Langage UML

Table des matières

Le langage de modélisation unifié, de l'anglais Unified Modeling Language (UML), est un langage de modélisation graphique à base de pictogrammes conçu pour fournir une méthode normalisée pour visualiser la conception d'un système. Il est couramment utilisé en développement logiciel et en conception orientée objet.

1. Introduction

Pour faire face à la complexité croissante des systèmes d'information, de nouvelles méthodes et outils ont été créées. La principale avancée des quinze dernières années réside dans la programmation orientée objet (P.O.O.).

Face à ce nouveau mode de programmation, les méthodes de modélisation classique (telle MERISE) ont rapidement montré certaines limites et ont dû s'adapter (cf. MERISE/2).

De très nombreuses méthodes ont également vu le jour comme Booch, OMT …

Dans ce contexte et devant le foisonnement de nouvelles méthodes de conception « orientée objet », l'Object Management Group (OMG) a eu comme objectif de définir une notation standard utilisable dans les développements informatiques basés sur l'objet. C'est ainsi qu'est apparu UML (Unified Modified Language « langage de modélisation objet unifié »), qui est issu de la fusion des méthodes Booch, OMT (Object Modelling Technique) et OOSE (Object Oriented Software Engineering).

Issu du terrain et fruit d'un travail d'experts reconnus, UML est le résultat d'un large consensus. De très nombreux acteurs industriels de renom ont adopté UML et participent à son développement.

En l'espace d'une poignée d'années seulement, UML est devenu un standard incontournable.

Ceci nous amène à nous questionner sur :

- les apports réels d'UML dans la modélisation
- la place des méthodes dites « traditionnelles » telle que MERISE.

UML est en effet apparu très tardivement, car l'approche objet se pratique depuis de très nombreuses années déjà.

Simula, premier langage de programmation à implémenter le concept de type abstrait à l'aide de classes, date de 1967 ! En 1976 déjà, Smalltalk implémente les concepts fondateurs de l'approche objet : encapsulation, agrégation, héritage. Les premiers compilateurs C++ datent du début des années 80 et de nombreux langages orientés objets "académiques" ont étayé les concepts objets (Eiffel, Objective C, Loops...).

Il y donc déjà longtemps que l'approche objet est devenue une réalité. Les concepts de base de l'approche objet sont stables et largement éprouvés. De nos jours, programmer "objet", c'est bénéficier d'une panoplie d'outils et de langages performants. L'approche objet est une solution technologique incontournable. Ce n'est plus une mode, mais un réflexe quasi-automatique dès lors qu'on cherche à concevoir des logiciels complexes qui doivent "résister" à des évolutions incessantes.

Toutefois, l'approche objet n'est pas une panacée :

- elle est moins intuitive que l'approche fonctionnelle.
- Malgré les apparences, il est plus naturel pour l'esprit humain de décomposer un problème informatique sous forme d'une hiérarchie de fonctions atomiques et de données, qu'en terme d'objets et d'interaction entre ces objets.

  Or, rien dans les concepts de base de l'approche objet ne dicte comment modéliser la structure objet d'un système de manière pertinente. Quels moyens doit-on alors utiliser

pour mener une analyse qui respecte les concepts objet ? Sans un cadre méthodologique approprié, la dérive fonctionnelle de la conception est inévitable...

- **l'application des concepts objet nécessite une très grande rigueur**.

  Le vocabulaire précis est un facteur d'échec important dans la mise en oeuvre d'une approche objet (risques d'ambiguïtés et d'incompréhensions). Beaucoup de développeurs (même expérimentés) ne pensent souvent objet qu'à travers un langage de programmation. Or, les langages orientés objet ne sont que des outils qui proposent une manière particulière d'implémenter certains concepts objet. Ils ne valident en rien l'utilisation de ces moyens techniques pour concevoir un système conforme à la philosophie objet.

  Connaître C++ ou Java n'est donc pas une fin en soi, il faut aussi savoir se servir de ces langages à bon escient. La question est donc de savoir "qui va nous guider dans l'utilisation des concepts objet, si ce ne sont pas les langages orientés objet ?".

  Enfin, comment comparer deux solutions de découpe objet d'un système si l'on ne dispose pas d'un moyen de représentation adéquat ? Il est très simple de décrire le résultat d'une analyse fonctionnelle, mais qu'en est-il d'une découpe objet ?

Pour remédier à ces inconvénients majeurs de l'approche objet, il faut donc :

1. **un langage** (pour s'exprimer clairement à l'aide des concepts objets)

   Le langage doit permettre de représenter des concepts abstraits (graphiquement par exemple), limiter les ambiguïtés (parler un langage commun, au vocabulaire précis, indépendant des langages orientés objet), faciliter l'analyse (simplifier la comparaison et l'évaluation de solutions).

2. une démarche d'analyse et de conception objet

Une démarche d'analyse et de conception objet est nécessaire afin de ne pas effectuer une analyse fonctionnelle et se contenter d'une implémentation objet, mais penser objet dès le départ, définir les vues qui permettent de décrire tous les aspects d'un système avec des concepts objets.

Il faut donc disposer d'un outil qui donne une dimension méthodologique à l'approche objet et qui permette de mieux maîtriser sa richesse.

La prise de conscience de l'importance d'une méthode spécifiquement objet ("comment structurer un système sans centrer l'analyse uniquement sur les données ou uniquement sur les traitements, mais sur les deux"), ne date pas d'hier. Plus de 50 méthodes objet sont apparues durant le milieu des années 90 (Booch, Classe-Relation, Fusion, HOOD, OMT, OOA, OOD, OOM, OOSE...). Aucune ne s'est réellement imposée.

L'absence de consensus sur une méthode d'analyse objet a longtemps freiné l'essor des technologies objet. Ce n'est que récemment que les grands acteurs du monde informatique ont pris conscience de ce problème. L'unification et la normalisation des méthodes objet dominantes (OMT, Booch et OOSE) ne datent que de 1995. UML est le fruit de cette fusion.

UML, ainsi que les méthodes dont il est issu, s'accordent sur un point : une analyse objet passe par une modélisation objet.

Qu'est-ce qu'un modèle ?

Un modèle est une abstraction de la réalité. L'abstraction est un des piliers de l'approche objet. Il s'agit d'un processus qui consiste à identifier les caractéristiques intéressantes d'une entité en vue

d'une utilisation précise. L'abstraction désigne aussi le résultat de ce processus, c'est-à-dire l'ensemble des caractéristiques essentielles d'une entité, retenues par un observateur.

Un modèle est une vue subjective, mais pertinente de la réalité. Un modèle définit une frontière entre la réalité et la perspective de l'observateur. Ce n'est pas "la réalité", mais une vue très subjective de la réalité. Bien qu'un modèle ne représente pas une réalité absolue, un modèle reflète des aspects importants de la réalité, il en donne donc une vue juste et pertinente.

Le caractère abstrait d'un modèle doit notamment permettre de faciliter la compréhension du système étudié. Il réduit la complexité du système étudié, permet de simuler le système, le représente et reproduit ses comportements. Concrètement, un modèle réduit (décompose) la réalité, dans le but de disposer d'éléments de travail exploitables par des moyens mathématiques ou informatiques.

UML permet donc de modéliser une application selon une vision objet.

L'appréhension d'UML est complexe car UML est à la fois :

- une norme,
- un langage de modélisation objet,
- un support de communication,
- un cadre méthodologique.

1.1. UML est une norme

Fin 1997, UML est devenu **une norme** OMG (Object Management Group).

L'OMG est un organisme à but non lucratif, créé en 1989 à l'initiative de grandes sociétés (HP, Sun, Unisys, American Airlines, Philips...). Aujourd'hui, l'OMG fédère plus de 850 acteurs du monde informatique. Son rôle est de promouvoir des standards qui garantissent l'interopérabilité entre applications orientées objet, développées sur des réseaux hétérogènes.

L'OMG propose notamment l'architecture CORBA (Common Object Request Broker Architecture), un modèle standard pour la construction d'applications à objets distribués (répartis sur un réseau).

CORBA fait partie d'une vision globale de la construction d'applications réparties, appelée OMA (Object Management Architecture) et définie par l'OMG. Sans rentrer dans les détails, on peut résumer cette vision par la volonté de **favoriser l'essor industriel des technologies objet**, en offrant un ensemble de **solutions technologiques non propriétaires**, qui suppriment les clivages techniques.

UML a été adopté (normalisé) par l'OMG et intégré à l'OMA, car il participe à cette vision et parce qu'il répond à la "philosophie" OMG.

1.2. UML est un langage de modélisation objet

Pour penser et concevoir objet, il faut savoir "prendre de la hauteur", jongler avec des concepts abstraits, indépendants des langages d'implémentation et des contraintes purement techniques. Les langages de programmation ne sont pas un support d'analyse adéquat pour "concevoir objet". Ils ne permettent pas de décrire des solutions en terme de concepts abstraits et constituent un cadre trop rigide pour mener une analyse itérative.

Pour conduire une analyse objet cohérente, il ne faut pas directement penser en terme de pointeurs, d'attributs et de tableaux, mais en terme d'association, de propriétés et de cardinalités... Utiliser le langage de programmation comme support de conception ne revient bien souvent qu'à juxtaposer de manière fonctionnelle un ensemble de mécanismes d'implémentation, pour résoudre un problème qui nécessite en réalité une modélisation objet.

L'approche objet nécessite une analyse réfléchie, qui passe par différentes phases exploratoires.

Bien que raisonner en terme d'objets semble naturel, l'approche fonctionnelle reste la plus intuitive pour nos esprits cartésiens... Voilà pourquoi il ne faut pas se contenter d'une implémentation objet, mais se discipliner à "penser objet" au cours d'une phase d'analyse préalable.

Toutes les dérives fonctionnelles de code objet ont pour origine le non respect des concepts de base de l'approche objet (encapsulation...) ou une utilisation détournée de ces concepts (héritage sans classification...). Ces dérives ne sont pas dues à de mauvaises techniques de programmation ; la racine du mal est bien plus profonde : programmer en C++ ou en Java n'implique pas forcément concevoir objet...

Les difficultés de mise en œuvre d'une approche "réellement objet" ont engendré bien souvent des déceptions, ce qui a longtemps constitué un obstacle important à l'essor des technologies objet. Beaucoup ont cédé au leurre des langages de programmation orientés objet et oublié que le code n'est qu'un "moyen". Le respect des concepts fondamentaux de l'approche objet prime sur la manière dont on les implémente. Ne penser qu'à travers un langage de programmation objet détourne de l'essentiel.

Pour sortir les technologies objet de cette impasse, l'OMG propose UML.

UML comble une lacune importante des technologies objet. Il permet d'exprimer et d'élaborer des modèles objet, indépendamment de tout langage de programmation. Il a été pensé pour servir de support à une analyse basée sur les concepts objet.

UML est un **langage formel**, défini par un **métamodèle**.

Le métamodèle d'UML décrit de manière très précise tous les éléments de modélisation (les concepts véhiculés et manipulés par le langage) et la sémantique de ces éléments (leur définition et le sens de leur utilisation).

En d'autres termes : **UML normalise les concepts objet**.

Un métamodèle permet de limiter les ambiguïtés et encourage la construction d'outils. Il permet aussi de classer les différents concepts du langage (selon leur niveau d'abstraction ou leur domaine d'application) et expose ainsi clairement sa structure. Enfin, on peut noter que le métamodèle d'UML est lui-même décrit par un méta-métamodèle de manière standardisée, à l'aide de MOF (Meta Object Facility : norme OMG de description des métamodèles).

Véritable clé de voûte de l'OMA, UML est donc un outil indispensable pour tous ceux qui ont compris que programmer objet, c'est d'abord concevoir objet. UML n'est pas à l'origine des concepts objets, mais il en constitue une étape majeure, car il unifie les différentes approches et en donne une définition plus formelle.

1.3. UML est un support de communication

UML est avant tout **un support de communication performant**, qui facilite la représentation et la compréhension de solutions objet.

Sa notation graphique permet d'**exprimer visuellement une solution objet**, ce qui **facilite la comparaison et l'évaluation** de solutions.

L'aspect formel de sa notation **limite les ambiguïtés** et les incompréhensions.

Son **indépendance** par rapport aux langages de programmation, aux domaines d'application et aux processus, en font un langage universel.

La notation graphique d'UML n'est que le support du langage. La véritable force d'UML, c'est qu'il repose sur un métamodèle. En d'autres termes : **la puissance et l'intérêt d'UML, c'est qu'il normalise la sémantique des concepts qu'il véhicule !**

Qu'une association d'héritage entre deux classes soit représentée par une flèche terminée par un triangle ou un cercle, n'a que peu d'importance par rapport au sens que cela donne à votre modèle. La notation graphique est essentiellement guidée par des considérations esthétiques, même si elle a été pensée dans ses moindres détails.

Par contre, utiliser une relation d'héritage, reflète l'intention de donner à votre modèle un sens particulier. Un "bon" langage de modélisation doit permettre à n'importe qui de déchiffrer cette intention de manière non équivoque. Il est donc primordial de s'accorder sur la sémantique des éléments de modélisation, bien avant de s'intéresser à la manière de les représenter.

Le métamodèle UML apporte une solution à ce problème fondamental.

UML est donc bien plus qu'un simple outil qui permet de "dessiner" des représentations mentales... **Il permet de parler un langage commun**, normalisé mais accessible, car visuel.

1.4. UML est un cadre méthodologique

Une autre caractéristique importante d'UML, est qu'il cadre l'analyse. UML permet de représenter un système selon différentes vues complémentaires : **les diagrammes**. Un diagramme UML est une représentation graphique, qui s'intéresse à un aspect précis du modèle ; c'est une perspective du modèle.

Chaque type de diagramme UML possède une structure (les types des éléments de modélisation qui le composent sont prédéfinis) et véhicule une sémantique précise (il 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. Les diagrammes permettent donc d'inspecter un modèle selon différentes perspectives et guident l'utilisation des éléments de modélisation (les concepts objet), car ils possèdent une structure.

Une caractéristique importante des diagrammes UML, est qu'ils **supportent l'abstraction**. Cela permet de mieux contrôler la complexité dans l'expression et l'élaboration des solutions objet.

UML opte en effet pour **l'élaboration des modèles**, plutôt que pour une approche qui impose une barrière stricte entre analyse et conception. Les modèles d'analyse et de conception ne diffèrent que par leur niveau de détail, il n'y a pas de différence dans les concepts utilisés. UML n'introduit pas d'éléments de modélisation propres à une activité (analyse, conception...) ; le langage reste le même à tous les niveaux d'abstraction.

Cette approche simplificatrice facilite le passage entre les niveaux d'abstraction. L'élaboration encourage une approche non linéaire, les "retours en arrière" entre niveaux d'abstraction différents sont facilités et la traçabilité entre modèles de niveaux différents est assurée par l'unicité du langage.

UML favorise donc le prototypage, et c'est là une de ses forces. En effet, modéliser une application n'est pas une activité linéaire. Il s'agit d'une tâche très complexe, qui nécessite une approche itérative, car il est plus efficace de construire et valider par étapes, ce qui est difficile à cerner et maîtriser.

UML permet donc non seulement de représenter et de manipuler les concepts objet, il sous-entend une démarche d'analyse qui permet de concevoir une solution objet de manière itérative, grâce aux diagrammes, qui supportent l'abstraction.

1.4.1. UML n'est pas une méthode

UML est un langage qui permet de représenter des modèles, mais il ne définit pas le processus d'élaboration des modèles. Qualifier UML de "méthode objet" n'est donc pas tout à fait approprié.

Une méthode propose aussi un processus, qui régit notamment l'enchaînement des activités de production d'une entreprise. Or UML n'a pas été pensé pour régir les activités de l'entreprise.

Les auteurs d'UML sont tout à fait conscients de l'importance du processus, mais ce sujet a été intentionnellement exclu des travaux de l'OMG. Comment prendre en compte toutes les organisations et cultures d'entreprises ? Un processus est adapté (donc très lié) au domaine d'activité de l'entreprise ; même s'il constitue un cadre général, il faut l'adapter au contexte de l'entreprise. Bref, améliorer un processus est une discipline à part entière, c'est un objectif qui dépasse très largement le cadre de l'OMA.

Cependant, même si pour l'OMG, l'acceptabilité industrielle de la modélisation objet passe d'abord par la disponibilité d'un langage d'analyse objet performant et standard, les auteurs d'UML préconisent d'utiliser une démarche :

- guidée par les besoins des utilisateurs du système,
- centrée sur l'architecture logicielle,
- itérative et incrémentale.

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

Une source fréquente de malentendus sur UML a pour origine la faculté d'UML de modéliser un processus, pour le documenter et l'optimiser par exemple. En fin de compte, qu'est-ce qu'un processus ? Un ensemble d'activités coordonnées et régulées, en partie ordonnées, dont le but est de créer un produit (matériel ou intellectuel). UML permet tout à fait de modéliser les activités (c'est-à-dire la dynamique) d'un processus, de décrire le rôle des acteurs du processus, la structure des éléments manipulés et produits, etc...

Une extension d'UML ("UML extension for business modeling") propose d'ailleurs un certain nombre de stéréotypes standards (extensions du métamodèle) pour mieux décrire les processus.

Le RUP ("**Rational Unified Process**"), processus de développement "clé en main", proposé par Rational Software, est lui aussi modélisé (documenté) avec UML. Il offre un cadre méthodologique générique qui repose sur UML et la suite d'outils Rational.

1.4.2. Conclusion

Comme UML n'impose pas de méthode de travail particulière, il peut être intégré à n'importe quel processus de développement logiciel de manière transparente. UML est une sorte de boîte à outils,

qui permet d'améliorer progressivement vos méthodes de travail, tout en préservant vos modes de fonctionnement.

Intégrer UML par étapes dans un processus, de manière pragmatique, est tout à fait possible. La faculté d'UML de se fondre dans le processus courant, tout en véhiculant une démarche méthodologique, facilite son intégration et limite de nombreux risques (rejet des utilisateurs, coûts...).

Intégrer UML dans un processus ne signifie donc pas révolutionner ses méthodes de travail, mais cela devrait être l'occasion de se remettre en question.

2. Le contexte d'apparition d'UML

2.1. Approche fonctionnelle versus approche objet

2.1.1. L'approche fonctionnelle

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

   La découpe fonctionnelle d'un problème (sur laquelle reposent les langages de programmation structurée) consiste à découper le problème en blocs indépendants. En ce sens, elle présente un caractère intuitif fort.

2. La réutilisabilité du code

   Le découpage d'un problème en blocs indépendants (fonctions et procédures) va permettre aux programmeurs de réutiliser les fonctions déjà développées (à condition qu'elles soient suffisamment génériques). La productivité se trouve donc accrue.

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

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

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

   - on a de nouvelles données à gérer (ex : DVD)
   - les traitements évoluent : l'affichage sera différent selon le type (livre, CD, DVD …)

4. Problèmes générés par la séparation des données et des traitements :

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

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

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

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

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

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

5. 1ère amélioration : rassembler les valeurs qui caractérisent un type, dans le type

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

   Par exemple, le délai avant rappel peut être défini pour chaque type de document. Cela permet donc de créer une fonction plus générique qui s'applique à tous les types de documents.

6. 2ème amélioration : centraliser les traitements associés à un type, auprès du type

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

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

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

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

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

Document
Code document Nom document Type document
Calculer date rappel

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

2.1.2. L'approche objet

2.1.2.1. Le concept d'objet

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

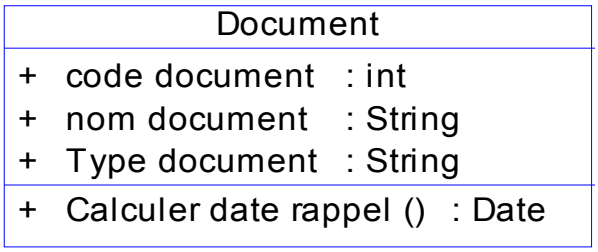
Un objet est une entité aux frontières précises qui possède une identité (un nom).

Un ensemble d'attributs caractérise l'état de l'objet.

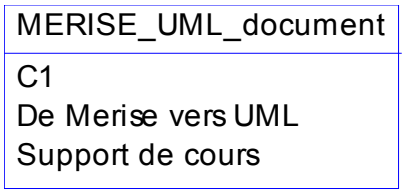
Un ensemble d'opérations (méthodes) en définissent le comportement.

Un objet est une instance de classe (une occurrence d'un type abstrait).

Une classe est un type de données abstrait, caractérisé par des propriétés (attributs et méthodes) communes à des objets et permettant de créer des objets possédant ces propriétés.



Classe : regroupement d'objets



Objet : instance d'une classe

2.1.2.2. Les autres concepts importants de l'approche objet

1. l'encapsulation

   L'encapsulation consiste à masquer les détails d'implémentation d'un objet, en définissant une interface.

   L'interface est la vue externe d'un objet, elle définit les services accessibles (offerts) aux utilisateurs de l'objet.

   L'encapsulation facilite l'évolution d'une application car elle stabilise l'utilisation des objets : on peut modifier l'implémentation des attributs d'un objet sans modifier son interface.

