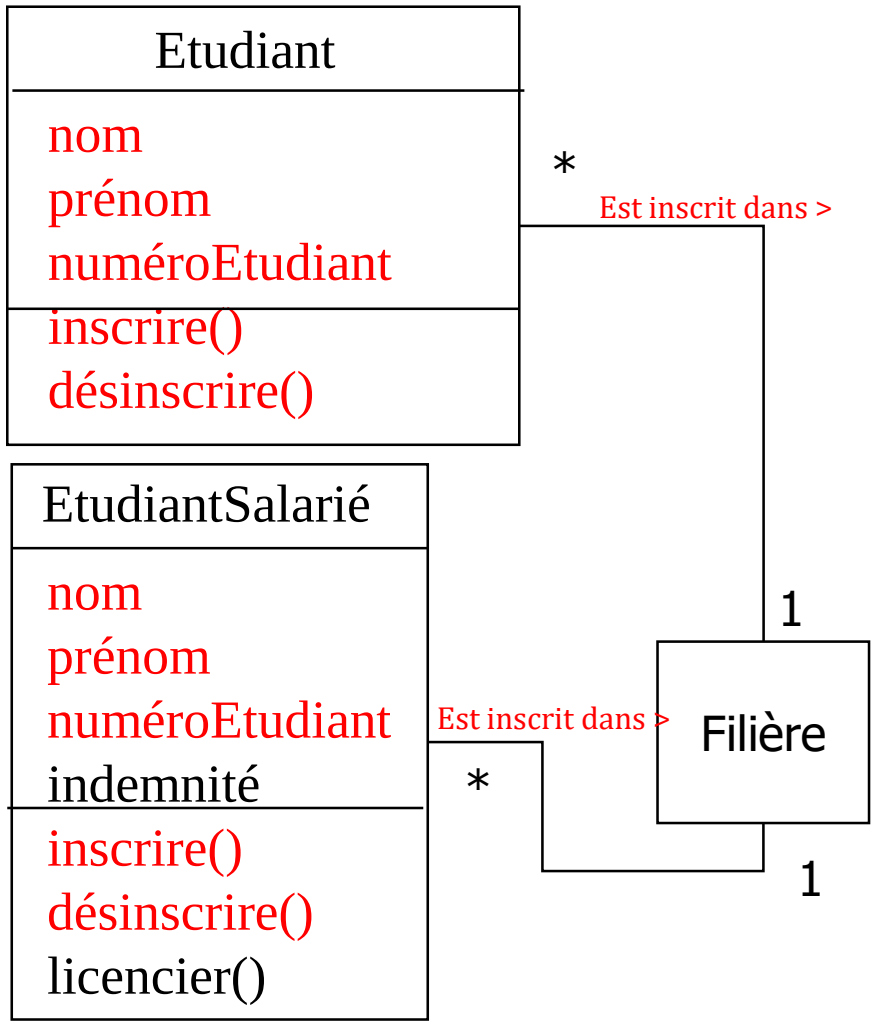
BUT Informatique
1A - Semestre 1
Introduction aux bases de données
(R1.05)

R. Fleurquin

Chapitre 6

Diagramme de classes : la relation de généralisation

Un petit exemple très parlant…



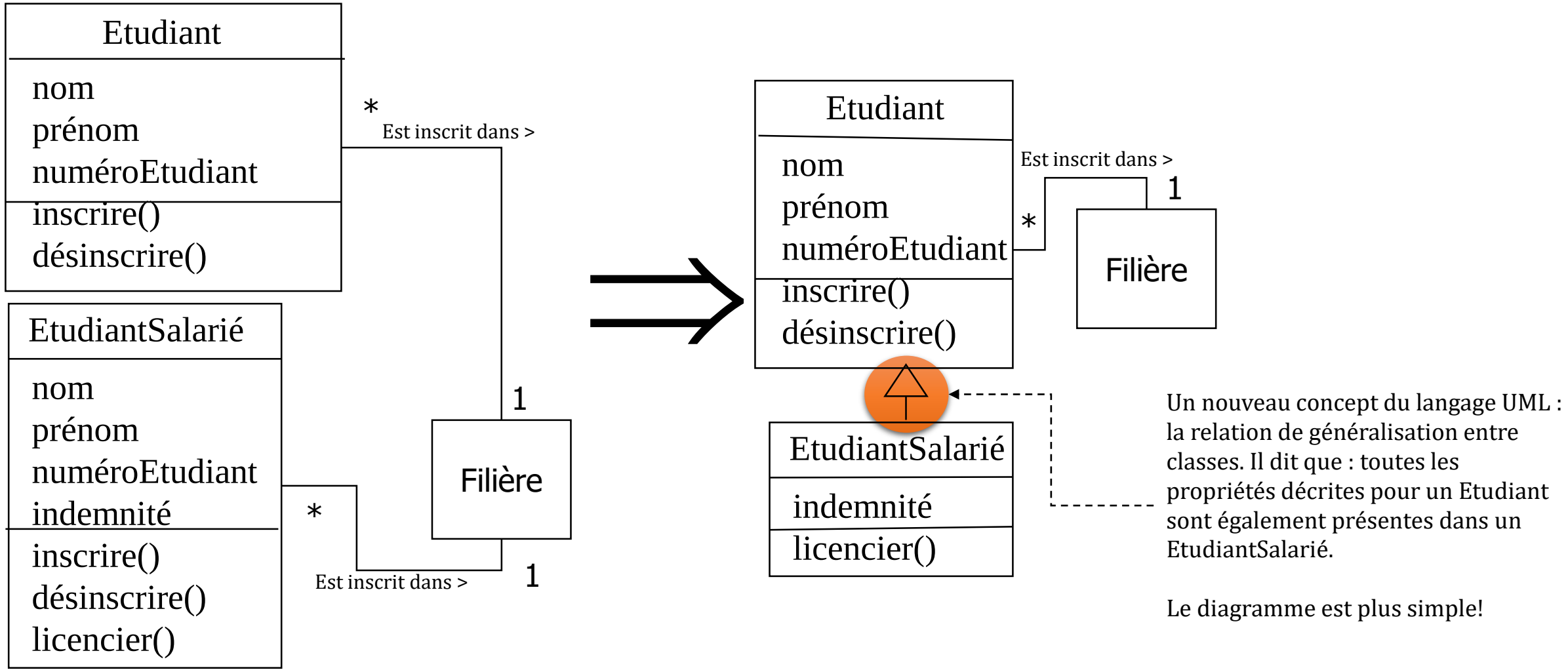
Pour un domaine on obtient ce modèle.

