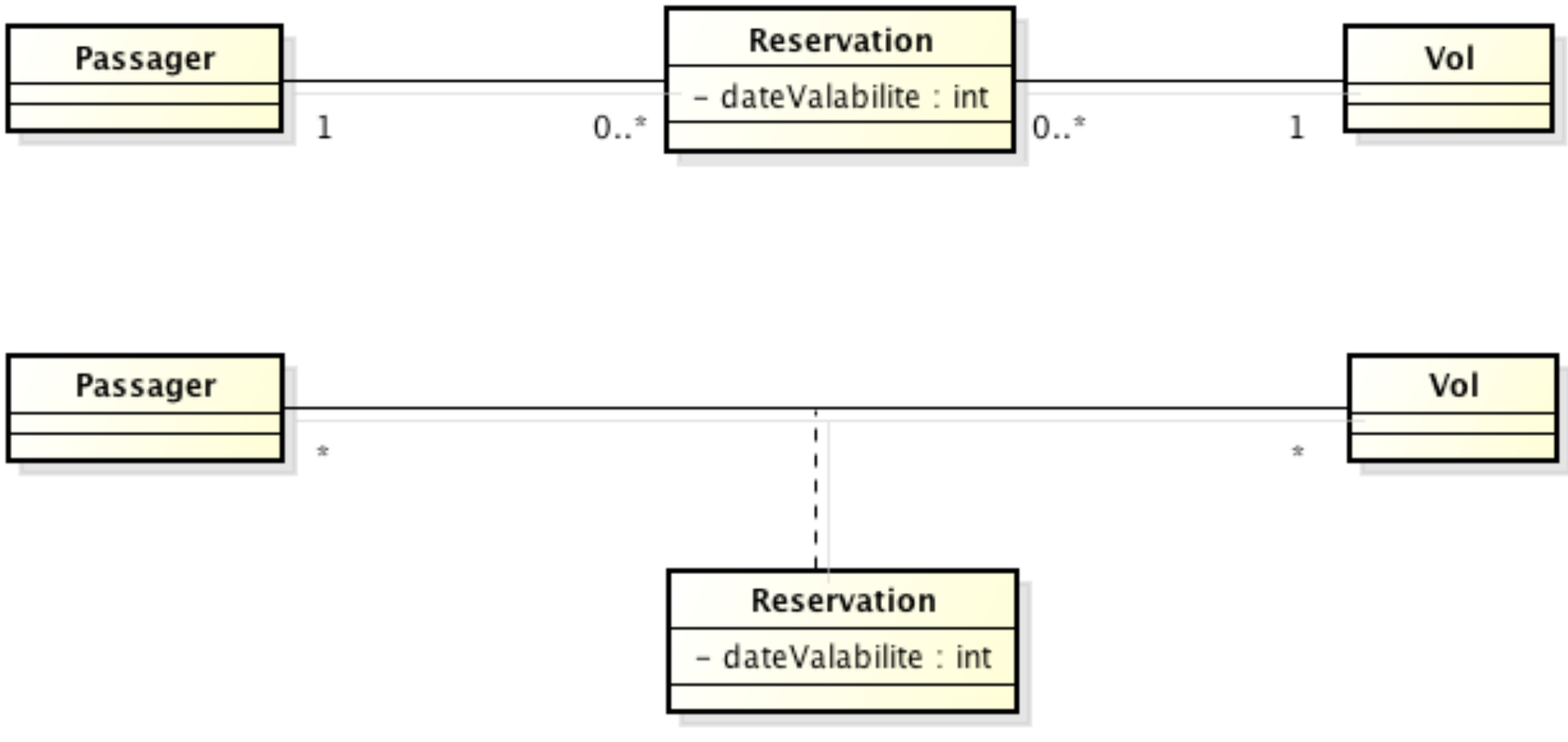
- Il arrive qu'un attribut ne puisse être attaché à aucune des deux classes d'une association

