www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

Introduction à la modélisation orientée objets avec UML

Olivier Sigaud

Edition 2005-2006

Table des matières

1 Vocation de ce document **2**

2 Présentation générale d'UML **3**
2.1 Introduction 3
2.2 Unified : historique des méthodes de conception 4
2.3 Modeling : analyse et conception 5
2.4 Language : méthodologie ou langage de modélisation ? 6
2.5 Différentes vues et diagrammes d'UML 7

3 Le diagramme des cas (vue fonctionnelle) **8**
3.1 Les cas d'utilisation 8
3.2 Liens entre cas d'utilisation : include et extend 9

4 Le diagramme des classes (vue structurelle) **10**
4.1 Introduction au diagramme des classes 10
4.2 Les différents niveaux de description 11
4.3 Les diagrammes de packages 11
4.4 Description d'une classe 12
4.4.1 Les attributs 12
4.4.2 Les opérations 13
4.5 Les interfaces 14
4.6 Les associations 15
4.6.1 Les cardinalités (ou multiplicités) 15
4.6.2 Attributs et classes d'association 16
4.6.3 Qualificatifs 17

4.6.4 Associations et attributs dérivés 17
4.6.5 Ajout de contraintes et de règles 18
4.7 Sous-types et généralisation 18
4.7.1 Agrégation et composition 19
4.8 Classes paramétriques 20

5 Les diagrammes de séquences (vue fonctionnelle) 21

6 Les diagrammes d'états (vue dynamique) 23
6.1 Etats et Transitions 24
6.2 Actions et activités 24
6.2.1 Exemple : diagramme d'états d'un réveil 25
6.2.2 Événements spéciaux 26
6.3 Ordonnancement 26
6.4 Diagrammes hiérarchisés 27
6.4.1 Parallélisme et synchronisation 28
6.5 Le diagramme d'activité (vue dynamique) 29
6.6 Extension de UML : les stéréotypes 29
6.7 Conclusion 30

7 Glossaire 31
7.1 Extrait français du glossaire 31
7.2 Glossaire complet en anglais 33

8 NETographie (dernière mise à jour : 2001-2002) 48
8.1 Programmation objets et UML 48
8.2 Les patterns (ou patrons) 49

1 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. Nous renvoyons à un polycopié sur le langage JAVA si ce n'est pas le cas ([Sig05b]). Par ailleurs, les aspects de mise en œuvre d'une démarche reposant sur UML font l'objet d'un polycopié complémentaire ([Sig05a]). Nous nous contentons ici d'exposer les éléments du langage standard de modélisation orientée objets UML, en décrivant sommairement les différentes vues des applications qu'il permet de modéliser.

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. Nous invitons le lecteur à consulter les ouvrages de référence ([JBR97a, JBR97b, JBR97c]) pour une information approfondie dès lors que ce premier tour d'horizon lui aura permis de s'orienter.

2 Présentation générale d'UML

2.1 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, qui se veut un instrument de capitalisation des savoir-faire puisqu'il propose un langage qui soit commun à tous les experts du logiciel, va dans le sens de cet assou-

plissement des contraintes méthodologiques.

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

2.2 Unified : historique des méthodes de conception

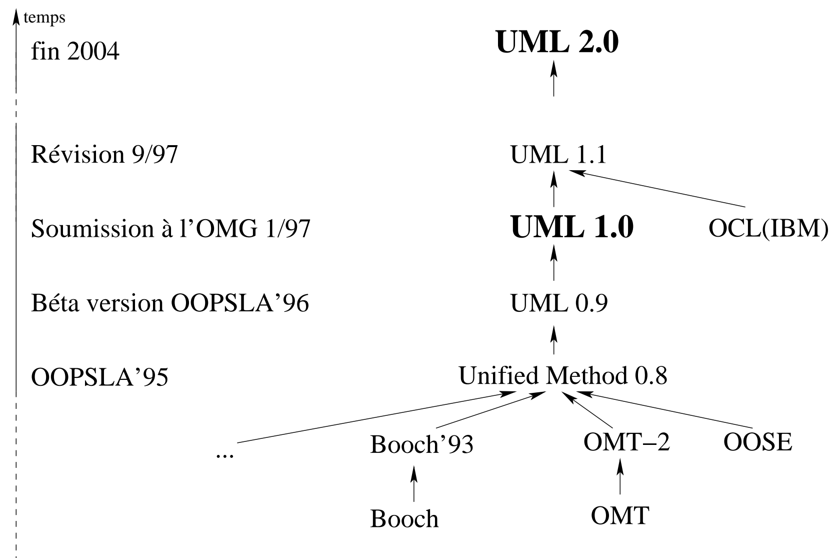

FIG. 1 – Historique de la constitution d'UML

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, ...) [url1]. 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 [Rum96],

– BOOCH de Grady BOOCH [Boo94],
– OOSE (*Object Oriented Software Engineering*) de Ivar JACOBSON à qui l'on doit les Use cases [JCJO92] [1].

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 [url3] 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** [2]. Leur initiative a été soutenue par de nombreuses sociétés, que ce soit des sociétés de développement (dont Microsoft, Oracle, Hewlet-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.

2.3 Modeling : analyse et conception

Une bonne méthodologie de réalisation de logiciels suppose une bonne maîtrise de la distinction entre l'analyse et la conception, distinction que nous exposons dans le polycopié complémentaire ([Sig05a]). 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

[1] cas d'utilisations, cf. la section 3 à la page 8
[2] voir la section 2.4 à la page 6
[3] *Object Management Group*, qui s'est rendu célèbre pour la norme CORBA [url4]

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 écrits 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 traités dans le polycopié complémentaire ([Sig05a]).

2.4 Language : 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 [JBR97a] 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 [4] dans [JBR97b] (sous forme de diagramme UML), cela accélère le développement des outils graphiques d'atelier de génie logiciel permettant ainsi d'aller de la spécification

[4] Les contraintes elles-mêmes font l'objet d'une définition formelle grâce au langage OCL (Object Constraint Language, voir la section 4.6.5 à la page 18).

(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 multi-tâches, dont l'exposé dépasse le cadre de ce document.

2.5 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, fera l'objet des sections 3 à la page 8 et 5 à la page 21. 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**, présentée dans la section 4 à la page 10, 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, sera introduite dans la section 6 à la page 23. 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 changement 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'interblocage. 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.

3 Le diagramme des cas (vue fonctionnelle)

3.1 Les cas d'utilisation

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

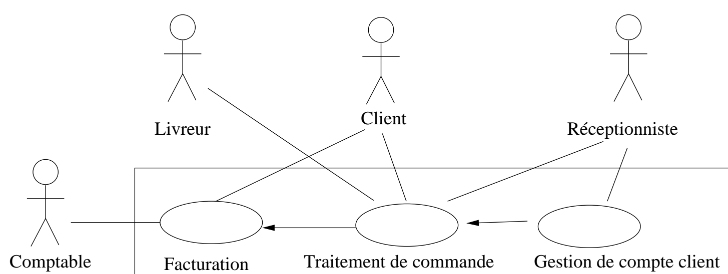

FIG. 2 – Exemple de diagramme de cas

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.

3.2 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). Cette notion – à utiliser avec discernement – permet d'identifier des cas particuliers (comme des procédures à suivre en cas d'incident) dès le début ou lorsque l'attitude face à un utilisateur spécifique du système doit être spécialisée ou adaptée. Il s'agit *grosso modo* d'une variation du cas d'utilisation normale.

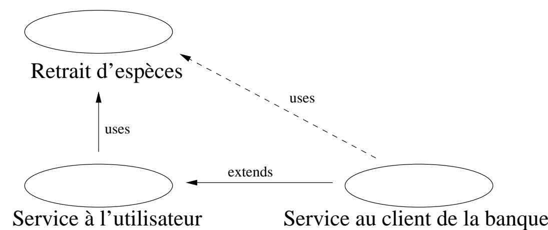

FIG. 3 – Exemple d'extension et d'utilisation

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 opé-

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

rations 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.

