Chapitre 3: Analyse et Modélisation d'un Système d'information avec UML-OCL

3ème Année Ingénieur

Semestre: S6

Ecole Supérieure en Informatique

Plan

- I. Introduction
- II. Modèles contextuels
- III. Modèles d'interaction
  - Modélisation des cas d'utilisation
  - Diagramme de séquence
- IV. Modèles Structurels
  - Diagramme de Classes
  - Diagramme d'Objets
- V. Modèles comportementaux (dynamiques)
  - Diagramme d'Etat et de Transition
  - Diagramme d'activité
- VI. Langage OCL
  - Introduction
  - Topologie des Contraintes
  - Types et opérations utilisables dans les expressions OCL

Références

1. Roger Pressman, SOFTWARE ENGINEERING: A PRACTITIONER'S APPROACH, EITH EDITION, Published by McGraw-Hill
2. Ian Sommerville SOFTWARE ENGINEERING, Ninth Edition, Addison Wisley
3. John W. Satzinger, Robert B. Jackson, Stephen D. Burd SYSTEMS ANALYSIS AND DESIGN IN A CHANGING WORLD, Sixth Edition, CENPAGE
4. P. André A. Vaillée Développement de logiciels avec UML2 et OCL

I. Introduction

❑ La modélisation de système est le processus de développement de modèles abstraits d'un système, chaque Modèle présentant une vue ou une perspective différente de ce système.

❑Modélisation du système a généralement consisté à représenter le système en utilisant une représentation graphique qui est maintenant presque toujours basée sur des notations dans le Langage de modélisation unifié (UML).

❑ Cependant, il est également possible de développer des modèles formels (mathématiques) comme une spécification détaillée du système.

I. Introduction (1)

❑ Ces modèles sont utilisés à la fois pour : des systèmes existants et des Systèmes à développer:

1. les modèles du système existant sont utilisés lors de l'ingénierie des exigences. Ils aident à clarifier ce que le système existant peut et peut servir de base pour discuter ses points forts et ses faiblesses. Cela conduit alors à des exigences pour le nouveau système.

2. Les modèles du nouveau système sont utilisés lors de l'ingénierie des exigences pour aider expliquer les exigences proposées aux autres intervenants du système.

❑ Les Ingénieurs utilisent ces modèles pour discuter des propositions de conception et documenter le système pour sa mise en œuvre.

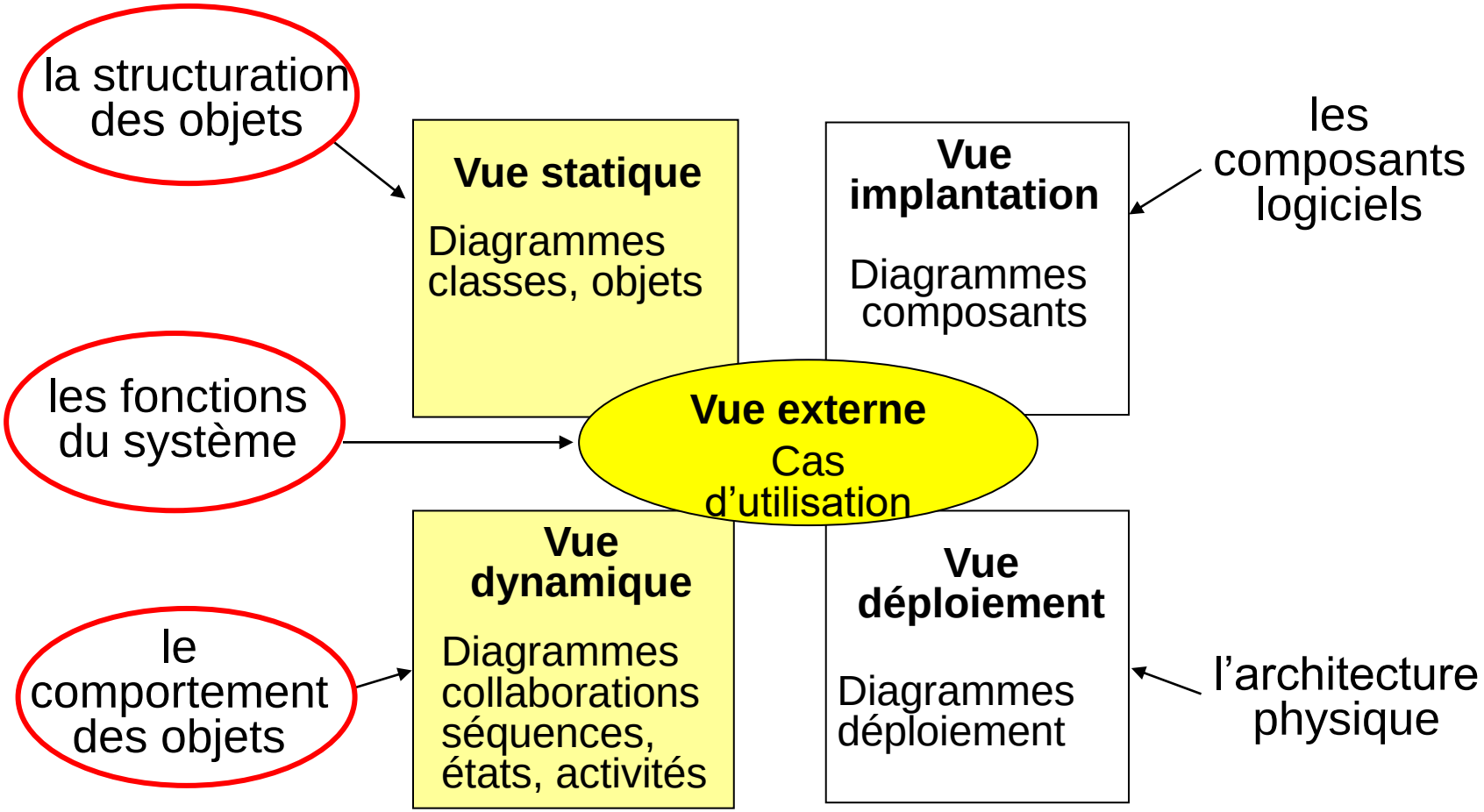
❑ Dans un processus d'ingénierie (IDM) piloté par un modèle, il est possible de générer une implémentation totale ou partielle du système à partir du modèle système.

I. Introduction (2)

❑ Vous pouvez développer différents modèles pour représenter le système sous différentes perspectives. Par exemple:

1. Une perspective externe, où vous modélisez le contexte ou l'environnement du système.
2. Une perspective d'interaction où vous modélisez les interactions entre un système et son environnement ou entre les composants d'un système.
3. Un point de vue structurel, où vous modélisez l'organisation d'un système ou la structure des données traitées par le système.
4. Une perspective comportementale, où vous modélisez le comportement dynamique du système et comment il répond aux événements.

❑ Ces points de vue ont beaucoup en commun avec l'approche en 4+1 vues de Krutchen (voir la référence [4]


la structuration des objets
Vue statique
Diagrammes classes, objets
Vue implantation
Diagrammes composants
les composants logiciels
les fonctions du système
Vue externe
Cas d'utilisation
le comportement des objets
Vue dynamique
Diagrammes collaborations séquences, états, activités
Vue déploiement
Diagrammes déploiement
l'architecture physique

II. Modèles contextuels

❑À un stade précoce de la spécification d'un système, vous devez décider des limites du système

- ➢ Cela implique de travailler avec les acteurs du système pour décider des fonctionnalité qui devraient être inclus dans le système et ce qui est fourni par l'environnement du système.

❑ Vous pouvez choisir un support automatisé pour certains processus métiers qui devraient être implémentés, mais d'autres devraient être des processus manuels ou pris en charge par différents Systèmes.

❑ Vous devriez envisager des chevauchements possibles en fonctionnalité avec des Systèmes et décider où les nouvelles fonctionnalités doivent être mises en œuvre.

❑ Ces décisions devrait être faite au début du processus pour limiter les coûts du système et le temps nécessaire pour comprendre les exigences et la conception du système.

II. Modèles contextuels (1)

- Dans certains cas, la limite entre un système et son environnement est relativement clair.

- Par exemple, lorsqu'un système automatisé remplace un système manuel par un Système informatisé, l'environnement du nouveau système est habituellement le même que le L'environnement du système existant.

Dans d'autres cas, il y a plus de souplesse et vous décider de ce qui constitue la frontière entre le système et son environnement pendant Le processus d'ingénierie des exigences.

II. Modèles contextuels (2)

❑Par exemple, disons que vous développez la spécification pour un **Système de gestion de patients en soins psychiatriques (SG-PSP)** . Ce système est destiné à gérer des informations sur Les patients qui fréquentent les cliniques de psychiatriques et les traitements qui ont été prescrits.

❑ En élaborant les spécifications pour ce système, vous devez décider si le système devrait se concentrer exclusivement sur la collecte d'informations sur les consultations (en utilisant d'autres systèmes pour recueillir des informations personnelles sur les patients)

❑ Ou si cela devrait recueillent également des informations personnelles sur les patients. L'avantage de s'appuyer sur d'autres systèmes pour les informations sur les patients, vous évitez de dupliquer les données. L'inconvénient majeur, cependant, l'utilisation d'autres systèmes peut rendre plus lent l'accès à l'information. Si ces systèmes ne sont pas disponibles, alors le SG-PSP ne peut pas être utilisé.

II. Modèles contextuels (3)

- La définition d'une limite de système n'est pas un jugement sans valeur.

- Les préoccupations sociales et organisationnelles peuvent signifier que la position d'une limite de système peut être déterminé par des facteurs non techniques.

- Par exemple, une limite de système peut être délibérément positionnée de sorte que le processus d'analyse puisse être effectué sur un seul site;

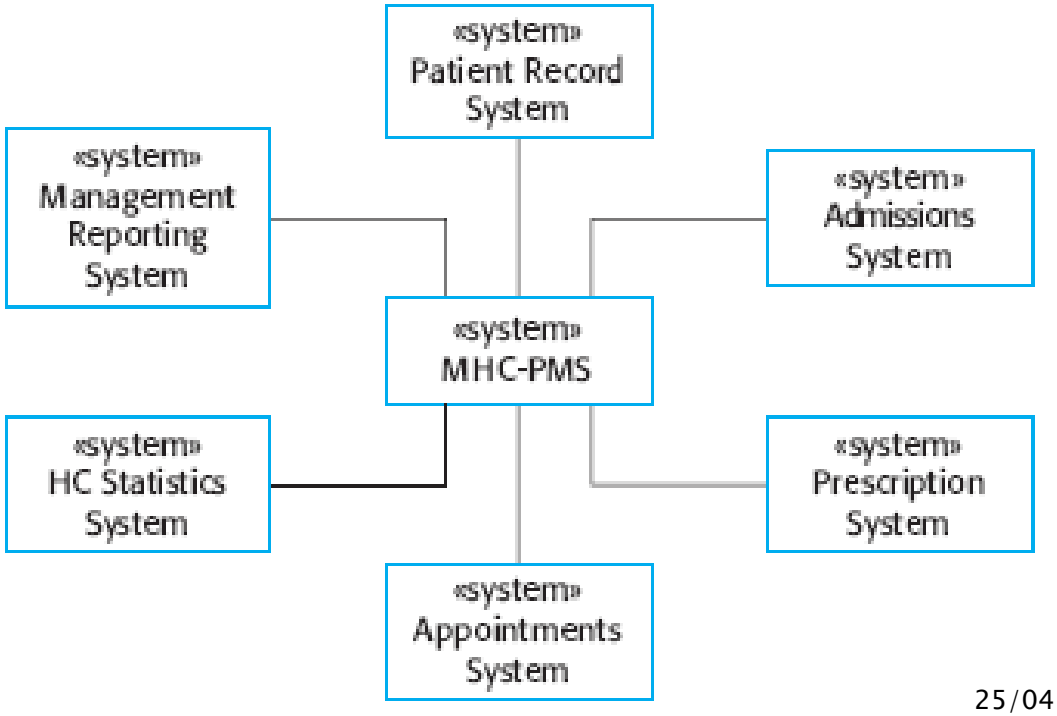
  - il peut être choisi pour qu'un gestionnaire particulièrement difficile ne soit pas nécessairement consulté; cela pourrait être positionné de sorte que le coût du système augmente et que la division de développement du système doit donc se développer pour concevoir et mettre en œuvre le système.

- Une fois que certaines décisions sur les limites du système ont été faites, une partie de l'activité d'analyse est la définition de ce contexte et les dépendances qu'un système à son environnement.

II. Modèles contextuels (4)

❑ La figure ci-dessous est un modèle de contexte simple qui montre le système d'information du patient et les autres systèmes dans son environnement.

❑De cette figure, vous pouvez voir que le PG-PSP (en anglais: **MHC-PMS: Mental Health Care-Patient Management System**) est connecté à un système de rendez-vous et à Système d'enregistrement des patient avec lequel il partage des données. Le système est également connecté à des systèmes pour les rapports de gestion et l'allocation des lits d'hôpital et un système de statistiques qui collecte les Informations pour la recherche. Enfin, il utilise un système de prescription pour générer Des Prescriptions des médicament pour les patients.



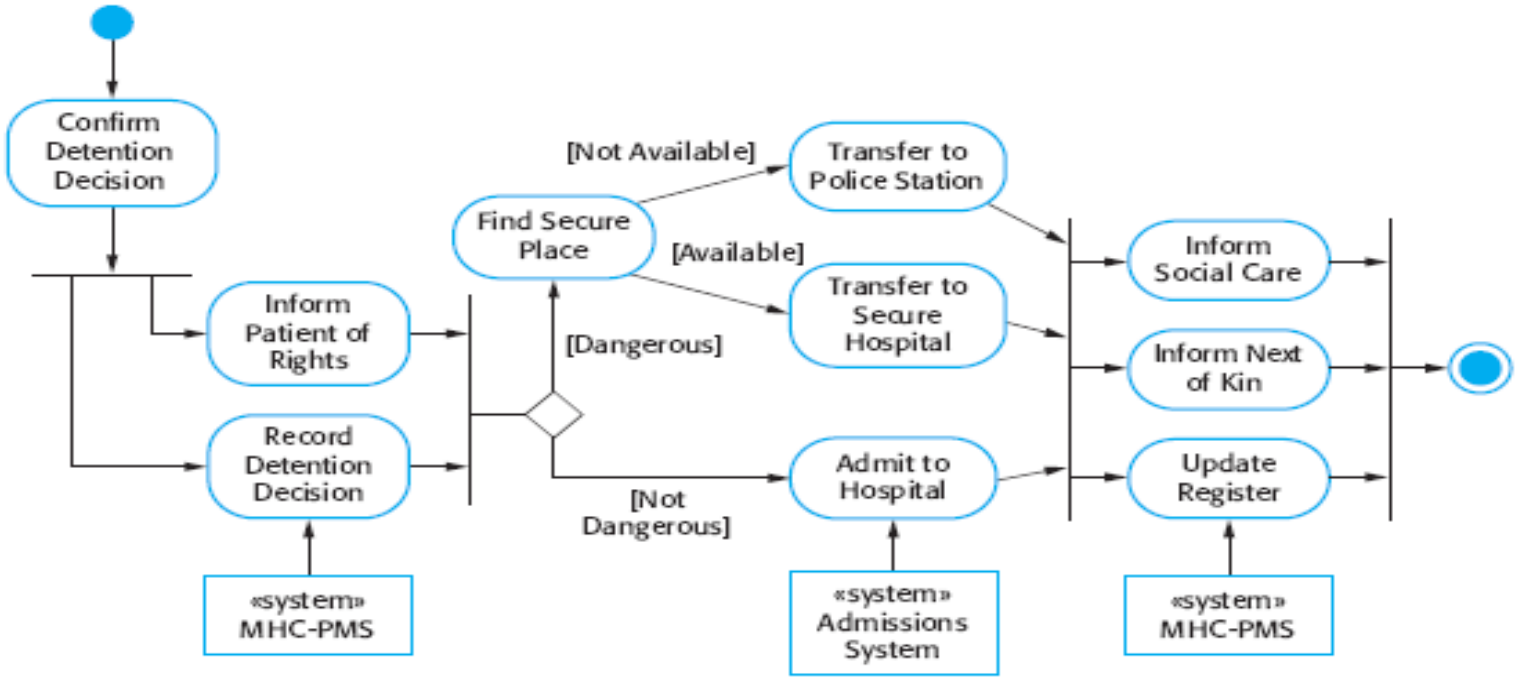
II. Modèles contextuels (5)

❑Les modèles de contexte montrent normalement que l'environnement comprend plusieurs autres systèmes automatismes.

❑ Cependant, ils ne montrent pas les types de relations entre les systèmes dans l'environnement et le système qui est spécifié.

❑ Ces relations peuvent affecter les exigences et la conception du système défini et doit être pris en compte.

Par conséquent, des modèles contextuels simples sont utilisés avec d'autres modèles, tels que Modèles de processus opérationnels. Ceux-ci décrivent des processus humains et automatisés dans lesquels des systèmes logiciels particuliers



II. Modèles contextuels (6)

- La figure ci-dessus est un modèle d'un processus système (diagramme d'activité) important qui montre les processus dans lesquels le PG-PSP est utilisé.

- Parfois, les patients souffrant de troubles mentaux, les problèmes de santé peuvent être un danger pour les autres ou pour eux-mêmes. Ils peuvent donc être détenus contre leur volonté dans un hôpital afin que le traitement puisse être administré.

- Cette détention est soumise à des garanties juridiques strictes - par exemple, la décision détenir un patient doit être régulièrement examiné afin que les personnes ne soient pas détenues indéfiniment sans raison valable.

- L'une des fonctions du PG-PSP est de veiller à ce que des garanties sont mises en œuvre.

La figure en question est un diagramme d'activité UML qui est destiné à présenter les activités qui constituent un processus système et le flux de contrôle d'une activité à l'autre.

III. Modèles d'interaction

- Tous les systèmes impliquent une interaction quelconque. Cela peut être l'interaction de l'utilisateur, qui Implique les entrées et les sorties des utilisateurs, l'interaction entre le système en cours de développement et d'autres systèmes ou l'interaction entre les composants du système.

- La modélisation de l'interaction de l'utilisateur est importante car elle permet d'identifier les besoins des utilisateurs.

- Système de modélisation à l'interaction du système souligne les problèmes de communication qui peuvent survenir.

- L'interaction des composants de modélisation nous aide à comprendre si une structure de système proposée est susceptible de fournir les performances et la fiabilité du système requis.

Cette partie couvre deux approches connexes de la modélisation de l'interaction:

1. Modélisation des cas d'utilisation, qui est surtout utilisée pour modéliser les interactions entre un Système et des acteurs externes (utilisateurs ou autres systèmes).

2. Diagrammes de séquence, qui sont utilisés pour modéliser les interactions entre les composants du système , bien que des agents externes puissent également être inclus.

III.1 Modélisation des cas d'utilisation

- La modélisation des cas d'utilisation a été initialement développée par Jacobson et al. (1993) et a été incorporée dans la première version de l'UML (Rumbaugh et al., 1999).

- La modélisation des cas d'utilisation est largement utilisée pour répondre à la description des exigences.

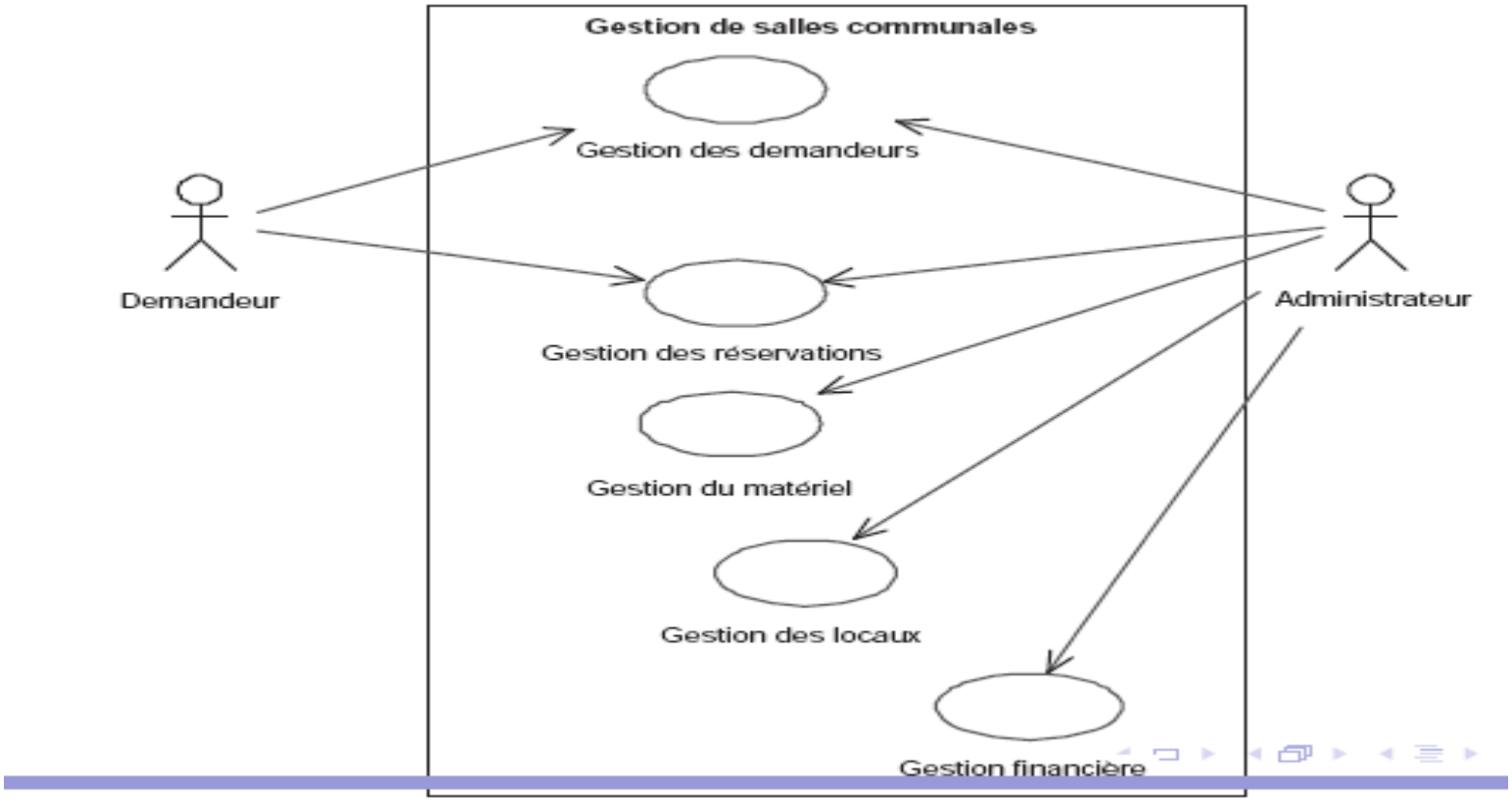
- Un cas d'utilisation peut être pris comme un scénario simple qui décrit ce que l'utilisateur attend d'un système.

