UML : petite introduction

Programmation Orientée Objet

Jean-Christophe Routier
Licence mention Informatique
Université des Sciences et Technologies de Lille

Modélisation

- Construire un bon logiciel :
  - Répondre aux objectifs fixés (satisfaire le client)
  - Avoir une base architecturale solide qui permette l'évolution
  - Mettre en place un processus de développement
    - ↳ maîtriser coûts et délais

- Définir des modèles pour
  - Spécifier la structure et les comportements attendus du système
  - Visualiser et contrôler l'architecture pour mieux la comprendre
    - ↳ simplifier, réutiliser, gérer les risques

De la niche à l'immeuble

- Construire une niche
  - Planches, un marteau, des clous, une personne, quelques heures.
  - Le client (le chien) sera probablement

- Construire une maison
  - Matériaux et outils plus « complexes »
  - Dessiner des plans,
  - Qualité $\Rightarrow$ tenir compte des contraintes (utilisation, besoins en éclairage, chauffage, eau, normes, …)
  - Une seule personne ?
  - Délais ? Coûts ?

- Construire un immeuble :
  - Prendre l'avis des investisseurs (style, forme, taille , etc.) (y compris les modifications)
  - Plannings temps et budget primordiaux
  - De nombreuses personnes réparties en équipes
  - Concevoir de nombreux plans et modèles
  - Le plus souvent, faire une maquette
  - Coordonner les différentes équipes, faciliter la communication entre elles
  - etc.

Pour une voiture, démarche similaire...

Système logiciel = Immeuble

- Ecrire beaucoup de lignes de code, même très propres, ne suffit pas

- Nécessité de penser au préalable l'architecture logicielle du système

Construction d'un modèle indispensable

Modèle

- Qu'est-ce que c'est ?
  « *Une simplification de la réalité* »
- Pourquoi ?
  « *Mieux comprendre le système à développer* »
  Servir d'interface entre les acteurs du projet
- Doit être proche de la réalité
- 4 objectifs :
  - Aider à visualiser un système tel qu'il est ou doit être.
  - Préciser la structure ou le comportement d'un système.
  - Fournir un canevas pour la construction du système.
  - Permettre de documenter les décisions prises.
- D'autant plus nécessaire que le système est complexe

Modélisation orientée objet

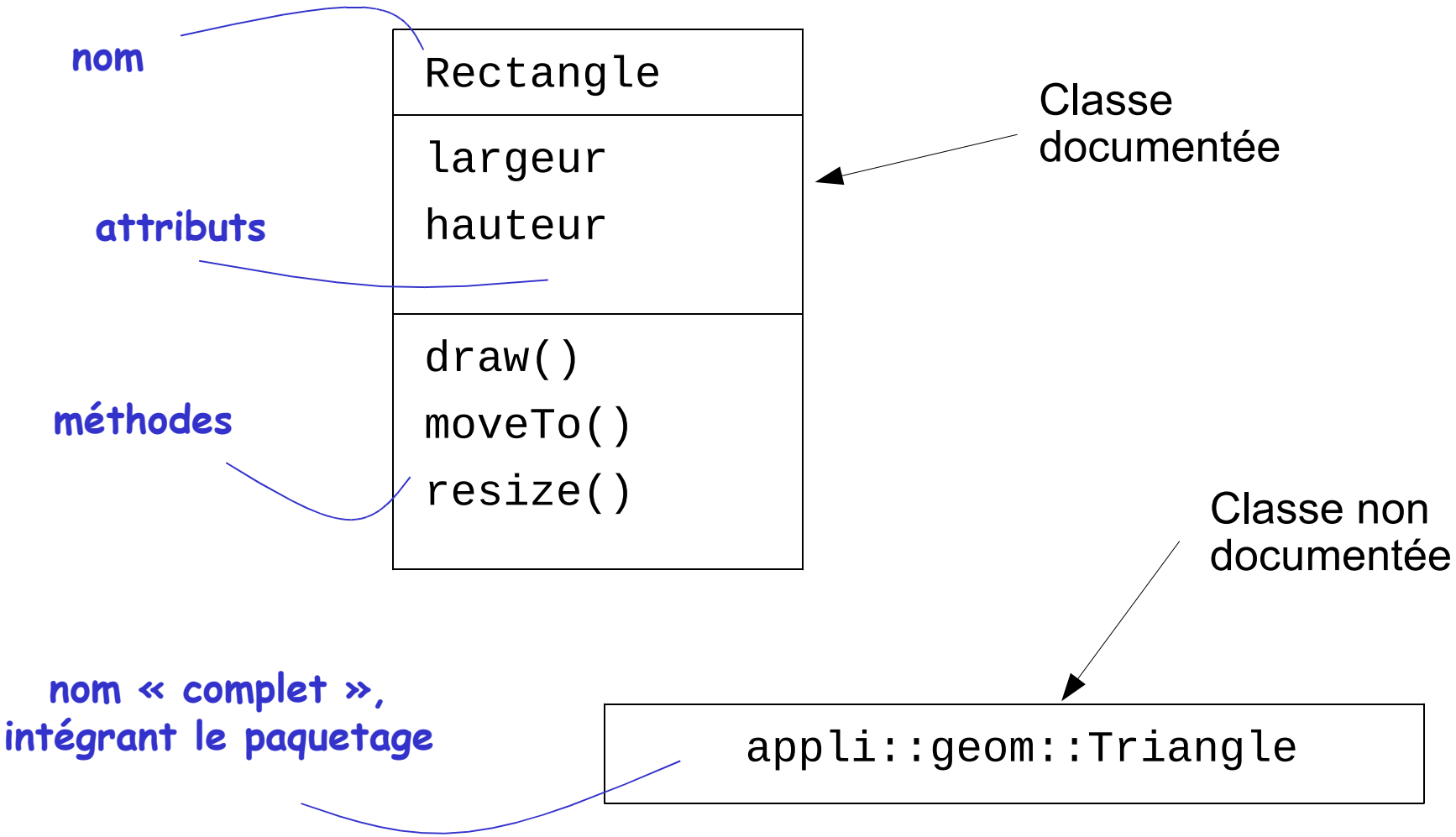
- UML : *Unified Modeling Language*
  - langage conçu pour l'écriture de plans d'élaboration de logiciels (pas une méthode)
  - né de la fusion de plusieurs méthodes objet, standard de fait
  - utilisable pour *visualiser*, *spécifier*, *construire* et *documenter*
- un méta-modèle :
  - Langage sans ambiguïtés
  - Peut servir de support pour tout langage objet
  - Notation graphique simple compréhensible par des non informaticiens et facilitant la communication