4 Le diagramme des classes (vue structurelle)

Le modèle qui va suivre a été particulièrement mis en exergue dans les méthodes orientées objets. Les sections 4.1 à la page 10 à 4.6 à la page 15 regroupent les concepts de base et dans les sections 4.7 à la page 18 à 4.8 à la page 20 présentent des notions plus avancées.

4.1 Introduction au diagramme des classes

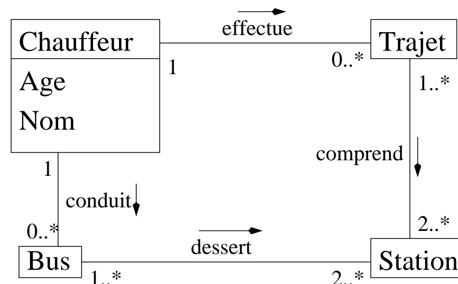

FIG. 4 – Un diagramme des classes

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.

4.2 Les différents niveaux de description

Selon l'activité de l'ingénieur, qu'il s'agisse d'analyse, de conception ou d'implémentation, le niveau de détail avec lequel est représenté le diagramme des classes change énormément. Avant de dessiner un diagramme des classes, il est crucial de savoir à partir duquel des trois points de vues l'on décide de se placer :

– le point de vue de l'analyse, qui en général se doit d'oublier tout aspect de mise en œuvre et, en ce sens, est complètement indépendant du logiciel (on n'y parlera pas de structuration des données : tableaux, pointeurs, listes, ...) ;
– le point de vue de la conception, qui cherche à identifier les interfaces, les types des objets, leur comportement externe et la façon interne de les mettre en œuvre, sans être encore fixé sur un langage ;
– le point de vue de l'implémentation, qui cherche à décrire une classe, ses attributs et ses méthodes en pensant déjà au code qui les implémentera et prend en compte les contraintes matérielles de temps d'exécution, d'architecture, etc.

Dans le cadre d'une analyse, seuls les noms des attributs et les principales méthodes publiques de la classe ont à être mentionnées. Dans le cadre d'une conception et, à plus forte raison, d'une implémentation, la description des classes devra être exhaustive. Mais les différences ne se limitent pas au seul niveau de description. De nombreuses classes spécifiques seront ajoutées lorsque l'on passe de l'analyse à la conception, l'organisation des diagrammes peut évoluer, etc.

4.3 Les diagrammes de packages

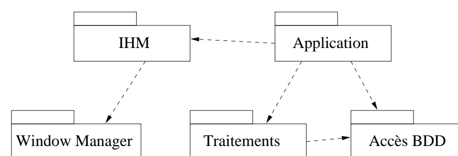

FIG. 5 – Exemple de diagramme de packages

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 ([Sig05a]).

4.4 Description d'une classe

L'ensemble des éléments qui décrivent une classe sont représentés sur la figure 6. 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.

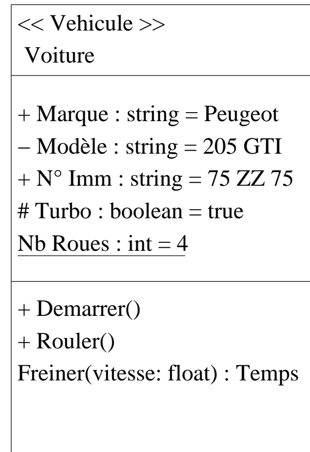

FIG. 6 – Exemple de représentation d'une classe

4.4.1 Les attributs

Pour une classe, un attribut est une forme dégénérée d'association (voir la section 4.6 à la page 15) 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.

4.4.2 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 pré-requis ou un invariant que doit satisfaire l'opération.

La liste des paramètres est de la forme <nom> : <type> = <valeur par défaut>

Il est 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).

4.5 Les interfaces

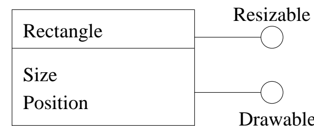

FIG. 7 – Exemple de représentation d'interfaces

La notion d'interface est légèrement variable d'un langage de programmation à l'autre. En C++, on conçoit plutôt l'interface comme la totalité des éléments d'un objet visibles de l'extérieur de cet objet. En pratique, il s'agit des méthodes et attributs publics, c'est-à-dire accessibles par toutes les autres classes de l'application. En JAVA, l'interface est plutôt vue comme une spécification externe des interactions qu'une classe peut avoir avec les autres classes de l'application. Distincte de la liste des méthodes publiques, l'interface est plutôt un « contrat » de la spécification de l'ensemble des traitements que la classe s'engage à effectuer si elle veut prétendre disposer d'une certaine interface. Une classe qui dispose d'une interface doit implémenter toutes les opérations décrites par celle-ci. La notion d'interface proposée par UML, bien que théoriquement indépendante de tout langage, semble davantage se rapprocher du sens donné à cette notion en JAVA. Un exemple de représentation d'interface apparaît sur la figure 7.

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

4.6 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.

4.6.1 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.

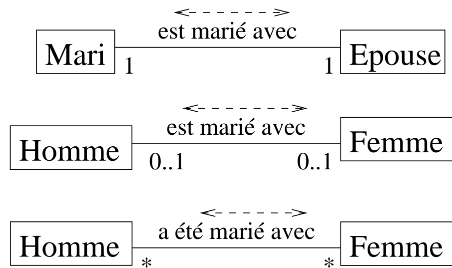

FIG. 9 – Cardinalités : exemples

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

4.6.2 Attributs et classes d'association

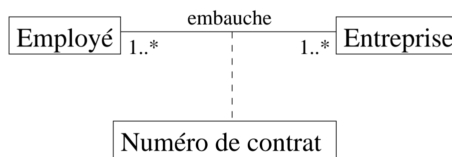

FIG. 10 – Attribut d'association

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.

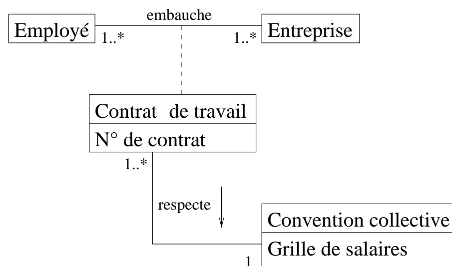

FIG. 11 – Classe d'association

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 n-aires, lorsque les liens sémantiques sont intrinsèquement partagés entre plusieurs objets.

4.6.3 Qualificatifs

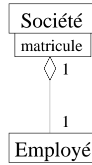

FIG. 12 – Représentation des qualificatifs

Un *qualificatif* est un attribut d'association (ou un ensemble d'attributs) dont les valeurs partitionnent l'ensemble des objets reliés à un objet à travers une association. On le représente comme il apparaît sur la figure 12. Par exemple, le numéro de sécurité sociale permet d'identifier sans ambiguïté tous les bénéficiaires d'une prestation sociale. Ou encore, la conjonction d'une ligne et d'une colonne permet d'identifier toute case sur un échiquier. La cardinalité indiquée est contrainte par l'emploi du qualificatif : une société emploie un grand nombre d'employés, mais un seul par matricule.

4.6.4 Associations et attributs dérivés

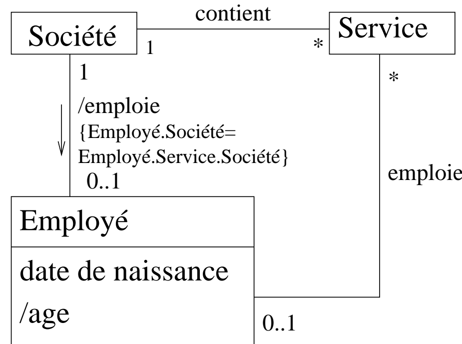

FIG. 13 – Associations et attributs dérivés

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.

