MASTER EN SCIENCES INFORMATIQUES

Contributions à la méthodologie de conversion de données en XML

Delcroix, Christine

Award date:
2000

Facultés Universitaires Notre-Dame de la Paix
Institut d'Informatique
Namur

Contributions à la méthodologie de conversion de données en XML

Christine DELCROIX

Mémoire présenté en vue
de l'obtention du grade
de Licencié en Informatique

Promoteur : J.-L. Hainaut

Année académique 1999-2000

Résumé

Ce mémoire s'intéresse à la production de documents XML à partir d'une base de données. Plus précisément, il propose une méthodologie permettant le développement automatique++ d'applications de conversion de données en XML. Chaque application est dédiée à une base de données et une structure de document(s) XML particulières. Cette méthodologie doit permettre de traiter les cas où la structure des documents XML a été fixée indépendamment de celle de la base de données pour pouvoir s'appliquer dans un contexte d'échange de données entre entreprises.

Abstract

This thesis is about XML documents production from a database. More precisely, it proposes a methodology allowing the automatic development of applications for converting data into XML documents. Each application is dedicated to a particular database and a particular XML documents structure. This methodology allows us to treat the cases where the structure of XML documents was defined independently of the structure of the database in order to be applied in a context of data exchange between companies.

Bien des personnes doivent être remerciées pour leur contribution à la réalisation de ce travail.

Scientifiquement parlant, c'est vers l'équipe DB-MAIN toute entière que je me tourne avec, cependant, une attention particulière pour Philippe. C'est avec beaucoup de patience qu'il répondit à mes nombreuses questions sans oublier les heures passées à la relecture de ce mémoire à une période où il avait, certes, bien d'autres choses à penser !

Dans un tout autre domaine, c'est à Samuel que je dois mes plus grands encouragements. Son soutien moral, logistique et disons, syntaxique me fut extrêmement précieux.

Enfin, je ne saurais citer toutes les personnes qui, à travers leurs mails, leurs coups de téléphone ou leurs visites s'enquirent de l'avancée de mon travail. Qu'elles en soient ici remerciées.

Table des matières

Introduction **7**

1. Le langage XML **10**

1. Présentation générale 11

2. Origine et objectifs 13

3. Syntaxe des documents XML 15

3.1. Le prologue *16*
3.2. Le corps *16*
3.3. L'épilogue *17*

4. Définitions de Type de Document (DTD) 18

4.1. Définitions de contraintes sur les éléments *18*
4.2. Définitions de contraintes sur les attributs *20*
4.3. Exemple récapitulatif *22*

2. Application de production de document(s) XML **24**

1. Présentation générale 25

2. Architecture du migrateur 28

2.1. L'extracteur *28*
2.2. Le convertisseur-formateur *30*
2.3. Architecture améliorée *31*
2.4. Remarque *31*

3. Modélisation de données et XML 32

3.1. Modélisation des types d'entité d'un schéma logique d'une base de données *34*
3.2. Modélisation des attributs d'un type d'entité *36*
3.3. Nombre de documents XML *38*

3. Démarche méthodologique pour la génération d'un migrateur **39**

1. Aperçu général de la démarche méthodologique 40

2. Rétro-ingénierie 41

3. Structure d'un document XML 42

3.1. Représentation de la structure d'un document XML (DTD) *42*
3.2. Restriction sur les DTD *48*
3.3. Lacunes de la représentation *49*
3.4. Le modèle XML *50*

4. Etablissement des correspondances 52

4.1. Comparaison. *52*
4.2. Approche transformationnelle *52*
4.3. Avantages de l'approche transformationnelle *55*

5. Démarche méthodologique enrichie 56

4. Construction d'un prototype de migrateur BD/XML **57**

1. Outils 58

1.1. Le langage de programmation *58*
1.2. Le module d'accès au DMS *58*

2. Architecture du prototype 60

3. Implémentation du migrateur 62

5. Mise en pratique de la démarche méthodologique 63

1. Outil : DB-MAIN 64

1.1. Présentation générale 64
1.2. Voyager 65
1.3. Historique des activités d'ingénierie 65
1.4. Le langage MDL 65

2. Démarche méthodologique 67

3. Etablissement d'une chaîne de transformations 68

4. Analyse de la chaîne de transformations 70

4.1. Création des méta-propriétés 71
4.2. Traitement du journal 71
4.3. Contrainte résultant de cette démarche 72

5. Génération du migrateur 73

6. Implémentation de la méthodologie 74

6. Etude de cas 76

1. Présentation du cas 77

2. Rétro-ingénierie 80

3. Elaboration du schéma XML 81

4. Analyse des transformations 85

5. Génération du migrateur et génération du DTD 86

6. Documents XML 87

Conclusion 89

Bibliographie 92

Annexes 94

A. Le modèle GER 95

1. Présentation générale 95
2. Description des constructions du modèle GER 95

B. Description de l'interface des classes du migrateur 104

Introduction

La capacité croissante des mémoires, leur coût de plus en plus accessible ainsi qu'une large diffusion de l'outil informatique ont contribué au stockage de quantités énormes de données, parfois même sans objectif réel.

Parallèlement à cela, le développement d'Internet a ouvert la porte aux échanges d'informations à un niveau planétaire avec un problème important à surmonter : celui de l'hétérogénéité tant du point de vue du *software* que du *hardware*.

Dans ce contexte, il a, entre autre, fallu trouver un langage commun de représentation et d'échange de données capable de traiter avec des systèmes très différents. Entre autres projets, c'est XML qui semble se démarquer en alliant simplicité et puissance. De plus, avantage considérable, il convient parfaitement au monde de l'Internet.

Bien que le futur de XML soit difficilement prévisible - sera-t-il uniquement utilisé comme format d'échange de données ou le sera-t-il également pour stocker des données de manière permanente ?-, il reste inséparable de la technologie des bases de données. Les interactions entre XML et les bases de données se retrouvent dans des situations très diverses. Par exemple,

- de nombreux sites web, vitrine d'activités commerciales, doivent être régulièrement remis à jour à partir d'informations stockées dans une base de données. Une manière de procéder consiste à extraire les informations de la base de données et de les stocker dans des documents XML qui seront directement visualisés sur Internet;
- de nombreuses organisations souhaitent échanger des informations de même nature avec d'autres organisations du même domaine d'application. Il va sans dire que des règles doivent être établies afin de définir une structure d'échange commune. XML s'avère particulièrement adéquat pour réaliser cet objectif car il offre un mécanisme de description de la structure de document. De plus, le traitement des documents XML peut être automatisé puisqu'ils sont basés sur une syntaxe formelle;

Il semble donc nécessaire de disposer d'applications de transfert de données de bases de donnés vers des documents XML ou inversément.

C'est précisément aux **applications de transfert de données d'une base de données vers un ou plusieurs document(s) XML** que nous nous intéressons dans le cadre de ce mémoire.

Plus exactement, l'objectif consiste à **développer une méthodologie** permettant d'aboutir à la création d'une application de production de document(s) XML. L'application qui en résulte est spécifique à une base de données particulière et à une forme de document(s) XML bien définie.

Plus précisément, nous nous intéressons au domaine de l'échange de données. Il est donc impératif de supporter le développement d'application de production de documents XML dont la structure peut avoir été fixée indépendamment de celle de la base de données.

Nous avons restreint notre étude aux seules applications permettant de produire des documents XML à partir de données sélectionnées sur une base structurelle (ex : les données correspondantes aux colonnes `Auteur`, `Titre` et `Année_de_Parution` de la table `Livre`). On ne considère donc pas les cas de sélection de données sur base de condition sur leur valeur (ex : les données correspondantes aux colonnes `Auteur`, `Titre` et `Année_de_Parution` de la table `Livre` pour lesquels la valeur correspondante à la colonne `Année_de_Parution` est supérieure à 1995).

Etat de l'art

Le projet SilkRoute (cf. [1]) poursuit un objectif relativement proche du nôtre, bien que plus large. Cependant, l'approche en est fondamentalement différente. Elle consiste à développer une application unique capable de répondre aux besoins de production de documents XML quelle que soit la base de données et la structure des documents XML. L'application est configurée pour chaque cas via un langage propriétaire relativement complexe. L'application se veut **générale** en permettant de supporter les cas où la structure des documents XML a été fixée indépendamment de celle de la base de données, **dynamique** car elle veut pouvoir répondre immédiatement aux requêtes et **efficace** en exploitant pleinement les capacités du moteur de la base de données (exclusivement relationnelle) pour la sélection des données. Il semble raisonnable de penser que notre approche aboutira à de meilleures performances sans toutefois permettre le dynamisme de celle du projet SilkRoute.

Oracle s'est également penché sur la production de documents XML. L'outil XSQL est relativement restreint puisqu'il ne produit que des documents XML dont la structure est en correspondance directe et simple avec celle de la base de données. Il ne permet pas de se conformer à une structure de documents XML préétablie comme dans le cas de l'échange de données entre entreprises d'un même domaine.

Organisation du mémoire

Le chapitre 1 contient une introduction au langage XML destiné au lecteur novice dans ce domaine. Il ne constitue en rien une description formelle et précise. Il ne fait qu'introduire aux principaux concepts de XML que nous aurons à utiliser.

Le chapitre 2 propose une architecture pour l'application de production de document(s) XML. Toutes les considérations faites dans ce chapitre sont indépendantes du choix des outils. Le chapitre expose également des pistes pour la conception de la structure du ou des document(s) XML destinés à contenir des données issues d'une base de données.

Le chapitre 3 présente une démarche méthodologique permettant d'aboutir à la construction d'une telle application à partir de la structure de la base de données, de celle du ou des document(s) XML à produire et des correspondances entre ces deux structures.

Le chapitre 4 décrit l'implémentation de l'application de production de document(s) XML. Il présente les outils qui ont été utilisés et précise l'architecture du chapitre 2.

Le chapitre 5 décrit la mise en pratique de la démarche méthodologique permettant d'aboutir à la construction de l'application de transfert des données. Ici aussi, une description des outils utilisés précède à une description de la démarche méthodologique, plus précise qu'au chapitre 3.

Le chapitre 6 illustre la démarche méthodologique sur un cas pratique et simple.

Enfin, une conclusion précède les deux annexes, la première présente le modèle GER (Generic Entity Relationship) largement utilisé dans le cadre de ce travail, alors que la seconde détaille l'implémentation d'une application de production de document(s) XML.

CHAPITRE 1

Le langage XML

L'objectif de ce chapitre est de donner un aperçu du langage XML.

La première section positionne le langage XML par rapport au langage HTML, plus répandu. Ensuite, nous retraçons le contexte dans lequel XML a été créé. Enfin, les deux dernières sections décrivent les éléments de la syntaxe du langage XML nécessaires à la compréhension du présent document. Ces dernières ne constituent en rien une description complète de la syntaxe du langage XML.

1. Présentation générale

XML est un langage de balisage[1] directement dérivé de SGML[2], tout comme HTML[3]. Mais, à la différence de ce dernier, il n'existe pas de balises prédéfinies en XML. C'est, entre autre, ce qui fait la force de XML. L'utilisateur peut créer librement toutes les balises qu'il souhaite. D'où le nom XML, acronyme de e**X**tensible **M**arkup **L**angage, indiquant que le langage XML n'est pas figé.

```
<meteo>
    <Bulletin id="P233Y667MF606"/>
    <Observations>
        <loc>Paris-Montsouris</loc>
        <date fmt="ISO-8601">1998-10-22T15:31:05</date>
        <temp unit="Celsius">12.3</temp>
    </Observations>
</meteo>
```

Figure 1.1- Exemple de bulletin météorologique écrit en XML.

Un document HTML doit se conformer à un certain nombre de règles (par exemple, la balise `<TITLE>` est autorisée; en revanche la balise `<TITRE>` ne l'est pas). Ces règles ont été établies par les concepteurs du langage HTML. Concevoir une nouvelle version de HTML consiste à modifier ces règles.

Dans le contexte XML, de telles règles peuvent également être établies. Elles sont, alors, rassemblées dans un fichier appelé DTD (**D**éfinitions de **T**ype de **D**ocument). Mais ici, n'importe quel utilisateur peut définir son propre DTD. Autrement dit, n'importe qui peut, en fonction de ses besoins, concevoir un langage c'est-à-dire un ensemble de contraintes auxquelles doivent se conformer les documents XML.
Par exemple, dans le domaine de la météorologie, on peut définir, via l'utilisation d'un DTD, un langage qui accepte uniquement des documents contenant les balises `<meteo>`, `<Bulletin>`, `<Observations>`, `<loc>`, `<date>` et `<temp>` (comme à la Fig. 1.1). C'est dans ce sens que l'on qualifie également XML de métalangage car il permet de définir des langages spécifiques à un domaine d'application. La section 4. détaille le fonctionnement du DTD.

