INTRODUCTION À UML 2.2

MODÉLISATION ORIENTÉE OBJET DE SYSTÈME LOGICIEL

Youssouf EL ALLIOUI

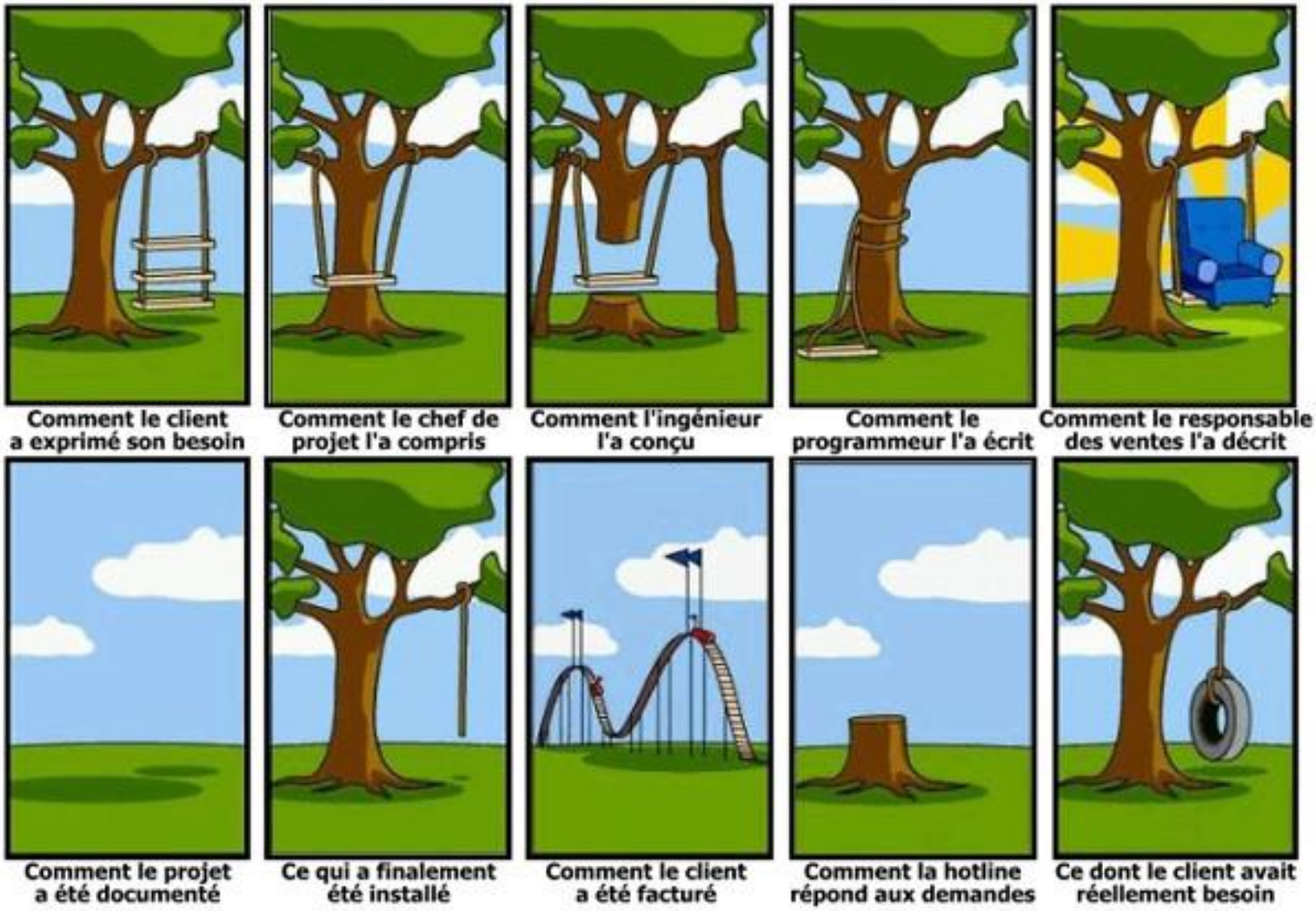
Plan du cours

جامعة الحسن الأول
UNIVERSITÉ HASSAN 1ER

Partie 1

Introduction à la Modélisation Orientée Objet

Problèmes de conception !



Qualité d'un logiciel (une solution informatique)

- **Adéquation entre :**
  - Le besoin effectif de l'utilisateur
  - Les fonctions offertes par le logiciel

- **Nécessité d'améliorer la communication :**
  - Moins de documents textuels et plus de modèles formels
  - A toutes les étapes du développement

Matériel el logiciel

- **Systèmes informatiques :**
  - **80 % de logiciel**
  - **20 % de matériel**
- **Depuis quelques années, la fabrication du matériel est assurée par quelques fabricants seulement**
  - **Le matériel est relativement fiable**
  - **Le marché est standardisé**

➔ **Les problèmes liés à l'informatique sont essentiellement des problèmes de logiciel**

La "crise du logiciel"

- **Etude sur 8.380 projets (Standish Group, 1995)**
  - **Succès : 16 %** des projets étaient conformes aux prévisions initiales
  - **Problématique : 53 %** (budget ou délais non respectés, défaut de fonctionnalités)
  - **Echec : 31 %** ont été abandonnés durant leur développement.

Le taux de succès décroît avec la taille des projets et la taille des entreprises

➔ **La naissance du Génie Logiciel (Software Engineering)**

- Cmment faire des logiciels de qualité ?
- Qu'attend-on d'un logiciel ?
- Quels sont les critères de qualité ?

Critères de qualité d'un logiciel

- **Utilité**
  - Adéquation entre le logiciel et les besoins des utilisateurs
- **Utilisabilité**
- **Fiabilité**
- **Interopérabilité**
  - Interaction avec des autres logiciels
- **Performance**
- **Portabilité**
- **Réutilisabilité**
- **Facilité de maintenance**
  - Un logiciel ne s'use pas pourtant, la maintenance absorbe une très grosse partie des efforts de développement

Poids de la maintenance !

	Répartition effort dév.	Origine des erreurs	Coût de la maintenance
Définition des besoins	6 %	56 %	82 %
Conception	5 %	27 %	13 %
Codage	7 %	7 %	1 %
Intégration tests	15 %	10 %	4 %
Maintenance	67 %		
	100 %	100 %	100 %

Réf. Zeltovitz, De Marco

Cycle de vie

➔ **La qualité du processus de fabrication est garante de la qualité du produit**

- **Pour obtenir un logiciel de qualité, il faut en maîtriser le processus d'élaboration**
  - **La vie d'un logiciel est composée de différentes étapes**
  - **La succession de ces étapes forme le cycle de vie du logiciel**
  - **Il faut contrôler la succession de ces différentes étapes**

Etapes du développement

- **Etude de faisabilité ou analyse de besoin**
- **Spécification**
  - **Déterminer les fonctionnalités du logiciel**
- **Conception**
  - **Déterminer la façon dont le logiciel fournit les différentes fonctionnalités recherchées**
- **Codage ou développement**
- **Tests**
  - **Essayer le logiciel sur des données d'exemple pour s'assurer qu'il fonctionne correctement**
- **Maintenance**

Modélisation

Modélisation

Modèle

- Représentations schématiques
- Formulation régoureuses
- …

Implantation

Qu'est ce qu'un modèle ?

- **Un modèle est une représentation abstraite de la réalité qui exclut certains détails du monde réel**
  - **Il permet de réduire la compléxité d'un phénomène en éliminant les détails qui n'influencent pas son comportement de manière significative**
  - **Il reflète ce que le concepteur croit important pour la compréhension et la prédiction du phénomène modélisé. Les limites du phénomène modélisé dépendent des objectifs du modèle.**

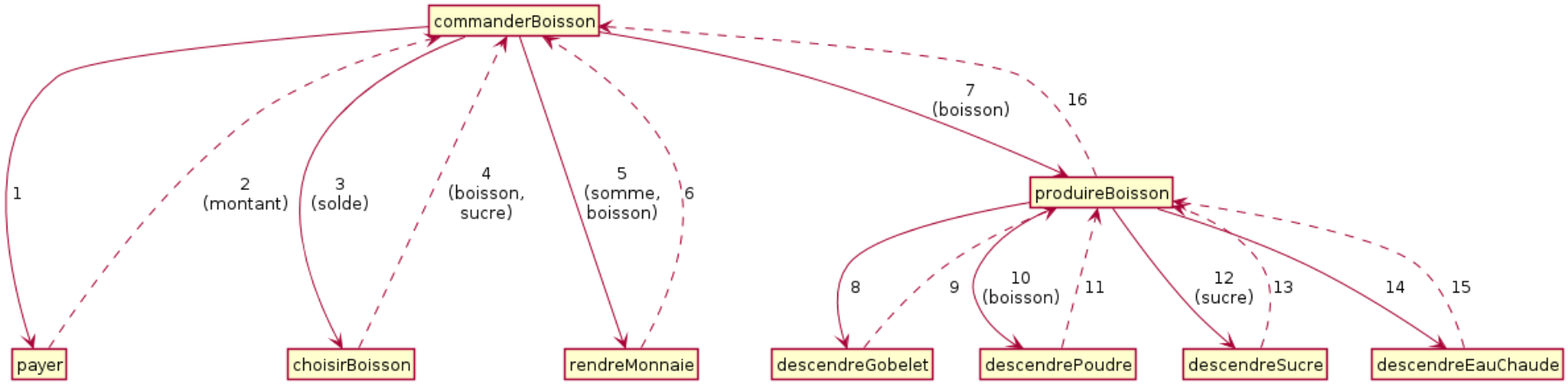
Langages de modélisation

- **Un langage de modélisation doit définir**
  - La sémantique des concepts
  - Une notation pour la représentation de concepts
  - Des régles de construction et d'utilisation des concepts
- **Des langages à différents niveaux de formalisation**
  - Langages formels (Z, V, VDM) : le plus souvent mathématiques, au grand pouvoir d'expression et permettant des preuves formelles sur les spécifications.
  - Langages semi-formels (Merise, UML, …) : le plus souvent graphiques, au pouvoir d'expression moindre mais plus facile d'emploi.
- **L'industrie du logiciel dispose de nombreux langages de modélisation :**
  - Adapter aux systèmes procéduraux (Merise)
  - Adapter aux systèmes temps réel (ROOM, SADT, …)
  - Adapter aux systèmes à objet (OMT, Booch, UML, …)
- **Le rôle des outils (AGL) est primordial pour l'utilisabilité en pratique des langages de modélisation**

Modélisation par décomposition fonctionnelle

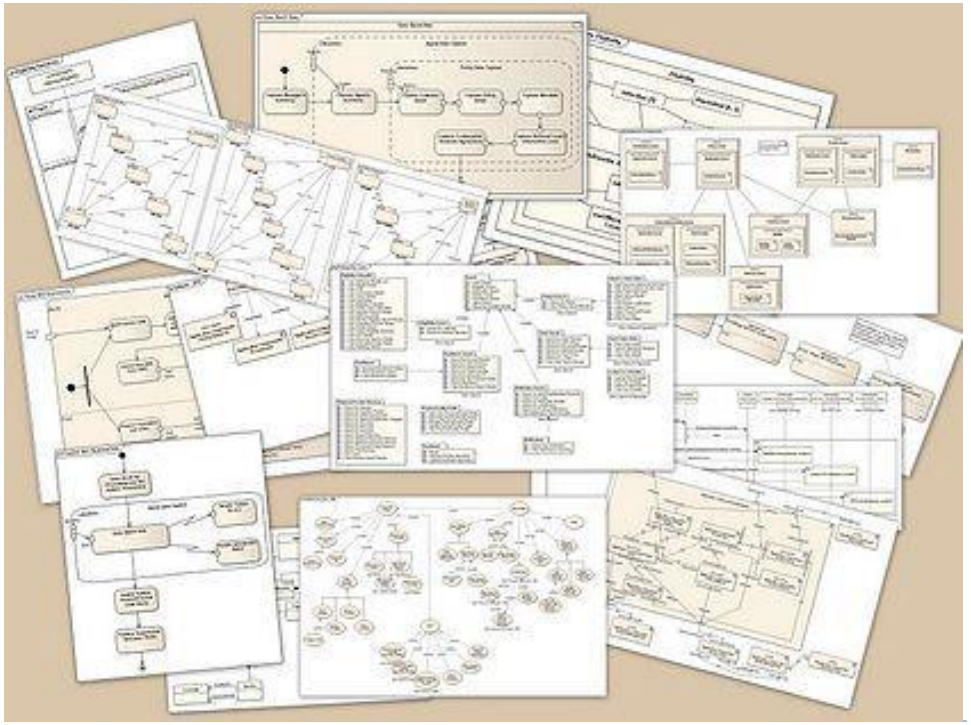
- **Approche descendante :**
  - Décomposer la fonction globale jusqu'à obtenir des fonctions simples à appréhender et donc à programmer.

➔ **C'est la fonction qui donne la forme du système**



UML - Unified Modeling Langage

- **Langage de modélisaton définissant un ensemble de diagrammes cohérents les uns avec les autres pour modéliser :**
  - **Définition des besoins**
  - **Conception**



UML - Unified Modeling Langage

- **Au milieu des années 90, les auteurs de Booch, OOSE, et OMT ont décidé de créer un langage de modélisation unifié avec pour objectifs :**
  - **Modéliser un système des concepts à l'exécutable, en utilisant les techniques orientées objet**
  - **Réduire la complexibilité de la modélisation**
  - **Utilisable par l'homme comme la machaine**
    - **Représentation graphiques mais disposant de qualités formelles suffisantes pour être traduites automatiquement en code source**
    - **Ces représentations ne disposent cependant pas de qualités formelles suffisantes pour justifier d'aussi bonnes propriétés mathématiques des langages de spécification formelle (Z,VDM, …)**
  - **Officiellement UML est né en 1994**

➔ **UML 2.0 date de 2005. Il s'agit d'une version majeure apportant des innovations radicales et étandant largement le champ d'application d'UML**

UML & L'OMG

- **OMG = Object Management Group (www.omg.org) :**
  - Fondé en 1989 pour standardiser et promouvoir l'objet
  - Version 1.0 d'UML (Unified Modeling Language) en janvier 1997
  - Version 2.5 en octobre 2012
- **Définition d'UML selon l'OMG :**
  - *"Langage visuel dédié à la spécification, la construction et la documentation des artéfacts d'un système logiciel"*
- **L'OMG définit le méta-modèle d'UML**
  - Syntaxe et interprétation en partie formalisées

➔ **Attention :**

UML est un langage... pas une méthode !

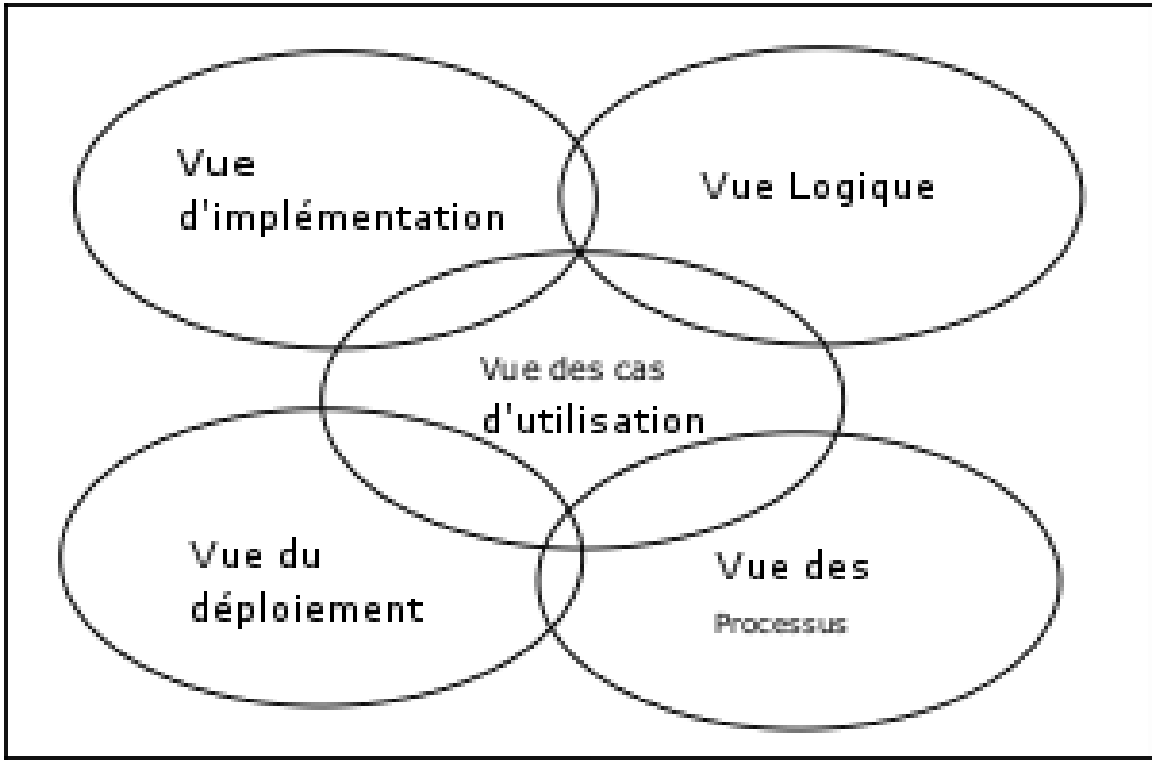
Différents modèles UML

- **UML peut être utilisé pour définir de nombreux modèles :**
  - **Modèles descriptifs vs prescriptifs**
    - **Descriptifs : Décrire l'existant (domaine, métier)**
    - **Prescriptifs : Décrire le futur système à réaliser**
  - **Modèles destinés à différents acteurs**
    - **Pour l'utilisateur : Décrire le quoi**
    - **Pour les concepteurs/développeurs : Décrire le comment**
  - **Modèles statiques vs dynamiques**
    - **Statiques : Décrire les aspects structurels**
    - **Dynamiques : Décrire comportements et interactions**
- **Les modèles sont décrits par des diagrammes (des graphes)**
  - **Chaque diagramme donne un point de vue différent sur le système**

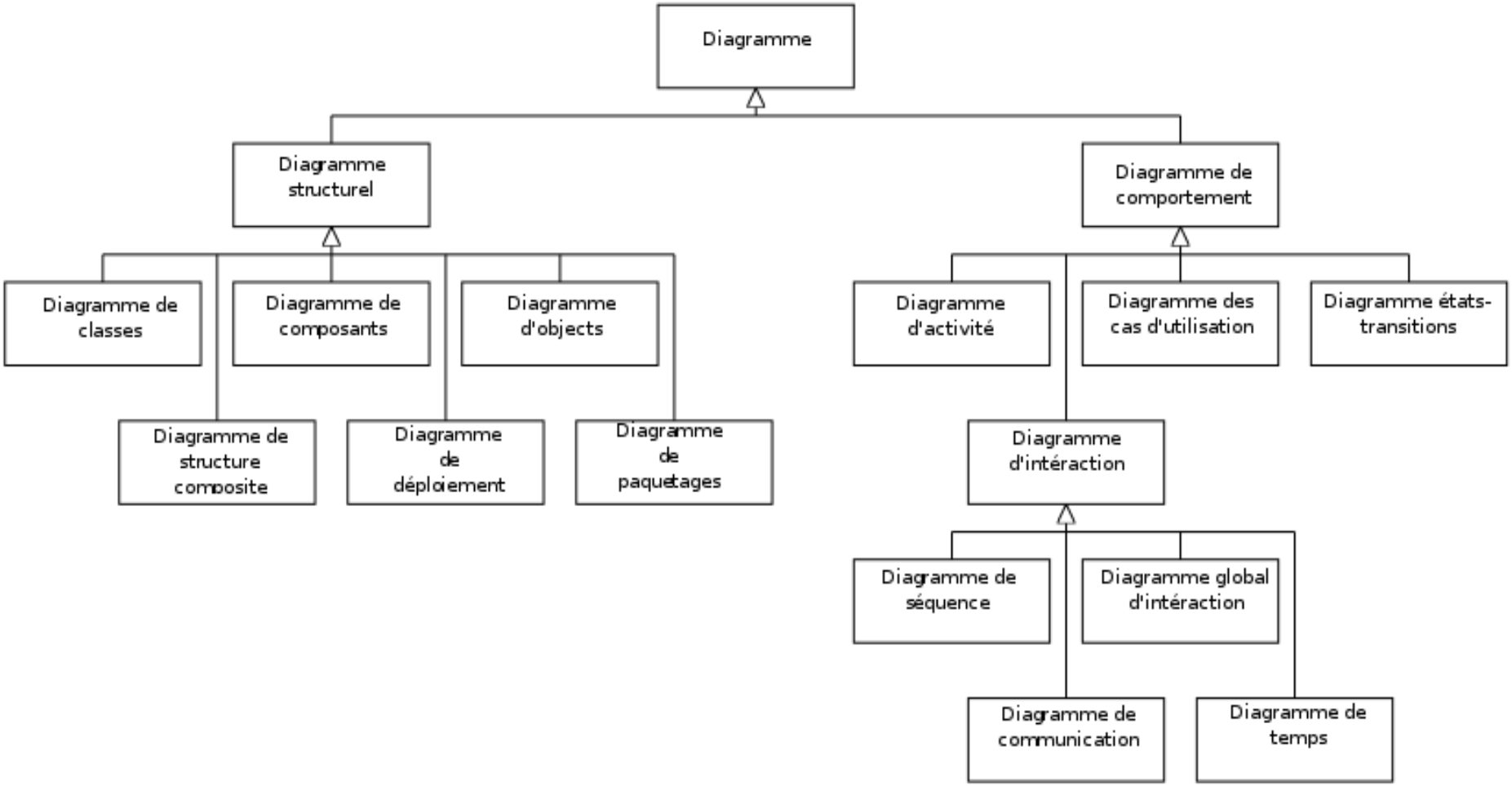
Les 4+1 vues d'un système

- **Vue des cas d'utilisation (use-case view) :**
  - description du modèle vu par les acteurs du système.
  - Elle correspond aux besoins attendus par chaque acteur (c'est le quoi et le qui).
- **Vue logique (logical view):**
  - définition du système vu de l'intérieur.
  - Elle explique comment peuvent être satisfaits les besoins des acteurs (c'est le comment).
- **Vue d'implémentation (implementation view) :**
  - cette vue définit les dépendances entre les modules.
- **Vue des processus (process view) :**
  - c'est la vue temporelle et technique, qui met en œuvre les notions de tâches concurrentes, stimuli, contrôle, synchronisation…
- **Vue de déploiement (deployment view) :**
  - cette vue décrit la position géographique et l'architecture physique de chaque élément du système (c'est le où).

