L3 Info – INF5A1
Jour 3

Notions d'UML
& quelques révisions fondamentales

Evaluation de cette unité

- Un CT (contrôle terminal = examen) de 1h30 sur papier. Documents autorisés. Conception, mise en œuvre de design patterns, tests.
- Un CC (contrôle continu), sous forme d'une application développée par groupes de 3 ou 4, ainsi que d'une note intermédiaire. Certaines parties des TP y seront consacrées.
- Coefficients : 60% CT, 40% CC

Aujourd'hui

- Diagrammes UML :
  - Diagramme de Cas d'utilisation
  - Diagramme de Classes
  - Diagramme de séquence
- Révisions fondamentales :
  - Héritage versus Association
  - static, final

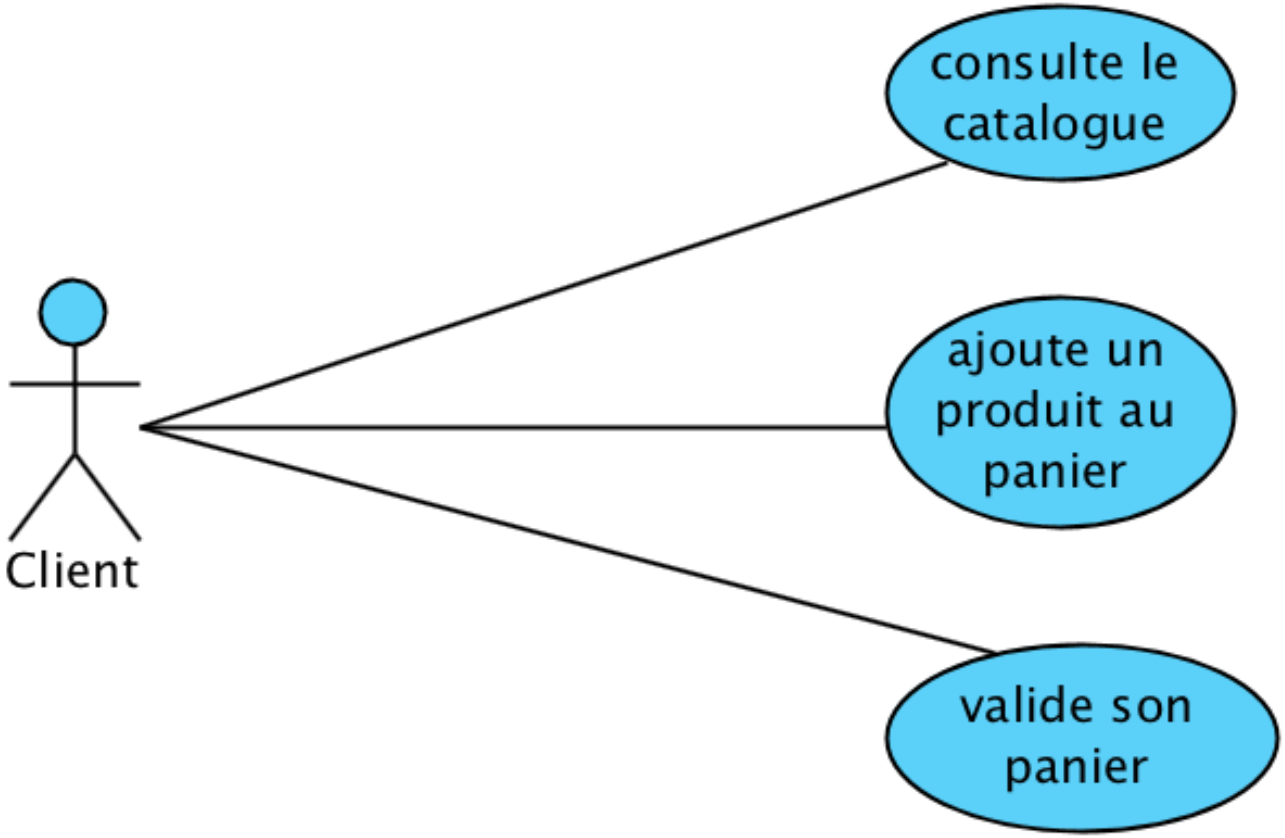
UML

- Unified Modeling Language
- Une « norme de fait »
- Langage essentiellement graphique permettant de décrire les différents aspects d'un développement d'application. Notamment :
  - Qui peut faire quoi avec le logiciel ?
  - Comment les classes sont-elles organisées les unes par rapport aux auteres ?
  - Comments les objets collaborent-ils, et selon quelle chronologie ?

1. Diagramme des Cas d'utilisation

- On définit des rôles (types d'utilisateurs)
- On définit les fonctionnalités importantes
- On lie les utilisateurs aux fonctionnalités qui leur sont dédiées
- On décrit comment les fonctionnalités s'organisent les unes par rapport aux autres (par ex. pour faire B, il faut d'abord faire A).