- Chaque cas d'utilisation représente une tâche discrète qui implique une interaction externe avec un système.

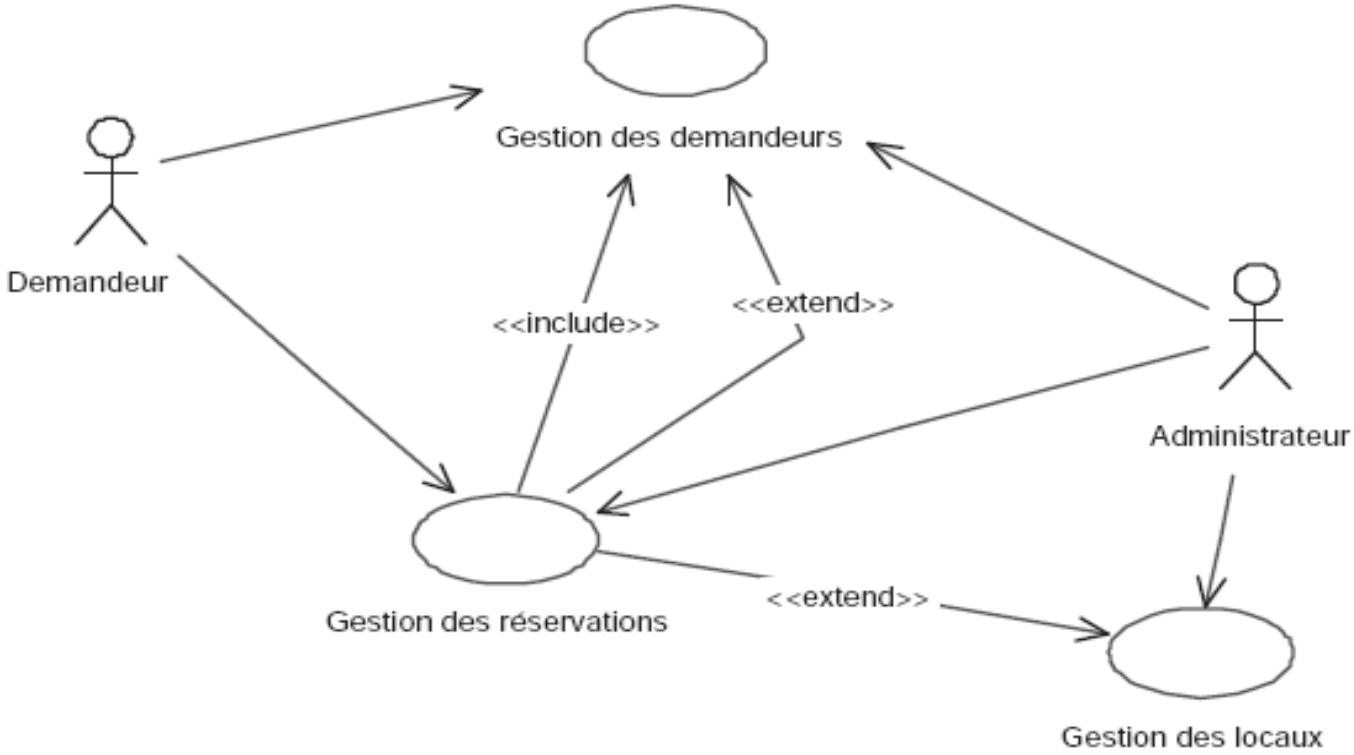
III.1 Modélisation des cas d'utilisation

En résumé :

- Par diagrammes de cas d'utilisation :
  - Acteurs
  - cas d'utilisation
  - relations
- Par cas d'utilisation :
  - descriptions textuelles
  - illustration : scenarios
  - objets : acteurs, système
  - interactions : séquences


Gestion de salles communales
Gestion des demandeurs
Demandeur
Administrateur
Gestion des réservations
Gestion du matériel
Gestion des locaux
Gestion financière

Cas d'utilisation, version préliminaire - Salles



Cas d'utilisation : Gestion des réservations
Acteurs primaires : Demandeur
invariant : Unicité de réservation Une salle n'est pas réservée pour deux demandeurs différents au même moment.
Description La gestion des réservations comprend la réservation des salles, la consultation des réservations, l'annulation des réservations. cas : Réservation Les éléments de la réservations sont saisis et recherches dans la base en fonction de critères donnes : salle, demandeur, matériel, durée, manifestation, date. A tout moment, il est possible de consulter le planning des réservations en cours. Si tous les éléments sont corrects et qu'il n'y a pas de conflit de réservation, le montant est calculé et la réservations confirmée. Le numéro de la réservation est fourni par le système au demandeur.
...

III.2 Diagramme de séquence

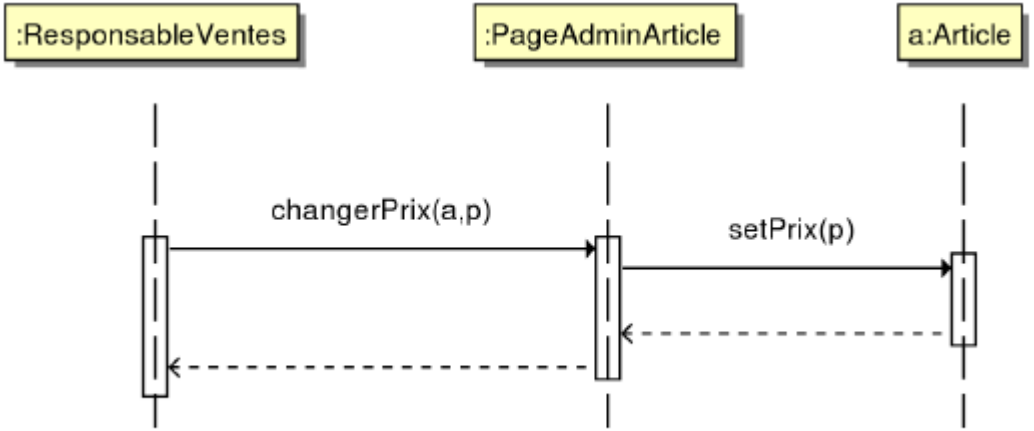
Objectif des diagrammes de séquence

- Les diagrammes de cas d'utilisation modélisent à **QUOI** sert le système, en organisant les interactions possibles avec les acteurs.
- Les diagrammes de classes permettent de spécifier la structure et les liens entre les objets dont le système est composé : ils spécifie **QUI** sera à l'oeuvre dans le système pour réaliser les fonctionnalités décrites par les diagrammes de cas d'utilisation.
- Les diagrammes de séquences permettent de décrire **COMMENT** les éléments du système interagissent entre eux et avec les acteurs.
  - Les objets au cœur d'un système interagissent en s'échangent des messages.
  - Les acteurs interagissent avec le système au moyen d'IHM (Interfaces Homme-Machine).

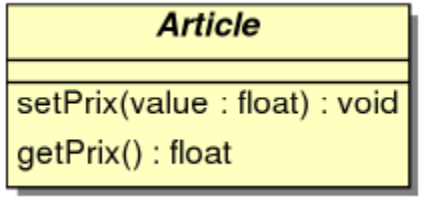
Exemple d'interaction

- Cas d'utilisation

- Diagramme de séquences correspondant :



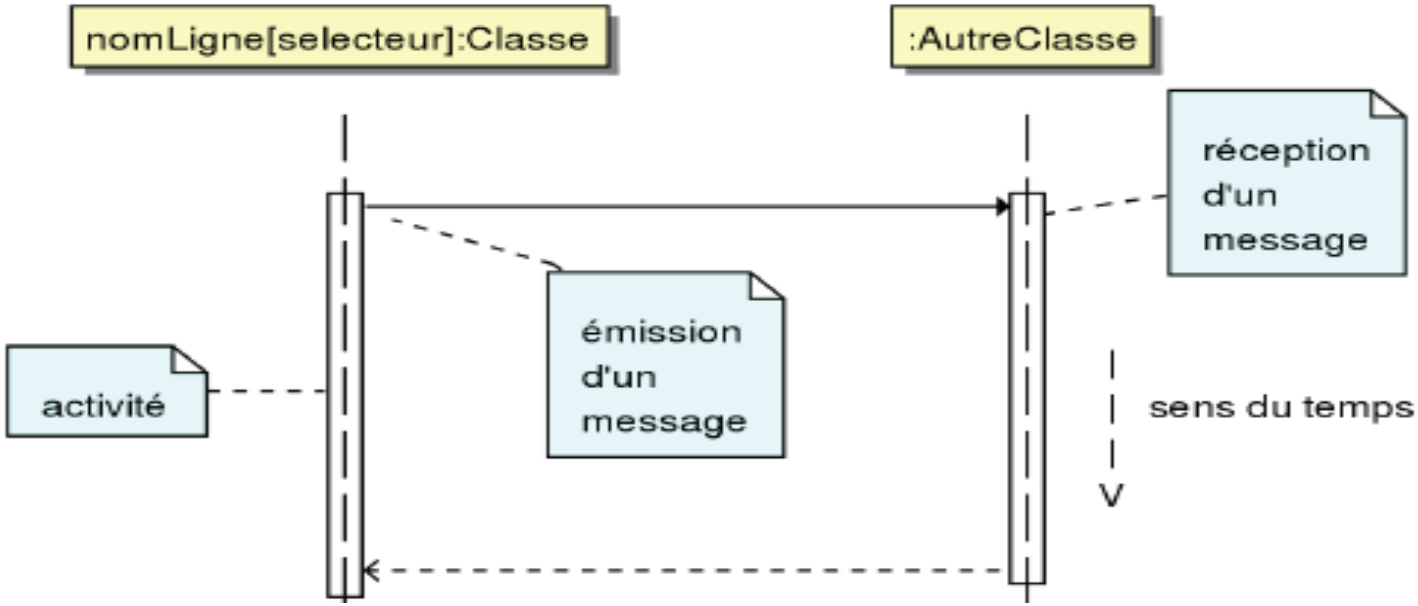
- Opérations nécessaires dans le diagramme de classes :



Ligne de vie

- Une **ligne de vie** se représente par un rectangle, auquel est accroché une ligne verticale pointillée,
- Une ligne de vie représente un participant à une interaction (objet ou acteur).
  nomLigneDeVie [selecteur]: nomClasseOuActeur

Dans le cas d'une collection de participants, un sélecteur permet de choisir un objet parmi n (par exemple objets[2]).



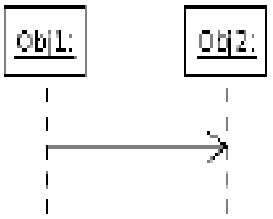
Messages

❑ Les principales informations contenues dans un diagramme de séquence sont les messages échangés entre les lignes de vie, présentés dans un ordre chronologique:

- **Un message** définit une communication particulière entre des lignes de vie.
- Plusieurs types de messages existent, les plus commun sont :
  - l'envoi d'un signal ;
  - l'invocation d'une opération(appel de méthode) ;
  - la création ou la destruction d'un objet ;

❑ Une interruption ou un évènement sont de bons exemples de signaux. Ils n'attendent pas de réponse et ne bloquent pas l'émetteur qui ne sait pas si le message arrivera à destination, le cas échéant quand il arrivera et s'il serra traité par le destinataire. Un signal est, par définition, un message asynchrone.

❑ Graphiquement, un message asynchrone se représente par une flèche en traits pleins et à l'extrémité ouverte partant de la ligne de vie d'un objet expéditeur et allant vers celle de l'objet cible.



– Représentation d'un message asynchrone.

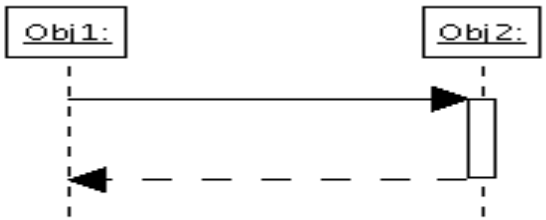
Messages

❑ L'invocation d'une opération est le type de message le plus utilisé en programmation objet. L'invocation peut être asynchrone ou synchrone. Dans la pratique, la plus part des invocations sont synchrones, l'émetteur reste alors bloqué le temps que dure l'invocation de l'opération.

❑ Graphiquement, un message synchrone se représente par une flèche en traits pleins et à l'extrémité pleine partant de la ligne de vie d'un objet expéditeur et allant vers celle de l'objet cible.

❑ Ce message peut être suivi d'une réponse qui se représente par une flèche en pointillé.

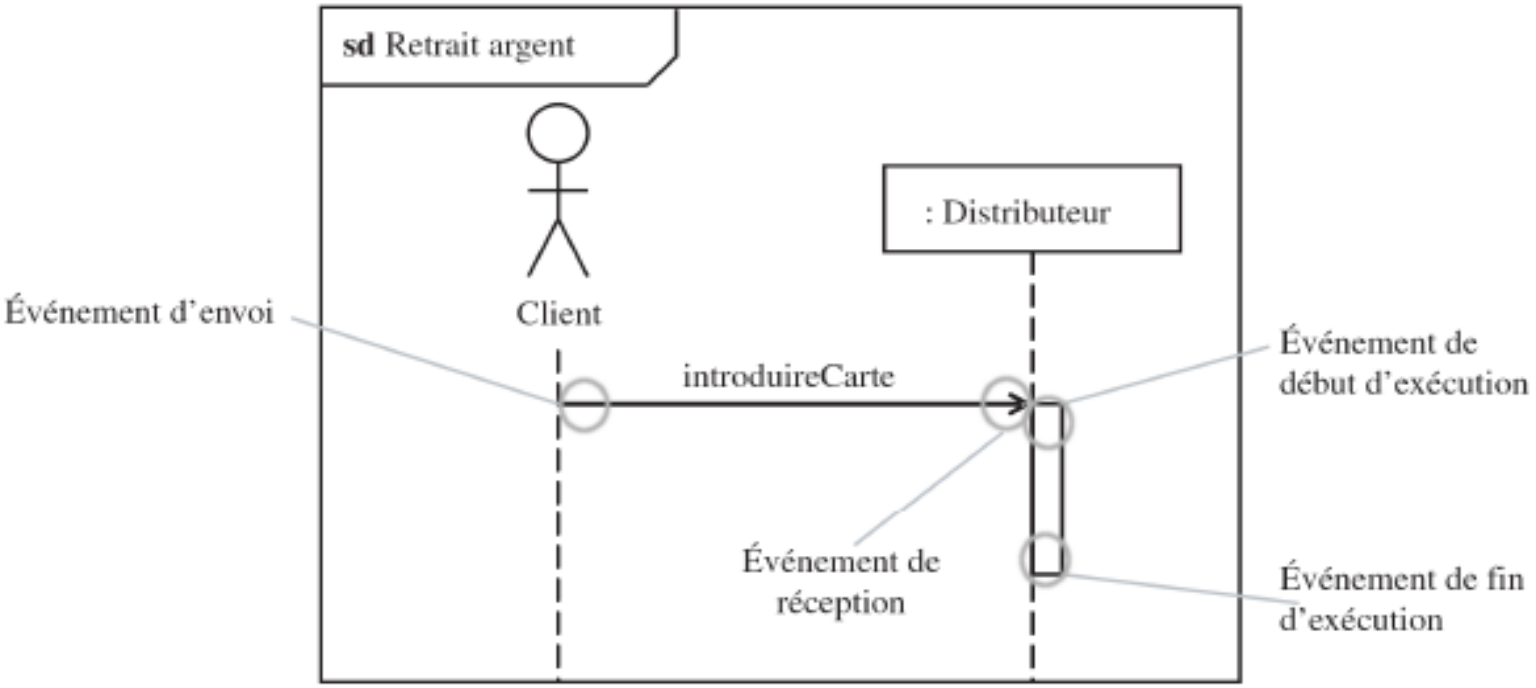
- La création d'un objet est matérialisée par une flèche qui pointe sur le sommet d'une ligne de vie.
- La destruction d'un objet est matérialisée par une croix qui marque la fin de la ligne de vie de l'objet.



Représentation d'un message synchrone.

Messages

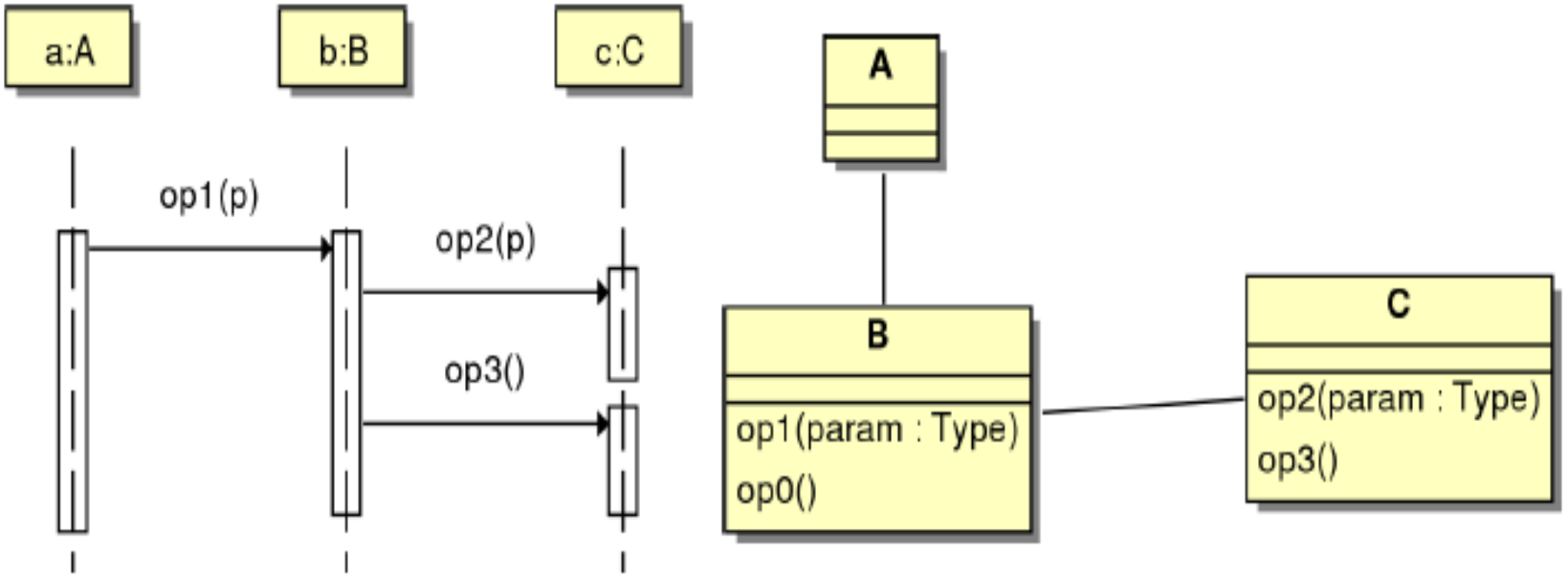
❑ UML permet de séparer clairement l'envoi du message, sa réception, ainsi que le début de l'exécution de la réaction et sa fin.



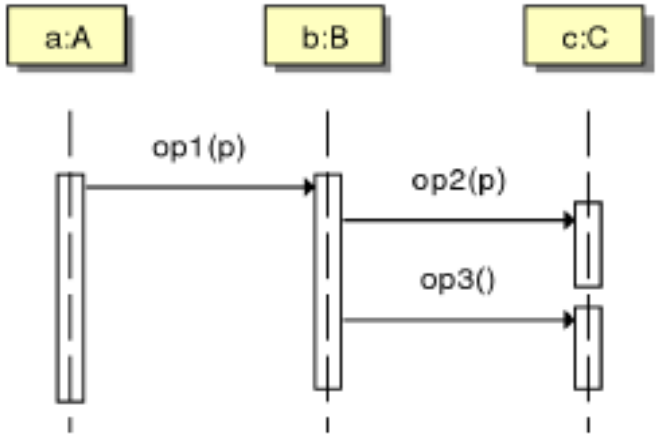
– Les différents évènement correspondant à un message asynchrone.

Correspondance messages / opérations

- Les messages synchrones correspondent à des opérations dans le diagramme de classes.
  - Envoyer un message et attendre la réponse pour poursuivre son activité revient à invoquer une méthode et attendre le retour pour poursuivre ses traitements.



implantation des messages synchrones



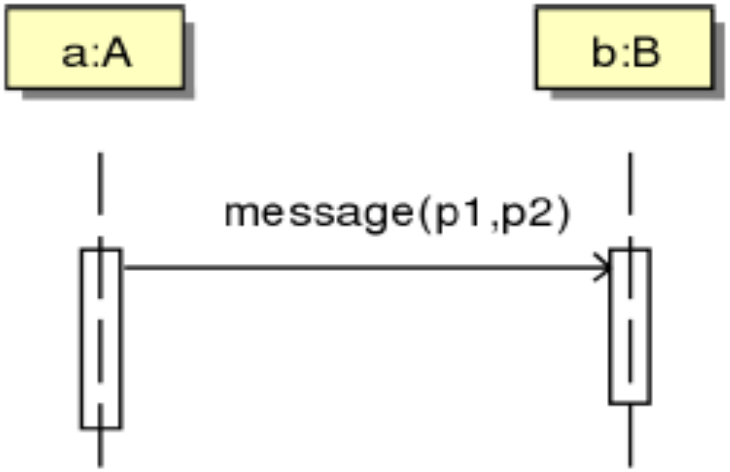

```
class B {
   C c;
   op1(p:Type){
      c.op2(p);
      c.op3();
   }
}
```

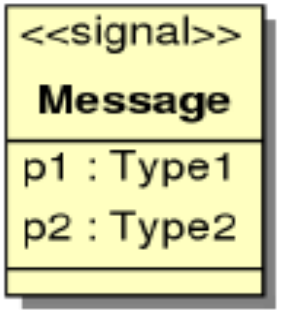
```
class C {
   op2(p:Type){
      ...
   }
   op3(){
      ...
   }
}
```

Correspondance messages / opérations

- Les messages asynchrones correspondent à des signaux dans le diagramme de classes.