---

1. On désigne par le mot **balise**, une commande insérée dans un document qui spécifie comment le contenu, ou une portion de ce contenu, devra être structuré. Les balises sont utilisées dans des spécifications de format qui stockent des documents sous forme de fichiers de texte telles que SGML, HTML ou XML. Dans ce contexte, une balise se présente sous la forme d'une suite de caractères délimitée par les caractères "<" et ">".
2. SGML (**S**tandard **G**eneralized **M**arkup **L**angage) (ISO 8879) est un standard permettant de spécifier un langage de balisage de document (via l'utilisation d'un DTD). Pour plus d'informations sur le langage SGML, consultez [2].
3. HTML (**H**yper**T**ext **M**arkup **L**angage) est une implémentation relativement simple de SGML (via la définition d'un DTD) destinée à la création de documents hypertextes pour le Web. Il fut inventé par Tim Berners-Lee du CERN, le laboratoire européen de physique des particules situé à Genève, en 1989. Pour plus d'informations sur le langage HTML, consultez [3].

XML permet de représenter des données destinées à diverses applications. Ainsi, un document XML peut servir à présenter des données à un utilisateur ou peut être échangé et manipulé par des applications sans intervention humaine.
A titre d'exemple, le bulletin météorologique de la figure 1.1 peut être affiché sur le Web afin d'informer les utilisateurs des mesures météorologiques du jour, mais peut également alimenter une base de données en vue d'une analyse statistique.

Contrairement à HTML, un document XML ne contient aucune information relative à la présentation de son contenu. La présentation d'un document XML est dictée par l'utilisation d'une feuille de style. Cette séparation du contenu et de la présentation donne l'avantage de pouvoir facilement présenter les mêmes données de différentes manières. Par exemple, dans un document XML contenant plusieurs bulletins météorologiques (cf. Fig. 1.1), on pourrait associer une feuille de style représentant les valeurs de température sous forme graphique et une autre feuille de style représentant ces mêmes valeurs sous la forme d'un tableau.

2. Origine et objectifs

HTML, un des principaux formats de documents sur le Web, est un langage qui comporte trois avantages principaux :

- la simplicité : HTML est facile à apprendre et à comprendre;
- les liens : les liens hypertextes sont très faciles à créer;
- la portabilité : étant donné la faible quantité de balises, il est facile d'intégrer la structure des documents HTML dans des navigateurs;

C'est ce qui a contribué à sa large acceptation.

Cependant, HTML présente différents problèmes notamment lorsqu'il s'agit de développer des moteurs de recherche. En effet, HTML n'offre qu'une recherche plein texte, ce qui la plupart du temps mène à des résultats qui n'ont rien à voir avec la recherche de départ. De plus, puisque la recherche porte sur le document entier, elle est peu efficace.
D'autre part, HTML n'est qu'un langage de présentation de données sur Internet et il n'est pas conçu pour répondre à des besoins de structuration et d'échange de données.
Enfin, les liens sous HTML sont souvent brisés. HTML a été conçu comme si les objets présents sur l'Internet ne pouvaient pas changer de place. Il y a donc nécessité d'améliorer le système de liens HTML.

SGML, quant à lui, représente la solution aux principaux inconvénients de HTML :

- le langage SGML n'a pas de balises prédéfinies. Les balises sont créées en fonction des besoins, ce qui permet une représentation intelligente des structures de données propres à un domaine d'application. Les moteurs de recherche pourraient aboutir à de meilleurs résultats s'ils exploitaient l'intelligence contenue dans les balises des documents;
- les liens hypertextes sont plus riches.

Malheureusement, SGML est trop complexe : les textes nécessitent une validation, les méthodes de liens hypertextes sont extrêmement complexes, ... Cette complexité est un frein à la construction de navigateurs capables d'opérer avec SGML[1].

1. Il est à remarquer que cette section contient une comparaison entre HTML et SGML qui ne respecte pas la chronologie des évènements. En effet, SGML est largement antécédent à HTML. En réalité, HTML est une implémentation simple de SGML.

A la fin de 1996, cherchant un compromis entre la simplicité de HTML et la puissance de SGML, un groupe de travail du consortion W3C (World Wide Web Consortium) nommé *XML Working Group* s'est attaqué aux problèmes soulevés ci-dessus. Le cahier de charges du nouveau langage, baptisé XML, était le suivant :

- XML doit être utilisable sans difficulté sur Internet;
- XML doit soutenir une grande variété d'applications;
- XML doit être compatible avec SGML;
- Il doit être facile d'écrire des programmes traitant les documents XML;
- Le nombre d'options dans XML doit être réduit au minimum, idéalement à aucune;
- Les documents XML devraient être lisibles par l'homme et raisonnablement clairs;
- La conception de XML devrait être préparée rapidement;
- La conception de XML doit être formelle et concise;
- Il doit être facile de créer des documents XML;
- La concision dans le balisage de XML est de peu d'importance.

C'est ainsi que naquit, en 1998, le langage XML.

3. Syntaxe des documents XML[1]

Un document XML peut être soumis à un ensemble de contraintes réunies dans un fichier appelé DTD. Cependant, la présence d'un DTD n'est pas obligatoire.
Lorsqu'un document XML est conforme aux spécifications de la syntaxe XML, on le qualifie de **bien formé**. Lorsqu'un document XML est bien formé, qu'il est soumis à un DTD et qu'il vérifie les contraintes exprimées dans ce DTD, on dit qu'il est **valide** par rapport à ce DTD.

Cette section s'attache à décrire les conditions à vérifier pour qu'un document XML soit bien formé.

Comme le montre la figure ci-dessous, un document XML est constitué de trois parties : le prologue, le corps du document et l'épilogue.

Prologue

```
<?xml version="1.0"?>
<!-commentaires et instructions de traitement->
<!DOCTYPE meteos SYSTEM "http://monserveur.com/Meteo.dtd>
<!-commentaires et instructions de traitement->
```

Corps

```
<meteos>
    <meteo>
        <Bulletin id="P233Y667MF606"/>
        <Observations>
            <loc>Paris-Montsouris</loc>
            <date fmt="ISO-8601">1998-10-22T15:31:05</date>
            <temp unit="Celsius">12.3</temp>
            <hygro>88</hygro>
            <nebulo>8</nebulo>
            <anemo>3</anemo>
            <pluvio base="24">6</pluvio>
        </Observations>
    </meteo>
    <meteo>
        <Bulletin id="P233Y667MF607"/>
        <Observations>
            <loc>Paris-Montsouris</loc>
            <date fmt="ISO-8601">1998-10-23T22:58:09</date>
            <temp unit="Celsius">13.8</temp>
            <anemo>5</anemo>
            <pluvio base="24">7</pluvio>
        </Observations>
    </meteo>
</meteos>
```

Epilogue

```
<!- commentaires et instructions de traitement->
```

Figure 1.2- Un exemple de document XML.

1. **Avertissement** : L'objectif de cette section est de fournir au lecteur les notions nécessaires à la compréhension du présent document. Cette section est loin d'être une description complète de la syntaxe XML. Notons que la même remarque est à faire pour la section suivante. Pour une description complète de la syntaxe XML, consultez [4].

3.1. Le prologue

Le prologue est optionnel dans un document XML. Il est utilisé, entre autre, pour :

- signaler le début des données XML;
- décrire la méthode d'encodage des caractères;
- fournir des options de configuration au *parser*[1] ou à l'application sous la forme de commentaires (entourés de `<!--` et de `-->`) ou d'instructions de traitement (entourés de `<?` et de `?>`);
- associer un fichier de contraintes (ou DTD) que le document doit vérifier (cf. Fig. 1.2, troisième ligne du prologue).

3.2. Le corps

Le corps est constitué d'un ou plusieurs **éléments** organisés selon une hiérarchie de type arborescente.
Chaque élément est délimité par des balises obligatoires appelées balise ouvrante et balise fermante. Elles sont constituées du **nom du type d'élément** entouré des signes < et >. Dans le cas de la balise fermante, le signe < est suivi du signe `/`. Entre ces deux balises figure le **contenu** de l'élément. Ce contenu peut être constitué :

- de données textuelles;
- d'autres éléments appelés dans ce cas, **éléments fils** ou **sous-éléments**;
- d'autres balisages tels que des commentaires ou des instructions de traitement.

Les noms des types d'élément doivent obligatoirement commencer par une lettre, un _ ou un : suivi de 0 ou plusieurs lettres, chiffres, `.`, `-`, _ ou `:`.

Dans certain cas, on peut vouloir définir des éléments de contenu vide. Il existe une notation plus concise dans ce cas (cf. Fig. 1.3).

```
<Bulletin id="P233Y667MF606"></Bulletin>

<Bulletin id="P233Y667MF606"/>
```

Figure 1.3- Deux notations valides permettent de représenter des éléments de contenu vide.

1. Le traitement d'un document XML nécessite le recours à un *parser*, c'est-à-dire à un programme d'analyse lexicale du document. Dans le cadre de XML, on distingue deux types de parser : le parser "non-validant" (*non-validating parser*) qui vérifie si le document XML est bien formé et le parser "validant" (*validating parser*) qui, en plus, vérifie si le document XML est conforme à un DTD référencé.

Pour attacher certaines informations à un élément (telle que l'unité de mesure de la température, cf. Fig. 1.2), deux alternatives sont possibles :

- définir un sous-élément de cet élément;

```
<temp>
   <unit>Celsius</unit>
   12.3
</temp>
```

- définir un **attribut** attaché à cet élément.

```
<temp unit="Celsius">12.3</temp>
```

Le choix d'une des deux méthodes revient au concepteur du document XML.

Un attribut est une paire nom-valeur pour laquelle la valeur figure entre guillemets (") ou entre apostrophes ('). Un attribut doit figurer à l'intérieur de la balise ouvrante de l'élément auquel il se rapporte ou à l'intérieur de la balise unique si l'élément est vide (tel l'élément `Bulletin` de l'exemple ci-dessus). Un élément ne peut se voir attacher deux attributs de même nom.

3.3. L'épilogue

L'épilogue est une partie optionnelle du document XML qui peut inclure des commentaires ou des instructions de traitement.

4. Définitions de Type de Document (DTD)

Les documents XML sont structurés par des balises. Il est possible de définir des contraintes sur ces balises. On peut, par exemple, fixer le nom des balises permises, leur ordre d'apparition, les attributs qui leur sont attachés. L'ensemble de ces contraintes porte le nom de **Définitions de Type de Document** (ou DTD).

Ce mécanisme s'avère particulièrement adéquat lorsqu'il s'agit, par exemple, de standardiser les messages échangés entre diverses sociétés d'un même domaine d'application. Il suffit de définir un DTD décrivant exactement les documents XML admis. Chaque société peut alors développer une application de traitement automatique de ces messages. De plus, les DTD sont écrits dans une grammaire formelle. Ils constituent donc une excellente documentation des structures de données utilisées dans un domaine bien précis.

Il existe deux types de DTD que l'on peut associer à un document XML :

- interne : l'ensemble des contraintes doit figurer dans le prologue du fichier XML entouré de crochets;
- externe : l'ensemble des contraintes constitue un fichier (*.dtd) qu'il faut référencer dans le prologue du fichier XML, comme le montre l'exemple de la figure 1.2.

Nous l'avons vu à la section 3., un document XML est constitué d'éléments et d'attributs. Un DTD doit donc être capable de définir les éléments d'un document XML qui sont admis, les attributs qui peuvent leur être assignés et les associations entre ces éléments. Le mécanisme DTD utilise deux mots-clefs pour ce faire : `ELEMENT` et `ATTLIST`.

4.1. Définitions de contraintes sur les éléments

Le mot-clef `ELEMENT` permet de déclarer un type d'élément.
Par exemple :

```
<!ELEMENT Observations (loc, date, temp, hygro, nebulo, anemo, pluvio)>
```

indique que le document XML admet des éléments de type `Observations` dont le contenu est constitué successivement d'un élément de type `loc`, d'un élément de type `date`, d'un élément de type `temp`, ... , et enfin, d'un élément de type `pluvio`.

Plus précisément, la déclaration d'un type d'élément est constituée successivement :

- des caractères `<!ELEMENT`;
- du nom du type d'élément (qui doit suivre la règle des noms énoncée à la section 3.);
- du **type de contenu**, décrit ci-dessous;
- du caractère `>`.

Le type de contenu peut être :

- EMPTY : les éléments de ce type n'ont pas de contenu (ni textuel, ni d'élément fils), cela n'empêche pas qu'ils peuvent avoir un attribut (cf. Fig. 1.3). La déclaration d'un type d'élément qui a ce type de contenu se fait en utilisant le mot-clef EMPTY, comme la montre la figure 1.4. L'extrait de document XML représenté en figure 1.3 se conforme à cette déclaration.

```
<!ELEMENT Bulletins EMPTY>
```

Figure 1.4- Exemple de déclaration d'un élément de type EMPTY.

