Stratégie de test au sein du processus d'évolution d'architecture de Sodifrance

Laurent Garnier

CONSERVATOIRE NATIONAL DES ARTS ET METIERS
CENTRE REGIONAL DES PAYS DE LA LOIRE
CENTRE D'ENSEIGNEMENT DE NANTES

MEMOIRE

présenté en vue d'obtenir le

DIPLÔME D'INGENIEUR C.N.A.M.

en

INFORMATIQUE

par

Laurent GARNIER

Stratégie de test au sein du processus d'évolution d'architecture de Sodifrance

Soutenu le 16 décembre 2011

JURY

Présidente : Mme METAIS, professeur Cnam Paris
Membres : M. BRIAND, professeur Ecole Polytechnique Nantes
M. BELLEIL, tuteur Cnam, professeur Université de Nantes
M. BRETON, tuteur entreprise, co-directeur DTOP, Sodifrance
M. PACAUD, architecte technique, DTOP, Sodifrance

Sommaire

Acronymes

ADM :	Architecture Driven Modernization (Modernisation dirigée par les modèles)
ANT :	Architecture N Tiers
API :	Application Programming Interface (Interface de programmation)
ASTM :	Abstract Syntax Tree Metamodel (métamodèle d'arbre de syntaxe abstrait)
BU :	Business Unit (Unité d'affaires)
CDO :	Connected Data Objects (Objets de données connectés)
CIM :	Computation Independent Model (Modèle des exigences)
DAO :	Data Access Object (Objet d'accès aux données)
HTML :	HyperText Markup Language (Langage de balises Hyper texte)
JAR :	Java ARchive (Archive Java)
KDM :	Knowledge Discovery Metamodel (Métamodèle de découverte de la connaissance)
MDA :	Model Driven Architecture (Architecture dirigée par les modèles)
MDE :	Model Driver Engineering (Ingénierie dirigée par les modèles
MIA :	Model In Action (Modèles en action)
OMG :	Object Management Group (groupe de standardisation des technologies objet)
ORM :	Object Relational Mapper (mapping (correspondance) objet / relationnel)
PDM :	Platform Definition Model (Modèle de description de la plate-forme)
PIM :	Platform Independent Model (modèle indépendant de la plate-forme)
POC :	Proof Of Concept (Preuve de concept)
PSM :	Platform Specific Model (modèle spécifique à la plate-forme)
SGBDR :	Système de Gestion de Base de Données Relationnelle
SLOC :	Source Line Of Code (Nombre de ligne de code source)
SQL :	Structured Query Language (Langage de requête structure)
UCBT :	Use Case Base Testing (Tests bases sur les cas d'utilisation)
UML :	Unified Modeling Language (langage de modélisation unifié)
XMI :	XML Metadata Interchange (standard pour l'échange de métadonnées)
XML :	eXtended Markup Language (langage à balises extensible)

Table des illustrations

Remerciements

Tout d'abord, je tiens à remercier MM. Faia et Breton, instigateurs puis promoteurs de l'idée de ce cursus d'ingénieur au CNAM en parallèle à mon activité professionnelle au sein de Sodifrance. Je remercie aussi toute l'équipe d'évolution d'architecture de Sodifrance dans laquelle j'évolue actuellement pour son soutien et son aide, aussi bien morale que technique. Merci aussi à M. Belleil d'avoir accepté d'être mon tuteur tout au long de ce mémoire, et aux différents intervenants du CNAM de Nantes pour leurs prestations de qualité.

Pour finir, je tiens à remercier tout particulièrement ma femme, pour toutes ces soirées où elle s'est retrouvée seule à gérer les enfants, la maison, pendant que son mari étudiait et rédigeait les lignes qui vont suivre. Sans elle non plus, rien n'aurait été possible, en fait sans elle, rien n'a vraiment de sens. Merci Valérie.

Avertissement

De nombreux termes de ce mémoire sont mentionnés en langue anglaise. Bien que, dans la plupart des cas, une traduction en français soit possible, les termes anglais peuvent être tout de même utilisés si la traduction (par une expression courte) dénature le concept désigné. Les mots ou expressions en langue anglaise sont écrits en italique.

1 INTRODUCTION

Le sujet de ce mémoire, « Stratégie de test au sein du processus d'évolution d'architecture de Sodifrance », a été défini d'un commun accord entre mes responsables et moi-même. Il comprend deux thèmes principaux qui sont la cartographie des tests d'une part, et l'automatisation des tests d'autre part. Ce sujet n'a pas été choisi sans raison, il vise à couvrir un vrai manque dans le processus de migration. En effet, les tests sont totalement absents de l'automatisation. Réalisés de façon entièrement manuelle, ils aboutissent à des consommations très importantes en phase de validation.

Le plan général de ce mémoire s'articule autour d'une première partie qui présente l'environnement et le processus d'évolution d'architecture de Sodifrance. La deuxième partie déclinera l'état de l'art, les travaux réalisés autour des deux objectifs fixés, et leurs utilisations dans les projets en cours. La troisième partie présentera les travaux connexes. Elle sera suivie d'une conclusion indiquant l'état de la réalisation des objectifs et donnera un aperçu des orientations futures.

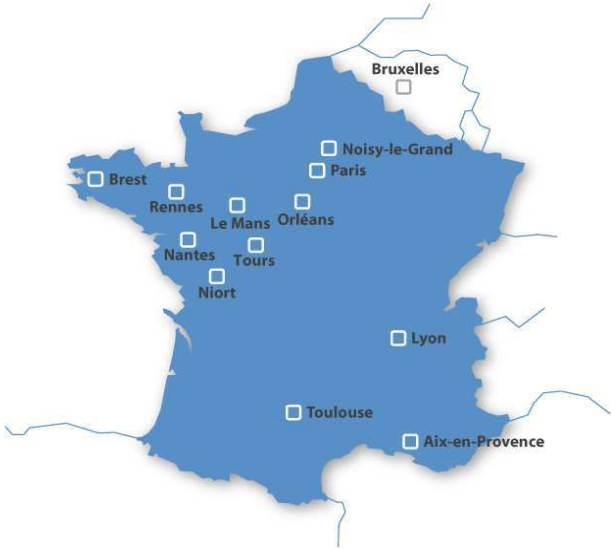
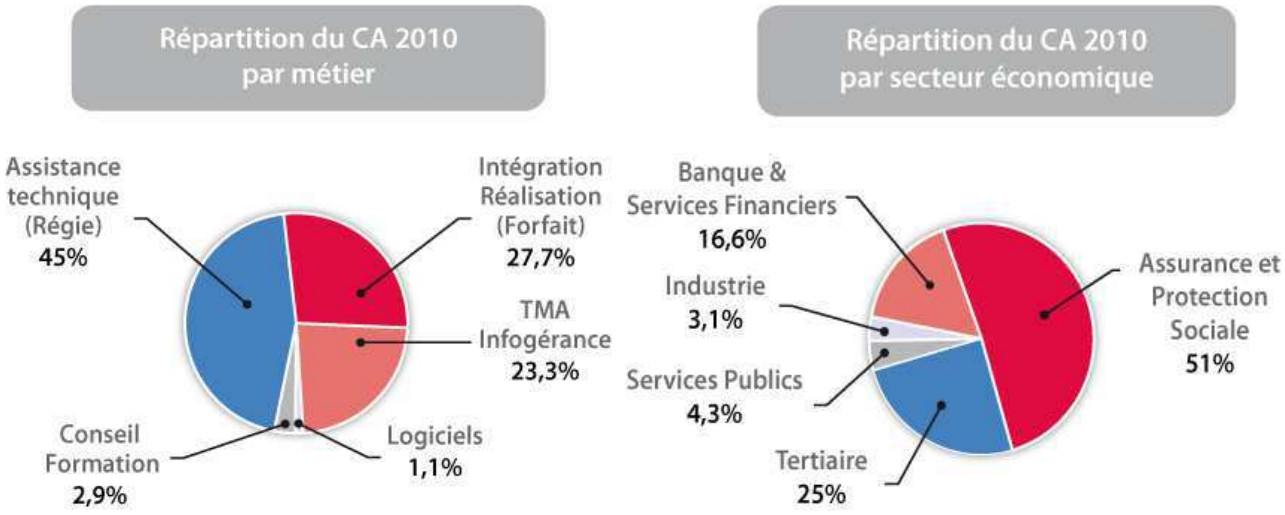


Figure 1 : Répartition des agences Sodifrance en France et en Belgique.



2 L'ENVIRONNEMENT

Avant d'entrer dans le vif du sujet, je vais présenter dans ce chapitre successivement l'entreprise Sodifrance, puis le processus « classique » d'évolution d'architecture utilisé lors des migrations d'applications.

2.1 Présentation de l'entreprise

Sodifrance a été fondée en 1986 par M. Francis Mazin. Partenaire historique du secteur bancaire, l'entreprise s'est orientée en 1992 vers la transformation automatisée de systèmes d'information. L'année 1999 marque un tournant au niveau organisationnel, en effet cette année survient le décès du fondateur, et l'entrée en bourse sur le second marché. L'équipe dirigeante s'organise alors autour d'un conseil de surveillance et d'un directoire, actuellement composé de :

- M. Franck Mazin : Président du directoire
- M. Yves Lennon : Directeur Général
- M. Frédéric Rivière De Précourt : Directeur administratif et financier
- Mme Anne-Laure Mazin : Directeur marketing et communication

Le groupe compte aujourd'hui 870 collaborateurs répartis en France, Belgique (cf. Figure 1) et Tunisie. Création en 2004 de la filiale Mia Software afin de capitaliser son savoir-faire dans le domaine de l'ingénierie des modèles (*Model Driven engineering*, MDE), et de commercialiser la suite d'analyse et de modernisation d'architecture MIA Studio.
Les chiffres clés de l'année 2010 sont :

- 24 ans d'existence
- 15 ans d'expérience dans l'automatisation des évolutions d'architectures
- 63 M d'euros de chiffre d'affaire
- Près de 70% du chiffre d'affaire réalisé dans les secteurs banque et assurance.

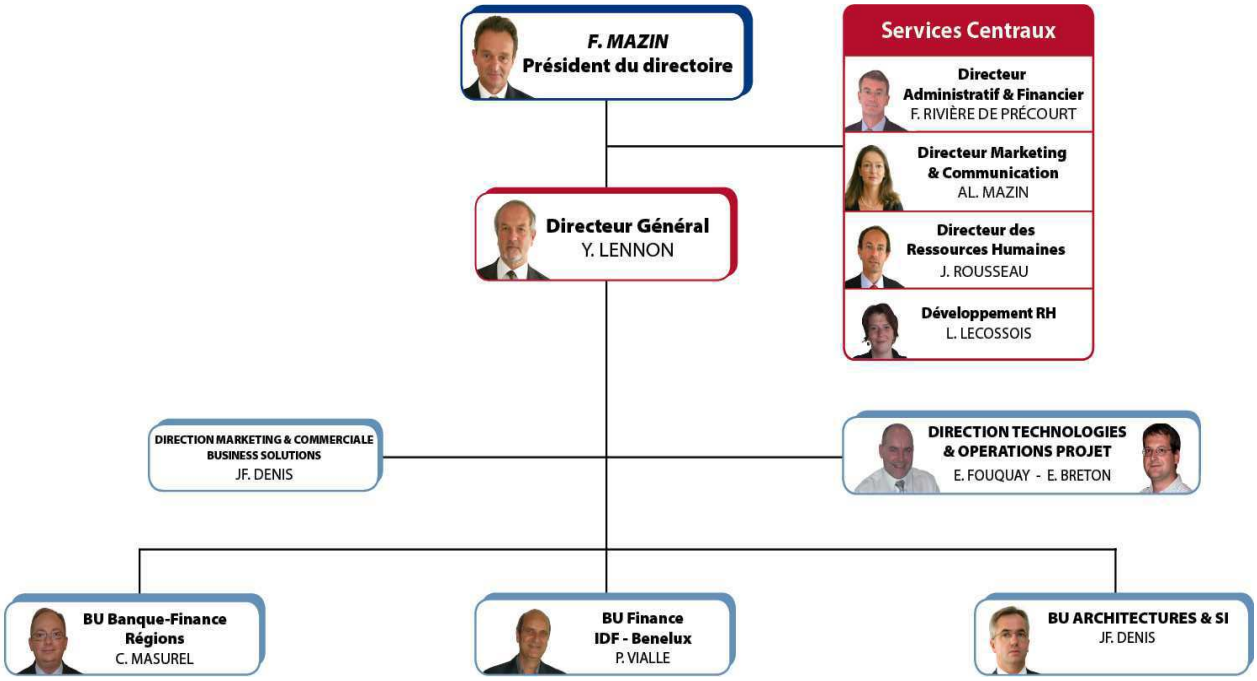


Figure 2 : Organisation fonctionnelle 2011 (source Sodifrance)

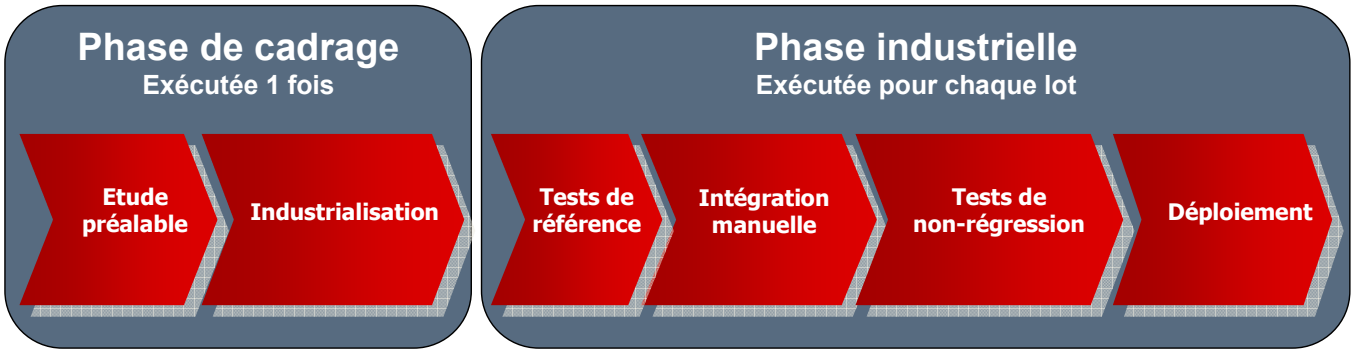


Figure 3 : Processus global de migration (source Sodifrance)

La Figure 2 présente l'organisation fonctionnelle du groupe. Le découpage matriciel s'articule autour des unités d'affaires (*Business Unit*, BU) réparties par région ou par métier pour la dimension verticale, et de départements techniques ou marketing pour la dimension transversale.

Je travaille actuellement dans l'équipe d'évolution d'architecture du département *Direction Technologies et Opérations Projet* (DTOP) sous la responsabilité de M. Breton. Cette équipe réalise l'ensemble des phases de cadrage lors des migrations.

2.2 Le processus d'évolution d'architecture

Avant de présenter la problématique proprement dite de ce mémoire, il convient de définir le contexte dans lequel il s'est déroulé. Comme indiqué précédemment, j'appartiens à l'équipe d'évolution d'architecture de Sodifrance. Je devais donc réussir à intégrer mes travaux dans le processus de migration existant.

2.2.1 Présentation générale du processus de migration

Le processus global de migration est divisé en deux grandes parties :
La phase de cadrage, qui permet de définir précisément ce qui va être fait :

- étude de la, ou des applications sources
- étude de l'architecture cible
- définition des lots de migration
- adaptation de l'outillage (*MIA Transformation*[1], *MIA Generation*[2])
- vérification de la validité de la solution par la réalisation d'un POC (*Proof Of Concept*)

Une fois la phase de cadrage terminée, l'ensemble des applications à migrer passe par la phase industrielle. Au cours de cette phase, nous trouvons:

- les tests de référence
- l'intégration manuelle du code issu de la génération (rendre le code compilable, corriger manuellement ce qui n'a pu être généré directement, tests unitaires).
- les tests de non régression (validation des résultats des tests de références de l'application source sur l'application migrée).
- le déploiement sur le site du client

[1] MIA Transformation : outil de transformation de modèles. http://www.mia-software.com/.../mia-transformation
[2] MIA Generation : outil de génération de code à partir de modèles. http://www.mia-software.com/.../mia-generation

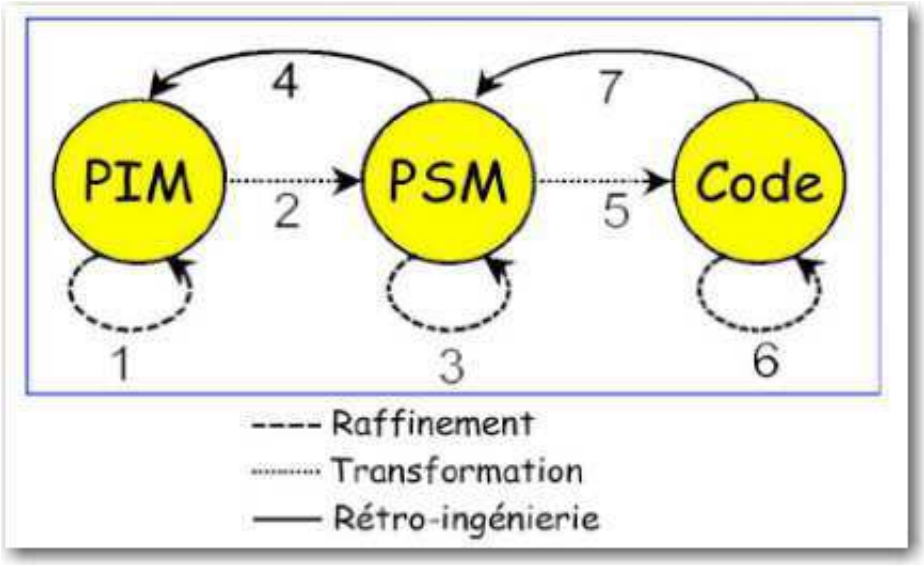


Figure 4 : Les transformations des modèles MDA (Villemin 2011, p.12)

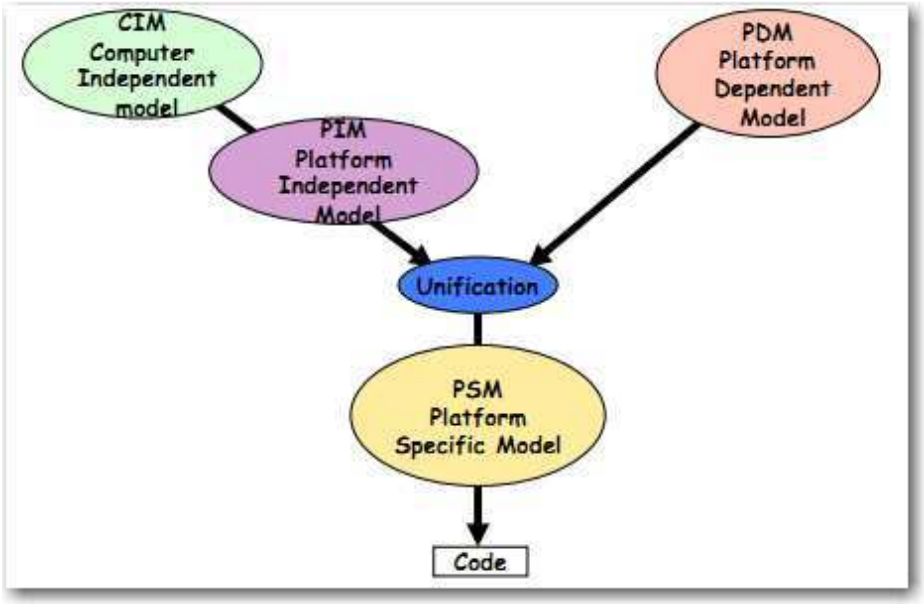


Figure 5 : Unification *PIM* et *PDM* pour produire le *PSM* puis le code (Villemin 2011, p.14)

La seule façon de valider une migration est de passer les scénarios des tests de référence, réalisés sur l'application source, sur l'application migrée. De cette manière, on peut garantir que la migration n'a pas introduit de régression et est conforme à l'application source, sur le périmètre des tests de référence bien entendu. C'est là que l'on s'aperçoit de l'importance de la couverture des tests de référence. En effet, s'ils ne couvrent qu'un petit pourcentage de l'application source, toutes les parties non couvertes risquent potentiellement de contenir des erreurs sur l'application cible.

2.2.2 Présentation du processus de migration industrielle

Afin de définir la phase de migration industrielle, il convient d'approfondir quelques notions.
Commençons par l'architecture dirigée par les modèles (*Model Driven Architecture,* MDA). Selon M. Villemin (Villemin 2011) « L'initiative d'architecture dirigée par les modèles de l'OMG[3] "*Model Driven Architecture*" (*MDA*) est motivée par le besoin de réduire les tâches de reconception des applications (nécessitées, entre autre, par l'évolution constante des technologies informatiques) ». Le *MDA*, qui est une démarche de développement, répond tout à fait à ce besoin, car il permet « la séparation des spécifications fonctionnelles des spécifications d'implantations sur une plate-forme donnée » (Villemin 2011). Cette séparation s'effectue avec l'utilisation de modèles différents décrivant :

- un modèle des exigences (*CIM : Computational Independent Model*)
- un modèle de traitements orientés métier (*PIM : Platform Independent Model*)
- un modèle d'architecture technique (*PDM : Platform Dependent Model*)
- un modèle d'implantation pour une plate-forme spécifique (*PSM : Platform Specific Model*).

Le passage d'un modèle à l'autre s'effectue par transformations successives, soit du modèle le plus abstrait jusqu'au code, soit du code en « remontant » jusqu'à un modèle abstrait (retro-ingénierie). La Figure 4 illustre ces différentes transformations. La Figure 5 quant à elle, illustre l'unification d'un modèle indépendant de la plate-forme (*Platform Independent Model*, PIM), par exemple une gestion de stock, avec un modèle dépendant de la plate-forme (*Platform Dependent Model*, PDM), par exemple un site web. Leur union produit un modèle spécifique à la plate-forme (*Platform Specific Model*, PSM), donc dans ce cas un site web de gestion de stock, qui lui-même permet de produire du code.

[3] OMG : *Object Management Group*. http://www.omg.org/

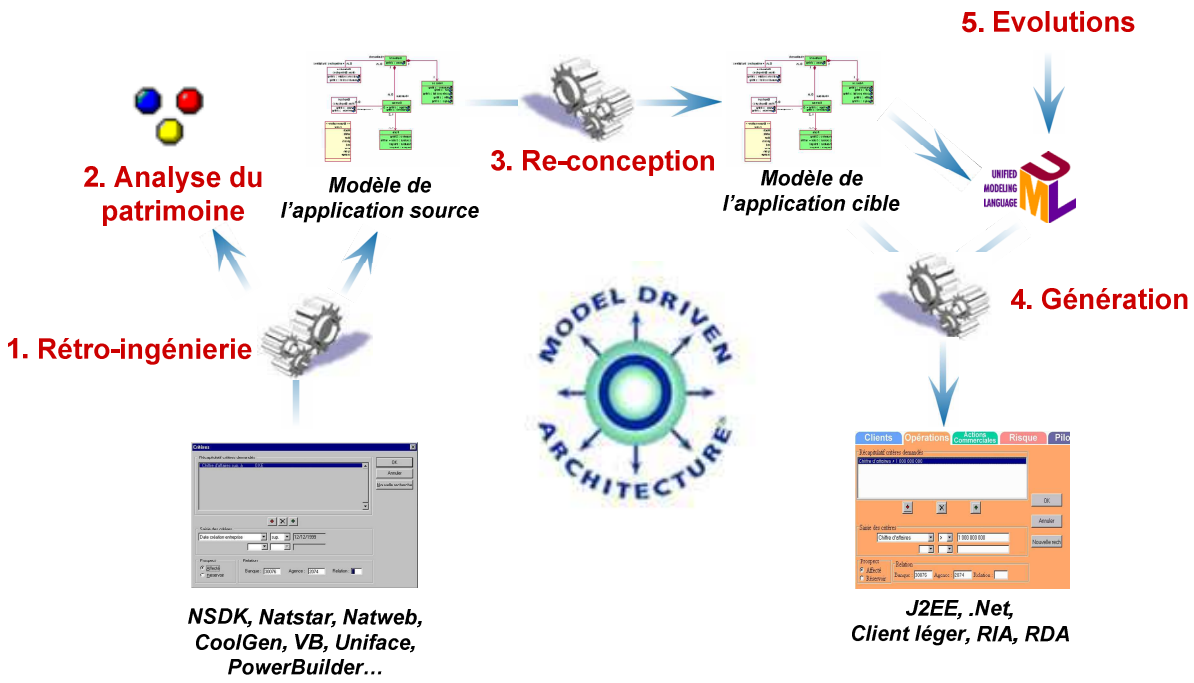


Figure 6 : Le processus d'évolution d'architecture (source Sodifrance)

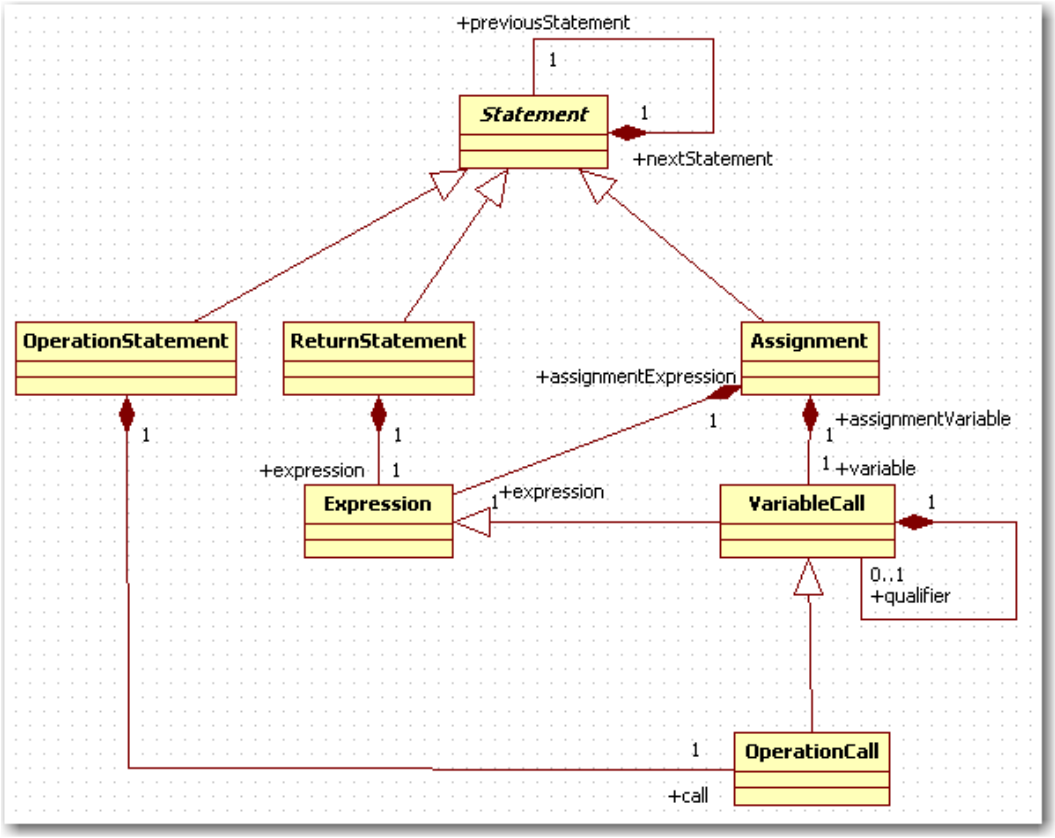


Figure 7 : Extrait du métamodèle architecture n-tiers (*ANT*)

Ce sont ces différents modèles que l'on retrouve dans le processus d'évolution d'architecture lors des phases de transformation (cf. Figure 6 étape 3).

Les sources sont analysées par des outils de d'analyse syntaxique développés par l'équipe R&D Sodifrance (cf. Figure 6 étape 1) et alimentent :

- un modèle basé sur un métamodèle spécifique au langage.
- un outil permettant une analyse du patrimoine[4] (cf. Figure 6 étape 2).

On trouve ici une première correspondance avec la retro-ingénierie du *MDA*.

Lors de l'étape 3, il y a une série de transformations qui sont effectuées avec l'outil *MIA Transformation*, à partir du modèle de l'application source, pour obtenir un modèle générique *Architecture N Tiers* (ANT[5]).

Le métamodèle ANT est un métamodèle propre à Sodifrance, se rapprochant d'UML pour la définition des classes ou paquetages, mais permettant en plus de représenter l'ensemble des informations d'un programme, dont notamment l'algorithmie ou les interfaces graphiques. La Figure 7 nous détaille un extrait de ce métamodèle.

