

# Méthodes de Développement Industriel (MDI)

**Mathieu Acher**

<http://www.mathieuacher.com>

*Associate Professor*

*University of Rennes 1*

**<http://mathieuacher.com/teaching/MDI/>**

# Objectifs de MDI

- Méthodes de développement industriel (MDI)
  - En fait: génie logiciel / software engineering
  - Comment développer des systèmes logiciels de plus en plus complexe?
- #1 Prendre conscience de la complexité des systèmes logiciels actuels et à venir
  - Les enjeux et l'impact sur le métier
- #2 Modélisation
  - UML, SysML
- #3 Design patterns, refactoring, test
  - OO avancé
- #4 Méthodes

# Outline

---

## ① UML History and Overview

## ② UML Language

- ① UML Functional View
- ② UML Structural View
- ③ UML Behavioral View
- ④ UML Implementation View
- ⑤ UML Extension Mechanisms

## ③ UML Internals

## ④ UML Tools

## ⑤ Conclusion

# Outline

---

## ① UML History and Overview

## ② UML Language

- ① UML Functional View
- ② UML Structural View
- ③ UML Behavioral View
- ④ UML Implementation View
- ⑤ UML Extension Mechanisms

## ③ UML Internals

## ④ UML Tools

## ⑤ Conclusion

# Méthodes de modélisation

---

- L'apparition du paradigme objet à permis la naissance de plusieurs méthodes de modélisation
  - OMT, OOSE, OOD, Fusion, ...
- Chacune de ces méthodes fournies une notation graphique et des règles pour élaborer les modèles
- Certaines méthodes sont outillées

# Méthodes de modélisation

---

- Entre 89 et 94 : le nombre de méthodes orientées objet est passé de 10 à plus de 50
- Toutes les méthodes avaient pourtant d'énormes points communs (objets, méthode, paramètres, ...)
- Au milieu des années 90, G. Booch, I. Jacobson et J. Rumbaugh ont chacun commencé à adopter les idées des autres. Les 3 auteurs ont souhaité créer un langage de modélisation unifié

# Méthodes de modélisation

---

- OMT de James Rumbaugh (*General Electric*)

fournit une représentation graphique des aspects statique, dynamique et fonctionnel d'un système ;

- OOD de Grady Booch

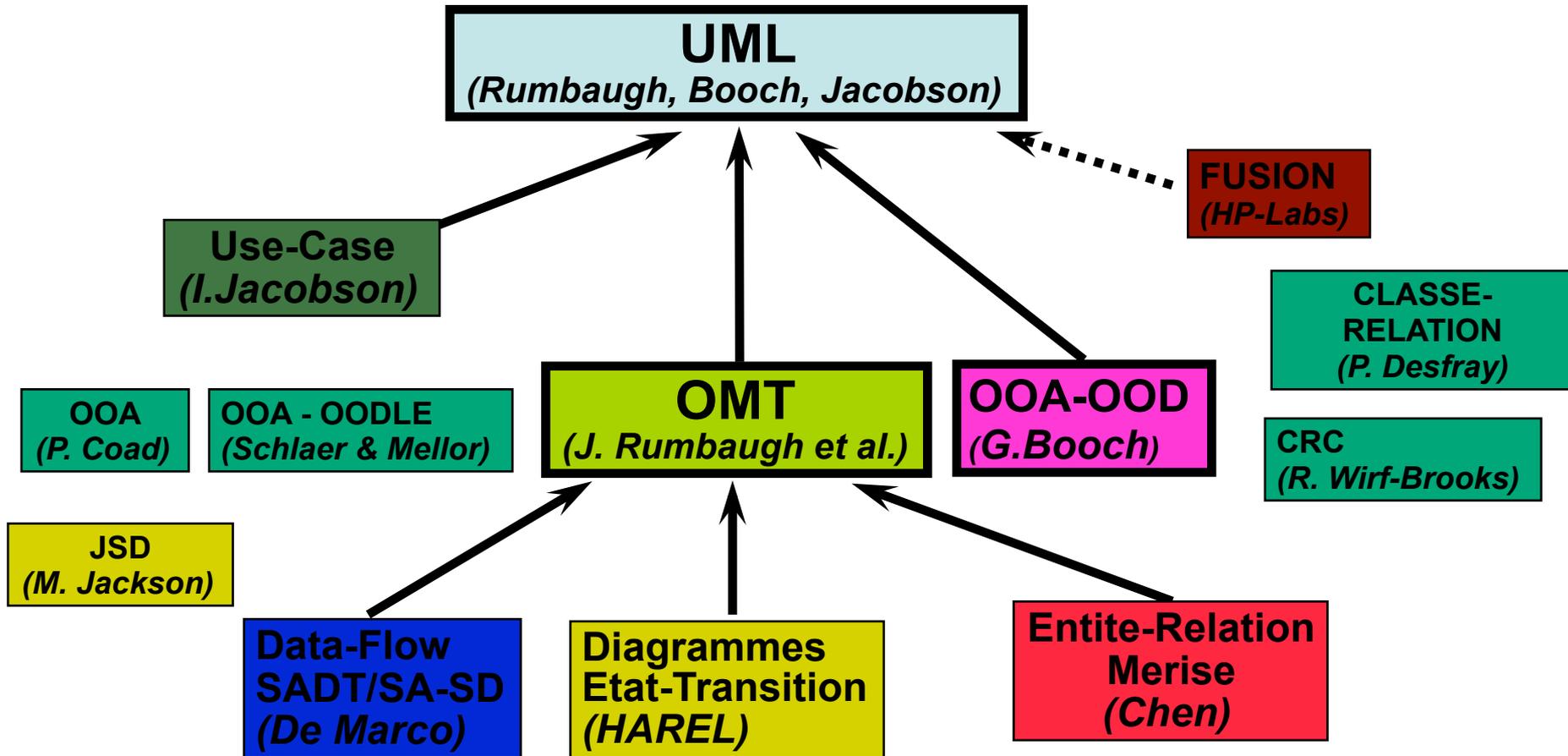
définie pour le *Department of Defense*, introduit le concept de paquetage (*package*) ;

- OOSE d'Ivar Jacobson (*Ericsson*)

fonde l'analyse sur la description des besoins des utilisateurs (cas d'utilisation, ou *use cases*).

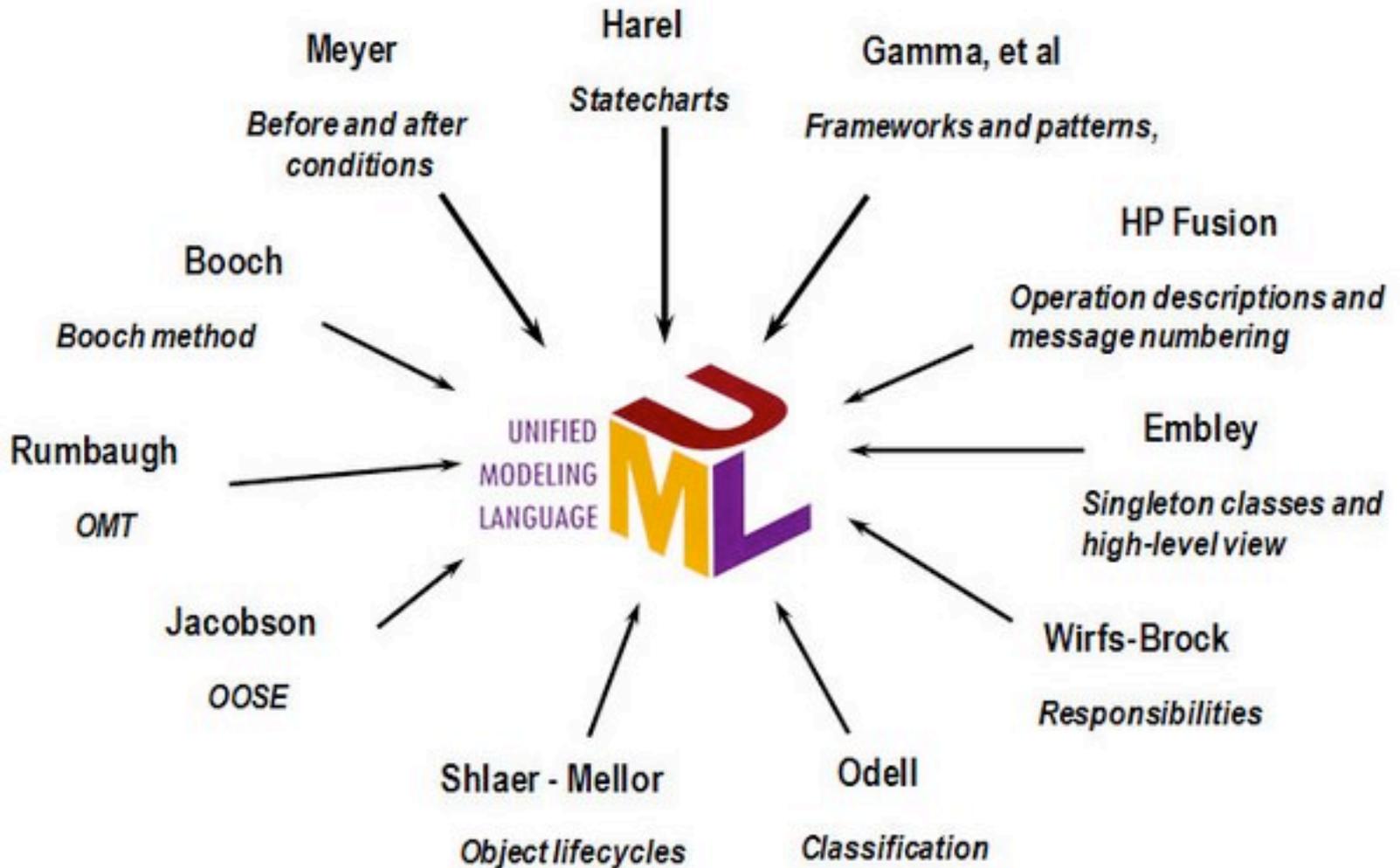
# Genealogy of UML

---

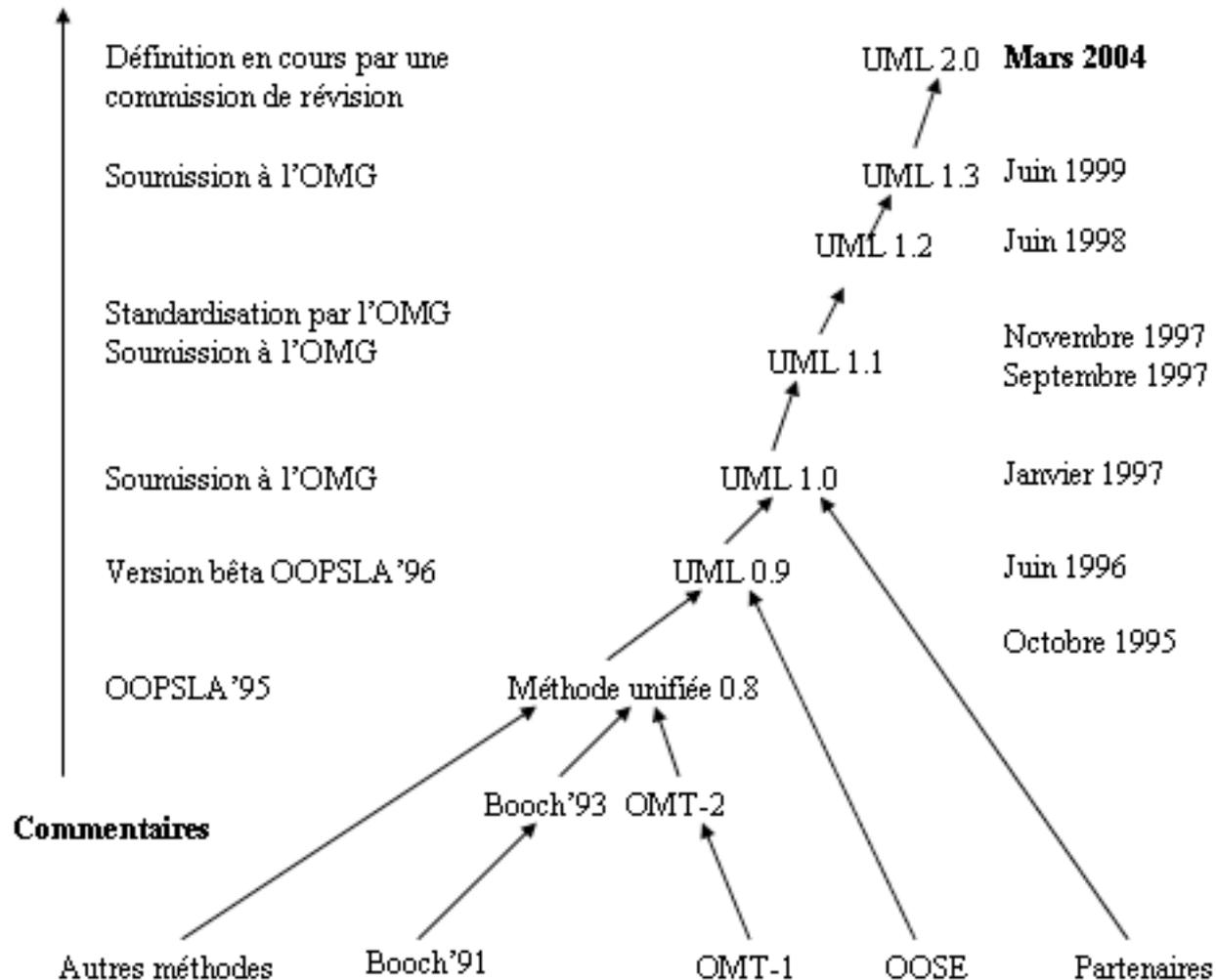


# Genealogy of UML

---



# UML History



# Aujourd'hui

---

- UML est le langage de modélisation orienté objet le plus connu et le plus utilisé au monde
- UML s'applique à plusieurs domaines
  - OO, RT, Deployment, Requirement, ...
- UML n'est pas une méthode
  - RUP
- Peu d'utilisateurs connaissent le standard, ils ont une vision outillée d'UML (vision utilisateur)
  - 5% forte compréhension, 45% faible compréhension, 50% aucune compréhension
- UML est fortement critiqué car pas assez formel
- Le marché UML est important et s'accroît
  - MDA et MDD, UML2.x, IBM a racheté Rational !!!

# UML Today

---

- Official website: <http://www.uml.org>
- Official specifications: <http://www.omg.org/spec/UML>

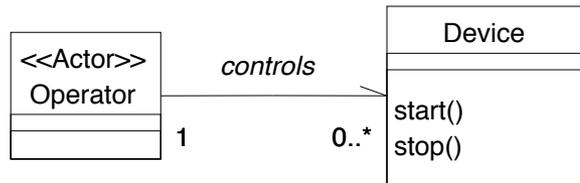


**OBJECT MANAGEMENT GROUP**

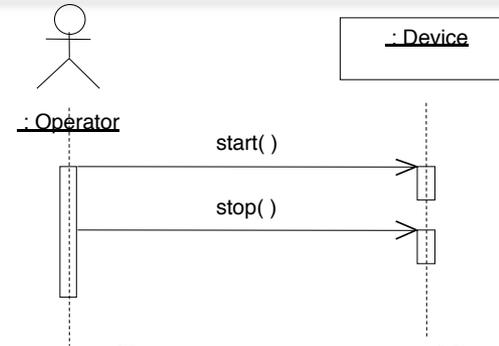
**Current version: v2.4.1, August 2011**

# UML: one model,

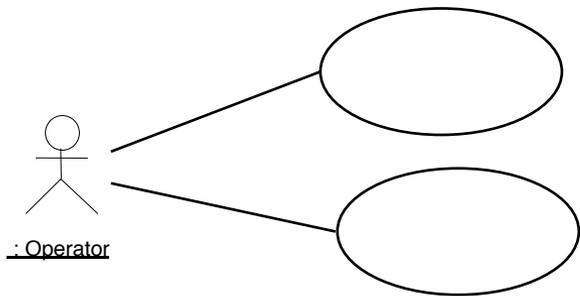
## *4 main dimensions, multiple views*



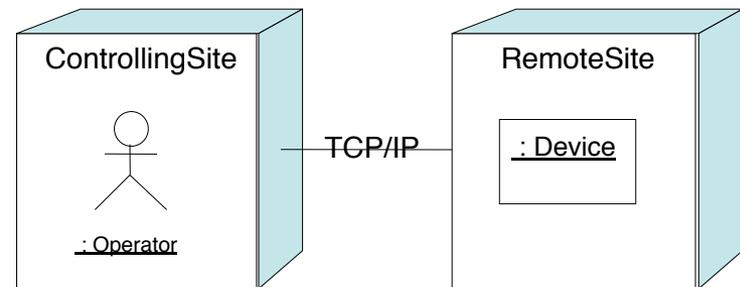
e.g., Class diagram



e.g., Sequence diagram



e.g., UseCase diagram



e.g., Implementation diagram

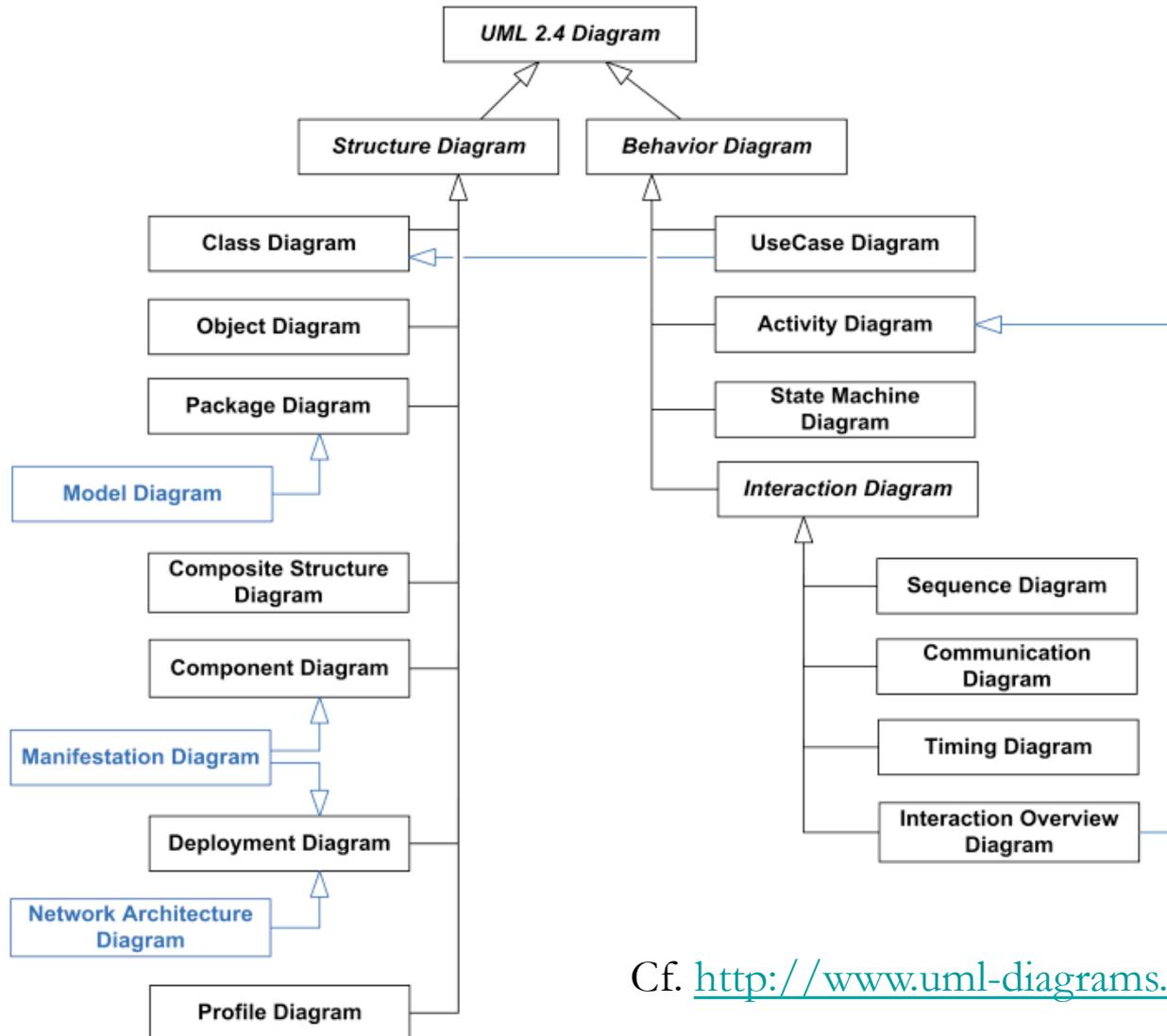
# The X diagrams of UML

---

## ■ Modeling along 4 main viewpoints:

- Static Aspect (*Who?*)
  - Describes objects and their relationships (Class, Object and Composite Structure Diagrams)
  - Structuring with packages (Package Diagram)
- User view (*What?*)
  - Use cases Diagram
- Dynamic Aspects (*When?*)
  - Interaction Diagrams
    - Sequence Diagram (scenario)
    - Communication Diagram (between objects)
    - Timing Diagram
  - State Machine Diagram (from Harel)
  - Activity Diagram
- Implementation Aspects (*Where?*)
  - Component and Deployment Diagrams

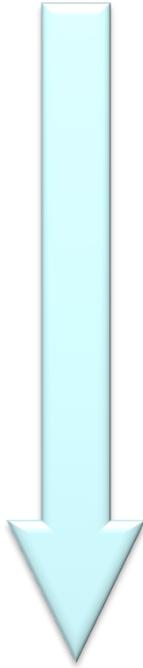
# UML 2.4 Overview



Cf. <http://www.uml-diagrams.org>

# UML : pourquoi ?

---



- Réfléchir
- Définir la structure « gros grain »
- Documenter
- Guider le développement
- Développer, Tester, Auditer

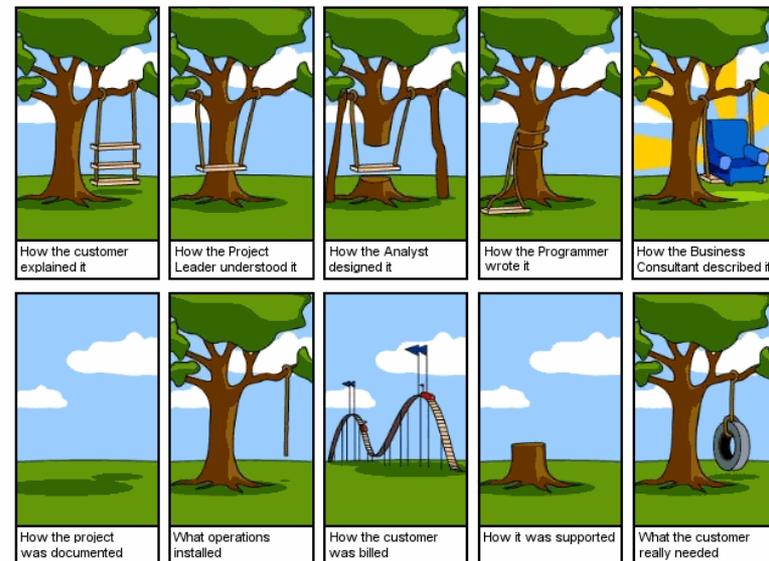
# Un peu de méthodologie...

---

- Une méthode de développement de logiciels, c'est :
  - Une notation
    - La syntaxe --- (semi-) graphique dans le cas de UML
  - Un méta-modèle
    - La sémantique --- paramétrable dans UML (*stéréotypes*)
  - Un processus
    - Détails dépendants du domaine d'activité :
      - Informatique de gestion
      - Systèmes réactifs temps-réels
      - *Shrink-wrap* software (PC)

# Processus de développement avec UML

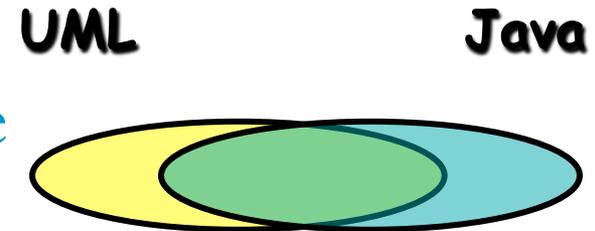
- Approche itérative, incrémentale, dirigée par les cas d'utilisation
  - Expression des besoins
    - Elaboration d'un modèle « idéal »
  - Analyse
    - passage du modèle idéal au monde réel
  - Conception
    - passage du modèle idéal au monde réel
  - Réalisation et Validation



# UML et Java ?

---

- UML Vers Java : Génération de code
- Java Vers UML : Rétro-ingénierie
- UML n'est pas une syntaxe graphique pour Java !
- Traduction (très) incomplète
- Pas de traduction standardisée
- Dépend du métier, des outils disponibles, ...
- Des outils industriels disponibles
- Des recherches en cours ...



**... vers une capitalisation du savoir-faire**

# UML et Java : vision globale

---

- **MODELES STATIQUES**

- Traduction des diagrammes de classes
- Classes, Associations, Généralisation
- Propriétés dérivées, Invariants

- **MODELES DYNAMIQUES**

- Traduction de pré/post conditions
- Traduction de diagrammes d'états / d'activités
- Traduction du langage d'actions (UML exécutable)

... traduction souvent incomplète mais très utile

# Outline

---

① UML History and Overview

② UML Language

① UML Functional View

② UML Structural View

③ UML Behavioral View

④ UML Implementation View

⑤ UML Extension Mechanisms

③ UML Internals

④ UML Tools

⑤ Conclusion

# Expression des besoins

---

- Sujet longtemps négligé (*e.g.*, OMT)
- Question de l'expression des besoins pourtant fondamentale
  - Et souvent pas si facile (*cible mouvante*)
    - *cf. syndrome de la balançoire*
- Object-Oriented Software Engineering (Ivar Jacobson *et al.*)
  - Principal apport : la technique des acteurs et des cas d'utilisation
  - Cette technique est intégrée à UML

# Quatre objectifs

---

- ① Se comprendre
- ② Représenter le système
- ③ Exprimer le service rendu
- ④ Décrire la manière dont le système est perçu

# Structural View

---

## Class and Object Diagrams

# Rappel

---

- **Un objet**

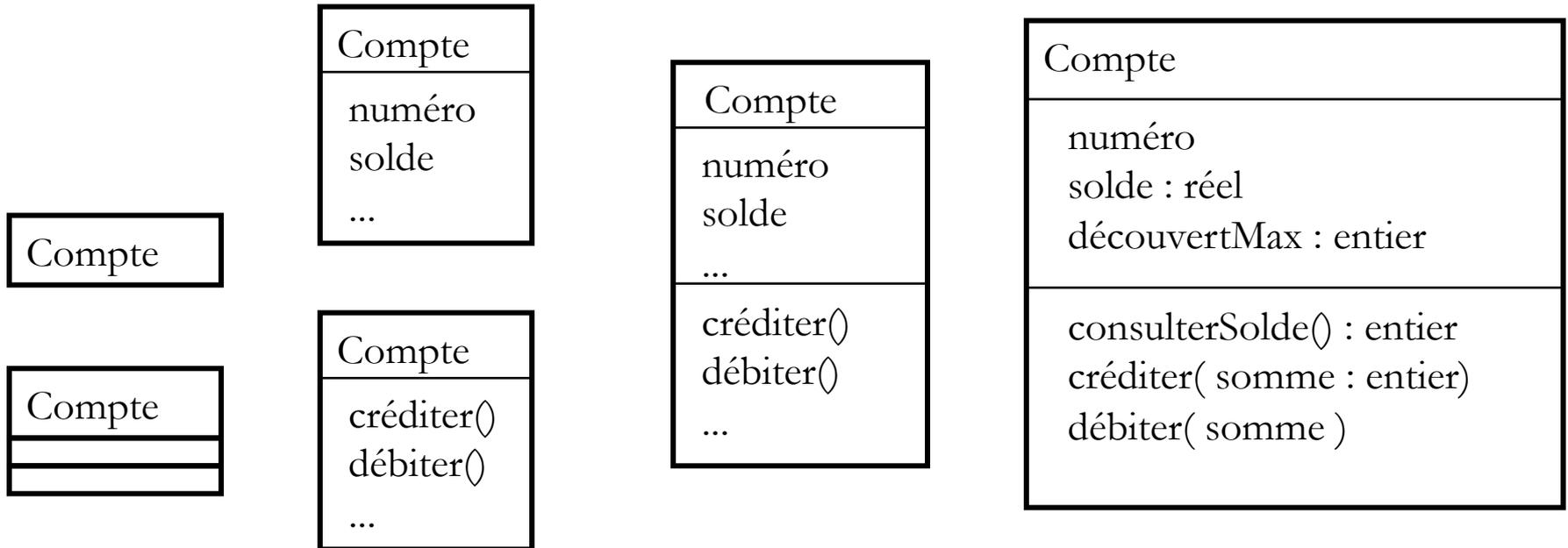
- Encapsulation d'un état et d'un ensemble d'opérations (qui s'appliquent à cet état)
  - Abstraction d'une entité réelle
  - Identité unique
  - Existence temporelle (création, modification, destruction)

- **Une classe**

- Description du comportement et des propriétés communs à un ensemble d'objets
  - Un objet est une instance d'une classe
  - Chaque classe a un nom

# Notations simplifiées pour les classes

---



## Note de style :

- les noms de classes commencent par une majuscule
- les noms d'attributs et de méthodes commencent par une minuscule

# Notation pour les classes

---

Nom de la classe

Attributs

nom

type

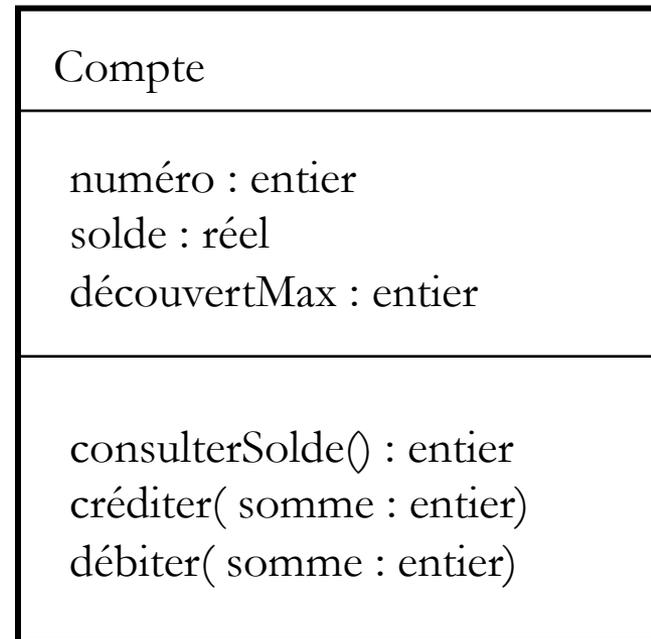
Opérations

nom

paramètre

type du résultat

Contraintes



{ inv: solde > découvertMax }

# Notations pour les objets

---

leCompteDePaul

: Compte

leCompteDePaul : Compte

leCompteDePaul : Compte

numéro = 6688

solde = 5000

découvertMax = -100

Convention :

- les noms d'objets commencent par une minuscule et sont soulignés

# Classe *vs.* Objets

Une **classe** spécifie la structure et le comportement d'un ensemble d'objets de même nature

- La structure d'une classe est constante

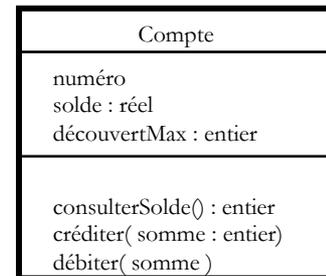


Diagramme de classes

M1

M0

- Des objets peuvent être ajoutés ou détruits pendant l'exécution
- La valeur des attributs des objets peut changer

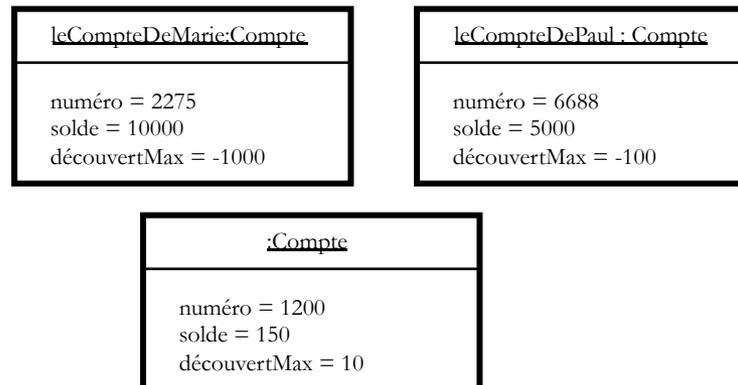
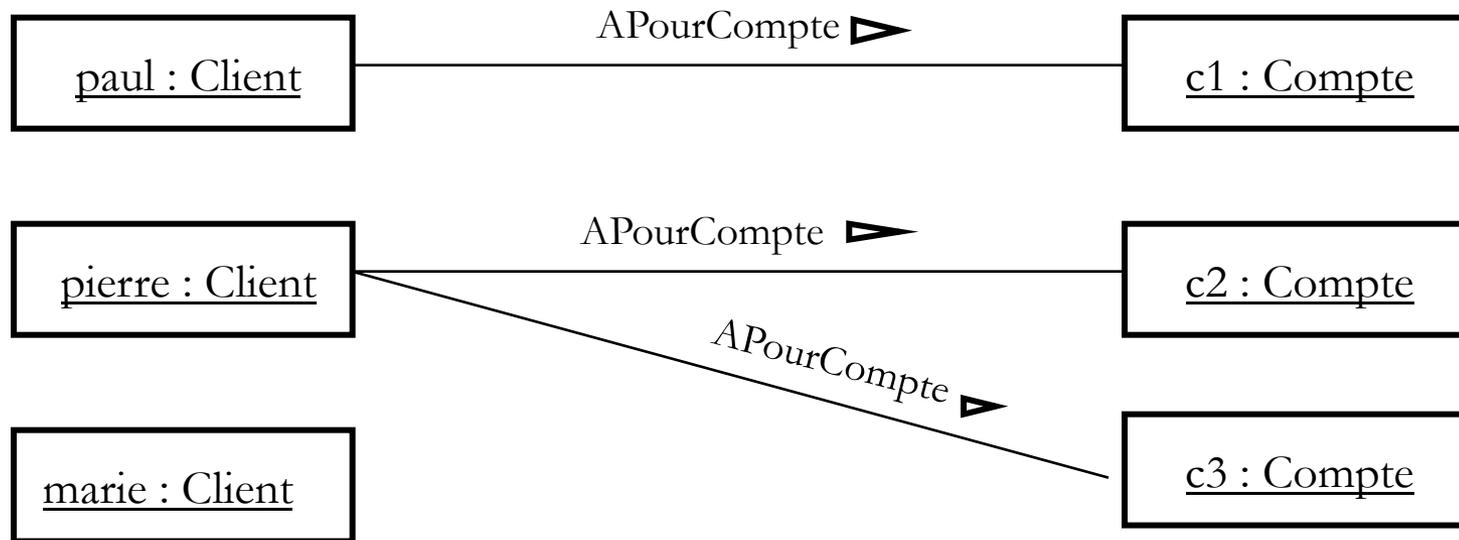


Diagramme d'objets

# Liens (entre objets)

---

Un **lien** indique une connexion entre deux objets

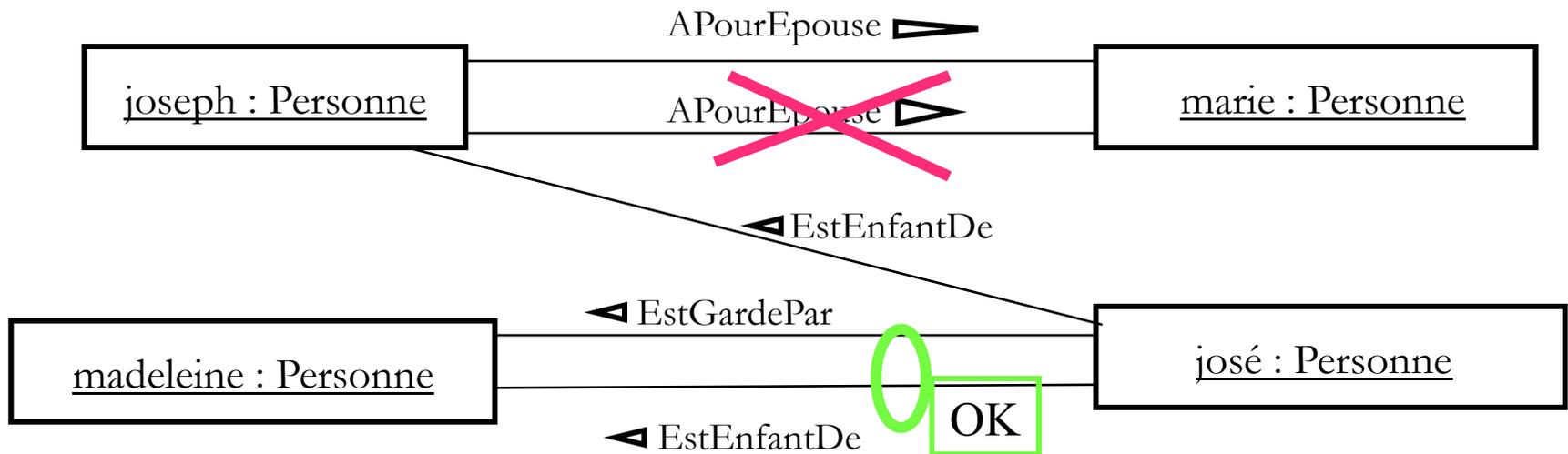


Note de style :

- les noms des liens sont des formes verbales et commencent par une majuscule
- ► indique le sens de la lecture (ex: « paul *APourCompte* c1 »)

# Contrainte sur les liens

Au maximum un lien d'un type donné entre deux objets donnés



# Rôles

---

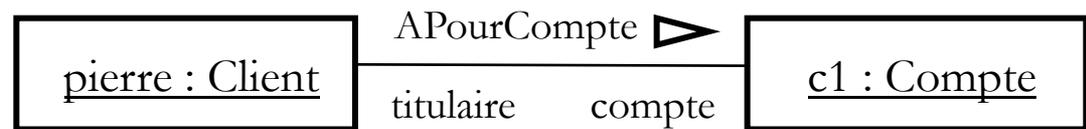
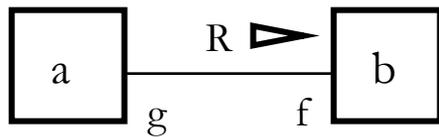
- Chacun des deux objets joue un rôle différent dans le lien



- Note de style :
  - choisir un groupe nominal pour désigner un rôle
  - si un nom de rôle est omis, le nom de la classe fait office de nom (avec la première lettre en minuscule)

# 3 noms pour 1 concept

- utilisations différentes selon le contexte



- $a R b$
- $b$  "joue le role de"  $f$  "pour"  $a$
- $a$  "joue le role de"  $g$  "pour"  $b$

- $pierre$  a pour compte  $c1$
- $c1$  joue le role de compte pour pierre
- $pierre$  joue le role de titulaire pour  $c1$

# Associations (entre classes)

- Une association décrit un ensemble de liens de même "sémantique"



Diagramme  
de classes  
(modélisation)

M1

M0

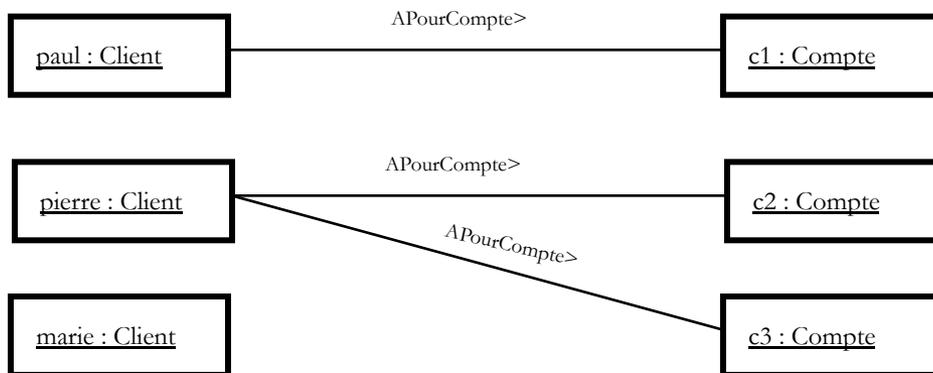


Diagramme  
d'objets  
(exemplaires)

# Association *vs.* Liens

---

- Un **lien** lie deux **objets**
- Une **association** lie deux **classes**
- Un **lien** est une instance d'**association**
- Une **association** décrit un ensemble de **liens**
- Des **liens** peuvent être ajoutés ou détruits pendant l'exécution (ce n'est pas le cas des associations)

Le terme "relation" ne fait pas partie du vocabulaire UML

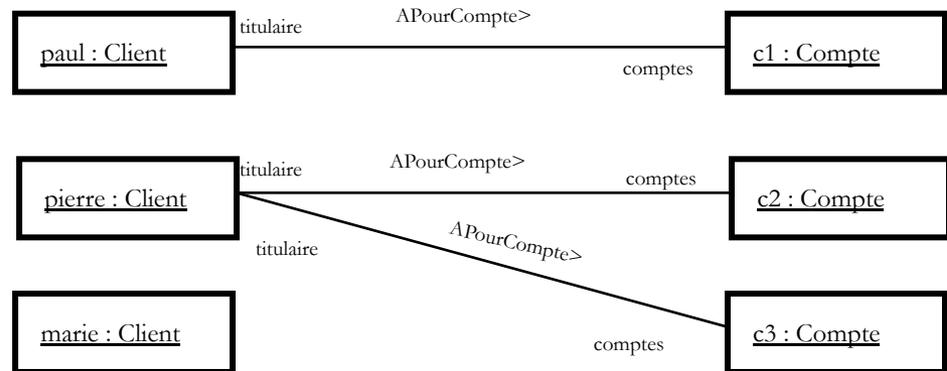
# Utiliser les rôles pour «naviguer»



M1

M0

paul.comptes = {c1}  
pierre.comptes = {c2,c3}  
marie.comptes = { }  
c1.titulaire = paul  
c2.titulaire = pierre  
c3.titulaire = pierre



Nommer en priorité les rôles

# Cardinalités d'une association

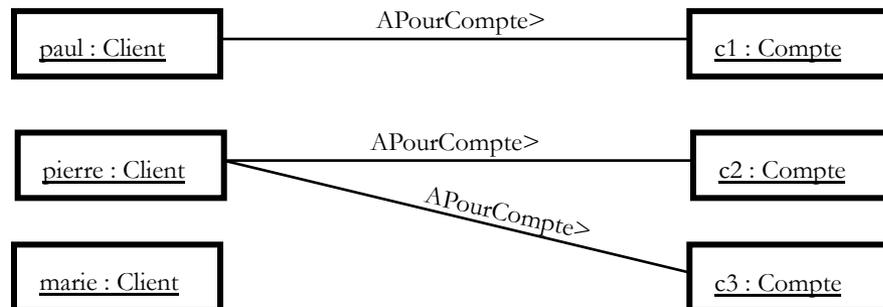
- Précise combien d'objets peuvent être liés à un seul objet source
- Cardinalité minimale et cardinalité maximale ( $C_{\min}..C_{\max}$ )
- Doivent être des constantes



« Tout **client** a toujours **0 ou plusieurs** comptes »  
« Tout **compte** a toujours **1 et 1 seul** titulaire »

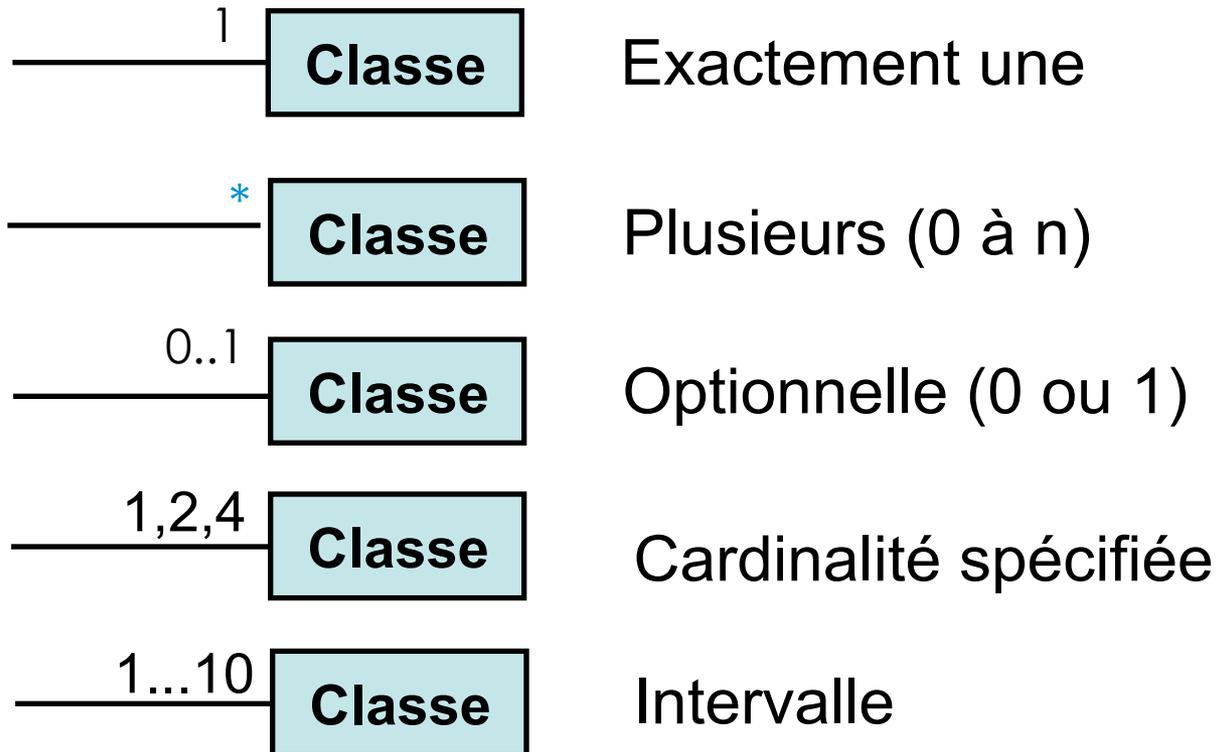
M1

M0



# Cardinalités d'une association

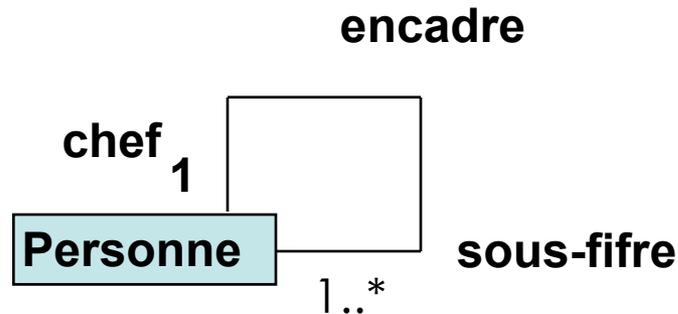
---



# Cas particulier d'association

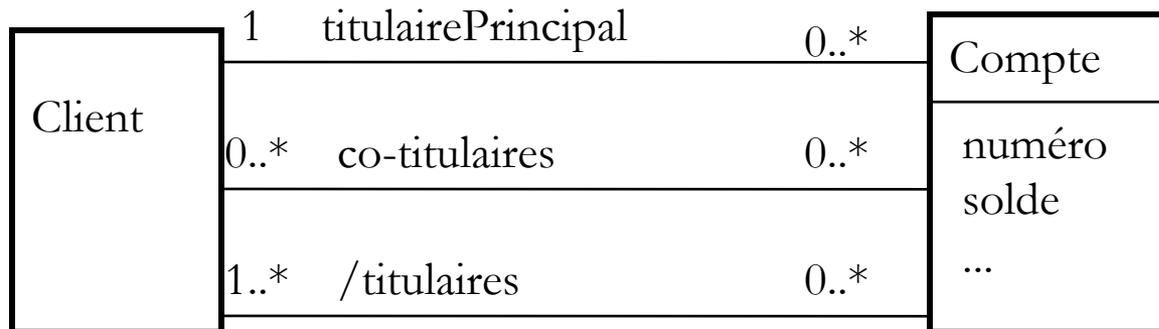
---

- Relation réflexive :



- Une relation réflexive lie des objets de même classe

# Contraintes entre associations



Les cardinalités sont loins d'être suffisantes pour exprimer toutes les contraintes...

... décrire les contraintes en langue naturelle (ou en OCL)

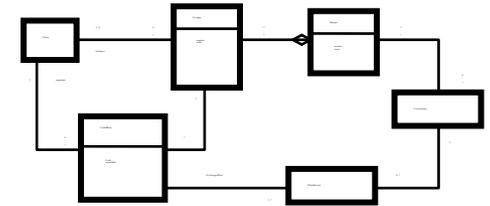
- (1) Un client ne peut pas être à la fois titulaire principal et co-titulaire d'un même compte.
- (2) Les titulaires d'un compte sont le titulaire principal et les co-titulaires le cas échéant

# Diagrammes de classes *vs.* d'objets

---

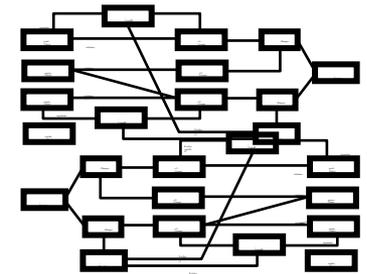
- Un diagramme de classes

- définit l'ensemble de tous les états possibles
- les contraintes doivent toujours être vérifiées



- Un diagramme d'objets

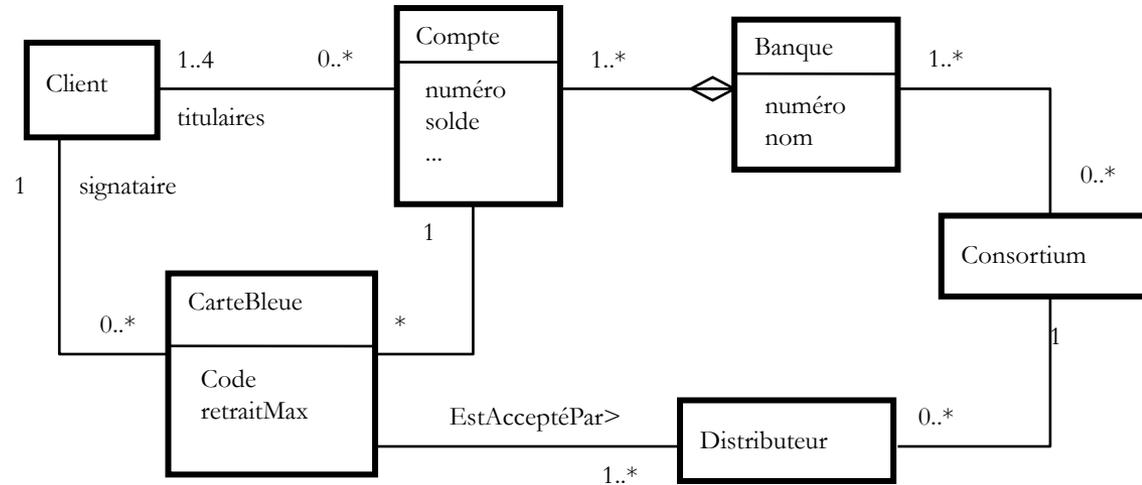
- décrit un état possible à un instant, un cas particulier
- doit être conforme au modèle de classes



- Les diagrammes d'objets peuvent être utilisés pour

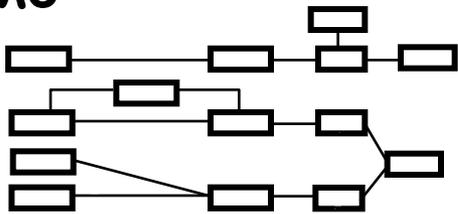
- expliquer un diagramme de classe (donner un exemple)
- valider un diagramme de classe (le « tester »)

# Diagrammes de classes *vs.* d'objets

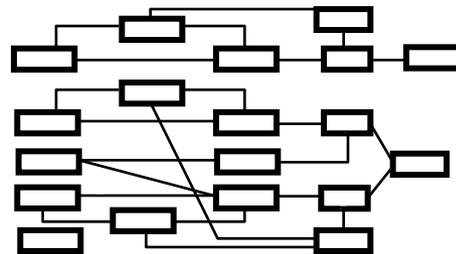


**M1**

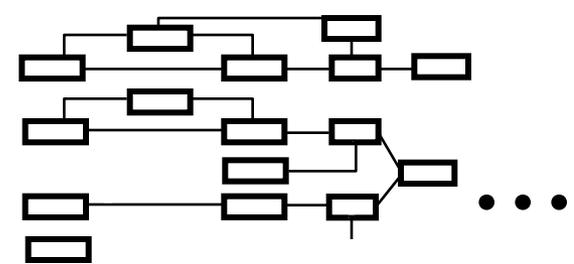
**M0**



$t_1$



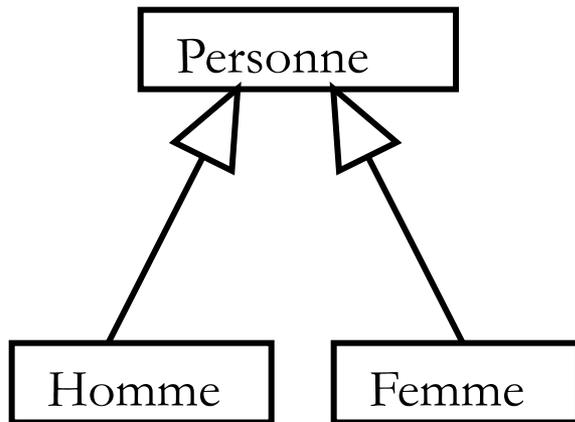
$t_2$



$t_3$

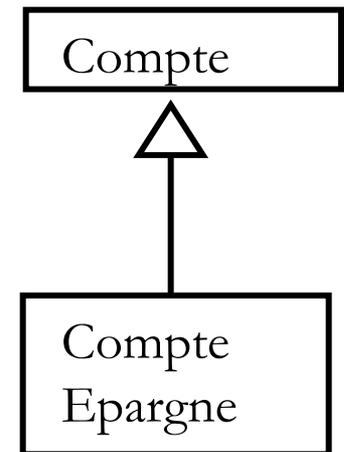
# Généralisation / Spécialisation

- Une classe peut être la généralisation d'une ou plusieurs autres classes.
- Ces classes sont alors des spécialisations de cette classe.



Cas général

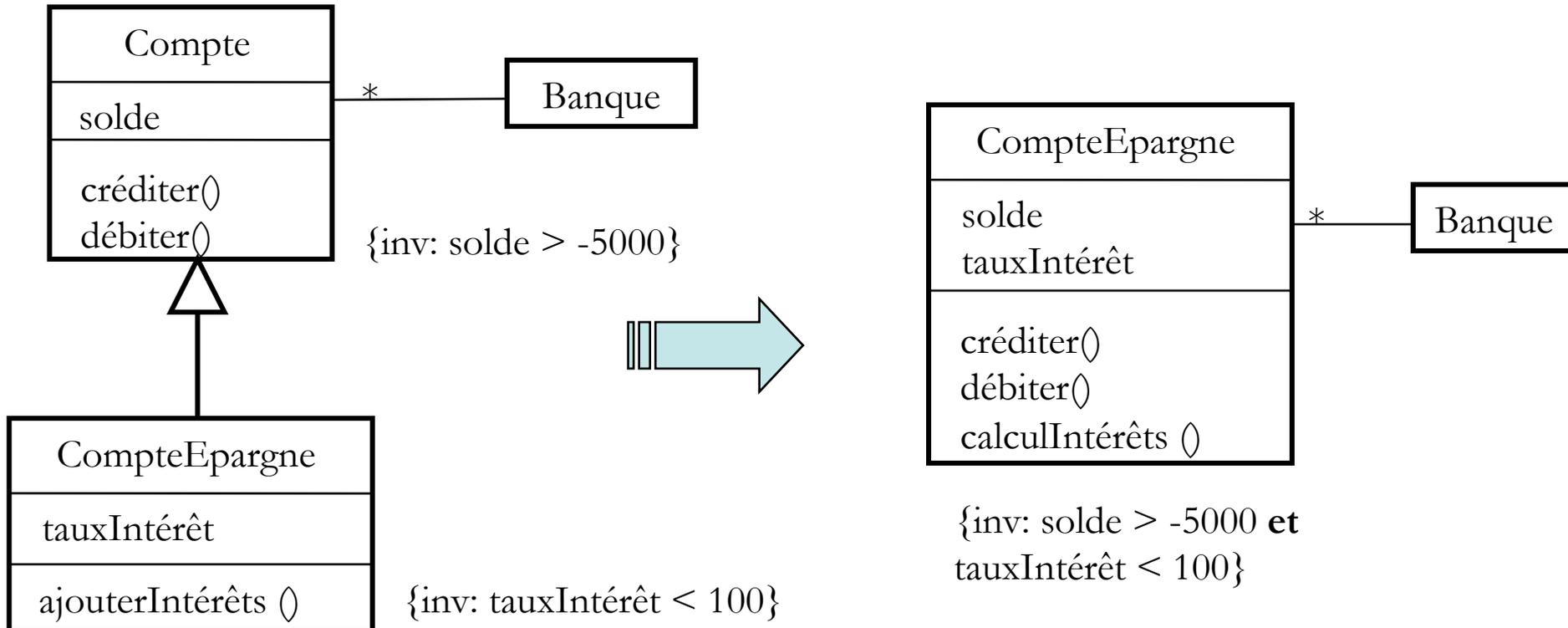
"Sous classes"  
Cas spécifique



- Deux points de vue lié (en UML) :
  - relation d'héritage
  - relation de sous-typage

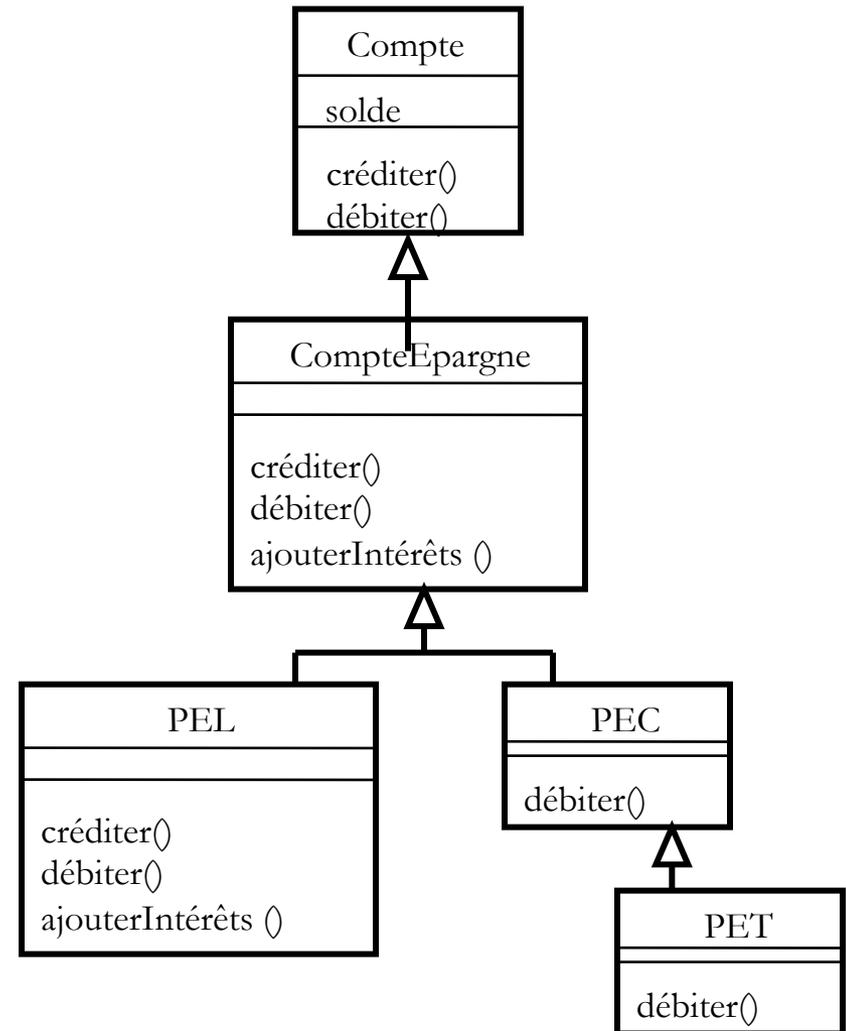
# Relation d'héritage

- Les sous-classes « héritent » des propriétés des super-classes (attributs, méthodes, associations, contraintes)



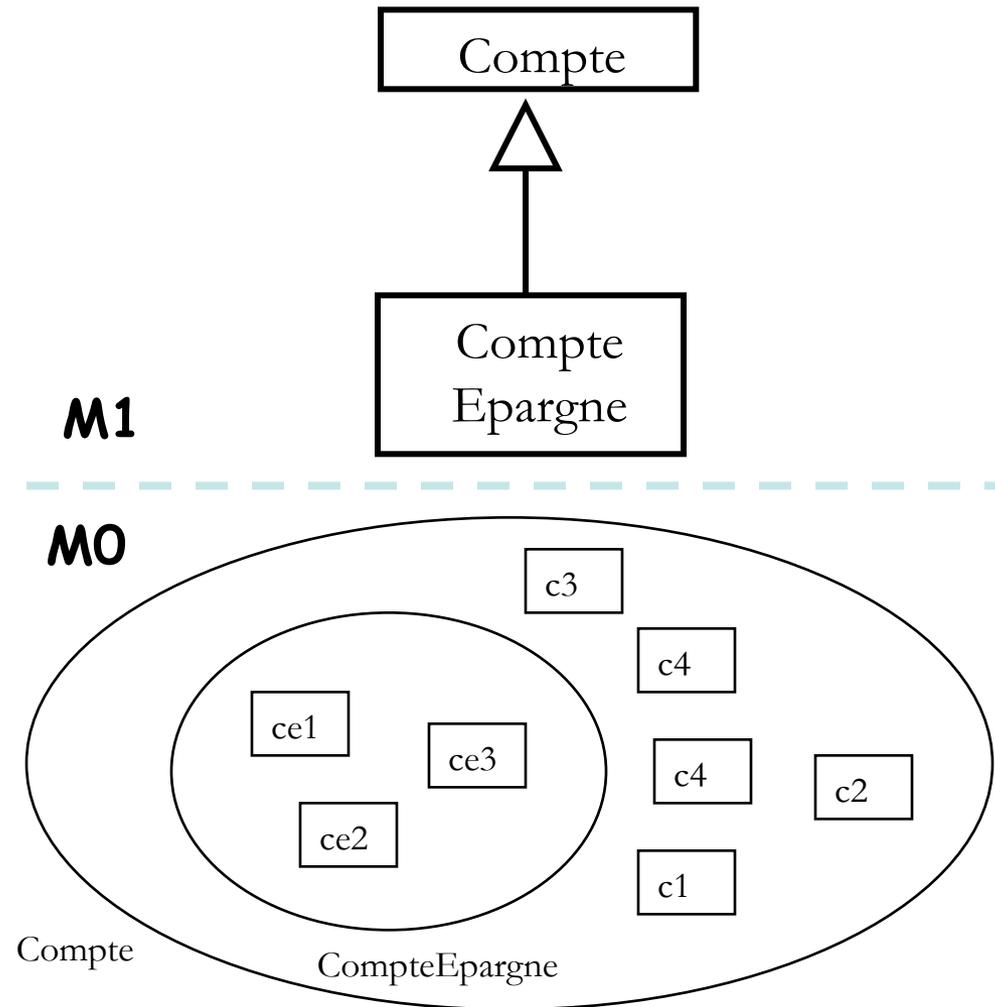
# Relation d'héritage et redéfinitions

- Une opération peut être "redéfinie" dans les sous-classes
- Permet d'associer des méthodes spécifiques à chaque classe pour réaliser une même opération



# Relation de sous typage : vision ensembliste

tout objet d'une sous-classe appartient également à la superclasse



# Synthèse des concepts de base

- Classe

- attribut
- méthode

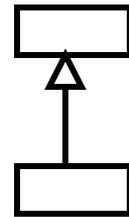


- Association

- rôle
- cardinalité

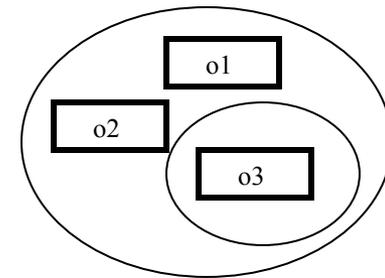
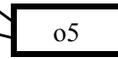
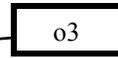
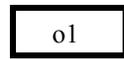
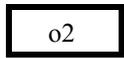
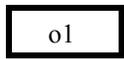


- Héritage



M1

M0



■ **Objet**

■ **Lien**

■ **Inclusion  
ensembliste**

# Outline

---

① UML History and Overview

② UML Language

① UML Functional View

② UML Structural View

③ UML Behavioral View

④ UML Implementation View

⑤ UML Extension Mechanisms

③ UML Internals

④ UML Tools

⑤ Conclusion

# Functional View

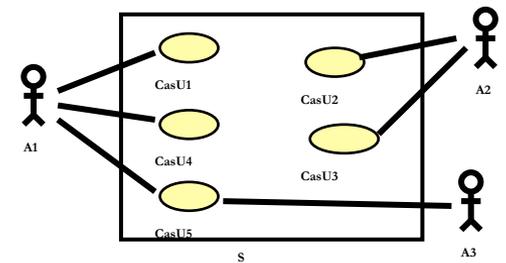
---

## Use Case Diagram

# Modèle des cas d'utilisation

---

- Buts :
  - modéliser le point de vue des **utilisateurs**
  - définir les limites précises du **système**
- Notation très simple, compréhensible par tous, y compris le client
- Permet de structurer :
  - les besoins (cahier des charges)
  - le reste du développement
- Modèle (principalement) pour communiquer



# Modèle des cas d'utilisation

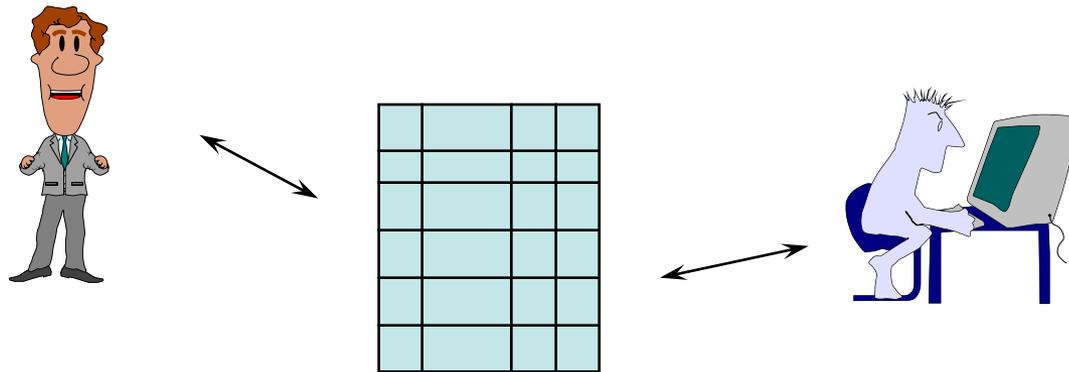
---

- Moyens :
  - Les acteurs UML
  - Les *use-cases* UML
  - Utilisation d'un dictionnaire du domaine

# Intérêt du dictionnaire

---

- Outil de dialogue
- Informel, évolutif, simple à réaliser
- Etablir et figer la terminologie
  - Permet de figer la terminologie du domaine d'application.
  - Constitue le point d'entrée et le référentiel initial de l'application ou du système.



