

UML, le langage de modélisation objet unifié

par Laurent Piechocki

Date de publication : 22/10/07

Dernière mise à jour : 14/09/09

Né de la fusion des méthodes objet dominantes (OMT, Booch et OOSE), puis normalisé par l'OMG en 1997, UML est rapidement devenu un standard incontournable. UML n'est pas à l'origine des concepts objet, mais il en donne une définition plus formelle et apporte la dimension méthodologique qui faisait défaut à l'approche objet.

Le but de cette présentation n'est pas de faire l'apologie d'UML, ni de restreindre UML à sa notation graphique, car le véritable intérêt d'UML est ailleurs !

En effet, maîtriser la notation graphique d'UML n'est pas une fin en soi. Ce qui est primordial, c'est d'utiliser les concepts objet à bon escient et d'appliquer la démarche d'analyse correspondante.

Cette présentation a donc pour objectif, d'une part, de montrer en quoi l'approche objet et UML constituent un "plus" et d'autre part, d'exposer comment utiliser UML dans la pratique, c'est-à-dire comment intégrer UML dans un processus de développement et comment modéliser avec UML.

Avertissement :

Les textes qui composent la présentation sont (volontairement) très synthétiques, à la manière de transparents qu'on projette au cours d'une formation.

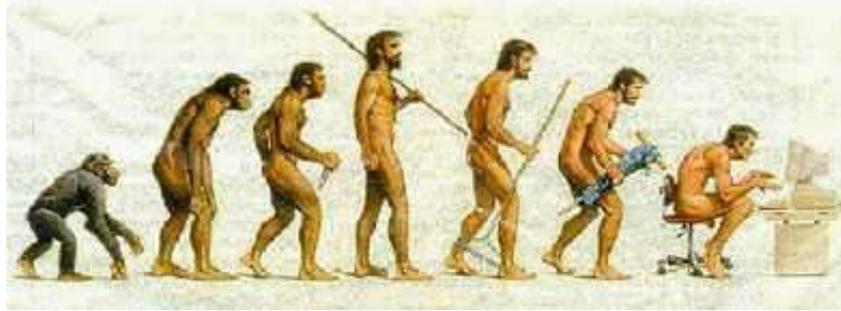
Il faut donc savoir lire entre les lignes, car il ne s'agit là que d'un "tour d'horizon". Cette présentation ne se substitue donc ni aux formations plus "académiques", ni aux ouvrages de référence.

I - PRESENTATION D'UML.....	4
I-A - Un peu d'Histoire.....	4
I-A-1 - Approche fonctionnelle vs. approche objet.....	4
I-A-1-a - La découpe fonctionnelle d'un problème informatique : une approche intuitive.....	4
I-A-1-b - Le "plus" de l'approche fonctionnelle : la factorisation des comportements.....	4
I-A-1-c - Le revers de la médaille : maintenance complexe en cas d'évolution.....	5
I-A-1-d - La séparation des données et des traitements : le piège !.....	5
I-A-1-e - 1ère amélioration : rassembler les valeurs qui caractérisent un type, dans le type.....	7
I-A-1-f - 2ème amélioration : centraliser les traitements associés à un type, auprès du type.....	7
I-A-1-g - Récapitulons.....	8
I-A-1-h - Objet ?.....	9
I-A-2 - Quels sont les autres concepts importants de l'approche objet ?.....	9
I-A-2-a - Encapsulation.....	9
I-A-2-b - Héritage (et polymorphisme).....	9
I-A-2-c - Agrégation.....	10
I-A-2-d - Résumé sur les concepts fondateurs de l'approche objet.....	11
I-A-2-e - L'approche objet, hier et aujourd'hui.....	11
I-A-2-f - L'approche objet : une solution parfaite ?.....	12
I-A-2-g - Quels sont les remèdes aux inconvénients de l'approche objet ?.....	12
I-B - Les méthodes objet et la genèse d'UML.....	12
I-B-1 - Méthodes ?.....	12
I-B-2 - A quoi sert UML ?.....	13
I-C - Avantages et inconvénients d'UML.....	14
I-C-1 - Les points forts d'UML.....	14
I-C-2 - Les points faibles d'UML.....	14
II - MODELISER AVEC UML.....	15
II-A - Qu'est-ce qu'un modèle ?.....	15
II-B - Comment modéliser avec UML ?.....	16
II-B-1 - Une démarche itérative et incrémentale ?.....	16
II-B-2 - Une démarche pilotée par les besoins des utilisateurs ?.....	16
II-B-3 - Une démarche centrée sur l'architecture ?.....	16
II-B-4 - Définir une architecture avec UML (détail de la "vue 4+1").....	17
II-B-5 - Résumons la démarche.....	18
II-B-6 - Elaboration plutôt que transformation.....	19
II-B-7 - Détail des différents niveaux d'abstraction (phases du macro-processus).....	19
II-B-8 - Activités des micro-processus d'analyse (niveau d'abstraction constant).....	19
II-B-9 - Synthèse de la démarche.....	20
II-B-10 - Les diagrammes UML.....	20
II-B-10-a - Comment "régérer" un modèle avec UML ?.....	21
II-B-10-b - Quelques caractéristiques des diagrammes UML.....	21
II-B-10-c - Les différents types de diagrammes UML.....	21
II-C - Les vues statiques d'UML.....	22
II-C-1 - LES PAQUETAGES.....	22
II-C-1-a - Paquetages (packages).....	22
II-C-1-b - Paquetages : relations entre paquetages.....	22
II-C-1-c - Paquetages : interfaces.....	23
II-C-1-d - Paquetages : stéréotypes.....	23
II-C-2 - LA COLLABORATION.....	24
II-C-2-a - Symbole de modélisation "collaboration".....	24
II-C-3 - INSTANCES ET DIAGRAMME D'OBJETS.....	25
II-C-3-a - Exemples d'instances.....	25
II-C-3-b - Objets composites.....	26
II-C-3-c - Diagramme d'objets.....	26
II-C-4 - LES CLASSES.....	27
II-C-4-a - Classe : sémantique et notation.....	27
II-C-5 - DIAGRAMME DE CLASSES.....	28
II-C-5-a - Diagramme de classes : sémantique.....	28
II-C-5-b - Associations entre classes.....	29

II-C-5-c - Documentation d'une association et types d'associations.....	29
II-C-5-d - Héritage.....	32
II-C-5-e - Agrégation.....	33
II-C-5-f - Composition.....	34
II-C-5-g - Agrégation et composition : rappel.....	34
II-C-5-h - Interfaces.....	34
II-C-5-i - Association dérivée.....	35
II-C-5-j - Contrainte sur une association.....	36
II-C-5-k - OCL.....	37
II-C-5-l - Stéréotypes.....	39
II-C-6 - DIAGRAMMES DE COMPOSANTS ET DE DEPLOIEMENT.....	40
II-C-6-a - Diagramme de composants.....	40
II-C-6-b - Diagramme de déploiement.....	41
II-D - Les vues dynamiques d'UML.....	42
II-D-1 - LES CAS D'UTILISATION.....	43
II-D-1-a - La conceptualisation : rappel.....	43
II-D-1-b - Cas d'utilisation (use cases).....	43
II-D-1-c - Eléments de base des cas d'utilisation.....	44
II-D-1-d - Exemples.....	45
II-D-2 - COLLABORATION ET MESSAGES.....	46
II-D-2-a - Synchronisation des messages.....	47
II-D-2-b - Objets actifs (threads).....	48
II-D-3 - DIAGRAMME DE SEQUENCE.....	49
II-D-3-a - Diagramme de séquence : sémantique.....	49
II-D-3-b - Types de messages.....	50
II-D-3-c - Activation d'un objet.....	50
II-D-3-d - Exemple complet.....	51
II-D-4 - DIAGRAMME D'ETATS-TRANSITIONS.....	52
II-D-4-a - Diagramme d'états-transitions : sémantique.....	52
II-D-4-b - Super-Etat, historique et souches.....	53
II-D-4-c - Actions dans un état.....	55
II-D-4-d - Etats concurrents et barre de synchronisation.....	55
II-D-4-e - Evénement paramétré.....	57
II-D-4-f - Echange de messages entre automates.....	57
II-D-5 - DIAGRAMME D'ACTIVITES.....	57
II-D-5-a - Diagramme d'activités : sémantique.....	57
II-D-5-b - Synchronisation.....	58
II-D-5-c - Couloirs d'activités.....	58
II-E - Conclusion.....	59
II-E-1 - Programmer objet ?.....	59
II-E-2 - Utiliser UML ?.....	59

I - PRESENTATION D'UML

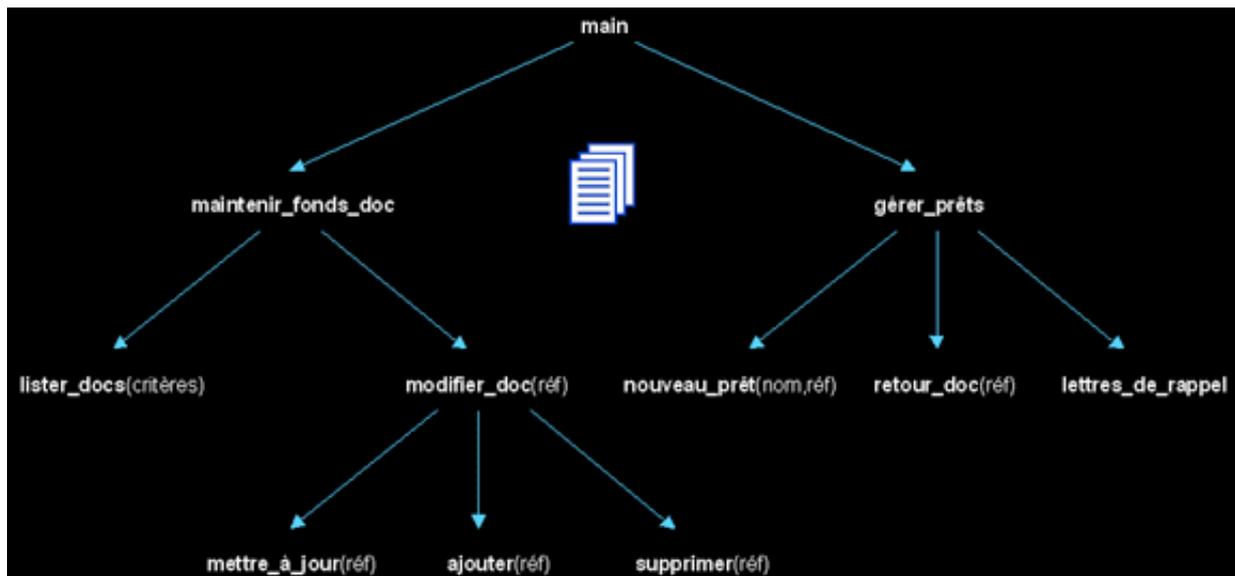
I-A - Un peu d'Histoire...



I-A-1 - Approche fonctionnelle vs. approche objet

I-A-1-a - La découpe fonctionnelle d'un problème informatique : une approche intuitive

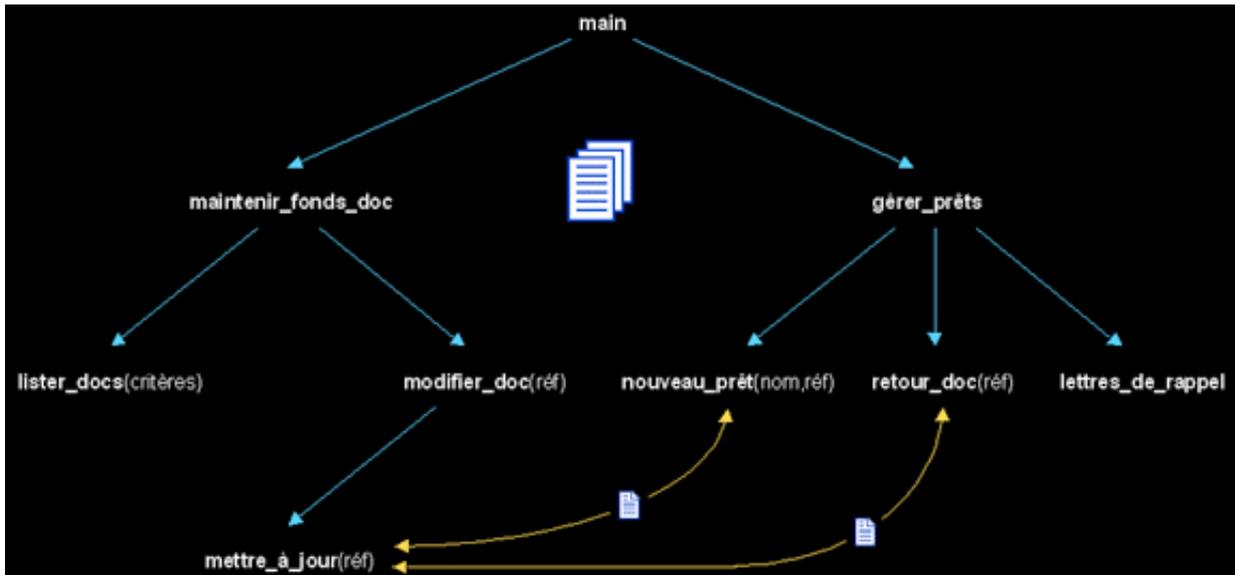
Exemple de découpe fonctionnelle d'un logiciel dédié à la gestion d'une bibliothèque :



Le logiciel est composé d'une hiérarchie de fonctions, qui ensemble, fournissent les services désirés, ainsi que de données qui représentent les éléments manipulés (livres, etc.). Logique, cohérent et intuitif.

I-A-1-b - Le "plus" de l'approche fonctionnelle : la factorisation des comportements

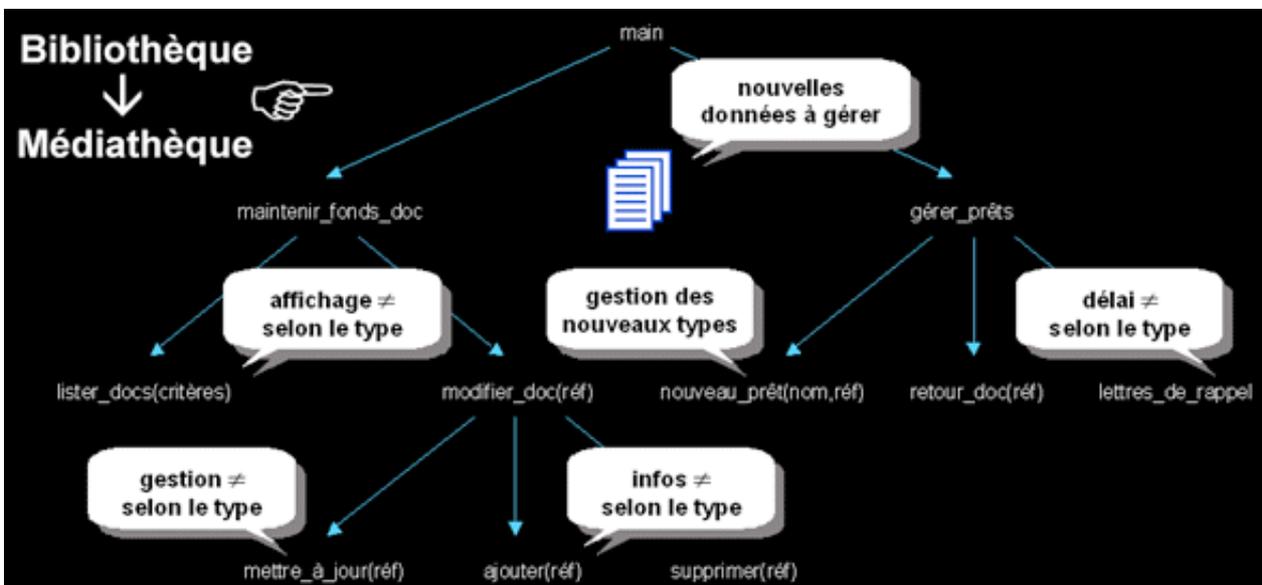
Une découpe fonctionnelle "intelligente" consiste à factoriser certains comportements communs du logiciel. En d'autres termes : pour réaliser une fonction du logiciel, on peut utiliser un ensemble d'autres fonctions, déjà disponibles, pour peu qu'on rende ces dernières un tant soit peu génériques. Génial !



I-A-1-c - Le revers de la médaille : maintenance complexe en cas d'évolution

Factoriser les comportements n'a malheureusement pas que des avantages. Les fonctions sont devenues interdépendantes : une simple mise à jour du logiciel à un point donné, peut impacter en cascade une multitude d'autres fonctions. On peut minorer cet impact, pour peu qu'on utilise des fonctions plus génériques et des structures de données ouvertes. Mais respecter ces contraintes rend l'écriture du logiciel et sa maintenance plus complexe.

En cas d'évolution majeure du logiciel (passage de la gestion d'une bibliothèque à celle d'une médiathèque par exemple), le scénario est encore pire. Même si la structure générale du logiciel reste valide, la multiplication des points de maintenance, engendrée par le chaînage des fonctions, rend l'adaptation très laborieuse. Le logiciel doit être retouché dans sa globalité :



I-A-1-d - La séparation des données et des traitements : le piège !

Examinons le problème de l'évolution de code fonctionnel plus en détail...

Faire évoluer une application de gestion de bibliothèque pour gérer une médiathèque, afin de prendre en compte de nouveaux types d'ouvrages (cassettes vidéo, CD-ROM, etc...), nécessite :

- de faire évoluer les structures de données qui sont manipulées par les fonctions,
- d'adapter les traitements, qui ne manipulaient à l'origine qu'un seul type de document (des livres).

Il faudra donc modifier toutes les portions de code qui utilisent la base documentaire, pour gérer les données et les actions propres aux différents types de documents.

Il faudra par exemple modifier la fonction qui réalise l'édition des "lettres de rappel" (une lettre de rappel est une mise en demeure, qu'on envoie automatiquement aux personnes qui tardent à rendre un ouvrage emprunté). Si l'on désire que le délai avant rappel varie selon le type de document emprunté, il faut prévoir une règle de calcul pour chaque type de document.

En fait, c'est la quasi-totalité de l'application qui devra être adaptée, pour gérer les nouvelles données et réaliser les traitements correspondants. Et cela, à chaque fois qu'on décidera de gérer un nouveau type de document !

```

struct Document
{
    char nom_doc[50];
    Type_doc type;
    Bool est_emprunte;
    char emprunteur[50];
    struct tm date_emprunt;
} DOC[MAX_DOCS];

void lettres_de_rappel()
{
    /* ... */
    for (i = 0; i < NB_DOCS; i++)
    {
        if (DOC[i].est_emprunte)
        {
            switch(DOC[i].type)
            {
                case LIVRE:
                    delai_avant_rappel = 20;
                    break;
                case CASSETTE_VIDEO:
                    delai_avant_rappel = 7;
                    break;
                case CD_ROM:
                    delai_avant_rappel = 5;
                    break;
            }
        }
    }
    /* ... */
}

void mettre_a_jour(int ref)
{
    /* ... */
    switch(DOC[ref].type)
    {
        case LIVRE:
            maj_livre(DOC[ref]);
            break;
        case CASSETTE_VIDEO:
            maj_k7(DOC[ref]);
            break;
        case CD_ROM:
            maj_cd(DOC[ref]);
            break;
    }
    /* ... */
}
    
```

I-A-1-e - 1ère amélioration : rassembler les valeurs qui caractérisent un type, dans le type

Une solution relativement élégante à la multiplication des branches conditionnelles et des redondances dans le code (conséquence logique d'une trop grande ouverture des données), consiste tout simplement à *centraliser dans les structures de données, les valeurs qui leurs sont propres* :

```

struct Document
{
    char nom_doc[50];
    Type_doc type;
    Bool est_emprunte;
    char emprunteur[50];
    struct tm date_emprunt;
    int delai_avant_rappel;
} DOC[MAX_DOCS];

void lettres_de_rappel()
{
    /* ... */
    for (i = 0; i < NB_DOCS; i++)
    {
        if (DOC[i].est_emprunte) /* SI LE DOC EST EMPRUNTE */
        {
            /* IMPRIME UNE LETTRE SI SON
            DELAI DE RAPPEL EST DEPASSE */

            if (date() >= (DOC[i].date_emprunt + DOC[i].delai_avant_rappel))
                imprimer_rappel(DOC[i]);
        }
    }
}
    
```

Quoi de plus logique ? En effet, le "délai avant édition d'une lettre de rappel" est bien une caractéristique propre à tous les ouvrages gérés par notre application.

Mais cette solution n'est pas encore optimale !

I-A-1-f - 2ème amélioration : centraliser les traitements associés à un type, auprès du type

Pourquoi ne pas aussi rassembler dans une même unité physique les types de données et tous les traitements associés ?

Que se passerait-il par exemple si l'on centralisait dans un même fichier, la structure de données qui décrit les documents et la fonction de calcul du délai avant rappel ? Cela nous permettrait de retrouver en un clin d'oeil le bout de code qui est chargé de calculer le délai avant rappel d'un document, puisqu'il se trouve au plus près de la structure de données concernée.

Ainsi, si notre médiathèque devait gérer un nouveau type d'ouvrage, il suffirait de modifier une seule fonction (qu'on sait retrouver instamment), pour assurer la prise en compte de ce nouveau type de document dans le calcul du délai avant rappel. Plus besoin de fouiller partout dans le code...

```

struct Document
{
    char nom_doc[50];
    Type_doc type;
    Bool est_emprunte;
    char emprunteur[50];
    struct tm date_emprunt;
    int delai_avant_rappel;
} DOC[MAX_DOCS];
    
```

```
int calculer_delai_rappel(Type_doc type)
{
    switch(type)
    {
        case LIVRE:
            return 20;
        case CASSETTE_VIDEO:
            return 7;
        case CD_ROM:
            return 5;
        /* autres "case" bienvenus ici ! */
    }
}
```

Ecrit en ces termes, notre logiciel sera plus facile à maintenir et bien plus lisible. Le stockage et le calcul du délai avant rappel des documents, est désormais assuré par une seule et unique unité physique (quelques lignes de code, rapidement identifiables).

Pour accéder à la caractéristique "délai avant rappel" d'un document, il suffit de récupérer la valeur correspondante parmi les champs qui décrivent le document. Pour assurer la prise en compte d'un nouveau type de document dans le calcul du délai avant rappel, il suffit de modifier une seule fonction, située au même endroit que la structure de données qui décrit les documents :

```
void ajouter_document(int ref)
{
    DOC[ref].est_emprunte = FAUX;
    DOC[ref].nom_doc = saisir_nom();
    DOC[ref].type = saisir_type();
    DOC[ref].delai_avant_rappel = calculer_delai_rappel(DOC[ref].type);
}

void afficher_document(int ref)
{
    printf("Nom du document: %s\n", DOC[ref].nom_doc);

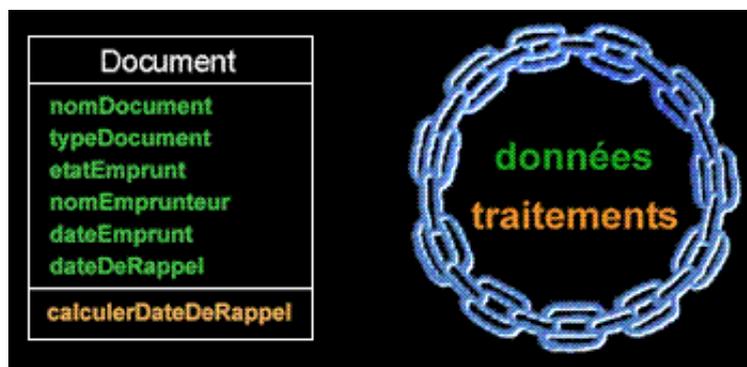
    /* ... */

    printf("Delai avant rappel: %d jours\n", DOC[ref].delai_avant_rappel);

    /* ... */
}
```

I-A-1-g - Récapitulons...

