UML (Unified Modeling Language)

Introduction à la modélisation orientée objets avec UML

I. Vocation de ce document

Ce document s'adresse à de futurs ingénieurs qui seront confrontés dans leur vie professionnelle au développement d'applications informatiques industrielles, en tant que concepteurs aussi bien que clients.

De par sa fonction, l'ingénieur, qu'il soit spécialiste d'informatique ou non, doit être capable de spécifier clairement le problème qu'il doit résoudre. S'il n'est pas informaticien, il aura sans doute à dialoguer avec des équipes de conception pour s'assurer que ses spécifications sont bien comprises. S'il est responsable d'une équipe de développement, il aura à assimiler les spécifications qu'il aura contribué à établir, puis il devra en mener l'analyse et la conception avant de confier le codage proprement dit à des développeurs, puis à dialoguer avec les clients pour s'assurer de leur satisfaction.

Dans tous les cas, l'ingénieur aura besoin d'un langage ou d'une méthode de spécification et de modélisation pour communiquer avec ses collaborateurs, clients et fournisseurs. C'est dans ce cadre que nous présentons quelques éléments du langage UML (*Unified Modeling Language*), qui s'est imposé comme un standard que rencontrent tous les ingénieurs dans l'industrie informatique qui utilisent des langages orientés objets.

Nous considérons dans ce document que le lecteur a déjà été formé aux principales notions de la programmation orientée objets.

Cette présentation est conçue comme un support pragmatique pour faciliter la tâche du lecteur lors de sa première utilisation d'UML, en lui présentant les aspects les plus utiles et les principales difficultés auxquelles il risque d'être confronté, plutôt que comme un manuel de référence ou un catalogue exhaustif.

II. *Présentation générale d'UML*

http://uml.free.fr/index-cours.html

a. Introduction

Le génie logiciel et la méthodologie s'efforcent de couvrir tous les aspects de la vie du logiciel. Issus de l'expérience des développeurs, concepteurs et chefs de projets, ils sont en constante évolution, parallèlement à l'évolution des techniques informatiques et du savoir-faire des équipes.

Comme toutes les tentatives de mise à plat d'une expérience et d'un savoir-faire, les méthodologies ont parfois souffert d'une formalisation excessive, imposant aux développeurs des contraintes parfois contre-productives sur leur façon de travailler. Avec la mise en commun de l'expérience et la maturation des savoir-faire, on voit se développer à présent des méthodes de travail à la fois plus proches de la pratique réelle des experts et moins contraignantes.

UML signifie Unified Modeling Language. La justification de chacun de ces mots nous servira de fil conducteur pour cette présentation.

b. **Unified** : historique des méthodes de conception

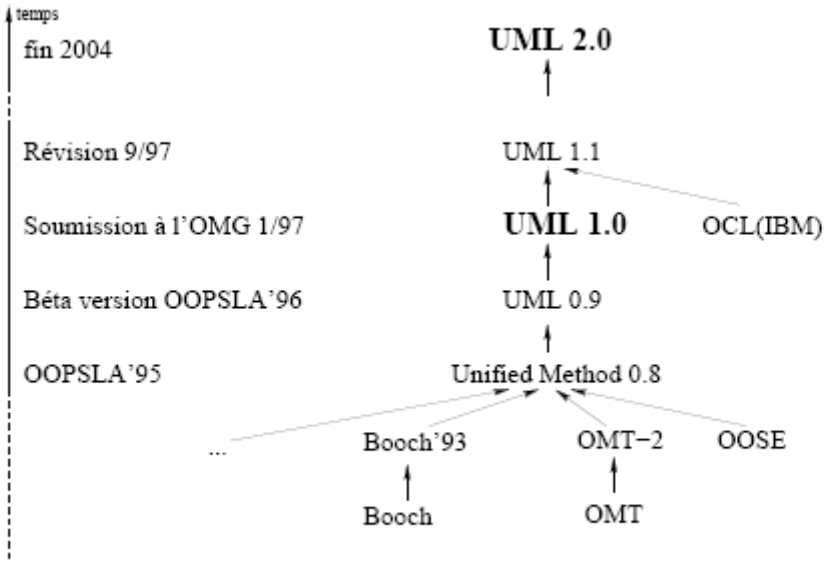


FIG. 1 – Historique de la constitution d'UML

À chacune des différentes phases de la conception d'un logiciel correspondent des problèmes ou des contraintes différentes. Naturellement, ces niveaux ont fait l'objet de recherches méthodologiques considérables depuis les années 80. Il en résulte que de nombreuses méthodes de développement ou d'analyse de logiciel ont vu le jour, chacune plus ou moins spécialisée ou adaptée à une démarche particulière, voire à un secteur industriel particulier (bases de données, matériel embarqué, ...).

Celles-ci ayant été développées indépendamment les unes des autres, elles sont souvent partiellement redondantes ou incompatibles entre elles lorsqu'elles font appel à des notations ou des terminologies différentes, voire à des faux amis.

De plus, à chaque méthode correspond un ou plusieurs moyens (plus ou moins formel) de représentation des résultats. Celui-ci peut être graphique (diagramme synoptique, plan physique d'un réseau, organigramme) ou textuel (expression d'un besoin en langage naturel, jusqu'au listing du code source).

Dans les années 90, un certain nombre de méthodes orientées objets ont émergé, en particulier les méthodes :

– OMT de James RUMBAUGH
– BOOCH de Grady BOOCH
– OOSE (*Object Oriented Software Engineering*) de Ivar JACOBSON à qui l'on doit les Use cases

En 1994, on recensait plus de 50 méthodologies orientées objets. C'est dans le but de remédier à cette dispersion que les « poids-lourds » de la méthodologie orientée objets ont entrepris de se regrouper autour d'un standard.

En octobre 1994, Grady Booch et James Rumbaugh se sont réunis au sein de la société RATIONAL dans le but de travailler à l'élaboration d'une méthode commune qui intègre les avantages de l'ensemble des méthodes reconnues, en corrigeant les défauts et en comblant les déficits. Lors de OOPSLA'95 (*Object Oriented Programming Systems, Languages and Applications*, la grande conférence de la programmation Orientée objets), ils présentent UNIFIED METHOD V0.8. En 1996, Ivar Jacobson les rejoint. Leurs travaux ne visent plus à constituer une **méthodologie**, mais un **langage.**

Leur initiative a été soutenue par de nombreuses sociétés, que ce soit des sociétés de développement (dont Microsoft, Oracle, Hewlett-Packard, IBM – qui a apporté son langage de contraintes OCL –, ...) ou des sociétés de conception d'ateliers logiciels. Un projet a été déposé en janvier 1997 à l'OMG 3 en vue de la normalisation d'un langage de modélisation. Après amendement, celui-ci a été accepté en novembre 97 par l'OMG sous la référence UML-1.1. La version UML-2.0 est annoncée pour la fin 2004.

c. **Modeling** : analyse et conception

Une bonne méthodologie de réalisation de logiciels suppose une bonne maîtrise de la distinction entre l'analyse et la conception.

Le lecteur verra qu'en pratique, le respect d'une distinction entre des phases d'analyse et de conception rigoureusement indépendantes n'est pas tenable, mais il est important d'avoir en tête la différence lorsqu'on s'apprête à réaliser un logiciel. Encore une fois, il est important de garder à l'esprit qu'UML n'offre pas une méthodologie pour l'analyse et la conception, mais un langage qui permet d'exprimer le résultat de ces phases.

Du point de vue des notations employées en UML, les différences entre l'analyse et la conception se traduisent avant tout par des différences de niveau de détail dans les diagrammes utilisés.
On peut ainsi noter les différences suivantes :

– Dans un diagramme de classes d'analyse, les seules classes qui apparaissent servent à décrire des objets concrets du domaine modélisé. Dans un diagramme de classes de conception, par opposition, on trouve aussi toutes les classes utilitaires destinées à assurer le fonctionnement du logiciel.

– Dans un diagramme de classes d'analyse, on peut se contenter de faire apparaître juste la dénomination des classes, avec parfois le nom de quelques attributs et méthodes quand ceux-ci découlent naturellement du domaine modélisé.

