

# I Contexte et problématique

Par Jean-Marc Jézéquel, Mariano Belaunde, Jean Bézivin, Sébastien Gérard, et Pierre-Alain Muller

## I.1- Introduction

### I.1.1- Situation dans le domaine du génie logiciel

Pris dans un sens large, le génie logiciel englobe les «règles de l'art» de l'ingénierie de la réalisation des systèmes manipulant de l'information, qu'on appelle encore systèmes à logiciel prépondérant. Autrefois cantonné à des domaines limités comme l'informatique de gestion, le calcul scientifique, ou encore le contrôle d'engins high-tech, le logiciel est aujourd'hui omniprésent. Il y a, par exemple, plus de logiciel dans une automobile de moyenne gamme récente ou dans le moindre téléphone portable que dans une capsule Apollo.

Pour les systèmes de plus en plus complexes qu'on cherche à construire, il s'agit de trouver un compromis entre un système parfaitement bien conçu (qui pourrait demander un temps infini pour être construit) et un système trop vite fait (qu'il serait difficile de mettre au point et de maintenir), en conciliant trois forces largement antagonistes : les délais, les coûts, et la qualité. Les délais doivent être tenus pour arriver sur le marché au moment opportun, la qualité doit être à un niveau compatible avec la criticité du système, et le coût doit rester en rapport avec les moyens financiers disponibles. Il ne s'agit donc pas de produire un « bon » système dans l'absolu, mais bien de produire, en suffisamment peu de temps, un système suffisamment bon et suffisamment bon marché compte tenu du contexte dans lequel il sera utilisé.

La maîtrise des processus d'élaboration du logiciel se décline en trois axes :

- **Programming-in-the-Small**, c'est à dire la maîtrise de la construction d'éléments de logiciels corrects.
- **Programming-in-the-Large**, c'est à dire la maîtrise de la complexité structurelle due à la taille des objets construits. Bien évidemment, cette complexité n'est pas le propre de l'informatique : on retrouvera le même type de problèmes (et de solutions) partout où il faut gérer de grands projets, par exemple le génie civil.
- **Programming-in-the-Variable**, c'est à dire la maîtrise de la malléabilité du logiciel (l'aspect *soft* du *software*). Un système logiciel n'est en effet pas rigide, il évolue dans le temps (maintenance évolutive) et dans l'espace (variantes régionales, fonctionnelles ou encore de plateforme cibles: c'est la notion de lignes de produits).

C'est principalement sur ce troisième axe que se focalise cet ouvrage.

### I.1.2- Variabilité du logiciel

Même si le logiciel partage bien des problèmes avec d'autres domaines dans la gestion de la complexité structurelle, il a en revanche des spécificités quant à sa malléabilité. C'est d'abord un produit abstrait (papier, fichier, programme) qu'il est matériellement très simple de faire évoluer. Déplacer quelques lignes de programme est beaucoup plus facile que de déplacer un pilier de pont ou d'ajouter un radiateur dans une maison ; par contre l'impact est très certainement moins bien maîtrisé ! D'autre part, on rencontre fréquemment une certaine difficulté à spécifier les besoins. L'automatisation apportée par l'informatique force à modéliser/comprendre des processus de fonctionnement et d'organisation de systèmes complexes (avion, entreprise, etc.) dont il est très difficile d'obtenir une vision globale (complète) et cohérente. L'automatisation fait en outre souvent apparaître de nouveaux besoins, de nouvelles possibilités et des défauts qui n'avaient pu être envisagés avant

l'automatisation. Les coûts élevés des logiciels en font des investissements lourds qui ne sont supportables que si leur durée de vie est longue. Mais un logiciel à longue durée de vie doit évoluer pour conserver son utilité. Au-delà de la problématique classique de la maintenance corrective et évolutive, on assiste aujourd'hui à une accélération de l'introduction de nouvelles plateformes technologiques (EJB, .Net, Web Services, etc.) qui impose de lourds efforts d'adaptation aux logiciels qui souhaitent en tirer parti.

Le principal problème du génie logiciel ne se pose plus en termes de « donnez-moi une spécification immuable et je produis une implantation de qualité » mais plutôt en « comment réaliser une implantation de qualité avec des spécifications continuellement mouvantes ». De plus, il arrive de plus en plus fréquemment de ne plus avoir à produire un produit donné à un moment donné, mais simultanément toute une gamme (ou famille) de produits pour prendre en compte des variations de fonctionnalités ou d'environnements. Cette idée, qui a révolutionné l'industrie automobile dans les deux décennies précédentes, est aussi de plus en plus largement adoptée dans l'industrie de l'informatique des télécoms. À cet égard on peut citer des constructeurs emblématiques tels que Microsoft, qui adaptent certains de leurs logiciels à des dizaines de plates-formes différentes ainsi qu'à des centaines d'environnements culturels ; ou encore Nokia qui supporte simultanément près d'une dizaine de milliers de versions différentes de ses téléphones mobiles. La maîtrise de cette double variabilité spatiale et temporelle est donc devenue un des enjeux majeurs de l'industrie du logiciel.

## **I.2- De l'objet aux modèles**

### **I.2.1- Le principe unificateur simple des années 80 : tout est objet**

Issue de deux décennies de recherche sur les langages de programmation, un principe unificateur visant à simplifier le développement de logiciels remporte un succès grandissant au cours des années 80 : il s'agit de considérer la notion d'objet comme la notion centrale autour de laquelle s'articule le développement du logiciel. Ce principe « tout est objet » est supporté avec plus de moins de jusqu'au boutisme dans des langages comme Smalltalk, Eiffel, ou dans une moindre mesure C++ puis Java ou C#. Il a aussi donné naissance aux premières méthodes d'analyse et de conception par objets, comme OMT ou Booch.

Il est incontestable que l'adoption plus ou moins stricte de ce principe a permis de franchir un seuil dans la complexité des problèmes traités par logiciel. Cependant il est apparu rapidement que ce principe n'était pas suffisant en lui-même. Par exemple, dès le début des années 90 apparaît l'idée que si les objets doivent être des unités les plus simples et les plus modulaires possible, alors la complexité est souvent repoussée dans la manière dont les objets interagissent et collaborent. T. Reenskaug montre qu'il peut être utile de représenter cette collaboration afin de pouvoir raisonner dessus. Un peu plus tard, des chercheurs comme J. Coplien ou E. Gamma (et d'autres) montrent l'importance de la notion de « design patterns » qui, au delà d'offrir des schémas de collaborations paramétrés réutilisables, permettent de capitaliser un savoir-faire de conception par objet, et sont devenus en quelques années un des outils de travail les plus indispensables aux développeurs de logiciels.