En résumé : centraliser les données d'un type et les traitements associés, dans une même unité physique, permet de limiter les points de maintenance dans le code et facilite l'accès à l'information en cas d'évolution du logiciel :

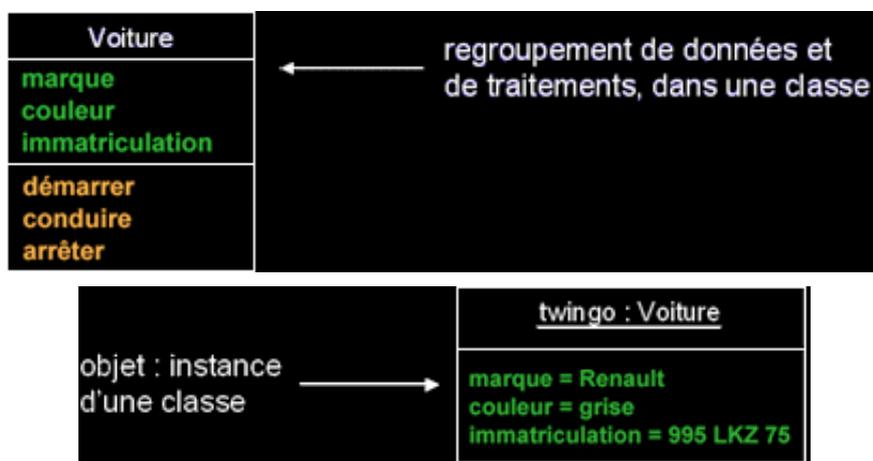


I-A-1-h - Objet ?

Les modifications que nous avons apporté à notre logiciel de gestion de médiathèque, nous ont amené à transformer ce qui était à l'origine une structure de données, manipulée par des fonctions, en une entité autonome, qui regroupe un ensemble de propriétés cohérentes et de traitements associés. Une telle entité s'appelle... un objet et constitue le concept fondateur de l'approche du même nom !

- Un objet est une entité aux frontières précises qui possède une identité (un nom).
- Un ensemble d'attributs caractérise l'état de l'objet.
- Un ensemble d'opérations (méthodes) en définissent le comportement.
- Un objet est une instance de classe (une occurrence d'un type abstrait).
- Une classe est un type de données abstrait, caractérisé par des propriétés (attributs et méthodes) communes à des objets et permettant de créer des objets possédant ces propriétés.

Exemple :



I-A-2 - Quels sont les autres concepts importants de l'approche objet ?

I-A-2-a - Encapsulation

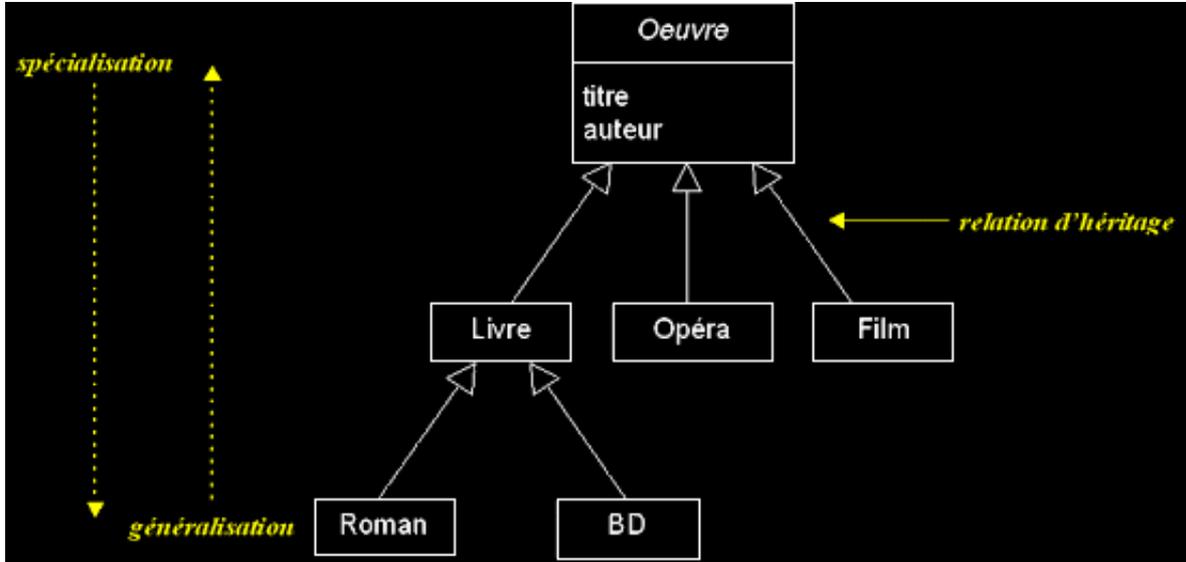
- Consiste à masquer les détails d'implémentation d'un objet, en définissant une interface.
- L'interface est la vue externe d'un objet, elle définit les services accessibles (offerts) aux utilisateurs de l'objet.
- L'encapsulation facilite l'évolution d'une application car elle stabilise l'utilisation des objets : on peut modifier l'implémentation des attributs d'un objet sans modifier son interface.
- L'encapsulation garantit l'intégrité des données, car elle permet d'interdire l'accès direct aux attributs des objets (utilisation d'accessieurs).

I-A-2-b - Héritage (et polymorphisme)

- L'héritage est un mécanisme de transmission des propriétés d'une classe (ses attributs et méthodes) vers une sous-classe.
- Une classe peut être spécialisée en d'autres classes, afin d'y ajouter des caractéristiques spécifiques ou d'en adapter certaines.
- Plusieurs classes peuvent être généralisées en une classe qui les factorise, afin de regrouper les caractéristiques communes d'un ensemble de classes.
- La spécialisation et la généralisation permettent de construire des hiérarchies de classes. L'héritage peut être simple ou multiple.
- L'héritage évite la duplication et encourage la réutilisation.

- Le polymorphisme représente la faculté d'une méthode à pouvoir s'appliquer à des objets de classes différentes.
- Le polymorphisme augmente la généricité du code.

Exemple d'une hiérarchie de classes :

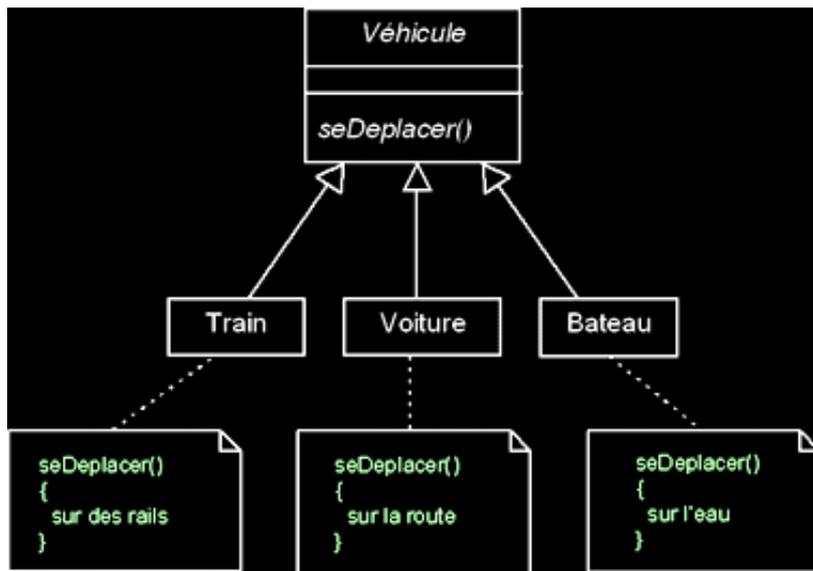


Polymorphisme, exemple :

```

Vehicule convoi[3] = {
    Train("TGV"),
    Voiture("twingo"),
    Bateau("Titanic")
};

for (int i = 0; i < 3; i++)
{
    convoi[i].seDeplacer();
}
    
```



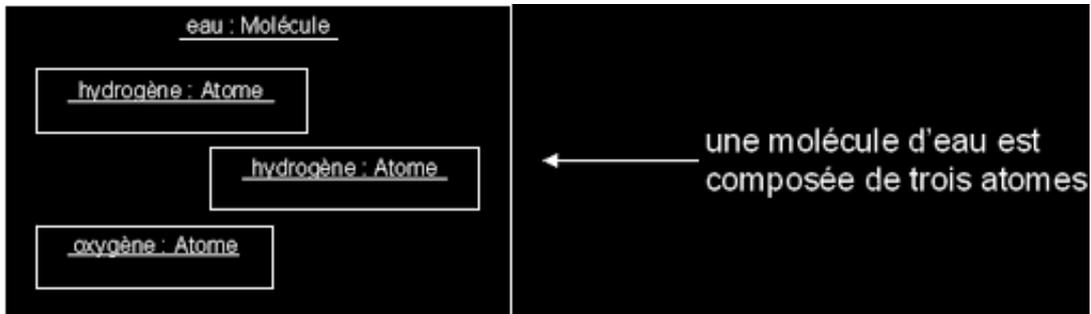
I-A-2-c - Agrégation

- Il s'agit d'une relation entre deux classes, spécifiant que les objets d'une classe sont des composants de l'autre classe.

Une relation d'agrégation permet donc de définir des objets composés d'autres objets.

- L'agrégation permet d'assembler des objets de base, afin de construire des objets plus complexes.

Agrégation, exemple :



I-A-2-d - Résumé sur les concepts fondateurs de l'approche objet

- la notion d'objet et de classe (d'objets)
- l'encapsulation (les interfaces des objets)
- l'héritage (les hiérarchies d'objets)
- l'agrégation (la construction d'objets à l'aide d'objets)

i *Remarque : Les langages orientés objet fournissent de nombreux autres mécanismes qui affinent ces concepts de base, favorisent la généricité du code ou améliorent sa robustesse.*

I-A-2-e - L'approche objet, hier et aujourd'hui

- Les concepts objet sont stables et éprouvés (issus du terrain)
 - Simula, 1er langage de programmation à implémenter le concept de type abstrait (à l'aide de classes), date de 1967 !
 - En 1976 déjà, Smalltalk implémente les concepts fondateurs de l'approche objet (encapsulation, agrégation, héritage) à l'aide de :
 - classes
 - associations entre classes
 - hiérarchies de classes
 - messages entre objets
 - Le 1er compilateur C++ date de 1980, et C++ est normalisé par l'ANSI.
 - De nombreux langages orientés objets académiques ont étayés les concepts objets : Eiffel, Objective C, Loops...
- Les concepts objet sont anciens, mais ils n'ont jamais été autant d'actualité
 - L'approche fonctionnelle n'est pas adaptée au développement d'applications qui évoluent sans cesse et dont la complexité croît continuellement.
 - L'approche objet a été inventée pour faciliter l'évolution d'applications complexes.
- De nos jours, les outils orientés objet sont fiables et performants
 - Les compilateurs C++ produisent un code robuste et optimisé.
 - De très nombreux outils facilitent le développement d'applications C++ :
 - générateurs d'IHM (Ilog Views, TeleUse...)
 - bibliothèques (STL, USL, Rogue Wave, MFC...)
 - environnements de développement intégrés (Developer Studio, Sniff+...)
 - outils de qualimétrie et de tests (Cantata++, Insure++, Logiscope...)
 - bases de données orientées objet (O2, ObjectStore, Versant...)

I-A-2-f - L'approche objet : une solution parfaite ?

En résumé, l'approche objet c'est :

- Un ensemble de concepts stables, éprouvés et normalisés.
- Une solution destinée à faciliter l'évolution d'applications complexes.
- Une panoplie d'outils et de langages performants pour le développement

Oui, MAIS..

Malgré les apparences, il est plus naturel pour nos esprits cartésiens, de décomposer un problème informatique sous forme d'une hiérarchie de fonctions atomiques et de données, qu'en terme d'objets et d'interaction entre ces objets. De plus, le vocabulaire précis est un facteur d'échec important dans la mise en oeuvre d'une approche objet.

- L'approche objet est moins intuitive que l'approche fonctionnelle !
 - Quels moyens utiliser pour faciliter l'analyse objet ?
 - Quels critères identifient une conception objet pertinente ?
 - Comment comparer deux solutions de découpe objet d'un système ?
- L'application des concepts objets nécessite une grande rigueur !
 - Le vocabulaire est précis (risques d'ambiguïtés, d'incompréhensions).
 - Comment décrire la structure objet d'un système de manière pertinente ?

I-A-2-g - Quels sont les remèdes aux inconvénients de l'approche objet ?

- Un langage pour exprimer les concepts objet qu'on utilise, afin de pouvoir :
 - Représenter des concepts abstraits (graphiquement par exemple).
 - Limiter les ambiguïtés (parler un langage commun).
 - Faciliter l'analyse (simplifier la comparaison et l'évaluation de solutions).
- Une démarche d'analyse et de conception objet, pour :
 - Ne pas effectuer une analyse fonctionnelle et se contenter d'une implémentation objet, mais penser objet dès le départ.
 - Définir les vues qui permettent de couvrir tous les aspects d'un système, avec des concepts objets.

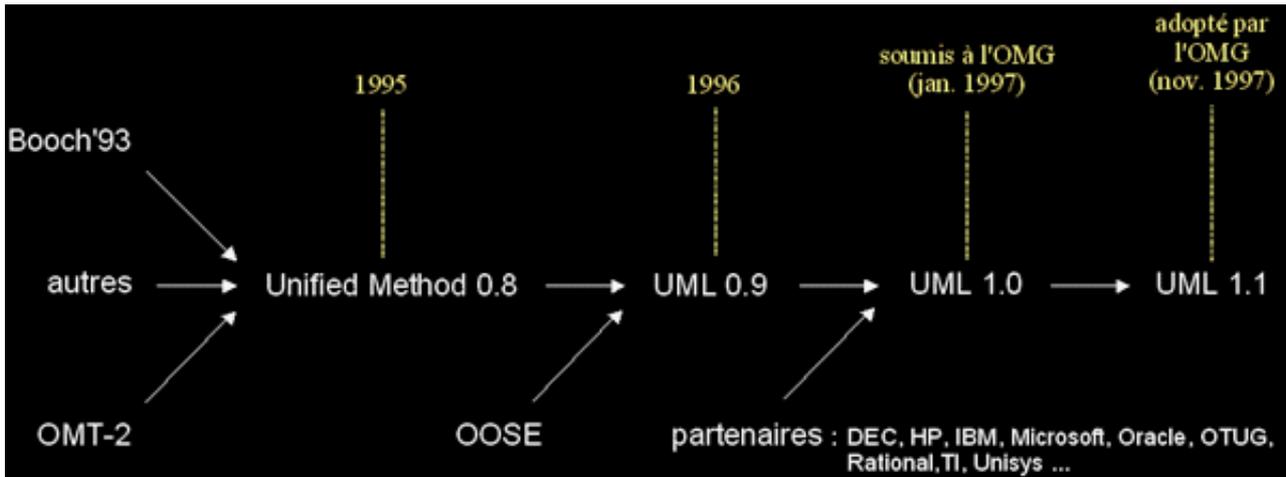
 *Bref : il nous faut un outil qui apporte une dimension méthodologique à l'approche objet, afin de mieux maîtriser sa richesse et sa complexité.*

I-B - Les méthodes objet et la genèse d'UML

I-B-1 - Méthodes ?

- Les premières méthodes d'analyse (années 70)
 - Découpe cartésienne (fonctionnelle et hiérarchique) d'un système.
- L'approche systémique (années 80)
 - Modélisation des données + modélisation des traitements (Merise, Axial, IE...).
- L'émergence des méthodes objet (1990-1995)
 - Prise de conscience de l'importance d'une méthode spécifiquement objet : comment structurer un système sans centrer l'analyse uniquement sur les données ou uniquement sur les traitements (mais sur les deux) ?
 - Plus de 50 méthodes objet sont apparues durant cette période (Booch, Classe-Relation, Fusion, HOOD, OMT, OOA, OOD, OOM, OOSE...) !
 - Aucune méthode ne s'est réellement imposée.
- Les premiers consensus (1995)
 - **OMT** (James Rumbaugh) : vues statiques, dynamiques et fonctionnelles d'un système

- Issue du centre de R&D de General Electric.
- Notation graphique riche et lisible.
- **OOD** (Grady Booch) : vues logiques et physiques du système
 - Définie pour le DOD, afin de rationaliser de développement d'applications ADA, puis C++.
 - Ne couvre pas la phase d'analyse dans ses 1ères versions (préconise SADT).
 - Introduit le concept de package (élément d'organisation des modèles)
- **OOSE** (Ivar Jacobson) : couvre tout le cycle de développement
 - Issue d'un centre de développement d'Ericsson, en Suède.
 - La méthodologie repose sur l'analyse des besoins des utilisateurs.
- L'unification et la normalisation des méthodes (1995-1997)
 - UML (Unified Modeling Language), la fusion et synthèse des méthodes dominantes :



- UML aujourd'hui : un standard incontournable
 - UML est le résultat d'un large consensus (industriels, méthodologistes...).
 - UML est le fruit d'un travail d'experts reconnus.
 - UML est issu du terrain.
 - UML est riche (il couvre toutes les phases d'un cycle de développement).
 - UML est ouvert (il est indépendant du domaine d'application et des langages d'implémentation).
 - Après l'unification et la standardisation, bientôt l'industrialisation d'UML : les outils qui supportent UML se multiplient (GDPro, ObjectTeam, Objecteering, OpenTool, Rational Rose, Rhapsody, STP, Visio, Visual Modeler, WithClass...).
 - XML (format d'échange standard de modèles UML).
- UML évolue mais reste stable !
 - L'OMG RTF (nombreux acteurs industriels) centralise et normalise les évolutions d'UML au niveau international.
 - Les groupes d'utilisateurs UML favorisent le partage des expériences.
 - De version en version, UML gagne en maturité et précision, tout en restant stable.
 - UML inclut des mécanismes standards d'auto-extension.
 - La description du métamodèle d'UML est standardisée (OMG-MOF).

 >>> UML n'est pas une mode, c'est un investissement fiable !

I-B-2 - A quoi sert UML ?

UML n'est pas une méthode ou un processus !

- Si l'on parle de méthode objet pour UML, c'est par abus de langage !
- Ce constat vaut aussi pour OMT ou d'autres techniques / langages de modélisation.
- Une méthode propose aussi un processus, qui régit notamment l'enchaînement des activités de production d'une entreprise.

- UML a été pensé pour permettre de modéliser les activités de l'entreprise, pas pour les régir (ce n'est pas CMM ou SPICE).
- Un processus de développement logiciel universel est une utopie :
 - Impossible de prendre en compte toutes les organisations et cultures d'entreprises.
 - Un processus est adapté (donc très lié) au domaine d'activité de l'entreprise.
 - Même si un processus constitue un cadre général, il faut l'adapter de manière précise au contexte de l'entreprise.

UML est un langage pseudo-formel

- UML est fondé sur un métamodèle, qui définit :
 - les éléments de modélisation (les concepts manipulés par le langage),
 - la sémantique de ces éléments (leur définition et le sens de leur utilisation).
- Un métamodèle est une description très formelle de tous les concepts d'un langage. Il limite les ambiguïtés et encourage la construction d'outils.
- Le métamodèle d'UML permet de classer les concepts du langage (selon leur niveau d'abstraction ou domaine d'application) et expose sa structure.
- Le métamodèle UML est lui-même décrit par un méta-métamodèle (OMG-MOF).
- UML propose aussi une notation, qui permet de représenter graphiquement les éléments de modélisation du métamodèle.
- Cette notation graphique est le support du langage UML.

UML cadre l'analyse objet, en offrant :

- différentes vues (perspectives) complémentaires d'un système, qui guident l'utilisation des concept objets,
- plusieurs niveaux d'abstraction, qui permettent de mieux contrôler la complexité dans l'expression des solutions objets.

UML est un support de communication

- Sa notation graphique permet d'exprimer visuellement une solution objet.
- L'aspect formel de sa notation limite les ambiguïtés et les incompréhensions.
- Son aspect visuel facilite la comparaison et l'évaluation de solutions.
- Son indépendance (par rapport aux langages d'implémentation, domaine d'application, processus...) en font un langage universel.

I-C - Avantages et inconvénients d'UML

I-C-1 - Les points forts d'UML

UML est un langage formel et normalisé

- gain de précision
- gage de stabilité
- encourage l'utilisation d'outils

UML est un support de communication performant

- Il cadre l'analyse.
- Il facilite la compréhension de représentations abstraites complexes.
- Son caractère polyvalent et sa souplesse en font un langage universel.

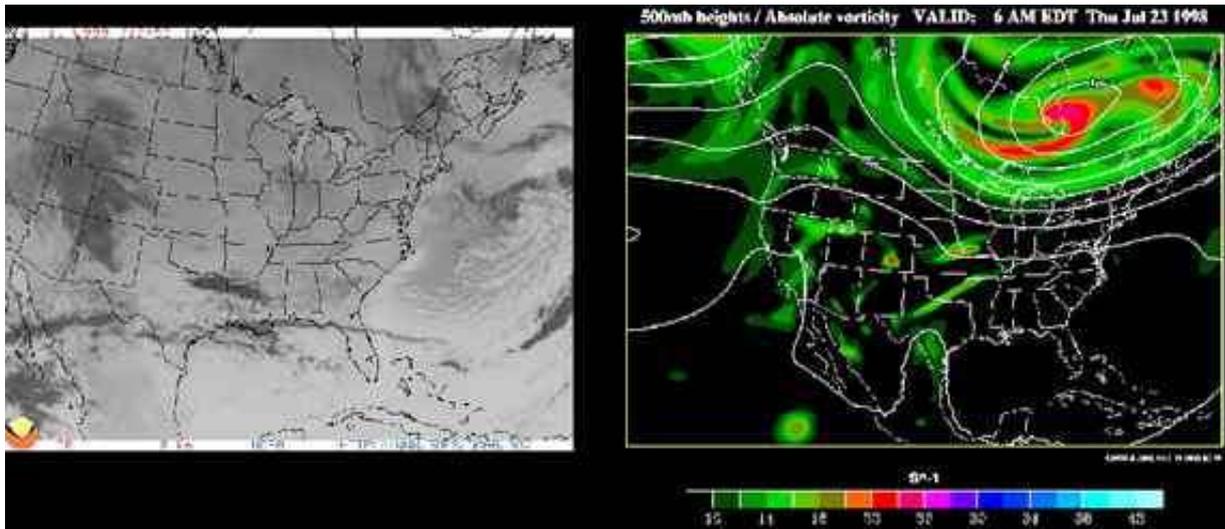
I-C-2 - Les points faibles d'UML

- La mise en pratique d'UML nécessite un apprentissage et passe par une période d'adaptation. Même si l'Espéranto est une utopie, la nécessité de s'accorder sur des modes d'expression communs est vitale en informatique. UML n'est pas à l'origine des concepts objets, mais en constitue une étape majeure, car il unifie les différentes approches et en donne une définition plus formelle.

- Le processus (non couvert par UML) est une autre clé de la réussite d'un projet. Or, l'intégration d'UML dans un processus n'est pas triviale et améliorer un processus est un tâche complexe et longue

Les auteurs d'UML sont tout à fait conscients de l'importance du processus, mais l'acceptabilité industrielle de la modélisation objet passe d'abord par la disponibilité d'un langage d'analyse objet performant et standard.

II - MODELISER AVEC UML



II-A - Qu'est-ce qu'un modèle ?

- Un modèle est une abstraction de la réalité
L'abstraction est un des piliers de l'approche objet.
 - Il s'agit d'un processus qui consiste à identifier les caractéristiques intéressantes d'une entité, en vue d'une utilisation précise.
 - L'abstraction désigne aussi le résultat de ce processus, c'est-à-dire l'ensemble des caractéristiques essentielles d'une entité, retenues par un observateur.
- Un modèle est une vue subjective mais pertinente de la réalité
 - Un modèle définit une frontière entre la réalité et la perspective de l'observateur. Ce n'est pas "la réalité", mais une vue très subjective de la réalité.
 - Bien qu'un modèle ne représente pas une réalité absolue, un modèle reflète des aspects importants de la réalité, il en donne donc une vue juste et pertinente.
- Quelques exemples de modèles
 - Modèle météorologique :
à partir de données d'observation (satellite ...), permet de prévoir les conditions climatiques pour les jours à venir.
 - Modèle économique :
peut par exemple permettre de simuler l'évolution de cours boursiers en fonction d'hypothèses macro-économiques (évolution du chômage, taux de croissance...).
 - Modèle démographique :
définit la composition d'un panel d'une population et son comportement, dans le but de fiabiliser des études statistiques, d'augmenter l'impact de démarches commerciales, etc...
- Caractéristiques fondamentales des modèles
Le caractère abstrait d'un modèle doit notamment permettre :
 - de faciliter la compréhension du système étudié

- > Un modèle réduit la complexité du système étudié.
- de simuler le système étudié
- > Un modèle représente le système étudié et reproduit ses comportements.

