

## IFT 2255 – Génie logiciel

# Modélisation

# La Trahison des images (Magritte)



# Modélisation en génie logiciel

- Activité consistant à créer une **représentation simplifiée** (modèle) d'un phénomène ou d'un système
- **Modèle** est une **abstraction** du système
- Modèle permet d'étudier **structure et fonctionnement** du système
- Modélisation accompli avec différents niveaux de formalisations

# À quoi sert un modèle ?

- Approfondir compréhension du système
  - Raffinement de l'analyse et de la conception
  - Représentation d'un système existant :  
**rétro-ingénierie** (*reverse engineering*)
- Réduire la **complexité** du problème par abstraction
- Réunir et visualiser un ensemble de détails choisis
- Favoriser la communication au sein de l'équipe
- Documenter
- Plus le formalisme est précis, plus on peut l'utiliser pour **générer** l'implémentation

# UML

# Unified Modeling Language (UML)

- **Norme** de l'OMG pour la modélisation OO
  - Object Management Group
  - Standardisé ISO en 2014
- Langage majoritairement **graphique** (diagrammes) rigoureux mais non formel
- Moyen de communication qui facilite la **représentation** et la **compréhension** du logiciel
- Notations pour décrire les **exigences**, la **conception** (et même code) et le **déploiement**
- **Extensible** (via les profiles)
- **Abstrait**: indépendant des langages de programmations, domaines d'application ou processus de développement

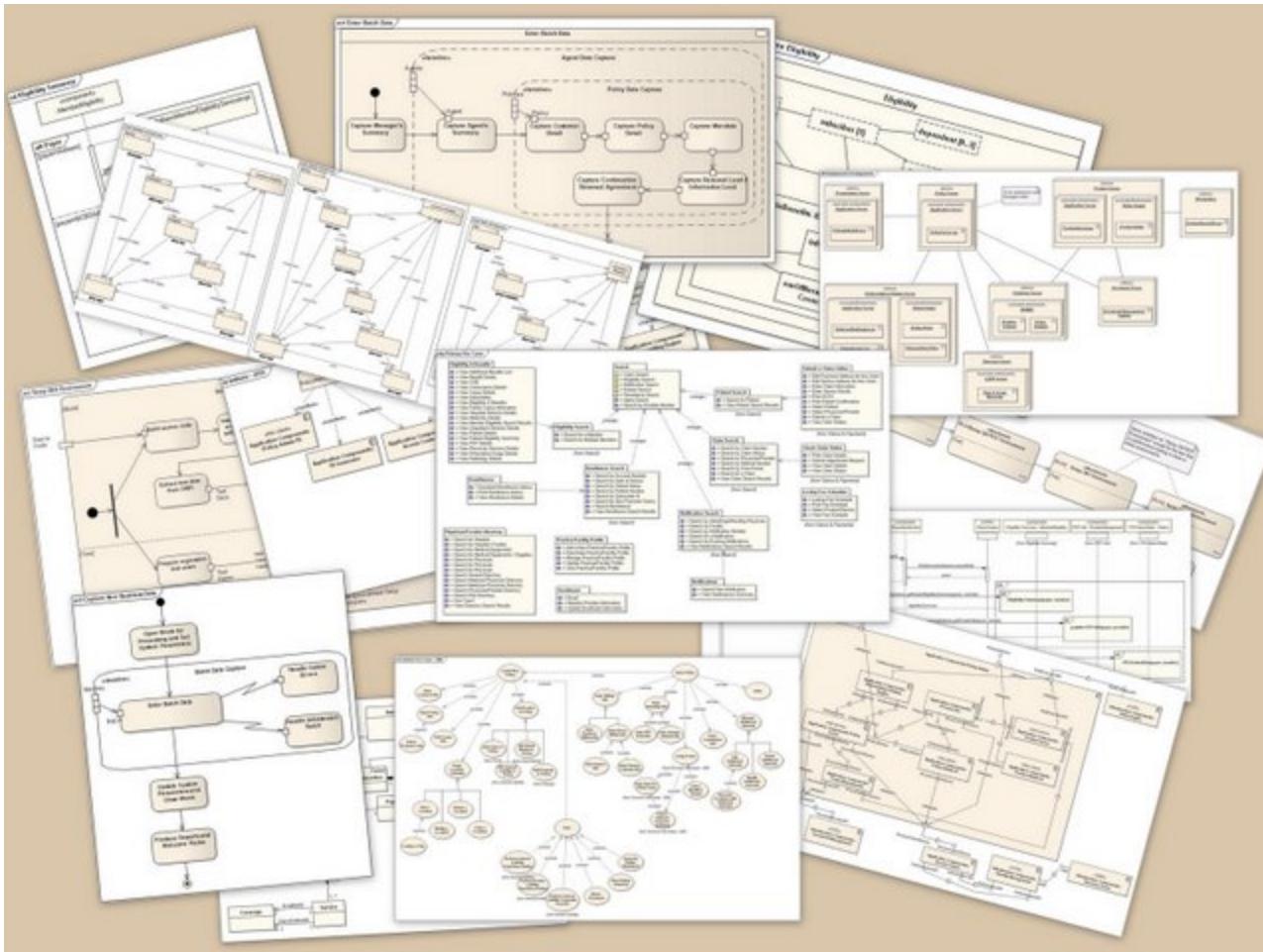
# UML n'est pas...

- Un langage de programmation
  - Peut être utilisé pour générer le code dans un langage donné
  - La modélisation est une activité d'abstraction
- Un processus
  - Le processus unifié est souvent utilisé quand on modélise en UML

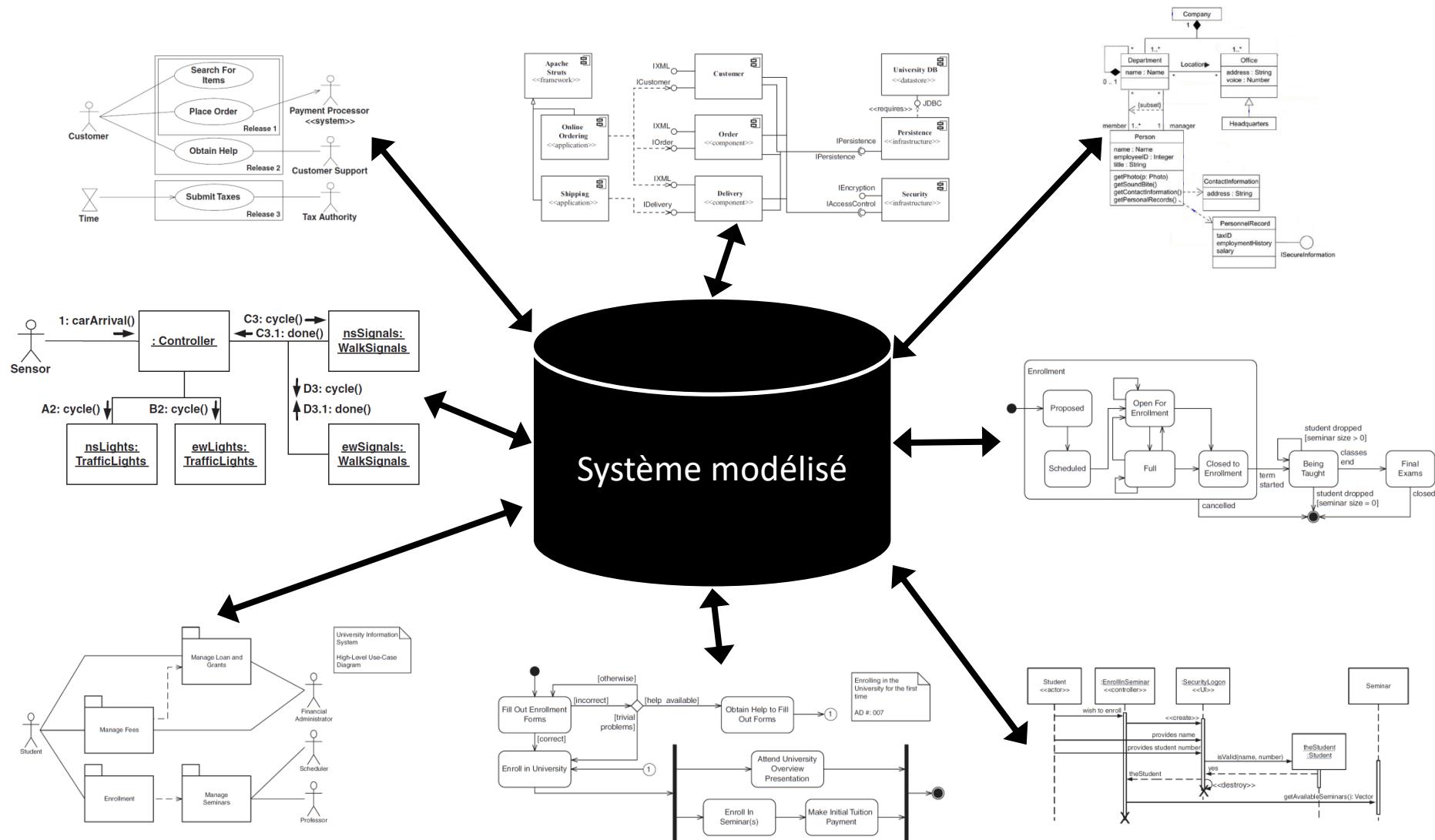
# UML comme cadre d'analyse

- L'analyse et la conception se font graduellement par l'élaboration et le raffinement de modèles
  - Pas de barrière stricte entre analyse et conception
  - Les modèles d'analyse et de conception ne diffèrent que par leur niveau d'abstraction
    - Ajout de détail en raffinant de façon **itérative et incrémentale**