L'encapsulation garantit l'intégrité des données, car elle permet d'interdire l'accès direct aux attributs des objets.

2. l'héritage

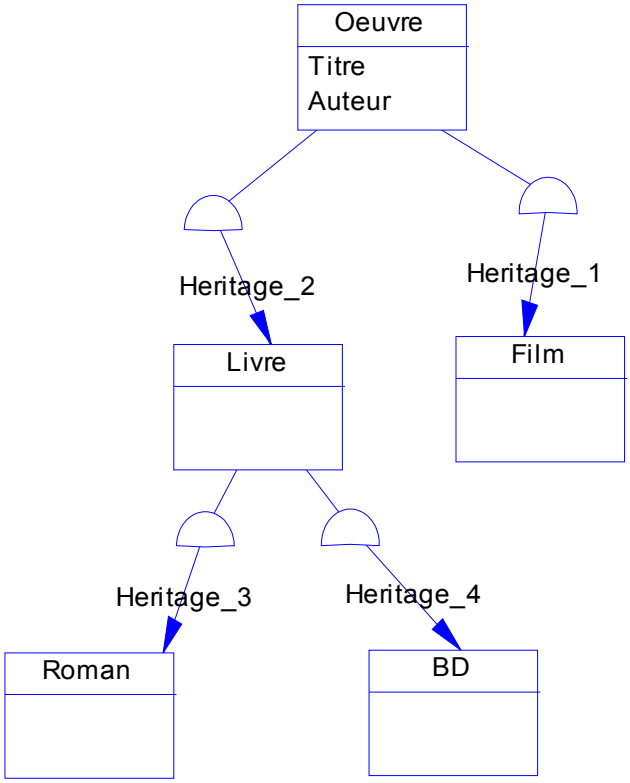
L'héritage est un mécanisme de transmission des propriétés d'une classe (ses attributs et méthodes) vers une sous-classe.

Une classe peut être spécialisée en d'autres classes, afin d'y ajouter des caractéristiques spécifiques ou d'en adapter certaines.

Plusieurs classes peuvent être généralisées en une classe qui les factorise, afin de regrouper les caractéristiques communes d'un ensemble de classes.

La spécialisation et la généralisation permettent de construire des hiérarchies de classes. L'héritage peut être simple ou multiple.

L'héritage évite la duplication et encourage la réutilisation.



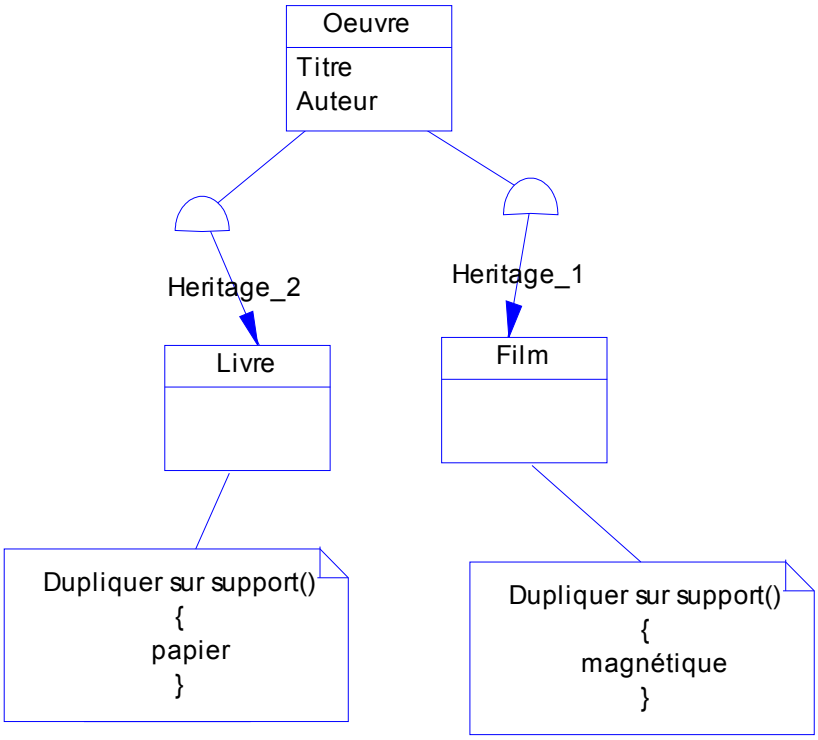
3. le polymorphisme

Le **polymorphisme** représente la faculté d'une même opération de s'exécuter différemment suivant le contexte de la classe où elle se trouve.

Ainsi, une opération définie dans une superclasse peut s'exécuter de façon différente selon la sous-classe où elle est héritée.

Ex : exécution d'une opération de calcul des salaires dans 2 sous-classes spécialisées : une pour les cadres, l'autre pour les non-cadres.

Le polymorphisme augmente la généricité du code.

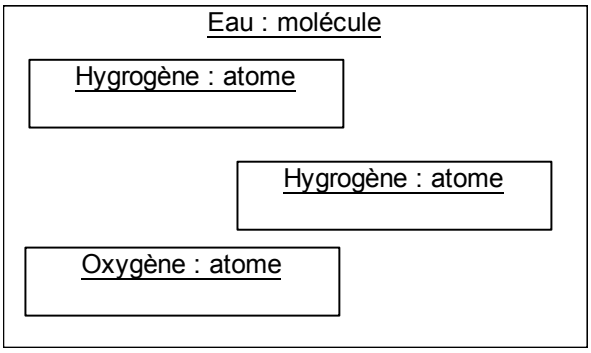


4. l'agrégation

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

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

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



2.1.2.3. Historique de l'approche objet

Les concepts objet sont stables et éprouvés (issus du terrain) :

- Simula, 1er langage de programmation à implémenter le concept de type abstrait (à l'aide de classes), date de 1967 !
- En 1976 déjà, Smalltalk implémente les concepts fondateurs de l'approche objet (encapsulation, agrégation, héritage) à l'aide de :

- classes
- associations entre classes
- hiérarchies de classes
- messages entre objets

- Le 1er compilateur C++ date de 1980, et C++ est normalisé par l'ANSI.
- De nombreux langages orientés objets académiques ont étayés les concepts objets : Eiffel, Objective C, Loops…

Les concepts objet sont anciens, mais ils n'ont jamais été autant d'actualité :

- L'approche fonctionnelle n'est pas adaptée au développement d'applications qui évoluent sans cesse et dont la complexité croit continuellement.
- L'approche objet a été inventée pour faciliter l'évolution d'applications complexes.

De nos jours, les outils orientés objet sont fiables et performants :

- Les compilateurs C++ produisent un code robuste et optimisé.
- De très nombreux outils facilitent le développement d'applications C++ :
  - bibliothèques (STL, USL, Rogue Wave, MFC...)
  - environnements de développement intégrés (Developper Studio, Sniff+...)
  - outils de qualimétrie et de tests (Cantata++, Insure++, Logiscope...)
  - bases de données orientées objet (O2, ObjectStore, Versant...)

2.1.2.4. Inconvénients de l'approche objet

L'approche objet est moins intuitive que l'approche fonctionnelle.

L'application des concepts objets nécessite une grande rigueur : le vocabulaire est précis (risques d'ambiguïtés, d'incompréhensions).

2.1.2.5. Solutions pour remédier aux inconvénients de l'approche objet

Il faut bénéficier d'un langage pour exprimer les concepts objet qu'on utilise, afin de pouvoir :

- représenter des concepts abstraits (graphiquement par exemple).
- limiter les ambiguïtés (parler un langage commun).
- faciliter l'analyse (simplifier la comparaison et l'évaluation de solutions).

Il faut également une démarche d'analyse et de conception objet, pour :

- ne pas effectuer une analyse fonctionnelle et se contenter d'une implémentation objet, mais penser objet dès le départ.
- définir les vues qui permettent de couvrir tous les aspects d'un système, avec des concepts objets.

2.2. La genèse d'UML

2.2.1. Historique des méthodes d'analyse

2.2.1.1. Les premières méthodes d'analyse (années 70)

Découpe cartésienne (fonctionnelle et hiérarchique) d'un système.

2.2.1.2. L'approche systémique (années 80)

Modélisation des données + modélisation des traitements (Merise ...).

2.2.1.3. L'émergence des méthodes objet (1990-1995)

Prise de conscience de l'importance d'une méthode spécifiquement objet :

- comment structurer un système sans centrer l'analyse uniquement sur les données ou uniquement sur les traitements (mais sur les deux) ?
- Plus de 50 méthodes objet sont apparues durant cette période (Booch, Classe-Relation, Fusion, HOOD, OMT, OOA, OOD, OOM, OOSE...) !
- Aucun méthode ne s'est réellement imposée.

2.2.1.4. Les premiers consensus (1995)

- OMT (James Rumbaugh) : vues statiques, dynamiques et fonctionnelles d'un système
  - issue du centre de R&D de General Electric.
  - Notation graphique riche et lisible.
- OOD (Grady Booch) : vues logiques et physiques du système
  - Définie pour le DOD, afin de rationaliser de développement d'applications ADA, puis C++.
  - Ne couvre pas la phase d'analyse dans ses 1ères versions (préconise SADT).
  - Introduit le concept de package (élément d'organisation des modèles).
- OOSE (Ivar Jacobson) : couvre tout le cycle de développement
  - Issue d'un centre de développement d'Ericsson, en Suède.
  - La méthodologie repose sur l'analyse des besoins des utilisateurs.

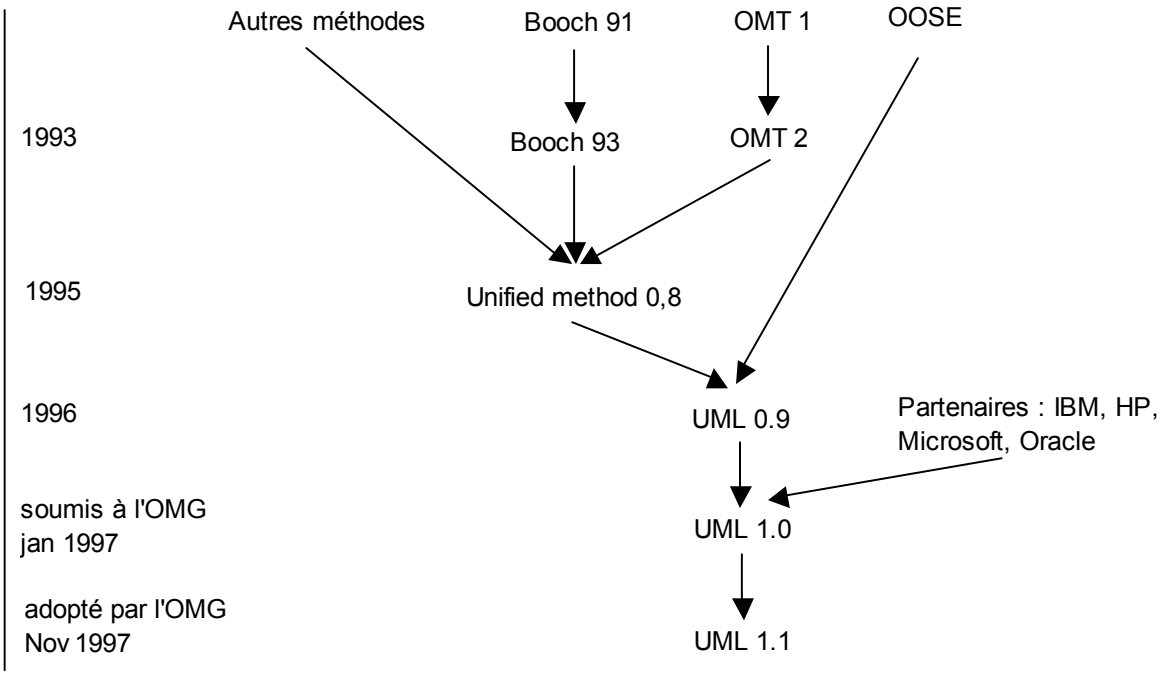
2.2.1.5. L'unification et la normalisation des méthodes (1995-1997)

En octobre 1994, G. Booch (père fondateur de la méthode Booch) et J. Rumbaugh (principal auteur de la méthode OMT) ont décidé de travailler ensemble pour unifier leurs méthodes au sein de la société Rational Software. Un an après, I . Jacobson (auteur de la méthode OOSE et des cas d'utilisation) a rejoint Rational Software pour travailler sur l'unification. Unified Modified Language (UML) est né.

Les travaux sur ce langage ont continué avec son adoption par de grands acteurs industriels comme HP, Microsoft, Oracle ou Unisys. Ce travail a abouti en 1997 à UML 1.0. Le langage a été soumis par Rational Software et ses partenaires à l'OMG comme réponse à un appel d'offres sur la standardisation des langages de modélisation.

L'appel d'offres de l'OMG a recueilli un avis favorable, puisque 6 réponses concurrentes sont parvenues à l'OMG. IBM et Object Time (méthode ROOM pour les systèmes temps réel réactifs) ont décidé de rejoindre l'équipe UML ; leur proposition était en fait une extension d'UML 1.0.

Certains autres auteurs qui ont répondu à l'appel d'offres ont abandonné leur proposition pour rejoindre à leur tour UML. En novembre 1997, UML a été adopté par l'OMG.



UML est donc le résultat d'un large consensus et tient compte des dernières avancées en matière de modélisation et de développement logiciel.

L'OMG RTF (nombreux acteurs industriels) centralise et normalise les évolutions d'UML au niveau international et de nombreux groupes d'utilisateurs UML favorisent le partage des expériences.

2.2.2. Cadre d'utilisation d'UML

2.2.2.1. UML n'est pas une méthode ou un processus

Si l'on parle de méthode objet pour UML, c'est par abus de langage.

Ce constat vaut aussi pour OMT ou d'autres techniques / langages de modélisation.

Une méthode propose aussi un processus, qui régit notamment l'enchaînement des activités de production d'une entreprise. Or, UML a été pensé pour permettre de modéliser les activités de l'entreprise, pas pour les régir.

Des méthodes orientées objet les plus connues, seules les méthodes OOSE et BOOCH incluent cet aspect « processus » de manière explicite et formelle. Ces 2 auteurs se sont d'ailleurs toujours démarqués des autres sur ce point.

Par leur nature, les processus doivent être adaptés aux organisations, à leurs domaines d'activité, à leur culture …

De ce fait, ni la normalisation ni la standardisation d'un processus de développement logiciel ne peut faire l'objet d'un consensus international suffisant pour aboutir à un standard acceptable et utilisable.

2.2.2.2. UML est un langage de modélisation

UML est un langage de modélisation au sens de la théorie des langages. Il contient de ce fait les éléments constitutifs de tout langage, à savoir : des concepts, une syntaxe et une sémantique.

De plus, UML a choisi une notation supplémentaire : il s'agit d'une forme visuelle fondée sur des diagrammes. Si l'unification d'une notation est secondaire par rapports aux éléments constituant le langage, elle reste cependant primordiale pour la communication et la compréhension.

2.2.2.3. UML décrit un méta modèle

UML est fondé sur un **métamodèle**, qui définit :

- les éléments de modélisation (les concepts manipulés par le langage),
- la sémantique de ces éléments (leur définition et le sens de leur utilisation).

Un métamodèle est une description très formelle de tous les concepts d'un langage. Il limite les ambiguïtés et encourage la construction d'outils.

Le métamodèle d'UML permet de classer les concepts du langage (selon leur niveau d'abstraction ou domaine d'application) et expose sa structure.

Le métamodèle UML est lui-même décrit par un méta-métamodèle (OMG-MOF).

UML propose aussi une notation, qui permet de représenter graphiquement les éléments de modélisation du métamodèle.

Cette notation graphique est le support du langage UML.

UML offre :

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

2.2.2.4. UML est un support de communication

Sa notation graphique permet d'exprimer visuellement une solution objet.

L'aspect formel de sa notation limite les ambiguïtés et les incompréhensions.

Son aspect visuel facilite la comparaison et l'évaluation de solutions.

Son indépendance (par rapport aux langages d'implémentation, domaine d'application, processus...) en font un langage universel.

2.2.3. Points forts d'UML

2.2.3.1. UML est un langage formel et normalisé

Il permet ainsi :

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

2.2.3.2. UML est un support de communication performant

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

2.2.4. Points faibles d'UML

2.2.4.1. Apprentissage et période d'adaptation

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

2.2.4.2. Le processus (non couvert par UML)

L'intégration d'UML dans un processus n'est pas triviale et améliorer un processus est un tâche complexe et longue.

3. Démarche générale de modélisation

3.1. Qu'est-ce qu'un modèle ?

3.1.1. Définition d'un modèle

Un **modèle** est une abstraction de la réalité

L'abstraction est un des piliers de l'approche objet : il s'agit d'un processus qui consiste à identifier les caractéristiques intéressantes d'une entité, en vue d'une utilisation précise.

L'abstraction désigne aussi le résultat de ce processus, c'est-à-dire l'ensemble des caractéristiques essentielles d'une entité, retenues par un observateur.

Un modèle est une vue subjective mais pertinente de la réalité. Un modèle définit une frontière entre la réalité et la perspective de l'observateur. Ce n'est pas "la réalité", mais une vue très subjective de la réalité.

Bien qu'un modèle ne représente pas une réalité absolue, un modèle reflète des aspects importants de la réalité, il en donne donc une vue juste et pertinente.

3.1.2. Caractéristiques fondamentales des modèles

Le caractère abstrait d'un modèle doit notamment permettre :

- de faciliter la compréhension du système étudié : un modèle réduit la complexité du système étudié.
- de simuler le système étudié : un modèle représente le système étudié et reproduit ses comportements.

3.2. Comment modéliser avec UML ?

3.2.1. Proposition de démarche

UML est un langage qui permet de représenter des modèles, mais il ne définit pas le processus d'élaboration des modèles : UML n'est donc pas une méthode de modélisation.

Cependant, dans le cadre de la modélisation d'une application informatique, les auteurs d'UML préconisent d'utiliser une démarche :

- itérative et incrémentale,
- guidée par les besoins des utilisateurs du système,
- centrée sur l'architecture logicielle.

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

3.2.1.1. Une démarche itérative et incrémentale

Pour modéliser (comprendre et représenter) un système complexe, il vaut mieux s'y prendre en plusieurs fois, en affinant son analyse par étapes.

Cette démarche doit aussi s'appliquer au cycle de développement dans son ensemble, en favorisant le prototypage.

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

3.2.1.2. Une démarche pilotée par les besoins des utilisateurs

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

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

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

- à 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.
- à chaque itération de la phase de test, on vérifie que les besoins des utilisateurs sont satisfaits.

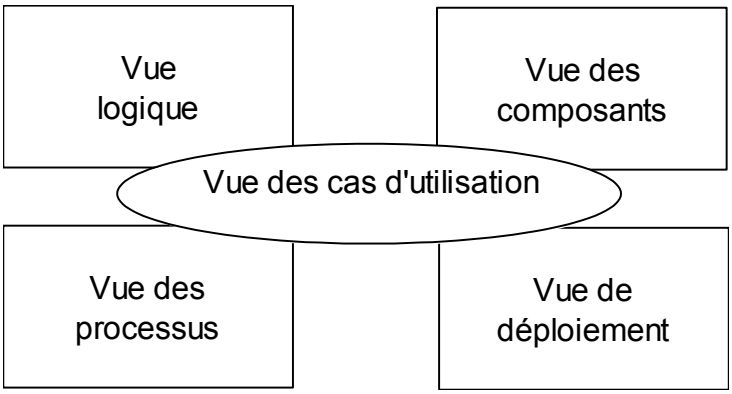
3.2.1.3. Une démarche centrée sur l'architecture

Une architecture adaptée est la clé de voûte du succès d'un développement.

Elle décrit des choix stratégiques qui déterminent en grande partie les qualités du logiciel (adaptabilité, performances, fiabilité...).

Ph. Kruchten propose différentes perspectives, indépendantes et complémentaires, qui permettent de définir un modèle d'architecture (publication IEEE, 1995). Ph. Kruchten défend l'idée que l'architecture logicielle doit être une discipline à part entière. Il propose que plusieurs perspectives concourent à l'expression de l'architecture d'un système et il explique qu'il est nécessaire de garantir la séparation et l'indépendance de ces différentes perspectives. L'évolution de l'une des perspectives ne doit pas avoir d'impact (sinon limité) sur les autres.

La relation entre les différentes perspectives a été représentée par ph. Kruchten dans le schéma suivant, dit « schéma 4+1 vues ».



3.2.2. La vue « 4+1 » de ph. Kruchten

3.2.2.1. La vue logique

Cette vue concerne « l'intégrité de conception ».

Cette vue de haut niveau se concentre sur l'abstraction et l'encapsulation, elle modélise les éléments et mécanismes principaux du système.

Elle identifie les éléments du domaine, ainsi que les relations et interactions entre ces éléments « notions de classes et de relations » :

- les éléments du domaine sont liés au(x) métier(s) de l'entreprise,
- ils sont indispensables à la mission du système,
- ils gagnent à être réutilisés (ils représentent un savoir-faire).

