Processus Systèmes d'Information - UML

➜ Enseignant : Marie Aimar,
aimar@emi.u-bordeaux.fr,

Bibliographie

Les best-of :

Conception orientée objet et applications,
G Booch (Addison-Wesley, 1992).
Le génie logiciel orienté objet,I. Jacobson,
M. Christerson, P. Jonsson, G. Overgaard (Addison-Wesley, 1993).

Les outsiders :

Méthodes orientées objet, 2nd édition,
I Graham (thompson Publisher, 1997).
Analyse des systèmes : de l'approche fonctionnelle à l'approche objet,
Ph. Larvet (InterEditions, 1994).

Spécial UML :

UML La notation unifiée de modélisation objet,
M. Lai (InterEditions, 1997).

Modélisation objet avec UML,
P.A. Muller (Eyrolles, 1998).

Les sites internet

Index to Object-Oriented Information Sources
http ://iamwww.unibe.ch/ scg/OOinfo/index.html

UML Resources
http ://www.rational.com/uml/adobe.html

Aspect-Oriented Programming Home Page
http ://www.parc.xerox.com/csl/projects/aop/

Objectifs du Cours

- ➔ **Architecture Logicielle** - Approcher les problèmes de l'analyse et de conception des systèmes informatiques (des programmes).
- ➔ Au travers du **filtre** des méthodes orientées objet - **UML**.

Ce que nous ne ferons pas !

- Traiter des problèmes de l'organisation d'un développement informatique :
  - gestion du temps,
  - gestion des équipes,
  - communication avec les autres entités de l'entreprise.
- Tous les concepts seront abordés uniquement au niveau du *Design* d'une application.

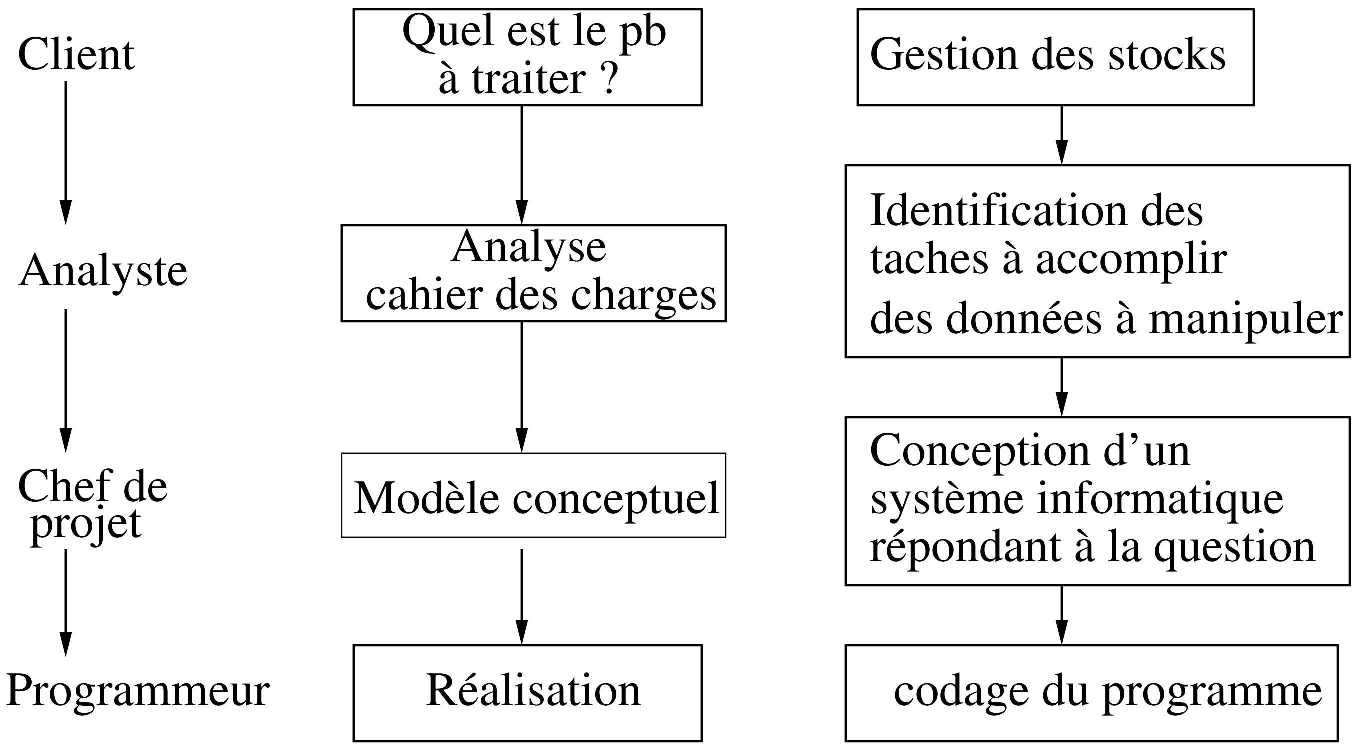
Combattre quelques idées reçues

- Il existe une étape **Analyse** - **Conception** du logiciel différente de l'analyse des besoins d'une entreprise (ou d'un service).
- Ne pas tenter de solutionner tous les problèmes d'un service avec un seul logiciel. Apprendre à cerner précisément le problème à traiter.
- Les méthodes orientées objet ne sont pas faites (et n'ont pas prétendue être faites) pour traiter les problèmes organisationels de management d'une entreprise ou d'un service.

Analyse des systèmes

1. Analyser un problème.
2. Proposer un modèle.
3. Concevoir un solution.
4. Réaliser (programmer) la solution.

Schema classique - caricatural



Les erreurs de cette vision des choses

➔ Vous n'interviendrez *"jamais"* sur un projet totalement nouveau

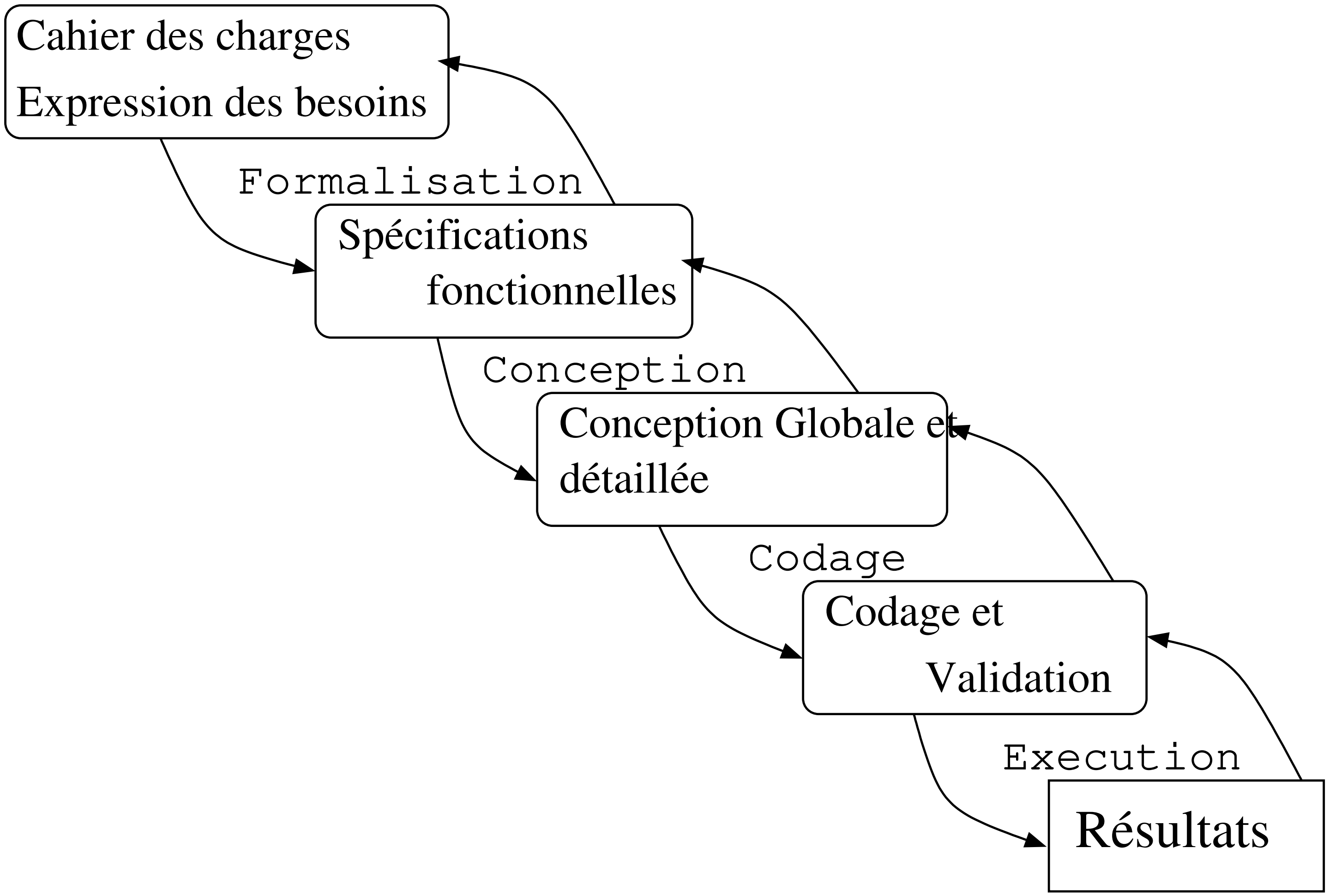
➔ Ignore les relations avec les autres applications de l'entreprise : la gestion des stocks peut communiquer avec la fabrication des marchandises, les commandes des clients . . .

➔ cloisonne les activités : organisation hiérarchique de l'analyste au programmeur, le client n'a pas de contact avec le réalisateur final.

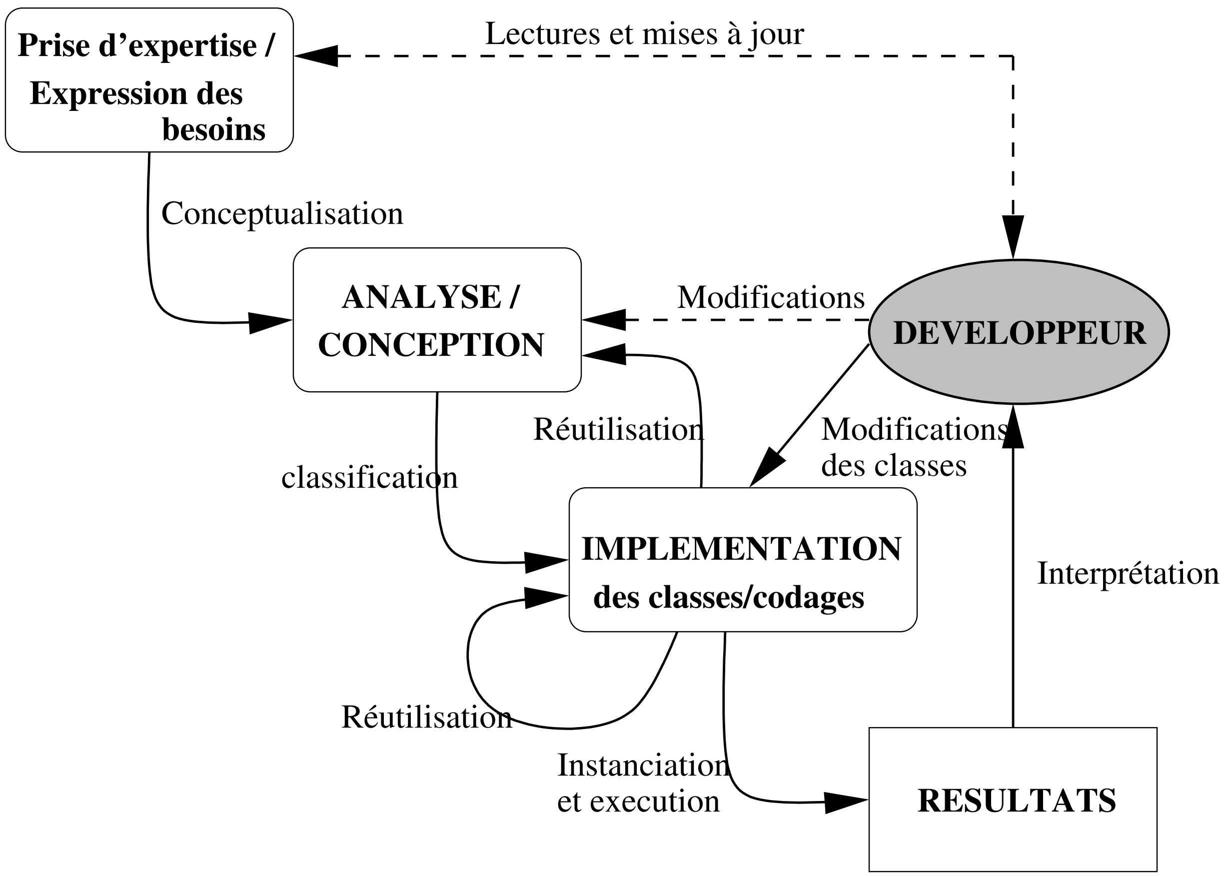
➔ Un client *connait pas* ses besoins.

Un système doit être facile à modifier, à enrichir de nouvelles fonctionnalités, au pire changer d'objectif. L'architecture doit donc être modulaire et facile à maintenir.

Développement en cascade



Développement Orienté Objet



Développement Orienté Objet

➔ Conséquences de l'application des méthodes OO : les phases d'analyse, de conception et de programmation sont très liées.

➔ Historique des méthodes orientées objet :

1. langages de programmation,
2. méthodes de conception,
3. méthode d'analyse.

Quelques repères

➔ Age de l'invention :

1967 - le langage de programmation SIMULA.
1970 - SMALLTALK (Palo Alto).

➔ Age de la confusion :

1980 - les langages ++.
Les méthodes de conception se multiplient

➔ Age de la maturité :

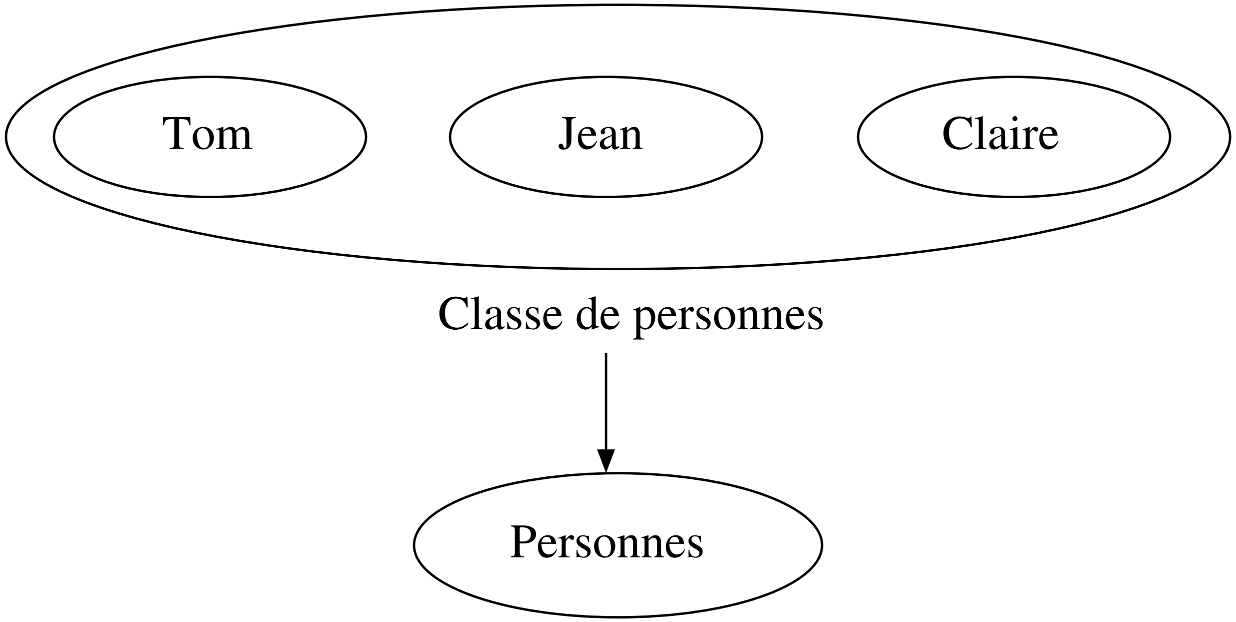
1990 - Object Management Group : standardisation.
Unification des méthodes OMT (Booch) OOSE (Jacobson) et Rumbaugh : Unified Modeling Language (version 1.0 1997).