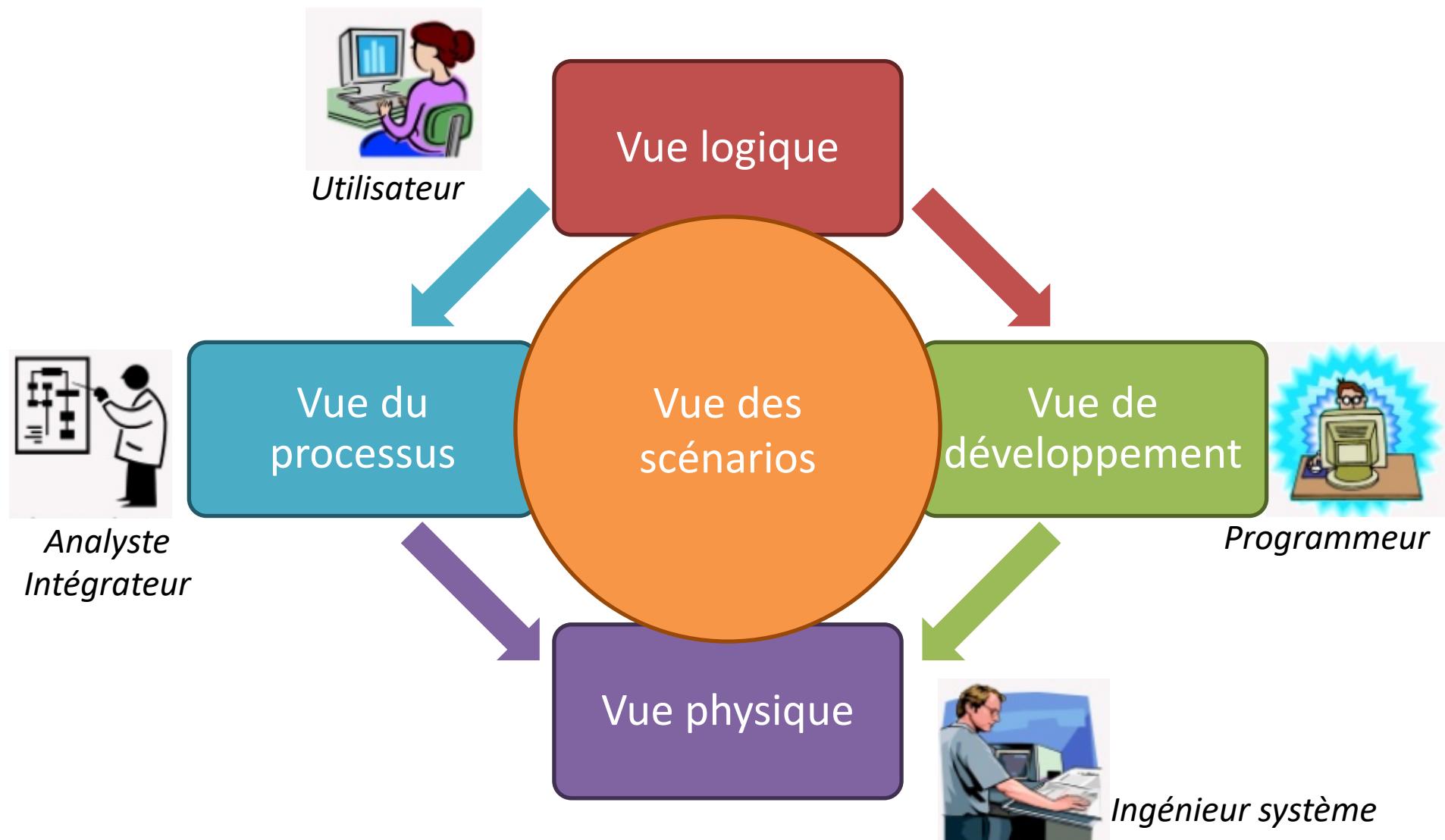
# Collection de langages



# Différentes vues sur le même système

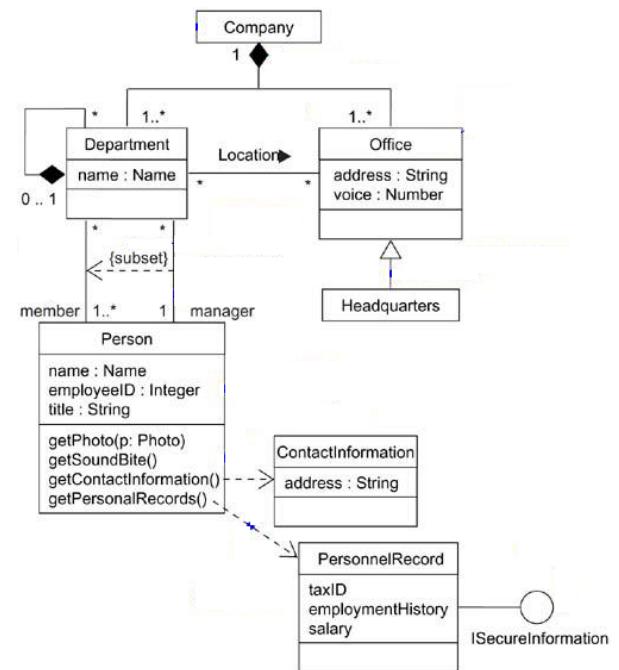


# Modèle 4+1



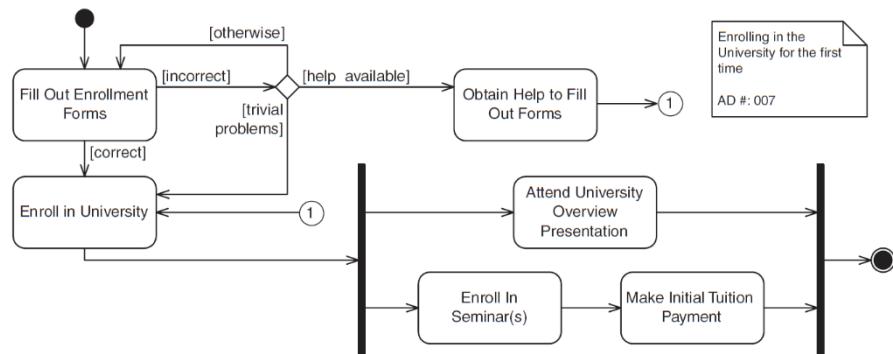
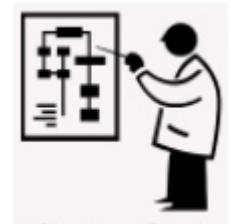
# Vue logique

- Vue de conception
- Décomposition orientée objet
- Exigences fonctionnelles: services que le système doit fournir aux utilisateurs
- Diagrammes UML impliqués
  - Diagramme de classes
  - Diagramme d'objet
  - Diagramme d'état
  - Diagramme de séquence
  - Diagramme de communication



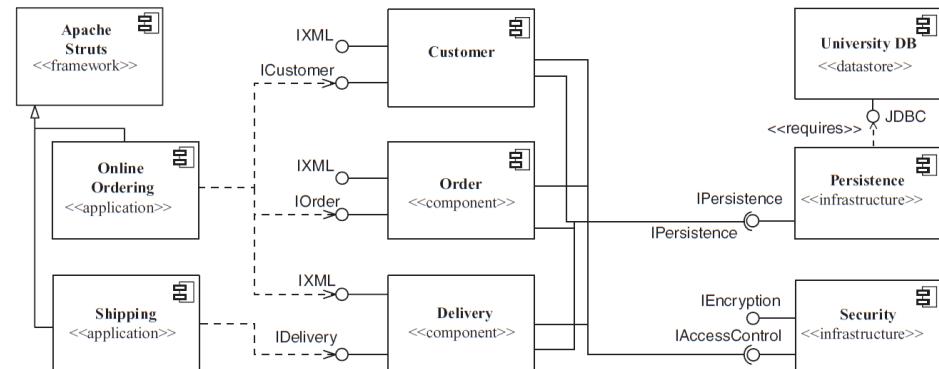
# Vue du processus

- Processus et leurs communications
- Exigences non fonctionnelles
  - Performance, scalabilité, débit du système
- Diagramme UML impliqué
  - Diagramme d'activité



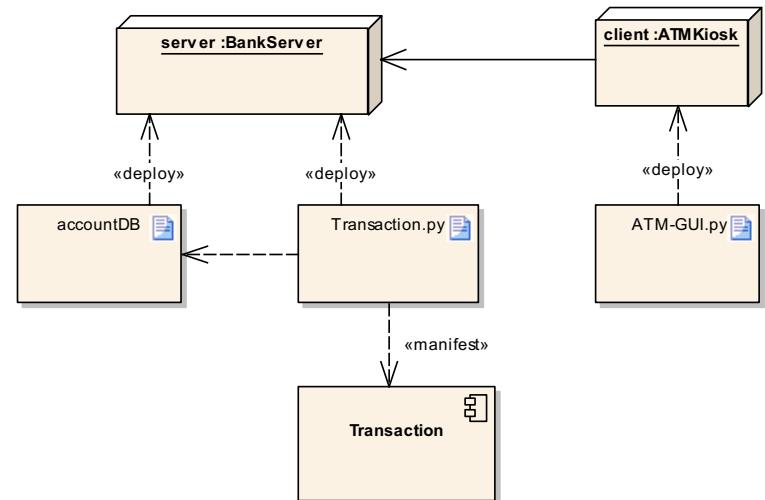
# Vue de développement

- Décomposition en sous-systèmes
- Organisation des modules
- Couches hiérarchiques, réutilisation, contraintes d'outils
- Diagrammes UML impliqués
  - Diagramme de composants
  - Diagramme de paquets



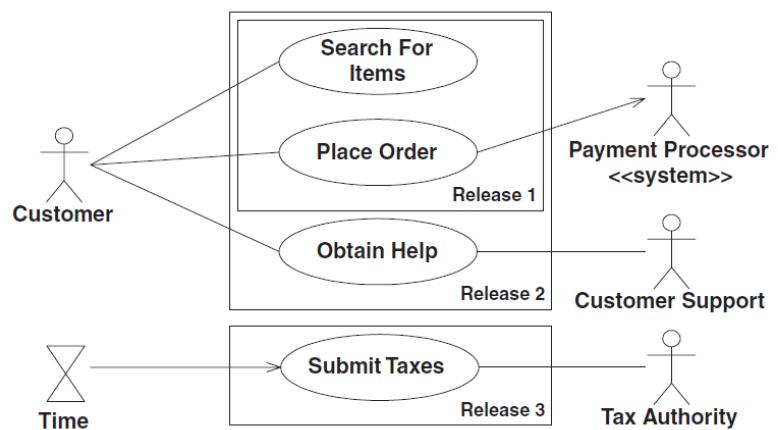
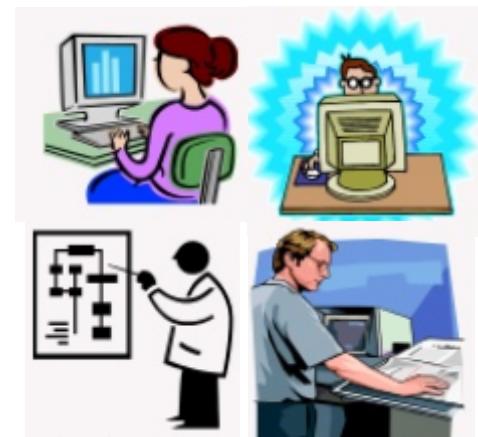
# Vue physique

- Lien entre logiciel et matériel hardware
- Exigences non fonctionnelles sur le matériel
  - Topologie, communication
- Diagramme UML impliqué
  - Diagramme de déploiement

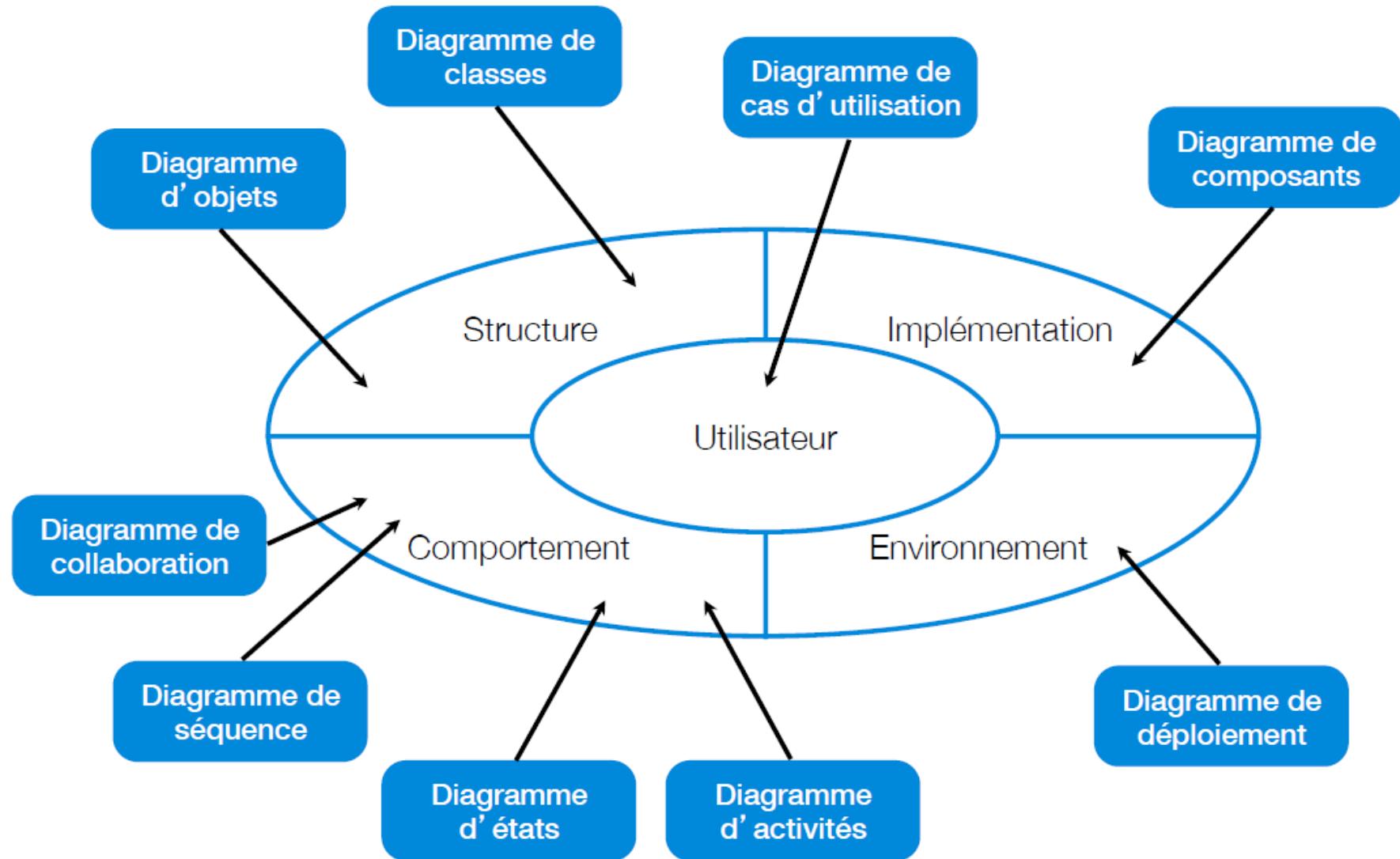


# Vue des scénarios

- Ce qui tient tout ensemble
- Cohérence du système, validité
  - Tests
- Diagramme UML impliqué
  - Diagramme de CU
  - Scénarios écrits des CUs



# Diagrammes UML 2.X



# Diagrammes utilisés dans ce cours

- Diagramme de **cas d'utilisations**
  - Exigences du système du p.d.v. des acteurs qui jouent différents rôles en interagissant avec le système
- Diagramme d'**activités**
  - Comportement dynamique du processus d'affaire, logique des procédures
- Diagramme de **classes**
  - Modélisation de la structure statique des entités et leurs relations
- Diagramme de **séquence**
  - Comment les objets communiquent entre eux au fil du temps pour réaliser chaque CU