# Use Case Diagram Example

---



Client

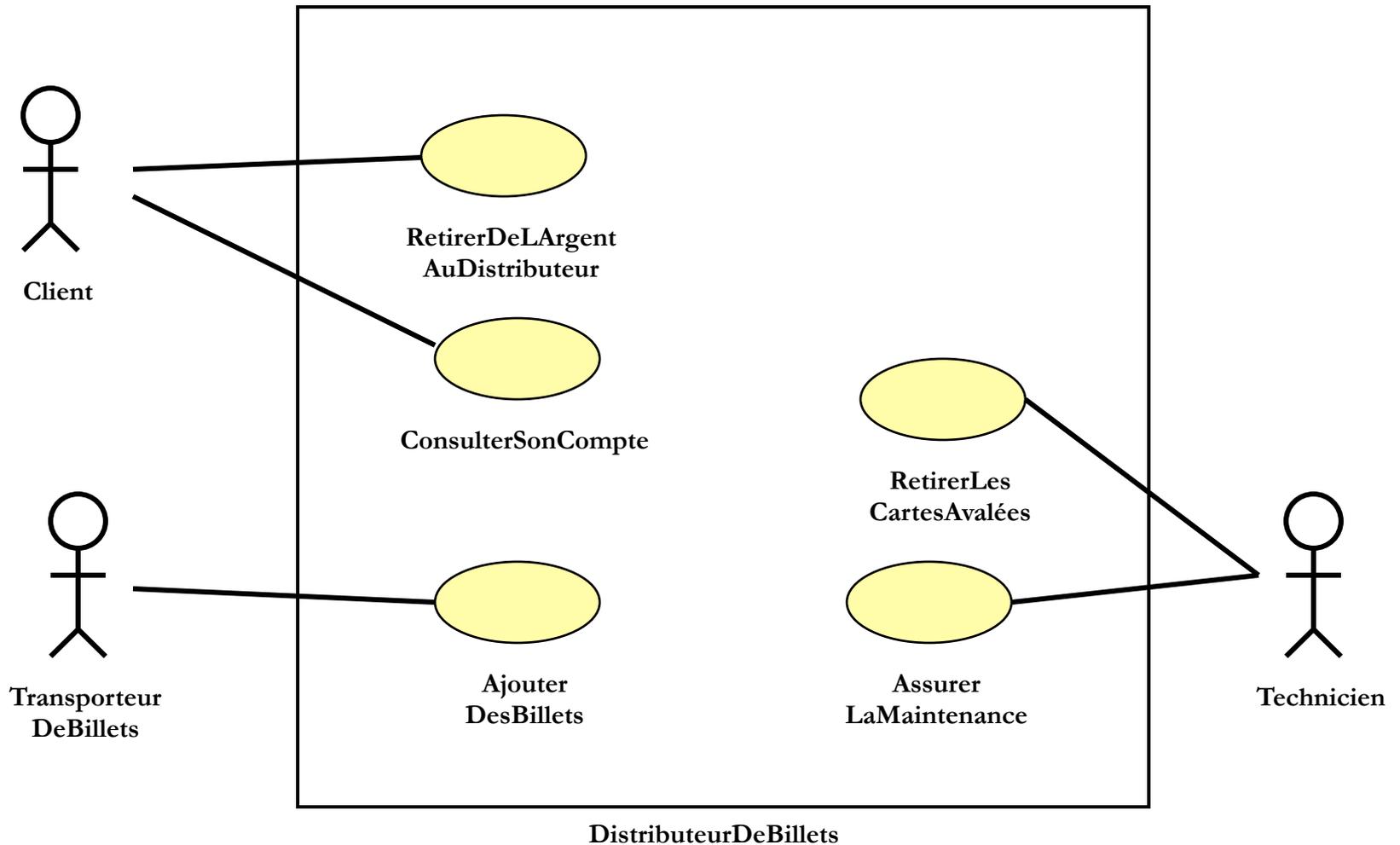


# Use Case Diagram Example

---

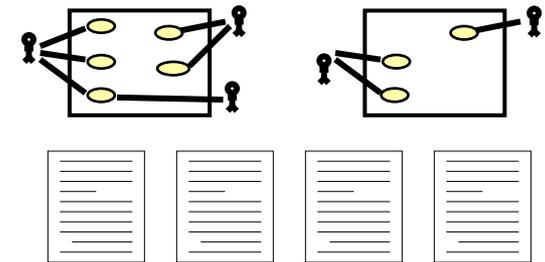
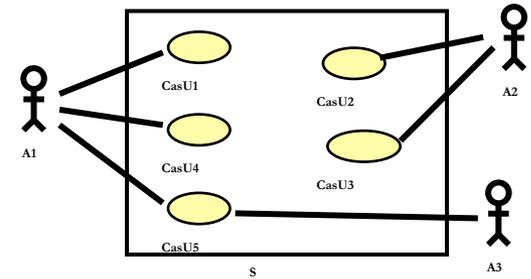


# Use Case Diagram Example



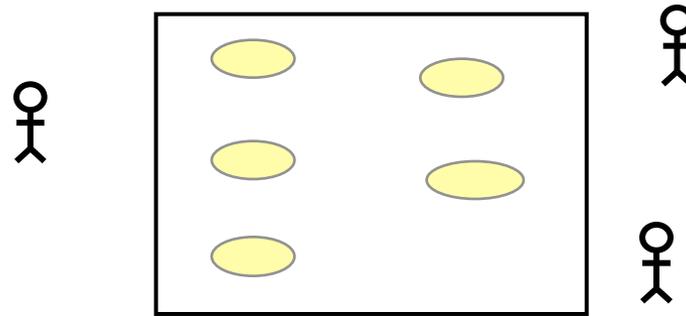
# Modèle de cas d'utilisation

- **Diagramme de cas d'utilisation**
- **Modèle de cas d'utilisation**
  - ◆ descriptions textuelles,
  - ◆ diagrammes de cas d'utilisation
  - ◆ diagrammes de séquences
  - ◆ dictionnaire de données
  - ◆ ...



# Éléments de base

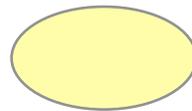
---



Acteurs



Cas d'utilisation



Systeme



# Cas d'utilisation (CU)

---

- Cas d'utilisation
  - une manière d'utiliser le système
  - une suite d'interactions entre un acteur et le système
- **Correspond à une fonction du système visible par l'acteur**
- **Permet à un acteur d'atteindre un but**
- **Doit être utile en soi**
- **Regroupe un ensemble de scénarii correspondant à un même but**

~~EnregistrerEntrée~~

RetirerDeLArgentAu  
Distributeur

~~Sidentifier~~

~~EntrerPendant  
LesHeuresDOuverture~~

~~TaperSonCode~~

# Systeme

---

- Le système est
  - modélisé par un ensemble de cas d'utilisation
  - vu comme une **boîte noire**
- Le système contient :
  - les cas d'utilisation,
  - mais pas les acteurs.
- Un modèle de cas d'utilisation permet de définir :
  - les fonctions essentielles du système,
  - les **limites du système**,
  - le système par rapport à son environnement,
  - **délimiter le cadre du projet !**

# Acteurs

---

- Un Acteur =
  - élément *externe*
  - qui *interagit* avec le système (prend des décisions, des initiatives. Il est "actif".)
- *Rôle* qu'un "utilisateur" joue par rapport au système



PorteurDeCarte



Gardien



Administrateur



Capteur  
AIncendie

# Acteurs vs. utilisateurs

---

Ne pas confondre les notions  
d'acteur et de personne utilisant le système

- Une même personne peut jouer plusieurs rôles  
ex: Maurice est directeur mais peut jouer le rôle de guichetier
- Plusieurs personnes peuvent jouer un même rôle  
ex: Paul et Pierre sont deux clients
- Un rôle par rapport au système plutôt qu'une position dans l'organisation  
ex: PorteurDeCarte plutôt qu'Enseignant
- Un acteur n'est **pas forcément** un être **humain**  
ex: un distributeur de billet peut être vu comme un acteur

# Différents types d'acteurs

---

Ne pas oublier d'acteurs :

- Utilisateurs principaux  
ex: client, guichetier
- Utilisateurs secondaires  
ex: contrôleur, directeur, ingénieur système, administrateur...
- Périphériques externes  
ex: un capteur, une horloge externe, ...
- Systèmes externes  
ex: système bancaires

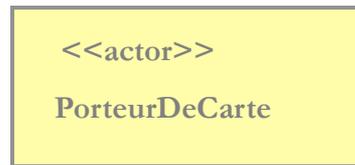
# Notation

---

- Notations alternatives pour les acteurs



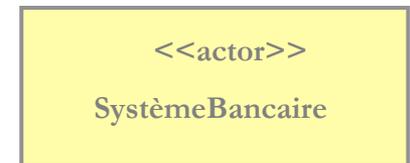
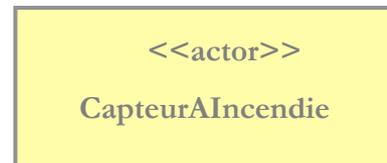
PorteurDeCarte



- **Note de style :**  
utiliser plutôt le stéréotype <<actor>> pour les acteurs non humains



PorteurDeCarte



# Un peu de méthodologie...

---

- **Description préliminaire du système**
  - **Choisir un identificateur**
    - Baptiser le système, le plus tôt possible
    - Risque d'être référencé dans toute la vie future de l'entreprise
  - **Brève description textuelle** (quelques lignes max.)
- **Description préliminaire des acteurs**
  - choisir un **identificateur représentatif** de son rôle
  - donner une brève **description textuelle**
- **Description préliminaire des cas d'utilisation**
  - choisir un **identificateur représentatif**
  - donner une **description textuelle simple**
    - la fonction réalisée doit être comprise de tous
    - préciser ce que fait le système, ce que fait l'acteur
    - pas trop de détails, se concentrer sur le scénario « normal »

# *Warning!!*

---

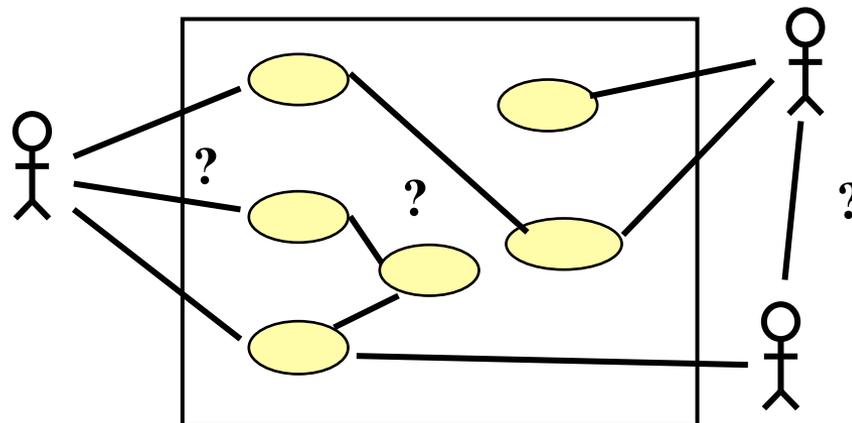
*"A common sign of a novice (or academic) use case modeler is a preoccupation with use case diagrams and use case relationships, rather than writing text. ... Use case diagrams and use case relationships are secondary in use case work. Use cases are text documents. Doing use case work means to write text."*

[Applying UML and Patterns, Craig Larman]

# Relations entre les éléments de base

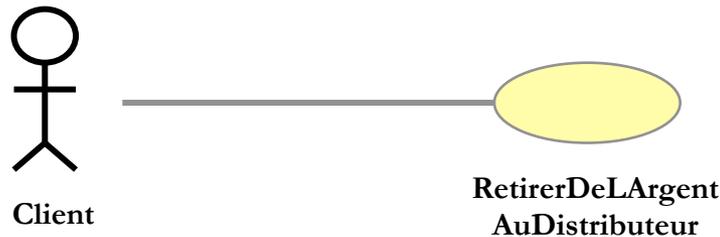
---

- Relations acteurs  $\leftrightarrow$  cas d'utilisation ?
- Relations acteurs  $\leftrightarrow$  acteurs ?
- Relations cas d'utilisation  $\leftrightarrow$  cas d'utilisation ?

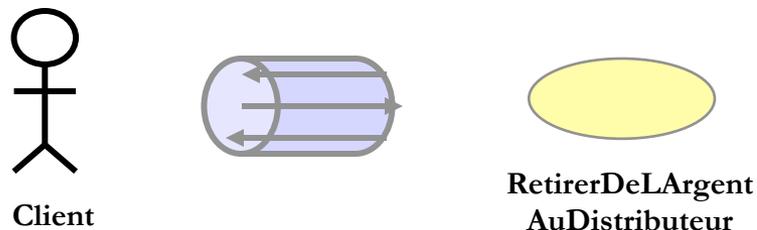


# Relation acteur $\leftrightarrow$ cas d'utilisation

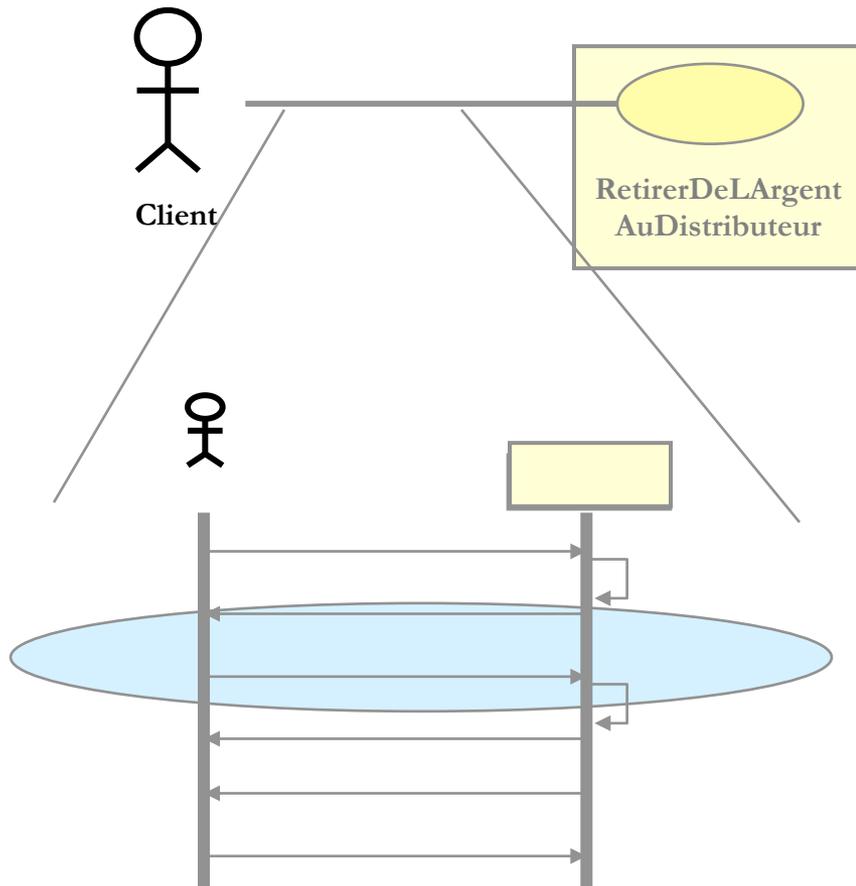
---



- Point de vue besoin: représente la possibilité d'atteindre un but
- Point de vue système: représente un canal de communication
  - Echange de messages, potentiellement dans les deux sens
  - Protocole particulier concernant le cas d'utilisation considéré

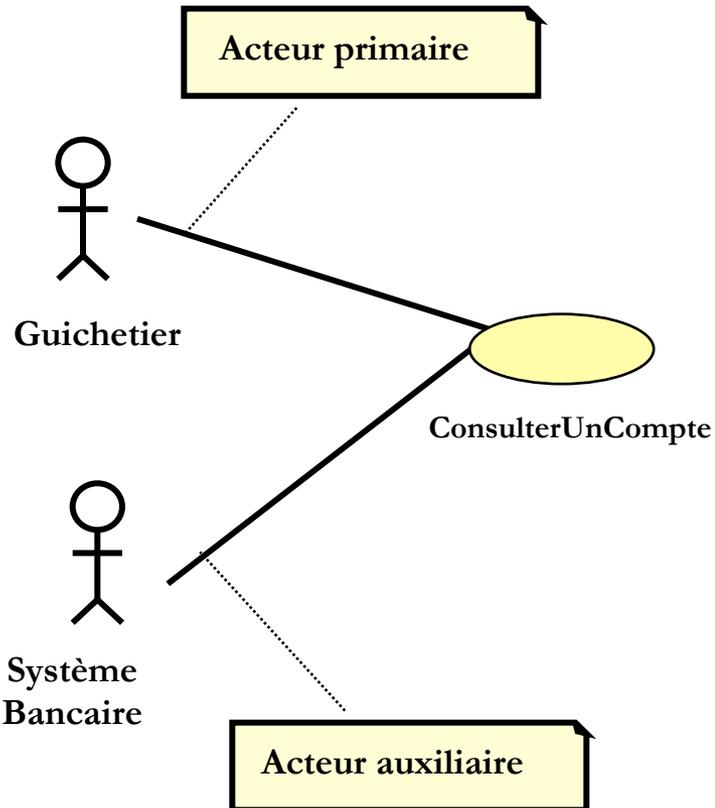


# Relation acteur $\leftrightarrow$ cas d'utilisation



- Description via des diagrammes de séquences "systèmes"
- Plus tard dans le cours ...

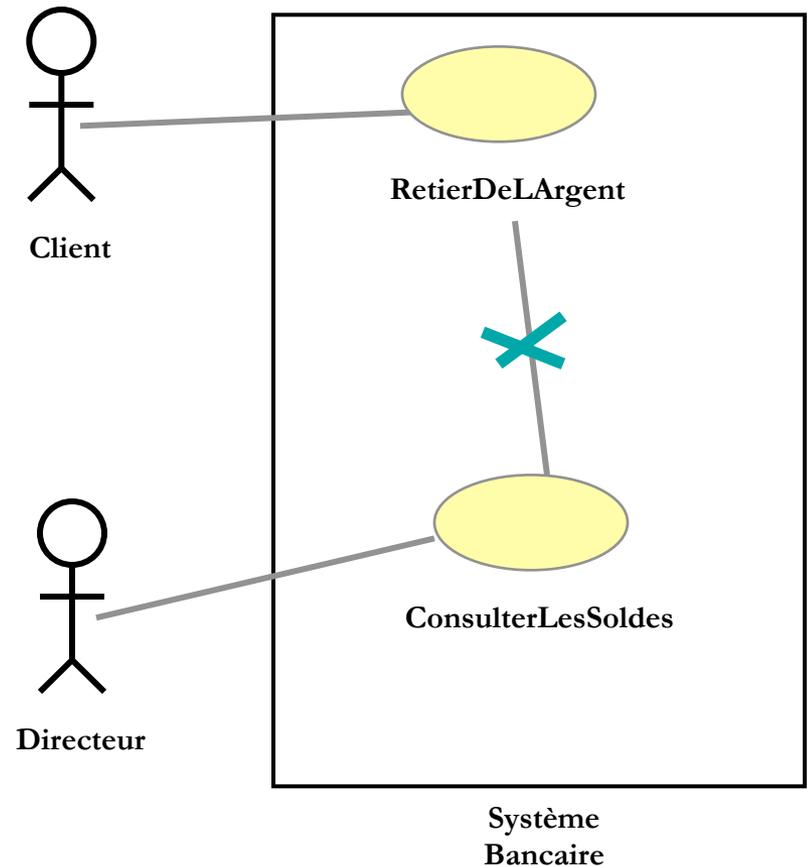
# Relation acteur $\leftrightarrow$ cas d'utilisation



- Acteur "primaire" »
  - utilise le système comme outil pour réaliser son but
  - initie généralement la communication
- Acteur(s) "auxiliaire(s)"
  - interviennent suite à l'intervention de l'acteur primaire
  - offrent généralement leurs services au système

# Relation cas d'utilisation ↔ cas d'utilisation

- Communications internes non modélisées.
- UML se concentre sur la description du système et de ses interactions avec l'extérieur
- Formalisme bien trop pauvre pour décrire l'intérieur du système. Utiliser les autres modèles UML pour cela.
- Autres relations possibles entre CU (slides suivants)



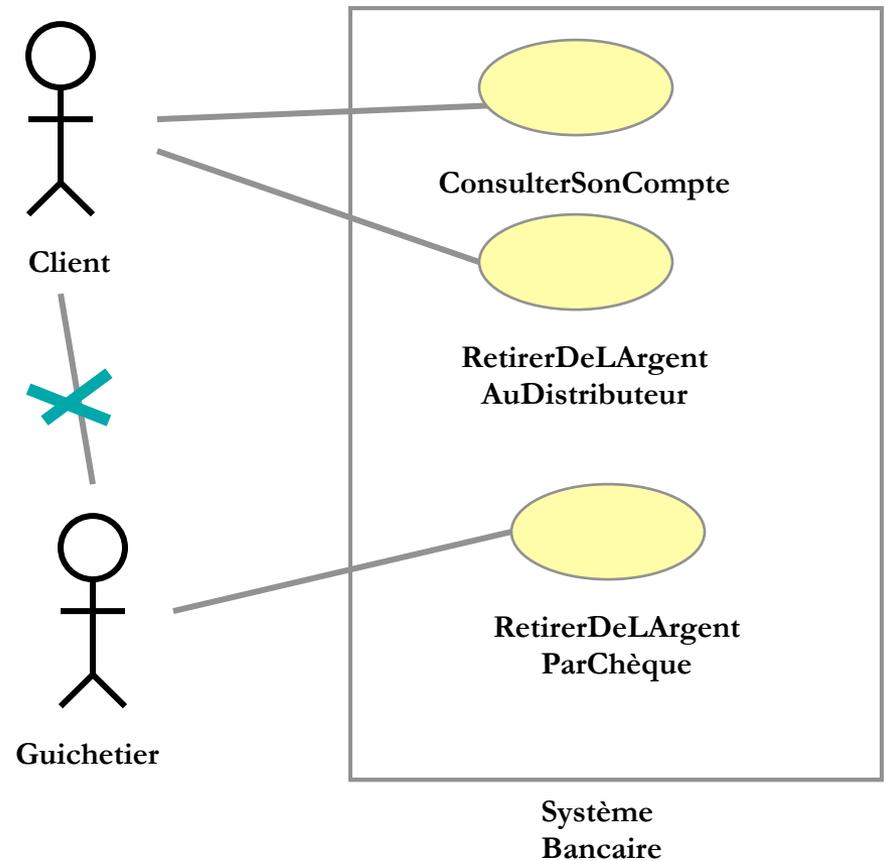
# Relation cas d'utilisation $\leftrightarrow$ cas d'utilisation

---

- Utilisation (*«include»* / *«uses»*)
  - Utilisation d'autres use-cases pour en préciser la définition
- Extension (*«extend»* / *«extends»*)
  - Un use-case étendu est une spécialisation du use-case père

# Relation acteur $\leftrightarrow$ acteur

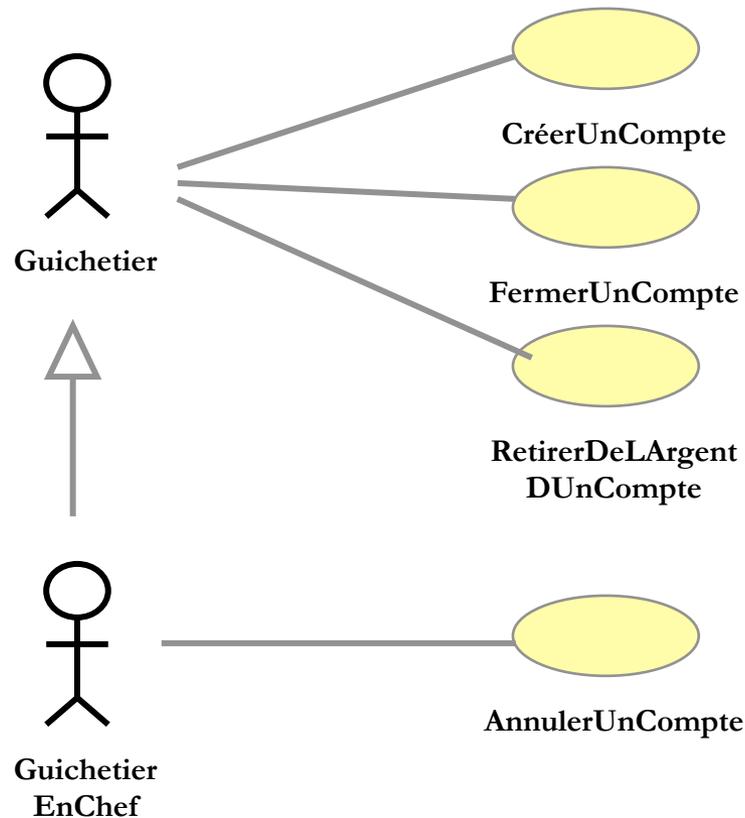
- Communications externes non modélisée
- UML se concentre sur la description du système et de ses interactions avec l'extérieur



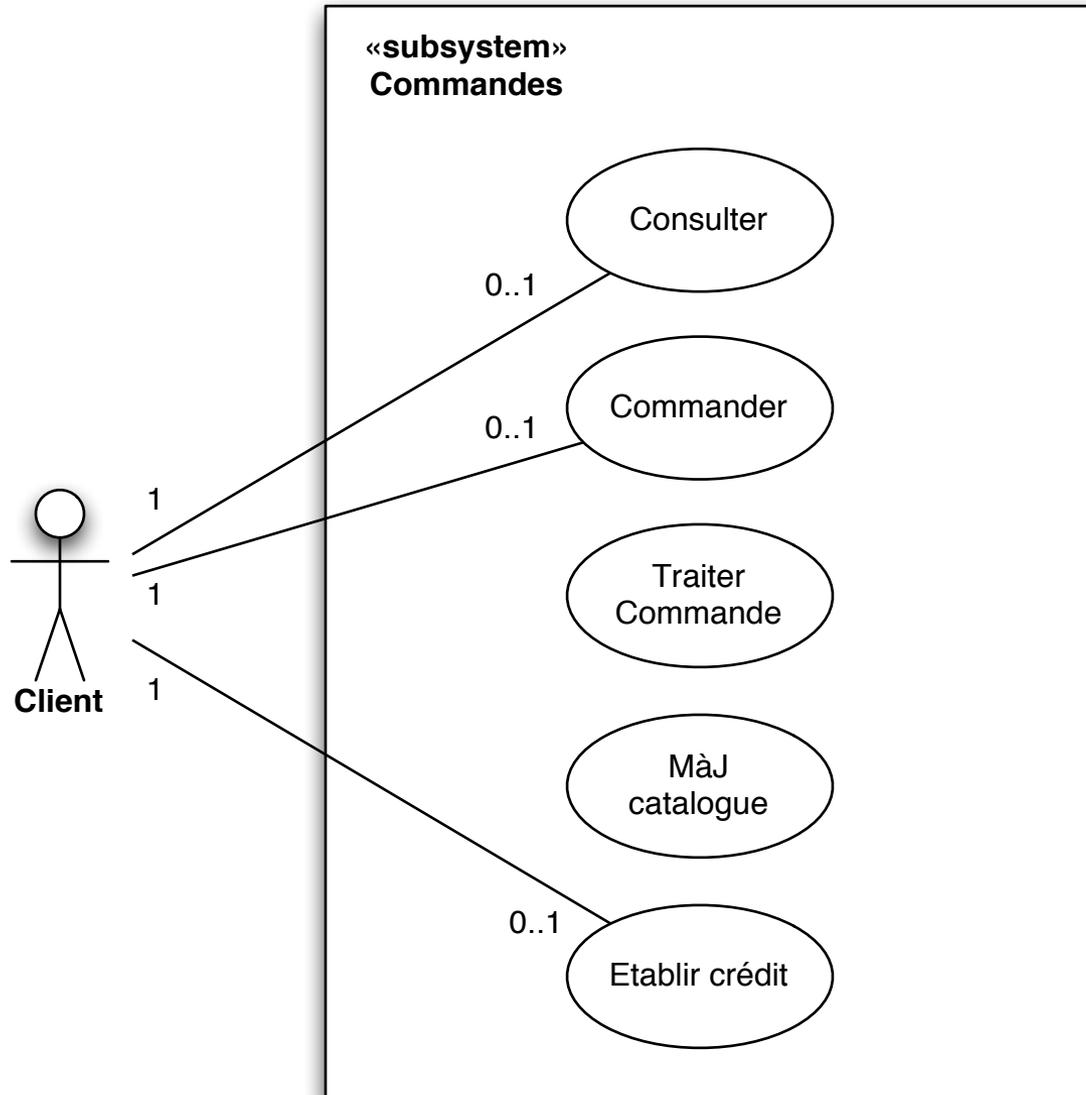
# Relation acteur $\leftrightarrow$ acteur : généralisation

---

- La seule relation entre acteurs est la relation de généralisation

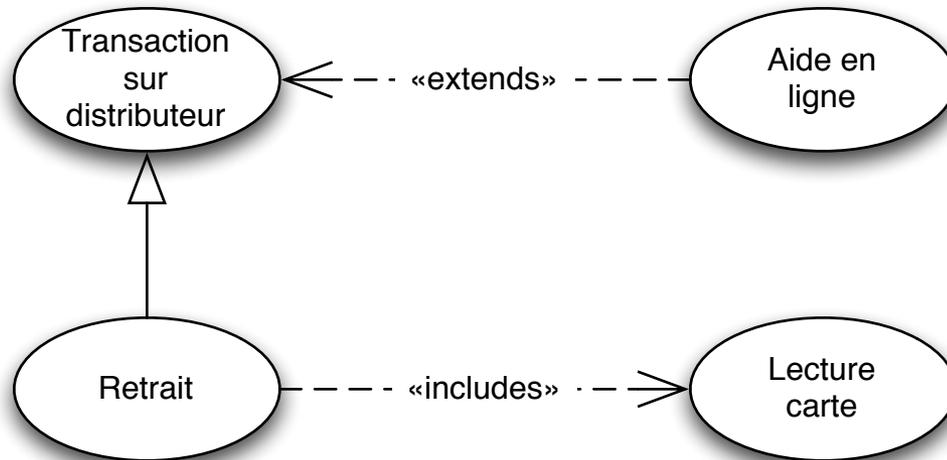


# Notation en résumé



# Notation en résumé

---



# Problèmes de la granularité

---

- A quel niveau décrire les cas d'utilisation ?
- Bonne question ... mais pas de réponse
  
- Trop haut
  - trop loin du système
  - trop abstrait et "flou"
  - trop complexe à décrire
  
- Trop bas
  - trop de cas d'utilisation
  - trop près de l'interface
  - trop loin des besoins métiers
  
- Conclusion: choisir le "bon" niveau ...

# Problèmes de la granularité

---

- Tous les cas d'utilisations n'ont pas à être au même niveau
- Différents niveaux de détail
- POURQUOI vs. COMMENT
- Décoration du niveau (*e.g.*, selon Cockburn)
- Non standardisé mais intuitif et utile

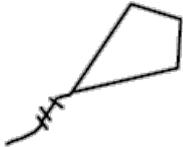
# Niveaux d'abstractions

Clouds  
Level



- Trop haut

Kite  
Level



- Niveau résumé : décrit un regroupement correspondant à un objectif plus global

Sea  
Level



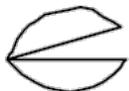
- **Niveau normal** : décrit un but de l'acteur qu'il peut atteindre via une interaction avec le système

Fish  
Level



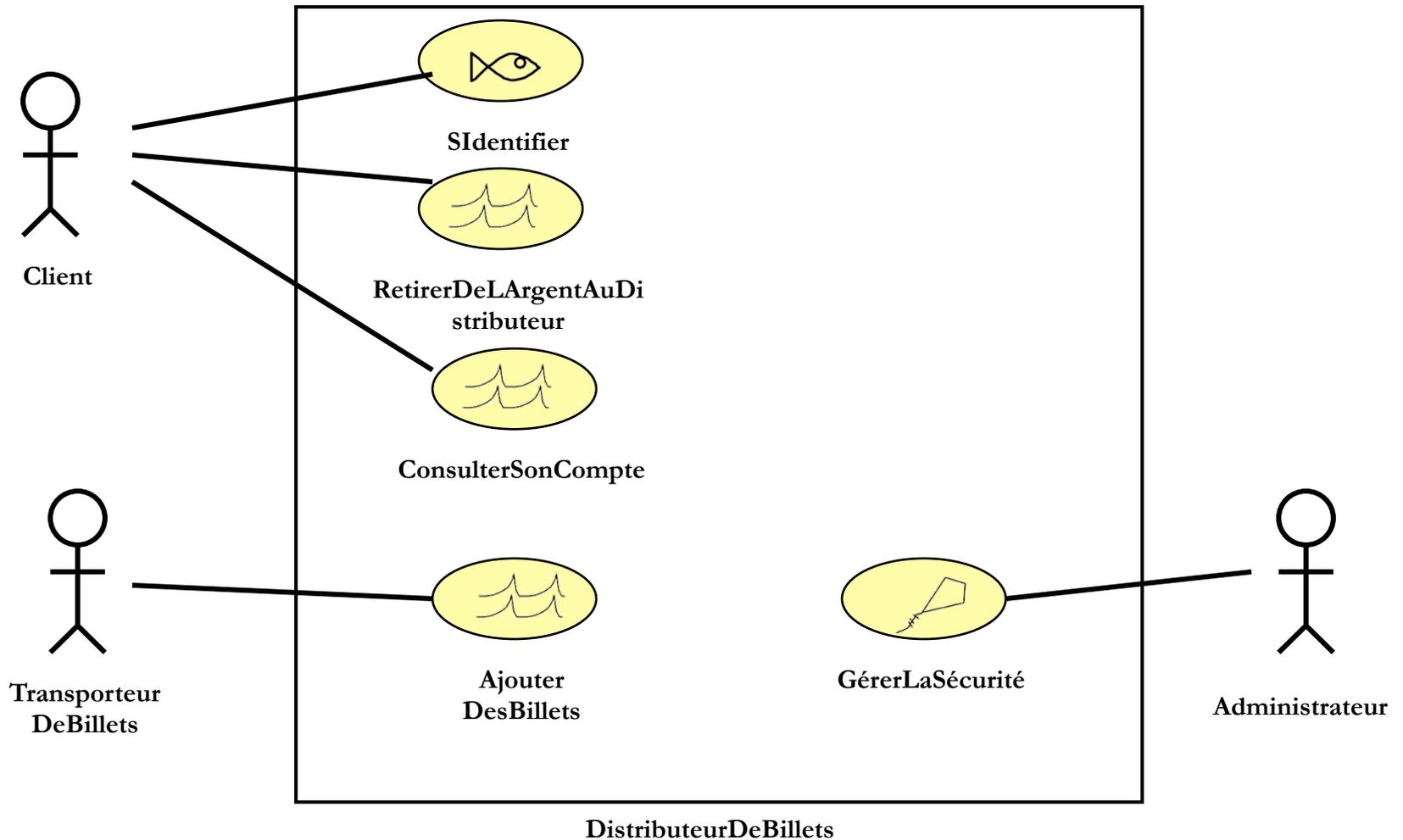
- Niveau détaillé : décrit une interaction avec le système, pas un but en soi

Clam  
Level



- Trop bas

# Exemple de marquage



# Cas d'utilisation *vs.* Scénarii

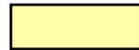
---

- Un scénario est un exemple :
  - une manière particulière d'utiliser le système ...
  - ... par un acteur particulier ...
  - ... dans un contexte particulier.
- cas d'utilisation = ensemble de scénarios
- scénario = une exécution particulière d'un CU

# Diagramme de séquences système

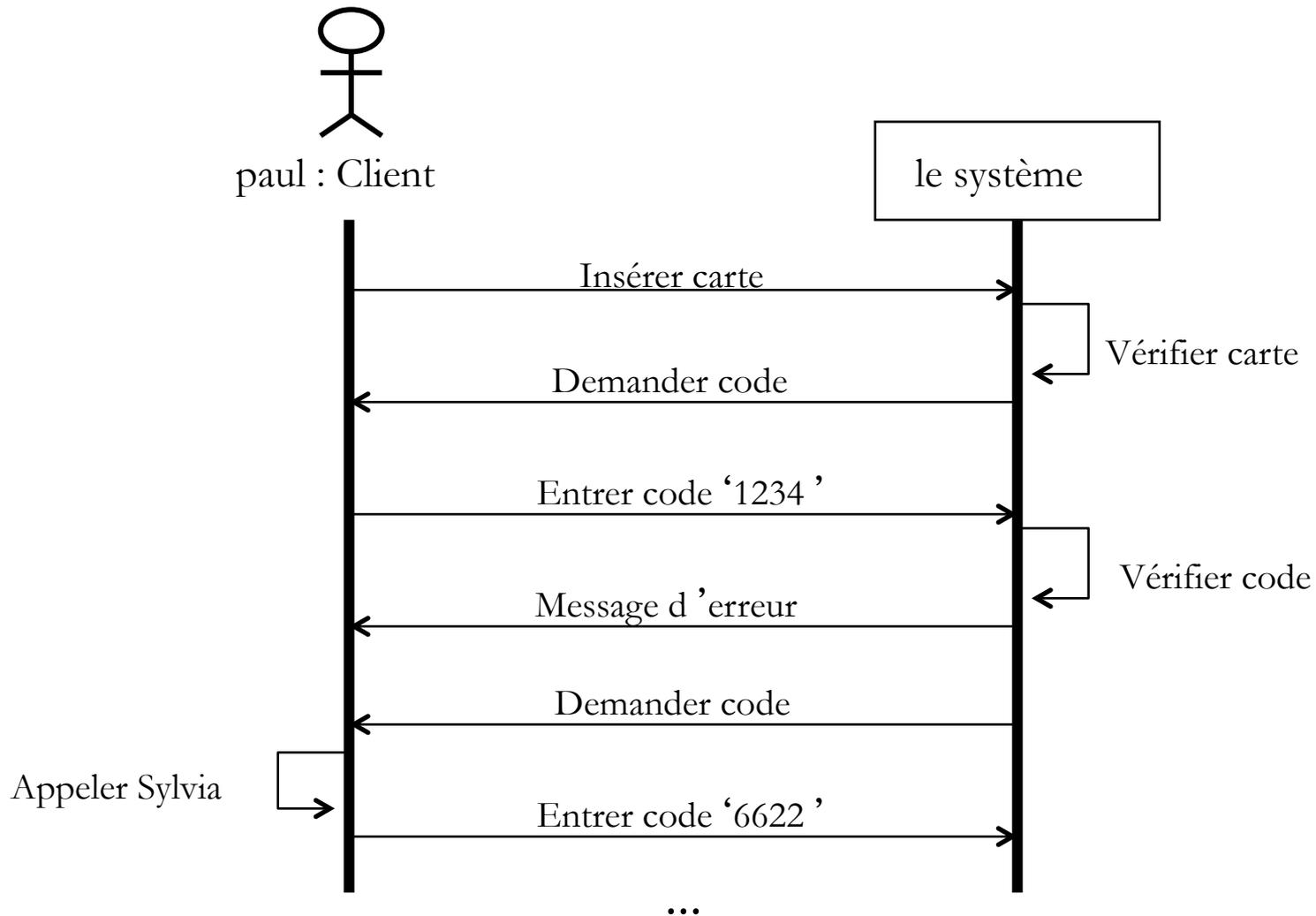
---

- Pour décrire un scénario : un diagramme de séquences
- Diagramme de séquences :
  - L'une des notations UML, une notation générale
  - Peut être utilisée dans de nombreux contextes
  - Permet de décrire une séquence de messages échangés entre différents objets
  - Différents niveaux de détails
- Pour décrire un scénario simple, deux objets : l'acteur et le système



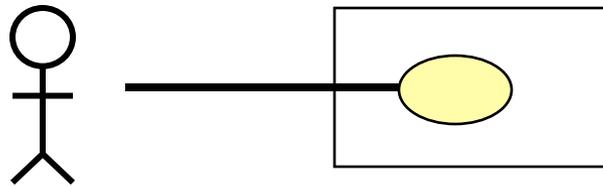
⇒ « Diagramme de séquences système »

# Exemple de diagramme de séquence système

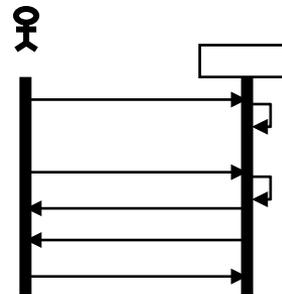
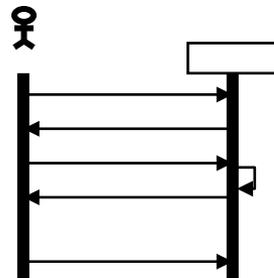
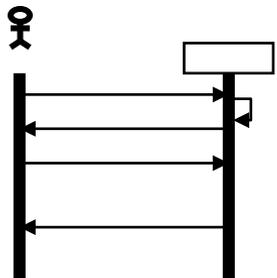


# Cas d'utilisation *vs.* scénarii

---



Niveau modèle



Niveau instances

# Un peu de méthodologie...

---

Stabilisation du modèle par **consensus**

**grandissant**

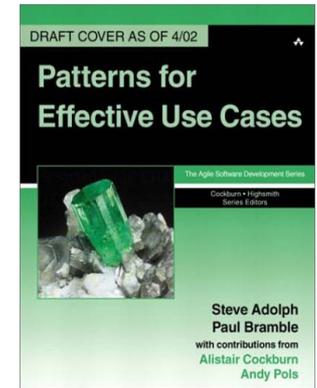
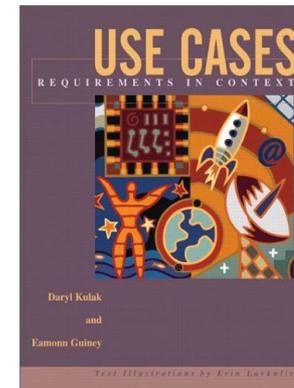
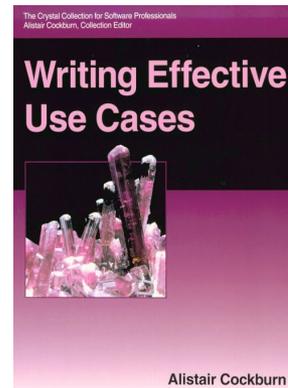
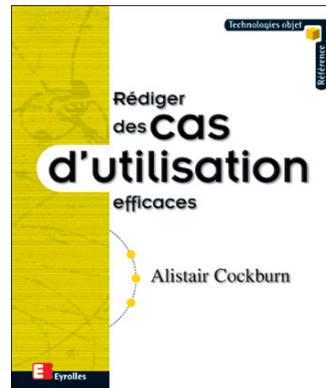
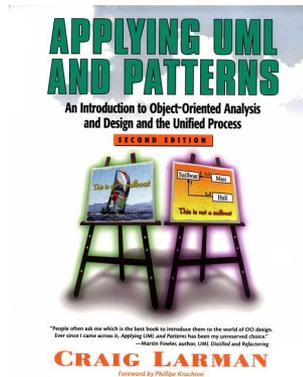
- Équivalent à définir une **table des matières** et des résumés pour chaque chapitre
  - Pas de règles strictes
  - Effectuer les meilleurs regroupement possibles
  - Rester simple !
  - Structuration possible en termes de paquetages
  - Culture d'entreprise

# Pour en savoir plus

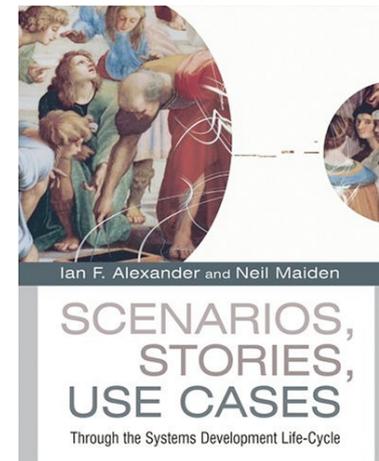
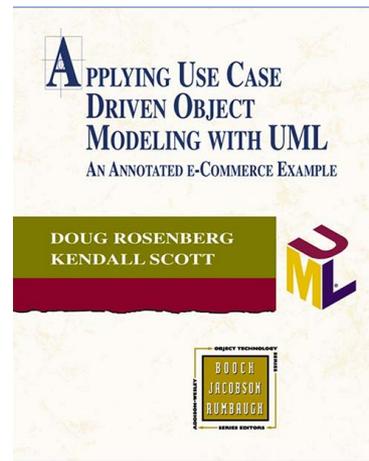
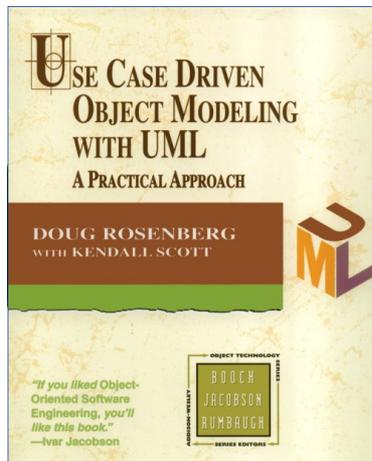
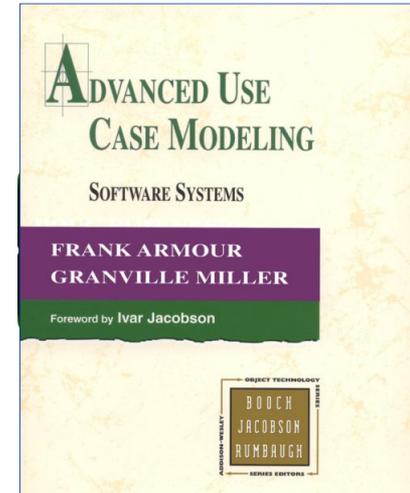
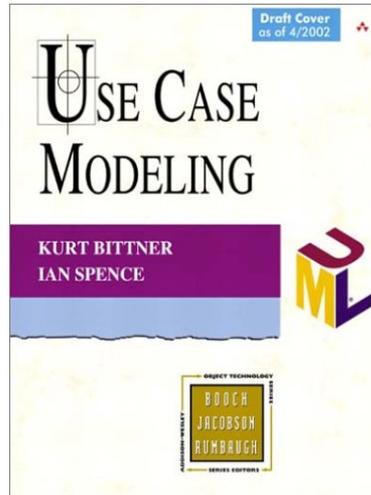
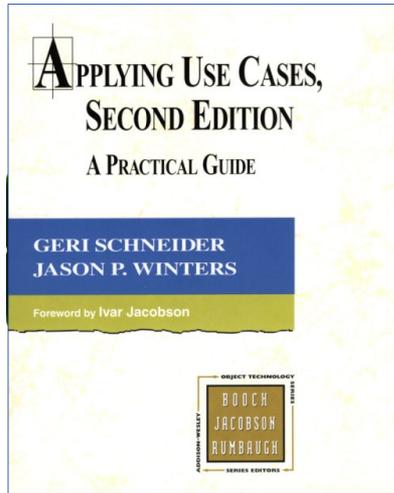
- Pour un template "standard" de description de cas d'utilisation :

<http://alistair.cockburn.us/Basic+use+case+template>

- Quelques livres :



# Pour en savoir *encore* plus



# Outline

---

① UML History and Overview

② UML Language

① UML Functional View

② UML Structural View

③ UML Behavioral View

④ UML Implementation View

⑤ UML Extension Mechanisms

③ UML Internals

④ UML Tools

⑤ Conclusion

---

# Diagramme de classe

## Utilisation avancée

# UML ?...une question de style et de contexte

---

- S'adapter ...
  - au niveau d'abstraction
  - au domaine d'application
  - aux outils utilisés
  - aux savants et ignorants
  - ingénierie vs. retro-ingénierie

# A. Raffinement du concept de CLASSIFICATEUR

---

- Les classificateurs (classifier) :
  - Classe
  - Classes abstraites
  - Énumération
  - Datatypes
  - Interfaces
  - Classes paramétrées

# Déclaration d'attributs

---

- Syntaxe: `[/][visibility]name[:type][multiplicity][=default][{prop}]`
- Propriétés:
  - `{readOnly}`, `{union}`, `{subset <p>}`, `{redefines <p>}`, `{ordered}`, `{unique}`, `{non-unique}`, `{prop-constraint}`.
  - Visibilité: `[+,~,#,-]`
- Peuvent être soulignés (attributs de classe)

```
age
+age
/age
- solde : Integer = 0
# age : Integer [0..1]
# numsecu : Integer {frozen}
# motsClés : String [*] {addOnly}
nbPersonne : Integer
```

# Attribut dérivé

---

- Attribut dont la valeur peut être déduite d'autres éléments du modèle
  - *e.g.*, *âge* si l'on connaît la date de naissance
  - notation : /age
- En termes d'analyse, indique seulement une contrainte entre valeurs et non une indication de ce qui doit être calculé et ce qui doit être mémorisé

# Déclaration d'opérations

---

- Syntaxe: [ / ] [ visibilité ] nom [ ( params ) ] [ : type ] [ { props... } ]  
params := [ in | out | inout ] nom [ : type ] [ =default ] [ { props... } ]

```
/getAge()  
+ getAge() : Integer  
- updateAge( in date : Date ) : Boolean  
# getName() : String [0..1]  
+getAge() : Integer {isQuery}  
+addProject() : { concurrency = sequential }  
+addProject() : { concurrency = concurrent }  
+main( in args : String [*] {ordered} )
```

# Propriétés des opérations

---

- **redefines** <operation-name>
- **query**
- **concurrent**: des appels multiples peuvent arriver simultanément et être traités de façon concurrente
- **guarded**: des appels multiples peuvent arriver simultanément mais seront traités un à la fois
- **sequential**: un seul appel peut arriver à la fois.

# Classe abstraite

---

- Encapsule un comportement commun
- Ne possède pas d'instances
- Possède des opérations abstraites
- Dans les langages de programmation OO :
  - deferred en Eiffel
  - abstract en Java
  - pure virtual en C++
  - méthodes `^self shouldNotImplement` en Smalltalk

*Etudiant*

**Etudiant  
{abstract}**

# Data Type

---

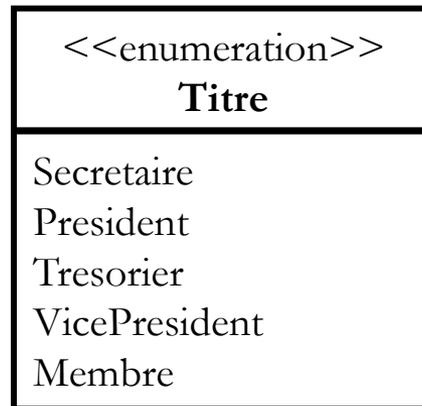
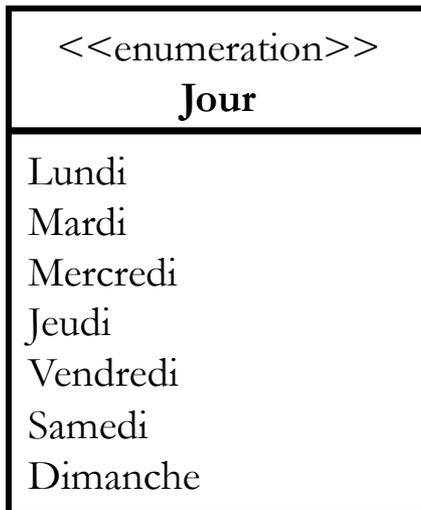
- Type, dont:
  - les valeurs n'ont pas d'identité
  - les opérations sont des fonctions “pures”
  
- Exemples de Data Type :
  - Enumeration
  - Types primitifs:
    - Boolean, Integer, UnlimitedNatural, String

<b>«dataType»</b> <b>Date</b>
day : Integer month : Integer year : Integer

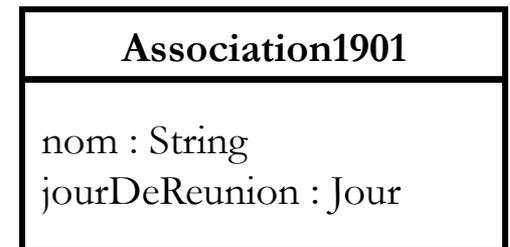
# Énumération (Data Type particulier)

---

- Définition



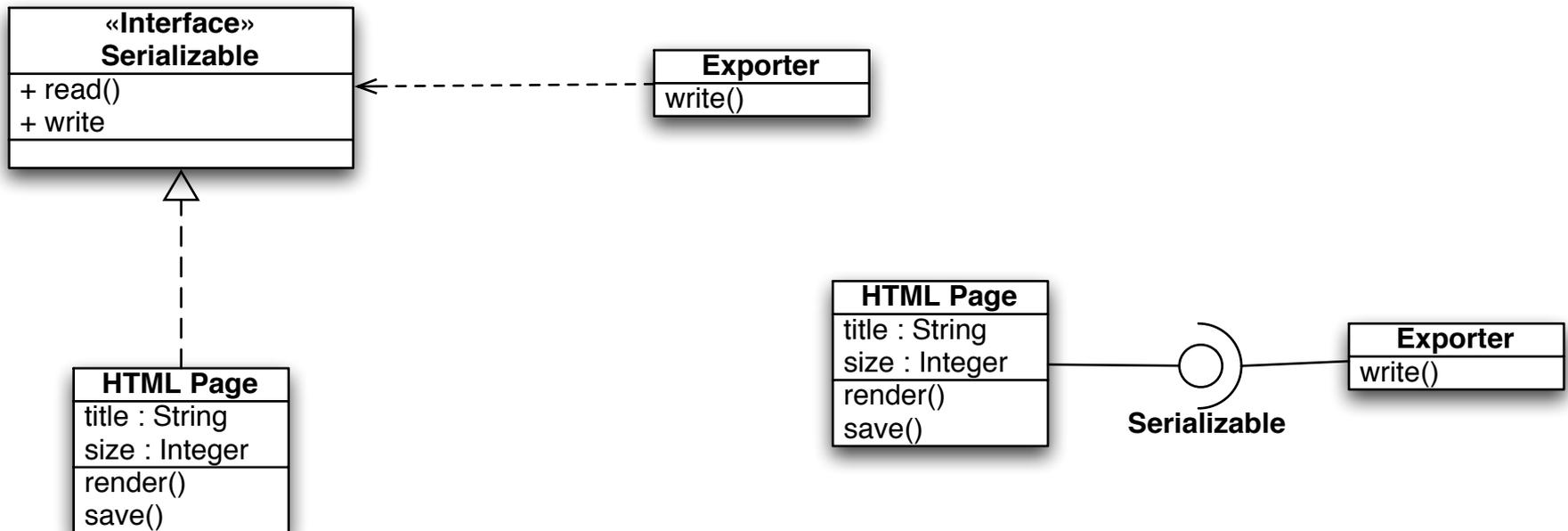
- Utilisation



# Interface

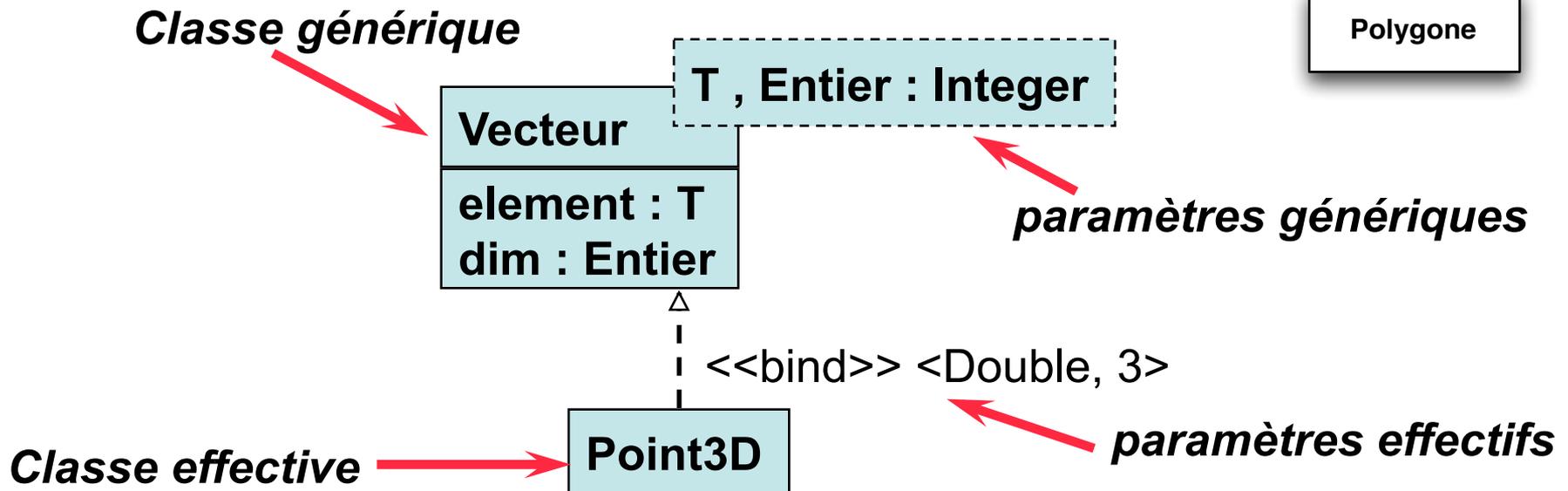
---

- Une déclaration d'un ensemble cohérent d'opérations
- Une interface peut être implémentée ou demandée



# Type paramétré (classe générique)

- Indispensable pour les classes conteneurs (*e.g.*, listes)
- Existe en Ada, Eiffel, C++ (templates) et Java (>1.5)
- N'est pas nécessaire en Smalltalk et Python



# + # - Visibilité des éléments

---

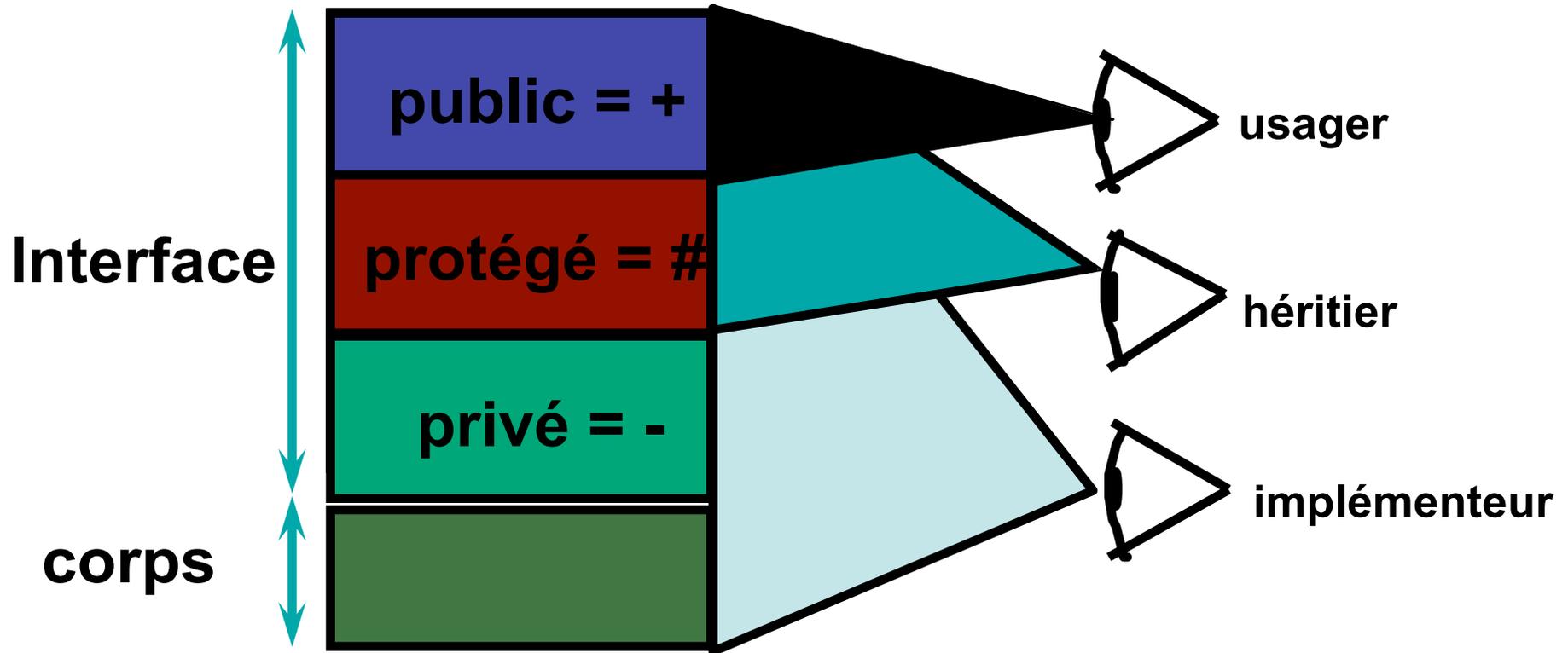
- Restreindre l'accès aux éléments d'un modèle
- Contrôler et éviter les dépendances entre classes et paquetages

+	public	visible
#	protégé	visible dans la classe et ses sous-classes
-	privé	visible dans la classe uniquement
~	package	visible dans la package uniquement

- Utile lors de la conception et de l'implémentation, pas avant !
- N'a pas de sens dans un modèle conceptuel (abstrait), *e.g.*, modèle d'analyse
- N'utiliser que lorsque nécessaire
- La sémantique exacte dépend du langage de programmation !

# + # - Visibilité des éléments

---



# B. Raffinement du concept d'ASSOCIATION

---

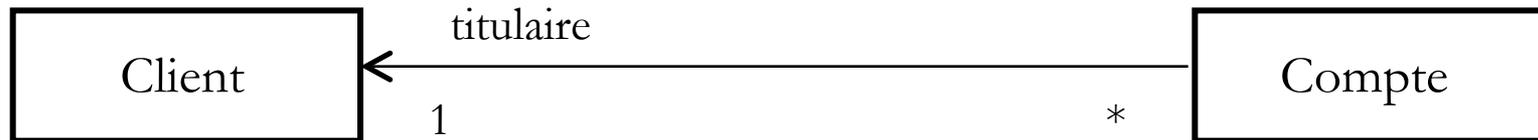
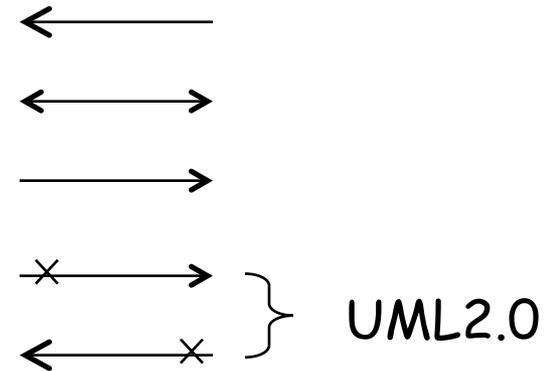
- Association uni/bi directionnelle
- Composition
- Agrégation
- {frozen}, {addonly}, {ordered}, {nonunique}
- Classes associatives
- Associations n-aires
- Associations attribuées
- Associations qualifiées

# Navigation

---

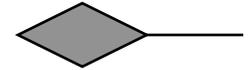
- Association unidirectionnelle

On ne peut naviguer que dans un sens

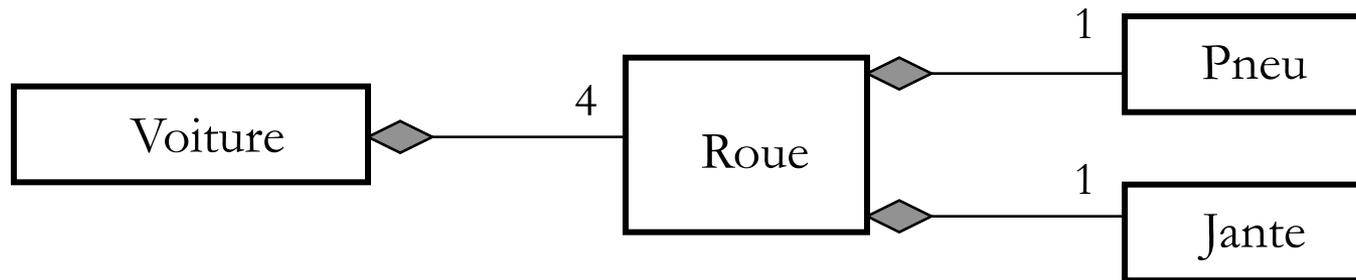


A priori, que dans les diagrammes de spécifications et d'implémentation  
En cas de doute, **ne pas mettre de flèche !!!**

# Composition



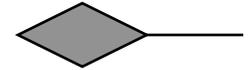
- Notion intuitive de "composants" et de "composites"



- composition =
  - cas particulier d'association
  - + contraintes décrivant la notion de "composant"...