– Dans un diagramme de classes de conception, par opposition, tous les attributs et toutes les méthodes doivent apparaître de façon détaillée, avec tous les types de paramètres et les types de retour.

– Dans un diagramme de séquence d'analyse, les communications entre les principaux objets sont écrites sous forme textuelle, sans se soucier de la forme que prendront ces échanges lors de la réalisation du logiciel. Dans un diagramme de séquence de conception, par opposition, les échanges entre classes figurent sous la forme d'appels de méthodes dont les signatures sont totalement explicitées. Les étapes permettant de passer de diagrammes d'analyse à des diagrammes de conception et les motivations de la formalisation progressive que cela entraîne sont traitées dans le polycopié complémentaire.

d. **Langage** : méthodologie ou langage de modélisation ?

Il est important de bien faire la distinction entre une méthode qui est une démarche d'organisation et de conception en vue de résoudre un problème informatique, et le formalisme dont elle peut user pour exprimer le résultat (voir le glossaire en annexe).

Les grandes entreprises ont souvent leurs propres méthodes de conception ou de réalisation de projets informatiques. Celles-ci sont liées à des raisons historiques, d'organisation administrative interne ou encore à d'autres contraintes d'environnement (défense nationale, ...) et il n'est pas facile d'en changer. Il n'était donc pas réaliste de tenter de standardiser une méthodologie de conception au niveau mondial.

UML n'est pas une méthode, mais un langage. Il peut donc être utilisé sans remettre en cause les procédés habituels de conception de l'entreprise et, en particulier, les méthodes plus anciennes telles que celle proposée par OMT sont tout à fait utilisables.

D'ailleurs, la société RATIONAL (principale actrice de UML) propose son propre processus de conception appelé OBJECTORY et entièrement basé sur UML.

Ainsi, UML facilite la communication entre clients et concepteurs, ainsi qu'entre équipes de concepteurs. De plus, sa sémantique étant formellement définie dans (sous forme de diagramme UML), cela accélère le développement des outils graphiques d'atelier de génie logiciel permettant ainsi d'aller de la spécification (haut niveau) en UML vers la génération de code (JAVA, C++, ADA, ...).

De plus, cela autorise l'échange électronique de documents qui deviennent des spécifications exécutables en UML. UML ne se contente pas d'homogénéiser des formalismes existants, mais apporte également un certain nombre de nouveautés telles que la modélisation d'architectures distribuées ou la modélisation d'applications temps-réel avec gestion du multitâches, dont l'exposé dépasse le cadre de ce document.

e. Différentes vues et diagrammes d'**UML**

Toutes les vues proposées par UML sont complémentaires les unes des autres, elles permettent de mettre en évidence différents aspects d'un logiciel à réaliser. On peut organiser une présentation d'UML autour d'un découpage en vues, ou bien en différents diagrammes, selon qu'on sépare plutôt les aspects fonctionnels des aspects architecturaux, ou les aspects statiques des aspects dynamiques. Nous adopterons

plutôt dans la suite un découpage en diagrammes, mais nous commençons par présenter les différentes vues, qui sont les suivantes :

1 - la **vue fonctionnelle**, interactive, qui est représentée à l'aide de **diagrammes de cas et de diagrammes des séquences**. Elle cherche à appréhender les interactions entre les différents acteurs/utilisateurs et le système, sous forme d'objectif à atteindre d'un côté et sous forme chronologique de scénarios d'interaction typiques de l'autre.

2 - la **vue structurelle**, ou **statique**, réunit les **diagrammes de classes et les diagrammes de packages**. Les premiers favorisent la structuration des données et tentent d'identifier les objets/composants constituant le programme, leurs attributs, opérations et méthodes, ainsi que les liens ou associations qui les unissent. Les seconds s'attachent à regrouper les classes fortement liées entre elles en des composants les plus autonomes possibles. A l'intérieur de chaque package, on trouve un diagramme de classes.

3 - la **vue dynamique**, qui est exprimée par **les diagrammes d'états,** Cette vue est plus algorithmique et orientée « traitement », elle vise à décrire l'évolution (la dynamique) des objets complexes du programme tout au long de leur cycle de vie. De leur naissance à leur mort, les objets voient leurs changements d'états guidés par les interactions avec les autres objets.

Le **diagramme d'activité** est une sorte d'organigramme correspondant à une version simplifiée du diagramme d'états. Il permet de modéliser des activités qui se déroulent en parallèle les unes des autres, quand ce parallélisme peut poser problème. En général, les diagrammes d'états à eux seuls ne permettent pas de faire apparaître les problèmes spécifiques posés par la synchronisation des processus en concurrence, pour assurer la cohérence du comportement et l'absence d'inter-blocage. Etablir un diagramme d'activité peut aider à mettre au point un diagramme d'états.

Outre les diagrammes précédemment mentionnés, il existe aussi les diagrammes suivants, que nous ne présenterons pas, dans la mesure où ils relèvent plus spécifiquement de la conception ou de l'implémentation.

1 - les **diagrammes de collaboration**, en appoint de la **vue fonctionnelle**. Proches des scénarios ou diagrammes de séquences, ces diagrammes insistent moins sur le séquencement chronologique des événements. En numérotant les messages pour conserver l'ordre, ils insistent sur les liens entre objets émetteurs et récepteurs de messages, ainsi que sur des informations supplémentaires comme des conditions d'envoi ou des comportements en boucle, ce que ne permettent pas les diagrammes de séquence, trop linéaires.

2 - les **diagrammes de déploiement**, spécifiques de l'implémentation, qui indiquent sur quelle architecture matérielle seront déployés les différents processus qui réalisent l'application.

III. Le diagramme des cas (vue fonctionnelle)

http://uml.free.fr/cours/i-p10.html

a. Les cas d'utilisation

Le diagramme des cas est un apport d'Ivar Jacobson à UML.

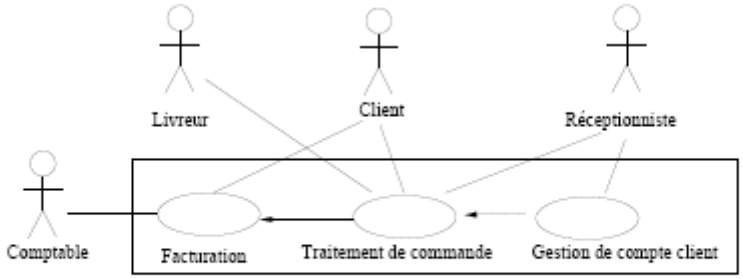


FIG. 2 – Exemple de diagramme de cas

Un cas d'utilisation (use case) modélise une interaction entre le système informatique à développer et un utilisateur ou acteur interagissant avec le système. Plus précisément, un cas d'utilisation décrit une séquence d'actions réalisées par le système qui produit un résultat observable pour un acteur.
Il y a en général deux types de description des use cases :

– une description textuelle de chaque cas ;
– le diagramme des cas, constituant une synthèse de l'ensemble des cas ;

Il n'existe pas de norme établie pour la description textuelle des cas. On y trouve généralement pour chaque cas son nom, un bref résumé de son déroulement, le contexte dans lequel il s'applique, les acteurs qu'il met en jeu, puis une description détaillée, faisant apparaître le déroulement nominal de toutes les interactions, les cas nécessitant des traitements d'exceptions, les effets du déroulement sur l'ensemble du système, etc.

b. Liens entre cas d'utilisation : include et extend

Il est parfois intéressant d'utiliser des liens entre cas (sans passer par un acteur), UML en fournit de deux types : la relation **utilise** (include) et la relation **étend** (extend).

Utilisation de cas : La relation **utilise** (include) est employée quand deux cas d'utilisation ont en commun une même fonctionnalité et que l'on souhaite factoriser celle-ci en créant un sous-cas, ou cas intermédiaire, afin de marquer les différences d'utilisation.

Extension de cas (extend) : Schématiquement, nous dirons qu'il y a extension d'un cas d'utilisation quand un cas est globalement similaire à un autre, tout en effectuant un peu plus de travail (voire un travail plus spécifique).