# Diagramme de classes

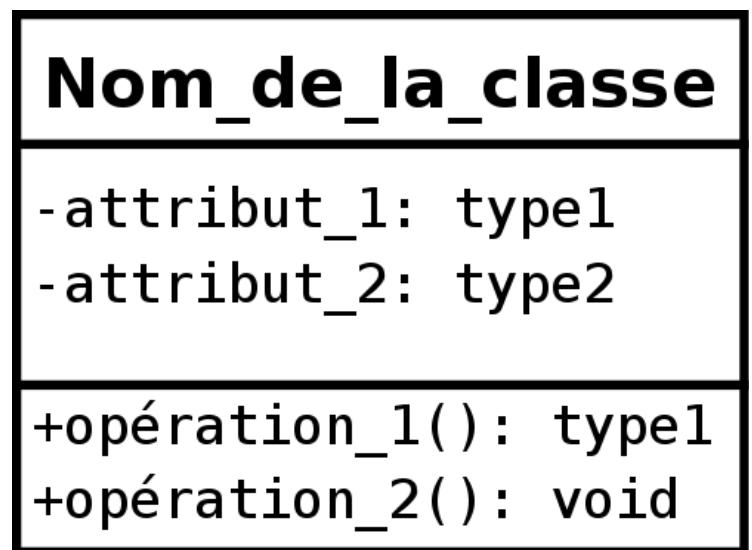
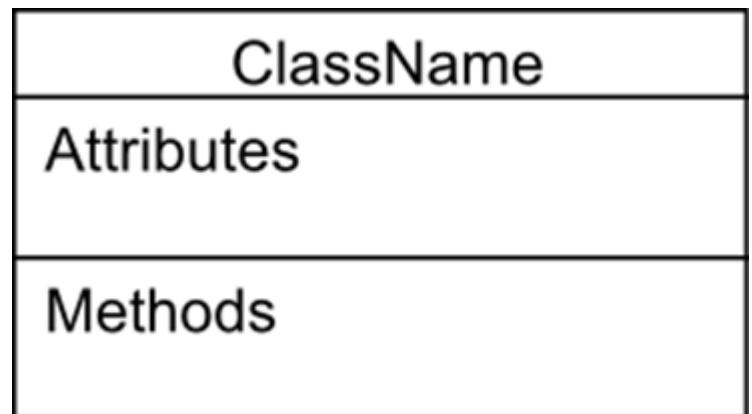
# UML

# Diagramme de classes

- Permet de décrire la structure statique du logiciel
- Indique les contraintes sur les objets et leurs relations
- **Classes** : attributs, opérations
- **Relations** : association, agrégation, composition, généralisation/spécialisation

# Représentation des classes

- Compartiment du **Nom**
  - Unique
  - Noms répétées - *alias*
- Compartiment des **Attributs**
  - Caractéristiques des objets
- Compartiment des **Operations**
  - Comportement des objets



# Paramètres

- **Attributs**

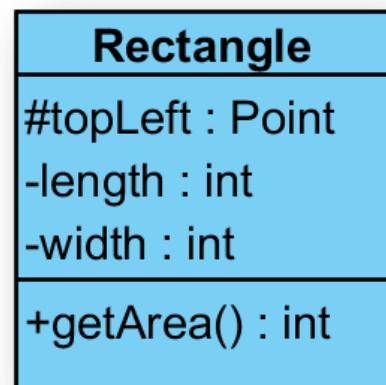
[visibilité] [portée] <nom> : <type> [= <valeur\_initiale>]

- **Opérations**

[visibilité] [portée] <nom> (<params>) : <type\_resultat>

# Visibilité

- Ensemble de préfixes pour les attributs et les opérations
  - + {Public}: élément visible par toutes les instances de toutes les classes
  - # {Protected}: élément visible par toutes les instances de la classe et de ses sous-classes
  - {Private}: élément visible que par les instances de la classe
  - ~ {Package}: élément visible par les classes du même paquet



# Portée statique

- **Attribut d'instance:** chaque instance a ces propres valeurs d'attribut
- **Attribut de classe:** toutes les instances de la classe partagent la même valeur de l'attribut
- Opération de classe: opération appliquée à tous les éléments de la classe
  - Opération statique ne dépend que d'attributs et d'opérations statiques
- Représenté par nom souligné ou avec {static}
- Portée par défaut est d'instance

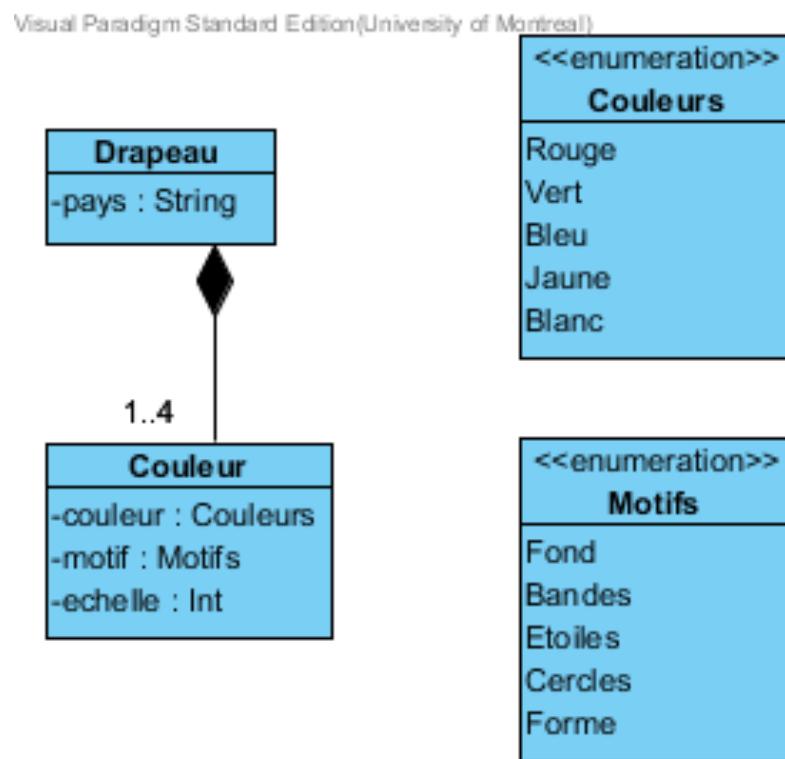
Element
-name : String -id : int = 0
+getName() : String -incrementId() : void

# Type

- primitif : **boolean, real, integer, string, date**
- Enumération: **Enum**
- Types définis par une classe ou interface
- Favoriser l'utilisation d'association si l'attribut a pour type une autre classe

# Enumération

- Déclare des types discrets ou un ensemble fini de valeurs possibles

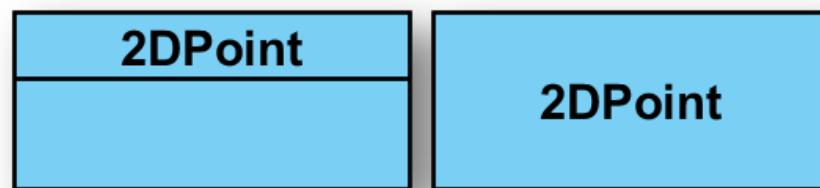
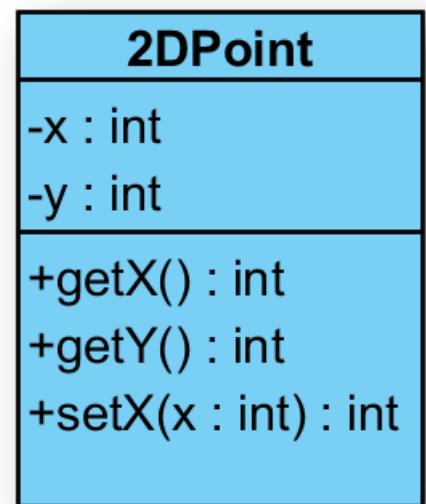


# Paramètres des Opérations

- <direction> <nom> : <type> [= valeur\_initiale]
- <direction> := in | out | inout
  - Par défaut IN

# Exemple d'une classe

- Exemple d'un point de deux dimensions
- Attributs: coordonnées
- Opérations (méthodes):  
getters et setters
- Représentations alternatives:
  - Souvent utilisé pour les alias



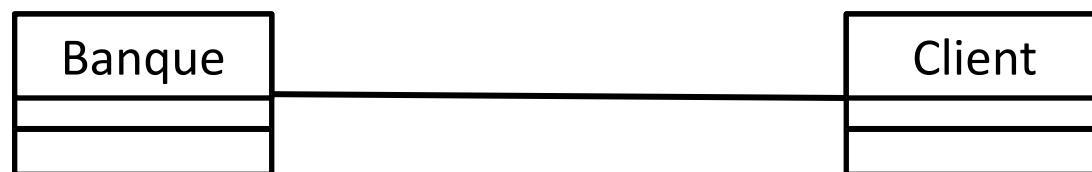
# Classes abstraites

- Les classes abstraites ne sont pas instanciables
- Les opérations abstraites ne disposent pas d'une implémentation
  - Une classe avec au moins une opération abstraite doit être abstraite
- Pour être utile, une classe abstraite doit être spécialisée
  - les classes concrètes sont forcées d'implémenter les opérations abstraites héritées
- Représenté avec *Italic* ou {abstract}
  - Utiliser {abstract} quand écrit à la main

<b>Vehicle</b>
-wheels : Wheel[4]
-engine : Engine
-position : Point
+move( <i>distance</i> : int) : void
+turn( <i>angle</i> : float) : void
+getPosition() : int

# Association

- Lien / connexion entre deux instances
  - Habituellement implémentée à l'aide d'attributs d'instances
  - Représentée visuellement par une ligne entre deux classes



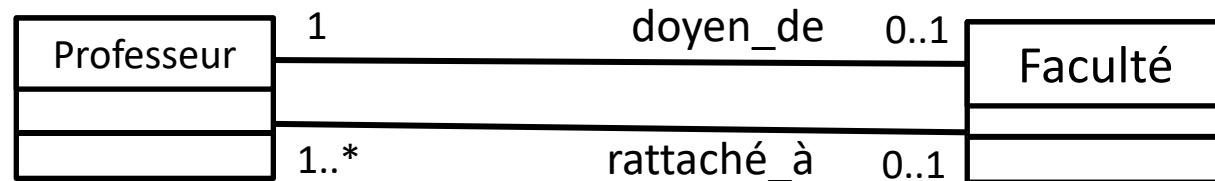
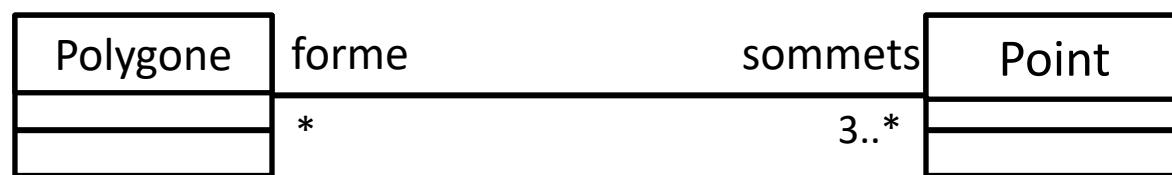
- Spécification
  - Cardinalité
  - Rôle(s)

# Cardinalité

- Précise le nombre d'instances participantes
- min..max
  - *min* et *max* sont des entiers naturels
  - *max* peut être non borné (\*)
  - si le min et le max sont égaux, utilise un seul nombre

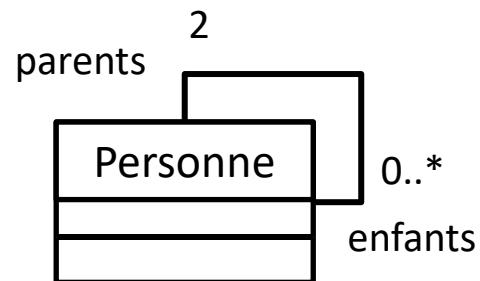
1	exactement une instance
0..1	optionnel
k	exactement <i>k</i> instances
* ou 0..*	Plusieurs optionnel
k..*	au moins <i>k</i> instances
0..k	optionnel jusqu'à k
k..j	entre <i>k</i> et <i>j</i> instances
k,j	ou <i>k</i> ou <i>j</i> instances

# Cardinalité



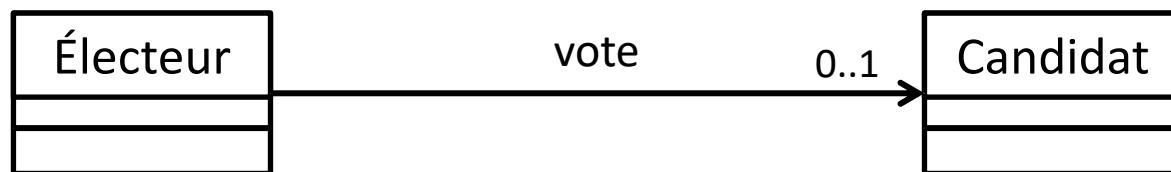
# Rôle

- Fonction sémantique que joue une classe dans une association
  - Chaque extrémité peut être annotée avec un rôle optionnel
- Information indispensable pour les associations réflexives



# Navigabilité restreinte

- Par défaut, une association est bidirectionnelle
- Restreindre la navigabilité de l'association
- Association unidirectionnelle: classe cible ne peut pas identifier la source



- À partir d'un électeur, on peut directement identifier le candidat pour lequel il a voté. À partir d'un candidat, on ne peut pas retrouver directement les électeurs qui ont voté pour lui.

