Modélisation UML

3IFA-UML

Stefan Duffner

INSA de Lyon, département Informatique

2024

Plan du cours

Pour en savoir plus...

- **Sur la modélisation en général :**
  - *Modèles et Métamodèles*
    Guy Caplat
- **Sites web :**
  - http://www.omg.org/uml (méta-modèle de référence d'UML)
  - http://www.uml-diagrams.org/
- **Sur UML et la modélisation objet :**
  - *Modélisation Objet avec UML*
    Pierre-Alain Muller, Nathalie Gaertner
    → **Chapitre sur la notation téléchargeable sur le site d'Eyrolles**
- **Sur les design patterns et la conception orientée objet :**
  - *UML 2 et les design patterns*
    Craig Larman
  - *Tête la première : Design Patterns*
    Eric Freeman & Elizabeth Freeman
- **...et plein d'autres ouvrages à la BMC !**

Outils de conception

Applications PC

- StarUML (WhiteStarUML)
- UMLet
- UML Designed (umldesigner.org)
- ObjectAid (Eclipse)
- Umple
- dia
- Violet
- Visual Studio (Enterprise) (Windows)
- Umbrello (Linux, KDE)

En ligne

- draw.io ($\equiv$ www.diagrameditor.com)
- umletino.com
- creately.com
- yuml.me
- online.visual-paradigm.com
- lucidchart.com
- planttext.com
- Umple Online

UML $\rightarrow$ code : compliqué et assez peu utilisé (à part pour le squelette)
code $\rightarrow$ UML : quelques outils font du “reverse engineering”

Plan du cours

Qu'est-ce qu'un modèle ?

Modèle = Objet conçu et construit (artefact) :

- Pour représenter un sujet d'études — **Représentativité**

  Exemple de sujet : les circuits électriques
- S'appliquant à plusieurs cas de ce sujet d'étude — **Généricité**

  Exemple de cas : des mesures (tension, intensité, ...) sur des circuits
- Incarnant un point de vue sur ces cas — **Abstraction**

  Exemple de point de vue : $U = RI$
  $\rightarrow$ Abstraction de la longueur des fils, la forme du circuit, ...

Un même sujet d'études peut avoir plusieurs modèles

$\rightarrow$ Chaque modèle donne un point de vue différent sur le sujet

Langages de modélisation

Langages utilisés pour exprimer un modèle :

- Langues naturelles : qui évoluent hors du contrôle d'une théorie
  Ex : Français, Anglais, ...
- Langages artificiels : conçus pour des usages particuliers
  - Langages formels : syntaxe définie par une grammaire
    Ex : Logique, langages informatique (C, Java, SQL, ...), ...

Pouvoir d'expression d'un langage :

$\rightarrow$ Ensemble des modèles que l'on peut exprimer

- Le choix du langage influence la conception du modèle...
  ...et donc la perception du sujet d'études !

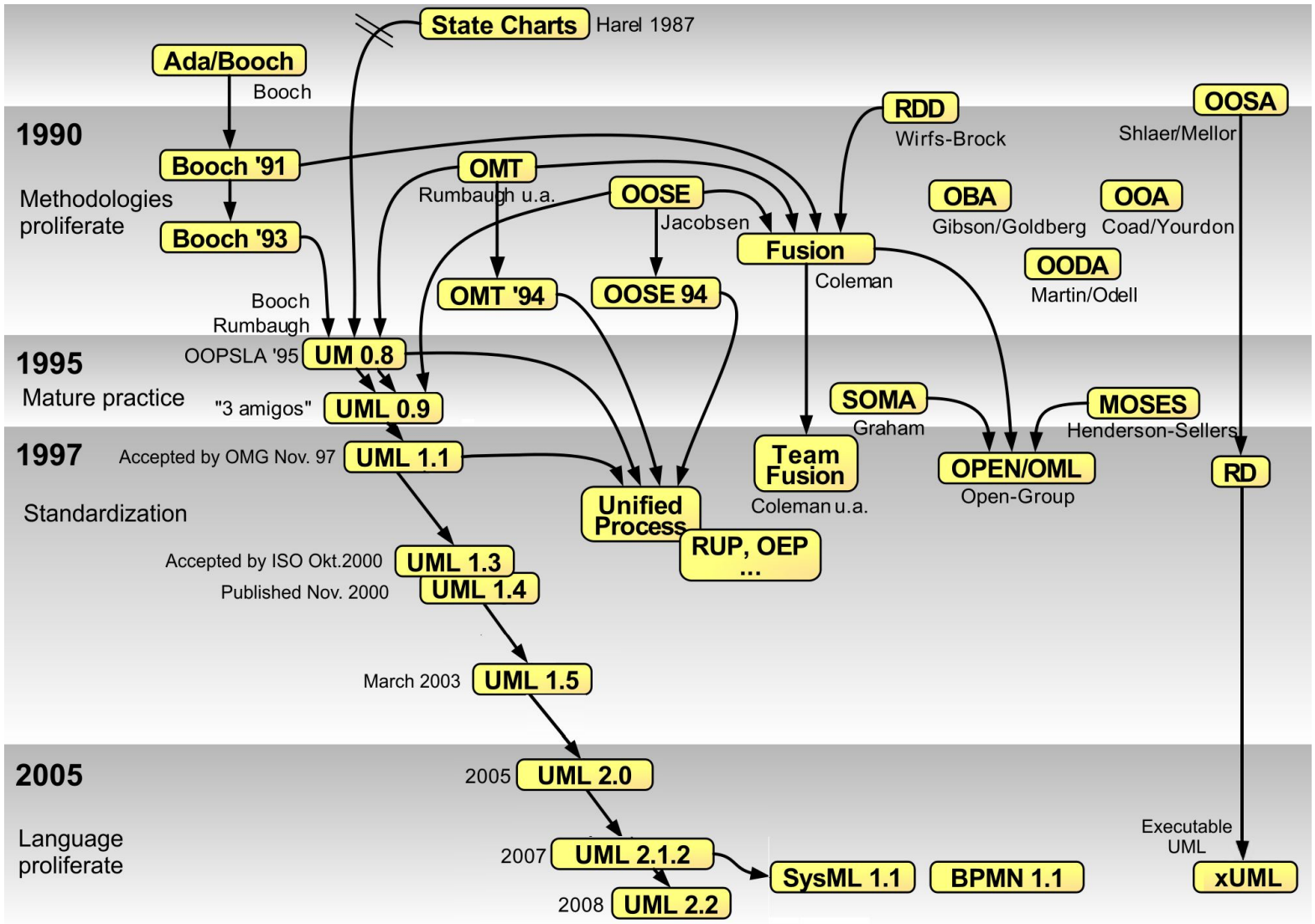
Interprétation d'un langage :

$\rightarrow$ Procédure pour comprendre un modèle (Sémantique)

- Modèle ambigü : Plusieurs interprétations différentes possibles
- Modèle exécutable : Interprétation exécutable par une machine

Plan du cours

Historique d'UML



[By Guido Zockoll, Axel Scheithauer, Marcel Douwe Dekker]

UML et l'OMG

OMG = Object Management Group (www.omg.org) :

- Fondé en 1989 pour standardiser et promouvoir l'objet
- Version 1.0 d'UML (Unified Modeling Language) en janvier 1997
- Version 2.5.1 en décembre 2017

Définition d'UML selon l'OMG :

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

L'OMG définit le méta-modèle d'UML

Attention : UML est un langage... pas une méthode

3 façons d'utiliser UML selon [Fowler 2003]

Mode esquisse (méthodes Agile) :

- Diagrammes tracés à la main, informels et incomplets

→ Support de communication pour concevoir les parties critiques

Mode plan :

- Diagrammes formels relativement détaillés
- Annotations en langue naturelle

→ Génération d'un squelette de code à partir des diagrammes
→ Nécessité de compléter le code pour obtenir un exécutable

Mode programmation (Model Driven Architecture / MDA) :

- Spécification complète et formelle en UML

→ Génération automatique d'un exécutable à partir des diagrammes
→ Limité à des applications bien particulières
→ Un peu utopique (...pour le moment ?)

Différents modèles UML

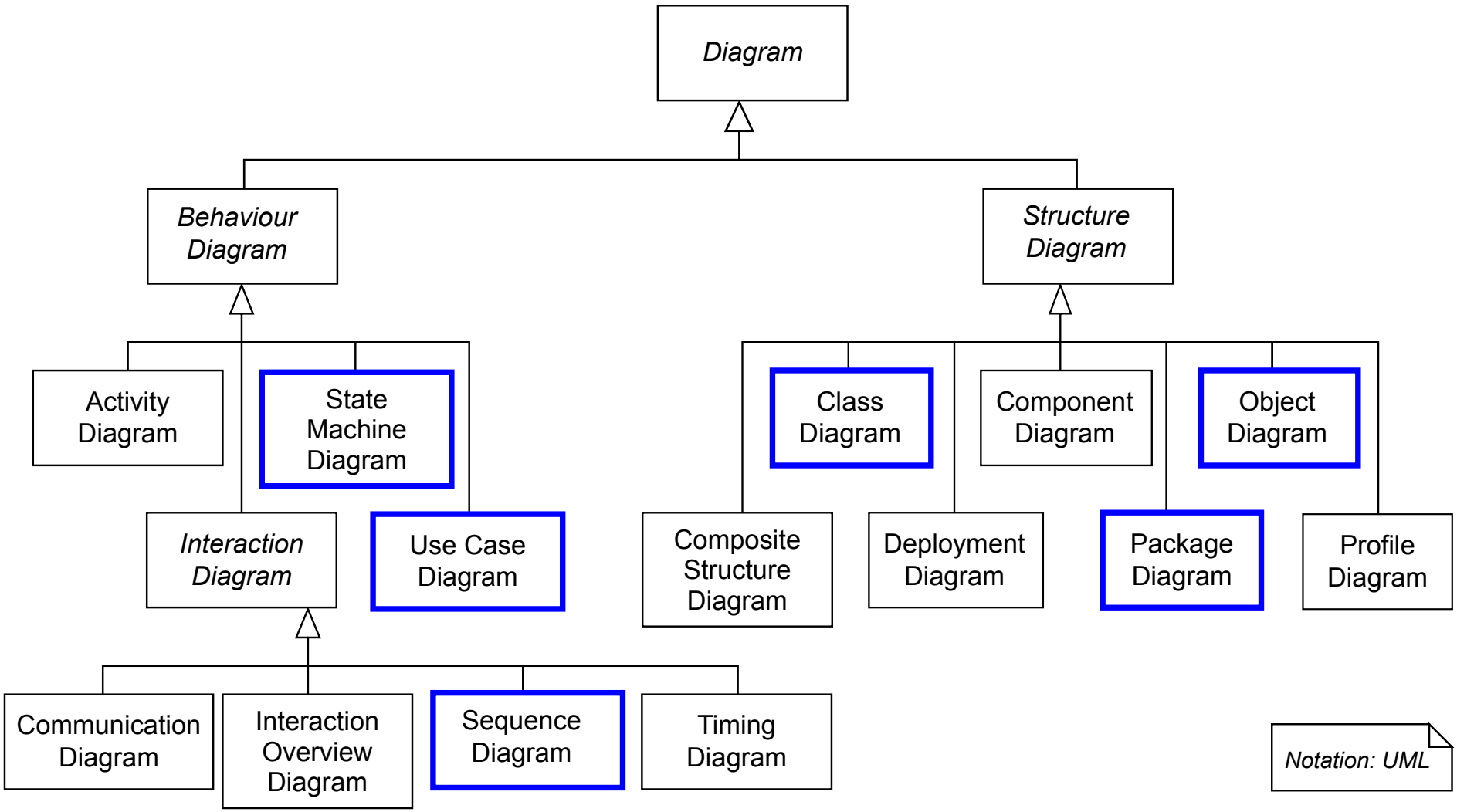
UML peut être utilisé pour définir de nombreux modèles :

- Modèles descriptifs vs prescriptifs
  - Descriptifs → Décrire l'existant (domaine, métier)
  - Prescriptifs → Décrire le futur système à réaliser
- Modèles destinés à différents acteurs
  - Pour l'utilisateur → Décrire le quoi
  - Pour les concepteurs/développeurs → Décrire le comment
- Modèles statiques vs dynamiques
  - Statiques → Décrire les aspects structurels
  - Dynamiques → Décrire comportements et interactions

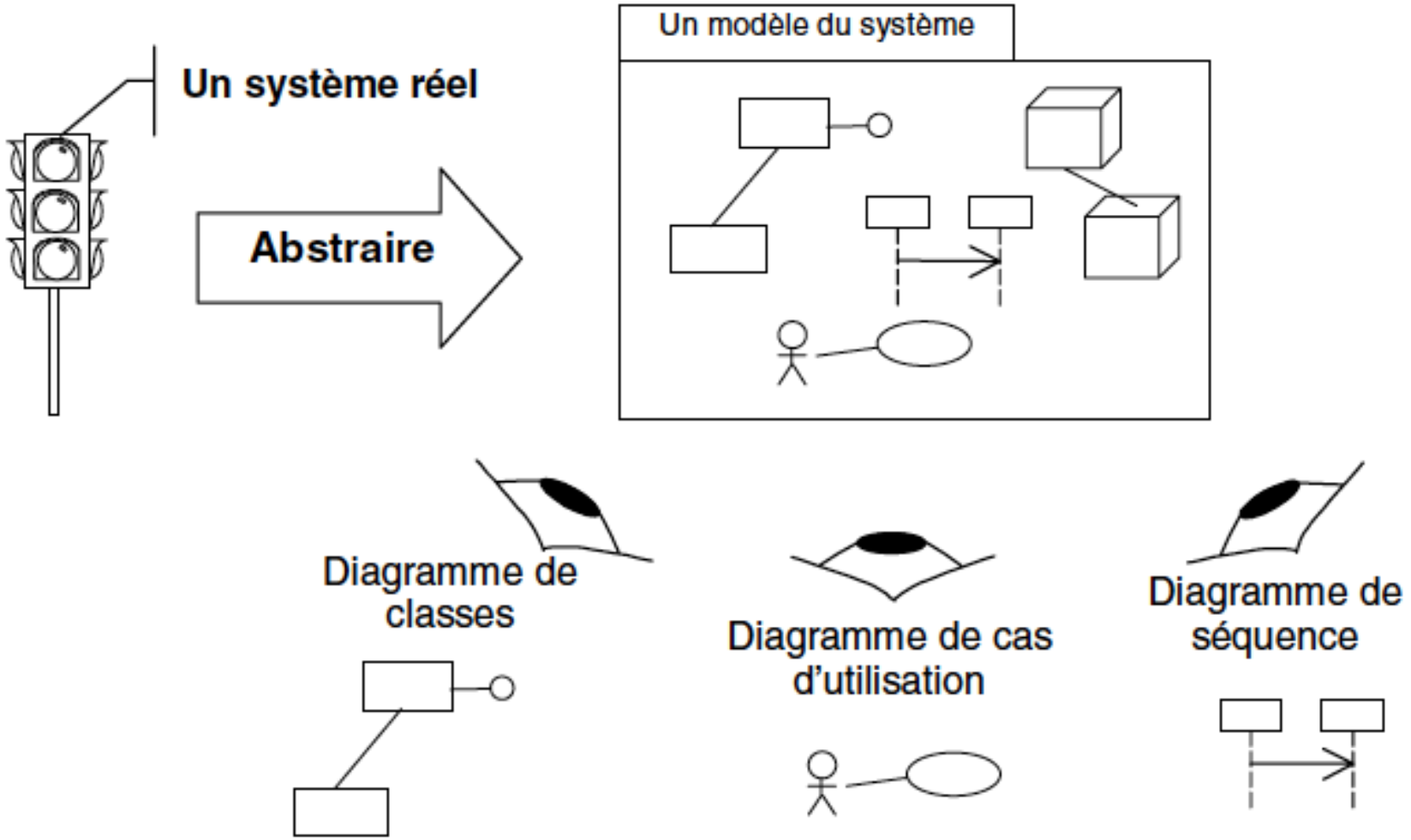
Les modèles sont décrits par des diagrammes (des graphes)

→ Chaque diagramme donne un point de vue différent sur le système

14 types de diagrammes d'UML 2.5



Diagrammes UML : Points de vue sur le système



[Image empruntée à Muller et Gaertner]

Notations communes à tous les diagrammes (1/2)

Stéréotypes et mots-clés :

- Définition d'une utilisation particulière d'éléments de modélisation
  $\rightarrow$ Interprétation (sémantique) particulière
- Notation : «`nomDuStéréotype`» ou `{nomDuMotClé}`
- Nombreux stéréotypes et mots-clés prédéfinis : «`interface`», «`invariant`», «`create`», «`actor`», `{abstract}`, `{bind}`, `{use}`...

Valeurs marquées :

- Ajout d'une propriété à un élément de modélisation
- Notation : `{` $\texttt{nom}_1$ `=` $\texttt{valeur}_1$`,` `...,` $\texttt{nom}_n$ `=` $\texttt{valeur}_n$ `}`
- Types prédéfinies (ex. : `payé : bool`)
  ou personnalisées (ex. : `auteur : chaîne`)
- Type + valeur (ex. : `payé : bool = true`)

Notations communes à tous les diagrammes (2/2)

Commentaires :

- Information en langue naturelle
- Notation : o- - - - - - - - - - - - - - - - - -

Relations de dépendance :

- Notation : `[source]- - - - - - - - - >[cible]`
  → Modification de la source peut impliquer une modification de la cible
- Nombreux stéréotypes prédéfinis : «`bind`», «`realize`», «`use`», «`create`», «`call`», ...

Plan du cours

Point de vue statique sur le système

Décrire la structure du système en termes de :

- Composants du système
  → Objets, Classes, Paquetages, Composants, ...
- Relations entre ces composants
  → Spécialisation, Association, Dépendance, ...

→ Pas de facteur temps

Différents diagrammes statiques que nous allons voir :

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

Plan du cours

Pourquoi une structuration orientée objet ?

Universalité du paradigme

- Réduire le décalage entre monde réel et logiciel

→ Objets réels ⇒ Objets conceptuels ⇒ Objets logiciels

Réutilisabilité et évolutivité facilitées par différents mécanismes

- Encapsulation, Modularité, Abstraction, Polymorphisme, Héritage

→ Faible couplage inter-objets / Forte cohésion intra-objet

Paradigme qui arrive à maturité

- Bibliothèques de classes, Design patterns, UML, Méthodologies de développement (UP, Agile, XP, ...), ...

→ Environnements de développement intégrés (IDE)

Qu'est-ce qu'un objet ?

Objet = Etat + Comportement + Identité

Etat d'un objet :

- Ensemble de valeurs décrivant l'objet
  → Chaque valeur est associée à un attribut (propriété)
  → Les valeurs sont également des objets (⇒ liens entre objets)

Comportement d'un objet :

- Ensemble d'opérations que l'objet peut effectuer
- Chaque opération est déclenchée par l'envoi d'un message
  → Exécution d'une méthode

Identité d'un objet :

- Permet de distinguer les objets indépendamment de leur état
  → 2 objets différents peuvent avoir le même état
- Attribuée implicitement à la création de l'objet
  → L'identité d'un objet ne peut être modifiée

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

Classe = regroupement d'objets similaires (appelés instances)

- Toutes les instances d'une classe ont les mêmes attributs et opérations
  → Abstraction des caractéristiques non communes

Classes sans instance

- Classes abstraites :
  → Certaines opérations peuvent être abstraites/virtuelles (non définies)
- Interfaces :
  → Pas d'attribut et toutes les opérations sont abstraites/virtuelles

Relation de spécialisation/généralisation entre classes

- Une classe A est une spécialisation d'une classe B si tout attribut/opération de B est également attribut/opération de A
- Implémentation par héritage

Plan du cours

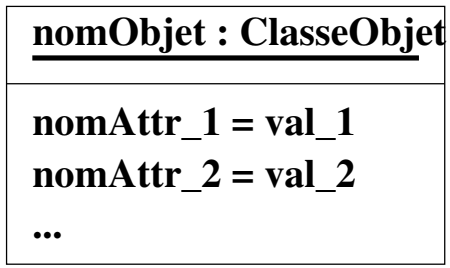
Diagrammes d'objets

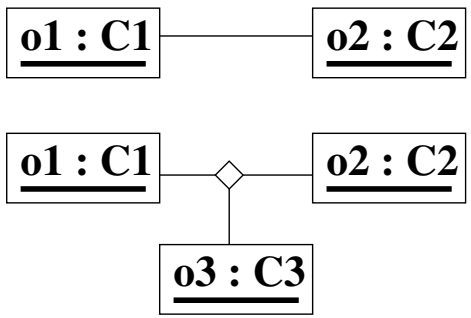
Objectif : Représenter les objets et leurs liens à un instant donné

- Utilisation : documenter des cas de test, analyser des exemples, ...

