3. UML - Unified Modeling Language
Diagrammes statiques

Laëtitia Matignon
laetitia.matignon@univ-lyon1.fr

Département Informatique - Polytech Lyon
Université Claude Bernard Lyon 1
2012 - 2013

Plan du cours

Plan

Qu'est-ce qu'UML ?

- UML = Unified Modeling Language
- UML = Langage unifié pour la modélisation objet

Définition d'UML selon l'OMG

Langage visuel dédié à la spécification, la construction et la documentation des artefacts d'un système logiciel

- OMG = Object Management Group (www.omg.org) : Fondé en 1989 pour standardiser et promouvoir l'objet

Qu'est-ce qu'UML ?

- UML = Unified Modeling Language
- UML = Langage unifié pour la modélisation objet

- UML est un **langage** universel de modélisation objet ... pas une méthode

Différence Langage – Méthode

- Langage de modélisation = notations, grammaire, sémantique
- Méthode = comment utiliser le langage de modélisation (recueil des besoins, analyse, conception, mise en oeuvre, validation, ...)

Qu'est-ce qu'UML ?

- UML = Unified Modeling Language
- UML = Langage unifié pour la modélisation objet

- UML est un langage universel de **modélisation objet** ...

Pourquoi modéliser ?

- La description de la POO nécessite un travail conceptuel : définition des classes, de leurs relations, des attributs, des opérations (implémentées par des méthodes), des interfaces, ...
- Il faut organiser ses idées, les documenter, organiser la réalisation, définir des modules, ...
- Modélisation = démarche antérieure à l'implémentation

Qu'est-ce qu'UML ?

- UML = Unified Modeling Language
- UML = Langage unifié pour la modélisation objet

- UML est un langage universel de modélisation objet
- UML est une **notation**, un outil de communication visuelle (diagrammes)
- UML est un langage de modélisation des applications construites à l'aide d'objets
- UML n'est pas un langage de programmation
- UML n'est pas un processus de développement
- UML est indépendant d'un langage de programmation
- UML est une **norme** maintenue par l'OMG
- Description exacte : `http://www.omg.org/uml`

Qu'est-ce qu'UML ?

- UML = Unified Modeling Language
- UML = Langage unifié pour la modélisation objet

- UML est un langage **universel** de modélisation objet

Pourquoi universel ?

Genèse d'UML : guerre des méthodes

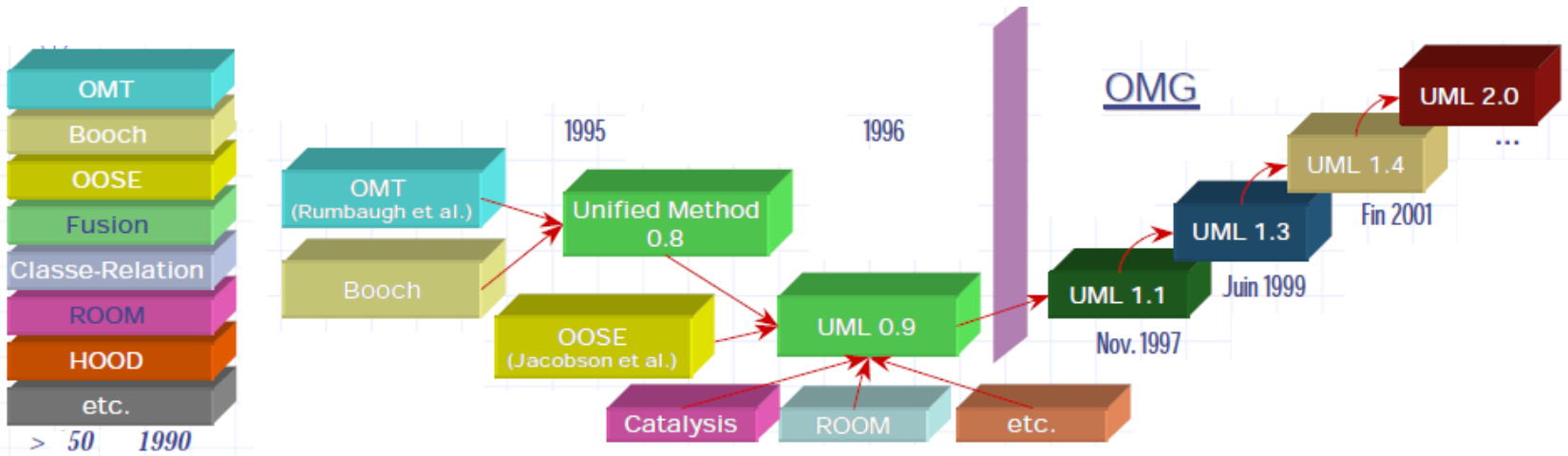
- fin80/début90 : orientation de + en + marquée vers les méthodes d'analyse OO → nécessité de méthodes adaptées.
  - Jusqu'au milieu des années 90, de nombreux modèles objets sont proposés (>50) :
    - Booch'91 puis Booch'93 de Grady Booch, pionnier de l'Orienté-Objet. Distingue 2 niveaux :
      - Logique : Diagrammes de classes, Diagramme d'instances, Diagramme états/transitions
      - Physique : Diagrammes de modules, Diagrammes de processus
    - Object Modeling Technique (OMT) de James Rumbaugh en 91 puis 94. 3 axes : Statique, Dynamique, Fonctionnel
    - Object-Oriented Software Engineering (OOSE) de Ivar Jacobson en 92, souvent appelée Objectory. Notion de Cas d'Utilisation.

Genèse d'UML : besoin d'unification

- Concepts OO proches mais notations graphiques différentes.
- La guerre des méthodes ne fait plus avancer la technologie des objets.
- Hors l'industrie a besoin de standards.
- Recherche d'un langage commun unique
  - Utilisable par toutes les méthodes
  - Adapté à toutes les phases du développement
  - Compatible avec toutes les techniques de réalisation

Genèse d'UML : Unification

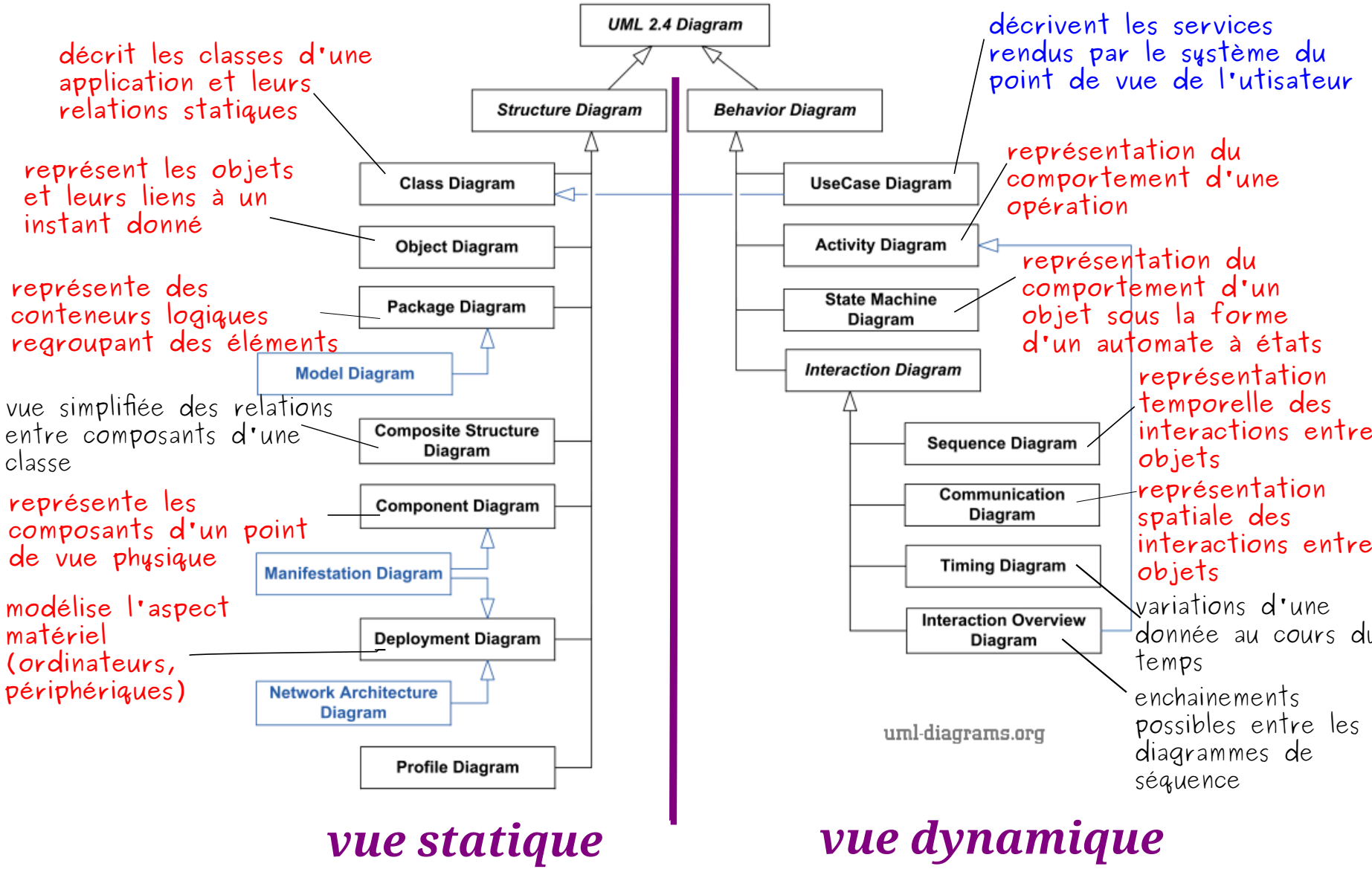
- Octobre 1995 : Unification de 2 méthodes (Méthode Unifiée v0.8).
- Fin 1995 : Jacobson (OOSE) les rejoint (UML 0.9).
- Consortium (IBM, Microsoft, Oracle, ...) s'y associent.
- Janvier 97 : Soumission à l'OMG de la version UML 1.0.
- Novembre 97 : UML1.1 adopté par OMG comme standard officiel
  - Notation unifiée pour toutes les méthodologies OO
- Version 2.0 en septembre 2004, Version 2.4.1 en août 2011.



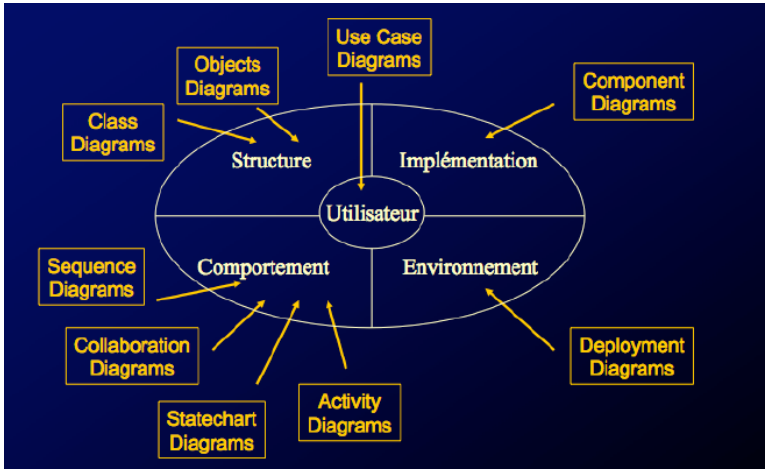
Les éléments organisationnels

- vues : décrivent un point de vue du système
- diagrammes : éléments graphiques qui représentent le problème et sa solution selon différents points de vue
- modèles d'éléments : briques de diagrammes représentant des notions

Les 14 diagrammes



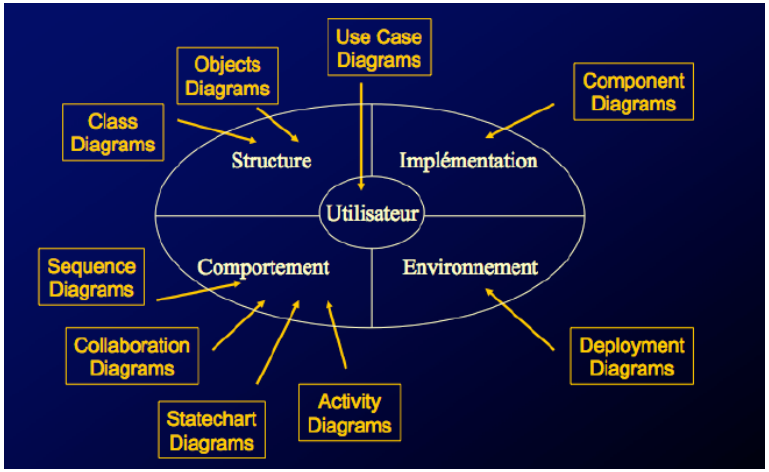
Diagrammes et vues d'architecture



Les diagrammes UML décrivent 4+1 vues d'architecture (4+1 est un vue modèle proposé par Philippe Kruchten en 1995) :

- vue utilisateur
  - buts et objectifs des clients du système, besoins requis par la solution
- vue structurelle
  - aspects statiques représentant la structure du problème
  - éléments de structure pour mettre en oeuvre une solution pour les besoins définis
- vue comportementale
  - aspects dynamiques du comportement du problème et de sa solution
  - interactions et collaborations entre les composants du système

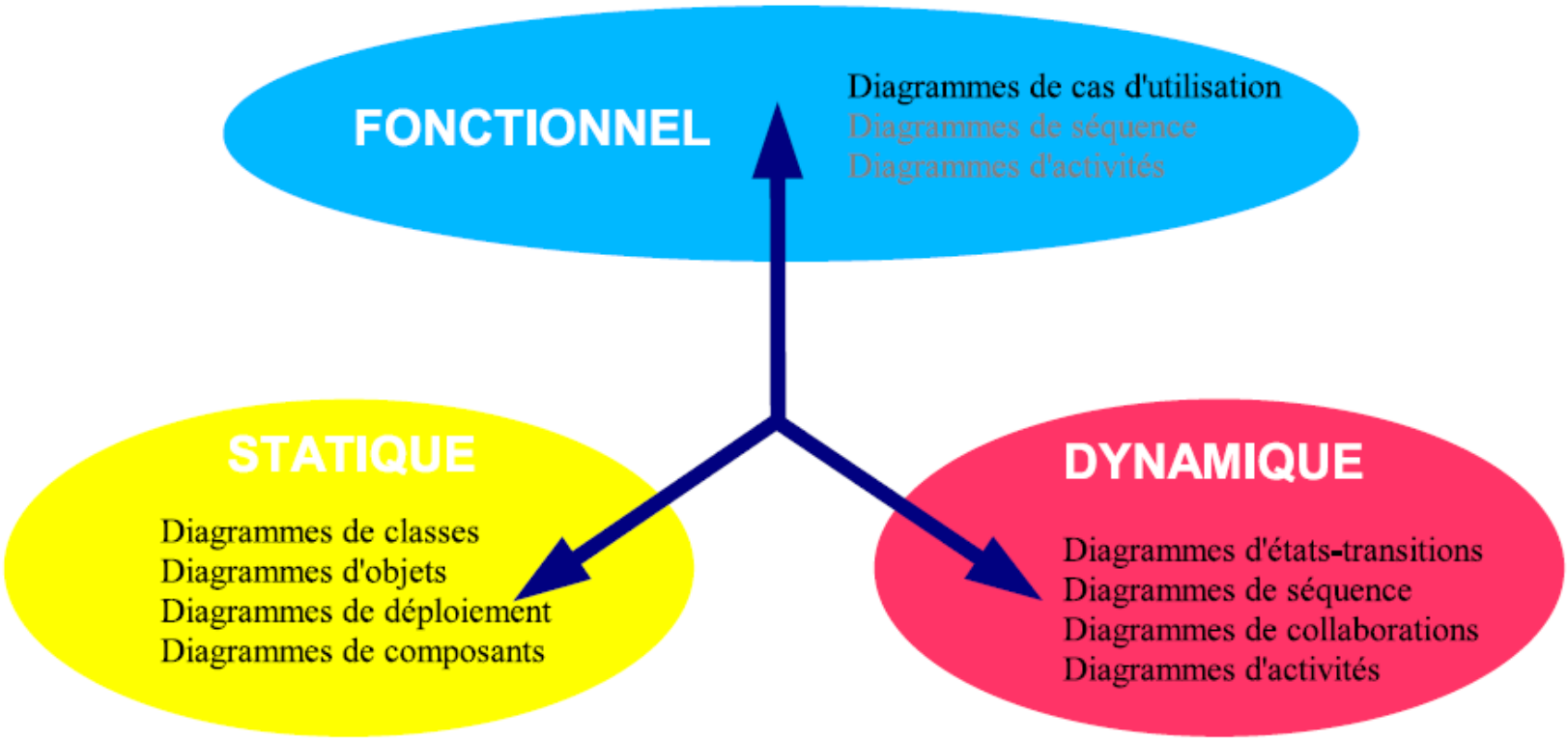
Diagrammes et vues d'architecture



Les diagrammes UML décrivent 4+1 vues d'architecture (4+1 est un vue modèle proposé par Philippe Kruchten en 1995) :

- vue implémentation
  - aspects de la structure et du comportement de la solution
- vue environnementale
  - aspects de la structure et du comportement du domaine dans lequel est réalisée la solution
  - topologie matérielle du système

Diagrammes selon les axes de modélisation

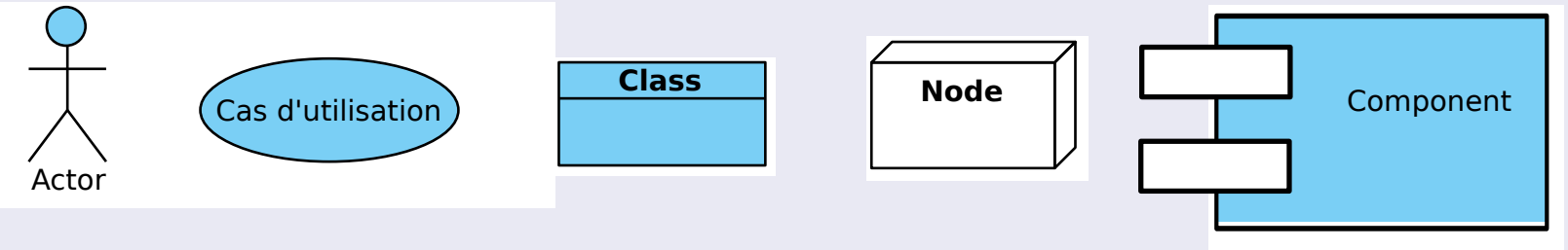


Plan

Notations communes

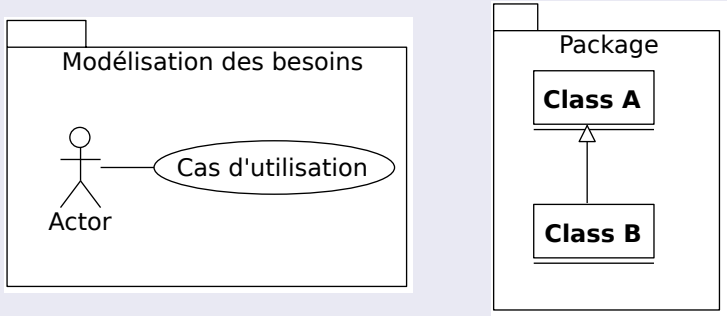
Classeur

Un classeur est un élément de modèle qui est doté d'une identité, possède des caractéristiques structurelles (attributs, participation à des relations) et comportementales (opérations).