# Navigation des associations

- L'objet Personne fait référence à plusieurs objets Chien



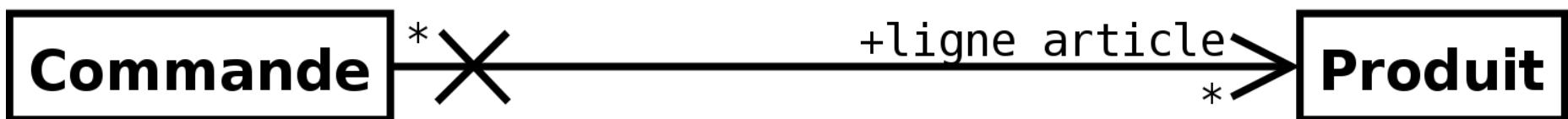
- L'objet Chien a une référence à un objet Person



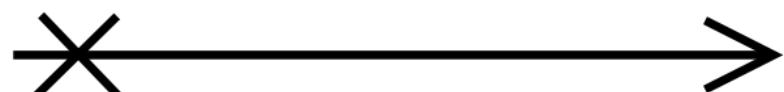
- Les deux objets ont des références à l'un à l'autre



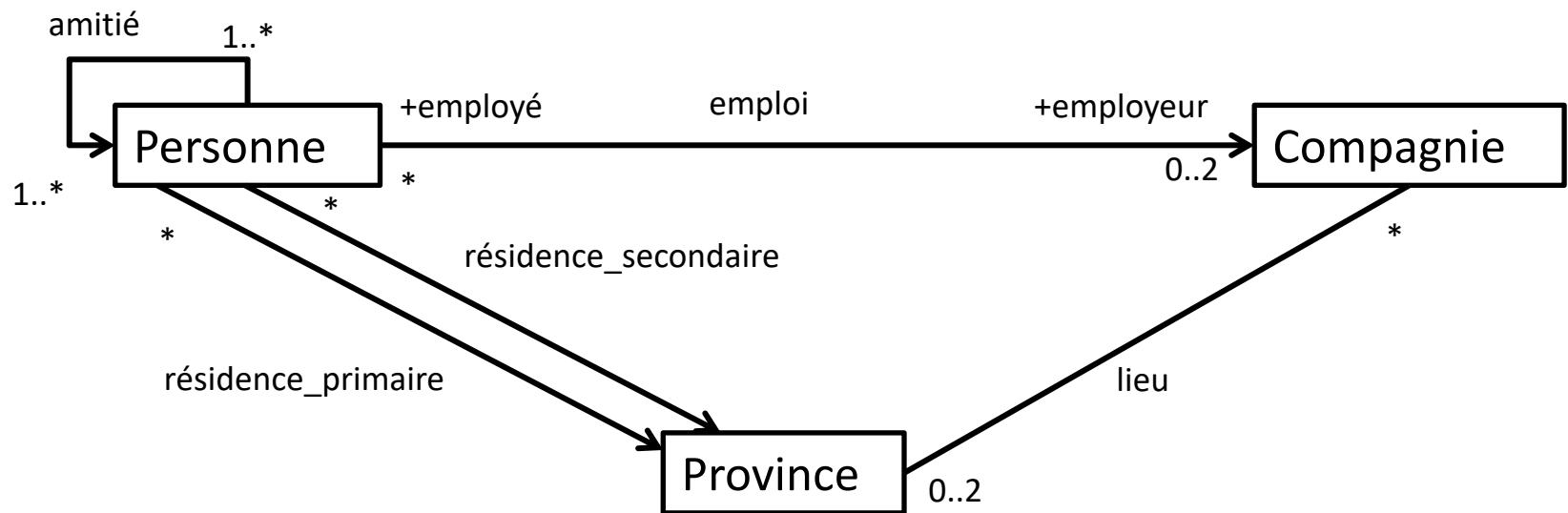
# Navigabilité dans VPP



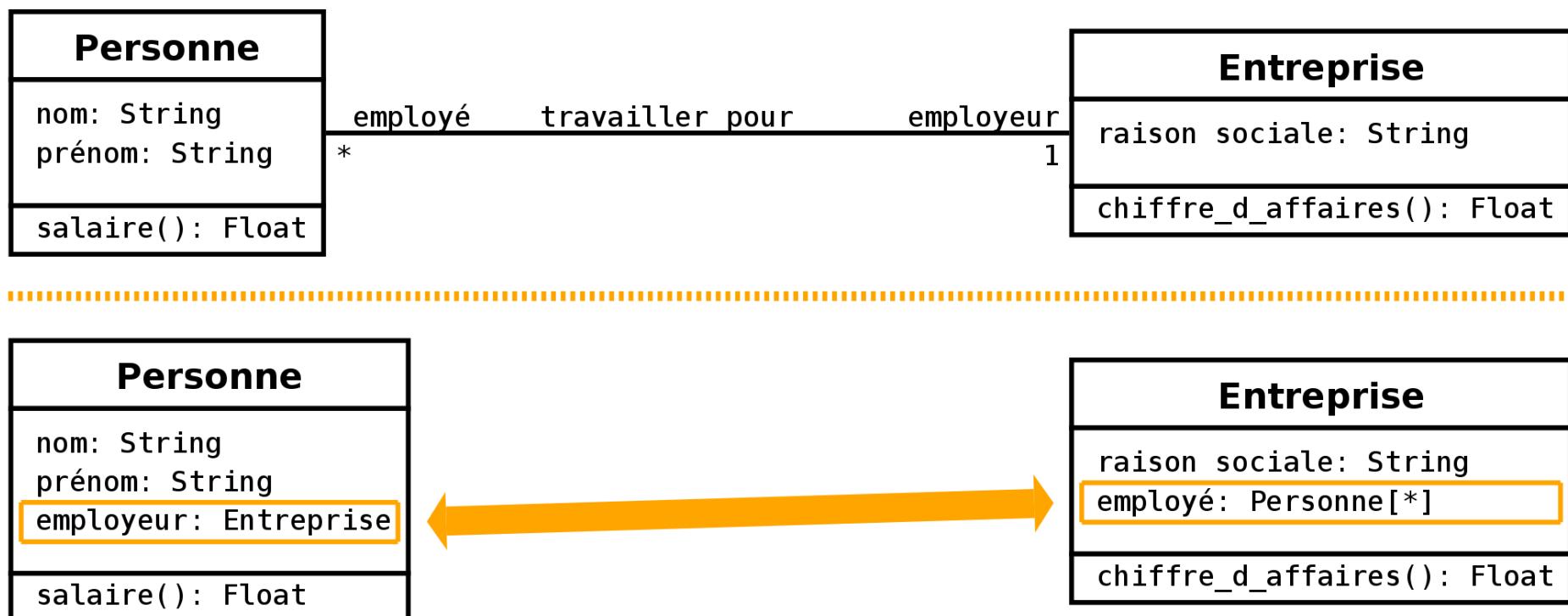
- Implicitement, ces trois notations ont la même sémantique



# Comment lire ce modèle ?

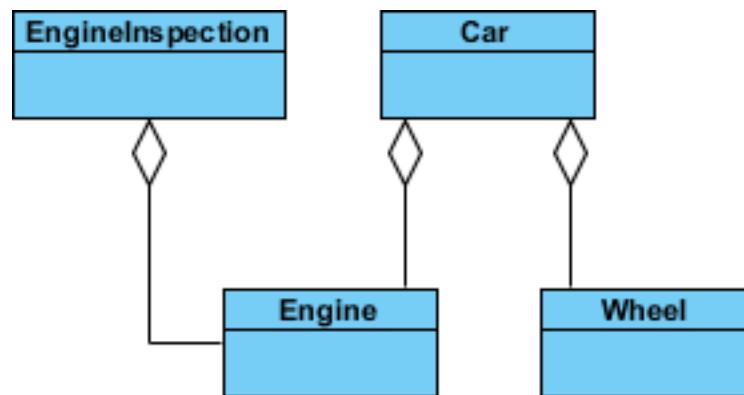


# Équivalence avec les attributs



# Agrégation

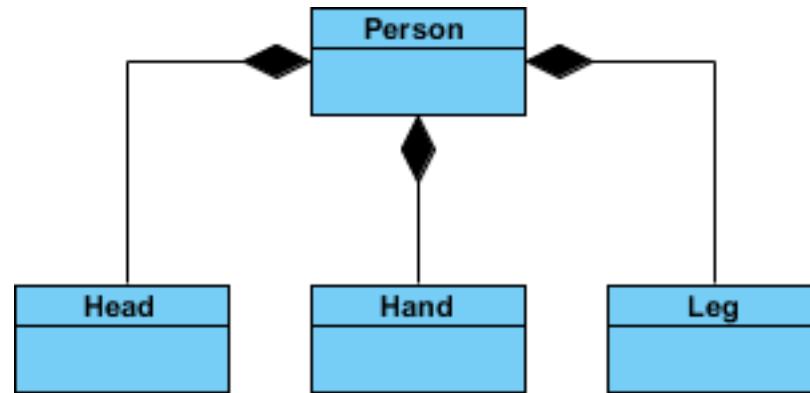
- Représente une relation « fait partie de »
- L'agrégé « fait partie de » l'agrégat
- L'Agrégé peut faire partie de plusieurs agrégats



- Représenté par un losange vide du côté agrégat

# Composition

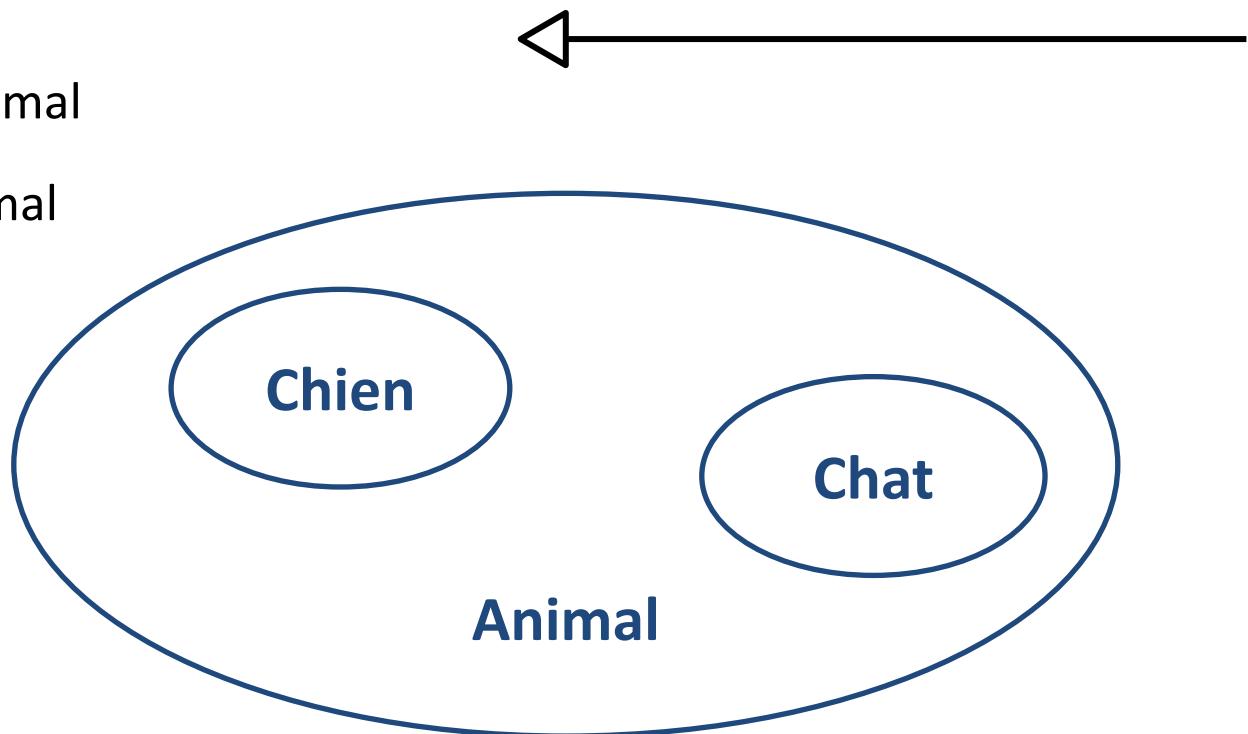
- Agrégation forte entre deux instances (composite et composant)
- Représente une relation « est dans »
- **Chaque composant est une partie d'un seul composite**
- Si le composite est détruit (ou copié), ses composants le sont aussi



- Représenté par un losange noir du côté composite

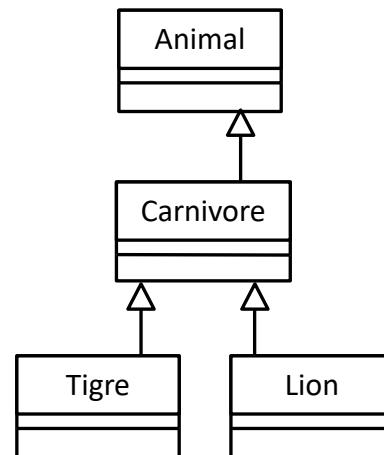
# Généralisation

- Relation d'héritage entre un élément de description générale et un élément plus spécifique
- Représente la relation « est un(e) »
- Ex:
  - Un chien *est un* animal
  - Un chat *est un* animal