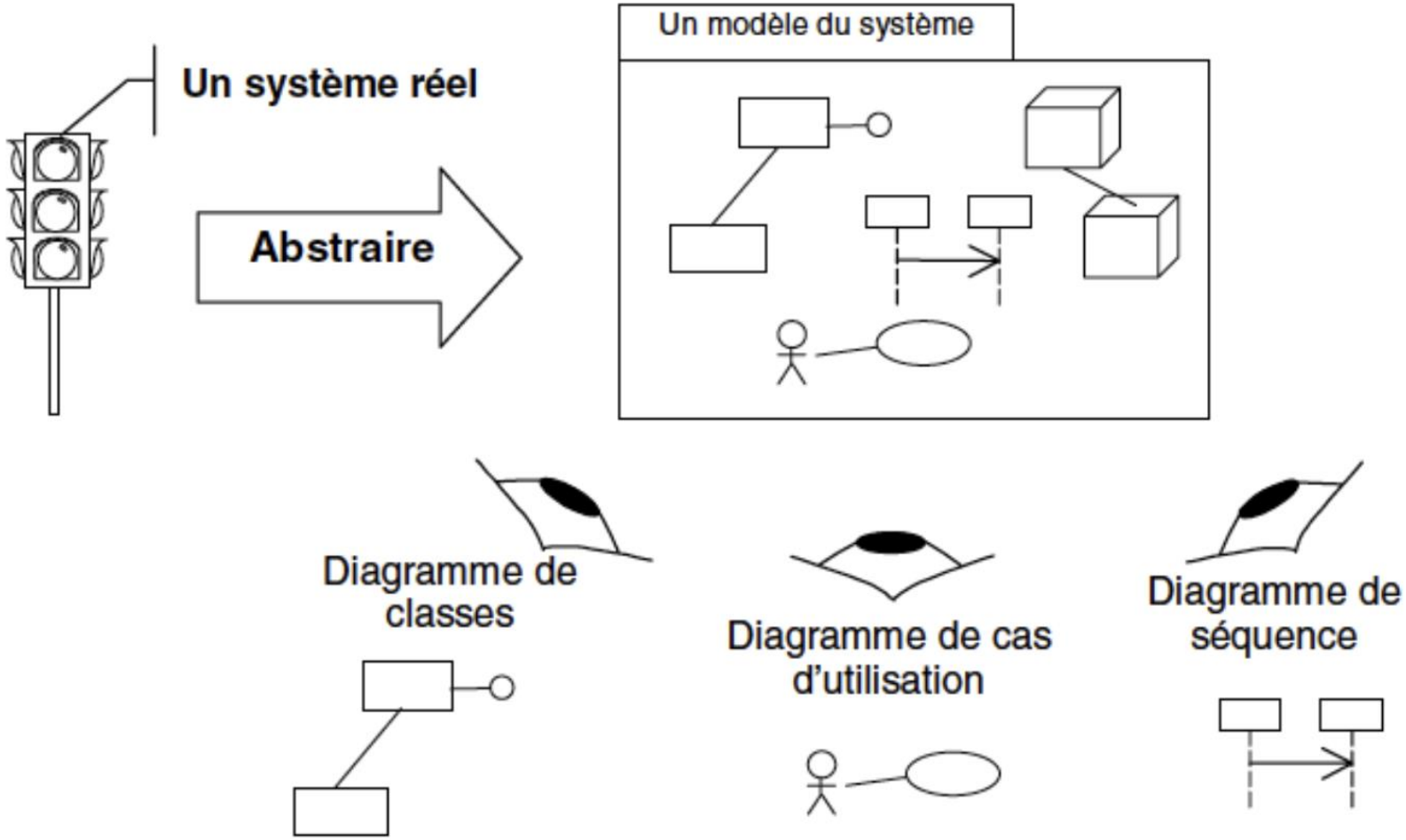
Les 4+1 vues d'un système



14 types de diagrammes d'UML 2.2



Diagrammes UML : Points de vue sur le système



[Image empruntée à Muller et Gaertner]

Notations communes à tous les diagrammes (1/2)

Stéréotypes et mots-clés

- **Définition d'une utilisation particulière d'éléments de modélisation**
- **Notation : «nomDuStéréotype» ou {nomDuMotClé} ou icone**
- **Nombreux stéréotypes et mots-clés prédéfinis : «interface», «invariant», «create», «actor», {abstract}, {bind}, {use}...**

Valeurs marquées

- Ajout d'une propriété à un élément de modélisation
- Notation : `{ nom1 = valeur1 , ..., nomn = valeurn }`
- Valeurs marquées prédéfinies (ex. : `derived : Bool`) ou personnalisées (ex. : `auteur : Chaîne`)

Commentaires

- Information en langue naturelle

Notations communes à tous les diagrammes (2/2)

Relations de dépendance

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

Contraintes

- **Relations entre éléments de modélisation**
  - Propriétés qui doivent être vérifiées
  - Attention : les contraintes n'ont pas d'effet de bord
- **Notation :** `o--------{ contrainte }`
- **Stéréotypes :** `«invariant», «précondition», «postcondition»`

Quiz

Plan

Partie 2

Modélisation Objet élémentaire avec UML

- Diagrammes de cas d'utilisation
- Diagrammes de classes
- Diagrammes d'objets
- Diagrammes de séquences
- Diagrammes d'activité

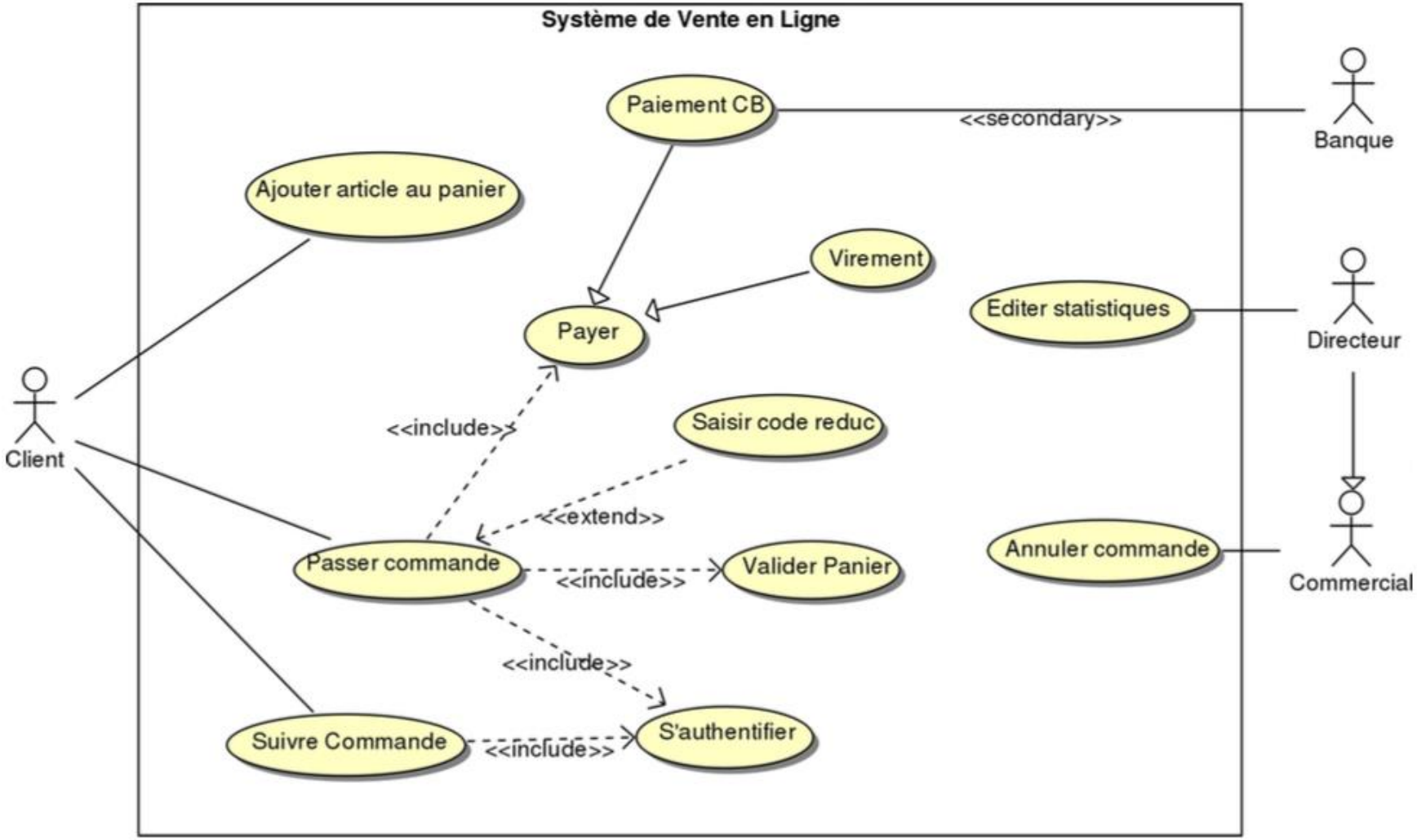
Modélisation des besoins

➔ **Avant de développer un système, il faut savoir précisément à QUOI il devera servir, càd à quels besoins il devera répondre**

- **Modéliser les besoins permet de :**
  - **Faire l'inventaire des fonctionnalités attendues**
  - **Organiser les besoins entre eux, de manière à faire apparaître des relations (réutilisations possibles, …)**

Avec UML, on modélise les besoins au moyen de diagrammes de cas d'utilisation

Exemple de diagramme de cas d'utilisation



Cas d'utilisation

- **Sémantique**
  - Un cas d'utilisation spécifie une fonction offerte par l'application à son environnement
  - Un cas d'utilisation est spécifié uniquement par un intitulé.
- **Graphique**
  - Un cas d'utilisation se représente par une ellipse contenant l'intitulé du cas d'utilisation

➔ **Un cas d'utilisation est l'expression d'un service réalisé de bout en bout, avec un déclenchement, un déroulement et une fin, pour l'acteur qui l'initie**

Acteurs

Sémantique

- Un acteur représente une entité extérieur au système, et qui interagit directement avec lui.
- Le concept d'acteur permet de classifier les entités externes au système.
- Un acteur est identifié par un nom

Graphique

- Un acteur se représente par un petit bonhomme et un nom (nom du rôle).

Système

Sémantique

- Un système représente une application dans le modèle UML.
- Il est identifié par un nom et regroupe un ensemble de cas d'utilisation qui correspondent aux fonctionnalités offertes par l'application à son environnement.
- L'environnement est spécifié sous forme d'acteurs liés aux cas d'utilisation.

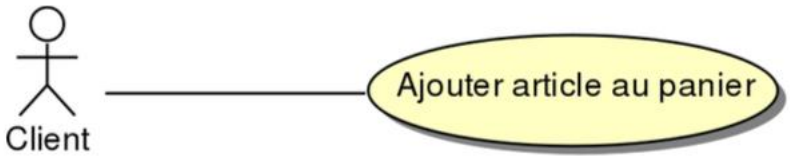
Graphique

- Un système se représente par un rectangle contenant le nom du système et les cas d'utilisation de l'application.
- Les acteurs, extérieurs au système, sont représentés et reliés aux cas d'utilisation qui les concernent.
- L'ensemble correspond à un diagramme de cas d'utilisation

Relations entre cas d'utilisation en acteurs

- **Sémantique :**
  - Les acteurs impliqués dans un cas d'utilisation lui sont liés par une **association**
  - Un acteur peut utiliser **plusieurs fois** le même cas d'utilisation

- **Graphique :**



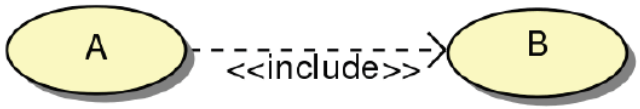
Relations entre cas d'utilisation - « include »

Sémantique

- Il est possible de spécifier qu'un cas d'utilisation inclut un autre cas d'utilisation
- le cas A inclut le cas B ➔ B est une partie obligatoire de A

Graphique

- La relation d'inclusion se représente à l'aide d'une flèche pointillée avec le stériotype « include »
- La flèche va du cas qui inclut vers le cas inclus



cas qui inclut

cas inclus

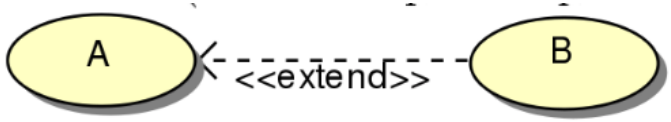
Relations entre cas d'utilisation - « extends »

Sémantique

- Il est possible de spécifier qu'un cas d'utilisation étend un autre cas d'utilisation
- Le cas B étend le cas A ➔ B est une partie optionelle de A

Graphique

- La relation d'extension se représente à l'aide d'une flèche pointillée avec le stériotype « extends »
- La flèche va du cas qui étend vers le cas étendu



le cas étendu cas qui étend

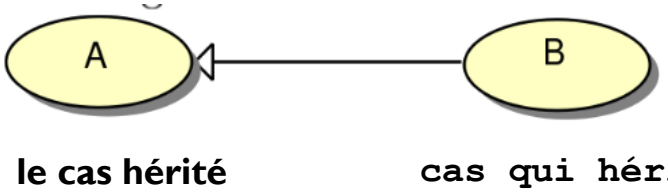
Relations entre cas d'utilisation - Généralisation

Sémantique

- Il est possible de spécifier une relation d'héritage entre cas d'utilisation
- Le cas A est une généralisation du cas B ➔ B est une sorte de A

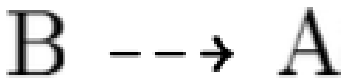
Graphique

- La relation d'héritage se représente par une flèche allant du cas d'utilisation qui hérite vers le cas d'utilisation hérité
- La flèche pointe vers l'élément général



Dépendances d'inclusion et d'extension

- Les inclusions et les extensions sont représentées par des dépendances
  - Lorsqu'un cas B inclut un cas A, B dépend de A
  - Lorsqu'un cas B étend un cas A, B dépend aussi de A
  - On note toujours la dépendance par une flèche pointillée qui se lit : B dépend de A



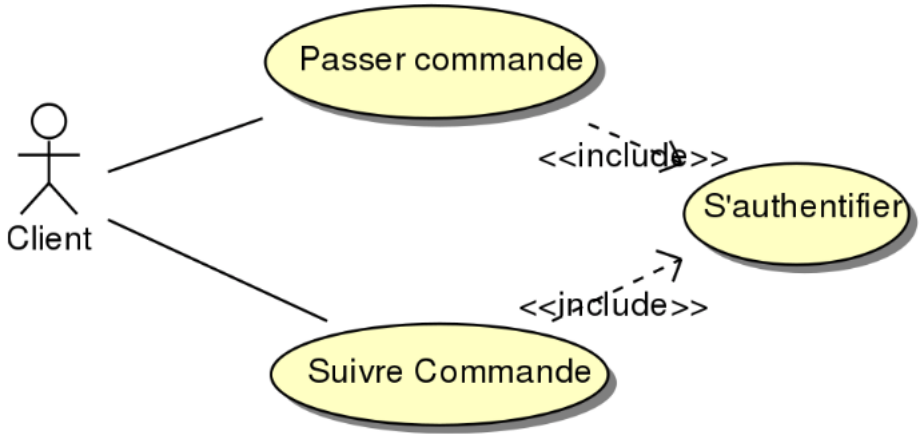
- Losqu'un élément A dépend d'un élément B, toute modification de B sera susceptible d'avoir un impact sur A.
- Les « include » et les « extends » sont des stéréotypes (entre guillements) des relations de dépendance.

Réutilisabilité avec les inclusions et les extensions

- Les relations entre cas permettent la réutilisabilité

Exemple

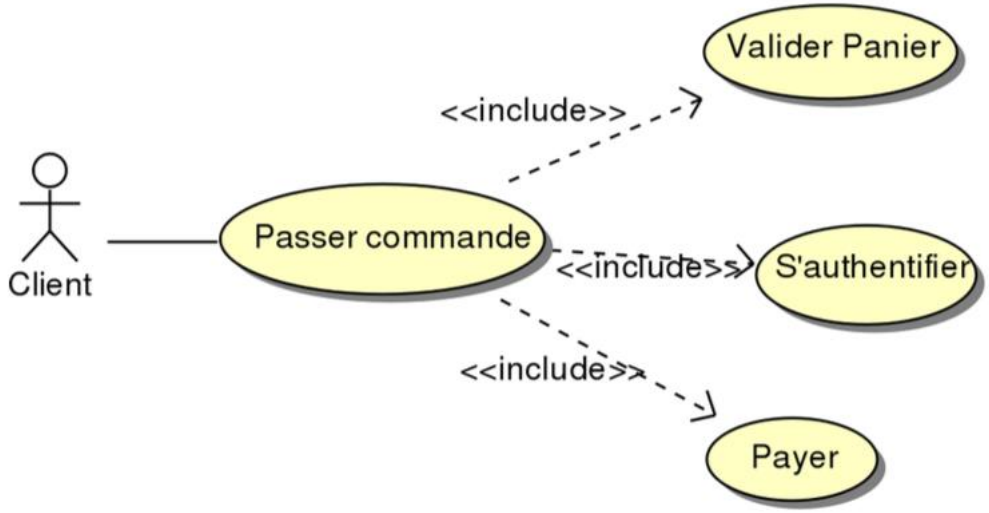
➔ Graçe au cas d'utilisation « s'authentifier », il sera inutile de développer plusieurs fois un module d'authentification.