À partir du début des années 90 apparaît aussi le besoin de ne plus seulement voir les objets comme des unités modulaires adaptées au *développement* logiciel, mais aussi comme des unités modulaires explicitant contractuellement leurs dépendances pour le *déploiement* du

logiciel. C'est la notion de composant logiciel telle que définie par C. Syspersky dans son ouvrage séminal «au-delà des objets : les composants».

Ces composants, en particulier lorsqu'ils sont disponibles sur étagères, permettent de construire, de déployer et de faire évoluer des applications complexes sans contrôle centralisé du développement logiciel. Mais pour permettre à des composants produits par des organisations différentes de coopérer, on a besoin d'une nouvelle couche d'intermédiation entre les systèmes d'exploitation et les composants eux-mêmes : c'est la notion d'intergiciel (middleware en anglais). Au milieu des années 90 l'OMG (Object Management Group), un organisme de normalisation international rassemblant les principaux industriels du domaine des technologies de l'information a cru possible de pouvoir standardiser un seul et unique intergiciel appelé CORBA. Outre les limitations initiales de CORBA qui empêchaient une réelle interopérabilité entre composants issus de sources différentes, c'était sans compter sur Microsoft qui, en faisant évoluer sa plate-forme de composant COM vers DCOM, offrait une solution concurrente, mais pas plus universelle que la précédente. Aussi vers la fin des années 90 voit-on apparaître de nombreuses nouvelles plates-formes logicielles comme les EJB, .Net ou encore les Web Services. Loin de se stabiliser, ce domaine voit au contraire une accélération de l'introduction de ces nouvelles plates-formes, ce qui entraîne des coûts vertigineux de portages d'applications.

Parallèlement à cette évolution, d'autres chercheurs comme G. Kiczales mettaient en évidence le fait que le type de modularité qu'on trouvait dans les objets et les composants était satisfaisant pour gérer beaucoup d'aspects fonctionnels, mais que par contre la gestion d'aspects non fonctionnels comme la distribution, la persistance, la tolérance aux défaillances etc. étaient en général très mal modularisés dans les applications construites par objets. Pour y remédier on a vu apparaître la notion de *programmation par aspect* dont le principe est justement de décrire de manière modulaire ces aspects transversaux puis dans un second temps de les *tisser* dans la trame d'une application orientée objet.

Ainsi donc, en partant au début des années 80 du concept unique et simplificateur « tout est objet », on se retrouve tout de même à la fin des années 90 avec une vaste variété de concepts qui s'appuient tous plus ou moins sur la notion d'objet, mais qui, s'ils semblent indispensables au développement d'applications de grande envergure, n'en rendent pas moins extrêmement complexe la maîtrise globale du développement de tels logiciels.

## **I.2.2- Modéliser pour maîtriser la complexité**

Comme pour les autres sciences, il faut donc recourir à la modélisation pour essayer de maîtriser cette complexité. Selon J. Rothenberg,

*Modeling, in the broadest sense, is the cost-effective use of something in place of something else for some cognitive purpose. It allows us to use something that is simpler, safer or cheaper than reality instead of reality for some purpose.*

*A model represents reality for the given purpose; the model is an abstraction of reality in the sense that it cannot represent all aspects of reality. This allows us to deal with the world in a simplified manner, avoiding the complexity, danger and irreversibility of reality.*

Un modèle fournit une abstraction d'un système physique qui permet aux ingénieurs de raisonner sur ce système en ignorant certains détails non pertinents. Toutes les formes d'ingénierie s'appuient sur la notion de modèle qui est essentielle pour la compréhension des

systèmes complexes du monde réel, et utilisent ces modèles pour prédire la qualité du système, raisonner à propos de propriétés spécifiques quand certains aspects du système sont modifiés, et communiquer les caractéristiques clés du système aux différentes parties prenantes. Le modèle peut être développé en tant que précurseur à la réalisation d'un système physique, ou peut être dérivé de systèmes existants ou d'un système en développement.

Comme il y a de nombreux aspects différents d'un système qui peuvent être intéressants pour telle ou telle activité, différents concepts et notations de modélisation doivent être utilisés pour mettre en évidence une ou plusieurs perspectives particulières, qu'on appelle souvent vues. De plus, il est parfois utile de compléter des modèles avec des règles de cohérence entre vues, voire même des règles permettant de transformer une représentation en une autre (par exemple une transformation de Fourier qui permet de passer du domaine temporel au domaine des fréquences en traitement du signal). Un modèle a donc trois caractéristiques principales :

1. Il y a (où il y aura) un original ;
2. Le modèle est une abstraction de l'original, qui ignore détails non pertinents ;
3. Un modèle est construit dans un objectif particulier, ce qui en général permet de déterminer quels détails peuvent être ignorés.

## **1.2.3- Un bref aperçu de UML, le langage de modélisation unifié**

### **1.2.3.1- Origine d'UML**

Une fois admise l'idée que la modélisation joue un rôle crucial dans le développement de logiciels, il faut encore disposer d'un langage suffisamment abstrait pour être indépendant de telle ou telle plate-forme matérielle ou logicielle, tout en étant suffisamment précis pour permettre de capturer aussi précisément que nécessaire les aspects fondamentaux d'un problème. Depuis les années soixante-dix, où est apparue l'idée qu'il était nécessaire de modéliser avant de programmer, de très nombreux langages et méthodes ont été proposés pour répondre à ce problème. Schématiquement, on voit apparaître trois générations.

Vers la fin des années 70, une standardisation autour d'approches centrées chacune sur une des dimensions de l'informatique :

- les fonctions à assurer, avec par exemple la méthode SADT;
- la structure statique, avec par exemple les diagrammes entité-relations (popularisés en France par la méthode Merise);
- le comportement dynamique, avec par exemple les machines à états.

Vers la fin des années quatre-vingt, les limitations de ces approches monodimensionnelles deviennent patentées. D'une part, l'évolution continue des services demandés à un logiciel donné et la complexité croissante de son interaction avec le monde extérieur rendent illusoire l'idée même de cahier des charges figé qui était la base de ces approches. Il n'y a plus de spécification à partir de laquelle on pourrait dériver une réalisation quasi-complète. D'autre part, lorsqu'on développe de grands systèmes, on ne peut plus simplement se limiter à l'une ou l'autre de ces dimensions, mais on doit prendre en compte les trois simultanément. De plus, il apparaît une quatrième dimension due à la décentralisation des logiciels grâce aux réseaux.