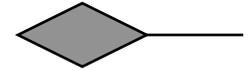
# Composition

---

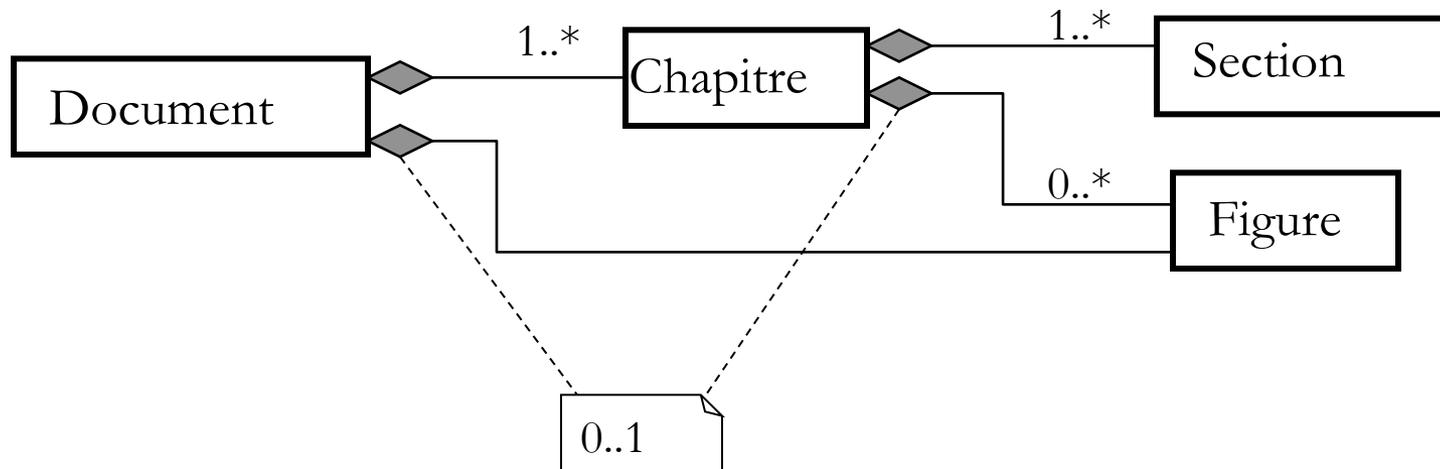


- Contraintes liées à la composition :
  - **Pas de partage** : un objet composant ne peut être que dans 1 seul objet composite
  - **Cycle de vie du composant lié au cycle de vie du composite** : si un objet composite est détruit, ses composants aussi
- Dépend de la situation modélisée !  
(e.g., vente de voitures vs. casse)

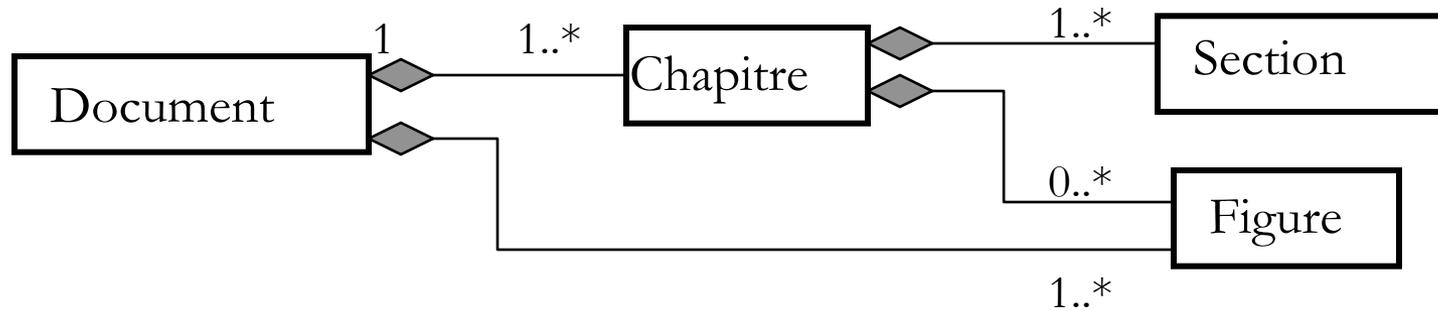
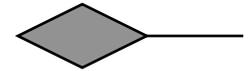
# Composition



- **Contraintes liées à la composition :**
  - **Pas de partage** : un objet composant ne peut être que dans 1 seul objet composite
  - **Cycle de vie du composant lié au cycle de vie du composite** : si un objet composite est détruit, ses composants aussi

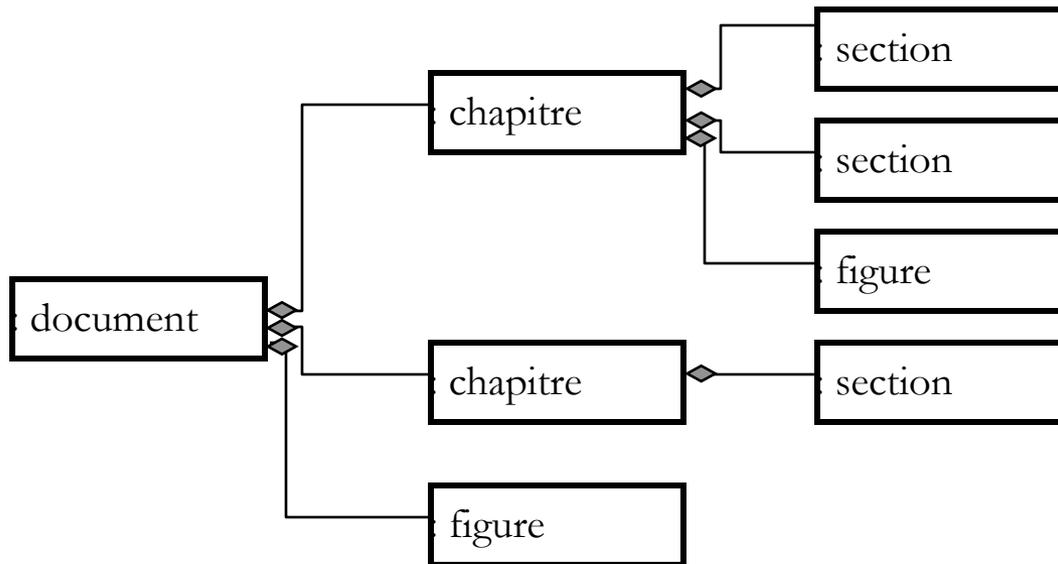


# Composition



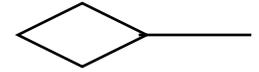
**M1**

**M0**

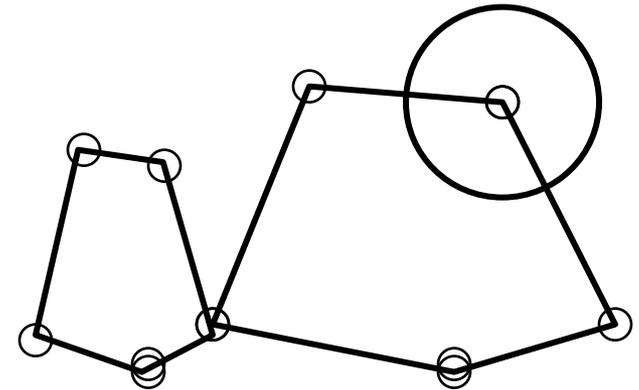
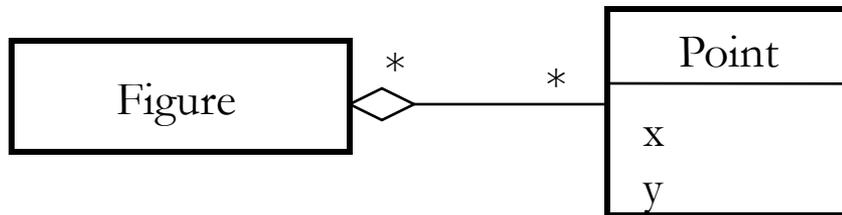


**Contrainte :**  
le graphe d'objets  
forme un arbre  
(ou une forêt)

# Agrégation



- agrégation =
  - cas particulier d'association
  - + contraintes décrivant la notion d'appartenance... (???)



Partage possible

Pas de consensus sur la signification exacte de l'agrégation

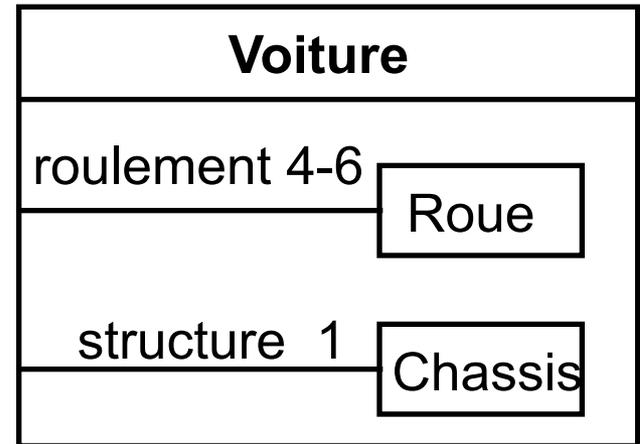
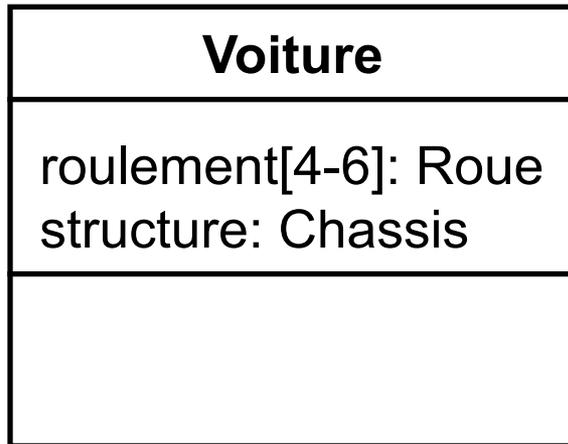
Utiliser avec précautions (ou ne pas utiliser...)

Supprimé en UML2.0

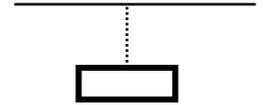
# Autre vues de la composition/agrégation

---

- Différentes formes suggérant l'*inclusion* :

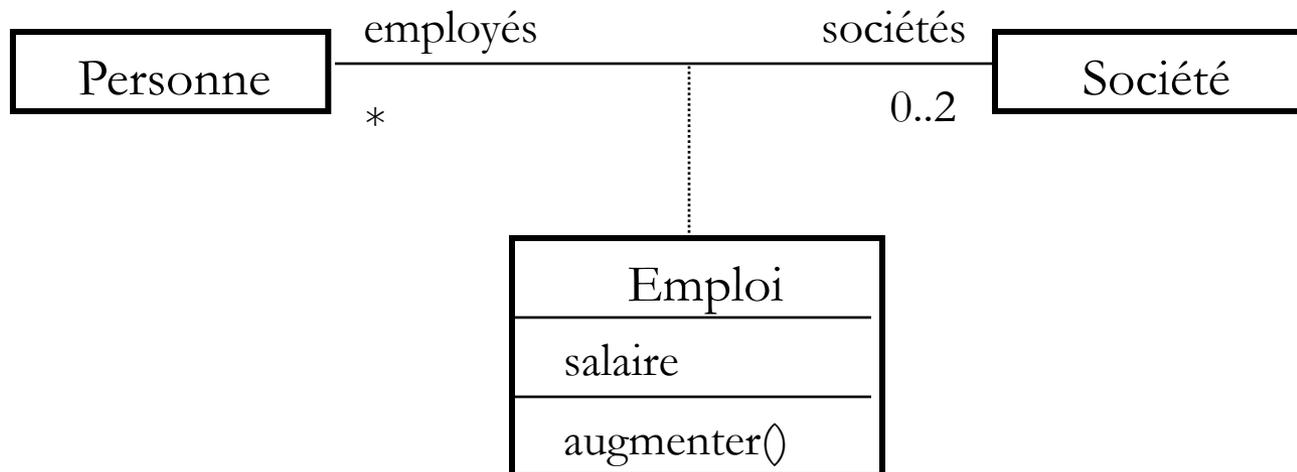


# Classes association



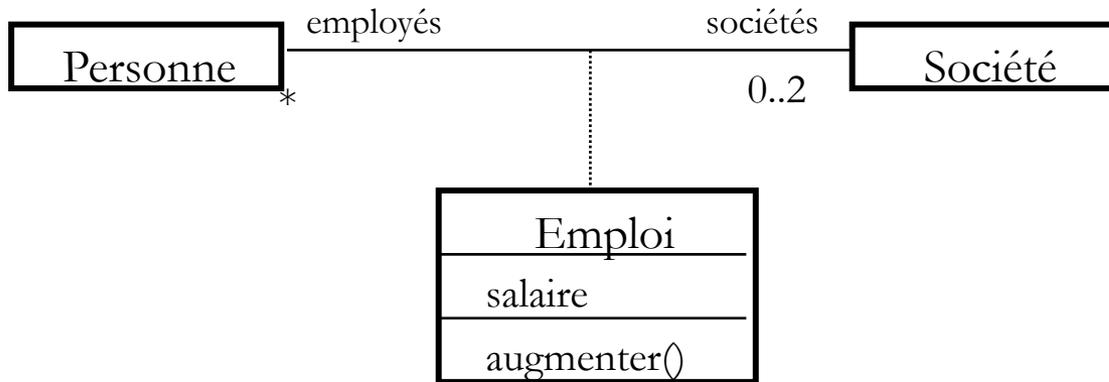
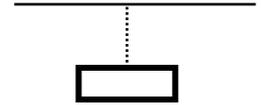
- Pour associer des attributs et/ou des méthodes aux associations

⇒ classes associations



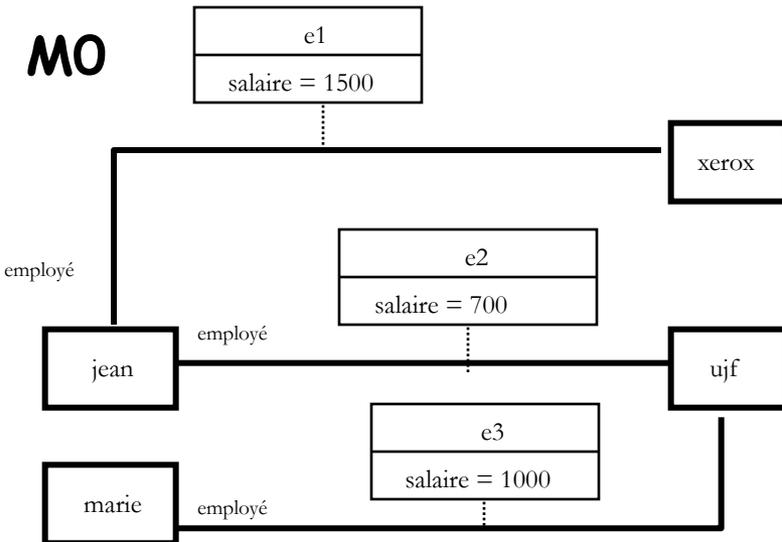
Le nom de la classe correspond au nom de l'association  
(problème: il faut choisir entre forme nominale et forme verbale)

# Classes association



M1

M0

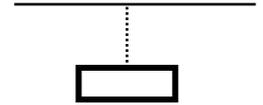


Le salaire est une information correspondant

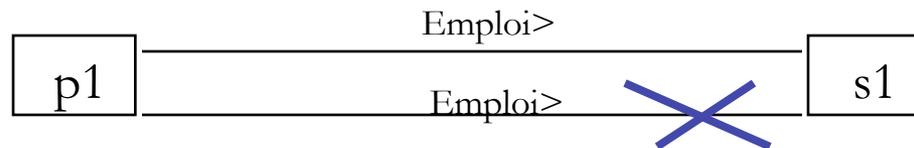
- ni à une personne,
- ni à une société,

mais à un emploi (un couple personne-société).

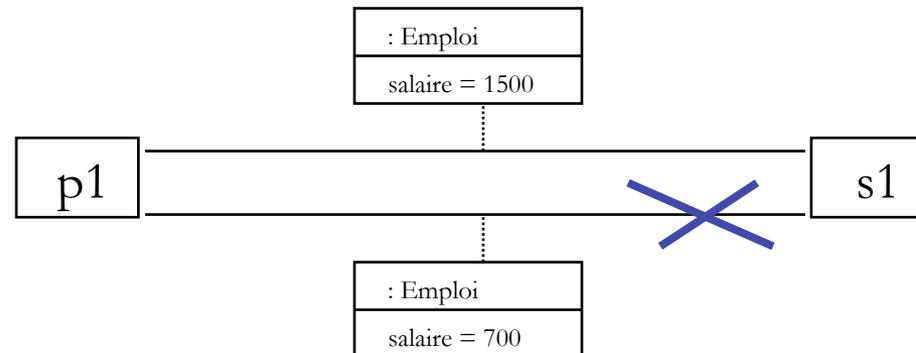
# Classes association



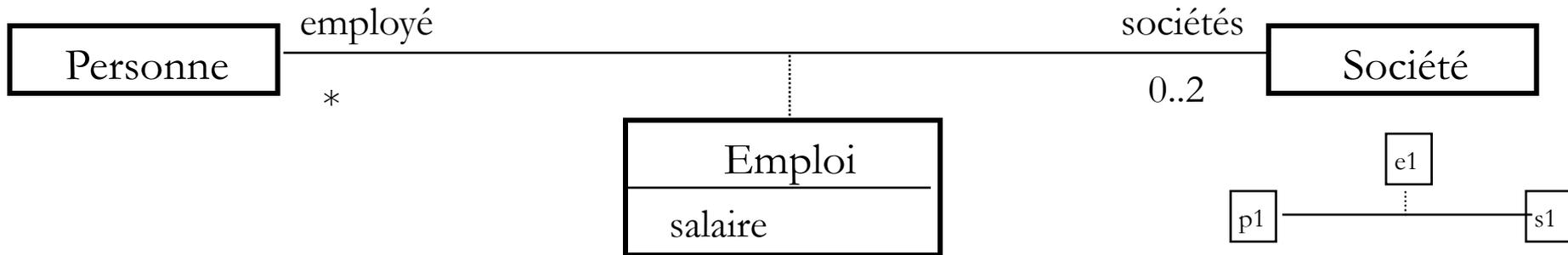
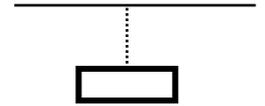
- **RAPPEL:** pour une association donnée, un couple d'objets ne peut être connectés que par un seul lien correspondant à cette association (sauf si l'association est décorée par {nonunique} en UML2.0)



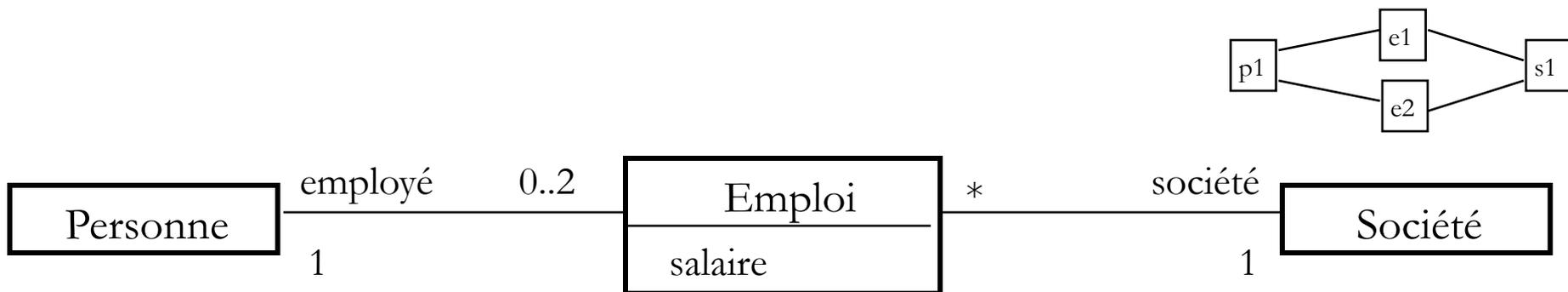
- Cette contrainte reste vraie dans le cas où l'association est décrite à partir d'une classe association



# Classes association

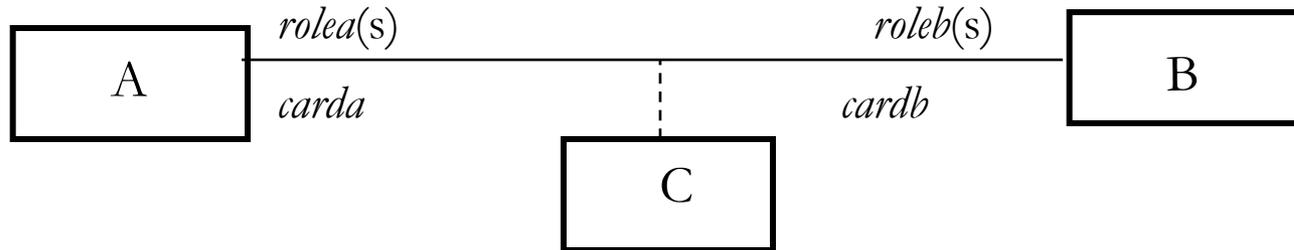
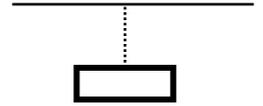


Ci-dessus, une personne peut avoir deux emplois, mais pas dans la même société

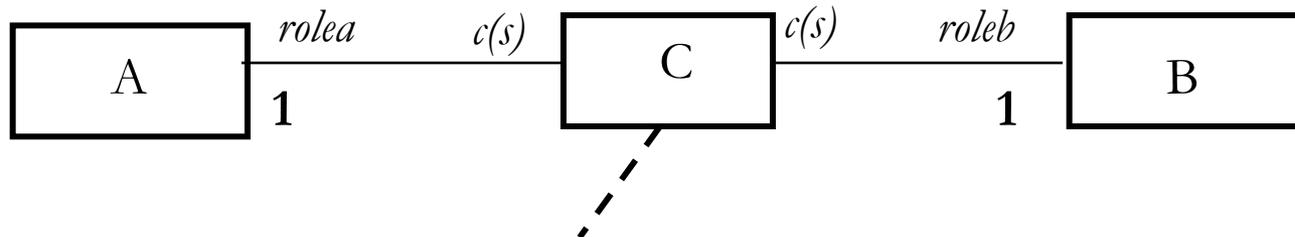


Ci-dessus, une personne peut avoir deux emplois dans la même société

# Classes association (sémantique)

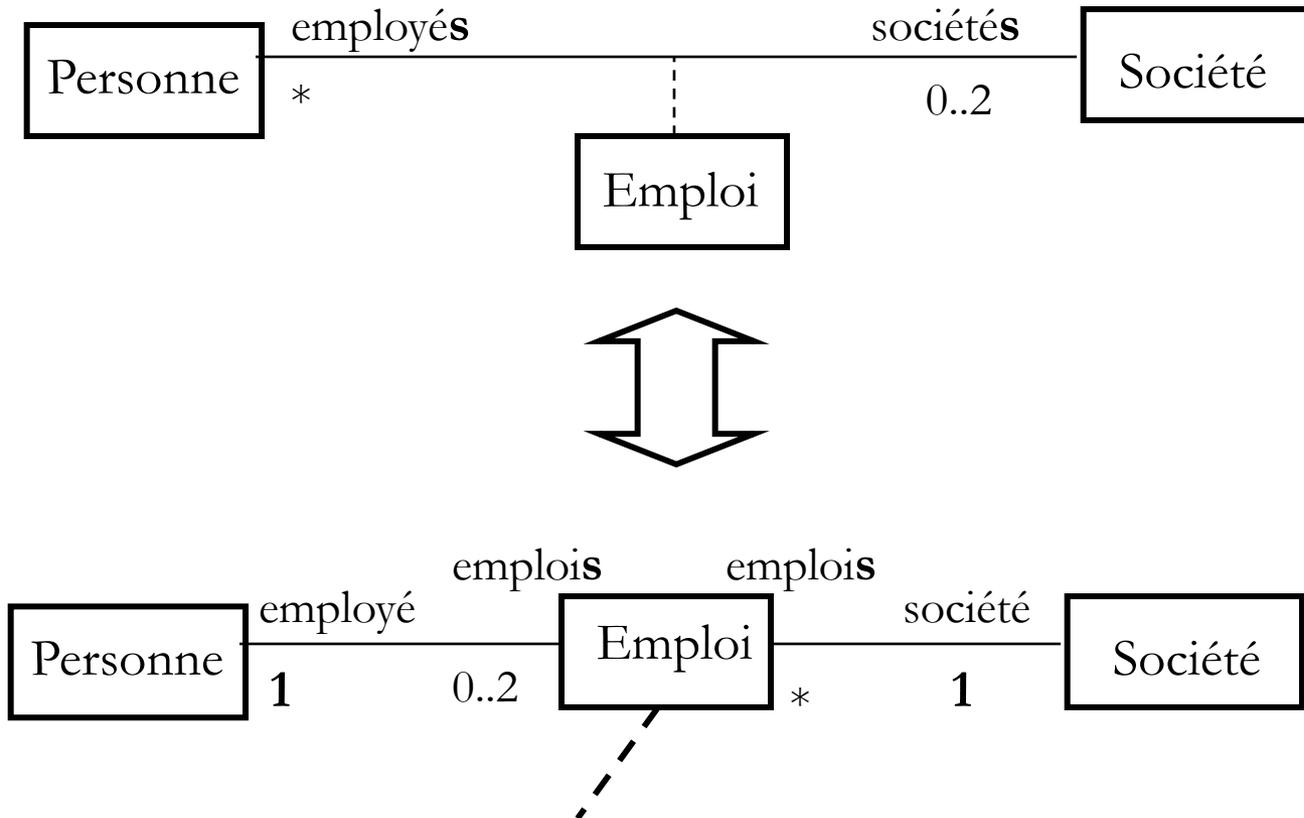
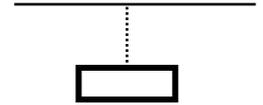


Transformation  
systématique  
pour revenir aux  
concepts de base



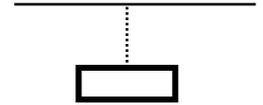
Il ne peut y avoir qu'un objet C entre un objet A et un objet B donné

# Classes association (sémantique)

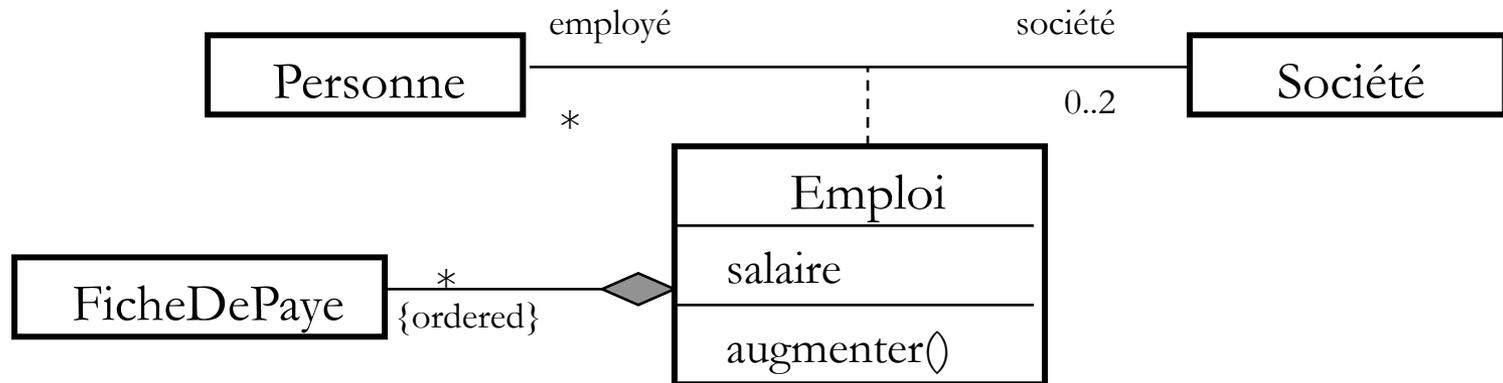


Il ne peut y avoir qu'un **Emploi** entre une **Personne** et une **Société**

# Classes association

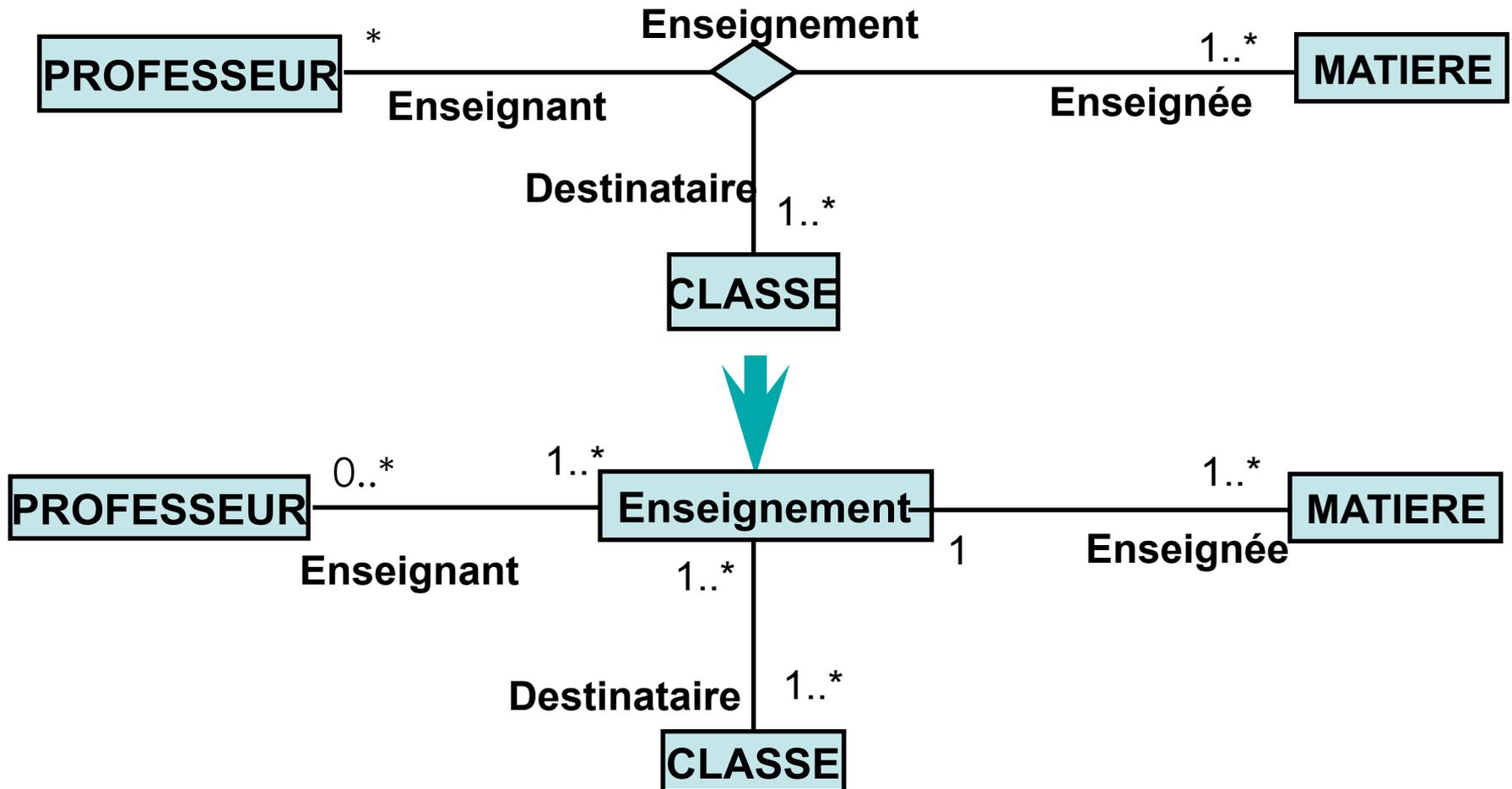


- Les classes association sont des associations mais aussi des classes.
- Elles ont donc les mêmes propriétés et peuvent par exemple être liées par des associations.



# Associations n-aires

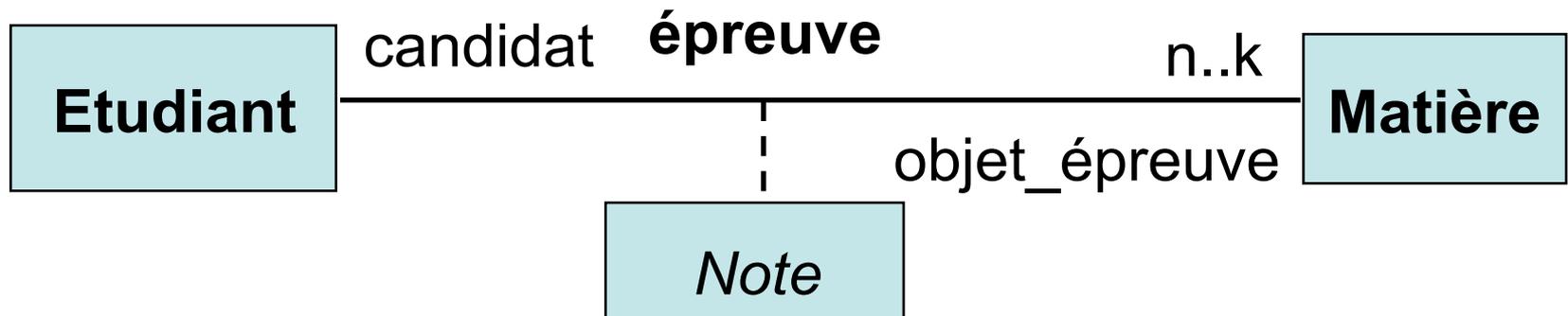
- Relations entre plus de 2 classes (*à éviter si possible*) :



# Associations attribuées

---

- L'attribut porte sur le lien



# Associations qualifiées

---

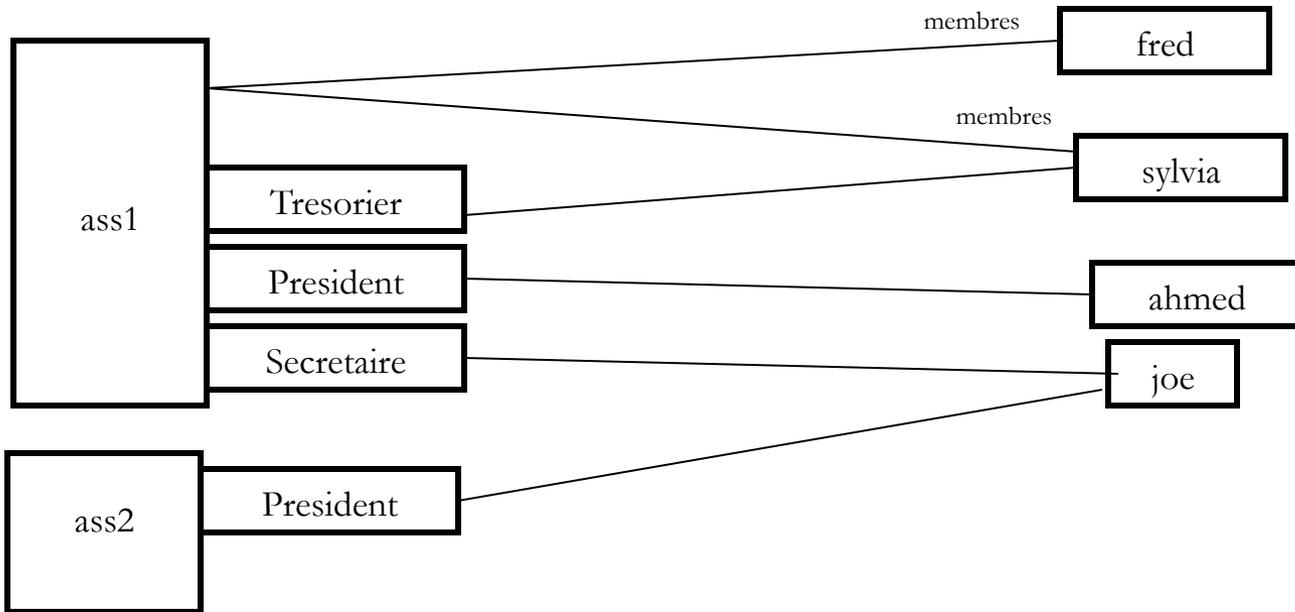
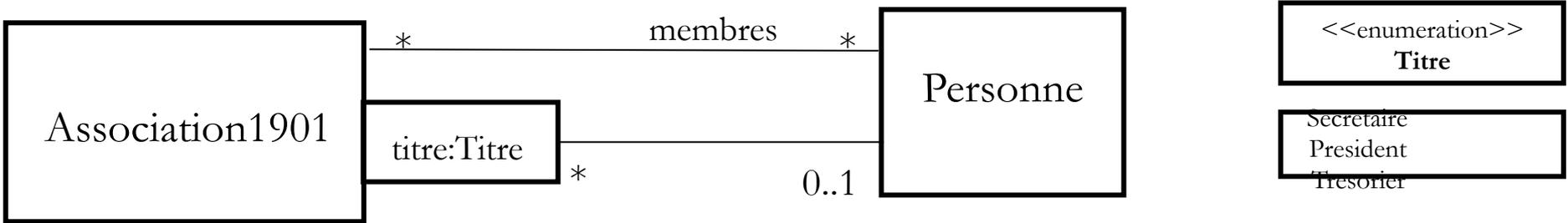
- Un qualificateur est un attribut (ou un ensemble d'attributs) dont la valeur sert à déterminer l'ensemble des instances associées à une instance via une association.



- "Pour un répertoire, à un nom donné on associe qu'un fichier (ou 0 s'il existe aucun fichier de ce nom dans ce répertoire)."
  - Correspond à la notion intuitive d'index absente ci-dessous



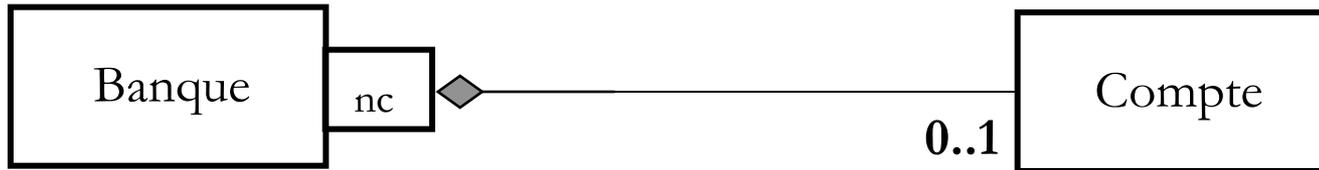
# Associations qualifiées - exemple



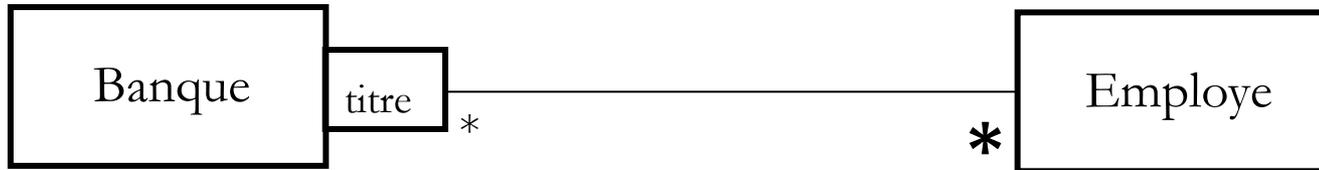
# Cardinalité des associations qualifiées

---

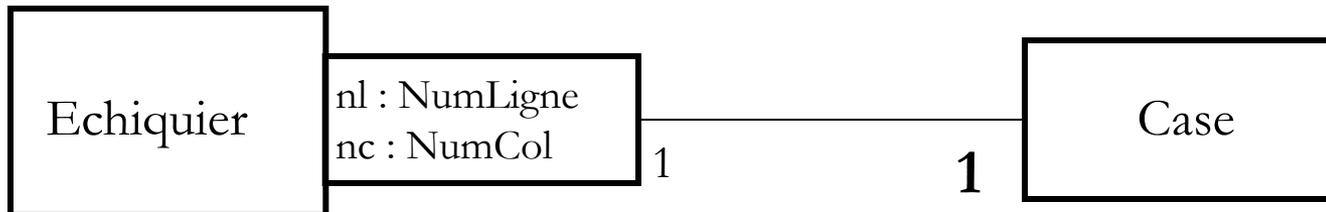
**Cas classique:** cardinalité 0..1



**Cas plus rare:** cardinalité \* (pas de contrainte particulière exprimée)



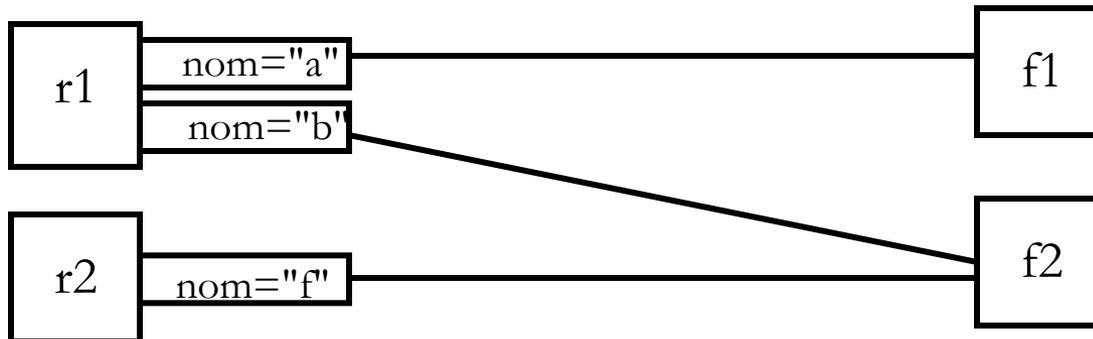
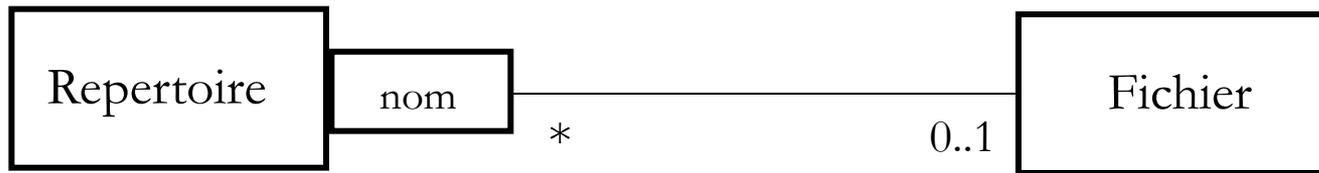
**Cas plus rare:** cardinalité 1 (généralement c'est une erreur)



0 comme cardinalité minimale, sauf si le domaine de l'attribut qualifieur est fini et toutes les valeurs ont une image.

# Attributs de l'association

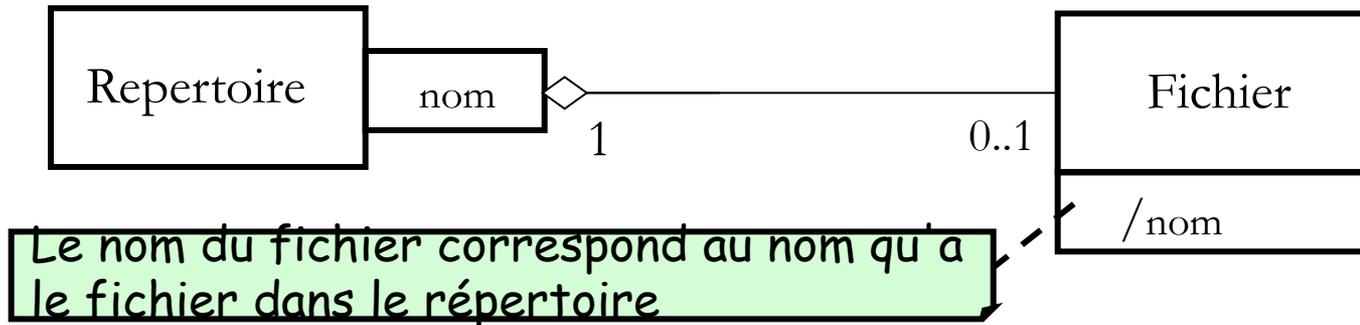
Les attributs qualifieurs sont des **attributs de l'association**, pas de la classe



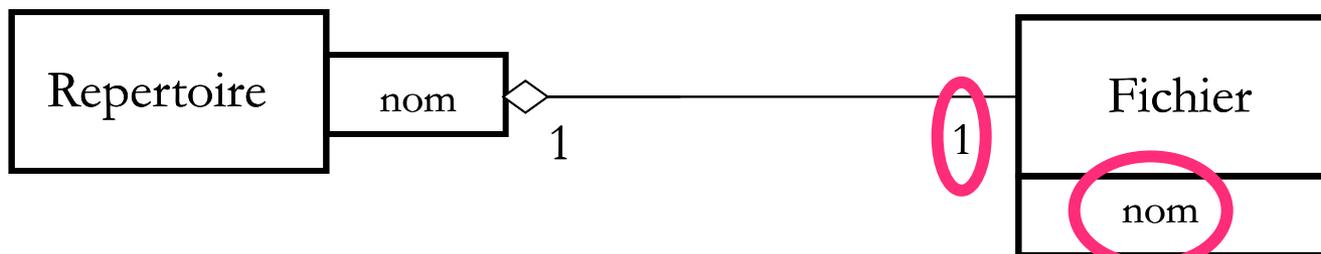
Exemple liens "hard" en Unix: un fichier peut correspondre à des noms différents dans des répertoires différents

# Problèmes classiques

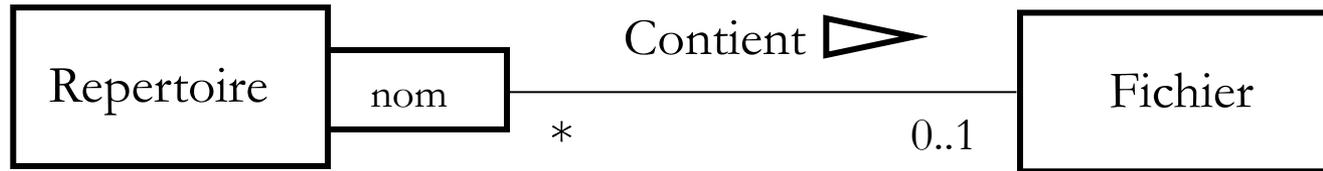
- Souvent l'index est également un attribut de classe indexée
- Solution correcte:



**2 erreurs communes:**



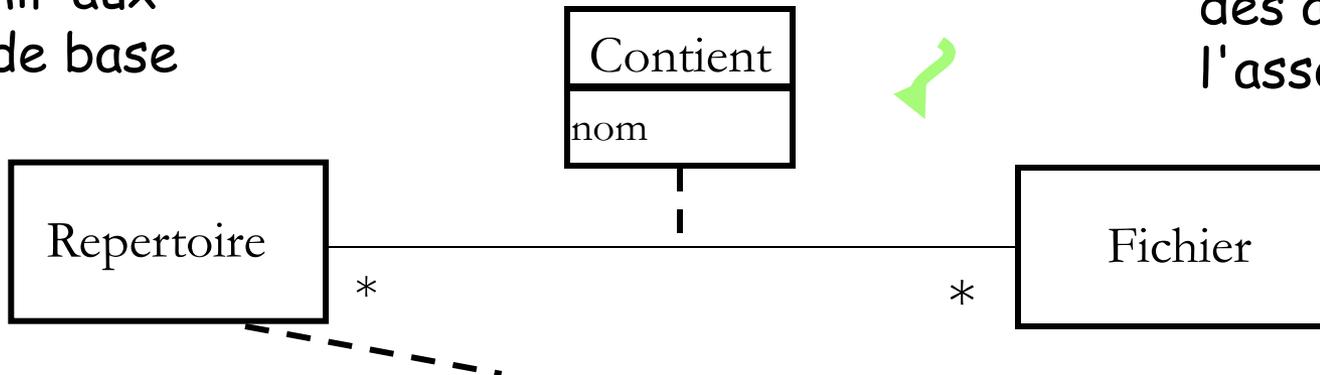
# Classes qualifiée (sémantique)



Transformation  
systématique  
pour revenir aux  
concepts de base



Les attributs du  
qualifieur sont  
des attributs de  
l'association

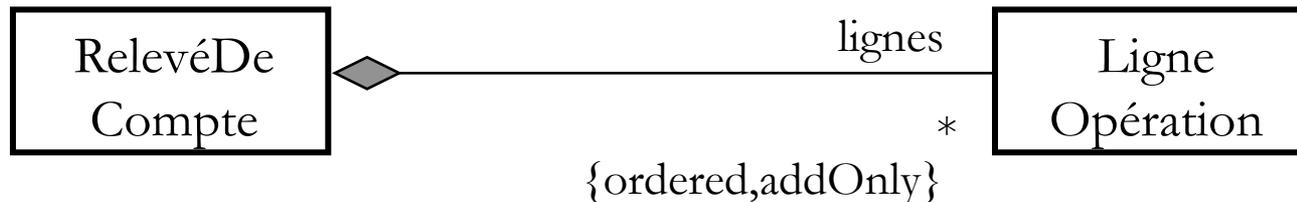


Un répertoire contient 0 ou 1  
fichier pour un nom donné

# Contraintes prédéfinies sur les associations

---

- Par exemple :
  - $\{ordered\}$ : les éléments de la collection sont ordonnés
  - $\{nonUnique\}$  : répétitions possibles (UML2.0)
  - $\{frozen\}$ : fixé lors de la création de l'objet, ne peut pas changer
  - $\{addOnly\}$ : impossible de supprimer un élément



- Il est possible de définir de nouvelles contraintes

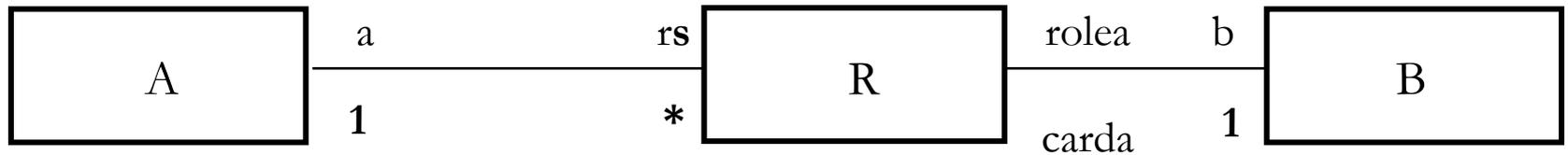
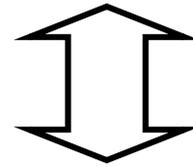
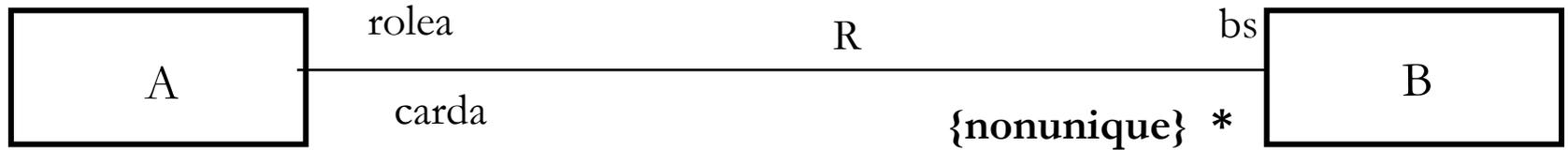
# Contraintes prédéfinies sur les rôles

---

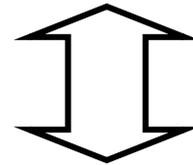
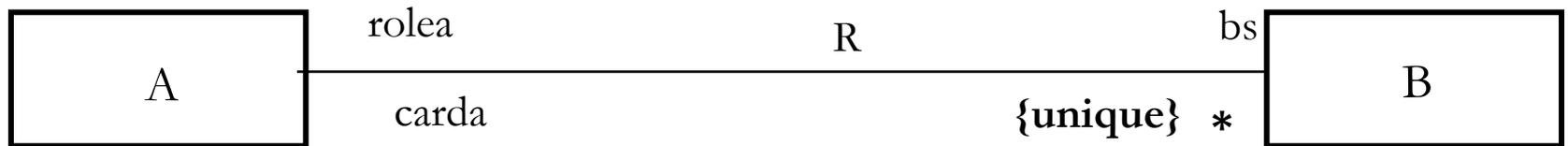
- **{redefines <end-name>}** : redéfinition d'un autre rôle
- **{subsets <property-name>}** : le rôle représente un sous-ensemble
- **{union}** : union dérivée de tous ses sous-ensembles

# Rôle non unique (sémantique)

---

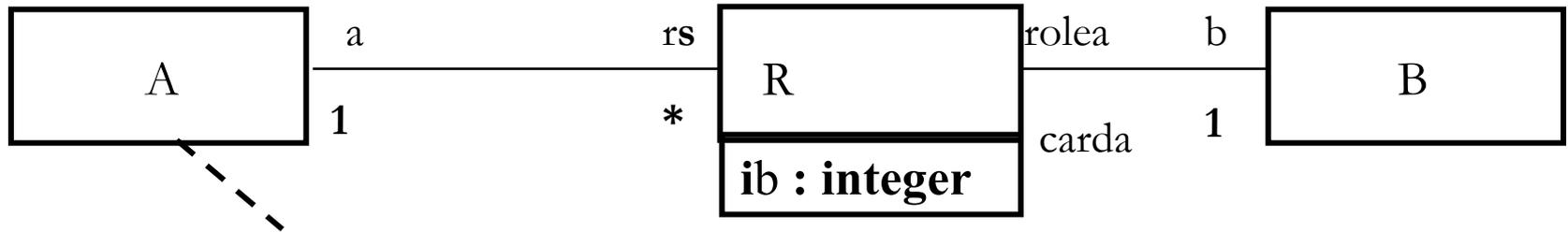
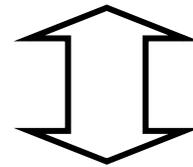
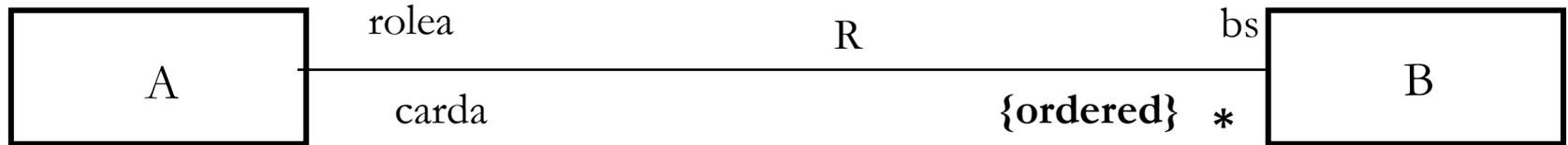


# Rôle unique (sémantique)



Entre un  $a$  et un  $b$  donné il n'y a qu'un  $r$

# Rôle ordonné (sémantique *par index*)



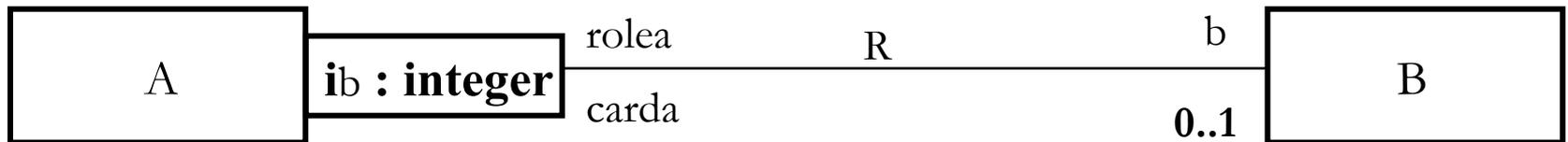
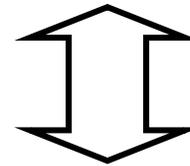
Pour tout  $a$ , tous les  $rs$  ont un indice  $ib$  différent et couvrant l'intervalle de 1 au nombre de  $bs$ .

(si {unique} on a aussi  
Entre un  $a$  et un  $b$  il n'y a qu'un  $r$ )

$rs \rightarrow isUnique(ib)$  and  
 $rs.i \rightarrow asSet = Sequence\{1..bs \rightarrow size()\}$

-- si {unique}  
 $rs \rightarrow isUnique(b)$

# Rôle ordonné (sémantique *par index*)

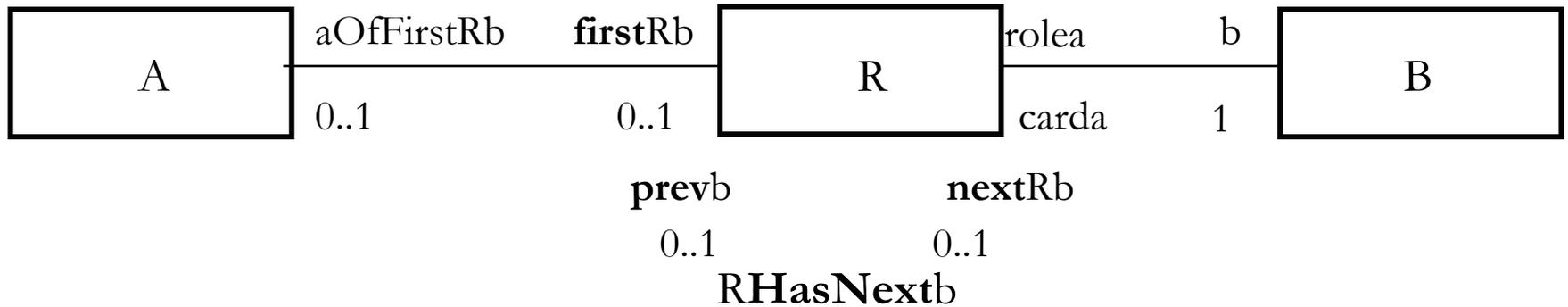
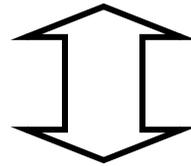
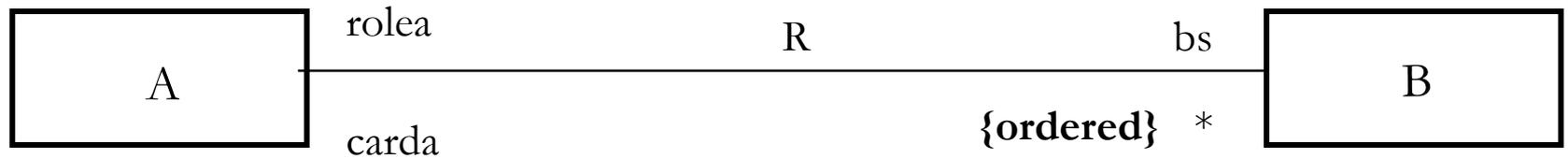


`R->isUnique(ib)`

and

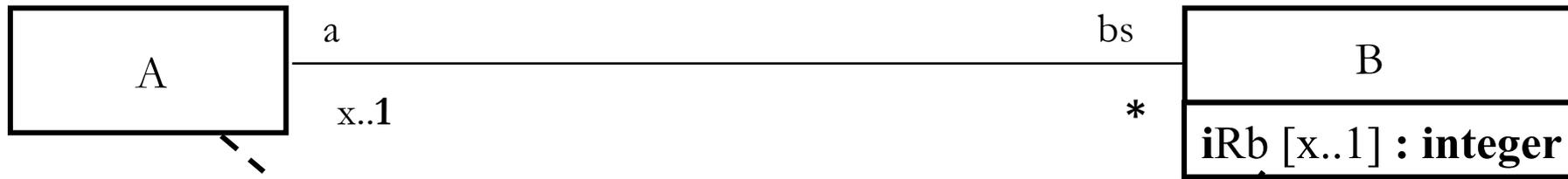
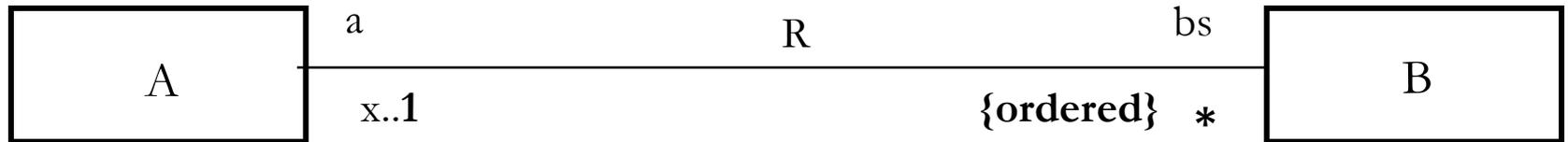
`R.ib->asSet = R.Set{1..b->size}`

# Rôle ordonné (sémantique *par chaînage*)



`aOfFirstRb->isEmpty xor prevb->isEmpty`  
`allNextRb()->excludes(self)`  
 avec `allNextRb() : Set(R) = nextRb.allNextRb()->including(nextRb)`

# Rôle ordonné 1-\* (sémantique *par index*)

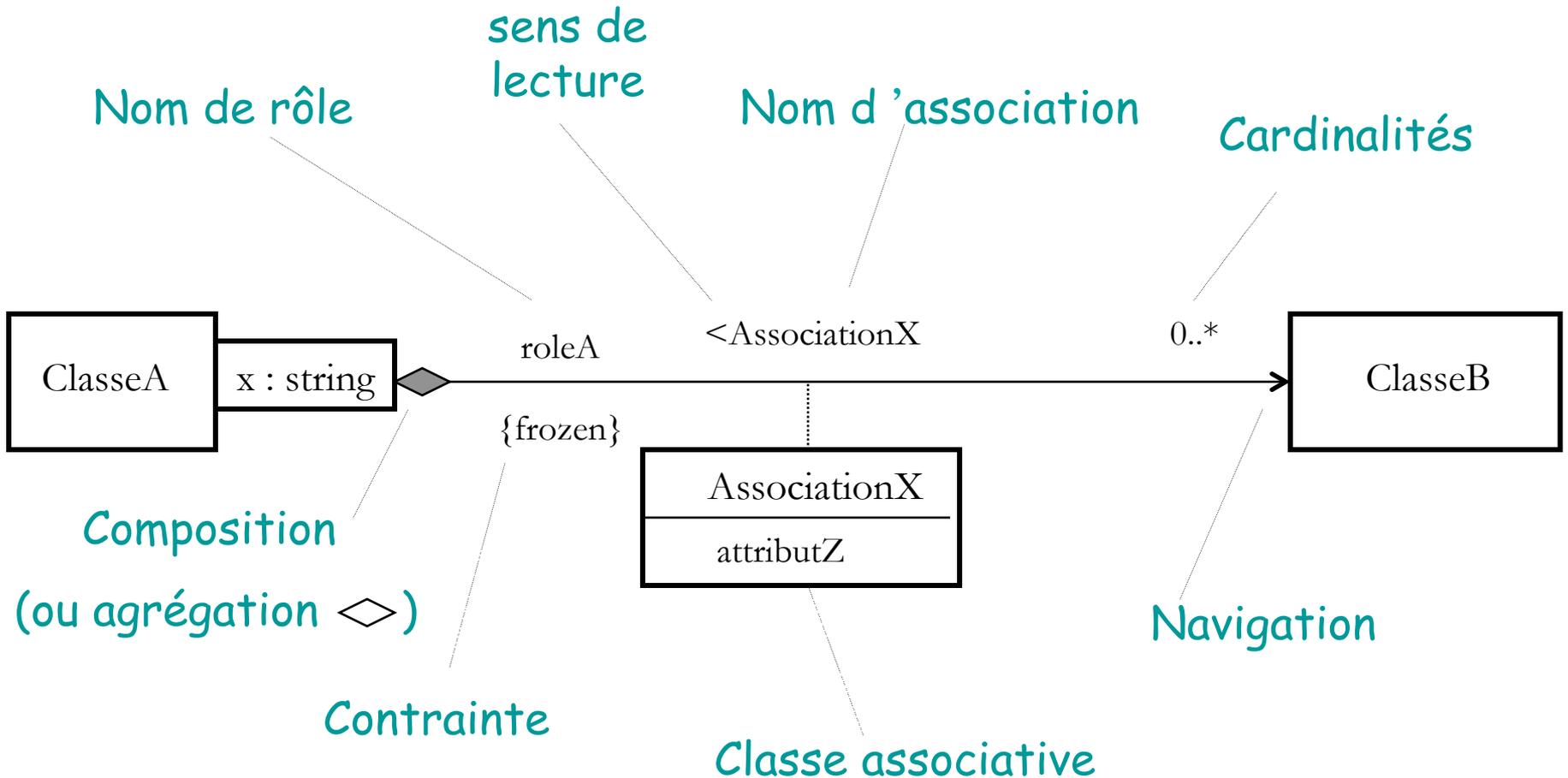


Pour tout  $a$ , tous les  $bs$  ont un indice  $iRb$  différent et couvrant l'intervalle de 1 au nombre de  $bs$ .