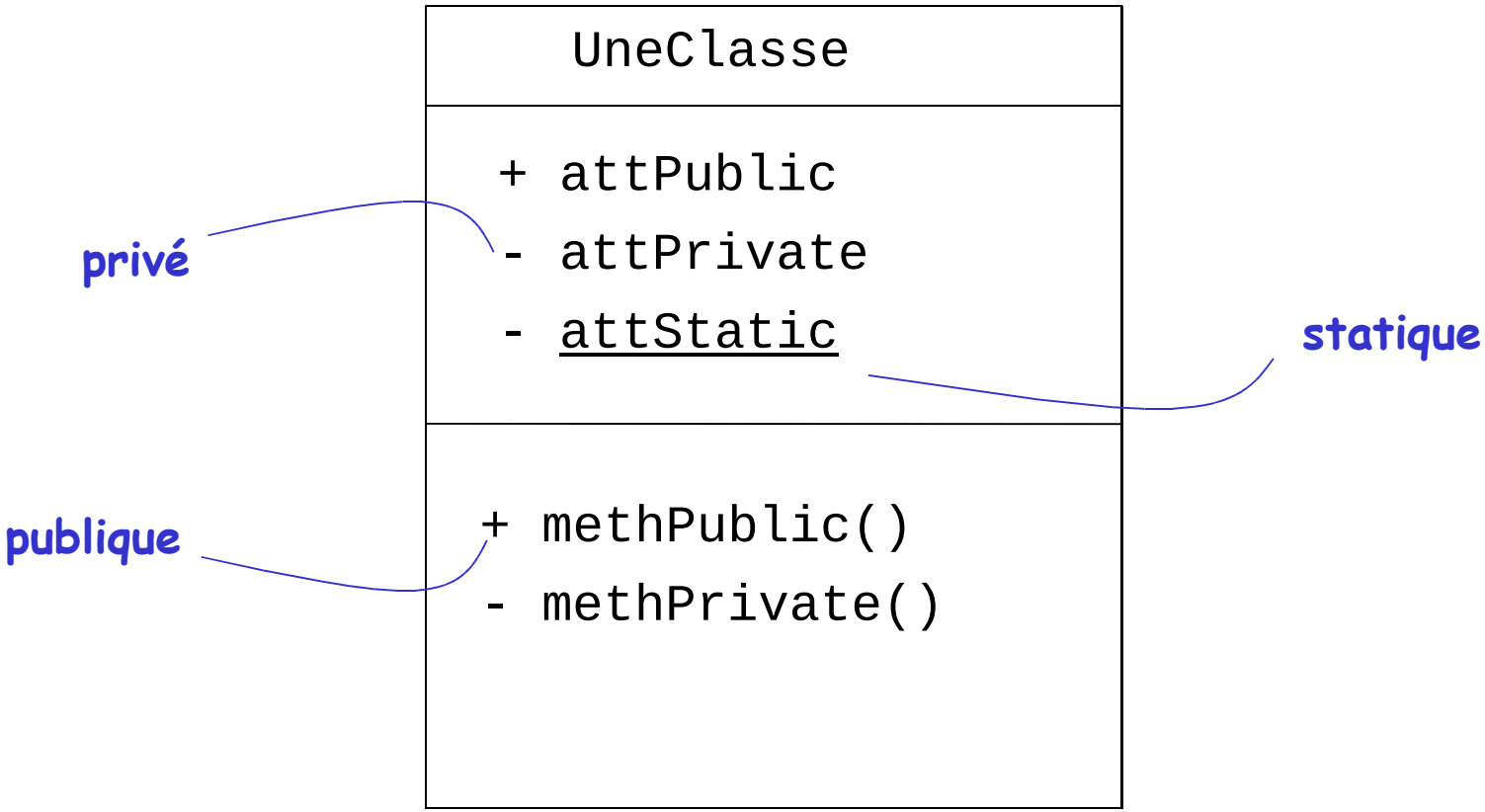
Statique et Dynamique

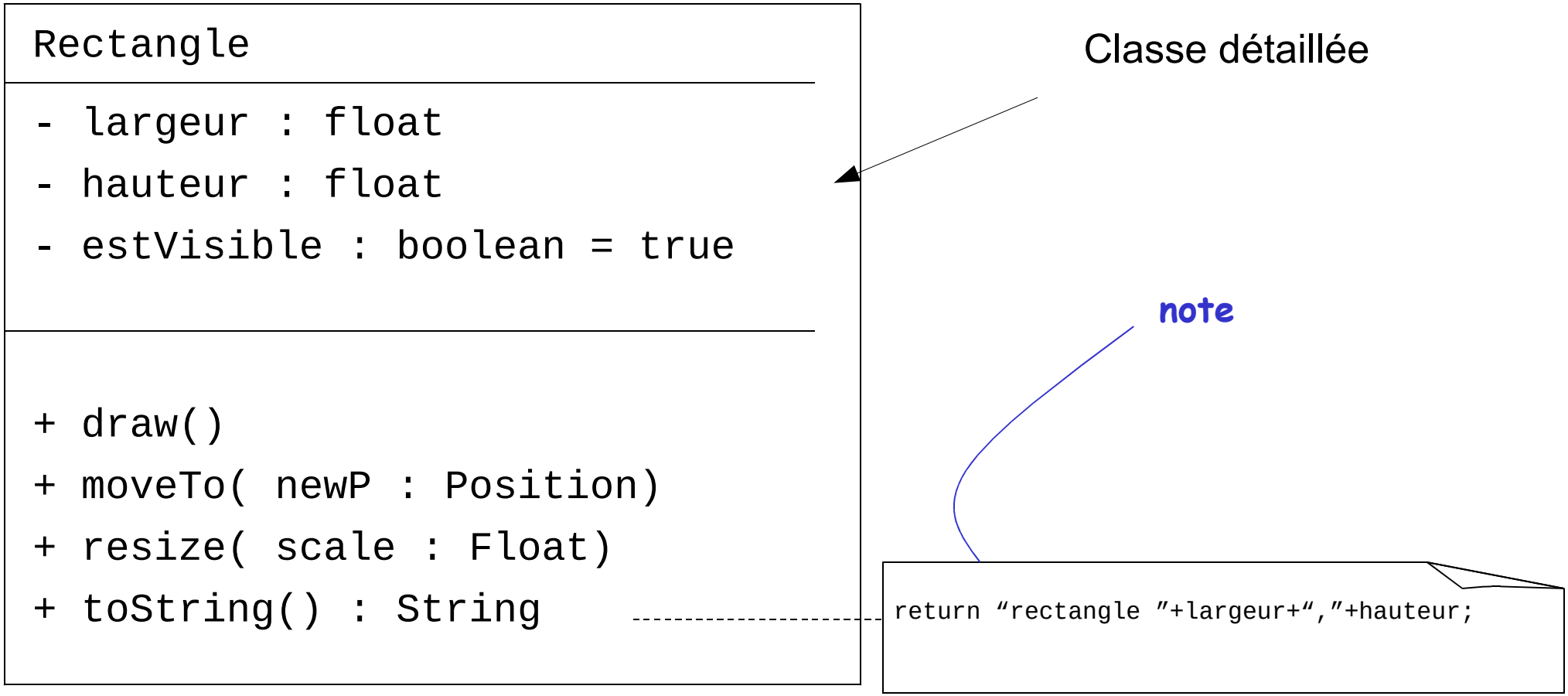
- Vues statiques
  - diagrammes d'objets,
  - diagrammes de classes,
  - diagrammes des cas d'utilisation,
  - diagrammes de déploiement.

- Vues dynamiques
  - diagrammes de séquences,
  - diagrammes de collaboration,
  - diagrammes d'états-transitions,
  - diagrammes d'activités.

