

Introduction à UML 2

Eric Cariou

Master Technologies de l'Internet 1^{ère} année

*Université de Pau et des Pays de l'Adour
UFR Sciences Pau – Département Informatique*

Eric.Cariou@univ-pau.fr

- ◆ Ce cours est basé initialement sur un cours de Laurence Duchien : <http://www.lifl.fr/~duchien/>

Introduction

- ◆ UML : *Unified Modeling Language*
- ◆ Normalisé par l'OMG (Object Management Group)
 - ◆ <http://www.omg.org/spec/UML/>
 - ◆ Dernière version : 2.4.1 (Août 2011)
- ◆ Notation standard pour la modélisation d'applications à base d'objets (et de composants depuis la version 2)
 - ◆ Mais utilisable dans de nombreux autres contextes de conception ou spécification
 - ◆ Exemple : schéma de BDD
- ◆ Langage utilisant une notation graphique

Modèles

- ◆ Un modèle est une représentation partielle de la réalité
 - ◆ Abstraction de ce qui est intéressant pour un contexte donné
 - ◆ Vue subjective et simplifiée d'un système
 - ◆ Avec UML, on va s'intéresser principalement aux modèles d'applications informatiques
 - ◆ Un modèle UML = des diagrammes UML
- ◆ Utilité des modèles
 - ◆ Faciliter la compréhension d'un système
 - ◆ Permettre également la communication avec le client
 - ◆ Vision de communication, de documentation
 - ◆ Définir voire simuler le fonctionnement d'un système
 - ◆ Dans ce cas, on se doit d'être le plus précis possible dans le contenu des modèles pour s'approcher du code
 - ◆ Vision de développement, de production

Historique

- ◆ UML hérite principalement des méthodes objets de Booch (Booch), OMT (Rumbaugh) et OOSE (Jacobson)
 - ◆ Mais intègre également d'autres approches, comme les machines à états de Harel
- ◆ But initial
 - ◆ Définir un processus/méthode de développement complet (de l'analyse à l'implémentation) orienté objet
- ◆ Problème
 - ◆ Pas de notation, langage pour écrire les modèles ou les artefacts définis par ce processus \Rightarrow devenu le but final d'UML
- ◆ UML n'est donc pas une méthode ou un processus
 - ◆ UML propose un ensemble de notations pour que chacun ait à sa disposition les éléments nécessaires à la conception d'une application

UML ≠ processus de développement

- ◆ UML indépendant du processus de conception et de développement : ne décrit pas comment il fonctionne
- ◆ Exemple de processus de conception et de développement
 - ◆ Processus itératif et incrémental
 - ◆ Définition du cahier des charges
 - ◆ Elaboration du logiciel : cycle de vie à itérer
 1. Analyse
 2. Spécification
 3. Implémentation
 4. Test
 - ◆ Chaque itération permet l'ajout de fonctionnalités en les définissant, les réalisant, les testant et les intégrant
 - ◆ Arrêt du processus itératif lorsque le logiciel produit répond complètement au cahier des charges

UML ≠ processus de développement

- ◆ UML fournit une notation/syntaxe pour les diagrammes et modèles définis pendant tout le cycle de développement
- ◆ UML permet de définir des modèles de niveaux différents
 - ◆ Analyse
 - ◆ Conception
 - ◆ Spécification d'implémentation
 - ◆
- ◆ Il faut préciser à quel niveau correspond un modèle
- ◆ On peut raffiner un modèle pour le spécifier à chaque niveau

Les diagrammes UML

- ◆ 13 diagrammes différents
 - ◆ Diagrammes structurels
 - ◆ De classes (class diagram)
 - ◆ D'objets (object diagram)
 - ◆ De composants (component diagram)
 - ◆ De structure composite (composite structure diagram)
 - ◆ De déploiement (deployment diagram)
 - ◆ De paquetages (package diagram)
 - ◆ Diagrammes de comportement
 - ◆ De cas d'utilisation (use case diagram)
 - ◆ D'activité (activity diagram)
 - ◆ D'états-transition (state diagram)
 - ◆ Diagrammes d'interaction
 - ◆ De séquence (sequence diagram)
 - ◆ Vue générale d'interaction (interaction overview diagram)
 - ◆ De communication (communication diagram)
 - ◆ De temps (timing diagram)

Les diagrammes UML

- ◆ Ces diagrammes permettent de définir une application selon plusieurs points de vue
 - ◆ Fonctionnel (cas d'utilisation)
 - ◆ Statique (classes, objets, structure composite)
 - ◆ Dynamique (séquence, états, activité, interaction, communication, temps)
 - ◆ Implémentation (composants, déploiement, paquetage)
- ◆ Les diagrammes seuls ne permettent pas de définir toutes les contraintes de spécification requises
 - ◆ Utilisation du langage textuel de contraintes OCL en complément
 - ◆ S'applique sur les éléments de la plupart des diagrammes

Plan

- ◆ *Diagrammes fonctionnels*
 - ◆ *Cas d'utilisation*
- ◆ Diagrammes statiques
- ◆ Diagrammes dynamiques
- ◆ Diagrammes d'implémentation

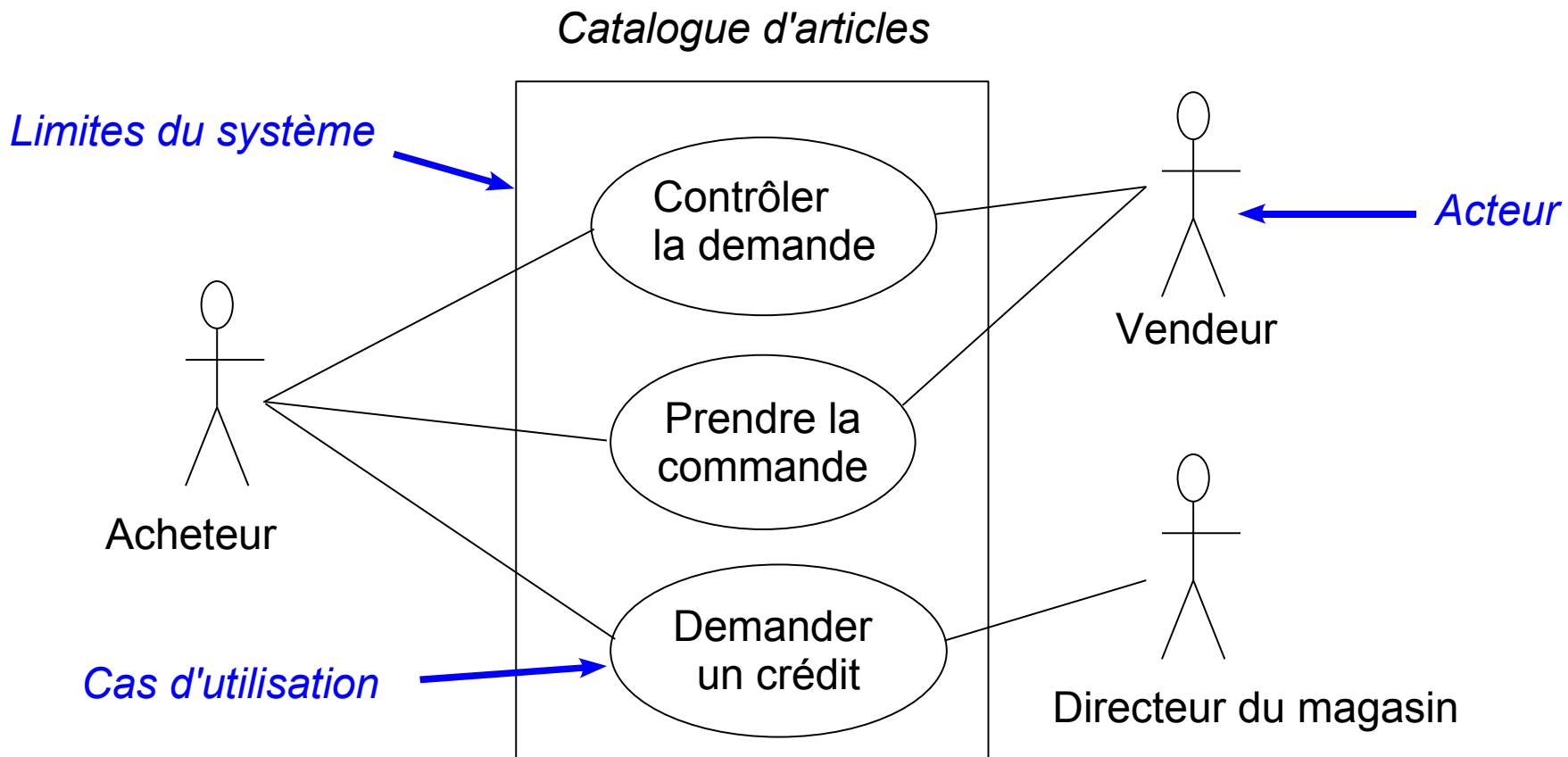
Diagramme de cas d'utilisation

- ◆ Description des interactions type entre un utilisateur et le système informatique
- ◆ Définition des cas d'utilisation à partir de discussions avec l'utilisateur sur ses attendus du système
- ◆ Enumération des principaux scénarios prévus
- ◆ Exemple : écriture d'un texte avec un traitement de texte
 - ◆ 2 cas d'utilisation : mettre du texte en gras, créer un index
- ◆ Propriétés des cas d'utilisation
 - ◆ Déterminer les fonctions visibles pour un utilisateur
 - ◆ Prendre en compte les objectifs des utilisateurs
 - ◆ De taille quelconque

Diagramme de cas d'utilisation

- ◆ Deux grandes approches
 - ◆ Objectif de l'utilisateur
 - ◆ Interaction du système
- ◆ Exemple : utilisation d'une feuille de style dans un traitement de texte
 - ◆ Interaction du système : définir un style, changer de style, déplacer un style d'un document vers un autre
 - ◆ Objectif de l'utilisateur : assurer un formatage cohérent, faire un format de document identique à un autre
 - ◆ Les interactions du système reflètent ce que l'utilisateur peut faire plus que le but réel de l'application
- ◆ Description d'un cas d'utilisation : de manière informelle, généralement en langage naturel

Diagramme de cas d'utilisation



Différents liens entre les cas d'utilisation/acteurs :

———— association

- - - - - > généralise

<< extend >> étend
- - - - - >

<< include >> inclut
- - - - - >

Plan

- ◆ Diagrammes fonctionnels
- ◆ *Diagrammes statiques*
 - ◆ *De classes*
 - ◆ *D'objets*
 - ◆ *De composants*
 - ◆ *De structure composite*
- ◆ Diagrammes dynamiques
- ◆ Diagrammes d'implémentation

Diagramme de classes

- ◆ Définition des éléments formant une application et de leurs relations
- ◆ Structuration statique de l'application
 - ◆ Définition des classes existantes
 - ◆ Définition de la structure interne des classes (attributs, opérations)
 - ◆ Définition des relations entre les classes
- ◆ 2 principaux types de relations entre classes
 - ◆ Association
 - ◆ Un client peut louer un certain nombre de vidéos
 - ◆ Sous-typage/généralisation
 - ◆ Un étudiant est une personne
- ◆ Important : documenter les diagrammes de classes

Diagramme de classes

- ◆ Exemple de 3 usages possibles d'un diagramme de classes
 - ◆ Diagramme conceptuel
 - ◆ Concepts métier du domaine étudié à un niveau abstrait
 - ◆ Sans lien avec l'implémentation
 - ◆ Diagramme de spécification
 - ◆ Première approche du logiciel par la définition de ses interfaces
 - ◆ Interface = type de l'objet, classe = implémentation de l'objet
 - ◆ Un type (ou interface) peut avoir plusieurs réalisations (liées à l'environnement, choix de conception/implémentation...)
 - ◆ Diagramme d'implémentation
 - ◆ Vision « bas-niveau » de l'implémentation du logiciel
- ◆ Concepts proches entre diag. classe et langages objet
 - ◆ Classe, interface, méthode, attributs, spécialisation ...
 - ◆ Manque le concept d'association (pouvant se traduire par des attributs)
 - ◆ Mais encore une fois, peut utiliser des diagrammes de classe pour modéliser autre chose que du code objet

Diagramme de classes

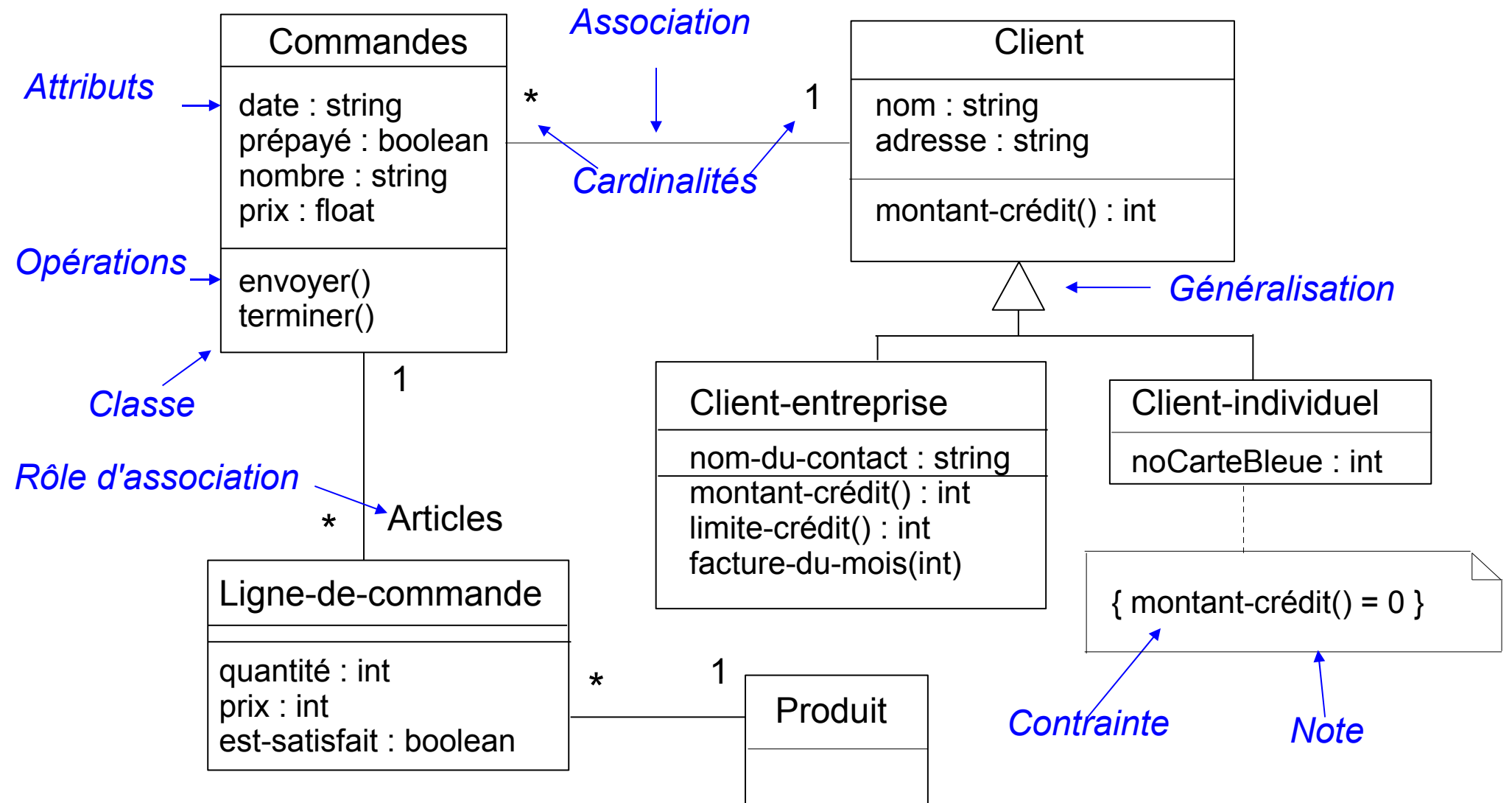


Diagramme de classes

◆ Attributs

- ◆ Élément caractérisant une partie de l'état d'un objet
- ◆ Syntaxe UML pour la définition d'un attribut :

```
visibilité nom [multiplicité] : type = init  
{propriétés}
```

