D'UML à MDA

En passant par les méta modèles

Antoine Clave

L'impact d'UML (Unified Modelling Language) sur l'informatique a été décrit dans plusieurs articles de La Lettre, cités en page 17. Langage de modélisation, UML permet une description complète du Système cible, depuis ses spécifications fonctionnelles, métier, jusqu'à son implémentation concrète.

Définition du MDA

MDA (acronyme de Model Driven Architecture, architecture dirigée par les modèles) est une démarche de développement proposée par l'OMG[1], permettant de séparer deux visions extrêmes du même système :

- ses spécifications fonctionnelles d'une part,
- son implémentation physique d'autre part,

incluant plusieurs aspects de la vie du logiciel, à savoir ses tests, ses exigences qualité, la définition des itérations successives, etc.

Pour ce faire, MDA propose de structurer les besoins selon une Architecture, indépendante du contexte technique, avant de se livrer à une transformation de cette modélisation fonctionnelle en modélisation technique, tout en testant chaque modèle produit. MDA a donc l'ambition de proposer une vision la plus large possible du cycle de vie du logiciel, ne se contentant pas uniquement de sa production. De plus, cette vision globale se veut décrite dans une syntaxe unifiée.

Cet article propose d'exposer la démarche MDA. Il ne s'étendra pas sur certaines caractéristiques de MDA, pourtant majeures, telles l'interopérabilité, les générateurs de code, etc.

Notion de méta-modèle

C'est UML qui nous a permis d'abandonner un processus de production de programme centré sur le code au profit d'un processus centré sur la modélisation. Dans ce passage la notion de méta-modèle fut cruciale ; ainsi si la représentation graphique du système est elle-même un modèle, la syntaxe décrivant le modèle constitue un méta modèle.

Un exemple simple permettra de bien cerner le pourquoi de cet empilement de modèles successifs.

Soit un programme écrit en C et destiné à piloter une machine numérique. L'exécution de ce programme est une instanciation de ce programme, une instanciation possible parmi une infinité d'autres. Ce programme C est un modèle en tant que représentation possible de toutes ces exécutions. Nous qualifions de modèle ce source C car ce source est une représentation (des services rendus par ce logiciel, une fois compilé et déployé), partielle mais juste.

Ce programme C est lui-même exprimé à partir d'une grammaire, celle de Kernighan et Richie, grammaire qui est capable d'exprimer une infinité de programmes C à la syntaxe correcte. Cette grammaire (souvent décrite aujourd'hui sous forme BNF[2]) est un modèle de tous les programmes C écrits de par le monde. Modèle de modèles des exécutions : on conviendra qu'elle est le méta modèle de toutes ces exécutions de logiciels développés en C.

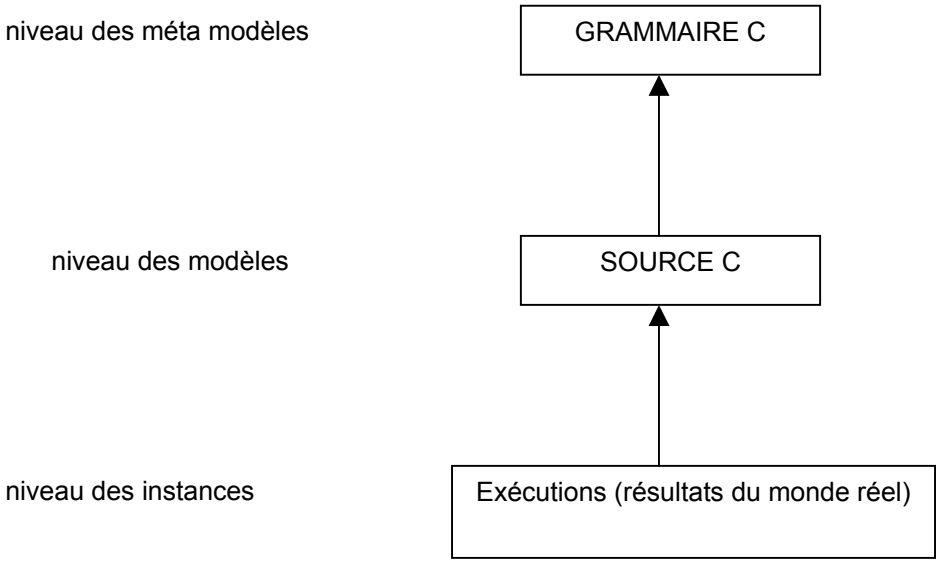
Le compilateur C, qui valide la bonne écriture du source, donc sa conformité à la grammaire C, est un vérificateur de méta modèle. Mais il ne valide pas le modèle : le modèle (rappelons-nous : le programme) est validé par les tests.