Décomposition grâce aux inclusions et aux extensions

- Quand un cas est trop complexe (faisant intervenir un trop grand nombre d'actions), on peut procéder à sa décomposition en cas plus simples

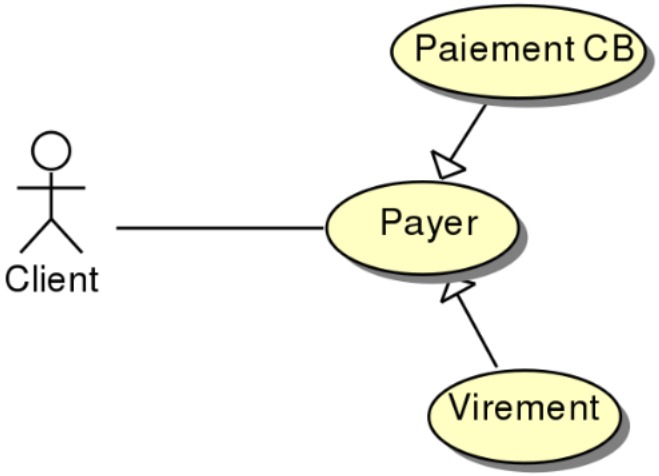
Exemple :



Généralisation

- **Un virement est un cas particulier de paiement**

  ➔ **Un virement est une sorte de paiement**

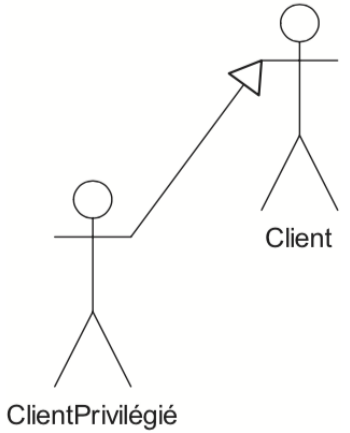


Cette relation de généralisation/spécialisation est présentée dans la plupart des diagrammes UML et se traduit par le concept d'héritage dans les langages orientés objet

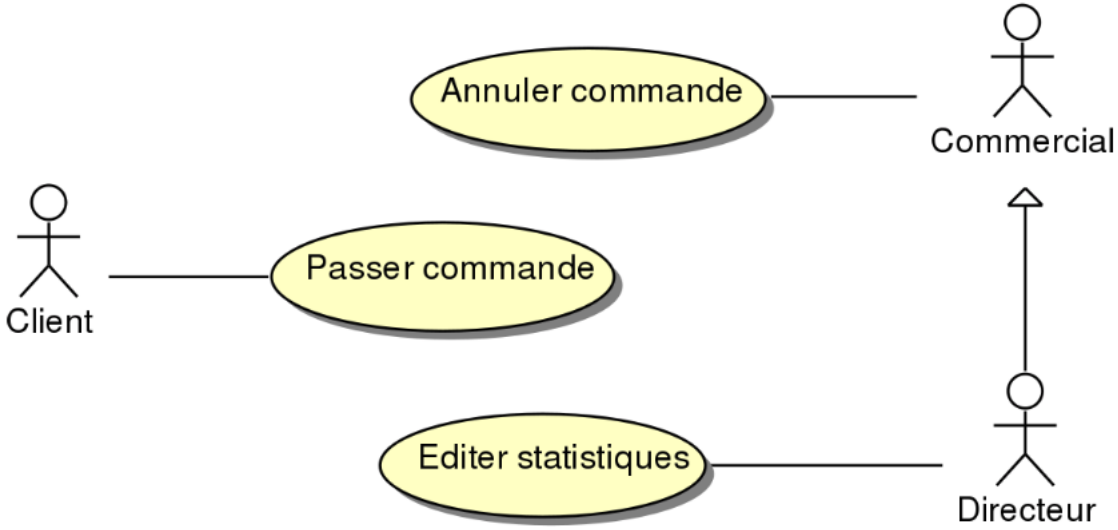
Relations entre acteurs

- **La généralisation :**

  ➔ **Une seule relation possible : la généralisation**



- **Exemple :**



Identification des acteurs

- Les principaux acteurs sont les utilisateurs du système

> **Attention :** Un acteur correspond à un rôle, pas à une personne physique

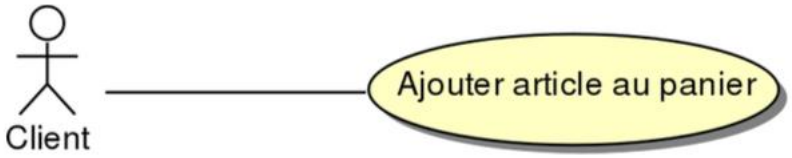
  - Une même personne physique peut être représentée par plusieurs acteurs si elle a plusieurs rôles
  - Si plusieurs personnes jouent le même rôle vis-à-vis du système, elles seront représentées par un seul acteur

- En plus des utilisateurs, les acteurs peuvent être :
  - Des périphériques manipulés par le système (imprimantes...) ;
  - Des logiciels déjà disponibles à intégrer dans le projet ;
  - Des systèmes informatiques externes au système mais qui interagissent avec lui, etc.

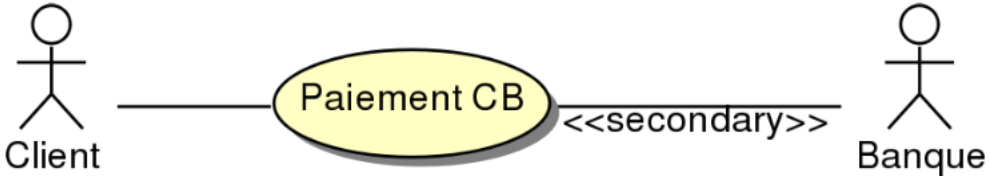
- Pour faciliter la recherche des acteurs, on se fonde sur les frontières du système.

Acteurs principaux et secondaires

- **L'acteur est dit principal pour un cas d'utilisation lorsque l'acteur est à l'initiative des échanges nécessaires pour réaliser le cas d'utilisation**



- **Les acteurs secondaires sont sollicités par le système alors que le plus souvent, les acteurs principaux ont l'initiative des interactions**
  - **Le plus souvent, les acteurs secondaires sont d'autres systèmes informatiques avec lesquels le système développé est inter-connecté.**



Recenser les cas d'utilisation

- **Il n'y a pas une manière mécanique et totalement objective de repérer les cas d'utilisation.**
  - **Il faut se placer du point de vue de chaque acteur et déterminer**
    - comment il se sert du système,
    - dans quels cas il l'utilise,
    - et à quelles fonctionnalités il doit avoir accès.
  - **Il faut éviter les redondances et limiter le nombre de cas en se situant au bon niveau d'abstraction (par exemple, ne pas réduire un cas à une seule action).**
  - **Il ne faut pas faire apparaître les détails des cas d'utilisation, mais il faut rester au niveau des grandes fonctions du système.**

Attention

Trouver le bon niveau de détail pour les cas d'utilisation est un problème dificile qui nécessite de l'expérience.

Description des cas d'utilisation

- Le diagramme de cas d'utilisation décrit les grandes fonctions d'un système du point de vue des acteurs, mais n'expose pas de façon détaillée le dialogue entre les acteurs et les cas d'utilisation.

Attention

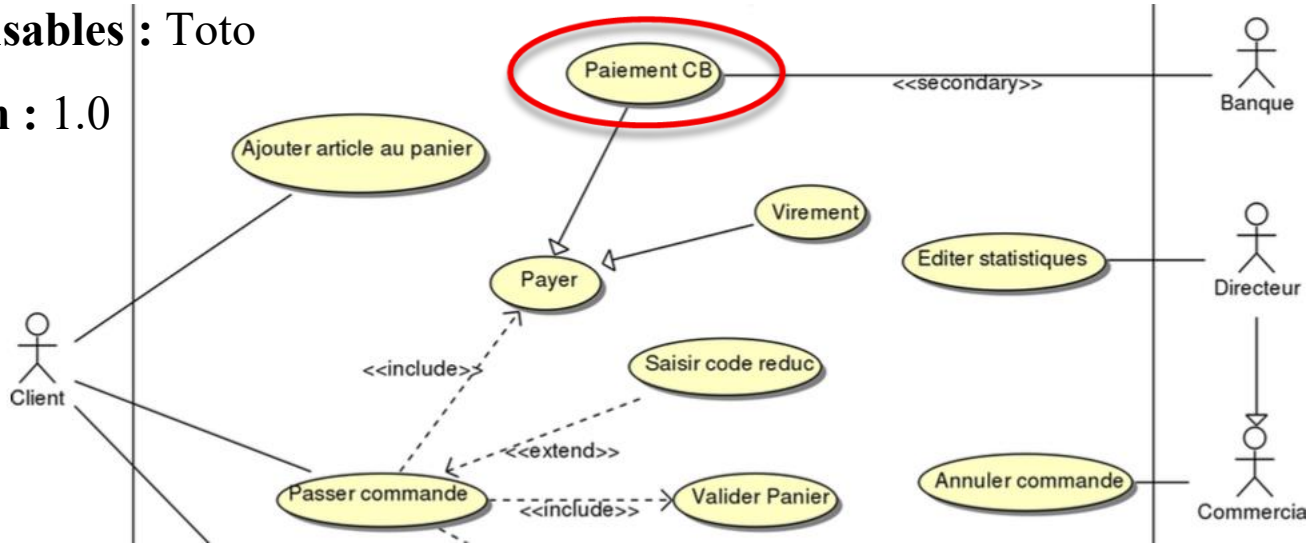
Un simple nom est tout à fait insufisant pour décrire un cas d'utilisation.

➔ **Chaque cas d'utilisation doit être documenté pour qu'il n'y ait aucune ambiguïté concernant son déroulement et ce qu'il recouvre précisément.**

Description textuelle (1/3)

Identification

- **Nom du cas :** Payer CB
- **Objectif :** Détailler les étapes permettant à client de payer par carte bancaire
- **Acteurs :** Client, Banque (secondaire)
- **Date :** 18/09
- **Responsables :** Toto
- **Version :** 1.0



Description textuelle (2/3)

- **Séquencements :**
  - **Le cas d'utilisation commence lorsqu'un client demande le paiement par carte bancaire**
  - **Pré-conditions**
    - Le client a validé sa commande
  - **Enchaînement nominal**
    1. Le client saisit les informations de sa carte bancaire
    2. Le système vérifie que le numéro de CB est correct
    3. Le système vérifie la carte auprès du système bancaire
    4. Le systéme demande au système bancaire de débiter le client
    5. Le système notifie le client du bon déroulement de la transaction
  - **Enchaînements alternatifs**
    1. En (2) : si le numéro est incorrect, le client est averti de l'erreur, et invité à recommencer
    2. En (3) : si les informations sont erronées, elles sont redemandées au client
  - **Post-conditions**
    - La commande est validée
    - Le compte de l'entreprise est crédité

Description textuelle (3/3)

- **Rubriques optionnelles**
  - Contraintes non fonctionnelles :
    - **Fiabilité :** les accès doivent être sécurisés
    - **Confidentialité :** les informations concernant le client ne doivent pas être divulgués
  - Contraintes liées à l'interface homme-machine :
    - Toujours demander la validation des opérations bancaires

Partie 2

Modélisation Objet élémentaire avec UML

- Diagrammes de cas d'utilisation
- **Diagrammes de classes**
- Diagrammes d'objets
- Diagrammes de séquences
- Diagrammes d'activité

Objectif

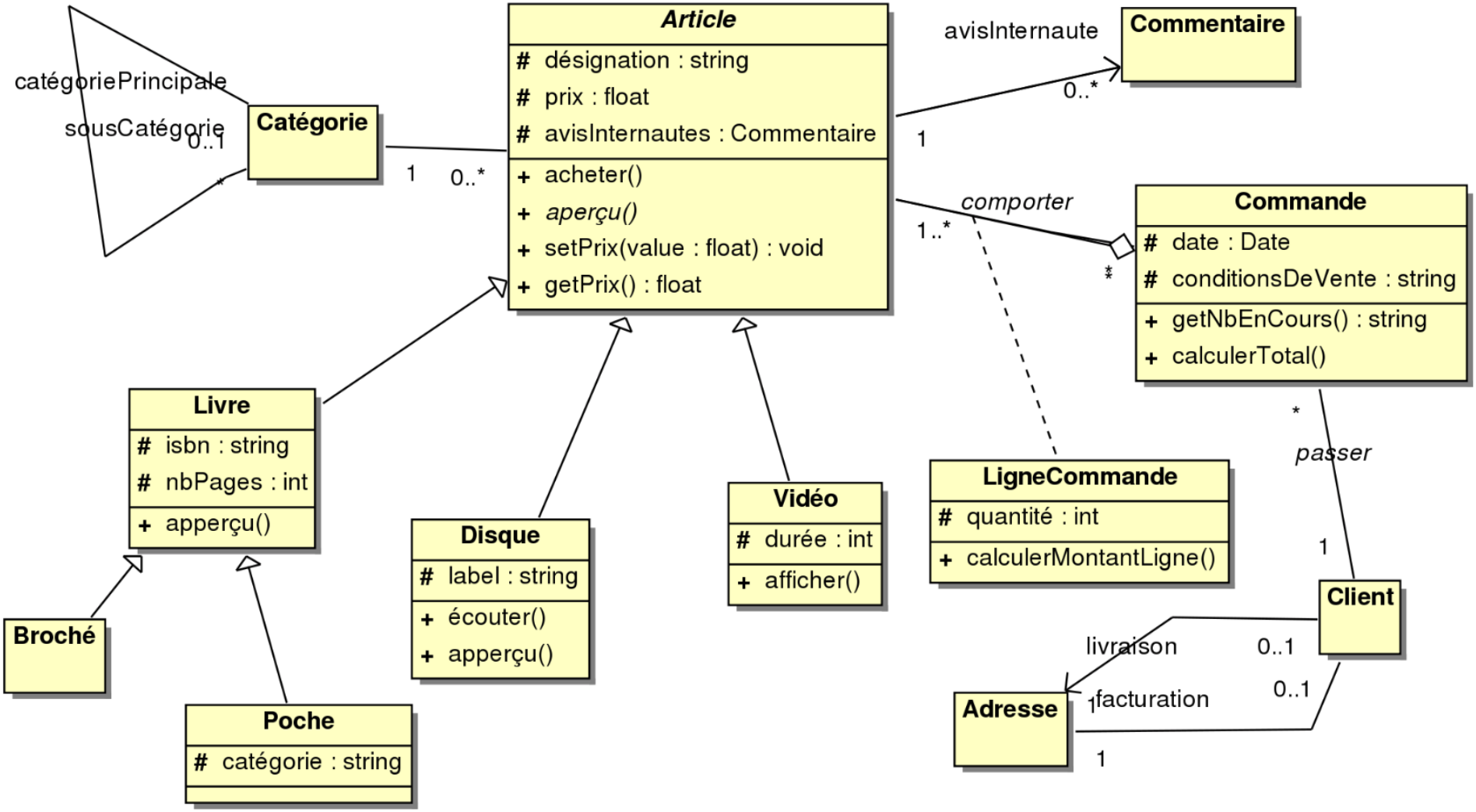
- Les diagrammes de cas d'utilisation modélisent à QUOI sert le système
- Le système est composé d'objets qui interagissent entre eux et avec les acteurs pour réaliser ces cas d'utilisation.

➔ Présenter les concepts UML relatifs à la vue structurelle (diagramme de classes)

➔ Présenter la notation graphique du diagramme de classes UML

➔ Expliquer la sémantique des classes UML (compatible avec la sémantique des langages de programmation orientés objet)

Exemple de diagramme de classes



Concepts et instances

- Une instance est la concrétisation d'un concept abstrait
- Les diagrammes de classes permettent de spéci□er la structure et les liens entre les objets dont le système est composé.
  - Concept : Stylo
  - Instance : le stylo que vous utilisez à ce moment précis est une instance du concept stylo : il a sa propre forme, sa propre couleur, son propre niveau d'usure, etc.
- Un objet est une instance d'une classe
  - Classe : Vidéo
  - Objets : Pink Floyd (Live à Pompey), 2001 Odyssée de l'Espace etc

  Une classe décrit un type d'objets concrets.

  - Une classe spécifie la manière dont tous les objets de même type seront décrits (désignation, label, auteur, etc)
- Un lien est une instance d'association
  - Association : Concept avis d'internaute qui lie commentaire et article
  - Lien : instance [Jean avec son avis négatif], [Paul avec son avis positif]

Vue structurelle du modèle UML

- La vue structurelle du modèle UML est la vue la plus utilisée pour spécifier une application.
- L'objectif de cette vue est de modéliser la structure des différentes classes d'une application orientée objet ainsi que leurs relations

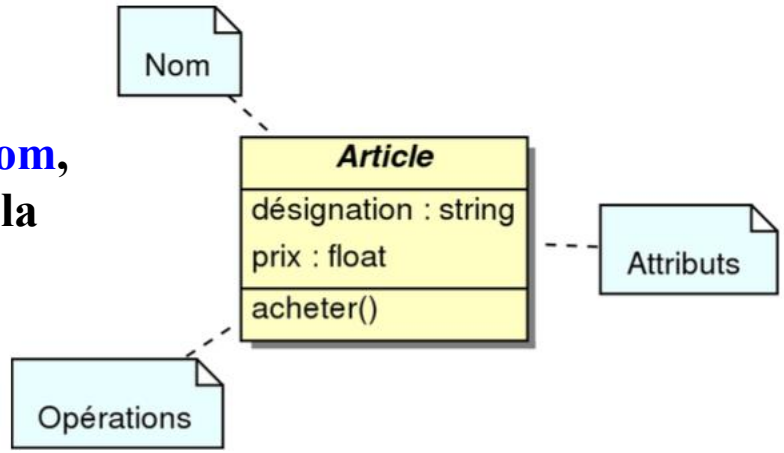
Classe & Objet

Sémantique

- Une classe définit la structure commune d'un ensemble d'objets
- Elle permet la construction d'objets instances de cette classe.
- Elle spécifie l'ensemble des caractéristiques qui composent des objets de même type (Nom, Caractéristiques, Comportement)
- Une classe est identifiée par son nom.

Graphique

- Une classe se représente à l'aide d'un rectangle, qui contient le nom, les attributs et les opérations de la classe



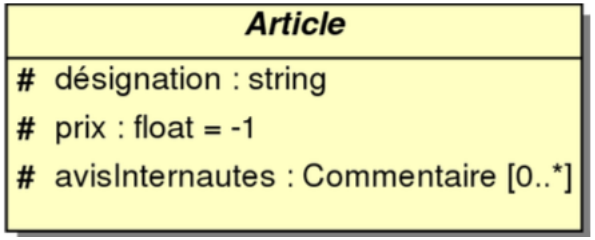
Propriétés (attribut) d'une classe

Sémantique

- Les classes peuvent posséder plusieurs propriétés.
- Une propriété a un nom et un type.
  - Le type peut êre soit une classe UML, soit un type de base (integer, string, boolean, char, real)
  - Les types des propriétés et leurs initialisations ainsi que les modificateurs d'accès peuvent êre précisés dans le modèle
- Les propriétés prennent des valeurs lorsque la classe est instanciée :
  - ils sont en quelque sorte des variables attachées aux objets.

Graphique

- Les propriétés d'une classe ou d'une interface se représentent dans le rectangle représentant la classe ou l'interface.
- Chaque propriété est représentée par son nom et son type



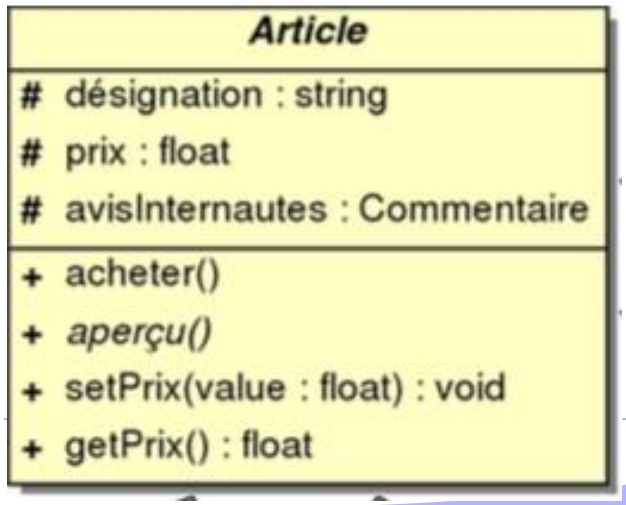
Opérations (Méthodes) d'une classe

Sémantique

- Les classes peuvent posséder plusieurs opérations
- Une opération est la spécification d'une méthode (sa signature) indépendamment de son implantation.
- Une opération a un nom, des paramètres et une valeur de retour
- Une opération peut lever des exceptions
- Les paramètres ainsi que la valeur de retour sont typés et ont un sens (in, out, inout, return)
- UML 2 autorise également la définition des opérations dans n'importe quel langage donné

Graphique

- Les opérations d'une classe se représentent dans le rectangle représentant la classe
- Chaque opération est représentée par son nom et ses paramétres

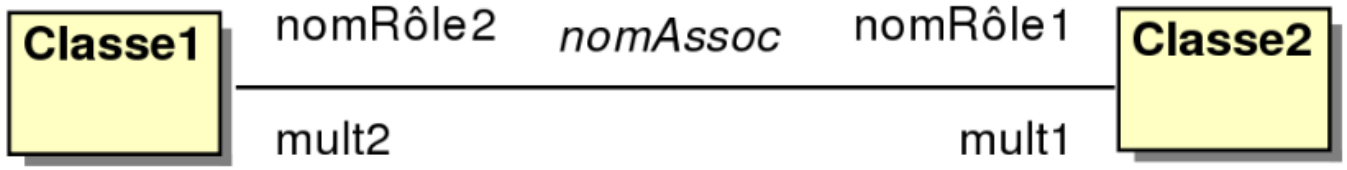


Relations entre classes

- Une relation d'héritage est une relation de généralisation/spécialisation permettant l'abstraction
- Une dépendance est une relation unidirectionnelle exprimant une dépendance sémantique entre les éléments du modèle (flèche ouverte pointillée)
- Une association représente une relation sémantique entre les objets d'une classe
- Une relation d'agrégation décrit une relation de contenance ou de composition

Association

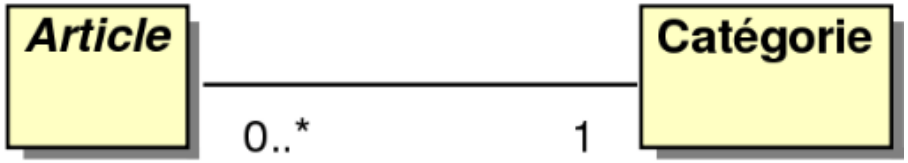
- **Une association est une relation structurelle entre objets.**
  - Une **association** est souvent utilisée pour **représenter les liens posibles entre objets** de classes données.
  - Elle est représentée par un **trait entre classes**