Un modèle réduit (décompose) la réalité, dans le but de disposer d'éléments de travail exploitables par des moyens mathématiques ou informatiques :
modèle / réalité ~ digital / analogique

II-B - Comment modéliser avec UML ?

- **UML est un langage qui permet de représenter des modèles, mais il ne définit pas le processus d'élaboration des modèles !**
- Cependant, dans le cadre de la modélisation d'une application informatique, les auteurs d'UML préconisent d'utiliser une démarche :
 - itérative et incrémentale,
 - guidée par les besoins des utilisateurs du système,
 - centrée sur l'architecture logicielle.

D'après les auteurs d'UML, un processus de développement qui possède ces qualités devrait favoriser la réussite d'un projet.

II-B-1 - Une démarche itérative et incrémentale ?

- L'idée est simple : pour modéliser (comprendre et représenter) un système complexe, il vaut mieux s'y prendre en plusieurs fois, en affinant son analyse par étapes.
- Cette démarche devrait aussi s'appliquer au cycle de développement dans son ensemble, en favorisant le prototypage.

Le but est de mieux maîtriser la part d'inconnu et d'incertitudes qui caractérisent les systèmes complexes.

II-B-2 - Une démarche pilotée par les besoins des utilisateurs ?

Avec UML, ce sont les utilisateurs qui guident la définition des modèles :

- Le périmètre du système à modéliser est défini par les besoins des utilisateurs (les utilisateurs définissent ce que doit être le système).
- Le but du système à modéliser est de répondre aux besoins de ses utilisateurs (les utilisateurs sont les clients du système).

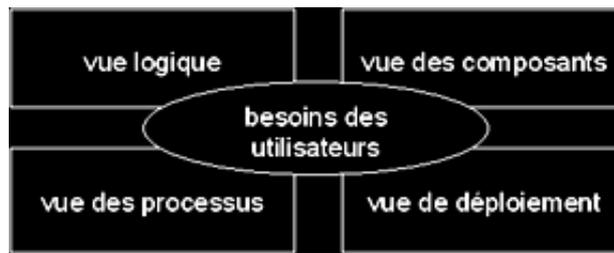
Les besoins des utilisateurs servent aussi de fil rouge, tout au long du cycle de développement (itératif et incrémental) :

- A chaque itération de la phase d'analyse, on clarifie, affine et valide les besoins des utilisateurs.
- A chaque itération de la phase de conception et de réalisation, on veille à la prise en compte des besoins des utilisateurs.
- A chaque itération de la phase de test, on vérifie que les besoins des utilisateurs sont satisfaits.

II-B-3 - Une démarche centrée sur l'architecture ?

- Une architecture adaptée est la clé de voûte du succès d'un développement.
Elle décrit des choix stratégiques qui déterminent en grande partie les qualités du logiciel (adaptabilité, performances, fiabilité...).
- Ph. Kruchten propose différentes perspectives, indépendantes et complémentaires, qui permettent de définir un modèle d'architecture (publication IEEE, 1995).

Cette vue ("4+1") a fortement inspiré UML :



II-B-4 - Définir une architecture avec UML (détail de la "vue 4+1")

La vue logique

- Cette vue de haut niveau se concentre sur l'abstraction et l'encapsulation, elle modélise les éléments et mécanismes principaux du système.
- Elle identifie les éléments du domaine, ainsi que les relations et interactions entre ces éléments :
 - les éléments du domaine sont liés au(x) métier(s) de l'entreprise,
 - ils sont indispensables à la mission du système,
 - ils gagnent à être réutilisés (ils représentent un savoir-faire).
- Cette vue organise aussi (selon des critères purement logiques), les éléments du domaine en "catégories" :
 - pour répartir les tâches dans les équipes,
 - regrouper ce qui peut être générique,
 - isoler ce qui est propre à une version donnée, etc...

La vue des composants

- **Cette vue de bas niveau (aussi appelée "vue de réalisation"), montre :**
 - L'allocation des éléments de modélisation dans des modules (fichiers sources, bibliothèques dynamiques, bases de données, exécutables, etc...).
 - En d'autres termes, cette vue identifie les modules qui réalisent (physiquement) les classes de la vue logique.
 - L'organisation des composants, c'est-à-dire la distribution du code en gestion de configuration, les dépendances entre les composants...
 - Les contraintes de développement (bibliothèques externes...).
 - La vue des composants montre aussi l'organisation des modules en "sous-systèmes", les interfaces des sous-systèmes et leurs dépendances (avec d'autres sous-systèmes ou modules).

La vue des processus

- Cette vue est très importante dans les environnements multitâches ; elle montre :
 - La décomposition du système en terme de processus (tâches).
 - Les interactions entre les processus (leur communication).
 - La synchronisation et la communication des activités parallèles (threads).

La vue de déploiement

- **Cette vue très importante dans les environnements distribués, décrit les ressources matérielles et la répartition du logiciel dans ces ressources :=**
 - La disposition et nature physique des matériels, ainsi que leurs performances.
 - L'implantation des modules principaux sur les noeuds du réseau.
 - Les exigences en terme de performances (temps de réponse, tolérance aux fautes et pannes...).

La vue des besoins des utilisateurs

- **Cette vue (dont le nom exact est "vue des cas d'utilisation"), guide toutes les autres.**
 - Dessiner le plan (l'architecture) d'un système informatique n'est pas suffisant, il faut le justifier !
 - Cette vue définit les besoins des clients du système et centre la définition de l'architecture du système sur la satisfaction (la réalisation) de ces besoins.

- A l'aide de scénarios et de cas d'utilisation, cette vue conduit à la définition d'un modèle d'architecture pertinent et cohérent.
- Cette vue est la "colle" qui unifie les quatre autres vues de l'architecture.
- Elle motive les choix, permet d'identifier les interfaces critiques et force à se concentrer sur les problèmes importants.

II-B-5 - Résumons la démarche...

Modéliser une application ?

Mais comme UML n'est pas un processus...

Quelle démarche utiliser ?

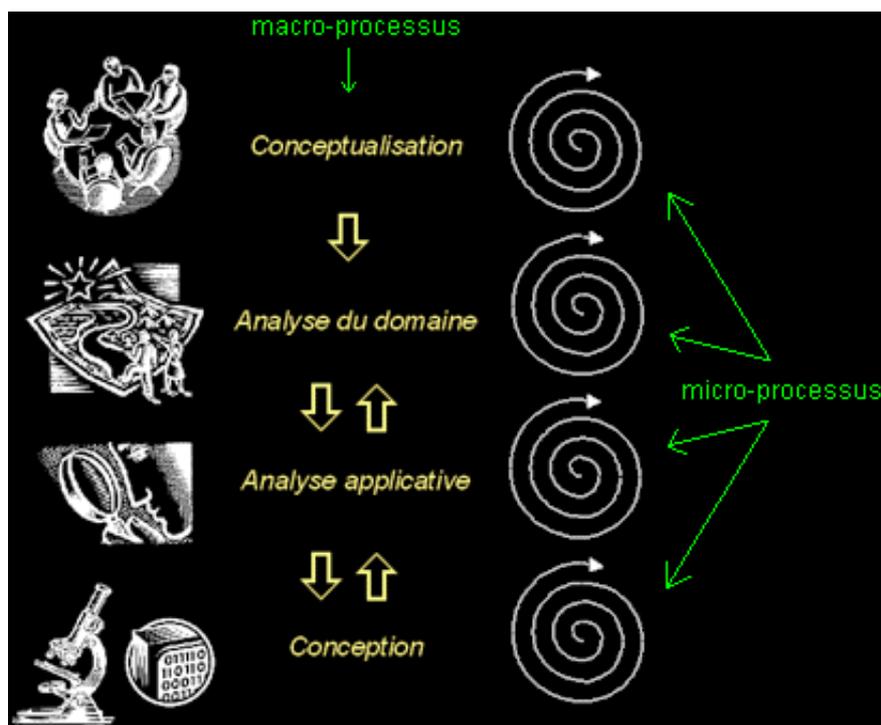
Trouver un bon modèle est une tâche difficile mais capitale !

- 1 Optez pour une approche itérative et incrémentale.
- 2 Centrez votre démarche sur l'analyse des besoins des utilisateurs.
- 3 Prenez grand soin à la définition de l'architecture de votre application (l'approche "4+1" permet de mieux la cerner).

OK OK , mais en pratique ?

- Bien qu'UML n'est pas un processus, il facilite une démarche d'analyse itérative et incrémentale, basée sur les niveaux d'abstraction.
- Les niveaux d'abstraction permettent de structurer les modèles.
- Un micro-processus régit les itérations à niveau d'abstraction constant.
- Un macro-processus régit le passage de niveau à niveau.
- Une démarche incrémentale consiste à construire les modèles de spécification et de conception en plusieurs étapes (cible = catégories).

Le schéma ci-dessous montre les niveaux d'abstraction principaux, qu'on peut identifier dans un processus de développement logiciel :



II-B-6 - Elaboration plutôt que transformation

UML opte pour l'élaboration des modèles, plutôt que pour une approche qui impose une barrière stricte entre analyse et conception :

- Les modèles d'analyse et de conception ne diffèrent que par leur niveau de détail, il n'y a pas de différence dans les concepts utilisés.
- UML n'introduit pas d'éléments de modélisation propres à une activité (analyse, conception...); le langage reste le même à tous les niveaux d'abstraction.

Cette approche simplificatrice facilite le passage entre les niveaux d'abstraction.

- L'élaboration encourage une approche non linéaire (les "retours en arrière" entre niveaux d'abstraction différents sont facilités).
- La traçabilité entre modèles de niveaux différents est assurée par l'unicité du langage.

II-B-7 - Détail des différents niveaux d'abstraction (phases du macro-processus)

Conceptualisation

- L'entrée de l'analyse à ce niveau, est le dossier d'expression des besoins client.
- A ce niveau d'abstraction, on doit capturer les besoins principaux des utilisateurs.
- Il ne faut pas chercher l'exhaustivité, mais clarifier, filtrer et organiser les besoins !
- Le but de la conceptualisation est :
 - de définir le contour du système à modéliser (de spécifier le "quoi"),
 - de capturer les fonctionnalités principales du système, afin d'en fournir une meilleure compréhension (le modèle produit sert d'interface entre les acteurs du projet),
 - de fournir une base à la planification du projet.

Analyse du domaine

- L'entrée de l'analyse à ce niveau, est le modèle des besoins clients (les "cas d'utilisation" UML).
- Il s'agit de modéliser les éléments et mécanismes principaux du système.
- On identifie les éléments du domaine, ainsi que les relations et interactions entre ces éléments :
 - les éléments du domaine sont liés au(x) métier(s) de l'entreprise,
 - ils sont indispensables à la mission du système,
 - ils gagnent à être réutilisés (ils représentent un savoir-faire).
- A ce stade, on organise aussi (selon des critères purement logiques), les éléments du domaine en "catégories" :
 - pour répartir les tâches dans les équipes,
 - regrouper ce qui peut être générique, etc...

Analyse applicative

- A ce niveau, on modélise les aspects informatiques du système, sans pour autant rentrer dans les détails d'implémentation.
- Les interfaces des éléments de modélisation sont définis (cf. encapsulation).
- Les relations entre les éléments des modèles sont définies.
- Les éléments de modélisation utilisés peuvent être propres à une version du système.

Conception

- On y modélise tous les rouages d'implémentation et on détaille tous les éléments de modélisation issus des niveaux supérieurs.
- Les modèles sont optimisés, car destinés à être implémentés.

II-B-8 - Activités des micro-processus d'analyse (niveau d'abstraction constant)

- **A chaque niveau d'abstraction, un micro-processus régit la construction des modèles.**

- **UML ne régit pas les activités des micro-processus, c'est le principe d'abstraction qui permet l'élaboration itérative et incrémentale des modèles.**
- **Exemple de micro-processus de construction d'un modèle :**

identifiez les classes (d'objets) :

- recherchez les classes candidates (différentes suivant le niveau d'abstraction) à l'aide de diagrammes d'objets (ébauches),
- filtrez les classes redondantes, trop spécifiques ou vagues, qui ne représentent qu'une opération ou un attribut,
- documentez les caractéristiques des classes retenues (persistances, nombre maximum d'instances, etc.).

identifiez les associations entre classes / interactions entre objets (instances) :

- recherchez les connexions sémantiques et les relations d'utilisation,
- documentez les relations (nom, cardinalités, contraintes, rôles des classes...),
- en spécification, filtrez les relations instables ou d'implémentation,
- définissez la dynamique des relations entre objets (les interactions entre instances de classes et les activités associées).

identifiez les attributs et les opérations des classes :

- recherchez les attributs dans les modèles dynamiques (recherchez les données qui caractérisent les états des objets),
- filtrez les attributs complexes (faites-en des objets) et au niveau spécification, ne représentez pas les valeurs internes propres aux mécanismes d'implémentation,
- recherchez les opérations parmi les activités et actions des objets (ne pas rentrer dans le détail au niveau spécification).

optimisez les modèles :

- choisissez vos critères d'optimisation (généricité, évolutivité, précision, lisibilité, simplicité...),
- utilisez la généralisation et la spécialisation (classification),
- documentez et détaillez vos modèles, encapsulez.

validez les modèles :

- vérifiez la cohérence, la complétude et l'homogénéité des modèles,
- confrontez les modèles à la critique (comité de relecture...).

II-B-9 - Synthèse de la démarche

Modéliser une application n'est pas une activité linéaire.

Il s'agit d'une tâche très complexe, qui nécessite une approche :

- itérative et incrémentale (grâce aux niveaux d'abstraction), car il est plus efficace de construire et valider par étapes, ce qui est difficile à cerner et maîtriser,
- centrée sur l'architecture (définie par la vue "4+1"), car il s'agit de la clé de voûte qui conditionne la plupart des qualités d'une application informatique,
- guidée par la prise en compte des besoins des utilisateurs (à l'aide des cas d'utilisation), car ce qui motive l'existence même du système à concevoir, c'est la satisfaction des besoins de ses utilisateurs.

II-B-10 - Les diagrammes UML

OK pour la démarche d'élaboration d'un modèle, mais...

II-B-10-a - Comment "rédiger" un modèle avec UML ?

- UML permet de définir et de visualiser un modèle, à l'aide de diagrammes.
- Un diagramme UML est une représentation graphique, qui s'intéresse à un aspect précis du modèle ; c'est une perspective du modèle, pas "le modèle".
- Chaque type de diagramme UML possède une structure (les types des éléments de modélisation qui le composent sont prédéfinis).
- Un type de diagramme UML véhicule une sémantique précise (un type de diagramme offre toujours la même vue d'un système).
- Combinés, les différents types de diagrammes UML offrent une vue complète des aspects statiques et dynamiques d'un système.
- Par extension et abus de langage, un diagramme UML est aussi un modèle (un diagramme modélise un aspect du modèle global).

II-B-10-b - Quelques caractéristiques des diagrammes UML

- Les diagrammes UML supportent l'abstraction. Leur niveau de détail caractérise le niveau d'abstraction du modèle.
- La structure des diagrammes UML et la notation graphique des éléments de modélisation est normalisée (document "UML notation guide").
- Rappel : la sémantique des éléments de modélisation et de leur utilisation est définie par le métamodèle UML (document "UML semantics").
- Le recours à des outils appropriés est un gage de productivité pour la rédaction des diagrammes UML, car :
 - ils facilitent la navigation entre les différentes vues,
 - ils permettent de centraliser, organiser, partager, synchroniser et versionner les diagrammes (indispensable avec un processus itératif),
 - facilitent l'abstraction, par des filtres visuels,
 - simplifient la production de documents et autorisent (dans certaines limites) la génération de code.

II-B-10-c - Les différents types de diagrammes UML



Vues statiques du système :

- diagrammes de cas d'utilisation
- diagrammes d'objets
- diagrammes de classes
- diagrammes de composants
- diagrammes de déploiement



Vues dynamiques du système :

- diagrammes de collaboration
- diagrammes de séquence
- diagrammes d'états-transitions
- diagrammes d'activités

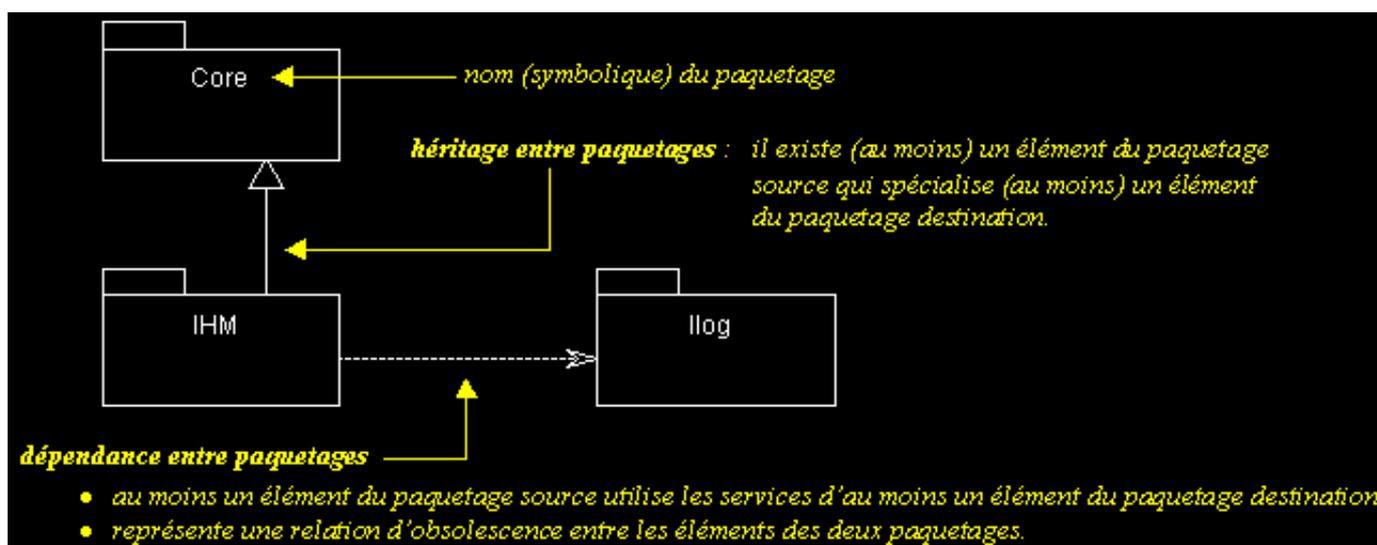
II-C - Les vues statiques d'UML

II-C-1 - LES PAQUETAGES

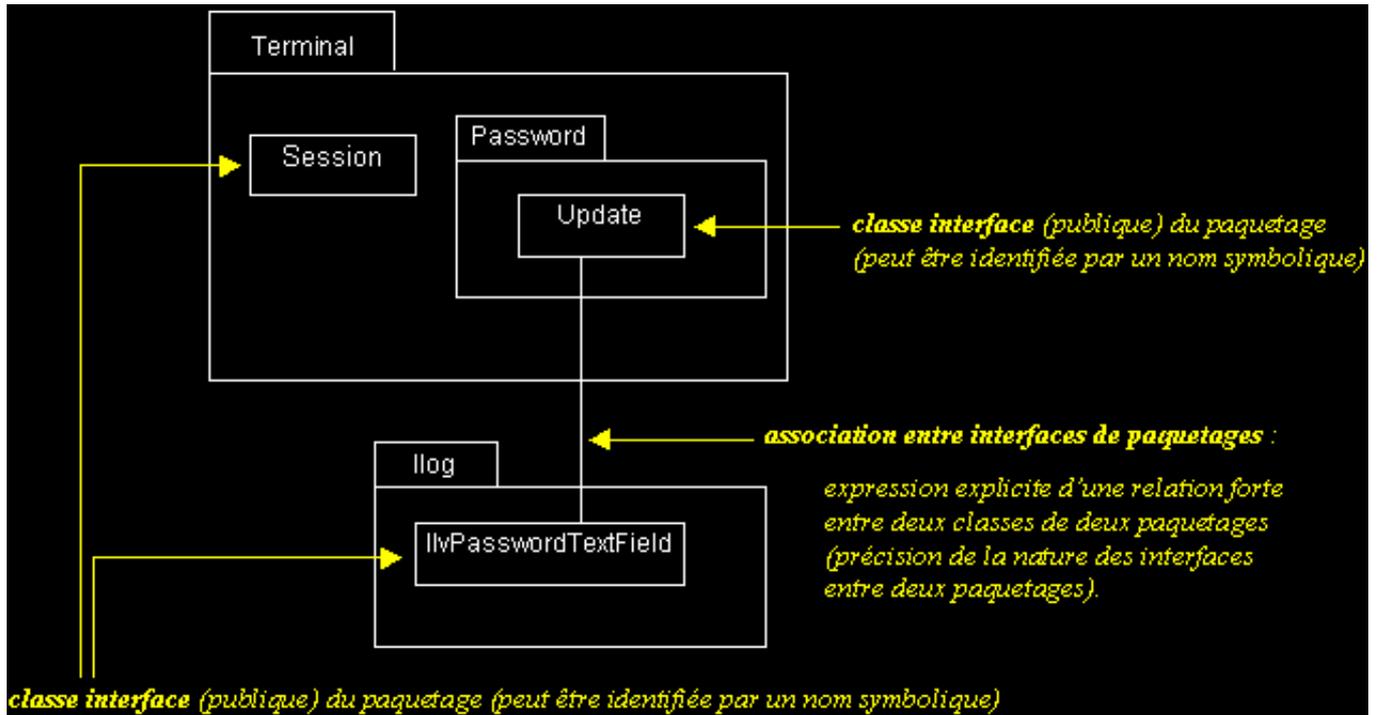
II-C-1-a - Paquetages (packages)

- 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.