4.6.5 Ajout de contraintes et de règles

UML propose d'ajouter des contraintes ou règles d'utilisation, entre accolades. La syntaxe de ces règles est donnée par le langage OCL, contribution d'IBM à UML.

Dans l'idéal, ces règles, opérant sur les classes ou les associations devraient correspondre à des assertions dans le code du programme, c'est-à-dire des tests de validité, pour éviter tout risque de comportement sémantiquement incorrect.

Il y a un certain nombre de contraintes prédéfinies dans le langage (tels `and, or, incomplete, ordered, disjoint` pour les contraintes entre liens), mais on peut aussi écrire ses propres contraintes à l'aide d'expressions algorithmiques.

4.7 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'ourang-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.

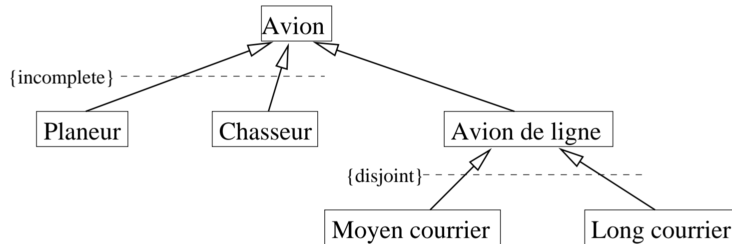

FIG. 14 – Exemple d'arbre de généralisation/spécialisation

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 supportés 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.

4.7.1 Agrégation et composition

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. Par exemple, un train est constitué d'une série de wa-

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

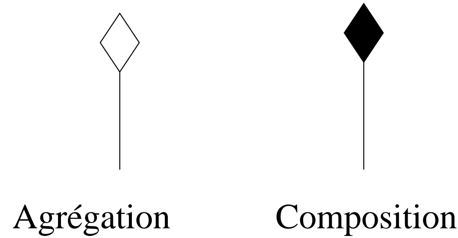

FIG. 15 – Représentations de l'agrégation et la composition

gons, mais ces 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.

4.8 Classes paramétriques

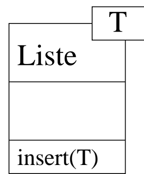

FIG. 16 – Représentation d'une classe paramétrique

Les classes paramétriques se rencontrent quand on veut représenter des collections d'objets dans un langage fortement typé, où la collection est amenée à opérer sur différents types. Afin d'éviter de créer une classe pour toutes les combinaisons collection/type d'objet, on préfère paramétrer toutes les collections par un type formel, sans avoir à décider du type d'objet sur lequel elles seront amenées à porter.

Les conteneurs d'objets sont en général codés sous la forme de classes paramétriques. C'est le cas dans la STL (*Standard Templates Library*), la librairie de classes paramétriques standard développée pour le C++. Une variante de la STL existe en JAVA, mais elle n'est pas représentée à l'aide de classes paramétriques. Cette notion n'est pas supportée en JAVA.

Par exemple, quel que soit le type d'objets que contient une liste, il y a des comportements communs à toutes les listes : renvoyer le premier ou le dernier élément, compter les éléments, insérer ou supprimer un élément, parcourir tous les éléments, etc.

Pour réaliser une seule fois les comportements communs, il suffit de représenter la classe générique *Liste*, dotée de toutes les méthodes et attributs utiles aux travaux à réaliser sur les objets de la collection et de lui associer le type générique « `T` ». À l'intérieur de la classe, « `T` » représente le type générique manipulé par la collection. Sur l'exemple de la figure 16, l'opération `insert()` prend comme paramètre une variable du type générique `T`.

Lorsque l'on souhaite par la suite manipuler un cas particulier de liste, une syntaxe proche du C++ est utilisée en UML, du type `Liste<arete>`

Enfin, si l'on souhaite expliciter tous les cas de listes dont on veut pouvoir se servir, on indique à la manière d'un sous-typage, mais en pointillé, les liens entre la liste mère paramétrique et ses instanciations.

Cette association n'a rien à voir avec un sous-typage, il ne s'agit en aucun cas de rajouter ou de spécialiser des méthodes particulières dans les nouvelles sous-classes, elles sont parfaitement caractérisées par la classe mère et le type T particulier sur lequel elles opèrent. En revanche, les compilateurs se chargent en général de vérifier qu'ont été définis convenablement sur tous les types T utilisés, tous les attributs et les opérations dont `Liste` se sert sur T.

5 Les diagrammes de séquences (vue fonctionnelle)

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.

On peut faire apparaître de nombreuses informations de contrôle le long de la ligne

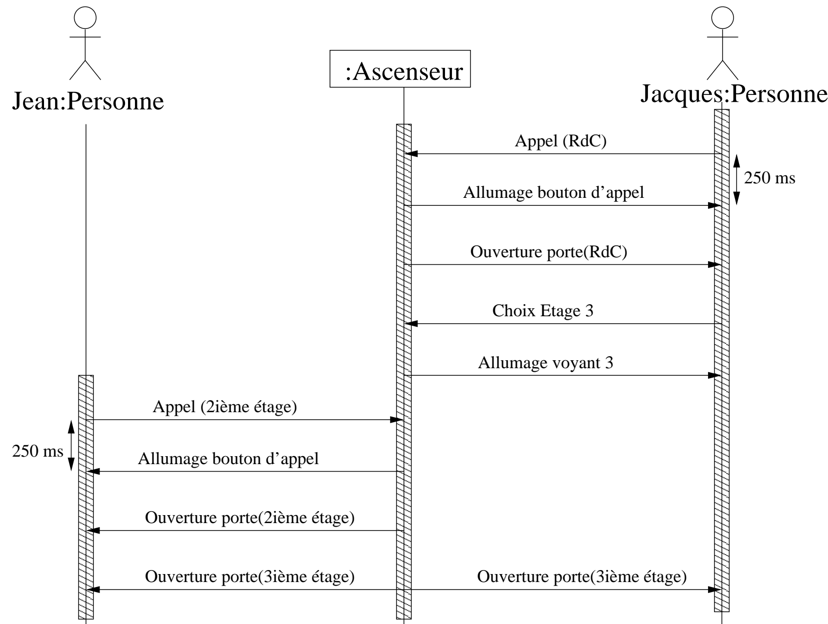

FIG. 17 – Un diagramme de séquences (ici, il s'agit d'un diagramme de séquence d'analyse)

de vie 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.

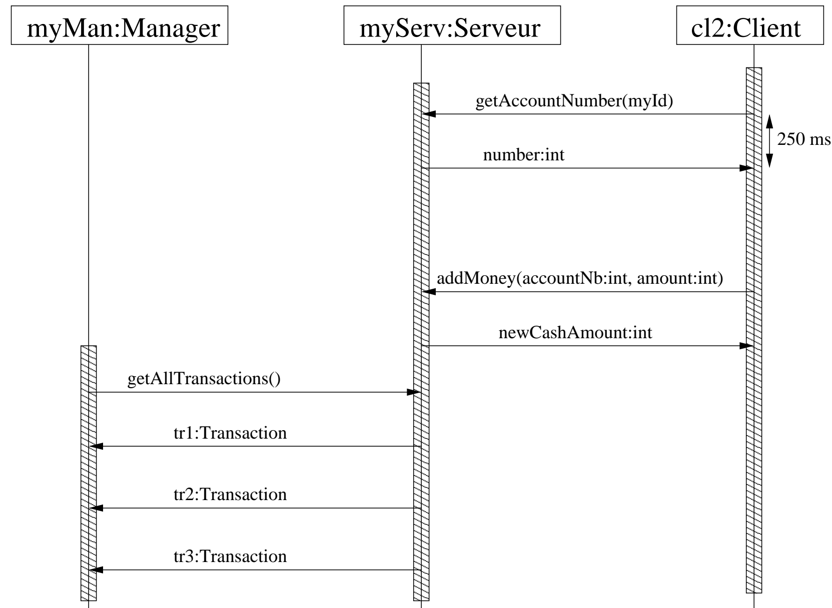

FIG. 18 – Un diagramme de séquences de conception

6 Les diagrammes d'états (vue dynamique)

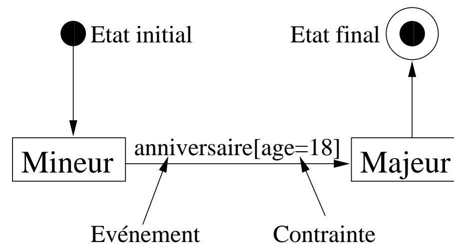

FIG. 19 – Le diagramme d'états d'un citoyen

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 [Har87]. 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).