$iRb \rightarrow isEmpty = a \rightarrow isEmpty$

$bs \rightarrow isUnique(iRb)$  and  
 $rs.i \rightarrow asSet = Sequence\{1..bs \rightarrow size()\}$

# Synthèse sur les associations



# C. Raffinement du concept de GÉNÉRALISATION

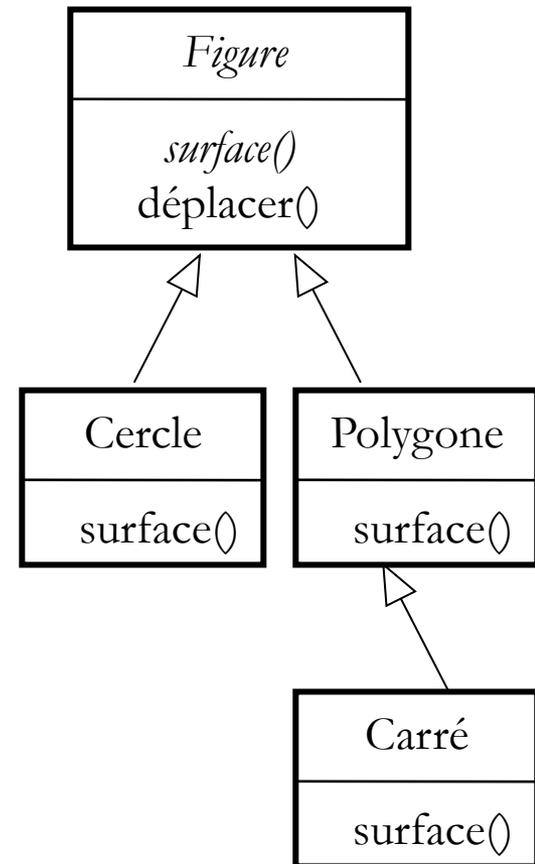
---

- Re-définition
- Classes abstraites
- Méthodes abstraites
- Héritage simple vs. Multiple
- Classification simple vs. Multiple
- Classification statique vs. dynamique

# Héritage et redéfinition

---

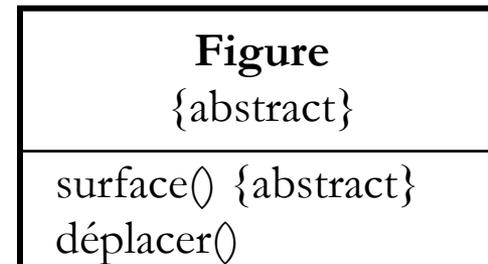
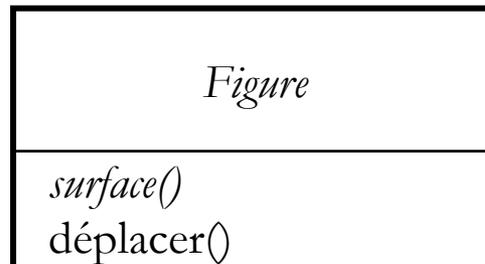
- Une sous classe peut redéfinir une méthode...
- ... à condition toutefois de rester compatible avec la définition originale



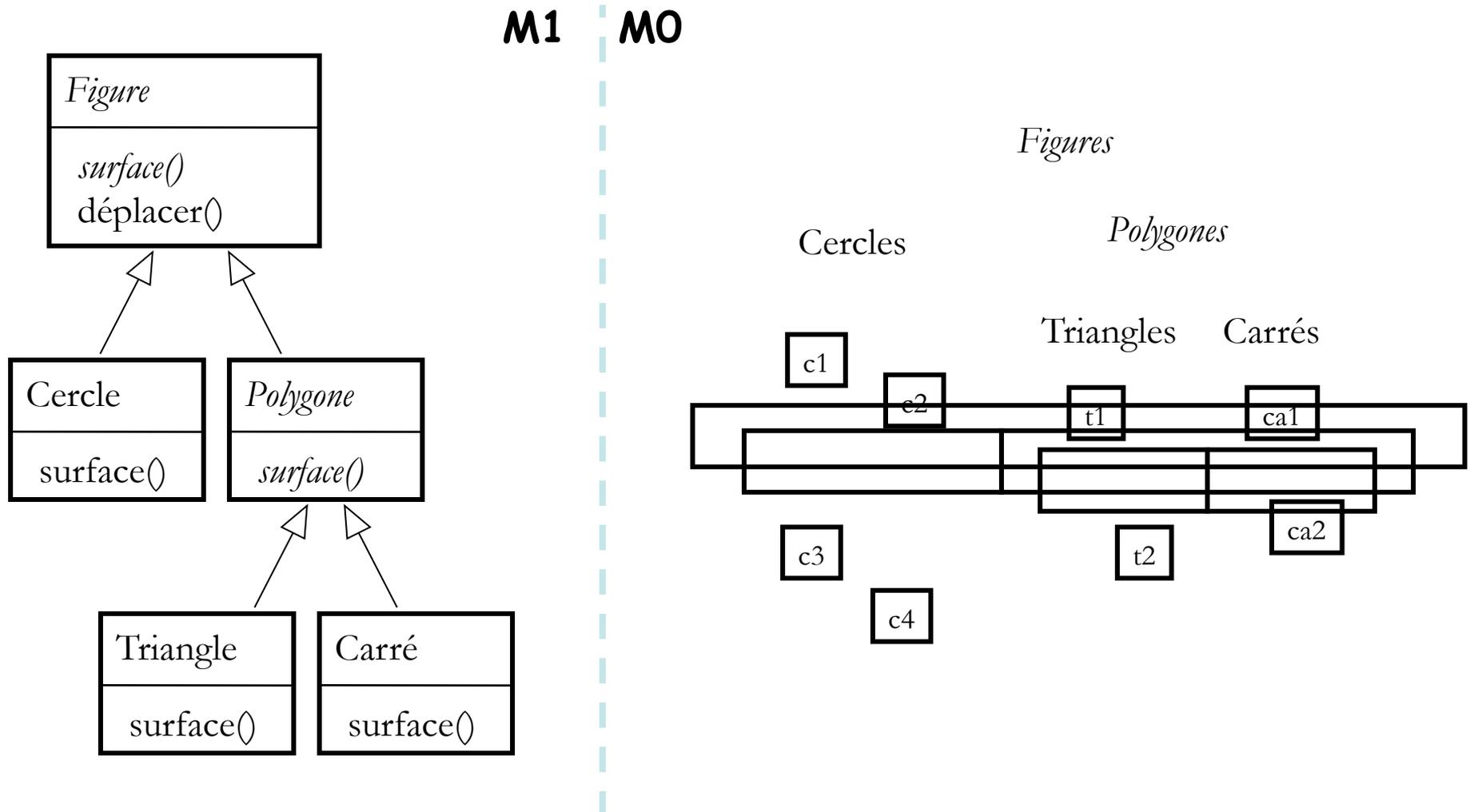
# Classes et méthodes abstraites

---

- Une classe abstraite
  - ne peut pas être instanciée
  - utile pour définir un comportement abstrait
  - peut contenir des méthodes abstraites
- Une méthode abstraite
  - doit être définie dans une sous classe
  - est dans un classe abstraite
- Notations équivalentes :



# Classes abstraites du point de vue ensembliste



# Classification *vs.* Héritage

---

Classification

Chien



`<<instanceOf>>`

medor

Héritage/Spécialisation

Chien



Animal

medor

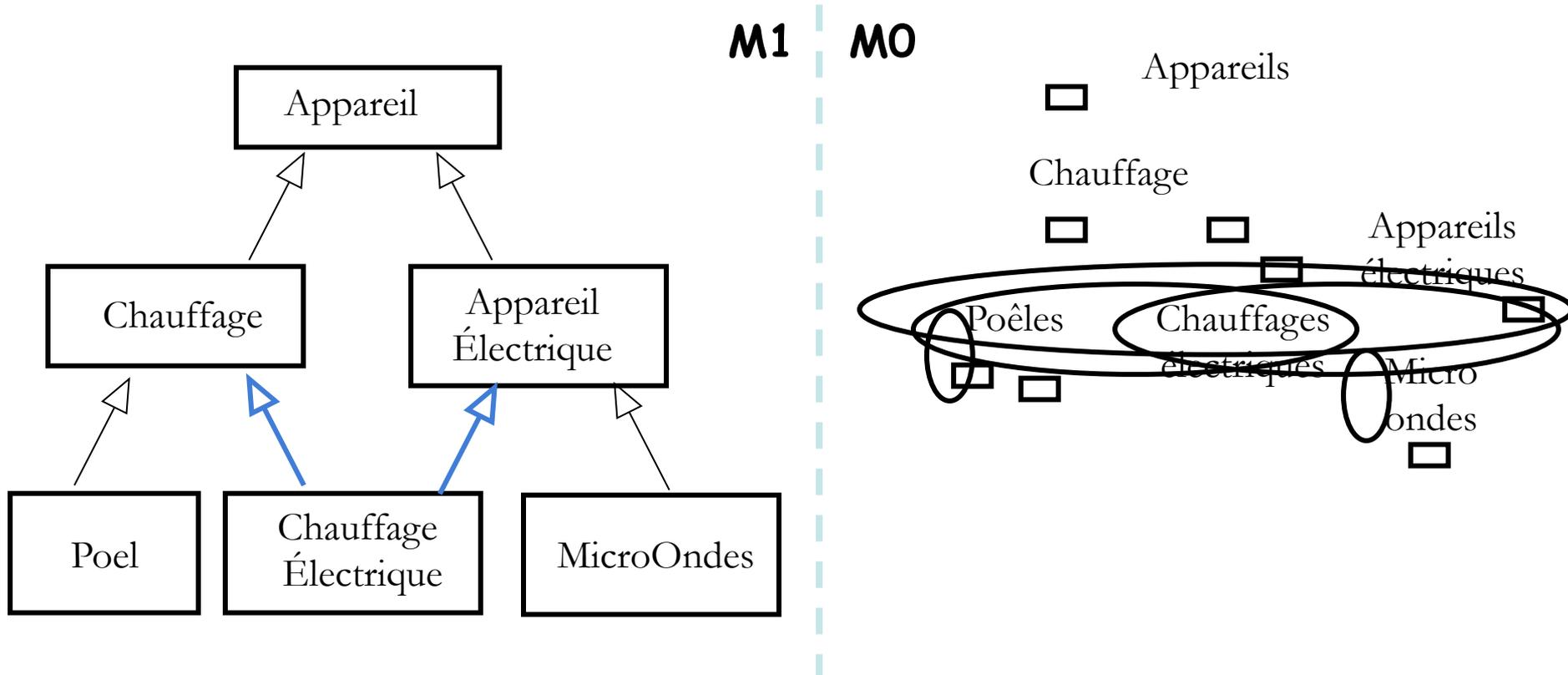
coco

M1

M0

# Héritage multiple

- Une classe peut hériter de plusieurs super-classes



- Interdit dans certains langages de programmation (e.g., Java et C#)

# Points liés à l'héritage et à la classification

---

- Les modèles orientés-objets ne font pas tous les mêmes hypothèses
  - Héritage simple *vs.* héritage multiple
    - Une classe peut elle hériter de plusieurs classes ?
  - Classification simple *vs.* classification multiple
    - Un objet peut-il être simultanément instance de plusieurs classes?
  - Classification statique *vs.* classification dynamique
    - Un objet peut-il changer de classe pendant l'exécution ?

# Hypothèses UML par défaut

---

- Sauf si le contraire est indiqué explicitement, en UML les hypothèses par défaut sont :
  - Héritage multiple
    - une classe peut hériter de plusieurs classes
  - Classification simple
    - un objet est instance d'une seule classe
  - Classification statique
    - un objet est créé à partir d'une classe donnée et n'en change pas

Deuxième partie  
(toujours UML)

# Où en est on ? (Récapitulatif)

---

- From Programming to Modeling
- Functional view with UML (use case diagram & model, system sequence diagram)
- Structural view with UML:
  - Class diagram (M1 = type level) *vs.* Object Diagram (M2 = instance level)
    - classifiers , associations, generalization
    - objects, links, polymorphism
- Now:
  - advanced UML (packages, OCL, etc.)
  - behavioral and implementation views with UML

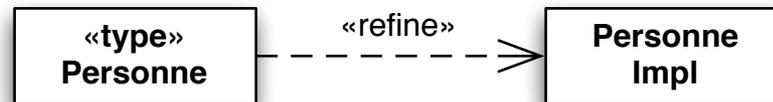
# D. Concept de DÉPENDANCE (1/3)

---

- Relation entre deux éléments (client et fournisseur)

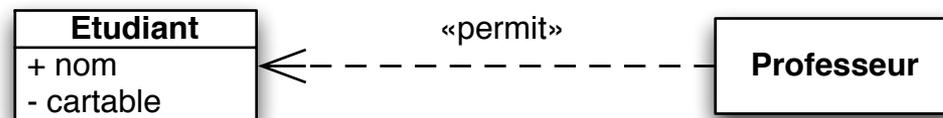
- Abstraction** : Entre éléments de différents niveaux sémantiques

- «derive» : éléments dérivés, pas forcément du même type
- «refine» : raffinement entre modèles



- «trace» : (aussi) - utilisé pour marquer les changements entre modèles
- Intérêt : Outillage de méthodes de développement (RUP, Catalysis, etc.)

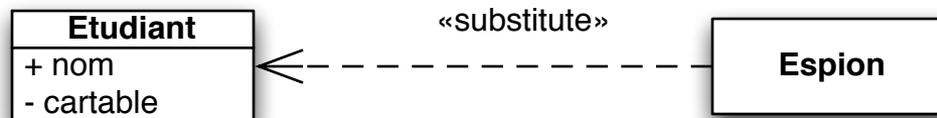
- Permission** (« permit ») : droits d'accès aux propriétés privées



# Concept de dépendance (2/3)

---

- Relation entre deux éléments (client et fournisseur)
  - **Réalisation** (« realize ») :
    - Relation entre deux ensembles d'éléments, de spécification et d'implémentation
    - Utilisé pour représenter: raffinement, optimisation, transformations, synthèses, etc.
  - **Substitution** (« substitute ») : Relation entre classificateurs. Les instances du fournisseur peuvent remplacer celles du client



# Concept de dépendance (3/3)

---

- Relation entre deux éléments (client et fournisseur)
  - **Usage** : Représentation des associations non permanentes. Très utiles pour estimer la testabilité d'un modèle ou l'impact d'un changement
    - «call» : dépendance entre opérations (où classes)
    - «create» : le client créé des instances du fournisseur (classificateurs)
    - «instantiate» : les opérateurs du client créent des instances du fournisseur
    - «send» : entre une opération et un signal

# E. Raffinement du concept de CONTRAİNTE

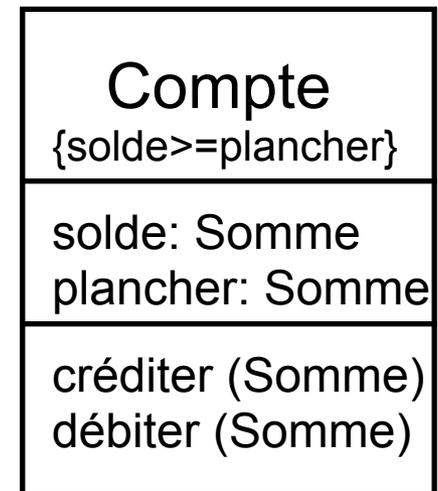
---

- Exemple de contraintes :
  - Invariant (sur une classe)
  - Pre-/Post- condition (sur une méthode)
  
- Remarque : exprimée sur un diagramme de classe, une contrainte réduit le nombre de diagrammes d'objet conformes

# Représentation des invariants

---

- Invariants = Propriétés vraies pour l'ensemble des instances de la classe
  - dans un état stable, chaque instance doit vérifier les invariants de sa classe
  - Des contraintes peuvent être ajoutées aux éléments du modèle UML
    - Notation : entre { }
  - Des contraintes peuvent être exprimées à l'aide d'OCL (Object Constraint Language)
    - e.g. `{solde >= plancher}`



# Représentation des pré/post conditions

---

- Préconditions

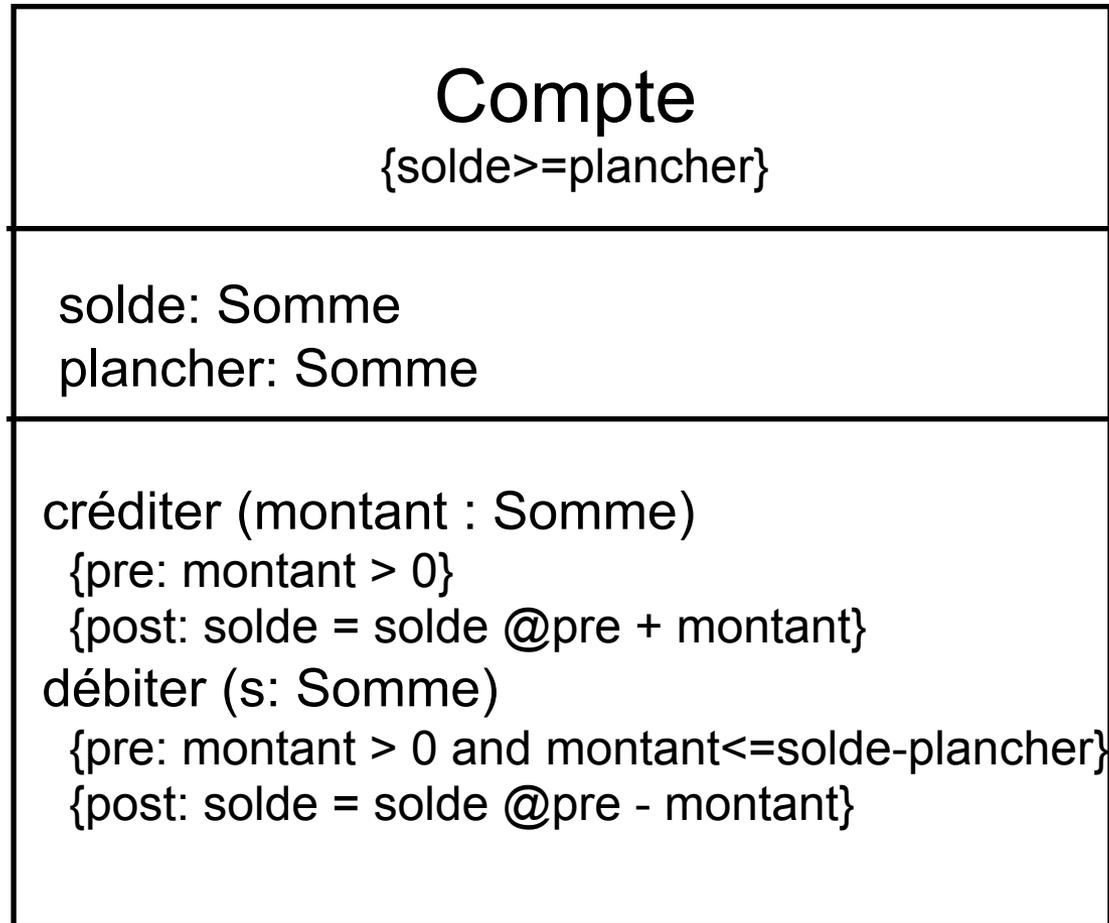
- {«precondition» *expression booléenne* OCL}
- Abrégé en: {pre: *expression booléenne* OCL}

- Postconditions

- {«postcondition» *expression booléenne* OCL}
- Abrégé en: {post: *expression booléenne* OCL}
- Operateur « valeur précédente » (idem *old* Eiffel):
  - *expression* OCL @pre

# Etre abstrait et précis avec UML

---

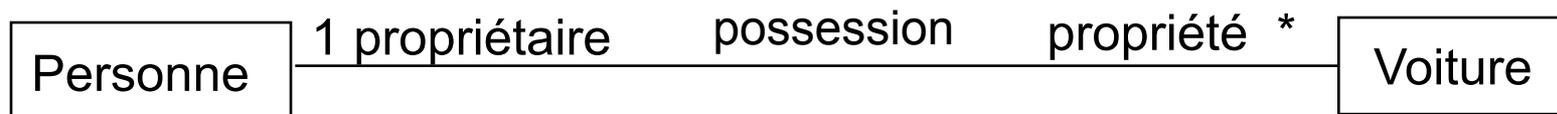


***Analyse précise ou “analyse par contrat”***

# Contraintes OCL navigant les relations

---

- Chaque association est un chemin de navigation
- Le contexte d'une expression OCL est le point de départ (la classe de départ)
- Les noms de rôle sont utilisés pour identifier qu'elle relation on veut naviguer



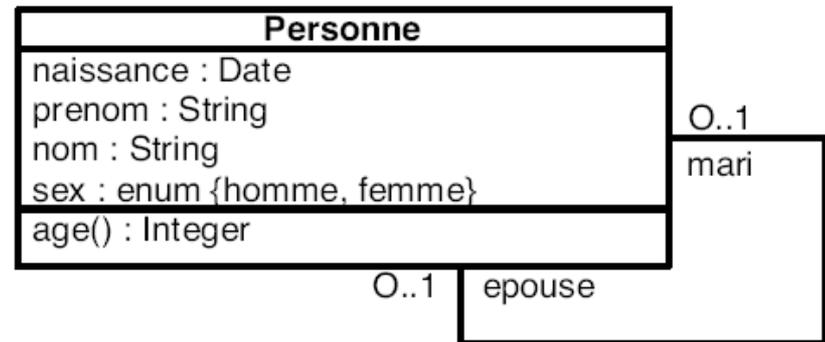
Context Voiture inv:  
self.propriétaire.age >= 18

# Object Constraint Language

OCL

# Motivations d'OCL

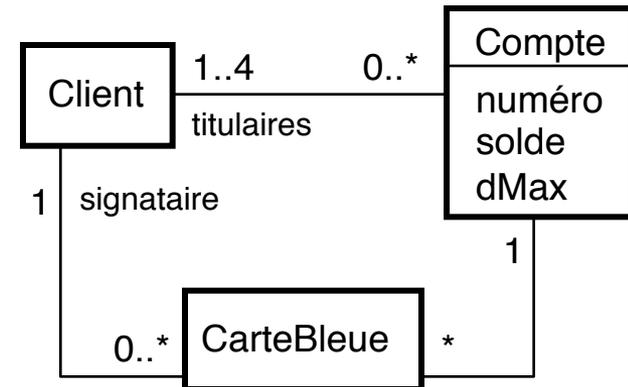
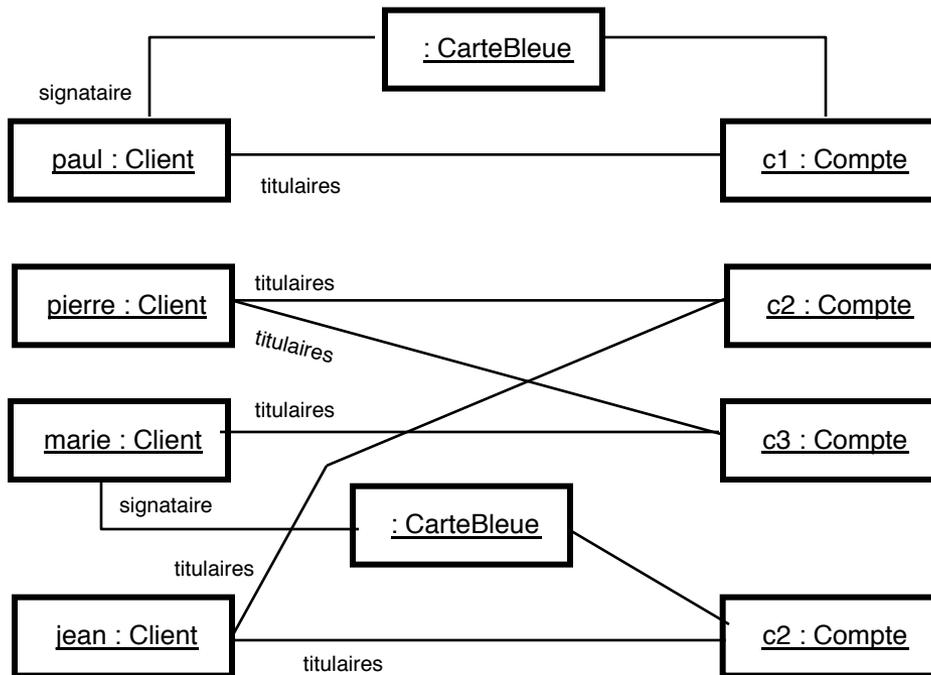
- Diagrammes exprimant certaines contraintes
  - graphiquement
    - contraintes structurelles (e.g. un attribut dans une classe)
    - contraintes de types (e.g. sous-typage)
    - contraintes diverses (e.g. composition, cardinalité, etc.)
  - via des propriétés prédéfinies
    - sur des classes (e.g. {abstract})
    - sur des rôles (e.g. {ordered})
- Les diagrammes UML manquent parfois de **précision**.
- Mais le langage naturel (français ou anglais) est souvent **ambigu**.



L'épouse du mari d'une personne est elle meme. En d'autres termes, le mari d'une femme l'a pour epouse.

# Illusions partagées ... ou non

- Attention aux contraintes supposées être évidentes ou vérifiées "naturellement"



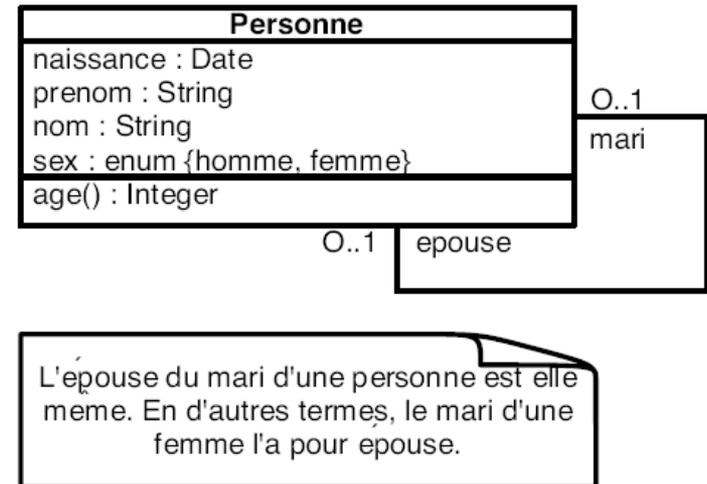
Le signataire d'une carte bleue est titulaire de ce compte.

# Expression des contraintes en langue naturelle

- Simple à mettre en oeuvre
  - utilisation des notes en UML + texte libre
  - compréhensible par tous
- Indispensable !
  - documenter les contraintes est essentiel
  - détecter les problèmes le plus tôt possible
- Problèmes
  - ambigu, imprécis
  - difficile d'exprimer clairement des contraintes complexes
  - difficile de lier le texte aux éléments du modèle

# Conception par contrats avec UML

- Ajout de *contraintes* à des éléments de modélisation.

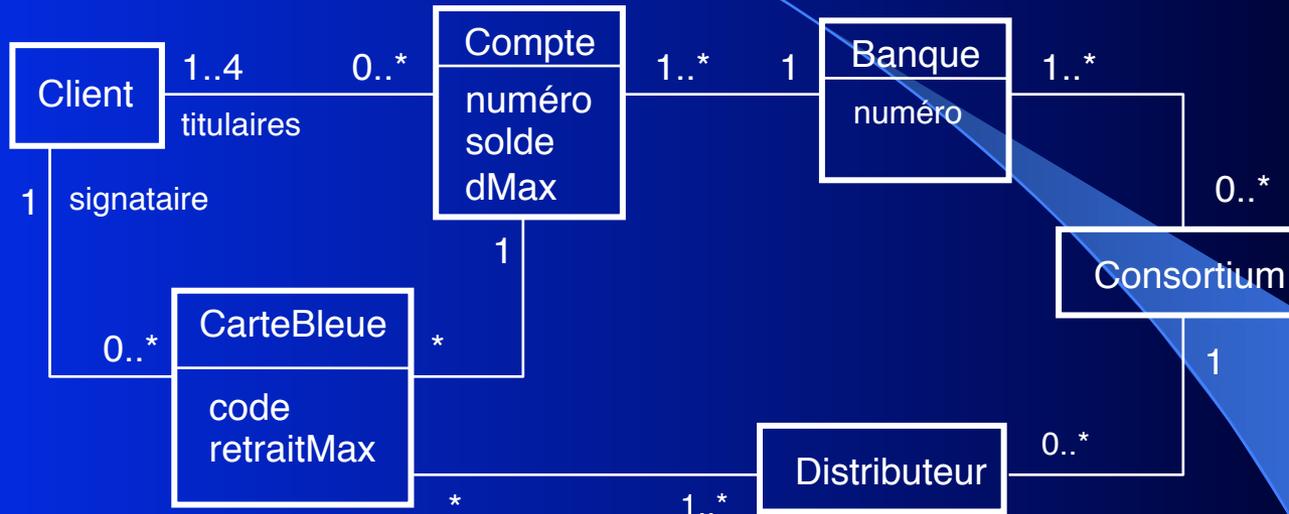


**{context** **Personne** **inv** :

`self.epouse ->notEmpty() implies self .epouse.mari = self`

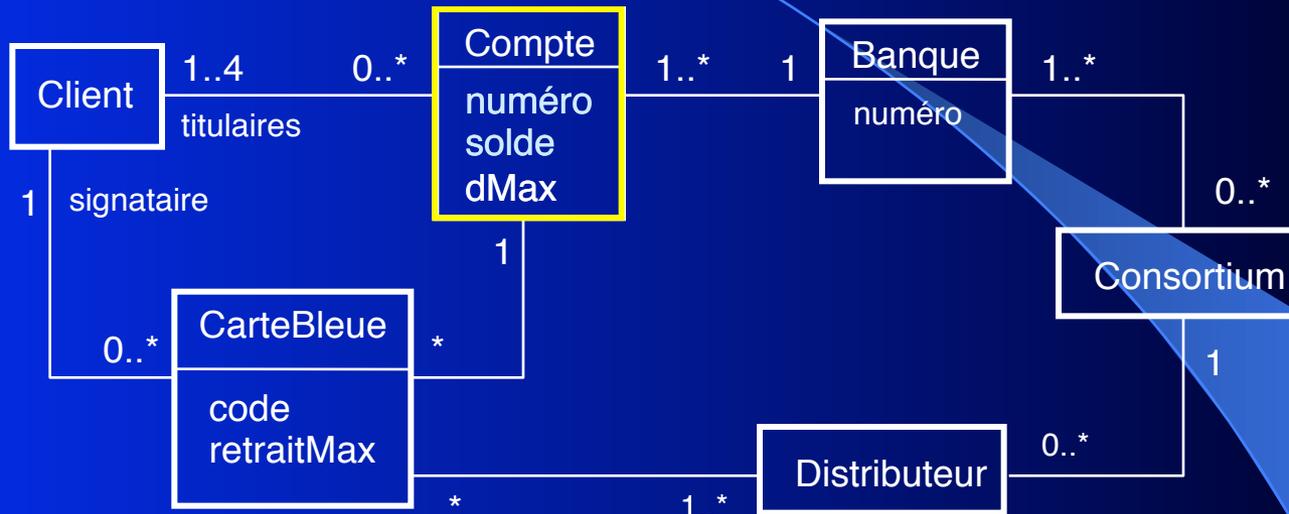
**and** `self.mari -> notEmpty() implies self .mari.epouse = self`}

# Exemple



- (1) Le solde d'un compte ne doit pas être inférieur au découvert maximum autorisé
- (2) Le signataire d'une carte bleue associée à un compte est le titulaire de ce compte.
- (3) Une carte bleue est acceptée dans tous les distributeurs des consortiums de la banque.
- (4) Les clients d'une banque non affiliée à un consortium ne peuvent pas avoir de carte bleues.

# Exemple



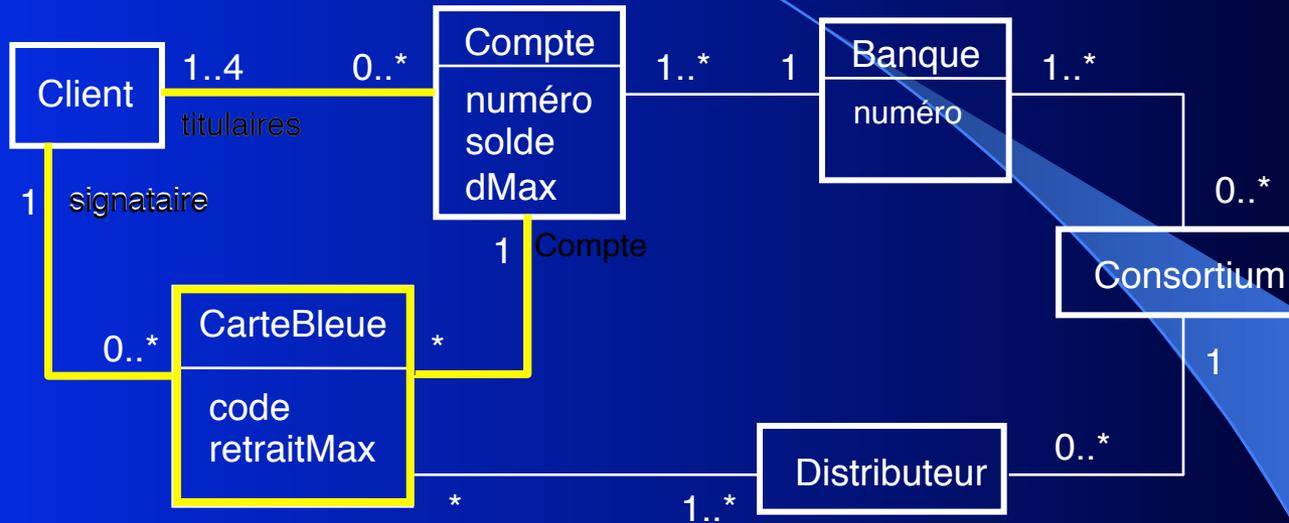
-- (1) Le solde d'un compte ne doit pas être inférieur au découvert maximum autorisé

*context* Compte

*inv:*  $dMax \geq 0$

*inv:*  $solde > dMax$

# Exemple

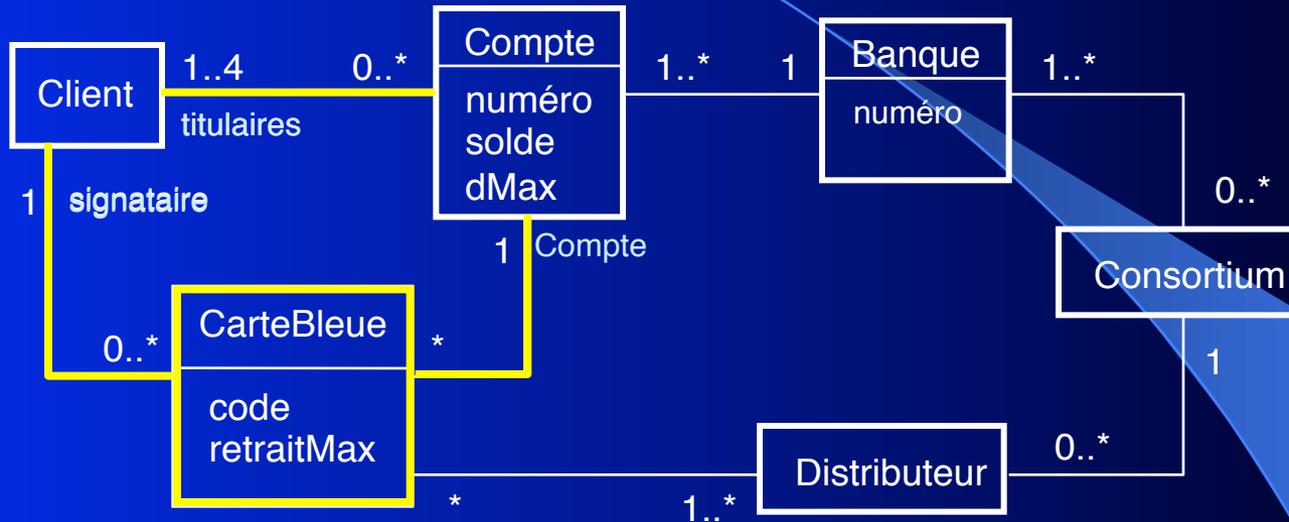


-- (2) Le signataire d'une carte bleue associée à un compte ~~est~~ le titulaire de ce compte.

**context** CarteBleue

**inv:** ~~self.signataire = self.Compte .titulaires~~

# Exemple

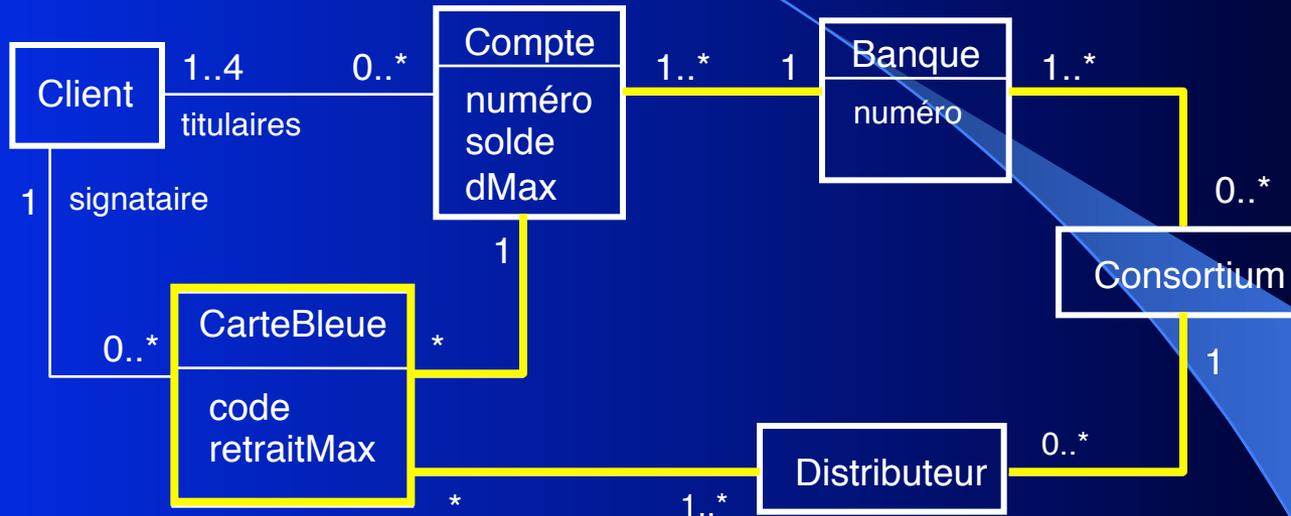


-- (2) Le signataire d'une carte bleue associée à un compte est **l'un des** titulaires de ce compte.

**context** CarteBleue

**inv:** *self.Compte.titulaires* -> **includes**(*self.signataire*)

# Exemple

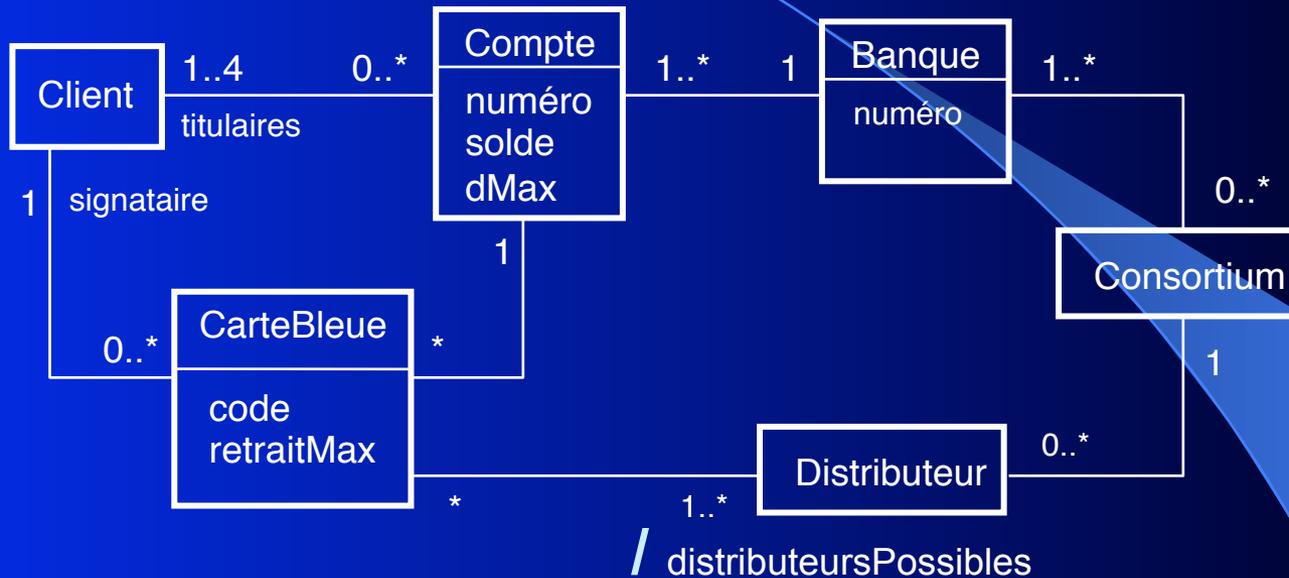


-- (3) Une carte bleue est acceptée dans tous les distributeurs des consortiums de la banque.

**context** CarteBleue

**inv:** *self.Distributeur = self.Compte.Banque.Consortium.Distributeur->asSet*

# Exemple



```
context CarteBleue::distributeursPossibles : set(Distributeur)
derive: self.Compte.Banque.Consortium.Distributeur->asSet
```

# Caractéristiques d'OCL

---

- Langage d'expressions (fonctionnel)
  - Valeurs, expressions, types
  - Fortement typé
  - Pas d'effets de bords
- Langage de spécification, pas de programmation
  - Haut niveau d'abstraction
  - Pas forcément exécutable
  - Permet de trouver des erreurs beaucoup plus tôt dans le cycle de vie

# Caractéristiques d'OCL

---

- Points faibles

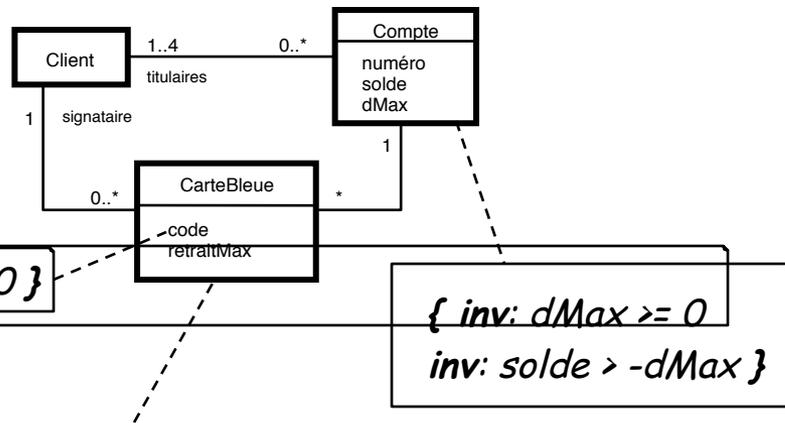
- Pas aussi rigoureux que VDM, Z ou B  $\Rightarrow$  pas de preuves possibles
- Puissance d'expression limitée

- Points forts

- Relativement simple à écrire et à comprendre
- Très bien intégré à UML
- Bon compromis entre simplicité et puissance d'expression
- Passage vers différentes plateformes technologiques

# Contexte d'une contrainte

- Contrainte toujours associée à un élément de modèle :  
le contexte de la contrainte.
- Deux techniques pour spécifier le contexte :



*{ inv: Compte.titulaires->includes(self.signataire)}*

**context** Compte

*inv: dmax >= 0*

*inv: solde > -dMax*

**context** CarteBleue

*inv: Compte.titulaires->includes(self.signataire)*

*inv: code > 0 and code <= 9999*

*inv: retraitMax > 10*

**context** Compte::solde : integer

*init: floor(depotInitial \* 10 / 100)*

# Où utiliser OCL

---

- OCL peut être utilisé pour décrire des prédicats

- *inv*: invariants de classes *inv: solde < decouvertMax*
- *pre*: pré-conditions d'opérations *pre: montantARetirer > 0*
- *post*: post-conditions d'opérations *post: solde > solde@pre*

- OCL peut également être utilisé pour décrire des expressions

- *def*: déclarer des attributs ou des opérations *def: nbEnfants:Integer*
- *init*: spécifier la valeur initiale des attributs *init: enfants->size()*
- *body*: exprimer le corps de méthodes {query} *body: enfants->select(age < a)*
- *derive*: définir des éléments dérivés (/) *derive: age < 18*

# Navigation des relations 0..\*

---

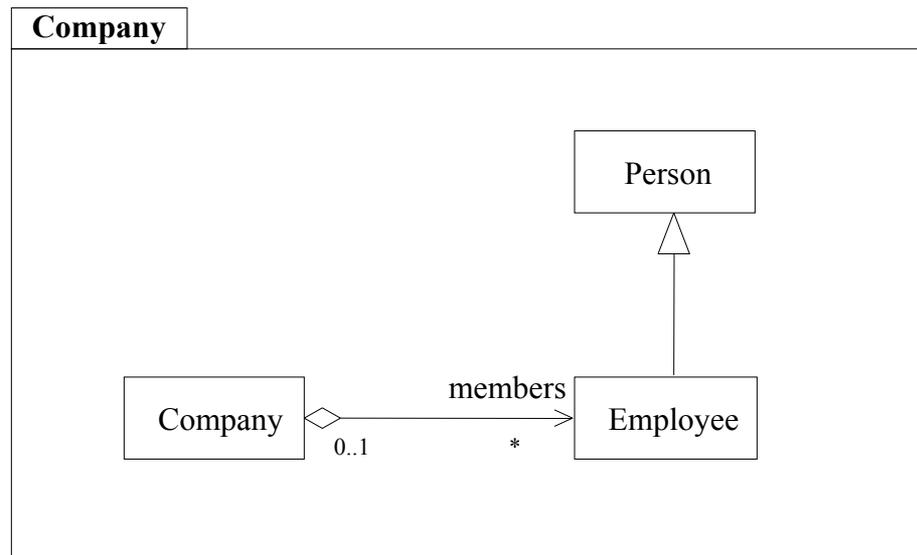
- Par navigation on n'obtient plus un scalaire, mais une *collection* d'objets
- OCL défini 3 sous-types de collections
  - **Set** : obtenu par navigation d'une relation 0..\*
    - *Context* *Personne inv: propriété* retourne un Set[Voiture]
    - chaque élément est présent au plus une fois
  - **Bag** : si plus d'un pas de navigation
    - un élément peut être présent plus d'une fois
  - **Sequence** : navigation d'une relation {ordered}
    - c'est un Bag ordonné
- Nombreuses opérations prédéfinies sur les types *collection*.

Syntaxe :  
Collection->opération

# Class Diagram: Conclusion

---

- Un diagramme de classes est un graphe d'éléments connectés par des relations.
- Un diagramme de classes est une vue graphique de la structure statique d'un système.



# Constituants d'une classe

---

- Concept représenté (nom)
- Classes héritées (concepts précisés)
- Relations avec autres classes
- Attributs (classe, nom, visibilité)
- Opérations (paramètres)
- Contraintes, invariants
- Généricité (classes paramétrées)
- Stéréotypes

# Structural View

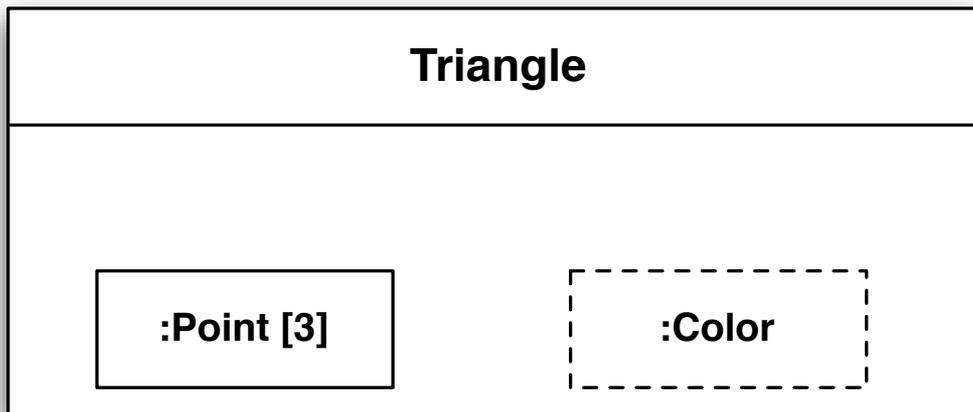
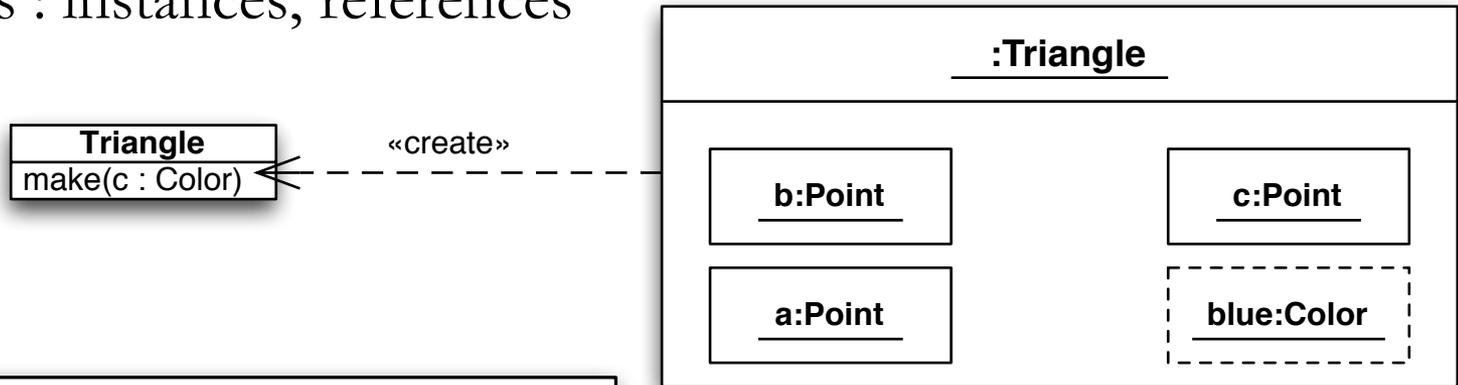
---

## Composite Structure Diagram

# Diagramme de composite structure

- Structure Interne

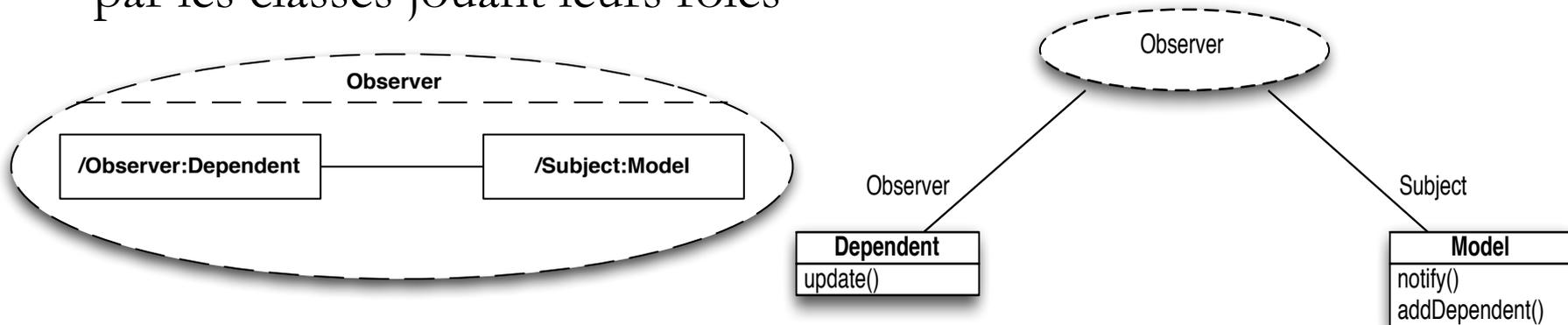
- Spécification d'éléments créés à l'intérieur d'un classificateur
- Eléments : instances, références



# Diagramme de composite structure

## • Collaboration

- Spécification de la structure d'un ensemble d'éléments (rôles) qui réalisent une tâche commune
- Représentation de la structure d'un patron de conception
- Éléments : rôle (d'une classe participant à une collaboration), connecteurs (permettant la communication entre deux classes)
- Les propriétés définies par les classificateurs doivent être possédées par les classes jouant leurs rôles

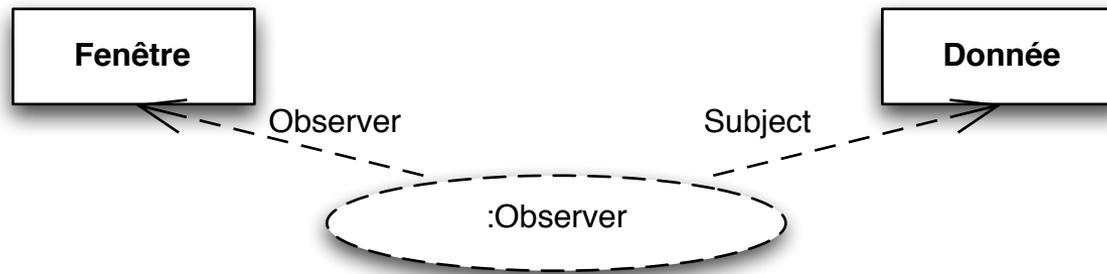


# Diagramme de composite structure

---

- Collaboration

- Exemple :



# Structural View

---

## Package Diagram

# Notion de package

---

- **Élément structurant les classes**

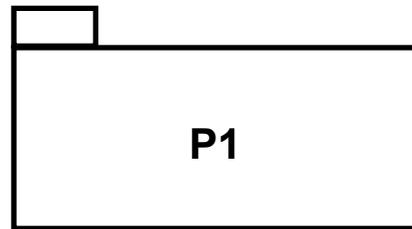
- Modularisation à l'échelle supérieure
- Un package (paquetage en français) partitionne l'application :
  - Il référence ou se compose des classes de l'application
  - Il référence ou se compose d'autres packages
- Un package régleme la visibilité des classes et des packages qu'il référence ou le compose
- Les packages sont liés entre eux par des liens d'utilisation, de composition et de généralisation
- Un package est la représentation informatique du contexte de définition d'une classe

**N.B.: une classe appartient à un et un seul package**

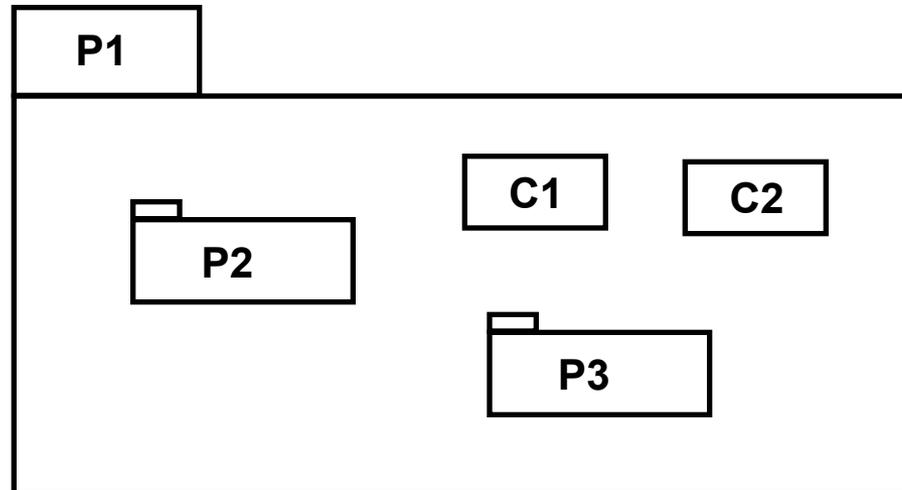
# Représentation d'un package

---

- Vue graphique externe



- Vue graphique externe et interne



# Visibilité dans un package

---

- **Réglementation de la visibilité des classes**
  - **Classes de visibilité publique :**
    - classes utilisables par des classes d'autres packages
  - **Classes de visibilité privée :**
    - classes utilisables seulement au sein d'un package
- **Représentation graphique :**

**{public}**

***CLASSE D'INTERFACE***

**Classe  
{private}**

***CLASSE DE CORPS***

**Package::Classe**

***CLASSE EXTERNE***

# Utilisation entre packages

---

- Définition

- Il y a utilisation entre packages si des classes du package utilisateur accèdent à des classes du package utilisé
- Pour qu'une classe d'un package p1 puisse utiliser une classe d'un package p2, il doit y avoir au préalable une déclaration **explicite** de l'utilisation du package p2 par le package p1

- Représentation graphique

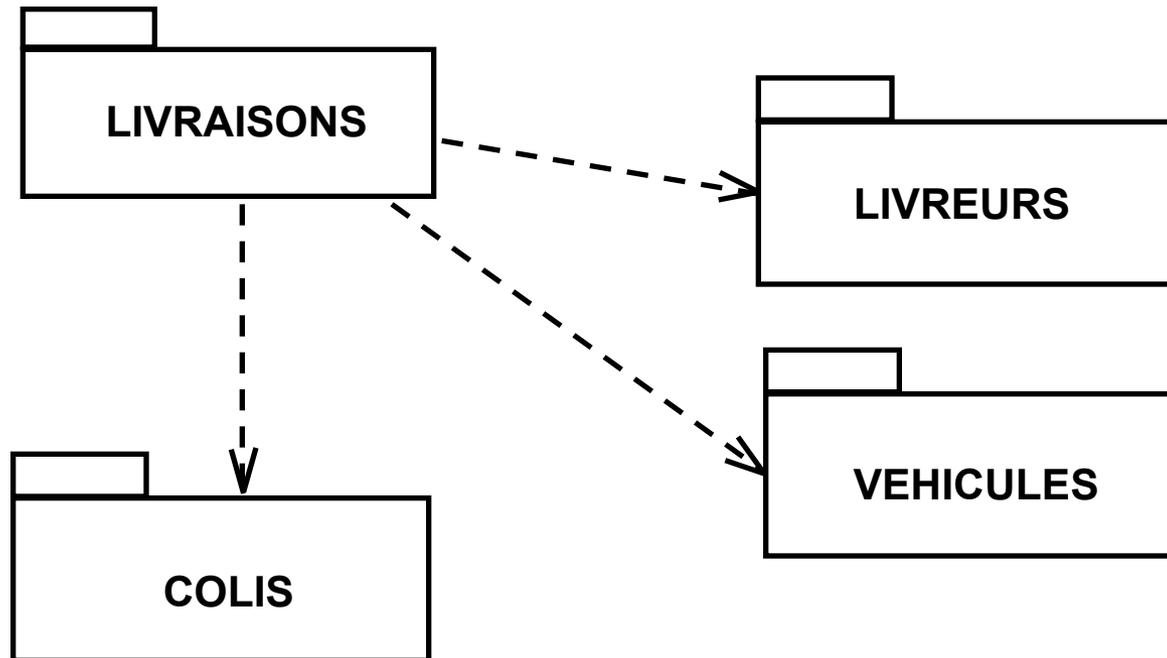


***Vue externe du package P1***

# Utilisation entre packages

---

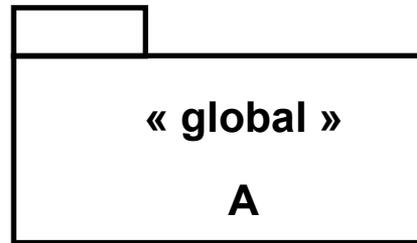
- Exemple (vue externe du package livraisons) :



# Stéréotypes de package

---

- Stéréotype « global »

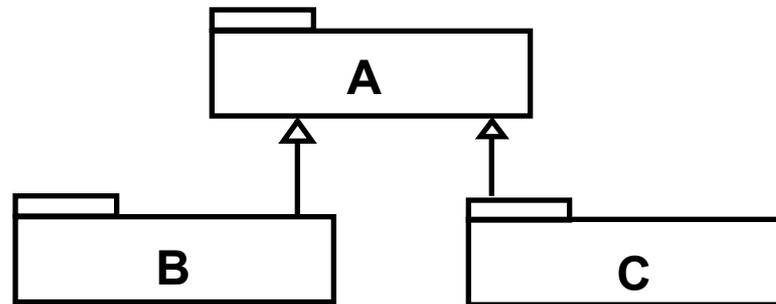


- Tous les packages du système ont une dépendance avec ce package
- Ne pas abuser de ce stéréotypes ;)

# Héritage entre packages

---

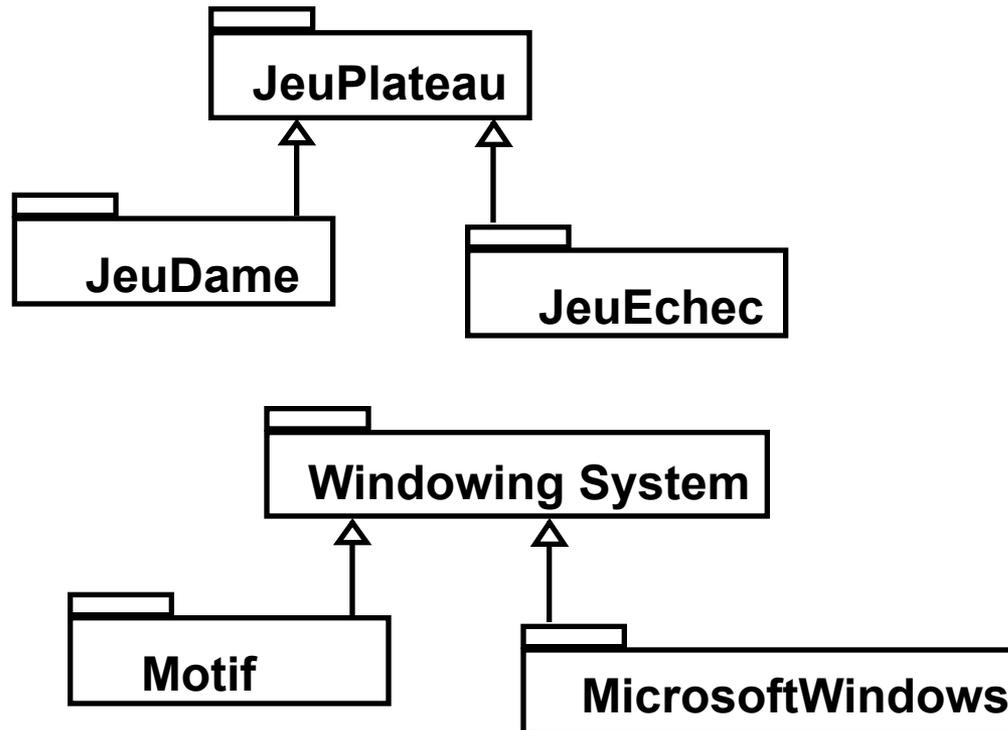
- Le package spécifique B doit être conforme à l'interface du package A
- Intérêt : substitution



# Héritage entre packages

---

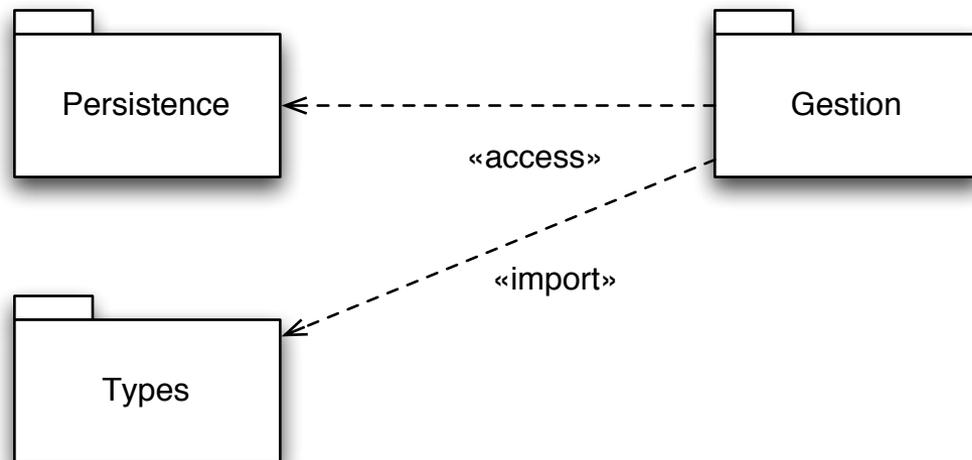
- Exemple :



# Dépendances entre packages

---

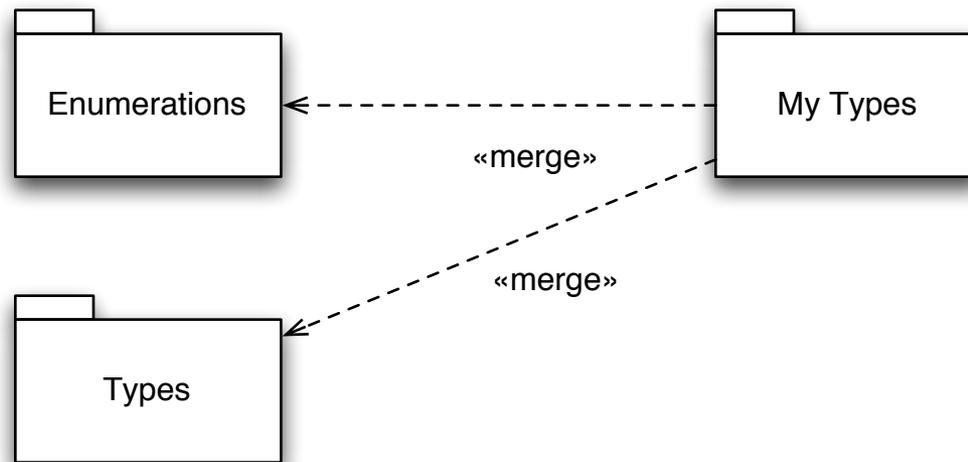
- **Importation («import»):**
  - les éléments de l'espace de nommage sont importés
- **Accès («access»):**
  - les éléments sont simplement utilisés



# Fusion (*merge*) de packages

---

- Une fusion définit la manière dont un paquetage étend un autre
- Composé de liens de généralisation et de redéfinition



# Utilité des packages

---

- Réponses au besoin
  - Contexte de définition d'une classe
  - Unité de structuration
  - Unité d'encapsulation
  - Unité d'intégration
  - Unité de réutilisation
  - Unité de configuration
  - Unité de production

# Structuration par packages (*vs.*) décomposition hiérarchique

---

- Pour les grands systèmes, il est nécessaire de disposer d'une unité de structuration
    - À un niveau supérieur,
    - Plus souple que :
      - La composition de classe
      - Le référencement de packages
- ⇒ domaines de structuration :
- ⇒ Packages décomposables en packages

# Outline

---

① UML History and Overview

② UML Language

① UML Functional View

② UML Structural View

③ UML Behavioral View

④ UML Implementation View

⑤ UML Extension Mechanisms

③ UML Internals

④ UML Tools

⑤ Conclusion

# Aspects dynamiques du système

---

- Jusqu'ici, le système est décrit *statiquement*:
  - Décrit les messages (méthodes ou opérations) que les instances des classes peuvent recevoir mais ne décrivent pas l'émission de ces messages
  - Ne montrent pas le lien entre ces échanges de messages et les processus généraux que l'application doit réaliser
- Il faut maintenant décrire comment le système évolue dans le temps
- On se focalise alors sur les *collaborations* entre objets. Rappel :
  - objets : simples
  - gestion complexe : par collaborations entre objets simples

# Behavioral View

---

Sequence  
Diagram

(scenarios)

# Diagramme de séquences (scénarios)

---

- Représentation des interactions entre acteurs et objets
- Vision temporelle d'une interaction
  - Chaque objet est symbolisé par une barre verticale (i.e., ligne de vie)
  - Le temps s'écoule de haut en bas, de sorte que la numérotation des messages est optionnelle.
  - Diagramme dual du diagramme de collaboration
- Souvent utilisé pour représenter une instance de cas d'utilisation
- De manière plus générale, représentation temporelle d'une interaction
  - Bien adapté pour de longues séquences
  - Ne visualise pas les liens
  - Complémentaire du diagramme de collaboration

# Diagramme de séquences (scénarios)

---

- Dérivé des scénarios de OMT :
  - Montrent des exemples de coopération entre objets dans la réalisation de processus de l'application
  - Illustrent la dynamique d'enchaînement des traitements à travers les messages échangés entre objets
  - le temps est représenté comme une dimension explicite
    - en général de haut en bas
- Les éléments constitutifs d'un scénario sont :
  - Un ensemble d'objets (et/ou d'acteurs)
  - Un message initiateur du scénario
  - La chronologie des messages échangés subséquentement
  - Les contraintes de temps (aspects temps réel)

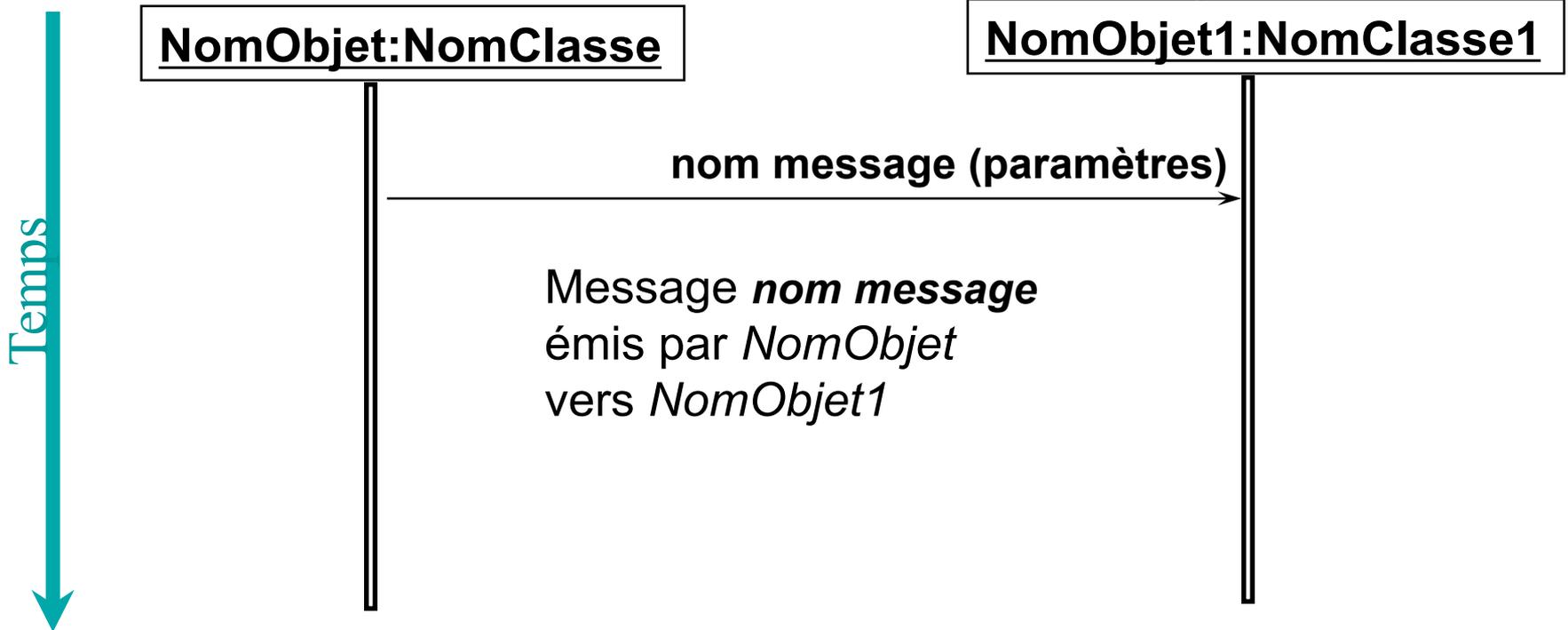
# Messages et Stimuli

---

- Un **Stimulus** est une communication entre deux instances, dans le but de déclencher une action.
- Un **Message** est une spécification de **Stimulus**, qui définit les rôles de l'émetteur et du récepteur.

