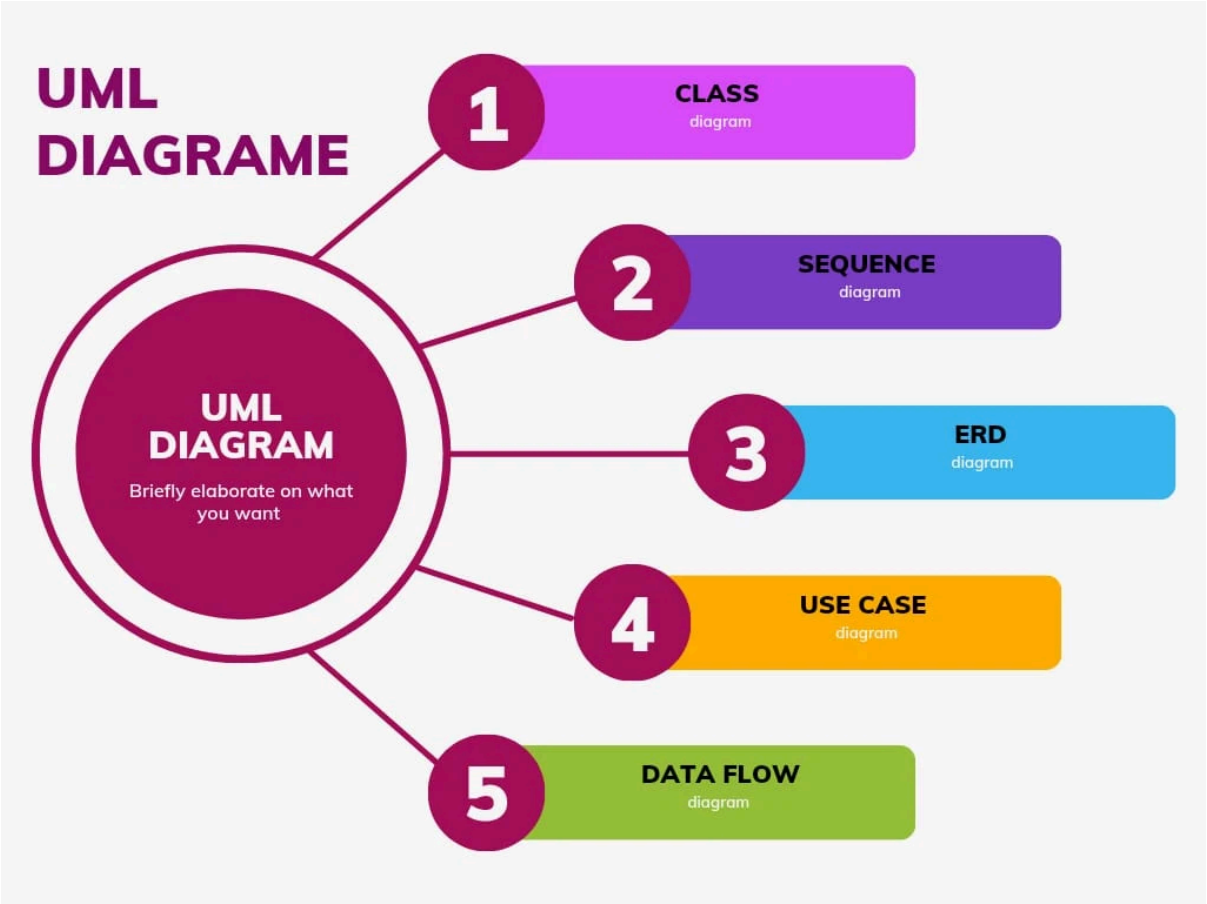
UML, le langage de modélisation objet unifié



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 ce cours d'introduction 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.

Ce cours d'introduction 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 ce cours d'introduction 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 pour donner les bases de UML.

A. PRÉSENTATION D'UML

Un peu d'Histoire

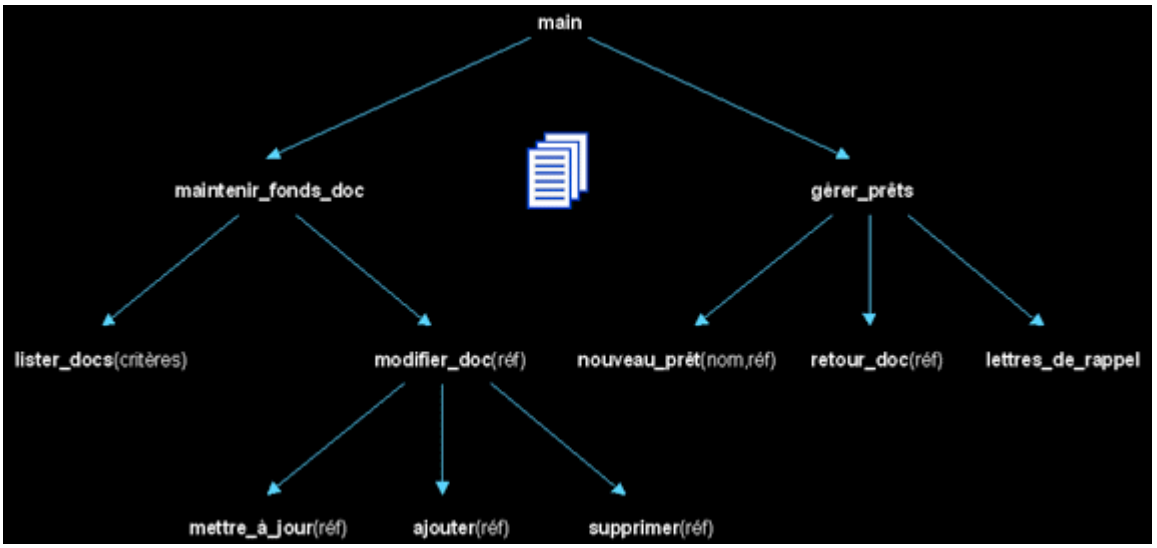
Approche fonctionnelle

La découpe fonctionnelle d'un problème informatique : une approche intuitive

La découpe fonctionnelle est une méthode essentielle en informatique pour comprendre et résoudre des problèmes complexes. Elle consiste à diviser un problème en plusieurs sous-problèmes plus petits et plus gérables. Cette approche permet de mieux organiser le développement du logiciel, de clarifier les responsabilités de chaque partie du système, et de faciliter la collaboration entre les membres de l'équipe de développement.

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

Pour illustrer cette méthode, prenons l'exemple d'un logiciel conçu pour gérer une bibliothèque. Ce logiciel doit permettre de gérer les livres, les utilisateurs, les emprunts et les retours. La découpe fonctionnelle de ce logiciel peut se faire de la manière suivante :



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.
En utilisant cette approche de découpe fonctionnelle, nous pouvons mieux organiser notre travail, clarifier les responsabilités et nous assurer que toutes les fonctionnalités nécessaires sont prises en compte. Cela nous aide également à identifier les interactions entre les différentes parties du système et à planifier le développement de manière plus efficace et structurée.

Le "plus" de l'approche fonctionnelle : la factorisation des comportements

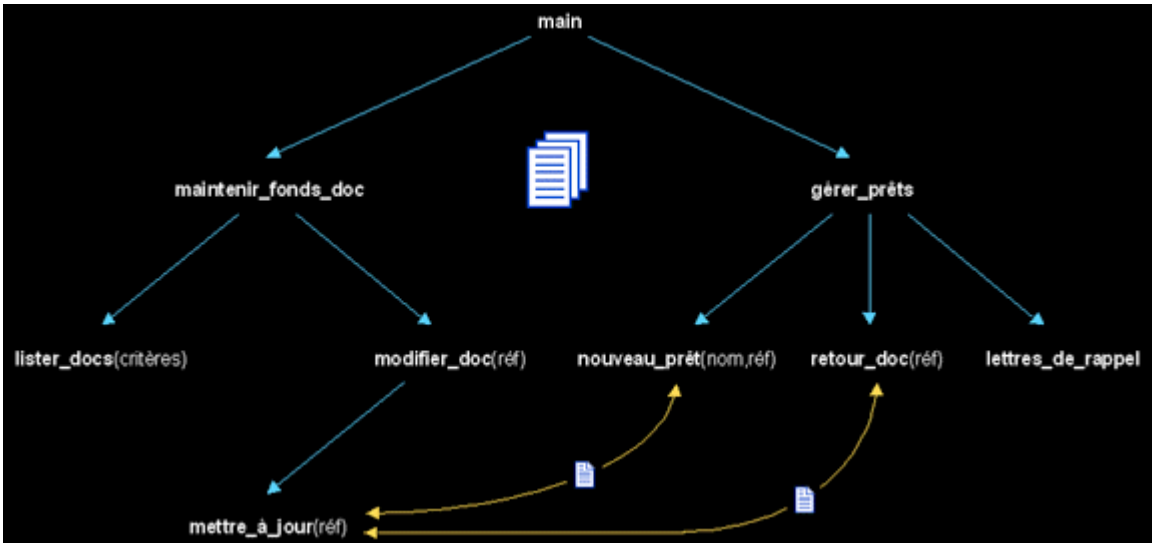
L'approche fonctionnelle en développement logiciel ne se limite pas simplement à découper un problème en parties plus petites. Elle permet également de factoriser certains comportements communs pour les réutiliser dans plusieurs parties du logiciel. Cette technique est essentielle pour créer des systèmes efficaces, maintenables et faciles à développer.

La factorisation des comportements

La factorisation des comportements consiste à identifier et isoler des fonctionnalités communes pour les rendre réutilisables à travers différentes parties du logiciel. Plutôt que de réécrire du code similaire à plusieurs endroits, on développe des fonctions génériques et flexibles qui peuvent être utilisées à plusieurs reprises. Cela réduit la redondance, facilite la maintenance et améliore la cohérence du code.

- Exemple de découpe fonctionnelle "intelligente" dans un logiciel de gestion de bibliothèque

Prenons à nouveau l'exemple d'un logiciel de gestion de bibliothèque. Voici comment nous pouvons factoriser certains comportements communs :



La factorisation des comportements est un "plus" indéniable de l'approche fonctionnelle. En développant des fonctions génériques réutilisables, on améliore l'efficacité, la maintenabilité et la cohérence du logiciel. Cette approche intelligente permet de réaliser des systèmes plus robustes et plus faciles à gérer.

Le revers de la médaille : maintenance complexe en cas d'évolution

Les défis de la maintenance avec la factorisation des comportements

Bien que la factorisation des comportements présente de nombreux avantages, elle n'est pas sans inconvénients. L'un des principaux défis est la complexité accrue de la maintenance du logiciel. Lorsqu'un logiciel est conçu avec des fonctions interdépendantes et génériques, une mise à jour ou une modification à un point précis peut avoir des répercussions en cascade sur d'autres parties du système.

Impact en cascade des mises à jour

Lorsque les fonctions sont fortement interdépendantes, une simple mise à jour peut affecter plusieurs autres fonctions de manière inattendue. Par exemple, si une fonction de validation des données est modifiée pour inclure de nouvelles règles, toutes les parties du logiciel qui utilisent cette fonction devront être testées pour s'assurer qu'elles fonctionnent toujours

correctement avec les nouvelles règles. Ce phénomène est souvent appelé "effet domino" et peut rendre la maintenance du logiciel plus complexe et plus coûteuse.

Utilisation de fonctions génériques et de structures de données ouvertes

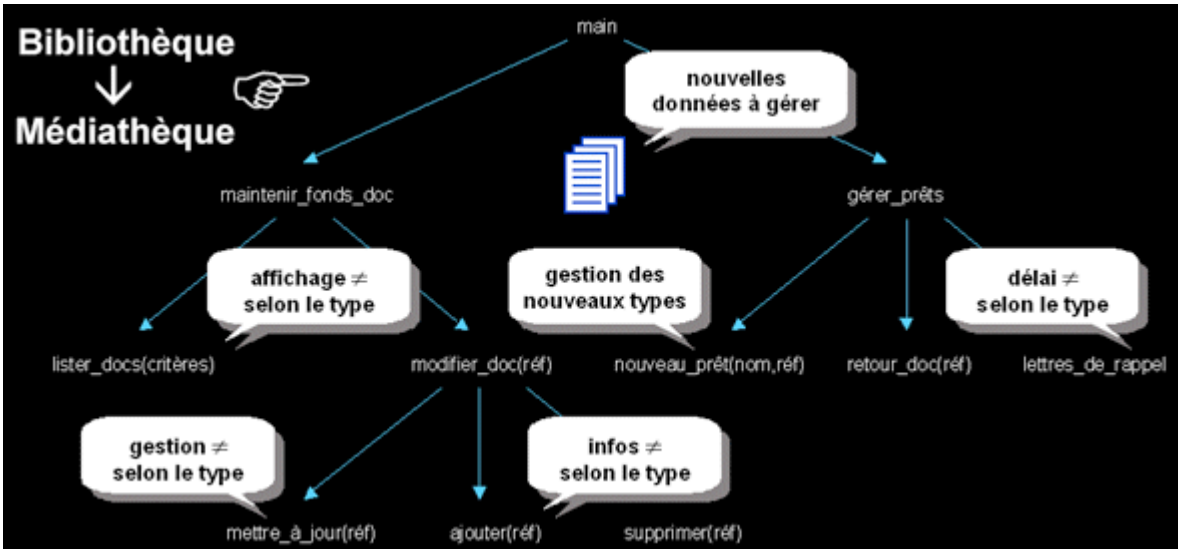
Pour minimiser l'impact des modifications, il est essentiel d'utiliser des fonctions génériques et des structures de données ouvertes. Cela signifie que les fonctions doivent être conçues pour être aussi flexibles et adaptables que possible. Par exemple, une fonction de validation des données pourrait être conçue pour accepter des paramètres configurables qui définissent les règles de validation, plutôt que d'avoir ces règles codées en dur. De même, l'utilisation de structures de données ouvertes permet de gérer des données variées sans nécessiter de modifications majeures du code.

Complexité accrue de l'écriture et de la maintenance du logiciel

Cependant, le respect de ces contraintes de flexibilité et d'ouverture rend l'écriture du logiciel initialement plus complexe. Les développeurs doivent anticiper une variété de scénarios d'utilisation et concevoir leurs fonctions de manière à pouvoir s'adapter facilement à ces scénarios. Cette complexité se traduit également par une maintenance plus difficile, car les développeurs doivent comprendre et gérer cette flexibilité et cette interopérabilité lors de chaque mise à jour ou modification du logiciel.

Évolution majeure du logiciel

En cas d'évolution majeure du logiciel, comme le passage de la gestion d'une bibliothèque à celle d'une médiathèque, les défis de la maintenance sont encore plus prononcés. Même si la structure générale du logiciel reste valide, la multiplication des points de maintenance due au chaînage des fonctions rend l'adaptation très laborieuse. Chaque fonction générique et chaque structure de données doivent être revisitées et ajustées pour répondre aux nouvelles exigences. Par exemple :



- **Ajout de nouvelles catégories de médias** : La gestion d'une médiathèque implique non seulement des livres, mais aussi des DVD, des CD, des magazines, etc. Chaque type de média a ses propres caractéristiques et règles de gestion.
- **Modification des fonctions de recherche** : Les critères de recherche devront être étendus pour inclure des filtres spécifiques à chaque type de média.
- **Mise à jour des notifications** : Les notifications devront être adaptées pour couvrir des événements spécifiques à chaque type de média, comme des rappels pour le retour de DVD ou des notifications de nouvelles acquisitions de magazines.

Adaptation globale du logiciel

Pour ces raisons, une évolution majeure nécessite souvent une révision et une mise à jour globale du logiciel. Chaque fonction et chaque interaction entre les fonctions doivent être soigneusement examinées et ajustées pour s'assurer qu'elles répondent aux nouvelles exigences. Cette tâche est particulièrement complexe et demande beaucoup de temps et d'efforts.

Bien que la factorisation des comportements offre de nombreux avantages en termes de réutilisabilité et de réduction de la redondance,

elle complique également la maintenance du logiciel, en particulier lors de mises à jour ou d'évolutions majeures. Les développeurs doivent trouver un équilibre entre la création de fonctions génériques et la gestion de la complexité induite par l'interdépendance des fonctions. Une planification soigneuse, des tests rigoureux et une documentation claire sont essentiels pour gérer efficacement ces défis.

La séparation des données et des traitements : le piège !

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

Lorsqu'on souhaite faire évoluer une application de gestion de bibliothèque pour gérer une médiathèque, il est nécessaire de prendre en compte de nouveaux types d'ouvrages, tels que des cassettes vidéo, des CD-ROM, etc. Cette évolution implique deux types de modifications majeures :

1. **Évolution des structures de données** : Les structures de données manipulées par les fonctions doivent être étendues pour inclure les nouveaux types de documents.
2. **Adaptation des traitements** : Les fonctions, initialement conçues pour manipuler un seul type de document (les livres), doivent être modifiées pour gérer les nouveaux types de documents.

Cela signifie que toutes les portions de code qui utilisent la base documentaire doivent être modifiées pour gérer les données et les actions spécifiques à chaque type de document.

- Exemple concret : Modification des lettres de rappel

Prenons l'exemple de la fonction qui génère les "lettres de rappel". Une lettre de rappel est un avis envoyé automatiquement aux personnes qui tardent à rendre un ouvrage emprunté. Si l'on souhaite que le délai avant rappel varie selon le type de document emprunté, il faut implémenter une règle de calcul distincte pour chaque type de document.

En pratique, cela signifie que presque toute l'application devra être adaptée pour gérer les nouvelles données et effectuer les traitements correspondants. Et cette adaptation devra être répétée chaque fois qu'un nouveau type de document sera ajouté !

- Exemple de code en C

Voici un exemple de code en C illustrant ce problème :

```c
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 (int i = 0; i < NB_DOCS; i++) {
        if (DOC[i].est_emprunte) {
            int delai_avant_rappel;
            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;
    }
    /* ... */
}
```

En séparant les données et les traitements, il est facile de tomber dans le piège d'une complexité accrue lors des évolutions du logiciel. Chaque ajout de nouveau type de document nécessite des modifications étendues dans tout le code, rendant la maintenance difficile et coûteuse. Une approche orientée objet, comme celle illustrée en Java, peut aider à atténuer certains de ces problèmes en encapsulant les comportements spécifiques dans des classes dédiées. Cependant, même avec cette approche, il est important de bien planifier et structurer le code pour faciliter les évolutions futures.

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 à centraliser dans les structures de données les valeurs qui leur sont propres.

Centralisation des caractéristiques dans les structures de données

En regroupant les caractéristiques propres à chaque type de document directement dans la structure de données, nous simplifions grandement le code. Par exemple, au lieu d'avoir des conditions pour déterminer le délai avant rappel en fonction du type de document, nous stockons directement cette information dans chaque instance de `Document`.

- Voici comment cela pourrait être implémenté en C :

```
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 (int 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]);
            }
        }
    }
}
```

Logique derrière cette solution

Cette approche est plus logique car elle encapsule les propriétés spécifiques aux types de documents directement dans les instances de `Document`. Le "délai avant édition d'une lettre de rappel" est bien une caractéristique propre à tous les ouvrages gérés par notre application. En centralisant cette information, nous rendons le code plus lisible, plus facile à maintenir et moins sujet aux erreurs.

Limites de cette solution

Bien que cette solution soit une amélioration par rapport à l'approche précédente, elle n'est pas encore optimale. Par exemple, elle ne résout pas entièrement le problème de la gestion des nouveaux types de documents. Chaque fois qu'un nouveau type de document est ajouté, il faut toujours mettre à jour les structures de données et les fonctions associées. Une meilleure solution pourrait impliquer l'utilisation de techniques plus avancées, telles que la programmation orientée objet ou des modèles de conception comme le **Pattern Strategy**, pour encore mieux gérer l'évolution et la maintenance du code.

Centralisation des données et des traitements : une stratégie pour simplifier la maintenance du code

Pourquoi centraliser ?

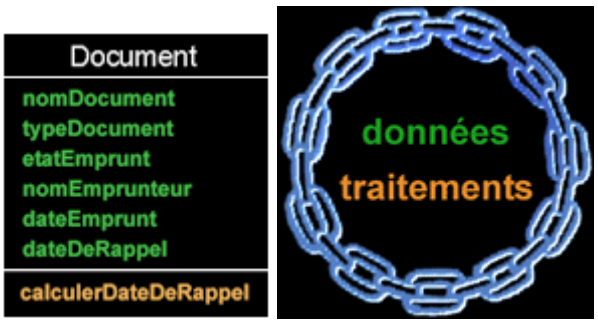
Lorsque nous développons un logiciel, il est crucial de trouver des moyens de le rendre aussi facile à maintenir que possible. Une méthode élégante pour y parvenir est de regrouper toutes les informations concernant un type de données spécifique ainsi que les actions qui lui sont associées dans un seul endroit.

En quoi cela nous aide-t-il ?

Imaginez que vous ayez une boîte à outils où chaque outil est rangé dans un compartiment dédié avec les instructions d'utilisation juste à côté. Lorsque vous avez besoin d'un outil particulier, vous savez immédiatement où le trouver et comment l'utiliser. C'est exactement ce que nous faisons en regroupant les données et les traitements dans une même unité physique dans notre code.

- Un exemple concret :

Prenons l'exemple de notre application de gestion de bibliothèque. Si nous voulons calculer le délai avant rappel pour un document emprunté, nous n'avons plus besoin de chercher à travers tout notre code. Au lieu de cela, nous avons une fonction dédiée qui fait ce calcul pour nous, juste à côté de la structure de données qui décrit le document.



Avantages pratiques :

- **Facilité de maintenance** : En regroupant les données et les traitements associés, nous réduisons les points de maintenance

dans le code. Si nous devons faire une modification, nous savons exactement où chercher.

- **Accès rapide à l'information** : Tout ce dont nous avons besoin est regroupé au même endroit, ce qui rend l'accès à l'information beaucoup plus facile. Cela nous fait gagner du temps et réduit le risque d'erreurs.
- **Lisibilité accrue** : Le code devient plus clair et plus facile à comprendre pour nous et pour les autres développeurs qui pourraient travailler sur le projet à l'avenir.

Centraliser les données d'un type et les traitements associés dans une même unité physique est une stratégie puissante pour simplifier la maintenance du code et améliorer son évolutivité. En adoptant cette approche, nous rendons notre code plus robuste, plus lisible et plus facile à gérer pour tous les membres de notre équipe de développement.

Approche objet

Comprendre le concept d'objet : une approche fondamentale en programmation

Qu'est-ce qu'un objet ?

Lorsque nous modifions notre logiciel de gestion de médiathèque, nous passons d'une simple structure de données manipulée par des fonctions à une entité plus complexe et autonome. Cette entité, qui regroupe un ensemble de propriétés cohérentes et de traitements associés, est appelée un objet.

Les caractéristiques d'un objet :

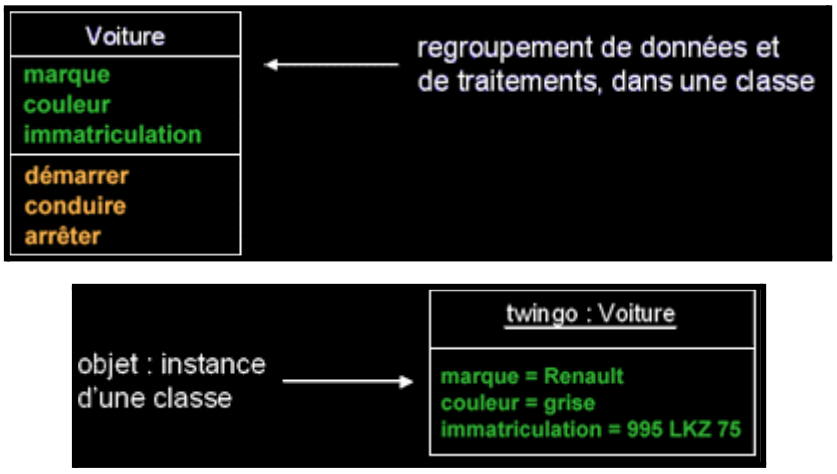
- **Identité** : Chaque objet a une identité unique qui le distingue des autres. C'est comme si chaque objet avait son propre nom qui le rendait reconnaissable.