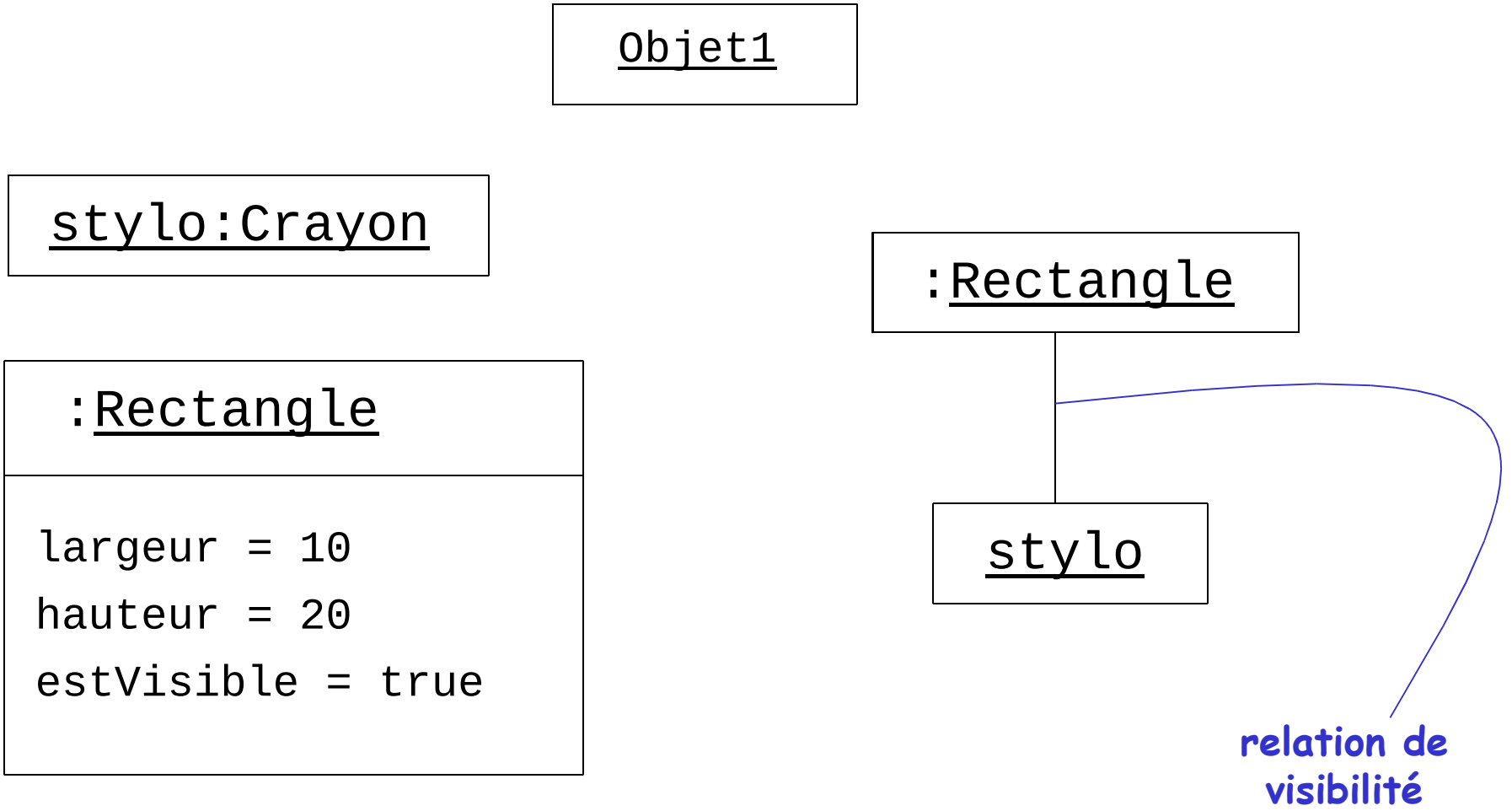
Classes


nom
Rectangle
largeur
hauteur
attributs
draw()
moveTo()
resize()
méthodes
Classe documentée
Classe non documentée
nom « complet », intégrant le paquetage
appli::geom::Triangle


UneClasse
+ attPublic
- attPrivate
- attStatic
+ methPublic()
- methPrivate()
privé
statique
publique


Rectangle
- largeur : float
- hauteur : float
- estVisible : boolean = true
+ draw()
+ moveTo(newP : Position)
+ resize(scale : Float)
+ toString() : String
Classe détaillée
note
return “rectangle ”+largeur+“,”+hauteur;

Objets (instances)



Relations

Identifier les classes ne suffit pas, elles coopèrent/ interagissent entre elles, il faut exprimer ces relations (le plus souvent binaires).

- Dépendances
  - relations d'utilisation
- Associations
  - relations structurelles, connexion sémantique
- Agrégation, composition

- Généralisations : interfaces
  - + relations d'héritage (relation « is a », cf. UE COO du S6)

Association

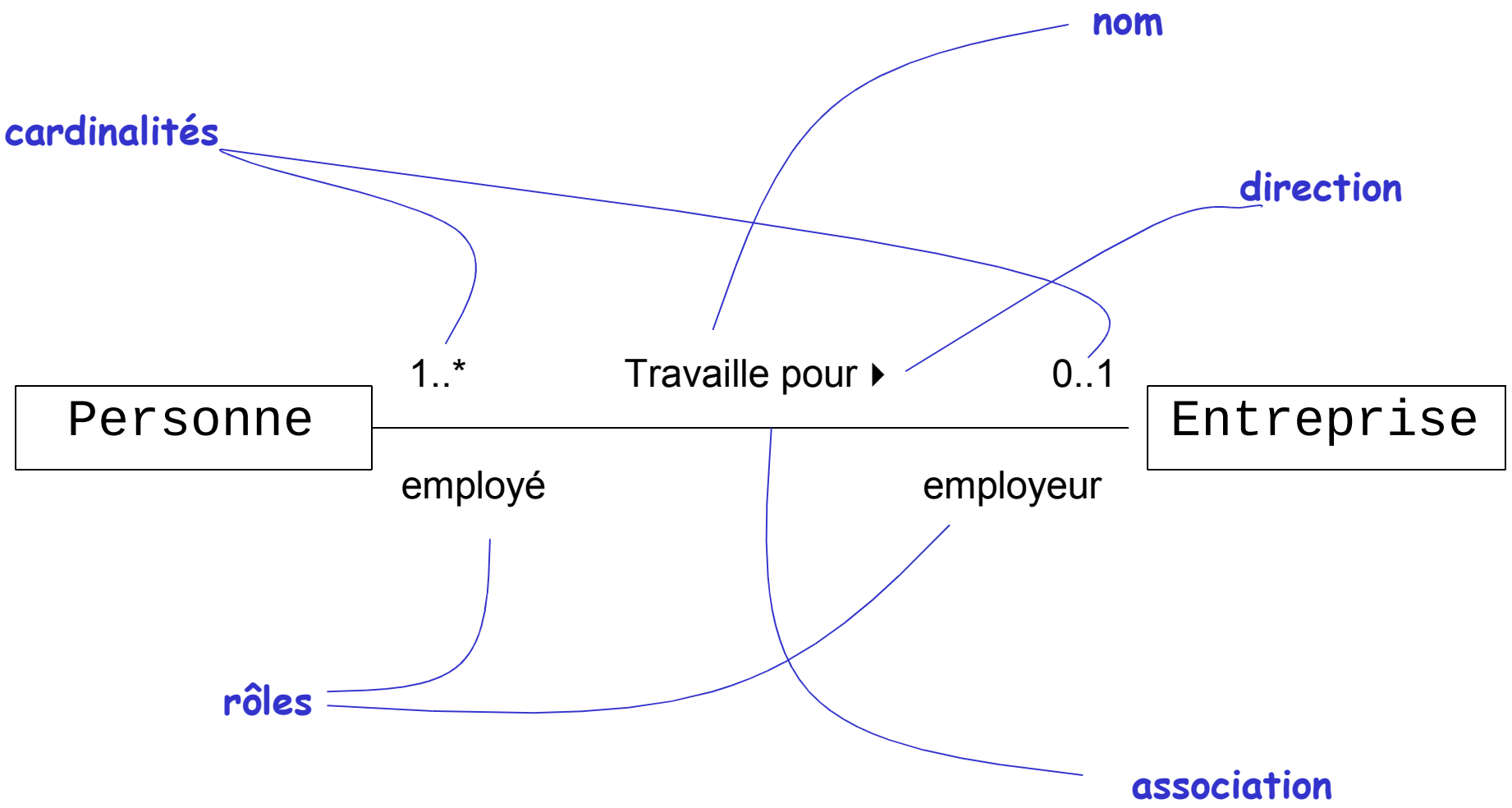
C’est une relation structurelle qui exprime une relation sémantique entre (le plus souvent) deux classes.

Elle est le plus souvent réflexive.