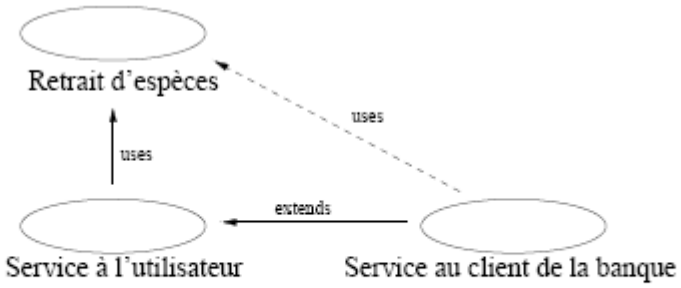


FIG. 3 – Exemple d'extension et d'utilisation

Par exemple, dans le cas d'un distributeur automatique de billets dans une banque, les utilisateurs du distributeur qui sont clients de la banque peuvent effectuer des opérations qui ne sont pas accessibles à l'utilisateur normal (par exemple, consultation de solde).

On dira que le cas « service au client de la banque » étend le cas « service à l'utilisateur ». Mais on peut dire aussi que les deux types de clients peuvent effectuer des retraits, si bien que les cas « service au client de la banque » et « service à l'utilisateur » utilisent tous les deux le cas « retrait d'espèces ». On représente cet exemple sur la figure 3.

IV. Le diagramme des classes (vue structurelle)

Le modèle qui va suivre a été particulièrement mis en exergue dans les méthodes orientées objets.

http://uml.free.fr/cours/i-p14.html

http://uml.free.fr/cours/i-p19.html + contrainte

a. Introduction au diagramme des classes

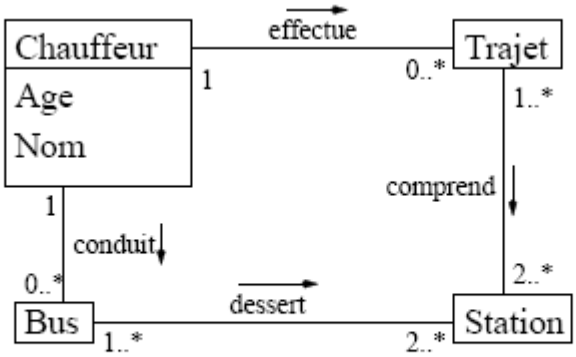


FIG. 4 – Un diagramme des classes

Un diagramme des classes décrit le type des objets ou données du système ainsi que les différentes formes de relation statiques qui les relient entre eux. On distingue classiquement deux types principaux de relations entre objets :

– les associations, bien connues des vieux modèles entité/association utilisés dans la conception des bases de données depuis les années 70

– les sous-types, particulièrement en vogue en conception orientée objets, puisqu'ils s'expriment très bien à l'aide de l'héritage en programmation.

La figure 4 présente un exemple de diagramme de classes très simple, tel qu'on pourrait en rencontrer en analyse. On voit qu'un simple coup d'œil suffit à se faire une première idée des entités modélisées et de leurs relations. Nous allons examiner successivement chacun des éléments qui le constituent. Auparavant, nous introduirons les packages.

b. Les diagrammes de packages

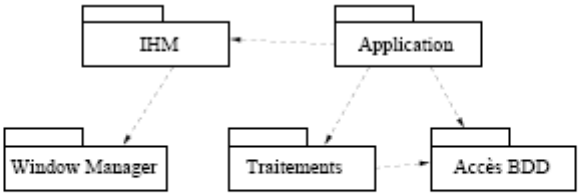


FIG. 5 – Exemple de diagramme de packages

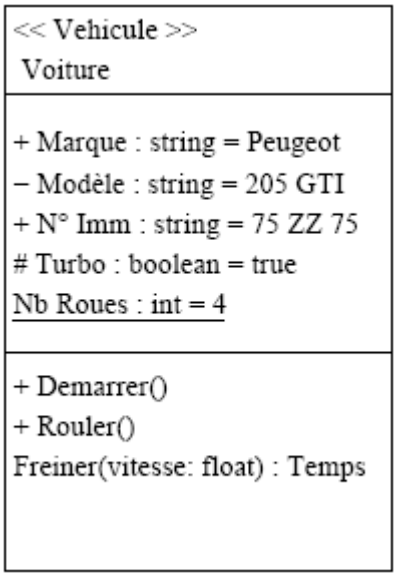
Il n'est pas toujours facile de décomposer proprement un grand projet logiciel en sous-systèmes cohérents. En pratique, il s'agit de regrouper entre elles des classes *liées* les unes aux autres de manière à faciliter la maintenance ou l'évolution du projet et de rendre aussi indépendantes que possible les différentes parties d'un logiciel. L'art de la conception de grands projets réside dans la division ou modularisation en « paquets » (package en JAVA), de faible dimension, minimisant les liens inter packages tout en favorisant les regroupements sémantiques.

Minimiser les liens inter-packages permet de confier la conception et le développement à des équipes séparées en évitant leurs interactions, donc l'éventuel cumul des délais, chaque équipe attendant qu'une autre ait terminé son travail. Les liens entre paquets sont exprimés par des relations de dépendance et sont représentés par une flèche en pointillé, comme il apparaît sur la figure 5. Certains outils vérifient qu'il n'y a pas de dépendances croisées entre paquets.

Diverses questions liées au découpage en paquets sont traitées dans le polycopié complémentaire.

c. Description d'une classe

L'ensemble des éléments qui décrivent une classe sont représentés On notera que l'on peut spécifier le package d'appartenance de la classe, dans lequel figure sa description complète, au dessus du nom de la classe.



i. Les attributs

Pour une classe, un attribut est une forme dégénérée d'association entre un objet de la classe et un objet de classe standard : c'est une variable qui lui est en général propre (dans certains cas elle peut être commune à la classe et non particulière à l'objet, on parle *d'attributs de classes*).

Dans le cadre d'un diagramme de classes UML en analyse, il faut s'interdire de représenter une variable propre à une classe sous forme d'attribut si la variable est elle-même instance d'une classe du modèle.

On représente donc les relations d'attribution entre classes sous forme d'associations et ce n'est qu'au moment du passage à la conception que l'on décidera quelles associations peuvent être représentées par des attributs.

En revanche, plus on se rapproche de la programmation, plus il faut envisager l'attribut comme un champ, une donnée propriété de l'objet, alors qu'une association est bien souvent une référence vers un autre objet.

La notation complète pour les attributs est la suivante :

visibilité nomAttribut : type = valeur par défaut

La visibilité (ou degré de protection) indique qui peut avoir accès à l'attribut (ou aux opérations dans le cas des opérations). Elle est symbolisée par un opérateur :

« + » pour une opération publique, c'est-à-dire accessible à toutes les classes (*a priori* associée à la classe propriétaire)

« # » pour les opérations protégées, autrement dit accessibles uniquement par les sous-classes (cf. section 4.7 à la page 18) de la classe propriétaire

« - » pour les opérations privées, inaccessibles à tout objet hors de la classe.

Il n'y a pas de visibilité par défaut en UML; l'absence de visibilité n'indique ni un attribut public ni privé, mais seulement que cette information n'est pas fournie. Le type est en général un type de base (entier, flottant, booléen, caractères, tableaux...), compte tenu de ce qui a été dit plus haut.

La valeur par défaut est affectée à l'attribut à la création des instances de la classe, à moins qu'une autre valeur ne soit spécifiée lors de cette création.

En analyse, on se contente souvent d'indiquer le nom des attributs. On note en les soulignant les attributs de classe, c'est-à-dire les attributs qui sont partagés par toutes les instances de la classe. Cette notion, représentée par le mot clef `static` en C++, se traduit par le fait que les instances n'auront pas dans la zone mémoire qui leur est allouée leur propre champ correspondant à l'attribut, mais iront chercher sa valeur directement dans la définition de la classe.

ii. Les opérations

Une opération, pour une classe donnée, est avant tout un travail qu'une classe doit mener à bien, un contrat qu'elle s'engage à tenir si une autre classe y fait appel. Sous l'angle de la programmation, il s'agit d'une méthode de la classe.

