

Approche objet avec UML (Unified Modeling Language)

Pr. Jean-Marc Jézéquel

IRISA - Univ. Rennes I

Campus de Beaulieu

F-35042 Rennes Cedex

Tel : +33 299 847 192 Fax : +33 299 842 532

e-mail : jezequel@irisa.fr

<http://www.irisa.fr/prive/jezequel>

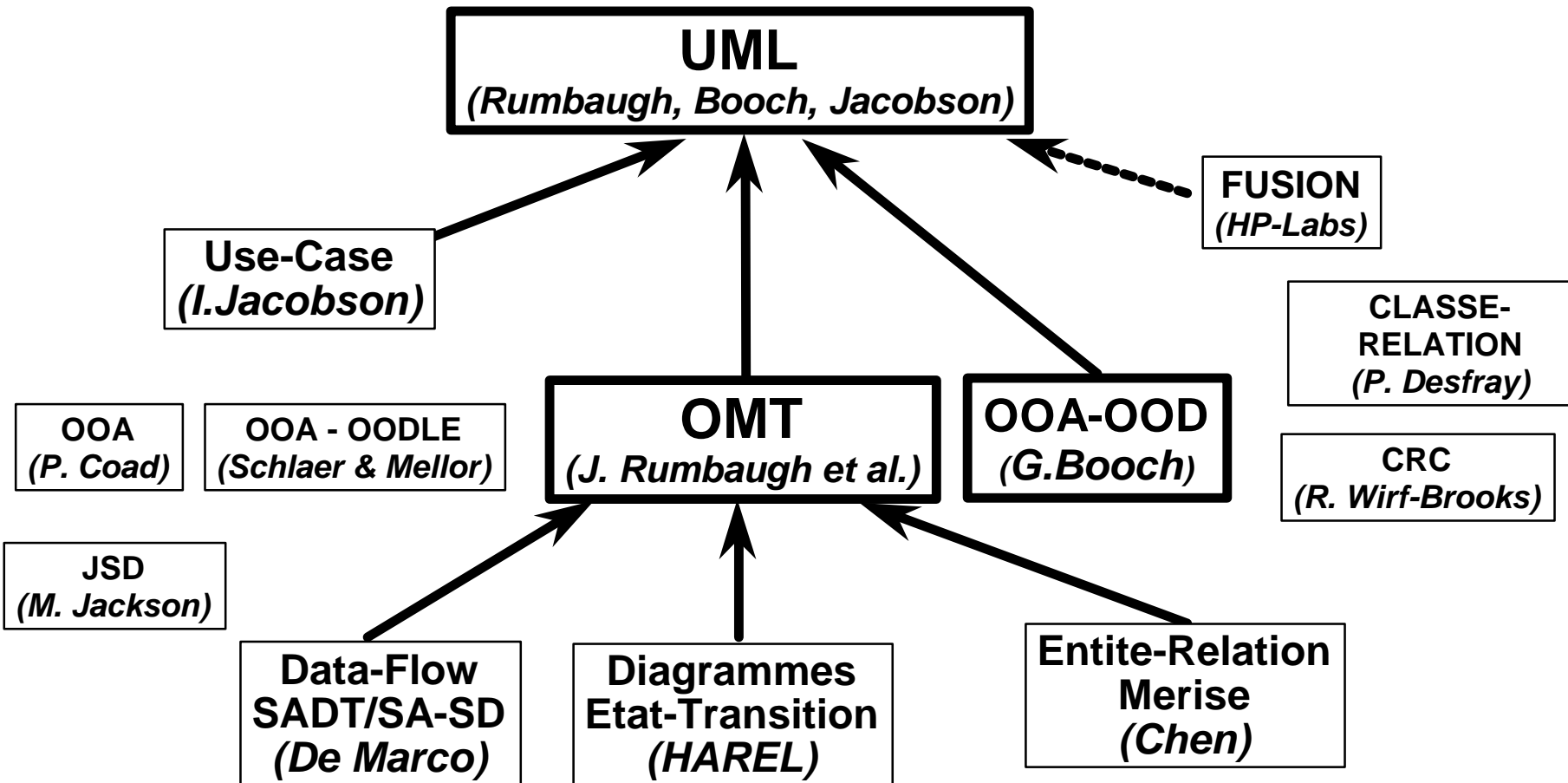
Ingénierie du logiciel

- Problèmes actuels posés en termes de ***lignes de produits*** et de ***maintenabilité***
 - coûts de maintenance > 4 x coûts de développement
 - maintenance évolutive et maintenance corrective
- Solution : Approche par modélisation
 - meilleure continuité entre spécification et réalisation
 - meilleure communication entre les acteurs d'un projet
 - meilleure résistance aux changements

Origines de l'approche objet

- Modélisation => Simulation
- Simula -> Simula 67
 - objet, classe, héritage, liaison dynamique....
- Mais aussi
 - OS : Moniteurs
 - ADT : classe abstraite
 - IA : frame
 - » unité autonome de connaissance
 - » intelligence, gestion complexité = propriété émergente du système

Généalogie de UML



Un peu de Méthodologie...

- Une méthode de développement de logiciels, c'est :
 - Une notation
 - » La syntaxe --- 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
 - Analyse
 - » Elaboration d 'un modèle « idéal »
 - Conception
 - » passage du modèle idéal au monde réel
 - Réalisation et Validation

Modélisation UML

- Modélisation selon 4 points de vue principaux :
 - Aspects statiques du système (*le QUI?*)
 - » Description des objets et de leurs relations
 - Modularité, contrats, relations, généricité, héritage
 - » Structuration en paquetages
 - Vision utilisateur du système (*le QUOI?*)
 - » Cas d'utilisation
 - Aspects dynamiques du système (*le QUAND?*)
 - » Diagramme de séquences (scénarios)
 - » Diagramme de collaborations (entre objets)
 - » Diagramme d'états-transitions (Harel)
 - » Diagramme d'activités
 - Vision implantation (*le OÙ?*)
 - » Diagramme de composants et de déploiement

Modélisation UML

■ Modélisation selon 4 points de vue principaux :

- Aspects statiques du système (*le QUI?*)
 - » Description des objets et de leurs relations
 - Modularité, contrats, relations, généricité, héritage
 - » Structuration en paquetages
- Vision utilisateur du système (*le QUOI?*)
 - » Cas d'utilisation
- Aspects dynamiques du système (*le QUAND?*)
 - » Diagramme de séquences (scénarios)
 - » Diagramme de collaborations (entre objets)
 - » Diagramme d'états-transitions (Harel)
 - » Diagramme d'activités
- Vision implantation (*le OÙ?*)
 - » Diagramme de composants et de déploiement



“Objet” (Définition)

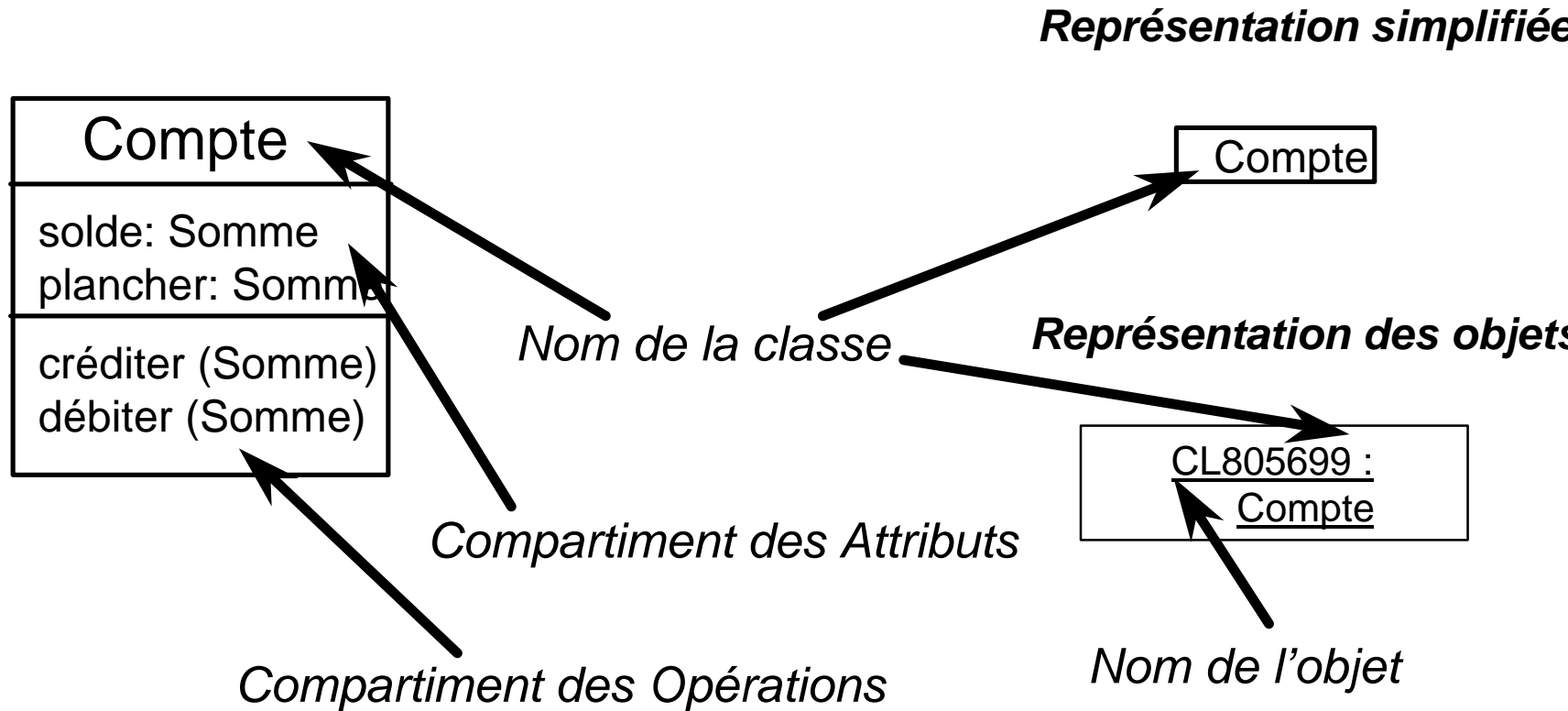
- Formellement : la fermeture transitive d'une fonction
- Concrètement : encapsulation d'un état avec un ensemble d'opérations travaillant sur cet état
 - abstraction d'une entité du monde réel
 - existence temporelle :
 - » création, évolution, destruction
 - identité propre à chaque objet
 - peut être vu comme une machine
 - » ayant une mémoire privée et une unité de traitement,
 - » et rendant un ensemble de services

Définition en temps que type

- Implantation d'un type de donné abstrait
- Description des propriétés et des comportements communs à un ensemble d'objets
 - un objet est une instance d'une classe (eq. variable vs. type)
- Chaque classe a un nom, et un corps qui défini :
 - les attributs possédés par ses instances
 - les opérations définies sur ses instances,
 - » et permettant d'accéder, de manipuler, et de modifier leurs attributs
 - une classe peut elle même être un objet (*Smalltalk*)

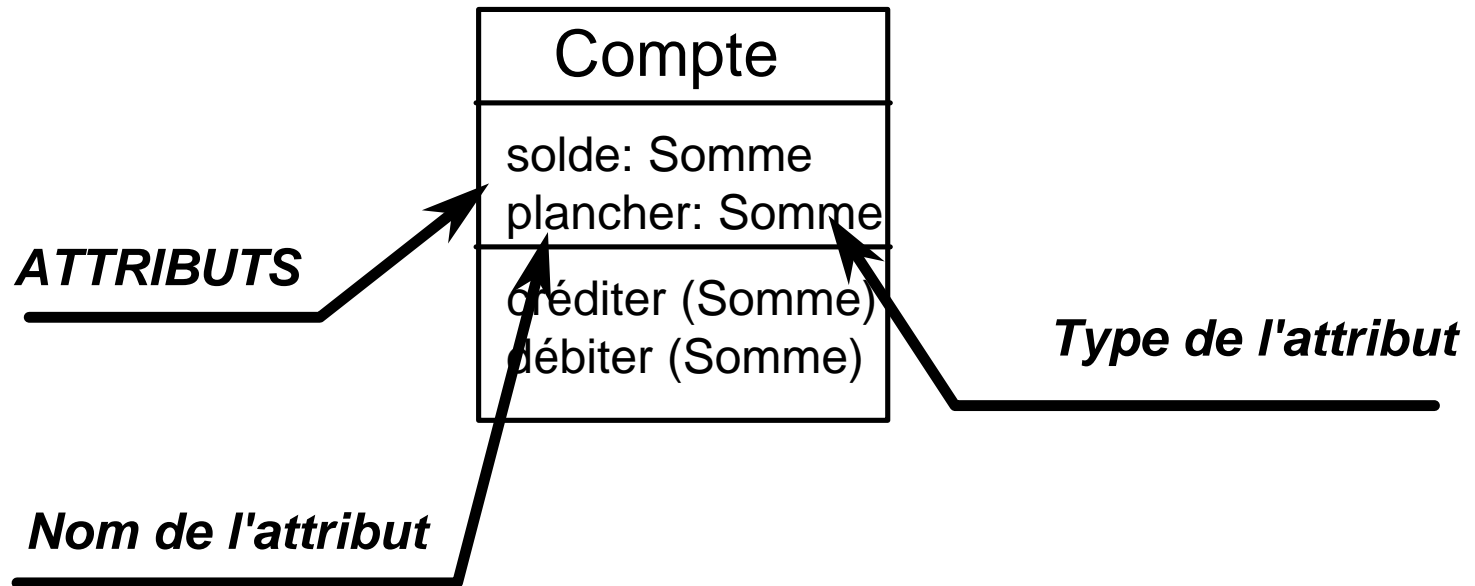
Notations UML pour classes et objets

■ Représentation d'une classe



Représentation des attributs

■ Caractérisation des attributs

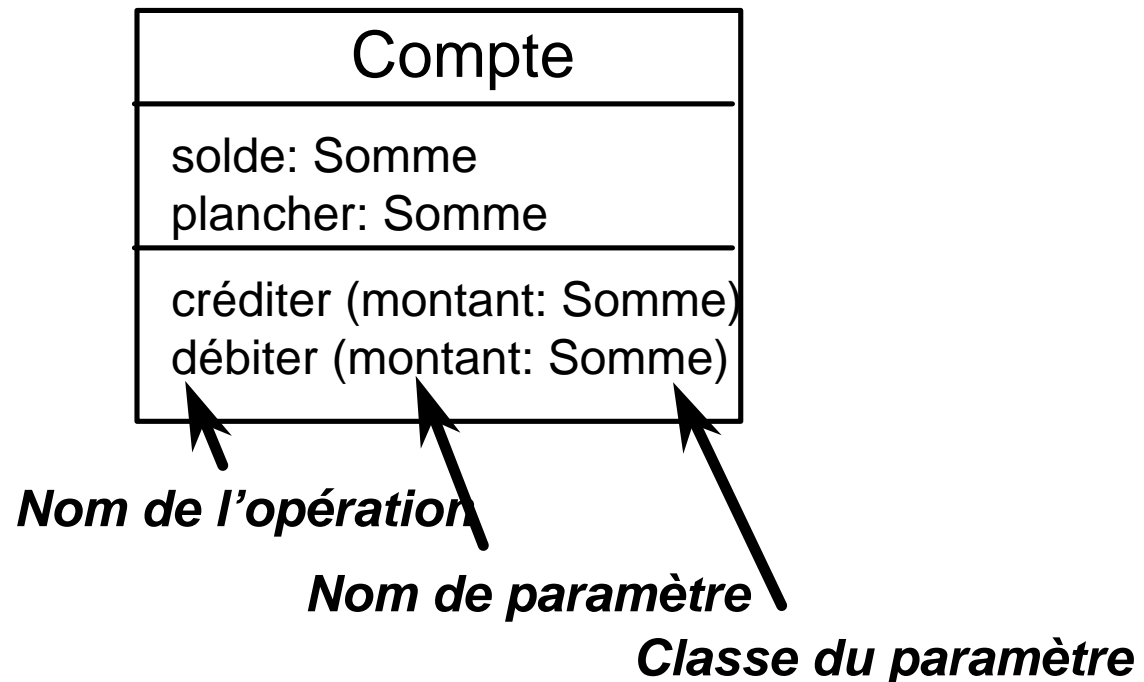


Attributs dérivés

- Attributs 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

Représentation des opérations

■ Vues graphiques

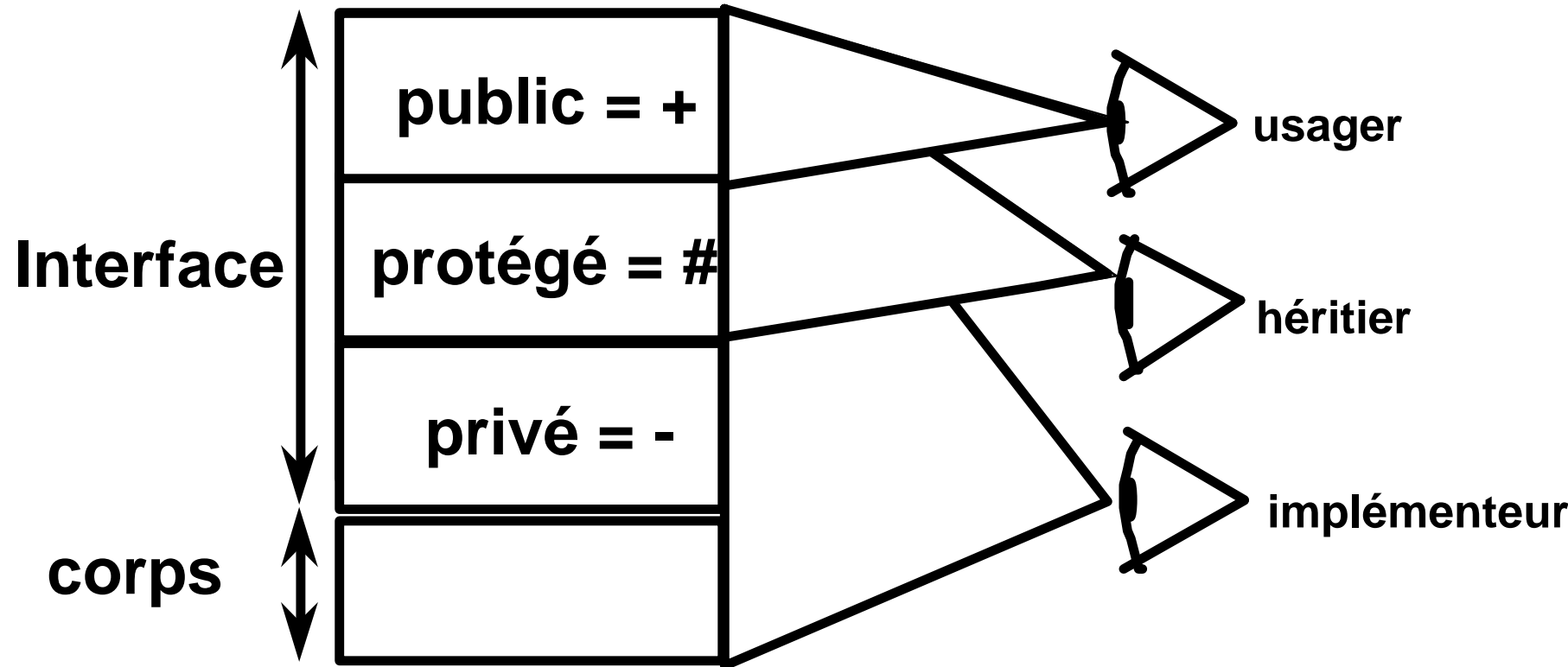


Définition en temps que module

- Modularité = Gestion complexité des systèmes
- Concept présent depuis longtemps en informatique
 - subroutines, unité de compilation (le fichier en C)
- Notion de module intégrée aux langages
 - Modula-2 (module), Ada83 (package), puis lang. à objets
- Favorise :
 - masquage d'information (abstraction)
 - encapsulation (facilite les modifications à portée locale)
 - dissociation interface/implantation=> composant réutilisable

Visibilité

- Différentes visibilités des membres d'une classe



Visibilité : Exemple

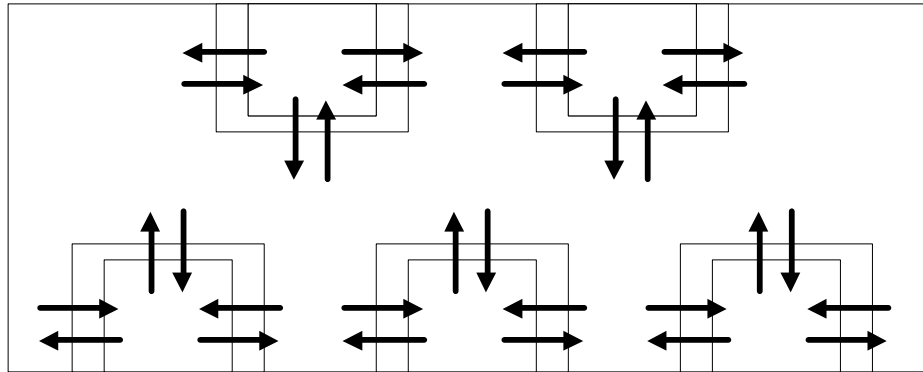
■ Représentation

Classe
+a1 : T1 -a2 : T2
#m1 (p1,P2,p3) +m2 (p1,P2,p3)

Principe de l'approche objet

- Structurer les systèmes autour des objets
 - Plutôt qu'autour des fonctions
- L'approche par modélisation facilite
 - Communication (et V&V)
 - » avec donneur d'ordre (Validation)
 - » entre les activités de dvp et de maintenance (Vérification)
 - Continuité entre les différentes phases du cycle de vie
 - » cf. Jackson et JSD
- Obtenir des systèmes modulaires et maintenables
 - Notion de *lignes de produits*
 - Assemblage de briques de base vs. dvp ad-hoc
 - Produire des *canevas d'application* vs. un programme

Approche OO : Modélisation et Composants



■ Canevas d'applications (*frameworks*)

- Composants changeables même après déploiement
- Garanties ?

Fonctionnelles , synchronisation, performances, QdS

Facteurs de qualité externes

■ Validité

- Aptitude à réaliser les tâches définies par sa spécification
Spécifications informelles => objectif difficile à atteindre

■ Robustesse

- Aptitude à fonctionner même dans des conditions anormales
Réaction non catastrophique à des situations non prévues par la spécification, forcément incomplète

■ Extensibilité

- Facilité d'adaptation d'un logiciel aux changements (correctifs ou évolutifs) de spécification
éviter les logiciels château de cartes

Facteurs de qualité externes

■ Réutilisabilité

- Aptitude à être réutilisé en partie pour de nouvelles applications
Cesser de réinventer, re-coder, et surtout, re-tester

■ Compatibilité

- Aptitude des logiciels à pouvoir être combinés entre eux
Paradigme de l'envoi de messages

■ Efficacité

- Bonne utilisation des ressources du matériel:
processeurs, mémoires, communications, etc.
A ne pas confondre avec temps réel...

Facteurs de qualité externes

■ Portabilité

- Facilité avec laquelle un produit logiciel peut-être adapté à différents environnements matériels et logiciels
 - » souvent incompatible avec efficacité optimale...

■ Vérifiabilité

- Facilité de préparation des procédures de validation (tests), de recette et de certification

■ Ergonomie

- Facilité avec laquelle les utilisateurs d'un composant peuvent apprendre à l'utiliser et à en tirer le meilleur parti
 - documentation à jour (=> extraite du code)

Les concepts de l'approche objet

- Pour aller vers la réalisation des objectifs de qualité externe, il faut :
- **modularité** (cf ci-dessus)
- **contrats** (exprimés en OCL)
- **relations** entre objets (abstraient implantation)
- **généricité** (modules paramétrés)
- **héritage** (classification, réutilisation) et liaison dynamique

Problème de la validité des composants

■ Intra-composant

- validité d'un composant *isolé*
 - » hypothèse : environnement correct
 - » problème : correction vis-à-vis d'une spécification

■ Inter-composants

- validité d'un *assemblage* de composants
 - » hypothèse : chaque composant valide vs sa spécification
 - » problème : *intégration* de systèmes

De la difficulté de la validation intra-composant

```
Acquérir une valeur positive n
Tant que n > 1 faire
    si n est pair
        alors n := n / 2
    sinon n := 3n+1
Sonner alarme;
```

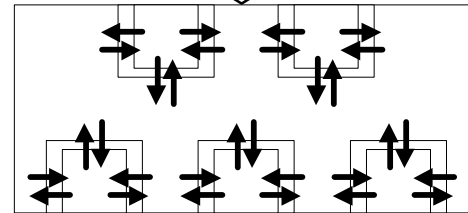
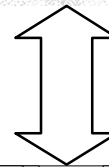
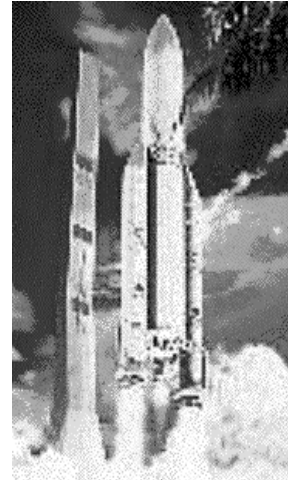
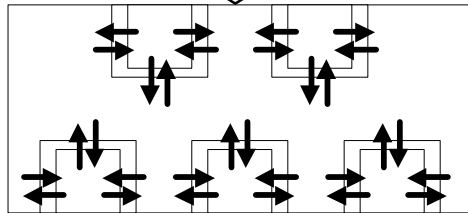
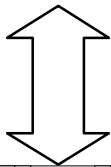
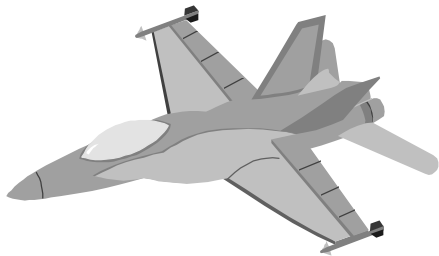
- Prouver que l'alarme est sonnée pour tout n?
- Indécidabilité de certaines propriétés
 - problème de l'arrêt de la machine de Turing...

■ Recours au test

- ici, si machine 32 bits, $2^{31} = 10^{10}$ cas de tests
- **5 lignes de code => 10 milliards de tests !**

Validité inter-composants :

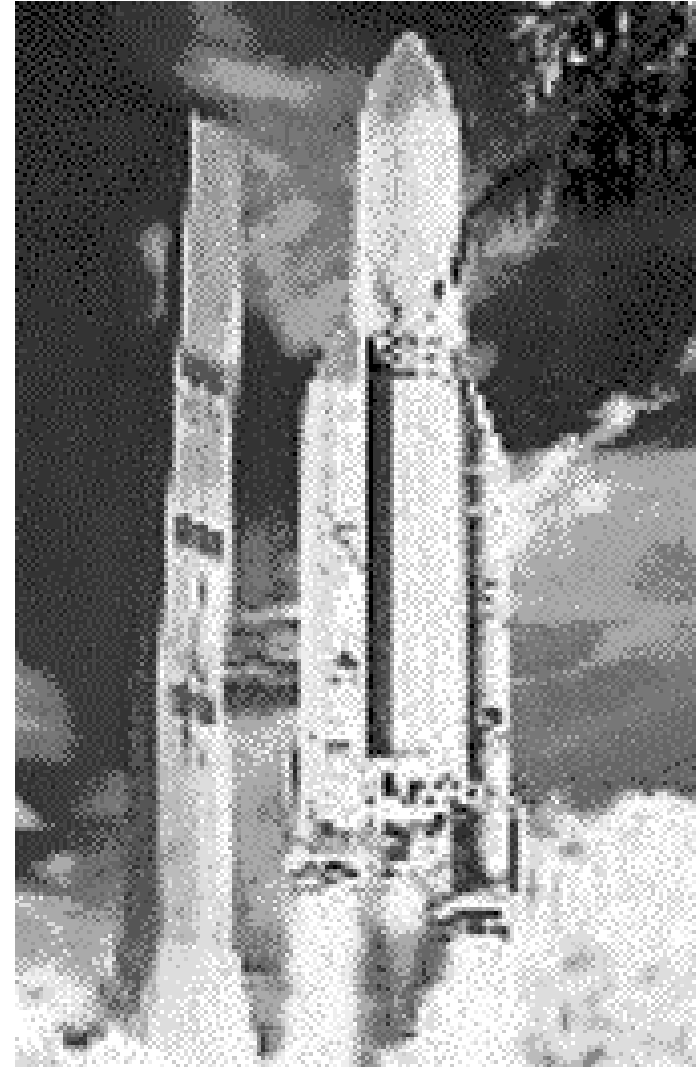
Peut-on (ré)-utiliser un composant?