On peut la compléter de 4 informations :

- Nom
- Rôles
- Multiplicité
- Agrégation

Exemple


nom
cardinalités
direction
1..*
Travaille pour
0..1
Personne
Entreprise
employé
employeur
rôles
association

Cardinalités

- Définissent la multiplicité des rôles

- Une cardinalité à une extrémité signifie qu'à l'autre extrémité ce nombre d'éléments doit exister pour tout objet de la classe.

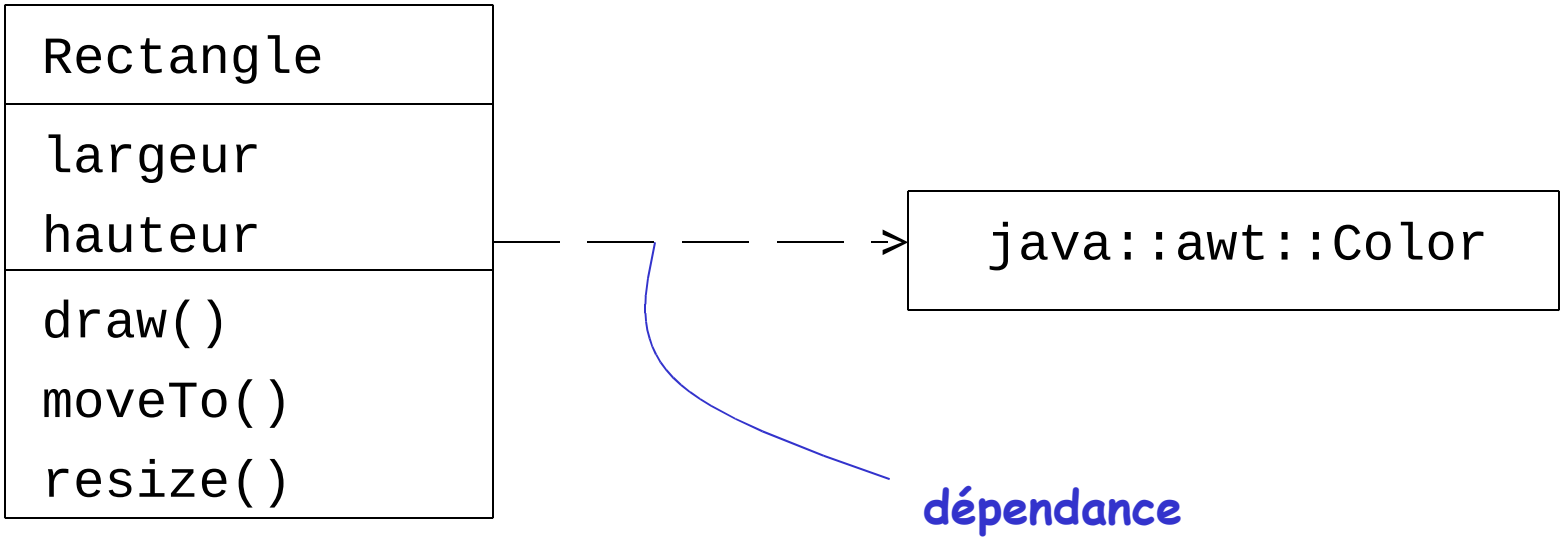
- Expressions possibles :
  - n : exactement *n*
  - n..m : de *n* à *m*
  - * : quelconque (équivalent à « 0..n » ou « 0.. »)
  - n..* : *n* ou plus
  - liste de cardinalités : 1..2,3..5 = 1 à 5 sauf 4

Il peut exister plusieurs relations entre les mêmes classes

Dépendance

Exprimer le fait qu'une classe en utilise une autre.

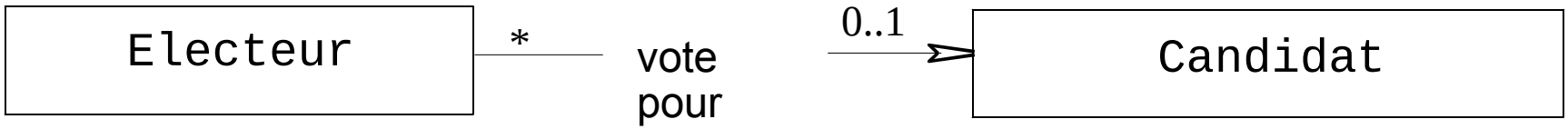
Toute modification de la classe utilisée risque d'avoir un impact sur la classe utilisatrice. Relation d'obsolescence.



Navigabilité restreinte

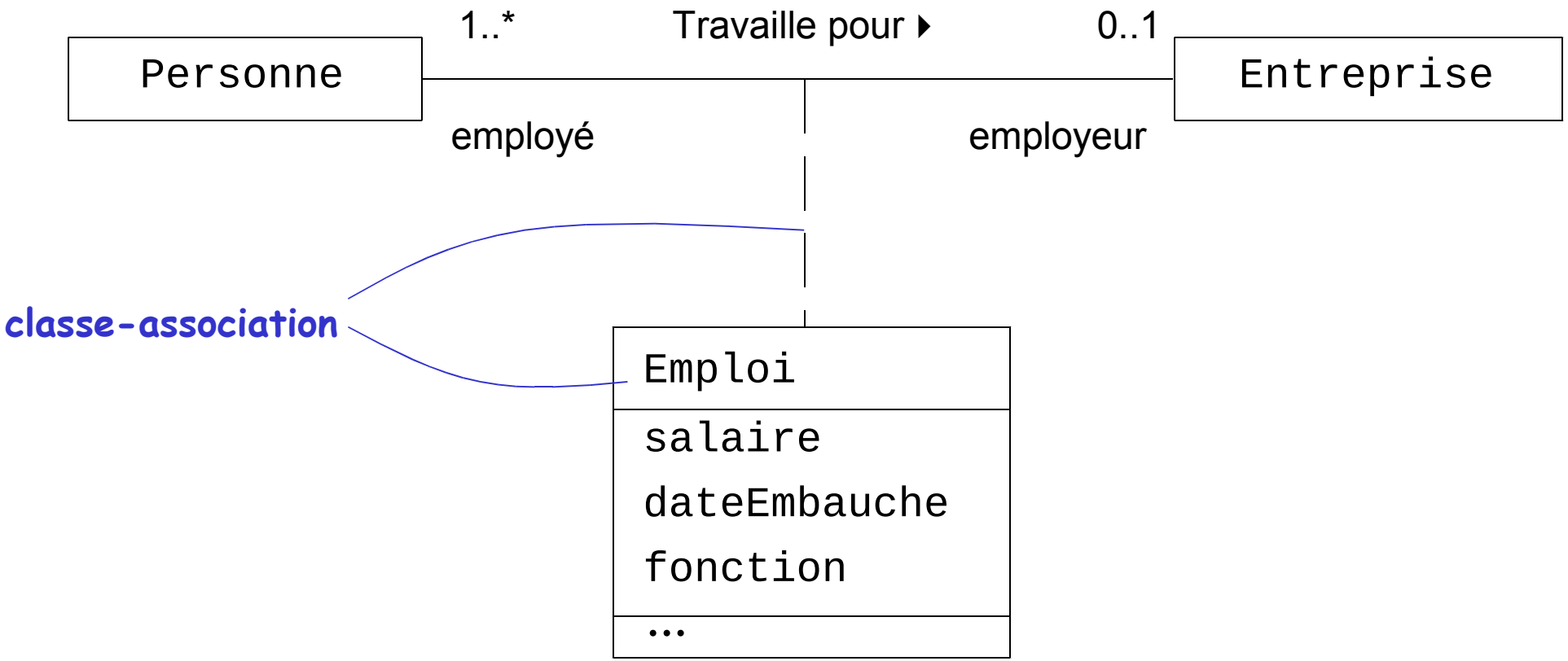
Rendre unidirectionnelle la relation

Pour indiquer que les instances d'une classe ne "connaissent" pas les instances d'une autre.