Cette vue organise aussi (selon des critères purement logiques), les éléments du domaine en "catégories" :

- pour répartir les tâches dans les équipes,
- regrouper ce qui peut être générique,
- isoler ce qui est propre à une version donnée, etc...

3.2.2.2. La vue des composants

Cette vue concerne « l'intégrité de gestion ».

Elle exprime la perspective physique de l'organisation du code en termes de modules, de composants et surtout des concepts du langage ou de l'environnement d'implémentation. Dans cette perspective, l'architecte est surtout concerné par les aspects de gestion du code, d'ordre de compilation, de réutilisation, d'intégration et d'autres contraintes de développement pur. Pour représenter cette perspective, UML fournit des concepts adaptés tels que les modules, les composants, les relations de dépendance, l'interface …

Cette vue de bas niveau (aussi appelée « vue de réalisation »), montre ainsi :

- l'allocation des éléments de modélisation dans des modules (fichiers sources, bibliothèques dynamiques, bases de données, exécutables, etc...). Cette vue identifie les modules qui réalisent (physiquement) les classes de la vue logique.

- l'organisation des composants, c'est-à-dire la distribution du code en gestion de configuration, les dépendances entre les composants...
- les contraintes de développement (bibliothèques externes...).
- l'organisation des modules en "sous-systèmes", les interfaces des sous-systèmes et leurs dépendances (avec d'autres sous-systèmes ou modules).

3.2.2.3. La vue des processus

Cette vue concerne « l'intégrité d'exécution ».

Cette vue est très importante dans les environnements multitâches ; elle exprime la perspective sur les activités concurrentes et parallèles. Elle montre ainsi :

- la décomposition du système en terme de processus (tâches).
- les interactions entre les processus (leur communication).
- la synchronisation et la communication des activités parallèles (threads).

3.2.2.4. La vue de déploiement

Cette vue concerne « l'intégrité de performance ». Elle exprime la répartition du système à travers un réseau de calculateurs et de nœuds logiques de traitements . Cette vue est particulièrement utile pour décrire la distribution d'un système réparti.

Elle montre :

- la disposition et nature physique des matériels, ainsi que leurs performances.
- l'implantation des modules principaux sur les nœuds du réseau.
- les exigences en terme de performances (temps de réponse, tolérance aux fautes et pannes...).

3.2.2.5. La vue des cas d'utilisation

Cette vue est particulière en ce sens qu'elle guide toutes les autres.

Cette vue permet :

- de trouver le « bon » modèle

  Les cas d'utilisation permettent de guider la modélisation. L'utilisation des scénarios et des cas d'utilisation s'avère plus rigoureuse et plus systématique que les entretiens et l'analyse des documents pour découvrir les abstractions du domaine.

- d'expliquer et de justifier ses choix

  Il est en effet nécessaire d'expliquer le système, de justifier les choix qui ont guidé sa conception et son fonctionnement pour pouvoir le construire, le maintenir et le tester. Pour cela UML offre des concepts adaptés tels que les scénarios et les cas d'utilisation.

3.2.3. Les niveaux d'abstraction

3.2.3.1. Une non-démarcation entre conception et analyse

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

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

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

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

3.2.3.2. Les niveaux d'abstraction

1. Conceptualisation

   L'entrée de l'analyse à ce niveau est le dossier d'expression des besoins client. A ce niveau d'abstraction, on doit capturer les besoins principaux des utilisateurs.

   Il ne faut pas chercher l'exhaustivité, mais clarifier, filtrer et organiser les besoins.

   Le but de la conceptualisation est :

   - de définir le contour du système à modéliser (de spécifier le "quoi"),
   - de capturer les fonctionnalités principales du système, afin d'en fournir une meilleure compréhension (le modèle produit sert d'interface entre les acteurs du projet),
   - de fournir une base à la planification du projet.

2. Analyse du domaine

   L'entrée de l'analyse à ce niveau, est le modèle des besoins clients (les "cas d'utilisation" UML).

   Il s'agit de modéliser les éléments et mécanismes principaux du système.

   On identifie les éléments du domaine, ainsi que les relations et interactions entre ces éléments :

   - les éléments du domaine sont liés au(x) métier(s) de l'entreprise,
   - ils sont indispensables à la mission du système,
   - ils gagnent à être réutilisés (ils représentent un savoir-faire).

   A ce stade, on organise aussi (selon des critères purement logiques), les éléments du domaine en "catégories", pour répartir les tâches dans les équipes, regrouper ce qui peut être générique, etc...

3. Analyse applicative

A ce niveau, on modélise les aspects informatiques du système, sans pour autant rentrer dans les détails d'implémentation.

Les interfaces des éléments de modélisation sont définis (cf. encapsulation). Les relations entre les éléments des modèles sont définies.

Les éléments de modélisation utilisés peuvent être propres à une version du système.

4. Conception

   On y modélise tous les rouages d'implémentation et on détaille tous les éléments de modélisation issus des niveaux supérieurs.

   Les modèles sont optimisés, car destinés à être implémentés.

3.3. L'utilisation de diagrammes

UML permet de définir et de visualiser un modèle, à l'aide de diagrammes.

3.3.1. Définition d'un diagramme

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).

3.3.2. 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").

3.3.3. Les différents types de diagrammes UML

Il existe 2 types de vues du système qui comportent chacune leurs propres diagrammes :

- les vues statiques :
  - diagrammes de cas d'utilisation
  - diagrammes d'objets

- diagrammes de classes
- diagrammes de composants
- diagrammes de déploiement

- les vues dynamiques :
  - diagrammes de collaboration
  - diagrammes de séquence
  - diagrammes d'états-transitions
  - diagrammes d'activités

4. Les Différents types de diagrammes

4.1. Vues statiques du système

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.

4.1.1. Diagrammes de cas d'utilisation

4.1.1.1. Définition du cas d'utilisation (use case)

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.

La détermination et la compréhension des besoins sont souvent difficiles car les intervenants sont noyés sous de trop grandes quantités d'informations : il faut clarifier et organiser les besoins des clients (les modéliser). Pour cela, les cas d'utilisation identifient les utilisateurs du système (acteurs) et leurs interactions avec le système. Ils permettent de classer les acteurs et structurer les objectifs du système.

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.

4.1.1.2. Éléments de modélisation des cas d'utilisation

1. L'acteur :

   La première étape de modélisation consiste à définir le périmètre du système, à définir le contour de l'organisation et à le modéliser. Toute entité qui est en dehors de cette organisation et qui interagit avec elle est appelé acteur selon UML.

Un acteur est un type stéréotypé représentant une abstraction qui réside juste en dehors du système à modéliser.

Un acteur représente un rôle joué par une personne ou une chose qui interagit avec le système. (la même personne physique peut donc être représentée par plusieurs acteurs en fonction des rôles qu'elle joue).

Pour identifier les acteurs, il faut donc se concentrer sur les rôles joués par les entités extérieures au périmètre.

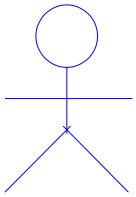
Dans UML, il n'y a pas de notion d'acteur interne et externe. Par définition, un acteur est externe au périmètre de l'étude, qu'il appartienne ou pas à la société.

Enfin, un acteur n'est pas nécessairement une personne physique : il peut être un service, une société, un système informatique …

Il existe 4 catégories d'acteurs :

- les acteurs principaux : les personnes qui utilisent les fonctions principales du système
- les acteurs secondaires : les personnes qui effectuent des tâches administratives ou de maintenance.
- le matériel externe : les dispositifs matériels incontournables qui font partie du domaine de l'application et qui doivent être utilisés.
- les autres systèmes : les systèmes avec lesquels le système doit interagir.

Formalisme :



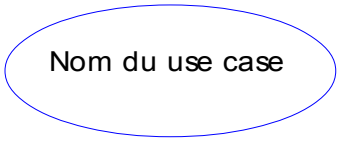
Nom acteur

2. Le cas d'utilisation :

   Le cas d'utilisation (ou use case) correspond à un objectif du système, motivé par un besoin d'un ou plusieurs acteurs.

   L'ensemble des use cases décrit les objectifs (le but) du système.

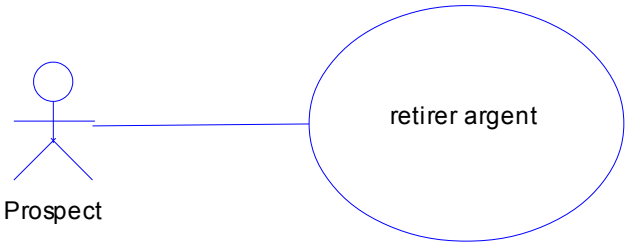
   Formalisme :



3. La relation :

   Elle exprime l'interaction existant entre un acteur et un cas d'utilisation.

   Formalisme :



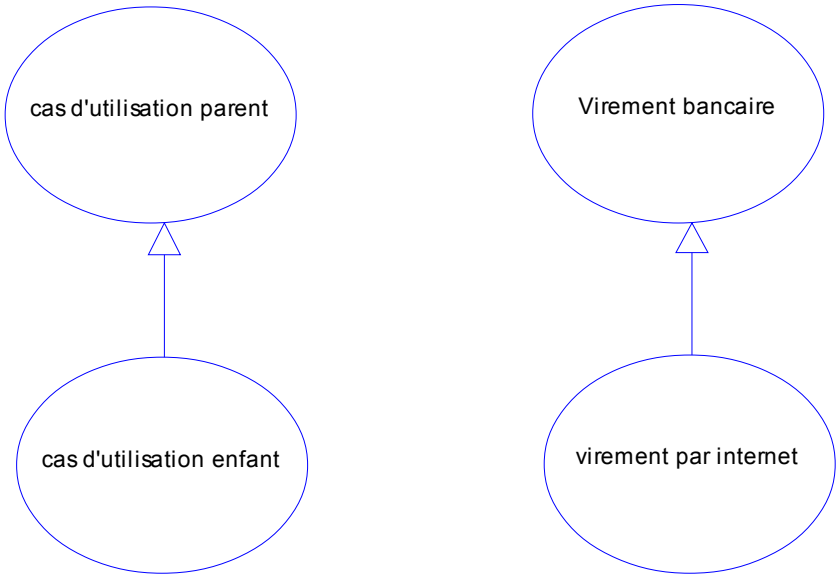
Il existe 3 types de relations entre cas d'utilisation :

- la relation de généralisation
- la relation d'extension
- la relation d'inclusion

4. La relation de généralisation :

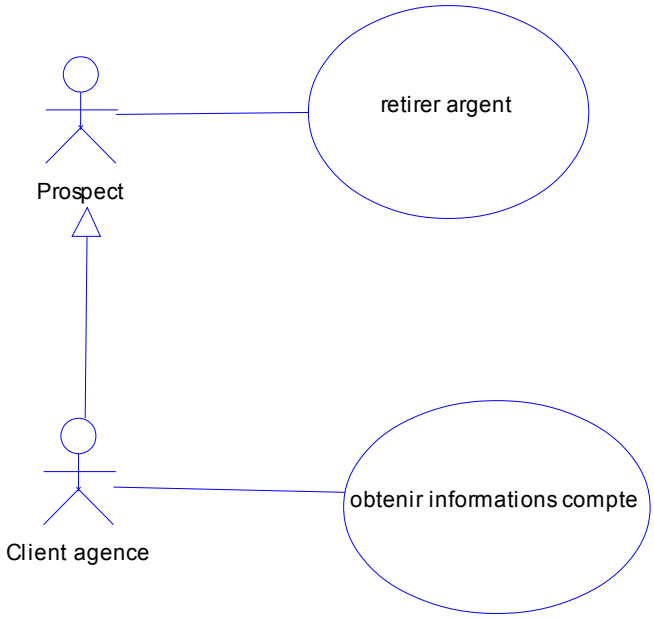
   Dans une relation de généralisation entre 2 cas d'utilisation, le cas d'utilisation enfant est une spécialisation du cas d'utilisation parent.

   Formalisme et exemple :



   NB : un acteur peut également participer à des relations de généralisation avec d'autres acteurs. Les acteurs « enfant » seront alors capables de communiquer avec les mêmes cas d'utilisation que les acteurs « parents ».

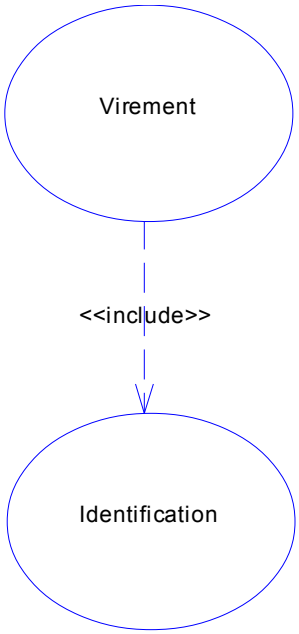
   Formalisme et exemple :



5. La relation d'inclusion :

   Elle indique que le cas d'utilisation source contient aussi le comportement décrit dans le cas d'utilisation destination. L'inclusion a un caractère obligatoire, la source spécifiant à quel endroit le cas d'utilisation cible doit être inclus. Cette relation permet ainsi de décomposer des comportements et de définir des comportements partageables entre plusieurs cas d'utilisation.

   Formalisme :



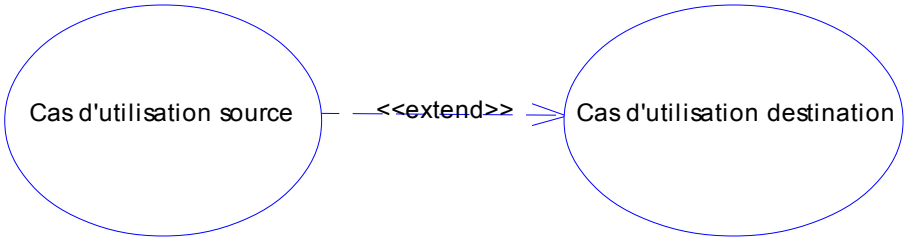
   Pour réaliser l'objectif « virement », on utilise obligatoirement « identification ».

6. La relation d'extension :

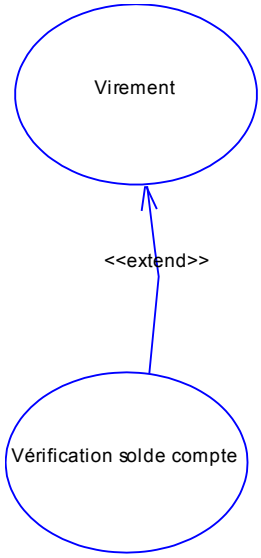
   Elle indique que le cas d'utilisation source ajoute son comportement au cas d'utilisation destination. L'extension peut être soumise à condition. Le comportement ajouté est inséré au

niveau d'un point d'extension défini dans le cas d'utilisation destination. Cette relation permet de modéliser les variantes de comportement d'un cas d'utilisation (selon les interactions des acteurs et l'environnement du système).

Formalisme :



Exemple :

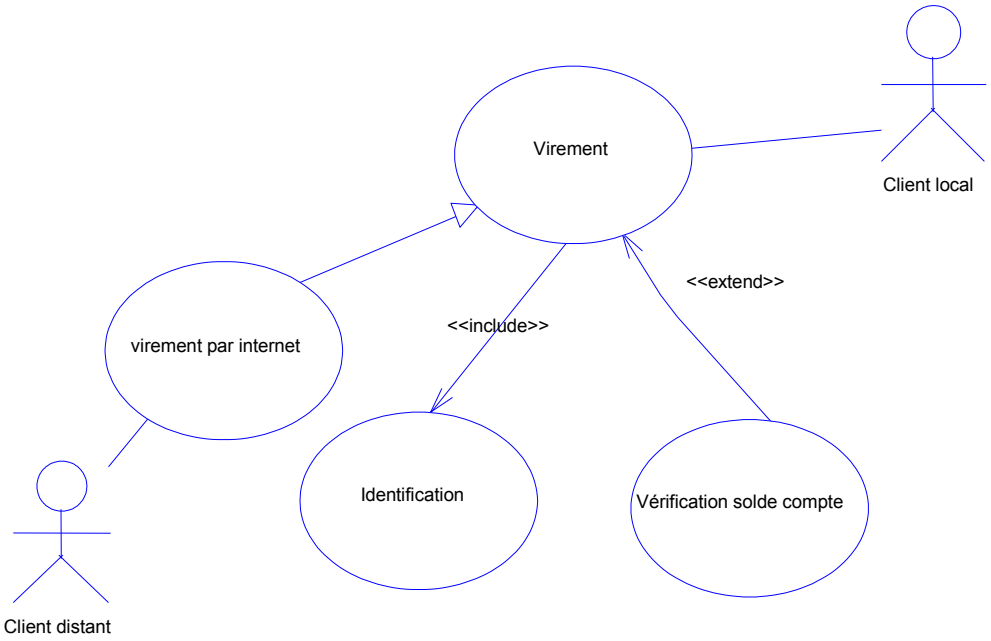


7. Paquetage :

   Un paquetage (package) est un groupement d'élément de modélisation. Un paquetage peut contenir aussi bien des paquetages emboîtés que des éléments de modélisation ordinaires. Le système entier peut être pensé comme un unique paquetage de haut niveau comprenant l'ensemble. Tous les éléments de modélisation d'UML, y compris les diagrammes, peuvent être organisés en paquetage.

   Les uses cases peuvent être organisés en paquetages (packages).

   Exemple de cas d'utilisation :



8. Les scénarios :

   Un cas d'utilisation est une abstraction de plusieurs chemins d'exécution. Une instance de cas d'utilisation est appelée : « scénario ».

   Chaque fois qu'une instance d'un acteur déclenche un cas d'utilisation, un scénario est créé (le cas d'utilisation est instancié). Ce scénario suivra un chemin particulier dans le cas d'utilisation.

   Un scénario ne contient pas de branche du type « Si condition … alors » car pendant l'exécution, la condition est soit vraie, soit fausse, mais elle aura une valeur.

   Après la description des cas d'utilisation, il est nécessaire de sélectionner un ensemble de scénarios qui vont servir à piloter l'itération en cours de développement.

   Le choix et le nombre de scénarios à retenir est une étape difficile à réaliser : l'exhaustivité est difficile, voire impossible à atteindre. Le nombre d'instances pour un cas d'utilisation peut être très important, voire infini.

   Les scénarios peuvent être classés en :

   - scénarios principaux : il correspond à l'instance principal du cas d'utilisation. C'est souvent le chemin « normal » d'exécution du cas d'utilisation qui n'implique pas d'erreurs.
   - Scénarios secondaires : il peut être un cas alternatif (un choix), un cas exceptionnel ou une erreur.

   Les scénarios sont utiles pour :

   - analyser et concevoir le système
   - justifier les choix effectués (ils serviront à la documentation des cas d'utilisation)
   - tester : les scénarios constituent le meilleur moyen de spécifier les tests.

4.1.1.3. Élaboration des cas d'utilisation

Un cas d'utilisation doit être avant tout simple, intelligible et décrit de manière claire et concise. Le nombre d'acteurs qui interagissent avec le cas d'utilisation est souvent limité. Il y a souvent un acteur pas cas d'utilisation.

Lors de l'élaboration d'un cas d'utilisation, il faut se demander :

- quelles sont les tâches de l'acteur ?
- quelles informations l'acteur doit-il créer, sauvegarder, modifier ou lire ?
- l'acteur devra-t-il informer le système des changements externes ?
- le système devra-t-il informer l'acteur des conditions internes ?
- quelles sont les conditions de démarrage et d'arrêt du cas d'utilisation ?

Les cas d'utilisation peuvent être présentés à travers de vues multiples : un acteur avec tous ses cas d'utilisation, un cas d'utilisation avec tous ses acteurs …

Un cas d'utilisation est une abstraction : il décrit de manière abstraite une famille de scénarios. Il ne faut donc pas réaliser trop de cas d'utilisation car cela constituerait un manque d'abstraction. Dans n'importe quels système, il y a relativement peu de cas d'utilisation mais beaucoup de scénarios. Un grand nombre de cas d'utilisation signifie par conséquent que l'essence du système n'est pas comprise.

4.1.1.4. Utilisation des cas d'utilisation

La portée des cas d'utilisation dépasse largement la définition des besoins du système. Les cas d'utilisation interviennent tout au long du cycle de vie du projet, depuis le cahier des charges jusqu'aux tests.

Intervenant	Utilisateur	Analyste	Architecte	Programmeur	Testeur
Rôle des cas d'utilisation	Exprimer	Comprendre	Concevoir	réaliser	vérifier

4.2.2. Diagrammes de classes

4.2.2.1. Définition du diagramme de classes

Le diagramme de classes exprime la structure statique du système en termes de classes et de relations entre ces classes.

L'intérêt du diagramme de classe est de modéliser les entités du système d'information.

Le diagramme de classe permet de représenter l'ensemble des informations finalisées qui sont gérées par le domaine. Ces informations sont structurées, c'est-à-dire qu'elles ont regroupées dans des classes. Le diagramme met en évidence d'éventuelles relations entre ces classes.