- **Attributs** : Les attributs sont les propriétés ou les caractéristiques de l'objet. Ils décrivent l'état actuel de l'objet. Par exemple, dans notre gestion de médiathèque, les attributs d'un livre pourraient inclure son titre, son auteur et sa date de publication.
- **Méthodes** : Les méthodes sont les actions ou les comportements associés à l'objet. Elles définissent ce que l'objet peut faire. Par exemple, une méthode pour un livre pourrait être "emprunter" ou "retourner".

La relation entre les objets et les classes :

Un objet est une instance d'une classe. Une classe est un plan ou un modèle pour créer des objets. Elle définit les attributs et les méthodes communs à tous les objets de cette classe. Par exemple, la classe "Livre" pourrait définir les attributs tels que le titre et l'auteur, ainsi que les méthodes telles que "emprunter" et "retourner".

- Exemple pour une voiture :



Comprendre le concept d'objet est essentiel en programmation. Les objets sont des entités autonomes dotées d'identité, d'attributs et de méthodes. Ils permettent de modéliser le monde réel de manière organisée et structurée. En utilisant des classes pour définir des types d'objets, nous pouvons créer des programmes plus flexibles, plus modulaires et plus faciles à maintenir.

Découverte des concepts clés de l'approche objet

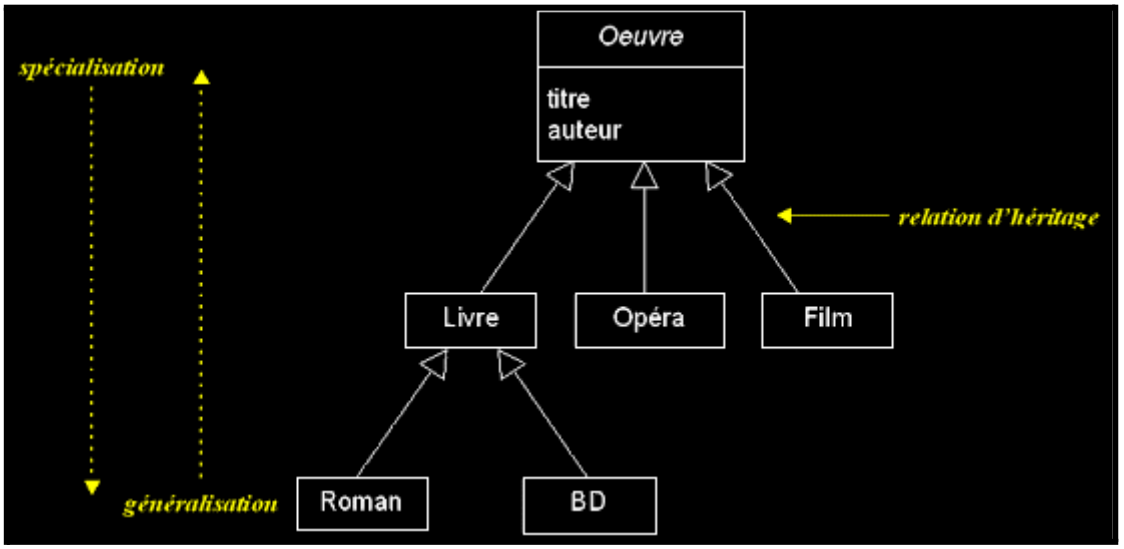
Encapsulation : protéger les détails

L'encapsulation consiste à cacher les détails d'implémentation d'un objet derrière une interface claire et bien définie. Cette interface définit les services accessibles aux utilisateurs de l'objet. L'encapsulation facilite l'évolution d'une application en stabilisant l'utilisation des objets : nous pouvons modifier l'implémentation des attributs d'un objet sans altérer son interface. De plus, elle garantit l'intégrité des données en interdisant l'accès direct aux attributs, obligeant l'utilisation de méthodes spéciales appelées accesseurs.

Héritage et polymorphisme : construire des relations et augmenter la flexibilité

L'héritage est un mécanisme puissant qui permet à une classe d'hériter des propriétés d'une autre classe. Il permet de spécialiser ou généraliser des classes pour créer des hiérarchies. Par exemple, une classe "Voiture" peut hériter des caractéristiques d'une classe "Véhicule". L'héritage évite la duplication de code et encourage la réutilisation.

- Exemple d'une hiérarchie de classes :

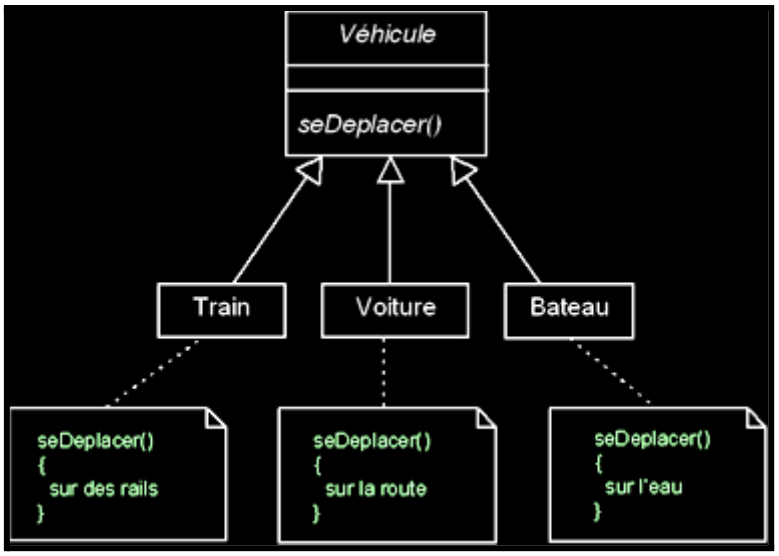


Le polymorphisme, quant à lui, représente la capacité d'une méthode à s'appliquer à des objets de classes différentes. Il augmente la généricité du code en permettant à une même méthode de traiter différents types d'objets de manière transparente. Par exemple, une méthode "seDeplacer()" peut être utilisée pour faire avancer un train, une voiture ou un bateau.

- Polymorphisme, exemple :

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

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

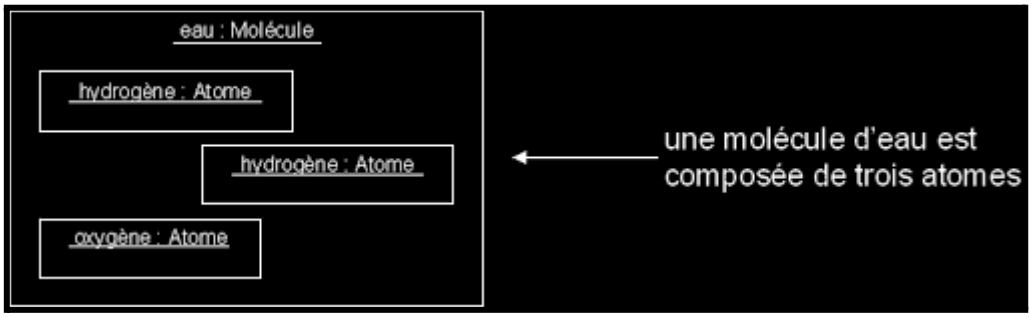


Agrégation : composer des objets complexes

L'agrégation est une relation entre deux classes, indiquant que les objets d'une classe sont des composants de l'autre classe. Cela permet de construire des objets complexes à partir d'objets de base. Par exemple, une

classe "eau : Molécule" peut agréger des objets de type "Atome" tels que l'hydrogène et l'oxygène pour former un ensemble cohérent.

- Agrégation, exemple :



Comprendre les concepts fondamentaux de l'approche objet

Les piliers de l'approche objet :

L'approche objet repose sur plusieurs concepts fondamentaux :

- **Objet et classe** : Un objet est une entité autonome qui possède des caractéristiques (attributs) et des actions (méthodes). Les objets appartiennent à des classes, qui définissent leurs propriétés communes.
- **Encapsulation** : C'est le fait de cacher les détails d'implémentation d'un objet derrière une interface claire. L'interface expose les services offerts par l'objet tout en masquant sa complexité interne.
- **Héritage** : Ce mécanisme permet à une classe d'hériter des propriétés d'une autre classe. Il favorise la réutilisation du code en permettant la spécialisation et la généralisation des classes.
- **Agrégation** : C'est une relation entre deux classes où les objets d'une classe sont des composants de l'autre classe. Cela permet de construire des objets complexes à partir d'objets plus simples.

Historique de l'approche objet :

Les concepts objet ont été développés au fil du temps et ont été éprouvés sur le terrain. Des langages comme Simula et Smalltalk ont été parmi les premiers à les mettre en œuvre dans les années 1960 et 1970. Depuis, de nombreux autres langages ont suivi, dont le célèbre C++ qui a été normalisé dans les années 1980.

Les avantages de l'approche objet :

L'approche objet a été conçue pour faciliter le développement d'applications complexes. Les outils et langages orientés objet sont fiables et performants, ce qui en fait une solution idéale pour de nombreux projets logiciels.

Les défis de l'approche objet :

Malgré ses nombreux avantages, l'approche objet peut être moins intuitive que l'approche fonctionnelle pour certaines personnes. Elle nécessite une grande rigueur dans l'analyse et la conception, ainsi qu'un vocabulaire précis pour éviter les ambiguïtés.

Les remèdes aux inconvénients :

Pour faciliter l'application de l'approche objet, il est essentiel d'avoir des outils qui permettent d'exprimer les concepts objet de manière claire et concise. De plus, une démarche d'analyse et de conception objet rigoureuse est nécessaire dès le départ d'un projet.

Conclusion :

L'approche objet offre une solution puissante pour le développement d'applications complexes, mais elle nécessite une bonne compréhension des concepts fondamentaux ainsi qu'une approche méthodologique pour en tirer pleinement parti.

Les méthodes objet et la genèse d'UML

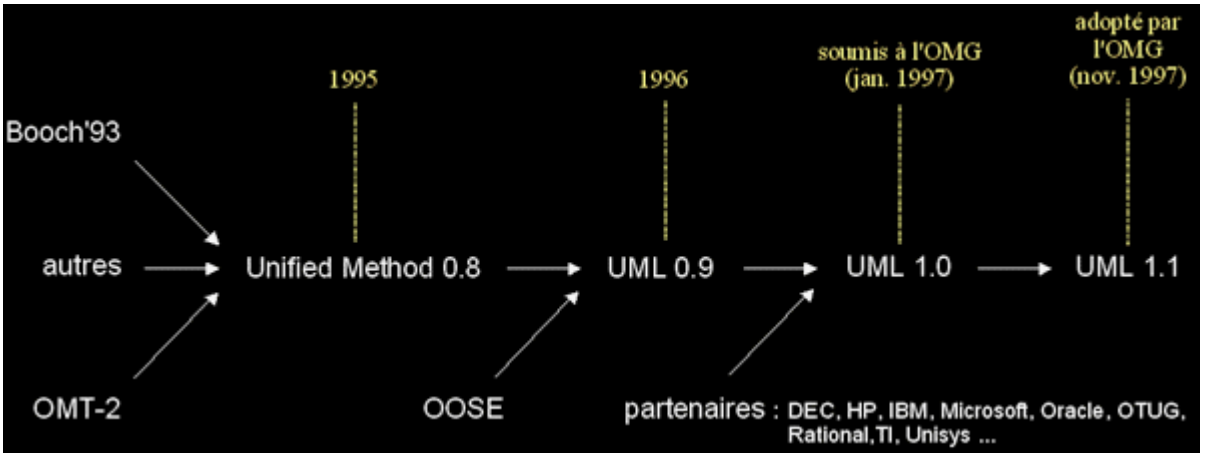
Comprendre l'évolution et l'utilité d'UML

L'évolution des méthodes d'analyse :

Les premières méthodes d'analyse, développées dans les années 70, se concentraient sur la découpe cartésienne des systèmes, c'est-à-dire une approche fonctionnelle et hiérarchique.

Dans les années 80, est apparue l'approche systémique avec des méthodes comme Merise, Axial, ou IE, qui combinaient la modélisation des données et des traitements pour une vue holistique des systèmes.

Puis, dans les années 90, avec l'avènement des systèmes complexes, est née la nécessité d'une méthode spécifiquement orientée objet pour structurer ces systèmes de manière équilibrée entre données et traitements.



La montée en puissance d'UML :

UML (Unified Modeling Language) est né de cette exigence. Il a fusionné et synthétisé les meilleures pratiques des méthodes objet dominantes pour devenir un standard incontournable dans le monde du développement logiciel.

Aujourd'hui, UML est reconnu comme un langage formel et normalisé qui offre une notation graphique précise et une structure bien définie. Il facilite la communication entre les membres d'une équipe de développement et encourage l'utilisation d'outils spécialisés.

Les points forts d'UML :

UML est un langage formel et normalisé, ce qui garantit sa précision et sa stabilité. Sa notation graphique facilite la communication en permettant de représenter visuellement des concepts abstraits complexes.

De plus, sa polyvalence et son caractère universel en font un langage largement utilisé dans l'industrie du développement logiciel.

Les points faibles d'UML :

Cependant, l'apprentissage et l'application d'UML peuvent être complexes et nécessitent une période d'adaptation. De plus, bien que UML offre un cadre d'analyse robuste, il ne couvre pas le processus de développement logiciel, ce qui peut poser des défis lors de son intégration dans un processus existant.

Conclusion :

UML représente une étape majeure dans l'évolution des méthodes d'analyse et de conception logicielle. Il offre un langage commun et formel pour exprimer des concepts objet de manière précise et universelle, ce qui en fait un outil indispensable dans le domaine du développement logiciel.

B. MODÉLISER AVEC UML

Qu'est-ce qu'un modèle ?

Un modèle, dans le contexte de l'informatique et de la modélisation des systèmes, est une représentation abstraite de la réalité. Il permet de simplifier et de comprendre des systèmes complexes en identifiant les caractéristiques essentielles et en les structurant de manière logique.

Caractéristiques fondamentales des modèles :

- **Abstraction** : Un modèle réduit la complexité du système étudié en mettant en évidence ses aspects les plus importants.
- **Simulation** : Un modèle permet de simuler le comportement du système étudié, offrant ainsi la possibilité d'explorer différentes configurations et scénarios.
- **Réduction** : En décomposant la réalité, un modèle fournit des éléments de travail exploitables par des méthodes mathématiques ou informatiques, facilitant ainsi leur manipulation et leur compréhension.

Comment modéliser avec UML ?

UML (Unified Modeling Language) est un langage de modélisation qui offre une notation graphique standardisée pour représenter des modèles. Bien qu'il ne définisse pas de processus spécifique pour élaborer des modèles, il est souvent utilisé dans le cadre de démarches itératives et incrémentales, pilotées par les besoins des utilisateurs et centrées sur l'architecture logicielle.

Une démarche itérative et incrémentale :

L'approche itérative et incrémentale consiste à diviser la modélisation et le développement d'un système en plusieurs étapes successives, en affinant

progressivement l'analyse et en développant le système par petites itérations.

Une démarche pilotée par les besoins des utilisateurs :

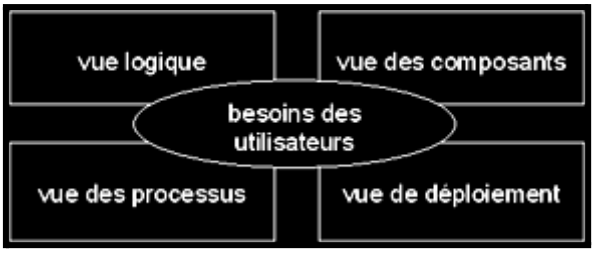
Les besoins des utilisateurs guident la définition et l'évolution des modèles. À chaque étape du processus, les besoins sont clarifiés, affinés et validés, assurant ainsi que le système répond aux attentes et aux exigences des utilisateurs.

Une démarche centrée sur l'architecture :

L'architecture logicielle, définissant les choix stratégiques et les structures fondamentales d'un système, joue un rôle crucial dans le succès d'un projet de développement. Une démarche centrée sur l'architecture veille à ce que les décisions architecturales soient cohérentes et adaptées aux besoins du système et de ses utilisateurs.

En intégrant ces principes dans la modélisation avec UML, les équipes de développement peuvent mieux maîtriser la complexité des systèmes informatiques et garantir la satisfaction des utilisateurs finaux.

L'approche centrée sur l'architecture met l'accent sur la définition de choix stratégiques qui déterminent les qualités fondamentales du logiciel, telles que son évolutivité, ses performances et sa fiabilité. Pour ce faire, différentes perspectives architecturales peuvent être explorées, comme le propose Philippe Kruchten avec son modèle "4+1".



Cette approche s'inscrit dans une vision holistique du développement logiciel, où chaque aspect de l'architecture est examiné en profondeur pour garantir la cohérence et la robustesse du système. Elle implique

également une collaboration étroite entre les différentes parties prenantes du projet, afin de s'assurer que les décisions architecturales correspondent aux besoins et aux contraintes de l'ensemble des intervenants.

Récapitulatif :

En combinant une démarche itérative et incrémentale, pilotée par les besoins des utilisateurs et centrée sur l'architecture, les équipes de développement sont en mesure de produire des modèles UML qui capturent de manière précise les spécifications du système. Ces modèles servent alors de guide tout au long du cycle de développement, permettant aux développeurs de concevoir, implémenter et tester le système de manière efficace et cohérente.

Conclusion :

La modélisation avec UML n'est pas simplement une question de dessin de diagrammes ; c'est une approche méthodologique qui nécessite une compréhension profonde des besoins du système et une réflexion stratégique sur son architecture. En suivant une démarche rigoureuse et en utilisant les concepts fondamentaux d'UML, les équipes de développement peuvent créer des solutions logicielles de haute qualité qui répondent aux attentes des utilisateurs et qui sont adaptées à leur environnement opérationnel.

Définir une architecture avec UML (détail de la "vue 4+1") :

Définir une architecture avec UML, à travers la vue "4+1", offre une perspective holistique de la conception logicielle. Cette approche se décline en cinq vues, chacune apportant un éclairage spécifique sur le système à concevoir :

1. **Vue Logique :** Cette vue, de haut niveau, se concentre sur l'abstraction et l'encapsulation des éléments principaux du système.

Elle identifie les composants du domaine métier, essentiels à la mission du système. Organisés en catégories selon des critères logiques, ces éléments permettent de structurer la conception et de clarifier les responsabilités au sein du système.

2. **Vue des Composants :** À un niveau plus bas, cette vue, également appelée "vue de réalisation", se penche sur l'allocation physique des éléments de modélisation dans des modules concrets, tels que des fichiers sources ou des exécutables. Elle illustre la manière dont les modules interagissent pour réaliser les fonctionnalités définies dans la vue logique, tout en prenant en compte les contraintes de développement et les dépendances entre les composants.
3. **Vue des Processus :** Dans les environnements multitâches, cette vue décompose le système en termes de processus ou de tâches. Elle met en lumière les interactions et la communication entre ces processus, ainsi que les mécanismes de synchronisation dans le cas d'activités parallèles. Cette vue est essentielle pour comprendre le fonctionnement dynamique du système dans un contexte d'exécution.
4. **Vue de Déploiement :** Dans les environnements distribués, cette vue décrit la répartition physique du logiciel sur les ressources matérielles disponibles. Elle identifie les équipements physiques et leurs performances, ainsi que l'implantation des modules sur le réseau. En spécifiant les exigences en termes de performances et de tolérance aux pannes, cette vue permet de garantir une mise en œuvre robuste et efficace du système.
5. **Vue des Besoins des Utilisateurs :** Enfin, cette vue, également connue sous le nom de "vue des cas d'utilisation", guide l'ensemble de la démarche architecturale. Elle met l'accent sur la satisfaction des besoins des utilisateurs en définissant les fonctionnalités et les scénarios d'utilisation du système. En reliant les autres vues de l'architecture aux besoins des utilisateurs, elle assure la cohérence et la pertinence de la conception architecturale.

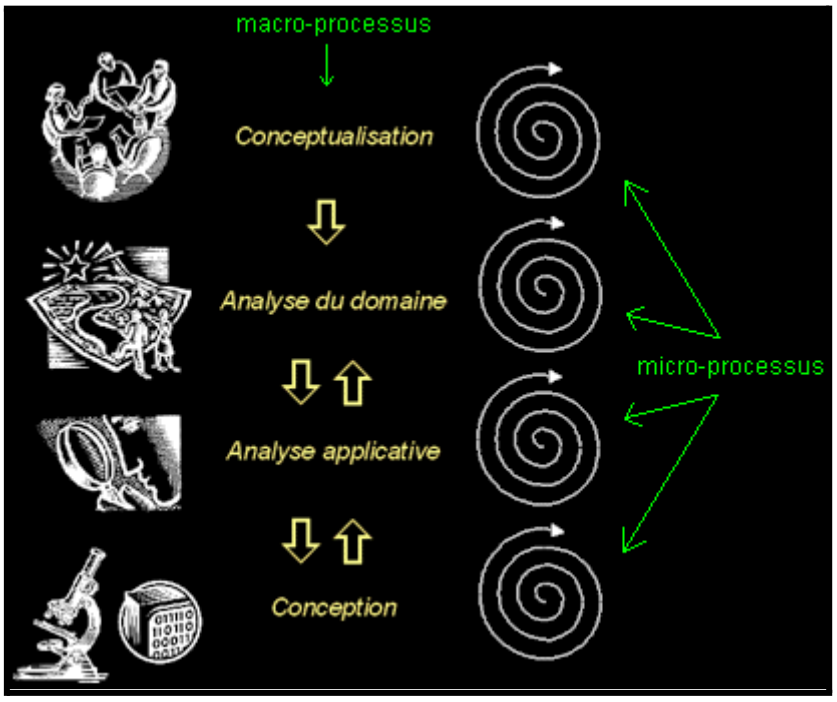
Récapitulatif de la démarche :

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

En pratique :

Bien qu'UML ne soit pas un processus en soi, il facilite une démarche d'analyse itérative et incrémentale, basée sur les niveaux d'abstraction. Cette approche permet de structurer les modèles et de construire les spécifications et conceptions en plusieurs étapes, en utilisant des catégories pour cibler les différents aspects du système.

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



Elaboration plutôt que transformation :

UML privilégie l'élaboration des modèles plutôt qu'une approche rigide qui sépare strictement l'analyse de la conception. Les modèles d'analyse et de

conception ne diffèrent que par leur niveau de détail, favorisant ainsi la traçabilité et les retours en arrière entre les différents niveaux d'abstraction. Cette approche simplifiée assure une cohérence et une continuité dans le processus de développement.

Détail des différents niveaux d'abstraction (phases du macro-processus) :

Conceptualisation :

Au début de l'analyse, nous nous concentrons sur le dossier d'expression des besoins du client, qui fournit une première vue des exigences de l'utilisateur. Cependant, notre objectif à ce stade n'est pas de tout capturer de manière exhaustive, mais plutôt de clarifier, filtrer et organiser les besoins principaux. Nous définissons ainsi le contour du système à modéliser, nous identifions les fonctionnalités clés pour une meilleure compréhension, et nous établissons une base pour la planification du projet.

