PRESENTATION D'UML

- **Un peu d'Histoire...**
- **Les méthodes objet et la genèse d'UML**
- **Avantages et inconvénients d'UML**

MODELISER AVEC UML

- **Qu'est-ce-qu'un modèle ?**
- **Comment modéliser avec UML ?**
- **Modéliser les vues statiques d'un système**
    - la conceptualisation et les cas d'utilisation
    - structurer ses modèles (paquetages, collaboration)
    - les objets, le diagramme d'objets et les classes
    - le diagramme de classes
    - OCL
    - les stéréotypes
    - le diagramme de composants
    - le diagramme de déploiement
- **Modéliser les vues dynamiques d'un système**
    - le diagramme de collaboration
    - synchronisation des messages
    - les objets actifs
    - le diagramme de séquence
    - le diagramme d'états-transitions
    - le diagramme d'activités
- **Synthèse et conclusion**

Les vues statiques d'UML

LES CAS D'UTILISATION

La conceptualisation : rappel

- Le but de la conceptualisation est de comprendre et structurer les besoins du client.
- Il ne faut pas chercher l'exhaustivité, mais clarifier, filtrer et organiser les besoins !
- Une fois identifiés et structurés, ces besoins :
  - définissent le contour du système à modéliser (ils précisent le but à atteindre),
  - permettent d'identifier les fonctionnalités principales (critiques) du système.
- Le modèle conceptuel doit permettre une meilleure compréhension du système.
- Le modèle conceptuel doit servir d'interface entre tous les acteurs du projet.
- Les besoins des clients sont des éléments de traçabilité dans un processus intégrant UML.

Le modèle conceptuel joue un rôle central, il est capital de bien le définir !

Cas d'utilisation (use cases)

- Il s'agit de la solution UML pour représenter le modèle conceptuel.
- Les use cases permettent de structurer les besoins des utilisateurs et les objectifs correspondants d'un système.
- Ils centrent l'expression des exigences du système sur ses utilisateurs : ils partent du principe que les objectifs du système sont tous motivés.
- Ils se limitent aux préoccupations "réelles" des utilisateurs ; ils ne présentent pas de solutions d'implémentation et ne forment pas un inventaire fonctionnel du système.
- Ils identifient les utilisateurs du système (acteurs) et leur interaction avec le système.
- Ils permettent de classer les acteurs et structurer les objectifs du système.
- Ils servent de base à la traçabilité des exigences d'un système dans un processus de développement intégrant UML.

Il était une fois...

Le modèle conceptuel est le type de diagramme UML qui possède la notation la plus simple ; mais paradoxalement c'est aussi celui qui est le plus mal compris !

Au début des années 90, Ivar Jacobson (inventeur de OOSE, une des méthodes fondatrices d'UML) a été nommé chef d'un énorme projet informatique chez Ericsson. Le hic, c'est que ce projet était rapidement devenu ingérable, les ingénieurs d'Ericsson avaient accouché d'un monstre. Personne ne savait vraiment quelles étaient les fonctionnalités du produit, ni comment elles étaient assurées, ni comment les faire évoluer...

Classique lorsque les commerciaux promettent monts et merveilles à tous les clients qu'ils démarchent, sans se soucier des contraintes techniques, que les clients ne savent pas exprimer leurs besoins et que les ingénieurs n'ont pas les ressources pour développer le mouton à cinq pattes qui résulte de ce chaos.

Pour éviter de foncer droit dans un mur et mener à bien ce projet critique pour Ericsson, Jacobson a eu une idée. Plutôt que de continuer à construire une tour de Babel, pourquoi ne pas remettre à plat les objectifs réels du projet ? En d'autres termes : quels sont les besoins réels des clients, ceux qui conditionneront la réussite du projet ? Ces besoins critiques, une fois identifiés et structurés, permettront enfin de cerner "ce qui est important pour la réussite du projet".

Le bénéfice de cette démarche simplificatrice est double. D'une part, tous les acteurs du projet ont une meilleure compréhension du système à développer, d'autre part, les besoins des utilisateurs, une fois clarifiés, serviront de fil rouge, tout au long du cycle de développement.
A chaque itération de la phase d'analyse, on clarifie, affine et valide les besoins des utilisateurs ; à chaque itération de la phase de conception et de réalisation, on veille à la prise en compte des besoins des utilisateurs et à chaque itération de la phase de test, on vérifie que les besoins des utilisateurs sont satisfaits.

Simple mais génial. Pour la petite histoire, sachez que grâce à cette démarche initiée par Jacobson, Ericsson a réussi à mener à bien son projet et a gagné une notoriété internationale dans le marché de la commutation.

Morale de cette histoire :

La détermination et la compréhension des besoins sont souvent difficiles car les intervenants sont noyés sous de trop grandes quantités d'informations. Or, comment mener à bien un projet si l'on ne sait pas où l'on va ?

Conclusion : il faut clarifier et organiser les besoins des clients (les modéliser).

Jacobson identifie les caractéristiques suivantes pour les modèles :

- Un modèle est une simplification de la réalité.
- Il permet de mieux comprendre le système qu'on doit développer.
- Les meilleurs modèles sont proches de la réalité.

Les use cases, permettent de modéliser les besoins des clients d'un système et doivent aussi posséder ces caractéristiques.
Ils ne doivent pas chercher l'exhaustivité, mais clarifier, filtrer et organiser les besoins !

Une fois identifiés et structurés, ces besoins :

- définissent le contour du système à modéliser (ils précisent le but à atteindre),

- permettent d'identifier les fonctionnalités principales (critiques) du système.

Les use cases ne doivent donc en aucun cas décrire des solutions d'implémentation. Leur but est justement d'éviter de tomber dans la dérive d'une approche fonctionnelle, où l'on liste une litanie de fonctions que le système doit réaliser.

Bien entendu, rien n'interdit de gérer à l'aide d'outils (Doors, Requisite Pro, etc...) les exigences systèmes à un niveau plus fin et d'en assurer la traçabilité, bien au contraire.

Mais un modèle conceptuel qui identifie les besoins avec un plus grand niveau d'abstraction reste indispensable. Avec des systèmes complexes, filtrer l'information, la simplifier et mieux l'organiser, c'est rendre l'information exploitable. Produisez de l'information éphémère, complexe et confuse, vous obtiendrez un joyeux "désordre" (pour rester poli).

Dernière remarque :

Utilisez les use cases tels qu'ils ont été pensé par leurs créateurs ! UML est issu du terrain. Si vous utilisez les use cases sans avoir en tête la démarche sous-jacente, vous n'en tirerez aucun bénéfice.

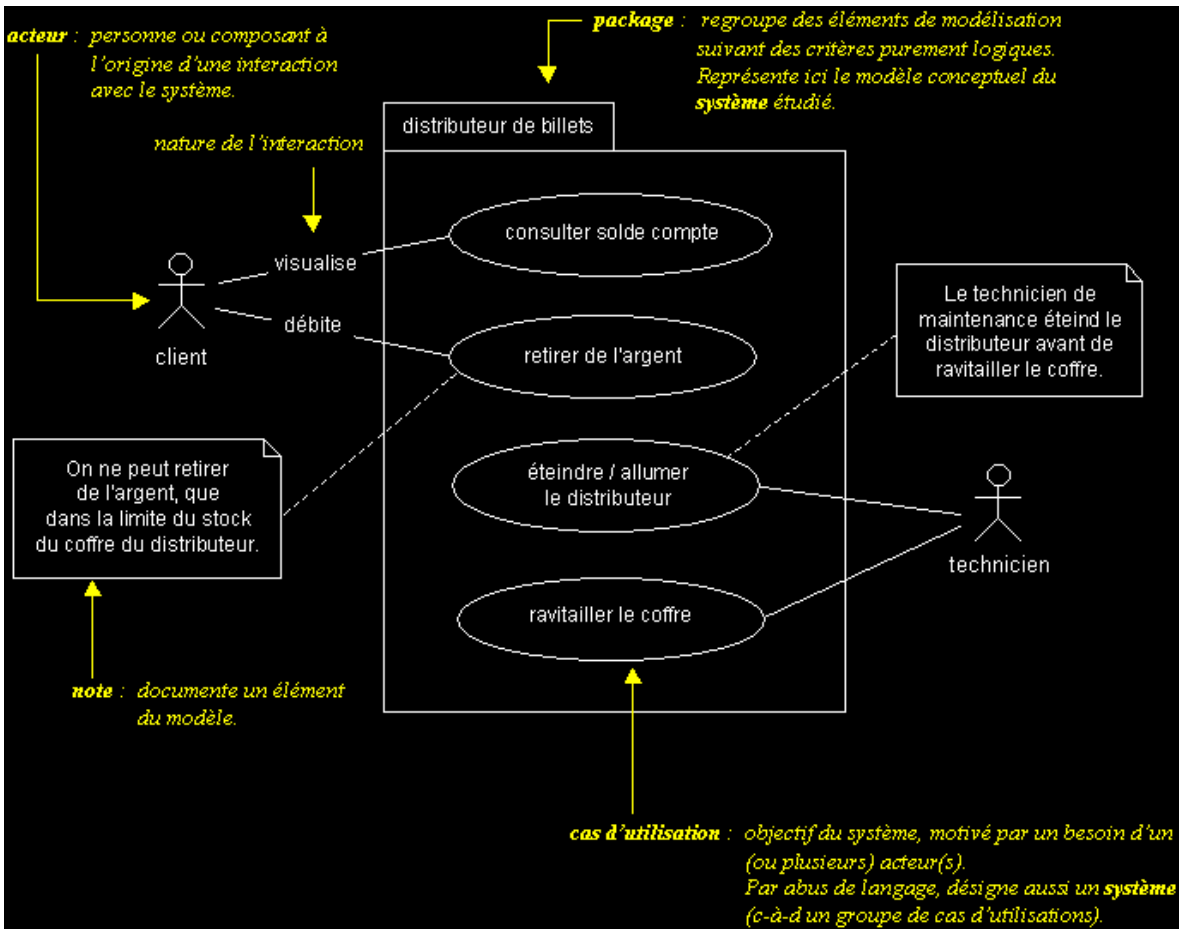
q **Eléments de base des cas d'utilisation**

- **Acteur** : entité externe qui agit sur le système (opérateur, autre système...).
  - L'acteur peut consulter ou modifier l'état du système.
  - En réponse à l'action d'un acteur, le système fournit un service qui correspond à son besoin.
  - Les acteurs peuvent être classés (hiérarchisés).

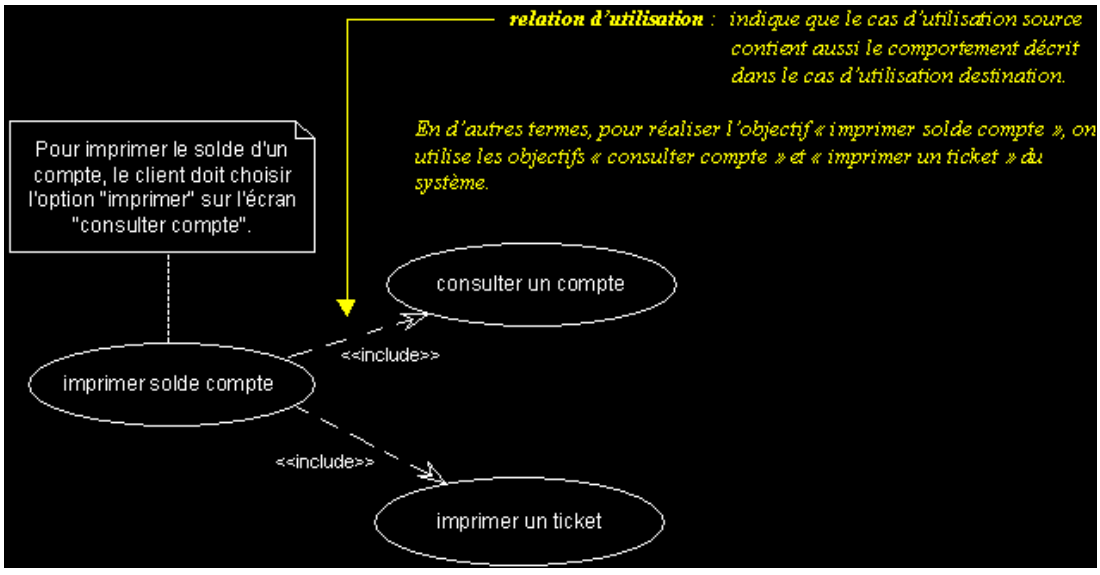
- **Use case** : ensemble d'actions réalisées par le système, en réponse à une action d'un acteur.
  - Les uses cases peuvent être structurés.
  - Les uses cases peuvent être organisés en paquetages (packages).
  - L'ensemble des use cases décrit les objectifs (le but) du système.

q **Exemples**

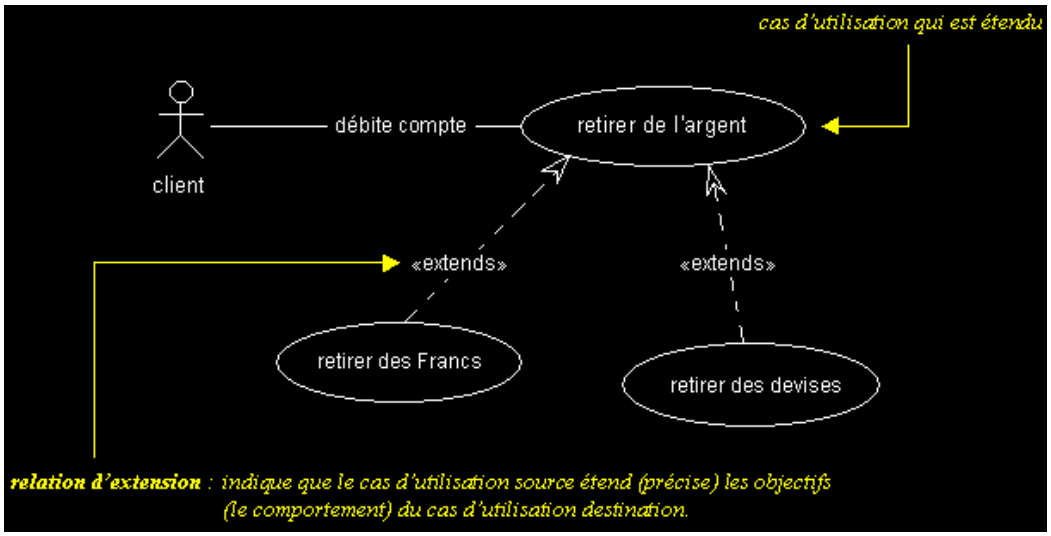
Cas d'utilisation standard :



Relation d'utilisation :



Relation d'extension :

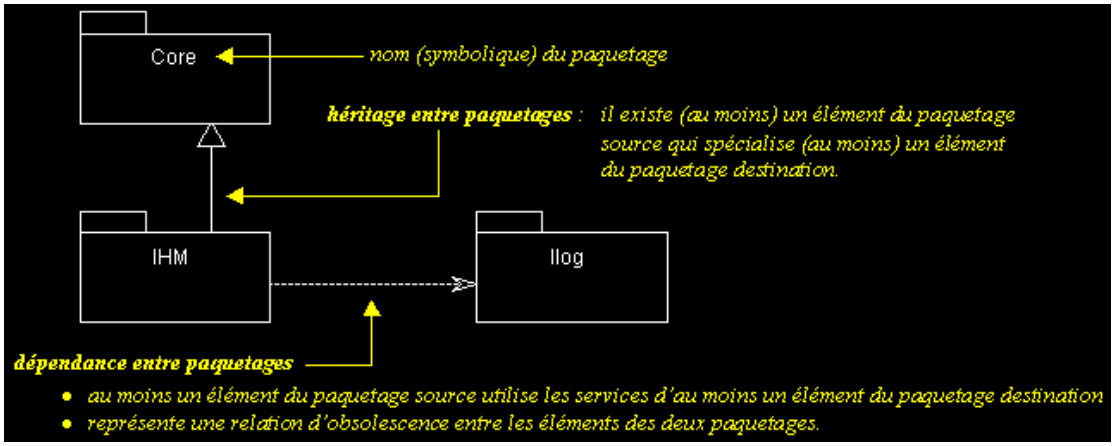

cas d'utilisation qui est étendu
client
débite compte
retirer de l'argent
«extends»
«extends»
retirer des Francs
retirer des devises
relation d'extension : indique que le cas d'utilisation source étend (précise) les objectifs (le comportement) du cas d'utilisation destination.

LES PAQUETAGES

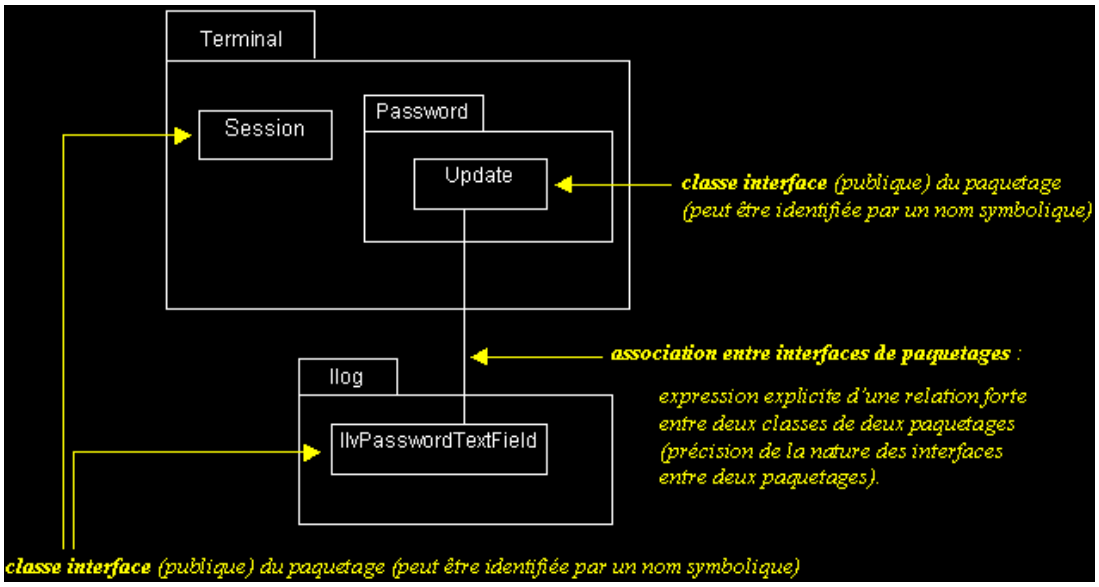
Paquetages (packages)

- Les paquetages sont des éléments d'organisation des modèles.
- Ils regroupent des éléments de modélisation, selon des critères purement logiques.
- Ils permettent d'encapsuler des éléments de modélisation (ils possèdent une interface).
- Ils permettent de structurer un système en catégories (vue logique) et sous-systèmes (vue des composants).
- Ils servent de "briques" de base dans la construction d'une architecture.
- Ils représentent le bon niveau de granularité pour la réutilisation.
- Les paquetages sont aussi des espaces de noms.