Le diagramme de classes comporte 6 concepts :

- classe
- attribut
- identifiant

- relation
- opération
- généralisation / spécialisation

4.2.2.2. Les notions utilisées par le diagramme de classes

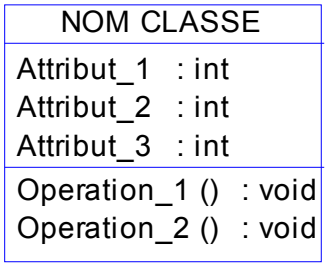
1. La notion de classe

   Définition : une classe est une description abstraite (condensée) d'un ensemble d'objets du domaine de l'application : elle définit leur structure, leur comportement et leurs relations.

   Représentation : les classes sont représentées par des rectangles compartimentés :

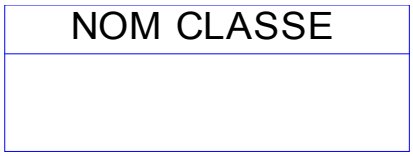
   - le 1er compartiment représente le nom de la classe
   - le 2ème compartiment représente les attributs de la classe
   - le 3ème compartiment représente les opérations de la classe

   Formalisme :

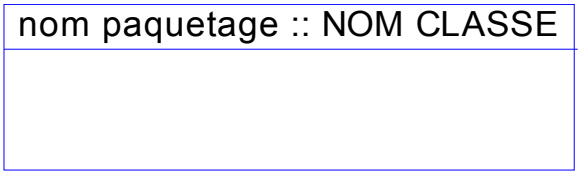


   Les compartiments d'une classe peuvent être supprimés (en totalité ou en partie) lorsque leur contenu n'est pas pertinent dans le contexte d'un diagramme. La suppression des compartiments reste purement visuelle : elle ne signifie pas qu'il n'y a pas d'attribut ou d'opération.

   Formalisme de représentation simplifiée :



   Formalisme de représentation avec chemin complet :



   Le rectangle qui symbolise une classe peut contenir un stéréotype et des propriétés.

   UML définit les stéréotypes de classe suivants :

   - « classe implémentation » : il s'agit de l'implémentation d'une classe dans un langage de programmation.

- « énumération » : il s'agit d'une classe qui définit un ensemble d'identificateurs formant le domaine de la valeur.
- « métaclasse » : il s'agit de la classe d'une classe, comme en Smalltalk.
- « powertype » : une classe est un métatype : ses instances sont toutes des sous-types d'un type donné.
- « processus » : il s'agit d'une classe active qui représente un flot de contrôles lourd.
- « thread » : il s'agit d'une classe active qui représente un flot de contrôles léger.
- « type » : il s'agit d'une classe qui définit un domaine d'objets et les opérations applicables à ces objets.
- « utilitaire » : il s'agit d'une classe réduite au concept de module et qui ne peut être instanciée.

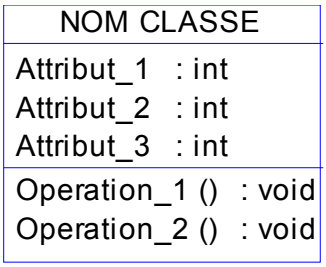
2. La notion d'attribut

Définition : Une classe correspond à un concept global d'information et se compose d'un ensemble d'informations élémentaires, appelées attributs de classe.

Un attribut représente la modélisation d'une information élémentaire représentée par son nom et son format.

Par commodité de gestion, on choisit parfois de conserver dans un attribut le résultat d'un calcul effectué à partir d'autres classes : on parle alors d'attribut dérivé. Pour repérer un attribut dérivé : on place un / devant son nom.

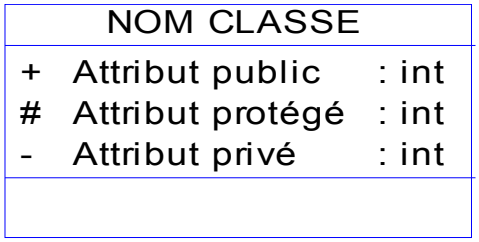
Formalisme :



UML définit 3 niveaux de visibilité pour les attributs :

- public (+) : l'élément est visible pour tous les clients de la classe
- protégé (#) : l'élément est visible pour les sous-classes de la classe
- privé (-) : l'élément n'est visible que par les objets de la classe dans laquelle il est déclaré.

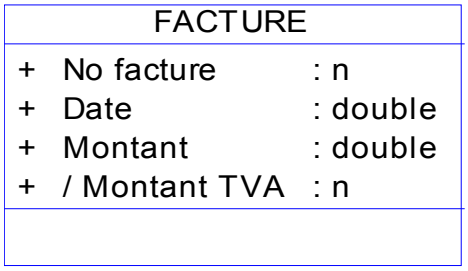
Formalisme :



3. La notion d'identifiant

   L'identifiant est un attribut particulier, qui permet de repérer de façon unique chaque objet, instance de la classe.

   Formalisme :

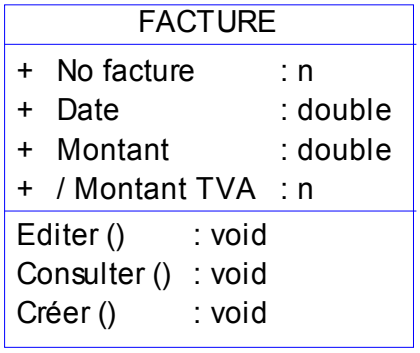


4. La notion d'opération

   Définition : l'opération représente un élément de comportement des objets, défini de manière globale dans la classe.

   Une opération est une fonctionnalité assurée par une classe. La description des opérations peut préciser les paramètres d'entrée et de sortie ainsi que les actions élémentaires à exécuter.

   Formalisme :



   Comme pour les attributs, on retrouve 3 niveaux de visibilité pour les opérations :

   - public (+) : l'opération est visible pour tous les clients de la classe
   - protégé (#) : l'opération est visible pour les sous-classes de la classe
   - privé (-) : l'opération n'est visible que par les objets de la classe dans laquelle elle est déclarée.

   Formalisme :

FACTURE	
+ No facture	: int
+ Date	: Date
+ Montant	: double
+ / Montant TVA	: double
+ Op publique ()	
# Op protégée ()	
- Op privée ()	

5. La notion de relation

   S'il existe des liens entre objets, cela se traduit nécessairement par des relations qui existent entre leurs classes respectives.

   Les liens entre les objets doivent être considérés comme des instances de relations entre classes.

   Il existe plusieurs types de relations entre classes :

   - l'association
   - la généralisation/spécialisation
   - la dépendance

4.2.2.3. L'association

L'association est la relation la plus courante et la plus riche du point de vue sémantique. Une association est une relation statique n-aire (le plus souvent : elle est binaire) : c'est-à-dire qu'elle relie plusieurs classes entre elles.

L'association existe entre les classes et non entre les instances : elle est introduite pour montrer une structure et non pour montrer des échanges de données.

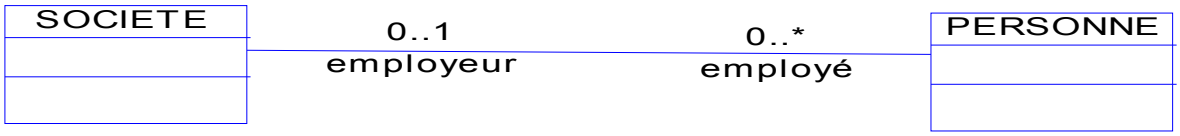
Une association n-aire possède n rôles qui sont les points terminaux de l'association ou terminaisons.

Chaque classe qui participe à l'association joue un rôle. Les rôles sont définis par 2 propriétés :

- la multiplicité : elle définit le nombre d'instances de l'association pour une instance de la classe. La multiplicité est définie par un nombre entier ou un intervalle de valeurs. La multiplicité est notée sur le rôle (elle est notée à l'envers de la notation MERISE).

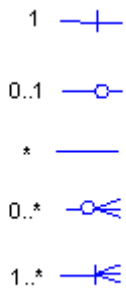
1	Un et un seul
0..1	Zéro ou un
N ou *	N (entier naturel)
M..N	De M à N (entiers naturels)
0..*	De zéros à plusieurs
1..*	De 1 à plusieurs

Formalisme et exemple en employant les noms des rôles et leur multiplicité :



Formalisme et exemple en employant le noms de l'association et la multiplicité des rôles :

La multiplicité peut également s'exprimer par des symboles :



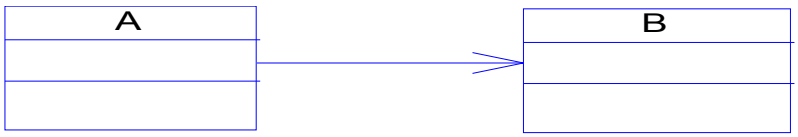
Les valeurs de multiplicité expriment les contraintes liées au domaine de l'application. Il est donc important de déterminer les valeurs de multiplicité optimales pour trouver le bon équilibre entre complexité et efficacité. La surestimation des valeurs de multiplicité entraîne un surcoût de taille de stockage et en vitesse d'exécution (requête avec plus de jointures).

- la navigabilité

  La navigabilité n'a rien à voir avec le sens de lecture de l'association. Une navigabilité placée sur une terminaison cible indique si ce rôle est accessible à partir de la source.

  Par défaut les associations sont navigables dans les 2 sens. Dans certains cas, une seule direction de navigation est utile : l'extrémité d'association vers laquelle la navigation est possible porte alors une flèche.

  Formalisme :



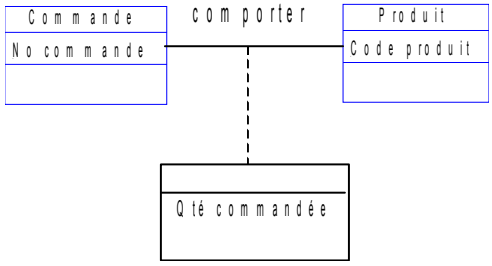
  Dans l'exemple ci-dessus, les instances de A voient les instances de B mais les instances de B ne voient pas les instances de A.

Les classes-association

Les attributs d'une classe dépendent fonctionnellement de l'identifiant de la classe. Parfois, un attribut dépend fonctionnellement de 2 identifiants, appartenant à 2 classes différentes. Par exemple, l'attribut « quantité commandée » dépend fonctionnellement du numéro de commande et du code produit. On va donc placer l'attribut « quantité commandée » dans l'association « comporter ».

Dans ce cas, l'association est dite « porteuse d'attributs ».

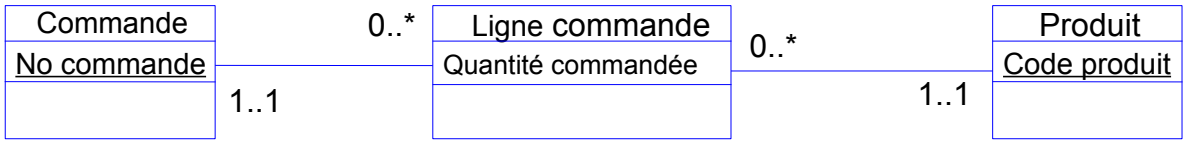
Une association porteuse d'attributs est appelée classe-association.



Exemple :

Toute classe-association peut être remplacée par une classe intermédiaire et qui sert de pivot pour une paire d'association.

Exemple :



De même, toute association (qui ne contient pas d'attributs) impliquant 3 classes ou plus peut donner lieu à une décomposition avec une classe intermédiaire.

L'agrégation

Dans UML, l'agrégation n'est pas un type de relation mais une variante de l'association.

Une agrégation représente une association non symétrique dans laquelle une des extrémités joue un rôle prédominant par rapport à l'autre extrémité.

L'agrégation ne peut concerner qu'un seul rôle d'une association.

L'agrégation se représente toujours avec un petit losange du côté de l'agrégat.

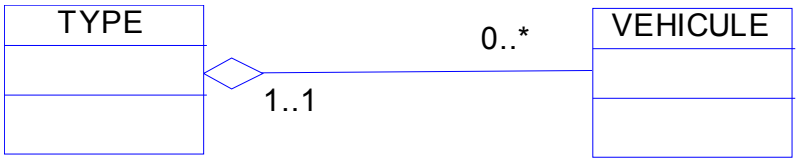
Le choix d'une association de type agrégation traduit la volonté de renforcer la dépendance entre classes. C'est donc un type d'association qui exprime un couplage plus fort entre les classes. L'agrégation permet de modéliser des relations de type maître et esclaves.

L'agrégation permet de modéliser une contrainte d'intégrité et de désigner l'agrégat comme contrainte.

A travers une telle contrainte, il est possible de représenter par exemple :

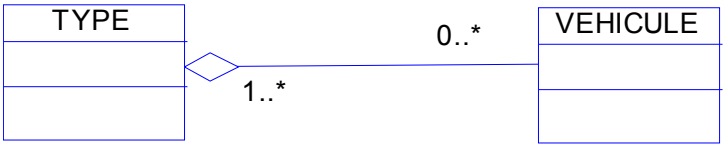
- la propagation des valeurs d'attributs d'une classe vers une autre classe
- une action sur une classe qui implique une action sur une autre classe
- une subordination des objets d'une classe à une autre classe

Formalisme et exemple :



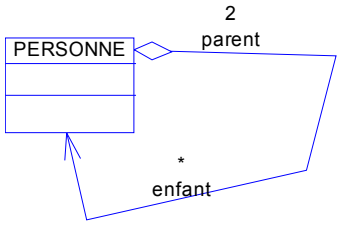
L'exemple ci-dessus montre que l'on veut gérer une classification de véhicules. Chaque véhicule est classifié selon son type. En conséquence, il sera possible de prendre connaissance pour un véhicule de l'ensemble des caractéristiques du type de véhicule auquel il est associé.

NB : un agrégat peut être multiple. Dans l'exemple ci-dessous, un véhicule peut appartenir à plusieurs types.



Cas particulier des associations réflexives :

On peut avoir des cas d'agrégation réflexive dès que l'on modélise des relations hiérarchiques ou des liens de parenté par exemple.

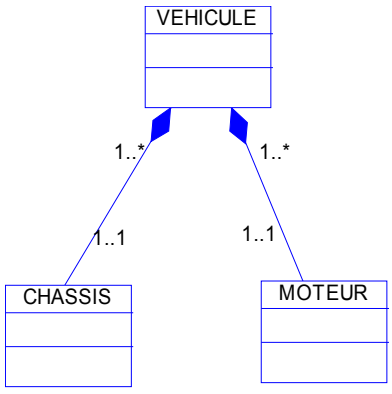


La composition

La composition est un cas particulier de l'agrégation dans laquelle la vie des composants est liée à celle des agrégats. Elle fait souvent référence à une contenance physique. Dans la composition l'agrégat ne peut être multiple.

La composition implique, en plus de l'agrégation, une coïncidence des durées de vie des composants : la destruction de l'agrégat (ou conteneur) implique automatiquement la destruction de tous les composants liés.

Formalisme et exemple :



4.2.2.4. La généralisation / spécialisation

Le principe de généralisation / spécialisation permet d'identifier parmi les objets d'une classe (générique) des sous-ensembles d'objets (des classes spécialisées) ayant des définitions spécifiques. La classe plus spécifique (appelée aussi classe fille, classe dérivée, classe spécialisée, classe descendante …) est cohérente avec la classe plus générale (appelée aussi classe mère, classe

générale …), c'est-à-dire qu'elle contient par **héritage** tous les attributs, les membres, les relations de la classe générale, et peut contenir d'autres.

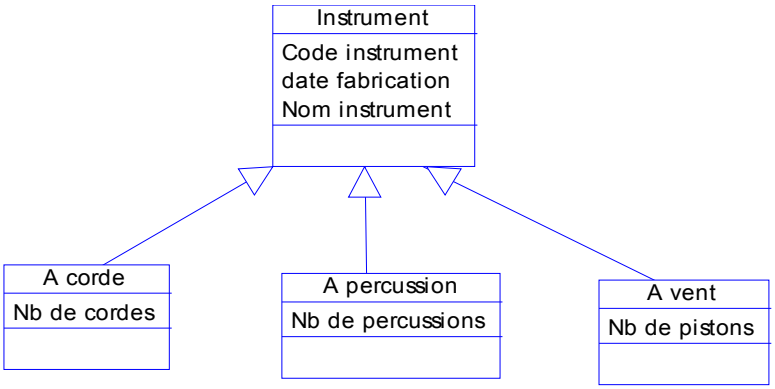
Les relations de généralisation peuvent être découvertes de 2 manières :

- la généralisation : il s'agit de prendre des classes existantes déjà mises en évidences) et de créer de nouvelles classes qui regroupent leurs parties communes ; il faut aller du plus spécifique au plus général.
- La spécialisation : il s'agit de sélectionner des classes existantes (déjà identifiées) et d'en dériver des nouvelles classes plus spécialisées, en spécifiant simplement les différences.

Ces 2 démarches, même si elles sont fondamentalement différentes, mènent au même résultat, à savoir la constitution d'une hiérarchie de classes reliées par des relations de généralisation.

La relation de généralisation est très puissante car elle permet de construire simplement de nouvelles classes à partir de classes existantes. Cependant, elle est très contraignante dans le sens où elle définit une relation très forte entre les classes. Ainsi, l'évolution d'une classe de base entraîne une évolution de toutes les classes qui en dérivent. Cet effet boule de neige peut avoir des conséquences aussi bien positives (quand c'est l'effet recherché) que négatives.

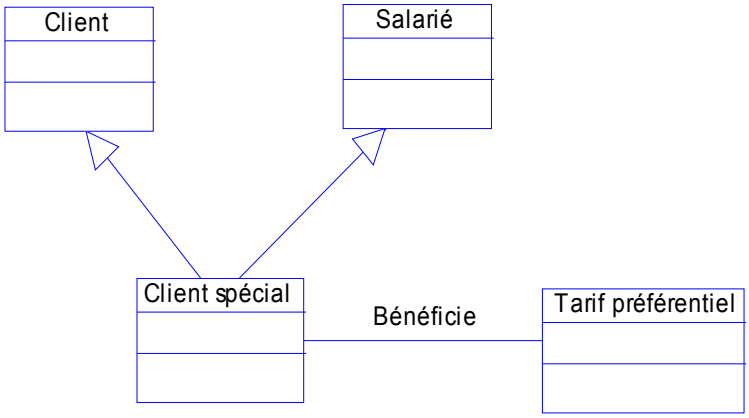
Formalisme et exemple :



La spécialisation multiple

Les classes peuvent avoir plusieurs superclasses ; dans ce cas, la généralisation est dite multiple et plusieurs flèches partent de la sous-classe vers les différentes superclasses. La généralisation multiple consiste à fusionner plusieurs classes en une seule classe.

Formalisme et exemple :



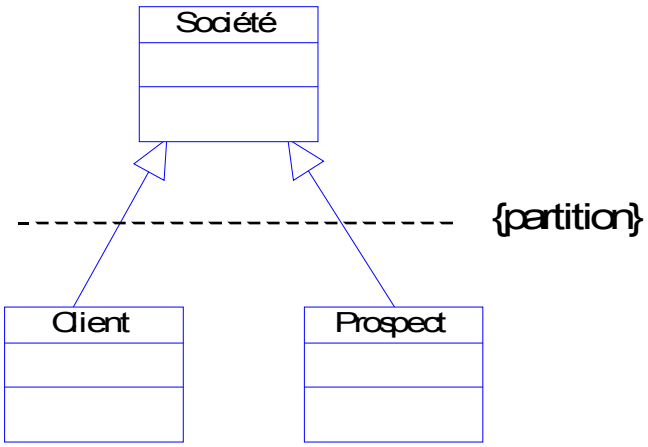
La classe « client spécial » est une spécialisation de client et de salarié. Ce modèle permet d'indiquer que l'on accorde des tarifs préférentiels aux salariés.

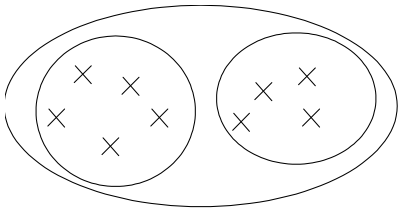
Les contraintes sur les associations

Il existe plusieurs types de contraintes sur les associations :

- La contrainte de partition

  Elle indique que toutes les instances d'une classe correspondent à une et une seule instance des classes liées.

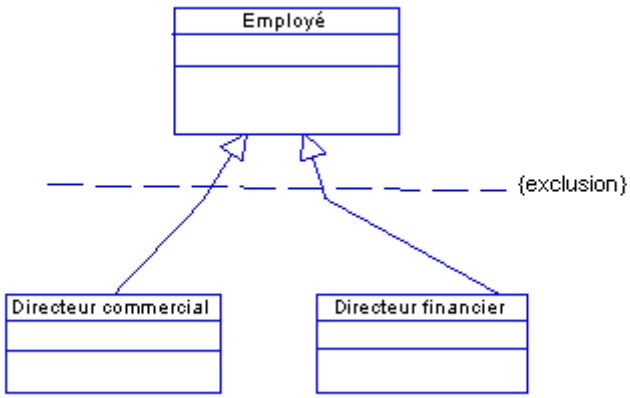


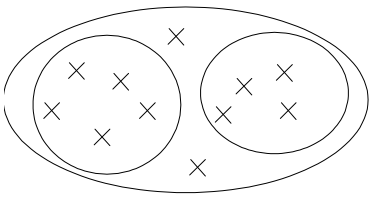


  Toutes les sociétés sont soit clientes, soit considérées comme des prospects.