# Diagramme de séquences : notation

*Objets = Instances de classes*



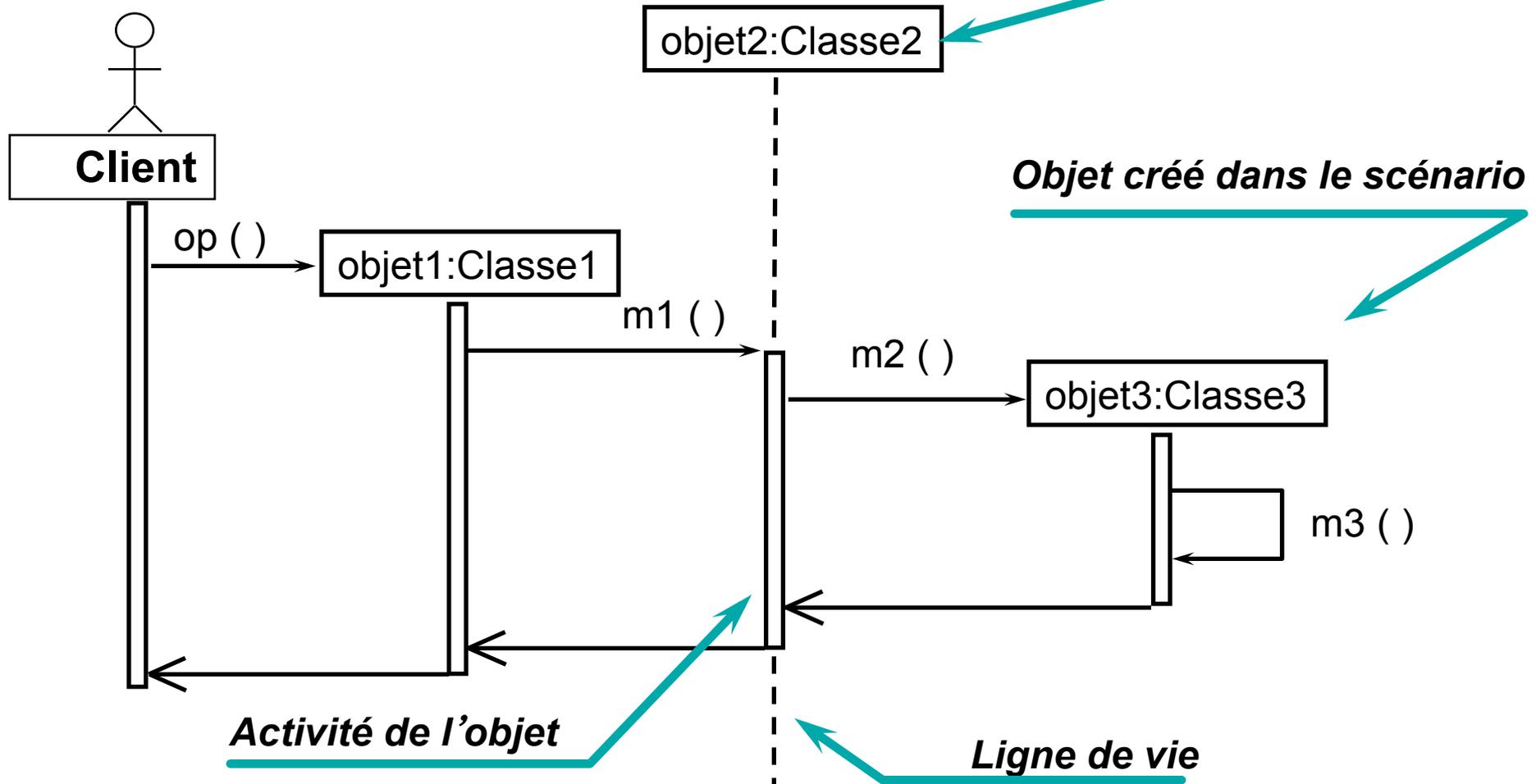
# Ligne de vie et activation

---

- La « ligne de vie » représente l'existence de l'objet à un instant particulier
  - Commence avec la création de l'objet
  - Se termine avec la destruction de l'objet
- L'activation est la période durant laquelle l'objet exécute une action lui-même ou via une autre procédure

# Diagramme de séquences : notation

**Objet existant avant et après l'activation du scénario**



# Messages

---

- Communication entre objets
  - Des paramètres
  - Un retour
  
- Cas particuliers
  - Les messages entraînant la construction d'un objet
  - La récursivité
  - La destruction d'objets

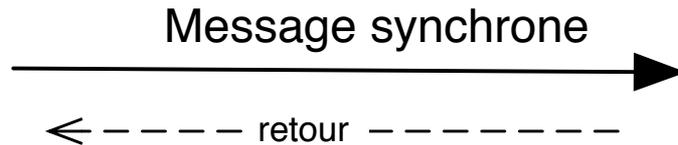
# Messages

---

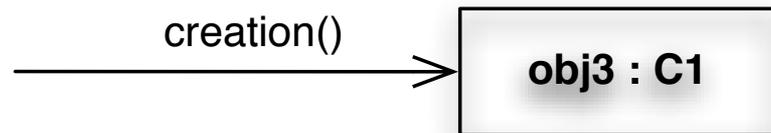
- Asynchrone



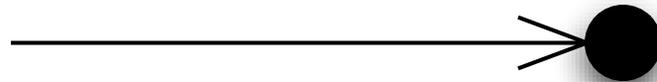
- Synchrone



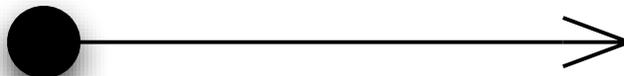
- Création



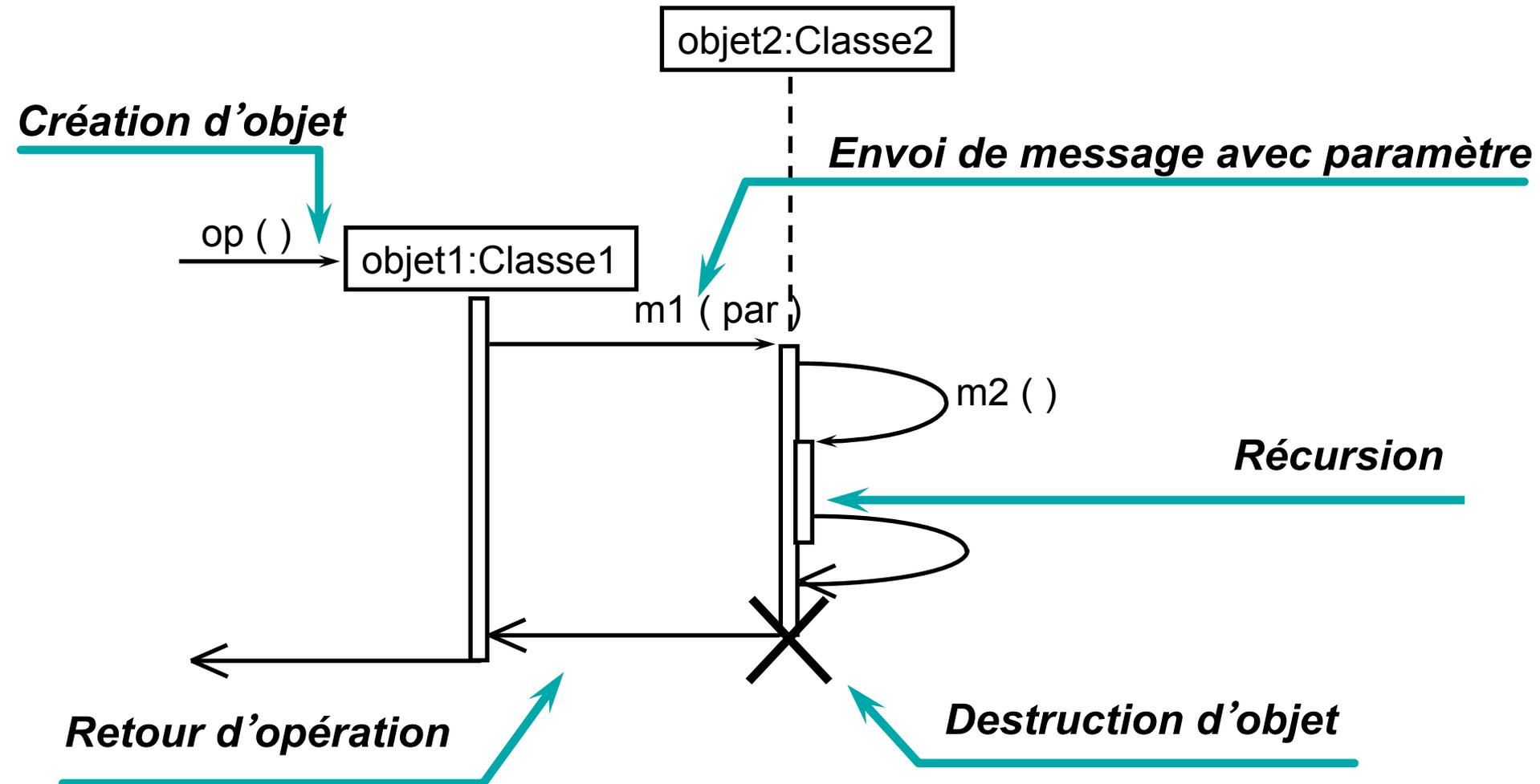
- Perdu



- Trouvé

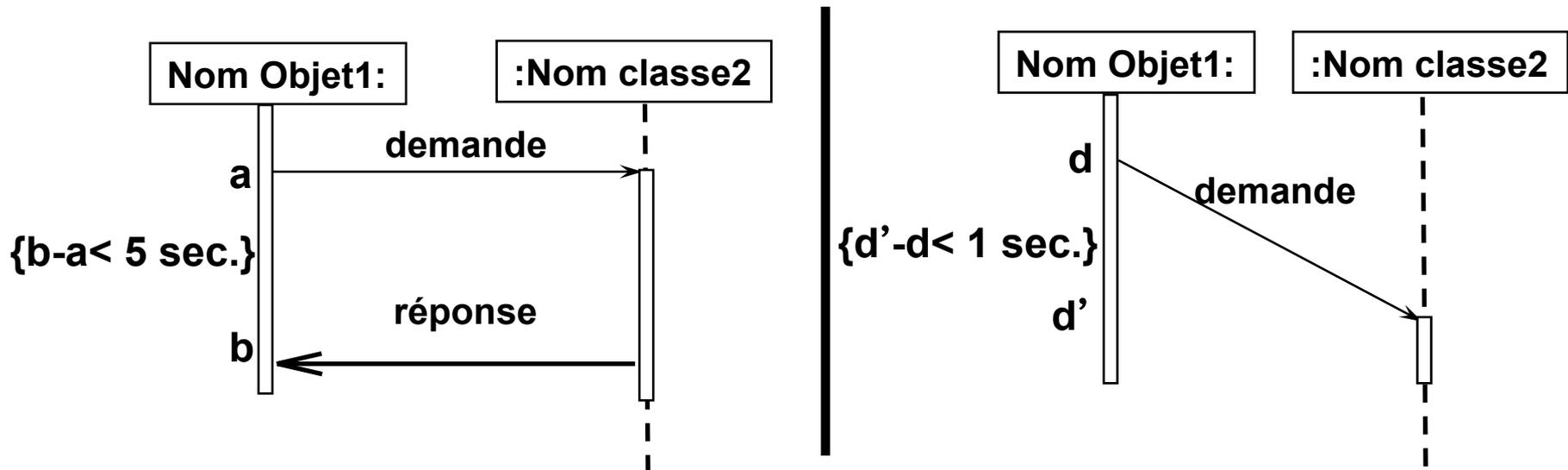


# Diagramme de séquences : notation

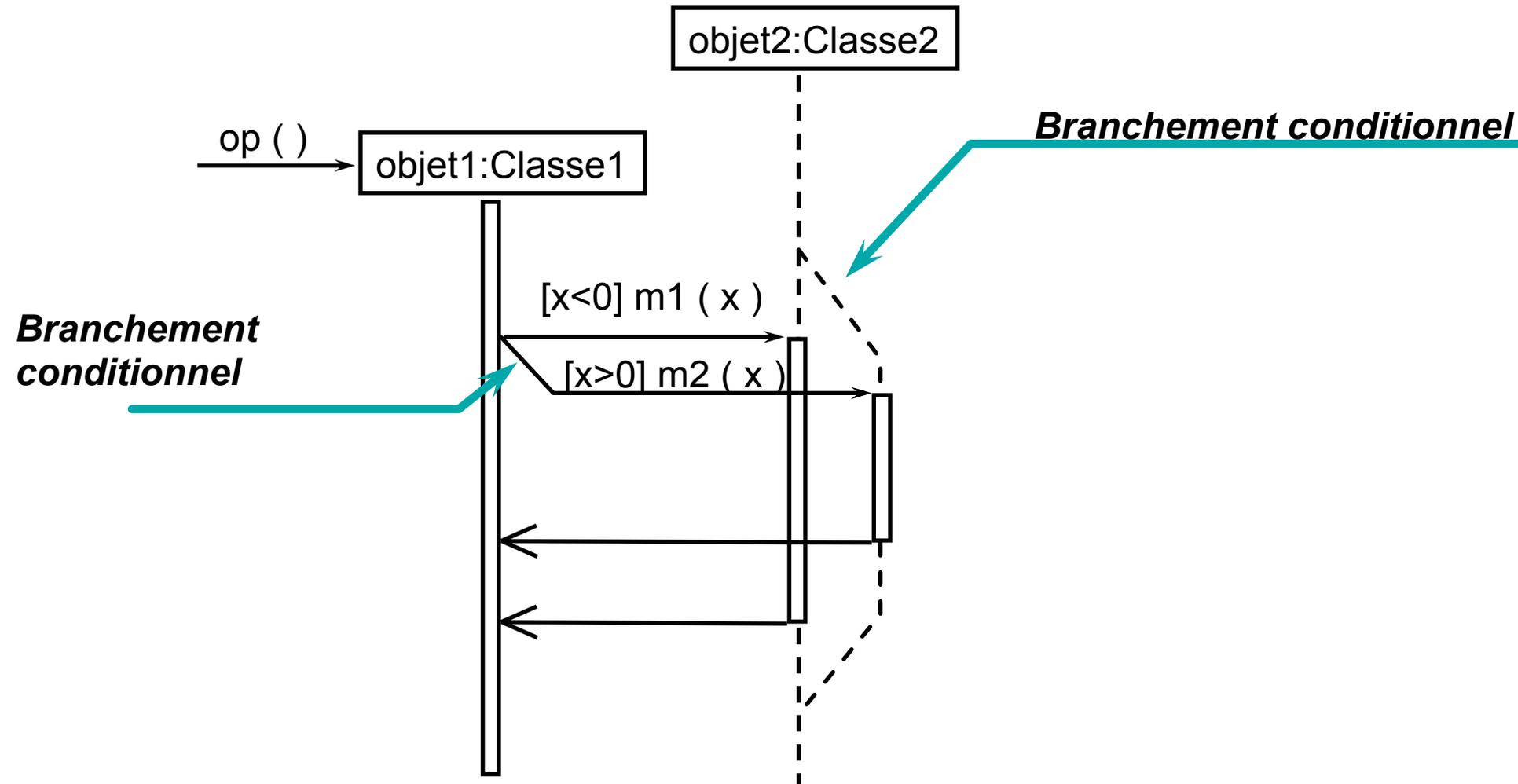


# Aspects asynchrones et temps réel

- Lecture du scénario et chronologie
  - Un scénario se lit de haut en bas dans le sens chronologique d'échange des messages.
  - Des contraintes temporelles peuvent être ajoutées au scénario

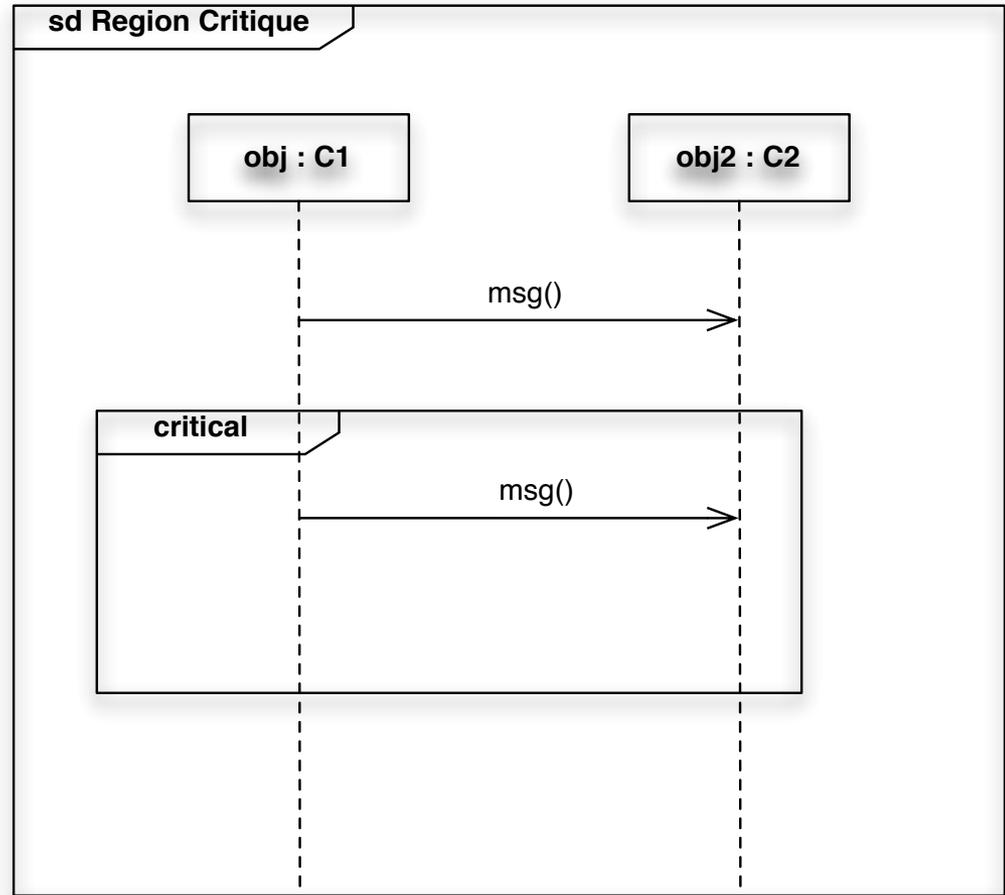


# Représentation de conditionnelles



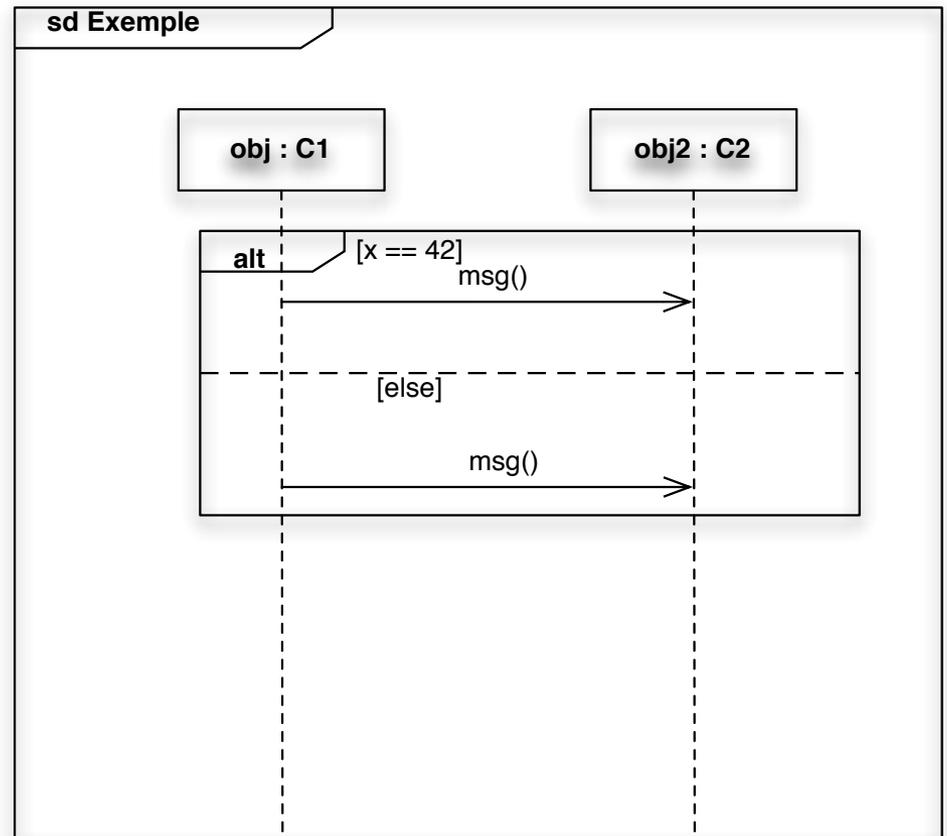
# Fragments (UML 2.x)

- Région définissant une sémantique précise à l'intérieur d'une interaction
- Raccourcis pour l'écriture de traces
- Composé de [1..\*] "opérandes"



# Fragments - alt

- Alternatives (alt):  
choix de  
comportement, entre  
deux opérandes



# Fragments - autres

---

- Option (**opt**): l'opérande est optionnelle. Elle peut ne pas exister
- Break (**break**): l'opérande est exécutée, à la place du reste de l'interaction (raccourci pour alternative)
- Parallèle (**par**): le comportement spécifié par les deux opérandes peut s'exécuter en parallèle (fusion entre les messages)
- Négatif: le fragment représente des traces invalides
- Région critique (**critical**): les traces spécifiées ne peuvent pas être intercalées
- Assertion (**assert**): spécification de la seule continuation valable
- Ignorer et Considérer (**ignore** | **consider**): spécification des messages significatifs
- Boucle (**loop**): l'opérande sera répétée autant de fois que défini par une garde
- Autres: strict, ordonnancement strict, ordonnancement faible

# Behavioral View

---

## Colaboration Diagram

(between objects)

# Diagramme de collaboration

---

- Les scénarios et diagrammes de collaboration:
  - Montrent des exemples de coopération des objets dans la réalisation de processus de l'application
- Les scénarios :
  - Illustrent la dynamique d'enchaînement des traitements d'une application en introduisant la dimension temporelle
- Les diagrammes de collaboration
  - Dimension temporelle représentée par numéros de séquence : définition d'un ordre partiel sur les opérations
  - Représentation des objets et de leurs relations
  - Utilisent les attributs et opérations

# Diagrammes de collaboration

---

- Représentation d'une collaboration entre rôles
  - Booch : Société d'objets collaborant (UML 1.5 : de rôles communicants)
- Représentation spatiale d'une interaction
  - Mise en avant de la structure
  - Représentation des structures complexes (récursives par exemple)
- Pas d'axe temporel
  - Diagramme dual du diagramme de séquence
- Des rôles ou des objets dans une situation donnée
- Des liens relient les objets qui se connaissent
- Les messages échangés par les objets sont représentés le long de ces liens
- L'ordre d'envoi des messages est matérialisé par un numéro de séquence

# Diagrammes de collaboration

---

- Représentation de collaborations de rôles

Description de la réalisation d'un Classifier ou d'une opération

Ensemble de rôles (ClassifierRoles + AssociationRoles)

/ ClassifierRoleName : ClassifierName

- Représentation de collaborations d'instances

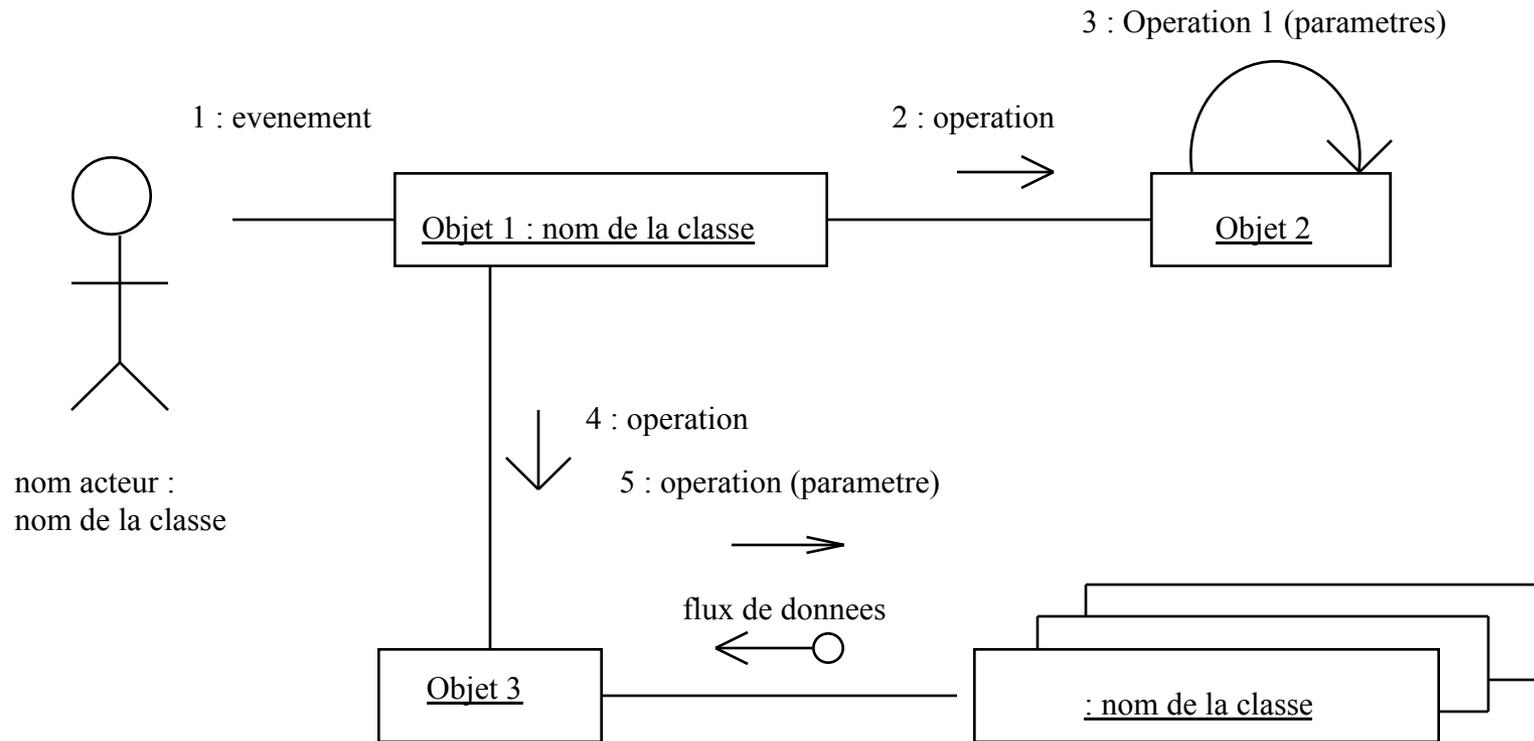
- Ensemble d'objets (Objets + Liens)

- Conforme à un diagramme de collaboration de rôles

- Représentation d'interactions

- Ensembles d'objets + Stimuli (Instances de Messages)

# Diagrammes de collaboration : notation



# Messages (exemples)

---

- **4 : Afficher (x, y)**

  - `--message simple`

- **3.3.1 : Afficher (x, y)**

  - `--message imbriqué`

- **4.2 : âge := Soustraire (Today, Birthday)**

  - `--message imbriqué avec valeur retournée`

- **[Age >= 18 ans] 6.2 : Voter ()**

  - `--message conditionnel`

- **a.4, b.6 / c.1 : Allumer (Lampe)**

  - `-- synchronisation avec d'autres flots d'exécution`

- **1 \* : Laver ()**

  - `--itération`

- **3.a, 3.b / 4 \*|| [i := 1..n] : Eteindre ()**

  - `--itération parallèle`

# Utilisation des diagrammes de collaboration

---

- Ils peuvent être attachés à :
  - Une classe
  - Une opération
  - Un use-case
- Ils s'appliquent
  - En spécification
  - En conception (illustration de *design patterns*)

# Éléments constitutifs

---

- Un contexte contenant les éléments mis en jeu durant l'opération :
  - Un acteur
  - Un ensemble d'objets, d'attributs et de paramètres
  - Des relations entre ces objets
- Des interactions
  - Des messages
  - Un message initiateur du diagramme provenant d'un
    - Acteur de l'application,
    - Objet de l'application.
  - Les numéros de séquence des messages échangés entre les objets de cet ensemble suite au message initiateur

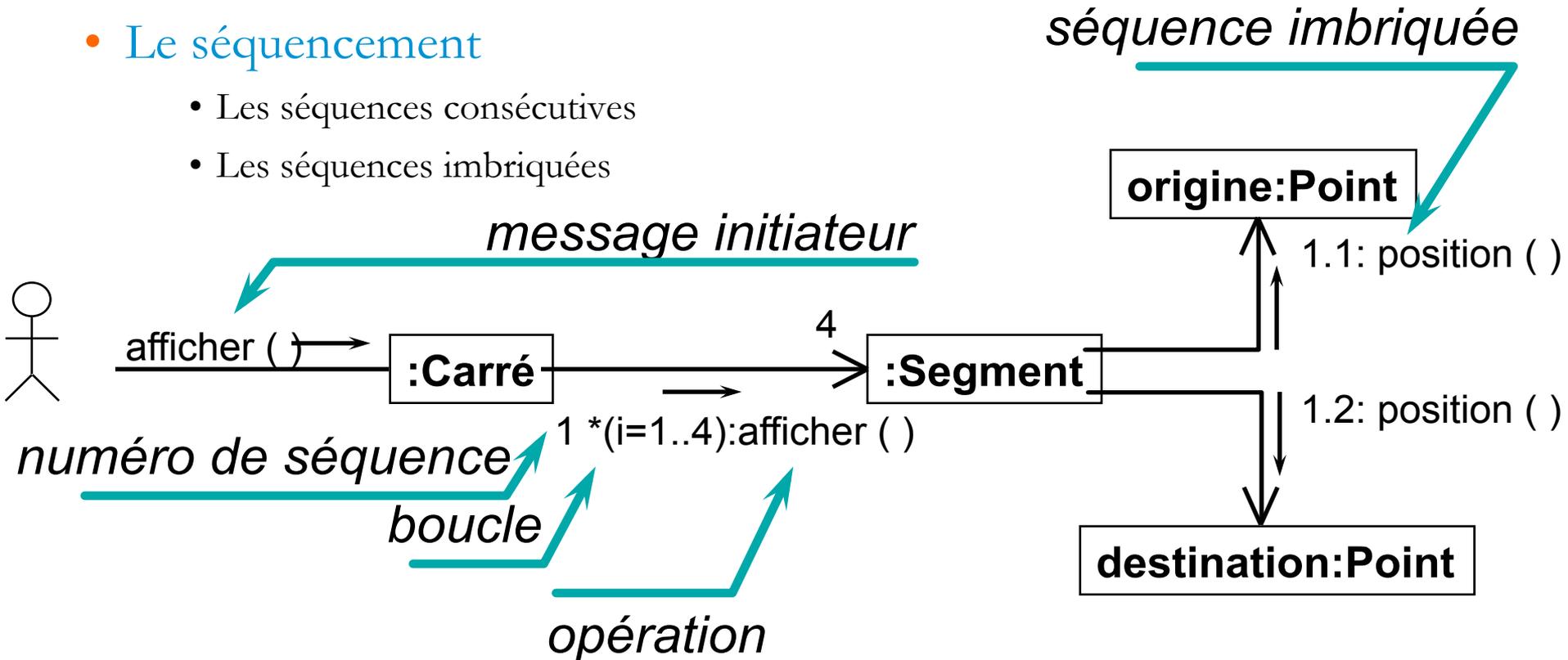
# Diagrammes de collaboration : notation

- Les messages

- Opérations
- Réception d'événements

- Le séquençement

- Les séquences consécutives
- Les séquences imbriquées



# Questions auxquelles répondent les collaborations

---

- Quel est l'objectif ?
- Quels sont les objets ?
- Quelles sont leurs responsabilités ?
- Comment sont-ils interconnectés ?
- Comment interagissent-ils ?

# Behavioral View

---

## State Machine Diagram

# Machines à états

---

- Machines à états = diagrammes d'états-transitions
- Il existe deux sortes de diagrammes d'état
  - De comportement (behavioral state machine)
  - De protocole (protocol state machine)

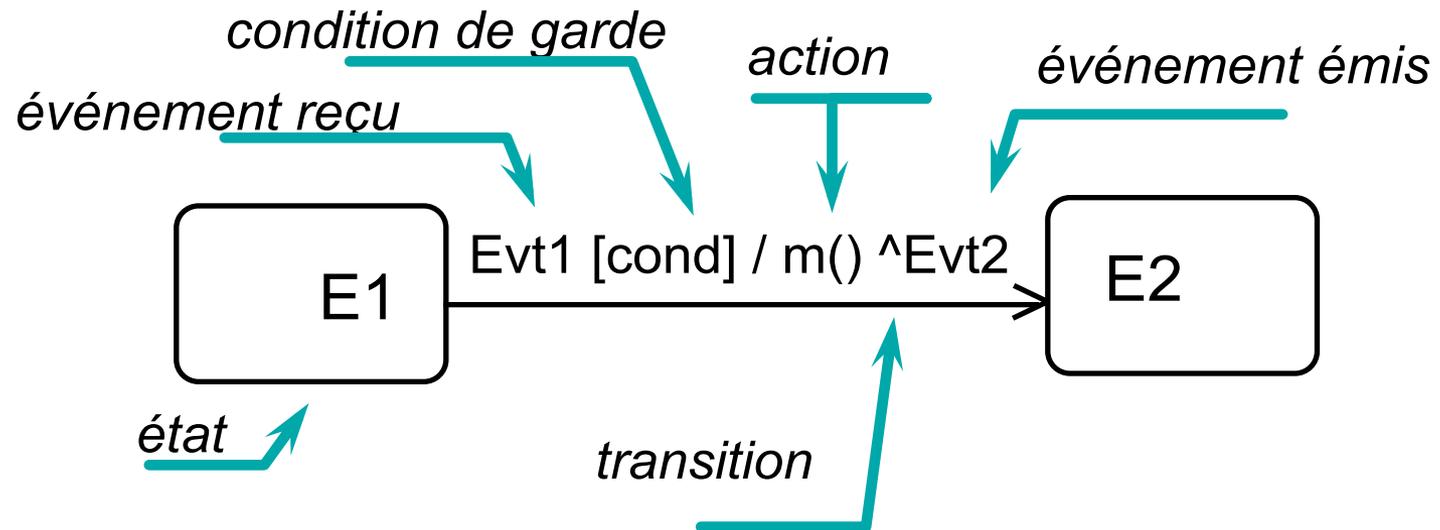
# Machines à états comportementale

---

- Attachés à une classe ou à une opération
  - Généralisation des scénarios
  - Description systématique des réactions d'un objet aux changements de son environnement
- Décrivent les séquences d'états d'un objet ou d'une opération :
  - En réponse aux «stimulis» reçus
  - En utilisant ses propres actions (transitions déclenchées)
- Réseau d'états et de transitions
  - Automates étendus
  - Essentiellement *Diagrammes d'Harel (idem OMT)*

# Machines à états : notation

---



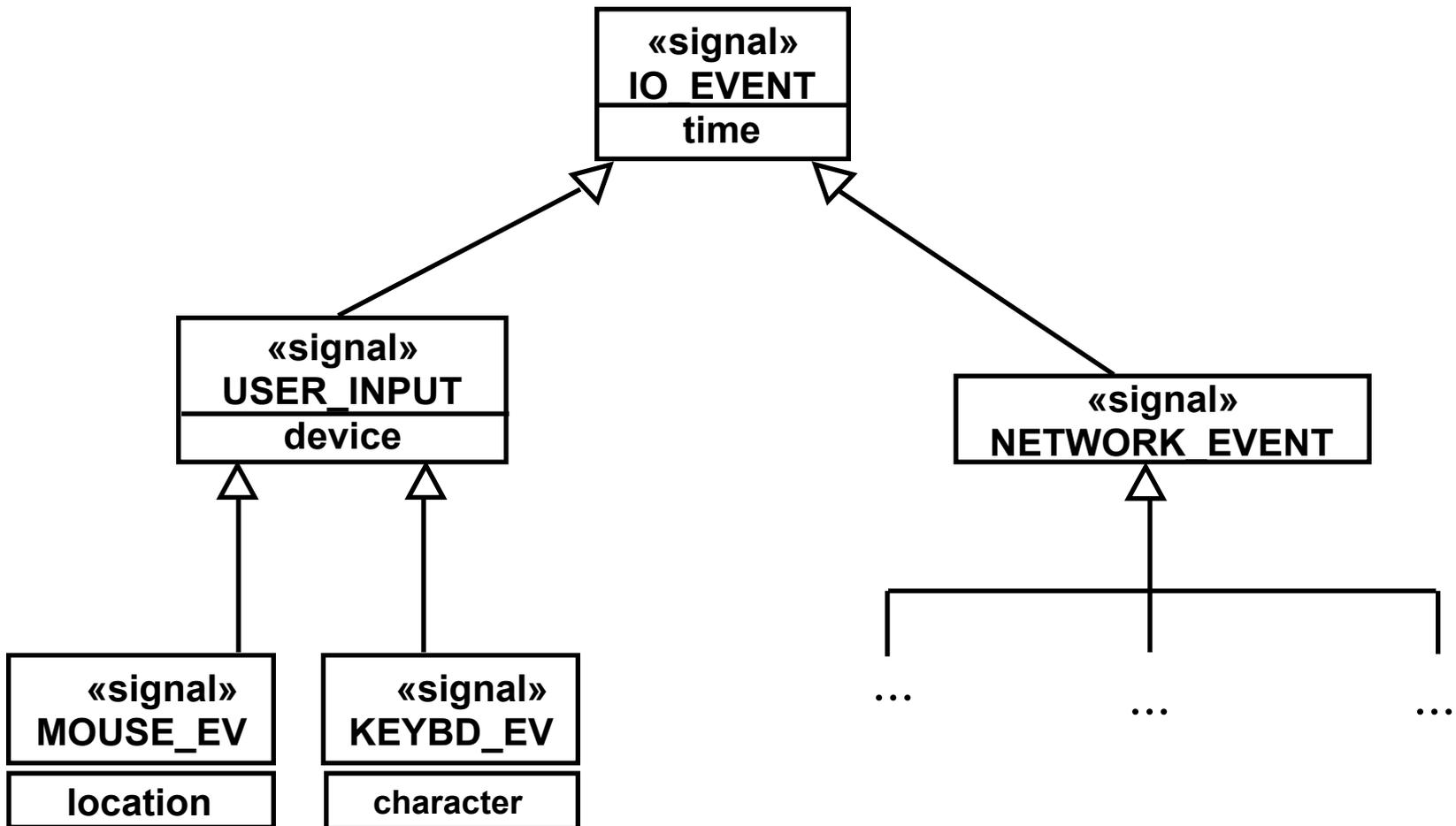
# Notion d'événements

---

- Stimuli auxquels réagissent les objets
  - Occurrence déclenchant une transition d'état
- Abstraction d'une information instantanée échangée entre des objets et des acteurs
  - Un événement est instantané
  - Un événement correspond à une communication unidirectionnelle
  - Un objet peut réagir à certains événements lorsqu'il est dans certains états.
  - Un événement appartient à une *classe d'événements* (classe stéréotypée «signal»).

# Notion d'événements

- Les événements sont considérés comme des objets



# Typologie d'événements

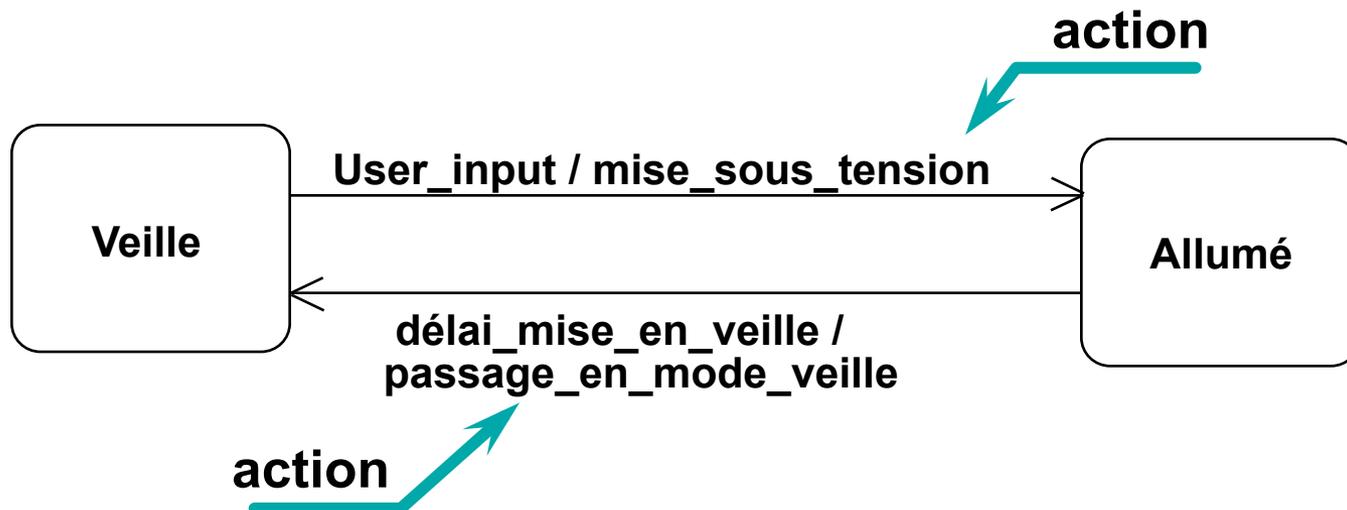
---

- Réalisation d'une condition arbitraire
  - transcrit par une condition de garde sur la transition
- Réception d'un signal issu d'un autre objet
  - transcrit en un événement déclenchant sur la transition
- Réception d'un appel d'opération par un objet
  - transcrit comme un événement déclenchant sur la transition
- Période de temps écoulée
  - transcrit comme une expression du temps sur la transition

# Notion d'action

---

- Action : opération *instantanée* (conceptuellement) et *atomique* (ne peut être interrompue)
- Déclenchée par un événement
  - Traitement associé à la transition
  - Ou à l'entrée dans un état ou à la sortie de cet état



# Notion d'état

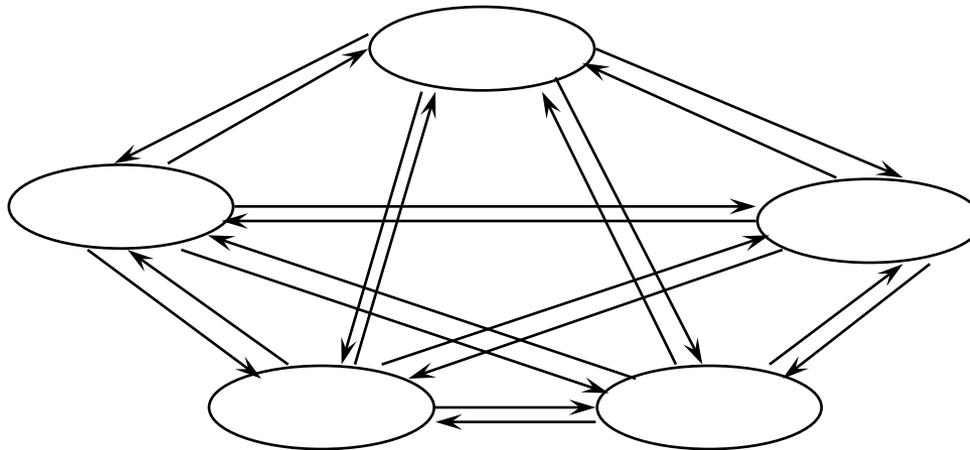
---

- Etat : situation stable d'un objet parmi un ensemble de situations prédéfinies
  - conditionne la réponse de l'objet à des événements
    - programmation réactive / « temps réel »
  - Intervalle entre 2 événements, il a une durée
- Peut avoir des variables internes
  - attributs de la classe supportant ce diagramme d'états

# Structuration en sous-états

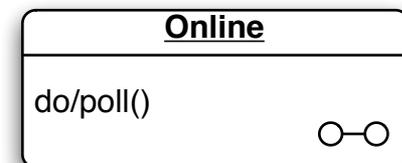
- Problème d'un diagramme d'états plats

- Pouvoir d'expression réduit, inutilisable pour de grands problèmes
- Explosion combinatoire des transitions.



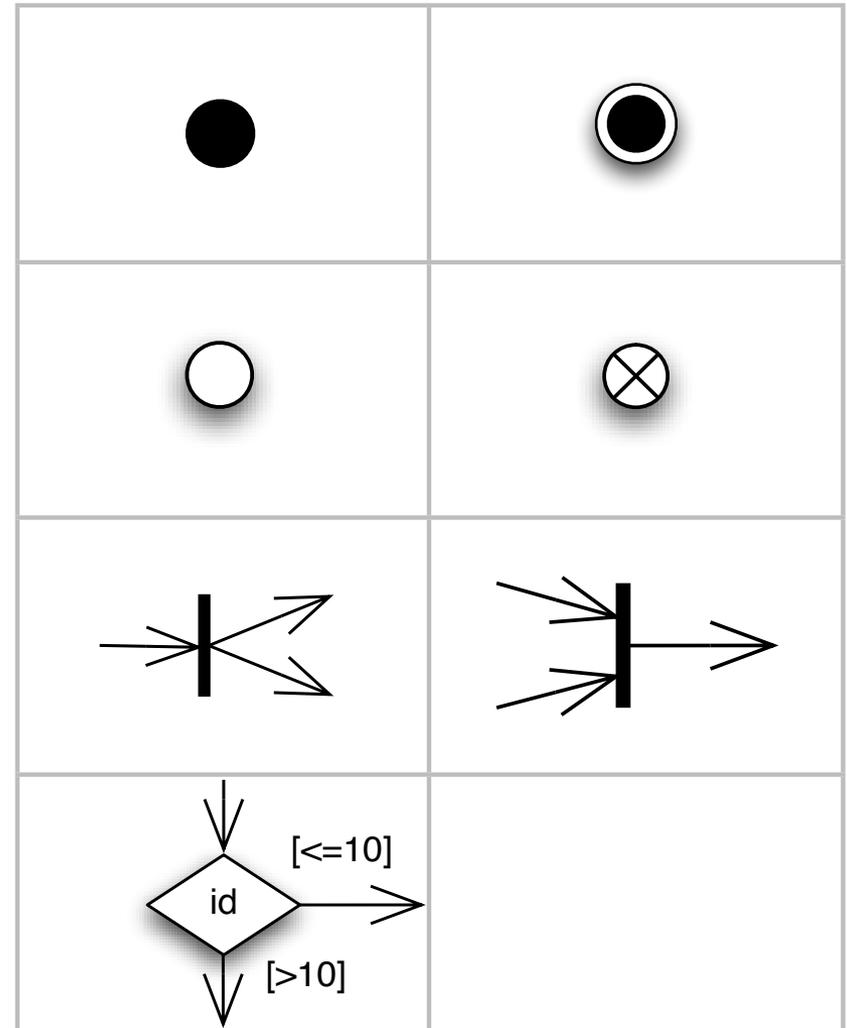
- Structuration à l'aide de super/sous états (+ hiérarchies d'événements)

- représentés par imbrication graphique ou



# Pseudo-états

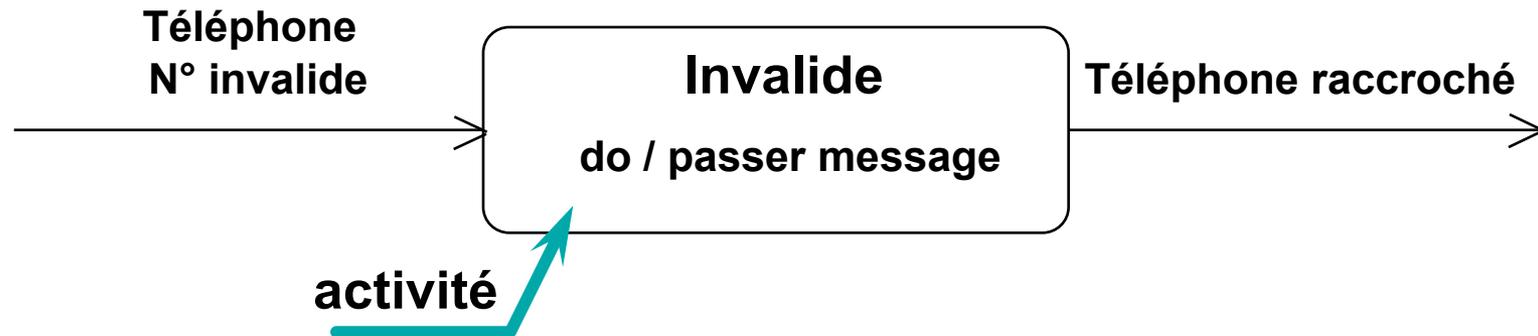
- Initial, final
- Entrée, sortie
- Fourchette, jonction
- Choix



# Notion d'activité dans un état

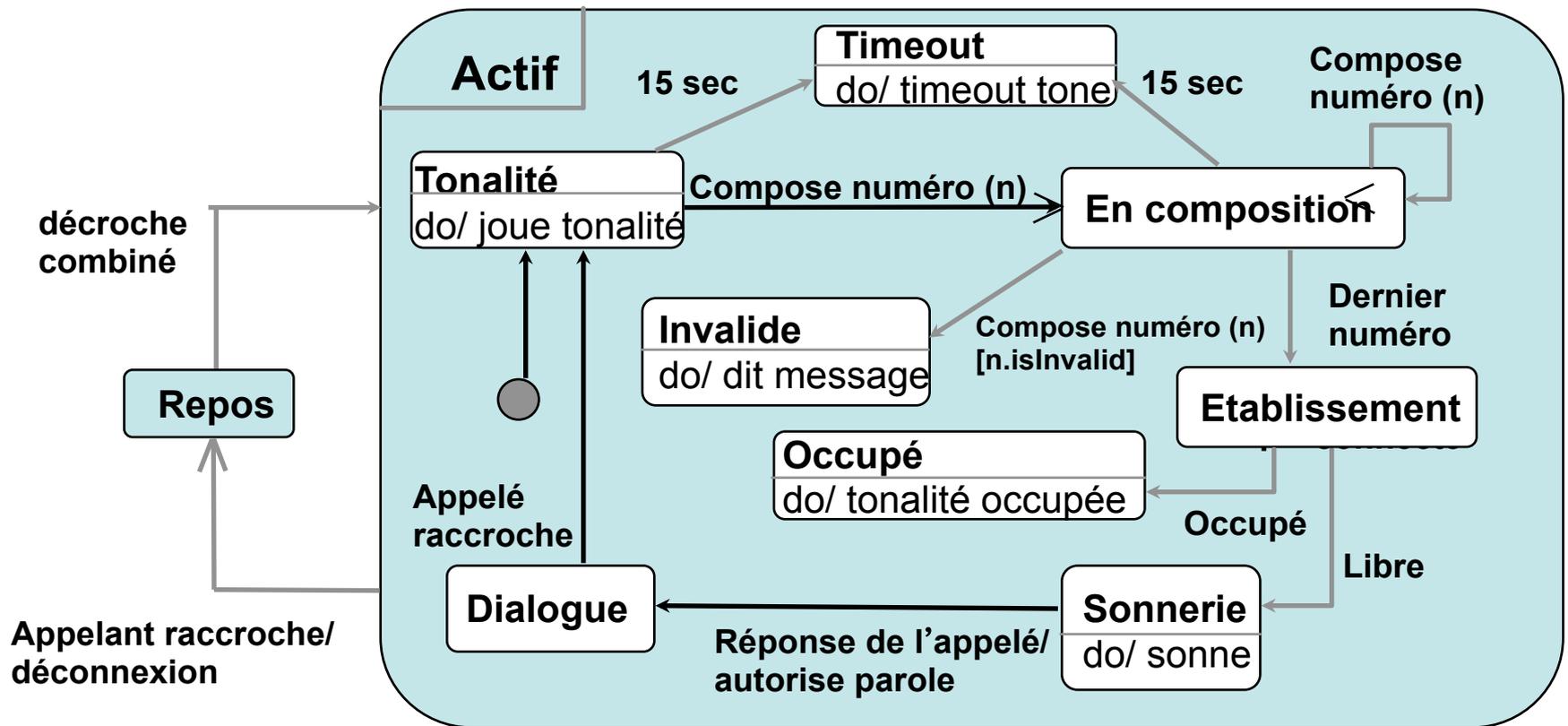
---

- **Activité** : opération se déroulant continuellement tant que l'on est dans l'état associé
  - *do / action*



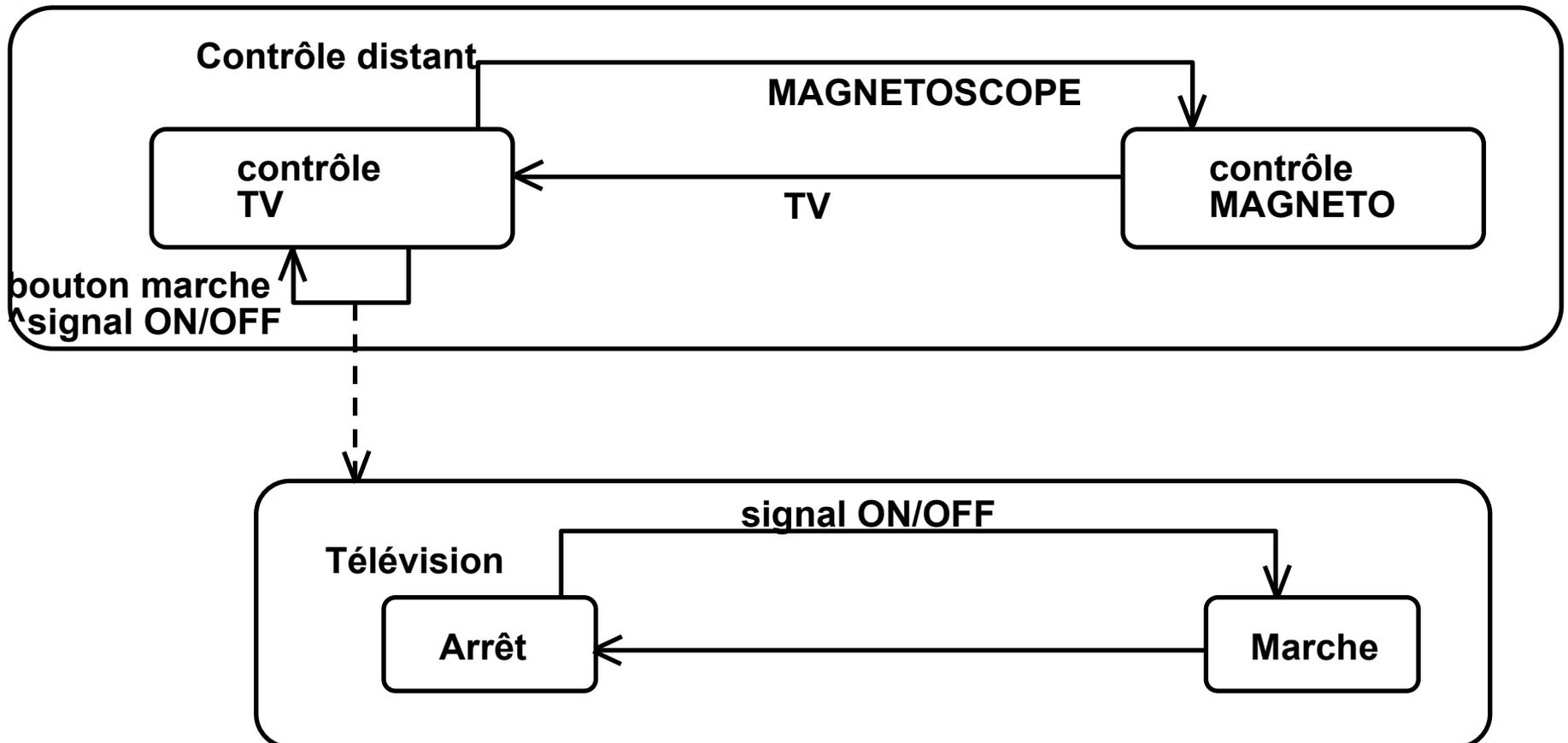
- Une activité peut être interrompue par un événement.