- ◆ visibilité : + (public), # (protégé) ou – (privé)
- ◆ nom : nom de l'attribut
- ◆ multiplicité : nombre d'attributs de ce type (tableau : [1..5])
- ◆ type : type de l'attribut
- ◆ init : valeur initiale de l'attribut
- ◆ propriétés : propriétés, contraintes associées à l'attribut

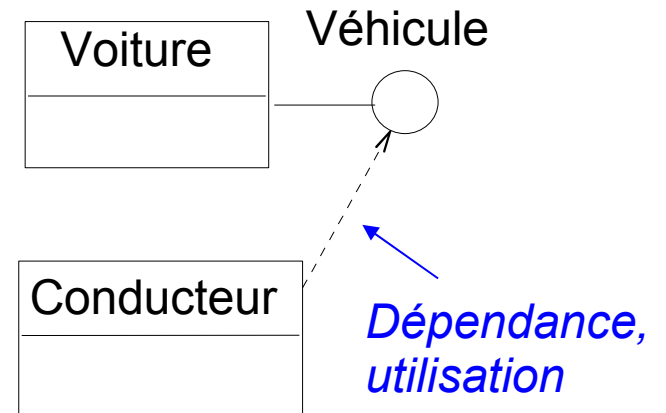
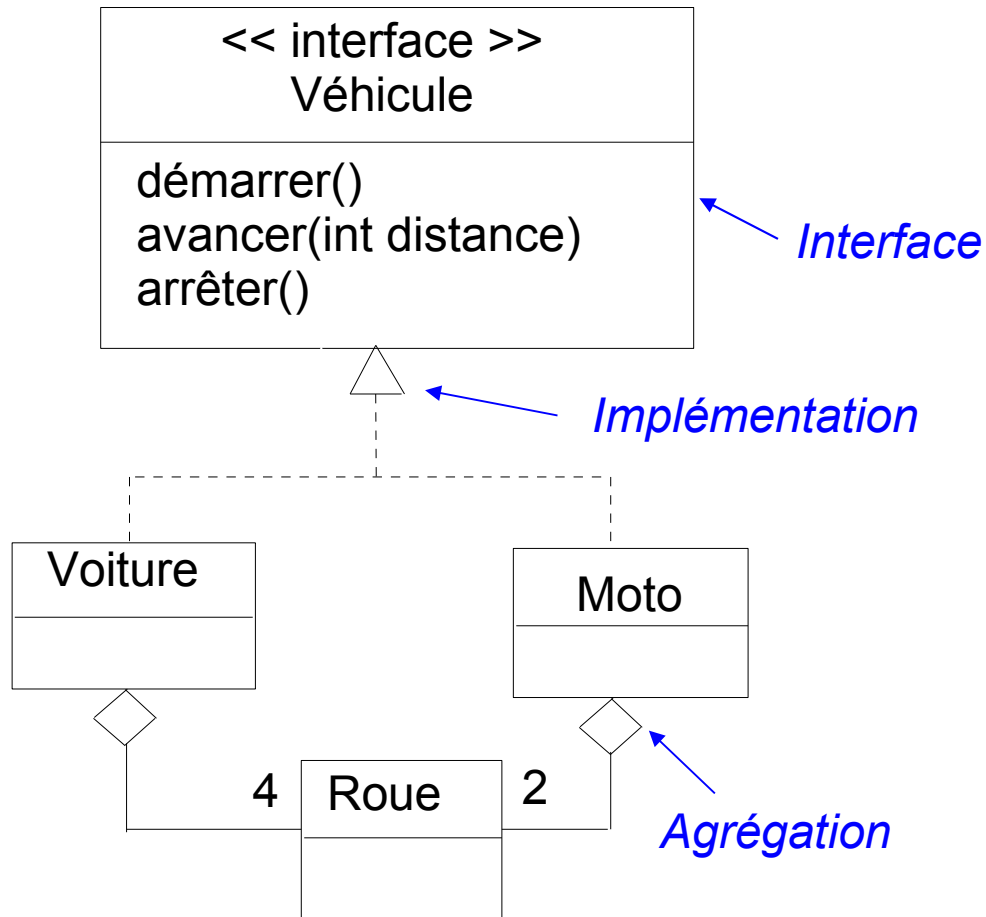
Diagramme de classes

- ◆ Opérations
 - ◆ Processus/fonction qu'une classe sait exécuter
 - ◆ Appelées également méthodes dans les langages objets
- ◆ Syntaxe UML pour la définition d'une opération :

```
visibilité nom(paramètres) : typeRetourné  
{propriétés}
```

- ◆ visibilité : + (public), # (protégé) ou – (privé)
- ◆ nom : nom de l'opération
- ◆ paramètres : liste des paramètres de l'opération
- ◆ typeRetourné : type de la valeur retournée par l'opération (si elle retourne une valeur)
- ◆ propriétés : propriétés, contraintes associées à l'opération

Diagramme de classes



◊ — Agrégation : les éléments existent toujours quand l'association est détruite

◆ — Composition : les éléments disparaissent quand l'association est détruite

◆ Variante avec classe abstraite

- ◆ Nom en italique
- ◆ Ne peut être instanciée

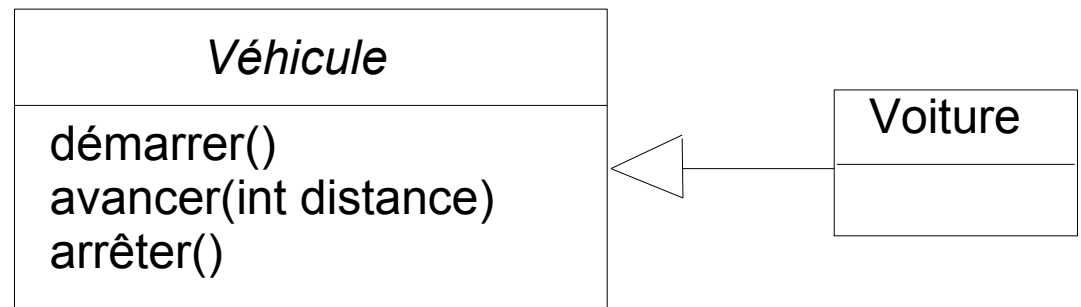
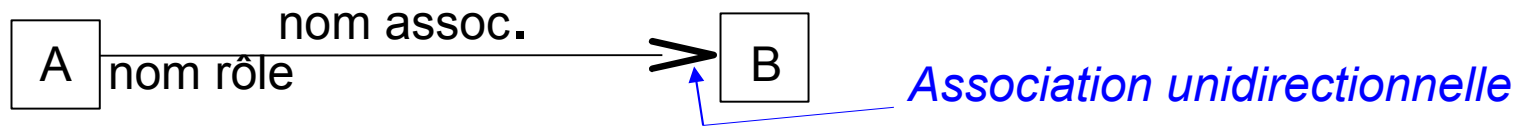


Diagramme de classes

◆ Détails sur associations



◆ Exemples de cardinalités d'associations

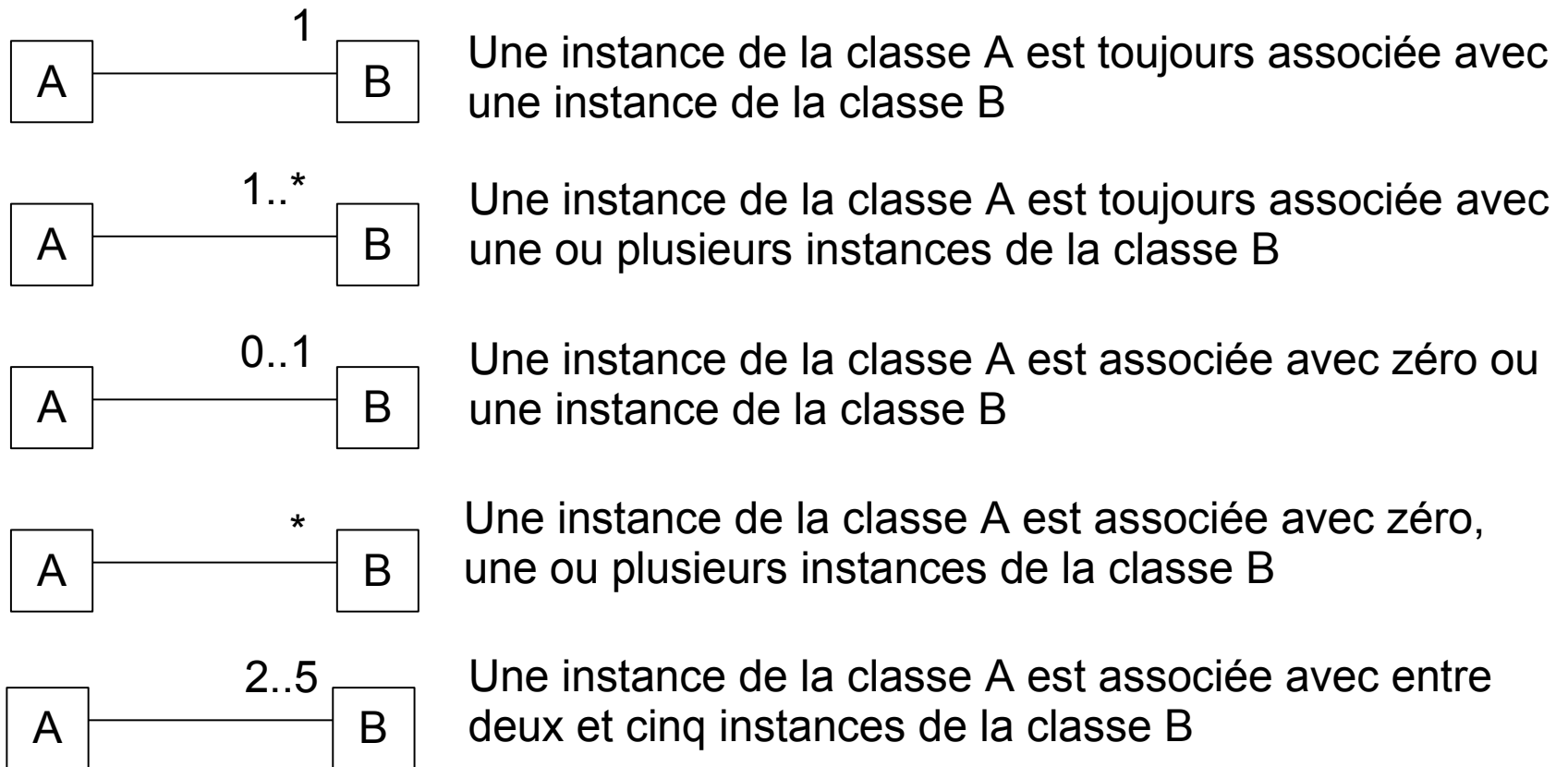


Diagramme de classes

◆ Enumeration

- ◆ Liste de valeurs manipulées comme un type

◆ Classe d'association

- ◆ A chaque couple des éléments de l'association, une instance d'une autre classe est associée
- ◆ Ici, à chaque employé d'une entreprise sont associées les informations sur son poste

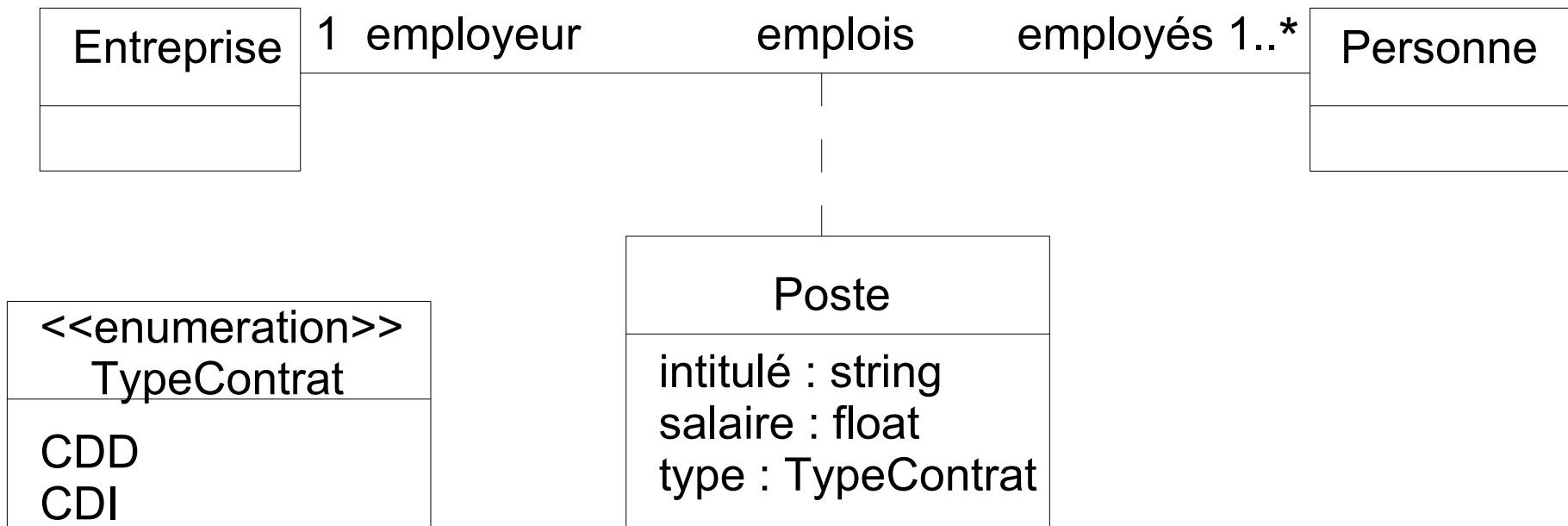
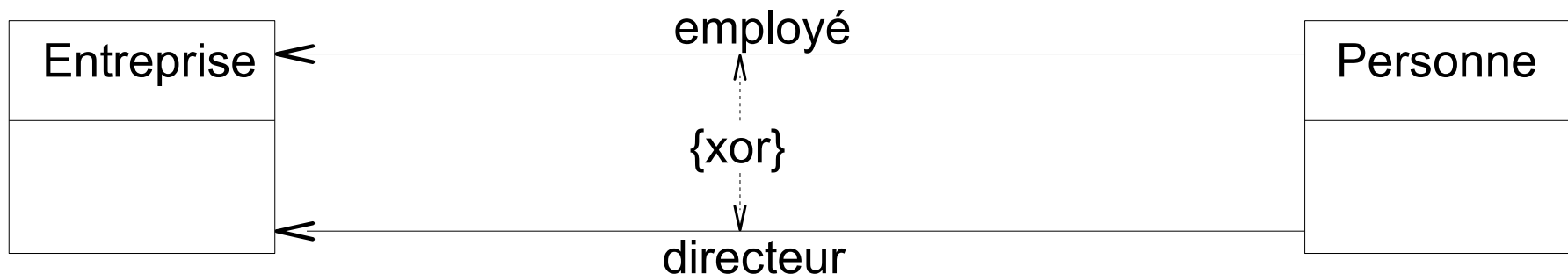