Pour répondre à ces défis, J. Rumbaugh et ses collègues de la société General Electric proposent alors la méthode OMT (Object Modelling Technique), fondée sur une approche par objets mais combinant autour de celle-ci les approches existantes pour les trois dimensions : services, architecture statique et comportement dynamique. Ceci est rendu possible par l'apparition à cette époque des premiers langages à objets réellement opérationnels dans un contexte industriel (Eiffel, C++, etc.). Beaucoup d'autres propositions allant dans le même sens sont faites à la même époque. G. Booch fait évoluer l'entreprise Rational Software du rôle de fournisseur de composants logiciels pour Ada au rôle de fabricant d'ateliers de génie logiciel. I. Jacobson formalise dans un livre ses trente ans d'expérience dans le développement d'applications par objets chez Ericsson (il avait commencé dès la fin des années 60, en codant en assembleur des conceptions par objets). Ceci conduit au début des années quatre-vingt-dix à un véritable foisonnement de méthodes d'analyse et de conception par objets.

Devant le succès de ces approches dans l'industrie, une très forte pression se développe pour leur standardisation. Les premières tentatives ayant échouées, c'est finalement le marché qui fait sa loi, quand J. Rumbaugh et I. Jacobson rejoignent G. Booch chez Rational Software. Tous trois s'engagent dans un processus de convergence de leurs méthodes qui conduira en 1997 à la standardisation par l'OMG du langage de modélisation unifiée UML (Unified Modelling Language). Celui-ci s'imposera alors très rapidement dans l'industrie. Dès l'année 2001, selon l'OMG, près de 90% des projets logiciels dans le monde l'utilisent à l'une ou l'autre étape du développement. Cette rapidité d'adoption n'est pas très surprenante car UML ne fait finalement qu'une synthèse (ou du moins tente de le faire) des meilleures pratiques existantes pour le développement par objets de logiciels. Grâce à cette acceptation rapide, de nombreux retours d'expérience ont été possibles, ce qui a conduit l'OMG à préparer une version 2.0 de UML en 2003.

### **I.2.3.2- Modéliser avec UML**

Il faut noter qu'UML n'est pas une méthode, c'est-à-dire un guide montrant comment en partant d'une expression des besoins on peut arriver à un logiciel validé. UML est simplement un langage permettant de décrire les différents artefacts produits lors d'un développement de logiciels. Ce développement est usuellement conduit avec une approche itérative, incrémentale, et pilotée par les cas d'utilisation (scénarios d'emploi, rôles assumés par les utilisateurs) : modèle d'expression des besoins, modèle d'analyse (construction d'un modèle idéal du monde), modèle de conception (passage du modèle idéal au monde réel), modèle d'implantation, modèle de tests pour la validation (voir chapitre suivant). Bien sûr, en fonction de l'activité menée lors du développement de logiciels, on n'utilisera pas UML exactement de la même manière. Par exemple le fait qu'un attribut soit privé pourra tout à fait avoir sa place dans un modèle d'implantation, voire de conception détaillée, mais pas dans un modèle d'analyse qui est censé abstraire tout détail relatif à l'implantation.

Pour présenter UML 2.0, il est commode de distinguer quatre grandes parties : l'infrastructure, la superstructure, le langage de contrainte OCL et le format d'échange des diagrammes.

L'infrastructure correspond à une factorisation des concepts objets fondamentaux, et doit être vue comme un ensemble d'éléments de modélisation à partir desquels il est pratique de construire de nouveaux langages de modélisation.

La superstructure UML est un exemple d'utilisation de l'infrastructure, et contient l'ensemble des éléments de modélisation destinés à être manipulés par les utilisateurs du langage. C'est la superstructure qui forme ce qui est communément appelé UML.

Le langage de contrainte OCL (Object Constraint Language) permet l'expression de contraintes, de règles et de requêtes sur un modèle et garantit la précision et la cohérence des modèles.

Le format d'échange des diagrammes complète la norme XMI dans le but d'échanger également les diagrammes. Au-delà des seules instances d'éléments de modélisation, ce format permet également de transférer les diagrammes d'un outil à un autre, assurant ainsi une réelle indépendance des modèles vis-à-vis des outils de modélisation.

### I.2.3.3- Structure du langage

UML est une famille de langages, composée d'un langage central étendu par un ensemble de sous-langages, définis en trois niveaux de conformité.

La notion de conformité avec UML se définit dans les termes des abstractions contenues par les différents paquetages qui composent le langage. Au minimum, une implémentation doit réaliser le niveau de base, qui contient les notions fondamentales de modélisation objet, regroupées dans un paquetage noyau dénommé *Kernel*. Au-delà du noyau, la conformité s'apprécie paquetage par paquetage, avec pour chaque paquetage les niveaux intermédiaire et complet, et en dernier lieu l'échange des éléments et diagrammes au moyen du format XMI.