Les concepts de base

- **Objets** : unités de base organisées en classes et partageant des traits communs (attributs ou procédures). Peuvent être des entités du monde réel, des concepts de l'application ou du domaine traité.
- **Encapsulation** : les structures de données et les détails de l'implémentation sont cachés aux autres objets du système. La seule façon d'accéder à l'état d'un objet est de lui envoyer un message qui déclenche l'exécution de l'une de ses méthodes. Les types d'objets peuvent être assimilés aux types de données abstraites en programmation. Abstraction et encapsulation sont complémentaires, l'encapsulation dessant des barrières entre les différentes abstractions.

- ➔ **Héritage** : chaque instance d'une classe d'objet hérite des caractéristiques (attributs et méthodes) de sa classe mais aussi d'une éventuelle super-classe. L'héritage est un des moyens d'organiser le monde c.-à-d. de décrire les liens qui unissent les différents objets.
- ➔ **Polymorphisme** : possibilité de recourir à la même expression pour dénoter différentes opérations. L'héritage est une forme particulière du polymorphisme caractéristique des systèmes orientés objet.
- ➔ **Modularité** : partition du programme qui crée des frontières bien définies (et documentées) à l'intérieur du programme dans l'objectif d'en réduire la complexité (Myers). Le choix d'un bon ensemble de modules pour un problème donné, est presque aussi dificile que le choix d'un bon ensemble d'abstractions.

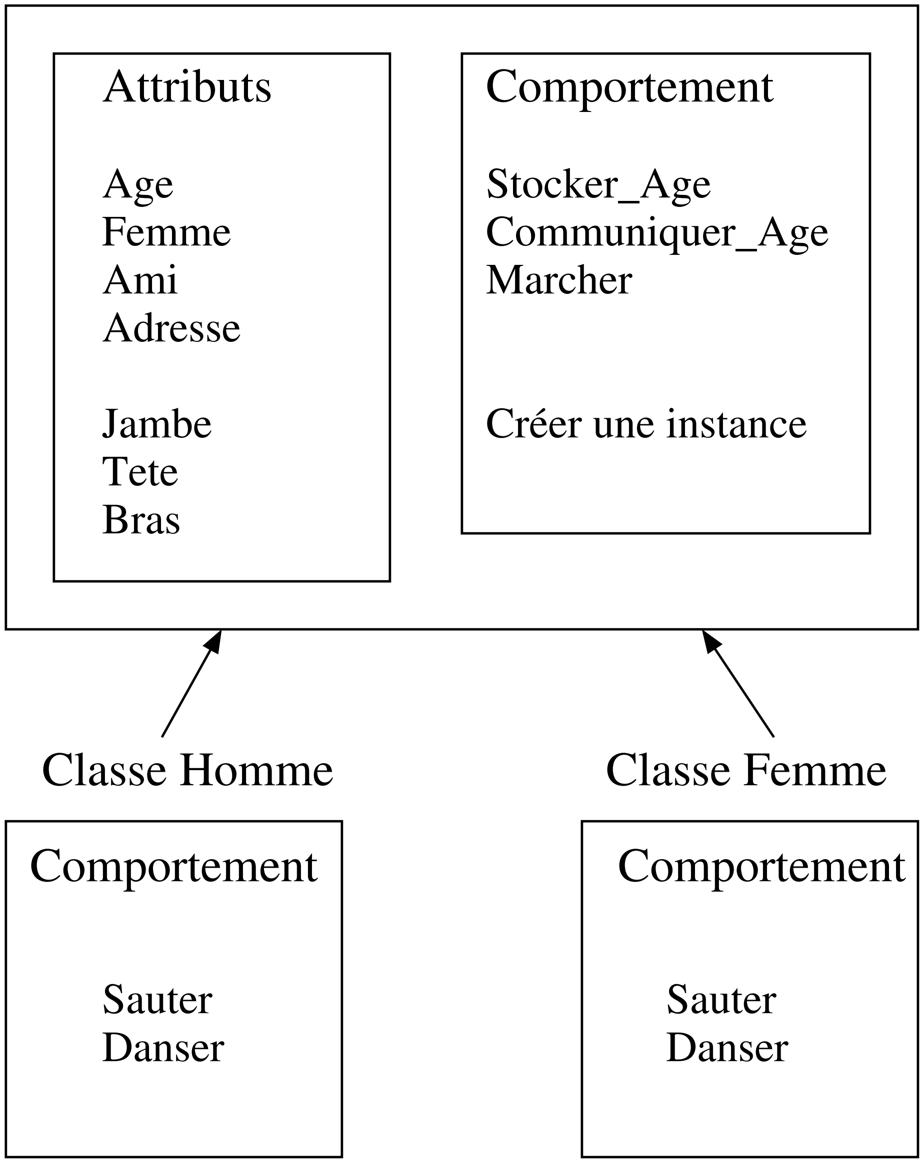
Faire des choix



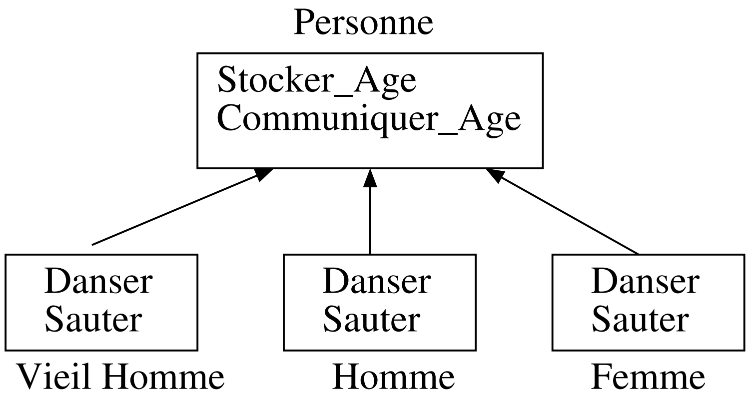
➔ Quelles sont les caractéristiques - attributs - d'une personne ?

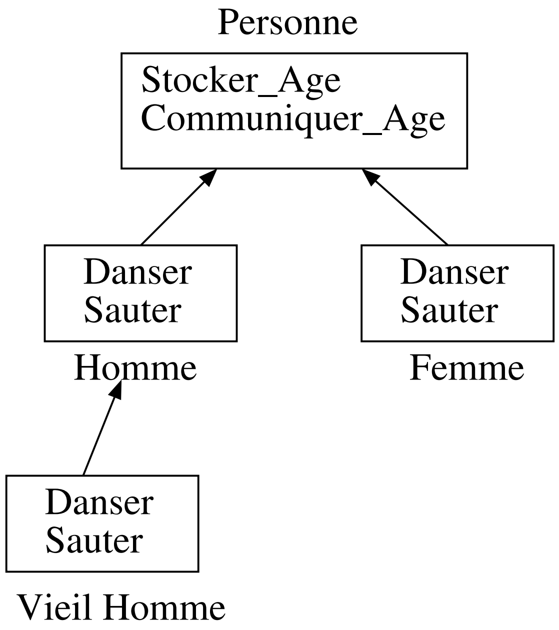
➔ Quels sont les comportements génériques - fonctions - d'une personne ?

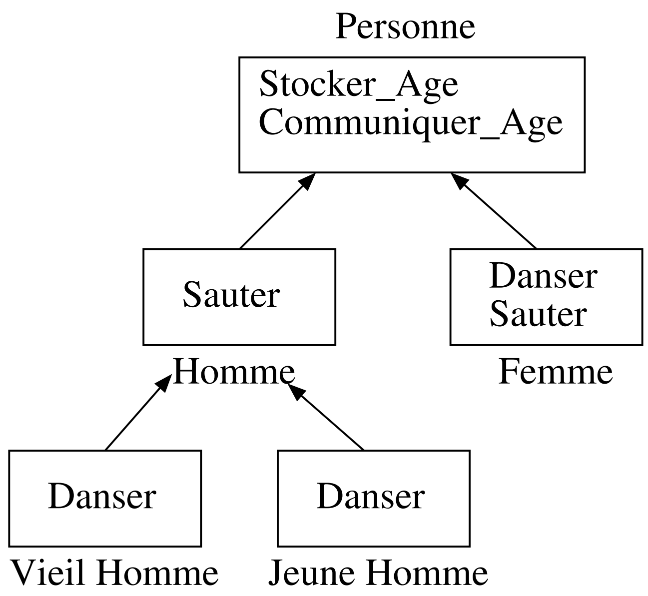
Classe Personne



3 manières d’ajouter une classe Vieil Homme







Trouver les bons objets

- Méthode de désagrégation / agrégation :
  - désagréger un module $\Rightarrow$ une suite de modules,
  - agréger une suite de modules $\Rightarrow$ un module.

Désagrégation

On part d'un tout que l'on éclate en plusieurs parties. Chaque partie, formant à son tour un tout, est susceptible d'être à nouveau éclatée en parties plus petites.
Il est difficile d'exprimer en décomposition logicielle ce qu'est une partie.
La conception fait l'hypothèse que le système est un tout. Pour détailler et exprimer la solution, on postule que ce tout est composé de parties cohérentes séparables.

Dans un premier temps, la décomposition est basée sur les entités du domaine du problème.
La désagrégation est très différente de la décomposition fonctionnelle puisqu'une fonctionnalité n'est pas une entité du monde concret.
La granularité de la taille des entités à utiliser est un facteur important de l'effort d'abstraction à réaliser.

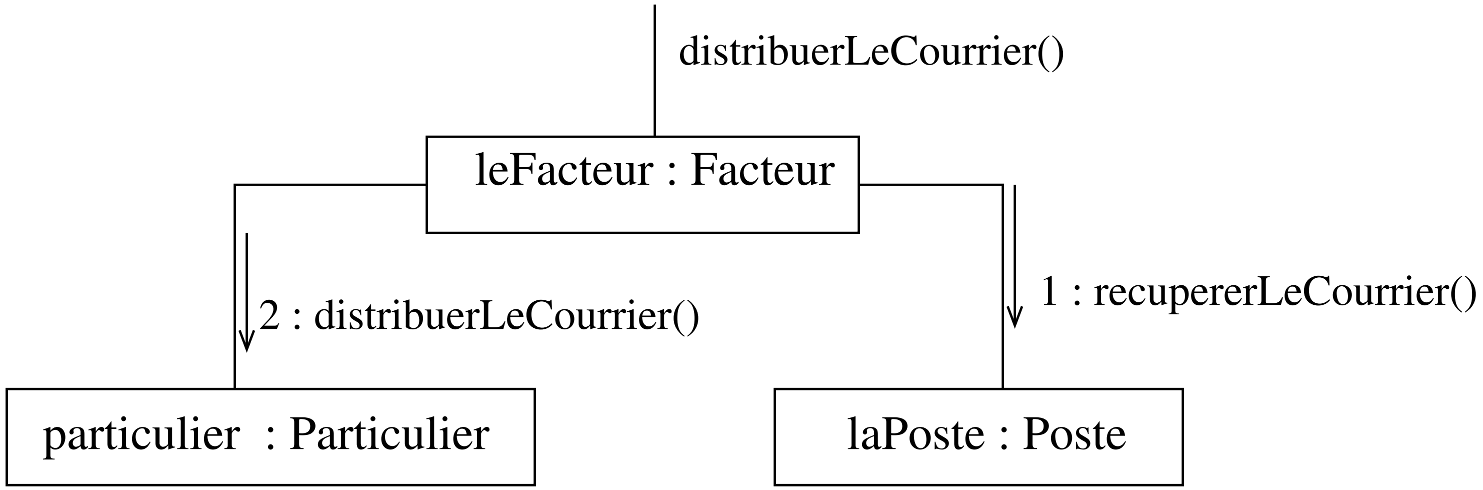
Comment faire trouver les bons Objets ?

c.-à-d. : 1) comment trouver un objet ?
2) comment distinguer un bon objet d'un mauvais ?

Quelques règles d´ecriture d'un module

- ➔ Un module représente un concept et tout le concept.
- ➔ Pour représenter une idée, il faut que cette idée existe.
- ➔ Ne pas regrouper dans un module des opérations qui n'ont pas de raisons particulières d'être ensemble (écriture de modules fourre-tout).
- ➔ Pour concrétiser une idée le choix du nom du module est un élément puissant d'expression (exemple les design patterns).
- ➔ Dans une première phase *"simpliste"* le choix des méthodes correspond aux verbes.

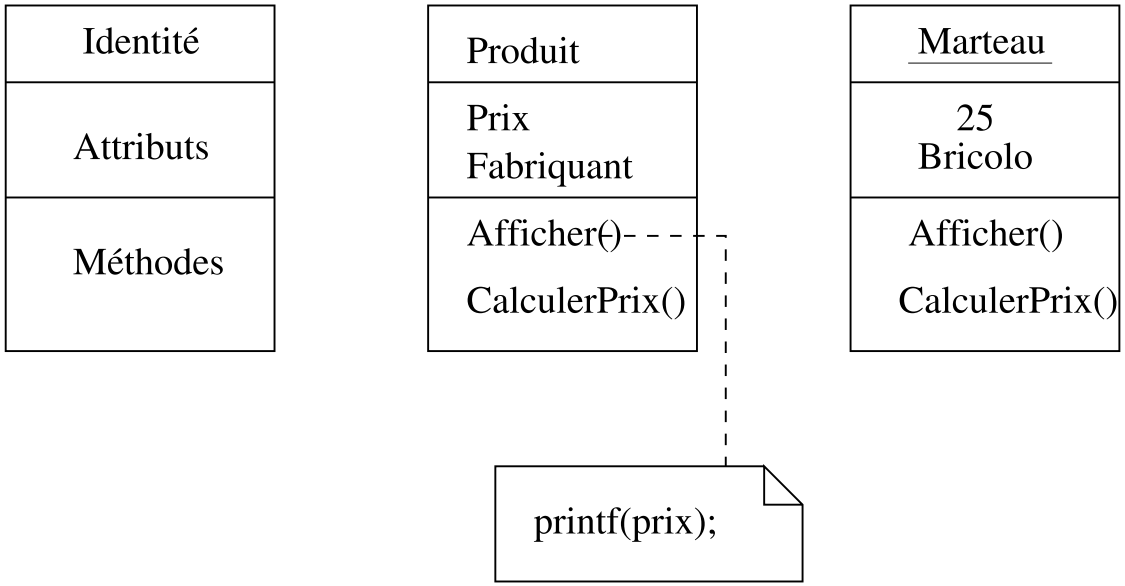
Un exemple.. enfin !



