Chapitre II.
Diagrammes des aspects statiques

Frédéric DADEAU

Département Informatique des Systèmes Complexes - FEMTO-ST

Bureau 410 C

Email : frederic.dadeau@univ-fcomte.fr

Modélisation et Conception Orientées Objet
Licence 3 – Année 2016-2017

Plan du cours

Diagrammes de classes et d'objets

Diagrammes de composants

Diagrammes de déploiement

Plan du cours

Les diagrammes de classes et d'objets

Le diagramme de classes

Il se présente sous la forme d'un réseau :

- de **classes**, décrivant un ensemble d'objets, et
- d'**associations**, decrivant un ensemble de liens.

Il donne les informations sur les liens d'un objet vers les autres objets.
Le but est de décrire ce qu'il faut réaliser, et non pas comment le réaliser.

Le diagramme d'objets

C'est une instanciation d'un diagramme de classe. Il se présente sous la forme d'un réseau :

- d'**objets**, instances des classes du diagramme associé, et
- de **liens**, décrivant les interactions existantes entre les objets

La notion de classe

Définition

Une classe est une abstraction qui représente un ensemble d'objets, ayant une structure et un comportement commun.

Par exemple : "Etudiant" est une classe d'objets.

Un objet est une **instance** de sa classe ; une classe est dite **instanciée** lorsqu'elle contient au moins un objet.

Concrètement ...

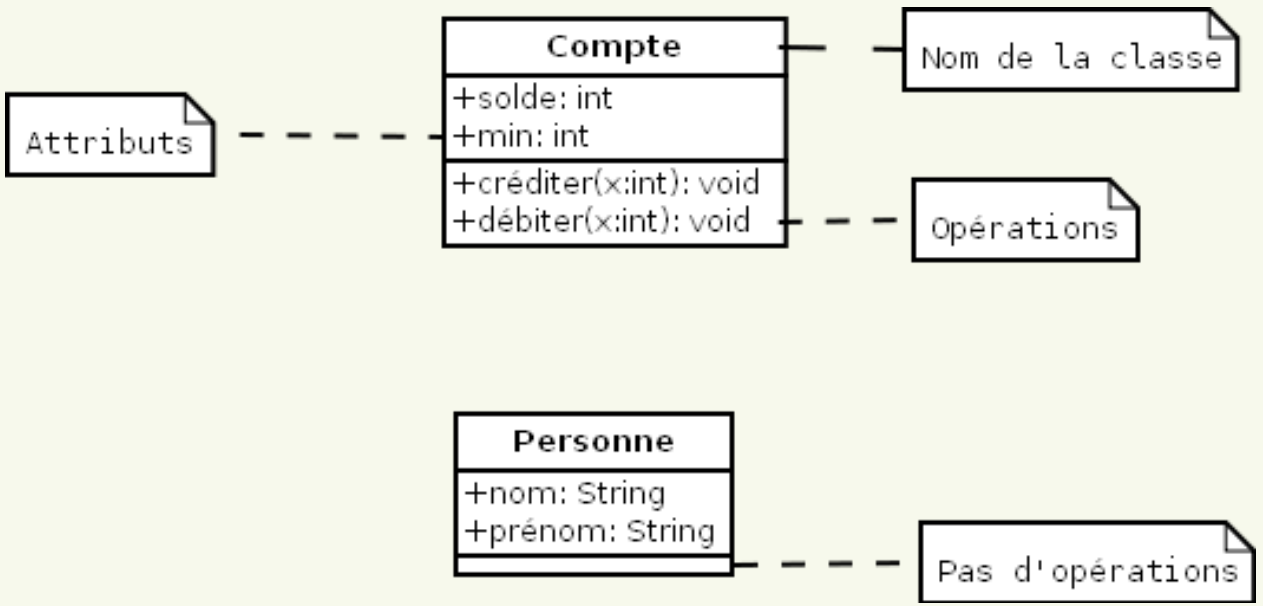
Une classe contient :

- un nom
- des attributs
- des opérations

Exemple et représentation

Classes **Compte** et **Personne**

Représentation des classes Compte et Personne



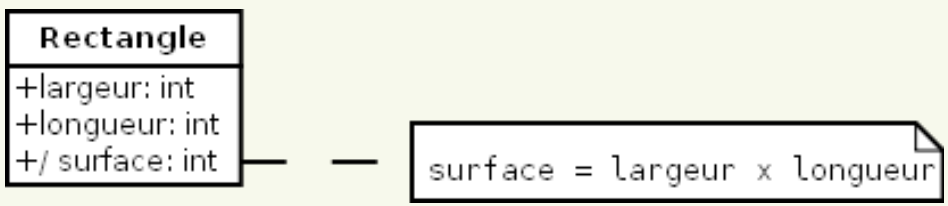
Les attributs

Un attribut est une propriété d'un objet de la classe qui comporte :

- un nom (unique dans sa classe)
- un type (facultatif, mais conseillé)
- une valeur initiale (facultative)

Un attribut est dit **dérivé** si sa valeur peut être calculée à partir d'attributs non-dérivés. Leurs noms sont précédés par '/'.

Attribut dérivé



Chaque attribut à une valeur pour chaque instance de classe. Les instances de classe peuvent avoir des valeurs identiques ou différentes pour un attribut donné.

Les opérations et les méthodes

Une opération est un service offert par les objets de la classe.

En UML, on distingue **opération** et **méthodes** :

- une opération spécifie un service
- une méthode implante une opération

Syntaxe d'une opération

Opération ($mode_1$ arg_1 : $Type_1$ = Val_1, . . ., $mode_N$ arg_N : $Type_N$ = Val_N) : $TypeR$

- Le *mode* peut être : in (paramètre d'entrée), out (paramètre de sortie), inout (paramètre d'entrée et de sortie). Le mode par défaut est in.
- Les paramètres, le type et la valeur initiale sont optionnels.

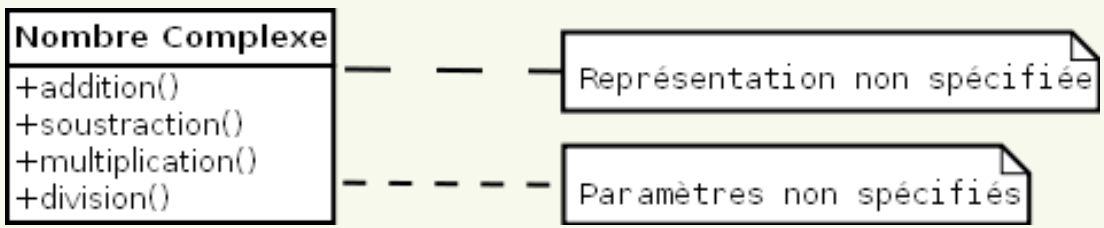
Visibilité des attributs et des opérations

Il y a 4 niveau de visibilité qui peuvent s'appliquer aux attributs et aux opérations.

- Le niveau **public** (**+**), qui indique qu'un élément est visible par tous les clients de la classe.
- Le niveau **protégé** (#), qui indique qu'un élément n'est accessible que pour les sous-classes de la classe où il apparaît.
- Le niveau **paquetage** (˜), qui indique qu'un élément est visible uniquement pour les classes définies dans le même paquetage.
- Le niveau **privé** (**-**), qui indique que seule la classe où est défini cet élément peut y accéder.

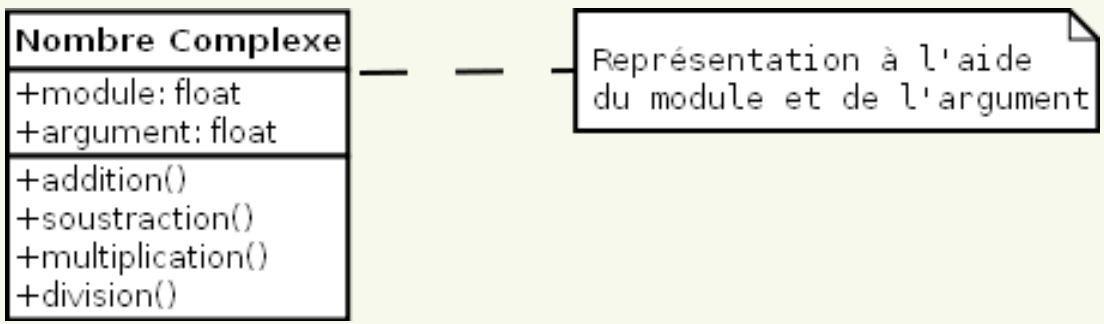
Exemple des nombres complexes

Nombres complexes, sans préciser l'implantation



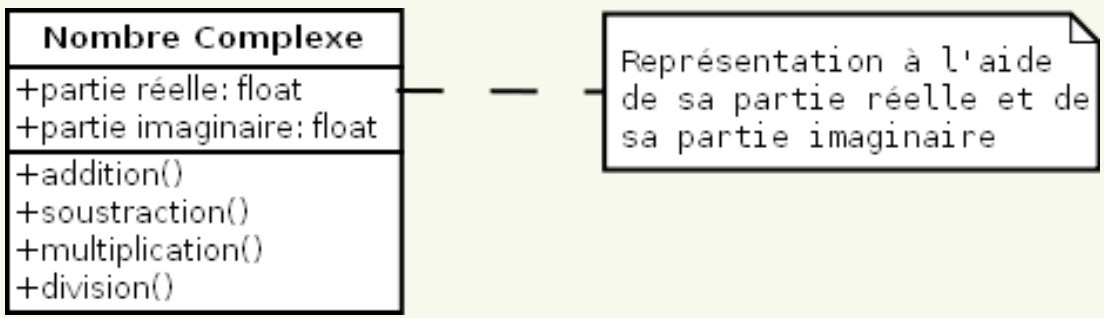
Exemple des nombres complexes

Nombres complexes, implantation à l'aide du module et de l'argument



Exemple des nombres complexes

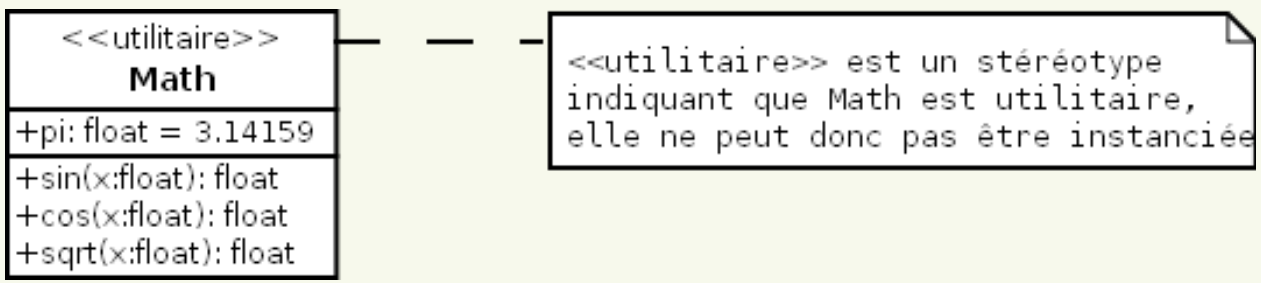
Nombres complexes, implantation à l'aide de la partie réelle et de la partie imaginaire



Classes utilitaires

Une classe utilitaire permet de regrouper un ensemble de valeurs (valeurs d'attributs) et d'opérations. Elle ne peut être instanciée et ne comporte que des attributs ou opérations de classe.

Classe utilitaire *Math*



Stéréotypes sur les classes

Une classe peut admettre un stéréotype, écrit entre chevrons au dessus du nom de la classe. Similairement, les stéréotypes peuvent s'appliquer à certaines opérations des classes.

Les stéréotypes permettent d'étendre la sémantique des éléments de modélisation. Ils permettent de définir de nouvelles classes d'éléments de modélisation, en plus du noyau prédéfini par UML. UML propose de nombreux stéréotypes "standards" mais d'autres plus spécifiques peuvent être créés pour des besoins de modélisation.

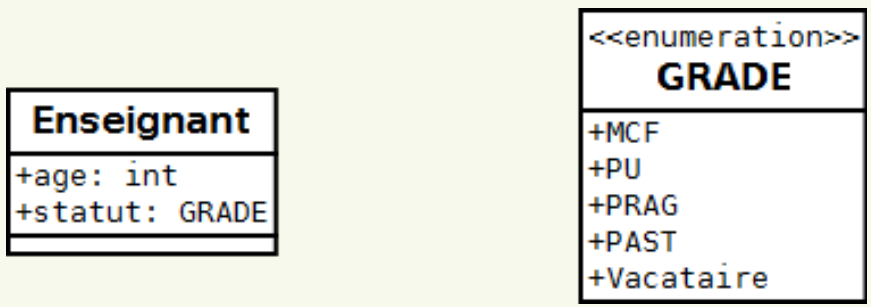
Quelques stéréotypes standards

- « thread », appliqué à une classe, qui indique que cette classe représente un processus léger
- « utility », appliqué à une classe, qui indique que cette classe est une classe utilitaire
- « constructor », « getter », « setter », appliqués à des opérations.

Stéréotype *enumeration*

Le stéréotype "enumeration" permet de définir une classe contenant un ensemble d'identificateurs servant de domaine de valeur d'un type.
Une classe ainsi stéréotypée peut alors être utilisée comme type d'attribut.