Multiplicités des associations

- La notion de multiplicité permet le contrôle du nombre d'objets intervenant dans chaque instance d'une association
  - Exemple :

    Un article n'appartient qu'à une seule catégorie (1) ; une catégorie concerne 0 ou plusieurs articles (*)



- La syntaxe est MultMin .. MultMax
  - * à la place de MultMax signifie plusieurs

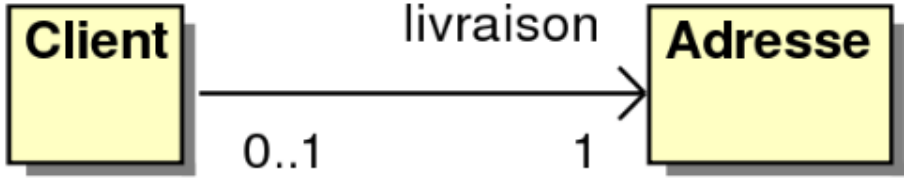
Association unidirectionnelle - Navigation

- En UML, il est possible de rendre chacune des extrémités d'une association navigable ou non
- La navigabilité permet de spécifier dans quel(s) sens il est possible de traverser l'association à l'exécution.
- On restreint la navigabilité d'une association à un seul sens à l'aide d'une flèche

Association unidirectionnelle (de 1 vers 1)

➔ Exemple :

- Les habitants peuvent naviguer vers leurs résidences (et pas l'inverse), ce qui permet d'obtenir, par exemple, le numéro de rue



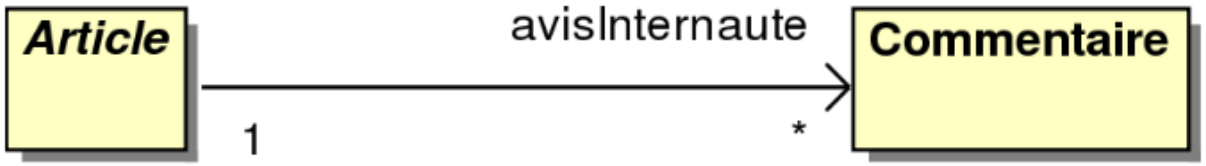

```
public class Adresse{...}

public class Client{
   private Adresse livraison;
   public void setAdresse(Adresse adresse){
      this.livraison = adresse;
   }
   public Adresse getAdresse(){
      return livraison;
   }
}
```

Implimentation

Association unidirectionnelle (de 1 vers *)

➔ Exemple :

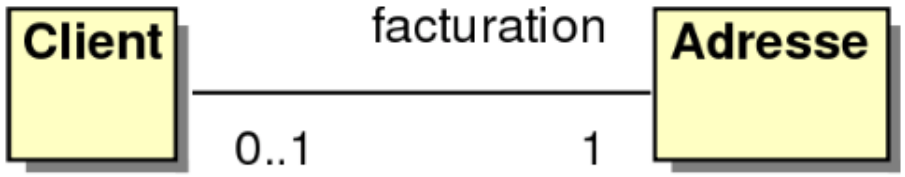
- Connaissant un article on connaît les commentaires, mais pas l'inverse




```
public class Commentaire{...}

public class Article{
  private Vector avisInternaute = new Vector();
  public void addCommentaire(Commentaire commentaire){
    avisInternaute.addElement(commentaire);
  }
  public void removeCommentaire(Commentaire commentaire){
    avisInternaute.removeElement(commentaire);
  }
}
```

Implimentation

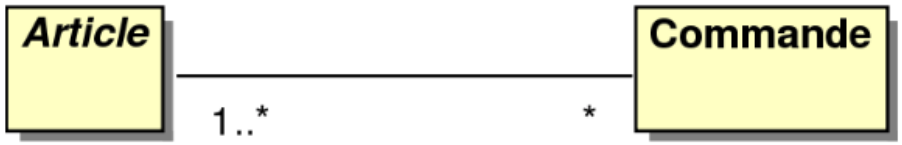
Association bidirectionnelle (de 1 vers 1)




```
public class Client{
  Adresse facturation;
  public void setAdresse(Adresse uneAdresse){
    if(uneAdresse!=null){
      this.facturation = uneAdresse;
      facturation.client = this; // correspondance
      }
    }
}
public class Adresse{
  Client client;
  public void setClient(Client unClient){
    this.client = client;
    client.facturation = this; // correspondance
  }
}
```

Implimentation

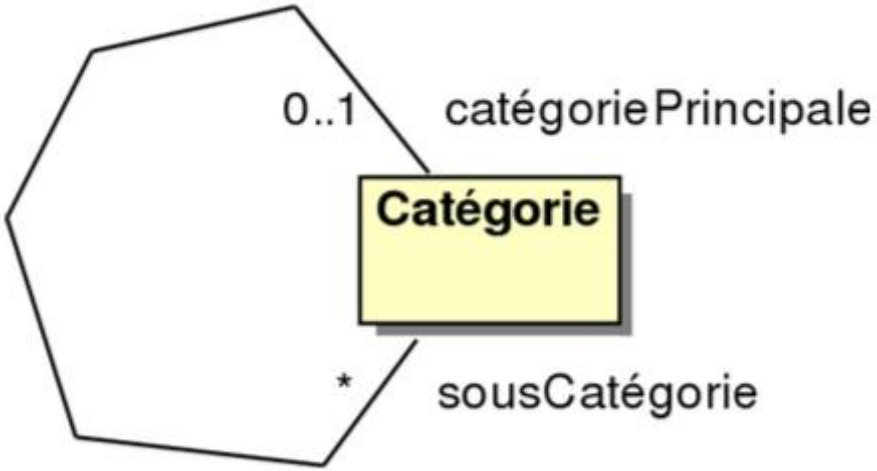
Association bidirectionnelle (de 1 vers *)



Plus difficile : gérer à la fois la cohérence et les collections

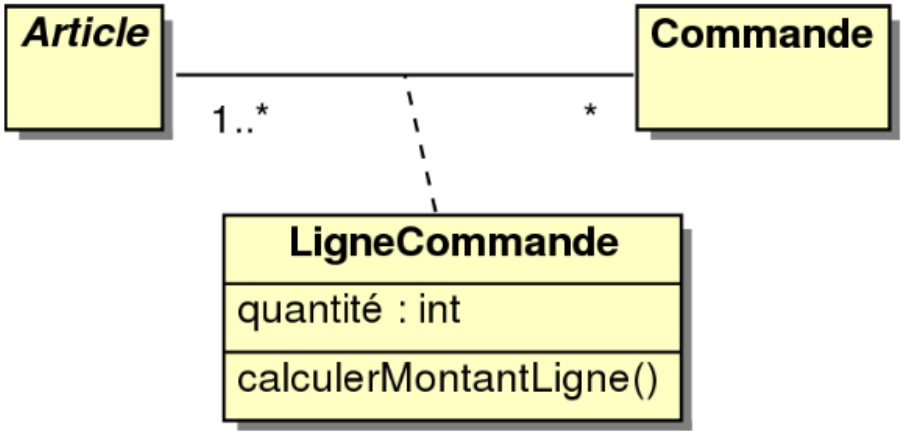
Associations réflexives

- L'association la plus utilisée est l'association binaire (reliant deux classs)
- Parfois, les deux extrémités de l'association pointent vers le même classeur. Dans ce cas, l'association est dite réflexive



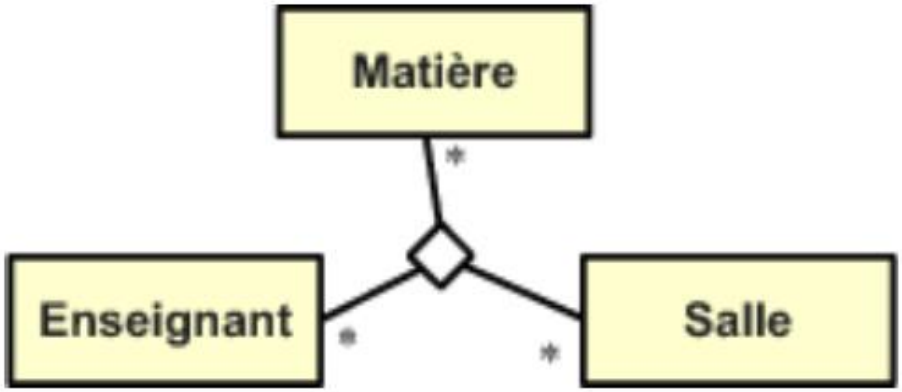
Classe-association

- Une association peut être raffinée et avoir ses propres attributs, qui ne sont disponibles dans aucune des classes qu'elle lie.
- Comme, dans le modèle objet, seules les classes peuvent avoir des attributs, cette association devient alors une classe appelée classe-association



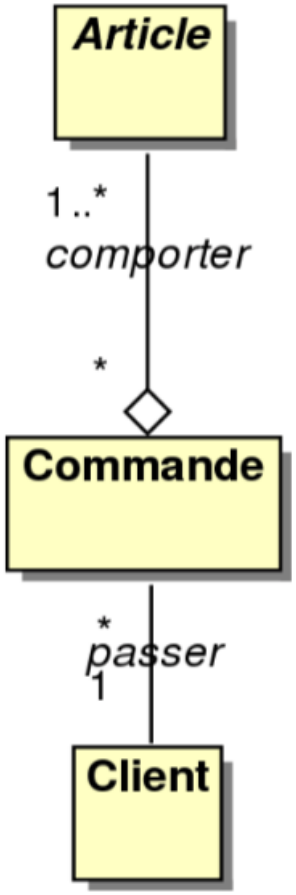
Associations n-aires

- **Une association n-aire lie plus de deux classes**
  - **Notation avec un losange central pouvant éventuellement accueillir une classe-association.**
  - **La multiplicité de chaque classe s'applique à une instance du losange**



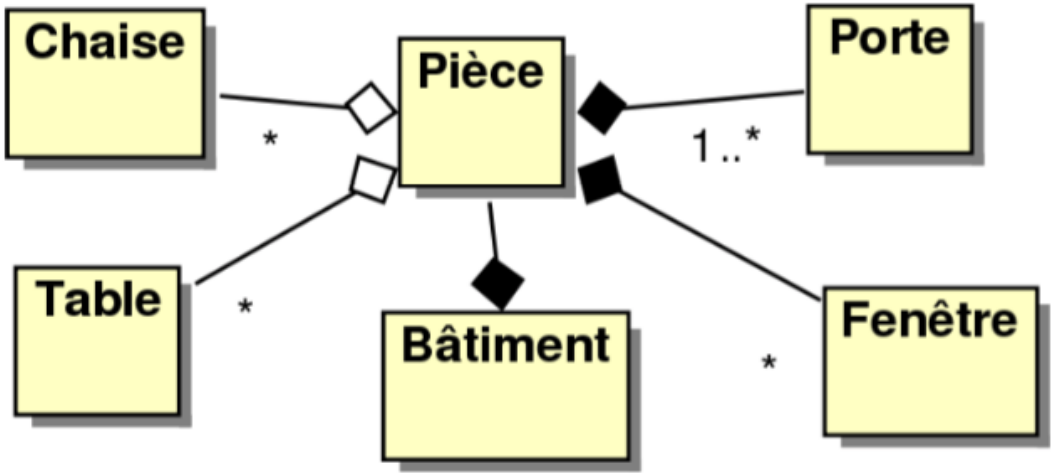
Association de type agrégation

- Une agrégation est une forme particulière d'association
- Elle représente la relation d'inclusion d'un élément dans un ensemble
  - L'ensemble est l'agrégat et la partie l'agrégé
- On représente l'agrégation par l'ajout d'un losange vide du côté de l'agrégat



Association de type composition

- La relation de composition décrit une contenance structurelle entre instances. On utilise un losange plein
- La destruction ou la copie de l'objet composite (l'ensemble) impliquent respectivement la destruction ou la copie de ses composants (les parties)



➔Une instance de la partie n'appartient jamais à plus d'une instance de l'élément composite

Composition ou agrégation ?

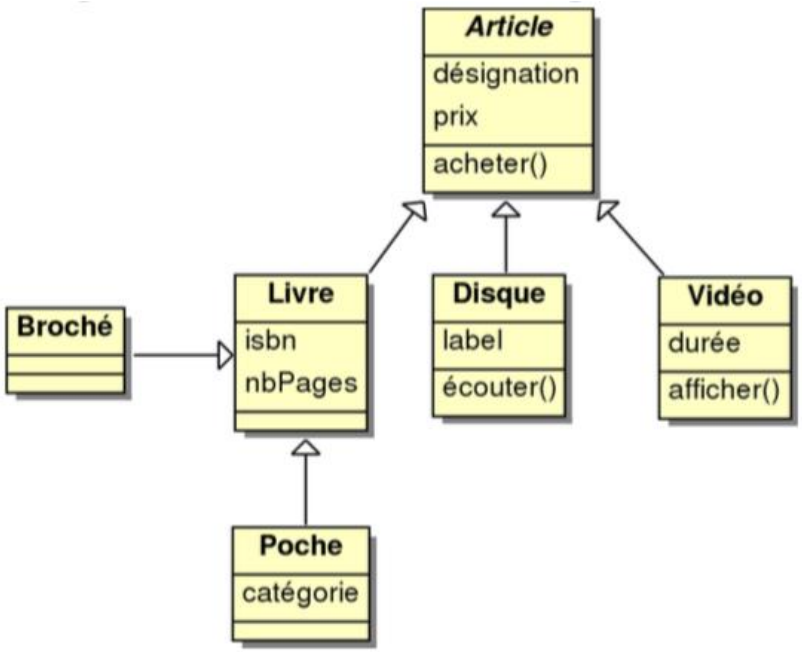
- Dès lors que l'on a une relation du tout à sa partie, on a une relation d'agrégation ou de composition.
- La composition est aussi dite agrégation forte
- Pour décider de mettre une composition plutôt qu'une agrégation, on doit se poser les questions suivantes :
  - Est-ce que la destruction de l'objet composite (du tout) implique nécessairement la destruction des objets composants (les parties) ? C'est le cas si les composants n'ont pas d'autonomie vis-à-vis des composites.
  - Lorsque l'on copie le composite, doit-on aussi copier les composants, ou est-ce qu'on peut les réutiliser, auquel cas un composant peut faire partie de plusieurs composites ?

➔ Si on répond par l'affirmative à ces deux questions, on doit utiliser une composition

Héritage

- **L'héritage une relation de spécialisation/généralisation**
  - **Les éléments spécialisés héritent de la structure et du comportement des éléments plus généraux (attributs et opérations)**
  - **Exemple : Par héritage d'Article, un livre a d'office un prix, une désignation et une opération acheter(), sans qu'il soit nécessaire de le préciser**

```
class Article {
   ...
   void acheter() {
      ...
   }
}
class Livre
        extends Article {
   ...
}
```



Encapsulation - Visibilité des classes, des propriétés et des opérations

Sémantique

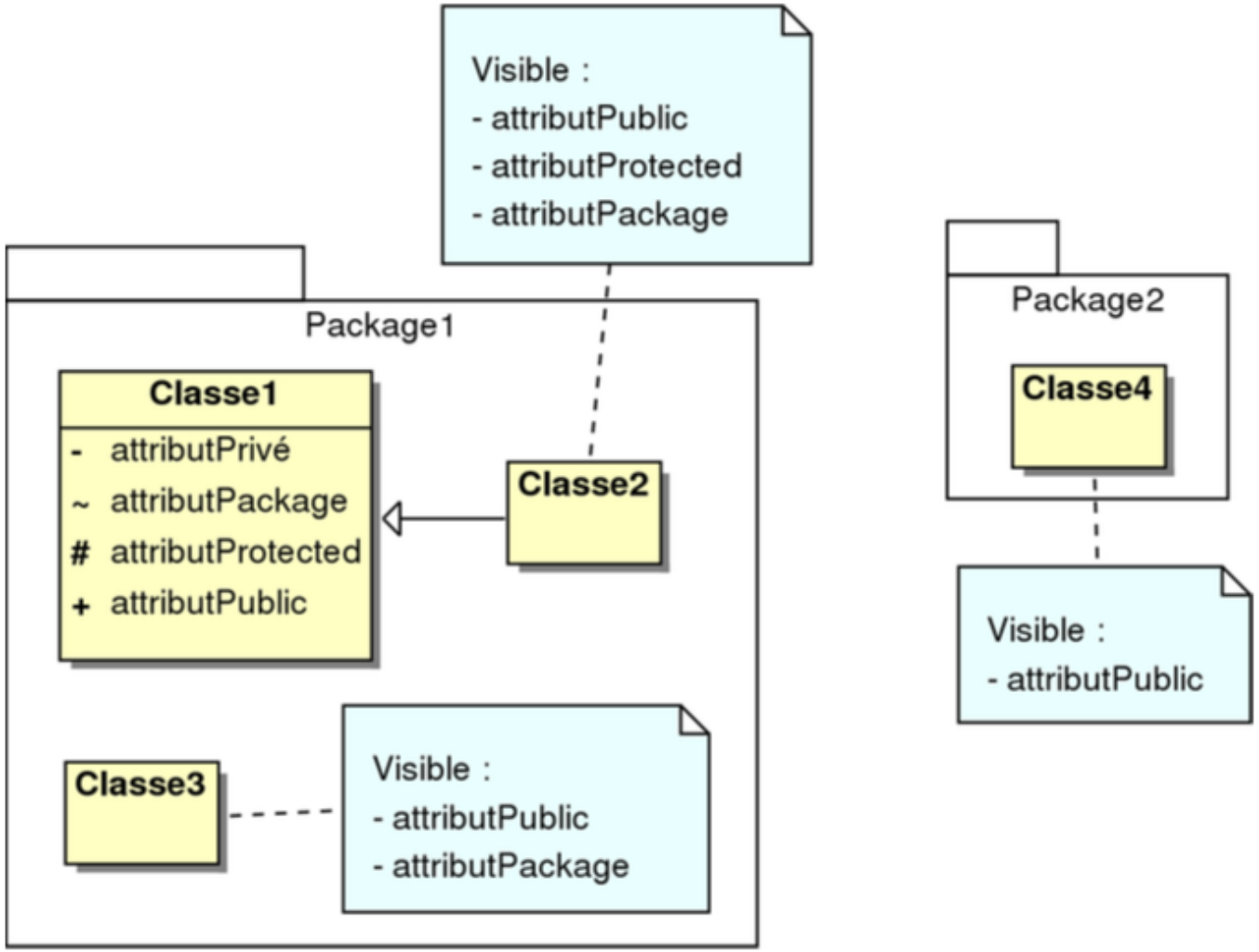
- Il est possible de préciser la visibilité des propriétés et des opérations des classes
- Les visibilités portent sur les accès aux propriétés et aux opérations
- Les visibilités proposées par UML 2.0 sont les suivantes : Public, Protected, Private et Package

Graphique

- Dans la représentation graphique de l'élément, les visibilités sont représentées de la façon suivante :

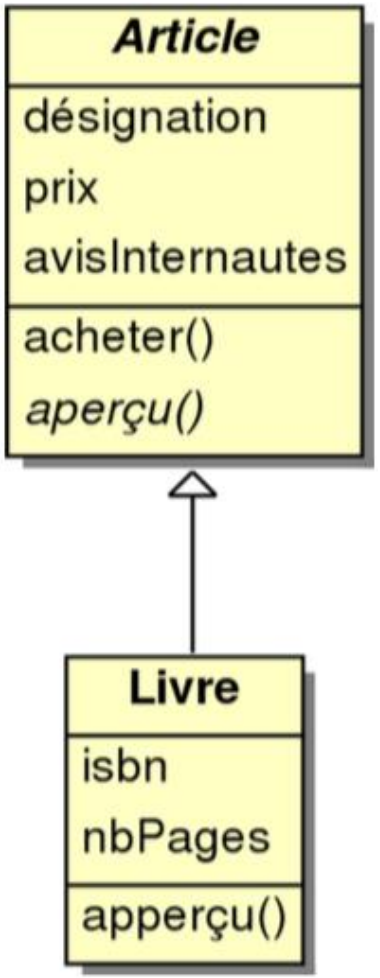
Public	**+**	**propriété ou classe visible partout**
Protected	**#**	**propriété ou classe visible dans la classe et par tous ses descendants**
Private	**-**	**ropriété ou classe visible uniquement dans la classe**
Package	**~**	**propriété ou classe visible uniquement dans le paquetage**

Exemple d'encapsulation



Relation d'héritage et propriétés

- La classe enfant possède toutes les propriétés de ses classes parents (attributs et opérations)
  - La classe enfant est la classe spécialisée (ici Livre)
  - La classe parent est la classe générale (ici Article)
- Toutefois, elle n'a pas accès aux propriétés privées.