Paquetage (*package*)

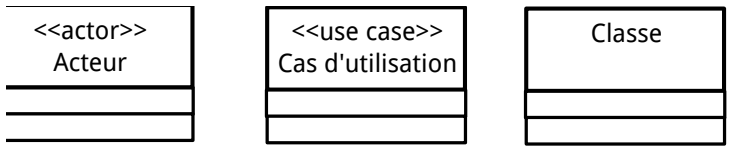
Un paquetage est un regroupement d'éléments de modèle ou de diagrammes.

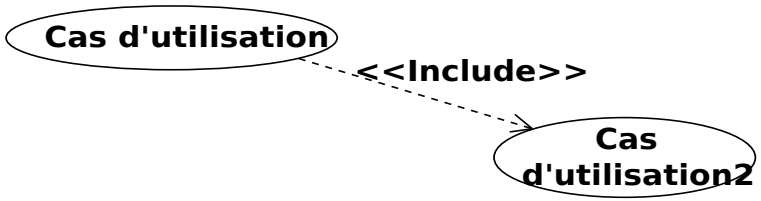


Notations communes

Stéréotype

- Annotation s'appliquant sur un élément de modèle
- Utilisation particulière d'éléments de modélisation avec interprétation (sémantique) particulière
- Modification de la signification d'un élément
- Notation : ≪**nomDuStéréotype**≫ avant le nom de l'élément auquel il s'applique ou **icône associée**
- Prédéfinis : ≪actor≫, ≪includes≫, ≪use case≫, ≪interface≫, ≪include≫ ... ≪class≫ stéréotype par défaut d'un classeur

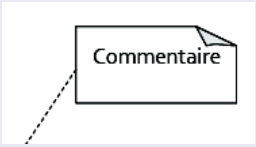




Notations communes

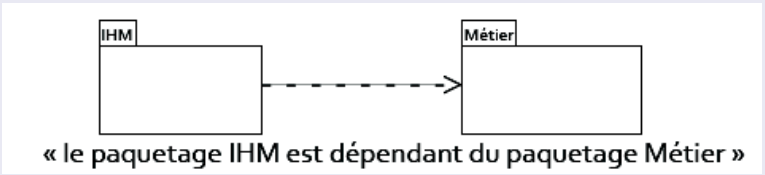
Commentaires

- Information textuelle (explication utile, observations, renvoi vers document), pas de contenu sémantique
- Notation :



Relations de dépendance

- Modification de la cible peut impliquer une modification de la source
- La source ≪dépend≫ de la cible
- Notation : `[source]--------------->[cible]`

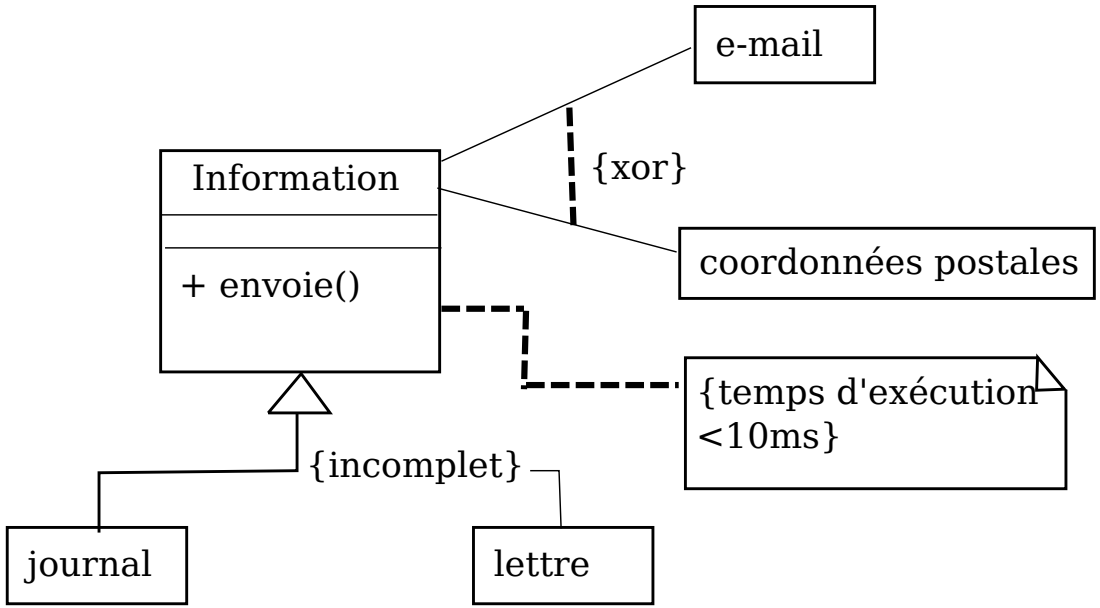


Toute modif dans paquetage `métier` peut avoir des conséquences sur paquetage `IHM`. Toute modif dans paquetage `IHM` n'a au vu de cette dépendance pas de conséquence sur paquetage `métier`.

- Nombreux stéréotypes prédéfinis : ≪extend≫, ≪include≫, ...

Contraintes

- Relation entre éléments de modélisation
- Définition de propriétés devant être vérifiées pour garantir la validité du système modélisé
- Notation : $- - - - - - - - - \{contrainte\}$
  3 types de contraintes :
  - Contraintes prédéfinies : $disjoint$, $overlapping$, xor, ...
  - Contraintes exprimées en langue naturelle (commentaires)
  - Contraintes exprimées avec OCL (Object Constraint Language)
- Stéréotypes : ≪précondition≫, ≪postcondition≫



Plan

Point de vue statique sur le système

Décrire la structure du système en termes de :

- Composants du système : Objets, Classes, Paquetages, Composants, ...
- Relations entre ces composants : Spécialisation, Association, Dépendance, ...
- Pas de facteur temps

Différents diagrammes statiques que nous allons voir

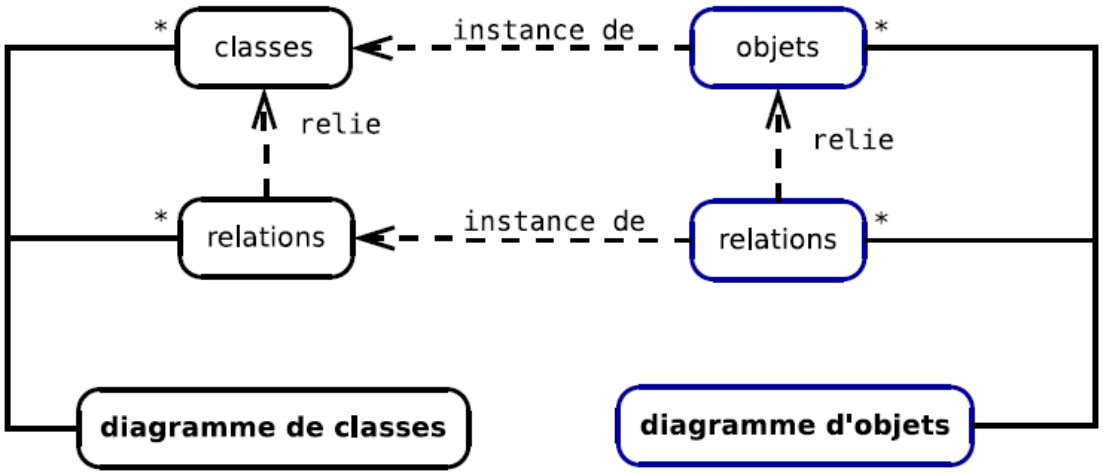
- Diagrammes d'objets (Cours)
- Diagrammes de classes (Cours + TD)
- Diagrammes de paquetages (Cours + TD)
- Diagrammes de composants (Cours)
- Diagrammes de déploiement (Cours)

Plan

Diagramme d'objets

Classes, objets et instances

- Classe = description formelle d'un ensemble d'objets ayant une sémantique/caractéristiques communes
- Objet ou instance de classe = concrétisation d'une classe
- Instance = toute concrétisation d'un concept abstrait
  - instance de classe : objet
  - instance d'association entre classes : lien



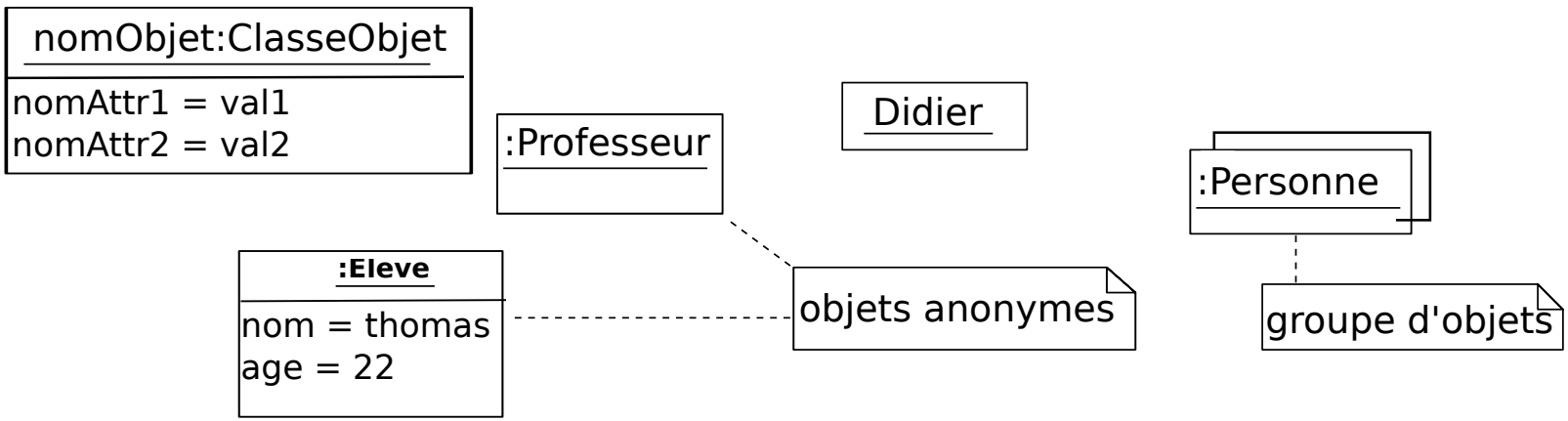
(extrait simplifié du méta-modèle d'UML)

Diagramme d'objets

Objectifs

- Représente les objets et leurs liens **à un instant donné**
- Modélise les **instances**

- Représentation UML des objets



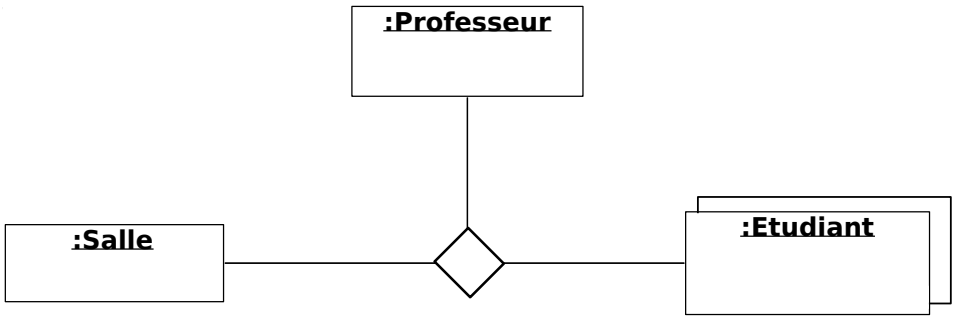
Communication entre objets

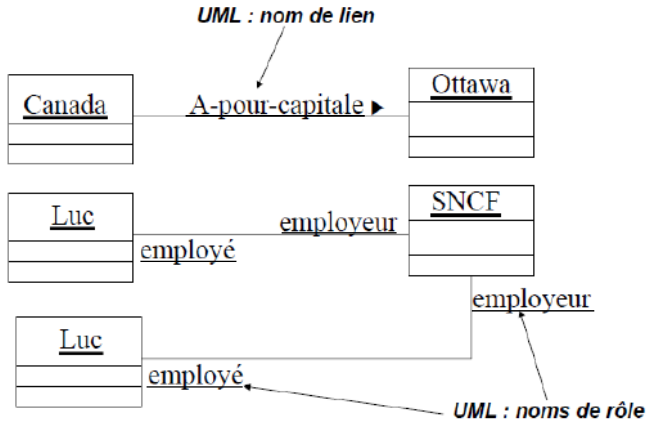
Liens entre objets

- Comportement global d'une application réside sur la communication entre les objets qui la composent
  - $\rightarrow$ réduction du couplage entre l'objet et l'environnement par envoi de messages entre objets qui se connaissent
- Unité de communication = message
- Lien entre objets
  - un objet connaît un autre objet $\rightarrow$ avoir une référence qui lui correspond (attributs, paramètres de méthodes, ...)
  - peut envoyer un message à l'autre objet

Représentation UML des liens

- Lien binaire : entre 2 objets
- Lien n-aire : entre n objets
- Possibilité de décorer les liens (nom, nom des rôles)
- La multiplicité se représente de manière explicite par les liens





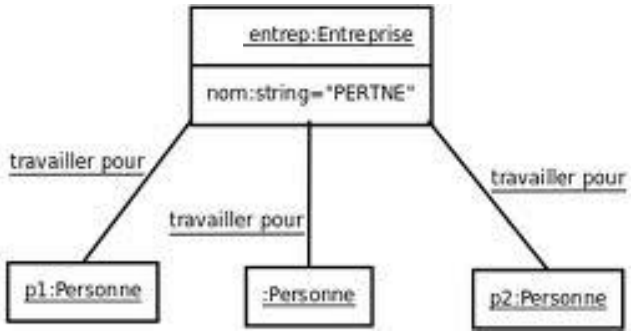


Diagramme d'objets

Utilisé pour

- Préciser une structure complexe trop difficile à comprendre avec un diagramme de classes
  - Illustrer un modèle de classes en montrant un exemple qui explique le modèle
  - Préciser certains aspects du système
  - Exprimer une exception en modélisant des cas particuliers ou des connaissances non généralisables
- Prendre une image (*snapshot*) d'un système à un moment donné
- Documenter des cas de test, analyser des exemples
- Montrer un contexte (collaborations sans messages)

Ne montre pas

- L'évolution du système dans le temps.

Exercice

Modéliser ceci par un diagramme d'objets :

- Pierre, Paul, Jacques, Marie et Anne sont étudiants au département Info.
- Robert et Suzie sont enseignants au département Info.
- Robert enseigne le C et le Java ; Suzie enseigne l'anglais et les maths.
- Pierre et Paul suivent les cours de Java ; Pierre suit le cours de C ; Marie et Anne suivent le cours d'anglais ; Anne suit le cours de maths.

Exercice

Plan

Diagramme de classes

Objectif

Décrire les classes d'une application et leurs relations statiques

Très nombreuses utilisations, à différents niveaux

- Diagrammes fondamentaux : les plus connus, les plus utilisés
- Permettent de modéliser plusieurs niveaux
  - conceptuel (domaine, analyse) (Modèle du domaine)
  - implémentation (code) (Modèle de conception)

Qu'est ce qu'une classe d'objets ?

Classe = regroupement d'objets similaires (appelés instances)

- Toutes les instances d'une classe ont les mêmes attributs et opérations
- **Abstraction** : factorisation des caractéristiques communes à une catégorie d'objets
- Une classe décrit une infinité d'instances

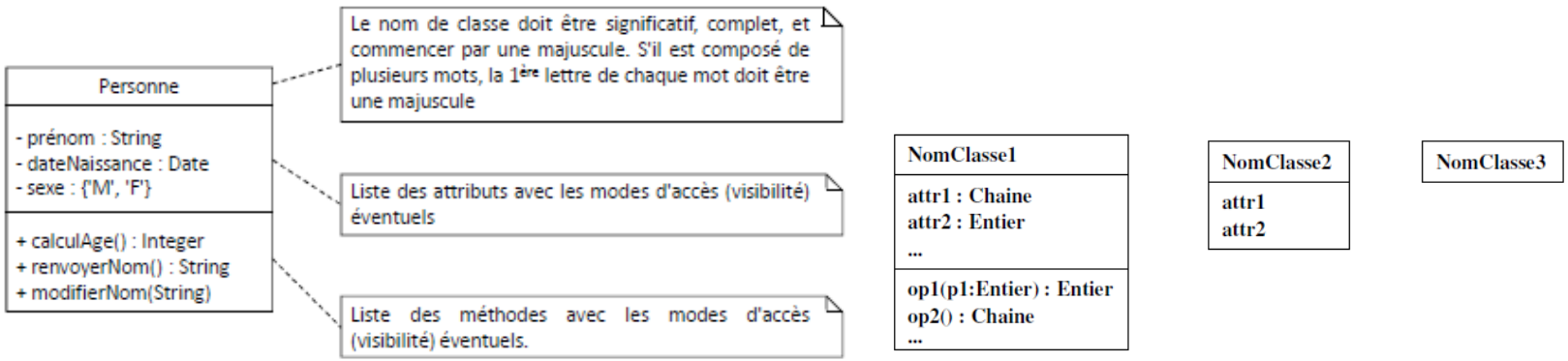
Représentation UML des classes

Rectangle composé de compartiments

- Compartiment 1 : Nom de la classe (commence par une majuscule, en gras)
- Compartiment 2 : Attributs
- Compartiment 3 : Opérations
- Possibilité d'ajouter des compartiments (exceptions, ...)