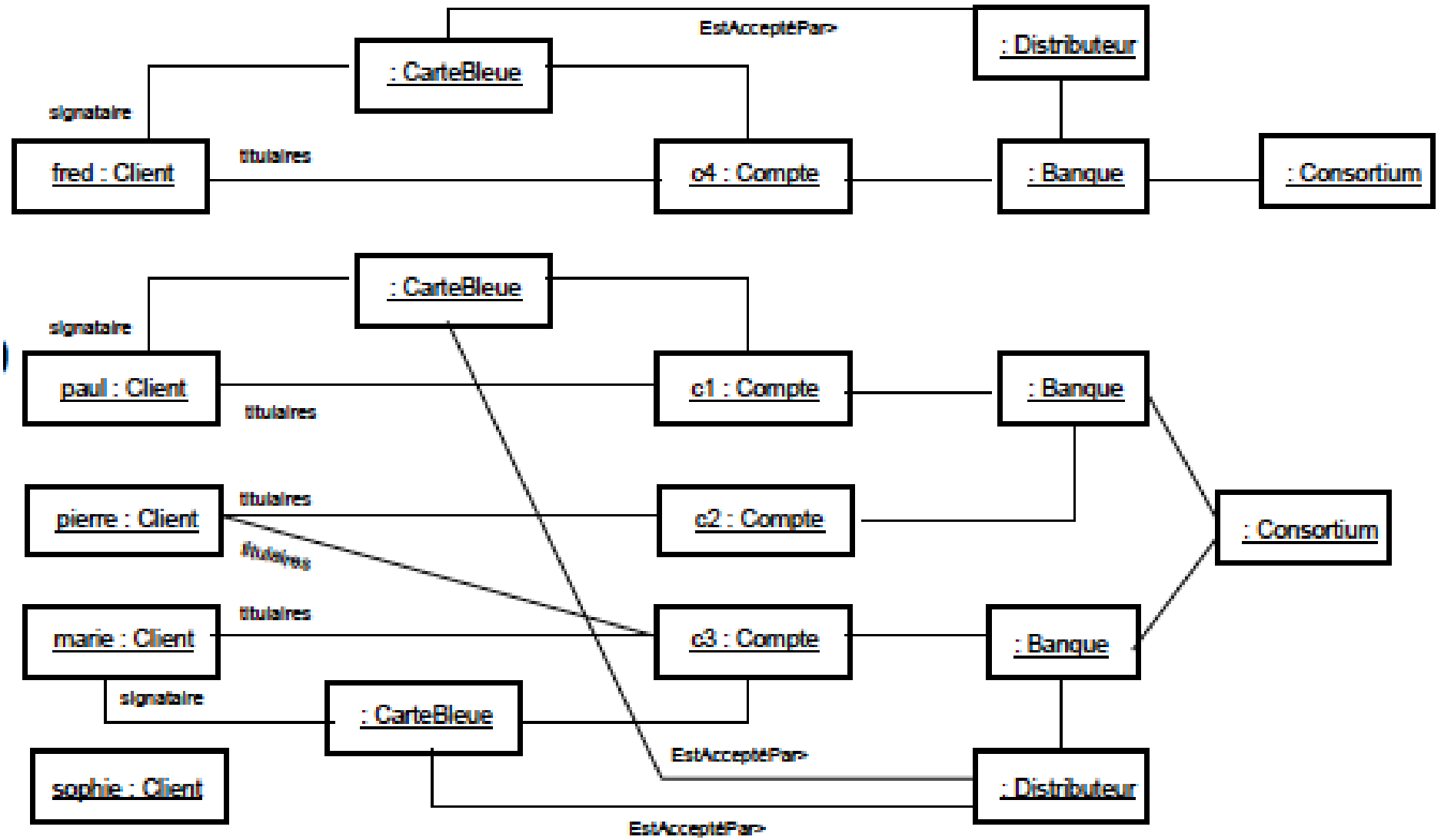
Moyen : Graphe

- Nœuds du graphe = Objets
  - Possibilité de supprimer le nom, la classe et/ou les attributs (objet anonyme, non typé ou d'état inconnu)
- Arêtes du graphe = Liens entre objets
  - Lien binaire : entre 2 objets
  - Lien n-aire : entre n objets
  - Possibilité de nommer les liens et les rôles
- Correspondance entre liens et attributs





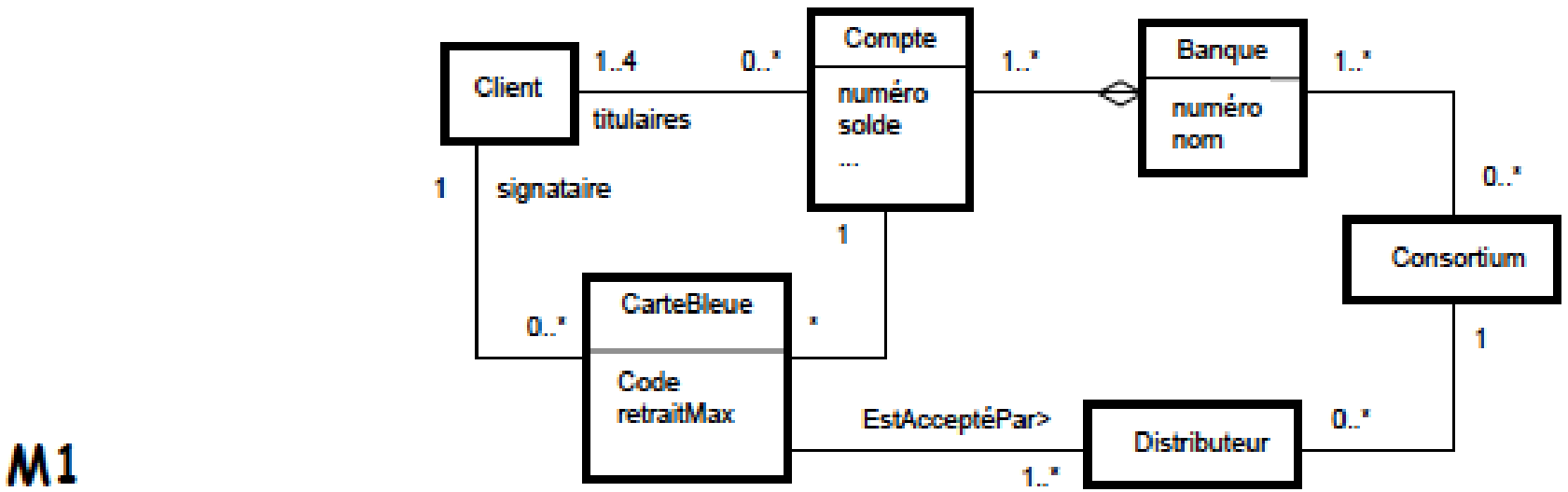
Exemple de diagramme d'objets



[Image empruntée à J.-M. Jezequel]

Plan du cours

Abstraction d'un ensemble de diagrammes d'objets



[Image empruntée à J.-M. Jezequel]

Diagrammes de classes

Un diagramme de classes est un graphe :

- Nœud du graphe = Classe
  $\rightarrow$ Abstraction d'un ensemble d'objets
- Arc du graphe = Relation entre des classes :
  - Relation d'association
    $\rightarrow$ Abstraction d'un d'ensemble de liens entre objets
  - Relation de généralisation / spécialisation
    $\rightarrow$ Factorisation de propriétés communes à plusieurs classes

Très nombreuses utilisations, à différents niveaux :

- Pendant la capture des besoins : Modèle du domaine
  $\rightarrow$ Classes = Objets du domaine
- Pendant la conception/implémentation : Modèle de conception
  $\rightarrow$ Classes = Objets logiciels

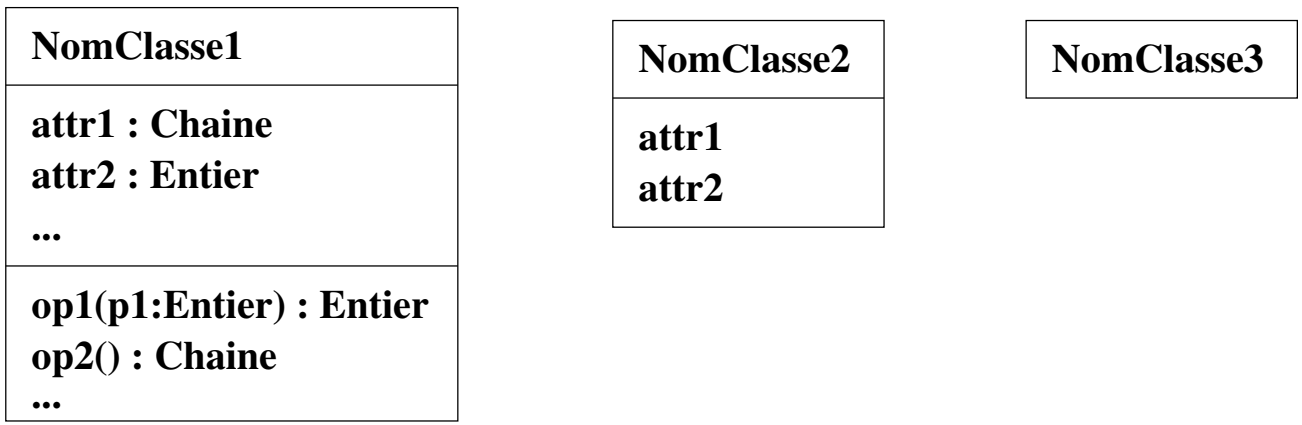
Représentation UML des classes

Rectangle composé de compartiments :

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

Différents niveaux de détail possibles :

→ Possibilité d'omettre attributs et/ou opérations



Représentation UML des attributs

- Format de description d'un attribut :

  `[Vis] Nom [Mult] [":" TypeAtt] ["=" Val] [Prop]`

  - `Vis` : `+` (public), `-` (privé), `#` (protégé), $\sim$ (package)
  - `Mult` : `[nbElt]` ou `[Min..Max]`
  - `TypeAtt` : type primitif (Entier, Chaîne, ...) ou classe
  - `Val` : valeur initiale à la création de l'objet
  - `Prop` : `{gelé}`, `{variable}`, `{ajoutUniquement}`, ...

- Attributs de classe (statiques) soulignés
- Attributs dérivés précédés de "/"

Exemples :

- `# onOff : Bouton`
- `- x : Réel`
- `coord[3] : Réel`
- `+ pi : réel = 3.14 {gelé}`
- `inscrits[2..8] : Personne`
- `/age : Entier`

Représentation UML des opérations

- Format de description d'une opération :

  `[Visibilité] Nom ["(" Arg ")"] [":" Type]`

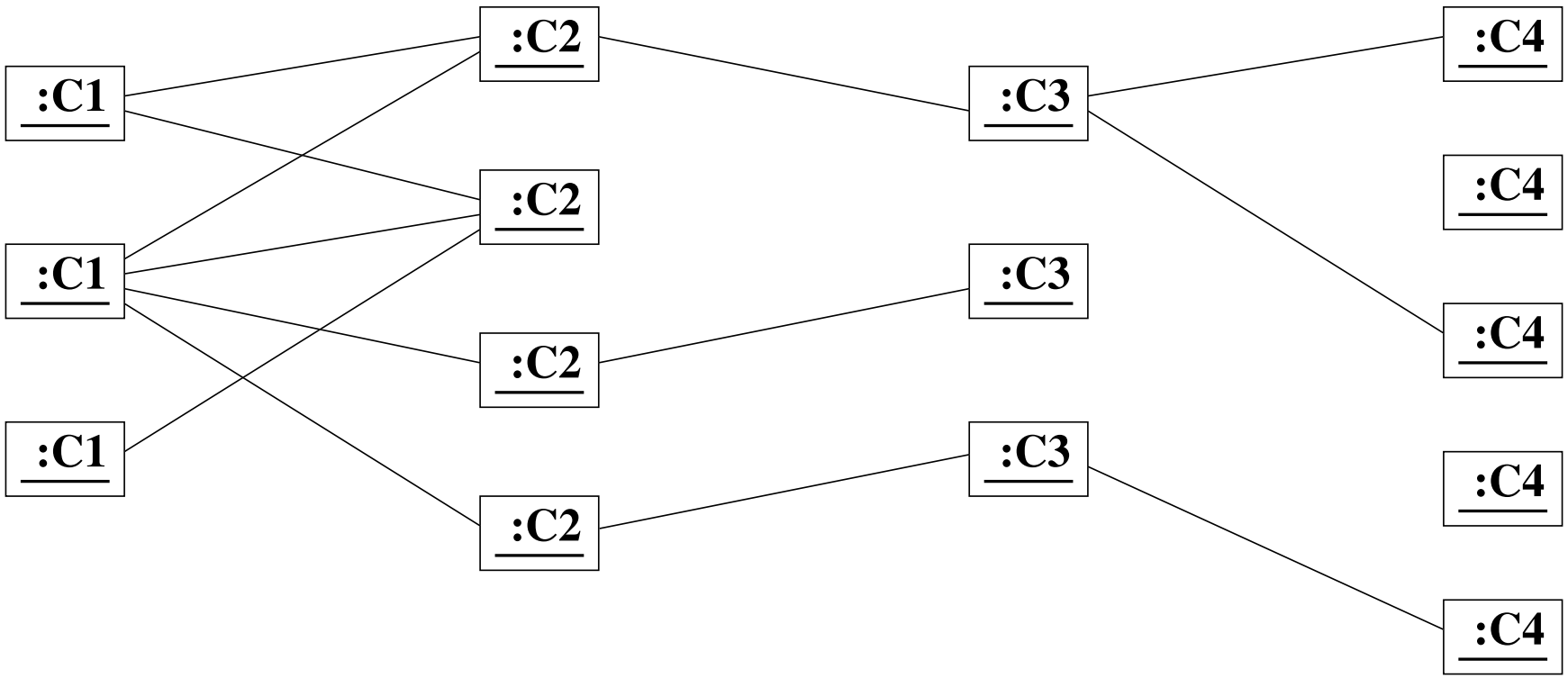
  - `Visibilité` : + (public), - (privé), # (protégé)
  - `Arg` : liste des arguments selon le format
    `[Dir] NomArgument : TypeArgument`
    où `Dir` = `in` (par défaut), `out`, ou `inout`
  - `Type` : type de la valeur retournée (type primitif ou classe)

- Opérations abstraites/virtuelles (non implémentées) en italique
- Opérations de classe (statiques) soulignées
- Possibilité de surcharger une opération :
  $\rightarrow$ même nom, mais paramètres différents
- Stéréotypes d'opérations : «`create`» et «`destroy`»

Associations entre classes

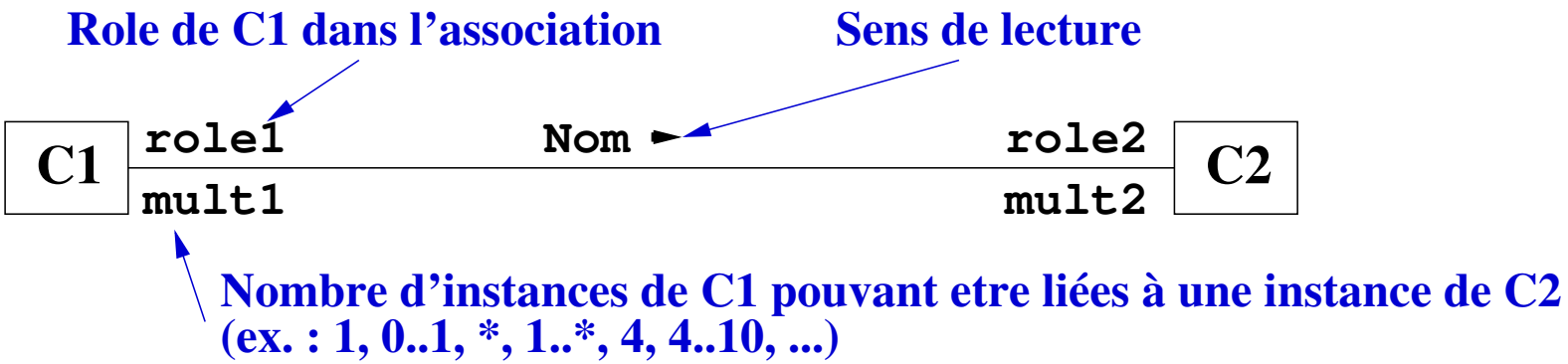
→ **Abstraction des relations définies par les liens entre objets**

Liens entre objets :



Associations entre classes d'objets :

Associations entre classes (1/2)

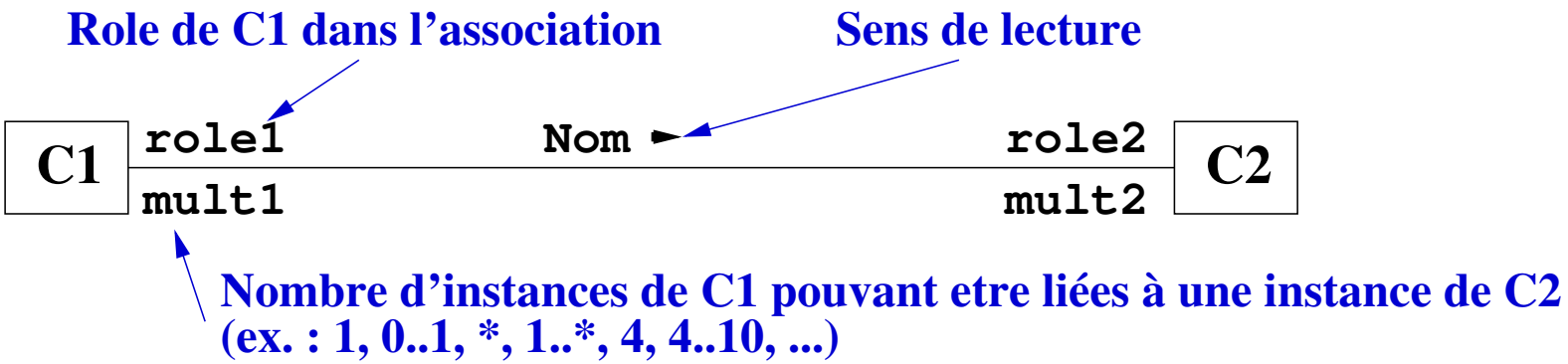


Interprétation en français :

- Un C1 `Nom` un C2 (ou un C2 `Nom` un C1 si sens de lecture inverse)
- Un C1 est `rôle1` d'un C2 et `mult1` C1 peuvent jouer ce rôle pour un C2
- Un C2 est `rôle2` d'un C1 et `mult2` C2 peuvent jouer ce rôle pour un C1

Exemple :

Associations entre classes (2/2)



Interprétation en langage de programmation orienté objet

- La classe C1 a un attribut de nom `rôle2`
  $\rightarrow$ Type = C2 si mult2 $\in \{1, 0..1\}$, ou collection de C2 sinon
- La classe C2 a un attribut de nom `rôle1`
  $\rightarrow$ Type = C1 si mult1 $\in \{1, 0..1\}$, ou collection de C1 sinon

Exemple :

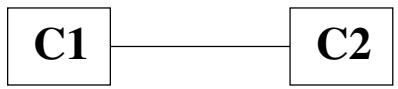
Navigabilité

Qu'est-ce que la navigabilité d'une association entre C1 et C2 ?

Capacité d'une instance de C1 (resp. C2) à accéder aux instances de C2 (resp. C1)

Par défaut :

Navigabilité dans les deux sens
→ C1 a un attribut de type C2 et C2 a un attribut de type C1



Spécification de la navigabilité :

Orientation de l'association
→ C1 a un attribut du type de C2, mais pas l'inverse

Attention :

Dans un diagramme de classes conceptuelles, toute classe doit être accessible à partir de la classe principale

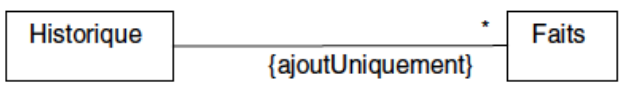
Attributs et contraintes sur les associations

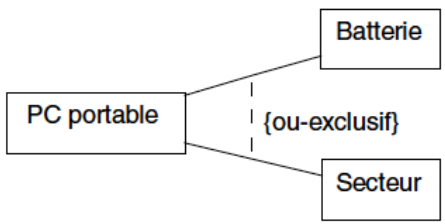
Attributs sur extrémités d'associations

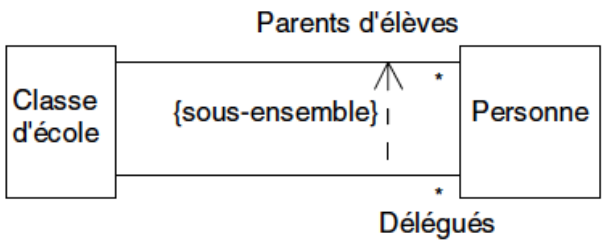
- `{variable}` : instance modifiable (par défaut)
- `{frozen}` : instance non modifiable
- `{addOnly}` : instances ajoutables mais non retirables (si mult. > 1)

Contraintes prédéfinies

- Sur une extrémité : `{ordered}, {unique}, ...`
- Entre 2 associations : `{subset}, {xor}, ...`



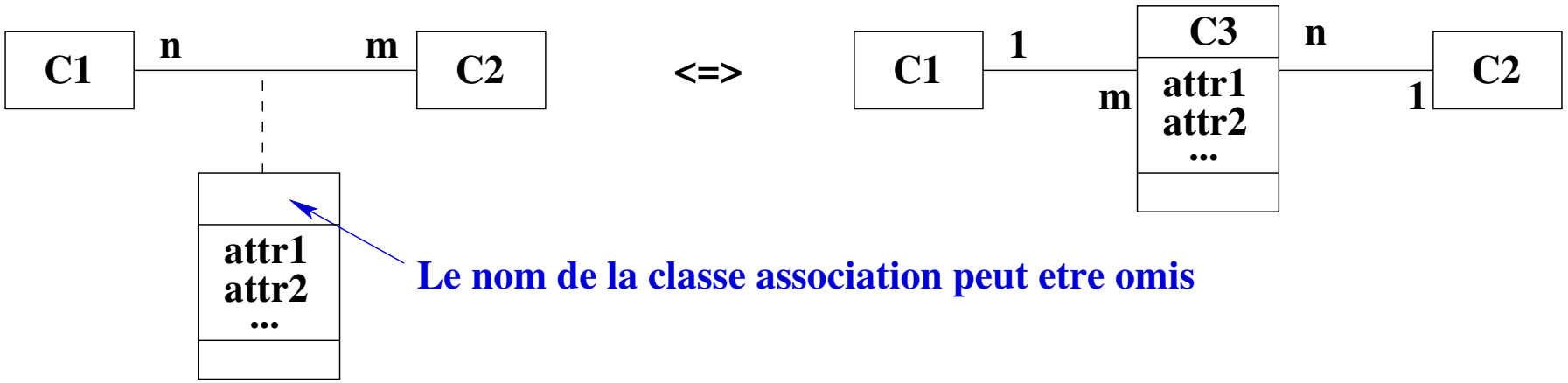




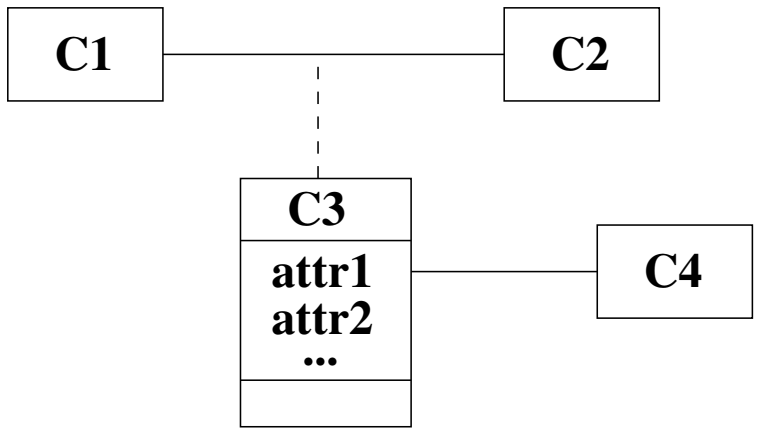
[images extraites de Muller et Gaertner]

Classes-associations

Association attribuée :

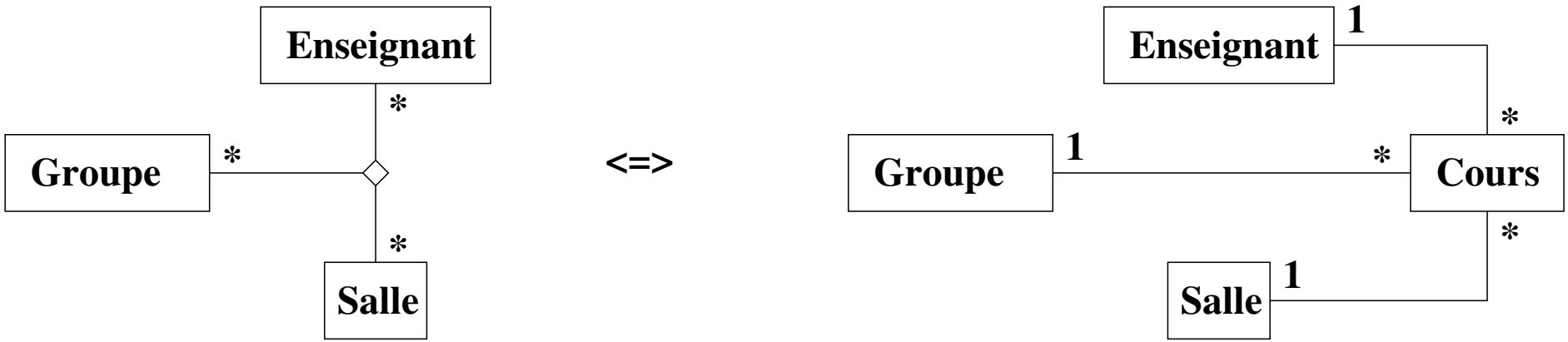


Une classe association peut participer à d'autres associations :