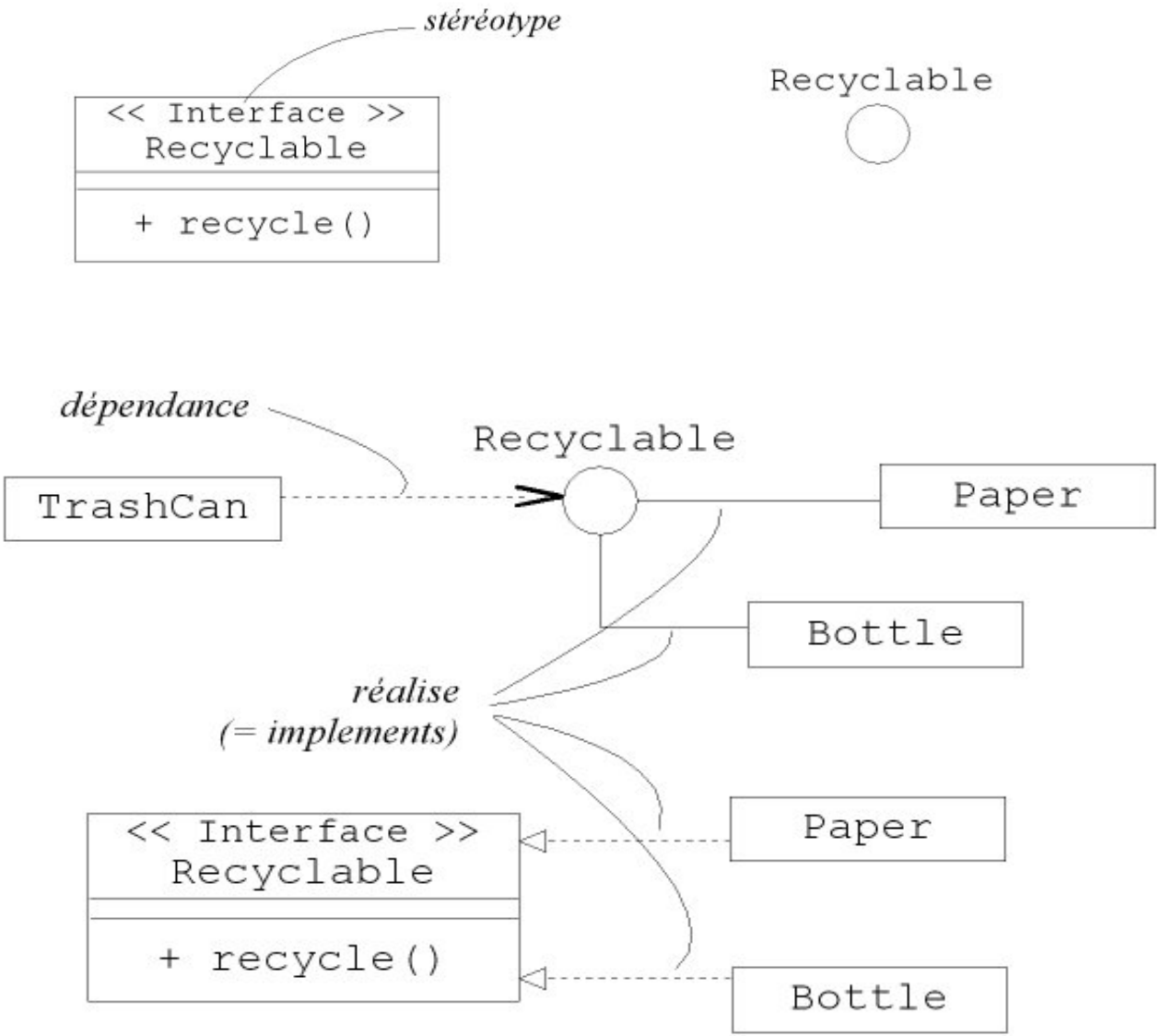


Classe-association

quand on veut exprimer les propriétés d'une relation



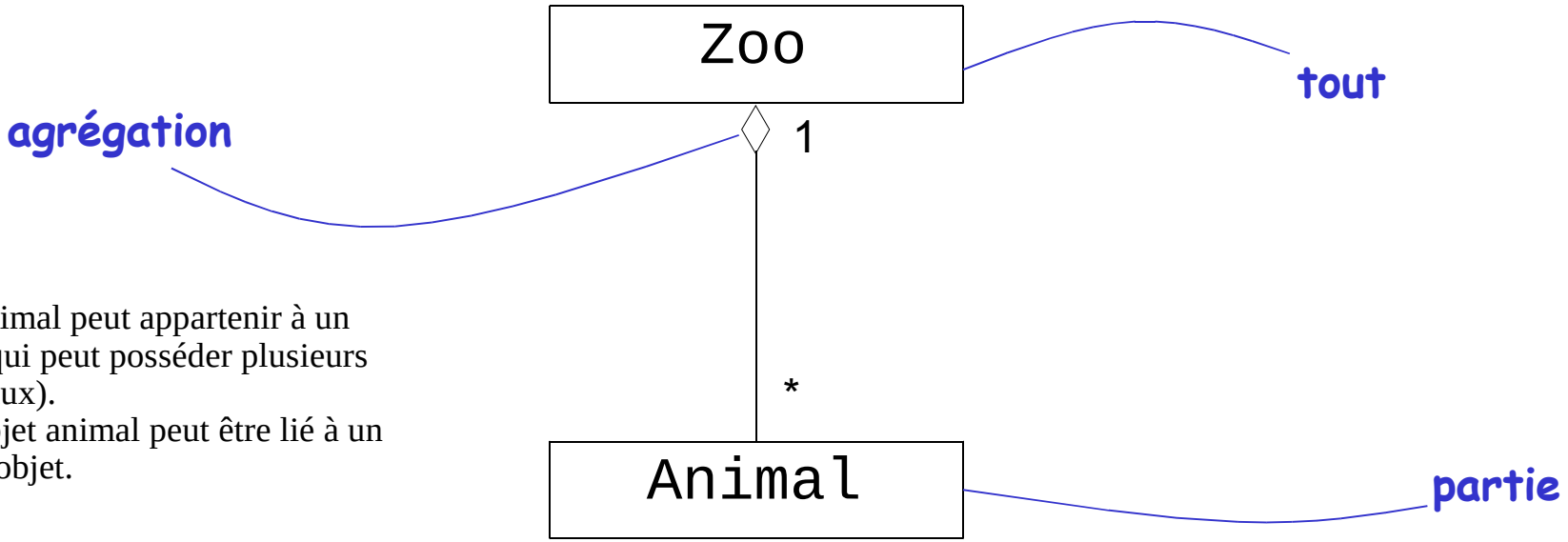
Interfaces


stéréotype
<< Interface >>
Recyclable
+ recycle()
Recyclable
dépendance
Recyclable
TrashCan
Paper
Bottle
réalise
(= implements)
<< Interface >>
Recyclable
+ recycle()
Paper
Bottle