Stéréotype enumeration



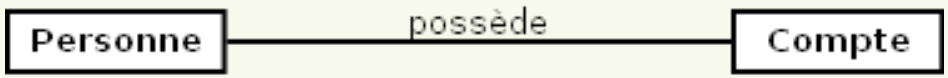
Remarque : C'est le seul cas où on tolère de typer un attribut avec une classe.

Définition d'une association

Une association est une relation *n*-aire entre *n* classes d'objets. Les relations les plus utilisées sont les relations binaires, entre deux classes.

Exemple d'association informel

Relation de possession entre une personne et un compte courant.



- Classes : Personne et Compte
- Association : relation "possède"

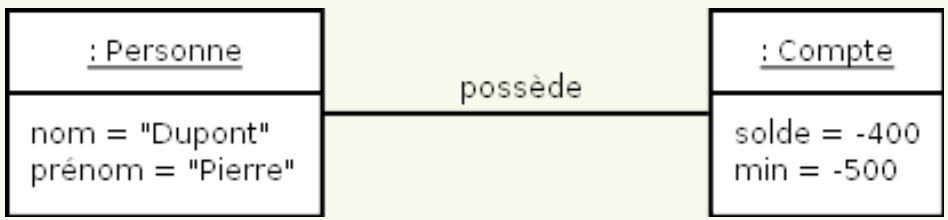
Définition d'une association

Une association est une relation *n*-aire entre *n* classes d'objets. Les relations les plus utilisées sont les relations binaires, entre deux classes.

On appelle **lien** l'instanciation d'une association entre deux objets.

Exemple de lien informel

Pierre Dupont possède le compte dont le solde est à -400.

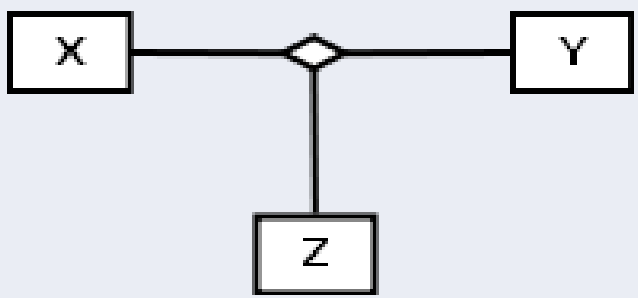


Représentation graphique d'une association

Une association compte au moins deux extrémités. Elle peut être binaire, ternaire, ou d'un autre ordre.

Représentation d'une association binaire

Représentation d'une association ternaire



Les multiplicités

La multiplicité précise combien d'instances d'une classe peuvent se rattacher à une seule instance d'une classe associée.

Valeurs de multiplicité

1	un et un seul
0..1	cas optionnel
N	si on connaît la valeur, on la précise
$M..N$	intervalle de valeurs (de $M \in \mathbb{N}$ à $N \in \mathbb{N}$)
$*$	0 ou plusieurs (identique à $0..*$)
$1..*$	au moins un

Graphiquement, la multiplicité se place du côté de la classe qu'elle décrit.

D'un point de vue mathématique

Les associations

Une association entre deux classes A et B est une relation binaire, au sens mathématique, c'est-à-dire un sous-ensemble de $A \times B$.

$$R \subseteq A \times B$$

Il est donc impossible d'avoir deux liens différents symbolisant une même association entre deux objets identiques.

D'un point de vue mathématique

Les relations

Soit R une relation entre deux classes A et B, avec les multiplicités M_A, associée à A, et M_B, associée à B. Les multiplicités M_A et M_B imposent les contraintes suivantes :

- Pour chaque objet b de la classe B, le nombre d'objets de la classe A liés à b appartient à M_A.

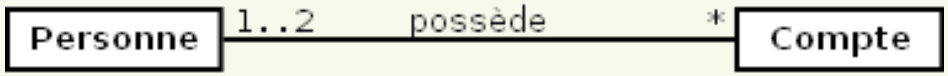
$$\forall b \in B.card(\{(a,b) \in R \wedge a \in A\}) \in M_A$$

- Pour chaque objet a de la classe A, le nombre d'objets de la classe B liés à a appartient à M_B.

$$\forall a \in A.card(\{(a,b) \in R \wedge b \in B\}) \in M_B$$

Un exemple : compte et personne

Diagramme de classes

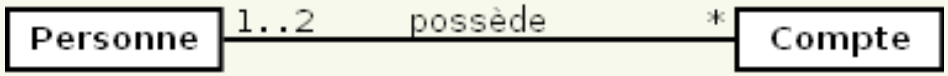


Ceci signifie :

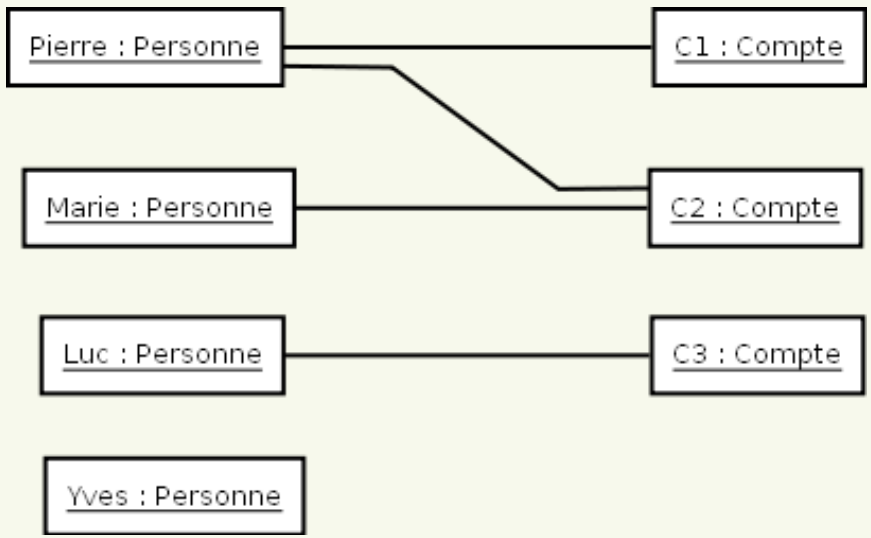
- Une personne peut ouvrir un nombre quelconque de comptes
- Un compte est ouvert pour une ou deux personnes

Un exemple : compte et personne

Diagramme de classes

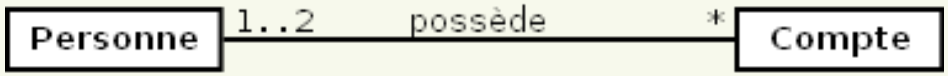


Un diagramme d'objets correct

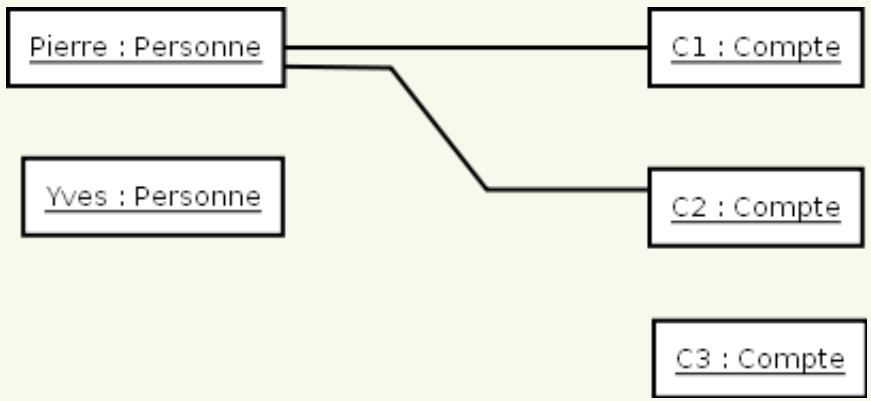


Un exemple : compte et personne

Diagramme de classes



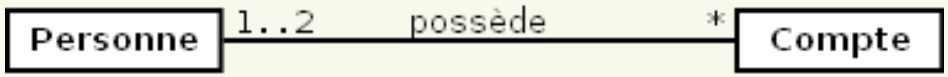
Un diagramme d'objets incorrect



Navigabilité d'une association

Par défaut, une association est navigable dans les deux sens.

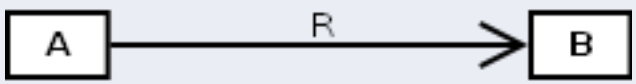
Diagramme de classes



- à partir d'une personne, on peut retrouver les comptes qui lui sont associés ;
- à partir d'un compte on peut retrouver son (ou ses deux) propriétaire(s)

Pour spécifier qu'un seul sens de navigation est autorisé, on peut orienter la relation.

Association orientée

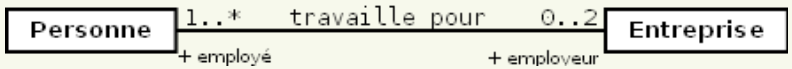


A partir d'un objet de la classe A, on peut accéder aux objets de la classe B qui lui sont liés. L'inverse n'est pas autorisé.

La notion de rôles

Les rôles désignent les extrémités d'une association. Ils peuvent être nommés.

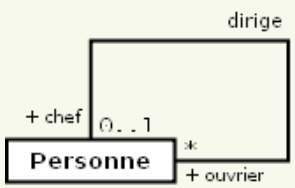
Exemple de rôles

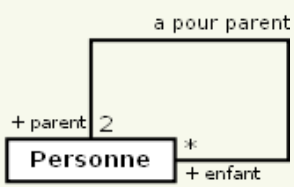


- une personne joue le rôle d'un employé
- une entreprise joue le rôle d'un employeur

Principalement utilisés dans les associations réflexives :

Deux exemples d'associations réflexives



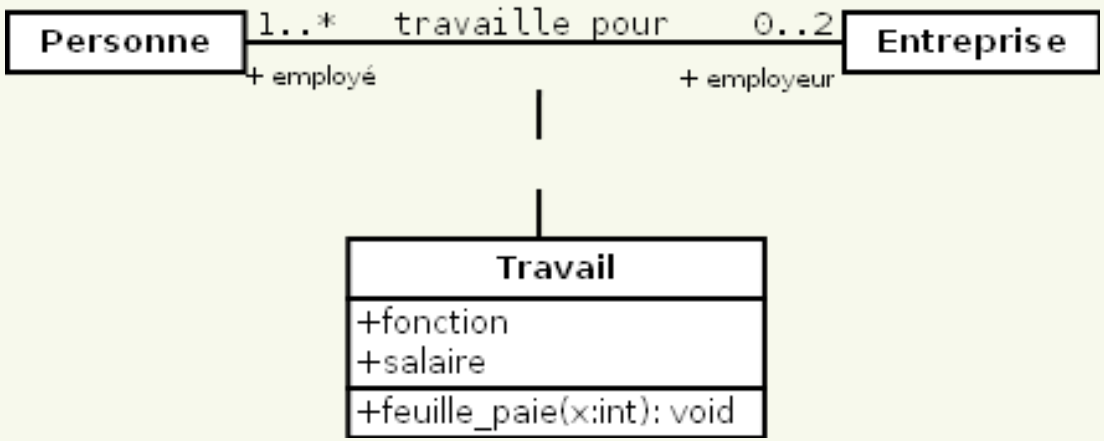


Classe-associations

Une classe-association est une entité qui combine les propriétés d'une classe et les propriétés d'une association.

Une association *R* entre deux classes *A* et *B* permet d'associer à tout lien entre un objet de la classe *A* et un objet de la classe *B* un objet de la classe *R*.

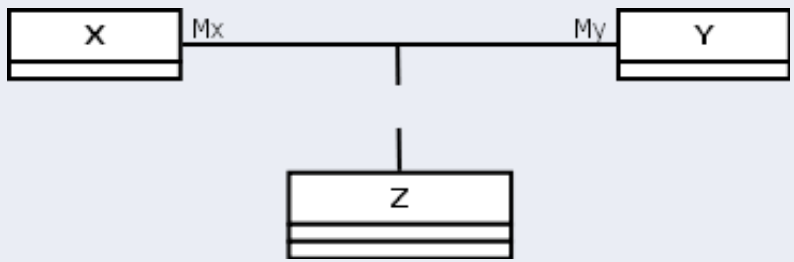
Exemple de classe-association



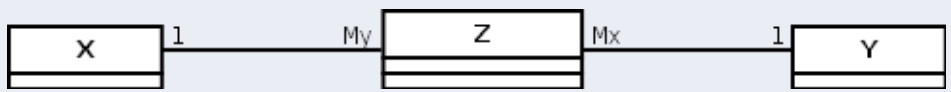
Classe-associations et multiplicités

Une classe-association entre deux classes peut se simuler par l'intermédiaire d'une classe et de deux associations.