Différents niveaux de détail possibles

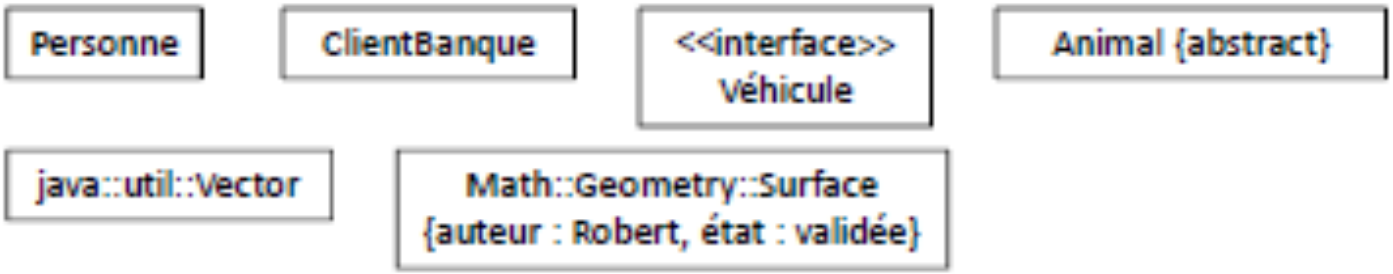
Possibilité d'omettre attributs et/ou opérations



Nom de la classe

- Tout est facultatif sauf le nom de la classe

<< stéréotype >> NomPaquetage1:: ... ::NomPaquetageN::NomClasse {abstract} {auteur : valeur, état : valeur, ...}



Attributs et Opérations

Encapsulation et visibilité des attributs et des opérations

Mécanisme consistant à rassembler les données et les méthodes au sein d'une structure en cachant l'implémentation de l'objet. 4 niveaux

- Public : attributs accessibles et modifiables par n'importe quelle autre classe, opérations utilisables par n'importe quelle autre classe,
- Privé : attributs inaccessibles pour toute autre classe, opérations inutilisables pour toute autre classe
- Protégé : attributs accessibles et modifiables uniquement par les classes dérivées, opérations utilisables uniquement par les classes dérivées.
- Aucun caractère, ni mot-clé : propriété ou classe visible uniquement dans le paquetage où la classe est définie

Attributs/Opérations de classe

Attributs (ou opérations) communs à l'ensemble des instances d'une classe (*static*)

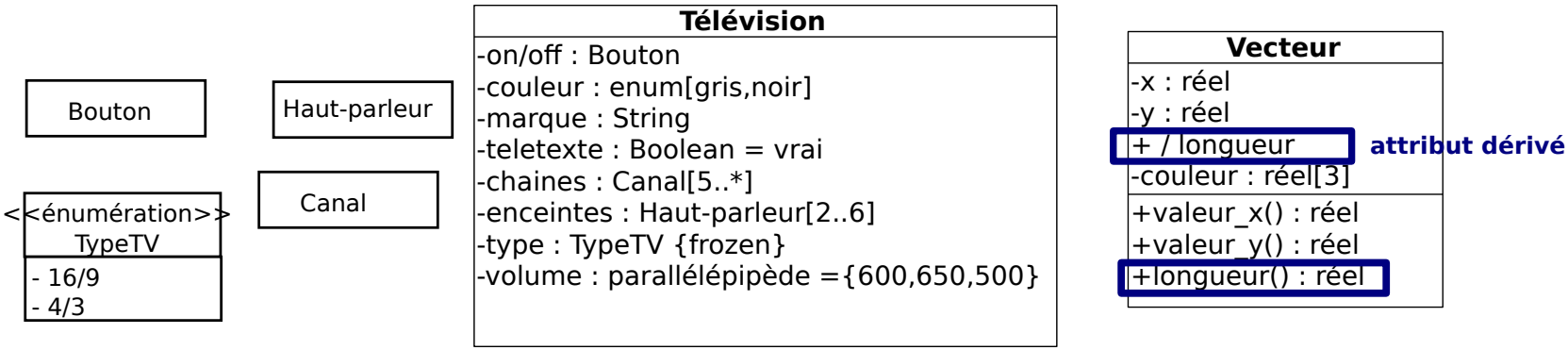
Représentation UML des attributs

- Format de description d'un attribut :

visibilité nom : type [multiplicité] = valeur_initiale {propriétés}

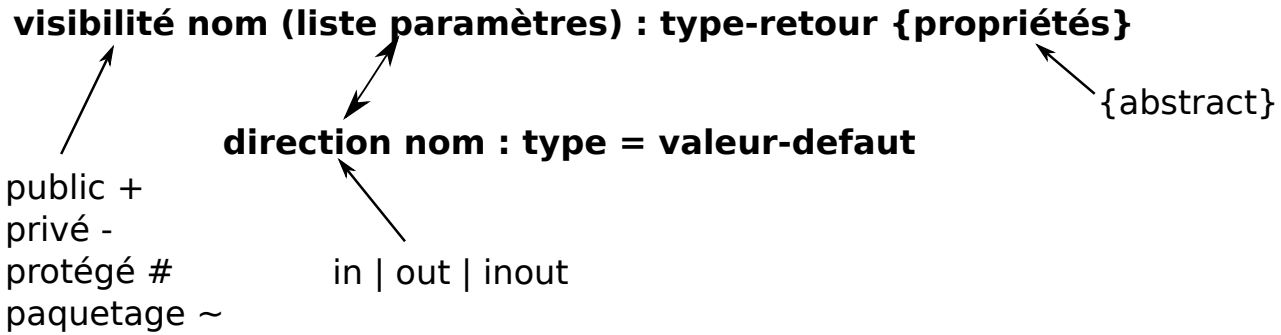
- public + / privé - / protégé # / paquetage ~
- facultatif mais impératif pour l'implémentation
- facultatif ex. couleurs : Saturation[3] points: Points[2..*]
- facultatif
- facultatif ex. {frozen} mise à jour interdite

- Attributs de classe (statiques) soulignés
- Attributs dérivés (calculés) précédés de " /"
- Enumération : stéréotype ≪enumeration≫

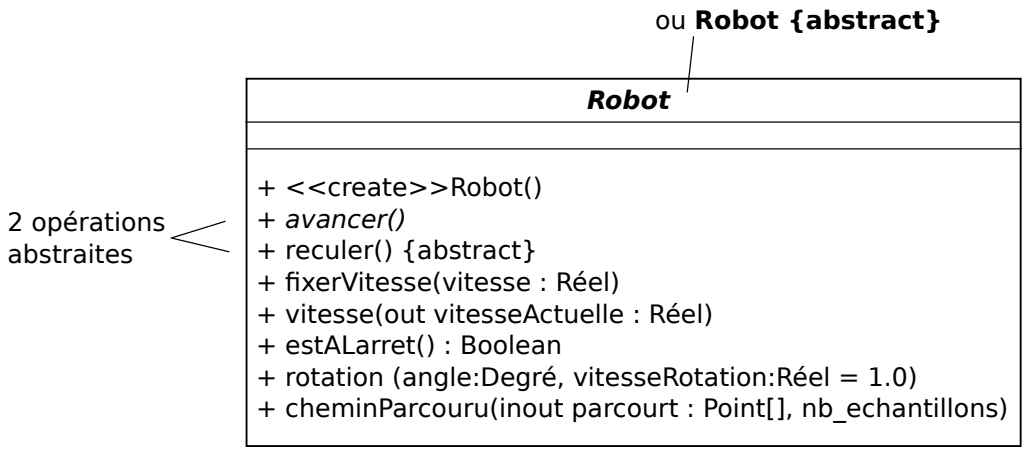


Représentation UML des opérations

- Format de description d'une opération :



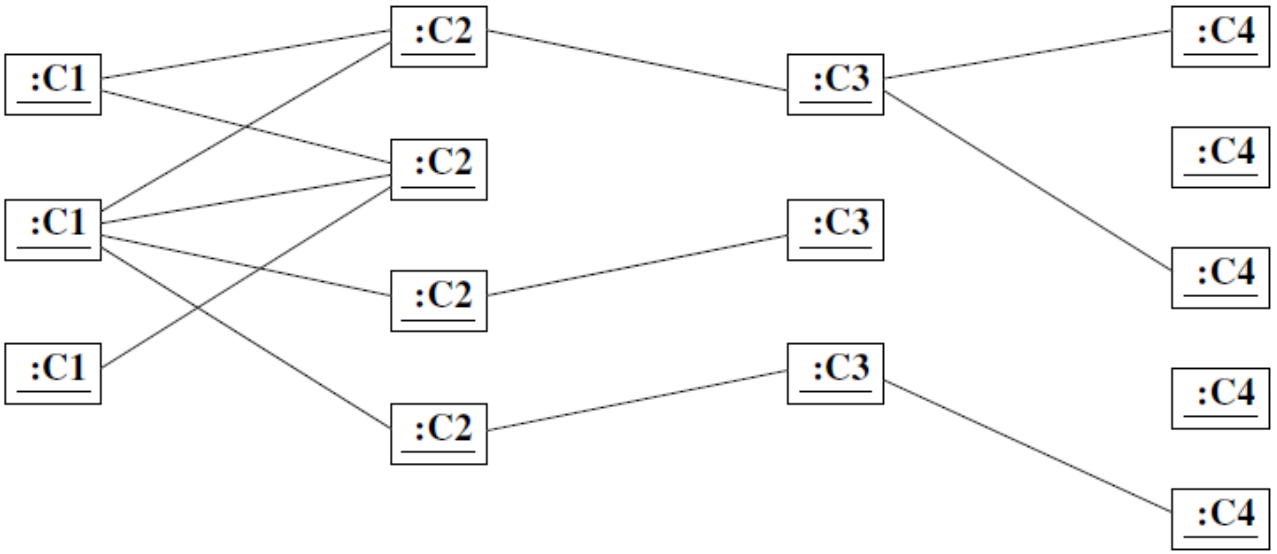
- *opérations abstraites* (non implémentées) / opérations de classe (statiques)
- Propriétés : {abstract}, {query} (l'opération n'altère pas l'état de l'instance concernée), pré- et post-conditions, description du contenu (commentaires, OCL)
- Stéréotypes d'opérations : constructeur ≪create≫, destructeur ≪destroy≫



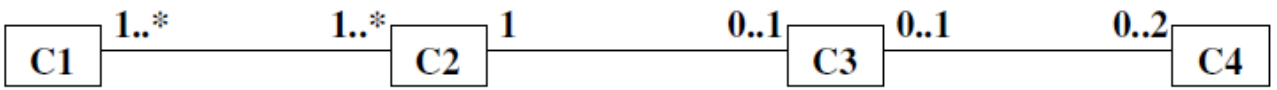
Associations entre classes

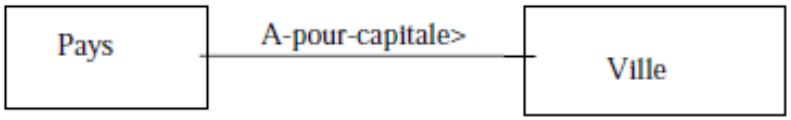
Abstraction des relations définies par les liens entre objets

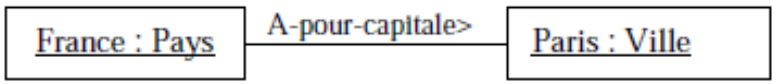
Liens entre objets :



Associations entre classes d'objets :



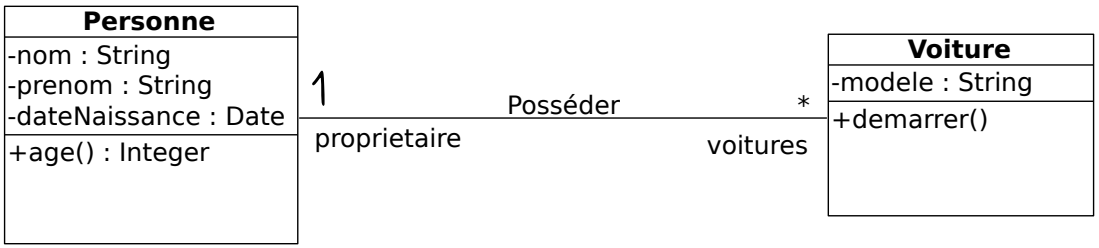


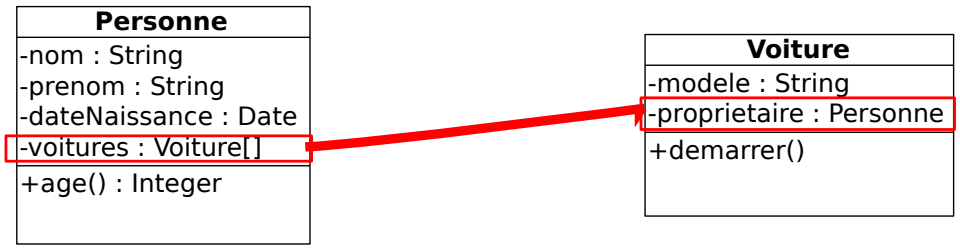


Associations entre classes

Notion d'associations

- Relation entre deux classes (binaire) ou plus (n-aire).
- Décrit les connexions structurelles entre les instances des classes associées
- 2 façons de modéliser une association

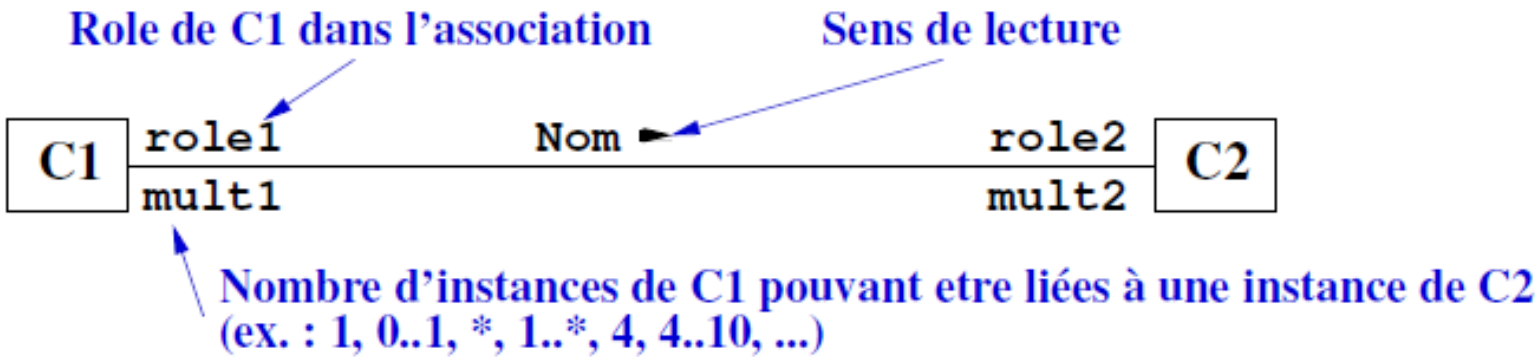




- Les associations sont utilisées pour matérialiser les relations entre éléments qui font partie de la modélisation.
- Les attributs sont utilisés pour modéliser des types de données dépourvus d'identité.

Associations entre classes

Nom, rôle et multiplicité

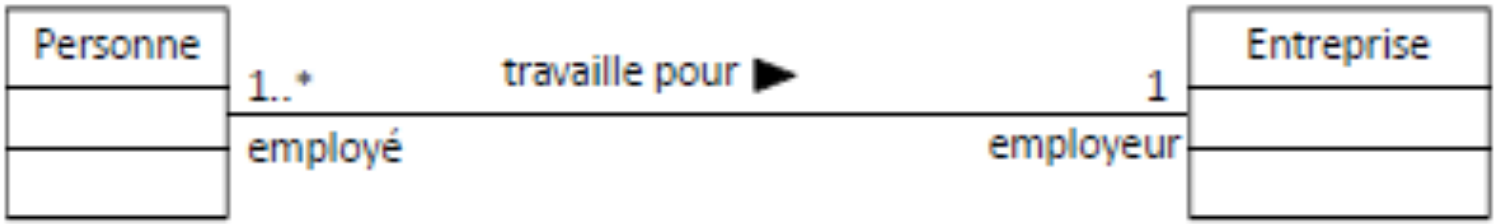


- Nom : forme verbale
- Sens de lecture de l'association indiqué éventuellement par une flèche
- Multiplicité : nombre (ou intervalle de nombres) d'instance(s) que l'association peut impliquer.
  - exactement un se note 1 ou $1..1$
  - plusieurs se note * ou $0..*$
  - au moins un se note $1..*$
  - de un à six se note $1..6$
- Rôle : forme nominale décrivant le statut de la classe dans l'association. Peut contenir un mode d'accès.

