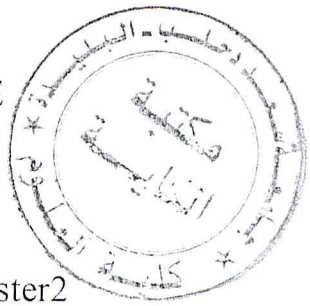
REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE
UNIVERSITE SAAD DAHLEB DE BLIDA

FACULTE DES SCIENCES
DEPARTEMENT D'INFORMATIQUE



Mémoire

Présenté pour l'obtention du Diplôme de Master2
En Informatique
Spécialité : Ingénierie de logiciel

Par : HENNI MANSOUR Imène

Sujet :

CONCEPTION ET IMPLEMENTATION D'UN SYSTEME DE TRANSFORMATION DE MODELE BASE SUR XSLT

Soutenu Publiquement le : 09 Septembre 2013

Président de Jury : M^R CHERIF ZAHAR . A (Maitre assistant A , USDB)
Encadreur : M^me GUEMRAOUI. L (Attachée de Recherche CERIST)
Promoteur : D^r BOUSTIA .N (Maitre de conférences USDB)
Examinateur 1: M^R SIDOUMOU . R (Maitre assistant A, USDB)
Examinateur 2 : M^R HAMMOUDA .M (Maitre assistant A , USDB)

Résumé

L'ingénierie dirigée par les modèles (IDM) a permis plusieurs améliorations significatives dans le développement de systèmes complexes en permettant de se concentrer sur une préoccupation plus abstraite que la programmation classique.
Ce travail interne à ce domaine adresse une importance particulière aux technique de transformation des modèles, plus particulièrement à la transformation basée sur XSLT.

Il consiste à élaborer un système de transformation écrit en Mtrans pour générer l'équitante du code en XSLT.
Pour développer ce système, nous avons utilisé différentes technologies informatique déjà existantes, qui nous ont aidées pour cette fin.

Mots-clés : ingénierie dirigée par les modèle, transformation de modèles, XSLT, Mtrans, système de transformation.

Abstract

The model-driven engineering (MDE) has several significant improvements in the development of complex systems in order to focus on a more abstract than conventional programming concern.

This internal work in this area Handles special importance to the technical processing models particularly the transformation based on XSLT.

It is to develop a processing system written in Mtrans equitante to generate code in XSLT.

To develop this system, we used various existing computer technologies, which helped us in our work.

Keywords : model-driven engineering, model transformation, XSLT, Mtrans, system transformation.

Remerciements

Merci Allah (mon dieu) de m'avoir donné la capacité d'écrire et de réfléchir, la force d'y croire, la patience d'aller jusqu'au bout du rêve et le bonheur de lever mes mains vers le ciel et de dire " Ya Kayoum "

J'adresse mes plus vifs remerciements à Madame L.GUEMRAOUI pour m'avoir, tout d'abord, donné l'opportunité de réaliser ce mémoire. Je tiens à la remercier également de la confiance qu'elle m'a témoignée en acceptant de diriger ce travail, et pour avoir suivi ce travail tout en me laissant une grande liberté d'action.

Je tiens à exprimer ma gratitude à Madame N.BOUSTIA, ma promotrice et Chargée de cours à l'Université SAAD DAHLEB BLIDA , de m'avoir fait l'honneur d'accepter d'examiner ce travail.

J'adresse mes remerciements Monsieur le presidents de jury, de m'avoir fait l'honneur d'accepter la présidence du jury. Qu'il me soit permis de lui exprimer ma reconnaissance.

Mes remerciements les plus sincères s'adressent également au examinateurs, pour avoir accepté d'examiner ce travail malgré leurs nombreuses occupations.

Je remercie plus particulièrement mes amis de l'USDB et specialement du CSC-Club qui, grâce à leur soutien et à leur bonne humeur, m'ont permis de passer cinq belles années au sein de l'Université.

J'adresse mes remerciements les plus sincères à mes enseignants de la faculté d'informatique de l'USDB qui m'ont guidé et apporté leur aide en dehors des heurs de cours.

Enfin, Je remercie de tout cœur mes parents pour la confiance, le soutien et l'aide qu'ils m'ont apportés durant toutes mes études ce qui m'a permis d'arriver jusque là.

Dédicaces

A celle qui m'a donné la vie, le symbole de tendresse, qui s'est sacrifiée pour mon bonheur et ma réussite, à ma mère .

A mon père, école de mon enfance, qui a été mon ombre durant toutes les années des études, et qui a veillé tout au long de ma vie à m'encourager, à me donner l'aide et à me protéger.
Que dieu les garde et les protège.

A mes deux adorables sœurs.

A mes amis, a la famille WEBDAYS.

A tout les membres du CSCClub

A tous ceux qui me sont chèrs.

A tous ceux qui m'aiment.

A tous ceux que j'aime.

Je dédie ce travail.

Table des matières

Table des figures

Liste des tableaux

Liste des abréviations

Abreviation	Description
API	Application Programming Interface
ATL	Atlas Transformation Language.
BNF	Backus Naur Form
HTML	Hyper Text Markup Language
IDE	Integrated Development Environment
IDM	Ingénierie Dirigée par les Modèles.
IEEE	Institute of Electrical and Electronics Engineers.
IHM	Interaction Homme Machine
MDD	Model Driven Development
MDE	Model Driven Engineering
MVC	Model View Controller
OMG	Object Management Group.
UML	Unified Modeling Language.
W3C	World Wide Web Consortium
XML	EXtensible Markup Language.
XSL	EXtensible Stylesheet Language.
XSLT	EXtensible Stylesheet Language Transformations.

Introduction générale

INTRODUCTION GENERALE

De nos jours, les entreprises opèrent dans un milieu en perpétuelle évolution. La mondialisation du marché et la concurrence farouche, les changements fréquents de techniques et de technologies, les clients de plus en plus puissants, les réglementations gouvernementales, et les responsabilités éthiques sont quelques exemples des pressions que subit l'entreprise actuelle.

Pour faire face à tous ces facteurs complexes, la modélisation des entreprises est devenue une préoccupation primordiale depuis le milieu du $20^{ième}$ siècle. De nouveaux paradigmes et de nouvelles formes d'organisations sont devenus indisponsables pour essayer de répondre aux exigences des entreprises imposées par les changements continus de l'environnement. C'est pour cette raison que la communauté informatique a constaté la nécessité de réfléchir à une approche qui réduira cette complexité et permettra une meilleure exploitation de ces plates-formes.

C'est ainsi que l'ingénierie dirigée par les modèles (IDM) a fait son apparition. Ce type d'ingénierie a apporté à ces technologies un nouveau niveau d'abstraction qui a permis la capitalisation des définitions de leurs applications et les a rendues plus accessibles.

Depuis la fin des années 80, et partant de l'approche d'ingénierie dirigée par les modèles, plusieurs langages de modélisation commerciaux et open sources ont commencé à submerger le marché du logiciel. Le plus connu d'entre eux était certainement le langage UML, qui jusqu'à aujourd'hui reste une référence en matière d'approche par modèle.

Cette submersion non contrôlée et surtout non standardisée a entrainé des difficultés perçues lors de la transmutation des modèles entre ces divers langages, d'où la nécessité de développer des processus de transformation adéquats. La nécessité d'un tel processus est perçue également lors d'une éventuelle abstraction d'entité ou bien le rajout d'une autre, lors d'une tentative de simplification ou la tentative de détailler un modèle donné. Il en est déduit que les processus de transformations présentent une des solutions appropriées pour régler les problèmes de transmutation entre les modèles. Toutefois, la rédaction des transformations est un travail complexe puisqu'elle doit répondre à des exigences diverses et hétérogènes et s'exprime au travers des langages comme : ATL, Kermeta ou XSLT .

Dans ce travail, on s'intéresse au langage XSLT, une recommandation du W3C pour la transformation de données XML est qui se trouve dérivé de ce dernier. Le langage XSLT offre une grande souplesse de transformation, Cependant il reste confronté à certaines failles, principalement sa syntaxe très verbeuse et peu lisible, Pour pallier ce problème, des solutions ont été proposées visant l'utilisation d'un langage abstrait, adapté à la transformation qui se base essentiellement sur la dissociation entre la syntaxe abstraite et concrète. L'objectif de se projet est de mener une étude afin de concevoir et d'implémenter un système de transformation de modèle, qui permet d'exprimer naturellement les règles de transformations et qui exploite XSLT non pas comme langage de spécification des transformations, mais comme environnement d'exécution des transformations.

ORGANISATION DU MEMOIRE

Le présent manuscrit est subdivisé en quatre (04) chapitres et quatre annexes.
Nous précisons ici l'objectif de chacun d'entre eux, un bref descriptif de leurs contenus et les points qui nous paraissent essentiels.

- ***Chapitre*1** : ÉTAT DE L'ART
- Partie 1 : L'ingénierie dirigée par les modèles (IDM) : cette partie introduit les notions de base de l'IDM et les techniques de transformation des modèles.
- Partie 2 : XSLT : cette partie introduit le langage xslt, son fonctionnement

ainsi que ses objectifs.

- Partie 3 : MTRANS : cette partie introduit Des notions sur Mtrans, allant de sa syntaxe jusqu'à sa correspondance avec XSLT .

- ***Chapitre2*** : CONCEPTION.

Ce chapitre englobe tout ce qui concerne la conception du système en incluant l'expression des besoins sous forme de diagramme de cas d'utilisation et une conception plus détaillée sous forme de diagrammes de classe et de séquence.

- ***Chapitre3*** : IMPLÉMENTATION.

Ce chapitre englobe la partie de la mise en œuvre du système ; il aborde la description de l'outil ainsi que le développement de l'application.

- ***Chapitre4*** : VALIDATION

Pour bien comprendre le processus d'implémentation, ce chapitre expose un exemple de mise en œuvre, extrait de la transformation d'un modèle objet vers un modèle relationnel.

- ***CONCLUSION GÉNÉRALE***

Enfin, une conclusion vient clôturer ce mémoire.

- ***ANNEXES***

Cette partie aborde des descriptions plus détaillées sur quelques notions rencontrées dans les différents chapitres du mémoire.

Chapitre 1

Etat de l'art

1.1 INTRODUCTION

Dans ce chapitre, nous allons tirer profit des notions existantes. L'ensemble des concepts et méthodes utilisés dans notre travail y sont présentés. Il comporte trois parties :

La première section aborde la définition ainsi que les concepts fondamentaux de l'ingénierie dirigée par les modèles. Elle aborde aussi le principe de transformation des modèles et ses diverses catégories selon une approche structurelle. Il se termine par une description du processus de transformation.

La deuxième section aborde le langage XSLT, elle commence par décrire les concepts fondamentaux de ce langage, allant de sa définition à une analyse de ses avantages et ses inconvénients.

Enfin la troisième section aborde le langage MTRAN, elle donne une description de ce langage, sa syntaxe, et le mapping possible entre XSLT et MTRAN.

1.2 L'INGENIERIE DIRIGEE PAR LES MODELES (IDM)

1.2.1 Présentation générale de l'IDM

L'ingénierie dirigée par les modèles (IDM), ou Model Driven Engineering (MDE) en anglais, ou aussi (MDD) Model-Driven Development est la discipline qui place les modèles au centre des processus d'ingénierie logicielle. Avec cette approche, le code final exécutable n'est plus considéré comme l'élément central dans le processus de développement mais comme un élément – naturellement important – qui résulte d'une transformation de modèles [SAM,10].

En pratique, cela nécessite de formaliser les modèles pour les rendre exploitables par la machine et de réaliser des programmes permettant de traiter des modèles. Dans la terminologie de l'IDM, ces programmes sont regroupés sous le terme de transformations de modèles. Les deux originalités de l'IDM sont donc d'une part des modèles plus formels,

et d'autre part des programmes de transformations de modèles[FRA,06], cette dernière sera exposée plus loin au cours de ce chapitre.

1.2.1.1 Modèle

Dans le domaine du génie logiciel, un modèle est une abstraction d'un système construite dans un but précis. On dit alors que le modèle représente le système. Un modèle est une abstraction dans la mesure où il contient un ensemble restreint d'informations sur un système. Il est construit dans un but précis dans la mesure où les informations qu'il contient sont choisies pour être pertinentes vis-à-vis de l'utilisation qui sera faite du modèle. A titre d'exemple, une carte routière est un modèle d'une zone géographique conçu pour circuler en voiture dans cette zone. Pour être utilisable, une carte routière ne doit inclure qu'une information très synthétique sur la zone cartographiée : une carte à l'échelle 1, bien que très précise, ne serait d'aucune utilité [FRA,06].

1.2.1.2 Méta-Modèle

Afin de rendre un modèle utilisable il est nécessaire de préciser le langage de modélisation dans lequel il est exprimé. On utilise pour cela un $méta - modèle$. Ce méta-modèle représente les concepts du langage de méta-modélisation utilisé et la sémantique qui leur est associée. En d'autres termes, le méta-modèle décrit le lien existant entre un modèle et le système qu'il représente. Dans le cas d'une carte routière, le méta-modèle utilisé est donné par la légende qui précise, entre autres, l'échelle de la carte et la signification des différents symboles utilisés. On dit qu'un modèle est conforme à un méta-modèle si l'ensemble des éléments du modèle sont définis par le méta-modèle. Cette notion de conformité est essentielle à l'ingénierie des modèles mais n'est pas nouvelle : un texte est conforme à un orthographe et une grammaire, un programme JAVA [1] est conforme au langage JAVA et document XML [2] [XML] est conforme à sa DTD [3] [DTD][FRA,06].

1.2.1.3 Meta-Meta-Modèle

De la même manière qu'il est nécessaire d'avoir un méta-modèle pour interpréter

1. www.java.com
2. www.w3.org/XML
3. www.w3.org/TR/html4/sgml/dtd.html

un modèle, pour pouvoir interpréter un méta-modèle il faut disposer d'une description du langage dans lequel il est écrit : un méta-modèle pour les méta-modèles. ce méta-modèle particulier est désigné par le terme de méta-méta modèle.En pratique, un méta-méta-modèle détermine le paradigme utilisé dans les Méta-modèles et que les modèles qui sont construits à partir de lui. De ce fait, le choix d'un méta-méta-modèle est un choix important et l'utilisation de différent méta-méta modèle conduit à des approches très différentes du point de vue théorique et du point de vue technique[FRA,06].

1.2.1.4 Synthèse

Pour résumer, la figure 1.1 représente les différentes relations qui existent entre les modèles, méta-modèles et méta-méta-modèles. On distingue sur cette figure les trois niveaux de modélisation M1, M2 et M3 correspondant respectivement au modèle, méta-modèle et méta-méta-modèle.

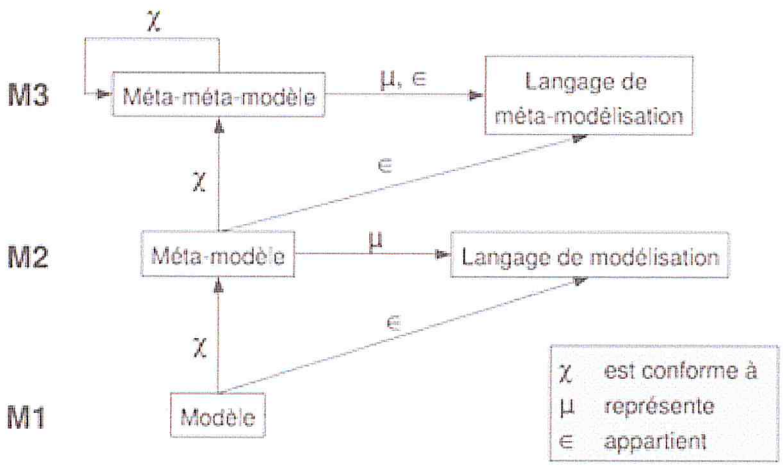


FIGURE 1.1: Niveaux de modélisation

1.2.2 Transformation de modèles

L'incompatibilité fréquente des modèles et leurs langages de modélisation et la nécessité d'interaction entre eux, ont engendré un problème au niveau de la transformation. C'est pour cela qu'un processus de transformation fiable était nécessaire. Cette

partie du chapitre aborde ce côté de l'IDM qui a fait l'objet d'une panoplie de travaux. La notion de règle de transformation sera détaillée par la suite avant de conclure par une description du processus de transformation.

1.2.2.1 Principaux Objectifs de la transformation

- ***L'optimisation*** : la transformation a pour but d'améliorer par exemple la qualité d'exécution du modèle, tout en préservant la sémantique du modèle.
- ***La simplification*** : la transformation va permettre de simplifier la complexité de la syntaxe du modèle.
- ***La migration*** : la transformation d'un modèle écrit dans un certain langage en un modèle écrit dans un autre langage, tout en gardant le même niveau d'abstraction.
- ***La synthèse*** : la transformation d'un modèle de haut niveau, très abstrait (spécification, modèle fonctionnel) vers un modèle de bas niveau plus concret (code généré, exécutable).
- ***La rétro – ingénierie*** : il s'agit d'une transformation de synthèse inverse qui permet d'extraire des spécifications de haut niveau d'un modèle de bas niveau.