➡ il peut être attaché à l'association



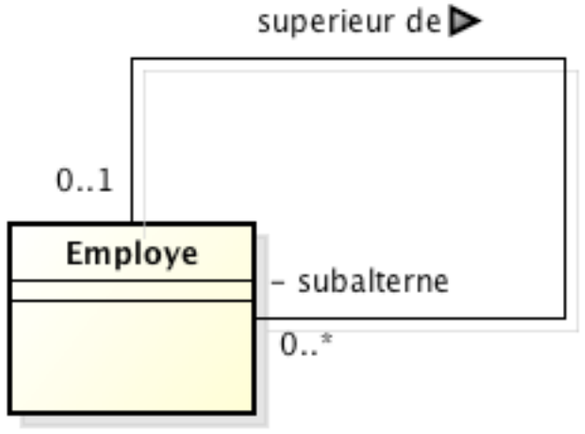
Classes d'association

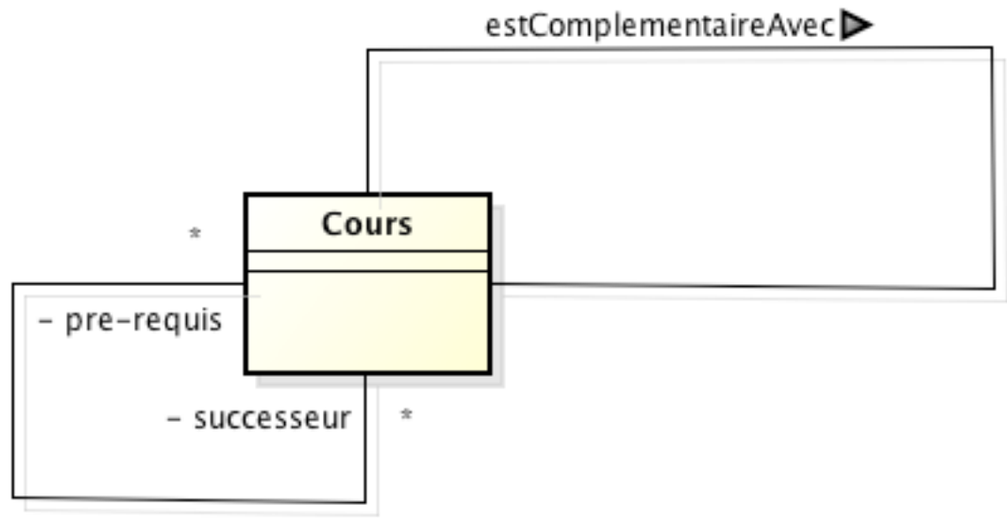
- Modélisations alternatives



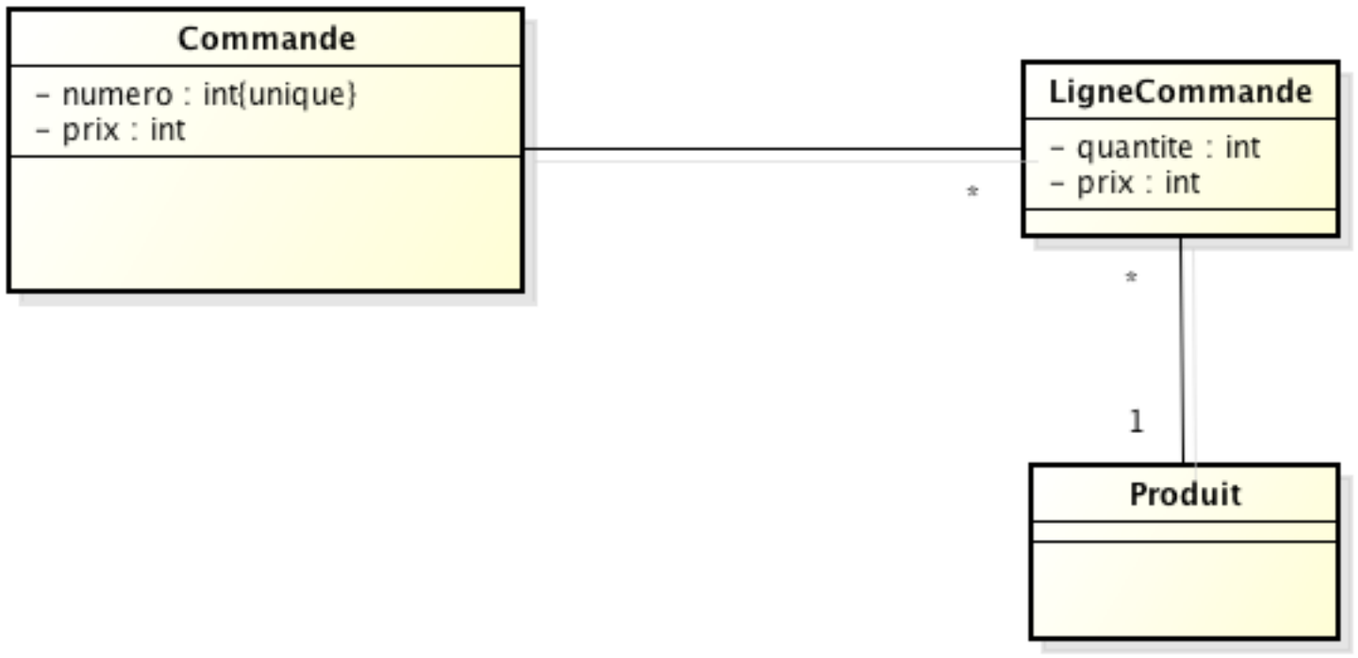
Association réflexive

- Une classe peut être associée à elle-même

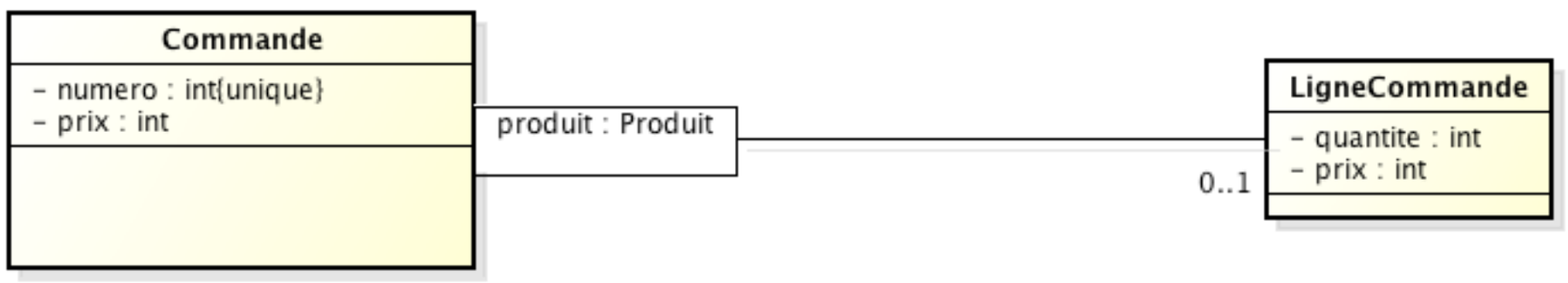




Association qualifiée

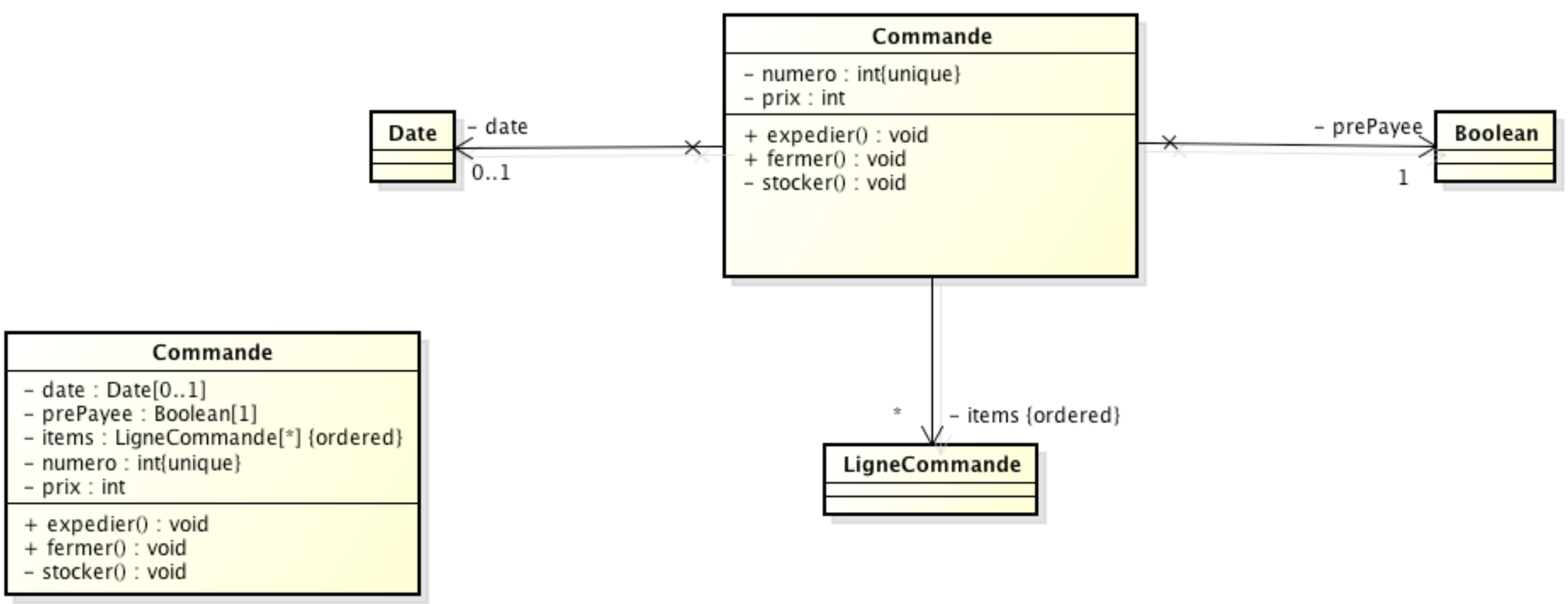


- le « qualifier » est utilisé pour identifier des objets reliés
- correspond à un tableau associatif/dictionnaire