Associations entre classes

Nom, rôle et multiplicité

- Exemple : *une personne travaille pour une et une seule entreprise. L'entreprise emploie au moins une personne. L'entreprise est l'employeur des personnes qui travaillent pour elle et une personne à un statut d'employé dans l'entreprise.*



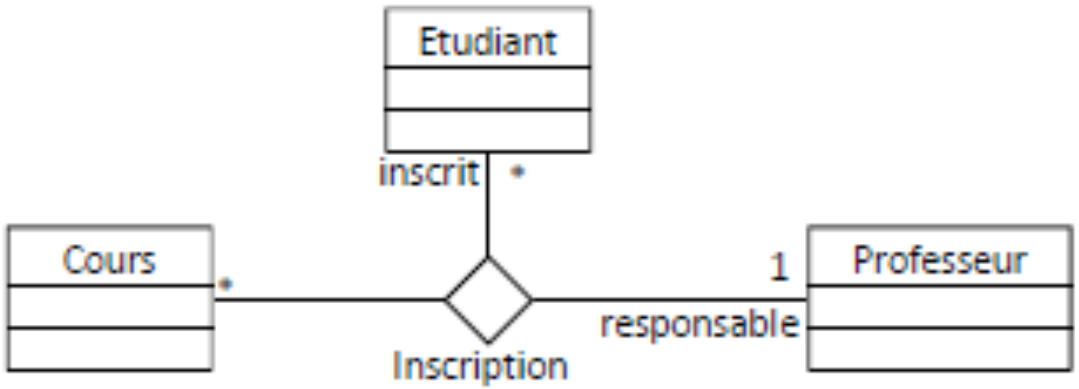
Incidence sur l'implémentation :

- La classe `Personne` a un attribut de nom `employeur`
  $\rightarrow$ Type = `Entreprise`
- La classe `Entreprise` a un attribut de nom `employé`
  $\rightarrow$ Type =collection de `Personne`

Associations entre classes

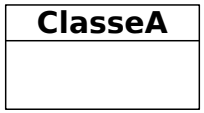
Associations n-aire

- Association entre au moins trois classes
- Chaque instance de l'association est un tuple de valeurs provenant chacune de leurs classes respectives
- Exemple : *un étudiant suit le cours d'un professeur responsable pendant un semestre. Il peut suivre plusieurs cours d'un même professeur responsable*
- Pour une paire (cours, étudiant), il n'existe qu'un professeur. Pour une paire (étudiant, professeur), il existe plusieurs cours. Pour une paire (cours, professeur), il existe plusieurs étudiants



Exercice 1 : Instanciation de diagrammes de classe

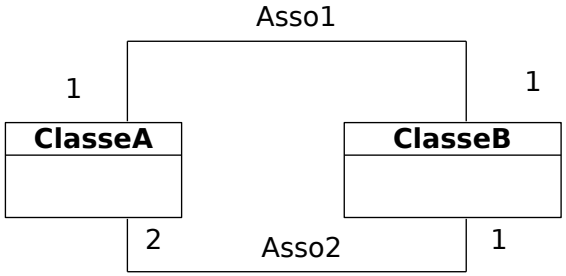
Proposer le plus petit diagramme d'objets respectant le diagramme de classes :

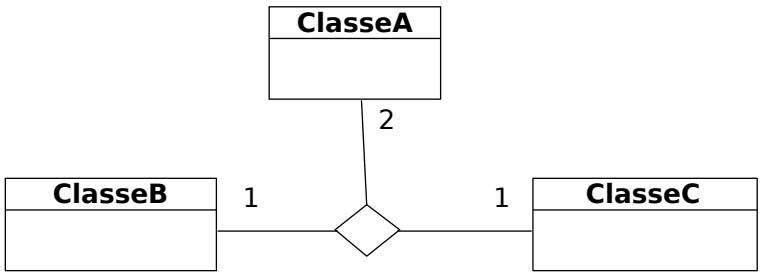
- 1- 

et composé d'au moins 3 objets
- 2- 

et composé d'au moins 3 objets
- 3- 

et composé d'au moins 2 objets de la ClasseA
- 4- 

et composé d'au moins 2 objets de la ClasseB
- 5- 

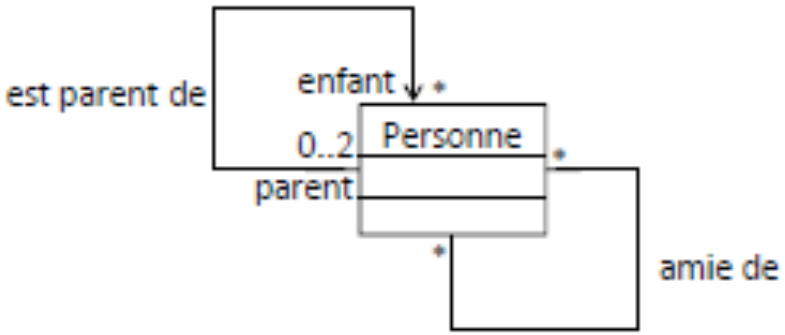
et composé d'au moins 1 objet
- 6- 

et composé d'au moins 1 objet

Associations entre classes

Réflexivité, symétrie/asymétrie

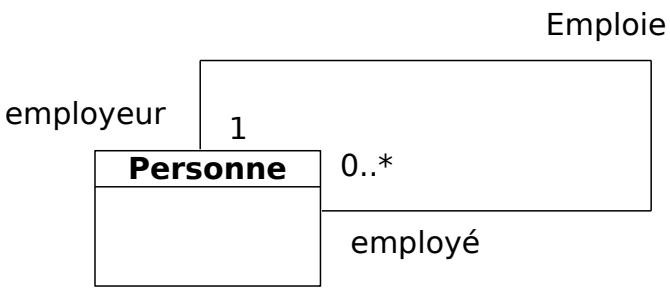
- Exemple : *une personne peut être parent d'autres personnes et enfant d'au plus deux personnes connues. Deux personnes peuvent être amies (on considère que l'amitié est réciproque).*
- Les associations de parenté et d'amitié sont réflexives : elles associent la classe Personne avec elle-même
- La relation de parenté est asymétrique, donc orientée (si une instance A est parent d'une instance B, l'inverse ne peut pas être possible simultanément)
- La relation d'amitié est symétrique (=non-orientée). Il est inutile de spécifier des rôles dans ce cas.

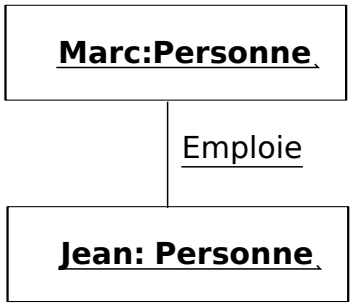


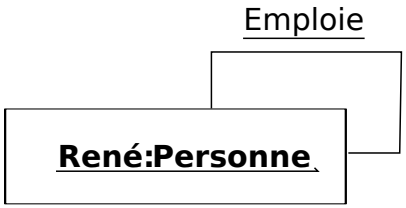
Associations entre classes

Réflexivité, symétrie/asymétrie

- Une instance de la classe impliquée peut être reliée à elle-même ou à d'autres instances de cette même classe.





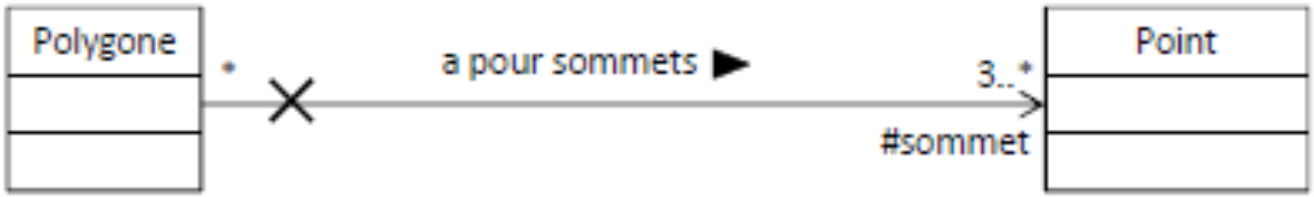


Associations entre classes
Navigabilité

- Indique s'il est possible de traverser une association
- Si un lien est naviguable d'un objet A vers un objet B, alors A a connaissance de B et peut solliciter l'interface de B
- Par défaut : Navigabilité dans les deux sens
- Exemple : *un polygone est défini par au moins 3 points jouant le rôle de sommets. Un point peut être sommet de plusieurs polygones. Les sommets du polygone ne sont accessibles que par la classe et ses descendants. On considère qu'il est inutile que les points aient un lien vers les polygones dans lesquels ils sont sommets.*

Associations entre classes
Navigabilité

- Indique s'il est possible de traverser une association
- Si un lien est naviguable d'un objet A vers un objet B, alors A a connaissance de B et peut solliciter l'interface de B
- Par défaut : Navigabilité dans les deux sens
- Exemple : *un polygone est défini par au moins 3 points jouant le rôle de sommets. Un point peut être sommet de plusieurs polygones. Les sommets du polygone ne sont accessibles que par la classe et ses descendants. On considère qu'il est inutile que les points aient un lien vers les polygones dans lesquels ils sont sommets.*

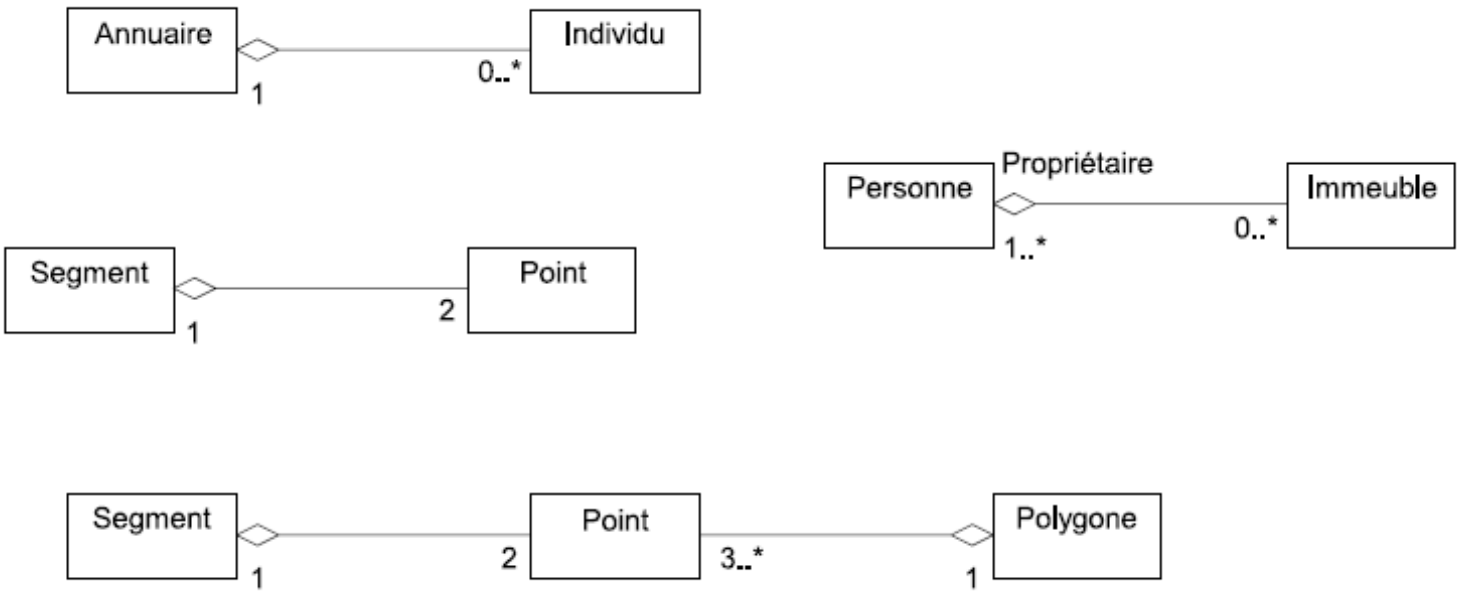


- Navigation possible uniquement du polygone vers les points : un polygone connait les points qui lui servent de sommets, mais un point ne connait pas les polygones dans lesquels il est sommet
- Incidence sur l'implémentation : la classe `Polygone` contiendra une collection de `Point`. La classe `Point` ne contiendra aucun attribut de type `Polygone`.

Associations particulières : agrégation

Agrégation

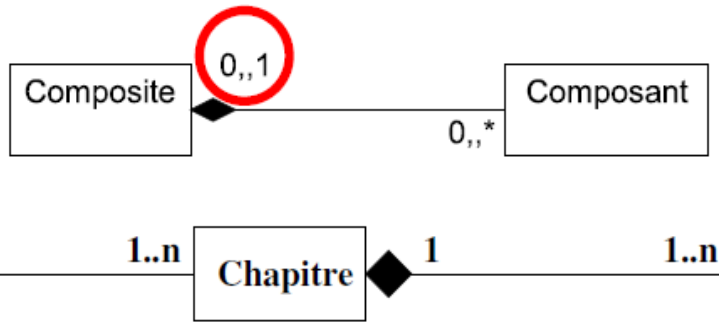
- Modéliser regroupement de parties dans un tout
- Association non-symétrique
  - Relation de dominance et de subordination
  - Une classe **fait partie** d'une autre classe
  - Une action sur une classe implique une action sur une autre classe
- Une classe peut appartenir à plusieurs agrégats

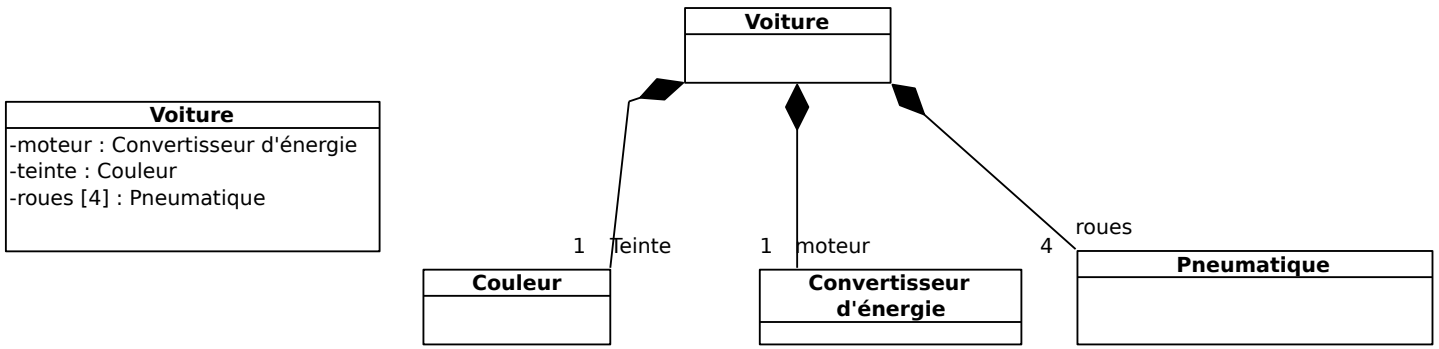


Associations particulières : composition

Composition

- Agrégation forte
- Contenance structurelle : Création/Copie/Destruction du composite → Création/Copie/Destruction de ses composants
- Un composant appartient à au plus un composite (mult. côté conteneur max 1)





Composition, Agrégation et Association

- Quelques questions à se poser :
  - assymétrie et lien de subordination entre instances des deux classes ($\rightarrow$ agrégation/composition) ou indépendance des objets ($\rightarrow$ association) ?
  - Propagation d'attributs ou d'opérations du tout vers les parties ? $\rightarrow$ agrégation/composition
  - Création et destruction des parties avec le tout ? $\rightarrow$ composition
- Remarques importantes :
  - Dans le doute, toujours utiliser une association (c'est la moins contrainte)
  - Choix entre composition et agrégation peut être laissé à la phase de conception

Exercice 2 : Identification des classes et relations

- Une voiture est caractérisée par une marque, un modèle et une motorisation. La marque d'une voiture est forcément l'une des suivantes : Citroën, Fiat, Ford, Nissan, Peugeot, Renault, Toyota ou Volkswagen. Le moteur d'une voiture est caractérisé par une désignation et une puissance. Une voiture possède 4 roues et le moteur actionne deux de ces roues. Modéliser cette situation à l'aide d'un diagramme de classes.
- Proposer un diagramme d'objets compatible avec ce diagramme de classes et comportant un seul objet `Voiture`.

Exercice 2 : Identification des classes et relations

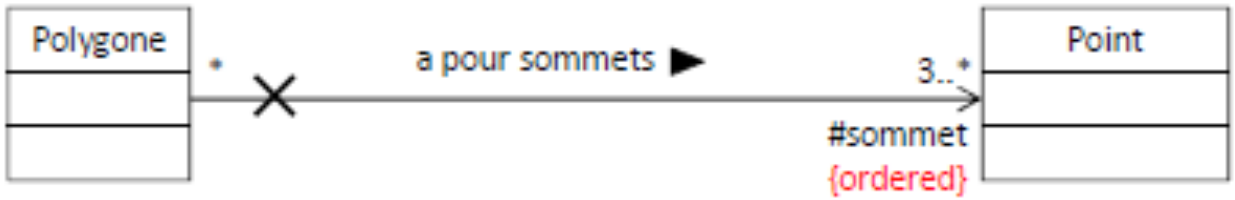
Exercice 2 : Identification des classes et relations

Associations : propriétés et contraintes

Contraintes prédéfinies (Utilisation du langage OCL)

- Sur une extrémité : relation d'ordre {`ordered`}, ...
- Entre 2 associations : inclusion {`subset`}, exclusion {`xor`}, ...

- Exemple 1 : *les sommets dans un polygone sont ordonnés :*

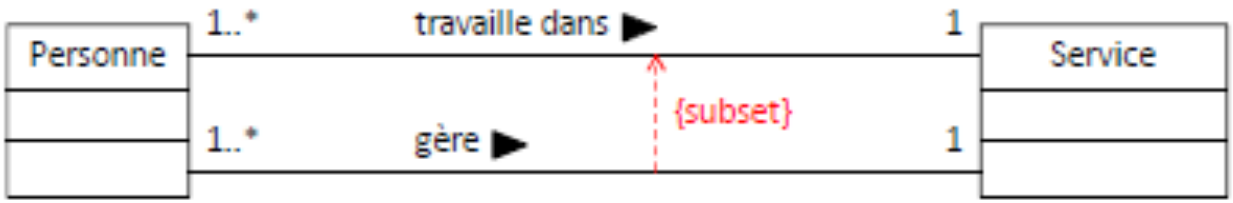


Associations : propriétés et contraintes

Contraintes prédéfinies (Utilisation du langage OCL)

- Sur une extrémité : relation d'ordre {`ordered`}, ...
- Entre 2 associations : inclusion {`subset`}, exclusion {`xor`}, ...