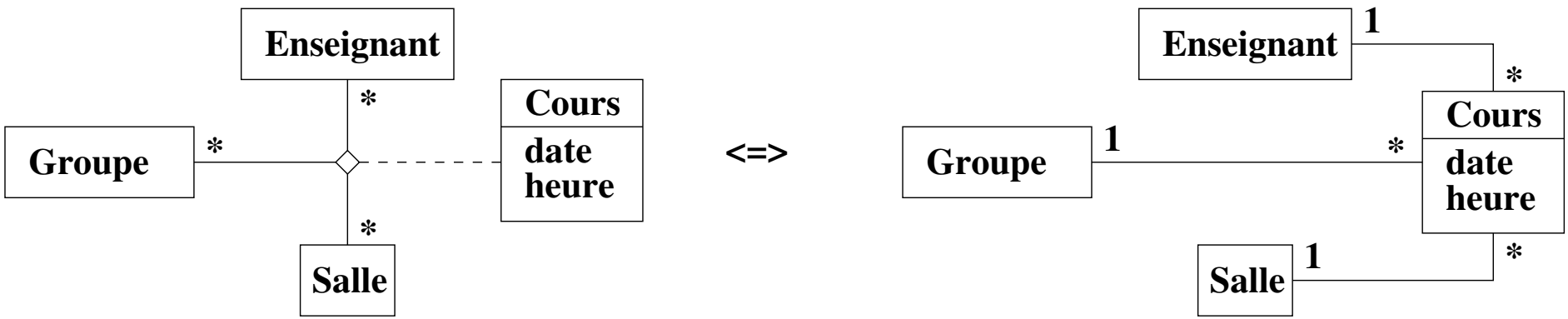


Associations n-aires

Associations entre n classes (avec $n > 2$)

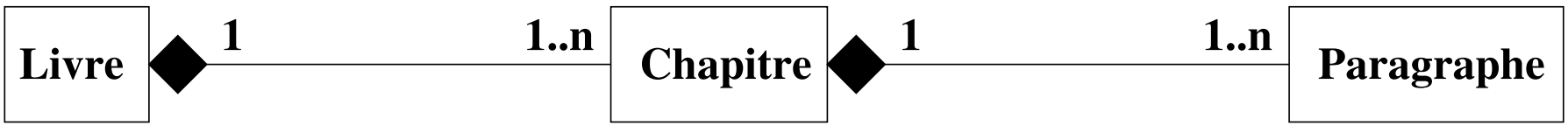


Classes-Associations n-aire



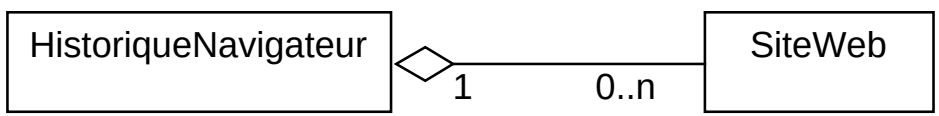
Associations particulières : composition et agrégation

Composition :



- Relation transitive et antisymétrique
- La création (copie, destruction) du composite (container) implique la création (copie, destruction) de ses composants
- Un composant appartient à au plus un composite

Agrégation :



- Simple regroupement de parties dans un tout

Généralisation et Héritage

Niveau conceptuel : Généralisation

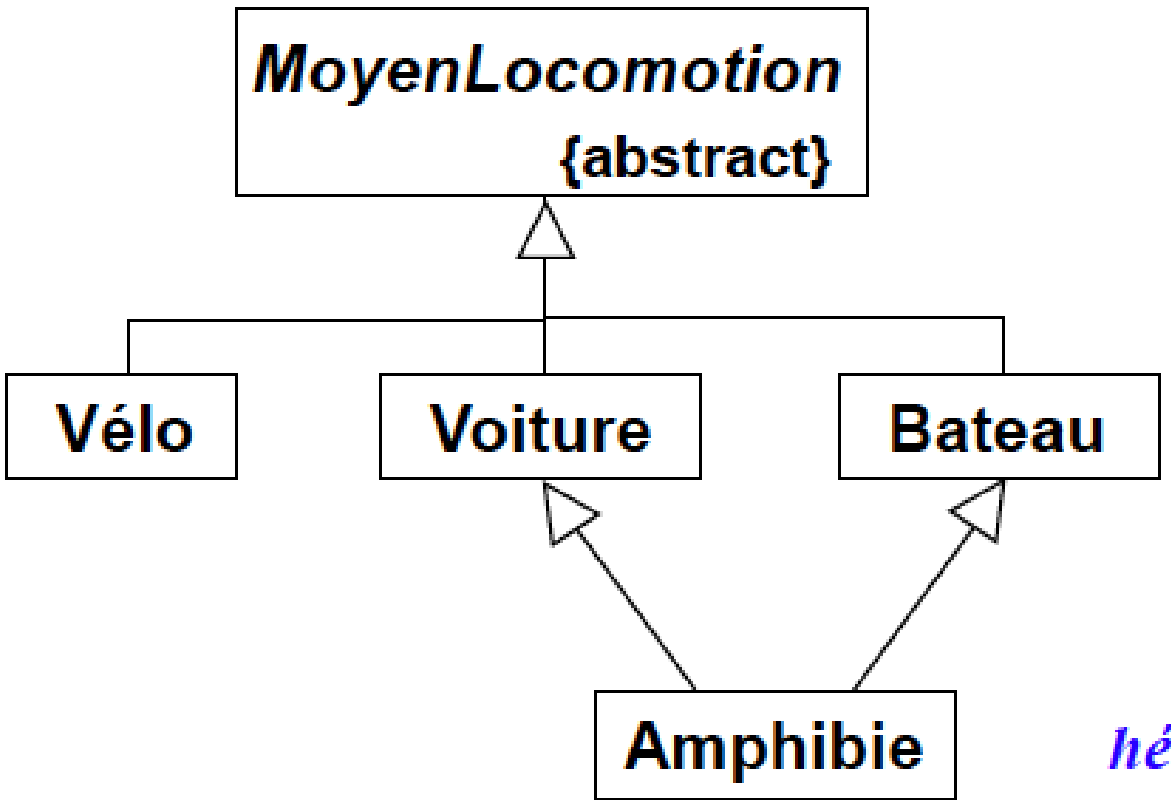
- Relation transitive, non réflexive, et non symétrique
- La sous-classe “est-une-sorte-de” la super classe
  $\rightarrow$ Toute instance de la sous-classe est instance de la super classe

Niveau implémentation : Héritage

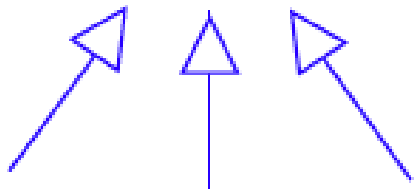
- Mécanisme proposé par les langages de programmation Objet
- "B hérite de A" signifie que B possède :
  - Toutes les propriétés de A (attributs, op., assoc., contraintes)
    $\rightarrow$ Possibilité de redéfinir les opérations de la sous-classe
    $\rightarrow$ Polymorphisme
  - Ainsi que des nouvelles propriétés qui lui sont propres

$\rightarrow$ Permet de factoriser les propriétés communes à plusieurs classes
$\rightarrow$ Une opération définie pour A est accessible aux sous-classes de A

Héritage et généralisation : Exemple



notation équivalente



héritage multiple

Héritage vs Composition

Problème :

Modéliser le fait qu'il y a des voitures bleues, des voitures rouges et des voitures vertes.

Solution 1 : Héritage

- Créer une classe abstraite Voiture
- Créer 3 classes VoitureBleue, VoitureRouge et VoitureVerte qui héritent de Voiture

Solution 2 : Composition

- Créer une classe Voiture et une classe Couleur (énumération)
- Créer une association entre Voiture et Couleur

Comment représenter ces 2 solutions en UML ?

Héritage vs Composition

Problème :

Modéliser le fait qu'il y a des voitures bleues, des voitures rouges et des voitures vertes.

Solution 1 : Héritage

- Créer une classe abstraite Voiture
- Créer 3 classes VoitureBleue, VoitureRouge et VoitureVerte qui héritent de Voiture

Solution 2 : Composition

- Créer une classe Voiture et une classe Couleur (énumération)
- Créer une association entre Voiture et Couleur

Comment représenter ces 2 solutions en UML ?
Quelle solution choisissez-vous ?

Héritage vs Composition

Problème :

Modéliser le fait qu'il y a des voitures bleues, des voitures rouges et des voitures vertes.

Solution 1 : Héritage

- Créer une classe abstraite Voiture
- Créer 3 classes VoitureBleue, VoitureRouge et VoitureVerte qui héritent de Voiture

Solution 2 : Composition

- Créer une classe Voiture et une classe Couleur (énumération)
- Créer une association entre Voiture et Couleur

Comment représenter ces 2 solutions en UML ?
Quelle solution choisissez-vous ?
Et si on veut modéliser le fait qu'il y a des personnes hommes et des personnes femmes ?

Héritage vs Délégation

Problème :

Appeler dans une classe B une opération `op()` d'une classe A ?

Solution 1 : Héritage

- Faire hériter B de A
- `op()` peut être appelée depuis n'importe quelle instance de B

Solution 2 : Délégation

- Ajouter une association de B vers A
  $\rightarrow$ Ajouter dans B un attribut `a` de type `A`
- `a.op()` peut être appelée depuis n'importe quelle instance de B

Comment représenter ces 2 solutions en UML ?

Héritage vs Délégation

Problème :

Appeler dans une classe B une opération `op()` d'une classe A ?

Solution 1 : Héritage

- Faire hériter B de A
- `op()` peut être appelée depuis n'importe quelle instance de B

Solution 2 : Délégation

- Ajouter une association de B vers A
  $\rightarrow$ Ajouter dans B un attribut `a` de type `A`
- `a.op()` peut être appelée depuis n'importe quelle instance de B

Comment représenter ces 2 solutions en UML ?
Quelle solution choisissez-vous ?

Héritage vs Délégation

Problème :

Appeler dans une classe B une opération `op()` d'une classe A ?

Solution 1 : Héritage

- Faire hériter B de A
- `op()` peut être appelée depuis n'importe quelle instance de B

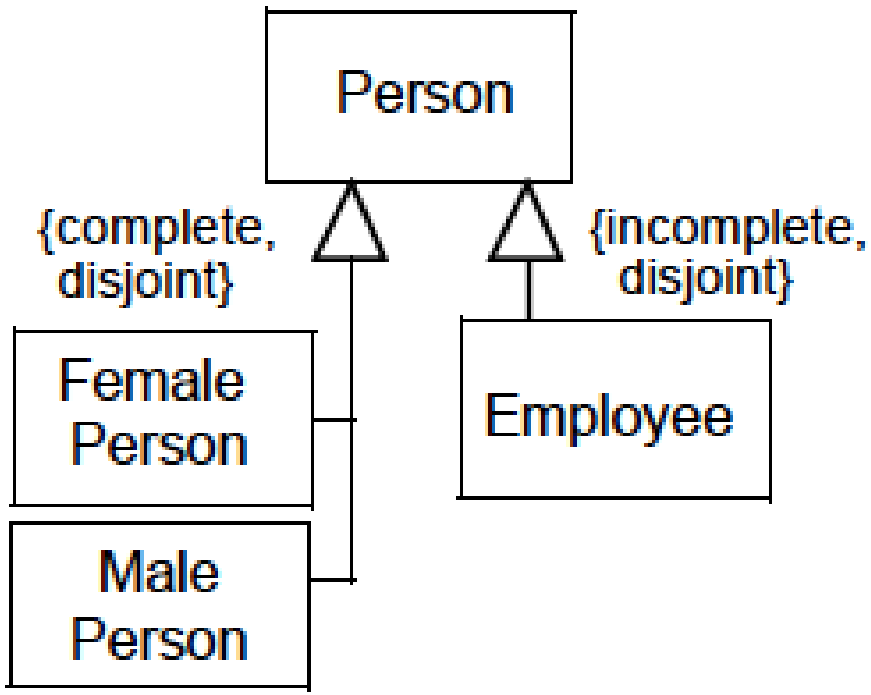
Solution 2 : Délégation

- Ajouter une association de B vers A
  → Ajouter dans B un attribut `a` de type `A`
- `a.op()` peut être appelée depuis n'importe quelle instance de B

Comment représenter ces 2 solutions en UML ?
Quelle solution choisissez-vous ?
→ On y reviendra avec les Design Patterns...

Contraintes sur les relations de généralisation

- `{disjoint}` ou `{overlapping}`
- `{complete}` ou `{incomplete}`
  → `{disjoint,complete}` ⇒ Partition
- `{leaf}` ou `{root}`

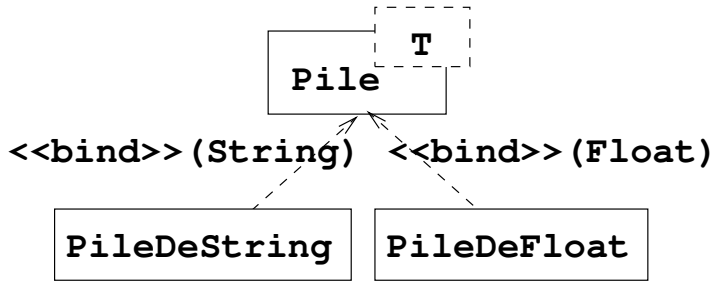


[Image empruntée à www.omg.org]

Classes génériques (templates)

Qu'est-ce qu'une classe générique ?

- Classe paramétrée par d'autres classes
  → Factorisation de code



Exemple de code C++

```
template <class T> class Pile{
  public:
    Pile() { ... }
    void empile(T e) { ... }
    ...
  private: ...
};
...
Pile<float> p1;
Pile<string> p2;
p1.empile(2.5);
p2.empile('a');
...
```

Exemple de code Java :

```
public class Pile<T> {
  public Pile() { ... }
  public void empile(T e) { ... }
  ...
  private ...
};
...
Pile<Float> p1 = new Pile<Float>();
Pile<String> p2 = new Pile<String>();
p1.empile(2.5);
p2.empile('a');
...
```

Classes abstraites

Qu'est-ce qu'une classe abstraite ?

- Classe qui ne peut être instanciée
  - Contient des opérations non définies :
    → `abstract` (Java) ou `virtual` (C++)
  - Doit être spécialisée en une ou plusieurs classes non abstraites
- Notation : propriété `{abstract}` ou nom en italique

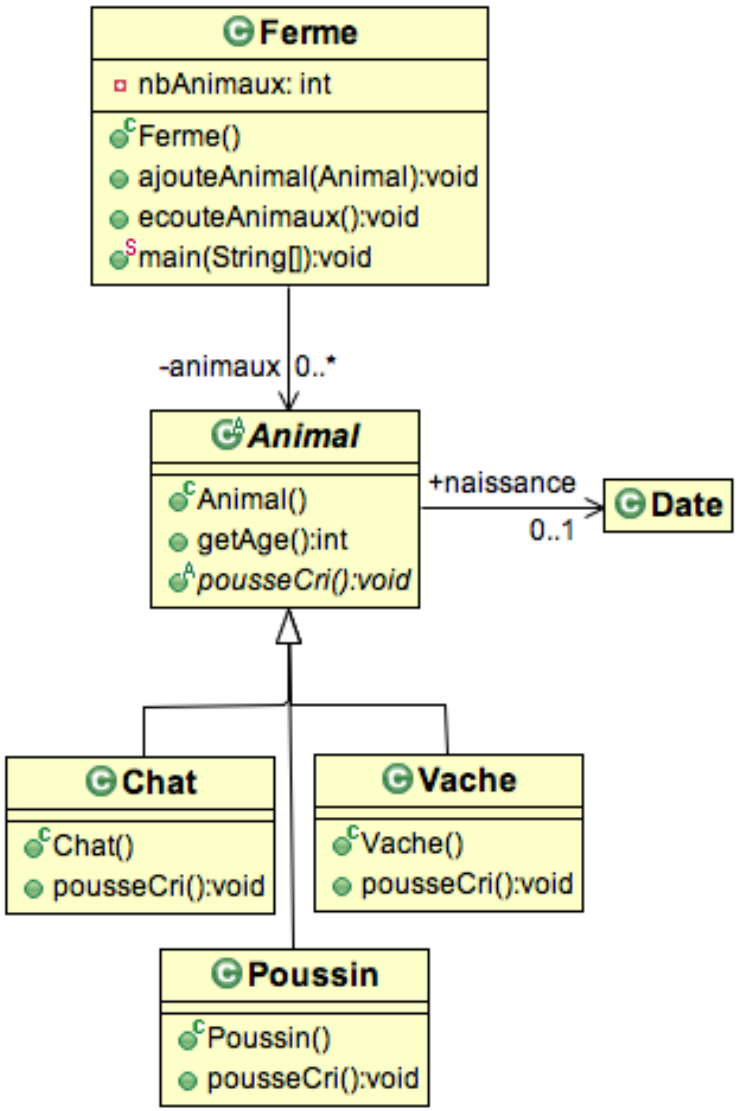
Pourquoi des classes abstraites ?

- Spécifier un comportement commun à plusieurs classes
- Manipuler des instances de classes différentes de façon uniforme
  → Polymorphisme

Bonne pratique :

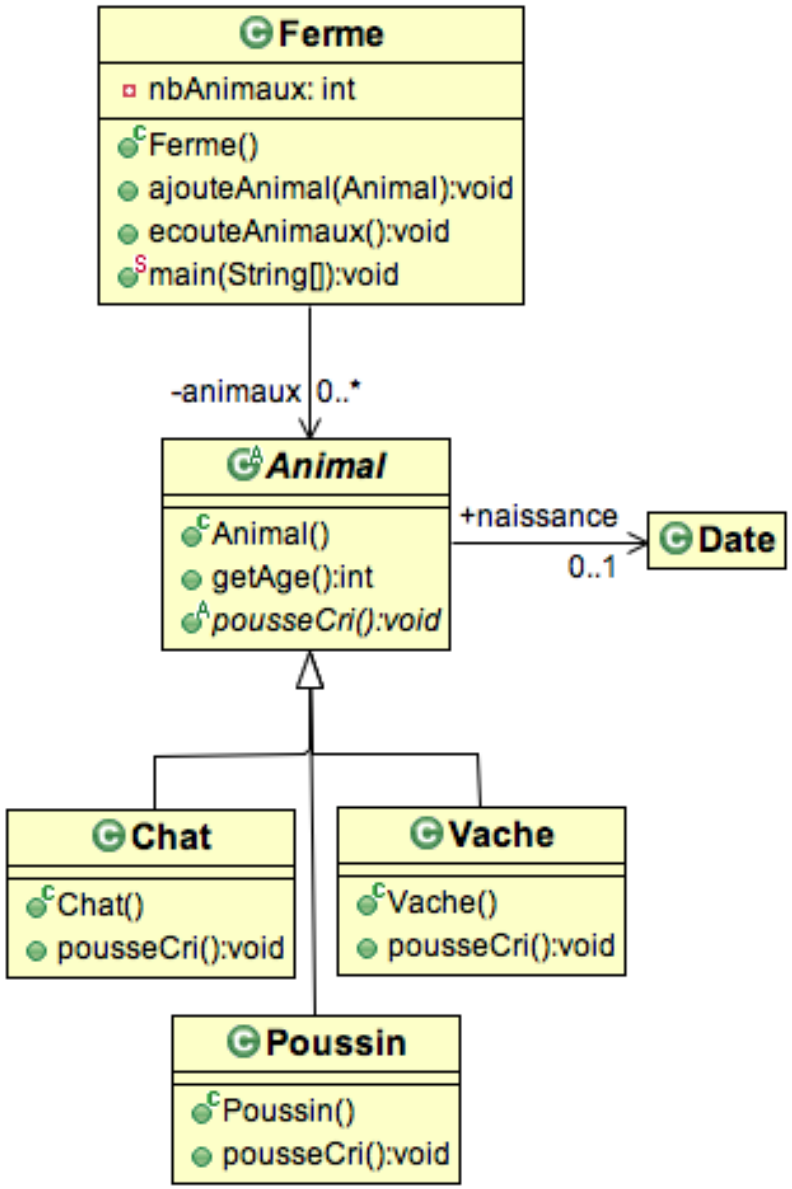
Dans une hiérarchie d'héritage, les classes qui ne sont pas des feuilles sont généralement abstraites

Classes abstraites et polymorphisme / Ex. C++




```
class Animal{
public:
    Date naissance;
    int getAge(){...}
    virtual void pousseCri() = 0;
};
class Poussin: public Animal{
public:
    void pousseCri(){cout << "Piou" << endl;}
};
class Ferme{
private:
    Animal* animaux[];
    int nbAnimaux;
public:
    Ferme(){...}
    void ajouteAnimal(Animal* a){
        animaux[nbAnimaux++] = a;
    }
    void ecouteAnimaux(){
        for (int i=0; i<nbAnimaux; i++)
            animaux[i]->pousseCri();
    }
};
int main(){
    Ferme f;
    f.ajouteAnimal(new Chat());
    f.ajouteAnimal(new Poussin());
    f.ajouteAnimal(new Vache());
    f.ecouteAnimaux();
}
```

Classes abstraites et polymorphisme / Ex. Java




```java
public abstract class Animal {
    public Date naissance;
    public Animal(){....}
    public int getAge(){....}
    public abstract void pousseCri();
}
public class Poussin extends Animal {
    public void pousseCri() {System.out.println("Piou");}
}
public class Ferme {
    private Animal[] animaux;
    private int nbAnimaux;
    public Ferme(){....}
    public void ajouteAnimal(Animal a){
        animaux[nbAnimaux++] = a;
    }
    public void ecouteAnimaux(){
        for (int i=0; i<nbAnimaux; i++)
            animaux[i].pousseCri();
    }
}
public static void main(String[] args) {
    Ferme f = new Ferme();
    f.ajouteAnimal(new Chat());
    f.ajouteAnimal(new Poussin());
    f.ajouteAnimal(new Vache());
    f.ecouteAnimaux();
}
```

Interface

Qu'est-ce qu'une interface ?