Diagramme de classes

- ◆ Contraintes sur les associations (en plus des cardinalités)
- ◆ Relation d'exclusion entre deux associations : soit l'une soit l'autre mais pas les deux à la fois



- ◆ Les éléments d'une association peuvent être ordonnés



- ◆ Une association peut être le sous-ensemble d'une autre

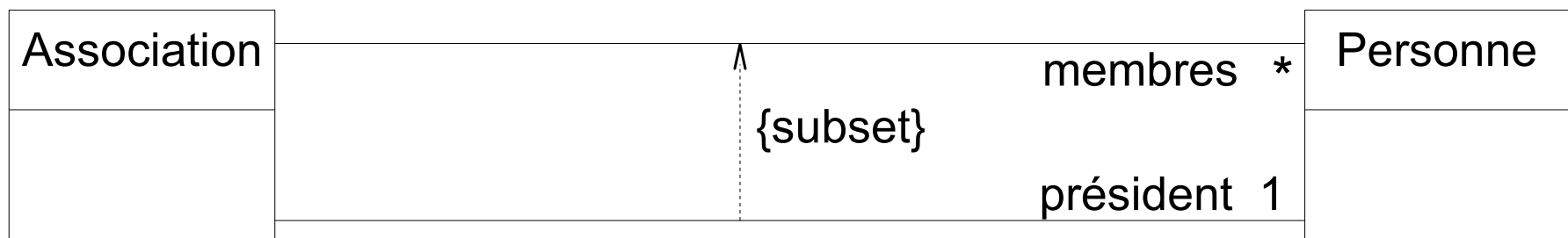
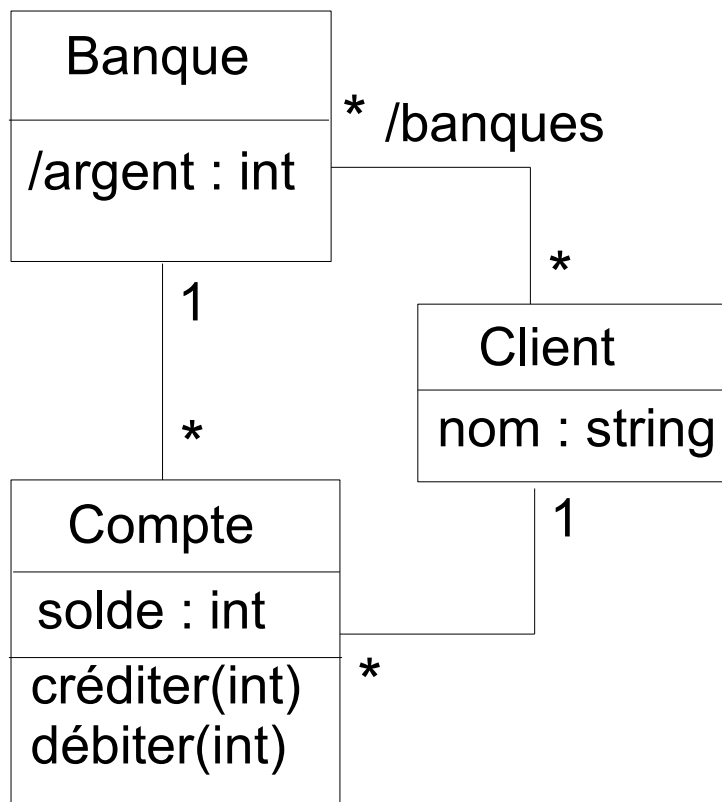


Diagramme de classes



- ◆ Éléments dérivés
 - ◆ Principalement pour attributs et associations
 - ◆ Se déduisent d'autres parties du diagramme
 - ◆ Nom de l'élément commence par /
- ◆ Exemples
 - ◆ L'ensemble des banques dont on est client se déduit de ses comptes bancaires
 - ◆ L'argent géré par une banque est la somme des soldes de ses comptes
- ◆ Ces éléments dérivés peuvent formellement être définis en OCL

Diagramme de classes

- ◆ Contraintes
 - ◆ Associations, attributs et généralisations spécifient des contraintes importantes (relations, cardinalités), mais ils ne permettent pas de définir toutes les contraintes
- ◆ UML permet d'ajouter des contraintes sur des éléments (classe, attribut, association, ...)
 - ◆ Soit des prédéfinies
 - ◆ Exemple : `{ordered}` et `{xor}` pour les associations
 - ◆ Soit des spécifiques définies par le concepteur
 - ◆ Pas de syntaxe précise préconisée, uniquement l'utilisation de `{ ... }`
 - ◆ En pratique, pour être précis, on exprimera ces contraintes en OCL
 - ◆ Exemple de contrainte explicite : on indique qu'un client individuel n'a pas de droit de crédit

Diagramme d'objets

- ◆ Objet = instance d'une classe
- ◆ Diagramme d'objets : ensemble d'objets respectant les contraintes du diagramme de classe
 - ◆ Respect des cardinalités
 - ◆ Chaque attribut d'une classe a une valeur affectée dans chaque instance de cette classe
- ◆ Diagramme de classes = définition d'un cas général
- ◆ Diagramme d'objets = définition d'un cas particulier de ce cas général

Diagramme d'objets

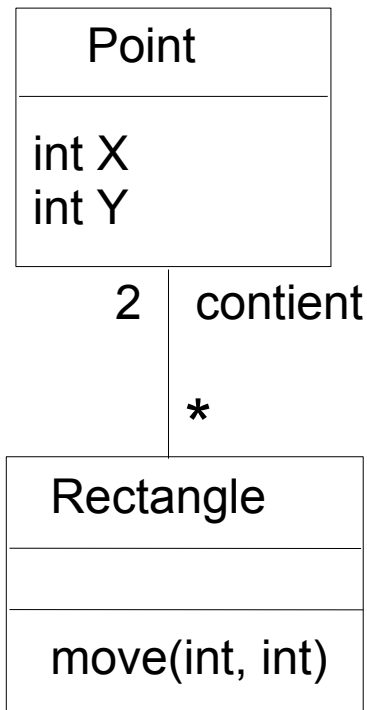
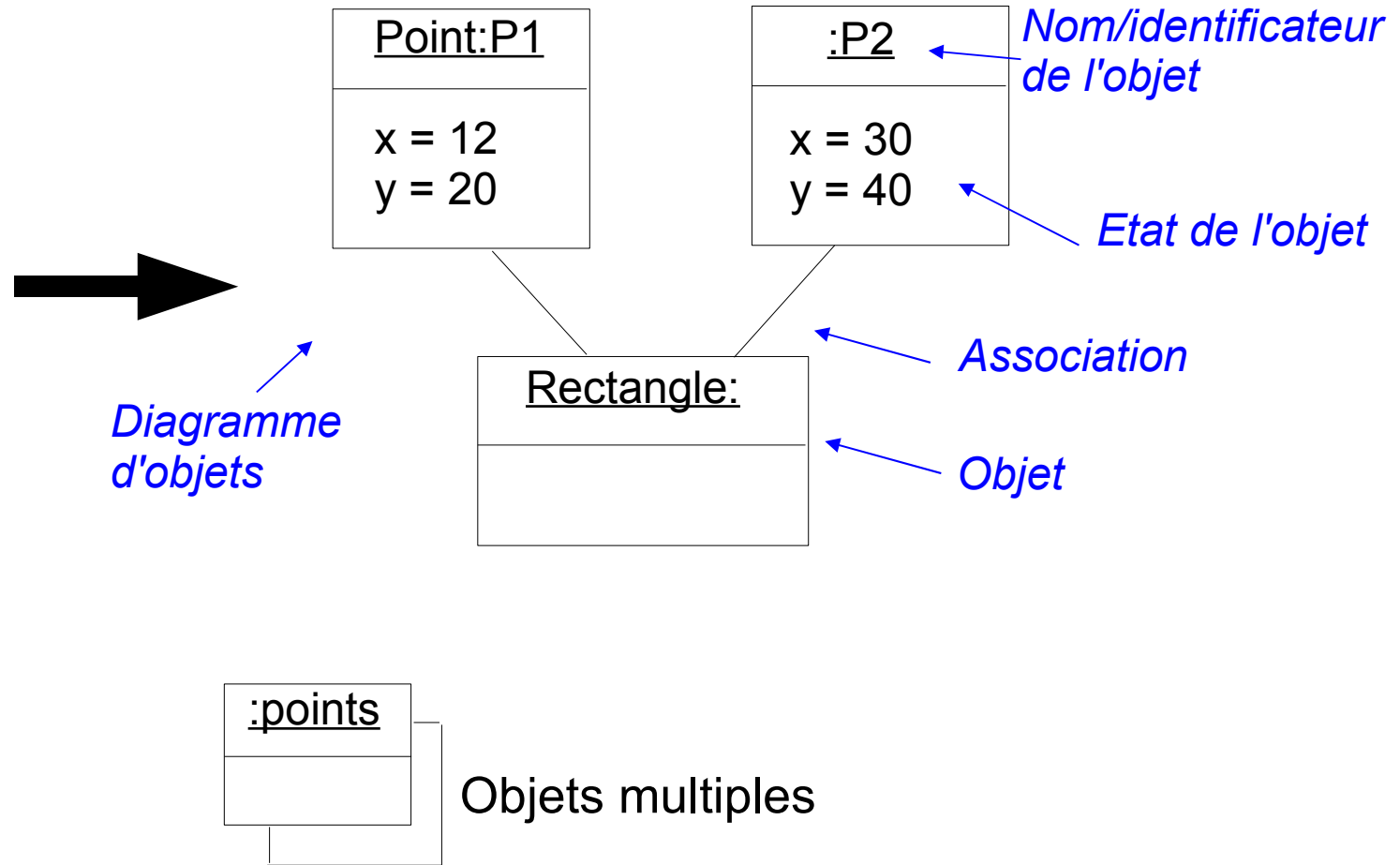


Diagramme de classe



identificateur de l'objet : NomClasse :NomObjet

Diagramme de composants

- ◆ Composant
 - ◆ Élément spécifiant ses interactions avec l'extérieur via la définition de ses interfaces fournies et requises
 - ◆ On connecte une interface requise d'un composant à l'interface fournie compatible d'un autre composant : assemblage
 - ◆ Composant composite
 - ◆ Composant peut être formé de composants internes assemblés par leurs interfaces
 - ◆ Composition hiérarchique de composants
- ◆ Port
 - ◆ Point d'interaction du composant
 - ◆ Associé à une interface d'opérations (en mode requis ou fourni)
- ◆ Connecteur
 - ◆ De délégation : lie un port du composite à un port d'un de ses éléments
 - ◆ D'assemblage : lie une interface d'un élément interne avec celle d'un autre élément interne

Diagramme de composants

- ◆ Ensemble de composants connectés entre eux par assemblage ou composition

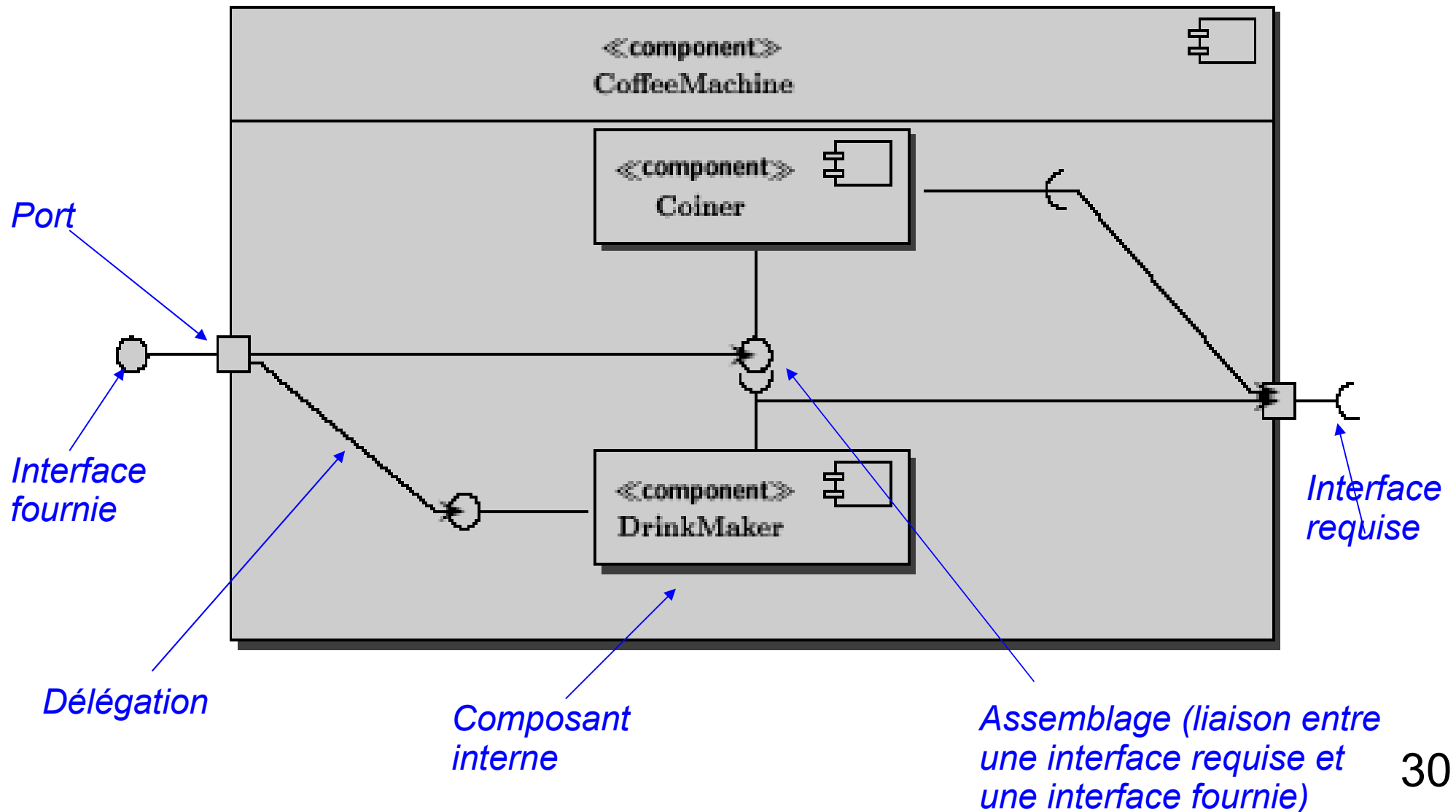
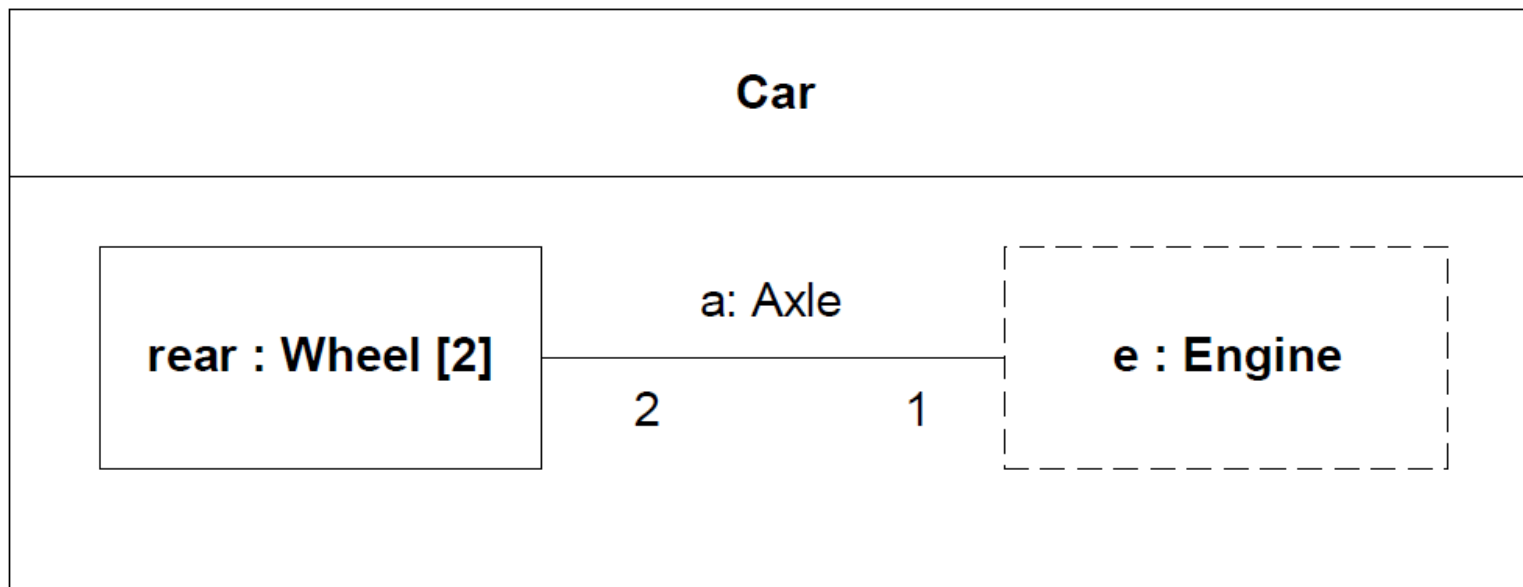