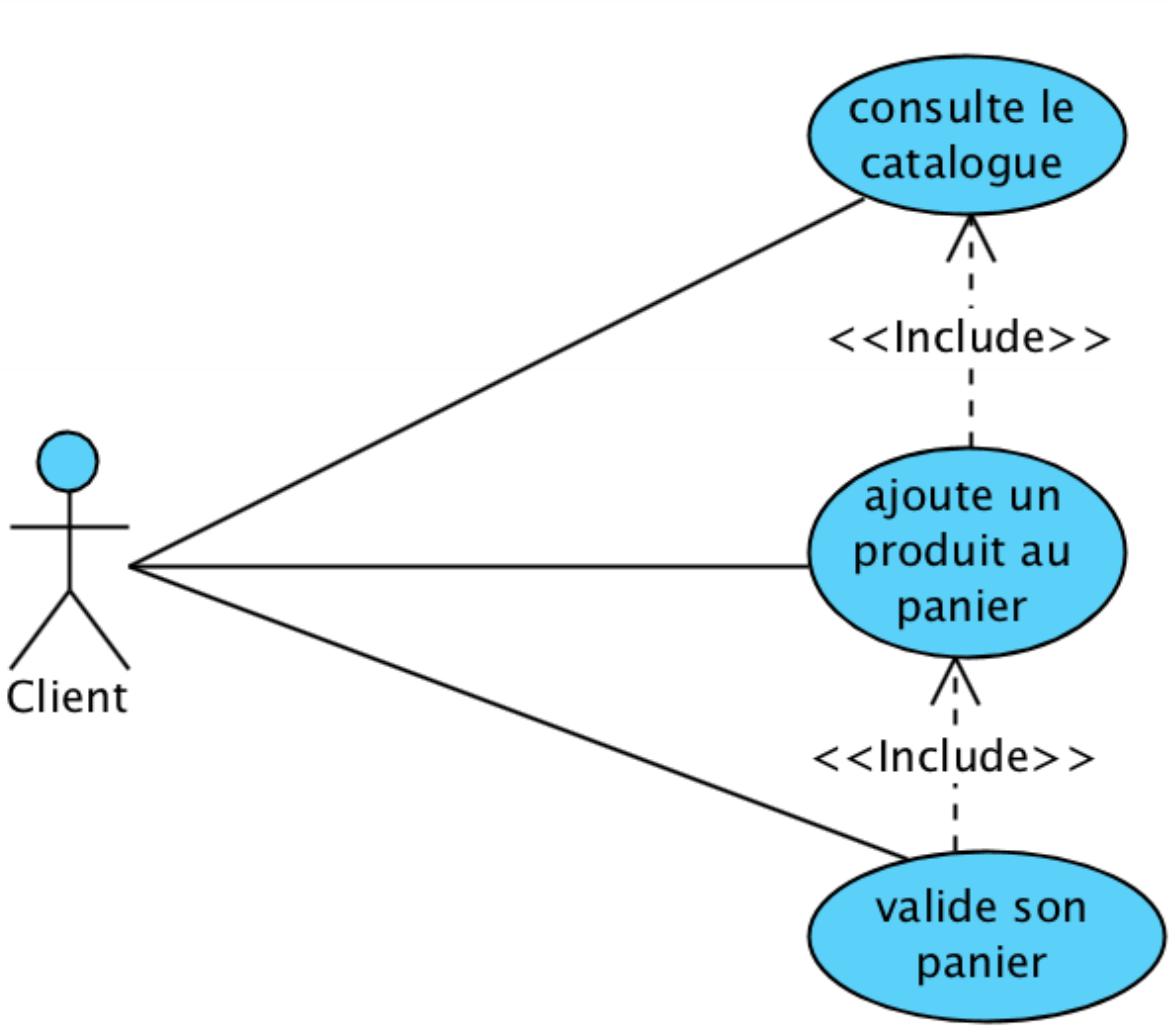
Exemple


consulte le catalogue
ajoute un produit au panier
valide son panier
Client

<< include >>

- Lorsqu'un cas d'utilisation nécessite qu'un autre cas ait été préalablement réalisé.
- Attention au sens de la flèche, qui correspond ici à une implication

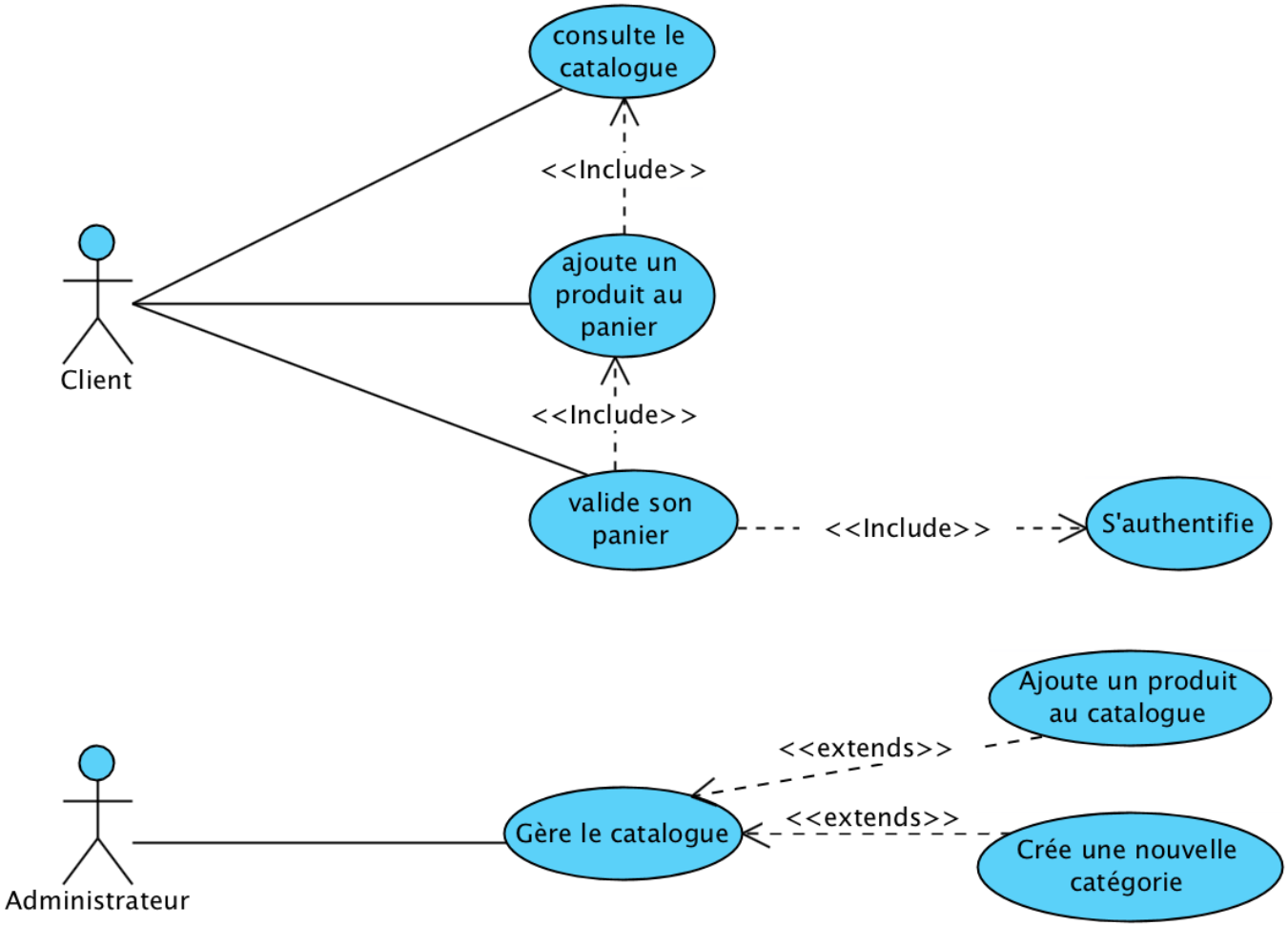
<< include >>


consulte le catalogue
<<Include>>
ajoute un produit au panier
<<Include>>
valide son panier
Client

<< extends >>

- Lorsqu'un cas d'utilisation permet l'accès à un autre cas
- Permet d'expliciter le détail d'un cas trop général
- Attention au sens de la flèche, souvent contre-intuitif lorsque l'on débute : le sous-cas extends le cas plus général