- Classe sans attribut dont toutes les opérations sont abstraites
  → Ne peut être instanciée
  → Doit être réalisée (implémentée) par des classes non abstraites
  → Peut hériter d'une autre interface

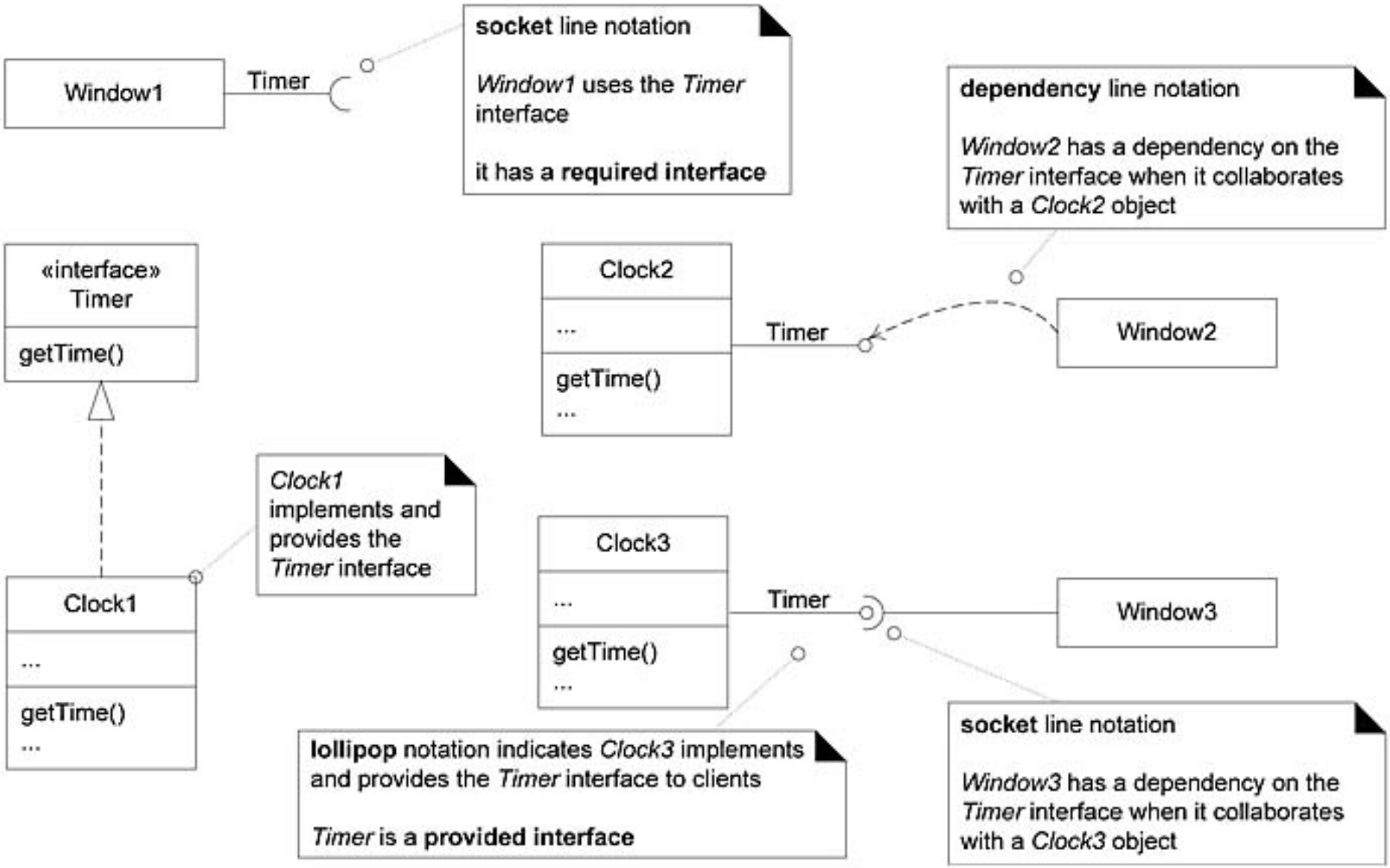
Pourquoi des interfaces ?

- Utilisation similaire aux classes abstraites
- En Java : une classe ne peut hériter de plus d'une classe, mais elle peut réaliser plusieurs interfaces

Notations UML :

- Nom : `«interface» Itf1`
- Héritage : `Itf1` —▷ `Itf2`
- Réalisation : `Class1` - - - -▷ `Itf1` ou `Class1` ——-O `Itf1`
- Utilisation : `Class2` - - «use» - -> `Itf1` ou `Class2` ——-C `Itf1`

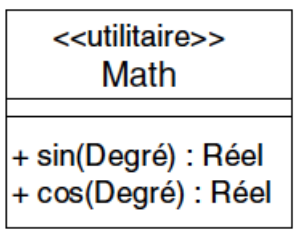
Exemple

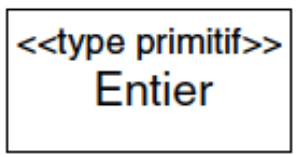


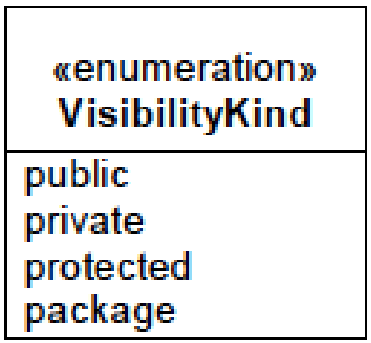
[image empruntée à Craig Larman]

Quelques stéréotypes et mots-clés

- `{abstract}` : classe abstraite (alternative : nom en italique)
- «`interface`» : interface
- «`énumération`» : instances appartiennent à un ensemble fini de littéraux
- «`type primitif`» : instances sont des valeurs d'un type primitif
- «`classe implémentation`» : implémentation d'une classe dans un langage de programmation
- «`utilitaire`» : variables et procédures globales







Synthèse des différentes relations

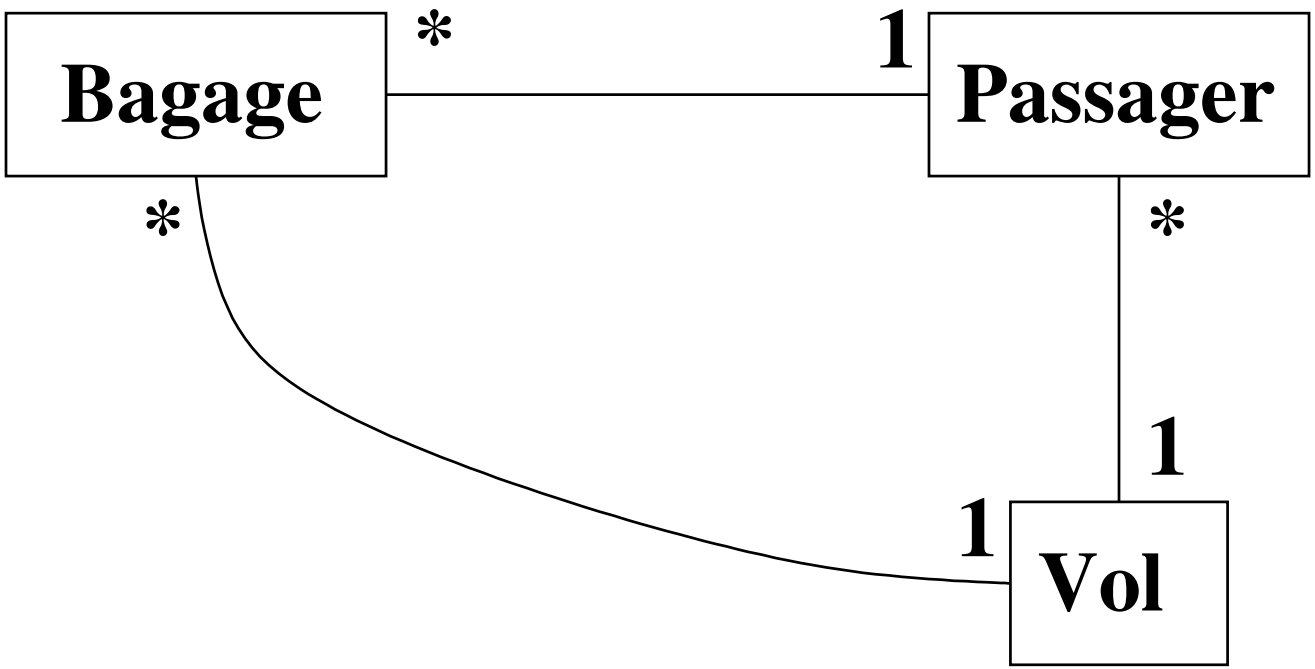
- Association
  - Navigable dans les 2 sens : ——————
  - Navigable dans un sens : ——————>
    ou —X——————>
- Composition : ——————◆
- Agrégation : ——————◇
- Généralisation : ——————▷
- Réalisation d'une interface : ——————O ou - - - - - - - - - - - - ▷
- Utilisation d'une interface : ——————C ou - - - - «use» - - - - >
- Intanciation d'une classe générique : - - - - «bind» (Type) - - - - >
- Dépendance : - - - - - - - - - - - - - >

Cherchez l'erreur (1/3)

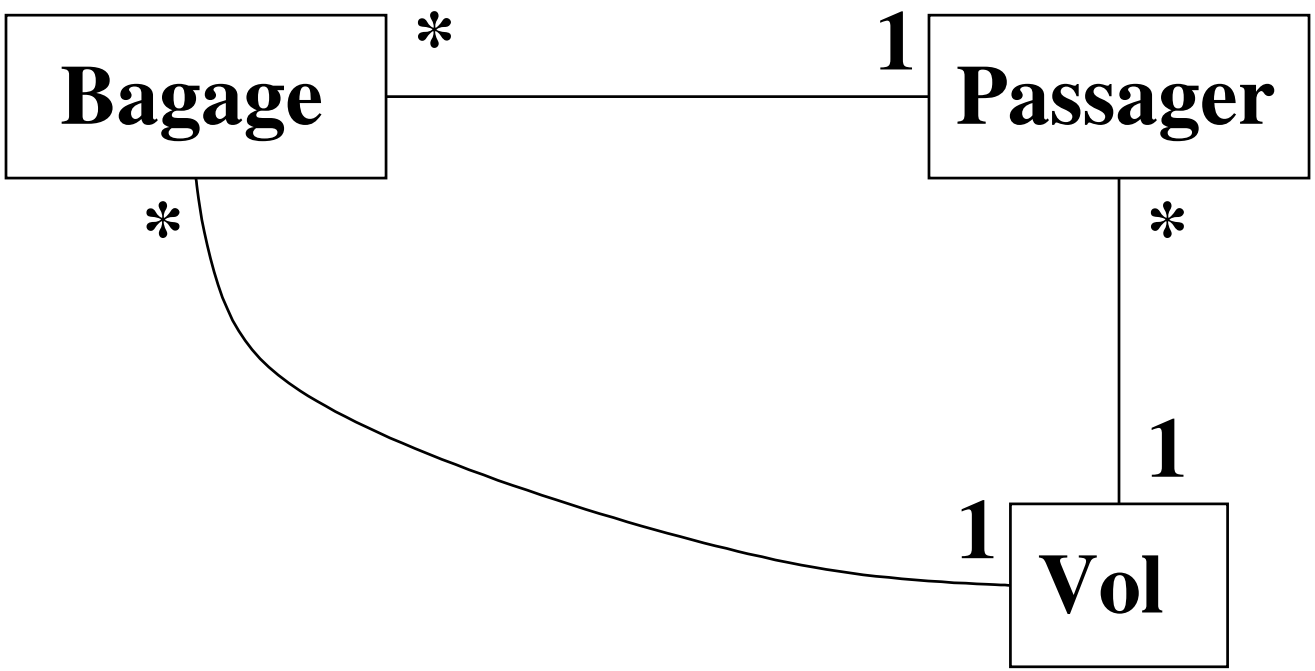
Cherchez l'erreur (1/3)

Navigabilité : Tout objet doit être accessible via une association

Cherchez l'erreur (2/3)

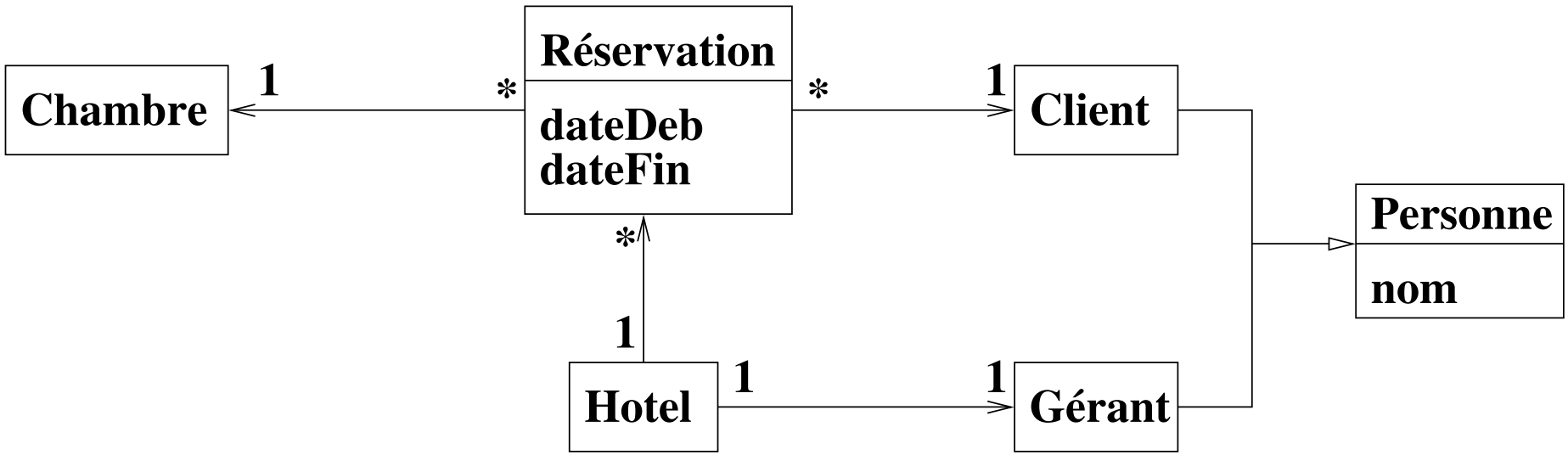


Cherchez l'erreur (2/3)

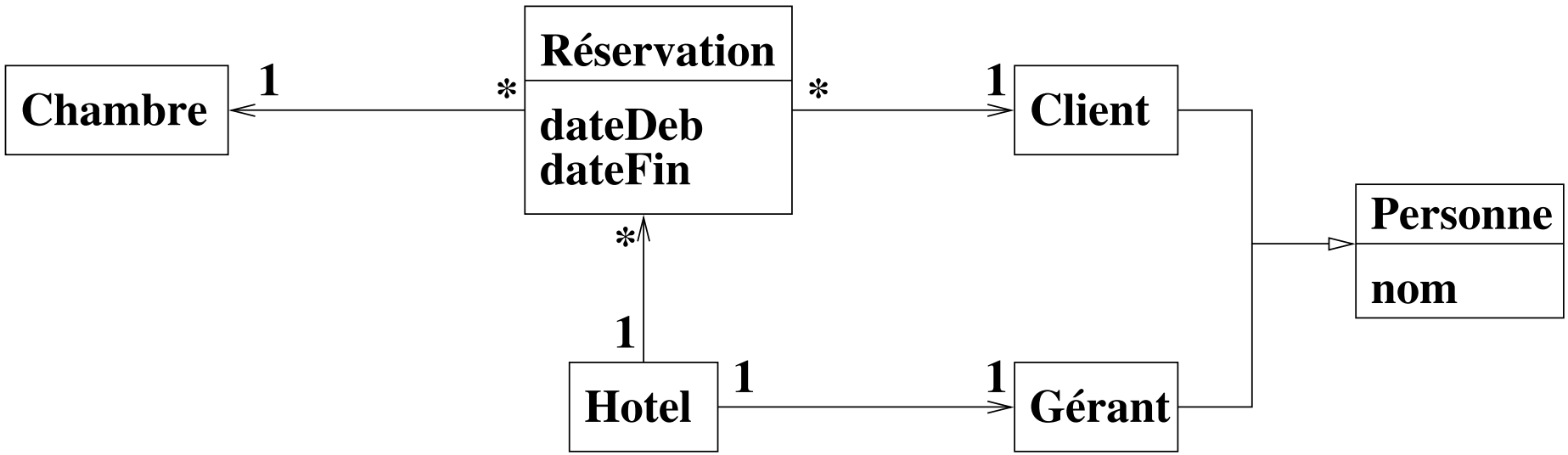


- Eviter les associations redondantes :
  → Le vol d'un bagage est obtenu à partir du passager
  → Les bagages d'un vol sont obtenus à partir des passagers

Cherchez l'erreur (3/3)



Cherchez l'erreur (3/3)



Navigabilité : En l'absence de réservations, on ne peut accéder ni aux chambres ni aux clients

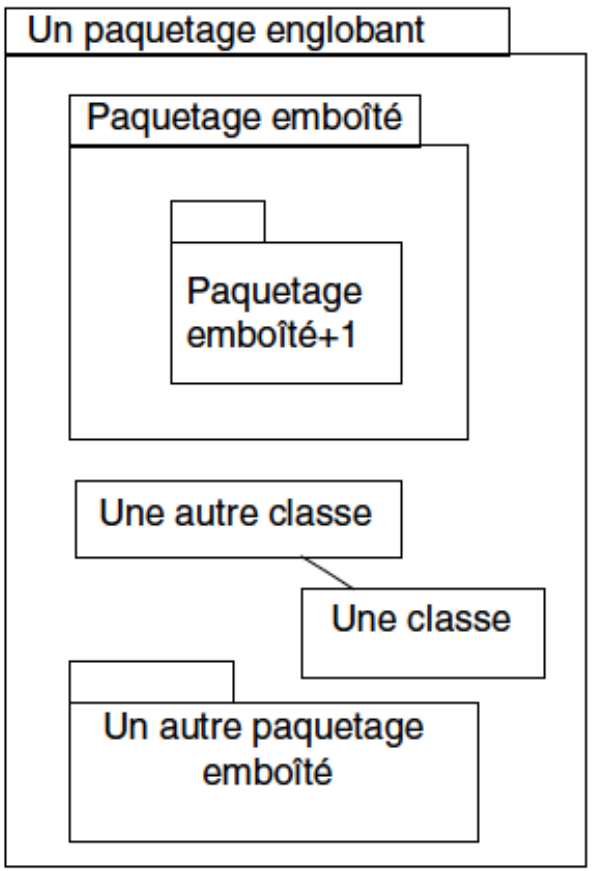
Plan du cours

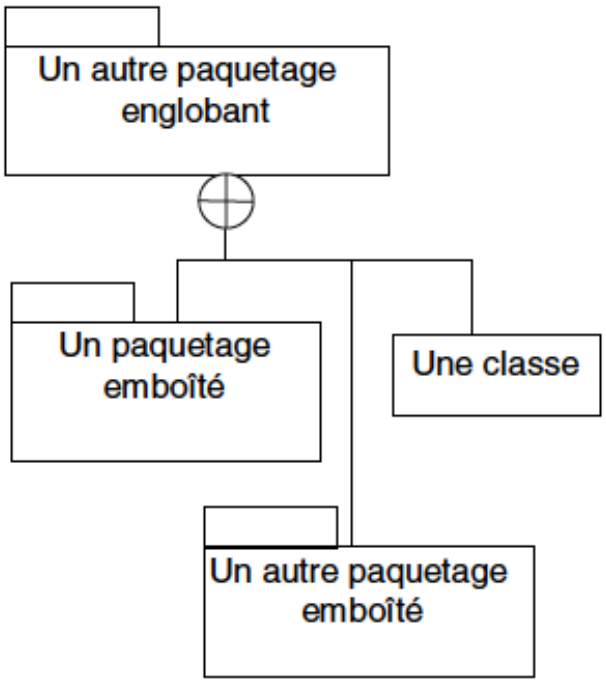
Diagrammes de paquetages

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

Élément de modélisation qui :

- Contient d'autres éléments de modélisation (classes, autres paquetages, ...)
  $\rightarrow$ Possibilité de ne pas représenter tous les éléments contenus
- Définit un espace de nom (namespace)

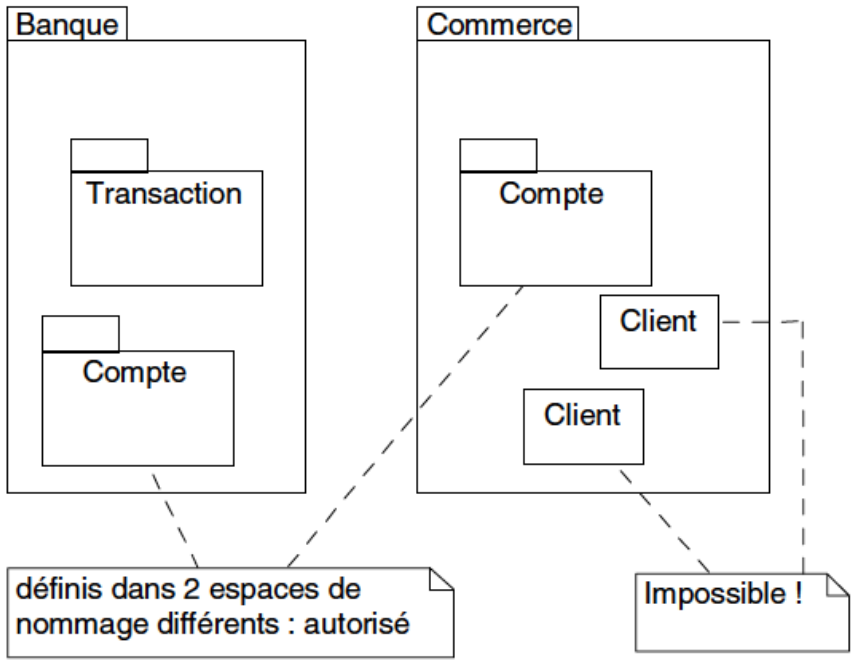




[Image extraite de Muller et Gaertner]

Espaces de nom et visibilité

- 2 éléments dans 2 paquetages différents sont différents, quel que soit leur nom



  - Nom complet = nom préfixé par les noms des paquetages englobants : `Banque::Compte` ≠ `Commerce::Compte`
- Visibilité d'un élément E dans un package P :
  - Public (+) : Elément visible par tous (utilisation du nom complet)
  - Privé (-) : Elément visible uniquement par
    - Les autres éléments de P
    - Les éléments englobés par E (si E est un package)

Dépendances entre paquetages

Reflètent les dépendances entre éléments des paquetages :

Une classe A dépend d'une classe B (noté A - - - - - - - - - > B) si :

- Il existe une association navigable de A vers B
  (ou A possède un attribut de type B)
- Une méthode de A a un paramètre ou une variable locale de type B

Stéréotypes sur les dépendances inter-paquetage

- A - - - «accède» - - - > B :
  Tout élément public de B est accessible par son nom complet depuis A
- A - - - «importe» - - - > B :
  Tout élément public de B est accessible par son nom depuis A
  → Création d'alias si nécessaire

Valeur marquée `{global}` :

Paquetage visible par tous les autres paquetages
→ Inutile de montrer les dépendances vers ce paquetage

Architecture logique

Qu'est-ce qu'une architecture logique ?