- Exemple 2 : *une personne travaille dans un service, et elle peut, en plus, gérer ce service. Plusieurs employés peuvent cogérer un service.*

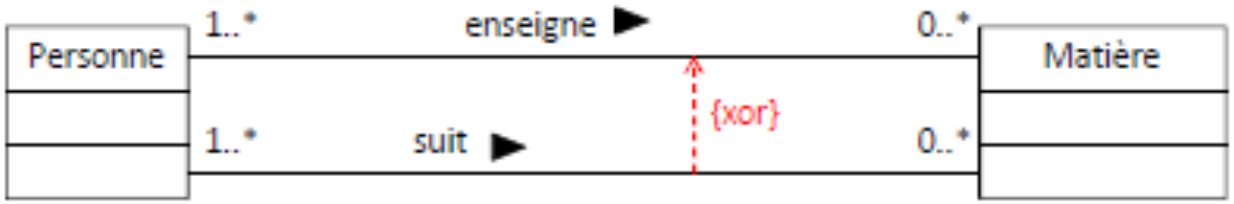


Associations : propriétés et contraintes

Contraintes prédéfinies (Utilisation du langage OCL)

- Sur une extrémité : relation d'ordre {`ordered`}, ...
- Entre 2 associations : inclusion {`subset`}, exclusion {`xor`}, ...

- Exemple 3 : *une personne (sans distinction de classe entre étudiant et enseignant) est associée à une matière, soit parce qu'elle l'enseigne, soit parce qu'elle la suit (mais jamais les deux simultanément)*



Associations : propriétés et contraintes

Contraintes prédéfinies (Utilisation du langage OCL)

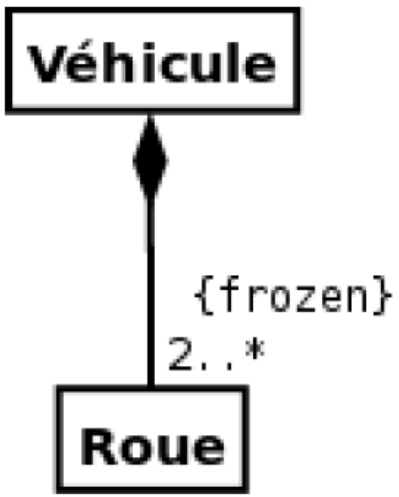
- Sur une extrémité : relation d'ordre {`ordered`}, ...
- Entre 2 associations : inclusion {`subset`}, exclusion {`xor`}, ...

- Exemple 4 : *un véhicule a des attributs propres, entre autres sa charge, et est d'une certaine catégorie. La charge maximale autorisée pour un véhicule dépend uniquement de la catégorie, pas du véhicule lui-même*

Associations : propriétés et contraintes

Propriétés sur extrémités d'associations

- {variable} : instance modifiable (par défaut)
- {frozen} : instance non modifiable
- {addOnly} : instances ajoutables mais non retirables (si mult.>1)



La contrainte {frozen} précise que le nombre de roues d'un véhicule ne peut pas varier.

Exercice3 : Associations : propriétés et contraintes

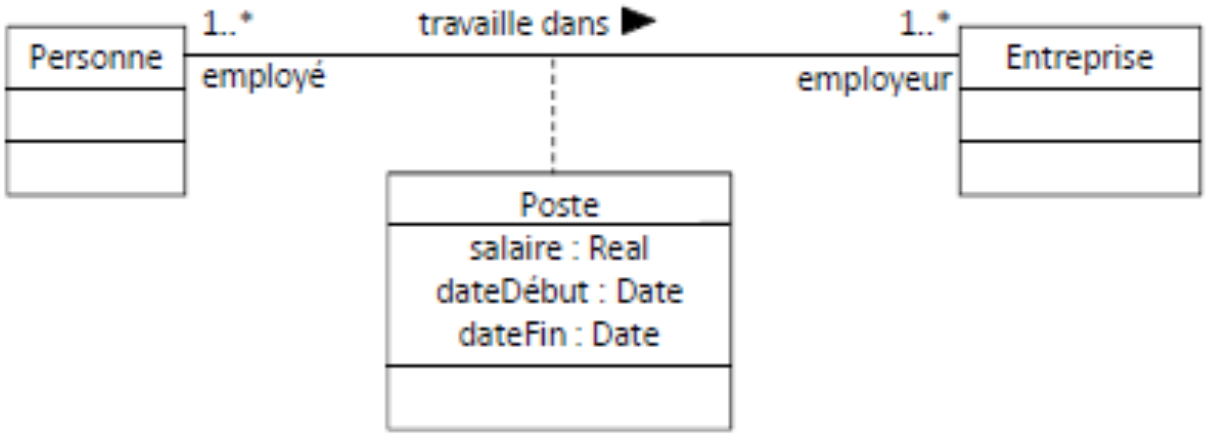
Modéliser avec un diagramme de classes les énoncés suivants (il y a toujours au moins un classe `Personne`) :

- Un comité est composé d'au moins 2 personnes qui sont ses membres. L'unique président du comité est également un membre du comité.
- Un hotel a plusieurs clients et un ou plusieurs employés. Les employés de l'hôtel n'ont pas le droit de prendre une chambre dans ce même hôtel.
- Une personne est née dans un pays et cela ne peut être modifié. Une personne a visité un certain nombre de pays, dans un ordre donné, et le nombre de pays visités ne peut que croître. Une personne aimerait encore visiter toute une liste de pays, et selon un ordre de préférence.

Exercice3 : Associations : propriétés et contraintes

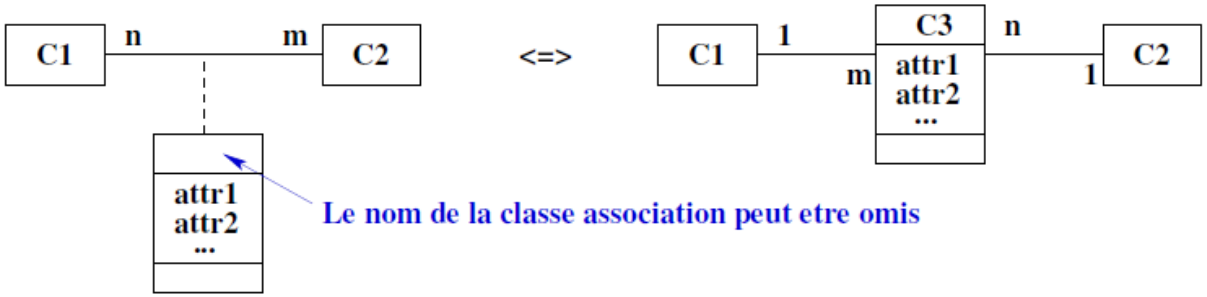
Classes-associations

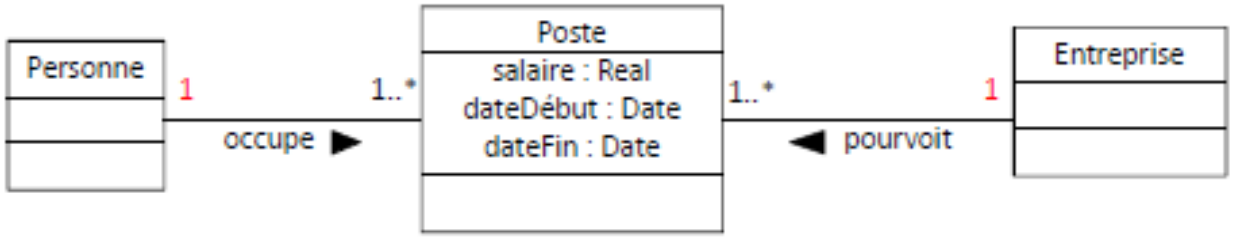
- Une association peut être raffinée et avoir ses propres propriétés, qui ne sont disponibles dans aucune des classes qu'elle lie.
- Une association peut être représentée par une classe pour ajouter attributs et opérations à des associations
- Exemple : *une personne à travaillé dans des entreprises, à des périodes données et avec un certain salaire.*



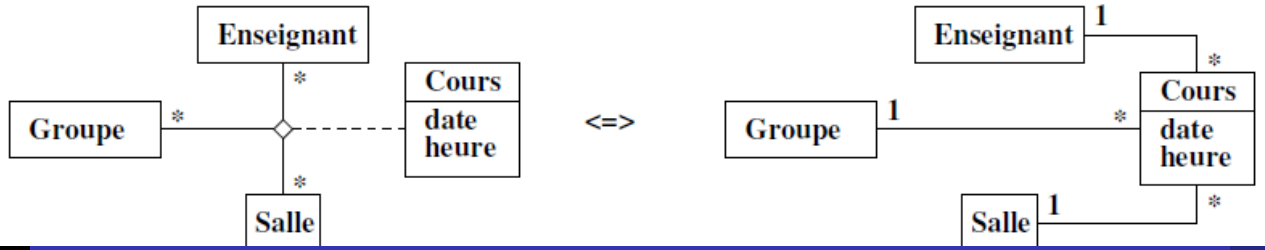
Classes-associations

- Une classe-association peut être remplacée par une classe intermédiaire qui sert de pivot (on sépare la relation initiale : attention au changement de multiplicité par rapport à la version avec classe-association)



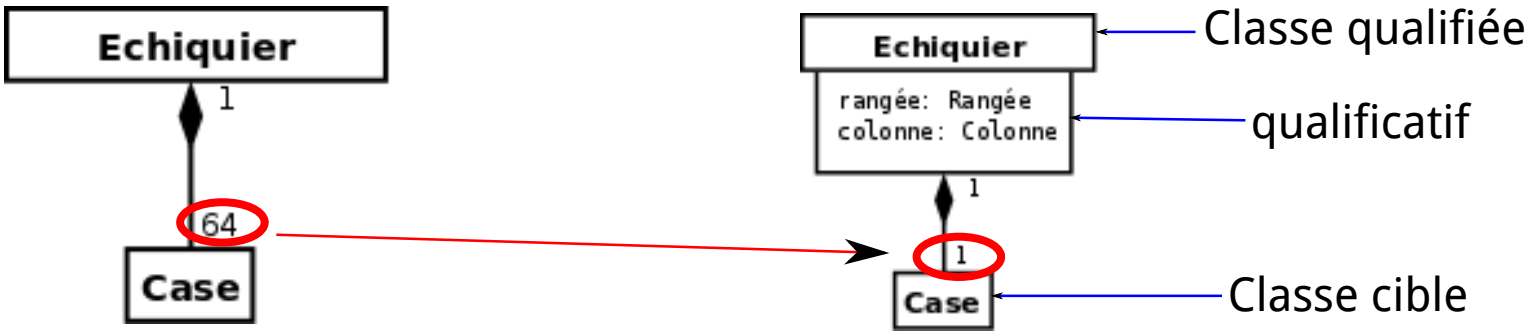


- Souvent lorsque relation de plusieurs à plusieurs : réduction des associations n-aires



Associations qualifiées

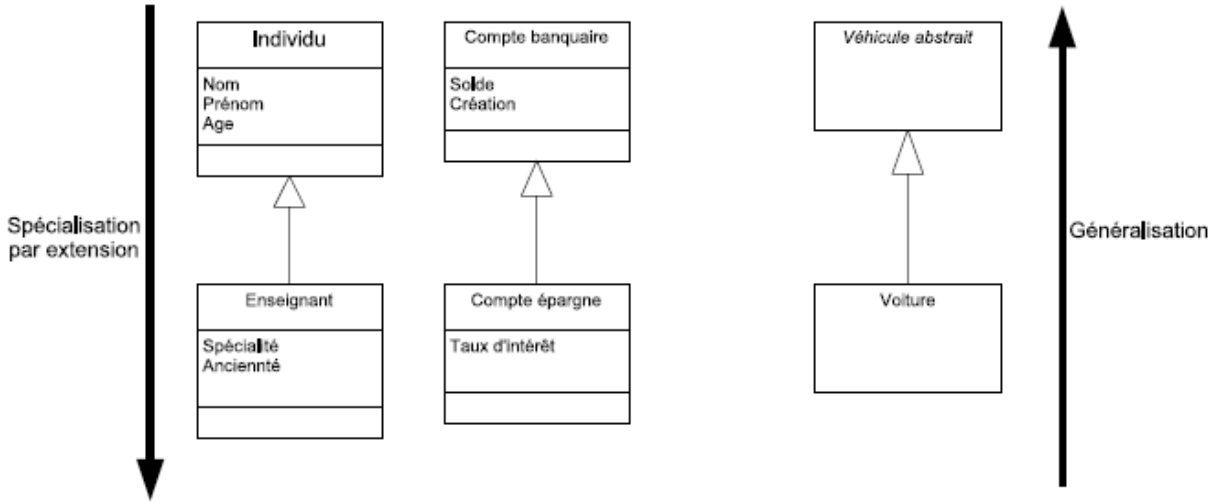
- Parfois préférable de restreindre la portée de l'association à quelques éléments ciblés (comme un ou plusieurs attributs) de la classe : **qualificatif**
- Un qualificatif permet de sélectionner un sous-ensemble d'instances : **objets cibles**
- Un qualificatif agit toujours sur une association dont la multiplicité est plusieurs (avant que l'association ne soit qualifiée) du côté cible.



Généralisation/Spécialisation

Deux interprétations

- Niveau conceptuel
  - Relation transitive, non réflexive, et non symétrique
  - Un concept est plus général qu'un autre : hiérarchie de classes
- Niveau implémentation : Héritage
  - La sous-classe hérite des attributs et méthodes de sa super classe
  - La sous-classe **est une sorte de** la super classe : Toute instance de la sous-classe est instance de la super classe
  - Ajout d'éléments propres, redéfinition

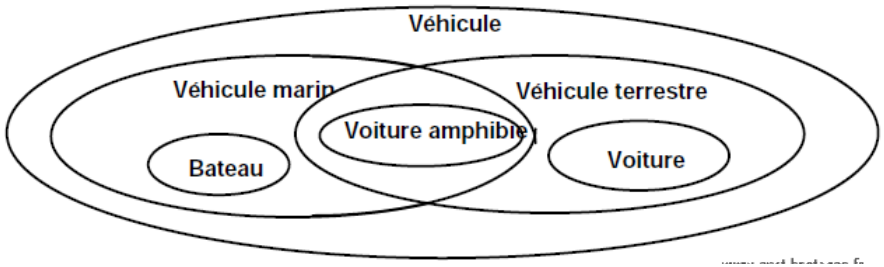


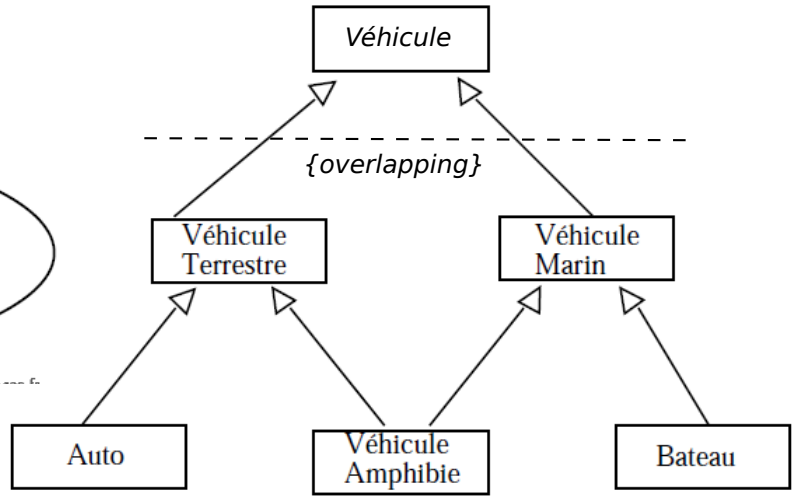
Généralisation/Spécialisation

Contraintes

Les seules contraintes pré-définies en UML pour la généralisation sont :

- {disjoint} (par défaut) = les sous-classes n'ont aucune instance en commun
  {overlapping} = les sous-classes peuvent avoir une ou plusieurs instances en commun





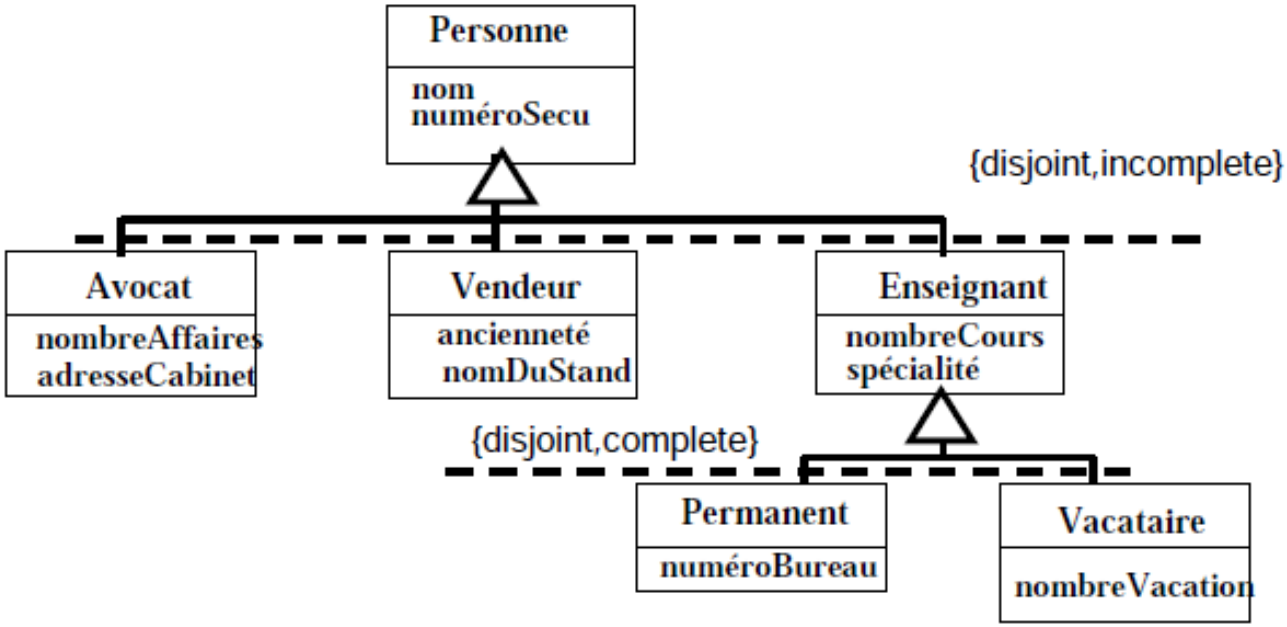
{overlapping} précise ici qu'il existe des véhicules amphibie qui sont issus d'un croisement des sous-classes de véhicule.