Terminologie de l'héritage

- Une classe enfant peut redéfinir (même signature) une ou plusieurs méthodes de la classe parent
  - Sauf indications contraires, un objet utilise les opérations les plus spécialisées dans la hiérarchie des classes
  - La surcharge d'opérations (même nom, mais signatures des opérations différentes) est possible dans toutes les classes
- Toutes les associations de la classe parent s'appliquent, par défaut, aux classes dérivées (classes enfant)
- Principe de substitution : une instance d'une classe peut être utilisée partout où une instance de sa classe parent est attendue

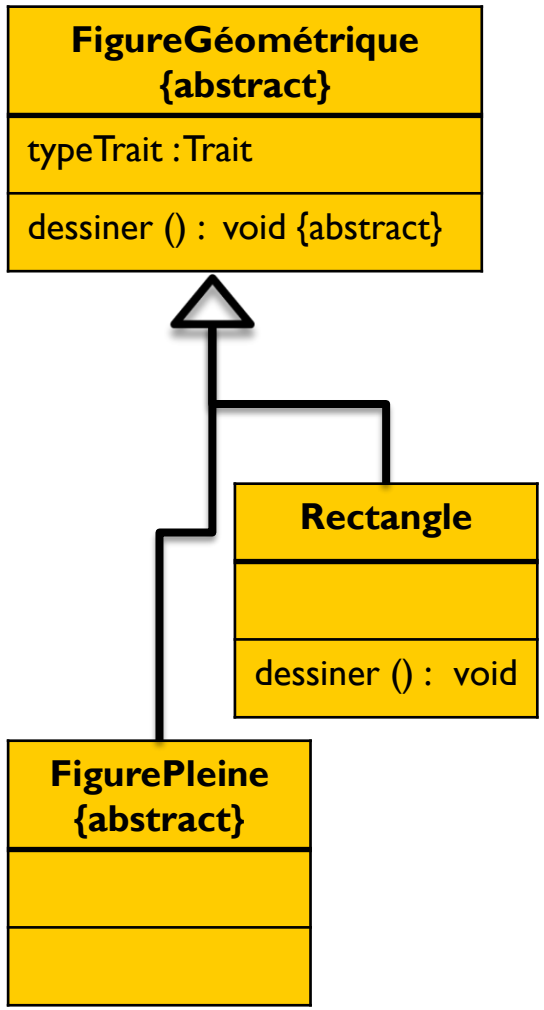
  Exemple
  - Toute opération acceptant un objet d'une classe Animal doit accepter tout objet de la classe Chat (l'inverse n'est pas toujours vrai)

Classes abstraites

Sémantique

- Une classe UML peut être abstraite. Dans ce cas, elle ne peut pas directement instancier un objet
- Dans une classe abstraite, il est possible de préciser que certaines propriétés et certaines opérations sont abstraites
- Une opération est dite abstraite lorsqu'on connaît son entête mais pas la manière dont elle peut être réalisée.
- Une classe est dite abstraite lorsqu'elle définit au moins une méthode abstraite ou lorsqu'une classe parent contient une méthode abstraite non encore réalisée.

➔ Il appartient aux classes enfant de définir les methodes abstraites.

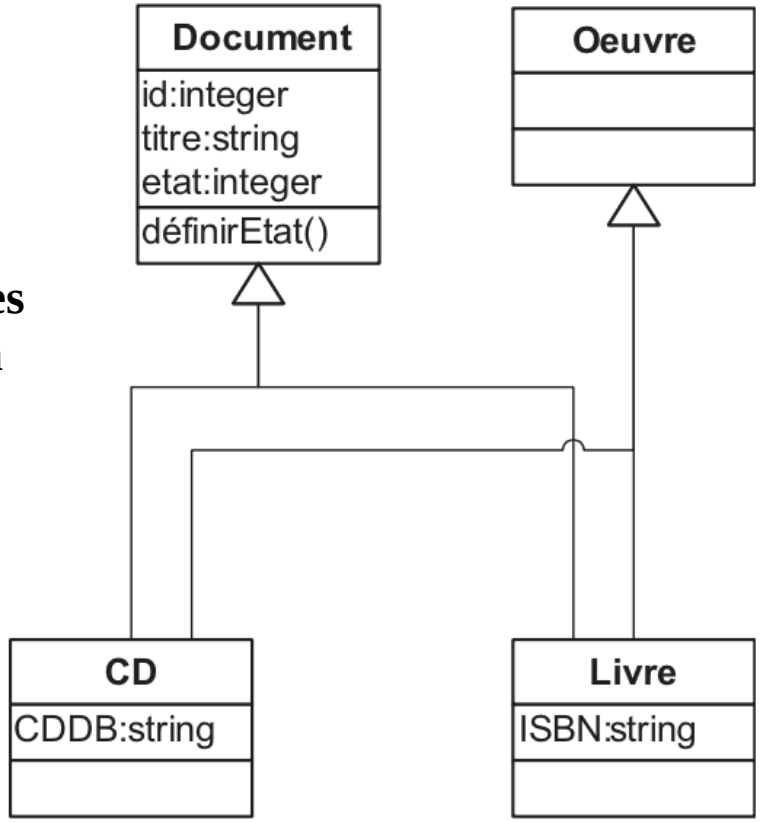


Graphique

- En UML 2.0, il n'existe pas de représentation graphique particulière pour les classes abstraites. En UML 1.4, il fallait mettre le nom de la classe en italique

Héritage multiple

- **Une classe peut avoir plusieurs classes parents. On parle alors d'héritage multiple.**
  - **Le langage C++ est un des langages objet permettant son implantation effective**
  - **Java ne le permet pas ..**



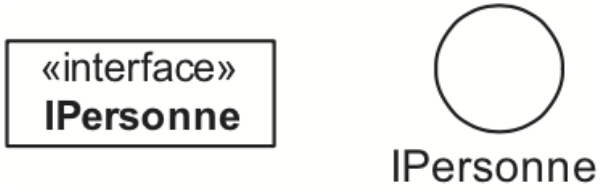
Interface

Sémantique

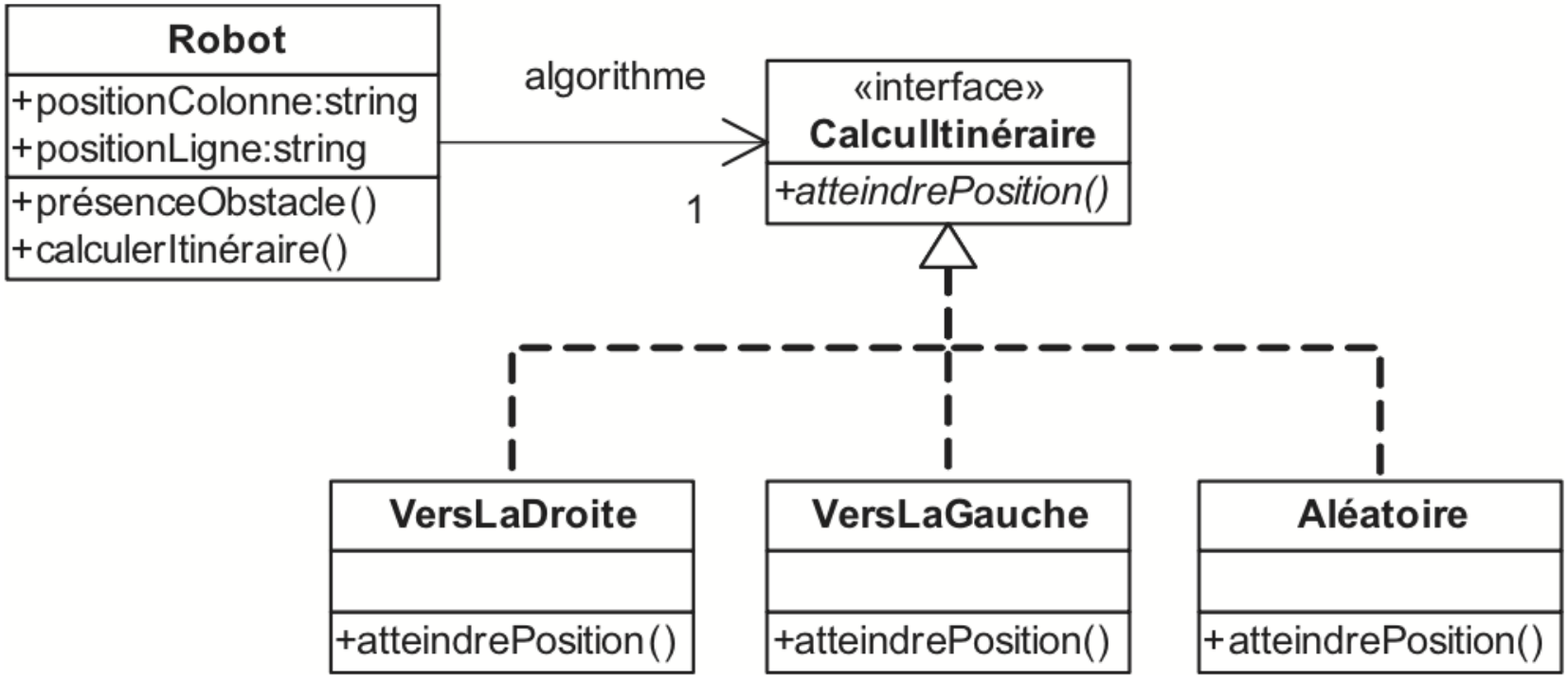
- En UML, une interface définit un contrat que doivent respecter les classes qui réalisent l'interface
- Les objets instances des classes qui réalisent des interfaces sont aussi des instances des interfaces
- Une classe peut réaliser plusieurs interfaces, et une interface peut être réalisée par plusieurs classes

Sémantique

- Une interface est identifiée par son nom
- Une interface se représente de deux façons : soit à l'aide d'un rectangle contenant le nom de l'interface, au-dessus duquel se trouve la chaîne de caractères «interface», soit à l'aide d'un cercle, au-dessous duquel se trouve le nom de l'interface



Exemple d'interface



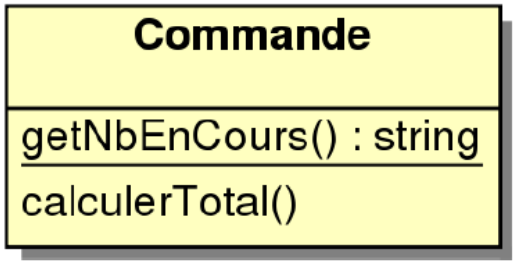
Attributs & Opérations de classe

Attributs de classe

- Par défaut, les valeurs des attributs définis dans une classe diffèrent d'un objet à un autre. Parfois, il est nécessaire de définir un attribut de classe qui garde une valeur unique et partagée par toutes les instances.
- Graphiquement, un attribut de classe est souligné

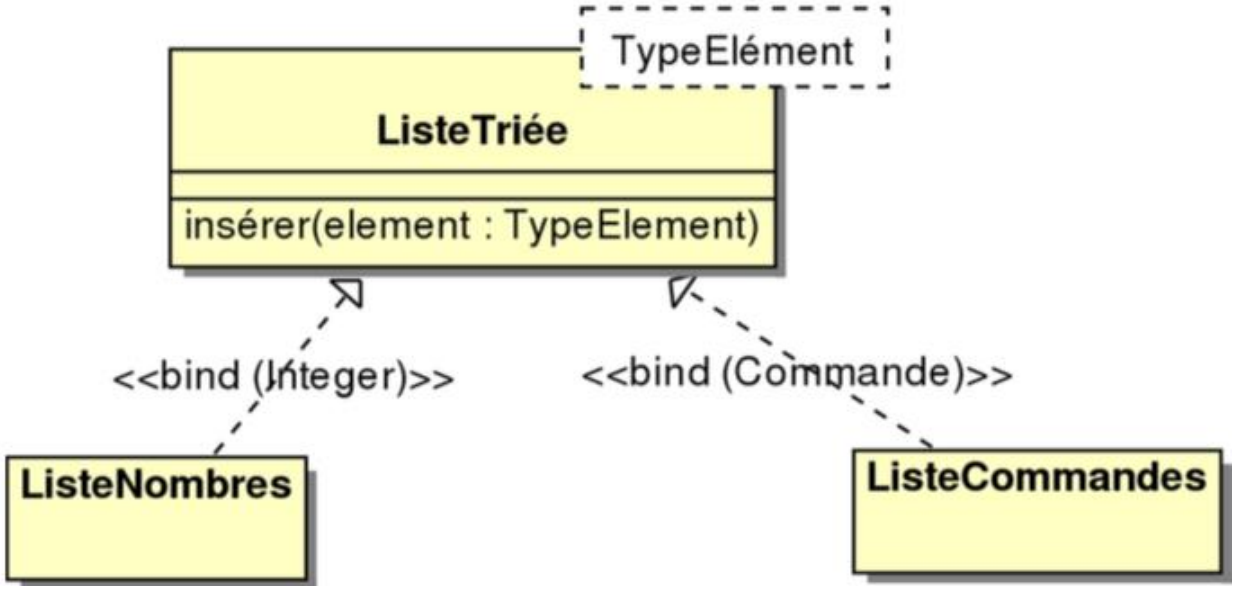
Opérations de classe

- Semblable aux attributs de classe
- Une opération de classe est une propriété de la classe, et non de ses instances
- Elle n'a pas accès aux attributs des objets de la classe.



Classe paramétrée

- **Pour définir une classe générique et paramétrable en fonction de valeurs et/ou de types :**
  - Définition d'une classe paramétrée par des éléments spécifiés dans un rectangle en pointil- lés ;
  - Utilisation d'une dépendance stéréotypée bind pour définir des classes en fonction de la classe paramétrée.



Diagrammes de classes à différentes étapes de la conception

- **On peut utiliser les diagrammes de classes pour représenter un système à différents niveaux d'abstraction**
  - Le point de vue spécification met l'accent sur les interfaces des classes plutôt que sur leurs contenus.
  - Le point de vue conceptuel capture les concepts du domaine et les liens qui les lient. Il s'intéresse peu ou prou à la manière éventuelle d'implémenter ces concepts et relations et aux langages d'implantation.
  - Le point de vue implantation, le plus courant, détaille le contenu et l'implantation de chaque classe
- **Les diagrammes de classes s'étoffent à mesure qu'on va de hauts niveaux à de bas niveaux d'abstraction (de la spécification vers l'implantation)**

Construction d'un diagramme de classes

- **Trouver les classes du domaine étudié**
  - Souvent, concepts et substantifs du domaine.
- **Trouver les associations entre classes**
  - Souvent, verbes mettant en relation plusieurs classes.
- **Trouver les attributs des classes**
  - Souvent, substantifs correspondant à un niveau de granularité plus fin que les classes. Les adjectifs et les valeurs correspondent souvent à des valeurs d'attributs.
- **Organiser et simplifier le modèle en utilisant l'héritage ;**
- **Tester les chemins d'accès aux classèes ;**
- **Itérer et raffiner le modèle.**

Partie 2

Modélisation Objet élémentaire avec UML

- Diagrammes de cas d'utilisation
- Diagrammes de classes
- **Diagrammes d'objets**
- Diagrammes de séquences
- Diagrammes d'activité

Objectif

- **Le diagramme d'objets représente les objets d'un système à un instant donné. Il permet de :**
  - Illustrer le modèle de classes (en montrant un exemple qui explique le modèle)
  - Préciser certains aspects du système (en mettant en évidence des détails imperceptibles dans le diagramme de classes)
  - Exprimer une exception (en modélisant des cas particuliers, des connaissances non généralisables . . .)

➔ **Si le diagramme de classes modélise des règles, le diagramme d'objets modélise des faits**

Représentation des objets

- Comme les classes, on utilise des cadres compartimentés.
- En revanche, les noms des objets sont soulignés et on peut rajouter son identifiant devant le nom de sa classe
- Les valeurs (a) ou l'état (f) d'un objet peuvent être spéci□ ées
- Les instances peuvent être anonymes (a,c,d), nommées (b,f), orphelines (e), multiples (d) ou stéréotypées (g)

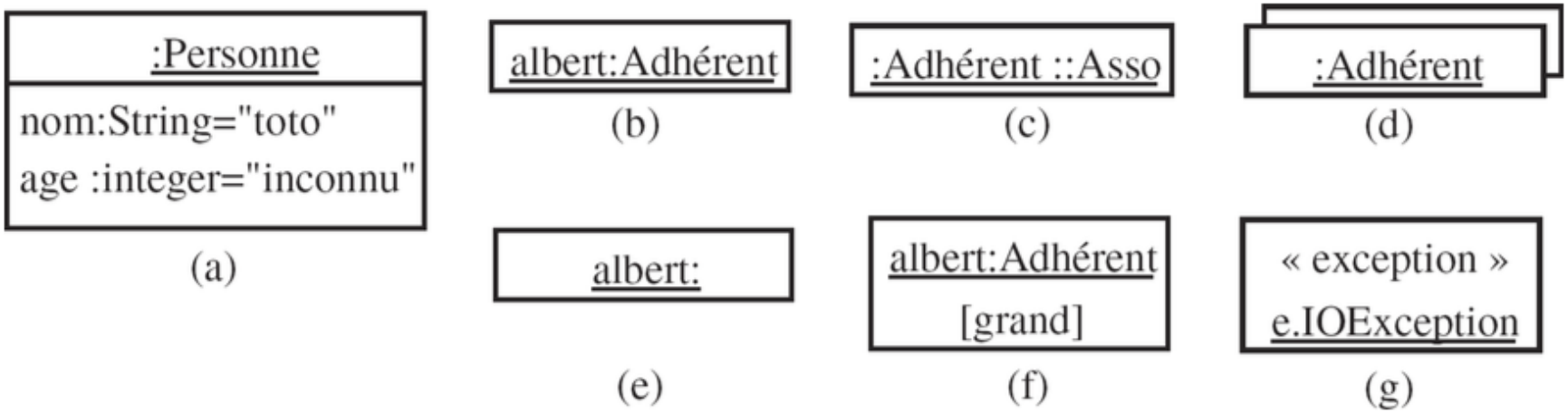
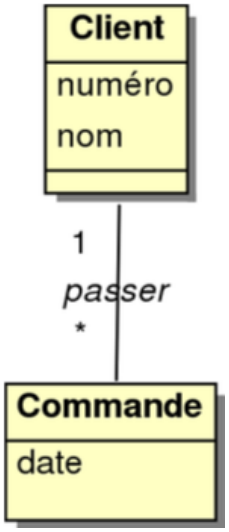
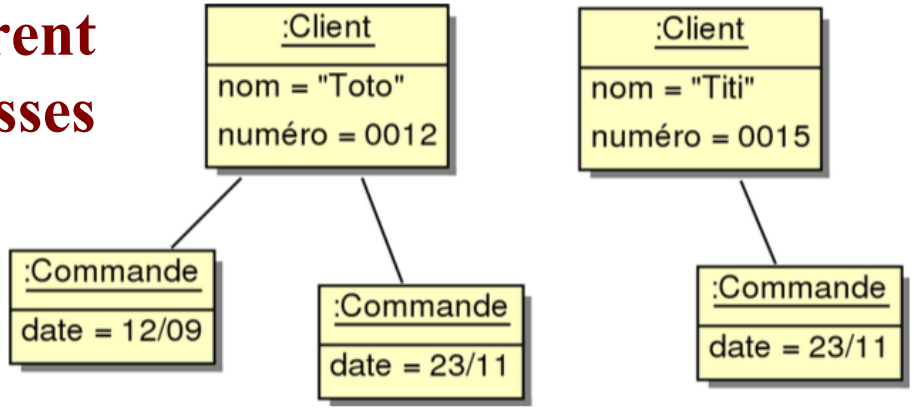


Diagramme de classes et diagramme d'objets

- **Le diagramme de classes contraint la structure et les liens entre les objets**



- **Diagramme d'objet cohérent avec le diagramme de classes**



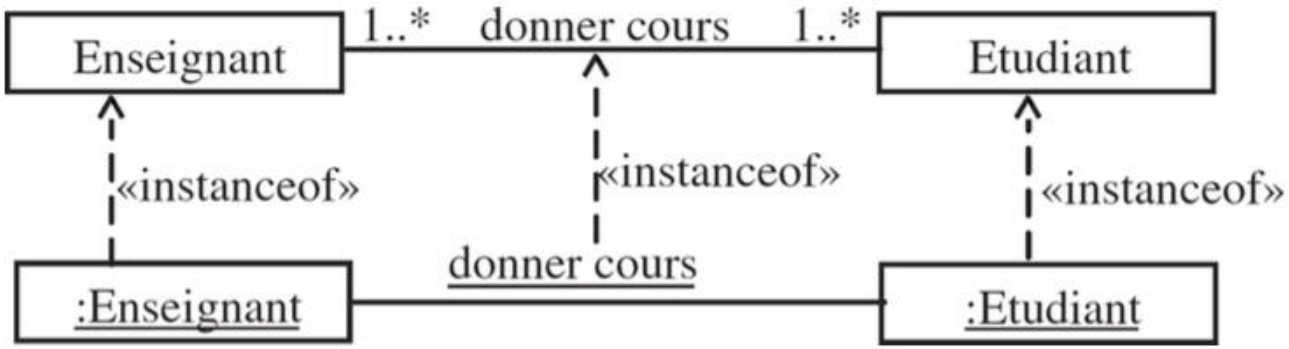
Liens

- Un lien est une instance d'une association
- Un lien se représente comme une association mais s'il a un nom, il est souligné