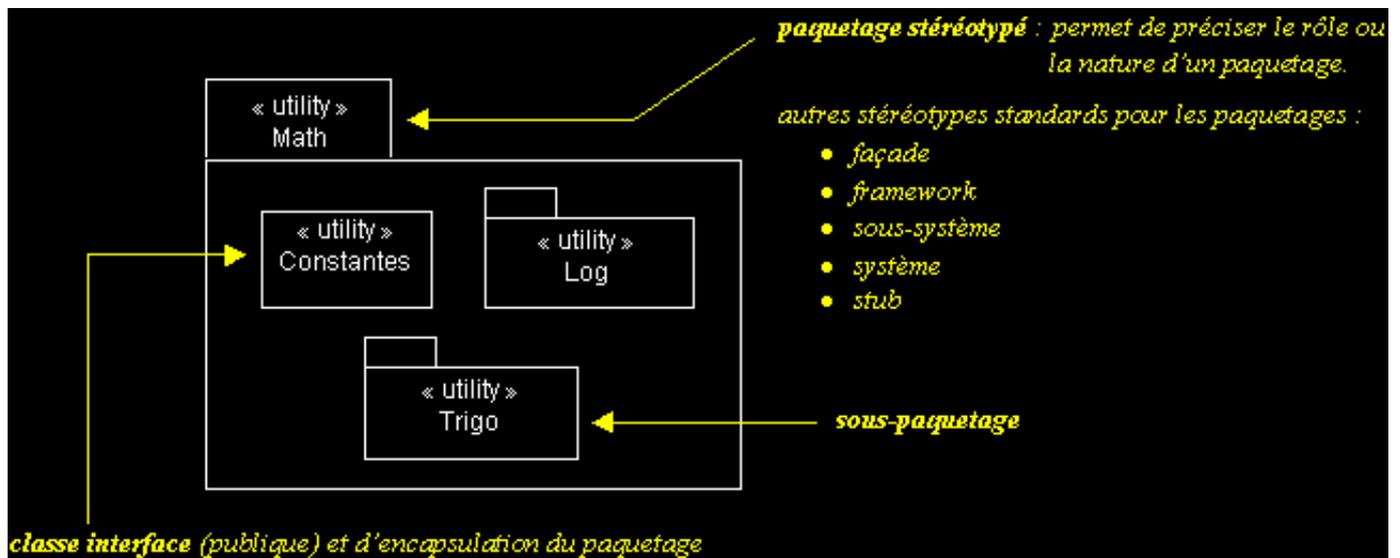
II-C-1-b - Paquetages : relations entre paquetages

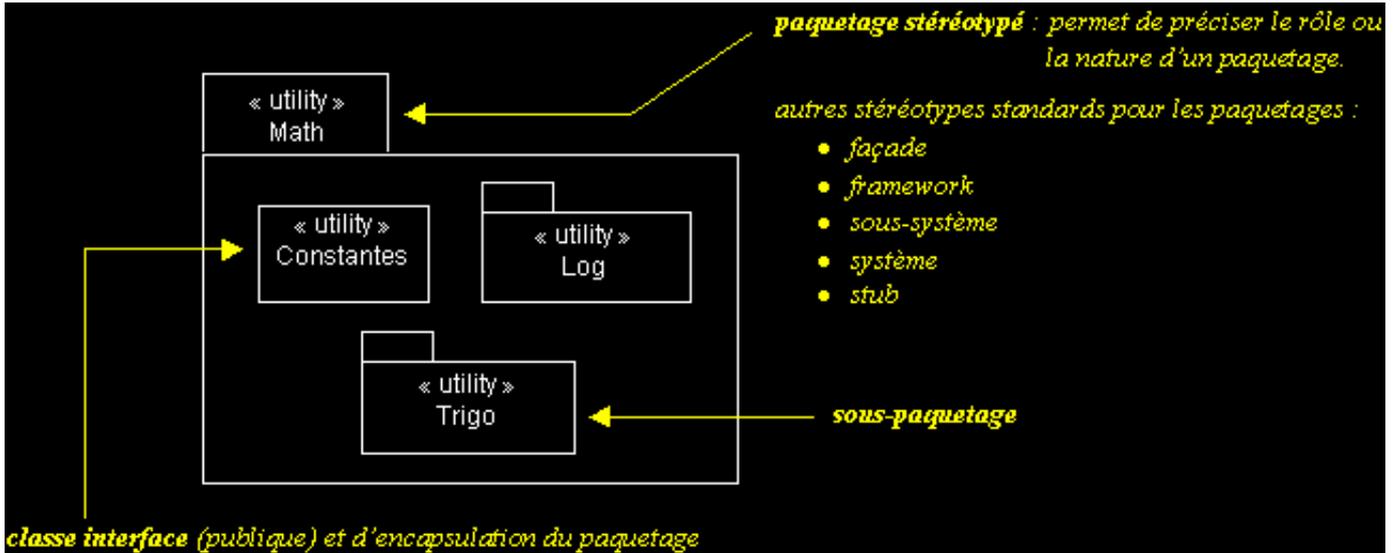


II-C-1-c - Paquetages : interfaces



II-C-1-d - Paquetages : stéréotypes



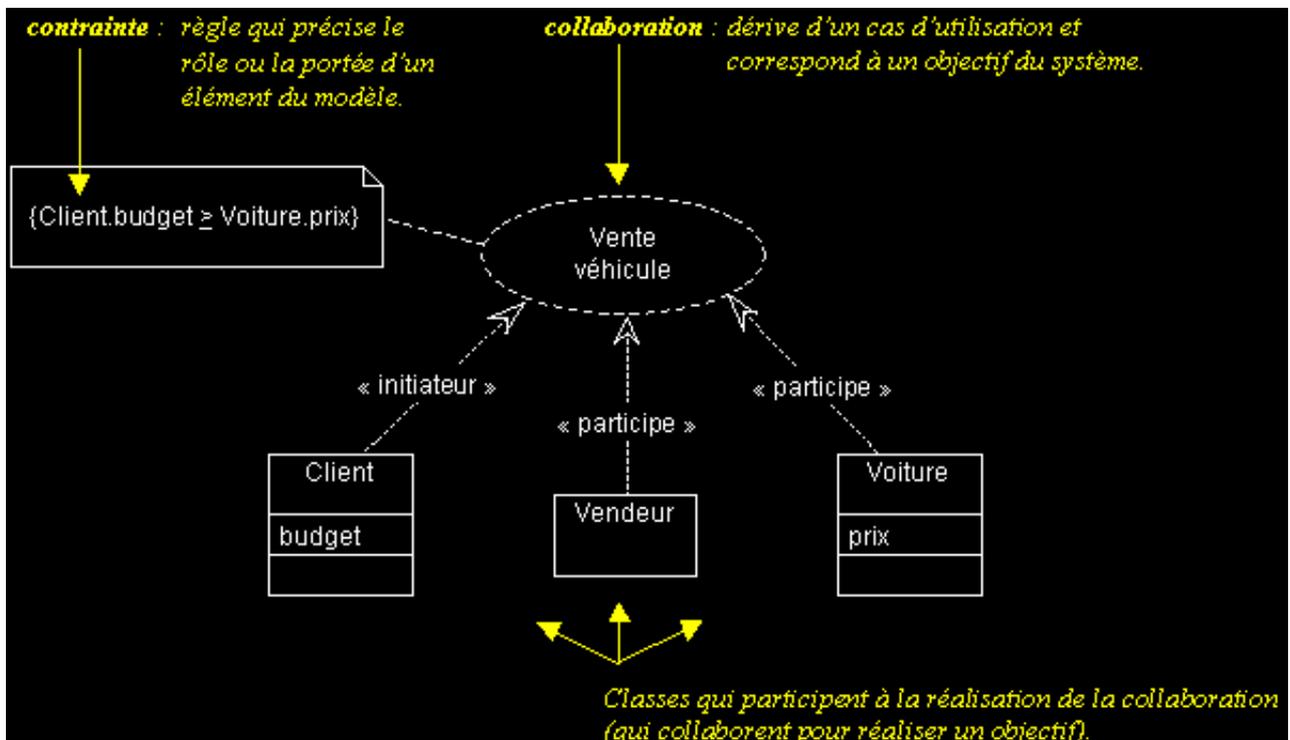


II-C-2 - LA COLLABORATION

II-C-2-a - Symbole de modélisation "collaboration"

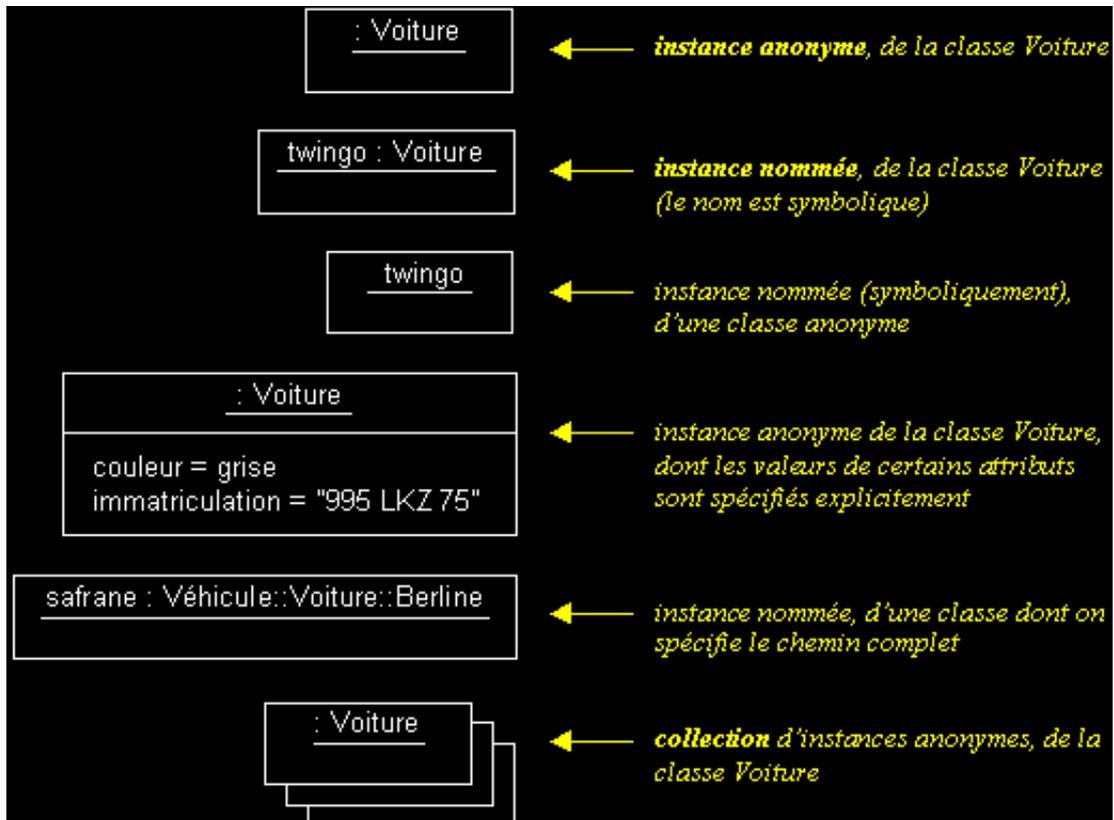
- Les collaborations sont des interactions entre objets, dont le but est de réaliser un objectif du système (c'est-à-dire aussi de répondre à un besoin d'un utilisateur).
- L'élément de modélisation UML "collaboration", représente les classes qui participent à la réalisation d'un cas d'utilisation.

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

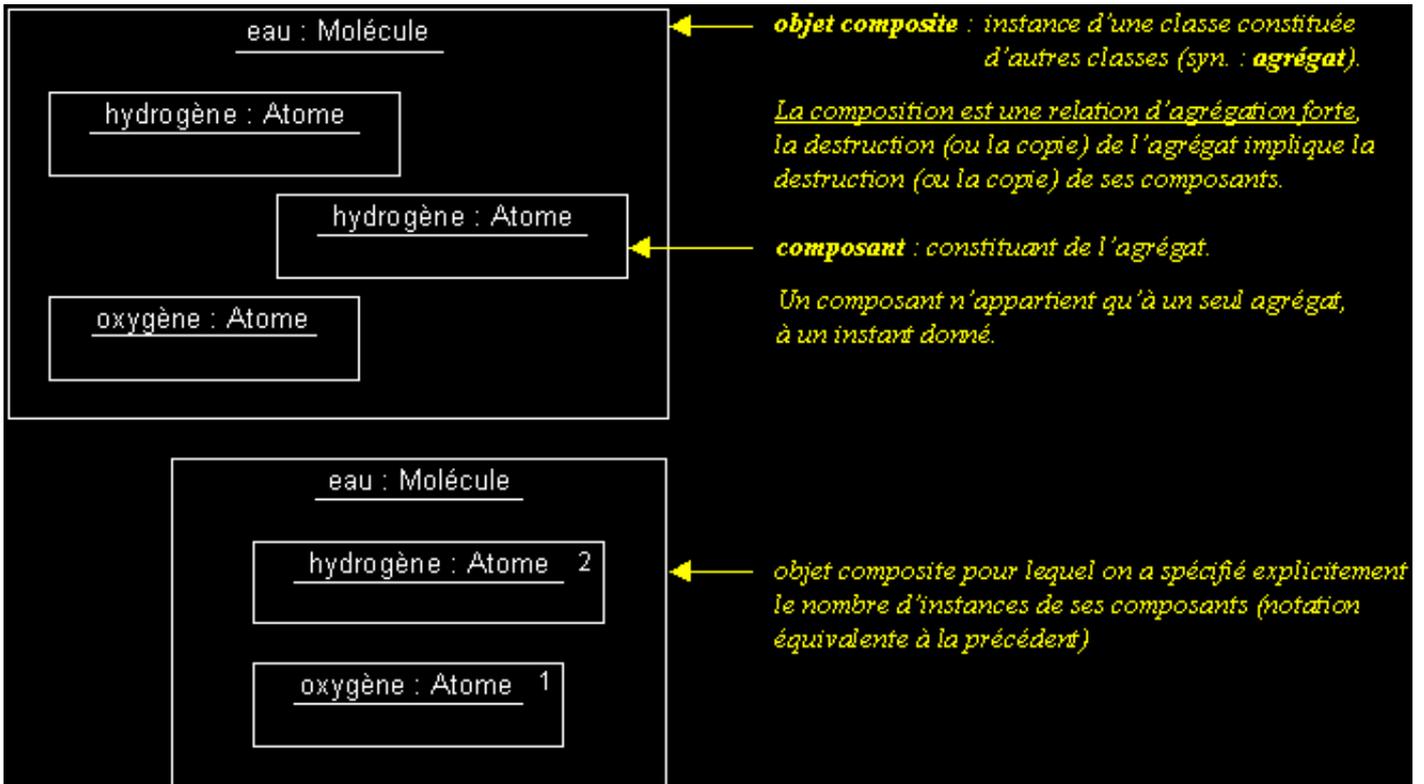


II-C-3 - INSTANCES ET DIAGRAMME D'OBJETS

II-C-3-a - Exemples d'instances



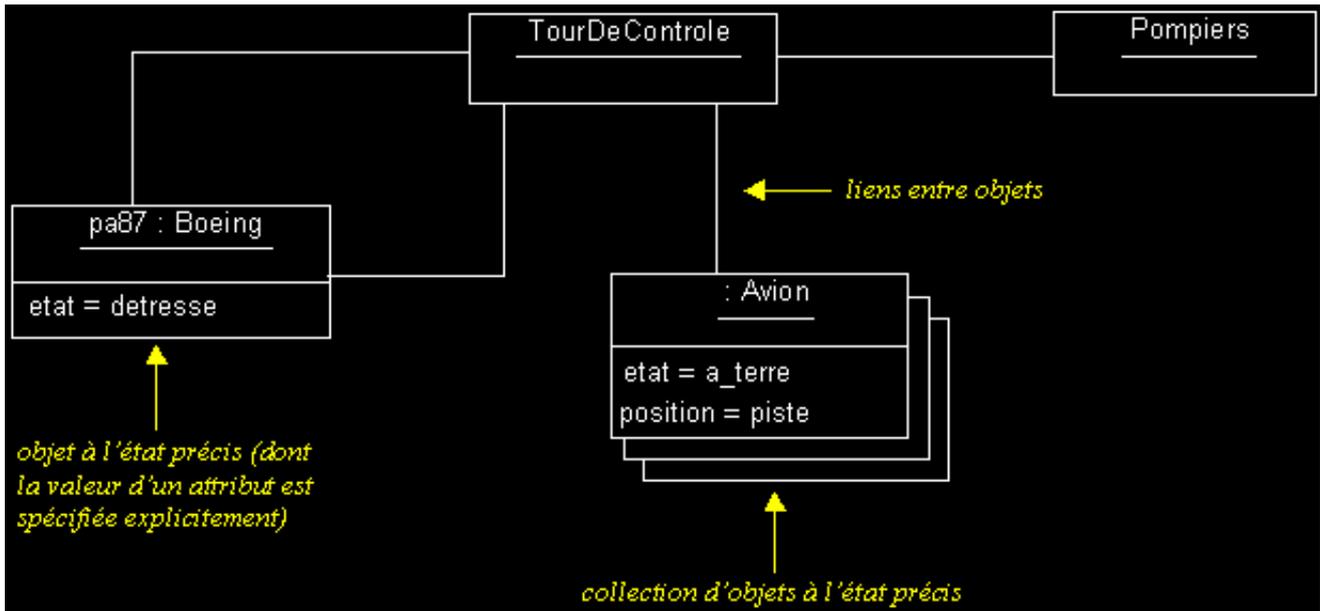
II-C-3-b - Objets composites



II-C-3-c - 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 :

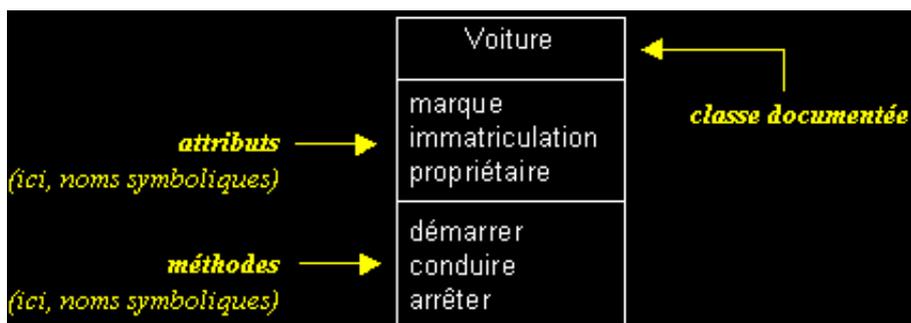


II-C-4 - LES CLASSES

II-C-4-a - 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.

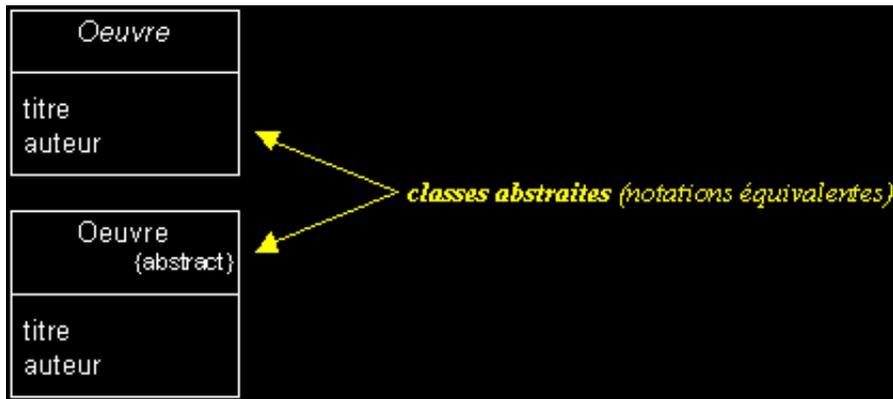
Documentation d'une classe (niveaux d'abstraction), exemples :



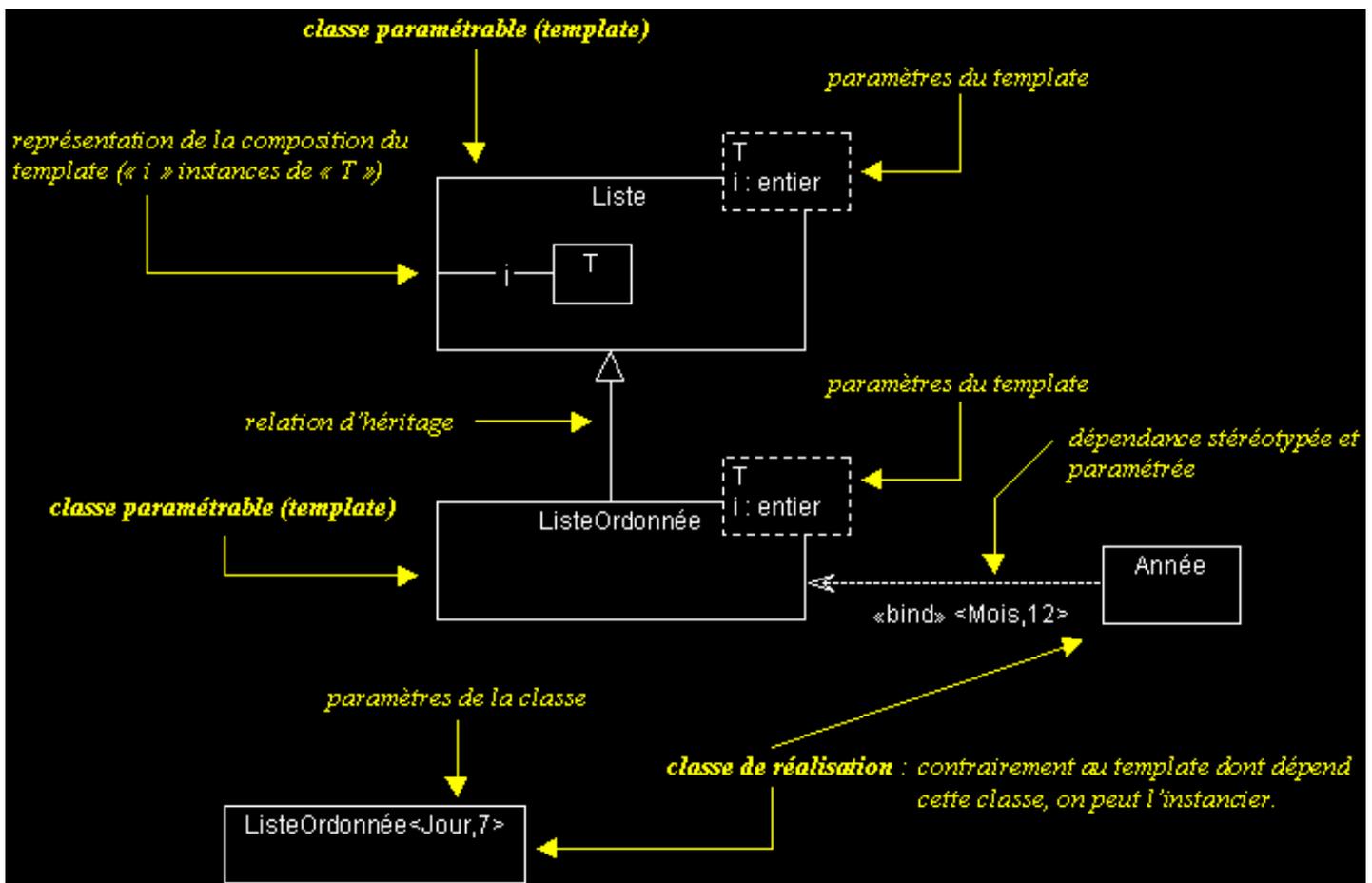
Attributs multivalués et dérivés, exemples :



Classe abstraite, exemple :



Template (classe paramétrable), exemple :



II-C-5 - DIAGRAMME DE CLASSES

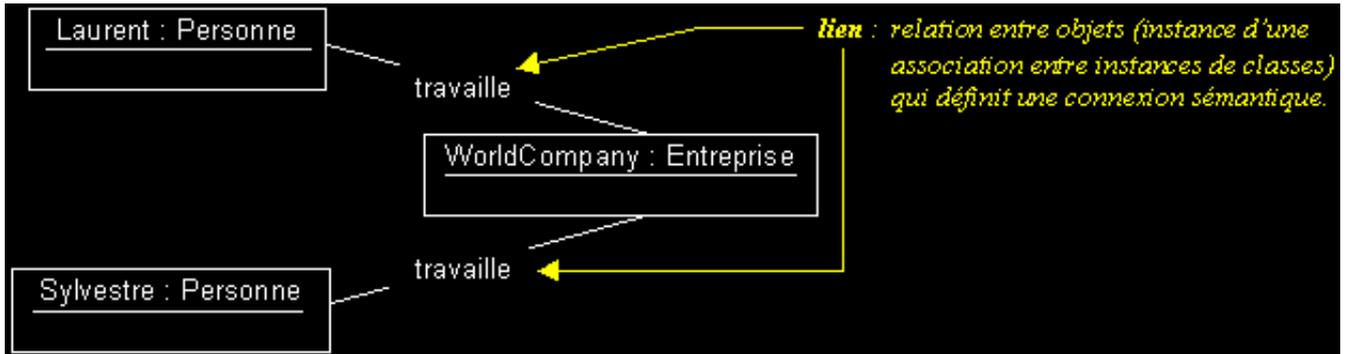
II-C-5-a - Diagramme de classes : sémantique

- 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.

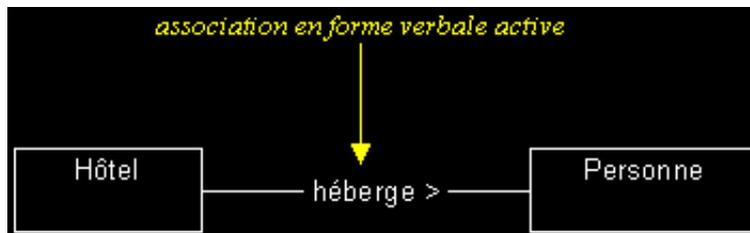
II-C-5-b - Associations entre classes

- 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.