  - Les signaux sont des objets dont la classe est stéréotypée signal et dont les attributs (porteurs d'information) correspondent aux paramètres du message.



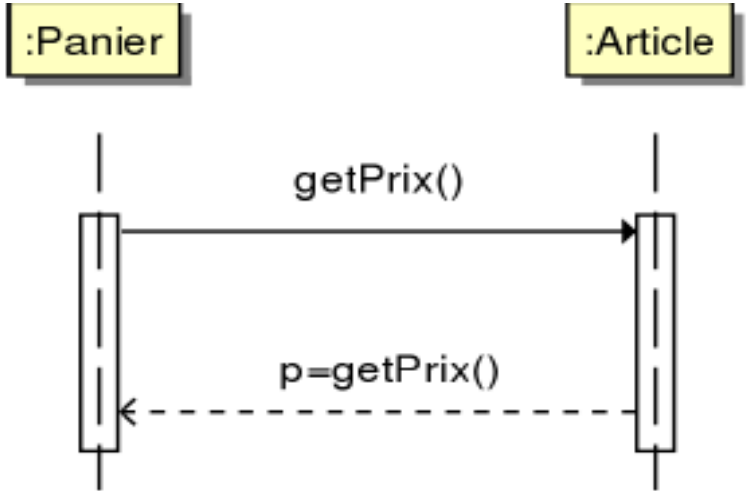


Messages de retour

- Le récepteur d'un message synchrone rend la main à l'émetteur du message en lui envoyant un message de retour

Les messages de retour sont optionnels : la fin de la période d'activité marque également la fin de l'exécution d'une méthode.

Ils sont utilisés pour spécifier le résultat de la méthode invoquée.



- Le retour des messages asynchrones s'effectue par l'envoi de nouveaux messages asynchrones.

Fragments d'interaction combinés

- Un fragment combiné représente des articulations d'interactions. Il est défini par un opérateur et des opérandes.

- L'opérateur conditionne la signification du fragment combiné. Il existe 12 d'opérateurs définis dans la notation UML 2.0.

- Les fragments combinés permettent de décrire des diagrammes de séquence de manière compacte.

- Les fragments combinés peuvent faire intervenir l'ensemble des entités participantes au scénario ou juste un sous-ensemble.

- Un fragment combiné se représente de la même façon qu'une interaction. Il est représenté par un rectangle dont le coin supérieur gauche contient un pentagone où figure le type de la combinaison, appelé opérateur d'interaction.

Fragments d'interaction combinés(1)

❑ Les opérandes d'un opérateur d'interaction sont séparés par une ligne pointillée. Les conditions de choix des opérandes sont données par des expressions booléennes entre crochets ([ ]).

La liste suivante regroupe les opérateurs d'interaction par fonctions :

- les opérateurs de choix et de boucle : **alternative, option, break et loop ;**
- les opérateurs contrôlant l'envoi en parallèle de messages : **parallel et critical region ;**
- les opérateurs contrôlant l'envoi de messages : **ignore, consider, assertion et negative ;**
- les opérateurs fixant l'ordre d'envoi des messages : **weak sequencing , strict sequencing.**

Nous n'aborderons que quelques-unes de ces interactions dans la suite de cette section.

alt : fragments multiple alternatifs (si alors sinon)

opt : fragment optionnel

par : fragment parallèle (traitements concurrents)

loop : le fragment s'exécute plusieurs fois

region : région critique (un seul thread à la fois)

neg : une interaction non valable

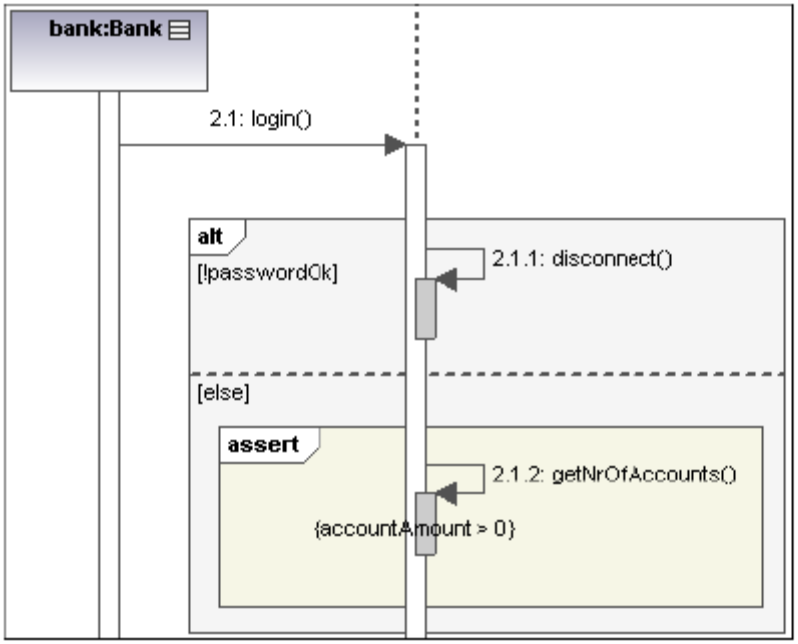
break : représente des scenario d'exception

ref : référence à une interaction dans un autre diagramme

sd : fragment du diagramme de séquence en entier

Opérateurs alt et opt

- L'opérateur alternative, ou alt, est un opérateur conditionnel possédant plusieurs opérandes .
- C'est un peu l'équivalent d'une exécution à choix multiple (condition switch en C++).
- Chaque opérande détient une condition de garde. L'absence de condition de garde implique une condition vraie (true).
- La condition else est vraie si aucune autre condition n'est vraie. Exactement un opérandedont la condition est vraie est exécuté. Si plusieur opérandes prennent la valeur vraie, le choix est non déterministe.
- L'opérateur option, ou opt, comporte une opérande et une condition de garde associée. Le sousfragment s'exécute si la condition de garde est vraie et ne s'exécute pas dans le cas contraire.



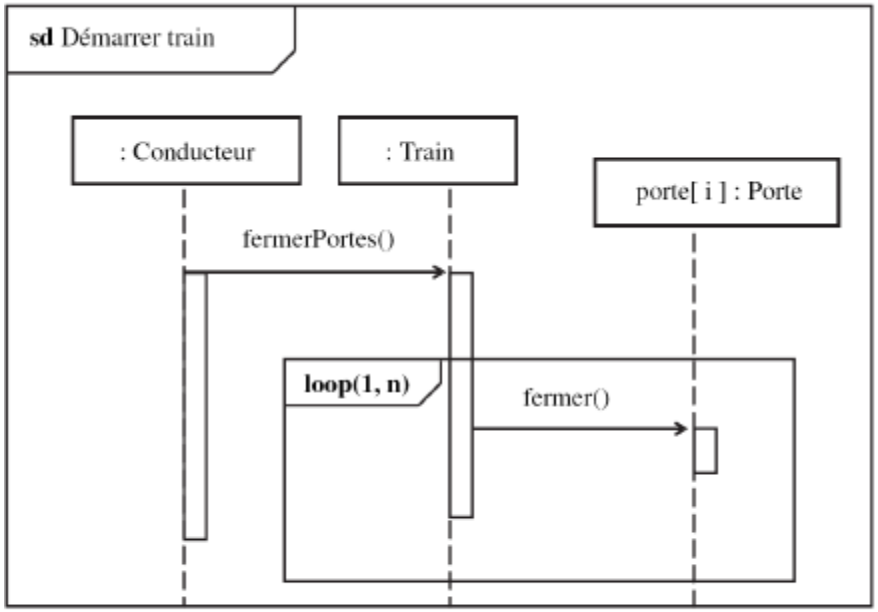
- Représentation d'un choix dans un diagramme de séquence.

Opérateur loop

- Un fragments combiné de type loop possède un sous-fragment et spécifie un compte minimum et maximum (boucle) ainsi qu'une condition de garde.
- La syntaxe de la boucle est la suivante :

loop['('<minInt> [',' <maxInt>] ')']

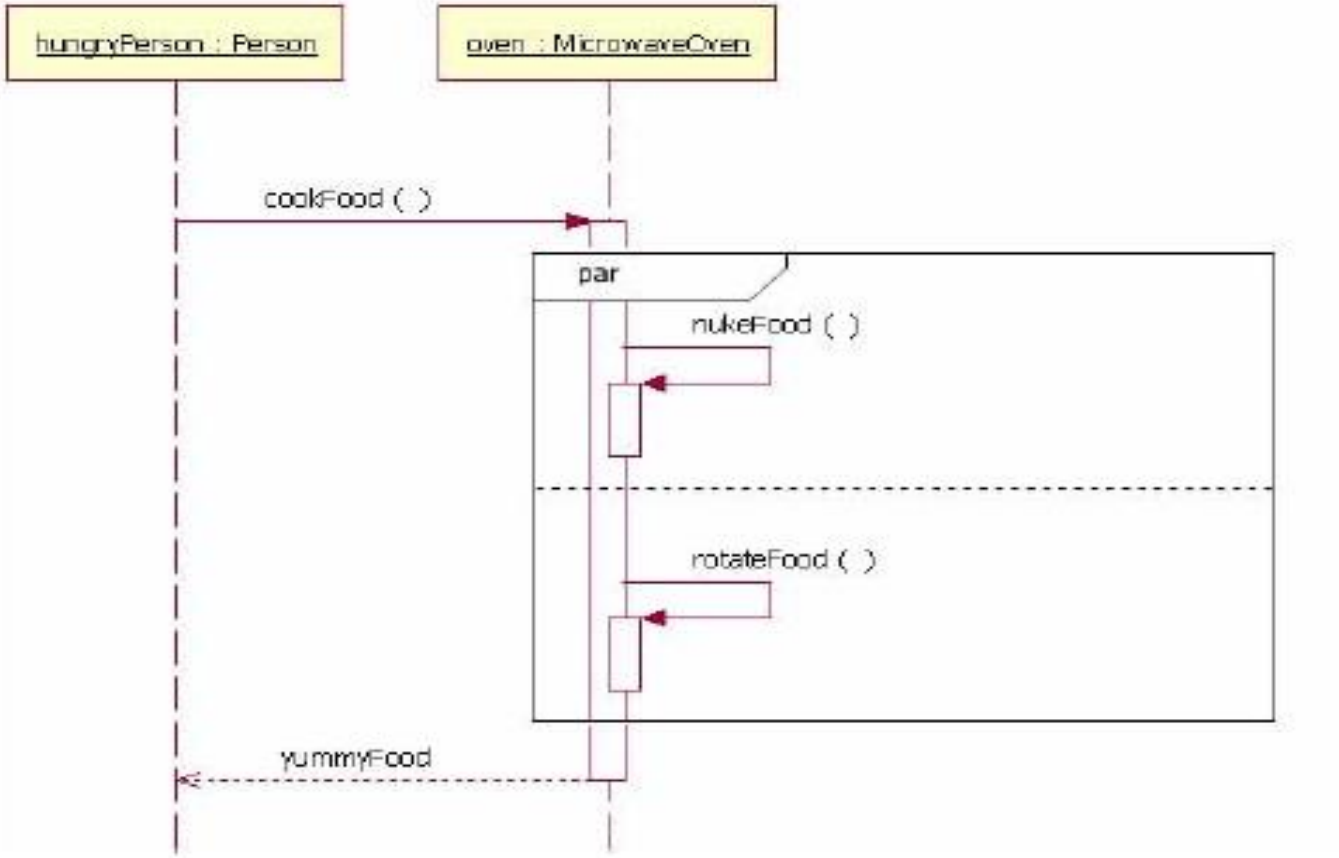
- La condition de garde est placée entre crochets sur la ligne de vie.
- La boucle est répétée au moins minInt fois avant qu'une éventuelle condition de garde booléenne ne soit testée. Tant que la condition est vraie, la boucle continue, au plus maxInt fois.



– Représentation d'une boucle dans un diagramme de séquence.

Opérateur par

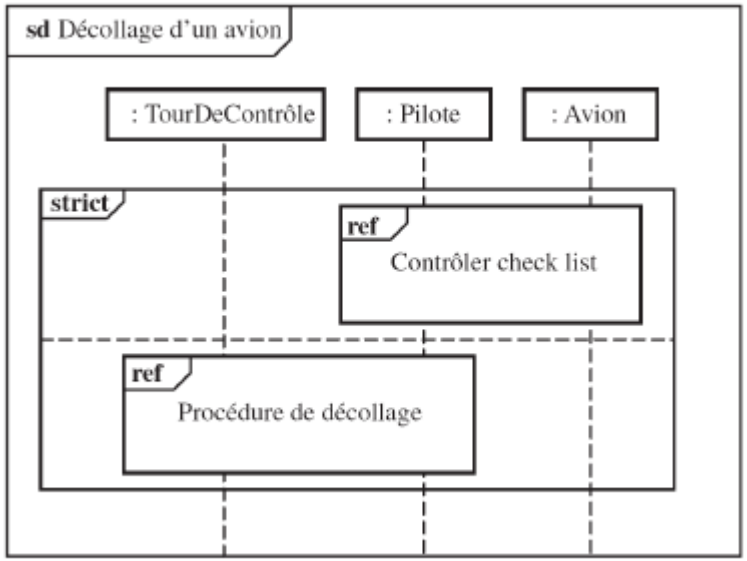
- Un fragments combiné de type **parallel, ou par, possède au moins deux sous-fragments exécutés** simultanément .
- La concurrence est logique et n'est pas nécessairement physique : les exécutions concurrentes peuvent s'entrelacer sur un même chemin d'exécution dans la pratique.



- *MicrowaveOven* est un exemple d'objet effectuant deux tâches en parallèle.

❑Un fragments combiné de type **strict sequencing, ou strict, possède au moins deux sous-fragments.**

❑ Ceux-ci s'exécutent selon leur ordre d'apparition au sein du fragment combiné. Ce fragment combiné est utile surtout lorsque deux parties d'un diagramme n'ont pas de ligne de vie en commun.



IV. Le Modèles Structurels

IV.1 Diagramme de Classes

- Le plus important : objectif final de l'analyse/conception.
- Décrit la structure interne du système, sous forme de **classes** (attributs + opérations) et de relations entre classes. Ne montre pas comment utiliser les opérations = description **statique.**
- Lors de l'analyse :

  → **classes du domaine** (correspondant aux « objets métiers »).
- Lors de la conception :

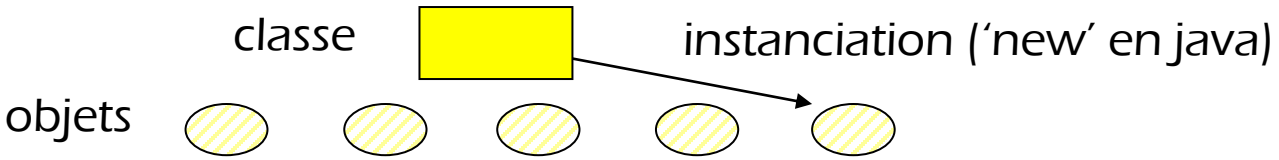
  → ajout des **classes « techniques »** liées aux choix de conception (interfaces utilisateurs, persistance, patrons de conception…).
- Lors de l'implantation :

  → ajout des **classes liées à l'implantation** dans un langage de programmation donné (structures de données spécifiques…).

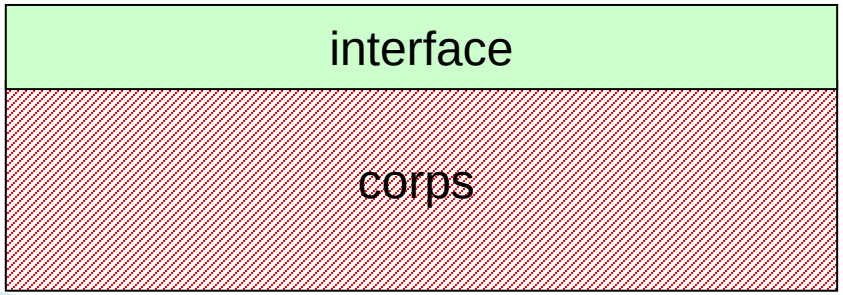
Concept de classe

- Décrit un ensemble d'objets (instances de la classe). Décrit leurs propriétés communes (attributs, opérations, relations).
- **Classe = type + module**

  Type : « fabrique » d'instances (objets) ayant les mêmes propriétés et les mêmes comportements



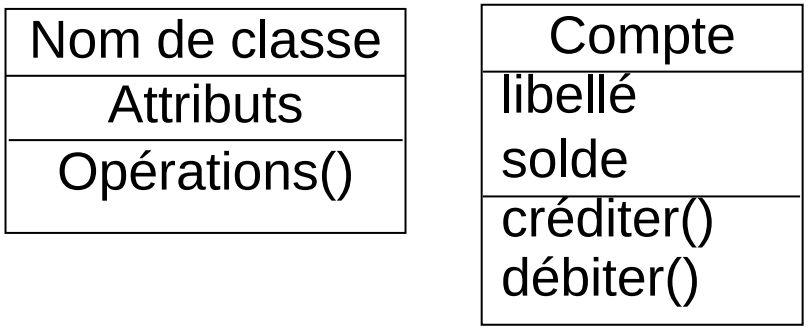
Module : interface visible + corps caché (utilisation possible sans connaître l'implantation; si le corps évolue sans impact sur l'interface => le reste du système n'est pas touché)



profil des opérations visibles (publiques)

implantation cachée (privée)

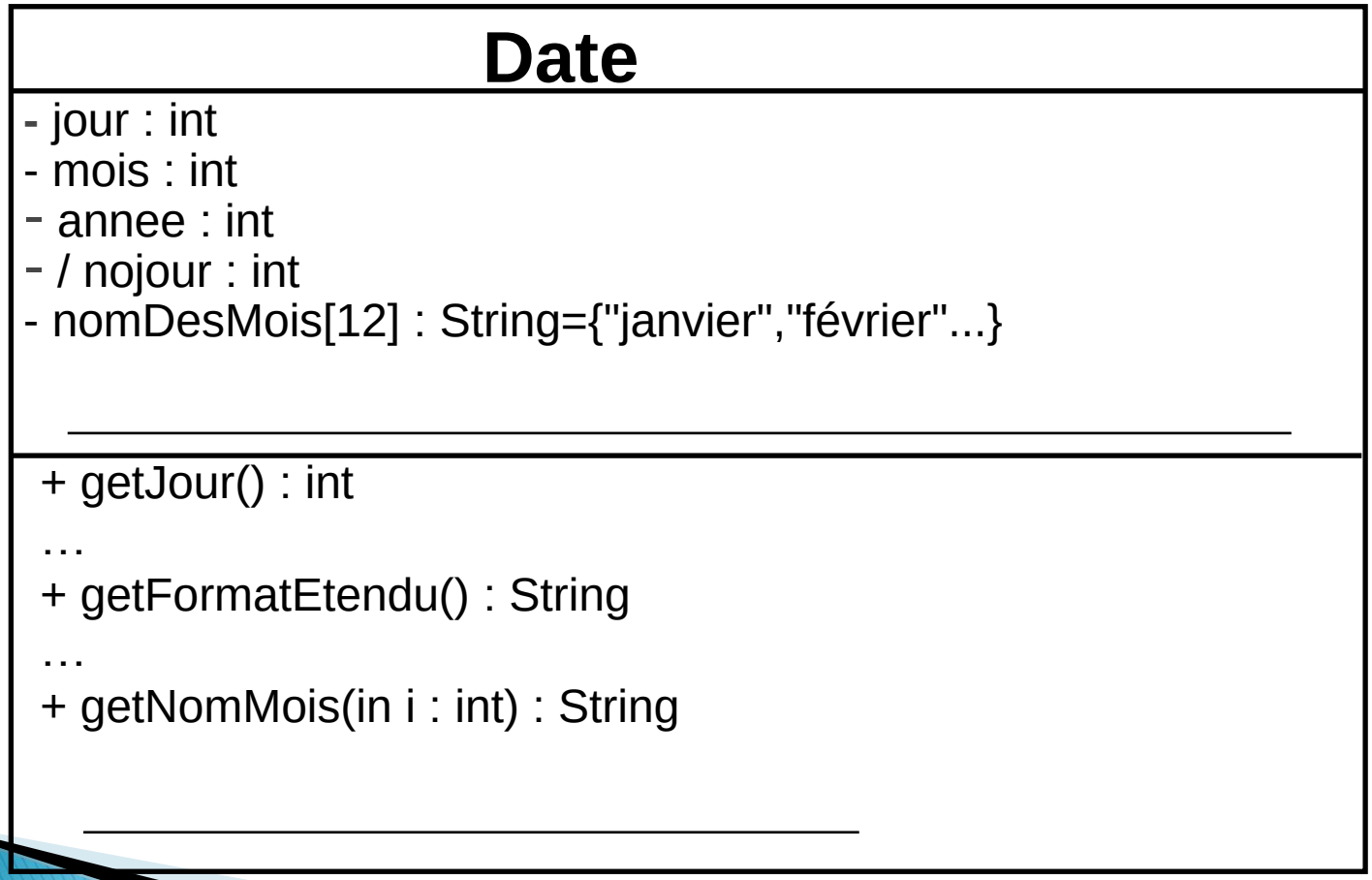
- Notation de base (suffisante au niveau analyse)



- Nombreuses notations supplémentaires (aux niveaux conception et implantation) :
  - Indicateurs de visibilité des attributs et opérations
    + public (visible par tous)
    - privé (visible dans la classe uniquement)
    # protégé (visible dans la classe et ses sous classes)
  - Types des attributs et profils des méthodes