Il y a de nombreux points communs entre les étudiants et les étudiants salariés.

En effet, un étudiant salarié est un étudiant (donc avec les propriétés afférentes) ET un salarié.

Ne pourrait-on pas « construire » un étudiant salarié depuis un étudiant et éviter cette redondance dans le diagramme?

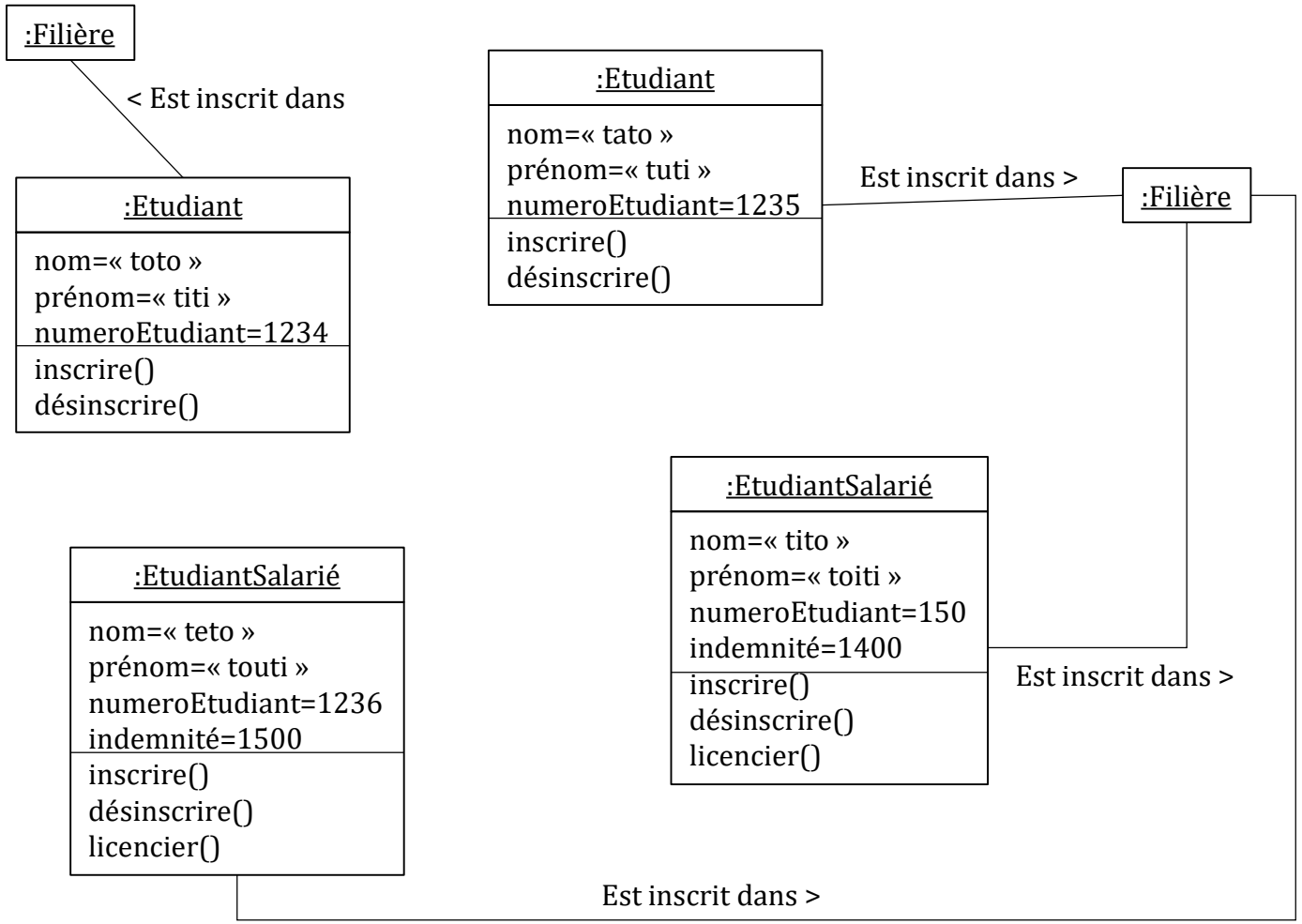
Après la classe, la super-classe comme outil de gestion de la complexité!