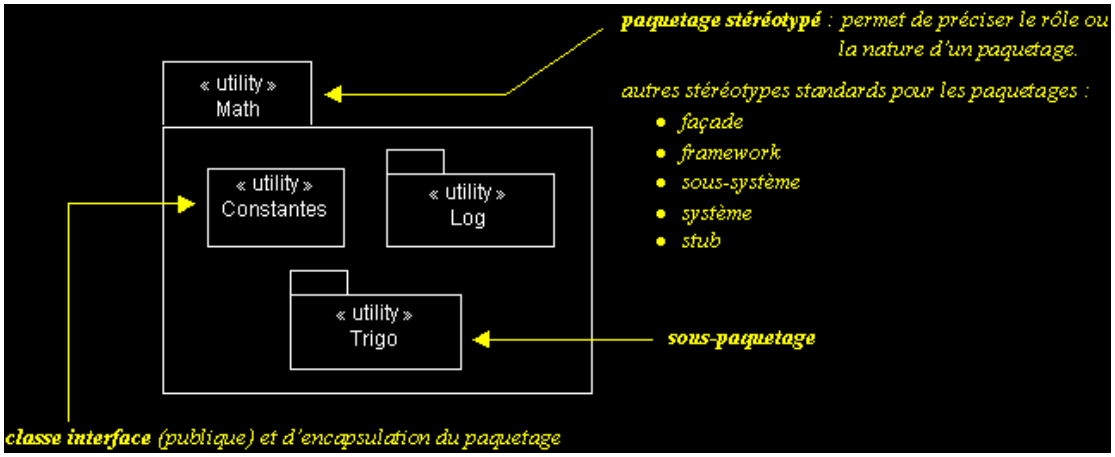
Paquetages : relations entre paquetages

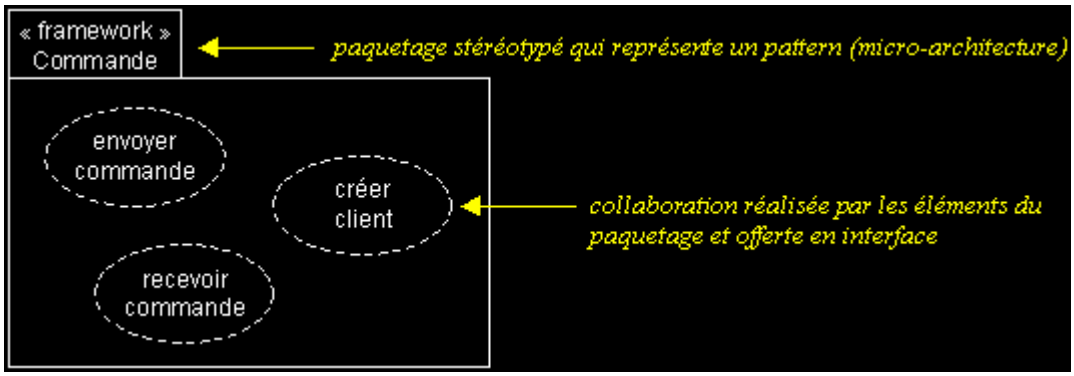


Paquetages : interfaces



q **Paquetages : <u>stéréotypes</u>**





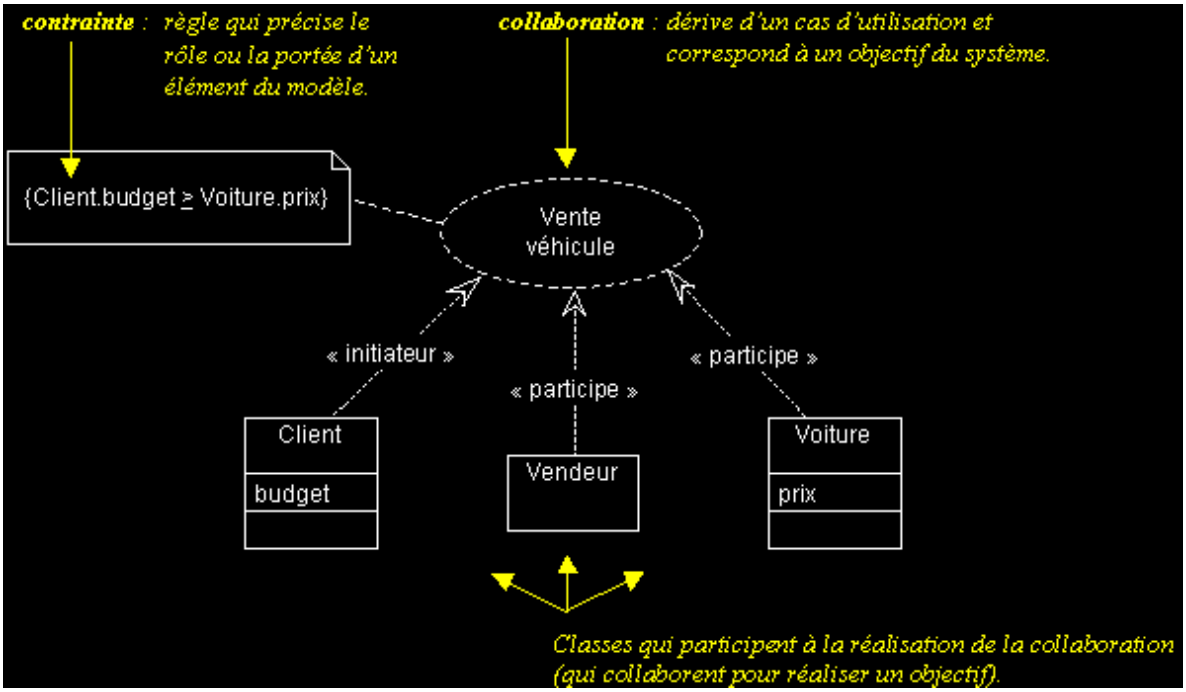
LA COLLABORATION

Symbole de modélisation "collaboration"

- Les collaborations sont des interactions entre objets, dont le but est de réaliser un objectif du système (c'est-à-dire aussi de répondre à un besoin d'un utilisateur).

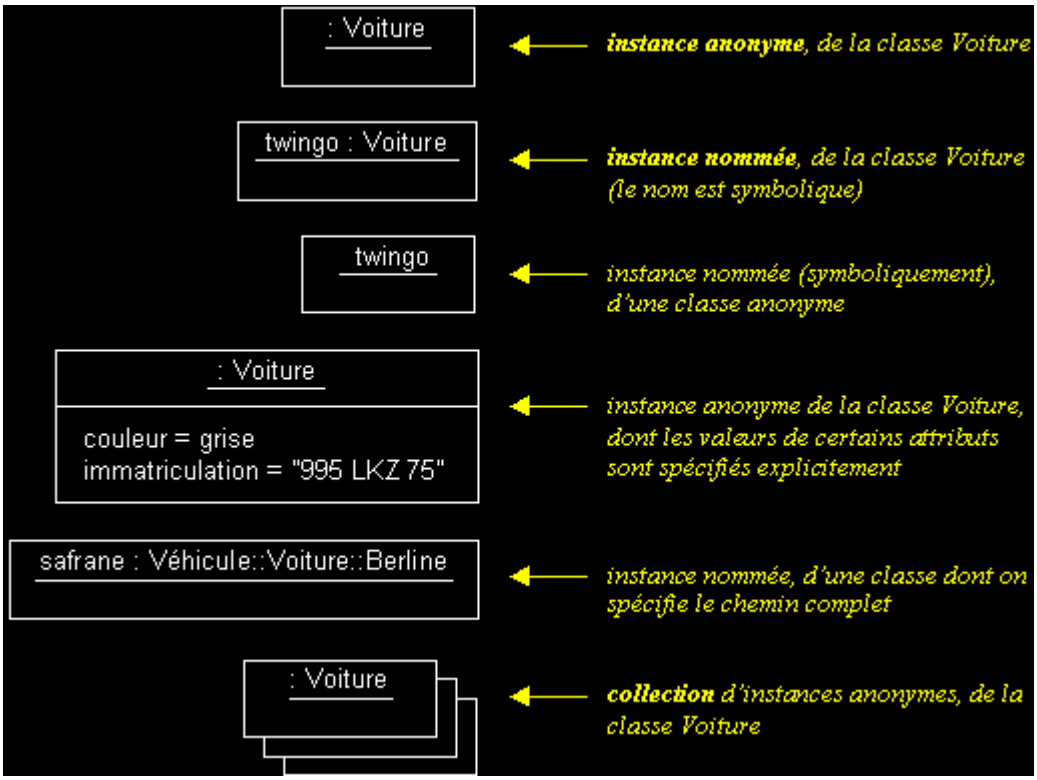
- L'élément de modélisation UML "collaboration", représente les classes qui participent à la réalisation d'un cas d'utilisation.

  Attention : ne confondez pas l'élément de modélisation "collaboration" avec le diagramme de collaboration, qui représente des interactions entre instances de classes.

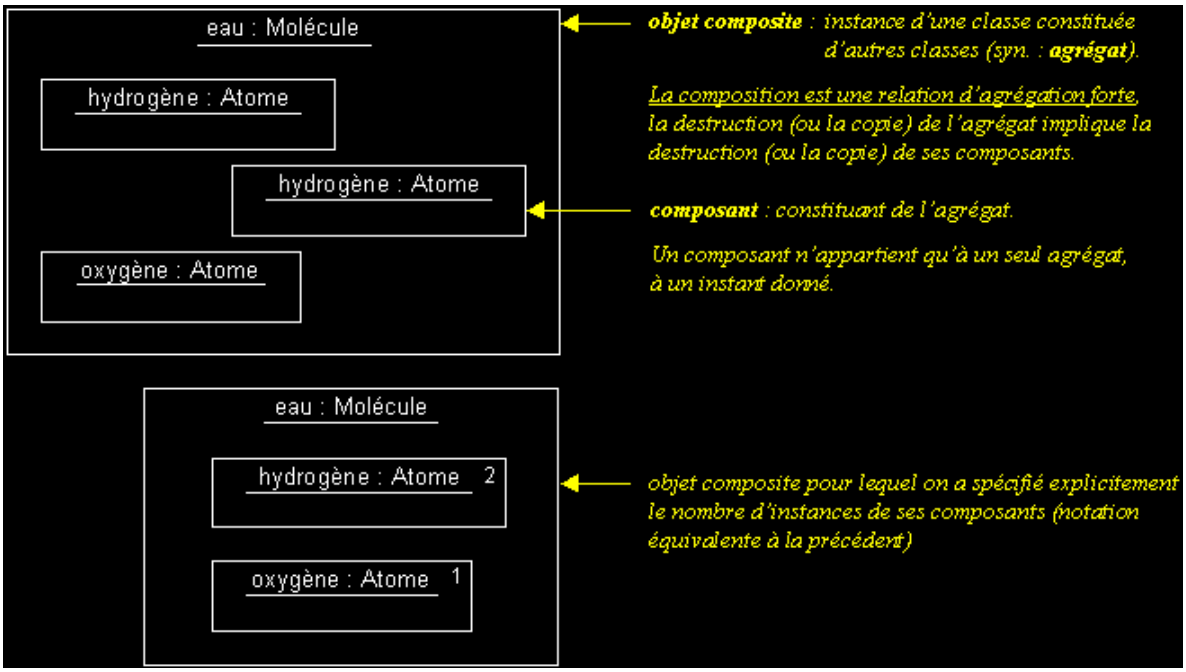


INSTANCES ET DIAGRAMME D'OBJETS

Exemples d'instances



Objets composites



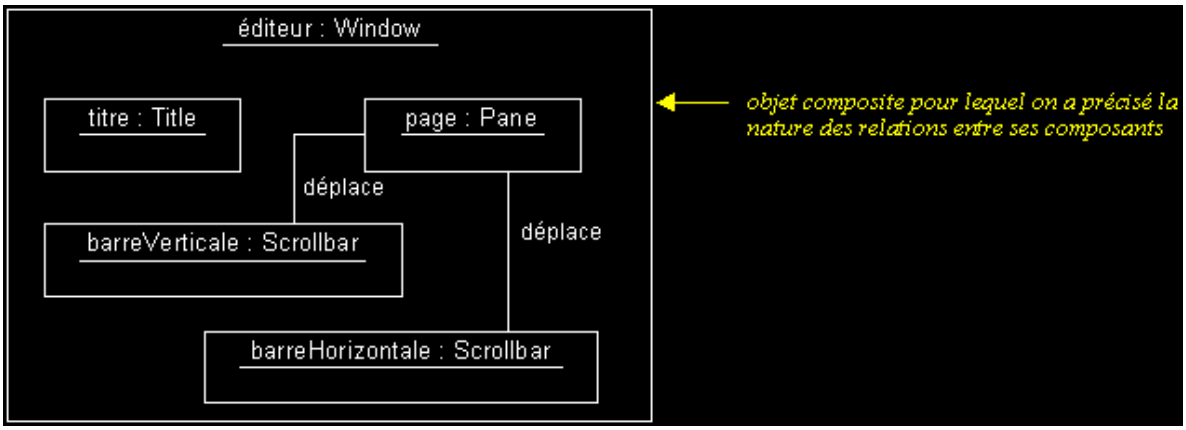
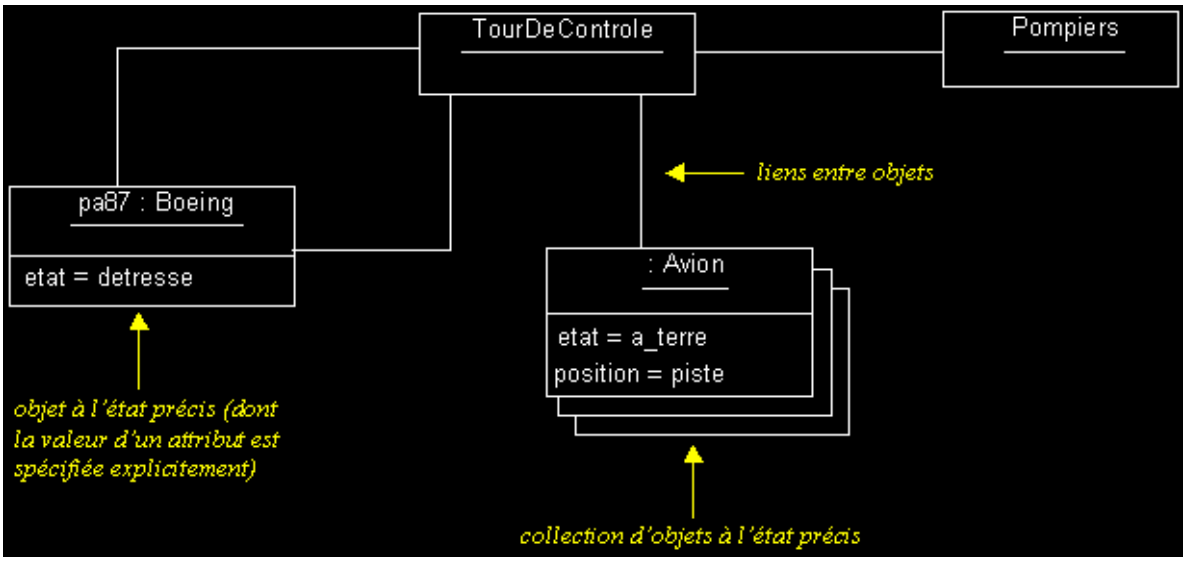


Diagramme d'objets

- Ce type de diagramme UML montre des objets (instances de classes dans un état particulier) et des liens (relations sémantiques) entre ces objets.
- Les diagrammes d'objets s'utilisent pour montrer un contexte (avant ou après une interaction entre objets par exemple).
- Ce type de diagramme sert essentiellement en phase exploratoire, car il possède un très haut niveau d'abstraction.

Exemple :


TourDeControle
Pompiers
liens entre objets
pa87 : Boeing
etat = detresse
: Avion
etat = a_terre
position = piste
objet à l'état précis (dont la valeur d'un attribut est spécifiée explicitement)
collection d'objets à l'état précis

LES CLASSES

Classe : sémantique et notation

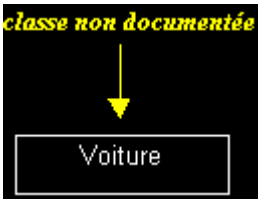
- Une classe est un type abstrait caractérisé par des propriétés (attributs et méthodes) communes à un ensemble d'objets et permettant de créer des objets ayant ces propriétés.

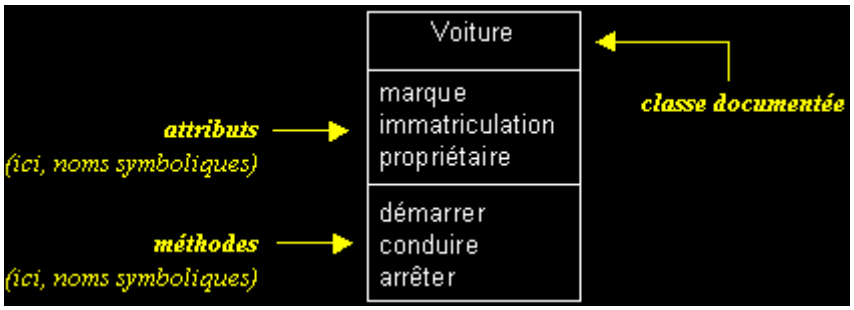
  Classe = attributs + méthodes + instanciation

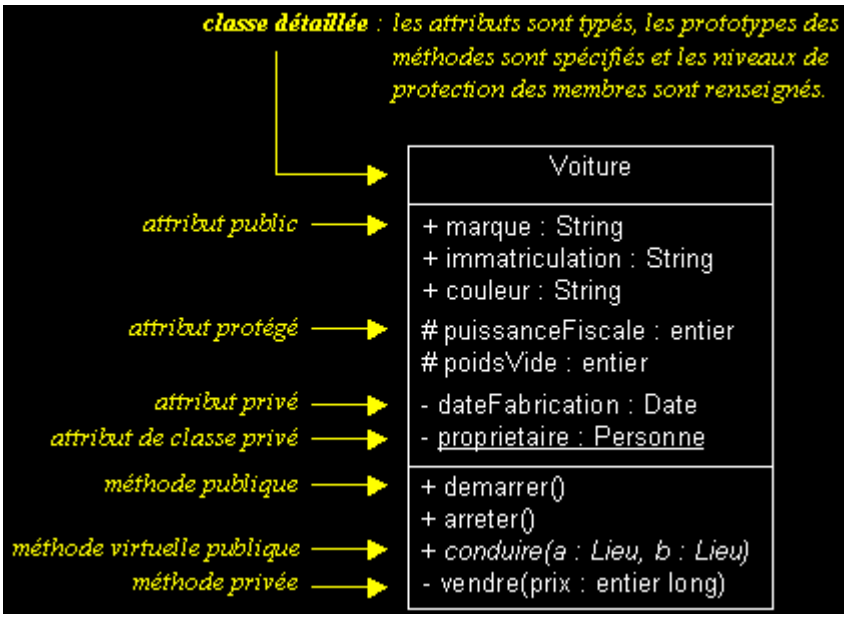
- Ne pas représenter les attributs ou les méthodes d'une classe sur un diagramme, n'indique pas que cette classe n'en contient pas.
  Il s'agit juste d'un filtre visuel, destiné à donner un certain niveau d'abstraction à son modèle.