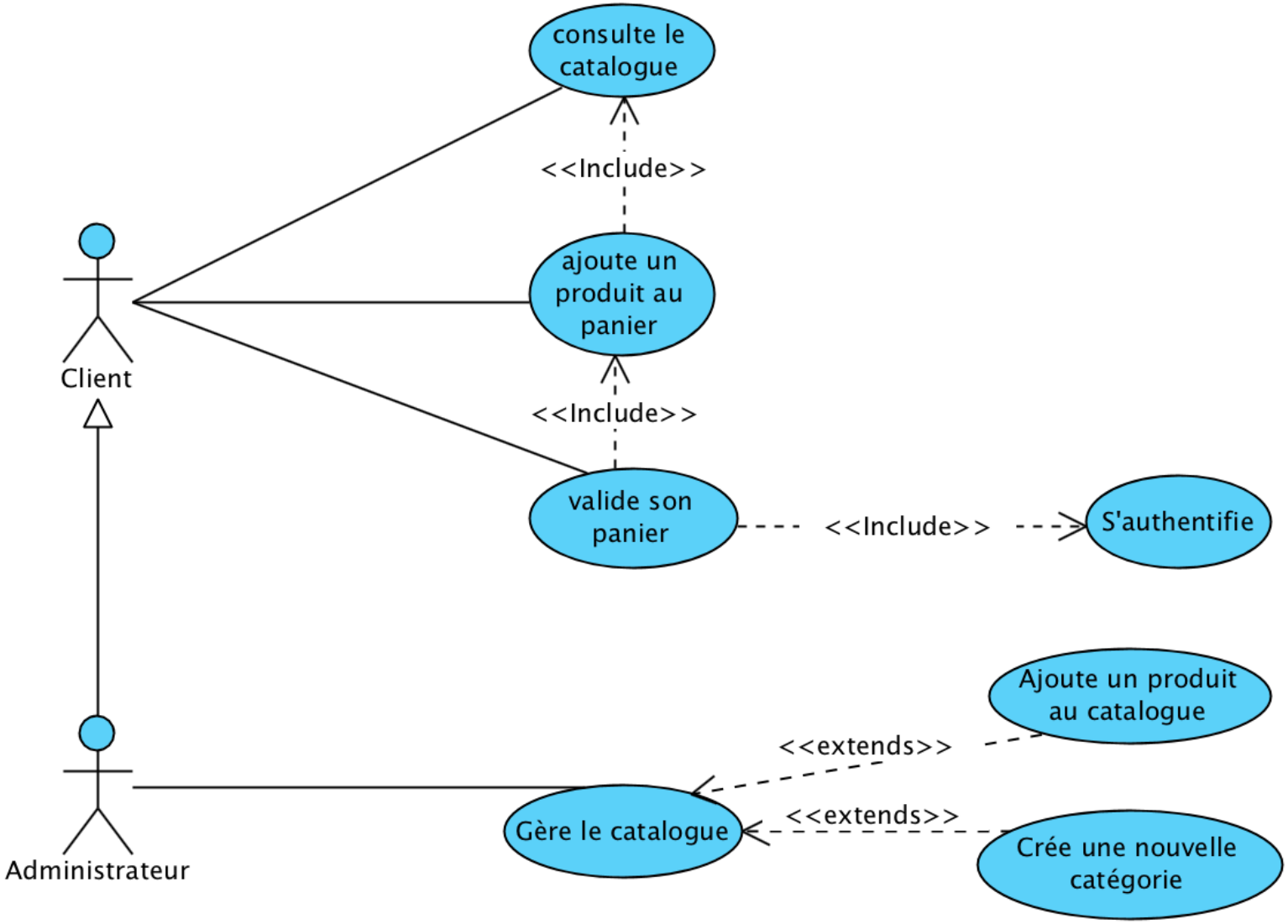
<< extends >>


consulte le catalogue
<<Include>>
ajoute un produit au panier
Client
<<Include>>
valide son panier
<<Include>>
S'authentifie
Ajoute un produit au catalogue
<<extends>>
Gère le catalogue
<<extends>>
Crée une nouvelle catégorie
Administrateur

Héritage

- Un utilisateur peut hériter d'un autre
- Tous les cas d'utilisation sont ainsi hérités
- Permet d'éviter une forte redondance, et ainsi facilité la lecture

Héritage


consulte le catalogue
<<Include>>
ajoute un produit au panier
<<Include>>
valide son panier
<<Include>>
S'authentifie
Client
Ajoute un produit au catalogue
<<extends>>
Gère le catalogue
<<extends>>
Crée une nouvelle catégorie
Administrateur

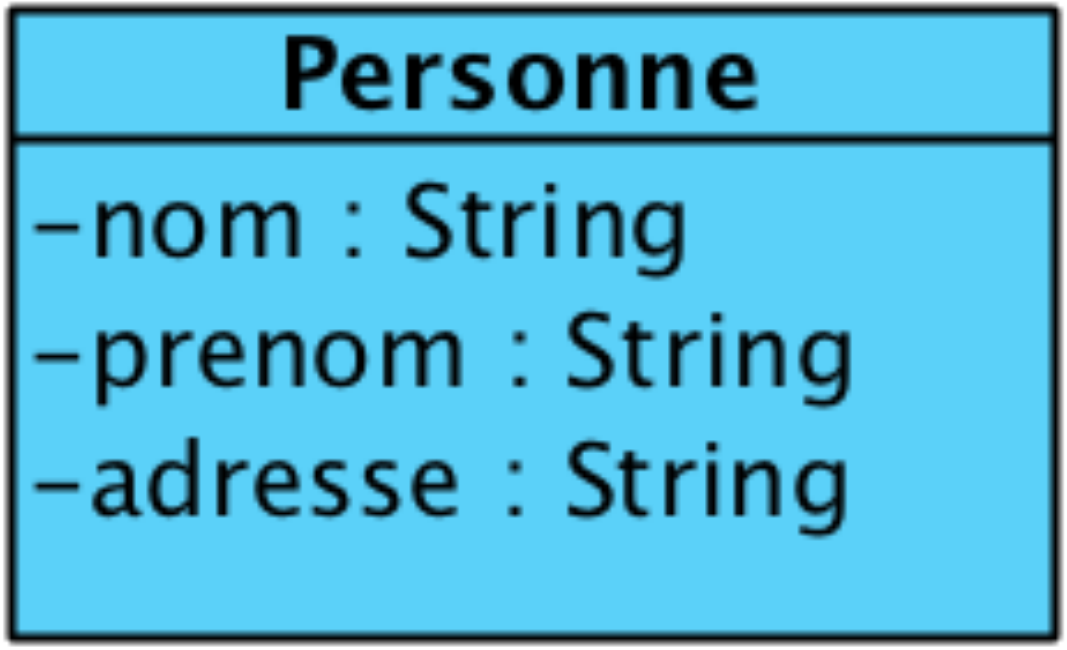
2. Diagrammes de classes

- Il s'agit d'un diagramme statique permettant d'établir les liens entre les classes
- Deux grands types de relations :
  - hierarchiques, de type héritage / implémentation
  - De type association
- Dans la mesure du possible essayer de représenter les liens hiérarchiques verticalement, et les autres horizontalement