1.2.2.2 Classification des méthodes de transformation

Les méthodes de transformation de modèles ont été classées selon la structure utilisée pour représenter le modèle, en voici trois grandes générations [BEZ,04] :

- ***Génération1 : Transformation de structures séquentielles d'enregistrement.***

Dans ce cas un script spécifie de manière déclarative comment un fichier d'entrée se réécrit en un fichier de sortie. Un fichier est composé d'enregistrements ou lignes. Un enregistrement est composé de champs, une ligne d'entrée peut produire plusieurs lignes en sortie (ex : scripts UNIX[4] ,ou PERL[5]). Ces systèmes sont plus lisibles et maintenables que d'autres systèmes de transformation, par contre ils nécessitent une analyse grammaticale du texte d'entrée et une adaptation du texte de sortie et permette

4. www.unix.org/
5. www.perl.org/

uniquement des transformations simples car ils se basent sur des syntaxes concrètes et non abstraites.

- ***Génération2 : Transformation d'arbres.***

Ces méthodes permettent le parcours d'un arbre d'entrée au cours duquel sont générés les fragments de l'arbre de sortie. Ces méthodes se basent généralement sur des documents au format XML et l'utilisation de XSLT [XSLT] ou XQuery [6] .

- ***Génération3 : Transformation de graphes.***

Ces méthodes utilisent des structures à forme graphique (orientée et étiquetée) appelées modèle, le modèle d'entrée est transformé en un modèle de sortie, le moteur de transformation utilisent dans ce cas un modèle, appelé modèle de transformation, tous ces modèles doivent obligatoirement être conformes aux méta-modèles qui les produisent.

1.2.2.3 Processus de transformation de modèle

Le processus de transformation de modèle passe par trois étapes essentielles, les deux dernières définissent un système de transformation [THO,08].

Définition des règles de transformation

Étant données un modèle exprimé dans un langage de modélisation L1, et un autre modèle exprimé dans un langage de modélisation L2, il s'agit dans cette étape d'élaborer une mise en correspondance des concepts de L1 et L2, en ayant recours au méta-modèle des deux langages, afin de mettre en place une base de règles exhaustive et générique. De cela un méta modèle dit méta-modèle de transformation sera généré, il servira par la suite comme générateur au modèle de transformation.

Les modèles instances de ce méta modèle sont des spécifications de transformation entre deux métamodèle spécifique, de cette manière, la spécification de transformation devient lisible (comprendre un modèle est plus facile que de comprendre un code), et

6. www.w3.org/XML/Query/

réutilisable (le métamodèle représente les règles de transformation de manière abstraite indépendante du langage de sa mise en œuvre).

Dans cette étape la transformation est toujours dans son niveau le plus abstrait, afin qu'elle soit matérialisée, les règles générées doivent être exprimées dans une plate-forme d'exécution.

Expression des règles de transformation

Dans cette étape, le modèle de transformation généré dans la première phase devra être exprimé dans un langage de spécification de règles.

Un langage de transformation peut-être déclaratif, impératif, ou hybride.

Dans la programmation déclarative, on décrit d'une part les données du problème à traiter et d'autre part les contraints sur ces données. Le programme s'exécute à partir de la situation courante décrite par les données tout en respectant les contraintes. Le langage déclaratif décrit ce qu'on devrait avoir à l'issue d'un certain nombre de données initiales. XSLT est un exemple de langage déclaratif de transformation.

Par opposition, un programme impératif décrit comment le résultat devrait être obtenu en imposant une suite d'action que la machine doit effectuer.

Quant au langage hybride, il regroupe à la fois les paradigmes de programmation déclarative et impérative : l'ordre d'exécution des modules doit être spécifié tandis qu'au sein d'un même module, la détermination de l'ordre d'exécution des règles n'est pas à la charge de l'utilisateur.

Exécutions des règles de transformation

Une fois les règles spécifiées exprimées dans un langage de transformation, elles doivent être intégrées dans un moteur d'exécution, Comme exprimé dans la figure 1.2, le moteur d'exécution prend en entrée le modèle et le méta-modèle source, ainsi que le modèle (règles) et le méta modèle du langage de transformation, il prend en plus comme entrée le méta-modèle du modèle cible à générer. Enfin, il utilise ces éléments pour établir la correspondance entre le modèle d'entrée et celui de sortie.

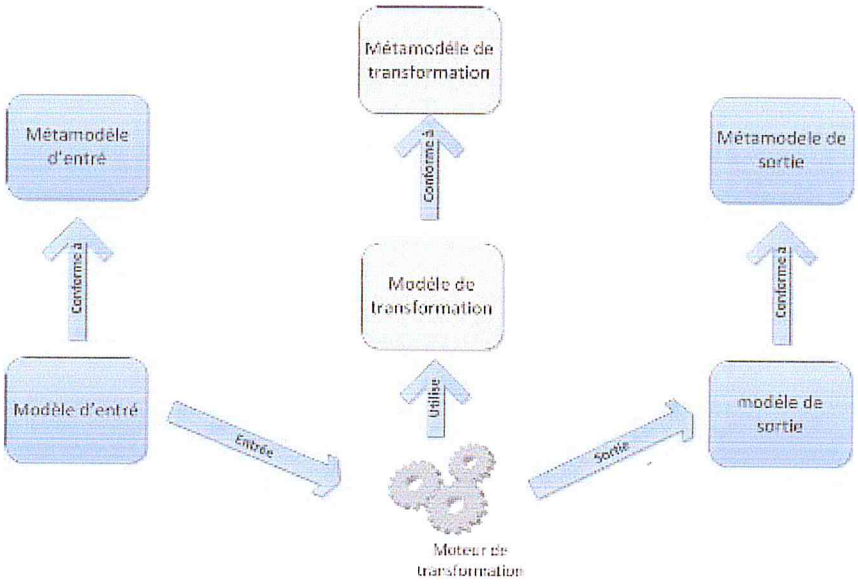


FIGURE 1.2: Exécution des règles de transformation

1.2.2.4 Synthèse

Nous avons introduit dans cette section les principes généraux de l'IDM, c'est-à-dire la méta modélisation d'une part et la transformation de modèle d'autre part. Ces deux axes constituent les deux problématiques clé de l'IDM sur lesquelles la plupart des travaux de recherche se concentrent actuellement. Dans la partie suivante, nous détaillerons d'avantage XSLT un langage qui appartient à la $2^{éme}$ génération " transformation d'arbre", et sur le quel se basera notre application.

1.3 XSLT

1.3.1 Introduction

XSLT (Extensible Stylesheet Langage Transformations) dérivant du XSL [7] représente un moyen simple et standard d'échanger des données de texte structurées entre différents programmes informatique ou pour transformer un document XML en un autre format basé sur du texte, une partie de son succès provient du fait qu'il est lisible et modifiable par l'homme, et tout ça simplement par un éditeur.

7. www.w3.org/Style/XSL

XSLT possède de nombreuses fonctions de traitement qui en font un langage de programmation complet. On peut créer des "fonctions", des boucles, calculer un maximum, faire des recherches dans un document XML, compter le nombre de résultats, etc

Dans notre travail nous allons nous baser sur la transformation de modèle.

Les caractéristiques principales d'une transformation XSLT sont :

• C'est un langage déclaratif et non procédural. Ce qui revient à dire qu'à la différence d'un langage de programmation classique, il ne spécifie pas le « comment ? « (les algorithmes) : il se contente de déclarer le « quoi ? ». [KAY, 05]

• Il est lui-même écrit en XML. Ce qui veut dire qu'il pourra être à son tour transformé par une nouvelle feuille de style XSLT, et ainsi de suite, à l'infini, Ou bien encore qu'il pourra être manipulé à l'aide de tous langage de programmation qu'on voudra, pourvu que ce langage implémente l'interface « Document Object Model (DOM) ». [KAY, 05]

• Les fichiers cibles peuvent être produit en quatre langages : soit HTML, soit XML, soit XHTML , soit texte.

1.3.2 Syntaxe

– *Syntaxe abstraite*

Les feuilles de styles XSLT sont constituées par un ensemble de règles de transformation (figure 1.3). Une règle de transformations se compose d'un pattern et d'un replacement. Le pattern est formé par des expressions XPath [8] qui indiquent sur quelles parties du document XML source doit s'appliquer la transformation. La partie replacement est composée d'appels à des fonctions de XSLT permettant de produire le fichier cible.

8. www.w3.org/TR/xpath

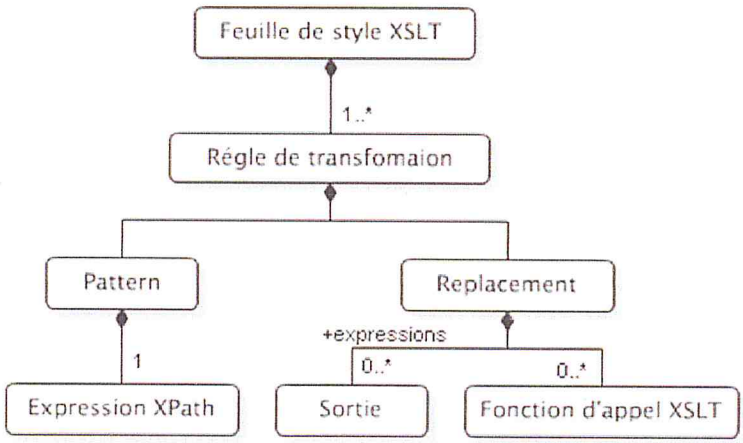


FIGURE 1.3: Structure d'une règle XSLT

– ***Syntaxe concrète***

D'un point de vue concret, un feuille de style commence et se termine par une paire de balise $< xsl : transform >$et$< /xsl : transform >$, ou bien $< xsl : stylesheet >$ et $< /xsl : stylesheet >$ Chaque règle correspond à un fragment $< xsl : template >$et $< /xsl : template >$ (figure 1.4).

L'attribut match contient une expression XPath permettant la localisation des nœuds. Cet attribut va définir les parts du document XML source sur lesquels s'applique la transformation. Des instructions d'exécution de transformation et les résultats output sont placés en mixte entre les deux balises $< xsl : template >$et $< /xsl : template >$.

```
1   <?xml version = "1.0"  encoding="ISO-8859-1"?>
2     <xsl:stylesheet>
3           <xsl:template match="noeud1 ">
4            traitements sous forme de balises
5           </xsl:template>
6   .....
7           <xsl:template match="noeud2 ">
8           traitements sous forme de balises
9           </xsl:template>
10          ...
11  </xsl:stylesheet>
```

Règle 1

Règle 2

FIGURE 1.4: Exemple de feuille de style XSLT

1.3.3 Processus de transformation XSLT

Le principe de fonctionnent du processus de transformation XSLT est decrit dans la figure 1.5

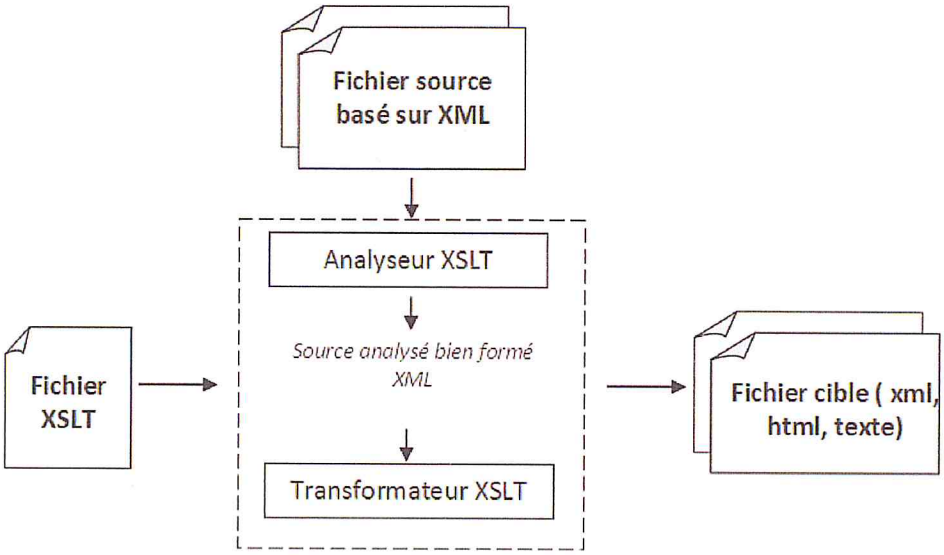


FIGURE 1.5: Vision conceptuelle du processus de transformation XSLT

L'entrée de l'analyseur correspond à un ou plusieurs fichiers XML et un fichier XSLT. L'exécution d'une transformation se réalise en deux (02) étapes principales :

- Analyse des fichiers sources et des fichiers de transformation XSLT : Dans cette étape XSLT vérifie si les fichiers sont bien formés ou non, et charge la feuille de style XSLT.

- Application des règles de transformation : Les règles de transformation sont ensuite appliquées en fonction du contenu du fichier source. Le parseur parcourt le document XML initial du début à la fin sous forme d'arbre, des éléments parents vers les éléments enfants sauf si la feuille de style change l'ordre de parcours.

1.3.4 Processeurs XSLT

Le rôle principale de processeur XSLT est d'appliquer une feuille de style XSLT à un document source XML et de produire un document résultat. Dans la mesure où chacun d'eux représentent une application du XML, leurs structure fondamentale est un arbre. Le processeur XSLT manipule donc en réalité trois arbres.

Parmis les différents processeurs XSLT les plus connu se trouve : SAXON, XALAN.

SAXON [9]est un processeur XSLT open source créer par MICHAEL KAY. Il s'agit

9. www.saxon.sourceforge.net

d'une application Java pouvant être directement exécuté à partir de l'invité, sans qu'un serveur web ou un navigateur ne soit requis. Le programme SAXON transformera le document XML en un document HTML par exemple, qui pourra être placé sur un serveur web. [KAY, 05]

XALAN [10]est un autre processeur XSLT open source proposé par l'organisation apache. Il était à l'origine, le dérivé d'un produit IBM nommé LOTUSXSL mais il mène depuis sa propre vie open source. XALAN est disponible en version java et C++. A l'instar de SAXON, XALAN-java est une application java exécutable à partir de l'invité.[KAY, 05]

1.3.5 Synthèse

Notre choix a aboutit à XSLT pour plusieurs raisons :

- Le stockage de données au format texte, ainsi on n'aura pas à manipuler du code.
- Indépendant de toutes plates-formes d'exécution.
- Standard libre du W3C, sur lequel se base plusieurs travaux qui couvrent divers domaine.
- Offre :
- Modularité. Les feuilles de style de XSLT peuvent être composées pour former des transformations complexes à partir de transformations plus simples.
- Interopérabilité. XSLT peut interagir avec d'autres langages comme Java, C# ou autres.
- Extensibilité. XSLT fournit deux mécanismes d'extensibilité. La première permet d'étendre le jeu d'instruction de XSLT lui-même, alors que le deuxième permet d'étendre l'ensemble des fonctions proposées par XPath.

Cependant le XSLT reste confronté à certaines failles, principalement :

- La syntaxe XSLT est très verbeuse et peu lisible, au point de compromettre la maintenabilité de transformation pourtant relativement simple.
- XSLT, manque une syntaxe abstraite de plus haut niveau pour la spécification de transformations.

10. www.xalan.apache.org/

```
1.  rules_transformation ::=  Entity [restriction]? to Entity
2.                              { attributes : [Attribute = attributes_transformation ,]*
3.                                roles : [Role = roles_transformation ,]* }
4.                              |  Entity
5.                              { attributes : [Attribute = attributes_transformation ,]*
6.                              roles : [Role = roles_transformation ,]* }
7.  restriction ::= Attribute operator value | Role operator value
8.  attributes_transformation ::= Attribute | Entity.Attribute |  †Literal†
9.                              | ( condition ? resultIfTrue : resultIfFalse )
10.                             |Attribute # Attribute
11. resultIfTrue ::= attributes_transformation
12. resultIfFalse ::= attributes_transformation
13. roles_transformation ::= Role | ( condition ? resultIfTrueBis : resultIfFalseBis )
14.                             | Role[Entity]
15. resultIfTrueBis ::= roles_transformation
16. resultIfFalseBis ::=roles_transformation
```

FIGURE 1.6: Syntaxe de MTRANS.

MTRANS permet également de créer de nouvelle entité. De plus, MTRANS permet d'énoncer les règles spécifiant le procédé de transformation des attributs et des rôles d'une entité. Ces règles prennent la forme d'une assignation dont la partie gauche correspond au nom de l'attribut ou du rôle transformé et la partie droite définit la valeur de cet attribut ou ce rôle dans le méta- modèle de destination.

La valeur assignée à un attribut peut être de quatre types :

- un attribut du métamodèle source,
- un attribut quelconque exact,
- une valeur conditionnelle
- une concaténation d'attributs contenus dans le métamodèle source.

La valeur assignée à un rôle peut être de trois types : un rôle du métamodèle source, une valeur conditionnelle ou un rôle du métamodèle source réduit à une entité[BEN,09].

1.4.3 Correspondance entre MTRANS et XSLT

Comme nous l'avons vu dans la section précédente, le langage XSLT peut être

Pour palier cette rédaction longue et douloureuse nous avons choisie d'étudier dans la partie suivante Mtrans : un langage dédié de spécification de transformations de modèles.

1.4 MTRANS

1.4.1 Introduction

MTRANS est un langage dédié à la transformation de modèles. Développé à un niveau d'abstraction au-dessus de XSLT, il l'utilise pour transformer des modèles, il est plus compact et facile à comprendre que XSLT.

Le langage MTRANS permet d'exprimer de manière plus commode, les règles définissant la méthode pour transformer les entités d'un modèle vers les entités d'un autre modèle. Dans ce contexte, une entité représente simplement un concept du modèle source ou cible.

1.4.2 Syntaxe

Une règle de transformations d'entité est divisée en deux parties : la première est utilisée pour spécifier l'entité source et la seconde pour l'entité destination.