- la contrainte d'exclusion

  Elle permet de préciser qu'une instance d'association exclut une autre instance. Par exemple, un employé ne peut être à la fois directeur financier et directeur commercial.

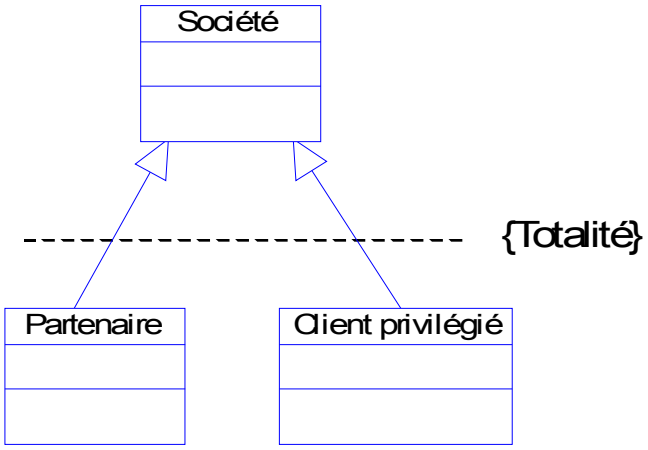


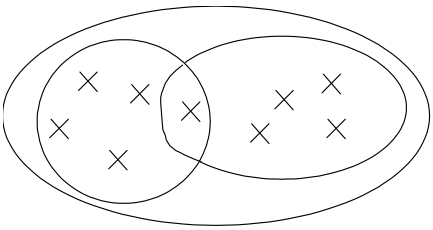


  Ici, un employé ne peut pas être à la fois directeur commercial et directeur financier. Mais tout employé n'est pas directeur commercial ou directeur financier (contrainte de partition).

- la contrainte de totalité

  Toutes les instances d'une classe correspondent au moins à une des instances des classes liées.





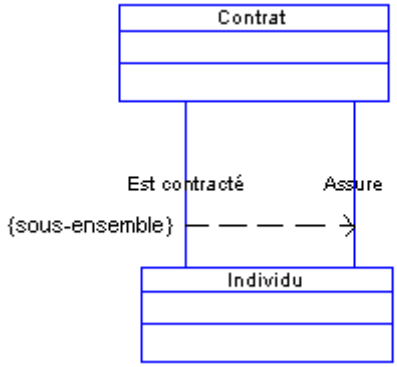
Toute société est au moins partenaire ou client privilégiée. Et elle peut être les 2 à la fois.

- la contrainte d'inclusion

  Elle permet de préciser qu'une collection est incluse dans une autre collection. (la flèche de la relation de dépendance indique le sens de la contrainte).

  Par exemple, on pourra indiquer que le contractant d'un contrat fait obligatoirement partie des individus assurés.

  Exemple :



4.2.2.5. La dépendance

Les relations de dépendance sont utilisées lorsqu'il existe une relation sémantique entre plusieurs éléments qui n'est pas de nature structurelle. Une relation de dépendance définit une relation unidirectionnelle entre un élément source et un élément cible.

Une dépendance est une relation entre deux éléments de modélisation dans laquelle toute modification effectuée sur un élément de modélisation (l'élément influent) affecte l'autre élément (élément dépendant).

UML définit 4 types de relation de dépendance. Pour chaque type de dépendance, un mot clé ou stéréotype entre guillemets peut être ajouté à la relation de dépendance pour préciser sa nature.

Les 4 types de relation sont :

- abstraction

Il s'agit d'une relation de dépendance entre éléments qui représentent un même concept à différents niveaux d'abstraction ou selon des points de vue distincts.

Les mots-clés sont :

- « dérive »

Représente un élément défini ou calculé à partir d'un autre. Par exemple, un attribut ou un rôle peut dériver d'autres attributs ou rôles.

- « raffine »

Représente une relation de dépendance entre des éléments sémantiques différents (analyse et conception par exemple).

- « réalise »

Représente une relation de dépendance entre une spécification (cible) et l'élément qui implémente cette spécification (source)

- « trace »

Représente l'historique des constructions présentes dans les différents modèles.

- Liaison

  Les paramètres formels d'une classe ou collaboration paramétrables doivent être liés à des valeurs. Cette dépendance crée une liaison entre la classe ou collaboration paramétrable (la cible) et la classe ou collaboration paramétrée (source).

  - « lie »

- Permission

  Un élément (source) a le droit d'accéder à un espace de nommage (cible)

  - « ami »

  Représente un élément source (paquetage, classe, opération …) qui a accès à l'élément destination (paquetage, classe, opération …) quelle que soit la spécification de visibilité de ce dernier.

- Utilisation

  Un élément (source) requiert la présence d'un autre élément (cible) pour son bon fonctionnement ou son implémentation.

  - « utilise »
  - « « appelle » »

  Représente une relation de dépendance entre une opération qui invoque une opération d'une autre classe. Cette relation est représentée en connectant les opérations ou les classes qui possèdent ces opérations.

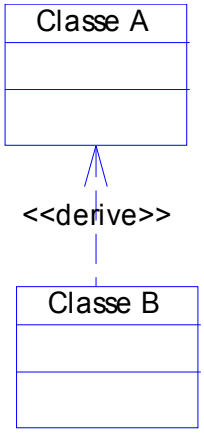
  - « crée »

  Représente le classificateur source qui crée une instance du classificateur cible

  - « instancie »

Représente une relation de dépendance entre classificateurs due à la création d'une instance du classificateur cible par une opération du classificateur source.

Formalisme :



4.2.2.6. L'interface

Une interface définit le comportement visible d'une classe. Ce comportement est défini par une liste d'opérations ayant une visibilité « public ». Aucun attribut ou association n'est défini pour une interface.

Une interface est en fait une classe particulière (avec le stéréotype « « interface » »).

UML représente les interfaces :

- soit au moyen de petits cercles reliés par un trait à l'élément qui fournit les services décrits par l'interface
- soit au moyen de classes avec le mot clé « « interface » ». Cette notation permet de faire figurer dans le compartiment des opérations la liste des services de l'interface.

Les relations possibles sur une interface sont :

- la fourniture

  Cette relation spécifie qu'une classe donnée fournit l'interface à ses clients : c'est-à-dire que la classe possède cette interface.

  Une classe peut fournir plusieurs interfaces à ses clients et chaque interface définit un des services de la classe. Cette technique permet de réduire la visibilité d'une classe.

  En effet, une classe qui expose ses opérations publiques les expose à toutes les autres classes du modèle. Le concept d'interface permet à une classe de définir plusieurs profils en permettant à chaque classe de n'utiliser que le profil qui l'intéresse. Une classe peut ainsi être vue avec plusieurs perspectives différentes en fonction de la classe qui l'utilise, ce qui augmente la réutilisabilité.

- l'utilisation

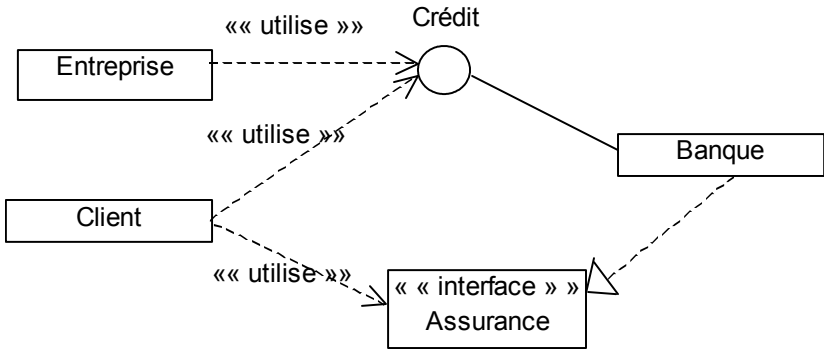
  Cette relation concerne toute classe client qui souhaite accéder à la classe interface de manière à accéder à ses opérations. C'est une relation d'utilisation standard.

- la réalisation

Cette relation n'est utilisée que pour les interfaces.

Une réalisation est une relation entre une classe et une interface. Elle montre que la classe réalise les opérations offertes par l'interface.

Exemple et formalisme :



L'exemple ci-dessus illustre la modélisation de 2 interfaces crédit et assurance d'une classe banque. Une relation de réalisation indique que la classe banque réalise l'interface assurance.

4.2.2.7. Élaboration d'un diagramme de classes

1. Généralités

   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.

2. Règles d'élaboration

   Les règles de normalisation des modèles entité-relation, issues de l'algèbre relationnelle, peuvent être utilement appliquées à un modèle de classes UML, même si UML ne contient aucune préconisation sur ces règles.

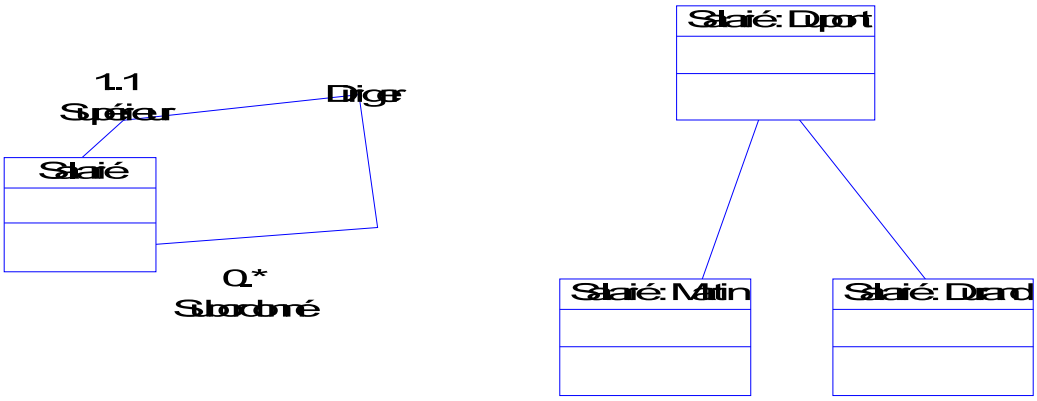
   Ces règles aident à conduire les travaux de modélisation en évitant le plus possible la redondance de l'information, tout en restant fidèle aux règles de gestion.

4.2.3. Diagrammes d'objets

Le diagramme d'objets permet de mettre en évidence des liens entre les objets. Les objets, instances de classes, sont reliés par des liens, instances d'associations.

A l'exception de la multiplicité, qui est explicitement indiquée, le diagramme d'objets utilise les mêmes concepts que le diagramme de classes. Ils sont essentiellement utilisés pour comprendre ou illustrer des parties complexes d'un diagramme de classes.

Exemple :



4.2.4. Diagrammes de composants

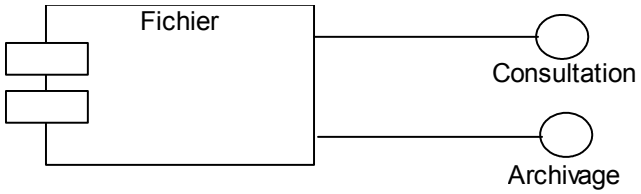
Les diagrammes de composants décrivent les composants et leurs dépendances dans l'environnement de réalisation.

En général, ils ne sont utilisés que pour des systèmes complexes.

Un composant est une vue physique qui représente une partie implémentable d'un système.

Un composant peut être du code, un script, un fichier de commandes, un fichier de données, une table … Il peut réaliser un ensemble d'interfaces qui définissent alors le comportement offert à d'autres composants.

Formalisme :



UML définit 5 stéréotypes aux composants :

- « « document » » : un document quelconque
- « « exécutable » » : un programme qui peut s'exécuter
- « « fichier » » : un document contenant un code source ou des données
- « « bibliothèque » » : une bibliothèque statique ou dynamique
- « « table » » : une table de base de données relationnelle

Pour montrer les instances des composants, un diagramme de déploiement peut être utilisé.

4.2.5. Diagrammes de déploiement

Les diagrammes de déploiement montrent la disposition physique des différents matériels (les nœuds) qui entrent dans la composition d'un système et la répartition des instances de composants, processus et objets qui « vivent » sur ces matériels.

Les diagrammes de déploiement sont donc très utiles pour modéliser l'architecture physique d'un système.

4.3. Vues dynamiques du système

Les cas d'utilisation sont centrés sur les besoins des utilisateurs. Ils aident à construire le bon système. Les cas d'utilisation ne fournissent pas pour autant la bonne manière de faire le système., ni la forme de l'architecture logicielle : ils disent ce qu'un système doit faire, non comment il doit le faire.

La vue des cas d'utilisation est une description fonctionnelle des besoins, structurée par rapport à des acteurs. Le passage à l'approche objet s'effectue en associant une collaboration à chaque cas d'utilisation.

Une collaboration décrit les objets du domaine, les connexions entre ces objets et les messages échangés par les objets.

Chaque scénario, instance du cas d'utilisation réalisé par la collaboration se représente par une interaction entre les objets décrits dans le contexte de la collaboration.

4.3.1. diagrammes de collaboration

4.3.1.1. Objectifs du diagramme de collaboration

Le diagramme de collaboration permet de mettre en évidence les interactions entre les différents objets du système.

Dans le cadre de l'analyse, il sera utilisé

- pour préciser le contexte dans lequel chaque objet évolue
- pour mettre en évidence les dépendances entre les différents objets impliqués dans l'exécution d'un processus ou d'un cas d'utilisation.

Un diagramme de collaboration fait apparaître les interactions entre des objets et les messages qu'ils échangent.

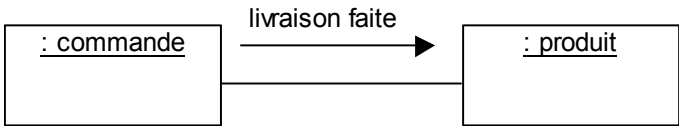
4.3.1.2. Les interactions

Une interaction définit la communication entre les objets sous la forme d'un ensemble partiellement ordonné de messages.

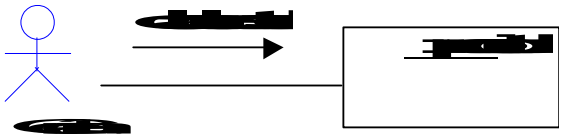
L'objet émetteur envoie un message à l'objet récepteur. Les objets représentés dans les diagrammes de collaboration ne sont pas nécessairement des instances d'entités. Certains messages peuvent avoir pour origine des acteurs que l'on pourra représenter.

Formalisme : l'interaction se représente par une flèche avec un texte décrivant le message.

Exemple d'interaction entre objets :



Exemple d'interaction entre un objet et un acteur ;

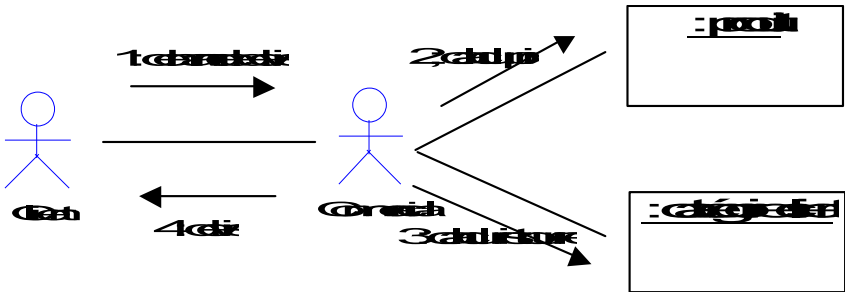


4.3.1.3. Les messages

Les messages sont le seul moyen de communication entre les objets. Ils sont décrits essentiellement par l'objet émetteur et l'objet récepteur. Leur description peut être complétée par un nom, une séquence, des arguments, un résultat attendu, une synchronisation, une condition d'émission.

La séquence permet de préciser l'ordre d'émission des messages.

Formalisme et exemple :

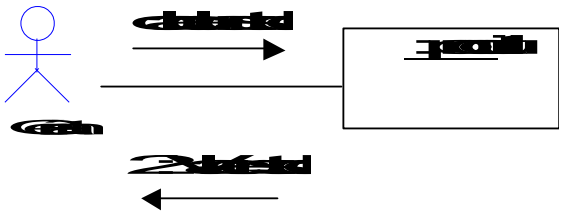


Le message 1 peut avoir comme arguments l'intitulé du produit souhaité, la quantité et la catégorie du client.

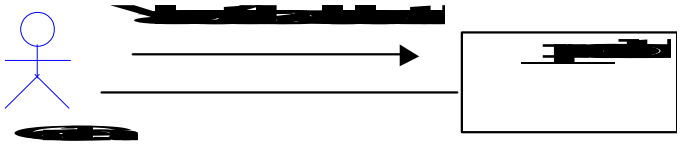
Certains messages peuvent solliciter un résultat. Ce cas peut être modéliser de 2 façons :

- un message de demande et un message de réponse
- indiquer sur le premier message le résultat attendu (lorsque le message en retour est attendu immédiatement).

Exemple :

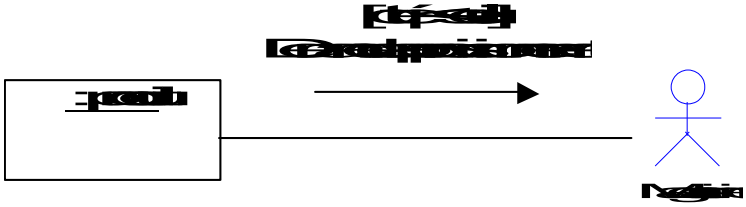


Ou



Ici, le message émis par le gérant implique la restitution immédiate du résultat du calcul : la valeur du stock.

Nb : l'émission de message peut également être soumis à une condition, qui s'exprime alors sur le texte du message.



Exemple : la demande de réapprovisionnement n'est envoyée au magasinier que lorsque la quantité en stock est inférieure au seuil de réapprovisionnement.

4.3.2. Diagrammes de séquence

Le diagramme de séquence est une variante du diagramme de collaboration.

Par opposition aux diagrammes de collaboration, les diagrammes de séquence possèdent intrinsèquement une dimension temporelle mais ne représente pas explicitement les liens entre les objets.

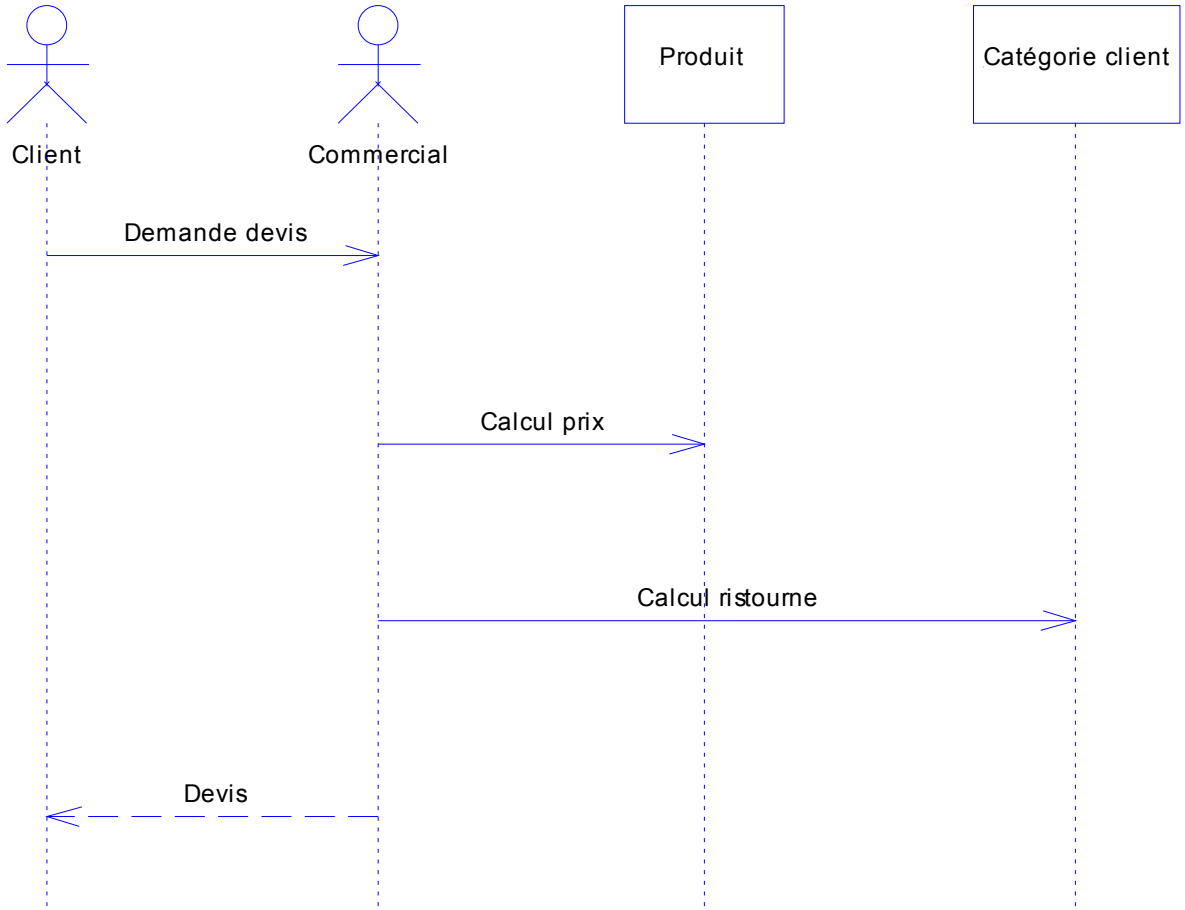
Ils privilégient ainsi la représentation temporelle à la représentation spatiale et sont plus aptes à modéliser les aspects dynamiques du système.