Objets et Classe d'objets

$$\boxed{\text{Objet} = \text{Etat} + \text{Comportement} + \text{Identité}}$$

➔ Conventions graphiques de UML

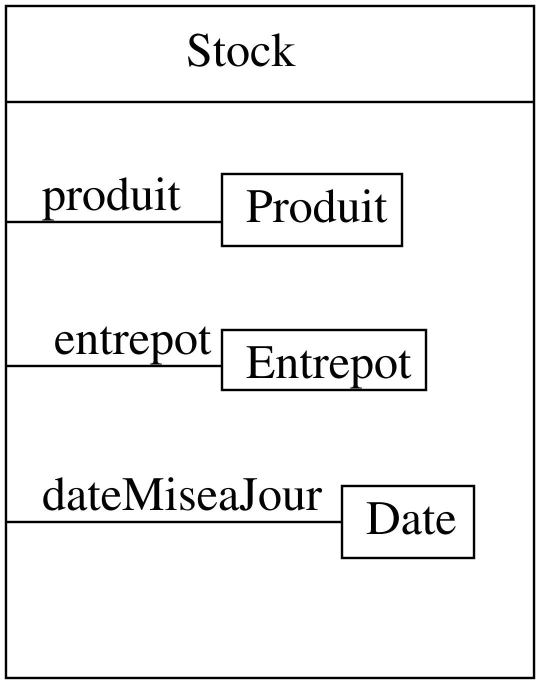


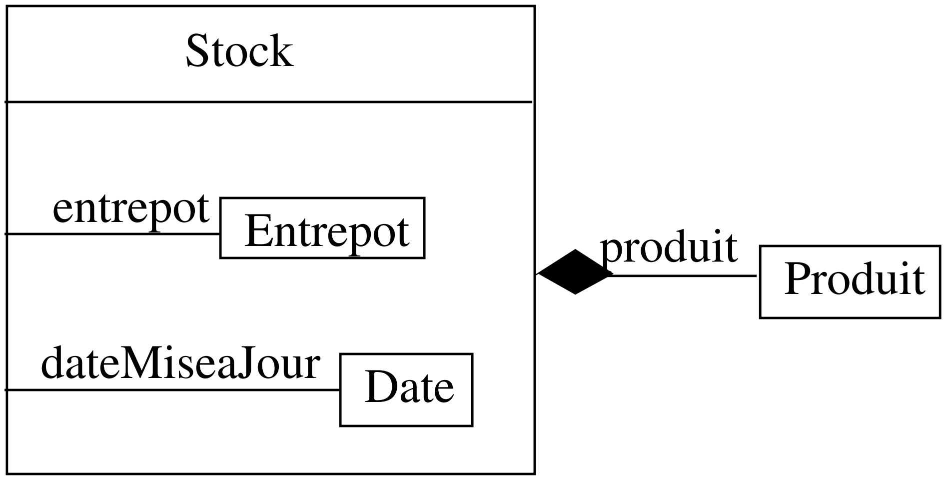
Les classes abstraites sont représentées uniquement par leur nom dans un rectangle. Il est possible de faire apparaitre le mot `abstract` sous ce nom.

La composition d'objets

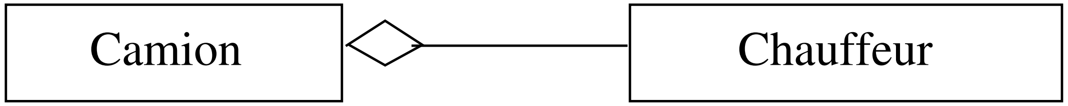
Les attributs d'un objet peuvent être eux-même des objets.

➜ **La composition par valeur** : la construction d'un objet physique implique la construction de ses attributs par valeur.





➜ **La composition par référence** : c'est un lien de référence qui peut être partagé par plusiseurs objets.



La construction du conteneur n'implique pas la construction de l'objet référencé.
NB : le losange se place du coté de l'objet référençant.

La visibilité des attributs et des méthodes

- ➔ Publique : un attribut ou une méthode publique est spécifée avec le signe +.
- ➔ Privée : un attribut ou une méthode privée est spécifée avec le signe −.
- ➔ Protected : un attribut ou une méthode protégée est spécifée avec le signe #.

Signature

La signature d'une méthode se compose de :

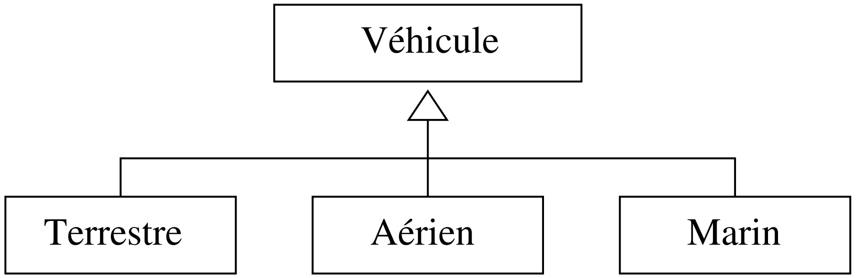
- ➔ son nom,
- ➔ le nombre et le type de ses paramètres en entrée,

Exemple : `public void afficher(String , Integer) ;`

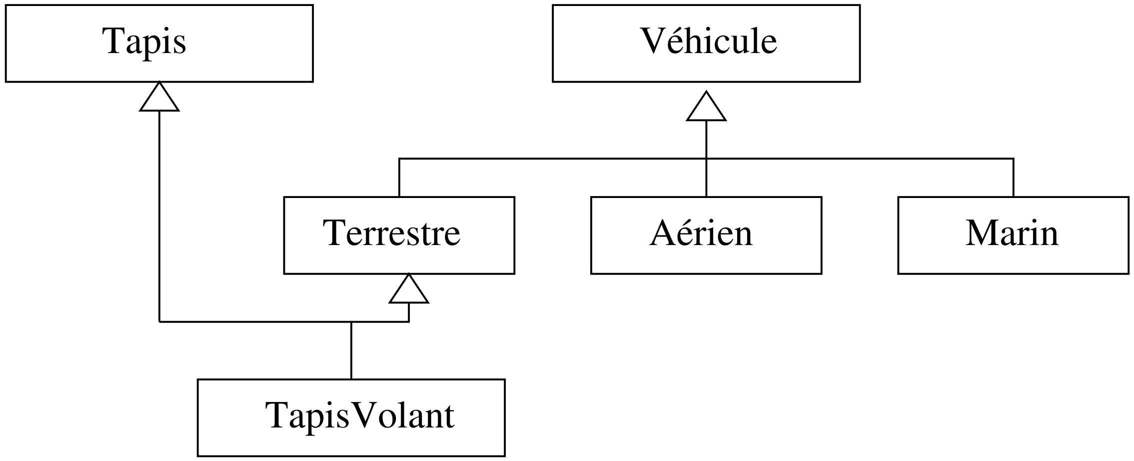
Dans un espace défini, deux méthodes peuvent avoir le même nom si elles n'ont pas la même signature.

Représentation de l'héritage

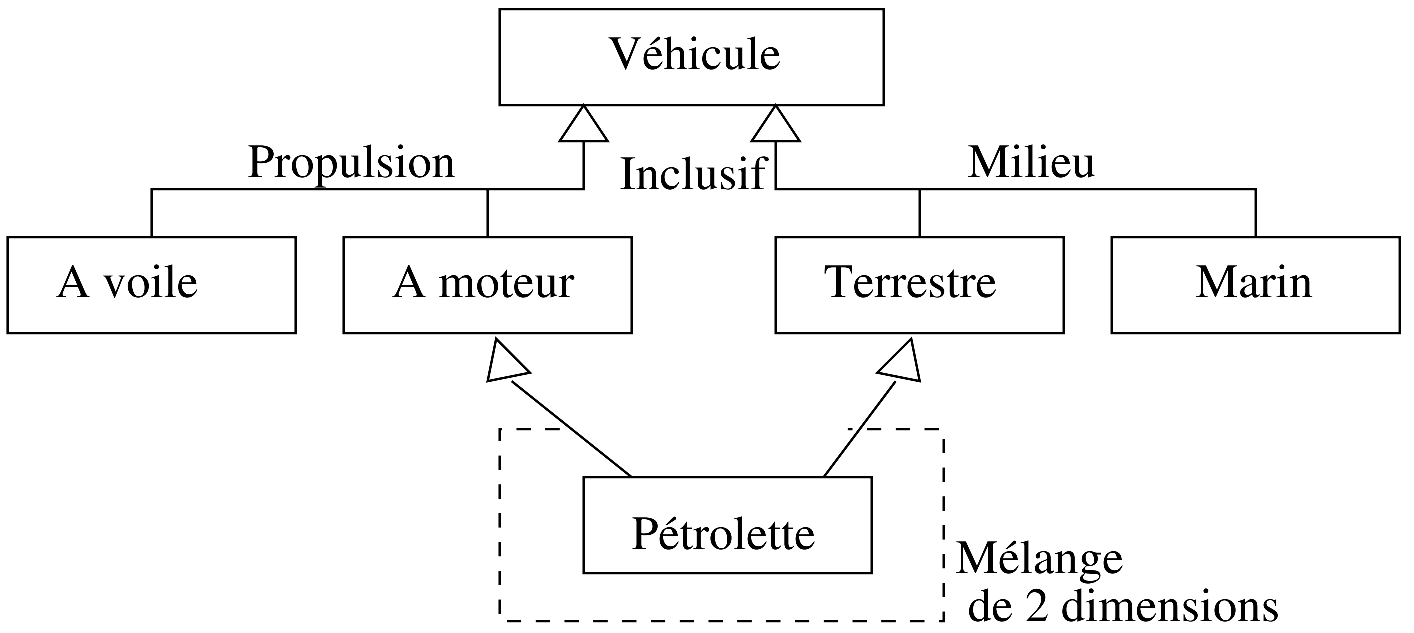
Héritage simple



Héritage multiple



Héritage multiple inclusif



Un exemple

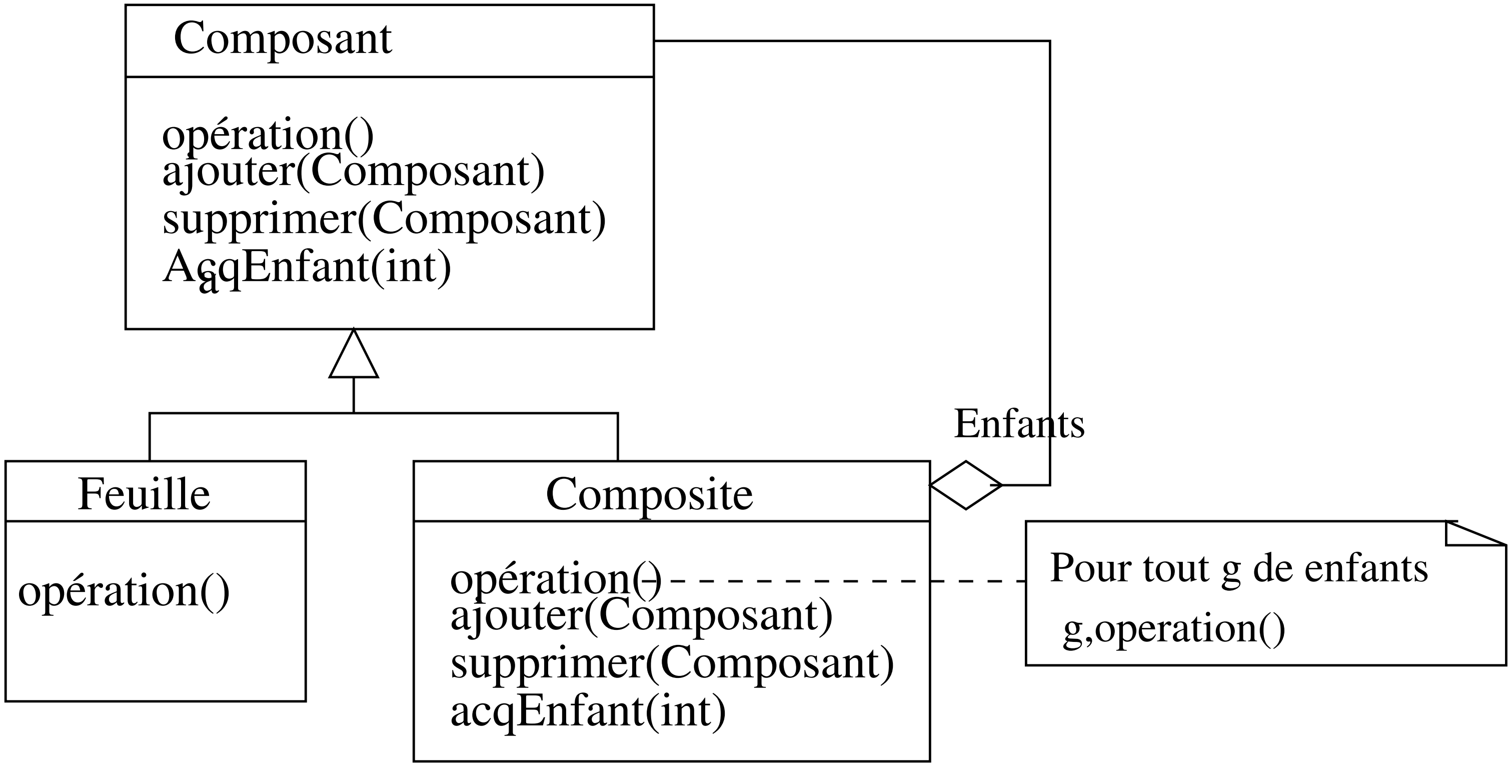
Contexte :

Un utilisateur d'interface graphique compose des dessins complexes à partir d'éléments simples tel que les carrés, les rectangles . . .Une représentation simple définit des classes primitives graphiques teles que `Texte, Ligne` . . .et quelques autres classes classes jouant le rôle de containers.

Problème :

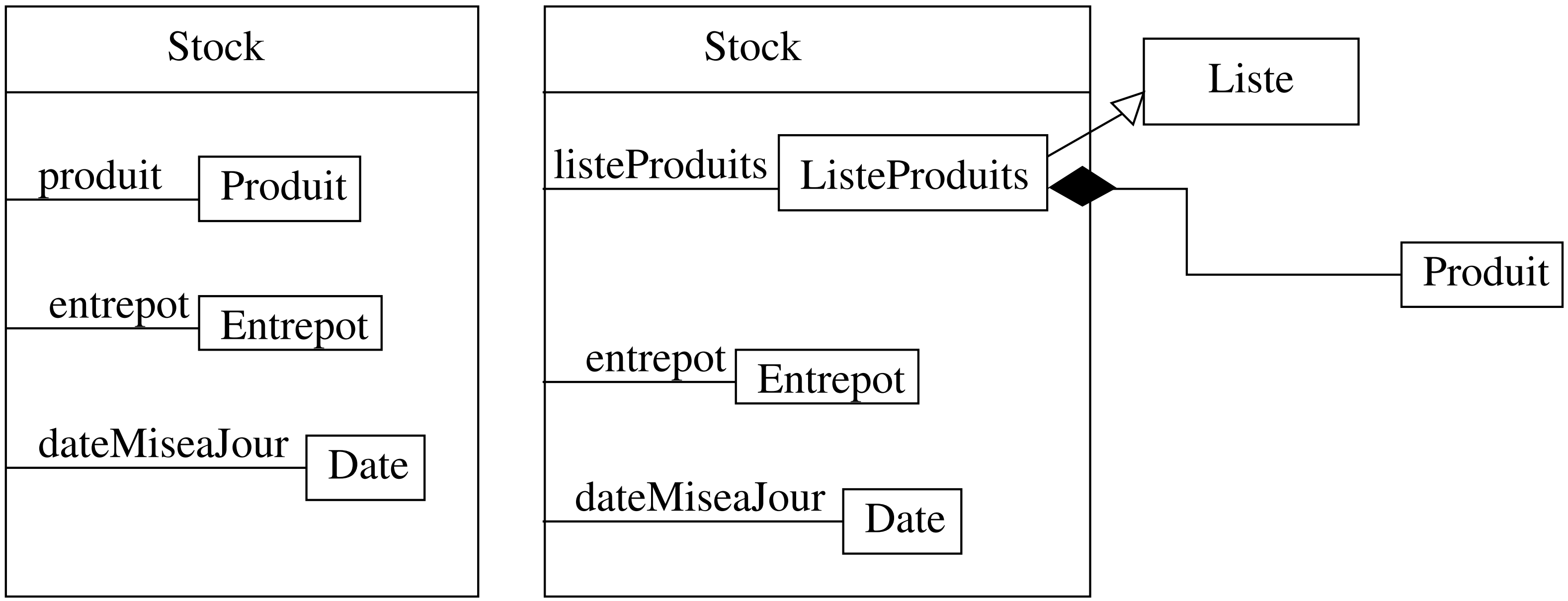
Cette approche ne permet pas de traiter de façon identique les objets containers et les objets primitives alors que l'utilisateur les traite de la même manière.