Toujours durant l'étape 3, de nouvelles transformations ANT vers ANT sont appliquées pour façonner les modèles en fonction de l'architecture cible définie par le client. Il est assez rare que les clients ne sachent pas vers quoi ils veulent migrer leurs applications. Dans ce cas, nous leur proposons un langage et une architecture cible. Mais le plus souvent, ils savent exactement vers quoi ils veulent aller. Certains ont même déjà développé des applications vers cette cible, d'autres ont poussé l'exercice jusqu'à produire leurs propres briques logicielles de base (*Framework*). A nous d'adapter notre outillage pour atteindre la cible fixée.

[4] MIA-Insight : http://www.mia-software.com/produits/mia-insight/

[5] Le métamodèle ANT a été conçu par les équipes R&D de Sodifrance. Comme UML, il permet de définir les applications en termes de classes, écrans ou package. Mais il permet aussi de stocker les informations d'algorithmie. Etant insensible au langage source, c'est devenu le métamodèle de générique de Sodifrance. La plupart des transformations et des générations s'effectuent vers ou à partir de ce dernier.

Par exemple, nous pouvons mettre en place des règles pour créer des nouveaux composants comme des paquetages, des classes, des variables ou des fonctions, mais aussi créer des instructions à l'intérieur de ces fonctions. En réalité, on trouvera au sein de cette troisième étape trois phases elles-mêmes assez distinctes, et qui là encore rejoignent les principaux modèles du *MDA*.

Dans un premier temps, on va chercher à effacer au maximum les références au langage et à l'architecture source pour obtenir un modèle « harmonisé ». On tente autant que possible de passer du *PSM* au *PIM*.

Ensuite, on va ajouter des informations indépendantes de l'architecture ou du langage cible sur les éléments déjà présents dans le modèle. Ceci afin de faciliter soit les transformations suivantes, soit la génération (*PIM* vers *PIM,* ou *PIM* vers *PSM* vers code).

Et pour terminer, on va ajouter toutes les informations nécessaires à la génération vers la solution cible (*PIM* vers *PSM*), avec cette fois-ci des données liées au langage, au choix d'architecture et bien entendu aux exigences spécifiques du client.

C'est aussi pendant l'étape 3 que l'on peut extraire des informations nécessaires pour alimenter des modèles UML génériques ou spécifiques au client. Cela trouve tout son sens si le client est déjà dans une démarche de développement dirigé par les modèles et maintient ses applications de cette façon (cf. Figure 6 étape 5).

Durant la phase de génération (cf. Figure 6 étape 4), l'outil *MIA Generation* parcourt le modèle cible obtenu par les transformations, et produit du code en fonction de scripts positionnés au niveau des objets du métamodèle. On est clairement dans la phase *PSM* vers code de la démarche MDA.

2.2.3 Les tests dans le processus de migration

Dans le cadre de ces projets d'évolution d'architecture, le résultat de la migration est validé en vérifiant l'iso-fonctionnalité entre l'application d'origine et l'application migrée. Pour cela, on contrôle que les tests sur les applications sources et cibles donnent les mêmes résultats. La charge consacrée à ces tests est donc loin d'être négligeable, et ce d'autant plus si l'on considère que ces activités sont menées de manière manuelle, alors que la migration de code en elle-même est en grande partie automatisée.

L'enjeu de ce travail de mémoire est donc d'explorer différentes pistes permettant d'optimiser la gestion des tests et de mettre en œuvre une série d'outils afin d'atteindre cet objectif.

Dans le cadre d'un processus de modernisation d'architecture, les tests sont organisés de la manière suivante :

- Tests de référence : les tests de référence ont pour but de définir le référentiel de test qui permettra d'établir l'iso-fonctionnalité. Ils sont établis sur l'application source.
- Tests unitaires : validation de manière indépendante des composants issus de la migration. Les tests unitaires sont réalisés lors de la phase d'intégration manuelle du code.
- Tests de non-régression : validation de l'iso-fonctionnalité de l'application migrée.

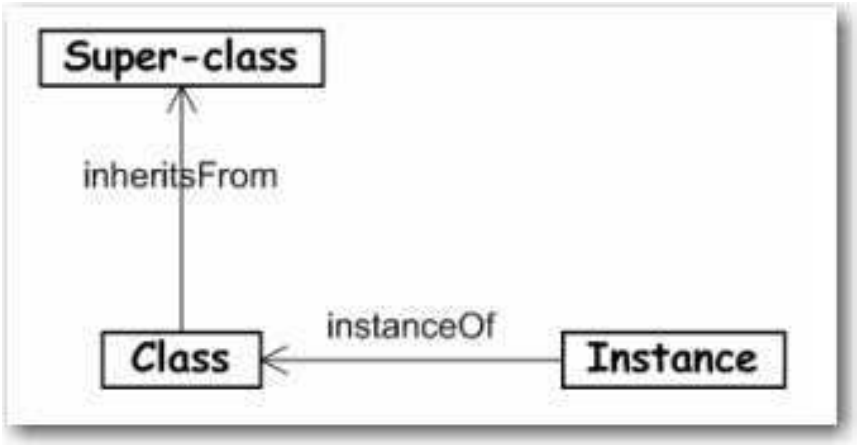


Figure 8 : Notions de base en technologie des objets (Bézivin 2004)

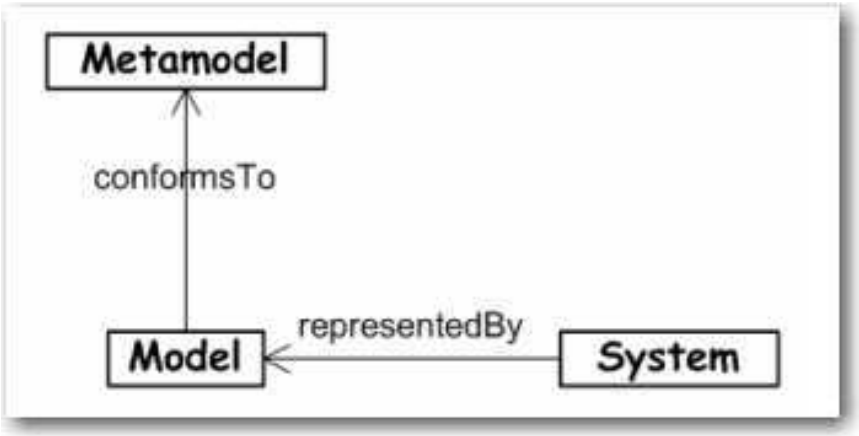


Figure 9 : Notions de base en ingénierie des modèles (Bézivin 2004)

3 LE TRAVAIL REALISE

Après un échange avec mon tuteur et notre hiérarchie, il a été convenu de traiter les problématiques de cartographie et d'automatisation des tests. En effet, c'est à ce niveau que se trouve le plus gros effort manuel et qu'il y a le plus d'incompréhension entre la vision fonctionnelle des clients et la nôtre qui est plutôt technique. On peut noter un élément très important puisqu'il a guidé l'ensemble de ces travaux : le résultat doit être exploitable par l'ensemble de notre chaîne d'évolution d'architecture. On verra par la suite que cela a orienté certains choix. Avant d'aborder le travail réalisé, faisons un tour d'horizon de l'état de l'art des principaux sujets.

3.1 Etat de l'art

3.1.1 Model Driven Engineering, Model Driven Architecture

L'architecture dirigée par les modèles (*Model-Driven Architecture*, *MDA*), présentée au § 2.2.2, est « une variante particulière d'une tendance plus générale nommée ingénierie des modèles » (*Model-Driven Engineering*, *MDE*). Tout comme un des principes de base de la technologie objet est « tout est objet » (cf. Figure 8), le principe de base du *MDE* est « tout est modèle » (cf. Figure 9) (Bézivin 2004).

Toujours selon M. Bézivin (Bézivin et al. 2007), le *MDE* est décrit en se basant sur deux règles fondamentales :

- Tout type de système peut être représenté dans un modèle.
- Un modèle doit être syntaxiquement conforme à un métamodèle.

Il faut bien faire la part des choses entre le système, qui se base sur le monde réel, et le modèle, qui est d'un niveau plus abstrait, celui de la modélisation.

Une fois ces notions précisées, les auteurs (Bézivin et al. 2007) déclinent les grands domaines d'applications du *MDE* qui reprennent ceux du *MDA* :

- *Forward engineering*, quand le système est créé à partir du modèle.
- *Reverse engineering*, quand le modèle est créé à partir du système.
- *Models at run-time*, quand le modèle coexiste avec le système qu'il représente.

Ce sont ces grands domaines que l'on retrouve dans le processus d'évolution d'architecture de Sodifrance. La retro-ingénierie correspond à la phase d'analyse du code source et a pour objectif d'alimenter un métamodèle spécifique au langage source, décrit comme le modèle initial (cf. Figure 6 étape 1).

La transformation de modèle, réalisée à l'aide de *MIA Transformation*, correspond aux transformations permettant dans un premier temps de passer du métamodèle initial à un métamodèle dérivé. Dans notre cas, la cible est un métamodèle générique, le métamodèle ANT (cf. Figure 6 étape 3). Ensuite, ce modèle ANT est à nouveau transformé pour répondre au mieux aux exigences de l'application cible (cf. 2.2.2). La dernière phase (cf. Figure 6 étape 4) est celle de génération. Elle correspond au *Forward Engineering*. L'outil *MIA Generation* utilise le modèle ANT comme modèle source et génère du code en fonction de scripts définis sur les objets du métamodèle.

On retrouve aussi ce découpage en trois phases dans le processus d'alimentation de la cartographie qui sera approfondi un peu plus loin. De manière simplifiée, il y a la phase de rétro-ingénierie qui alimente un métamodèle spécifique. Ensuite une transformation permet de passer de ce métamodèle au métamodèle de cartographie. Et pour terminer, des services permettent de générer du code (scripts de rejeu de test) à partir du métamodèle de cartographie.

3.1.2 Cartographie d'application

Dans le cadre de la cartographie applicative, on peut trouver plusieurs objectifs : avoir une vision de haut niveau de ses applications, ou au contraire, avoir une connaissance très fine de celles-ci en descendant jusqu'aux lignes de codes qui les composent. Le groupe de travail modernisation d'architecture dirigée par les modèles (*Architecture-Driven Modernization task force*, ADM[6]) a défini plusieurs métamodèles pour répondre à ces besoins, les deux principaux sont le métamodèle de découverte de la connaissance (*Knowledge Discovery Metamodel*, KDM[7]) et le métamodèle d'arbre syntaxique abstrait (*Abstract Syntax Tree Metamodel*, ASTM[8]).

KDM permet d'obtenir une vision à gros grain des composants des applications, comme les classes, écrans ou les données, mais ne descend pas en-dessous des méthodes. ASTM quant à lui, offre une vision à grain fin de l'application et permet de représenter l'ensemble de l'algorithmie des programmes. ASTM est prévu pour être un complément de KDM afin d'avoir une vision d'ensemble de son parc applicatif.

On retrouve ces deux niveaux de représentation dans les métamodèles de Sodifrance. ASTM correspond à nos métamodèles spécifiques à chaque langage source et au métamodèle ANT. Ils permettent de modéliser cent pour cent de l'application source. A contrario, le métamodèle de cartographie est du niveau de KDM et se borne à modéliser les éléments importants (classes, écrans, fonctions, etc.) ainsi que les relations qui les unissent. D'ailleurs, notre métamodèle de cartographie s'inspire fortement de KDM (nommage, classe de base), mais nous n'avons pas suivi l'ensemble des recommandations de l'ADM. Nous avions des besoins d'extension de la cartographie pour prendre en compte des sujets divers comme les tests ou le suivi d'intégration (cf. Figure 15). Toutefois, cela n'a pas été un critère de choix décisif, car MM. Dehlen et al. (Dehlen et al. 2008) démontrent que l'on peut aussi étendre KDM selon ses besoins.

[6] ADM : Architecture Driven Modernization (Object Management Group 2011a)
[7] KDM : Knowledge Discovery Metamodel, (Object Management Group 2010)
[8] ASTM : Abstract Syntax Tree Metamodel (Object Management Group 2011b)

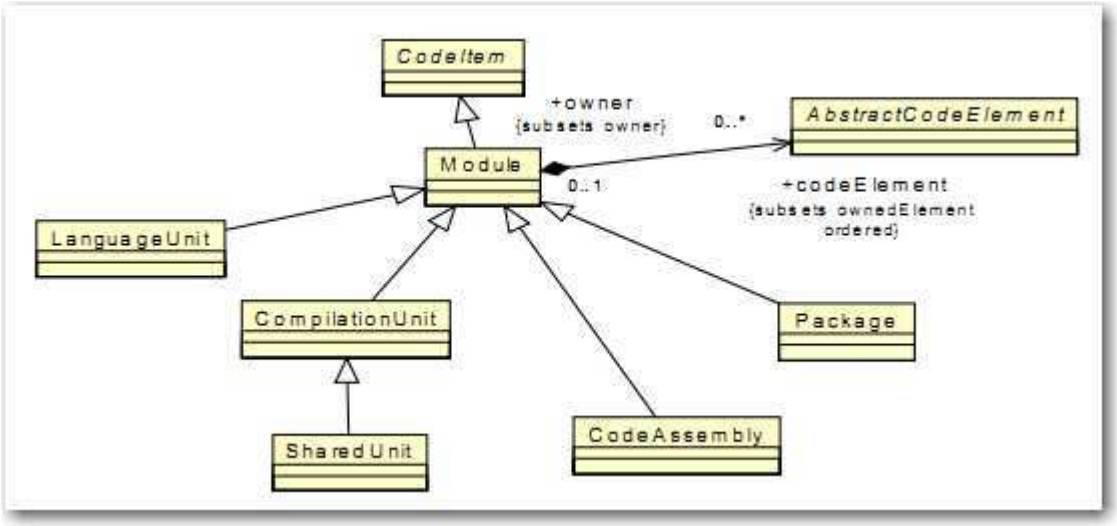


Figure 10 : Diagramme de classe extrait du métamodèle KDM

(Object Management Group 2010, p.69)

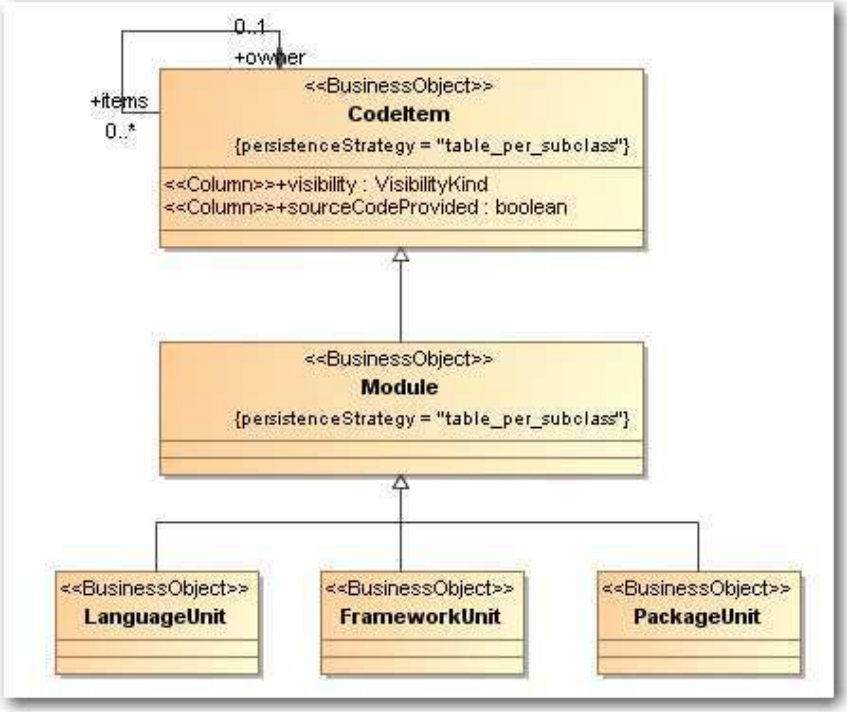


Figure 11 : Diagramme de classe extrait du métamodèle « Migration Platform »

(source Sodifrance, partie du métamodèle « Migration Platform »)

En réalité, les critères principaux qui ont orienté le choix de définir un métamodèle propriétaire sont d'ordre un peu plus pratique :

- La simplification du métamodèle. Nous n'avons utilisé que ce qui nous servait réellement. Par exemple, les informations concernant la compilation ne sont pas présentes dans notre métamodèle. On remarque bien les similitudes entre les Figure 10 et Figure 11, mais effectivement, la classe « *CompilationUnit* » ou sa sous-classe « *SharedUnit* » ne sont pas présentes dans notre métamodèle.
- Nous n'avons pas réussi à obtenir une manière fiable d'enregistrer nos modèles, parfois très volumineux (maquettes réalisées avec CDO[9] non concluantes). Donc nous avons opté pour le stockage des informations dans une base de données relationnelle accessible par le biais de l'ORM[10] *Hibernate*[11]. Du fait de cette simplification du métamodèle et de la stratégie de persistance appliquée (une table par classe), les jointures sont elles aussi plus simples, et surtout avec un nombre d'auto-jointures limité.
- Le résultat de la cartographie doit être disponible le plus simplement possible pour le reste de la chaîne d'évolution d'architecture. Dans notre cas, une archive Java (Jar) incluant l'ensemble de la couche DAO *Hibernate* est mise à disposition des outils. Cela répond tout à fait à ce besoin.
- Enfin, nous sommes très libres dans nos choix concernant l'évolution du métamodèle. Un inconvénient est que l'on s'isole des standards et du reste de la communauté. Mais en contrepartie, cela nous permet de protéger un certain savoir-faire.

[9] CDO : Connected Data Objects (Eclipse project CDO 2011)
[10] ORM : Object-relational mapping (Crucianu s. d., p.173)
[11] Hibernate : cadre de développement libre pour la correspondance objet-relationnel (Crucianu s. d., p.173)

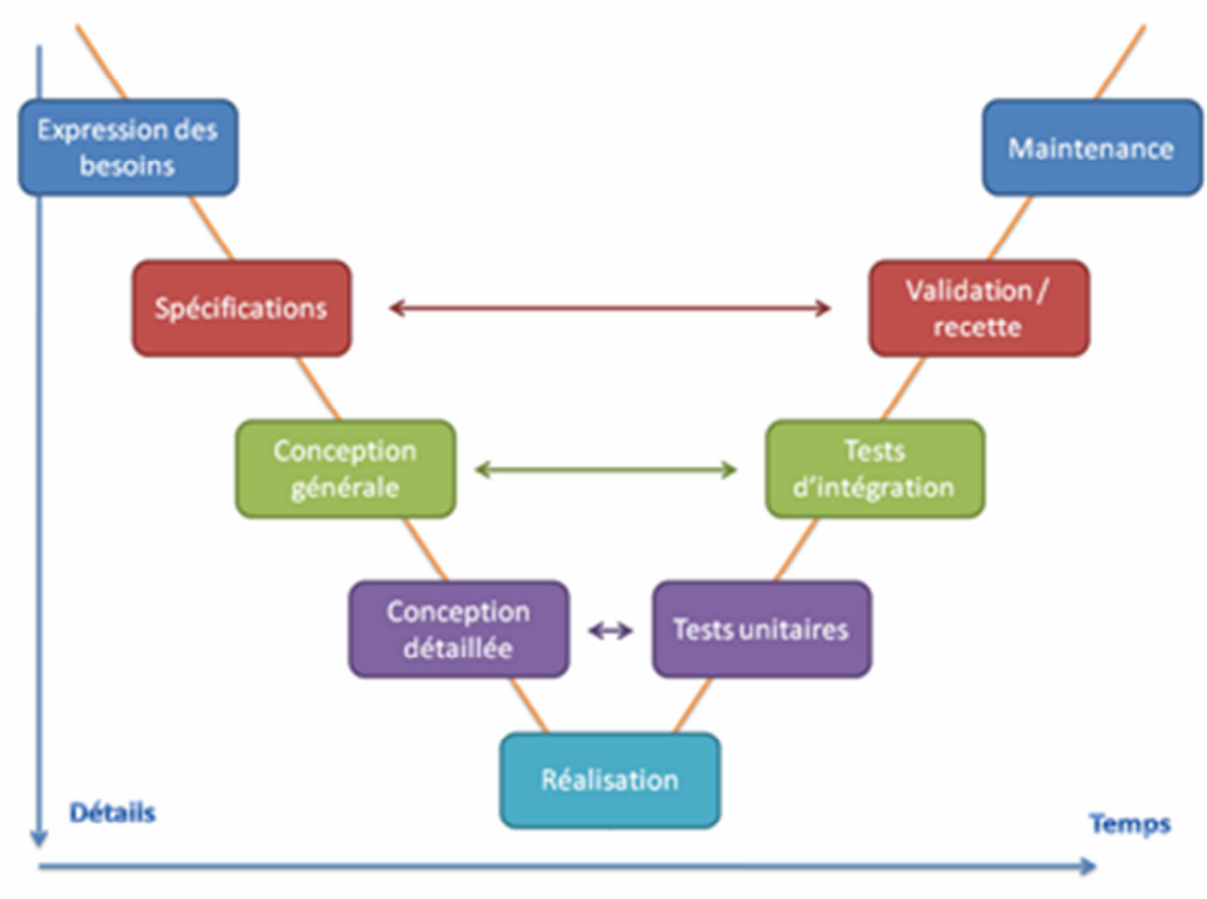


Figure 12 : Le modèle en V (Mirman 2011)

3.1.3 Test

3.1.3.1 Généralités

M. Félix donne une définition très claire de ce qu'est un test (FELIX 2011) : « Toute fabrication de produit suit les étapes suivantes : conception, réalisation et test. Avec le test, on s'assure que le produit final correspond à ce qui a été demandé… ».

Selon l'institut des ingénieurs électriciens et électroniciens (The Institute of Electrical and Electronics Ehgineers 345 East 47th Street, New York, NY 10017, USA s. d.), « le test est un processus manuel ou automatique, qui vise à établir qu'un système vérifie les propriétés exigées par sa spécification, ou à détecter des différences entre les résultats engendrés par le système et ceux qui sont attendus par la spécification ».

M. Gianas affirme que quels que soient les modèles de conception adoptés, en cascade, en Y ou en V comme le montre par exemple la Figure 12, on retrouve tout au long du processus de création du produit les principaux niveaux de tests (Régis-Gianas 2010):

- Test unitaire: le test de composants logiciels individuels.
- Tests d'intégration : tests effectués pour montrer des défauts dans les interfaces et interactions de composants ou systèmes intégrés.
- Test d'acceptation : test formel en rapport avec les besoins, exigences et processus métier, conduit pour déterminer si un système satisfait ou non aux critères d'acceptation et permettre aux utilisateurs, clients ou autres entités autorisées de déterminer l'acceptation ou non du système.
- Test de régression : tests d'un programme préalablement testé, après une modification, pour s'assurer que des défauts n'ont pas été introduits ou découverts dans des parties non modifiées du logiciel, comme suites des modifications effectuées. Ces tests sont effectués quand le logiciel ou son environnement est modifié.

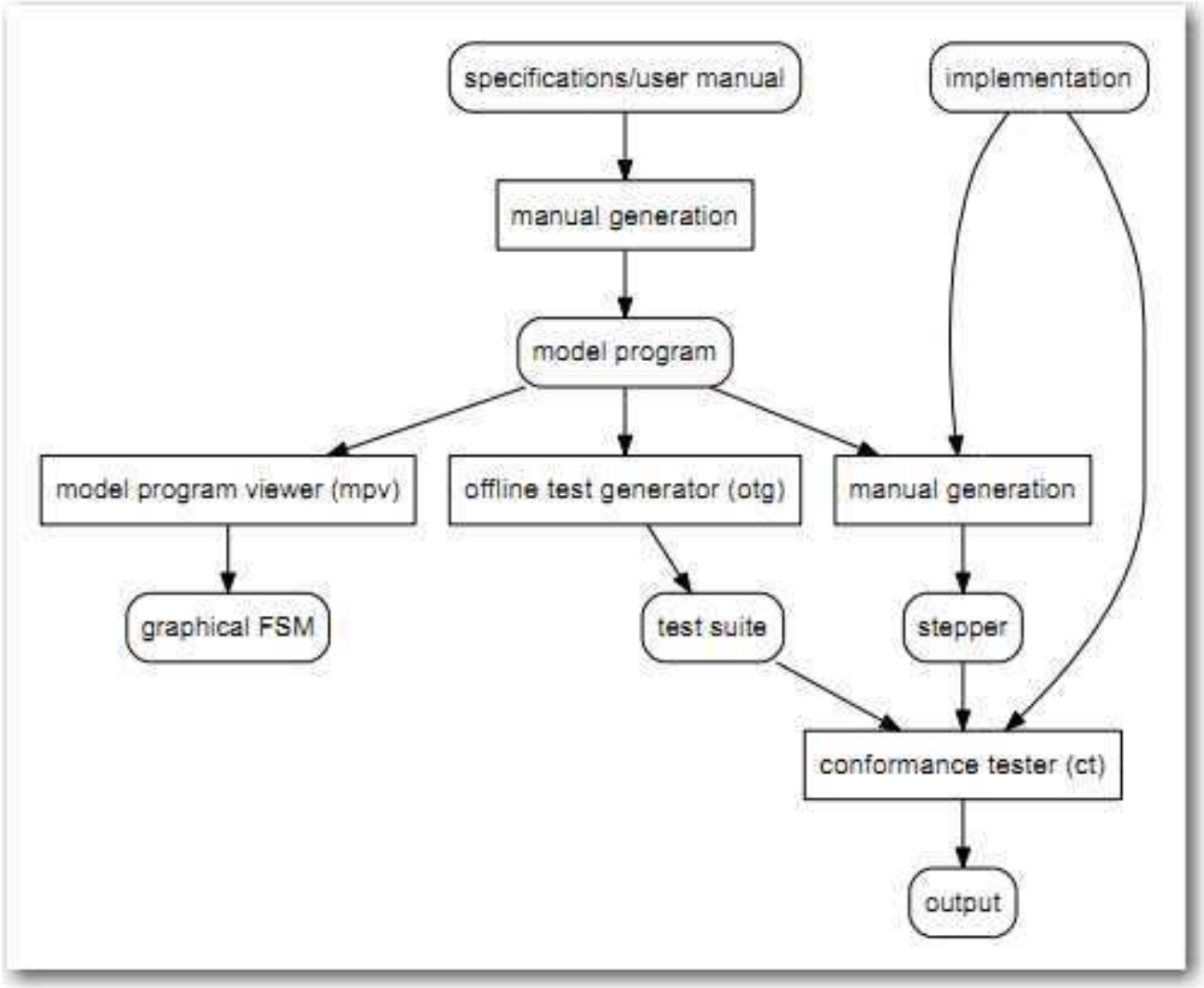


Figure 13 : Implémentation des tests avec NModel (Chinnapongse et al. 2009)

3.1.3.2 Tests basés sur les modèles (Model Based Testing)

De même que les modèles tendent à s'imposer dans le monde du développement, la part que représentent les tests basés sur les modèles commence à se faire de plus en plus visible. Assez longtemps délaissée, cette discipline fait désormais l'objet d'études sérieuses et approfondies. Bien que de nombreux documents traitent de ce sujet, ils sont souvent ciblés sur une technologie, un langage ou un environnement.

Dans leur article, MM. Hernandez et al. (Hernandez et al. s. d.) ciblent le contexte des applications web et des différentes technologies qui les entoure (HTML, JavaScript, Flash, ActionScript, etc.). Dans l'objectif de diminuer les coûts de maintenance des scripts de test, le principe d'automatisation des tests est acquis, et l'utilisation d'un métamodèle UML vise à ne pas avoir à réécrire l'ensemble des tests pour chaque environnement cible (utilisation des principes *MDE* de transformation des *PIM* en *PSM*). Cependant, tout en utilisant la notion de *PIM*, le métamodèle proposé est exclusivement à destination des environnements web.

Pour MM. Chinnapongse et al. (Chinnapongse et al. 2009), la cible adressée concerne les appareils portables (Smartphone, etc.). Globalement, le processus décrit consiste à

- définir un modèle comportemental de l'application (créé manuellement à partir de la documentation de l'appareil) avec l'interface de programmation (*Application Programming Interface*, API) NModel[12].
- le transformer en machine à états finis (cf. Figure 13, branche *model programmer view*, mpv)
- générer une suite de tests (cf. Figure 13, branche *offline test generator*, otg)
- exécuter les tests (cf. Figure 13, branche *conformance tester*, ct)

Ici aussi, le contexte adressé est restreint puisqu'il se limite aux appareils portables, et aux applications développées en *.Net*, qui peuvent être testée avec NModel.