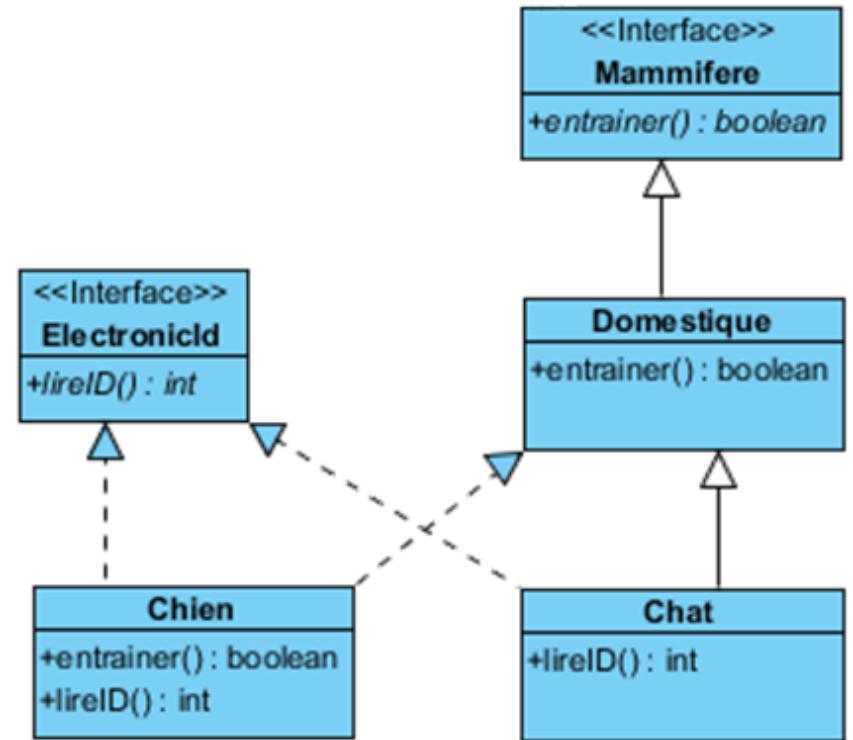
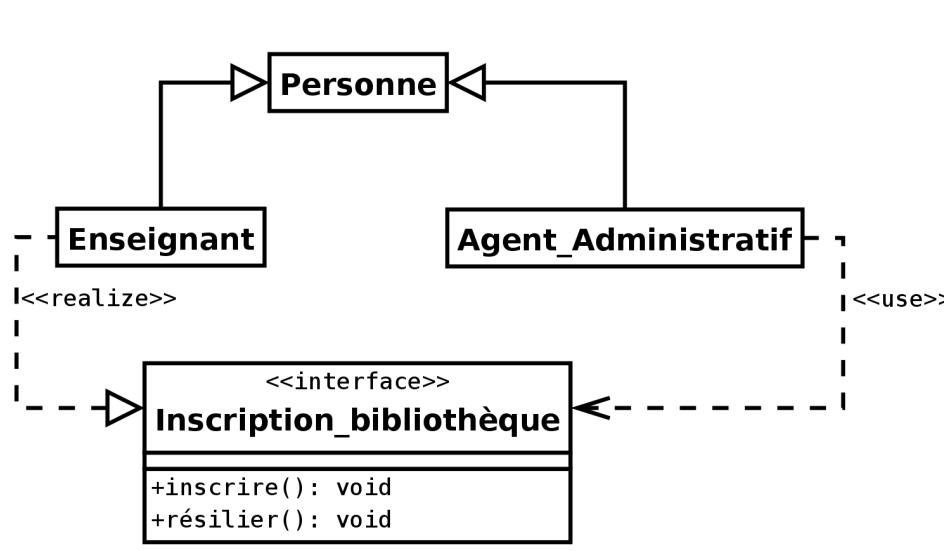
# Généralisation

- Relation
  - Non réflexive
  - Non symétrique
  - Transitive



# Interface/Implémentation

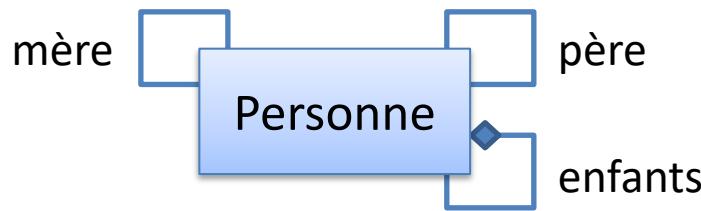
- Les interfaces ne peuvent avoir aucune implémentation
- Une classe qui implémente une interface doit fournir une implémentation de toutes les méthodes de cette interface
- Les interfaces peuvent être multiplement héritées,
  - Pas les classes abstraites



# Limitations – OCL

- Contraintes du diagramme de classes
  - Structurelles : associations, héritage, références...
  - De type : typage des attributs et signatures des méthodes
  - De visibilité, d'implémentation (abstrait ou non) etc...

- D'autres contraintes ne peuvent pas être exprimer sur le diagramme de classes



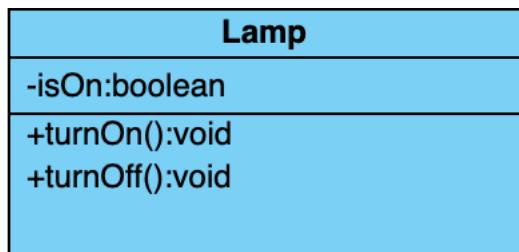
- Une personne peut-elle être sa propre mère ?
  - Son propre enfant ?
  - Son propre père ?

## Object Constraint Language (OCL):

Context Personne inv:

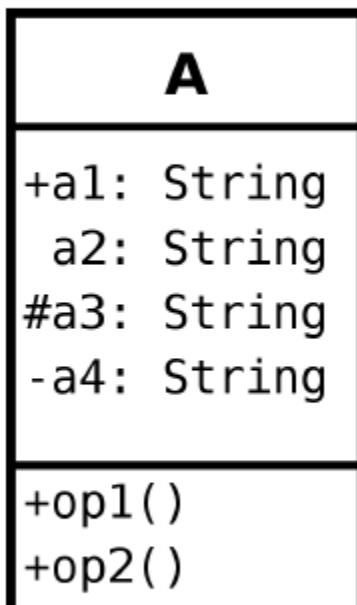
```
self->closure(p : Personne | p.mère ) ->excludes (self)
```

# Correspondance en code



```
class Lamp {  
  
    // instance variable  
    private boolean isOn;  
  
    // method  
    public void turnOn() {  
        isOn = true;  
    }  
  
    // method  
    public void turnOff() {  
        isOn = false;  
    }  
}
```

# Correspondance en Java

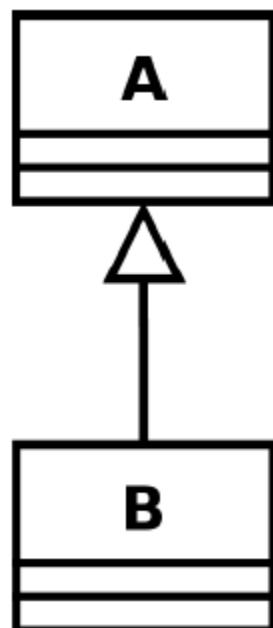


```
public class A {  
    public     String a1;  
    package   String a2;  
    protected String a3;  
    private    String a4;  
    public void op1() {  
        ...  
    }  
    public void op2() {  
        ...  
    }  
}
```

# Correspondance en Java



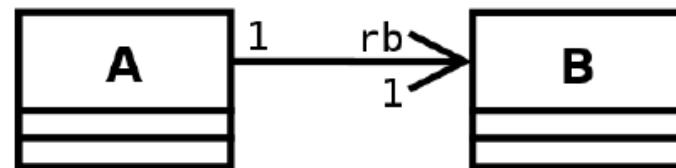
```
public abstract class A {  
    ...  
}
```



```
public class A {  
    ...  
}
```

```
public class B extends A {  
    ...  
}
```

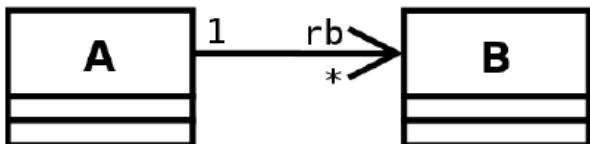
# Correspondance en Java



```
public class A {  
    private B rb;  
    public void addB( B b ) {  
        if( b != null ) {  
            this.rb=b;  
        }  
    }  
}
```

```
public class B {  
    ... // La classe B ne connaît pas l'existence de la classe A  
}
```

# Correspondance en Java



```
public class A {  
    private ArrayList <B> rb;  
    public A() { rb = new ArrayList<B>(); }  
    public void addB(B b){  
        if( !rb.contains( b ) ) {  
            rb.add(b);  
        }  
    }  
  
    public class B {  
        ... // B ne connaît pas l'existence de A  
    }  
}
```

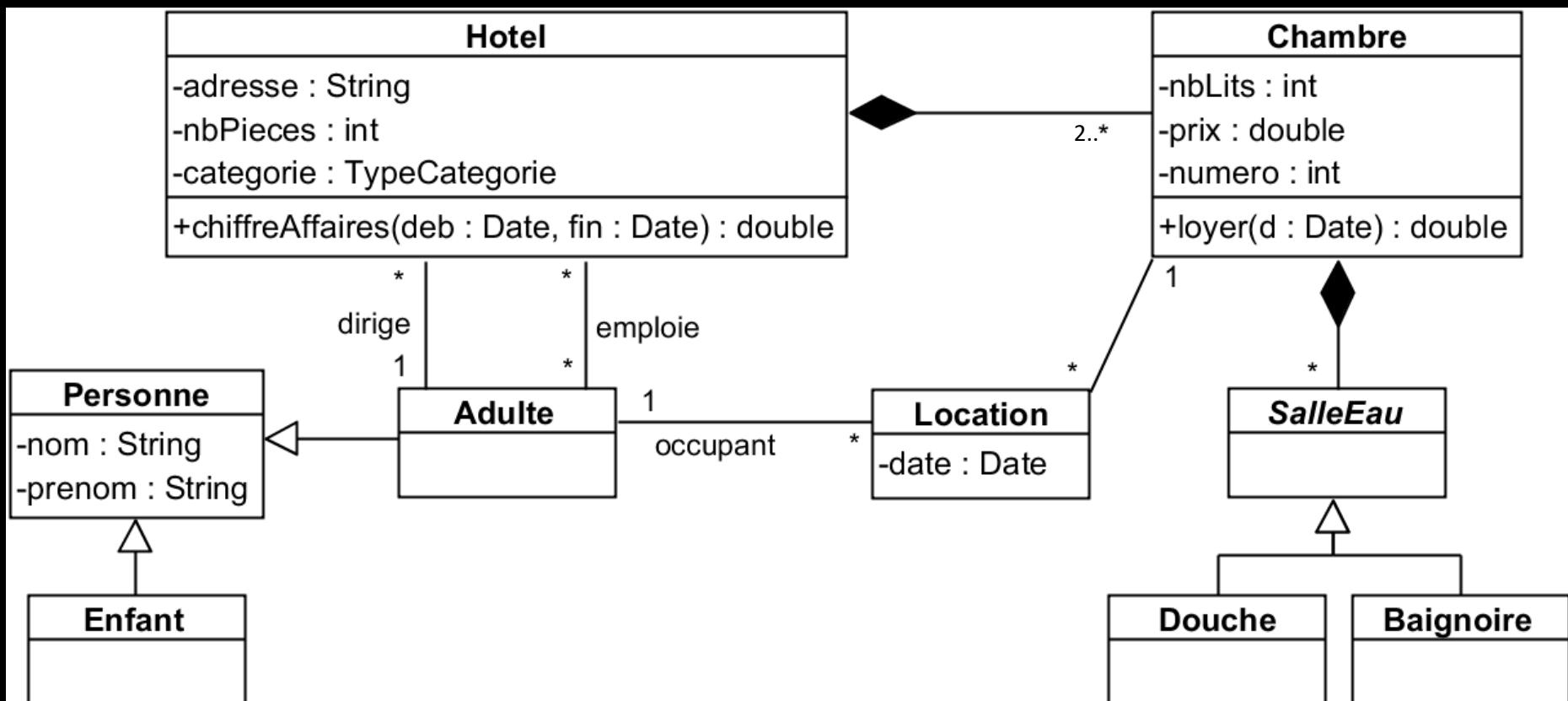
# QUESTION

*Modéliser le problème suivant sous forme d'un diagramme de classe UML:*

1. Un hôtel a les caractéristiques suivantes : une adresse, un nombre de pièces et une catégorie. Il est composé d'au moins deux chambres. Chaque chambre dispose d'une salle d'eau : douche ou bien baignoire. Une chambre est caractérisée par le nombre et de lits qu'elle contient, son prix et son numéro.
2. Un hôtel héberge des personnes. Il peut employer du personnel et il est impérativement dirigé par un directeur. On ne connaît que le nom et le prénom des employés, des directeurs et des occupants. Certaines personnes sont des enfants et d'autres des adultes (faire travailler des enfants est interdit).
3. On veut pouvoir savoir qui occupe quelle chambre à quelle date. Pour chaque jour de l'année, on veut pouvoir calculer le loyer de chaque chambre en fonction de son prix et de son occupation (le loyer est nul si la chambre est inoccupée). La somme de ces loyers permet de calculer le chiffre d'affaires de l'hôtel entre deux dates.