Ariane 501 Vol de qualification

Kourou, ELA3 -- 4 Juin 1996, 12:34 UT

- H0 -> H0+37s : nominal
- Dans SRI 2:
 - BH (Bias Horizontal) > 2^{15}
 - `convert_double_to_int(BH)` fails!
 - exception SRI -> crash SRI2 & 1
- OBC disoriented
 - Angle attaque > 20° ,
 - charges aérodynamiques élevées
 - Séparation des boosters



Ariane 501 : Vol de qualification

Kourou, ELA3 -- 4 Juin 1996, 12:34 UT

- H0 + 39s: auto-destruction (coût: 500M€)



Pourquoi ? (cf. *IEEE Comp.* 01/97)

- Pas une erreur de programmation
 - Non-protection de la conversion = décision de conception ~1980
- Pas une erreur de conception
 - Décision justifiée vs. trajectoire Ariane 4 et contraintes TR
- Problème au niveau du test d'intégration
 - As always, could have theoretically been caught. But huge test space vs. limited resources
 - Furthermore, SRI useless at this stage of the flight!

Pourquoi? (cf. *IEEE Computer* 01/97)

- Réutilisation dans Ariane 5 d'un composant de Ariane 4 ayant une contrainte « cachée » !
 - Restriction du domaine de définition
 - » Précondition : $\text{abs}(\text{BH}) < 32768.0$
 - Valide pour Ariane 4, mais plus pour Ariane 5

Specification = contrat *entre un composant et ses clients*

- Dans la vie réelle, différents types de contrats
 - Du « Contrat social » de Jean-Jacques Rousseau au “cash & carry”
- De même, plusieurs types de contrats dans un monde réparti

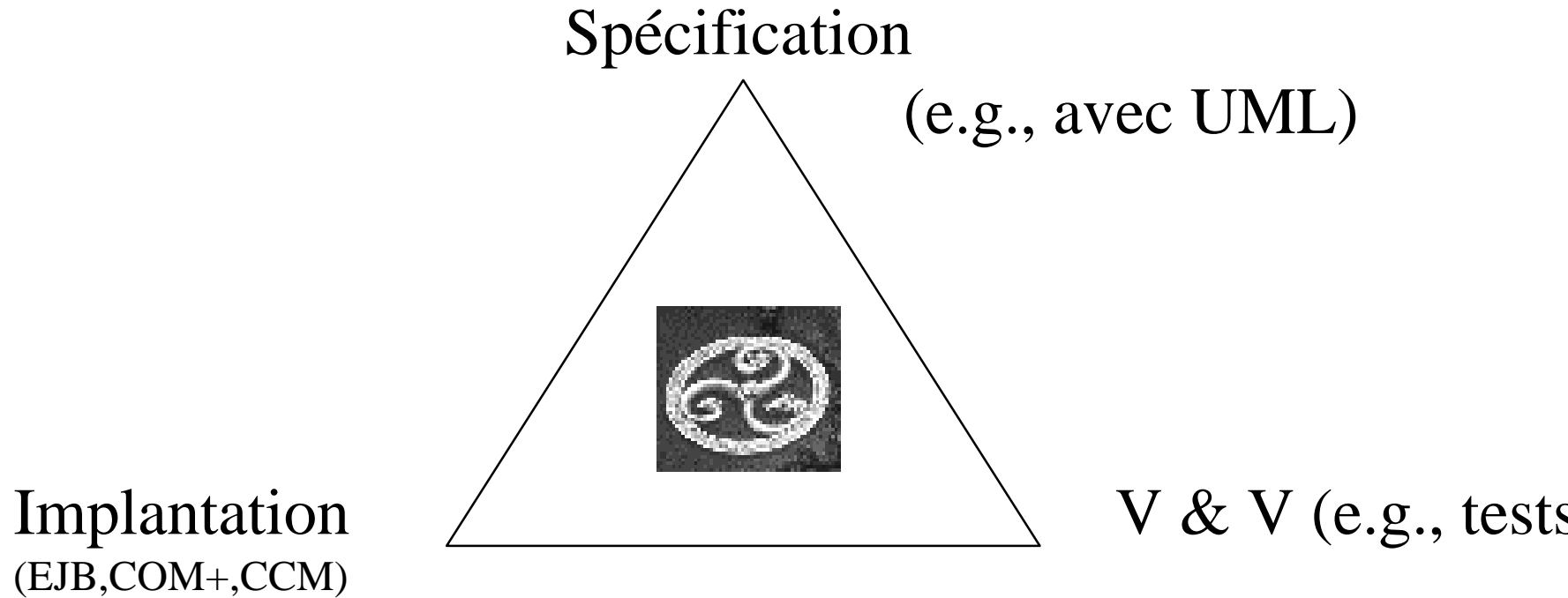


Quatre niveaux de contrats logiciels

- Élémentaire (syntaxique)
 - le programme compile...
- Comportemental (fonctionnel)
 - pré et post conditions
- Synchronisations
 - e.g. *path expressions*, etc. [McHale]
- Qualité de service (quantitative)
 - Négociation dynamique possible

*Cf. IEEE Computer
July 1999*

Composant de confiance?



Confiance = cohérence entre ces 3 aspects

Représentation des contrats en UML

- Typage par signature des méthodes insuffisant
 - besoin de pouvoir exprimer des restrictions
 - » valeurs d'entrées et de sorties
 - besoin de préciser la sémantique
 - » ce que fait une méthode (le quoi) sans entrer dans le détail comment : Indépendance vis-à-vis de l'implantation
- Inspirée par la notion de Type Abstrait de Données:
Spécification = Signature +
 - Préconditions (conditions sous lesquelles une routine peut être appelée)
 - Postconditions (propriétés garanties par une routine)
 - Invariants de classe (vrai à l'entrée et à la sortie des routines)

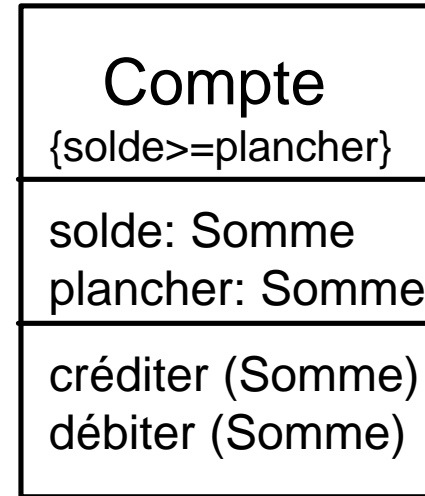
OCL : Object Constraint Language

- Langage de description de contraintes de UML
 - des contraintes restreignant les domaines de valeurs peuvent être ajoutées aux éléments du modèle UML
- Contrainte = expression booléenne (sans effet de bord) portant sur
 - opérations usuelles sur types de base (Boolean, Integer...)
 - attributs d'instances et de classes
 - opérations de « query » (fonctions sans effet de bord)
 - associations du modèle UML
 - états des StateCharts associés

Représentation des contraintes OCL

Directement dans le modèle

- notation entre { } **accrochée à un élément de modèle**



Dans un document séparé,
en précisant le **contexte**

context Compte inv:
solde >= plancher

- Invariants = Propriétés vraies pour l'ensemble des instances de la classe
 - » dans un état stable, chaque instance doit vérifier les invariants de sa classe

Précondition:

ce qui doit être respecté par le client

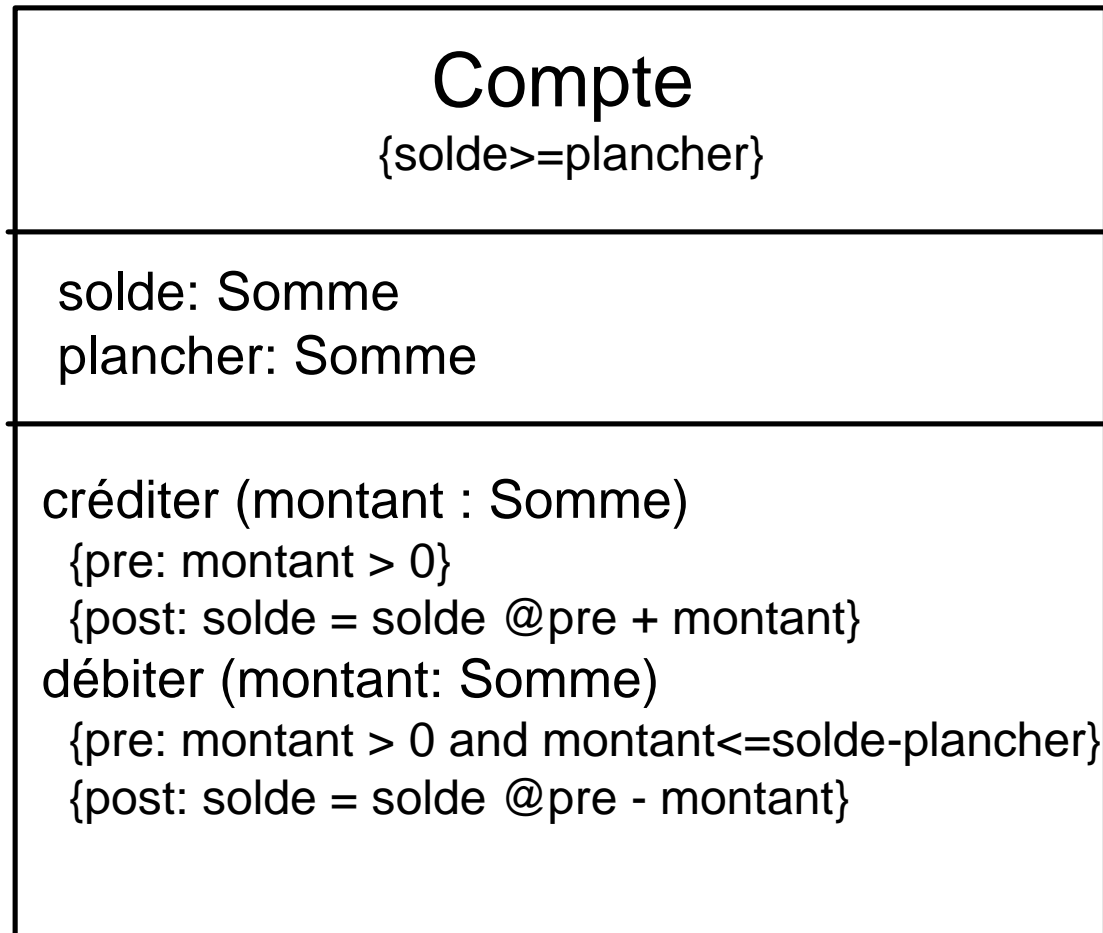
- Spécification des conditions nécessaires pour qu'un client soit autorisé à appeler une méthode
 - exemple: montant > 0
- Notation en UML
 - {«precondition» *OCL boolean expression*}
 - Abbreviation: {pre: *OCL boolean expression*}

Postcondition:

Ce qui doit être assué par l'implantation

- Spécification de ce qui sera vrai à la complétion d'un appel valide à une méthode
 - exemple: `solde = solde @pre + montant`
- Notation en UML
 - {«postcondition» *OCL boolean expression*}
 - Abbreviation: {post: *OCL boolean expression*}
 - Opérateur pour accéder à la valeur « d'avant » (idem *old Eiffel*):
 - » *OCL expression @pre*

Etre abstrait et précis avec UML



Analyse précise ou “analyse par contrat”

Conception par Contrat

- Le contraire de la *Programmation Défensive*
 - Essayer de tester tout ce qui pourrait poser problème
 - » code lourd et complexe à maintenir et tester
 - » quoi faire lorsqu'on détecte un problème?
- Assertions des contrats :
 - jouent un rôle crucial dans la séparation nette des responsabilités dans un système modulaire
 - contrat entre l'appelant d'une routine (le client) et l'implantation de la routine (le contractant) :

Pourvu que le client appelle la routine dans des conditions où l'invariant de classe du contractant et la précondition de la routine sont respectés, alors le contractant promet que lorsque la routine terminera, le travail spécifié dans la postcondition sera effectué, et l'invariant de classe sera respecté.

Intérêt pratique des contrats

- Specification, documentation
 - *Not a software fault tolerance gadget*
 - *Might help system fault tolerance...*
- Help V&V
 - When assertions are monitored
 - Never doing debugging again
- Help allocate responsibilities during integration
 - No longer have to find a scapegoat ;-)

Contract Violations: Preconditions

- The client broke the contract.
 - The provider does not have to fulfill its part of the contract.
 - If contracts are monitored, an exception should be raised
 - » making it easy to identify the exact origin of the fault.

Unhandled exception: Routine failure. Exiting program.

Exception history:

=====		
Object Routine		
Type of exception	Description	Line
=====		
#<BANK_ACCOUNT5f0c0>		BANK_ACCOUNT:deposit
precondition violated	positive_amount	63

#<USER 5f000>		USER:test
Routine failure		90

#<DRIVER 5f010>		DRIVER:make
Routine failure		18

Contract violations: Postconditions

- The implementation of a method did not comply with its promise: This is a bug

Unhandled exception: Routine failure. Exiting program.

Exception history:

=====		
Object Routine		
Type of exception	Description	Line
=====		
#<BANK_ACCOUNT5f0c0>		BANK_ACCOUNT:deposit
postcondition violated	deposited	70

#<USER 5f000>		USER:test
Routine failure		90

#<DRIVER 5f010>		DRIVER:make
Routine failure		18

- Again, easy to trace...(between lines 63-70)

Modélisation UML

- Modélisation selon 4 points de vue principaux :
 - Aspects statiques du système (*le QUI?*)
 - » Description des objets et de leurs relations
 - Modularité, contrats, relations, généricité, héritage
 - » Structuration en paquetages
 - Vision utilisateur du système (*le QUOI?*)
 - » Cas d'utilisation
 - Aspects dynamiques du système (*le QUAND?*)
 - » Diagramme de séquences (scénarios)
 - » Diagramme de collaborations (entre objets)
 - » Diagramme d'états-transitions (Harel)
 - » Diagramme d'activités
 - Vision implantation (*le OU?*)
 - » Diagramme de composants et de déploiement

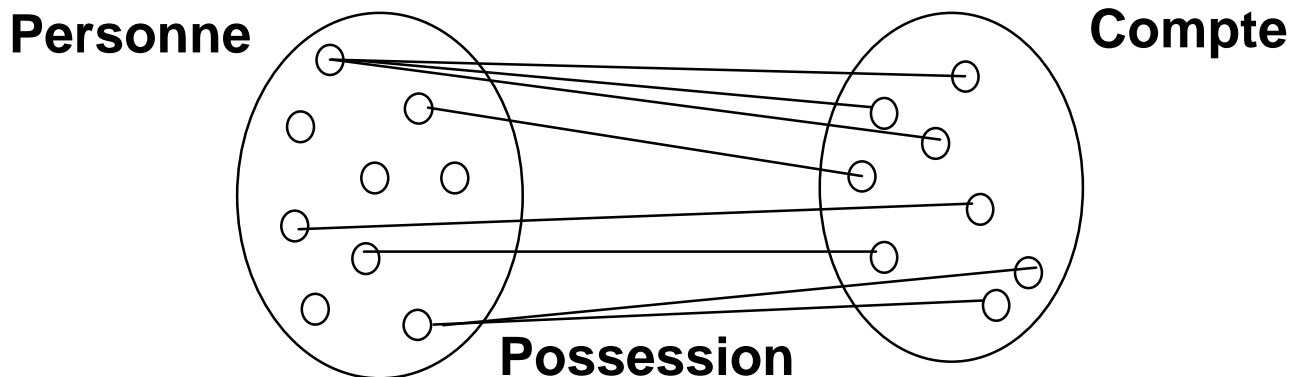


Relation entre classes

- Deux points de vue :
 - Une relation met en correspondance des éléments d'ensembles
 - Une relation permet la description d'un concept à l'aide d'autres concepts
- Une contrainte :
 - Une relation est un lien stable entre deux objets

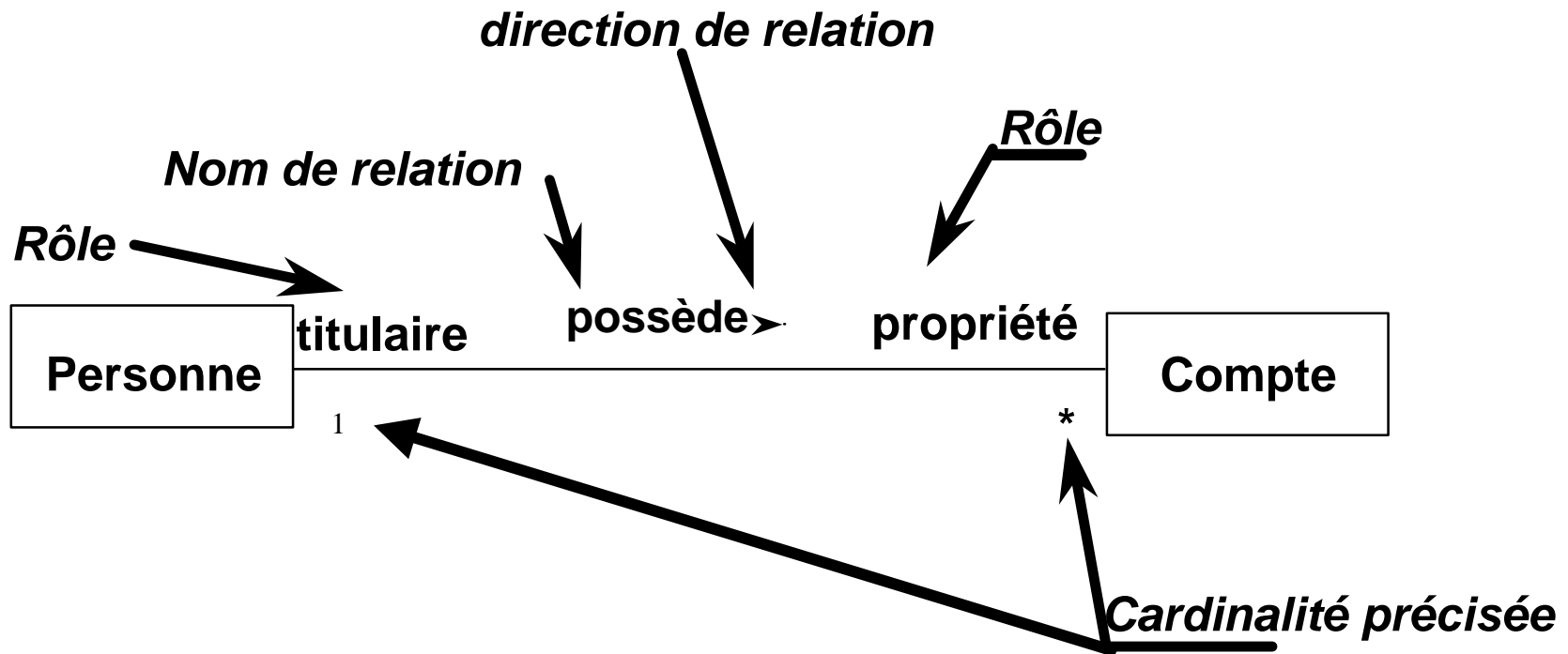
Vue ensembliste d'une relation :

Graphe de la relation

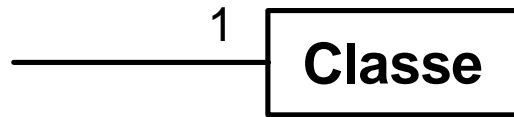


**Une association met en correspondance
des éléments d'ensembles**

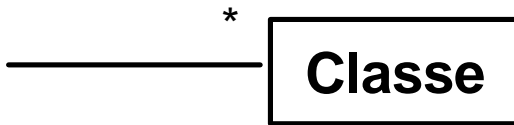
Représentation des relations : direction, rôle, cardinalité



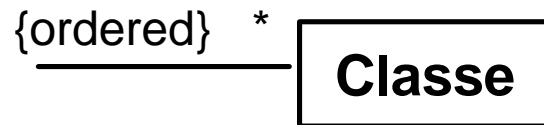
Cardinalité d'une relation



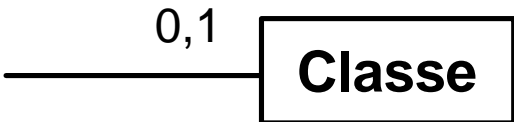
Exactement une



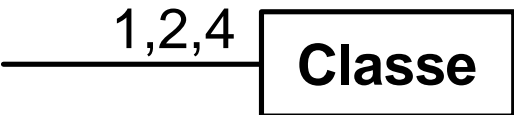
Plusieurs (0 à n), non ordonnés



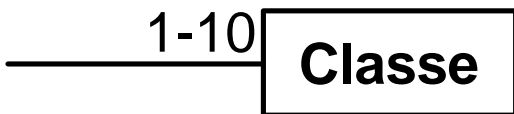
Plusieurs (0 à n), ordonnés



Optionnelle (0 ou 1)



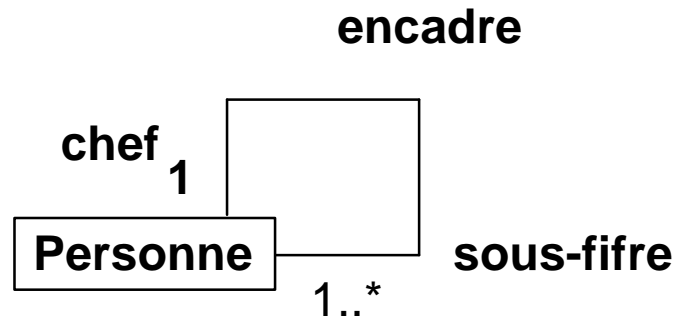
Cardinalité spécifiée



Intervalle

Cas particuliers de relations

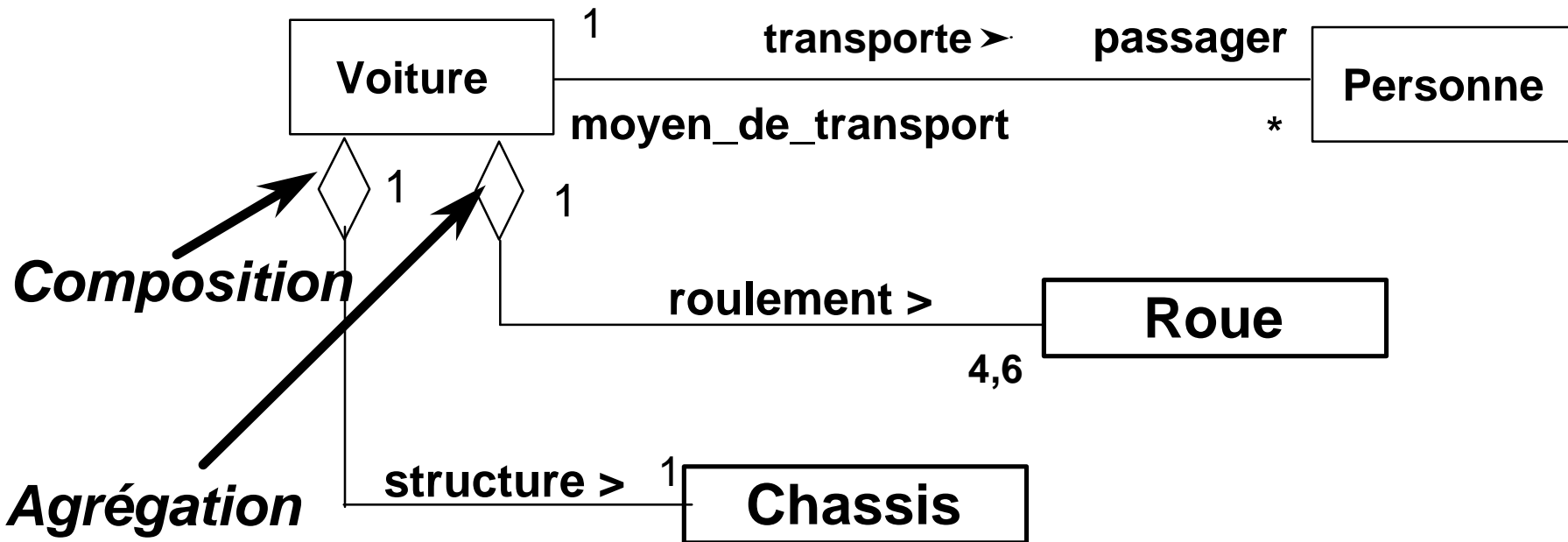
■ Relations réflexives



Une relation réflexive lie des objets de même classe

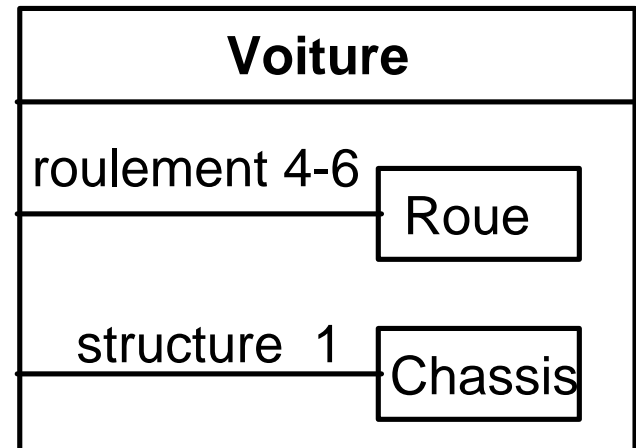
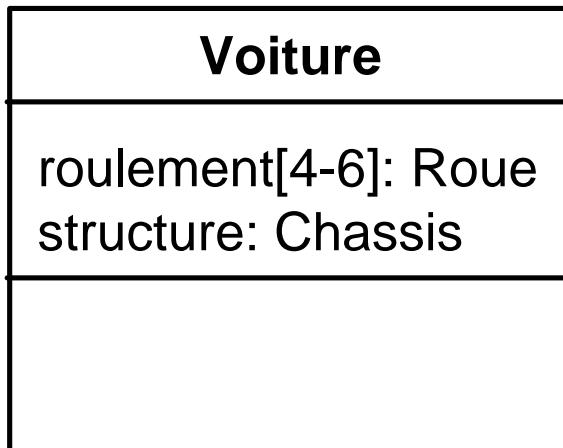
Composition et Agrégation