Et donc il apparaît que tous ces développements obéissent à une même logique interne, logique qui se révèle dès qu'on atteint un niveau suffisant d'abstraction. Cette logique, c'est la syntaxe du C.

Un des principaux enjeux est l'interopérabilité rendue possible car s'appuyant sur des descriptions communes. Ceci apparaît dès qu'on veut faire communiquer des programmes C compilés par différents compilateurs : ces derniers, qui ont validé la conformité au méta modèle, rendent possible cette interopérabilité (à condition évidemment que les modèles eux-mêmes s'y prêtent).

[1] OMG : Object Management Group, Association de professionnels de l'informatique orientée objet, http://www.omg.org

[2] BNF : Il ne s'agit pas ici de la Bibliothèque Nationale de France mais de l'acronyme de Backus Naur Form (ou Backus Normal Form), une notation métasyntaxique utilisée pour spécifier la syntaxe des langages de programmation, des langages de commande et tout autre langage apparenté.



Du modèle au méta modèle : 3 niveaux d'abstraction

Ceci dit, ce méta modèle est inopérant à s'adjuger la méta représentation de programmes écrits dans des langages s'écartant du C.
Rendre un pilote écrit en C activable par un programme Delphi (syntaxe Pascal) oblige à passer par une interface, souvent écrite en assembleur.
Or c'est justement le but visé par MDA : rendre ces représentations compatibles à un certain niveau d'abstraction.

Le danger apparut que si UML se révélait un très bon langage de modélisation, il était lui-même dédié à un type d'activités : la modélisation, technique ou fonctionnelle, des services rendus par un Système.

Il se révèle en revanche inopérant pour modéliser d'autres secteurs d'activités de l'industrie logicielle, telles que les tests, les entrepôts de données (data warehouses), les normes qualité, etc.

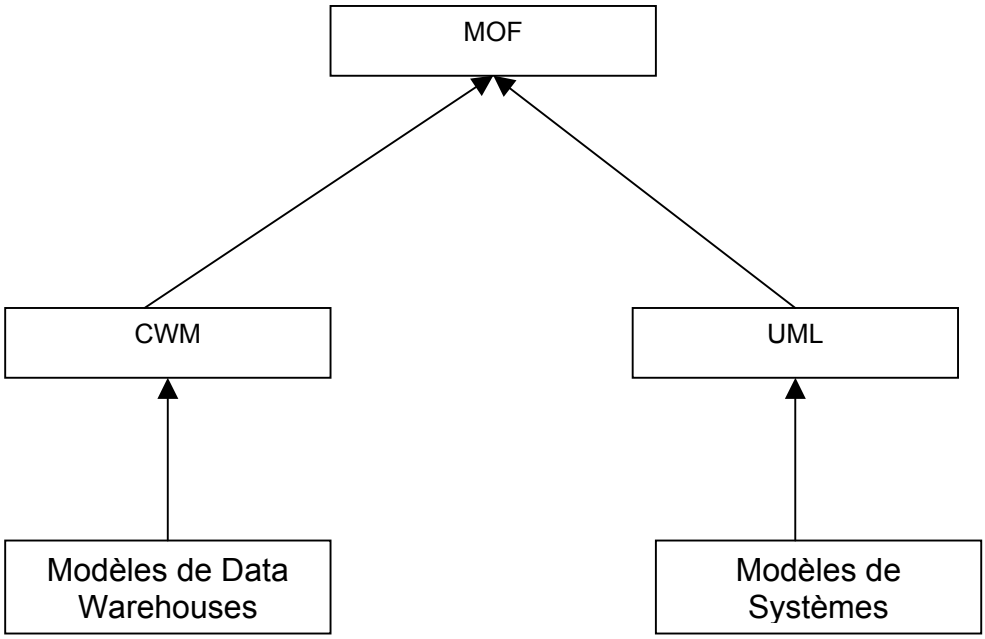
Donc, si c'est un bon langage de modélisation, ce n'en est qu'un possible parmi beaucoup d'autres. Et il devint urgent d'appuyer UML lui-même sur un autre langage : le langage de définition de modèle, langage qui servirait à la fois à décrire UML et les autres modèles, qui eux seraient dédiés aux autres activités logicielles citées ci-dessus.