Simulation d'une classe association

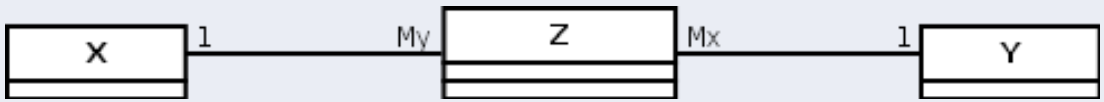


Se simule par :



Classe-associations et multiplicités

Simulation d'une classe association

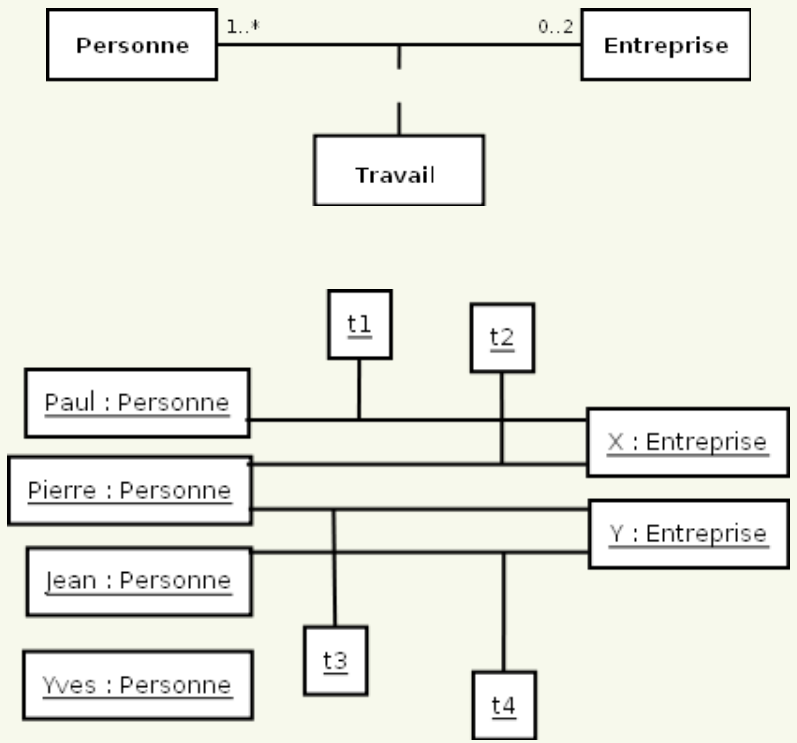


Lecture des multiplicités

- Chaque objet de la classe Z est associé à un objet de la classe X et un objet de la classe Y
- Chaque objet de la classe X est associé à n_Y objets de la classe Z, avec $n_Y \in M_Y$
- Chaque objet de la classe Y est associé à n_X objets de la classe Z, avec $n_X \in M_X$

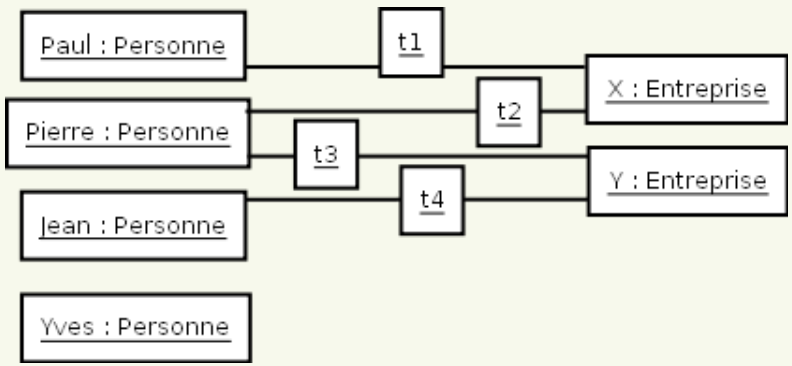
Classes-associations et objets

Classe-association



Classes-associations et objets

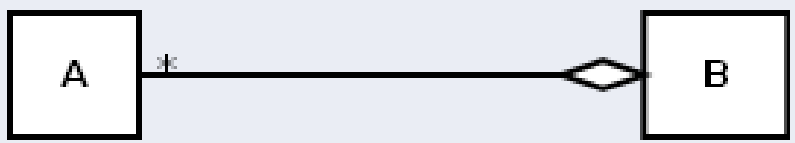
Classe-association simulée



Agrégation et composition (1/2)

L'agrégation est un association binaire particulière qui modélise une relation entre un "tout" et une "partie", autrement dit, une relation d'appartenance.

Notation de l'agrégation



Un objet de la classe B, appelé **agrégat** contient un certain nombre d'objets de la classe A, appelés **composants**.

Exemple d'agrégation

Diagramme de classes

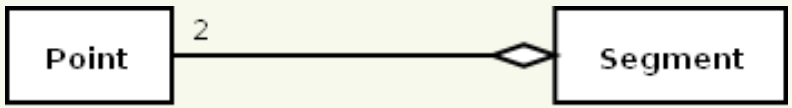
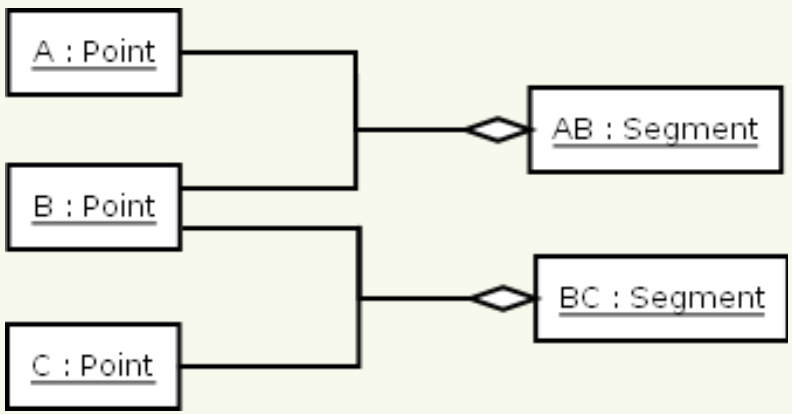


Diagramme d'objets



Remarque sur l'agrégation

La relation d'agrégation interdit les cycles dans les diagrammes d'objets.

Par contre, dans le diagramme de classes, on peut définir une classe A en agrégation avec elle-même.

Diagramme de classes et diagramme d'objet

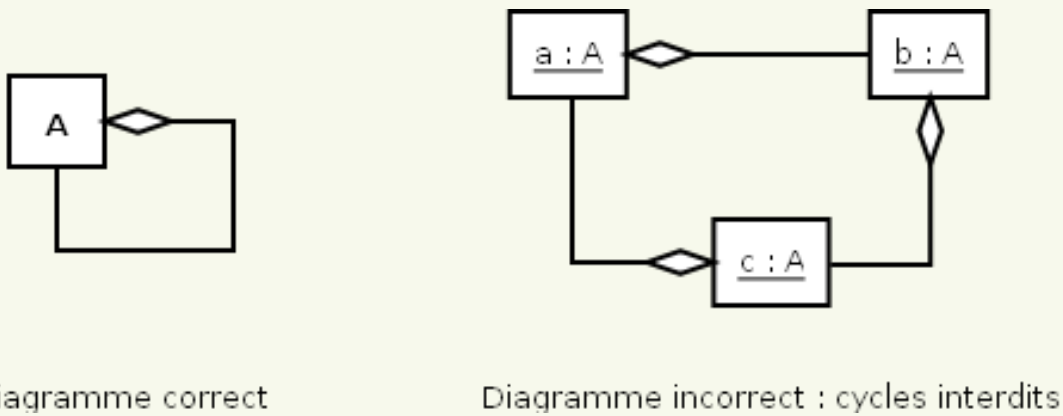


Diagramme correct

Diagramme incorrect : cycles interdits

Dans une agrégation, certaines parties peuvent être partagées. Par défaut, il n'y a pas de contrainte de multiplicité sur l'agrégat.

Agrégation et composition (2/2)

La composition est une agrégation particulière pour laquelle une partie ne peut pas être partagée entre plusieurs agrégats. Autrement dit, la multiplicité du côté de l'agrégat est 0 ou 1.

Notation de la composition, équivalence avec l'agrégation

Les cycles de vies des composants et de l'agrégat sont liés : si l'agrégat est détruit, ses composants le sont aussi.

Exemple de composition

Diagramme de classes

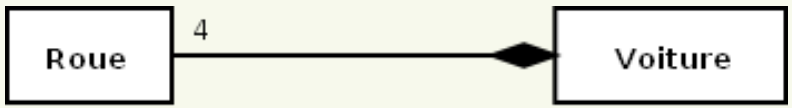
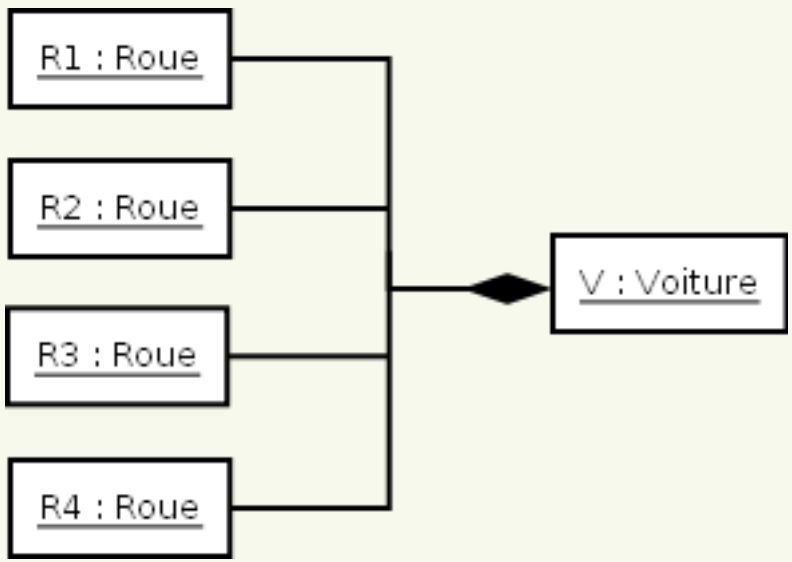


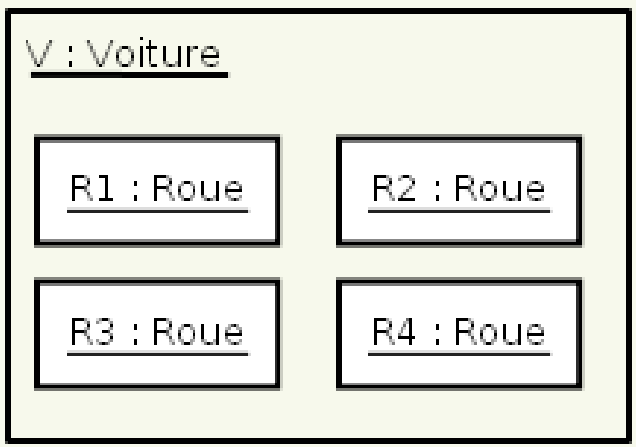
Diagramme d'objets



Remarque sur la composition

Il est également possible de représenter les composants d'un objet à l'intérieur de l'objet lui-même.

Représentation



Attention : ceci n'est possible que pour les compositions car deux objets ne peuvent pas partager un même composant.

Associations *n*-aires

Une association *n*-aire est une relation entre *n* classes.

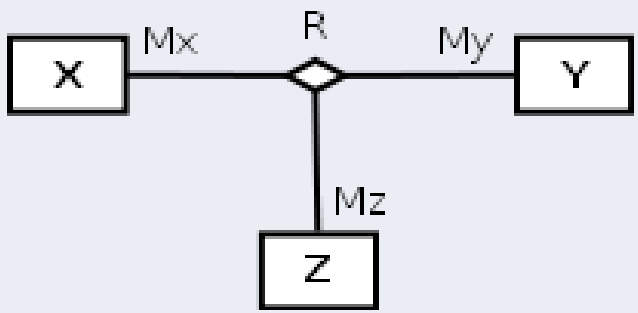
Mathématiquement parlant ...

Une association *n*-aire entre les classes $A_1, A_2, \ldots, A_n$ est un sous-ensemble R de $A_1 \times A_2 \times \ldots \times A_n$, donc un ensemble de *n*-uplets de la forme $(a_1, a_2, \ldots, a_n)$ avec $\forall i \in 1, 2, \ldots, n, a_i \in A_i$.

Associations *n*-aires, multiplicités

La multiplicité associée à une classe pose une contrainte sur le nombre de *n*-uplets de la relation lorsque les $n-1$ autres valeurs sont fixées.