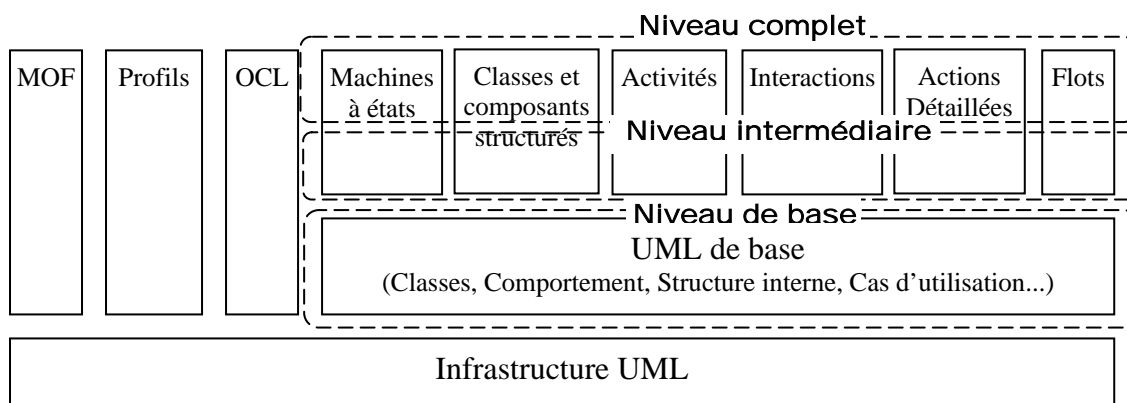
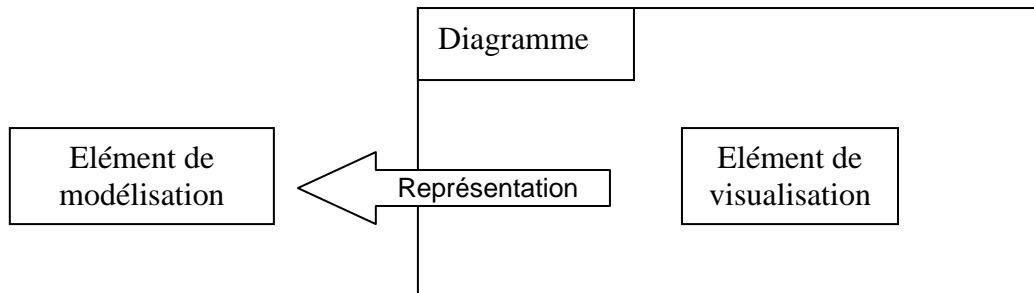


Figure 1 : Structure du langage UML et niveaux de conformité

UML propose des éléments de modélisation et de visualisation pour représenter les différents aspects des systèmes logiciels. Les éléments de modélisation sont généralement manipulés au travers de représentations graphiques (les diagrammes UML), qui contiennent les éléments de visualisation qui correspondent aux éléments de modélisation.

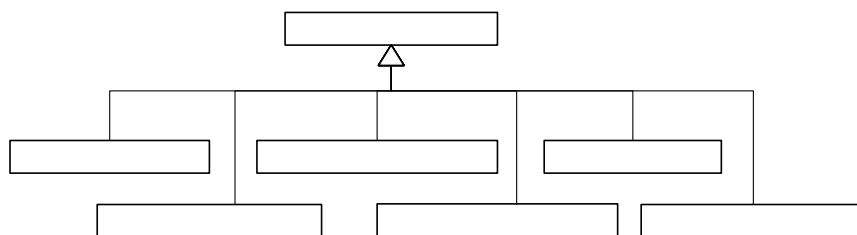


**Figure 2 : Visualisation des éléments de modélisation au sein de diagrammes**

UML définit treize types de diagrammes, et permet en outre mélanger différents types de diagrammes, par exemple pour combiner des aspects structurels et comportementaux au sein d'une même représentation.

#### **I.2.3.4- Modélisation de la structure**

Les diagrammes de structure montrent la structure statique des objets qui participent à un système, dans une spécification indépendante du temps. Les éléments d'un diagramme de structure représentent les concepts pertinents d'une application, tant abstraits, que réels ou pour la réalisation.



**Figure 3 : UML définit 6 types de diagrammes structurels**

- Les diagrammes de classes représentent les classes et leurs relations. UML définit différents types de relations afin de modéliser finement les notions d'association, de composition et de classification.
- Les diagrammes d'objets visualisent des cas particuliers, dans lesquels des objets sont liés à d'autres objets, généralement dans le cadre d'une interaction.
- Les diagrammes de composites viennent combler un manque entre les diagrammes de classes et les diagrammes d'objets. Ces diagrammes permettent notamment de spécifier les structures internes.
- Les diagrammes de composants expriment les relations entre les éléments de réalisation, comme les composants, leurs interfaces et leurs connexions.
- Les diagrammes de déploiement représentent les environnements d'exécution, avec les cibles de déploiement, les dispositifs et les chemins de communication. Il est également possible de montrer comment des instances se répartissent sur les différents nœuds.

- Les diagrammes de paquetage apportent une image de haut niveau de la structure des modèles. Les paquetages servent à organiser les modèles. Ils peuvent contenir et référencer d'autres paquetages. Chaque paquetage peut être utilisé pour regrouper des éléments de modélisation et des diagrammes.

### I.2.3.5- Modélisation du comportement

Les diagrammes de comportement montrent le comportement dynamique des objets d'un système, y compris leurs méthodes, collaborations, activités et historiques des états. Le comportement dynamique d'un système peut être vu comme la suite de changements appliqués au système au cours du temps.

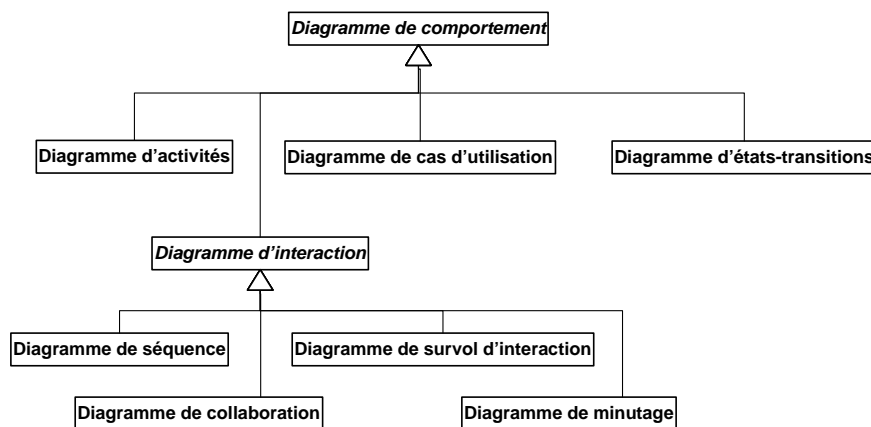


Figure 4 : UML définit 7 types de diagrammes comportementaux

- Les diagrammes d'activités représentent le comportement, sous la forme de séquences d'actions, au sein de flots de contrôle. Les actions coordonnées par les activités peuvent être initiées par la terminaison d'autres actions, parce que des objets ou des données sont disponibles, ou sur occurrences d'événements externes au flot de contrôle.
- Les diagrammes de cas d'utilisation permettent de segmenter les exigences, en exprimant les attentes de chaque catégorie d'utilisateurs (les acteurs) en cas d'utilisation, c'est-à-dire en familles d'interactions.
- Les diagrammes d'états-transitions capturent le comportement d'un groupe d'objets (typiquement instances d'une classe) sous la forme d'un graphe d'états, reliés par des transitions déclenchées par des événements.
- Les diagrammes d'interactions offrent plusieurs formalismes pour représenter les sociétés d'objets collaborant. Selon les cas, l'éclairage peut-être mis sur la structure, sur le comportement ou sur le temps.