C'est dans ce contexte d'abstraction de plus en plus poussée que la démarche MDA s'insère. Tous basés sur le même langage, UML et les autres méta modèles s'avèrent capables de communiquer et de ne plus développer d'incompatibilités d'intégrations. Ce langage commun est le MOF : Meta Object Facility. Le MOF offre un modèle abstrait d'objets génériques, et leurs associations, un ensemble de règles pour exprimer le cycle de vie, la composition, la fermeture sémantique[1], la possibilité de manipuler un modèle exprimé en MOF à partir d'interfaces IDL[2].

À l'heure actuelle, et à ma connaissance, le CWM (Common Warehouse Model) est le langage de modélisation standardisé par l'OMG dédié aux entrepôts de données et, comme UML, basé sur le MOF, et donc compatible avec UML. Une des conséquences les plus visibles est leur sérialisation[3] possible en XMI (« dialecte » XML), et donc toute modélisation en UML ou CWM sera lisible, interprétable, par tout système compatible XML, adjoint de la DTD (ou du schéma) XMI.

---

[1] Fermeture sémantique : caractéristique d'une expression, d'une modélisation, d'une représentation dont le sens est complètement défini et non ambigu.
[2] IDL : Interface Definition Language : Langage d'interfaçage des objets sous Corba. Le fichier IDL contient les définitions des interfaces que le client ou le serveur supporte.
[3] Sérialisation : transformation d'un ensemble de données en un flot, dirigé vers un périphérique (de masse, ou port série, ou autre) pour être transmis à un récepteur qui par un processus inverse, le reconstituera. Il s'agit d'un algorithme trivial pour des données structurées en champs, mais très complexe quand il s'agit d'instances de classes complètes, ou d'arbres, de graphes, etc.