Diagramme de structure composite

- ◆ Diagramme conceptuellement assez proche d'un diagramme de composants
- ◆ Définit l'architecture interne d'une classe
 - ◆ Les éléments qui la forment (les *parts*)
 - ◆ Les interactions entre ces éléments (d'une manière proche des diagrammes de collaboration)



Plan

- ◆ Diagrammes fonctionnels
- ◆ Diagrammes statiques
- ◆ *Diagrammes dynamiques*
 - ◆ *D'états*
 - ◆ *De séquence*
 - ◆ *D'activité*
 - ◆ *De communication*
 - ◆ *Vue générale d'interaction*
 - ◆ *De temps*
- ◆ Diagramme d'implémentation

Diagrammes dynamiques

- ◆ Définition des aspects dynamiques d'une application, plusieurs points de vue
- ◆ Diagrammes d'états
 - ◆ Description du comportement d'un objet ou de l'opération d'un objet
 - ◆ Extension des diagrammes de Harel
- ◆ Diagrammes d'activité : diagrammes de flot de données
- ◆ Définition des interactions entre des objets
 - ◆ Description de la coopération d'un ensemble d'objets
 - ◆ 2 types de diagrammes d'interaction
 - ◆ Diagrammes de séquence : mise en avant de l'évolution et de l'enchaînement temporel des messages échangés
 - ◆ Diagrammes de communication : mise en avant des liens entre les objets et les messages échangés au travers de ces liens

Diagramme d'états

- ◆ Diagrammes d'états : comportement interne d'un objet
 - ◆ La définition de tous les états possibles d'un objet
 - ◆ La définition de tous les changements d'états via des transitions
- ◆ Associé à un objet ou à une opération

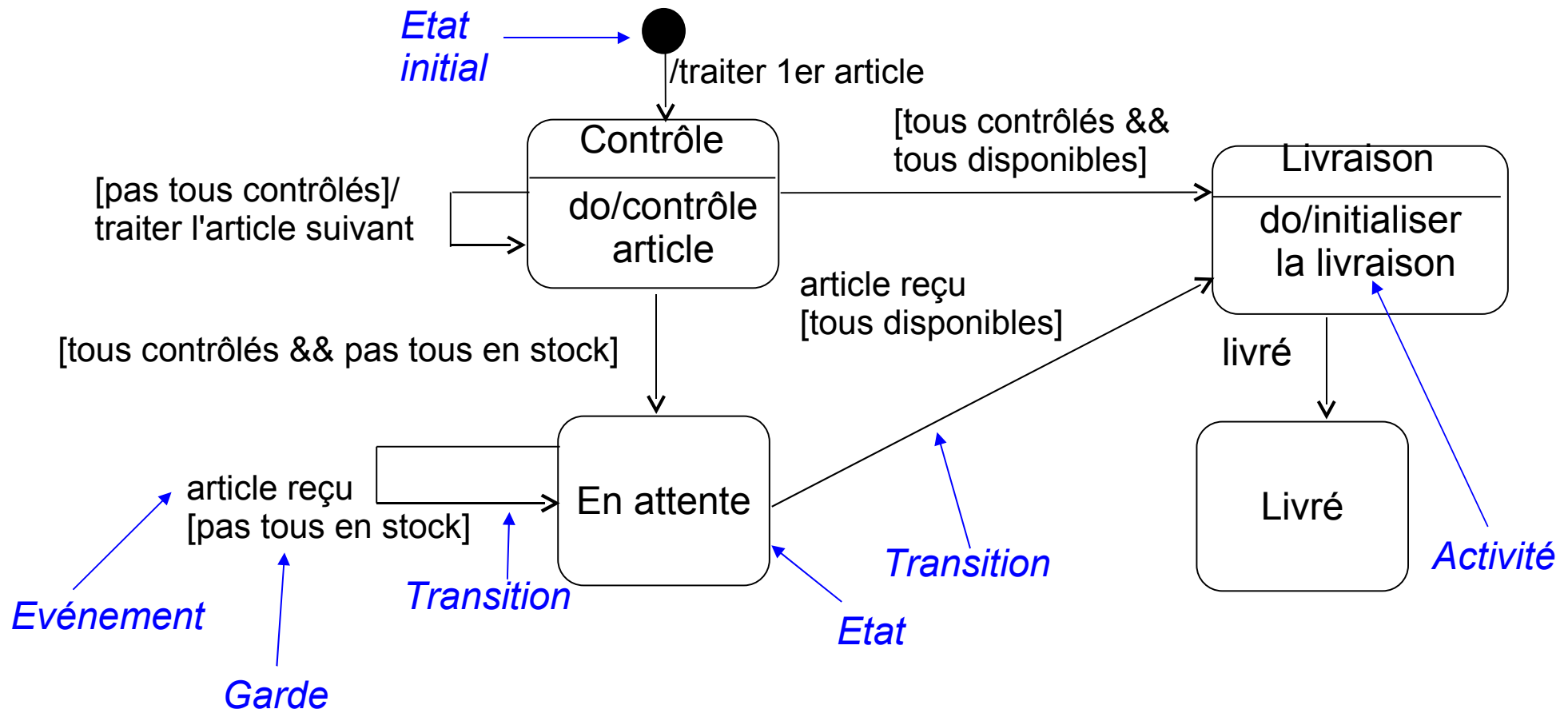


Diagramme d'états

- ◆ Diagrammes d'états – syntaxe
 - ◆ Syntaxe d'une transition
 - ◆ événement [garde] / action
 - ◆ Chaque partie est optionnelle
 - ◆ La transition est suivie si l'événement a été généré et que la garde est valide
 - ◆ Exécute alors l'action avant de rentrer dans l'état ciblé par la transition
 - ◆ Attention : pas deux transitions possibles partant d'un même état
 - ◆ Syntaxe des activités que l'on peut associer à un état
 - ◆ `do / action` : action exécutée dans l'état
 - ◆ `entry / action` : action exécutée à l'entrée dans l'état
 - ◆ `exit / action` : action exécutée à la sortie de l'état
 - ◆ `evt / action` : transition interne pour l'occurrence de l'événement `evt`
- ◆ Lien avec l'objet associé au diagramme d'états
 - ◆ Les actions peuvent être les méthodes de la classe de l'objet
 - ◆ Peut utiliser les attributs de l'objet, par exemple dans les gardes des transitions

Diagramme d'états

- ◆ Diagrammes d'états : notion d'état composite
 - ◆ Permet de structurer de manière hiérarchique les états et les transitions
 - ◆ Exemple d'une commande annulée sans super état

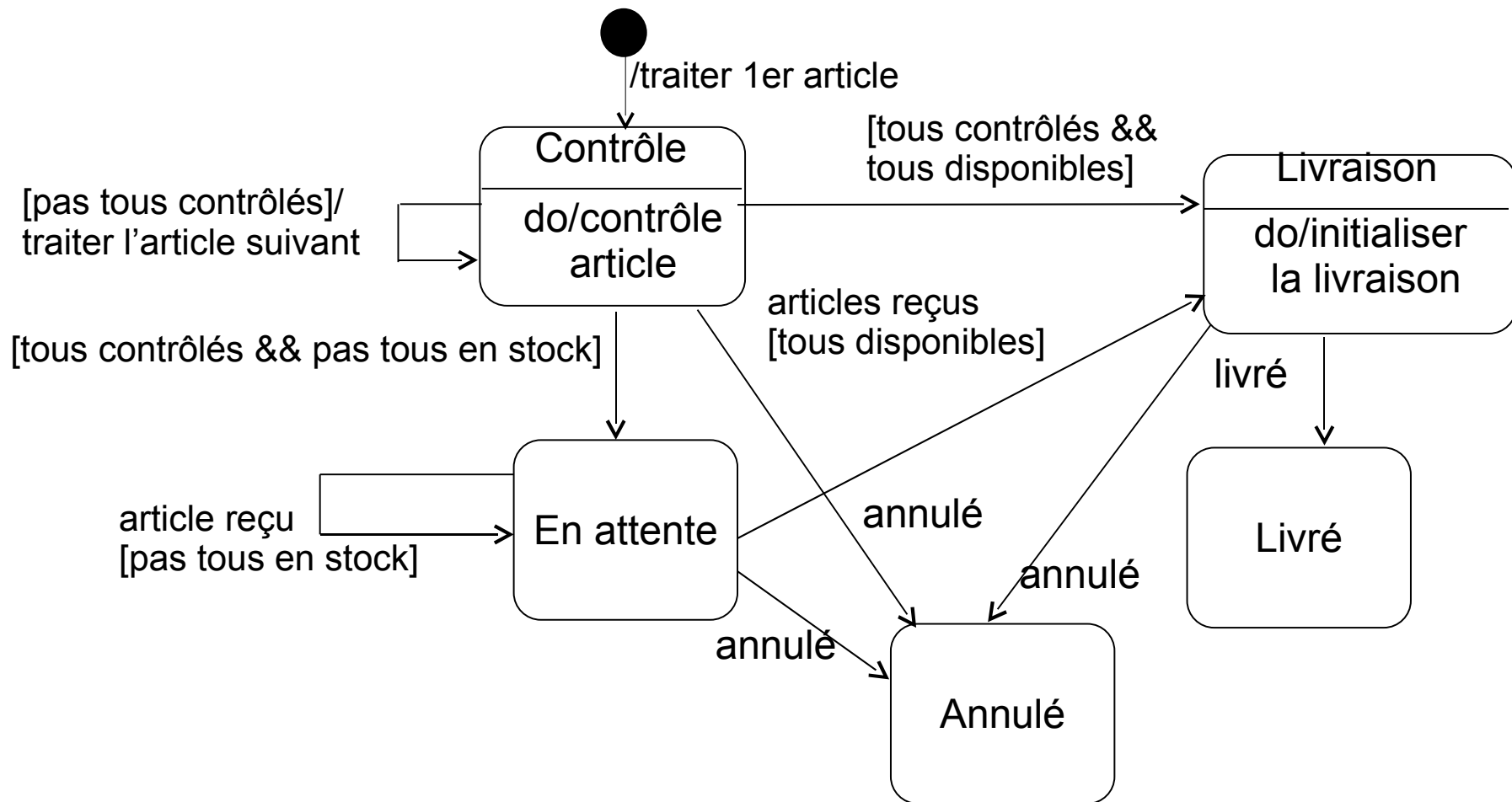


Diagramme d'états

- ◆ Diagramme d'états : notion d'état composite
- ◆ Exemple d'une commande annulée avec super état
- ◆ Permet de factoriser la transition associée à l'événement Annuler et de définir 3 états principaux (Actif, Livré, Annulé)

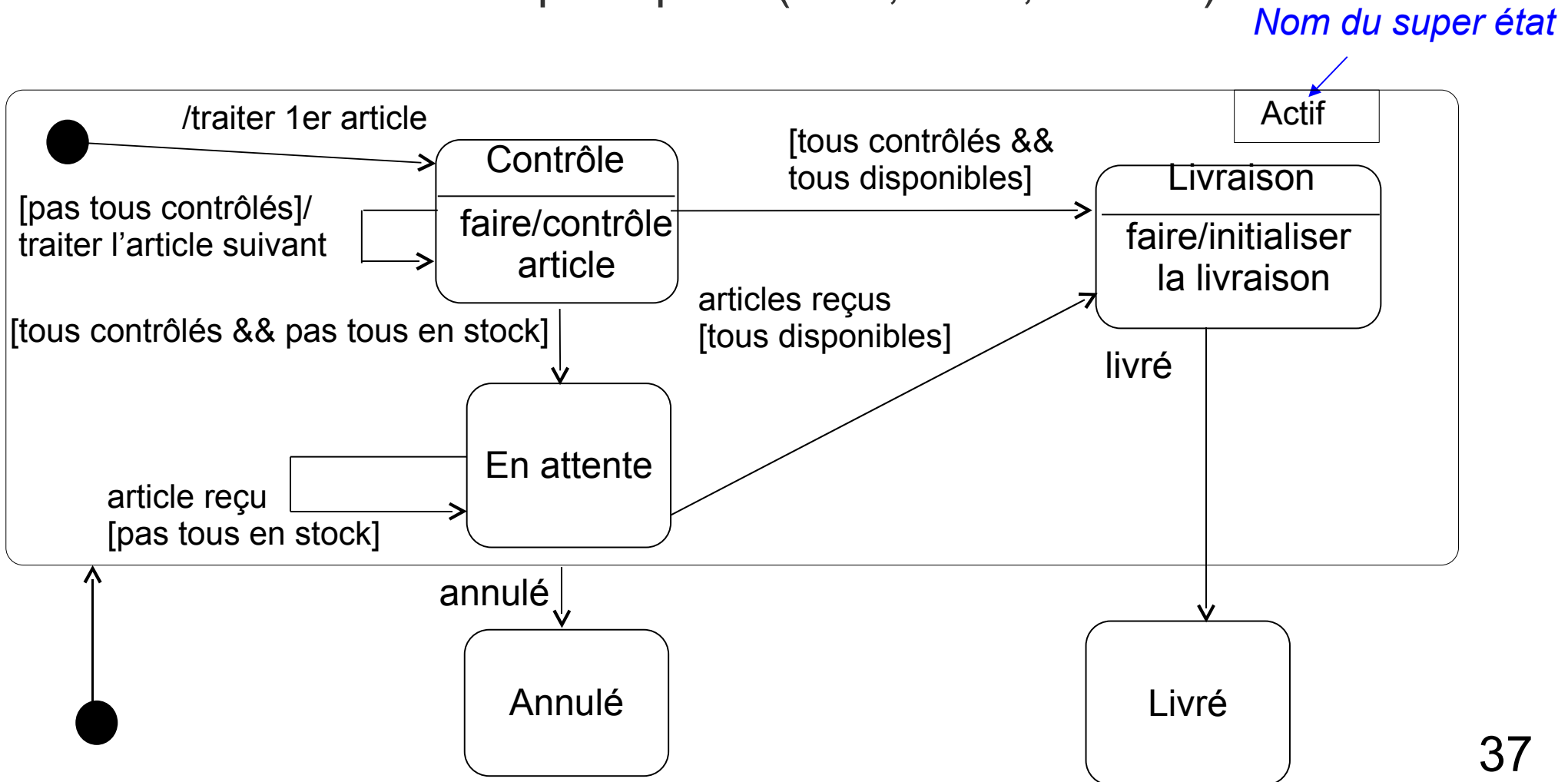


Diagramme d'états

◆ Diagramme d'états concurrents

- ◆ Plusieurs sous-parties parallèles au sein d'un composite
- ◆ Possibilité d'ajouter des éléments de synchronisation entre les sous-parties