- ANY : le contenu des éléments de ce type n'est pas contraint. C'est le mot-clef ANY qui permet de déclarer un tel type d'élement;

- MIXED : ici, les éléments contiennent d'autres éléments ou des données textuelles (cf. Fig. 1.5). La déclaration des éléments de ce type est plus complexe que dans les deux cas précédents. Elle fait l'objet du paragraphe ci-dessous.

```
<livre isbn="4587-8569-785">
    <Auteur>Ken Follet</Auteur>
    <Titre>Les pilliers de la terre</Titre>
    Roman historique
</livre>
```

Figure 1.5- Exemple d'élément dont le contenu est de type MIXED.

- ELEMENT : dans ce cas, les éléments ne contiennent que des éléments, pas de données textuelles, comme l'illustre la figure 1.6. La déclaration de tels types d'élément ressemble sous de nombreux points à celle des types d'élements de type de contenu MIXED. Elle est décrite ci-dessous.

```
<livre isbn="4587-8569-785">
    <Auteur>Ken Follet</Auteur>
    <Titre>Les pilliers de la terre</Titre>
    <Commentaires>Roman historique</Commentaires>
</livre>
```

Figure 1.6- Exemple d'élément dont le contenu est de type ELEMENT.

Les types de contenu MIXED et ELEMENT ne se bornent pas à imposer que le contenu d'un élément soit uniquement composé d'autres éléments ou soit également constitué de données textuelles. Il précise quels sont les types d'éléments qui peuvent constituer l'élément contraint, dans quel ordre ils peuvent apparaître, quel nombre d'occurences d'un même type d'élément peut apparaître. Par exemple, pour le cas de la figure 1.6, le DTD pourrait contenir une contrainte telle que

```
<!ELEMENT livre (Auteur,Titre,Commentaires)>
```

indiquant que chaque élément de type `livre` doit être constitué d'un élément de type

Auteur suivi d'un élément de type Titre et enfin, d'un élément de type Commentaires.

Le mot-clef #PCDATA peut être utilisé à la place d'un type d'élément pour représenter des données textuelles (uniquement dans le cas de contenu MIXED). Par exemple, l'élément de la figure 1.5 se conforme à la contrainte suivante :

```
<!ELEMENT livre (Auteur,Titre,#PCDATA)>
```

Lorsque le type de contenu d'un type d'élément est MIXED et que son contenu est exclusivement constitué du mot-clef #PCDATA, on parle d'un type de contenu #PCDATA.

Les types d'éléments entre parenthèses peuvent être suivi d'un opérateur de cardinalité : ?,+ ou * indiquant que l'élément contraint peut contenir respectivement 0-1, 1-N, 0-N éléments de ce type. Ainsi, la contrainte

```
<!ELEMENT livre (Auteur+,Titre,Commentaires?)>
```

indique que tout élément de type livre doit contenir au moins un élément de type Auteur, un et un seul élément de type Titre et, éventuellement un élément de type Commentaires.

Le caractère | séparant deux types d'éléments (éventuellement soumis à l'action d'un opérateur de cardinalité) signale qu'un choix exclusif entre ces deux types d'éléments doit être fait. Ainsi, la contrainte

```
<!ELEMENT livre ((Auteur+,Titre)|ISBN)>
```

est vérifiée dans les deux cas suivants :

```
<livre>
   <Auteur>Didier Martin</Auteur>
   <Auteur>Mark Birbeck</Auteur>
   <Titre>Professionnal XML</Titre>
</livre>
```

```
<livre>
   <ISBN>1-861003-11-0</ISBN>
</livre>
```

4.2. Définitions de contraintes sur les attributs

Le mot-clef ATTLIST permet de décrire les attributs admis dans le document.
La contrainte

```
<!ATTLIST Bulletins id ID #REQUIRED>
```

indique que le type d'élément Bulletins admet un attribut de nom id qui est de **type** ID et de **type de contenu** #REQUIRED.

Il y a plusieurs types d'attribut. Citons parmi eux :

- `ID` : chaque valeur de cet attribut doit être unique au sein de l'ensemble de toutes les valeurs prises par tous les attributs de type `ID` que comporte le document. Puisque la valeur prise par un attribut de type `ID` est unique dans le document, ce dernier identifie sans équivoque **un** élément du document. Notons que la portée d'un attribut `ID` est plus large que celle des attributs identifiants dans le cadre des bases de données.
- `IDREF` : un attribut de ce type doit prendre ses valeurs dans l'ensemble des valeurs prises par les attributs de type `ID` du document. Puisque cette valeur est prise par un seul attribut de type `ID`, l'attribut `IDREF` référence un seul élément du document.
  Ici aussi, la portée de cet attribut est quelque peu différente de celle rencontrée dans le contexte des bases de données, à travers le concept de clef référentielle. En effet, si un élément a un attribut de type `IDREF`, les valeurs prises par cet attribut ne désignent pas obligatoirement des éléments de même type. En revanche, dans le contexte des bases de données, une clef référentielle désigne toujours des instances d'un même type d'entité.
- `IDREFS` : un attribut de type `IDREFS` peut prendre plusieurs valeurs. Mais, chacune de ces valeurs doit être choisie parmi l'ensemble des valeurs prises par tous les attributs de type `ID` du document. Les valeurs prises par un même attribut de type `IDREFS` peuvent désigner des éléments de type différent.

Ce mécanisme des `ID/IDREFS` permet d'exprimer des liens inter-éléments. Il faut toutefois veiller à ne pas associer à ces derniers la même sémantique que dans le cas d'une base de données.

Il y a plusieurs types de contenu d'attribut. Parmi eux, citons :

- `#REQUIRED` : l'attribut doit apparaître pour chaque élément de ce type.
- `#IMPLIED` : la présence de l'attribut dans un élément de ce type est optionnelle;
- `#FIXED` : l'attribut a une valeur par défaut. Qu'il apparaisse explicitement pour chaque élément ou pas, sa valeur est toujours la valeur par défaut. Pour déclarer un attribut de ce type de contenu, on ajoute au mot-clef `#FIXED`, la valeur par défaut entre guillemets.
- *default* : ici, l'attribut peut ne pas apparaître. Dans ce cas, sa valeur supposée est la valeur par défaut. Si il apparaît, il peut contenir une autre valeur que la valeur par défaut. La déclaration d'un attribut de ce type de contenu consiste simplement faire suivre le reste de la déclaration, de la valeur par défaut entre guillemets.

4.3. Exemple récapitulatif

Le DTD suivant :

```
<!ELEMENT livre (Titre)>                          (1)
<!ELEMENT Titre (#PCDATA)>                        (2)

<!ATTLIST livre Auteur IDREF #REQUIRED            (3)
                ISBN   ID    #REQUIRED>           (4)

<!ELEMENT auteur (Nom, Prenom)>                   (5)
<!ELEMENT Nom    (#PCDATA)>                       (6)
<!ELEMENT Prenom (#PCDATA)>                       (7)

<!ATTLIST auteur numero ID #REQUIRED>             (8)
```

est vérifié dans le cas suivant :

```
<auteur numero="1">
   <Nom>Follet</Nom>
   <Prenom>Ken</Prenom>
</auteur>

<livre ISBN="254189637" Auteur="1">
   <Titre>Les Pilliers de la terre</Titre>
</livre>
```

En effet, passons en revue les différents constituants du document XML et observons qu'ils se conforment aux déclarations du DTD :

- le document XML ne contient que des éléments de type `auteur`, `Nom`, `Prenom`, `livre` et `Titre`. Tous ces types d'élément ont été déclarés dans le DTD;
- l'élément de type `auteur` a un contenu de type `ELEMENT` constitué d'un élément de type `Nom` suivi d'un autre de type `Prenom`. C'est exactement ce qu'impose la contrainte `(5)` du DTD. Les éléments `Nom` et `Prenom` ont un contenu purement textuel. C'est ce qu'imposent les contraintes `(6)` et `(7)` du DTD;
- une analyse similaire à la précédente peut être menée pour la deuxième partie du document XML pour l'élément de type `livre`;
- l'élément de type `auteur` a un attribut de nom `numero`, comme exigé par la contrainte `(8)` du DTD. Cette contrainte indique que l'attribut `numero` est de type `ID`. Cela signifie qu'il doit prendre une valeur qu'aucun autre attribut de type `ID` ne peut prendre. Nous pouvons observer que le seul autre attribut de type `ID` est l'attribut `ISBN` attaché au type d'élément `livre`. Puisque ce dernier ne prend pas la valeur "`1`", l'attribut `numero` a une valeur qui n'est prise par aucun autre attribut de type `ID` présent dans le document et donc, vérifie bien la contrainte `(8)` du DTD;
- une analyse similaire peut mener à la conclusion que la contrainte `(4)` du DTD est vérifiée par le document ;

- l'élément de type `livre` a un attribut de nom `Auteur`, comme exigé par la contrainte `(3)` du DTD. Cette contrainte indique que l'attribut `Auteur` est de type `IDREF`. Cela signifie que chaque valeur qu'il prend doit également être prise par un attribut de type `ID` dans le même document. Ceci est vérifié car la valeur prise par l'attribut `Auteur` est celle de l'attribut `numero` attaché à l'élément de type `auteur`. Ce mécanisme, plus connu sous le nom `ID/IDREF` permet de lier des éléments entre eux, tel un livre avec son auteur.

CHAPITRE 2

Application de production de document(s) XML

De nombreuses situations nécessitent la génération de document(s) XML à partir de données stockées dans une base de données. On parle dans ce cas de **migration** de données d'une base de données vers un ou plusieurs document(s) XML ou de **production** de document(s) XML à partir d'une base de données. Nous avons évoqué quelques-unes de ces situations dans le chapitre introductif.

Dans ce chapitre, nous allons nous intéresser à la construction d'une application permettant d'automatiser la migration de données d'une base de données vers un ou plusieurs documents XML.

Cette application, de conception simple, est construite pour opérer sur une base de données et une structure de documents XML particulières. Nous verrons dans le chapitre suivant comment automatiser la construction d'une telle application afin de pouvoir supporter n'importe quelle structure de base de données et de documents XML.

Dans ce chapitre, nous donnerons également des pistes de modélisation de données (issues d'une base de données) en XML.

1. Présentation générale

La migration de données d'une base de données vers un ou plusieurs document(s) XML peut être automatisée par le recours à une application spécifique que nous appelerons dorénavant **migrateur** ou **générateur de documents XML**.

Les figures 2.1 et 2.2 illustrent une base de données et un document XML susceptibles d'être les produits d'entrée et de sortie d'un migrateur. La base de données enregistre les données relatives aux activités d'une chaîne de magasins semblables à Belgique Loisirs, par exemple.

CLIENT

NCli	Nom	Prénom	Adresse	Téléphone	NCompte
1	Dupont	Jean	3, Rue du Rempart 4000 MONS	0475/85 69 32	001-0458625-85
2	Pierret	Martin	5, Grand Rue 5000 NAMUR	081/85 63 52	375-8569852-89

LIVRE

ISBN	Titre	Auteurs	Prix	Année	Editeur
1-861003-11-0	Les pilliers de la terre	Follet	495	1992	Pocket
5-785456-07-5	Professional XML	Martin+Birbeck+Kay	2185	1999	Wrox

ACHAT

NCli	ISBN	Date	Ville
1	1-861003-11-0	02-05-1993	MONS
1	5-785456-07-5	05-07-2000	MONS
2	1-861003-11-0	16-07-1995	NAMUR

Figure 2.1- Base de données "Client-Livre-Achat".

```
<Client-Livre-Achat>
   <CLIENT NCli="1">
      <Nom>Dupont</Nom>
      <Prenom>Jean</Prenom>
      <Adresse>3,Rue du Rempart 4000 MONS</Adresse>
      <Telephone>0475/85 69 32</Telephone>
      <NCompte>001-0458625-85</NCompte>
   </CLIENT>
   <CLIENT NCli="2">
      <Nom>Pierret</Nom>
      <Prenom>Martin</Prenom>
      <Adresse>5,Grand Rue 5000 NAMUR</Adresse>
      <Telephone>081/85 63 52</Telephone>
      <NCompte>375-8569852-89</NCompte>
   </CLIENT>
   <LIVRE ISBN="1-861003-11-0">
      <Titre>Les pilliers de la terre</Titre>
      <Auteur>Follet</Auteur>
      <Prix>495</Prix>
      <Annee>1992</Annee>
      <Editeur>Pocket</Editeur>
   </LIVRE>
   <LIVRE ISBN="5-785456-07-5">
      <Titre>Professional XML</Titre>
      <Auteur>Martin</Auteur>
      <Auteur>Birbeck</Auteur>
      <Auteur>Kay</Auteur>
      <Prix>2185</Prix>
      <Annee>1999</Annee>
      <Editeur>Wrox</Editeur>
   </LIVRE>
   <ACHAT NCli="1" ISBN="1-861003-11-0">
      <Date>02-05-1993</Date>
      <Ville>MONS</Ville>
   </ACHAT>
   <ACHAT NCli="1" ISBN="5-785456-07-5">
      <Date>05-07-2000</Date>
      <Ville>MONS</Ville>
   </ACHAT>
   <ACHAT NCli="2" ISBN="1-861003-11-0">
      <Date>16-07-1995</Date>
      <Ville>NAMUR</Ville>
   </ACHAT>
</Client-Livre-Achat>
```

Figure 2.2- Document XML construit à partir de la base de données "Client-Livre-Achat" de la figure 2.1.