[12] NModel : est un outil de test basé sur les modèles et un cadre de développement écrit en C#. http://nmodel.codeplex.com/

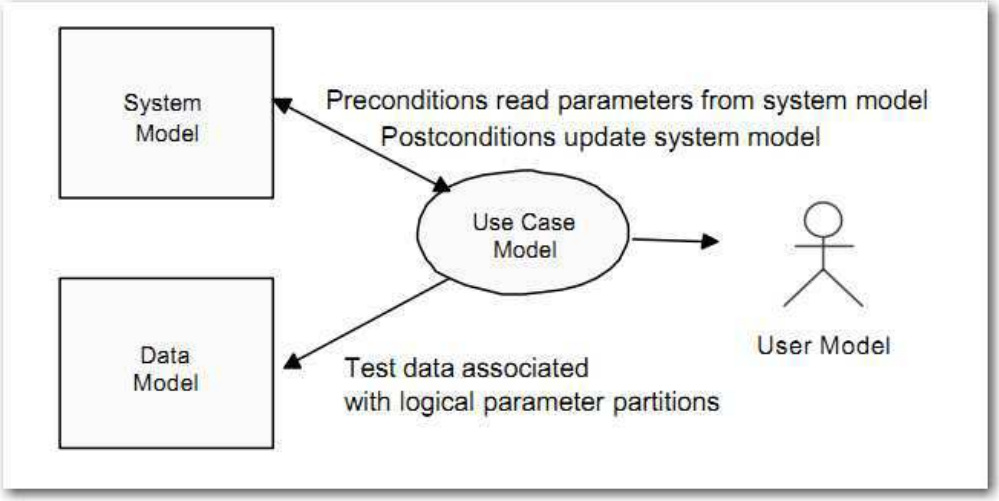


Figure 14 : Relations entre les sous modèles des tests basés sur les cas d'utilisation (*Use Case Base Testing*, UCBT) (Williams 2001)

M. Williams (Williams 2001) se sert des cas d'utilisation UML pour décrire le modèle de test et produire des suites de test. « De même que les classes et leurs concepts associés nécessitent un métamodèle bien défini afin de produire du code source, les cas d'utilisation ont besoin d'un métamodèle précis pour produire des cas de test valides et robustes. ». Lors des phases de modélisation, les annotations portées sur les cas d'utilisation déterminent le sous-modèle cible : modèle système, modèle de données, modèle des cas d'utilisation ou modèle utilisateur (cf. Figure 14). Avec cette technique de tests basés sur les cas d'utilisation (*Use Case Based Testing*, UCBT[13]), les modèles annotés servent à la génération de suites de test exploitables par l'outil TCBeans[14]. Les cibles de cet outil sont des programmes qui possèdent des interfaces de programmation (*Application Programming Interface*, API).

[13] UCBT : technique d'IBM pour générer des cas de test à partir de cas d'utilisations.
http://www.research.ibm.com/softeng/TESTING/ucbt.htm
[14] TCBeans : logiciel de test conçu pour élaborer, exécuter et organiser des tests fonctionnels.
https://www.research.ibm.com/haifa/projects/verification/gtcb/index.html

Une idée forte qui se dégage de ces différentes approches des tests basés sur les modèles, est la réponse à la question suivante : pourquoi faut-il mettre en œuvre des tests basés sur les modèles ? La réponse est très simple : le coût ! (Strohmeier 1996, p.2)

De manière générale, plus une anomalie est découverte tard, plus elle coûte cher à résoudre. Sans parler de la constitution des jeux de tests, la façon la plus efficace de tester une application est d'utiliser un outil de rejeu de test. On évite ainsi le passage fastidieux des étapes de test à la main. Ensuite vient le problème de maintenance de ces tests. C'est là que les tests basés sur les modèles prennent tout leur sens. On rejoint là encore les avantages du *MDE* dans le monde du développement. On retrouve entre autres l'indépendance entre la logique métier et la plate-forme technologique ciblée, une meilleure qualité de codage grâce à la génération du code à partir des modèles, et cela « force » les concepteurs à formaliser les spécifications.

Contrairement aux recherches présentées ci-dessus, nous avons essayé de ne pas limiter notre processus à certaines technologies ou langages. En effet, nos clients viennent d'horizons très larges et ont le plus souvent des parcs applicatifs hétérogènes. Dans la mesure du possible, ils attendent que nous prenions en compte l'ensemble de leur patrimoine.

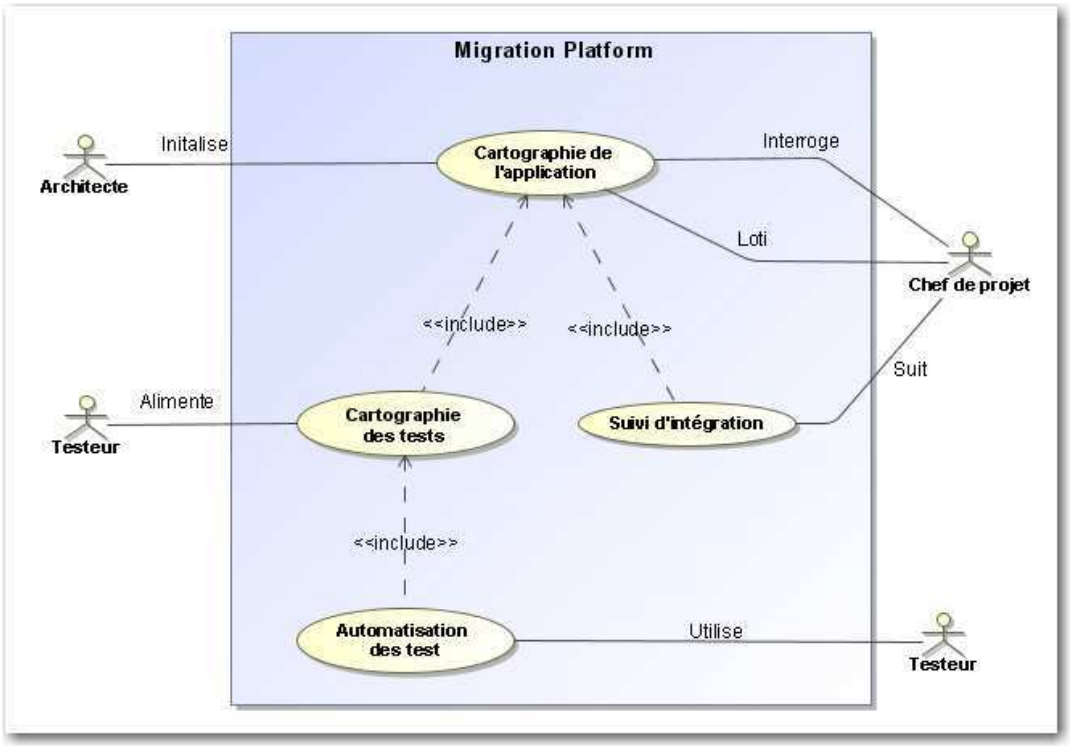


Figure 15 : Cas d'utilisation de la plate-forme de migration (« Migration Platform »)

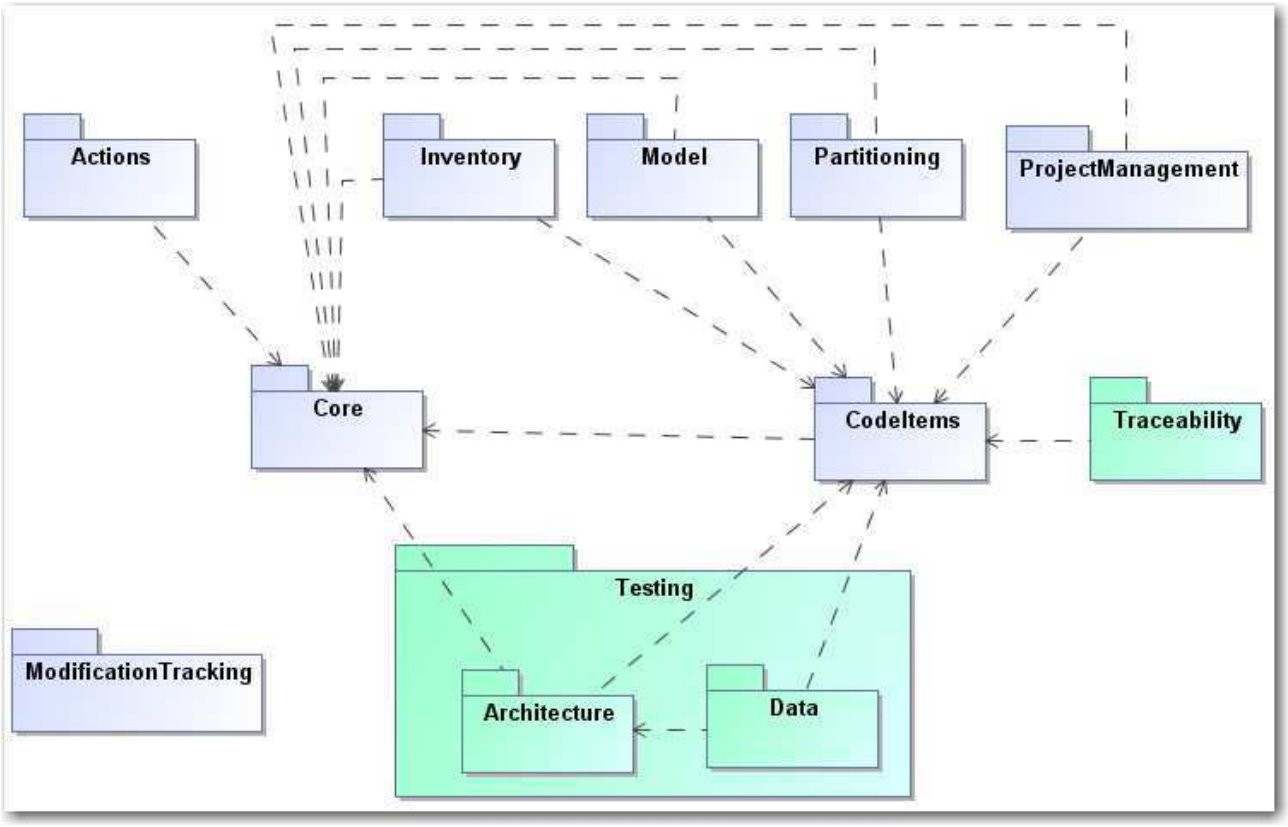


Figure 16 : Vue d'ensemble des paquetages constituant le métamodèle « *Migration Platform* »

(Source Sodifrance, les parties que j'ai modélisées sont en vert)

3.2 Plate-forme de migration (« *Migration Platform* »)

La Figure 15 représente les principaux cas d'utilisation du système « *Migration Platform* ».

Ils sont répartis de la manière suivante :

- La cartographie de l'application représente la base de l'ensemble et contiendra tous les composants applicatifs du projet (fenêtres, composants graphiques, classes, fonctions).
- La cartographie des tests intègre les différentes données propres aux tests et s'appuie sur la cartographie de l'application pour indiquer quels sont les composants utilisés par les tests.
- Le suivi d'intégration quant à lui permet d'avoir un historique des niveaux d'intégration par composant.
- Ensuite vient l'automatisation des tests qui s'appuie sur la cartographie des tests. Elle permet, comme on le verra un peu plus loin, de spécifier une relation entre les composants sources et cibles présents dans la cartographie de l'application.

Les différents acteurs sont l'architecte, qui a pour rôle principal d'initialiser et de vérifier que la cartographie de l'application est correcte ; le testeur qui alimente la cartographie des tests d'une part, et génère les scripts de tests d'autre part ; le chef de projet qui utilise la cartographie de l'application pour obtenir différentes visions de celle-ci, et qui se sert du suivi d'intégration pour avoir une idée réaliste de l'état d'avancement de l'intégration. Ces différents cas d'utilisation sont détaillés dans les paragraphes suivants.

La Figure 16 représente la répartition des paquetages constituant le système plate-forme de migration (« *Migration Platform* »). On peut remarquer que les deux principaux paquetages cœur (« *Core* ») et éléments de code (« *CodeItems* ») forment un noyau autonome. Les autres « fonctionnalités » viennent se greffer sur ce noyau et sont indépendantes entre elles, mis à part les deux paquetages architecture de test (« *Testing.Architecture* ») et données de test (« *Testing.Data* ») qui sont regroupées au sein du paquetage test.

3.2.1 La cartographie des applications

Avant d'approfondir les deux principaux objectifs de ce mémoire, se pose le problème essentiel de la cartographie des applications. En effet, pour pouvoir indiquer quels composants sont utilisés par un scénario de test, il faut déjà pouvoir lister de manière exhaustive l'ensemble des composants de l'application et les identifier de manière unique.

Comme indiqué dans l'état de l'art, nous ne nous sommes pas servis de *KDM*. Cependant, bien que fortement simplifié, nous nous en sommes grandement inspirés pour concevoir le métamodèle « Migration Platform ». La réalisation de ce métamodèle est le fruit d'un travail commun entre mon responsable, M. Breton, un collègue, M. Pacaud et moi-même. Plus précisément, mon rôle a été de concevoir les parties afférentes aux tests :

- Le paquetage architecture des tests (« *Testing Architecture* », cf. Figure 20)
- Le paquetage données de test (« *Testing Data* », cf. Figure 21)
- Le paquetage traçabilité (« *Traceability* », cf. Figure 28)

Ces diagrammes de classe sont détaillés dans les § 3.2.2 et 3.2.3.

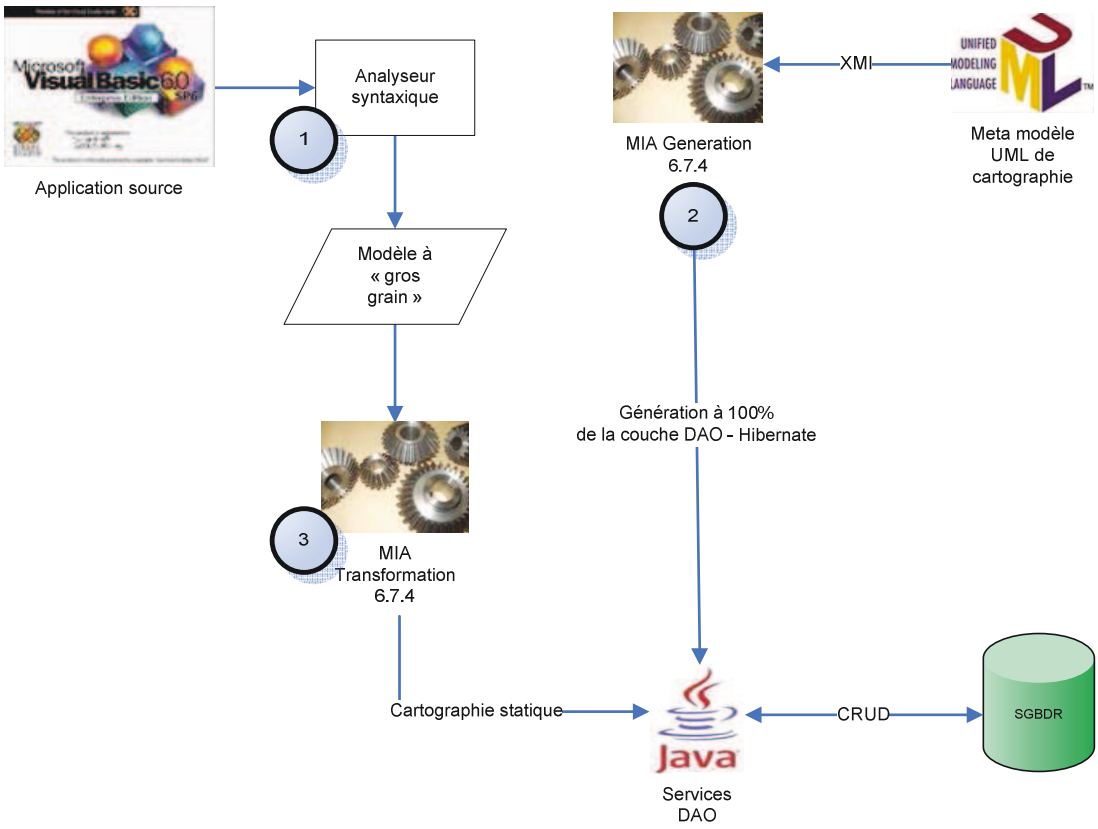


Figure 17 : Processus d'alimentation de la cartographie d'application

Le métamodèle « *Migration Platform* » a été réalisé avec le modeleur UML *MagicDraw*, sous la forme d'un modèle UML. Il est décrit principalement par des paquetages, des classes, des attributs et des relations. *MIA Generation* permet d'importer un fichier XMI[15] exporté depuis *MagicDraw*. Nous nous en sommes servi afin de générer entièrement la couche de persistance DAO[16] Hibernate dans le langage Java[17] (cf. Figure 17 étape 2). Nous remplissons ainsi deux des contraintes initialement fixées, à savoir une façon simple d'exposer les données aux outils de la chaîne d'évolution d'architecture. Mais aussi, une solution performante, car même en cas de volumétrie importante, c'est sur le moteur de base de données relationnelle (SGBDR[18]) que repose principalement ce problème, or il est justement prévu pour ça. Cette couche de persistance sera intégrée aux outils sous la forme d'une archive Java (Jar) déposée dans des répertoires prédéfinis, par exemple : <répertoire installation *MIA Generation*>\tools\lib.

La Figure 17 qui présente le processus de cartographie des applications, permet de voir l'ensemble des actions qui, en partant de l'analyse du code source, permet de peupler la base de données « Migration Platform ». En premier lieu se déroule l'analyse syntaxique du code source (cf. Figure 17, étape 1) afin de produire un modèle de l'application à « gros grain ». Ce modèle est du même niveau que *KDM* et ne descend pas jusqu'aux instructions mais s'arrête aux méthodes et aux relations qui les unissent. On évite aussi, grâce à ce niveau de modèle, des problèmes de volumétrie. En effet, il n'est pas rare d'avoir à « découper » des applications sources en plusieurs parties afin que le volume des modèles soit compatible avec les outils MIA, on parle ici de modèles XMI d'environ une centaine de mégaoctets. Ce découpage pose de sérieux problèmes de résolution de type à l'analyseur, lorsqu'une classe qui est décrite dans un modèle est utilisée dans un autre.

Après avoir déposé l'archive Java issue de l'étape 2 dans le répertoire prévu à cet effet, et effectué quelques paramétrages (adresse de la base de données, type de serveur), *MIA Transformation* peut lire le modèle de l'application source fourni par l'analyseur syntaxique, et peupler la base de données « *Migration Platform* » (cf. Figure 17, étape 3).

[15] XMI : XML Metadata Interchange
[16] DAO : Data Access Object
[17] Java : langage de programmation orienté objet
[18] SGBDR : Système de Gestion de Base de Données Relationnelle.

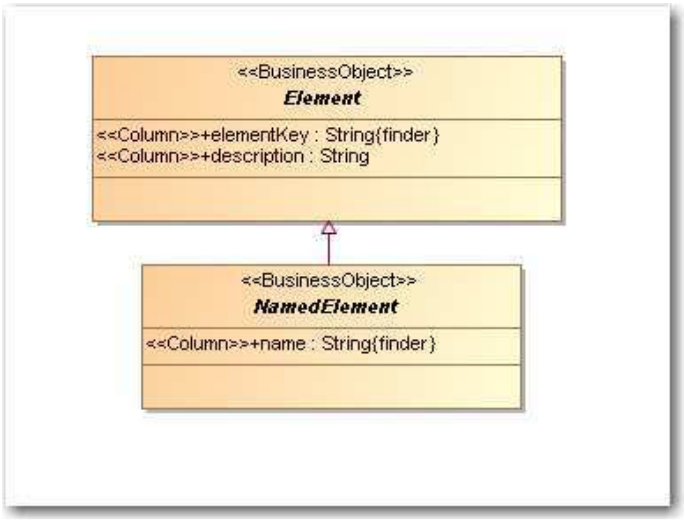


Figure 18 : Diagramme de classe du paquetage « Core »

(source Sodifrance, extrait du métamodèle « Migration Platform »)

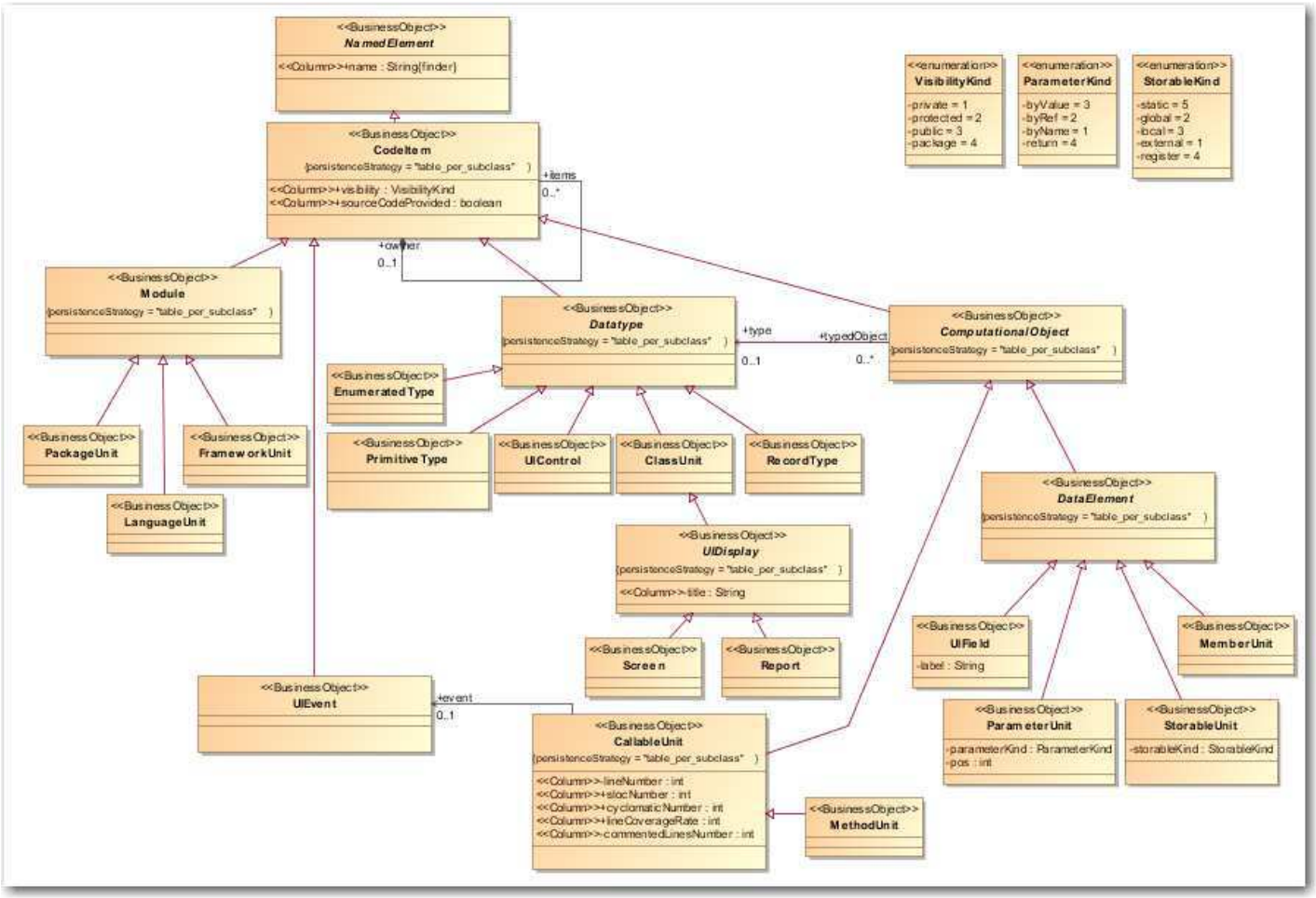
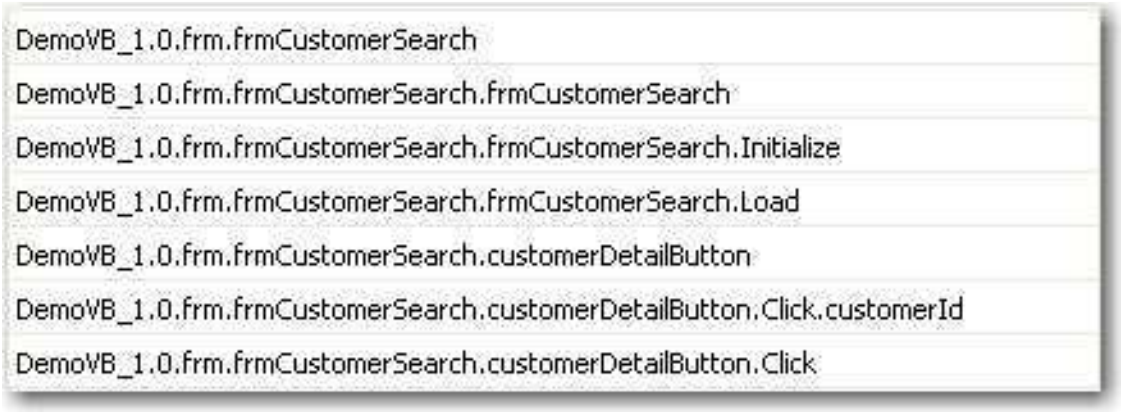


Figure 19 : Diagramme de classe du paquetage « CodeItems »

(source Sodifrance, extrait du métamodèle « Migration Platform »

Attardons-nous un instant sur la manière dont sont identifiés les éléments au sein de la base. Les classes servant à la cartographie héritent de la classe « *Element* » (cf. Figure 18 et Figure 19). Cette classe possède une propriété « *elementKey* » qui est essentielle. En effet, c'est elle qui assure l'identification de manière unique de chaque élément. La construction de cette clé répond à des règles très strictes qui, on le verra par la suite, devront être respectées à tous les niveaux du processus de migration. Ce formalisme consiste à reprendre l'ensemble de la clé de l'élément parent, et à y ajouter l'identifiant de l'élément courant.

Tableau 1 : Exemples de clefs



Le Tableau 1 montre un exemple concret de cette notation avec le modèle « *DemoVB_1.0* ». Le conteneur « *frmCustomerSearch* » se trouve dans le répertoire « *frm* ». L'écran « *frmCustomerSearch* » contient les événements « *Initialize* » et « *Load* », et a aussi un contrôle graphique nommé « *customerDetailButton* ». Ce dernier a un événement « *Click* » qui utilise une variable « *customerId* ».

Ce système de classification arborescente permet de lister l'ensemble des composants de l'application de manière exhaustive. L'élément racine correspond au modèle présent en base et représente l'application, vient ensuite s'il y a lieu, l'arborescence des répertoires, puis les fichiers. Dans le modèle de l'application source, le type du fichier : écran, classe, module, etc., a déjà été déterminé par l'analyseur syntaxique. Enfin, on retrouve les méthodes, leurs propriétés, et le type de retour s'il s'agit des fonctions. Le fait d'avoir intégré l'arborescence complète des fichiers, permet de prendre en compte plusieurs fichiers portant le même nom mais présents dans des répertoires différents.