La notation complète pour les opérations est la suivante :

visibilité nomOpération (listeParamètres) : typeRetour {propriété}

La propriété, notée en français, ou sous forme d'équation logique, permet d'indiquer un prérequis ou un invariant que doit satisfaire l'opération.

Il et souhaitable de distinguer deux familles d'opérations, celles susceptibles de changer l'état de l'objet (ou un de ses attributs) et celles qui se contentent d'y accéder et de le visualiser sans l'altérer. On parle de modifiants, ou mutateurs et de requêtes ou accesseurs. On parle également d'opérations d'accès (qui se contentent de renvoyer la valeur d'un attribut) ou d'opérations de mise à jour qui se cantonnent à mettre à jour la valeur d'un attribut.

Notons qu'une opération ne se traduit pas toujours par une unique méthode. Une opération est invoquée sur un objet (un appel de procédure) alors qu'une méthode est le corps de cette même procédure. En cas de polymorphisme, quand un super-type a plusieurs sous-types, une même opération correspond à autant de méthodes qu'il y a de sous-types qui ont redéfini cette opération (+1).

d. Les associations

Les associations représentent des relations entre objets, c'est-à-dire entre des instances de classes. En général, une association est nommée. Par essence, elle a deux rôles, selon le sens dans lequel on la regarde. Le rapport entre un client et ses demandes n'a rien à voir avec celui qui unit une demande à son client, ne serait-ce que dans le sens où un client peut avoir un nombre quelconque de demandes alors qu'une demande n'a en général qu'un client propriétaire.

On cherchera, pour plus de lisibilité et pour faciliter les phases suivantes de la conception, à expliciter sur le diagramme le nom d'un rôle. En l'absence de nom explicite, il est d'usage de baptiser le rôle du nom de la classe cible.

À un rôle peut être ajoutée une indication de navigabilité, qui exprime une obligation de la part de l'objet source à identifier le ou les objets cibles, c'est-à-dire en connaître la ou les références. Quand une navigabilité n'existe que dans un seul sens de l'association, l'association est dite monodirectionnelle. On place généralement une flèche sur le lien qui la représente graphiquement, mais ce lien peut être omis quand il n'y a pas d'ambiguïté.

Une association est bidirectionnelle quand il y a navigabilité dans les deux sens, et induit la contrainte supplémentaire que les deux rôles sont inverses l'un de l'autre.

i. Les cardinalités (ou multiplicités)

Un rôle est doté d'une multiplicité qui fournit une indication sur le nombre d'objets d'une même classe participant à l'association. La notation est la suivante :

1	:	Obligatoire (un et un seul)
0..1	:	Optionnel (0 ou 1)
0..* ou *	:	Quelconque
n..*	:	Au moins *n*
n..*m*	:	Entre *n* et *m*
l,*n*,*m*	:	*l*, *n*, ou *m*

FIG. 8 – Cardinalités : notation

Les termes que l'on retrouve le plus souvent sont : 1,*, 1..* et 0..1 (cf. figure 9).

Mais on peut imaginer d'autres cardinalités comme 2, pour les extrémités d'une arête d'un graphe. Il faut noter que les cardinalités se lisent en UML dans le sens inverse du sens utilisé dans MERISE. Ici, la multiplicité qualifie la classe auprès de laquelle elle est notée.

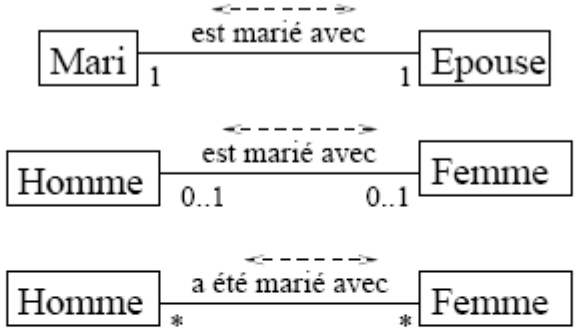


FIG. 9 – Cardinalités : exemples

Ainsi, sur la figure 4, on indique qu'un chauffeur peut conduire un nombre quelconque de bus.

ii. Attributs et classes d'association

Lorsque le lien sémantique est porteur de données qui constituent une classe du modèle, on utilise des classes d'association, comme c'est le cas sur la figure 11.

On peut aussi avoir à utiliser des associations aires, lorsque les liens sémantiques sont intrinsèquement partagés entre plusieurs objets.

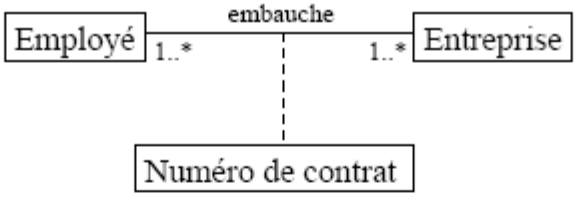


FIG. 10 – Attribut d'association

Il est fréquent qu'un lien sémantique entre deux classes soit porteur de données qui le caractérisent. On utilise alors des attributs d'association, comme c'est le cas sur la figure 10.

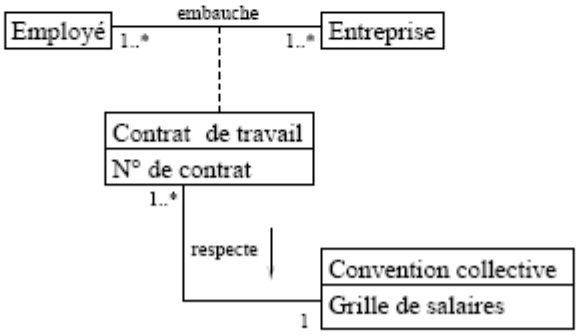


FIG. 11 – Classe d'association

iii. Associations et attributs dérivé

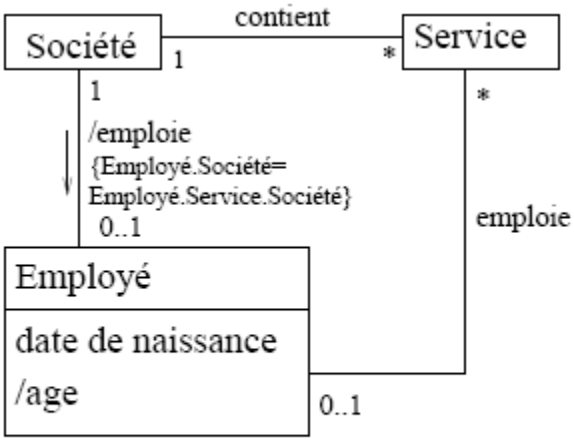


FIG. 13 – Associations et attributs dérivés

On parle d'attribut (ou d'association) dérivé(e) lorsque l'attribut ou l'association en question découle (ou dérive) d'autres attributs de la classe ou de ses sous-classes. On les symbolise comme il apparaît sur la figure 13 par le préfixe « / ».

Bien que les informations qu'ils portent soient souvent redondantes, puisqu'elles dérivent d'autres, il est utile de les faire figurer, en spécifiant le fait qu'ils dérivent d'autres, pour que les classes qui voudraient se servir d'une telle information puissent y accéder (sans passer par des chemins complexes ou avoir à faire des calculs).

On peut voir de tels attributs comme des caches sauvant une valeur pour ne pas avoir à la calculer sans cesse, valeur à mettre à jour au moment opportun, en accord avec les attributs dont ils dépendent. Les repérer comme dérivés évite les risques d'incohérence.

e. Sous-types et généralisation

Les relations de généralisation/spécialisation nous semblent naturelles car elles sont omniprésentes dans notre conception du monde. Toute classification, par exemple la taxonomie, ou classification des espèces animales, exprime des relations de généralisation/spécialisation.

Ainsi, les chimpanzés sont des « primates », qui eux-mêmes sont des mammifères, qui sont des animaux. On dira que le type chimpanzé est un sous-type du type primate, qui lui-même est un sous-type du type mammifère, qui lui-même est un sous-type du type animal. Réciproquement, le type primate est une généralisation des sous-types qu'il recouvre, à savoir le chimpanzé, le gorille, l'orang-outang, et l'homme.