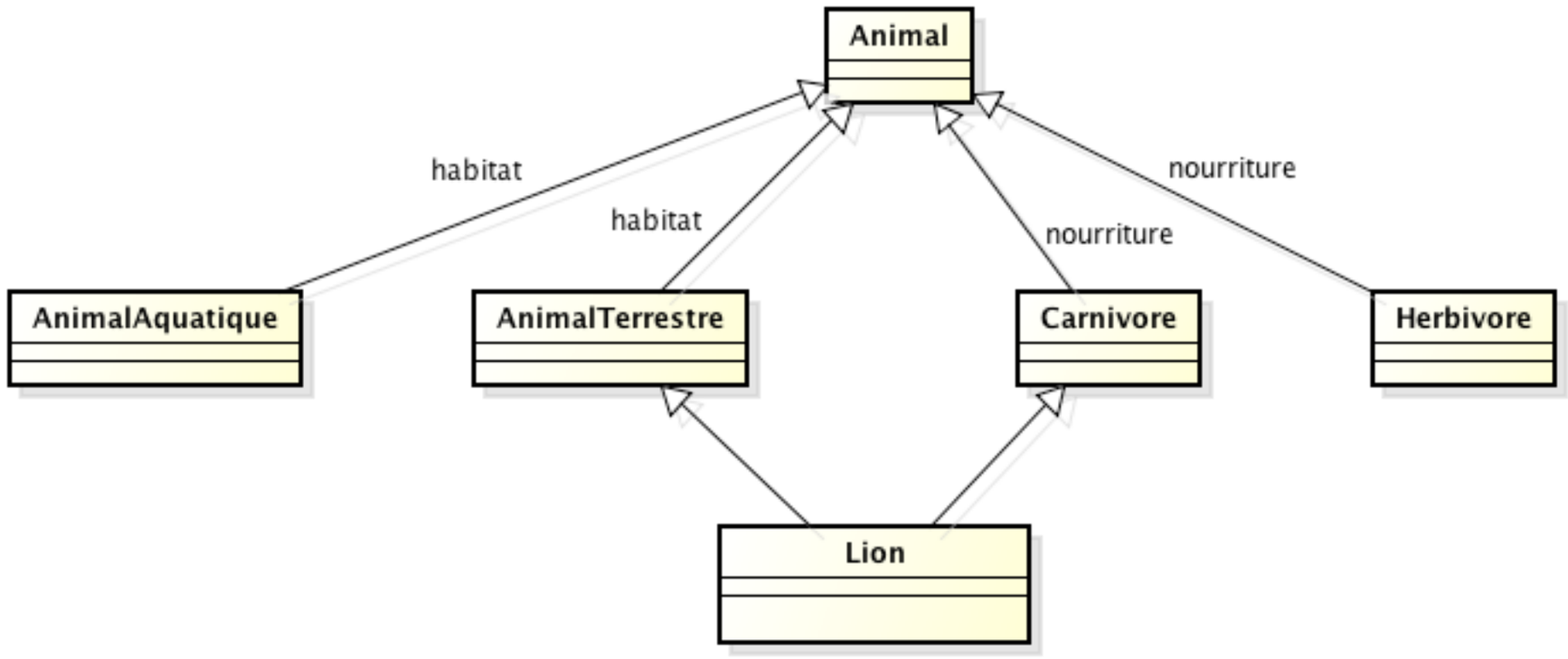


Propriétés

- attributs et associations, comment choisir ?



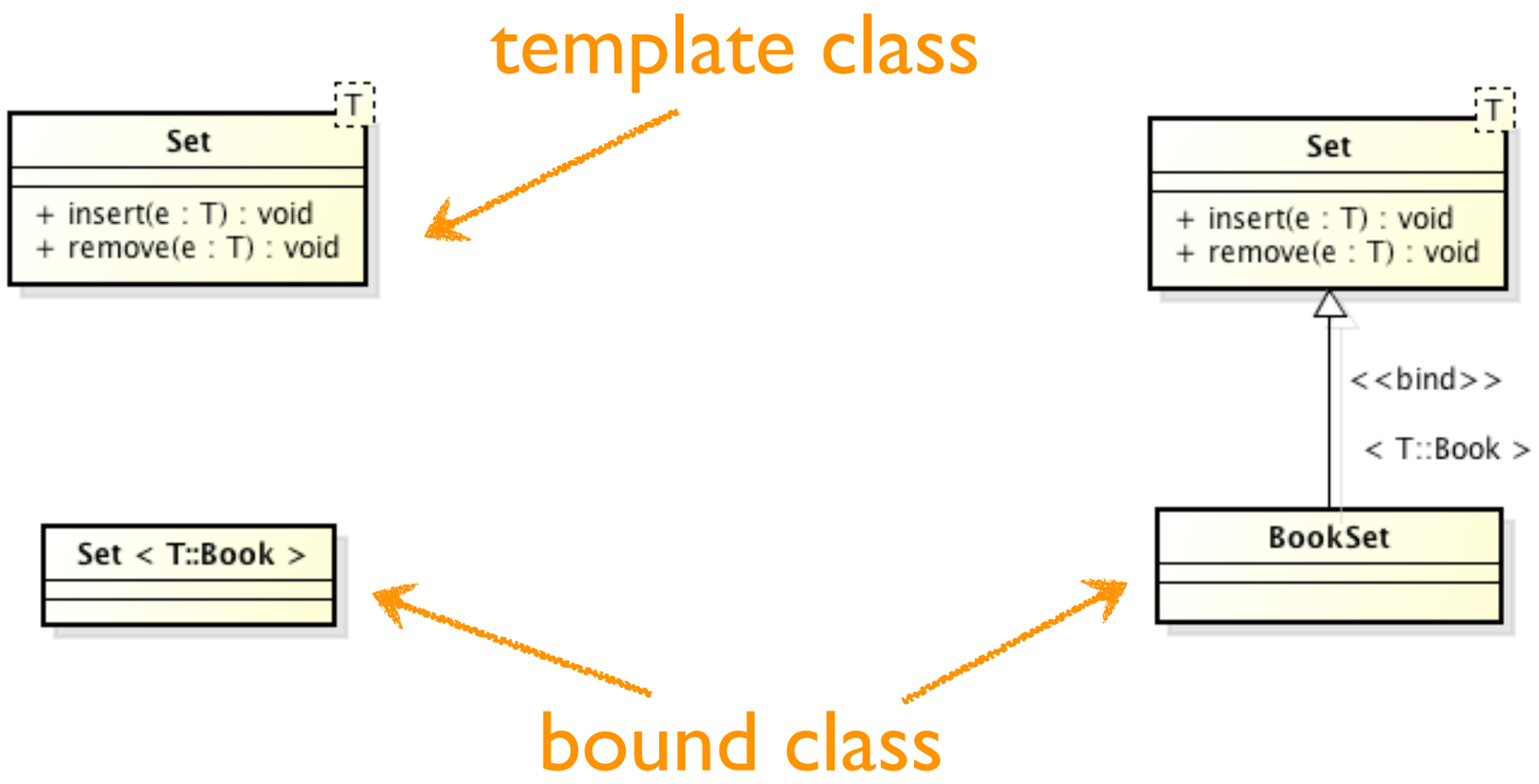
Héritage multiple



- accepter implicitement mais pas défini explicitement

Classes paramétrisées

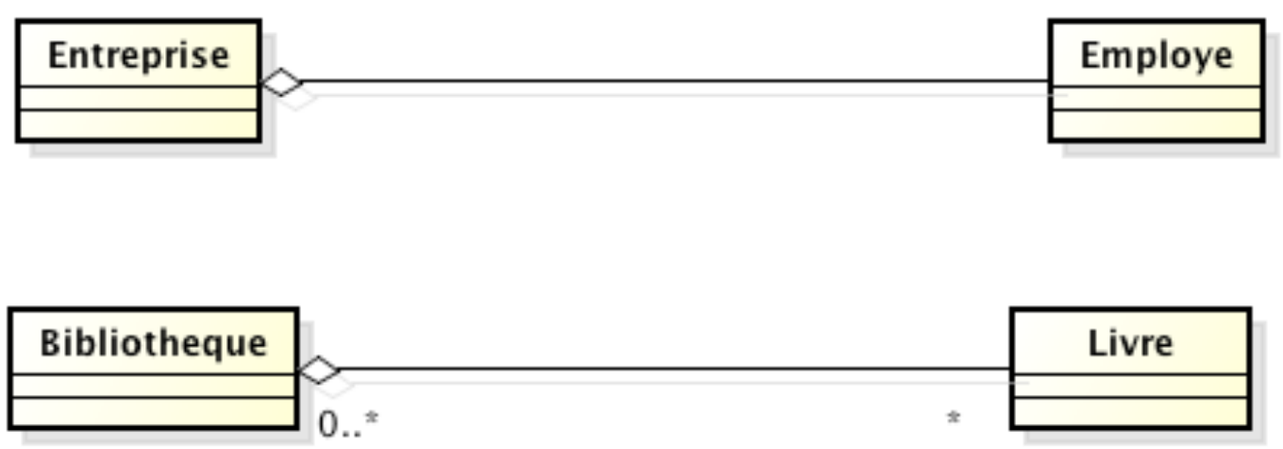
- définition d'une classes en fonction d'un paramètre de type



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:

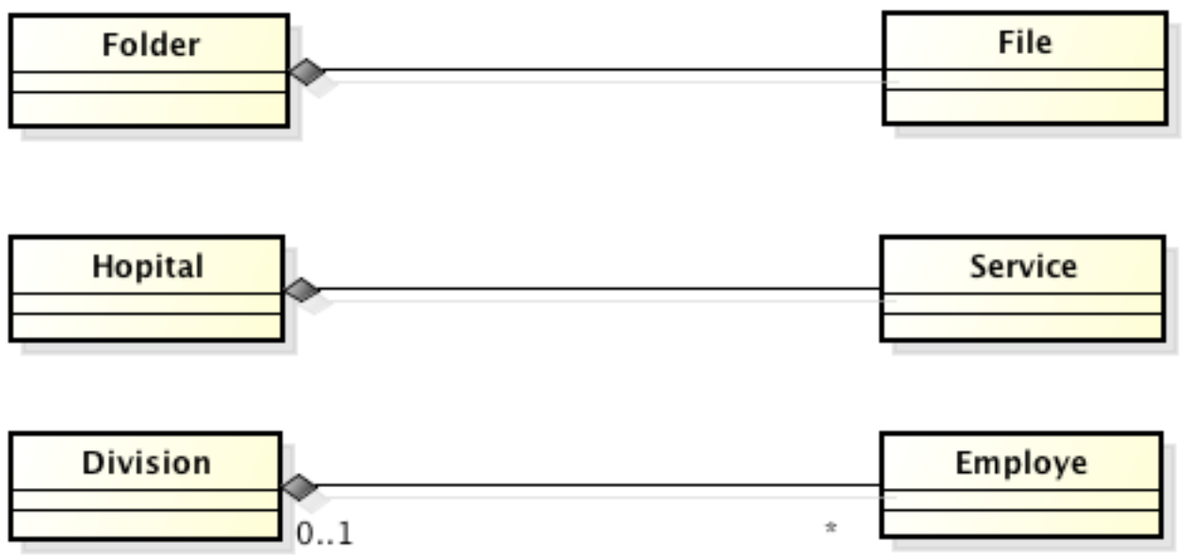
- il y a une relation de type est-une-partie-de
- il y a une relation de type est-composé-de

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

Composition

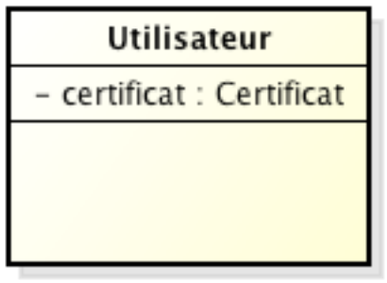
Une composition est une forme forte d'agrégation

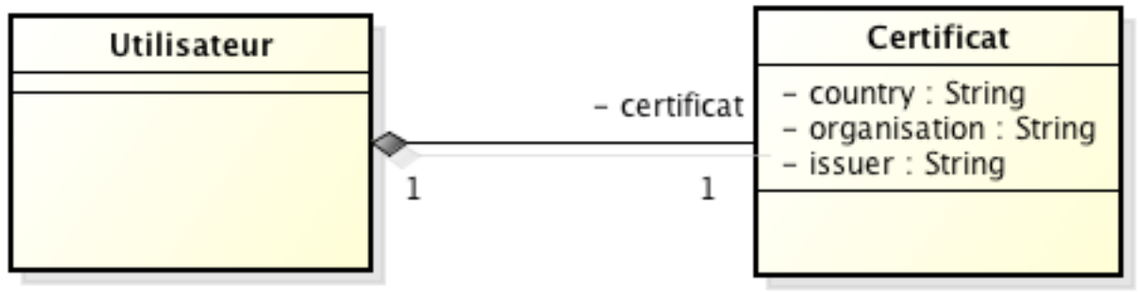
- si l'agrégat est détruit, alors ses parties le sont aussi