- Cas particuliers de relations :
 - Notion de *tout* et *parties*



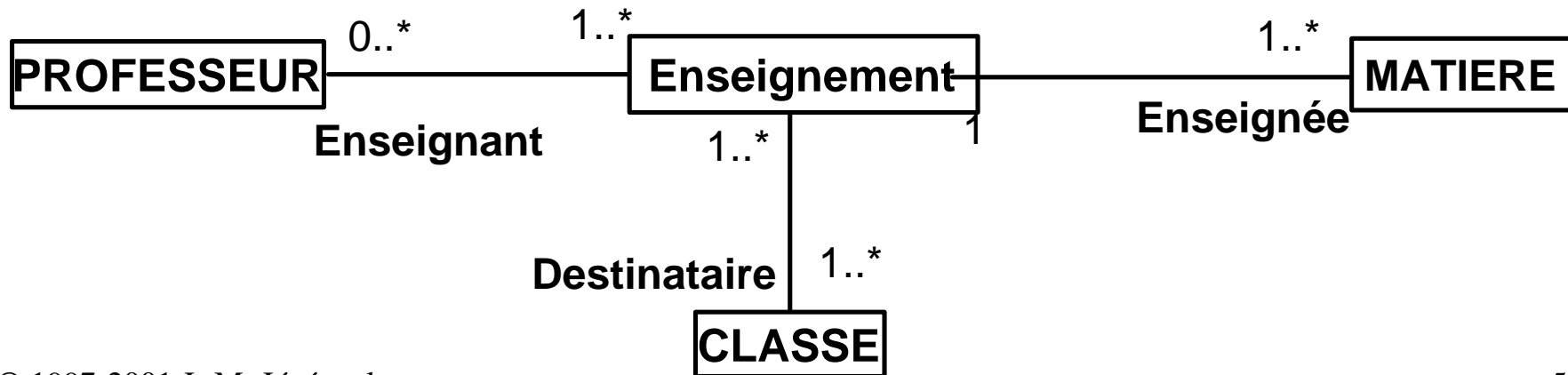
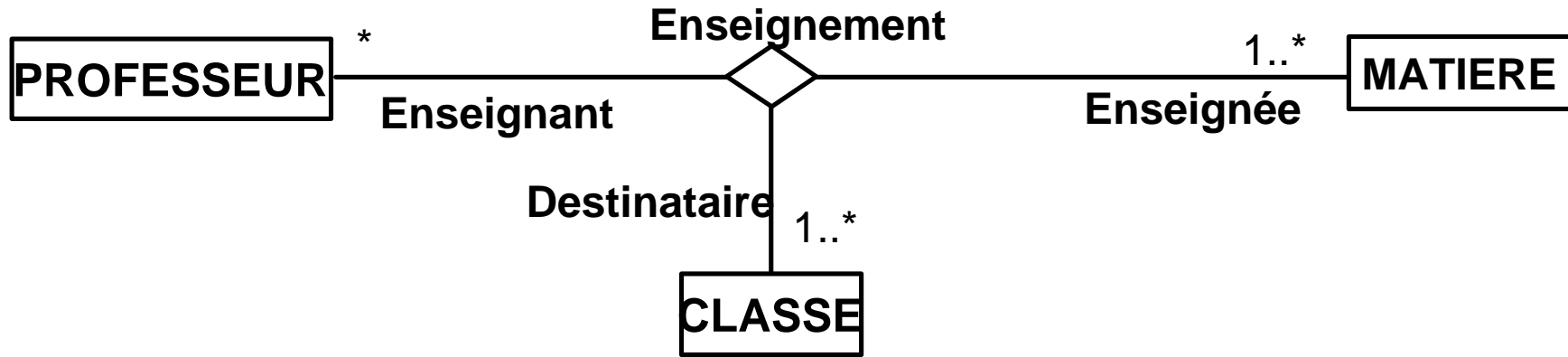
Autre vues de la composition/agrégation

- Différentes formes suggérant *l'inclusion*



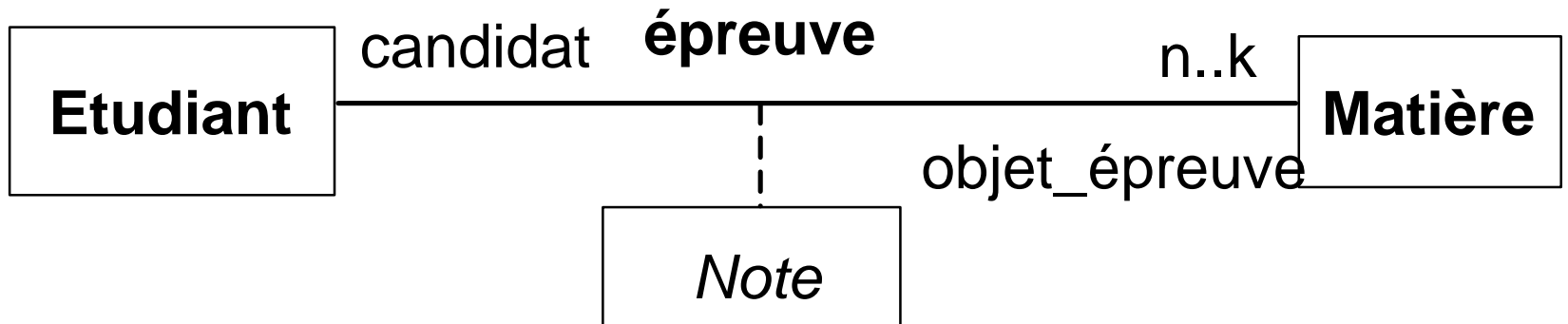
Relations n-aires

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



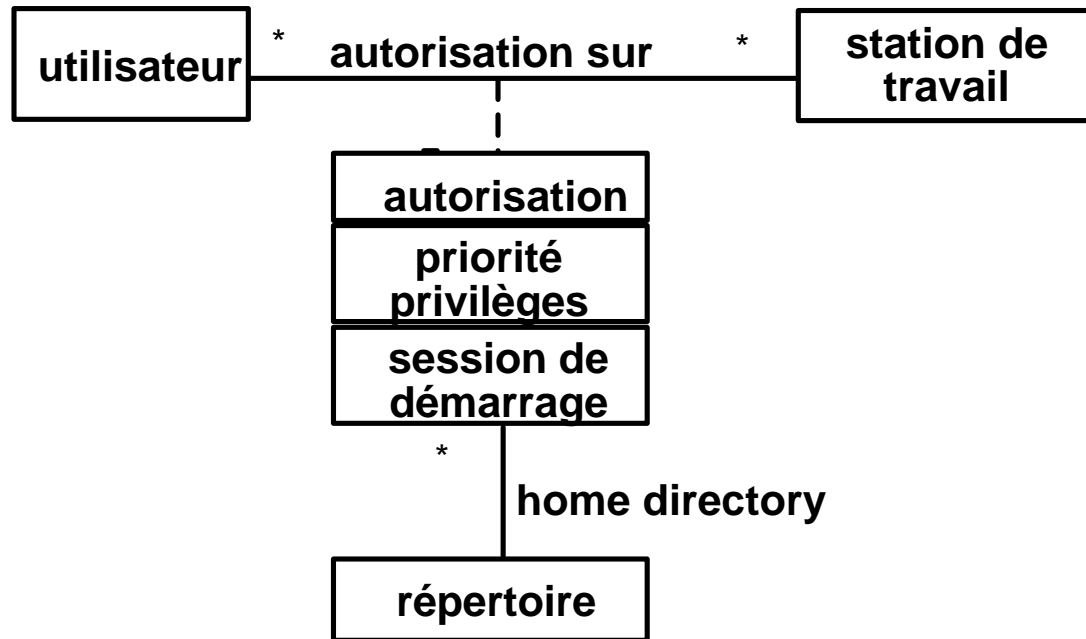
Relations attribuées

- L'attribut porte sur le lien



Les relations en tant que classes

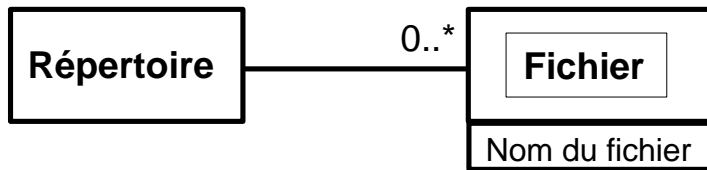
- Pratique dans certains cas
 - Relations ternaires.
 - La relation a des opérations appelées : *classe de liaison*



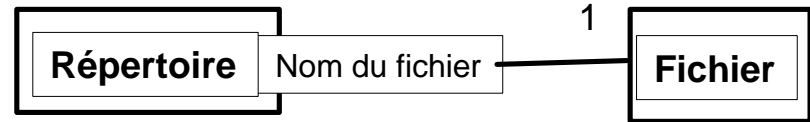
Qualifieurs de relations

- Un qualifieur est un attribut spécial qui permet, dans le cas d'une relation 1-vers-plusieurs ou plusieurs-vers-plusieurs, de réduire la cardinalité. Il peut être vu comme une clé qui permet de distinguer de façon unique un objet parmi plusieurs.

Qualifieurs de relations : Exemple



Relation non qualifiée

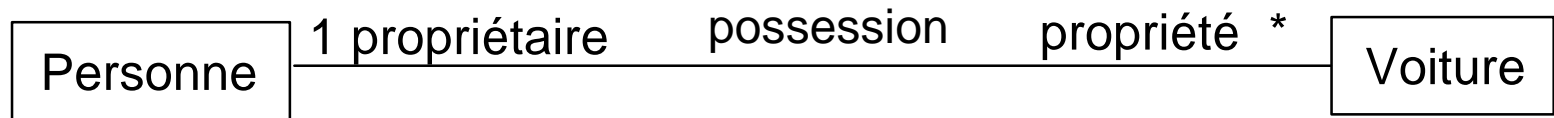


Relation qualifiée

Un répertoire + un nom de fichier
identifient de façon unique un fichier

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 rôles sont utilisés pour identifier quelle relation on veut naviguer



Context Voiture inv:
self.propriétaire.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

Opérations de base sur collections

■ *isEmpty*

- vrai si la collection n'a pas d'éléments

Context Personne inv:
age<18 implique propriété->isEmpty

■ *notEmpty*

- vrai si la collection a au moins un élément

■ *size*

- nombre d'éléments dans la collection

■ *count (elem)*

- nombre d'occurrences de *elem* dans la collection

Opération *collect*

- Syntaxes possibles
 - collection->collect(elem:T | expr)
 - collection->collect(expr)
 - collection.expr
- Par exemple :
 - context Personne inv:
propriété->collect(passager)->count(self)=1
- ou en raccourci :
 - context Personne inv:
propriété.passager->count(self)=1

Opération *select*

- Syntaxes possibles
 - `collection->select(elem:T | expr)`
 - `collection->select(elem | expr)`
 - `collection->select(expr)`
- Sélectionne le sous-ensemble de *collection* pour lequel la propriété *expr* est vraie
- e.g.

context Personne inv:

`propriété->select(v: Voiture | v.kilometrage<100000)->notEmpty`

- ou en raccourci :

context Personne inv:

`propriété->select(kilometrage<100000)->notEmpty`

Opération *forAll*

- Syntaxes possibles
 - collection->forall(elem:T | expr)
 - collection->forall(elem | expr)
 - collection->forall(expr)
- Vrai si *expr* est vrai pour chaque élément de *collection*
- e.g.

```
context Personne inv:  
propriété->forall(v: Voiture | v.kilometrage<100000)
```

- ou en raccourci :

```
context Personne inv:  
propriété->forall(kilometrage<100000)
```

Autres opérations OCL

- **exists (expr)**
 - Vrai si *expr* est vrai pour au moins un élément de la collection
- **includes(elem), excludes(elem)**
 - vrai si *elem* est présent (resp. absent) dans la collection
- **includesAll(coll)**
 - vrai si tous les éléments de *coll* sont dans la collection
- **union (coll), intersection (coll)**
 - opérations classiques ensemblistes
- **asSet, asBag, asSequence**
 - conversions de type

Modélisation UML

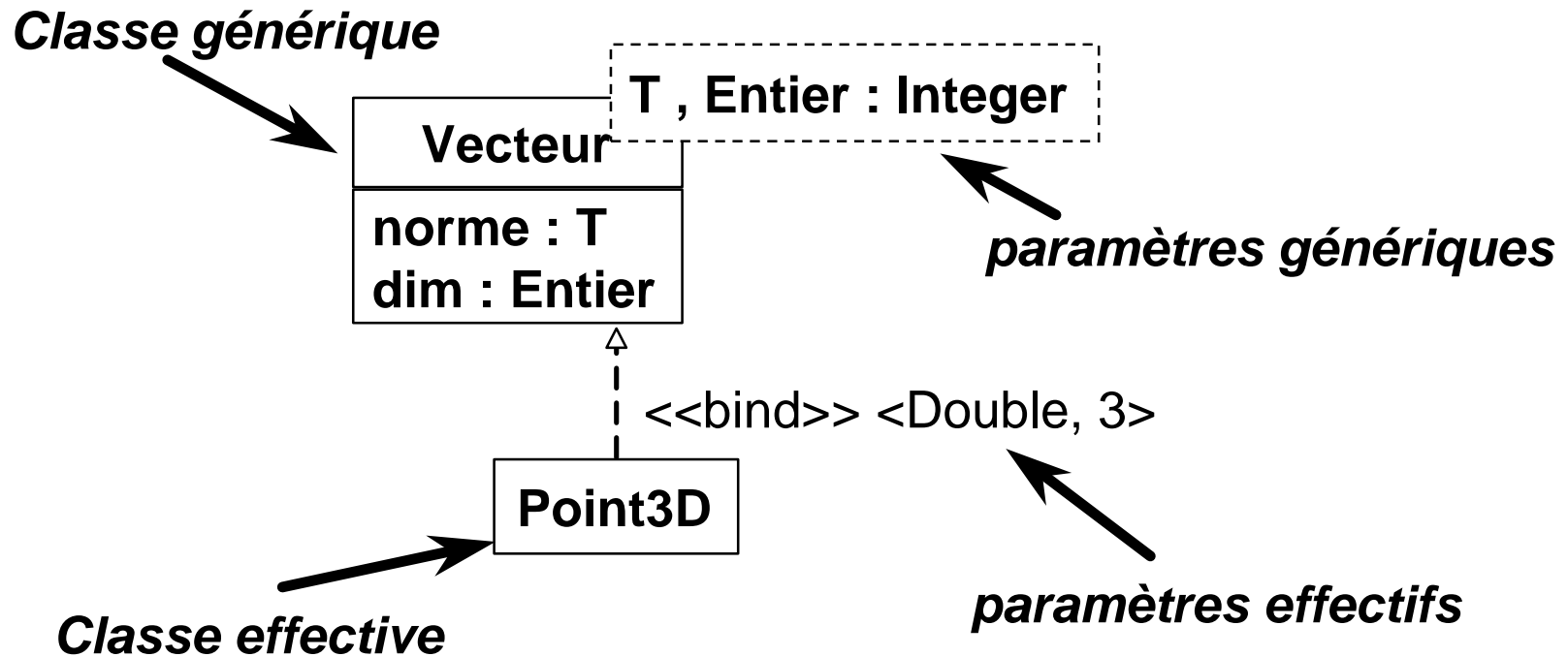
- Modélisation selon 4 points de vue principaux :
 - Aspects statiques du système (*le QUI?*)
 - » Description des objets et de leurs relations
 - Modularité, contrats, relations, généricité, héritage
 - » Structuration en paquetages
 - Vision utilisateur du système (*le QUOI?*)
 - » Cas d'utilisation
 - Aspects dynamiques du système (*le QUAND?*)
 - » Diagramme de séquences (scénarios)
 - » Diagramme de collaborations (entre objets)
 - » Diagramme d'états-transitions (Harel)
 - » Diagramme d'activités
 - Vision implantation (*le OÙ?*)
 - » Diagramme de composants et de déploiement



La généricité (Classes paramétrées)

- Indispensable pour les classes “conteneurs”
 - En Pascal, liste d’entiers, de réels, etc.
 - en C (C++) liste de (void *)
- Solution en Ada : donner un nom formel au type des éléments du conteneur (Liste[T])
 - Solution similaire en Eiffel (idem Ada) ou C++ (templates)
 - N’existe pas en Java = l’un de ses 3 points faibles principaux
 - Pas ce problème en Smalltalk (typage dynamique)
- Classes génériques vs. classes effectives

Représentation de la généricité



Modélisation UML

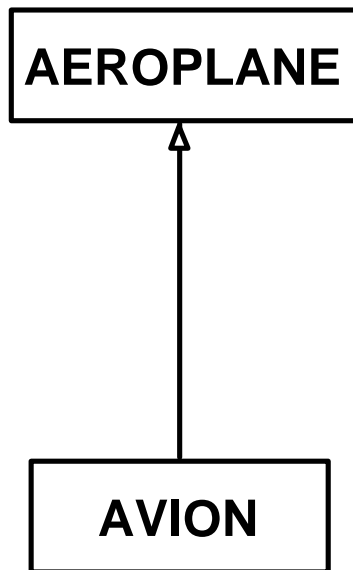
- Modélisation selon 4 points de vue principaux :
 - Aspects statiques du système (*le QUI?*)
 - » Description des objets et de leurs relations
 - » Modularité, contrats, relations, généricité, héritage
 - » Structuration en paquetages
 - Vision utilisateur du système (*le QUOI?*)
 - » Cas d'utilisation
 - Aspects dynamiques du système (*le QUAND?*)
 - » Diagramme de séquences (scénarios)
 - » Diagramme de collaborations (entre objets)
 - » Diagramme d'états-transitions (Harel)
 - » Diagramme d'activités
 - Vision implantation (*le OÙ?*)
 - » Diagramme de composants et de déploiement



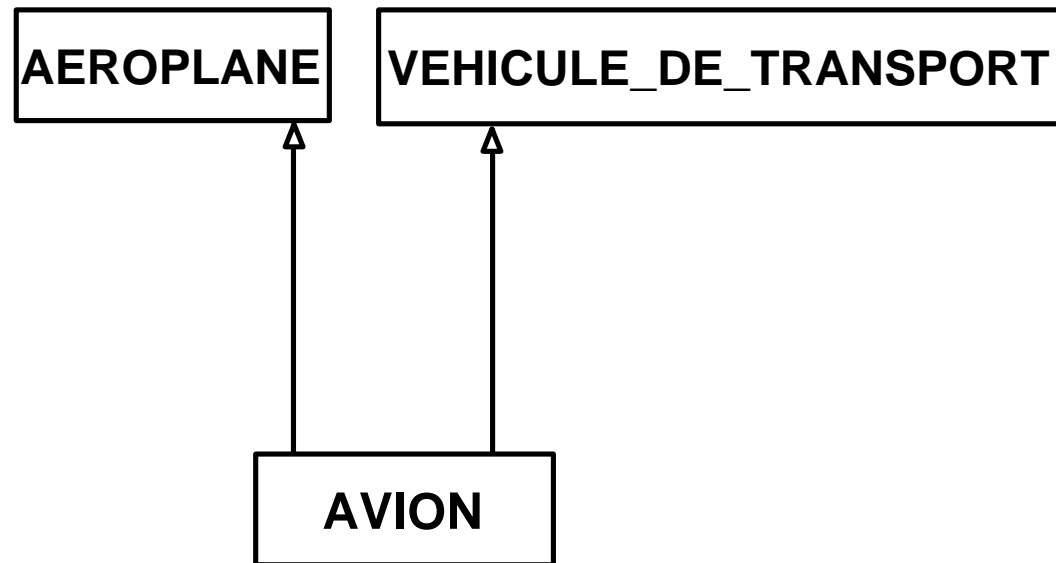
Héritage (généralisation)

- Caractéristique des langages dit “orientés objets”
 - Partage d’attributs et d’opérations entre classes
 - Relation hiérarchique Super-classe/Sous-classe
 - Héritage multiple => graphe orienté (graphe d’héritage)

Héritage simple



Héritage multiple



Utilisation de l'héritage

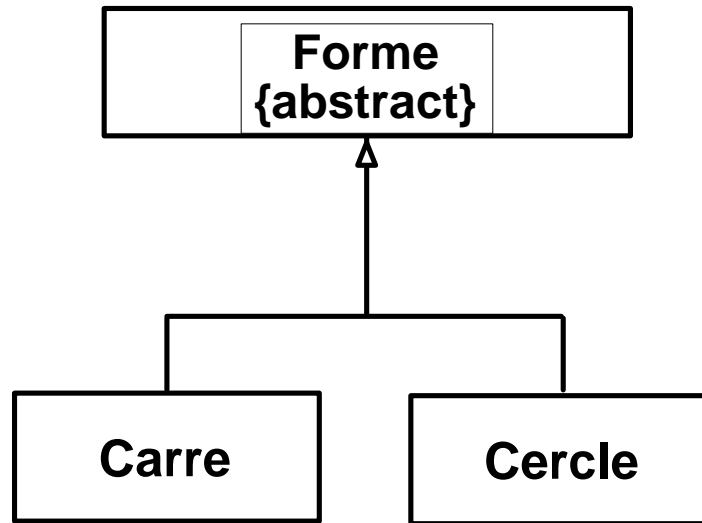
- Mécanisme d'extension de module
 - ajout de fonctionnalités dans la sous-classe
 - “customisation” et combinaison de composants logiciels
 - réutilisation de code
- Mécanisme de classification : sous-typage
 - relation X est-une-sort-de Y (est substituable à)
 - organisation des systèmes complexes (cf. Linnaeus)
 - réutilisation d'interface

Classes abstraites

- Capturent des comportements communs
- Servent à structurer un système
- Ont des opérations dont l'implantation est absente
 - deferred en Eiffel
 - pure virtual en C++
 - abstract en Java
- Ne peuvent donc pas être instanciées
- Sémantique des opérations abstraites spécifiable
 - invariants, pre et post-conditions

Représentation de classes abstraites

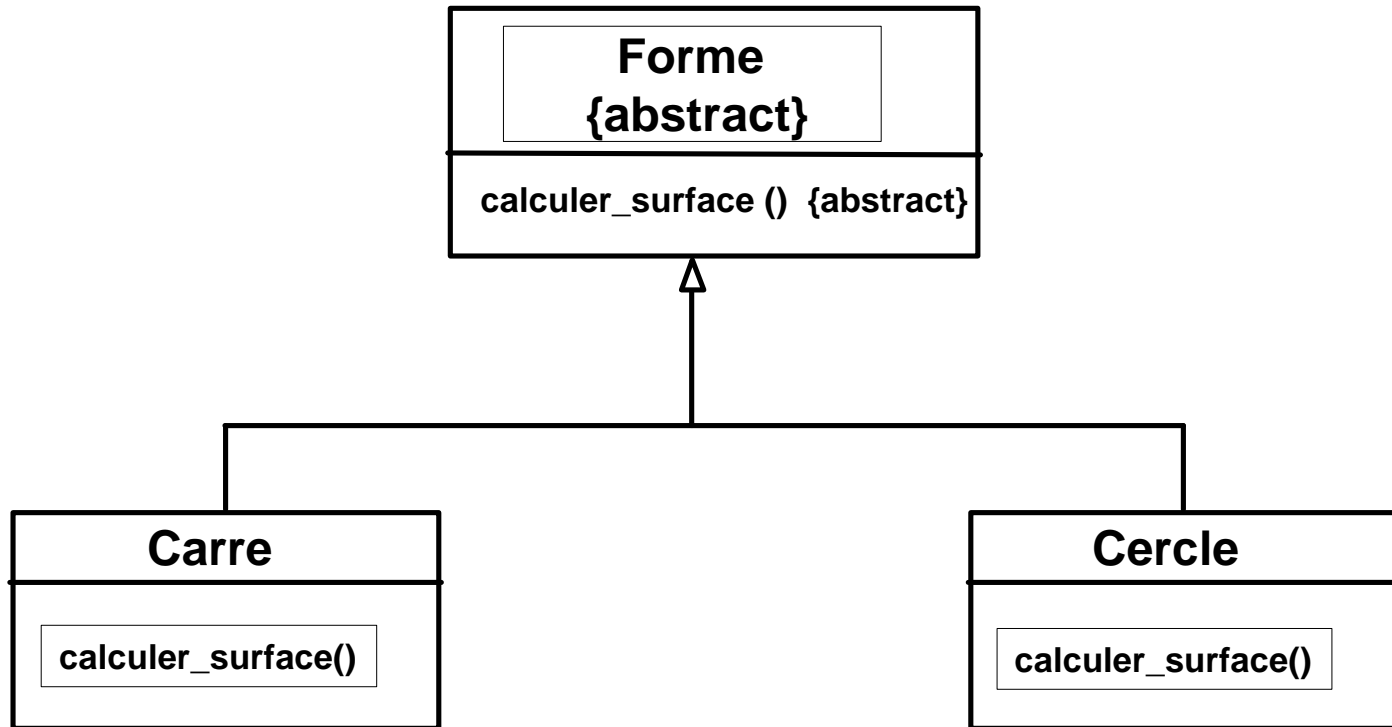
■ Classes sans instances immédiates



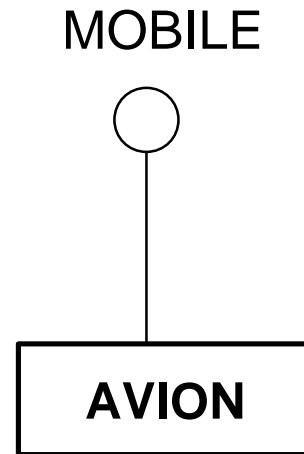
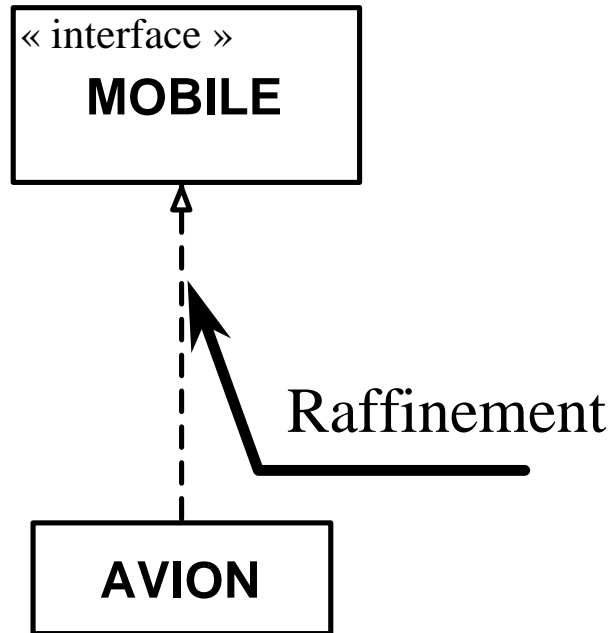
Une instance de «**Forme**» est
obligatoirement une instance de la
classe **Carre** ou de la classe **Cercle**

Représentation des opérations abstraites

- Opération sans corps d'une classe abstraite

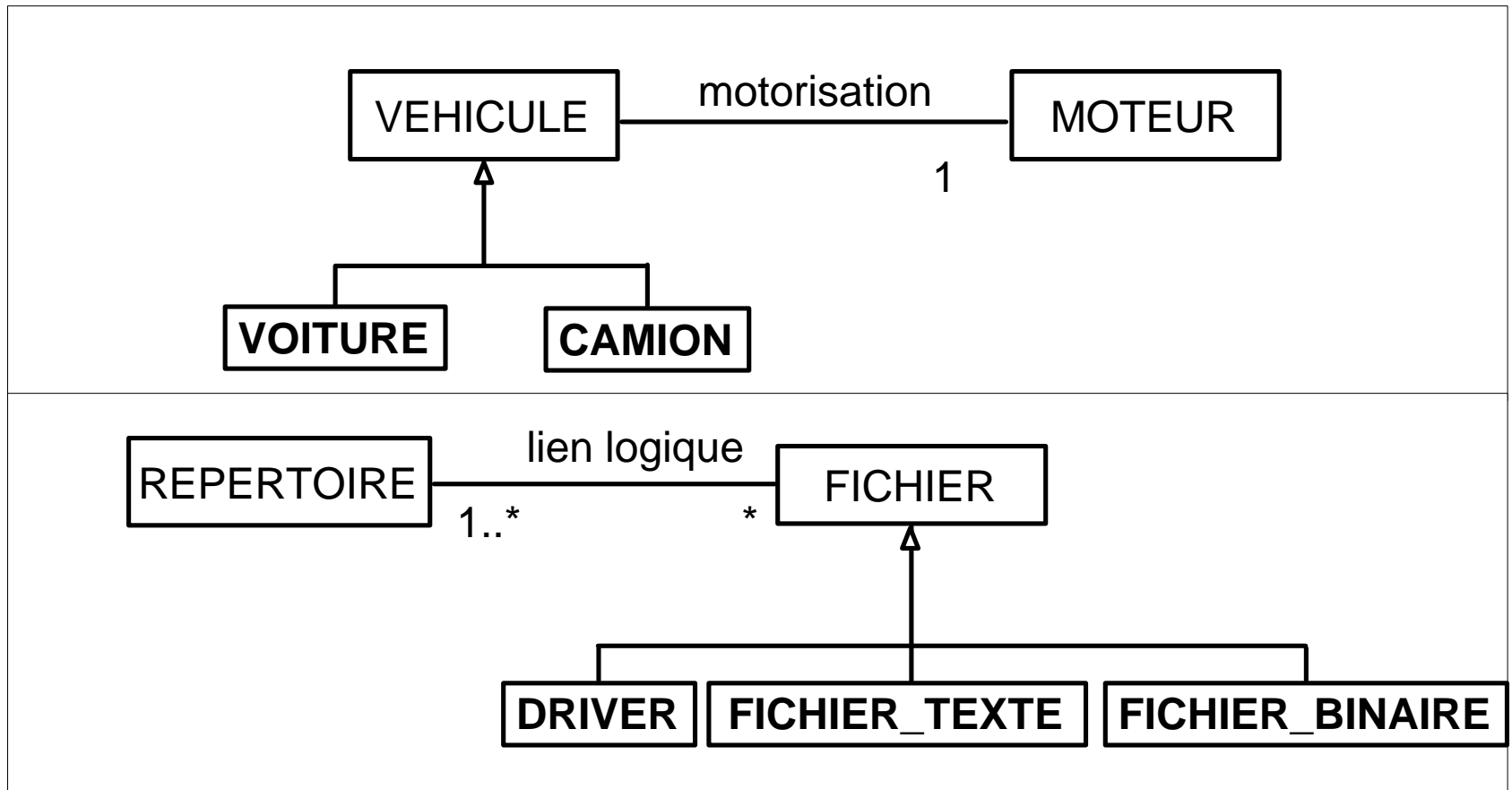


Interfaces et « lollipop »