Un nouveau concept du langage UML : la relation de généralisation entre classes. Il dit que : toutes les propriétés décrites pour un Etudiant sont également présentes dans un EtudiantSalarié.

Le diagramme est plus simple!

Un petit diagramme d'objets…

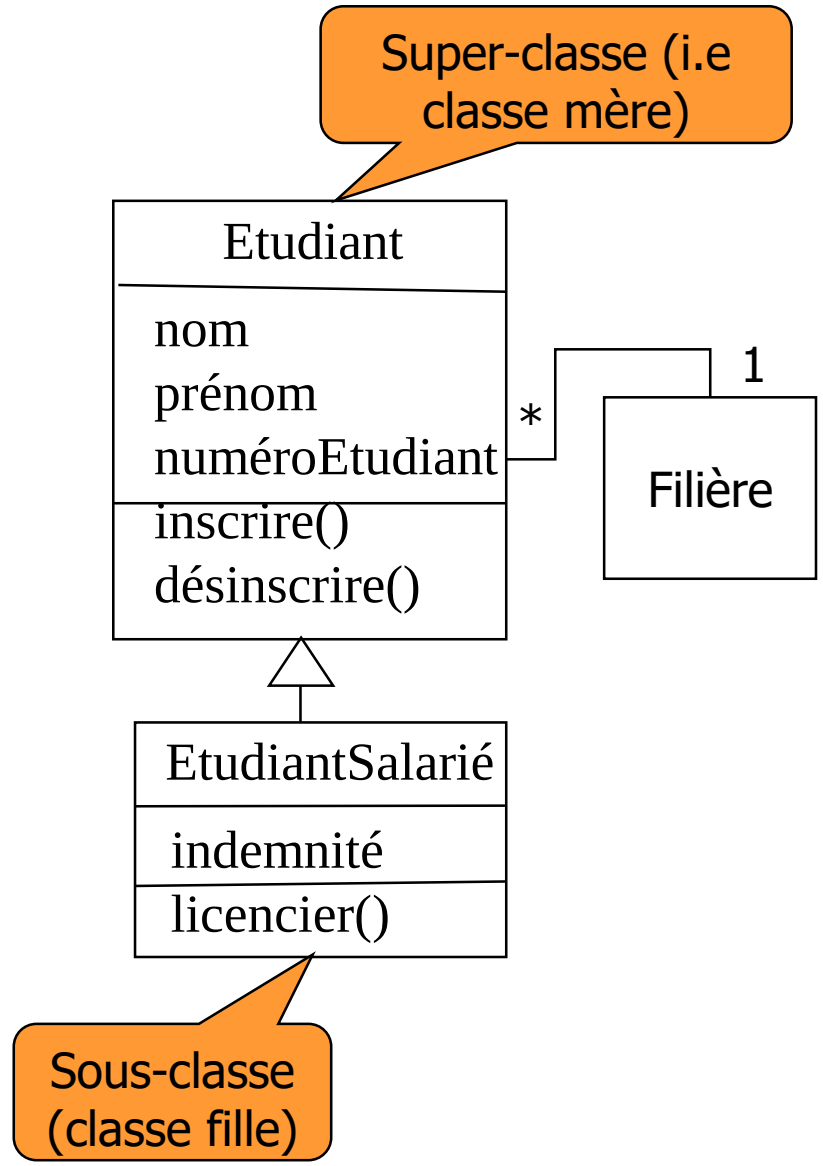


On le voit les objets instances de la classe EtudiantSalarié ont récupéré l'intégralité des propriétés décrites dans la classe Etudiant.

Tout se passe avec la relation de généralisation UML comme si la classe EtudiantSalarié récupérait tout le code de Etudiant.

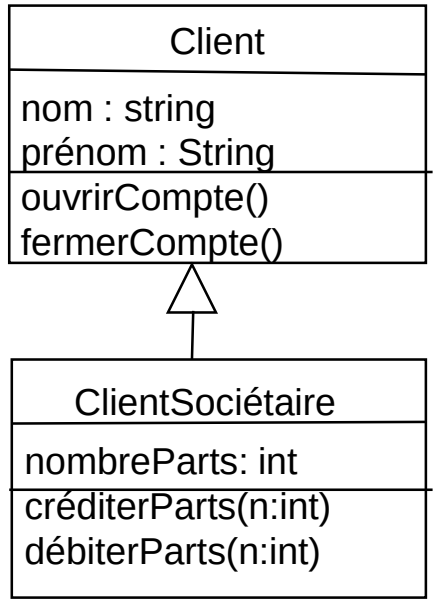
Après la classe, la super-classe comme outil de gestion de la complexité!

- La généralisation UML manifeste la ressemblance de deux concepts, dont l'un peut être vu comme un cas particulier (une extension) de l'autre.
  - Un Etudiant salarié est un étudiant particulier qui dispose de propriétés supplémentaires liées au fait qu'il est salarié.
- La sous-classe UML présente (on dit qu'elle hérite) toutes les propriétés (attributs, méthodes, associations, contraintes) de sa super-classe sans qu'il soit besoin de les rappeler.
  - La généralisation est un mécanisme de réutilisation de code. La sous-classe récupère tout le « code » de sa super-classe.
  - On a ainsi des diagrammes plus simples car on factorise les propriétés communes à plusieurs classes dans un seul endroit.
  - On gagne également en maintenabilité car une modification des propriétés d'un Etudiant ne se fera que dans la classe Etudiant. L'étudiant salarié récupèrera sans travail supplémentaire cette modification.