  De même, ne pas spécifier les niveaux de protection des membres d'une classe ne veut pas dire qu'on ne représente que les membres publics. Là aussi, il s'agit d'un filtre visuel.

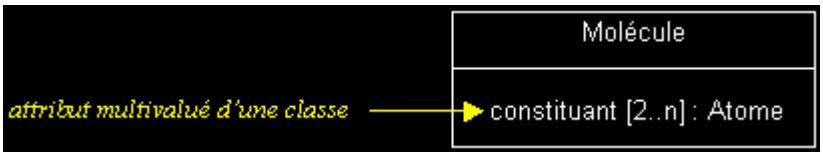
Documentation d'une classe (niveaux d'abstraction), exemples :

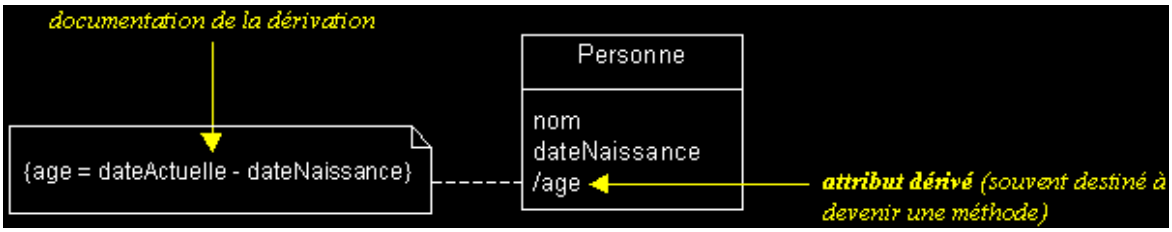




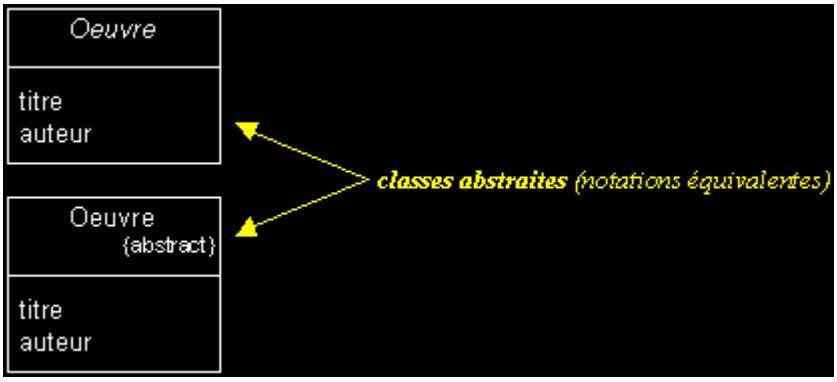


Attributs multivalués et dérivés, exemples :





Classe abstraite, exemple :



Template (classe paramétrable), exemple :

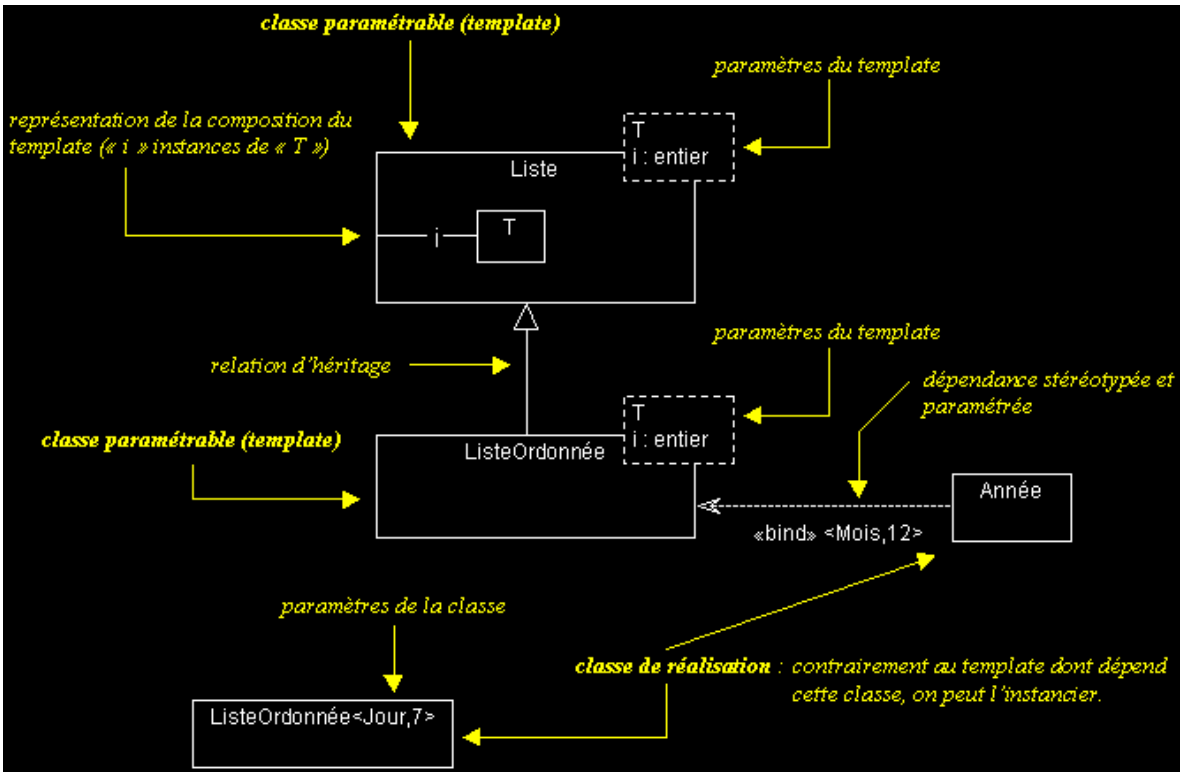


DIAGRAMME DE CLASSES

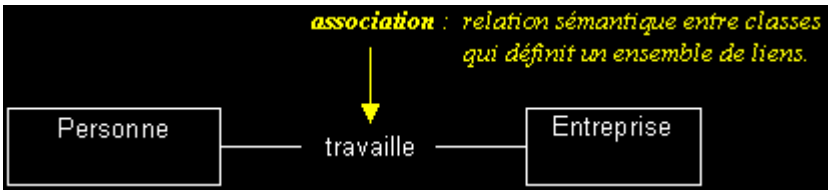
Diagramme de classes : sémantique

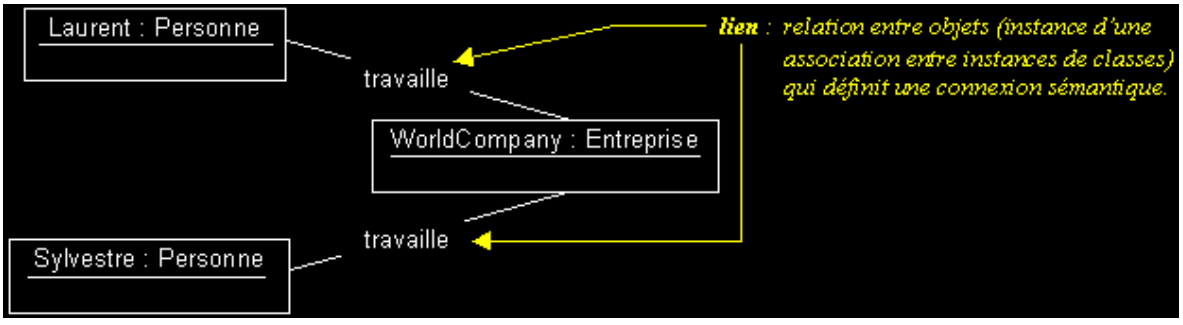
- Un diagramme de classes est une collection d'éléments de modélisation statiques (classes, paquetages...), qui montre la structure d'un modèle.
- Un diagramme de classes fait abstraction des aspects dynamiques et temporels.
- Pour un modèle complexe, plusieurs diagrammes de classes complémentaires doivent être construits.

  On peut par exemple se focaliser sur :
  - les classes qui participent à un cas d'utilisation (cf. collaboration),
  - les classes associées dans la réalisation d'un scénario précis,
  - les classes qui composent un paquetage,
  - la structure hiérarchique d'un ensemble de classes.
- Pour représenter un contexte précis, un diagramme de classes peut être instancié en diagrammes d'objets.

Associations entre classes

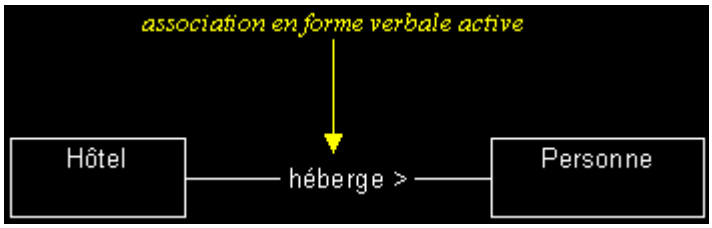
- Une association exprime une connexion sémantique bidirectionnelle entre deux classes.
- L'association est instanciable dans un diagramme d'objets ou de collaboration, sous forme de liens entre objets issus de classes associées.



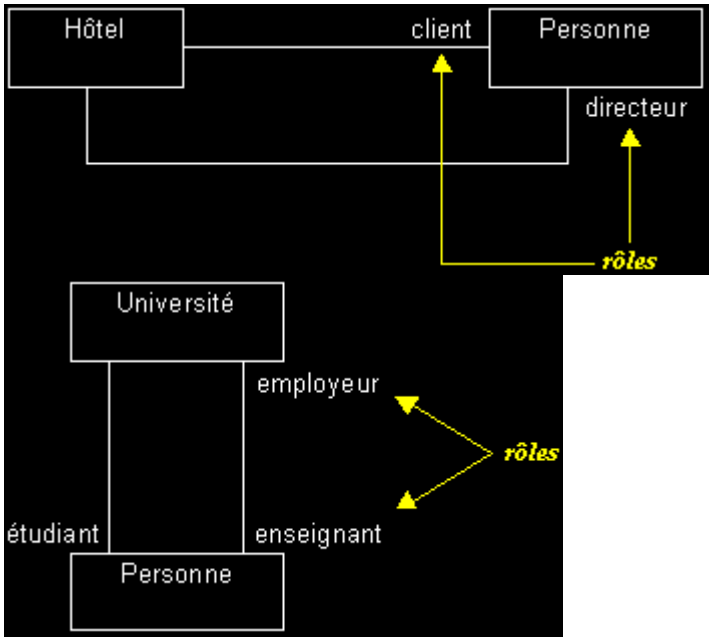


Documentation d'une association et types d'associations

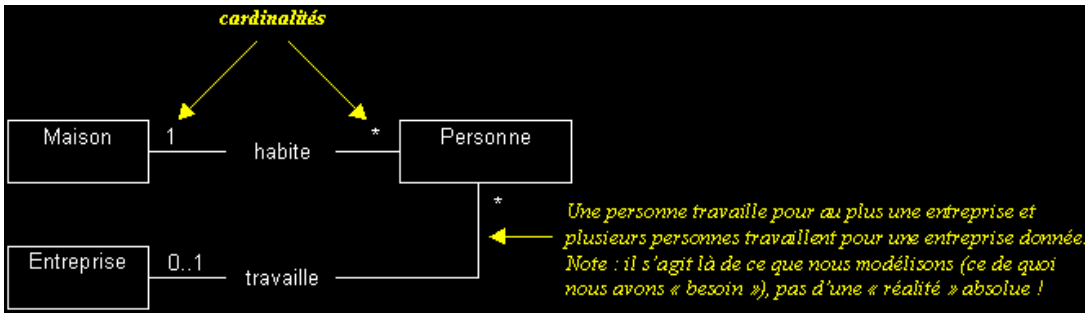
- **Association en forme verbale active** : précise le sens de lecture principal d'une association.
  Voir aussi : association à navigabilité restreinte.



- **Rôles** : spécifie la fonction d'une classe pour une association donnée (indispensable pour les associations réflexives).



- **Cardinalités** : précise le nombre d'instances qui participent à une relation.



Expression des cardinalités d'une relation en UML :

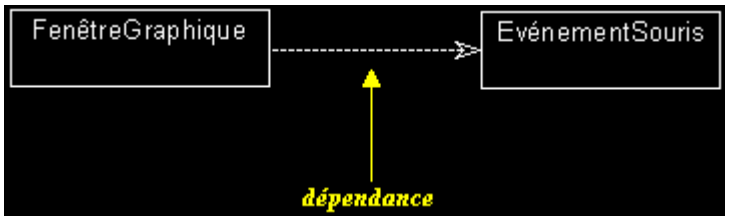
n : exactement "n" (n, entier naturel > 0)
exemples : "1", "7"

n..m : de "n" à "m" (entiers naturels ou variables, m ≥ n)
exemples : "0..1", "3..n", "1..31"

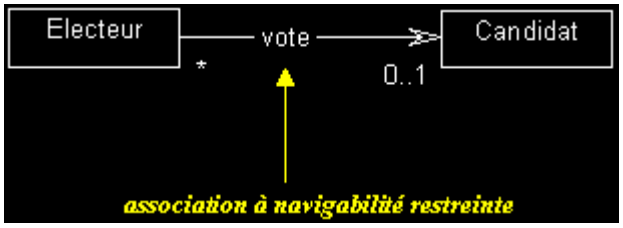
*** :** plusieurs (équivalent à "0..n" et "0..*")

n..* : "n" ou plus (n, entier naturel ou variable)
exemples : "0..*", "5..*"

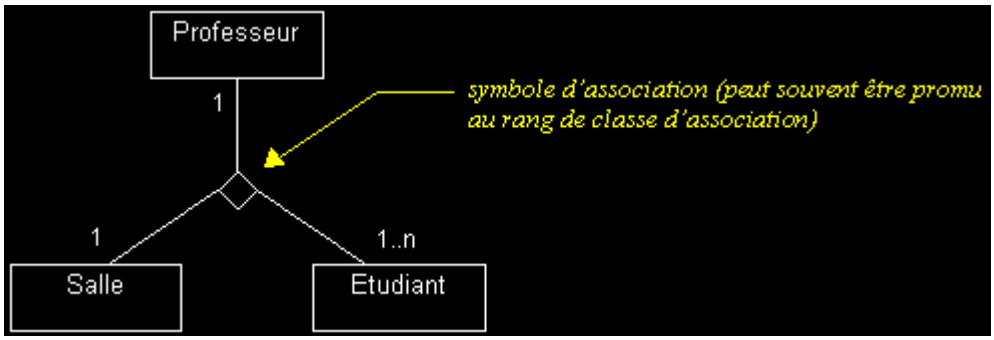
- **Relation de dépendance** : relation d'utilisation unidirectionnelle et d'obsolescence (une modification de l'élément dont on dépend, peut nécessiter une mise à jour de l'élément dépendant).



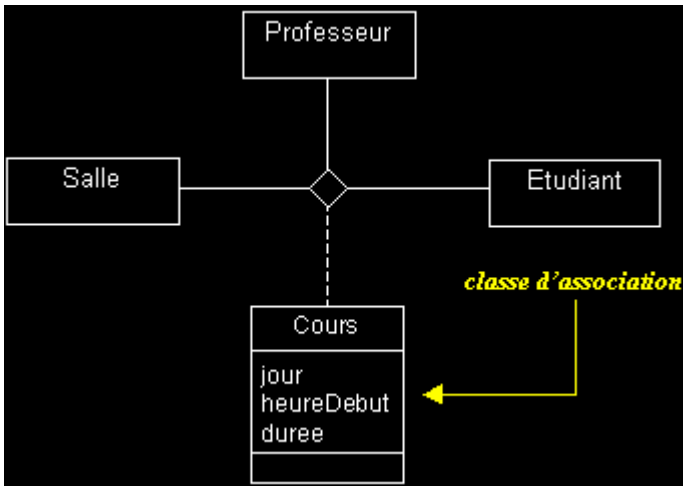
- **Association à navigabilité restreinte**
Par défaut, une association est navigable dans les deux sens. La réduction de la portée de l'association est souvent réalisée en phase d'implémentation, mais peut aussi être exprimée dans un modèle pour indiquer que les instances d'une classe ne "connaissent" pas les instances d'une autre.

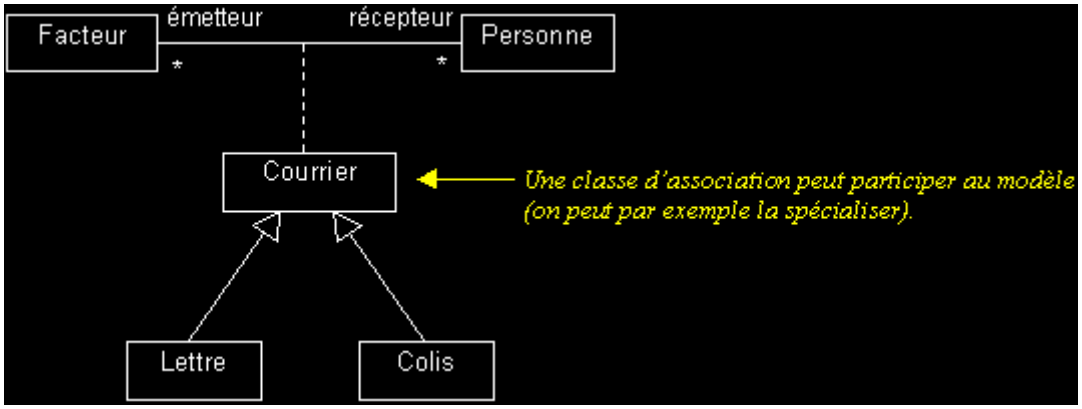


- **Association n-aire** : il s'agit d'une association qui relie plus de deux classes...
Note : de telles associations sont difficiles à déchiffrer et peuvent induire en erreur. Il vaut mieux limiter leur utilisation, en définissant de nouvelles catégories d'associations.



- **Classe d'association** : il s'agit d'une classe qui réalise la navigation entre les instances d'autres classes.





- **Qualification** : permet de sélectionner un sous-ensemble d'objets, parmi l'ensemble des objets qui participent à une association.
  La restriction de l'association est définie par une clé, qui permet de sélectionner les objets ciblés.

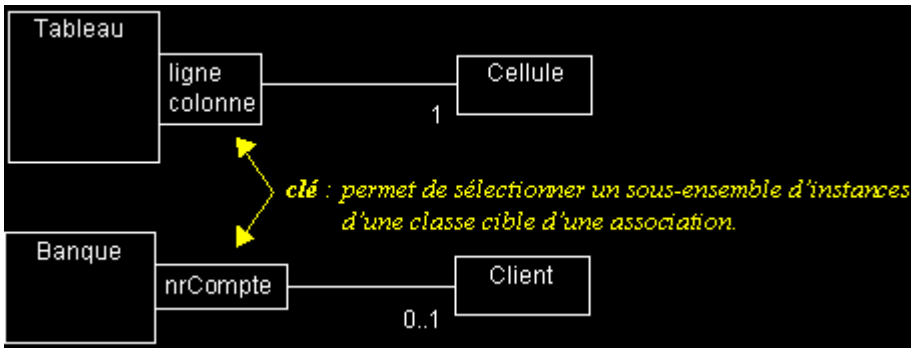


Tableau
ligne
colonne
Cellule
1
clé : permet de sélectionner un sous-ensemble d'instances
d'une classe cible d'une association.
Banque
nrCompte
Client
0..1

DIAGRAMME DE CLASSES (suite...)