6.1 Etats et Transitions

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/ ».

6.2 Actions et activités

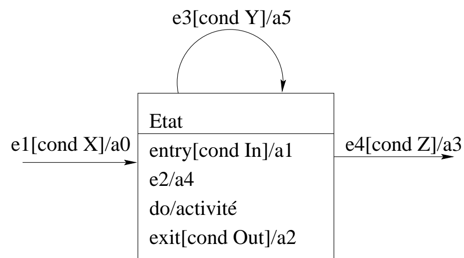

FIG. 20 – Les différents types d'action et d'activités

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.

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

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, comme cela apparaîtra dans la section 6.3 à la page 26.

6.2.1 Exemple : diagramme d'états d'un réveil

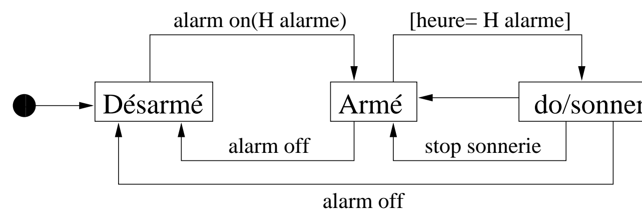

FIG. 21 – Exemple : diagramme d'états d'un réveil

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.

6.2.2 Événements spéciaux

UML offre la possibilité de distinguer deux types d'événements spécifiques, que sont les événements d'entrée (notés entry/) et les événements de sortie (notés exit/).

Toute action reliée à un événement d'entrée sera exécutée à chaque fois qu'on entre dans l'état, par n'importe quelle transition qui y conduit. A l'opposé, l'action liée à un événement de sortie est déclenchée à chaque fois que l'on sort de l'état, quelle que soit la transition incriminée.

En cas d'auto-transition, ou transition propre, on effectue les opérations correspondant à une sortie puis les opérations correspondant à une entrée. Un plantage d'ordinateur constitue un bon exemple de transition propre : vous étiez en train de saisir un texte, l'ordinateur plante (événement), vous le rebootez (action associée) puis vous reprenez votre activité à zéro (le contexte n'a pas été sauvé).

6.3 Ordonnancement

L'ordre d'exécution des actions et activités d'un état donné est fixé de la façon suivante, selon le type d'événement déclencheur :

En entrée On commence par réaliser l'action sur la transition d'entrée, puis l'action d'entrée, puis l'activité associée à l'état. Sur l'exemple de la figure 20, si toutes les conditions sont vérifiées, le déclenchement de l'événement e_1 entraîne la séquence d'actions $a0$, $a1$, puis le lancement de l'activité.

En interne On commence par interrompre l'activité en cours, puis on exécute l'action interne, puis on reprend l'activité. Le contexte de l'activité est sauvé lors de son interruption, en vue de la reprise. Sur l'exemple de la figure 20, si toutes les conditions sont vérifiées, le déclenchement de l'événement e_2 entraîne l'interruption de l'activité, l'exécution de l'action $a4$, puis la reprise de l'activité.

En sortie On commence par interrompre l'activité en cours, puis on exécute l'action de sortie, puis l'action sur la transition de sortie. Cette fois, le contexte de l'activité n'est pas sauvé lors de son interruption. Sur l'exemple de la figure 20, si toutes les conditions sont vérifiées, le déclenchement de l'événement e_4 entraîne l'interruption de l'activité, puis l'exécution des actions $a2$ puis $a3$.

Transition propre On commence par interrompre l'activité en cours, puis on exécute l'action de sortie, puis l'action associée à la transition propre, puis l'action d'entrée, puis l'activité associée à l'état. On note que cette fois l'activité est réinitialisée, puisqu'on est sorti de l'état. Sur l'exemple de la figure 20, si toutes les conditions sont vérifiées, le déclenchement de l'événement e_3 entraîne l'interruption de l'activité, puis l'exécution des actions $a2$, $a5$, puis $a1$, puis le lancement de l'activité.

6.4 Diagrammes hiérarchisés

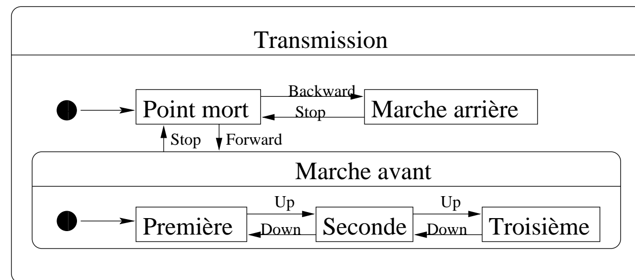

FIG. 22 – Exemple de diagramme hiérarchisé : transmission automatique

De découpage en découpage, un diagramme d'états devient vite illisible. Trop d'états s'enchaînent et leurs liens se multiplient. Les automates hiérarchiques visent à résoudre ce problème. La construction d'automates hiérarchiques permet de retrouver au niveau dynamique des notions structurelles propres aux diagrammes de classes.

On peut raffiner la description d'un état en le décomposant en sous-états, pour faire apparaître un comportement dynamique interne à l'état.

On peut au contraire donner une vue plus synthétique en rassemblant sous un même état un ensemble d'états constituant un comportement dynamique « isolable ».

Dans un cas comme dans l'autre, il faut respecter des contraintes d'homogénéité sur les événements, actions et auto-transitions afin que le comportement externe du sur-état coïncide avec celui du diagramme plus précis qu'il contient. En particulier, quand on entre dans le sur-état, il faut préciser lequel des sous-états est le sous-état initial.

Par ailleurs, le fait de faire apparaître des états emboîtés permet d'associer à l'état englobant les actions et transitions qui sont communes à tous les sous-états. L'intérêt principal de la hiérarchie est donc la factorisation des actions et activités.

En pratique, un sous-état hérite de son sur-état les actions internes et les transitions de sortie, mais il n'hérite pas des transitions en entrée ni des activités.

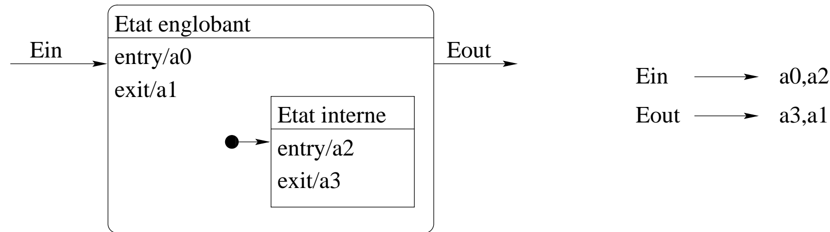

FIG. 23 – Ordonnancement

La réalisation d'automates hiérarchiques pose de nouveaux problèmes d'ordonnancement. La logique qui préside à cet ordonnancement est simple, elle apparaît sur la figure 23.

- Quand on entre dans l'état englobant, on entre dans l'état interne initial.
- On commence par entrer dans l'état le plus englobant pour aller vers le plus interne.
- On commence par sortir de l'état le plus interne avant de sortir de l'état le plus englobant.