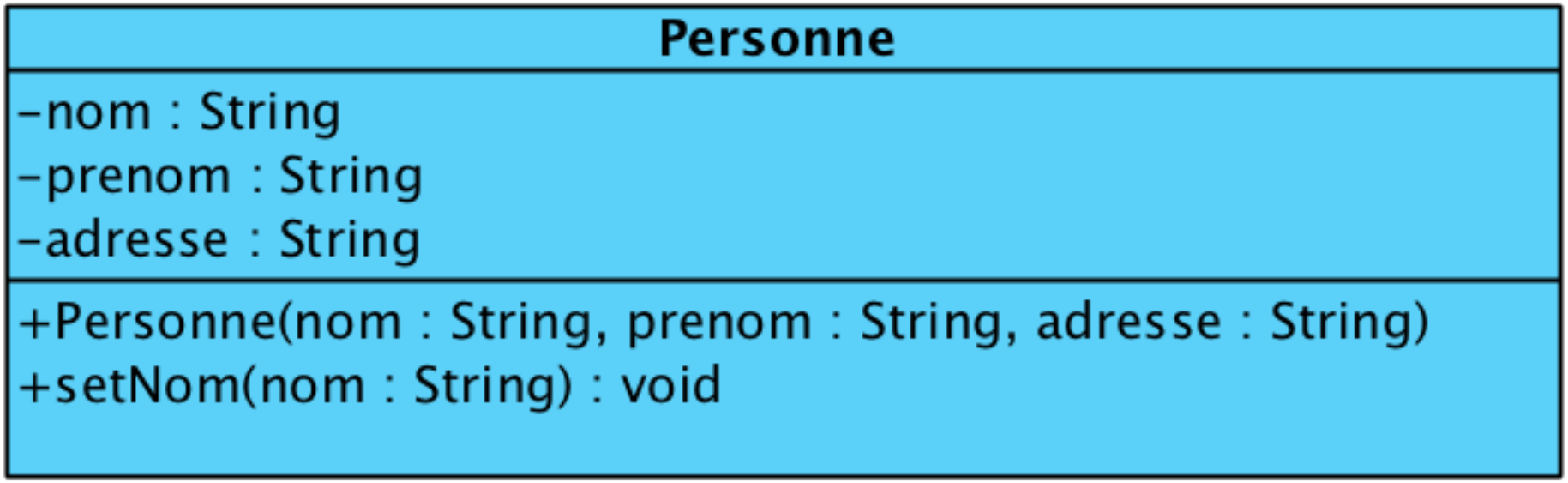
Au niveau d'une classe

- 3 niveaux :
  - nom de la classe
  - attributs
  - méthodes
- Seul le premier niveau est impératif
- Les deux autres niveaux seront détaillés seulement si on les juge pertinents (selon l'usage auquel le diagramme est destiné)

exemple : nom + attributs



exemple 2 : nom + constructeurs + méthodes



Différents niveaux d'encapsulation

- public : +
- private : -
- protected : #
- package : par défaut

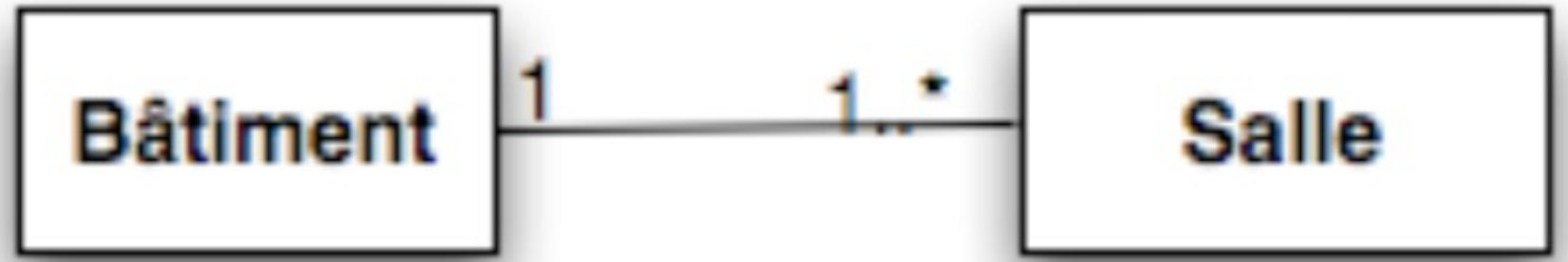
- On choisit toujours le niveau d'encapsulation maximum (on peut toujours élargir par la suite)

Association

- composition : losange plein du côté du contenant
- agrégation : losange vide du côté du contenant
- association simple : trait simple

- On peut nommer la relation (possède, contient, gère, etc.)

Association (simple)



Agrégation

Composition

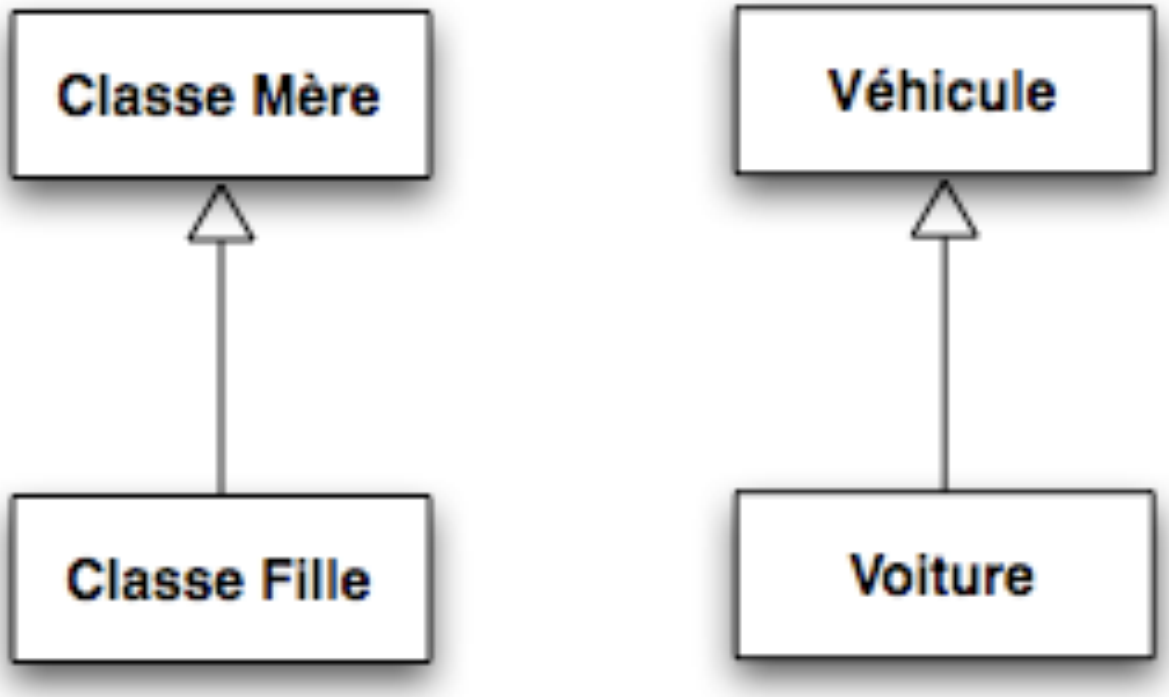
- Liaison maximale entre contenant et ses composants
- Durée de vie du composant liée à celle de son contenant
- Un composant ne peut pas appartenir à plus d’un contenant (contre-exemple : mur mitoyen)