Ces relations sont particulièrement utiles et répandues en analyse et conception orientées objets parce que, derrière toute idée de généralisation, il y a une idée de description commune, aussi bien pour ce qui est des attributs que des comportements.

Analyser ou concevoir en utilisant une hiérarchie de types, de classes ou d'objets permet donc de factoriser des attributs ou des comportements communs et n'avoir à décrire pour chaque sous-type spécialisé que ce qu'il a de spécifique.

En termes de modélisation, on parle de classification simple ou multiple. Dans le cadre de la classification simple, un objet ne peut appartenir qu'à un type et un seul, type qui peut éventuellement hériter d'un super-type. En classification multiple, un objet peut être décrit par plusieurs types qui n'ont pas forcément de rapport entre eux.

Par exemple, du point de vue de la zoologie, un bœuf est un mammifère, tandis que du point de vue du consommateur c'est un aliment riche en protéines.

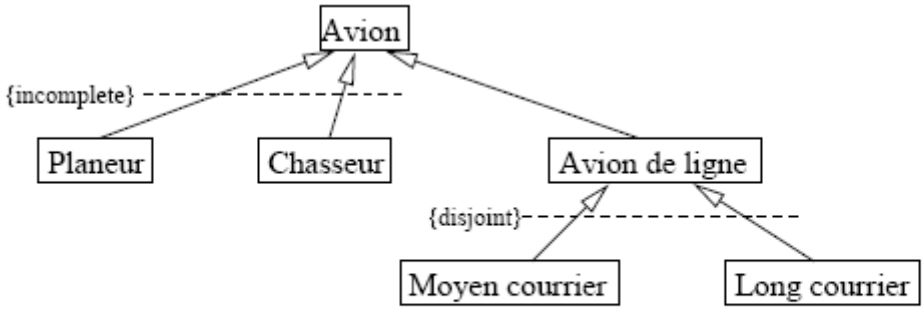


FIG. 14 – Exemple d'arbre de généralisation/spécialisation

La classification dynamique permet pour sa part de changer le type d'un objet en cours d'exécution du programme (en général quand on ne peut connaître son type à la création). À l'opposé, une classification statique ne permet pas, une fois instancié dans un type donné, de changer le type de l'objet.

La multiplicité ou la dynamicité des classifications sont à manipuler avec précaution car ces deux notions ne sont pas supportées par tous les langages. Ces aspects peuvent bien souvent être plus proprement (au sens du langage objets qui les traduira) pris en compte par la notion de classe abstraite ou d'interface.

Du point de vue de la programmation, bien entendu, l'héritage de classes sera utilisé pour réaliser ce sous-typage. Selon le langage que l'on utilise, toutefois, les contraintes qui pèsent sur l'héritage ne sont pas les mêmes. En C++, il est possible, moyennant de grandes précautions, d'utiliser l'héritage multiple entre classes. En JAVA, c'est impossible, mais les objets peuvent par contre disposer de plusieurs interfaces qui distinguent les différentes catégories de service qu'ils sont capables de rendre.

i. Agrégation et composition

http://uml.free.fr/cours/i-p15.html

Il est des cas particuliers d'associations qui posent souvent problème, ce sont les relations de la forme « partie de », pour lesquels plusieurs définitions existent et donc plusieurs modèles et manières de faire. Aussi faut-il s'entendre entre concepteurs sur les définitions qui vont suivre.

La composition, représentée par un losange noir, indique que l'objet « partie de » ne peut appartenir qu'à un seul tout. On considère en général que les parties d'une composition naissent et meurent avec l'objet propriétaire. Par exemple, les chambres d'un hôtel entretiennent une relation de composition avec l'hôtel. Si on rase l'hôtel, on détruit les chambres.

À l'inverse, on parle d'agrégation quand les objets « partie de » sont juste référencés par l'objet, qui peut y accéder, mais n'en est pas propriétaire. Cette relation est notée par un losange blanc.

FIG. 15 – Représentations de l'agrégation et la composition

Par exemple, un train est constitué d'une série de wagons peuvent être employés pour former d'autres trains. Si le train est démantelé, les wagons existent toujours. La confusion entre composition et agrégation illustre parfaitement la relative perméabilité entre l'analyse et la conception. Devant la difficulté à décider de la nature d'une relation, décision qui relève pourtant de l'analyse, on s'appuie généralement sur la conception pour fixer son choix.

En pratique, on se demande si l'objet « partie de » peut ou doit être détruit lorsqu'on détruit l'objet qui le contient et, si la réponse est affirmative, on choisit une relation de composition.

V. Les diagrammes de séquences (vue fonctionnelle)

http://uml.free.fr/cours/i-p19.html

Les diagrammes de séquences mettent en valeur les échanges de messages (déclenchant des événements) entre acteurs et objets (ou entre objets et objets) de manière chronologique, l'évolution du temps se lisant de haut en bas.

Chaque colonne correspond à un objet (décrit dans le diagramme des classes), ou éventuellement à un acteur, introduit dans le diagramme des cas. La *ligne de vie* de l'objet représente la durée de son interaction avec les autres objets du diagramme.

Un diagramme de séquences est un moyen semi-formel de capturer le comportement de tous les objets et acteurs impliqués dans un cas d'utilisation.

On peut indiquer un type de message particulier : les retours de fonction qui, bien entendu, ne concernent aucun message mais signifient la fin de l'appel de l'objet appelé. Ils permettent d'indiquer la libération de l'objet appelant (ou de l'acteur). Un emploi abusif de retours de fonction peut alourdir considérablement le diagramme, aussi un usage parcimonieux est-il conseillé.

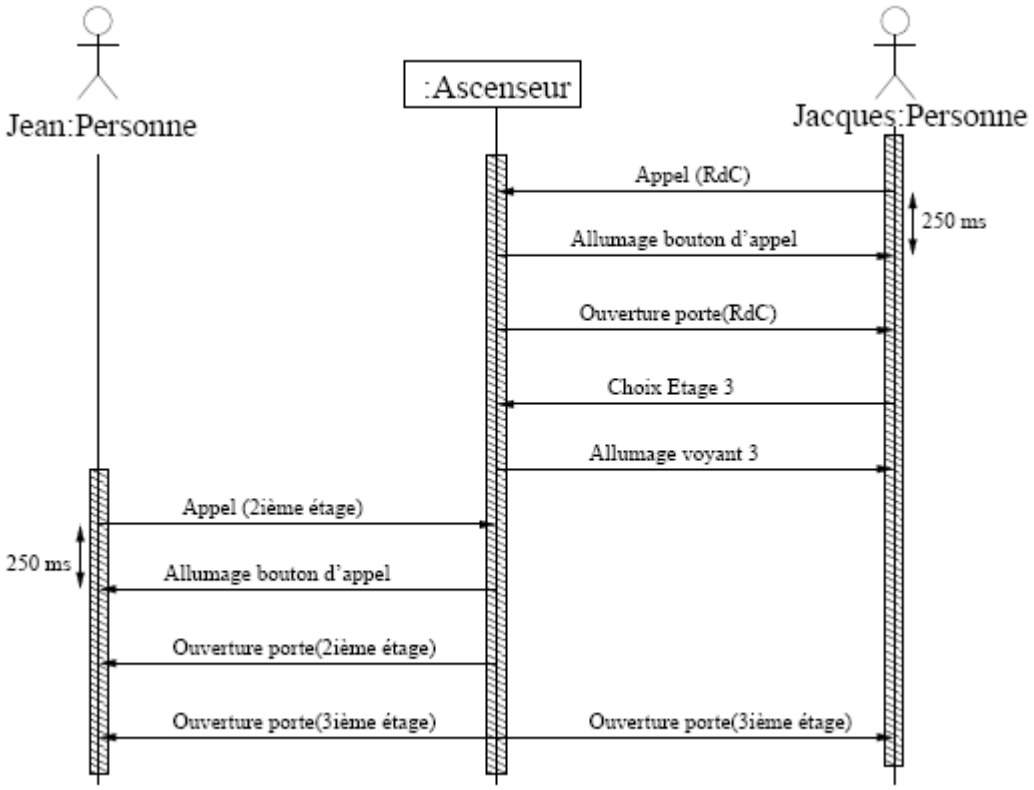


FIG. 17 – Un diagramme de séquences (ici, il s'agit d'un diagramme de séquence d'analyse)

On peut faire apparaître de nombreuses informations de contrôle le long de la ligne d'un objet. Par exemple, sur la figure 17, on a fait apparaître le délai de 250 millisecondes entre le moment où l'utilisateur appuie sur un bouton et le moment où le voyant correspondant s'allume.