6.4.1 Parallélisme et synchronisation

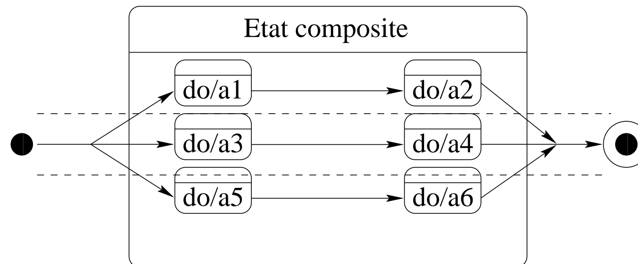

FIG. 24 – Parallélisme et synchronisation

Les automates permettent de modéliser le parallélisme. Un même objet ou une collection d'objets peuvent se trouver à un instant donné dans plusieurs états distincts représentant différents aspects de leur fonctionnement. On peut ainsi associer à un objet plusieurs automates qui fonctionnent en parallèle.

Mais, par ailleurs, on peut vouloir coordonner ces automates ou les synchroniser. Cela se fait à l'aide de flots de contrôle convergents et divergents. Sur la figure 24, les activités A1, A3 et A5 seront lancées en même temps. A2 sera lancée à la fin de A1, A4 à celle de A3 et A6 à celle de A5. Par contre, la transition vers l'état final n'aura

lieu qu'une fois que A2, A4 et A6 seront toutes terminées.

6.5 Le diagramme d'activité (vue dynamique)

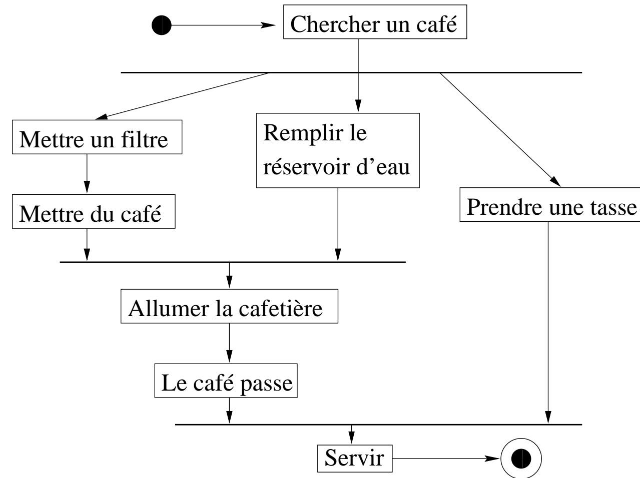

FIG. 25 – Exemple de diagramme d'activité : faire un café

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.

6.6 Extension de UML : les stéréotypes

Avant de conclure, il faut signaler que le langage UML est extensible grâce à la notion de stéréotype. Un certain nombre de stéréotypes sont prédéfinis. C'est le cas par exemple pour la notion d'interface, qui est définie en UML par un stéréotype, mais un utilisateur peut en définir d'autres pour les besoins d'un projet.

Ainsi, rien n'empêche d'étendre le nombre de diagrammes utilisables. Par exemple, pour certaines applications, on peut utiliser un modèle dérivé du Grafcet [url5] en alternative aux modèles des StateCharts [Har87] sur lequel s'appuie le langage UML pour décrire l'aspect dynamique des objets. Toutefois, il ne faut pas abuser de cette

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

possibilité de redéfinition, dans la mesure où elle remet en cause le caractère standard des vues proposées par UML.

6.7 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.

Quand 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 modification d'une classe sur différentes vues, ...) d'autres sont encore du domaine de la recherche, nous ne les aborderons pas.

[5] Sur ces deux points, voir le polycopié complémentaire [Sig05a].

7 Glossaire

Le glossaire ci-dessous définit la majorité des termes utilisés dans UML. Il a été établi par l'OMG [6], organisation internationale qui se charge de standardiser ou normaliser les pratiques de la programmation et de la conception orientée objets. Nous avons choisi de conserver ce glossaire en anglais pour éviter les problèmes de traduction que posent tous les vocabulaires techniques. Nous commençons néanmoins par donner un extrait de glossaire en français.

7.1 Extrait français du 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 formés 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 système 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

[6] *Object Management Group*

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,

7.2 Glossaire complet en anglais

abstract : class A class that cannot be directly instantiated. Contrast :concrete class.

abstraction : The essential characteristics of an entity that distinguish it from all other kinds of entities. An abstraction defines a boundary relative to the perspective of the viewer.

action : The specification of an executable statement that forms an abstraction of a computational procedure. An action typically results in a change in the state of the system, and can be realized by sending a message to an object or modifying a link or a value of an attribute.

action sequence : An expression that resolves to a sequence of actions.

action state : A state that represents the execution of an atomic action, typically the invocation of an operation.

activation : The execution of an action.

active class : A class whose instances are active objects. See : active object.

active object : An object that owns a thread and can initiate control activity. An instance of active class. See : active class, thread.

activity graph : A special case of a state machine that is used to model processes involving one or more classifiers. Contrast : statechart diagram.

actor [class] : A coherent set of roles that users of use cases play when interacting with these use cases. An actor has one role for each use case with which it communicates.

actual parameter : Synonym : argument.

aggregate [class] : A class that represents the whole in an aggregation (whole-part) relationship. See : aggregation.

aggregation : A special form of association that specifies a whole-part relationship between the aggregate (whole) and a component part. See : composition.

analysis : The part of the software development process whose primary purpose is to formulate a model of the problem domain. Analysis focuses what to do, design focuses on how to do it. Contrast : design.

analysis time : Refers to something that occurs during an analysis phase of the software development process. See : design time, modeling time.

architecture : The organizational structure and associated behavior of asystem. An architecture can be recursively decomposed intoparts that interact through interfaces, relationships thatconnect parts, and constraints for assembling parts. Partsthat interact through interfaces include classes, components and subsystems.

argument : A binding for a parameter that resolves to a run-time instance. Synonym : actual parameter. Contrast : parameter.

artifact : A piece of information that is used or produced by a software development process. An artifact can be a model, a description, or software. Synonym : product.

association : The semantic relationship between two or more classifiers that specifies connections among their instances.

association class : A model element that has both association and class properties. An association class can be seen as an association that also has class properties, or as a class that also has association properties.

association end : The endpoint of an association, which connects the association to a classifier.

attribute : A feature within a classifier that describes a range of values that instances of the classifier may hold.

behavior : The observable effects of an operation or event, including its results.

behavioral feature : A dynamic feature of a model element, such as an operation or method.

behavioral model aspect : A model aspect that emphasizes the behavior of the instances in a system, including their methods, collaborations, and state histories.

binary association : An association between two classes. A special case of an n-ary association.

binding : The creation of a model element from a template by supplying arguments for the parameters of the template.

boolean : An enumeration whose values are true and false.

boolean expression : An expression that evaluates to a boolean value.

cardinality : The number of elements in a set. Contrast : multiplicity.

child : In a generalization relationship, the specialization of another element, the parent. See : subclass, subtype. Contrast : parent.

call : An action state that invokes an operation on a classifier.

class : A description of a set of objects that share the same attributes, operations, methods, relationships, and semantics. A class may use a set of interfaces to specify collections of operations it provides to its environment. See : interface.

classifier : A mechanism that describes behavioral and structural features. Classifiers include interfaces, classes, datatypes, and components.

classification : The assignment of an object to a classifier. See dynamic classification, multiple classification and static classification.

class diagram : A diagram that shows a collection of declarative (static) model elements, such as classes, types, and their contents and relationships.

client : A classifier that requests a service from another classifier. Contrast : supplier.

collaboration : The specification of how an operation or classifier, such as a use case, is realized by a set of classifiers and associations playing specific roles used in a specific way. The collaboration defines an interaction. See : interaction.

collaboration : diagram A diagram that shows interactions organized around the structure of a model, using either classifiers and associations or instances and links. Unlike a sequence diagram, a collaboration diagram shows the relationships among the instances. Sequence diagrams and collaboration diagrams express similar information, but show it in different ways. See : sequence diagram.