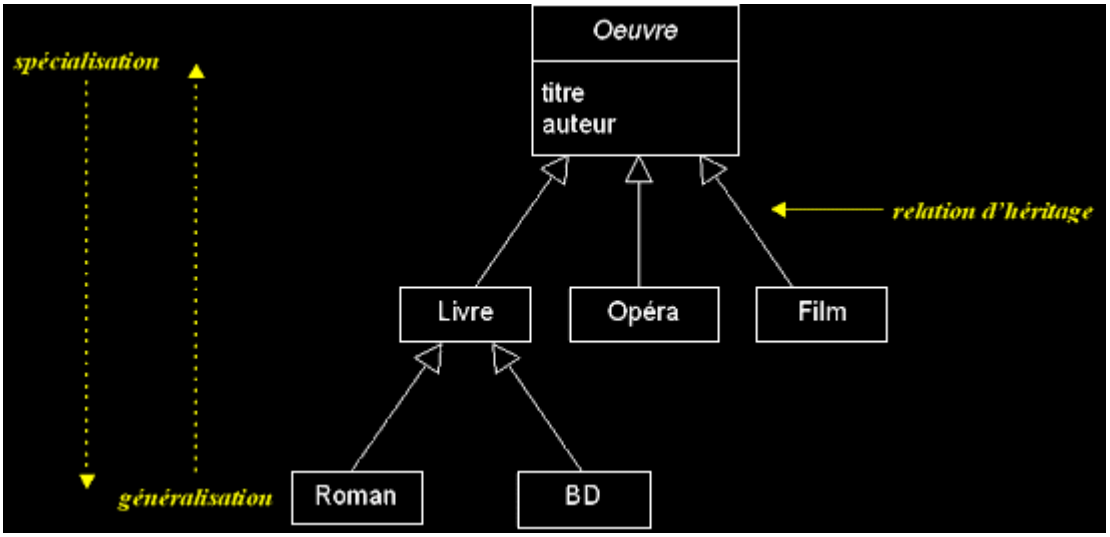
Héritage

Les hiérarchies de classes permettent de gérer la complexité, en ordonnant les objets au sein d'arborescences de classes, d'abstraction croissante.

- **Spécialisation**
  - Démarche descendante, qui consiste à capturer les particularités d'un ensemble d'objets, non discriminés par les classes déjà identifiées.
  - Consiste à étendre les propriétés d'une classe, sous forme de sous-classes, plus spécifiques (permet l'extension du modèle par réutilisation).

- **Généralisation**
  - Démarche ascendante, qui consiste à capturer les particularités communes d'un ensemble d'objets, issus de classes différentes.
  - Consiste à factoriser les propriétés d'un ensemble de classes, sous forme d'une super-classe, plus abstraite (permet de gagner en généricité).

- **Classification**
  - L'héritage (spécialisation et généralisation) permet la classification des objets.
  - Une bonne classification est stable et extensible : ne classifiez pas les objets selon des critères instables (selon ce qui caractérise leur état) ou trop vagues (car cela génère trop de sous-classes).
  - Les critères de classification sont **subjectifs**.
  - Le principe de substitution (Liksow, 1987) permet de déterminer si une relation d'héritage est bien employée pour la classification :
    "Il doit être possible de substituer n'importe quel instance d'une super-classe, par n'importe quel instance d'une de ses sous-classes, sans que la sémantique d'un programme écrit dans les termes de la super-classe n'en soit affectée."
  - Si Y hérite de X, cela signifie que "Y est une sorte de X" (analogies entre classification et théorie des ensembles).



Agrégation

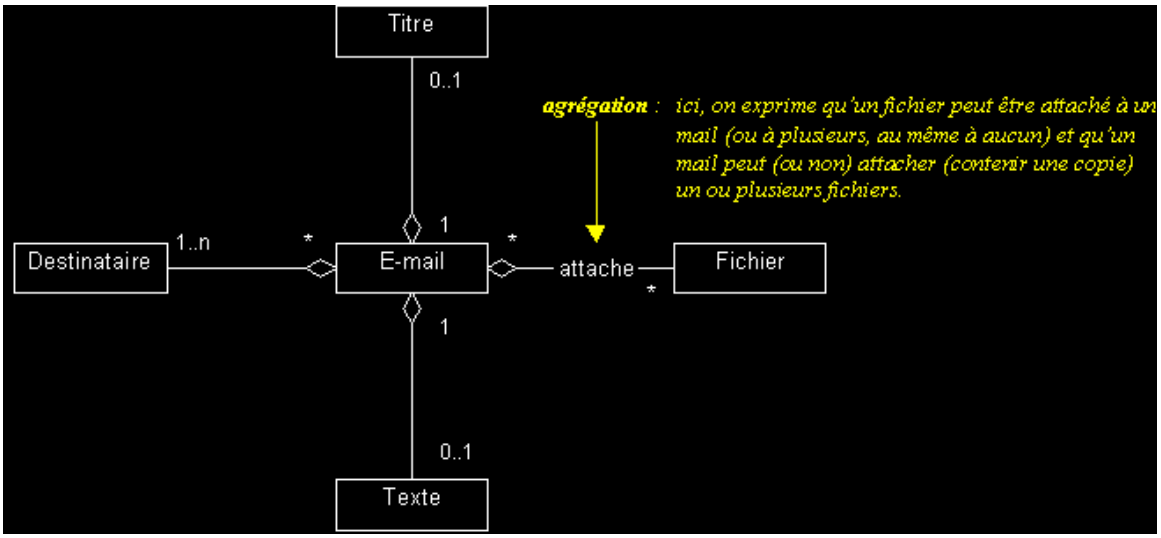
- L'agrégation est une association non symétrique, qui exprime un couplage fort et une relation de subordination.
  Elle représente une relation de type "ensemble / élément".

- UML ne définit pas ce qu'est une relation de type "ensemble / élément", mais il permet cependant d'exprimer cette vue **subjective** de manière explicite.

- Une agrégation peut notamment (mais pas nécessairement) exprimer :
  - qu'une classe (un "élément") fait partie d'une autre ("l'agrégat"),
  - qu'un changement d'état d'une classe, entraîne un changement d'état d'une autre,
  - qu'une action sur une classe, entraîne une action sur une autre.

- A un même moment, une instance d'élément agrégé peut être liée à plusieurs instances d'autres classes (l'élément agrégé peut être partagé).

- Une instance d'élément agrégé peut exister sans agrégat (et inversement) : les cycles de vies de l'agrégat et de ses éléments agrégés peuvent être indépendants.



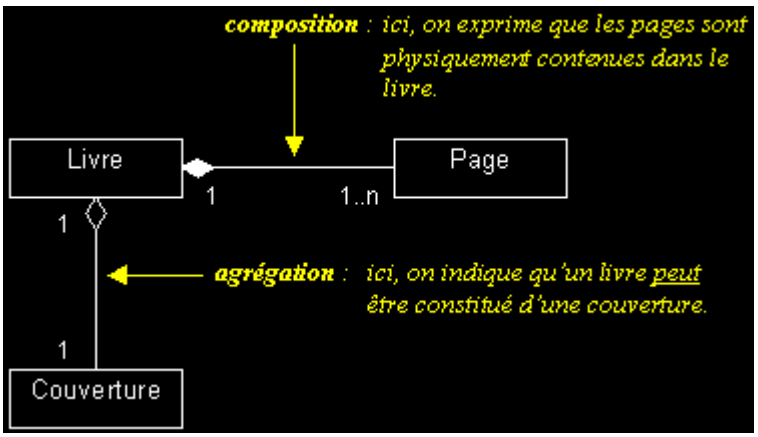
Composition

- La composition est une agrégation forte (agrégation par valeur).

- Les cycles de vies des éléments (les "composants") et de l'agrégat sont liés : si l'agrégat est détruit (ou copié), ses composants le sont aussi.

- A un même moment, une instance de composant ne peut être liée qu'à un seul agrégat.

- Les "objets composites" sont des instances de classes composées.

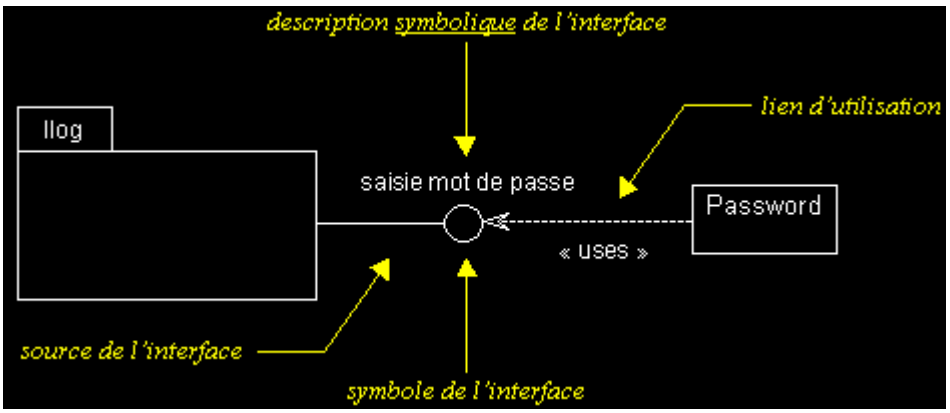


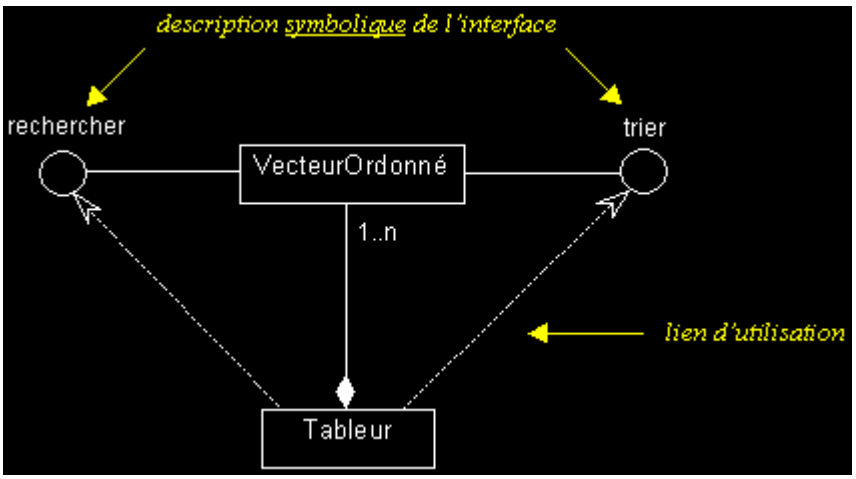
Agrégation et composition : rappel

- L'agrégation et la composition sont des vues subjectives.
- Lorsqu'on représente (avec UML) qu'une molécule est "composée" d'atomes, on sous-entend que la destruction d'une instance de la classe "Molécule", implique la destruction de ses composants, instances de la classe "Atome" (cf. propriétés de la composition).
- Bien qu'elle ne reflète pas la réalité ("rien ne se perd, rien ne se crée, tout se transforme"), cette abstraction de la réalité nous satisfait si l'objet principal de notre modélisation est la molécule...
- En conclusion, servez vous de l'agrégation et de la composition pour ajouter de la sémantique à vos modèles lorsque cela est pertinent, même si dans la "réalité" de tels liens n'existent pas !

Interfaces

- Une interface fournit une vue totale ou partielle d'un ensemble de services offerts par une classe, un paquetage ou un composant. Les éléments qui utilisent l'interface peuvent exploiter tout ou partie de l'interface.
- Dans un modèle UML, le symbole "interface" sert à identifier de manière explicite et symbolique les services offerts par un élément et l'utilisation qui en est faite par les autres éléments.


description symbolique de l'interface
lien d'utilisation
Ilog
saisie mot de passe
Password
« uses »
source de l'interface
symbole de l'interface


description symbolique de l'interface
rechercher
trier
VecteurOrdonné
1..n
lien d'utilisation
Tableur

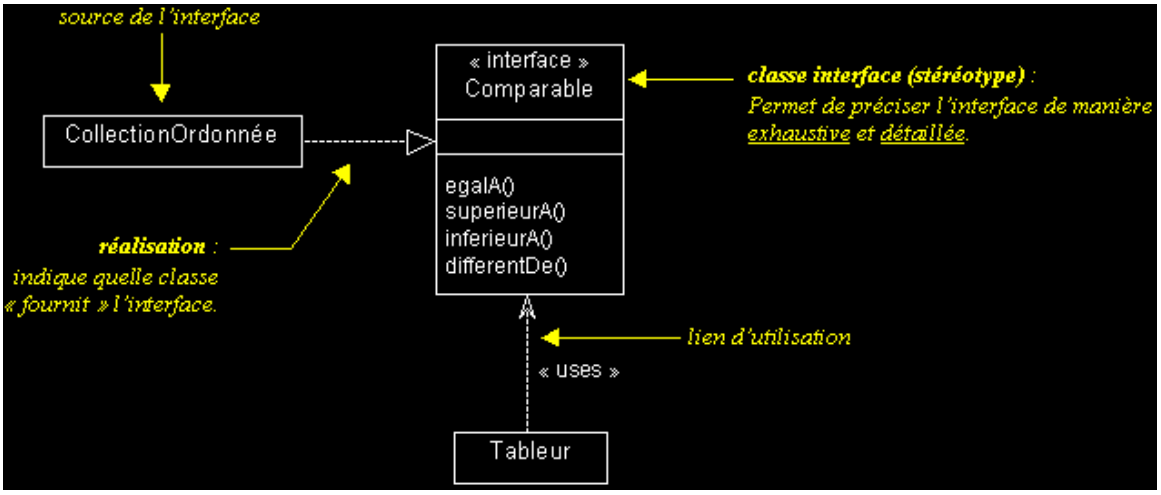
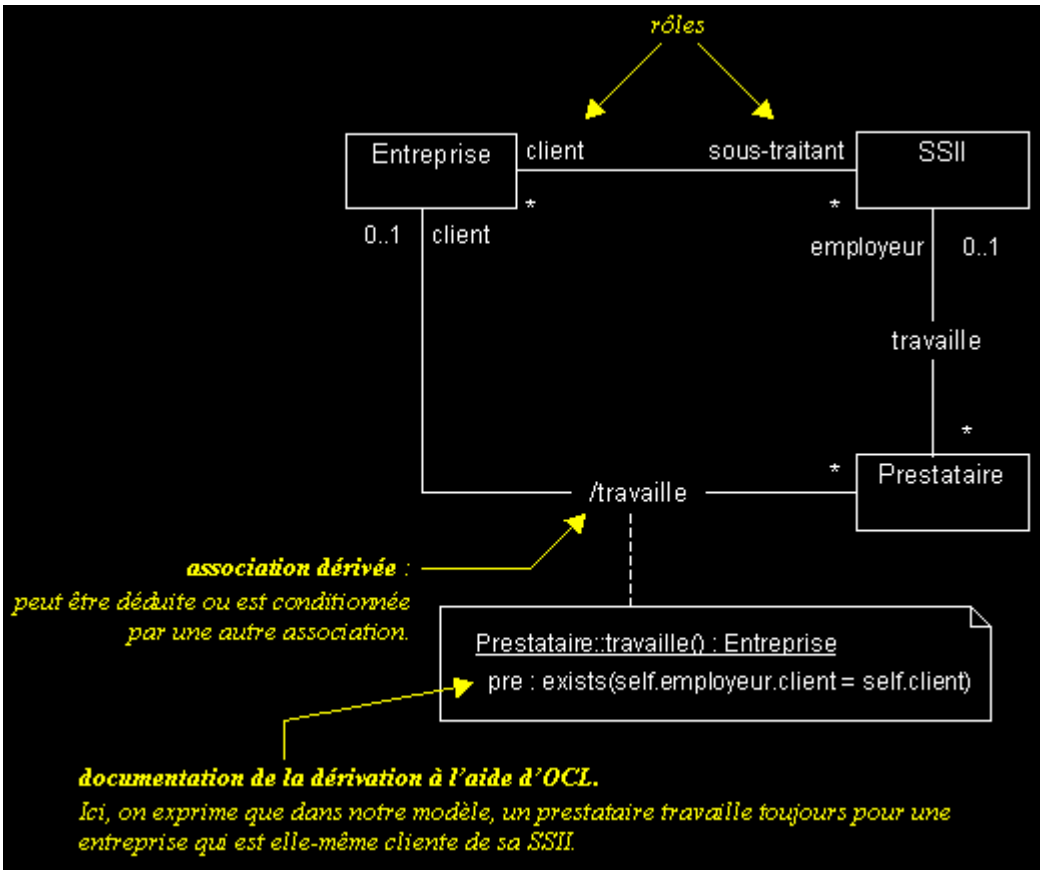

source de l'interface
« interface »
Comparable
classe interface (stéréotype) :
Permet de préciser l'interface de manière exhaustive et détaillée.
CollectionOrdonnée
egalA()
superieurA()
inferieurA()
differentDe()
réalisation :
indique quelle classe « fournit » l'interface.
lien d'utilisation
« uses »
Tableur

DIAGRAMME DE CLASSES (suite...)

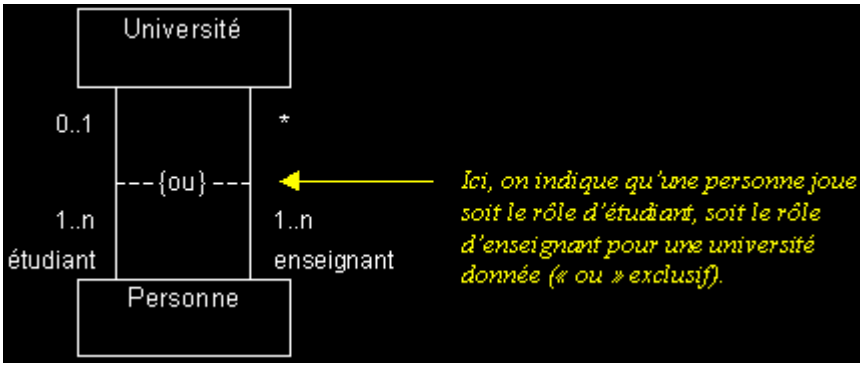
Association dérivée

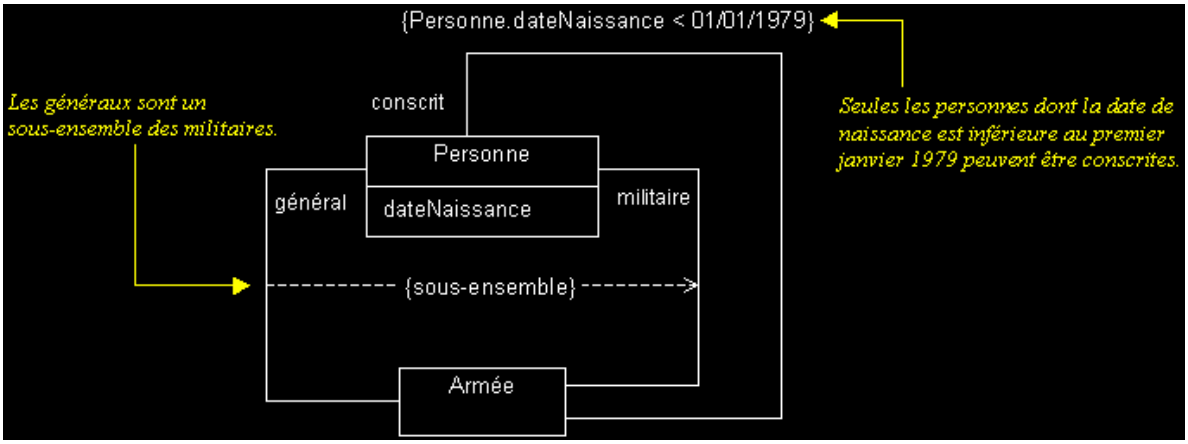
- Les associations dérivées sont des associations redondantes, qu'on peut déduire d'une autre association ou d'un ensemble d'autres associations.
- Elles permettent d'indiquer des chemins de navigation "calculés", sur un diagramme de classes.
- Elles servent beaucoup à la compréhension de la navigation (comment joindre telles instances d'une classe à partir d'une autre).