Deux notions, liées au contrôle des interactions s'avèrent utiles :

– la première est la condition qui indique *quand* un message doit être envoyé. Le message ne sera transmis que si la condition est vérifiée. On indique les conditions entre crochets au-dessus de l'arc du message ;

– la seconde est la façon de marquer la répétitivité d'un envoi de message. Par exemple, si l'on doit répéter un appel pour toute une collection d'objets (pour tous les éléments de la liste des demandes), on fera précéder le dénominateur du message par un « * ».

Un diagramme des séquences permet de vérifier que tous les acteurs, les classes, les associations et les opérations ont bien été identifiés dans les diagrammes de cas et de classes. Il constitue par ailleurs une spécification utile pour le codage d'un algorithme ou la conception d'un automate.

Le diagramme de séquence de conception ci-dessous permet de voir un exemple dans lequel la signature des méthodes est à peu près formalisée.

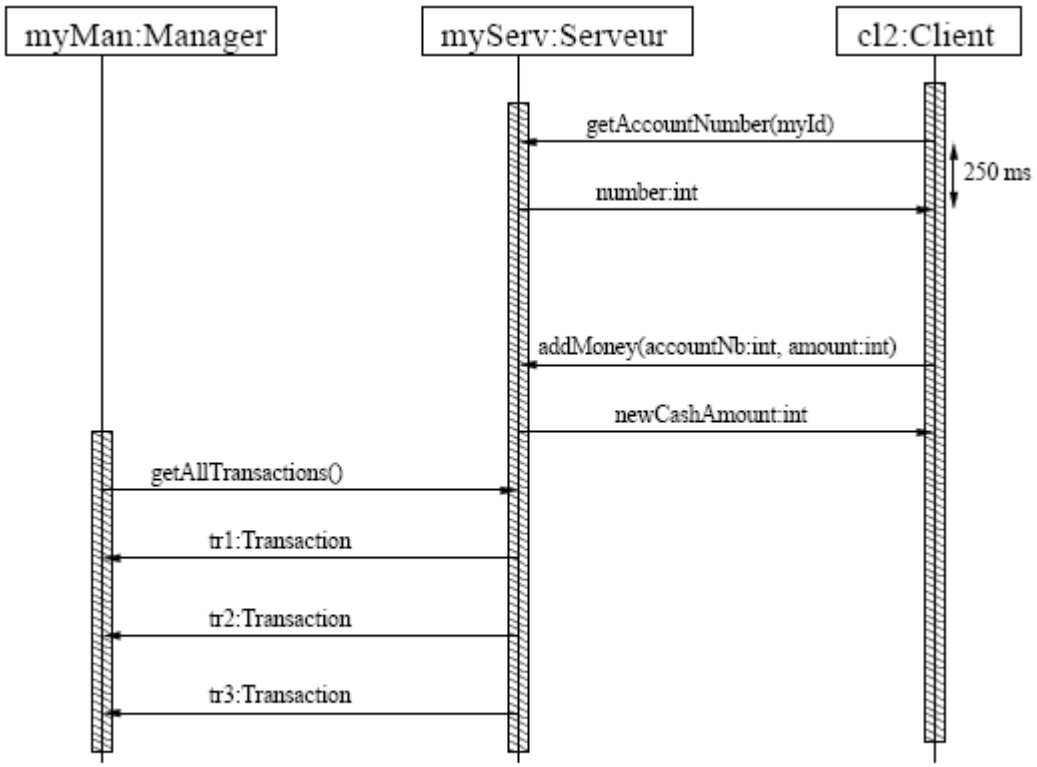


FIG. 18 – Un diagramme de séquences de conception

VI. *Les diagrammes d'états (vue dynamique)*

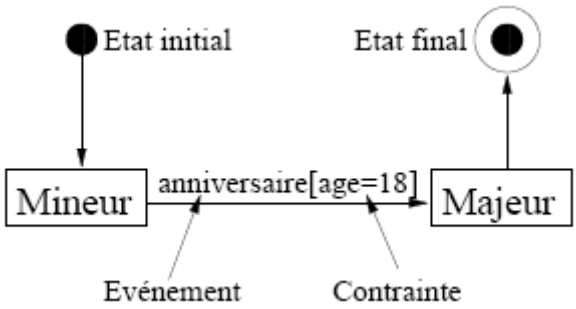


FIG. 19 – Le diagramme d'états d'un citoyen

Les diagrammes d'états décrivent tous les états possibles d'un objet (vu comme une machine à états). Ils indiquent en quoi ses changements d'états sont induits par des événements. Les modèles orientés objets s'appuient la plupart du temps sur les Statecharts de David Harel c'est aussi le cas d'UML.

Si les diagrammes de séquences regroupaient tous les objets impliqués dans un unique cas d'utilisation, les diagrammes d'états indiquent tous les changements d'états d'un seul objet à travers l'ensemble des cas d'utilisation dans lequel il est impliqué.

C'est donc une vue synthétique du fonctionnement dynamique d'un objet. Les diagrammes d'états identifient pour une classe donnée le comportement d'un objet tout au long de son cycle de vie (de la naissance ou état initial, symbolisée par le disque plein noir, à la mort ou état final, disque noir couronné de blanc).

a. Etats et Transitions

http://uml.free.fr/cours/i-p20.html

On distingue deux types d'information sur un diagramme d'états :

– des états, comme l'état initial, l'état final, ou les états courants (sur la figure 19, Mineur et Majeur).

– et des transitions, induisant un changement d'état, c'est-à-dire le passage d'un état à un autre.

Une transition est en général étiquetée par un label selon la syntaxe : NomÉvénement [Garde (ou contrainte)] / NomAction

Sur la figure 19, l'événement `anniversaire` fait passer de l'état Mineur dans l'état Majeur, si l'âge est 18 ans, Une garde est une condition attachée à une transition.

La transition gardée ne sera franchie que si la condition de garde est satisfaite.

En général, un état a une activité associée, qu'on indique sous le nom de l'état avec le mot-clef «do/».

b. Actions et activités

Il est important de faire la distinction entre une action (ponctuelle) attachée à une transition et une activité (continue), attachée à un état. On dira qu'une action se caractérise par un traitement bref et *atomique* (*insécable* donc *non préemptif*). En revanche, une activité n'est pas nécessairement instantanée et peut être interrompue par l'arrivée d'un événement extérieur, et de l'action qu'il induira.

Quand une transition ne dispose pas de label (donc pas d'événement), il est sous-entendu que la transition aura lieu dès la fin de l'activité. On parle de transition automatique.

Une auto-transition est une transition d'un état vers lui-même. On n'indique de telles transitions que si elles répondent à un événement externe auquel l'objet doit répondre.

Si un état répond à un événement à l'aide d'une action qui ne provoque pas un changement d'état, on parle d'action interne. On l'indique, pour alléger le diagramme, en écrivant : NomÉvénement / NomActionInterne dans le corps de l'état. Le comportement en cas d'action interne est complètement différent de celui de l'auto-transition

i. Exemple : diagramme d'états d'un réveil

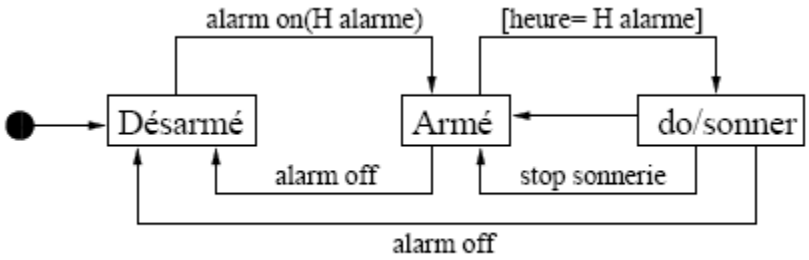


FIG. 21 – Exemple : diagramme d'états d'un réveil

On veut modéliser un réveil matin qui dispose de trois boutons : `alarm on/off`, `arrêt de la sonnerie` et `réglage de l'alarme`. À tout moment, le réveil dispose d'une heure d'alarme, l'heure à laquelle il doit sonner. Cette heure d'alarme peut être modifiée, on suppose simplement qu'on fixe une heure d'alarme quand on met l'alarme sur « `on` ».