Héritage des relations

- Les relations sont héritées par les sous classes :



Héritage des contrats

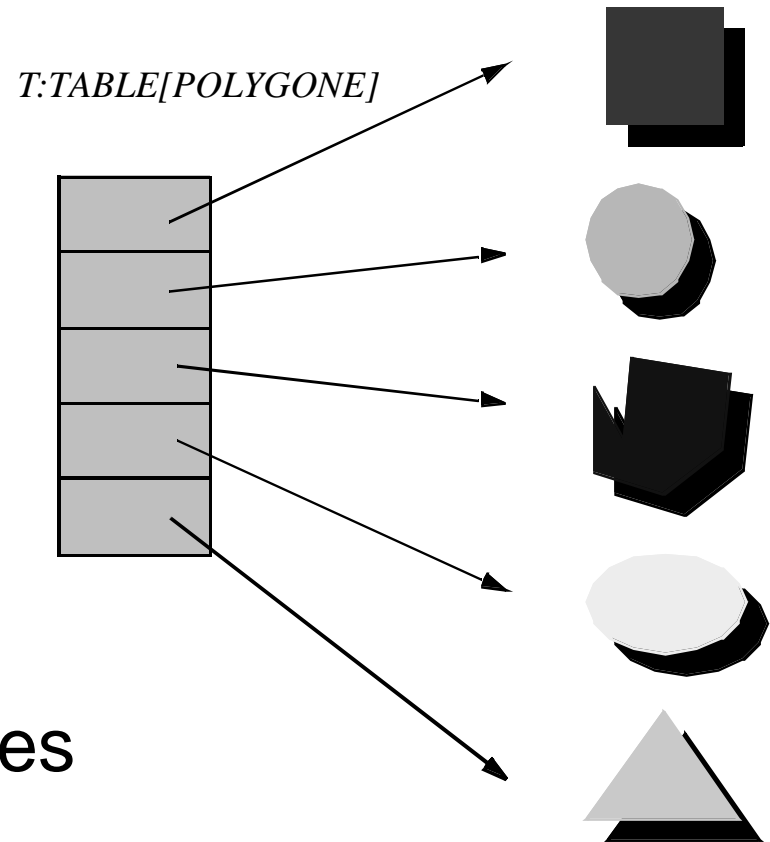
- LSP (Liskov Substituability Principle)
- Analogy with subcontracting
 - a routine subcontracts its implementation to its redefined version
- Preconditions can only be *weakened*
 - in Eiffel: **require else**
- Postconditions can only be strengthened
 - in Eiffel: **ensure then**

Polymorphisme et liaison dynamique

- Polymorphisme : possibilité de changer de forme
 - f : FORME; c : CERCLE; k : CARRE;
 - $f := c$; $f := k$
- Liaison dynamique : l'effet de l'appel d'une opération d'un objet dépend de sa forme effective à l'exécution
 - $f.imprimer$; -- *différent selon que f est CERCLE ou CARRE*
- Espace de nommage réduit et uniforme
 - Mise en facteur des parties communes
 - Possibilité de “conteneurs” hétérogènes

Polymorphisme : exemple

- T contient des FORMES
 - en fait des instances de sous-classes de FORME
- Programmes de type :
 - $T[1] \leftarrow \text{carré}$
 $T[2] \leftarrow \text{cercle}$
...
 - Pour tout i
 $T[i].\text{imprimer}$
- Si ajouts ultérieurs:
 - e.g. triangle
- Pas de modifications globales
 - case-less programming



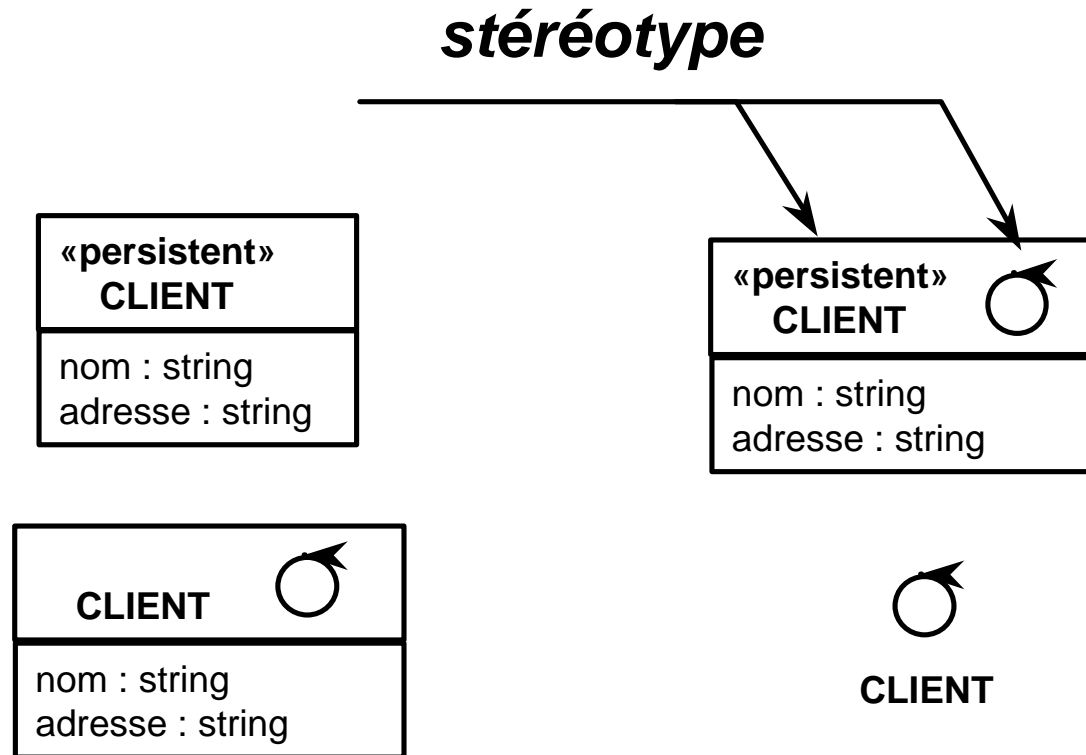
Héritage et typage

- typage statique :
 - $x.f$ légal \Leftrightarrow tout objet désigné par x “comprends” le message f
- liaison dynamique :
 - la bonne interprétation de f est choisie
- langages à objets :
 - Smalltalk : typage dynamique, liaison dynamique
 - C++ : typage (partiellement) statique, liaison dynamique pour les fonctions “virtuelles”
 - Eiffel, Java : typage statique, liaison dynamique
 - UML : au choix (!)

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 ...

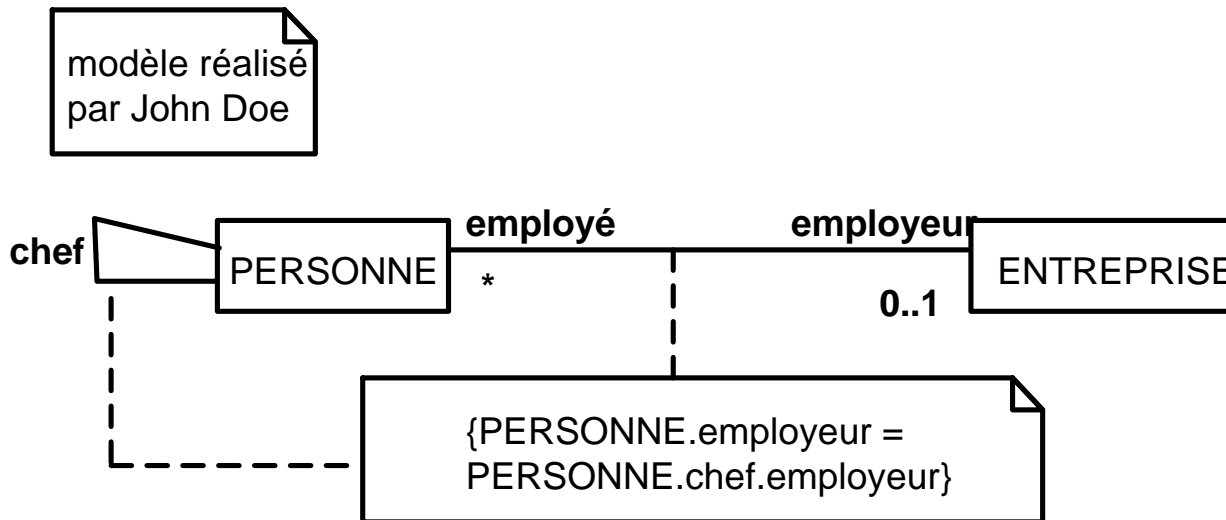
Notations pour les stéréotypes



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



Modélisation UML

- Modélisation selon 4 points de vue principaux :
 - Aspects statiques du système (*le QUI?*)
 - » Description des objets et de leurs relations
 - Modularité, contrats, relations, généricité, héritage
 - » Structuration en paquetages
 - Vision utilisateur du système (*le QUOI?*)
 - » Cas d'utilisation
 - Aspects dynamiques du système (*le QUAND?*)
 - » Diagramme de séquences (scénarios)
 - » Diagramme de collaborations (entre objets)
 - » Diagramme d'états-transitions (Harel)
 - » Diagramme d'activités
 - Vision implantation (*le OÙ?*)
 - » Diagramme de composants et de déploiement

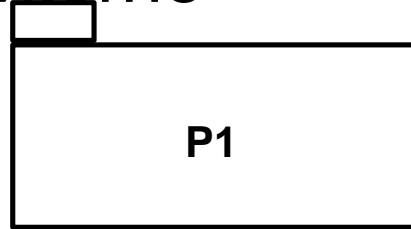


Notion de package

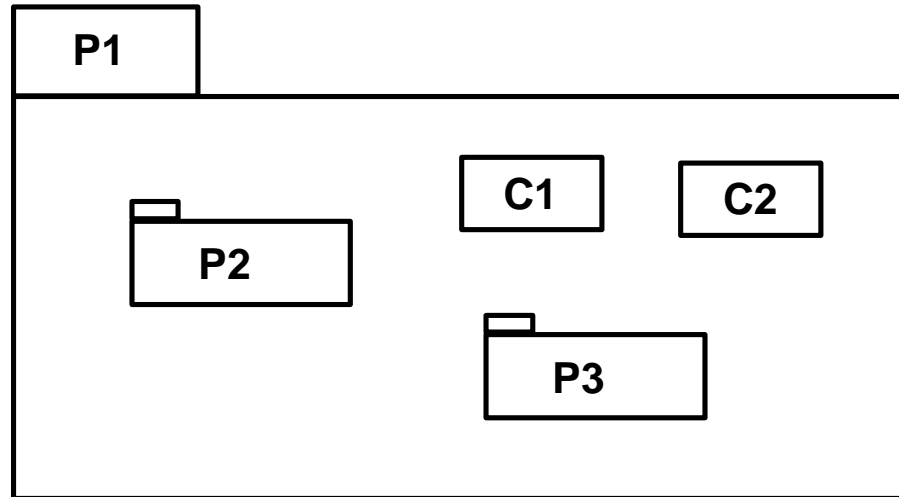
- Élément structurant les classes
 - Modularisation à l'échelle supérieure
 - Un package partitionne l'application :
 - » Il référence ou se compose des classes de l'application
 - » Il référence ou se compose d'autres packages
 - Un package régleme la visibilité des classes et des packages qu'il référence 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

Représentation d'un package

- Vue graphique externe

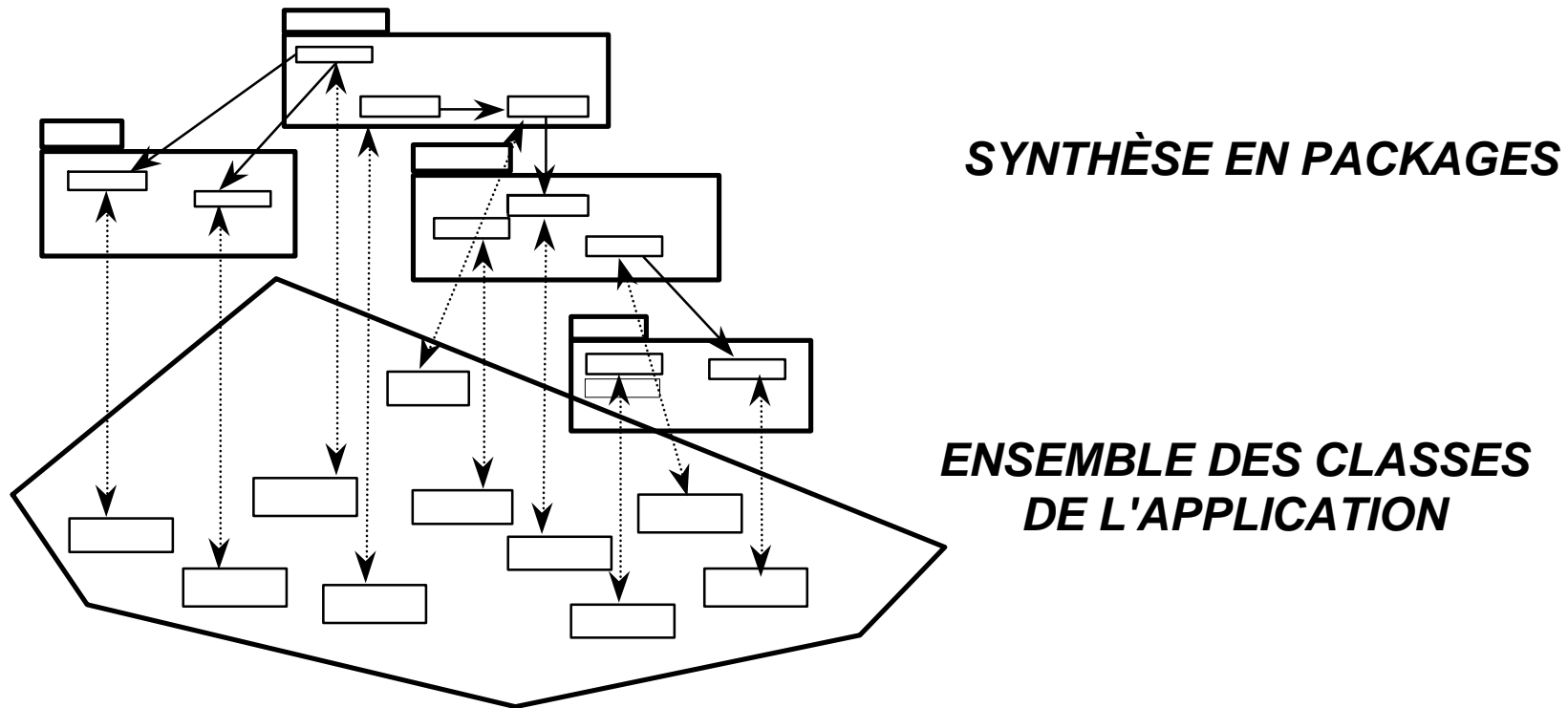


- Vue graphique externe et interne



Partitionnement d'une application

- Définition de vues partielles d'une application



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

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
{private }**

Package::Classe

CLASSE D'INTERFACE

CLASSE DE CORPS

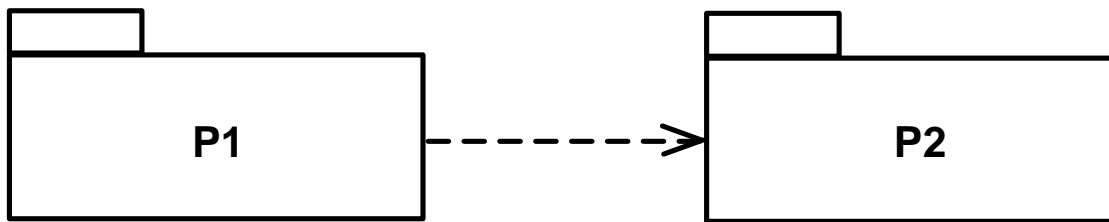
CLASSE EXTERNE

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

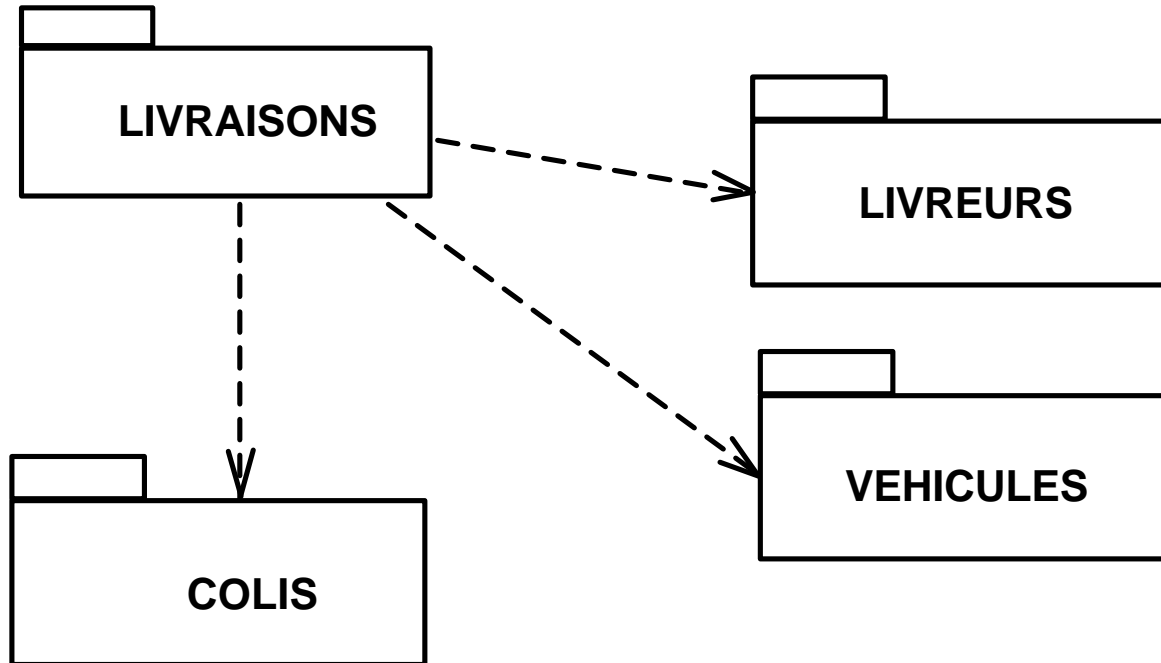
■ Représentation graphique



Vue externe du package P1

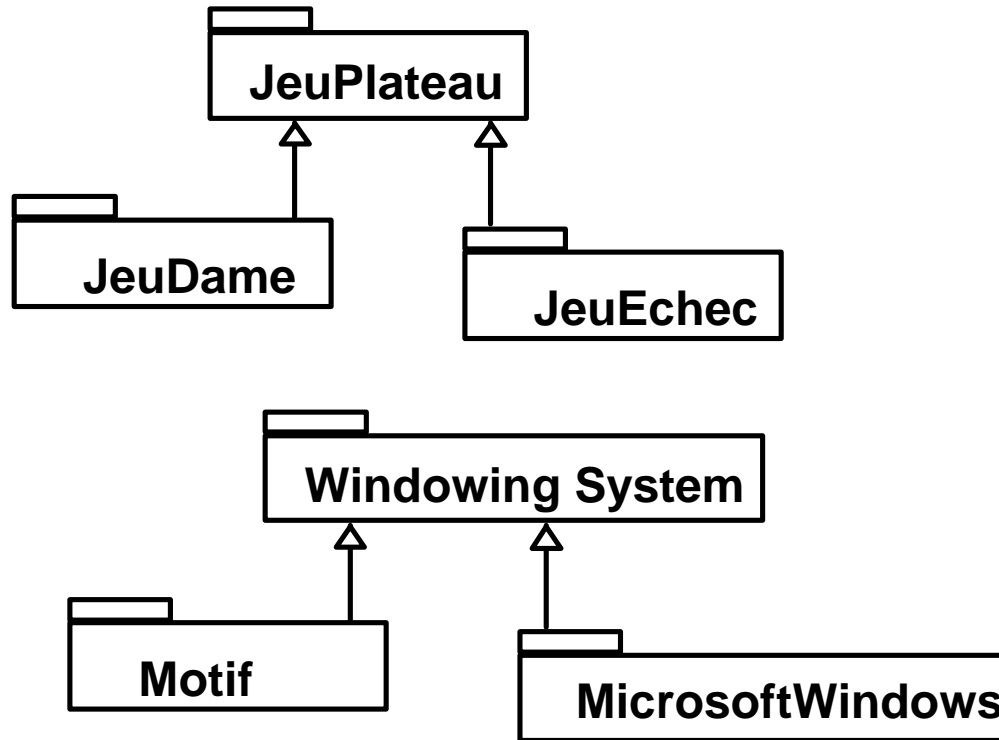
Utilisation entre packages

- Exemple (vue externe du package livraisons)



Héritage entre packages

■ Exemples



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

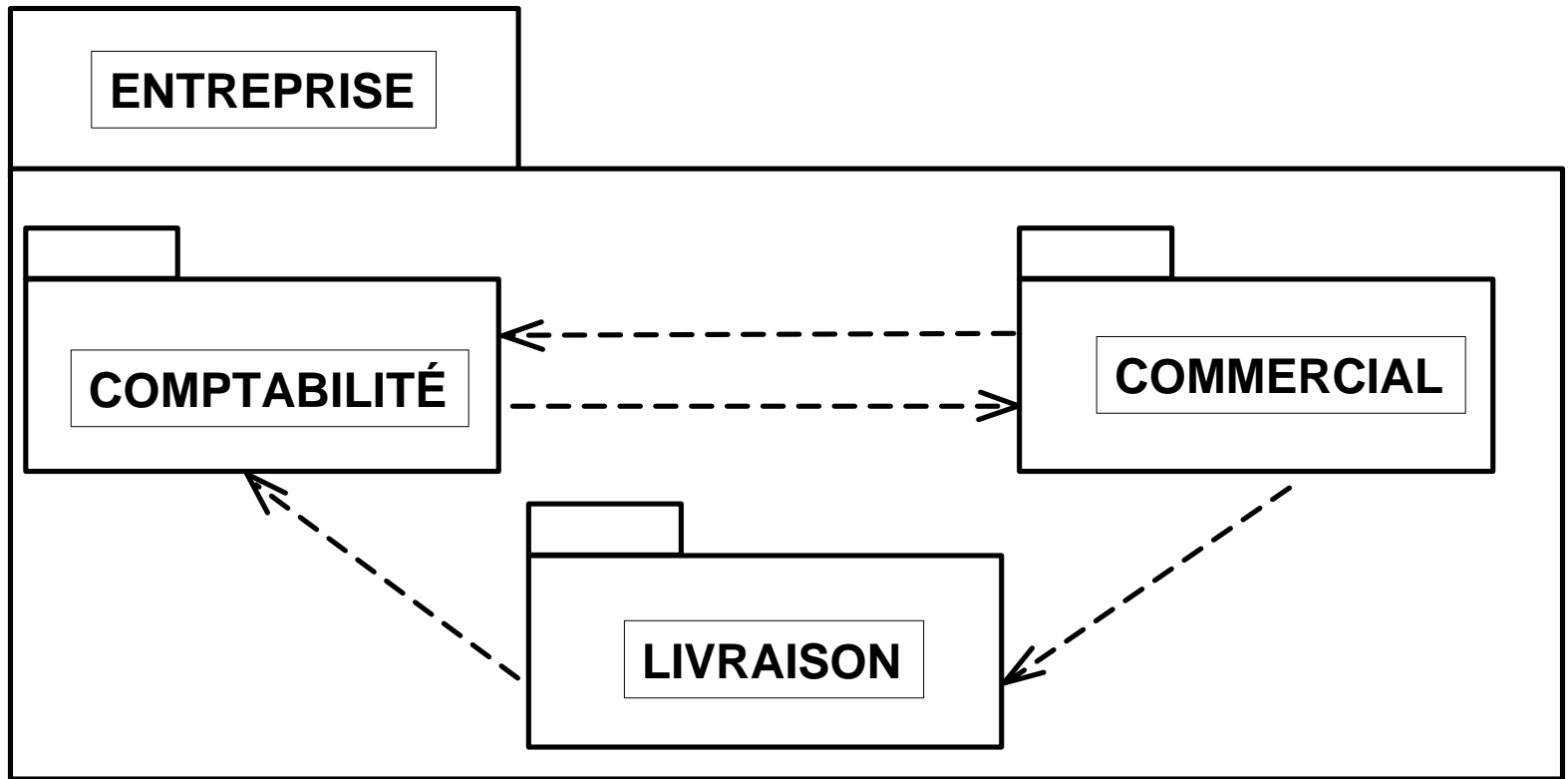
Structuration par packages (vs) décomposition hiérarchique