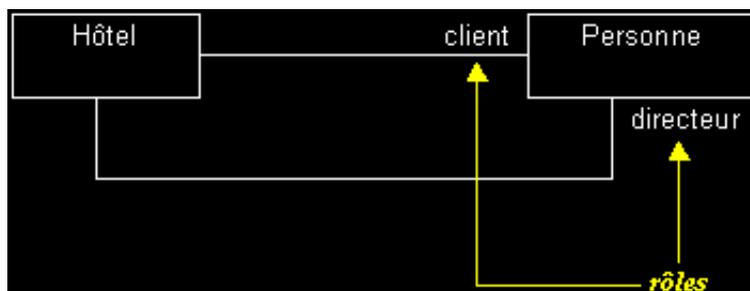


II-C-5-c - Documentation d'une association et types d'associations

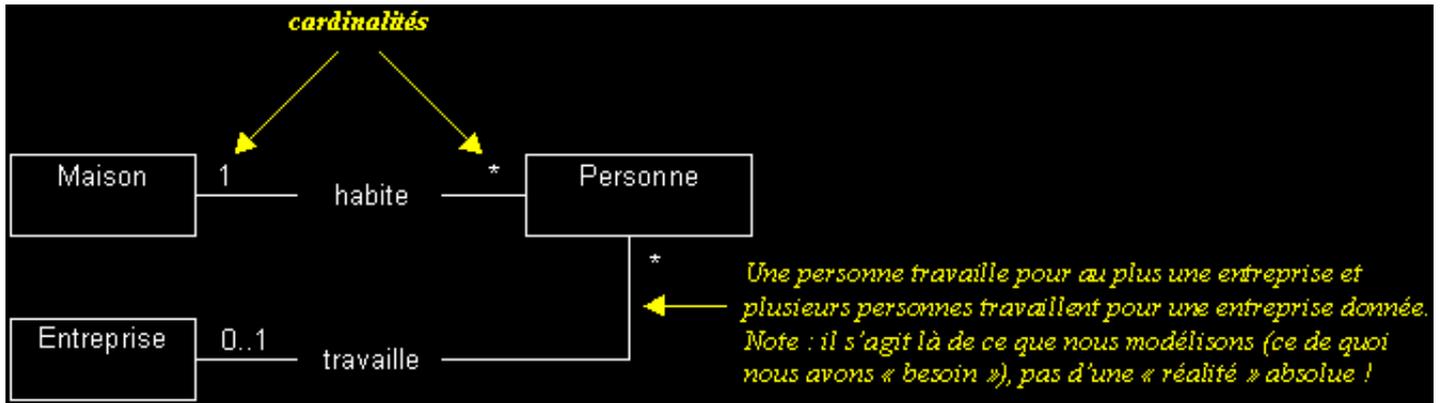
Association en forme verbale active : précise le sens de lecture principal d'une association.
Voir aussi : association à navigabilité restreinte.



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



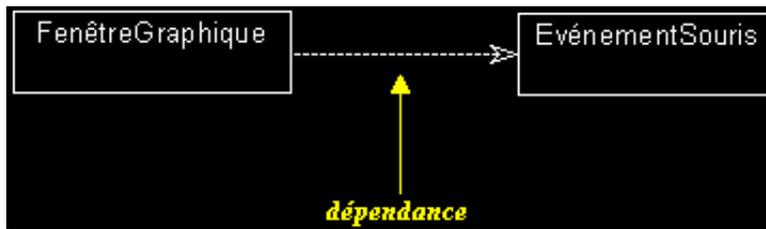
Cardinalités : précise le nombre d'instances qui participent à une relation.



Expression des cardinalités d'une relation en UML :

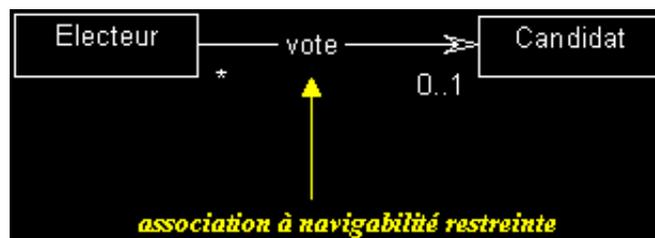
- n : exactement "n" (n, entier naturel > 0)
exemples : "1", "7"
- n..m : de "n" à "m" (entiers naturels ou variables, m > n)
exemples : "0..1", "3..n", "1..31"
- * : plusieurs (équivalent à "0..n" et "0..*")
- n..* : "n" ou plus (n, entier naturel ou variable)
exemples : "0..*", "5..*"

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).



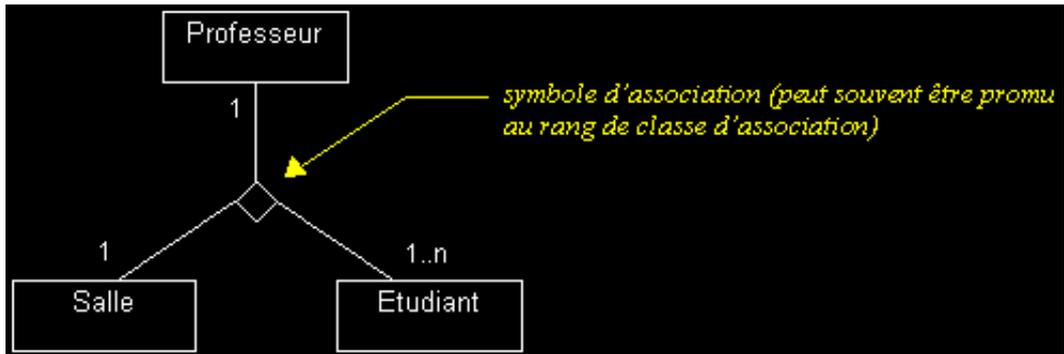
Association à navigabilité restreinte

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.

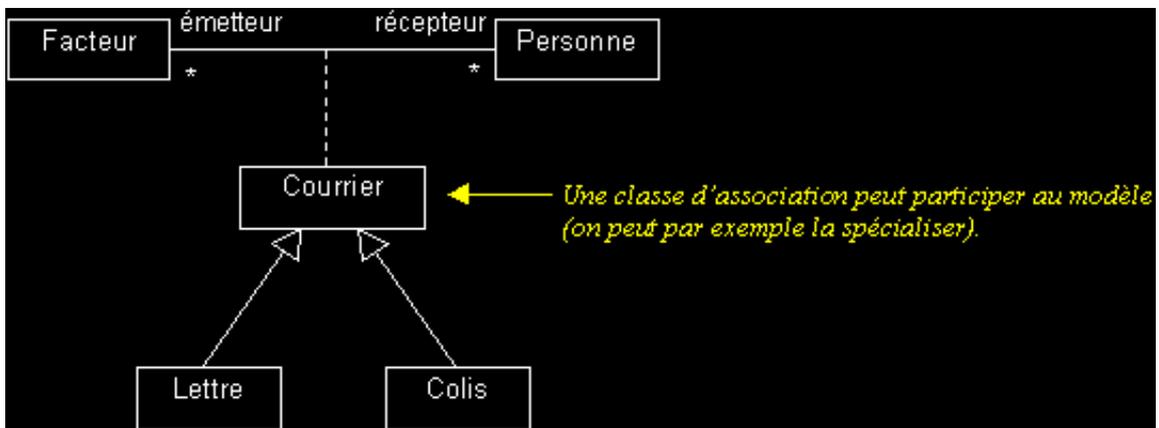
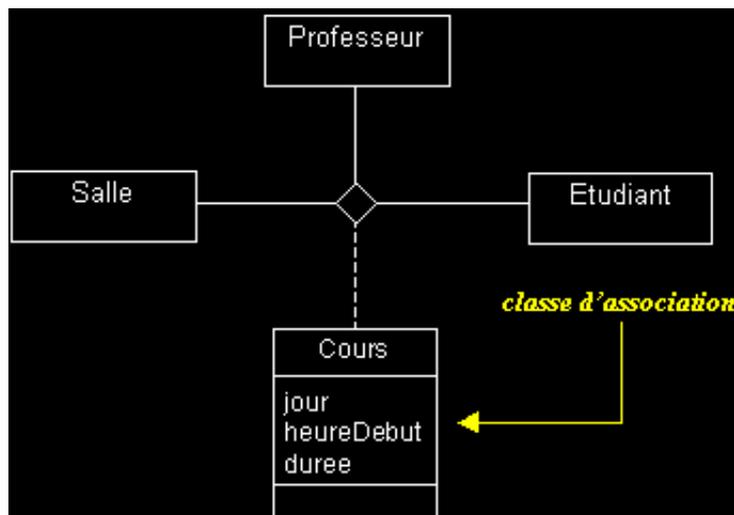


Association n-aire : il s'agit d'une association qui relie plus de deux classes...

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.

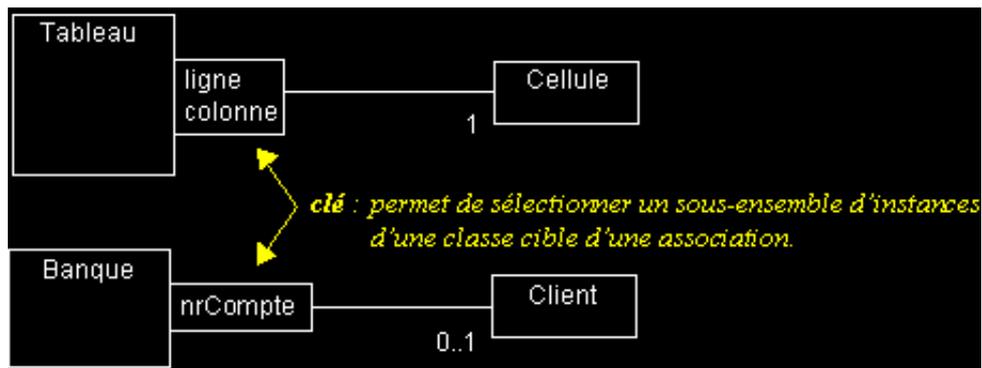


Classe d'association : il s'agit d'une classe qui réalise la navigation entre les instances d'autres classes.



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.



II-C-5-d - Héritage

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

Spécialisation

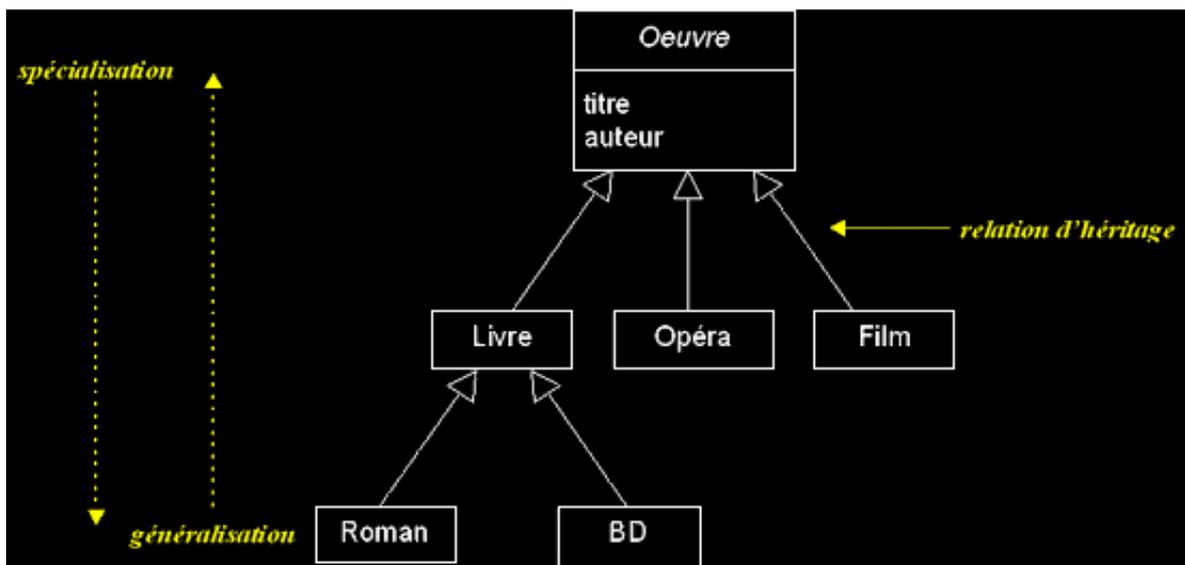
- Démarche descendante, qui consiste à capturer les particularités d'un ensemble d'objets, non discriminés par les classes déjà identifiées.
- Consiste à étendre les propriétés d'une classe, sous forme de sous-classes, plus spécifiques (permet l'extension du modèle par réutilisation).

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 généralité).

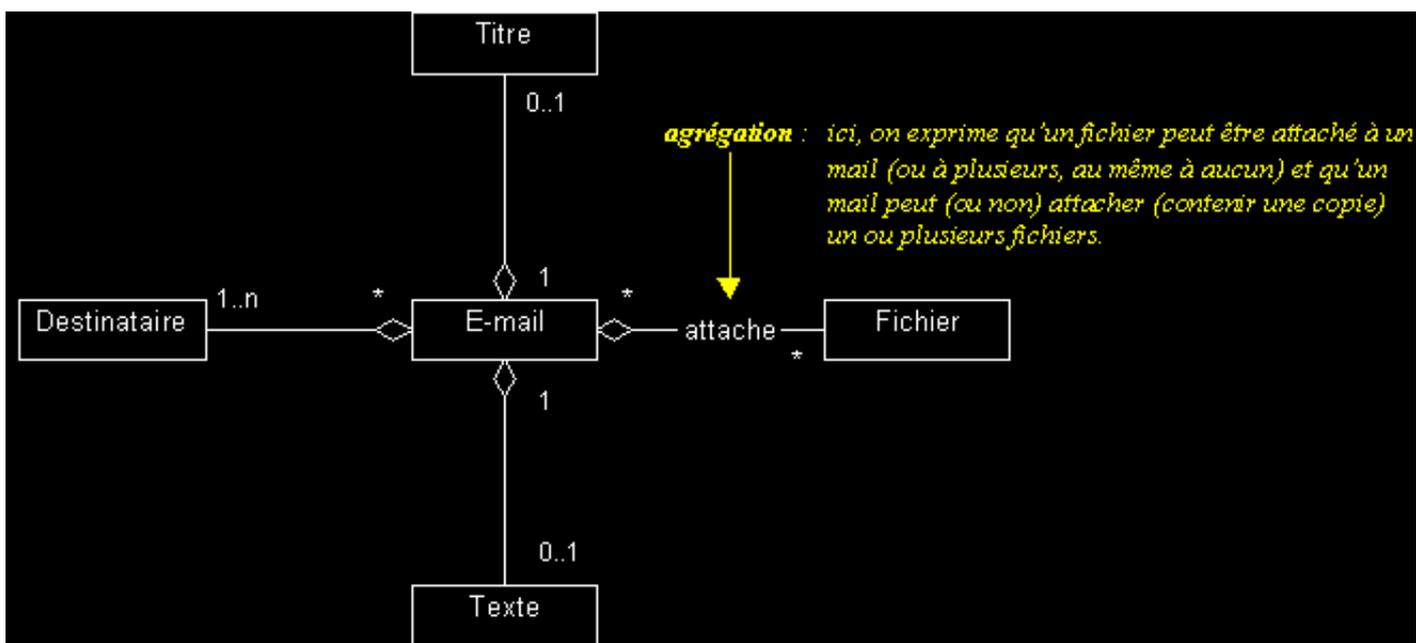
Classification

- 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 (Liksov, 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).



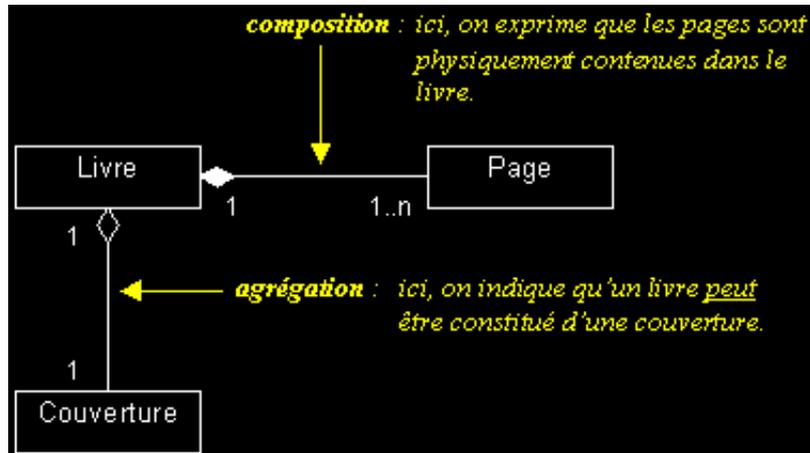
II-C-5-e - Agrégation

- 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.



II-C-5-f - 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.

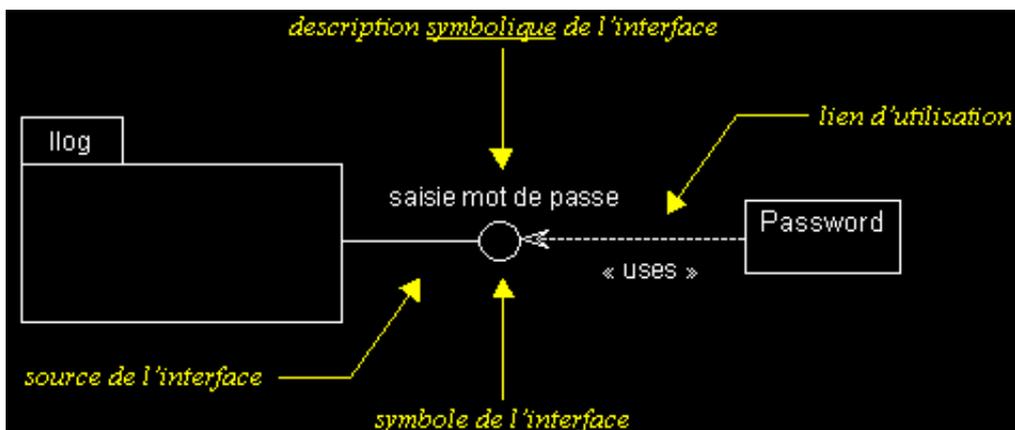


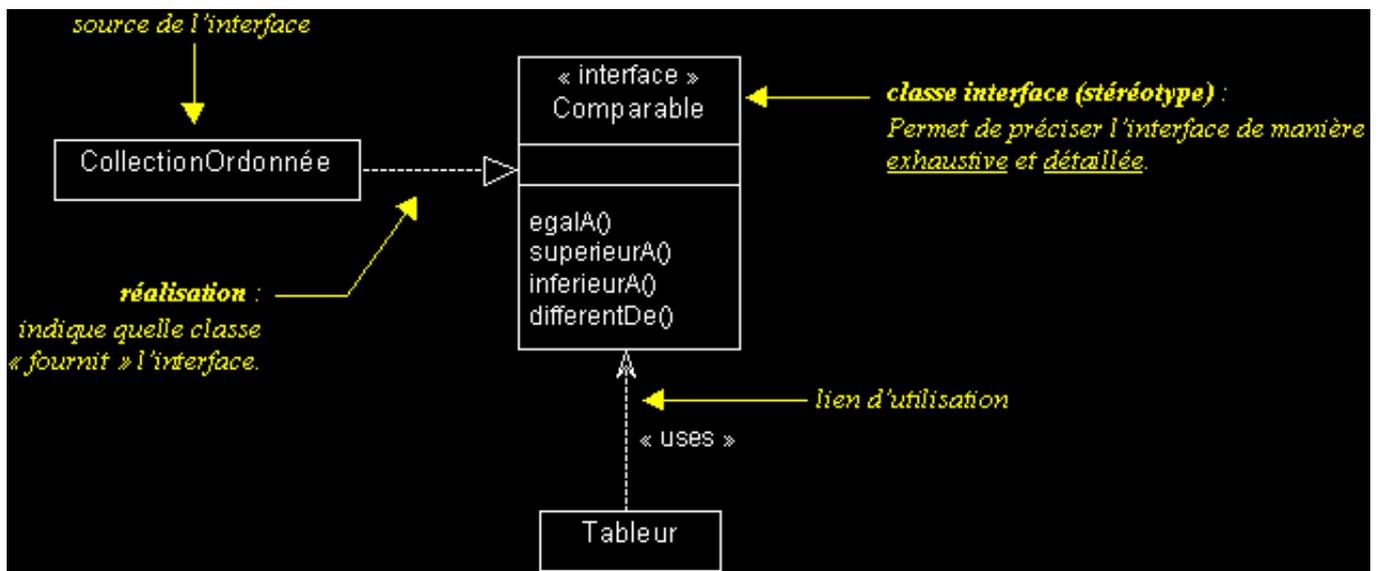
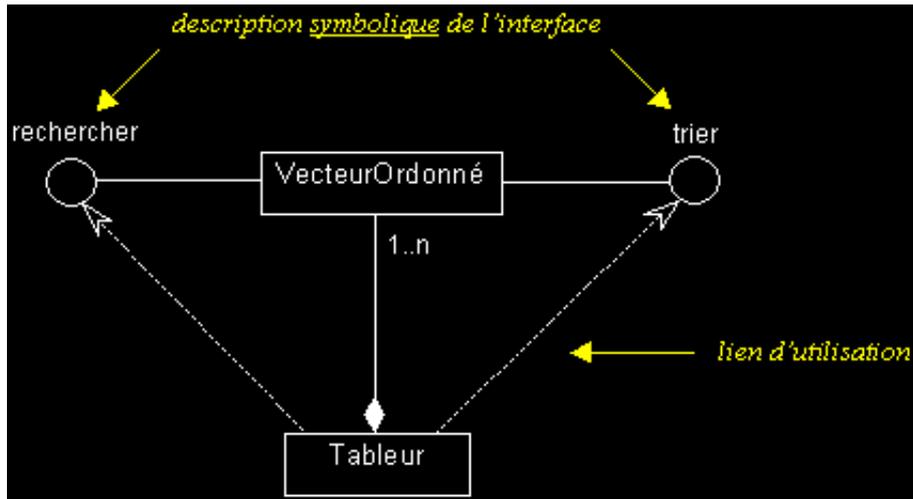
II-C-5-g - Agrégation et composition : rappel

- 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 !