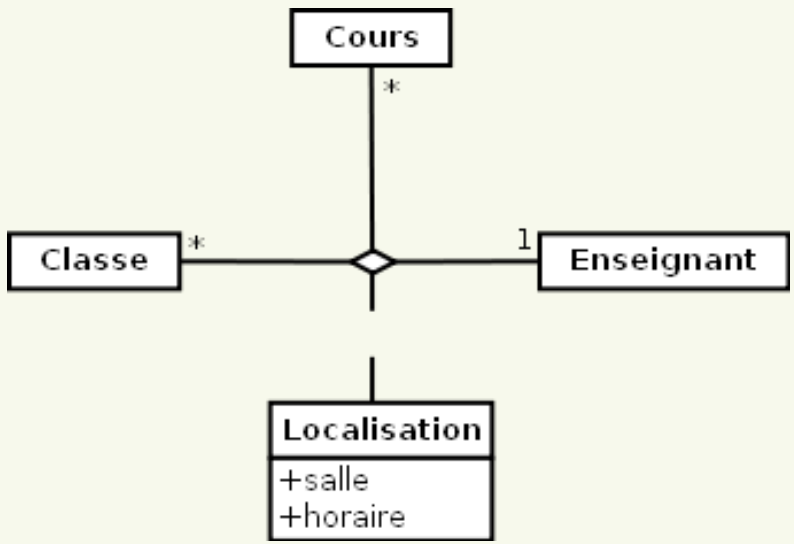
Notation



- $\forall y_0 \in Y.\forall z_0 \in Z.card((x, y_0, z_0) \in R \wedge x \in X) \in M_X$
- $\forall x_0 \in X.\forall z_0 \in Z.card((x_0, y, z_0) \in R \wedge y \in Y) \in M_Y$
- $\forall x_0 \in X.\forall y_0 \in Y.card((x_0, y_0, z) \in R \wedge z \in Z) \in M_Z$

Exemple d'association *n*-aire

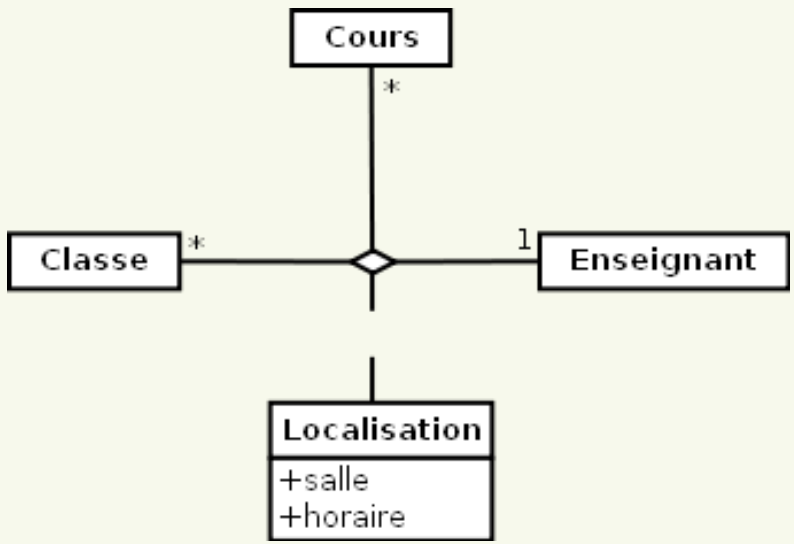
Association ternaire



- un enseignant peut faire un même cours à plusieurs classes (multiplicité "*" associée à la classe Classe)

Exemple d'association *n*-aire

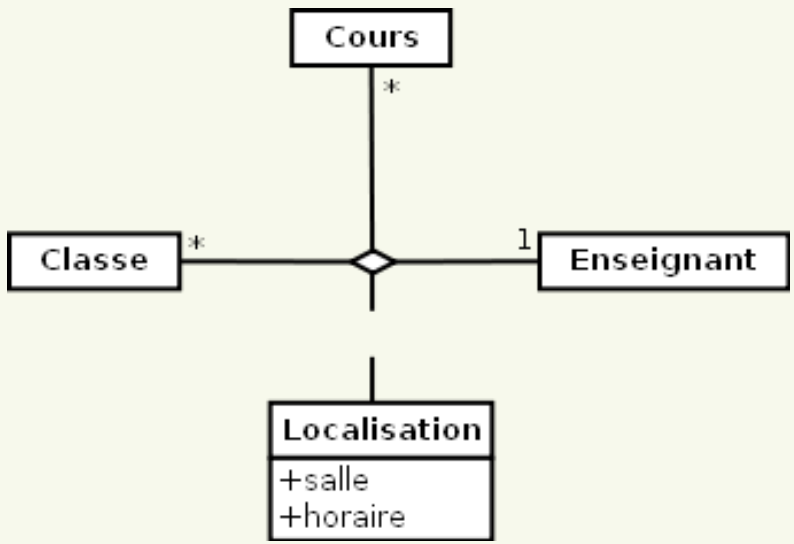
Association ternaire



- dans une classe, un cours est effectué par un seul enseignant (multiplicité "1" associée à la classe Enseignant)

Exemple d'association *n*-aire

Association ternaire



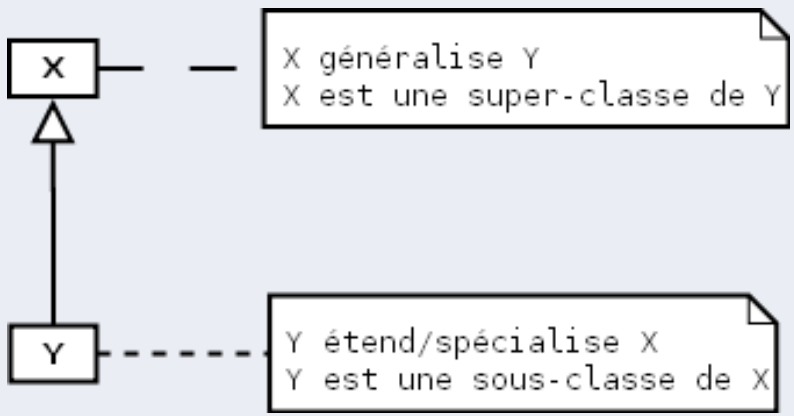
- un enseignant peut faire plusieurs cours dans une même classe (multiplicité "*" associée à la classe Cours)
  Un enseignant peut donc enseigner plusieurs matières.

L'extension

Une classe Y **étend** ou **spécialise** une classe X si tout objet de la classe Y est, ou peut être considéré comme, un objet de la classe X.

A l'inverse, on dira que la classe X **généralise** la classe Y.

Notation de l'extension



Les propriétés de l'extension

Propriété d'héritage

Les attributs et les opérations définis dans X, qui ne sont pas privés, et qui ne sont *redéfinis* dans Y, sont *hérités* dans Y.

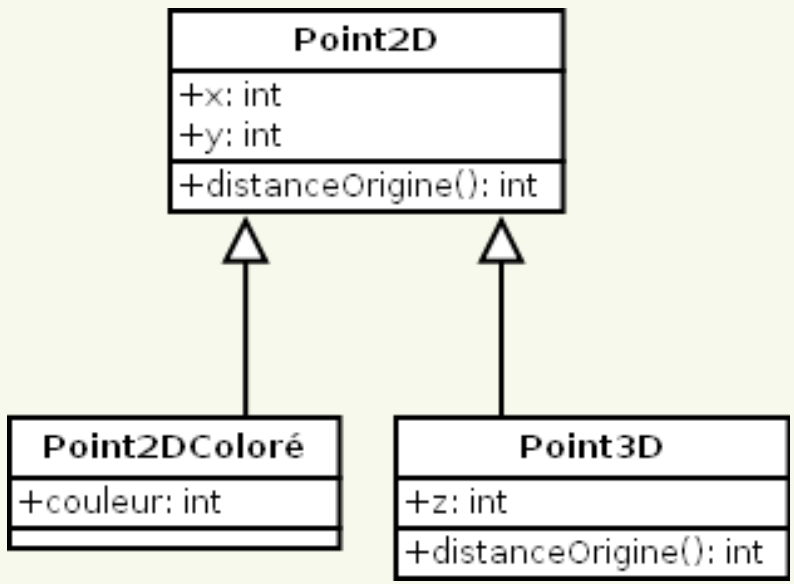
Propriété de substitution

La propriété de substitution exprime que tout objet de la classe Y *est* ou *peut être considéré comme* un objet de la classe X.

Chaque fois qu'on a besoin d'une instance de X, on peut utiliser à la place une instance de Y.

Un exemple d'extension

Exemple avec des points



Hiérarchie des classes

L'extension permet d'exprimer la notion de hiérarchie entre différentes classes.

Hiérarchie de véhicules

On considère différents types de *véhicules* :

- Les *véhicules aériens* : les *avions*
- Les *véhicules aquatiques* : les *bateaux* et les *sous-marins*
- Les *véhicules terrestres* : les *chars* et les *voitures*

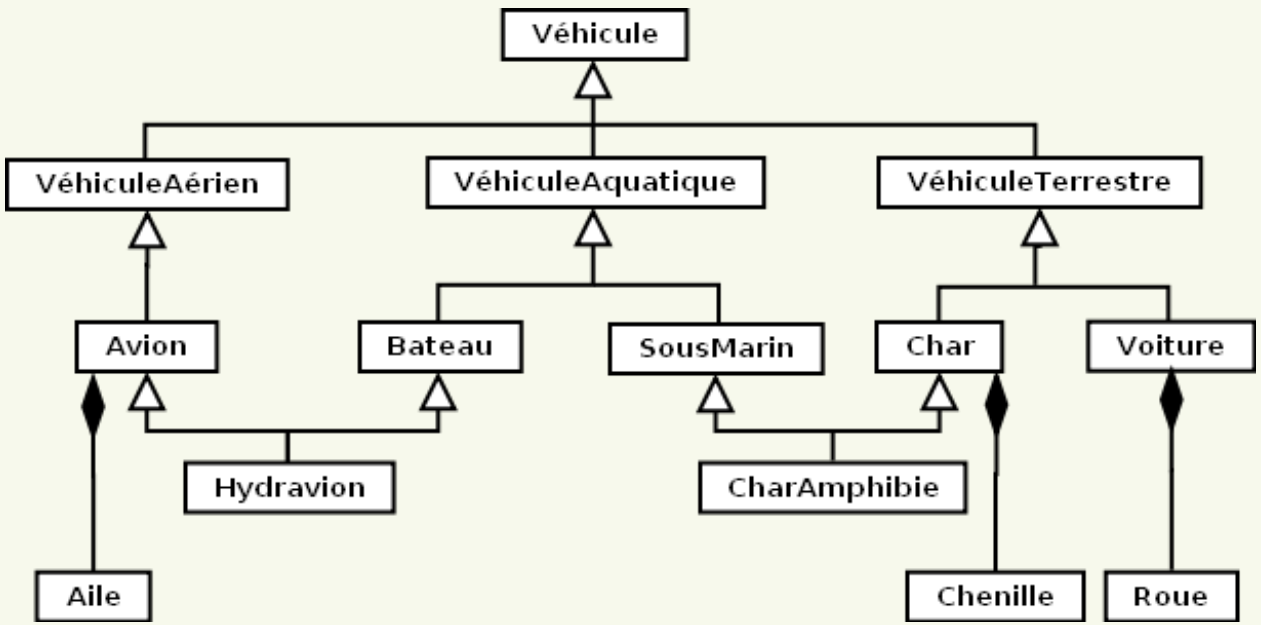
On précise :

- qu'un *hydravion* est à la fois un avion et un bateau,
- qu'un *char amphibie* est à la fois un sous-marin et un char
- qu'un avion est composé d'*ailes*
- qu'un char est composé de *chenilles*
- qu'une voiture est composée de *roues*

Hiérarchie des classes

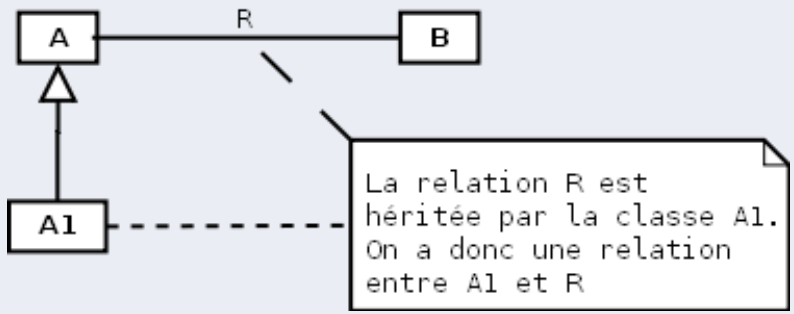
L'extension permet d'exprimer la notion de hiérarchie entre différentes classes.

Hiérarchie de véhicules (diagramme)



Héritage de relation

Relation héritée



Dans l'exemple précédent : un hydravion est, en tant qu'avion, également composé d'ailes, et un char amphibie est, en tant que char, également composé de chenilles.

Classe abstraite

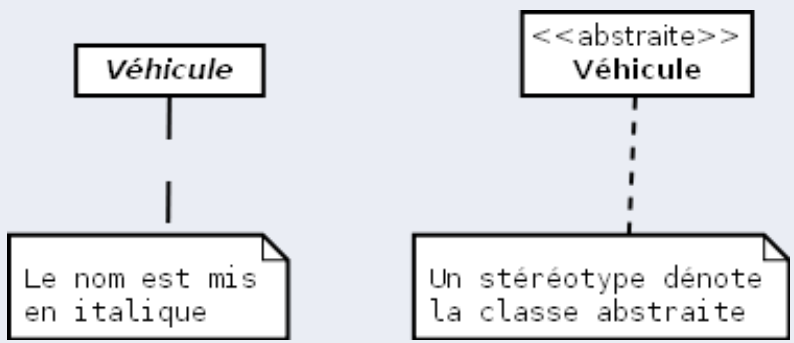
Une classe abstraite est une classe qui ne peut être instanciée, et qui peut contenir des opérations abstraites.

Une opération abstraite est une opération sans implantation, c'est-à-dire à laquelle ne correspond aucun code. Une classe abstraite peut aussi contenir des opérations concrètes (= non-abstraites).

Les classes abstraites servent notamment dans les hiérarchies de classes, où elles permettent de regrouper des attributs et des opérations communs à plusieurs classes.

En programmation objet (par exemple, en Java) une classe concrète doit implanter les opérations abstraites dont elle hérite.