## I.3- Modèles : du contemplatif au productif

Le statut des modèles évolue d'une phase « contemplative » à une phase « productive » : auparavant les modèles étaient considérés comme des éléments de documentation ; ils sont maintenant de plus en plus souvent devenus des entités de première classe, traités comme entrées ou sorties par des processeurs automatiques.

### I.3.1- Modèle et réalité en logiciel

« Ne pas confondre la carte et le terrain » est une des citations les plus célèbres attribuées au stratège chinois Sun Tse pour rappeler qu'une abstraction peut laisser de côté certains détails critiques, et qu'il y a une différence entre travailler sur un modèle ou sur la réalité. Magritte a aussi capturé de façon saisissante cette différence entre le modèle et la réalité au travers de son célèbre dessin, « *Ceci n'est pas une pipe* » (cf. Figure 5) : en effet, si on peut fumer la pipe, on ne peut fumer son dessin. D'un autre côté, il est bien connu que le logiciel est un artefact spécial dans le sens où il est immatériel. Ainsi, le modèle d'un logiciel et le logiciel lui-même ont la même nature, et passer de l'un à l'autre ne pose pas de problème particulier (comme c'est bien connu de n'importe quel auteur de compilateur dans le monde). Donc dans le cas du logiciel, nous sommes en quelque sorte dans un cas particulier qui nous autorise à confondre dans une certaine mesure la carte et le terrain.



Figure 5 : Ceci n'est pas une pipe par Magritte

### I.3.2- Assigning meaning to models : la notion de métamodèle

Tout modèle explicite est exprimé dans un certain langage. Ce langage peut être le français (plus ou moins contraint), comme par exemple dans le cas d'expression des besoins informels, ou à l'autre extrémité du spectre un langage de programmation ayant une syntaxe et une sémantique bien définies, en passant par des langages graphiques montrant sous forme de diagrammes des enchaînements de tâches, des comportements, les architectures de déploiement de produits aussi bien que des étapes de projets. Il y a donc une grande variété de modèles et donc de langages.

Dans la lignée de R. Floyd qui dès les années 60 proposait d'affecter un *sens* à des programmes (*Assigning meaning to programs*), nous avons besoin d'un moyen pour caractériser la signification d'un modèle, c'est-à-dire la sémantique du langage dans lequel il est exprimé.

Dans notre contexte, nous utiliserons le mot « métamodèle » pour désigner la structure des modèles exprimés dans un langage donné. Un métamodèle peut-être vu comme une forme simplifiée d'ontologie qui définit un ensemble de concepts (et de relations entre ces concepts)

sur lesquels une communauté d'utilisateurs s'est mise d'accord. Par exemple la communauté du génie logiciel à objets, réunie sous l'égide de l'OMG, s'est mise d'accord sur le métamodèle UML.

Un métamodèle sert de filtre fournissant une spécification précise de l'ensemble des détails d'un système qui peuvent être sélectionnés pour former un modèle de ce système. Du point de vue de définition d'un langage, un métamodèle décrit des concepts de base, comme des noms ou des nombres à utiliser, et exhibe une structure interne qui peut être hiérarchique comme dans les langages textuels, ou former un graphe dans le cas de beaucoup de langages dont la syntaxe concrète est constituée de diagrammes. Comme pour la notion de syntaxe abstraite pour les langages textuels, un métamodèle a les deux rôles suivants :

- il contraint l'ensemble des modèles possibles ;
- et il fournit la structure de ce modèle.

On a souvent recours à un langage logique défini sur le métamodèle pour spécifier des conditions contextuelles particulières permettant de contraindre plus précisément l'ensemble des modèles possibles (par exemple pour imposer que des noms soient définis correctement, qu'un cycle ne puisse se produire dans la fermeture transitive d'une relation, ou encore tout simplement qu'un nombre soit borné d'une façon particulière, etc.). Par exemple pour UML, le langage d'expression de contraintes utilisé dans le métamodèle est le même que celui utilisé au niveau des modèles, à savoir OCL (Object Constraint Language).

De plus, en UML2.0, des notions auxiliaires permettent la paramétrisation des éléments de modélisation et la gestion des modèles et des métamodèles. Par exemple le concept de profil est disponible pour définir une spécialisation d'un métamodèle, dans le but de réaliser une personnalisation, par exemple pour une cible particulière ou pour suivre une démarche méthodologique spécifique.

### **I.3.3- Le méta-métamodèle racine de l'OMG**

Le langage UML est un langage de modélisation généraliste: il propose une multitude de concepts pour représenter divers aspects d'un système, tant statiques que dynamiques. Beaucoup de ces concepts sont très génériques de façon à pouvoir être utilisés dans un grand nombre de situations. Dans ce sens, UML cherche à concrétiser l'utopie du langage universel applicable quelque soit le domaine, si besoin est, en utilisant des artifices permettant d'étendre sa sémantique (via le mécanisme des profils). En dépit des avantages de cette approche, telle que une certaine uniformisation de l'outillage et de la notation, les limitations de cette approche généraliste sont connues : risques d'ambiguïtés, d'imprécisions, de représentations non adaptées aux spécificités d'un domaine.

La *métamodélisation* est une approche alternative à ce problème. A défaut de pouvoir définir un langage unique et universel, on va proposer un moyen uniforme pour définir des langages ou des familles de langages. Dans le cas de l'OMG, le standard en question est le MOF, pour Meta Object Facility. Du fait du parti pris pour les technologies objets, les concepts de base de ce métalangage s'appuient sur une structuration par objets : des classes, des associations entre classes, des attributs de classes, l'héritage entre classes et des paquetages. A cela il faut ajouter les contraintes, qui sont un moyen permettant de préciser la validité des modèles. Il est important néanmoins de souligner que la sémantique complète d'un langage ne peut en aucun cas être exhaustivement définie par la donnée de sa syntaxe

abstraite – le diagramme de classes définissant le métamodèle conforme MOF, même augmenté des contraintes en OCL. En fait, la signification des données et la sémantique d'exécution, quand elle existe, sont dans la plupart des cas données sous forme de texte en langue naturelle.