Analyse du domaine :

Une fois les besoins principaux clarifiés, nous entrons dans la modélisation des éléments et mécanismes principaux du système. Cette phase s'appuie sur le modèle des besoins clients, souvent exprimé à travers des cas d'utilisation UML. Nous identifions les éléments du domaine, qui sont étroitement liés au métier de l'entreprise et sont essentiels à la mission du système. De plus, nous organisons ces éléments en catégories logiques pour faciliter la répartition des tâches dans les équipes de développement et encourager la réutilisation des composants.

Analyse applicative :

À ce niveau, nous nous concentrons sur les aspects informatiques du système sans plonger dans les détails d'implémentation. Nous définissons les interfaces des éléments de modélisation, en mettant l'accent sur

l'encapsulation pour assurer une meilleure modularité et réduire la complexité. De plus, nous déterminons les relations entre les différents éléments des modèles, en précisant leurs interactions pour une meilleure compréhension. Il est important de noter que les modèles à ce niveau peuvent être propres à une version spécifique du système, ce qui nous permet d'adapter notre approche en fonction des besoins du projet.

Conception :

Dans cette phase, nous allons plus loin dans la modélisation en détaillant tous les éléments nécessaires à l'implémentation du système. Nous optimisons les modèles produits, en tenant compte des contraintes et des exigences techniques, car ils sont destinés à être implémentés. C'est à ce stade que nous prenons des décisions concrètes concernant l'architecture logicielle, les algorithmes utilisés et d'autres aspects techniques de la conception du système.

Activités des micro-processus d'analyse (niveau d'abstraction constant) :

Chaque niveau d'abstraction est caractérisé par un ensemble d'activités spécifiques, guidées par un micro-processus qui facilite la construction des modèles.

- **Identification des classes d'objets :** Cette activité consiste à rechercher les classes candidates en utilisant des diagrammes d'objets pour esquisser les premières idées. Ensuite, nous filtrons les classes redondantes ou trop spécifiques pour ne conserver que celles qui représentent des concepts essentiels. Enfin, nous documentons les caractéristiques des classes retenues, telles que leur persistance ou le nombre maximal d'instances autorisées.
- **Identification des associations entre classes / interactions entre objets :** À ce stade, nous recherchons les connexions sémantiques entre les classes et définissons les relations d'utilisation entre elles.

Ces relations sont documentées avec des détails tels que leur nom, leurs cardinalités et les rôles des classes impliquées. De plus, nous spécifions la dynamique des relations entre les objets en décrivant leurs interactions et les activités associées.

- **Identification des attributs et des opérations des classes :** Nous recherchons les attributs dans les modèles dynamiques, en identifiant les données qui caractérisent les états des objets. Ensuite, nous filtrons les attributs complexes et nous ne représentons pas les valeurs internes propres aux mécanismes d'implémentation au niveau de spécification. Pour les opérations, nous les trouvons parmi les activités et les actions des objets, sans entrer dans les détails d'implémentation.
- **Optimisation des modèles :** À ce stade, nous choisissons nos critères d'optimisation, tels que la généricité, l'évolutivité, la précision, la lisibilité et la simplicité. Nous utilisons des techniques telles que la généralisation et la spécialisation pour organiser nos modèles de manière à les rendre plus efficaces et plus faciles à comprendre. De plus, nous documentons et détaillons nos modèles pour les rendre plus clairs et plus exploitables.
- **Validation des modèles :** Enfin, nous vérifions la cohérence, la complétude et l'homogénéité de nos modèles pour nous assurer qu'ils reflètent correctement les exigences du système. Nous confrontons également nos modèles à la critique en les soumettant à un comité de relecture ou à d'autres mécanismes de validation pour garantir leur qualité et leur pertinence.

Synthèse de la démarche :

Modéliser une application est un processus complexe et itératif qui nécessite une approche bien structurée et cohérente. Voici quelques points à retenir pour récapituler la démarche :

1. **Approche itérative et incrémentale :** Cette approche consiste à construire et à valider les modèles par étapes, en commençant par

des versions simples et en les affinant progressivement. Cela permet de mieux cerner et maîtriser la complexité du système à modéliser.

2. **Centrage sur l'architecture :** L'architecture du système est la clé de voûte qui conditionne la plupart des qualités de l'application. En se concentrant sur la définition et la validation de l'architecture à travers la vue "4+1" proposée par UML, on s'assure que le système répondra efficacement aux besoins et aux contraintes identifiés.
3. **Prise en compte des besoins des utilisateurs :** Les besoins des utilisateurs sont au cœur de la démarche de modélisation. En utilisant des cas d'utilisation pour guider la définition des modèles, on s'assure que le système développé sera en phase avec les attentes et les exigences des utilisateurs finaux.
4. **Activités des micro-processus d'analyse :** Chaque niveau d'abstraction est caractérisé par un ensemble d'activités spécifiques visant à construire et à valider les modèles. De l'identification des classes et des associations à l'optimisation et à la validation des modèles, ces activités permettent de garantir la qualité et la pertinence des modèles produits à chaque étape du processus.

En résumé, modéliser une application avec UML nécessite une approche itérative et structurée, centrée sur l'architecture et guidée par les besoins des utilisateurs. En suivant cette démarche, les développeurs peuvent construire des modèles de qualité qui serviront de base solide pour le développement et la mise en œuvre du système informatique.

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).

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.

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

C. Modéliser les vues statiques d'un système

Les vues statiques d'UML

LES CAS D'UTILISATION (USE CASE)

La conceptualisation :

- Le but de la conceptualisation est de comprendre et structurer les besoins du client.
- Il ne faut pas chercher l'exhaustivité, mais clarifier, filtrer et organiser les besoins !
- Une fois identifiés et structurés, ces besoins :
    - définissent le contour du système à modéliser (ils précisent le but à atteindre),
    - permettent d'identifier les fonctionnalités principales (critiques) du système.

- Le modèle conceptuel doit permettre une meilleure compréhension du système.
- Le modèle conceptuel doit servir d'interface entre tous les acteurs du projet.
- Les besoins des clients sont des éléments de traçabilité dans un processus intégrant UML.
- Le modèle conceptuel joue un rôle central, il est capital de bien le définir !

Cas d'utilisation (use cases)

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

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.

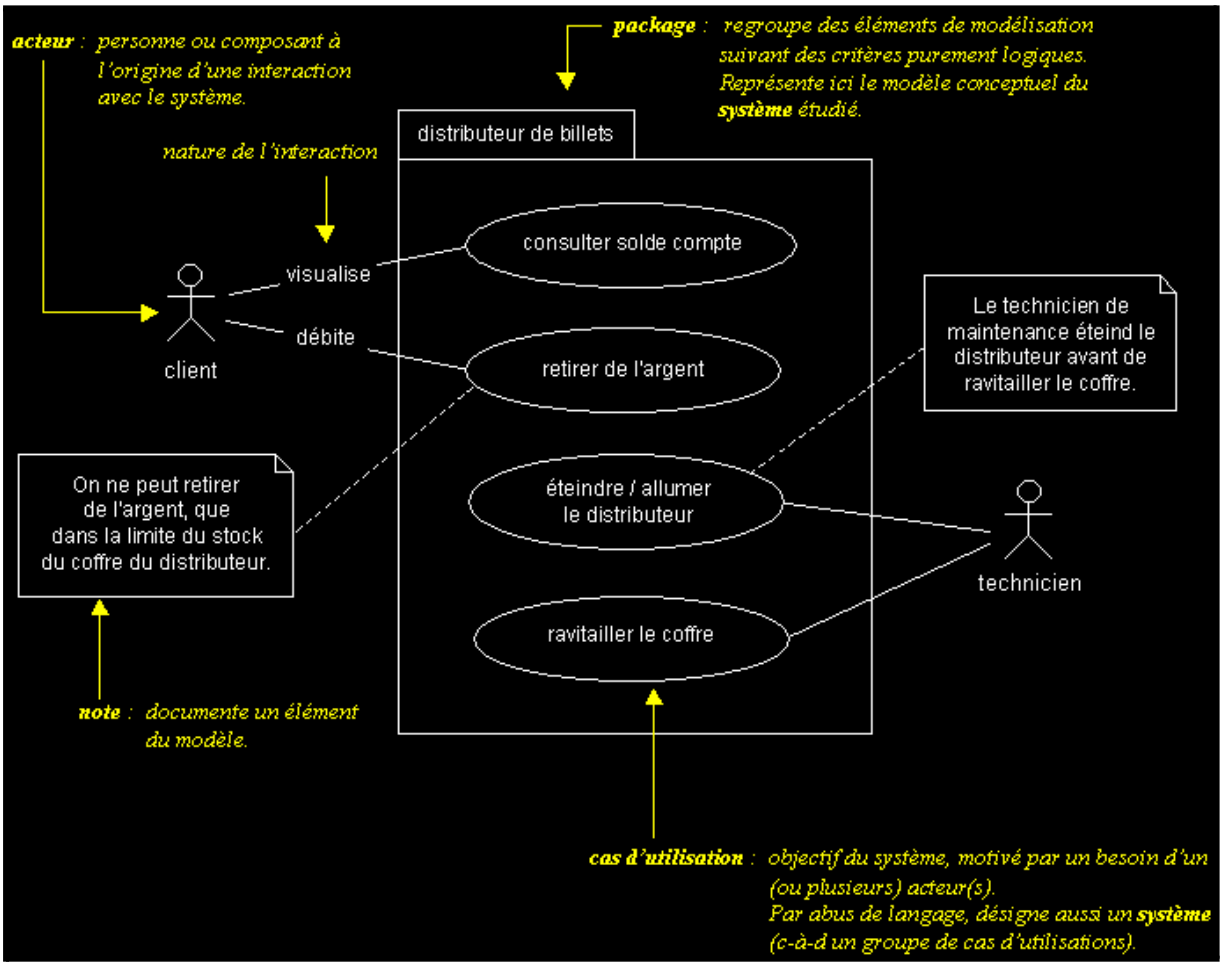
Eléments de base des cas d'utilisation

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

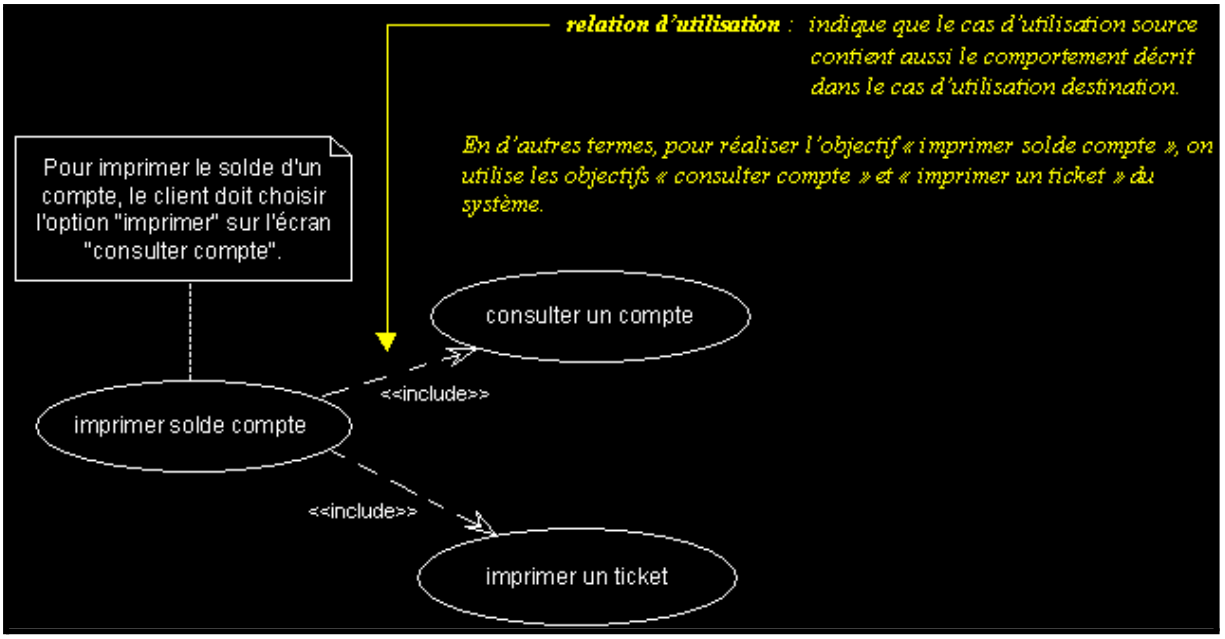
- **Use case** : ensemble d'actions réalisées par le système, en réponse à une action d'un acteur.
    - Les uses cases peuvent être structurées.
    - Les uses cases peuvent être organisées en paquetages (packages).
    - L'ensemble des use cases décrit les objectifs (le but) du système.

Exemples

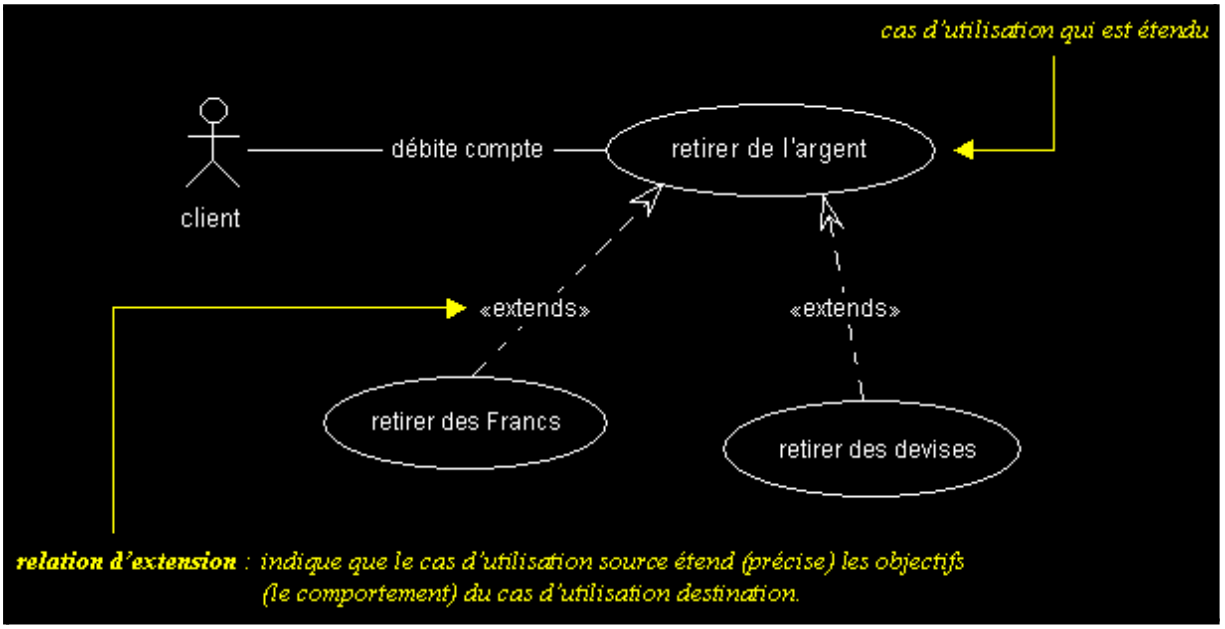
Cas d'utilisation standard :



Relation d'utilisation :



Relation d'extension :

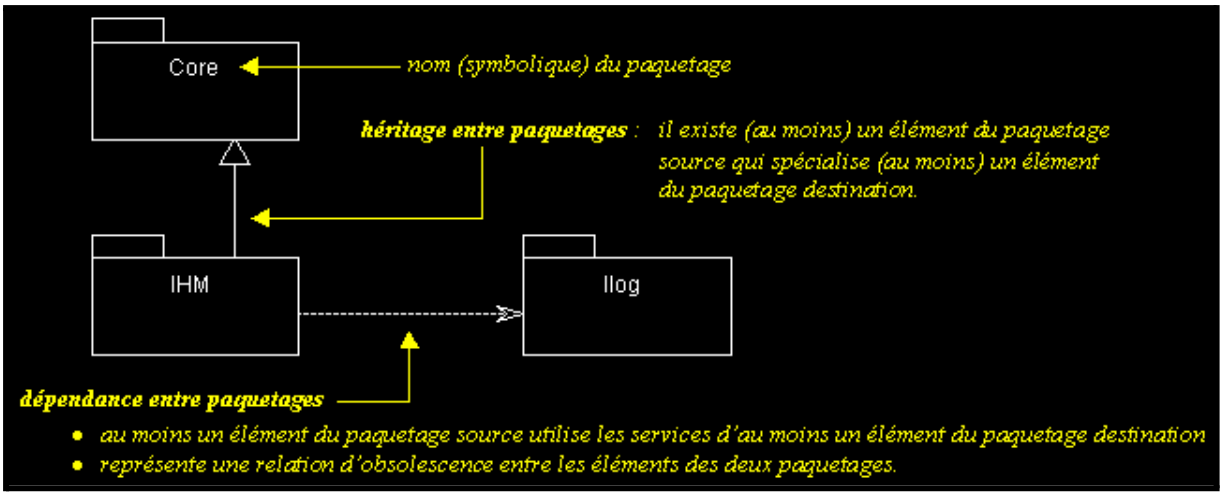


LES PAQUETAGES

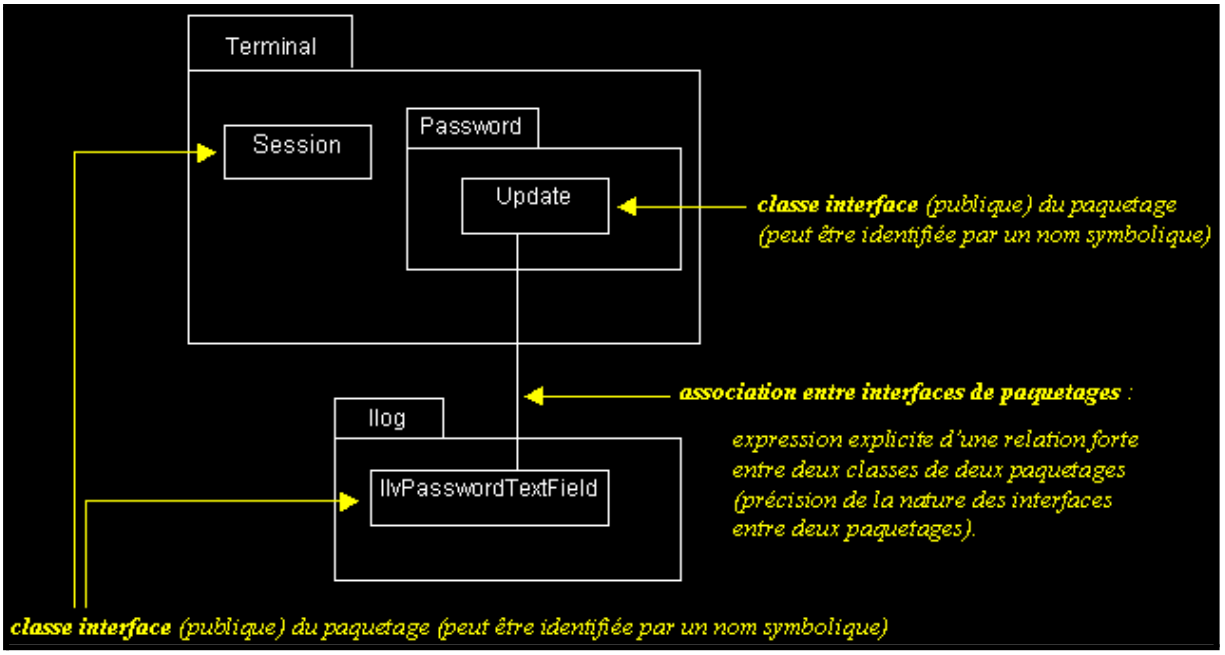
Paquetages (packages)

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

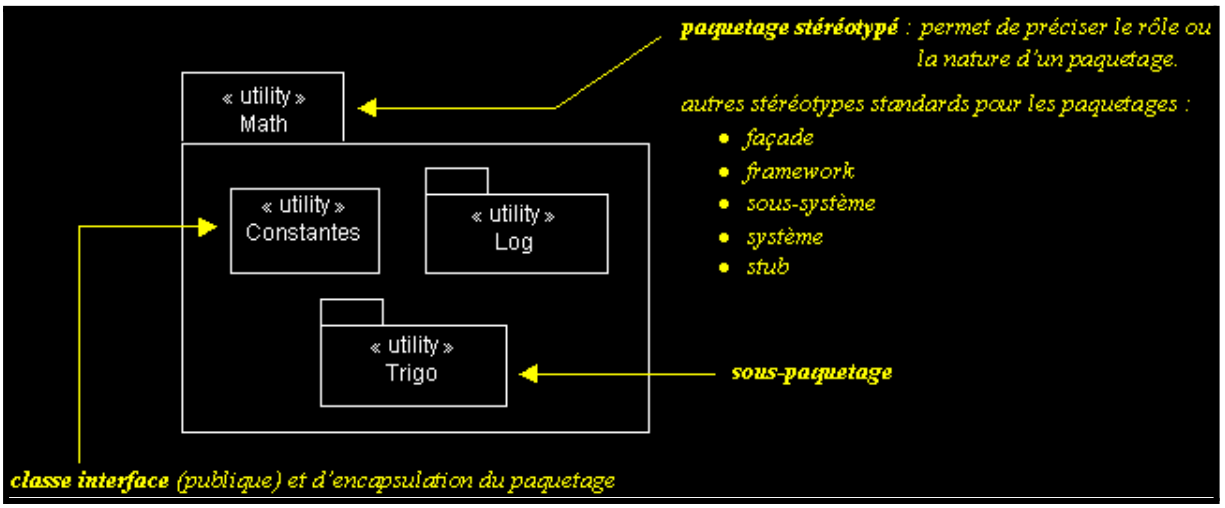
Paquetages : relations entre paquetages

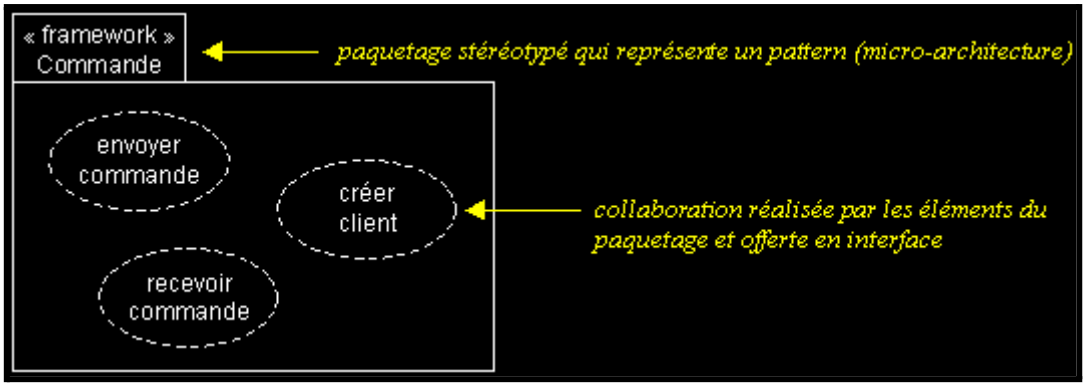


Paquetages : interfaces



Paquetages : stéréotypes



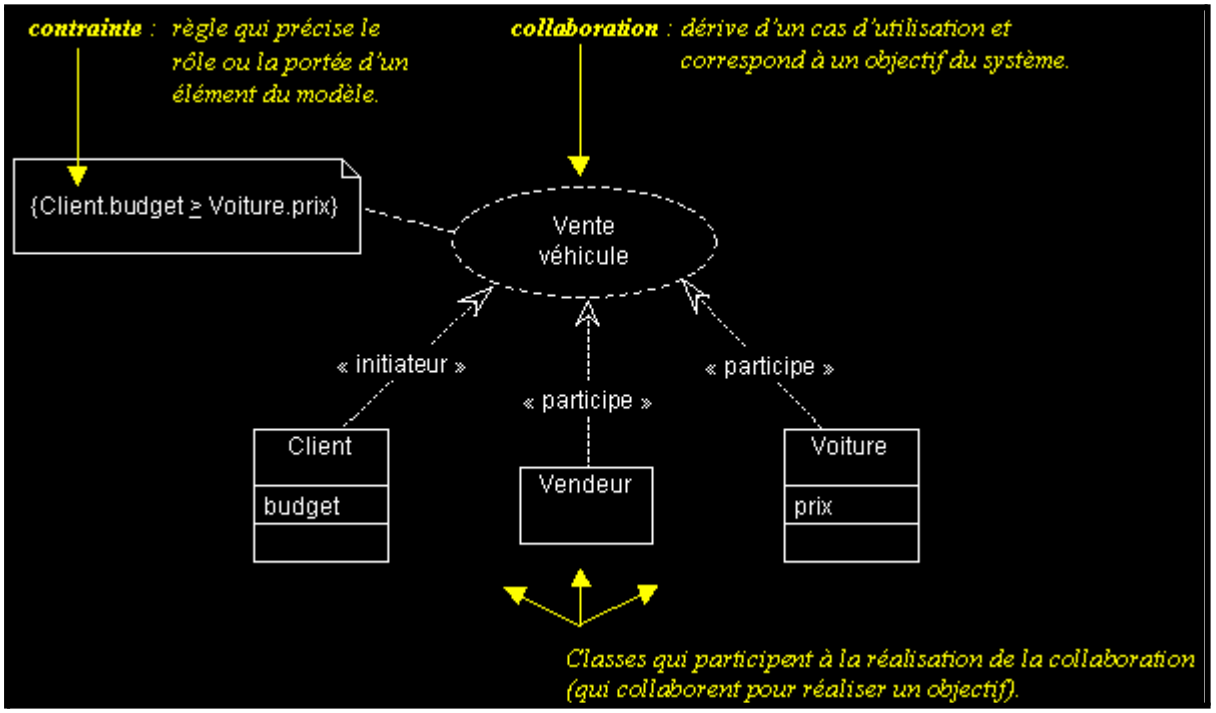

« framework »
Commande
paquetage stéréotypé qui représente un pattern (micro-architecture)
envoyer commande
créer client
recevoir commande
collaboration réalisée par les éléments du paquetage et offerte en interface