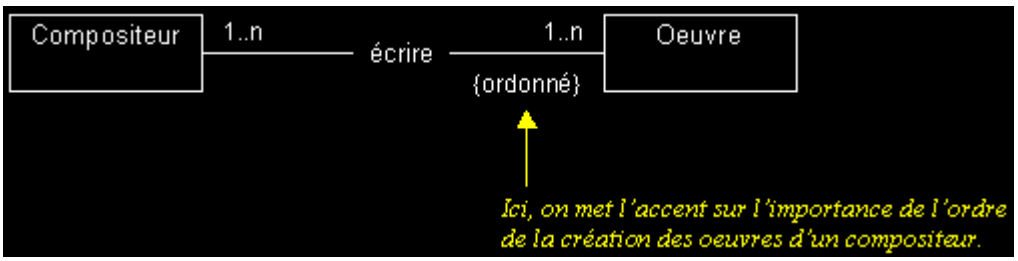


Contrainte sur une association

- Les contraintes sont des expressions qui précisent le rôle ou la portée d'un élément de modélisation (elles permettent d'étendre ou préciser sa sémantique).
- Sur une association, elles peuvent par exemple restreindre le nombre d'instances visées (ce sont alors des "expressions de navigation").
- Les contraintes peuvent s'exprimer en langage naturel. Graphiquement, il s'agit d'un texte encadré d'accolades.







OCL

- UML formalise l'expression des contraintes avec OCL (Object Constraint Language).
- OCL est une contribution d'IBM à UML 1.1.
- Ce langage formel est volontairement simple d'accès et possède une grammaire élémentaire (OCL peut être interprété par des outils).
- Il représente un juste milieu, entre langage naturel et langage mathématique. OCL permet ainsi de limiter les ambiguïtés, tout en restant accessible.
- OCL permet de décrire des invariants dans un modèle, sous forme de pseudo-code :

    - pré et post-conditions pour une opération,
    - expressions de navigation,
    - expressions booléennes, etc...

- OCL est largement utilisé dans la définition du métamodèle UML.

Nous allons nous baser sur une étude de cas, pour introduire brièvement OCL.

Monsieur Formulain, directeur d'une chaîne d'hôtels, vous demande de concevoir une application de gestion pour ses hôtels. Voici ce que vous devez modéliser :

Un hôtel Formulain est constitué d'un certain nombre de chambres. Un responsable de l'hôtel gère la location des chambres. Chaque chambre se loue à un prix donné (suivant ses prestations).

L'accès aux salles de bain est compris dans le prix de la location d'une chambre. Certaines chambres comportent une salle de bain, mais pas toutes. Les hôtes de chambres sans salle de bain peuvent utiliser une salle de bain sur le palier. Ces dernières peuvent être utilisées par plusieurs hôtes.

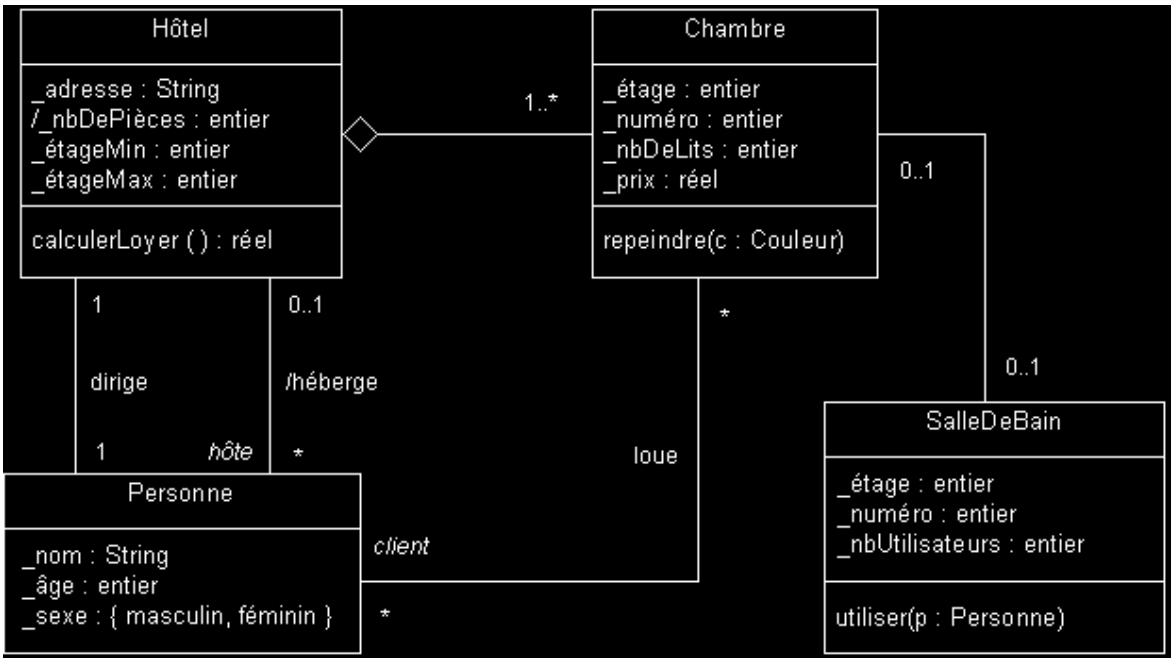
Les pièces de l'hôtel qui ne sont ni des chambres, ni des salles de bain (hall d'accueil, cuisine...) ne font pas partie de l'étude (hors sujet).

Des personnes peuvent louer une ou plusieurs chambres de l'hôtel, afin d'y résider. En d'autre termes : l'hôtel héberge un certain nombre de personnes, ses hôtes (il s'agit des personnes qui louent au moins une chambre de l'hôtel...).

Le diagramme UML ci-dessous présente les classes qui interviennent dans la modélisation d'un hôtel Formulain, ainsi que les relations entre ces classes.

Attention : le modèle a été réduit à une vue purement statique. La dynamique de l'interaction entre instances n'est pas donnée ici, pour simplifier l'exemple. Lors d'une modélisation complète, les vues dynamiques complémentaires ne devraient pas être omises (tout comme la conceptualisation préalable par des use cases)...

Remarque : cet exemple est inspiré d'un article paru dans JOOP (Journal of Object Oriented Programming), en mai 99.



OCL permet d'enrichir ce diagramme, en décrivant toutes les contraintes et tous les invariants du modèle présenté, de manière normalisée et explicite (à l'intérieur d'une note rattachée à un élément de modélisation du diagramme).
Voici quelques exemples de contraintes qu'on pourrait définir sur ce diagramme, avec la syntaxe OCL correspondante.

Attention !
Les exemples de syntaxe OCL ci-dessous ne sont pas détaillés, référez-vous au document de la norme UML adéquat ("OCL spécification"). Il ne s'agit là que d'un très rapide aperçu du pouvoir d'abstraction d'OCL...

Un hôtel Formulain ne contient jamais d'étage numéro 13 (superstition oblige).

```
context Chambre inv:
self._étage <> 13

context SalleDeBain inv:
self._étage <> 13
```

Le nombre de personnes par chambre doit être inférieur ou égal au nombre de lits dans la chambre louée. Les enfants (accompagnés) de moins de 4 ans ne "comptent pas" dans cette règle de calcul (à hauteur d'un enfant de moins de 4 ans maximum par chambre).

```
context Chambre inv:
client->size <= _nbDeLits or
  (client->size = _nbDeLits + 1 and
            client->exists(p : Personne | p._âge < 4))
```

L'étage de chaque chambre est compris entre le premier et le dernier étage de l'hôtel.

```
context Hôtel inv:
```

```
self.chambre->forAll(c : Chambre | c._étage <= self._étageMax and
                        c._étage >= self._étageMin)
```

Chaque étage possède au moins une chambre (sauf l'étage 13, qui n'existe pas...).

```
context Hôtel inv:
Sequence{_étageMin.._étageMax}->forAll(i : Integer |
      if i <> 13 then
        self.chambre->select(c : Chambre | c._étage = i)->notEmpty)
      endif)
```

On ne peut repeindre une chambre que si elle n'est pas louée. Une fois repeinte, une chambre coûte 10% de plus.

```
context Chambre::repeindre(c : Couleur)
pre: client->isEmpty
post: _prix = _prix@pre * 1.1
```

Une salle de bain privative ne peut être utilisée que par les personnes qui louent la chambre contenant la salle de bain et une salle de bain sur le palier ne peut être utilisée que par les clients qui logent sur le même palier.

```
context SalleDeBain::utiliser(p : Personne)
pre: if chambre->notEmpty then
       chambre.client->includes(p)
     else
       p.chambre._étage = self._étage
     endif
post: _nbUtilisateurs = _nbUtilisateurs@pre + 1
```

Le loyer de l'hôtel est égal à la somme du prix de toutes les chambres louées.

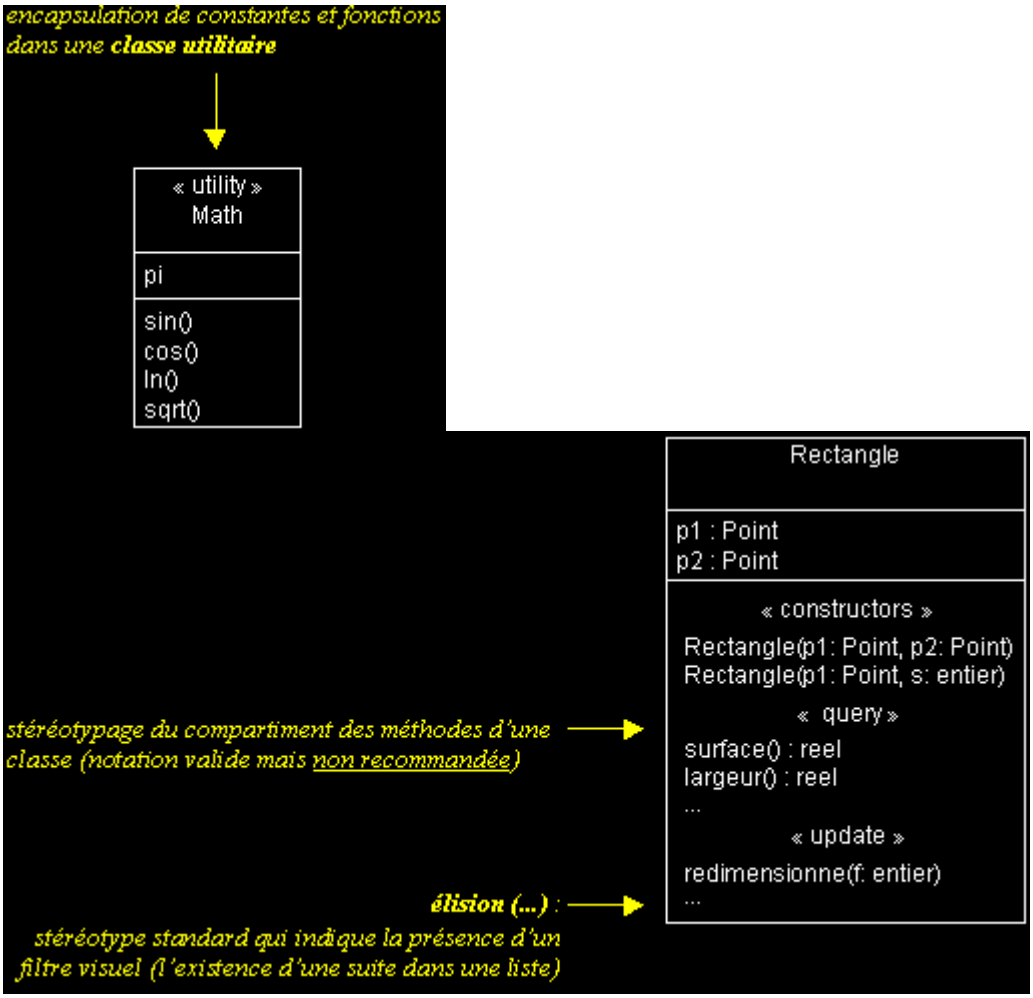
```
context Hôtel::calculerLoyer() : réel
pre:
post: result = self.chambre->select(client->notEmpty)._prix->sum
```

Stéréotypes

- Les stéréotypes permettent d'étendre la sémantique des éléments de modélisation : il s'agit d'un mécanisme d'extensibilité du métamodèle d'UML.

- Les stéréotypes permettent de définir de nouvelles classes d'éléments de modélisation, en plus du noyau prédéfini par UML.

- Utilisez les stéréotypes avec modération et de manière concertée (notez aussi qu'UML propose de nombreux stéréotypes standards).


encapsulation de constantes et fonctions dans une classe utilitaire
« utility »
Math
pi
sin()
cos()
ln()
sqrt()
Rectangle
p1 : Point
p2 : Point
« constructors »
Rectangle(p1: Point, p2: Point)
Rectangle(p1: Point, s: entier)
« query »
surface() : reel
largeur() : reel
...
« update »
redimensionne(f: entier)
...
stéréotypage du compartiment des méthodes d'une classe (notation valide mais non recommandée)
élision (...) :
stéréotype standard qui indique la présence d'un filtre visuel (l'existence d'une suite dans une liste)

DIAGRAMMES DE COMPOSANTS ET DE DEPLOIEMENT

Diagramme de composants

- Les diagrammes de composants permettent de décrire l'architecture physique et statique d'une application en terme de modules : fichiers sources, librairies, exécutables, etc.
  Ils montrent la mise en oeuvre physique des modèles de la vue logique avec l'environnement de développement.

- Les dépendances entre composants permettent notamment d'identifier les contraintes de compilation et de mettre en évidence la réutilisation de composants.

- Le composants peuvent être organisés en paquetages, qui définissent des sous-systèmes. Les sous-systèmes organisent la vue des composants (de réalisation) d'un système. Ils permettent de gérer la complexité, par encapsulation des détails d'implémentation.

Modules (notation) :

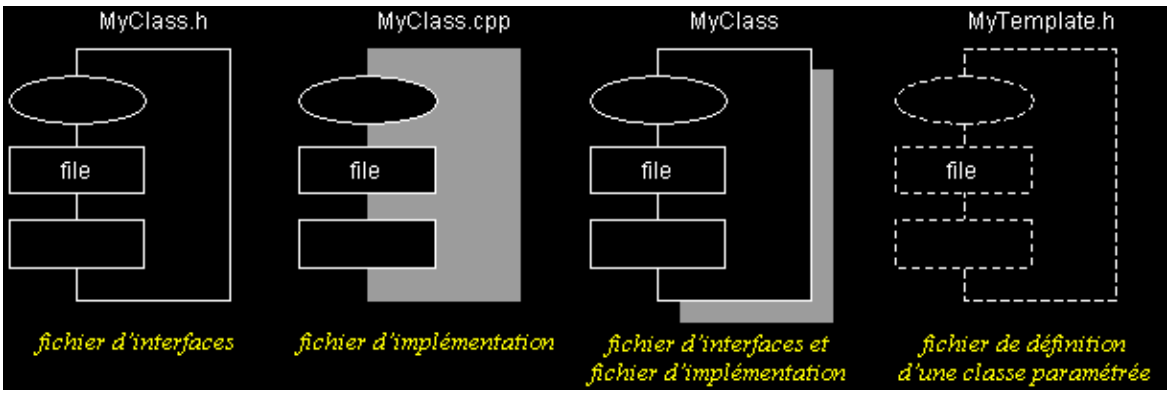


Diagramme de composants (exemple) :