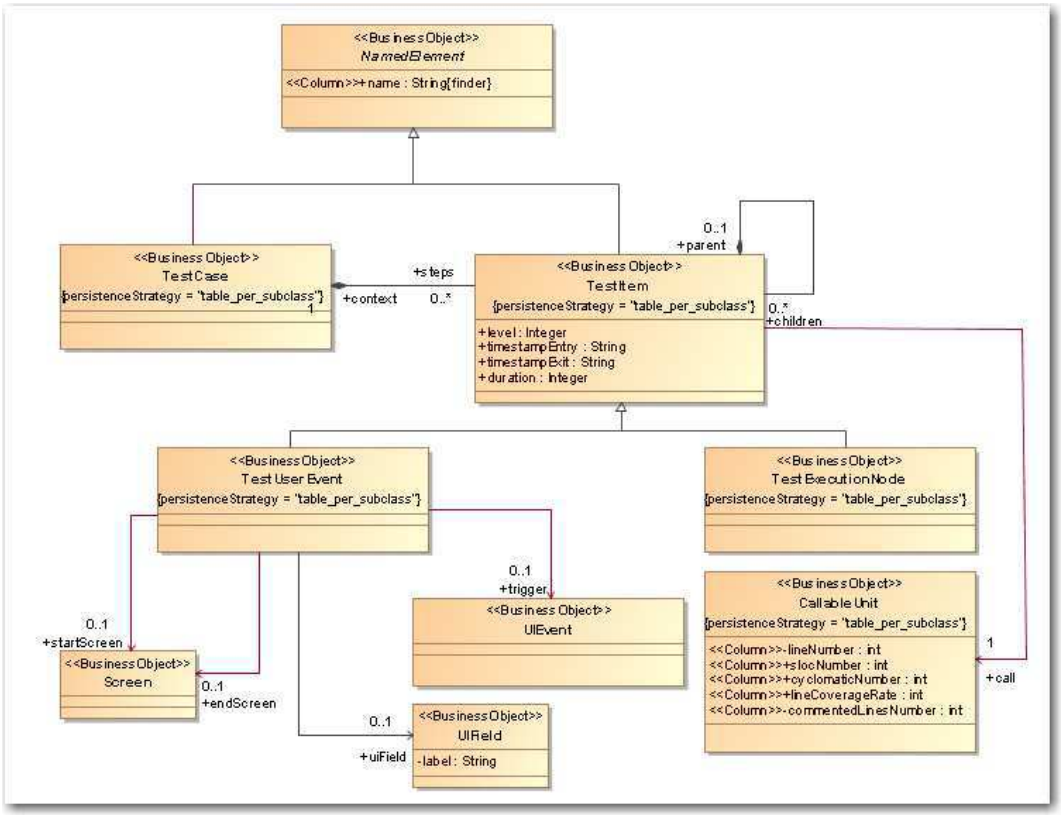


Figure 20 : Les classes du paquetage architecture des tests (« *Testing.Architecture* »)

du métamodèle « *Migration Platform* »

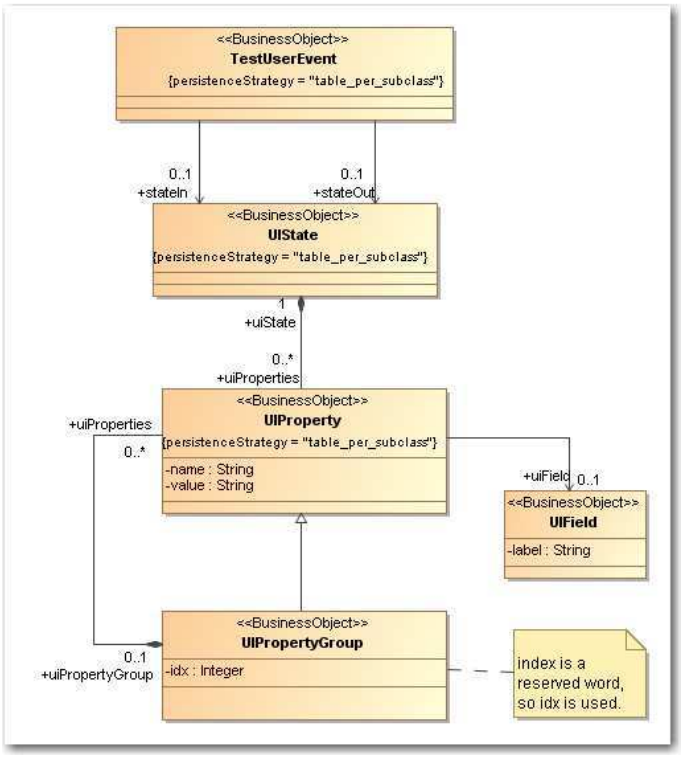


Figure 21 : Les classes du paquetage données de test (« *Testing.Data* »)

du métamodèle « *Migration Platform* »

3.2.2 Objectif 1 : cartographie des tests

L'instrumentation du code source, détaillé au § 4.2, doit fournir, à la suite du passage des tests de références, deux types de flux XML. Dans le premier, on va chercher à ajouter du code pour chaque méthode, juste après l'entrée et juste avant chaque instruction de sortie. On obtient ainsi une arborescence des appels de méthodes. Le second flux permet de connaître pour les méthodes de type « événement utilisateur » l'état de l'ensemble des éléments graphiques d'un écran à un instant donné. Il est alimenté, comme le précédent, lors de entrée et de la (ou des) sortie(s) de la méthode.

Un service Java permet de lire ces flux et de les intégrer à la base « *Migration Platform* ». Cela revient à insérer les tests de référence en base. Ils sont vus comme des cas de test, composés d'éléments, qui sont soit des appels de méthode simple : « *testExecutionNode* » (cf. Figure 20), soit des appels de méthodes événementielles : « *testUserEvent* ».

Ces derniers ont alors un état d'entrée et de sortie de l'écran : « *UIState* » (cf. Figure 21). Il est lui-même composé d'un ensemble de paires clé / valeur permettant de représenter les propriétés de chaque composant de l'écran.

En Visual Basic 6, langage de programmation sur lequel a été réalisé une grande partie des maquettes, l'introspection n'est pas totalement implémentée. C'est-à-dire qu'on ne peut pas connaitre dynamiquement l'ensemble des propriétés d'un composant graphique par exemple. En revanche, si on connaît le nom des propriétés pour lesquelles on veut la valeur, la commande « `CallByName` » permet d'interroger dynamiquement la propriété du composant et d'obtenir sa valeur. Il a donc fallu définir par le biais d'un fichier de configuration les composants graphiques et les propriétés faisant l'objet d'une « surveillance » de la part de l'instrumentation.

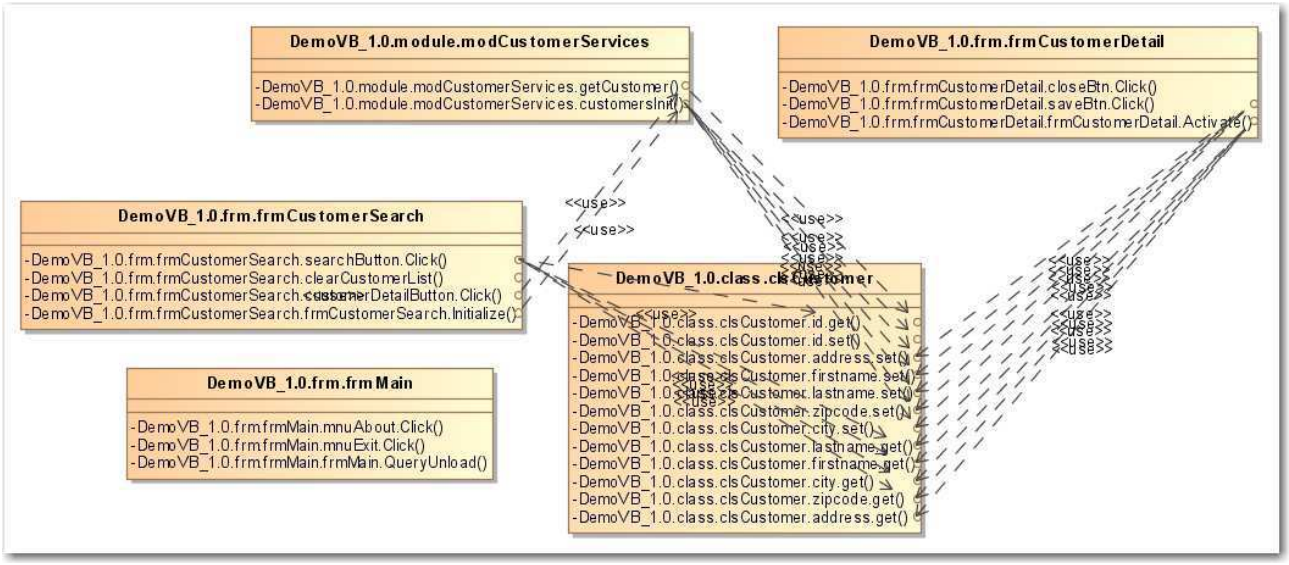


Figure 22 : Exploitation de la cartographie pour produire un diagramme de classe

3.2.2.1 Vision statique et dynamique de la cartographie

Avant l'insertion des tests, la base de donnée ne contient que ce que l'analyseur syntaxique a pu lui fournir. Pour garder le langage Visual Basic en exemple, l'instruction :

```
Public Function getCustomer() as clsCustomer
```

sera interprétée comme une fonction renvoyant une instance de la classe « `clsCustomer` ». Dans le modèle, il en résulte la création d'un lien entre cette fonction et la classe « `clsCustomer` ». En revanche, l'instruction suivante ne donnera pas les mêmes résultats :

```
Public Function getCustomer() as Object
```

En effet, même si dans le corps de la méthode, les instructions font que le type renvoyé sera effectivement une instance de la classe « `clsCustomer` », l'analyseur syntaxique ne réussira pas à résoudre cette relation. On arrive alors aux limites de l'analyse statique du code. Grâce à l'instrumentation de toutes les méthodes du code source, la trace XML permettra de mettre en évidence, à l'intérieur de la fonction « `getCustomer` » l'accès au constructeur de la classe « `clsCustomer` » ou à des méthodes liées à cette classe. On obtient donc une vision dynamique de l'application. On pourra donc ajouter la liaison non résolu par l'analyseur syntaxique entre la fonction et la classe

Au niveau de la base « *Migration Platform* », on ajoute à la vision statique de l'analyseur la vision dynamique fournie par les tests de références. Au passage, on peut noter qu'on répond à une des exigences initiales : le point d'entrée étant un cas de test, on connait l'exhaustivité des composants utilisés par ce cas de test.

Concernant le langage Visual Basic, il peut être utilisé de manière assez libre et devenir très peu typé. Cette vision dynamique est indispensable afin d'obtenir une idée claire de l'ensemble des relations entre les composants de l'application.

3.2.2.2 Représentation graphique de la cartographie des tests

J'ai étudié et maquetté sous forme de services Java plusieurs types de représentations graphiques.

A partir des données de la cartographie d'application, j'ai produit un modèle UML qui exporté au format XMI puis importé dans *MagicDraw*, permet d'obtenir un diagramme de classe et de visualiser les relations entre les opérations (cf. Figure 22).

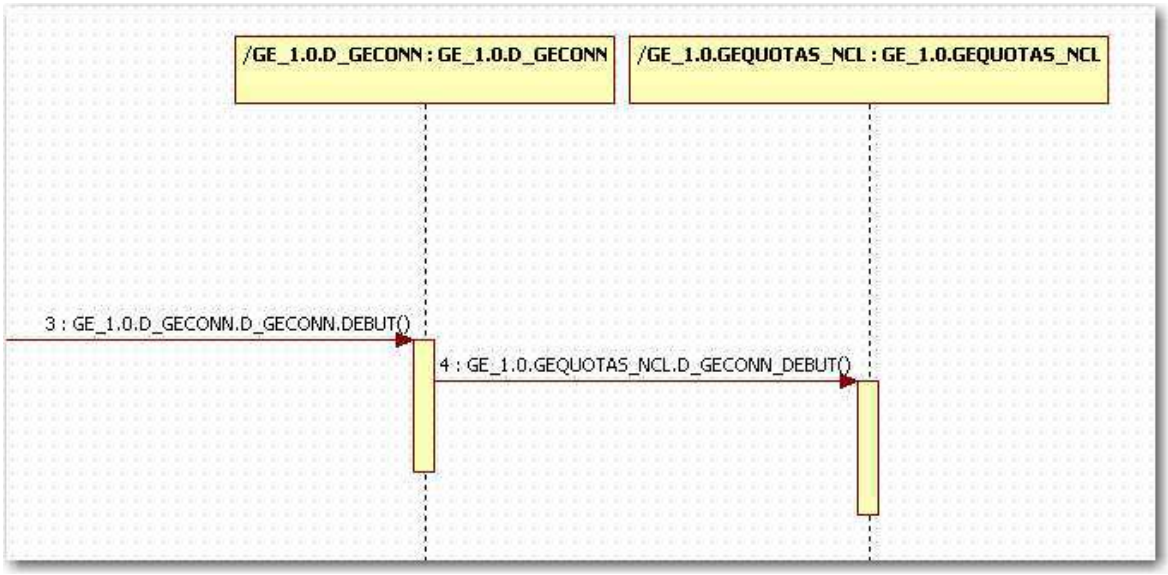


Figure 23 : Exploitation de la cartographie de test pour produire un diagramme de séquence

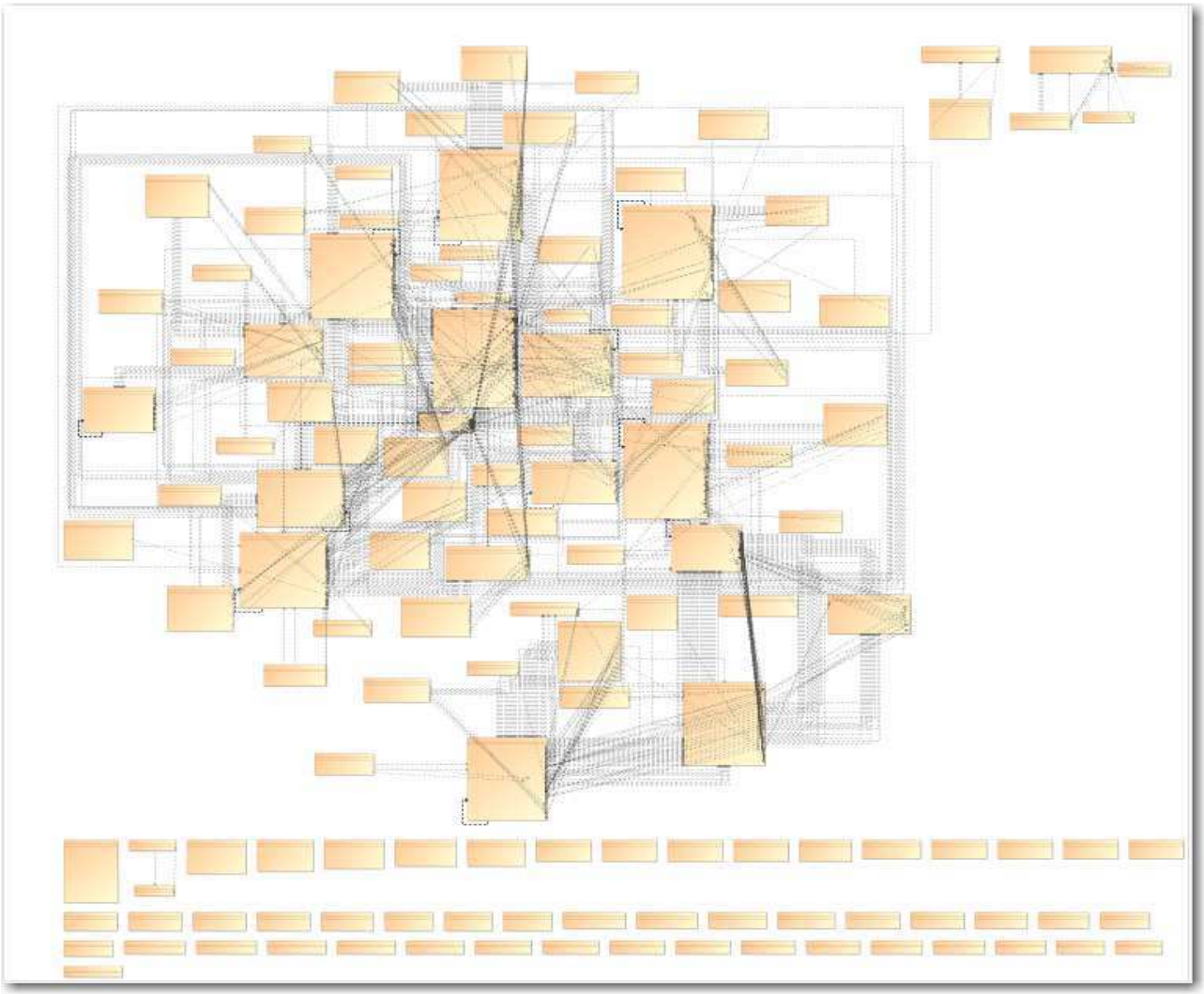


Figure 24 : Exploitation de la cartographie pour produire un diagramme de classes… **inutilisable**

Ensuite, à partir de la cartographie de test, j'ai produit un diagramme de séquence, plus approprié à représenter les cas de tests, car il permet de garder la chronologie entre les appels. Contrairement à UML 1.3, UML 2 n'intègre plus lors dans son format d'import de fichier XMI la description des diagrammes. Elles sont déportées dans des extensions spécifiques à chaque modeleur. J'ai donc utilisé la version UML 1.3 et le modeleur Star UML afin d'importer le diagramme de séquence (cf. Figure 23).

Ces deux types de représentation, bien que reconnues de par leur formalisme, ne conviennent pas ou ne répondent pas de façon satisfaisante à nos besoins. On observe aisément que les problèmes de volumétrie vont vite devenir rédhibitoires. Sur une application un peu plus conséquente, le diagramme de classe mis en forme de manière automatique devient très rapidement illisible (cf. Figure 24).

On arrive au même souci avec le diagramme de séquence sur des cas de test plus imposants, d'ailleurs la Figure 23 n'est qu'un extrait d'un diagramme beaucoup plus important. Partant du principe que le besoin initial est de représenter des objets et leurs relations, je me suis orienté vers un logiciel de graphe reconnu : *yEd*[19]. Bien sûr l'idée n'est pas d'opposer *MagicDraw* et *yEd*, ces deux logiciels ayant chacun leurs fonctionnalités, mais de chercher à obtenir une représentation de l'information exploitable, même en cas de volumes de données important.

[19] *yEd* : Editeur graphique : http://www.yworks.com/.../yEd.html

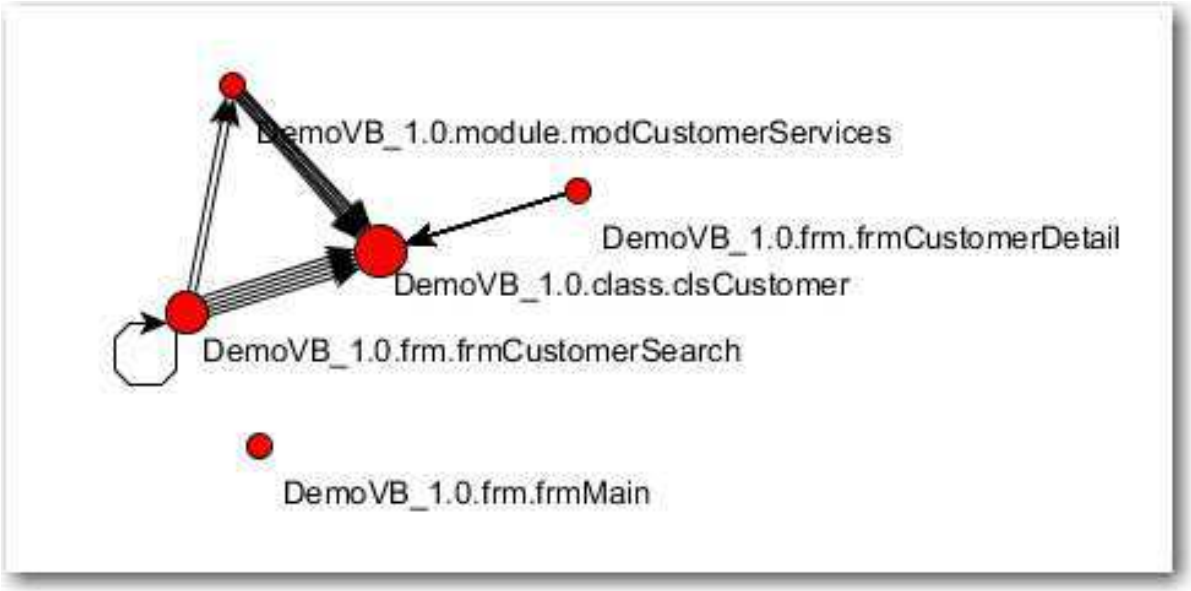


Figure 25 : Exploitation de la cartographie pour produire un graphe

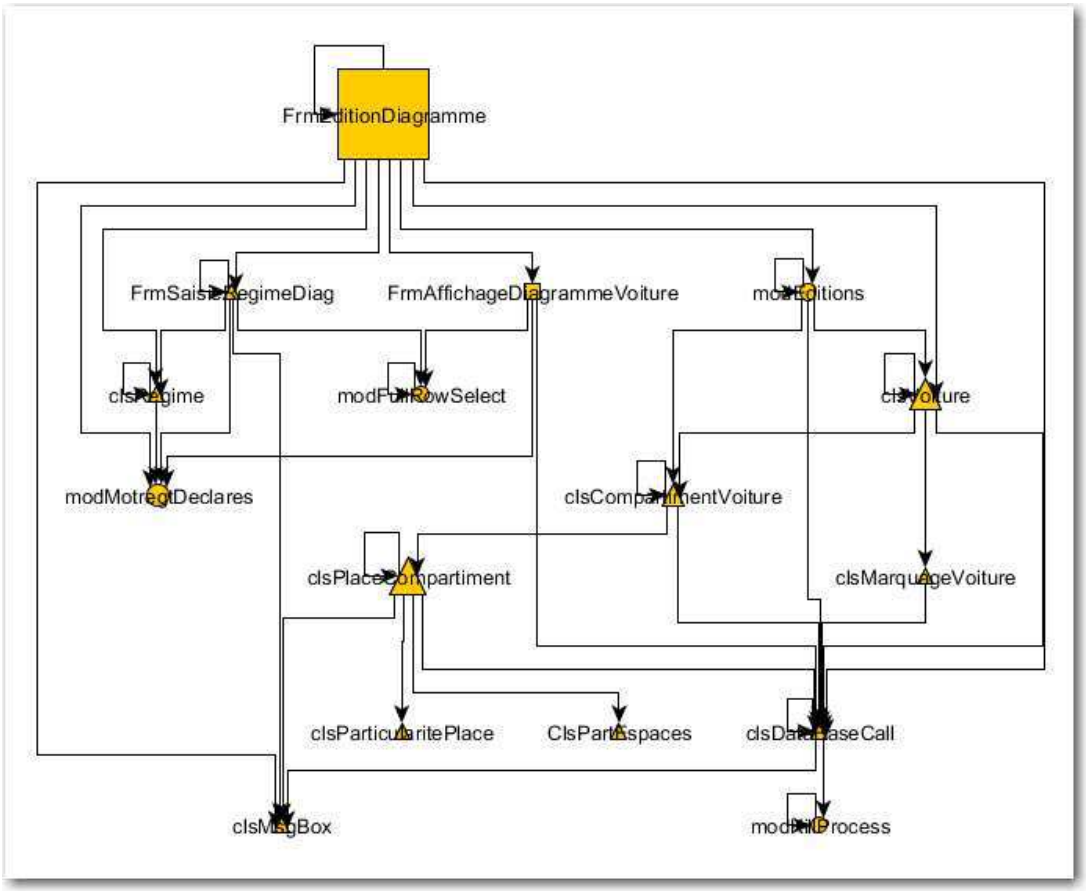


Figure 26 : Graphe hiérarchique d'appels entre éléments

yEd est un logiciel spécialisé dans la représentation et la mise en forme des graphes. Les graphes sont composés de nœuds reliés entre eux par des arcs. Par exemple, la Figure 25 a les mêmes données d'origine que la Figure 24, simplifié puisqu'on ne détaille pas les méthodes, et mise en forme selon l'algorithme « organique » définit par *yEd*. Il a pour particularité de donner un « graphe groupé, une répartition équilibrée des nœuds et peu de croisements d'arcs »[20]. Outre les nombreuses options de mises en forme, il y a des fonctionnalités poussées de recherche et de sélection des nœuds, selon des critères aussi divers que leurs noms, leurs couleurs, les prédécesseurs de, les successeurs de.

Par exemple, il est très simple de sélectionner un nœud ainsi que tous ses descendants, et ensuite de créer un nouveau graphe à partir de cette sélection. Ici, la Figure 26 reprend les éléments sélectionnés dans un graphe beaucoup plus important et les disposent sous forme hiérarchique.

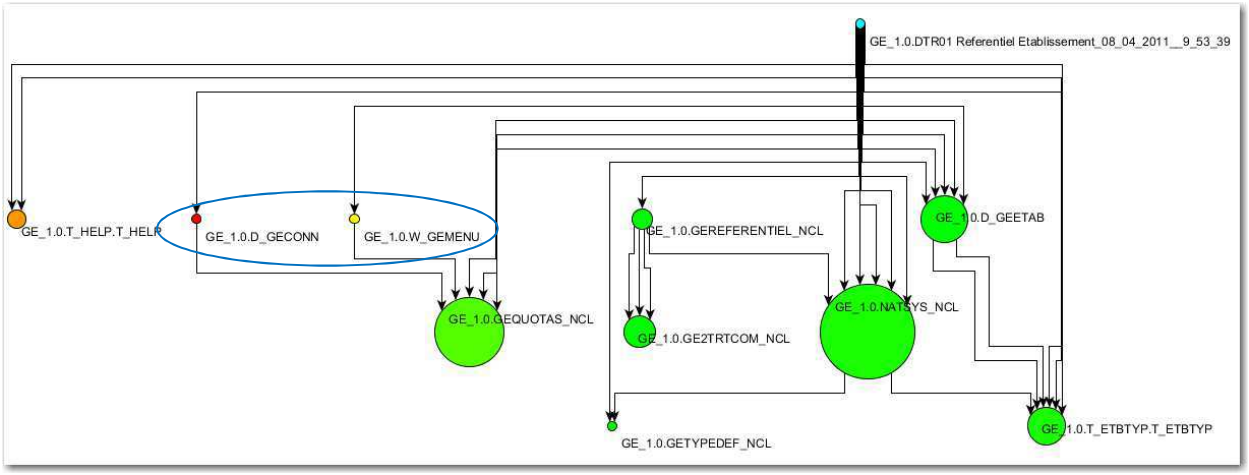


Figure 27 : Exploitation de la cartographie de test pour produire un graphe hiérarchique

Un inconvénient tout de même si on compare le graphe hiérarchique au diagramme de séquence, on peut voir qu'on perd la chronologie exacte des appels de méthodes. En fait l'ordre général est conservé, mais quand une méthode en appelle deux autres, on ne peut plus savoir laquelle des deux a été appelée en premier. Dans la Figure 27, on ne sait pas quel élément de « `GE_1.0.D_GECONN` » ou « `GE_1.0.W_GEMENU` » est appelé en premier à partir de « `GE_1.0.DTR01 Référentiel Etablissement_08_04_2011__9_53_39` ».

[20] source documentation yEd

yEd donne la possibilité de modifier l'aspect graphique des éléments du graphe. Ainsi, nous avons déterminé une formule[21] donnant une valeur, un poids, en fonction de la complexité des composants. Le poids d'une classe ou d'un écran correspond à la somme des poids de ses méthodes. Il est ensuite très facile de modifier la taille d'un élément en fonction de son poids, et donc de sa complexité. Il en ressort des graphes dans lesquels les composants les plus complexes, en général ceux qui devront recevoir une attention particulière, sont facilement identifiables (cf. Figure 27).

La troisième piste utilisée pour représenter les informations de cartographie a été le développement d'un plugin Eclipse par un stagiaire en Master 2 Informatique que M. Pacaud et moi-même avons encadré, sous la responsabilité de M. Breton. Ce plugin décrit plus en détail dans le § 4.1, est plutôt à destination des chefs de projet qui souhaitent suivre l'avancement de l'intégration des projets. On y retrouve sous forme arborescente la cartographie de l'application ainsi que les scénarios de tests et l'avancement de l'intégration de chaque composant. L'outil est complété par des vues donnant des indications de volumétrie selon des angles différents : nombre de composants graphiques par type, nombres d'événements graphiques par type d'événement et de composant, etc.

[21] Cette formule met en relation le nombre de lignes de code (*Source Line Of Code*, SLOC), le nombre cyclomatique, et pour un écran, le nombre de composants graphiques.

3.2.3 Objectif 2 : automatisation des tests

Les tests sur les applications cibles se font à l'aide d'outils de rejeu de test, comme *FlexMonkey*[22] pour une application Flash par exemple. Habituellement, il faut jouer les scénarios de test sur l'application à tester afin que l'outil enregistre les actions qui sont effectuées sur l'interface graphique. L'idée de l'automatisation des tests, c'est de mettre en place un processus dont la finalité permettra d'alimenter automatiquement les outils de rejeu de test. Or, l'ensemble du processus mis en place vise à atteindre cet objectif. L'instrumentation du code source remplace la phase d'enregistrement des actions sur l'application cible.

L'outil de test doit être en mesure d'identifier chaque composant de l'application cible afin de pouvoir tester les valeurs de ses propriétés ou d'effectuer des actions dessus. On commence à percevoir le problème de l'automatisation lorsqu'on sait que le processus d'évolution d'architecture intègre les conventions de nommage du langage cible, ou encore les spécifications propres à chaque client.

Au cours de la migration, il faut donc avoir une solution pour établir un lien entre les composants de la source et ceux de la cible. Ceci est rendu possible en intégrant trois actions dans le processus de migration industrielle.

Une première action, en tout début de processus, consiste à « attacher » une clé à chaque composant avant que ceux-ci n'aient été renommés ou modifiés. Cette clé est construite selon le même format que celui décrit au § 3.2.1.