Généralisation/Spécialisation

Contraintes

Les seules contraintes pré-définies en UML pour la généralisation sont :

- {complete} (liste exhaustive de classe) / {incomplete} (les sous-classes spécifiées ne couvrent pas la super-classe)



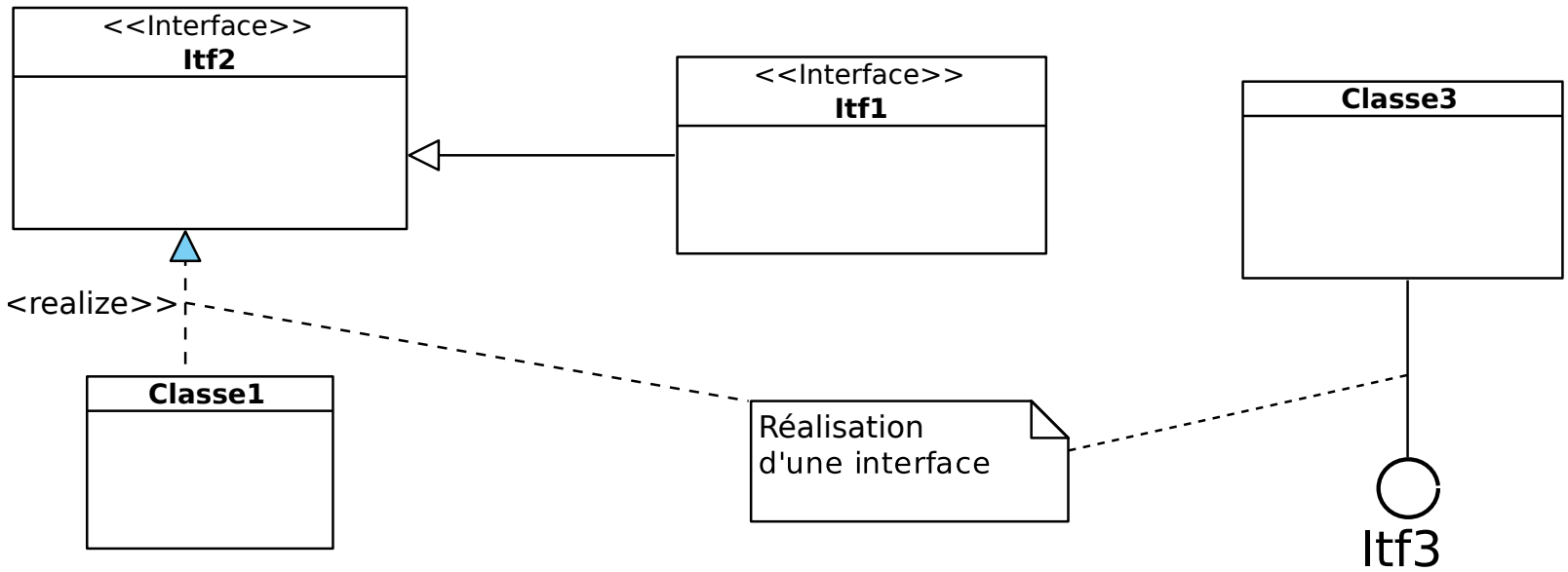
{incomplete} exprime l'idée que des instances de la classe Personne sont des avocats, des vendeurs, des enseignants ou d'autres personnes ... {complete} exprime l'idée qu'un enseignant ne peut être autre chose qu'un permanent ou vacataire.

Interface

Qu'est-ce qu'une interface ?

Classe sans attributs dont toutes les opérations sont abstraites (classe abstraite pure)

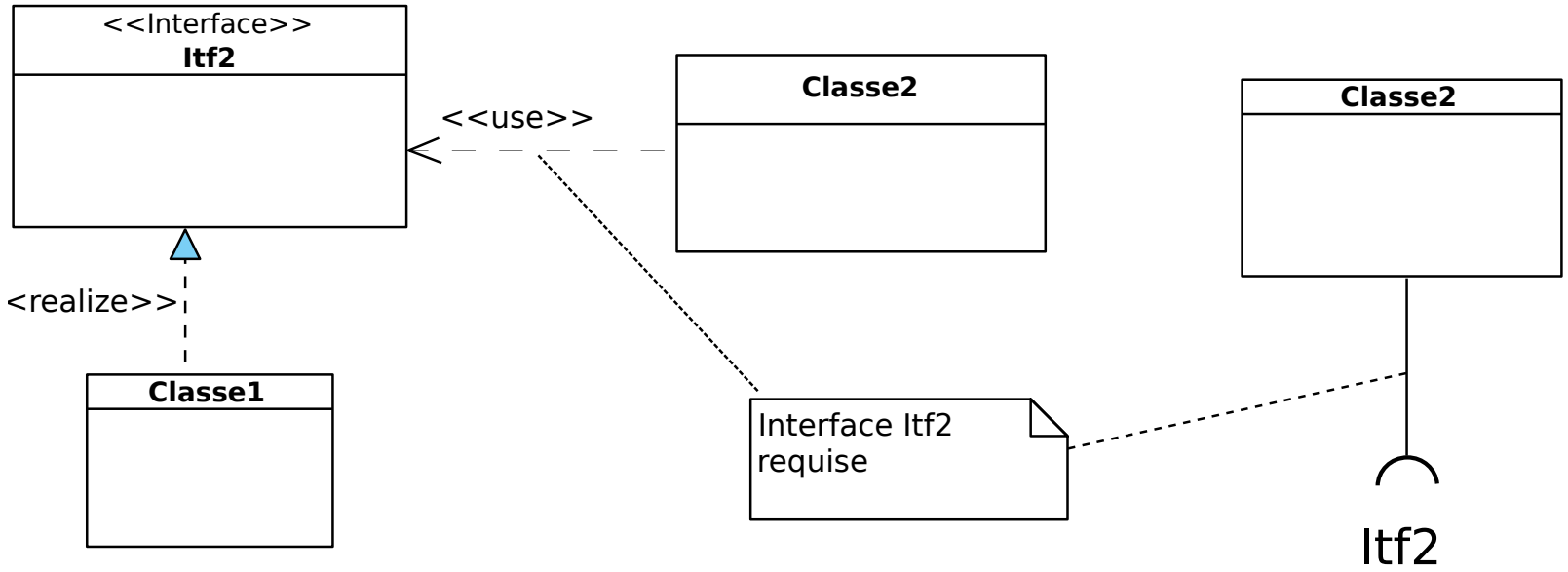
- Liste de services, savoir faire
- Ne peut être instanciée
- Doit être réalisée (implémentée) par des classes non abstraites
- Peut hériter d'une autre interface



Stéréotype `<<realize>>` facultatif.

Interface

Une classe peut aussi simplement dépendre d'une interface (interface requise).



Réalisation

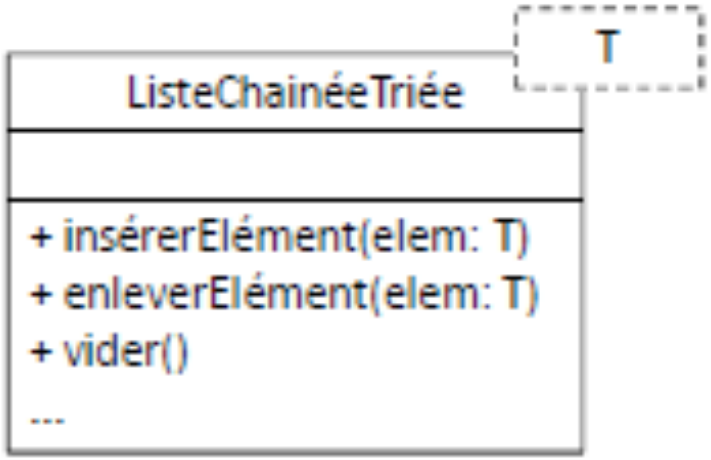
```
public class Classe1
    implements Itf2 {
  ...
}
```

Utilisation

```
public class Classe2 {
   public Classe2(Itf2 iface)
       { // public constructor
     ...
  }
```

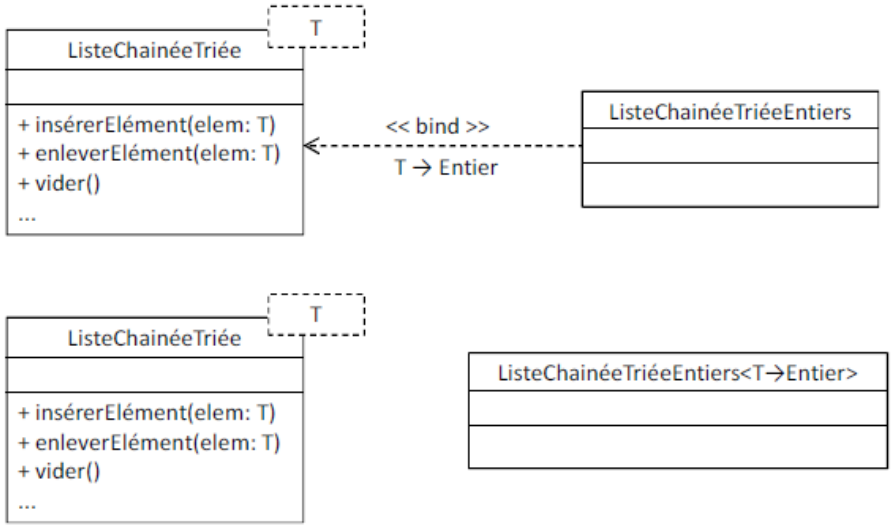
Classes génériques/paramétrables

- Objectif : regrouper les comportements associés à la structure de la classe indépendamment des objets qu'elle contient
  - Souvent utilisée pour les classes correspondant à des "collections" d'élément(s) : tableau dynamique, liste (simplement chainée, doublement chainée, triée ou non, ...), table de hashage, ensemble, ...
    $\rightarrow$ le paramètre est alors le type d'objet contenu
  - Disponible en C++ (patrons de classe = `templates`) et en Java (depuis la version 1.5 classes génériques)

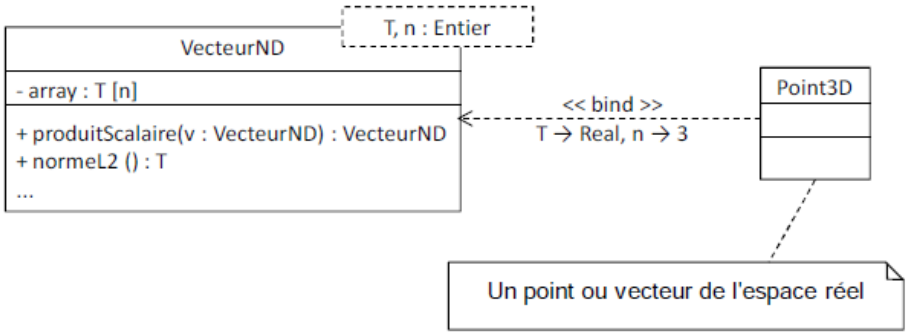


Classes génériques/paramétrables

- Paramétrisation d'une classe paramétrable = instanciation des paramètres (*binding*). Deux notations possibles.



- Différents paramètres possibles (type spécifié ou non (ex : T))



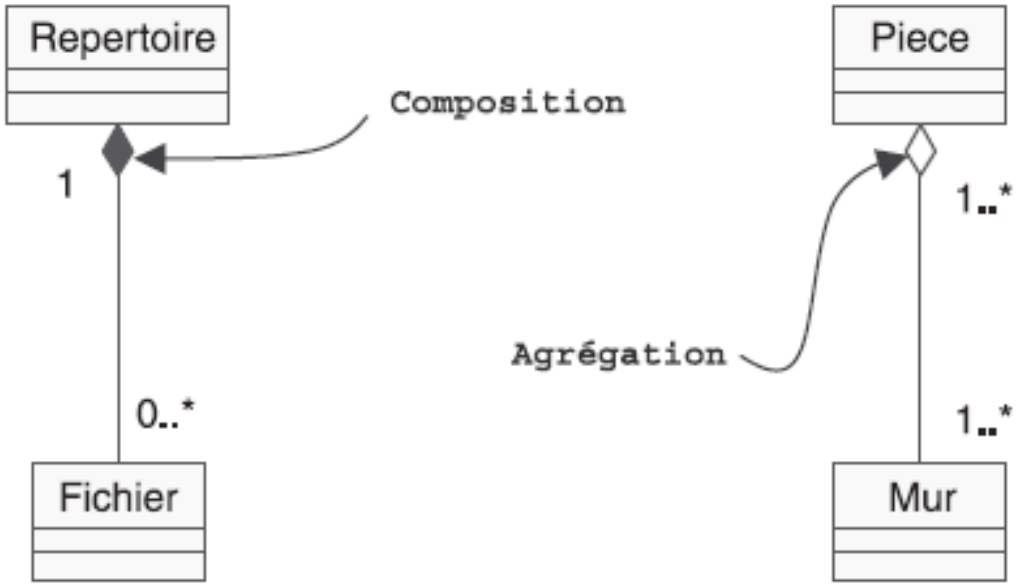
Exercice 4 : Associations entre classes

Modéliser les phrases suivantes par un diagramme de classes, notamment déterminez la relation statique appropriée (généralisation, composition, agrégation ou association)

- 1- Un répertoire contient des fichiers.
- 2- Une pièce a des murs.
- 3- A l'université, un bâtiment d'enseignement dispose d'un certain nombre de salles et de chaises. A un instant donné, une chaise est obligatoirement à l'intérieur d'une salle. Une chaise peut être déplacée dans une autre salle selon les besoins.
- 4- Les modems et les claviers sont des périphériques d'entrée/sortie (il en existe d'autres).
- 5- Une transaction boursière est un achat ou une vente.
- 6- Un compte bancaire peut appartenir à une personne physique ou morale.
- 7- Deux personnes peuvent être mariées.
- 8- Un pays possède plusieurs villes et une seule capitale.

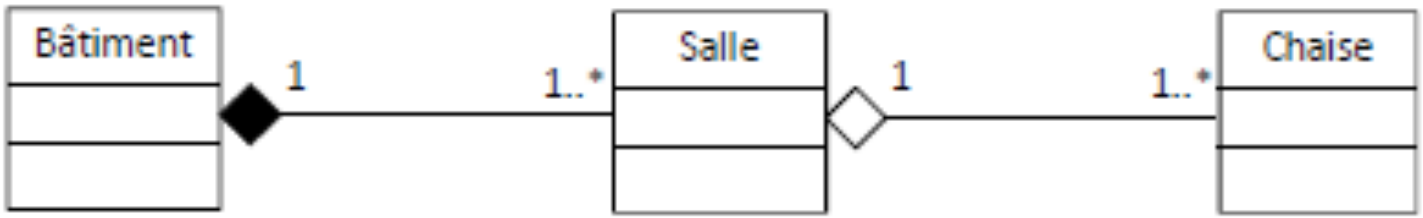
Exercice

- 1- Un répertoire contient des fichiers.
- 2- Une pièce a des murs.



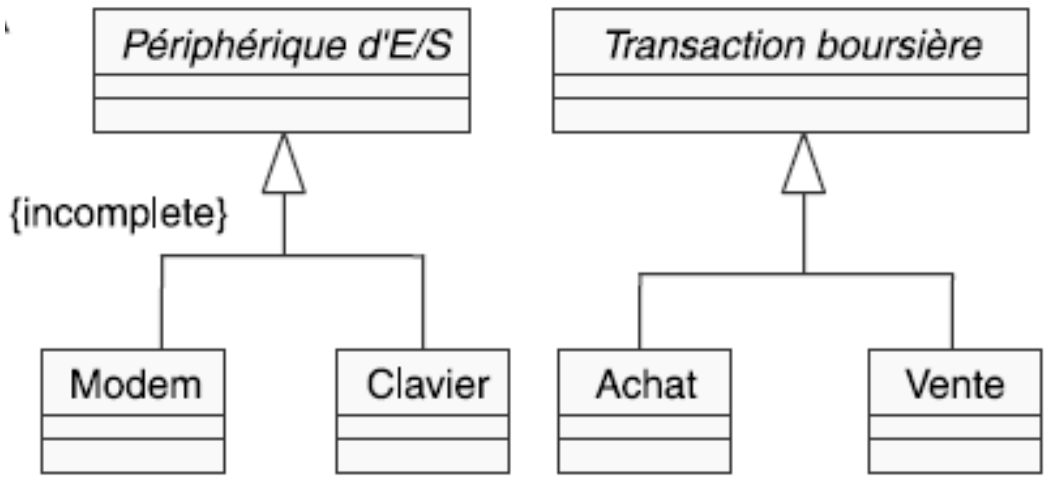
Exercice

- 3- A l'université, un bâtiment d'enseignement dispose d'un certain nombre de salles et de chaises. A un instant donné, une chaise est obligatoirement à l'intérieur d'une salle. Une chaise peut être déplacée dans une autre salle selon les besoins.