Le Meta Object Facility, modèle commun à UML et à CWM

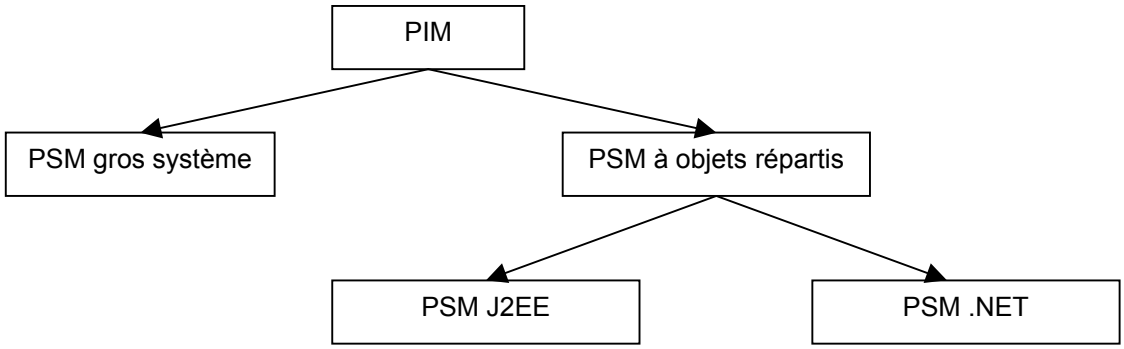
Mise en œuvre MDA : 2 étapes majeures

Les 2 étapes majeures dans la mise en œuvre du processus MDA sont :

- *Réalisation d'un Modèle Indépendant de toute plate forme* : le PIM (Platform Independent Model) ; enrichissement de ce modèle, stabilisation.
- *Génération du Modèle Spécifique de la Plate Forme* : le PSM (Platform Specific Model), jusqu'à la génération de code incluse, aboutissant à un modèle exécutable. Cette génération se fait par transformations successives du PSM.

PIM (*Platform Independant Model*) et PSM (*Platform Specific Model*) sont deux concepts fondamentaux dans ce processus.

Dans la perspective MDA, tout livrable cohérent est un modèle : c'est une description complète du Système cible. Une conséquence majeure est que tout modèle est testable, et pas uniquement les modèles PSM. Pour fixer les idées, nous dirons grossièrement que si les Spécifications fonctionnelles et l'Analyse correspondent aux activités du PIM, la transformation du PIM en PSM, et l'écriture des règles de transformation du PIM en PSM dans le contexte technique précis du système à produire correspondent à la Conception. Si le PSM inclut le code et sa production automatisée, l'adaptation du générateur de code est à prévoir, cette adaptation comprend l'écriture de règles qui feront elles-mêmes l'objet d'un document à livrer.
La modélisation selon le processus MDA fournira donc à la fois la documentation et l'implémentation.



La génération d'un PSM *à partir d'un* PIM *peut se faire en plusieurs étapes*

Le PIM

Le PIM est exprimé en UML ; essentiellement composé du Dossier des Spécifications fonctionelles et du Dossier d'Analyse, les diagrammes UML disponibles pour décrire ce modèle fonctionnel (appelé Architecture fonctionnelle) sont :

Modèle / parties du modèle	Composantes	Diagrammes UML possibles
Dossier des Spécifications fonctionnelles	Description générale (modèle du métier, cahiers de procédure) Identification et description détaillée des Cas d'Utilisation Description des Entités métiers *Description des Interfaces Homme Machine*	Diagramme des Cas d'Utilisation
Description d'un *Cas d'Utilisation*	Pré conditions Description des Activités Règles de Gestion Post conditions	Diagramme des Activités, diagramme des Séquences, diagramme de collaboration
Description des Entités métiers	Description fonctionnelle, statique et dynamique	Diagramme de Classes Diagramme des États et Transitions Diagramme des Séquences
Description des Interfaces Hommes machines	Maquette de chaque écran (HTML) Circulation entre les écrans (Carte de navigation)	Diagramme des États et Transitions
Dossier d'Analyse	Composants fonctionnels (paquetages de classes), identification des services.	Diagramme de Classes, de séquences, de collaboration.

Le PIM décrit le métier et les services rendus par le Système ; le souci du Chef de Projet est sa complétude, son univocité, sa clarté et sa cohérence. Il peut intégrer des spécifications en matière de sécurité, de persistance, de gestion des transactions… Dans le cadre d'une démarche MDA, (et en plus si ses objectifs comprennent la production partiellement automatisée du code), il vérifiera que toutes les descriptions seront intégralement traduites dans le diagramme des classes. En clair, cela signifie que chaque état sera traduit par la valeur d'une variable d'état ; que chaque message sera implémenté par une méthode, que chaque classe cible d'un événement sera abonnée à cet événement, que l'empaquetage sera compatible avec les règles de visibilité, etc.