- Regroupement des classes logicielles en paquetages
  $\rightarrow$ Point de départ pour un découpage en sous-systèmes

Objectifs d'une architecture logique :

- Encapsuler et décomposer la complexité
- Faciliter le travail en équipes
- Faciliter la réutilisation et l'évolutivité

$\rightarrow$ Forte cohésion intra paquetage
$\rightarrow$ Faible couplage inter paquetages

Exemples d'architectures logiques :

- Architecture en couches
- Architecture Modèle - Vue - Contrôleur (MVC)
- Architecture Multi-tiers
- ...

Plan du cours

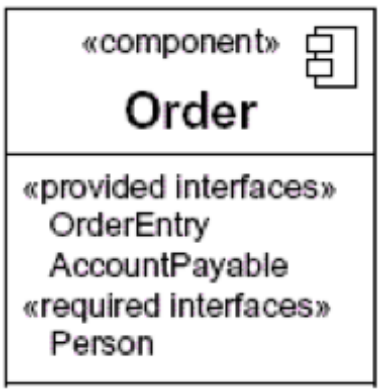
Diagrammes de composants

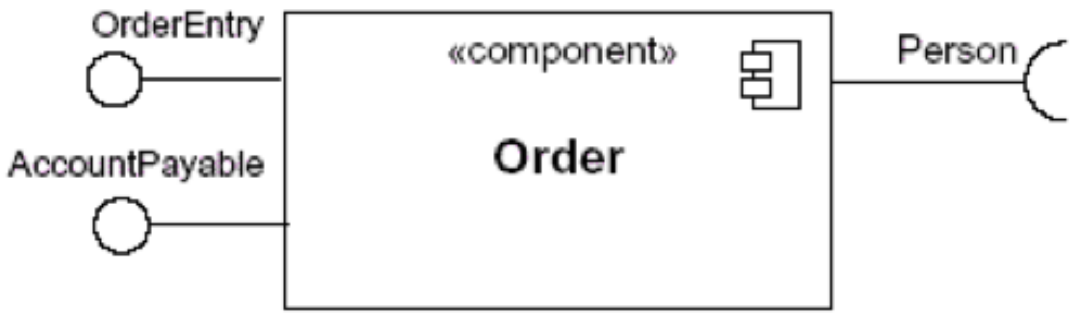
Composant :

- Encapsule l'état et le comportement d'un ensemble de classifiers (classes, composants...)
- Spécifie les services fournis et requis :
  - Interfaces fournies : «`provided interfaces`» ou ——O
  - Interfaces requises : «`required interfaces`» ou ——C

→ Substituable à tout composant qui offre/requiert les m̂ interfaces

Exemple :

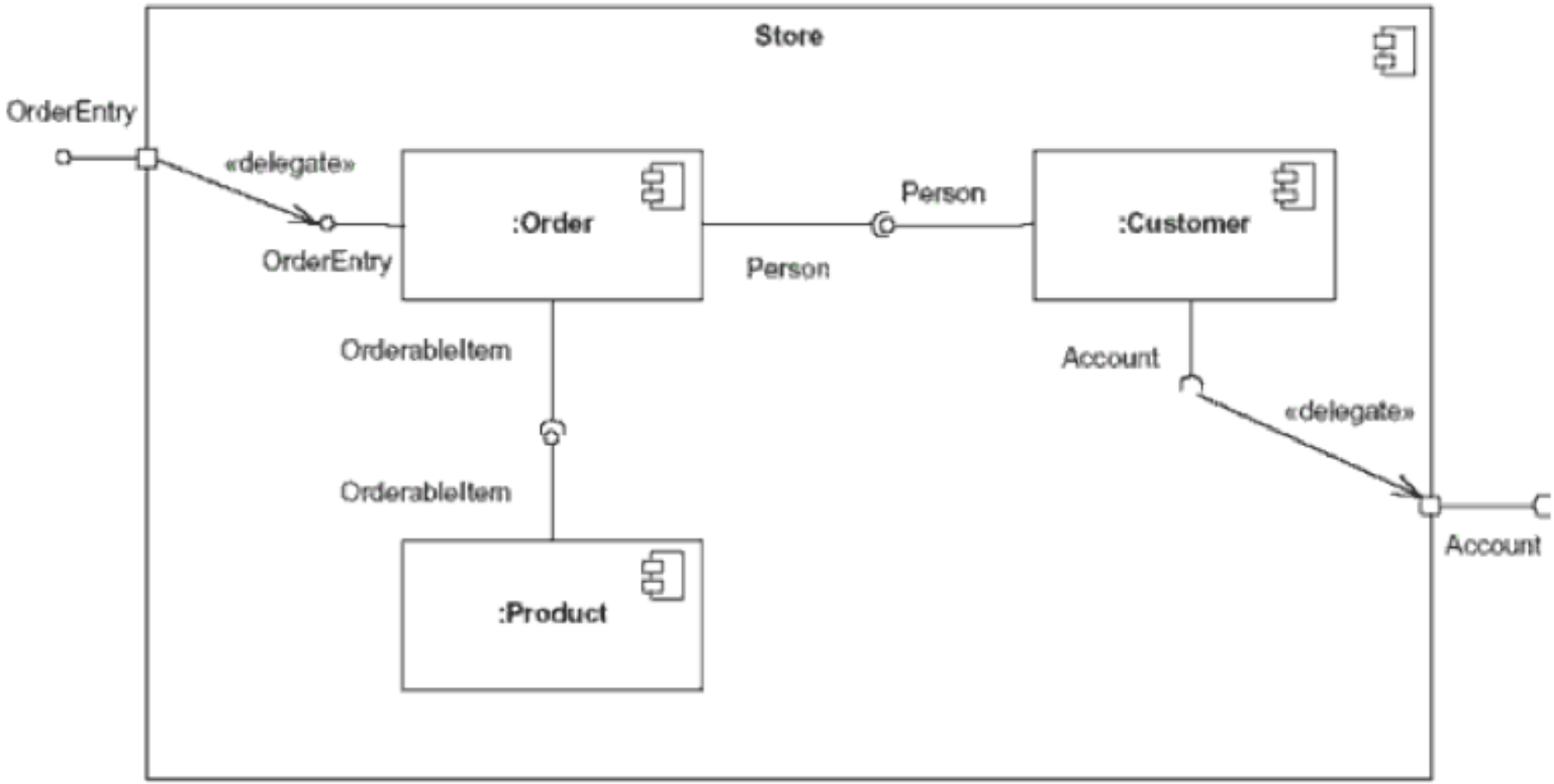




[Image extraite de www.ibm.com/developerworks/rational]

Relations entre composants

→ Inclusion entre composants et Fourniture ou Utilisation d'interfaces



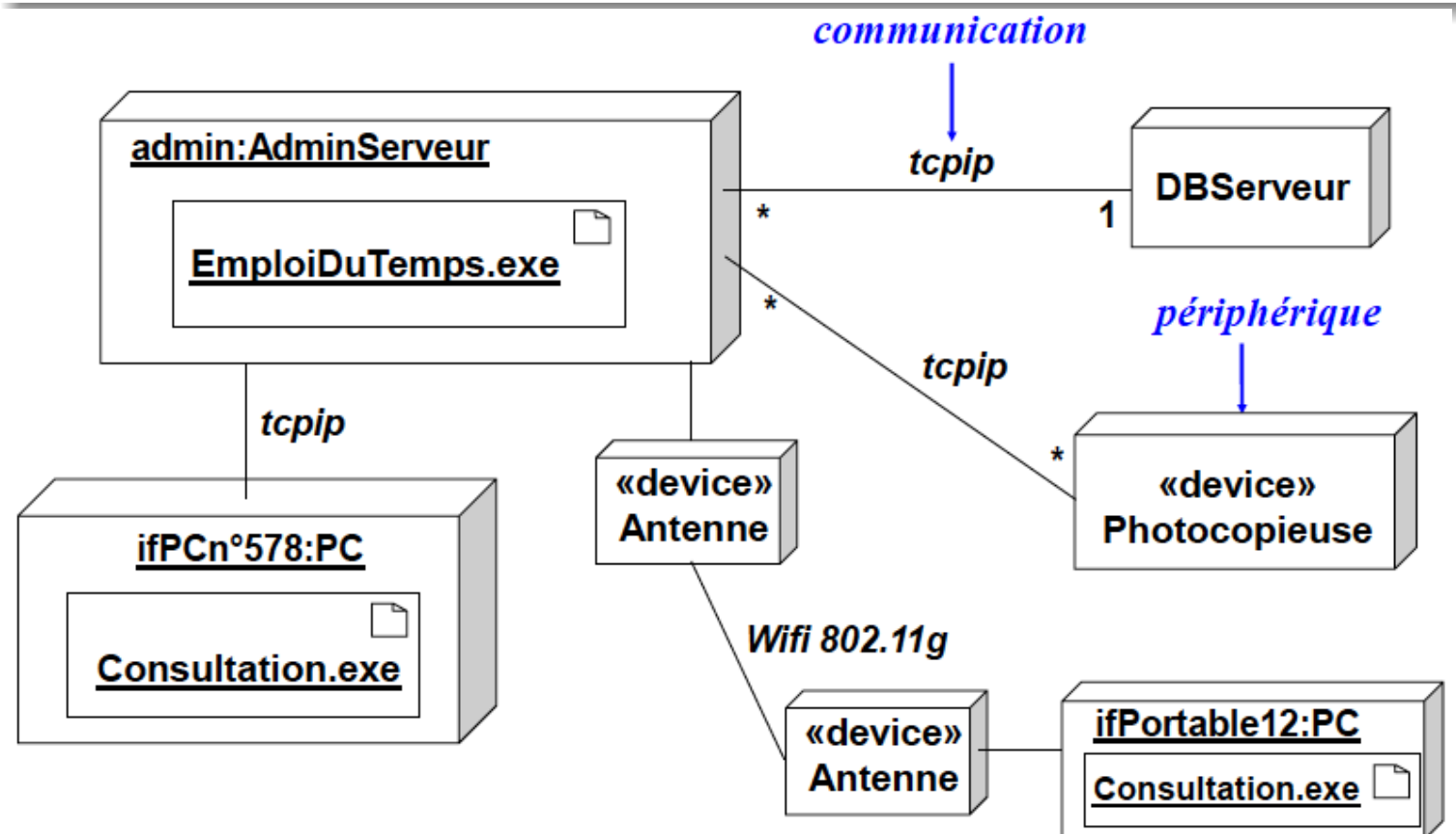
[Image extraite de www.ibm.com/developerworks/rational]

Plan du cours

Diagrammes de déploiement

Modéliser le déploiement du système sur une architecture physique

- Disposition des artefacts sur les nœuds physiques
  - Artefacts : instances de composants, processus, ...
  - Nœuds physiques : Ordinateur, Téléphone, Imprimante, ...
- Moyens de communication entre les nœuds



Plan du cours

Modéliser le comportement avec UML

Décrire le comportement du système en termes d'interactions

- Système vu comme une boite noire :
  - Qui interagit avec le système, et dans quel but ?
    → Diagramme de cas d'utilisation
  - Comment interagit le système avec son environnement ?
    → Diagramme de séquence système
- Système vu comme une boite blanche :
  - Comment interagissent les objets du système ?
    → Diagramme de séquence et/ou de communication

Décrire l'évolution du système dans le temps

- Comment évoluent les états des objets ?
  → Diagrammes d'états-transitions

Plan du cours

Cas d'utilisation

Pourquoi faire ?

Permettre au client de décrire ses besoins

- Parvenir à un accord (contrat) entre clients et développeurs
- Point d'entrée pour les étapes suivantes du développement

Qu'est ce qu'un cas d'utilisation ?

- Usage que des acteurs font du système
  - Acteur : Entité extérieure qui interagit avec le système
    → Une même personne peut jouer le rôle de différents acteurs
    → Un acteur peut être un autre système (SGBD, Horloge, ...)
  - Usage : Séquence d'interactions entre le système et les acteurs
- Généralement composé de plusieurs scénarios (instances)
  → Scénario de base et ses variantes (cas particuliers)
  → Description des scénarios à l'aide de diagrammes de séquence

Comment découvrir les cas d'utilisation ?

- Délimiter le périmètre du système
- Identifier les acteurs interagissant avec le système :
  - Ceux qui utilisent le système
  - Ceux qui fournissent un service au système
- Identifier les acteurs principaux

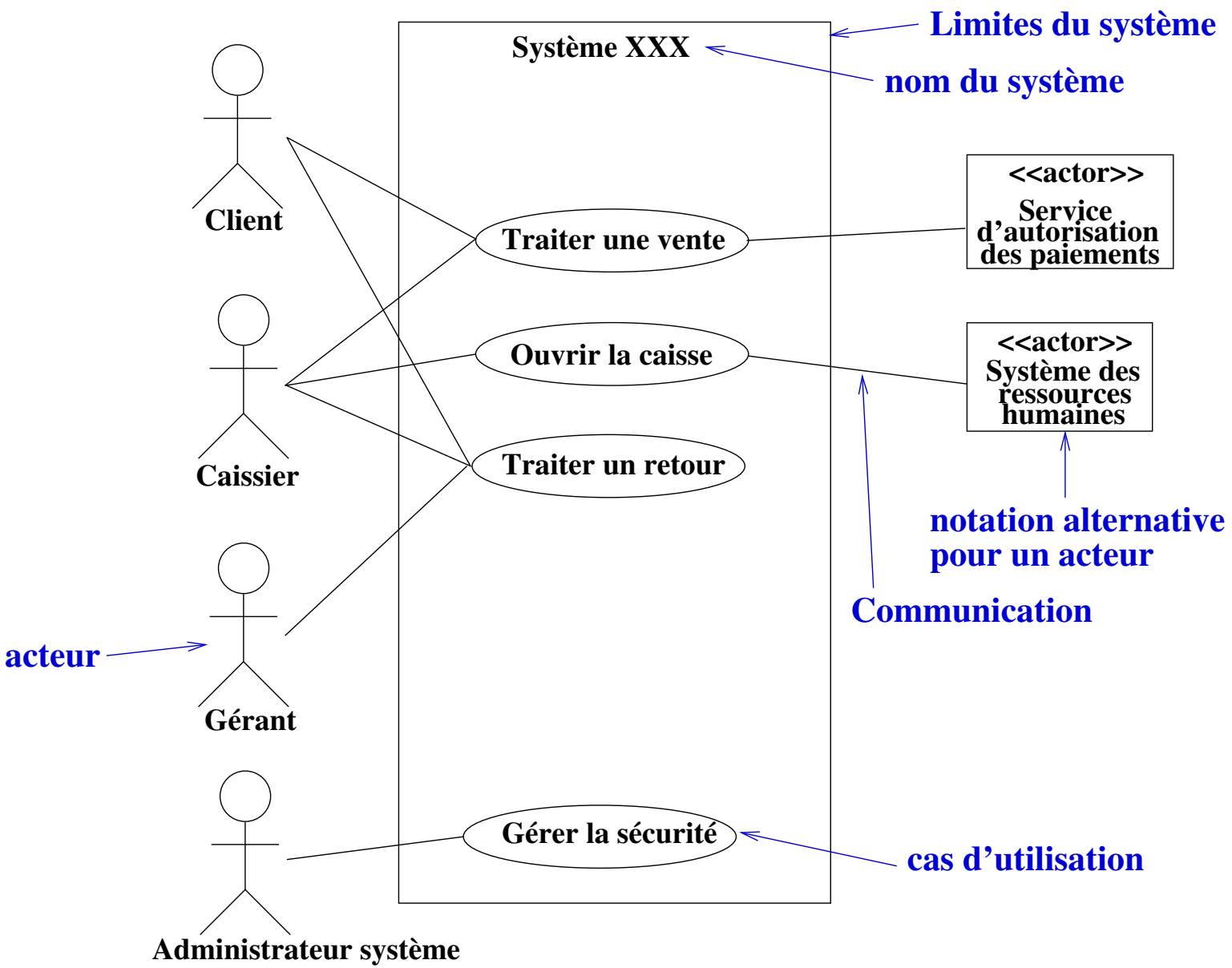
  → Ceux qui utilisent le système pour atteindre un but
- Définir les cas d'utilisation correspondant à ces buts

  → Nom = Verbe à l'infinitif + Groupe nominal

Comment décrire les cas d'utilisation ?

- Diagramme de cas d'utilisations
  → Récapitulatif graphique des interactions entre acteurs et cas
- Diagramme de séquence
  → Description de chaque scénario
  → Séquences d'interactions entre les acteurs et le système

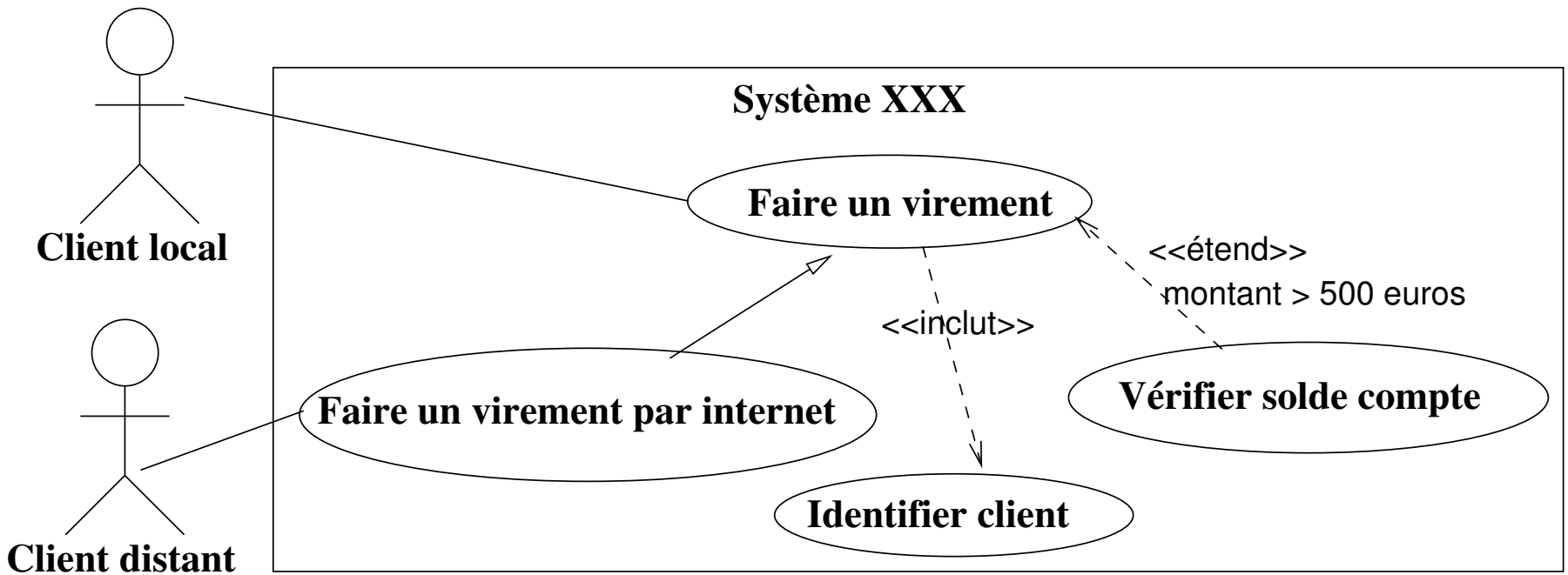
Diagrammes de cas d'utilisation



Relations entre cas d'utilisation

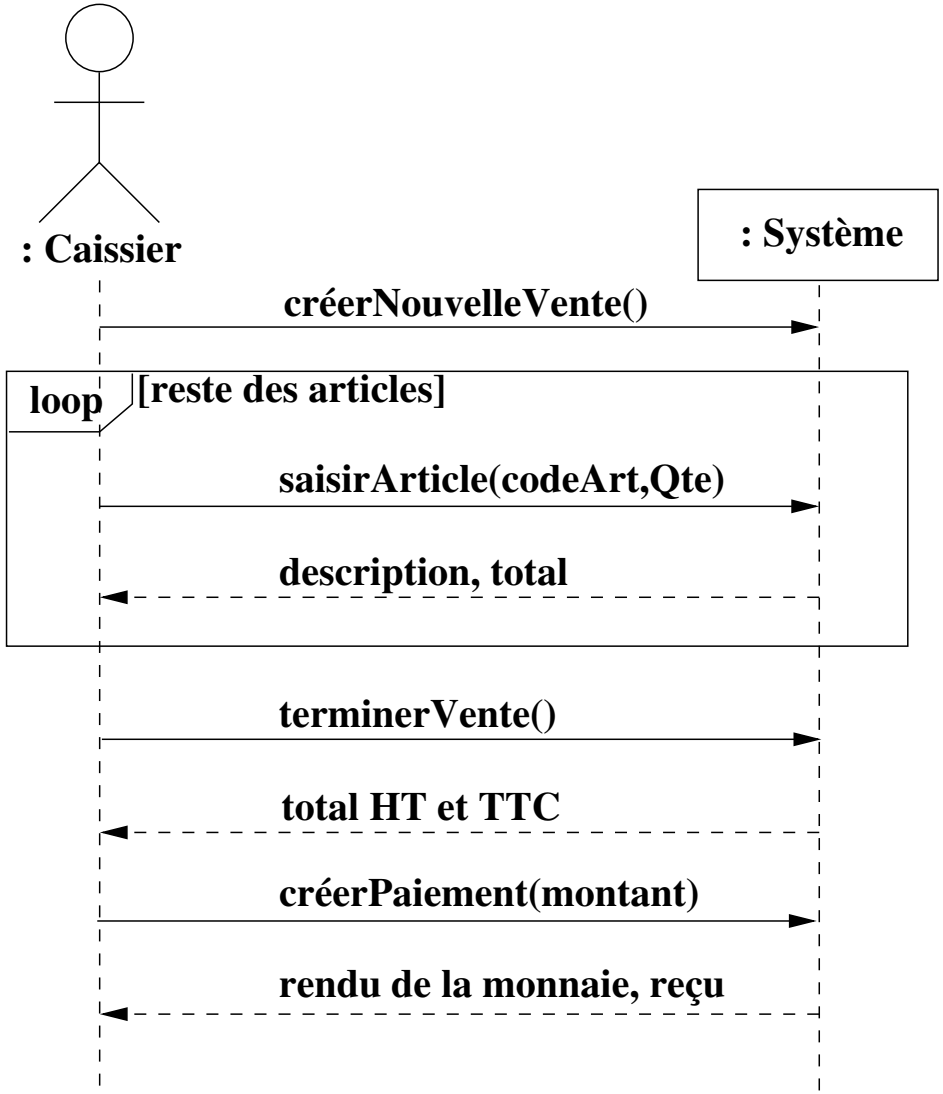
- Généralisation : ——▷
- Inclusion : - - - - - «include» - - - - - >
- Extension : - - - - - «extend» - - - - - > (préciser la condition)

→ A utiliser avec modération



Diagrammes de séquence d'un cas d'utilisation

Représentation graphique d'un scénario d'un cas d'utilisation



→ Le système vu comme une boite noire

Plan du cours

Diagrammes d'interaction

→ **Point de vue temporel sur les interactions**

Pendant la capture des besoins (système = boite noire) :

→ Interactions entre acteurs et système

- Décrire les scénarios des cas d'utilisation

Pendant la conception (système = boite blanche) :

→ Interactions entre objets

- Réfléchir à l'affectation de responsabilités aux objets
  - Qui crée les objets ?
  - Qui permet d'accéder à un objet ?
  - Quel objet reçoit un message provenant de l'IHM ?
  - ...

  de façon à avoir un faible couplage et une forte cohésion
- Elaboration en parallèle avec les diagrammes de classes

  → Contrôler la cohérence des diagrammes !