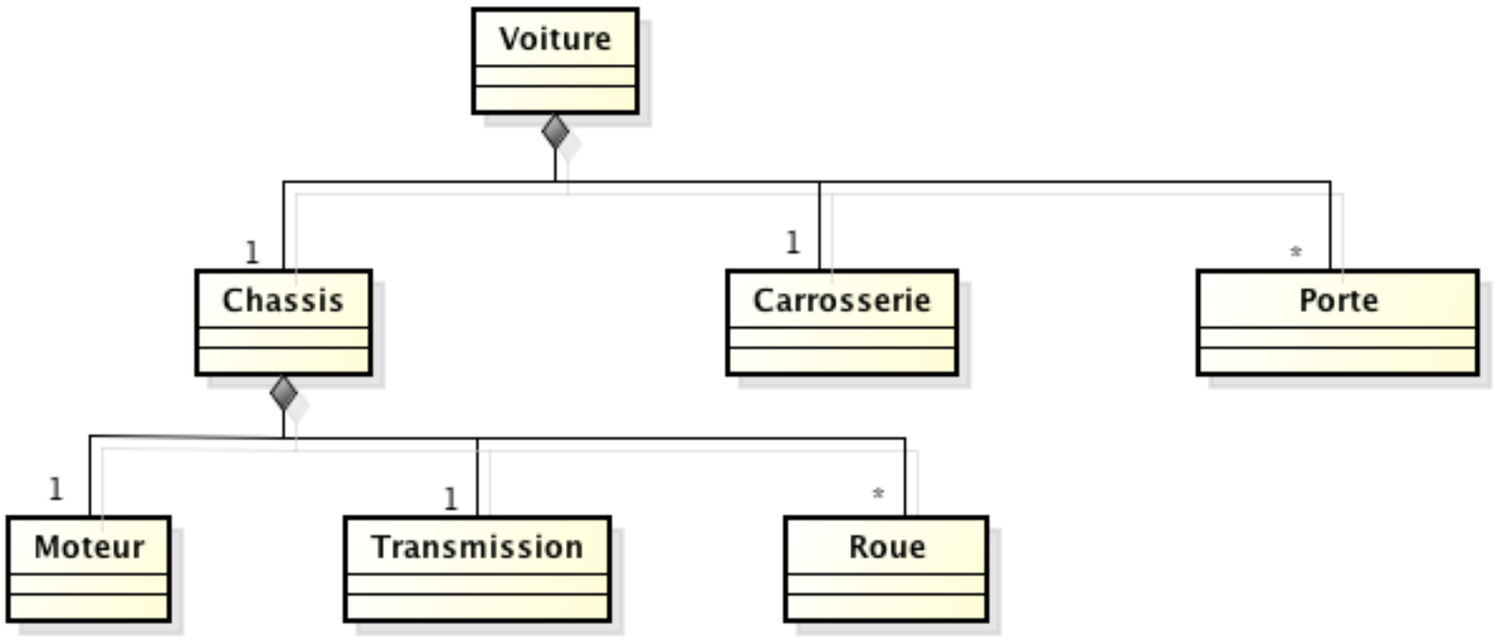
Composition

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

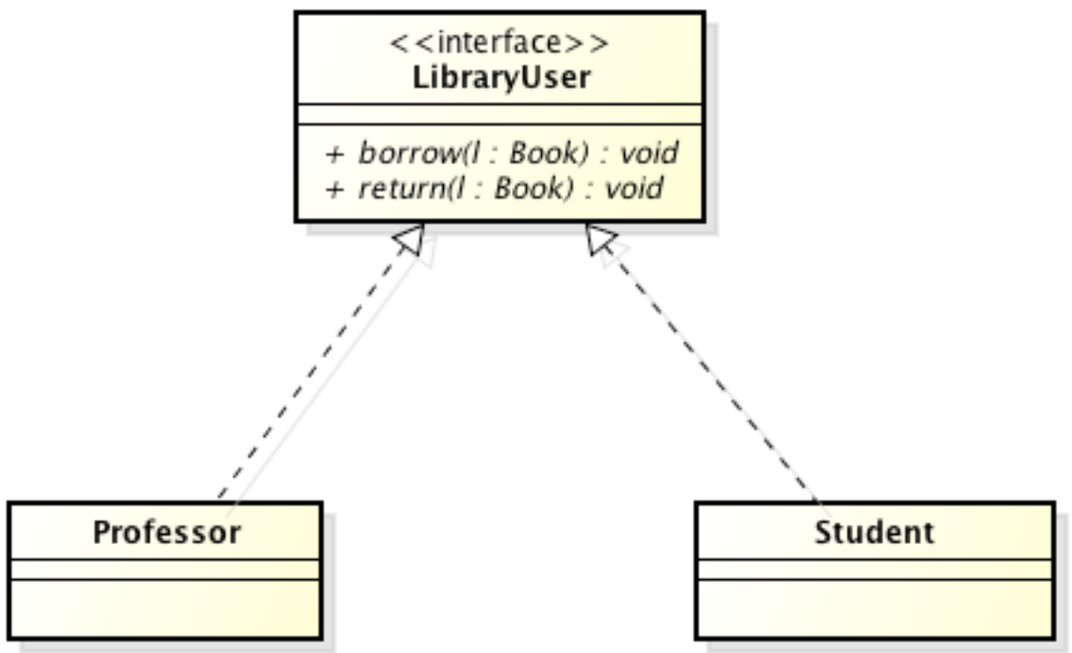
Propagation

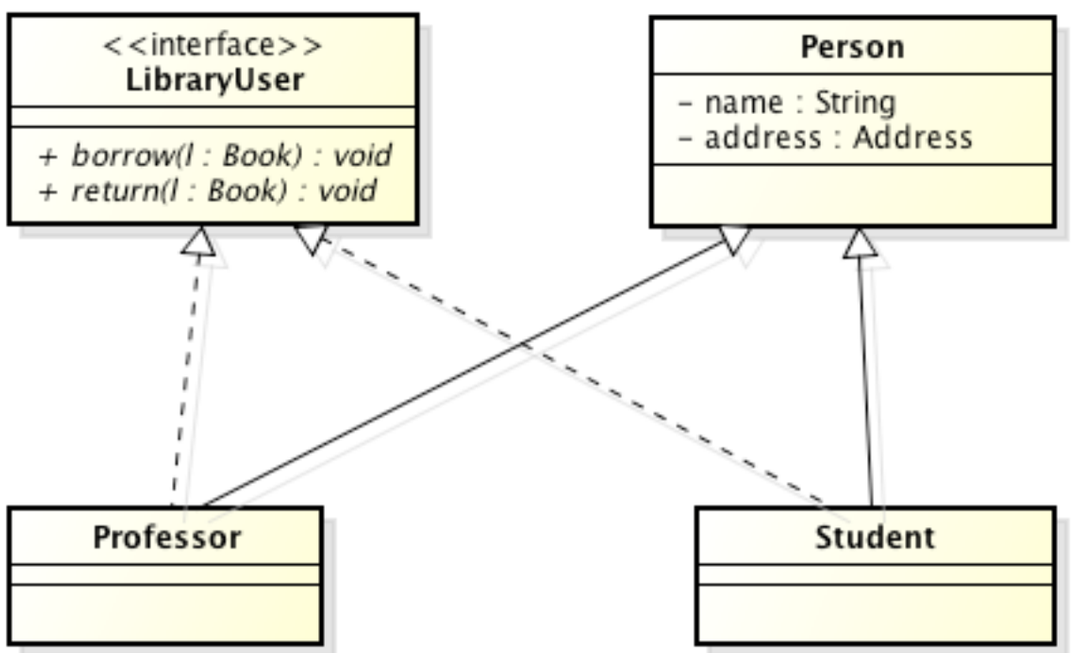
- La propagation est à l'agrégation ce que l'héritage est à la généralisation.
- La différence majeure est que:
  - ➡ *L'héritage* est un mécanisme *implicite*
  - ➡ La *propagation* doit être *programmée*

Les interfaces

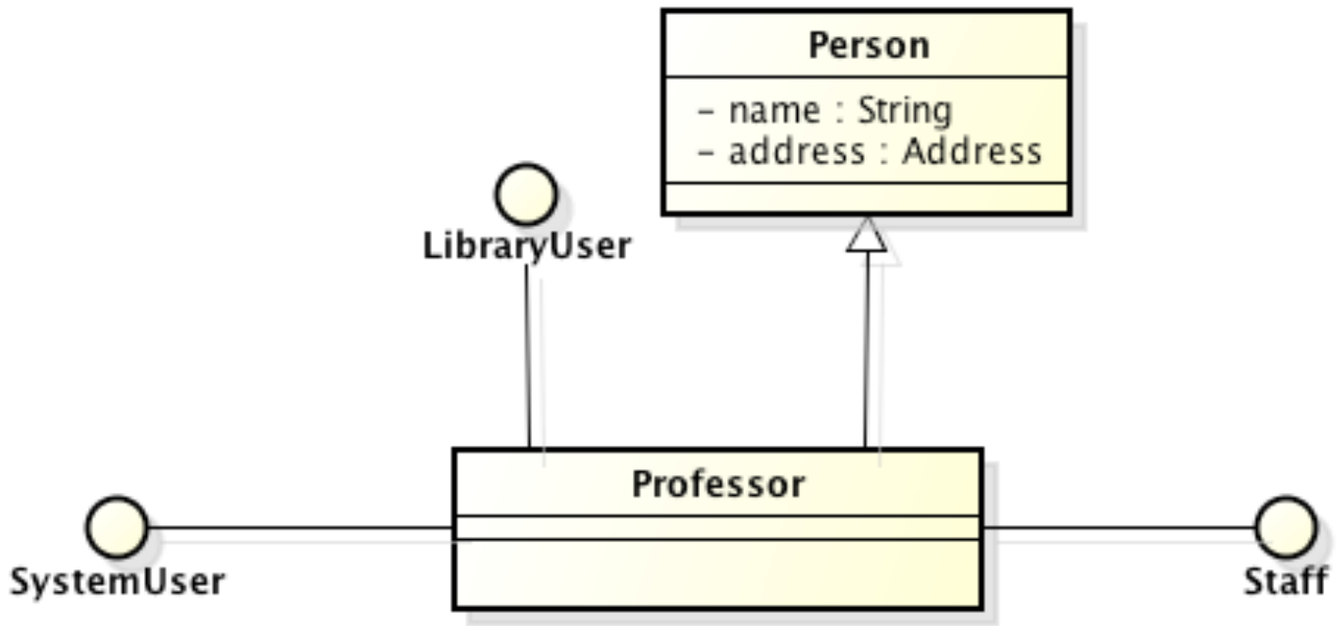
Une ***interface*** décrit une *portion du comportement visible* d'un ensemble d'objets.