Le PSM

La transformation du PIM en PSM consiste à insérer dans l'Architecture fonctionnelle une Architecture technique. Cette transformation du modèle PIM en modèle PSM peut passer par plusieurs étapes, en fin de compte suivre un chemin de transformation qui peu à peu permet de s'approcher du PSM. Chacune de ces étapes est décrite en UML et montre comment les composants fonctionnels sont transformés en composants techniques.

Les transformations des composants fonctionnels en composants techniques suivent des règles, déjà anciennes (énoncées dans Unified Process[1]) ; elles laissent une large plage d'initiative au Concepteur Technique. Le détail de ces règles dépassant largement le cadre de cet article, nous n'allons en examiner que les premières étapes. Mais elles pourront être exposées dans un article ultérieur si le besoin s'en fait sentir. Cela permettra de démonter ce « continuum » dans la modélisation, à savoir l'absence de rupture entre la spécification fonctionnelle, description technique et production du code.

Exemple rapide de passage du PIM au PSM

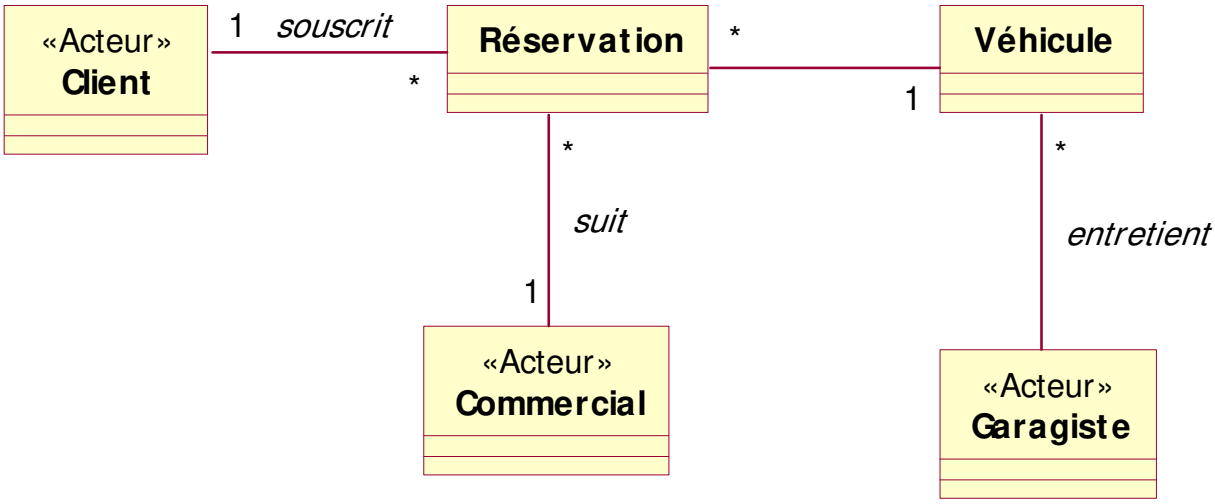
L'exemple développé rapidement ci-après n'a pour but que de fixer quelques idées phares. Il ne correspond bien entendu pas à l'étape complète.

[1] Unified Process ou UP : méthode de conduite de projet dont les auteurs sont ceux d'UML (Rumbaugh, Booch & Jacobson). Rational Software en a fait un produit commercial : RUP ou Rational Unified Process. Son intérêt réside en la systématisation des 3 descriptions : fonctionnelle, statique et dynamique, réparties au cours d'itérations, qui toutes s'appuient sur les mêmes étapes. Elle essaie de fournir une vision continue du projet, des spécifications au code, en utilisant une succession de modèles.

Soit une Agence de Location de Voitures dont nous voulons développer une offre de services sur Internet.
Ainsi, un Client connecté, après s'être identifié, aura la possibilité de réserver un Véhicule (le contrat définitif de location sera rédigé à l'Agence de retrait du Véhicule).