Composition

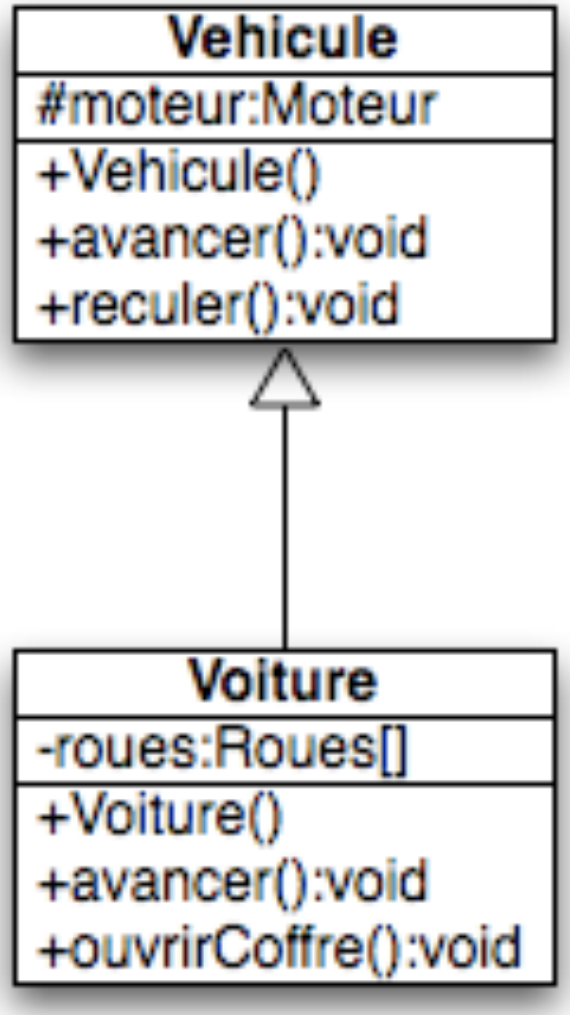
Classes versus Instances

- Attention, un diagramme de classes n'est pas un diagramme d'instance
- Il a des conséquences sur les liens entre les instances créées, mais il y a des différences
- Exemple : dans la diag. précédent, une instance de moteur ne peut pas être liée simultanément à un Bateau et à une Voiture, alors que les classes le sont.

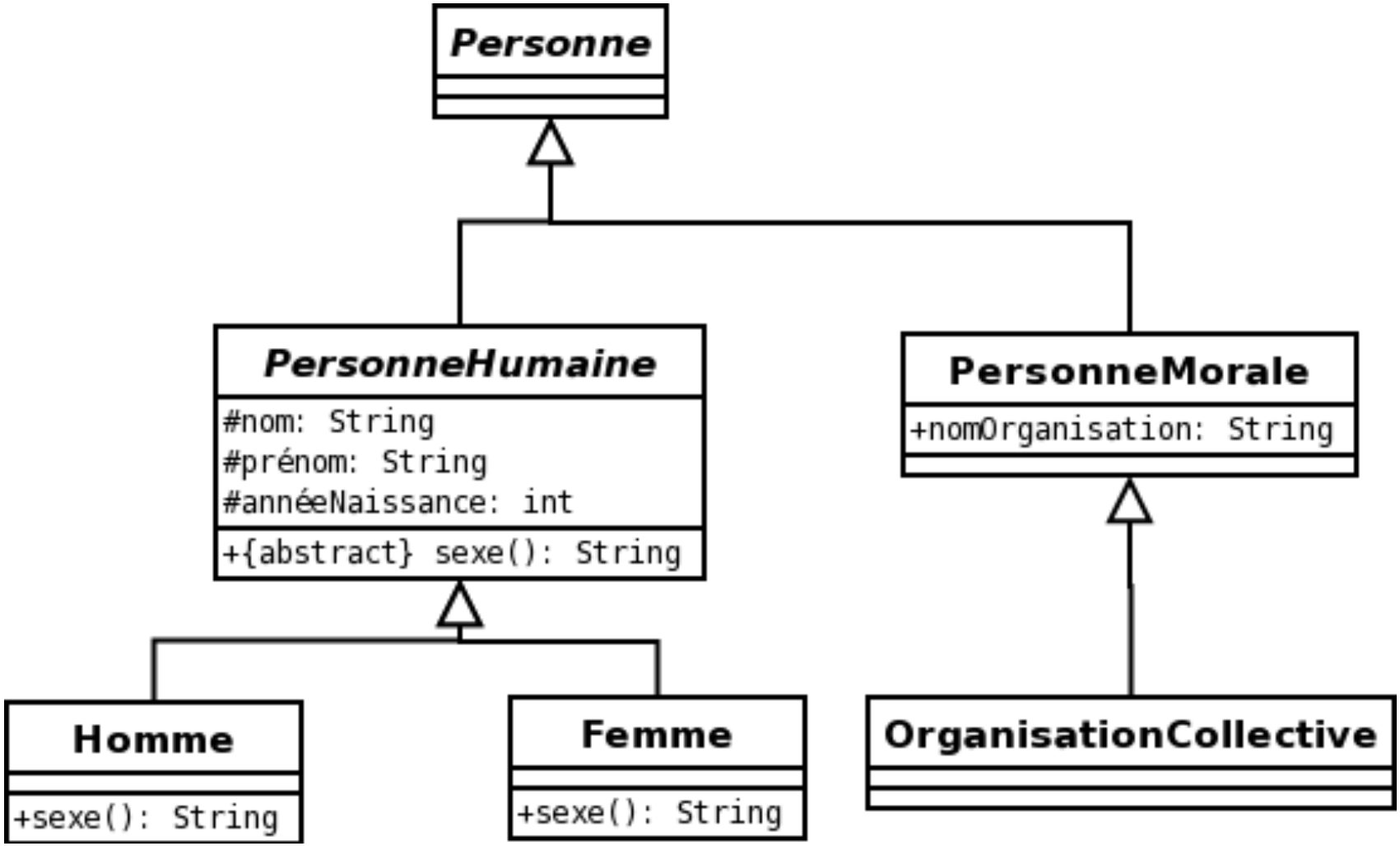
Héritage : extends



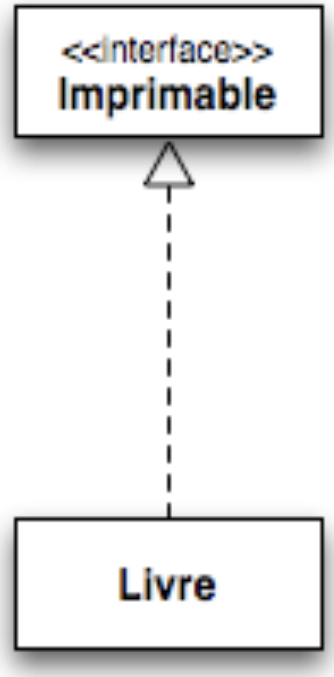
Héritage (2)