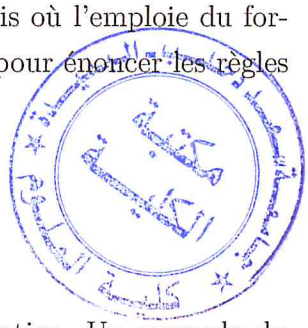
En appliquant des restrictions, la règle peut être limitée aux entités respectant une certaine condition.

Une restriction est également composée de deux parties : une partie droite qui détermine un rôle ou un attribut de l'entité et une partie gauche qui correspond à la valeur acceptée pour ce rôle ou cet attribut. La figure 1.6 représente la syntaxe Mtrans[BEN,09].

utilisé pour exprimer la transformation de modèle. Néanmoins, il semble difficile d'utiliser directement XSLT vue sa syntaxe très verbeuse.

Pour pallier à ce problème, des solutions ont été proposées, citons principalement les solutions visant l'utilisation du langage abstrait Mtrans [PEL,00] , [MIK, 01] , Ces solutions sont adaptées à la transformation qui se base essentiellement sur la dissociation entre la syntaxe abstraite et concrète.

A partir de règles abstraites basées sur les modèles, la génération automatique des règles concrètes à appliquer aux modèles est possible. Dans le cas précis où l'emploie du formalisme MTRANS pour exprimer les règles abstraites et XSLT pour énoncer les règles concrètes sont faites, voici juste un aperçu de ce mapping.



– ***Transformation d'entité*** :

Consiste à ajouter un élément dans l'arbre XML de destination. Un exemple du schéma général de la règle de transformation d'entité est illustré dans la figure 1.7.

A la fin de cette règle de transformations d'entité, des règles de correspondances des attributs et des rôles de cette même entité seront ajoutés.

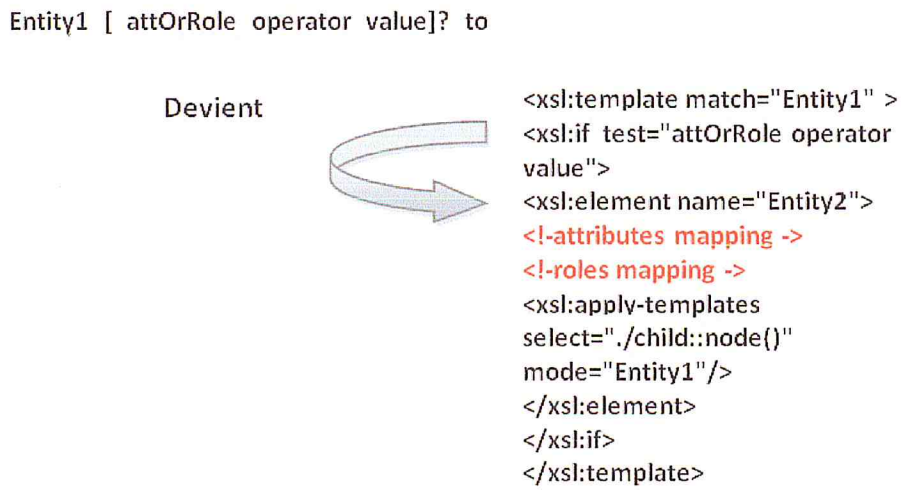


FIGURE 1.7: Transformation d'entité MTRANS vers XSLT

A la fin de la règle de transformations d'entité, on doit encore ajouter les règles de

correspondances des attributs et des rôles de cette même entité. Cette opération est réalisée grâce à l'appel de règles (xsl :apply-templates) sur les nœuds enfants.

– ***Transformation d'attribut*** :

XML peut utiliser deux façons pour représenter un attribut, il peut être un attribut propriétaire de l'entité, ou un nœud entité. Par conséquent, la transformation d'attribut tient compte de ces deux cas.

Les attributs peuvent prendre 3 type de resultats :

Attribute = value

Attribute1 = Attribute2

Attribute1 = (Attribute2 operator 'value' ? Attribute3 : Attribute4)

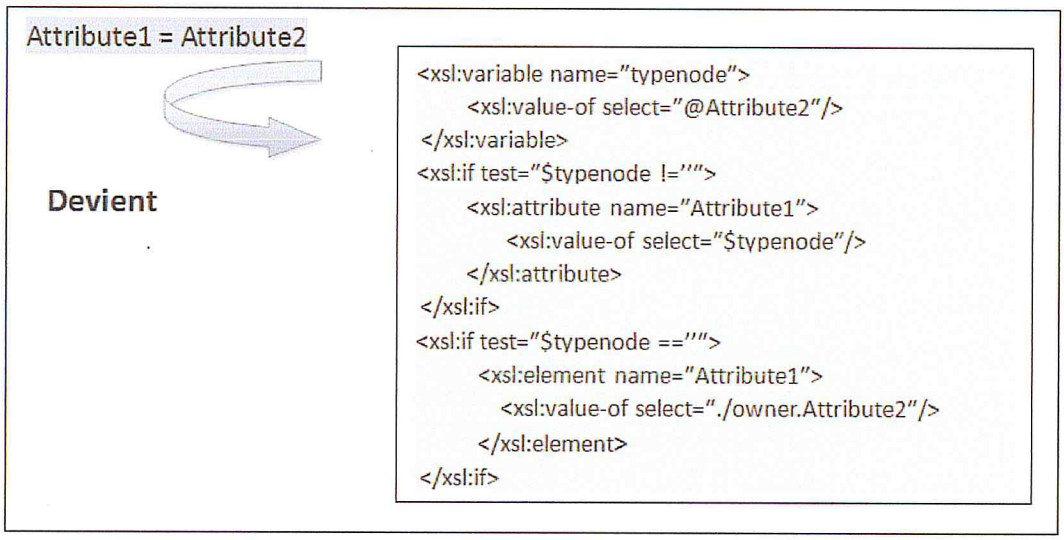


FIGURE 1.8: Exemple de mapping d'attribut.

– ***Transformation de rôles*** :

Tous comme la transformation d'attribut, le rôle possède trois types de transformation :

– Role1 = Role2

- Role1 = Role2 [Entity1]
- Role1 = (Role2 operator 'value' ? Role3 : Role4)

1.4.4 Synthèse

Le langage XSLT est adapté à transformer un document XML en un autre. Mais il est difficile et est sujet aux erreurs d'écrire manuellement des programmes XSLT. Le MTrans permet d'exprimer les règles de transformation d'une manière plus compacte et plus lisible.

L'utilisation d'un tel langage est donc nécessaire surtout que comme le montrent les correspondances entre MTRANS et XSLT, il semble aisé de générer automatiquement, à partir des règles du niveau abstrait en MTRANS, des règles XSLT plus concrètes adaptées aux modèles à transformer.

De plus, contrairement aux solutions de transformation basées sur un code exécutable rigide dans lesquelles le code devrait être remodelé pour pouvoir faire évoluer une transformation, l'architecture de transformation utilisant des feuilles de style XSLT et un processeur pour les interpréter est pleinement évolutive. En effet, pour pouvoir faire évoluer une transformation entre deux modèles, il suffira simplement de changer la feuille de style utilisée, sans pour autant changer la manière dont elle est utilisée.

1.5 CONCLUSION

Globalement, il a été présenté dans cette étude de ce premier chapitre une brève étude de ce qui fera par la suite une base informationnelle pour la conception et la mise en œuvre de notre futur application. Le deuxième chapitre de ce mémoire concerne la conception d'une application, visant la transformation de modèle basée sur XSLT en utilisant le langage Mtrans.

Chapitre 2

Conception du système

2.1 INTRODUCTION

Une estimation de la taille et la durée d'accomplissement du projet, nous a mené à choisir un cycle de développement en cascade, qui nous semblait adapté à un projet de cette dimension, ce dernier commence par une phase « d'analyse et spécification des exigences », suivie d'une phase de conception répartie sur deux sous phases « conception générale » et « conception détaillée », suivie d'une phase « d'implémentation », et enfin d'une phase « d'intégration » et de « mise en production ». Chacune de ces phases fournit un produit en sortie (document ou source) qui sera utilisé dans la phase qui la succède. La figure 2.1 montre l'architecture du cycle de développement adopté.

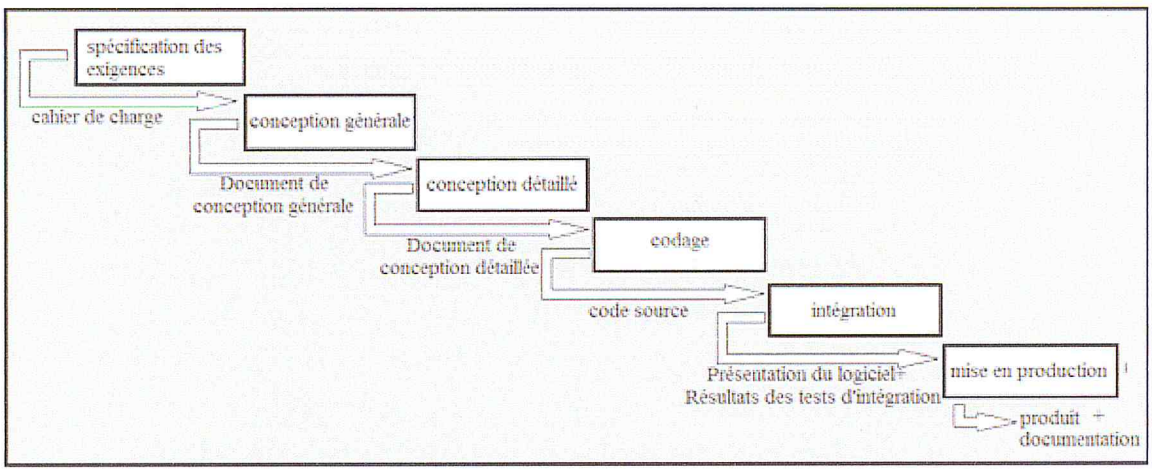


FIGURE 2.1: Cycle de vie adopté pour le développement de l'application.

Afin de mieux séparer entre les phases conceptuelles de celles de mise en œuvre, nous avons préféré répartir les phases de ce cycle sur deux chapitres. Ce premier chapitre expose les phases spécification des exigences, conception générale et détaillée.

2.2 ANALYSE ET SPECIFICATION DES EXIGENCES

2.2.1 Analyse Des Besoins

Ce projet est construit autour du thème « transformation de modèle », cette

technique qui s'est imposée dans le domaine de l'ingénierie dirigée par les modèles comme un bon support d'aide au problème de passage entre modèles. Cette phase consiste à analyser puis construire un cahier de charge au tour de ce thème. Ce cahier nous sera nécessaire dans la phase de conception, ce cahier de charges est dressé au tour de Quatre besoins essentiels :

- ***Besoin1*** **:** Il s'agit de développer une interface d'édition de MTRAN qui permet à l'utilisateur de faire saisir du code Mtrans d'une manière simple et rapide.
- ***Besoin2*** **:** Créer un compilateur Mtrans qui se décompose d'un analyseur lexical et un analyseur syntaxique, ce dernier nous permet de vérifier si notre code source et conforme à la grammaire imposée par Mtrans.
- ***Besoin3*** **:** Faire une transformation « mapping » entre le code déjà validé par le compilateur et XSLT, par rapport à des règles prédéfinies.
- ***Besoins4*** **:** Afficher le code XSLT généré.

2.2.2 Specification Des Exigences Logicielles

Le choix des langages représentatifs des données devra être bien étudié, en prenant en considération non seulement la capacité de ces langages à représenter des modèles, mais aussi leurs facilités de manipulation par l'utilisateur. Pour ce qui est de l'application, elle devra satisfaire tous les besoins exprimés en prenant en considération l'aspect qualité du logiciel. Pour cela les critères de qualité présentés dans le titre suivant, doivent être satisfaits.

2.2.2.1 Qualité logiciel et norme qualitatif

Selon l'IEEE la qualité logicielle est le degré avec lequel un système, un composant ou un processus satisfait à ses exigences spécifiées .

Ces exigences sont souvent associées à des critères de qualité, vue par la communauté de l'assurance qualité comme des unités mesurables, reflétant la bonne ou mauvaise qualité d'un logiciel, pour cela de nombreuses approches de qualité ont été publiées, dans la plus célèbre est appelée approche de MacCall , Cette approche a constitué un point d'appui pour une norme qualitatif ISO [1] connu sous la référence ISO9126 . Cette

1. www.iso.org

norme fera objet de modèle de norme de qualité à satisfaire pour notre application, sans prétendre bien sûr la satisfaction de tous les critères mentionnés dans son contenu, faute de perdre notre objectif initial, mais juste satisfaire ses principaux caractéristiques donner dans la figure 2.2, une description plus détaillée de ces caractéristiques est donné dans le tableau 2.1.

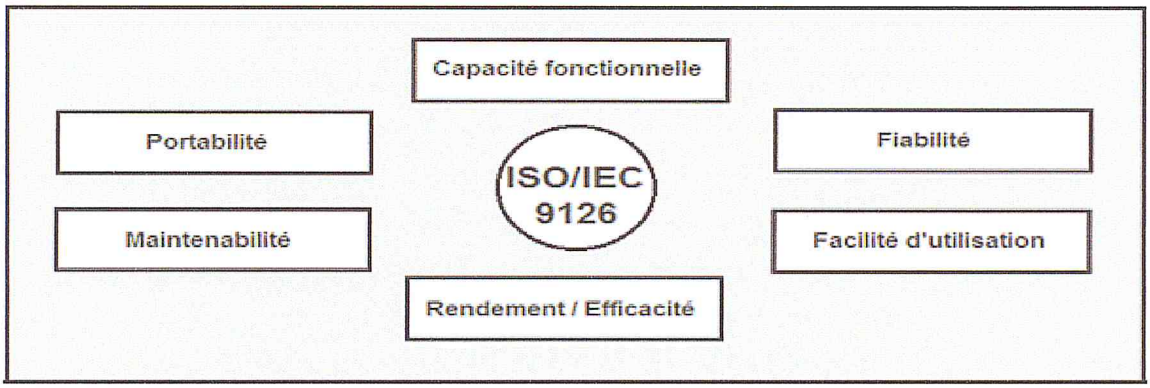


FIGURE 2.2: Les six principales caractéristiques de qualité logicielle de la norme ISO9126.

Caractéristique	Description
Capacité fonctionnelle	Capacité d'un logiciel à rependre à ses besoins fonctionnels exprimés dans le cahier de charges.
Fiabilité	Capacité d'un logiciel à maintenir son niveau de service dans des conditions précises et pendant une période déterminée.
Facilité d'utilisation	Capacité d'un logiciel à offrir un environnement qui demande peu d'effort à l'utilisateur pour exécuter ses tâches.
Rendement/ Efficacité	Capacité d'un logiciel à requérir un dimensionnement rentable et proportionné de la plate-forme d'hébergement en regard des autres exigences.
Maintenabilité	Capacité d'un logiciel à requérir peu d'effort à son évolution par rapport aux nouveaux besoins.
Portabilité	Capacité d'un logiciel à être transféré d'une plate-forme ou d'un environnement à un autre.

TABLE 2.1: Description des principales caractéristiques de qualité de la norme ISO9126.

2.2.2.2 Fonctionnalité de l'application

Pour mieux satisfaire les besoins décrits précédemment, et les critères de qualité fixés comme objectif , nous avons remarqué qu'il était préférable de répartir les diverses fonctionnalités de l'application sur des unités de traitement qu'on a mentionné « Centre de traitement », chaque centre inclut une ou plusieurs fonctionnalités offertes à l'utilisateur, essayant par leurs intermédiaires la satisfaction d'un ou plusieurs critères de qualité fixé précédemment, ces centres de traitement sont données dans le tableau 2.2.

Centre de traitement	Description	Critère de qualité ciblé
Centre d'édition	Ce centre englobe toutes les fonctionnalités d'éditions et traitements de modèles.	Capacité fonctionnelle
Centre de validation (Compilation)	Ce centre fournit des outils d'analyse et validation des divers types de données, permet de faire une validation syntaxique et lexical du code source Mtrans.	Capacité fonctionnelle
Centre de transformation	Ce centre vise à faire la transformation entre Mtrans et XSLT	Capacité fonctionnelle
Centre de sécurité	Ce centre englobe des fonctionnalités de gestion de pannes et erreur susceptibles d'être déclenché aux cours du fonctionnement de l'application, du a une intervention de l'utilisateur ou celle de la plateforme d'exécution.	fiabilité
Centre d'aide	Ce centre englobe des fonctionnalités d'aide à l'utilisateur. Elles lui offrent des informations sur le mode d'emploie de l'application et d'autre information susceptible de l'intéresser.	Facilité d'utilisation
Centre de maintenance	Ce centre dirige l'utilisateur vers des liens de documentation utiles, susceptible d'être un support d'aide pour d'éventuelles maintenances ou extensions de l'application.	Maintenabilité

TABLE 2.2: Description des fonctionnalités de l'application.

2.2.3 Specification semi-formelle des besoins

Une étude approfondie des besoins fonctionnels s'avère indispensable avant d'entamer la conception, afin d'obtenir de manière plus formelle une vue globale sur les exigences de l'application. Cette partie présente alors une modélisation de ces besoins en faisant recours aux concepts fondamentaux du langage de référence UML [2] [UML] . A savoir le diagramme de cas d'utilisation.

2.2.3.1 Diagramme de cas d'utilisations

UML ne sert ici qu'à formaliser les besoins, c'est-à-dire à les représenter sous une forme graphique suffisamment simple pour être compréhensible par toutes les personnes impliquées dans le projet. N'oublions pas que bien souvent, le maître d'ouvrage et les utilisateurs ne sont pas des informaticiens. Il leur faut donc un moyen simple d'exprimer leurs besoins. C'est précisément le rôle des diagrammes de cas d'utilisation. Ils permettent de recenser les grandes fonctionnalités d'un système [CHA, 10].