Chaque appui sur les boutons constitue un événement qui déclenche une transition, si le réveil était dans un état sensible à l'événement.

– Si le réveil est désarmé et si on appuie sur `alarm on`, il passe dans l'état armé.

– Si le réveil est armé ou est en train de sonner et si on appuie sur `alarm off`, il passe dans l'état désarmé.

– Si le réveil est en train de sonner et si l'on appuie sur `arrêt`, il passe dans l'état armé.

Par ailleurs, si l'heure d'alarme est atteinte et si le réveil est armé, il se met à sonner. La transition automatique indique que, quand la sonnerie cesse toute seule – à la fin de l'activité do/sonner –, le réveil passe automatiquement dans l'état armé.

c. Le diagramme d'activité (vue dynamique)

http://uml.free.fr/cours/i-p21.html

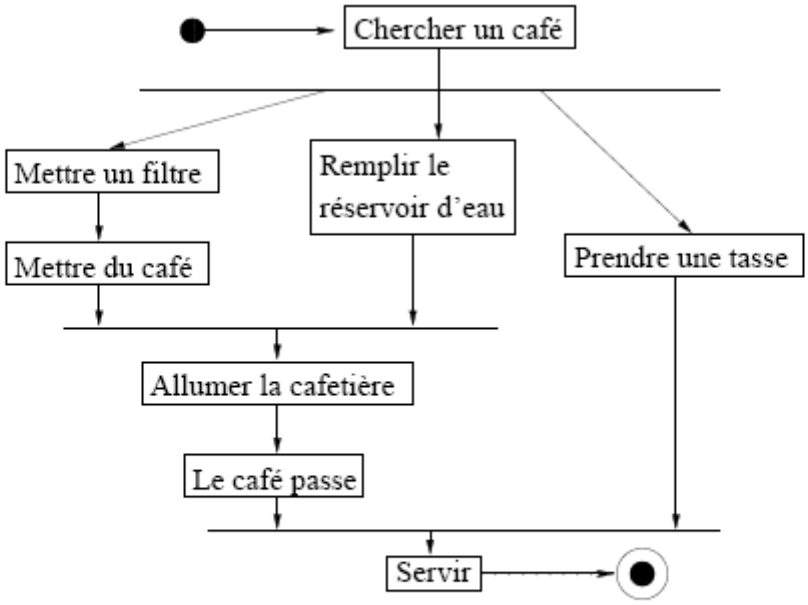


FIG. 25 – Exemple de diagramme d'activité : faire un café

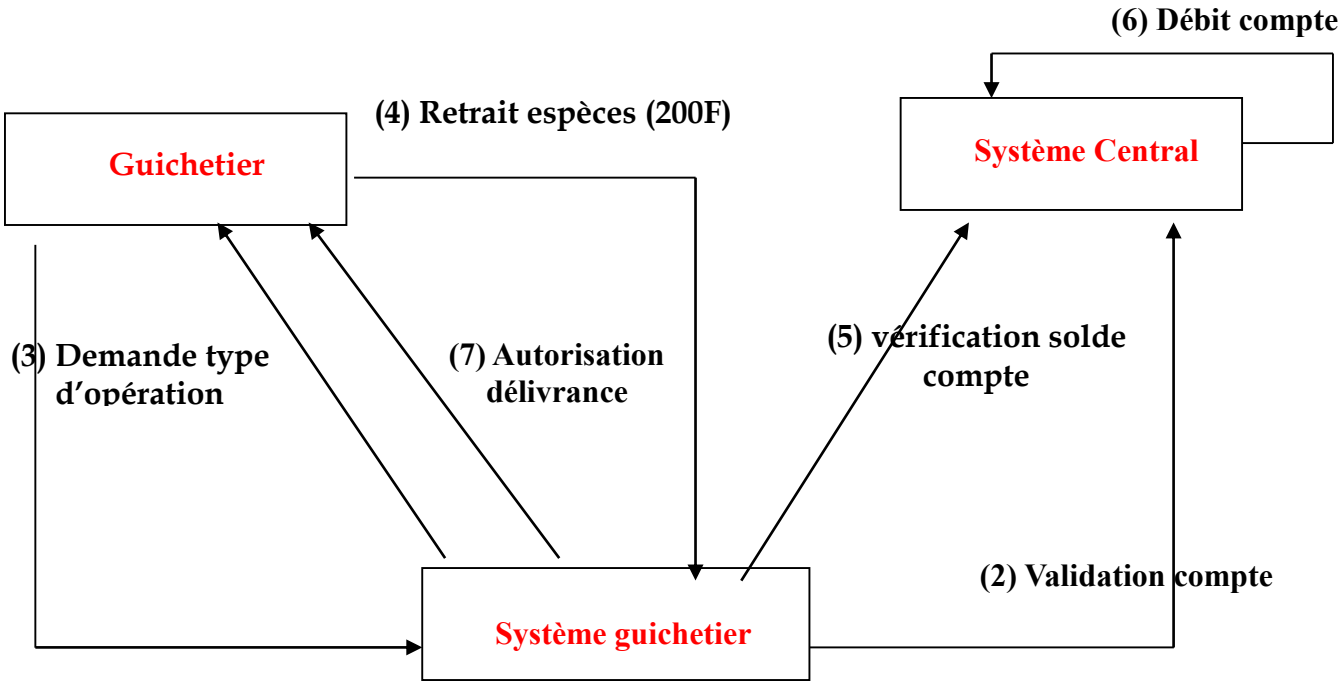
Le diagramme d'activité est un cas particulier de diagramme d'états, dans lequel à chaque état correspond une activité constituant un élément d'une tâche globale à réaliser. Le but de ce diagramme est de mettre en évidence les contraintes de séquentialité et de parallélisme qui pèsent sur la tâche globale.

Ainsi, sur la figure 25, on voit que, pour se faire un café, on peut simultanément mettre un filtre à la cafetière, remplir le réservoir d'eau et prendre une tasse mais que, par contre, il faut attendre d'avoir mis un filtre pour mettre du café.

d. Diagramme de collaboration

http://uml.free.fr/cours/i-p18.html

Les diagrammes de collaboration montrent des interactions entre objets.



e. Diagramme de composant et de déploiement

http://uml.free.fr/cours/i-p17.html

VII. Conclusion

Il faut retenir de ce document qu'UML fournit un moyen visuel standard pour spécifier, concevoir et documenter les applications orientées objets, en collectant ce qui se faisait de mieux dans les démarches méthodologiques préexistantes.

En fin de compte, l'intérêt de la normalisation d'un langage de modélisation tel que UML réside dans sa stabilité et son indépendance vis-à-vis de tout fournisseur d'outil logiciel.

Quant au langage lui-même, il a été conçu pour couvrir tous les aspects de l'analyse et de la conception d'un logiciel, en favorisant une démarche souple fondée sur les interactions entre les différentes vues que l'on peut avoir d'une application. Il permet enfin de fournir directement une bonne partie de la documentation de réalisation, dans le cours même des processus d'analyse et de conception. Finalement, les bénéfices que l'on retire d'UML sont multiples.

– D'une part, le langage, tel qu'il a été conçu, incite l'adoption d'une démarche de modélisation souple et itérative qui permet de converger efficacement vers une bonne analyse et une bonne conception.