comment : An annotation attached to an element or a collection of elements. A note has no semantics. Contrast : constraint.

compile time : Refers to something that occurs during the compilation of a software module. See : modeling time, run time.

component : A physical, replaceable part of a system that packages implementation and provides the realization of a set of interfaces. A component represents a physical piece of implementation of a system, including software code (source, binary or executable) or equivalents such as scripts or command files.

component diagram : A diagram that shows the organizations and dependencies among components.

composite [class] : A class that is related to one or more classes by a composition relationship. See : composition.

composite aggregation : Synonym : composition.

composite state : A state that consists of either concurrent (orthogonal) substates or sequential (disjoint)

substates : See : substate.

composition : A form of aggregation association with strong ownership and coincident lifetime as part of the whole. Parts with non-fixed multiplicity may be created after the composite itself, but once created they live and die with it (i.e., they share lifetimes). Such parts can also be explicitly removed before the death of the composite. Composition may be recursive. Synonym : composite aggregation.

concrete class : A class that can be directly instantiated. Contrast : abstract class.

concurrency : The occurrence of two or more activities during the same time interval. Concurrency can be achieved by interleaving or simultaneously executing two or more threads. See : thread.

concurrent substate : A substate that can be held simultaneously with other substates contained in the same composite state. See : composite state. Contrast : disjoint substate.

constraint : A semantic condition or restriction. Certain constraints are predefined in the UML, others may be user defined. Constraints are one of three extensibility mechanisms in UML. See : tagged value, stereotype.

container : 1. An instance that exists to contain other instances, and that provides operations to access or iterate over its contents. (for example, arrays, lists, sets). 2. A component that exists to contain other components.

containment hierarchy : A namespace hierarchy consisting of model elements, and the containment relationships that exist between them. A containment hierarchy forms a graph.

context : A view of a set of related modeling elements for a particular purpose, such as specifying an operation.

datatype : A descriptor of a set of values that lack identity and whose operations do not have side effects. Datatypes include primitive pre-defined types and user-definable types. Pre-defined types include numbers, string and time. User-definable types include enumerations.

defining model [MOF] : The model on which a repository is based. Any number of repositories can have the same defining model.

delegation : The ability of an object to issue a message to another object in response to a message. Delegation can be used as an alternative to inheritance. Contrast : inheritance.

dependency : A relationship between two modeling elements, in which a change to one modeling element (the independent element) will affect the other modeling element (the dependent element).

deployment diagram : A diagram that shows the configuration of run-time processing nodes and the components, processes, and objects that live on them. Components represent run-time manifestations of code units. See : component diagrams.

derived element : A model element that can be computed from another element, but that is shown for clarity or that is included for design purposes even though it adds no semantic information.

design : The part of the software development process whose primary purpose is to decide how the system will be implemented. During design strategic and tactical decisions are made to meet the required functional and quality requirements of a system.

design time : Refers to something that occurs during a design phase of the software development process. See : modeling time. Contrast : analysis time.

development process : A set of partially ordered steps performed for a given purpose during software development, such as constructing models or implementing models.

diagram : A graphical presentation of a collection of model elements, most often rendered as a connected graph of arcs (relationships) and vertices (other model elements). UML supports the following diagrams : class diagram, object diagram, use case diagram, sequence diagram, collaboration diagram, state diagram, activity diagram, component diagram, and deployment diagram.

disjoint substate : A substate that cannot be held simultaneously with other substates contained in the same composite state. See : composite state. Contrast : concurrent substate.

distribution unit : A set of objects or components that are allocated to a process or a processor as a group. A distribution unit can be represented by a run-time composite or an aggregate.

domain : An area of knowledge or activity characterized by a set of concepts and terminology understood by practitioners in that area.

dynamic classification : A semantic variation of generalization in which an object may change its classifier. Contrast : static classification.

element : An atomic constituent of a model.

entry action : An action executed upon entering a state in a state machine regardless of the transition taken to reach that state.

enumeration : A list of named values used as the range of a particular attribute type. For example, RGBColor = {red, green, blue}. Boolean is a predefined enumeration with values from the set {false, true}.

event : The specification of a significant occurrence that has a location in time and space. In the context of state diagrams, an event is an occurrence that can trigger a transition.

exit action : An action executed upon exiting a state in a state machine regardless of the transition taken to exit that state.

export : In the context of packages, to make an element visible outside its enclosing namespace. See : visibility. Contrast : export [OMA], import.

expression : A string that evaluates to a value of a particular type. For example, the expression (7 + 5 * 3) evaluates to a value of type number.

extend : A relationship from an extension use case to a base use case, specifying how the behavior defined for the extension use case augments (subject to conditions specified in the extension) the behavior defined for the base use case. The behavior is inserted at the location defined by the extension point in the base use case. The base use case does not depend on performing the behavior of the extension use case. See extension point, include.

facade : A stereotyped package containing only references to model elements owned by another package. It is used to provide a “public view” of some of the contents of a package.

feature : A property, like operation or attribute, which is encapsulated within a classifier, such as an interface, a class, or a datatype.

final state : A special kind of state signifying that the enclosing composite state or the entire state machine is completed.

fire : To execute a state transition. See : transition.

focus of control : A symbol on a sequence diagram that shows the period of time during which an object is performing an action, either directly or through a subordinate procedure.

formal parameter : Synonym : parameter.

framework : 1. A stereotyped package consisting mainly of patterns. See : pattern. 2. An architectural pattern that provides an extensible template for for applications within a specific domain.

generalizable element : A model element that may participate in a generalization relationship. See : generalization.

generalization : A taxonomic relationship between a more general element and a more specific element. The more specific element is fully consistent with the more general element and contains additional information. An instance of the more specific element may be used where the more general element is allowed. See : inheritance.

guard condition : A condition that must be satisfied in order to enable an associated transition to fire.

implementation : A definition of how something is constructed or computed. For example, a class is an implementation of a type, a method is an implementation of an operation.

implementation inheritance : The inheritance of the implementation of a more specific element. Includes inheritance of the interface. Contrast : interface inheritance.

import : In the context of packages, a dependency that shows the packages whose classes may be referenced within a given package (including packages recursively embedded within it). Contrast : export.

include : A relationship from a base use case to an inclusion use case, specifying how the behavior for the base use case contains the behavior of the inclusion use case. The behavior is included at the location which is defined in the base use case. The base use case depends on performing the behavior of the inclusion use case, but not on its structure (i.e., attributes or operations). See extend.

inheritance : The mechanism by which more specific elements incorporate structure and behavior of more general elements related by behavior. See generalization. instance An entity to which a set of operations can be applied and which has a state that stores the effects of the operations. See : object.