II-C-5-h - Interfaces

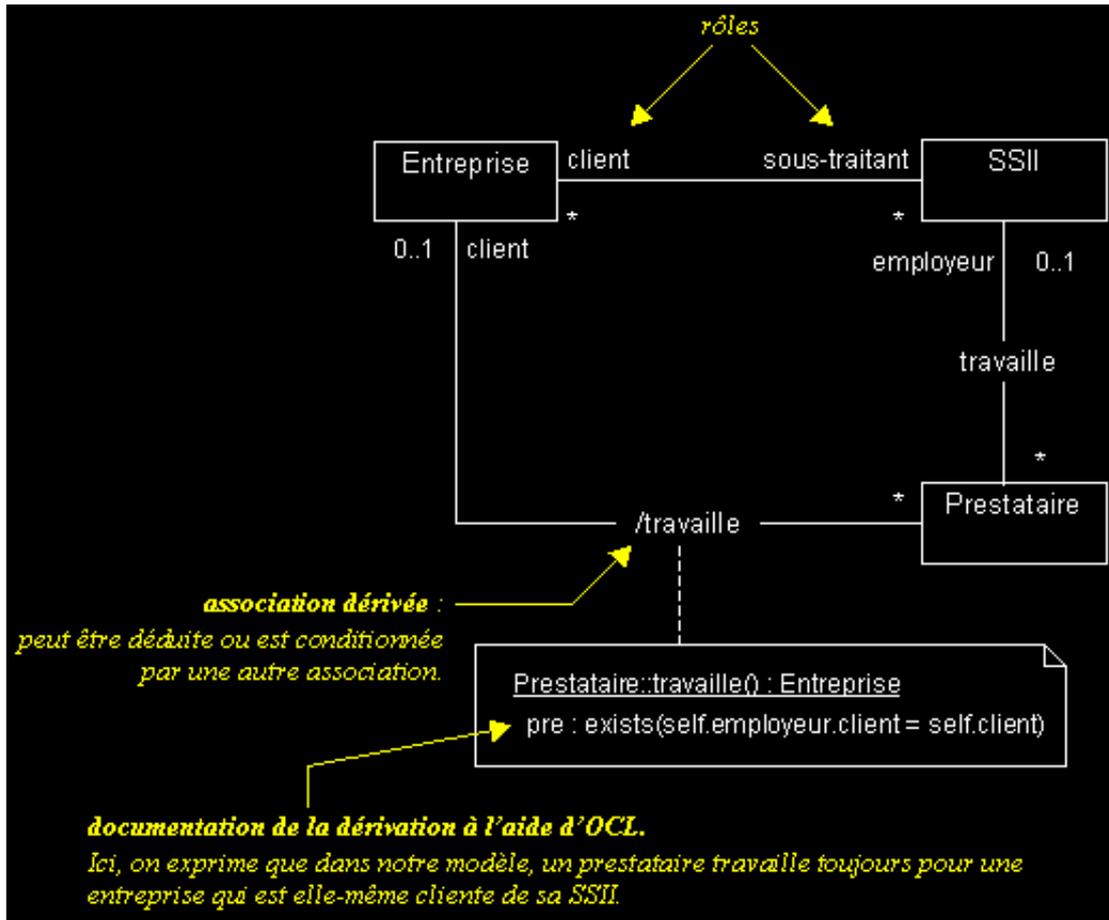
- Une interface fournit une vue totale ou partielle d'un ensemble de services offerts par une classe, un paquetage ou un composant. Les éléments qui utilisent l'interface peuvent exploiter tout ou partie de l'interface.
- Dans un modèle UML, le symbole "interface" sert à identifier de manière explicite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





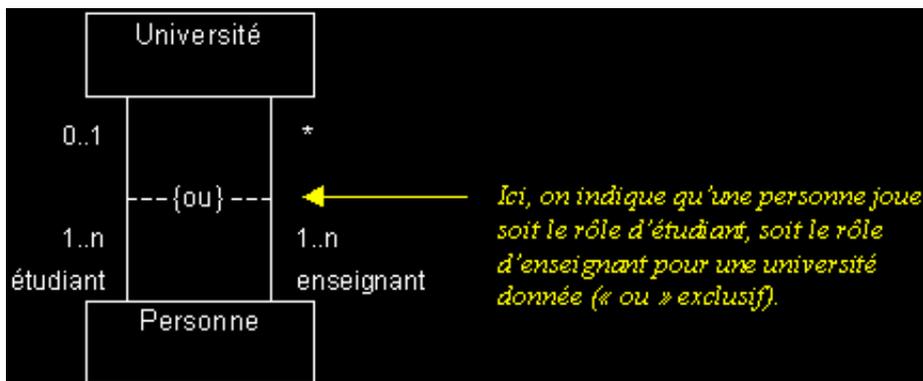
II-C-5-i - Association dérivée

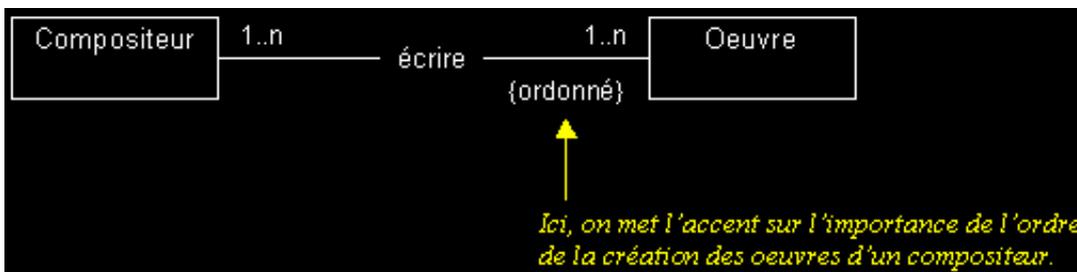
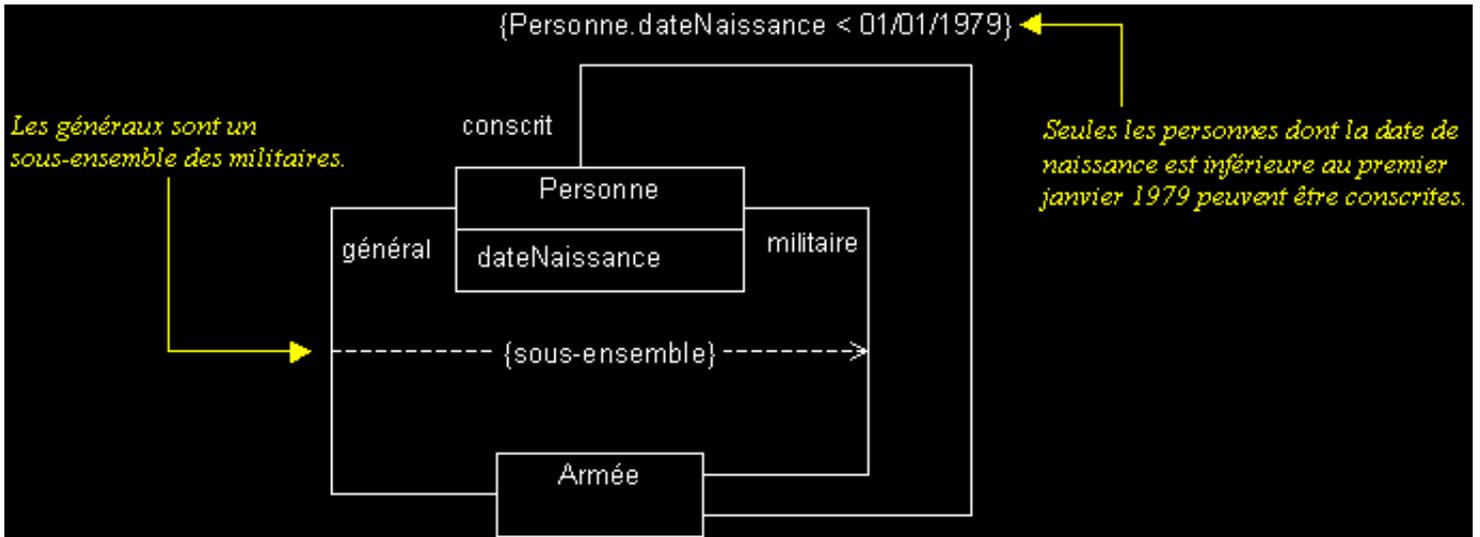
- Les associations dérivées sont des associations redondantes, qu'on peut déduire d'une autre association ou d'un ensemble d'autres associations.
- Elles permettent d'indiquer des chemins de navigation "calculés", sur un diagramme de classes.
- Elles servent beaucoup à la compréhension de la navigation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



II-C-5-j - 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.





II-C-5-k - 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.

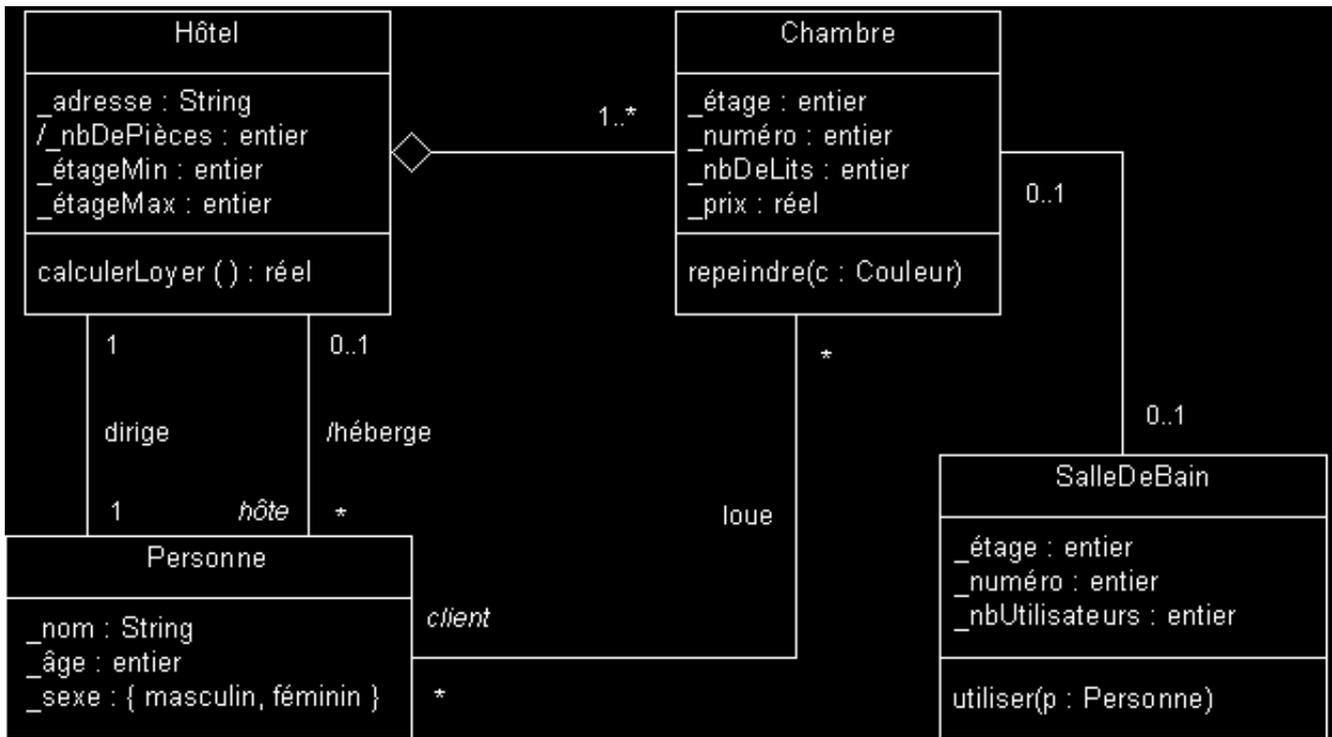
Les pièces de l'hôtel qui ne sont ni des chambres, ni des salles de bain (hall d'accueil, cuisine...) ne font pas partie de l'étude (hors sujet).

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)...

i 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

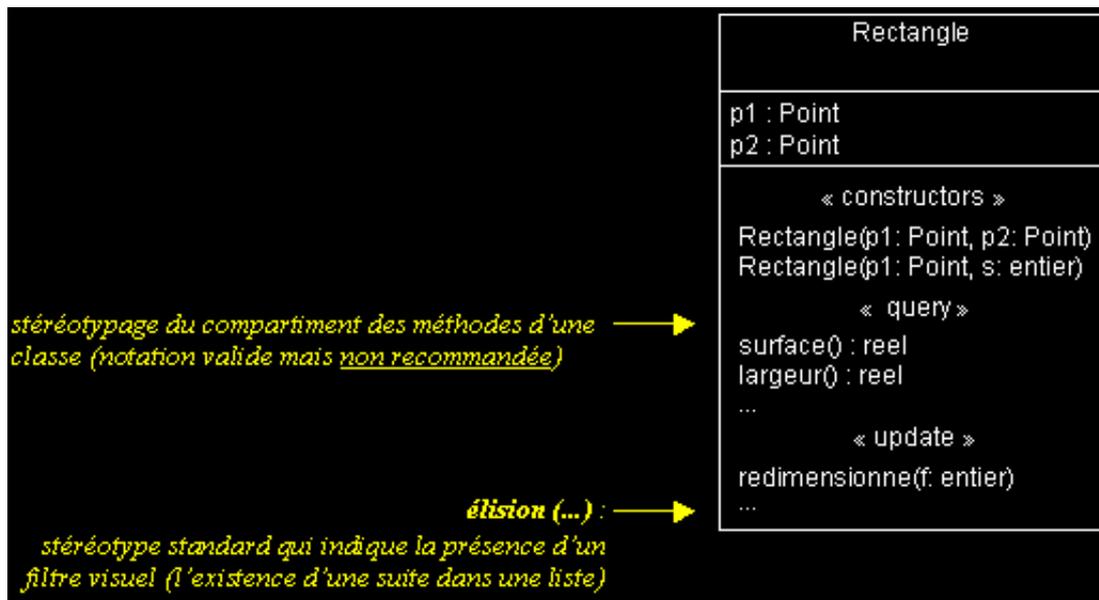
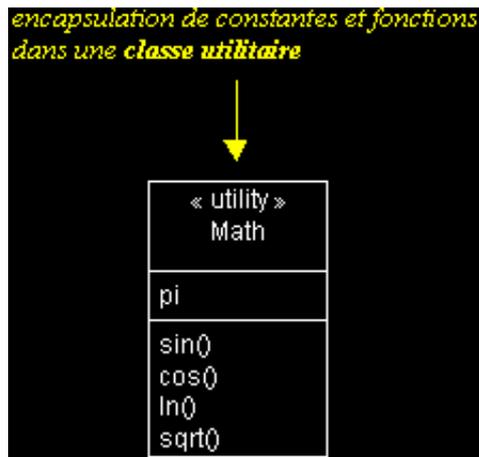
```
context SalleDeBain::utiliser(p : Personne)
pre: if chambre->notEmpty then
  chambre.client->includes(p)
else
  p.chambre._étage = self._étage
endif
post: _nbUtilisateurs = _nbUtilisateurs@pre + 1
```

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
```

II-C-5-I - 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).



II-C-6 - DIAGRAMMES DE COMPOSANTS ET DE DEPLOIEMENT

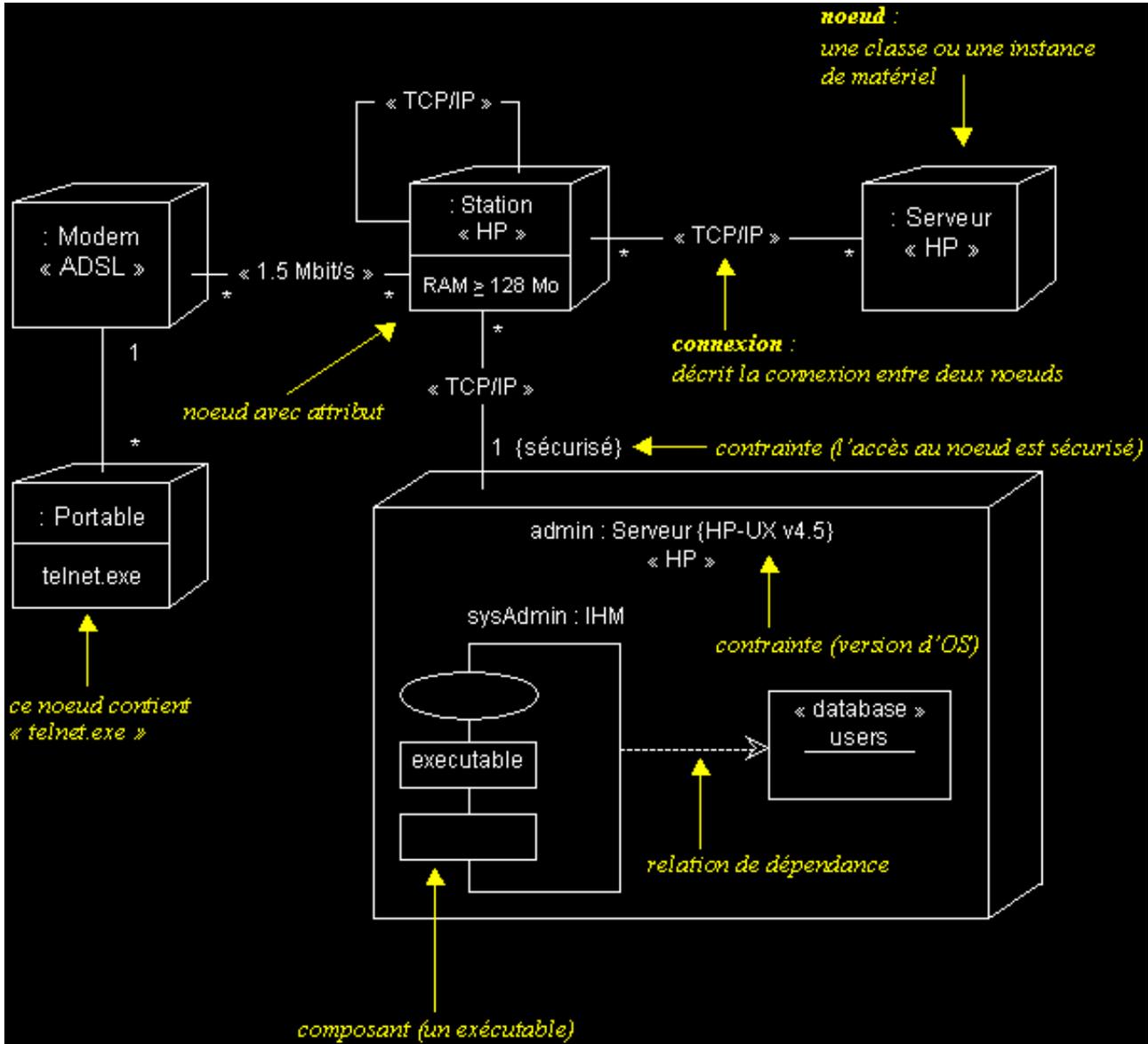
II-C-6-a - Diagramme de composants

- Les diagrammes de composants permettent de décrire l'architecture physique et statique d'une application en terme de modules : fichiers sources, bibliothèques, 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.
- Les 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) :

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").



II-D - Les vues dynamiques d'UML



II-D-1 - LES CAS D'UTILISATION

II-D-1-a - 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.

II-D-1-b - 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 pensés 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.

II-D-1-c - Eléments de base des cas d'utilisation

Acteur : entité externe qui agit sur le système (opérateur, autre système...).

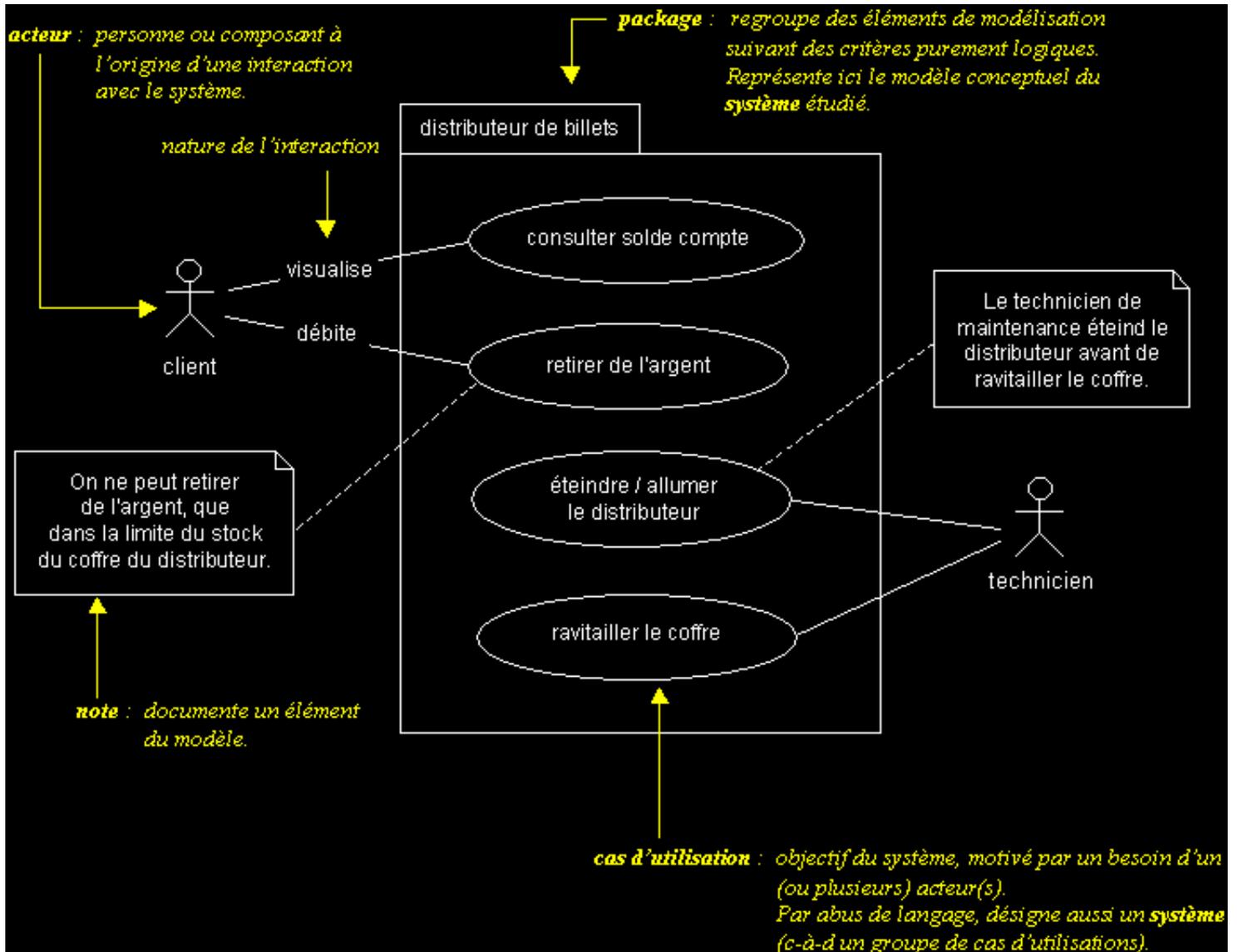
- L'acteur peut consulter ou modifier l'état du système.
- En réponse à l'action d'un acteur, le système fournit un service qui correspond à son besoin.
- Les acteurs peuvent être classés (hiérarchisés).

Use case : ensemble d'actions réalisées par le système, en réponse à une action d'un acteur.

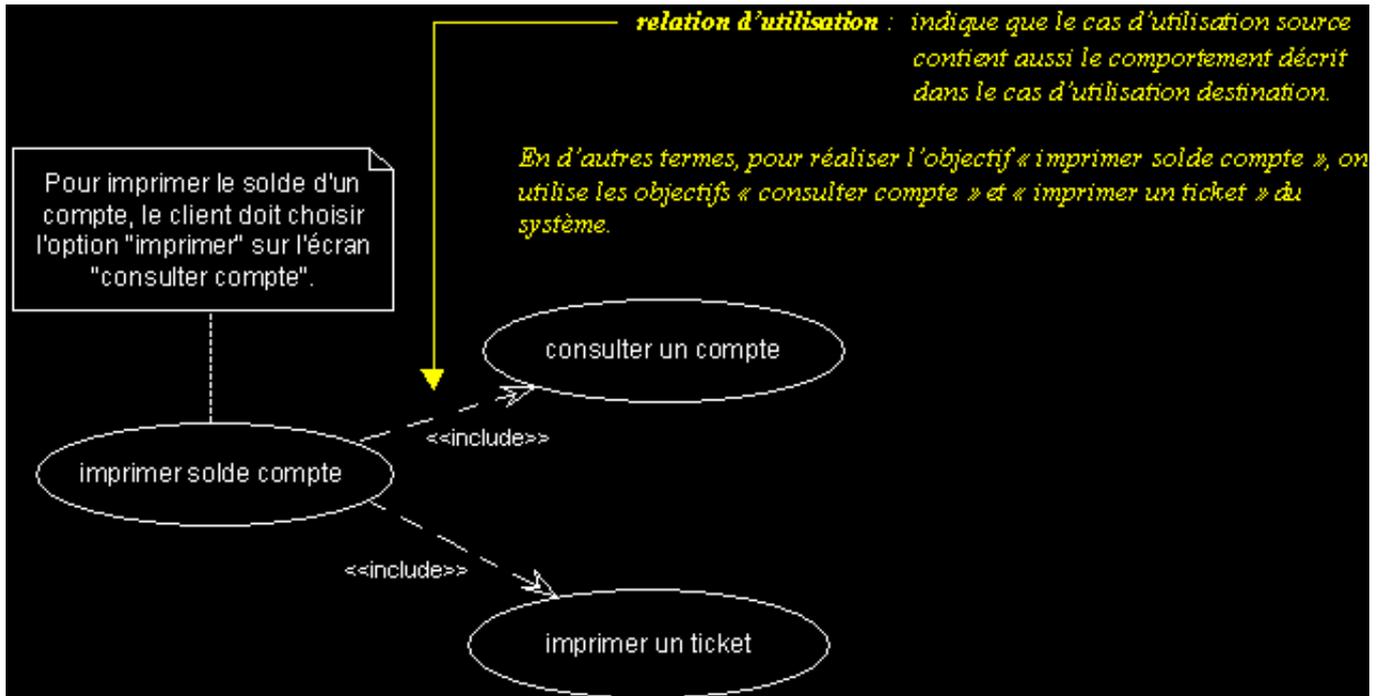
- Les use cases peuvent être structurés.
- Les use cases peuvent être organisés en paquetages (packages).
- L'ensemble des use cases décrit les objectifs (le but) du système.

II-D-1-d - Exemples

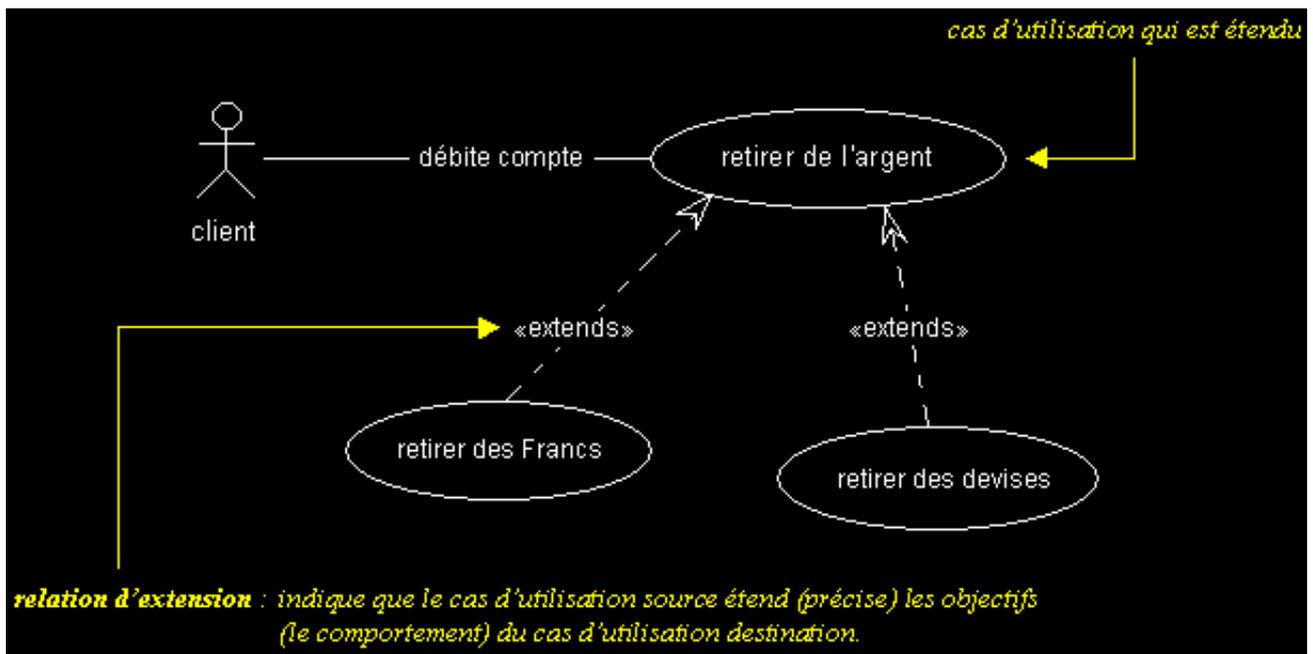
Cas d'utilisation standard :



Relation d'utilisation :



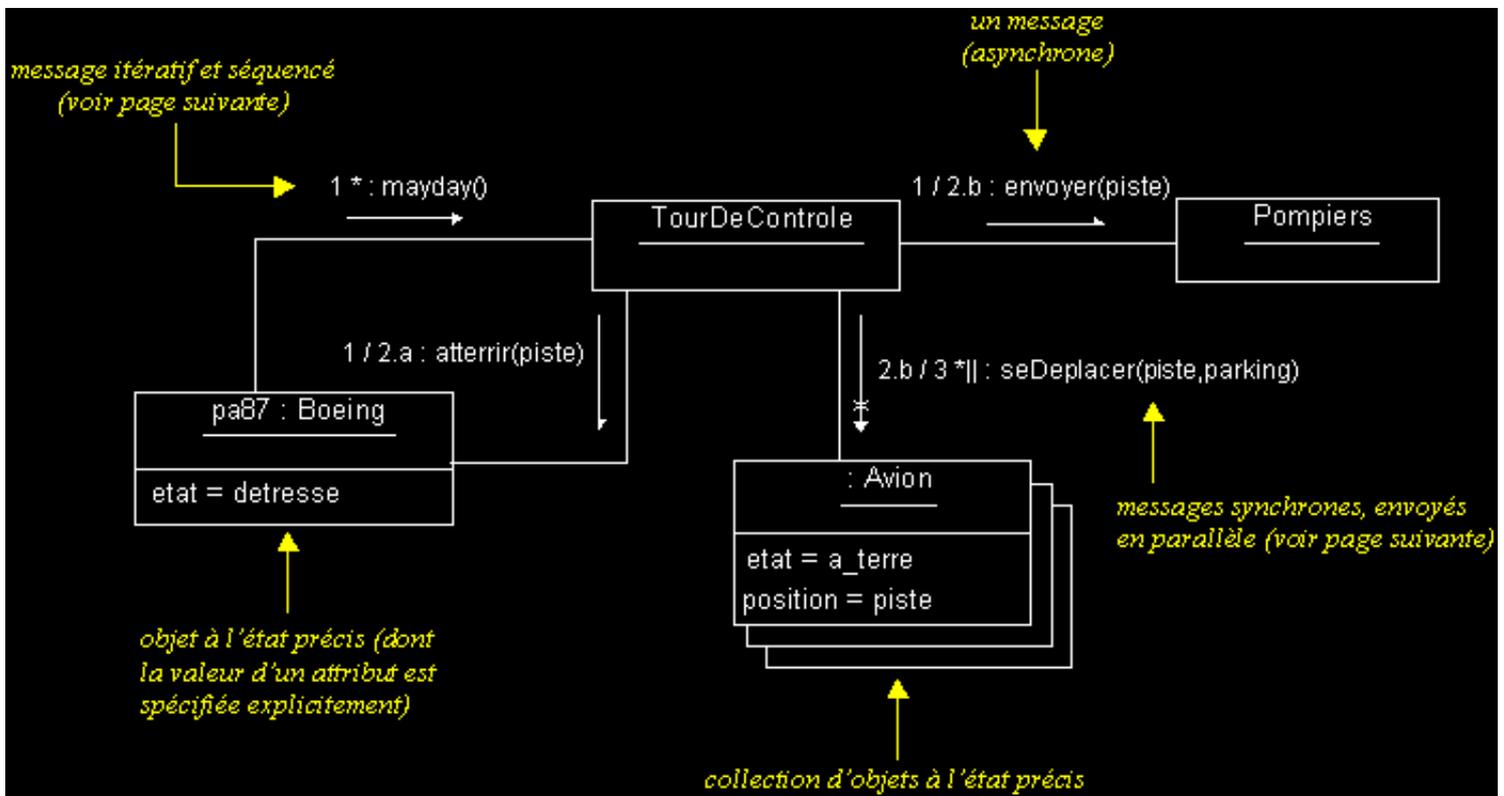
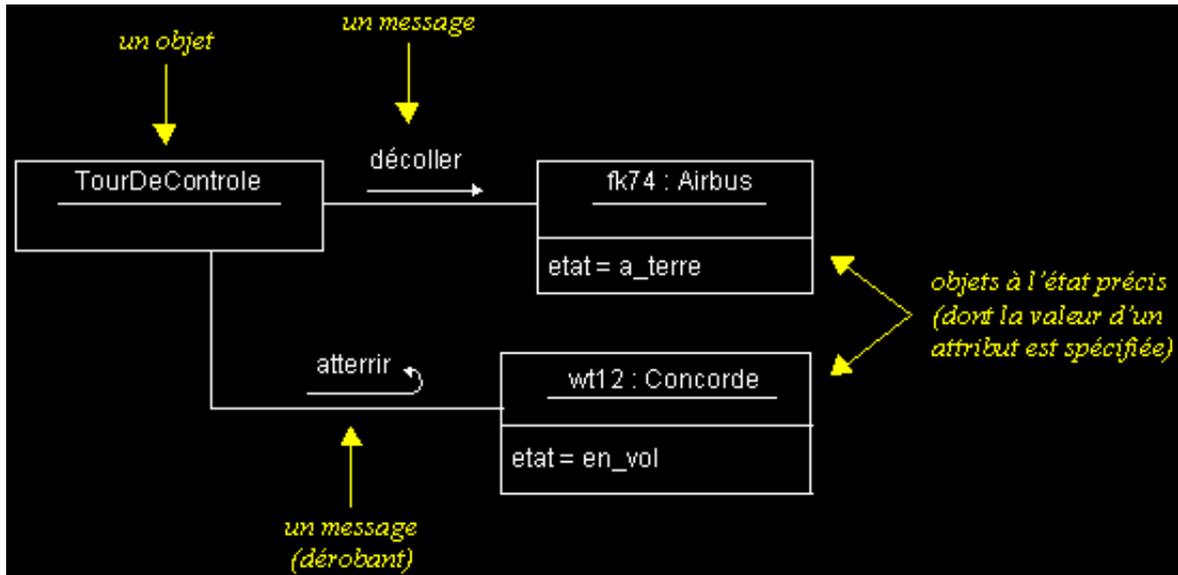
Relation d'extension :



II-D-2 - COLLABORATION ET MESSAGES

- 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 :



II-D-2-a - 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"] ] [ség] [ "*" [ "|" ] [ "iter" ] ] ":" [ r ":" ] msg (" [par]" )
```