- opérations et méthodes de classe : soulignées
- méthodes abstraites : en italiques
- attributs calculés : notés / attribut : type

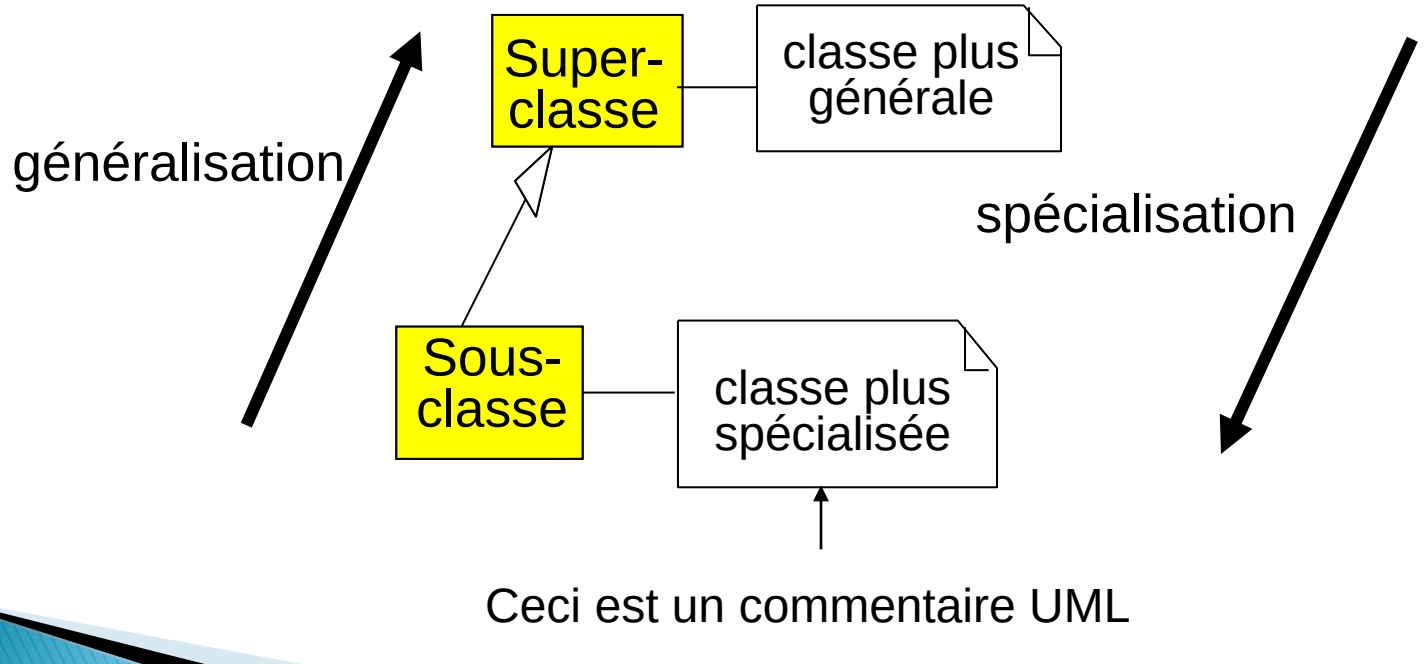
Exemple



Relations entre les Classes

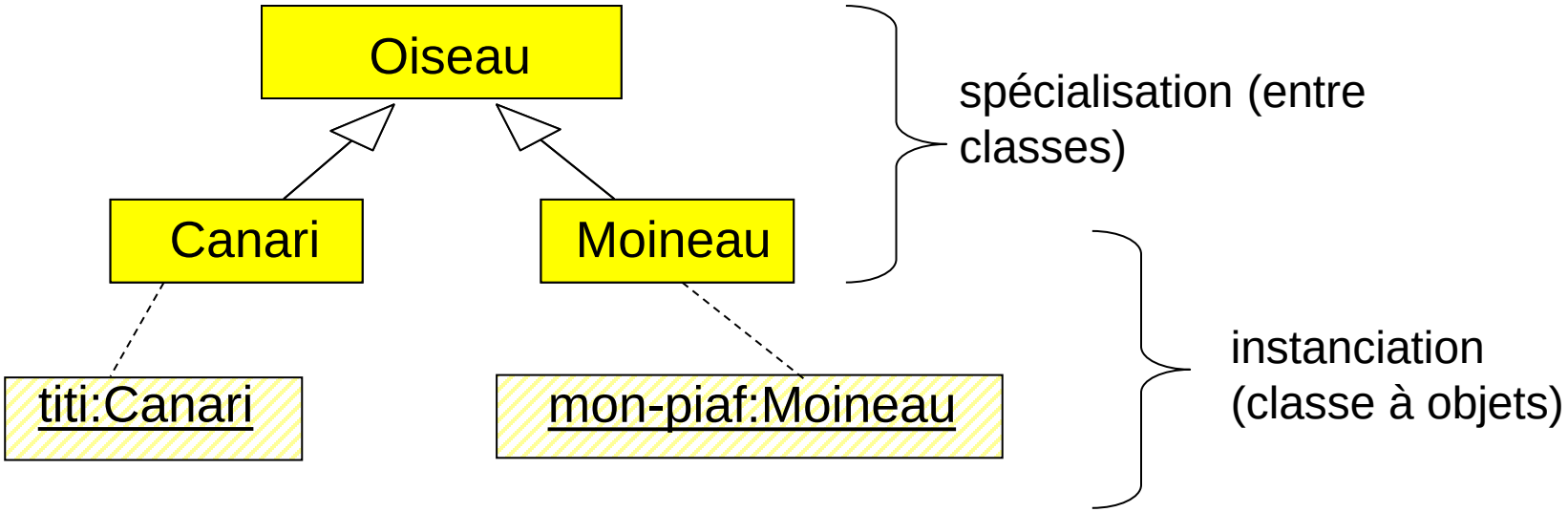
La hiérarchisation des classes

- La hiérarchisation des classes permet de gérer la complexité.
- **Généralisation** : factorisation des éléments communs de classes (attributs, opérations); favorise la réduction de la complexité.
- **Spécialisation** : adapter une classe générale à un cas particulier; favorise la réutilisation.



Remarques

- Ne pas confondre spécialisation et instanciation !

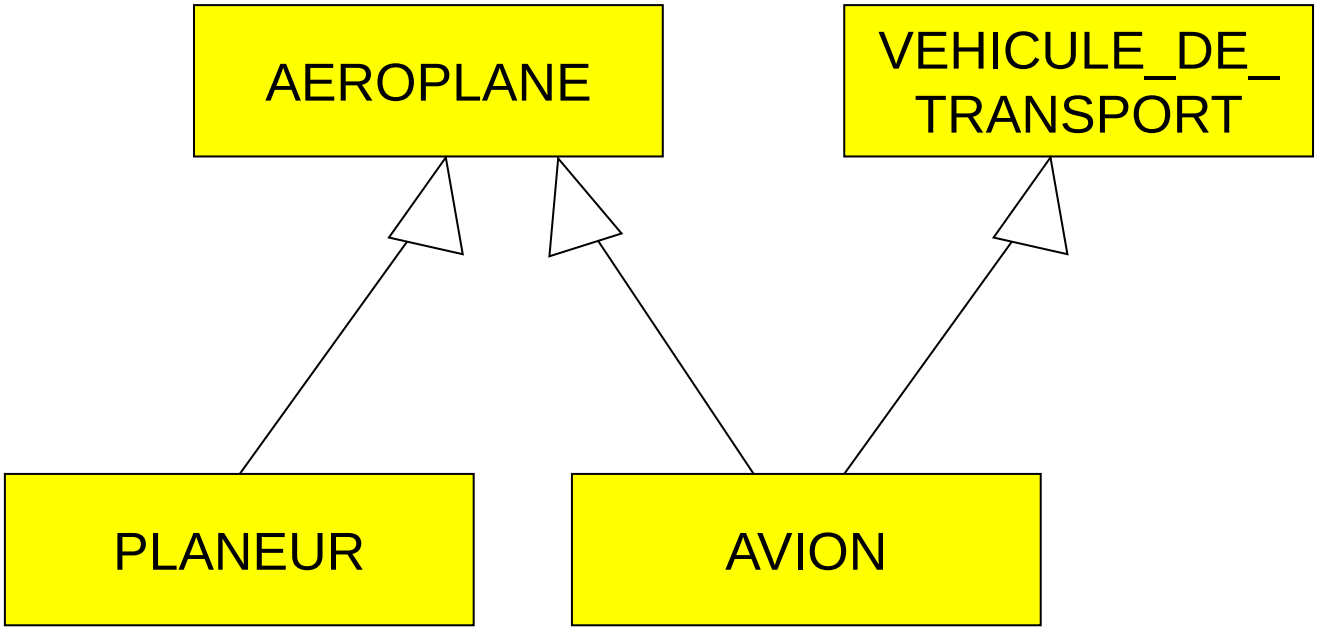


- Notation UML des objets : identificateur:classe ou :classe (objet anonyme)
- Les objets de la classe spécialisée **héritent** de la description des attributs (variables) et des opérations (méthodes) de la super-classe.
- Elles peuvent en **ajouter** d'autres et/ou en **redéfinir** certaines.

Héritage multiple (plusieurs super classes)

Autorisé dans la notation UML.

Exemple :

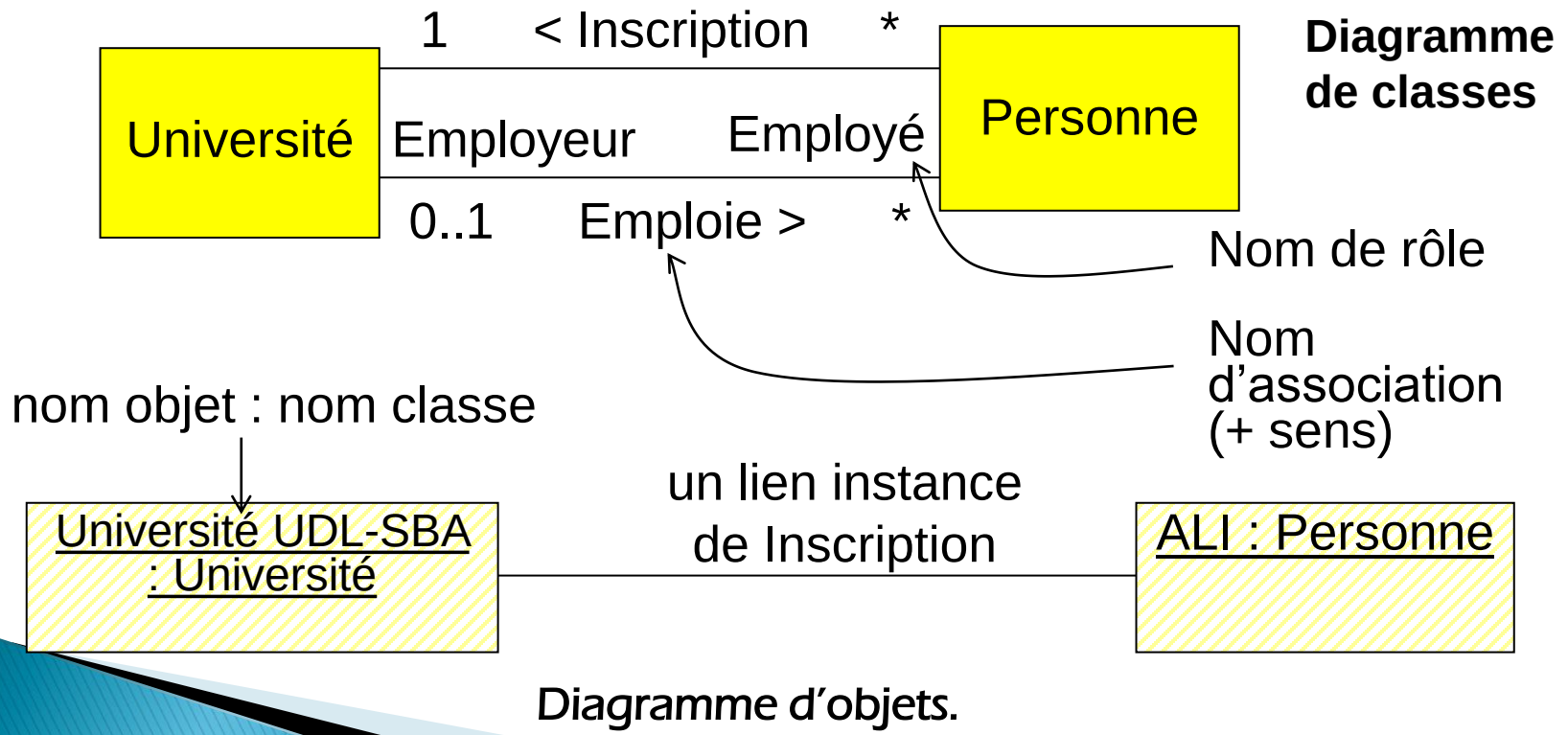


Le concept d'association

Exprime une connexion sémantique entre classes. Décrit un ensemble de **liens** (instances de l'association).

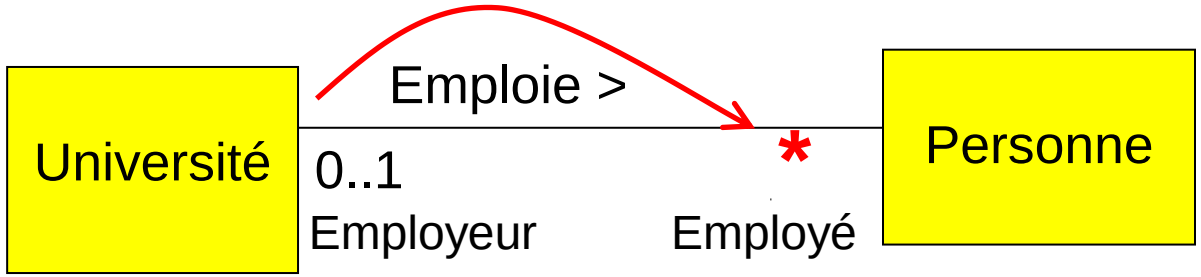
Le concept de rôle caractérise les extrémités.

Les multiplicités (cardinalités) caractérisent le nombre d'instances des classes impliquées dans l'association.



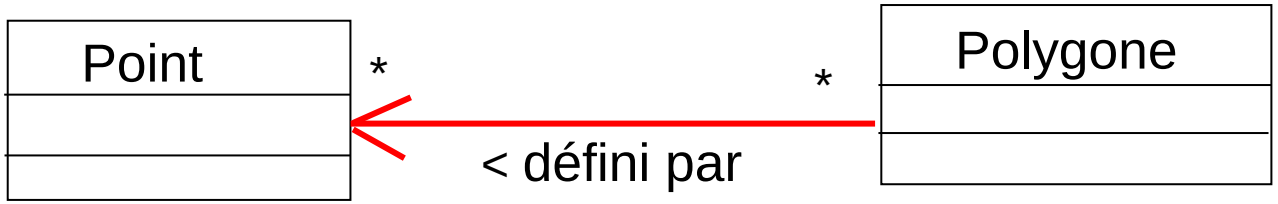
Multiplicités :

1	un et un seul	M..N	de M à N (entiers naturels)
*	plusieurs	1..*	de 1 à plusieurs
0..1	zéro ou un	0..*	de zéro à plusieurs



Sens de parcours et implantation :

On trouvera dans Polygone un attribut correspondant à une collection de Points (tableau, ArrayList... en Java) et dans Point une collection de Polygones.

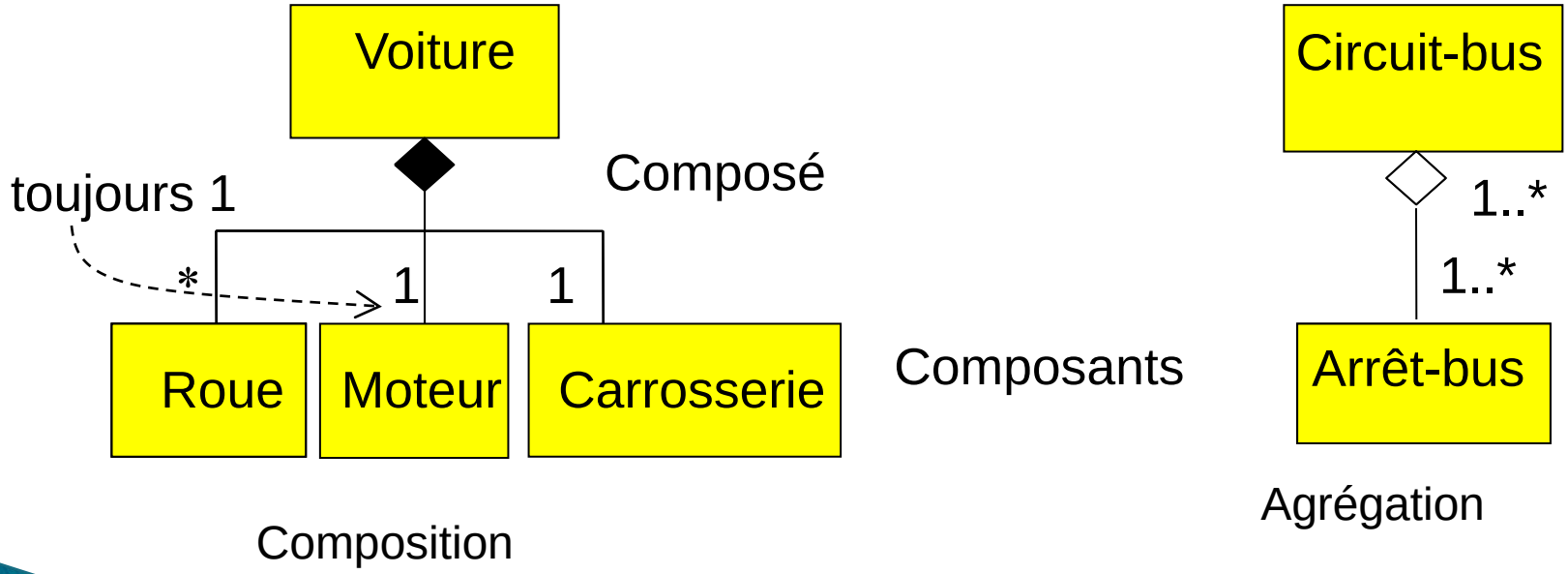


Ici le sens de parcours est limité à Polygone vers Point.
L'association ne sera implantée par une collection que dans la classe Polygone.

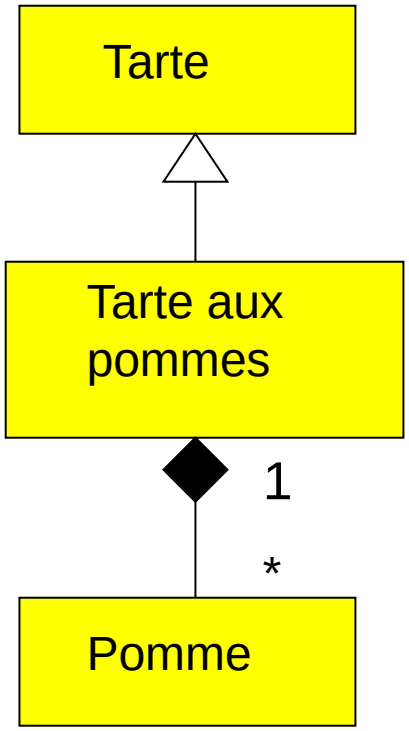
Associations spécialisées

L'agrégation est une association qui décrit une relation d'inclusion entre un tout (l'agrégat) et ses parties.

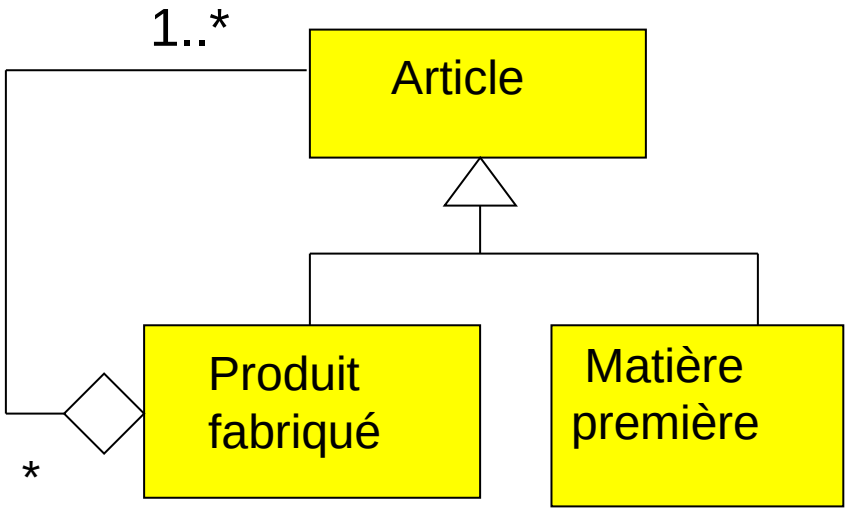
Si les durées de vie des objets sont liées on parle de **composition** (composé/composant), symbolisée par un losange plein du côté du composé; sinon (pour des durées de vie indépendantes) l'agrégation est symbolisée par un losange vide.



Ne pas confondre généralisation/spécialisation et agrégation ! Quand une classe est une spécialisation d'une autre elle est **de même nature** ce qui n'est pas le cas avec l'agrégation. Ces relations peuvent être associées.



Une pomme n'est pas de même nature qu'une tarte !