L'exemple du pattern `Composite`



Association de classes

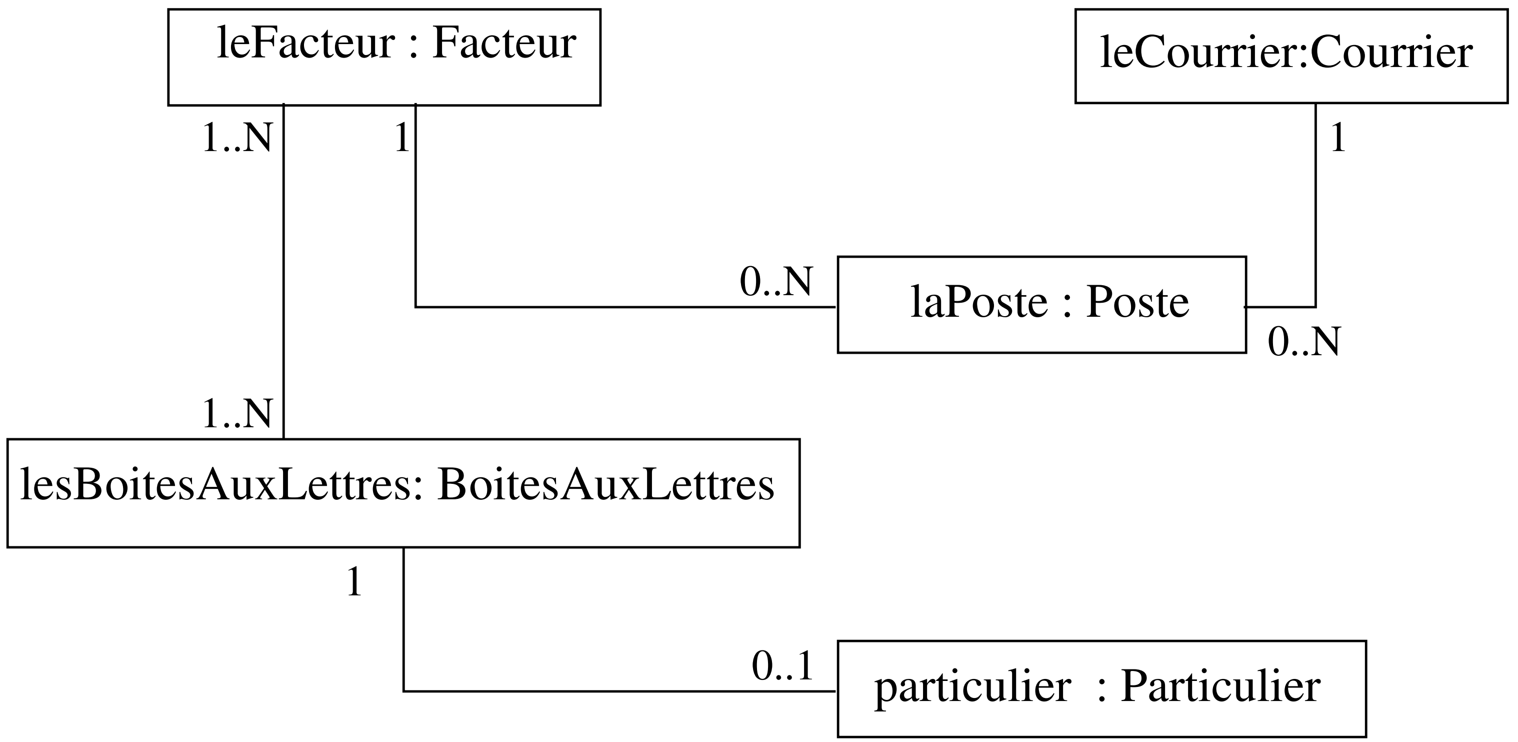
Les liens d'association doivent être portés sur une classe pas sur les champs (instances).



Arité d'une association

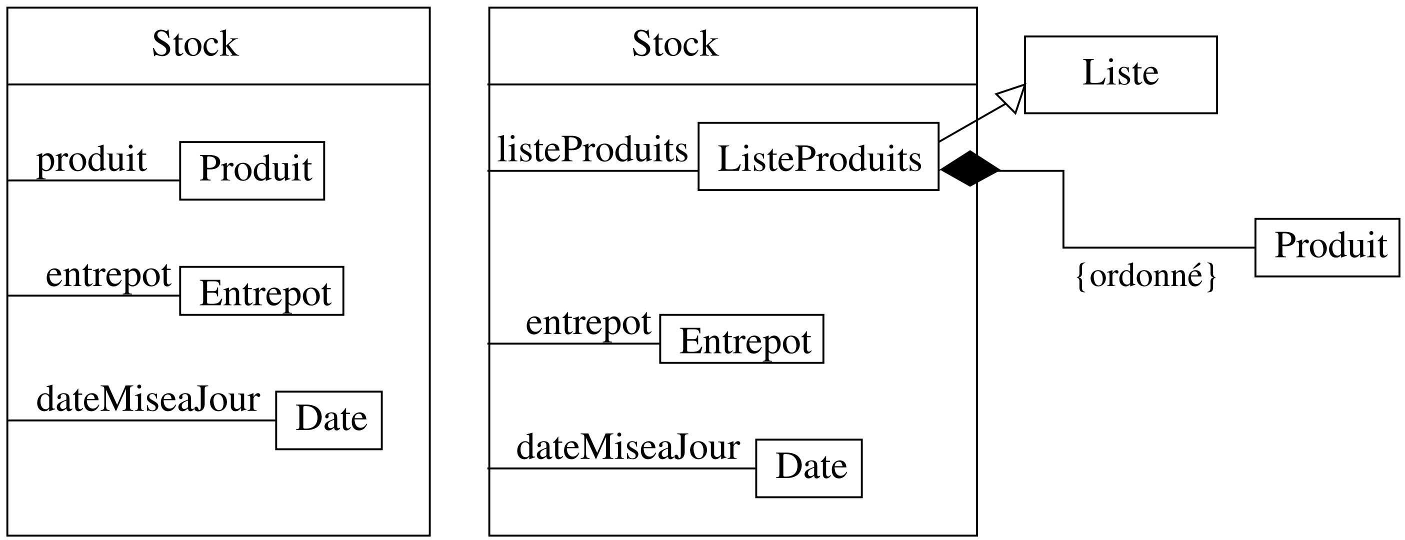
1	1 est une seule
0..1	0 ou 1 association
M..N	de M à N (M et N entiers naturels)
$\star$	de 0 à plusieurs
0..$\star$	de 0 à plusieurs
1..$\star$	de 1 à plusieurs

Marquer les arités dans le diagramme de classe



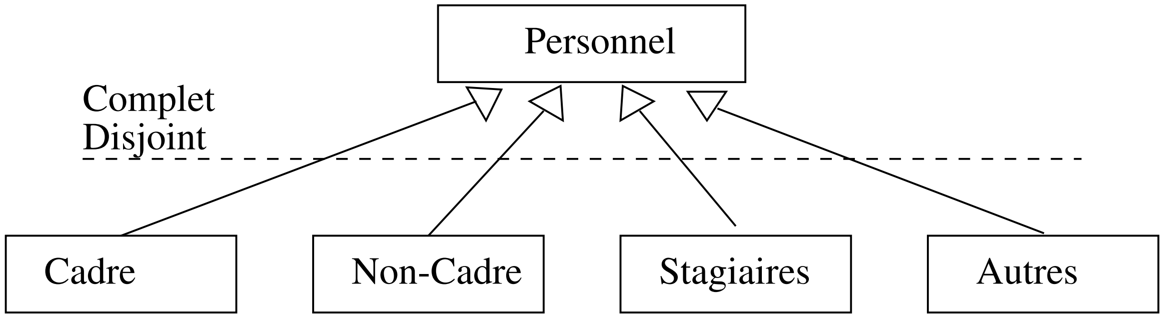
Les contraintes

Il est possible de préciser les contraintes d'une association directement sur le diagramme de classe.



Les contraintes - 2

Les liens d'héritage peuvent aussi être étiquetés par des contraintes.



Les Méta-Classes

- ➔ Définition :

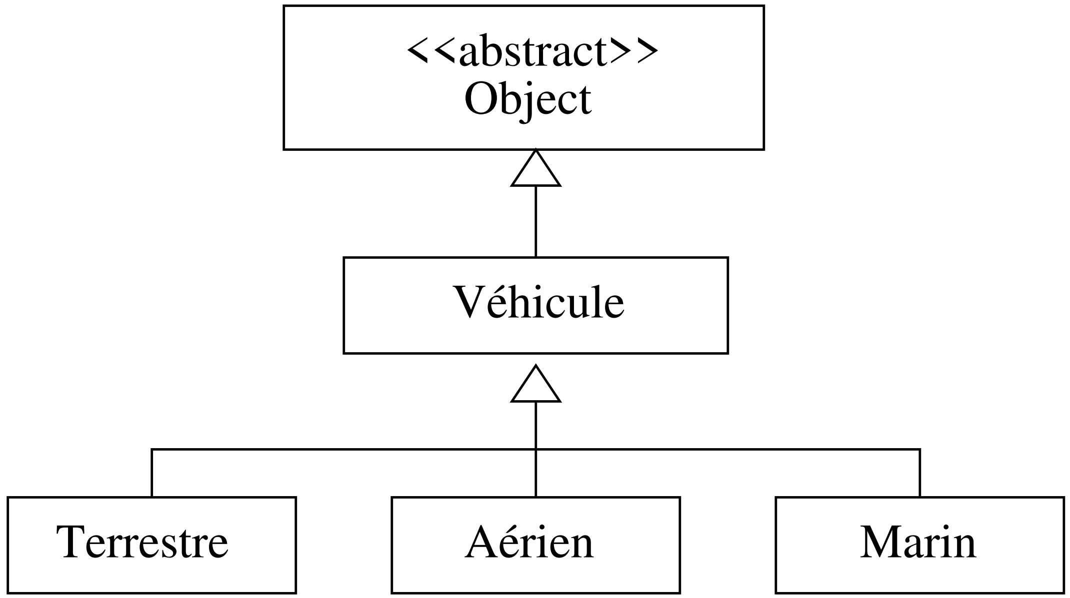
  Une méta-classe définit les caractéristiques d'une classe, c'est un modèle générique de classse (attribut ou comportement).

- ➔ Exemple :

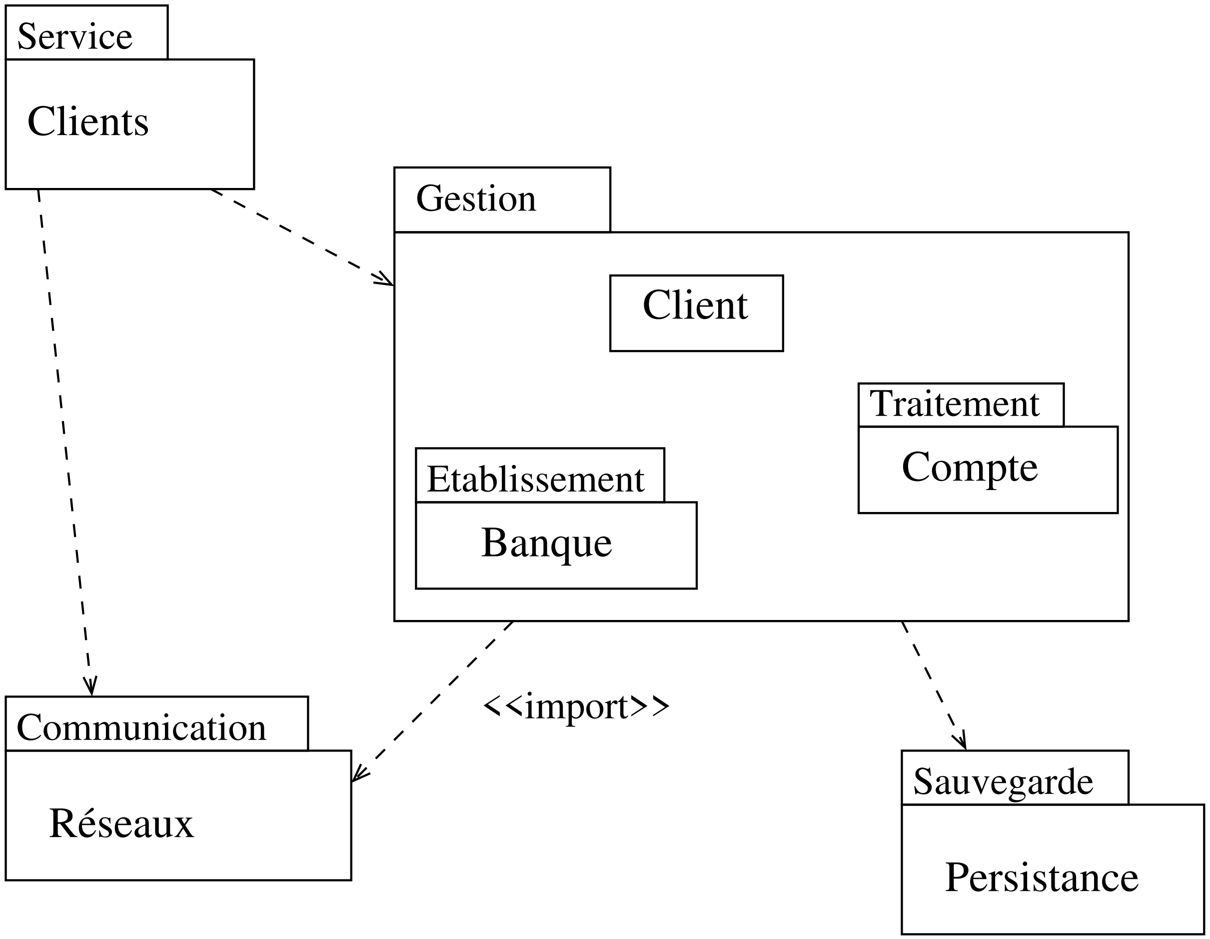
  classe abstraite, interface, par extension (abus de langage !) toute classe d'une classe.

- ➔ On notera qu'une méta-classe est également un objet dont la classe est la classe de base de référence à partir de laquelle tous les objets du système sont construits (classe Objet en Java).
- ➔ A l'étape d'analyse et de conception, il n'existe pas de différence entre une classe et une méta-classe.

Les Méta-Classes

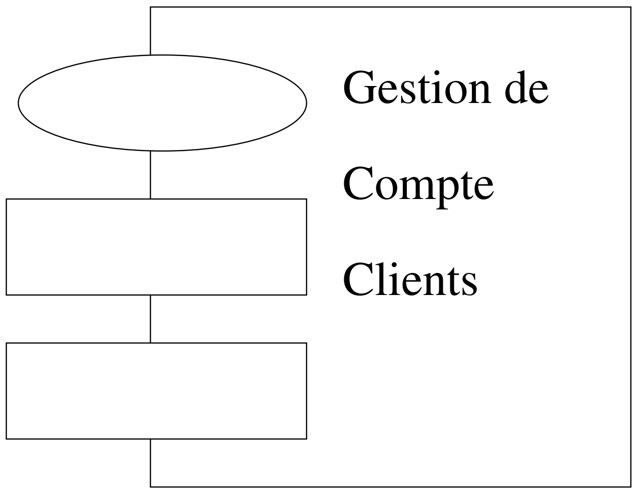


Représentation des Paquetages



Représentation de modules

Les modules sont des unités de compilation. Certains langages de programmation n'ont aucune correspondance avec ce concept.