Après avoir donné un petit aperçu sur l'utilité, et ce que peut représenter un diagramme de cas d'utilisation, on vous expose dans la figure 2.3 le diagramme de cas d'utilisation de notre futur application, les principaux cas d'utilisations seront exprimés plus en détails par la suite.

2. www.uml.org

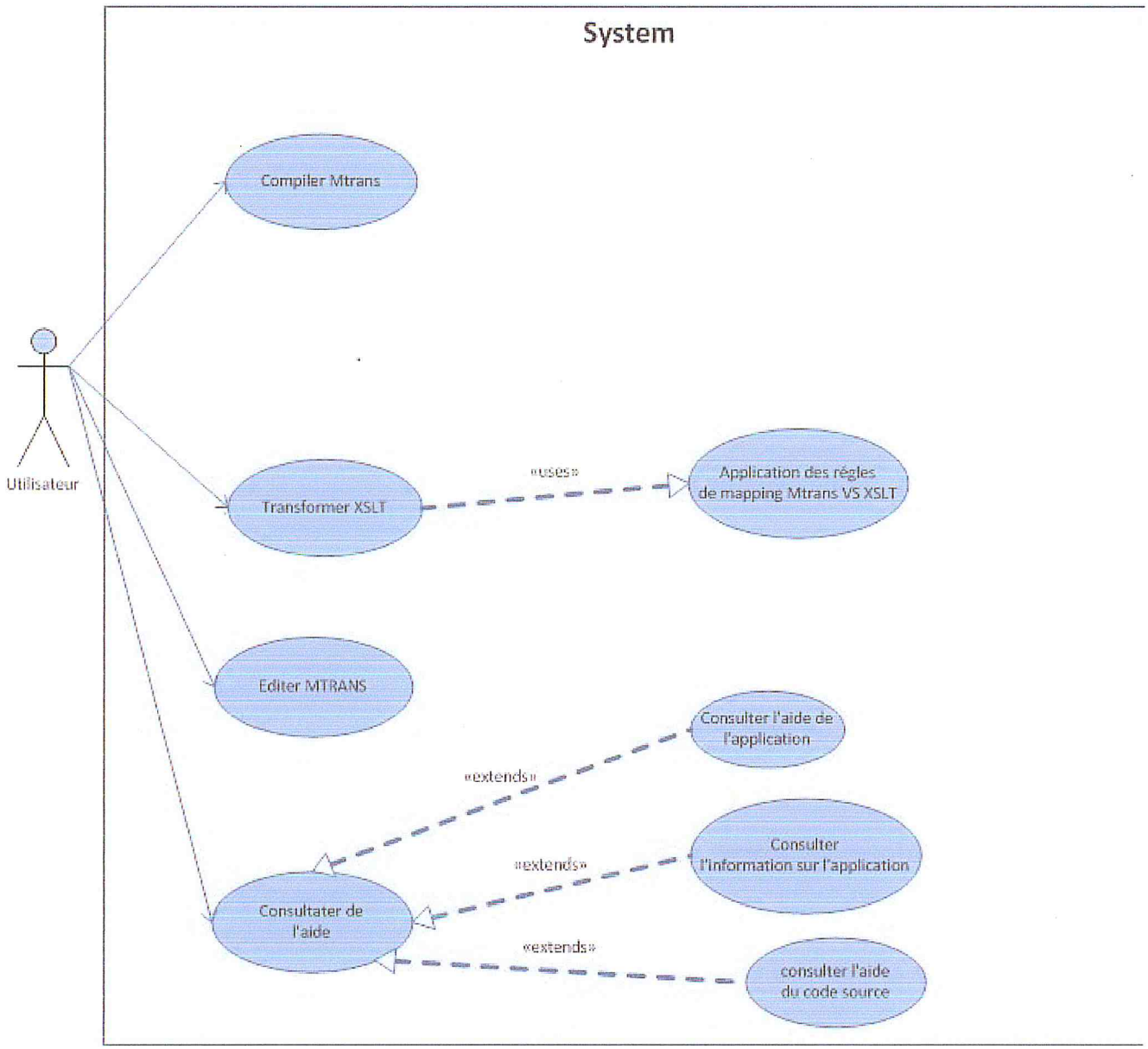


FIGURE 2.3: Diagramme de cas d'utilisation

- Le cas d'utilisation "consulter l'aide" est décrit dans le tableau 2.3

- Le cas d'utilisation " Compilation Mtrans " est décrit dans le tableau 2.4

Item	Description
Nom	Consulter l'aide
Description	Grace à ce cas d'utilisation l'utilisateur peut consulter divers options d'aide, à savoir une rubrique d'aide qui lui donnera les informations sur les fonctionnalités de l'application, ou des information utiles sur l'application elle même, des liens vers des tutoriaux sont également proposés.

TABLE 2.3: Description du cas d'utilisation "consulter l'aide"

Item	Description
Nom	Compilation Mtrans
Description	Grace à ce cas d'utilisation, l'utilisateur pourra vérifier si son code source édité en Mtrans est correcte et conforme à la grammaire Mtrans.

TABLE 2.4: Tableau2.4-Description du cas d'utilisation " Compilation Mtrans "

- Cas d'utilisation " Transformation XSLT " est décrit dans le tableau 2.5

Item	Description
Nom	Transformation XSLT
Description	Grace à ce cas d'utilisation, l'utilisateur peut transformer le code Mtrans grace aux règles de mapping implémentées.

TABLE 2.5: Description du cas d'utilisation " Transformation du code source "

- Cas d'utilisation " Affichage du code source XSLT " est décrit dans le tableau 2.6

Item	Description
Nom	Affichage du code source XSLT
Description	Grace à ce cas d'utilisation, l'utilisateur pourra consulter le code source XSLT généré après le mapping effectué précédemment.

TABLE 2.6: Description du cas d'utilisation " Affichage du code source en XSLT "

2.3 CONCEPTION ARCHITECTURALE (GENERALE)

Il s'agit dans cette phase d'apporter des solutions conceptuelles aux besoins exprimés dans le cahier de charges élaboré précédemment, pour cela nous allons épuiser les

outils offerts par UML afin d'apporter une vision modulaire de notre future application. Pour ce faire nous allons essayer de découper notre application en petites unités (objet) pour les faire assembler par la suite dans un diagramme de classe. Une telle décomposition devra faire preuve d'une forte cohésion et d'un faible couplage afin de réduire sa complexité et faciliter sa maintenance ou réutilisabilité. Pour ce fait nous allons prendre en compte l'utilisation des patrons de conception.

2.3.1 Patron de conception

– Définition Pattern

Les patrons sont des solutions éprouvées à des problèmes spécifiques et récurrents. « Un patron décrit un problème devant être résolu, une solution, et le contexte dans lequel cette solution est considérée. Il nomme une technique et décrit ses coûts et ses avantages. Il permet à une équipe d'utiliser un vocabulaire commun pour décrire leurs modèles » [JON, 97]. Ils capturent les bonnes pratiques de conception et cela à différents niveaux d'abstraction en allant des patrons architecturaux jusqu'aux patrons d'implémentation.

– Définition : Design Pattern

Les patrons de conception ont été, généralement représentés sous forme de diagrammes avec des notations objet comme UML. Ces diagrammes décrivent un ensemble de composants et leurs interactions. L'utilisation ou l'application d'un patron peut être difficile ou inexacte. En effet, les diagrammes utilisés ne représentent pas les patrons d'une façon formelle et ils n'indiquent pas comment appliquer un patron, ni quelles sont les modifications que l'on peut apporter à une instance du patron [ELB, 04].

Nous mentionnons dans ce qui suit quelque uns des patrons utilisés dans la conception de notre application.

2.3.1.1 MVC2

Vue que cette application suit le principe d'IHM [IHM] , un tel patron de conception ne pourra être que plus bénéfique, il nous apportera une séparation entre les données et

les interfaces d'un coté et les données et contrôleurs d'un autre.

2.3.1.2 Composite

Nous allons utilisée ce patron de conception pour mieux structurer le système de données (Modèle de l'MVC).

2.3.1.3 Stratégie

Utilisée pour permettre le choix d'une stratégie ou une autre, il nous sera bénéfique pour la gestion des contrôleurs du MVC.

- D'autre patterns ont été également utilisés et seront mentionné dans le diagramme de classe.

2.3.2 Diagramme de classes

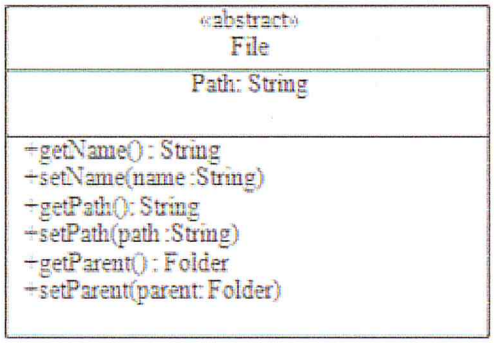
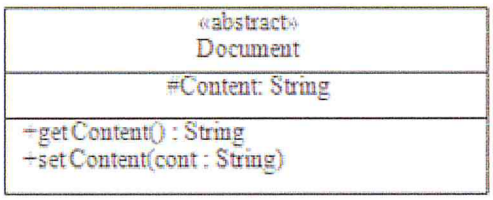
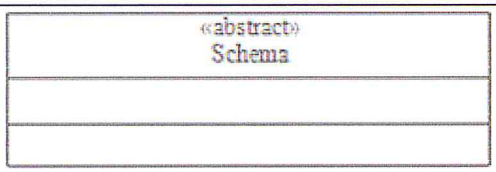
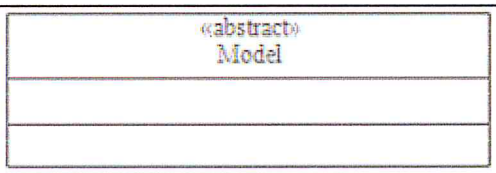
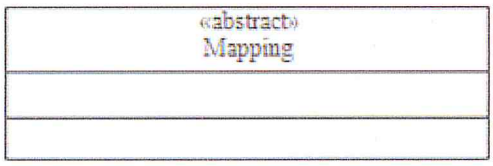
Le diagramme de classes est considéré comme le plus important de la modélisation orientée objet, il est le seul obligatoire lors d'une telle modélisation.

Alors que le diagramme de cas d'utilisation montre un système du point de vue des acteurs, le diagramme de classes en montre la structure interne. Il permet de fournir une représentation abstraite des objets du système qui vont interagir ensemble pour réaliser les cas d'utilisation.

Il s'agit d'une vue statique car on ne tient pas compte du facteur temporel dans le comportement du système. Le diagramme de classes modélise les conceps du domaine d'application ainsi que les concepts internes créés de toutes pièces dans le cadre de l'implémentation d'une application [LAU,09].

2.3.2.1 Description des classes

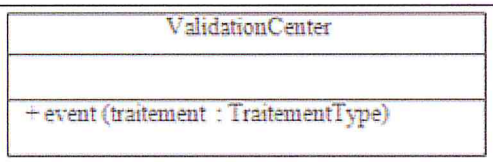
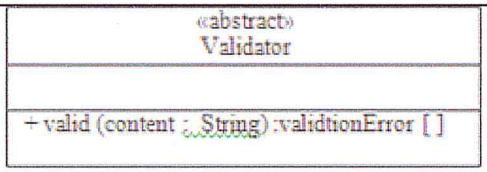
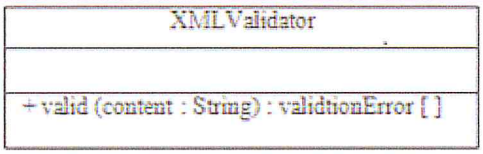
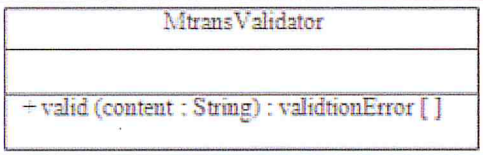
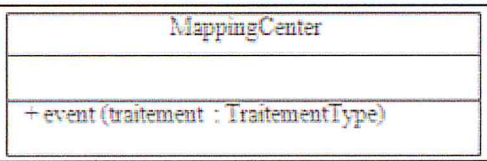
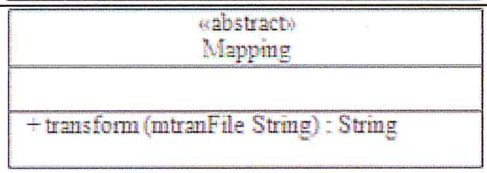
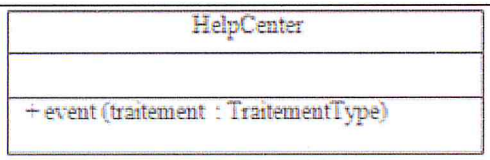
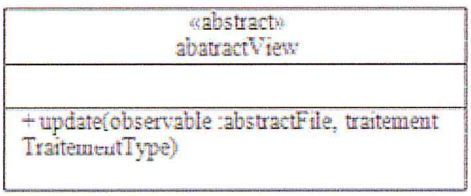
Avant d'exposer notre diagramme de classe nous allons décrire dans cette partie les classes objets constituant notre application.

	- Classe abstraite. - Mère de tout modèle concret dans le MVC. - Représente un fichier de données. - Contiens des informations de localisation du fichier dans le système de fichier de la plateforme d'exécution. - Contiens un pointeur sur le répertoire père. - Hérite la classe « AbstractFile ».
	- Classe abstraite. - Représente la base de stockage des données source du langage Mtrans. - Équipé d'une fonction de sauvegarde de trace sur les données après mise à jour. - Hérite la classe « File ».
	- Classe abstraite. - Mère des modèles MVC « DTD ». - Hérite la classe « document ».
	- Classe abstraite. - Mère des modèles MVC « XML ». - Hérite la classe « document ».
	- Classe abstraite. - Mère des modèles MVC « MTRANS ». - Hérite la classe « document ».

Classe	Description
«abstract» AbstractFile +addObserver(view: abstractView) : Boolean	- Classe abstraite. - Représente un objet abstrait observé dans le MVC. - Les observateurs (vues) peuvent s'abonner à cet objet via la méthode « addObserver ». - La notification des observateurs se fait à travers la méthode « notifyObservers ».

DTD	- Représente la base de stockage des données source d'un document DTD. - Hérite la classe « Schema ».
XML	- Représente la base de stockage des données source d'un document XML. - Hérite la classe « Model ».
XSLT	- Représente la base de stockage des données source d'un document XSLT. - Hérite la classe « Transformer ».
MTRANS	- Représente la base de stockage des données source d'un document MTRANS. - Hérite la classe « Transformer ».
«abstract» Folder +addDocument(doc File) : Boolean +remouveDocument(doc : File): Boolean +getChild(index : int) : File +child_name_exist() : Boolean	- classe abstraite. - Représente un dossier dans le système de fichier. - Support la contenance d'objets de type « document ». - Équipé d'une fonction de vérification d'existence de nom pour un élément fils (fils portants le même nom non autorisé). - Hérite la classe « File ».
WorkSpace +addDocument(doc File) : Boolean +getInstance() : WorkSpace	- Représente un dossier de base, incluant tous les projets et modèles de l'utilisateur. - Créé en instance unique (applique le pattern singleton). - Hérite la classe « Folder ».

«abstract» MappingProject +addDocument(doc File) : Boolean	- Représente le dossier incluant tout les données d'un projet de transformation. -Hérite la classe « Folder ».
Validatorerror +localization : Int +errorMessage :String +errorType :String	- Représente une donnée de type erreur de validation. - Inclut une information de localisation de l'erreur. - Inclut une information sur le type de l'erreur. - Inclut une description de l'erreur - Associé aux objets «document».
«abstract» ControlCenter + event (traitement : TraitementType)	- Classe abstraite. - Classe mère de toutes les classes de contrôle MVC. - Définit une méthode abstraite « event », Contenant les traitements de contrôle d'un événement déclenché.
Controller	-La classe de contrôle du MVC. - Contient tout les centres de contrôles - Décide du traitement à faire selon le «TraitementType» envoyé par la méthode «event » - Hérite la classe ControlCenter.
«abstract» EditionCenter + event (traitement : TraitementType) + setContent (content : String)	- Représente le centre de contrôle « édition ». - Traite plusieurs type d'événement concernant l'édition des entités utilisées pour la transformation.
Print	- Fait partie intégrante du centre de contrôle « édition » - Se charge d'établir un lien de communication avec les pilotes de sortie connectés au terminal (imprimante, fax, etc.), pour d'éventuelles impressions d'entités édités.

	- Représente le centre de contrôle « validation ». - S'occupe de l'analyse lexicale et syntaxique et ainsi que la validation des entités fournis par l'utilisateur. - Se charge d'informer l'utilisateur des erreurs en cas de problème.
	- Classe abstraite. - Fournit une méthode abstraite «valid » contenant l'algorithme d'analyse et contrôle de conformité pour une entité fournie. - Fait partie intégrante du centre « validation ».
	- Contient l'algorithme d'analyse et contrôle d'un document « XML ». - Hérite la classe « validator ».
	- Contient l'algorithme d'analyse et contrôle d'un document « Mtrans ». - Hérite la classe « validator ».
	- Représente le centre de contrôle « Transformation ». - Se charge d'effectuer les transformations dans l'application.
	- Classe abstraite. - Classe mère de toute transformation. - Fournit une méthode abstraite « transform » prenant un modèle d'entrée et un modèle de transformation, elle fournit un modèle en sortie. Fait partie intégrante du centre «Mapping».
	- Représente le centre de contrôle « aide » - Fournit des outils d'aide à l'utilisateur. - Hérite la classe « controlCenter ».
	- Classe abstraite. - Classe mère de tous les composants graphiques abonnés aux modèles MVC de l'application. - Fournit une méthode abstraite « update » pour la mise à jour des interfaces.