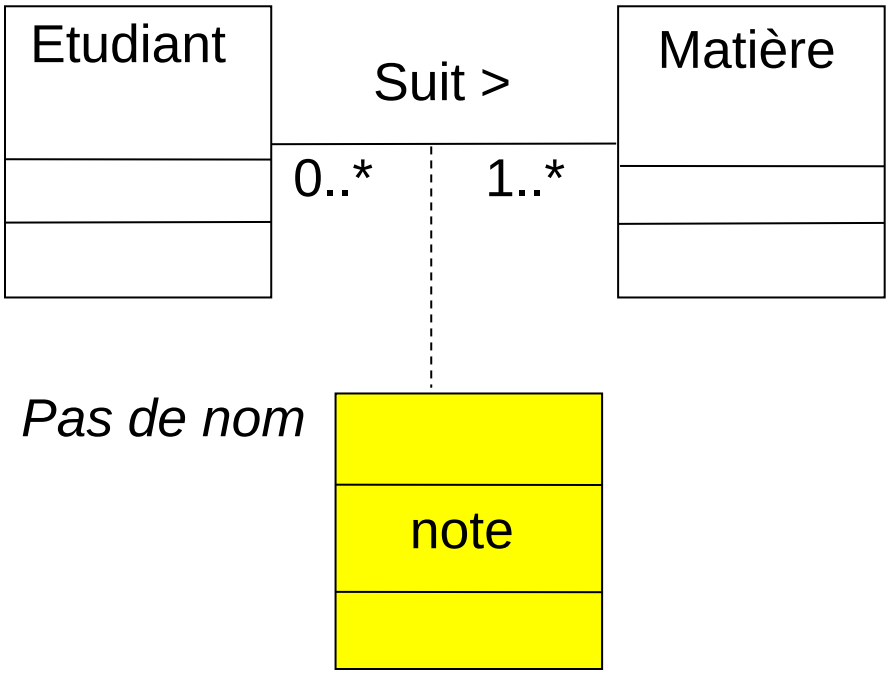


(design pattern « Composite »)

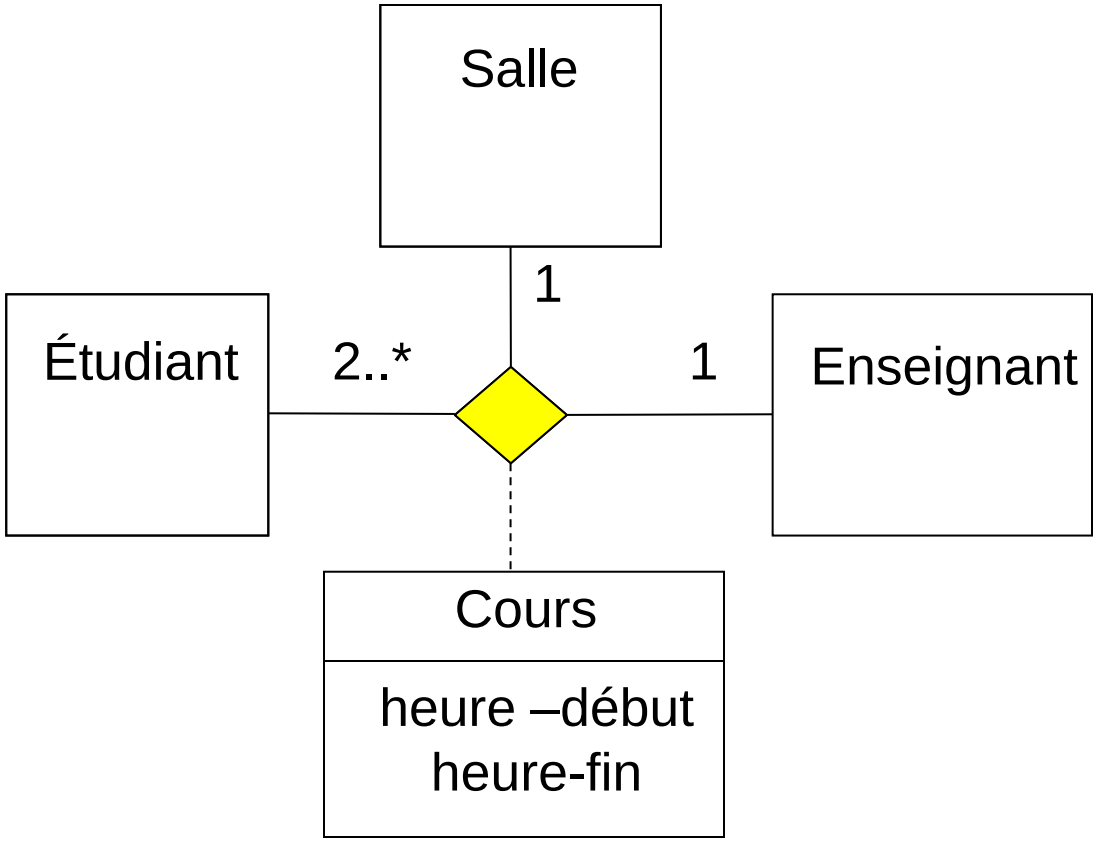
Un article est acheté (matière première) ou fabriqué à partir d'autres articles et/ou matières premières.

Autres concepts

Classe-association : *association porteuse d'attributs et/ou d'opérations, représentée comme une classe anonyme associée à l'association.*

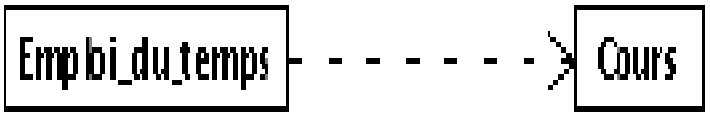


Association d'arité n : représentée par un losange avec n 'pattes' auquel peut être associé une classe porteuse d'attributs et/ou d'opérations.

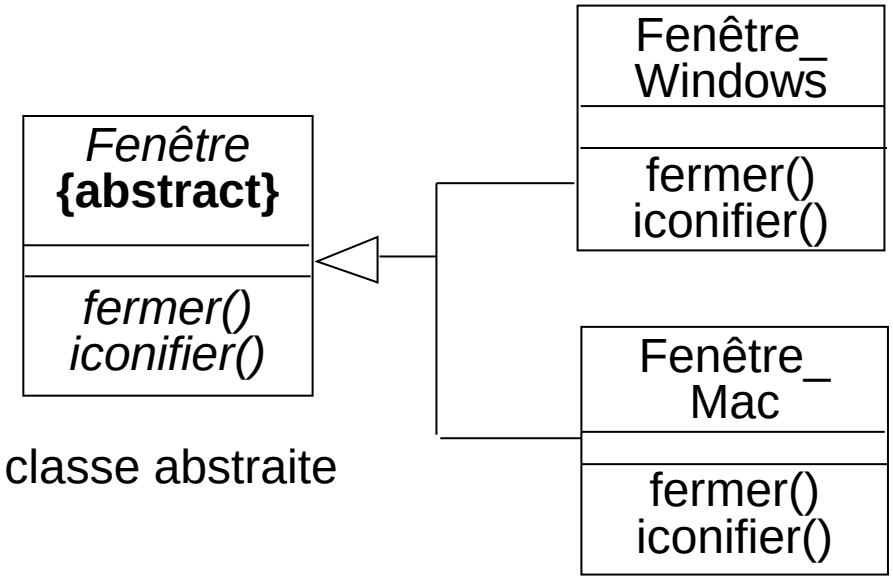


Dépendance

- Une dépendance est une relation unidirectionnelle exprimant une dépendance sémantique entre les éléments du modèle.
- Elle est représentée par un trait discontinu orienté .
- Elle indique que la modification de la cible implique une modification de la source.
- La dépendance est souvent stéréotypée pour mieux expliciter le lien sémantique entre les éléments du modèle .



Classe abstraite : non instanciable directement ; décrit des mécanismes généraux et laisse non décrit certains aspects (méthodes abstraites) ; spécialisée par des classes concrètes (instanciables) qui précisent les méthodes abstraites.



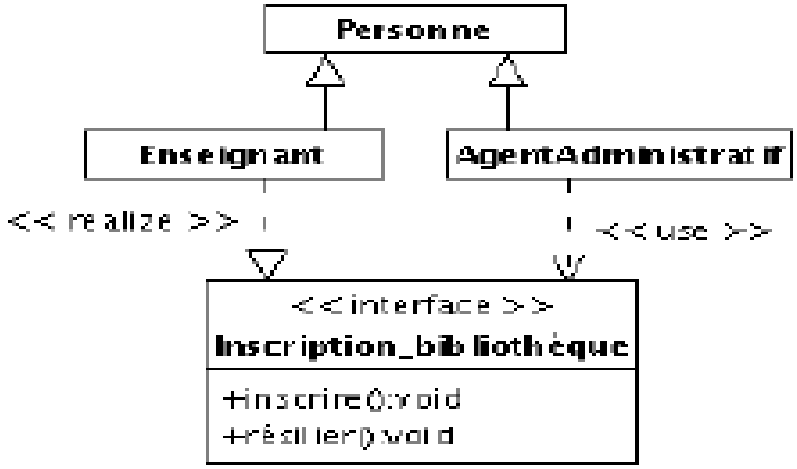
classe abstraite

Interface : classe ne contenant que des opérations abstraites ; une interface précise les fonctionnalités (le «service») que les classe qui les implantent doivent fournir.

❑Une interface est représentée comme une classe excepté l'absence du mot-clef abstract (car l'interface et toutes ses méthodes sont, par définition, abstraites) et l'ajout du stéréotype « interface » .

❑ Une interface doit être réalisée par au moins une classe. Graphiquement, cela est représenté par un trait discontinu terminé par une flèche triangulaire et le stéréotype « realize ».

❑ Une classe (classe cliente de l'interface) peut dépendre d'une interface (interface requise). On représente cela par une relation de dépendance et le stéréotype « use ».



Exemple de diagramme mettant en œuvre une interface

Élaboration d'un diagramme de classes

- **Trouver les classes du domaine étudié.** Cette étape empirique se fait généralement en collaboration avec un expert du domaine. Les classes correspondent généralement à des concepts ou des substantifs du domaine.

- **Trouver les associations entre classes.** Les associations correspondent souvent à des verbes, ou des constructions verbales, mettant en relation plusieurs classes, comme « est composé de », « pilote », « travaille pour ». Attention, méfiez vous de certains attributs qui sont en réalité des relations entre classes.

-**Trouver les attributs des classes.** Les attributs correspondent souvent à des substantifs, ou des groupes nominaux, tels que « la masse d'une voiture » ou « le montant d'une transaction ». Les adjectifs et les valeurs correspondent souvent à des valeurs d'attributs. Vous pouvez ajouter des attributs à toutes les étapes du cycle de vie d'un projet (implémentation comprise).

- **Organiser et simplifier le modèle en éliminant les classes redondantes et en utilisant l'héritage.**

- **Vérifier les chemins d'accès aux classes.**

- **Itérer et raffiner le modèle.** Un modèle est rarement correct dès sa première construction. La modélisation objet est un processus non pas linéaire mais itératif.

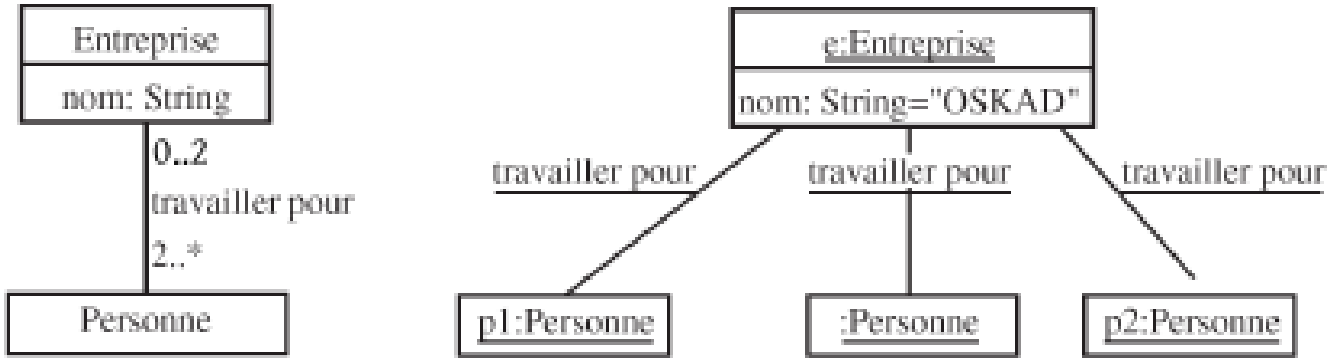
IV.2 Diagramme d'objets (object diagram)

❑ Un diagramme d'objets représente des objets (i.e. instances de classes) et leurs liens (i.e. instances de relations) pour donner une vue de l'état du système à un instant donné.

❑ Un diagramme d'objets permet, selon les situations, d'illustrer le modèle de classes (en montrant un exemple qui explique le modèle):

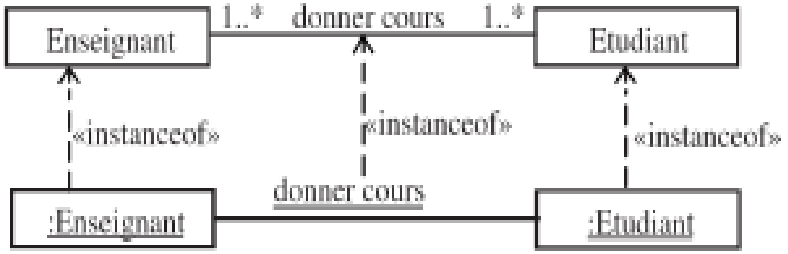
- de préciser certains aspects du système (en mettant en évidence des détails imperceptibles dans le diagramme de classes),
- d'exprimer une exception (en modélisant des cas particuliers, des connaissances non généralisables . . .), ou de prendre une image (snapshot) d'un système à un moment donné.

❑ Le diagramme de classes modélise les règles et le diagramme d'objets modélise des faits.



Relation de dépendance d'instanciation

❑ La relation de dépendance d'instanciation (stéréotypée « instanceof ») décrit la relation entre un classeur et ses instances. Elle relie, en particulier, les liens aux associations et les objets aux classes.



Dépendance d'instanciation entre les classeurs et leurs instances.

V. Modèles comportementaux (dynamiques)

- Les diagrammes d'états et d'activités servent à préciser comment un système se comporte au cours du temps, autrement dit sa **dynamique**.

- Ils décrivent le comportement complet du système ou d'un composant du système (une classe) à la différence des diagrammes d'interactions ou de séquences qui décrivent des cas (scénarios).

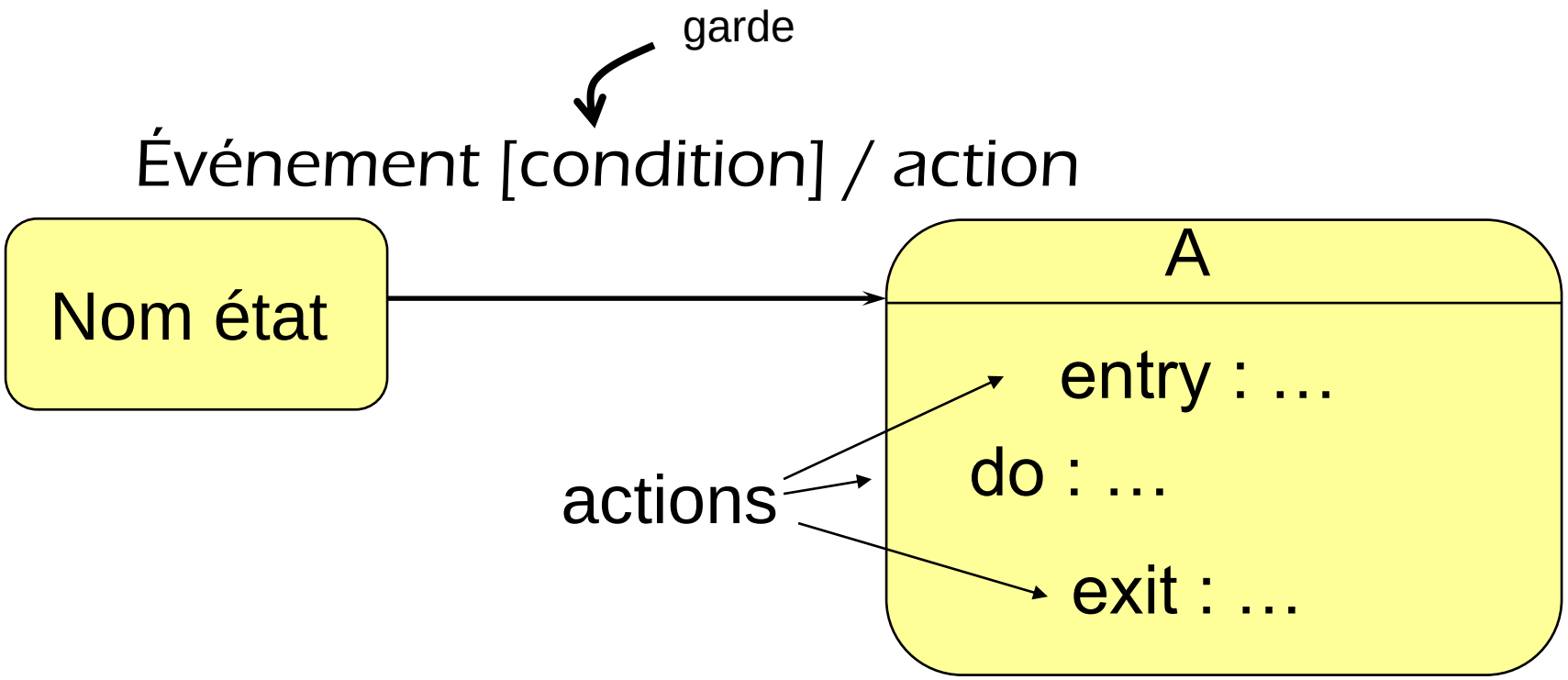
- Niveau **macroscopique** (cahier des charges) : comportement d'un système complexe.

- Niveau **microscopique** (conception détaillée) : comportement d'un objet (classe) complexe.

V.1 Diagramme d'Etat et Transition

- Les diagrammes d'états servent à décrire le cycle de vie des classes (objets) complexes.
- Ces diagrammes décrivent **tous les états possibles** de l'objet **et les transitions possibles** entre ces états.

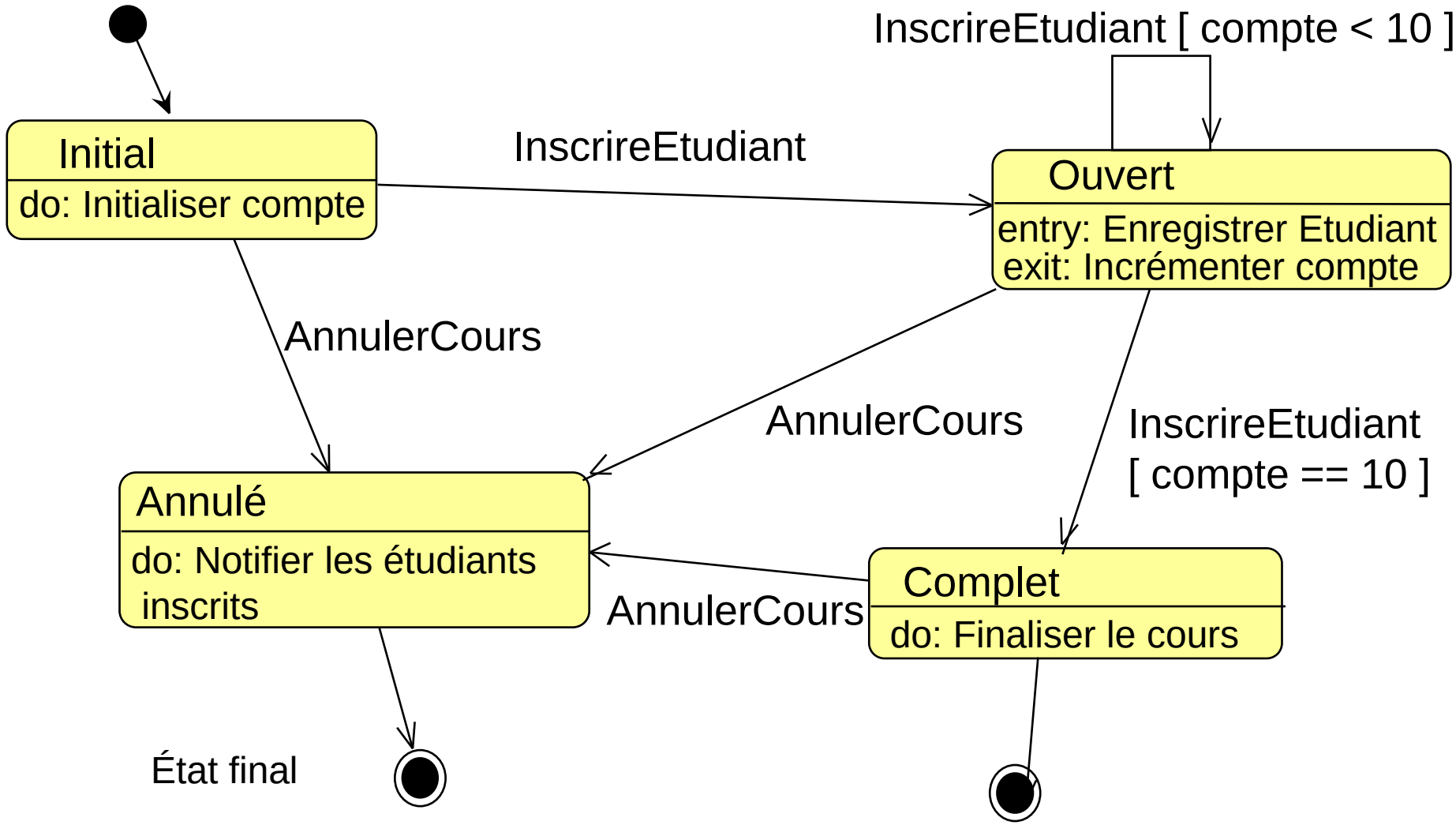
- On les appelle aussi « automates à états finis » ou « diagrammes de transitions ».
- En UML on peut faire figurer des **conditions** (gardes) sur les transitions et des **actions** sur les transitions et sur les états (au début de l'état, pendant l'état, à la fin de l'état).