Le diagramme de classes

- ➔ Le diagramme de classe est une **vue statique** du modèle.
- ➔ Il décrit la structure interne des classes et leurs relations (dépendances, héritage, composition . . .) .
- ➔ Les termes : *static structural diagram* et *class diagram* sont équivalents dans la terminologie UML.
- ➔ Il est une collection des éléments du modèle déclaratif. Ces éléments sont classifiés grâce au mécanisme de typage.

Le diagramme d'objets

- ➔ C'est le graphe des instances des différentes classes d'objets.
- ➔ Il est lui-même une instance du diagramme de classes.
- ➔ Il sert uniquement à illustrer des exemples.

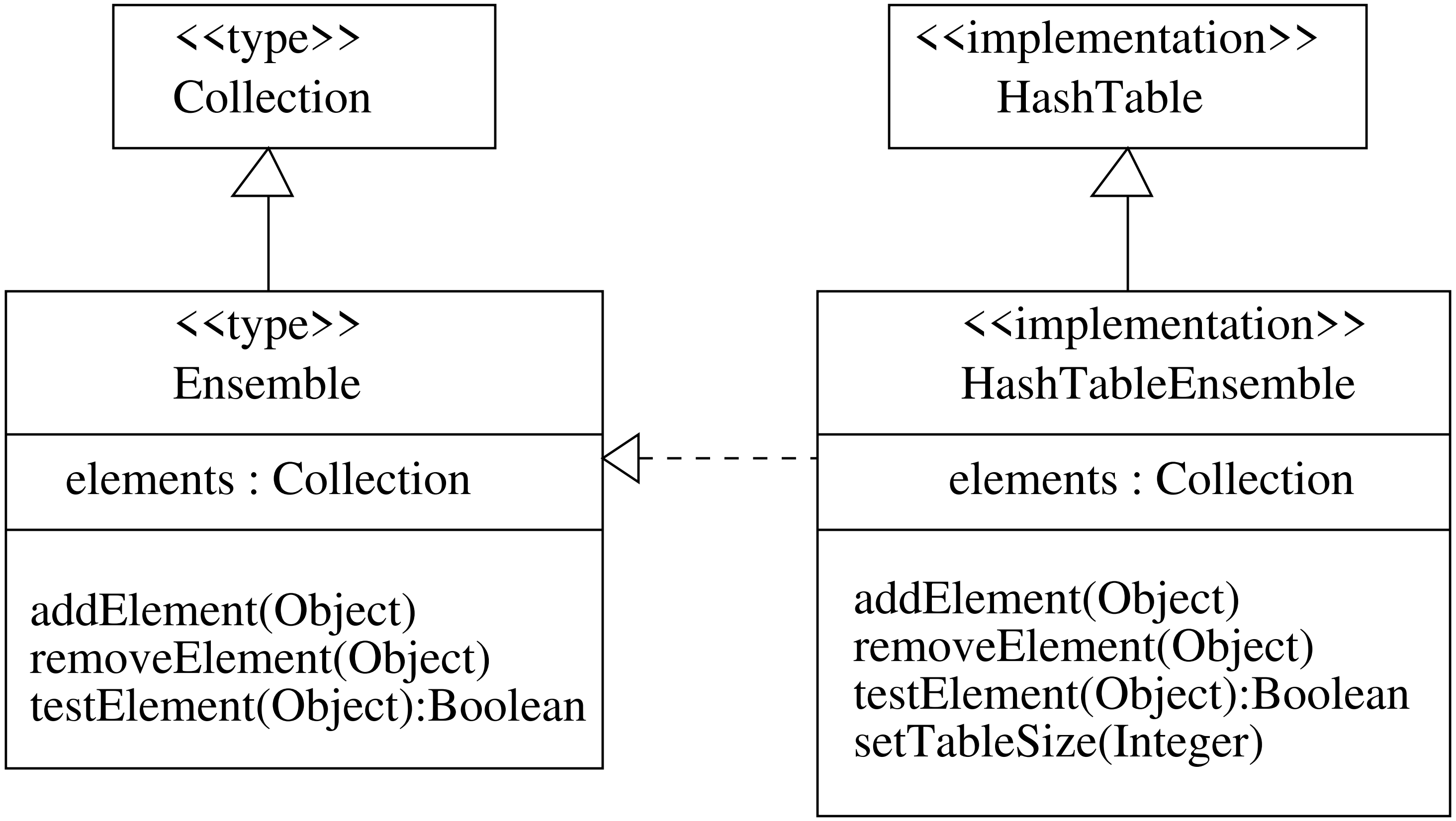
Classifier

➔ Il est également possible de construire un diagramme qui ne contient que les interfaces et les classes abstraites. On parle alors de **Méta-modèle**.

Type vs Implementation

- ➔ Une classe peut-être spécialisée par des classes d'implémentation ou de typage.
- ➔ Un **type** se caractérise par un rôle (modifiable) qu'un objet peut adopter puis abandonner.
- ➔ Une **implémentation** définit la structure de données physique et les procédures qui caractérisent un objet.
- ➔ Un objet peut avoir plusieurs types (qui peut être changé dynamiquement) mais une seule implémentation.
- ➔ NB : si leur usage est différent, leur structure interne est identique.

Type vs Implementation



Cas d'utilisation[1]

Problème :

➔ Les besoins d'un système (cf cahier des charges) sont souvent exprimés de manière non structurée, sans forte cohérence (imprécision, oublis, contradictions).

Principes de la démarche :

➔ Décrivent le comportement du système du point de vue d'un utilisateur.

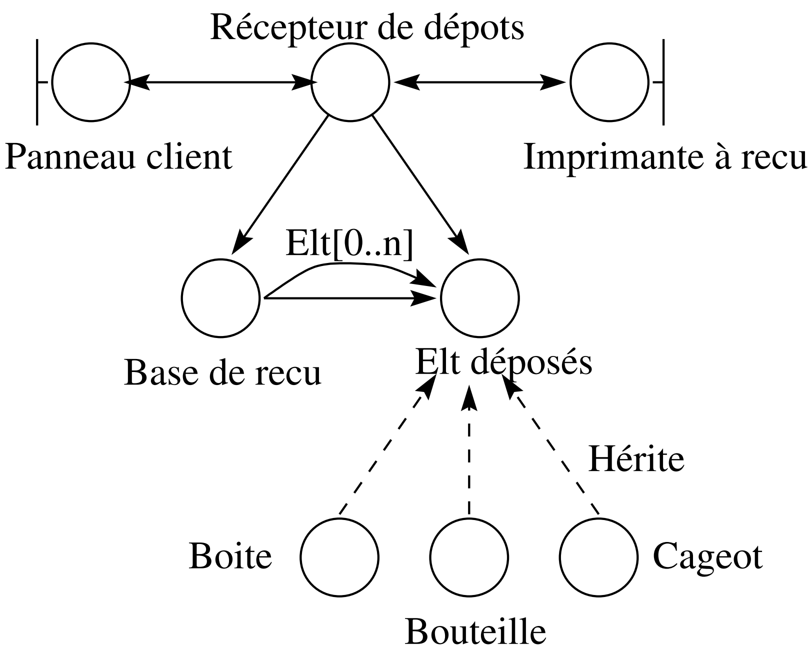
➔ Permettent de définir les limites du système et les relations entre le système et l'environnement.

➔ La structuration de la démarche s'effectue par rapport aux interactions d'**une seule catégorie d'utilisateurs** à la fois.

[1]Ont été formalisés par I. Jacobson *Object-Oriented Software Engineering, A Use Case Driven Approach*, Addison-Wesley, 1992

Le modèle des cas d'utilisation

- Le modèle des cas d'utilisation représente une partie du modèle des besoins.
- Il est issu du modèle d'analyse. Une fois les **objets identifiés** et **décrits**, on peut exprimer comment ces objets participent au cas d'utilisation.



Représentation UML

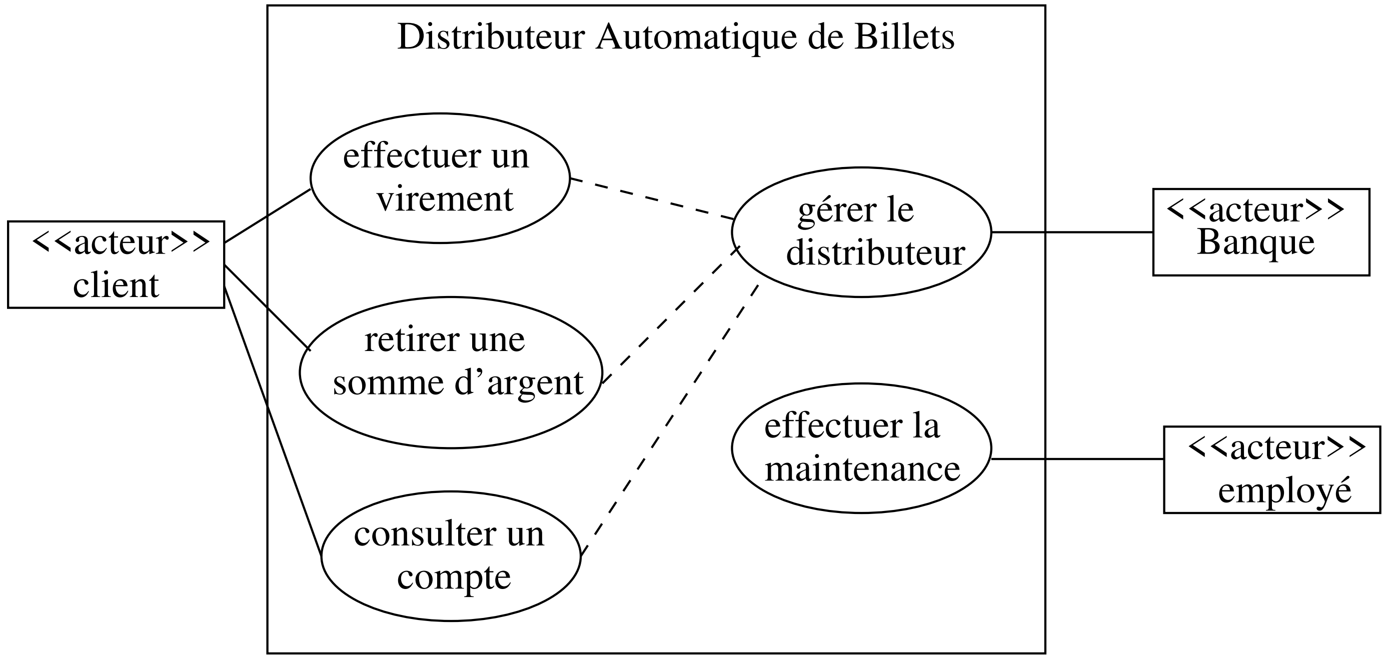
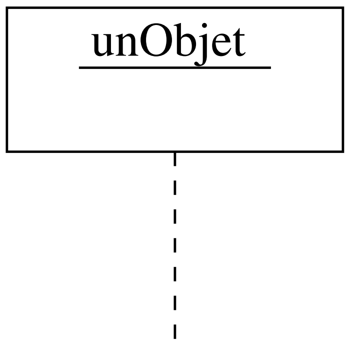


Diagramme de séquence

- ➔ Montrent des interactions entre objets selon un point de vue temporel.
- ➔ Pas de représentation explicite du contexte des objets.
- ➔ Notation[2] :

  Un objet est matérialisé par un rectangle est une barre verticale appelée ligne de vie



[2]Object Message Sequence Chart, Siemens Pattern Group. Wiley 1996 *Pattern-riented Softawre*

Diagramme de séquence - 2

➔ L'ordre d'envoi d'un message est donné par la position sur l'axe vertical.

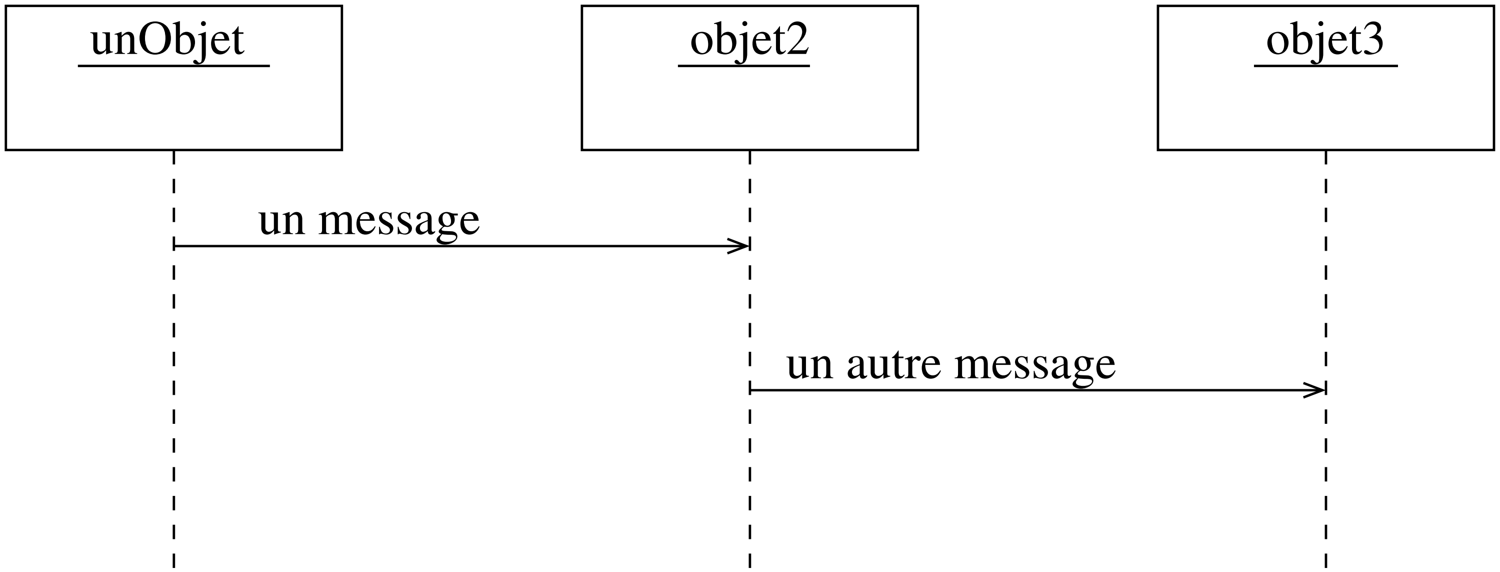


Diagramme de séquence - utilisation

Deux utilisations possibles :

➜ Documentation des cas d'utilisation : description des interactions entre objets sans détails de synchronisation. Les flèches correspondent à des **événements** qui surviennent dans le domaine de l'application. Pas de distinction entre **flots de contrôle** et **flots de données**.