En revanche, ils ne rendent pas compte du contexte des objets de manière explicite, comme les diagrammes de collaboration.

Le diagramme de séquence permet de visualiser les messages par une lecture de haut en bas. L'axe vertical représente le temps, l'axe horizontal les objets qui collaborent. Une ligne verticale en pointillé est attachée à chaque objet et représente sa durée de vie.

Les messages sont représentés comme dans le diagramme de collaboration. (NB : un message de retour sera représenté avec des traits en pointillés)

Exemple :



4.3.2.1. Les interactions

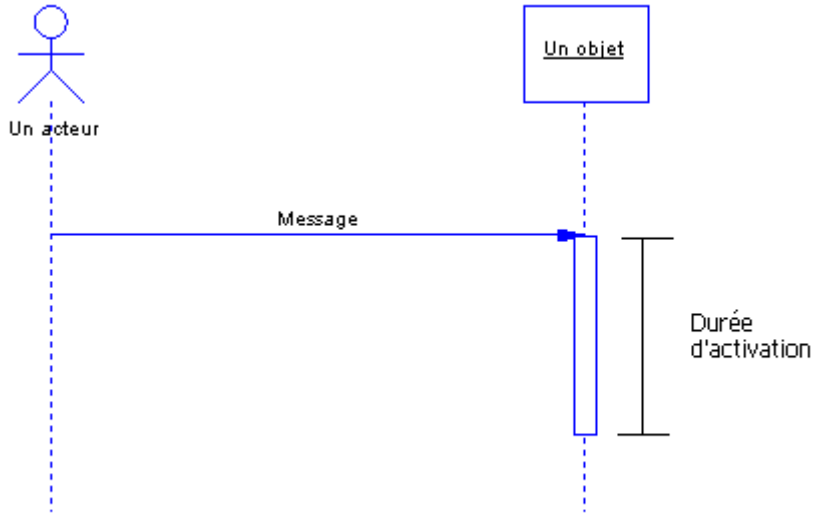
L'interaction se traduit par l'envoi d'un message entre objets. Le diagramme de séquence insiste sur la chronologie des objets en utilisant la ligne de vie des objets.

4.3.2.2. Les activations

Les diagrammes de séquence permettent de représenter les périodes d'activité des objets.

Une période d'activité correspond au temps pendant lequel un objet effectue une action, soit directement, soit par l'intermédiaire d'un autre objet qui lui sert de sous-traitant.

Formalisme : les périodes d'activité se représentent par des bandes rectangulaires placées sur la ligne de vie des objets.



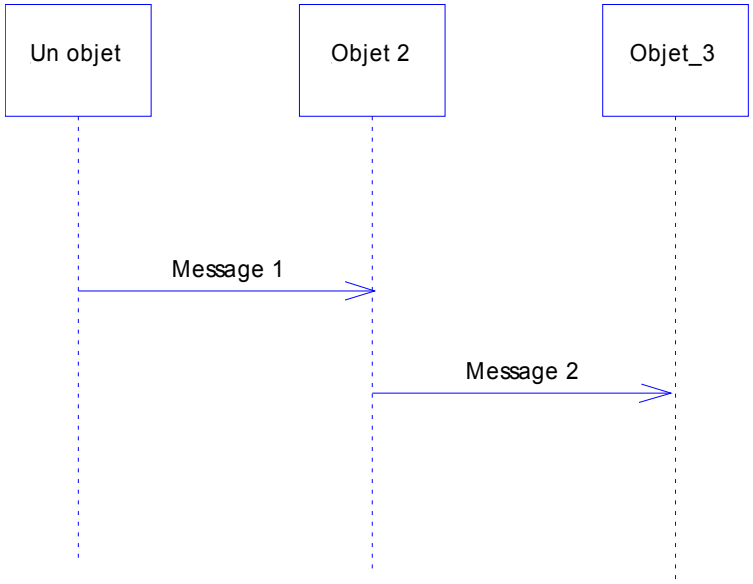
4.3.2.3. Les catégories de message

Les diagrammes de séquence distinguent 3 catégories d'envois de message :

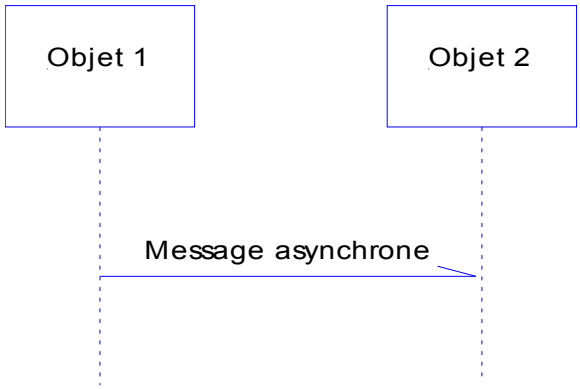
- flot de contrôle à plat

  Cette catégorie d'envois est utilisée pour indiquer le progression vers la prochaine étape d'une séquence.

  Formalisme : une flèche simple symbolise de tels messages.



  Alternativement, une demi-flèche peut être utilisée pour représenter explicitement des messages asynchrones pour des systèmes concurrents (la flèche pleine correspond alors à un message avec attente de prise en compte).

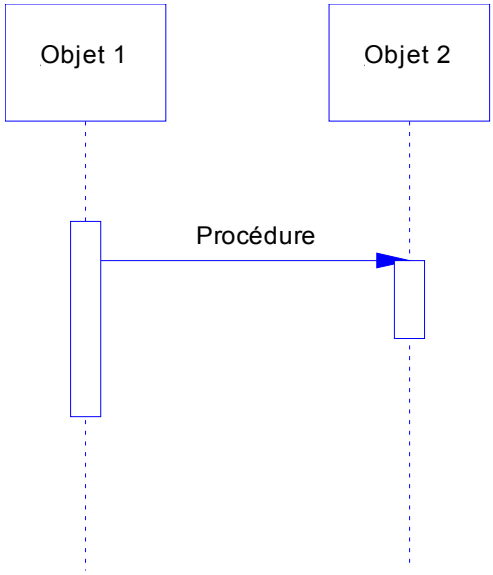


- appel de procédure (ou flot de contrôle emboîté)

  Dans un contrôle emboîté, la séquence emboîtée doit se terminer pour que la séquence englobante reprenne le contrôle.

  Un objet poursuit donc son exécution une fois le comportement initié par le message terminé.

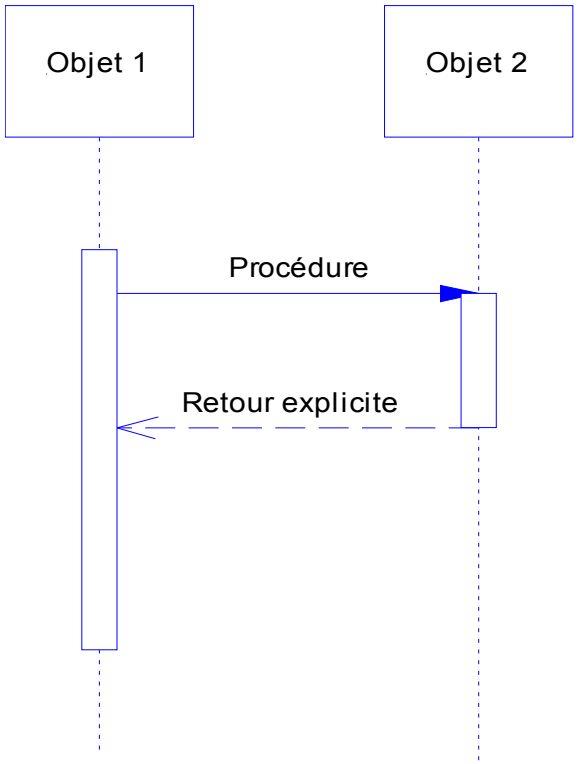
  Formalisme : Des flèches à extrémités pleines symbolisent de tels messages.



  L'objet 1 récupère le contrôle quand l'objet 2 a fini sa tâche.

- Retour de procédure

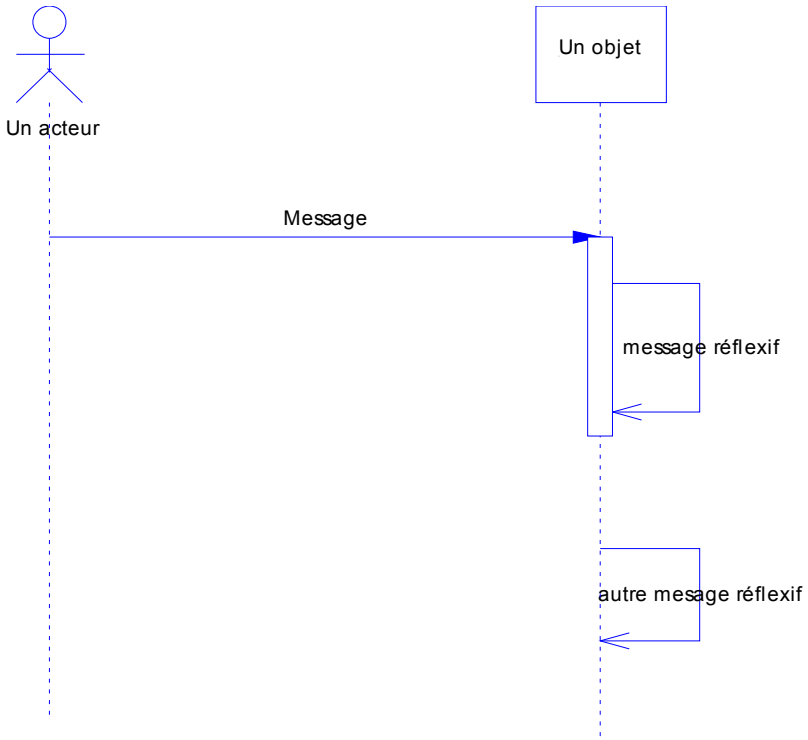
  Le retour de procédure est implicite à la fin d'une activation. Néanmoins, en cas d'envois de messages asynchrones, il s'avère utile pour montrer la fin de l'exécution d'une sous-procédure et le renvoi éventuel de paramètres.



4.3.2.4. Les messages réflexifs

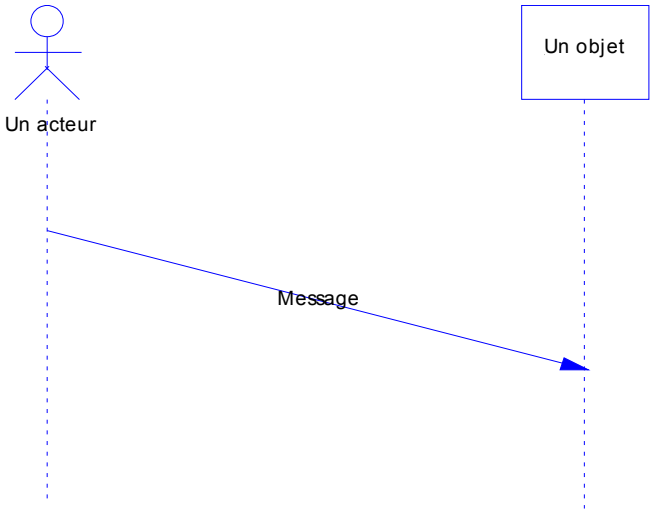
Un objet peut s'envoyer un message.

Formalisme : Cette situation se représente par une flèche qui revient en boucle sur la ligne de vie de l'objet.



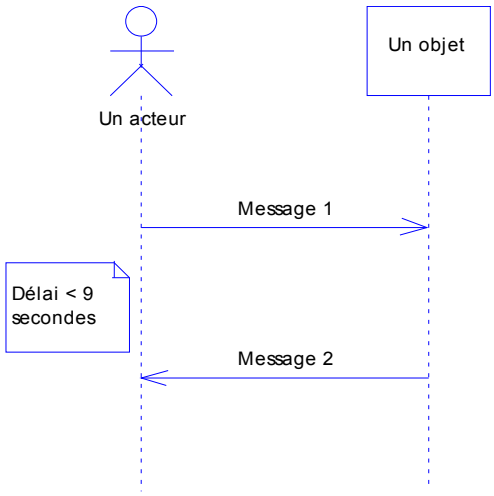
4.3.2.5. Les contraintes temporelles

Une flèche qui symbolise un message peut être représentée en oblique pour matérialiser les délais de transmission non négligeables par rapport à la dynamique générale de l'application.



Des annotations temporelles concernent les messages peuvent également être ajoutées.

Exemple :

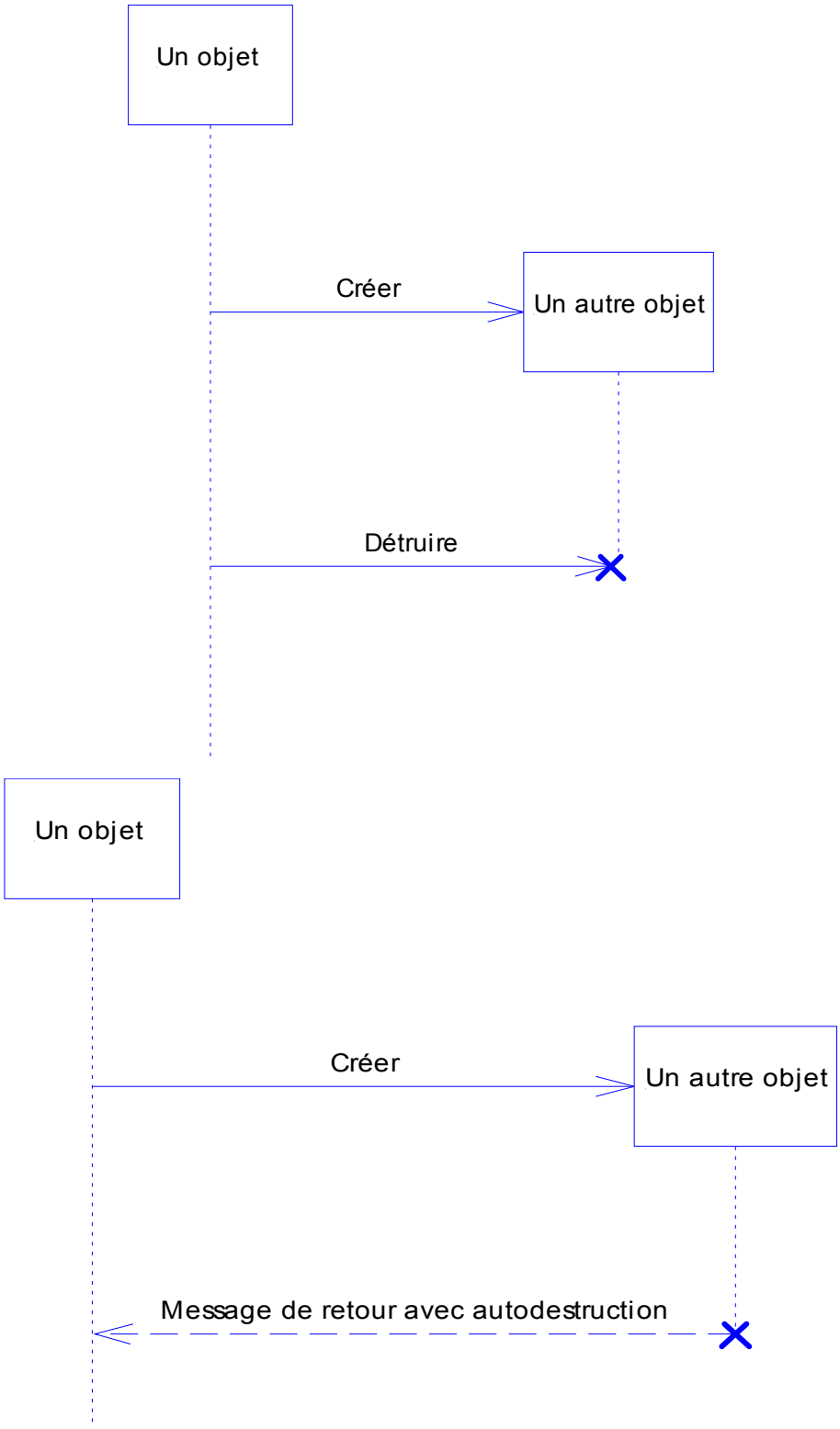


4.3.2.6. La ligne de vie

La ligne de vie des objets est représentée par une ligne verticale en traits pointillés, placée sous le symbole de l'objet concerné. Cette ligne de vie précise l'existence de l'objet concerné durant un certain laps de temps.

En général, une ligne de vie est représentée sur toute la hauteur du diagramme de séquence. Par contre, elle peut débuter et s'interrompre à l'intérieur du diagramme.

Formalisme : la création se représente en faisant pointer le message de création sur le rectangle qui symbolise l'objet créé. La destruction est indiquée par la fin de la ligne de vie et par une croix (X), soit à la hauteur du message qui cause la destruction, soit après le dernier message envoyé par un objet qui se suicide.



Pour représenter la collaboration entre les objets, on dispose donc dans UML de 2 diagrammes (collaboration et séquence). On peut utiliser l'un ou l'autre, voire les 2.

On peut ainsi décrire l'ensemble des interactions avec des messages complets (nom, séquence, résultat attendu, synchronisation et condition d'émission) sur un diagramme de collaboration et

compléter cette description par des diagrammes de séquence ne visualisant que les noms des messages.

4.3.3. diagrammes d'états-transitions

Ils ont pour rôle de représenter les traitements (opérations) qui vont gérer le domaine étudié. Ils définissent l'enchaînement des états de classe et font donc apparaître l'ordonnancement des travaux.

Le diagramme d'états-transition est associé à une classe pour laquelle on gère différents états : il permet de représenter tous les états possibles ainsi que les événements qui provoquent les changements d'état.

4.3.3.1. Caractéristiques et règles de construction

1. État

   Un état correspond à une situation durable dans laquelle se trouvent les objets d'une classe. On lui associe les règles de gestion et les activités particulières.

   État : objets d'une classe
   + règles de gestion
   + changements d'états

   La représentation symbolique des **états** d'une classe d'objets est la suivante (rectangle aux bords arrondis) :

   Exemple pour une commande :

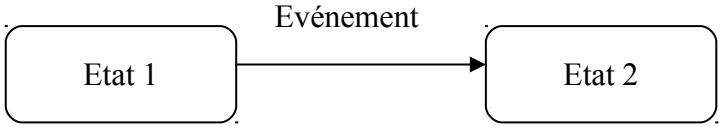
   - État "en préparation"
   - État "en cours"

2. Événements et transitions

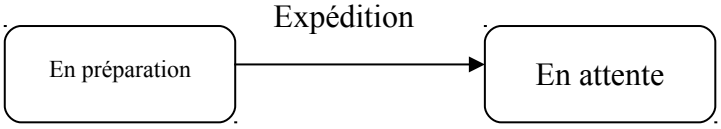
   Un objet passe d'un état à un autre suite à un **événement**, certains événements pouvant ne pas provoquer de changement d'état.

   Une transition est une relation entre 2 états. Elle est orientée ce qui signifie que l'état 2 est possible si certains événements sont vérifiés.

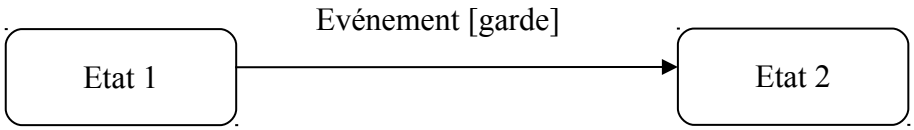
   Sa représentation symbolique est une flèche sur laquelle est annoté l'événement qui concourt au changement d'état.

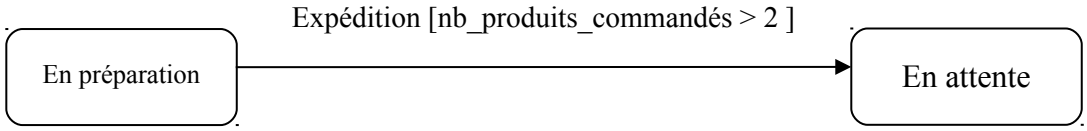
   Exemple : Une commande passera dans l'état "En attente" dès lors qu'elle aura été expédiée

La transition peut être soumise à la vérification d'une expression appelée "**expression de garde**" notée ainsi :



Exemple : La commande n'est expédiée que si la commande comporte au moins 3 produits.