Vehicule
#moteur:Moteur
+Vehicule()
+avancer():void
+reculer():void
Voiture
-roues:Roues[]
+Voiture()
+avancer():void
+ouvrirCoffre():void

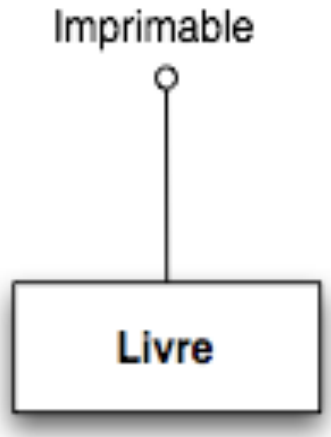
Classes abstraites : italiques



Interfaces



Version classique



Version abrégée

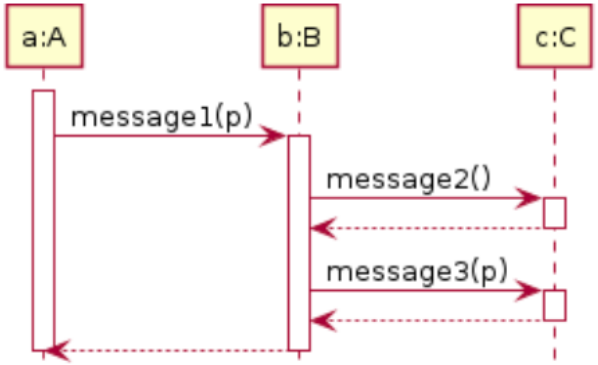
Diagrammes de Séquences

- Interactions entre objets au fil du temps
- Un diagramme correspond à un scénario possible d'un cas d'utilisation
- **Acteurs** de ce scénario disposés en haut
- Acteur principal à gauche du diagramme
- Acteurs secondaires disposés horizontalement de gauche à droite
- La **ligne du temps** va de haut en bas

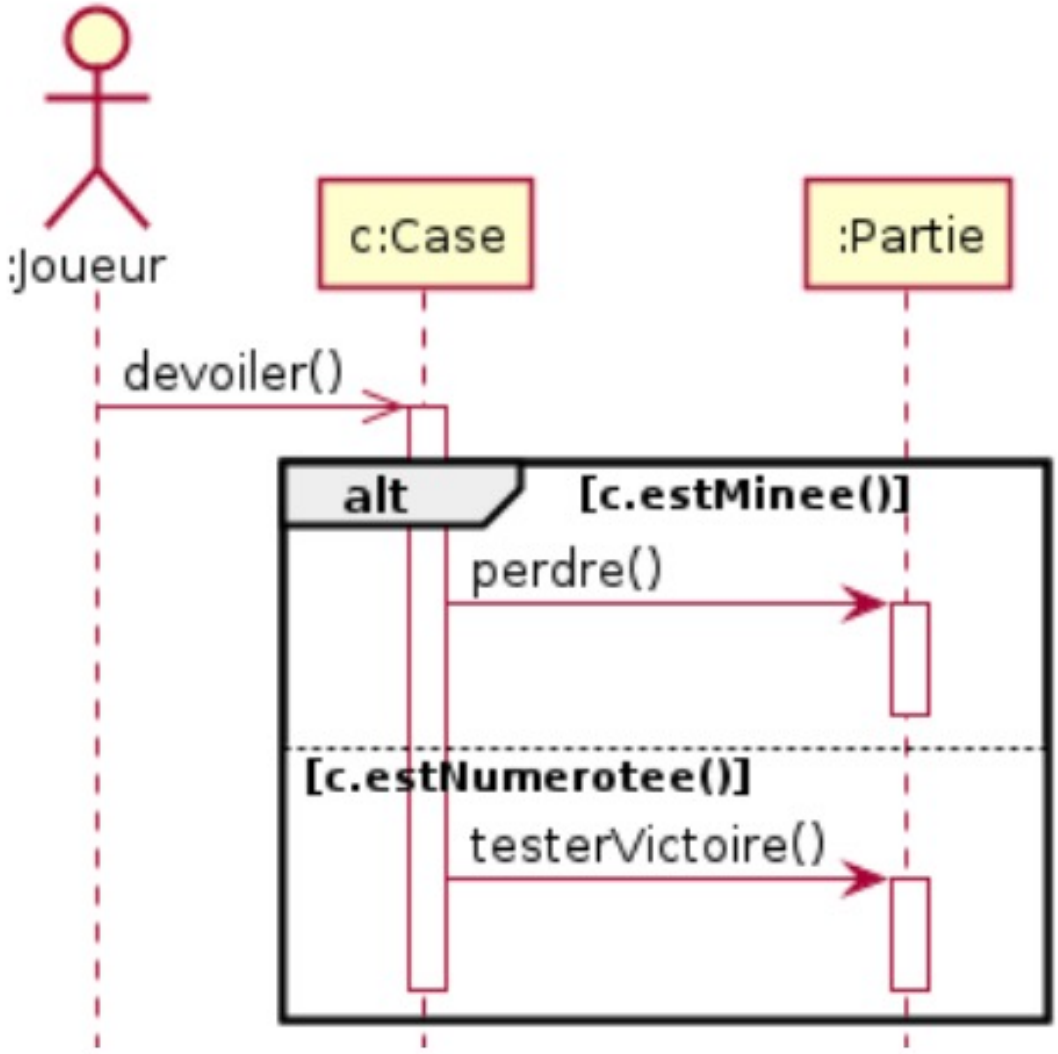
Diagramme de séquence : exemple

(illustrations par Pierre Gérard, IUT Paris XIII)