- ◆ **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

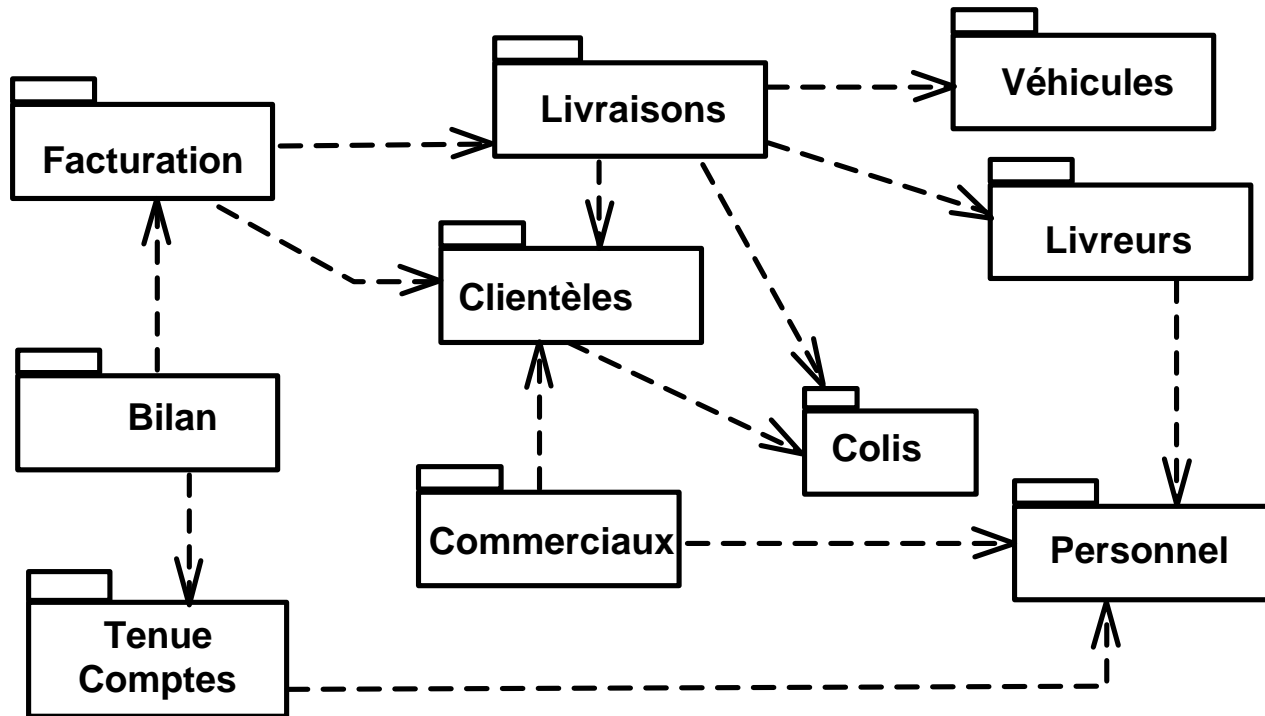
Exemple : Package entreprise

■ Exemple de composition



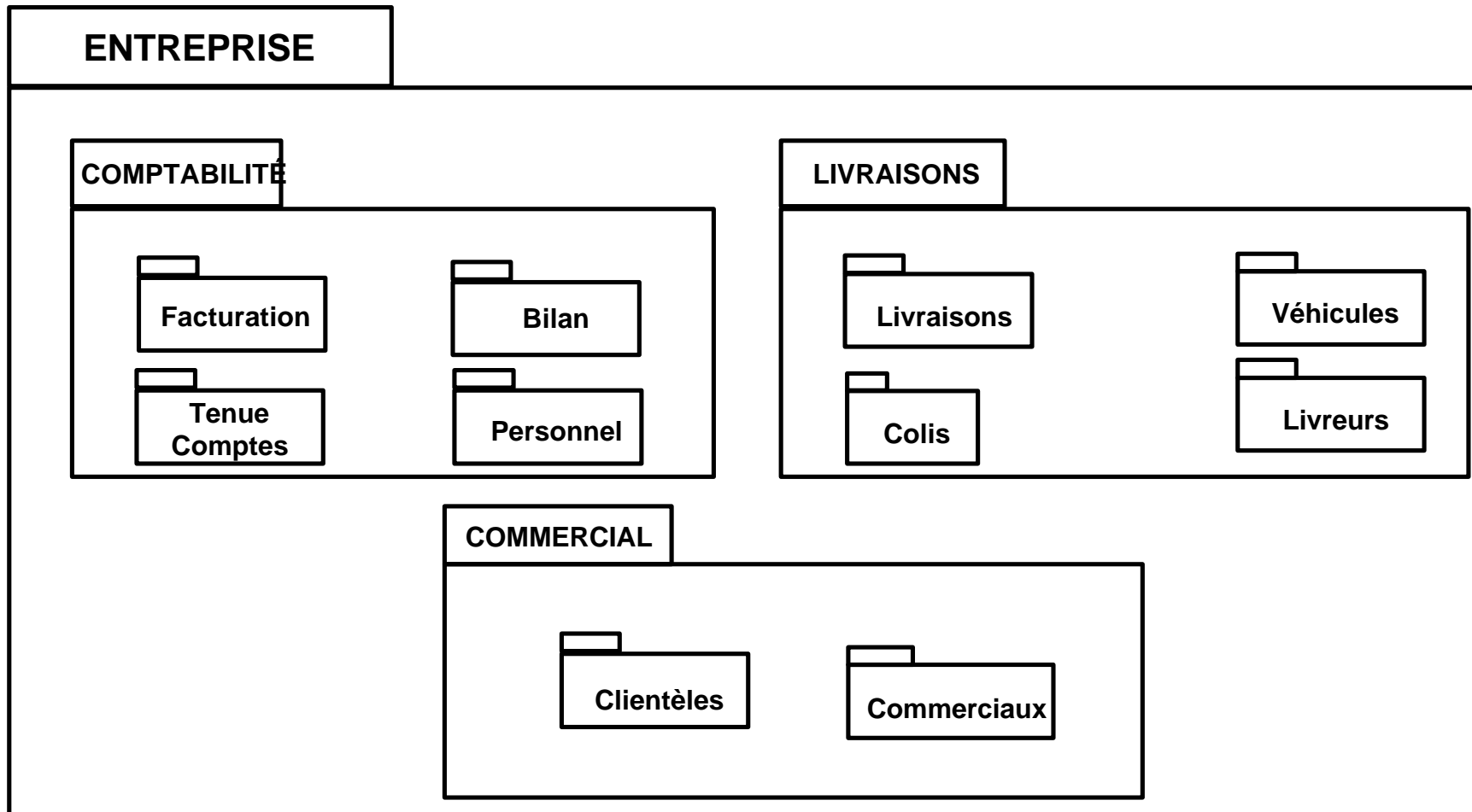
Exemple : Package entreprise

- Ensemble des packages terminaux de l'application




Exemple : Package entreprise

■ Composition des packages en sous-packages



Modélisation UML

■ Modélisation selon 4 points de vue principaux :

- Aspects statiques du système (*le QUI?*)
 - » Description des objets et de leurs relations
 - Modularité, contrats, relations, généricité, héritage
 - » Structuration en paquetages
-  Vision utilisateur du système (*le QUOI?*)
 - » Cas d'utilisation
- Aspects dynamiques du système (*le QUAND?*)
 - » Diagramme de séquences (scénarios)
 - » Diagramme de collaborations (entre objets)
 - » Diagramme d'états-transitions (Harel)
 - » Diagramme d'activités
- Vision implantation (*le OÙ?*)
 - » Diagramme de composants et de déploiement

Expression des besoins et OOAD

- Sujet longtemps négligé (e.g. OMT)
 - dérive de certains projets vers le « conceptuel »
- Question de l'expression des besoins pourtant fondamentale
 - Et souvent pas si facile (*cible mouvante*)
 - » cf. *syndrome de la balance*
- 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

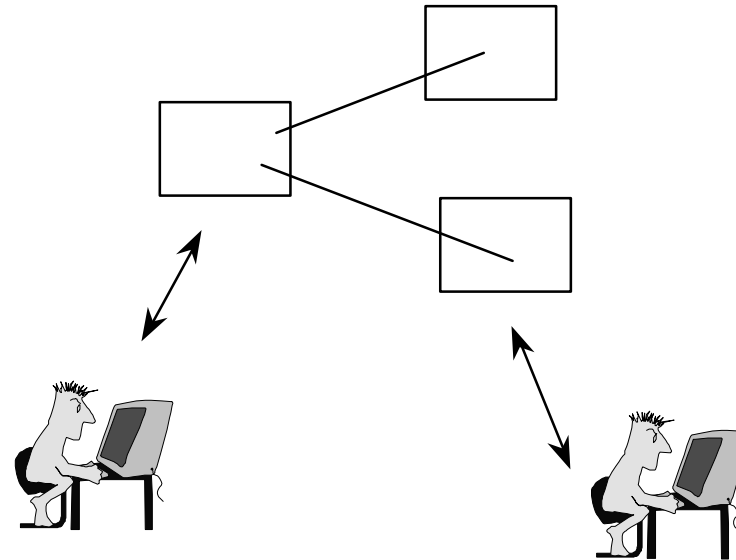
☆ Se comprendre



🕒 Représenter le système

🕒 Exprimer le service rendu

🕒 Décrire la manière dont le système est perçu

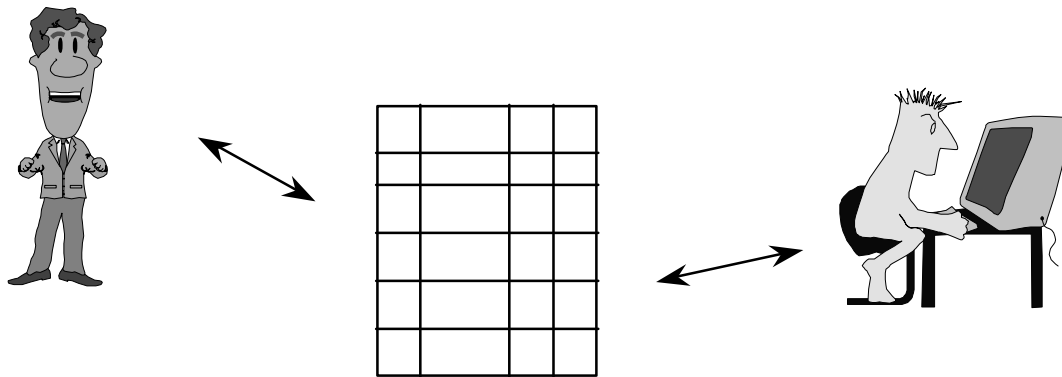


Les moyens

- On imagine le système « fini »
- On montre comment on interagit avec lui
 - Les acteurs UML
 - Les *use-cases* UML

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.



Exemple de dictionnaire

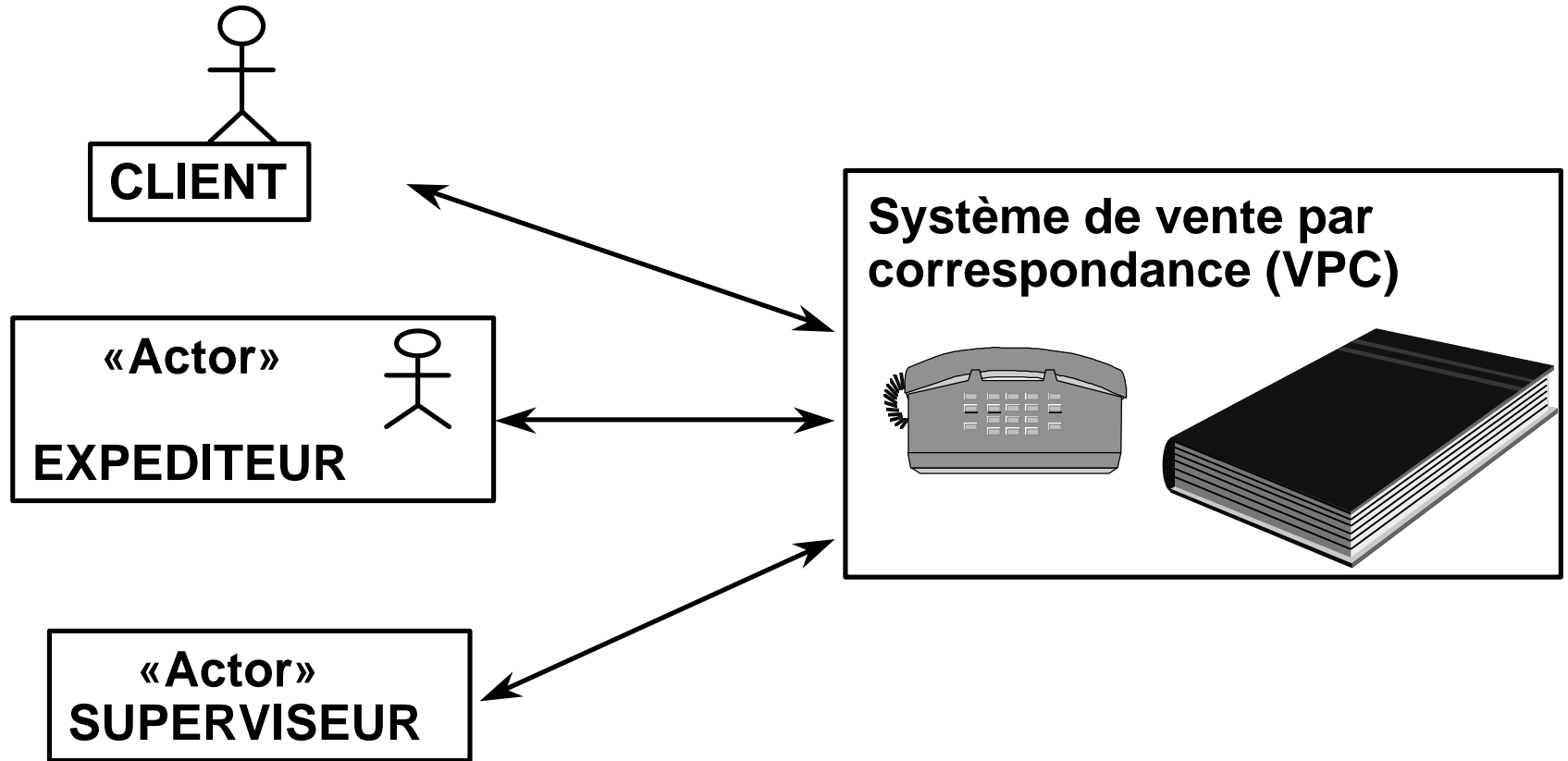
– Dictionnaire d'un simulateur de vol

Notion	Définition	Traduit en ...	Nom informatique
Pilotage	Action de piloter un avion en enchaînant des manoeuvres élémentaires	Package	Pilotage
Instrument	Organe d'interaction entre le pilote et l'avion ou entre l'avion et le pilote	Classe abstraite	Instrument
Manette des gaz	Instrument qui permet d'agir sur la quantité de carburant injectée dans le moteur	Classe	Manette_gaz
Action	Définition	Traduit en ...	Nom informatique
Mettre les gaz à fond	Action qui permet d'injecter le maximum de carburant pour atteindre la vitesse maximale	Opération	Mettre_a_fond

Acteurs

- Entité externe au système et amenée à interagir avec lui. Un acteur «joue un rôle» vis-a-vis du système
- Un acteur est une classe
- Un acteur peut représenter un être humain, un autre système, ...
- L'identification des acteurs permet de délimiter le système

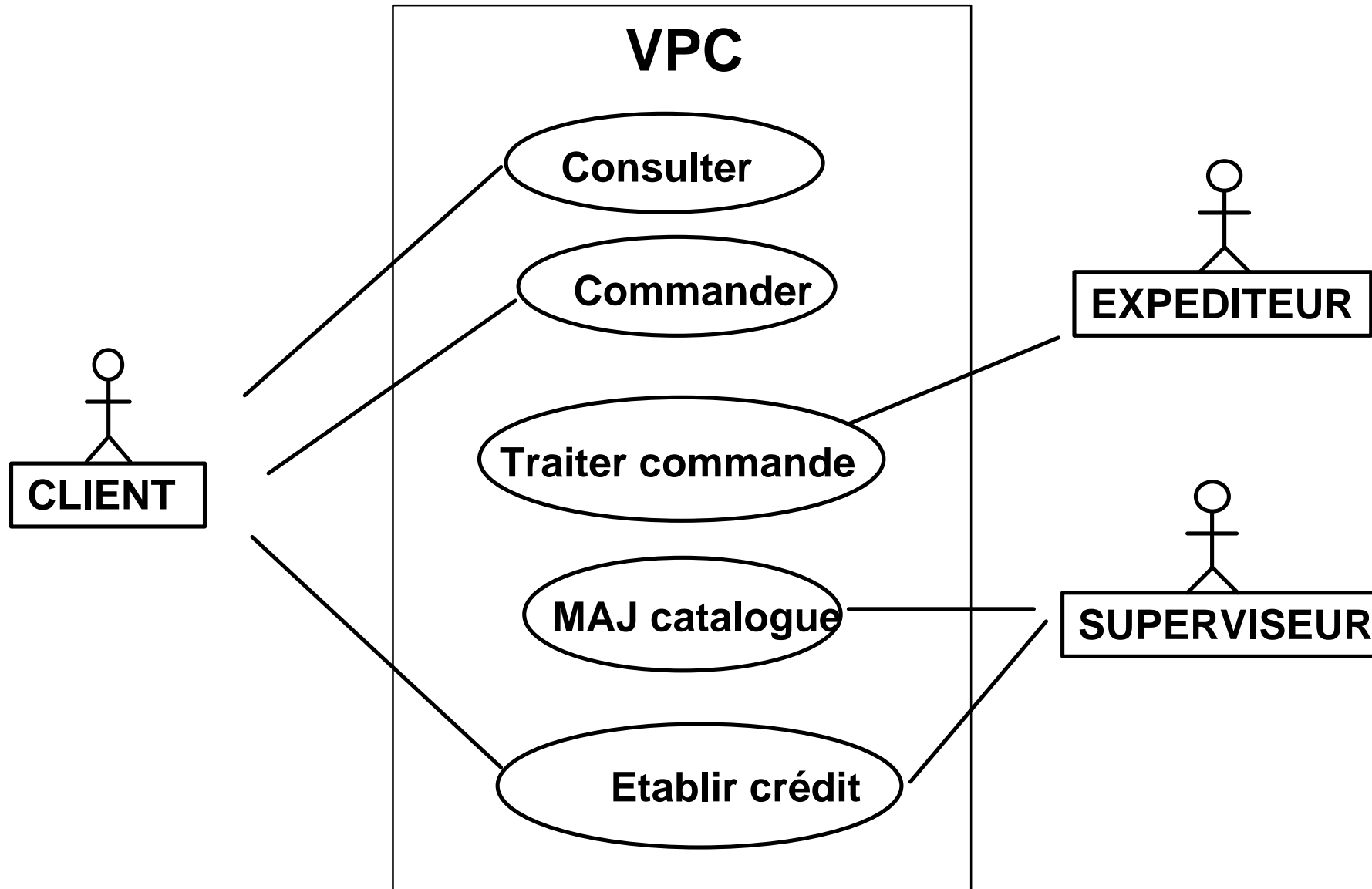
Acteurs : notations



Les cas d'utilisation (use-cases)

- Un cas d'utilisation est une manière particulière d'utiliser le système
 - séquence d'interactions entre le système et un ou plusieurs acteurs
 - Ils s'expriment par des diagrammes de séquences
- La compilation des cas d'utilisation décrit de manière informelle le service rendu par le système
 - fournissent une expression "fonctionnelle" du besoin
 - peuvent piloter la progression d'un cycle en spirale
- Les cas d'utilisation sont nommes en utilisant la terminologie décrite dans le dictionnaire

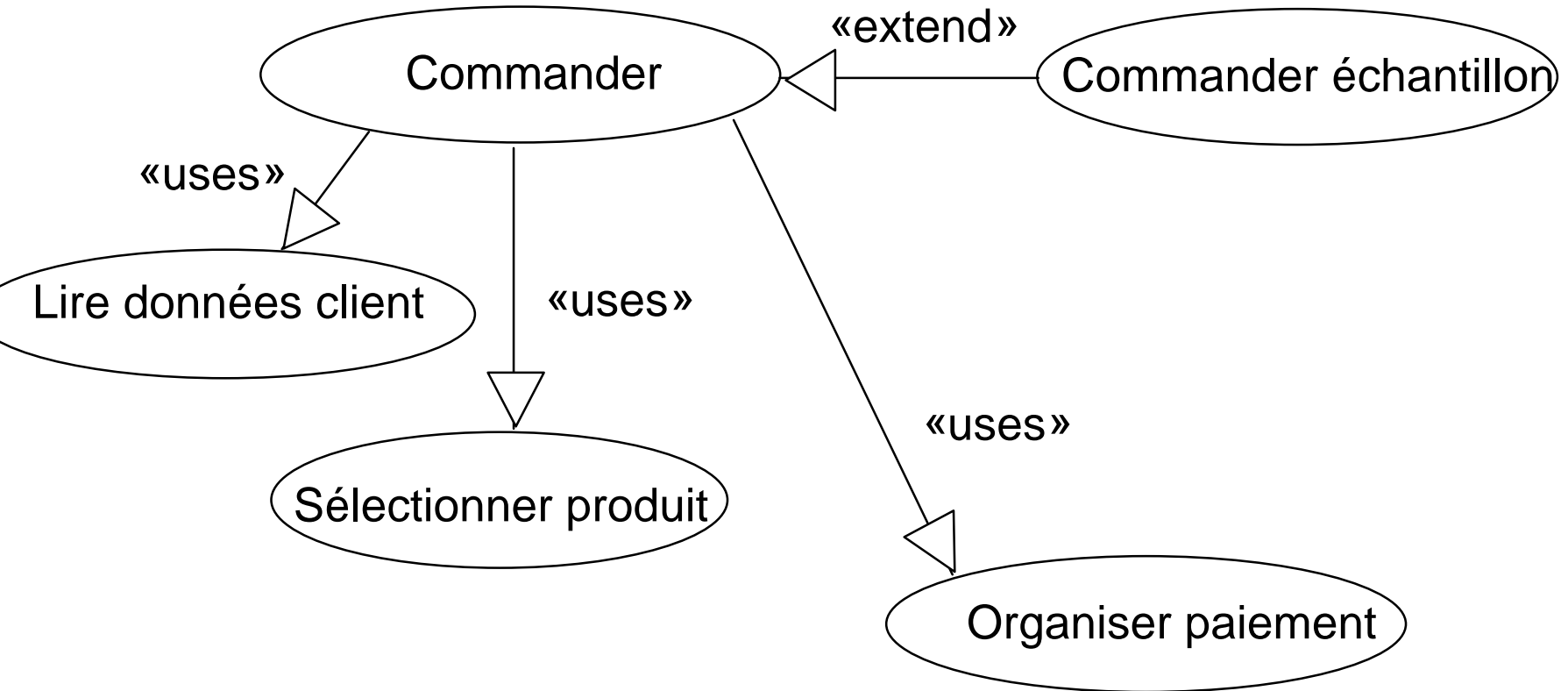
Cas d'utilisation : exemple et notation



Relations sur les *use-cases*

- Communication
 - Lien entre le use case et l'acteur.
- Utilisation («*uses*»)
 - Utilisation d'autres use-cases pour en préciser la définition
- Extension («*extends*»)
 - Un use-case étendu est une spécialisation du use-case père

Relations sur les *use-cases* : notation



Exprimer le service rendu

- Besoins fondamentaux : manières d'utiliser le système
 - Représentation globale par cas d'utilisation
 - \forall taille du système, seulement de 3 à 10 Use Cases
- Besoins opérationnels : interactions avec le système
 - Représentation détaillée par raffinement des cas d'utilisation
 - Début de décomposition fonctionnelle : ne pas aller trop loin

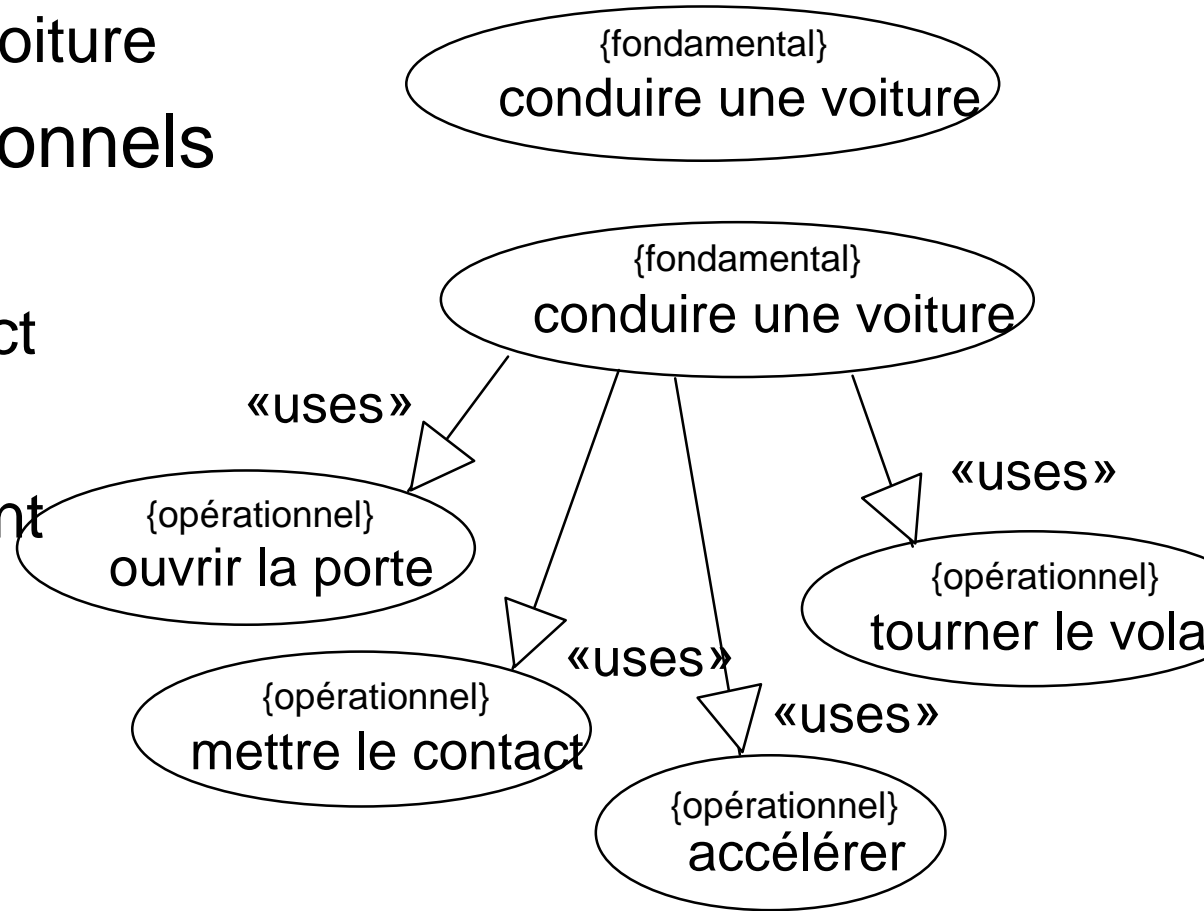
Besoins fondamentaux et opérationnels

■ Besoin fondamental :

- Conduire une voiture

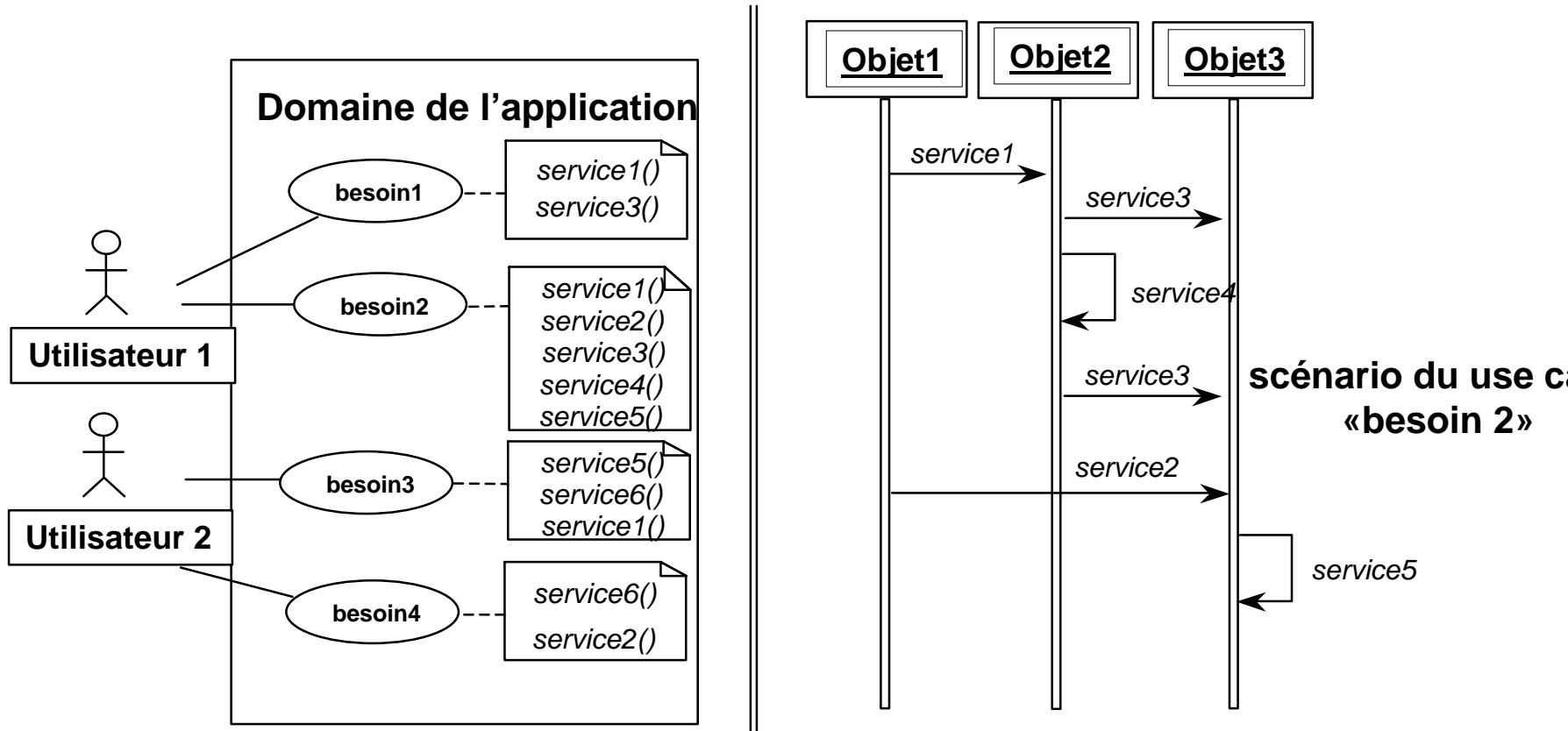
■ Besoins opérationnels

- Ouvrir la porte
- Mettre le contact
- Accélérer
- Tourner le volant
- ...