Attention
Naturellement, on ne représente pas les multiplicités qui n'ont aucun sens au niveau des Objets

Relation de dépendance d'instanciation

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



Partie 2

Modélisation Objet élémentaire avec UML

- Diagrammes de cas d'utilisation
- Diagrammes de classes
- Diagrammes d'objets
- **Diagrammes de séquences**
- Diagrammes d'activité

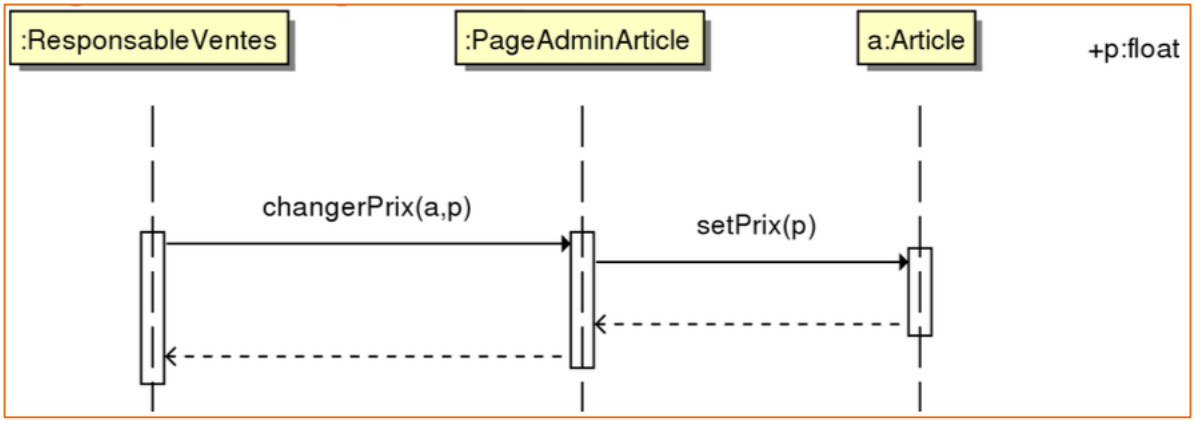
Objectif

- 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 coeur d'un système interagissent en s'échangent des messages
  - Les acteurs interagissent avec le système au moyen d'IHM (Interfaces Homme-Machine)

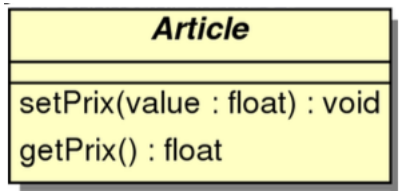
Exemple d'interaction

- **Cas d'utilisation**

- **Diagramme de séquences correspondant**



- **Opérations nécessaires dans le diagramme de classes**

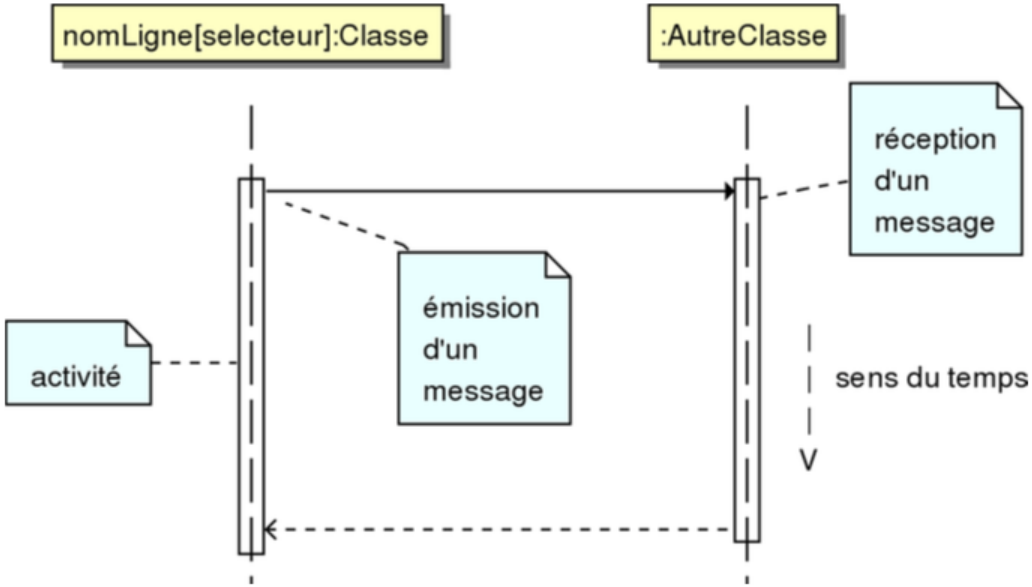


Ligne de vie

- Une ligne de vie 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 (objets ou acteurs)
- Plusieurs types de messages existent, dont les plus courants :
  - l'envoi d'un signal
  - l'invocation d'une opération (appel de méthode)
  - la création ou la destruction d'un objet
- La réception des messages provoque une période d'activité (rectangle vertical sur la ligne de vie) marquant le traitement du message
  - spécification d'exécution dans le cas d'un appel de méthode).

Principaux types de messages

- Un message synchrone bloque l'expéditeur jusqu'à la réponse du destinataire. Le flot de contrôle passe de l'émetteur au récepteur.
  - Typiquement : appel de méthode
    - Si un objet A invoque une méthode d'un objet B, A reste bloqué tant que B n'a pas terminé

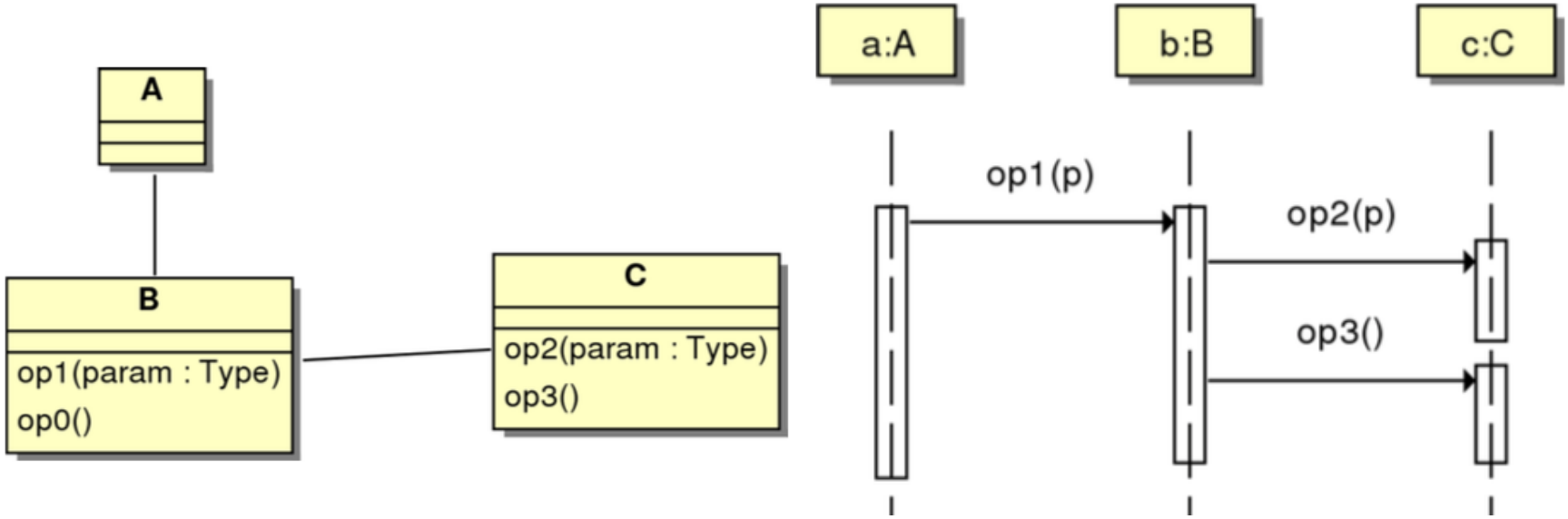
  - On peut associer aux messages d'appel de méthode un message de retour (en pointillés) marquant la reprise du contrôle par l'objet émetteur du message synchrone.

- Un message asynchrone n'est pas bloquant pour l'expéditeur. Le message envoyé peut être pris en compte par le récepteur à tout moment ou ignoré
  - Typiquement : envoi de signal (voir stéréotype de classe signal).

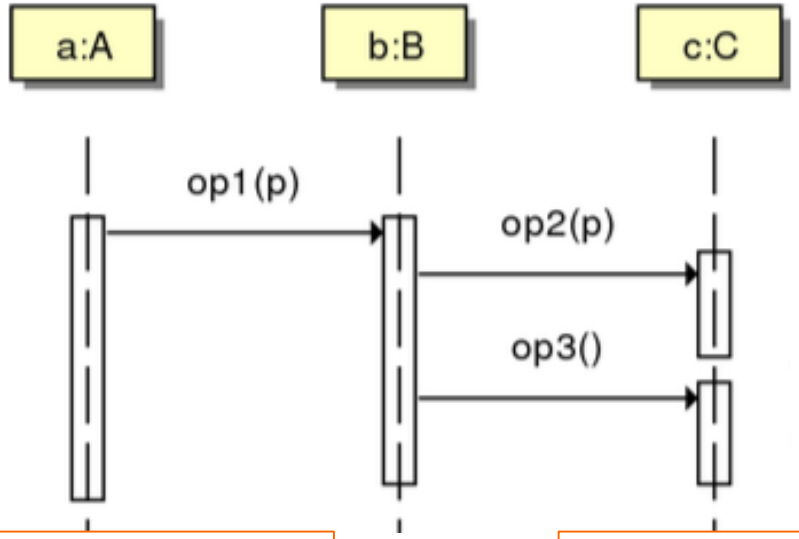
Correspondance messages / opérations

- **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



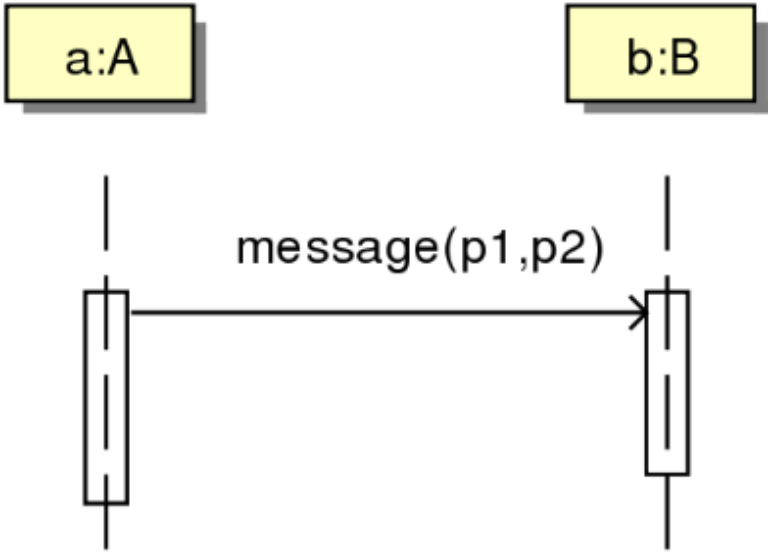

```
class B {
   C c;
   op1(p:Type){
      c.op2(p);
      c.op3();
   }
}
```

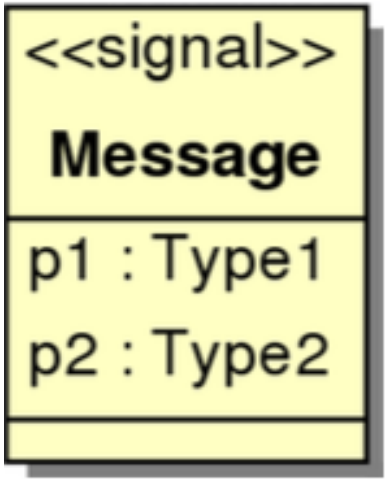
```
class C {
   op2(p:Type){
      ...
   }
   op3(){
      ...
   }
}
```

Correspondance messages / signaux

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

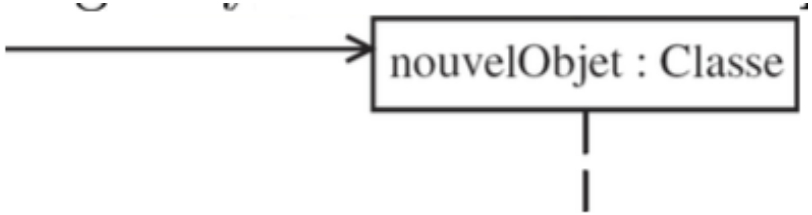
Les signaux sont des objets dont la classe est stéréotypée « signal » et dont les attributs (porteurs d'information) correspondent aux paramètres du messag



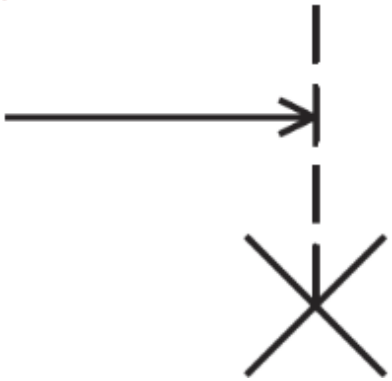


Création et destruction de lignes de vie

- La création d'un objet est matérialisée par une flèche qui pointe sur le sommet d'une ligne de vie
  - On peut aussi utiliser un message asynchrone ordinaire portant le nom create



- La destruction d'un objet est matérialisée par une croix qui marque la fin de la ligne de vie de l'objet



Messages complets, perdus et trouvés

- Un message complet est tel que les événements d'envoi et de réception sont connus
  - Un message complet est représenté par une flèche partant d'une ligne de vie et arrivant à une autre ligne de vie
- Un message perdu est tel que l'événement d'envoi est connu, mais pas l'événement de réception.

  - La flèche part d'une ligne de vie mais arrive sur un cercle indépendant marquant la méconnaissance du destinataire
- Un message trouvé est tel que l'événement de réception est connu, mais pas l'événement d'émission

Syntaxe des messages

- La syntaxe des messages est :

```
nomSignalOuOperation(parametres)
```

- La syntaxe des arguments est la suivante :

```
nomParametre=valeurParametre
```

- Pour un argument modifiable :

```
nomParametre:valeurParametre
```

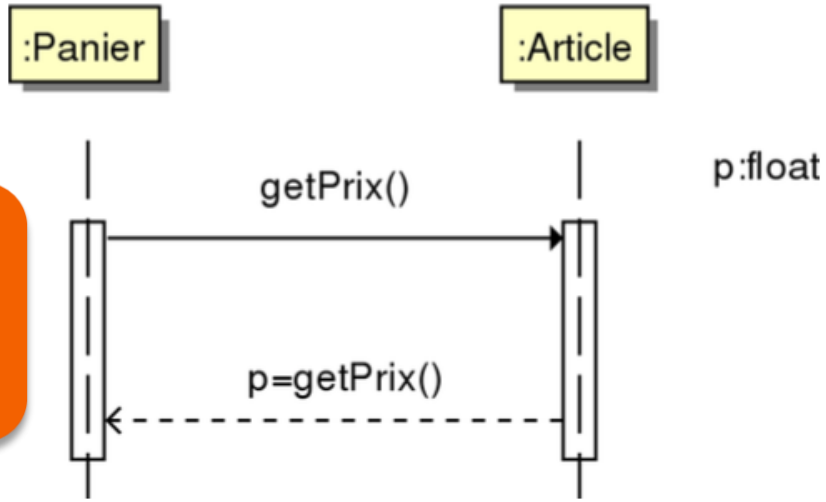
➔ Exemples

- `appeler("Capitaine Hadock", 54214110)`
- `afficher(x,y)`
- `initialiser(x=100)`
- `f(x:12)`

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 spécifiifier le résultat de la méthode invoquée**

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



Syntaxe des messages de retour

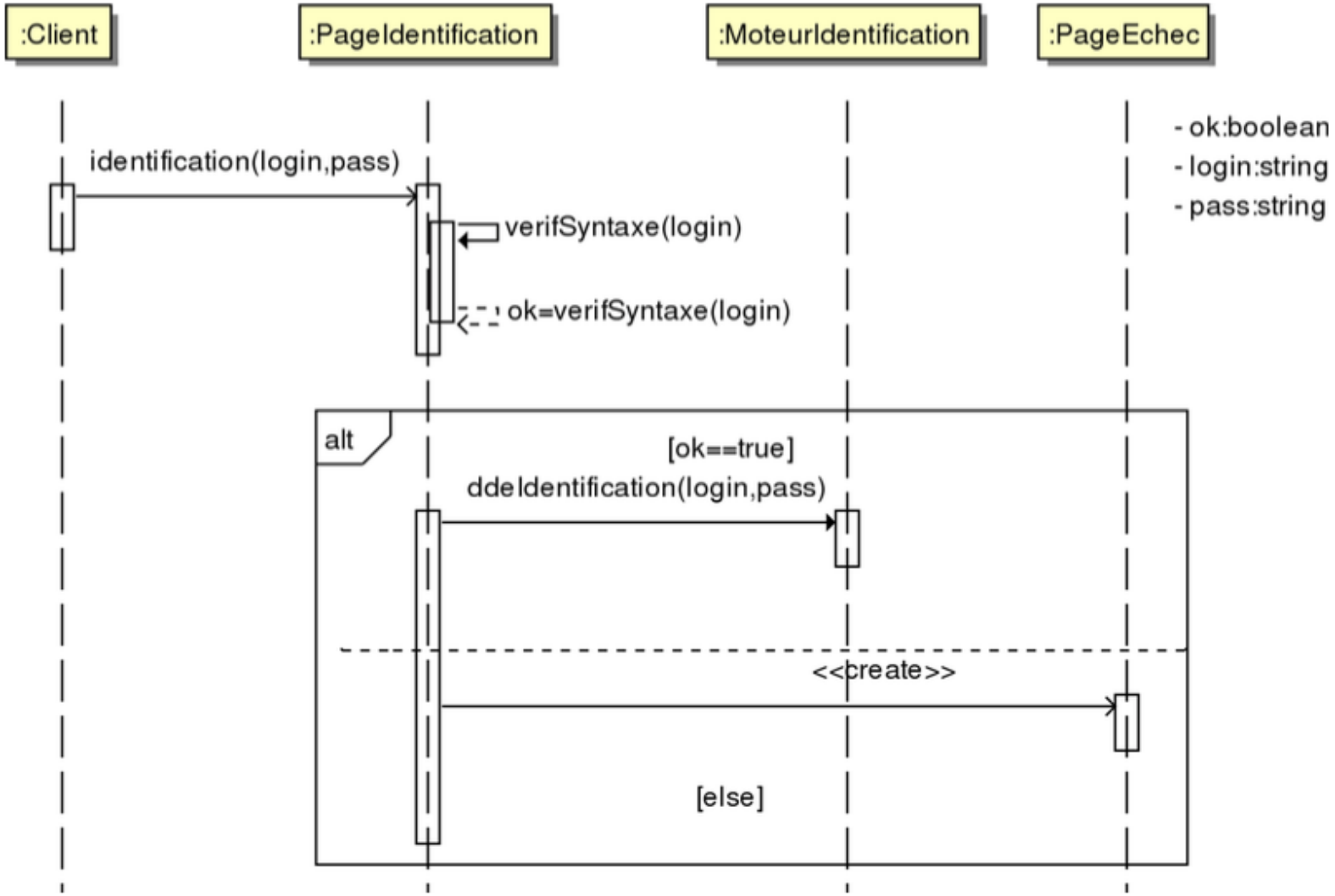
- La syntaxe des messages de retour est :
  - `attributCible=nomOperation(params):valeurRetour`
- La syntaxe des paramètres est :
  - `nomParam=valeurParam`

    Ou
  - `nomParam:valeurParam`
- Les messages de retour sont représentés en pointillés.