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



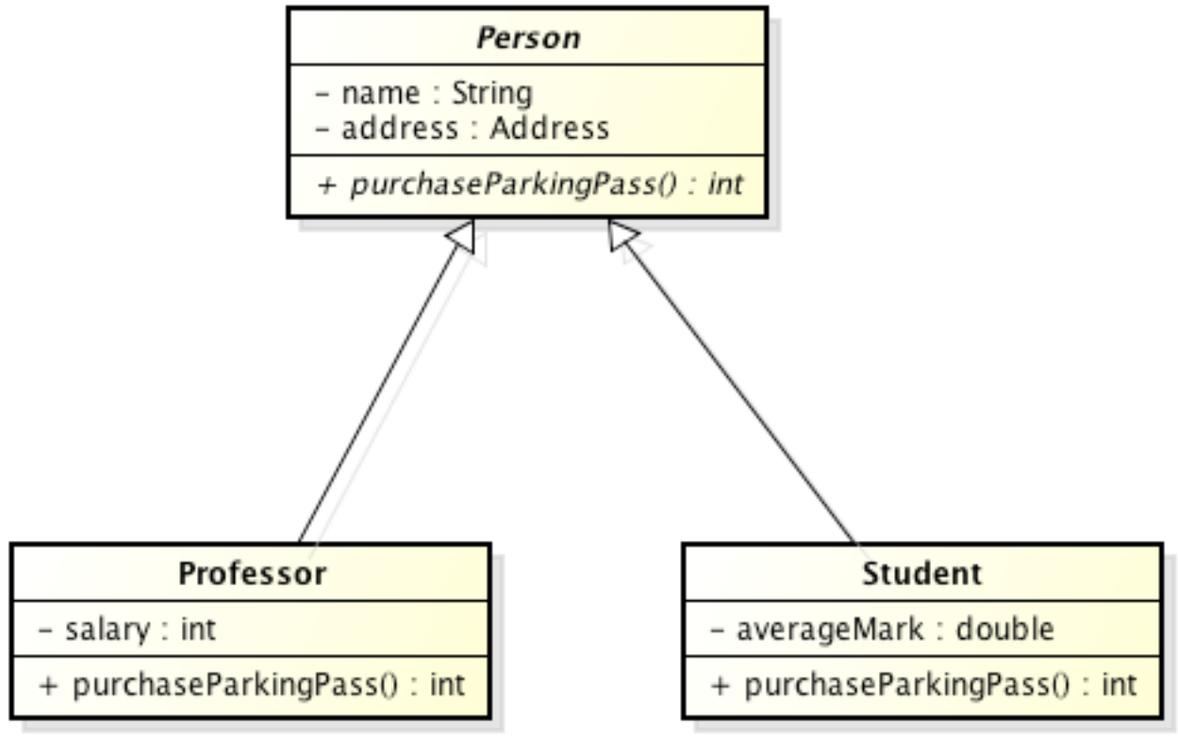


Socket notation

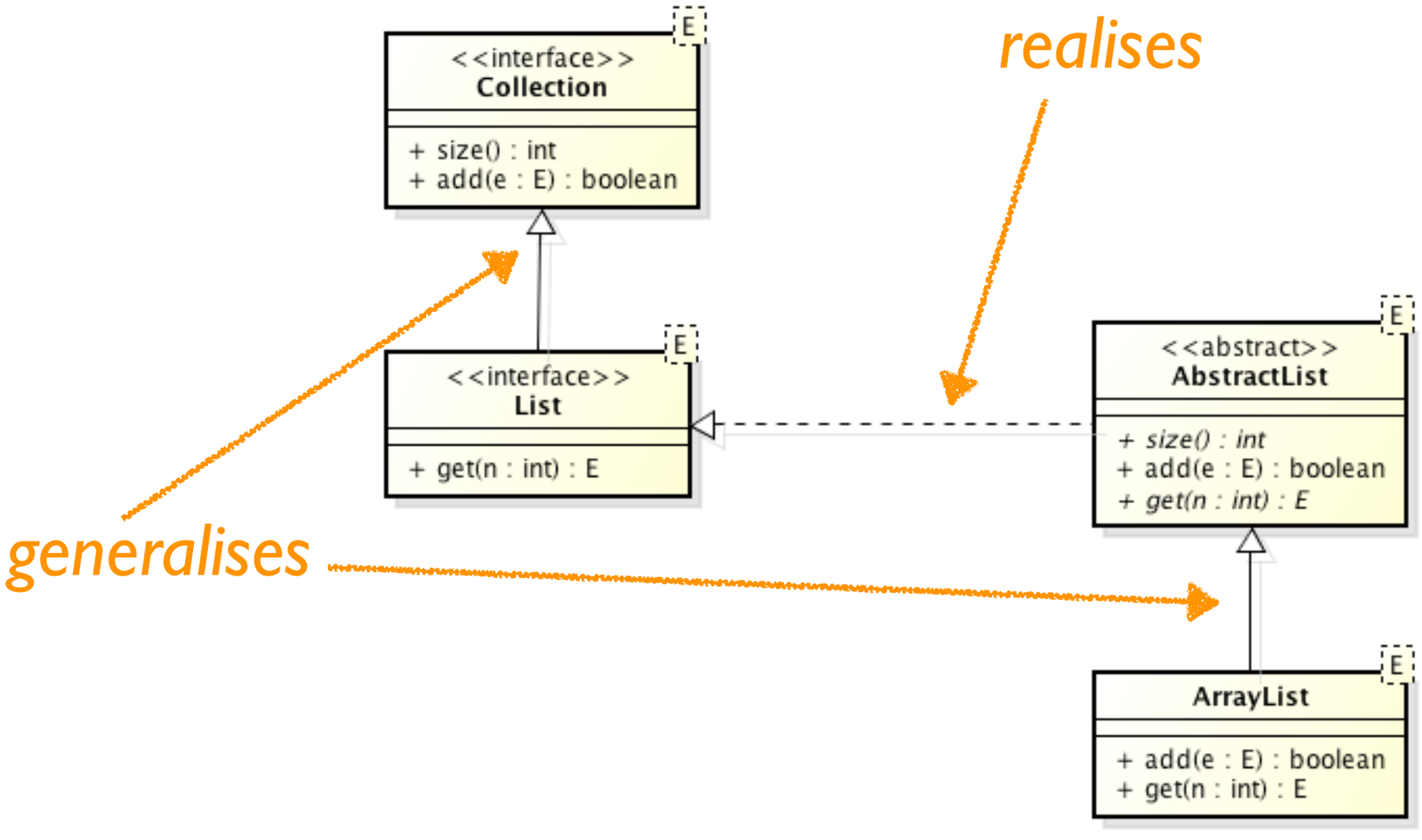


Classes abstraites

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

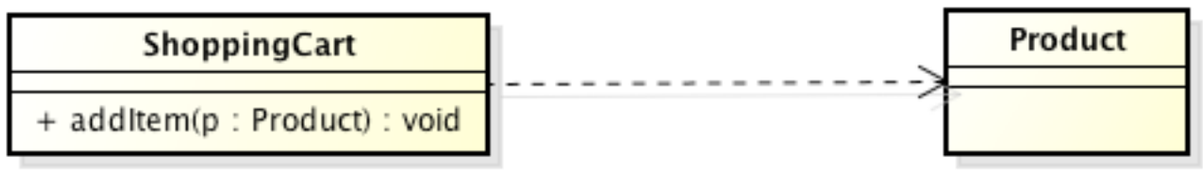


Interfaces et classes abstraites



Dépendances

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



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
- ***Eliminer*** **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'informations est toujours plus facile à maitriser*