# QUESTION

*Modéliser le problème suivant sous forme d'un diagramme de classe UML:*



# Diagramme de séquence UML

# Rappel d'un CU

1. Le client introduit sa carte bancaire
2. Le logiciel lit la carte et vérifie que la carte est valide
3. Le logiciel demande au client de saisir son code
4. Le client saisit son code confidentiel
5. Le logiciel vérifie que le code correspond
6. Le client choisit une opération de retrait
7. Le logiciel demande le montant à retirer

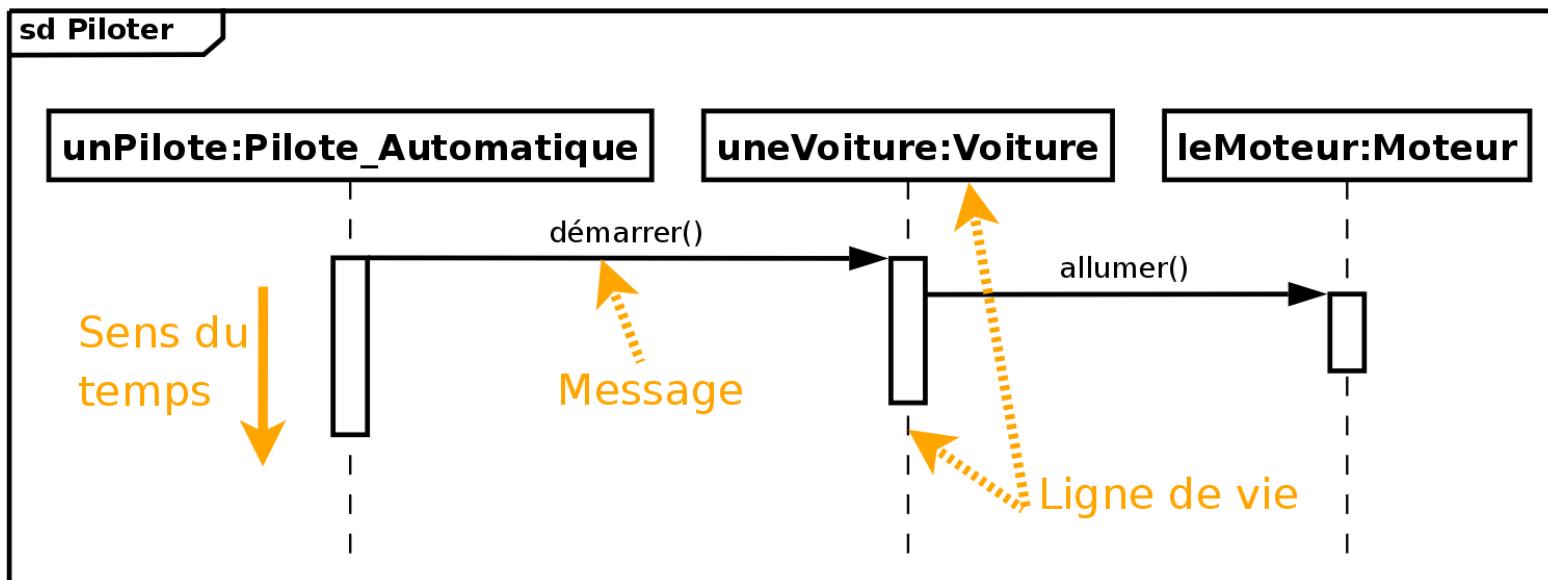
...

# Du CU au diagramme de séquence

- Un CU est réalisé par une collaboration
  - Ensemble d'objets qui s'échangent des messages et travaillent ensemble pour accomplir une tâche
- Un diagramme de séquence
  - Décrit les CU en mettant en évidence les **interactions** entre les instances des classes (objets) du logiciel
  - Montre la séquence **au cours du temps** des échanges de messages entre les objets participant à un scénario

# Diagramme de séquence

- Capture le comportement dynamique du système
- Composé d'entités dynamiques
  - Les objets (pas de classes à l'exécution)
  - Leur ligne de vie
  - Les messages échangés



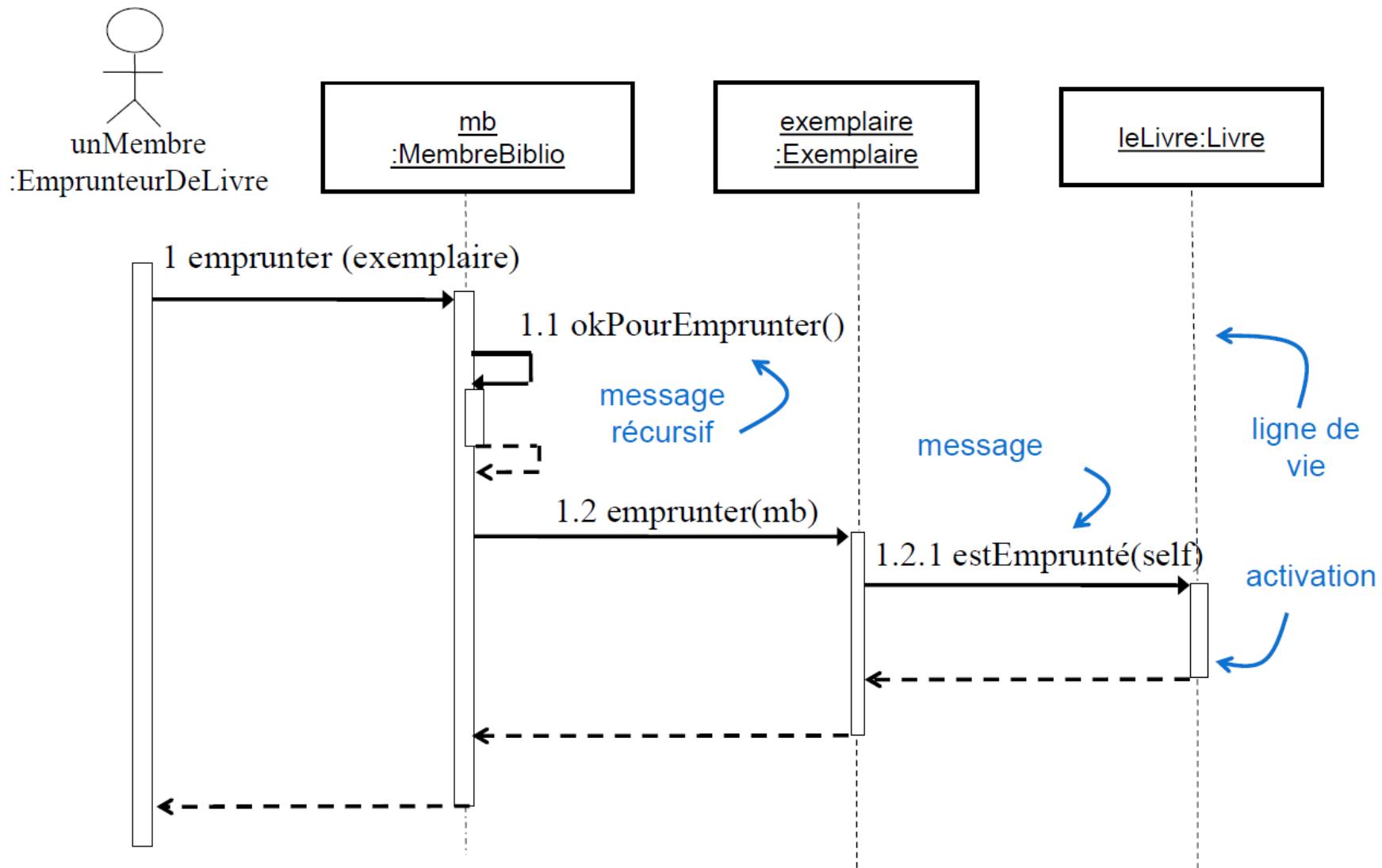
# Dimensions

- Dimension verticale : **le temps**
  - L'ordre d'envoi d'un message est déterminé par sa position sur l'axe vertical du diagramme
  - Le temps s'écoule de haut en bas
- Dimension horizontale : **les objets** (et les acteurs)
  - L'ordre de disposition des objets sur l'axe horizontal est sans importance (visez l'intuitivité)

# Éléments graphiques

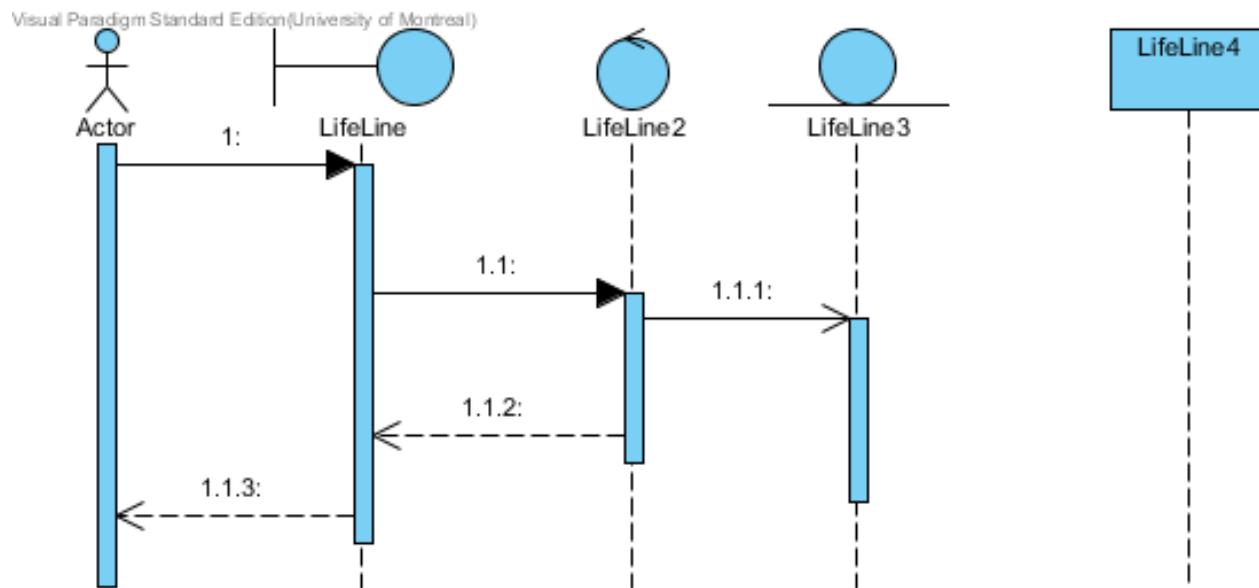
- Les **objets** qui interagissent dans le scénario
- Représentation graphique de la **ligne de vie** de chaque objet et de ses activations
- Les différents **types de messages** envoyés
  - simple, synchrone, asynchrone
- Les **indications de contrôle**
  - Branchement conditionnel et itération
  - Création et destruction d'objets
  - Délais de transmission
  - Contraintes temporelles

# Éléments graphiques



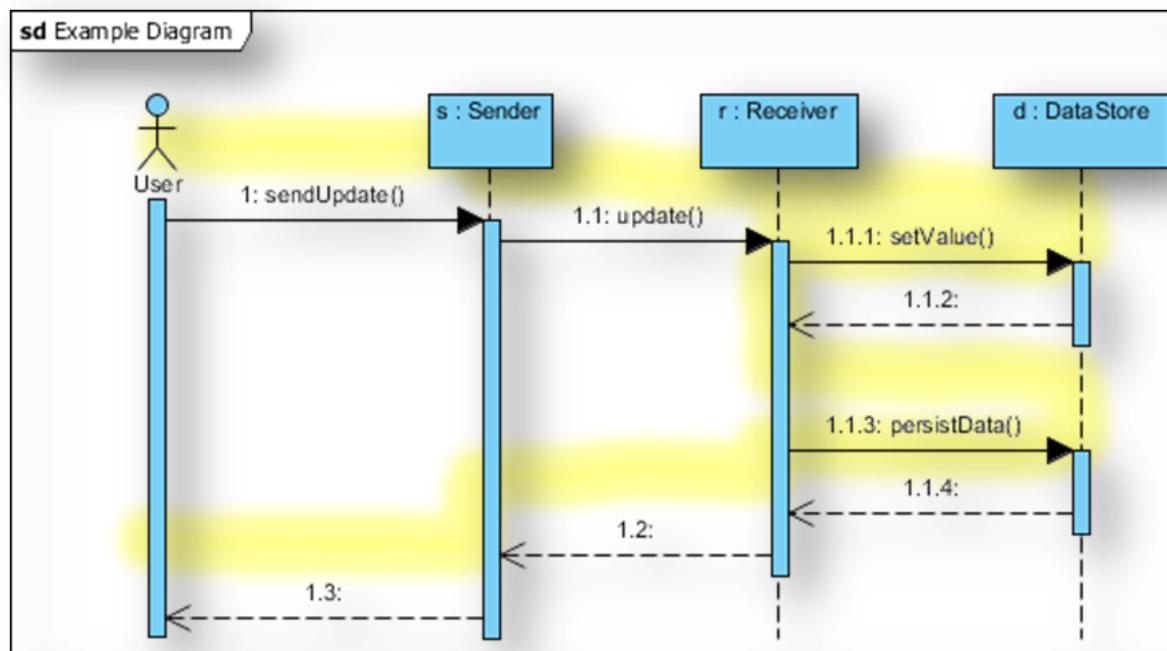
# Acteurs et objets

- Chaque ligne de vie représente un acteur ou objet différent
- La ligne de vie représente la présence de l'instance
- L'activation représente une période d'activité ou d'exécution
- Peut avoir la forme des stéréotypes