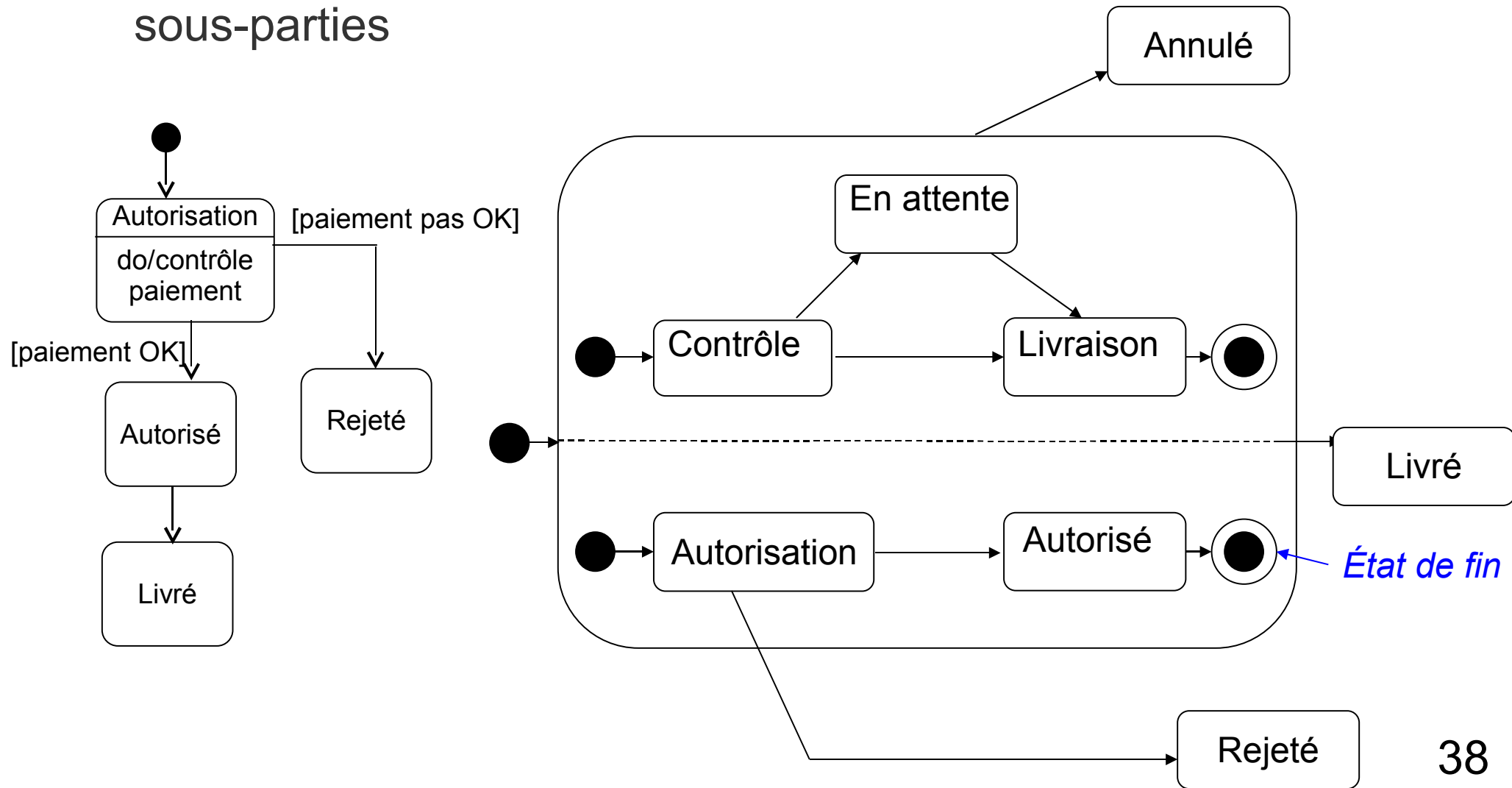
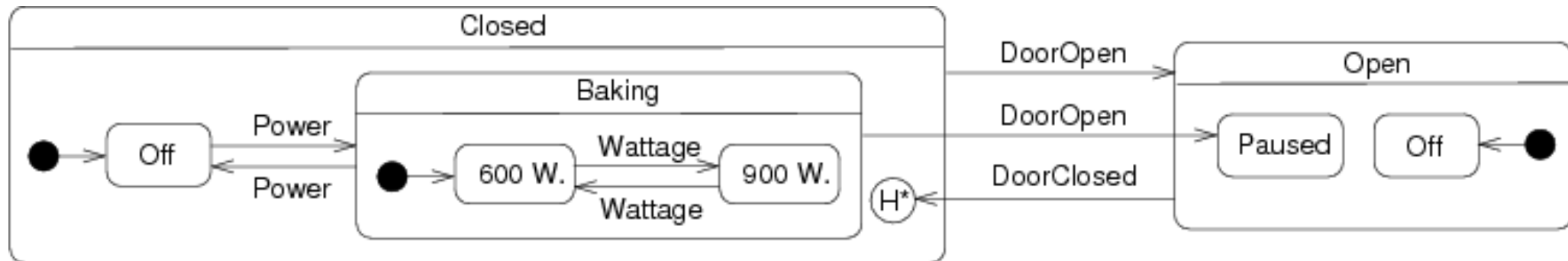


Diagramme d'états

◆ Etats historiques

- ◆ Dans un état composite, permet de revenir dans l'état interne qui était celui qu'on a quitté en dernier
- ◆ Deep history (H^*) : si dernier état est un composite, réactive également son dernier état interne et ainsi de suite jusqu'au bout de la hiérarchie
- ◆ Shallow history (H) : ne réactive que le « premier » niveau (donc si dernier état est un composite, prend son état initial)



◆ Exemple

- ◆ Hiérarchie initiale d'états actifs : Closed / Baking / 900W
- ◆ Puis événements DoorOpen et DoorClosed
- ◆ Si deep history (comme sur le diag.) : retrouve Closed / Baking / 900W
- ◆ Si shallow history : Closed / Baking / 600W

Diagramme d'activités

- ◆ Diagrammes d'activités
 - ◆ A utiliser
 - ◆ Pour analyser un cas d'utilisation
 - ◆ Pour comprendre un flot de données traversant plusieurs cas d'utilisation
 - ◆ Description des comportements parallèles
 - ◆ Modélisation de flot de données (workflow)
 - ◆ Dérivé de diagrammes d'événements, de réseaux de Petri, de SDL
 - ◆ Inconvénient
 - ◆ Lien entre activité et objet pas défini clairement
 - ◆ Selon le niveau de modélisation, une activité correspond à
 - ◆ Conception : une tâche qui est exécutée soit par un humain ou par un ordinateur
 - ◆ Spécification/implémentation : une méthode ou le comportement d'une classe

Diagramme d'activités

◆ Diagrammes d'activités

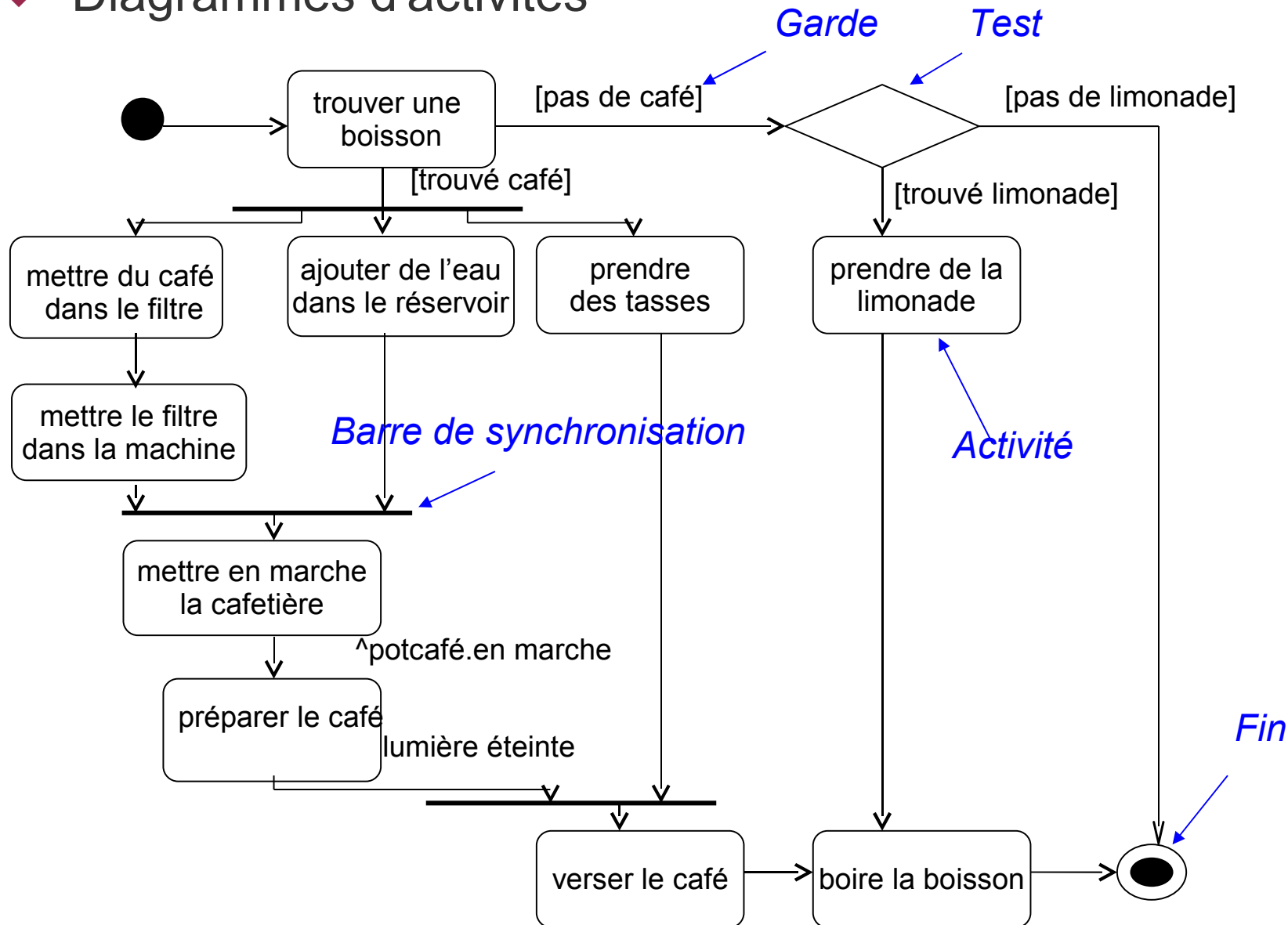


Diagramme d'activités

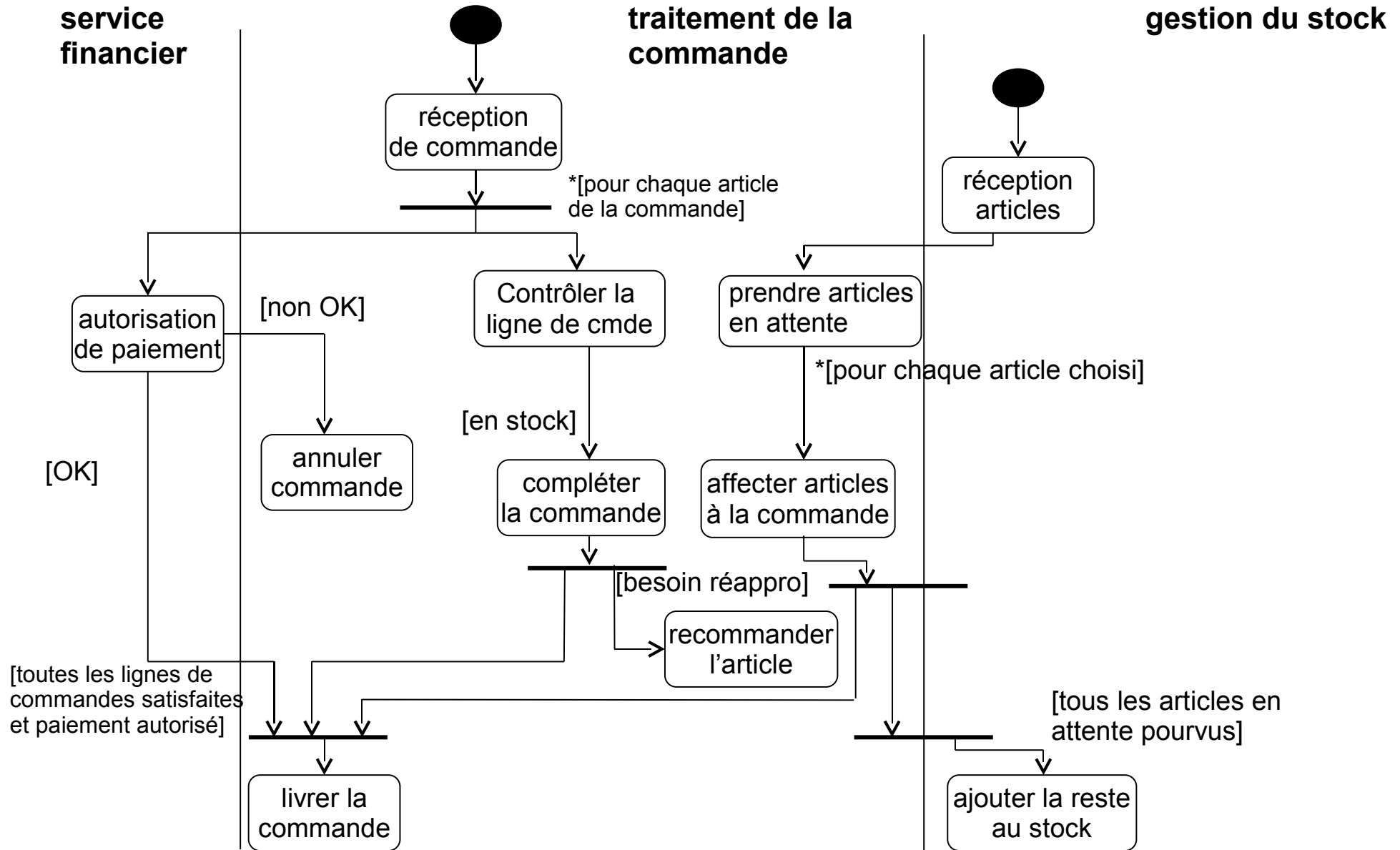


Diagramme de séquence

- ◆ Interaction entre objets
 - ◆ Chaque objet est représentée par une ligne verticale
 - ◆ Temps s'écoule de haut en bas
 - ◆ Précision des messages échangés entre les objets
 - ◆ Message = appel de méthode
- ◆ Permet de spécifier l'ordonnancement temporel des interactions entre les objets
 - ◆ Enchaînement / imbrication des appels de méthodes
- ◆ Nouveauté UML 2 : ajout de cadres pour définir des boucles, des alternatives ...
 - ◆ Mais peut vite devenir assez peu lisible en pratique