Il s'agit donc de construire une application qui génère automatiquement un ou plusieurs document(s) XML à partir des données d'une base de données. Plus précisément, cette application doit assurer :

- l'**extraction** des données de la base de données :
  Dans le cas illustré ci-dessus, l'extraction des données porte sur la totalité des données de la base de données. Lorsque les documents XML ne doivent pas contenir toutes les données de la base de données, l'extraction peut être plus sélective.
- la **conversion** et le **formatage** des données :
  Le terme "formatage des données" désigne la construction du ou des document(s) XML à partir des données extraites de la base de données (après conversion) et d'un balisage adéquat.
  Dans notre exemple, la conversion des données se réduit à la désagrégation des données correspondant à l'attribut `Auteurs` du type d'entité `LIVRE`. Les autres données n'ont pas besoin d'être transformées.

Un générateur de documents XML, tel que nous l'envisageons, est spécifique à une base de données et à une structure de document XML. Sa construction nécessite donc de connaître:

- la structure de la base de données;
- les modes d'accès à la base de données qui dépendent directement du DBMS[1] supportant la base de données;
- la structure du ou des document(s) XML à produire : elle doit être, par exemple, établie par un analyste (voir section 3.);
- les correspondances entre la structure de la base de données et celle du ou des document(s) XML. Dans notre exemple, les correspondances sont triviales à établir, à l'exception de l'attribut `Auteurs` du type d'entité `LIVRE` qui donne naissance à un ou plusieurs éléments de type `Auteur`, fils d'un élément de type `Livre`.
  Les correspondances structurelles permettent de déduire les règles de conversion à appliquer aux données extraites de la base de données.

1. DBMS (**D**ata**B**ase **M**anagement **S**ystem) est une collection de programmes qui permettent le stockage, la modification et l'extraction de données d'une base de données.

2. Architecture du migrateur

Le migrateur a deux tâches à effectuer :

- l'extraction des données;
- la conversion et le formatage des données.

Ces deux tâches sont à exécuter de manière séquentielle. Il semble donc naturel de les isoler dans des modules spécifiques et d'enchaîner l'exécution de ces deux modules. La figure 2.3 illustre une architecture construite sur ce principe.

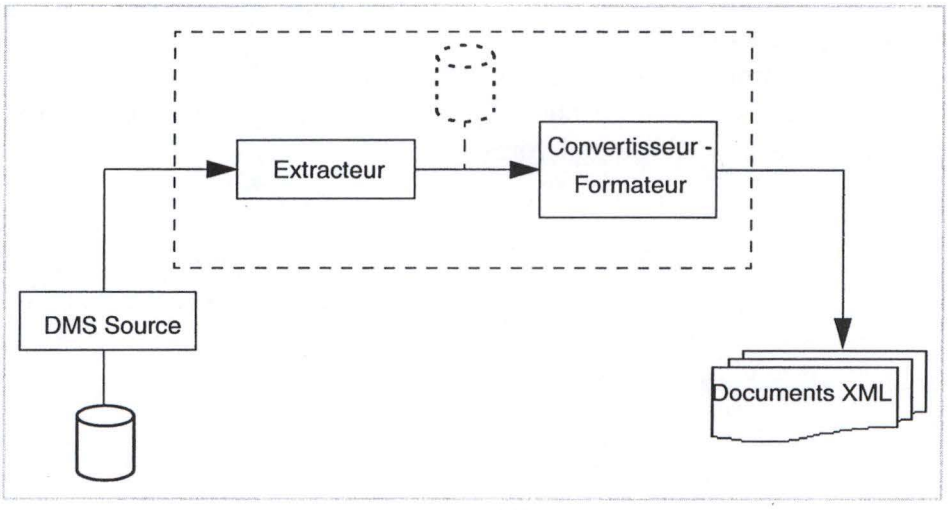


Figure 2.3- Architecture d'un migrateur.

On peut observer, sur la figure ci-dessus, un cylindre en pointillé entre les deux modules. Il symbolise un stockage éventuel des données entre les deux modules.

2.1. L'extracteur

Comme on peut s'y attendre, ce module est chargé d'extraire les données de la base de données.

Il y a différentes façons de construire la requête d'extraction. Par exemple, si on travaille dans un cadre relationnel, il faut déterminer le nombre et la forme des requêtes à exécuter. Ces choix dépendent de la structure du ou des document(s) XML à produire. Ainsi, la figure 2.4 montre que le recours à des requêtes indépendantes les unes des autres est suffisant dans certains cas, alors que dans d'autres, une autre stratégie peut sembler plus appropriée (cf. Fig. 2.5).

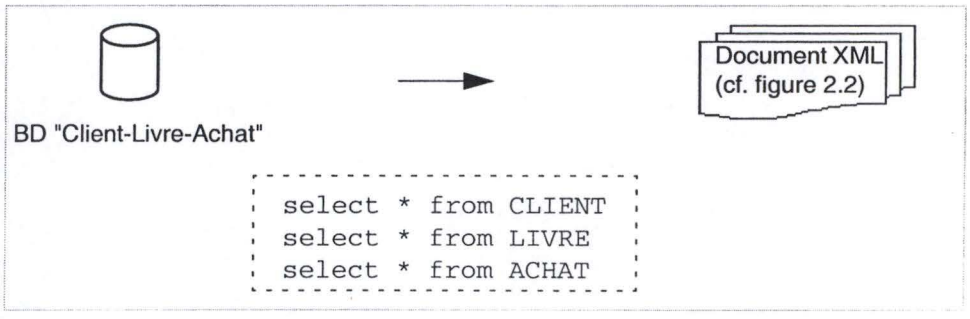


Figure 2.4- Requêtes portant sur une seule table

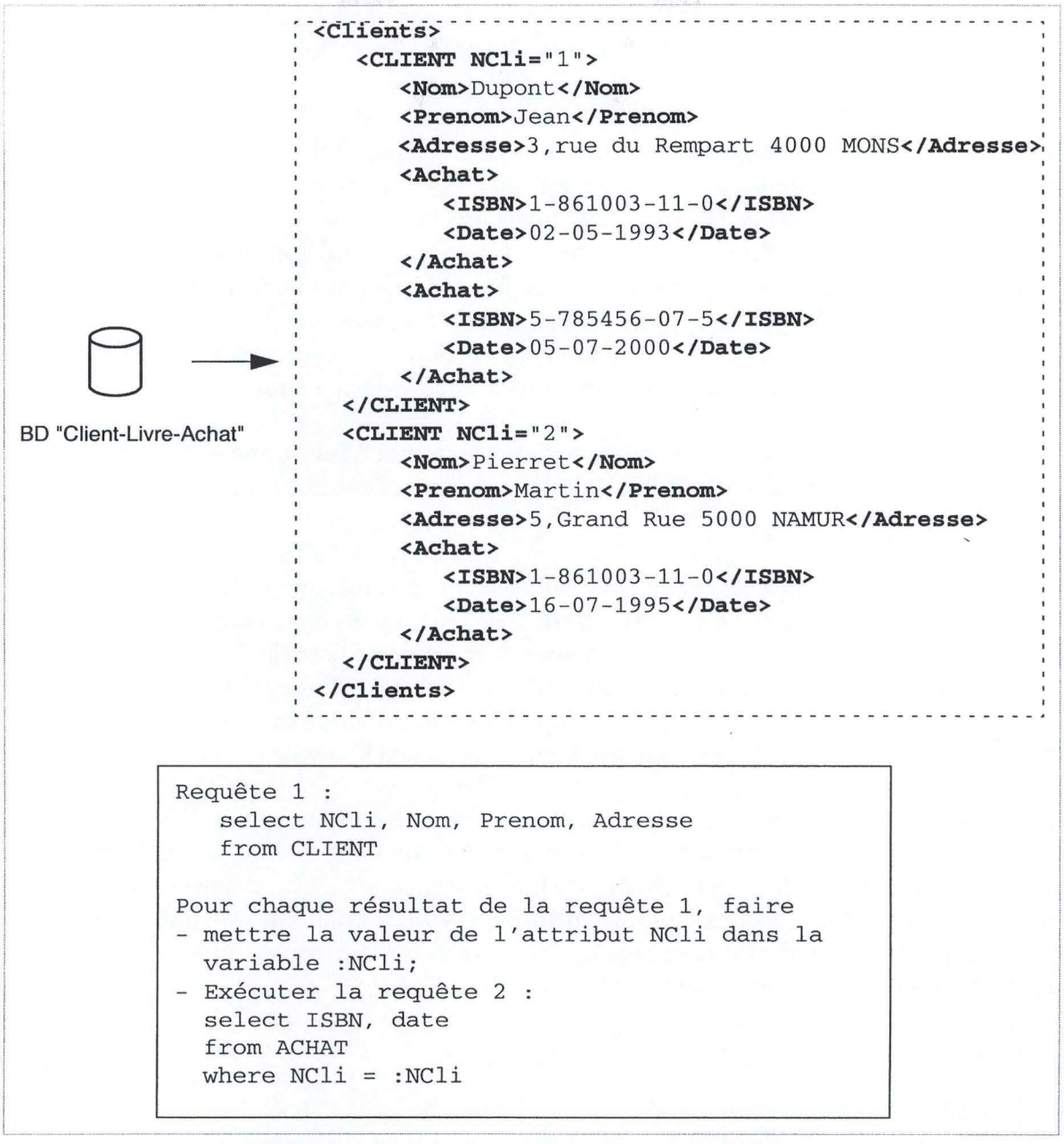


Figure 2.5- Exemple pour lequel le recours à deux requêtes où la seconde est paramétrisée par le résultat de la première est nécessaire.

Evidemment, quand cela est possible, il est préférable d'exploiter au maximum les capacités du DBMS pour ordonner les données (par exemple, pour regrouper les achats d'un même client, cf. Fig. 2.5). Sinon, ce travail reste à faire "à la main".

2.2. Le convertisseur-formateur

A ce stade, les données extraites de la base de données n'ont subi aucun traitement (si ce n'est un éventuel ordonnancement). Le rôle du convertisseur-formateur est donc de les **transformer** si besoin est.

Les transformations des données sont déduites des correspondances entre les structures de la base de données et celles du ou des document(s) XML à produire.
Par exemple, dans le cas de la production du document XML de la figure 2.2, à partir de la base de données de la figure 2.1, on observe que l'attribut `Auteurs` du type d'entité `LIVRE` est la concaténation de plusieurs attributs. Par contre, dans le document XML à produire, les éléments de type `Auteur` ne contiennent qu'une seule valeur. Il y a donc lieu de mener une transformation de désagrégation sur les données correspondantes à l'attribut `Auteurs`. C'est le convertisseur-formateur qui se charge de cette transformation.

Enfin, le convertisseur-formateur est chargé de **"formater" les données** c'est-à-dire de construire le ou les document(s) XML à partir des données extraites et d'un balisage adéquat.

Notons que la complexité du convertisseur-formateur est directement liée à celle des correspondances structurelles entre la base de données et le ou les document(s) XML. Ainsi, dans le cas où on souhaite développer un migrateur entre une base de données et un DTD public (ou *vocabulary*[1]) servant de standard d'échange, il y a de fortes chances que le travail de transformation de données sera important car la structure du document a été élaborée de manière complètement indépendante de celle de la base de données.

Par contre, si on souhaite faire migrer une base de données sur un autre DMS, une façon de procéder consiste à extraire de la base de données initiale la totalité de ses données et à les stocker dans des documents XML; ensuite, la nouvelle base de données est remplie à partir de ces mêmes documents. Ces documents XML auront une structure proche de celle de la base de données d'origine et les transformations à effectuer sur les données seront très légères, voire inexistantes.

1. Dans de nombreux domaines (médical, légal, scientifique), une structure de documents XML (souvent appelée *vocabulary*) sous la forme d'un DTD a été fixée afin de standardiser les échanges de données. Il reste à chaque entreprise le soin de développer les applications générant des documents XML conformes à cette structure commune.

2.3. Architecture améliorée

L'objectif de ce travail est d'automatiser autant que possible la construction du migrateur étant donnés une structure de base de données, une structure de document XML, les correspondances entre ces deux structures et les fonctions d'accès au DMS.

Cependant, si les fonctions d'accès et de consultation de chaque DMS sont différentes, il est difficile d'automatiser la construction de l'extracteur qui les utilise.
Une solution à ce problème consiste à isoler ces fonctions dans un module indépendant pour lequel on définit une interface standard. L'extracteur utilise cette interface standard dans tous les cas et il ne reste qu'à écrire ce module pour chaque DMS ou groupe de DMS en respect de l'interface définie. Nous avons baptisé ce module **module d'accès au DMS**.