Agrégation/Composition

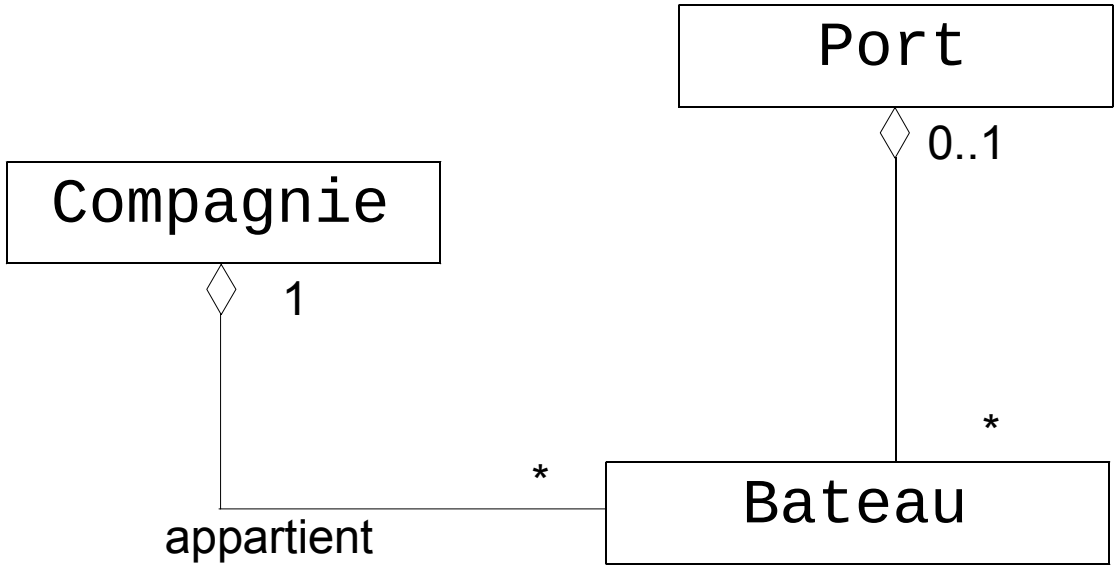
Association « tout/partie », relation de possession « has-a »

Agrégation : le tout est responsable de la gestion de ses parties.
Relation de subordination.

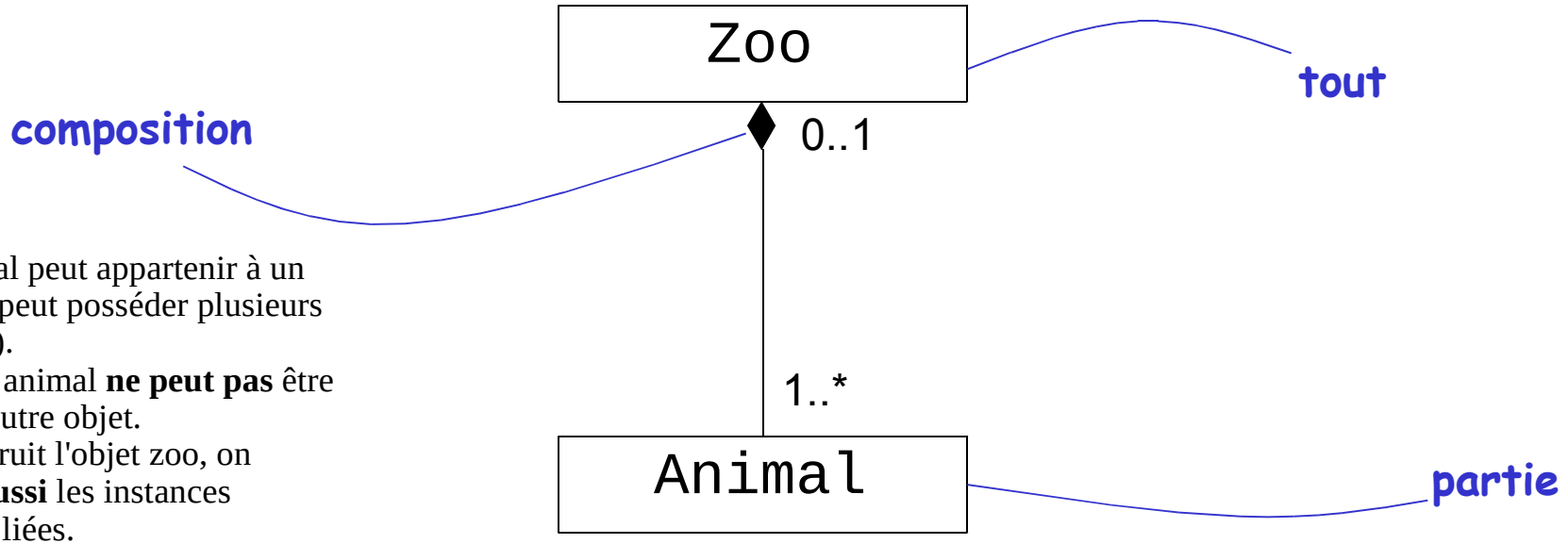


Un animal peut appartenir à un zoo (qui peut posséder plusieurs animaux).
Un objet animal peut être lié à un autre objet.

la partie est éventuellement partagée

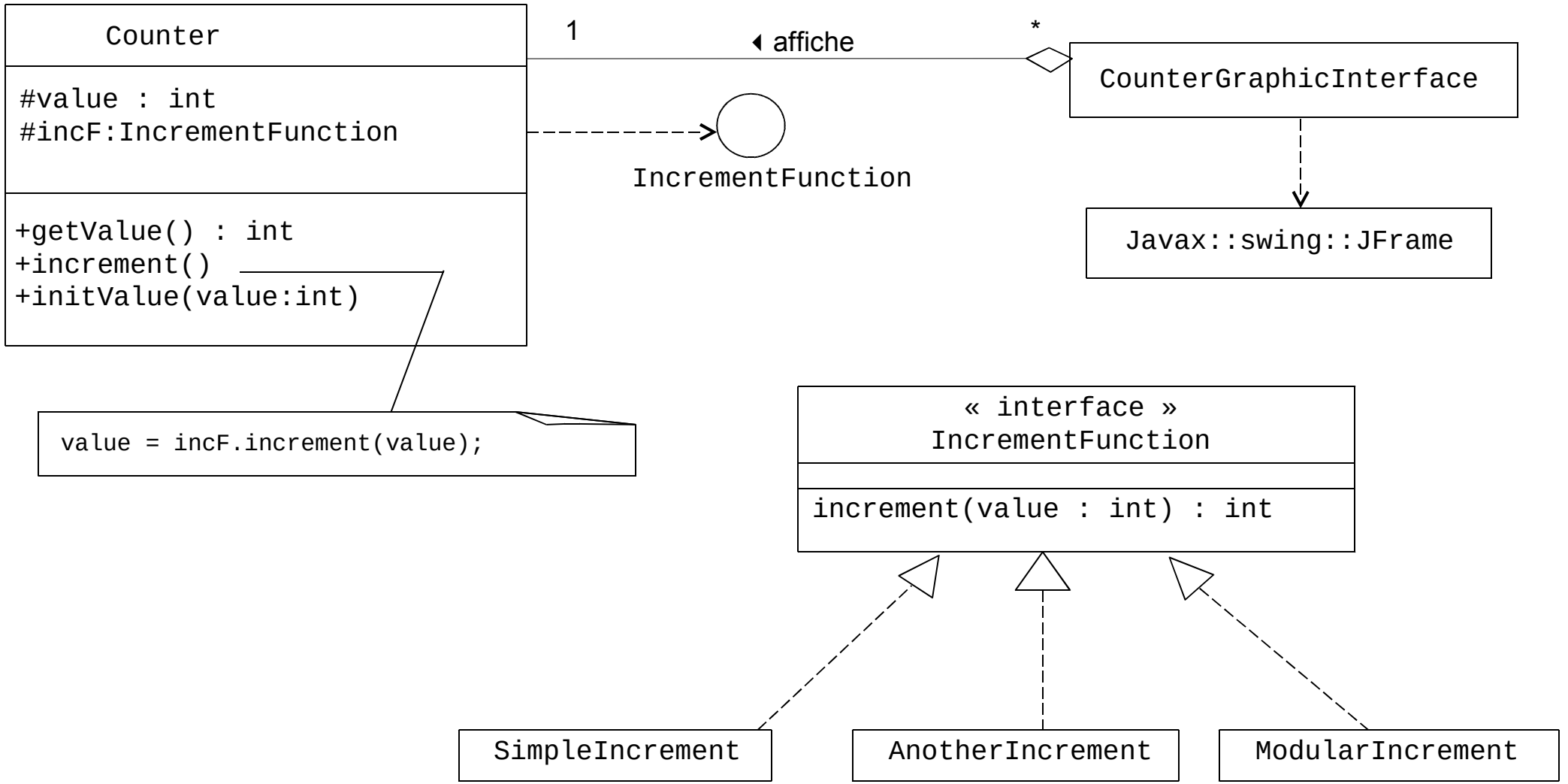


Composition : agrégation **forte,** la partie n'est pas partagée



Un animal peut appartenir à un zoo (qui peut posséder plusieurs animaux).
Un objet animal **ne peut pas** être lié à un autre objet.
Si on détruit l'objet zoo, on détruit **aussi** les instances animaux liées.