# Exemple de diagramme d'états



# Émission d'événements

- Automate d'états d'une télécommande double (TV + MAGNETOSCOPE) :

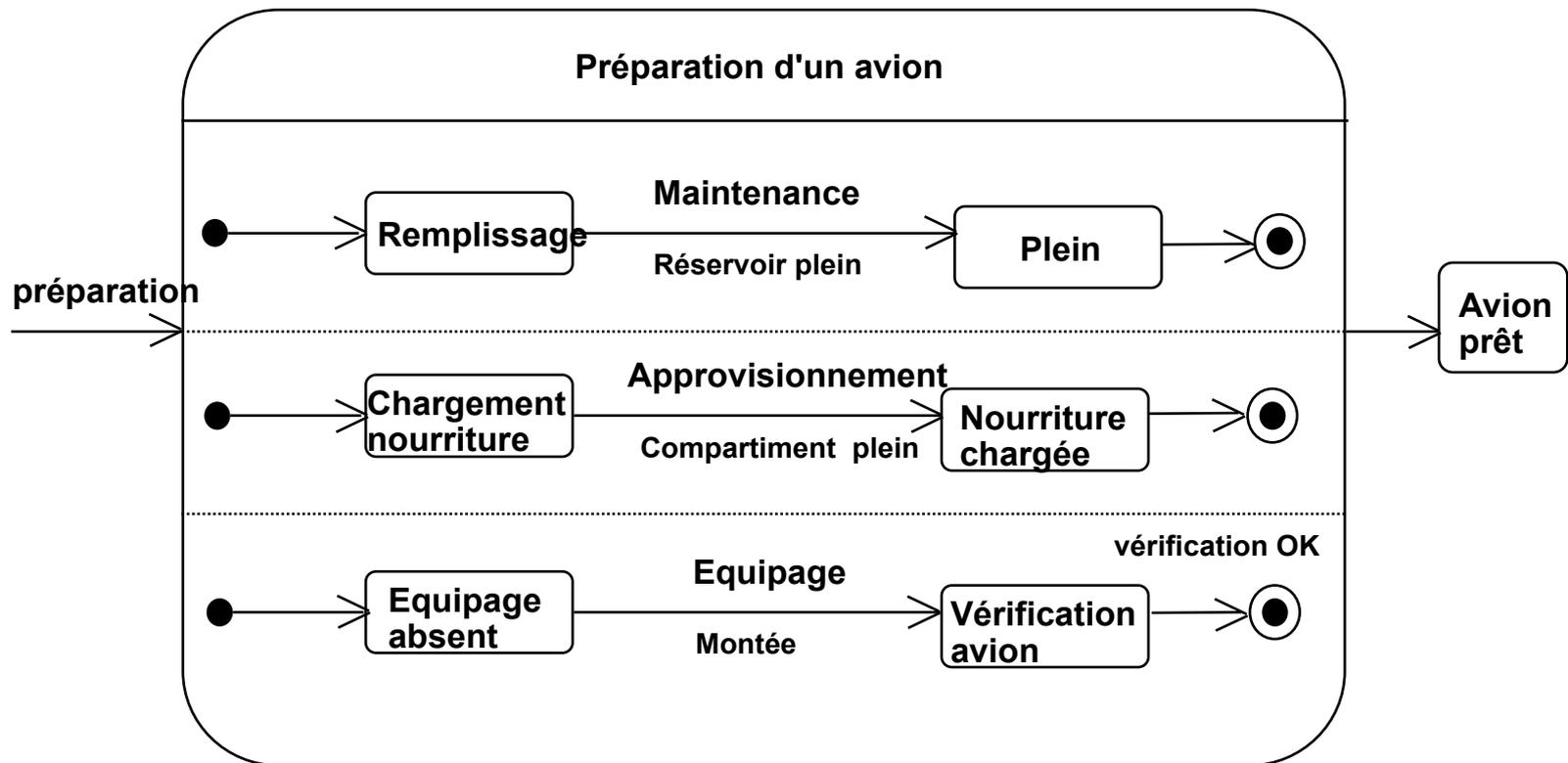


# Diagrammes d'états concurrents

---

- Utilisation de sous-états concurrents pour ne pas à avoir à expliciter le produit cartésien d'automates
  - si 2+ aspects de l'état d'un objet sont indépendants
  - activités parallèles
- Sous-états concurrents séparés par pointillés
  - « *swim lanes* »

# Diagrammes d'états concurrents : exemple



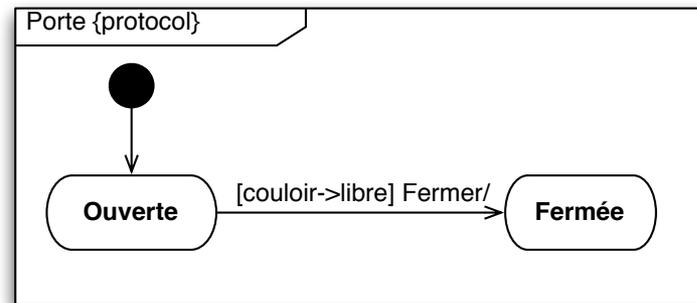
# Etat-transition (résumé)

---

- **Format :**
  - événement (arguments) [conditions] / action ^événements provoqués
- **Déclenchement :**
  - par un événement (peut être nul).
    - Peut avoir des arguments.
  - Conditionné par des expressions booléennes sur l'objet courant, l'événement, ou d'autre objets.
- **Tir de la transition :**
  - Exécute certaines actions instantanément.
  - Provoque d'autres événements ; globaux ou vers des objets cibles.

# Machine à états de protocole

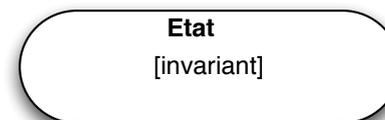
- Spécialisation des machines à états de comportement
- Toujours attachées à un classificateur (e.g., une classe)
- Représente un cycle de vie d'un objet et spécifient quels messages sont acceptés à chaque état
- Notation similaire aux machines à états comportementales, mais noté {protocol}



- Transitions : Spécifient une pré et une post condition



- Etats : Spécifient un invariant



# Behavioral View

---

## Activity Diagram

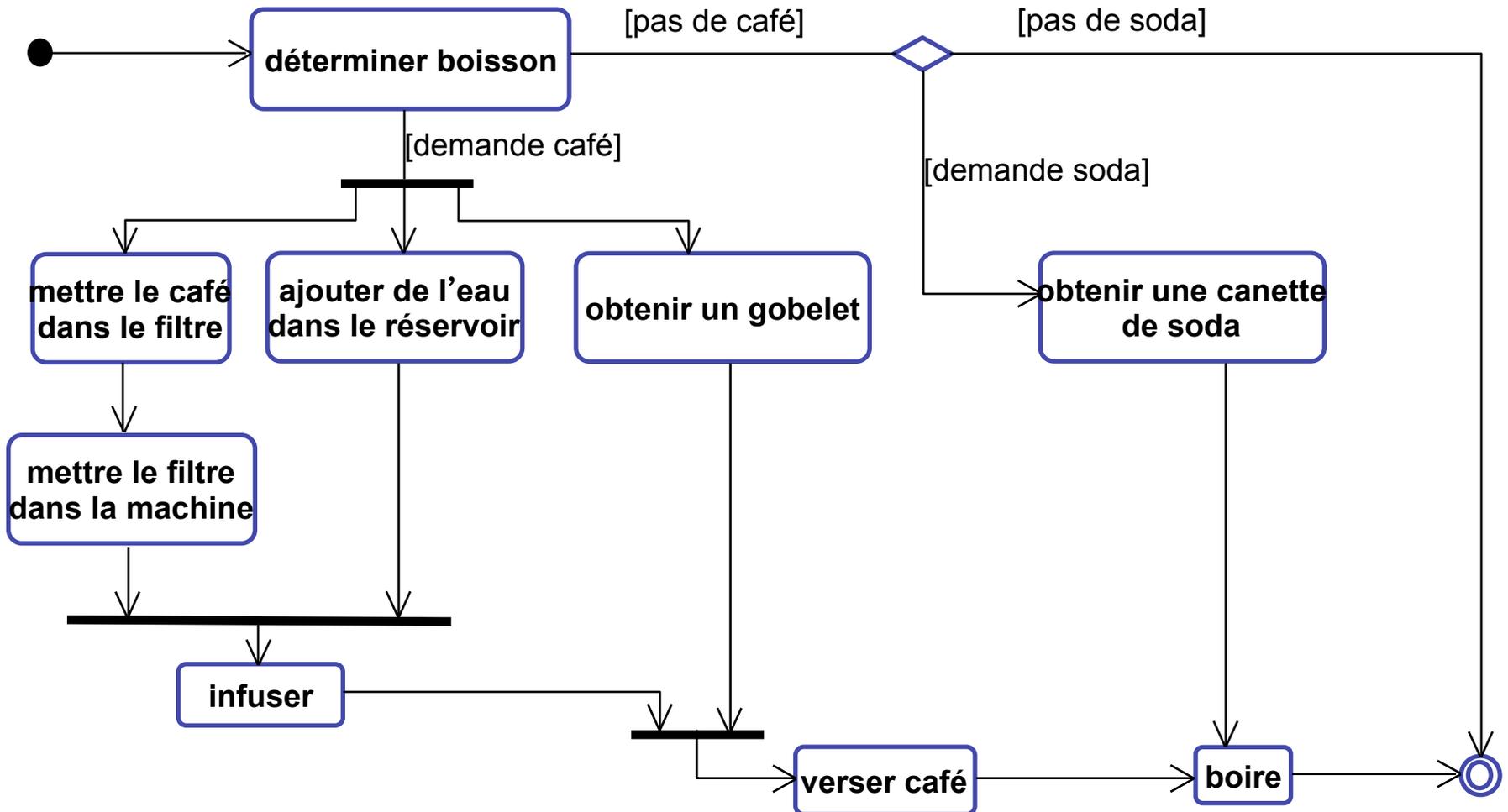
# Diagramme d'activité

---

- Spécifie la séquence et les conditions pour coordonner différents comportements (plutôt que les classificateurs contenant ces comportements)
- Composés d'un ensemble d'actions et de flots (de contrôle et de données)
- Spécifient une opération (conception)
- Peuvent spécifier un workflow
- Sémantique basée sur les réseaux de Petri
- Attachés à une classe, une opération, ou un *use-case (workflow)*

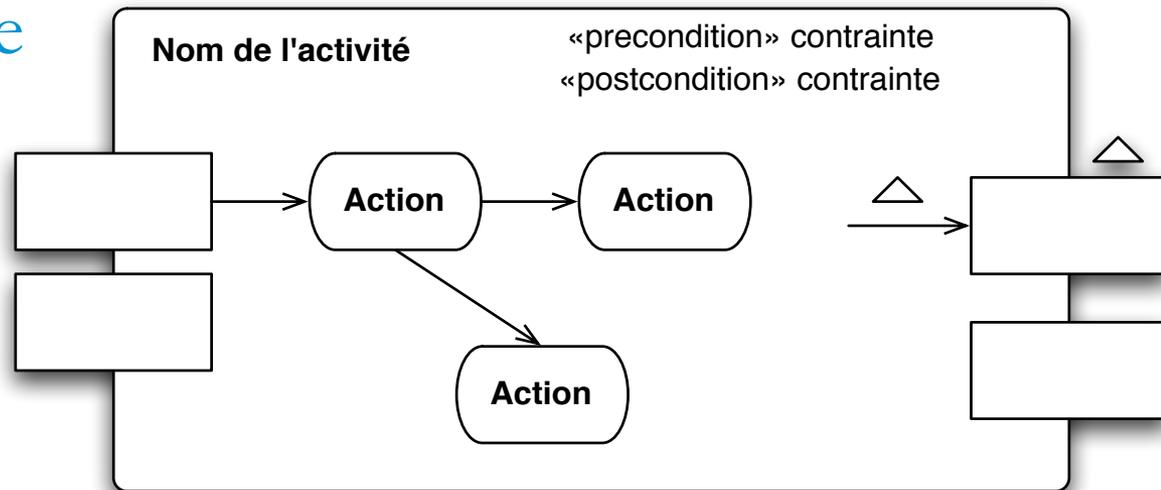
# Diagramme d'activité : exemple

## opération *PréparerBoisson* de la classe *Personne*



# Diagramme d'activité : notation

- Paramètres d'entrée et de sortie
- Pré et post-conditions
- Exceptions



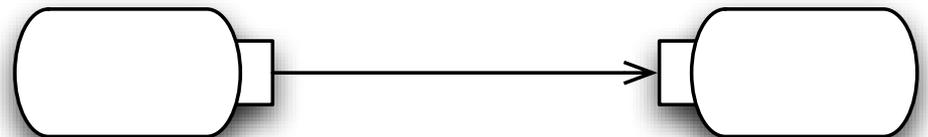
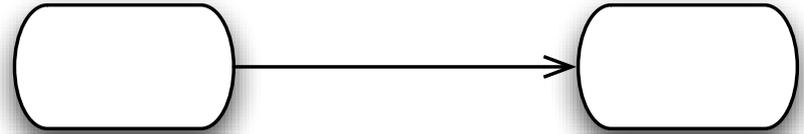
{activity} Activité
Attribut : type
Attribut : type
Operation(param)
Operation(param)

# Diagramme d'activité : notation

---

- Flots :

- De contrôle:  
déclenche une action une fois que l'action précédente est finie
- D'objet: chemin par lequel peuvent passer des objets et des données



# Diagramme d'activité : notation

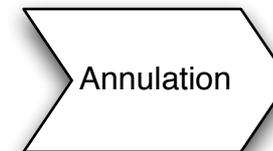
---

- **Objet :**

- Instance d'un classificateur, potentiellement dans un état particulier
- Événements



{ordering = LIFO}



# Stéréotypes optionnels

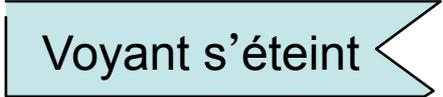
---

- Emission de signal



Allumer cafetière

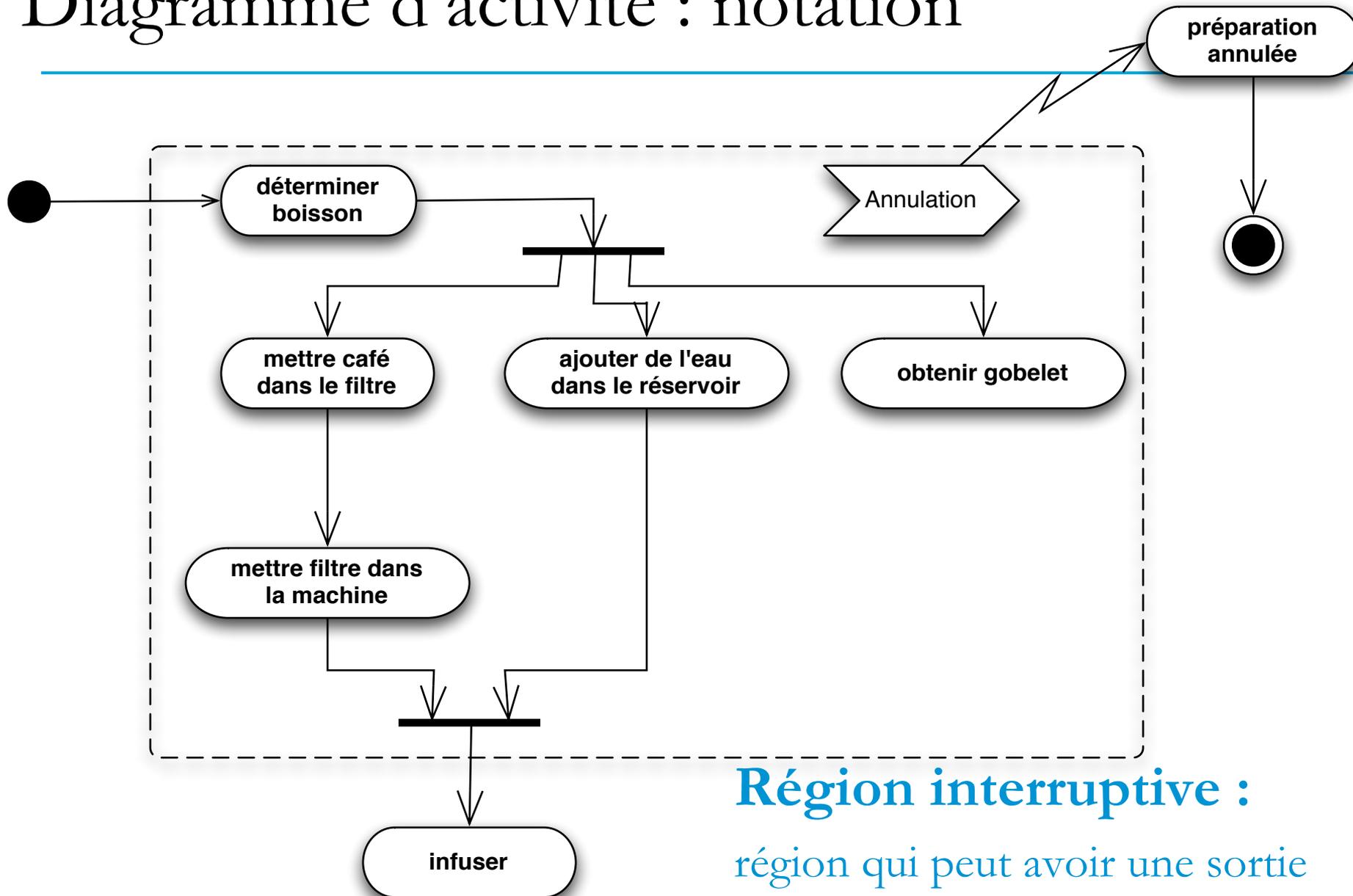
- Réception de signal



Voyant s'éteint

- On obtient une syntaxe graphique proche de SDL
  - langage de description de spécifications
  - populaire dans le monde télécom

# Diagramme d'activité : notation

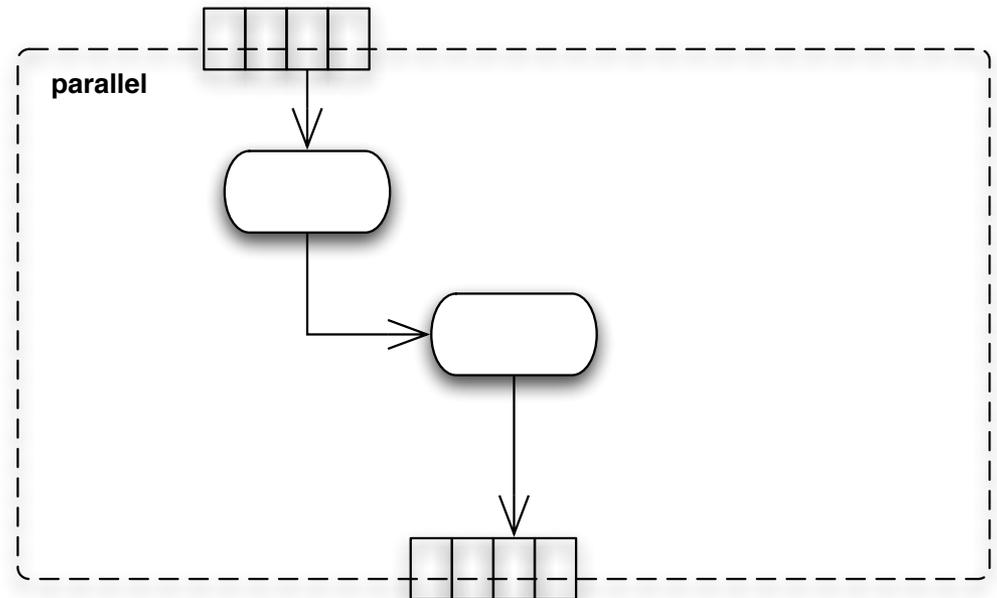


**Région interruptive :**  
région qui peut avoir une sortie  
alternative au flot normal

# Diagramme d'activité : notation

---

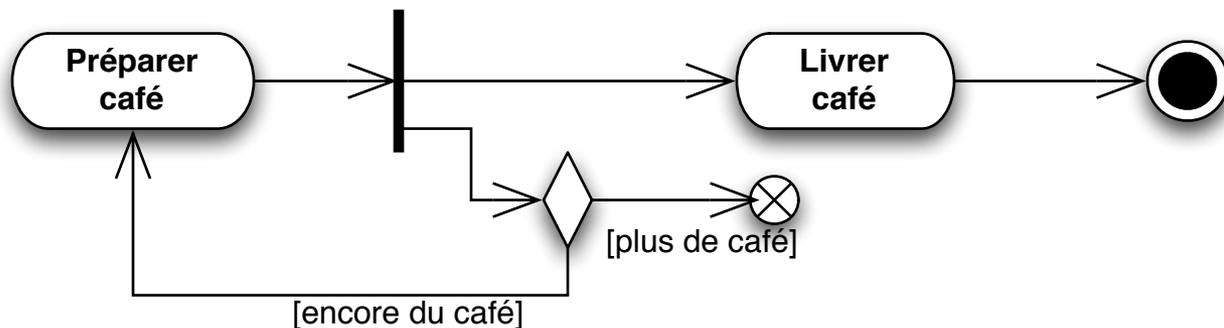
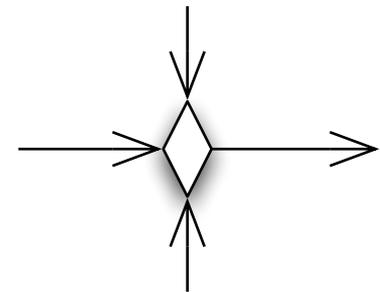
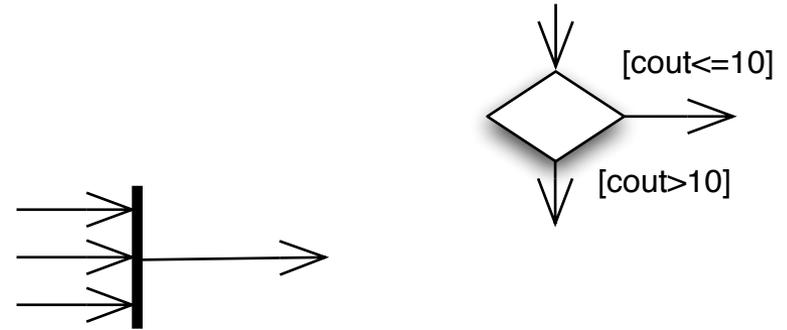
- Région d'expansion :
  - Région qui s'exécute plusieurs fois, selon les éléments de la collection passée en entrée
  - Types d'exécution: parallèle, itérative ou stream



# Diagramme d'activité : notation

- **Autres nœuds :**

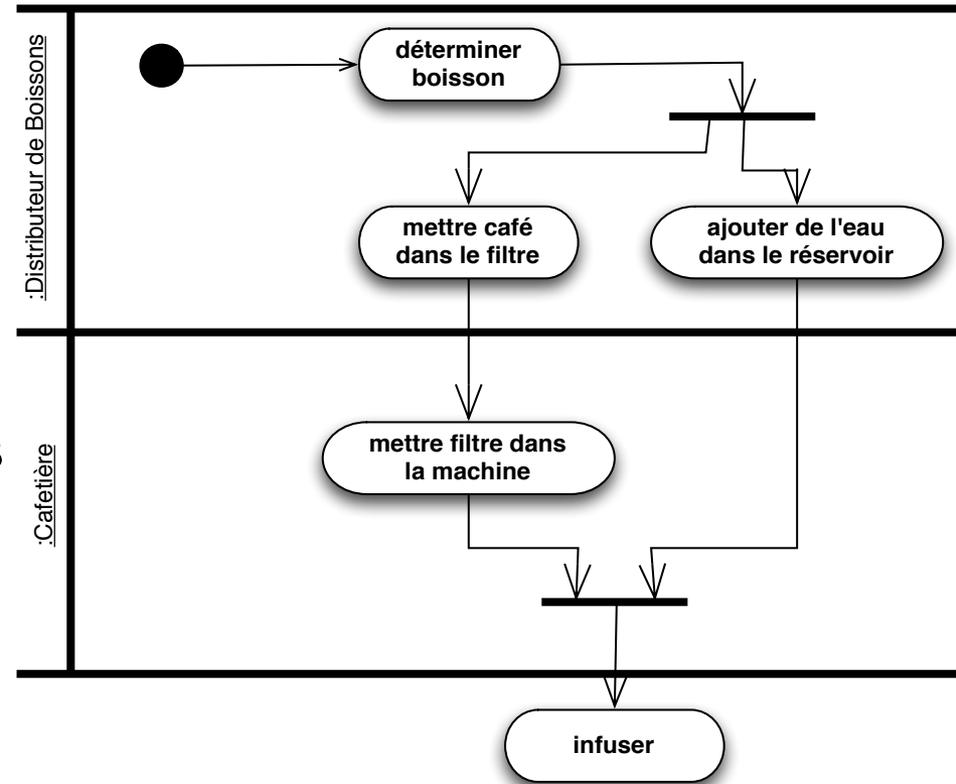
- Fin d'activité, fin de flot
- Fourchette
- Jointure
- Décision: branchement sur plusieurs transitions
- Fusion: accepte un flot parmi plusieurs



# Diagramme d'activité : notation

- **Partition:**

- Groupement d'actions ayant des caractéristiques communes
- Utilisées pour allouer des actions dans des ressources différentes (Classificateurs, Nœuds)



# Liens modèles statiques/dynamiques

---

- Le modèle dynamique définit des séquences de transformation pour les objets
  - Diagramme d'état généralisant pour chaque classe ayant un comportement réactif aux événements les scénarios et collaborations de leurs instances
    - Les variables d'état sont des attributs de l'objet courant
    - Les conditions de déclenchement et les paramètres des actions exploitent les variables d'état et les objets accessibles
  - Diagrammes d'activités associés aux opérations/transitions/méthodes
- Les modèles dynamiques d'une classe sont transmis par héritage aux sous-classes

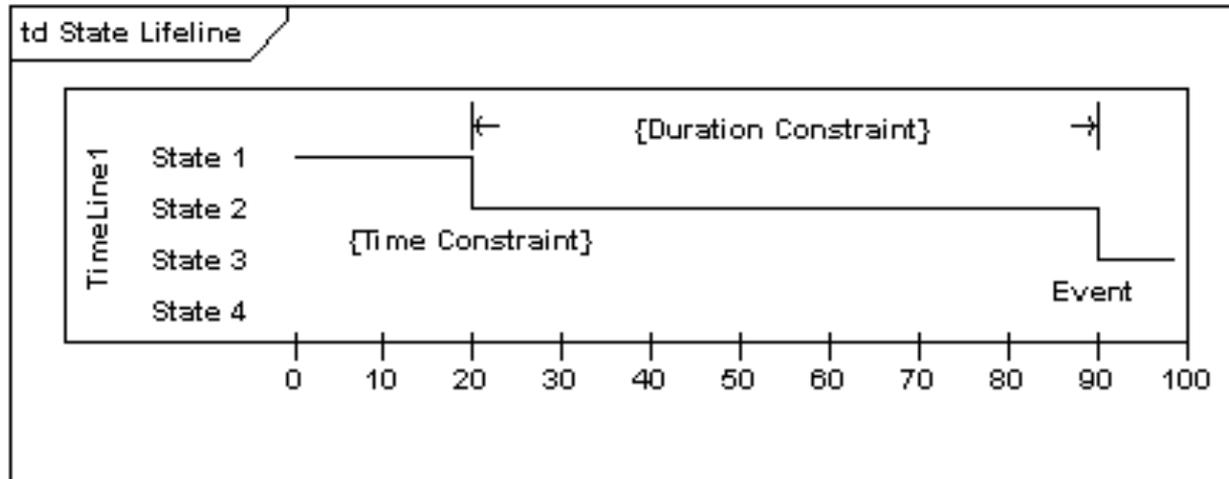
# Behavioral View

---

## Timing Diagram

# Diagramme temporel

Autre vue comportementale semblable aux diagrammes de séquence, mais présentée avec une syntaxe temporelle inspirée des diagrammes utilisés en circuits logiques



# Outline

---

① UML History and Overview

② UML Language

① UML Functional View

② UML Structural View

③ UML Behavioral View

④ UML Implementation View

⑤ UML Extension Mechanisms

③ UML Internals

④ UML Tools

⑤ Conclusion

# Implementation View

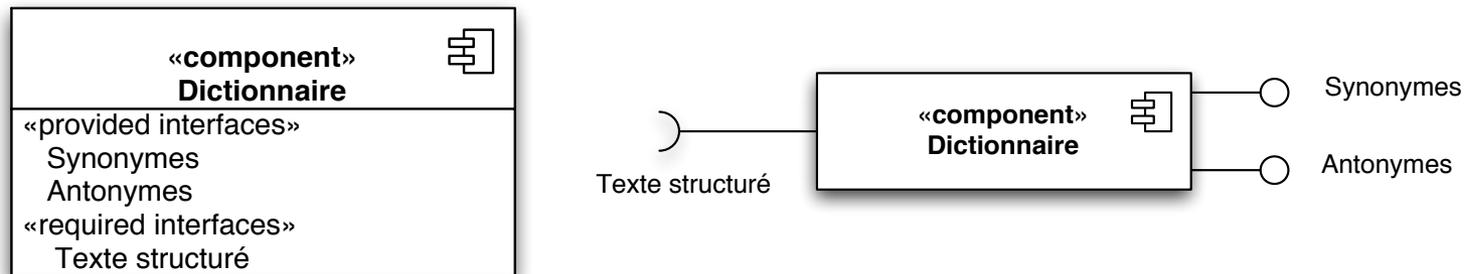
---

## Component Diagram

# Constituants d'un diagramme de composants

- **Composant**

- Partie remplaçable d'un système
- Son comportement est spécifié par des interfaces requises et fournies

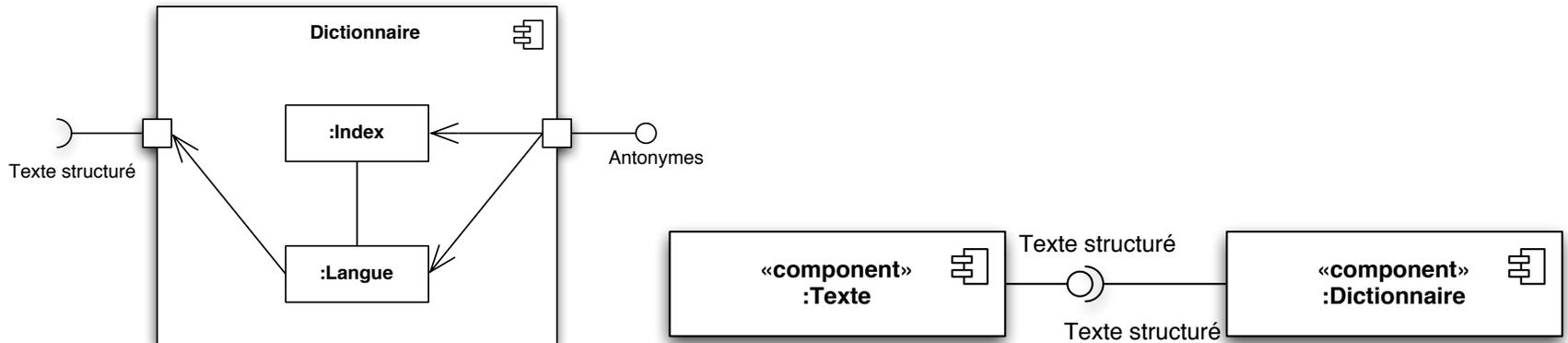


- **Interface (fournies/requise)**

# Constituants d'un diagramme de composants

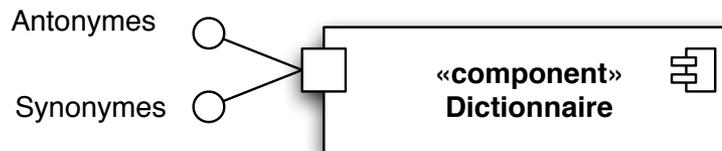
## • Connecteur

- Liaison entre un contrat externe (port) et la réalisation
- Assemblage, délégation



## • Port

- Point d'interaction entre un composant et son environnement
- La nature de ces interactions est spécifiée par des interfaces



# Implementation View

---

## Deployment Diagram

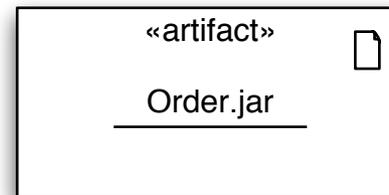
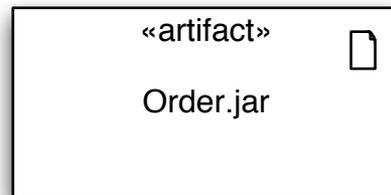
# Constituants d'un diagramme de déploiement

---

- **Artefact**

- Spécification d'une pièce physique d'information utilisée dans le processus de développement
- Fichiers source, modèles, scripts, binaires, document, etc.
- Un artefact est un classificateur: il possède des propriétés et des opérations
- Il peut s'associer avec d'autres artefacts

- Notation :

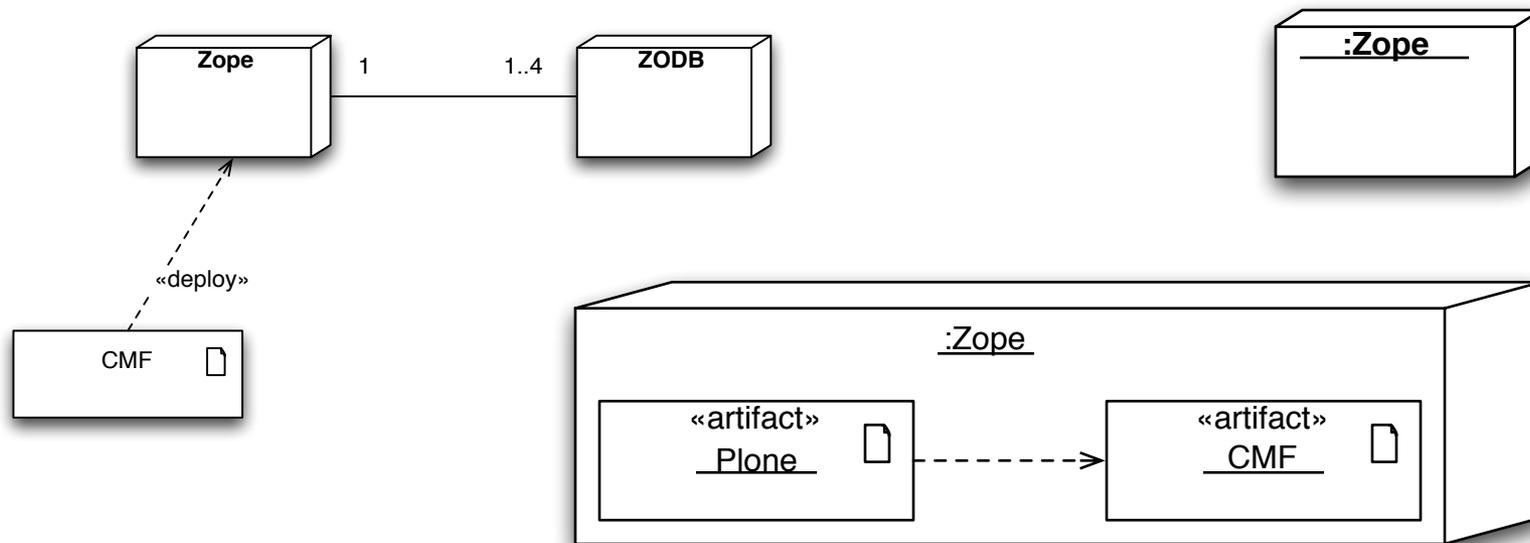


- Stéréotypes standards : « source », « executable »

# Constituants d'un diagramme de déploiement

- Nœud

- Ressource logique dans laquelle sont déployées les artefacts
- Notation : connexion, instances



# Constituants d'un diagramme de composants

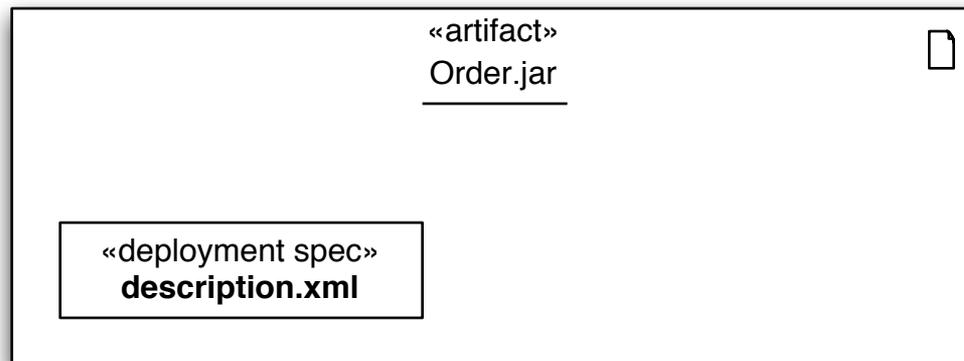
---

- **Chemin de communication**

- Association entre deux nœud, permettant l'échange de signaux et de messages

- **Spécification de déploiement**

- Ressource (artefact) déterminant les paramètres d'exécution d'un artefact



# Outline

---

① UML History and Overview

② UML Language

① UML Functional View

② UML Structural View

③ UML Behavioral View

④ UML Implementation View

⑤ UML Extension Mechanisms

③ UML Internals

④ UML Tools

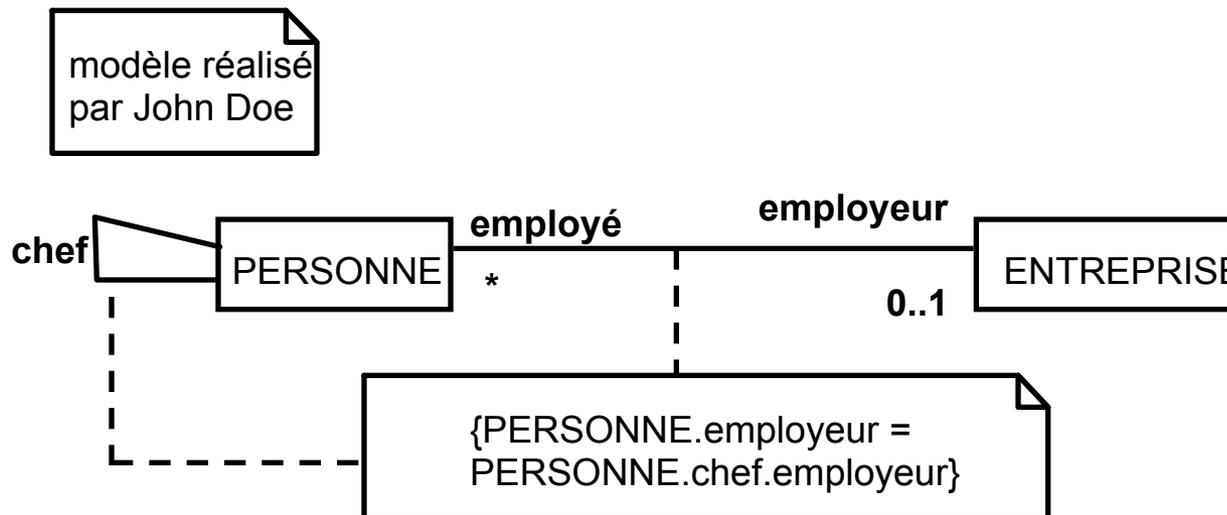
⑤ Conclusion

# Les notes

---

- Compléments de modélisation

- Attachés à un élément du modèle ou libre dans un diagramme
- Exprimés sous forme textuelle
- Elles peuvent être typées par des stéréotypes



# Extension d'UML : motivation

---

- Malgré sa richesse, UML n'est pas adapté à tous les domaines.
- Formes d'extension:
  - Ajout de nouveaux éléments.
  - Création de nouvelles propriétés.
  - Spécification d'une nouvelle sémantique.
- Mécanismes disponibles:
  - Stéréotypes, Étiquettes, Contraintes.

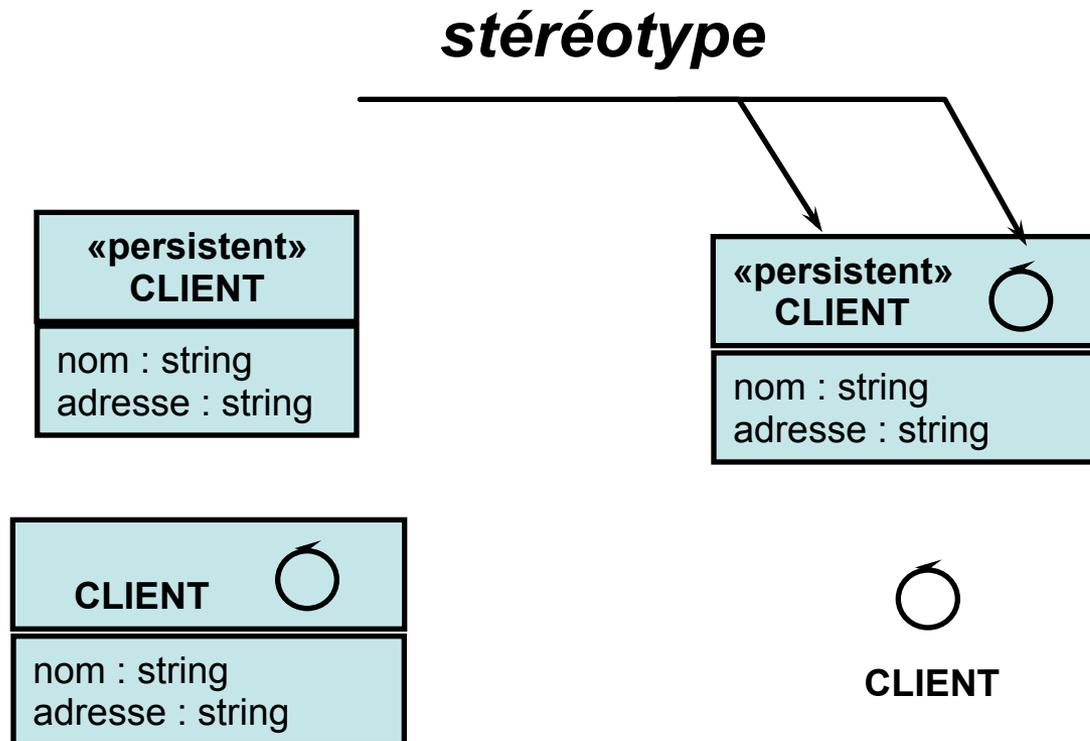
⇒ Profils.

# Les stéréotypes

---

- Nouveaux éléments de modélisation instanciant
  - Des classes du méta modèle UML (pour les stéréotypes de base UML)
  - Des extensions de classes du méta modèle UML (pour les stéréotypes définis par l'utilisateur)
- Peuvent être attachés aux éléments de modélisations et aux diagrammes :
  - Classes, objets, opérations, attributs, généralisations, relations, acteurs, uses-cases, événements, diagrammes de collaboration ...

# Les stéréotypes : notation



- Peuvent introduire:
  - une nouvelle notation
  - des contraintes d'utilisation
- Très utiles pour l'interprétation d'un modèle

# Étiquettes (tagged values)

---

- Pair de la forme “étiquette=valeur”, attachée aux éléments de modélisation

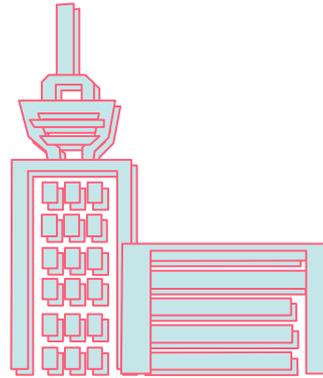
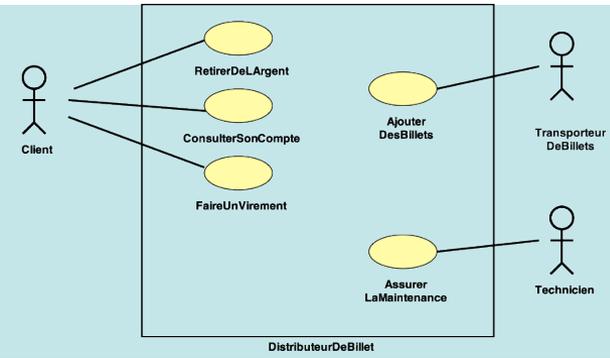
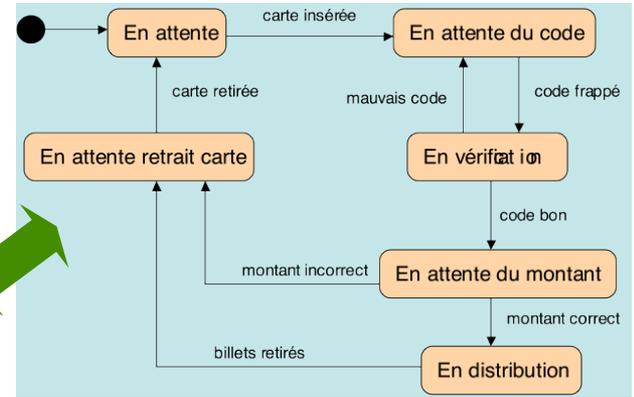
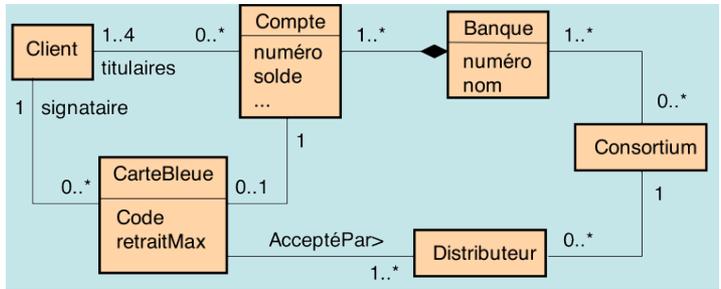
<b>Personne</b> {author = G. Sunyé, status = incomplete}
nom : String secu : Integer naissance : Date

# Les profils UML

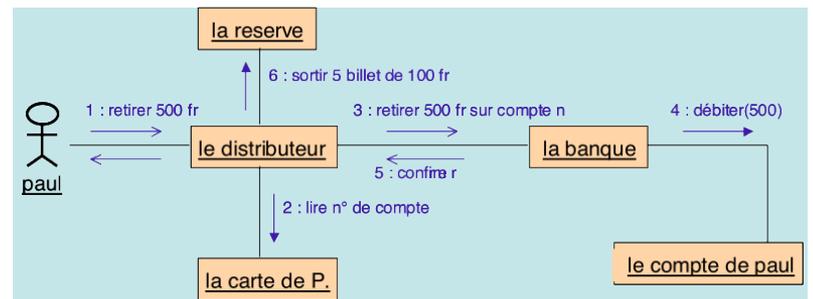
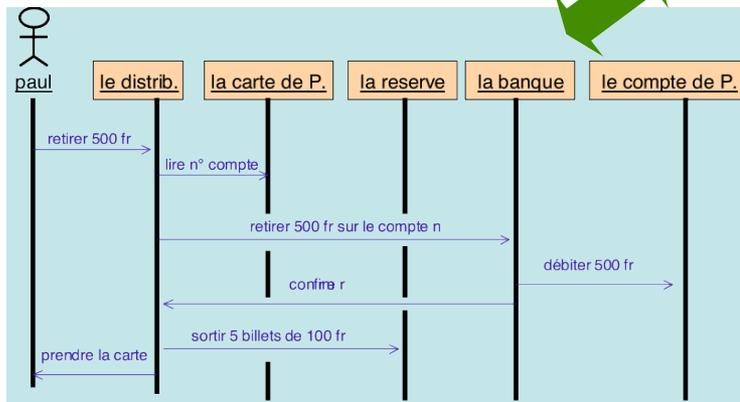
---

- Mécanisme d'extension, permettant d'adapter UML à l'aide d'un ensemble de stéréotypes
- Le profil “Standard” contient les stéréotypes utilisés dans UML
- Autres profils: J2EE/EJB, COM, .NET, CCM, etc.

# Troisième partie (UML / méthode)



système



# UML 2.x : Structural Modeling Diagrams

---

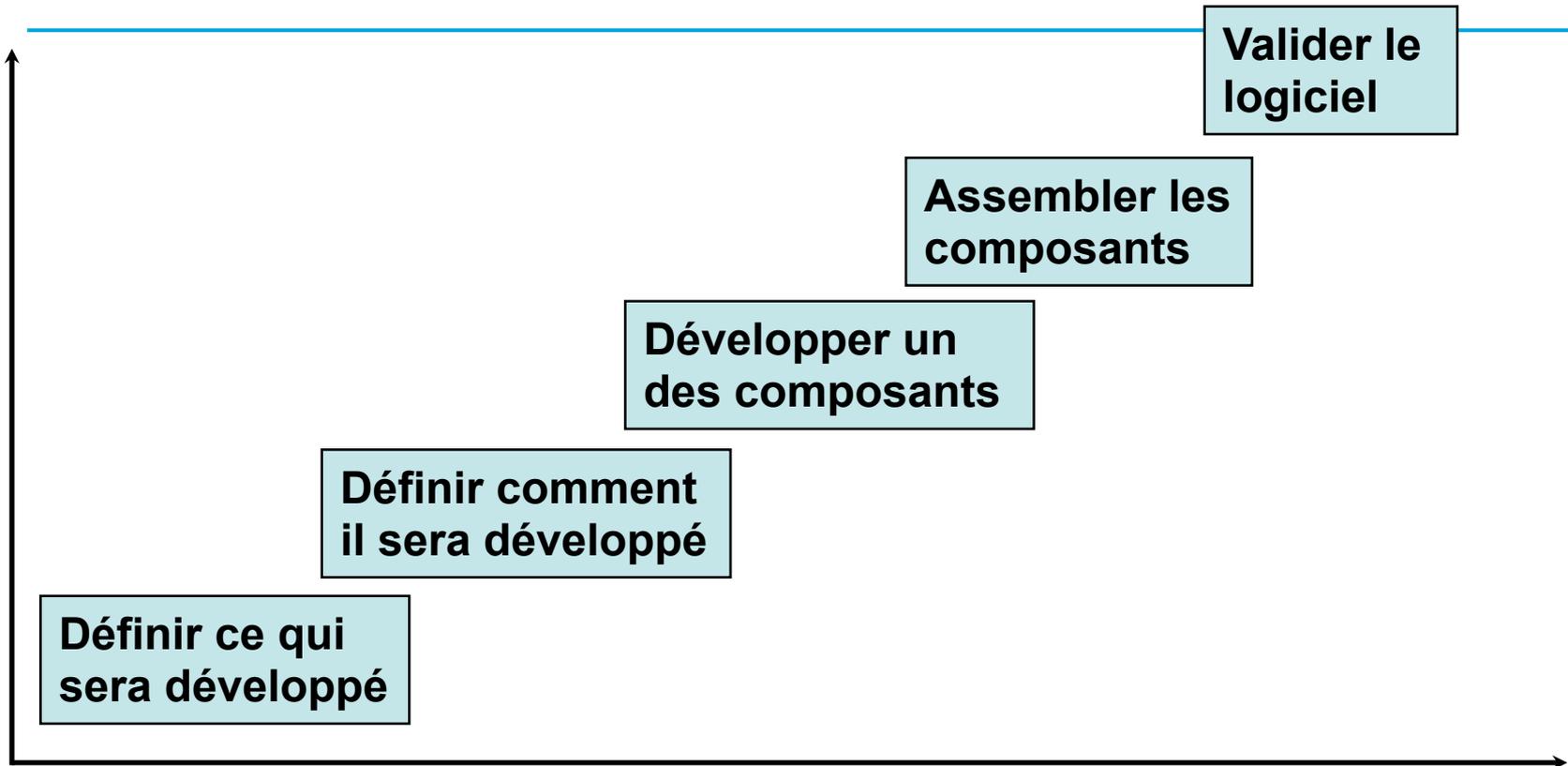
- Structure diagrams define the static architecture of a model. They are used to model the 'things' that make up a model - the classes, objects, interfaces and physical components. In addition they are used to model the relationships and dependencies between elements.
  - Package diagrams are used to divide the model into logical containers or 'packages' and describe the interactions between them at a high level
  - Class or Structural diagrams define the basic building blocks of a model: the types, classes and general materials that are used to construct a full model
  - Object diagrams show how instances of structural elements are related and used at run-time.
  - Composite Structure diagrams provide a means of layering an element's structure and focusing on inner detail, construction and relationships
  - Component diagrams are used to model higher level or more complex structures, usually built up from one or more classes, and providing a well defined interface
  - Deployment diagrams show the physical disposition of significant artefacts within a real-world setting.

# UML 2.x : Behavioral Modeling Diagrams

---

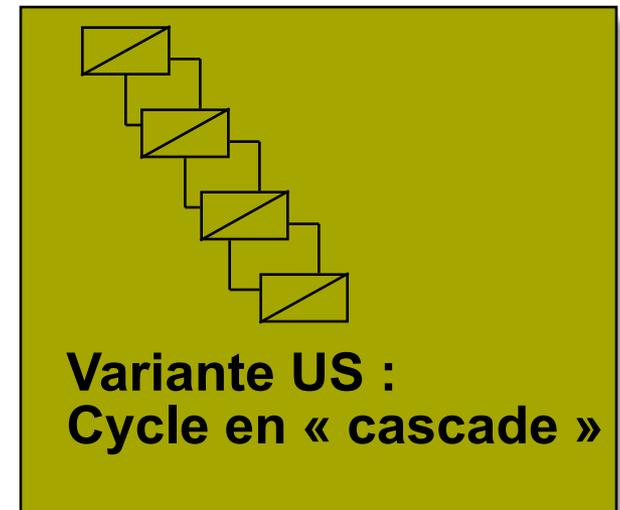
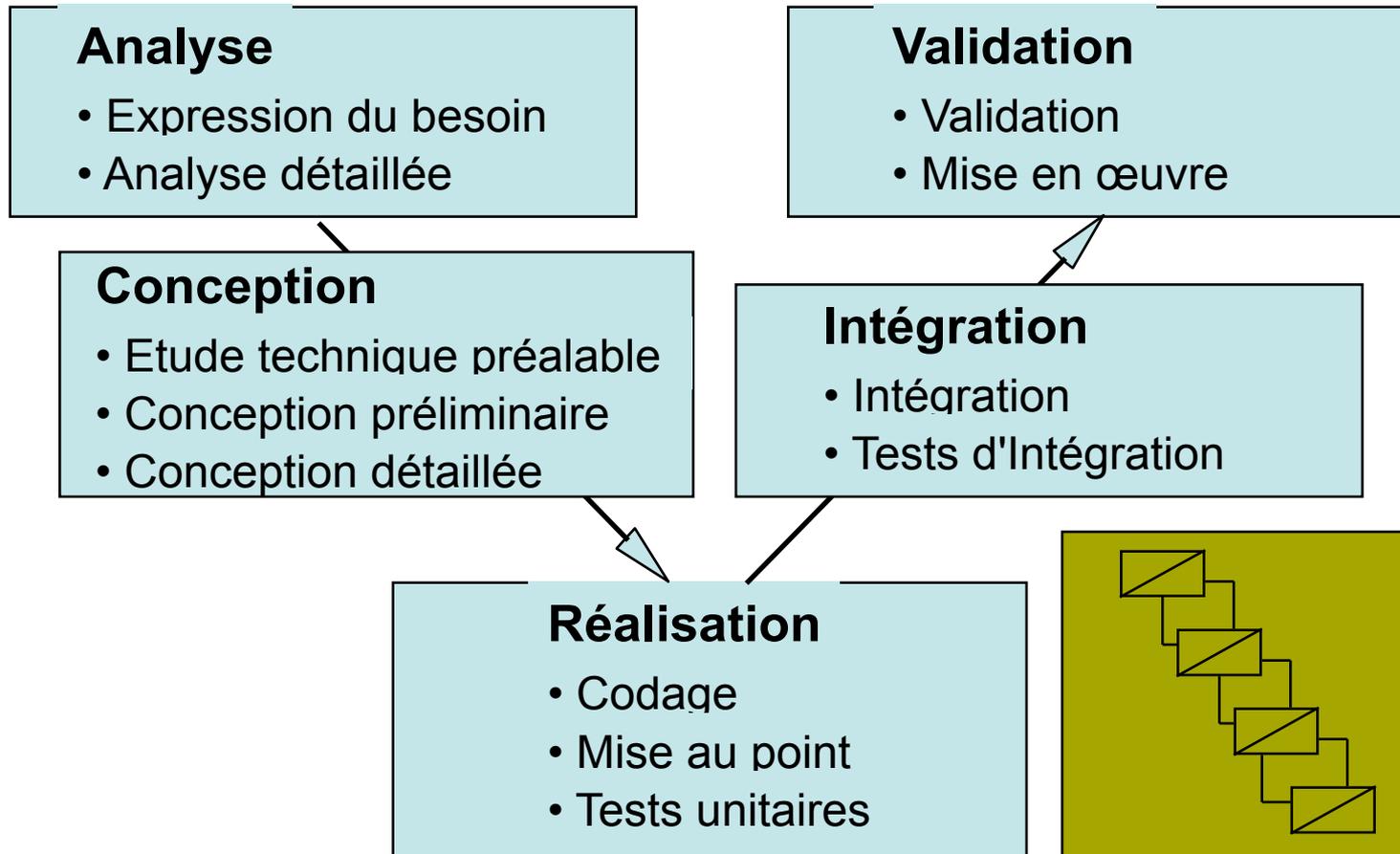
- Behavior diagrams capture the varieties of interaction and instantaneous state within a model as it 'executes' over time.
  - Use Case diagrams are used to model user/system interactions. They define behavior, requirements and constraints in the form of scripts or scenarios
  - Activity diagrams have a wide number of uses, from defining basic program flow, to capturing the decision points and actions within any generalized process
  - State Machine diagrams are essential to understanding the instant to instant condition or "run state" of a model when it executes
  - Communication diagrams show the network and sequence of messages or communications between objects at run-time during a collaboration instance
  - Sequence diagrams are closely related to Communication diagrams and show the sequence of messages passed between objects using a vertical timeline
  - Timing diagrams fuse Sequence and State diagrams to provide a view of an object's state over time and messages which modify that state
  - Interaction Overview diagrams fuse Activity and Sequence diagrams to provide allow interaction fragments to be easily combined with decision points and flows

# Activités du développement de logiciels

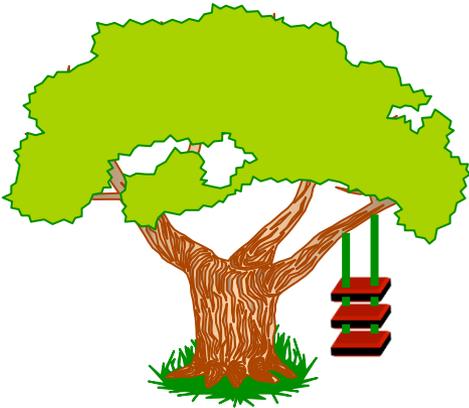


- L'organisation de ces activités et leur enchaînement définit le *cycle de développement* du logiciel

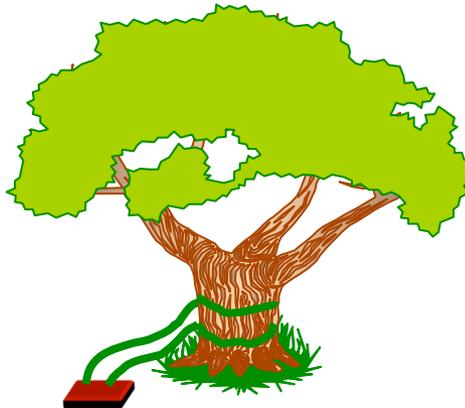
# Cycle de vie en V normalisé AFNOR



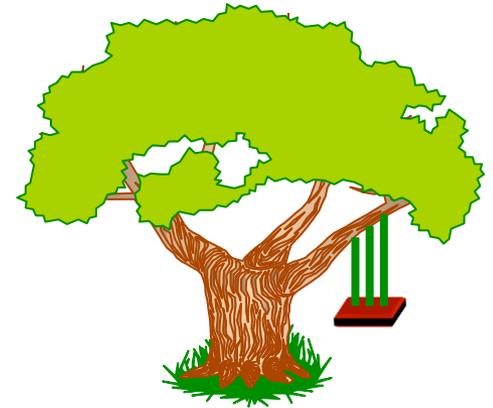
# Problèmes avec le processus classique...