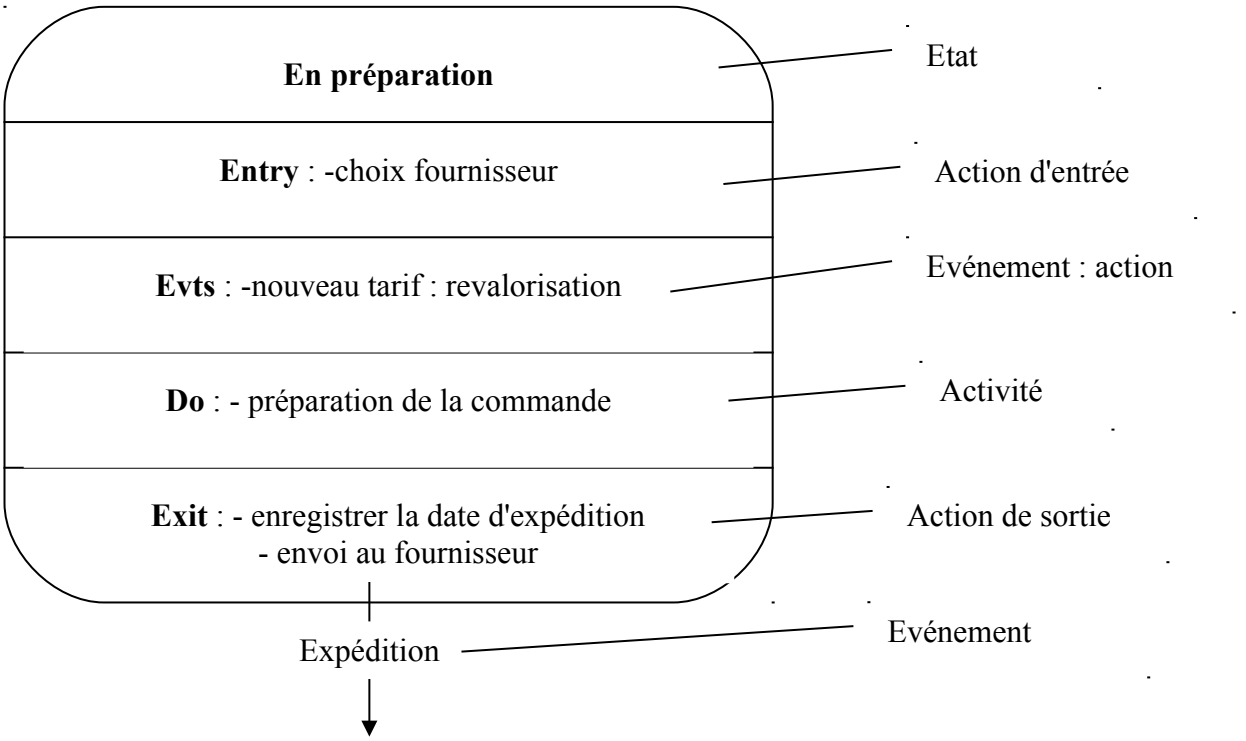


3. Les traitements

Les opérations de description des classes sont décrites dans le diagramme d'états-transitions sous forme **d'actions** et **d'activités**.

- Une **action** est une opération élémentaire et instantanée. Elle peut être associée à l'événement lui-même ou à l'entrée dans l'état ou à la sortie de l'état
- Une **activité** est une opération qui dure et qui est donc associée à un état. Elle peut être séquentielle ou cyclique :
  - La fin d'une activité **séquentielle** correspond à la sortie de l'état : une transition automatique est générée.
  - une activité **cyclique** ne se termine que par une transition de sortie identifiée.

Exemple de formalisme sur une commande "en préparation" :



4. La hiérarchie des états

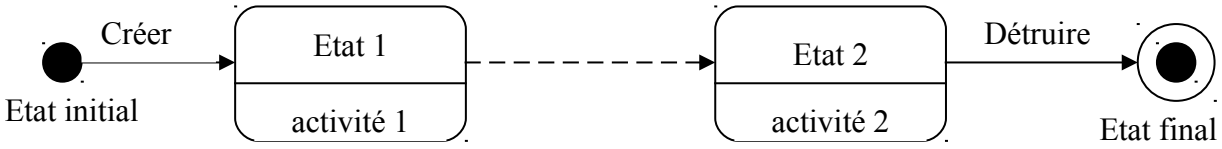
Pour faciliter la compréhension de traitements complexes, il est possible de créer des **superétats** qui contiendront des actions et activités communes à d'autres états (sous-états).

5. Les états prédéfinis

   Deux états sont prédéfinis :

   - l'état initial d'un objet : il est obligatoire et unique
   - l'état final : selon les événements, il peut exister plusieurs états finaux

   Leurs symbolismes sont les suivants :



4.3.4. Diagrammes d'activités

Le diagramme d'activité est attaché à une catégorie de classe et décrit le déroulement des activités de cette catégorie. Le déroulement s'appelle "**flot de contrôle**". Il indique la part prise par chaque objet dans l'exécution d'un travail. Il sera enrichi par les **conditions** de séquencement.

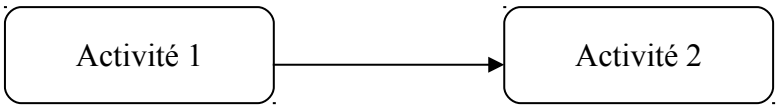
Il pourra comporter des synchronisations pour représenter les déroulements parallèles.

La notion de couloir d'activité va décrire les responsabilités en répartissant les activités entre les différents acteurs opérationnels.

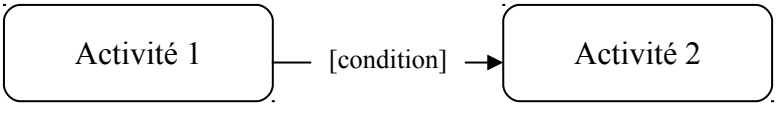
4.3.4.1. Le déroulement séquentiel des activités

Le diagramme d'états-transitions vu précédemment présente déjà un séquencement des activités d'une classe.

Le diagramme d'activités va modifier cette représentation pour n'en conserver que le séquencement. La notion d'état disparaît. On obtient ce graphe :



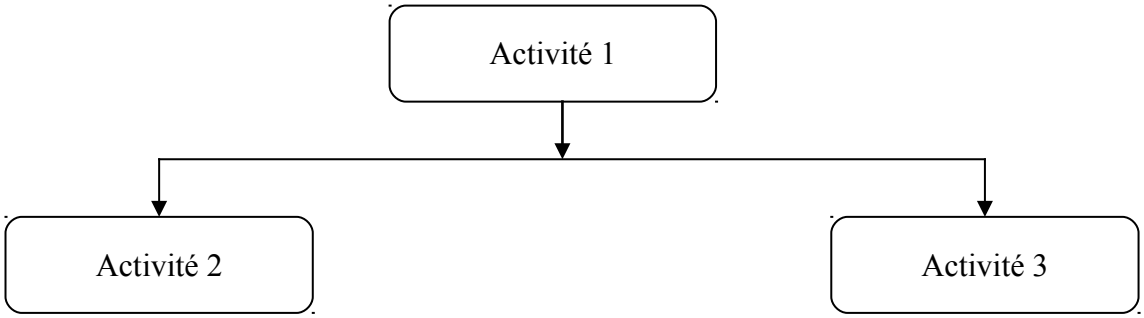
Comme dans le diagramme d'états-transitions, la transaction peut être complétée par une **condition de garde**.



4.3.4.2. La synchronisation

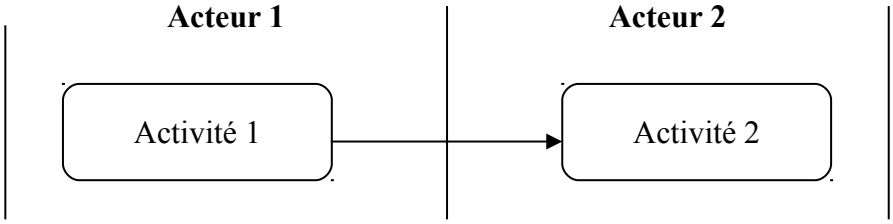
Les flots de contrôle parallèles sont séparés ou réunis par des barres de synchronisation.

Les activités 2 et 3 seront simultanées.



4.3.4.3. Les couloirs d'activités

Le diagramme d'activités fait intervenir les acteurs de chaque activité. Chaque activité sera placée dans une colonne (couloir) qui correspond à l'acteur.



5. Le processus unifié

Le processus unifié est un processus de développement logiciel : il regroupe les activités à mener pour transformer les besoins d'un utilisateur en système logiciel.

Caractéristiques essentielles du processus unifié :

- **Le processus unifié est à base de composants**,
- **Le processus unifié utilise le langage UML** (ensemble d'outils et de diagramme),
- **Le processus unifié est piloté par les cas d'utilisation**,
- **Centré sur l'architecture**,
- **Itératif et incrémental**.

5.1. Le processus unifié est piloté par les cas d'utilisation

5.1.1. Présentation générale

L'objectif principal d'un système logiciel est de rendre service à ses utilisateurs ; il faut par conséquent bien comprendre les désirs et les besoins des futurs utilisateurs. **Le processus de développement sera donc centré sur l'utilisateur**. Le terme utilisateur ne désigne pas seulement les utilisateurs humains mais également les autres systèmes. L'utilisateur représente donc une personne ou une chose dialoguant avec le système en cours de développement.

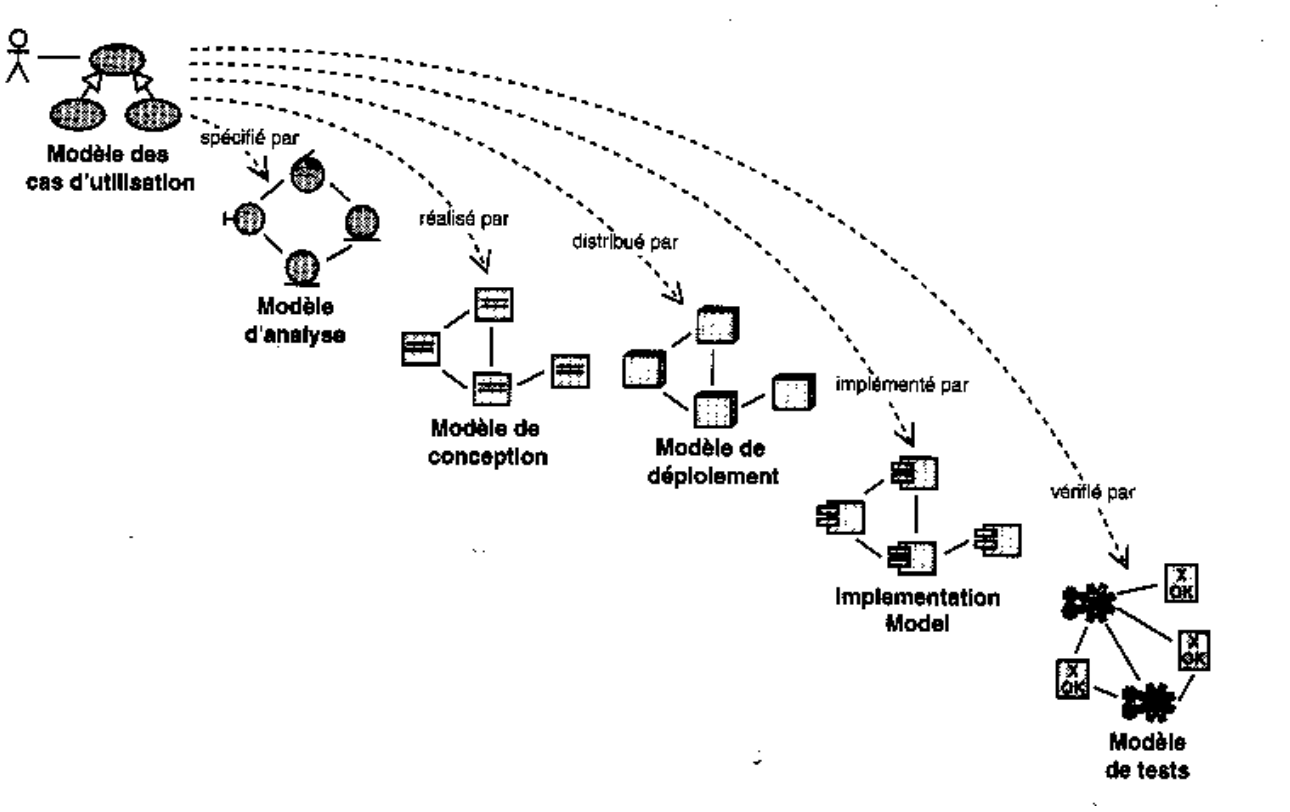
Ce type d'interaction est appelé cas d'utilisation.

Les cas d'utilisation font apparaître les besoins fonctionnels et leur ensemble constitue le modèle des cas d'utilisation qui décrit les fonctionnalités complètes du système.

Nous ne sommes pas dans une approche de type fonctionnelle mais une approche pilotée par des cas d'utilisation.

5.1.2. Stratégie des cas d'utilisation

Les cas d'utilisation ne sont pas un simple outil de spécification des besoins du système. Ils vont complètement guider le processus de développement à travers l'utilisation de modèles basés sur l'utilisation du langage UML.



- A partir du modèle des cas d'utilisation, les développeurs créent une série de modèles de conception et d'implémentation réalisant les cas d'utilisation.
- Chacun des modèles successifs est ensuite révisé pour en contrôler la conformité par rapport au modèle des cas d'utilisation.
- Enfin, les testeurs testent l'implémentation pour s'assurer que les composants du modèle d'implémentation mettent correctement en œuvre les cas d'utilisation.

Les cas d'utilisation garantissent la cohérence du processus de développement du système. S'il est vrai que les cas d'utilisation guident le processus de développement, ils ne sont pas sélectionnés de façon isolée, mais doivent absolument être développés "en tandem" avec l'architecture du système.

5.2. Le processus unifié est centré sur l'architecture

Dès le démarrage du processus, on aura une vue sur l'architecture à mettre en place. L'architecture d'un système logiciel peut être décrite comme les différentes vues du système qui doit être construit. L'architecture logicielle équivaut aux aspects statiques et dynamiques les plus significatifs du système. L'architecture émerge des besoins de l'entreprise, tels qu'ils sont exprimés par les utilisateurs et autres intervenants et tels qu'ils sont reflétés par les cas d'utilisation.

Elle subit également l'influence d'autres facteurs :

- la plate-forme sur laquelle devra s'exécuter le système ;

- les briques de bases réutilisables disponibles pour le développement ;
- les considérations de déploiement, les systèmes existants et les besoins non fonctionnels (performance, fiabilité..)

5.2.1. Liens entre cas d'utilisation et architecture

Tout produit est à la fois forme et fonction. Les cas d'utilisation doivent une fois réalisés, trouver leur place dans l'architecture. L'architecture doit prévoir la réalisation de tous les cas d'utilisation. L'architecture et les cas d'utilisation doivent évoluer de façon concomitante.

5.2.2. Marche à suivre

- L'architecte crée une ébauche grossière de l'architecture, en partant de l'aspect qui n'est pas propre aux cas d'utilisation (plate forme..). Bien que cette partie de l'architecture soit indépendante des cas d'utilisation. L'architecte doit avoir une compréhension globale de ceux ci avant d'en esquisser l'architecture.
- Il travaille ensuite, sur un sous ensemble des cas d'utilisation identifiés, ceux qui représentent les fonctions essentielles du système en cours de développement.
- L'architecture se dévoile peu à peu, au rythme de la spécification et de la maturation des cas d'utilisation, qui favorisent, à leur tour, le développement d'un nombre croissant de cas d'utilisation.

Ce processus se poursuit jusqu'à ce que l'architecture soit jugée stable.

5.3. Le processus unifié est itératif et incrémental

Le développement d'un produit logiciel destiné à la commercialisation est une vaste entreprise qui peut s'étendre sur plusieurs mois. On ne va pas tout développer d'un coup. On peut découper le travail en plusieurs parties qui sont autant de mini projets. Chacun d'entre eux représentant une itération qui donne lieu à un incrément.

Une itération désigne la succession des étapes de l'enchaînement d'activités, tandis qu'un incrément correspond à une avancée dans les différents stades de développement.

Le choix de ce qui doit être implémenté au cours d'une itération repose sur deux facteurs :

- Une itération prend en compte un certain nombre de cas d'utilisation qui ensemble, améliorent l'utilisabilité du produit à un certain stade de développement.
- L'itération traite en priorité les risques majeurs.

Un incrément constitue souvent un additif.

A chaque itération, les développeurs identifient et spécifient les cas d'utilisations pertinents, créent une conception en se laissant guider par l'architecture choisie, implémentent cette conception sous forme de composants et vérifie que ceux ci sont conformes aux cas d'utilisation. Dés qu'une itération répond aux objectifs fixés le développement passe à l'itération suivante.

Pour rentabiliser le développement il faut sélectionner les itérations nécessaires pour atteindre les objectifs du projet. Ces itérations devront se succéder dans un ordre logique.

Un projet réussi suivra un déroulement direct, établi dés le début par les développeurs et dont ils ne s'éloigneront que de façon très marginale. L'élimination des problèmes imprévus fait partie des objectifs de réduction des risques.

Avantages d'un processus itératif contrôlé :

- Permet de limiter les coûts, en termes de risques, aux strictes dépenses liées à une itération.
- Permet de limiter les risques de retard de mise sur le marché du produit développé (identification des problèmes dès les premiers stades de développement et non en phase de test comme avec l'approche « classique »).
- Permet d'accélérer le rythme de développement grâce à des objectifs clairs et à court terme.
- Permet de prendre en compte le fait que les besoins des utilisateurs et les exigences correspondantes ne peuvent être intégralement définis à l'avance et se dégagent peu à peu des itérations successives

L'architecture fournit la structure qui servira de cadre au travail effectué au cours des itérations, tandis que les cas d'utilisation définissent les objectifs et orientent le travail de chaque itération. Il ne faut donc pas mésestimer l'un des trois concepts.

5.4. Le cycle de vie du processus unifié

Le processus unifié répète un certain nombre de fois une série de cycles.

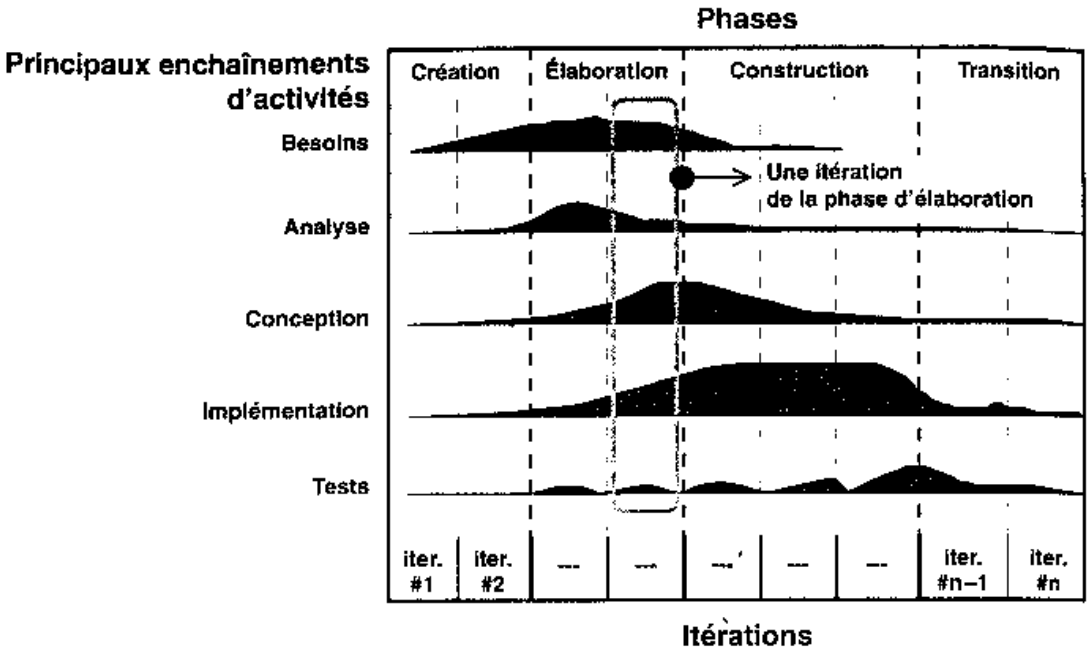
Tout cycle se conclut par la livraison d'une version du produit aux clients et s'articule en 4 phases : création, élaboration, construction et transition, chacune d'entre elles se subdivisant à son tour en itérations.