Diagrammes de séquence vs diagrammes de communication

Diagrammes de séquence :

Structuration en termes de

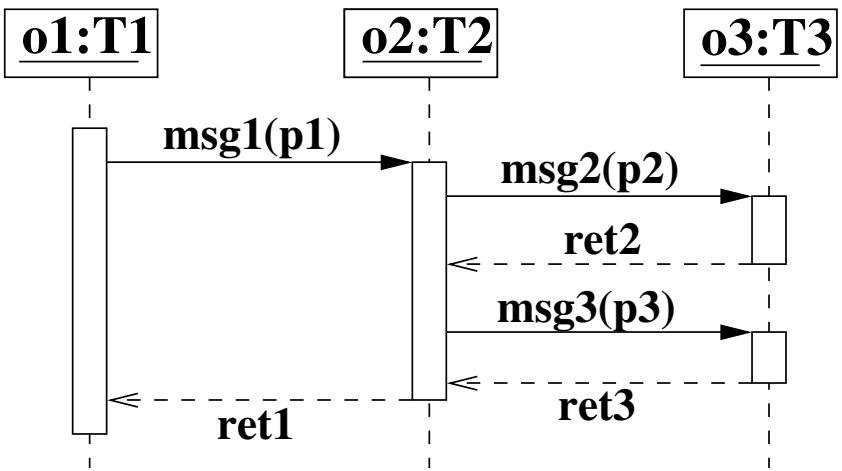
- temps → axe vertical
- objets → axe horizontal

Diagrammes de communication :

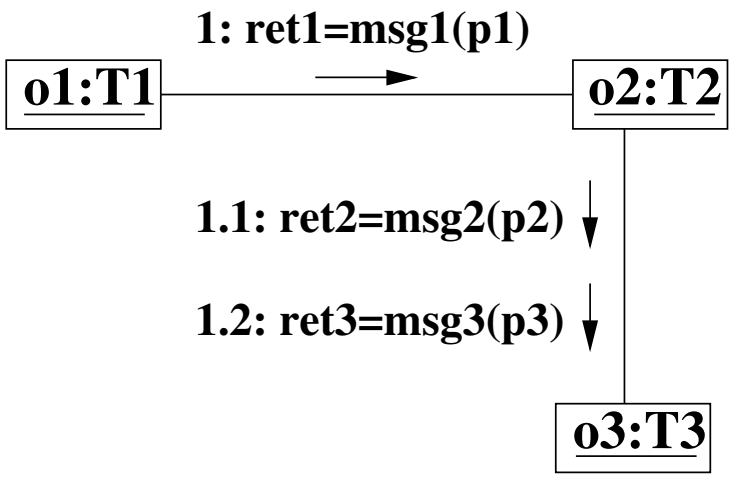
Structuration en multigraphe

- Numérotation des arcs pour modéliser l'ordre des interactions

Exemple :

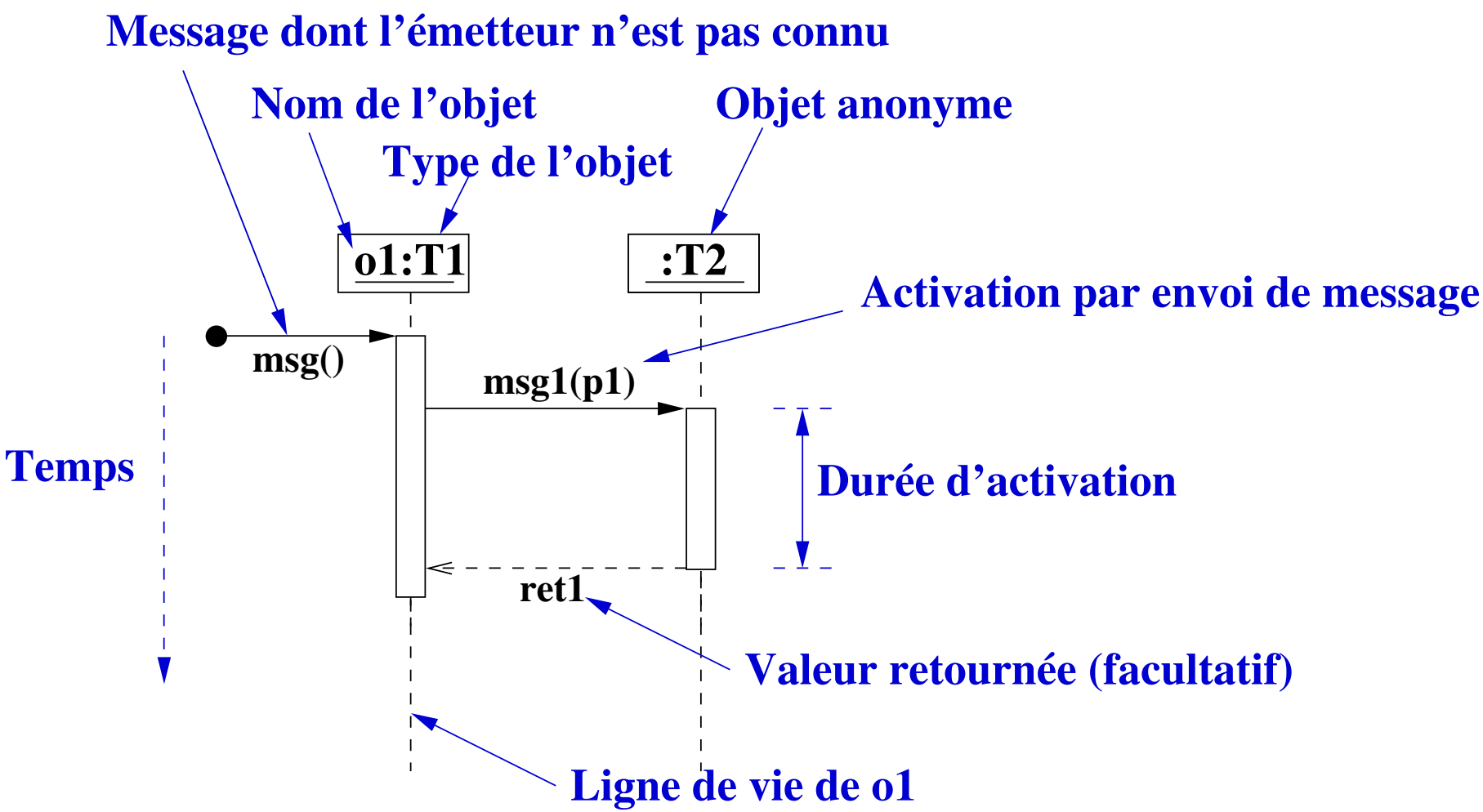


Exemple :



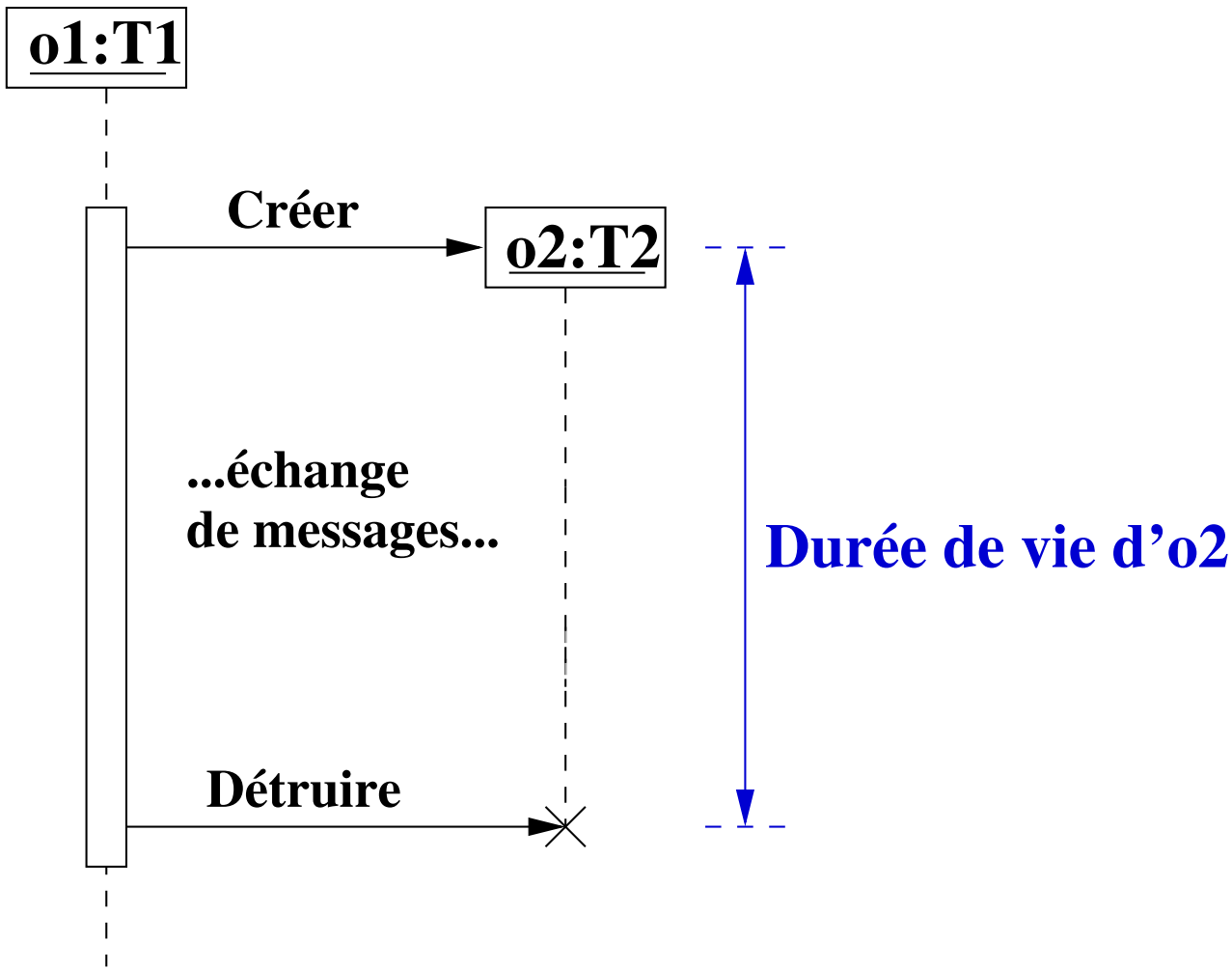
Diagrammes de séquence

Ligne de vie et activation



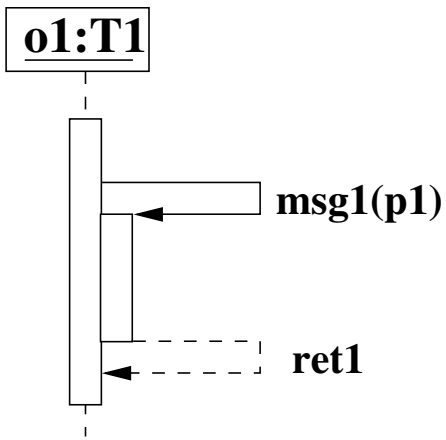
Diagrammes de séquence

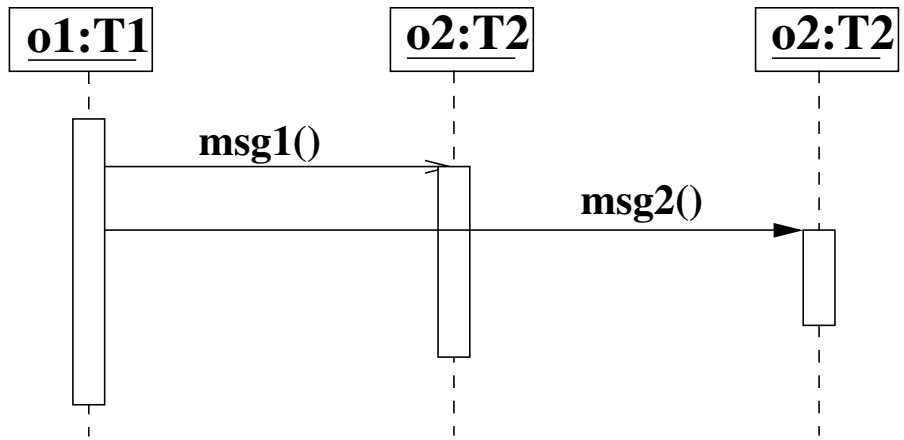
Création et destruction d'objets

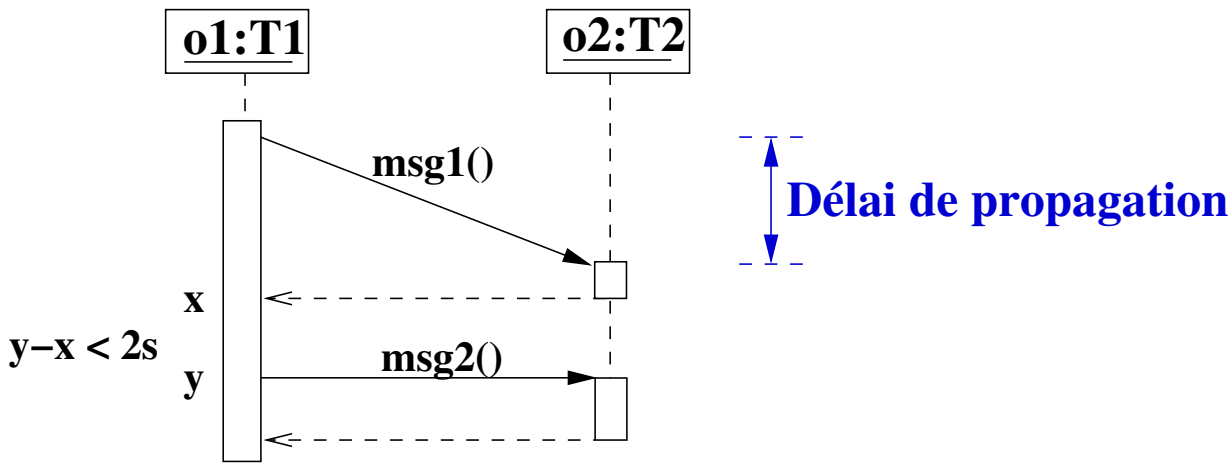


Diagrammes de séquence

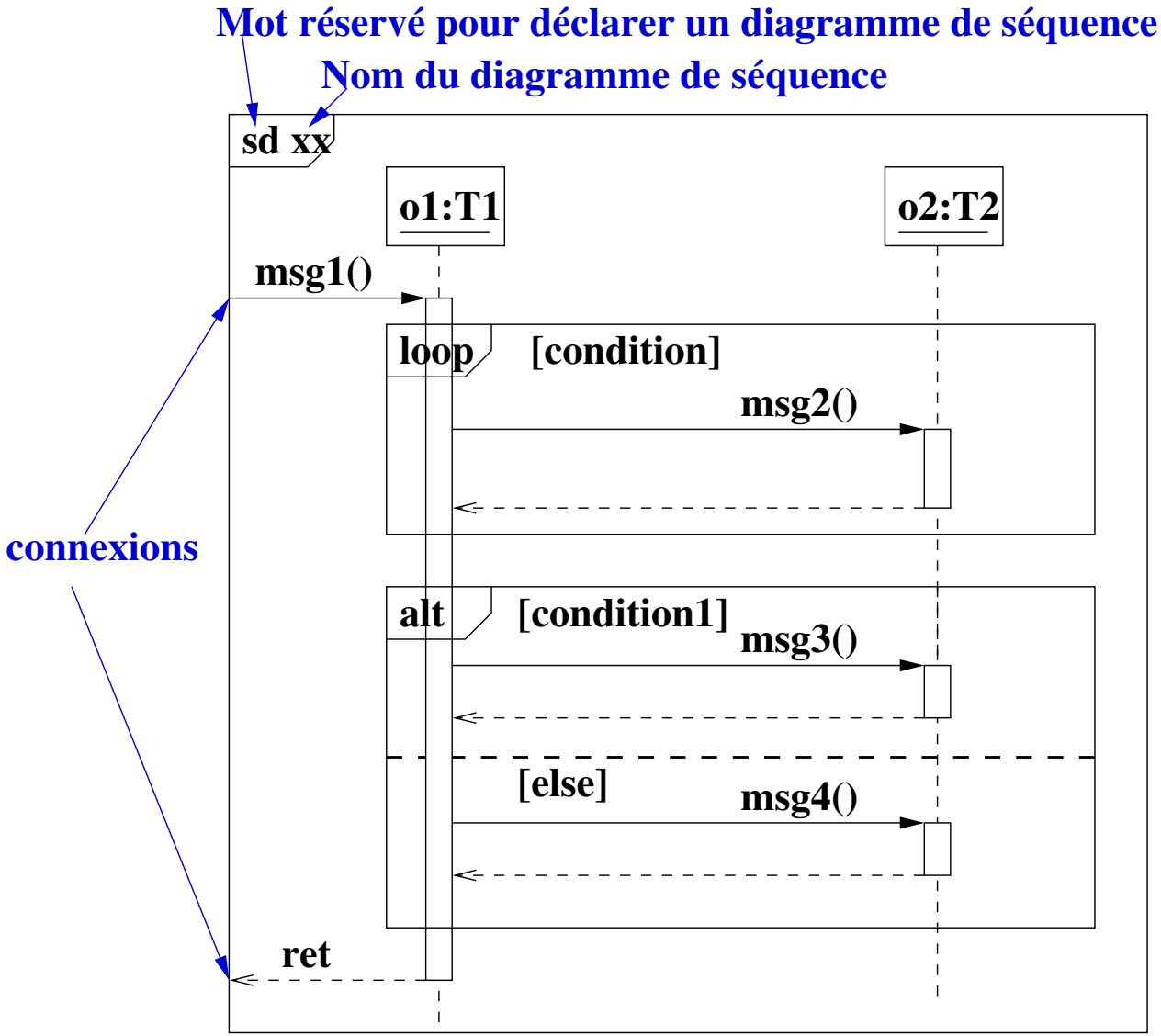
Messages réflexifs, messages asynchrones et contraintes temporelles