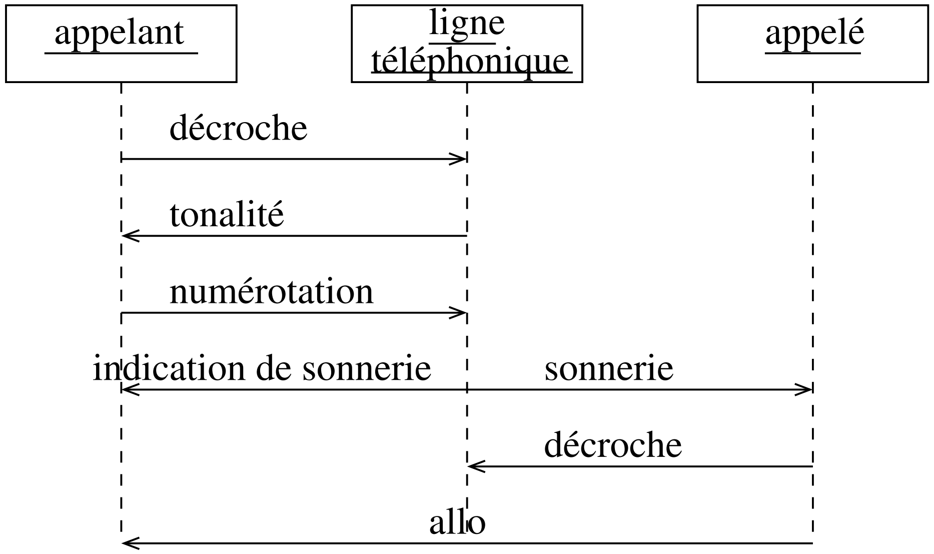
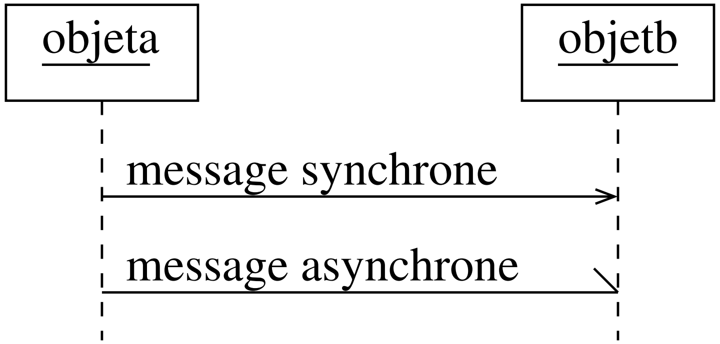
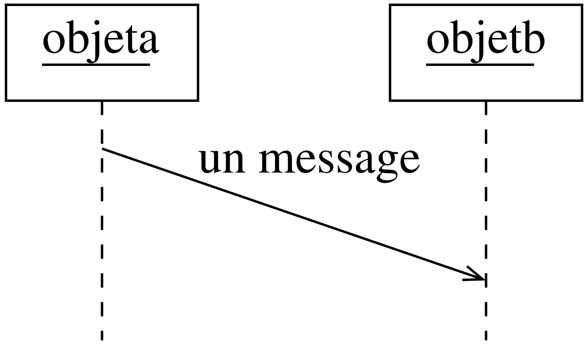


Diagramme de séquence - utilisation - 2

➜ Représentation précise des interactions entre objets.

le concept de message unifie toutes les formes de communication : appel de procédures, événemnt discret, signal de flots, interruption matérielle . . .





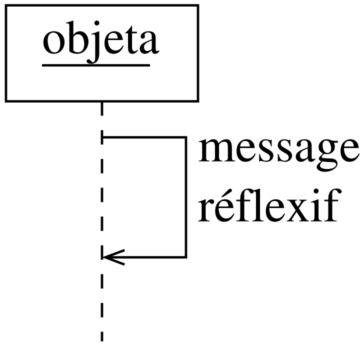


Diagramme de séquence - utilisation - 3

➔ Un message réflexif peut aussi être un point d'entrée dans une activité qui s'exerce au sein d'un objet (par exemple composite).

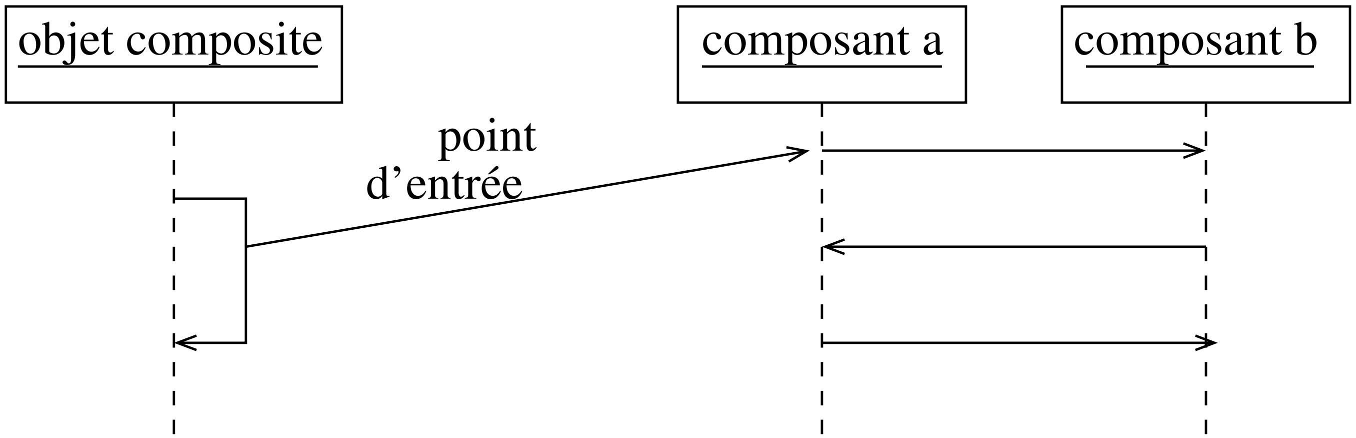


Diagramme de séquence - utilisation - 4

➜ Création et destruction d'un objet.

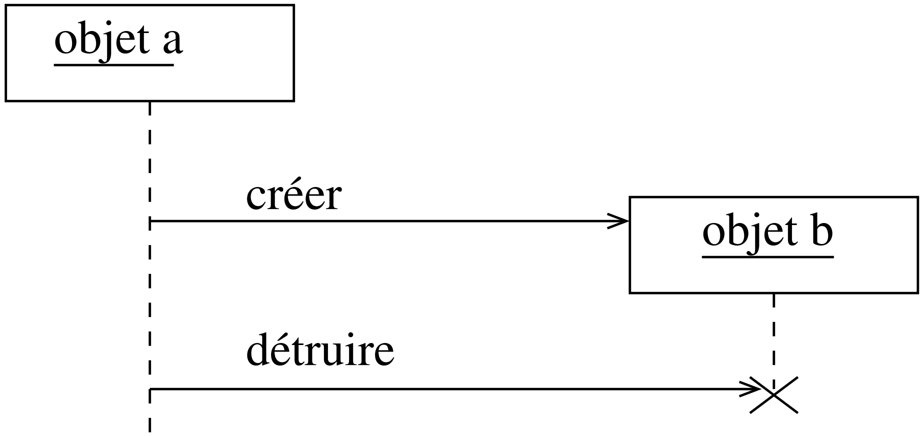


Diagramme de séquence - utilisation - 5

- ➔ Représentation des périodes d'activité des objets = temps pendant lequel un objet effectue une action.
- ➔ le début et la fin de la bande rectangulaire correspondent au début et à la fin d'une période d'activité.

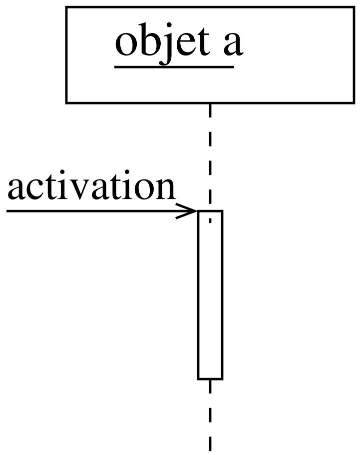


Diagramme de séquence - utilisation - 6

➔ Les digrammes de séquence permettent également de représenter les périodes d'activité des objets.

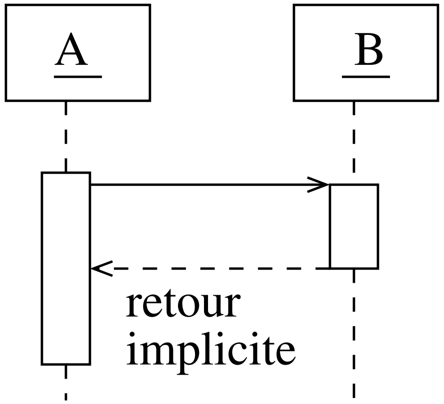


Diagramme de séquence - utilisation - 7

➔ Dans le cas d'envoi asynchrone, le retour doit être signalé.

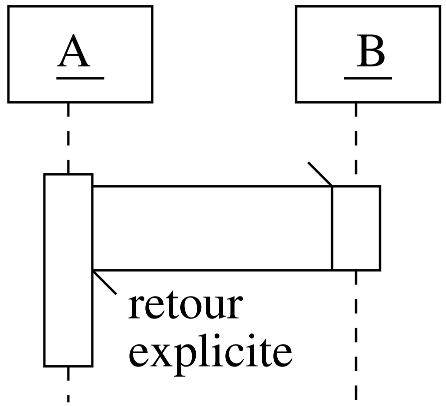
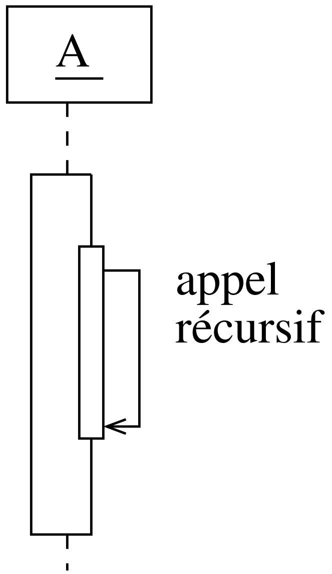


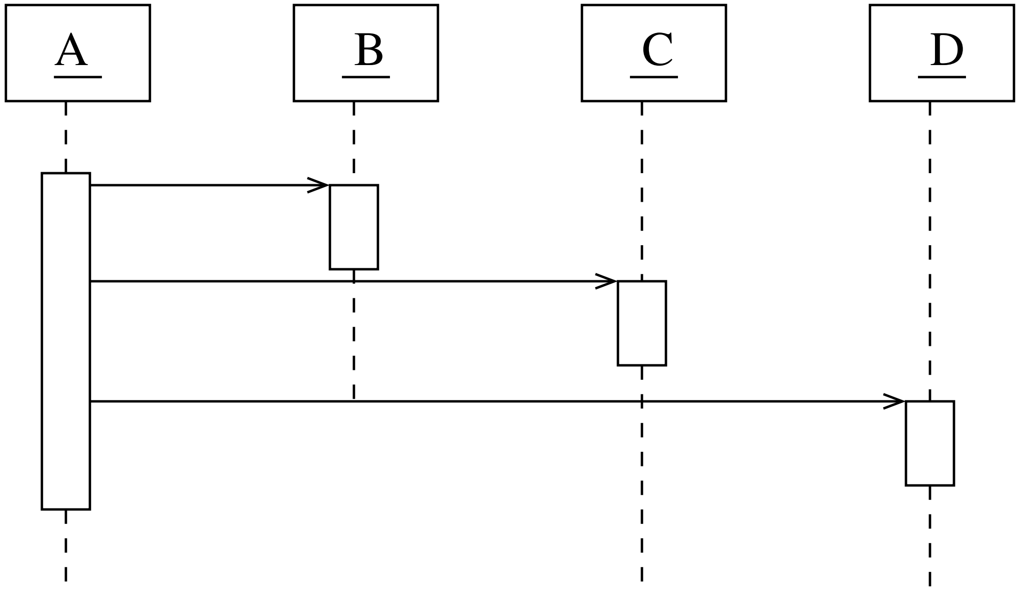
Diagramme de séquence - utilisation - 8

- Cas des messages récursifs



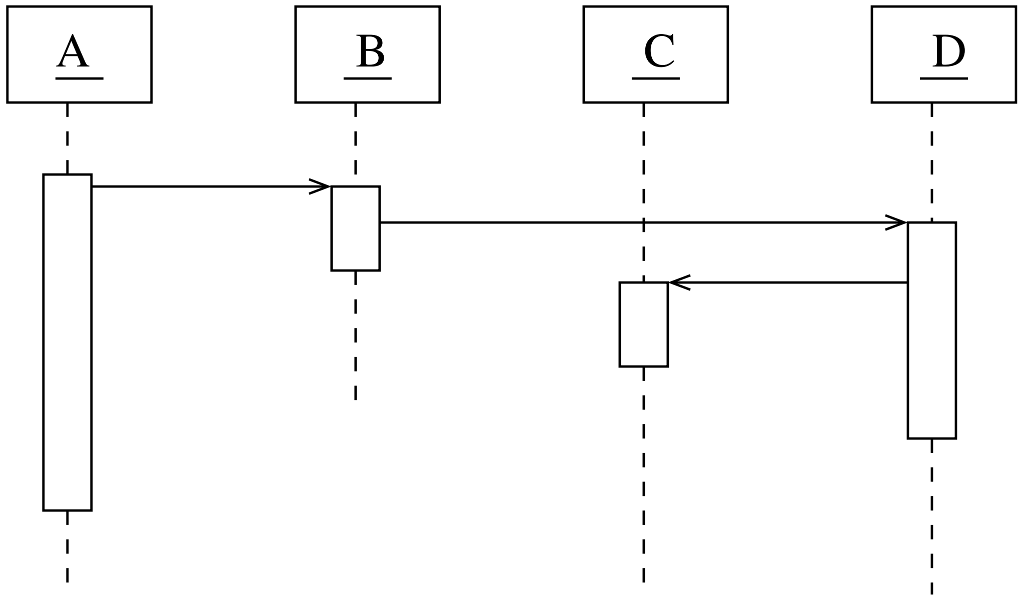
Mode centralisé - Mode décentralisé

➔ Les diagrammes de séquences reflètent le choix des structures de controle.

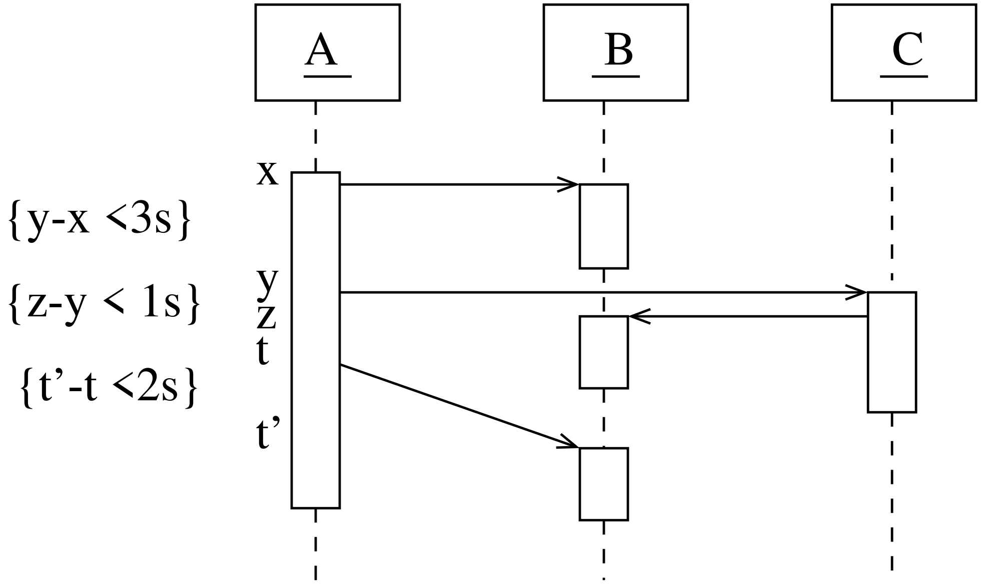


Mode centralisé - Mode décentralisé

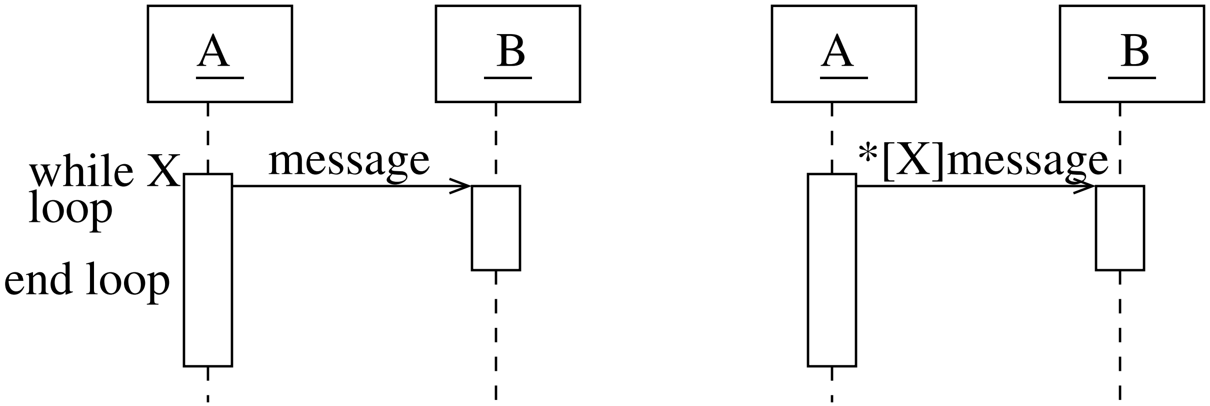
➜ Envoi décentralisé de messages.