Un second exemple

- Un client sociétaire est un client qui possède des parts dans la banque et des méthodes pour gérer ce nouvel aspect.

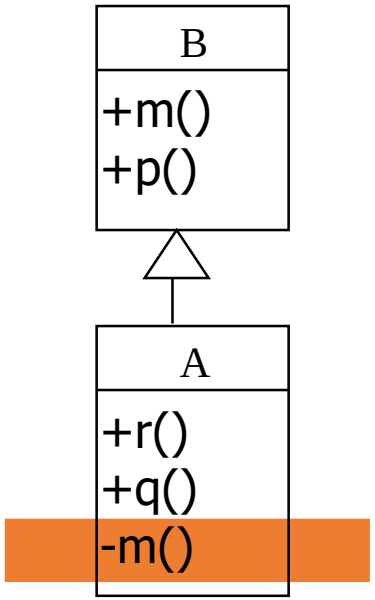


TRES IMPORTANT (cela sera compris en R2.01 en fin d'année)!

- UML offre le concept de généralisation et Java celui d'héritage. L'héritage Java est une version moins puissante de la généralisation UML.
  - La généralisation fait hériter les attributs et les méthodes (comme Java) mais rajoute les associations et les contraintes (qui n'existent pas en Java).
  - La généralisation est multiple (une classe peut avoir plusieurs super-classes) alors qu'en en java l'héritage est simple (une classe a au plus une seule super-classe)
- Par contre généralisation UML et héritage sont tous les deux :
  1. un mécanisme de réutilisation de code mais également
  2. un mécanisme de sous-typage.
- Dans R1.05 seul le premier point de la généralisation sera utilisé. Mais le second est fondamental et sera étudié plus tard(R2.01)…

- Du fait de cette double sémantique (réutilisation/sous-typage) de la Généralisation UML la réutilisation de code ne peut plus se faire n'importe comment (dommage!).
  - Une sous-classe doit toujours être construite de manière à être capable d'offrir au minimum les propriétés offertes par ses super-classes.
  - Il est donc impossible de faire une réutilisation à la carte : on ne peut pas choisir d'hériter certaine des propriétés de sa super-classe mais d'en refuser d'autres…

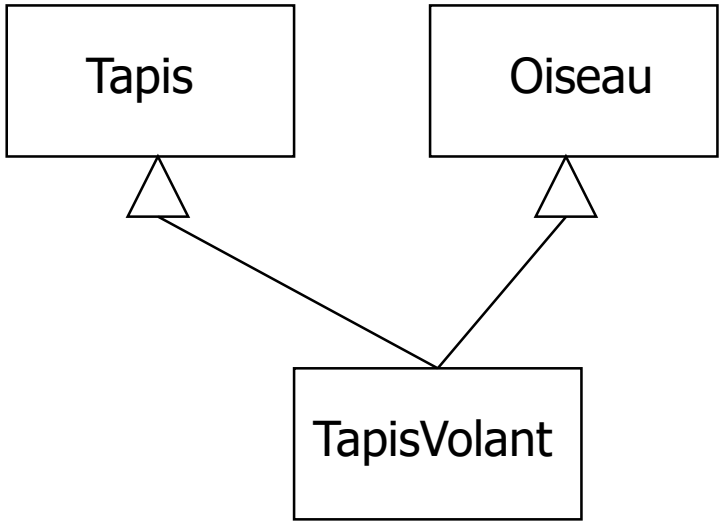
- UML et le compilateur Java autorisent donc une sous-classe à :
  - ajouter de nouveaux attributs (JAVA, UML)
  - ajouter de nouvelles méthodes (JAVA, UML)
  - redéfinir des méthodes héritées sous réserve de conserver la même signature (JAVA, UML, cf *Redéfinition R2.01*)
  - Redéfinir (masquer) des attributs hérités en conservant ou non le typage (JAVA, UML, cf *Redéfinition R2.01*)
  - ajouter certaines contraintes et de nouvelles associations (UML uniquement)
- Mais ils interdisent
  - De diminuer l'accès à des membres des super-classes.
  - De supprimer dans la sous-classes des membres hérités



Il est interdit de « retirer » ou de diminuer la visibilité dans une sous-classe des membres qui étaient offerts par la super-classe. Sinon une instance de A ne pourrait plus se « substituer » à une instance de A

```
B unB=new A();
unB.m() /* N'est plus possible! */
```

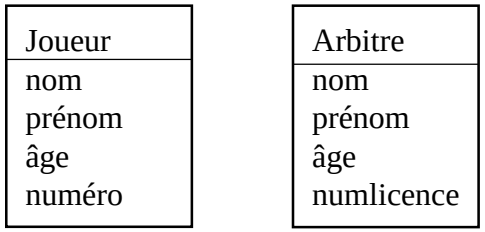
Du coup la restriction sur les possibilités de réutilisation de code est très importante