# Messages

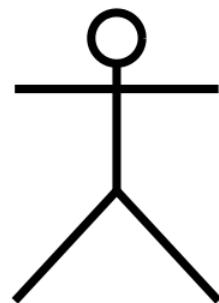
- Messages numérotés de façon à refléter l'imbrication des envois
  - Message 1.1.3 envoyé après que la réponse au message 1.1.2 ait été reçue
  - Attendre la réponse 1.1.4 avant de pouvoir répondre au message 1.1



# Retour

**sd Rechercher livre**

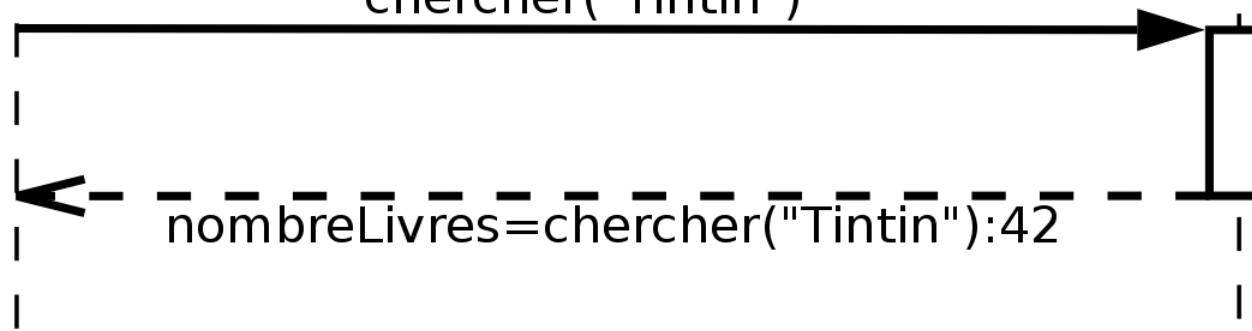
+nombreLivres:Integer



Client

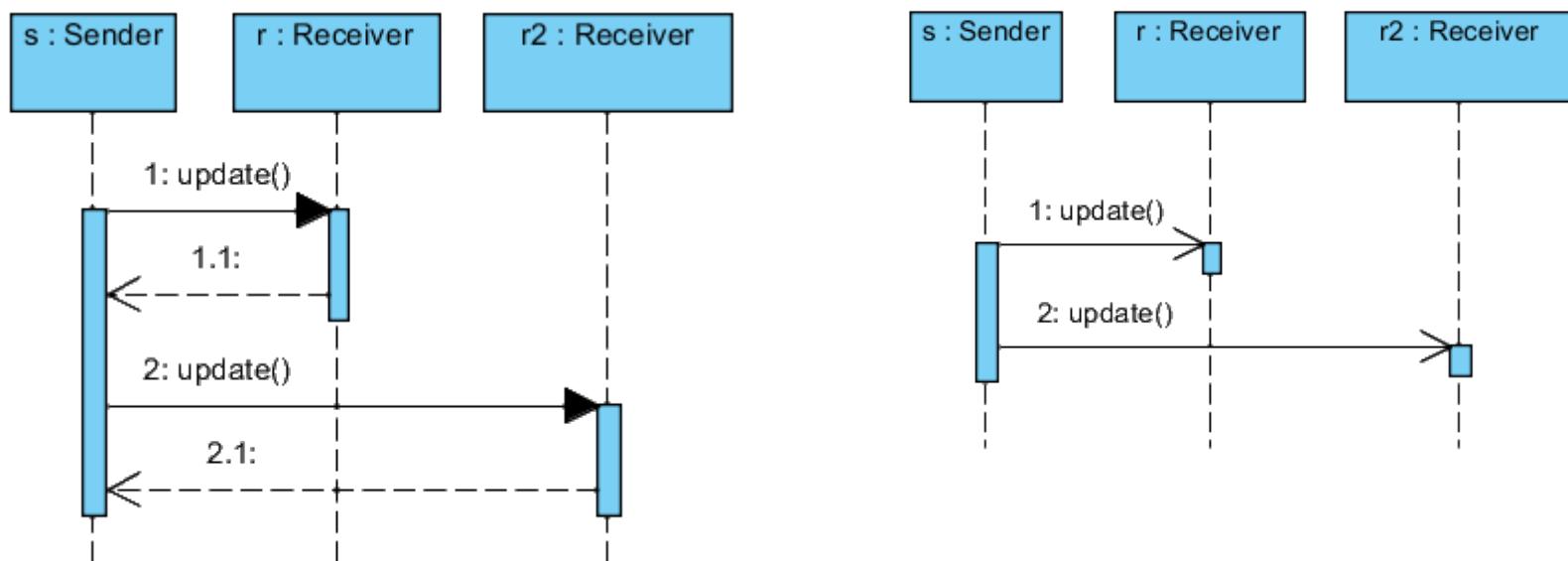
**:Médiathèque**

chercher("Tintin")



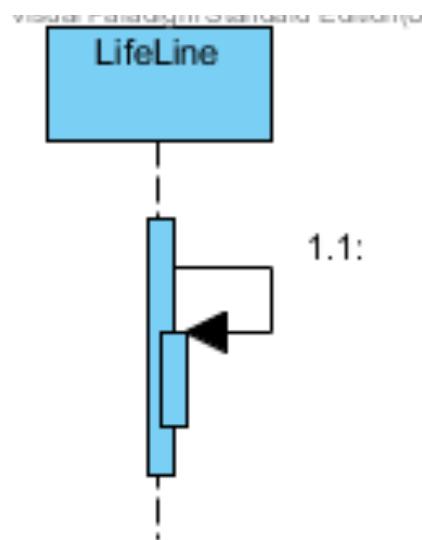
# Flux de messages

- Les objets actifs envoient deux types de messages
  - **Synchrone** : attend la réponse du destinataire avant de poursuivre
  - **Asynchrone** : poursuit son activité sans attendre la réponse à son message; elle lui sera signalée ultérieurement



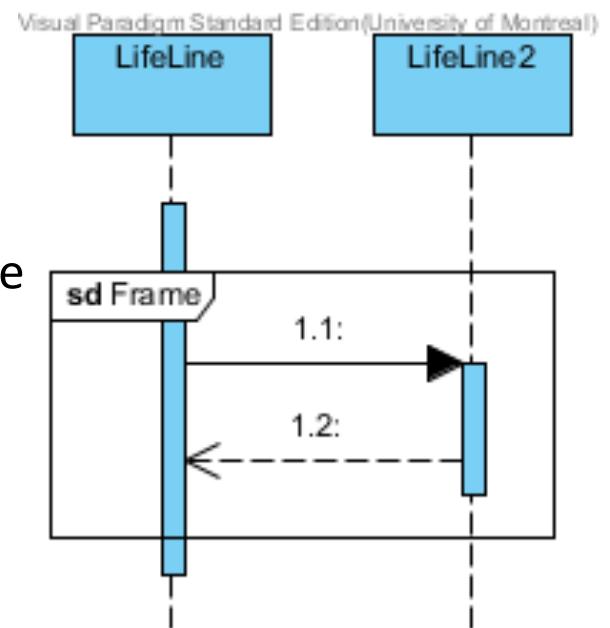
# Messages à soi-même

- Message échangé avec le même objet
- Typiquement pour les appels de méthodes privées, mais pas nécessairement



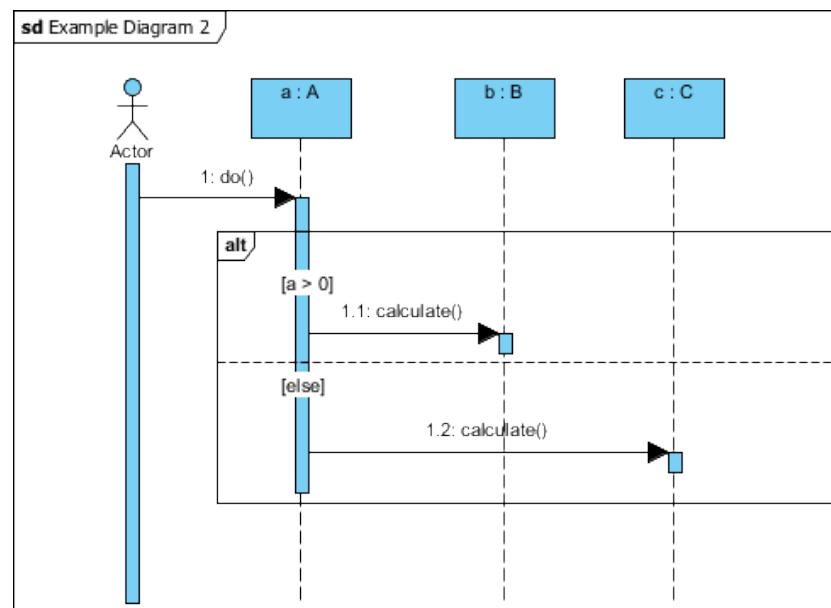
# Fragments

- Sections démarquée d'un diagramme de séquence
- Indique:
  - un **comportement particulier**, comme un cycle
  - une **référence** à un autre diagramme de séquence
  - un segment **optionnel ou conditionnel**
- Défini par:
  - un rectangle qui démarque les éléments affectés par le fragment
  - une étiquette avec l'indication du type de fragment et le cas échéant un nom

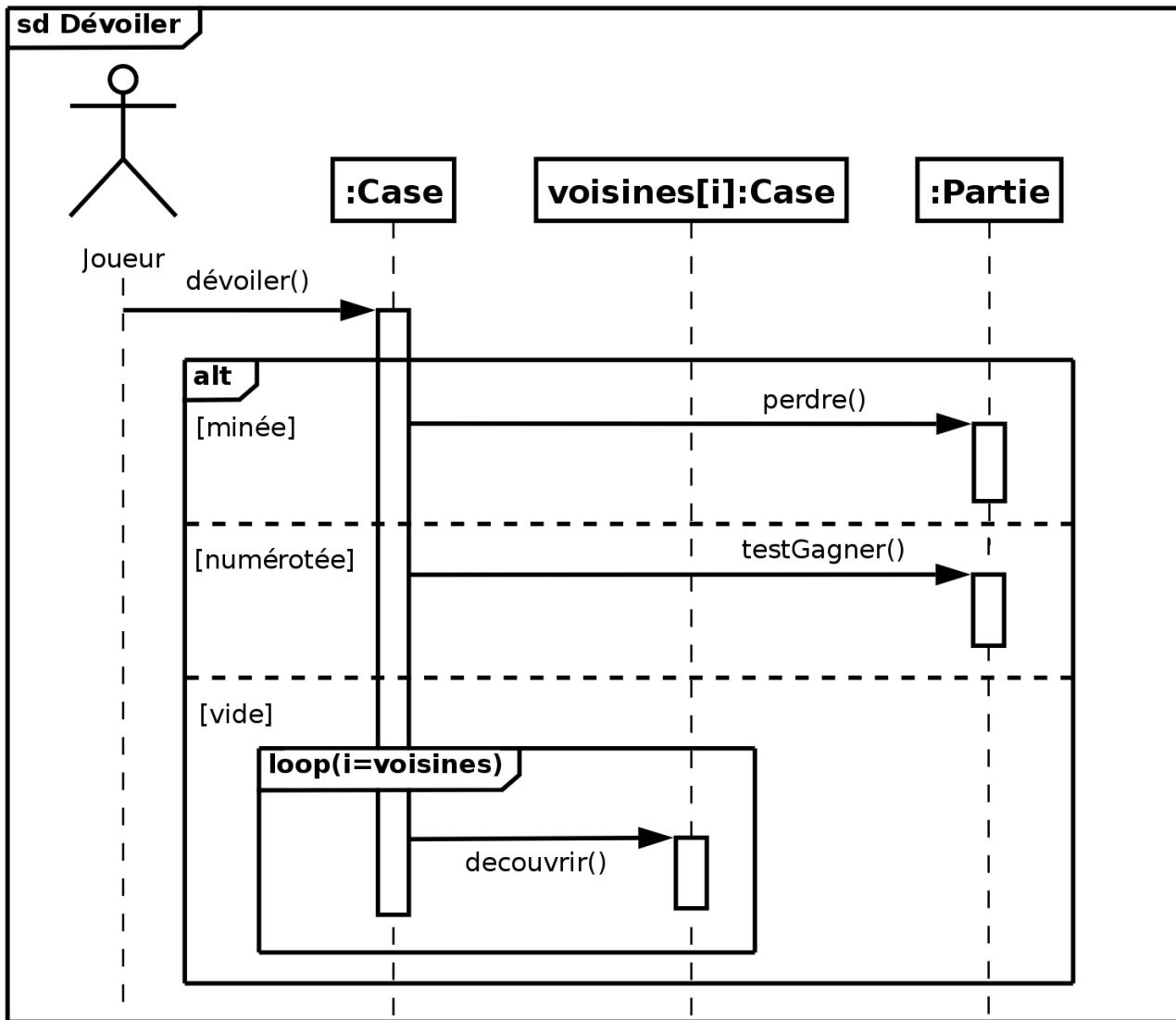


# Fragments ALT

- **alt:** fragment avec des **alternatives** pour la logique conditionnelle
  - conditions sont exprimées dans les gardes
  - comportement if-then-else
  - seulement un des segments est exécuté