Au standard MOF de l'OMG sont associés un certain nombre de spécifications permettant de définir, d'une part, comment on *opérationnalise* un métamodèle – c'est-à-dire comment on le manipule en utilisant un langage de programmation ou d'interface, par exemple le mapping vers IDL, ou le mapping JMI vers Java - et d'autre part, comment on représente textuellement des instanciations d'un langage défini avec le MOF. C'est, dans le dernier cas, le rôle des standards XMI et HUTN (voir chapitre 2).

L'apport essentiel du MOF dans l'ingénierie des modèles est de permettre d'uniformiser le mode de représentation et la façon de manipuler les métamodèles, les modèles et les données. Classiquement, la métamodélisation à l'OMG distingue 4 niveaux de modélisation: le méta-métamodèle (M3), occupé par le modèle MOF lui-même, les méta-modèles (M2), les modèles (M1) et enfin les données (M0), que certains interprètent comme « le monde réel ». Dans la pratique, cette vision a évolué vers une interprétation *relative* des couches de modélisation, avec une relation de conformité entre modèles de niveaux différents. Un modèle M sera dit conforme à un modèle M' si M' est le métamodèle de M. Un modèle «utilisateur», quand il a par exemple pour objectif de décrire un langage, peut ainsi être vu comme un métamodèle, et une simple donnée peut, dans de nombreux cas, être vue comme l'instance d'un élément d'un modèle. Cette uniformisation est notamment essentielle pour pouvoir définir des formalismes de transformation de modèles génériques (Voir chapitre 4).

On peut noter que le paradigme *d'instanciation* qui est utilisé dans le monde objet existe également dans d'autres espaces technologiques, notamment dans le monde XML où le *schéma* sert de formalisme générique pour définir tout langage basé sur XML.

### **I.3.4- Correspondances entre espaces technologiques**

Dans le contexte d'évolution technologique galopante dans lequel nous nous trouvons, se développe un sentiment de méfiance vis à vis de nouvelles solutions. Les différentes propositions que l'on a pu voir arriver dans les dernières années ont apporté plus de possibilités mais moins de certitudes sur la meilleure façon de résoudre les problèmes. Dans ces conditions il est évidemment difficile de promettre que l'ingénierie dirigée par les modèles pourra nous apporter tout ce que les autres technologies n'ont pas pu nous fournir. La solution miracle, ce que nos collègues américains appellent le "silver bullet" n'existe pas, sinon cela se saurait.

Pour avoir un regard lucide sur la situation des offres, pour pouvoir établir des guides neutres sur la meilleure façon de résoudre un problème spécifique sans a priori partisan, il est important de proposer une vision globale de cette offre technologique. Une démarche que l'on pourrait qualifier d'agile devrait permettre de choisir la meilleure technologie ou le meilleur ensemble complémentaire de technologies pour résoudre un problème particulier.

Dans [I] l'idée des espaces technologiques a été proposée pour répondre à ce besoin. Un espace technologique (ET en abrégé) est à la fois un contexte précis de travail, un système uniforme de représentation, un corps de connaissances, un ensemble d'outils et de solutions et même une communauté d'utilisateurs partageant des savoirs et des savoir-faire, une littérature commune, des matériaux pédagogiques, etc.

Les ET sont parfois difficiles à délimiter exactement, mais on peut les reconnaître assez facilement. Quelques exemples permettront de mieux identifier le concept :

- L'espace des langages de programmation (ou du "grammarware" comme récemment nommé par l'équipe de P. Klint du CWI) est l'un des plus anciens et des plus formalisés. Le système de représentation commun est basé sur les arbres de syntaxe abstraite. Il y a ici aussi de nombreux sous-espaces correspondant à des langages spécifiques, par exemple Java, Haskell, Lisp ou Prolog.
- Un autre espace beaucoup plus récent est aussi basé sur des arbres, mais une forme différente d'arbres. C'est l'espace des documents XML avec toutes ses variantes, ses outils associés et sa dynamique de développement.
- Un peu antérieur à l'espace XML, on peut citer l'ET des ontologies ou plus précisément de l'ingénierie ontologique qui est une branche de l'ingénierie des connaissances. De nombreux outils partagés comme l'outil de manipulation d'ontologies Protégé servent souvent de fédérateur à la communauté.
- Un espace très important est celui des bases de données qui a montré depuis longtemps son efficacité dans le domaine de l'indépendance de la plate-forme et qui a élaboré des standards de langages très stables comme SQL (voir chapitre 6).
- Le plus récent des ET est donc le Modelware qui correspond à l'ingénierie dirigée par les modèles. Il englobe plusieurs sous-familles comme le MDA™.

Il n'y a pas de solution technologique uniformément supérieure aux autres. Il n'y en aura sans doute pas. Chaque ET possède des avantages et des inconvénients. On peut les comparer suivant de très nombreux critères, par exemple:

- La modularité,
- L'exécutabilité ou aptitude à produire aisément des systèmes directement exécutables,
- L'adaptabilité qui permet en particulier de spécialiser facilement des solutions générales,
- La traçabilité permettant de mettre en relation des éléments produits à différents stades d'un processus d'élaboration,
- La stabilité dans le temps<sup>1</sup>,
- La transformabilité, etc.

Les ET ne sont pas des îles et s'influencent mutuellement. Des bonnes idées d'un ET peuvent parfois être reprises dans un autre ET.

Les ET sont structurés souvent de façon très similaire. Le système global de représentation d'un ET est souvent construit autour d'un métamétamodèle. Pour prendre deux exemples, le Grammarware peut se représenter autour du formalisme EBNF et l'espace des documents XML autour du métamétamodèle des documents bien formés. Par ailleurs, on trouve en général une organisation multi niveau propre à chaque ET comme support de ses notions de type. Un programme par exemple s'appuie sur une grammaire comme un document s'appuie sur un schéma, comme un modèle s'appuie sur un métamodèle. Un document XML bien formé pourra de plus être valide par rapport à une DTD donnée.

Lorsque certaines fonctionnalités existent dans un autre espace, il est parfois plus intéressant de les utiliser dans cet espace que de les réimplanter. Prenons par exemple le problème de la sérialisation des graphes dans l'ET ModelWare. Au lieu de définir un standard complet de sérialisation de graphes, il a semblé plus commode de définir une transformation

---

<sup>1</sup> L'une des idées force de l'OMG est d'affirmer que la pérennité des investissements informatiques est garantie par la meilleure stabilité dans le temps du MDA™. Cette affirmation reste à valider.

standard des graphes en arbres XML et d'utiliser XMI, la sérialisation standard d'arbre XML, puisque de très nombreux outils existent pour assister dans leur manipulation. De la même façon lorsque l'exécutabilité est requise, on pourra utiliser un pont entre l'ET modelware et l'ET Java avec le standard JMI. Ces notions seront étudiées en détail dans le chapitre 2 de cet ouvrage.