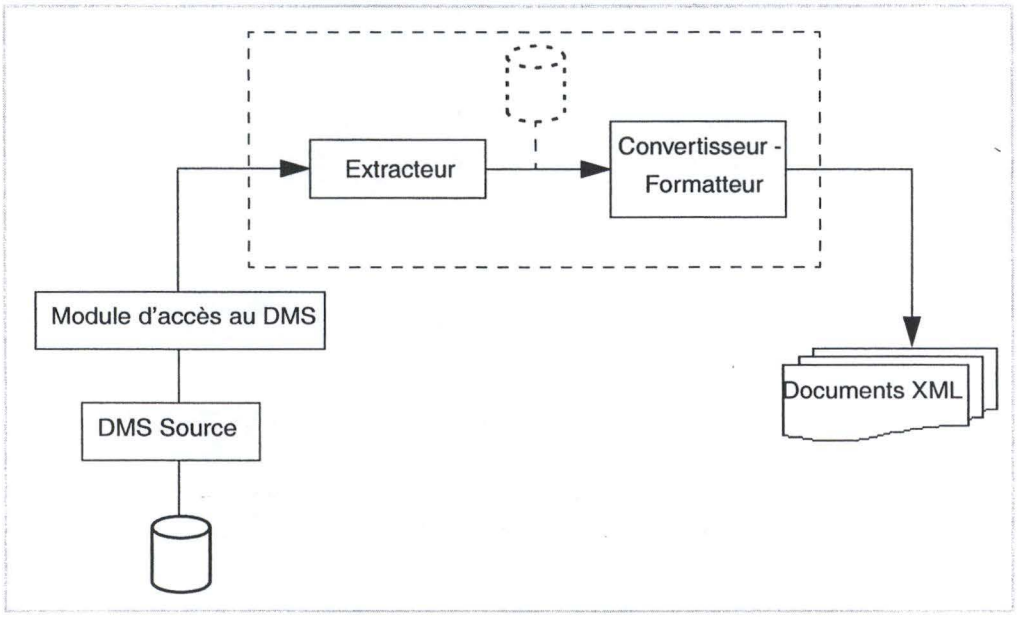


Figure 2.6- Architecture améliorée d'un migrateur.

Cette façon de procéder permet de rendre le migrateur indépendant des fonctions d'accès au DMS.

2.4. Remarque

Nous faisons l'hypothèse que la base de données est dans un état cohérent. Si tel n'était pas le cas, il faudrait effectuer une étape de nettoyage des données soit avant l'extraction des données, soit après la production du/des document(s) XML. Cette étape constitue à elle-seule un problème extrêmement long et complexe, qui dépasse la portée de notre travail.

3. Modélisation de données et XML

Dans cette section, nous allons donner un aperçu des différents choix qui s'offrent à l'analyste lors de la conception de la structure du ou des document(s) XML. Notons que :

- cette étape de conception des structures du ou des document(s) XML peut ne pas être nécessaire. En effet, le but de la migration peut être simplement de se conformer à un *vocabulary* fixé et dans ce cas, la structure du ou des document(s) XML est déjà établie;
- la modélisation des données présentée dans cette section se base sur la structure de la base de données. Nous supposons donc que celle-ci a été conçue judicieusement;

La structure[1] d'une base de données peut être représentée sous la forme d'un graphe orienté où chaque type d'entité correspond à un noeud et chaque contrainte référentielle correspond à un arc orienté. Nous avons choisi de l'orienter dans le sens inverse vis-à-vis de la manière traditionnelle de représentation des contraintes référentielles. Ainsi, la structure de la base de données "Client-Livre-Achat" (cf. Fig. 2.1) correspond au graphe suivant :

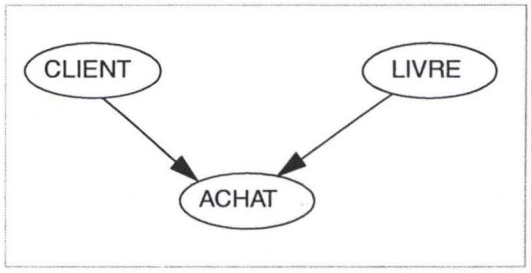


Figure 2.7- Représentation sous forme de graphe de la structure de la base de données "Client-Livre-Achat".

En revanche, la structure d'un document XML (décrite par son DTD) est de nature arborescente.
En guise d'illustration, les figures 2.8 et 2.9 représentent respectivement le DTD associé au document XML de la figure 2.2 et l'arbre représentant ce DTD. Ce dernier est construit de la manière suivante : le noeud racine correspond au type d'élément racine (`Client-Livre-Achat`, dans notre exemple). Ensuite, on associe un noeud par type d'élément apparaissant dans le DTD. Ces noeuds sont organisés hiérarchiquement en suivant exactement le même ordre d'imbrication qu'entre les types d'élément dans le DTD.

1. Dans cette section, nous considérons que la structure de la base de données est décrite au niveau logique.

```
<!ELEMENT Client-Livre-Achat(CLIENT*, LIVRE*, ACHAT*)>


<!ELEMENT CLIENT(Nom, Prenom, Adresse, Telephone, NCompte)>
<!ELEMENT Nom (#PCDATA)>
<!ELEMENT Prenom (#PCDATA)>
<!ELEMENT Adresse (#PCDATA)>
<!ELEMENT Telephone (#PCDATA)>
<!ELEMENT NCompte (#PCDATA)>
<!ATTLIST CLIENT NCli ID #REQUIRED>

<!ELEMENT LIVRE(Titre, Auteur+, Prix, Annee, Editeur)>
<!ELEMENT Titre (#PCDATA)>
<!ELEMENT Auteur (#PCDATA)>
<!ELEMENT Prix (#PCDATA)>
<!ELEMENT Annee (#PCDATA)>
<!ELEMENT Editeur (#PCDATA)>
<!ATTLIST LIVRE ISBN ID #REQUIRED>

<!ELEMENT ACHAT(Date, Ville)>
<!ELEMENT Date (#PCDATA)>
<!ELEMENT Ville (#PCDATA)>
<!ATTLIST ACHAT NCli IDREF #REQUIRED
                ISBN IDREF #REQUIRED>
```

Figure 2.8- DTD correspondant au document XML de la figure 2.2.

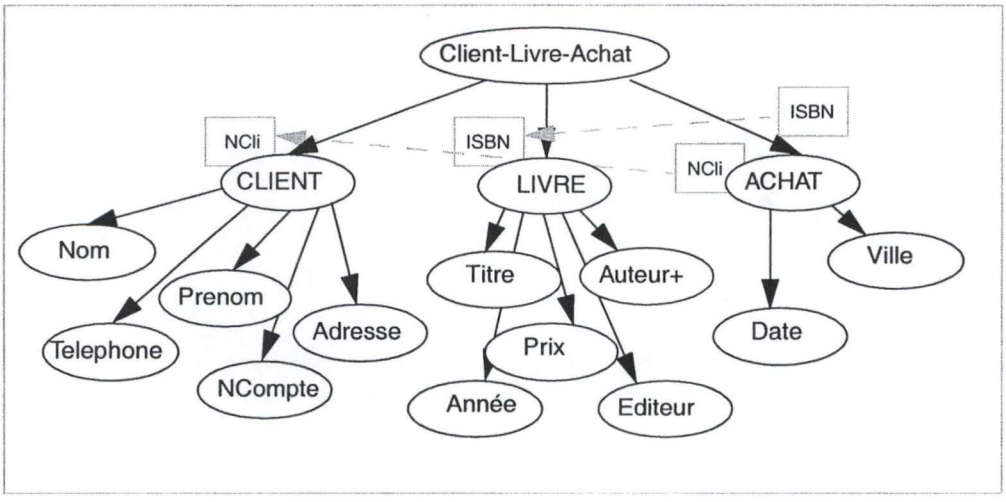


Figure 2.9- Représentation en arbre du DTD de la figure 2.8.

Vu de cette manière, concevoir la structure d'un document XML destiné à contenir les données d'une base de données revient à construire un arbre (représentant le DTD) à partir d'un graphe (représentant la structure de la base de données). Or, il existe de multiples façons de dériver un arbre d'un graphe, parmi lesquelles il faut choisir. Ce choix peut être orienté par l'usage que l'on destine au document XML généré[1].

3.1. Modélisation des types d'entité d'un schéma logique d'une base de données

Lors de la conception d'une base de données, on regroupe au sein d'un même type d'entité toutes les informations concernant une même entité de la vie réelle telle un client, un livre, ... De même, dans le contexte de XML, les types d'élément sont souvent définis comme représentant un objet de la vie réelle ou une propriété de cet objet. Un type d'entité d'une base de données est donc naturellement modélisé par un type d'élément. Une fois qu'on a associé un type d'élément à chaque type d'entité, comment agencer ces différents types d'éléments entre eux ?

3.1.1.Structure plate

Une première solution consiste à procéder de la manière suivante :

- on crée un type d'élément racine artificiel qui porte, par exemple, le nom de la base de données;
- chaque type d'élément modélisant un type d'entité est un descendant direct du type d'élément racine. De plus, il porte le nom du type d'entité qui lui correspond. L'ordre de ces types d'élément est à priori sans importance.

Cette solution est appelée **structure plate** car la profondeur de l'arbre résultant est de deux.

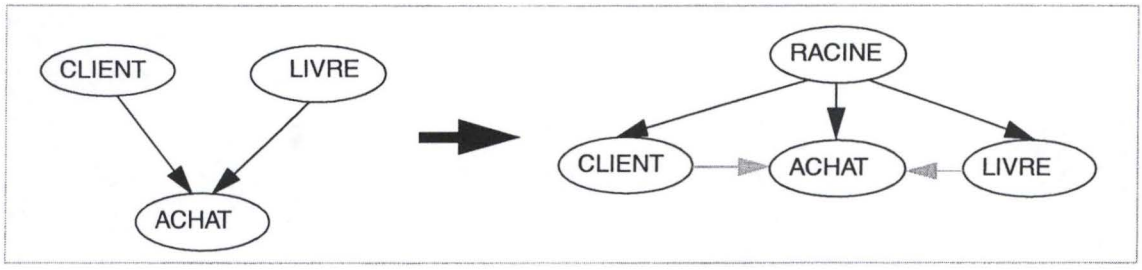


Figure 2.10- Agencement des types d'élément correspondant aux tables de la base de données en structure plate.

Cela correspond exactement à la façon dont on a construit le DTD de la figure 2.9 à partir de la base de données représentée à la figure 2.7.

1. Notons que cet avis n'est pas partagé par tous les auteurs dans ce domaine. Ainsi, dans [5], les auteurs estiment que les choix de modélisation des données ne doivent pas être influencés par l'usage pour lequel on construit ce document XML. Les choix de modélisation ne doivent prendre en compte que la nature de l'information à structurer. La raison de ce choix est que l'usage d'un document peut évoluer. En revanche, le contenu de l'information restera probablement plus stable.

La figure 2.10 illustre cette technique de construction du DTD. Elle montre également que les liens inter-type d'entité (représentés par des flèches grisées cf. Fig. 2.10) ne correspondent pas à des liens hiérarchiques père-fils. Ces liens doivent être pris en charge soit

- par le mécanisme des attributs ID/IDREF : ce mécanisme ne permet pas de représenter toutes les contraintes référentielles (par exemple, les contraintes référentielles portant sur un groupe de plusieurs attributs ne peuvent pas être exprimées avec ce mécanisme).
- par une application extérieure. Dans ce cas, les attributs correspondant à une contrainte référentielle sont traduits en un type d'élément ou un attribut classique de XML et une application extérieure se charge de la gestion de la contrainte.

Notons également que dans certains cas, les contraintes référentielles n'ont pas besoin d'être traduites en XML. Par exemple, si on veut réaliser la migration d'une base de données vers une autre base de données de même structure mais gérée par un DBMS différent. Une manière de procéder consiste à extraire toutes les données de la base de données initiale et à les stocker dans des documents XML et ensuite de décharger ces documents XML dans la nouvelle base de données. Puisque nous avons des garanties sur les données de la base de données initiale (elle est supposée être dans un état cohérent) et qu'il y a un contrôle à l'entrée de la base de données cible (les contraintes sont vérifiées par le DBMS), il n'est pas nécessaire d'avoir d'autres garanties. Aussi, les contraintes référentielles n'ont pas besoin d'être traduites dans le document XML.

3.1.2.Structure hiérarchisée

Une deuxième solution consiste à extraire un ou plusieurs arbres du graphe initial. Ces "sous-arbres" doivent former ce que nous appelons, une "partition" du graphe, c'est-à-dire qu'ils doivent vérifier les deux conditions suivantes :

- l'union des sommets de tous les sous-arbres doit couvrir l'ensemble des sommets du graphe;
- un sommet du graphe ne peut appartenir à deux sous-arbres.