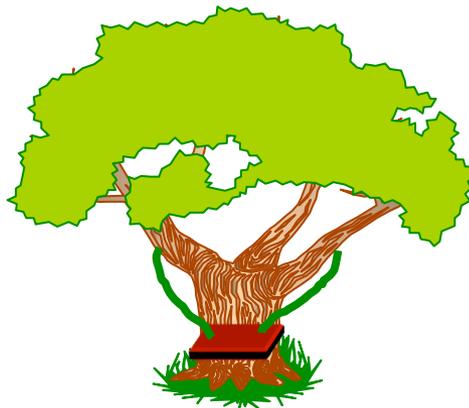
Ce que demande l'utilisateur



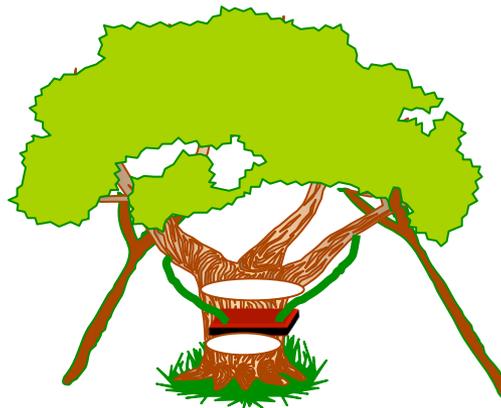
Ce que l'analyste a spécifié



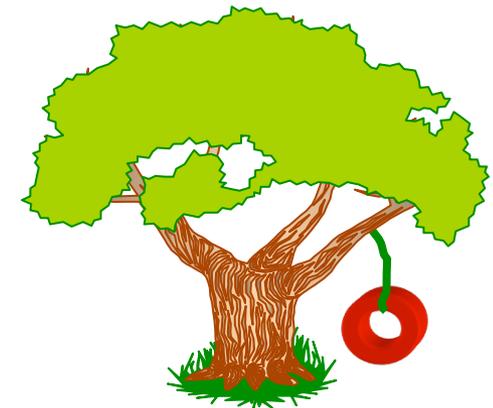
Ce que prévoit le concepteur



Ce que le programmeur a écrit



Ce que la mise au point a fait



Ce que l'utilisateur n'a pas su exprimer

# Why do projects fail so often

---

- Unrealistic or unarticulated project goals
- Inaccurate estimates of needed resources
- Badly defined system requirements
- Poor reporting of the project's status
- Unmanaged risks
- Poor communication among customers, developers, and users
- Use of immature technology
- Inability to handle the project's complexity
- Sloppy development practices
- Poor project management
- Stakeholder politics
- Commercial pressures

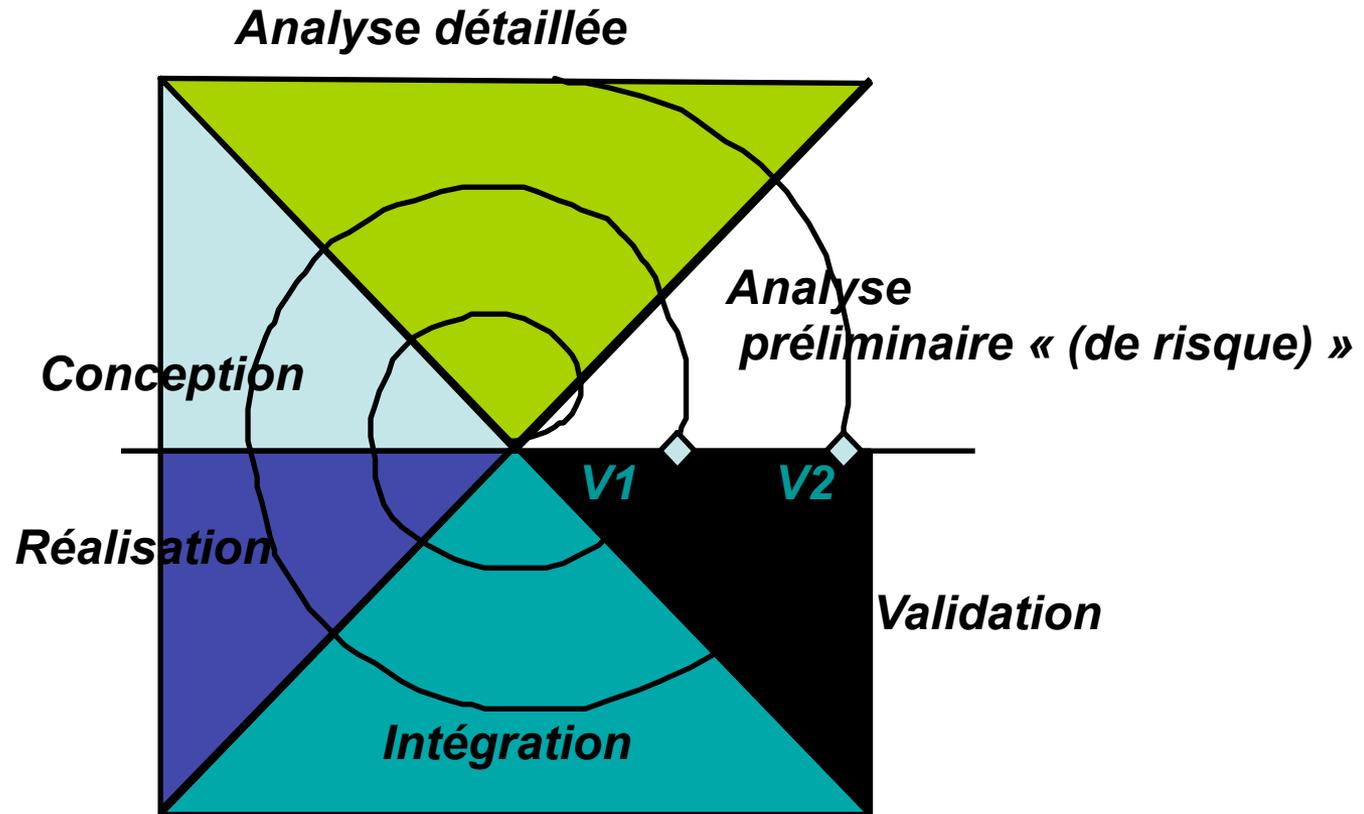
# Problèmes du processus classique

---

- Organisation « industrielle » héritée du XIX<sup>ème</sup> siècle
  - rassurant pour les managers
  - hiérarchie malsaine dans les rôles
  - antinomie : Coplien 's organizational pattern
    - *Architects Also Implement*
- cycle management <> cycle développement
- linéarité implicite
  - temps d 'approbation des documents => effet tampon
  - coût de la (non-) modification d 'un document « final »
  - irréaliste pour un projet innovant, donc à risques

# Cycle de vie en « spirale »

---



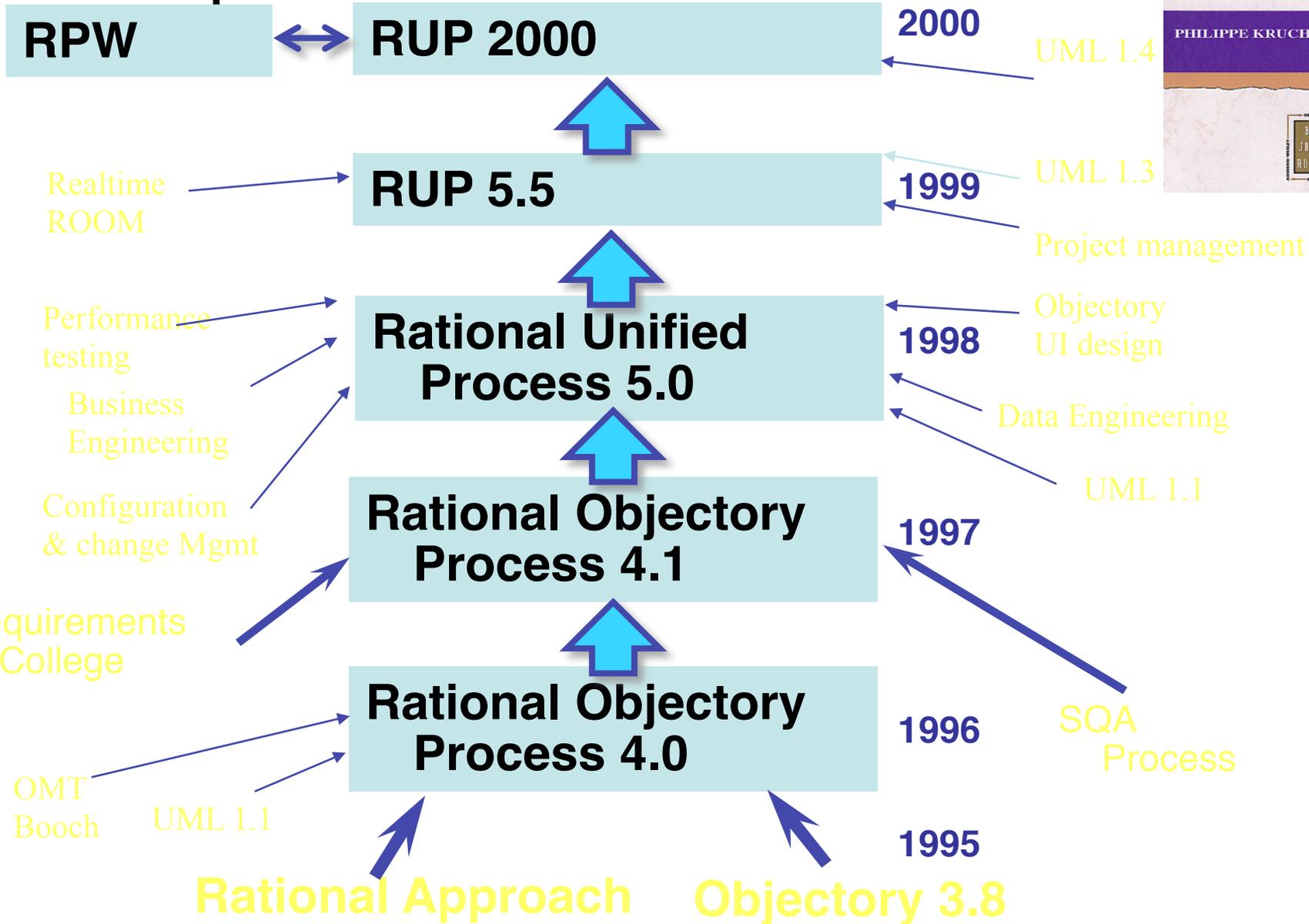
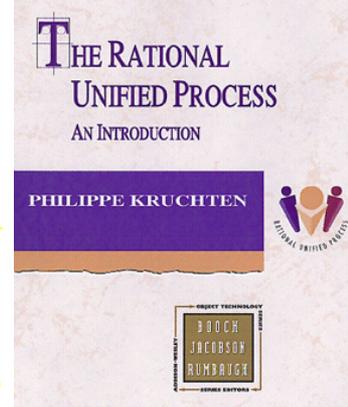
*Synergie avec approche par objets*

# Intérêts du cycle de vie en « spirale »

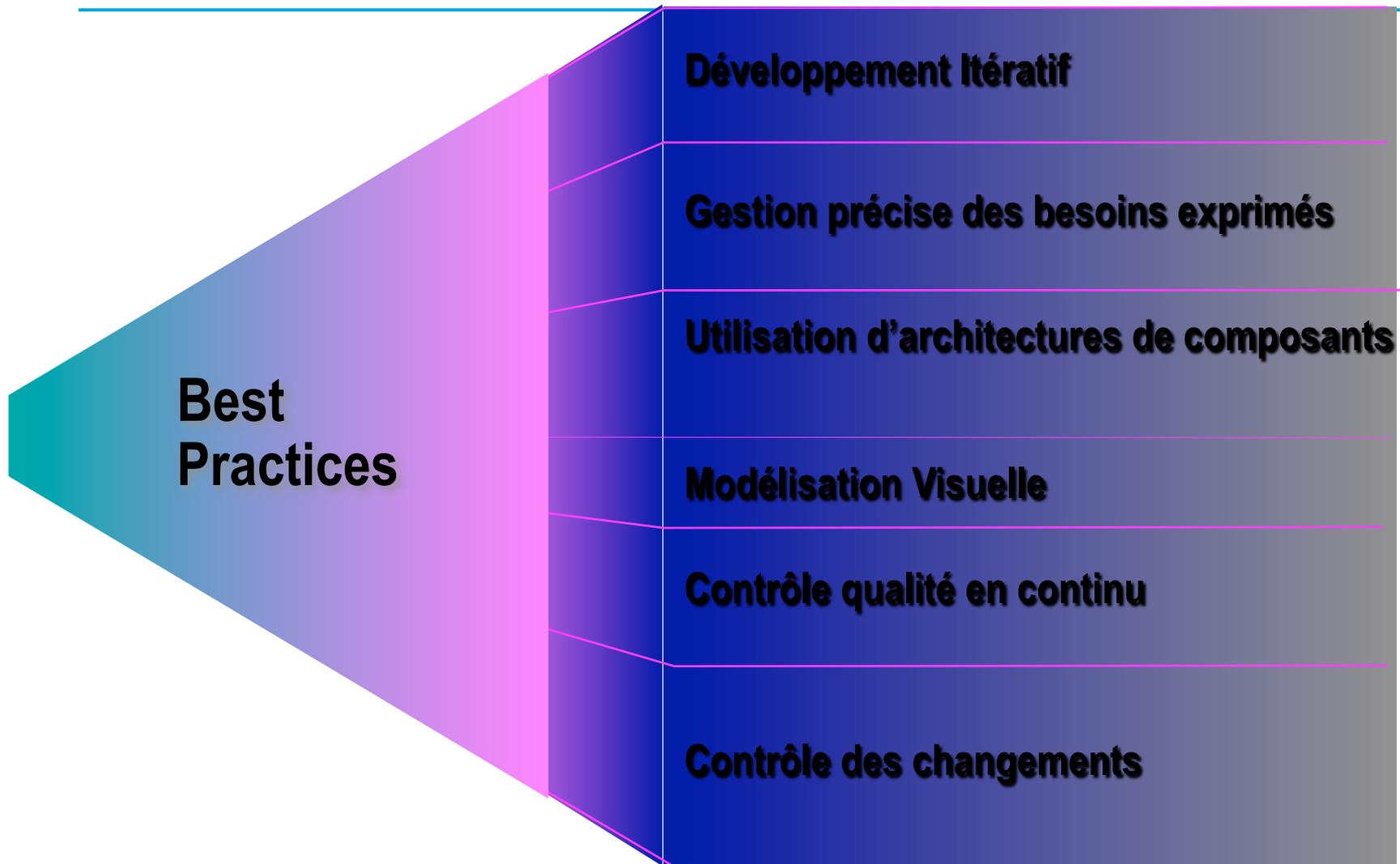
---

- **Bien adapté au développements innovants**
  - les progrès sont tangibles : c'est du logiciel qui « tourne » et pas seulement des kilos de documents
  - possibilité de s'arrêter « à temps », i.e. avant que l'irréalisabilité du projet ait créé un gouffre financier
- **Moins simple à manager**
  - difficile à gérer en situation contractuelle
  - mal contrôlé => on retombe dans le *hacking*
- **Production des incréments asservie sur 2 parmi 3 :**
  - période (e.g. release toutes les 2 semaines)
  - fonctionnalités (releases découpés suivant use-cases)
  - niveau de qualité (problème de la mesure)

# Exemple du RUP



# Principes du Rational Unified Process



# Point clés

---

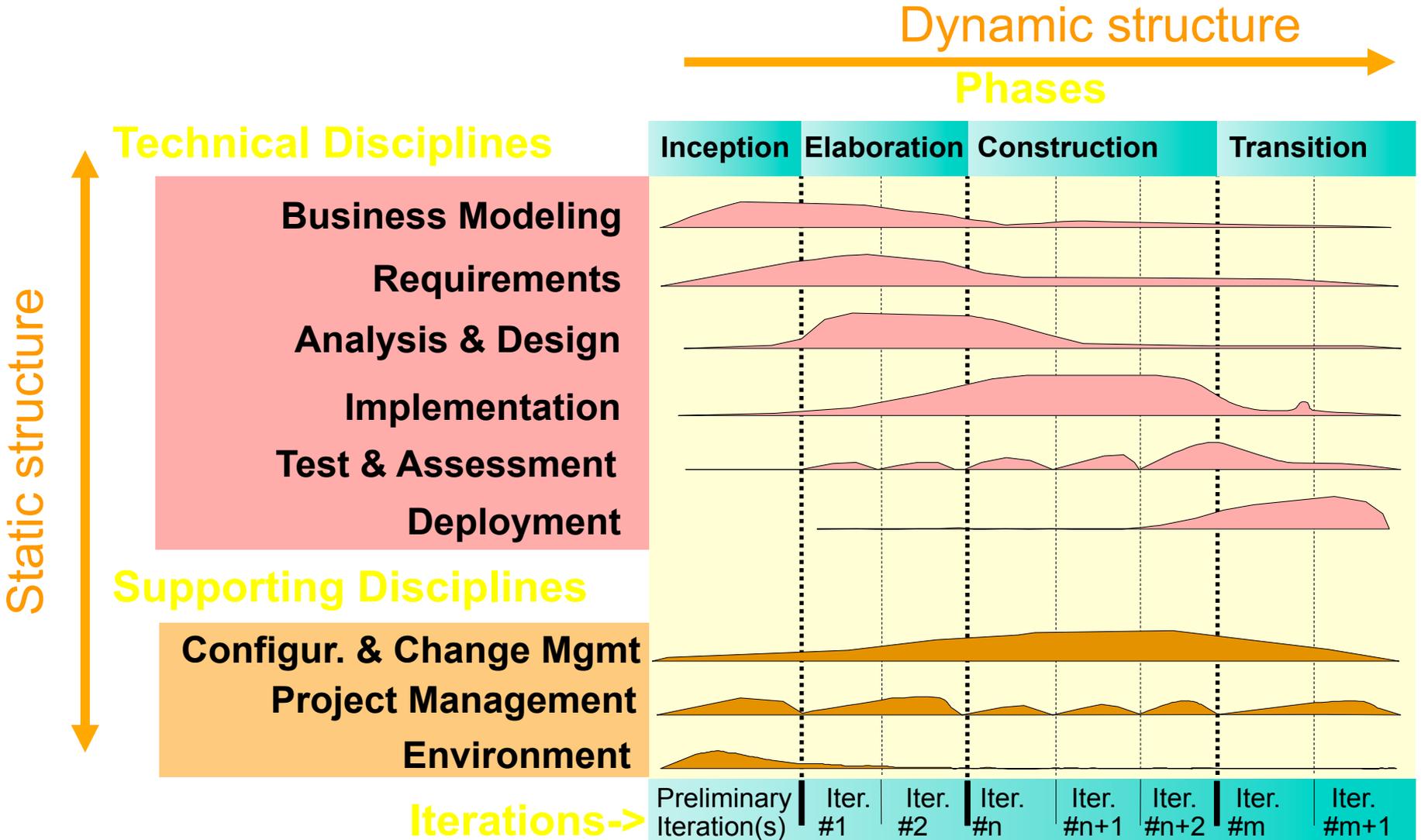
- Développer seulement ce qui est nécessaire
- Minimiser la paperasserie
- flexibilité
  - besoins, plan, utilisation des ressources, etc...
- Apprendre de ses erreurs précédentes
- Réévaluer les risques régulièrement
- Établir des critères de progrès
  - objectifs et mesurables
- Automatiser

# Architecture du processus

---

- 2 structures orthogonales
- Structure statique
  - Workers, artifacts, activities, workflows
  - authoring et configuration du processus
    - ☞ SPEMSi, ingénierie des méthodes et des processus
- Structure dynamique
  - Structure du cycle de vie : phases, itérations
  - Mise en oeuvre du processus : planification, exécution
    - ☞ gestion des activités, suivi de projet

# Les 2 dimensions du processus



# Phases du développement itératif



Inception: définition de la portée du projet

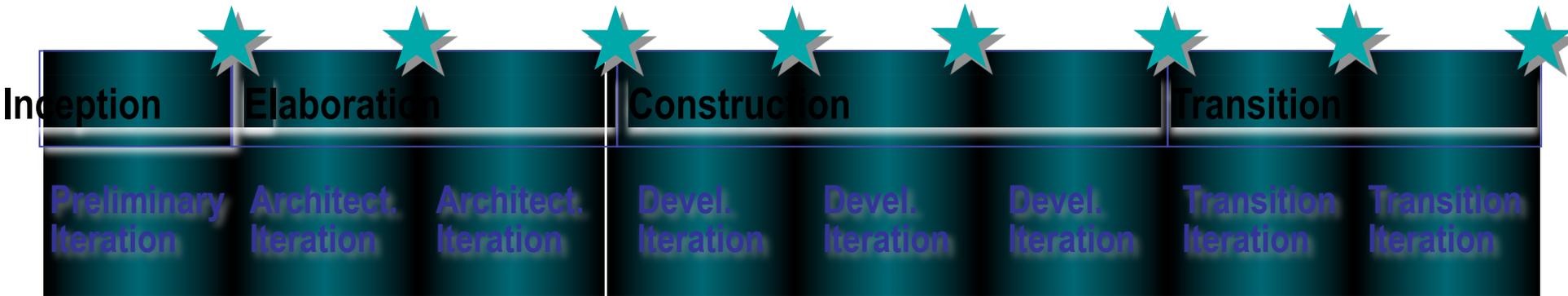
Elaboration: planification du projet, spécification des fonctionnalités, architecture de base

Construction: réalisation du produit

Transition: transfert du produit vers les utilisateurs

# Itérations

## Livraison d'exécutables



---

Une *itération* est une séquence d'activités avec un plan bien établi et un critère d'évaluation, résultant en la livraison d'un logiciel *exécutable*.

# Phases et itérations : 2 exemples

- **Petit projet de commerce électronique**

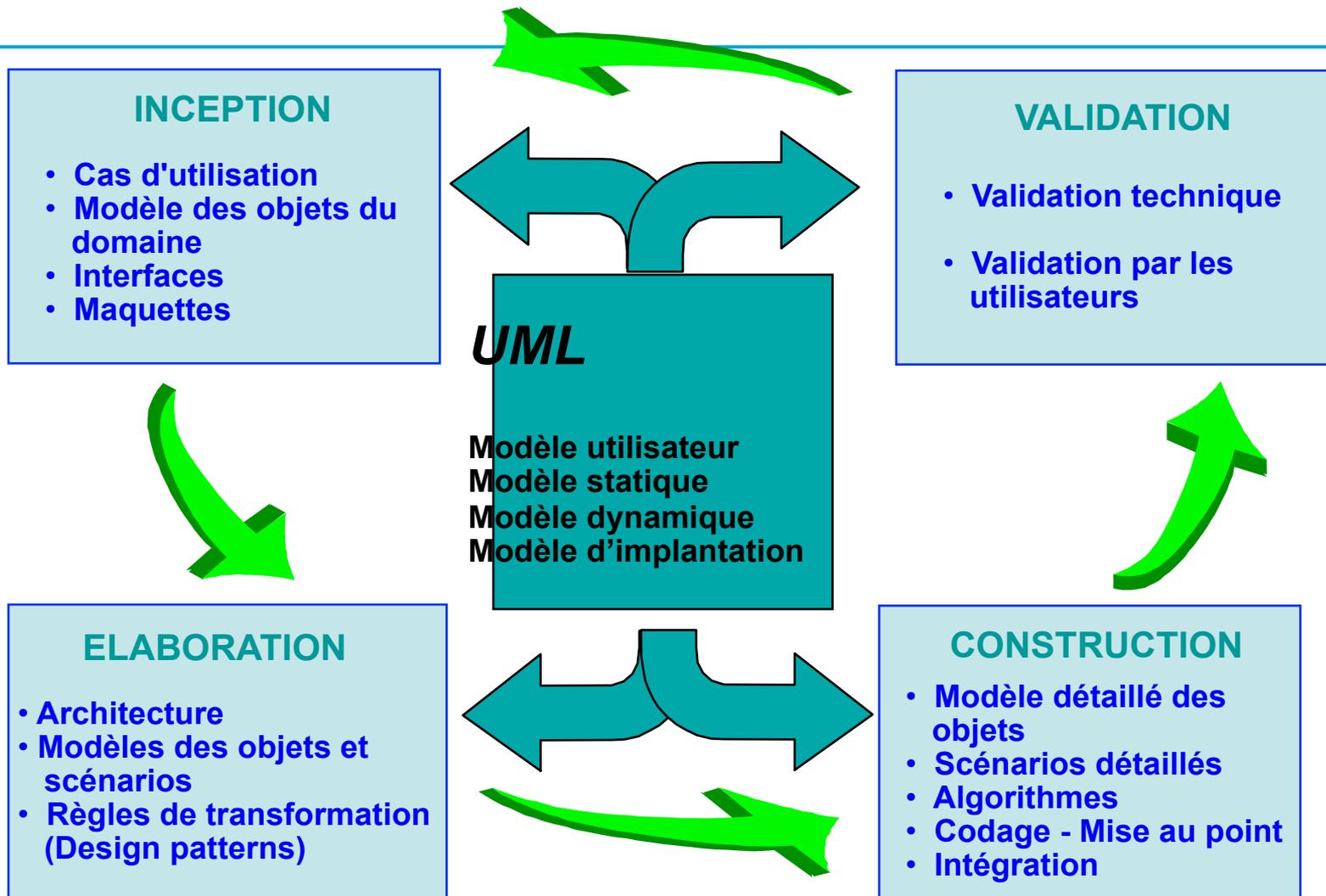
- Intégration à un mainframe
- 5 personnes

- **Grand projet d'infrastructure**

- Gros travail d'architecture nécessaire
- 20 personnes

	No. of Iterations				Project Length	Iteration Length
	Inception	Elaboration	Construction	Transition		
e-business	0.2	1	3	1	3-4 months	2-3 weeks
infrastructure	1	3	3	2	9-12 months	5-7 weeks

# Vision «générique» d'un cycle UML



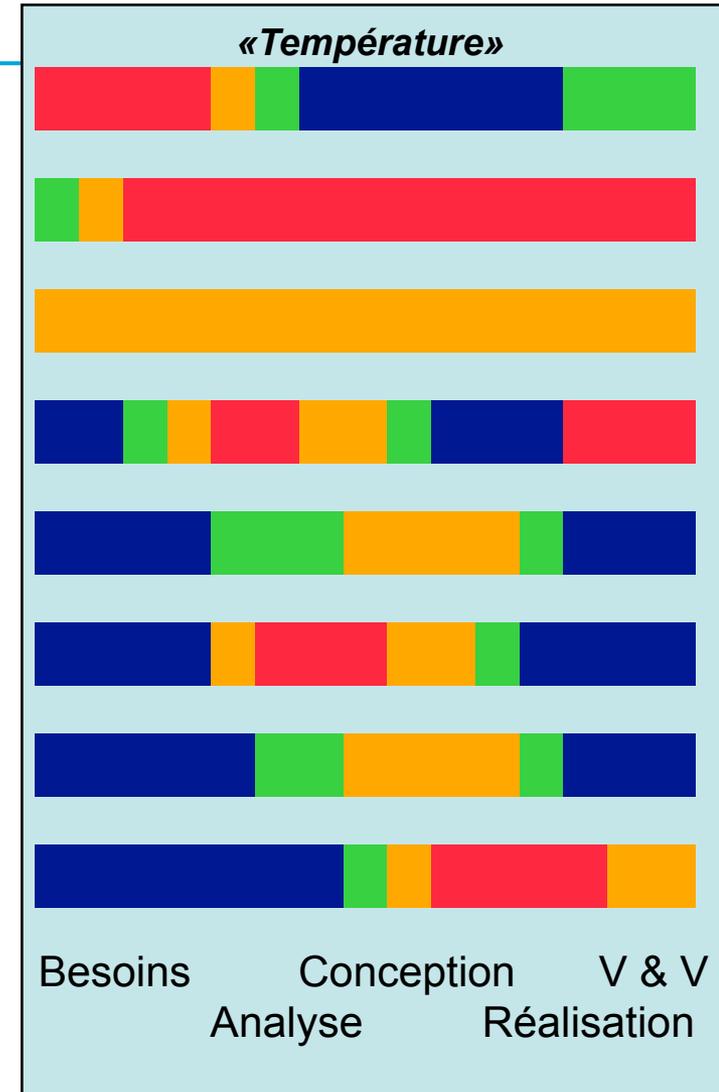
# Processus de développement avec UML

---

- Approche itérative, incrémentale, dirigée par les cas d'utilisation
  - Expression des besoins
  - Analyse
    - Elaboration d'un modèle « idéal »
  - Conception
    - passage du modèle idéal au monde réel
  - Réalisation et Validation

# Température des diagrammes UML

- Diagramme de cas d'utilisations
- Diagramme de classes
- Diagramme de paquetages
- Diagramme de séquences
- Diagramme de collaborations
- Diagramme d'états-transitions
- Diagramme d'activités
- Diagramme d'implantation



# Objectif d'une analyse avec UML

---

- Construction d'un modèle d'analyse
- Modèle unique mais quadri-dimensionnel
  - Aspects utilisation du système
  - Aspects statiques du système
  - Aspects dynamiques du système (comportemental)
  - Aspects déploiement et implantation

# Construction d'un modèle d'analyse

---

- Hypothèse d'un monde idéal
  - processeur suffisamment rapide
  - mémoire suffisamment grande et rapide
  - communications rapides et fiables
  - pas de défaillances
- Sélection des aspects cruciaux d'un problème
- Pas de solution unique :
  - modélisation = interprétation
  - « *Quoiqu'on en dise, dans la vie scientifique, les problèmes ne se posent pas d'eux-mêmes. Rien ne va de soi. Rien n'est donné. Tout est construit.* » G. Bachelard

# Modélisation UML

---

- **Modélisation selon 4 points de vue principaux :**
  - **Vision utilisateur du système**
    - Cas d'utilisation
  - **Aspects statiques du système**
    - Description des données et de leurs relations
    - Structuration en paquetages
  - **Aspects dynamiques du système (comportemental)**
    - Diagramme de séquences (scénarios)
    - Diagramme de collaborations (entre objets)
    - Diagramme d'états-transitions (Harel)
    - Diagramme d'activités
  - **Vision implantation**
    - Diagramme de composants et de déploiement

# Etude de cas

- Un serveur de réunions virtuelles
  - Adaptation du concept d'IRC à un contexte de réunions de travail au sein d'une entreprise géographiquement dispersée
- Cette étude de cas va nous servir de fil conducteur pour décrire un processus d'analyse et de conception avec UML

# Cahier des charges (1/2)

---

- Il s'agit de réaliser la partie serveur d'une application client-serveur permettant de faire des réunions virtuelles multimédia sur Internet. L'objectif de cette application est de permettre d'imiter le plus possible le déroulement de réunions de travail classiques. Cependant, dans la première version de ce projet, les interventions des participants se feront en mode mono-média seulement (i.e. échanges en forme textuelle).
- Le serveur devra permettre de planifier et de gérer le déroulement de plusieurs réunions simultanées. Des programmes clients existeront dans l'avenir pour plusieurs plate-formes (Mac, Windows, Unix) afin de permettre à des personnes désirant organiser des réunions virtuelles ou y participer de dialoguer avec le serveur en utilisant un protocole ad hoc développé au dessus de IP.

# Cahier des charges (2/2)

---

- Après s'être connecté au serveur (à l'aide d'un nom de login et d'un mot de passe mémorisé par le système), une personne a la possibilité de planifier des réunions virtuelles (choix d'un nom, définition du sujet, date de début et durée prévue, ordre du jour), de consulter les détails d'organisation d'une réunion, de les modifier (seulement l'organisateur), d'ouvrir et de clôturer une réunion (seulement l'animateur), d'entrer (virtuellement) dans une réunion précédemment ouverte, et d'en sortir. En cours de réunion, un participant peut demander à prendre la parole. Quand elle lui est accordée, il peut entrer le texte d'une intervention qui sera transmise en ``temps-réel'' par le serveur à tous les participants de la réunion.
- Plusieurs sortes de réunions doivent pouvoir être organisables :
  - Réunions standards, avec un organisateur qui se charge de la planification de la réunion et désigne un animateur chargé de choisir les intervenants successifs parmi ceux qui demandent la parole.
  - Réunions privés, qui sont des réunions standards dont l'accès est réservé à un groupe de personnes défini par l'organisateur
  - Réunions démocratiques, qui sont planifiées comme des réunions standards, mais où les intervenants successifs sont choisis automatiquement par le serveur sur la base d'une politique premier demandeur-premier servi.

# Démarche de modélisation avec UML

---

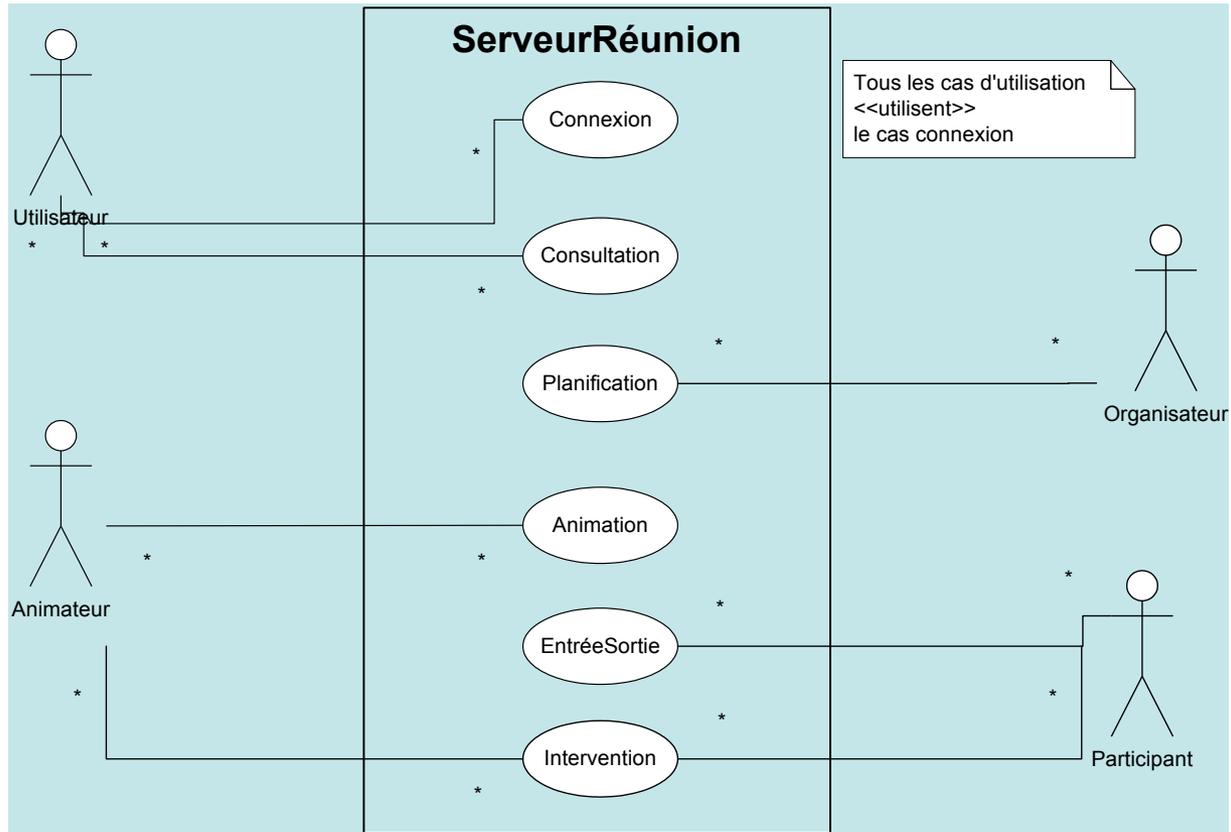
- Construction du diagramme de cas d'utilisations
  - Grande découpe fonctionnelle (10% de l'effort)
- Construction du diagramme de classes
  - à partir des noms des données du problème (30%)
- Construction de diagrammes de séquences et de collaborations
  - instances des cas d'utilisation (25%)
- Généralisation à l'aide de diagrammes d'états-transitions
  - à partir des diag. Séquences (15%)
- Affiner et préciser la solution (20%)

# Cas d'utilisation

---

- Grande découpe fonctionnelle
  - pilote les incréments dans la spirale
  - chaque incrément correspond à la réalisation d'un cas d'utilisation
- 1 diagramme global + texte 10 lignes / cas

# Diagramme de cas d'utilisations



# Exemple de cas d'utilisation :

## Planification

---

- La *planification* d'une réunion virtuelle est effectuée par une personne jouant le rôle d'organisateur pour cette réunion. Ceci consiste à créer une nouvelle réunion dans le système (ou à la mettre à jour si elle existe déjà) en faisant le choix d'un nom, la définition du sujet, de la date de début et la durée prévue, ainsi que l'ordre du jour.

# Processus de construction du diagramme de classes

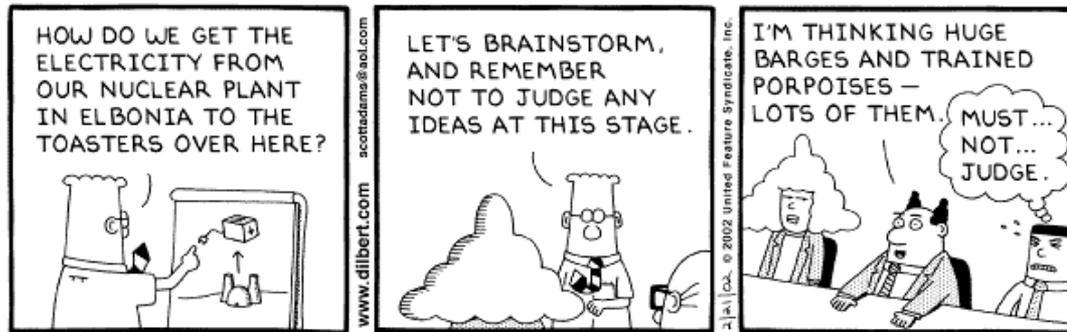
---

- Identifier les classes d'objets
  - Garder les bonnes classes
  - constitution du dictionnaire de données
- Identifier les associations.
  - Garder les bonnes associations
- Identifier les attributs.
  - Garder les bons attributs
- Raffiner au moyen de l'héritage
  - Généralisations et raffinages
- Itérer la modélisation
- Grouper les classes en modules

# Identification des classes

---

- A l'aide des noms des données du problème
  - processus de remue-méninges (brainstorming)



Copyright © 2002 United Feature Syndicate, Inc.

- **Eliminer les classes :**
  - redondantes (noms synonymes)
  - non pertinentes (vs. le modèle)
  - trop vagues (non réifiable facilement)
  - attributs, opérations, rôles de relations
  - constructions liées à l'implantation

# Souligner les noms dans le cahier des charges

---

- Il s'agit de réaliser la partie serveur d'une application client-serveur permettant de faire des réunions virtuelles multimédia sur Internet. L'objectif de cette application est de permettre d'imiter le plus possible le déroulement de réunions de travail classiques. Cependant, dans la première version de ce projet, les interventions des participants se feront en mode mono-média seulement (i.e. échanges en forme textuelle).
- Le serveur devra permettre de planifier et de gérer le déroulement de plusieurs réunions simultanées. Des programmes clients existeront dans l'avenir pour plusieurs plate-formes (Mac, Windows, Unix) afin de permettre à des personnes désirant organiser des réunions virtuelles ou y participer de dialoguer avec le serveur en utilisant un protocole ad hoc développé au dessus de IP.

# Classes potentielles

- Serveur
- Application
- Réunion
- Internet
- Objectif
- Déroulement
- Version
- Projet
- Intervention
- Participant
- Mode
- Programme
- Plate-formes
- Personne
- Protocole
- ...

Implantation

Redondant serveur

OK

Implantation

Non pertinent

Action

Implantation

Non pertinent

OK (?)

Rôle relation Personne-Réunion

Implantation

Implantation

Implantation

OK

Implantation

# Identification des relations entre classes

---

- Recherche des phrases verbales
- Eliminer les relations :
  - entre classes éliminées
  - non pertinentes ou liées à l'implantation
  - qui sont en fait des actions
  - pouvant être dérivées d'autres relations
  - Raffiner la sémantique des relations
  - ajouter les rôles
  - qualifier les relations (sélecteur)
  - spécifier la multiplicité

# Identification des attributs

---

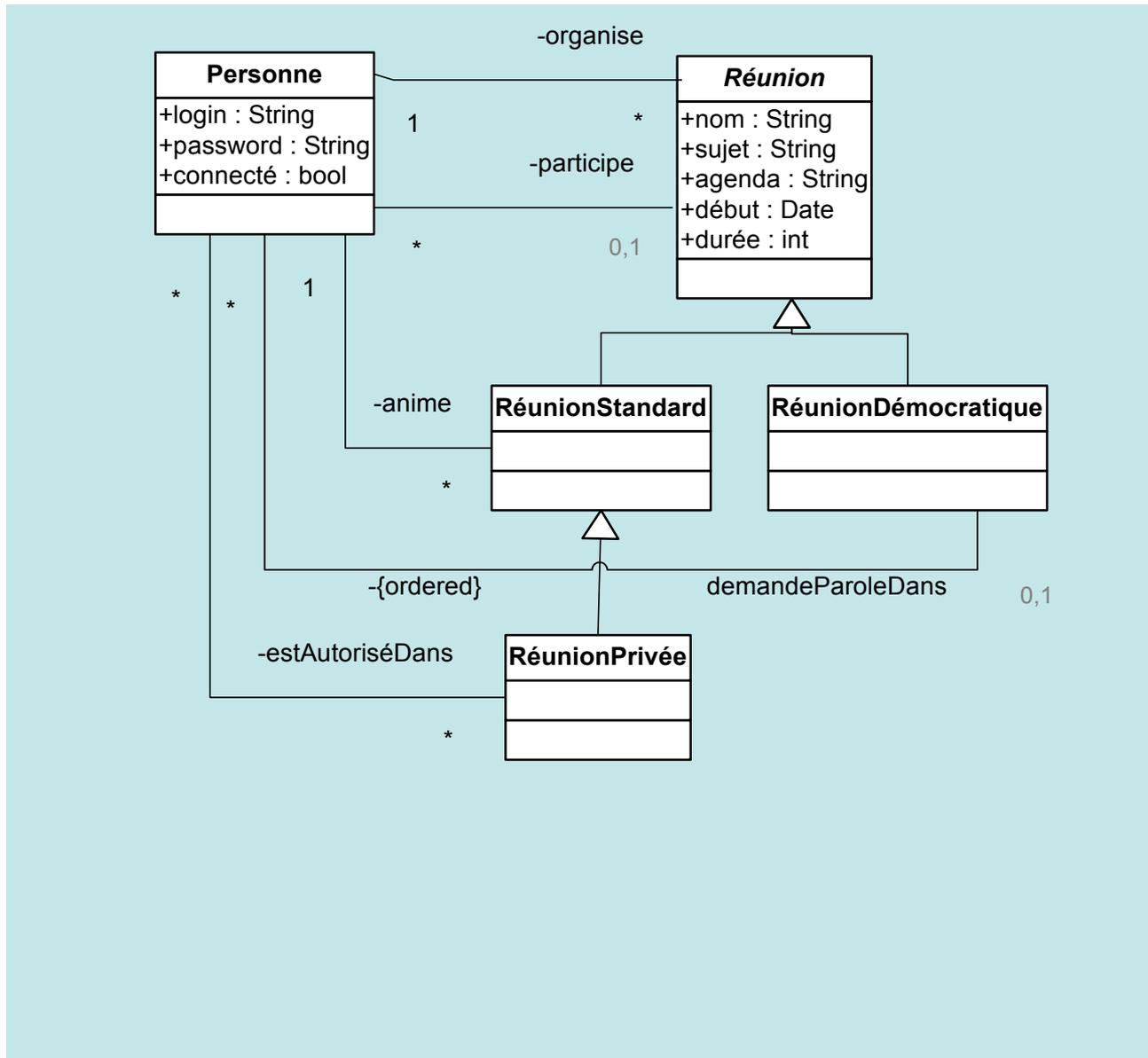
- **Propriétés d'objets individuels**
  - à rechercher à l'aide des adjectifs (couleur, poids...) ou propositions substantives du problème
- **Éliminer les attributs non nécessaires ou incorrects**
  - s'ils sont en fait des objets
  - sélecteurs de relations
  - identificateurs (clef de BD)
  - attributs de relations
  - valeurs internes ou détails d'implantation

# Raffiner au moyen de l'héritage

---

- **Généralisation à l'aide de super-classes**
  - Recherche de classes avec des attributs, relations ou opérations similaires
- **Spécialisation à l'aide de sous-classes**
  - Différentes variantes d'une même classe
    - Réunion privée, démocratique ...
  - sous-classe ou attribut pour distinguer ?

# Diagramme de classes d'analyse



# Itérer la modélisation

---

- **Classes manquantes ou en trop**
  - asymétries : ajout de classes par analogies
  - scinder les classes disparates en classes plus élémentaires
  - A quoi sert une classe si pas d'attributs ou d'opérations?
- **Relations manquantes, en trop ou mal placées**
  - si aucune opération ne traverse une relation...
- **Attributs**
  - accéder à un objet par un attribut -> relation qualifiée
- **Il faut parfois attendre d'avoir fait les autres vues pour pouvoir itérer**

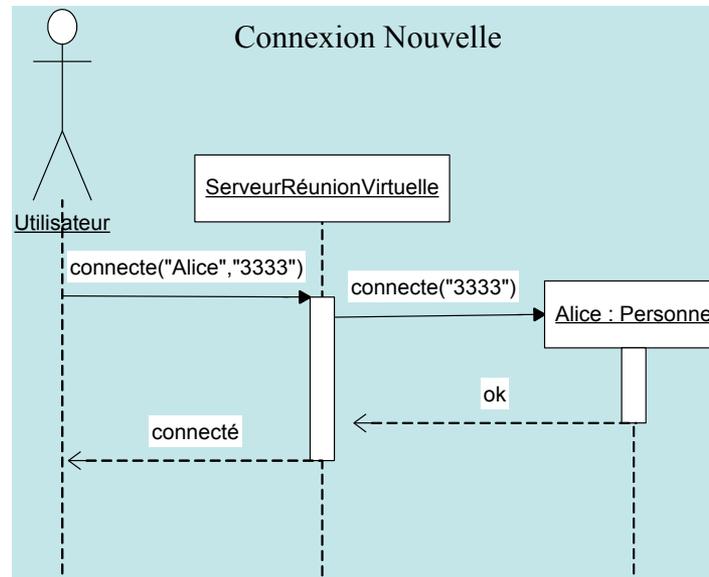
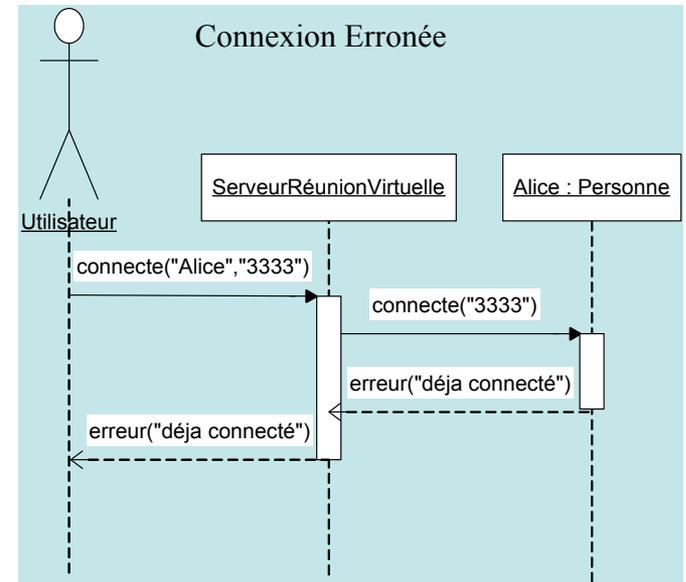
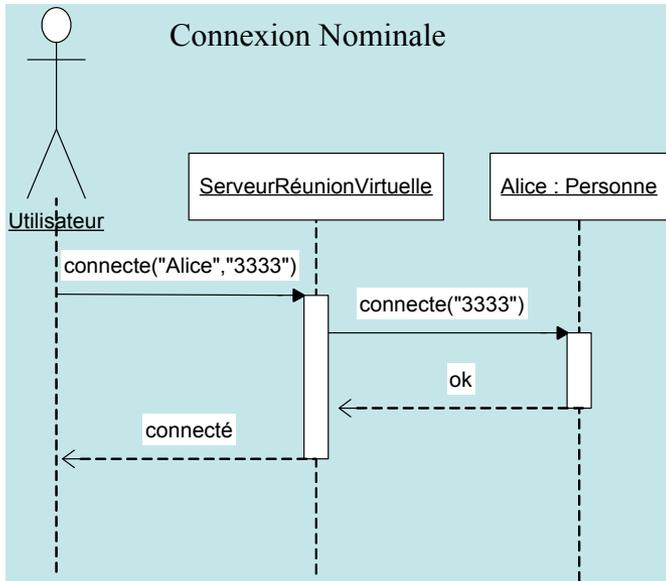
# Diagrammes dynamiques

---

- Pour chaque cas d'utilisation
  - un scénario nominal (diagramme de séquence)
    - tout se passe bien
  - quelques scénarios exceptionnels
    - montrent des variations sur le scénario optimal
- Alternativement (Catalysis)
  - description par pre/post de l'état du système avant/après chaque occurrence d'événement

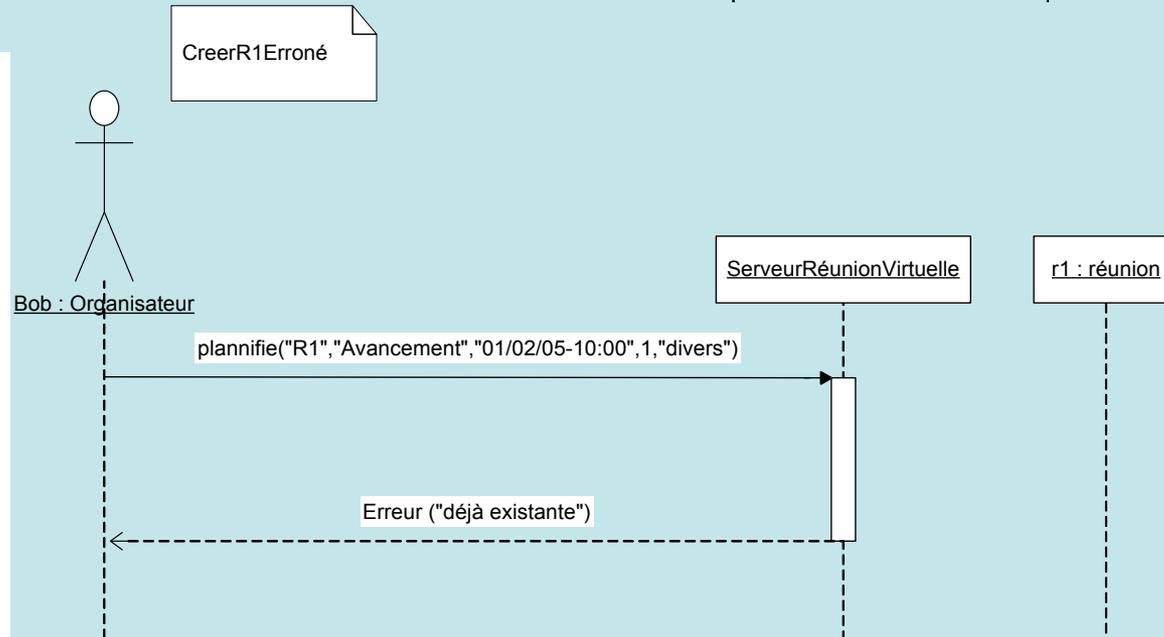
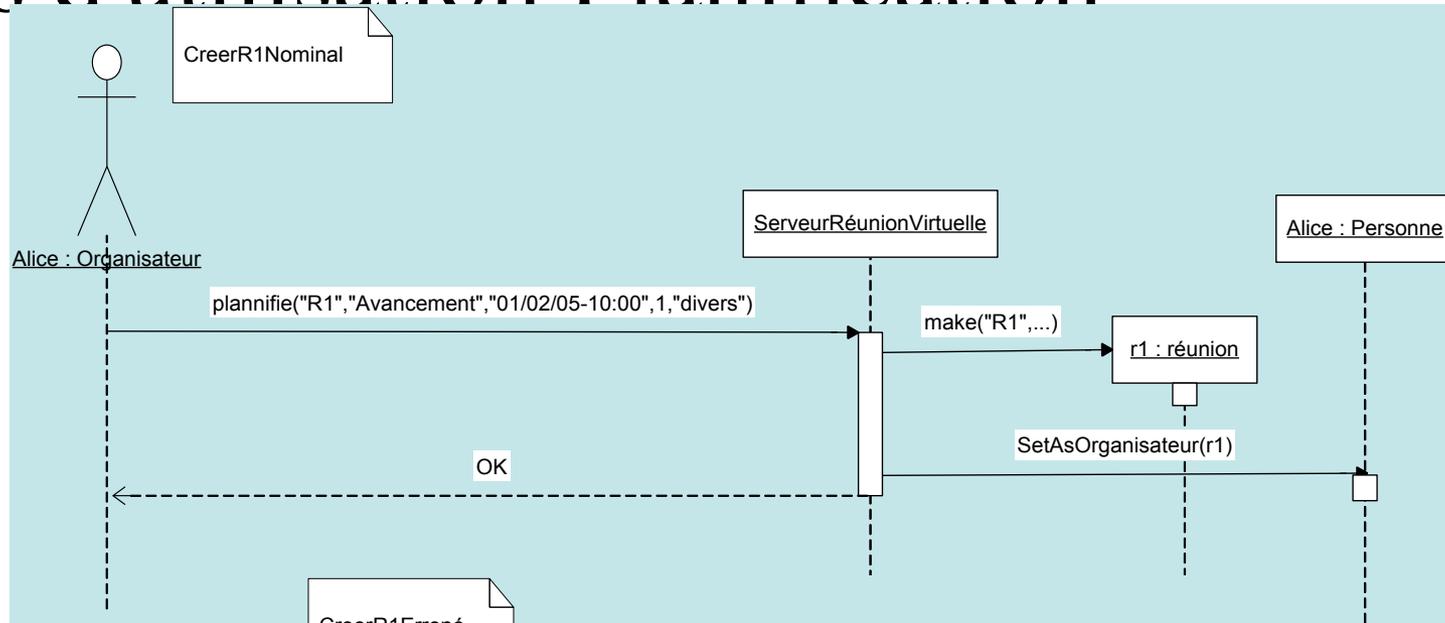
# Exemples de scénarios :

## Cas d'utilisation Connexion



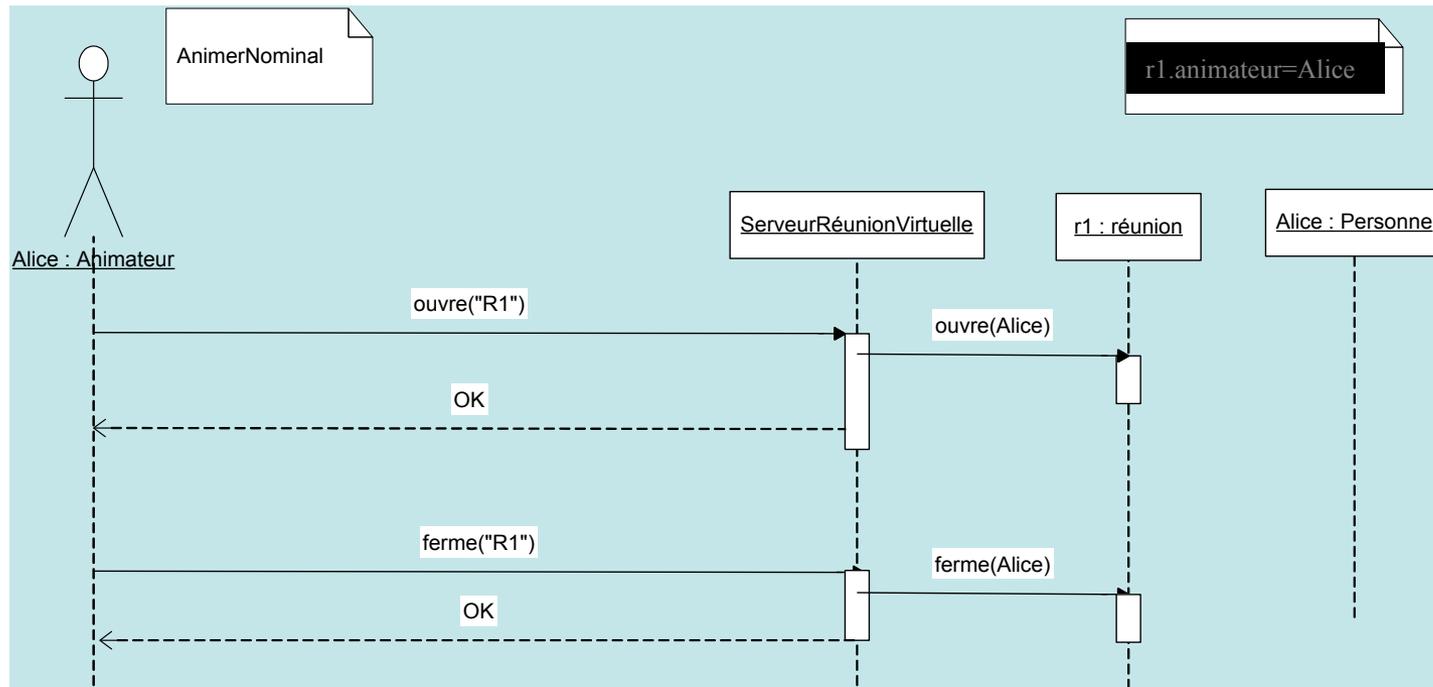
# Exemples de scénarios :

## Cas d'utilisation Planification



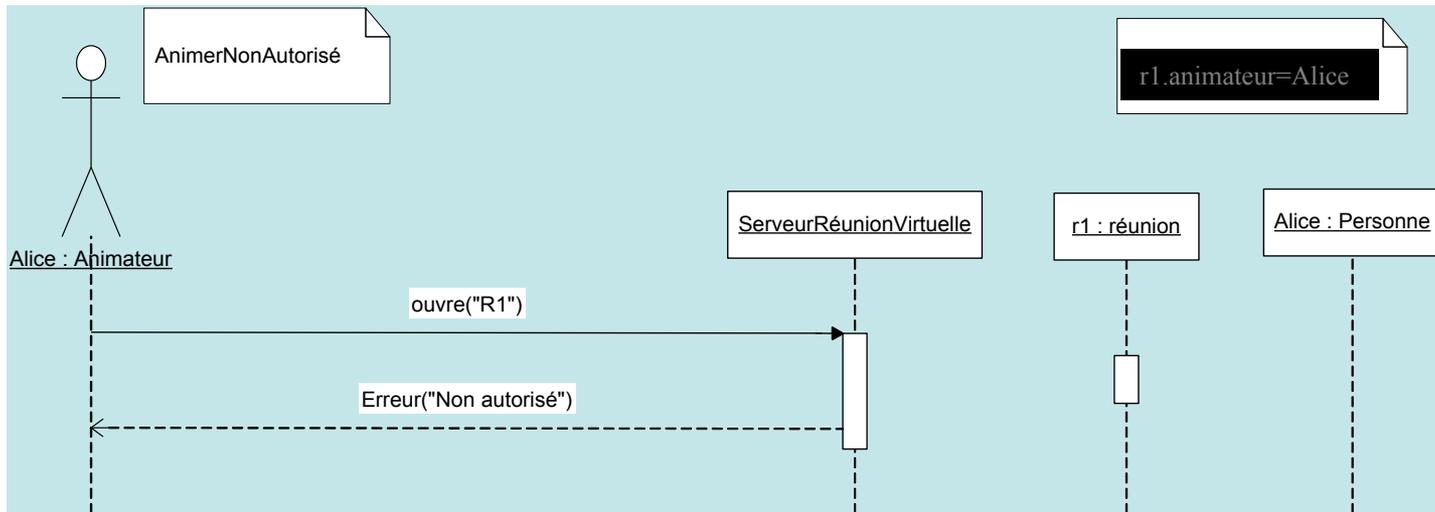
# Exemples de scénarios :

## Cas d'utilisation Animation 1/3



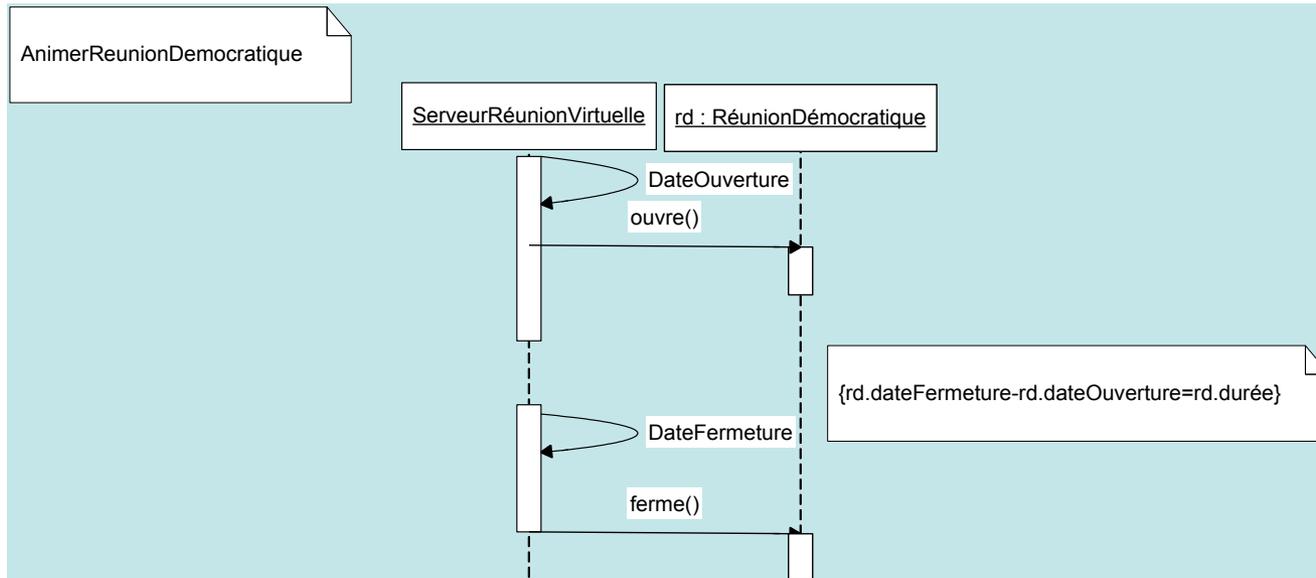
# Exemples de scénarios :

## Cas d'utilisation Animation 2/3



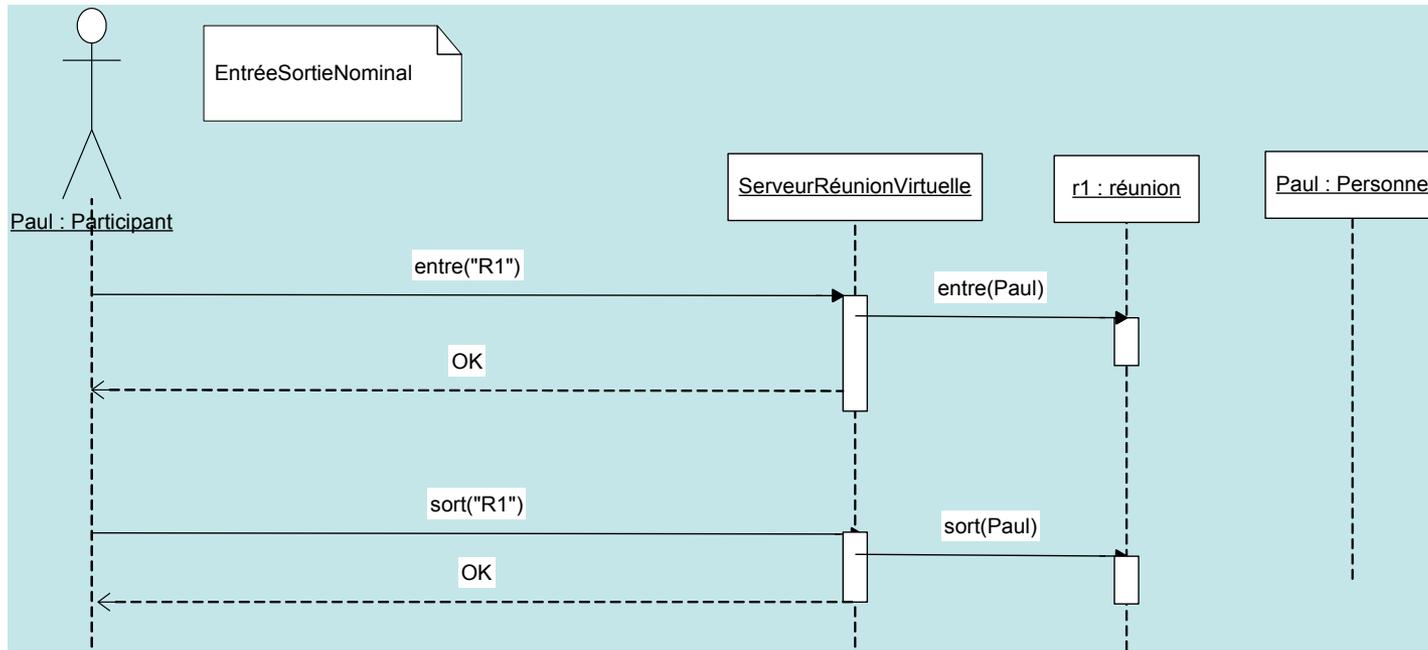
# Exemples de scénarios :

## Cas d'utilisation Animation 3/3



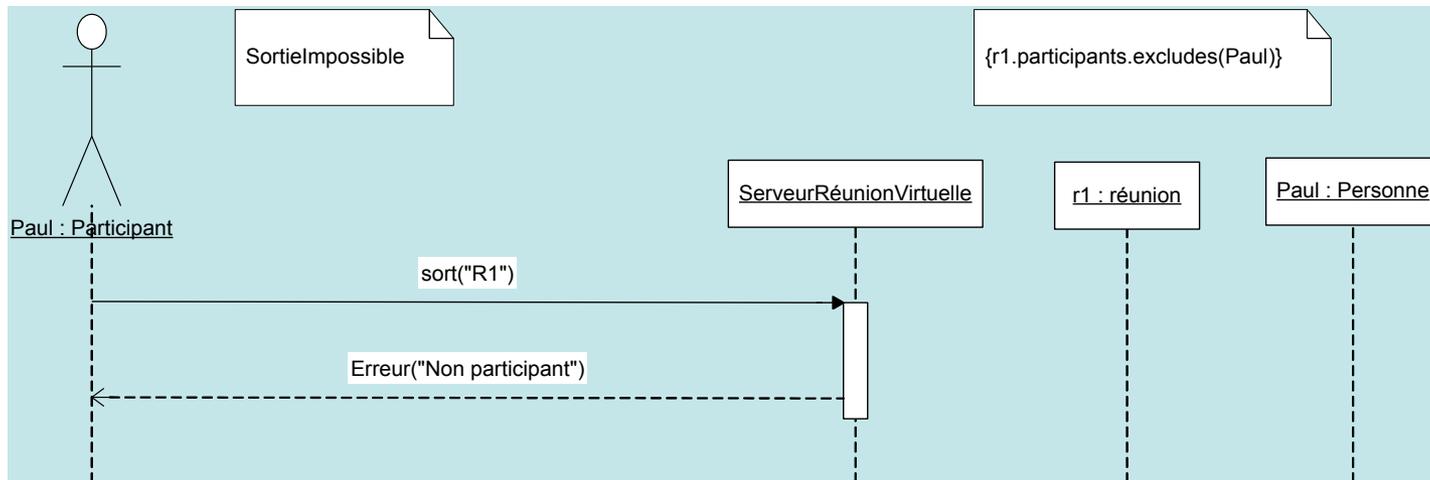
# Exemples de scénarios :

## Cas d'utilisation EntréeSortie 1/4



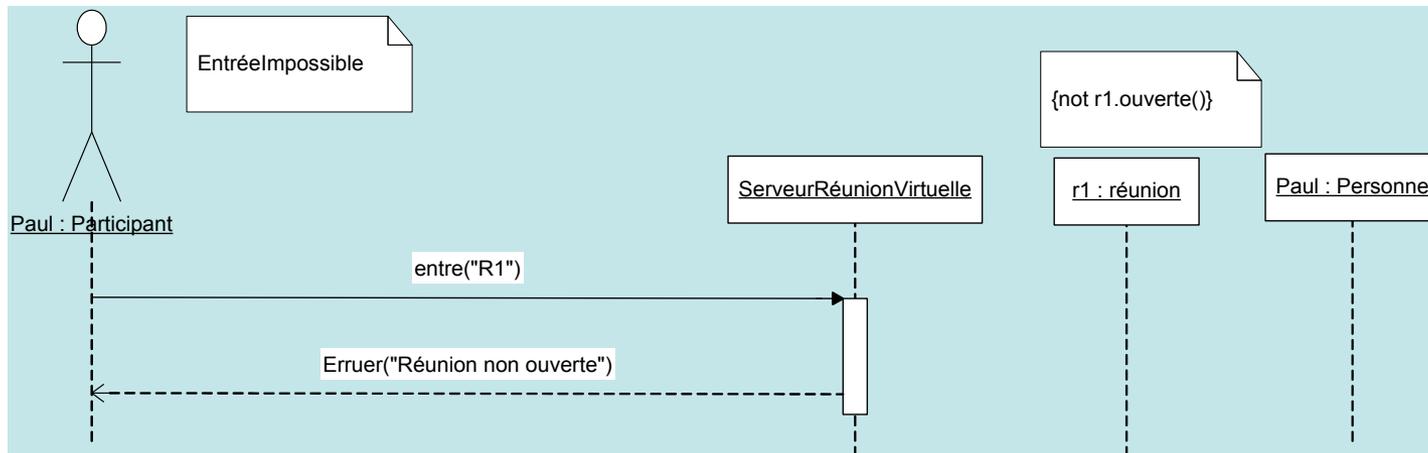
# Exemples de scénarios :

## Cas d'utilisation EntréeSortie 2/4



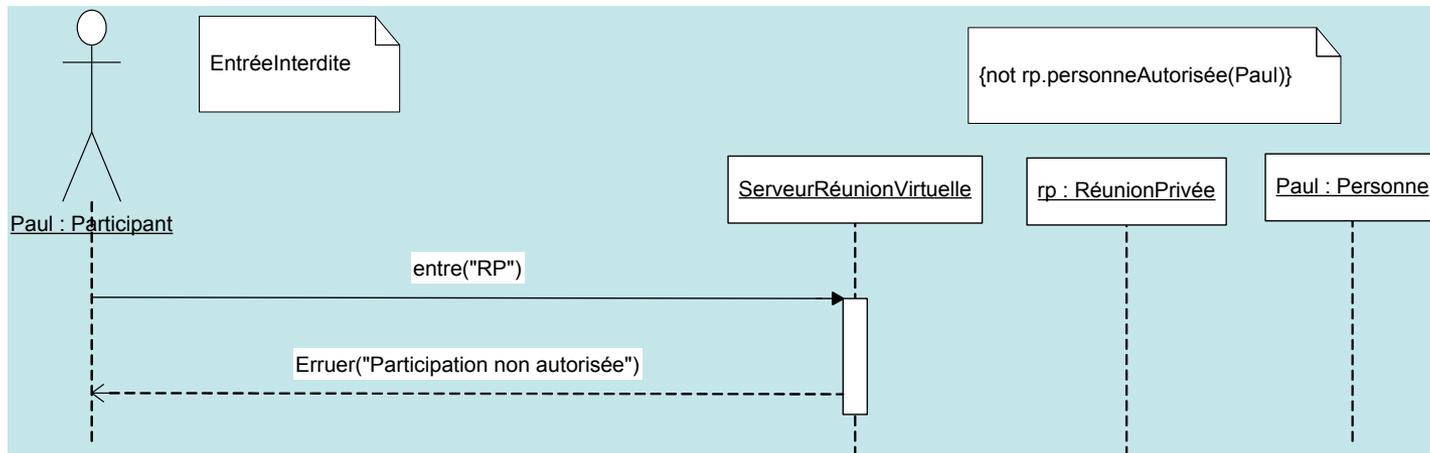
# Exemples de scénarios :