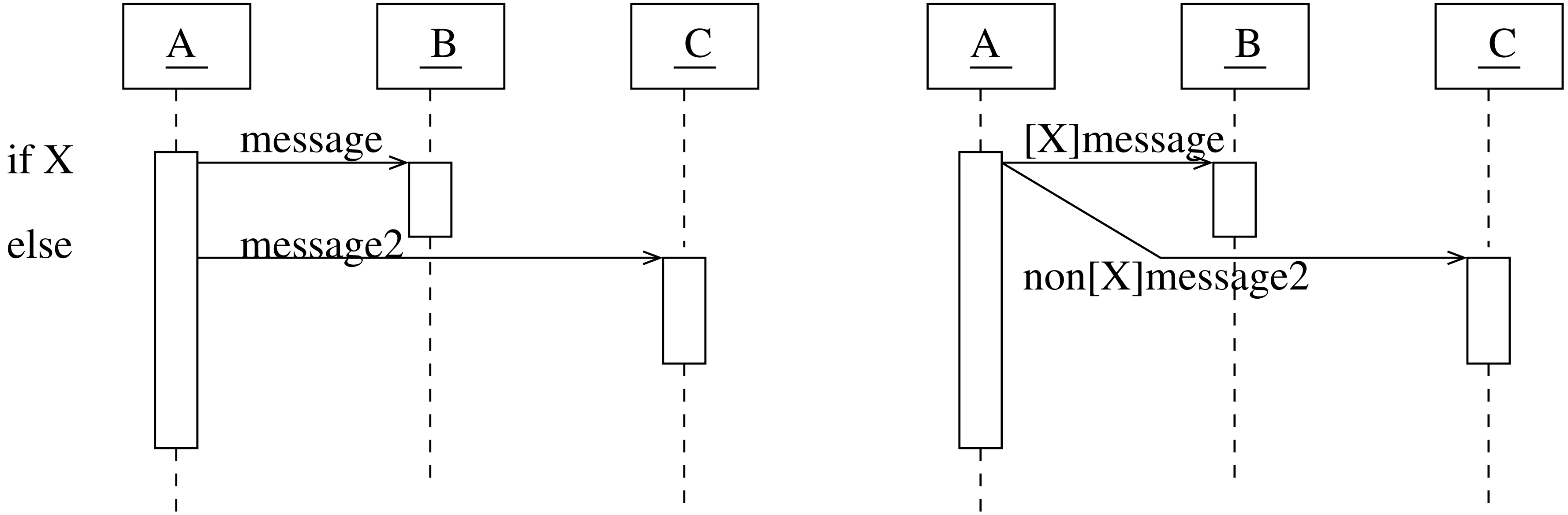
Expression des contraintes temporelles



Boucles et branchements



Branchement Conditionnel



Les diagrammes de collaboration

- Rôle :

  Ils montrent les interactions entre les objets.

  Ils expriment le contexte d'un groupe d'objets.

  Ils sont des extensions du diagrammes d'objets.

- Une interaction est réalisée par un groupe d'objets qui collaborent en échangeant des messages.
- Ces messages sont représentés le long des liens qui relient les objets avec des flêches orientées vers le destinataire du message.

Diagramme de collaboration

La numérotation donne l'ordre d'envoi des messages. Le temps n'est pas représenté.

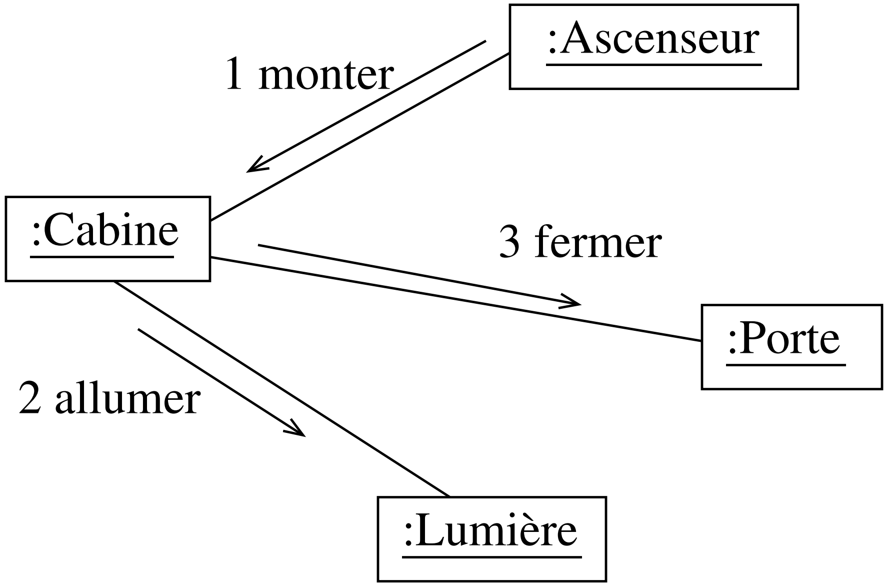
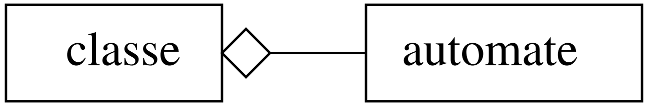


Diagramme de collaboration - 2

- ➔ Le diagramme de collaboration montre simultanément les interactions entre les objets et les relations structurelles qui permettent ces interactions.
- ➔ Ne pas confondre ces liens avec ceux de composition des diagrammes de classes ou d'objets.

Les diagrammes d'états-transitions

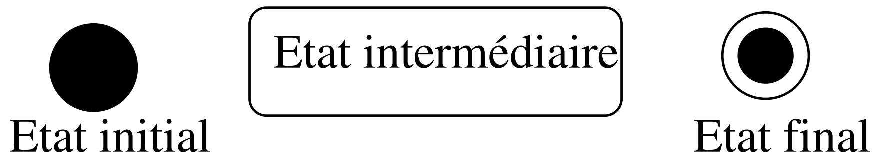
- Visualisent des automates déterministes[3].
- On relie l'automate à la classe considérée. On ne représente pas les automates des objets qui ne changent pas (ou peu) d'état.



[3]formalisme de Harel, D. 1987. *Statecharts : a Visual Formalism for Complex Systems*. Science of Computer Programming vol 8.

Les diagrammes d'états-transitions - 2

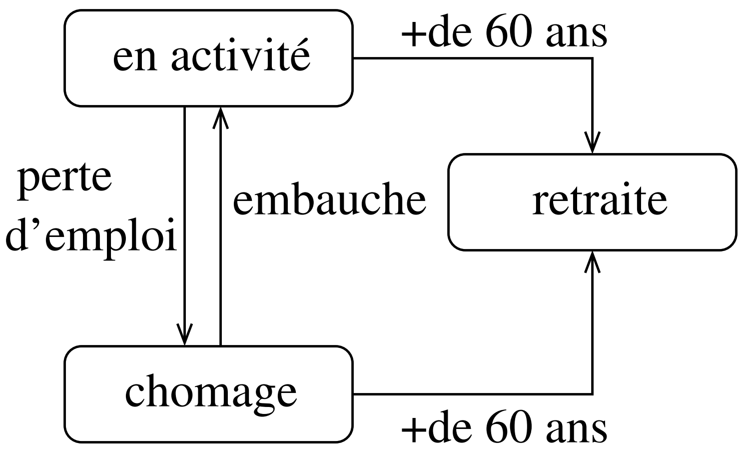
- Un objet est à tout moment dans un état donné.
- L'état d'un objet est constitué des valeurs instantanées de ses attributs.



Les diagrammes d'états-transitions - 3

➔ L'objet passe d'un état à un autre par les transitions.

➔ Déclenché par un événement, les transitions permettent le passage d'un état à un autre instantanément.



Les événements

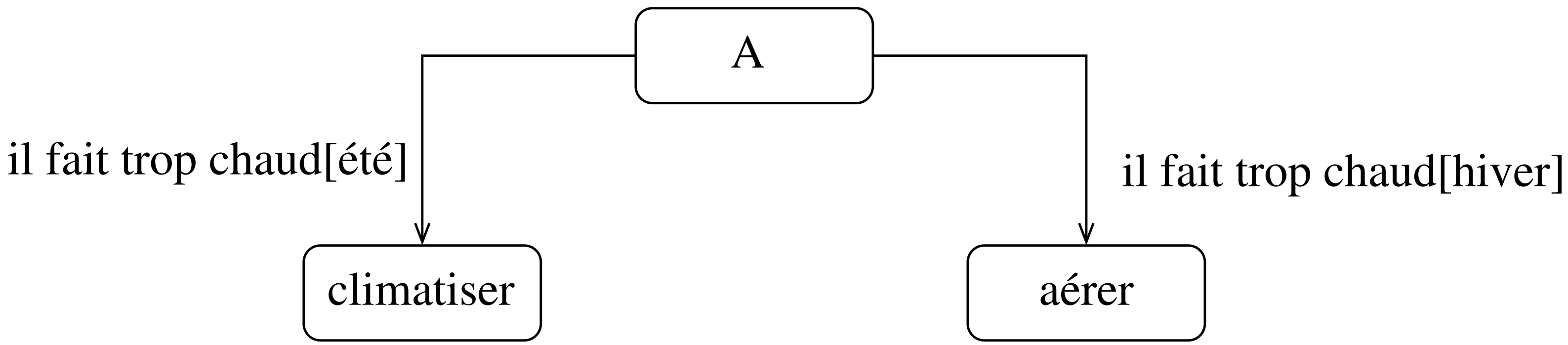
- La syntaxe d'un événement dans un diagramme est la suivante :

  Nom_événement (Nom_paramètre : type, ...)[condition]

  condition est la garde qui valide ou non le déclenchement d'une transition quand l'événement s'est produit.

- On peut associer à chaque transition une **action** à exécuter lors du franchissement dû à un événement. Les spécifications de l'action sont contenues dans l'objet destinataire.

- **Exemple** : l'événement `il fait trop chaud` entraîne la climatisation ou l'ouverture des fenêtres selon la saison.

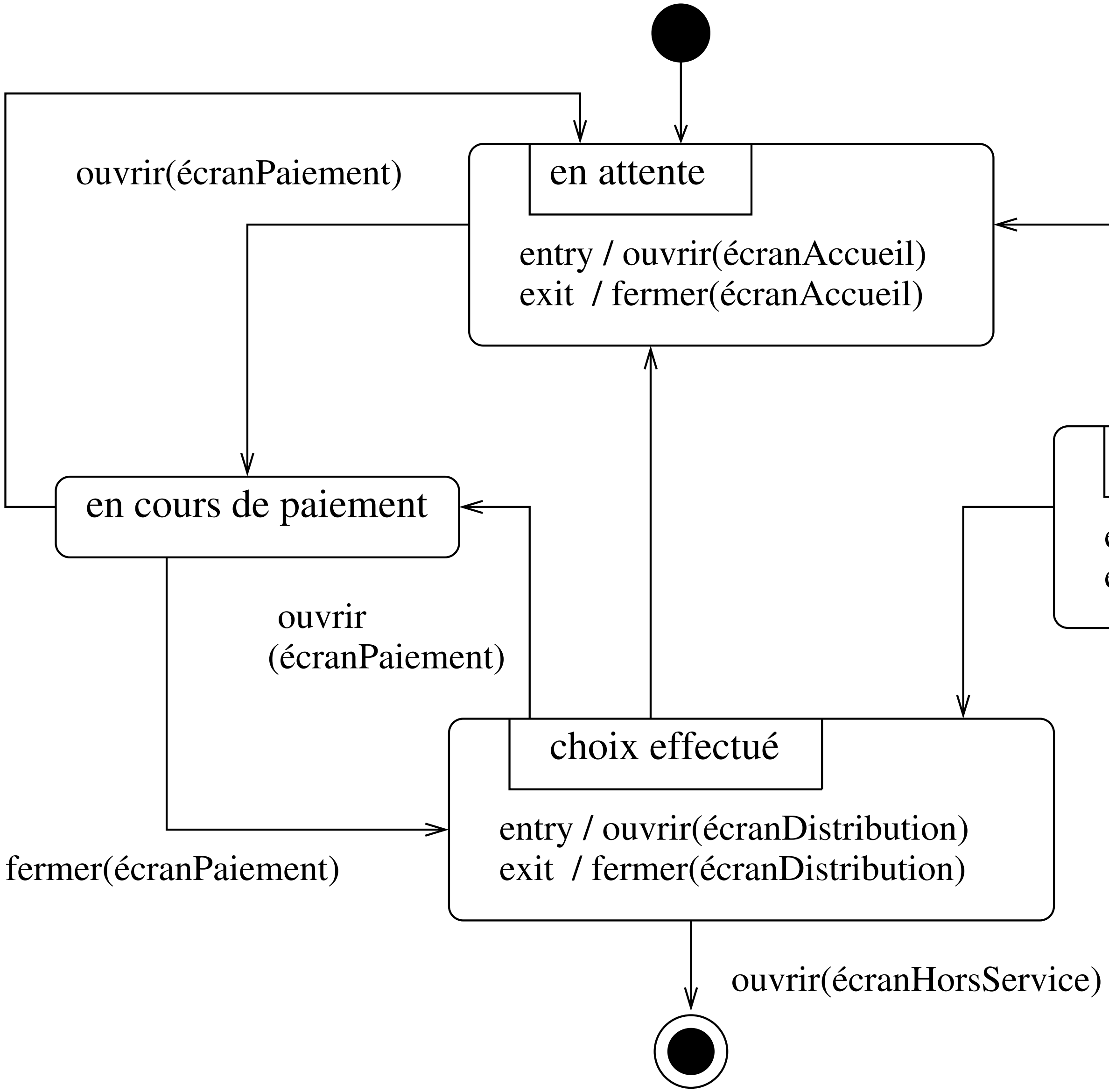


Les événements

➔ Il est possible de préciser les actions à exécuter lorsque l'on est dans un état donné, en entrant ou en sortant.
Pour cela UML donne plusieurs mots-clés :

- **entry** : action à exécuter dès l'entrée dans l'état.
- **exit** : action à exécuter lors de la sortie de l'état.
- **on** : action interne provoquée par un événement qui ne provoque pas le passage à un nouvel état.
- **do** :activité à exécuter (une activité est une action dont le temps d'exécution est non négligeable).