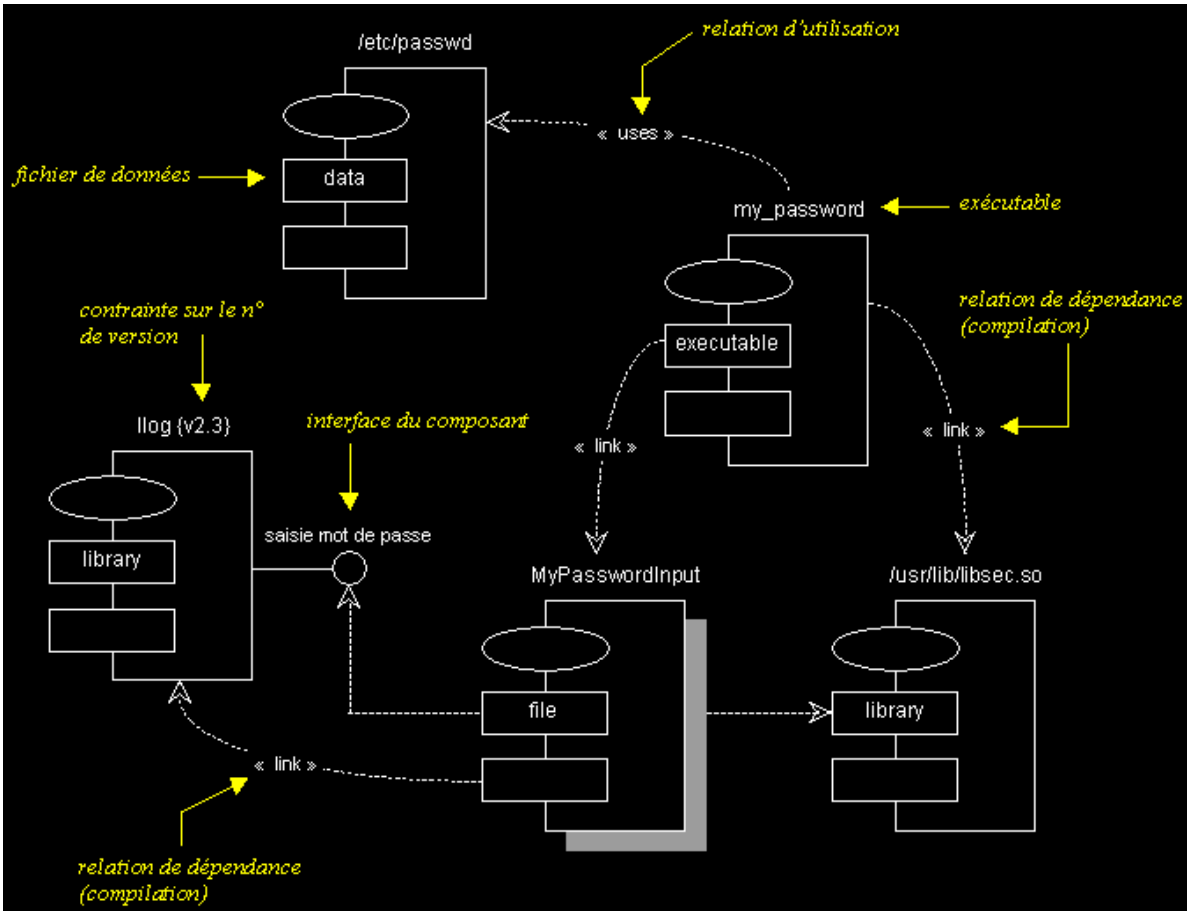
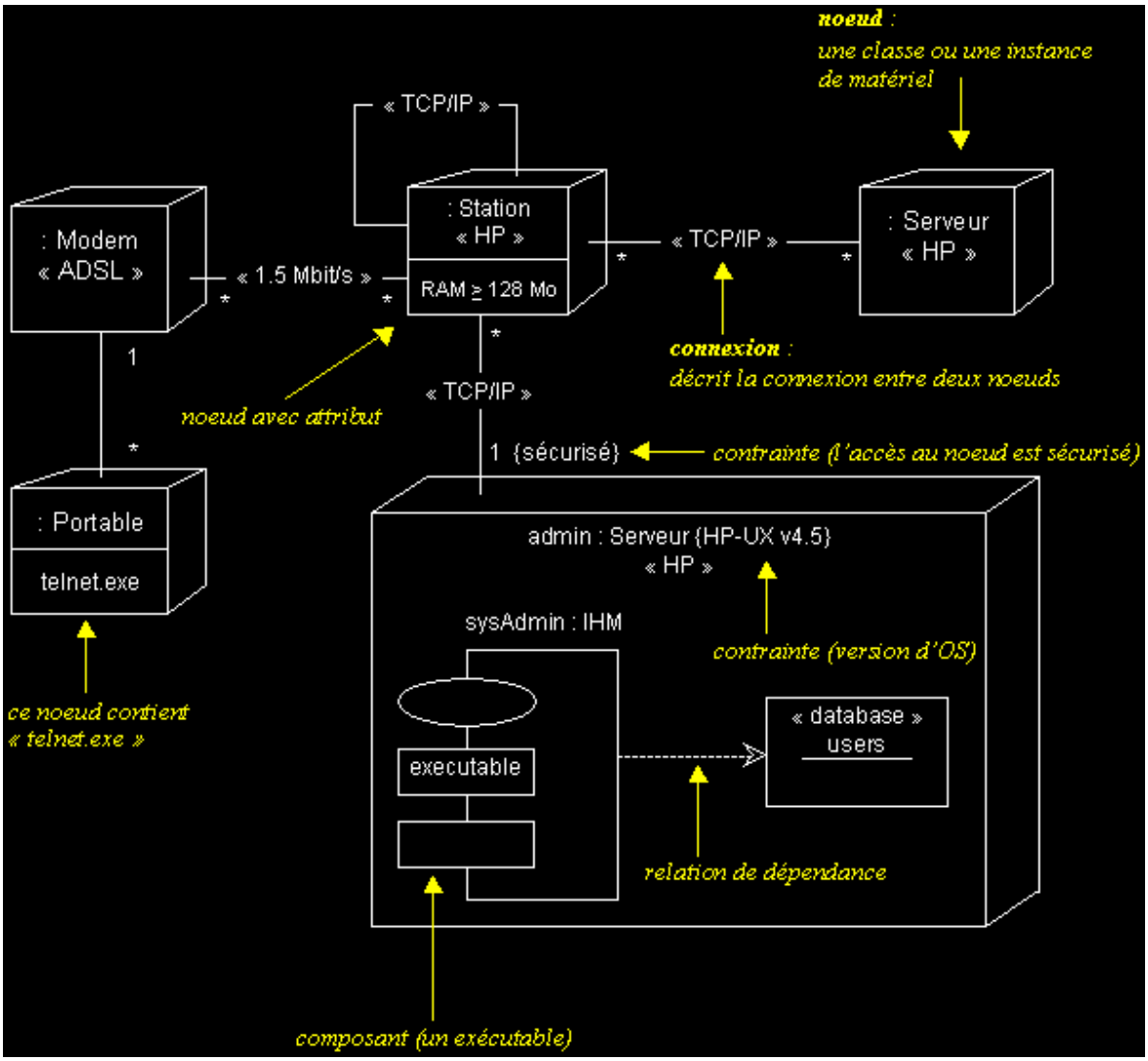


Diagramme de déploiement

- Les diagrammes de déploiement montrent la disposition physique des matériels qui composent le système et la répartition des composants sur ces matériels.
- Les ressources matérielles sont représentées sous forme de noeuds.
- Les noeuds sont connectés entre eux, à l'aide d'un support de communication.
  La nature des lignes de communication et leurs caractéristiques peuvent être précisées.
- Les diagrammes de déploiement peuvent montrer des instances de noeuds (un matériel précis), ou des classes de noeuds.
- Les diagrammes de déploiement correspondent à la vue de déploiement d'une architecture logicielle (vue "4+1").


noeud :
une classe ou une instance de matériel
« TCP/IP »
: Station « HP »
RAM ≥ 128 Mo
: Modem « ADSL »
« 1.5 Mbit/s »
« TCP/IP »
: Serveur « HP »
connexion :
décrit la connexion entre deux noeuds
noeud avec attribut
« TCP/IP »
1 {sécurisé}
contrainte (l'accès au noeud est sécurisé)
: Portable
telnet.exe
admin : Serveur {HP-UX v4.5}
« HP »
sysAdmin : IHM
contrainte (version d'OS)
« database »
users
executable
relation de dépendance
ce noeud contient « telnet.exe »
composant (un exécutable)

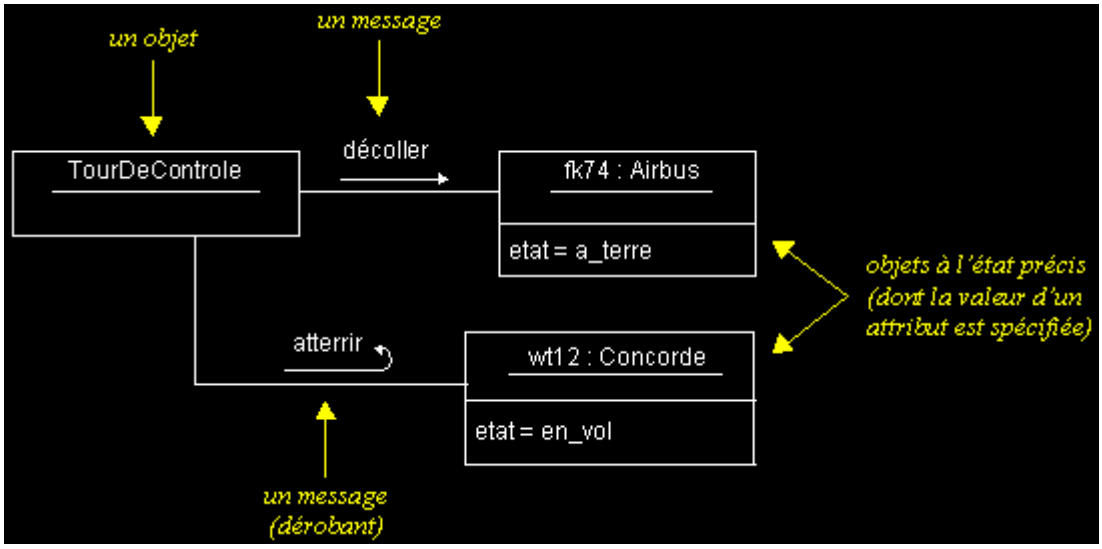
Les vues dynamiques d'UML

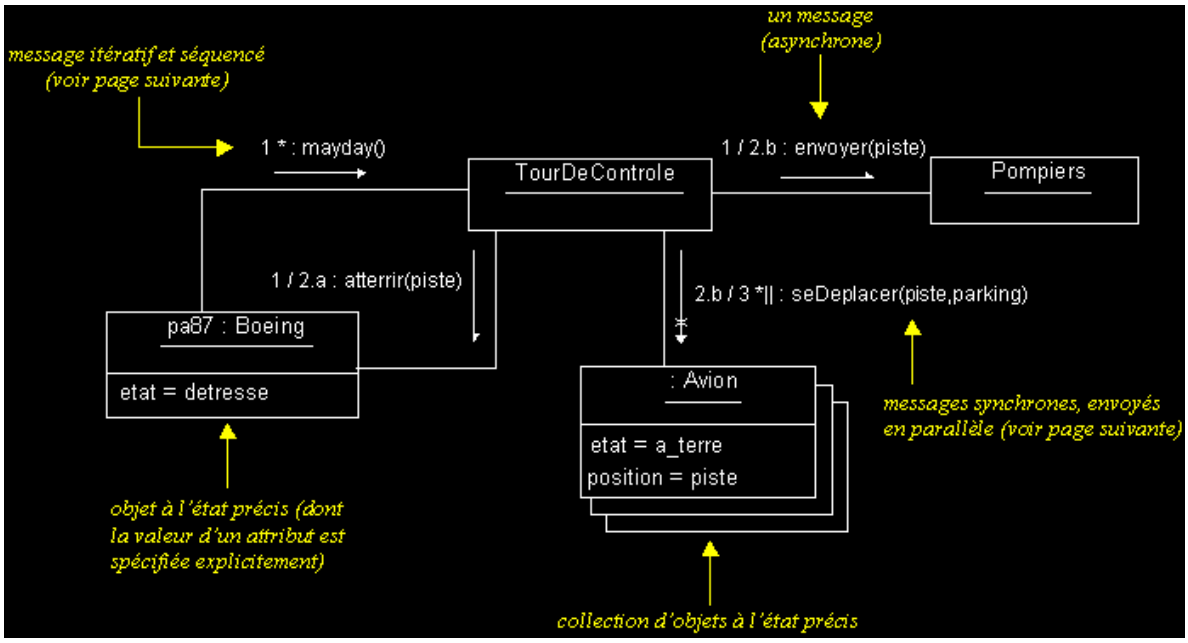
COLLABORATION ET MESSAGES

Diagramme de collaboration

- Les diagrammes de collaboration montrent des interactions entre objets (instances de classes et acteurs).
- Ils permettent de représenter le contexte d'une interaction, car on peut y préciser les états des objets qui interagissent.

Exemples :





Synchronisation des messages

- UML permet de spécifier de manière très précise l'ordre et les conditions d'envoi des messages sur un diagramme dynamique.

- Pour chaque message, il est possible d'indiquer :
  - les clauses qui conditionnent son envoi,
  - son rang (son numéro d'ordre par rapport aux autres messages),
  - sa récurrence,
  - ses arguments.

- La syntaxe d'un message est la suivante :

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

  - pré : prédécesseurs (liste de numéros de séquence de messages séparés par une virgule ; voir aussi "séq").
    Indique que le message courant ne sera envoyé que lorsque tous ses prédécesseurs le seront aussi (permet de synchroniser l'envoi de messages).

  - cond : garde, expression booléenne.
    Permet de conditionner l'envoi du message, à l'aide d'une clause exprimée en langage naturel.

  - séq : numéro de séquence du message.
    Indique le rang du message, c'est-à-dire son numéro d'ordre par rapport aux autres messages. Les messages sont numérotés à la façon de chapitres dans un document, à l'aide de chiffres séparés par des points. Ainsi, il est possible de représenter le niveau d'emboîtement des messages et leur précédence.
    Exemple : l'envoi du message 1.3.5 suit immédiatement celui du message 1.3.4 et ces deux messages font partie du flot (de la famille de messages) 1.3.
    Pour représenter l'envoi simultané de deux messages, il suffit de les indexer par une lettre.
    Exemple : l'envoi des messages 1.3.a et 1.3.b est simultané.

  - iter : récurrence du message.
    Permet de spécifier en langage naturel l'envoi séquentiel (ou en parallèle, avec "||") de messages. Notez qu'il est aussi possible de spécifier qu'un message est récurrent en omettant la clause d'itération (en n'utilisant que "*" ou "*||").

  - r : valeur de retour du message.
    Permet d'affecter la valeur de retour d'un message, pour par exemple la retransmettre dans un autre message, en tant que paramètre.

  - msg : nom du message.

  - par : paramètres (optionnels) du message.

Exemples :

3 : bonjour()

Ce message a pour numéro de séquence "3".

[heure = midi] 1 : manger()

Ce message n'est envoyé que s'il est midi.

1.3.6 * : ouvrir()

Ce message est envoyé de manière séquentielle un certain nombre de fois.

3 / *||[i := 1..5] : fermer()

Représente l'envoi en parallèle de 5 messages. Ces messages ne seront envoyés qu'après l'envoi du message 3.

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

Ce message (numéro 2.5) ne sera envoyé qu'après les messages 1.3 et 2.1, et que si "t < 10s".

1.3 / [disk full] 1.7.a * : deleteTempFiles()
1.3 / [disk full] 1.7.b : reduceSwapFile(20%)

Ces messages ne seront envoyés qu'après l'envoi du message 1.3 et si la condition "disk full" est réalisée. Si cela est le cas, les messages 1.7.a et 1.7.b seront envoyés simultanément. Plusieurs messages 1.7.a peuvent être envoyés.

Objets actifs (threads)

- UML permet de représenter des communications entre objets actifs de manière concurrente.
- Cette extension des diagrammes de collaboration permet notamment de représenter des communications entre processus ou l'exécution de threads.

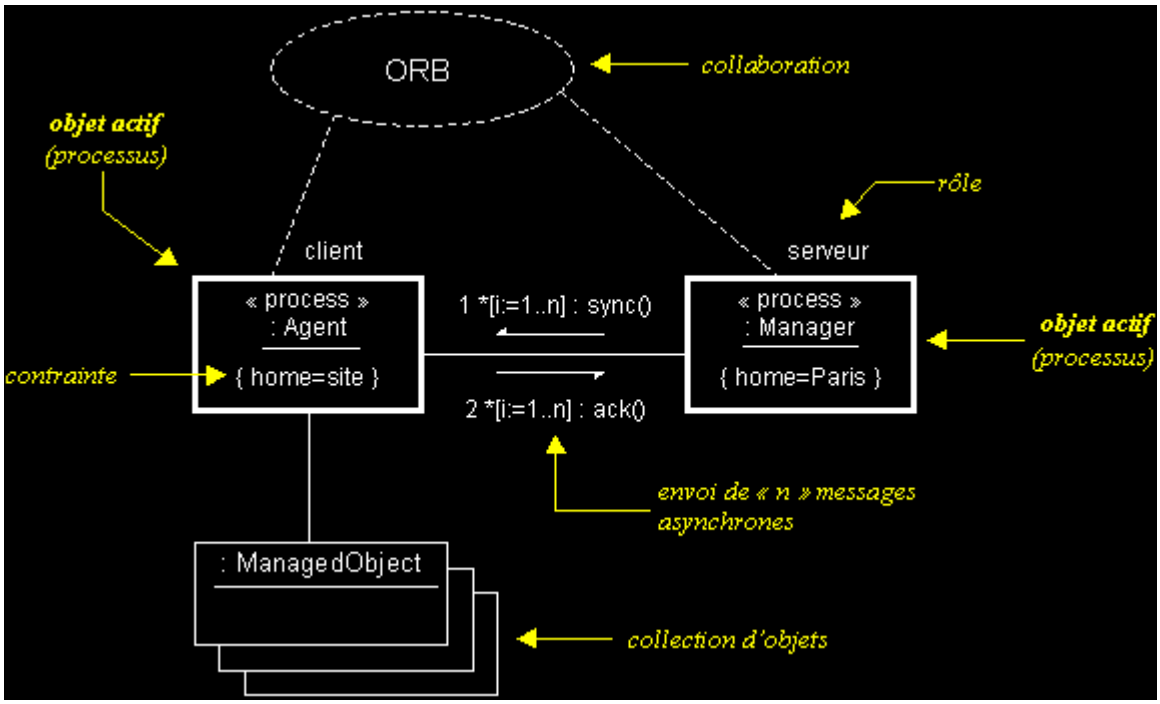
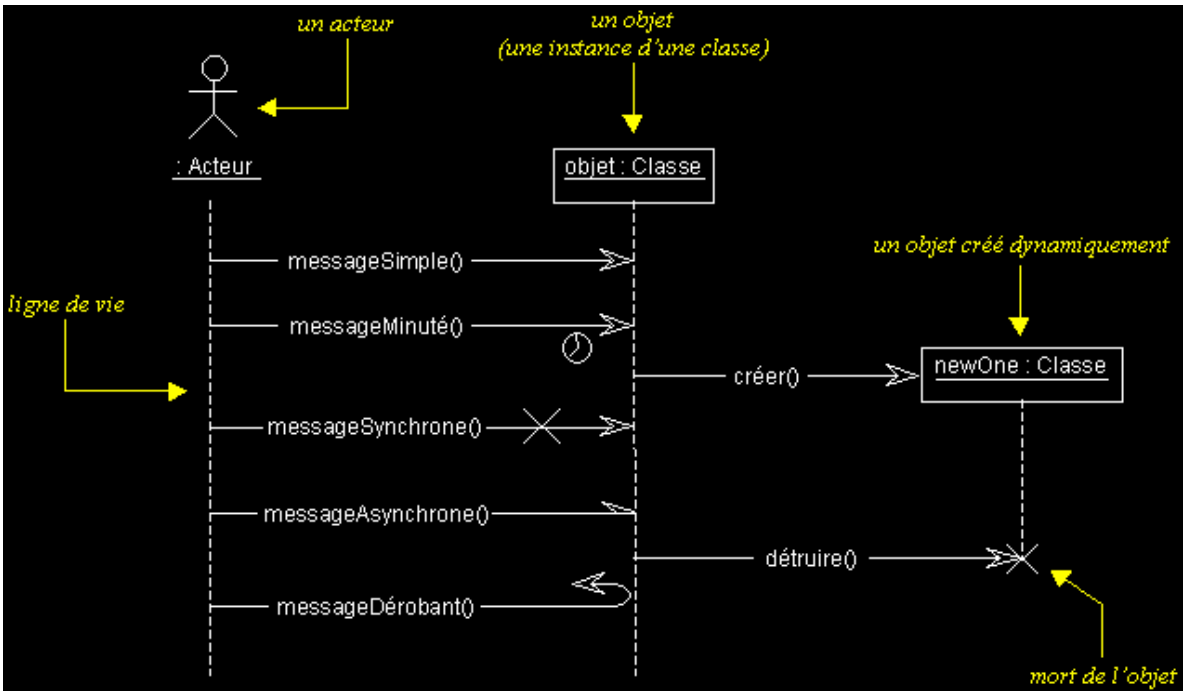

ORB
collaboration
objet actif
(processus)
rôle
client
serveur
« process »
: Agent
{ home=site }
1 *[i:=1..n] : sync()
2 *[i:=1..n] : ack()
« process »
: Manager
{ home=Paris }
objet actif
(processus)
contrainte
envoi de « n » messages
asynchrones
: ManagedObject
collection d'objets

DIAGRAMME DE SEQUENCE

Diagramme de séquence : sémantique

- Les diagrammes de séquences permettent de représenter des collaborations entre objets selon un point de vue temporel, on y met l'accent sur la chronologie des envois de messages.
- Contrairement au diagramme de collaboration, on n'y décrit pas le contexte ou l'état des objets, la représentation se concentre sur l'expression des interactions.
- Les diagrammes de séquences peuvent servir à illustrer un cas d'utilisation.
- L'ordre d'envoi d'un message est déterminé par sa position sur l'axe vertical du diagramme ; le temps s'écoule "de haut en bas" de cet axe.
- La disposition des objets sur l'axe horizontal n'a pas de conséquence pour la sémantique du diagramme.
- Les diagrammes de séquences et les diagrammes d'état-transitions sont les vues dynamiques les plus importantes d'UML.