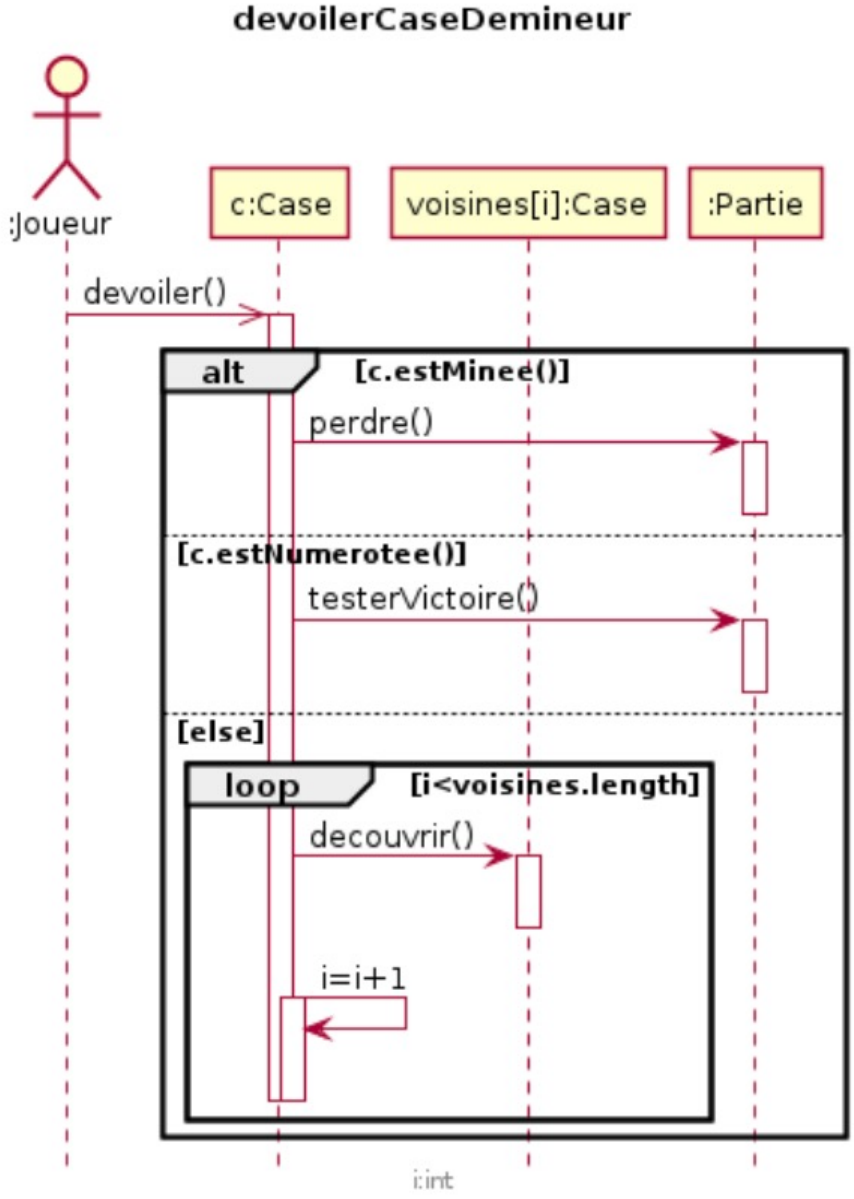
```
1  class B {
2      C c;
3      message1 (p:Type) {
4          c.message2 ();
5          c.message3 (p);
6      }
7  }
8  class C {
9      message2 () {
10         ...
11     }
12     message3 (p:Type) {
13         ...
14     }
```



Alt : si, alors, sinon



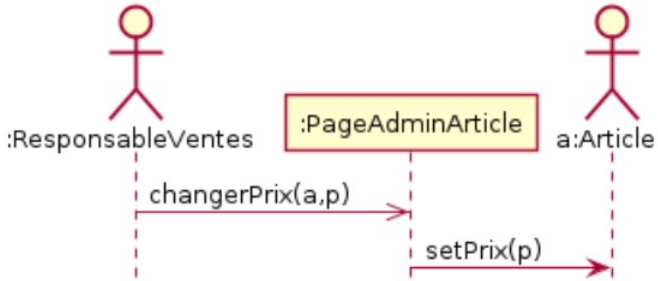
Boucle : loop


devoilerCaseDemineur
:Joueur
c:Case
voisines[i]:Case
:Partie
devoiler()
alt
[c.estMinee()]
perdre()
[c.estNumerotee()]
testerVictoire()
[else]
loop
[i<voisines.length]
decouvrir()
i=i+1
i:int