Notation



Exercice

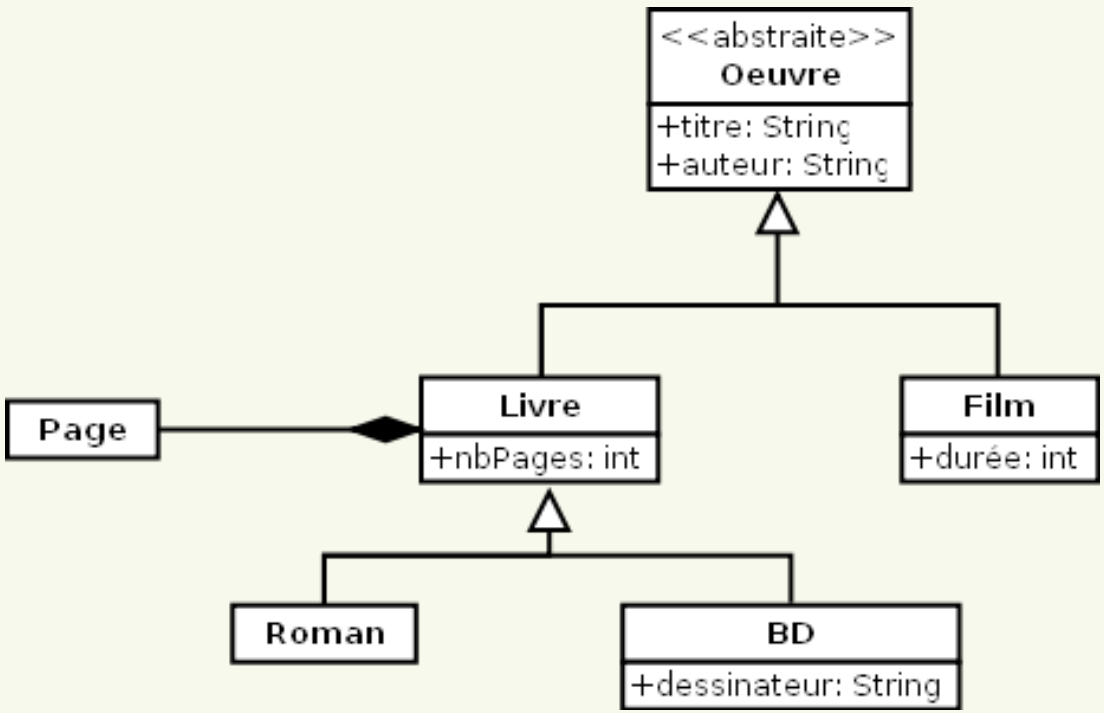
Description

On souhaite modéliser des œuvres, qui sont soit des bandes dessinées, des romans, ou des films. Ces œuvres ont un titre et un auteur. Les livres ont un certain nombre de pages, et une bande dessinée possède en plus un dessinateur. Les films, eux, ont une durée bien définie.

Réaliser le diagramme de classes correspondant.

Exercice

Une solution possible



Les interfaces

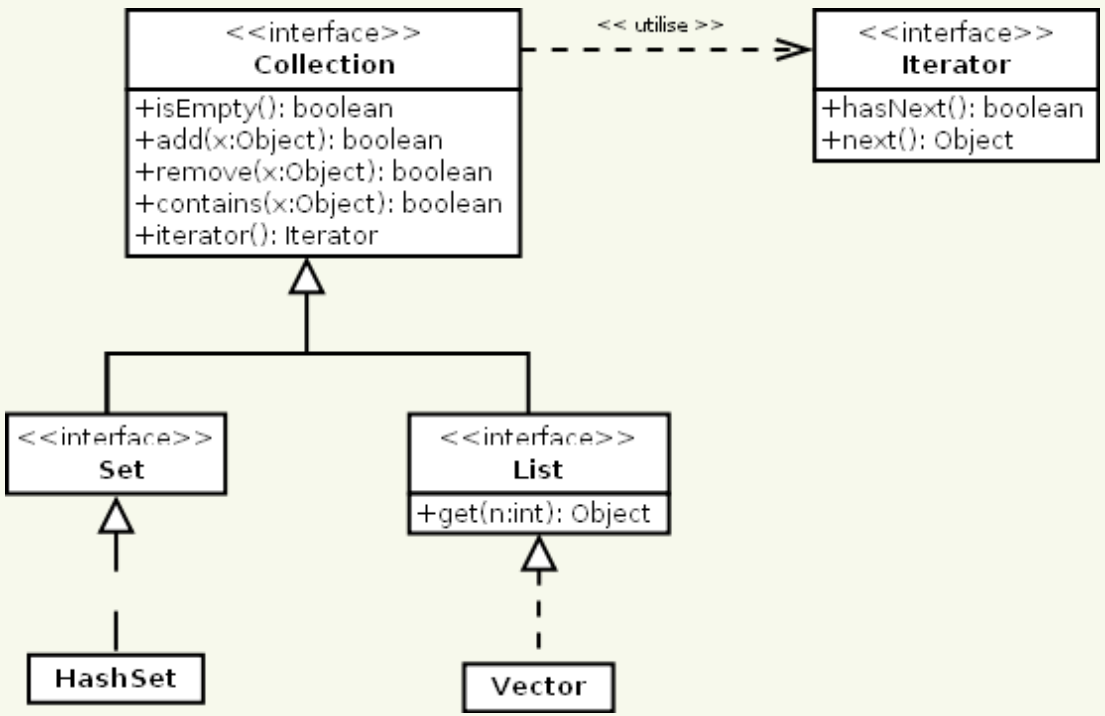
Une interface spécifie un ensemble d'opérations qui constituent un ensemble cohérent. Elle contient uniquement des opérations abstraites, sans implantation.

Une interface est équivalente à une classe abstraite qui ne contient que des opérations abstraites.

Une classe *implante* (ou *implémente* ou *réalise*) une interface lorsqu'elle fournit toutes les opérations de l'interface.

Exemple d'interfaces

Les collections en Java



On note un lien d'utilisation entre les deux interfaces, et deux liens de réalisation (HashSet implante Set et Vector implante List).

Du diagramme de classes au code Java ...

Objectif : produire du code Java à partir d'un diagramme de classes UML

Etapes de traduction

1. Traduction des classes et des interfaces

⇒ avec les attributs, les opérations, etc.

2. Traduction des associations entre les classes

⇒ Dépendante des multiplicités

Traduction des classes

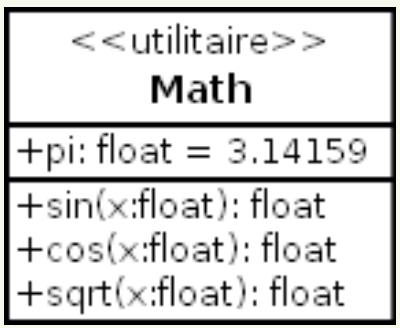
Les notions de classe, d'interface, d'attributs et d'opérations sont très similaires en UML et en Java.

Equivalences Java ↔ UML

classe UML	classe Java
classe abstraite UML	classe abstraite Java
interface UML	interface Java
attribut UML	attribut Java
opération UML	méthode Java

Exemple de traduction

Classe utilitaire *Math*




```
class Math {
  // Constructeur privé
  private Math() { }

  // attribut final (pi est une constante)
  static final float pi = 3.1415926535f ;

  static float sin(float x) { ... }
  static float cos(float x) { ... }
  static float sqrt(float x) { ... }
}
```

Traduction des relations binaires

Plusieurs choix sont possibles, dépendant :

- de la navigabilité de la relation
- des multiplicités
- de l'évolution de la relation au cours de l'exécution

La question ...

... de la navigabilité

- A $\rightarrow$ B : on doit pouvoir accéder, à partir d'une instance de A, aux instances de B qui lui sont liées.
- A $\leftarrow$ B : on doit pouvoir accéder, à partir d'une instance de B, aux instances de A qui lui sont liées.
- A $\leftrightarrow$ B : on doit à la fois pouvoir accéder aux instances de B à partir de A et aux instances de A à partir de B

$\Rightarrow$ Ajout d'attributs pointant sur l'/les objet(s) lié(s) dans les classes "source" des associations navigables
$\Rightarrow$ Nommage des attributs avec le rôle du côté de la destination de l'association navigable

La question ...

... des multiplicités

On distintingue plusieurs sortes de multiplicités

- Multiplicité simple : au plus une instance de B est associée à une instance de A. (ie. multiplicité "1" ou "0..1")
- Multiplicité de cardinalité fixe : l'ensemble des instances de B associées à une instance de A est un entier n où n est fixé (par ex. $n = 5$)
- Multiplicité à cardinalité variable : l'ensemble des instances de B associées à une instance de A a une cardinalité variable (par ex. $M..N$, $0..*$, etc.)

$\Rightarrow$ Contraintes d'intégrité sur les liens différentes à prendre en compte.

La question ...

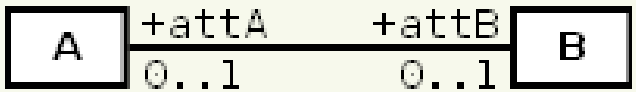
... de l'évolution des liens au cours de l'exécution du programme

Au cours de l'exécution du système des instances se créeent, tout comme les liens entre ces instances. Ces liens peuvent :

- soit être créés une fois pour toutes, et ne plus être modifiés par la suite
  $\Rightarrow$ les liens sont créés au moment de la création des objets, c'est le constructeur qui peut créer les liens.
- soit évoluer au cours de l'exécution, par exemple, être créés, supprimés, déplacés.
  $\Rightarrow$ les classes fournissent les méthodes permettant l'ajout, la suppression ou la modification d'un lien.

Exemple 1 : multiplicité simple

Situation



Une instance de A (resp. de B) peut-être soit isolée, soit liée à une instance de B (resp. de A).

Principe de traduction

- pour coder un objet $b \in B$ associé à un objet $a \in A$, on utilise un attribut *attB* de type B dans la classe A. Cet attribut a pour valeur *null* pour les instances isolées de A.
- pour coder un objet $a \in A$ associé à un objet $b \in B$, on utilise un attribut *attA* de type A dans la classe B. Cet attribut a pour valeur *null* pour les instances isolées de B.

Exemple 1 : multiplicité simple

Codage en Java

```
public class A {
   public B attB;
}

public class B {
   public A attA;
}
```

Problème potentiel :

```
A a1 = new A();
A a2 = new A();
B b1 = new B();
a1.attB = b1;
b1.attA = a2;
a2.attB = null;
```

$\Rightarrow$ cohérence des liens non respectée !

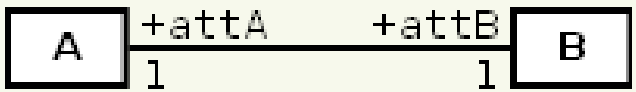
Contraintes à respecter :

$$\forall a \in A, \quad a.attB \neq null \Rightarrow a.attB.attA = a$$

$$\forall b \in B, \quad b.attA \neq null \Rightarrow b.attA.attB = b$$

Exemple 2 : multiplicité simple

Situation



Même représentation que précédemment.

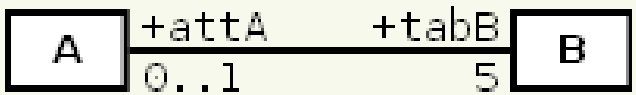
Contraintes à respecter :

$$\forall a \in A, \quad a.attB \neq null \wedge a.attB.attA = a$$

$$\forall b \in B, \quad b.attA \neq null \wedge b.attA.attB = b$$

Exemple 3 : multiplicité de cardinalité fixe

Situation



Utilisation d'un tableau tabB pour stocker les instances de B associées à l'instance courante de A.

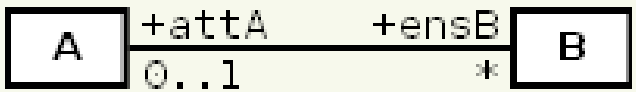
Contraintes à respecter :

$$\forall a \in A, \quad a.tabB \neq null \wedge (\forall i, a.tabB[i] \neq null \wedge a.tabB[i].attA = a)$$

$$\forall b \in B, \quad b.attA \neq null \Rightarrow (\exists i, b.attA.tabB[i] = b)$$

Exemple 4 : multiplicité de cardinalité variable

Situation



Utilisation d'un ensemble ensB pour stocker les instances de B associées à l'instance courante de A.

Contraintes à respecter :

$$\forall a \in A,$$
$$a.ensB \neq null \wedge$$
$$(\forall o \in a.ensB, o \neq null) \wedge$$
$$(\forall b \in a.ensB, b.attA = a)$$

$$\forall b \in B, \quad b.attA \neq null \Rightarrow b \in b.attA.ensB$$

Quelques notions d'OCL

UML formalise l'expression des contraintes avec OCL (Object Constraint Language).

- OCL est une contribution d'IBM à UML 1.1.
- Ce langage formel est volontairement simple d'accès et possède une grammaire élémentaire (OCL peut être interprété par des outils).
- OCL permet ainsi de limiter les ambiguïtés du langage naturel, tout en restant accessible.
- OCL permet de décrire des invariants dans un modèle, sous forme de pseudo-code :
  - pré et post-conditions pour une opération,
  - expressions de navigation,
  - expressions booléennes, etc...

OCL permet d'enrichir ce diagramme, en décrivant toutes les contraintes et tous les invariants du modèle présenté, de manière normalisée et explicite (à l'intérieur d'une note rattachée à un élément de modélisation du diagramme).