Exemple :



Types de messages

Comme vous pouvez le voir dans l'exemple ci-dessus, UML propose un certain nombre de stéréotypes graphiques pour décrire la nature du message (ces stéréotypes graphiques

s'appliquent également aux messages des diagrammes de collaborations) :

- **message simple**
  Message dont on ne spécifie aucune caractéristique d'envoi ou de réception particulière.

- **message minuté** (timeout)
  Bloque l'expéditeur pendant un temps donné (qui peut être spécifié dans une contrainte), en attendant la prise en compte du message par le récepteur. L'expéditeur est libéré si la prise en compte n'a pas eu lieu pendant le délai spécifié.

- **message synchrone**
  Bloque l'expéditeur jusqu'à prise en compte du message par le destinataire. Le flot de contrôle passe de l'émetteur au récepteur (l'émetteur devient passif et le récepteur actif) à la prise en compte du message.

- **message asynchrone**
  N'interrompt pas l'exécution de l'expéditeur. Le message envoyé peut être pris en compte par le récepteur à tout moment ou ignoré (jamais traité).

- **message dérobant**
  N'interrompt pas l'exécution de l'expéditeur et ne déclenche une opération chez le récepteur que s'il s'est préalablement mis en attente de ce message.

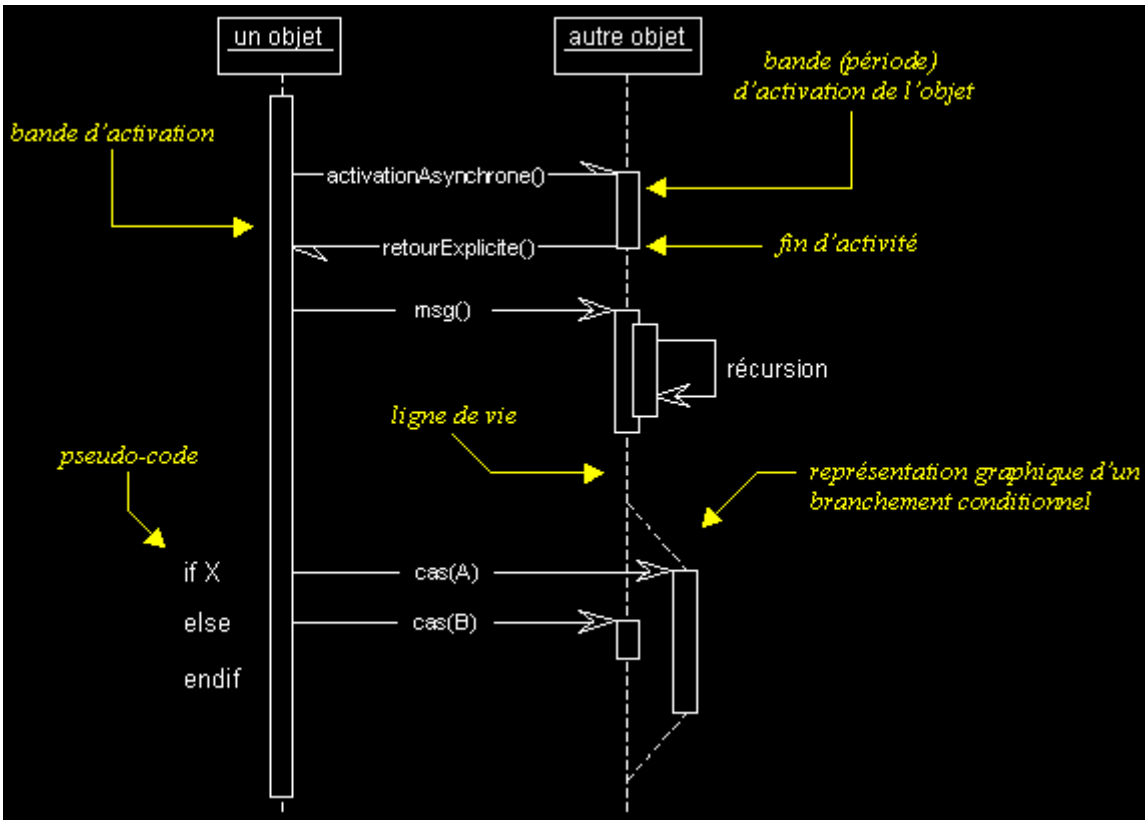
Activation d'un objet

Sur un diagramme de séquence, il est aussi possible de représenter de manière explicite les différentes périodes d'activité d'un objet au moyen d'une bande rectangulaire superposée à la ligne de vie de l'objet.

On peut aussi représenter des messages récursifs, en dédoublant la bande d'activation de l'objet concerné.

Pour représenter de manière graphique une exécution conditionnelle d'un message, on peut documenter un diagramme de séquence avec du pseudo-code et représenter des bandes d'activation conditionnelles.

Exemple :



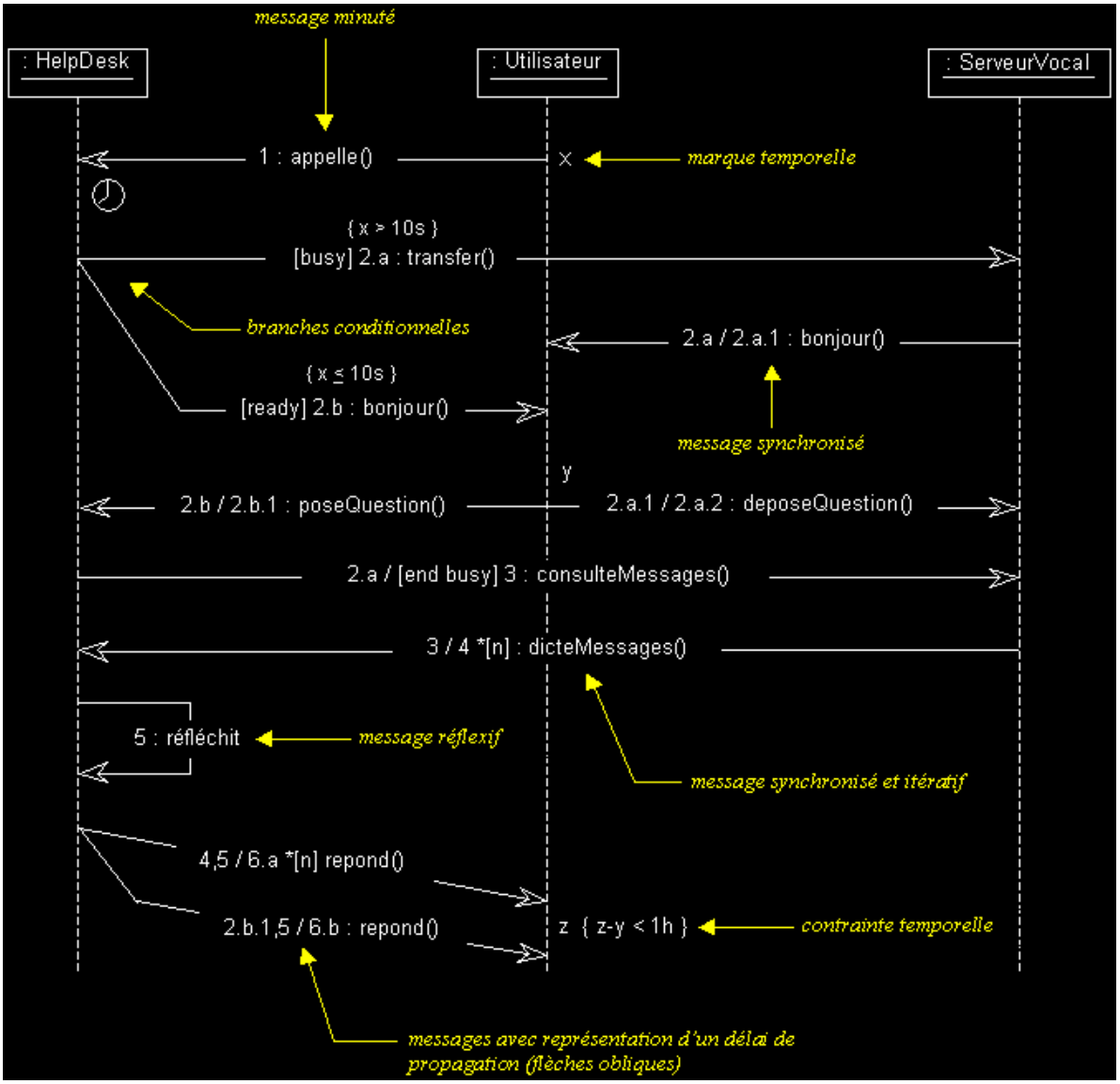
Commentaires :

- Ne confondez la période d'activation d'un objet avec sa création ou sa destruction. Un objet peut être actif plusieurs fois au cours de son existence (voir exemple ci-dessus).
- Le pseudo-code peut aussi être utilisé pour indiquer des itérations (avec incrémentation d'un paramètre d'un message par exemple).
- Le retour des messages asynchrones devrait toujours être matérialisé, lorsqu'il existe.
- Notez qu'il est fortement recommandé de synchroniser vos messages, comme sur l'exemple qui suit...
- L'exemple qui suit présente aussi une alternative intéressante pour la représentation des branchements conditionnels. Cette notation est moins lourde que celle utilisée dans l'exemple ci-dessus.
- Préférez aussi l'utilisation de contraintes à celle de pseudo-code, comme dans l'exemple qui suit.

Exemple complet

Afin de mieux comprendre l'exemple ci-dessous, veuillez vous référer aux chapitres sur la synchronisation des messages. Notez aussi l'utilisation des contraintes pour documenter les

conditions d'envoi de certains messages.



Commentaire :

Un message réflexif ne représente pas l'envoi d'un message, il représente une activité interne à l'objet (qui peut être détaillée dans un diagramme d'activités) ou une abstraction d'une autre interaction (qu'on peut détailler dans un autre diagramme de séquence).

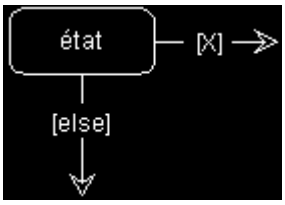
DIAGRAMME D'ETATS-TRANSITIONS

Diagramme d'états-transitions : sémantique

- Ce diagramme sert à représenter des automates d'états finis, sous forme de graphes d'états, reliés par des arcs orientés qui décrivent les transitions.
- Les diagrammes d'états-transitions permettent de décrire les changements d'états d'un objet ou d'un composant, en réponse aux interactions avec d'autres objets/composants ou avec des acteurs.
- Un état se caractérise par sa durée et sa stabilité, il représente une conjonction instantanée des valeurs des attributs d'un objet.
- Une transition représente le passage instantané d'un état vers un autre.
- Une transition est déclenchée par un événement. En d'autres termes : c'est l'arrivée d'un événement qui conditionne la transition.
- Les transitions peuvent aussi être automatiques, lorsqu'on ne spécifie pas l'événement qui la déclenche.
- En plus de spécifier un événement précis, il est aussi possible de conditionner une transition, à l'aide de "gardes" : il s'agit d'expressions booléennes, exprimées en langage naturel (et encadrées de crochets).

états, transition et événement, notation :

transition conditionnelle :

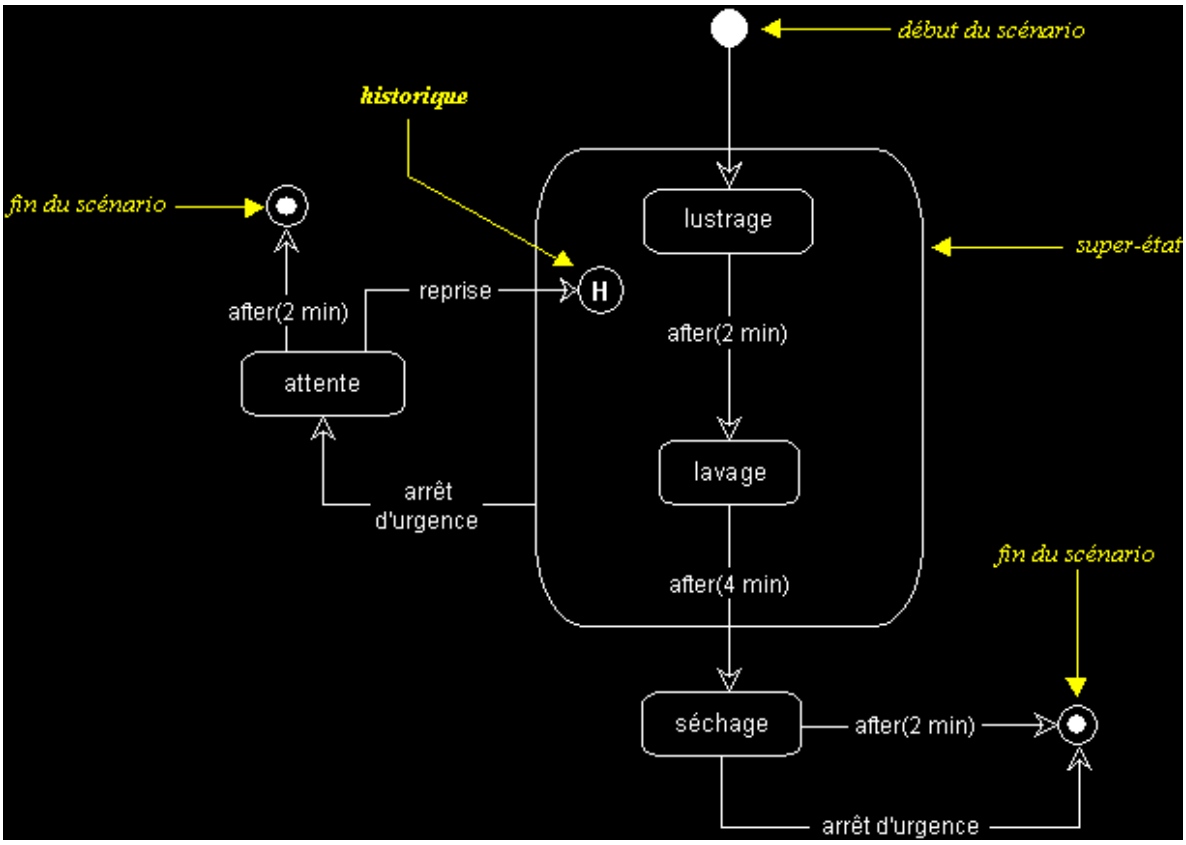


Super-Etat, historique et souches

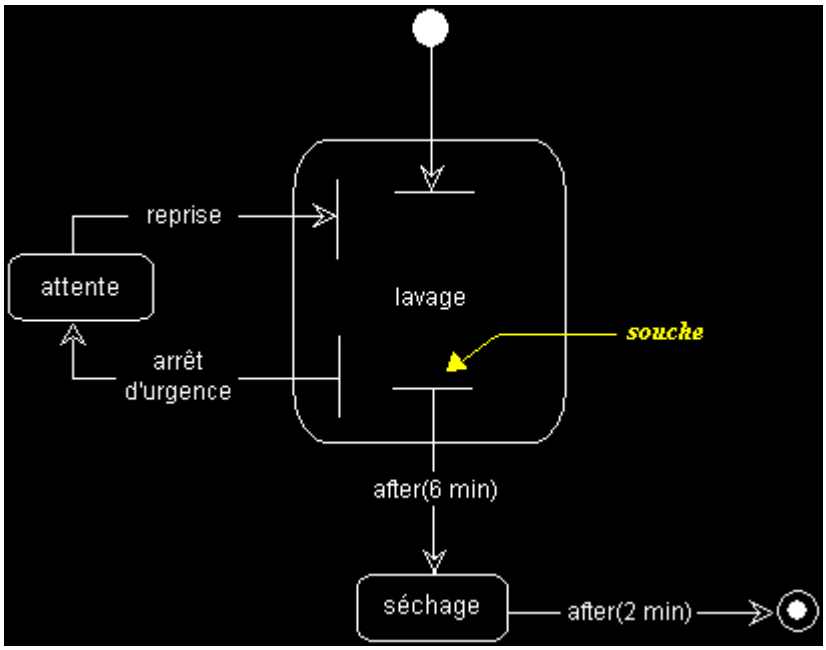
- Un super-état est un élément de structuration des diagrammes d'états-transitions (il s'agit d'un état qui englobe d'autres états et transitions).
- Le symbole de modélisation "historique", mémorise le dernier sous-état actif d'un super-état, pour y revenir directement ultérieurement.

Exemple :

Le diagramme d'états-transitions ci-dessous, montre les différents états par lesquels passe une machine à laver les voitures.
En phase de lustrage ou de lavage, le client peut appuyer sur le bouton d'arrêt d'urgence. S'il appuie sur ce bouton, la machine se met en attente. Il a alors deux minutes pour reprendre le lavage ou le lustrage (la machine continue en phase de lavage ou de lustrage, suivant l'état dans lequel elle a été interrompue), sans quoi la machine s'arrête. En phase de séchage, le client peut aussi interrompre la machine. Mais dans ce cas, la machine s'arrêtera définitivement (avant de reprendre un autre cycle entier).



- **souches** : afin d'introduire plus d'abstraction dans un diagramme d'états-transitions complexe, il est possible de réduire la charge d'information, tout en matérialisant la présence de sous-états, à l'aide de souches, comme dans l'exemple ci-dessous.