Soient les Cas d'Utilisation : LE CLIENT RÉSERVE UN VÉHICULE, LE COMMERCIAL SUIT UNE RÉSERVATION, LE GARAGE ENTRETIENT UN VÉHICULE.

On propose le diagramme de Classes d'Analyse suivant (très simplifié : il manque les attributs et les méthodes, fournis par les autres diagrammes UML) :



Ce diagramme est partie prenante d'un PIM (Platform Independant Model) en ce sens qu'il ne présuppose aucune solution technique d'implémentation.

Il participe à l'expression de spécifications purement fonctionnelles. Il va servir de base à la génération d'un PSM (Platform Specific Model) par injection successive de choix de conceptions techniques.

Ces choix se résument à l'adoption d'une architecture Windows, à objet répartis ; pour le moment, la MOA ne s'est pas fixée sur le choix entre C# .NET[1] et Java J2EE[2].

Dans le monde Windows, à objets répartis, la transformation en PSM se fait généralement en 2 étapes.

On commence par prendre en compte le fait que l'architecture est à objets répartis ; pour cela, pour chaque classe du modèle d'Analyse, on applique la règle de transformation (exposée ci-dessous) qui consiste à produire en 3 classes de conception primaire : classes *Entité*, classe *Contrôle*, classe *Interface*.

Dans une deuxième étape, on transforme ces classes de conception primaire en composants techniques, directement implémentables.

[1] NET n'est pas un langage ou un logiciel, mais le nom de la nouvelle stratégie de Microsoft, comprenant un ensemble de technologies permettant l'interconnexion des informations, des individus, des systèmes informatiques et des terminaux.
[2] J2EE : Java 2 Enterprise Edition. Version 2 de Java.

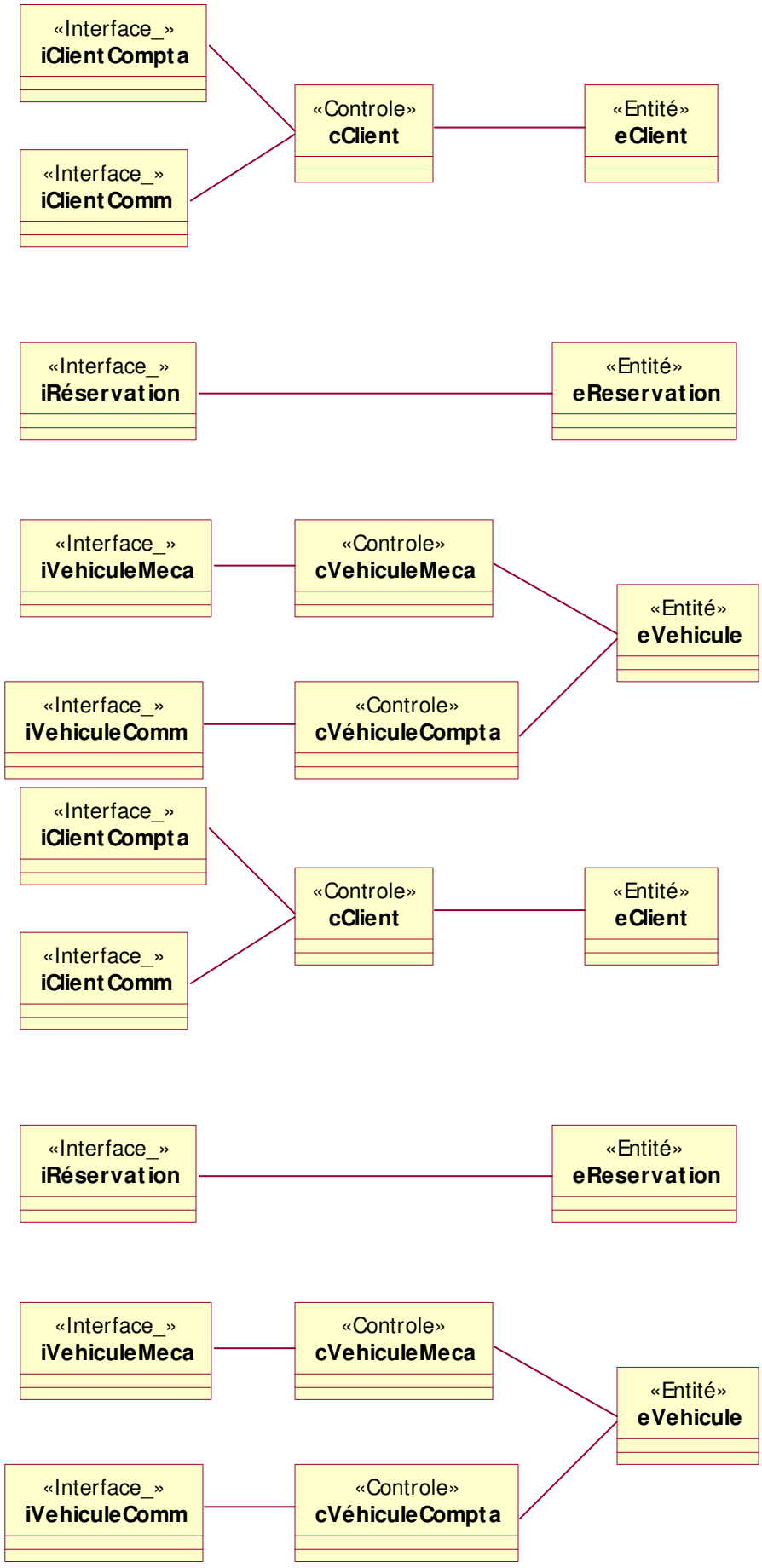
Conception primaire : première transformation des classes d'analyse.