LA COLLABORATION

Symbole de modélisation "collaboration"

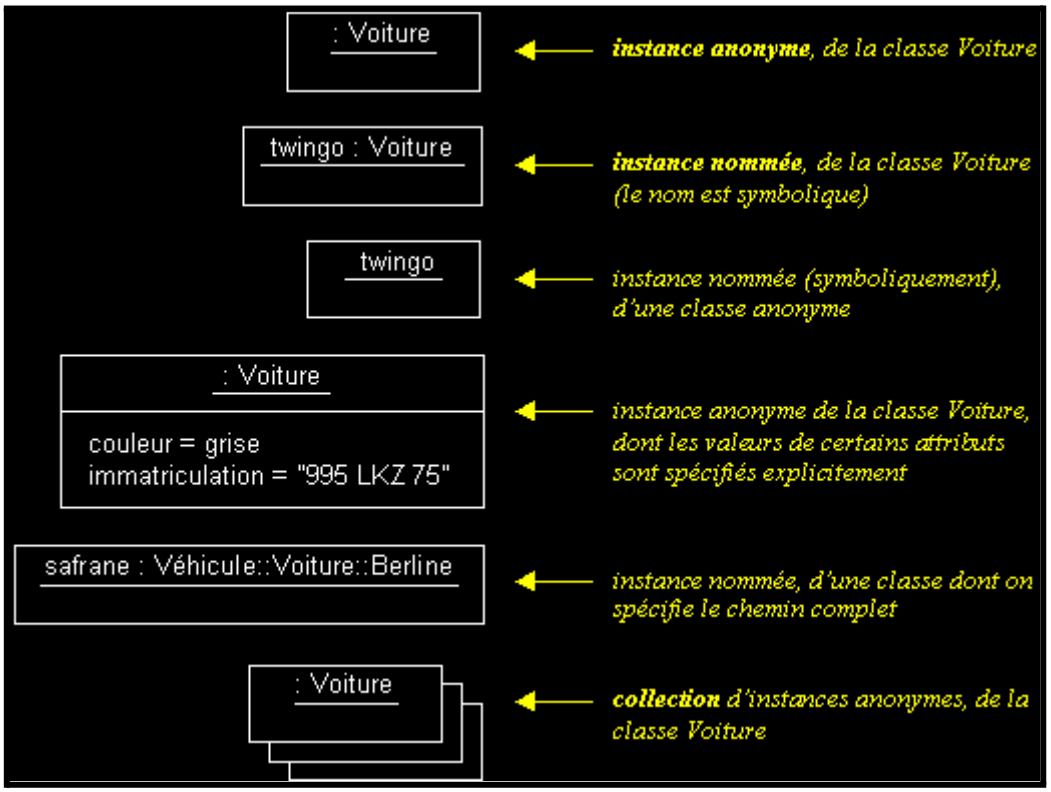
- Les collaborations sont des interactions entre objets, dont le but est de réaliser un objectif du système (c'est-à-dire aussi de répondre à un besoin d'un utilisateur).
- L'élément de modélisation UML "collaboration", représente les classes qui participent à la réalisation d'un cas d'utilisation.

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

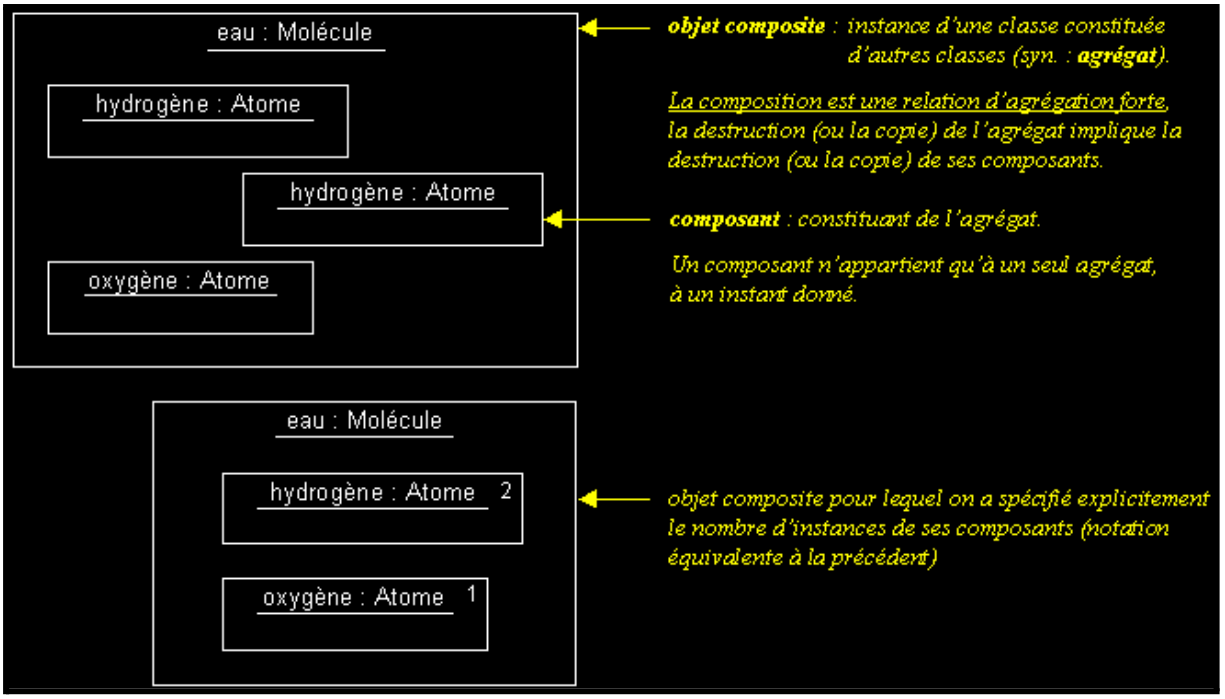


INSTANCES ET DIAGRAMME D'OBJETS

Exemples d'instances



Objets composites



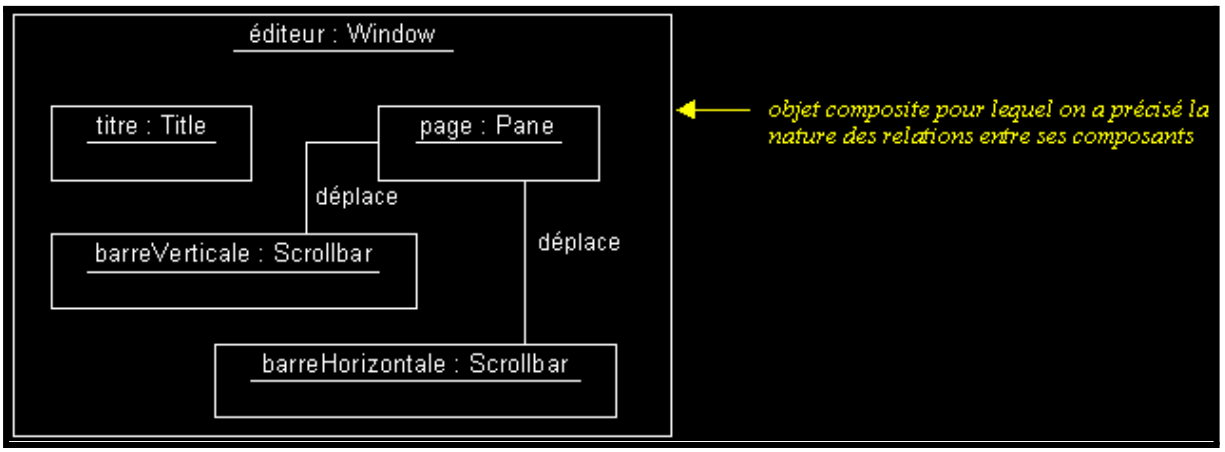
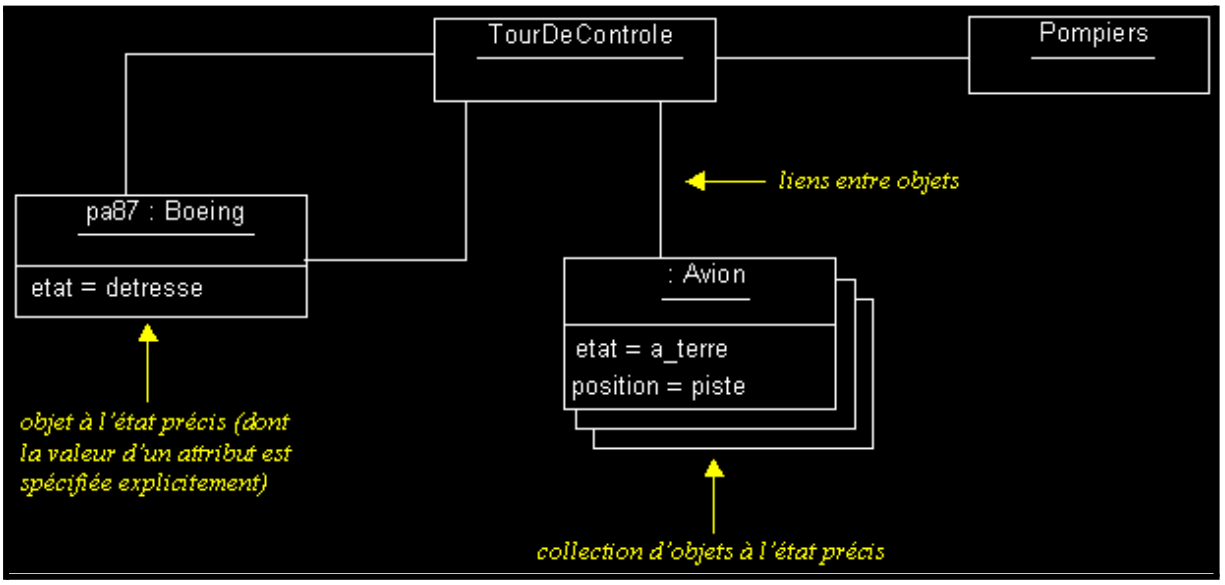


Diagramme d'objets

- Ce type de diagramme UML montre des objets (instances de classes dans un état particulier) et des liens (relations sémantiques) entre ces objets.
- Les diagrammes d'objets s'utilisent pour montrer un contexte (avant ou après une interaction entre objets par exemple).

- Ce type de diagramme sert essentiellement en phase exploratoire, car il possède un très haut niveau d'abstraction.

Exemple :



LES CLASSES

Classe : sémantique et notation

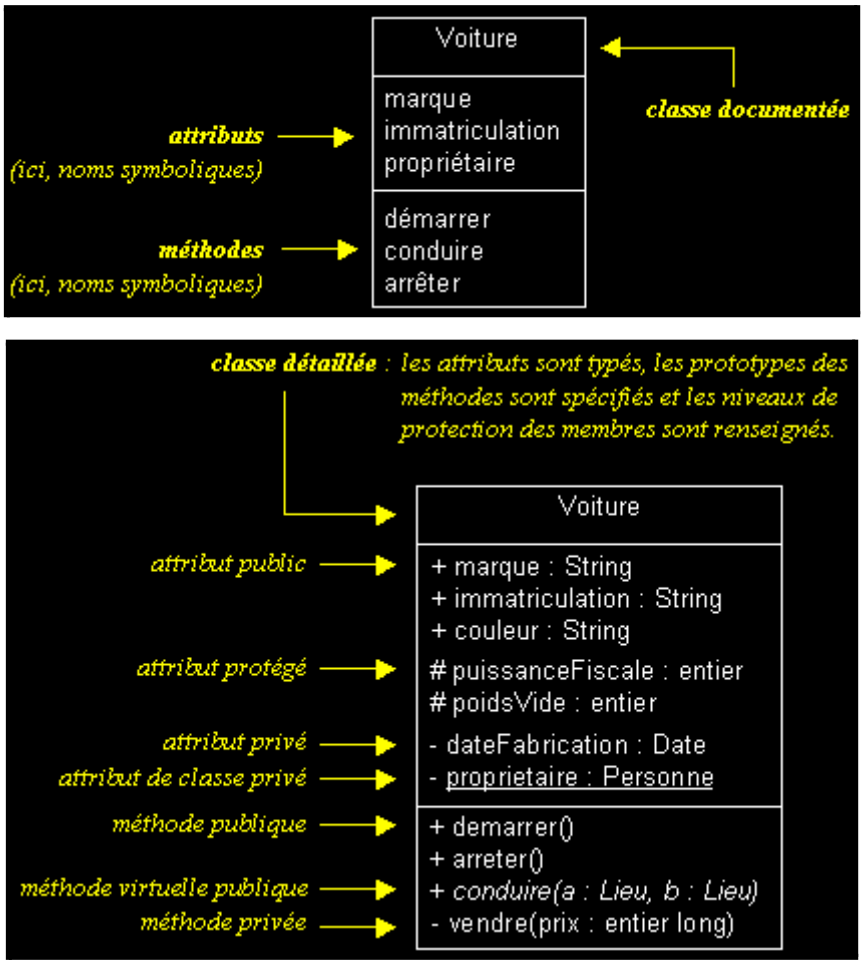
- Une classe est un type abstrait caractérisé par des propriétés (attributs et méthodes) communes à un ensemble d'objets et permettant de créer des objets ayant ces propriétés.

  Classe = attributs + méthodes + instanciation

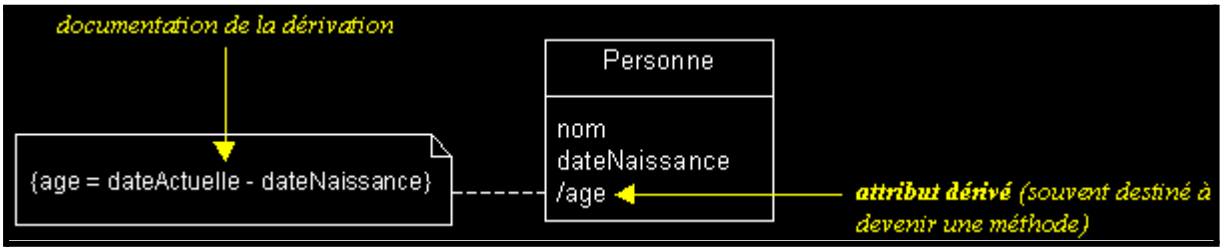
- Ne pas représenter les attributs ou les méthodes d'une classe sur un diagramme, n'indique pas que cette classe n'en contient pas.
  Il s'agit juste d'un filtre visuel, destiné à donner un certain niveau d'abstraction à son modèle.

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

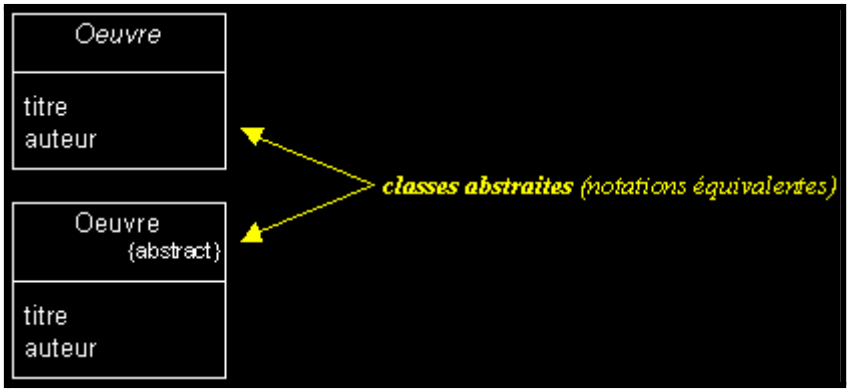
Documentation d'une classe (niveaux d'abstraction), exemples :



Attributs multivalués et dérivés, exemples :



Classe abstraite, exemple :



Template (classe paramétrable), exemple :

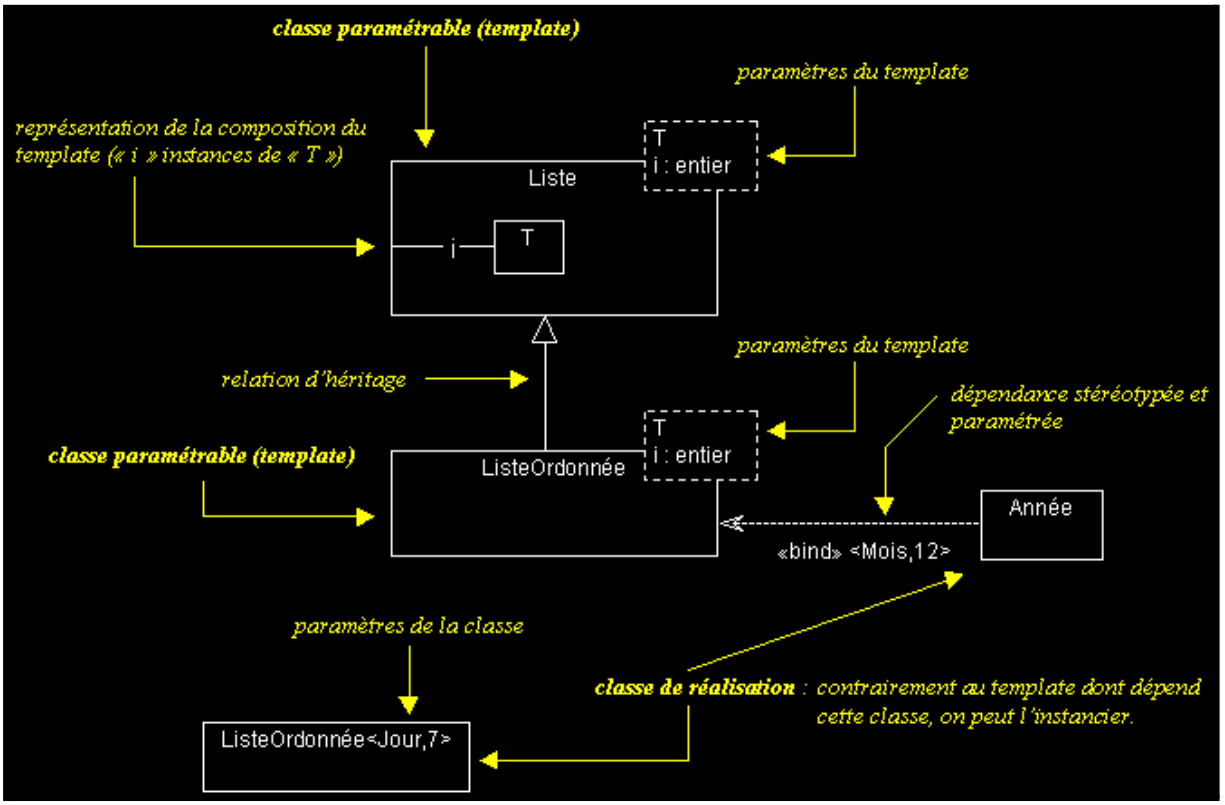


DIAGRAMME DE CLASSES

Diagramme de classes : sémantique

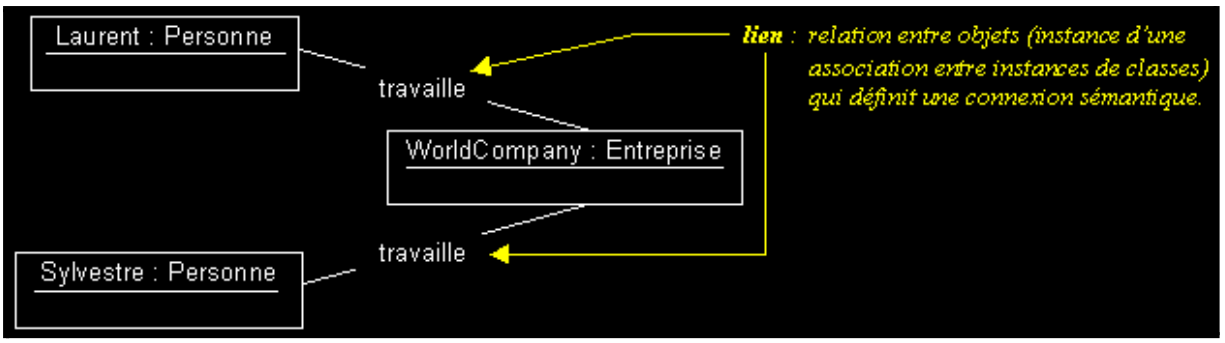
- Un diagramme de classes est une collection d'éléments de modélisation statiques (classes, paquetages...), qui montre la structure d'un modèle.
- Un diagramme de classes fait abstraction des aspects dynamiques et temporels.
- Pour un modèle complexe, plusieurs diagrammes de classes complémentaires doivent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

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

- Pour représenter un contexte précis, un diagramme de classes peut être instancié en diagrammes d'objets.