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

interaction : A specification of how stimuli are sent between instances to perform a specific task. The interaction is defined in the context of a collaboration. See collaboration.

interaction diagram : A generic term that applies to several types of diagrams that emphasize object interactions. These include collaboration diagrams and sequence diagrams.

interface : A named set of operations that characterize the behavior of an element.

interface inheritance : The inheritance of the interface of a more specific element. Does not include inheritance of the implementation. Contrast : implementation inheritance.

internal transition : A transition signifying a response to an event without changing the state of an object.

layer : The organization of classifiers or packages at the same level of abstraction. A layer represents a horizontal slice through an architecture, whereas a partition represents a vertical slice. Contrast : partition.

link : A semantic connection among a tuple of objects. An instance of an association. See : association.

link end : An instance of an association end. See : association end.

message : A specification of the conveyance of information from one instance to another, with the expectation that activity will ensue. A message may specify the raising of a signal or thecall of an operation.

metaclass : A class whose instances are classes. Metaclasses are typically used to construct metamodels.

meta-metamodel : A model that defines the language for expressing a metamodel. The relationship between a meta-metamodel and a metamodel is analogous to the relationship between a metamodel and a model.

metamodel : A model that defines the language for expressing a model.

metaobject : A generic term for all metaentities in a metamodeling language. For example, metatypes, metaclasses, metaattributes, and metaassociations.

method : The implementation of an operation. It specifies the algorithm or procedure associated with an operation.

model [MOF] : An abstraction of a physical system, with a certain purpose. See : physical system. Usage note : In the context of the MOF specification, which describes a meta-metamodel, for brevity the meta-metamodel is frequently to as simply the model.

model aspect : A dimension of modeling that emphasizes particular qualities of the metamodel. For example, the structural model aspect emphasizes the structural qualities of the metamodel.

model elaboration : The process of generating a repository type from a published model. Includes the generation of interfaces and implementations which allows repositories to be instantiated and populated based on, and in compliance with, the model elaborated.

model element [MOF] : An element that is an abstraction drawn from the system being modeled. Contrast : view element. In the MOF specification model elements are considered to be metaobjects.

modeling time : Refers to something that occurs during a modeling phase of the software development process. It includes analysis time and design time. Usage note : When discussing object systems, it is often important to distinguish between modeling-time and run-time concerns. See : analysis time, design time. Contrast : run time.

module : A software unit of storage and manipulation. Modules include source code modules, binary code modules, and executable code modules. See : component.

multiple classification : A semantic variation of generalization in which an object may belong directly to more than one classifier. See : static classification, dynamic classification.

multiple inheritance : A semantic variation of generalization in which a type may have more than one supertype. Contrast : single inheritance.

multiplicity : A specification of the range of allowable cardinalities that a set may assume. Multiplicity specifications may be given for roles within associations, parts within composites, repetitions, and other purposes. Essentially a multiplicity is a (possibly infinite) subset of the non-negative integers. Contrast : cardinality.

multi-valued [MOF] : A model element with multiplicity defined whose Multiplicity Type : : upper attribute is set to a number greater than one. The term multi-valued does not pertain to the number of values held by an attribute, parameter, etc. at any point in time. Contrast : single-valued.

n-ary association : An association among three or more classes. Each instance of the association is an n-tuple of values from the respective classes. Contrast : binary association.

name : A string used to identify a model element.

namespace : A part of the model in which the names may be defined and used. Within a namespace, each name has a unique meaning. See : name.

node : A node is classifier that represents a run-time computational resource, which generally has at least a memory and often processing capability. Run-time objects and components may reside on nodes.

object : An entity with a well-defined boundary and identity that encapsulates state and behavior. State is represented by attributes and relationships, behavior is represented by operations, methods, and state machines. An object is an instance of a class. See : class, instance.

object diagram : A diagram that encompasses objects and their relationships at a point in time. An object diagram may be considered a special case of a class diagram or a collaboration diagram. See : class diagram, collaboration diagram.

object flow state : A state in an activity graph that represents the passing of an object from the output of actions in one state to the input of actions in another state.

object lifeline : A line in a sequence diagram that represents the existence of an object over a period of time. See : sequence diagram.

operation : A service that can be requested from an object to effect behavior. An operation has a signature, which may restrict the actual parameters that are possible.

package : A general purpose mechanism for organizing elements into groups. Packages may be nested within other packages.

parameter : The specification of a variable that can be changed, passed, or returned. A parameter may include a name, type, and direction. Parameters are used for operations, messages, and events. Synonyms : formal parameter. Contrast : argument.

parameterized element : The descriptor for a class with one or more unbound parameters. Synonym : template.

parent : In a generalization relationship, the generalization of another element, the child. See : subclass, subtype. Contrast : child.

participate : The connection of a model element to a relationship or to a reified relationship. For example, a class participates in an association, an actor participates in a use case.

partition : 1. activity graphs : A portion of an activity graphs that organizes the responsibilities for actions. See : swimlane. 2. architecture : A set of related classifiers or packages at the same level of abstraction or across layers in a layered architecture. A partition represents a vertical slice through an architecture, whereas a layer represents a horizontal slice. Contrast : layer.

pattern : A template collaboration.

persistent object : An object that exists after the process or thread that created it has ceased to exist.

postcondition : A constraint that must be true at the completion of an operation.

precondition : A constraint that must be true when an operation is invoked.

primitive type : A pre-defined basic datatype without any substructure, such as an integer or a string.

process : 1. A heavyweight unit of concurrency and execution in an operating system. Contrast : thread, which includes heavyweight and lightweight processes. If necessary, an implementation distinction can be made using stereotypes.2. A software development process the steps and guidelines by which to develop a system. 3. To execute an algorithm or otherwise handle something dynamically.

projection : A mapping from a set to a subset of it.

property : A named value denoting a characteristic of an element. A property has semantic impact. Certain properties are predefined in the UML ; others may be user defined. See : tagged value.

pseudo-state : A vertex in a state machine that has the form of a state, but does not behave as a state. Pseudo-states include initial and history vertices.

physical system : 1. The subject of a model. 2. A collection of connected physical units, which can include software, hardware and people, that are organized to accomplish a specific purpose. A physical system can be described by one or more models, possibly from different viewpoints. Contrast : system.

published model [MOF] : A model which has been frozen, and becomes available for instantiating repositories and for the support in defining other models. A frozen modelŠs model elements cannot be changed.

qualifier : An association attribute or tuple of attributes whose values partition the set of objects related to an object across an association.

receive [a message] : The handling of a stimulus passed from a sender instance. See : sender, receiver.

receiver [object] : The object handling a stimulus passed from a sender object. Contrast : sender.

reception : A declaration that a classifier is prepared to react to the receipt of a signal.

reference : 1. A denotation of a model element. 2. A named slot within a classifier that facilitates navigation to other classifiers. Synonym : pointer.

refinement : A relationship that represents a fuller specification of something that has already been specified at a certain level of detail. For example, a design class is a refinement of an analysis class.

run time : The period of time during which a computer program executes. Contrast : modeling time.

relationship : A semantic connection among model elements. Examples of relationships include associations and generalizations.

repository : A facility for storing object models, interfaces, and implementations.

requirement : A desired feature, property, or behavior of a system.

responsibility : A contract or obligation of a classifier.

reuse : The use of a pre-existing artifact.

role : The named specific behavior of an entity participating in a particular context. A role may be static (e.g., an association end) or dynamic (e.g., a collaboration role).

scenario : A specific sequence of actions that illustrates behaviors. A scenario may be used to illustrate an interaction or the execution of a use case instance. See : interaction.

schema [MOF] : In the context of the MOF, a schema is analogous to a package which is a container of model elements. Schema corresponds to an MOF package. Contrast : metamodel, package.

semantic variation point : A point of variation in the semantics of a metamodel. It provides an intentional degree of freedom for the interpretation of the metamodel semantics.

send [a message] : The passing of a stimulus from a sender instance to a receiver instance. See : sender, receiver.

sender [object] : The object passing a stimulus to a receiver object. Contrast : receiver.

sequence diagram : A diagram that shows object interactions arranged in time sequence. In particular, it shows the objects participating in the interaction and the sequence of messages exchanged. Unlike a collaboration diagram, a sequence diagram includes time sequences but does not include object relationships. A sequence diagram can exist in a generic form (describes all possible scenarios) and in an instance form (describes one actual scenario). Sequence diagrams and collaboration diagrams express similar information, but show it in different ways. See : collaboration diagram.

signal : The specification of an asynchronous stimulus communicated between instances. Signals may have parameters.

signature : The name and parameters of a behavioral feature. A signature may include an optional returned parameter.

single inheritance : A semantic variation of generalization in which a type may have only one supertype. Contrast : multiple inheritance.

single valued [MOF] : A model element with multiplicity defined is single valued when its Multiplicity Type : : upper attribute is set to one. The term single-valued does not pertain to the number of values held by an attribute, parameter, etc., at any point in time, since a single-valued attribute (for instance, with a multiplicity lower bound of zero) may have no value. Contrast : multi-valued.