Illustration du besoin d'OCL

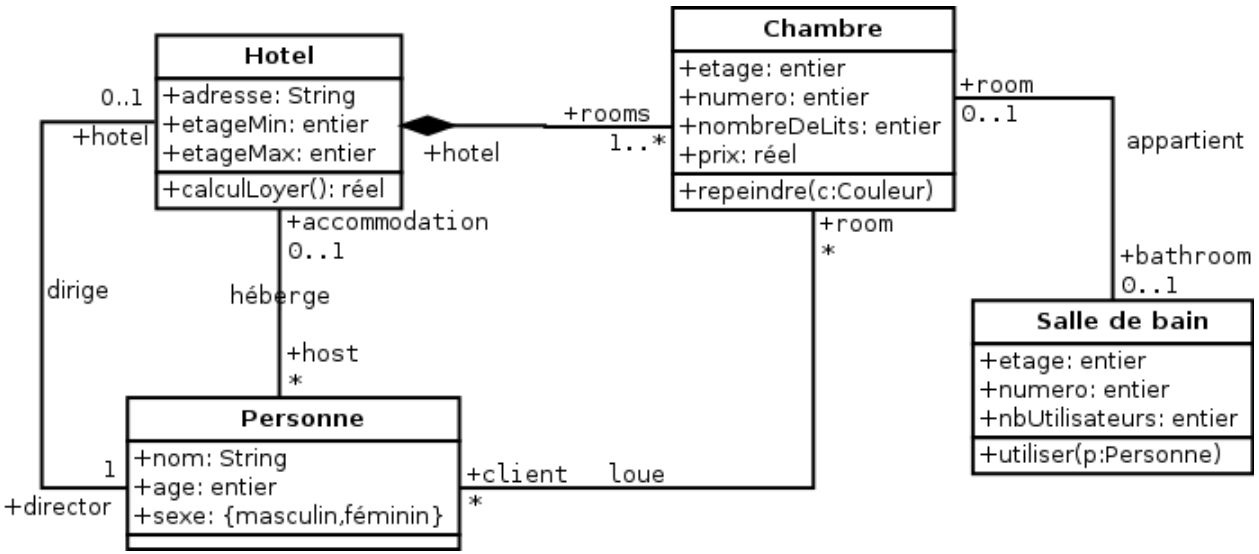
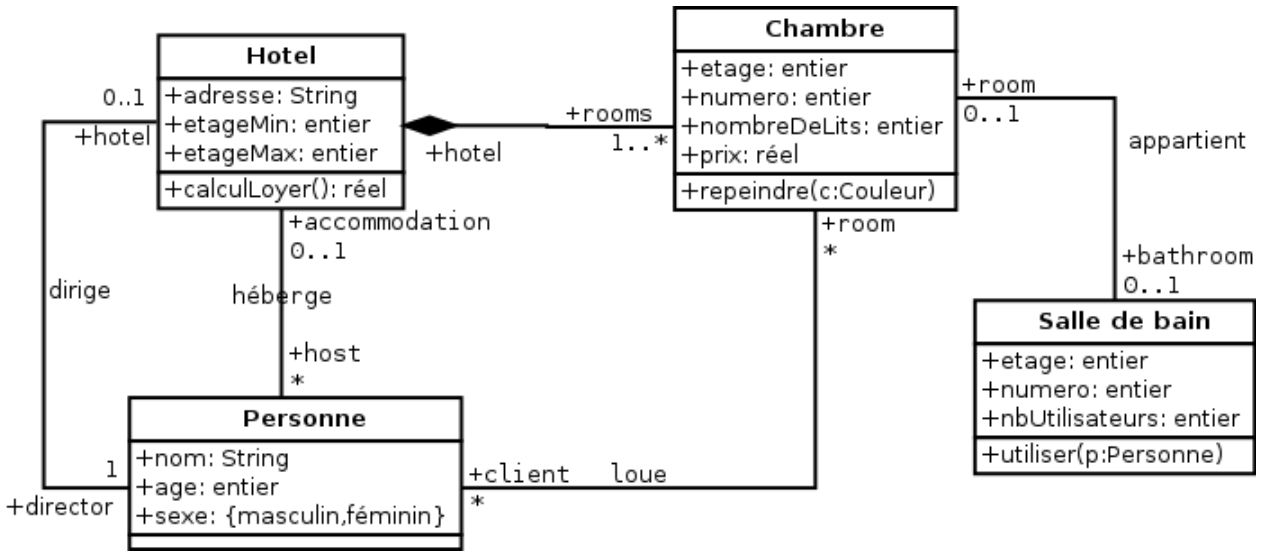
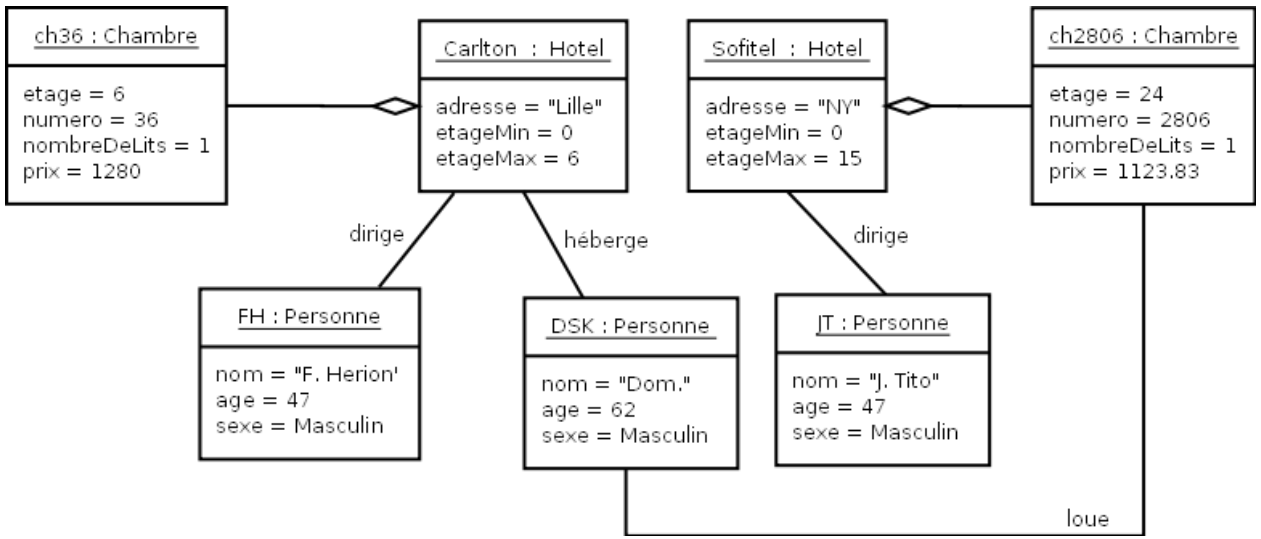


Illustration du besoin d'OCL



Un client peut-il louer une chambre d'un hôtel qui ne l'héberge pas ?

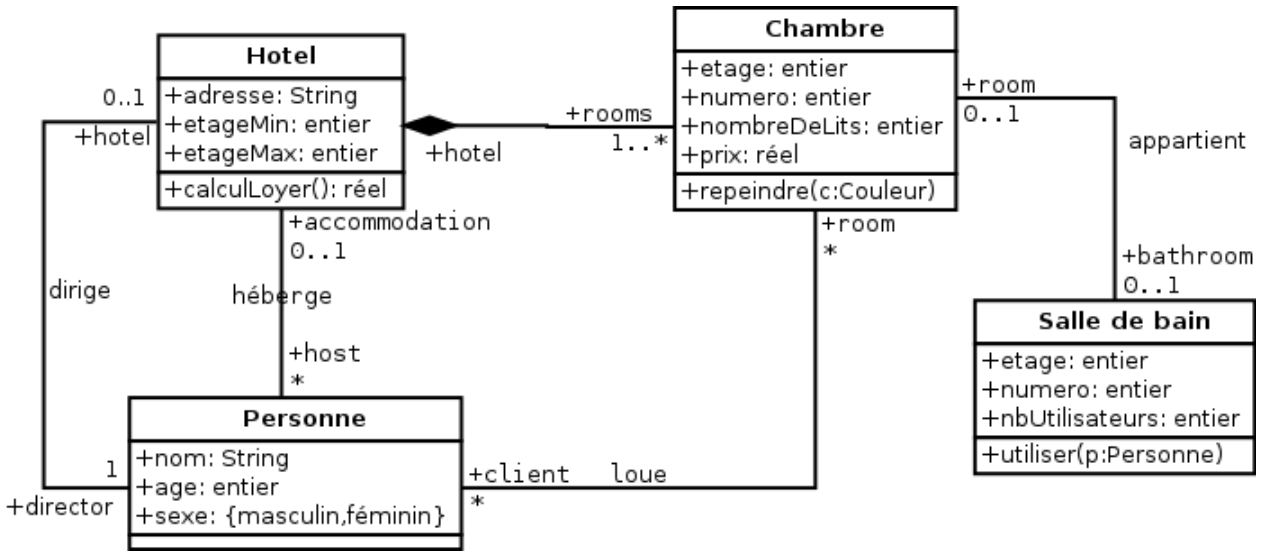
Illustration du besoin d'OCL



Un client peut-il louer une chambre d'un hôtel qui ne l'héberge pas ?

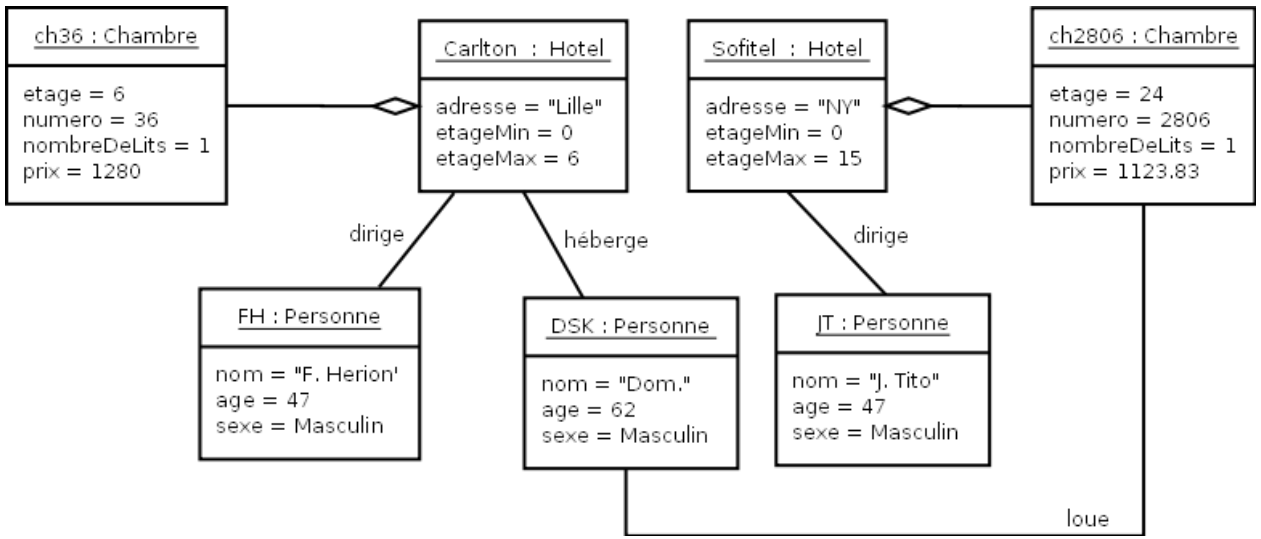
Oui, c'est tout à fait possible. Les associations formant un cycle sont généralement source de possibles incohérences sémantiques qu'il est impossible de corriger sans donner d'informations supplémentaires.

Illustration du besoin d'OCL



Une chambre peut-elle avoir un numéro d'étage incohérent par rapport à l'hôtel auquel elle est rattachée ?

Illustration du besoin d'OCL



Une chambre peut-elle avoir un numéro d'étage incohérent par rapport à l'hôtel auquel elle est rattachée ?

Oui c'est possible. Les relations entre les valeurs d'attributs d'instances liées sont impossibles à exprimer avec le diagramme de classe seul.

Syntaxe des contraintes OCL (1/5)

Une contrainte s'applique sur tous les objets de la classe considérée.
Le contexte définit la portée de la contrainte. Il peut s'agir :

- d'une classe : la contrainte devra alors être un invariant, précisé par `inv`

```
context MaClasse
    inv: ...
```

- d'une opération : la contrainte pourra alors être une précondition (`pre`) ou une postcondition (`post`).

```
context MaClasse :: MonOperation(i : Integer)
    pre: ...
    post: ...
```

Les découpages sont possible à l'intérieur d'un même contexte. La conjonction des contraintes proposées sera alors considérée.

Syntaxe des contraintes OCL (2/5)

Les invariants, préconditions et postconditions expriment des contraintes qui sont des prédicats de logique du premier ordre portant sur :

- l'instance courante de la classe, `self`
- les attributs d'une instance, `instance.att` (`self.att` peut s'écrire `att`)
- les collections d'objets associés aux classes `instance.role`
- les valeurs à l'état avant l'opération dans les postconditions uniquement, `att@pre`
- les paramètres d'entrée ou de sortie des opérations, pour les pré- et postconditions
- la valeur de retour d'une opération, `result`

Syntaxe des contraintes OCL (2/5)

Quelques opérateurs de manipulation des données :

- entiers, réels : `=`, `<>`, `<=`, `>=`, `+`, `-`, `*`, `/`, `x.div(y)`, `x.mod(y)`
- chaines : `s.concat(t)`, `s.size()`, `s.subString(debut,fin)`
- booléens : `and`, `or`, `not`, `xor`, `=`, `implies`, `if ... then ... else ... endif`
- objets : `obj.isOclUndefined()` (eq. à `obj = null`), `obj.isOclInstance(Classe)`, `obj.isOclNew()`

Syntaxe des contraintes OCL (3/5)

On compte quatre type de collections :

- `Set` : pas de doublons, pas d'ordre
- `Bag` : doublons possibles, pas d'ordre
- `OrderedSet` : pas de doublons, ordonné
- `Sequence` : doublons possibles, pas d'ordre

L'expression `instance.role` renverra différents types en fonction des multiplicités du côté du `role` :

- multiplicité 0 ou 1, `instance.role` est un objet (éventuellement la valeur `null` si aucun objet n'est associé à `instance`)
- multiplicité N (> 1) ou M..N, `instance.role` est un `Set` (pas de doublons, pas d'ordre).