État de départ Transition État d'arrivée

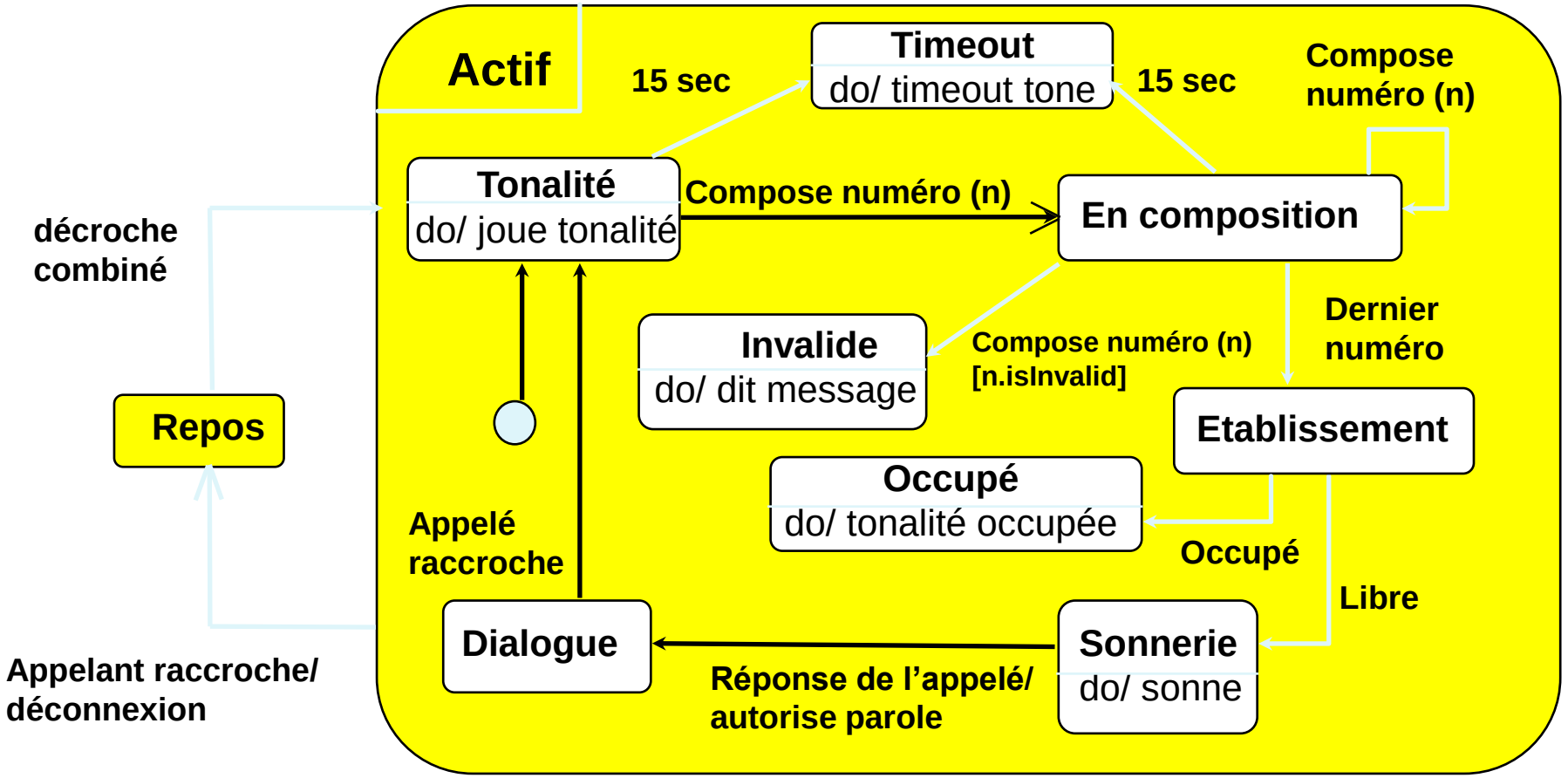
- État initial
- État final

Exemple 1 : États de la classe Cours



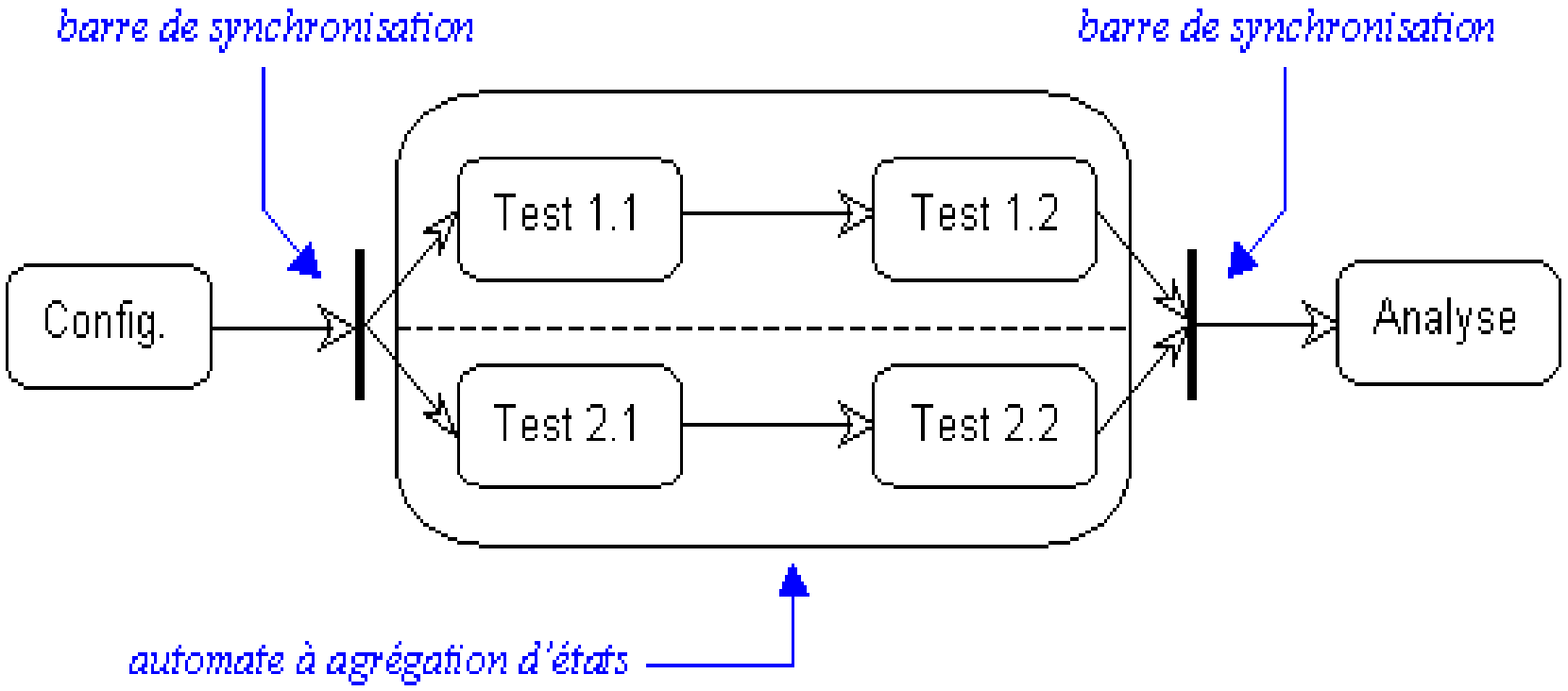
Les événements des transitions correspondent souvent dans ce cas à des appels de méthodes

Exemple 2 : états du système téléphone (diagramme avec décomposition d'état)

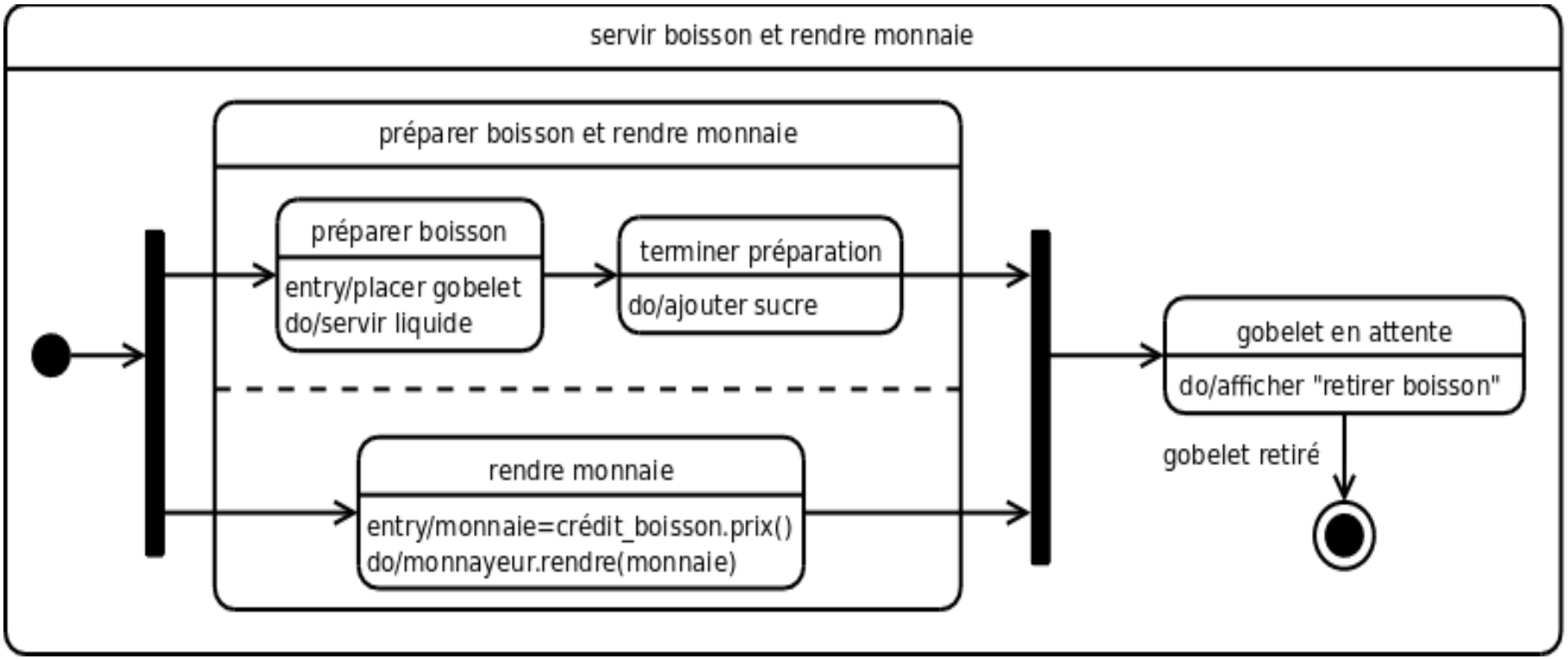


Les événements des transitions correspondent souvent dans ce cas à des actions sur le système

Exemple 3 : Diagramme avec activités parallèles



Exemple 4 : sous état parallèle d'un distributeur de boisson

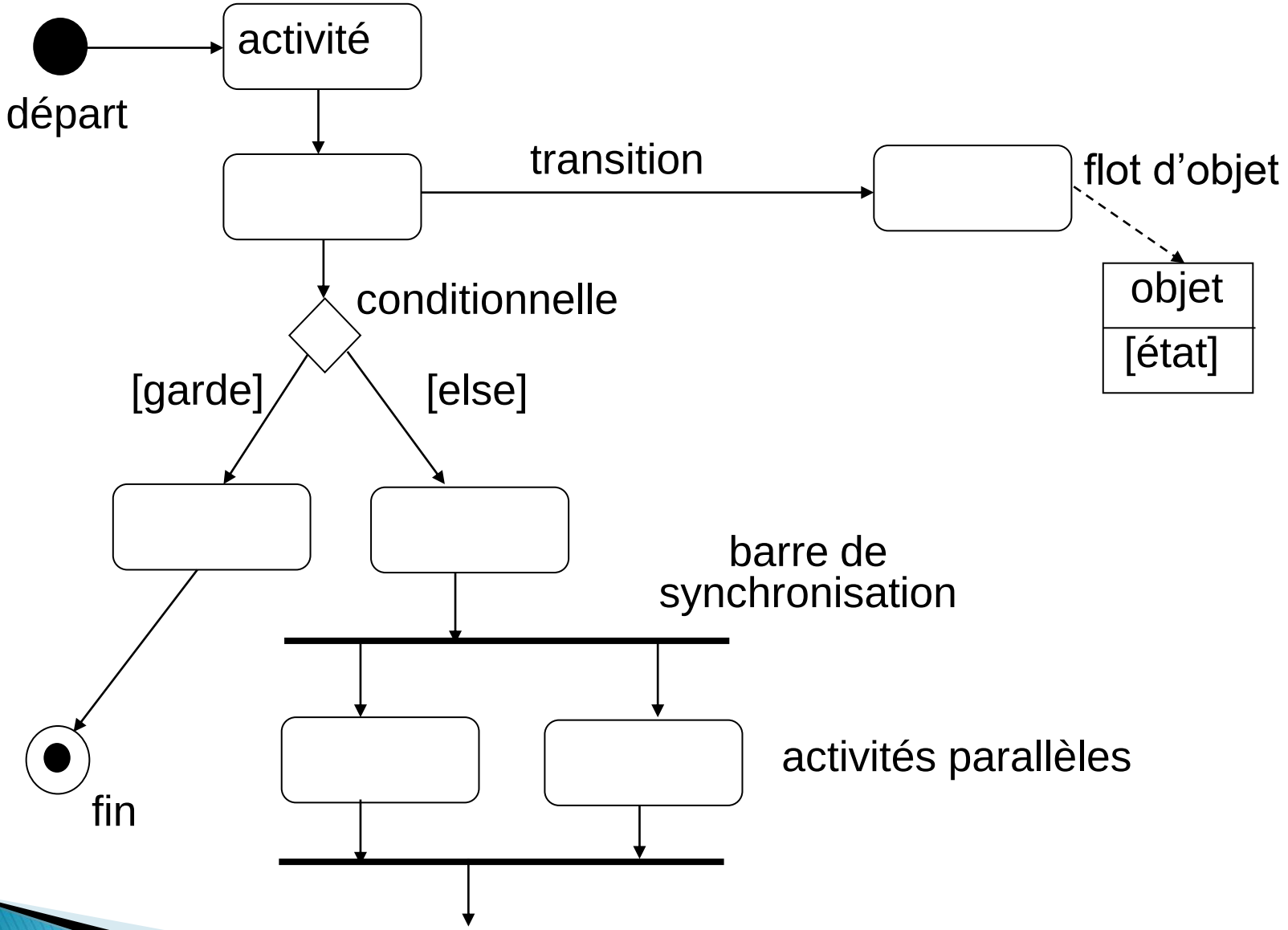


V.2 Les diagrammes d'activités

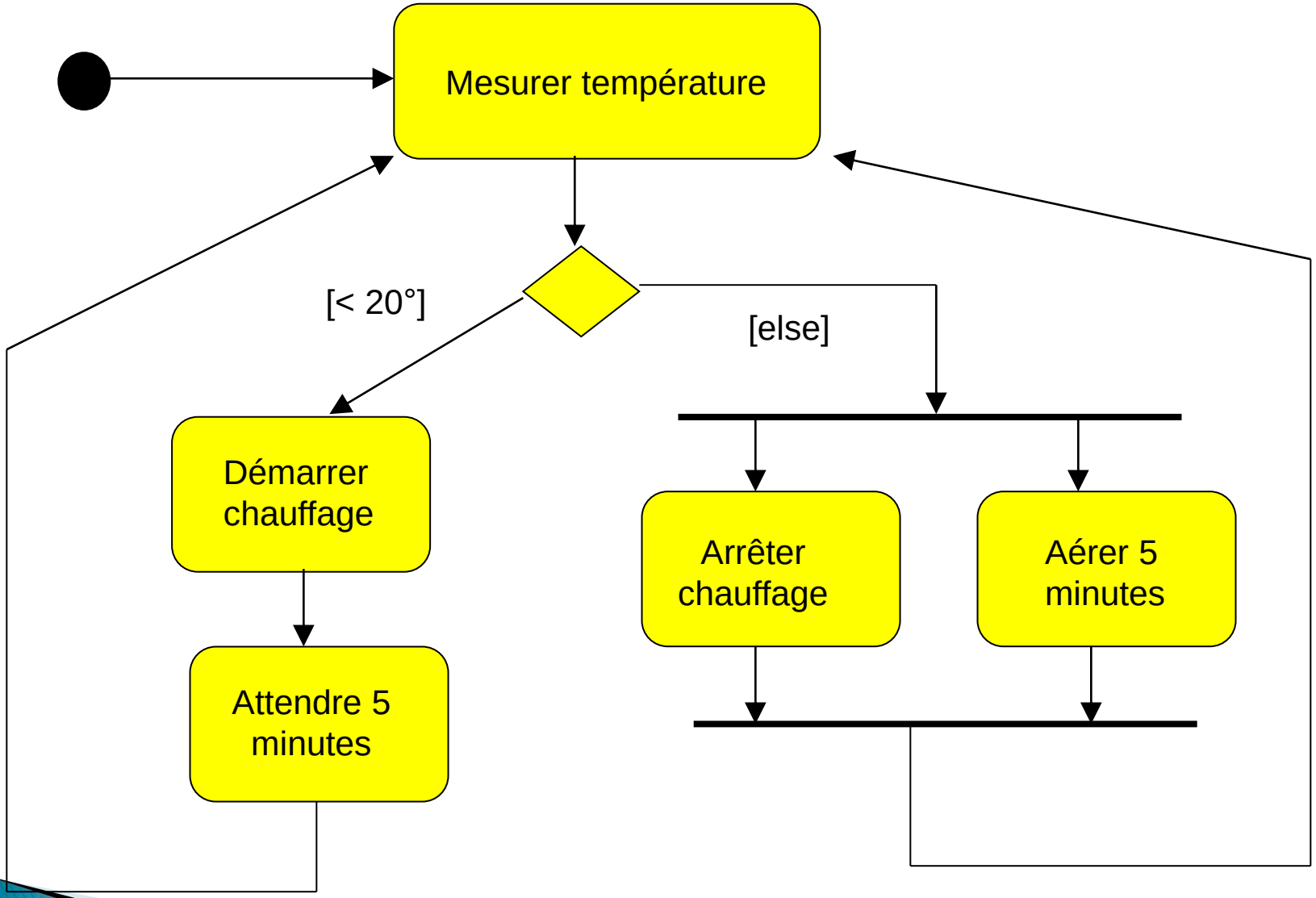
- Ils permettent de décrire le flot de contrôle entre activités : séquence, choix, itération, parallélisme.
- Il s'agit en gros d'organigrammes, incluant éventuellement du parallélisme.

A un niveau **macroscopique**, les diagrammes d'activité permettent de décrire des **enchaînements de fonctionnalités ou cas (« processus métiers »)**. Il complètent donc bien les cas d'utilisation au niveau de l'ingénierie des besoins.

A un niveau **microscopique**, les diagrammes d'activité permettent, par exemple, de décrire **l'algorithme d'une action complexe (méthode)**.