specification : A declarative description of what something is or does. Contrast : implementation.

state : A condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for some event. Contrast : state [OMA].

statechart diagram : A diagram that shows a state machine. See : state machine.

state machine : A behavior that specifies the sequences of states that an object or an interaction goes through during its life in response to events, together with its responses and actions.

static classification : A semantic variation of generalization in which an object may not change classifier. Contrast : dynamic classification.

stereotype : A new type of modeling element that extends the semantics of the metamodel. Stereotypes must be based on certain existing types or classes in the metamodel. Stereotypes may extend the semantics, but not the structure of pre-existing types and classes. Certain stereotypes are predefined in the UML, others may be user defined. Stereotypes are one of three extensibility mechanisms in UML. See : constraint, tagged value.

stimulus : The passing of information from one instance to another, such as raising a signal or invoking an operation. The receipt of a signal is normally considered an event. See : message.

string : A sequence of text characters. The details of string representation depend on implementation, and may include character sets that support international characters and graphics.

structural feature : A static feature of a model element, such as an attribute.

structural model aspect : A model aspect that emphasizes the structure of the objects in a system, including their types, classes, relationships, attributes, and operations.

subactivity state : A state in an activity graph that represents the execution of a non-atomic sequence of steps that has some duration.

subclass : In a generalization relationship, the specialization of another class ; the superclass. See : generalization. Contrast : superclass.

submachine state : A state in a state machine which is equivalent to a composite state but its contents is described by another state machine.

substate : A state that is part of a composite state. See : concurrent state, disjoint state.

subpackage : A package that is contained in another package.

subsystem : A grouping of model elements that represents a behavioural unit in a physical system. A subsystem offers interfaces and has operations. In addition, the model elements of a subsystem can be partitioned into specification and realization elements. See package. See : physical system.

subtype : In a generalization relationship, the specialization of another type ; the supertype. See : generalization. Contrast : supertype.

superclass : In a generalization relationship, the generalization of another class ; the subclass. See : generalization. Contrast : subclass.

supertype : In a generalization relationship, the generalization of another type ; the subtype. See : generalization. Contrast : subtype.

supplier : A classifier that provides services that can be invoked by others. Contrast : client.

swimlane : A partition on a activity diagram for organizing the responsibilities for actions. Swimlanes typically correspond to organizational units in a business model. See : partition.

synch state : A vertex in a state machine used for synchronizing the concurrent regions of a state machine.

system : A top-level subsystem in a model. Contrast : physical system.

tagged value : The explicit definition of a property as a name-value pair. In a tagged value, the name is referred as the tag. Certain tags are predefined in the UML ; others may be user defined. Tagged values are one of three extensibility mechanisms in UML. See : constraint, stereotype.

template : Synonym : parameterized element.

thread [of control] : A single path of execution through a program, a dynamic model, or some other representation of control flow. Also, a stereotype for the implementation of an active object as lightweight process. See process.

time event : An event that denotes the time elapsed since the current state was entered. See : event.

time expression : An expression that resolves to an absolute or relative value of time.

timing mark : A denotation for the time at which an event or message occurs. Timing marks may be used in constraints.

top level : A stereotype of package denoting the top-most package in a containment hierarchy. The topLevel stereotype defines the outer limit for looking up names, as namespaces "see" outwards. For example, opLevel subsystem represents the top of the subsystem containment hierarchy.

trace : A dependency that indicates a historical or process relationship between two elements that represent the same concept without specific rules for deriving one from the other.

transient object : An object that exists only during the execution of the process or thread that created it.

transition : A relationship between two states indicating that an object in the first state will perform certain specified actions and enter the second state when a specified event occurs and specified conditions are satisfied. On such a change of state, the transition is said to fire.

type : A stereotype of class that is used to specify a domain of instances (objects) together with the operations applicable to the objects. A type may not contain any methods. See : class, instance. Contrast : interface.

type expression : An expression that evaluates to a reference to one or more types.

uninterpreted : A placeholder for a type or types whose implementation is not specified by the UML. Every uninterpreted value has a corresponding string representation. See : any [CORBA].

usage : A dependency in which one element (the client) requires the presence of another element (the supplier) for its correct functioning or implementation.

use case [class] : The specification of a sequence of actions, including variants, that a system (or other entity) can perform, interacting with actors of the system. See : use case instances.

use case diagram : A diagram that shows the relationships among actors and use cases within a system.

use case instance : The performance of a sequence of actions being specified in a use case. An instance of a use case. See : use case class.

use case model : A model that describes the functional requirements of a system in terms of use cases.

utility : A stereotype that groups global variables and procedures in the form of a class declaration. The utility attributes and operations become global variables and global procedures, respectively. A utility is not a fundamental modelling construct, but a programming convenience.

value : An element of a type domain.

vertex : A source or a target for a transition in a state machine. A vertex can be either a state or a pseudo-state. See : state, pseudo-state.

view : A projection of a model, which is seen from a given perspective or vantage point and omits entities that are not relevant to this perspective.

view element : A view element is a textual and/or graphical projection of a collection of model elements.

view projection : A projection of model elements onto view elements. A view projection provides a location and a style for each view element.

visibility : An enumeration whose value (public, protected, or private) denotes how the model element to which it refers may be seen outside its enclosing namespace.

8 NETographie (dernière mise à jour : 2001-2002)

8.1 Programmation objets et UML

url1 - `http ://www.csioo.com/cetusfr/software.html` : carrefour Cetus : Orienté Objets ; excellent, plus de 8000 liens.

url2 - `http ://www.cyber-espace.com/ronald/` : espace objet francophone.

url3 - `http ://www.rational.com/UML/resources.html` : site de RATIONAL (principal acteur de UML)

url4 - `http ://www.omg.org` : *Object Management Group*

url5 - `http ://www.lurpa.ens-cachan.fr/grafcet/grafcet_fr.html` : *Tout sur le* Grafcet

url6 - `http ://www.ensta.fr/ osigaud/in204/index.html` : *Le site web du cours*

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

8.2 Les patterns (ou patrons)

- http ://www.csioo.com/cetusfr/oo_patterns.html : liste de références concernant les patterns (ou patrons)
- http ://st-www.cs.uiuc.edu/users/patterns/patterns.html : page d'entrée principale pour les patterns

Références

[Boo94] G. Booch. *Object-Oriented Software Analysis and Design with Applications*. Benjamin Cummings, 1994.

[GHJV96] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns : catalogue de modèles de conception réutilisables*. ITP, France, 1996.

[Har87] D. Harel. Statecharts : A visual formalism for complex systems. *Science of Computer Programming*, 8, 1987.

[JBR97a] I. Jacobson, G. Booch, and J. Rumbaugh. *The Objectory Software Development Process*. Addison Wesley, 1997.

[JBR97b] I. Jacobson, G. Booch, and J. Rumbaugh. *Unified Modeling Language Reference Manual*. Addison Wesley, 1997.

[JBR97c] I. Jacobson, G. Booch, and J. Rumbaugh. *Unified Modeling Language User Guide*. Addison Wesley, 1997.

[JCJO92] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. *Object-Oriented Software Engineering : A Use case Driven Approach*. Addison Wesley, 1992.

[Lor97] M. Lorenz. *Object-Oriented Software Development*. Prentice Hall, 1997.

[Mey97] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997.

[Rum96] J. Rumbaugh. *Modélisation et conception orientées objets*. Masson, France, 1996.

[Sig05a] O. Sigaud. *Introduction à la conduite de projet reposant sur un langage orienté objets*. support du cours « Génie logiciel et programmation orientée objets » de l'ENSTA, 2005.

[Sig05b] O. Sigaud. *Introduction à la programmation orientée objets avec Java*. support du cours « Génie logiciel et programmation orientée objets » de l'ENSTA, 2005.