Diagramme de séquence

◆ Diagramme de séquence

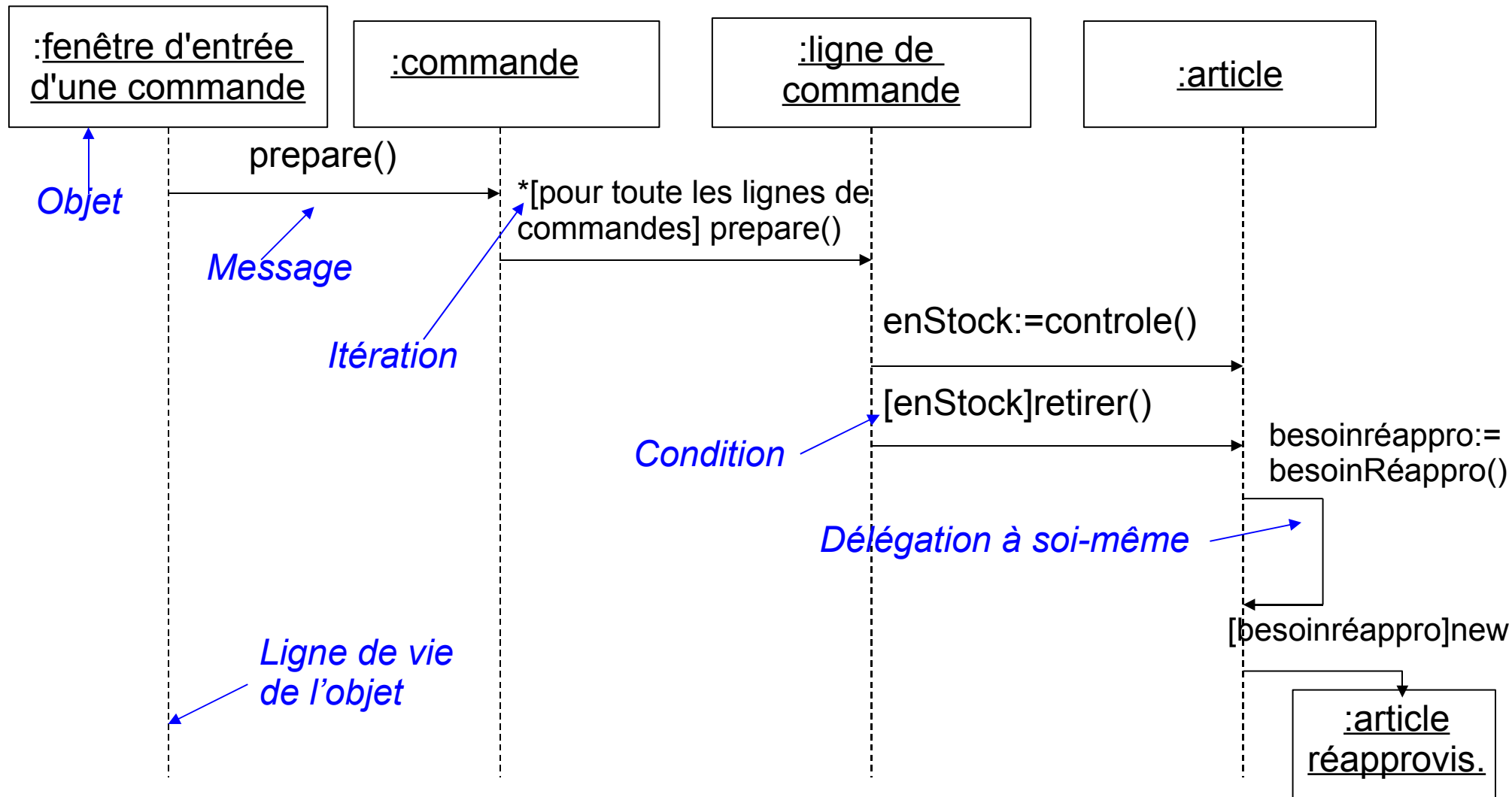


Diagramme de séquence

◆ Diagramme de séquence (suite)

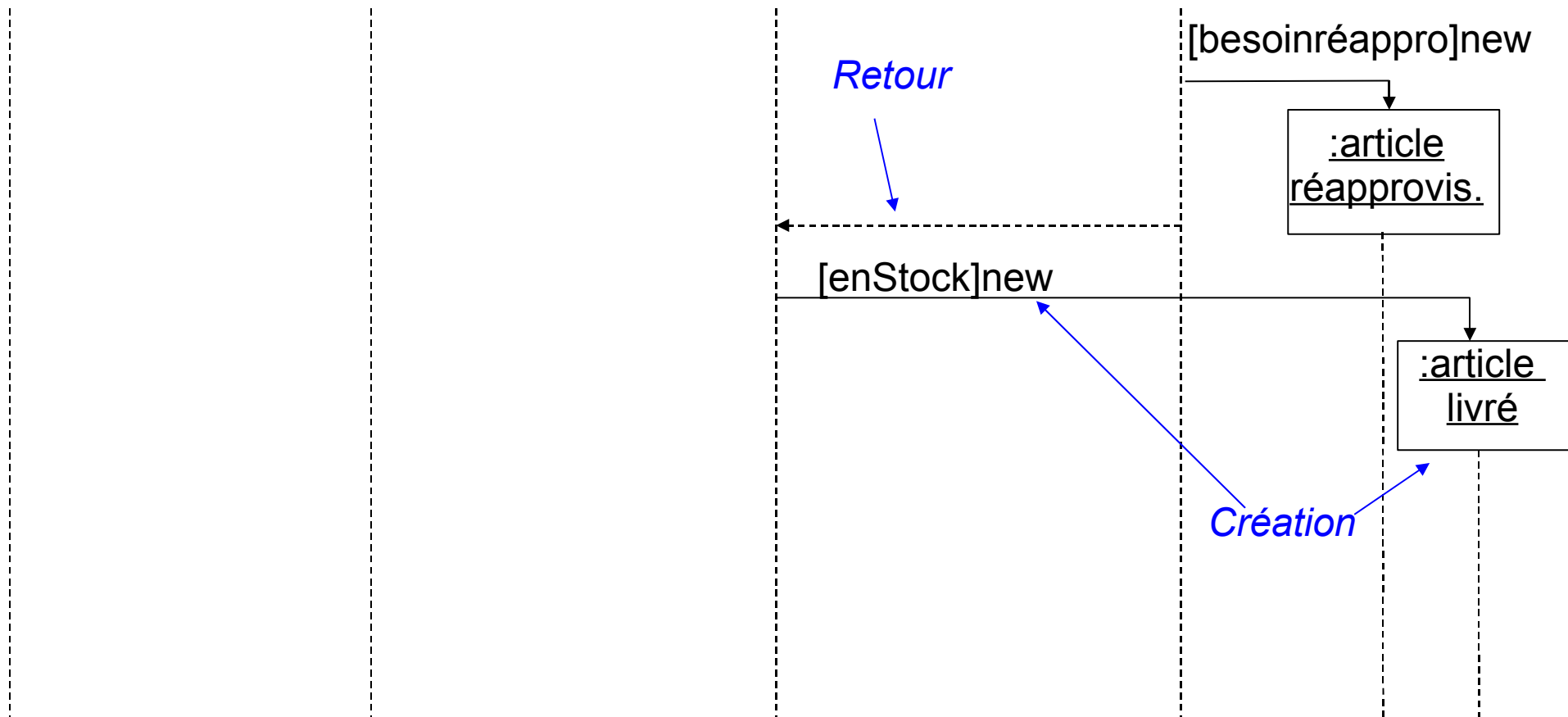


Diagramme de séquence

- ◆ Diagrammes de séquence et processus concurrents
 - ◆ Précise explicitement quand les objets sont actifs (au sens flot de contrôle d'un processus / thread)

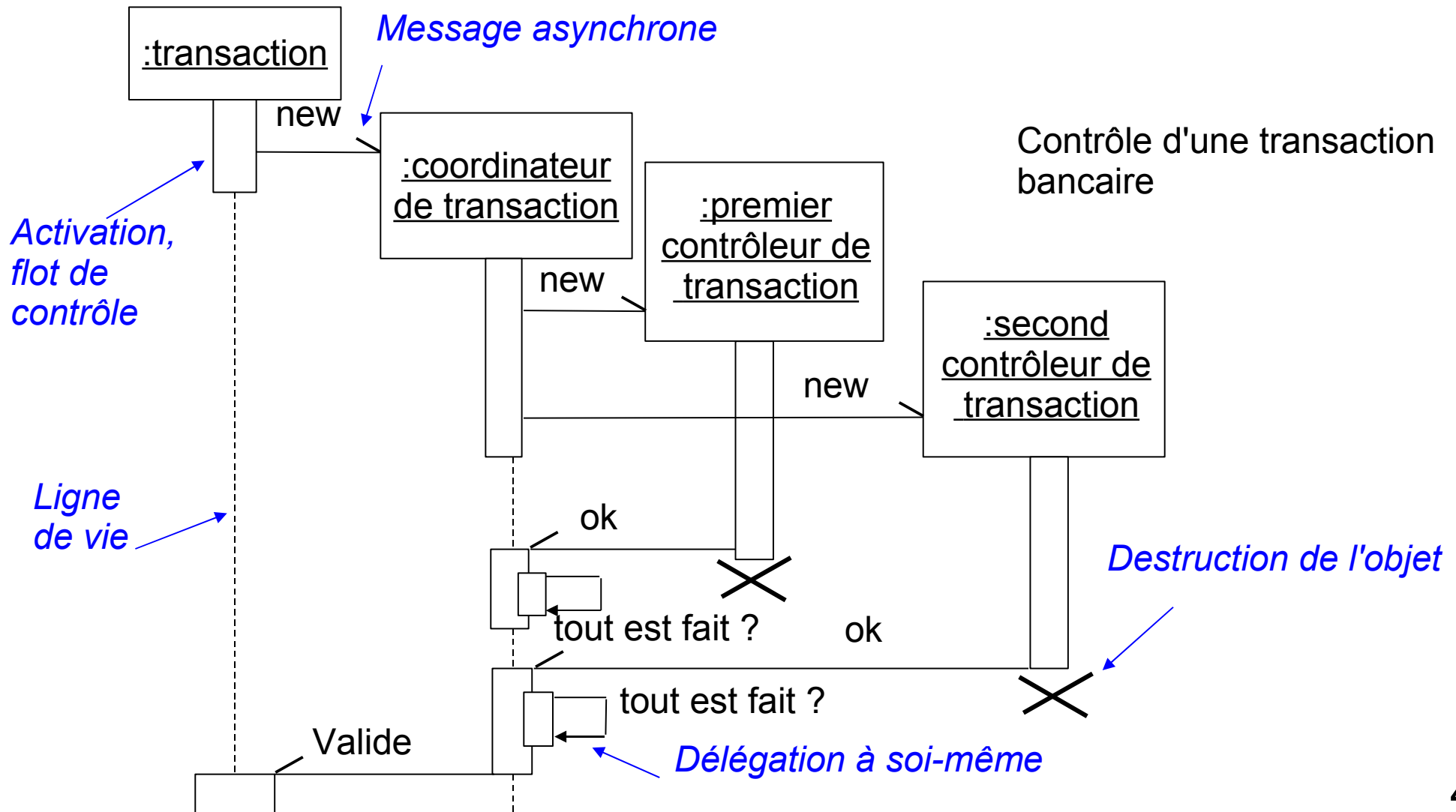


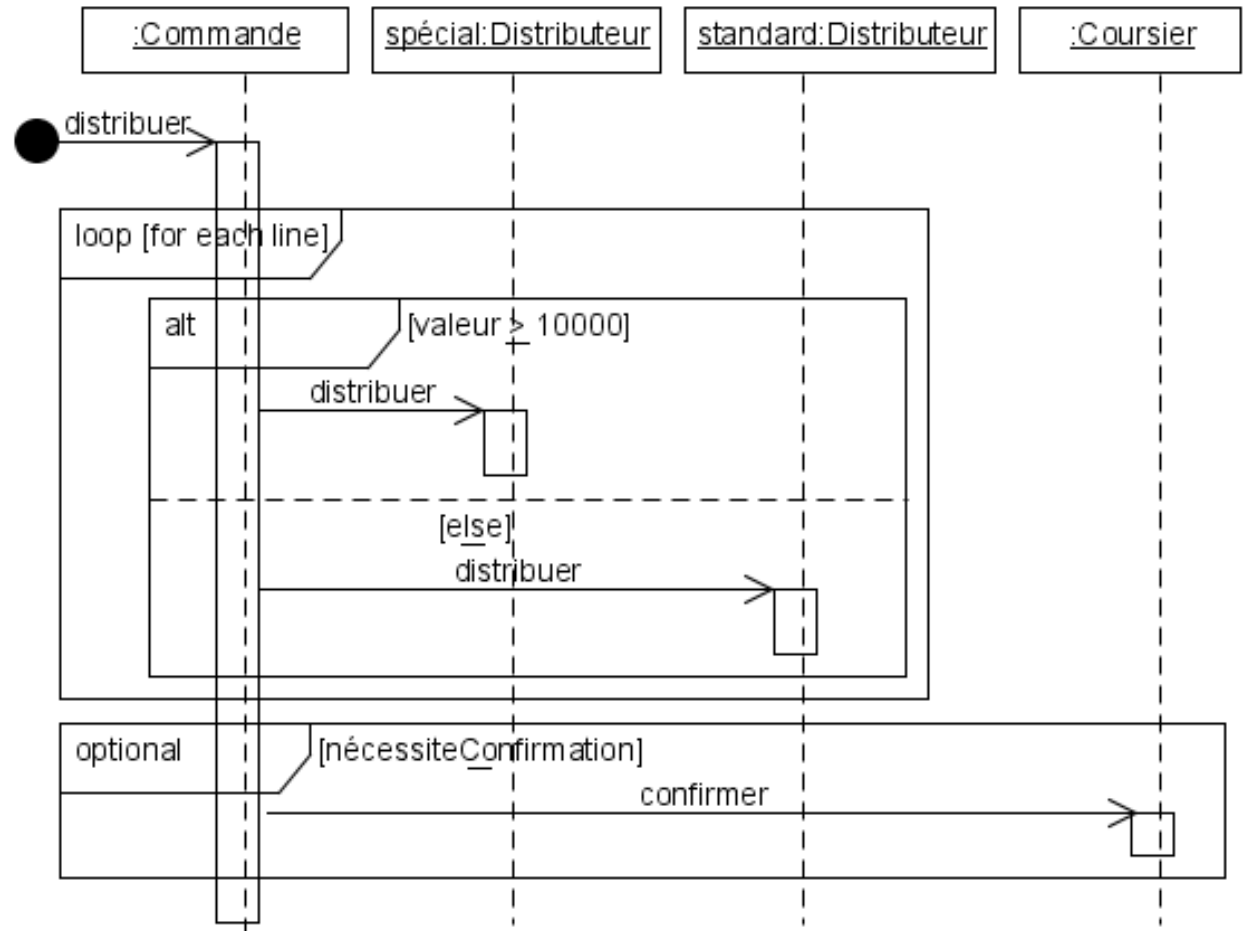
Diagramme de séquence

- ◆ Cadre d'interaction
 - ◆ Cadre qui englobe une partie du diagramme de séquence (un fragment) pour définir un fonctionnement non séquentiel
- ◆ Types de cadres
 - ◆ Alt
 - ◆ Alternative (if-then-else) entre deux parties selon une garde
 - ◆ Loop
 - ◆ Boucle
 - ◆ Opt
 - ◆ Partie optionnelle (if-then) selon une garde
 - ◆ Par
 - ◆ Deux parties en parallèle
 - ◆ Region
 - ◆ Partie en exécution mutuelle (processus / thread)