## Cas d'utilisation EntréeSortie 3/4



# Exemples de scénarios :

## Cas d'utilisation EntréeSortie 4/4

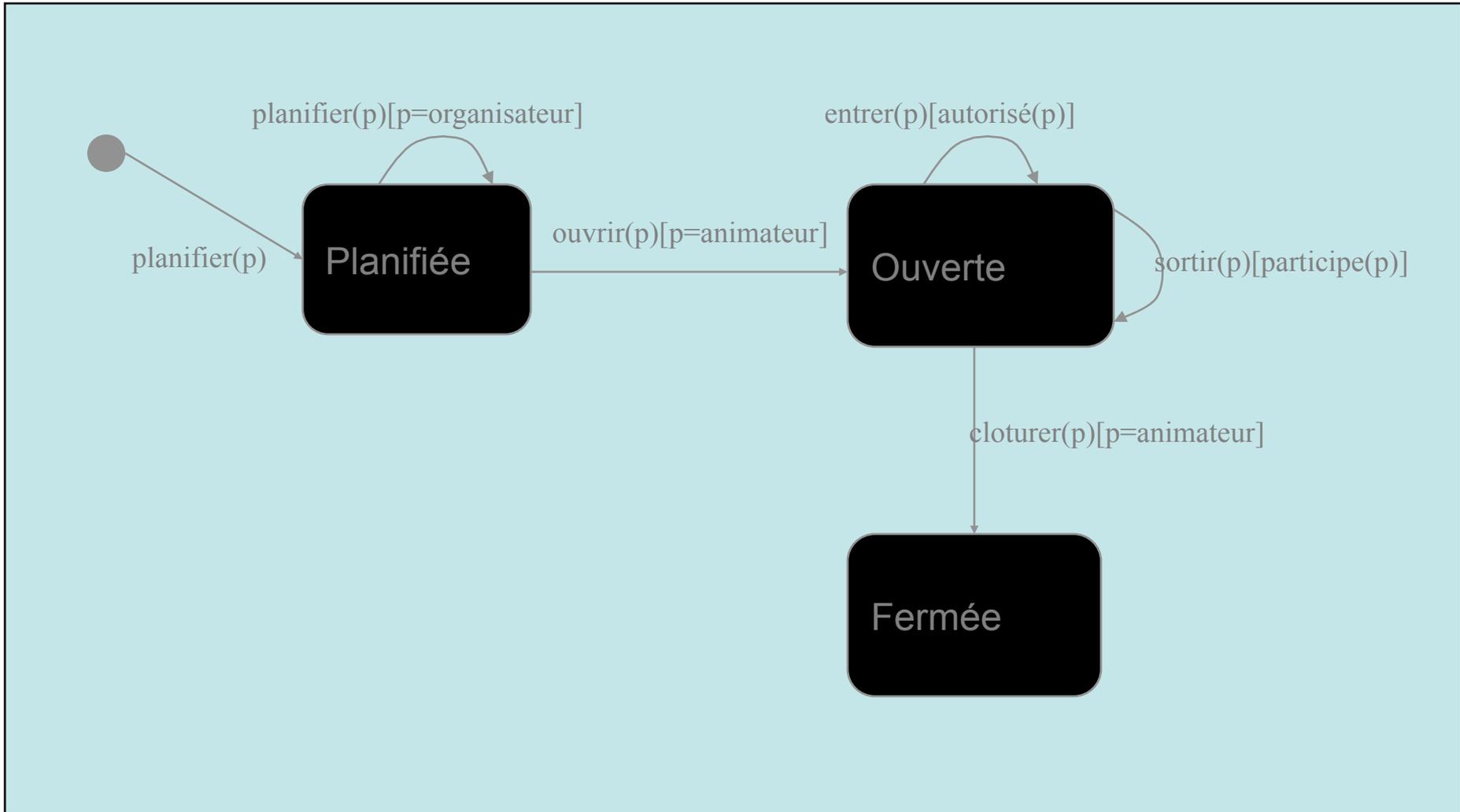


# Construction des diagrammes d'états

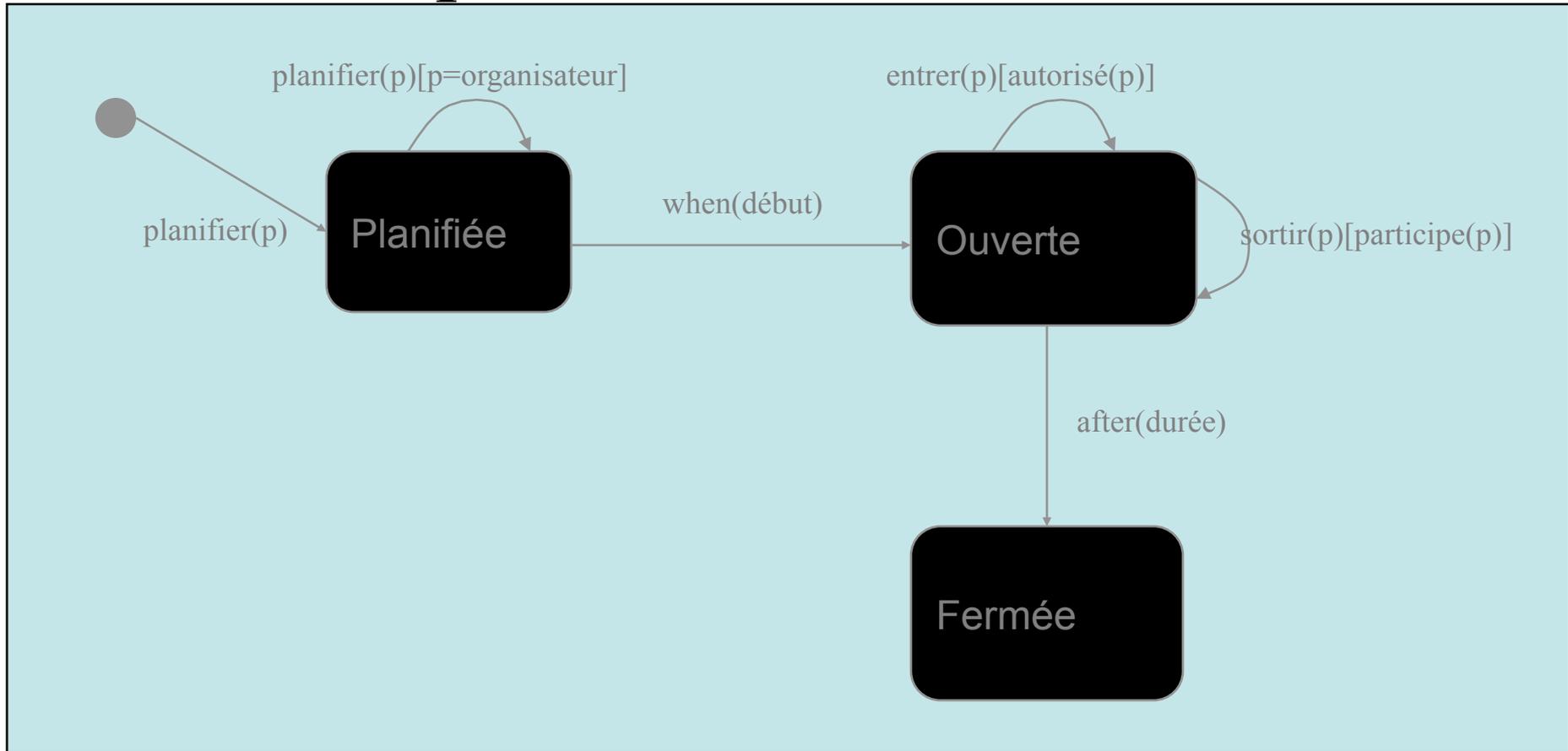
---

- Généralisation pour une classe donnée de l'ensemble des scénarios qui mettent en jeu ses instances
  - en suivant les transition de l'automate, on doit pouvoir retrouver tous les scénarios

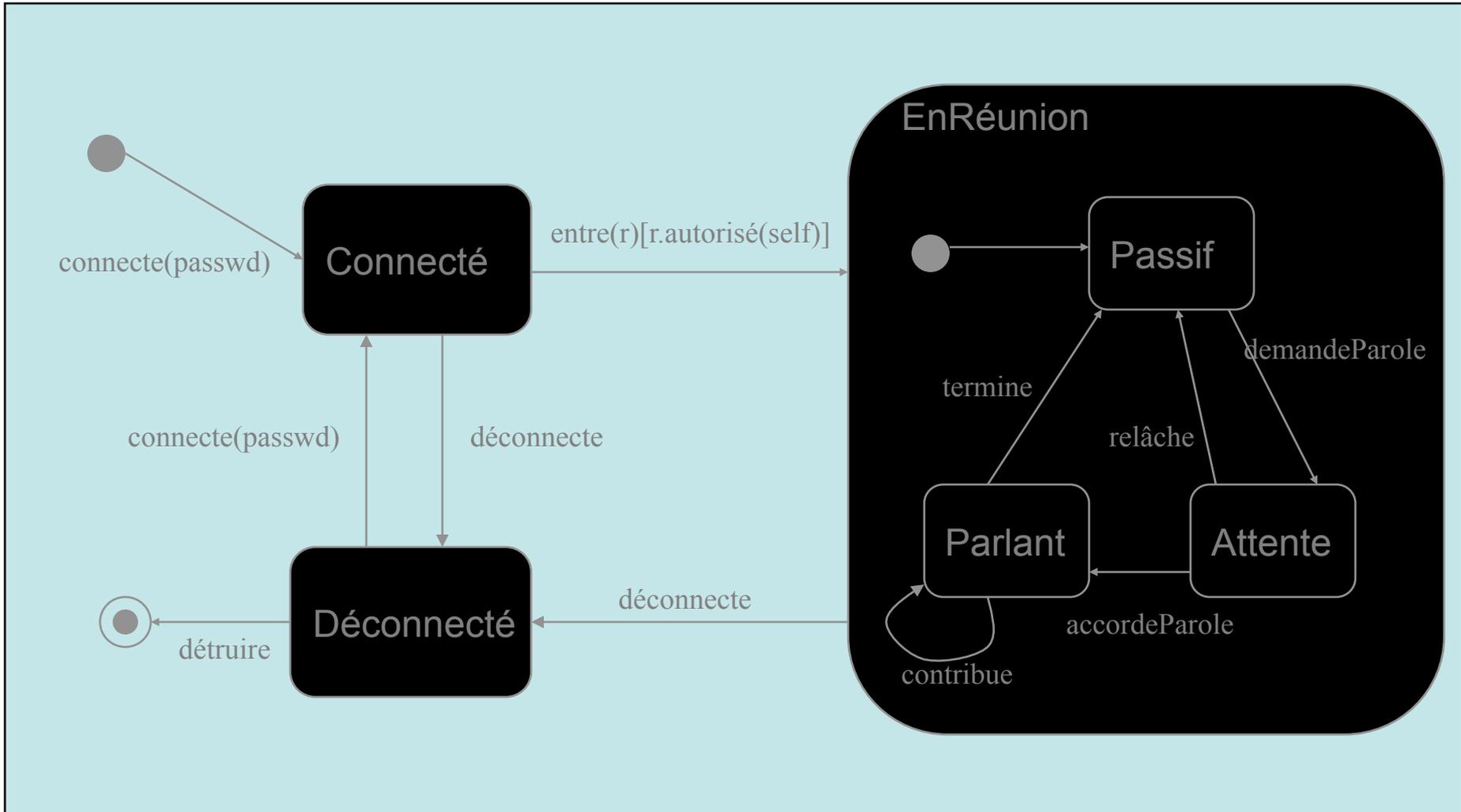
# Diagramme d'états : Réunion



# Diagramme d'états : Réunion démocratique



# Diagramme d'états : Personne



# Conseils pratiques

---

- Réfléchir au problème avant de commencer
  - Soigner le nommage, insister sur le nommage des relations et des rôles
- Faire simple!
  - «*Things must be as simple as possible, but no simpler*». A. Einstein
  - éviter toute complication nuisible
    - utiliser les qualifieurs
    - éviter les relations ternaires, quaternaires (trop complexe)
    - se dégager de l'implémentation : raisonner objets, classes, messages, relations, attributs, opérations
  - ne pas s'inquiéter si les possibilités de la notation ne sont pas toutes exploitées

# Conseils pratiques (suite)

---

- **Approche incrémentale**
  - Itérer
  - Confronter ses modèles aux autres
  - Savoir s'arrêter avant d'atteindre la perfection...
    - prise en compte qualité (niveau de précision), coûts, délais...
    - asservissement au processus de développement
- **Faire simple (encore)**
  - *Il semble que la perfection soit atteinte, non quand il n'y a plus rien à ajouter mais quand il n'y a plus rien à retrancher - Antoine de Saint-Exupéry, Terre des hommes*

# Critères de qualité d'un bon modèle d'analyse avec UML (1/2)

---

- Cas d'utilisation

- environ 6 cas, nommage correct des acteurs (noms) et des cas (actions)
- pour chaque cas, présence d'un court texte d'explication
- complétude vs. Cahier des charges

- Diagramme de classes

- nommage correct des classes (noms)
- définitions présentes dans un dictionnaire
- nommage de toutes les relations, présence des cardinalités
- attributs seulement de types simples
- non duplication attributs/rerelations (utilisation héritage)

# Critères de qualité d'un bon modèle d'analyse avec UML (2/2)

---

- Diagrammes de séquence

- attachés à un cas d'utilisation
- pour chaque cas d'utilisation,
  - présence d'un scénario nominal
  - quelques scénarios exceptionnels
- chaque opération (message reçu par un objet) définie dans la classe correspondante du diagramme statique

- Diagrammes d'états

- chaque automate est attaché à une classe
- chaque attribut/opération utilisée sur l'automate doit être défini dans la classe englobante
- en suivant les transitions de l'automate, on doit pouvoir retrouver tous les scénarios

# Quatrième partie (au-delà de UML)

# Outline

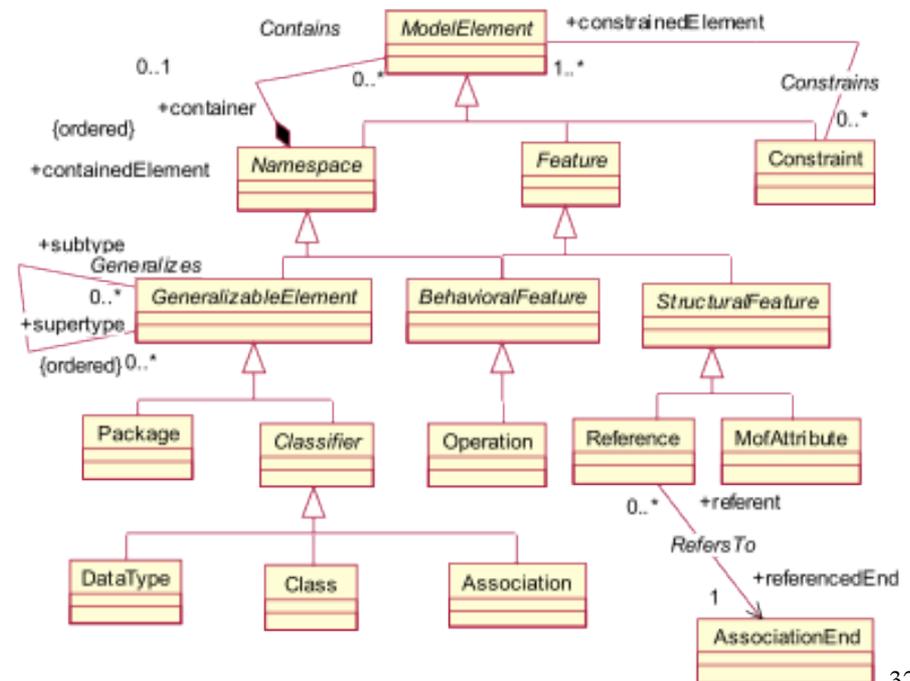
---

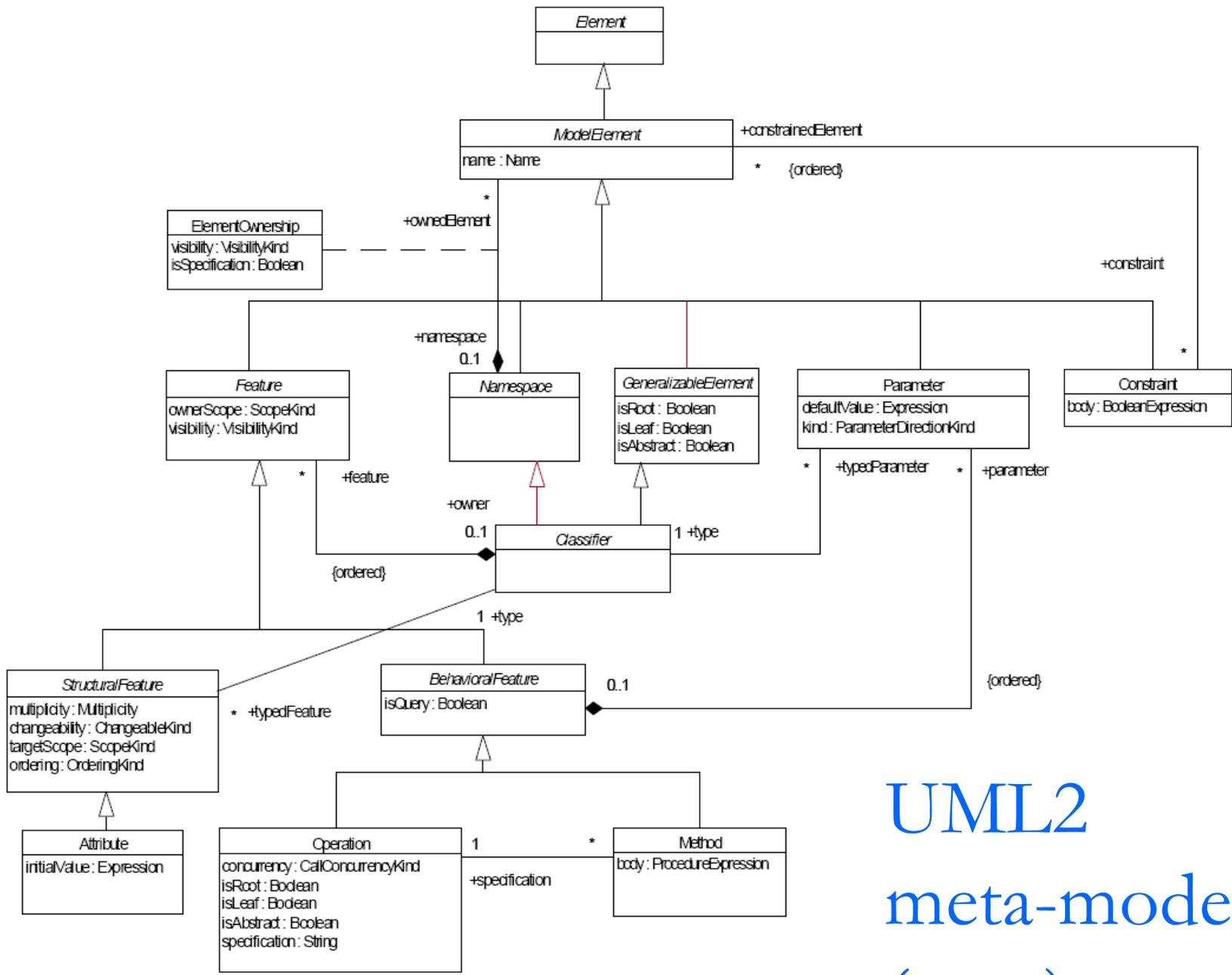
- ① UML History and Overview
- ② UML Language
  - ① UML Functional View
  - ② UML Structural View
  - ③ UML Behavioral View
  - ④ UML Implementation View
  - ⑤ UML Extension Mechanisms
- ③ UML Internals
- ④ UML Tools
- ⑤ Conclusion

# Assigning Meaning to Models

- If a UML model is no longer just
  - fancy pictures to decorate your room
  - a graphical syntax for C++/Java/C#/Eiffel...
- Then tools must be able to manipulate models

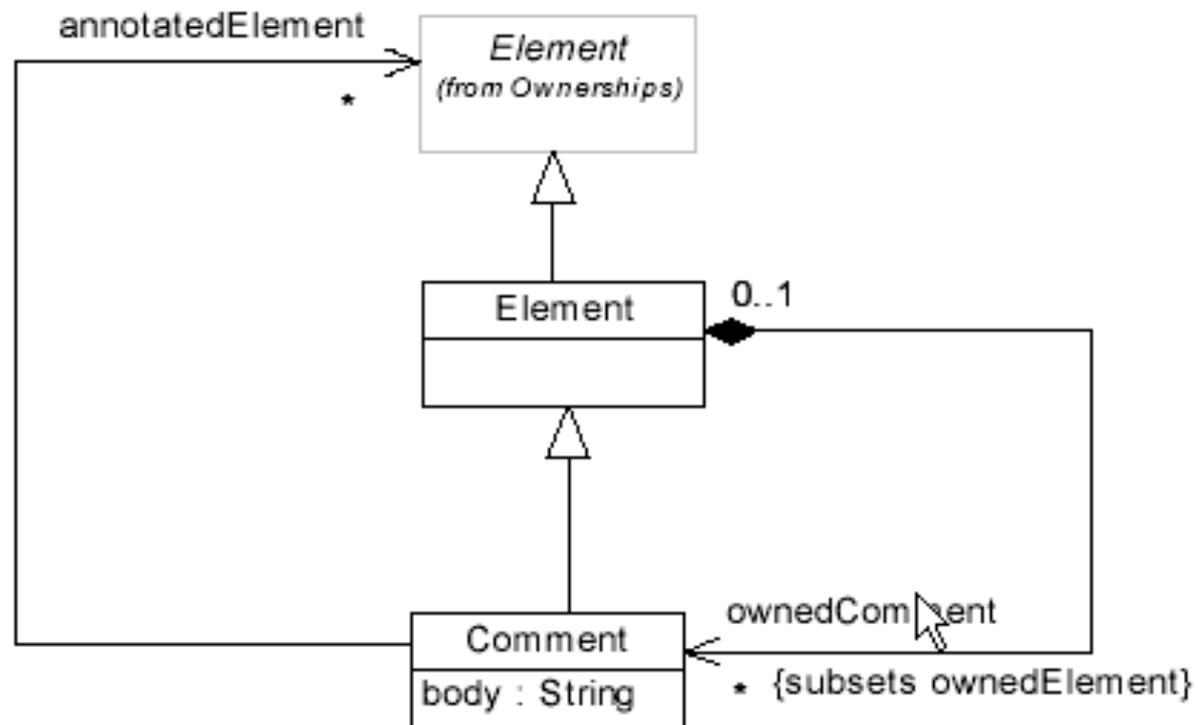
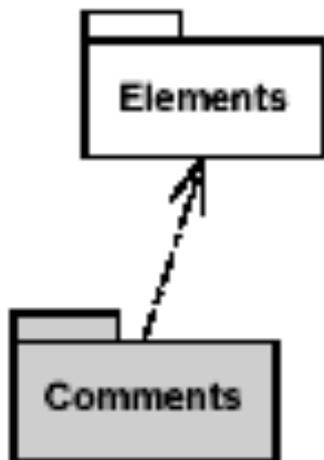
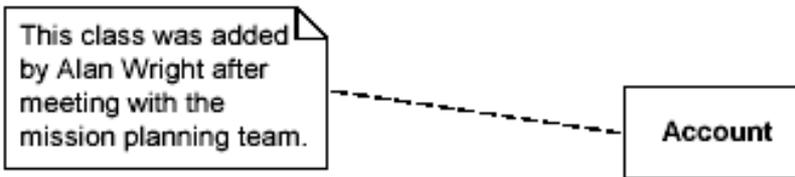
- Let's make a model
- of what a model is!
- => *meta-modeling*
  - & meta-meta-modeling..





# UML2 meta-model (part.)

# Zoom: comments



# Generalizations

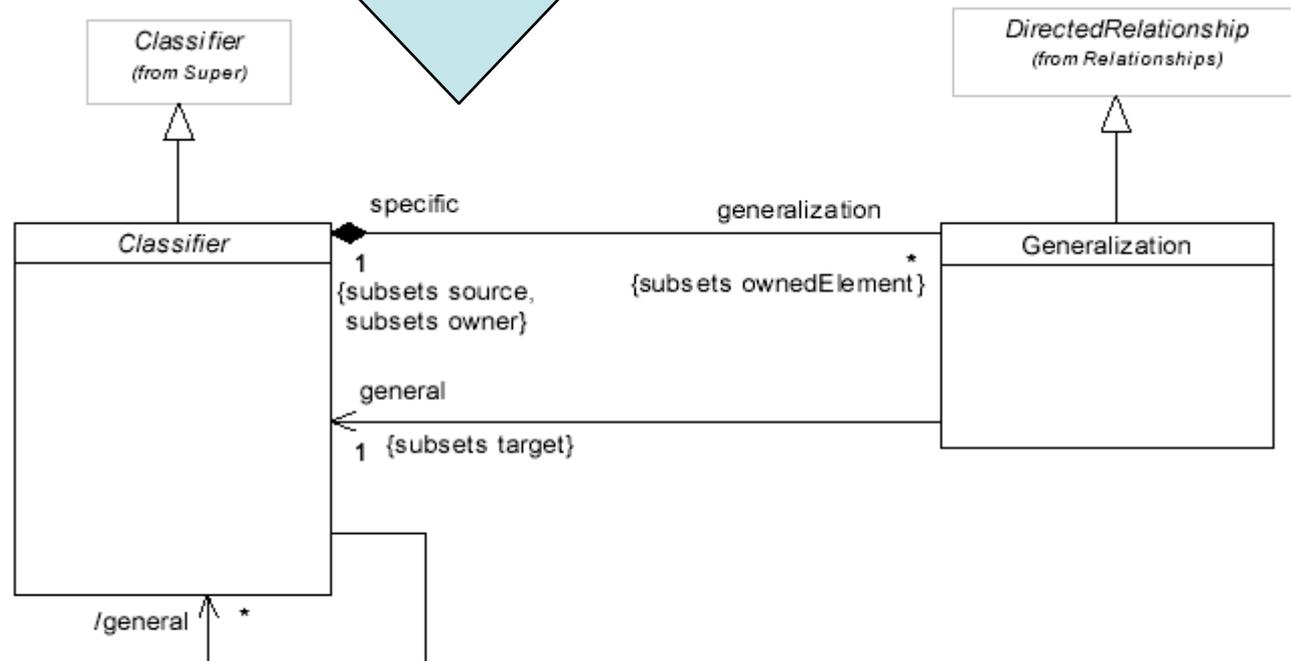
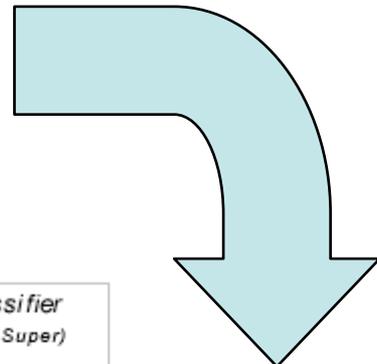
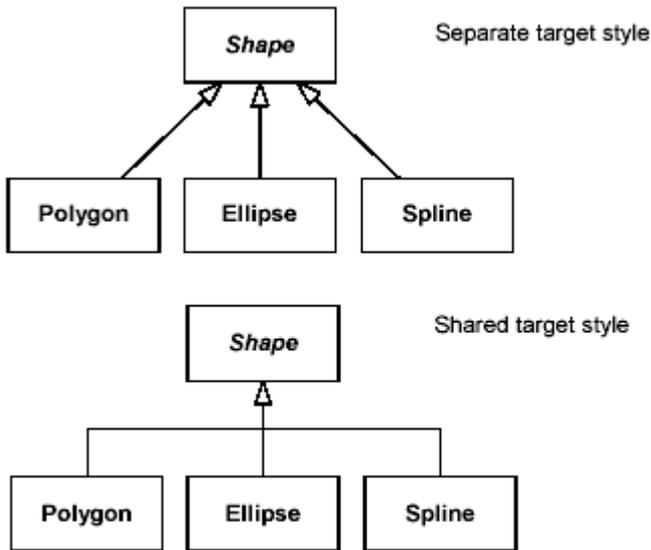
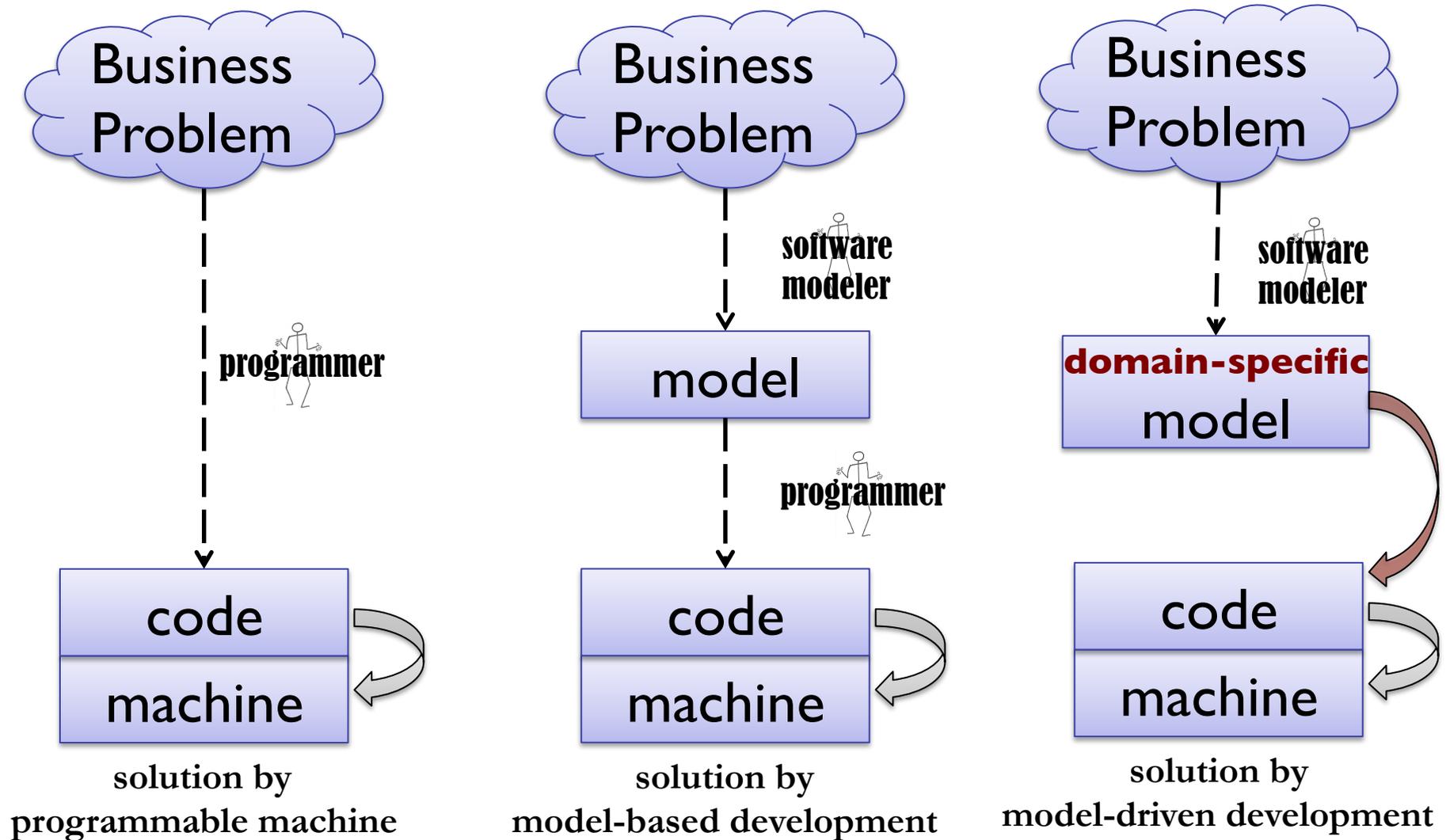


Figure 3-33. Examples of generalizations between

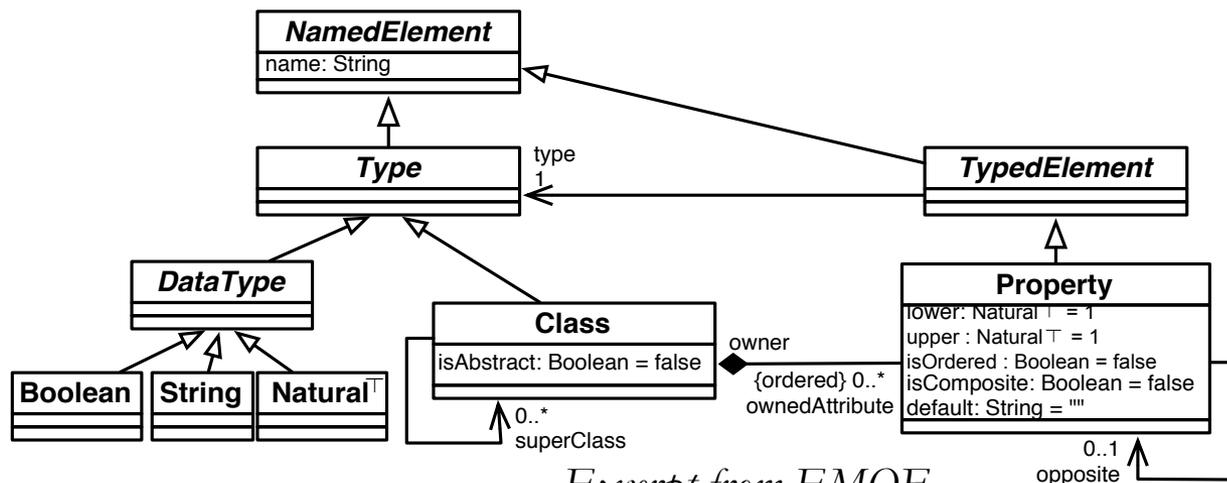
Figure 3-32. The elements defined in the Generalizations package.

# Evolution in Software Modeling



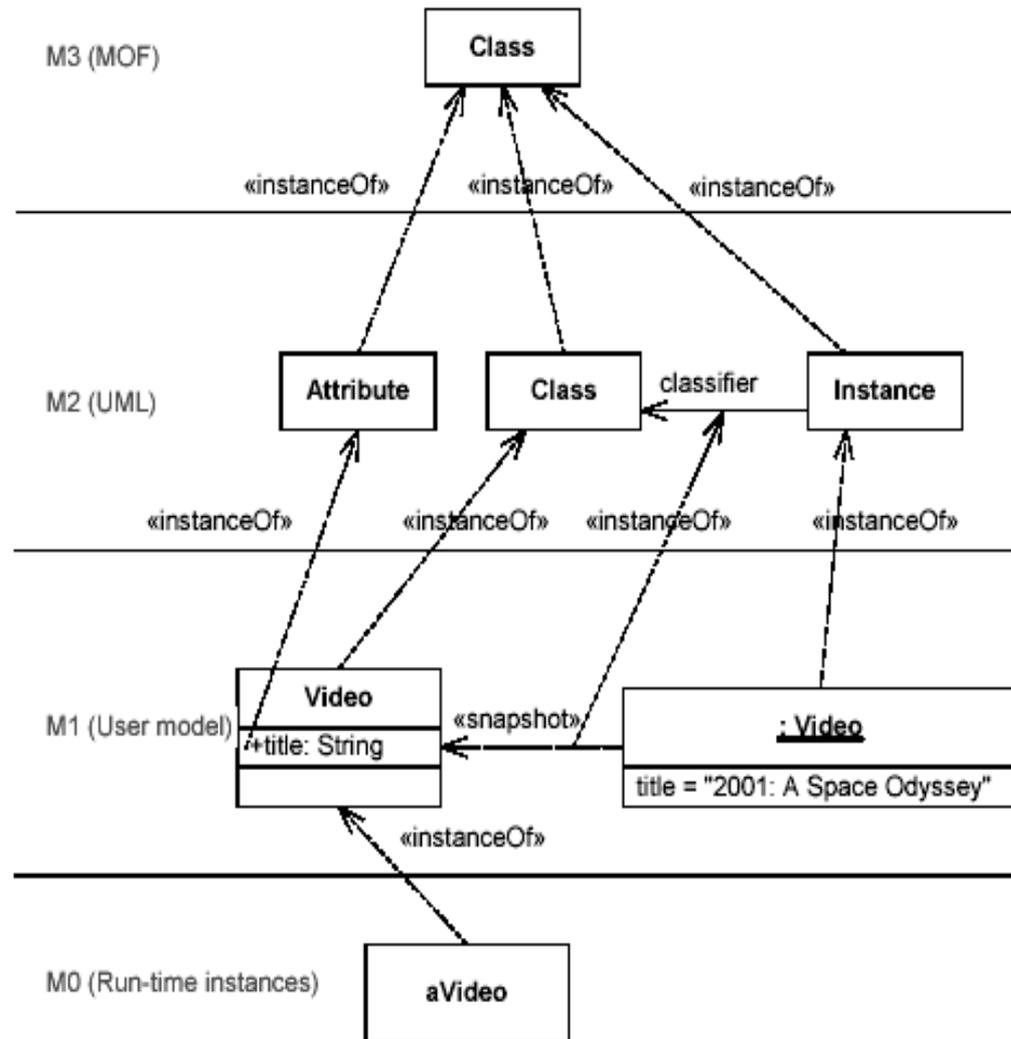
# Assigning Meaning to MetaModels

- **OMG (Essential) MOF:**
  - Provides language constructs for specifying a DSL metamodel
    - mainly based on Object-Oriented constructs: package, classes, properties (attribute and reference), and multiple inheritance.
    - specificities: composition, opposite...
  - Cf. <http://www.omg.org/spec/MOF/>

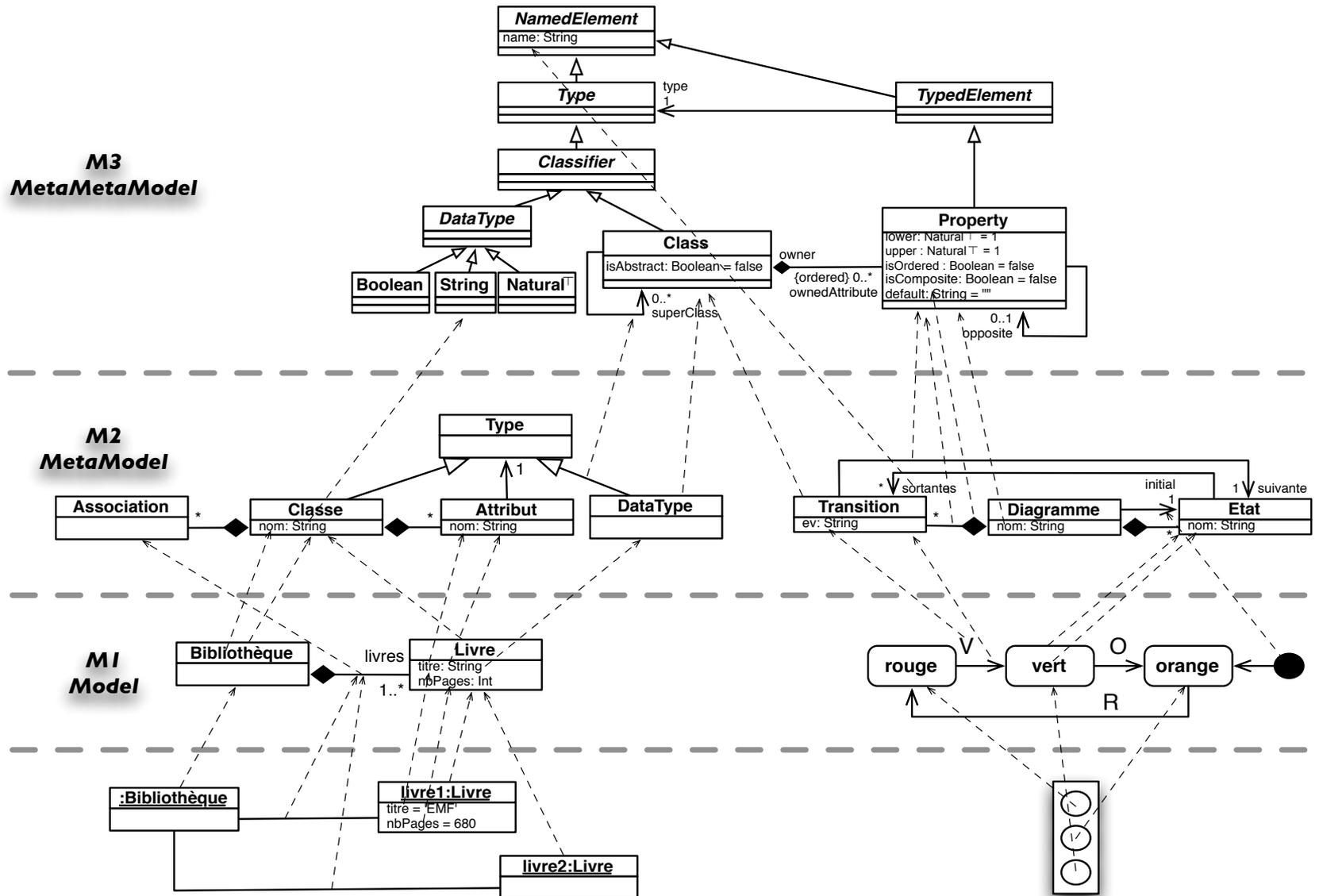


*Excerpt from EMOF*

# The 4 layers in practice



# The 4 layers in practice



# Outline

---

- ① UML History and Overview
- ② UML Language
  - ① UML Functional View
  - ② UML Structural View
  - ③ UML Behavioral View
  - ④ UML Implementation View
  - ⑤ UML Extension Mechanisms
- ③ UML Internals
- ④ UML Tools
- ⑤ Conclusion

# Some UML Tools

---

- Some open source tools:

- ↳ Eclipse Papyrus MDT, Modelio (Modeliosoft)...

- Some commercial tools:

- ↳ Magic Draw (No Magic), Enterprise Architect (Sparx Systems), Objecteering (Objecteering Software), genmymodel...

- *Drawing tools (to be avoided):*

- ↳ Dia, Visio...

- (Some) more complete lists:

- [http://en.wikipedia.org/wiki/List\\_of\\_Unified\\_Modeling\\_Language\\_tools](http://en.wikipedia.org/wiki/List_of_Unified_Modeling_Language_tools)
  - <http://www.umltools.net>

# Eclipse Papyrus MDT

---

- Environment for editing any kind of EMF model and particularly supporting UML and related modeling languages such as SysML and MARTE.
- Papyrus also offers an advanced support of UML profiles
- Official Eclipse project for UML2



# Outline

---

- ① UML History and Overview
- ② UML Language
  - ① UML Functional View
  - ② UML Structural View
  - ③ UML Behavioral View
  - ④ UML Implementation View
  - ⑤ UML Extension Mechanisms
- ③ UML Internals
- ④ UML Tools
- ⑤ Conclusion

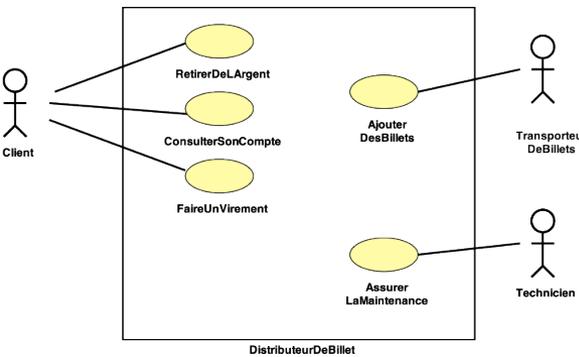
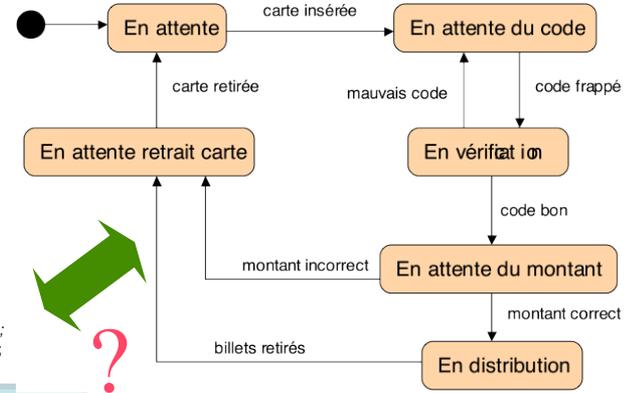
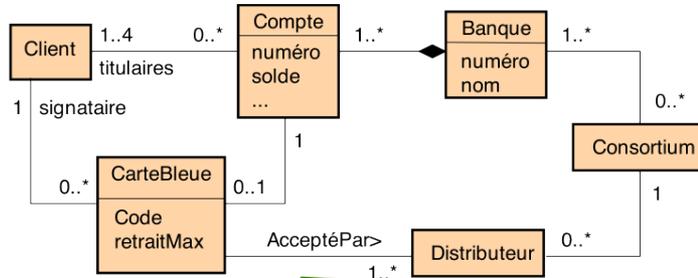
# Oui, Oui, Oui, mais ...

---

pour "l'informaticien moyen"

la seule chose importante dans le logiciel  
c'est ...

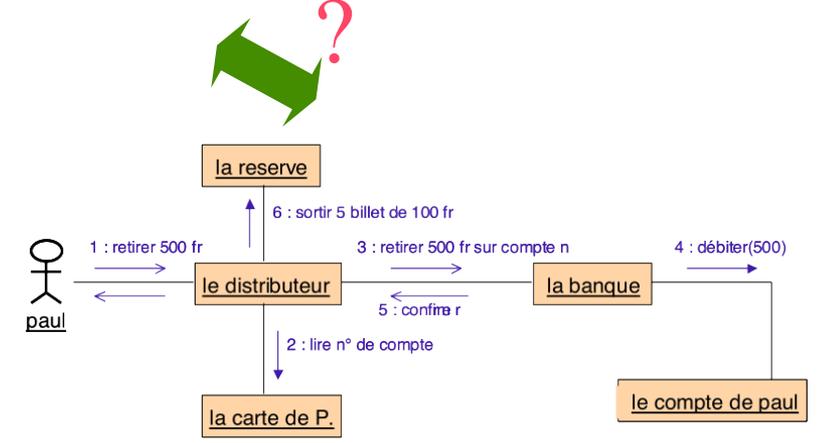
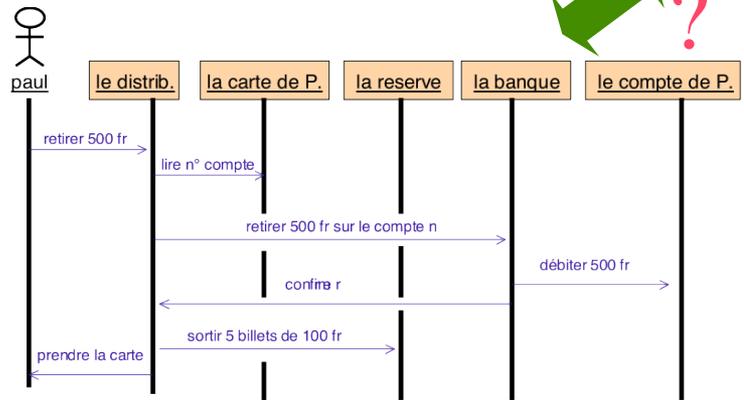
**le Code !**



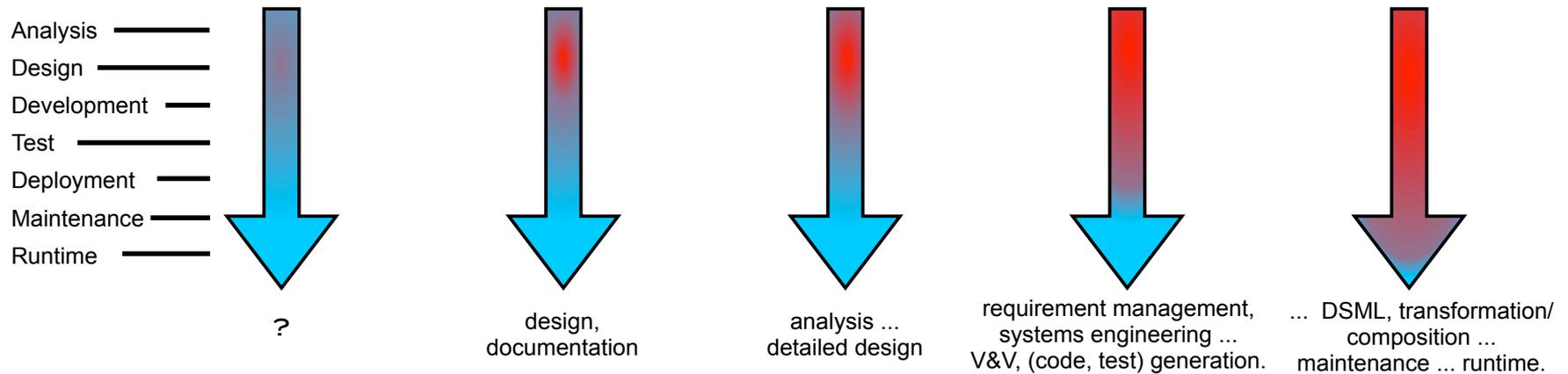
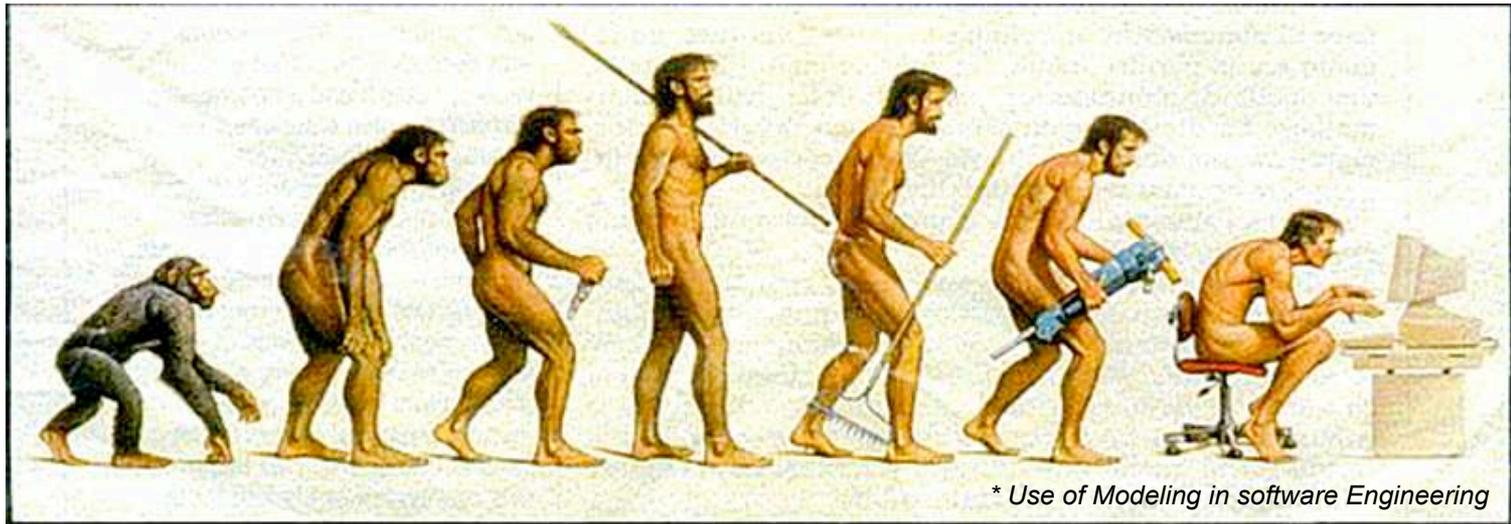
```

class Compte {
private Vector opérations ;
private int découvertMax ;
private Vector opérations ;
private int découvertMax ;
private Client titulaire ;
public int lireSolde()
{ int r = 0 ;
for (i=0;opérations.size();i++)
return r ; }
}
class Compte {
private Vector opérations ;
private int découvertMax ;
private Client titulaire ;
public int lireSolde()
{ int r = 0 ;
for (i=0;opérations.size();i++)
return r ; }
}
class Compte {
private Vector opérations ;
private int découvertMax ;
private Client titulaire ;
public void debiter ( int montant )
{ if (montant <= 0
|| lireSolde()-montant < lireSolde)
throw new Exception() ;
...
}
}
class Compte {
private Vector opérations ;
private int découvertMax ;
private Client titulaire ;
public void debiter ( int montant )
{ if (montant <= 0
|| lireSolde()-montant < lireSolde)
throw new Exception() ;
...
}
}

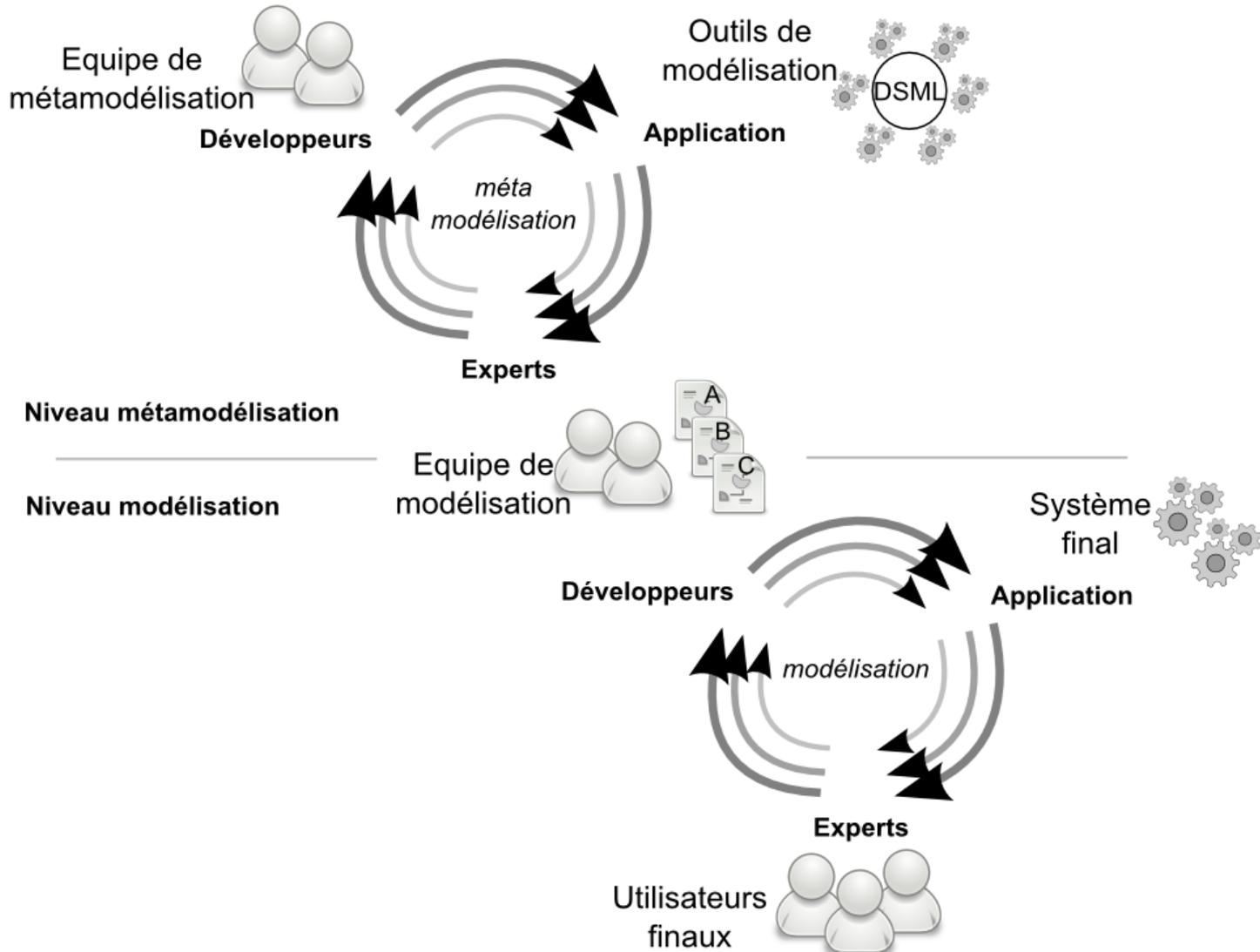
```



# Adoption of Software Modeling

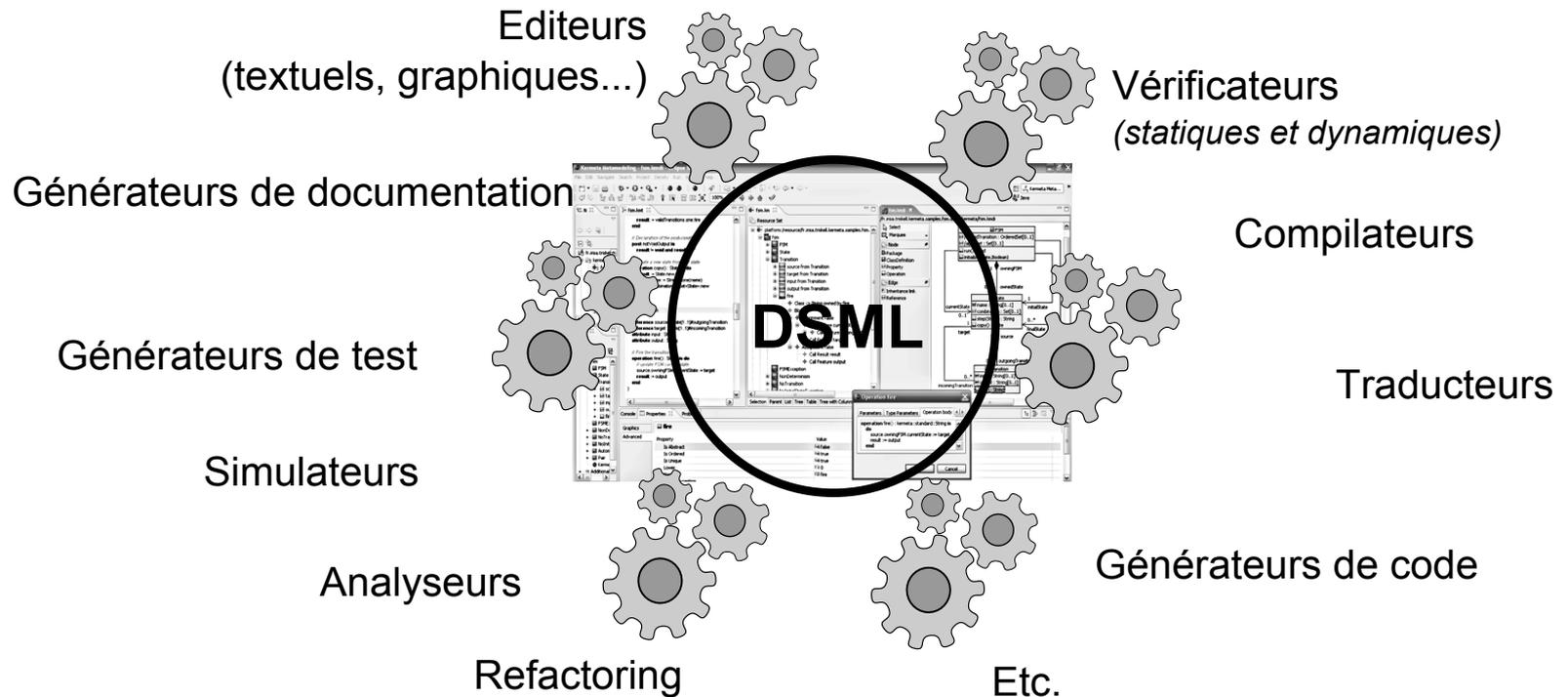


# Modeling and Metamodeling



# Metamodeling

---



# MDE Principles

---

1. Everything relevant to the development process is a model
2. All the meta-models can be written in a language of a unique meta-meta-model
  - Same M3 helps Interoperability of tools defined at M2
3. A development process can be modeled as a partially ordered set of model transformations, that take models as input and produce models as output

# Consequences

---

1. Models are aspect oriented. Conversely: Aspects are models
  2. Transformations are aspects weavers. They can be modeled.
  3. Every meta-model defines a domain specific language
  4. Software development has two dimensions: M1-model development and M2-transformation development
- Related: Software Language Engineering

# INGÉNIERIE DIRIGÉE PAR LES MODÈLES

## Des concepts à la pratique

*Maîtrisez la complexité de vos développements logiciels !*

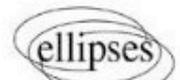
*Une approche didactique et pragmatique pour les étudiants et ingénieurs dans l'ingénierie du logiciel, et pour les responsables de projets informatiques.*

<http://www.amazon.fr/dp/2729871969>



Jean-Marc Jézéquel  
Benoît Combemale  
Didier Vojtisek

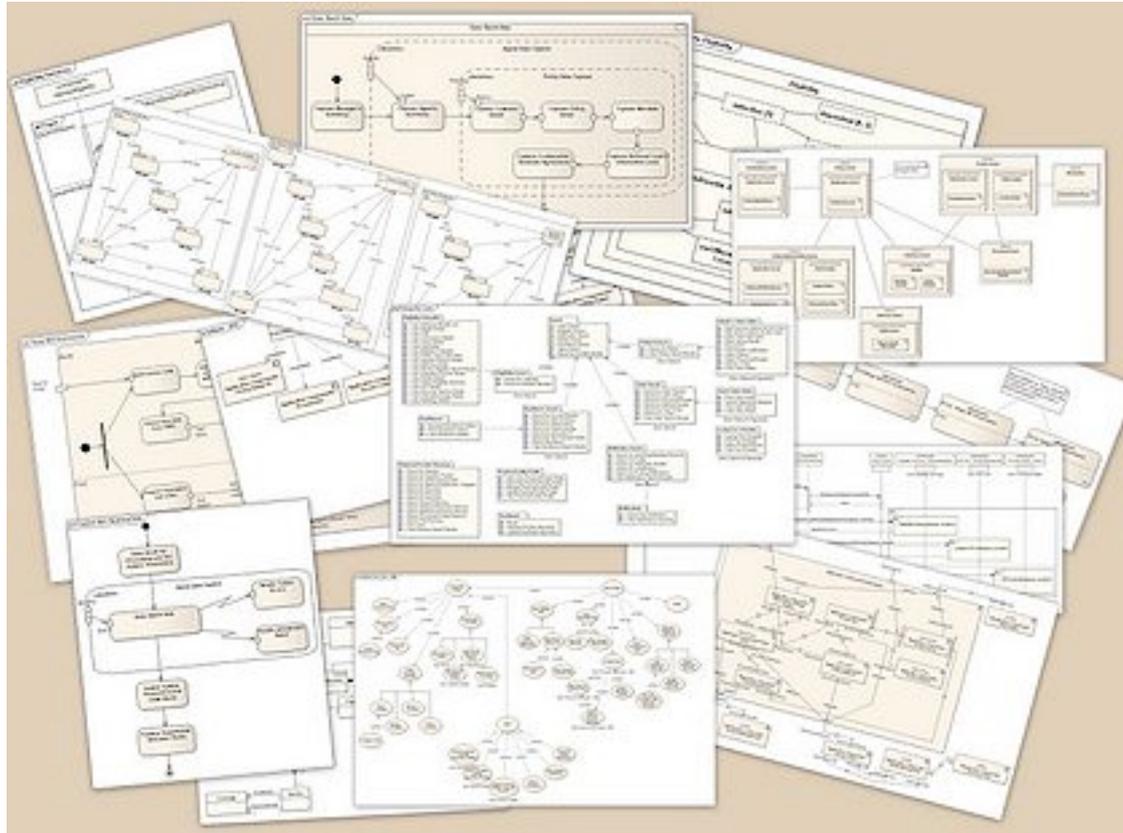
L'INGÉNIERIE DIRIGÉE  
par les MODÈLES  
Des concepts  
à la pratique



# Challenges

---

- Language definition problems (Q/V/T)
  - Expressive, easy to use language(s) for transformations
- Technological Issues
  - Tool set, interoperability...
- Software Engineering Issues
  - From requirements to tests and SCM
- Theoretical issues
  - A transformation is a *mapping* between semantic domains
    - Beyond Code Generation: Proof, Performance Evaluation, Test Synthesis...
  - Compositional weaving, cf. Aspect Oriented Software Dvp



Cf. [http://en.wikipedia.org/wiki/Unified\\_Modeling\\_Language](http://en.wikipedia.org/wiki/Unified_Modeling_Language)