- pré** : prédécesseurs (liste de numéros de séquence de messages séparés par une virgule ; voir aussi "séc").
 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.
- séc** : 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 précedence.
 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 :

Ce message a pour numéro de séquence '3'

```
3 : bonjour()
```

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

```
[heure = midi] 1 : manger()
```

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

```
1.3.6 * : ouvrir()
```

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

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

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

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

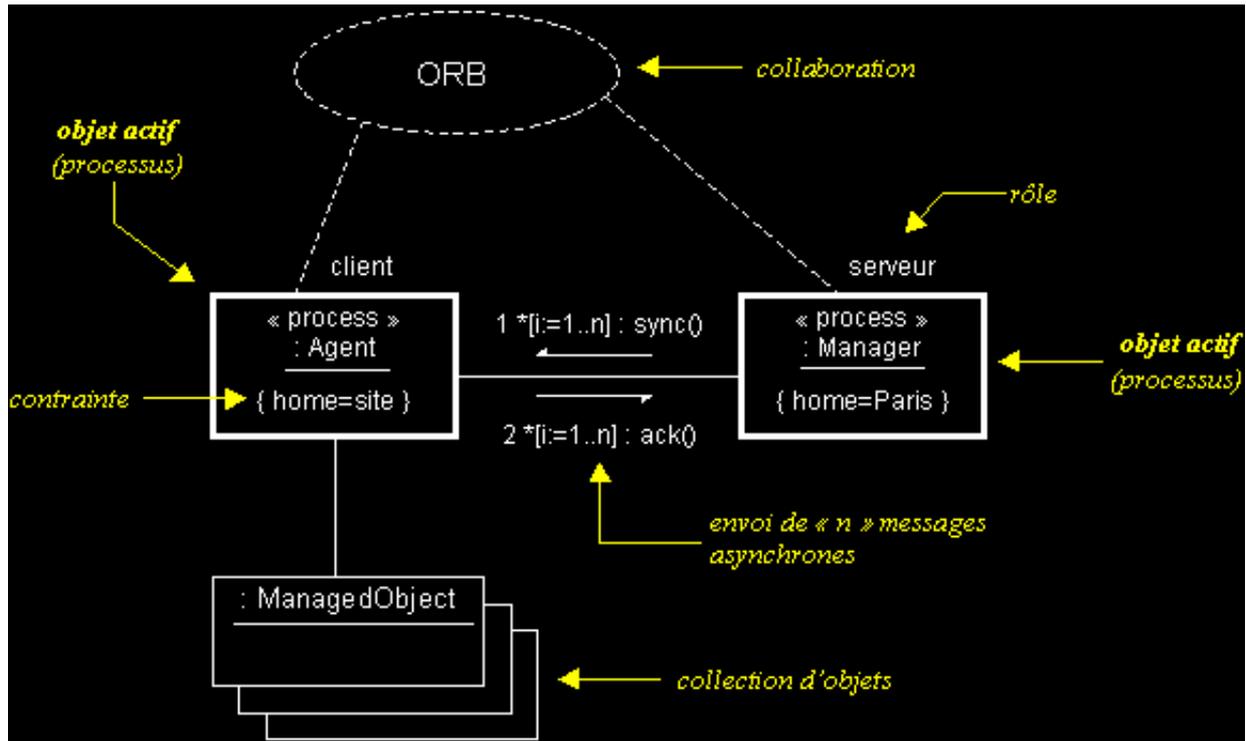
Ces messages ne seront envoyés qu'après l'envoi du message 1.3 et si la condition 'disk full' est réalisée. Si cela est le cas, les messages 1.7.a et 1.7.b seront envoyés simultanément. Plusieurs messages 1.7.a peuvent être envoyés.

```
1.3 / [disk full] 1.7.a * : deleteTempFiles()
```

```
1.3 / [disk full] 1.7.b : reduceSwapFile(20%)
```

II-D-2-b - 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.

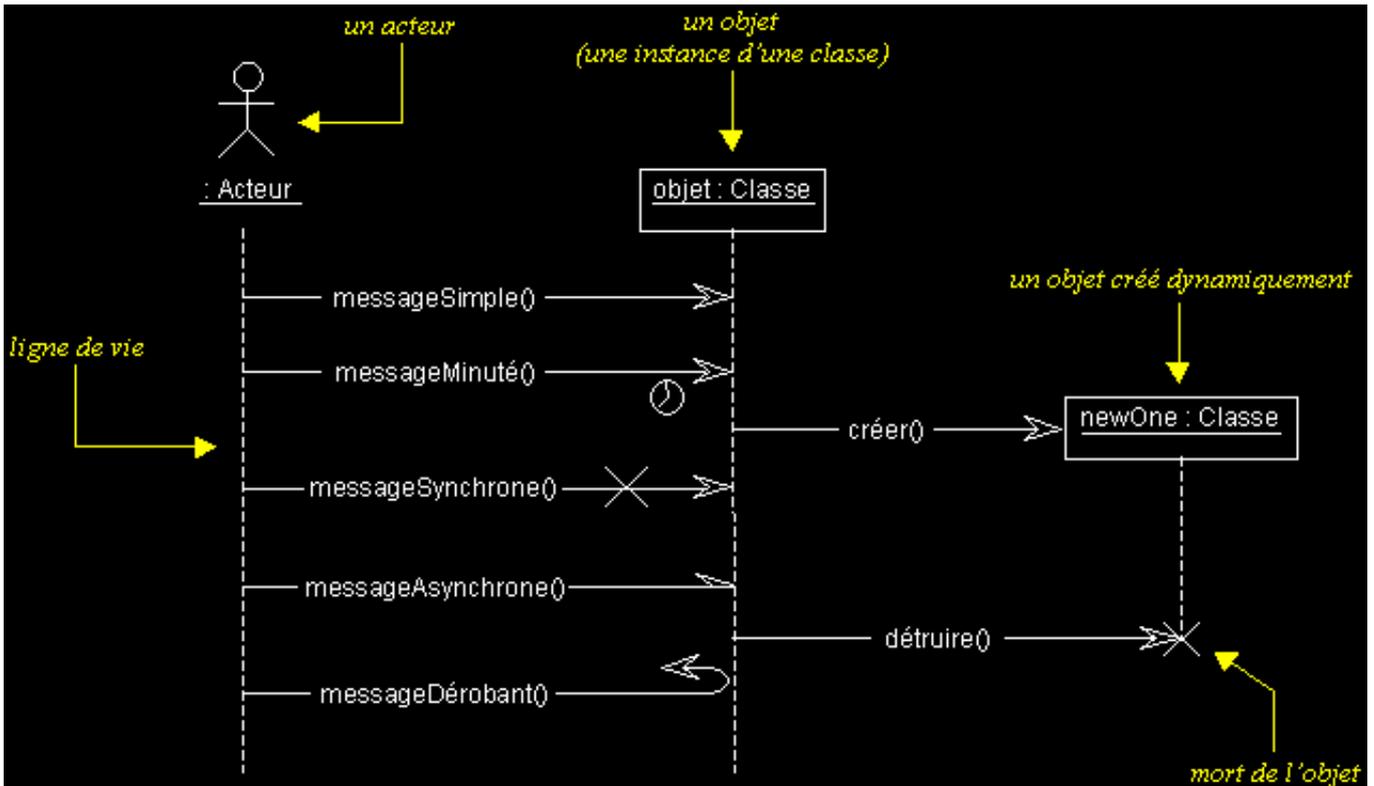


II-D-3 - DIAGRAMME DE SEQUENCE

II-D-3-a - 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 :



II-D-3-b - 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)
Bloquent 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
Bloquent l'expéditeur jusqu'à la 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.

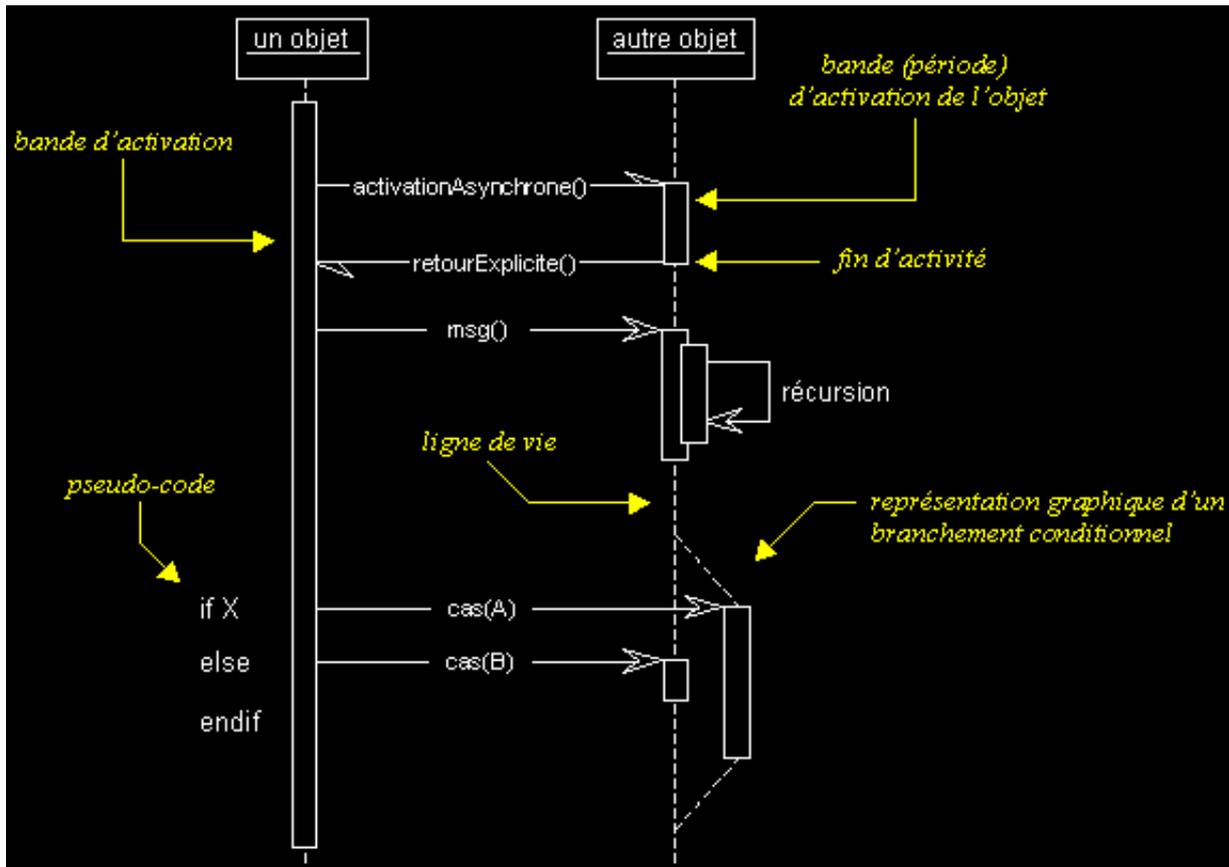
II-D-3-c - Activation d'un objet

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 :

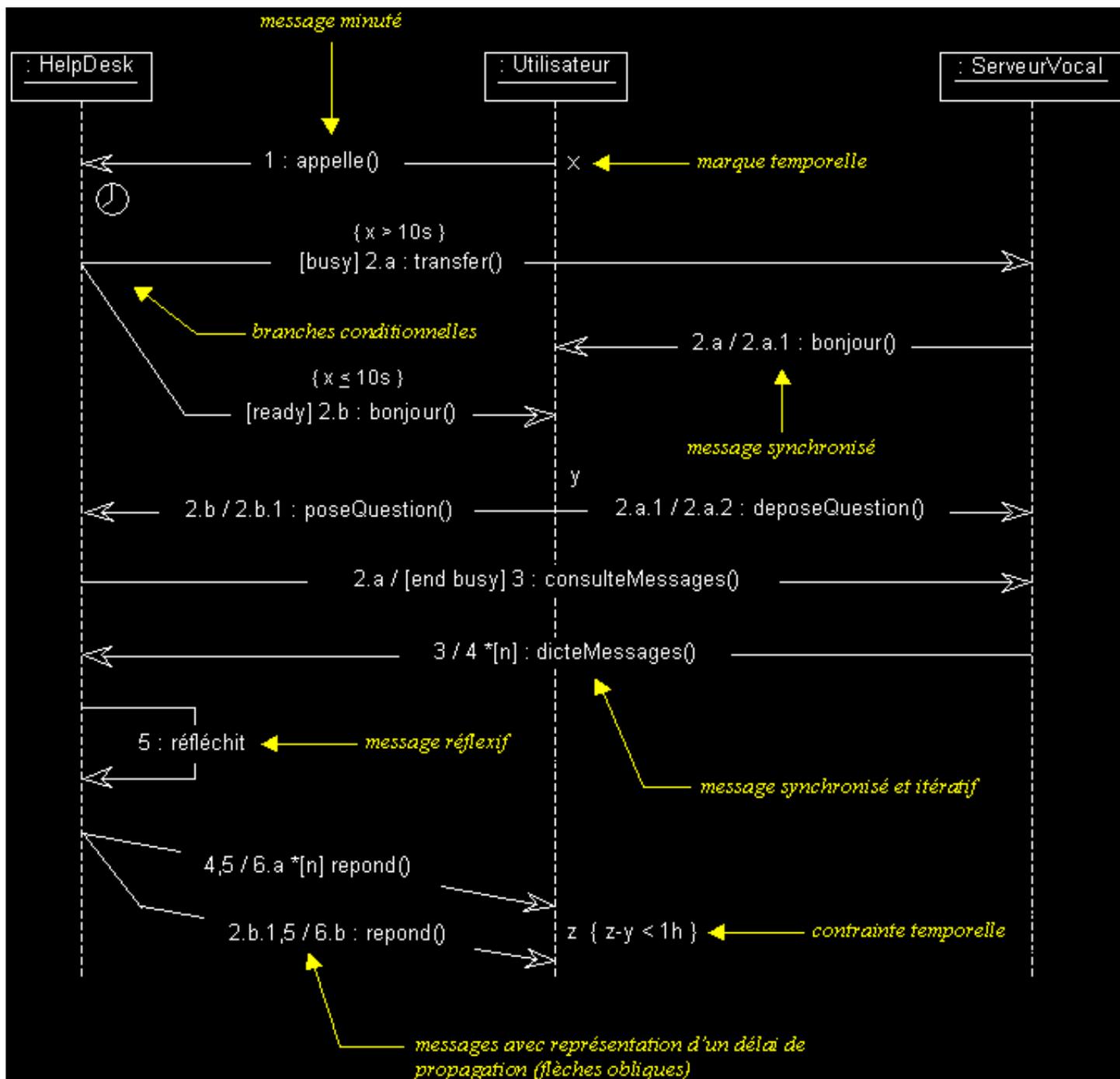


Commentaires :

- 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.

II-D-3-d - 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).

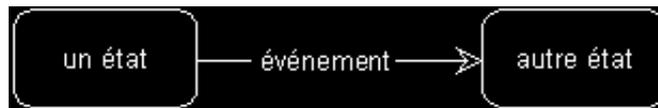
II-D-4 - DIAGRAMME D'ETATS-TRANSITIONS

II-D-4-a - Diagramme d'états-transitions : sémantique

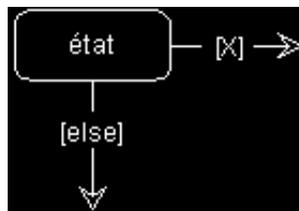
- Ce diagramme sert à représenter des automates d'états finis, sous forme de graphes d'états, reliés par des arcs orientés qui décrivent les transitions.