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

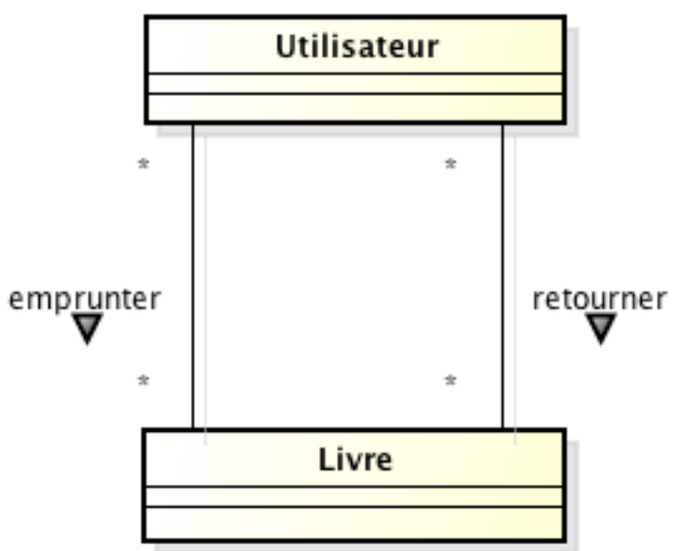
Identifier des associations

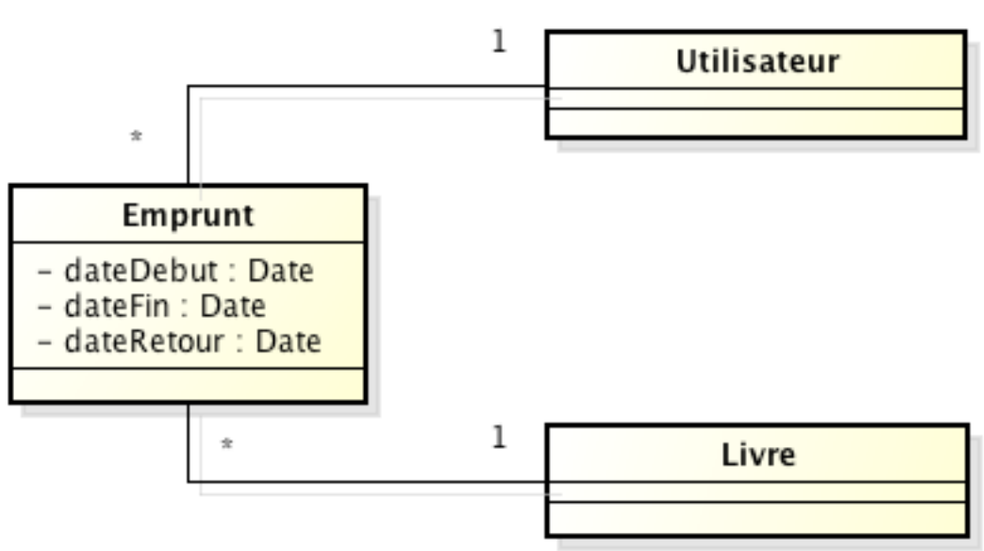
- Spécifier ensuite la multiplicité à chaque extrémité de l'association
- Étiqueter clairement cette association

Actions versus associations

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

Exemple: *des utilisateurs peuvent emprunter et retourner des livres en utilisant le SI de la bibliothèque.*



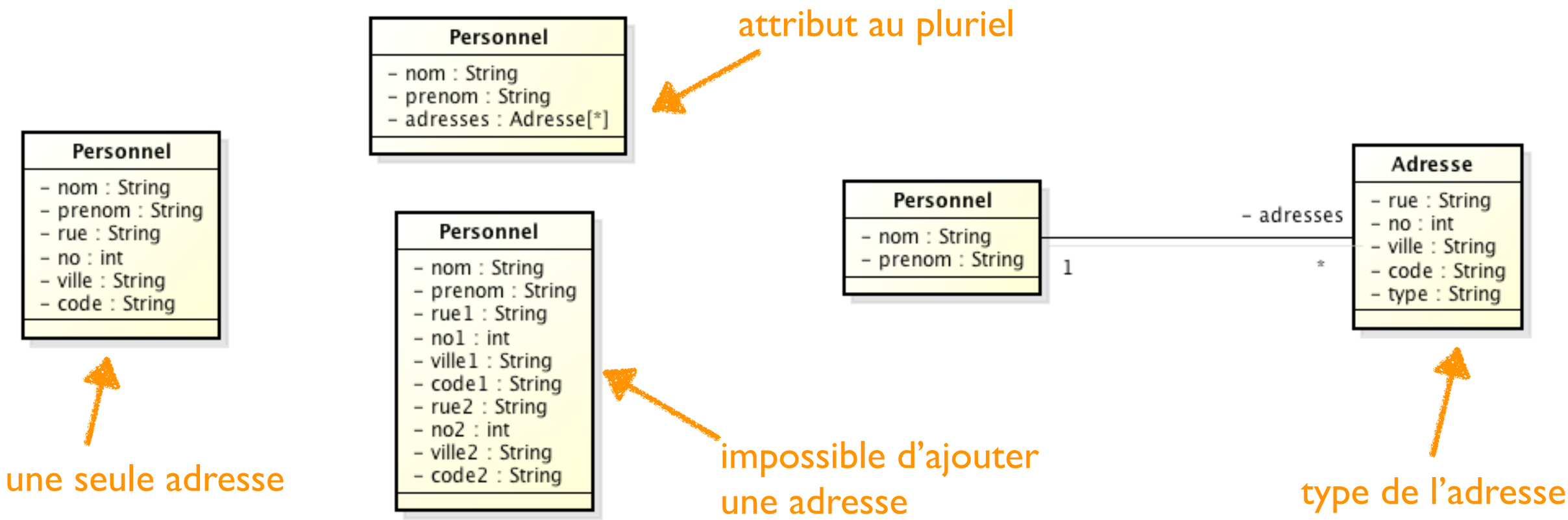


Identifier les attributs

- Déterminer l'information qui doit être maintenue dans chacune des classes
- Plusieurs noms ayant été rejetés comme classes peuvent devenir 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** »

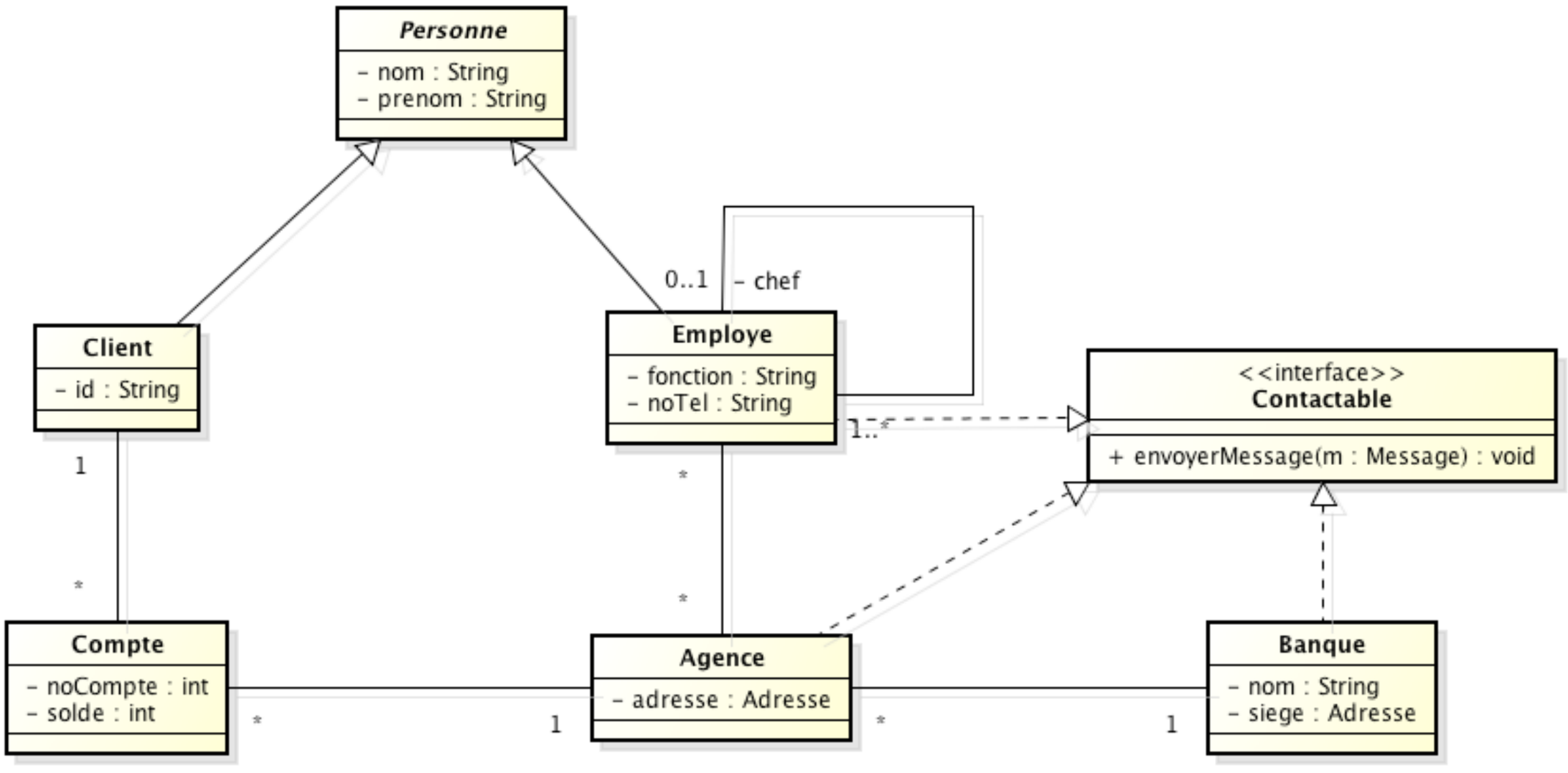
Identifier les interfaces et les généralisations