Chaque cycle se traduit par une nouvelle version du système. Ce produit se compose d'un corps de code source réparti sur plusieurs composants pouvant être compilés et exécutés et s'accompagne de manuels et de produits associés. Pour mener efficacement le cycle, les développeurs ont besoin de construire toutes les représentations du produit logiciel :

Modèle des cas d'utilisation	Expose les cas d'utilisation et leurs relations avec les utilisateurs
Modèle d'analyse	Détaille les cas d'utilisation et procède à une première répartition du comportement du système entre divers objets
Modèle de conception	Définit la structure statique du système sous forme de sous système, classes et interfaces ; Définit les cas d'utilisation réalisés sous forme de collaborations entre les sous systèmes les classes et les interfaces
Modèle d'implémentation	Intègre les composants (code source) et la correspondance entre les classes et les composants
Modèle de déploiement	Définit les nœuds physiques des ordinateurs et l'affectation de ces composants sur ces nœuds.
Modèle de test	Décrit les cas de test vérifiant les cas d'utilisation
Représentation de l'architecture	Description de l'architecture

Tous ces modèles sont liés. Ensemble, ils représentent le système comme un tout. Les éléments de chacun des modèles présentent des dépendances de traçabilité ; ce qui facilite la compréhension et les modifications ultérieures.

Présentation du cycle de vie de UP



Phase	Description et Enchaînement d'activités
Phase de création	Traduit une idée en vision de produit fini et présente une étude de rentabilité pour ce produit - Que va faire le système pour les utilisateurs ? - A quoi peut ressembler l'architecture d'un tel système ? - Quels sont l'organisation et les coûts du développement de ce produit ? On fait apparaître les principaux cas d'utilisation. L'architecture est provisoire, identification des risques majeurs et planification de la phase d'élaboration.
Phase d'élaboration	Permet de préciser la plupart des cas d'utilisation et de concevoir l'architecture du système. L'architecture doit être exprimée sous forme de vue de chacun des modèles. Émergence d'une **architecture de référence**. A l'issue de cette phase, le chef de projet doit être en mesure de prévoir les activités et d'estimer les ressources nécessaires à l'achèvement du projet.
Phase de construction	Moment où l'on construit le produit. L'architecture de référence se métamorphose en produit complet, elle est maintenant stable. Le produit contient tous les cas d'utilisation que les chefs de projet, en accord avec les utilisateurs ont décidé de mettre au point pour cette version. Celle ci doit encore avoir des anomalies

	qui peuvent être en partie résolue lors de la phase de transition.
Phase de transition	Le produit est en version bêta. Un groupe d'utilisateurs essaye le produit et détecte les anomalies et défauts. Cette phase suppose des activités comme la fabrication, la formation des utilisateurs clients, la mise en œuvre d'un service d'assistance et la correction des anomalies constatées.(où le report de leur correction à la version suivante)

5.5. Conclusion : un processus intégré

Le processus unifié est basé sur des composants. Il utilise UML et est basé sur les cas d'utilisation, l'architecture et le développement incrémental. Pour mettre en pratique ces idées il faut recourir à un processus multi-facettes prenant en considération les cycles, les phases, les enchaînements d'activités, la réduction des risques, le contrôle qualité, la gestion de projet et la gestion de configuration. Le processus unifié a mis en place un cadre général (framework) intégrant chacune de ces facettes.

6. Comparaisons entre MERISE et UML

Il ne s'agit pas ici d'élire une méthode ou un langage numéro 1 (UML n'est pas le remplaçant de MERISE) mais d'essayer de comprendre quels sont les apports de ces 2 démarches et en quoi elles peuvent êtres complémentaires.

Il conviendra ensuite au lecteur d'adapter leur utilisation (qui peut être combinée) en fonction du problème à explorer.

Reprenons pour cela les principaux points clé de MERISE que nous comparerons un par un à ceux d'UML :

- les principes
- la modélisation du métier
- la démarche

6.1. Les principes

MERISE repose sur 5 principes fondamentaux :

- l'approche systémique
- les cycles de construction du système d'information
- l'approche fonctionnelle
- la séparation données-traitements
- approche qui part du général vers le particulier

6.1.1. L'approche systémique

MERISE trouve ses fondements dans la théorie systémique qui découpe l'entreprise ne 3 sous-systèmes :

- le système de décision
- le système d'information
- le système opérant

Tout système étudié ne doit pas l'être de façon à tenir compte du système complet dans lequel il évolue.

Dans UML, l'approche par les cas d'utilisation constitue de ce fait une approche systémique. Le système étudié est considéré comme une boîte noire qui réagit à des sollicitations extérieurs qui sont formalisées par des flèches et dont l'origine est l'acteur.

Les cas d'utilisation traduisent ainsi la structure du système à modéliser . La structure interne du système (composée d'objets)est modélisée par les diagrammes de collaboration.

6.1.2. Les cycles de construction du système d'information

MERISE est une démarche qui repose sur 3 cycles :

- le cycle de vie
- le cycle d'abstraction
- le cycle de décision

6.1.2.1. Le cycle de vie

Dans la méthode Merise on distingue différentes périodes qui vont de la conception du système d'information à sa maintenance. On se situe dans une situation dynamique en considérant que le système d'information va naître, grandir, être entretenu puis va disparaître et être remplacé par un autre.

UML ne définit pas de cycle de vie. Le cycle de développement sous-jacent est itératif et incrémental, guidé par les cas d'utilisation et centré sur l'architecture.

6.1.2.2. Le cycle d'abstraction

Dans MERISE, ce cycle permet d'isoler à un niveau spécifique, les éléments significatifs contribuant à la description du système d'information. Les niveaux conceptuel, logique ou organisationnel, physique ou opérationnel se situent dans ce cycle.

UML offre plusieurs diagrammes pour modéliser le système aux différents niveaux d'abstraction (diagramme de séquences, de classes …). Mais à la différence de MERISE, il n'y a pas de diagramme spécifique par niveau d'abstraction. Le formalisme UML est le même tout au long du processus de fabrication.

UML laisse le soin de présenter les diagrammes cohérents qui contiennent des objets de même niveau de préoccupation.

6.1.5.3. Le cycle de décision

Il concerne les différentes décisions et choix qui sont effectués tout au long du cycle de vie, et permettent de faire valider petit à petit le système que l'on est en train de construire. Ces décisions marquent la fin d'une étape et le début d'une autre, ce sont des passages obligés avant de poursuivre l'étude du projet. Ces étapes de validation sont fondamentales dans MERISE et permettent l'appropriation du système d'information par l'ensemble de la communauté.

UML, tout comme MERISE, se soucie d'associer étroitement les utilisateurs dans les tâches d'analyse et de conception (notamment au niveau des cas d'utilisation).

6.1.3. L'approche fonctionnelle

MERISE propose une approche descendante où le système réel est décomposé en activités, elles-mêmes déclinées en fonctions.

Les fonctions sont composées de règles de gestion, elles-mêmes regroupées en opérations. Ces règles de gestion au niveau conceptuel génèrent des modules décomposés en module plus simple et ainsi de suite jusqu'à obtenir des modules élémentaires. Ceci correspond bien aux langages de

programmation structurée mais la réutilisation des modules présente des difficultés. En effet, les modules sont difficilement extensibles et exploitables pour de nouveaux systèmes.

A ce niveau, UML se démarque fortement de MERISE en proposant une architecture qui rend le système le plus indépendant possible des besoins, où les classes donnent naissance à des composants réutilisables dans différents contextes de besoins.

Le degré de réutilisabilité est donc plus fort dans UML mais nécessite un plus haut niveau d'abstraction.

6.1.4. La séparation données-traitements

Tout au long de sa démarche, la méthode MERISE est caractérisée par les approches conjointes et parallèles des données et des traitements, approches qui se traduisent par des modèles spécifiques. Les modèles de données rendent compte de l'aspect statique du système alors que les modèles de traitement rendent compte des aspects dynamiques.

UML, tout comme MERISE, traite des données et des traitements. Mais au lieu de les séparer, elle les associe.

Elle repose en effet sur une véritable approche objet qui intègre données et traitements.

6.1.5. L'approche qui part du général vers le particulier

MERISE reposant sur une démarche fonctionnelle : elle applique une méthode de type descendante : la vision globale du système doit être affinée par intégrations successives des différentes orientations retenues.

UML permet aussi bien une approche descendante qu'une approche ascendante.

Les différents diagrammes fournissent les supports pour matérialiser une démarche progressive, allant du global vers le détaillé.

L'avantage d'UML réside dans le fait que le concept d'objet peut être décliné sur tout le cycle d'abstraction, ce qui facilite une analyse ascendante.

6.2. La modélisation métier

Pour décrire le métier de l'entreprise, MERISE s'appuie sur les concepts de :

- domaine
- acteur
- flux
- modèles conceptuels et organisationnels.

6.2.1. Le domaine

Une des premières étapes dans MERISE consiste à délimiter le domaine de l'étude. Le domaine regroupe l'ensemble des processus contenus dans le système à étudier. Il est très important car il sert à fixer les limites du cadre de l'étude.

On retrouve dans UML la même importance à définir le domaine d'étude. De la définition fiable et stable du domaine d'étude dépend en effet le choix des cas d'utilisation, des acteurs …

6.2.2. L'acteur

Dans MERISE, le concept d'acteur apparaît très tôt dans la conception du système d'information (ex : modèle de flux).

MERISE distingue 2 types d'acteurs : des acteurs internes et externes. Les acteurs externes sont des acteurs qui n'appartiennent pas au domaine défini. L'acteur externe n'appartenant pas au champ d'étude, seules ses principales caractéristiques sont décrites.

Pour UML, le concept d'acteur signifie de fait acteur externe.

La différence avec MERISE réside dans le fait que les acteurs sont analysés du point de vue du rôle qu'ils jouent vis-à-vis du système (la même personne peut jouer plusieurs rôles).

6.2.3. Les flux

La modélisation des flux tient une place importante dans MERISE et se retrouve tout au long du processus de modélisation des traitements : du diagramme de flux très général visant à décrire le domaine au modèle organisationnel des traitements (MOT) qui décrit les processus en détails.

MERISE adopte ici aussi une approche descendante qui part du général vers le particulier.

Dans UML, le diagramme des cas d'utilisation est une représentation externe qui montre les liens de haut niveau sans représenter le détail des échanges. Au niveau interne, les diagrammes de séquence et les diagrammes d'activités permettent de représenter les flux.

La différence essentielle dans la modélisation des flux entre MERISE et UML provient de la façon d'aborder les processus : on les découpe par fonctions dans MERISE et on les aborde par les rôles joués par les acteurs dans UML.

6.2.4. Les modèles conceptuels et organisationnels

6.2.4.1. Les modèles conceptuels

- Le modèle conceptuel des données

  Dans MERISE, le MCD est la représentation de l'ensemble des données mémorisables du domaine sans tenir compte des aspects techniques de stockage ni se référer aux conditions d'utilisation par tel ou tel traitement.

  Pour élaborer un MCD, MERISE s'appuie sur les concepts suivants :

  - propriété
  - entité
  - association

  Le concept de propriété

  La propriété est la modélisation de l'information élémentaire. C'est un ensemble de données ayant la même structure et représentant des informations analogues. La modélisation des propriétés doit éviter les synonymes et les polysèmes.

Il n'y a pas de différence à ce niveau entre MERISE et UML. Les attributs des objets sont comparables aux propriétés des entités.

UML possède simplement un formalisme spécifique pour les informations calculées (cf. attribut dérivé) alors qu'elles sont fortement déconseillées dans MERISE et qu'elles n'interviennent que dans le MPD (modèle physique des données) après dégradation.

Le concept d'entité

L'entité est un groupe de propriétés permettant de modéliser un ensemble d'objets de même nature.

Dans UML, la notion d'entité correspond à la composante statique de la notion de classe dans le diagramme des classes.

La notion de classe est toutefois plus large que la notion d'entité : elle contient en effet des opérations qui ne sont pas modélisées dans MERISE. Les classes ont un comportement qui résulte de l'analyse de leurs responsabilités, offrent des services qui sont réalisés par des méthodes et des opérations ; les instances communiquent grâce à des messages. Autant de notions inconnues dans MERISE.

De plus, les classes ne sont pas utilisées uniquement pour modéliser les objets métier, elles servent aussi à modéliser les objets techniques.

Le diagramme des classes d'UML est donc plus riche que le MCD de MERISE : on peut transformer un MCD en diagramme de classes mais l'inverse n'est pas possible.

Ceci reflète la différence fondamentale entre UML et MERISE : UML intègre l'objet et associe donc au sein du même concept des aspects statiques et dynamiques.

Cette différence se traduit par la modélisation des opérations dans les classes mais également par la portée de la modélisation : à l'inverse de MERISE, les objets ne se limitent pas à la modélisation du processus métier.

Les classes d'UML respecteront par conséquent les mêmes règles de normalisation que dans MERISE mais sont crées dans une optique différente des entités.

Quant au formalisme, il n'y a pas de différence notoire entre MERISE et UML.

NB : la portée des attributs (public, protégé, ou privé) apparaissent dans MERISE 2 au niveau du modèle organisationnel des données.

Le concept d'association

Une association modélise un ensemble de liens de même nature entre deux ou plusieurs occurrences d'entités.

On ne note pas de différence flagrantes entre MERISE et UML à ce niveau. Les différences se situent essentiellement au niveau du formalisme des cardinalités par exemple (sens de lecture inversé par exemple …), ou de la description de l'association (description des 2 rôles en UML).

Les associations de MERISE qui contiennent des propriétés se traduisent en UML par des classes-associations.

Quant au concept de généralisation / spécialisation et la notion d'héritage qui en découle, on le retrouve également dans MERISE 2, tout comme la modélisation des contraintes (partition, exclusion, totalité ..)

Il faut noter qu'UML exprime certaines notions (comme l'agrégation ou la composition) avec un symbolisme particulier qui n'est pas présent dans MERISE.

La normalisation du modèle

La normalisation est un ensemble de règles introduites à l'origine dans le modèle relationnel. Elles ont pour objectif de garantir la cohérence de la base de données lors des différentes manipulations (insertion, mise à jour, suppression).

Transposées au modèle conceptuel des données, elles permettent de rapprocher le formalisme entité-association de MERISE, du modèle relationnel.

MERISE propose ainsi des règles de validation :

- pertinence
- identification
- vérification
- unicité
- homogénéité

Ensuite, le modèle doit respecter au moins les 3 premières formes normales.

Cette normalisation a fait la force de MERISE pour modéliser les données.

A la différence de MERISE, UML ne se préoccupe pas de normalisation. Rien n'empêche toutefois d'appliquer les règles de normalisation préconisées dans MERISE au diagramme de classes.

- Le modèle conceptuel des traitements

Dans MERISE, le modèle conceptuel des traitements (MCT) a pour objectif de représenter les activités exercées par le domaine. Il exprime ce que fait le domaine et non, où, comment, quand et par qui, ces activités sont réalisées (cette modélisation est réalisée au niveau organisationnel). A ce stade, on fait abstraction de l'organisation du domaine.

Le MCT repose sur plusieurs concepts :

- les événements
- les opérations
- la synchronisation

Les événements

MERISE distingue deux types d'événement :

- des événements déclencheurs provoquant une opération
- des événements résultats produits par une opération suite à une règle d'émission

Dans UML, les concepts de flux assez similaires. Seul le mode de représentation diffère (cf. diagramme de séquences).

Les opérations

Une opération décrit le comportement du domaine et de son système d'information face à un ou plusieurs événements. Une opération est déclenchée par la survenance d'un événement, ou de plusieurs événements synchronisés. Elle comprend l'ensemble des

activités que le domaine peut effectuer à partir des informations fournies par l'événement, et de celles déjà connues dans la mémoire du système d'information.

Dans UML, les opérations sont représentées dans des diagrammes d'activités. Les activités sont hiérarchiques et donc se décrivent elles-mêmes par des activités.

La synchronisation

La synchronisation est une condition préalable au démarrage d'une opération. Elle se traduit par une opération logique.

Dans UML, les synchronisations peuvent apparaître dans les diagrammes d'activités et d'états-transition.

6.2.4.2. Les modèles organisationnels

- Le modèle organisationnel des traitements

  Le MCT permet de décrire les fonctions sans référence aux ressources pour en assurer le fonctionnement.

  Le modèle organisationnel des traitements décrit la manière dont ces fonctions sont matériellement assurées.

  Dans UML, les aspects dynamiques de l'organisation sont étudiés par les diagrammes de séquence et les diagrammes d'activités. Le diagramme d'activités permet de visualiser le séquencement des tâches en les attribuant aux travailleurs d'interface et de contrôle. Les échanges et les contrôles apparaissent de façon claire.

  Ainsi, l'enchaînement des tâches dans le MOT trouve son équivalent dans le diagramme d'activités.

  Le diagramme de séquences apporte par contre une dimension supplémentaire par apport au MOT en faisant apparaître les objets entités.

  Toutefois, on retrouve également ce lien dans MERISE 2 au travers du MOTA (Modèle organisationnel des traitements analytique).

- Le modèle organisationnel des données

  Le MOD utilise le même formalisme et les mêmes concepts que le MCD.

  Les caractéristiques de chaque donnée sont enrichies par des notions de confidentialité, de sécurité, de besoin d'historique. On distingue les données privées, protégées et partagées.

  On retrouve ces notions dans UML, notamment au niveau du diagramme de classes.

6.3. La démarche

Deux points seront abordés pour comparer les démarches utilisées dans MERISE et UML :

- les modèles
- les étapes du processus d'élaboration du système d'information

6.3.1. Les modèles utilisés

MERISE utilise différents types de modèle en fonction du niveau d'abstraction et de la nature de l'objet étudié (données ou traitements) et se fonde avant tout sur une analyse fonctionnelle descendante.

UML propose une approche en 2 temps :

- la couche métier se rapproche des niveaux conceptuel et organisationnel
- la couche ressource de rapproche des niveaux logique et physique

Les diagrammes de la couche métier sont complétés dans la couche ressource sans changement de formalisme.

Il existe des équivalences entre les modèles MERISE et les modèles UML.

- ***Le diagramme de cas d'utilisation*** ne montre que les acteurs, les cas d'utilisation et leur relation : il ne matérialise pas les flux et les échanges. Il ne faut donc pas le confondre avec le diagramme des flux.
- ***Le diagramme de classes et le diagramme d'objets*** sont proches des MCD et MOD. Les différences fondamentales résident dans l'intégration des méthodes dans les classes et les objets.

  De plus, le diagramme des classes ne se limite pas aux seules entités métier comme le MCD et contient également des paquetages et des interfaces.
- ***Le diagramme de collaboration*** n'a pas d'équivalence
- ***Le diagramme d'états-transitions*** n'a pas d'équivalence dans MERISE. Il s'apparente au CVO (cycle de vie des entités et objets) dans MERISE 2
- ***Le diagramme d'activités*** présente des similitudes avec le diagramme des flux et se rapproche du MCT et MOT (MCTA et MOTA dans MERISE 2) pour fournir une vue globale de l'organisation.
- ***Le diagramme de composants*** montre la mise en œuvre physique des modèles logiques avec l'environnement de développement. Elle tient compte des problématiques de programmation, compilation, réutilisation…
- ***Le diagramme de déploiement*** est spécifique à UML.

Il n'est cependant pas très intéressant d'établir des liens de correspondance entre les modèles de MERISE et d'UML car les 2 modèles ne sont pas réalisés avec les mêmes objectifs et n'utilisent pas toujours les mêmes concepts.

6.3.2. Les étapes d'élaboration du système d'information

MERISE identifie 3 grandes étapes dans le processus d'élaboration d'un système d'information : la conception, la réalisation et la maintenance, qu'on peut détailler ainsi :

- Conception
  - Schéma directeur
  - Étude préalable

- Étude détaillée
- Réalisation
  - Étude technique
  - Production logicielle
  - Mise en œuvre
- Maintenance
  - Mise en service - Maintenance

A la différence de MERISE, UML ne propose pas de cycle précis : les organisations sont libres de choisir le cycle qui leur convient.

UML fonctionne sur un principe d'itérations qui ne s'oppose pas aux phases définies dans MERISE. MERISE découpe plus au travers de ses phases l'analyse métier et l'architecture logicielle. Dans UML, l'architecture logicielle a une place prépondérante et est intégrée très en amont dans l'élaboration du système d'information.

Dans UML, l'avancement du projet est mesuré par le nombre de cas d'utilisation, de classes... réellement implantées et non par la documentation produite.

Les itérations servent en outre à répartir l'intégration et les tests tout au long du processus d'élaboration du système d'information .

6.4. Conclusion

En résumé, on retiendra que :

- les niveaux d'abstraction ne sont pas nettement distingués dans UML
- il n'y a pas de différence de formalisme en fonction des niveaux d'abstraction dans UML
- UML intègre l'objet et a été conçu pour et autour de l'objet.
- UML est plus proche du langage de programmation que MERISE

7. Conclusion générale

Les principes de base de MERISE sont ses points forts. MERISE permet de modéliser le métier de l'entreprise indépendamment des techniques, aux niveaux conceptuel et organisationnel. Le système informatique est un sous-ensemble du système d'information.

Les modèles en nombre limité (6) sont progressivement élaborés et enrichis, et constituent des supports de communication et de participation pour les utilisateurs.

UML présente des caractéristiques voisines. Les modèles basés sur un nombre déterminé de diagrammes en fonction de la vue sont progressivement enrichis. Mais UML reste incontournable si l'entreprise veut utiliser les techniques objet.

Ainsi, même s'il existe des points de convergence, le passage de MERISE à UML n'est pas uniquement d'ordre syntaxique.