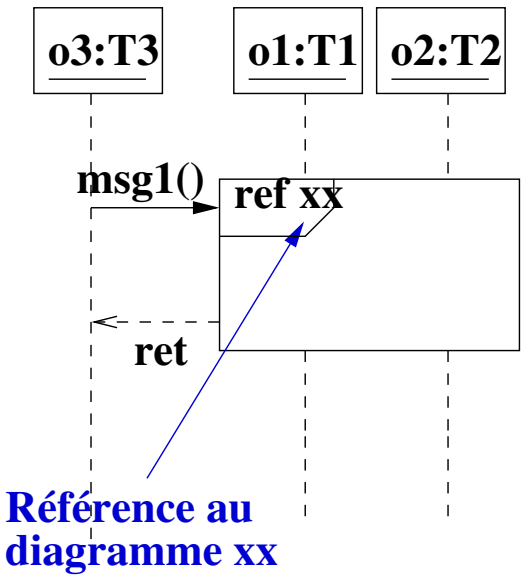






Diagrammes de séquence : Cadres

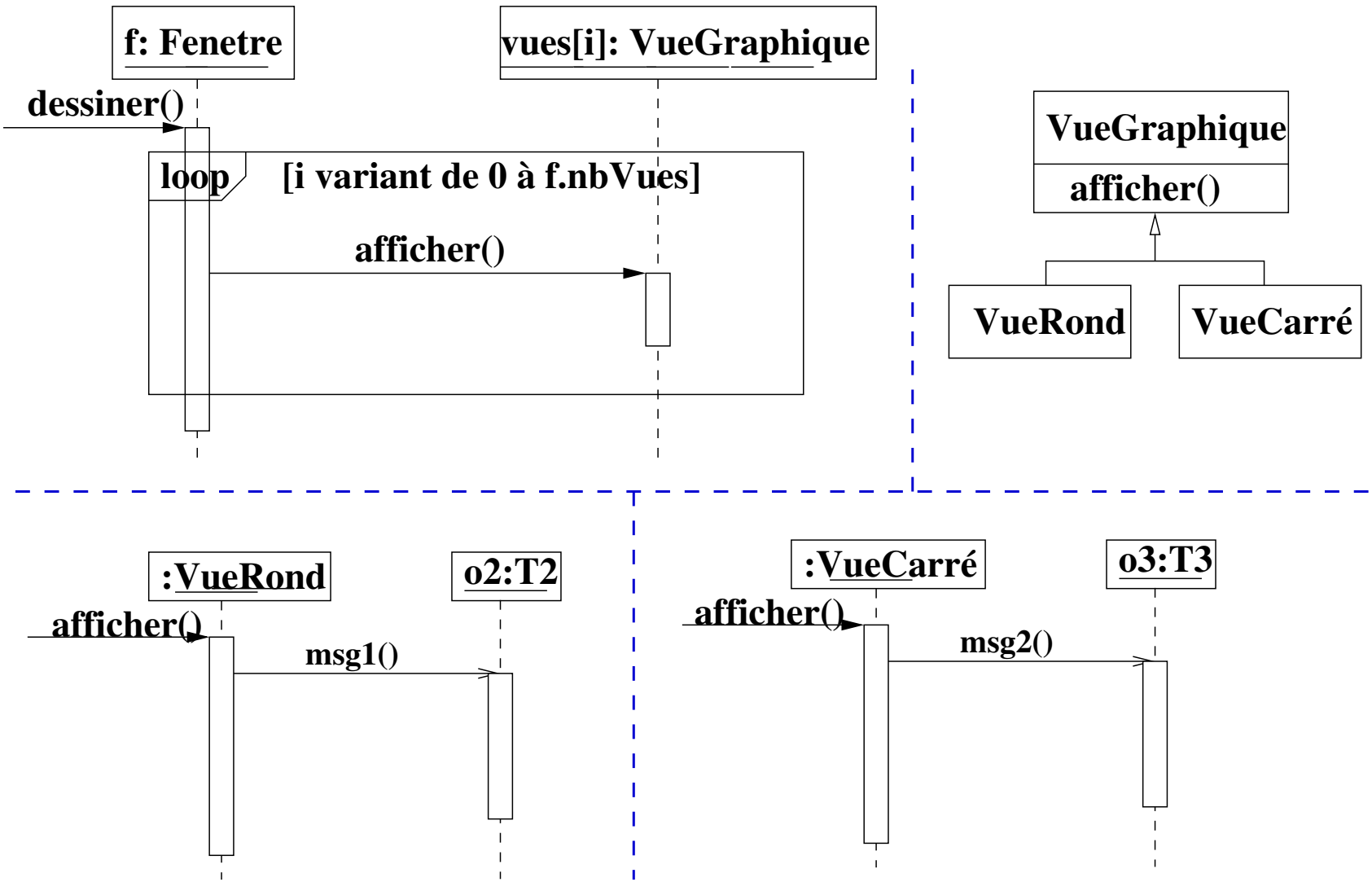




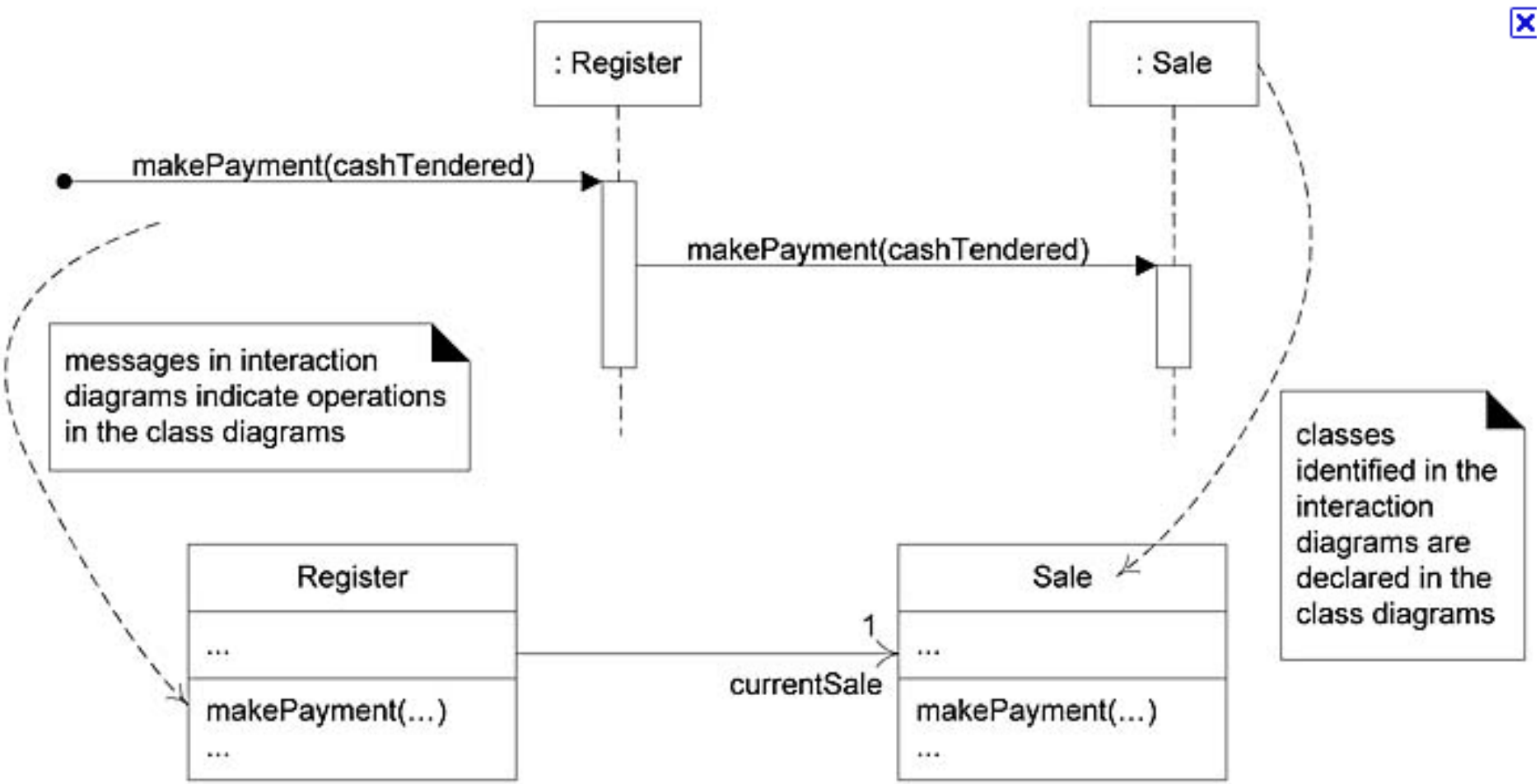
→ Existe aussi : par, opt, critique

Diagrammes de séquence

Messages polymorphes



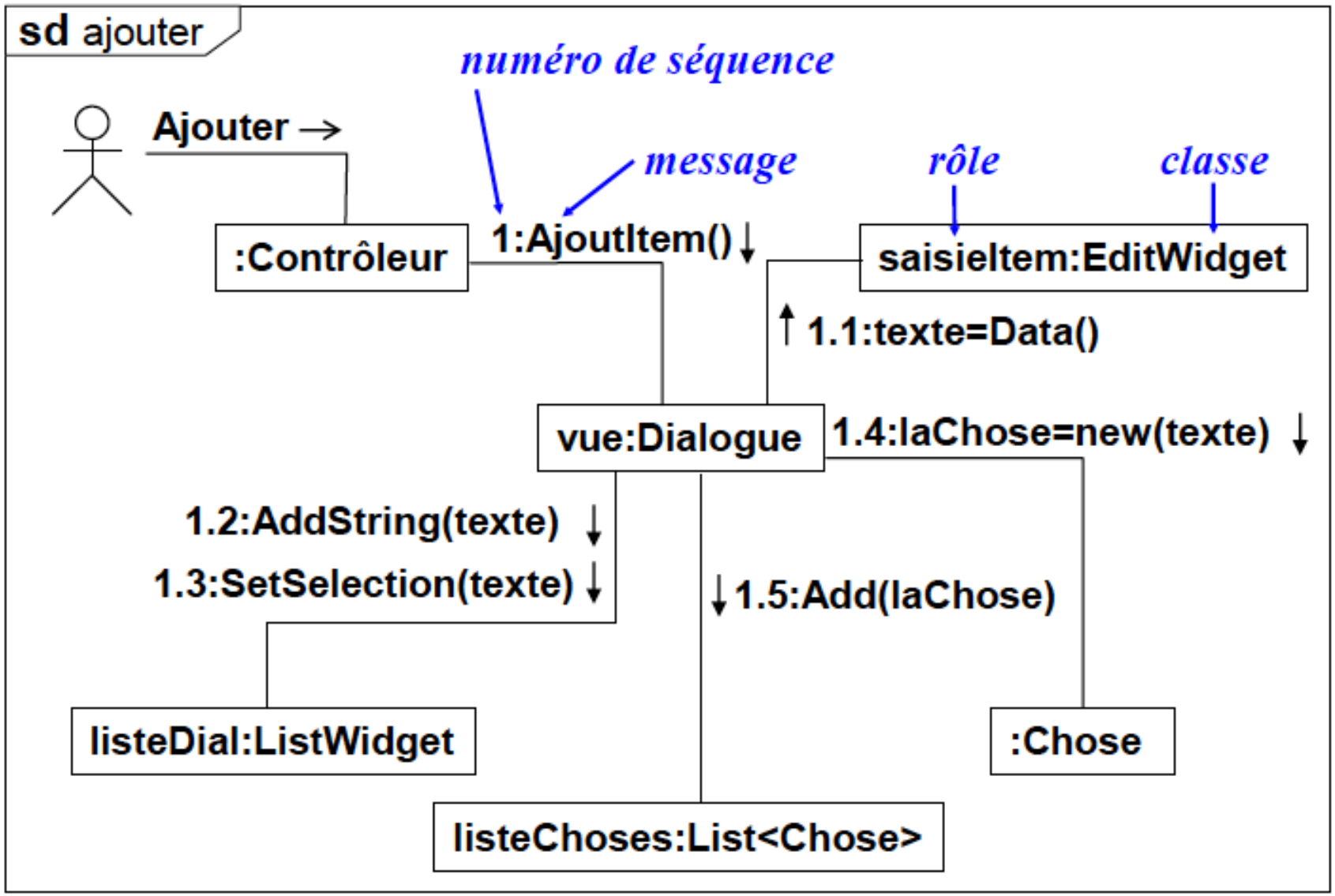
Liens entre diag. de classes et d'interaction



[Figure extraite du livre de C. Larman]

Diagrammes de communication

Présentation alternative d'une séquence d'interactions



Plan du cours

Diagrammes d'états-transitions

Utilisés pour modéliser :

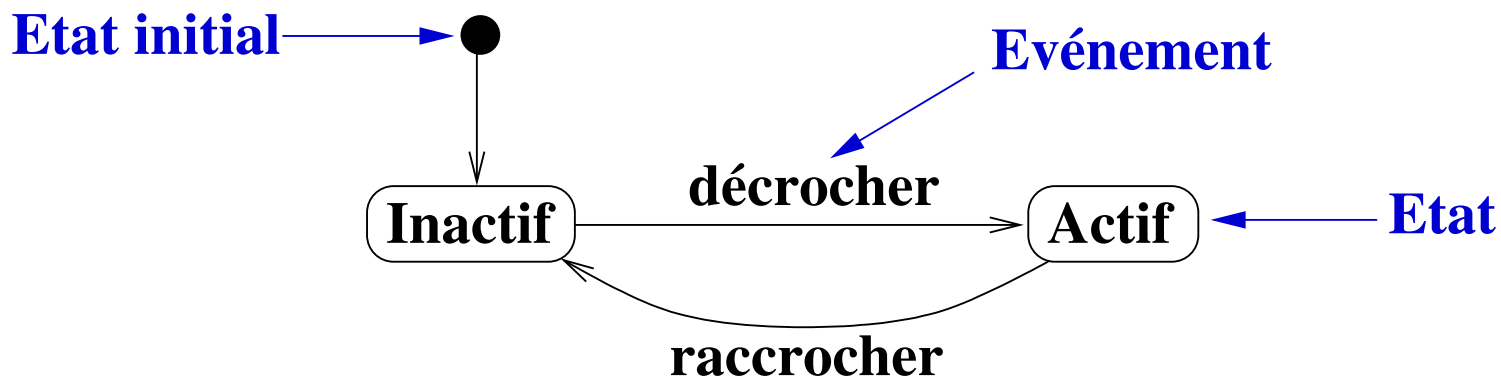
- Le cycle de vie des objets
  - Evolution de l'état des objets
  - Comportement face à l'arrivée d'événements

  $\rightarrow$ Intéressant si les réponses aux événements dépendent des états
- Mais aussi : protocoles complexes (GUI, ...), processus métier, ...

Événements

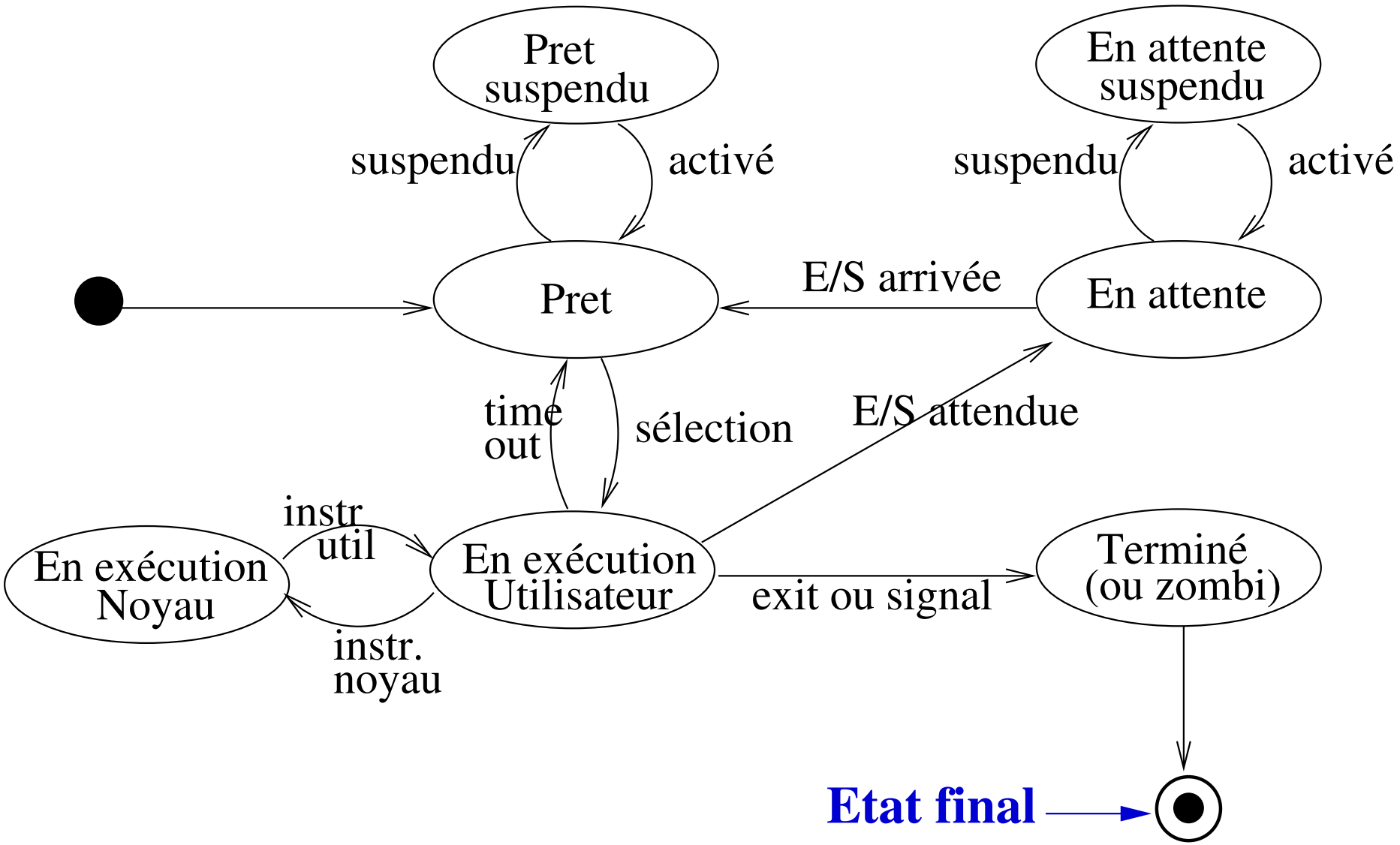
- Occurrence d'un fait significatif ou remarquable
  - Réception d'un signal
  - Réception d'un message
  - Expiration d'une temporisation
  - ...
- Les événements déclenchent les transitions d'un état vers un autre
  $\rightarrow$ Evénement «perdu» si aucune transition spécifiée pour lui
- Il y a un état initial, mais pas toujours d'état final

Exemple :



Exemple de diagramme d'état

Modélisation des états d'un processus



Différents types d'événements :

- Signaux :

$$Q0 \xrightarrow{\texttt{nomSignal}} Q1$$

- Appels d'opérations :

$$Q0 \xrightarrow{\texttt{nomOp(paramètres)}} Q1$$

$\rightarrow$ Les paramètres peuvent être typés ou non

- Evénements temporels :

$$Q0 \xrightarrow{\texttt{after(qté tps)}} Q1$$

$\rightarrow$ Passage dans l'état $Q1$ `qté tps` après l'arrivée dans l'état $Q0$

- Evénements de changement :

$$Q0 \xrightarrow{\texttt{when(cond)}} Q1$$

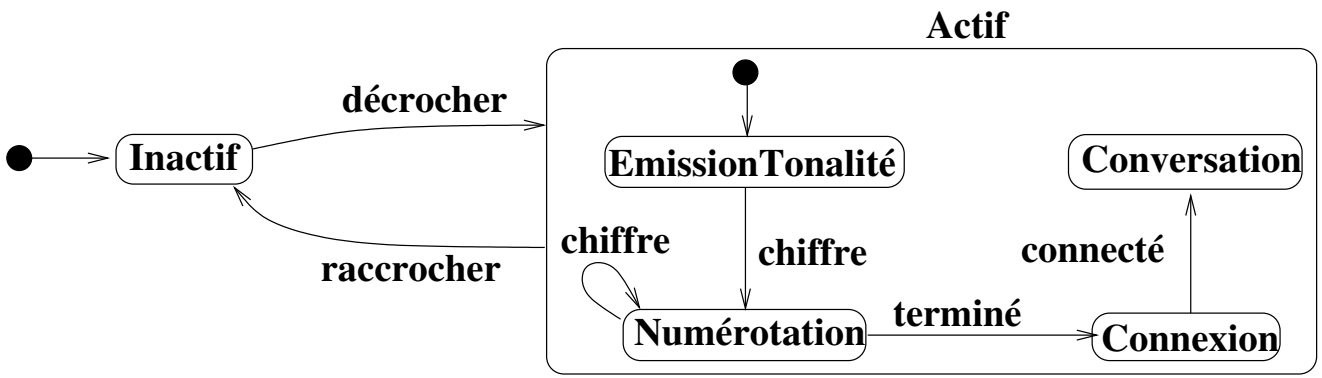
$\rightarrow$ Passage dans l'état $Q1$ quand `cond` devient vraie

Etats imbriqués (ou composites)

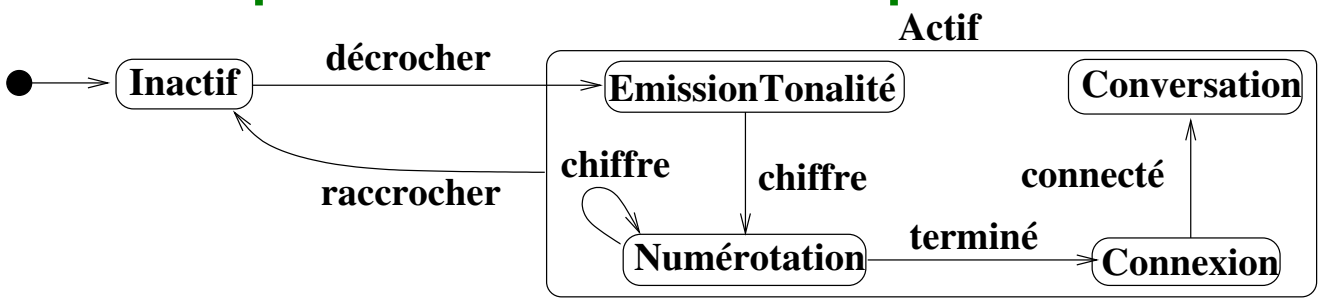
Un état peut contenir des sous-états

- Permet de factoriser les transitions de sortie du composite
  → Chaque transition de sortie s'applique à tous les sous-états
- Une seule transition d'entrée

Exemple :



Notation alternative pour l'état initial du composite :

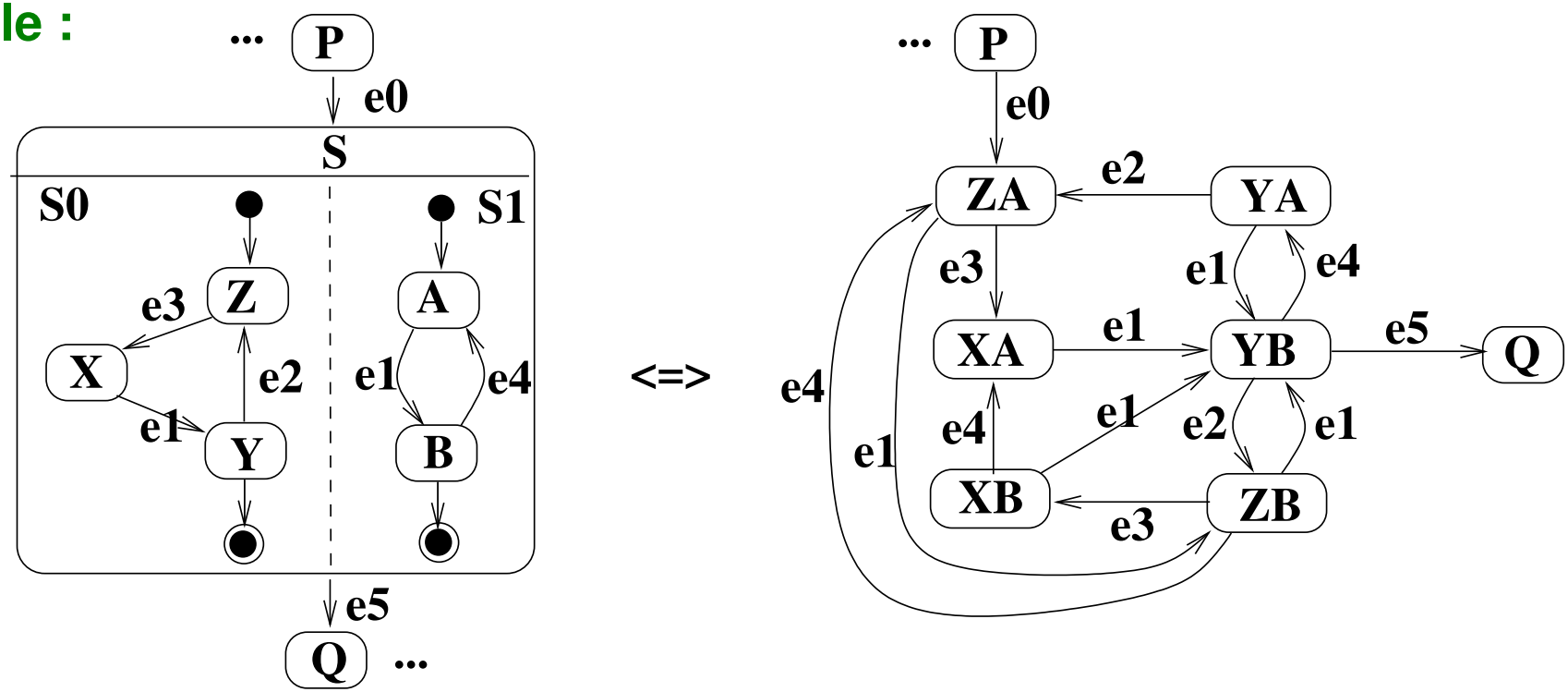


Etats concurrents (1/2)

Plusieurs sous-automates peuvent être concurrents :

- Etat courant = n-uplet de sous-états
  $\rightarrow$ Etats de $S = \{(Z, A), (Z, B), (X, A), (X, B), (Y, A), (Y, B)\}$
- Exécution indépendante des sous-automates
- Un évt peut déclencher une transition dans plusieurs sous-auto.
- Sortie possible quand tous les sous-auto. sont dans un état final

Exemple :