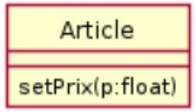
Interactions entre diagrammes

- Cas d'utilisation

- Séquences



- Classes pour spécifier les opérations nécessaires



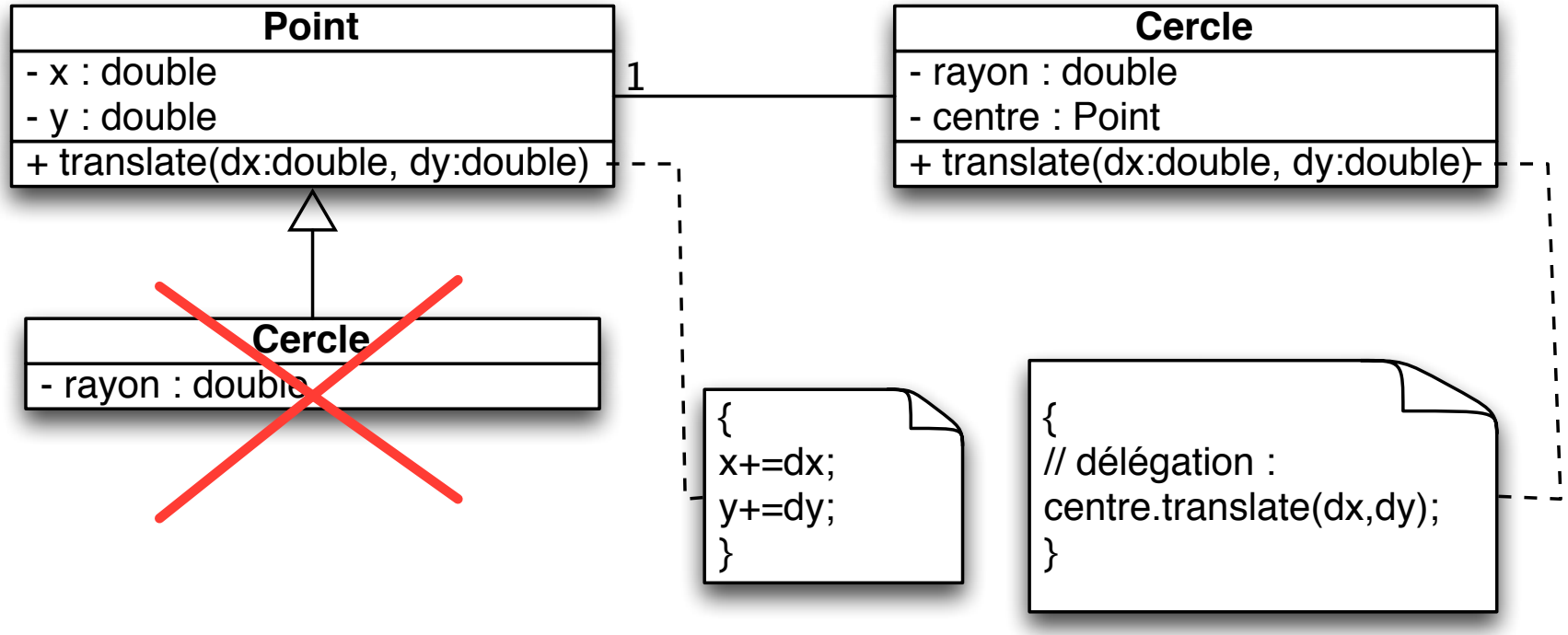
Quelques recommandations

- Pensez à qui s'adressent vos diagrammes
- Eventuellement, faire plusieurs versions (de la plus légère à la plus riche) :
  - pour communiquer avec le client
  - pour interfacer les développements de plusieurs développeurs
  - pour détailler les parties plus complexes
- Le but est d'être informatif : faire ni trop, ni trop peu

Rappels fondamentaux

- Héritage versus Association
- Polymorphisme
- Méthodes virtuelles
- static et final

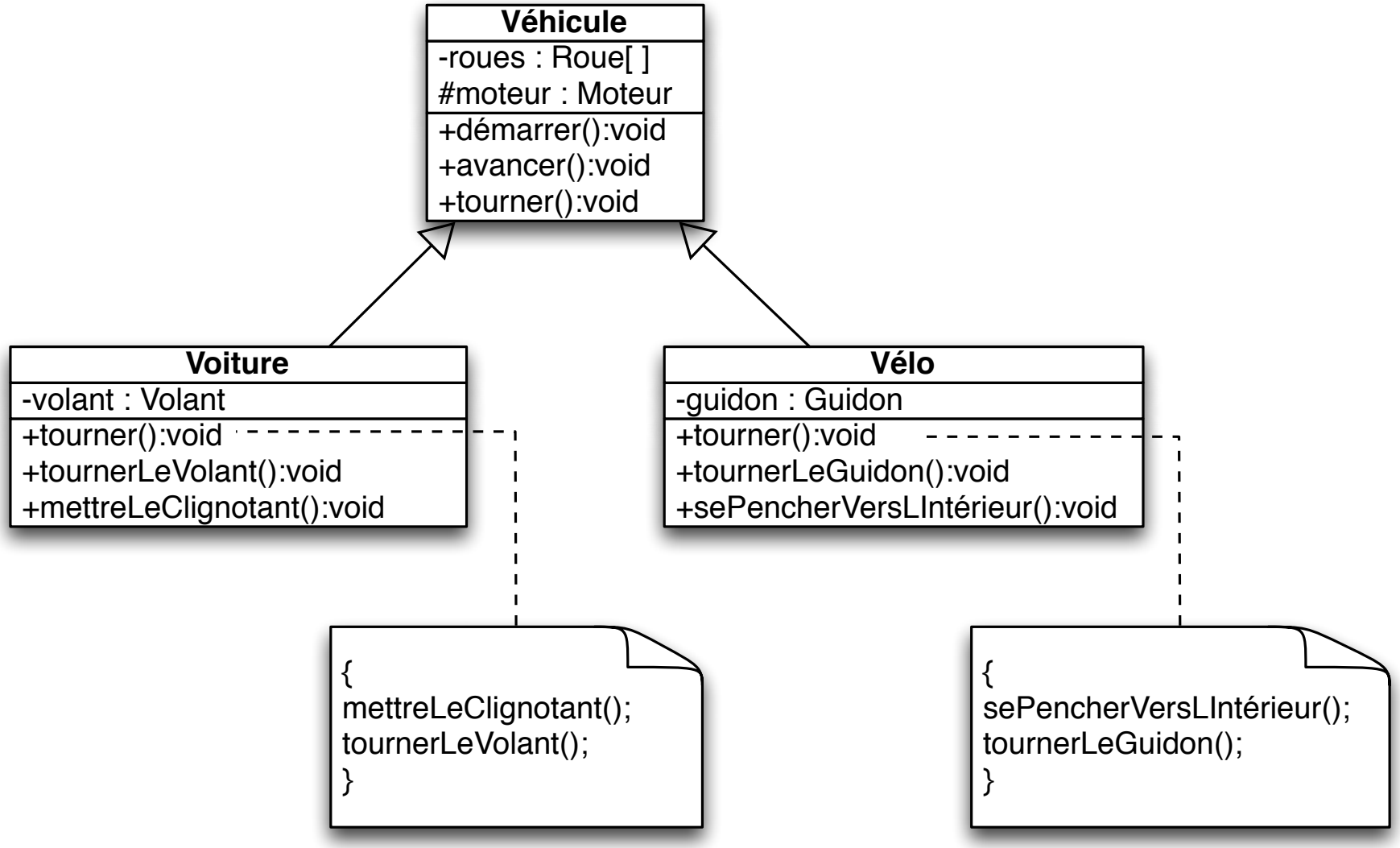
Héritage versus Association



Héritage → relation « est-un »

- Le but premier de l'héritage est le polymorphisme
- Ne doit pas être utilisé comme astuce pour simplifier le code
- Vérifier la sémantique des liens entre classes en interrogeant la relation « est-un » :
  - est-ce qu'une Voiture est un Véhicule ?
  - est-ce qu'un Cercle est un Point ? Si oui, les sommets d'un polygone peuvent être des Cercles !

Polymorphisme et méthodes virtuelles



Méthodes virtuelles (2)

Code de test :

```
Véhicule[ ] parcAuto=new Véhicule[2];
parcAuto[0]=new Voiture();
parcAuto[1]=new Vélo();
for (int i=0;i<parcAuto.length;i++)
   {
   parcAuto[i].démarrer();
   parcAuto[i].avancer();
   parcAuto[i].tourner();
   }
```

révisions CM1 : static, final

- Notions encore parfois confondues
- Notions orthogonales :
  - static : appartient à la classe et non à ses instances
  - final : ne peut plus être modifié après initialisation
  - static final : appartient à la classe et ne peut pas être modifié (pseudo-constante)
  - non static non final : appartient aux instances, modifiable
- Le tout peut être public, private, protected ou package

static : quand l'utiliser ?

- Quand le traitement ou la donnée ne dépend d'aucune instance. Ex. :
  - Math.PI
  - System.out
  - Math.abs(double v)
- Ne doit pas être utilisé comme moyen simple d'accéder à des attributs. Certains débutant mettent tout en static car ils manipulent mal les références sur leurs objets.
- La grande majorité des attributs sont static, à n'utiliser que pour ce pour quoi il est prévu...