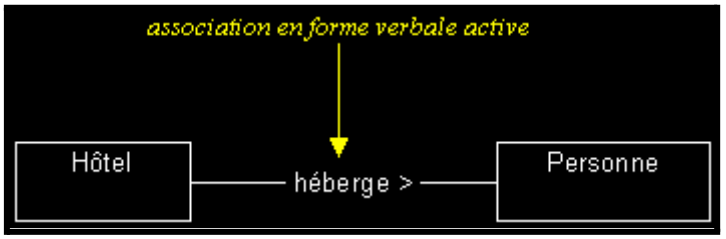
Associations entre classes

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

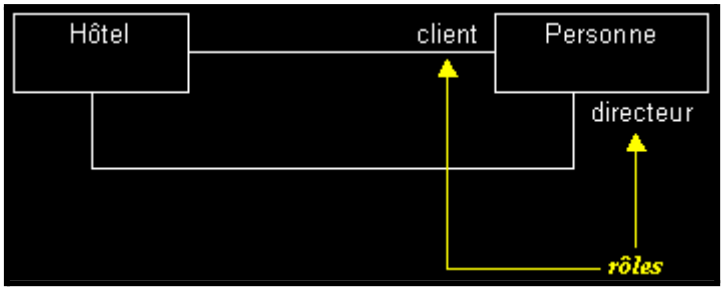


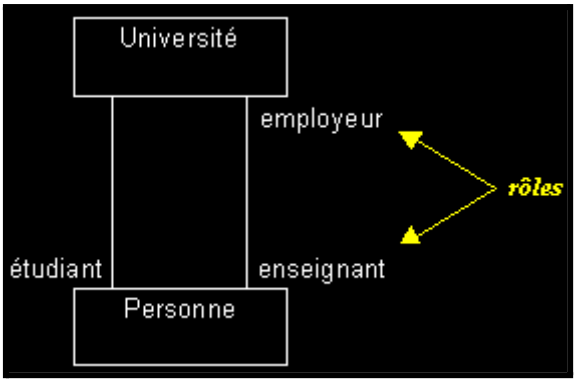
Documentation d'une association et types d'associations

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

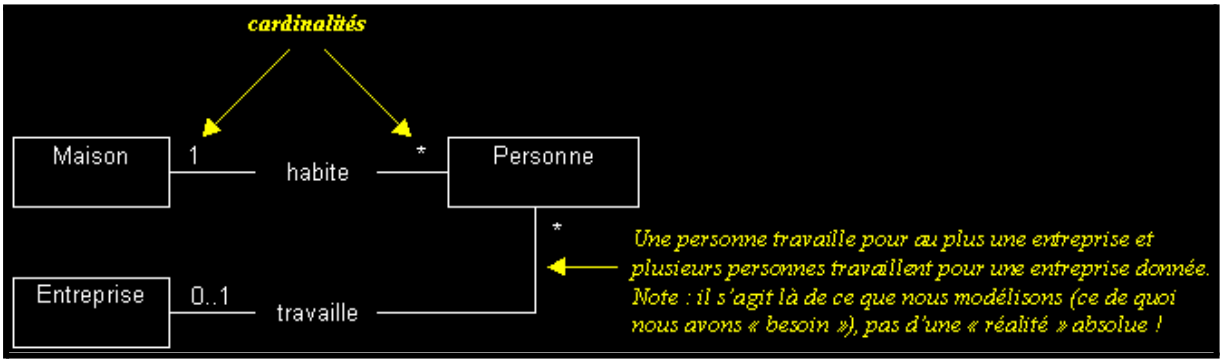


- **Rôles :** spécifie la fonction d'une classe pour une association donnée (indispensable pour les associations réflexives).





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



Expression des cardinalités d'une relation en UML :

1. **n :** Exprime exactement "n" instances (où "n" est un entier naturel supérieur à 0).
    - **Exemples :** "1" (exactement une instance), "7" (exactement sept instances).
2. **n..m :** Indique une plage allant de "n" à "m" instances (où "n" et "m" sont des entiers naturels ou des variables, et "m" est supérieur à "n").
    - **Exemples :** "0..1" (de zéro à une instance), "3..n" (de trois à un nombre indéterminé d'instances), "1..31" (d'une à trente et une instances).
3. ***** : Indique plusieurs instances, équivalent à "0..n" ou "0..*".
    - **Exemple :** "0..*" (zéro ou plus d'instances).
4. **n..* :** Indique "n" ou plus d'instances (où "n" est un entier naturel ou une variable).

- **Exemples :** "0.." *(zéro ou plus d'instances), "5..*" (cinq ou plus d'instances).

Cette notation aide à définir le nombre minimum et maximum d'instances qu'une entité peut avoir dans une relation en UML.

Comprendre les types de relations et les cardinalités en UML

Les termes "**one to many**", "**many to many**", "**many to one**" et "**one to one**" sont des types de relations couramment utilisés pour décrire les associations entre entités dans un modèle de données, comme en UML (Unified Modeling Language).
Voici comment ils se différencient et comment ils se rapportent aux cardinalités :

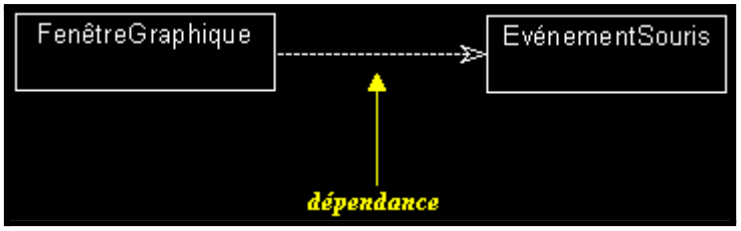
1. **One to One (1:1)** :
   - **Description :** Une instance d'une entité est associée à une seule instance d'une autre entité.
   - **Cardinalité UML :** La cardinalité serait "1..1" pour les deux entités. Par exemple, un passeport est lié à une seule personne, et cette personne a un seul passeport.
2. **One to Many (1:n)** :
   - **Description :** Une instance d'une entité est associée à plusieurs instances d'une autre entité.
   - **Cardinalité UML :** La cardinalité de l'entité du côté "one" est "1", tandis que la cardinalité de l'entité du côté "many" est "0..n" ou "1..n". Par exemple, un auteur peut écrire plusieurs livres, mais chaque livre est écrit par un seul auteur.
3. **Many to One (n:1)** :
   - **Description :** Plusieurs instances d'une entité sont associées à une seule instance d'une autre entité.
   - **Cardinalité UML :** C'est l'inverse de "One to Many". Par exemple, plusieurs étudiants (many) sont inscrits dans un seul cours (one). La cardinalité du côté des étudiants serait "0..n" ou "1..n", et celle du cours serait "1".

4. **Many to Many (n:n)** :
    - **Description :** Plusieurs instances d'une entité sont associées à plusieurs instances d'une autre entité.
    - **Cardinalité UML :** Les deux entités auront des cardinalités de "0..n" ou "1..n". Par exemple, les étudiants peuvent s'inscrire à plusieurs cours, et chaque cours peut avoir plusieurs étudiants.

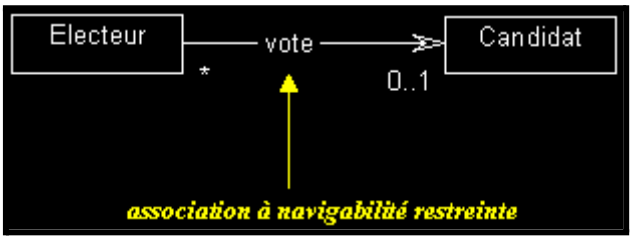
- **Exemples de Cardinalités en UML pour chaque relation :**
    - **One to One :**
        - Entité A : 1..1
        - Entité B : 1..1
    - **One to Many :**
        - Entité A (One) : 1..1
        - Entité B (Many) : 0..n ou 1..n
    - **Many to One :**
        - Entité A (Many) : 0..n ou 1..n
        - Entité B (One) : 1..1
    - **Many to Many :**
        - Entité A : 0..n ou 1..n
        - Entité B : 0..n ou 1..n

Les termes "one to many", "many to many", "many to one" et "one to one" décrivent la nature de la relation entre deux entités, tandis que les cardinalités en UML précisent le nombre exact ou la plage de nombres des instances associées.

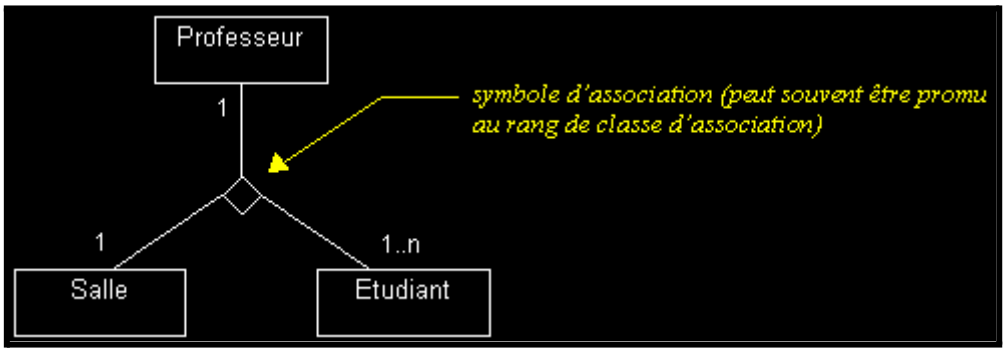
- **Relation de dépendance :** relation d'utilisation unidirectionnelle et d'obsolescence (une modification de l'élément dont on dépend, peut nécessiter une mise à jour de l'élément dépendant).



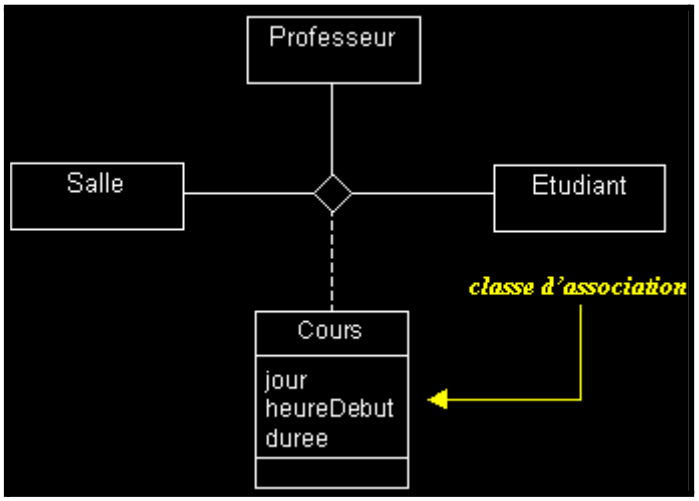
- **Association à navigabilité restreinte**
  Par défaut, une association est navigable dans les deux sens. La réduction de la portée de l'association est souvent réalisée en phase d'implémentation, mais peut aussi être exprimée dans un modèle pour indiquer que les instances d'une classe ne "connaissent"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

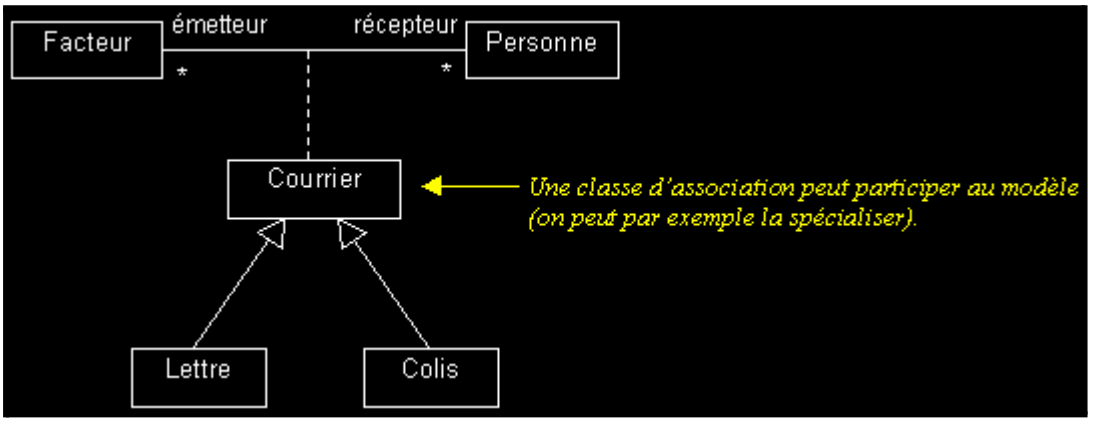


- **Association n-aire :** il s'agit d'une association qui relie plus de deux classes...
  Note : de telles associations sont difficiles à déchiffrer et peuvent induire en erreur. Il vaut mieux limiter leur utilisation, en définissant de nouvelles catégories d'associations.

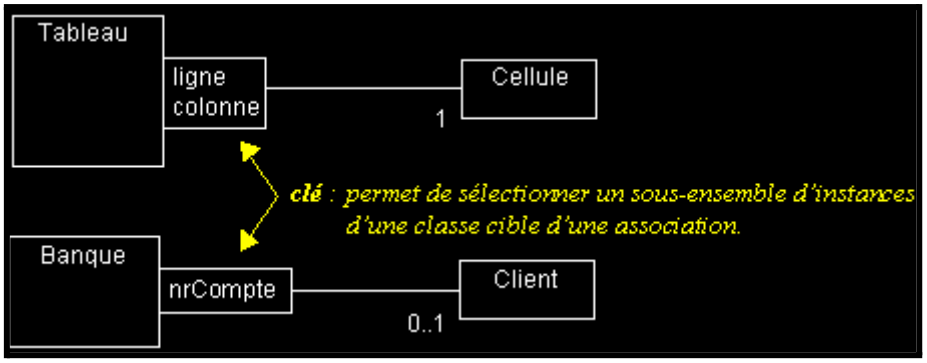


- **Classe d'association :** il s'agit d'une classe qui réalise la navigation entre les instances d'autres classes.





- **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.



Héritage

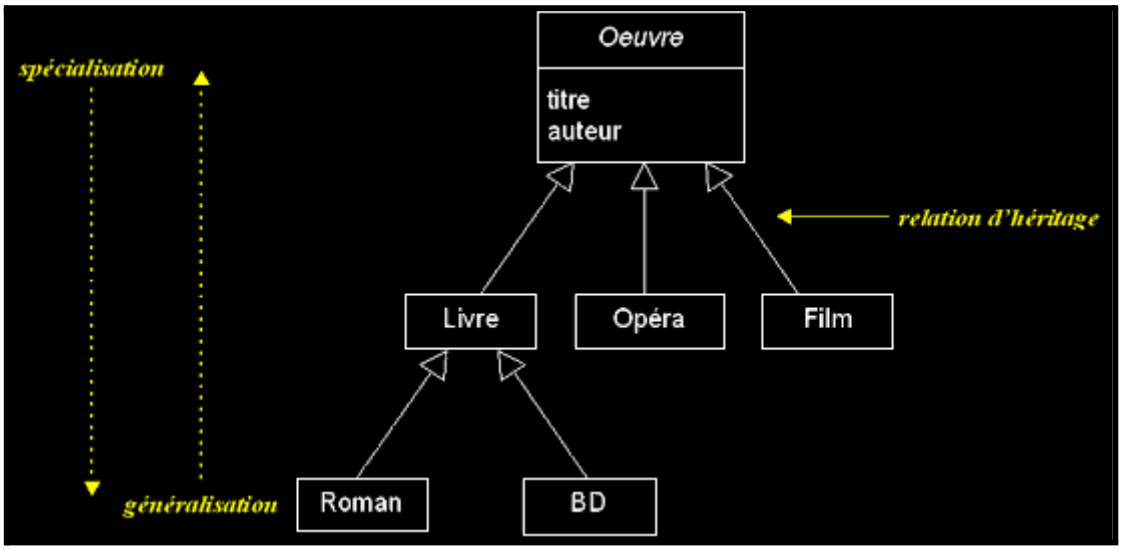
Les hiérarchies de classes permettent de gérer la complexité, en ordonnant les objets au sein d'arborescences de classes, d'abstraction croissante.

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

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

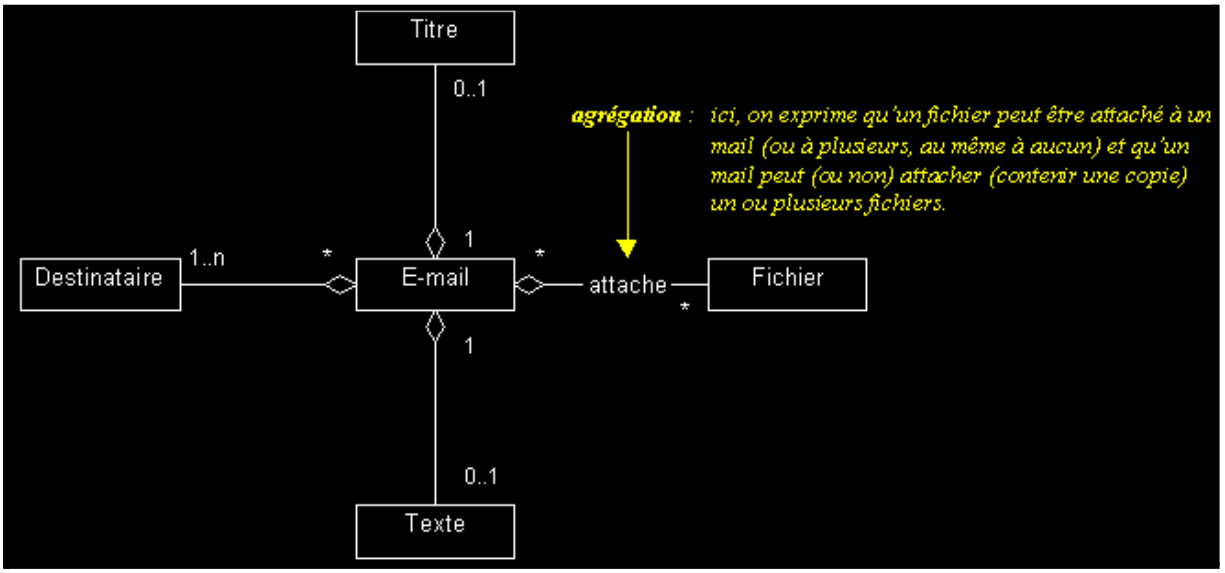
- **Classification**
  - L'héritage (spécialisation et généralisation) permet la classification des objets.
  - Une bonne classification est stable et extensible : ne classifiez pas les objets selon des critères instables (selon ce qui caractérise leur état) ou trop vagues (car cela génère trop de sous-classes).
  - Les critères de classification sont subjectifs.

- Le principe de substitution (Liksow, 1987) permet de déterminer si une relation d'héritage est bien employée pour la classification :
  "Il doit être possible de substituer n'importe quel instance d'une super-classe, par n'importe quel instance d'une de ses sous-classes, sans que la sémantique d'un programme écrit dans les termes de la super-classe n'en soit affectée."
- Si Y hérite de X, cela signifie que "Y est une sorte de X" (analogies entre classification et théorie des ensembles).



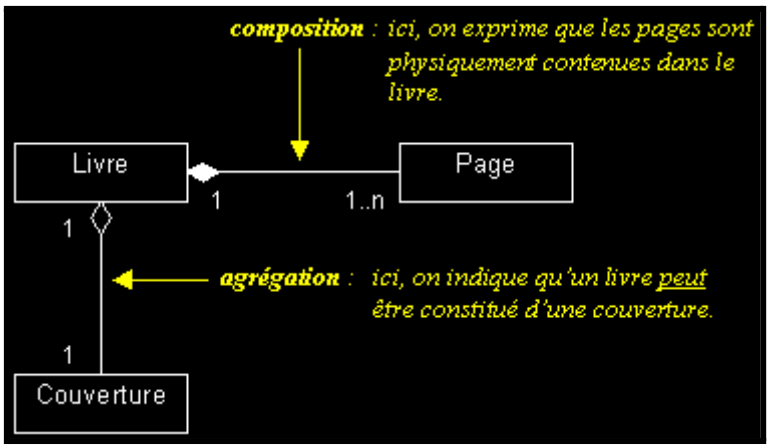
Agrégation

- L'agrégation est une association non symétrique, qui exprime un couplage fort et une relation de subordination.
  Elle représente une relation de type "ensemble / élément".
- UML ne définit pas ce qu'est une relation de type "ensemble / élément", mais il permet cependant d'exprimer cette vue subjective de manière explicite.
- Une agrégation peut notamment (mais pas nécessairement) exprimer :
  - qu'une classe (un "élément") fait partie d'une autre ("l'agrégat"),
  - qu'un changement d'état d'une classe, entraîne un changement d'état d'une autre,
  - qu'une action sur une classe, entraîne une action sur une autre.
- A un même moment, une instance d'élément agrégé peut être liée à plusieurs instances d'autres classes (l'élément agrégé peut être partagé).
- Une instance d'élément agrégé peut exister sans agrégat (et inversement) : les cycles de vies de l'agrégat et de ses éléments agrégés peuvent être indépendants.



Composition

- La composition est une agrégation forte (agrégation par valeur).
- Les cycles de vies des éléments (les "composants") et de l'agrégat sont liés : si l'agrégat est détruit (ou copié), ses composants le sont aussi.
- A un même moment, une instance de composant ne peut être liée qu'à un seul agrégat.
- Les "objets composites" sont des instances de classes composées.