Utiles pour l'établissement de scénarios

■ Modélisation d'exemples issus des use-cases



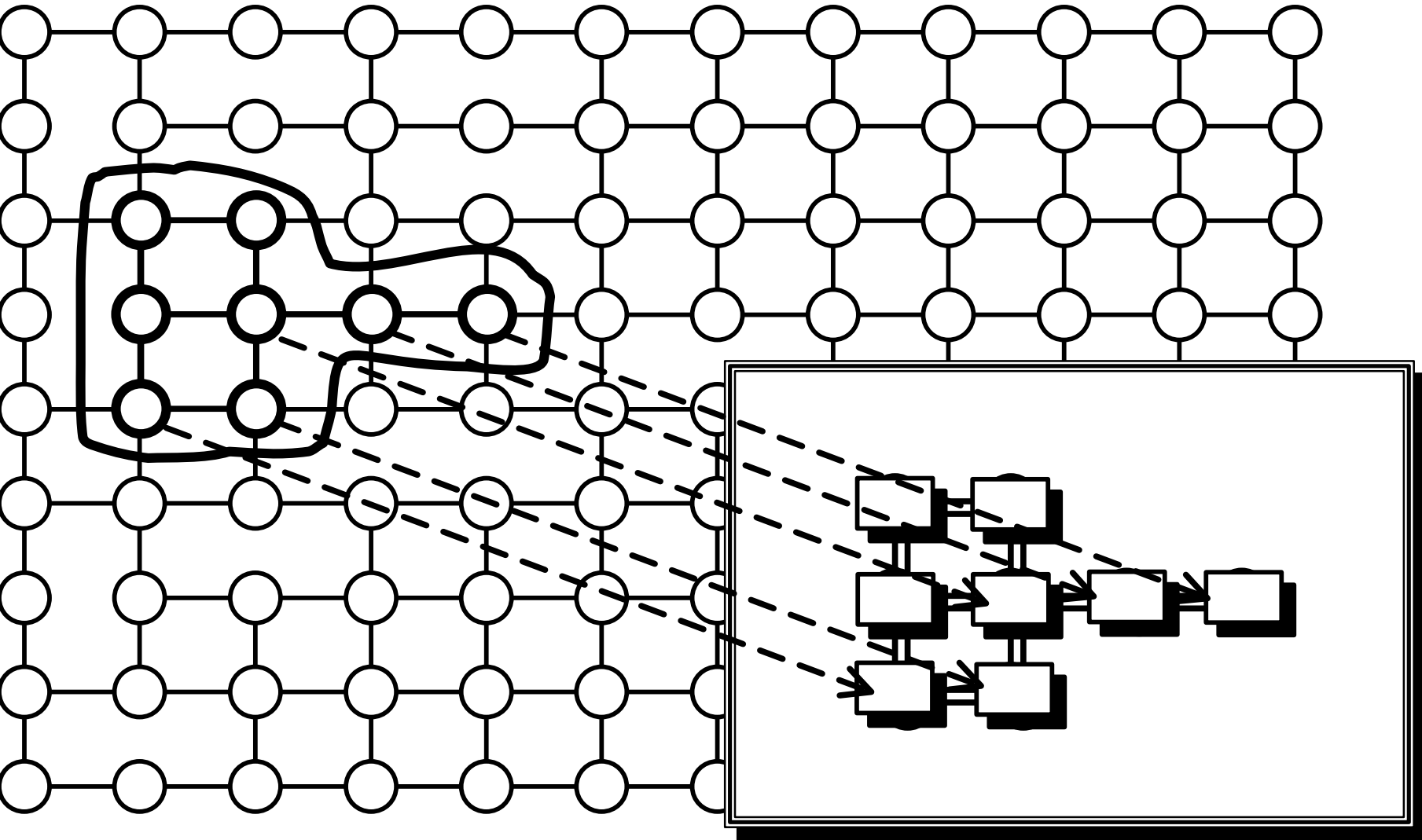
Modélisation UML

- Modélisation selon 4 points de vue principaux :
 - Aspects statiques du système (*le QUI?*)
 - » Description des objets et de leurs relations
 - Modularité, contrats, relations, généricité, héritage
 - » Structuration en paquetages
 - Vision utilisateur du système (*le QUOI?*)
 - » Cas d'utilisation
 -  – Aspects dynamiques du système (*le QUAND?*)
 - » Diagramme de séquences (scénarios)
 - » Diagramme de collaborations (entre objets)
 - » Diagramme d'états-transitions (Harel)
 - » Diagramme d'activités
 - Vision implantation (*le OU?*)
 - » Diagramme de composants et de déploiement

Aspects dynamiques du système

- Jusqu'ici, système décrit *statiquement*:
 - Décrivent 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 d'abord sur les **collaborations** entre objets. Rappel :
 - objets : simples
 - gestion complexité : par collaborations entre objets simples

Collaborations (au niveau instances)



Modélisation UML

- Modélisation selon 4 points de vue principaux :
 - Aspects statiques du système (*le QUI?*)
 - » Description des objets et de leurs relations
 - Modularité, contrats, relations, généricité, héritage
 - » Structuration en paquetages
 - Vision utilisateur du système (*le QUOI?*)
 - » Cas d'utilisation
 - Aspects dynamiques du système (*le QUAND?*)
 - » Diagramme de séquences (scénarios)
 - » Diagramme de collaborations (entre objets)
 - » Diagramme d'états-transitions (Harel)
 - » Diagramme d'activités
 - Vision implantation (*le OU?*)
 - » Diagramme de composants et de déploiement



Diagrammes de séquences (scénarios)

■ Dérivés des scénarios de OMT :

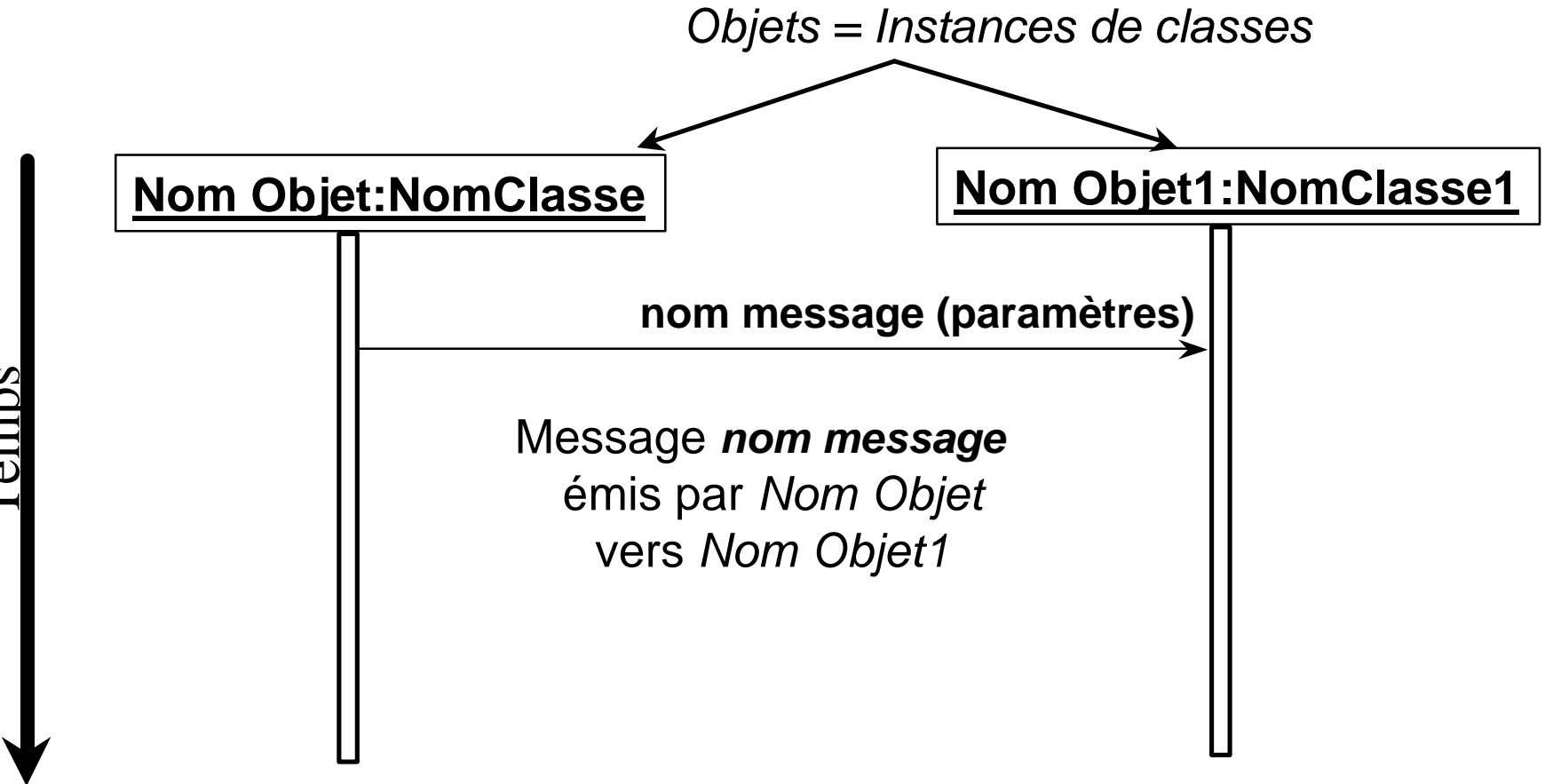
- Montrent des exemples de coopération entre objets dans la réalisation de processus de l'application
- Illustrent la dynamique d'enchaînement des traitements à travers les messages échangés entre objets
- le temps est représenté comme une dimension explicite
» en général de haut en bas

■ 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)

Syntaxe graphique

■ Objets et messages

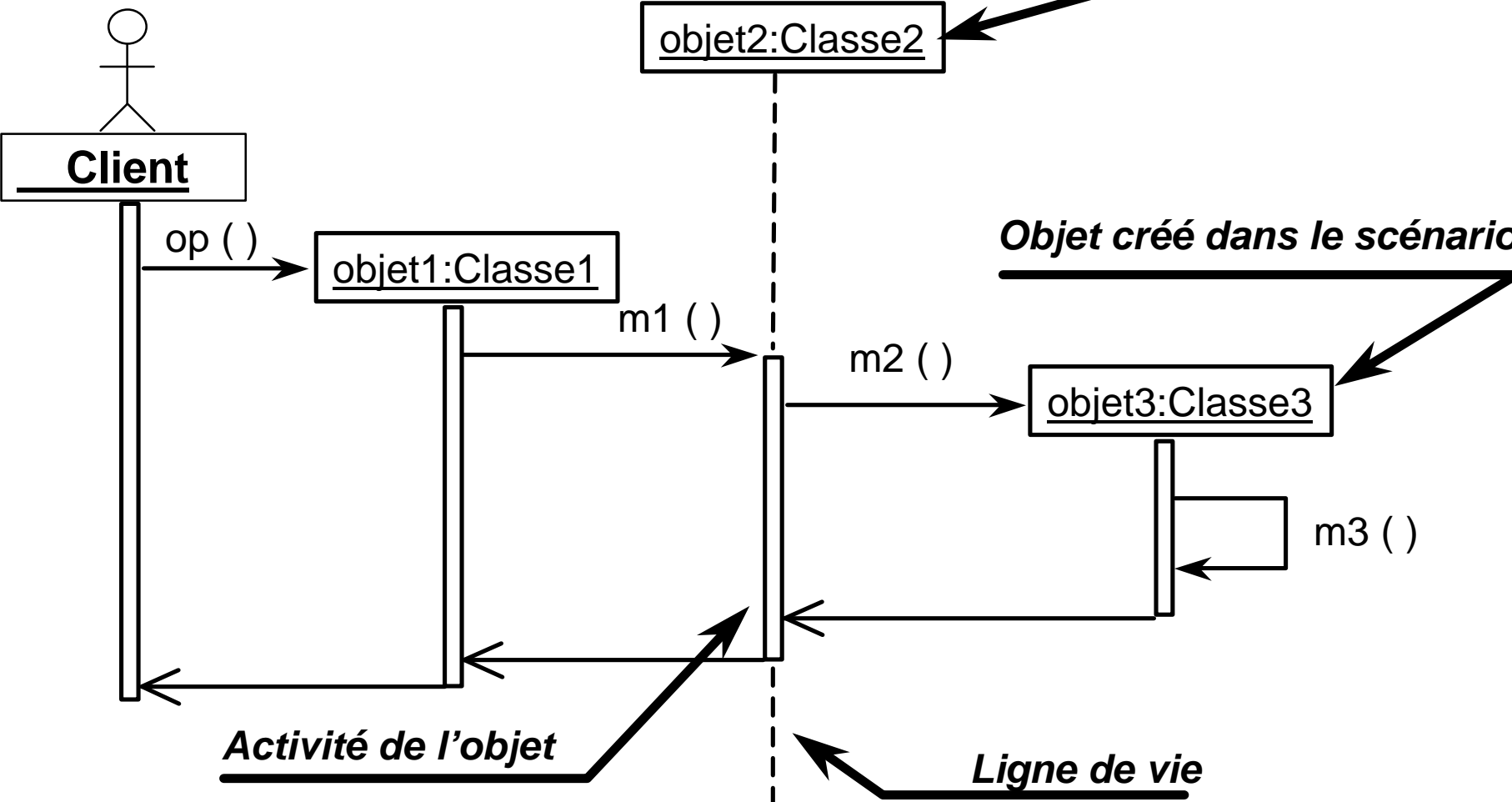


Ligne de vie et activation

- 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

Notation

Objet existant avant et après l'activation du scénario

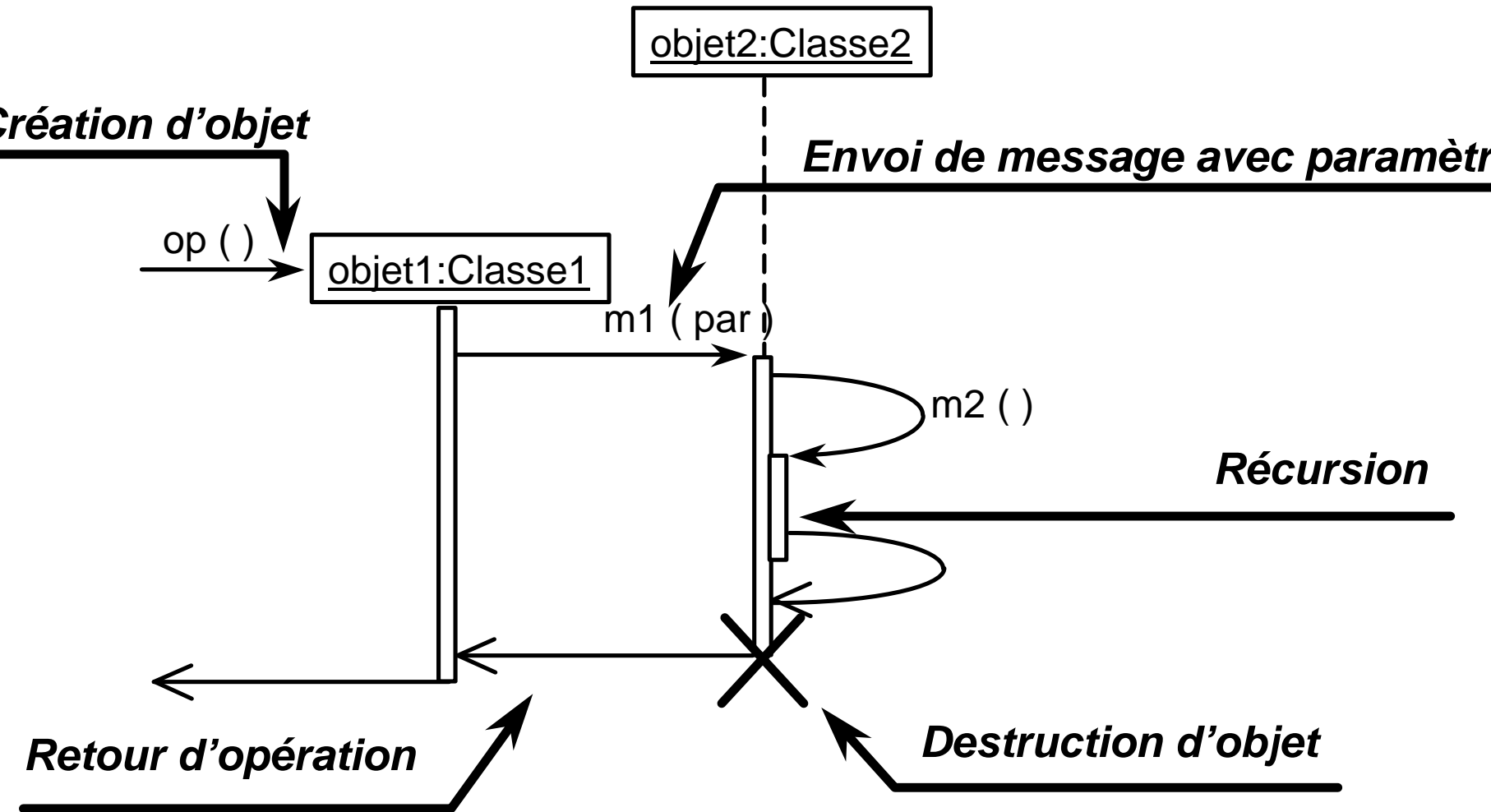


Messages

- Communication entre objets
 - Des paramètres
 - Un retour

- Cas particuliers
 - Les messages entraînant la construction d'un objet
 - La récursion
 - Les destructions d'objets

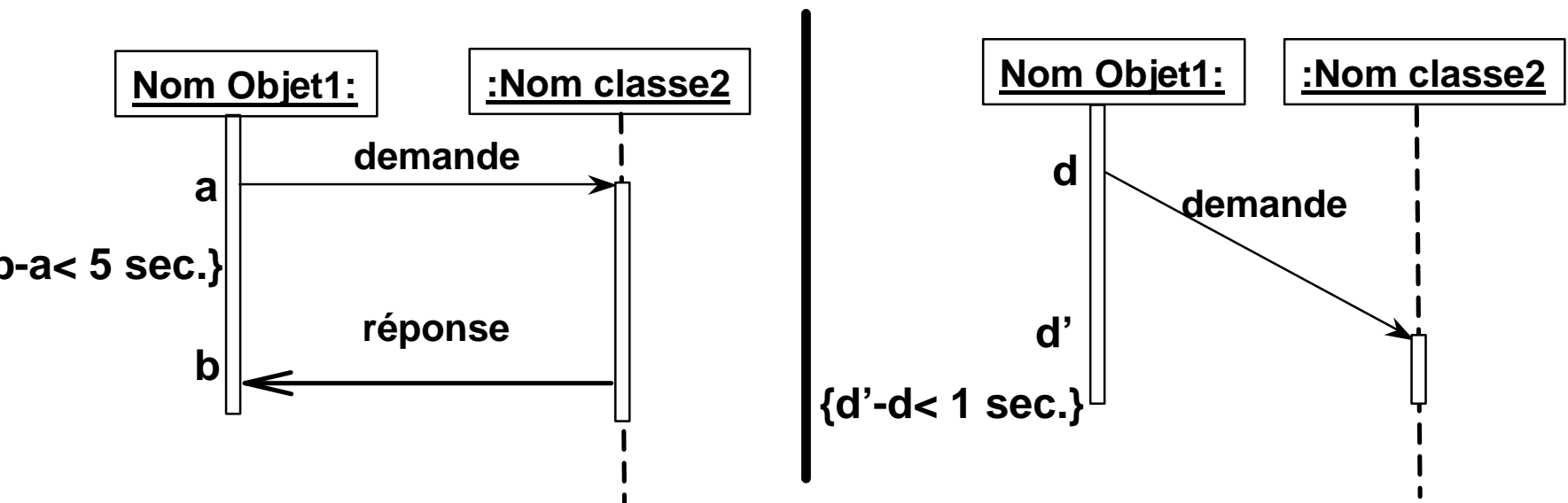
Notations



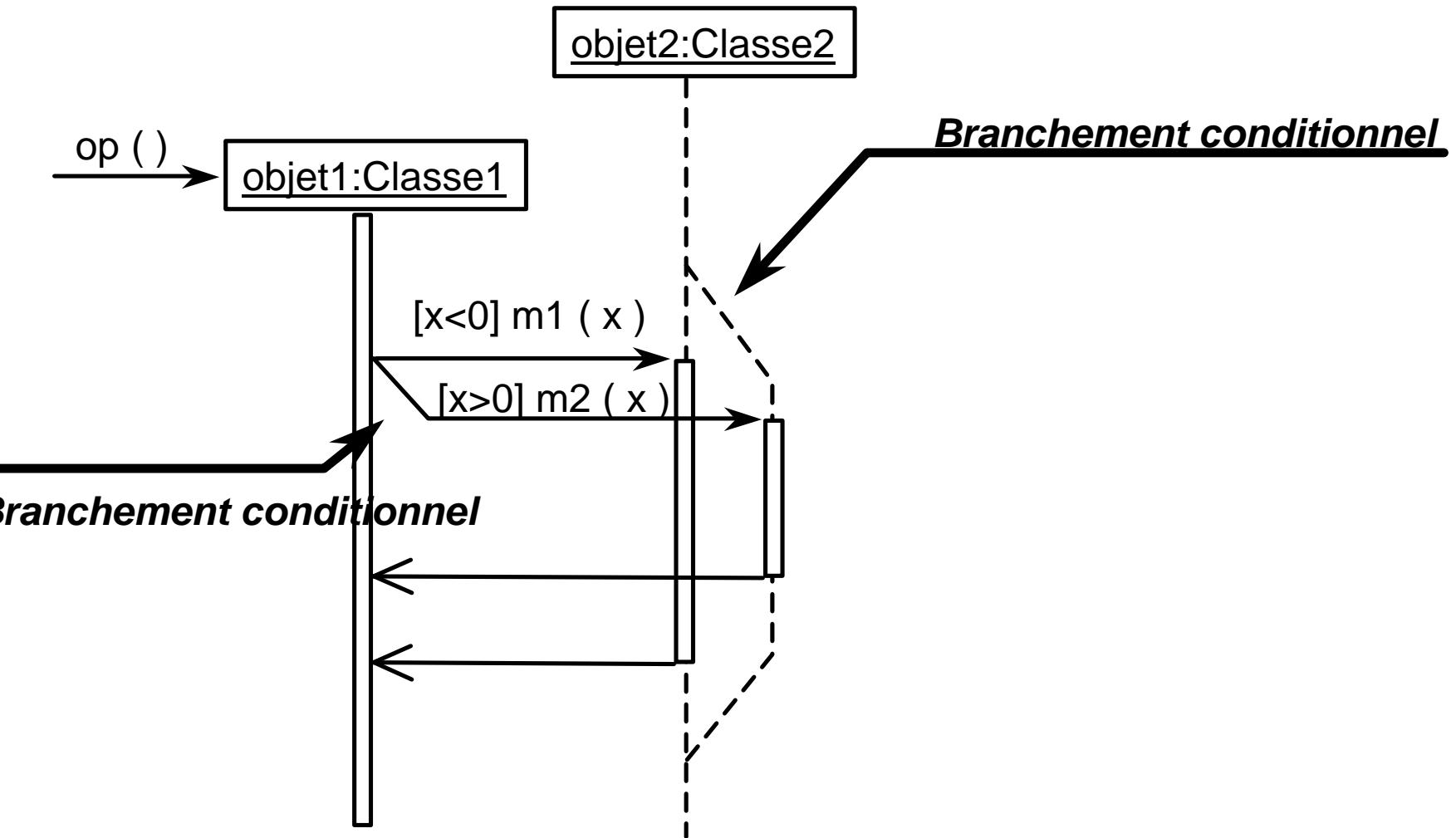
Aspects asynchrones et temps réel

■ 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



Modélisation UML

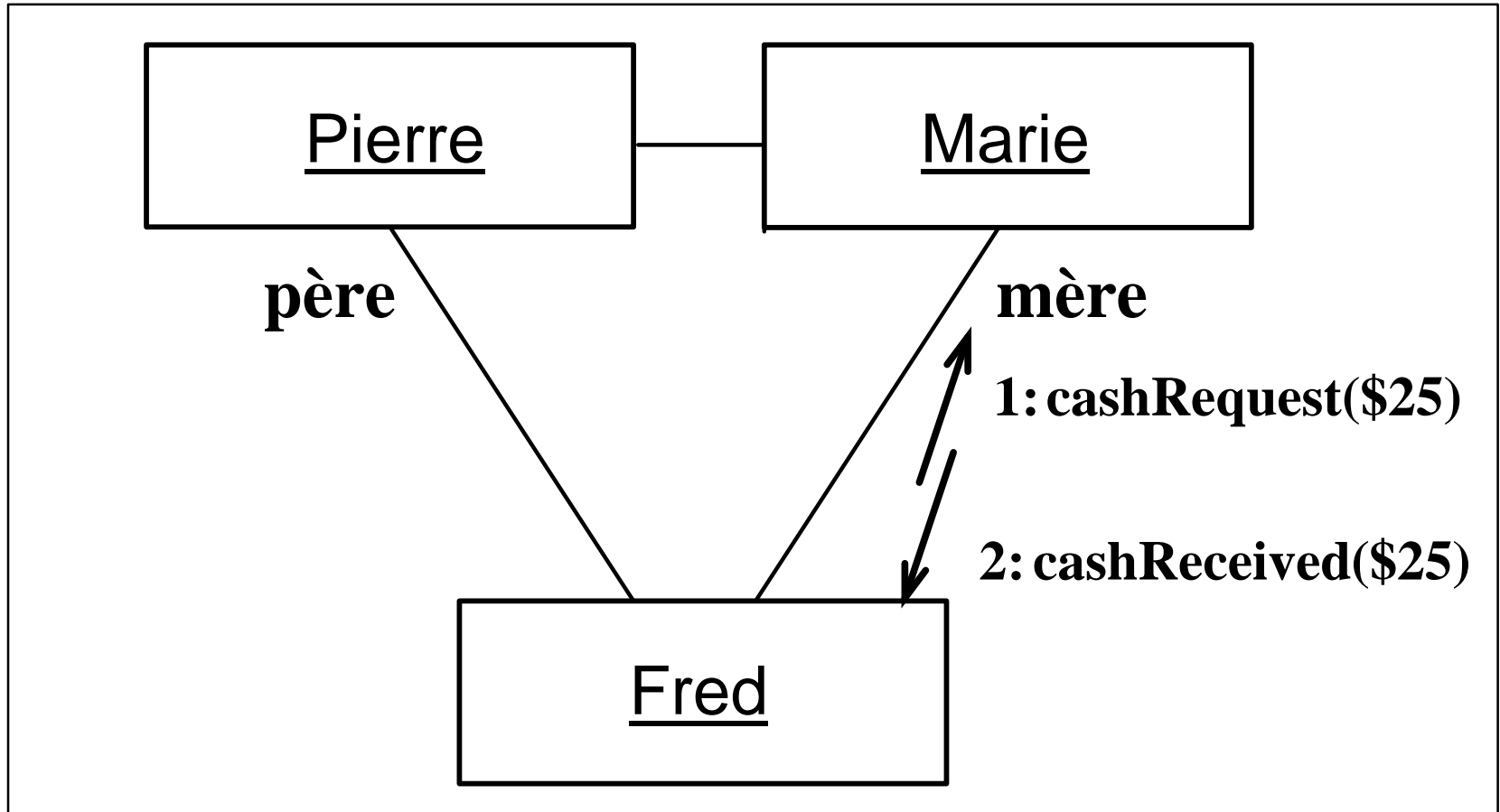
- Modélisation selon 4 points de vue principaux :
 - Aspects statiques du système (*le QUI?*)
 - » Description des objets et de leurs relations
 - Modularité, contrats, relations, généricité, héritage
 - » Structuration en paquetages
 - Vision utilisateur du système (*le QUOI?*)
 - » Cas d'utilisation
 - Aspects dynamiques du système (*le QUAND?*)
 - » Diagramme de séquences (scénarios)
 - » Diagramme de collaborations (entre objets)
 - » Diagramme d'états-transitions (Harel)
 - » Diagramme d'activités
 - Vision implantation (*le OU?*)
 - » Diagramme de composants et de déploiement