- 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 :



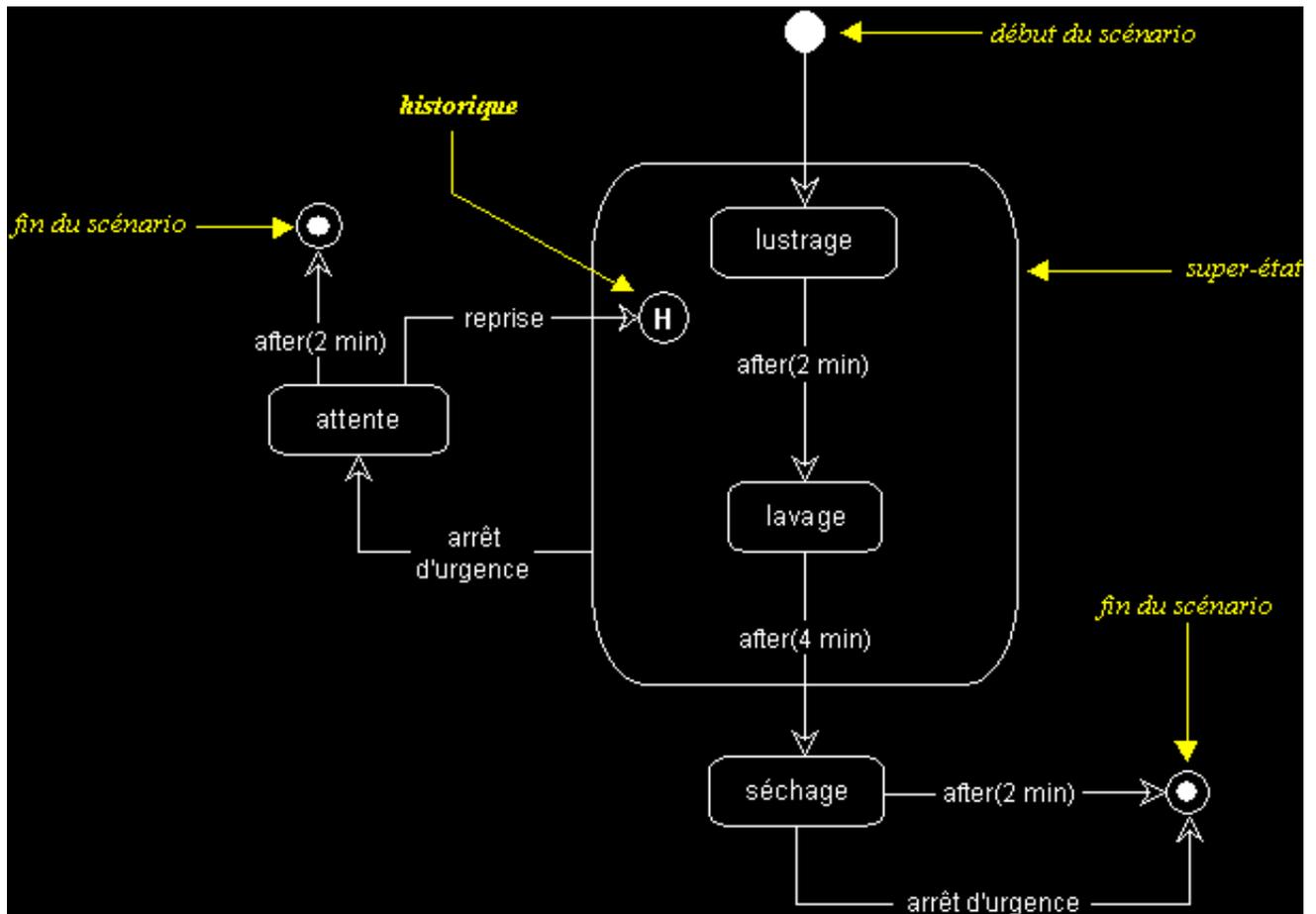
II-D-4-b - Super-Etat, historique et souches

- 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.

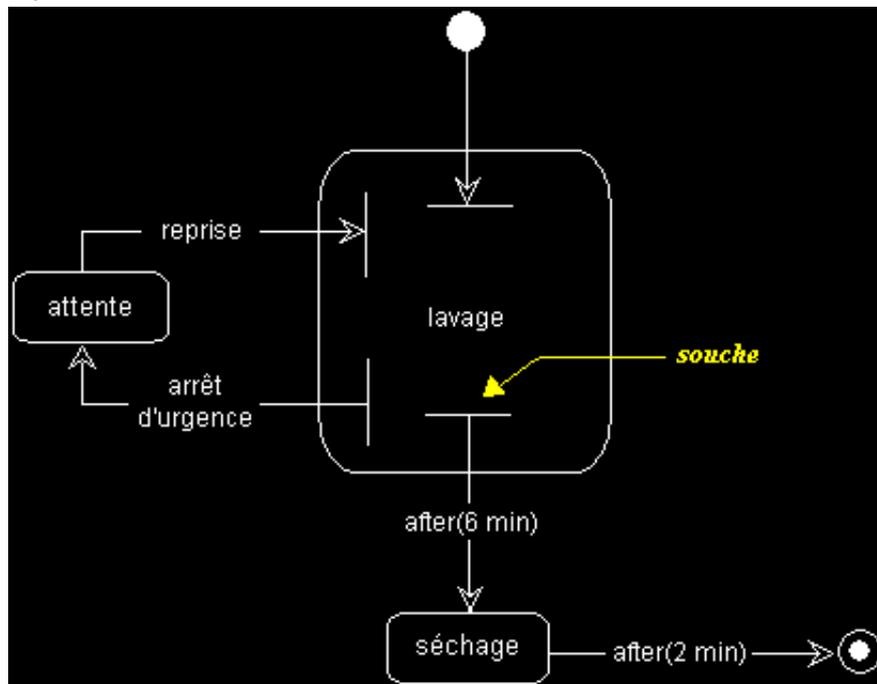
Exemple :

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).



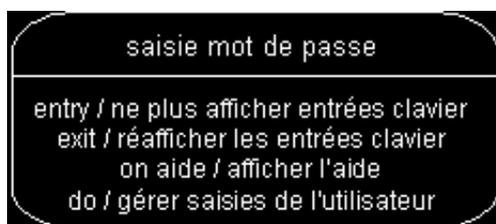
- **souches** : afin d'introduire plus d'abstraction dans un diagramme d'états-transitions complexe, il est possible de réduire la charge d'information, tout en matérialisant la présence de sous-états, à l'aide de souches, comme dans l'exemple ci-dessous.



II-D-4-c - Actions dans un état

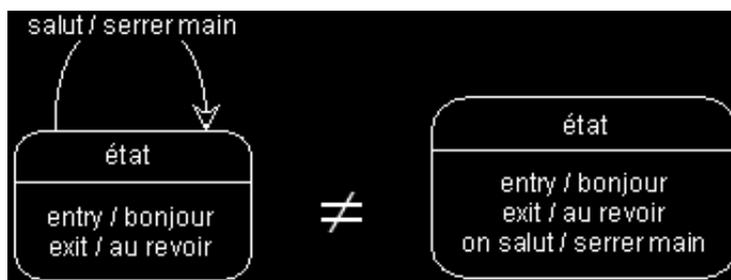
- On peut aussi associer une action à l'événement qui déclenche une transition.
La syntaxe est alors la suivante : événement / action
- Ceci exprime que la transition (déclenchée par l'événement cité) entraîne l'exécution de l'action spécifiée sur l'objet, à l'entrée du nouvel état.
Exemple : il pleut / 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 :



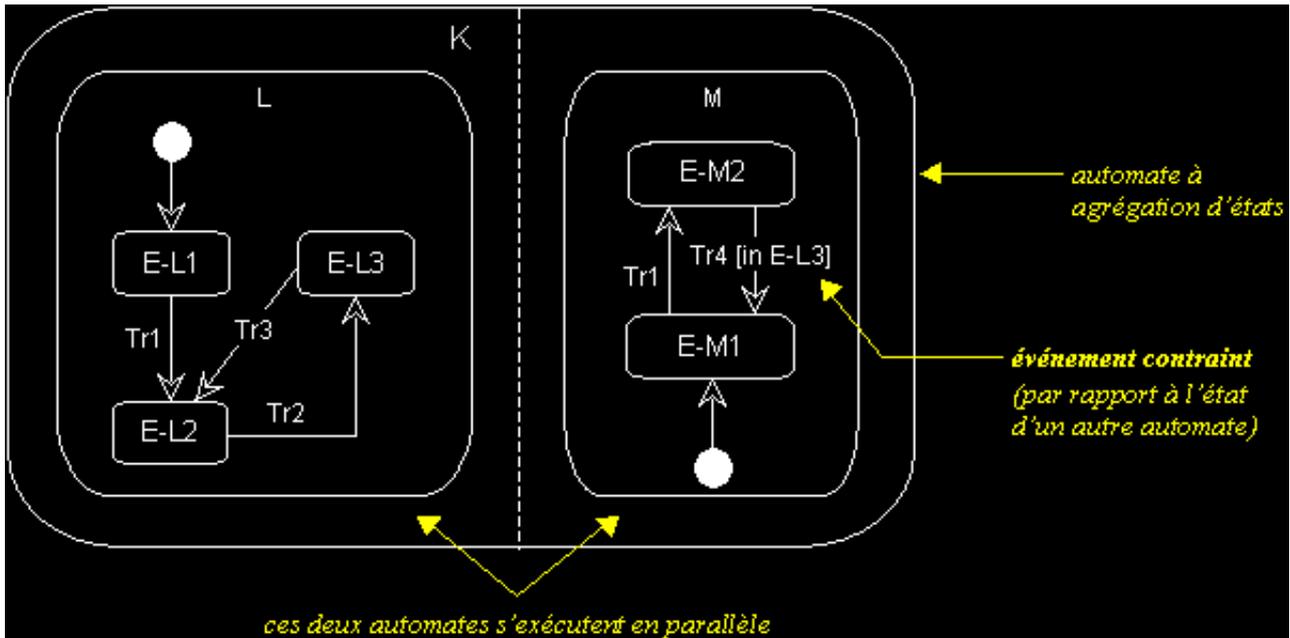
Remarque :

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 :



II-D-4-d - 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).

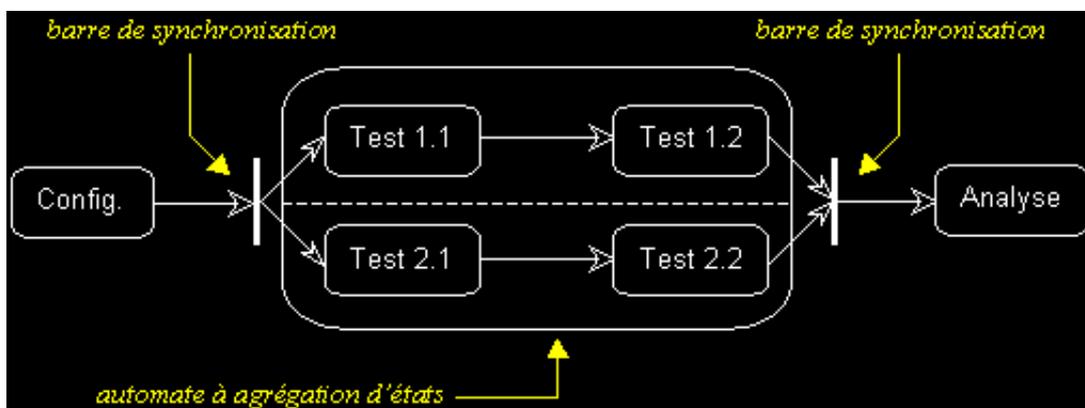
Après l'événement Tr1, K passe dans l'état composite (E-L2, E-M2). Par la suite, si l'événement Tr2 survient, K passe dans l'état composite (E-L3, E-M2). Si c'est Tr4 qui survient, M ne passe pas dans l'état E-M1, car cette transition est contrainte par l'état de L ("[in E-L3]").

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]").

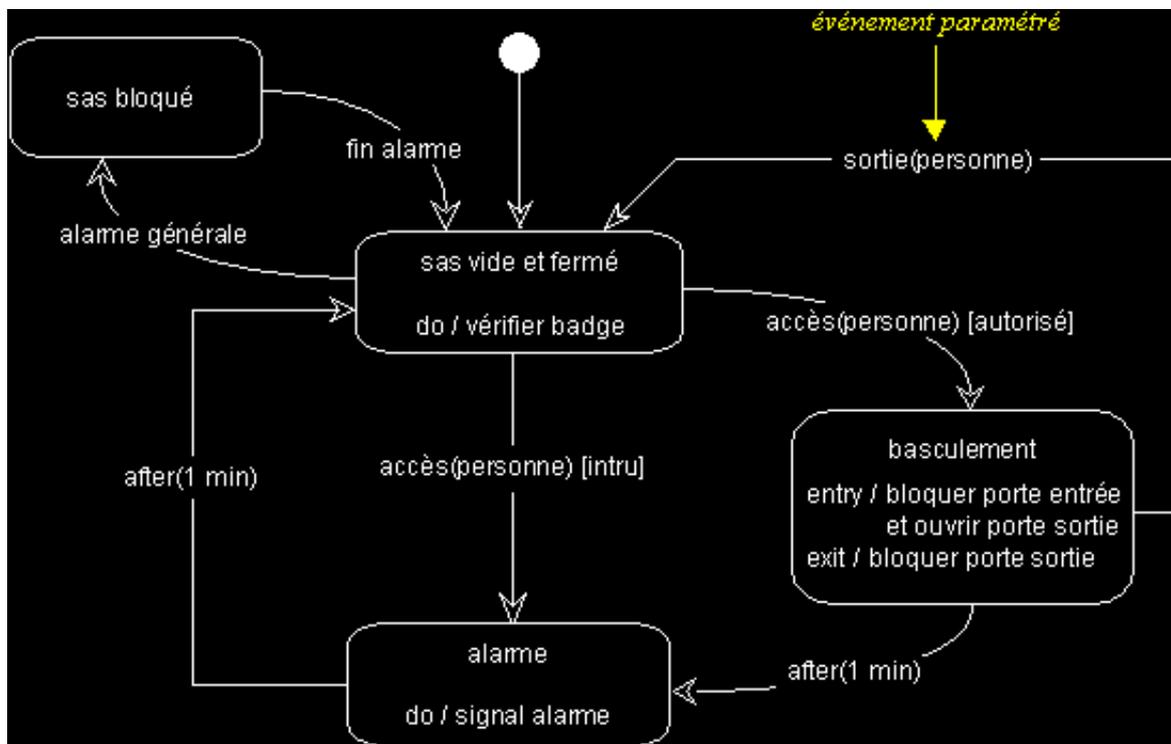
On peut aussi utiliser un symbole spécial : 'la barre de synchronisation'!

- 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.



II-D-4-e - Evénement paramétré

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



II-D-4-f - Echange de messages entre automates

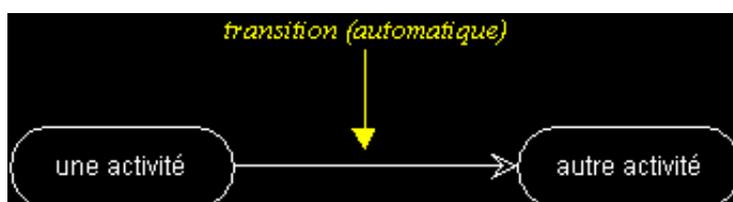
Il est aussi possible de représenter l'échange de messages entre automates dans un diagramme d'états-transitions. Cette notation particulière n'est pas présentée ici. Veuillez vous référer à "l'UML notation guide".

II-D-5 - DIAGRAMME D'ACTIVITES

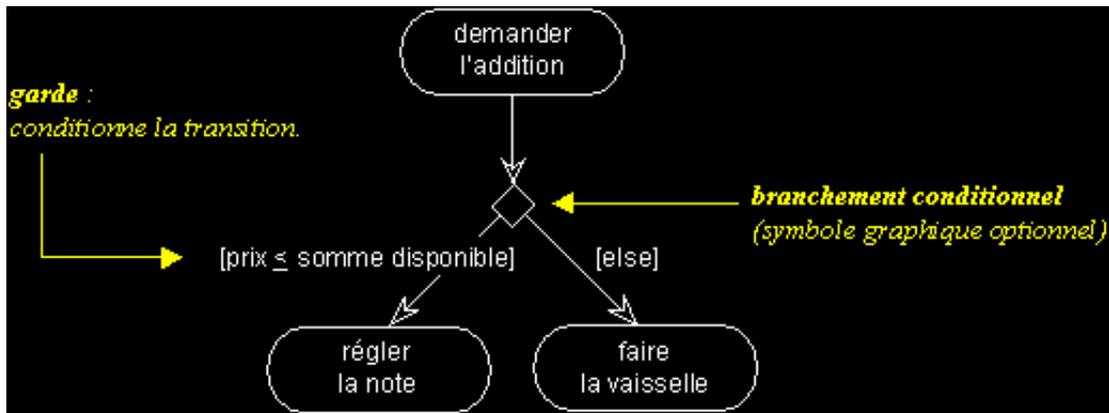
II-D-5-a - Diagramme d'activités : sémantique

- 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 des diagrammes 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 :



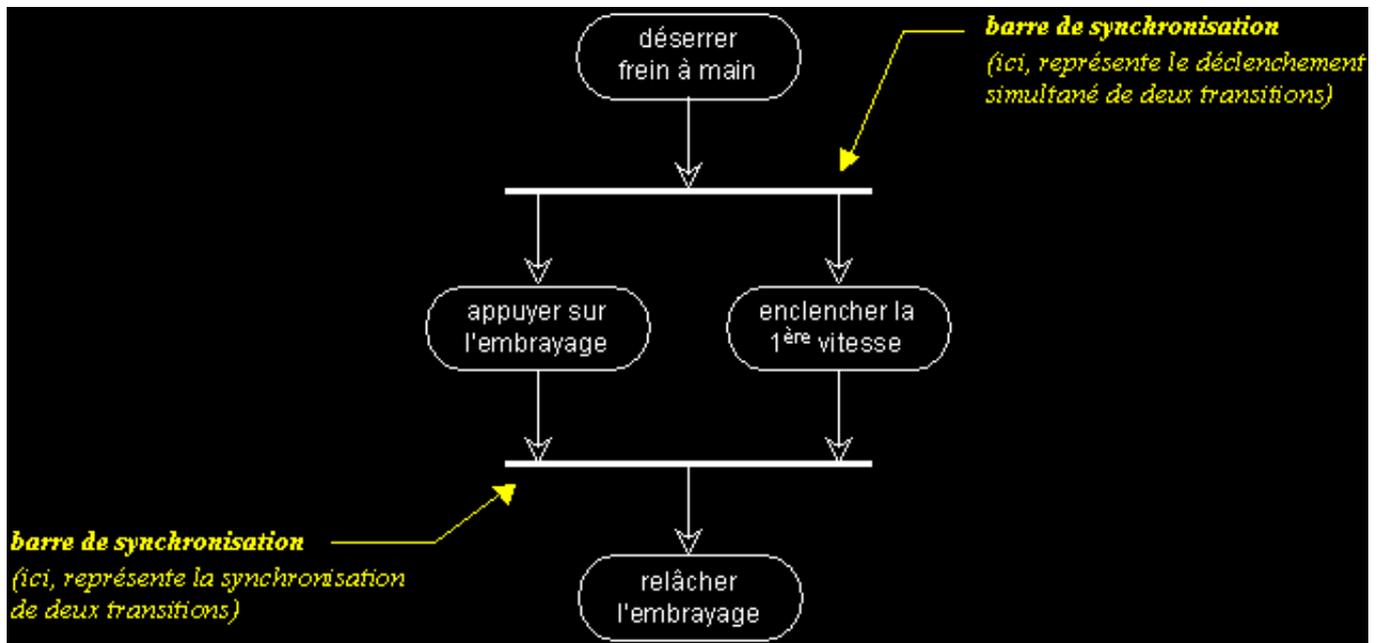
II-D-5-b - Synchronisation

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 :

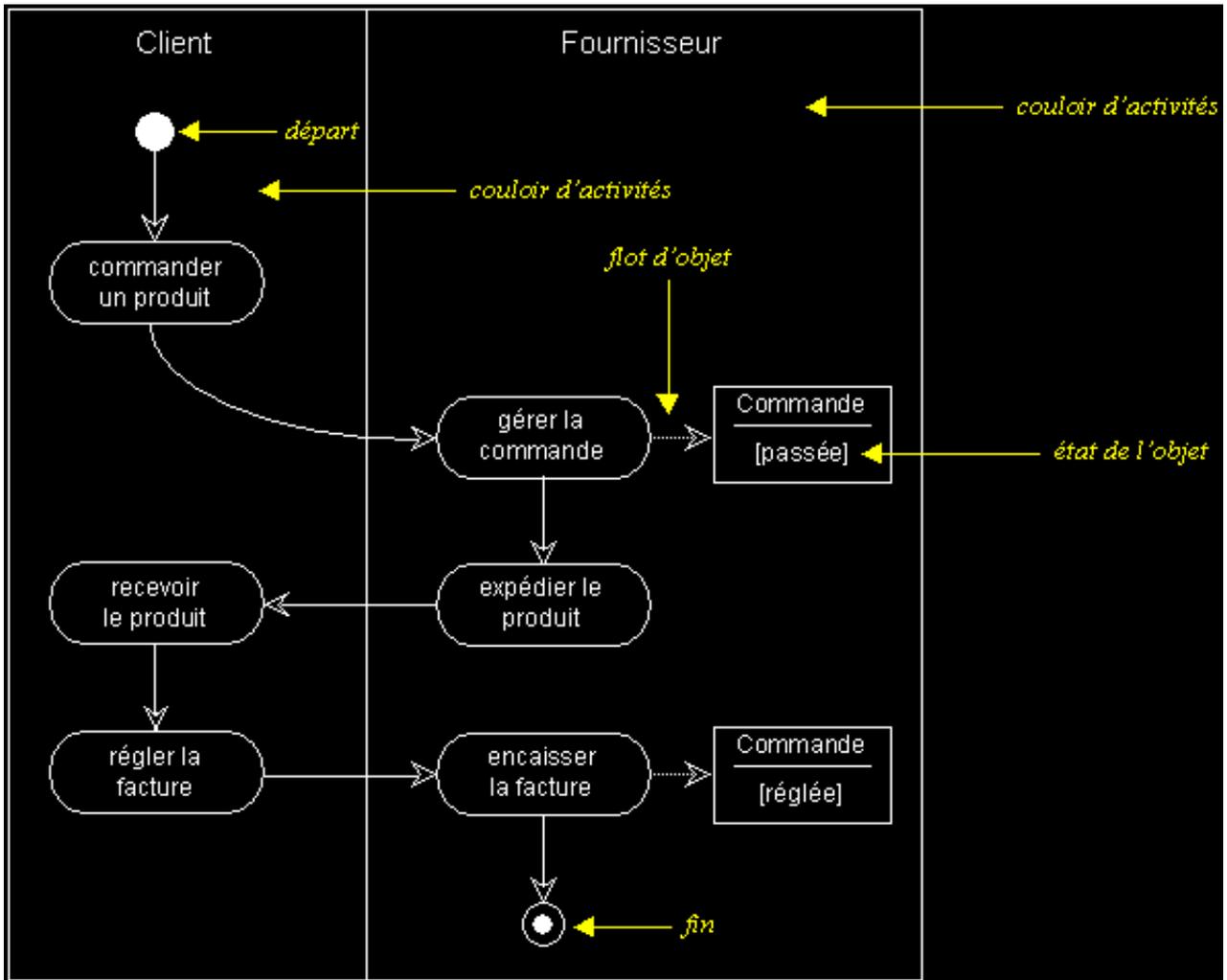
- Les transitions qui partent d'une barre de synchronisation ont lieu en même temps.
- On ne franchit une barre de synchronisation qu'après réalisation de toutes les transitions qui s'y rattachent.

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



II-D-5-c - 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.



II-E - Conclusion

II-E-1 - Programmer objet ?

- Programmer en C++ n'est pas "concevoir objet" !
 - Seule une analyse objet conduit à une solution objet (il faut respecter les concepts de base de l'approche objet).
 - Le langage de programmation est un moyen d'implémentation, il ne garantit pas le respect des concepts objet.
- UML n'est qu'un support de communication !
 - L'utilisation d'UML ne garantit pas le respect des concepts objet : à vous de vous en servir à bon escient.

💡 >> Concevoir objet, c'est d'abord concevoir un modèle qui respecte les concepts objet !
>> Le langage et UML ne sont que des outils.

II-E-2 - Utiliser UML ?

- Multipliez les vues sur vos modèles !
 - Un diagramme n'offre qu'une vue très partielle et précise d'un modèle.
 - Croisez les vues complémentaires (dynamiques / statiques).
- Restez simple !

- Utilisez les niveaux d'abstraction pour synthétiser vos modèles (ne vous limitez pas aux vues d'implémentation).
- Ne surchargez pas vos diagrammes.
- Commentez vos diagrammes (notes, texte...).
- Utilisez des outils appropriés pour réaliser vos modèles !