Syntaxe des contraintes OCL (4/5)

Ces opérations s'appliquent en utilisant l'opérateur `->` suite au nom de la collection.
Quelques opérations utiles :

- `coll->size()` : compte le nombre d'objets d'une collection (renvoie un entier)
- `coll->sum()` : somme les éléments d'une collection de valeurs numériques (renvoie un réel)
- `coll->isEmpty()` : équivalent à `coll->size() = 0`
- `coll->notEmpty()` : équivalent à `coll->size() <> 0`
- `coll->count(obj)` : compte le nombre d'occurrences de `obj` dans `coll`
- `coll->includes(obj)` : équivalent à `coll->count(obj) > 0`
- `coll->excludes(obj)` : équivalent à `coll->count(obj) = 0`
- `coll->includesAll(coll2)` : vérifie si tous les éléments de `coll2` sont inclus dans `coll`

Syntaxe des contraintes OCL (5/5)

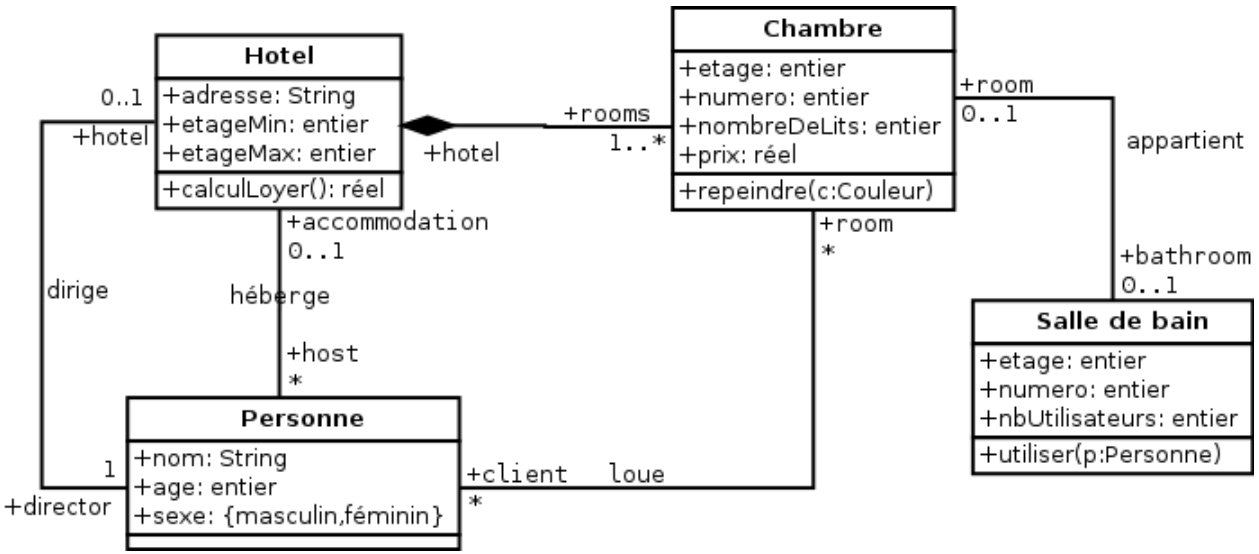
Ces opérations permettent d'effectuer des requêtes sur les collections.
Quelques opérations utiles :

- `coll->collect(role)` : renvoie un `Bag` contenant les objets liés à ceux de `coll` par le rôle `role`
- `coll->exists(expr)` : vérifie si `expr` est satisfait pour au moins un élément de la collection (booléen)
- `coll->forAll(expr)` : vérifie si `expr` est satisfait pour tous les éléments de la collection (booléen)
- `coll->select(expr)` : renvoie la collection composée des éléments de `coll` qui satisfont `expr`

Il est possible de définir des attributs ou des opérations temporaires afin de simplifier et compacter les expressions. `context` *MaClasse*

```
inv: let val1:Boolean = self.att1 >= 1 and self.att1 <= 140
     let majeur(age : Integer):Boolean = age >= 18 in
     val1 and (self.attr2 = 42 or majeur(self.attr3)
```

Quelques exemples de spécifications en OCL

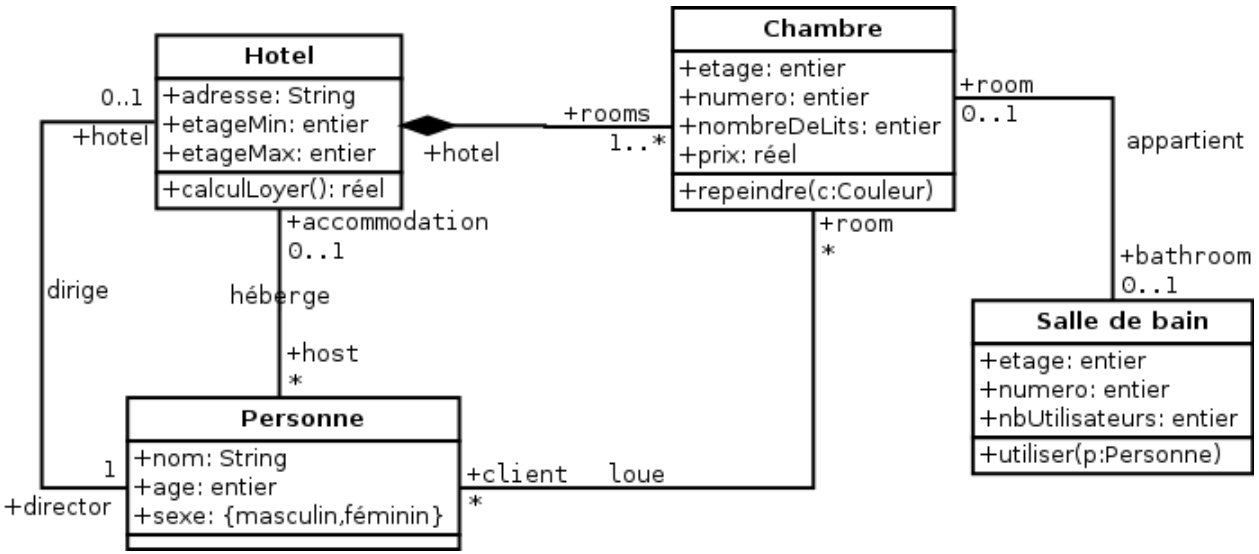


Un hôtel ne contient jamais d'étage numéro 13 (superstition oblige)

```
context Chambre inv:
self.etage <> 13
```

```
context SalleDeBain inv:
self.etage <> 13
```

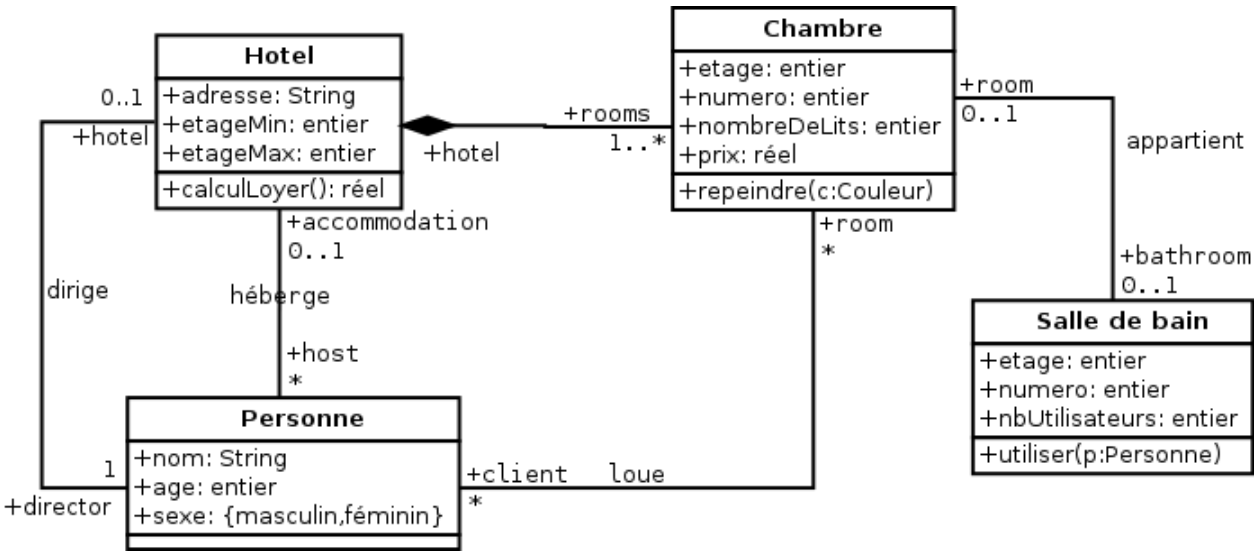
Quelques exemples de spécifications en OCL



On ne peut repeindre une chambre que si elle n'est pas louée. Une fois repeinte, une chambre coûte 10% de plus.

```
context Chambre::repeindre(c : Couleur)
pre: client->isEmpty()
post: prix = prix@pre * 1.1
```

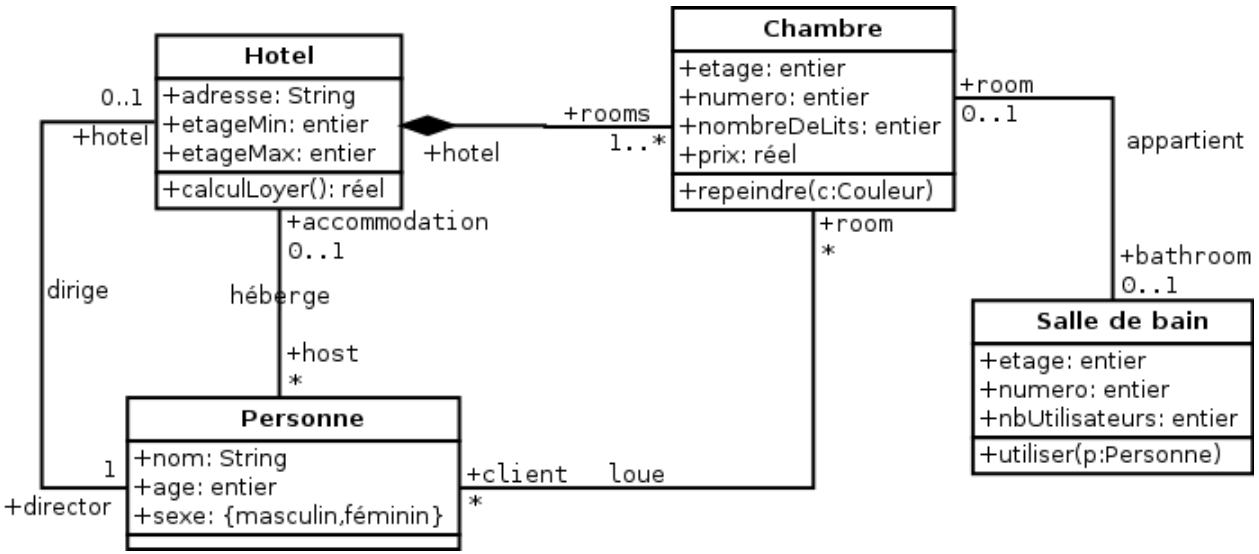
Quelques exemples de spécifications en OCL



L'étage de chaque chambre est compris entre le premier et le dernier étage de l'hôtel.

```
context Hotel inv:
self.rooms->forAll(c : Chambre | c.etage <= self.etageMax and
c.etage >= self.etageMin)
```

Quelques exemples de spécifications en OCL



Une personne ne peut pas avoir loué un chambre dans un hôtel qui ne l'a pas hébergé

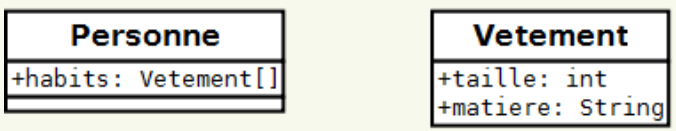
```
context Personne
inv: self.room->forall(c : Chambre |
self.accommodation <> null implies
self.accommodation->rooms->contains(c))
```

Quelques erreurs classiques de modélisation

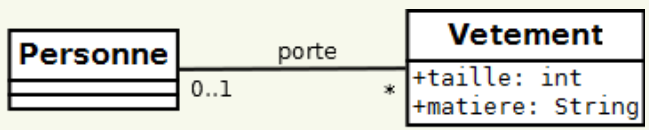
Types des attributs de classe

Si le modèle est bien fait, les attributs des classes présentées sont tous de type basique/prédéfini (entier, réel, booléen, caractère, chaîne). Les classes ne doivent pas contenir d'attributs typés comme des objets d'une autre classe (sauf si cette dernière est une classe d'énumération).