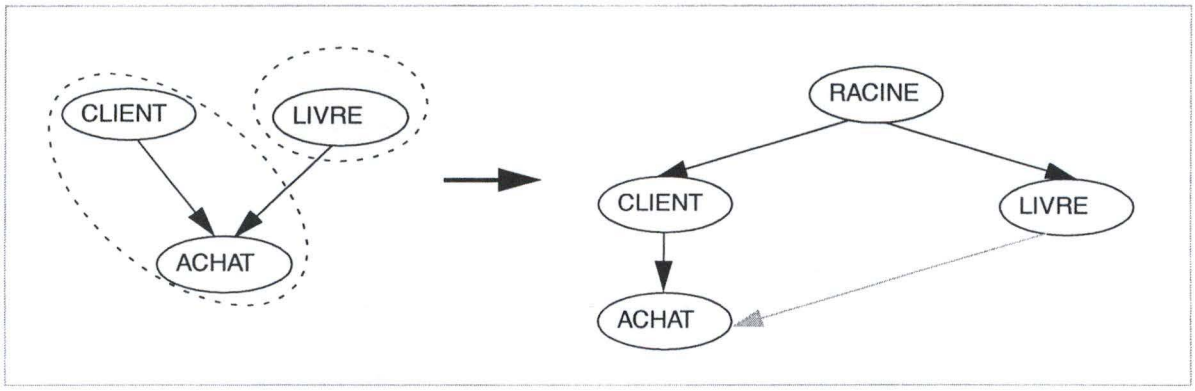
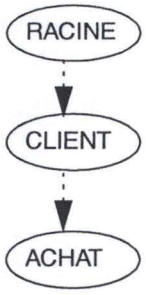


Figure 2.11- Agencement des types d'élément représentant les tables de la base de données en structure hiérarchisée

Il reste à ajouter un noeud racine unique et artificiel en guise de père de tous les noeuds racines des différents sous-arbres pour obtenir l'arbre représentant la structure du document XML.

La figure 2.11 montre que certains liens inter-types d'entité correspondent à des liens hiérarchiques père-fils (ceux représentés par des flèches noires). Ils sont donc gérés par la nature hiérarchique des documents XML. En effet, dans la figure 2.12, l'achat du livre dont le numéro d'ISBN est 1-861003-11-0 en date du 02-05-1993 est relié au client Dupont par le simple fait que l'élément représentant ce livre est un descendant direct de l'élément représentant ce client.
Les autres liens (correspondants aux flèches grisées) doivent être pris en charge par d'autres techniques comme dans le cas de la structure plate (cf. Section 3.1.1. Page 34).

Ce choix de conception s'avère particulièrement adéquat dans des situations comme celle-ci : supposons que nous travaillons sur la base de données représentée en figure 2.1. Supposons également que nous voulions générer un document XML afin d'afficher l'historique complet des achats passés par chaque client. Générer un document XML tel que celui décrit par sa structure et son contenu à la figure 2.12 permet d'accélérer le traitement ultérieur. En effet, les achats de chaque client sont déjà regroupés au sein de l'élément correspondant au client considéré. L'affichage de l'historique des achats pour chaque client en est fortement simplifié.




```
<Clients>
   <CLIENT NCli="1">
      <Nom>Dupont</Nom>
      <Prenom>Jean</Prenom>
      <Adresse>3,rue du Rempart 4000 MONS</Adresse>
      <Achat>
         <ISBN>1-861003-11-0</ISBN>
         <Date>02-05-1993</Date>
      </Achat>
      <Achat>
         <ISBN>5-785456-07-5</ISBN>
         <Date>05-07-2000</Date>
      </Achat>
  </CLIENT>
  <CLIENT NCli="2">
      <Nom>Pierret</Nom>
      <Prenom>Martin</Prenom>
      <Adresse>5,Grand Rue 5000 NAMUR</Adresse>
      <Achat>
         <ISBN>1-861003-11-0</ISBN>
         <Date>16-07-1995</Date>
      </Achat>
  </CLIENT>
</Clients>
```

Figure 2.12- Exemple d'exploitation judicieuse de la structure hiérarchisée.

3.2. Modélisation des attributs d'un type d'entité

Alors qu'à chaque type d'entité d'une base de données, on fait correspondre sans équivoque un type d'élément, le problème est beaucoup plus délicat pour un attribut dans un type d'entité. En effet, à ce dernier peut correspondre soit un type d'élément, descendant direct du type d'élément associé au type d'entité qui le contient, soit un attribut attaché au type d'élément associé à ce même type d'entité. Par exemple, la figure 2.13 montre deux for-

mes de représentation du type d'entité CLIENT de la base de données Client-Livre-Achat de la figure 2.1.

```
<Clients>
   <CLIENT>
      <NCli>1</NCli>
      <Nom>Dupont</Nom>
      <Prenom>Jean</Prenom>
      <Adresse>3,rue du Rempart 4000 MONS</Adresse>
      <Telephone>0475/85 69 32</Telephone>
      <NCompte>001-0458625-85</NCompte>
   </CLIENT>
   [...]
</Clients>
```

OU

```
<Clients>
   <CLIENT
      NCli ="1"
      Nom ="Dupont"
      Prenom ="Jean"
      Adresse ="3,rue du Rempart 4000 MONS"
      Telephone ="0475/85 69 32"
      NCompte="001-0458625-85">
   </CLIENT>
   [...]
</Clients>
```

Figure 2.13- Deux méthodes de représentation d'un attribut dans un type d'entité

Bien que la littérature soit très riche à ce sujet ([5]), le choix de l'une ou l'autre de ces techniques semble le plus souvent relever de l'arbitraire ou d'une question de style.

Une façon de trancher est de considérer un type d'élément comme un *container* de données alors qu'un attribut est plutôt utilisé pour annoter le contenu des éléments. Par exemple, dans la figure 1.2, l'attribut unit attaché au type d'élément temp indique que le contenu de cet élément correspond à une température mesurée en degré Celsius.

Dans certains cas, le choix est facilité par les différences intrinsèques entre les attributs et les types d'éléments :

- les éléments peuvent contenir d'autres éléments aussi bien que du contenu textuel alors que les attributs n'ont qu'un contenu textuel unique. Par exemple, dans un cadre non relationnel, il s'avère donc plus adéquat de représenter un attribut composé par un type d'élément.

- les attributs peuvent prendre des valeurs par défaut ou être soumis à des contraintes (ex : ID, IDREF, ...). Ainsi, si on veut exprimer le caractère identifiant d'un attribut, on le traduira par un attribut de type ID. Notons que la notion d'identifiant d'un type d'entité d'une base de données diffère de la notion d'attribut de type ID, ce qui implique de faire quelques aménagements.

Quelle que soit l'option que l'on a choisie (attribut ou type d'élément), le nom de l'attribut sert souvent à identifier le type d'élément ou l'attribut XML qui lui correspond. Néanmoins, si l'on choisit de traduire les attributs d'un type d'entité par des types d'élément, il faut s'assurer que la base de données ne contiennent pas deux attributs de même nom car il ne peut y avoir deux types d'élément de même nom. Une solution couramment adoptée consiste à préfixer le nom de l'attribut du nom du type d'entité qui le contient tel que le montre la figure 2.14. Cette solution même si théoriquement n'élimine pas tous les risques de collision de noms, donne d'excellents résutats dans la pratique.

```
<Client-Livre-Achat>
   <CLIENT>
      <CLIENT.NCli>1</CLIENT.NCli>
      <CLIENT.Nom>Dupont</CLIENT.Nom>
      <CLIENT.Prenom>Jean</CLIENT.Prenom>
      <CLIENT.Adresse>3,rue du Rempart 4000 MONS</CLIENT.Adresse>
      <CLIENT.Telephone>0475/85 69 32</CLIENT.Telephone>
      <CLIENT.NCompte>001-0458625-85</CLIENT.NCompte>
   </CLIENT>
[...]
</Client-Livre-Achat>
```

Figure 2.14- Eviter les collisions de noms

3.3. Nombre de documents XML

Jusque maintenant, notre discussion a porté sur les choix de conception de la structure d'**un** document contenant des données d'une base de données. Il est évident que l'on peut choisir de générer plusieurs documents XML.

Par exemple, on peut construire un document XML par table de la base de données. Ceci présente l'avantage d'éviter les collisions de noms qui peuvent apparaître lorsque deux attributs de la base de données, modélisés par des types d'éléments, portent le même nom. Malheureusement, les liens inter-types d'entité ne peuvent être modélisés dans ce cas.

Certains auteurs (cf. [5]) envisagent l'utilisation de documents XML comme support d'enregistrement de données persistantes. Dans ce cas, il y a une recherche judicieuse à mener en ce qui concerne le nombre et la taille des documents XML à produire. En effet, plus le document XML est de taille importante, plus la recherche en son sein est lente (le document entier doit être passé en revue). Si générer plusieurs documents XML facilite la recherche en leur sein, il faut être conscient que, si le nombre de documents est élevé, il est impossible d'exploiter le mécanisme de liaison inter-élément de XML (attribut de type ID/IDREF). De plus, il faut développer des mécanismes de recherche du bon document XML. Il y a donc un juste milieu à trouver.

CHAPITRE 3

Démarche méthodologique pour la génération d'un migrateur

L'objectif de notre travail n'est pas seulement de supporter la production de document(s) XML à partir d'une base de données via l'utilisation d'une application spécifique à cette tâche. Nous voulons également **automatiser** autant que faire se peut, la **construction** de cette application.

A cette fin, nous allons proposer une démarche méthodologique permettant d'aboutir à la construction d'un migrateur de données d'une base de données vers un ou plusieurs document(s) XML (telle celle décrite au chapitre précédent). Cette démarche est constituée d'un certain nombre d'étapes, certaines étant manuelles, d'autres semi-automatiques, d'autres l'étant entièrement.

1. Aperçu général de la démarche méthodologique

L'application de support à la migration de données a été conçue pour s'appliquer à une base de données particulière afin de produire un type (structure) de document(s) XML fixé. La construction d'une telle application nécessite donc de connaître :

- la structure de la base de données;
- la structure du/des document(s) XML;
- les correspondances structurelles BD/XML.

La démarche méthodologique est constituée de deux parties : une première partie vise à rendre les différents matériaux de construction disponibles sous un format adéquat et une seconde partie prend en charge la génération proprement dite de l'application de migration.

La figure suivante illustre cette démarche méthodologique. La ligne horizontale apparaissant en pointillés sépare les deux parties.

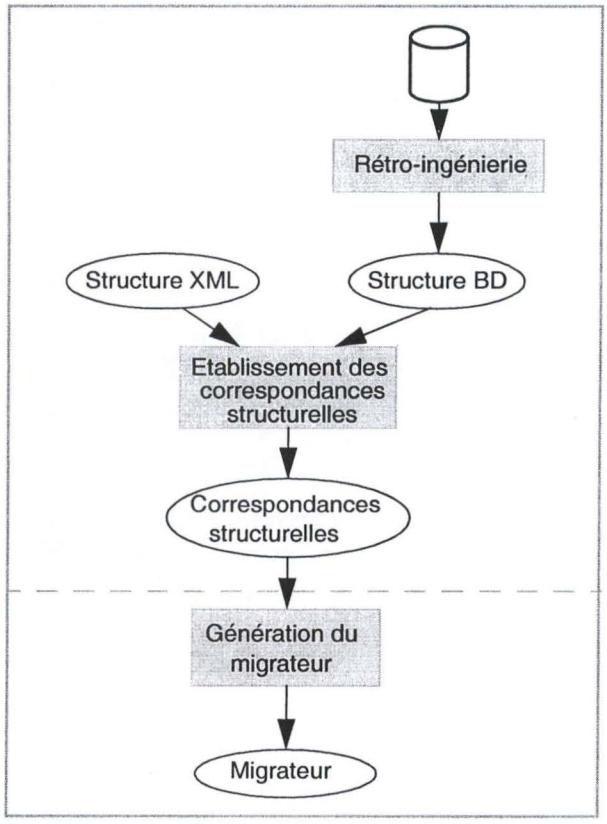


Figure 3.1- Démarche méthodologique pour la génération d'une application de support à la migration de données BD/XML.

La suite de ce chapitre contient une description des différents processus et produits[1] constitutifs de la première partie de cette démarche.

1. Nous désignons par **produit** tout document utilisé, modifié ou produit durant le cycle de vie d'un système d'informations. Un **processus**, quant à lui, désigne un groupe d'opérations agissant sur un ou plusieurs produits.

2. Rétro-ingénierie

Si la base de données existe bel et bien, cela n'implique en rien que sa structure soit correctement spécifiée. En effet, le cycle de vie d'une base de données, constitué d'une étape de conception et d'un certain nombre d'étapes de maintenance, présente de multiples occasions de corruption et de perte de la spécification de la structure.
Par exemple, l'étape de conception est censée mener non seulement à l'établissement de la structure de la base de données (établie suite à une analyse des besoins) mais également à une documentation complète de cette structure. Il va sans dire que cette documentation peut avoir été égarée au cours du temps ou qu'elle peut même n'avoir jamais existé.
Les multiples étapes de maintenance, quant à elles, peuvent entraîner des changements structurels qui ne sont pas toujours correctement documentés.