– D'autre part, parce qu'il est formalisé (c'est-à-dire que sa sémantique est clairement spécifiée et décrite, on parle de langage semi-formel) et standard, le langage est supporté par différents outils, qui peuvent rendre des services dans la transition des différentes vues au code et du code aux différentes vues. Dans le premier cas, les outils deviennent capables de générer des squelettes de code. Dans le second cas, il s'agit d'engendrer des vues UML à partir du code, c'est le *reverse engineering* 5.

– Enfin, on peut imaginer à termes des outils de détection automatiques d'incohérence entre les différentes vues. Certains de ces aspects sont opérationnels (la détection d'interdépendances entre packages, la propagation des modifications d'une classe sur différentes vues, ...) d'autres sont encore du domaine de la recherche, nous ne les aborderons pas.

Glossaire

Acteur : un acteur est une entité externe au système à développer, mais il est actif dans les transactions avec celui-ci. Un acteur peut non seulement être humain (par exemple un client) mais encore purement matériel ou logiciel. Il est utilisé dans les diagrammes de cas.
Action : opération *instantanée* (ininterruptible) réalisée par un état ou une transition d'un état vers un autre état.
Activité : opération *continue* réalisé par un état lorsqu'il est actif.
Agrégation : une forme spéciale d'association de la forme « partie de » dans laquelle une classe (l'agrégat) est constituée d'un ensemble d'autres classes.
Cache : moyen de conserver en mémoire une information précédemment calculée dans le but de la rendre accessible rapidement. Par exemple les attributs dérivés sont un moyen de réaliser un cache.
Collections : type de données formées à partir d'éléments de base. Une collection peut être ordonnée (comme les listes) ou non (comme les ensembles).
Composition : forme particulière d'agrégation dans laquelle la vie de l'élément composite et celle des composants sont liées. Les composants peuvent être assimilés à des parties « physiques » du composite.
Entité/association : modèle assez ancien et essentiellement utilisé pour les systèmes de base de données (Merise, MCX).
État : situation d'un objet à un instant donné. La situation d'un objet est définie par ses attributs et par l'état des objets dont il dépend. Un objet peut changer d'état par le franchissement d'une transition. Un état composite (ou super-état) peut contenir plusieurs états. Il existe trois états particuliers appelés pseudo-état :

- l'état d'entrée (représenté par un disque noir), l'état de sortie (disque noir dans un cercle) et l'historique (un H dans un cercle).
- Le pseudo-état historique est une forme de pseudo-état d'entrée pour lequel l'objet (ou l'état composite) reprend la dernière situation active qu'il avait avant sa désactivation.
- L'état est l'élément principal du diagramme d'états proposé par UML.

Événement : un changement significatif (qui a une influence) dans l'environnement ou l'état d'un objet, parfaitement localisée dans le temps et dans l'espace. Un événement peut être constitué par la réception d'un message.
Extends : relation de dépendance de type *extension* entre deux use cases.
Formalisme : ensemble de notations associé à une sémantique formelle (et non ambiguë).
Garde (condition de) : condition qui doit être satisfaite pour valider le déclenchement de la transition qui lui est associée.
Généralisation : relation entre une classe plus générale et une classe plus spécifique. Une instance de l'élément le plus spécifique peut être utilisé à l'endroit où l'utilisation de l'élément le plus général est autorisé.
Héritage : caractéristique de la programmation orientée objets qui permet à une classe (la fille) héritant d'une autre classe (la mère) d'avoir l'ensemble des attributs et méthodes de la classe mère prédéfinis lors de sa création.
Interface : construction permettant de décrire le comportement visible de l'extérieur d'une classe, d'un objet ou d'une autre entité. L'interface comprend en particulier la signature des opérations de cette classe. Une interface est une classe abstraite sans attributs ni méthodes, contenant seulement des opérations abstraites.
Méthode (1) : démarche d'organisation et de conception en vue de résoudre un problème informatique (par opposition à un formalisme utilisé par cette méthode).
Méthode (2) : corps de la procédure invoquée par un objet lors de la demande d'une opération. Il peut y avoir plusieurs méthodes pour une même opération.
Message : mécanisme par lequel les objets communiquent entre eux. Un message est destiné à transmettre de l'information et/ou à demander une réaction en retour. La réception d'un message est à considérer comme un événement.

La syntaxe de l'émission d'un message est de la forme :
nomObjetDestination.nomMessage(parametresEventuels)
Un message peut être respectivement synchrone ou asynchrone suivant que l'émetteur attend ou non une réponse.
Opération : demande faite par message à un objet. L'objet invoque alors la méthode correspondante qui peut – en cas d'héritage – être définie dans la classe mère de la classe destination.
Signature : ensemble d'informations d'une méthode comprenant le nombre, l'ordre et le type des paramètres. La signature et le contexte d'appel permet au compilateur de choisir entre plusieurs méthodes de même nom pour une même opération donnée.
Stéréotype : moyen proposé par UML pour décrire une extension au langage. Un stéréotype permet de regrouper sous un même nom un ensemble de classes ayant des caractéristiques communes.
Transition : permet à un objet de passer d'un état source à un autre état destination. Si les états source et destination sont identiques, on parle d'auto-transition.
Uses : relation de dépendance de type *utilisation* entre deux use cases
Visibilité (ou protection) : caractéristique des attributs déterminant l'aspect de confidentialité de ceux-ci vis-à-vis des autres classes. Les visibilités suivantes sont reconnues par UML :

- \+ (publique) – toute classe à accès à cet attribut,
- # (protégé) – seules les méthodes de la classe ou d'une classe fille ont accès à cet attribut,
- − (privé) – l'accès et restreint aux méthodes de la classe même,

Exercice

a. Cas d'utilisation

Dans un établissement scolaire, on désire gérer la réservation des salles de cours ainsi que du matériel pédagogique (ordinateur portable ou/et Vidéo projecteur). Seuls les enseignants sont habilités à effectuer des réservations (sous réserve de disponibilité de la salle ou du matériel).

Le planning des salles peut quant à lui être consulté par tout le monde (enseignants et étudiants).

Par contre, le récapitulatif horaire par enseignant (calculé à partir du planning des salles) ne peut être consulté que par les enseignants.

Enfin, il existe pour chaque formation un enseignant responsable qui seul peut éditer le récapitulatif horaire pour l'ensemble de la formation.

b. Diagramme de séquence

Le déroulement normal d'utilisation d'un distributeur automatique de billets est le suivant :

- le client introduit sa carte bancaire
- la machine vérifie alors la validité de la carte et demande le code au client
- si le code est correct, elle envoie une demande d'autorisation de prélèvement au groupement de banques. Ce dernier renvoie le solde autorisé à prélever.
- le distributeur propose alors plusieurs montants à prélever
- le client saisit le montant à retirer
- après contrôle du montant par rapport au solde autorisé, le distributeur demande au client s'il désire un ticket
- Après la réponse du client, la carte est éjectée et récupérée par le client
- les billets sont alors délivrés (ainsi que le ticket)
- le client récupère enfin les billets et son ticket

Travail à Faire :
Modéliser cette situation à l'aide d'un diagramme de séquence en ne prenant en compte que le cas où tout se passe bien.

NB : on identifiera les scénarios qui peuvent poser problème en incluant des commentaires dans le diagramme

c. Gestion d'entrepôt de Stockage Diagramme de séquence & Classe

Pour faciliter sa gestion, un entrepôt de stockage envisage de s'informatiser.

Le logiciel à produire doit allouer automatique un emplacement pour le chargement des camions qui convoient le stock à entreposer.

Le fonctionnent du système informatique doit être le suivant :

- déchargement d'un camion : lors de l'arrivée d'un camion, un employé doit saisir dans le système les caractéristiques de chaque article ; le système produit alors une liste où figure un emplacement pour chaque article ;
- chargement d'un camion : les caractéristiques des articles à charger dans un camion sont saisies par un employé afin d'indiquer au système de libérer des emplacements.
- Le chargement et le déchargement sont réalisés manuellement.
- Les employés de l'entrepôt sont sous la responsabilité d'un chef dont le rôle est de superviser la bonne application des consignes.
-

Travail à Faire :
Donner le Diagramme de séquence pour le cas déchargement d'un camion
Donner le Diagramme de collaboration correspondant
Donner le Diagramme des classes