La deuxième action se situe quant à elle en toute fin de processus. A ce niveau, les composants ont pu être renommés, déplacés, ou même avoir changés de type. En effet un client peut demander à changer tous les boutons « OK » en liens hypertexte « Valider ». On reconstruit une nouvelle clé, en suivant les mêmes règles qu'avec la clé d'origine et encore une fois, on l'« attache » au composant. Chaque composant se retrouve donc « agrémenté » de deux clés, celle de la source et celle de la cible.

[22] FlexMonkey : outil de rejeu de test spécifique aux applications Flash : http://www.gorillalogic.com/flexmonkey

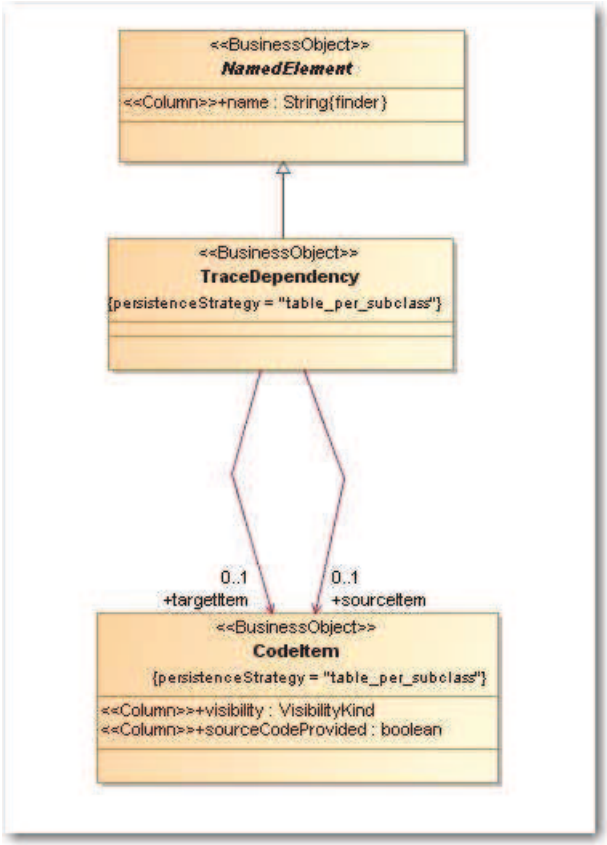


Figure 28 : Les classes du paquetage traçabilité (« Traceability)

du métamodèle « *Migration Platform* »

L'action finale se résume à insérer dans la base de données Migration Platform, les nouveaux composants cibles mais aussi leurs relations avec les composants issus de la cartographie statique par le biais du paquetage de traçabilité (« *TraceDependency* », cf. Figure 28).

3.2.3.1 Génération des scripts de test

La génération des scripts de test se fait alors en parcourant les enregistrements de test, qui pointent sur des composants sources, et pour chaque composant, en cherchant le composant cible qui lui est associé. Pour chaque événement en base, on retrouvera trois actions de génération :

- Positionner les propriétés qui n'auront pas été changées de manière événementielle. Par exemple, dans le scénario de test, on coche une case, et il n'y a pas d'événement associé à cette action. L'instrumentation du code source n'aura donc pas ajouté de code pour suivre l'évolution de cette case à cocher. En revanche, l'événement *Click* du bouton OK déjà existant, aura été instrumenté. L'instrumentation vérifiera l'ensemble des zones de l'écran et informera que la valeur de la case à cocher a changé. Il restera alors soit à modifier

la propriété, soit à déclencher un événement afin de modifier la propriété. On peut noter deux choses importantes :

- seules les propriétés modifiées depuis le dernier appel sont remontées par l'instrumentation.
- il ne faut pas tenir compte des propriétés du composant sur lequel porte l'événement en cours. Elles seront modifiées par l'événement lui-même.

- Déclencher l'événement. Après avoir positionné les propriétés des autres composants, on peut déclencher l'événement enregistré. Dans notre exemple, il s'agira du clic sur le bouton.
- Vérifier les propriétés après l'événement. L'événement *Click* du bouton OK aura déclenché des appels aux services métier, et les données présentes à l'écran auront pu être modifiées. L'instrumentation nous fournit les valeurs des propriétés des composants de l'écran à la fin de l'événement. Il reste donc à vérifier que les données attendues ont bien été mises à jour par l'appel à l'événement *Click* et aux services métier.

Pour être précis, il y a bien quelques cas particuliers à gérer, comme les événements ou propriétés parasites qu'il y peu d'intérêt à observer. Par exemple, toujours pour Visual Basic, le changement de ligne dans une `Combobox` changera les propriétés `Text` et `SelectedIndex`. Seule la propriété `SelectedIndex` sera à positionner, le texte de la `Combobox` sera mis à jour implicitement.

Certaines propriétés ne font en fait référence qu'à une seule action de l'utilisateur. Le clic dans la cellule d'un tableau mettra à jour les propriétés `ColIndex` puis `RowIndex`. Ceci est dû au fonctionnement de l'instrumentation qui ne peut tester qu'une propriété à la fois.

Au final sur la cible, il ne faudra générer qu'un clic sur la grille dans la cellule correspondant aux coordonnées des propriétés `ColIndex` et `RowIndex`.

Un événement utilisateur peut demander deux événements sur la cible. Cliquer sur une ligne d'une `ComboBox` peut obliger à effectuer auparavant un `Scroll` afin que la ligne à cliquer soit visible. Cette limitation peut provenir de la technologie de l'application cible. Pour Flex[23] par exemple, on est bien dans ce cas. On obtient donc un événement source qui doit en produire deux sur la cible, le `Scroll` puis le `Click`.

[23] Flex : Le logiciel est un outil Open Source de développement d'applications web

Figure 29 : Vue de synthèse du plugin Eclipse (source Sodifrance)

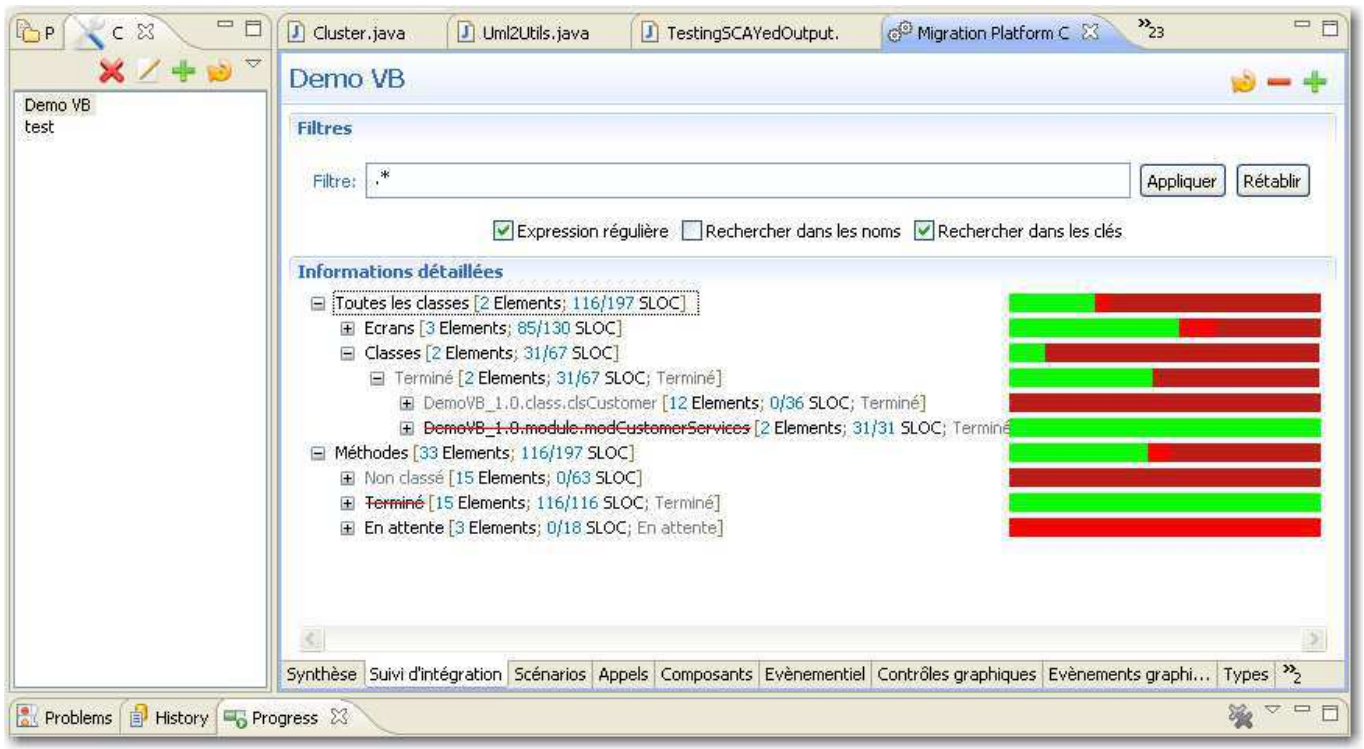


Figure 30 : Vue du suivi d'intégration du plugin Eclipse (source Sodifrance)

4 LES TRAVAUX CONNEXES

4.1 Réalisation d'un plugin Eclipse

Durant la réalisation de ce mémoire, j'ai eu à encadrer le stage en entreprise d'un étudiant en Master 2 en Informatique, spécialité architecture logicielle de l'université de Nantes. L'objectif de son stage était la réalisation d'un plugin Eclipse permettant de suivre l'avancement de l'intégration des projets de migration. Cet outil de suivi est centré sur la base de cartographie à laquelle des notions de suivi ont été greffées, et sur l'analyse des codes sources en cours d'intégration.

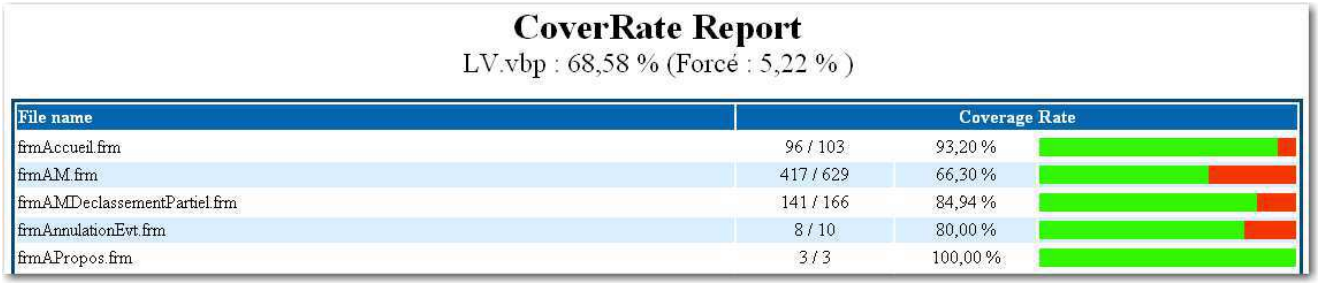
Dans le processus classique de migration, les scripts de génération des méthodes et des classes ont été modifiés afin d'intégrer deux types de balises :

- Une balise identifiant qui ne doit pas être supprimée.
- Des balises de début et de fin de méthode.

Elles ont toutes comme information principale l'« `elementKey` », leur identifiant en base.

Lorsque l'intégrateur commence à travailler sur une méthode, il supprime la balise de début.

La méthode est alors reconnue comme étant en cours d'intégration. Ensuite, lorsque l'intégrateur estime qu'il a terminé sa méthode, il supprime la balise de fin. La méthode passe alors en statut terminé. Si on prend pour exemple une méthode qui contient dix lignes de code et qui a ses balises de début et de fin, elle comptera pour zéro ligne intégrée. Avec cette même méthode, si la balise de début a été supprimée, elle comptera pour cinq lignes intégrées. Et enfin si cette méthode n'a plus ni balise de début ni balise de fin, elle comptera pour l'ensemble de ses lignes de code, donc pour dix lignes intégrées. Ensuite, pour connaître le niveau d'intégration de la classe, on calcule le rapport entre le nombre de lignes de code intégrées et le nombre de lignes de code total. Certes, ce mécanisme assez basique, laissant des actions manuelles à la charge des intégrateurs n'est pas parfait. En effet, la plupart des erreurs proviennent justement de ces interventions humaines. Mais il permet tout de même d'avoir sur l'ensemble de l'application une idée assez précise de l'avancement de l'intégration.



CoverRate Report

LV.vbp : 68,58 % (Forcé : 5,22 %)

File name	Coverage Rate		
frmAccueil.frm	96 / 103	93,20 %	
frmAM.frm	417 / 629	66,30 %	
frmAMDeclassementPartiel.frm	141 / 166	84,94 %	
frmAnnulationEvt.frm	8 / 10	80,00 %	
frmAPropos.frm	3 / 3	100,00 %	

Figure 31 : Copie d'écran de la page Html du taux de couverture de l'application LV

```
998   Private Sub Image1_Click()
1000
1001     Dim BeginPage, EndPage, NumCopies, Orientation, i
1002     ' Affecte la valeur True à la propriété
1003     ' CancelError.
1004     CommonDialog1.CancelError = True
1006     On Error GoTo ErrHandler
1008     ' Affiche la boîte de dialogue Impression.
1009     CommonDialog1.ShowPrinter
1011     ' Récupère les valeurs sélectionnées par
1012     ' l'utilisateur dans la boîte de dialogue.
1013     BeginPage = CommonDialog1.FromPage
1015     EndPage = CommonDialog1.ToPage
1016     NumCopies = CommonDialog1.Copies
1018     Orientation = CommonDialog1.Orientation
1020     For i = 1 To NumCopies
1022     'données à envoyer à votre imprimante
1023      frmAM.PrintForm
1025      Printer.EndDoc
1027     Next
1029     Exit Sub
1030  ErrHandler:
1032     ' L'utilisateur a choisi le bouton Annuler.
1033     Exit Sub
1034
1035  End Sub
```

Figure 32 : Copie d'écran de la page HTML détaillant le code d'une méthode de l'application LV

4.2 Instrumentation

J'ai réalisé une suite d'outils permettant d'injecter du code dans des applications Visual Basic. L'objectif de ces injections est de produire des traces au format XML qui seront interprétées puis insérées dans la base de cartographie.

4.2.1 Taux de couverture

Le taux de couverture des tests de références est calculé de la manière suivante :

- On ajoute un appel à une méthode de comptage pour chaque ligne de code. Chaque appel prend en paramètre un compteur qui devient son identifiant. Dans le même temps, on insère dans une table de base de données une ligne correspondant à la ligne de code traitée, avec bien sûr le même identifiant.
- Ensuite, lors de l'exécution de l'application instrumentée, après chaque ligne de code, il y aura l'appel à la fonction de comptage. Cette dernière peut selon un paramétrage, soit produire une trace XML donnant la liste des lignes appelées, soit mettre à jour directement la ligne correspondant à son identifiant en base.
- Au final, on se retrouve avec une table contenant l'ensemble des lignes de l'application, et pour chacune d'elle, un indicateur spécifiant si elle a été utilisée lors de l'exécution ou non. De simples requêtes suffisent ensuite pour connaître le taux de couverture[24].

J'ai aussi développé un outil produisant un site web statique permettant de mieux visualiser le taux de couverture (cf. Figure 31). La première version de l'outil de couverture était d'une certaine manière plus « intelligente » car elle n'ajoutait pas d'appels à chaque ligne de code, mais uniquement au début des méthodes et dans chaque branche (conditions, boucles), en indiquant le nombre de lignes de code de chaque bloc. La Figure 32 démontre la nécessité d'instrumenter toutes les lignes de l'application.

En effet, ici si l'instruction « `CommonDialog.ShowPrinter` » provoque une erreur, le reste des lignes jusqu'au traitement d'erreur ne sera pas exécuté. Or, avec la logique précédente, dès l'entrée dans la méthode, l'ensemble des lignes se trouvant hors d'une condition ou d'une boucle, aurait été considérées comme passées, ce qui est faux, puisque la gestion d'erreur force le débranchement jusqu'à la balise « `ErrHandler:` », donc les lignes 1013 à 1018 n'ont pas été exécutées.

[24] Taux de couverture : $(\frac{100\,.\ nombre\ de\ lignes\ utilisées}{nombre\ de\ lignes\ total})$.

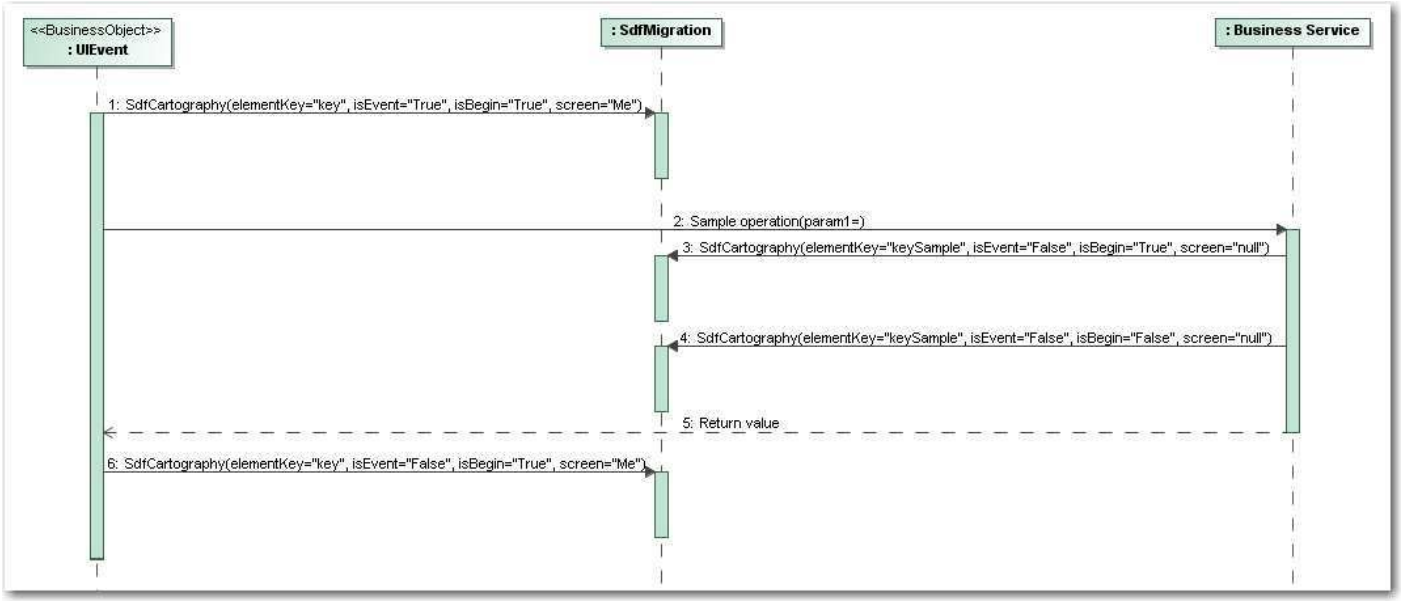


Figure 33 : Diagramme de séquence d'appels à l'opération « *SdfCartography* »

de la classe « *SdfMigration* ».

4.2.2 Cartographie dynamique, état des composants graphiques

La cartographie dynamique est basée sur l'ajout d'appels à des fonctions de trace en début et en fin de méthode. L'ensemble des méthodes de l'application source étant instrumentées, cela permet d'obtenir un graphe des appels entre méthodes et donc de résoudre les appels dynamiques (cf. § 3.2.2.1).

A noter que les méthodes évènementielles, produiront en plus un état de l'écran courant avant et après l'évènement. Cela donnera la possibilité de connaitre les impacts d'un évènement sur les données affichées à l'écran. Le diagramme de séquence d'appel à l'instrumentation (cf. Figure 33) illustre ce propos. En début d'événement ou de méthode, le booléen `isBegin` est à Vrai. Le booléen `isEvent` dépend du type de méthode. La clé correspond à la clé utilisée en base et l'écran est l'écran courant dans le cas d'un événement. En fin de méthode, seul le booléen `isBegin` passe à Faux.

Cet outil d'instrumentation était spécifique au Visual Basic, mais un nouvel outil fonctionnant à base de règles prend maintenant en charge plusieurs langages (VB, NSDK, NatStar, etc.). La contrainte pour cet outil est de respecter le format XML qui avait été défini, ainsi que de produire des clés conformes à la spécification.

4.3 Partenariat avec la société Kalios

Même si nous en avons la possibilité, notre but n'est pas de générer des scripts pour tous les outils de rejeu de test du marché. La réalisation de maquettes a permis de valider la viabilité de ce concept (maquette VB6 vers Java/Flex, outil de rejeu FlexMonkey). Cependant, c'est avec la société Kalios[25], qui développe le logiciel sYnopsis[26], que nous cherchons à poursuivre nos efforts. Le logiciel sYnopsis se présente comme une « surcouche » aux outils de test. Il permet d'organiser de manière efficace les scénarios de test et facilite de manière significative la maintenance des scripts de test. De plus, il permet de produire de la documentation et possède des connecteurs aussi bien vers des outils de suivi de test comme Quality Center[27], que vers des outils de rejeu comme Test Complete[28] ou Quick Test Pro[29].

[25] Kalios : http://www.kalios.com/
[26] sYnopsis : produit de pilotage d'outils de rejeu de test. sYnopsis
[27] HP Quality Center : outil de suivi de test. QC
[28] Test Complete : outil de rejeu. http://smartbear.com/products/qa-tools/automated-testing/
[29] HP Quick Test Pro : outil de rejeu. QTP

Dans le processus d'initialisation d'un projet sYnopsis, il y a une phase d'importation des composants graphiques des écrans depuis l'outil de rejeu. La description des scenarios de test ne pourra commencer qu'après cette initialisation. Or, cette phase de capture des composants graphiques reste manuelle et fastidieuse. L'idée est de se servir des données présentes dans la base de cartographie pour générer un flux XML qui permettra d'initialiser le référentiel d'objets sYnopsis.

Dans un deuxième temps, les bonnes pratiques de ce logiciel indiquent de créer pour chaque écran, au moins trois « modules » : un de saisie, un de vérification, et un par composant de navigation (par exemple par bouton). Là encore, les données présentes en base de cartographie vont pouvoir nous aider dans cette démarche.

Dans un troisième temps, une fois le référentiel des objets initialisé et les principales briques des scénarios générées, on va pouvoir exploiter les données de la cartographie de test afin de créer un fichier que sYnopsis pourra importer. On rejoint ici ce qui a déjà été fait pour l'automatisation des tests (cf. 3.2.3.1). Il restera à adapter le service afin qu'il produise un flux conforme à ce qu'attend sYnopsis.

J'ai actuellement la charge des échanges techniques avec la société Kalios afin de réaliser une maquette reprenant ces trois points et prouvant la viabilité du concept.

Tableau 2 : Etat d'avancement des différents objectifs

Objectifs fixés	Modélisation	Réalisation	Maquette	Tests avec une application réelle	Utilisé en production
Cartographie applicative	✓	✓	✓	✓	✓
Cartographie des tests	✓	✓	✓	⚠	⚠
Automatisation des tests	✓	✓	✓	⚠	⚠

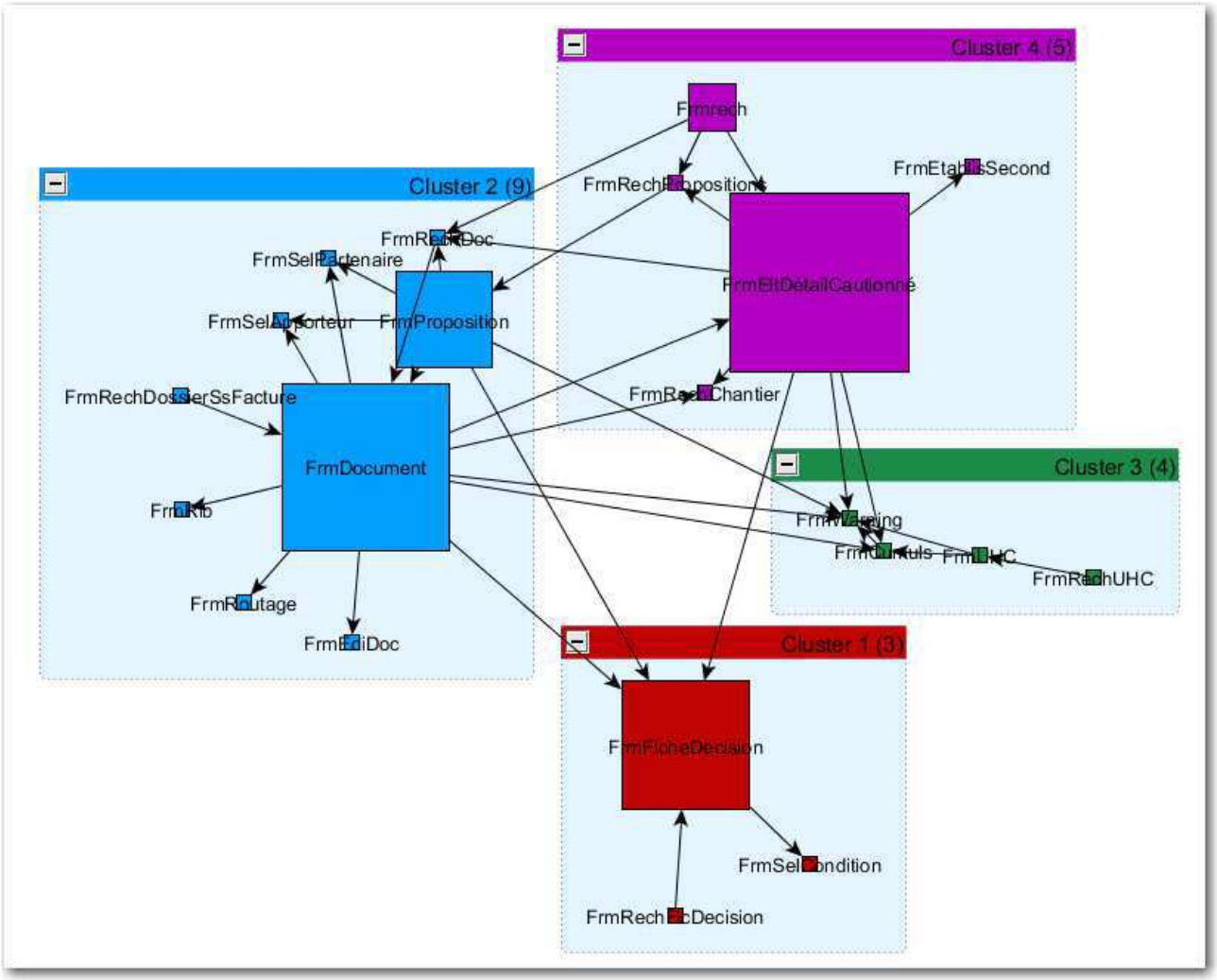


Figure 34 : Utilisation d'une fonctionnalité de *yEd* pour obtenir un premier niveau de lotissement

5 CONCLUSION

Ce sujet de mémoire, de prime abord assez complexe dans sa réalisation, a abouti à la production d'une suite d'outils qui s'intègrent dans le processus d'évolution d'architecture. Actuellement, tous les projets de migration utilisent la cartographie des applications, des tests ou le suivi de l'intégration. Il a fallu inventer un certain nombre de solutions, jongler avec l'existant, l'état de l'art et les objectifs initiaux. Mais au final, le résultat me semble satisfaisant. Même s'il reste des chantiers à concrétiser (cf. Tableau 2), grâce à la base « *Migration Platform* », c'est l'ensemble du processus de migration industrielle qui a gagné en visibilité et en robustesse. Les phases de test qui, on peut le dire, étaient souvent les parents pauvres de la migration, bénéficient pleinement de cette réussite. Je pense qu'on peut dire que les objectifs fixés lors de la rédaction du sujet de ce mémoire, ont été atteints. Cette réussite est en grande partie fondée sur la cohérence de la clé qui identifie chaque composant de manière unique. Le respect du format qui la constitue est primordial tout au long du processus de migration.

Pour la suite, il y a principalement deux thèmes qui sont pressentis comme pouvant apporter un réel gain :

- monter en niveau d'abstraction sur les tests de référence, c'est-à-dire produire des tests de niveau fonctionnel, retrouver des actions qui ont du sens pour l'utilisateur.
- le lotissement « intelligent » des applications. Dans mon idée, on pourrait se servir du logiciel *yEd* afin d'obtenir un premier niveau de lotissement basé sur les adhérences entre les éléments (cf. Figure 34). Ensuite, toujours au sein du logiciel *yEd*, un expert fonctionnel serait en mesure d'intervenir pour affiner l'appartenance des éléments à tel ou tel lot. Une fois le lotissement établi, il serait possible d'insérer les données dans la base de cartographie. Ce travail permettrait d'avoir une vision des lots qui composent l'application, puis de déterminer un ordre dans l'intégration des différents lots.