Ce sont là quelques raisons parmi d'autres pour lesquelles il est nécessaire de procéder à une étape au cours de laquelle l'analyste essaie de comprendre et de re-documenter la base de données d'une application. Plus précisément, cette étape, appelée étape de **rétro-ingénierie,** aboutit à l'élaboration des schémas physique, logique et/ou conceptuel.

Cette étape est extrêmement complexe. Ceci est dû notamment au fait de l'absence quasi généralisée de documentation. D'autre part, dans le cas où la base de données s'apparente à un ensemble de fichiers permanents (ex : COBOL), il n'est pas rare de constater que les définitions des structures de données soient dispersées à travers tout le code procédural.

L'étude du processus de rétro-ingénierie[1] dépasse largement la portée de ce travail, c'est pourquoi nous supposerons, pour la suite, disposer de la structure de la base de données décrite à un certain niveau d'abstraction.

Une fois l'étape de rétro-ingénierie menée à son terme, nous disposons de la structure de la base de données. Si la structure des documents XML est également disponible, l'étape suivante consiste à établir les correspondances entre la structure de la base de données et celle des documents XML.

Pour établir ces correspondances, il est nécessaire de disposer d'un formalisme de représentation commun. Le formalisme que nous utilisons est le modèle *GER*. C'est un modèle général de représentation de structures de données. Il fait l'objet d'une description à l'annexe A.

Nous supposons qu'on dispose des structures de la base de données représentées conformément au modèle GER.

1. Pour plus de renseignements sur la rétro-ingénierie, consultez [6].

3. Structure d'un document XML

Dans cette section, nous faisons l'hypothèse que la structure du ou des document(s) XML à produire a déjà été fixée (cf. Section 3. Page 42).

Nous allons voir comment le modèle GER peut être utilisé pour représenter un DTD. Il est important de faire remarquer que :

- nous nous limitons à la représentation d'un sous-ensemble de DTD;
- la représentation que nous avons adoptée présente des lacunes.

3.1. Représentation de la structure d'un document XML (DTD)

Afin de donner un aperçu des conventions de représentation d'un DTD que nous avons prises, les figures 3.2 et 3.3 donnent un exemple d'un DTD et de sa représentation dans le modèle GER.

```
<!ELEMENT Catalogue(Editeur*,Livre*)>

<!ELEMENT Editeur(NomSociete, Adresse+, Auteur*)>
<!ELEMENT NomSociete(#PCDATA)>

<!ELEMENT Adresse(Rue, Ville, Province, Pays, CodePostal)>
<!ELEMENT Rue(#PCDATA)>
<!ELEMENT Ville(#PCDATA)>
<!ELEMENT Province(#PCDATA)>
<!ELEMENT Pays(#PCDATA)>
<!ELEMENT CodePostal(#PCDATA)>

<!ELEMENT Auteur(Nom, Prenom, Biographie, Portrait)>
<!ELEMENT Nom(#PCDATA)>
<!ELEMENT Prenom(#PCDATA)>
<!ELEMENT Biographie(#PCDATA)>
<!ELEMENT Portrait>
<!ATTLIST Auteur AuteurID ID #REQUIRED>

<!ELEMENT Livre(Titre, Resume, Genres, Prix?)>
<!ELEMENT Titre(#PCDATA)>
<!ELEMENT Resume(#PCDATA)>
<!ELEMENT Genres(Genre, Genre, Genre)>
<!ELEMENT Genre(#PCDATA)>
<!ELEMENT Prix(#PCDATA)>

<!ATTLIST Livre ISBN ID #REQUIRED
                auteurs IDREFS #REQUIRED>
```

Figure 3.2- Exemple d'un DTD.

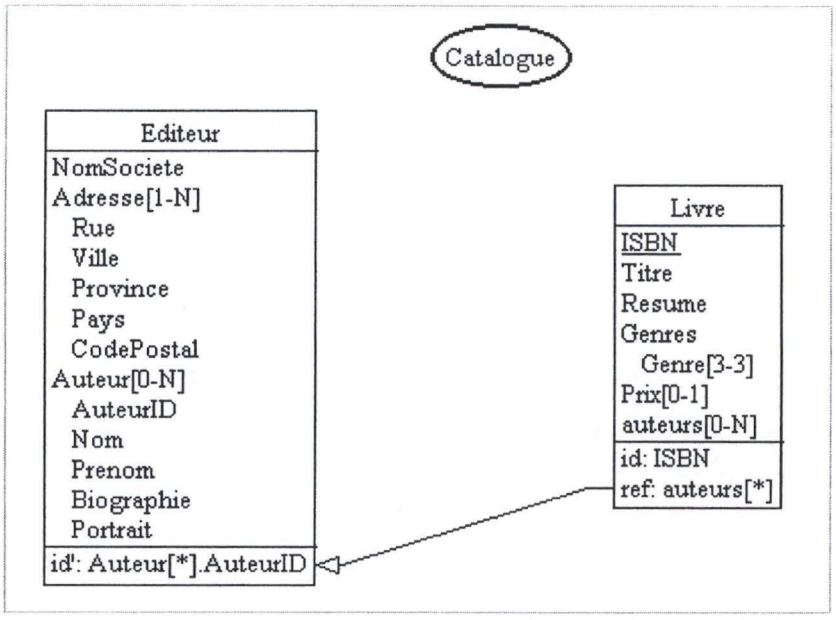


Figure 3.3- Représentation du DTD de la figure 3.2 dans le modèle GER.

Nous allons, à présent, énoncer huit règles qui décrivent la représentation d'un DTD dans le modèle GER.

Afin d'éviter toute confusion, nous utiliserons les termes **attribut_xml** et **attribut_G** pour désigner respectivement le concept d'attribut dans le contexte de XML et dans le modèle GER. D'autre part, nous désignerons par le terme **parent d'un attribut,** le type d'entité ou l'attribut composé qui contient **directement** cet attribut.

- **Règle 1** :
  Le type d'élément racine est représenté par un schéma qui porte son nom.

- **Règle 2 :**
  Tout type d'élément, descendant direct du type d'élément racine soumis à l'opérateur de cardinalité *, est représenté par un type d'entité qui porte le même nom que lui (cf. Fig. 3.4).

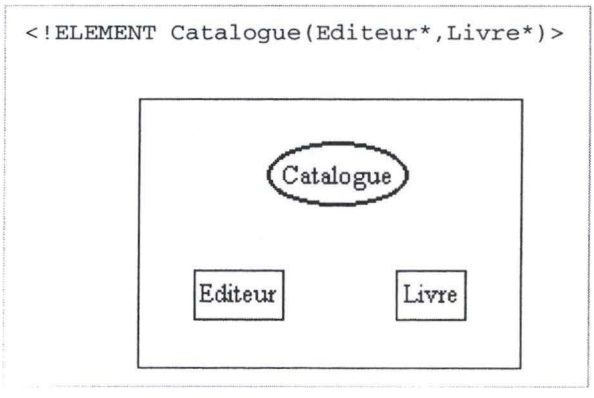


Figure 3.4- Représentation du type d'élément racine et des types d'élément qui sont ses descendants directs.

- **Règle 3 :**
  Tout type d'élément E, qui n'est pas un descendant direct du type d'élément racine, est représenté par un attribut_G de même nom. Le parent de cet attribut_G est la représentation du type d'élément, père de E.
  Par exemple, sur la figure 3.5, on montre comment représenter les types d'éléments NomSociété, Adresse et Auteur. Ils sont descendants du type d'élément Editeur représenté par un type d'entité.

  Si ce type d'élément est soumis à un opérateur de cardinalité +, *, ?, il est représenté par un attribut_G dont la cardinalité prend respectivement les valeurs [1-N], [0-N] et [0-1].
  Sur l'exemple de la figure 3.5, NomSociété n'est soumis à aucun opérateur de cardinalité, sa représentation est un attribut qui a comme valeur de cardinalité, la valeur par défaut, à savoir [1-1]. Par contre, Adresse est soumis à l'opérateur de cardinalité +, ce qui se traduit par une valeur de cardinalité [1-N].

  Ce type d'élément est représenté par un attribut_G simple ou composé en fonction de son type de contenu et de la présence ou non d'un attribut_xml qui lui est attaché.
  En effet,
  - si le type d'élément a un contenu de type #PCDATA et n'a aucun attribut_xml qui lui est attaché, il est représenté par un attribut_G simple. C'est le cas du type d'élément NomSociete;
  - si le type d'élément a un contenu de type ELEMENT, il est représenté par un attribut_G composé. C'est le cas du type d'élément Adresse. Ici, il n'y a aucune attribut_xml qui est lié à Adresse mais ce n'est pas une condition obligatoire;
  - si le type d'élément a un contenu de type EMPTY et a un attribut_xml qui lui est attaché, il est représenté par un attribut_G composé;
  - les autres combinaisons ne sont pas acceptées.

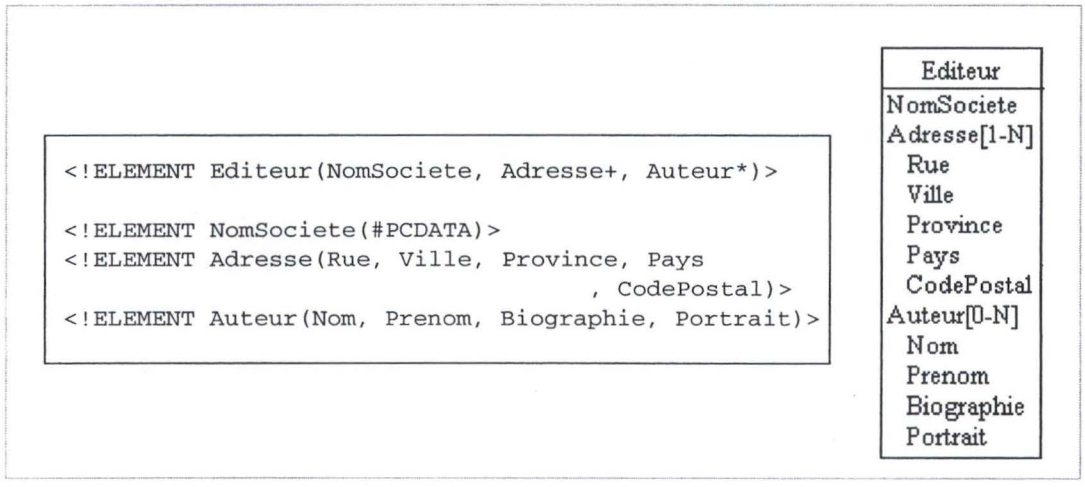


Figure 3.5- Représentation d'un type d'élément qui n'est pas un descendant direct du type d'élément racine.

- **Règle 4 :**
  Tout attribut_xml de type `ID` attaché à un type d'élément `E` est représenté par un attribut_G simple, mono-valué et de même nom dont le parent est la représentation du type d'élément `E`.

  Si l'attribut_xml est de type de contenu `#REQUIRED`, l'attribut_G qui le représente est obligatoire (cardinalité `[1-1]`); s'il est de type de contenu `#IMPLIED`, l'attribut_G est facultatif (cardinalité `[0-1]`).

  De plus, l'attribut_G représentant un attribut_xml de type `ID` est un identifiant du type d'entité qui le contient (directement ou pas). C'est toujours un identifiant secondaire sauf quand il est de type `#REQUIRED` et que le parent de l'attribut_xml est le type d'élément dont la représentation est le type d'entité.

  Sur la figure 3.6, le type d'élément `Auteur` a un attribut_xml de type `ID`, nommé `AuteurID`. Il est représenté par un attribut_G simple dont le parent est l'attribut_G composé `Auteur`. De plus, il est identifiant du type d'entité `Editeur`.
  De même, le type d'élément `Editeur` a un attribut_xml de type `ID` nommé `NomSociete`, ce dernier est représenté par un attribut_G simple du type d'entité `Editeur` représentant le type d'élément de même nom. Ce dernier est un identifiant primaire.
  Remarquons (anticipativement) que la représentation d'un attribut_xml de type `ID` est un attribut simple toujours positionné le premier parmi les autres attributs_G de son parent.