View +update(observable :abstractFile, traitement TraitementType)	- Classe concrète mère de tous les composants graphiques abonnés aux modèles MVC de l'application. - Associée au modèle. - Hérite la classe « AbstractView ».
Editor	-Représente un conteneur graphique.
MtransEditor	-Représente un conteneur graphique.
MtransTextuelEditor	- Représente un composant graphique d'édition textuel tel qu'un « bloc note ».
MtransGraphicEditor	- Représente un composant graphique d'édition graphique spécifique au langage Mtrans.
XSLTEditor	- Représente un composant graphique d'affichage textuel tel qu'un « bloc note ».
MenuBar	-Représente la barre de menu de l'application. -Contient presque toute les fonctionnalités de l'application.
ToolBar	-Représente la barre d'outils de l'application. -Définie des boutons raccourcis, vers quelque fonctionnalité utilisée souvent par l'utilisateur.
ProjectExplorer	-Représente un explorateur du système de fichier de l'application.

Consol	-Représente un conteneur graphique.
ErrorWindows	-Représente une console pour affichage des erreurs de validation éventuelles dans le document en cour.

TABLE 2.7: Description des classes de l'applications

2.3.2.2 Découpage en package

Les paquetages permettent typiquement de définir des sous-systèmes. Un sous-système est formé d'un ensemble de classes ayant entre elles une certaine relation logique. Souvent, un paquetage fait l'objet d'une réalisation largement indépendante, et peut être confiée à un groupe, ou à un individu n'ayant pas un contact étroit avec les responsables d'autres paquetages [TCO, 02].

Nous avons préférer précéder cette étape sur l'exposition du diagramme de classe globale, à fin de rajouter une plus grande clarté à ce dernier. La figure 2.4 montre cette décomposition en package via un diagramme de package. Nous avons précéder dans notre logique de découpage suivant le modèle MVC (Modèle, vue, contrôle) sur lequel l'application est fondée, est maintenir ainsi les modules ayant une forte cohésion, dans le même package, pour réduire au maximum les canaux de communication entre packages, est ne permettre une telle communication qu'à travers des interfaces bien définit.

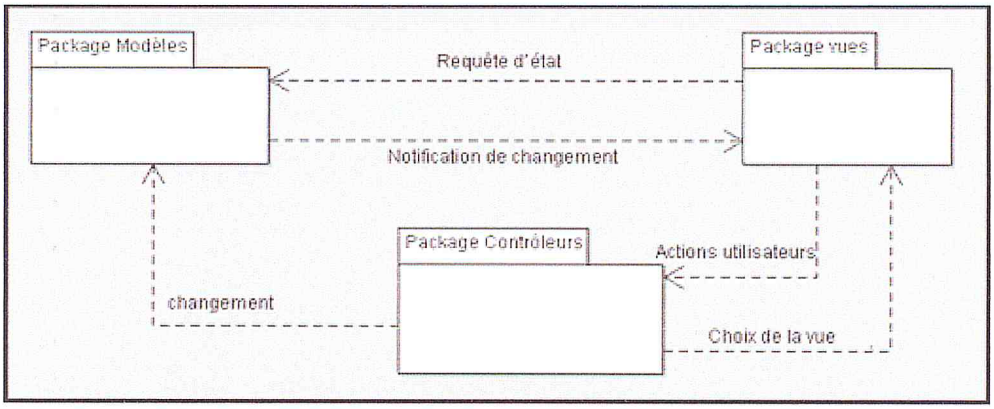


FIGURE 2.4: Diagramme de package de l'application.

2.3.3 Diagramme de classe global

Nous vous exposons dans la figure 2.5 le diagramme de classe global de notre application.

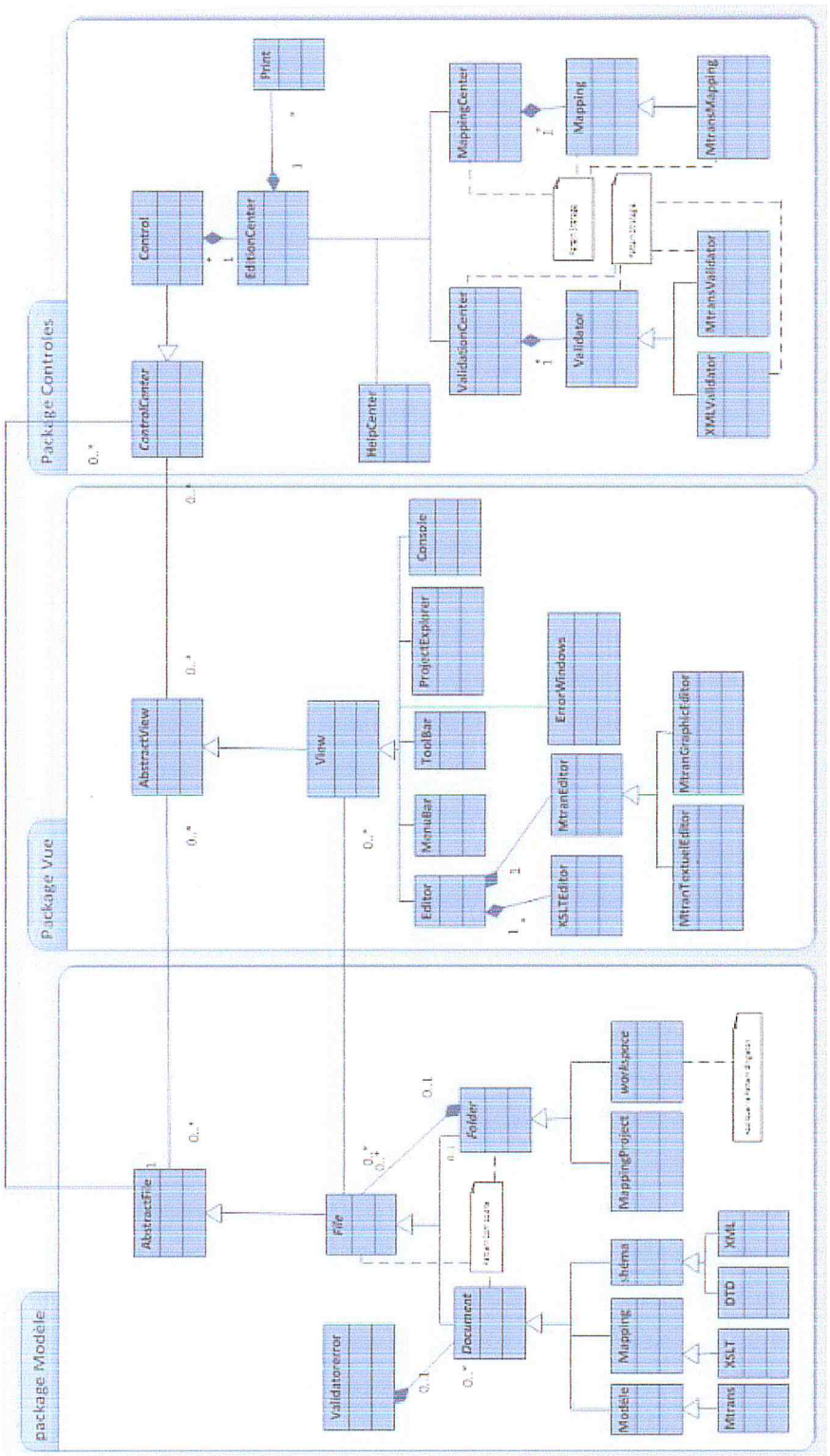


FIGURE 2.5: Diagramme de classe global.

2.4 CONCEPTION DETAILLEE

Dans cette phase nous vous exposerons des fonctionnalités détaillées, qui ne pouvait pas être exprimer dans la phase de conception générale, portant sur les caractéristiques fonctionnelles du Compilateur et l'organisation du système de données, un découpage en composant de l'application est également décrit.

2.4.1 Description du processus de compilation

Nous décrivons dans la figure 2.6 par l'intermédiaire d'un diagramme de séquence UML, le processus de compilation d'un d'un code source ecrit en Mtrans. Chaque flèche continue dans le diagramme représente un appel de fonction (une flèche discontinue présente son rappel) ces fonction sont enclenché à l'intérieur des objets MVC «Modèles», «vues», «contrôleurs» ainsi qu'un objet modélisant un utilisateur, la lecture du diagramme se fait de haut en bas suivant un axe temporel vertical.

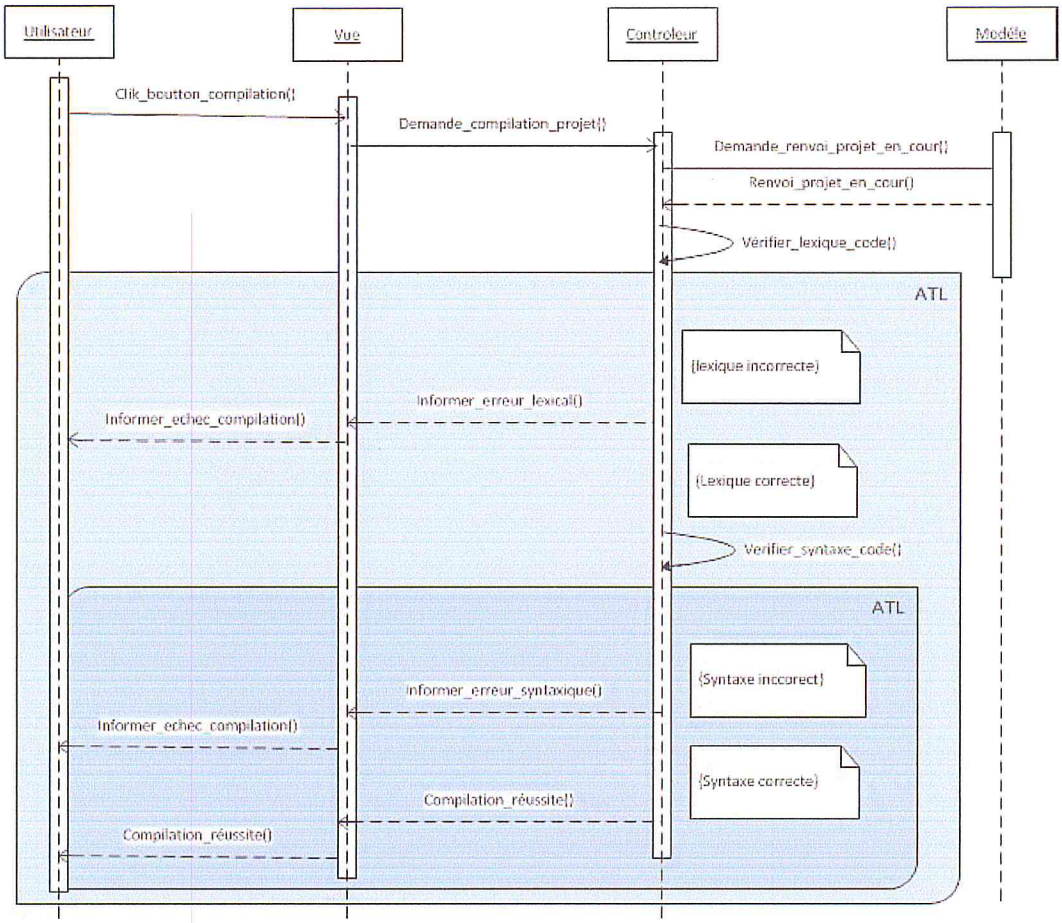


FIGURE 2.6: Diagramme de séquence de Compilation.

2.4.2 Description du processus de mapping

Nous décrivons dans la figure 2.7 par l'intermédiaire d'un diagramme de séquence UML, le processus de mapping (transformation) d'un code source ecrit en Mtrans. La lecture du diagramme se fait de haut en bas suivant un axe temporel vertical.

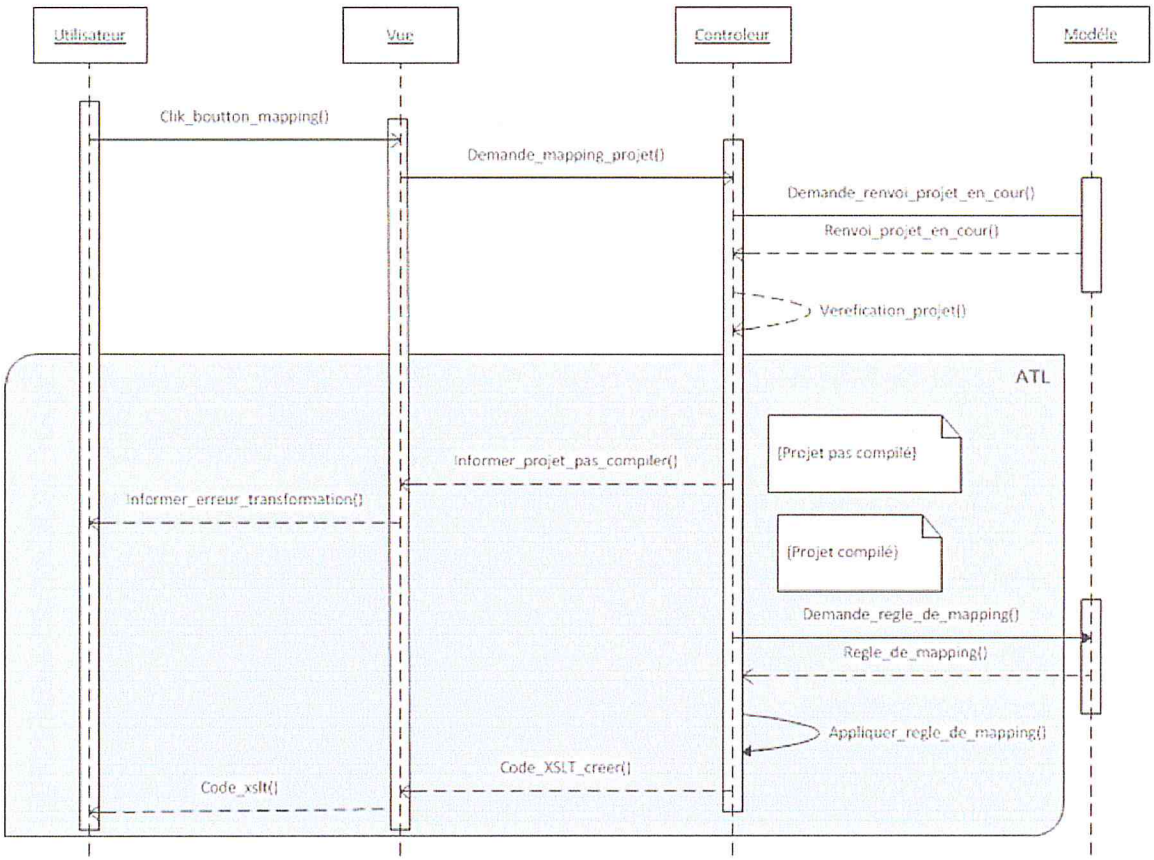


FIGURE 2.7: Diagramme de séquence de mapping.

2.4.3 Decoupage en composant de l'application

Nous avons découpé l'application en composant de façon à privilégier la réutilisabilité de ces derniers dans d'autre implémentation. L'application à été découpée de façon à avoir le maximum de composants réutilisables qui respecte les principes de faible couplage et forte cohésion. La figure 2.8 montre via un diagramme de composant ce découpage.

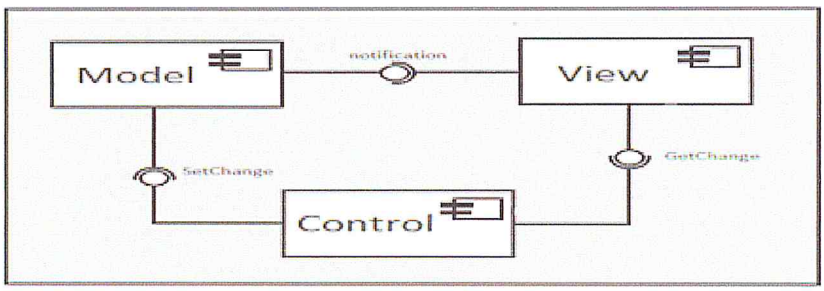


FIGURE 2.8: Diagramme de composant de l'application.

Le diagramme est composé de trois composants, « model » contenant les différents modèles du MVC, « control » contenant les contrôleurs et « view » contenant les vues de celui-ci.

2.5 CONCLUSION

Nous avons exposé dans ce chapitre notre travail d'analyse et de conception de l'application, en faisant appel aux divers outils de modélisation UML. Notre cycle de vie se trouve à ce stade au niveau « codage ». Nous allons utiliser les informations générées dans les phases précédentes pour développer la source de cette application.

Le prochain chapitre est réservé au reste des phases du cycle de vie de développement

Chapitre 3

Implémentation du Système

3.1 INTRODUCTION

Nous aborderons dans ce chapitre les phases restantes de notre cycle de développement représentant les phases de mise en œuvre. Il débutera par la phase "Codage" et s'achèvera par la "génération de produit final".