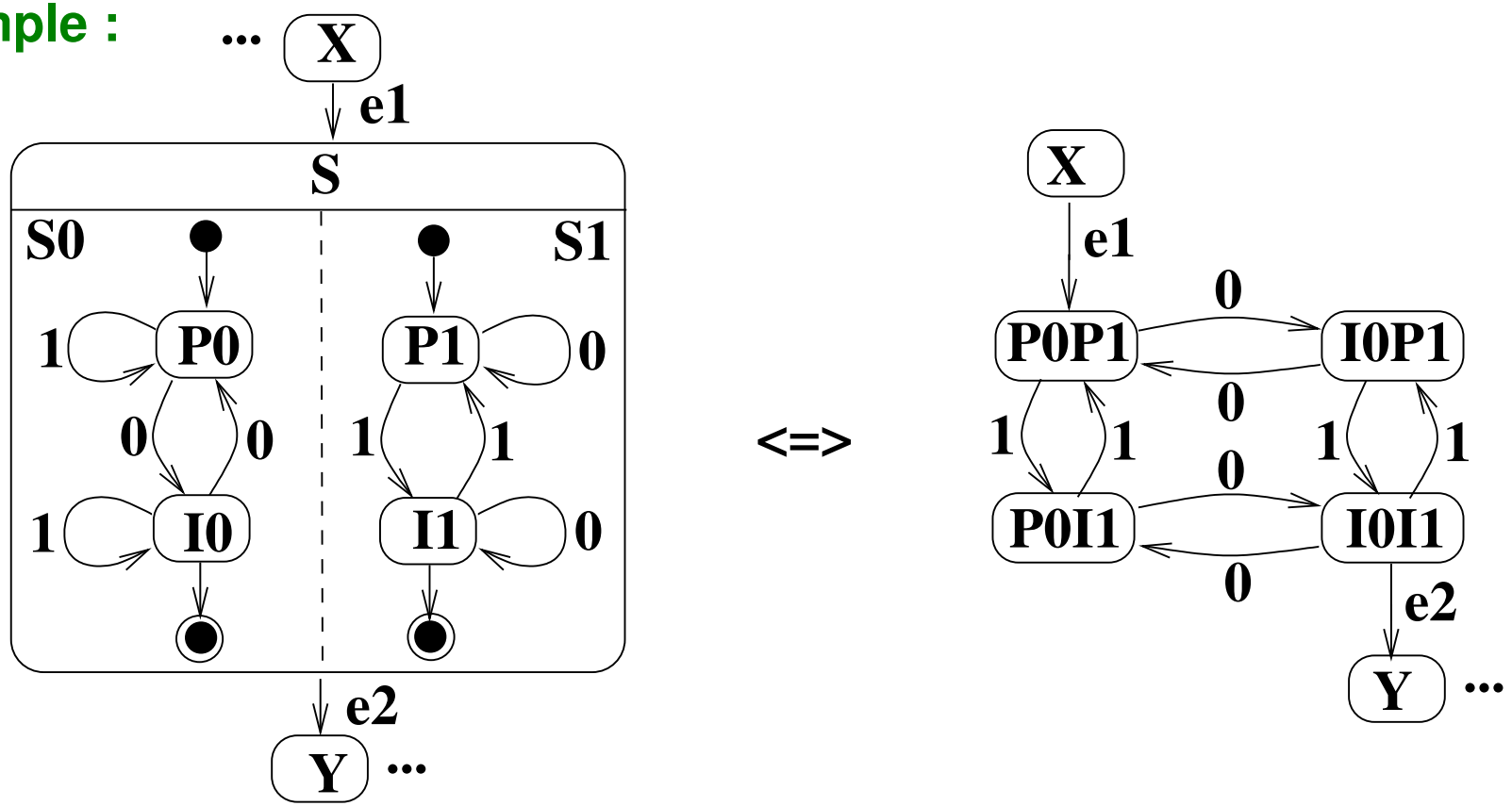


Etats concurrents (2/2)

Cas où les sous-automates sont complets :

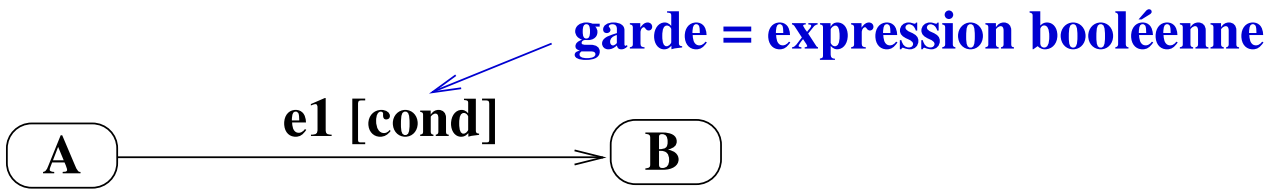
- Equivalent au produit d'automates finis
  → Intersection des langages reconnus par les sous-automates

Exemple :



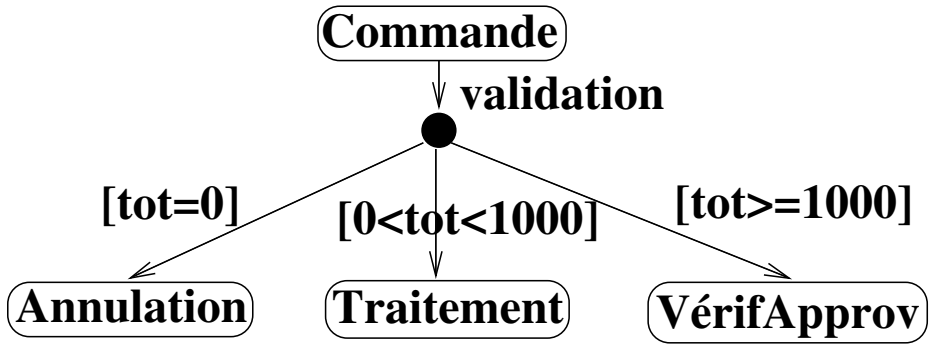
Gardes et transitions composites

Conditions de garde :



- Transition de A vers B réalisée si `cond` est vraie quand `e1` arrive
  → Si `cond` est fausse alors `e1` est perdu

Transitions composites :



- Factorisation de l'événement déclencheur `validation`
- Les gardes doivent être mutuellement exclusives pour que l'automate soit déterministe

Actions et activités

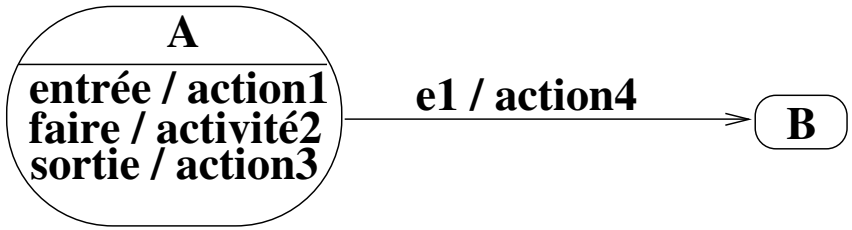
Actions (envoi de signaux, invocation d'opérations, ...) :

- Peuvent être exécutées :
  - Lors d'une transition (ex. : `action4`)
  - En entrant dans un état (ex. : `action1`)
  - En sortant d'un état (ex. : `action3`)
- Sont atomiques (ne peuvent être interrompues par un événement)

Activités :

- Peuvent être exécutées dans un état (ex. : `activité2`)
- Peuvent être continues ou non
- Sont interrompues à l'arrivée d'un événement en sortie de l'état

Exemple :

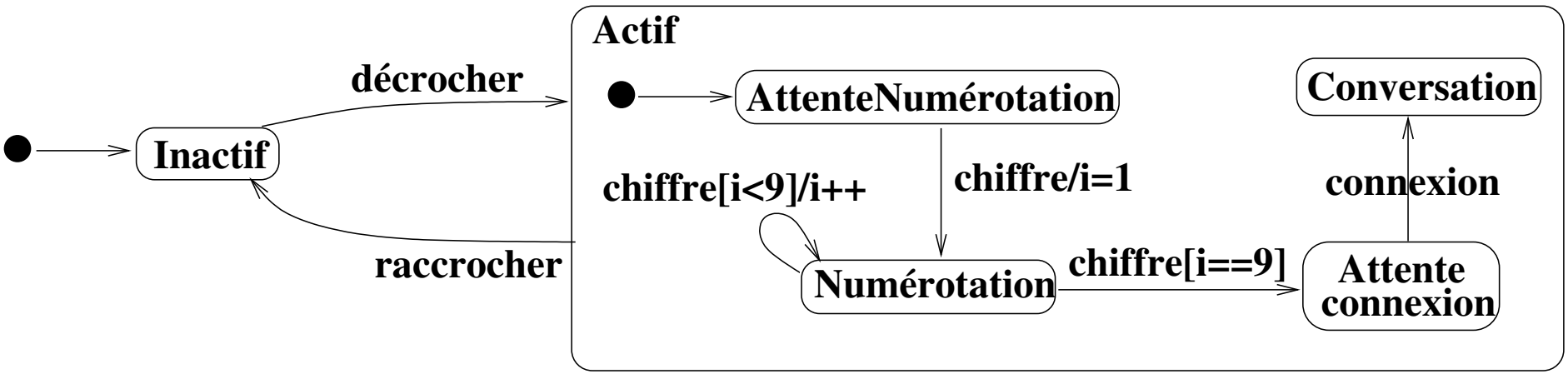


Ordre d'exécution : `action1 - activité2 - action3 - action4`

Quelques conseils...

- Pas de transition sans événement
- L'automate doit être déterministe (en général...)
  - Si plusieurs transitions partant d'un état ont le même événement, alors il doit y avoir des gardes qui garantissent le déterminisme
- Tous les états doivent être accessibles depuis l'état initial
- S'il y a des états terminaux alors, pour chaque état non terminal, il doit exister un chemin de cet état vers un état terminal (...en général)

Exemple : Fonctionnement d'un téléphone



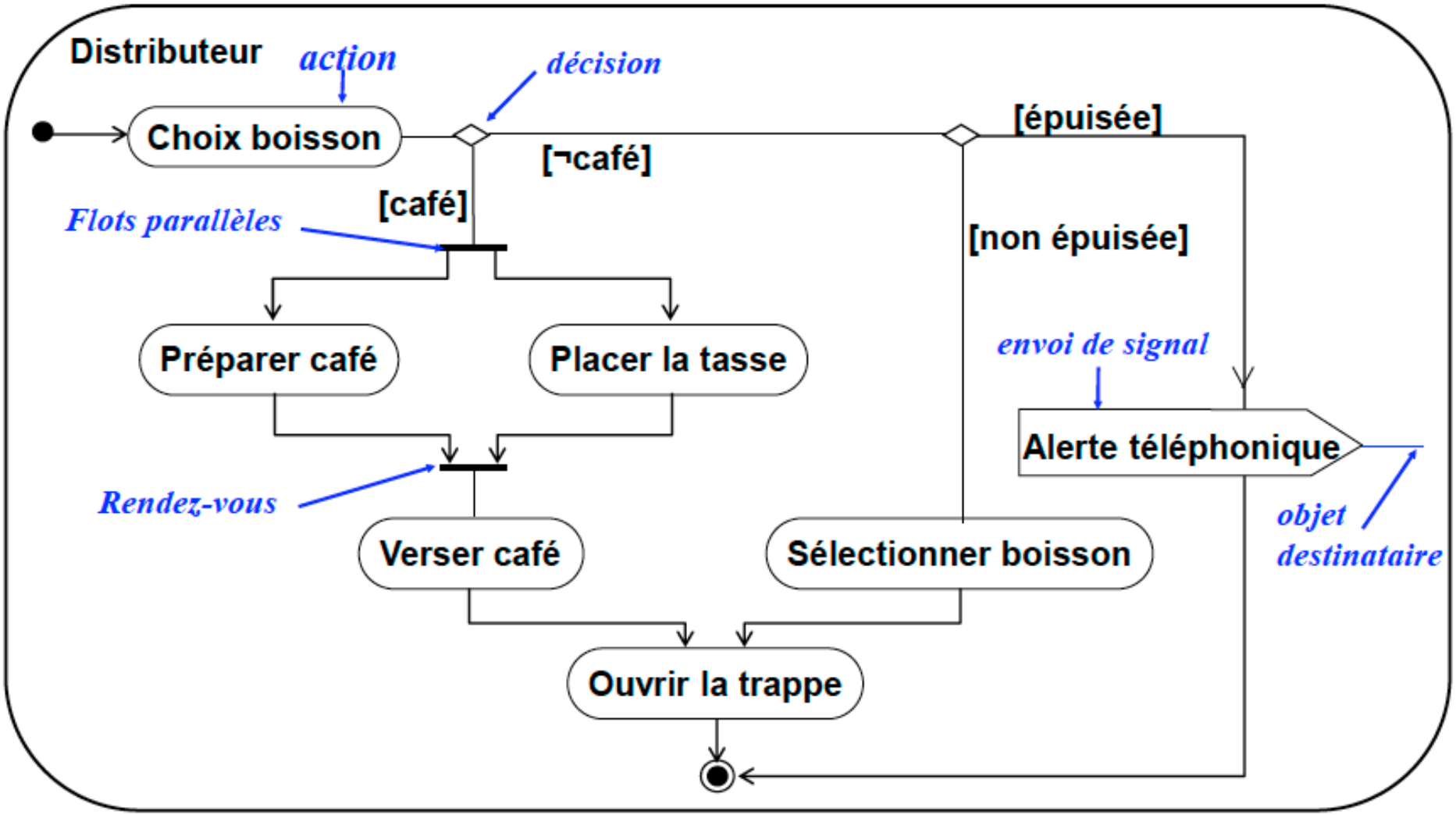
Compléter ce diagramme :

- Emission d'une tonalité quand on décroche
- Emission d'un bip quand on compose un chiffre
- Cas d'un faux numéro
- ...

Diagrammes d'activités

Variante des diagrammes d'états-transitions

→ Modélisation de flux



[Image empruntée à J.-L. Sourrouille]

Plan du cours

De UML à la conception orientée objet

Drawing UML diagrams is a reflection of making decisions about the object design. The object design skills are what really matter, rather than knowing how to draw UML diagrams.

Fundamental object design requires knowledge of :

- *Principles of responsibility assignments*
- *Design patterns*

[Extrait du livre de C. Larman]

Principes de conception orientée objet

Ou comment concevoir des logiciels maintenables et réutilisables

Protection des variations : Identifier les points de variation et d'évolution, et séparer ces aspects de ceux qui demeurent constants

Faible couplage : Réduire l'impact des modifications en affectant les responsabilités de façon à minimiser les dépendances entre classes

Forte cohésion : Faciliter la compréhension, gestion et réutilisation des objets en concevant des classes à but unique

Indirection : Limiter le couplage et protéger des variations en ajoutant des objets intermédiaires

Composer au lieu d'hériter : Limiter le couplage en utilisant la composition (boite noire) au lieu de l'héritage (boite blanche) pour déléguer une tâche à un objet

Programmer pour des interfaces : Limiter le couplage et protéger des variations en faisant abstraction de l'implémentation des objets

Ces principes se retrouvent dans beaucoup de Design Patterns...

Patrons de conception (Design patterns)

Patrons architecturaux vs patrons de conception

- Patrons architecturaux : Structuration globale en paquetages
  → Couches, MVC, Client-serveur, Multi-tiers, ...
- Patrons de conception : Structuration détaillée en classes
  - Attributs et opérations des classes : qui doit savoir, qui doit faire ?
  - Relations entre classes : délégation, héritage, réalisation, ... ?

Description d'un patron de conception

- Nom → Vocabulaire de conception
- Problème : Description du sujet à traiter et de son contexte
- Solution : Description des éléments, de leurs relations/coopérations et de leurs rôles dans la résolution du problème
  → Description générique
  → Illustration sur un exemple
- Conséquences : Effets résultant de la mise en œuvre du patron
  → Complexité en temps/mémoire, impact sur la flexibilité, portabilité, ...

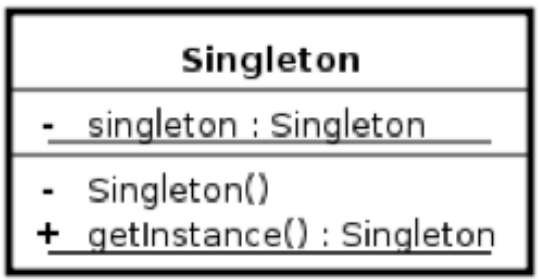
Plan du cours

Singleton

Problème :

Assurer qu'une classe possède une seule instance et rendre cette instance accessible globalement

Solution générique [Wikipedia] :




```
public static synchronized Singleton getInstance(){
    if (_singleton == null)
        _singleton = new Singleton();
    return _singleton;
}
```

Attention :

Parfois considéré comme un anti-pattern... à utiliser avec modération !

Plan du cours

Iterator (1/3)

Problème :

Fournir un accès séquentiel aux éléments d'un agrégat d'objets indépendamment de l'implémentation de l'agrégat (liste, tableau, ...)

Illustration sur un exemple en C++ :

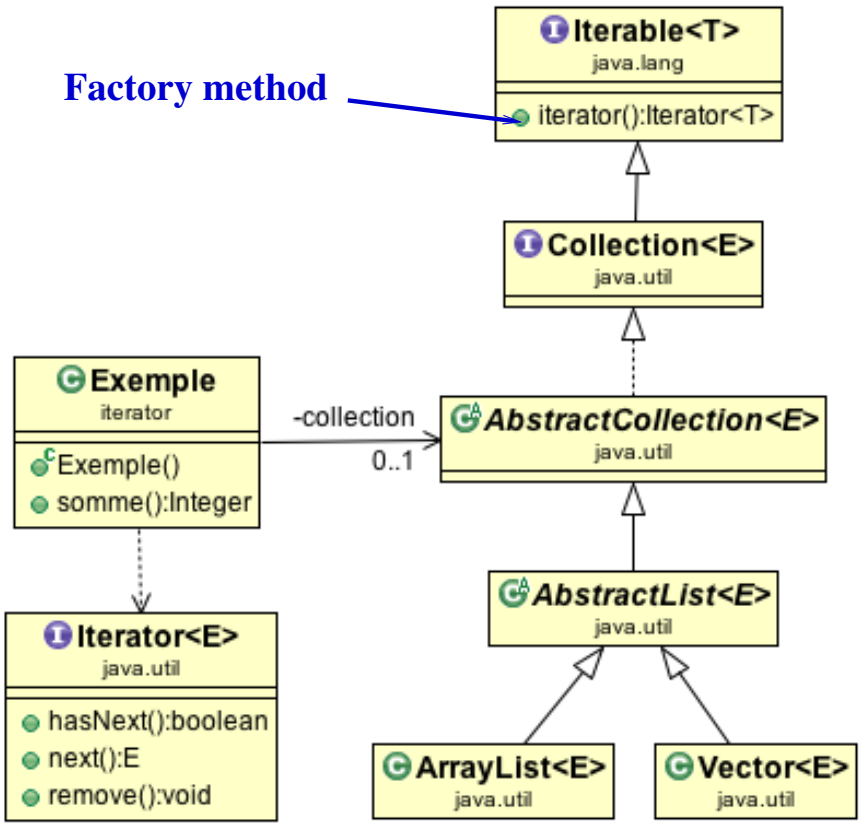
```
class Exemple{
private:
    vector<int> collection;
public:
    int somme(){
        int somme = 0;
        vector<int>::iterator it = collection.begin();
        while (it != collection.end()){
            somme += *it;
            it++;
        };
        return somme;
    }
};
```

Avantage : On peut remplacer `vector<int>` par un autre container sans modifier le code de `somme()`

Iterator (2/3)

Illustration sur un exemple en Java :

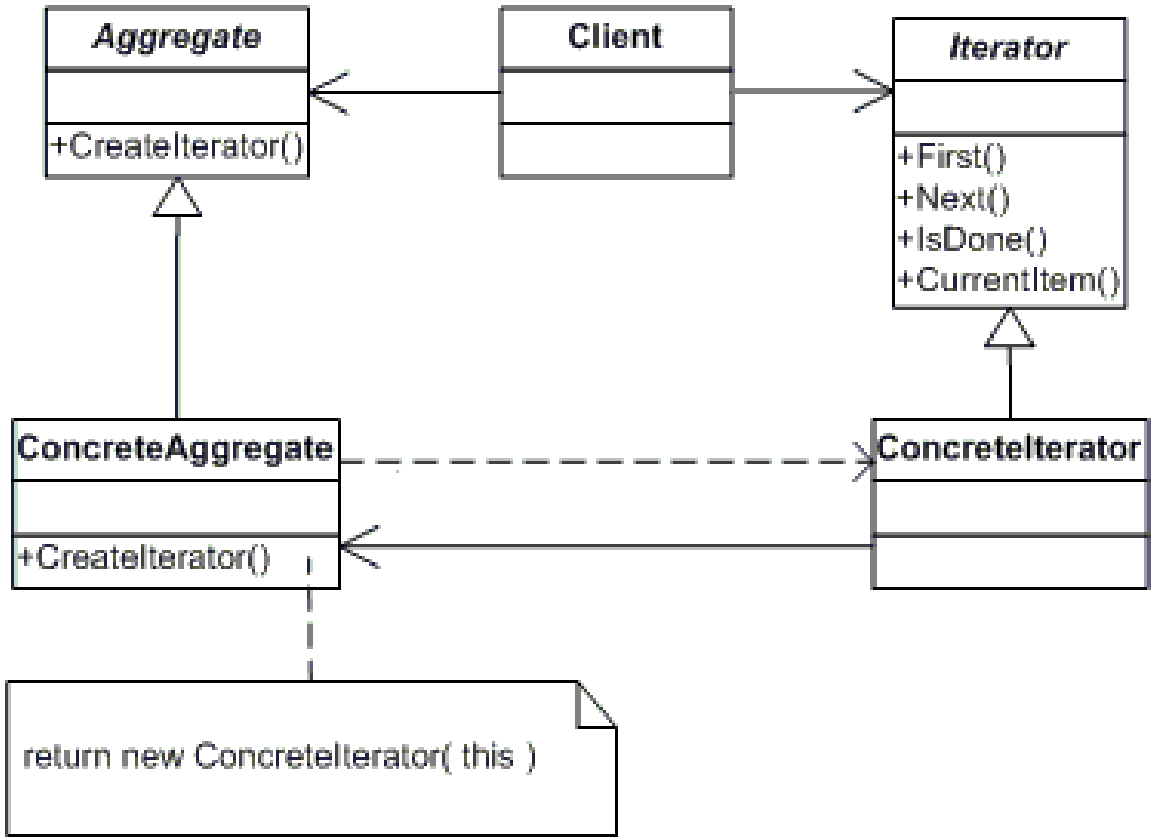
```
public class Exemple {
    private AbstractCollection<Integer> collection;
    public Exemple(){
        collection = new ArrayList<Integer>();
        // ... ajout d'éléments dans collection
    }
    public Integer somme(){
        Integer somme = 0;
        Iterator<Integer> it = collection.iterator();
        while (it.hasNext())
            somme += it.next();
        return somme;
    }
}
```



- Question : Pourquoi séparer `Iterator` de `Collection` ?
- Exercice : Dessiner le diagramme de séquence de `somme()`

Iterator (3/3)

Solution générique :



Avantages :

- Protection des variations : Client est protégé des variations d'Aggregate
- Forte cohésion : Séparation du parcours de l'agrégation
- Possibilité d'avoir plusieurs itérateurs sur un même agrégat en même tps