Typage d'un attribut



Incorrect : attribut typé comme un objet



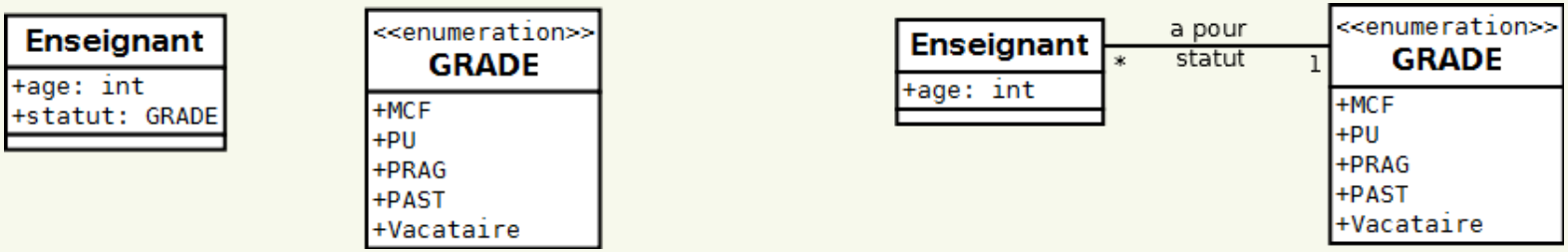
Correct : utilisation d'une association

Quelques erreurs classiques de modélisation

Types des attributs de classe

Si le modèle est bien fait, les attributs des classes présentées sont tous de type basique/prédéfini (entier, réel, booléen, caractère, chaîne). Les classes ne doivent pas contenir d'attributs typés comme des objets d'une autre classe (sauf si cette dernière est une classe d'énumération).

Typage d'un attribut avec une énumération

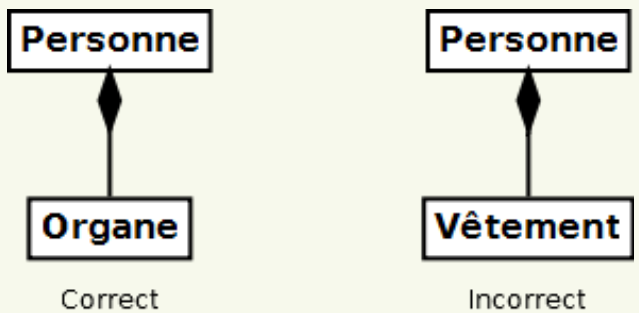


Quelques erreurs classiques de modélisation

Agrégation/compostion vs. association simple

L'agrégation (ou la composition) indique une relation entre "un tout" et "une partie". On doit retrouver entre ces deux entités une relation de type composant/composé.

Agrégation/composition vs. association



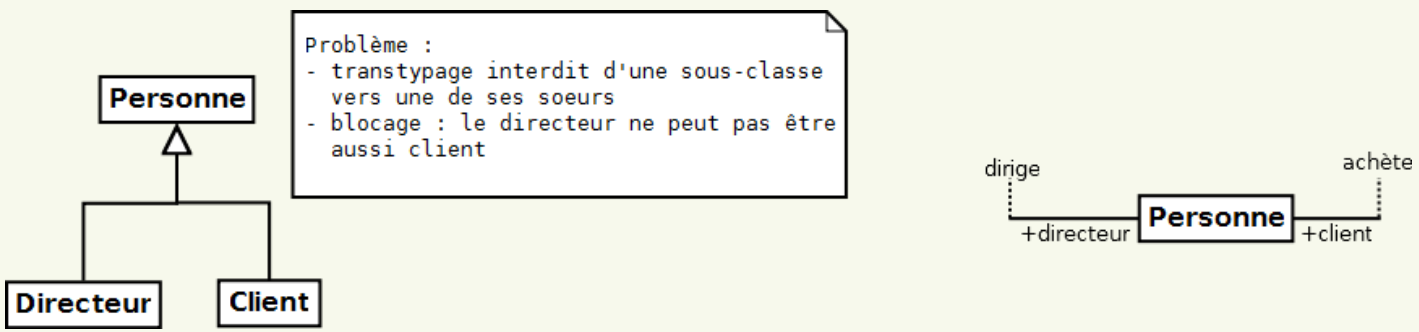
En cas de doute, une association "simple" peut toujours remplacer une agrégation/composition.

Quelques erreurs classiques de modélisation

Analyse vs. Conception

Un système (le diagramme de classe) est amené à être mis en pratique et à faire évoluer son état (le diagramme d'objet) au cours du temps. Il faut faire attention à ne pas figer une hiérarchie de classe, car un objet ne peut pas changer de classe pour se transtyper vers une classe soeur.

Analyse vs. conception



Quelques erreurs classiques de modélisation

Erreurs classiques de modélisation objet

- Croire que deux objets avec les mêmes valeurs d'attibuts sont identiques
- Ne pas nommer les associations dans les diagrammes de classe
- Ne pas mettre de multiplicités sur une association
- Ne pas nommer les liens dans les diagrammes d'objet
- Ne pas mettre de valeurs aux attributs des objets
- Ne pas respecter le (peu de) formalisme UML existant
- Mettre des multiplicités sur les relations d'héritage
- Mettre un attribut du type X dans une classe et associer celle-ci à la classe X
- Mettre des agrégations/compositions partout ...

Plan du cours

Diagrammes de composants

Le but principal du diagramme de composants est de montrer les relations structurelles entre les composants d'un système.

En UML 1.1, les diagrammes de composants étaient exclusivement dédiés à la description de l'architecture physique d'un système en terme de fichiers tels que des bibliothèques ou des exécutables.

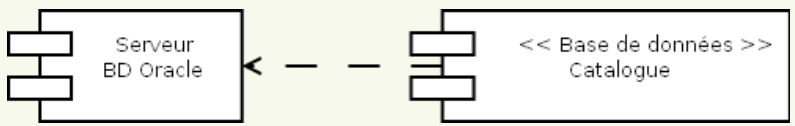
A partir d'UML 2.0, les diagrammes de composants permettent de spécifier de manière plus fine l'architecture logique d'un système en termes de modules, par exemple implantés dans des classes (dont la structure est décrite dans le diagramme de classe), et leurs interactions en termes d'interfaces fournies ou requises.

Diagrammes de composants

Les diagrammes de composants peuvent permettre de décrire l'architecture physique d'une application en termes de fichiers.

Diagramme de composants

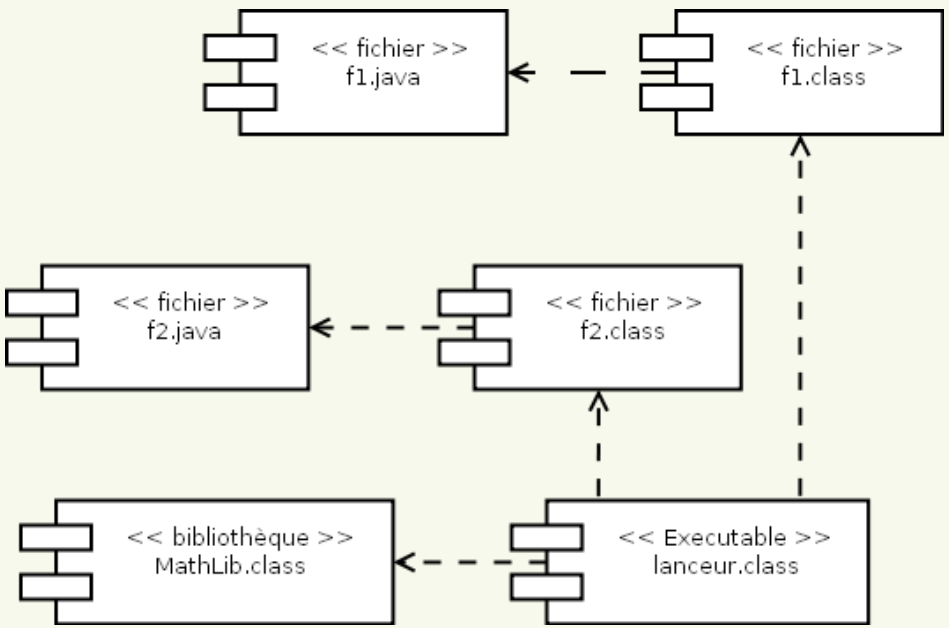
Composants d'un serveur Oracle utilisant une base de données servant de catalogue.



Diagrammes de composants

Composants

Application construite à partir de deux fichiers source et d'une bibliothèque mathématiques.



Diagrammes de composants

Les diagrammes de composants peuvent également servir à décrire la composition de composants logiques constituant une application ou un composant au sens plus large.

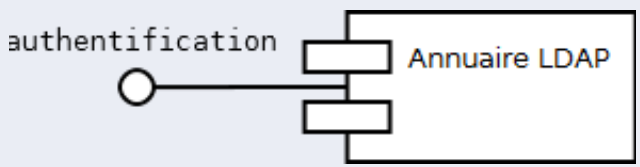
Dans ce cas, un composant est une classe structurée représentant une partie modulaire du système, encapsulant un certain contenu. Chaque composant peut être remplacé par un autre composant au sein de son environnement.

Le comportement d'un composant est ainsi défini en termes d'interface *fournies* et *requises*.

Diagrammes de composants

Interfaces fournies

Une interface fournie par un composant est un point de connexion offert par le composant pour réaliser une fonctionnalité (service, calcul, etc.).



Diagrammes de composants

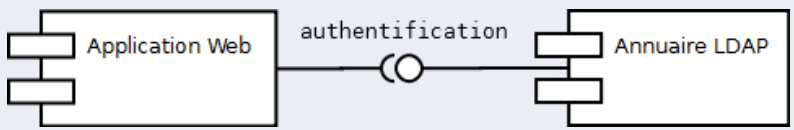
Interfaces requises

Une interface requise par un composant est un point de connexion offert au composant par un autre pour réaliser une fonctionnalité (service, calcul, etc.).

Diagrammes de composants

Composition de composants

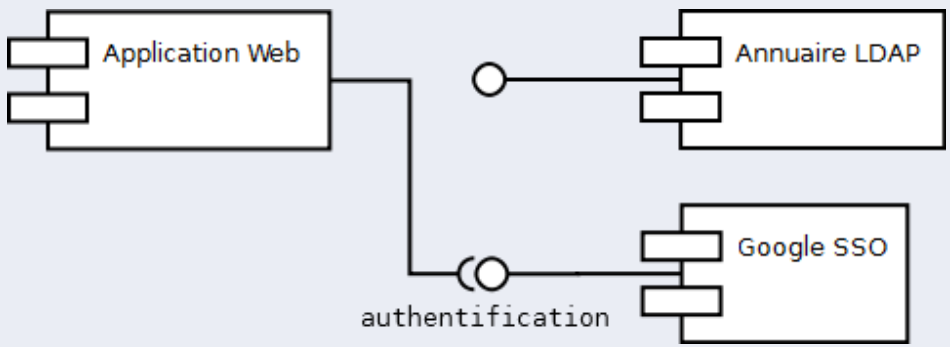
Les composants se composent entre eux par connexion des interfaces fournies/requises compatibles (sémantiquement).



Diagrammes de composants

Composition de composants

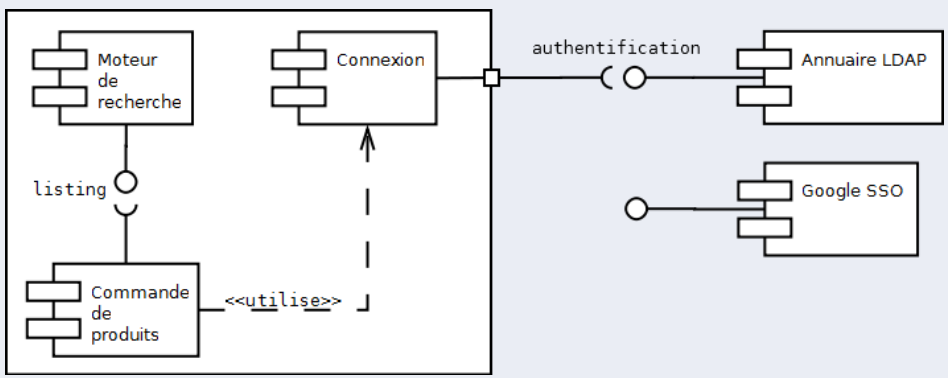
Les composants sont substituables. On peut donc remplacer un composant, par un autre qui propose des services compatibles.



La vérification de la compatibilité des composants est laissée à la discrétion des concepteurs.

Encapsulation de composants

Un composant peut être composé d'autres composants internes. Dans ce cas, les interfaces fournies et requises par le super-composant doivent être connectées à des interfaces fournies et requises au sein des sous-composants, via des ports spécifiques.



Cet exemple fait également apparaître une dépendance d'utilisation entre deux composants.

Plan du cours

Diagrammes de classes et d'objets

Diagrammes de composants

Diagrammes de déploiement

Diagrammes de déploiement

Les diagrammes de déploiement décrivent la disposition physique des matériels et la répartition des composants sur ces matériels.

On retrouve donc dans les diagrammes de déploiement des notations issues du diagramme de composants.

Diagrammes de déploiement

Diagramme de déploiement