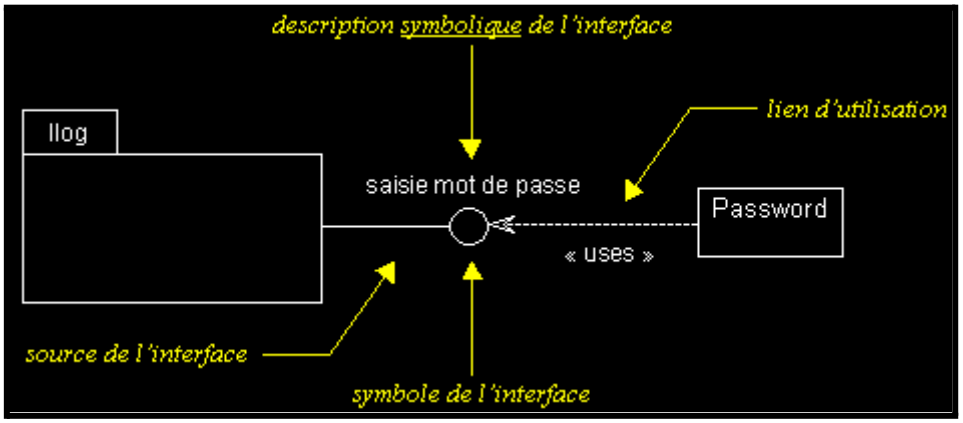


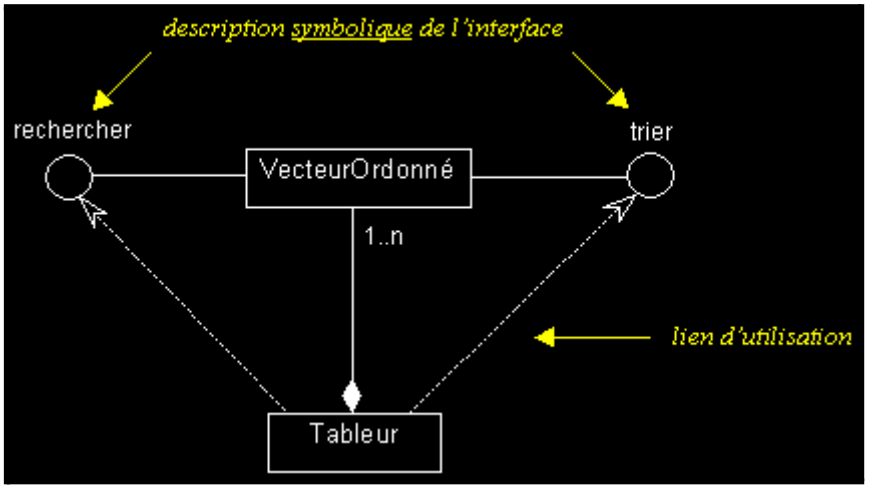
Agrégation et composition : rappel

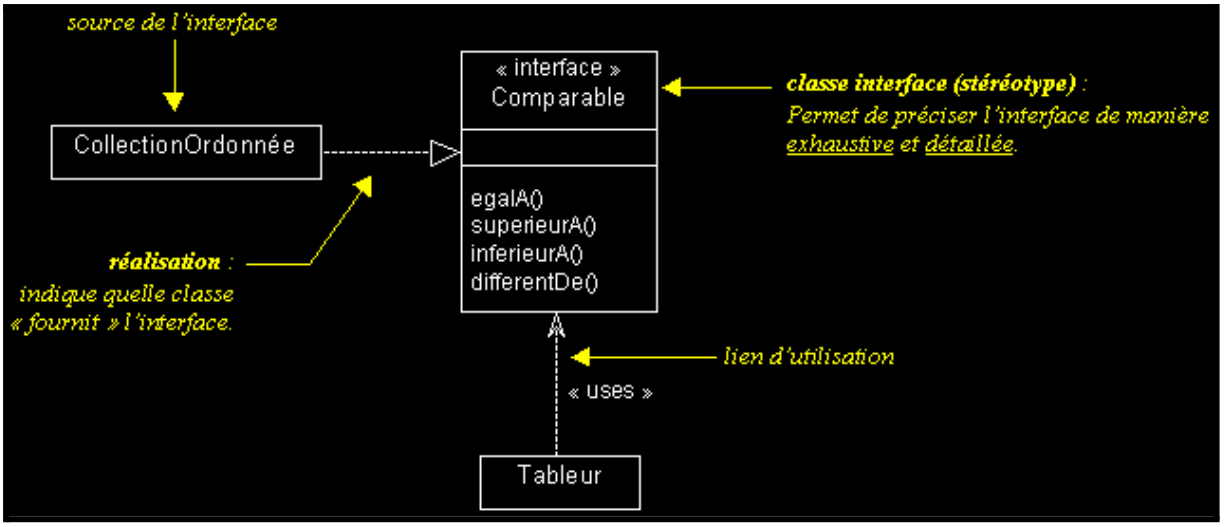
- L'agrégation et la composition sont des vues subjectives.
- Lorsqu'on représente (avec UML) qu'une molécule est "composée" d'atomes, on sous-entend que la destruction d'une instance de la classe "Molécule", implique la destruction de ses composants, instances de la classe "Atome" (cf. propriétés de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

Interfaces

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

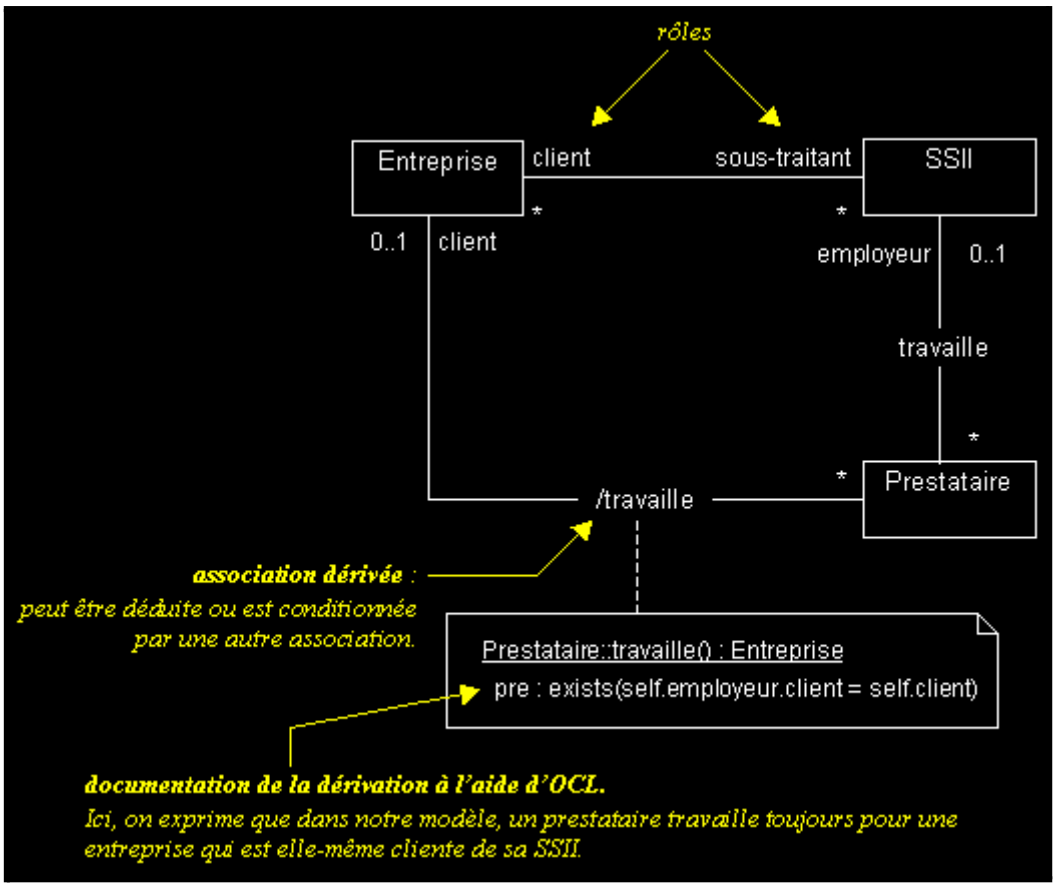






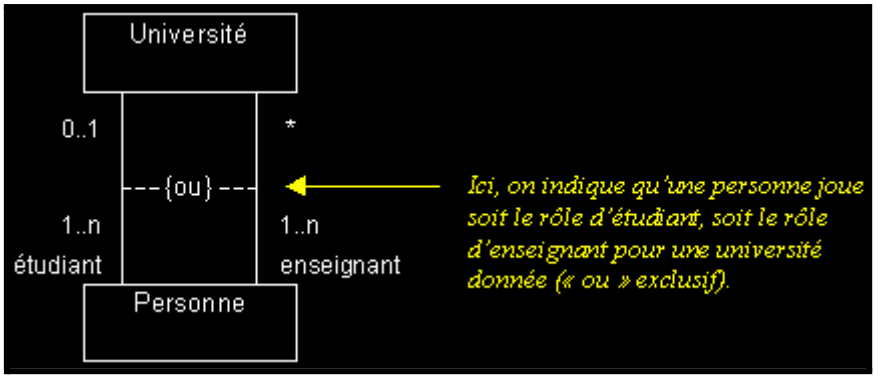
Association dérivée

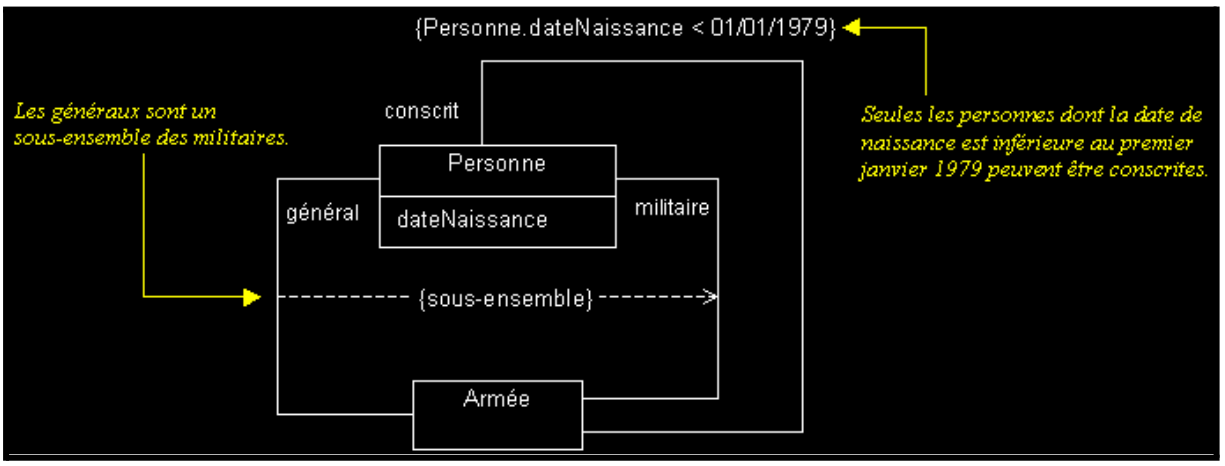
- Les associations dérivées sont des associations redondantes, qu'on peut déduire d'une autre association ou d'un ensemble d'autres associations.
- Elles permettent d'indiquer des chemins de navigation "calculés", sur un diagramme de classes.
- Elles servent beaucoup à la compréhension de la navigation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

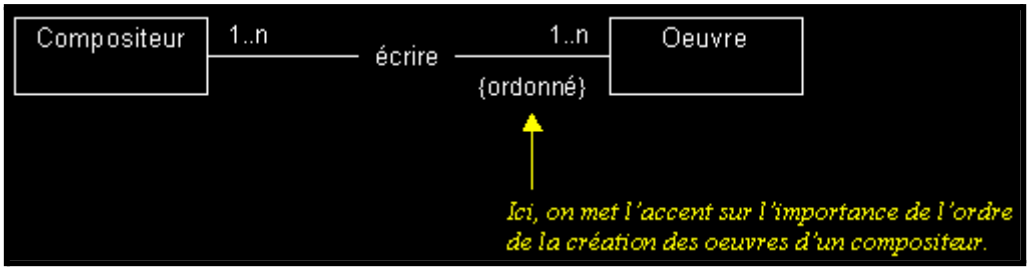


Contrainte sur une association

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







OCL

- UML formalise l'expression des contraintes avec OCL (Object Constraint Language).
- OCL est une contribution d'IBM à UML 1.1. (actuellement en version 2.4)
- Ce langage formel est volontairement simple d'accès et possède une grammaire élémentaire (OCL peut être interprété par des outils).
- Il représente un juste milieu, entre langage naturel et langage mathématique. OCL permet ainsi de limiter les ambiguïtés, tout en restant accessible.

- OCL permet de décrire des invariants dans un modèle, sous forme de pseudo-code :
    - pré et post-conditions pour une opération,
    - expressions de navigation,
    - expressions booléennes, etc...

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

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

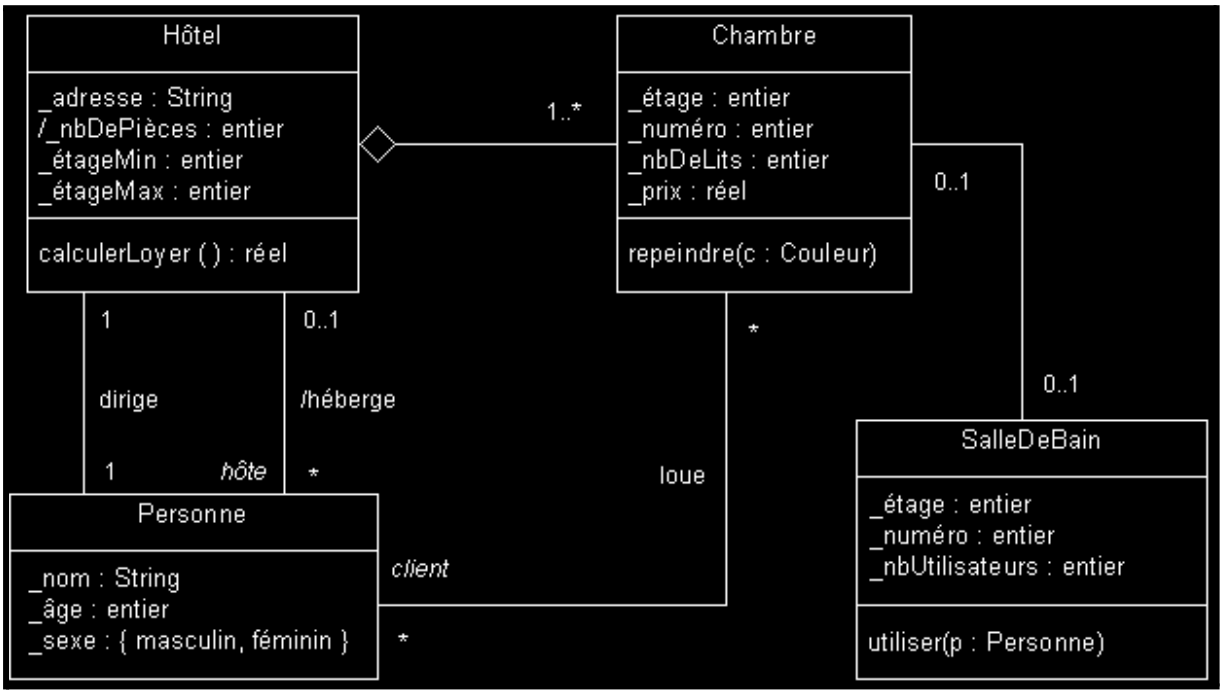
Monsieur Formulain, directeur d'une chaîne d'hôtels, vous demande de concevoir une application de gestion pour ses hôtels.
Voici ce que vous devez modéliser :

Un hôtel Formulain est constitué d'un certain nombre de chambres. Un responsable de l'hôtel gère la location des chambres. Chaque chambre se loue à un prix donné (suivant ses prestations).
L'accès aux salles de bain est compris dans le prix de la location d'une chambre. Certaines chambres comportent une salle de bain, mais pas toutes. Les hôtes de chambres sans salle de bain peuvent utiliser une salle de bain sur le palier. Ces dernières peuvent être utilisées par plusieurs hôtes.
Les pièces de l'hôtel qui ne sont ni des chambres, ni des salles de bain (hall d'accueil, cuisine...) ne font pas partie de l'étude (hors sujet).
Des personnes peuvent louer une ou plusieurs chambres de l'hôtel, afin d'y résider. En d'autre termes : l'hôtel héberge un certain nombre de personnes, ses hôtes (il s'agit des personnes qui louent au moins une chambre de l'hôtel...).

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

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

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



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

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

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

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

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

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

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

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

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

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

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

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

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

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

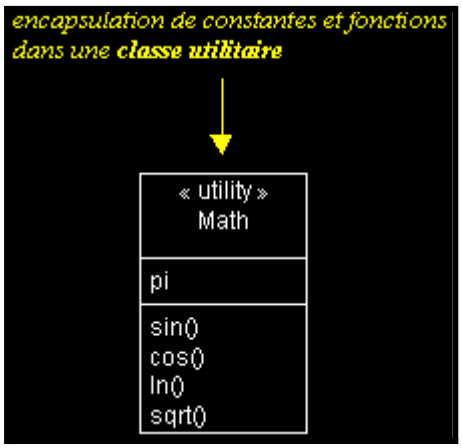
```
context SalleDeBain::utiliser(p : Personne)
 pre: if chambre->notEmpty then
        chambre.client->includes(p)
      else
        p.chambre._étage = self._étage
      endif
 post: _nbUtilisateurs = _nbUtilisateurs@pre + 1
```

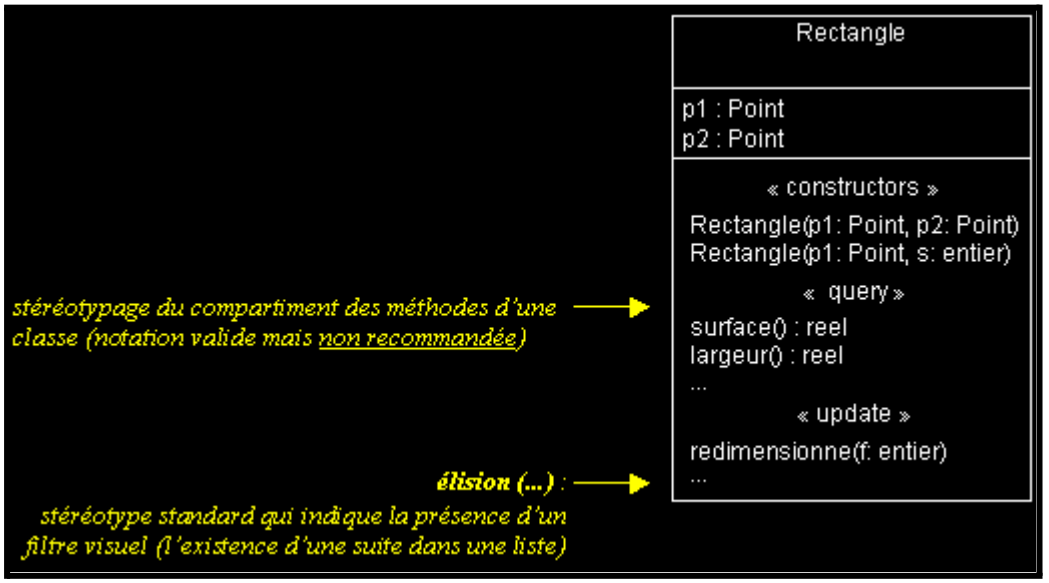
- Le loyer de l'hôtel est égal à la somme du prix de toutes les chambres louées.

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

Stéréotypes

- Les stéréotypes permettent d'étendre la sémantique des éléments de modélisation : il s'agit d'un mécanisme d'extensibilité du métamodèle d'UML.
- Les stéréotypes permettent de définir de nouvelles classes d'éléments de modélisation, en plus du noyau prédéfini par UML.
- Utilisez les stéréotypes avec modération et de manière concertée (notez aussi qu'UML propose de nombreux stéréotypes standards).





DIAGRAMMES DE COMPOSANTS ET DE DÉPLOIEMENT

Diagramme de composants

- Les diagrammes de composants permettent de décrire l'architecture physique et statique d'une application en termes de modules : fichiers sources, librairies, exécutables, etc.
  Ils montrent la mise en œuvre 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) :

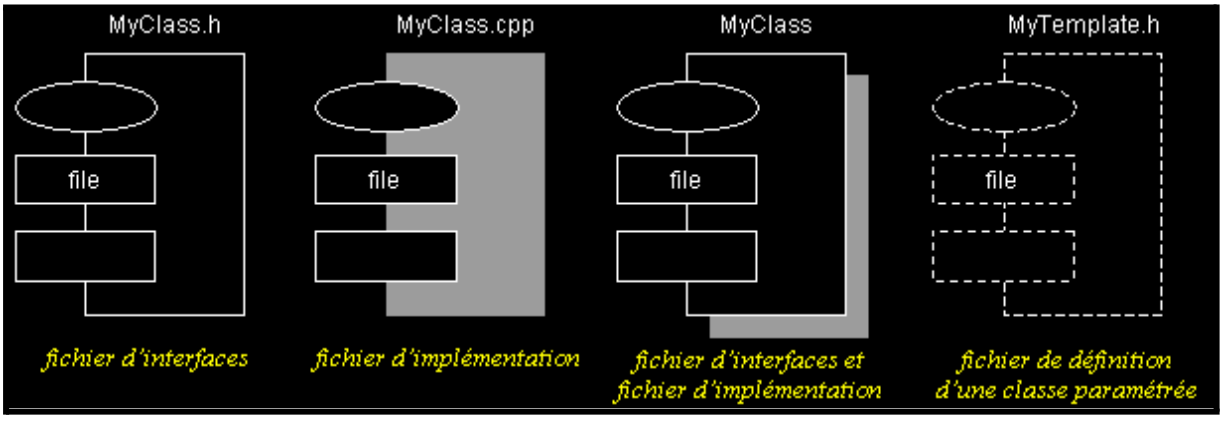


Diagramme de composants (exemple) :

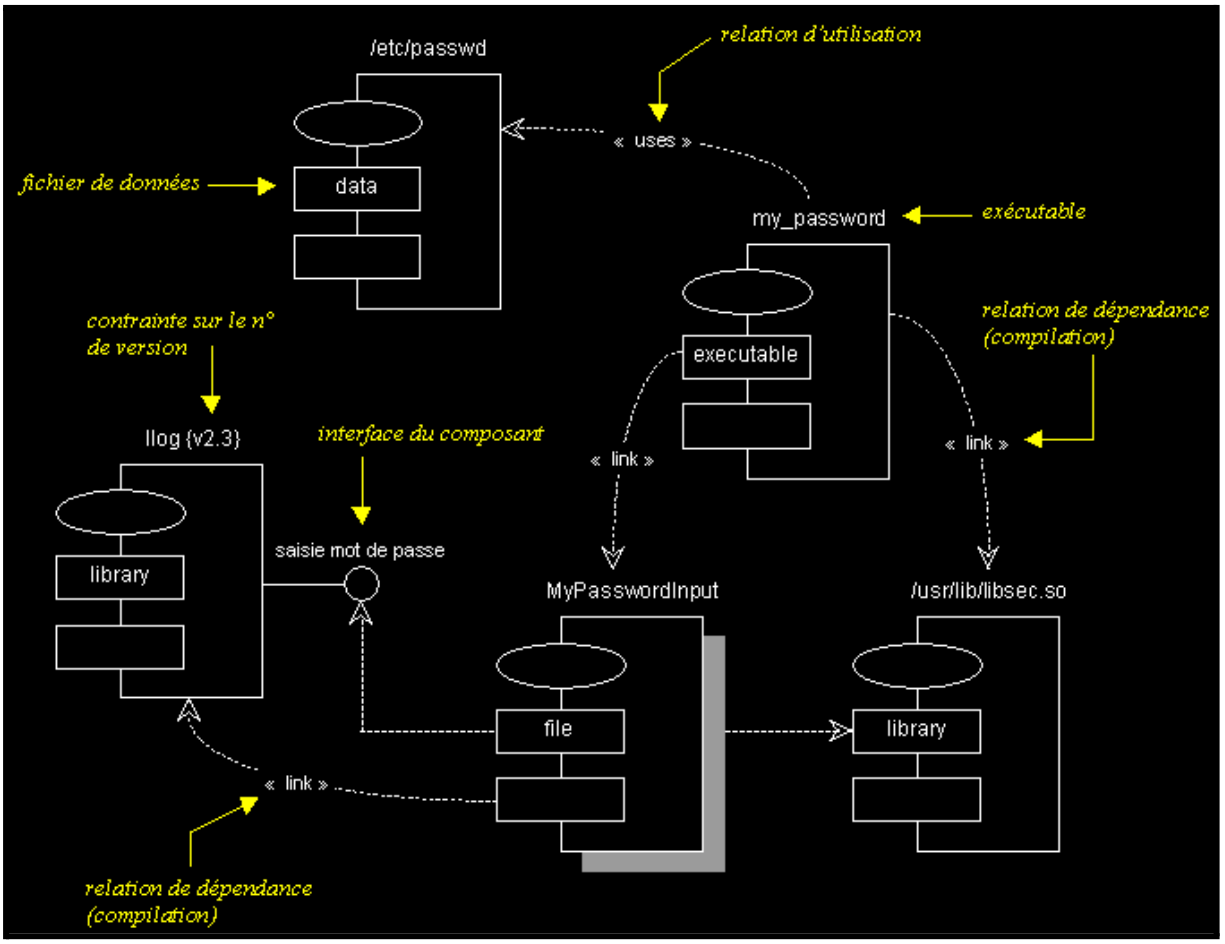


Diagramme de déploiement

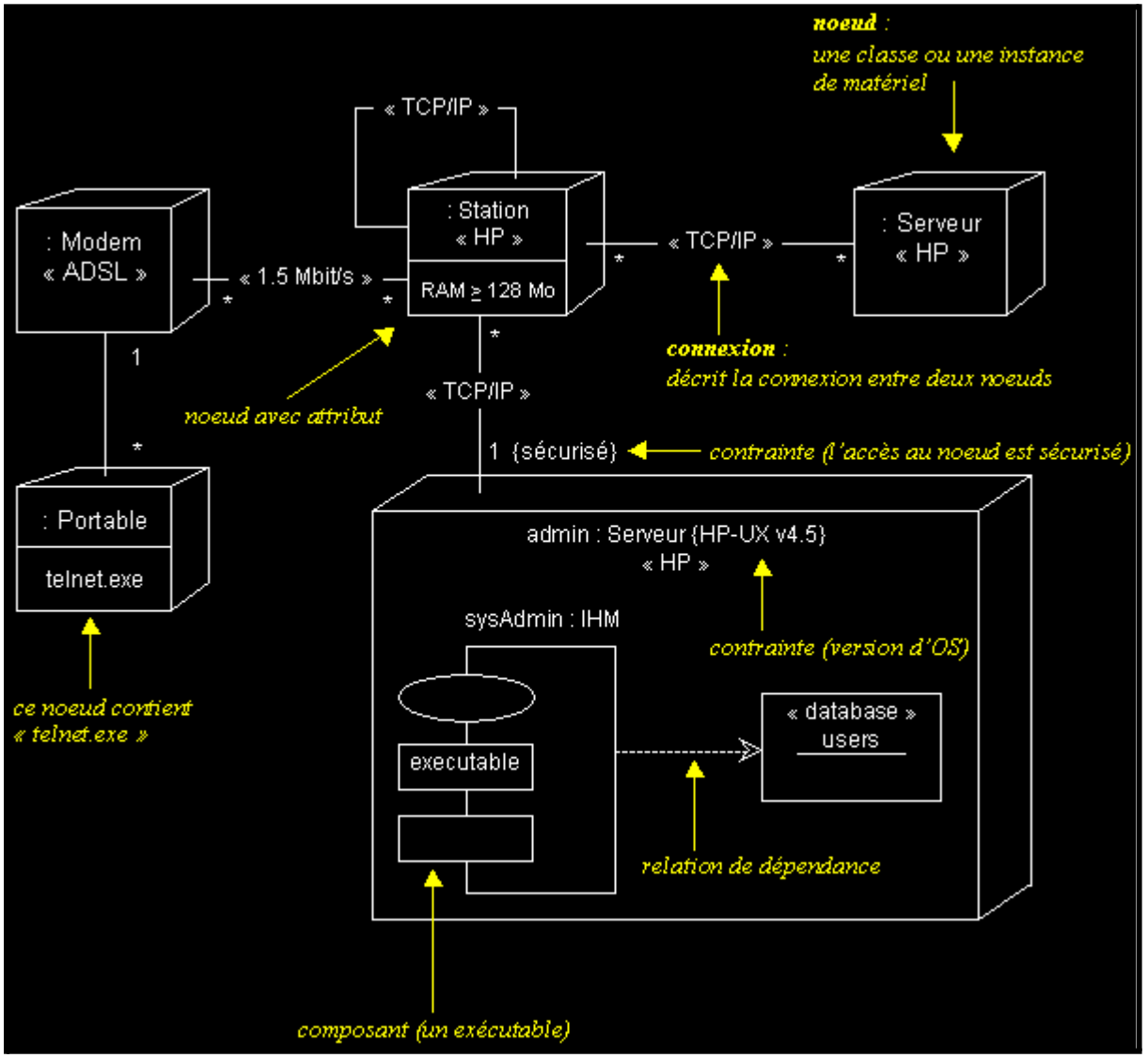
- Les diagrammes de déploiement montrent la disposition physique des matériels qui composent le système et la répartition des composants sur ces matériels.