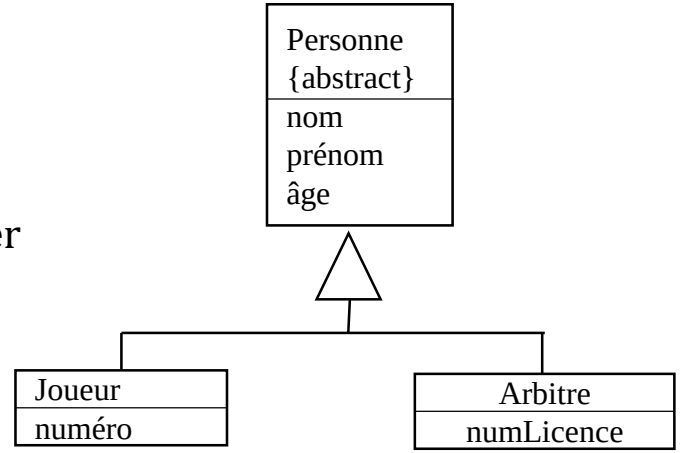
Notre tapis volant qui voulait ne récupérer de Oiseau que sa capacité à voler devra être capable de se comporter comme un oiseau et donc peut être avoir des plumes, savoir chanter et pondre des œufs!

Classe abstraite

- La factorisation de certaines propriétés communes à plusieurs classes dans une super-classe peut engendrer la création d'objets qui n'ont pas de sens dans le domaine.
- Il est donc possible de déclarer une classe abstraite pour indiquer que l'on interdit son instanciation (aucun objet instance directe de la classe ne peut être généré).
- Le caractère abstrait d'une classe est manifesté par une contrainte {abstract} sous le nom de la classe ou en écrivant le nom de la classe en italique (déconseillé sur vos copies car illisible le plus souvent!).



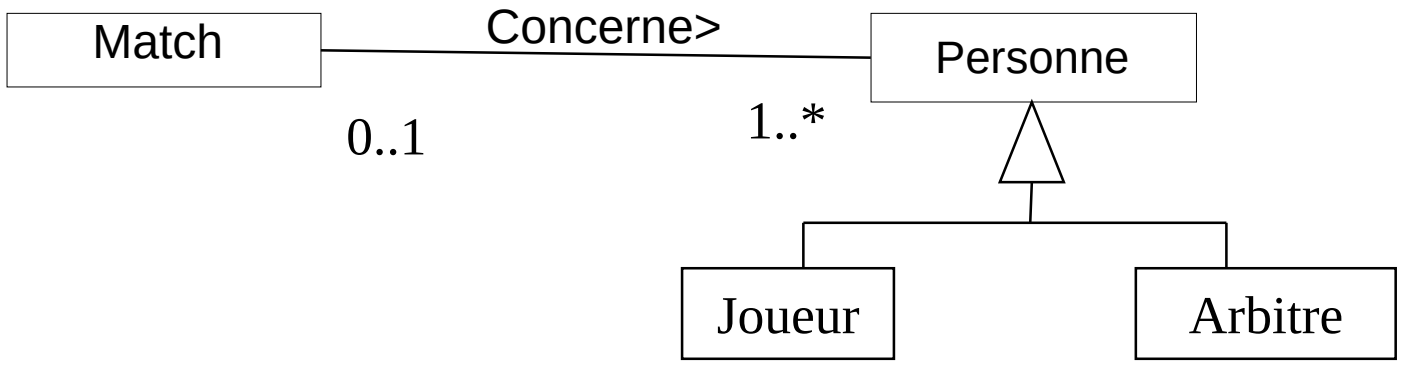
Pour supprimer la redondance et factoriser les parties communes



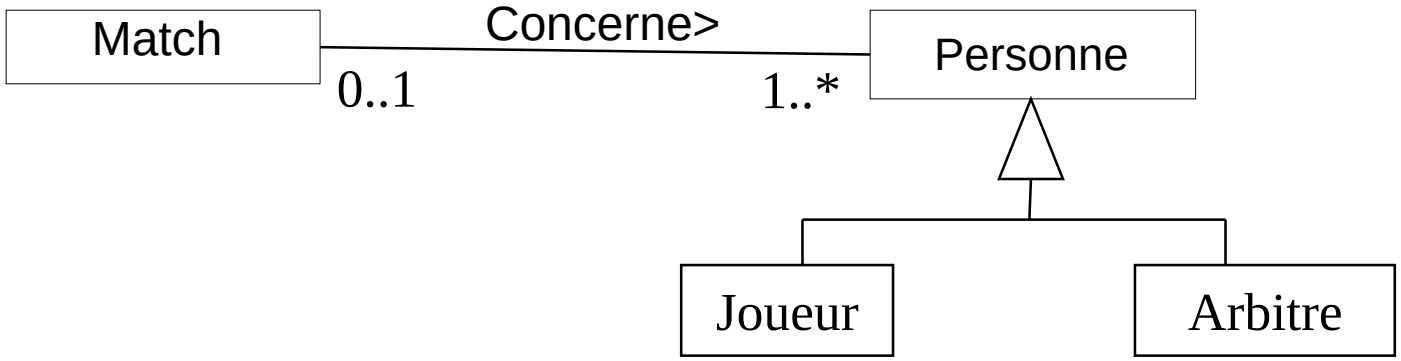
Comme dans le domaine il n'y a que des Joueurs ou des Arbitres, le concept de Personne n'a pas de réalité.