3.2 CODAGE

Arrivé à ce stade, il ne reste qu'à commencer à écrire notre code en se basant sur les résultats obtenus des chapitres précédents, Pour maintenir l'utilisateur à une portée de vision plus approchée du logiciel, nous avons préféré donner un nom à notre application, cette dernière a été nommée « Mapper» et sera par conséquent appelée ainsi pour le reste des phases de développement. La version qui sera développée et exposée dans ce mémoire est la version 1.0, elle représente le produit final de ce projet.

3.2.1 Langage et outil de développement

Nous avons choisie comme langage de développement pour notre application un langage orienté objet, qui a été jugé en conformité avec la conception que nous avons établis. il s'agit du langage JAVA, pourvu d'une grande sécurité, la richesse de ses bibliothèques, son adaptation à plusieurs plateformes, la qualité présentée par ses composantes graphiques (Swing) qui suivent le modèle MVC, sa facilité de déploiement en réseau (RMI) et le fait qu'on peut avoir plusieurs « Look And Feel », en font de lui un langage redoutable puissant et performant. Une grande partie de sa syntaxe est empruntée de C et C++.

Pour ce qui est du choix de l'outils de développement intégrés (IDE) pour java, notre choix a abouti sur l'un d'entre eux considéré comme l'un des plus puissant IDE JAVA mis sur le marché "Eclipse".

Eclipse est un projet open source à l'origine développé par IBM pour ses futurs outils de développement. Le but est de fournir un outil modulaire capable non seulement de faire du développement en java mais aussi dans d'autres langages et d'autres activités.

Cette polyvalence est liée au développement de modules réalisés par la communauté ou des entités commerciales.

3.2.2 Développement du projet

3.2.2.1 Architecture du projet

Lors de la création d'un nouveau projet dans éclipse la plateforme se charge de créer pour l'utilisateur un répertoire du projet, ce dernier contient la source du programme à développer, il s'occupe en autre d'établir les liens nécessaire avec les librairies (APIs) standards java conxtenu dans la machine virtuelle installé dans la plateforme d'exécution. le programmeur devra par la suite organiser son espace de travail selon ses besoins. Pour notre cas, l'organisation du projet a été établit comme suit :

Figure 3.1

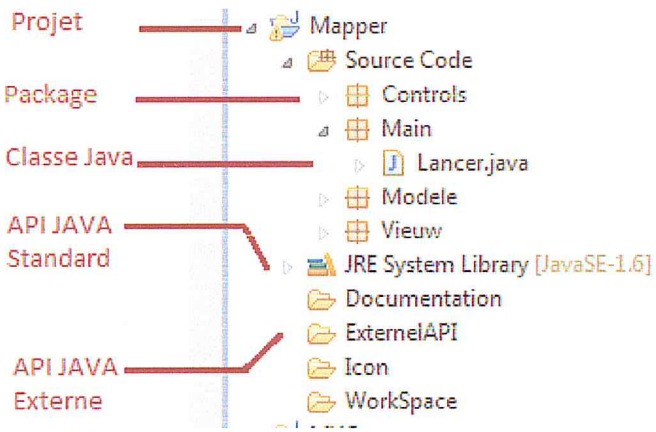


FIGURE 3.1: Architecture de « Mapper 1.0 » dans eclipse.

- Le répertoire « source code » contient la source de l'application.

- Le projet contient également des sous-répertoires de stockage réservé pour le stockage, les icons utilisée dans l'application, un répertoire de documents utile pour la maintenance est également aménagée, et un autre stockant les APIs JAVA externe utilisées

dans l'application, un autre répertoire nommé « WorkSpace » est proposé à l'utilisateur comme répertoire de stockage par défaut de ses éditions.

3.2.2.2 APIs JAVA utilisées

Plusieurs APIs standards et non standard ont été utilisés pour le développent de « Mapper 1.0».

APIs Standard JAVA

Il nous a été indispensable pour développer notre application de prendre comme soutien les APIs standards fournis par la machine virtuelle Java, ces derniers nous ont été utile pour le développement des interfaces d'interaction avec l'utilisateur, le tableau 3.1 décrit ces APIs.

API	Description
Swing	inclut un grand nombre d'interfaces graphiques, pouvant satisfaire la plus part des besoins de développent en cette matière, l'avantage de cette API est que ces composants graphiques sont dites légères est n'occupent donc pas énormément de ressources (processeur, carte graphique) pour leur bon fonctionnent,
Event	sert comme capteur d'événements déclenché par les interfaces graphiques, ces événements sont en relation direct avec les actions de l'utilisateur (click souris ou clavier, navigation curseur etc.).

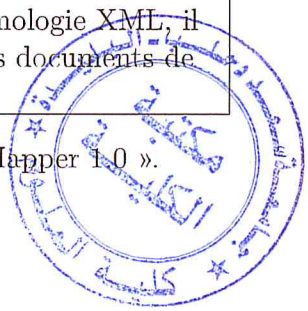
TABLE 3.1: Description des APIs standards utilisées dans « Mapper 1.0 ».

APIs non Standard JAVA

Plusieurs APIs non standards ont été également utilisées, chacune avait un rôle définit, nous avons pris en considération pour la sélection de ces derniers, l'organisme qui les a développé, ainsi que des sondages de satisfaction du publique de développeurs en cette matière.

API	Description
DTD Parser	Cet API propose un bon analyseur syntaxique et sémantique pour les schémas DTD, il figure comme projet de développement indépendant de la machine virtuel Java, il est développer par Sun Microsystems, la version utilisée est la 1.1.Nous allons nous servir de cette API pour développer notre validateur de méta-modèle.
Bounce	Une API graphique développé par un groupe de développement américain nommé « edankert » [EDA], il travail sur plusieurs projets ayant un attachement direct avec la technologie XML, il offre une bonne interface graphique d'édition des documents de balisage tel XML.

TABLE 3.2: Description des APIs externes utilisées dans « Mapper 1.0 ».



3.2.3 Code source

3.2.3.1 Création de modèles

Les modèles de l'application sont comme exprimé dans le diagramme de classe globale sous forme de patron composite, il ressemble plus à un système de fichier miniaturisé. Tout les classes Modèle héritent la capacité d'être observé par les vues, que ça soit des fichiers de données ou leurs répertoires conteneurs, un tel dynamisme va faciliter la notification des vues et permettre un passage directe de notification, au lieu de céder tout cette charge au répertoire de l'espace de travail.

3.2.3.2 Création de vues

Les interfaces graphique de l'application ont été conçu à base des composants graphique de l'API « Swing » exprimé ci-dessus, ces classes héritent toutes la classe « observer » du MVC, elles observent le modèle globale sur plusieurs niveaux, allant d'une observation la plus générale fixé sur l'espace de travail (l'exploreur de projet à ce type d'observation), jusqu'au observation les plus fines se concentrant sur un document contenant le modèle MVC d'une entité édité XML,DTD ou Mtrans(les éditeurs d'entité ont cette portée d'observation).

3.2.3.3 Création de contrôleurs

Les contrôleurs de « MAPPER» sont concentré dans le package « Controllers », chaque sous-package de celui-ci contient la source d'un centre de traitement définit (voir chapitre 2). Ces centres héritent tous la classe « Control », qui est en contact direct avec les vues et modèles de l'application.

3.3 INTEGRATION

Arrivé à cette étape, le codage de l'application est achevé. Nous vous présentons dans ce qui suit le produit final « Mapper 1.0 ».

3.3.1 Présentation de « Mapper 1.0 »

3.3.1.1 Barre de menu

Elle est répartie sur quatre menus comme exposé dans la figure 3.2 , elle supporte toutes les options instaurées.

FIGURE 3.2: Barre de menu « Mapper »

– Menu Fichier

Ce menu (voir figure 3.3) contient toute les options d'archivage des données édités par un utilisateur, elle supporte la création et la sauvegarde des projets MTRANS ainsi que quelques options utiles.

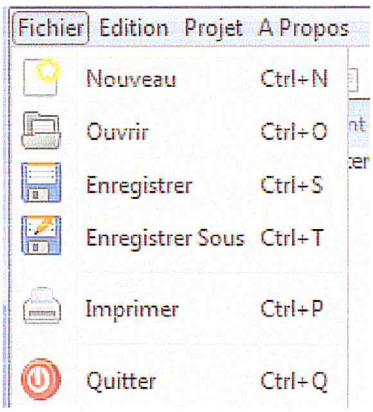


FIGURE 3.3: Menu Fichier « Mapper 1.0 »

– Menu Edition

Ce menu (voir figure 3.4) supporte des options liées à l'édition du code source, ainsi que des options de restauration de données.

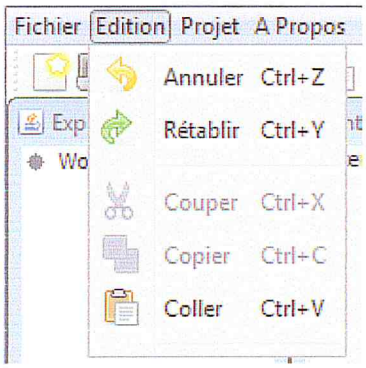


FIGURE 3.4: Menu Editionde « Mapper 1.0 »

– Menu Projet

Ce menu (voir figure 3.5) supporte les options de gestion de projet, l'utilisateur pourra a travers cette fenetre compiler ou bien transformer vers XSLT le projet.

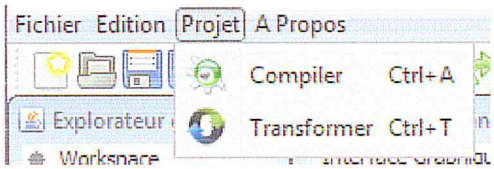


FIGURE 3.5: Menu projet de « Mapper 1.0 »

– Menu A Propos

Ce menu (voir figure3.6) propose de l'aide à l'utilisateur, ainsi qu'une documentation nécessaire a la comprehension du langage Mtrans.

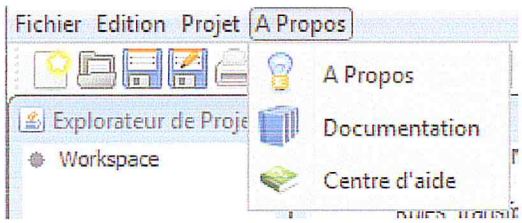


FIGURE 3.6: Menu A Propos de « Mapper 1.0 »

3.3.1.2 Barre d'outils

Cette barre (voir figure3.7) supporte des options souvent utilisées par les utilisateurs, elle a comme objectif de faciliter le travail de ces derniers.

FIGURE 3.7: Barre d'outils de « Mapper 1.0 »

3.3.1.3 Explorateur de projet

Cet anglet (voir figure 3.8) représente graphiquement le système de fichier de notre application ou se fait la sauvegarde du code Mtrans et du code XSLT géneré suite au mapping effectué par l'utilisateur.

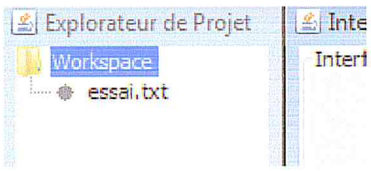


FIGURE 3.8: Explorateur de projet de « Mapper 1.0 »

3.3.1.4 Anglet d'édition

Cet anglet (voir figure 3.9) contient deux partitions, une interface Mtrans dédiée à l'édition du code Mtrans, elle est considérée comme un facilitateur de saisie, et une feuille d'affichage simple du code Mtrans après la compilation du code.

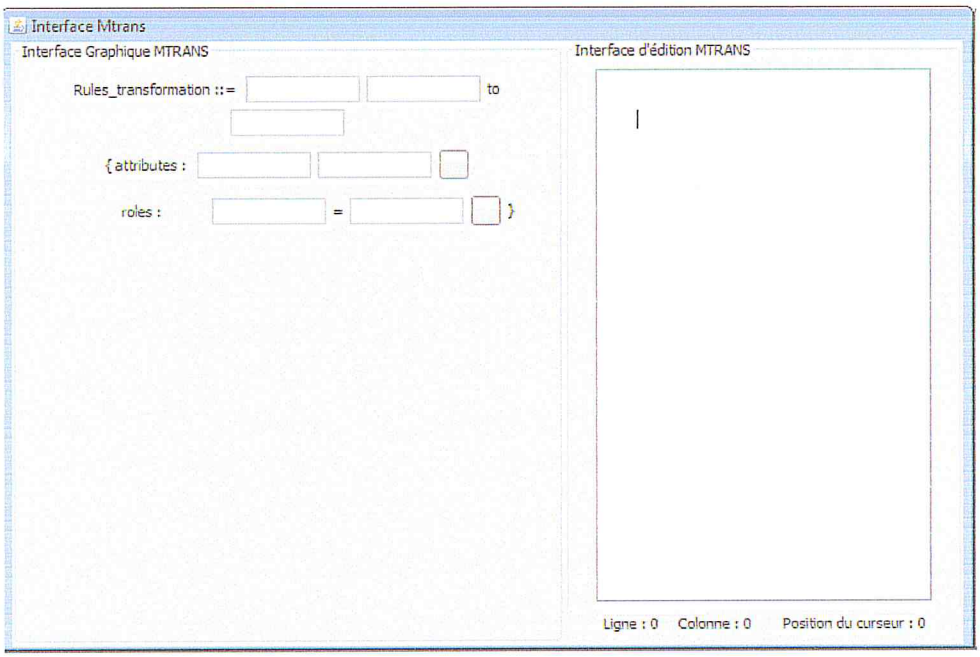


FIGURE 3.9: Anglet d'édition du code Mtrans

3.3.1.5 Anglet d'affichage XSLT

Cet anglet (voir figure 3.10)permet d'afficher le résultat de la transformation du code MTRANS vers XSLT, l'utilisateur pourra par la suite conserver le projet et s'affichera par la suite dans l'anglet : explorateur de projet.

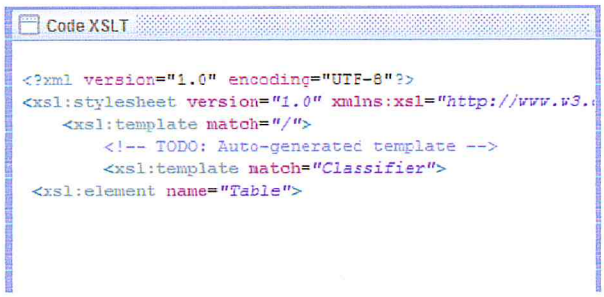


FIGURE 3.10: Anglet d'affichage XSLT de « Mapper 1.0 ».

FIGURE 3.11: Console de « Mapper »

3.3.1.6 Console

Lors d'une analyse des fichiers édités il se peut que l'analyseur rencontre des non-conformités avec la grammaire, cela génère une erreur lors de l'analyse. Cette dernière sera transmise à la console (voir figure 3.11) pour que l'utilisateur puisse la voir. Elle indique non seulement la source qui vient de déclencher l'erreur, mais aussi la ligne et la colonne exacte de l'erreur dans le fichier.

3.3.2 Tests d'intégrités

Nous vous exposons dans cette partie quelques tests d'intégrité qui ont été appliqués sur l'application à fin de vérifier son bon fonctionnement dans des conditions d'erreurs.

3.3.2.1 Conformité avec le système de fichier de la plateforme

Ce test a comme objectif de savoir si « Mapper 1.0» arrive à détecter l'existence d'un fichier pour un chemin spécifique dans la plateforme d'exécution. L'application via un contrôle détecte si un nom de fichier du même nom que le fichier de l'entité a été crée dans le même répertoire, si le fichier existe un choix est offert à l'utilisateur (voir figure 3.12) soit de supprimer le fichier existant ou d'annuler l'opération

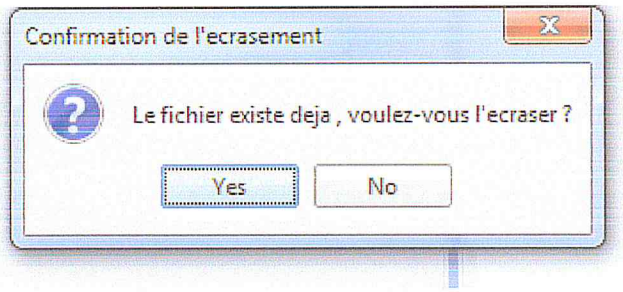


FIGURE 3.12: Fenêtre de dialogue pour l'écrasement de fichier existant

3.3.2.2 Compilation et exécution

Ce test a pour objectif de savoir si notre application arrive à détecter les erreurs de compilation commises par l'utilisateur, le centre de validitation a la charge de détecter ce genre d'erreur et informer l'utilisateur. L'information sur l'erreur est affichée dans la console. (voir figure 3.13), cette analyse d'erreur est utilisée pour la validation ou l'exécution d'un mapping vers XSLT.

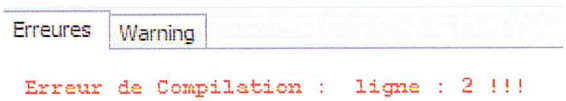


FIGURE 3.13: Détection d'erreur de compilation dans « Mapper ».

3.4 MISE EN PRODUCTION

Nous arrivons à la dernière étape de note cycle de développement. A ce stade, l'application est achevée et peut enfin être utilisée pour la compilation et la transformation. Nous abordons dans cette phase, l'installation de « Mapper » ainsi que la documentation fournis avec l'application pour permettre de comprendre ses divers composants.

3.4.1 Produit « Mapper1.0 »

3.4.1.1 Installation de « Mapper 1.0 »

Le répertoire « Mapper » contenant l'application « Mapper 1.0 » comme donné dans la figure 3.14 , est réparti sur cinq répertoires et un fichier d'exécution « Globe1.0.jar

» .

L'utilisateur n'a alors que lancer le ficher d'exécution « Mapper1.0.jar » pour accéder à l'application.