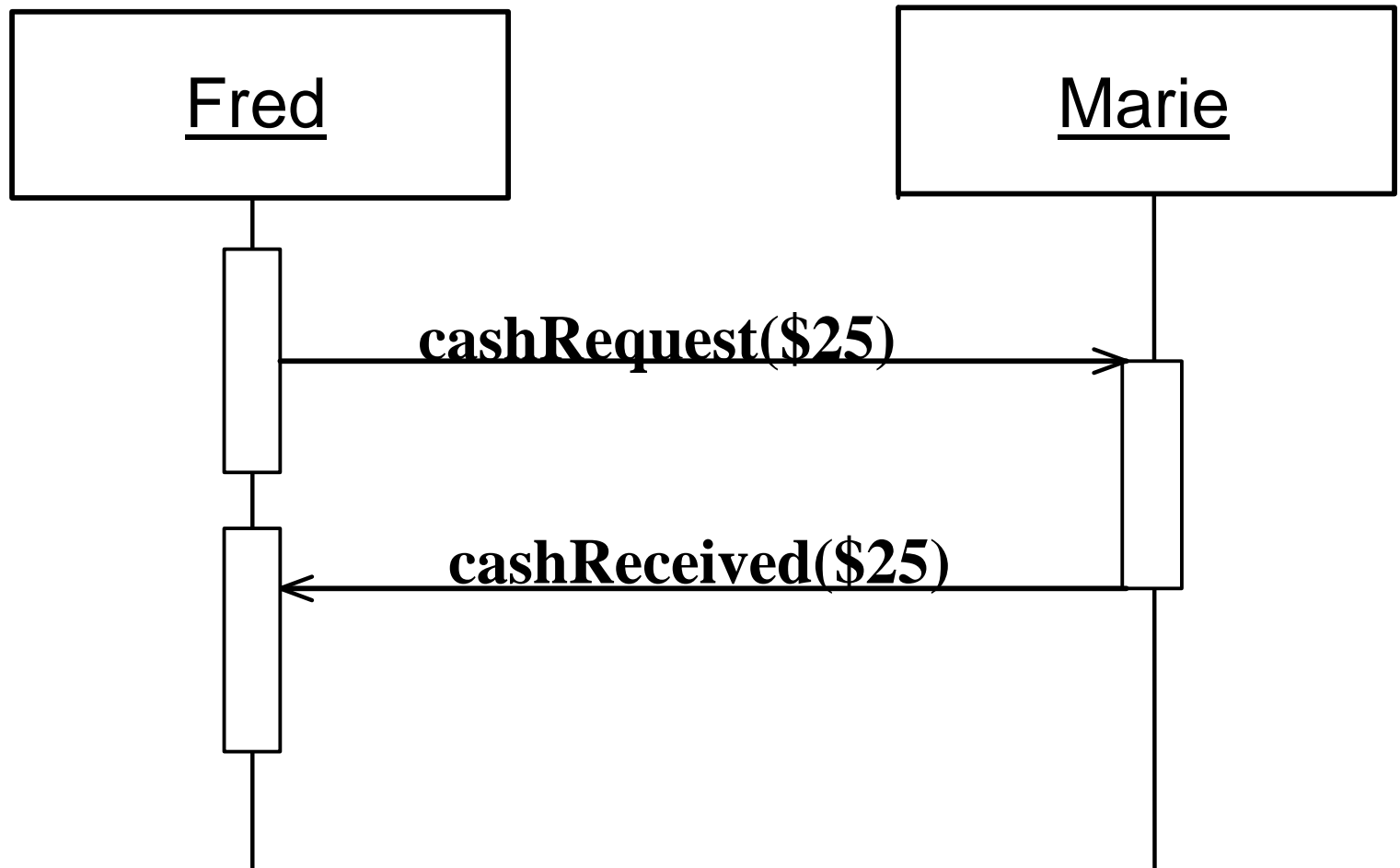
Diagrammes de collaboration

- 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

Représentation d'une collaboration (niveau instance)



Equivalent au diagramme de séquence:



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

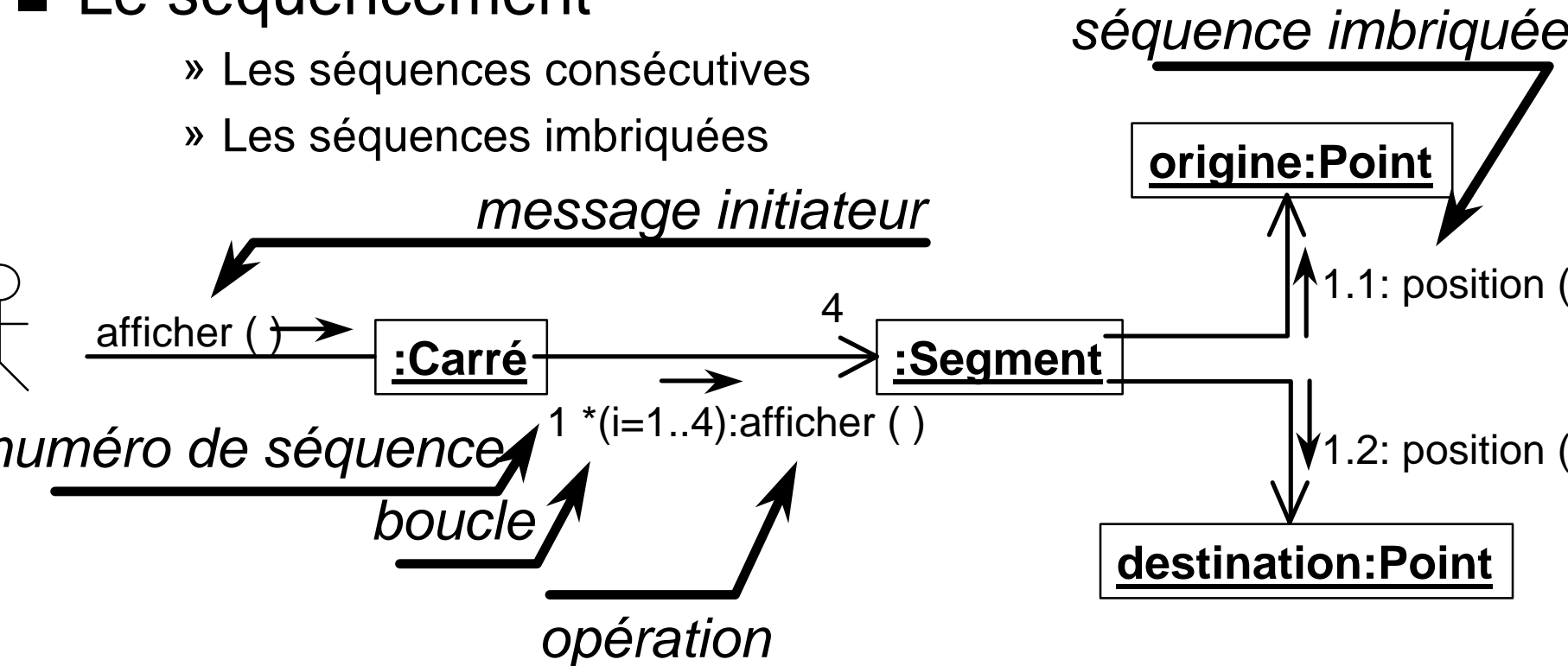
Syntaxe graphique

■ 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 ?

Collaboration : Niveau Specification

Définition du *ClassifierRole*

- A ClassifierRole is a named slot for an object participating in a Collaboration.
- Object behavior is represented by its participation in the overall behavior of the Collaboration.
- Object identity is preserved through this constraint:

"In an instance of a collaboration, each ClassifierRole maps onto at most one object."

Collaboration de Niveau Specification

un exemple simple

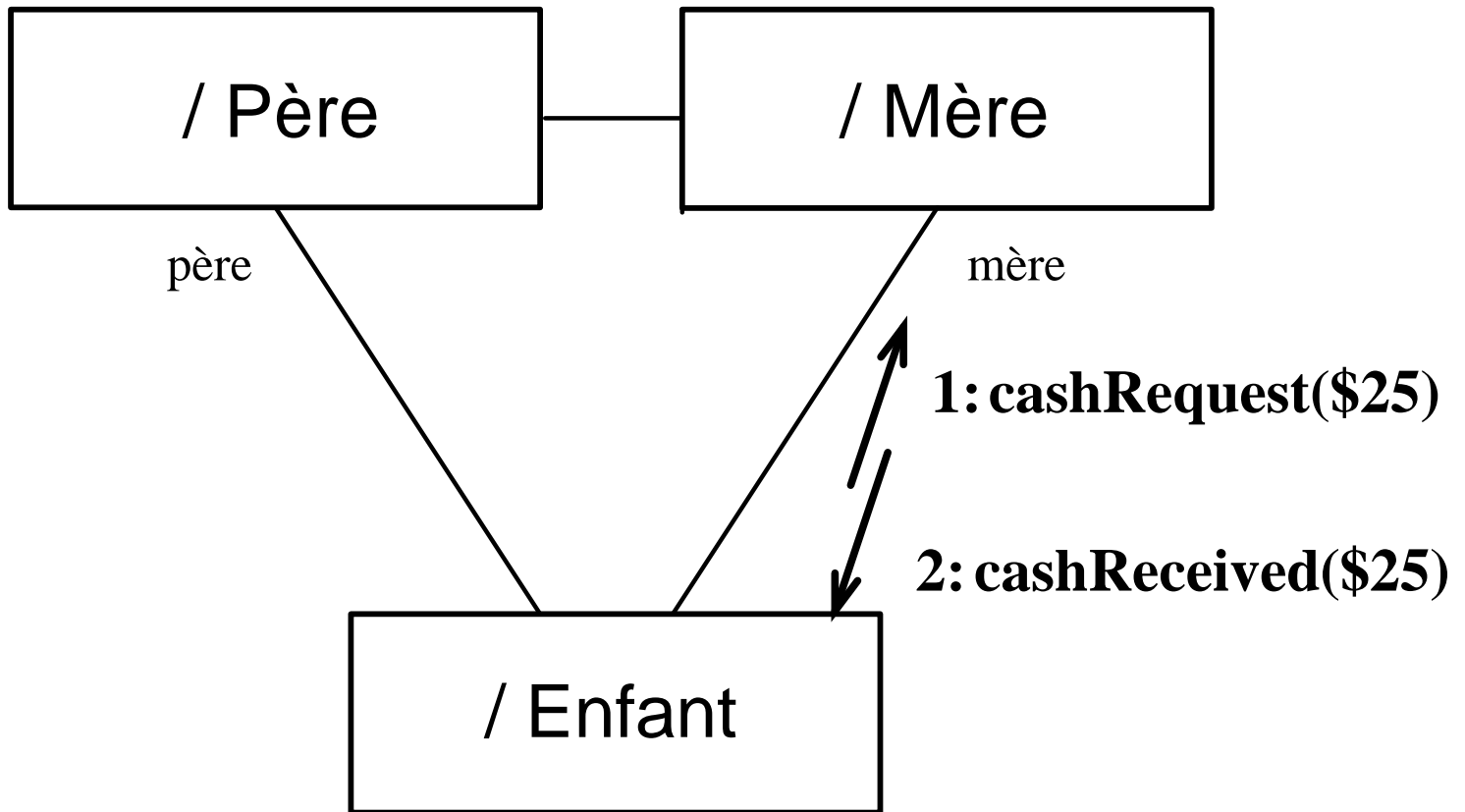
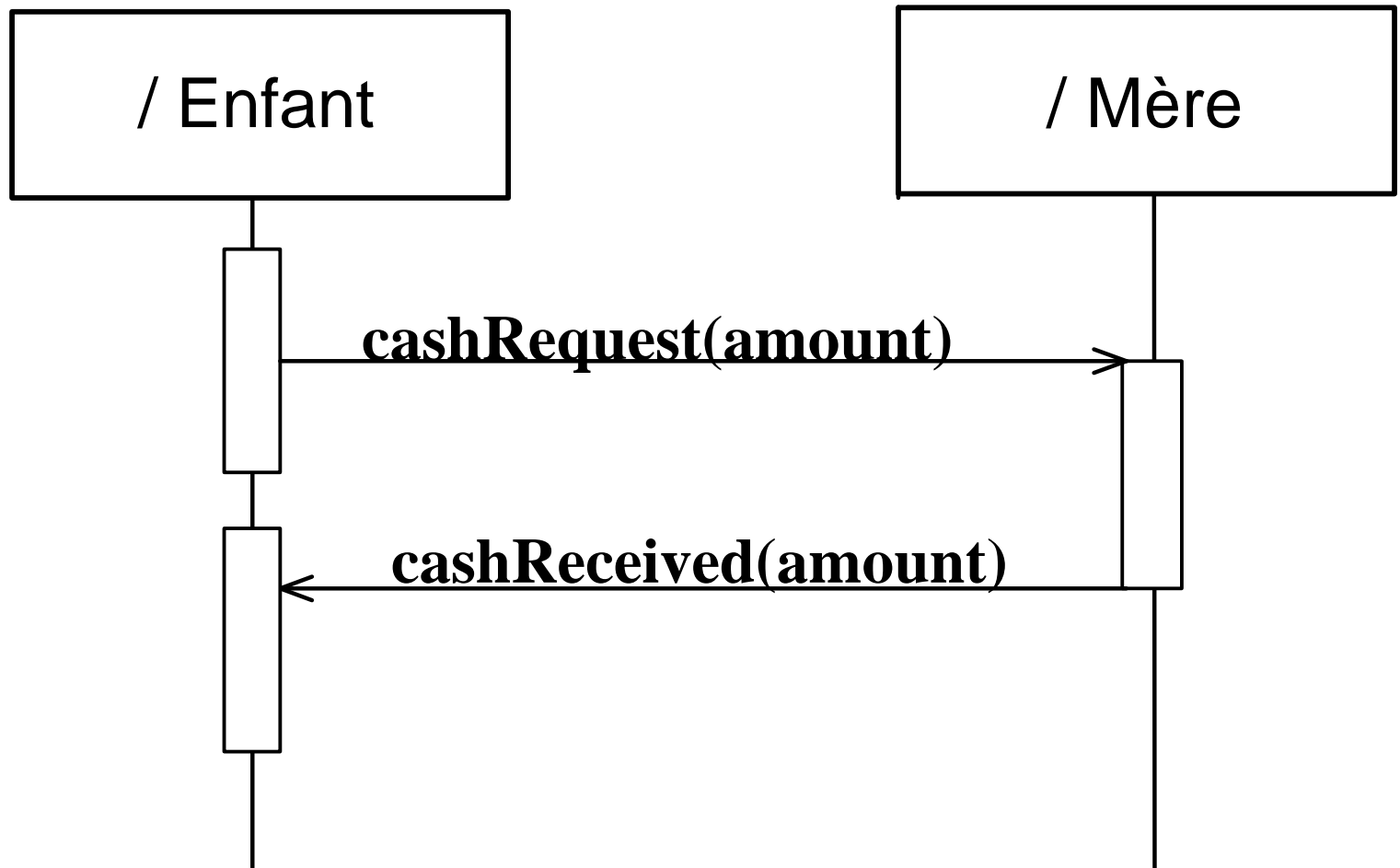


Diagramme de sequence de niveau spécification



Modélisation UML

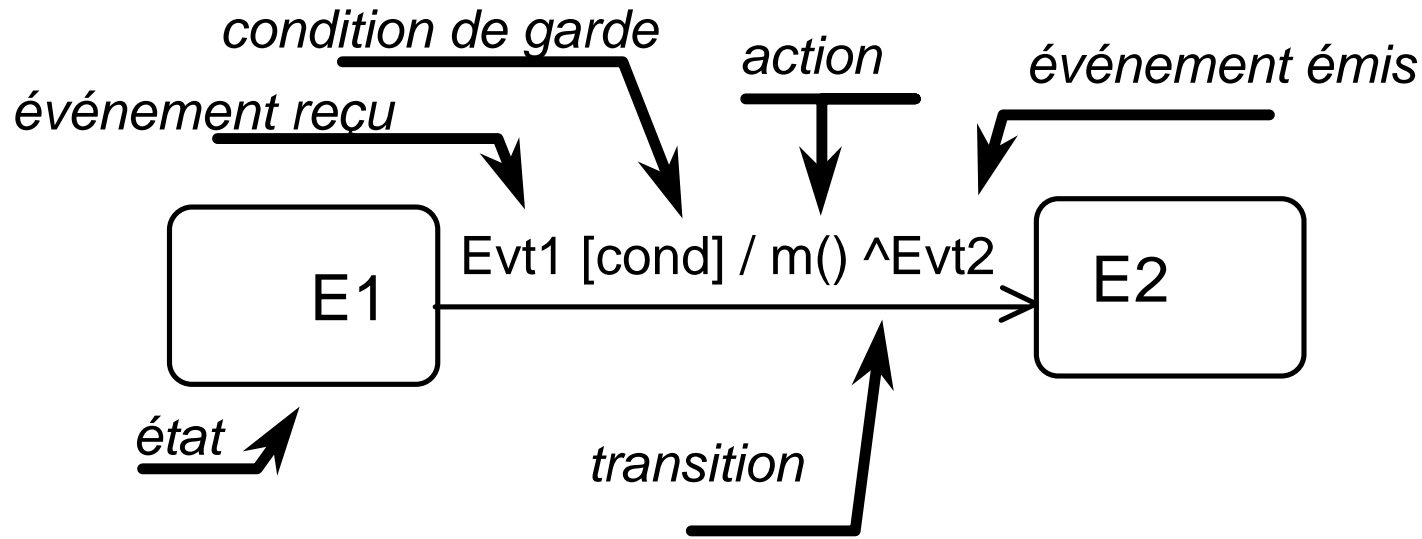
- Modélisation selon 4 points de vue principaux :
 - Aspects statiques du système (*le QUI?*)
 - » Description des objets et de leurs relations
 - Modularité, contrats, relations, généricité, héritage
 - » Structuration en paquetages
 - Vision utilisateur du système (*le QUOI?*)
 - » Cas d'utilisation
 - Aspects dynamiques du système (*le QUAND?*)
 - » Diagramme de séquences (scénarios)
 - » Diagramme de collaborations (entre objets)
 - » Diagramme d'états-transitions (Harel)
 - » Diagramme d'activités
 - Vision implantation (*le OU?*)
 - » Diagramme de composants et de déploiement



Les diagrammes d'états

- Attachés à une classe
 - 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 de Harel (idem OMT)*

Syntaxe graphique: diagramme d'états



Syntaxe :

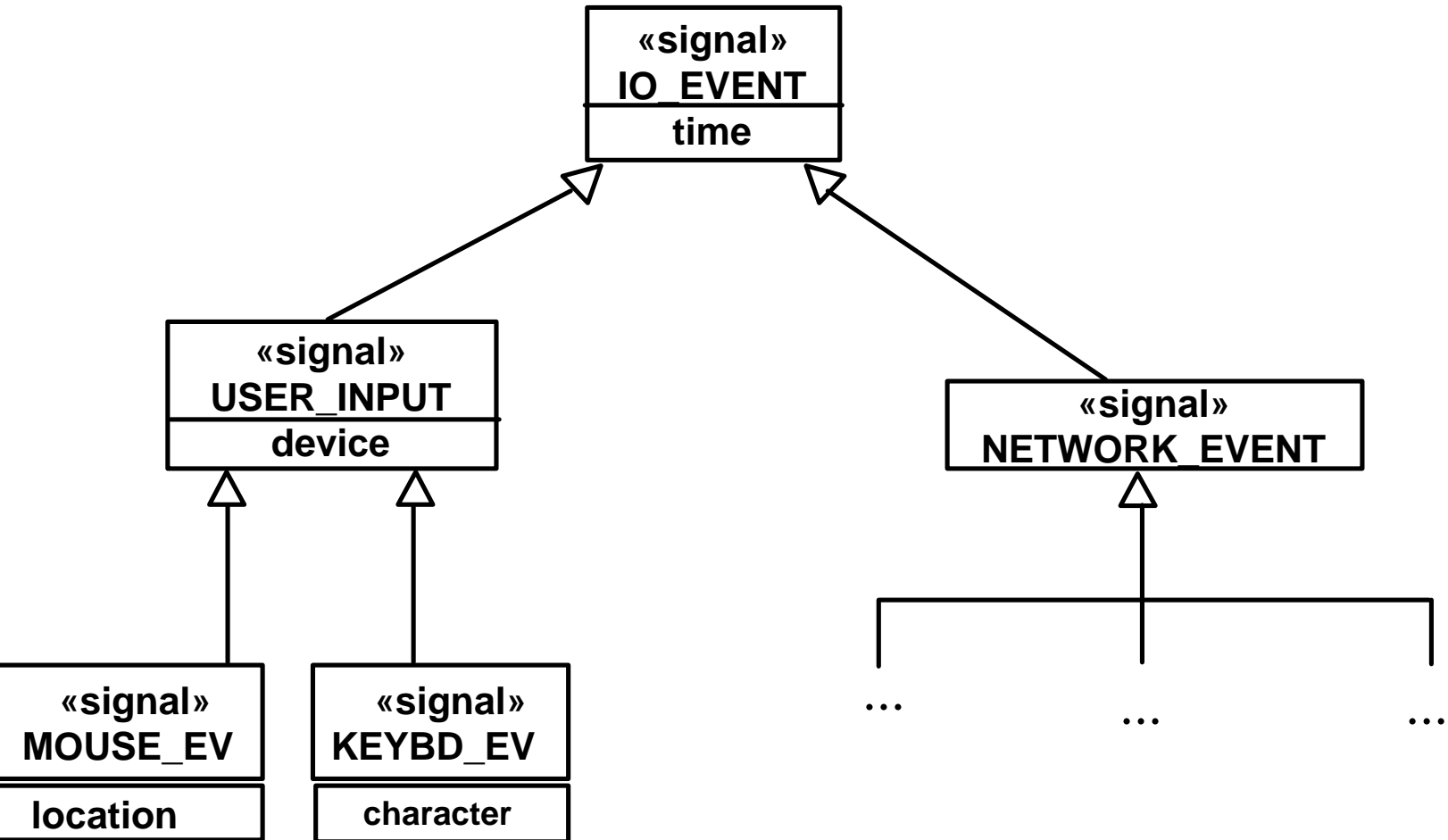
ÉvénementReçu (param : type, ...) [condition de garde] / Action ^ÉvénementsEm

Notion d'événements

- Stimulis 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»).

Les événements

- Les événements sont considérés comme des objets

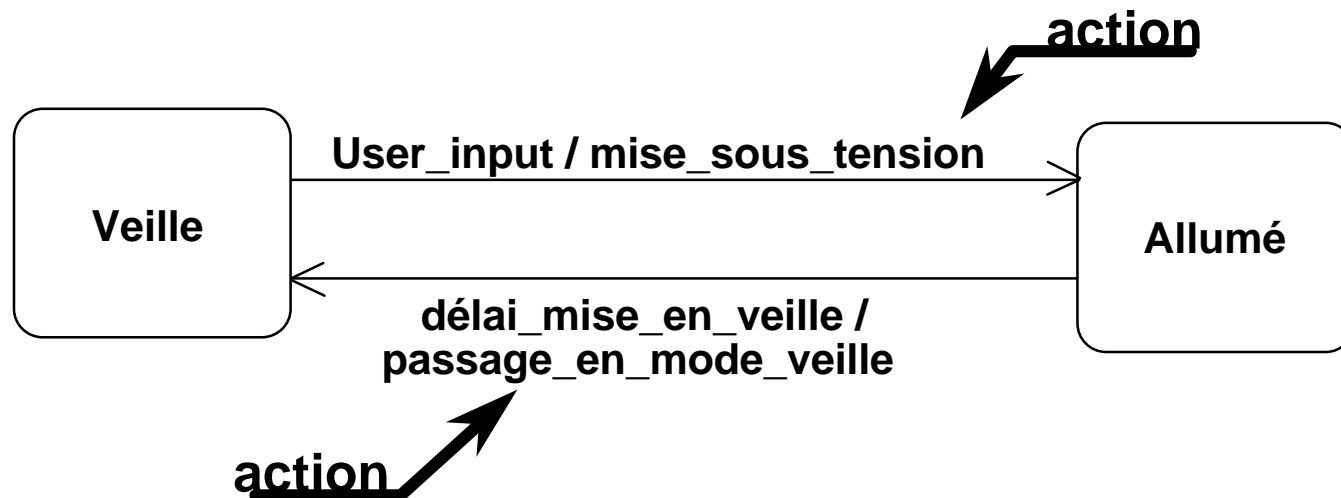


Typologie d'événements

- 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

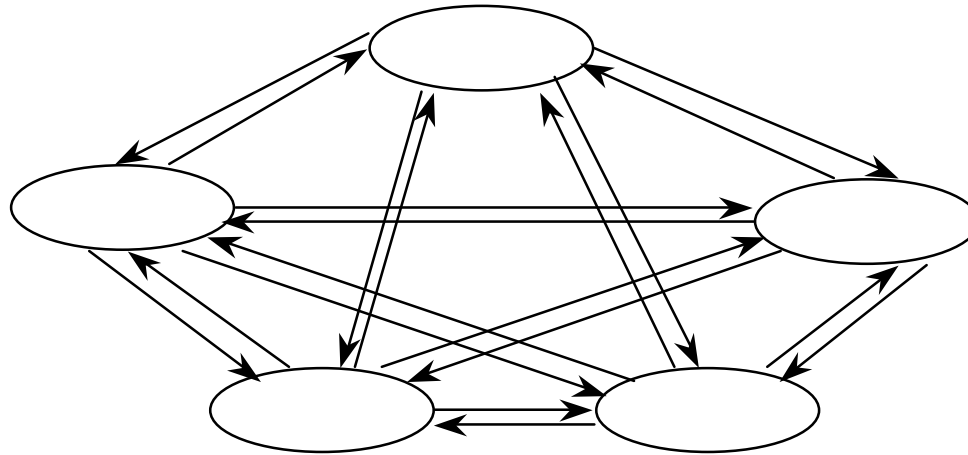


Notion d'états

- Etat : situation stable d'un objet parmi un ensemble de situations pré-définies
 - 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

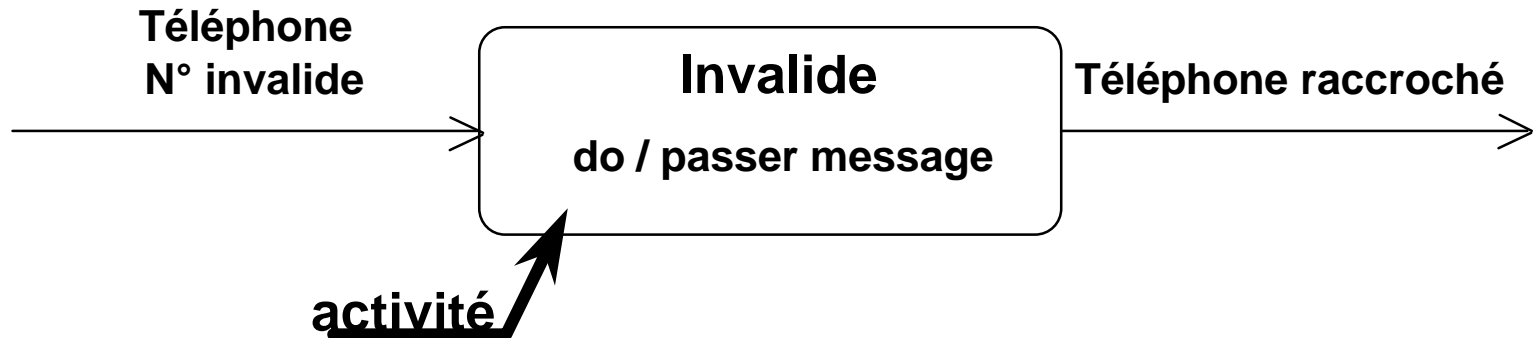
- Problème d'un diagramme d'états plats
 - Pouvoir d'expression réduit, inutilisable pour de grands problèmes
 - Explosion combinatoire des transitions.