La connaissance des différents ET et des ponts disponibles entre ces différents espaces permet une agilité maximale. Elle permet aussi de bien placer en perspective les apports et les limites du MDA™. Le MDA™ ne sera pas l'outil miracle, mais dans certains cas il offrira des solutions plus intéressantes que celles des autres ET.

## **I.4- Le MDA selon l'OMG**

Depuis plus de dix ans, l'OMG (« Object Management Group ») propose des standards autour de CORBA (« Common Object Request Broker Architecture»). Celui-ci avait pour objectif de définir un intergiciel standard, ainsi qu'une série de services particuliers, en charge de la gestion transparente des applications distribuées dans un environnement hétérogène. En parallèle à cette activité, au cours des cinq dernières années, l'OMG a également travaillé sur le langage unifié de modélisation, « UML (Unified Modelling Language) », qui comme on l'a vu précédemment s'est rapidement imposé comme le langage de modélisation de nombreuses méthodes de développement logiciel. Maintenant que ces deux standards majeurs, CORBA et UML, ont atteint un niveau de maturité suffisant, l'OMG a décidé de rapprocher ces deux standards sous un même chapeau relevant de l'ingénierie de la modélisation. Cette « nouvelle » technologie porte le nom de « MDA » pour « Model Driven Architecture », i.e. Ingénierie dirigée par les modèles.

Derrière le MDA se cache le très ancien concept du génie logiciel de séparation de la description des fonctionnalités d'un système, de la description de sa réalisation. Autrement dit, le MDA n'est rien d'autre que la séparation du *quoi* et du *comment* au travers de l'utilisation de modèle comme pierre de base du développement d'un produit. La technologie du MDA définit alors un certain nombre de recommandations que le reste de ce paragraphe vise à décrire. Les définitions suivantes sont directement extraites du guide de l'utilisateur MDA défini par l'OMG [[3]] et sont données ici pour mémoire. Cependant il est clair qu'avec le recul, la plupart de ces définitions sont discutables et feront certainement l'objet de révisions à court terme.

### **I.4.1- La notion de CIM**

Un CIM (« Computation Independent Model ») représente principalement l'environnement d'un système et ses besoins. Il ne donne pas de détails sur la structure interne du système, ni même sur son exécution possible. Le CIM s'apparente à un modèle métier, c'est-à-dire un modèle ne manipulant que des artefacts directement issus du domaine applicatif dont la spécification modélisée est issue. Ce modèle est très utile pour établir des ponts de communication et faciliter ainsi la compréhension entre les experts du « métier » maîtrisant les besoins et les experts en charge de la réalisation de ces besoins.

### **I.4.2- La notion de PIM**

Un modèle indépendant de la plateforme (PIM pour « Platform Independent Model ») décrit un système sans en exposer les détails relatifs à une plateforme d'exécution support. Ce modèle se concentre ainsi sur la description des parties d'un système qui ne sont pas en liens direct avec la plateforme d'exécution sous-jacente. Un tel modèle peut être construit tout aussi bien à partir d'un langage de modélisation généraliste, qu'à partir d'un langage dédié métier.

Une façon usuelle de construire un modèle indépendamment d'une plateforme est de s'appuyer sur une technologie « neutre », i.e. non orientée vers une technologie spécifique d'implantation, une machine virtuelle. Celle-ci propose des services définis indépendamment d'une technologie. Donc, même si une machine virtuelle peut constituer une plateforme en soi, elle peut facilement être portée sur de « véritables » plateformes de conception (CORBA, EJB...). Un modèle s'appuyant sur une machine virtuelle peut ainsi continuer à être considéré comme un modèle indépendant d'une plateforme.

### **I.4.3- La notion de PSM**

Un modèle spécifique à une plateforme propose une vue d'un système en mettant en avant comment celui-ci est supporté par cette plateforme d'exécution. Un PSM est un PIM lié à une plate-forme spécifique.

### **I.4.4- La notion de modèle de plateforme**

Selon l'OMG, un modèle de plateforme fournit un ensemble de concepts technologiques qui constituent la plateforme, ainsi que les services fournis par celle-ci. Par exemple, une plateforme spécifique au modèle de composant de CORBA propose des composants particuliers tels que les type de composant entité, session, processus... Cette notion de modèle de plateforme est probablement l'un des concepts les plus ambigus dans le MDA, aussi reprendrons nous la discussion sur sa finalité dans le chapitre 4.

### **I.4.5- La transformation de modèles**

Indispensable pour automatiser le passage d'un modèle à un autre, la notion de transformation de modèles est au coeur du MDA. Il y a bien sûr longtemps que l'on utilise des outils de transformation en informatique. Une première génération d'outils a pu être développée en combinant au moyen de *pipes* des commandes Unix comme *awk* ou *sed* (qui permettent de transformer des fichiers textes composés de lignes successives). Une seconde génération d'outils est apparue plus récemment dans le contexte des documents structurés XML, avec par exemple XSLT. Mais le problème avec ces deux premières générations d'outils est qu'il est difficile de concevoir et surtout de maintenir des transformations complexes. Une limite raisonnable en termes de lisibilité et de maintenabilité pour les outils de première génération est probablement de l'ordre de la centaine de lignes de code, et sans doute pas plus du millier de lignes pour un programme XSLT. Or, dans certains domaines comme par exemple celui des applications réparties temps-réel (télécommunications, contrôle de trafic aérien, systèmes C3I, etc.) la complexité (et le savoir-faire des ingénieurs qu'il est important de capitaliser) réside au moins autant, si ce n'est plus, dans la *manière* de concevoir le système que dans l'élaboration de modèles métiers, c'est-à-dire dans la transformation d'un PIM vers un PSM autant que dans l'élaboration du PIM.