# Alternatives et itérations



# Invocation de méthodes

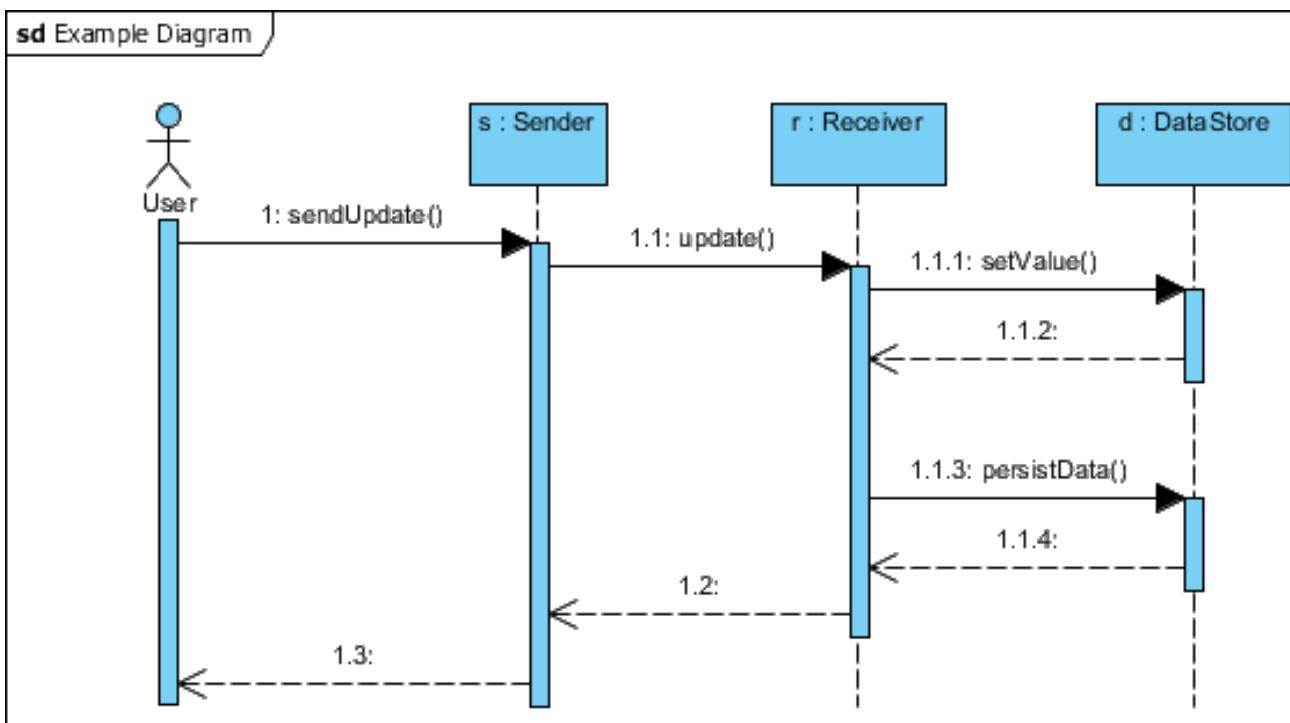
- Un message est une opération ou méthode présente dans la classe de l'objet qui reçoit le message
  - Doit être appelé avec les bons paramètres
  - Le retour de cette méthode est la réponse à ce message
- La cohérence avec diagramme de classes doit être maintenue
- Si  $a:A$  envoie  $m(1)$  à  $b:B$ :
  - $B$  doit avoir la méthode  $m(:int)$  définie
  - Il existe une dépendance de  $A$  vers  $B$ , soit par association soit par type d'attribut

# Cohérence avec le diagramme de classe

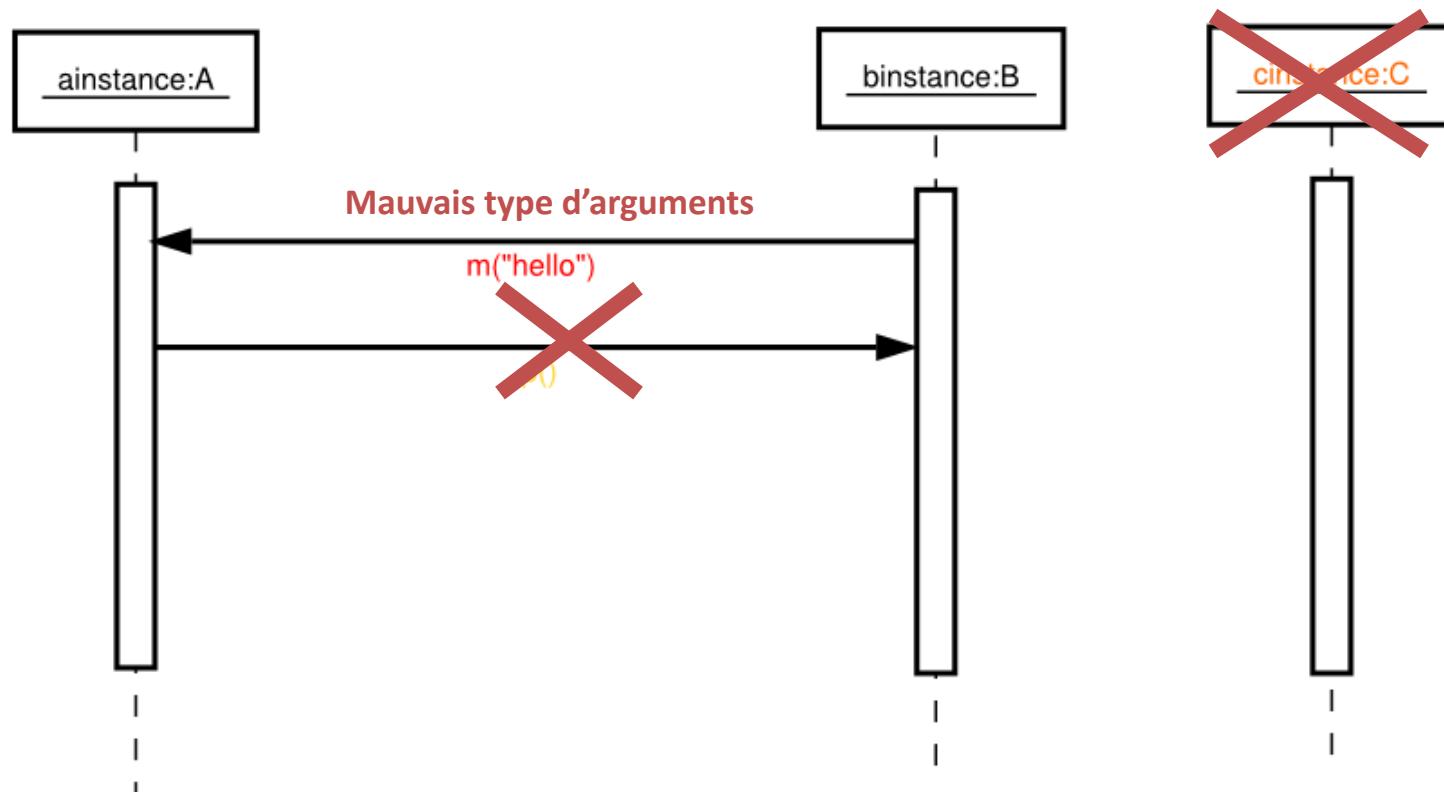
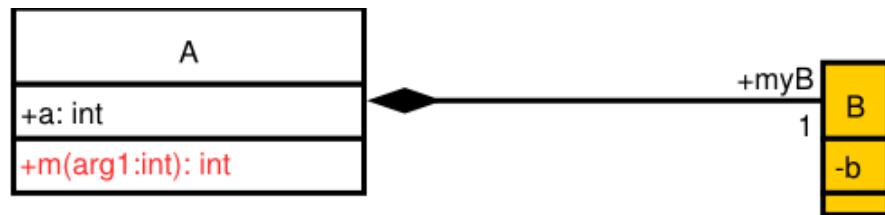
Sender
+sendUpdate(s : String) : void

Receiver
+update(s : String) : boolean

DataStore
+persistData() : boolean
+setValue(s : String) : void

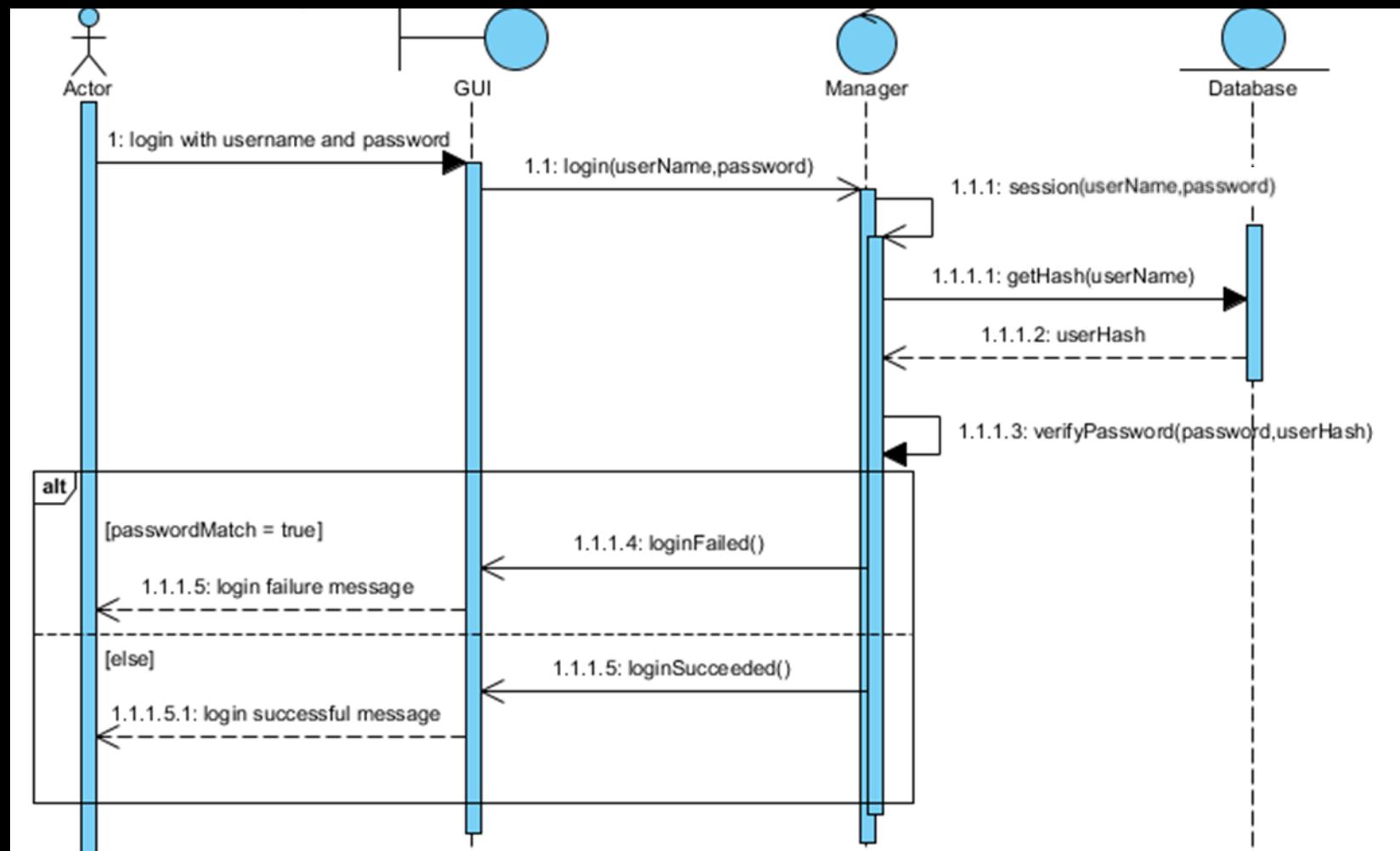


# Cohérence avec le diagramme de classe



# QUESTION

*Quel serait un diagramme de classe pour ce diagramme de séquence ?*



# QUESTION

*Quel serait un diagramme de classe pour ce diagramme de séquence ?*