- Les ressources matérielles sont représentées sous forme de nœuds.

- Les nœuds sont connectés entre eux, à l'aide d'un support de communication.
  La nature des lignes de communication et leurs caractéristiques peuvent être précisées.

- Les diagrammes de déploiement peuvent montrer des instances de nœuds (un matériel précis), ou des classes de nœuds.

- Les diagrammes de déploiement correspondent à la vue de déploiement d'une architecture logicielle (vue "4+1").



Les vues dynamiques d'UML

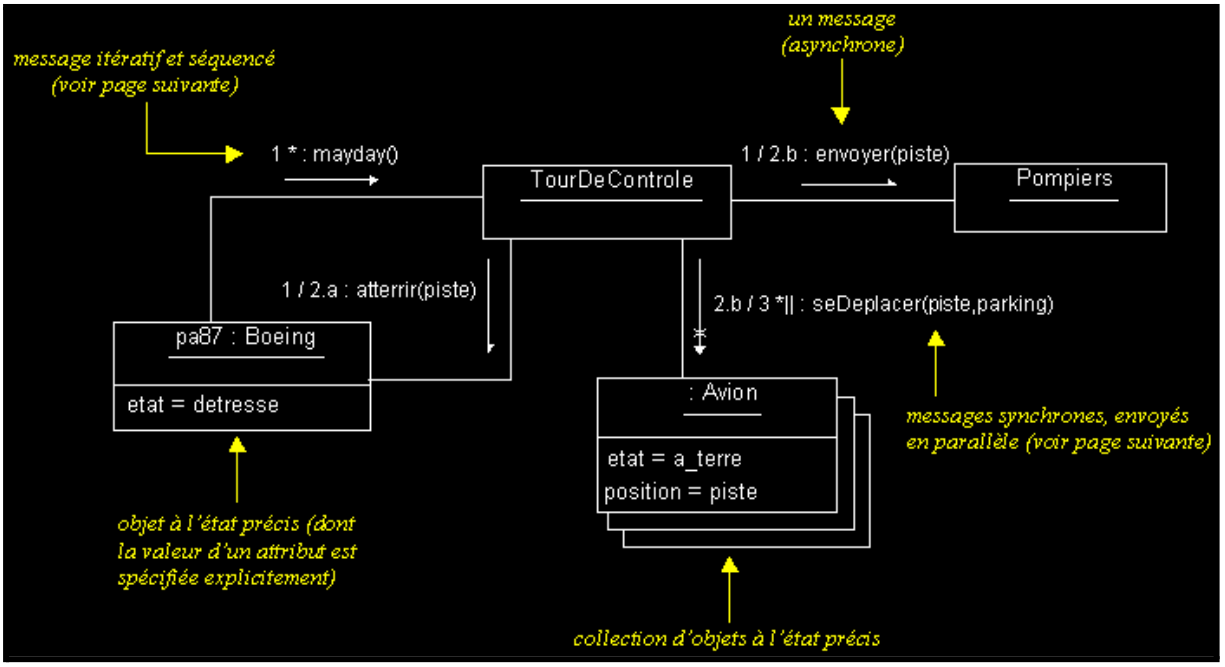
COLLABORATION ET MESSAGES

Diagramme de collaboration

- Les diagrammes de collaboration montrent des interactions entre objets (instances de classes et acteurs).

- Ils permettent de représenter le contexte d'une interaction, car on peut y préciser les états des objets qui interagissent.

Exemples :



Synchronisation des messages

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

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

- La syntaxe d'un message est la suivante :

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

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

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

  - séq : numéro de séquence du message.
    Indique le rang du message, c'est-à-dire son numéro d'ordre par rapport aux autres messages. Les messages sont numérotés à la façon de chapitres dans un document, à l'aide de chiffres séparés par des points. Ainsi, il est possible de représenter le niveau d'emboîtement des messages et leur précédence.

Exemple : l'envoi du message 1.3.5 suit immédiatement celui du message 1.3.4 et ces deux messages font partie du flot (de la famille de messages) 1.3.
Pour représenter l'envoi simultané de deux messages, il suffit de les indexer par une lettre.
Exemple : l'envoi des messages 1.3.a et 1.3.b est simultané.

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

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

- msg : nom du message.

- par : paramètres (optionnels) du message.

Exemples :

- 3 : bonjour()
  Ce message a pour numéro de séquence "3".

- [heure = midi] 1 : manger()
  Ce message n'est envoyé que s'il est midi.

- 1.3.6 * : ouvrir()

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

- 3 / *||[i := 1..5] : fermer()
  Représente l'envoi en parallèle de 5 messages. Ces messages ne seront envoyés qu'après l'envoi du message 3.

- 1.3,2.1 / [t < 10s] 2.5 : age := demanderAge(nom,prenom)
  Ce message (numéro 2.5) ne sera envoyé qu'après les messages 1.3 et 2.1, et que si "t < 10s".

- 1.3 / [disk full] 1.7.a * : deleteTempFiles()
  1.3 / [disk full] 1.7.b : reduceSwapFile(20%)
  Ces messages ne seront envoyés qu'après l'envoi du message 1.3 et si la condition "disk full" est réalisée. Si cela est le cas, les messages 1.7.a et 1.7.b seront envoyés simultanément.
  Plusieurs messages 1.7.a peuvent être envoyés.

Objets actifs (threads)

- UML permet de représenter des communications entre objets actifs de manière concurrente.

- Cette extension des diagrammes de collaboration permet notamment de représenter des communications entre processus ou l'exécution de threads.

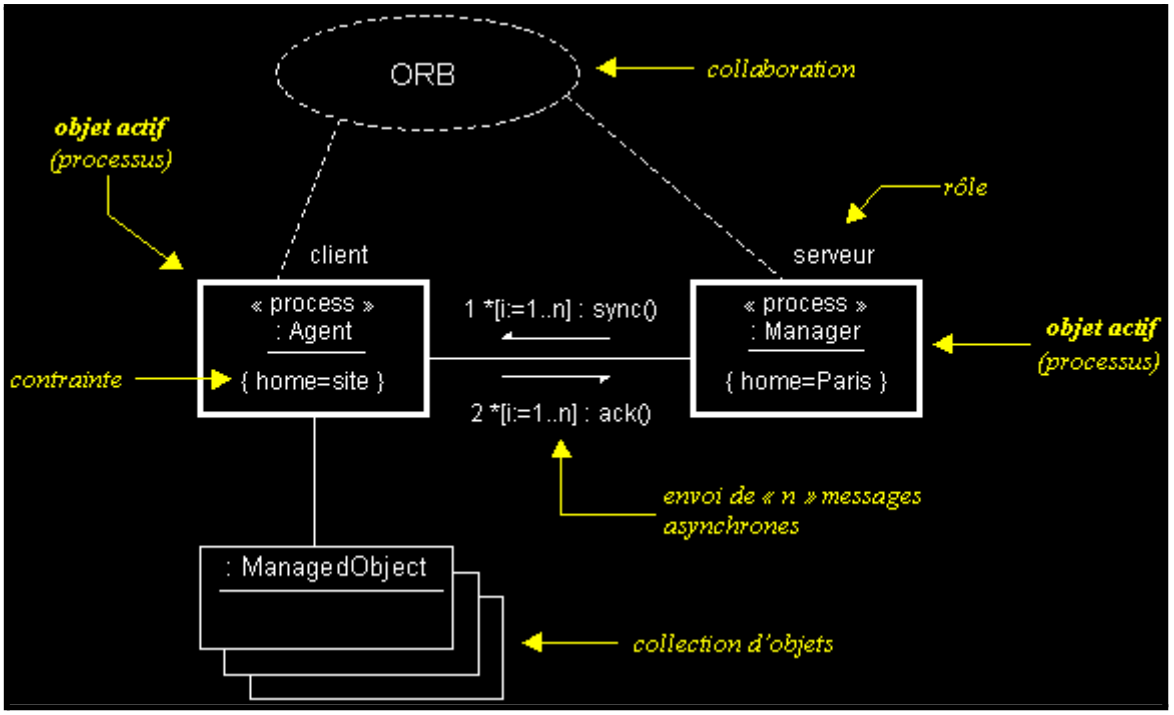

ORB
collaboration
objet actif
(processus)
rôle
client
serveur
« process »
: Agent
{ home=site }
1 *[i:=1..n] : sync()
2 *[i:=1..n] : ack()
« process »
: Manager
{ home=Paris }
objet actif
(processus)
contrainte
envoi de « n » messages
asynchrones
: ManagedObject
collection d'objets

DIAGRAMME DE SÉQUENCE

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.

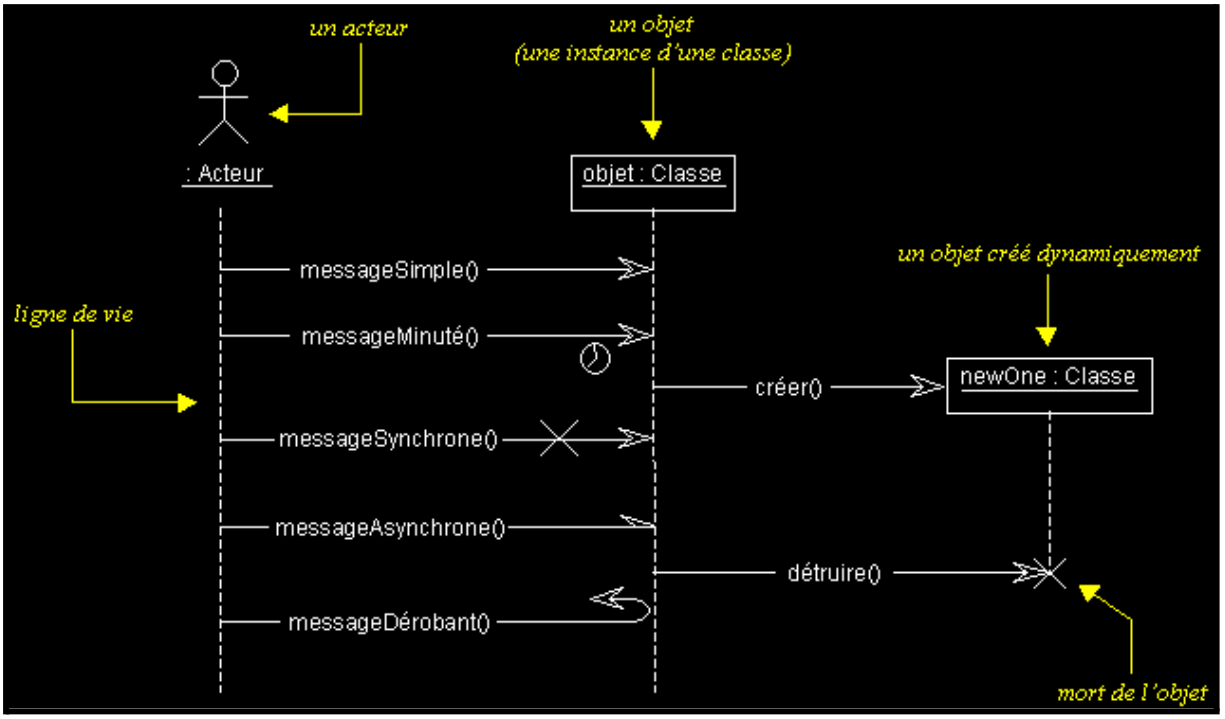
- Les diagrammes de séquences peuvent servir à illustrer un cas d'utilisation.

- 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.

Exemples :



Types de messages

Comme vous pouvez le voir dans l'exemple ci-dessus, UML propose un certain nombre de stéréotypes graphiques pour décrire la nature du message (ces stéréotypes graphiques s'appliquent également aux messages des diagrammes de collaborations) :

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

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

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

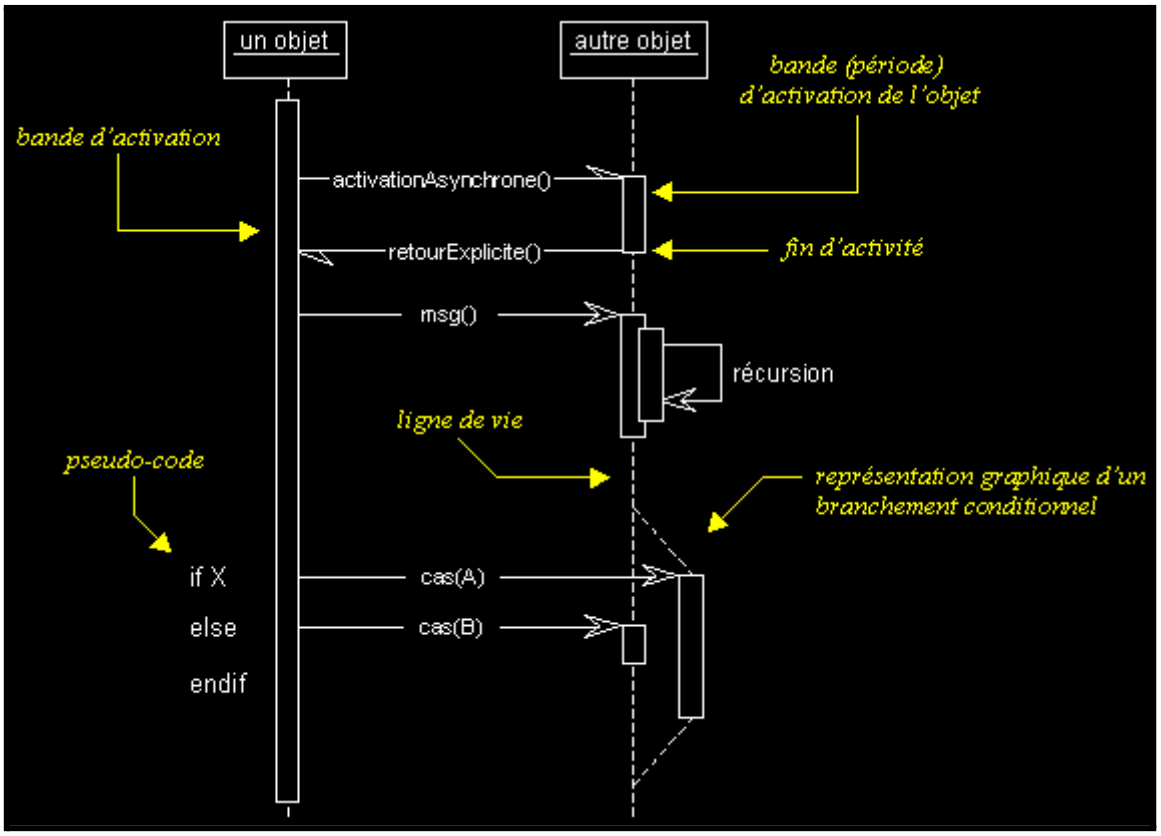
- **message asynchrone**
  N'interrompt pas l'exécution de l'expéditeur. Le message envoyé peut être pris en compte par le récepteur à tout moment ou ignoré (jamais traité).

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

Activation d'un objet

Sur un diagramme de séquence, il est aussi possible de représenter de manière explicite les différentes périodes d'activité d'un objet au moyen d'une bande rectangulaire superposée à la ligne de vie de l'objet.
On peut aussi représenter des messages récursifs, en dédoublant la bande d'activation de l'objet concerné.
Pour représenter de manière graphique une exécution conditionnelle d'un message, on peut documenter un diagramme de séquence avec du pseudo-code et représenter des bandes d'activation conditionnelles.

Exemple :



Commentaires :

- Ne confondez la période d'activation d'un objet avec sa création ou sa destruction. Un objet peut être actif plusieurs fois au cours de son existence (voir exemple ci-dessus).

- Le pseudo-code peut aussi être utilisé pour indiquer des itérations (avec incrémentation d'un paramètre d'un message par exemple).

- Le retour des messages asynchrones devrait toujours être matérialisé, lorsqu'il existe.

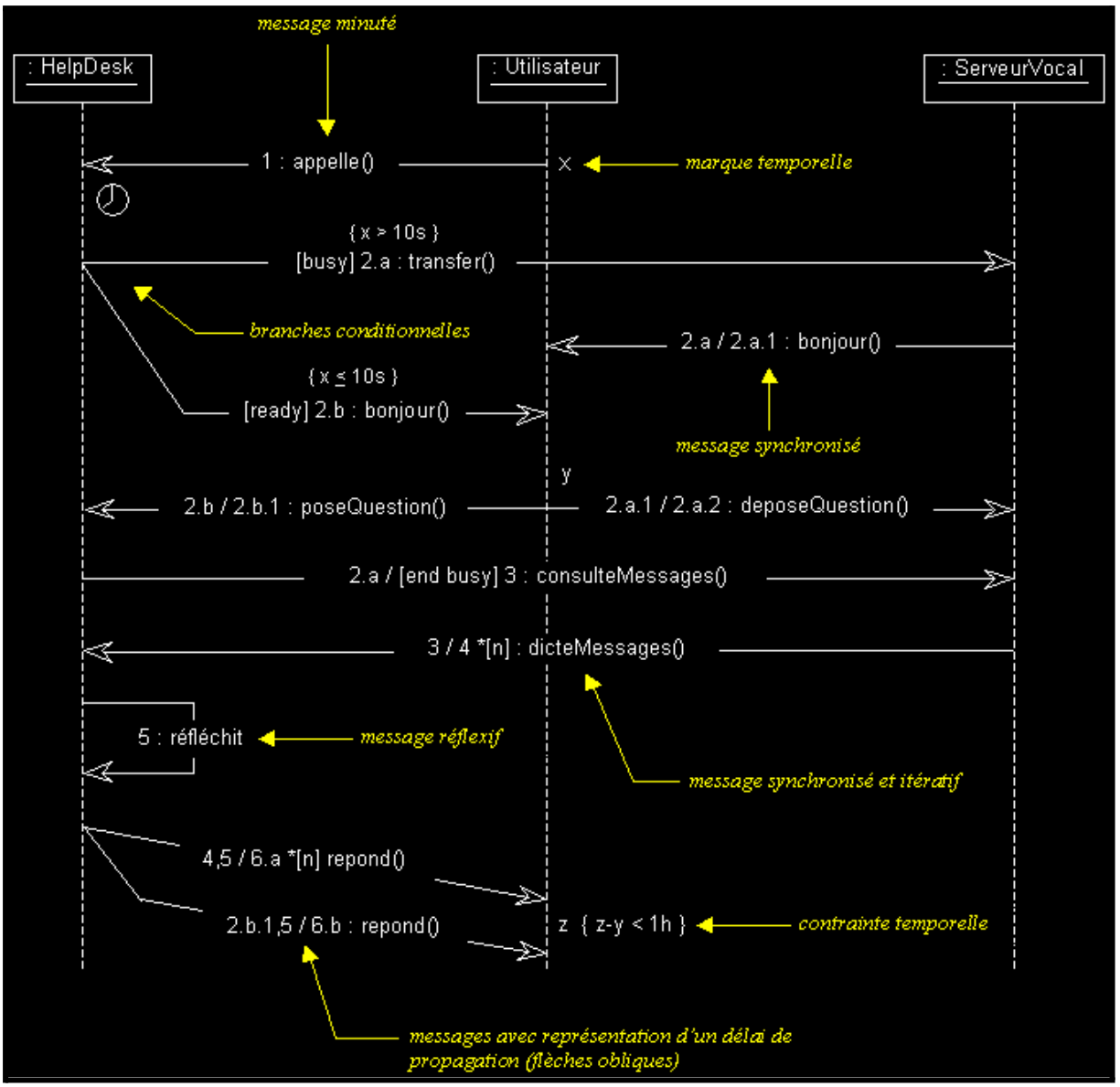
- Notez qu'il est fortement recommandé de synchroniser vos messages, comme sur l'exemple qui suit...

- L'exemple qui suit présente aussi une alternative intéressante pour la représentation des branchements conditionnels. Cette notation est moins lourde que celle utilisée dans l'exemple ci-dessus.

- Préférez aussi l'utilisation de contraintes à celle de pseudo-code, comme dans l'exemple qui suit.

Exemple complet :

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



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

DIAGRAMME D'ÉTATS-TRANSITIONS

Diagramme d'états-transitions : sémantique

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

- Les diagrammes d'états-transitions permettent de décrire les changements d'états d'un objet ou d'un composant, en réponse aux interactions avec d'autres objets/composants ou avec des acteurs.

- Un état se caractérise par sa durée et sa stabilité, il représente une conjonction instantanée des valeurs des attributs d'un objet.

- Une transition représente le passage instantané d'un état vers un autre.

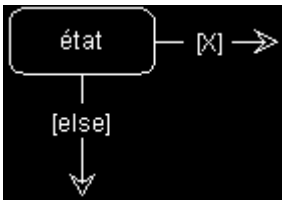
- Une transition est déclenchée par un événement. En d'autres termes : c'est l'arrivée d'un événement qui conditionne la transition.

- Les transitions peuvent aussi être automatiques, lorsqu'on ne spécifie pas l'événement qui la déclenche.

- En plus de spécifier un événement précis, il est aussi possible de conditionner une transition, à l'aide de "gardes" : il s'agit d'expressions booléennes, exprimées en langage naturel (et encadrées de crochets).