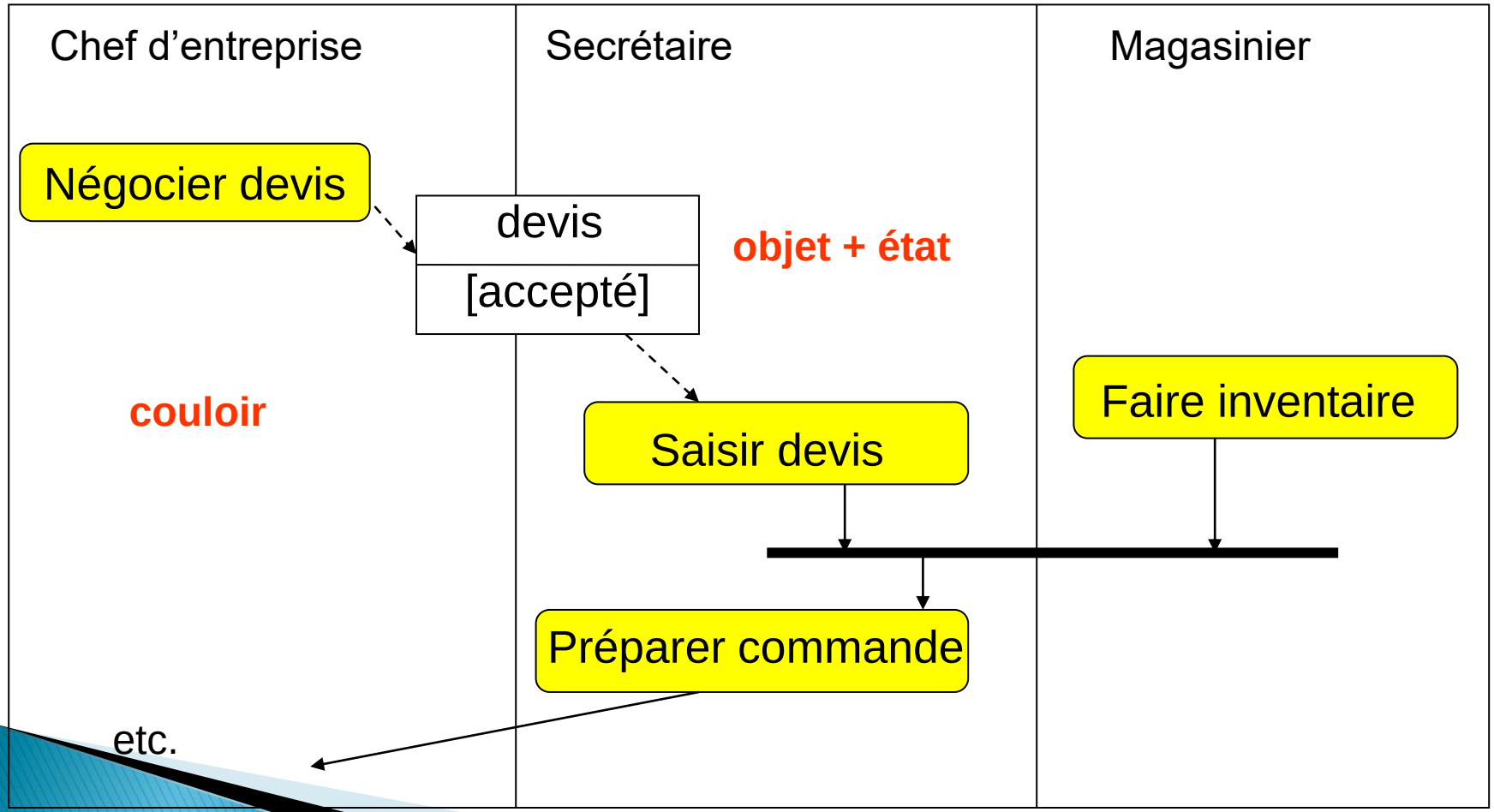

activité
départ
transition
flot d'objet
objet
[état]
conditionnelle
[garde]
[else]
barre de
synchronisation
fin
activités parallèles

Exemple 5: dynamique d'un système de chauffage

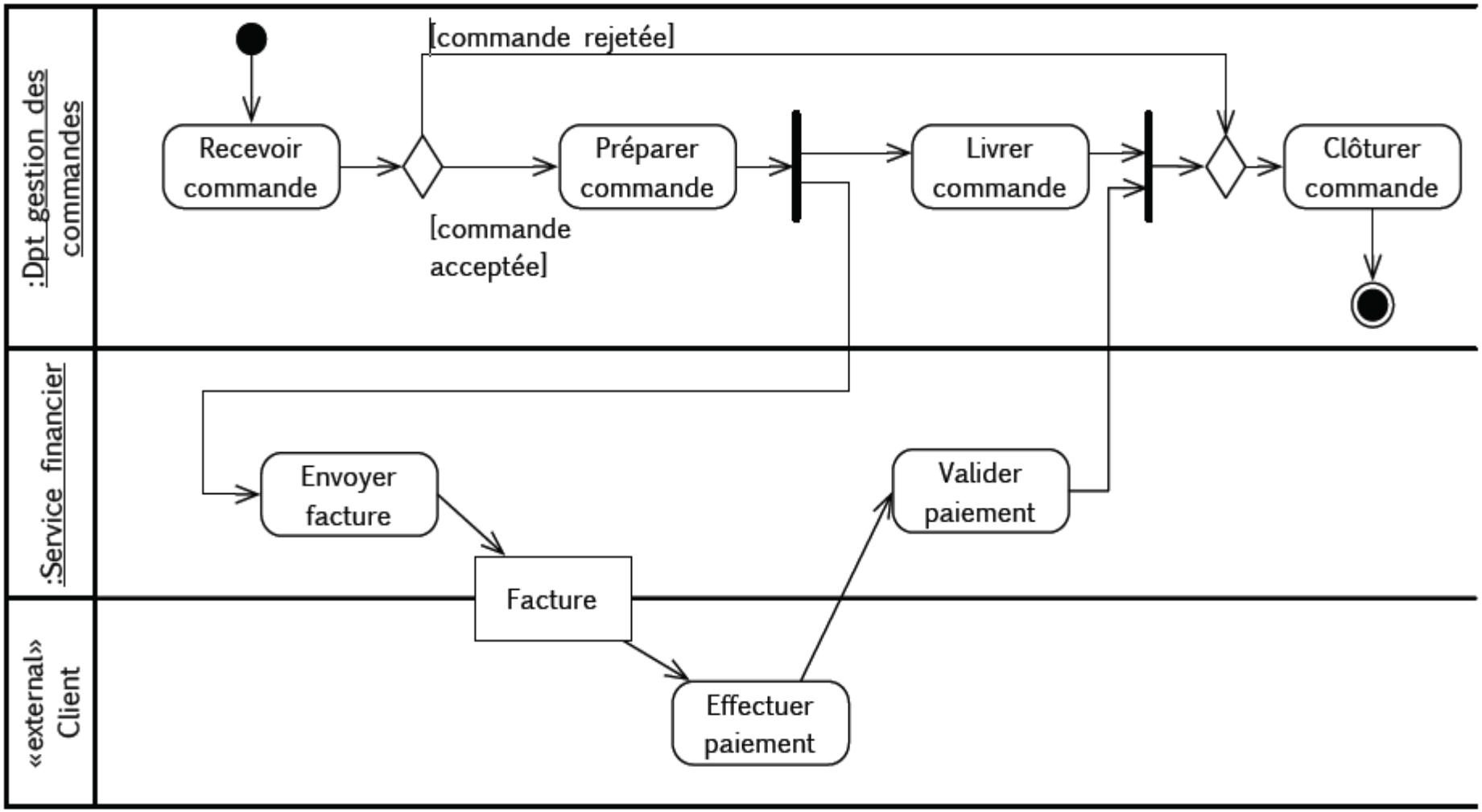


Extension aux processus métiers

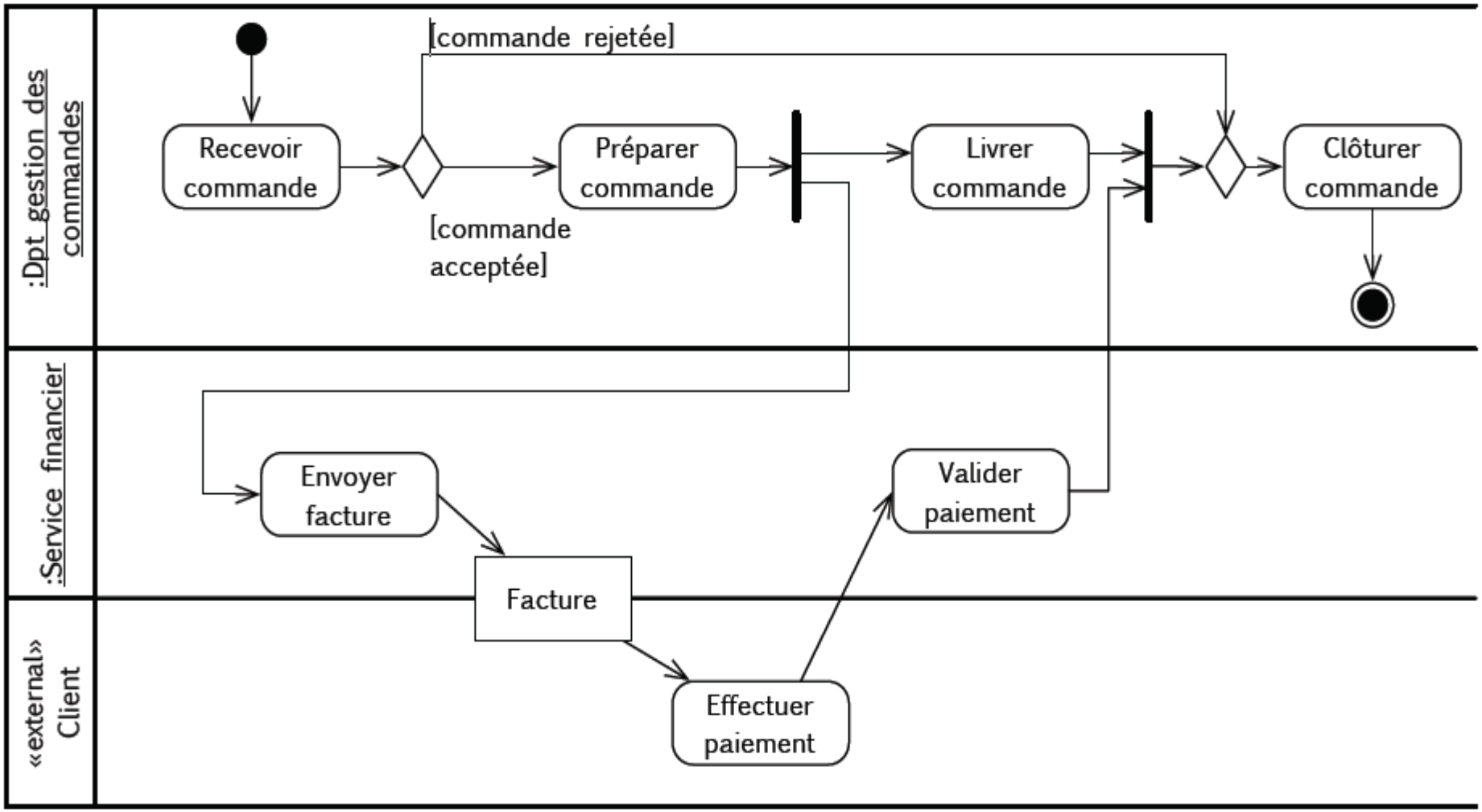
Des extensions aux diagrammes d'activité ont été proposées pour représenter aussi l'organisation (style MOT Merise, avec des « couloirs » ou « swimlane ») : extensions UML ->Business Process Modeling (BPM).



Exemple 6: processus de gestion des commandes



Exemple : processus de gestion des commandes



VI. Langage OCL

- Introduction
- Topologie des Contraintes
- Types et opérations utilisables dans les expressions OCL

VI.1. Introduction

1 Présentation du langage OCL

- Est un langage formel, basé sur la logique des prédicats du premier ordre, pour annoter les diagrammes UML en permettant notamment l'expression de contraintes.

- Est un langage de spécification formelle:
  - purement déclaratif (pas d'effets de bord)
  - fortement typé.

- Permet d'exprimer de l'information supplémentaire à propos de UML qui est exprimé suivant les différentes formes de contraintes:

  Contraintes structurelles : les attributs dans les classes, les différents types de relations entre classes (généralisation, association, agrégation, composition, dépendance), la cardinalité et la navigabilité des propriétés structurelles, etc. ;

  Contraintes de type : typage des propriétés, etc. ;

  Contraintes diverses : les contraintes de visibilité, les méthodes et classes abstraites (contrainte abstract), etc.

1 Présentation du langage OCL

- Plus avantageux par rapport aux langages formels traditionnels : qui sont peu utilisables par les utilisateurs et les concepteurs qui ne sont pas rompus à l'exercice des mathématiques
  - rester facile à écrire …
  - et facile à lire

- Dans le cadre de l'ingénierie des modèles, la précision du langage OCL est nécessaire pour pouvoir traduire automatiquement les contraintes OCL dans un langage de programmation afin de les vérifier pendant l'exécution d'un programme.

- Développé par IBM en 1995 : Fait partie d'UML depuis la version 1.1

2 Principe+

- **La notion de contrainte** Une contrainte est une expression à valeur booléenne que l'on peut attacher à n'importe quel élément UML. Elle indique en général une restriction ou donne des informations complémentaires sur un modèle.

- **Langage déclaratif** Les contraintes ne sont pas opérationnelles.

On ne peut pas invoquer de processus ni d'opérations autres que des requêtes.
On ne décrit pas le comportement à adopter si une contrainte n'est pas respectée.

- **Langage sans effet de bord** Les instances ne sont pas modifiées par les contraintes.

3 Utilisation

Les contraintes servent dans plusieurs situations :

– description d'invariants sur les classes et les types

– pré-conditions et post-conditions sur les opérations et méthodes

– contraintes sur la valeur retournée par une opération ou une méthode

– règles de dérivation des attributs

– description de cibles pour les messages et les actions

– expression des gardes (conditions dans les diagrammes dynamiques)

– invariants de type pour les stéréotypes

Les contraintes servent en particulier à décrire la sémantique d'UML ou de ses diverses extensions, en participant à la définition des profils.

VI.2. Topologie des Contraintes

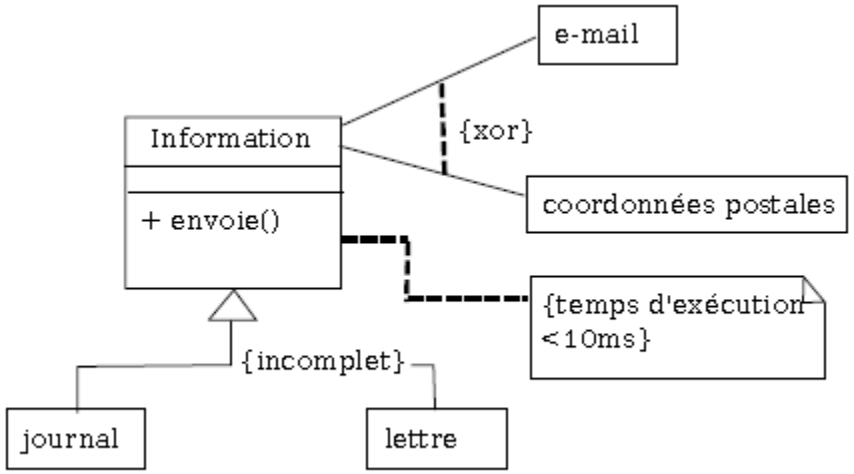
- Relation entre éléments de modélisation : propriété qui doit être vraie,

- Notation : - - - - - - - - - - {contrainte}

  3 types de contraintes :

  - ❑ Contraintes prédéfinie: disjoint, overlapping, xor, ...
  - ❑ Contraintes exprimées en langue naturelle (commentaires)
  - ❑ Contraintes exprimées avec OCL (Object Constraint Language)

  Stéréotypes : précondition, post-condition



VI.2. Topologie des Contraintes(1)

Illustration par l'exemple

- Prenons comme exemple une application bancaire à laquelle Il nous faut gérer
  - des comptes bancaires ;
  - des clients ;
  - et des banques.
- De plus, on aimerait intégrer les contraintes suivantes dans notre modèle :
  - ❑ un compte doit avoir un solde toujours positif ;
  - ❑ un client peut posséder plusieurs comptes ;
  - ❑ une personne peut être cliente de plusieurs banques ;
  - ❑ un client d'une banque possède au moins un compte dans cette banque ;
  - ❑ un compte appartient forcément à un client ;
  - ❑ une banque gère plusieurs comptes ;
  - ❑ une banque possède plusieurs clients.

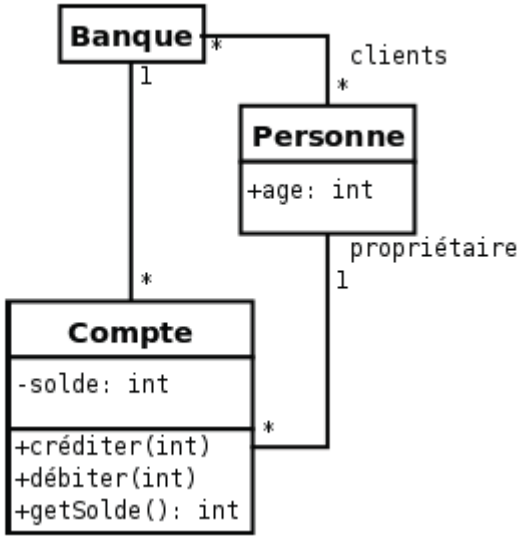
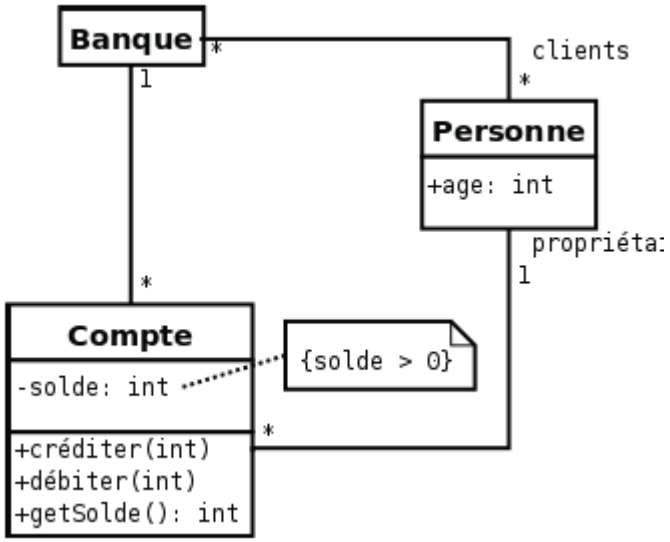


Diagramme de classes modélisant une banque, ses clients et leurs comptes.



Ajout d'une contrainte sur le diagramme de la figure ci-dessus

Contexte (context)

Une contrainte est toujours associée à un élément de modèle. C'est cet élément qui constitue le contexte de la contrainte. Il existe deux manières pour spécifier le contexte d'une contrainte OCL :

- En écrivant la contrainte entre accolades ({}) : dans une note (L'élément pointé par la note est alors le contexte de la contrainte ;
- En utilisant le mot-clef context dans un document accompagnant le diagramme .
- Le contexte peut être utilisé à l'intérieur d'une expression grâce au mot-clef self (self objet désigné par le contexte)

Syntaxe

context <élément>

<élément> peut être une classe, une opération, etc. Pour faire référence à un élément *op* (comme un opération) d'un classeur *C* (comme une classe), ou d'un paquetage... il faut utiliser les :: comme séparateur (comme *C*::*op*).

Exemple

Le contexte est la classe *Compte* :

context : Compte

Le contexte est l'opération *getSolde()* de la classe *Compte* :

context Compte :: getSolde()

Invariants (inv)

Un invariant exprime une contrainte prédicative sur un objet, ou un groupe d'objets, qui doit être respectée en permanence.

Syntaxe

inv : <expression_logique>

<expression_logique> est une expression logique qui doit toujours être vraie.

Exemple :

Le solde d'un compte doit toujours être positif.

Context: Compte

inv : solde > 0

Les femmes (au sens de l'association) des personnes doivent être des femmes (au sens du genre).

context Personne

inv : femme->forAll(genre=Genre::femme)

Préconditions et postconditions (pre, post)

- Une précondition (respectivement une postcondition) permet de spécifier une contrainte prédicative qui doit être vérifiée avant (respectivement après) l'appel d'une opération.
- Dans l'expression de la contrainte de la postcondition, deux éléments particuliers sont utilisables :
  - l'attribut result qui désigne la valeur retournée par l'opération ;
  - et <nom_attribut>@pre qui désigne la valeur de l'attribut <nom_attribut> avant l'appel de l'opération.

Syntaxe

• Précondition :

pre : <expression_logique>

• Postcondition :

post : <expression_logique>

<expression_logique> est une expression logique qui doit toujours être vraie.

Exemple

Concernant la méthode débiter de la classe Compte, la somme à débiter doit être positive pour que l'appel de l'opération soit valide et, après l'exécution de l'opération, l'attribut solde doit avoir pour valeur la différence de sa valeur avant l'appel et de la somme passée en paramètre.

context Compte::débiter(somme : Real)
pre : somme > 0
post : solde = solde@pre - somme

Le résultat de l'appel de l'opération getSolde doit être égal à l'attribut solde.

context Compte::getSolde() : Real
post : result = solde

Même si cela peut sembler être le cas dans ces exemples, nous n'avons pas décrit comment l'opération est réalisée, mais seulement les contraintes sur l'état avant et après son exécution.

Résultat d'une méthode (body)

Ce type de contrainte permet de définir directement le résultat d'une opération.

Syntaxe

body : <requête>

<requête> est une expression qui retourne un résultat dont le type doit être compatible avec le type du résultat de l'opération désignée par le contexte.

Exemple

le résultat de l'appel de l'opération getSolde doit être égal à l'attribut solde.

context Compte::getSolde() : Real
body : solde

Définition d'attributs et de méthodes (def et let...in)

Parfois, une sous-expression est utilisée plusieurs fois dans une expression.

let permet de déclarer et de définir la valeur (*i.e.* initialiser) d'un attribut qui pourra être utilisé dans l'expression qui suit le *in*.

def est un type de contrainte qui permet de déclarer et de définir la valeur d'attributs comme la séquence *let...in*.
def permet également de déclarer et de définir la valeur retournée par une opération interne à la contrainte.

Syntaxe de let...in

let <déclaration> = <requête> in <expression>
Un nouvel attribut déclaré dans <déclaration> aura la valeur retournée par l'expression <requête> dans toute l'expression <expression>.

Syntaxe de def

def : <déclaration> = <requête>

<déclaration> peut correspondre à la déclaration d'un attribut ou d'une méthode. <requête> est une expression qui retourne un résultat dont le type doit être compatible avec le type de l'attribut, ou de la méthode, déclaré dans <déclaration>. Dans le cas où il s'agit d'une méthode, <requête> peut utiliser les paramètres spécifiés dans la déclaration de la méthode.