Les raisons qui font que cette transformation peut-être complexe sont multiples. Prenons l'exemple du contrôle de trafic aérien. C'est un cas caractéristique dans lequel le modèle métier abstrait est assez stable (c'est grosso modo le même depuis une cinquantaine d'années) et en fait pas si complexe. Mais il faut bien sûr tenir compte de nombreux points de vues liés aux différents métiers concernés, ce qui fait que typiquement divers langages de modélisation sont utilisés au-delà d'UML. De plus, le plus souvent le logiciel doit être conçu, réalisé, et déployé sur de nombreuses variantes de plates-formes : à cause de la dimension internationale de ce genre de projets, l'hétérogénéité est la règle. D'autre part, il est souhaitable de pouvoir réutiliser les mêmes solutions techniques (par exemple pour gérer la tolérance aux défaillances ou encore la sécurité) pour toute une famille de produits.

Ceci induit la nécessité de pouvoir adapter et spécialiser des transformations génériques, ainsi que de pouvoir composer des transformations réutilisables. Et bien sûr, il faut être capable de maintenir et de faire évoluer ces transformations pour de longues périodes (quinze ans et plus). On peut par exemple envisager [I] d'utiliser pour les transformations de modèles les techniques qui ont fait leurs preuves en génie logiciel.

- Les transformations doivent être modélisées, par exemple avec UML en s'appuyant sur la puissance de la technologie objet.
- Les transformations doivent être conçues comme des composants réutilisables, s'appuyant sur des notions comme la conception par contrat qui est naturelle à mettre en oeuvre dans un environnement UML à l'aide d'OCL.
- Les transformations doivent être implantées de manière concrète et mises à disposition sous forme de bibliothèques de composants et de canevas spécialisables.
- Les transformations sont des éléments de programmes complexes qu'il est nécessaire de valider, par exemple à l'aide de tests. Dans ce cas les jeux de données d'entrée et de sortie des tests sont des modèles (par exemple UML), et l'oracle de test peut-être facilement élaboré à l'aide des contraintes OCL dans le schéma classique d'une conception par contrat.
- Les transformations devront être maintenues et évoluer pour répondre à de nouveaux besoins, ce qui rend nécessaire le fait de les gérer en configuration. Mais comme ce sont aussi des modèles, on pourra aussi envisager de les faire évoluer automatiquement à l'aide de transformations de transformations.

Il apparaît donc clairement qu'orthogonalement au cycle de vie classique du logiciel, une deuxième dimension (celle de la transformation de modèles) intervient de manière de plus en plus prégnante dans la construction d'applications logicielles complexes et bouleverse les pratiques des équipes de développement.

La transformation de modèle étant donc l'une des pièces maîtresses du MDA, nous reviendrons sur ses enjeux au chapitre 4.

## **I.5- Au-delà du MDA : l'ingénierie des modèles**

### **I.5.1- Principes de base**

Si le MDA est une façon nouvelle *d'appréhender* le développement logiciel, il faut cependant noter qu'il est surtout une tentative de formalisation d'un ensemble de pratiques déjà existantes et souvent très largement répandues dans l'industrie, comme la programmation générative, l'utilisation de *wizards* (omniprésente dans les environnements de développement Windows), ou encore plus récemment la programmation par aspects.

Dans la vision du MDA à l'OMG, apparaît la problématique centrale de l'existence d'une multitude de plates-formes technologiques de plus en plus complexes (par exemple .Net, Java/EJB, etc.), avec en corollaire le problème du coût de migration des systèmes informatiques entre plates-formes technologiques qui devient rédhibitoire. La solution proposée par l'OMG consiste à découpler les parties "métiers" et les parties "technologiques" dans les systèmes. Mais rien n'interdit d'avoir une vision plus large sur cette problématique. En effet comme on l'a vu, en dehors du code, de nombreux modèles sont développés et co-

existent au cours d'un développement logiciel, incluant par exemple des modèles de métiers, de besoins, de test, d'architecture, de déploiement, etc.

Le maintien en cohérence de tous ces modèles, ainsi que le passage automatisé entre certains d'entre eux, forme une problématique plus large que le MDA stricto sensu, que nous appelons ingénierie des modèles. Attachons-nous à en résumer les principes de base.

1. *Tout artefact de développement logiciel est un modèle d'un aspect d'un système*  
Ce premier principe découle directement de la définition de ce qu'est un modèle.
2. *Un modèle est décrit par un métamodèle qui est un langage spécialisé pour l'aspect considéré*
3. *La construction d'un système informatique revient à effectuer un tissage d'aspects.*

La séparation des aspects, pris en compte dans divers modèles, est nécessaire. Cette séparation est beaucoup mieux gérée par un système à modèles séparés plutôt que par une projection sur le code comme c'est le cas pour la programmation par aspects.

Compte tenu de cet ensemble de principes, il est clair que *le MDA est un cas particulier d'ingénierie des modèles*, qui se focalise sur un aspect particulier (la dépendance vis-à-vis d'une plate-forme technologique), et qui ne propose rien d'autre que de tisser l'aspect plate-forme lors du passage PIM vers PSM.

## **I.5.2- Généralisation au-delà d'une perspective constructiviste du logiciel**

En fait, d'un point de vue scientifique, cette ingénierie des modèles peut encore se généraliser au-delà d'une perspective constructiviste du logiciel. Cette généralisation déborde du cadre de cet ouvrage, mais le lecteur intéressé pourra se référer à la synthèse produite par l'Action Spécifique CNRS sur l'ingénierie dirigée par les modèles dont l'objectif était : *"Etudier la problématique scientifique liée à l'ingénierie dirigée par les modèles en relation avec les récents développements industriels et normatifs"*. Cette synthèse est disponible sur le site Web de ce groupe de travail : <http://www-adele.imag.fr/mda/as>. Une autre source d'information pertinente est le séminaire Dagstuhl #04101 qui s'est déroulé en mars 2004. On pourra en consulter les principaux résultats sur le site <http://www.dagstuhl.de/04101/>.

## **Bibliographie**

- [1] Bézivin, J., Farcet, N., Jézéquel, J.M., Langlois, B., et Pollet, D. *Reflective model driven engineering*. in G. Booch P. Stevens, J. Whittle, editor, Proceedings of UML 2003, San Francisco, volume 2863 of LNCS, pages 175-189. Springer, (October 2003)
- [2] Kurtev, I., Bézivin, J., Aksit, M. : Technological Spaces: An Initial Appraisal, Int. Federated Conf. (DOA, ODBASE, CoopIS), Industrial track, Irvine, Ca, 2002.
- [3] OMG MDA User Guide OMG document technique omg/2003-06-01, Juin 2003.