Comme on peut le constater, que ce soit avec la cartographie d'application ou avec la cartographie de test, les axes de recherche sur ces thèmes sont nombreux et prometteurs. La volonté stratégique de Sodifrance pousse à la concrétisation de ces objectifs, d'où l'opportunité pour moi de réaliser ce mémoire sur ces sujets porteurs et innovants. Je remercie encore tous ceux qui, de près ou de loin, ont participé ou ont rendu possible cette aventure.

6 BIBLIOGRAPHIE

Blanc, X., 2005. *MDA en action* 1er éd., Eyrolles.

Bézivin, J., 2004. Sur les principes de base de l'ingénierie des modèles. *RSTI-L'Objet*, 10(4), p.145–157.

Bézivin, J., Barbero, M. & Jouault, F., 2007. On the applicability scope of model driven engineering. Available at: http://atlanmod.emn.fr/www/papers/MDEApplicabilityScope.pdf.

Bézivin, J., Jouault, F. & Valduriez, P., 2004. On the need for megamodels. Available at: http://www.softmetaware.com/oopsla2004/bezivin-megamodel.pdf.

Chinnapongse, V. et al., 2009. Model-Based Testing of GUI-Driven Applications. Dans S. Lee & P. Narasimhan, éd. *Software Technologies for Embedded and Ubiquitous Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, p. 203-214. Available at: http://www.springerlink.com/index/10.1007/978-3-642-10265-3_19 [Consulté août 18, 2011].

Combemale, B., Crégut, X. & Pantel, M., 2007. Transformation de modèles : Principes, Standards et Exemples. Available at: http://combemale.perso.enseeiht.fr/teaching/metamodeling0708/IDM-Transfo_v1.1.pdf [Consulté novembre 15, 2011].

Crucianu, M., Développement d'applications avec les bases de données. Available at: http://cedric.cnam.fr/~crucianm/src/Cours-NFA011.pdf.

Dehlen, V., Madiot, F. & Bruneliere, H., 2008. Representing Legacy System Interoperability by Extending KDM. *Proceedings of the MMSS*, 8. Available at: http://atlanmod.emn.fr/www/papers/IKDMpaper_reduced_final.pdf.

Diaw, S., Lbath, R. & Coulette, B., Etat de l'art sur le développement logiciel dirigé par les modèles. Available at: ftp://ftp.irit.fr/IRIT/MACAO/Article_TSI-IDM-final-coulette.pdf.

Eclipse project CDO, 2011. CDO - Eclipsepedia. Available at: http://wiki.eclipse.org/CDO [Consulté septembre 14, 2011].

FELIX, P., 2011. Test et Validation du Logiciel.pdf (Objet application/pdf). Available at: http://dept-info.labri.fr/~felix/Annee2008-09/S4/McInfo4_ASR%20Tests/1.pdf [Consulté août 18, 2011].

Hernandez, Y. et al., A meta-model to support regression testing of web applications. Available at: http://users.cis.fiu.edu/~clarkep/research/areas/REUPubs/HernandezKingPavaClarke.pdf.

Mirman, F., 2011. Amélioration continue, agilité, management, ... parlons-en: Cycle en V, MOA/MOE ont perdu ! *Amélioration continue, agilité, management,... parlons-en*. Available at: http://fredericmirman.blogspot.com/2010/03/cycle-en-v-moamoe-ont-perdu.html [Consulté septembre 19, 2011].

Object Management Group, 2011a. Architecture-Driven Modernization Task Force. Available at: http://adm.omg.org/ [Consulté septembre 14, 2011].

Object Management Group, 2010. OMG Architecture-Driven Modernization (ADM): Knowledge Discovery Meta-Model (KDM) Specification. Available at: http://www.omg.org/spec/KDM/1.0/PDF/08-01-01.pdf.

Object Management Group, 2011b. OMG/ASTM Architecture-driven Modernization: Abstract Syntax Tree Metamodel (ASTM) Specification. Available at: http://www.omg.org/cgi-bin/doc?formal/11-01-05.pdf.

Object Management Group, 2002a. OMG/MOF 1.4 Meta Object Facility (MOF) Specification. Available at: http://doc.omg.org/formal/2002-04-03.pdf.

Object Management Group, 2006. OMG/MOF 2.0 Meta Object Facility (MOF) Specification. Available at: http://doc.omg.org/formal/2006-01-01.pdf.

Object Management Group, 2002b. OMG/MOF 2.0 Query Views Transformations (QVT) Request For Proposal (RFP). Available at: http://www.omg.org/cgi-bin/doc?ad/02-04-10.pdf.

Object Management Group, 2011c. OMG/MOF 2.0 Query Views Transformations (QVT) Specification. Available at: http://www.omg.org/spec/QVT/1.1/PDF/11-01-01.pdf.

Object Management Group, 2007. OMG/XMI XML Metadata Interrchange (XMI) Specification. Available at: http://www.omg.org/cgi-bin/doc?formal/07-12-01.pdf.

Projet ACCORD, 2011. La démarche MDA. Available at: http://www.infres.enst.fr/projets/accord/lot1/lot_1.1-5.pdf [Consulté octobre 10, 2011].

Régis-Gianas, Y., 2010. Génie Logiciel Avancé. Available at: http://www.pps.jussieu.fr/~yrg/gl/gl-slides-cours-1.pdf [Consulté septembre 19, 2011].

Strohmeier, A., 1996. Cycle de vie du logiciel. *Génie logiciel: principes, méthodes et techniques*, p.1–28.

The Institute of Electrical and Electronics Ehgineers 345 East 47th Street, New York, NY 10017, USA, IEEE Standard Glossary of Software Engineering Terminology, Std 61012-1990. Available at: http://www.idi.ntnu.no/grupper/su/publ/ese/ieee-se-glossary-610.12-1990.pdf.

Villemin, F.-Y., 2011. L'architecture dirigée par les modèles. Available at: http://deptinfo.cnam.fr/Enseignement/CycleSpecialisation/MAI/Doc/MDA11.pdf [Consulté octobre 10, 2011].

Williams, C.E., 2001. Toward a test-ready meta-model for use cases. Dans p. 270–287. Available at: http://subs.emis.de/LNI/Proceedings/Proceedings07/TowaraTest-ReadMeta_19.pdf.

7 ANNEXES

7.1 L'architecture dirigée par les modèles (Model Driven Architecture, MDA)

7.1.1 Introduction

L'initiative d'architecture dirigée par les modèles de l'OMG « *Model Driven Architecture* » (MDA) est motivée par les besoins de réduction des coûts de reconception et de maintenance des applications informatiques. Dans les années 1990, CORBA, spécification de l'OMG, devait fournir un environnement standard et ouvert permettant à tout type d'application d'interopérer avec les autres.

Il a été suivi d'autres intergiciels (EJB, DCOM, Web services). Paradoxalement, cette succession d'intergiciels, prévus initialement pour simplifier les communications entre applications réparties, a produit l'effet inverse. En effet, une application utilisant un intergiciel en devient fortement dépendante. Dès lors, il est extrêmement difficile de changer d'intergiciel.

MDA tente de répondre à cette problématique en effectuant une séparation entre le métier des applications, et les techniques informatiques utilisées pour les réaliser, que cela concerne les plates-formes d'exécution ou encore les langages de programmation.

7.1.2 Philosophie du MDA

7.1.2.1 Les avantages du MDA

Les avantages attendus de MDA étaient alors :

- De pérenniser les savoirs faires.

  Les métiers des entreprises n'évoluent que très peu en comparaison des technologies informatiques utilisées pour concevoir les applications. Il est donc évident que le fait de séparer les spécifications métier des spécifications techniques va dans la bonne direction.

- De gagner en productivité

  L'évolution de la modélisation « simple » qui consistait à avoir quelques schémas à la fin de la phase d'analyse, la plupart du temps obsolètes dès le début de la phase de conception, vers le MDA apporte un réel gain de productivité et rejoint le premier point : la pérennité des modèles. L'automatisation des transformations du MDA permet d'obtenir des gains de productivité sur des opérations de générations de code, de documentation, d'exécution de tests ou de validations.

- Prise en compte des plates-formes d'exécution.

  MDA prend en compte le fait qu'une application peut être déployée sur différentes plates-formes, par exemple, une parte sur du J2EE, et une autre partie sur du .Net.

La principale préconisation pour remplir ces objectifs, outre la séparation entre les exigences métier et les techniques de réalisation, est l'utilisation des modèles pour représenter l'information.

7.1.2.2 Domaine d'application du MDA

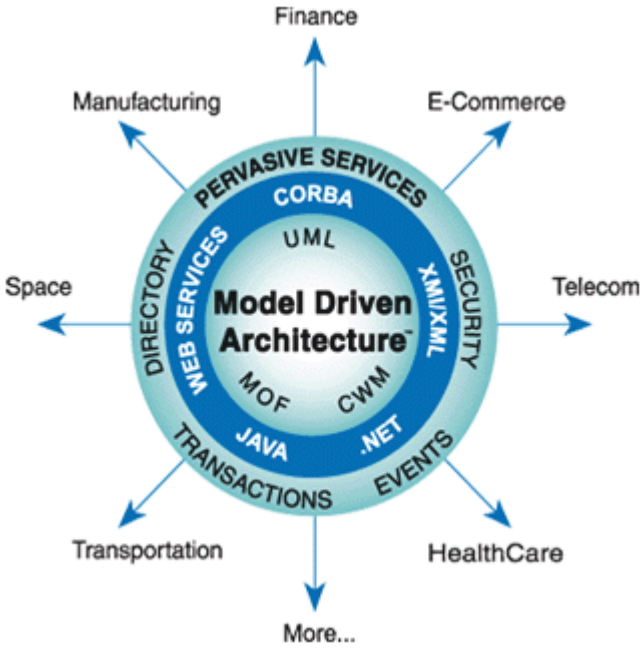


Figure 35 : *Model Driven Architecture* (Projet ACCORD 2011)

Mais quel est le domaine d'application du MDA ?
Les outils proposant de plus en plus d'opérations sur les modèles (générations, tests, validations, etc.), où commence et où s'arrête le MDA ? Comme on va le détailler un peu plus loin, MDA n'a d'autre préconisations que l'utilisation des modèles, qu'il classe en modèle d'exigence (*Computational Independent Model*, CIM), modèle d'analyse ou de conception (*Platform Independent Model*, PIM) et modèle de code (*Platform Specific Model*, PSM). MDA n'est pas une méthode mais une approche. A chaque entreprise de définir sa propre méthode pour appliquer au mieux les préconisations de MDA à son contexte.

7.1.3 Architecture du MDA

7.1.3.1 Les principaux modèles du MDA

Les principaux modèles du MDA sont :

- Le modèle des exigences (*Computational Independent Model*, CIM)
  Il s'agit d'un modèle de haut niveau qui représente l'application et qui permet de définir les services qu'elle va offrir et les relations qu'elle aura avec les autres entités. Ces modèles ont pour objectif d'être pérennes et de refléter la relation entre les exigences du client et l'application. Dans les CIM, aucune information sur le fonctionnement de l'application ne doit être incluse. En UML, on peut représenter un modèle d'exigences avec un diagramme de cas d'utilisation. Il offre la faculté de définir l'ensemble des acteurs et des cas d'utilisation sans en détailler le fonctionnement.
- Le modèle d'analyse ou de conception abstraite (*Platform Independent Model*, PIM).
  Il a pour objectif de structurer l'application en modules et sous-modules. Ces modèles font le lien entre le modèle des exigences et le code de l'application et se doivent d'être pérennes. Pour ce faire, ils ne doivent pas contenir d'informations sur les plates-formes ou langages qui seront utilisés.
- Le modèle des plates-formes (*Platform Dependent Model*, PDM)
  Ce modèle sert à décrire une architecture technique qui sera appliquée au PIM pour obtenir le PSM.
- Le modèle de code (*Platform Specific Model*, PSM).
  Contrairement au modèle d'analyse ou de conception, le modèle de code est lié à une plate-forme d'exécution. Il sert principalement à faciliter la génération de code. Les PSM peuvent être obtenus par application de profils UML, c'est-à-dire l'adaptation d'UML à un domaine particulier, ou par l'utilisation de modèles de plates-formes (PDM) lors de la transformation du PIM. Etant liés à une plate-forme d'exécution, les modèles de code n'ont pas pour vocation d'être pérennes.

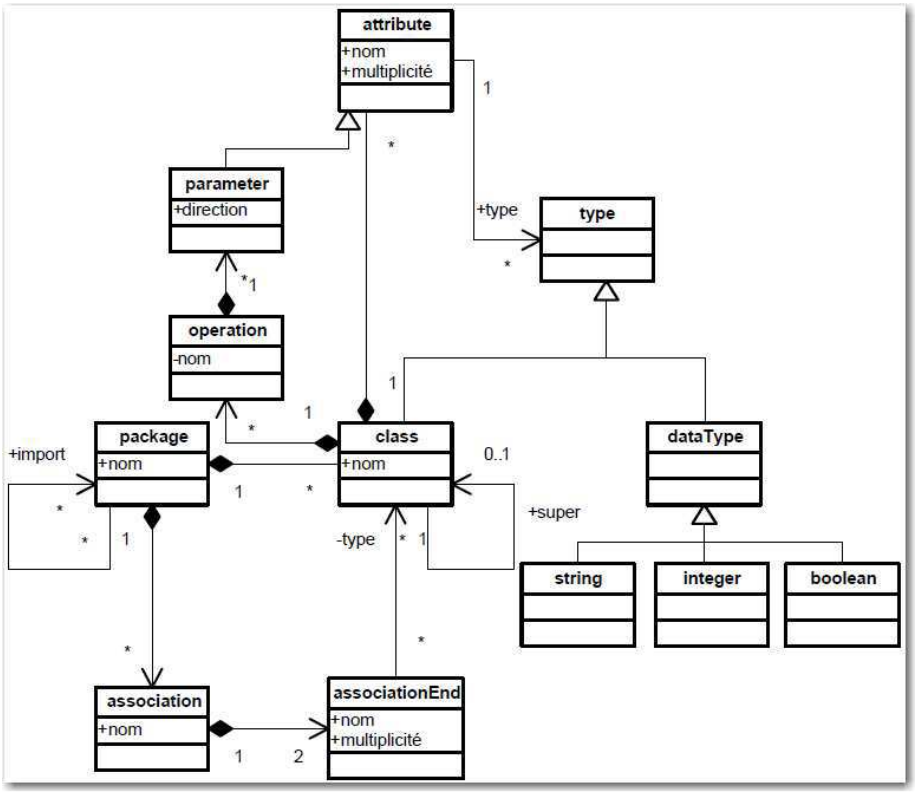


Figure 36 : diagramme de classes du MOF1.4

(Blanc 2005, p.39)(Object Management Group 2002a)

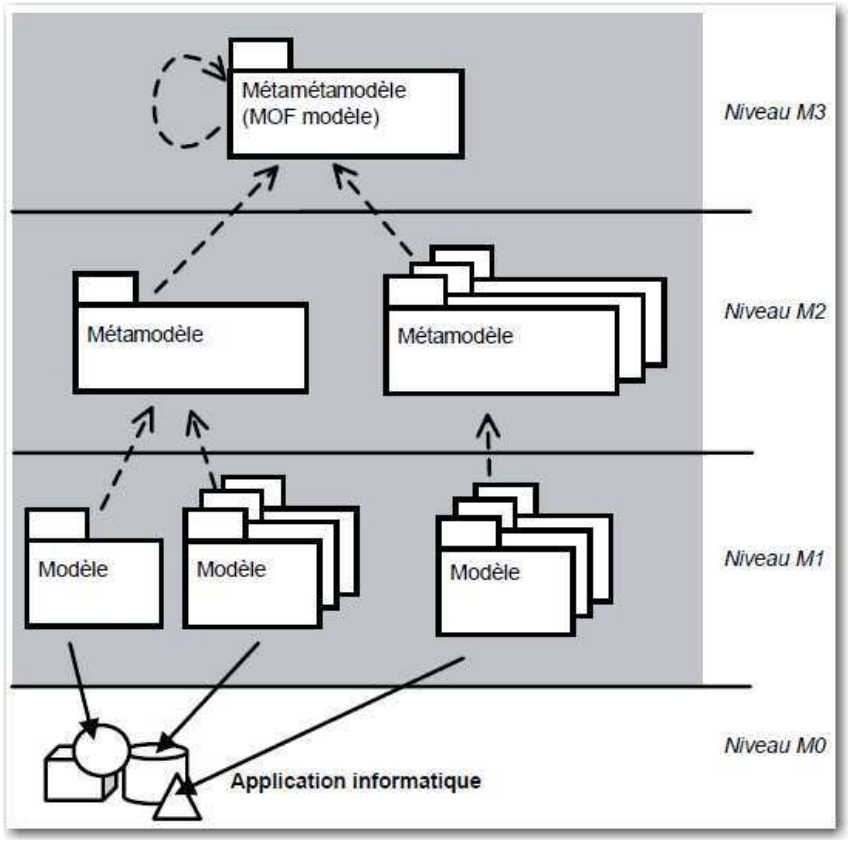


Figure 37 : les quatre niveaux de l'architecture du MDA (Blanc 2005, p.40)

7.1.3.2 L'architecture du MDA

MDA définit son formalisme de modélisation, c'est-à-dire qu'il décrit la façon dont un modèle doit être structuré. Pour cela il utilise encore un modèle, qui sera le métamodèle du modèle à définir. Cela pose inévitablement la question : peut-on remonter indéfiniment dans la hiérarchie des modèles ? La réponse est non. Dans MDA, le métamétamodèle, modèle du niveau le plus élevé s'appelle « *Meta Object Facility* » (MOF) et a la particularité de s'autodéfinir. En effet, si on devait réaliser le diagramme de classe du métamodèle du MOF, on obtiendrait le diagramme de classe de la Figure 36, qui n'est autre que le diagramme de classe du MOF lui-même. MDA limite ainsi son architecture à quatre niveaux (cf. Figure 37). Dans sa version 2.0, le MOF est constitué de deux parties : EMOF (*Essential MOF*), pour l'élaboration des métamodèles sans association, et CMOF (*Complete MOF*) pour l'élaboration des métamodèles avec associations (Diaw et al. s. d.).
Il faut bien comprendre que cette architecture ne sert pas à indiquer à quel niveau appartiennent les PIM et autres PSM, mais à garantir une cohérence entre les modèles utilisés. Le MOF, de niveau M3, défini la structure que chaque métamodèle de niveau M2 devra respecter. Ensuite chaque métamodèle défini à son tour la structure des modèles de niveau M1.

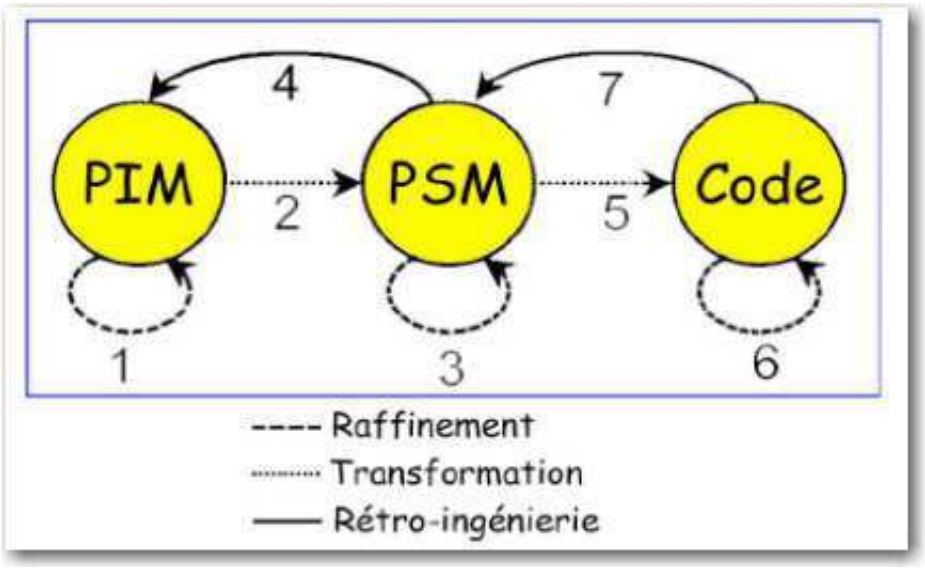


Figure 38 : Les transformations des modèles MDA (Villemin 2011, p.12)

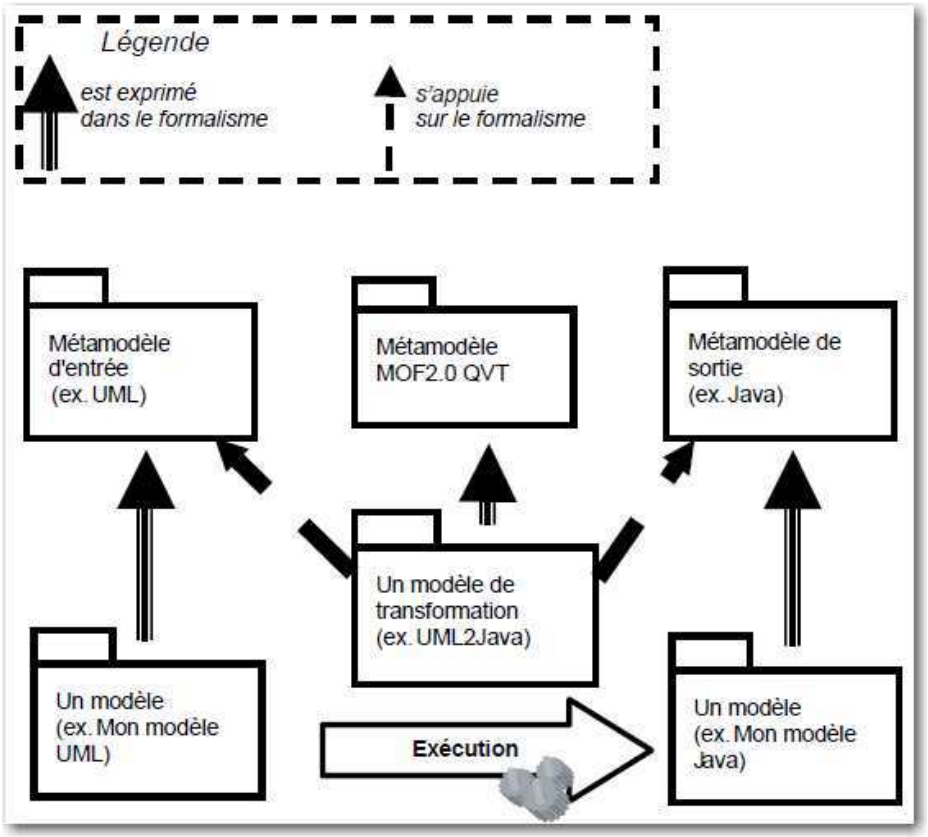


Figure 39 : transformations de modèles (Blanc 2005, p.11)

7.1.4 Les transformations

7.1.4.1 Les transformations de modèles

Le passage entre les différents modèles présentés auparavant se fait par le biais de l'exécution de transformations. La Figure 38 illustre les différentes transformations que l'on peut trouver généralement dans MDA.

La transformation ou projection et la retro-ingénierie peuvent être des transformations entre deux modèles conformes à un même métamodèle, on parle alors de transformation endogène, ou entre des modèles respectant des métamodèles différents, il s'agit dans ce cas d'une transformation exogène. Le raffinement est une transformation endogène un peu particulière, car elle s'appuie non seulement sur un métamodèle commun au modèle d'entrée et de sortie de la transformation, mais c'est aussi le modèle d'entrée qui sert de modèle de sortie.

La transformation sert à passer d'un niveau abstrait vers un niveau plus concret, plus on avance dans les transformations, plus on se rapproche du modèle de code.

La retro-ingénierie fait le trajet inverse et remonte des niveaux concrets jusqu'aux niveaux abstraits.

Le raffinement sert le plus souvent à ajouter de l'information sans changer le sens des objets.

MDA préconise d'utiliser les modèles dans son approche, et c'est tout naturellement qu'il propose de modéliser les transformations elles même. Une transformation est vue comme une application, avec ses exigences, ses modèles de conception et de code. Afin de modéliser les transformations, MDA définit le standard MOF2.0 *Query View Transformation* (QVT)(Object Management Group 2011c). QVT est le métamodèle décrivant les modèles de transformation. QVT se limite aux transformations entre modèles.

La Figure 39 illustre une transformation entre deux modèles, le modèle UML en entrée, qui est conforme au métamodèle UML, et le modèle Java en sortie, qui pour sa part est conforme au métamodèle Java. Le modèle de transformation UML2Java se doit d'être conforme au métamodèle QVT.

7.1.4.2 Query Views Transformations (QVT)

QVT devait répondre aux propositions de l'OMG QVT *Request For Proposal* (Object Management Group 2002b) dont voici les principales exigences (Combemale et al. 2007) :

- normaliser un moyen d'exprimer des correspondances (transformations) entre langages définis avec MOF.
- exprimer des requêtes (*Query*) pour filtrer et sélectionner des éléments d'un modèle (y compris sélectionner les éléments source d'une transformation).
- proposer un mécanisme pour créer des vues (*Views*) qui sont des modèles déduits d'un autre pour en révéler les aspects spécifiques.
- formaliser une manière de décrire des transformations (*Transformations*).

Le standard OMG : MOF QVT (Combemale et al. 2007) :

- Syntaxe abstraite en MOF 2.0 (et syntaxe concrète textuelle et graphique)
- Possibilité de plusieurs modèles (conformes à des métamodèles issus de MOF)
- Langage de requête s'appuyant sur OCL
- Gestion automatique des liens de traçabilité
- MOF QVT devra permettre plusieurs scenarios d'exécution (transformations unidirectionnelles, multidirectionnelles, etc.).

Dans sa dernière version, QVT 2.0 propose trois langages de transformation (cf. Figure 40) :

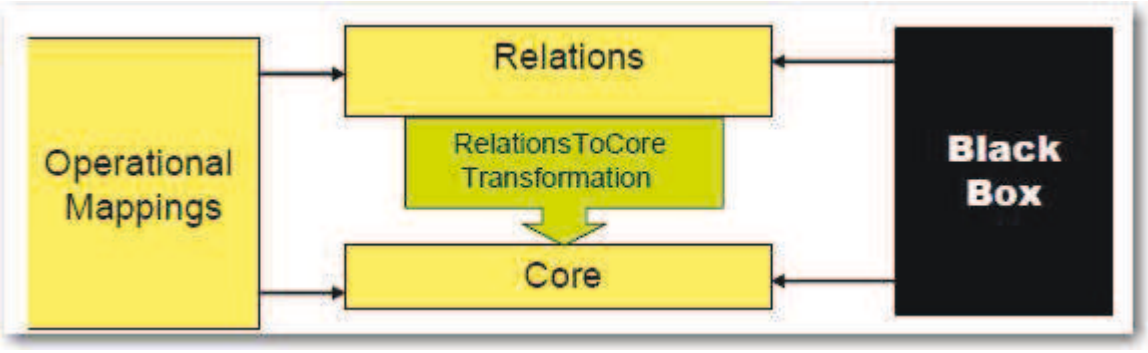


Figure 40 : Les relations entre les métamodèles de QVT (Object Management Group 2011c)

La partie déclarative de QVT est structurée en une architecture à deux couches composée de :

- « Relations » qui est un langage de haut niveau orienté utilisateur. Il permet d'établir des relations entre les modèles MOF participant à la transformation. Les traces entre les éléments de modèles impliqués dans les transformations sont créées implicitement. « Relations » peut être utilisé soit comme formalisme de représentation des relations entre modèles soit pour être traduit en modèle « *Core* » par une transformation « *RelationsToCore* ».
- « *Core* » qui est un langage technique de bas niveau défini par une syntaxe textuelle. Contrairement à « Relations », toutes les traces de transformations doivent être décrites afin d'être créées.

La partie impérative est composée pour sa part de :

- « *Operational Mappings* » qui est une extension de « *Relations* » et de « *Core* » avec des constructions impératives (extension d'OCL). Cela autorise une syntaxe concrète et un plus procédural à destination des programmeurs standards.
- « *Black Box MOF Operation* » qui permet d'invoquer des fonctionnalités de transformation implémentées dans un langage externe.

La combinaison de ces trois langages donne un « langage hybride ».