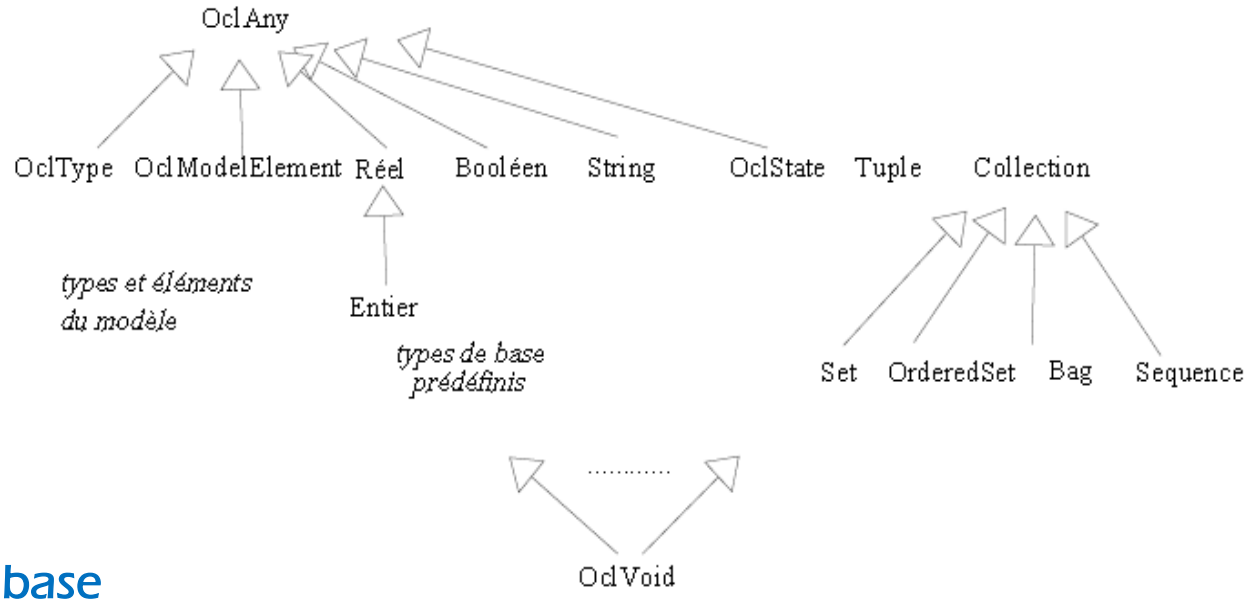
Exemple

Pour imposer qu'une personne majeure doit avoir de l'argent, on peut écrire indifféremment :

context Personne
inv : let argent=compte.solde->sum() **in** age>=18 implies argent>0
context Personne
def : argent : int = compte.solde->sum()
context Personne
inv : age>=18 implies argent>0

VI.2. Types et opérations utilisables dans les expressions OCL

La hiérarchie des types en OCL



Types de base

Prédéfinis

- Entier (Integer)
- Réel (Real)
- String
- Booléen (Boolean)

Spéciaux

- OclModelElement (énumération des éléments du modèle)
- OclType (énumération des types du modèle)
- OclAny (tout type autre que Tuple et Collection)
- OclState (pour les diagrammes d'états)
- OclVoid sous-type de tous les types

Entier

opérateurs = <> + - * / abs div mod max min < > <= >=
- est unaire ou binaire

Réel

opérateurs = <> + - * / abs oor round max min < > <= >=
- est unaire ou binaire

String

opérateurs = size() concat(String) toUpper() toLower()
substring(Entier, Entier)
Les chaînes de caractères constantes s'écrivent entre deux simples quotes : 'voici une chaîne'

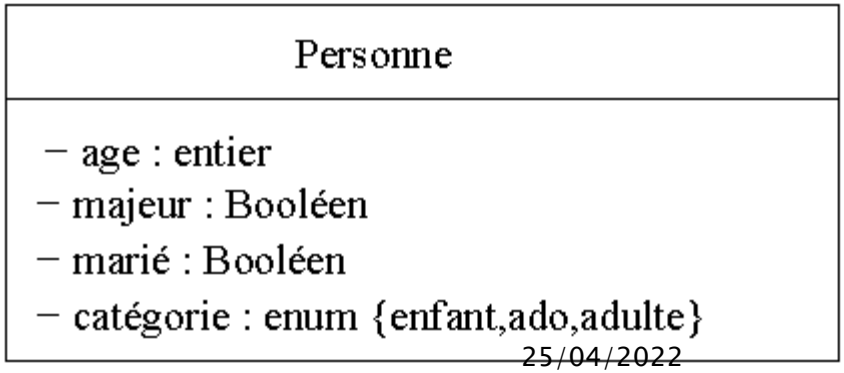
Booléen

opérateurs = or xor and not
b1 implies b2
if b then expression1 else expression2 endif

EXEMPLES

context Personne inv :
marié implies majeur

context Personne inv :
if age >=18 then majeur=vrai
else majeur=faux
endif



Précédence des opérateurs

. ->
not - unaire
* /
+ -
if then else
< > <= >=
<> =
and or xor
implies

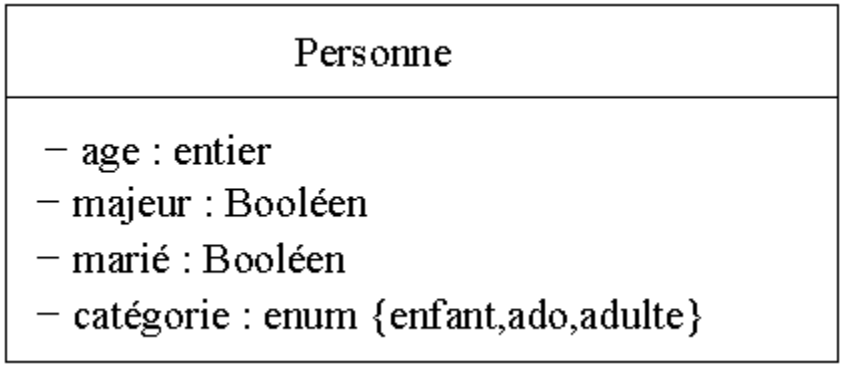
Type	Exemples de valeurs	Opérateurs
Boolean	*true* ; *false*	*and* ; *or* ; *xor* ; *not* ; *implies* ; *if-then-else-endif* ; ...
Integer	1 ; –5 ; 2 ; 34 ; 26524 ; ...	* ; + ; – ; / ; *abs*() ; ...
Real	1,5 ; 3,14 ; ...	* ; + ; – ; / ; *abs*() ; *floor*() ; ...
String	"To be or not to be ..."	*concat*() ; *size*() ; *substring*() ; ...

E1	*E2*	*P1 and P2*	*P1 or P2*	*P1 xor P2*	*P1 implies P2*
VRAI	VRAI	VRAI	VRAI	FAUX	VRAI
VRAI	FAUX	FAUX	VRAI	VRAI	FAUX
FAUX	VRAI	FAUX	VRAI	VRAI	VRAI
FAUX	FAUX	FAUX	FAUX	FAUX	VRAI

expression	*not expression*
VRAI	FAUX
FAUX	VRAI

Types énumérés

Leurs valeurs apparaissent précédées de #



context Personne inv :
if age <=12 then catégorie =#enfant
else if age <=18 then catégorie =#ado
else catégorie=#adulte
endif
endif

Types des modèles

- OCL est un langage typé dont les types sont organisés sous forme de hiérarchie. Cette hiérarchie détermine comment différents types peuvent être combinés.
- Par exemple, il est impossible de comparer un booléen (*Boolean*) avec un entier (*Integer*) ou une chaîne de caractères (*String*).
- Par contre, il est possible de comparer un entier (*Integer*) et un réel (*Real*), car le type entier est un *sous-type* du type réel dans la hiérarchie des types OCL.
- Bien entendu, la hiérarchie des types du modèle UML est donnée par la relation de généralisation entre les classeurs du modèle UML.

Les types des modèles utilisables en OCL sont les classificateurs (classeurs) , notamment les classes, les interfaces et les associations.

Types reliés par une relation de spécialisation

❑ Prise en compte des spécialisations entre classes du modèle avec opérations OCL dédiées à la gestion des types :

- ✓ oclIsKindOf(t) : vrai si l'objet est du type t ou un de ses sous-types (conversion ascendante ou descendante vers t)

  la conversion ascendante sert pour l'accès à une propriété redéfinie
  la conversion descendante sert pour l'accès à une nouvelle propriété

- ✓ oclIsTypeOf(t) : vrai si l'objet est du type t (vrai si t est supertype direct)
- ✓ oclAsType(type) : l'objet est caste en type t (vrai si t est supertype indirect)
- ✓oclInState (s : OclState) : Cette opération est utilisée dans un diagramme d'états-transitions. Elle est vraie si l'objet décrit par le diagramme d'états-transitions est dans l'état s passé en paramètre. Les valeurs possibles du paramètre s sont les noms des états du diagramme d'états-transitions. On peut faire référence à un état imbriqué en utilisant des «::» (par exemple, pour faire référence à un état B imbriqué dans un état A, on écrit : A::B).
- ✓oclIsNew () : doit être utilisée dans une postcondition. Elle est vraie quand l'objet au titre duquel elle est invoquée est créé pendant l'opération (i.e. l'objet n'existait pas au moment des préconditions).

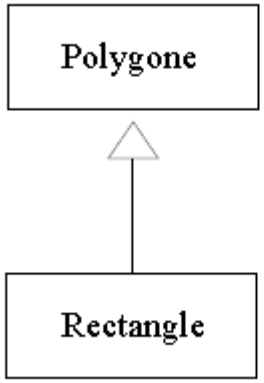
Exemple :

Une classeB hérite d'une classeA :

- dans le contexte de classeB, self.oclIsKindOf(ClasseB) et self.oclIsKindOf(ClasseA) sont vraies ;
- dans le contexte de classeA, self.oclIsKindOf(ClasseB) est fausse ;

Dans le contexte Sociéte, directeur.oclIsTypeOf(Personne) est vraie alors que self.oclIsTypeOf(Personne) est fausse.

Exemple



p : Polygone
r : Rectangle

Nous pourrions écrire par exemple les expressions
(r est supposé désigner une instance de Rectangle) :

- p = r
- p.oclAsType(Rectangle)(faux)
- r.oclIsTypeOf(Rectangle) (vrai)
- r.oclIsKindOf(Polygone) (vrai)

VI.3. Navigation dans les modèles

- Pour accéder aux caractéristiques (attributs, terminaisons d'associations, opérations) d'un objet, OCL utilise la notation pointée : *<objet>.<propriété>*.
- Cependant, de nombreuses expressions ne produisent pas comme résultat un objet, mais une collection. Le langage OCL propose plusieurs opérations de base sur les collections. Pour accéder ce type d'opération, il faut, utiliser non pas un point, mais une flèche : <collection> -> <opération>.

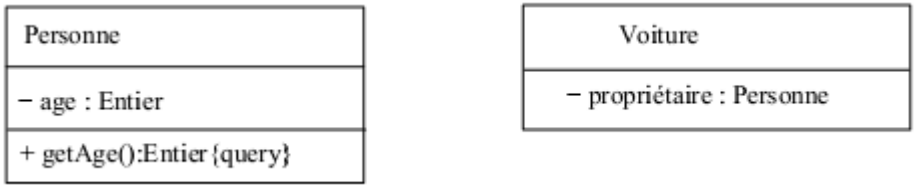
«::» permet de désigner un élément (comme une opération) dans un élément englobant (comme un classeur ou un paquetage) ;

«.» permet d'accéder à une caractéristique (attributs, terminaisons d'associations, opérations) d'un objet ;

«->» permet d'accéder à une caractéristique d'une collection.

Accès aux attributs

L'accès (navigation) vers un attribut s'effectue en mentionnant l'attribut, comme nous l'avons déjà vu, derrière l'opérateur d'accès noté '.'



Pour la classe Voiture de la figure ci-dessus, on peut ainsi écrire la contrainte ci-dessous.

context Voiture inv propriétaireMajeur :
self.propriétaire.age >= 18

Accès aux opérations query

Il se fait avec une notation identique (utilisation du '.').

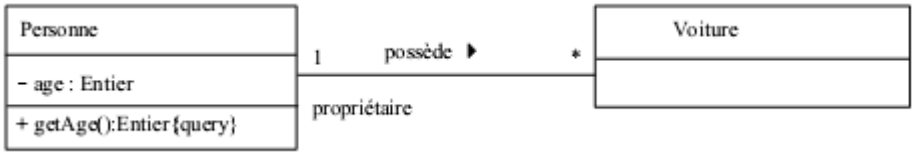
Pour la classe Voiture de la figure ci-dessus, on peut ainsi réécrire la contrainte propriétaireMajeur.

context Voiture inv :
self.propriétaire.getAge() >= 18

Accès aux extrémités d'associations

La navigation le long des liens se fait en utilisant :
– soit les noms de rôles
– soit les noms des classes extrémités en mettant leur première lettre en minuscule, à condition qu'il n'y ait pas ambiguïté

Pour la classe Voiture de la figure , on peut ainsi réécrire la contrainte propriétaireMajeur.



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

context Voiture inv :
self.personne.age >= 18

Opération sur les éléments d'une collection

La syntaxe d'une opération portant sur les éléments d'une collection est la suivante :

- Dans tous les cas, l'expression <expression> est évaluée pour chacun des éléments de la collection <collection>. L'expression <expression> porte sur les caractéristiques des éléments en les citant directement par leur nom. Le résultat dépend de l'opération <opération>.
- Parfois, dans l'expression <expression>, il est préférable de faire référence aux caractéristiques de l'élément courant en utilisant la notation pointée : <élément>.<propriété>. Pour cela, on doit utiliser la syntaxe suivante :

<collection> -> <opération>(<élément> | <expression>)

Exemples

Une société a au moins un employé :
context Société **inv :** self.employé->notEmpty()

Une société possède exactement un directeur :
context Société **inv :** self.directeur->size()=1

Le directeur est également un employé :
context Société **inv :** self.employé->includes(self.directeur)

Navigation vers les classes association

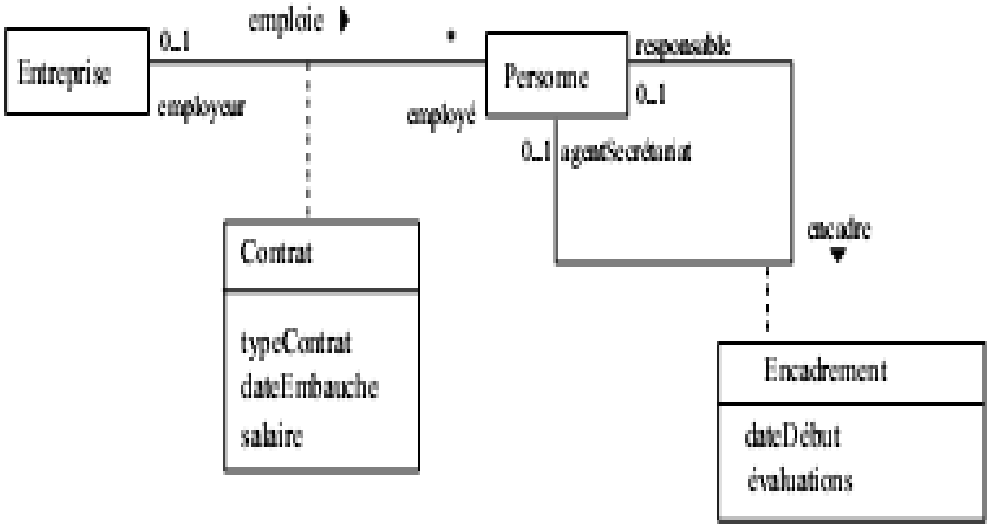
Pour naviguer vers une classe association, on utilise le nom de la classe (en mettant le premier caractère en minuscule). Par exemple dans le cas du diagramme de la figure ci-dessous

context p :Personne inv :
p.contrat.salaire >= 0

Ou encore, on précise le nom de rôle opposé (c'est même obligatoire pour une association réflexive) :

context p :Personne inv :
p.contrat[employeur].salaire >= 0

Pour naviguer depuis une classe association, on utilise les noms des rôles ou de classes. La navigation depuis une classe association vers un rôle ne peut donner qu'un objet. Dans l'exemple ci-dessous, c est un lien, qui associe par définition une seule entreprise à un seul employé.



context c :Contrat inv :
c.employé.age >= 16

VI.4 Eléments du langage

Désignation de la valeur antérieure

Dans une post-condition, il est parfois utile de pouvoir désigner la valeur d'une propriété avant l'exécution de l'opération. Ceci se fait en suffixant le nom de la propriété avec @pre.

Par exemple, pour décrire une nouvelle méthode de la classe Personne :

context Personne::feteAnniversaire()

pré : age < 140

post : age = age@pre + 1

Définition de variables et d'opérations

On peut définir des variables pour simplifier l'écriture de certaines expressions. On utilise pour cela la syntaxe :

let variable : type = expression1 in expression2

Par exemple, avec l'attribut dérivé impôt dans Personne, on pourrait écrire :

```
context Personne inv :
let montantImposable : Réel = contrat.salaire*0.8 in
if (montantImposable >= 100000)
then impot = montantImposable*45/100
else if (montantImposable >= 50000)
then impot = montantImposable*30/100
else impot = montantImposable*10/100
endif
endif
```

OCL 2.0 QUICK REFERENCE

OCL Constructs

context
Specifies the context for OCL expressions.

```
context Account
```

inv
States a condition that must always be met by all instances of a context type.

```
context Account
  inv: balance >= 0
```

pre
States a condition that must be true at the moment when an operation starts its execution.

```
context Account::deposit(amt: Integer): void
  pre: amt > 0
```

post
States a condition that must be true at the moment when an operation ends its execution.

```
context Account::deposit(amt: Integer): void
  post: balance = balance@pre + amt
```

init
Specifies the initial value of an attribute or association role.

```
context Account::balance: Integer
  init: 0
```

derive
Specifies the value of a derived attribute or association role.

```
context Account::interest: Real
derive: balance * .03
```

body
Defines the result of a query operation.

```
context Account::getBalance(): Integer
  body: balance
```

def
Introduces a new attribute or query operation.

```
context Account
  def: getBalance(): Integer = balance
```

package
Specifies explicitly the package in which OCL expressions belong.

```
package BankSystem::Accounting
context Account
  inv: balance >= 0
endpackage
```

OCL Expressions

self
Denotes the contextual instance.

```
context Account
  inv: self.balance >= 0
```

result
In a postcondition, denotes the result of an operation.

```
context Account::getBalance(): Integer
  post: result = balance
```

@pre
In a postcondition, denotes the value of a property at the start of an operation.

```
context Account::deposit(amt: Integer): void
  post: balance = balance@pre + amt
```

Navigation
Navigation through attributes, association ends, association classes, and qualified associations.

```
context Account
   inv: self.balance >= 0 -- dot notation
   -- collection operator (->)
   inv: owners->size() > 0
   -- association class, TransInfo
   inv: transactions.TransInfo->forall(amount > 0)
   -- qualified association, owners
   inv: not owners['primary'].isOclUndefined()
```

if-then-else expression
Conditional expression with a condition and two expressions.

```
context Account::interestRate: Real
derive: if balance > 5000 then .03 else .02 endif
```

let-in expression
Expression with local variables.

```
context Account::canWithdraw(amt: Integer): boolean
  def: let newBalance: Integer = balance - amt
       in newBalance > minimumBalance
```

Messaging (^)
Indicates that communication has taken place.

```
context Account::deposit(s: Sequence(Integer)): void
  pre: s->forAll(amt: Integer | amt > 0)
  post: balance = balance@pre + s->sum()
  post: s->forAll(amt: Integer | self^deposit(amt))
```

OCL Standard Library

Basic Types

Type	Values	Operations
Boolean	false, true	or, and, xor, not, =, <>, implies
Integer	-10, 0, 10, ...	=, <>, <, >, <=, >=, +, -, *, /, mod(), div(), abs(), max(), min(), round(), floor()
Real	-1.5, 3.14, ...	
String	'Carmen'	=, <>, concat(), size(), toLower(), toUpper(), substring()

OclAny
Supertype of all UML and OCL types

Operation	Description
=	True if *self* and the argument are the same
<>	True if *self* and the argument are not the same
oclIsNew()	True if *self* was created during the operation
oclIsUndefined()	True if *self* is undefined
oclAsType(type)	*self* as of the given type, *type*
oclIsTypeOf(type)	True if *self* is an instance of the given type, *type*
oclIsKindOf(type)	True if *self* conforms to the given type, *type*
oclisInState(state)	True if *self* is in the given state, *state*
T::allInstance()	Set of all instances of the type *T*

OclVoid

Type with one single instance (*undefined*) that conforms to all others types

Operation	Description
oclIsUndefined()	Always true

OclMessage

Messages that can be sent to and received by objects

Operation	Description
hasReturned()	Is the operation (*self*) called and returned?
result()	Result of the operation (*self*) or undefined
isSignalSent()	Is *self* a sending of a UML signal?
isOperationCall()	Is *self* a UML operation call?

Tuple

A tuple consists of named parts each of which can have a distinct type.

```
-- Tuple(name: String, age: Integer)
Tuple {name: String = 'John, age: Integer = 20}
```

Collection Types

Four collection types (Set, OrderedSet, Bag, and Sequence) with Collection as the abstract supertype.

Collection constants

Set {1, 2, 3} -- Set(Integer)
OrderedSet {'apple, 'pear', 'orange'} -- OrderedSet(String)
Bag {1, 1, 2, 2} -- Bag(Integer)
Sequence {1..(4 + 6), 15} – Sequence(Integer)

Standard operations

Operation	Description
count(o)	Number of occurrences of *o* in the collection (*self*)
excludes(o)	Is *o* not an element of the collection?
excludesAll(c)	Are all the elements of *c* not present in the collection?
includes(o)	Is *o* an element of the collection?
includesAll(c)	Are all the elements of *c* contained in the collection?
isEmpty()	Does the collection contain no element?
notEmpty()	Does the collection contain one or more elements?
size()	Number of elements in the collection
sum()	Addition of all elements in the collection

Collection operations

Operation	Set	OrderedSet	Bag	Sequence
=	O	O	O	O
<>	O	O	O	O
-	O	O		
append(o)		O		O
asBag()	O	O	O	O
asOrderedSet()	O	O	O	O
asSequence()	O	O	O	O
asSet()	O	O	O	O
at(i)*		O		O
excluding(o)	O	O	O	O
first()		O		O
flatten()	O	O	O	O
including(o)	O	O	O	O
indexOf(o)		O		O
insertAt(i, o)		O		O
intersection(c)	O		O	
last()		O		O
prepend(o)		O		O
subOrderedSet(l, u)		O		
subsequence(l, u)				O
symmetricDifference(c)	O			
union(c)	O	O	O	O

*OCL uses 1-based index for ordered sets and sequences.

including(o): new collection as *self* but with *o* added
excluding(o): new collection as *self* but with *o* removed

Iteration operations

Operation	Description
any(expr)	Returns any element for which *expr* is true
collect(expr)	Returns a collection that results from evaluating *expr* for each element of *self*
collectNested(expr)	Returns a collection of collections that result from evaluating *expr* for each element of *self*
exists(expr)	Has at least one element for which *expr* is true?
forAll(expr)	Is *expr* true for all elements?
isUnique(expr)	Does *expr* has unique value for all elements?
iterate(x: S; y: T| expr)	Iterates over all elements
one(expr)	Has only one element for which *expr* is true?
reject(expr)	Returns a collection containing all elements for which *expr* is false
select(expr)	Returns a collection containing all elements for which *expr* is true
sortedBy(expr)	Returns a collection containing all elements ordered by *expr*

```
accounts->any(a: Account | a.balance > 1000)
accounts->collect(name) -- all the names
accounts->collectNested(owners)
accounts->exists(balance > 5000)
accounts->forAll(balance >= 0)
accounts->isUnique(name)
accounts->iterate(a: Account; sum: Integer = 0 | sum + a.balance)
accounts->one(name = "Carmen")
accounts->reject(balance > 1000)
accounts->select(balance <= 1000)
accounts->sortedBy(balance)
```