Diagramme de séquence

◆ Exemple d'utilisation des cadres

```
-- tiré de [Fowler2004]
procedure distribuer
  foreach (ligne)
    if (produit.valeur
        > $10000
        spécial.distribuer
    else
        standard.distribuer
    endif
  end for
  if (nécessiteConfirmation)
    coursier.confirmer
  end procedure
```



Diagrammes de collaboration

- ◆ Diagramme d'interaction « équivalent » au diagramme de séquence
 - ◆ Met en avant la vue structurelle au lieu de temporelle
- ◆ UML 1.X : deux types de diagrammes de collaboration
 - ◆ Au niveau classe (spécification)
 - ◆ Au niveau instance
- ◆ Notion de rôle : un élément a une fonction particulière
- ◆ Deux niveaux / étapes
 - ◆ Définition du diagramme de collaboration qui représente une interaction
 - ◆ L'utilisation d'une collaboration pour montrer l'interaction d'éléments dans un diagramme de classes ou d'objets
 - ◆ Ces éléments sont liés à un rôle de la collaboration

Diagramme de communication

- ◆ Diagramme de communication
 - ◆ Nouveau nom du (des ?) diagramme de collaboration en UML 2
- ◆ Du ? Des ?
 - ◆ Le diagramme de collaboration au niveau classe semble avoir disparu ...
 - ◆ Dans ce cours, ce diagramme là sera tout de même présenté (d'où la conservation du nom « diagramme de collaboration » dans les transparents qui suivent)
 - ◆ Diagramme de collaboration au niveau instance = diagramme de communication
 - ◆ Diagramme de collaboration au niveau classe = ???
- ◆ Diagramme de séquence vs collaboration
 - ◆ Le diagramme de séquence n'existe qu'au niveau instance
 - ◆ Au lieu de supprimer le diagramme de collaboration au niveau classe, il aurait mieux vullu ajouter un diagramme de séquence au niveau classe ...

Diagramme de collaboration (instance)

- Diagramme de collaboration au niveau instance

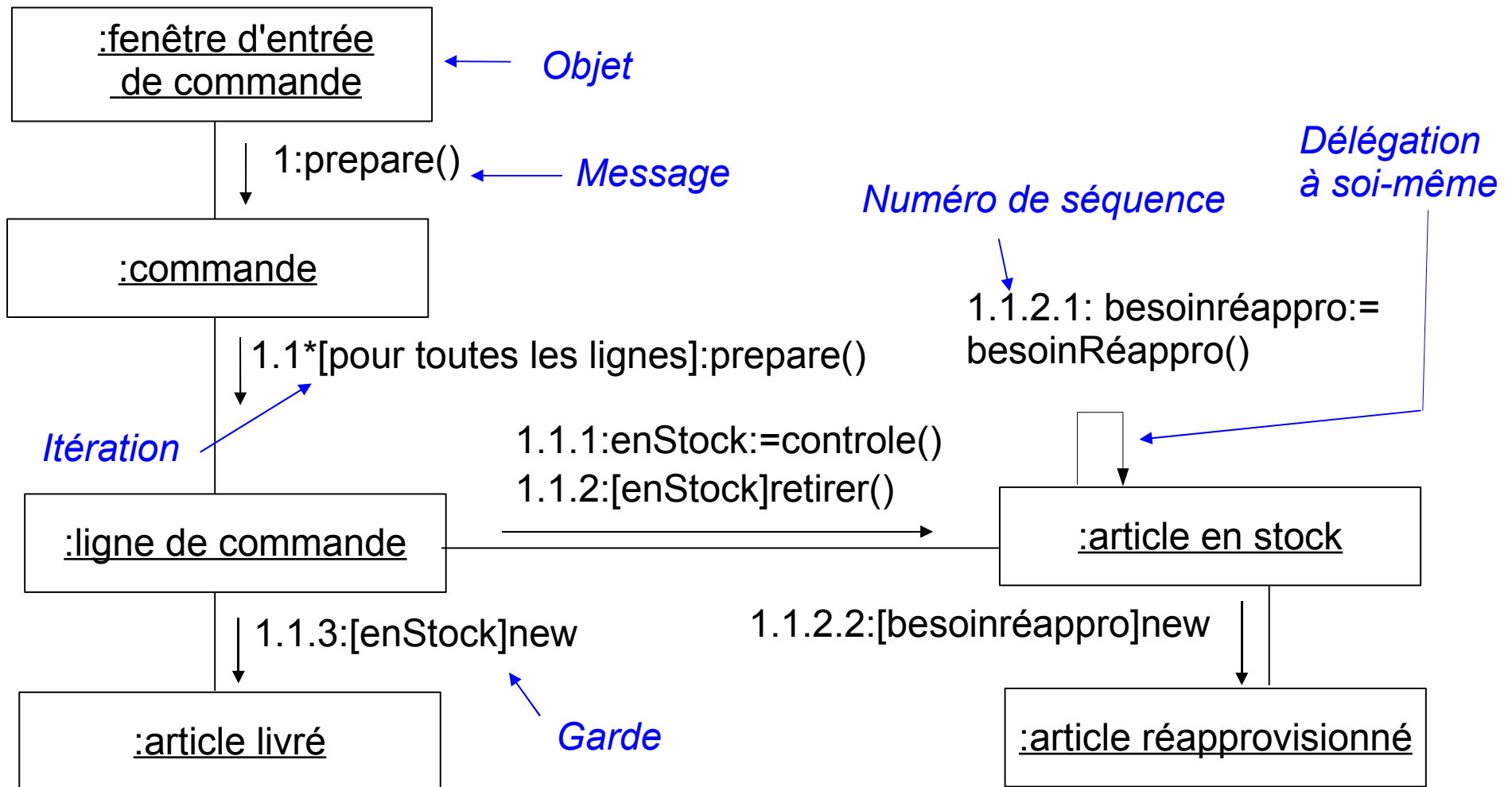


Diagramme de collaboration (classe)

- ◆ Diagramme de collaboration au niveau classe

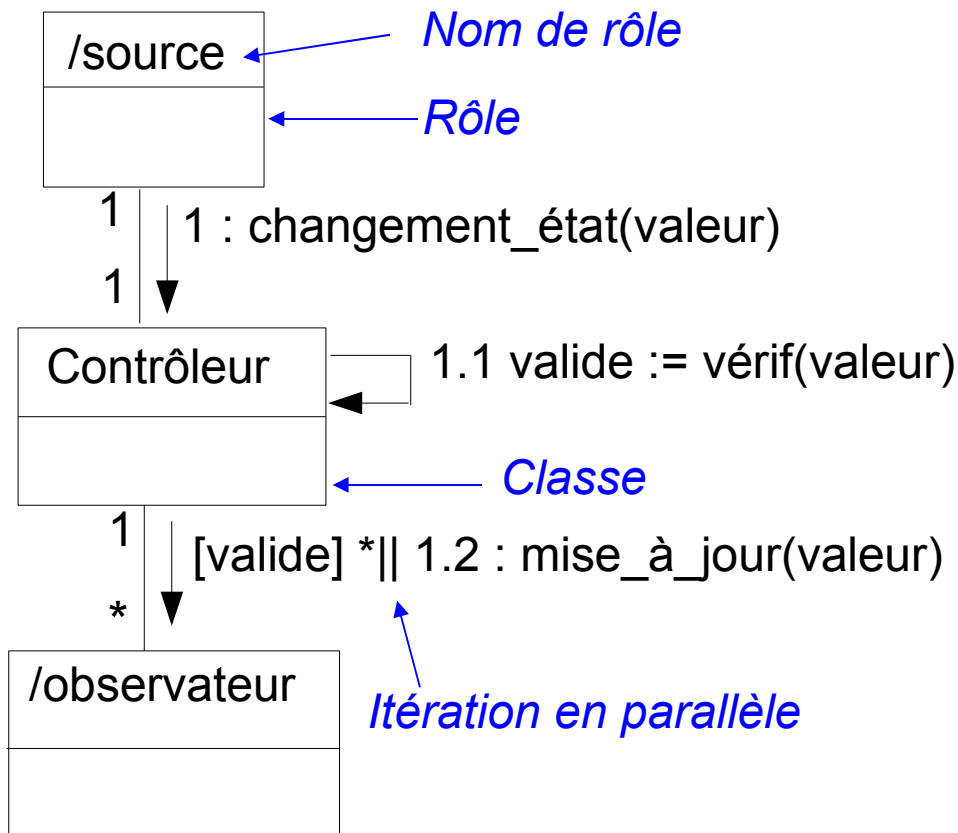


Diagramme de collaboration nommée « Observation »

Nom d'un rôle : /roleName [: className]

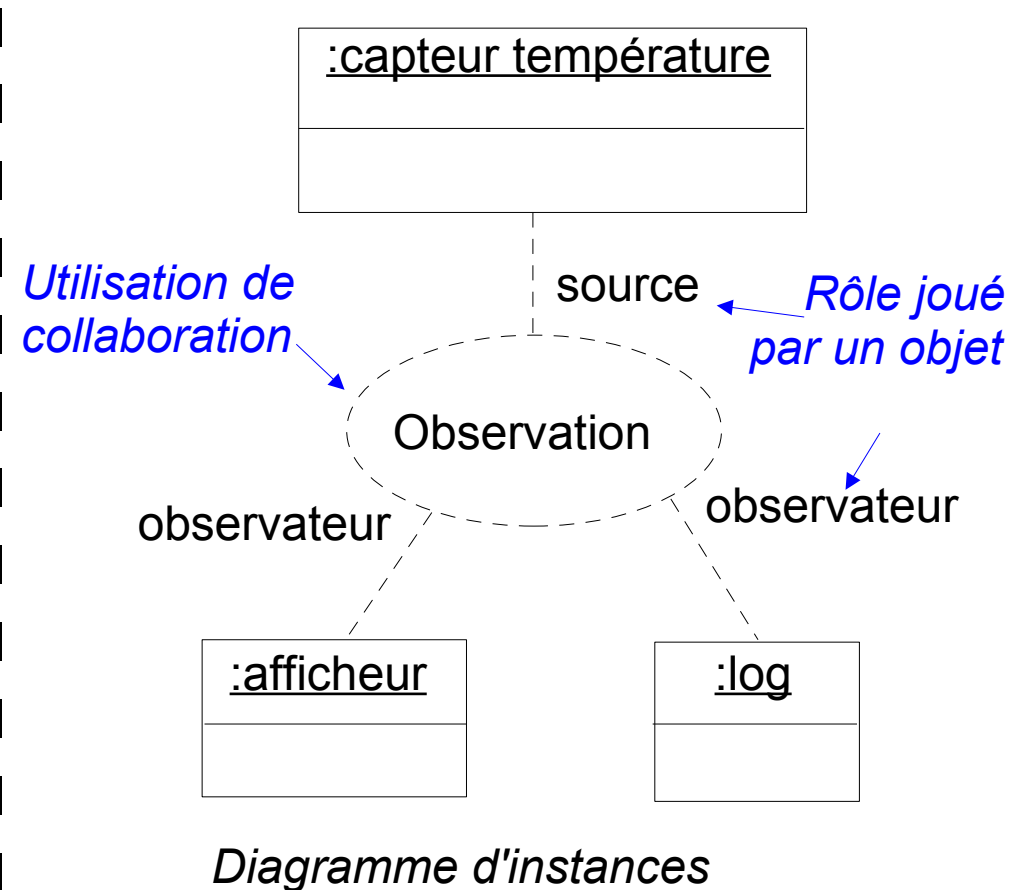


Diagramme de collaboration

◆ Informations portées sur les messages :

◆ `[pré "/"] [["cond"]] [séq] ["*" [" | | "] [["iter"]]]
": "] [r " := "] msg" ("[par]") "`

◆ `pré` : numéro des messages prédécesseurs

◆ `cond` : condition, garde d'envoi du message

◆ `séq` : numéro de séquence du message

◆ `*` : itération, `| |` : en parallèle

◆ `iter` : détaille l'itération

◆ `r` : stocke la valeur de retour du message

◆ `msg` : nom de l'opération à appeler

◆ `par` : paramètres de l'opération

◆ Exemples

◆ `[heure = midi] 1 : manger()`

◆ `3 / * | | [i := 1..5] : fermer()`

◆ `1.3, 2.1 / [t < 10s] 2.5 : age := demanderAge(nom) 53`

Types de messages

- ◆ 4 types de messages utilisables dans diagramme de collaboration et de séquence

—————▶ Appel de procédure, flot de contrôle imbriqué

—————> Flot de contrôle à plat (message généralement asynchrone)

—————\ Message asynchrone

-----> Retour d'appel de procédure

Diagrammes dynamiques – conclusion

- ◆ Diagrammes d'interaction (séquence ou collaboration)
 - ◆ Pour comprendre la coopération entre les objets
- ◆ Diagrammes d'états
 - ◆ Pour comprendre le comportement interne d'un objet
- ◆ Diagrammes d'activités
 - ◆ Pour analyser un cas d'utilisation
 - ◆ Pour comprendre un flot de données traversant plusieurs cas d'utilisation
 - ◆ Pour comprendre les applications multi-activités