Actions dans un état

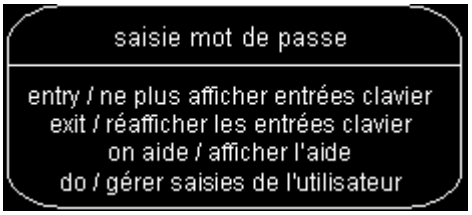
- On peut aussi associer une action à l'événement qui déclenche une transition.
  La syntaxe est alors la suivante : événement / action

- Ceci exprime que la transition (déclenchée par l'événement cité) entraîne l'exécution de l'action spécifiée sur l'objet, à l'entrée du nouvel état.
  Exemple : il pleut / ouvrir parapluie

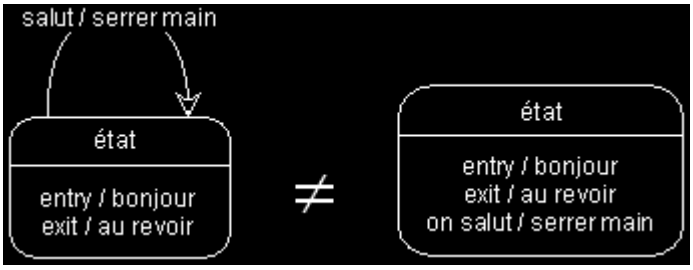
- Une action correspond à une opération disponible dans l'objet dont on représente les états.

- Les actions propres à un état peuvent aussi être documentées directement à l'intérieur de l'état.
  UML définit un certain nombre de champs qui permettent de décrire les actions dans un état :
  - entry / *action* : action exécutée à l'entrée de l'état
  - exit / *action* : action exécutée à la sortie de l'état
  - on *événement* / *action* : action exécutée à chaque fois que l'événement cité survient
  - do / *action* : action récurrente ou significative, exécutée dans l'état

Exemple :

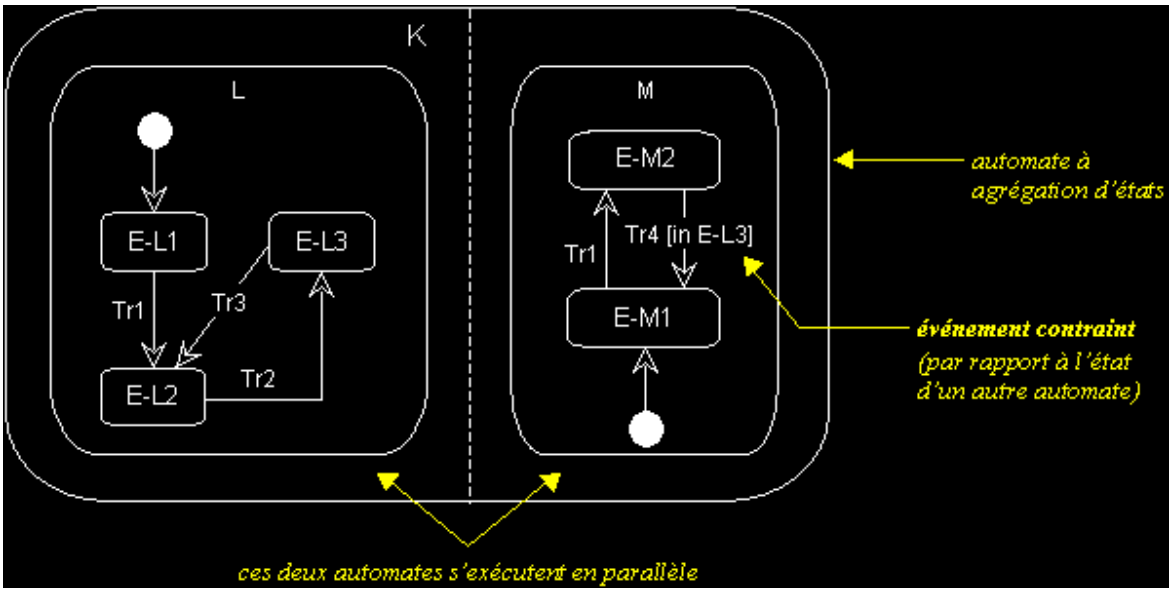


Remarque :
Attention, les actions attachées aux clauses "entry" et "exit" ne sont pas exécutées si l'événement spécifié dans la clause "on" survient. Pour indiquer qu'elles peuvent être exécutées plusieurs fois à l'arrivée d'un événement, représentez l'arrivée d'un événement réflexif, comme suit :



Etats concurrents et barre de synchronisation

Pour représenter des états concurrents sur un même diagramme d'états-transitions, on utilise la notation suivante :



Dans l'exemple ci-dessus, l'automate K est composé des sous-automates L et M.

L et M s'activent simultanément et évoluent en parallèle. Au départ, l'objet dont on modélise les états par l'automate K est dans l'état composite (E-L1, E-M1).

Après l'événement Tr1, K passe dans l'état composite (E-L2, E-M2). Par la suite, si l'événement Tr2 survient, K passe dans l'état composite (E-L3, E-M2). Si c'est Tr4 qui survient, M ne passe pas dans l'état E-M1, car cette transition est contrainte par l'état de L ("[in E-L3]").

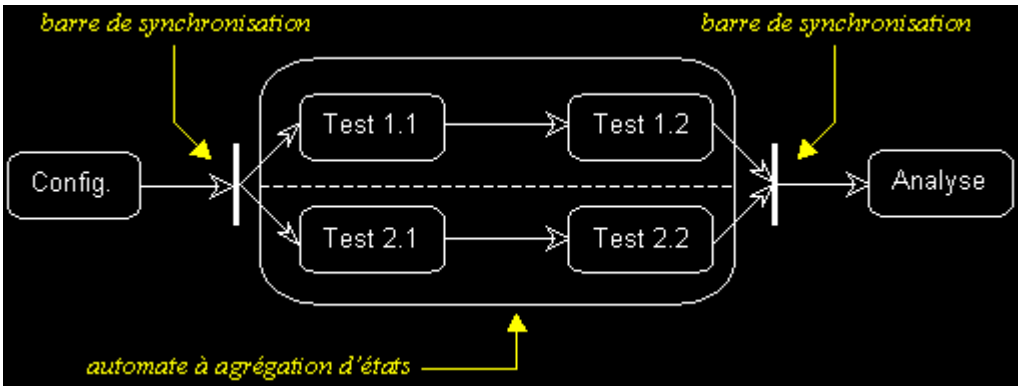
Dans l'état composite (E-L3, E-M2), si Tr3 survient, K passe dans l'état composite (E-L2, E-M2). Si

c'est Tr4 qui survient, K passe dans l'état composite (E-L3, E-M1). Et ainsi de suite...

Attention : **la numérotation des événements n'est pas significative.** Pour synchroniser les sous-automates d'une agrégation d'états, il faut contraindre les transitions, comme dans l'exemple ci-dessus ("[in E-L3]").

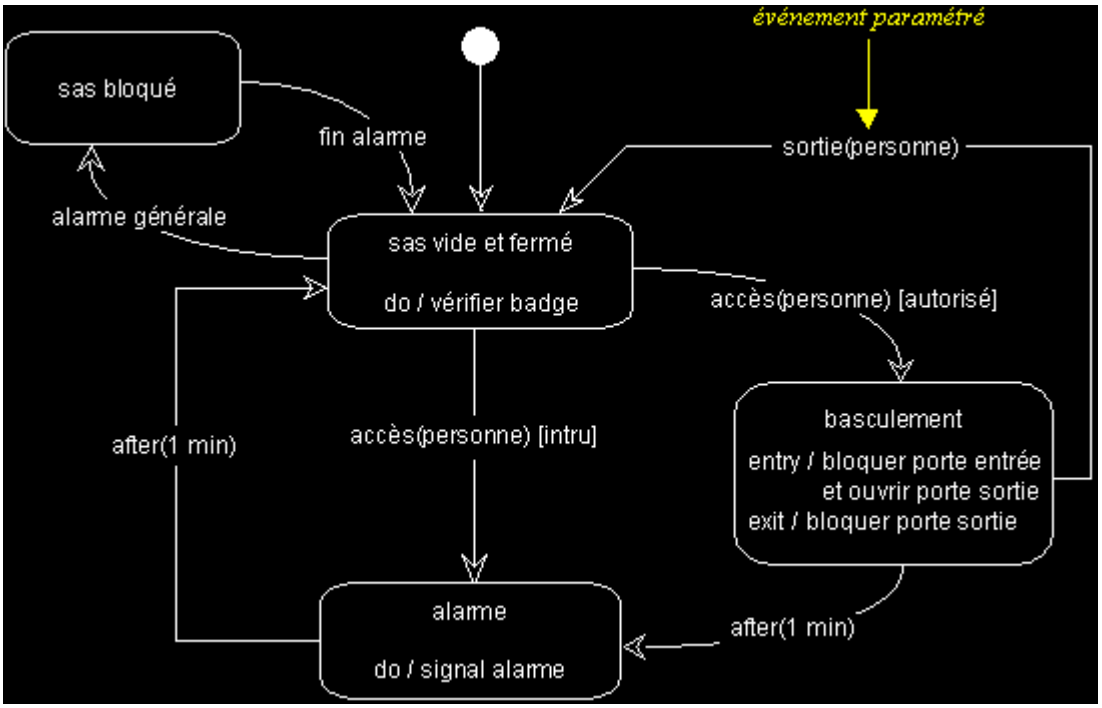
On peut aussi utiliser un symbole spécial : "**la barre de synchronisation**".

- La barre de synchronisation permet de représenter graphiquement des points de synchronisation.
- Les transitions automatiques qui partent d'une barre de synchronisation ont lieu en même temps.
- On ne franchit une barre de synchronisation qu'après réalisation de toutes les transitions qui s'y rattachent.



Evénement paramétré

UML permet aussi de paramétrer les événements, comme dans l'exemple suivant :



Echange de messages entre automates

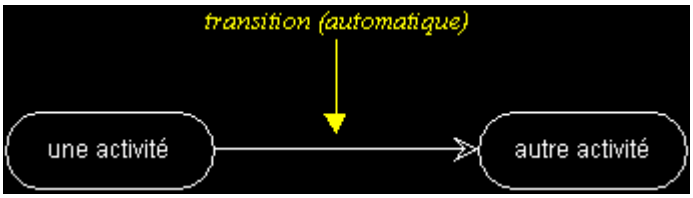
Il est aussi possible de représenter l'échange de messages entre automates dans un diagramme d'états-transitions. Cette notation particulière n'est pas présentée ici. Veuillez vous référer à "l'UML notation guide".

DIAGRAMME D'ACTIVITES

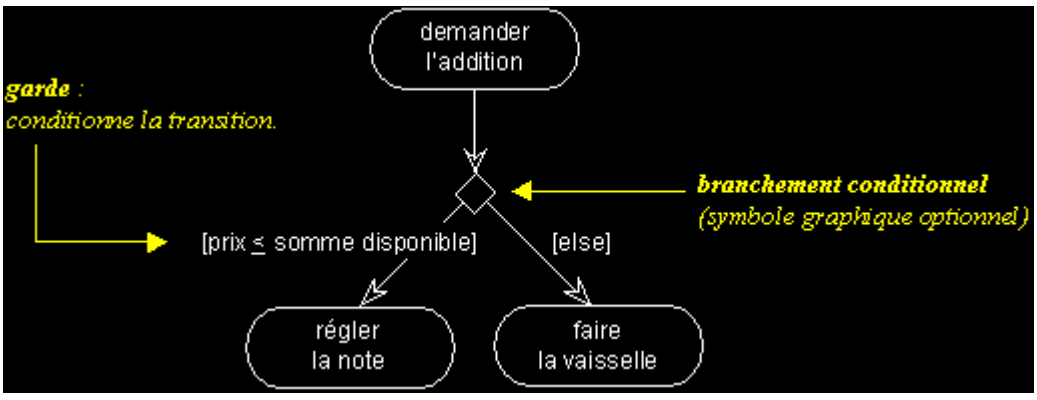
Diagramme d'activités : sémantique

- UML permet de représenter graphiquement le comportement d'une méthode ou le déroulement d'un cas d'utilisation, à l'aide de diagrammes d'activités (une variante desdiagrammes d'états-transitions).
- Une activité représente une exécution d'un mécanisme, un déroulement d'étapes séquentielles.
- Le passage d'une activité vers une autre est matérialisé par une transition.
- Les transitions sont déclenchées par la fin d'une activité et provoquent le début immédiat d'une autre (elles sont automatiques).
- En théorie, tous les mécanismes dynamiques pourraient être décrits par un diagramme d'activités, mais seuls les mécanismes complexes ou intéressants méritent d'être représentés.

activités et transition, notation :



Pour représenter des **transitions conditionnelles**, utilisez des gardes (expressions booléennes exprimées en langage naturel), comme dans l'exemple suivant :



Synchronisation

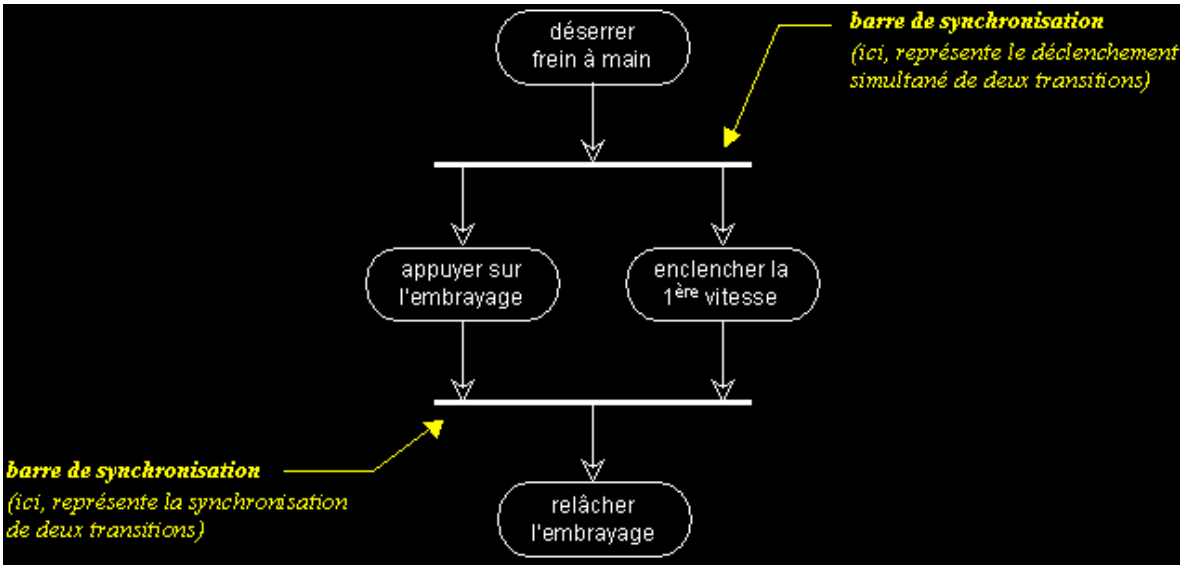
Il est possible de synchroniser les transitions à l'aide des "**barres de synchronisation**" (comme

dans les diagrammes d'états-transitions).

Une barre de synchronisation permet d'ouvrir et de fermer des branches parallèles au sein d'un flot d'exécution :

- Les transitions qui partent d'une barre de synchronisation ont lieu en même temps.
- On ne franchit une barre de synchronisation qu'après réalisation de toutes les transitions qui s'y rattachent.

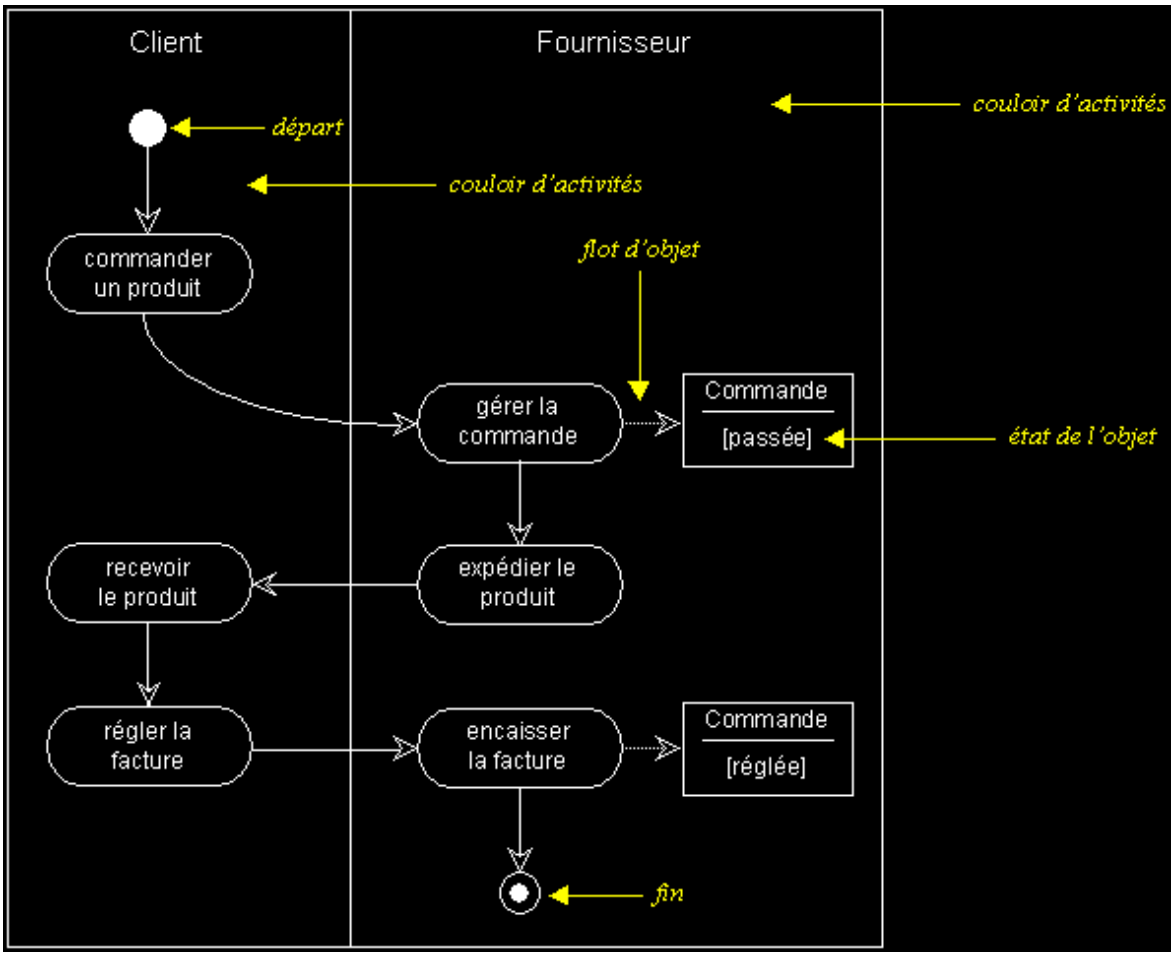
L'exemple suivant illustre l'utilisation des barres de synchronisation :



Couloirs d'activités

Afin d'organiser un diagramme d'activités selon les différents responsables des actions représentées, il est possible de définir des "couloirs d'activités".

Il est même possible d'identifier les objets principaux, qui sont manipulés d'activités en activités et de visualiser leur changement d'état.


Client
Fournisseur
couloir d'activités
départ
couloir d'activités
commander un produit
flot d'objet
gérer la commande
Commande
[passée]
état de l'objet
recevoir le produit
expédier le produit
régler la facture
encaisser la facture
Commande
[réglée]
fin