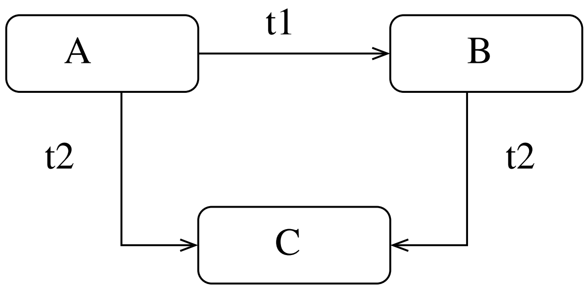
Exemple d'un diagramme d´états

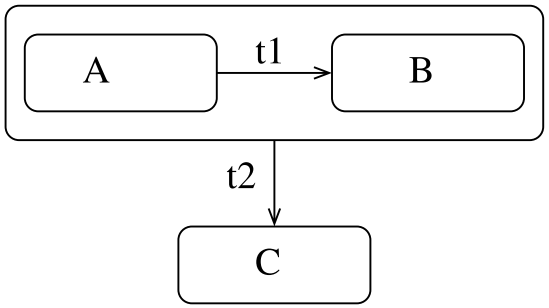


Généralisation d'états

➔ Pour remédier au problème de l'explosion du nombre des états et de leur connexions, il est possible de définir des **super-classes** d'états et des **sous-classes** qui en héritent. La démarche d'abstraction est identique à la **généralisation/spécification** des classes.

➔ Un état peut-être décomposé en plusieurs **sous-états disjoints** (ou-exclusif), un objet ne peut-être que dans un seul état à la fois.

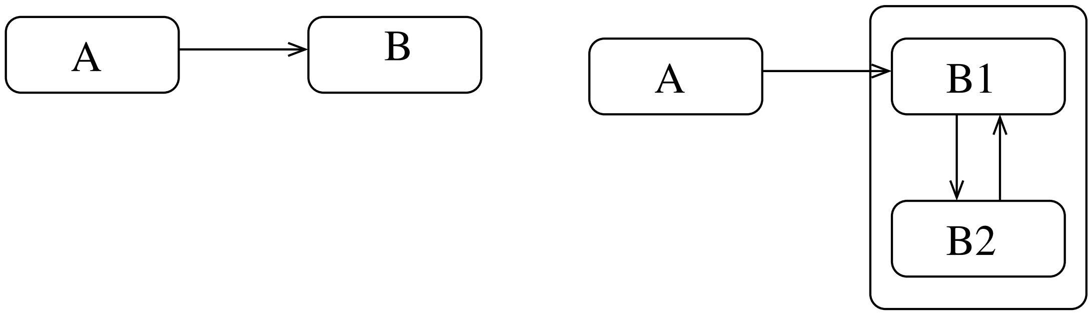




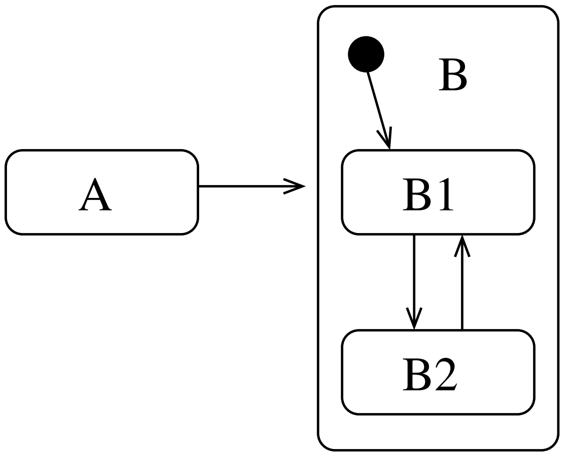
Généralisation d'états - 2

- Un sous-état hérite des variables d'états et de transitions externes de sa super-classe.
- Un seul état (le super-état ou un des sous-états) hérite des transitions d'entrée car un seul état peut-être la cible d'une transition.
- Si la décomposition a pour objectif de définir un état particulier pour le traitement d'une transition interne, cette transition ne fera pas l'objet d'un héritage.

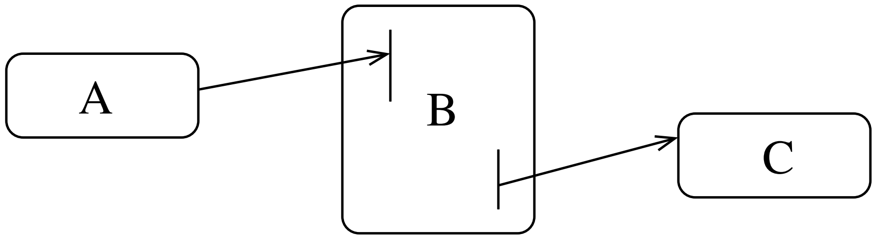
Cas d'une décomposition qui rompt la barrière d'asbtraction :



Solution

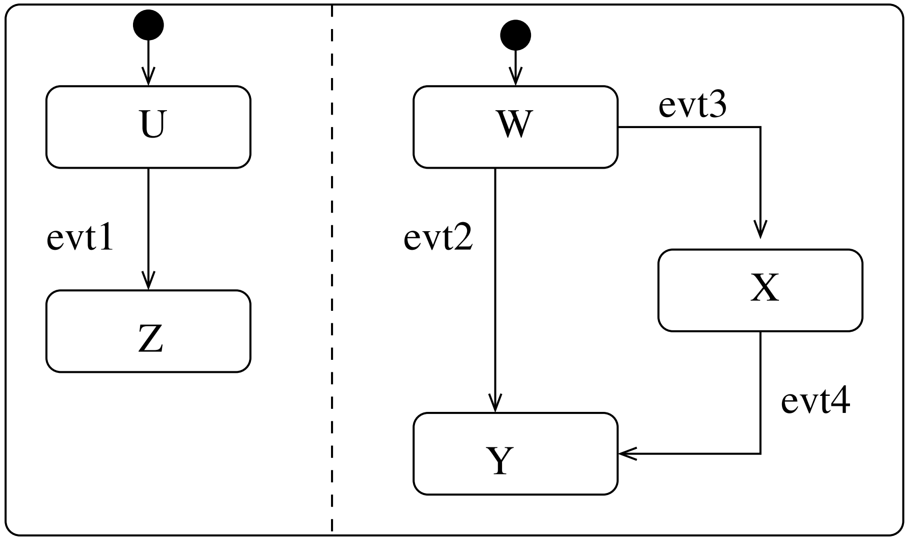


Représentation simplifiée

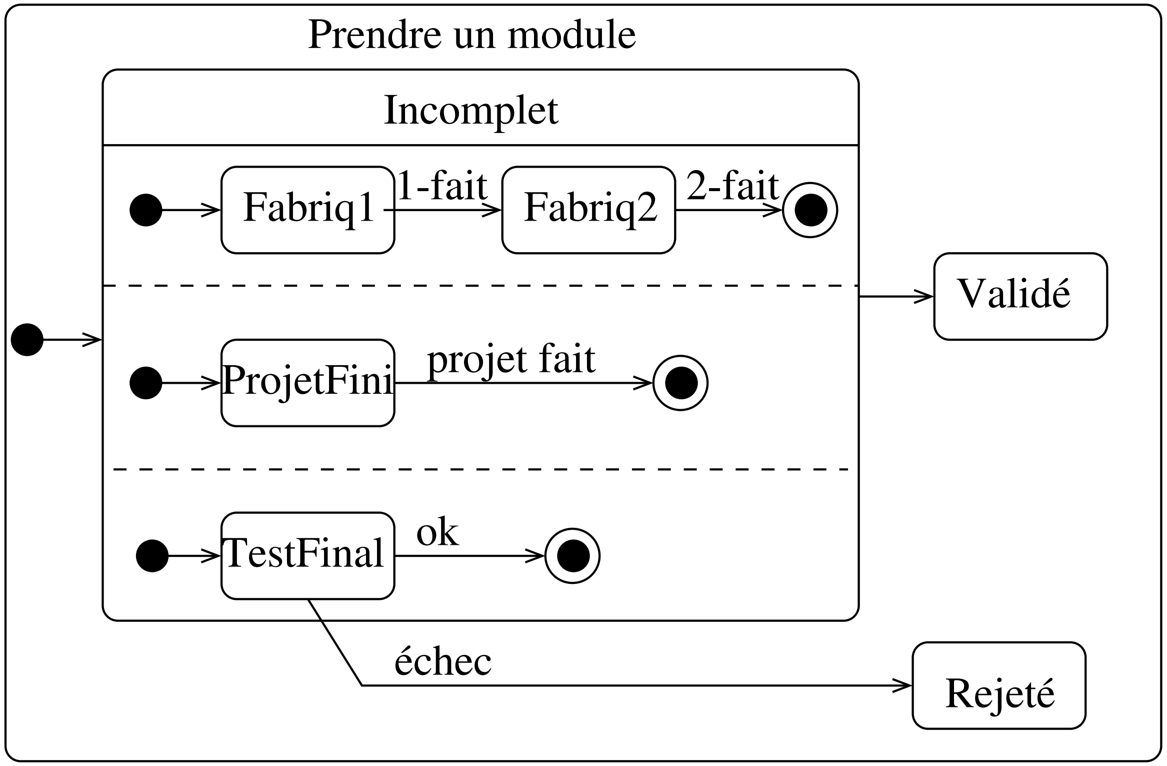


Agrégation d'états

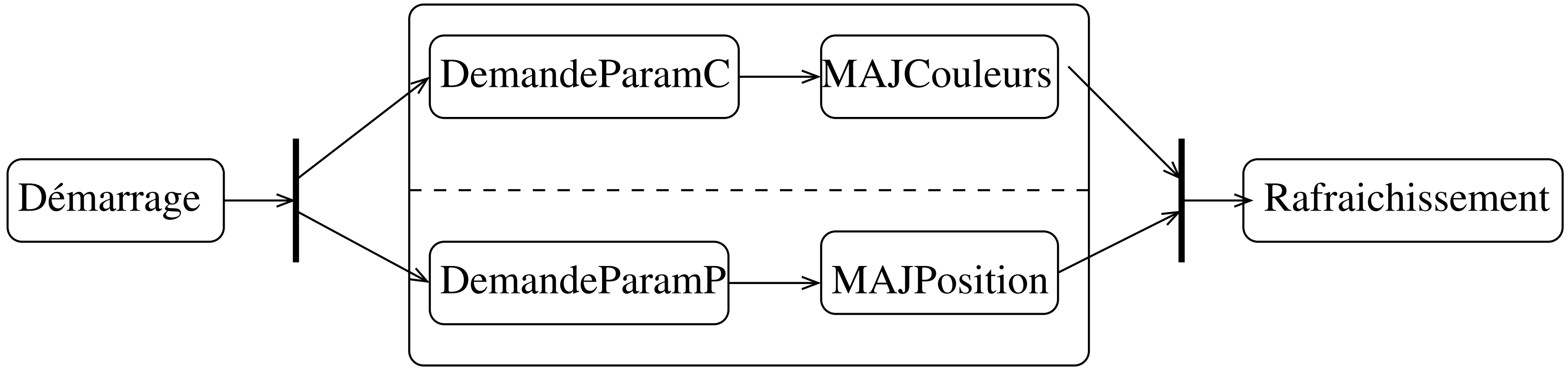
➔ L'agrégation d'états est un état composé de plusieurs automates qui évoluent simultanément et indépendamment.



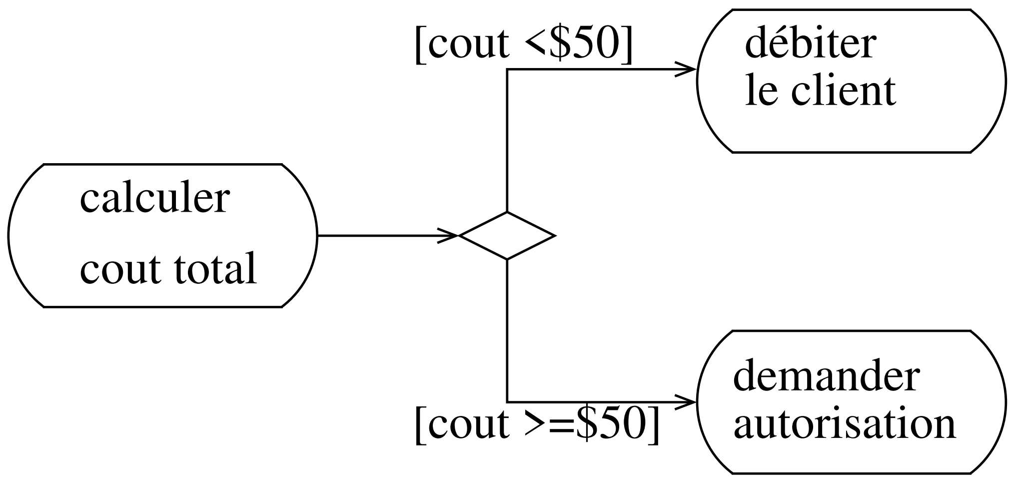
Agrégation d'états - 2



Transitions complexes

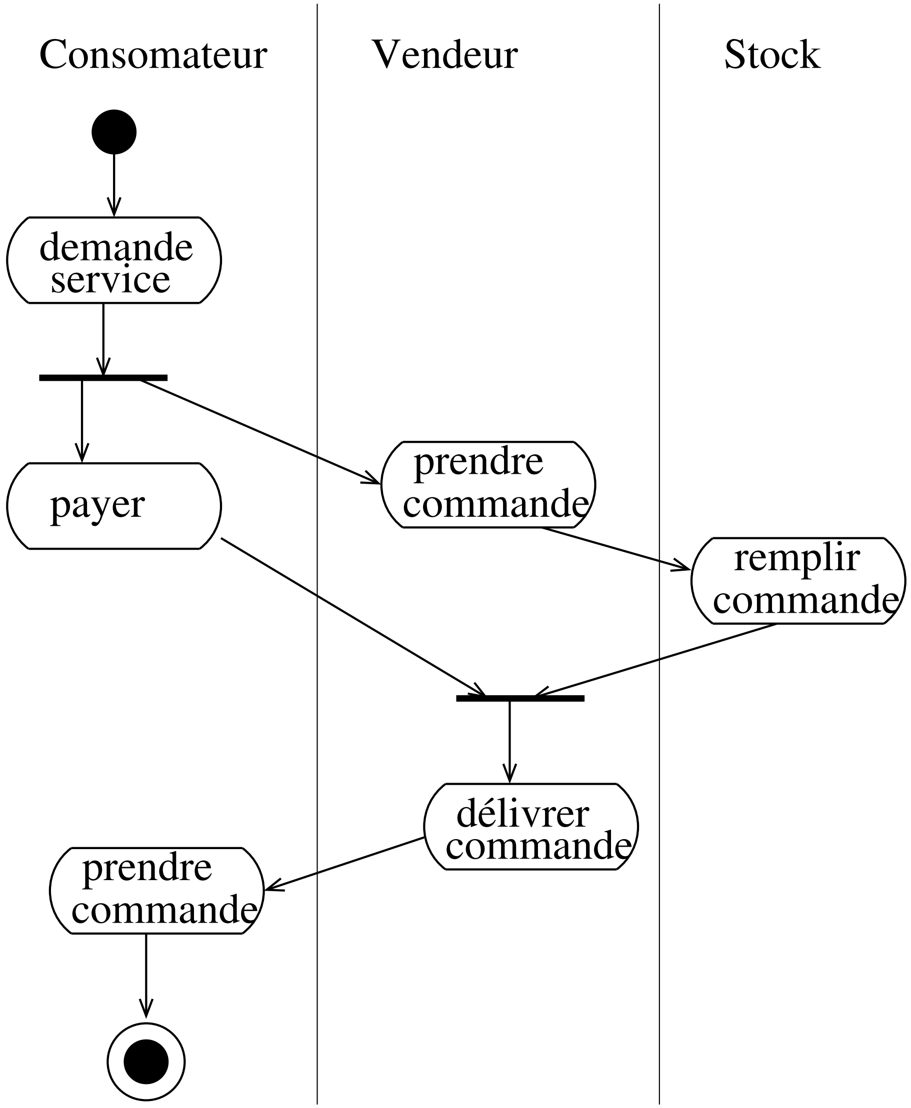


Décisions



[cout <$50]
débiter
le client
calculer
cout total
demander
autorisation
[cout >=$50]

Diagrammes d’activités


Consomateur
Vendeur
Stock
demande service
prendre commande
payer
remplir commande
délivrer commande
prendre commande

Action et Objets