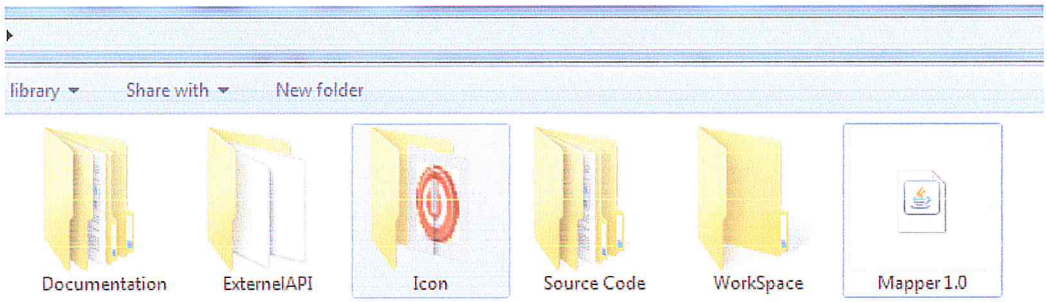


FIGURE 3.14: Contenu du répertoire Mapper

3.4.1.2 A propos de Mapper

Cette fenêtre (Voir Figure 3.15) fournit des informations utiles sur le constructeur. Elle est accessible à partir du menu «A Propos».

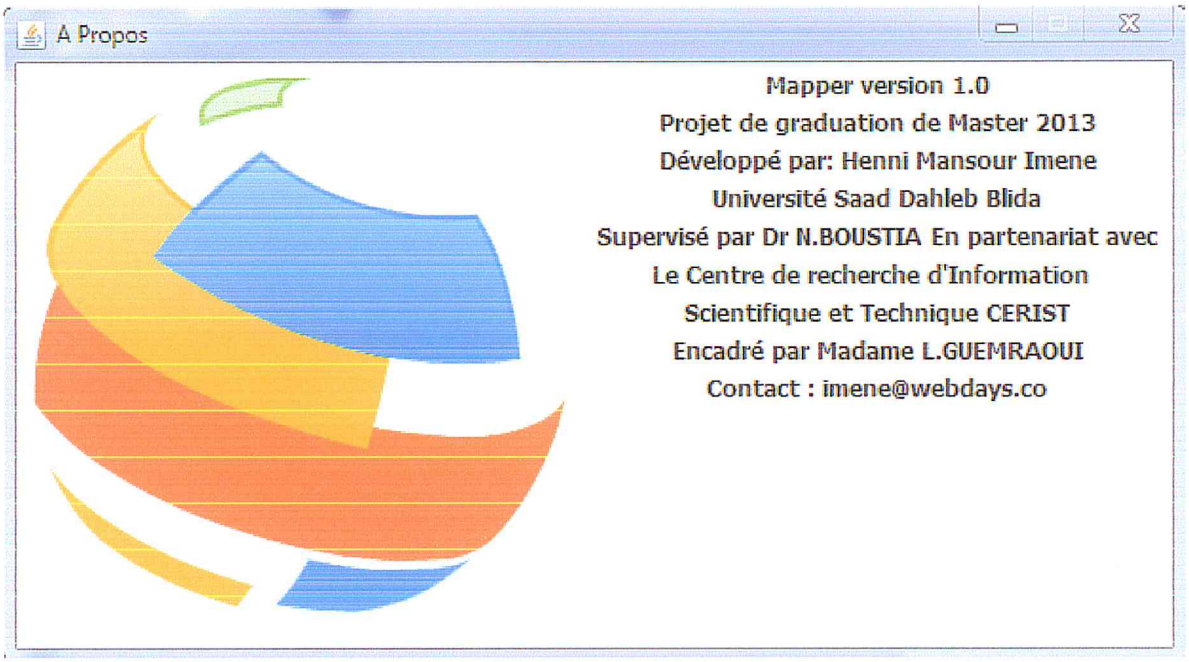


FIGURE 3.15: A Propos de « Mapper 1.0 »

3.4.2 Documentation

Il est mis à la disposions de l'utilisateur, plusieurs documents et tutoriaux qui vont l'aider à mieux comprendre le fonctionnement de « Mapper 1.0 » ainsi qu'une documentation sur le développement de l'application.

3.5 CONCLUSION

Nous avons abordé dans ce chapitre les trois phases d'implémentation de notre application « Mapper 1.0 ».

Nous clôturons, avec ce chapitre, notre cycle de développent avec pour résultat l'application « Mapper 1.0 » et sa documentation.

Chapitre 4

Validation : étude d'un exemple de transformation Mtrans-XSLT sous MAPPER 1.0

4.1 INTRODUCTION

Pour valider notre travail, nous exposons dans ce chapitre un exemple basique d'un processus de transformation Mtrans vers XSLT. Ce processus passe par deux étapes : compilation et mapping vers XSLT.

4.2 EXEMPLE DE MISE EN ŒUVRE : TRANSFORMATION D'UN MODELE ORIENTE OBJET VERS UN MODELE RELATIONNEL

Pour bien comprendre le processus d'implémentation, nous proposons l'étude d'un extrait de transformation d'un modèle objet vers un modèle relationnel, pour une meilleures claretée et une plus grande visibilitée du résultat.

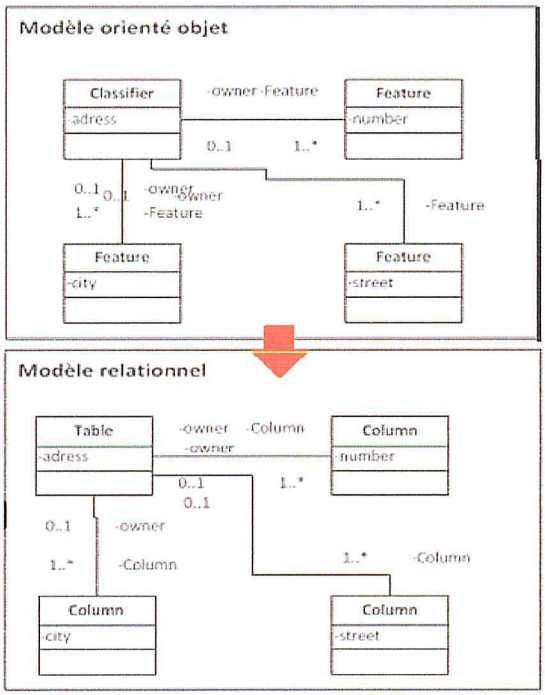


FIGURE 4.1: Un extrait de modèle objet, un extrait de modèle relationnel en UML

D'après une étude [1] qui a été menée, les règles de transformation pourraient être resumées dans le tableau 4.1

Modèle orienté objet	Modèle relationnel
Classifier	Table
feature	Column

TABLE 4.1: règles de transformation

Lorsque de telles règles de transformations sont formulées formellement, la transformation d'un modèle source à un modèle cible consiste simplement à appliquer ces régles sur le modèle source jusqu'à ce que plus aucune règle ne soit applicable grâce a des parseurs bien connus.

4.3 PROCESSUS DE TRANSFORMATION SOUS MAPPER 1.0

Chaque transformation sous MAPPER passe par trois principaux modules :
- ***Editeur****Mtrans* : édition des régles de transformation en MTRANS.
- ***Compilateur****Mtrans* : Compilation des régles éditées en MTRANS.
- ***Transformateur*** : transforrmation des règles MTRANS compilèes vers XSLT.

Dans cette section nous allons detailler le fonctionnement de chaque module à travèrs l'exemple explécité dans la secton 4.2.

4.3.1 Editeur MTRANS

Mapper est mené d'une interface d'édition convivialle qui offre un gabarit de conception de règles MTRANS avec des champs de saisie permettant une saisie regureuse et une facilité d'utilisation.

La figure 4.2 illustre notre exemple écrit en utilisant cette interface.

1. cette étude a été menée au niveau de l'Université de Namur, Belgique, dans le cadre d'un projet de recherche, intitulé : « Transformations de modèles, basées sur XSLT. »

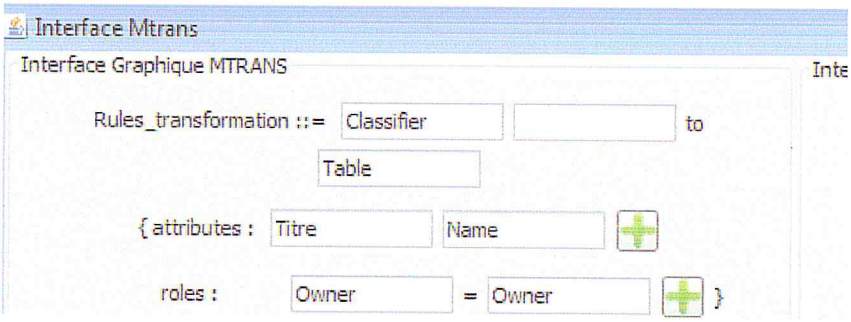


FIGURE 4.2: Exemple Mtrans sous interface graphique Mapper 1.0

4.3.2 Compilateur MTRANS

Le rôle d'un compilateur, est de vérifier si le code introduit est correcte au niveau léxical et syntaxique.

4.3.2.1 Description du compilateur

Analyse lexicale

Le programme est découpé en morceaux (identificateurs, littéraux, ponctuation),ces morceaux subissent une première transformation simple :

– Les littéraux : attribut, rôle, sont traduits en leur valeur.

– Les symboles de ponctuation : (,) , : := , = , { , } ;... et les identificateurs correspondant en fait à des mots réservés : rôles, attributs, set_of_rules, sont traduits en mots-clefs.

– Les autres identificateurs sont représentés par des chaînes.

– Les espaces et les commentaires sont supprimés.

Analyse syntaxique

Cette étape consiste principalement à vérifier si le code introduit est conforme à la grammaire Mtrans.

Le résultat obtenu de l'analyse lexical sera transformé en XML, "attributs" et "roles" seront considérés comme des nœuds, apres l'obtention d'un code semi structuré écrit en

XML, "Mapper" fera la correspondance avec la DTD.

Diagramme UML correspondant à la grammaire MTRANS

Pour realiser l'analyseur syntaxique, en se basant sur le BNF de MTRANS [BEN,09] on a pu exprimer en UML, un extrait du méta-modèle MTRANS, qui se résume dans la figure 4.3 .

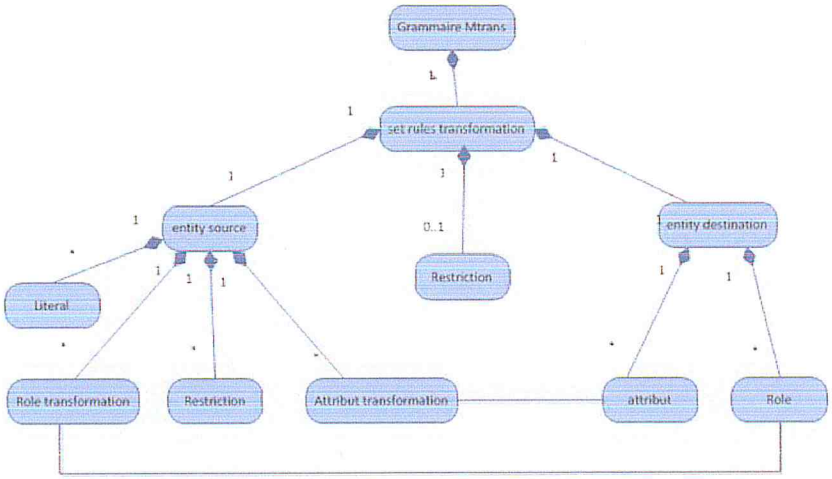


FIGURE 4.3: Representation du méta-modèles Mtrans en UML

Expression de la grammaire MTARANS en DTD

Ainsi, le méta-modèle MTRANS a été défini par un diagramme UML .On appliquant Le mapping diagramme de classe UML vers DTD défini dans l'annexe C , sur les méta-modèles Mtrans, nous obtenons leurs schémas DTD. La figure 4.4 donne un aperçu de la DTD MTRANS.

mtrans.dtd

```
<!ELEMENT rules_transformation(rule_transformation*)>
<!ELEMENT rule_transformation (AO,  Entitys, AF)
<!ELEMENT Entitys (entity,restriction,to,entity,AO,attributes+,roles,AF)>
<!ELEMENT entity(#PCDATA)>
```

FIGURE 4.4: Extrait de la DTD MTRANS

4.3.2.2 Expression des règles de transformations en Mtrans

Reprenons en détail l'exemple proposé à la figure 4.1. Voici les méta-modèles détaillés sous une autre forme :

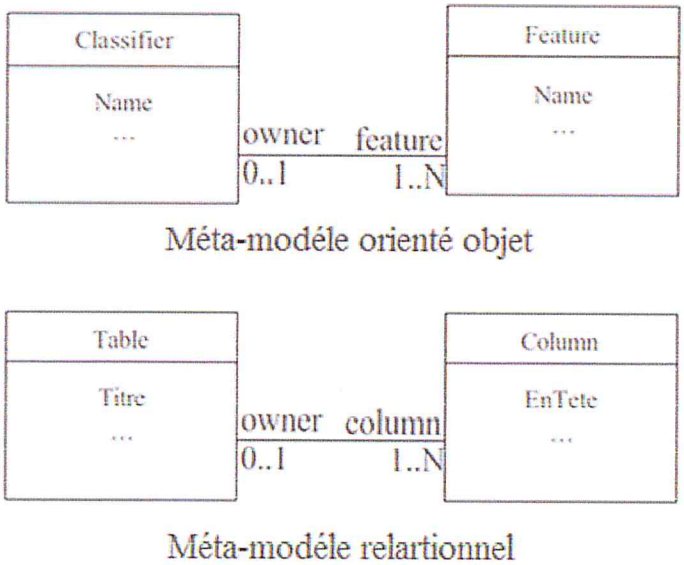


FIGURE 4.5: Méta-modèles objet et méta-modèles relationnel

A partir de ces méta-modèles, on peut composer des règles de transformation en respectant la syntaxe MTRANS .

La figure 4.6 représente le code Mtrans correspondant à notre exemple exprimé sous Mapper 1.0.

Code Mtrans

```
setOfRules
{ Classifier to Table
{ attributes :  Titre = Name, ...
 roles :  owner = owner
}
 Feature to Column
 { attributes : EnTete = Name, ...
  roles :  column = feature
}
}
```

FIGURE 4.6: Code Mtrans exprimé sous Mapper 1.0

La figure 4.7 illustre un extrais du code XML généré suite à la transformation du code Mtrans introduit

Mtrans.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!DOCTYPE  rules_transformation SYSTEM  "mtrans.dtd">

<rules_transformation>
<rule_transformation>
<AO>&AO;</AO>
<Entitys>
          <entity>Classifier</entity>
          <restriction></restriction>
           <to>to </to>
           <entity>Table</entity>
           <AO>&AO; </AO>


           <attributes>
                          <attribute>titre</attribute>
                          <reçoit>=</reçoit>
                          <attribute_transformation>Name</attribute_transformation>
           </attributes>

            <roles>
                        <Role>owner</Role>
                        <reçoit>=</reçoit>
                        <Role_transformation>owner</Role_transformation>
            </roles>
        <AF> &AF;</AF>

</Entitys>

<Entitys>
```

FIGURE 4.7: Code XML équivalent au code MTRANS

4.3.3 Mapping vers XSLT

4.3.3.1 Expression du modèle de transformation

Pour procéder à la transformation du code Mtrans, nous devrons appliquer les règles vue dans la section 1.4.3, Voyons de plus pret le mapping d'une de ces règles :

- Entity1 to Entity2 donne le code XSLT exprimé dans la figure 4.8

4.3.3.2 Exécution de la transformation

Le moreau de code MTRANS illustré dans la figure 4.9, montre un exemple

```
<xsl:template match="Entity1">

  <xsl:element name="Entity2">
       <!--attributes mapping-->
       <!--roles mapping-->
       <xsl:apply-templates select="./child::node()" mode="Entity1"/>
  </xsl:element>

</xsl:template>
```

FIGURE 4.8: code XSLT correspondant a la transformation d'entité.

de règle de transformation de modele objet vers un modele relationnel , qui interprète la transformation d'un classifer en un Table .