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

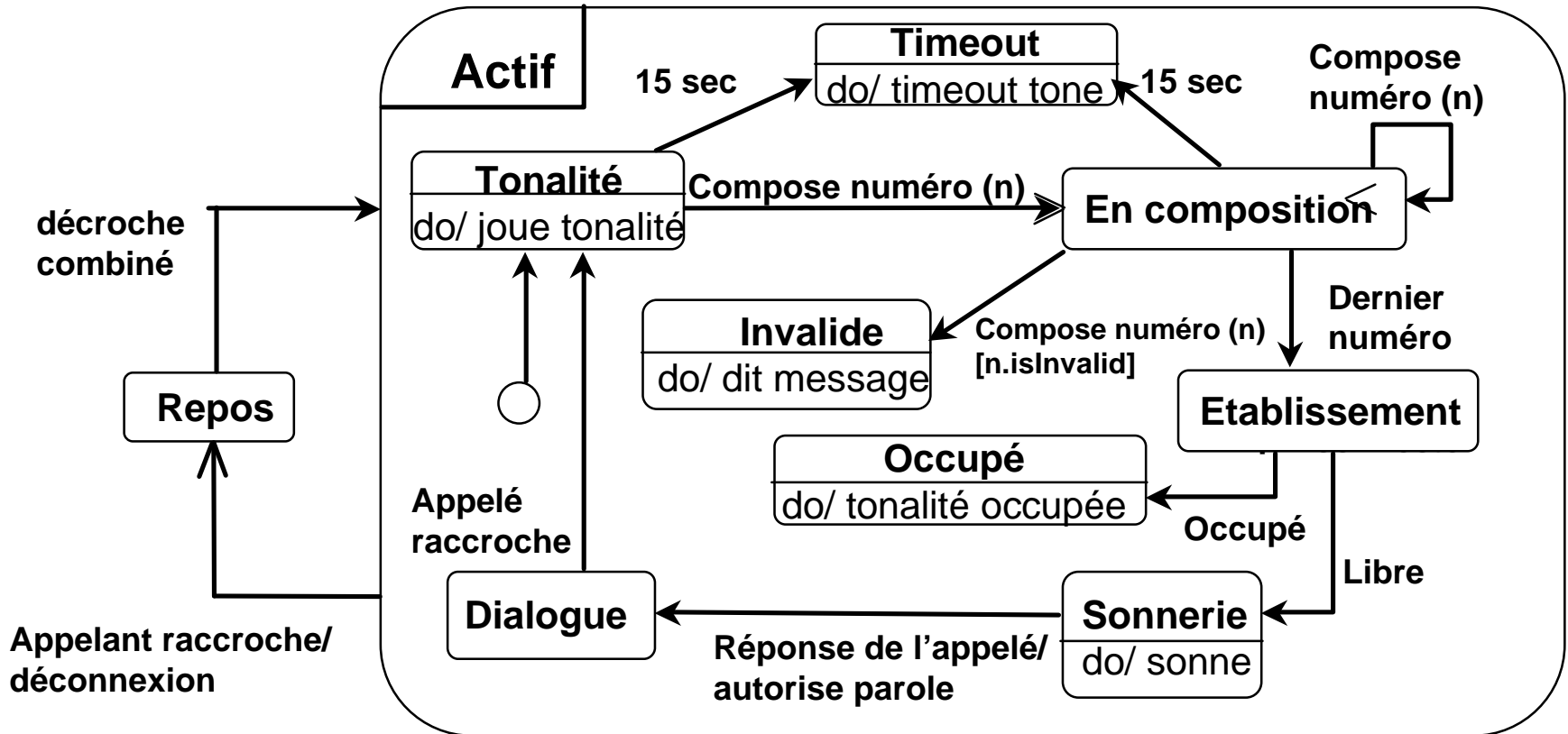
Notion d 'activité dans un état

- Activité : opération se déroulant continuellement tant qu 'on est dans l 'état associé
 - *do/ action*
- Une activité peut être interrompue par un événement.



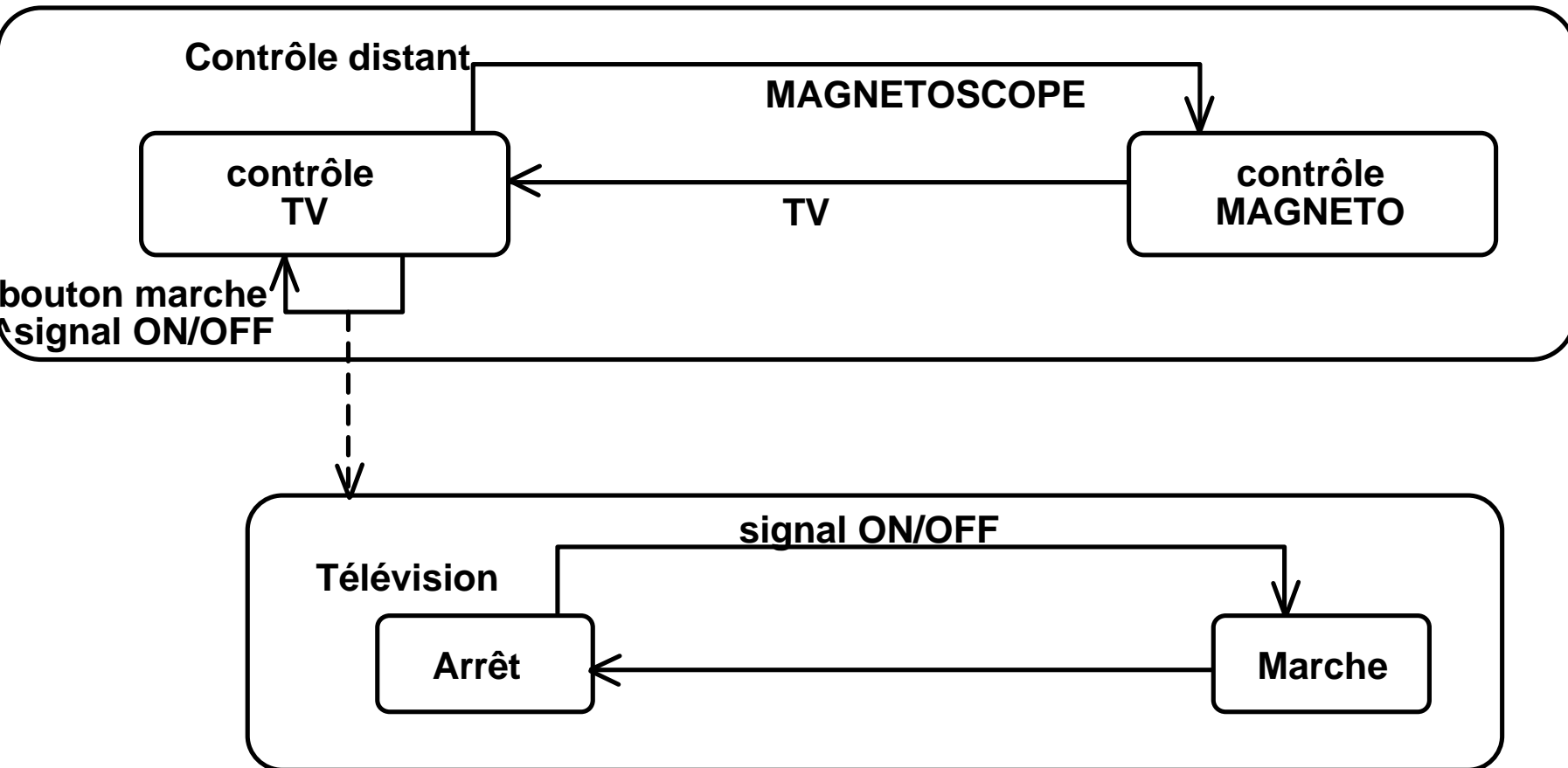
Exemple de diagramme d'états

■ un téléphone :



Émission d'événements

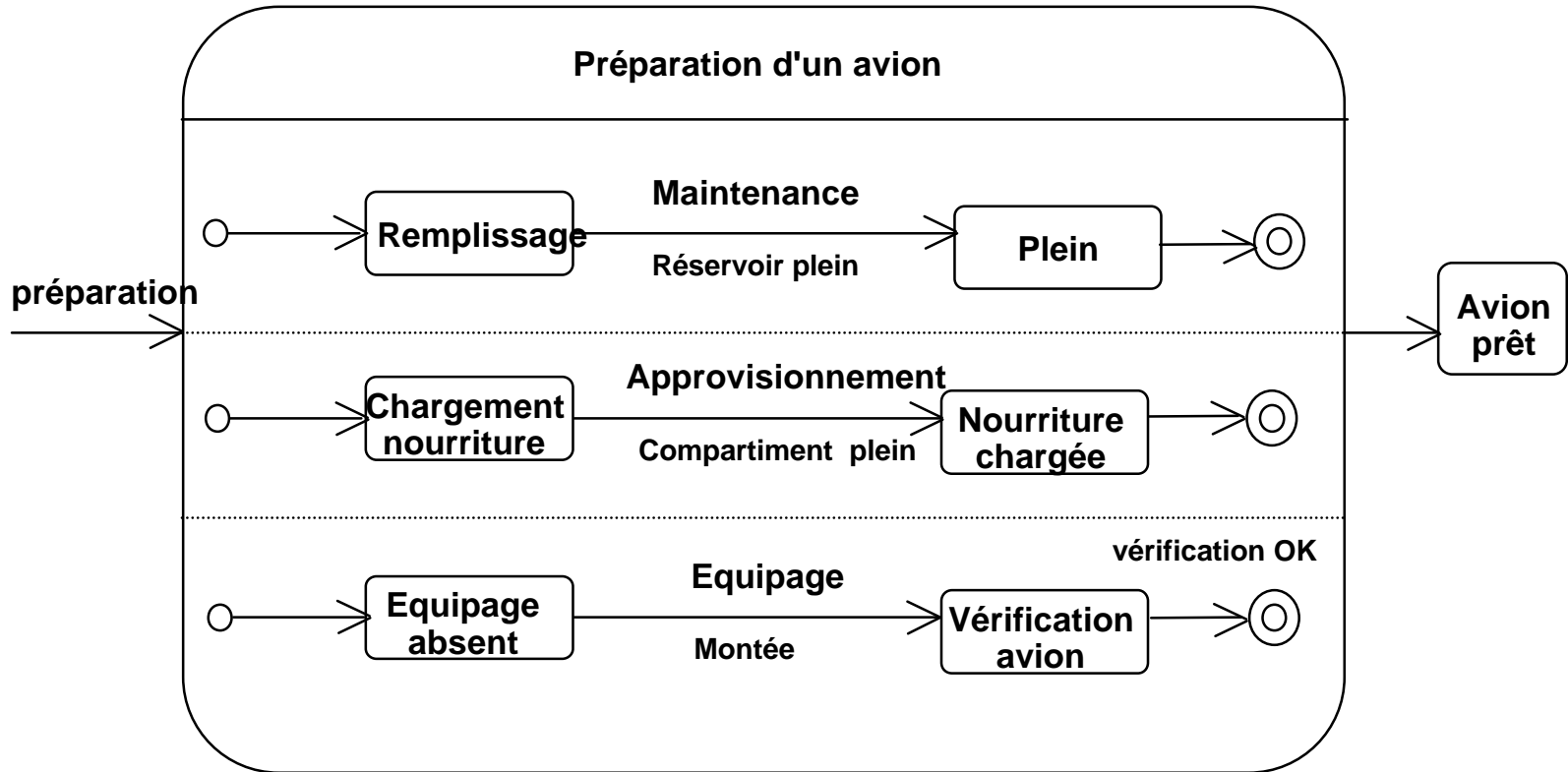
- Automate d'états d'une télécommande double :
 - TV + MAGNETOSCOPE



Diagrammes d'états concurrents

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

Exemple de concurrence



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.

Modélisation UML

- Modélisation selon 4 points de vue principaux :
 - Aspects statiques du système (*le QUI?*)
 - » Description des objets et de leurs relations
 - Modularité, contrats, relations, généricité, héritage
 - » Structuration en paquetages
 - Vision utilisateur du système (*le QUOI?*)
 - » Cas d'utilisation
 - Aspects dynamiques du système (*le QUAND?*)
 - » Diagramme de séquences (scénarios)
 - » Diagramme de collaborations (entre objets)
 - » Diagramme d'états-transitions (Harel)
 - » Diagramme d'activités
 - Vision implantation (*le OÙ?*)
 - » Diagramme de composants et de déploiement



Les diagrammes d'activité

- Traitements effectués par une opération
 - Description d'un flot de contrôle procédural
 - » Réseau d'actions et de transitions : automate dégénéré
 - » La transition s'effectue lorsque l'opération est terminée
 - Pas de déclenchement par événement asynchrone
 - » Sinon utilisation diagrammes d'états classiques
- Attachés à
 - une classe,
 - une opération,
 - ou un *use-case* (*workflow*)

Etat-action et décision

- Etat-action = raccourci pour un état où il y a :
 - une action interne
 - au moins une transition sortante
 - » production d'un événement implicite : action accomplie
 - Pas de production/réaction à des événements explicites

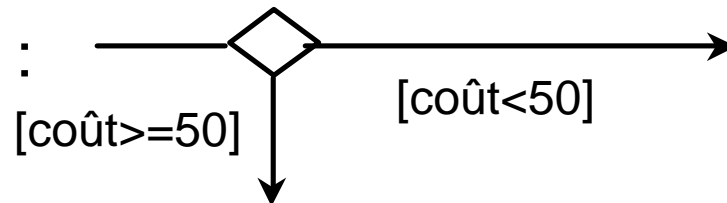
- Modélisation d'une étape dans l'exécution d'un algorithme

- Notation :

obtenir un gobelet

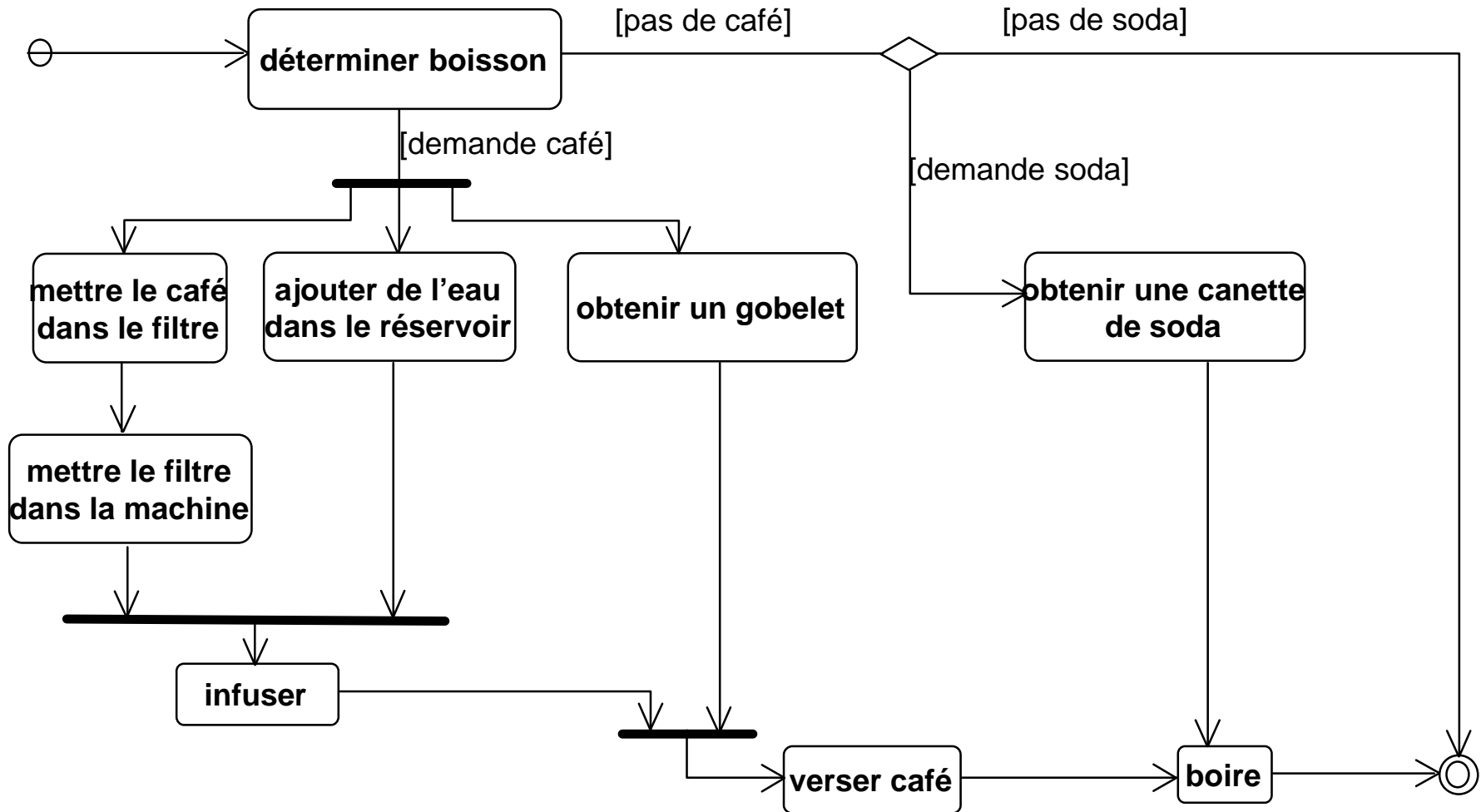
- Décision = branchement sur plusieurs transitions

- Notation :



Exemple de diagramme d'activité

opération PréparerBoisson de la classe Personne



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

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

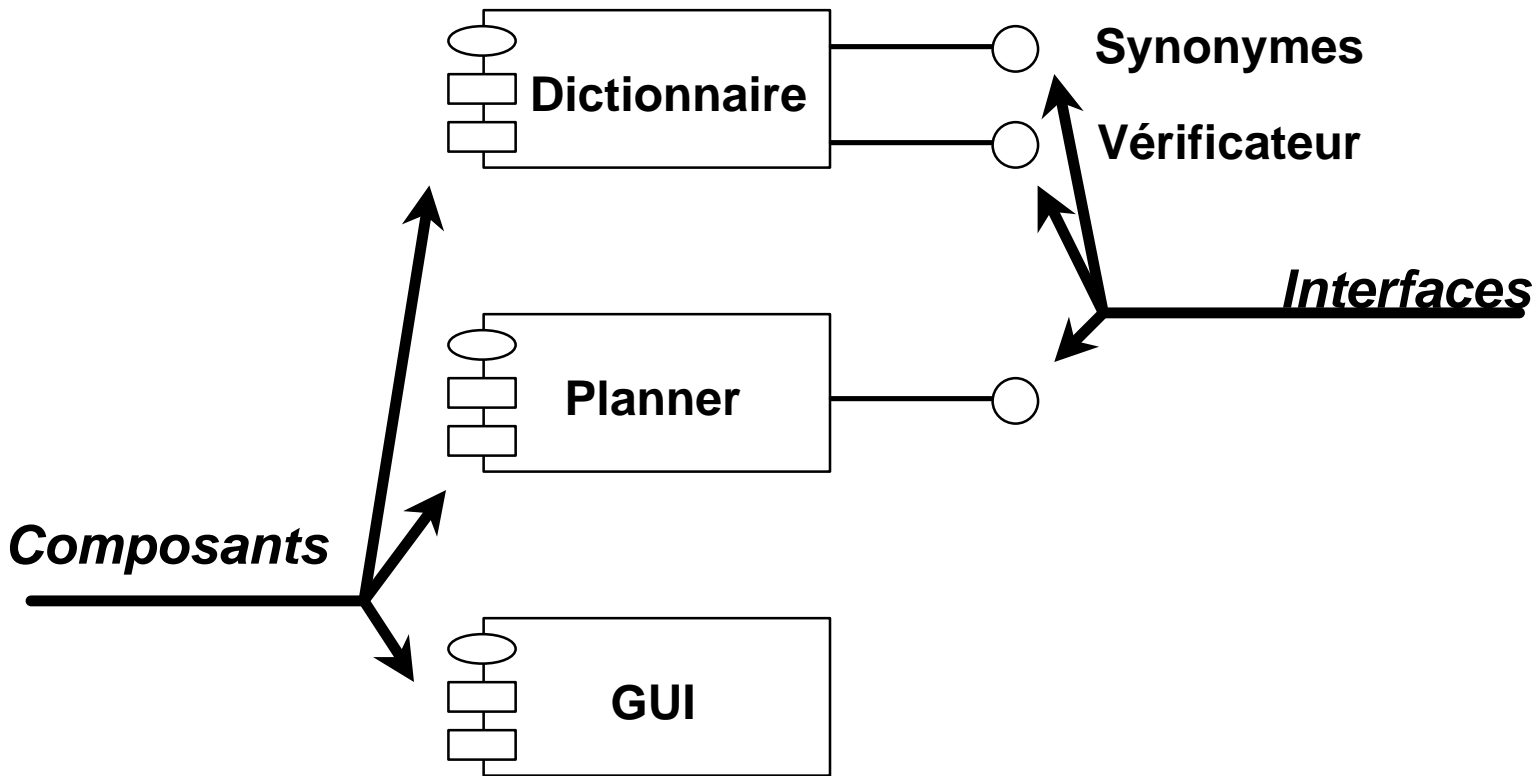
Modélisation UML

- Modélisation selon 4 points de vue principaux :
 - Aspects statiques du système (*le QUI?*)
 - » Description des objets et de leurs relations
 - Modularité, contrats, relations, généricité, héritage
 - » Structuration en paquetages
 - Vision utilisateur du système (*le QUOI?*)
 - » Cas d'utilisation
 - Aspects dynamiques du système (*le QUAND?*)
 - » Diagramme de séquences (scénarios)
 - » Diagramme de collaborations (entre objets)
 - » Diagramme d'états-transitions (Harel)
 - » Diagramme d'activités
 - ➔ – Vision implantation (*le OÙ?*)
 - » Diagramme de composants et de déploiement

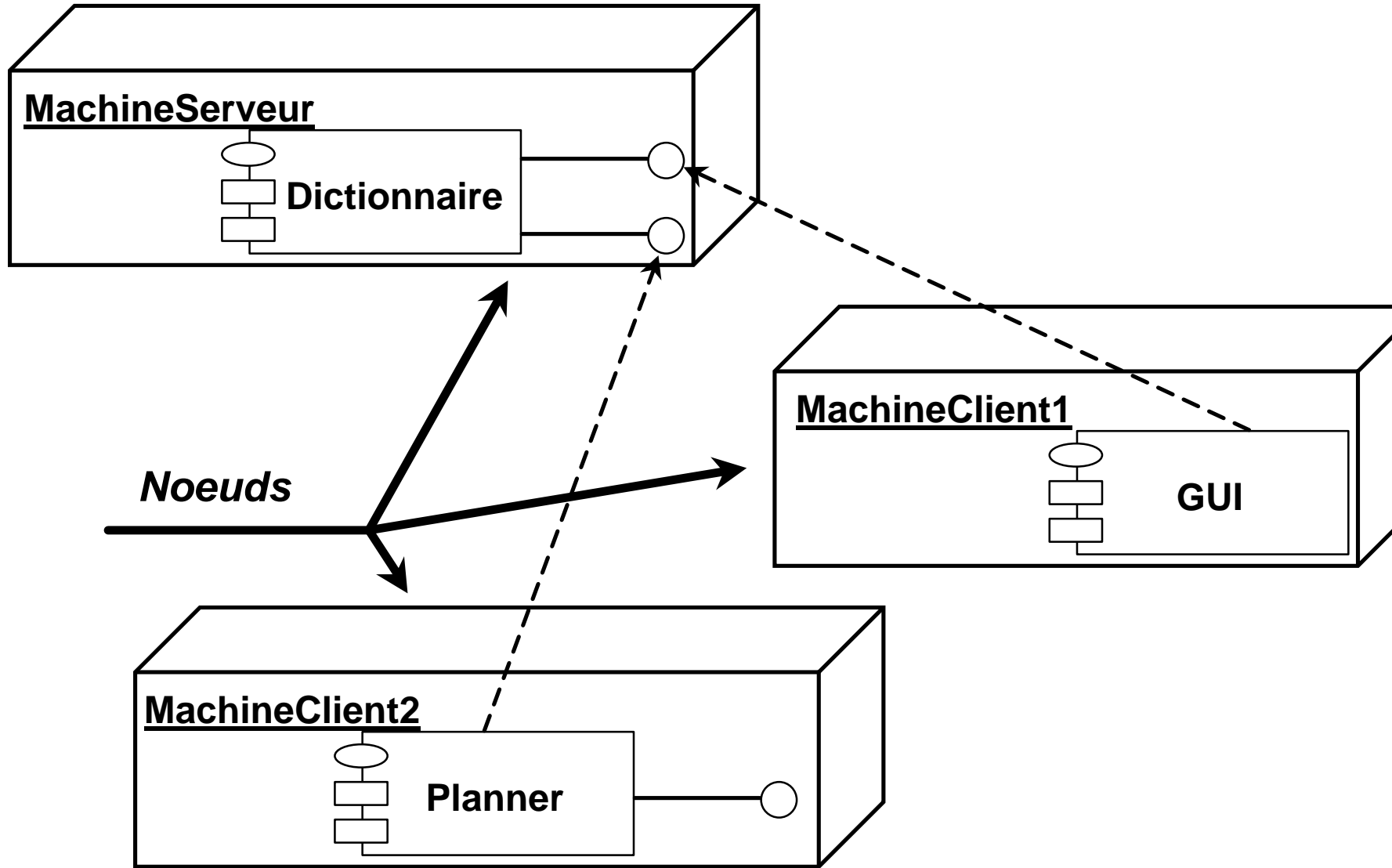
Diagrammes d'implantation

- Diagrammes de composants
 - Dépendances entre composants logiciels
 - » code source
 - » binaires, DLL
 - » exécutables
- Diagrammes de déploiement
 - Configuration des composants
 - Localisation sur les noeuds d'un réseau physique

Exemples de composants



Exemple de déploiement



Modélisation UML

- Modélisation selon 4 points de vue principaux :
 - Aspects statiques du système (*le QUI?*)
 - » Description des objets et de leurs relations
 - Modularité, contrats, relations, généricité, héritage
 - » Structuration en paquetages
 - Vision utilisateur du système (*le QUOI?*)
 - » Cas d'utilisation
 - Aspects dynamiques du système (*le QUAND?*)
 - » Diagramme de séquences (scénarios)
 - » Diagramme de collaborations (entre objets)
 - » Diagramme d'états-transitions (Harel)
 - » Diagramme d'activités
 - Vision implantation (*le OU?*)
 - » Diagramme de composants et de déploiement

Processus de développement avec UML