Fragment combiné

- Un fragment combiné permet de décomposer une interaction complexe en fragments suffisamment simples pour être compris.
  - Recombiner les fragments restitue la complexité.
- Un fragment combiné se représente de la même façon qu'une interaction. Il est représenté un rectangle dont le coin supérieur gauche contient un pentagone
  - Dans le pentagone figure le type de la combinaison (appelé opérateur d'interaction)

Exemple de fragment avec l'opérateur conditionnel



Type d'opérateurs d'interaction

- **Opérateurs de branchement (choix et boucles) :**

  `alternative`, `option`, `break` et `loop`

- **Opérateurs contrôlant l'envoi en parallèle de messages :**

  `parallel` et `critical region`

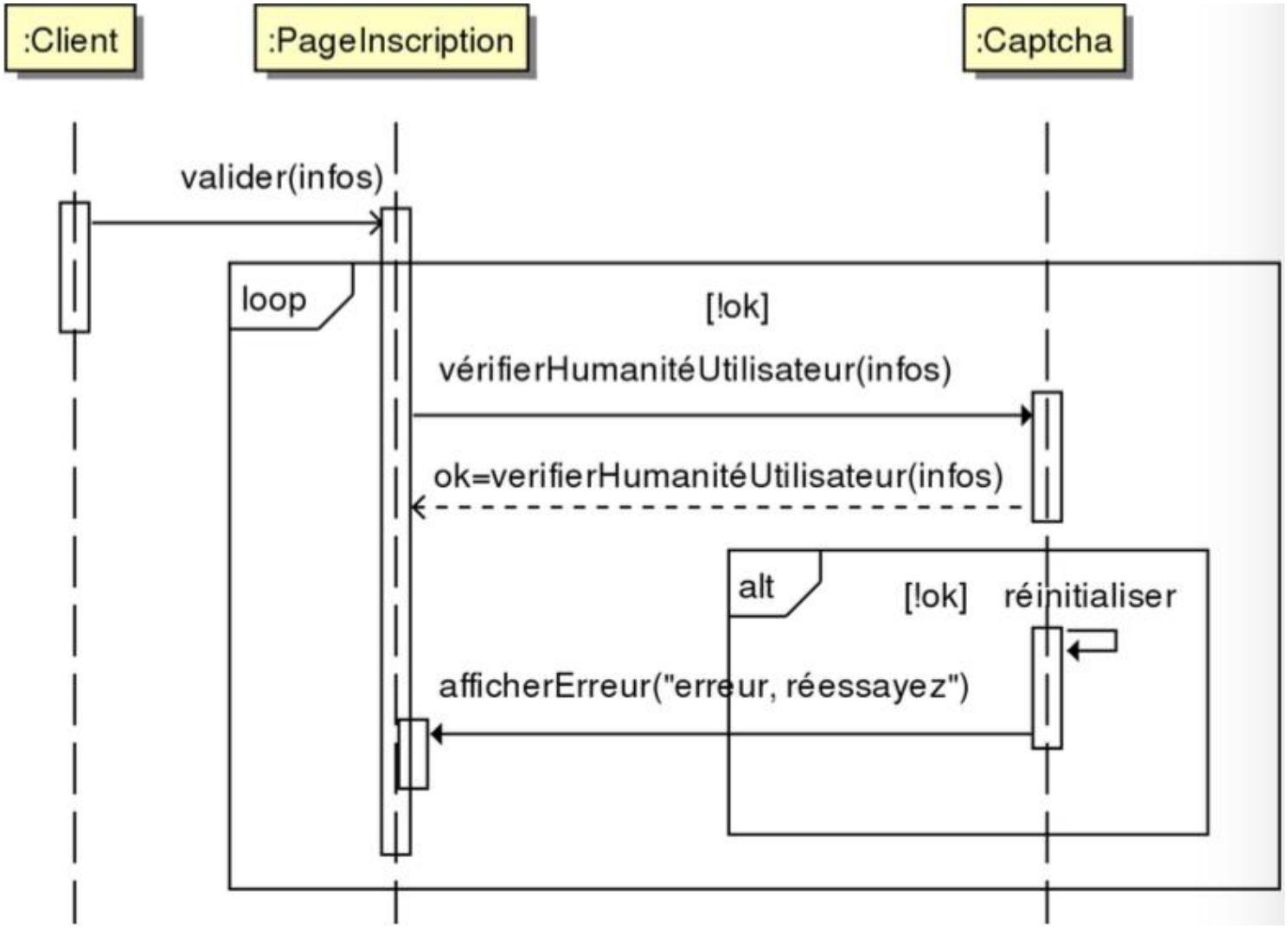
- **Opérateurs contrôlant l'envoi de messages :**

  `ignore`, `consider`, `assertion` et `negative`

- **Opérateurs fixant l'ordre d'envoi des messages :**

  `weak sequencing` et `strict sequencing`

Opérateur de boucle



Syntaxe de l'opérateur `loop`

Syntaxe d'une boucle :

```
loop(minNbIterations, maxNbIterations)
```

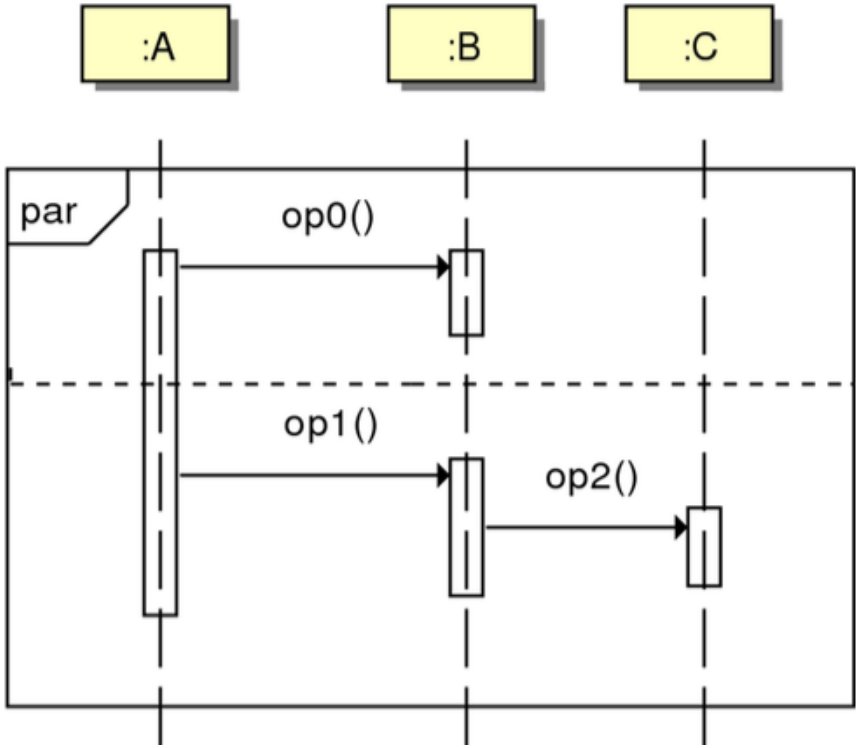
- La boucle est répétée au moins `minNbItérations` fois avant qu'une éventuelle condition booléenne ne soit testée (la condition est placée entre crochets dans le fragment)
- Tant que la condition est vraie, la boucle continue, au plus `maxNbItérations` fois.

Notations :

- `loop(valeur)` est équivalent à `loop(valeur,valeur)`.
- `loop` est équivalent à `loop(0,*)`, où `*` signifie "illimité"

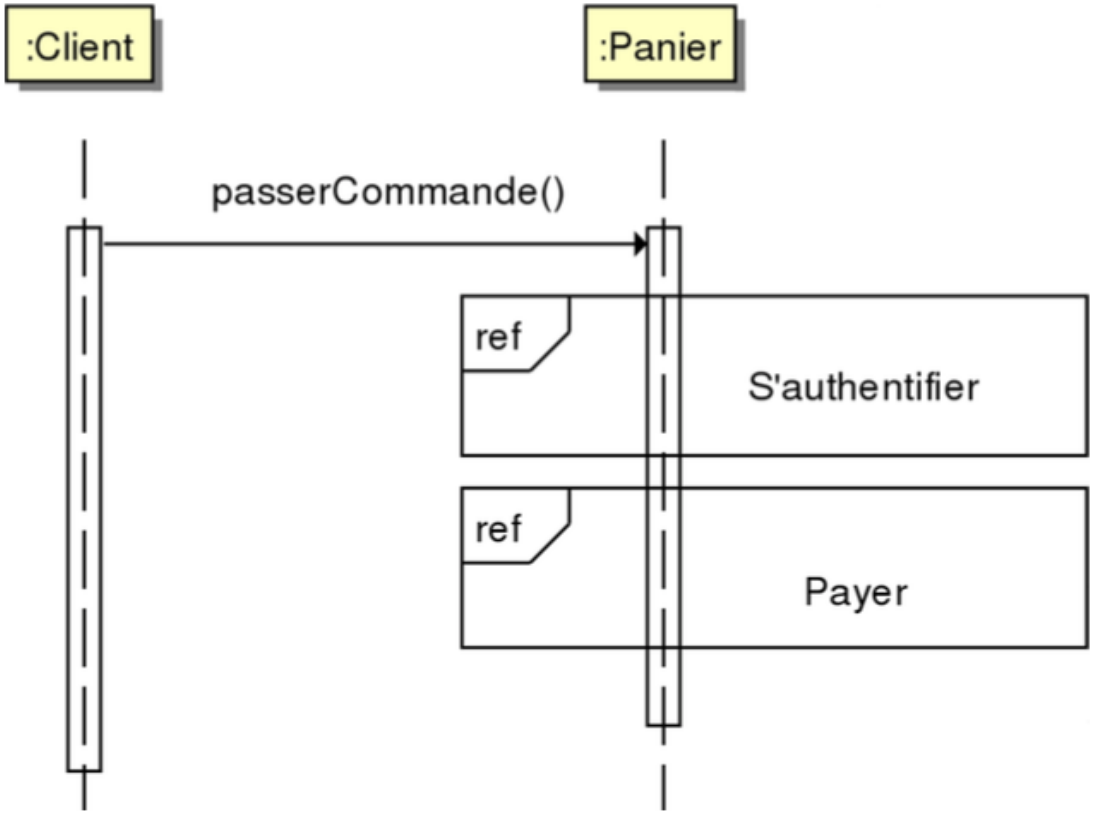
Opérateur parallèle

- L'opérateur "par" permet d'envoyer des messages en parallèle
- Ce qui se passe de part et d'autre de la ligne pointillée est indépendant



Réutilisation d'une interaction

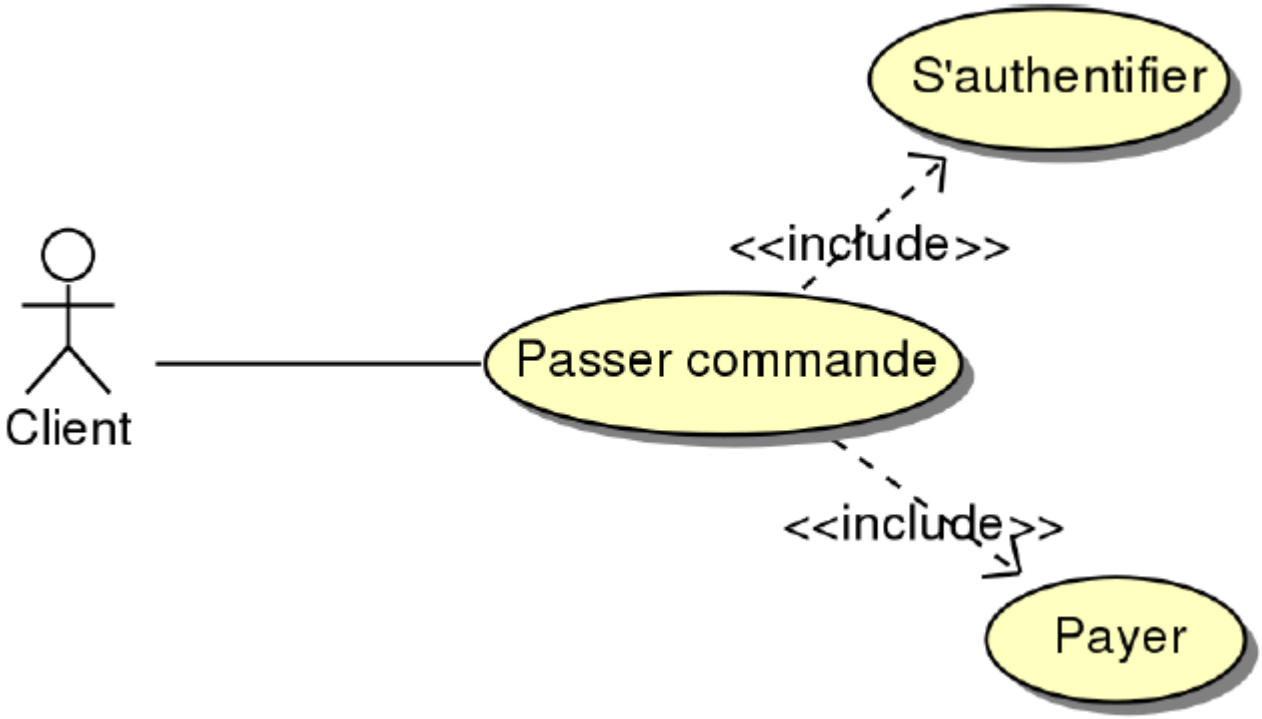
- Réutiliser une interaction consiste à placer un fragment portant la référence ref là où l'interaction est utile



➔ On spécifie le nom de l'interaction dans le fragment

Utilisation d'un DS pour modéliser un UC

- Chaque cas d'utilisation donne lieu à un diagramme de séquences
- Les inclusions et les extensions sont des cas typiques de la réutilisation par référencement



Partie 2

Modélisation Objet élémentaire avec UML

- Diagrammes de cas d'utilisation
- Diagrammes de classes
- Diagrammes d'objets
- Diagrammes de séquences
- **Diagrammes d'activité**

Objectifs

- **Diagramme d'activité est utilisé pour:**
  - **Modéliser un workflow dans un use case ou entre plusieurs use cases.**
  - **Spécifier une opération (décrire la logique d'une opération)**

Le diagramme d'activité est le plus approprié pour modéliser la dynamique d'une tâche, d'un use case lorsque le diagramme de classe n'est pas encore stabilisé

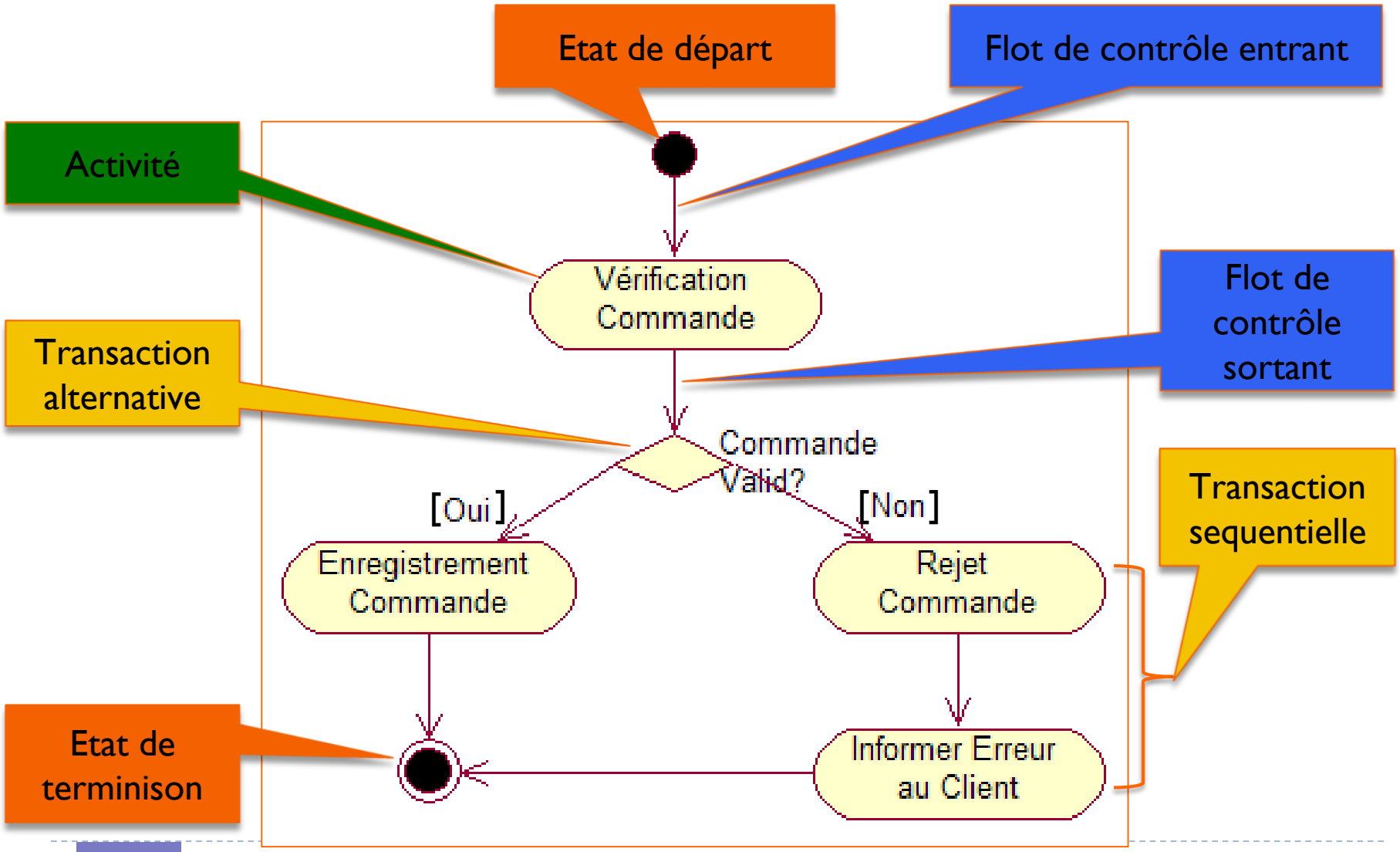
Notion du diagramme d'activité

Diagramme d'activité =

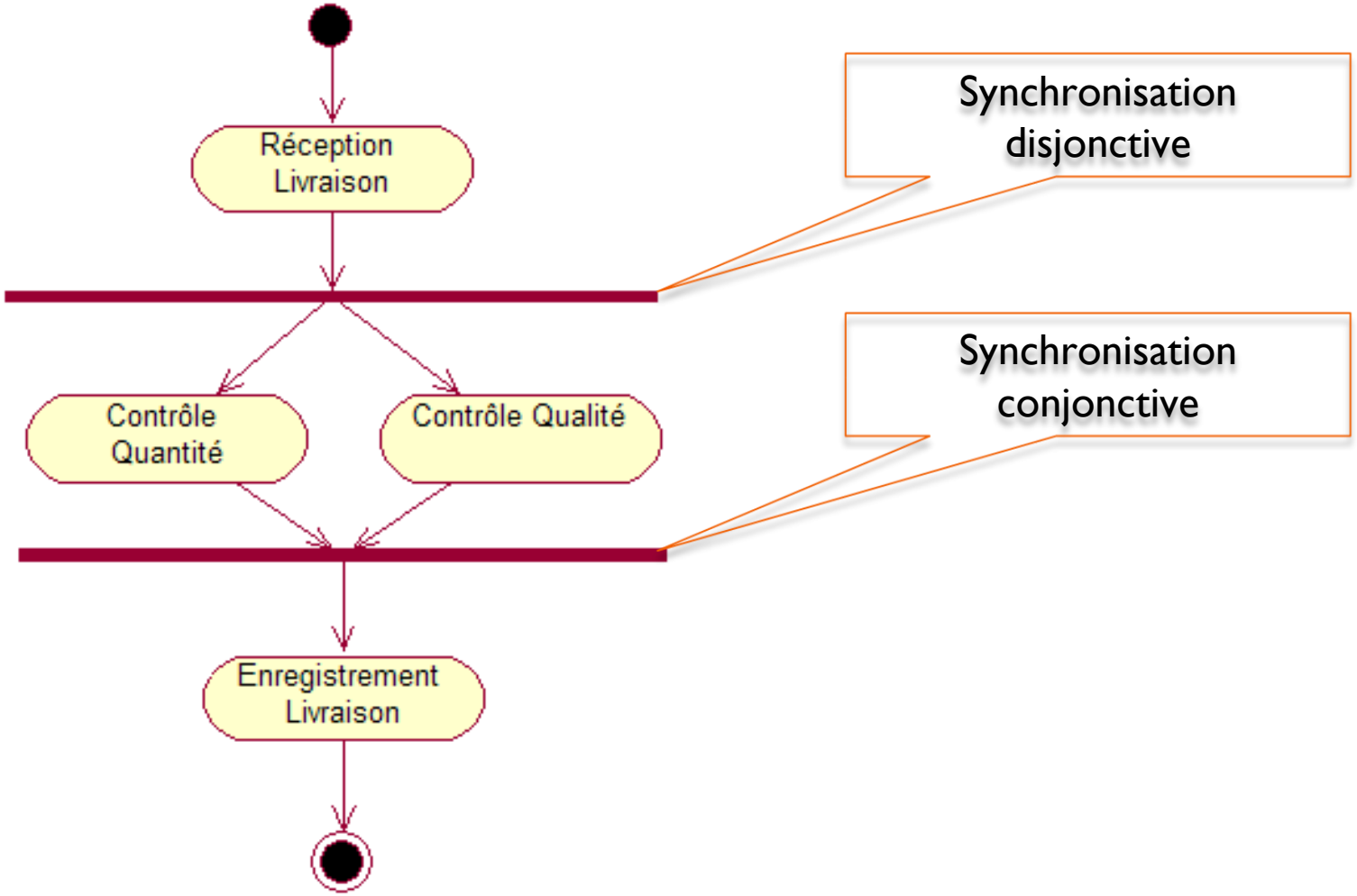
- **ensemble d'activités liés par:**
  - ***Transition (sequentielle)***
  - ***Transitions alternatives (conditionnelle)***
  - ***Synchronisation (disjonction et conjonctions d'activités)***
  - ***Itération***
- **+ *2* états: état de *départ* et état de *terminaison***