Diagramme de vue globale d'interaction

- ◆ Sorte de « mélange » d'un diagramme de séquence et d'un diagramme d'activité

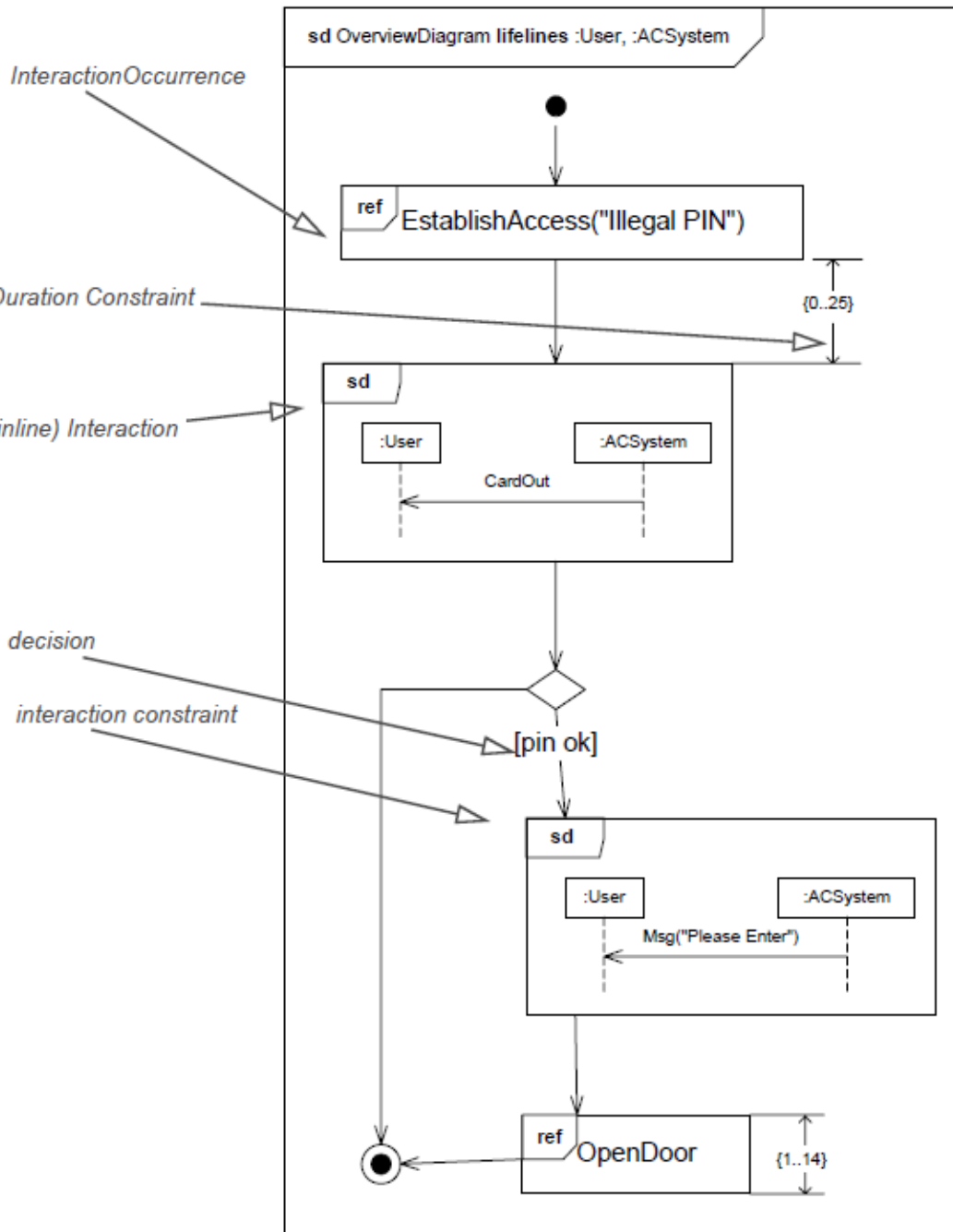
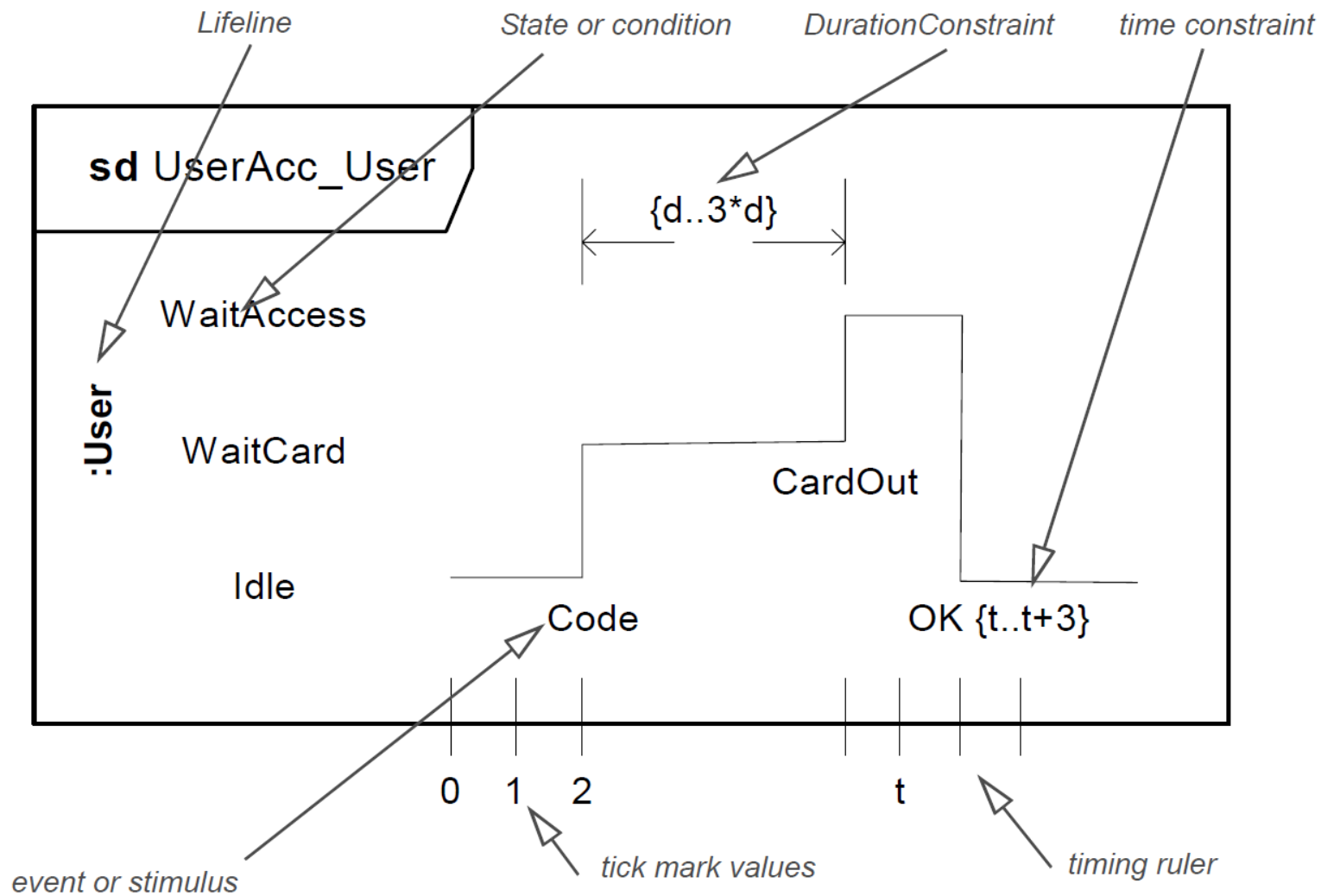


Diagramme de temps

- ◆ Evolution de l'état du système selon un point de vue principalement temporel



Plan

- ◆ Diagrammes fonctionnels
- ◆ Diagrammes statiques
- ◆ Diagrammes dynamiques
- ◆ *Diagrammes d'implémentation*
 - ◆ *De paquetages*
 - ◆ *De déploiement*

Diagrammes d'implémentation

- ◆ Mise en place de l'application sur un environnement
- ◆ Diagramme de paquetages
 - ◆ Description de l'organisation du code des applications
 - ◆ Utile au programmeur
- ◆ Diagramme de déploiement
 - ◆ Description du déploiement sur un réseau
 - ◆ Aspects liés à la topologie, à l'intégration des systèmes et aux communications

Diagramme de paquetages

- ◆ Regrouper les classes dans des “packages”
- ◆ Disposer d'heuristiques pour regrouper les classes
 - ◆ Heuristique la plus utilisée : la dépendance entre les classes
 - ◆ Une dépendance existe entre 2 éléments si le changement de définition d'un élément peut modifier un changement dans l'autre élément
 - ◆ Dépendances entre classes
 - ◆ Envoi d'un message (appel de méthode)
 - ◆ Une classe fait partie des données d'une autre classe
 - ◆ Une classe mentionne une autre classe comme un paramètre d'une opération
 - ◆ Idéalement, seules les modifications de l'interface de la classe affectent les autres classes

Diagramme de paquetages

- ◆ Exemple de diagramme de paquetages
 - ◆ Note : les classes contenues dans les packages ne sont pas représentées ici

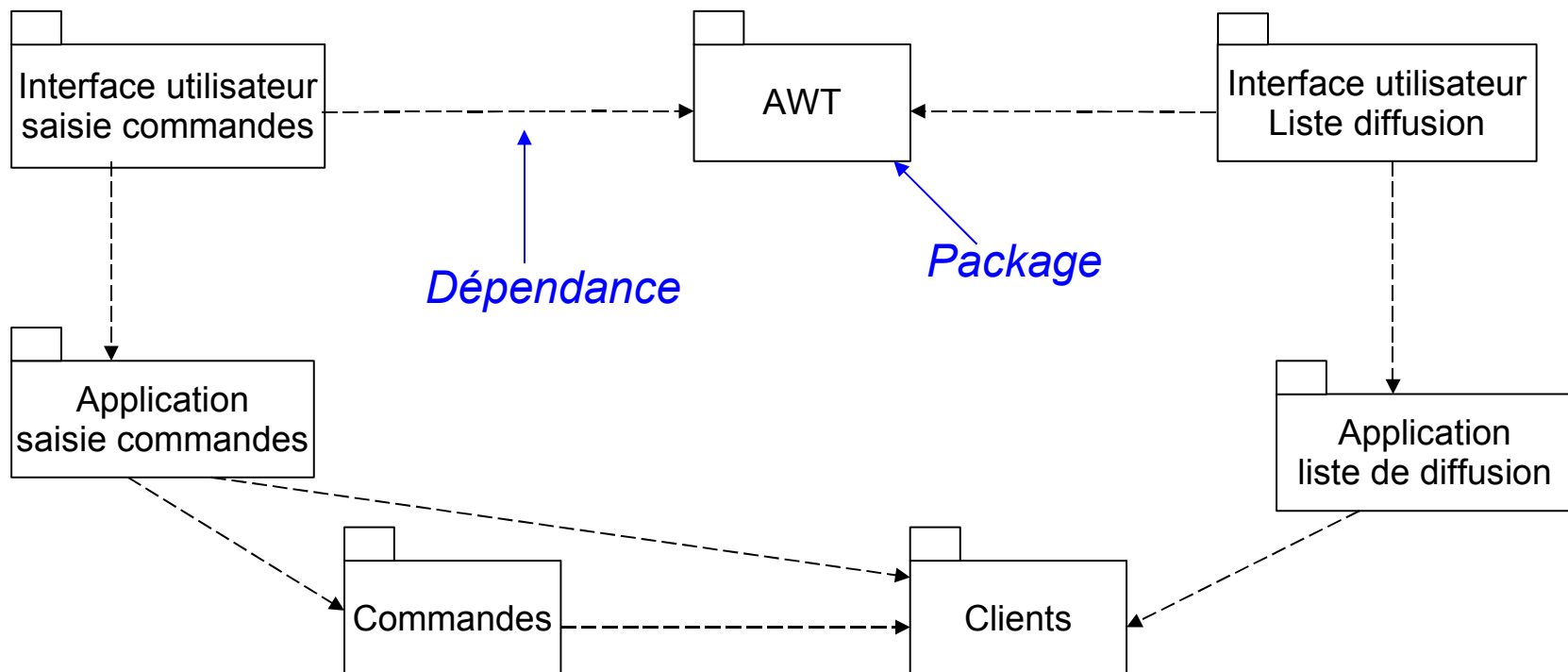
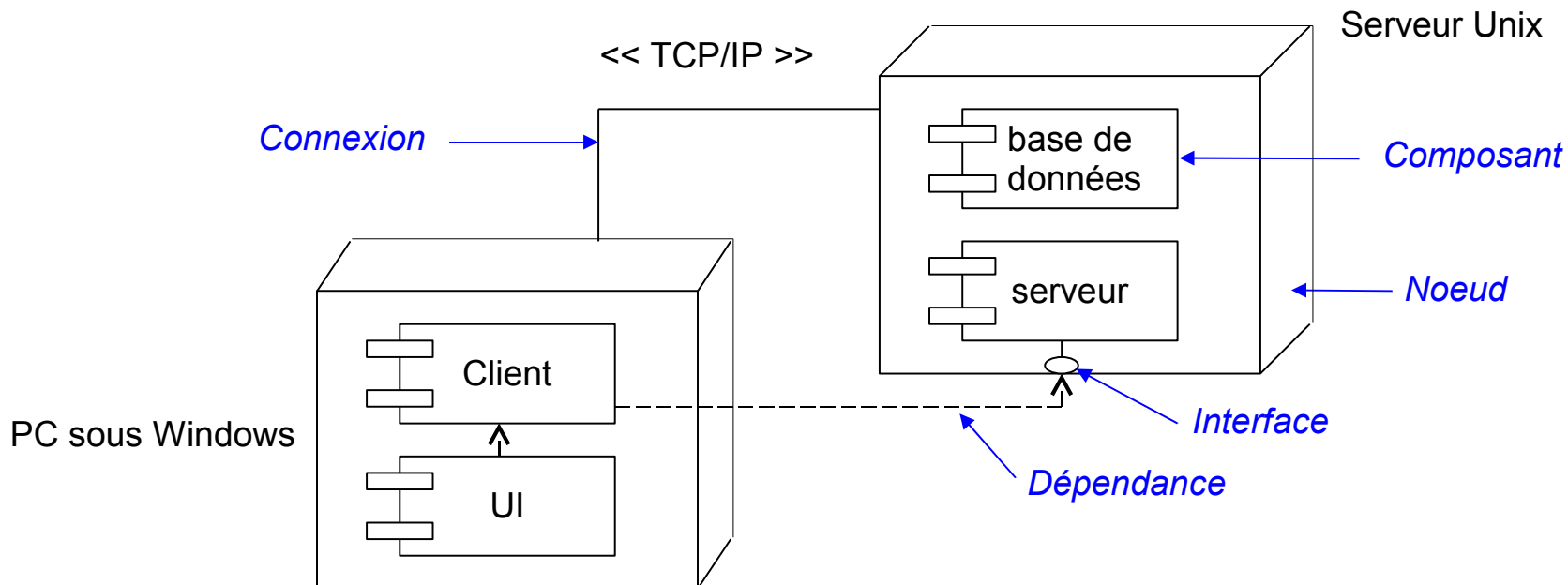


Diagramme de déploiement

- ◆ Diagramme de déploiement
 - ◆ Relation entre le logiciel et le matériel
 - ◆ Placement des composants et objets dans le système réparti
 - ◆ Noeud = unité informatique (périphérique, capteur, mainframe, PC,...)
 - ◆ Connexion
 - ◆ Composant = module physique de code



Conclusion sur UML

◆ Avantages d'UML

- ◆ Un certain consensus autour de l'utilisation d'UML : standard de fait dans l'industrie
- ◆ Notation avec une syntaxe très riche, tout en restant intuitive
- ◆ Intégration dans des ateliers de génie logiciel avec production de squelettes de codes et autres transformations automatiques des modèles
- ◆ Langage de contraintes OCL pour spécifications précises à utiliser en complément

◆ Inconvénients d'UML

- ◆ Notation majoritairement graphique pouvant se révéler insuffisante ou trop chargée d'un point de vue expressivité
- ◆ Sémantique floue ou mal définie pour certains types de diagrammes
- ◆ Lien parfois difficile entre les vues et diagrammes d'une même application