Exemple : compteur



Cas d'utilisation (use cases)

- représenter le *modèle conceptuel*
- identifier les *acteurs* et leurs interactions avec le système

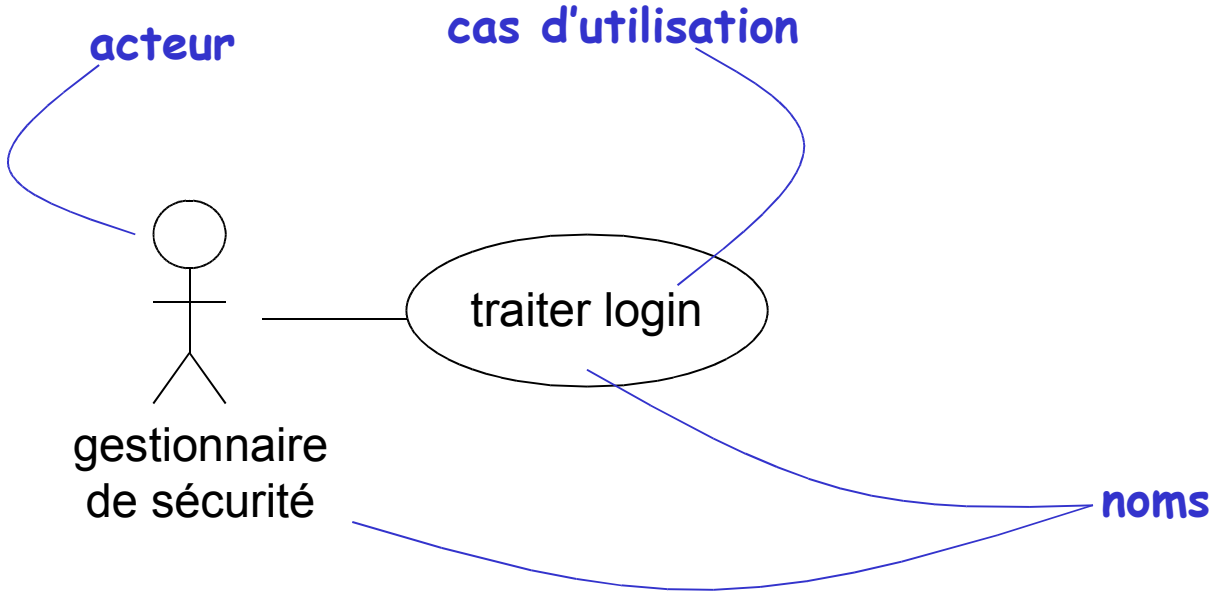
- permettre de structurer les **besoins des utilisateurs** et les objectifs du système

- permettre de définir le comportement attendu du système, sans en préciser la réalisation
  - on ne s'occupe que des objectifs, pas des solutions d'implémentation
  - clarification et organisation des besoins

Représentation

- acteur : *une personne ou un composant à l'origine d'une interaction avec le système*