- ***Swimlanes:* represente le lieu, le responsable des activités.**

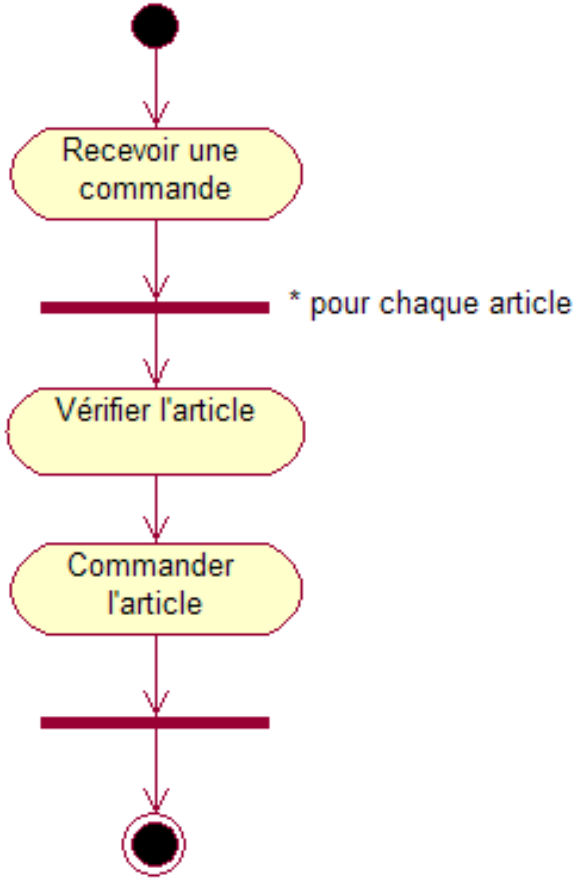
Notion du diagramme d'activité

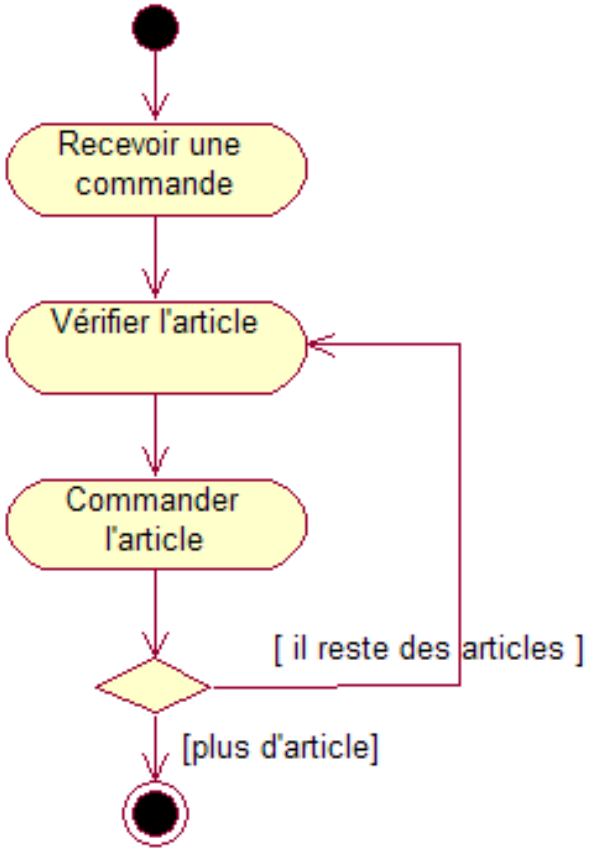


Notion du diagramme d'activité



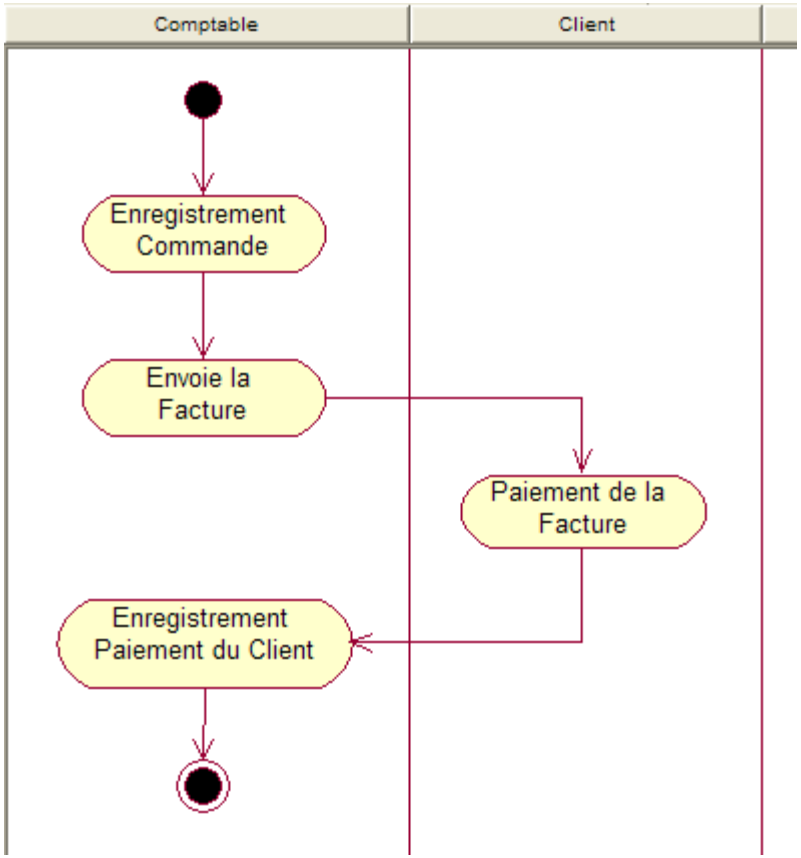
Notion du diagramme d'activité





Itération

Notion du diagramme d'activité



Swimlanes

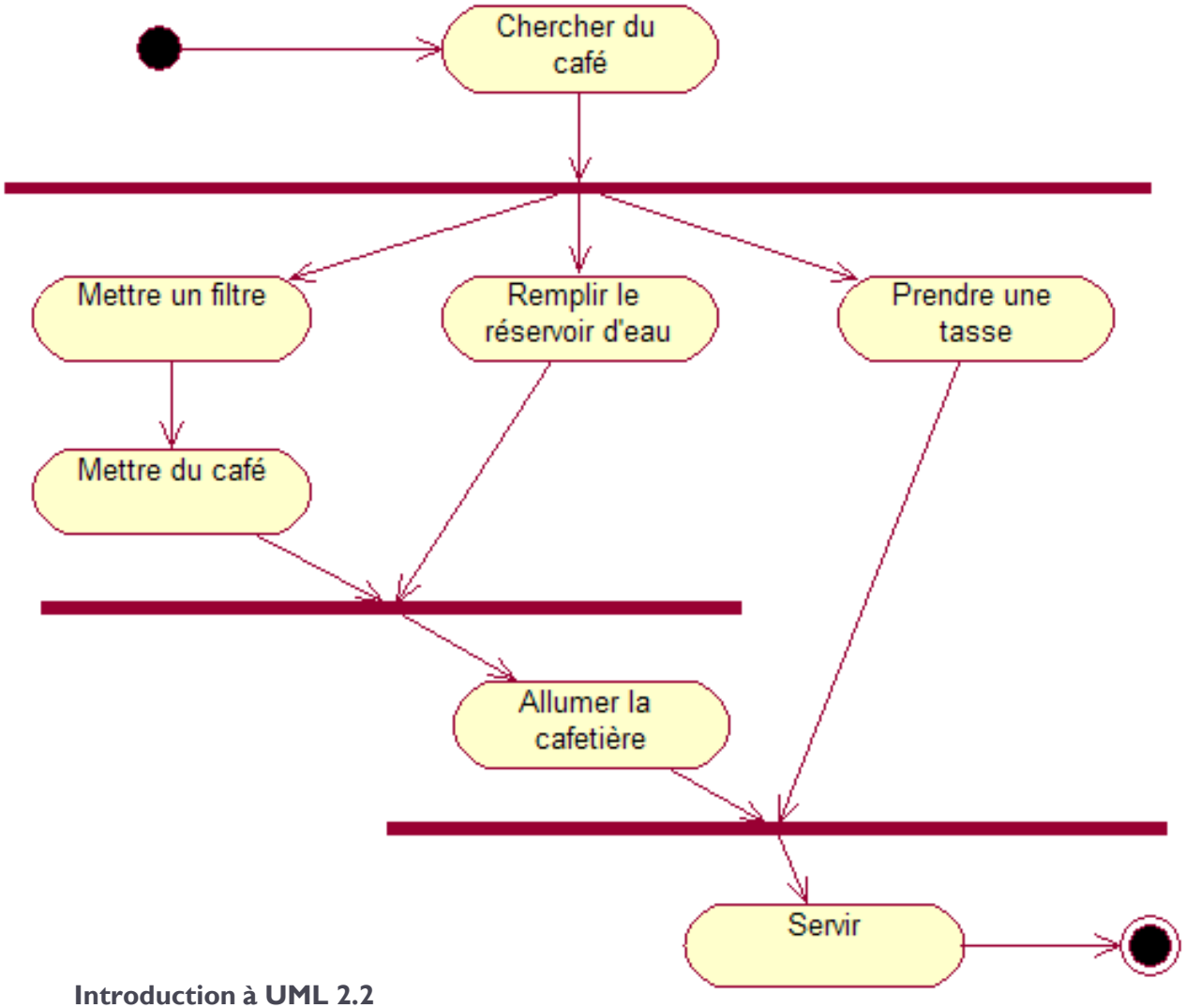
Construction un diagramme d'activité

1. **Identifiez la portée (« scope ») du diagramme d'activité**
   Commencez en identifiant ce que vous allez modéliser. Un seul use case? Une partie d'un use case ? Un « workflow » qui inclut plusieurs use cases ? Une méthode de classe ?
2. **Ajouter l'état de *départ* et de *terminaison***
3. **Ajouter les activités**
   Si vous modélisez un use case, introduisez une activité pour chaque use case principal. Si vous modélisez un « workflow », introduisez une activité pour chaque processus principal, souvent un use case. Enfin, si vous modélisez une méthode, il est souvent nécessaire d'avoir une activité pour chaque grand étape de la méthode.
4. **Ajouter des transitions (séquentielles), des transitions alternatives (conditionelles), des synchronisations entre des activités, des itérations.**
5. **Identifier des swimlanes et répartir des activités identifiées dans ces swimlanes.**

Exercice 1: Cafetière

- **Construire un diagramme d'activité représentant l'utilisation d'une cafetière électrique:**
  - **premier état: chercher du café**
  - **dernier état: Servir du café**

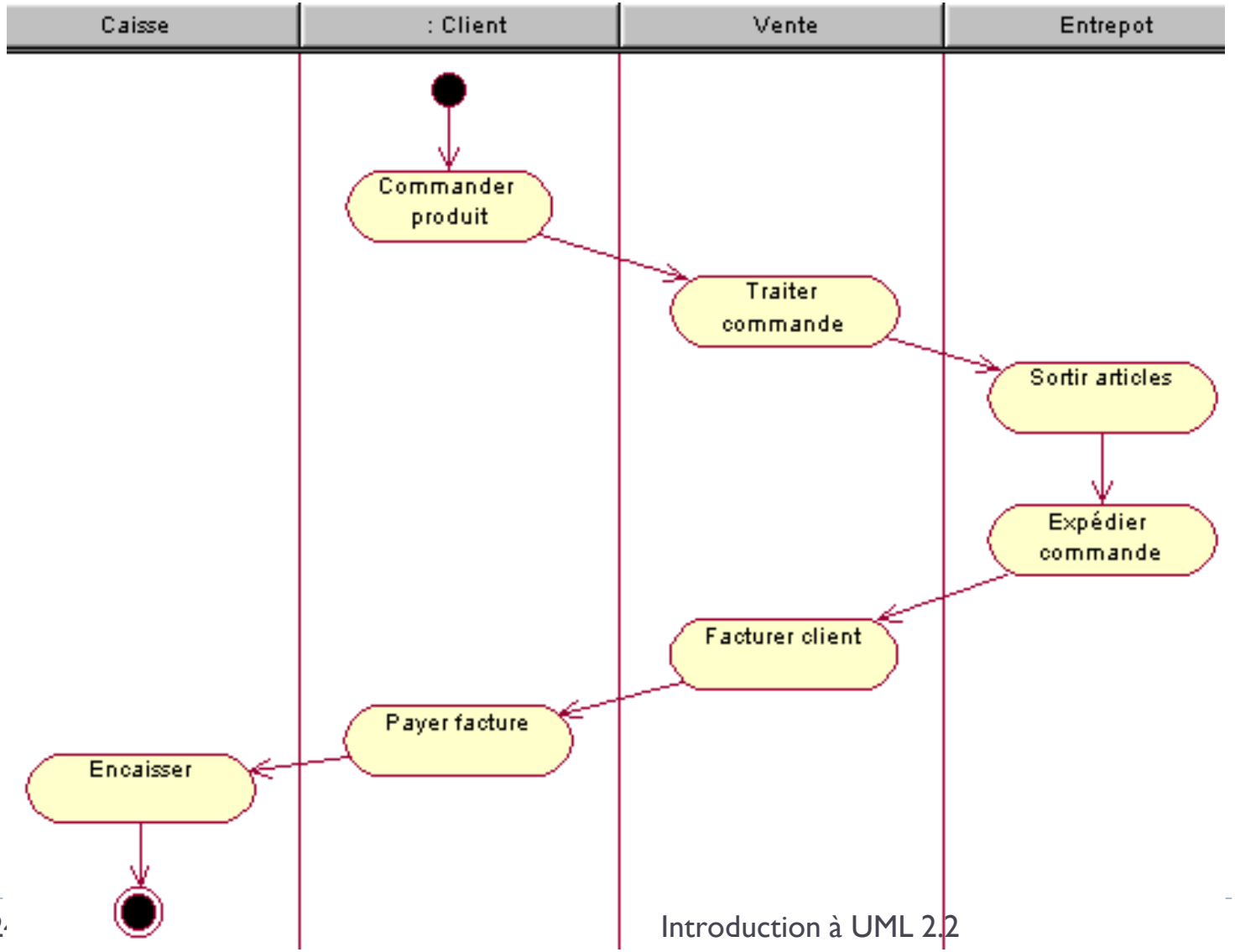
Cafetière: Solution possible



Exercice 2: Commander un produit

- **Construire un diagramme d'activité pour modéliser le processus de commander d'un produit. Le processus concerne les acteurs suivants:**
  - **Client: qui commande un produit et qui paie la facture**
  - **Caisse: qui encaisse l'argent du client**
  - **Vente: qui s'occupe de traiter et de facturer la commande du client**
  - **Entrepôt: qui est responsable de sortir les articles et d'expédier la commande.**

Commander un Produit: Solution possible



MonAuto : Use Case

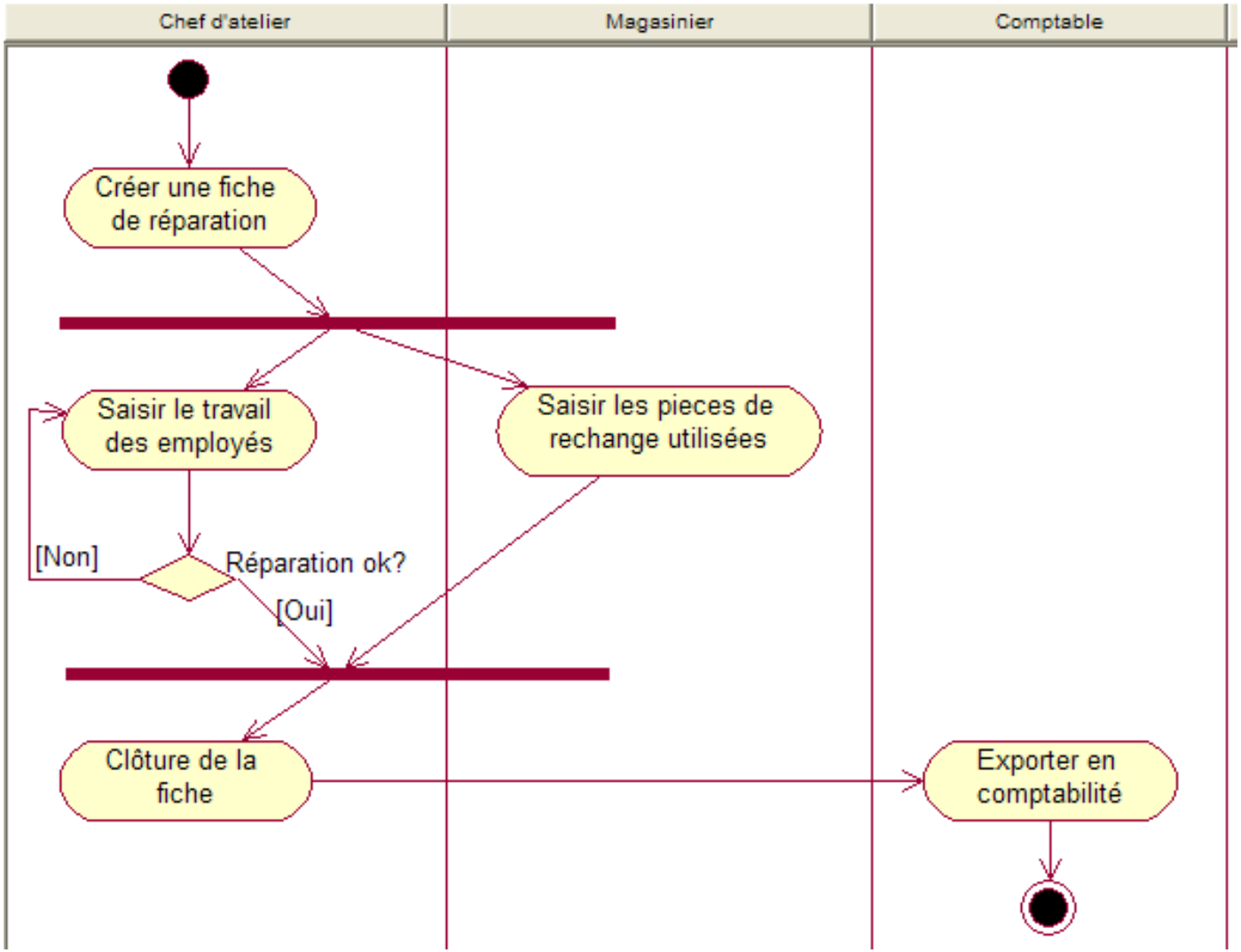
Le logiciel de gestion des réparations est destiné en priorité au chef d'atelier, il devra lui permettre de saisir les fiches de réparations et le travail effectué par les divers employés de l'atelier.
Pour effectuer leur travail, les mécaniciens et autres employés de l'atelier vont chercher des pièces de rechange au magasin. Lorsque le logiciel sera installé, les magasiniers ne fourniront des pièces que pour les véhicules pour lesquels une fiche de réparation est ouverte; ils saisiront directement les pièces fournies depuis un terminal installé au magasin.

Lorsqu'une réparation est terminée, le chef d'atelier va essayer la voiture. Si tout est en ordre, il met la voiture sur le parc clientèle et bouclera la fiche de réparation informatisée. Les fiches de réparations bouclées par le chef d'atelier devront pouvoir être importées par le comptable dans le logiciel comptable.

Exercice 3. Créer un diagramme d'activité pour tout le traitement d'une réparation.

Exercice 4. Créer un diagramme d'activité pour le use case « Créer une fiche de réparation »

MonAuto: Solution possible



MonAuto : Use Case

Exercice 2. Créer un diagramme d'activité pour le use case « Créer une fiche de réparation »

Pour créer une fiche de réparation, le chef d'atelier saisit les critères de recherche de voitures dans le système. Le logiciel de gestion des réparation lui donne la liste des voitures correspondant aux critères entrés. Si la voiture existe, le chef d'atelier va sélectionner la voiture. Le logiciel va, ensuite, fournir les informations sur le véhicule. Si la voiture est sous garantie, le chef devra saisir la date de demande de réparation. Si la voiture n'existe pas, le chef va saisir les informations concernant ce nouveau véhicule. Dans tous les cas, le chef d'atelier devra saisir la date de réception et de restitution. Si le dommage de la voiture est payé par l'assurance, le logiciel va fournir une liste d'assurances au chef d'atelier. Ce dernier sélectionnera l'assurance adéquate. Enfin, le logiciel enregistre la fiche de réparation.

MonAuto : Solution possible