- 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


Personne
- nom : String
- prenom : String
0..1
- chef
Client
- id : String
Employe
- fonction : String
- noTel : String
<<interface>>
Contactable
+ envoyerMessage(m : Message) : void
1
*
1..*
*
*
Compte
- noCompte : int
- solde : int
Agence
- adresse : Adresse
Banque
- nom : String
- siege : Adresse
*
1
*
1

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

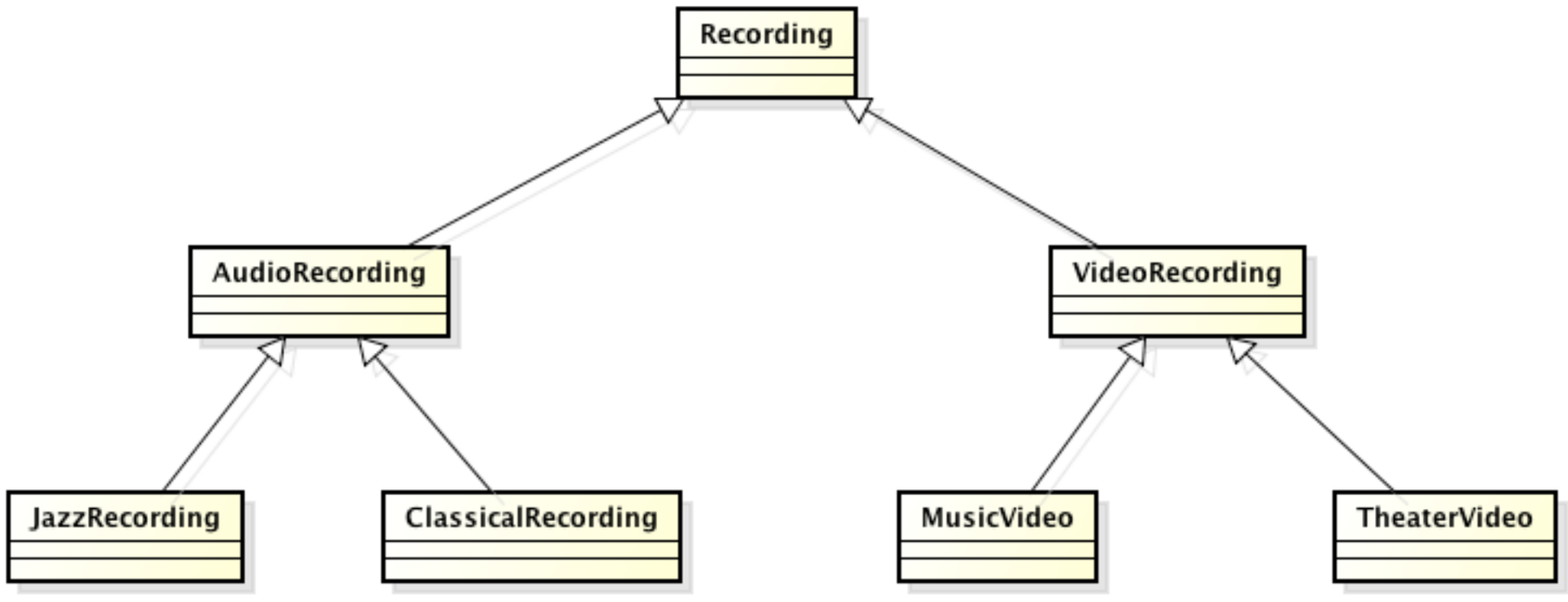
Une *généralisation* décrit une relation entre classes dans un diagramme de classes

- 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, ...

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

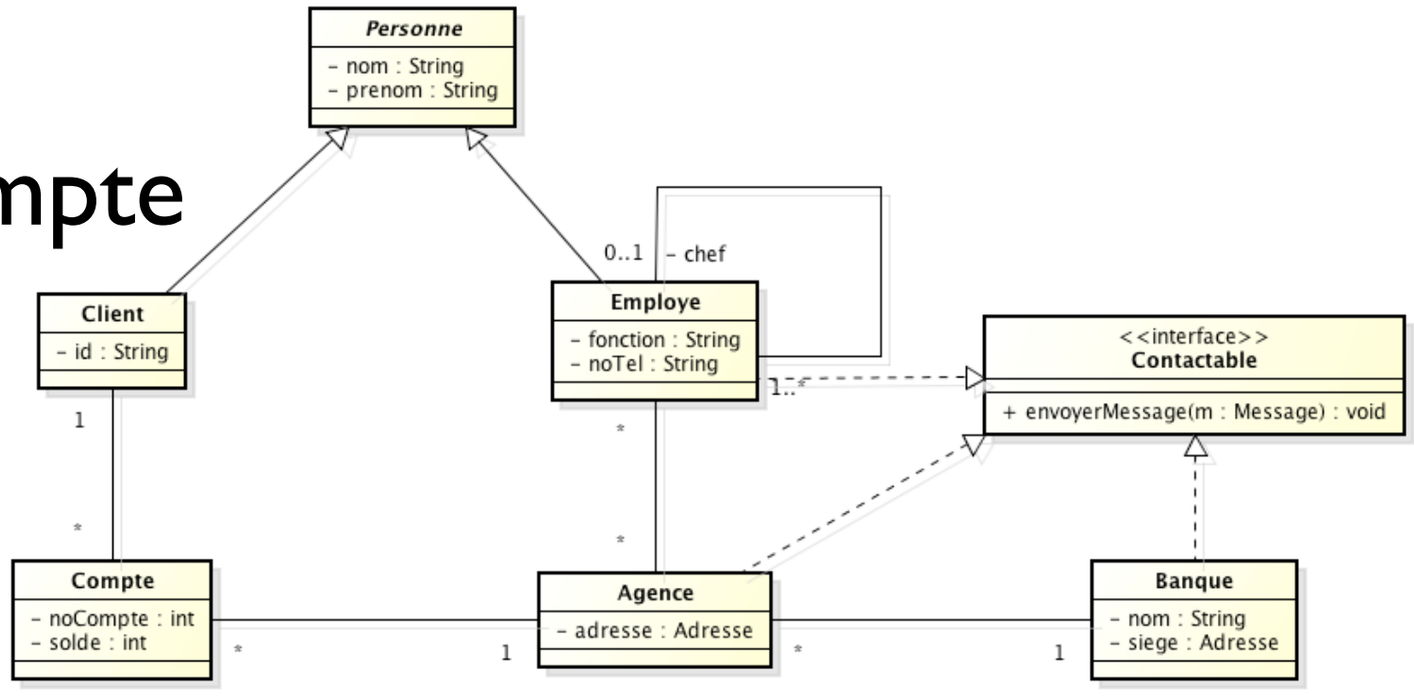
- Afin de déterminer les responsabilités
  - Effectuer une analyse de cas
  - Recherche des verbes et des noms décrivant des actions

Catégories de responsabilités

- Fixer et obtenir la valeur d'un attribut
- Créer et initialiser de nouvelle 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*