```
Code Mtrans
setOfRules
{ Classifier to Table
{ attributes :  Titre = Name, ...
 roles :  owner = owner
}
```

FIGURE 4.9: Etrait de règle de transformation de modele objet vers un modele relationnel exprimè en MTRANS.

En appliquant le mapping MTRANS vers XSLT, on génère, à partir des règles du niveau abstrait en MTRANS, des règles XSLT plus concrètes adaptées aux modèles à transformer. La figure 4.10 traduit le morceau de code XSLT . La totalité du code de transformation de notre exemple est présenté dans l'annexe D.

Mtrans_to_XSLT.xsl

```
        <xsl:template match="Classifier">
 <xsl:element name="Table">
<xsl:variable name="typenode">
      <xsl:value-of select="@Name"/>
 </xsl:variable>
<xsl:if test="$typenode !=''">
      <xsl:attribute name="Titre">
           <xsl:value-of select="$typenode"/>
      </xsl:attribute>
 </xsl:if>
<xsl:if test="$typenode ==''">
      <xsl:element name="Titre">
           <xsl:value-of select="./owner.Name"/>
            </xsl:element>
 </xsl:if>
   <xsl:apply-templates select="./child::node()" mode="single"/>

<xsl:variable name="typenode">
      <xsl:value-of select="@owner"/>
 </xsl:variable>
<xsl:if test="$typenode !=''">
      <xsl:attribute name="owner">
           <xsl:value-of select="$typenode"/>
      </xsl:attribute>
 </xsl:if>
<xsl:if test="$typenode ==''">
      <xsl:element name="owner">
 <xsl:apply-templates select="./owner.owner"  mode="single"/>
    </xsl:element>
 </xsl:if>
  </xsl:element>
```

FIGURE 4.10: Etrait de règle de transformation d'un modele objet vers un modèle relationnel exprimè en XSLT .

4.4 CONCLUSION

Nous avons étudié dans ce chapitre un extrait de transformation d'un modèle objet vers un modèle relationnel, en commençant par une étude indépendante de la plateforme jusqu'à l'exécution de cette transformation dans « Mapper 1.0 ».
Avec ce chapitre nous venons de conclure la deuxième partie de notre mémoire qui englobe tous ce qui concerne la conception et la mise en œuvre de notre application.

Conclusion générale

Le langage XSLT permet de définir des règles de transformations destinées à transformer un document XML bien formé (valide pour une DTD particulière) en un autre.

Xslt presente de nombreux avantages,c'est un langage générique indépendant de toutes plates-formes d'exécution. Son évolutivite, son interopérabilité ainsi que son extensibilité en font de lui un langage complet.

Mais ce langage reste confronté à de nombreuses failles principalement :

- Sa syntaxe longue, illisible et pénible.
- Le coût élevé de sa maintenance pour les programmes associés.
- Son exécution n'est pas conviviale pour la transformation de modèles. Il n'y a pas de messages d'erreur qui dépendent du domaine d'application. Par exemple, si nous voulons transformer un concept qui n'existe pas, le processeur XSLT n'informe pas l'utilisateur.

Pour pallier ce probléme, des solutions ont été proposées, citon principalement les solutions retenues par Mikaël Peltier [MIK,01] qui exploitent XSLT non comme langage de programmation, mais comme environnement d'éxécution des transformations.
En conséquence, nous avons adopté cette solution et nous l'avons implementé.

Nous avons proposé d'étudier le langage MTRANS, ce langage permet d'exprimer naturellement les règles de transformations. L'utilisation d'un tel langage est donc nécessaire surtout que comme le montrent les correspondances entre MTRANS et XSLT, il semble aisé de générer automatiquement, à partir des règles du niveau abstrait en MTRANS, des règles XSLT plus concrètes adaptées aux modèles à transformer.

Notre contribution porte aussi sur la conception et l'implementation d'un système de transformation « MAPPER », muni d'un compilateur qui permet de générer des modèles XSLT. Il a été conçu et mis en œuvre pour servir cet objectif,

Cependant, ce système de transformation est loin d'être complet et nécessite encore de nombreuses recherches afin de l'améliorer et d'en dégager tout le potentiel.

Nos perspectives :

- A court terme, Intégrer des outils de débugage ainsi que des outils de mesure des performances sous Mapper, comme le taux d'occupation mémoire et processeur.
- A longs terme, Intégrer Mapper au outils XSLT déjà existants et étendre la solution vers d'autres langage de transformation tel que ATL.
- Pour conclure nous pensons que l'expérience vécue dans le cadre de ce projet nous a permis de mettre en pratique nos connaissance théoriques acquises au cours de ces cinq dernière années d'étude dans le domaine informatique, et nous a appris la manière de s'adapter avec les technologies modernes notamment dans la conception et la programmation.

Annexes

Annexe A

XML

A.1 DEFINITION

XML « langage extensible de balisage » est un langage informatique de balisage générique (c'est-à-dire un langage qui présente de l'information encadrée par des balises et il est possible d'utiliser une seule déclaration de balise pour tout un groupe de balises préfixées d'un nom identique), Il sert essentiellement à stocker/transférer des données de type texte Unicode structurées en champs arborescents.

Il est qualifié d'extensible car il permet à l'utilisateur de définir les balises des éléments.

A.2 LES REGLES SYNTAXIQUES DE XML

XML utilise des balises pour limiter des présentations concrètes ayant un sens précis.

-Une balise est une chaine de caractère (nom de la balise nommé élément) commençant par le signe < et se terminant par le signe >

Exemple : <reseau_Pétri_modulaire>

-Il doit toujours y avoir une balise ouvrante et une balise fermante pour un même élément, La balise fermante commence par </.

Par exemple : < reseau_Pétri_modulaire > </ reseau_Pétri_modulaire >

-Entre deux balises (ouvrante et fermante) peut y avoir du texte ou pas.

Exemple : <module> </module> ou encore <module> ceci est un module </module>

-Lorsqu'il n'y a pas du texte entre deux balises (ouvrante et fermante) on peut écrire une forme raccourcie : <balise/>

Exemple : <module> </module> deviens <module/>

-Les balises ne doivent pas se chevaucher. Ceci est interdit.

Exemple <module> <place> texte </module> </place> est faux <module> <place> texte </place> </module> est juste.

Les éléments peuvent porter des attributs, délimités par des " ou des '.En général, on utilise les guillemets doubles "

Exemple : <place idf="1" nom="p1" marquage_initil="0" />

-Les documents XML doivent respecter une autre règle : un élément doit contenir tous les autres. On appelle cet élément "élément racine".

-Un prologue peut être placé au tout début du fichier pour indiquer différentes informations (il est optionnel). Ça ressemble à ça : < ?xml version="1.0" encoding="iso-8859-1" standalone="yes" ?>

Le premier attribut : indique des informations sur la version XML utilisé

Le deuxième : est encodage du document Le dernière : indique si le fichier XML est susceptible de recevoir une DTD externe (yes) ou non (no).

Annexe B

DTD

B.1 DEFINITION

XML permet d'utiliser un fichier afin de vérifier qu'un document XML est conforme à une syntaxe donnée. La norme XML définit ainsi une définition de document type appelée DTD (Document Type Definition), c'est-à-dire une grammaire permettant de vérifier la conformité du document XML.

La norme XML n'impose pas l'utilisation d'une DTD pour un document XML, mais elle impose par contre le respect des règles de base de la norme XML.

Ainsi on parle de :

-Document valide pour un document XML comportant une DTD.

-Document bien formé pour un document XML ne comportant pas de DTD mais respectant les règles de base du XML.

Une DTD peut être définie de 2 façons, soit sous forme interne, c'est-à-dire en incluant la grammaire au sein même du document sous forme externe, soit en appelant un fichier contenant la grammaire à partir d'un fichier local ou bien en y accédant par son URI ce qui est le cas des fichiers XML de PetriMod.

B.2 LES REGLES SYNTAXIQUES D'UNE DTD :

Comme pour la syntaxe XML l'information est représenté sous forme de balises, en trouve essentiellement trois type de balises, représentant trois type d'information différentes.

Des déclarations d'éléments, des déclarations d'attributs et des déclarations d'entités.

-Déclarations d'éléments

Un élément est défini par la balise ELEMENT dont la syntaxe est :

< !ELEMENT nom-de-balise modèle-de-contenu>

Un modèle de contenu peut avoir plusieurs déclarations :

Données pures : il s'agit de texte libre représenté par : #PCDATA

exemple : < !ELEMENT module #PCDATA >

Un contenu libre représenté par : ANY

Elément vide : EMPTY

Eléments fils : (palce|transition)+

exemple : < !ELEMENT module (place* | transition* | arc*) >

On peut avoir un contenu mixte comme : (#PCDATA| transition | arc)*

On définit une expression de composition des éléments fils via des operateurs de composition :

Séquence (...,...)

Alternative (...|...)

Regroupement de fragments d'expression(...)

Les operateurs d'occurrence complémentaires à ceux de composition sont :

* :0 a n occurrences

+ : au moins 1 occurrence

? : 0 ou 1 occurrence

Déclarations d'attributs

Il est possible d'ajouter des propriétés à un élément particulier en lui affectant un attribut, c'est-à-dire une paire clé/valeur, en respectant la syntaxe :

< ! ATTLIST Elément Attribut Type >

Type : représente le type de donnée de l'attribut, il en existe trois :

- Littéral : il permet d'affecter une chaîne de caractères à un attribut. Pour déclarer un tel type il faut utiliser le mot clé CDATA
- L'énumération : cela permet de définir une liste de valeurs possibles pour un attribut donné, afin de limiter le choix de l'utilisateur.

La syntaxe de ce type d'attribut est : < ! ATTLIST place nom(Valeur1 | Valeur2 | ...) >.

Pour définir une valeur par défaut il suffit de faire suivre l'énumération par la valeur désirée entre guillemets.

- Atomique : il permet de définir un identifiant unique pour chaque élément grâce au mot clés ID.

Enfin chacun de ces types d'attributs peut être suivi d'un mot clés particulier permettant de spécifier le niveau de nécessité de l'attribut :

#IMPLIED signifie que l'attribut est optionnel, c'est-à-dire non obligatoire

#REQUIRED signifie que l'attribut est obligatoire

#FIXED signifie que l'attribut sera affecté d'une valeur par défaut s'il n'est pas défini. Il doit être immédiatement suivi de la valeur entre guillemets

Remarque1 :

Il existe un autre type de composant pour une DTD ce que l'on appelle une notation.

La syntaxe de déclaration est la suivante :

<!NOTATION nom-notation (PUBLIC|SYSTEM) uri-notation ? Application ?>

Une notation est un mécanisme simple déclarant une entité non analysable la liant à une application via un identificateur.

Remarque2 :

La création et la validation DTD des flux XML, peut être réalisée en JAVA à l'aide de l'analyseur DOM.

Annexe C

Mapping du diagramme de classe UML Vers DTD XML

Permet d'utiliser les DTD afin de vérifier qu'un document XML est conforme à une syntaxe donnée, c'est-à-dire une grammaire permettant de vérifier la conformité du document XML. Ceci nous est utiles du temps qu'on peut exprimer nos modèle avec XML, et vérifier leur conformité en usant des DTD. Ainsi, la DTD joue le rôle de méta-modèle pour des documents XML pouvant alors être considérés comme des modèles.

Toutefois, afin de rapprocher les deux espaces techniques, XML et les modèles, nous avons mené une étude sur les travaux visant la génération automatique des documents XML à partir des diagrammes de classe UML. Une panoplie de travaux à été trouvée dans la littérature, citons : [KUD, 03], [NAR, 05], [SIN, 03], [CON, 00], qui abordent en générale, la création de DTD et de schémas XML à partir des diagrammes de classes. Nous avons adopté les résultats des travaux de [KUD, 03], vu leur simplicité et leur parfaite correspondance à nos besoins. Nous vous exposons dans le tableau suivant un résume du mapping UML vers DTD. Nous avons pris en compte dans ce mapping que les concepts exprimés dans les méta-modèles utilisés dans ce mémoire, nous avons fait abstraction du reste des concepts faisant partie des diagrammes de classe UML tel « l'agrégation ».

UML	DTD
Classe	Un Élément portant le même nom de la classe UML. L'élément a un attribut 'id' de type «ID» pour l'identifier. L'élément a un attribut 'name' (non obligatoire) portant le même nom que l'instance de la classe.
Attribut	Attribut d'élément portant le même nom que l'attribut UML et contenant les mêmes données en format «CDATA».
Association (1-1) à sens unique entre deux classes	Attribut référence (IDREF) de l'ID de l'élément représentant la classe cible de l'association, dans l'élément représentant la classe source.
Association (1-N) à sens unique entre deux classes	Un élément fils de l'élément présentant la classe source de l'association, ayant un attribut référence (IDREF) de l'ID de l'élément représentant la classe cible de l'association.
Relation de composition entre deux classes	L'élément représentant la classe composant et fils de l'élément représentant la classe composé.
Classe abstraite hérité	Non représentée, ces propriétés sont intégrer directement dans la classe qui l'hérite.
Enumération	Non représenté, sauf comme type d'attribut dans une classe (*voir attribut*).

Annexe D

Résultat de la transformation obtenu lors de la validation

Les régles de transformation génerées grace a notre application Mapper 1.0 dans l'exemple de validation sont listées ci dessou :

```
<xsl:template match="Classifier">
        <xsl:element name="Table">
                <xsl:variable name="typenode">
                <xsl:value-of select="@Name"/>
                        </xsl:variable>
                                <xsl:if test="$typenode !=">
                                <xsl:attribute name="Titre">
                                <xsl:value-of select="$typenode"/>
                                 </xsl:attribute>
                                </xsl:if>
                                <xsl:if test="$typenode ==">
                                <xsl:element name="Titre">
                                 <xsl:value-of select="./owner.Name"/>
                        </xsl:element>
                                </xsl:if>
<xsl:apply-templates select="./child::node()"  mode="single"/>
                        <xsl:variable name="typenode">
```

```
                                    <xsl:value-of select="@owner"/>
                                </xsl:variable>
                                <xsl:if test="$typenode !=">
                                  <xsl:attribute name="owner">
                                <xsl:value-of select="$typenode"/>
                                </xsl:attribute>
                                </xsl:if>
                                <xsl:if test="$typenode ==">
                                        <xsl:element name="owner">
<xsl:apply-templates select="./owner.owner" mode="single"/>
                                            </xsl:element>
                                </xsl:if>
                        </xsl:element>
                        </xsl:template>
  <xsl:template match="owner.owner" mode="single">
        <xsl:element name="owner">
 <xsl:apply-templates select="./child::node()" mode="single"/>
        </xsl:element>
  </xsl:template>
<xsl:template match="Feature">
                        <xsl:element name="Column">
                        <xsl:variable name="typenode">
                        <xsl:value-of select="@Name"/>
                        </xsl:variable>
                                <xsl:if test="$typenode !=">
                                        <xsl:attribute name="EnTete">
                                 <xsl:value-of select="$typenode"/>
                                </xsl:attribute>
                           </xsl:if>
                                <xsl:if test="$typenode ==">
                                        <xsl:element name="EnTete">
                                        <xsl:value-of select="./owner.Name"/>
                                        </xsl:element>
                            </xsl:if>
```

```
<xsl:apply-templates select="./child::node()"  mode="single"/>
                <xsl:variable name="typenode">
        <xsl:value-of select="@feature"/>
                </xsl:variable>
                        <xsl:if test="$typenode !=">
                        <xsl:attribute name="column">
                         <xsl:value-of select="$typenode"/>
                    </xsl:attribute>
</xsl:if>
<xsl:if test="$typenode ==">
<xsl:element name="column">
<xsl:apply-templates select="./owner.feature"  mode="single"/>
   </xsl:element>
</xsl:if>
</xsl:element>
</xsl:template>
<xsl:template match="owner.feature" mode="single">
  <xsl:element name="column">

<xsl:apply-templates select="./child::node()" mode="single"/>
  </xsl:element>
</xsl:template>
```

Bibliographie

[BEN,09] Benoit Georges , Transformations de modèles, basées sur XSLT, « Article»,Universitè de Namur, Belgique.2009

[BEZ, 04] Jean Bézivin. Sur les principes de base de l'ingénierie des modèles . « Publication », Université de Nantes,2004

[CHA,10] B.Charroux, A.Osmani and Y.Thierry-mieg, UML2 3^{éme}édition, « Livre », Pearson Education France, 2010.

[CON, 00] R. Conrad, D. Scheffner, & J.C. Freytag," XML conceptual modeling using UML". Proceedings of the19th International Conference on Conceptual Modeling (ER'2000). Octobre 2000.

[ELB,04] G.El boussaidi and H.Milil, « Les patrons de conception : Représentation et mise en œuvre », Laboratoire de recherche sur les Technologies du Commerce Electronique Université du Québec à Montréal, 2004.

[FRA, 06] Franck Fleurey. Langage et méthode pour une ingénierie des modèles fiable. « Thèse de doctorat ». Université de Rennes 1. 2006

[JON,97] R.Johnson, « Frameworks = (components+patterns) », « Article », Communications of the ACM, 1997.

[KAY,05] Michael Kay. La référence du développeur XSLT. « Livre »,Paperback edition, 2005.

[LAU,09] Laurent Audibert, UML 2 - de l'apprentissage à la pratique. « Livre », Ellipses, 2009

[KUD, 03] T. Kudrass & T. Krumbein, "Rule-Based Generation of XML DTDs from UML Class Diagrams.", Proceedings of ADBIS. p. 339-354, 2003.

[MIK,01] Mikaël Peltier, Jean Bézivin Gabriel Guillaume. MTRANS : Ageneral framework, based on XSLT, for model transformation. Workshop on Transformations in UML (WTUML), Italie, 2001.

[NAR, 05] K. Narayanan & S. Ramaswamy. "Specifications for Mapping UML Models to XML Schemas". Proceedings of the 4th Workshop in Software Model Engineering (WiSME 2005), Montego Bay, Jamaica, 2005.

[PEL, 00] Mikaël Peltier, François Ziserman, Jean Bézivin. « Concrete and Abstract Spaces in Model Engineering, Information Systems Engineering : Conceps and Industrial Applications » World (ISICA), Multiconference on Systemics, Cybernetics and Informatics, 2000.

[SAM,10] Samba Diaw, Rédouane Lbath, Bernard Coulette : État de l'art sur le développement logiciel basé sur les transformations de modèles. Technique et Science Informatiques,Université de Toulouse (2010)

[SIN, 03] J. Singh." Mapping UML Diagrams to XML". Thèse de master, Jawaharlal Nehru University, New Delhi, 2003.

[TCO,02.] Institut de Télécommunications de l'Ecole d'Ingénieurs du Vaud, « Introduction à la programmation orientée objets avec UML », Institut de Télécommunications de l'Ecole d'Ingénieurs du Canton de Vaud –suisse-, 2002.

[THO, 08] Frédéric Thomas,Contribution à la prise en compte des plates-formes logicielles d'exécution dans une ingénierie générative dirigée par les modèles. Thése de doctorat. Université d'Evry, France.2008.