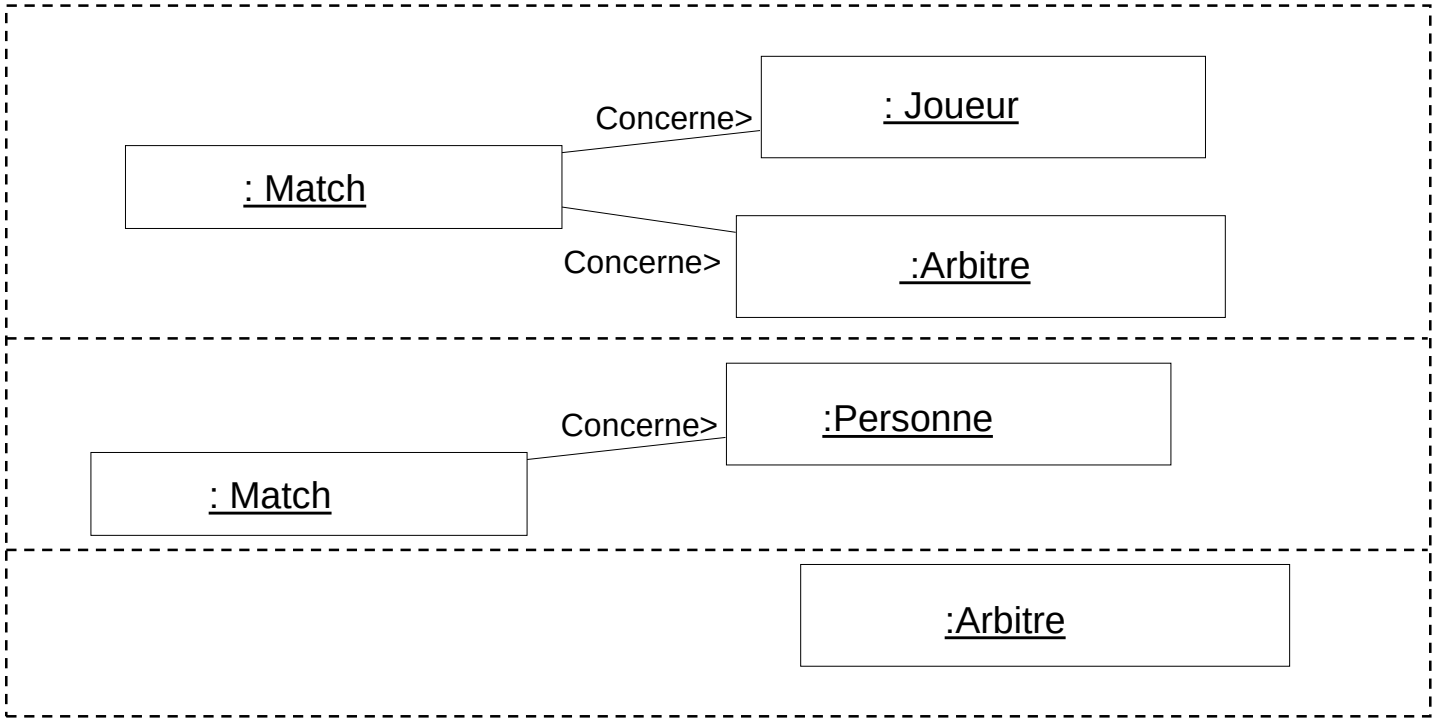
On note la classe Personne abstraite pour éviter d'introduire des objets de type Personne qui n'ont pas de sens dans le domaine!

Le lien entre association et généralisation



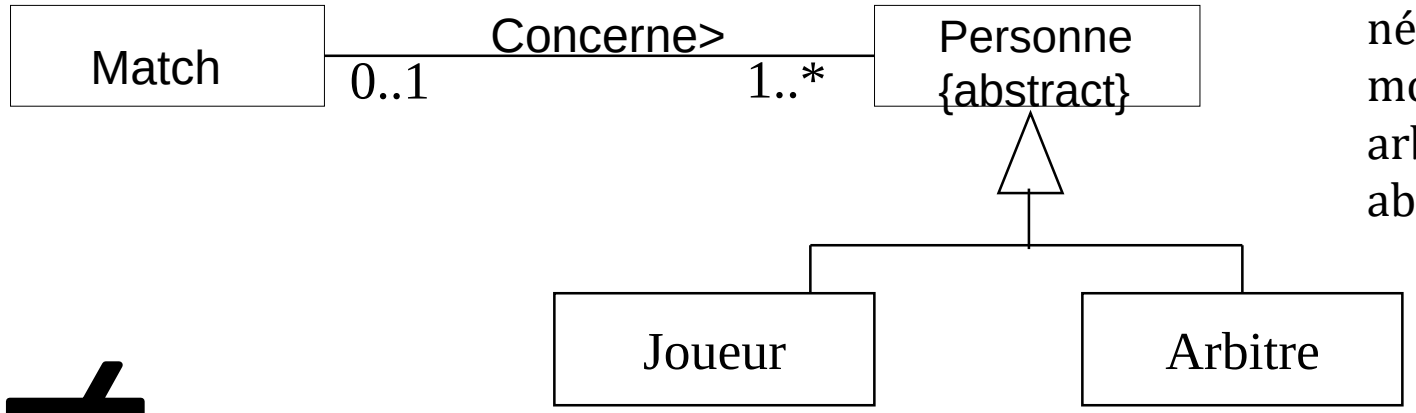
- Les classes Joueur et Arbitre hérite toutes les propriétés de la classe Personne et donc en particulier l'association « concerne »