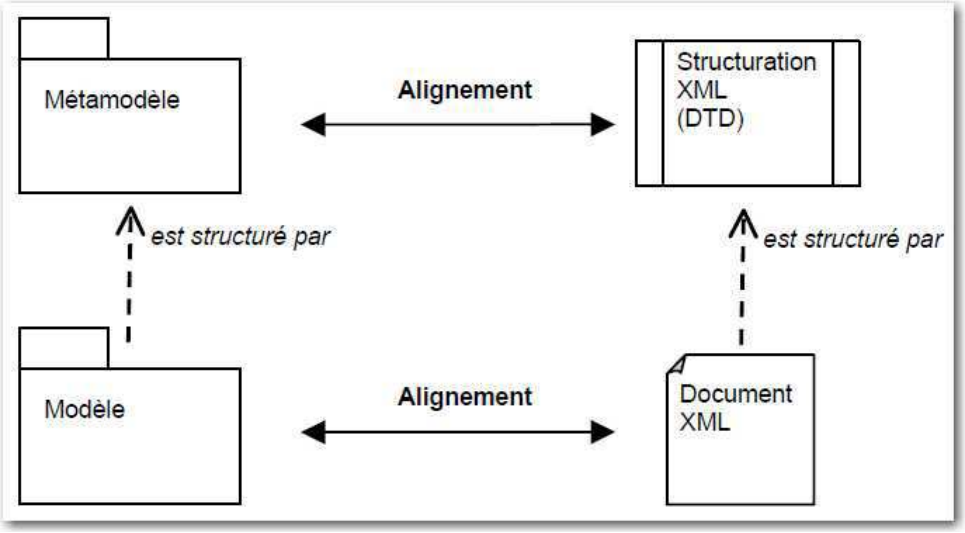


Figure 41 : Alignement entre modèle/métamodèle et DTD/document XML (Blanc 2005, p.103)

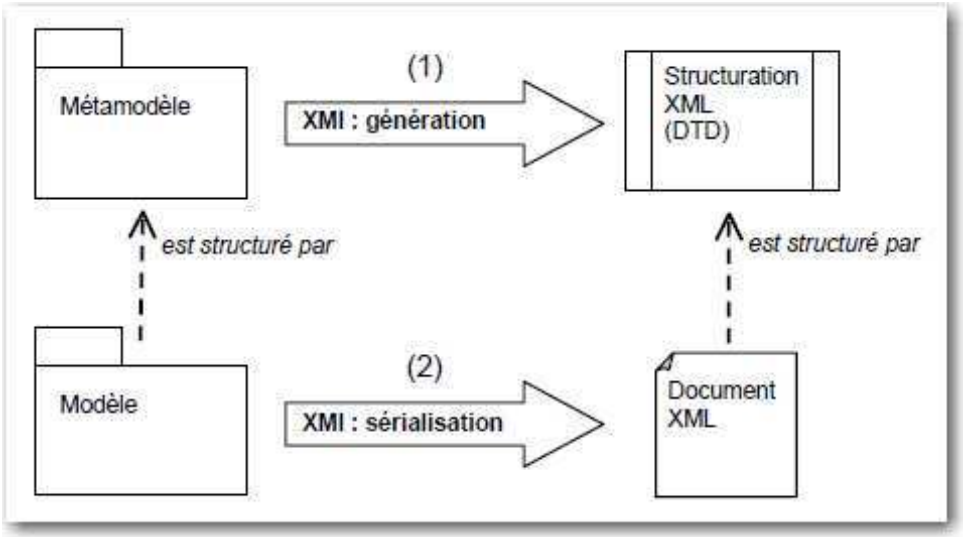


Figure 42 : XMI et la structuration des balises XML (Blanc 2005, p.104)

7.1.4.3 XMI

Les modèles ne disposant pas de représentation concrète, l'OMG a décidé de standardise *XML Metadata Interchange* (XMI)(Object Management Group 2007). Ce format permet de représenter un modèle sous forme de document XML. De même qu'en MDA un modèle est conforme à un métamodèle, en XML, un document peut être conforme à une DTD[30] ou à un schéma de définition XML (XSD[31]). MDA s'est servi de cette analogie pour établir un alignement entre métamodèle et DTD d'une part, et modèle et document XML d'autre part (cf. Figure 41). Cet alignement permet de définir une génération du métamodèle en DTD (cf. Figure 42, 1) et une sérialisation du modèle en XML (cf. Figure 42, 2) et de profiter ainsi du mécanisme de validation des documents XML.

Les documents XML peuvent être structurés par une DTD ou un schéma XML, cela dépend de la version du XMI (DTD ≤ XMI 1.2 ≤ XSD).

7.1.5 L'avenir du MDA ?

Aujourd'hui, les réflexions évoluent pour définir ce que M. Bézivin nomme les mégamodèles (Bézivin et al. 2004). Il s'agit de monter encore en niveau d'abstraction et de voir les modèles, les métamodèles et les transformations qui composent le MDA comme les éléments d'un modèle. Ce modèle serait bien sûr conforme à un métamodèle : le mégamodèle. Le respect de l'architecture à quatre niveaux du MDA garanti la cohérence des modèles qui composent une application mais ne gère pas forcément les relations qui pourraient exister entre des modèles d'applications distinctes. Le mégamodèle tente de combler cette lacune et a pour objectif de faciliter les échanges et le partage d'informations entre les différents acteurs qui utilisent l'approche MDA.

7.1.6 Conclusion

Le respect des préconisations du MDA (séparation entre les exigences métier d'une application et son implémentation sur une plate-forme donnée, le respect de l'architecture MDA à quatre niveaux, utilisation des transformations automatisées) permet d'obtenir un haut niveau de qualité des applications ainsi produites. La maturité atteinte par le MDA en fait un des modes de développement et/ou de maintenance des applications très attractif. Il est vrai qu'au niveau d'une entreprise, l'investissement initial n'est pas neutre (compréhension, formation, etc.), mais en général, les gains de productivité couvrent cet effort.

[30] DTD : *Document Type Definition*
[31] XSD : *XML Schema Definition*

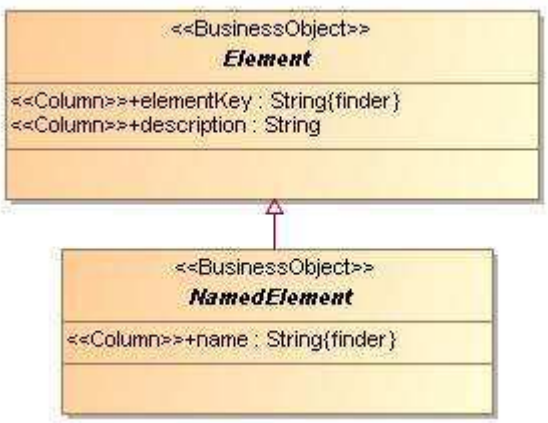


Figure 43 : CoreDiagram Diagram (Source Sodifrance, métamodèle « Migration Platform »)

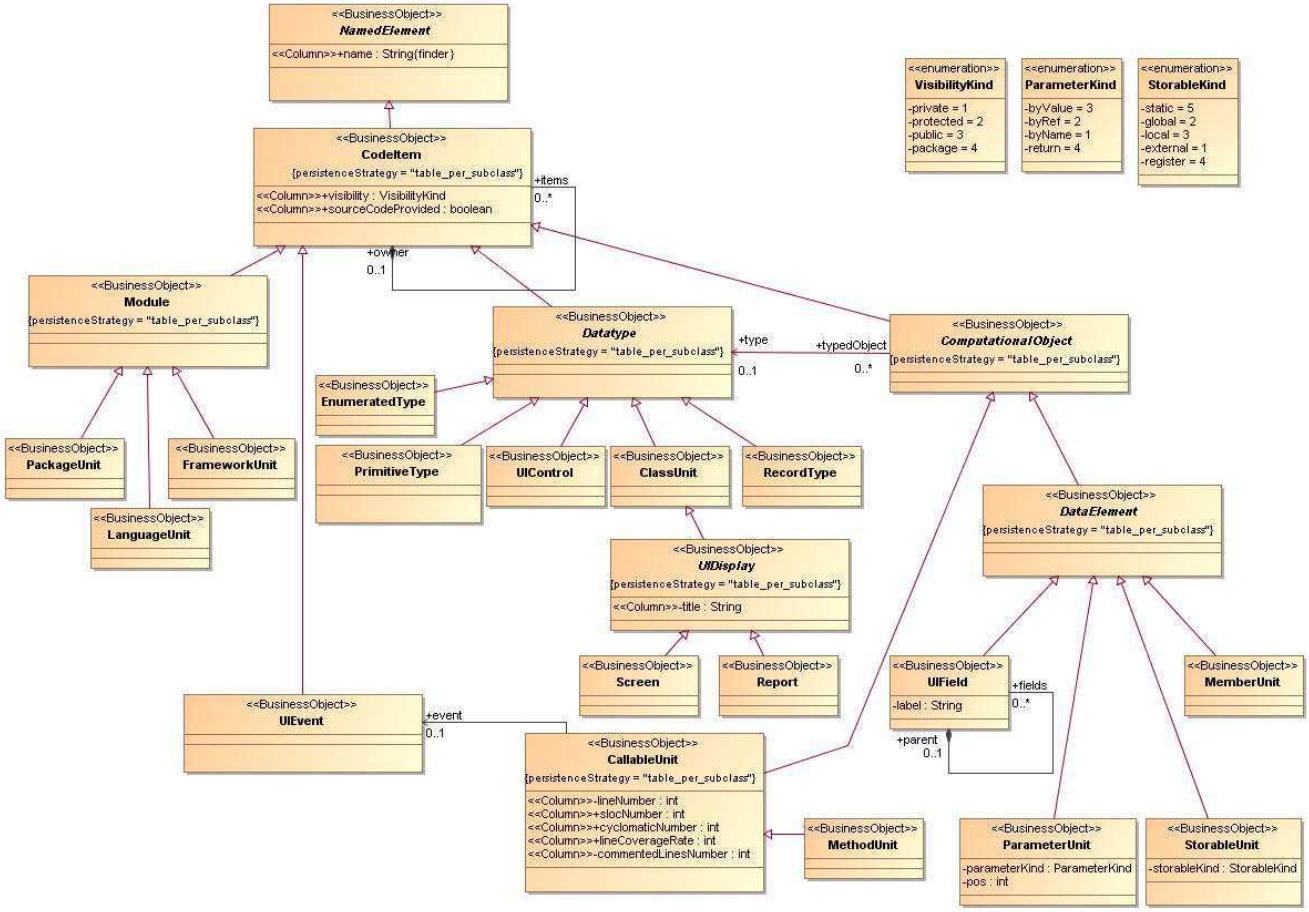


Figure 44 : CodeItemsDiagram Diagram (Source Sodifrance, métamodèle « Migration Platform »)

7.2 Documentation partielle du métamodèle « Migration Platform »

La documentation qui suit est directement extraite des informations présentes dans le modeleur UML *MagicDraw* sur les classes du méta-modèle « Migration Platform ».

7.2.1 Core

Composition (Figure 43)

- NamedElement - A named element represents elements with names.
- Element - An element is an atomic constituent of a model. Element is an abstract element.

7.2.2 CodeItems

Composition (Figure 44)

- CodeItem - CodeItem class represents the named elements determined by the programming language.
- Module - The Module class is a generic modeling element that represents an entire software module or a component, as determined by the programming language and the software development environment. A module is a discrete and identifiable program unit that contains other program elements and may be used as a logical component of the software system. Usually modules promote encapsulation (i.e., information hiding) through a separation between the interface and the implementation. In the context of representing existing software systems, modules provide the context for establishing the associations between the programming language elements that are owned by them, especially when the same logical component of a software product line is compiled multiple times with different compilation options and linked into multiple executables. Instances of the Module class represent the logical containers for program elements determined by the programming language.
- PackageUnit - The PackageUnit class is a subtype for Module that logical collections of program elements, as directly supported by some programming languages, such as Java.
- ClassUnit - The ClassUnit is an element that represents user-defined classes in object-oriented languages.
- Datatype - Datatype class represents the named elements determined by the programming language that describes datatypes.

- LanguageUnit - LanguageUnit is a logical container that owns definitions of primitive and predefined datatypes for a particular language, as well as other common elements for a particular programming language.
- MethodUnit - The MethodUnit represents member functions owned by a ClassUnit.
- CallableUnit - The CallableUnit represents a basic stand-alone element that can be called, such as a procedure or a function.
- DataElement - The DataElement class is a generic element that defines the common properties of several concrete classes that represent the named data items of existing software systems (for example, global and local variables, record files, and formal parameters).
- ParameterUnit - ParameterUnit class is a concrete subclass of the DataElement class that represents a formal parameter; for example, a formal parameter of a procedure.
- StorableUnit - StorableUnit represents a variable of existing software system - a computational object to which different values of the same datatype can be associated at different times. From the runtime perspective, a StorableUnit element represents a single computational object, which is identified either directly (by name) or indirectly (by reference). StorableUnit represents both global and local variables.
- ComputationalObject - ComputationalObject class represents the named elements determined by the programming language, which describe certain computational objects at the runtime, for example, procedures, and variables.
- PrimitiveType - The PrimitiveType is a generic element that represents primitive data types determined by various programming languages.
- Screen - The Screen is a compound unit of display, such as a Web page or character-mode terminal that is used to present capture information.
- Report - The Report is a compound unit of display, such as a printed report, that is used to present information.
- UIDisplay - The UIDisplay is the superclass of Screen and Report. It represents a compound unit of display.
- UIControl - The UIControl is a graphical control (text boxes, combo boxes, panel, menu…).
- UIEvent - The UIEvent class is an element representing events provided by a UIControl.

- FrameworkUnit - FrameworkUnit is a logical container that owns definitions of primitive and predefined datatypes for a given framework, which may be either off-the-shelf (distributed as a black-box component) or proprietary (source files may then be provided).
- ParameterKind - ParameterKind datatype defines the kind of parameter passing conventions.
- StorableKind - StorableKind enumeration data type defines several common properties of a StorableUnit related to their life-cycle, visibility, and memory type.
- VisibilityKind - VisibilityKind enumeration data type defines several common properties of a CodeItem related to its visibility.
- RecordType -
- EnumeratedType -
- UIField - UIField class is a concrete subclass of the DataElement class, and represents a graphical component. It can be composed by a set of UIField, for a window by example.
- MemberUnit - MemberUnit class is a concrete subclass of the DataElement class that represents a member of a class type.

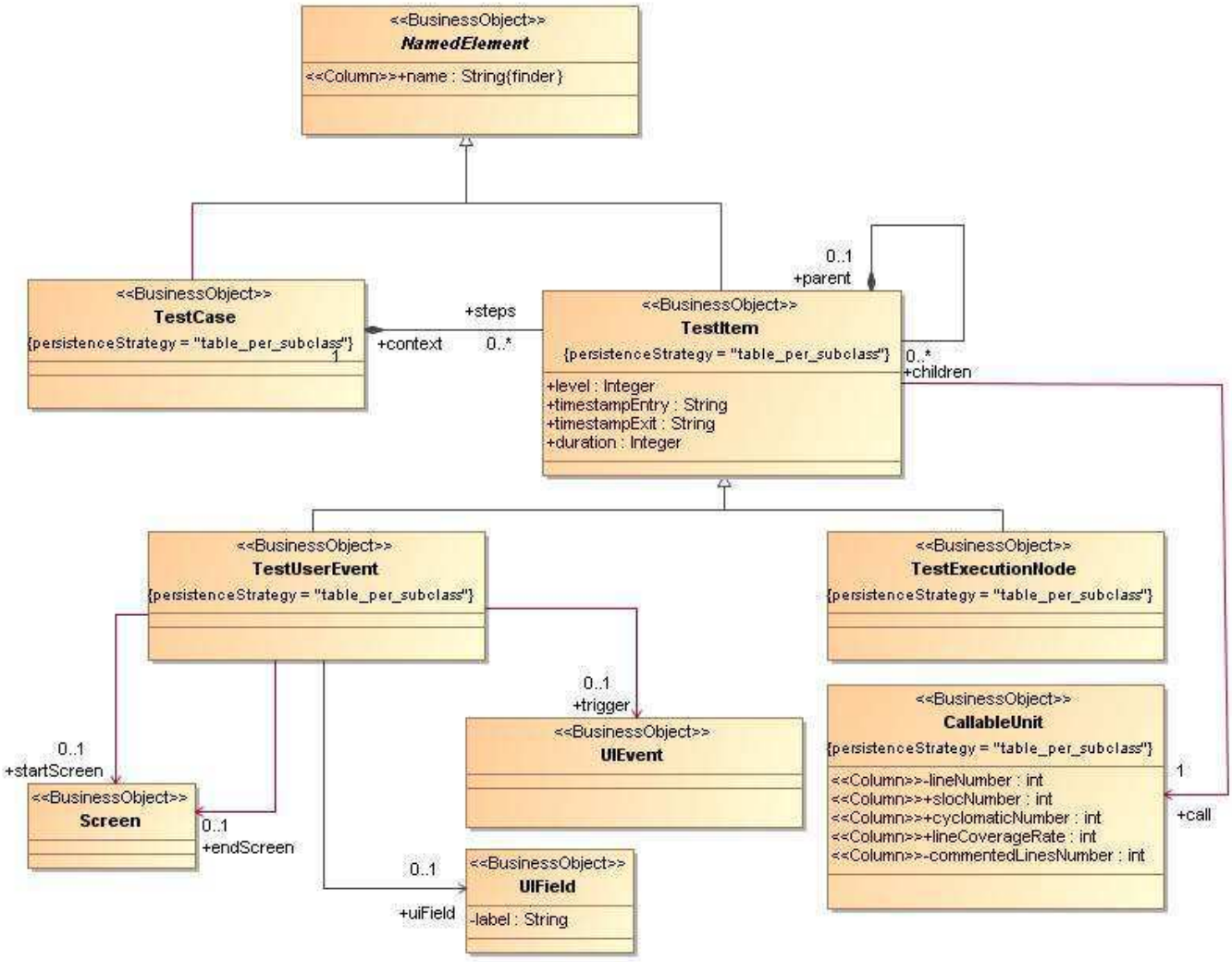


Figure 45 : TestArchitectureDiagram Diagram

7.2.3 Test Architecture

Composition (Figure 45)

An individual line of a Test Case. Each Test Step should include instructions and an expected result.

- TestItem - The TestItem is the superclass of TestUserEvent and TestExecutionNode. It represents a test step.
- TestUserEvent - The TestUserEvent is used to capture user actions on the application.
- TestExecutionNode - The TestExecutionNode is used to capture the behavior of the application in a test step.
- TestCase - A test case is a sequence of steps to test the correct behaviour/functionalities, features of an application.

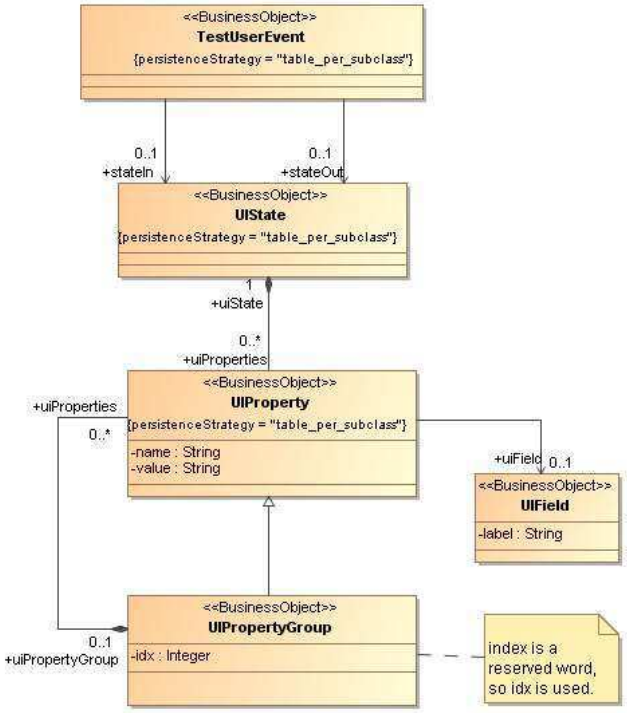


Figure 46 : TestDataDiagram Diagram

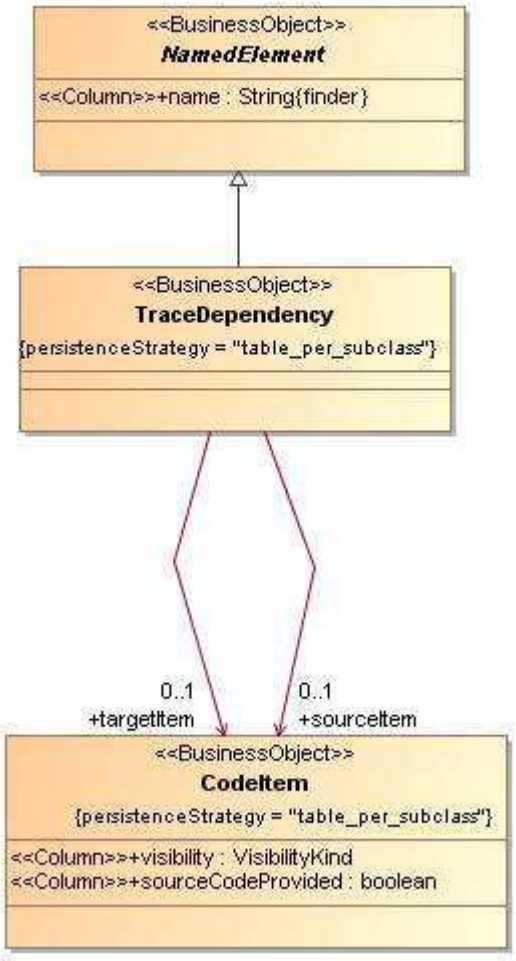


Figure 47 : TraceabilityDiagram Diagram

7.2.4 Test Data

Composition (Figure 46)

- UIState - An UIState represents the state of a screen at the beginning (stateIn) or the end (stateOut) of an event. It's composed by a set of UIProperty.
- UIProperty - An UIProperty represents a property of an UIField. It allows capturing the value of all properties of an UI component at the beginning or the end of an UIEvent.
- UIPropertyGroup - UIPropertyGroup specializes UIProperty. It represents the upper level of a set of UIProperty. By example, a grid is composed by a set of rows, which are composed by a set of columns.

7.2.5 Traceability

Composition (Figure 47)

- TraceDependency - A TraceDependency indicates that the target CodeItem is the result of the transformation of the source CodeItem.

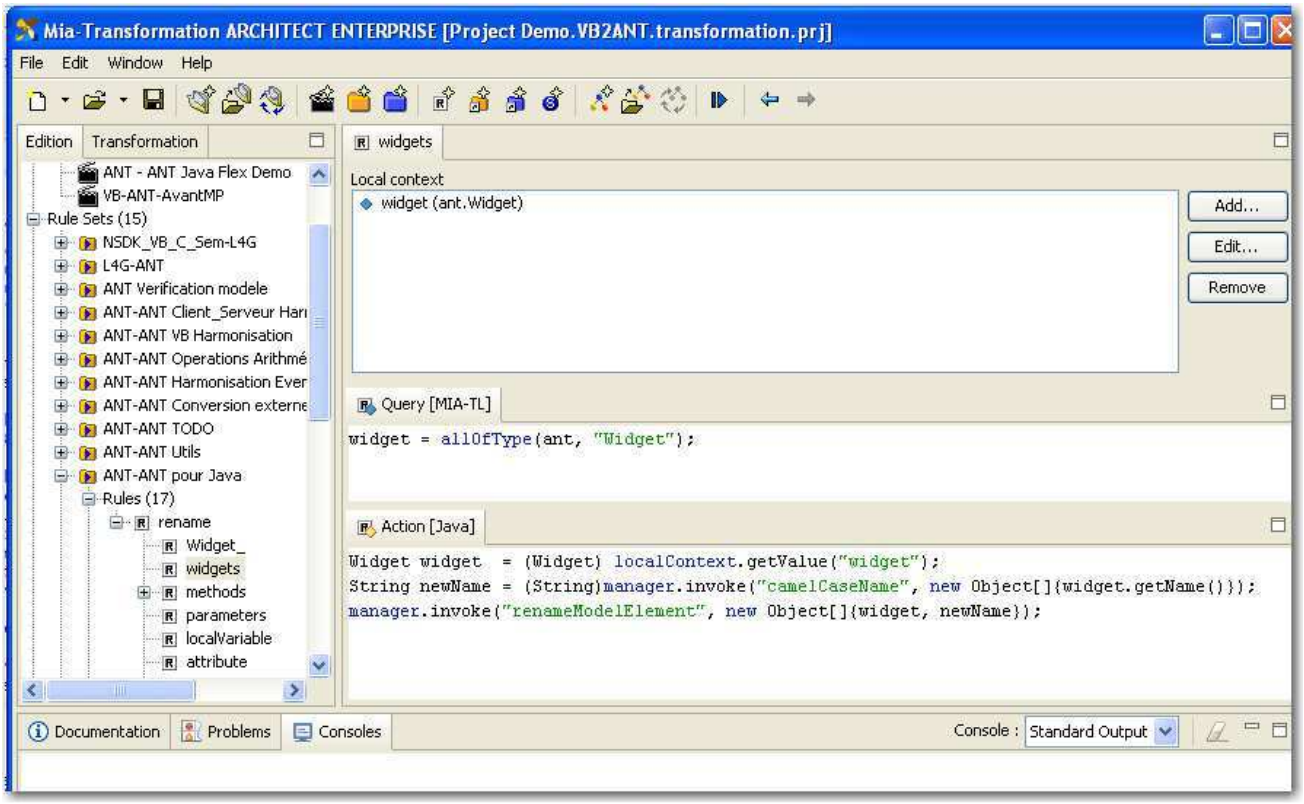


Figure 48 : Copie d'écran du logiciel *MIA Transformation* en mode développement

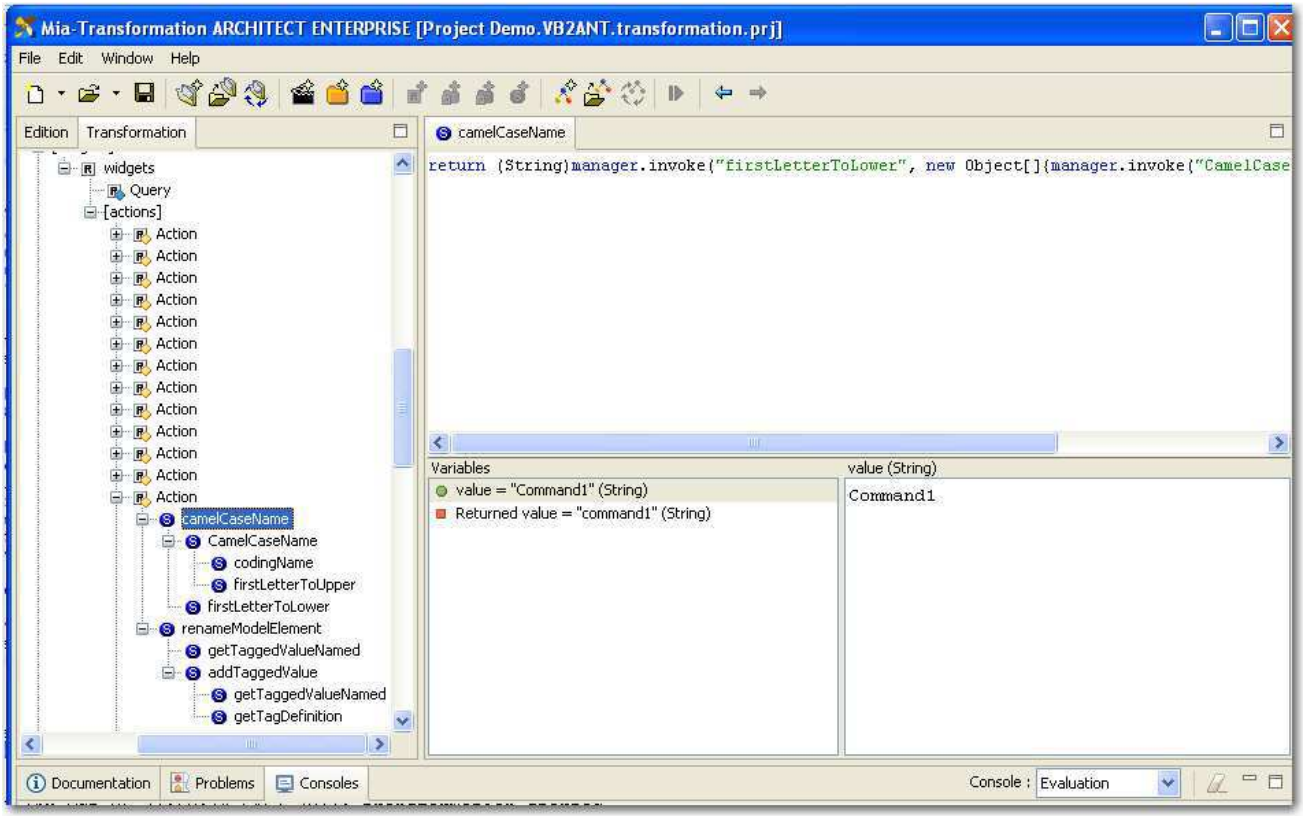


Figure 49 : Copie d'écran du logiciel *MIA Transformation* en mode trace

7.3 *MIA Transformation*

MIA Transformation est un environnement de développement permettant l'écriture de scripts et l'exécution de ces scripts afin d'effectuer des transformations de modèles. Ces modèles doivent respecter un métamodèle supporté par l'outil afin de pouvoir être lus. Ensuite, il faut définir des règles de transformation par le biais de script écrits soit en Java, soit en *MIA-TL*, un langage simplifiant l'écriture des scripts. La Figure 48 illustre l'écriture d'une règle nommée « `rename widgets` ». Pour toutes les occurrences des objets de type « `widget` », ainsi que pour ses sous-types dans le modèle, les scripts « `camelCaseName` » et « `renameModelElement` » seront appliqués. La Figure 49 montre la trace résultant de l'exécution de cette règle. On peut observer la valeur en entrée du script « `camelCaseName` », et la valeur retournée. Ce petit exemple démontre de manière simple la façon dont on peut agir sur les éléments du ou des modèles traités. Bien entendu, les scripts peuvent être bien plus élaborés, affinant les recherches selon plusieurs propriétés, effectuant des actions plus complexes qu'une simple modification du nom, comme par exemple l'ajout d'instructions.