Une classe d'analyse donnera naissance à une classe Entité, Contrôle, Interface :	À condition que …
ENTITE	La classe est persistante
INTERFACE	Certaines de ses propriétés sont publiées
CONTRÔLE	Des extractions en base, des calculs (modélisés dans les RG) sont nécessaires

Nous décidons donc :

La classe …	Donne naissance à :	Pour la raison suivante :
CLIENT	iClientComm iClientCompta (Classes interfaces)	Des propriétés sont accessibles à la comptabilité et aux commerciaux (propriétés exclusives, d'où 2 interfaces différentes)
	cClientCtrl (Classe contrôle)	Prendra en charge les contrôles et les calculs : cumuls, statistiques, suivi…
	eClient (Classe entité)	Classe de mémorisation de toutes les propriétés
RESERVATION	iRéservation (Classe interface)	Prend en charge la publication de certaines propriétés, de certains champs calculés…
	eReservation (Classe entité)	Classe de mémorisation de toutes les propriétés
	Il n'y a pas de Classe CTRL car les calculs sont pris en charge par une autre Classe et les extractions depuis la base assez triviales.	
VEHICULE	iVéhiculeMéca (Classes interfaces) iVéhiculeComm	Suivi du véhicule par le garage (entretiens) et par les commerciaux
	cVéhiculeMéca (Classes contrôles) cVéhiculeComm	Pour les calculs de coûts d'entretien et de réparation et les extractions dédiées au suivi mécanique Pour les calculs d'amortissement, les bilans de suivi et de rentabilité…
	eVéhicule (Classe entité)	Classe de mémorisation de toutes les propriétés

D’où le diagramme des classes de conception primaire :



Ce diagramme sera complété : on ajoutera les multiplicités (cardinalités) des associations, les méthodes de collaboration et les attributs.

Nous remarquons qu'une classe d'analyse peut, sans obligation, générer une ou plusieurs classes de conception. Notons l'importance des choix fonctionnels qui conditionnent ces choix architecturaux.