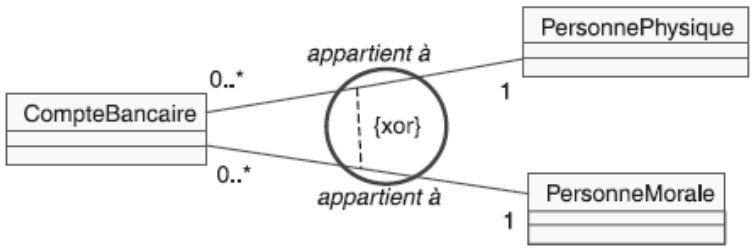
Exercice

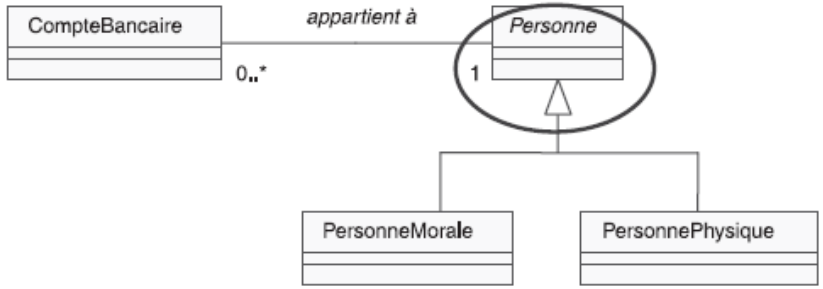
- 4- Les modems et les claviers sont des périphériques d'entrée/sortie (il en existe d'autres).
- 5- Une transaction boursière est un achat ou une vente.



Exercice

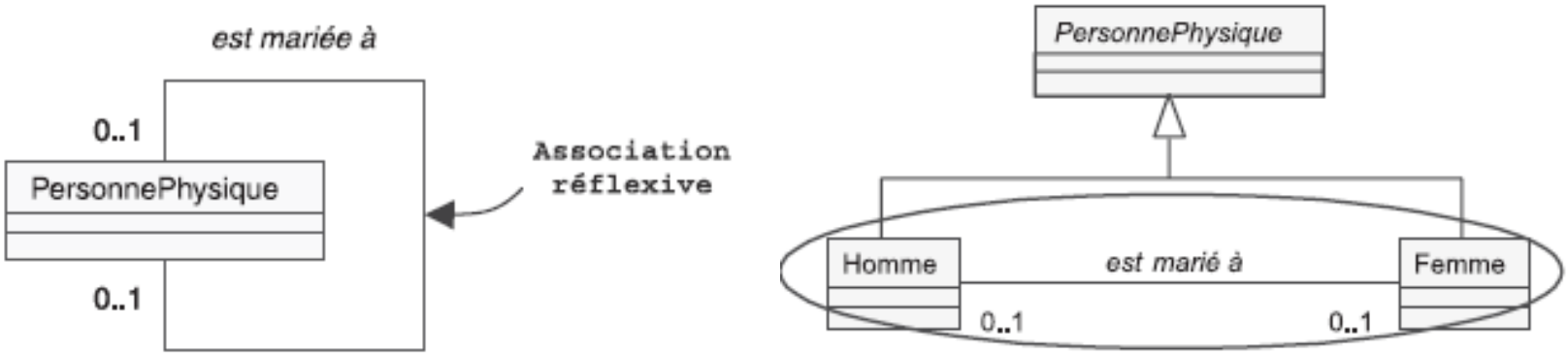
- 6- Un compte bancaire peut appartenir à une personne physique ou morale.





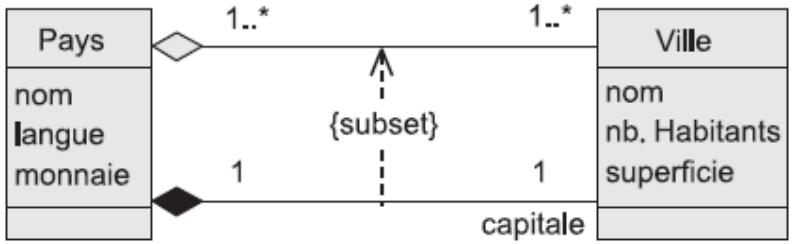
Exercice

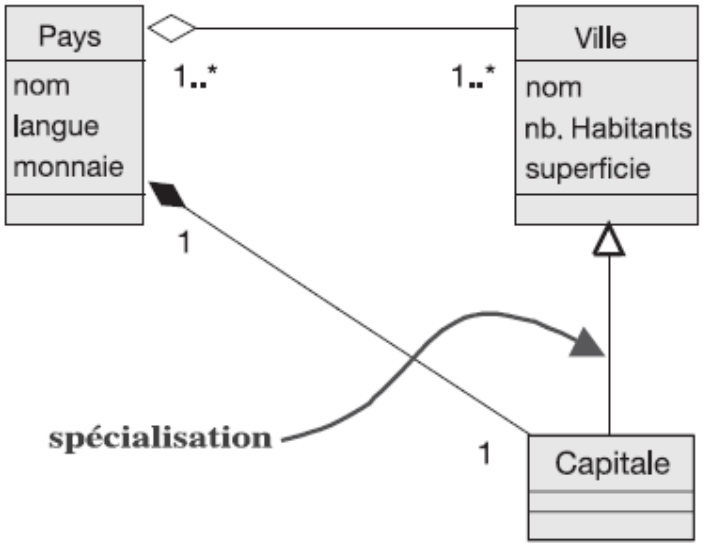
- 7- Deux personnes peuvent être mariées.



Exercice

- 8- Un pays possède plusieurs villes et une seule capitale.





Exercice 5 : Interfaces et héritage multiple

Les étudiants sont des personnes qui peuvent s'incrire et se désinscrire à l'université. Les enseignants, identifiés par un numéro, sont des personnes qui dispensent des cours à l'université. Un cours, caractérisé par son intitulé et ses crédits, n'est dispensé que par un seul enseignant. Les doctorants peuvent s'inscrire et se désinscrire à l'université comme les étudiants et dispenser des cours comme les enseignants.

- Proposer une modélisation de cette situation en utilisant l'héritage multiple.
- Proposer une modélisation de cette situation en utilisant une interface pour s'affranchir de l'héritage multiple.

Exercice 5 : Interfaces et héritage multiple

Exercice 5 : Interfaces et héritage multiple

Plan

Traduction du diagramme de classe vers Java/C++

Implémentation de certains concepts objet vus au premier cours :

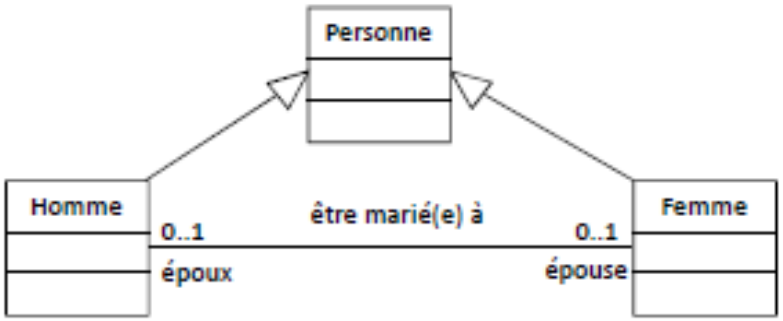
- Définition d'une classe
- Encapsulation : public, protected, private
- Membres statiques (attributs et méthodes de classe)
- Héritage
- Méthode abstraite (= virtuelle pure), classe abstraite, interface

Implémentation restant à étudier :

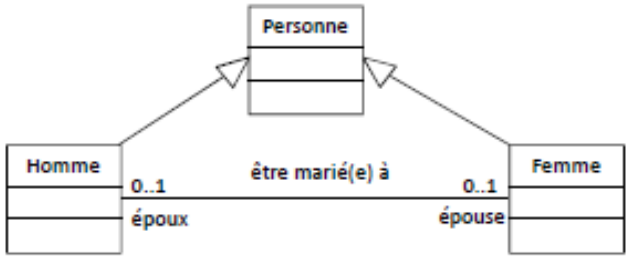
- Association (binaire/n-aire, classe-association, avec contrainte et navigabilité, agrégation, composition)

Association binaire

- Multiplicité (1 ,1) ou (0..1, 0..1)
- L'association, si elle est navigable dans les deux sens, est implémentée par ajout dans la classe A (resp. B) d'un **unique attribut de la classe B** (resp. A).
- Lien : référence en Java, pointeur en C++
- **L'instance de l'attribut est supposée avoir été créée auparavant !**
- Exemple avec deux sous-classes d'une classe mère commune



Association binaire



Homme.java

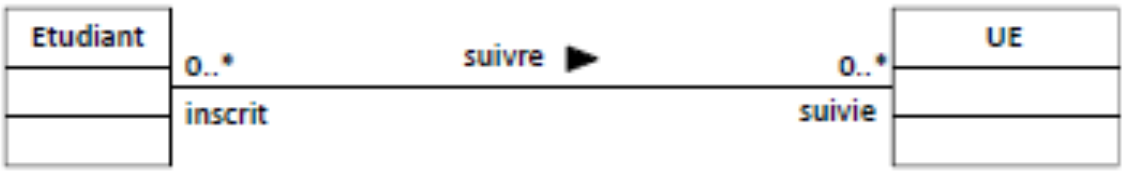
```
public class Home extends
    Personne {
  private Femme epouse;
  public Homme() {
// Celibataire par defaut
   epouse = null; }
//mutateur
  public void modifierEpouse(
      Femme f) {
   epouse = f;}
//accesseur
  public Femme renvoyerEpouse() {
   return epouse;}
  public void marrierA(Femme f) {
   epouse = f;
   f.modifierEpoux(this); }
};
```

Femme.java

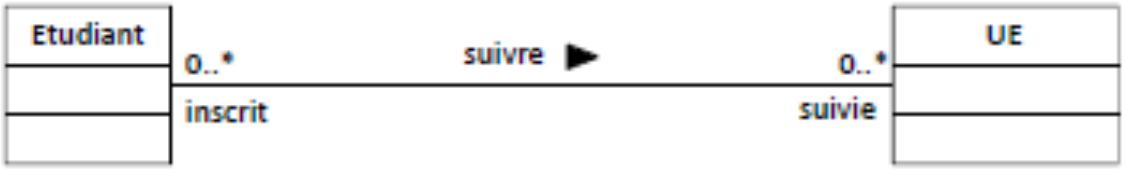
```
public class Femme extends
    Personne {
private Homme epoux;
public Femme() {
 // Celibataire par defaut
 epoux = null;  }
//mutateur
public void modifierEpoux(Homme h
    ) {
 epoux = h; }
//accesseur
public Femme renvoyerEpouse() {
return epoux;}
public void marrierA(Homme h) {
epoux = h;
h.modifierEpouse(this);}
};
```

Association binaire

- Multiplicité (0..* , 0..*)
- L'association, si elle est navigable dans les deux sens, est implémentée par ajout dans la classe A (resp. B) **d'un attribut correspondant à une "collection"** (au sens large) d'éléments. Chaque élément représente un lien vers une instance de la classe B (resp. A).
- Collection : peut être implémentée en tableau dynamique, liste chainée, ...
  - Java : Vector, List, ArrayList, Set, ...
  - C++ : std : :vector, std : :list, ...
- Vérification qu'une même instance n'est pas liée plusieurs fois
- Exemple avec :



Association binaire



Etudiant.java

```
public class Etudiant {
 private Vector<UE> listUESuivies;
 public Etudiant() {  listUESuivies = new Vector<UE>(); }
 public int renvoyerNbUESuivies() {return listUESuivies.size();}
 public void ajouterUE(UE u) {
  if (suitDeja(u)==false)
   listUESuivies.add(u);
 }
 public boolean suitDeja(UE u) {return listUESuivies.contains(u);}
 public UE renvoyerUESuivieIndice(int index) {return listUESuivies.
     elementAt(index);}
 public void suivre(UE u) {
  if (suitDeja(u)==false) {
   listUESuivies.add(u);//Les listes contiennent des ref vers des
       instances qui sont
   u.ajouterEtudiant(this);//supposees avoir ete creees auparavant !
  }
 }
 ... // + methodes figurant dans le diagramme
};
```

Association binaire

- Dans la classe Etudiant (resp. UE), la liste listUESuivies (resp. listEtudiants) n'est pas propriétaire des instances d'UE (resp. Etudiant). Les listes contiennent des références (Java)/pointeurs (C++) vers des instances qui sont supposées avoir été créées auparavant !
- L'action d'inscrire un étudiant à une UE ne crée pas de nouvelle instance d'Etudiant ni d'UE (car l'association n'est pas une relation de composition dans le cas présent)

Association avec sens de navigation

- Dans cet exemple, la navigation est possible uniquement du polygone vers les points : un polygone connait les points qui lui servent de sommets, mais un point ne connait pas les polygones dans lesquels il est sommet.
- Incidence sur l'implémentation : dans la classe Point, aucun attribut ne fait intervenir la classe Polygone

Association avec sens de navigation

Polygone.java

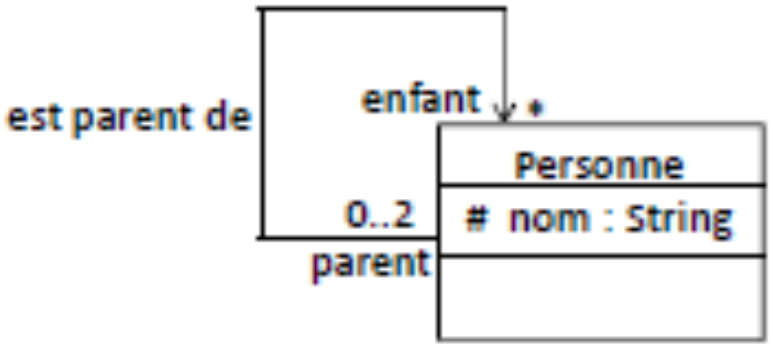
```
public class Polygone {
 protected Vector<Point> listSommets;
 public Polygone() {listSommets = new Vector<Point>();...}
 public int renvoyerNbSommets() {return listSommets.size();}
 public void ajouterSommet(Point pt) {listSommets.add(pt);}
 public boolean aPourSommet(Point pt) {
  return listSommets.contains(pt);}
 public UE renvoyerSommetIndice(int index) {
   return listSommets.elementAt(index);}
 ... // + methodes figurant dans le diagramme
};
```

Point.java

```
public class Point {
 public int x, y;
 ...
 public Point() {x=0; y=0;}
};
```

Association réflexive

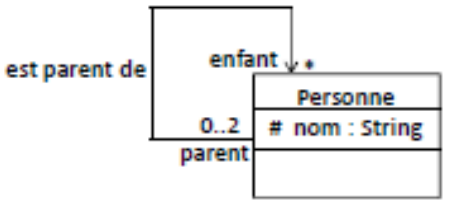
- Exemple : *une personne peut être parent d'autres personnes et enfant d'au plus deux personnes connues.*
- La relation de parenté est asymétrique : si une instance A est parent d'une instance B, l'inverse ne peut pas être possible simultanément.
- Une personne ne peut pas être son propre parent : vérifications effectués lors de la création des liens de parenté



Association réflexive

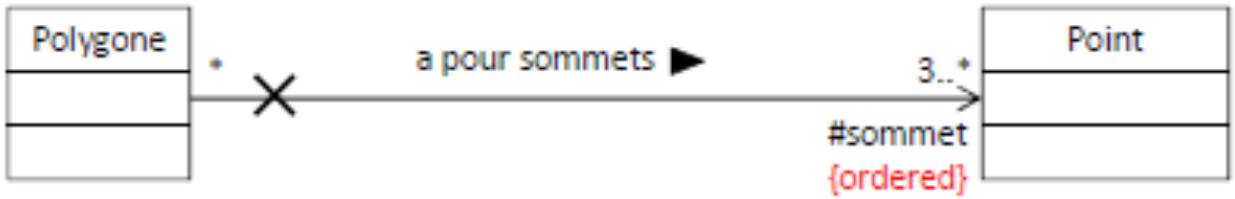
Personne.java

```
public class Personne {
 protected String nom;
 protected Vector<Personne> listEnfants;
 protected Personne parent[];
 public Personne(String s) {
   nom = new String(s);
   listEnfants = new Vector<Personne>();
   parent = new Personne[2];
   parent[0] = null;
   parent[1] = null;
 }
 public boolean estParentDe(Personne p) {
   return listEnfants.contains(p)}
 public void devenirParentDe(Personne p) {
//Verif des liens de parente
   if (p.estParentDe(this)==false && p!=this) {
     listEnfants.add(p);
     if (p.parent[0]==null)
       p.parent[0] = this;
     else if (p.parent[1]==null)
       p.parent[1] = this;
     else {//ecrase parent[0]
       p.parent[0].listEnfants.remove(p);
       p.parent[0] = this;
     }
   }
  }
```



Contraintes

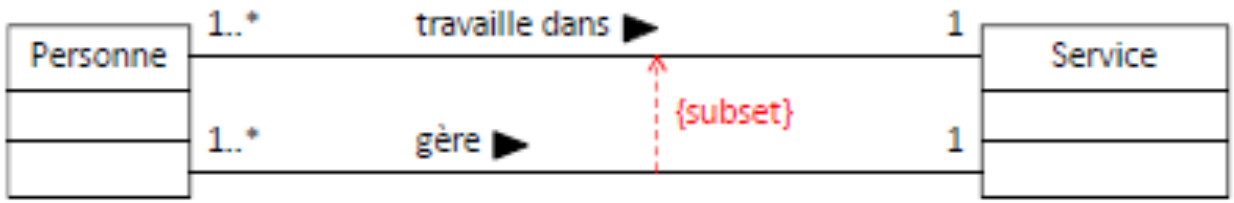
- Exemple 1 : *les sommets dans un polygone sont ordonnés.*



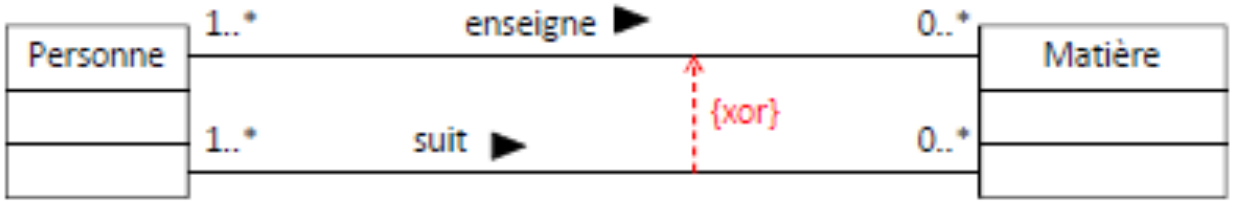
- La notion d'ordre est déjà présente dans la plupart des collections d'objet :
  - Un Vector (Java) ou un std : :vector (C++) : accès via un itérateur ou un indice
  - Une LinkedList (Java) ou une std : :list (C++) : accès via un itérateur

Contraintes

- Contrainte d'inclusion : une personne travaille dans un service, et elle peut, en plus, gérer ce service. Plusieurs employés peuvent cogérer un service



- Contrainte d'exclusivité : une personne (sans distinction de classe entre étudiant et enseignant) est associée à une matière, soit parce qu'elle l'enseigne, soit parce qu'elle la suit (mais jamais les deux simultanément)

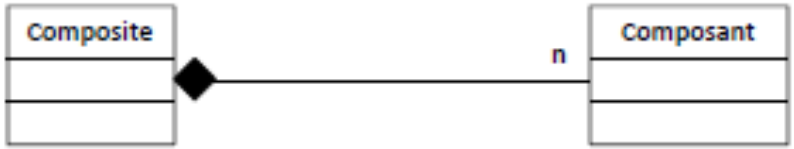


- Ces contraintes ne peuvent pas être représentée par le type de collection. Elles doivent êtres maintenues lors de la création des liens.