Deux exemples de transformations qui peuvent être assez complexes à réaliser :

- Les fonctions sont mal typées et ne renvoient qu'un type « `object` ». Résolution du type de la fonction et affectation au type de retour (« `ReturnStatement` »).
- Les requêtes SQL sont sous la forme de chaînes de caractères dans le code source, analyse de la chaîne en question, extraction des paramètres en entrée et en sortie de la requête. A partir de cette analyse, création d'une couche d'accès aux données « propre » avec par exemple, une classe par table, une méthode par requête, et positionnement des paramètres correspondant aux variables nécessaires.

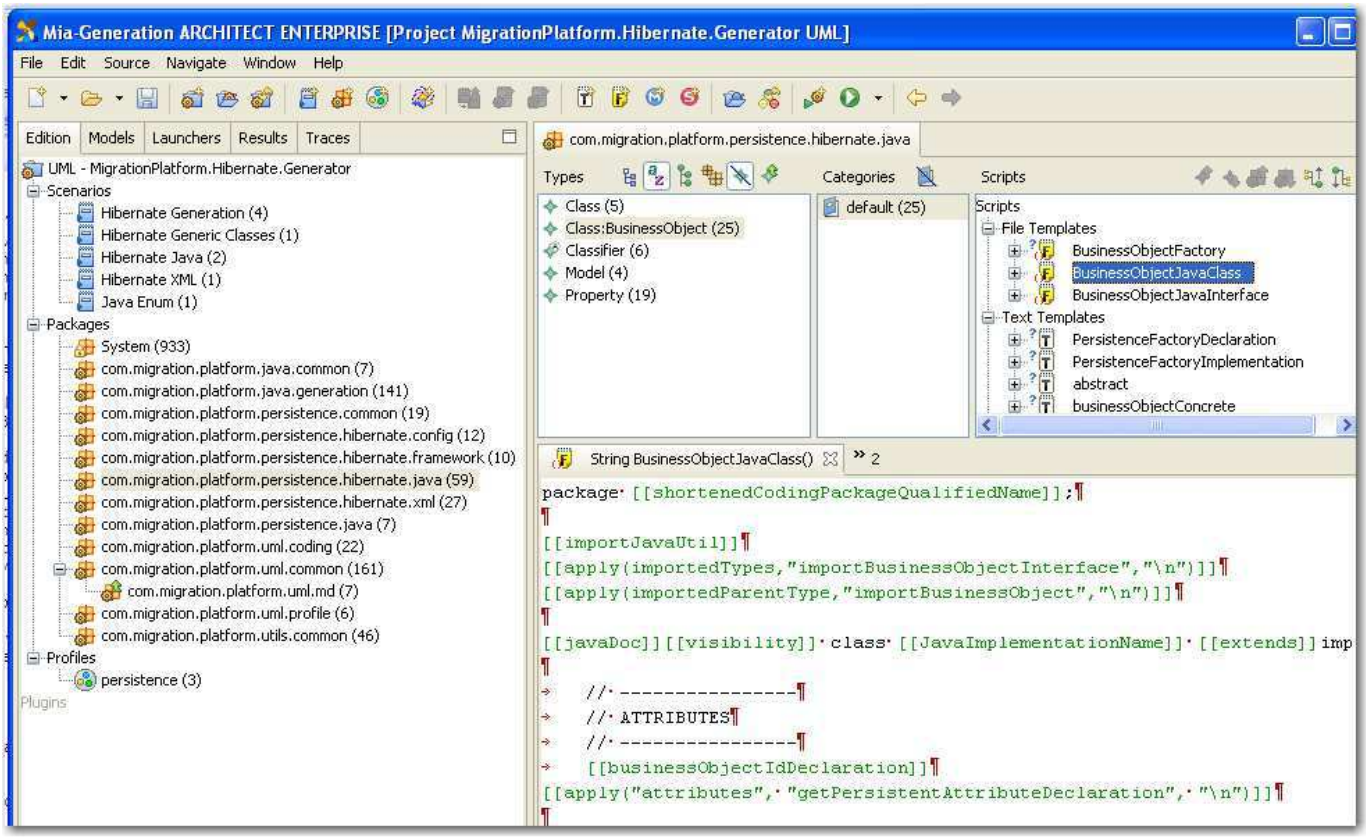


Figure 50 : Copie d'écran du logiciel *MIA Generation* en mode développement

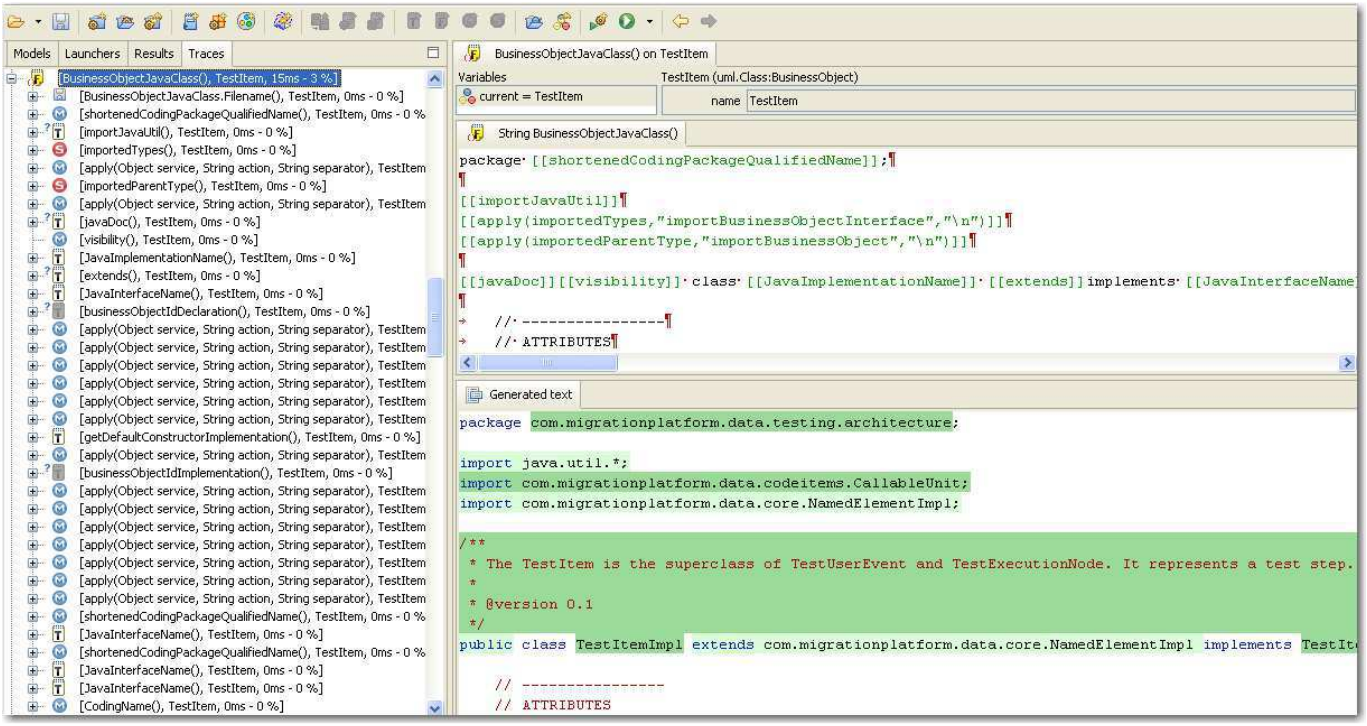


Figure 51 : Copie d'écran du logiciel *MIA Generation* en mode trace

7.4 *MIA Generation*

L'outil *MIA Generation* est le complément idéal de *MIA Transformation*. Dans le cadre de nos évolutions d'architecture, c'est sur lui que repose la phase de génération à partir des modèles issus des phases de transformation. Même si le principe se rapproche de *MIA Transformation*, des scripts positionnés sur des objets du métamodèle, *MIA Generation* est plus simple à appréhender, car beaucoup plus concret. A l'issue d'une génération, on obtient immédiatement du code. Si on modifie un script, on peut tout de suite vérifier si le résultat convient. La Figure 50 détaille l'environnement de développement de *MIA Generation*. Sur la partie gauche de l'écran, on trouve l'ensemble des `Packages`, ils contiennent les scripts qui sont de quatre types :

- *File template* : ce sont les scripts qui produisent les fichiers
- *Text template* : scripts qui renvoient du texte. Ils peuvent appeler d'autres « *Text template* » ou « *Macro* ».
- *Macro* : scripts java qui renvoient la plupart du temps du texte.
- *Service* : scripts java qui renvoient des collections d'objets.

Le script de l'exemple traite les classes stéréotypées « `BusinessObject` ». Donc pour toutes les occurrences d'objets de ce type dans le modèle, le « *File Template* » « `BusinessObjectJavaClass` » sera appelé. Il produira un fichier qui respectera le format du script. Dans la partie en bas à droite de l'écran, on voit le contenu du script. Les portions écrites en noir seront générées à l'identique dans le fichier. Les portions entre double crochets sont des appels soit à des « *Text Template* », soit à des *Macros*, qui produisent eux-mêmes du texte qui sera aussi généré dans le fichier cible. La Figure 51 illustre ce propos. Il s'agit du résultat de la génération du *File template* `BusinessObjectJavaClass` sur la classe « `TestItem` » du modèle. La partie en haut à droite reprend le « *File Template* » responsable de la génération. Dans la partie en bas à droite, on a le résultat de la génération.

On voit que le texte `package` est repris tel quel, alors que le *Text Template* « `shortenedCodingPackageQualifiedName` » renvoi le texte suivant : « `com.migrationplatform.data.testing.architecture` » qui est construit à partir des informations du modèle.

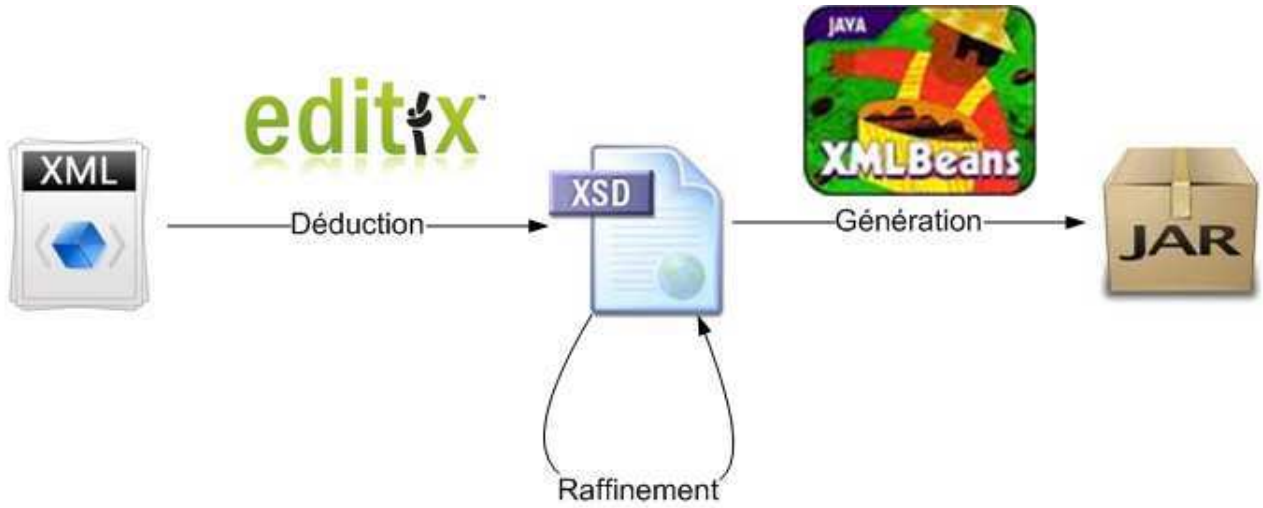


Figure 52 : Processus de constitution du Jar à partir d'un flux XML.

7.5 Du XML au Jar

Comme indiqué précédemment dans le document, pour que les briques qui composent la chaîne d'évolution d'architecture soient facilement intégrées, le plus simple est qu'elles soient au format archive Java (Jar). Dans le cas présent, l'objectif de ce Jar est de fournir une API permettant de s'affranchir de la gestion du formatage XML, que ce soit en lecture ou en écriture (désérialisation / sérialisation) et de ne manipuler que des objets Java dans les programmes.

Pour obtenir ce résultat avec les fichiers XML, j'ai utilisé dans un premier temps l'outil Editix[32], un éditeur XML, dont une des fonctionnalités est de produire un fichier *XML Schema Desciption* (XSD) à partir d'un fichier XML. Cela permet d'obtenir un premier niveau de fichier XSD.

Ensuite, il est possible de retravailler le document XSD, pour lui spécifier par exemple un espace de nommage, ou pour affiner les types de données manipulées qui sont déterminés par défaut par Editix comme des chaînes de caractères.

Pour terminer, j'utilise la classe `SchemaCompiler` de XmlBeans[33], une bibliothèque d'outils de manipulations de fichiers XML pour Java, permet de générer à partir du fichier XSL une API de manipulation de fichier XML correspondant au format spécifié.

Après l'import de ce Jar dans les programmes Java, il n'y a plus qu'à manipuler une grappe d'objet qui correspond en tout point au format qui définit le flux XML initial.

Soit le document XML suivant :

<?xml version="1.0" encoding="UTF-8"?>

```
<tests >
	<test name="test1">
		<event name="evenement1">test evenement1</event>
		<event name="evenement2">test evenement2</event>
	</test>
	<test name="test2">
		<event name="evenementTer">test evenement3</event>
	</test>
</tests>
```

[32] Editix : http://www.editix.com/
[33] XmlBeans : http://xmlbeans.apache.org/

Editix déterminera à partir de ce flux le schéma XSD suivant :

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema elementFormDefault="qualified" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
	<xsd:element name="tests">
		<xsd:complexType>
			<xsd:choice minOccurs="0" maxOccurs="unbounded">
				<xsd:element ref="test"/>
			</xsd:choice>
		</xsd:complexType>
	</xsd:element>
	<xsd:element name="test">
		<xsd:complexType mixed="true">
			<xsd:choice minOccurs="0" maxOccurs="unbounded">
				<xsd:element ref="event"/>
			</xsd:choice>
			<xsd:attribute name="name" type="xsd:string" use="required"/>
		</xsd:complexType>
	</xsd:element>
	<xsd:element name="event">
		<xsd:complexType>
			<xsd:simpleContent>
				<xsd:extension base="xsd:string">
				<xsd:attribute name="name" type="xsd:string" use="required"/>
				</xsd:extension>
			</xsd:simpleContent>
		</xsd:complexType>
	</xsd:element>
</xsd:schema>
```

La code java permettant de parcourir le fichier XML se résume alors à :

```
TestsDocument testsDocument = TestsDocument.Factory.parse(file);
Tests tests = testsDocument.getTests();
for (Test test : tests.getTestArray()) {
	for (Event event : test.getEventArray()) {
		System.out.println(event.getName());
		System.out.println(event.getStringValue());
	}
}
```

Comme on peut le constater, cette méthode simplifie grandement le parcours ou la constitution des fichiers XML en Java. Ce principe de manipulation des fichiers XML m'a beaucoup facilité la tâche tout au long de mes travaux lors de ce mémoire.

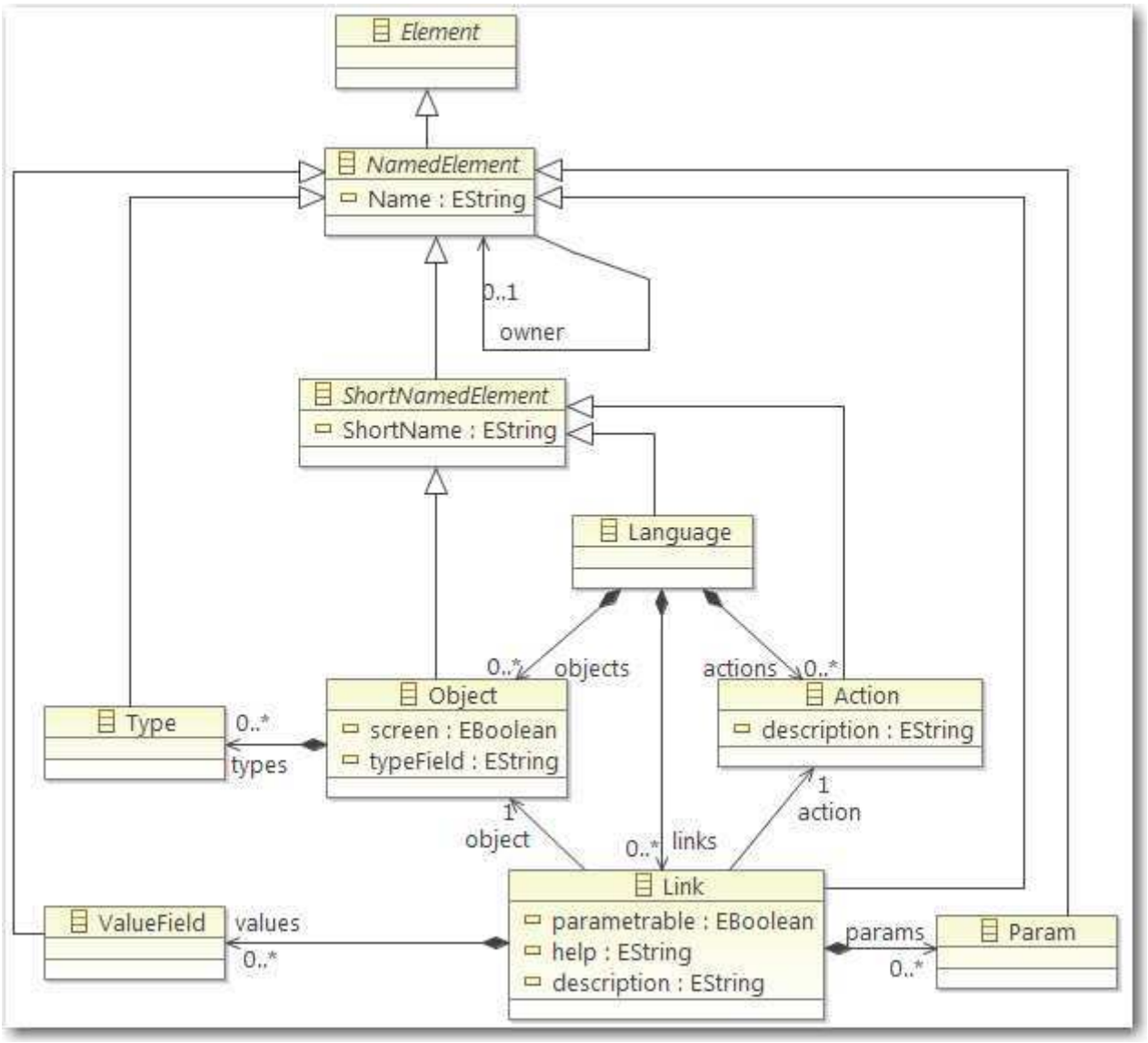


Figure 53 : métamodèle de paramétrage sYnopsis

7.6 *Du métamodèle au Jar*

Dans le cadre de notre partenariat avec la société Kalios, j'ai dû trouver une solution pour produire un ensemble cohérent de fichiers à partir d'un fichier de configuration XML ressemblant à ceci :

```
<?xml version="1.0" encoding="iso-8859-1"?>
<env_file xmlns="http://www.kalios.com/synopsis/import">
  <objecttypes>
    <objecttype ref="#TCVBBTN">
      <label>Button</label>
      <flag_typeobjet_default>0</flag_typeobjet_default>
      <type_field>WndClass</type_field>
      <types_robots>
        <type id="0">ThunderCommandButton</type>
        <type id="1">VB.VB.CommandButton</type>
      </types_robots>
      <actions>
        <action ref="#TCVBBTNCLK">
          <description>Click</description>
          <help_param>Click on the button</help_param>
          <label>Click</label>
          <parametrable>0</parametrable>
          <script>VBButton_doClick</script>
        </action>
        …
      </actions>
    </objecttype>
    …
  </objecttypes>
</env_file>
```

Etant donné le nombre conséquent de fichiers à générer, j'ai opté pour l'utilisation de l'approche MDA conjointe avec les outils MIA. Il me fallait donc un métamodèle capable de représenter le contenu de ce fichier de configuration. La création et l'alimentation du modèle correspondant au métamodèle s'est faite par le biais d'une transformation avec *MIA Transformation*. La génération quant à elle, a été laissée aux bons soins de *MIA Generation*.

Jusqu'ici, pas de problème de majeur, encore fallait-il réussir à passer du diagramme de classes représentant le métamodèle à quelque chose d'interprétable par les outils MIA. Les outils disponibles avec « Eclipse Modeling Framwork » (EMF[34]) m'ont permis de résoudre ce problème. En effet, j'ai créé un projet de type « EMF Project » vide. Ensuite j'ai ajouté un diagramme Ecore et grâce au plugin « Eclipse Modeling Tool », j'ai pu me servir du modeleur UML directement dans Eclipse (cf. Figure 53). Ecore, le métamétamodèle proposé par EMF est compatible avec le métamétamodèle de l'OMG, le MOF 2.0 (Object Management Group 2006)(Bézivin et al. 2004). Le modèle Ecore et le diagramme de classe qui le représente (fichier avec extension « ecorediag ») restent synchronisés à chaque sauvegarde de l'un d'entre eux.

[34] EMF : http://eclipse.org/modeling/emf/

L'étape suivante consiste à utiliser le générateur de modèle d'EMF. On déclare donc un nouvel « EMF Generator Model ». Il aura pour rôle de générer les classes java d'implémentation du modèle à partir du modèle Ecore. Une fois ces classes générées, il reste à exporter le projet en « Deployable plug-ins and fragments » dans le répertoire « plugin » des outils MIA.

A l'issue de cette action, *MIA Transformation* et *MIA Generation* sont à même d'ouvrir, de manipuler et de sauvegarder des modèles conformes au métamodèle définit par le diagramme de classes (cf. Figure 54).

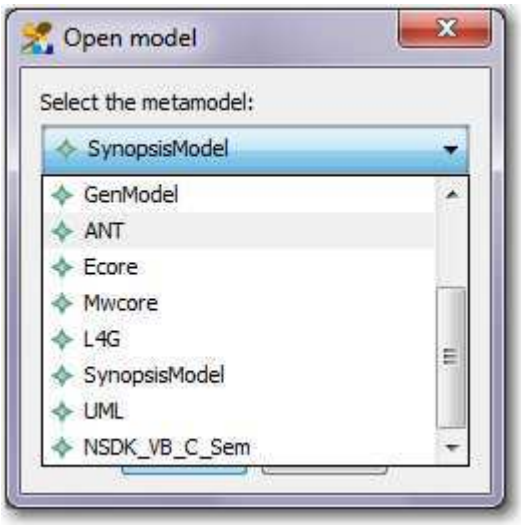


Figure 54 : sélection du métamodèle avec les outils MIA

Voici par exemple un modèle chargé dans *MIA Generation* (cf. Figure 54) correspondant au fichier de configuration XML initial.

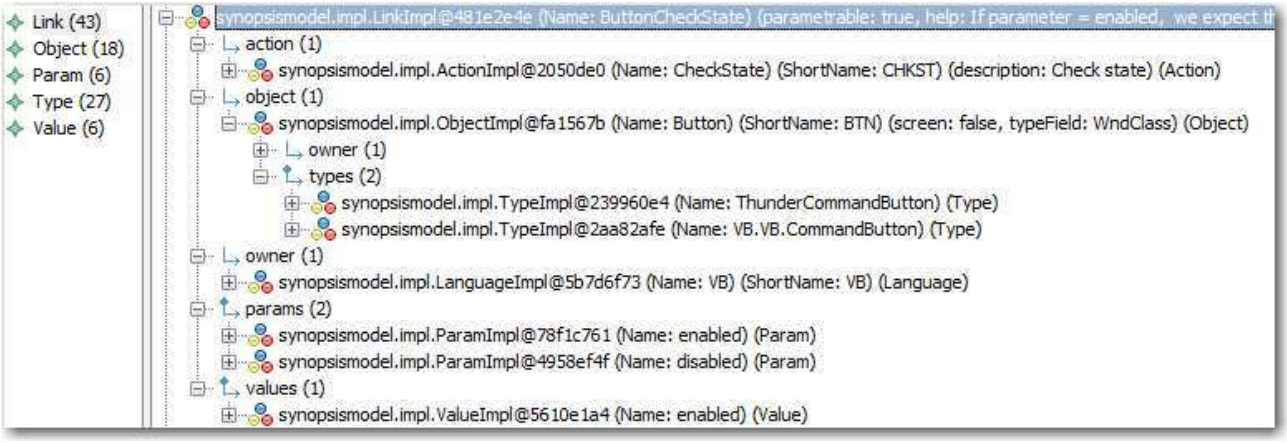


Figure 55 : exemple de modèle chargé dans *MIA Generation*

La figure qui suit illustre l'utilisation de *MIA Generation* qui permet dans le cas présent de générer du texte à chaque occurrence d'objet « Link » du modèle.

String writeScriptTechniquesSpec()

```
[[DocumentationTag(getScriptNameSpec)]]¶
//· Descrition· · :· [[description]]¶
//· Paramétrage· :· [[help]]¶
function· [[getScriptNameSpec]]·()·{¶
··addEnvironmentFunction("[[getScriptNameSpec]]");¶
··[[BeginTag]]¶
··return· [[getScriptName]]();¶
··[[EndTag]]¶
}¶
```

Generated text

```
// Start of documentation : Spec_VB_doButtonCheckState.
// End of documentation : Spec_VB_doButtonCheckState.
// Descrition  : Check button state (enabled/disabled)
// Paramétrage : If parameter = enabled,  we expect the
function Spec_VB_doButtonCheckState() {
  addEnvironmentFunction("Spec_VB_doButtonCheckState");
  // Start of user code : Spec_VB_doButtonCheckState.
  return VB_doButtonCheckState();
  // End of user code : Spec_VB_doButtonCheckState.
}
```

Figure 56 : exemple de génération dans *MIA Generation*

Stratégie de test au sein du processus d'évolution d'architecture de Sodifrance

---

RESUME

Mots clés : évolution d'architecture, ingénierie dirigée par les modèles, transformation, génération, test dirigé par les modèles

Ce mémoire a pour objectif de répondre à deux attentes essentielles non couvertes actuellement par le processus d'évolution d'architecture de Sodifrance. En premier lieu, la cartographie des tests, qui à partir d'une cartographie d'application, permet d'indiquer l'ensemble des composants impliqués dans l'exécution d'un cas de test. Et dans un second temps, l'automatisation des tests, c'est-à-dire, disposer d'un processus permettant de générer les informations nécessaires aux outils de rejeu de test du marché. Ce processus consiste, lors du passage des tests de références sur l'application source, à alimenter la cartographie de test. Cela permet au final d'initialiser les outils de rejeu de test avec les informations relatives à l'application cible issue de la migration.

SUMMARY

Keywords: architecture evolution, model driven engineering, transformation, generation, model driven testing

This paper aims to answer two essential needs not currently covered by the Sodifrance's architecture evolution process. Firstly, test mapping, which, by drawing on from an application cartography, will indicate which components are involved in the execution of a test case. Secondly, test automation, that is to say, have a process for generating the information required typically by any of the test replay tools available on the market. This process consists of loading the test mapping during reference test constitution phase on the source application. Then initialize the test replay tool with that information about the target application derived from the migration.