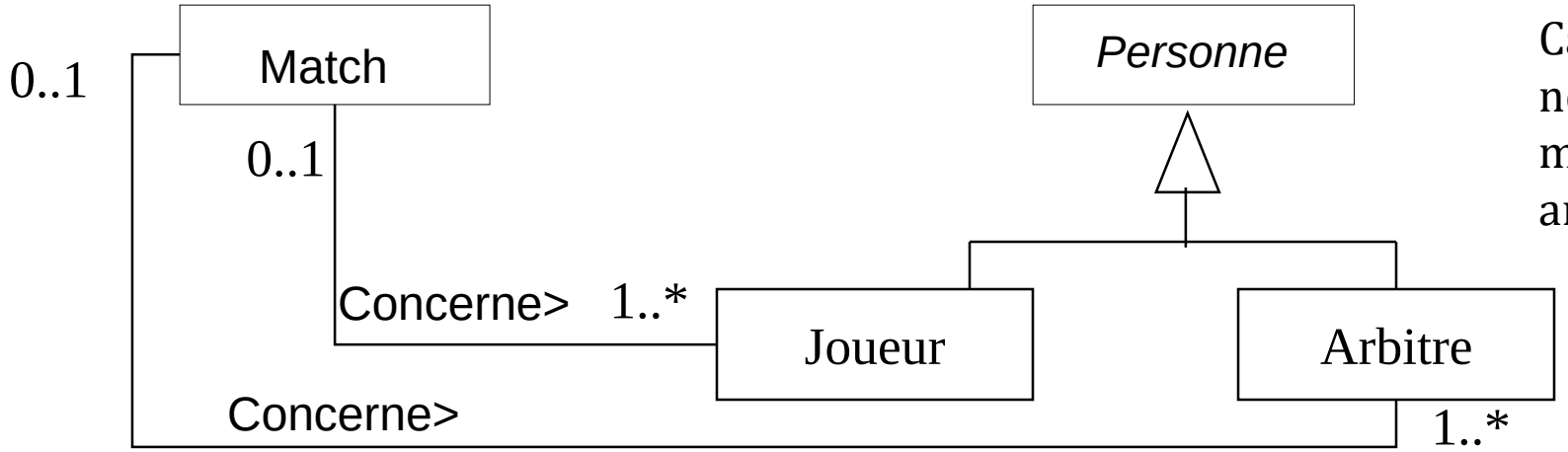
3 Diagrammes d'objets compatibles

Attention aux cardinalités des associations partant des super-classes

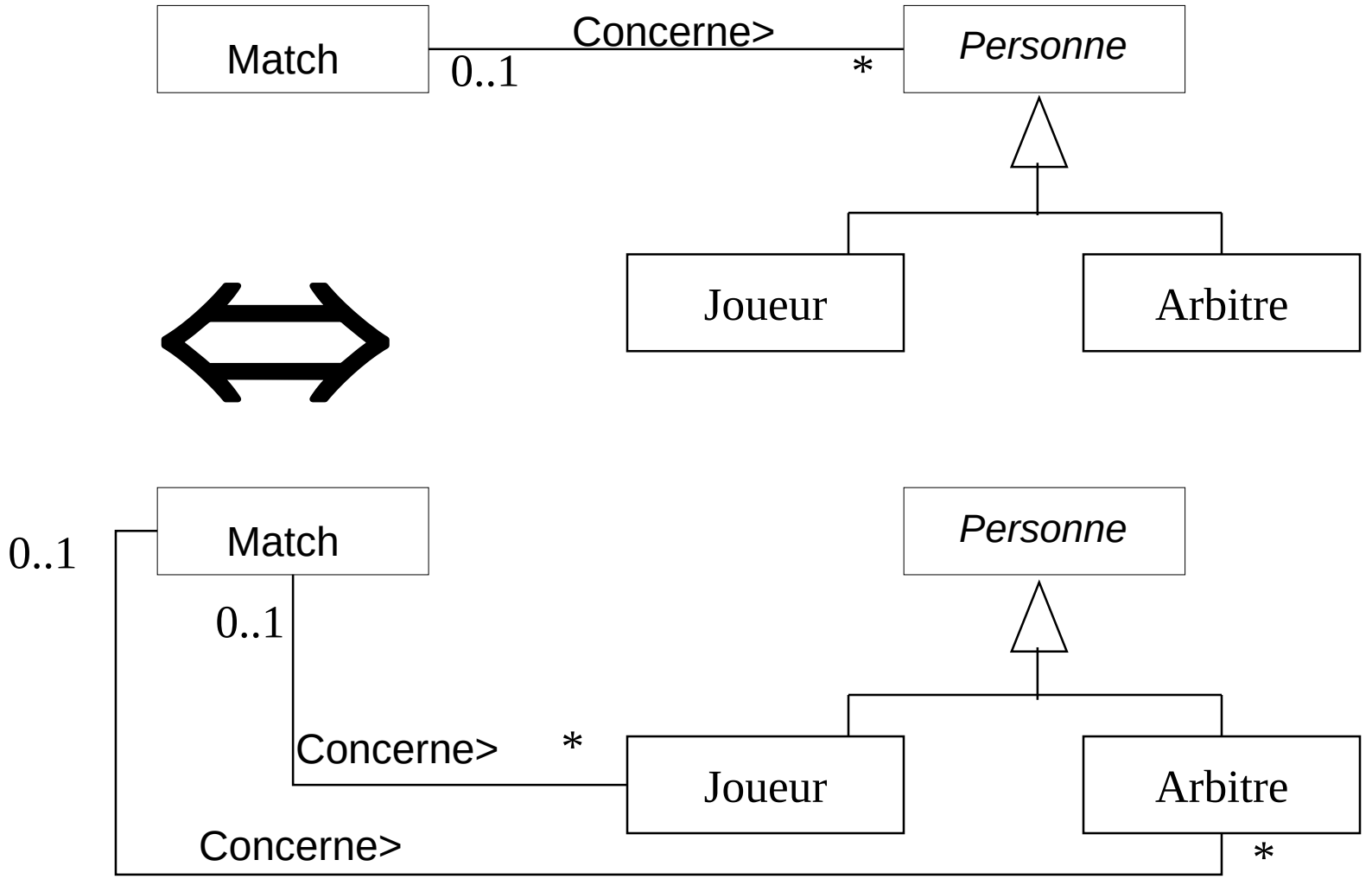


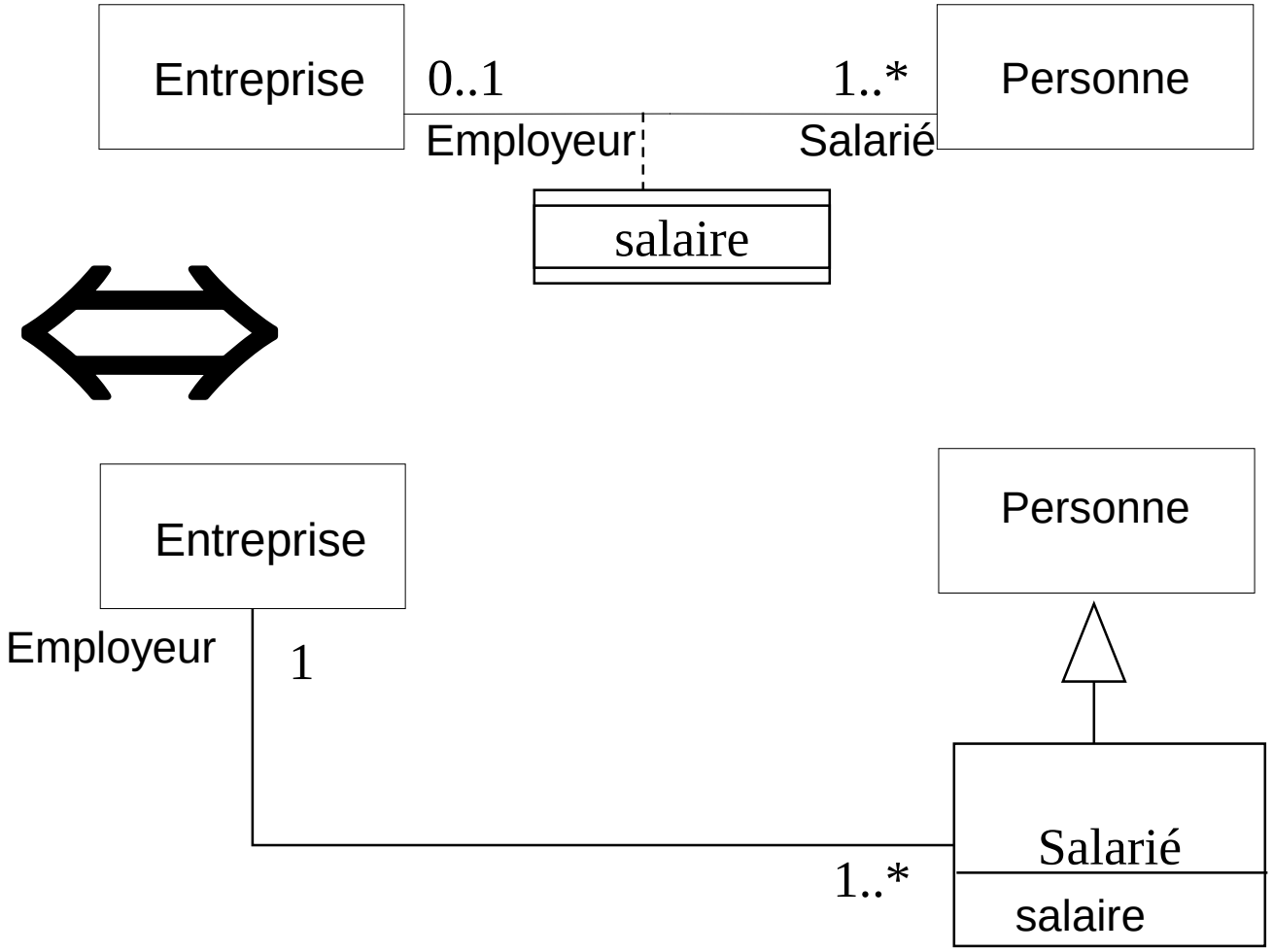
Car ici il y a nécessairement au moins 1 joueur ou 1 arbitre (Personne est abstraite)

≠



Car ici il y a nécessairement au moins 1 joueur et 1 arbitre


Match
Concerne>
0..1
*
Personne
Joueur
Arbitre
0..1
Match
Personne
0..1
Concerne>
*
Joueur
Arbitre
Concerne>
*


Entreprise
0..1
1..*
Personne
Employeur
Salarié
salaire
Entreprise
Personne
Employeur
1
Salarié
1..*
salaire