états, transition et événement, notation :

transition conditionnelle :



Super-État, historique et souches

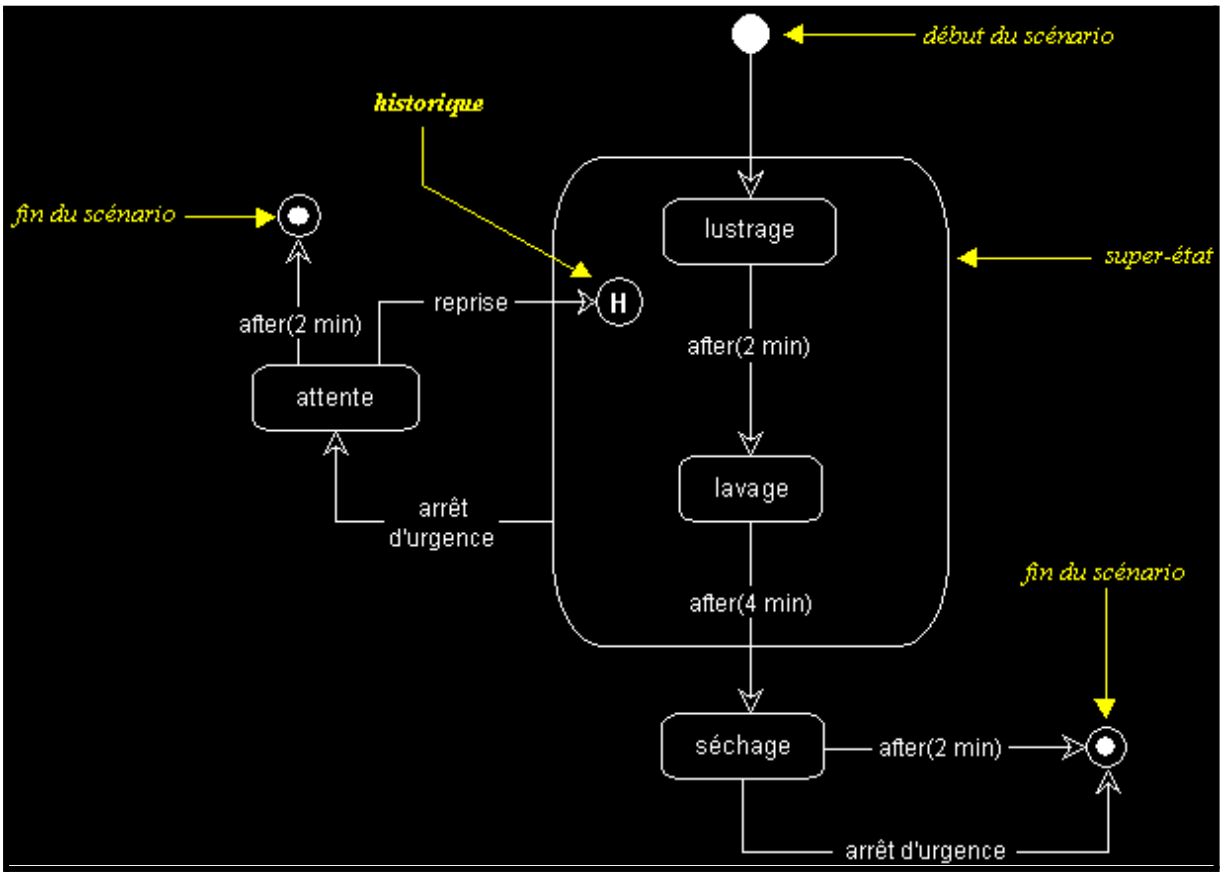
- Un super-état est un élément de structuration des diagrammes d'états-transitions (il s'agit d'un état qui englobe d'autres états et transitions).

- Le symbole de modélisation "historique", mémorise le dernier sous-état actif d'un super-état, pour y revenir directement ultérieurement.

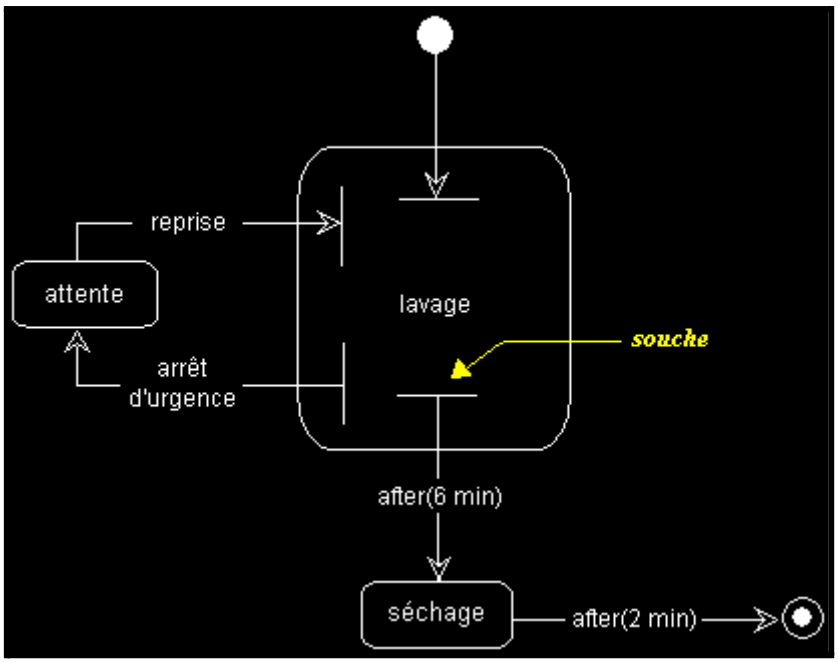
Exemple :

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

client peut aussi interrompre la machine. Mais dans ce cas, la machine s'arrêtera définitivement (avant de reprendre un autre cycle entier).



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



Actions dans un état

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

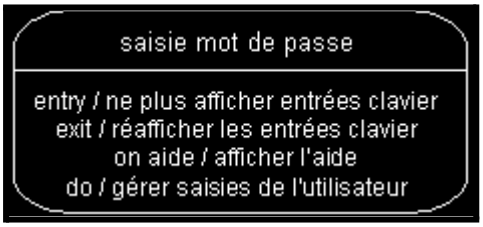
- Ceci exprime que la transition (déclenchée par l'événement cité) entraîne l'exécution de l'action spécifiée sur l'objet, à l'entrée du nouvel état.
  Exemple : *il pleut / ouvrir parapluie*

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

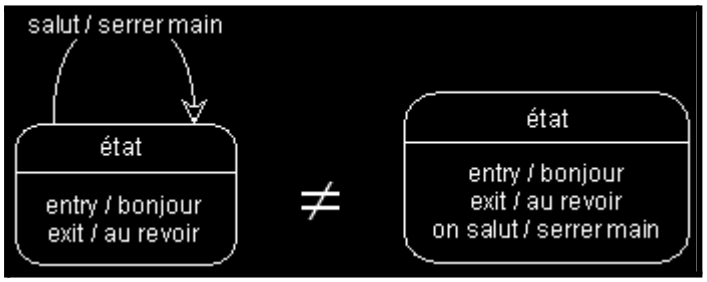
- Les actions propres à un état peuvent aussi être documentées directement à l'intérieur de l'état.
  UML définit un certain nombre de champs qui permettent de décrire les actions dans un état :
  - *entry / action* : action exécutée à l'entrée de l'état

- *exit / action* : action exécutée à la sortie de l'état
- *on événement / action* : action exécutée à chaque fois que l'événement cité survient
- *do / action* : action récurrente ou significative, exécutée dans l'état

Exemple :

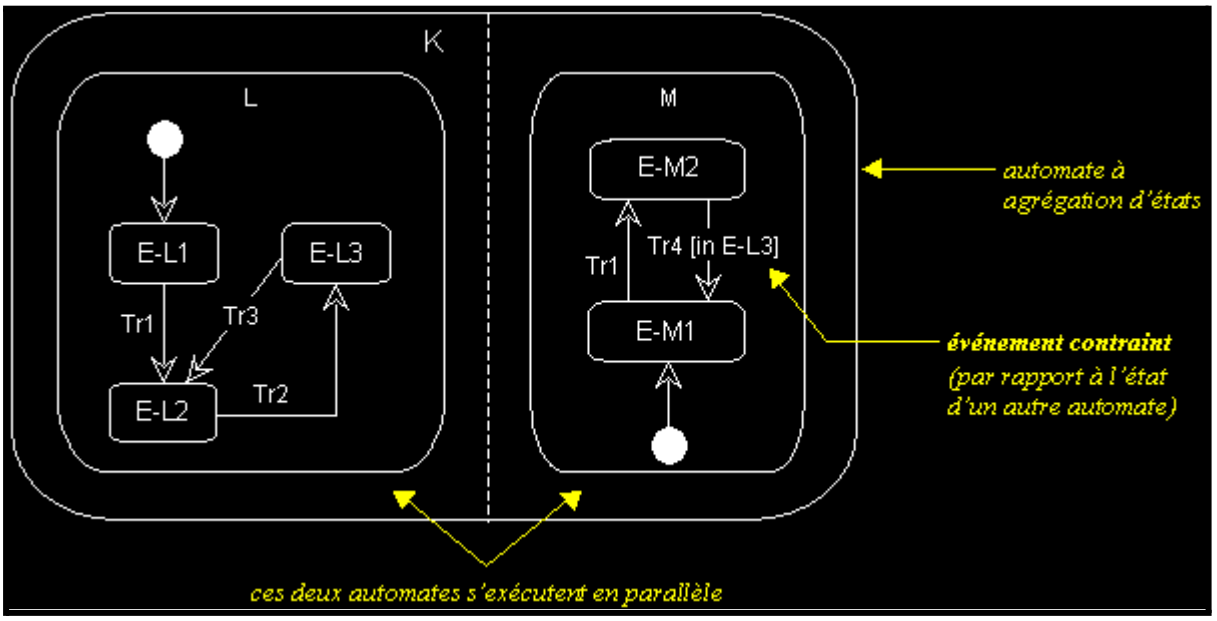


Remarque :
Attention, les actions attachées aux clauses "entry" et "exit" ne sont pas exécutées si l'événement spécifié dans la clause "on" survient. Pour indiquer qu'elles peuvent être exécutées plusieurs fois à l'arrivée d'un événement, représentez l'arrivée d'un événement réflexif, comme suit :



Etats concurrents et barre de synchronisation

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



Dans l'exemple ci-dessus, l'automate K est composé des sous-automates L et M.
L et M s'activent simultanément et évoluent en parallèle. Au départ, l'objet dont on modélise les états par l'automate K est dans l'état composite (E-L1, E-M1).
Après l'événement Tr1, K passe dans l'état composite (E-L2, E-M2). Par la suite, si l'événement Tr2 survient, K passe dans l'état composite (E-L3, E-M2). Si c'est Tr4 qui survient, M ne passe pas dans l'état E-M1, car cette transition est contrainte par l'état de L ("[in E-L3]").
Dans l'état composite (E-L3, E-M2), si Tr3 survient, K passe dans l'état composite (E-L2, E-M2). Si c'est Tr4 qui survient, K passe dans l'état composite (E-L3, E-M1). Et ainsi de suite...

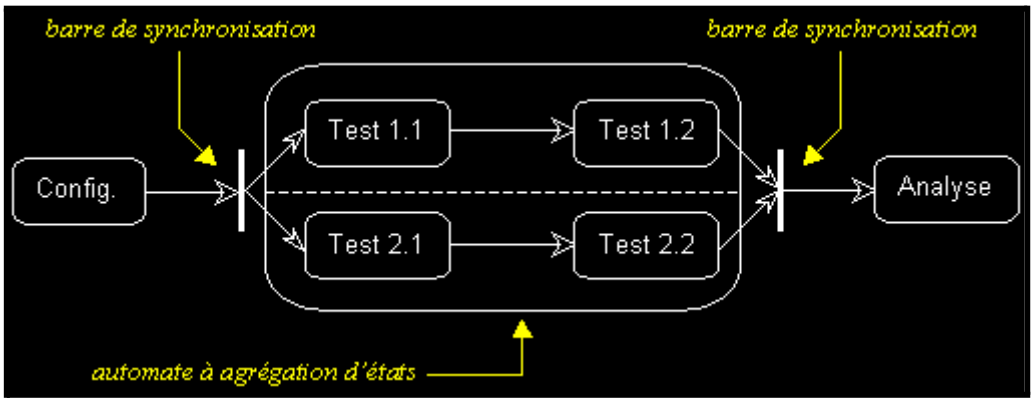
Attention : **la numérotation des événements n'est pas significative**. Pour synchroniser les sous-automates d'une agrégation d'états, il faut contraindre les transitions, comme dans l'exemple ci-dessus ("[in E-L3]").

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

- La barre de synchronisation permet de représenter graphiquement des points de synchronisation.

- Les transitions automatiques qui partent d'une barre de synchronisation ont lieu en même temps.

- On ne franchit une barre de synchronisation qu'après réalisation de toutes les transitions qui s'y rattachent.



Echange de messages entre automates

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

DIAGRAMME D'ACTIVITÉS

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).

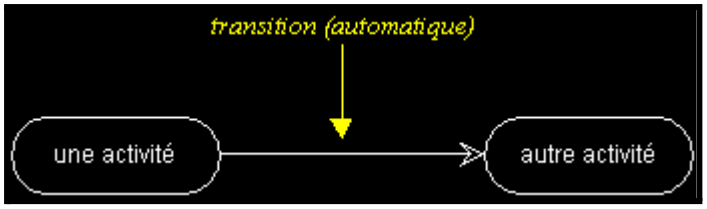
- Une activité représente une exécution d'un mécanisme, un déroulement d'étapes séquentielles.

- Le passage d'une activité vers une autre est matérialisé par une transition.

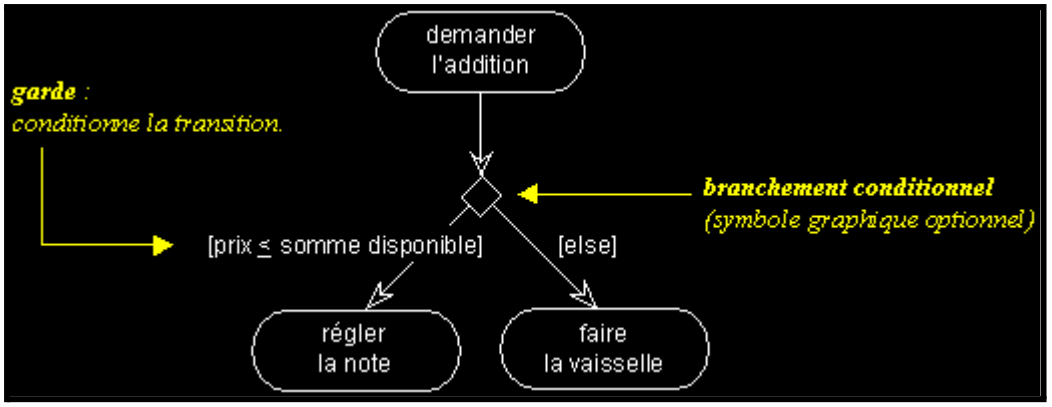
- Les transitions sont déclenchées par la fin d'une activité et provoquent le début immédiat d'une autre (elles sont automatiques).

- 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 :



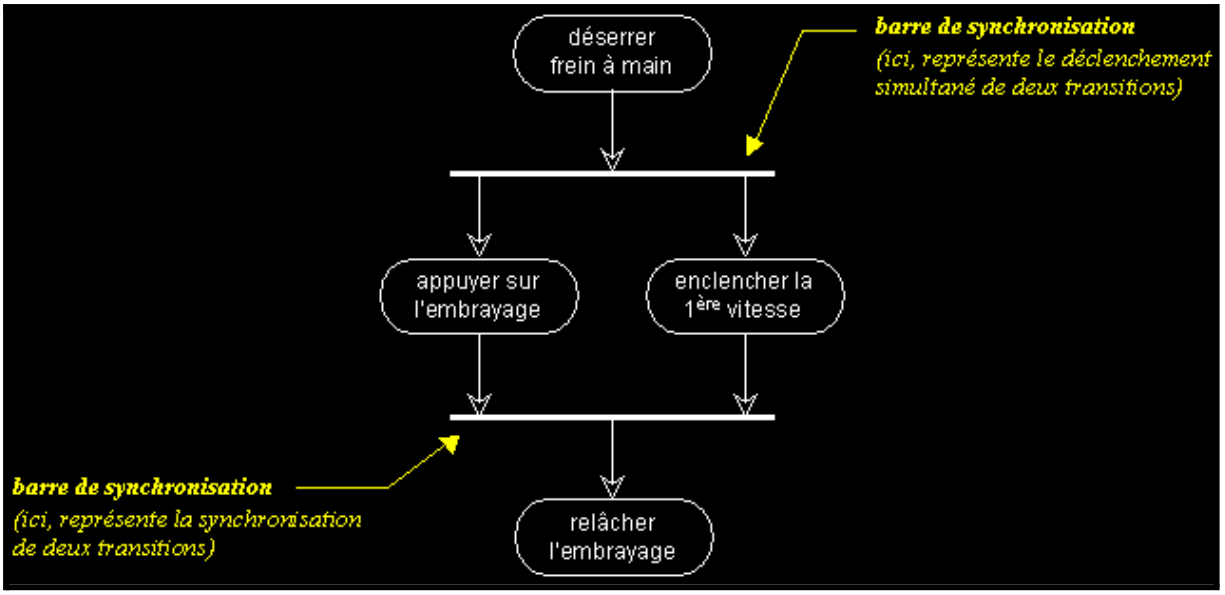
Synchronisation

Il est possible de synchroniser les transitions à l'aide des "**barres de synchronisation**" (comme dans les diagrammes d'états-transitions).
Une barre de synchronisation permet d'ouvrir et de fermer des branches parallèles au sein d'un flot d'exécution :

- Les transitions qui partent d'une barre de synchronisation ont lieu en même temps.

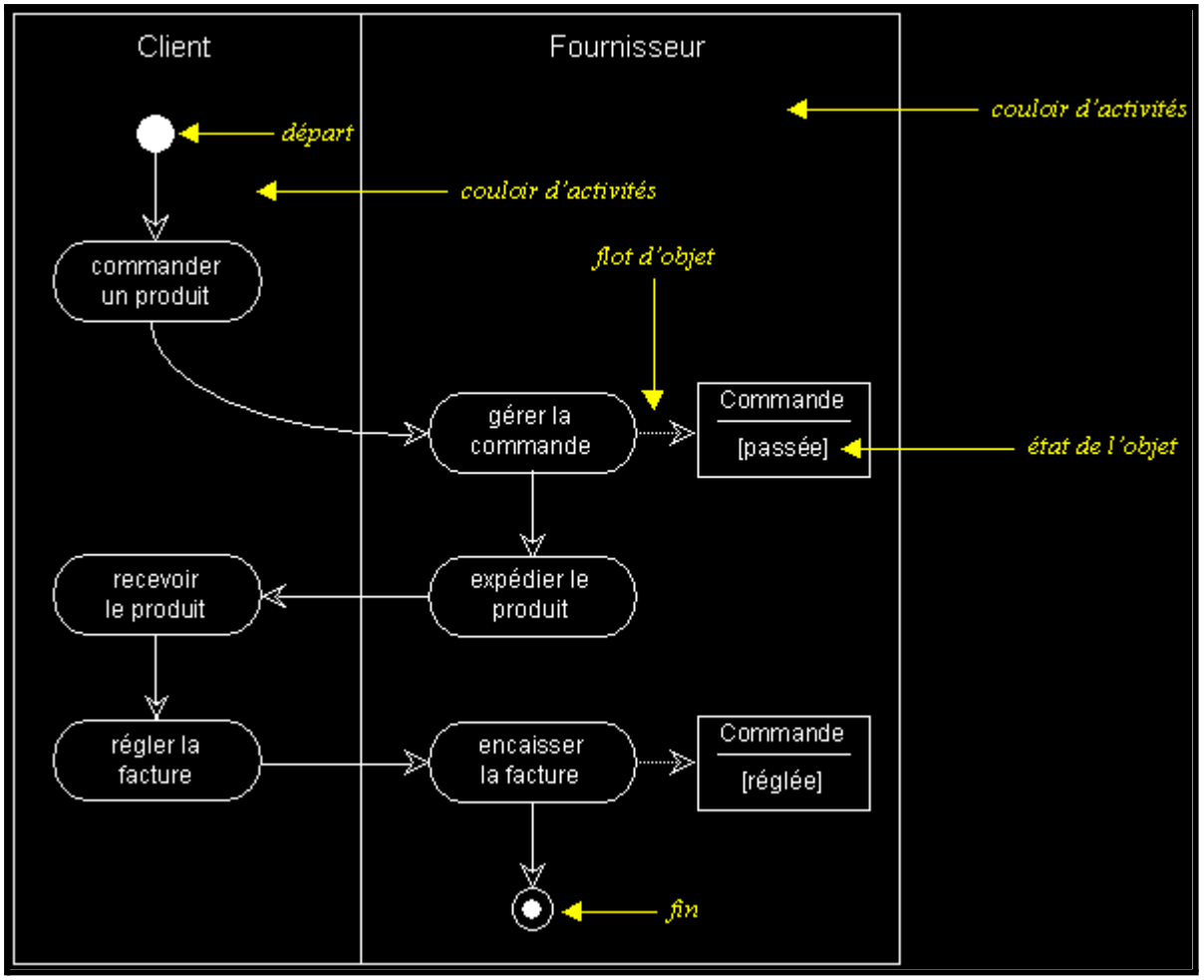
- On ne franchit une barre de synchronisation qu'après réalisation de toutes les transitions qui s'y rattachent.

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



Couloirs d'activités

Afin d'organiser un diagramme d'activités selon les différents responsables des actions représentées, il est possible de définir des "couloirs d'activités". Il est même possible d'identifier les objets principaux, qui sont manipulés d'activités en activités et de visualiser leur changement d'état.


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

Conclusion

Programmer orienté objet ?

Programmer en Java signifie naturellement travailler dans un paradigme orienté objet, mais cela ne garantit pas automatiquement une conception orientée objet. Seule une analyse approfondie en termes de conception orientée objet peut aboutir à une solution véritablement orientée objet. Il est crucial de respecter les concepts fondamentaux de cette approche. Le langage de programmation Java fournit les moyens pour implémenter ces concepts, mais ne garantit pas à lui seul leur application correcte.

UML : un support de communication

UML est avant tout un outil de communication. Son utilisation ne garantit pas le respect des concepts objets : c'est à vous de l'utiliser judicieusement. Concevoir orienté objet, c'est d'abord élaborer un modèle qui respecte les concepts objets. Le langage de programmation et UML ne sont que des outils pour atteindre cet objectif.

Utiliser UML efficacement

1. **Multipliez les vues sur vos modèles** :
    - Un diagramme ne fournit qu'une vue partielle et précise d'un modèle.
    - Croisez les vues complémentaires (dynamiques/statiques).
2. **Restez simple** :
    - Utilisez les niveaux d'abstraction pour synthétiser vos modèles sans vous limiter aux vues d'implémentation.
    - Ne surchargez pas vos diagrammes.
3. **Commentez vos diagrammes** :
    - Ajoutez des notes et du texte explicatif pour clarifier vos diagrammes.
4. **Utilisez des outils appropriés** :

- Employez des outils adéquats pour réaliser et maintenir vos modèles UML.

Pour programmer de manière efficace en Java et respecter les concepts objets, il est crucial de concevoir un modèle orienté objet solide. UML et le langage de programmation sont des outils qui, bien utilisés, facilitent cette démarche. Cependant, ils ne remplacent pas une bonne analyse et une conception réfléchie.