Ce modèle reflète une structuration en 3 couches :
- la couche de *présentation* (les classes interfaces),
- la couche de *persistance* (les classes entités),
- et entre les deux la couche des *traitements*, appelée aussi la couche métier (les classes contrôles).

Une deuxième transformation visera à appliquer soit une architecture .NET, soit une architecture J2EE, soit une architecture CORBA[1], ou autre : nous aurons alors construit un modèle de conception détaillée. La génération de composants NET, CORBA ou J2EE se fait toujours à partir de classes de conception primaire ; donc cette première structuration en couches est indispensable pour préparer la génération des composants architecturaux. Néanmoins l'exposer dépasserait le cadre de cet article.

Ce modèle sera ensuite amendé et affiné de façon à être présenté au générateur de code, qui pourra, le cas échéant, produire un squelette composé des paquetages, des déclarations de classes, de méthodes, d'attributs. Ce squelette recevra le corps des méthodes (le*s instructions des méthodes sont toujours construites à la main*) puis pourra être compilé, testé et déployé. L'itération suivante pourra alors commencer.

Conclusion

Adopter MDA, c'est admettre qu'il n'y aura consensus ni sur les plates formes, ni sur les Systèmes d'Exploitation, ni sur les réseaux, ni sur les architectures logicielles ; c'est se protéger contre le fait qu'à tout moment risque d'apparaître sur le marché la nouvelle solution miracle qui développera tous les arguments commerciaux possibles cherchant à remettre en cause nos pratiques antérieures.

En revanche, grâce à MDA, on impose aux fournisseurs de s'inscrire dans une pratique permettant les échanges avec notre environnement logiciel.

Adopter MDA, c'est imposer à ses fournisseurs une démarche standardisée, unifiée, fondée sur une syntaxe communément admise. C'est s'imposer une modélisation fonctionnelle préalable, d'un niveau d'abstraction suffisant pour exprimer le métier de notre entreprise de manière rigoureusement indépendante de toute solution technique. C'est clairement identifier dans son patrimoine logiciel ce qui constitue le cœur du métier de l'entreprise, le cœur de son Système d'Information.
Mais pour pouvoir adopter MDA, il faut disposer d'une vision très claire de son Système d'Information et de la répartition des responsabilités entre les composantes. Cela implique une discipline stricte dans la gestion des fonctionnalités et des responsabilités confiées à chaque sous-système. Adopter MDA, c'est enrichir considérablement sa vision fonctionnelle en soumettant à celle-ci toute considération technique. ▲

antoine.clave@wanadoo.fr

[1] CORBA : Common Object Request Broker Architecture. Standard de gestion d'objets distribués, mis au point par l'OMG, dont l'objectif est de permettre à des applications développées dans des langages différents de communiquer même si elles ne sont pas sur la même machine.

Pour aller plus loin

Les fondamentaux avant tout :

- Le site de l'OMG bien sûr : www.omg.org

- SODIFRANCE est un acteur français proposant des outils : Model In Action et une démarche intéressante. Des versions d'évaluation sont bien entendu disponibles : www.model-in-action.fr

- ARC STYLER est un outil développé par Interactive Objects Software Gmbh : www.io-software.com

- L'Université de Nantes est un pôle de recherche important sur MDA, animé par plusieurs auteurs reconnus, dont Jean BEZIVIN : www.sciences.univ-nantes.fr

Quelques articles de La Lettre d'ADELI au sujet d'UML

- UML : vers un monde lisible - Dominique Vauquier - Lettre n°52 - Juillet 2003

- UML et la modélisation : qualités et défauts, principes de modélisation - Antoine Clave - Lettre n°53 - Octobre 2003

- UML - Dominique Vauquier - Lettre n°54 - Janvier 2004

- Les six impasses de la conception des processus : vers une alternative à l'approche fonctionnelle - Dominique Vauquier - Lettre n°55 - Avril 2004