- cas d'utilisation : *un objectif du système, un besoin d'un acteur*



Inclusion

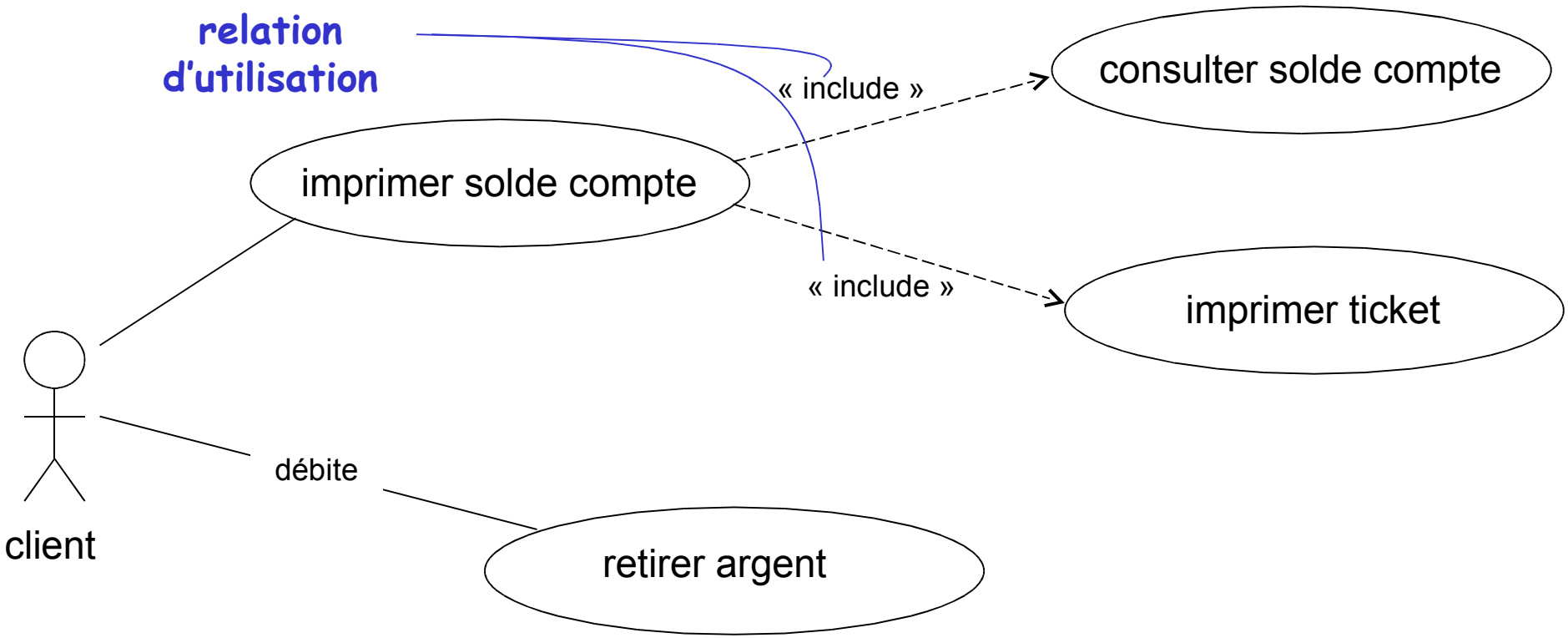
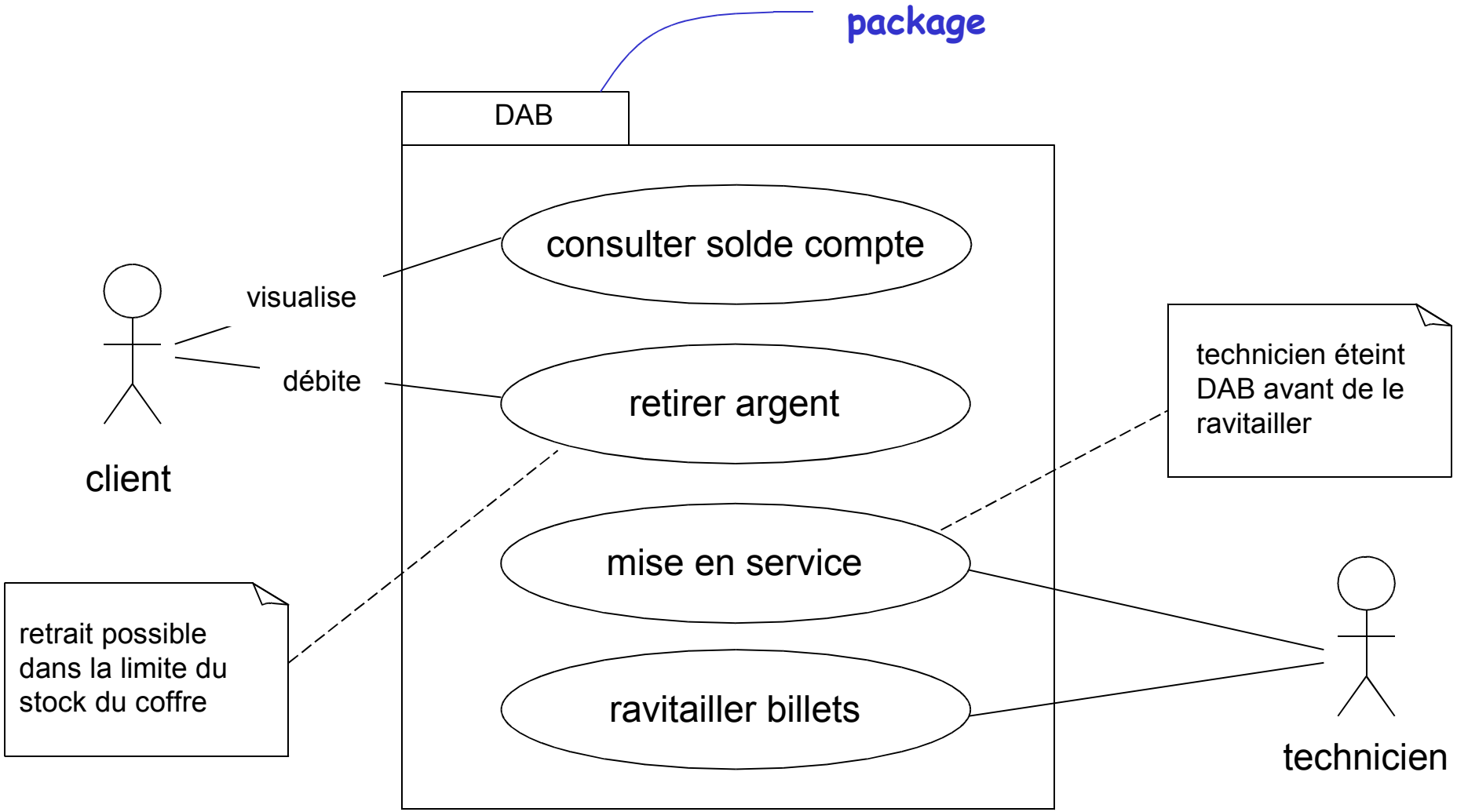
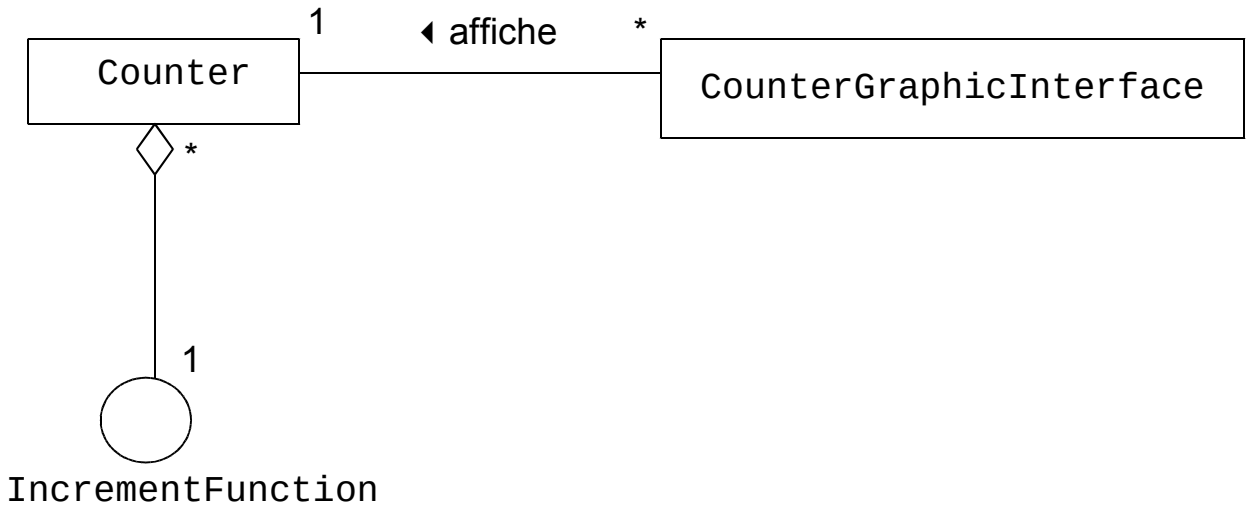
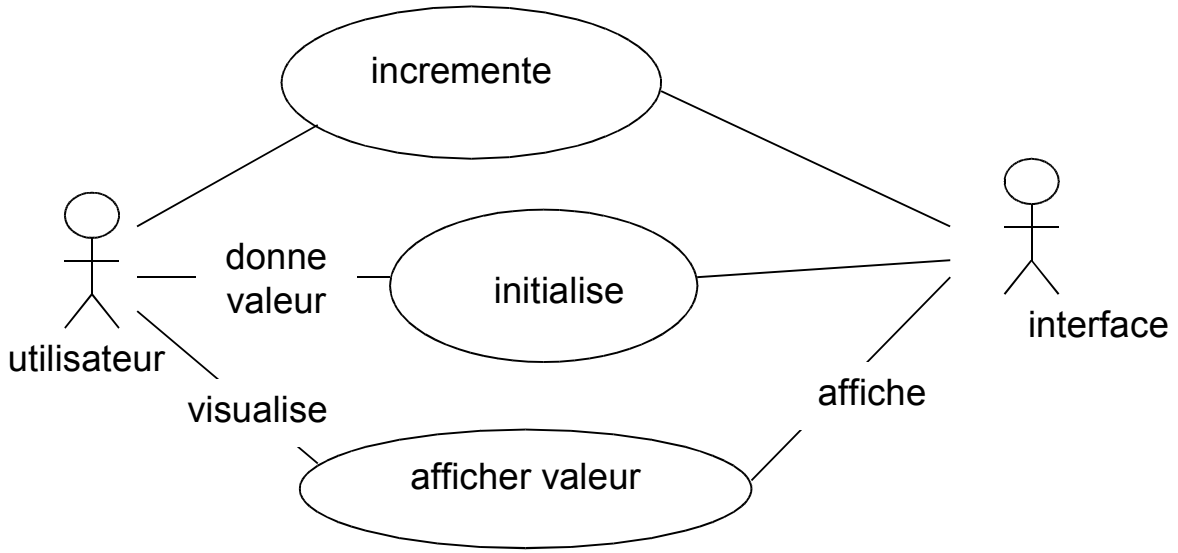

relation
d'utilisation
« include »
consulter solde compte
imprimer solde compte
« include »
imprimer ticket
client
débite
retirer argent

Diagramme de cas d'utilisation




1
affiche
*
Counter
CounterGraphicInterface
*
1
IncrementFunction


incremente
donne
valeur
initialise
utilisateur
interface
visualise
affiche
afficher valeur