Agrégation et composition

- Agrégation : Implémentation similaire à une association binaire
- Composition :
  - Inclusion structurelle d'un composant dans un composite
  - La vie du Composant est liée à celle du Composite : la création (resp. destruction) du Composite entraîne la création (resp. destruction) de ses Composants
  - La composition peut être implémentée par l'ajout d'un attribut de classe Composant dans la classe Composite et **instanciation dans la classe Composite**.

Agrégation et composition



Composant.java

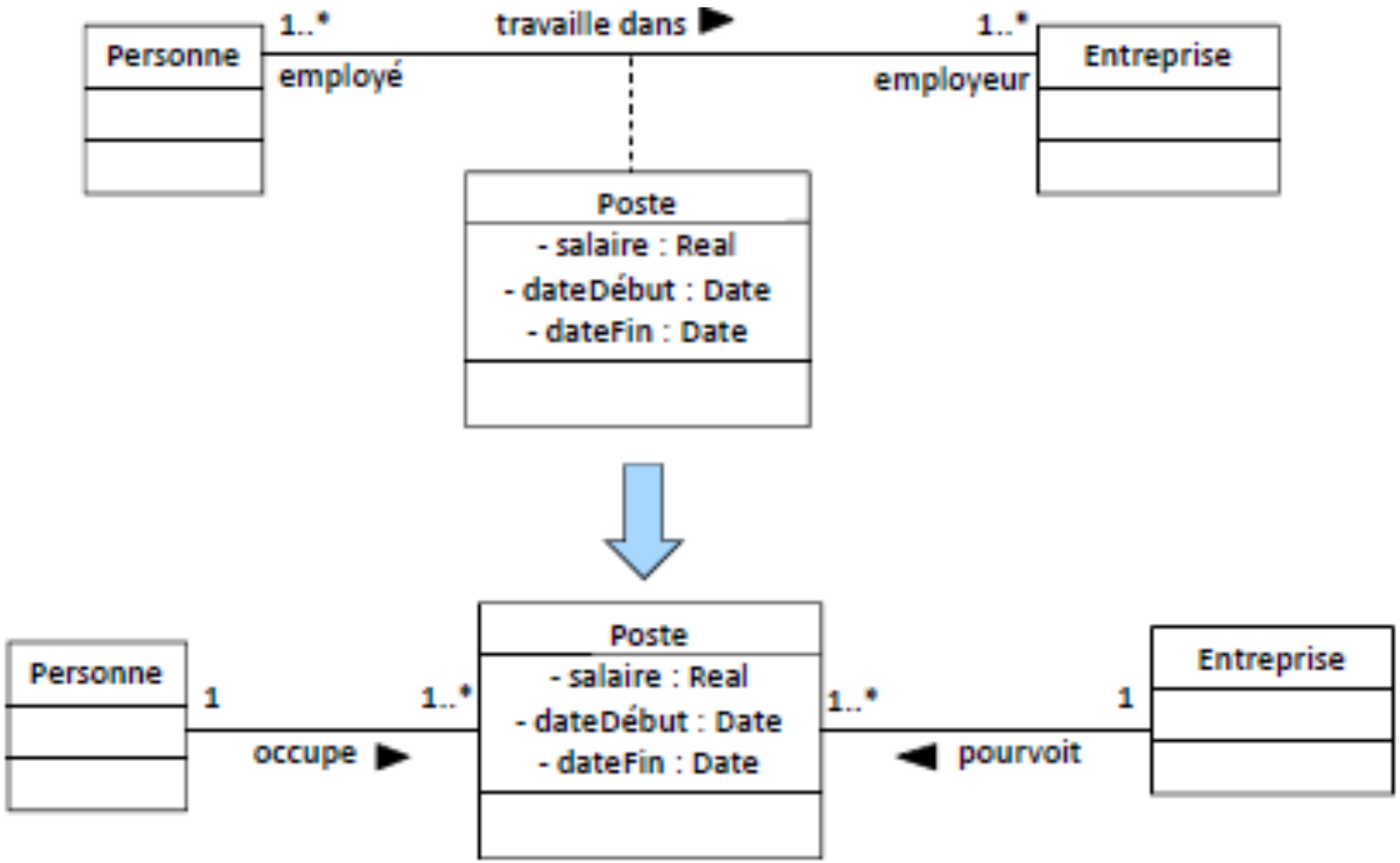
```
public class Composant {
  private Composite comp;
  public Composant() {comp = null;}
  void modifierComposite(Composite c) {comp = c;}
};
```

Composite.java

```
public class Composite {
  private Composant tabComposants[];
  public Composite() {
    tabComposants = new tabComposants[n];
    for (int i=0; i<n; i++) {
      tabComposants[i] = new Composant();
      tabComposants[i].modifierComposite(this);
    }
  }
...
}
```

Classe-association

- Implémentation de la classe-association par ajout d'une classe



Plan

Diagramme de paquetages

Qu'est-ce qu'un paquetage (package) ?

Élément de modélisation qui contient d'autres éléments de modélisation

- Regroupement d'éléments (classes, cas d'utilisation, interfaces, diagrammes, paquetages, ...) dans des ensembles fortements cohérents
- Possibilité de ne pas représenter tous les éléments contenus
- Tout élément n'appartient qu'à un seul paquetage

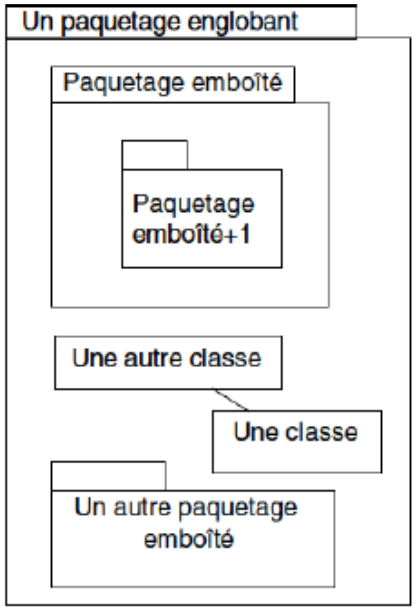
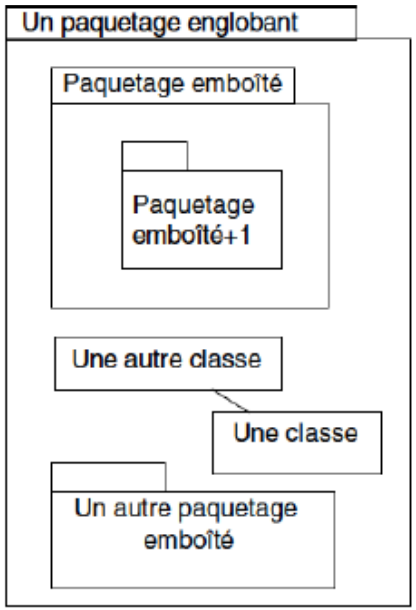


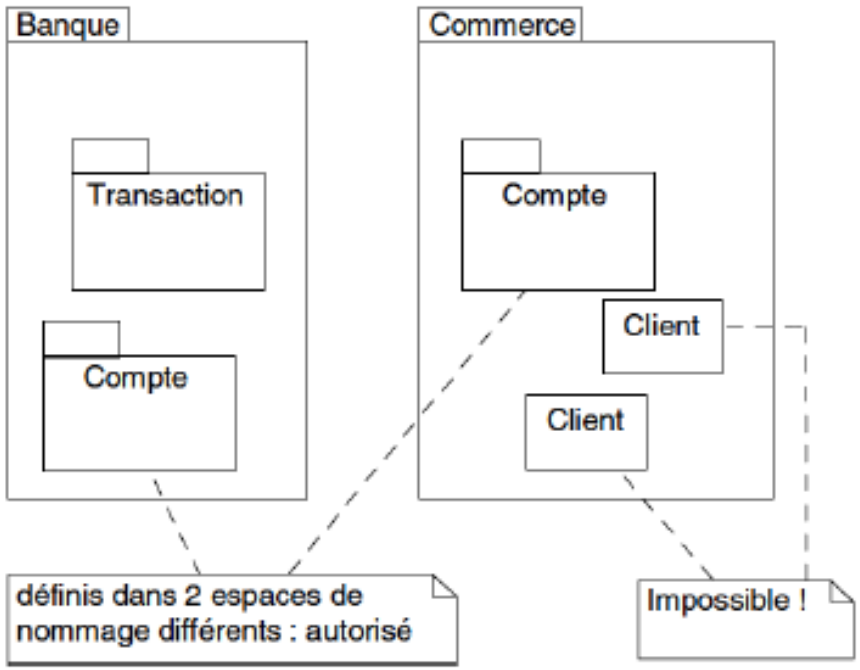
Diagramme de paquetages

- Objectifs
  - Structurer/organiser éléments et diagrammes (notamment les classes), donner une vision globale plus claire
  - Temporiser la propagation des changements dans l'application
  - Définit un espace de nom



Espaces de nom

- 2 éléments ne peuvent pas avoir le même nom dans un paquetage
- 2 éléments dans 2 paquetages différents sont différents, quel que soit leur nom
- Nom complet = nom préfixé par les noms des paquetages englobants : `Banque::Compte` ≠ `Commerce::Compte`

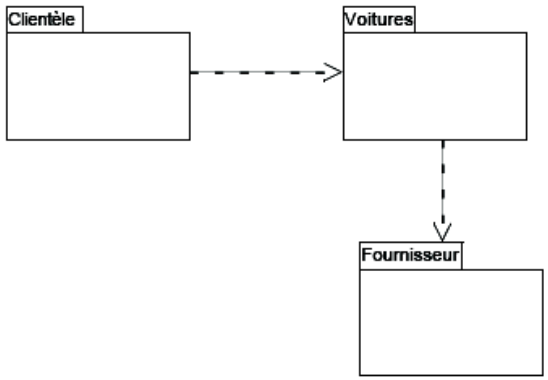


Dépendances

Définition

Une classe A dépend d'une classe B si A a des attributs de type B ou si certaines de ses méthodes renvoient, ou prennent en argument, des objets de type B. La dépendance entre paquetages est définie de manière similaire.

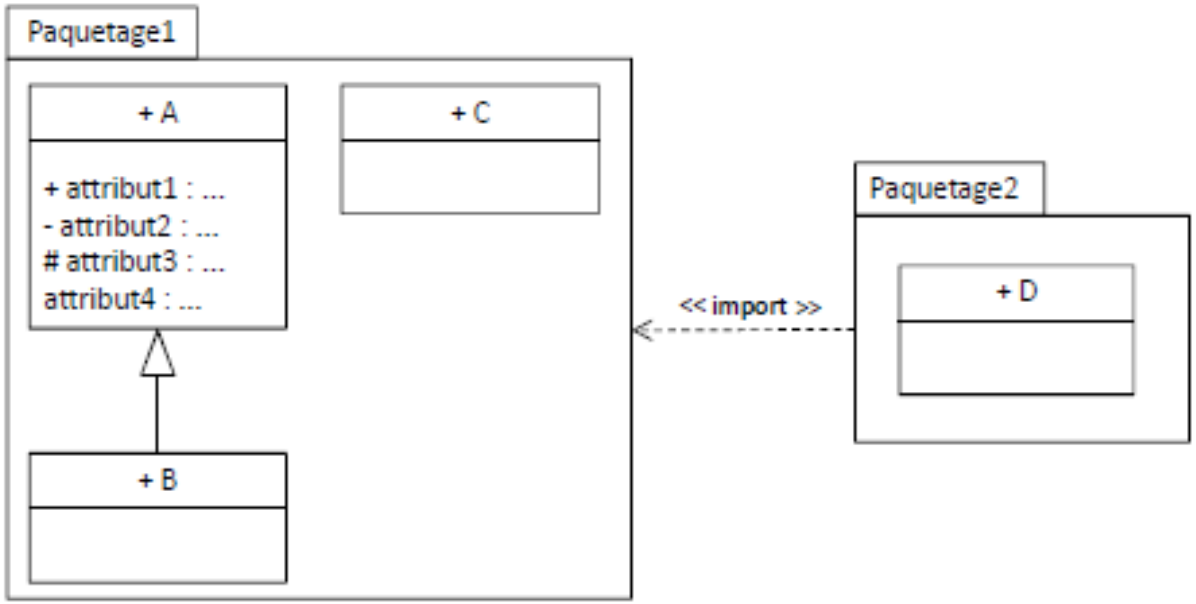
- La source voit la cible, mais pas l'inverse
- Représenté par une dépendance avec le stéréotype « import »
- Dépendances non transitives : modifier Fournisseur n'oblige pas forcément à modifier Clientèle



Visibilité

Visibilité d'une classe

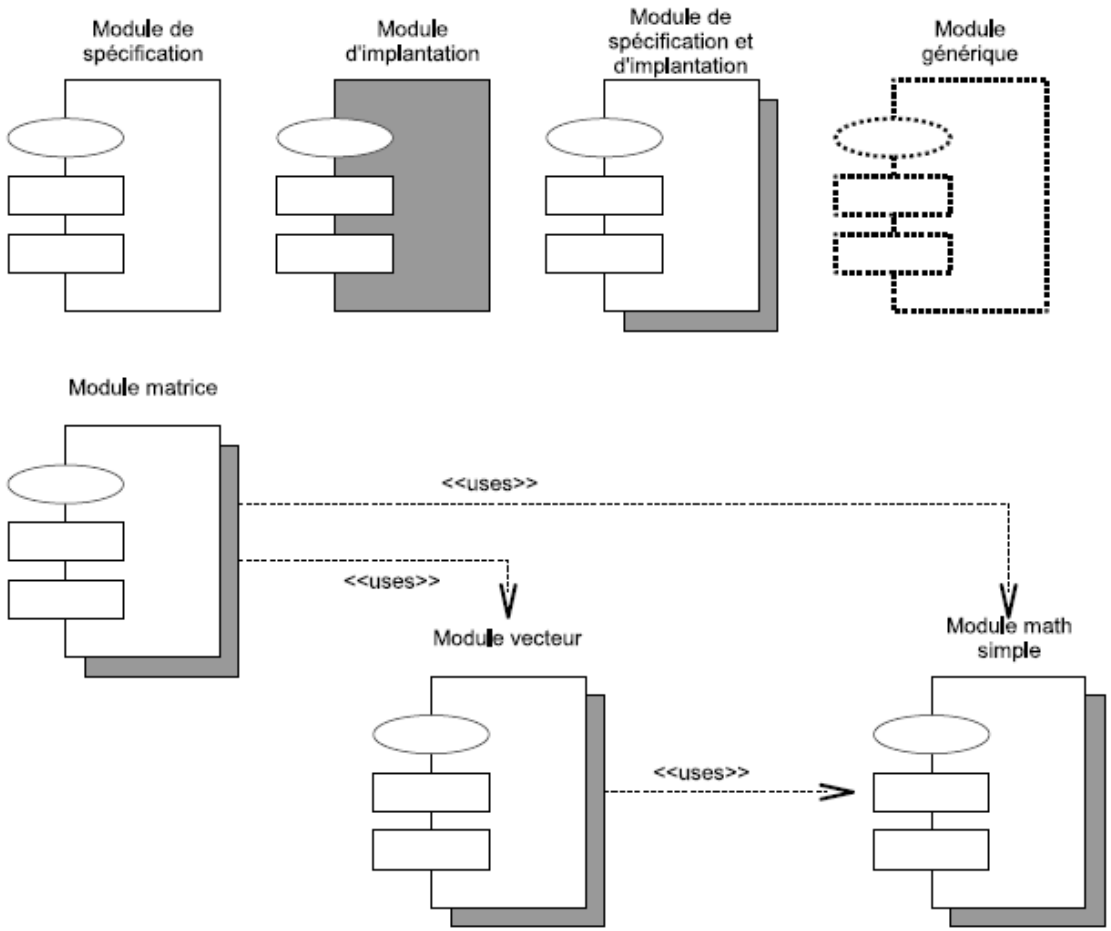
- Classe publique (+) : La classe est visible pour tous les éléments qui importent le paquetage englobant la classe.
- Classe privée (-) : La classe n'est visible que pour les classes qui appartiennent au même paquetage.
- Classe protégée (#) : La classe est visible pour les classes qui appartiennent au même paquetage et pour les classes contenues dans les paquetages enfant du paquetage englobant.



Plan

Diagramme de composants

- Permet de décrire l'architecture statique d'une application en terme de modules (fichiers sources, fichiers compilés, ...)
- Les dépendances entre modules permettent d'identifier les contraintes de compilation et de mettre en évidence la réutilisation des composants



Plan

Diagramme de déploiement

- Modélise l'**aspect matériel** de l'application
- Montre la disposition physique des différentes ressources matérielles (PC, Modem, Station de travail, Serveur, etc.) qui composent le système, leurs interconnexions et la répartition des programmes/exécutables sur ces matériels.
- Chaque ressource est matérialisée par un noeud représenté par un cube comportant un nom
- Les associations entre noeuds sont des chemins de communication qui permettent l'échange d'informations
- Un système est généralement décrit par un petit nombre de diagrammes de déploiement (généralement un seul suffit)