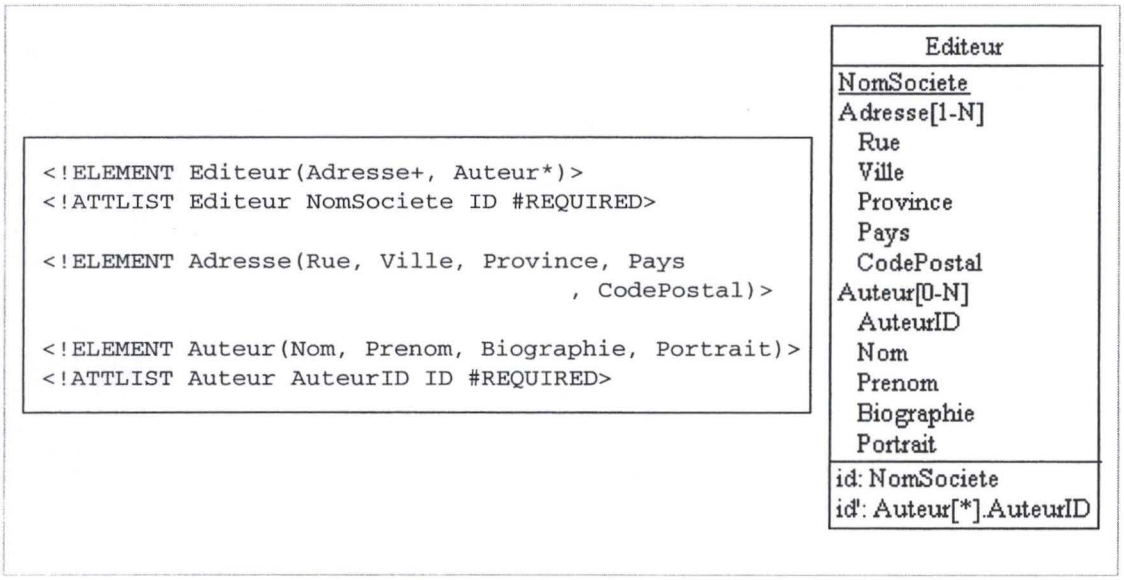


Figure 3.6- Représentation graphique d'un attribut de type ID.

- **Règle 5 :**
  Tout attribut_xml de type IDREF attaché à un type d'élément E est représenté par un attribut_G simple, mono-valué et de même nom, qui a pour parent, la représentation de E.
  Le caractère obligatoire ou facultatif de cet attribut_G est choisi de la même façon que pour la représentation d'un attribut_xml de type ID (cf. Règle 4).
  De plus, l'attribut_G est soumis à une contrainte référentielle (cf. Section 3.3. Page 49 pour déterminer la cible de cette contrainte référentielle).

  Par exemple, le type d'élément `Livre` a un attribut qui lui est attaché. Ce dernier nommé `auteurs` est de type IDREF. Il est représenté par un attribut_G dont le parent est le type d'entité `Livre` qui représente le type d'élément de même nom. Il est soumis à une contrainte référentielle (cf. 3.7).
  Notons (anticipativement) que la position de l'attribut_G permettant de représenter un attribut_xml de type IDREF est la dernière parmi les autres attributs_G de son parent.

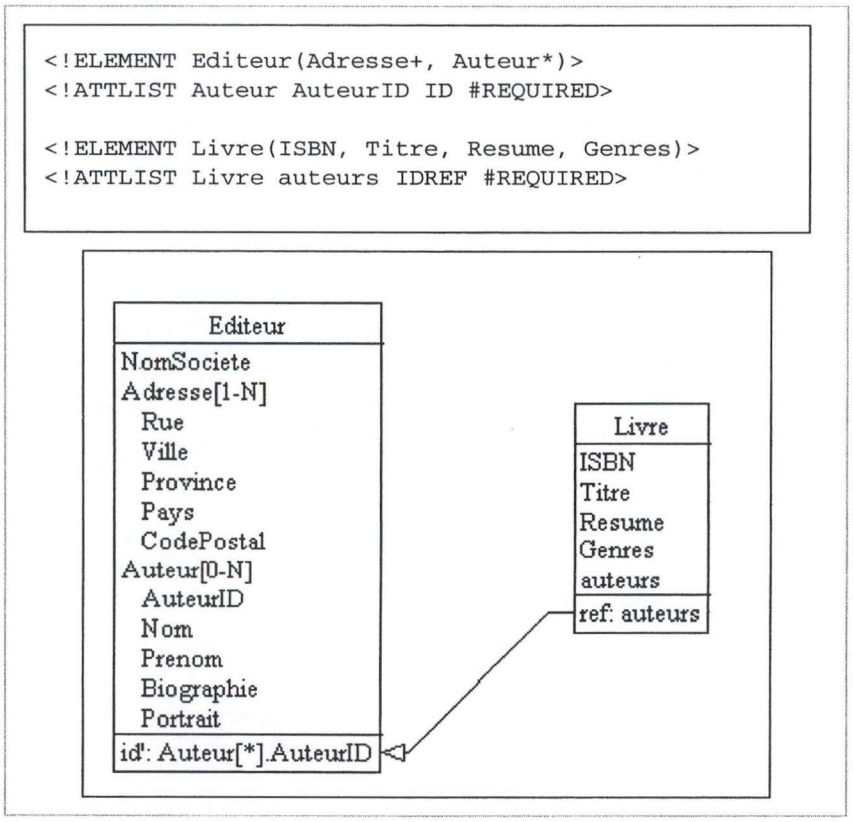


Figure 3.7- Représentation graphique d'un attribut de type IDREF.

- **Règle 6 :**
  Tout attribut_xml de type IDREFS attaché à un type d'élément E est représenté par un attribut_G simple multivalué qui a pour parent, la représentation de E. Nous avons adopté les mêmes conventions de représentation qu'en règle 5.

- **Règle 7 :**
  L'ordre des attributs_G d'un même type d'entité ENT (représentant un type d'élément E) est régi comme suit :
  - L'attribut_G représentant un attribut_xml de type ID attaché à E prend la première place dans ENT;
  - Les attributs_G représentant les types d'élément fils de E viennent ensuite dans un ordre identique à celui du contenu de E;
  - les attributs_G représentant les attributs_xml de type IDREF(S) attaché à E suivent dans un ordre calqué sur celui de leur apparition dans le DTD.

  Les mêmes conventions sont utilisées pour les attributs au sein d'un attribut composé.

- **Règle 8 :**
  On peut vouloir *in fine* générer non pas un seul document XML mais plusieurs. Il est possible de représenter ce fait par la notion de collection. De cette manière, il est possible de construire directement un migrateur qui génère plusieurs documents XML.

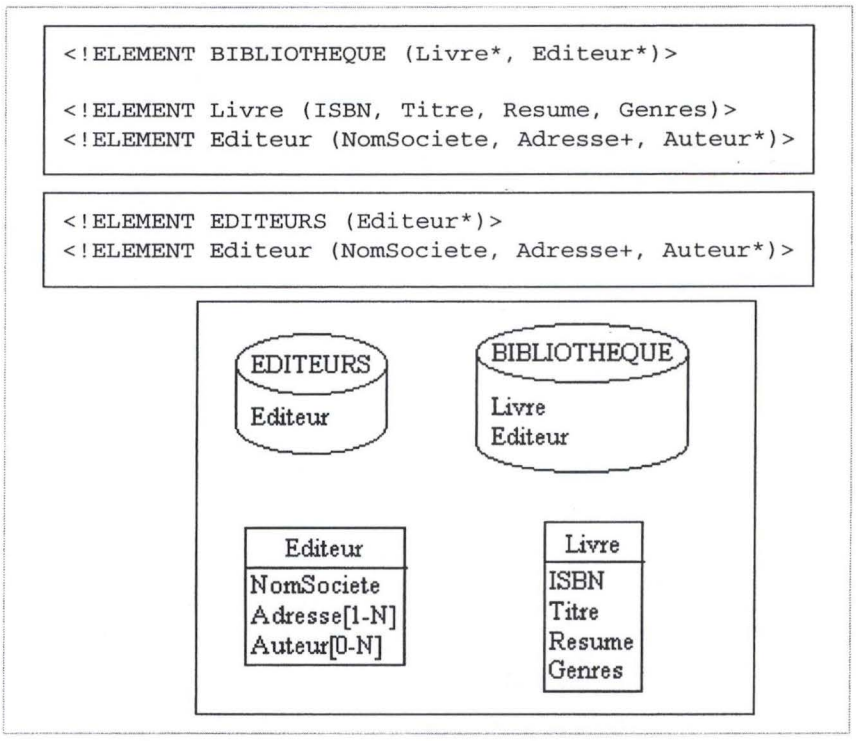


Figure 3.8- Représentation de la structure de plusieurs documents XML sur un même schéma.

Par exemple, la figure 3.8 représente deux DTD distincts qui partagent certaines déclarations.

3.2. Restriction sur les DTD

Ces règles ne permettent pas de représenter tous les DTD. Nous allons décrire précisément les DTD qui peuvent être représentés de cette manière. Dans certains cas, nous indiquerons comment ces règles de représentation peuvent être étendues afin de pouvoir considérer un ensemble plus large de DTD.

- Seuls les concepts de type d'élément et d'attribut sont pris en charge. Il existe d'autres concepts en XML qui ne sont pas supportés;
- Pour les types d'élément :
  - Seuls les types de contenu `EMPTY`, `ELEMENT` et `#PCDATA` sont admis. Les types de contenu `MIXED` et `ANY` ne sont pas supportés;
  - Il n'y a que le séparateur de type d'élément " , " qui peut être utilisé. Le caractère " | " n'est pas admis;
  - Tout type d'élément, descendant direct du type d'élément racine doit être soumis à l'opérateur de cardinalité `*`. Il n'est pas possible de représenter d'autre cardinalité puisque le modèle GER ne permet pas d'exprimer le nombre d'instances d'un type d'entité;
- Pour les attributs :
  - Tout attribut doit être de type `ID`, `IDREF` ou `IDREFS`. Les autres attributs ne sont pas supportés. Notons qu'ici, il est possible d'étendre le modèle afin de les prendre en charge, par exemple, en les représentant toujours comme attribut_G mais dont le nom serait, par exemple, écrit en gras;
  - Le type de contenu d'un attribut ne peut prendre que les valeurs `#REQUIRED` ou `#IMPLIED`.
- Agencement des types d'élément et des attributs :
  Seuls les agencements décrits à la règle 3 sont admis. En fait, il y a deux situations qui ne sont pas admises :
  - un type d'élément dont le contenu est de type `EMPTY` et auquel n'est associé aucun attribut. Dans ce cas, il est difficile de le représenter car les attributs_G sont des containers de données et ce type d'élément n'en est pas un.
  - un type d'élément dont le contenu est de type `#PCDATA` auquel est associé un attribut. Dans XML, ce type d'agencement représente deux containers de données (le type d'élément et l'attribut). Idéalement, pour le représenter dans le modèle GER, il faudrait utiliser un attribut_G composé portant le nom du type d'élément. Celui-ci contiendrait deux attributs : un attribut_G simple de même nom que l'attribut_xml et un deuxième attribut_G simple de nom `#PCDATA`.

3.3. Lacunes de la représentation

La représentation des DTD selon le modèle GER conduit à des lacunes sémantiques dont il faut avoir conscience.

- **Portée de l'identifiant dans un schéma représentant un DTD et dans celui représentant la structure d'une base de données :**
  Un attribut_xml de type `ID` identifie sans équivoque un élément au sein de tous les éléments (tous types confondus) du document XML. Autrement dit, la valeur d'un attribut_xml de type ID est unique parmi toutes les valeurs prises par les attributs_xml de type ID du document.
  En revanche, un attribut_G identifiant d'un type d'entité désigne une entité parmi toutes les entités de ce type. Autrement dit, un attribut_G identifiant ne prend jamais deux fois la même valeur (indépendamment des valeurs prises par les autres attributs_G identifiants).
  Il y a donc une différence importante dans la portée de l'identifiant.
  Une solution à ce problème consiste à préfixer la valeur de chaque attribut_G identifiant par le nom du type d'entité et des attributs composés qui le contiennent. Si nous appliquons ce traitement à la base de données, la sémantique associée à un attribut identifiant dans un schéma représentant la structure d'une base de données et dans un autre représentant un DTD est équivalente.

- **Ordre :**
  Les types d'éléments descendants directs du type d'élément racine (tel `Editeur` et `Livre`, cf. Fig. 3.2) sont ordonnés dans le DTD. Ainsi, dans tout document XML conforme à ce DTD, les éléments de type `Editeur` précèdent ceux de type `Livre`.
  Cette notion d'ordre n'est pas présente dans le schéma représentant ce DTD car les types d'entité ne sont pas ordonnés au sein d'un schéma.
  Il est possible de pallier cette lacune en usant de subtilité, par exemple, on peut ajouter un chiffre représentant cet ordre en suffixe du nom de chaque type d'entité.

- **Représentation d'un attribut de type `IDREF` :**
  Un attribut_xml de type `IDREF` est représenté par un attribut_G sur lequel est défini une contrainte référentielle.
  Ce choix de représentation implique que l'attribut_xml de type `IDREF` référence toujours des éléments de même type; ce qui n'est pas obligatoire dans XML. Cependant, comme les données du document XML proviennent d'une base de données, le mécanisme `ID/IDREF` ne sera exploité que pour représenter des contraintes référentielles et donc, la représentation n'est pas réductrice.
  La contrainte référentielle a pour cible l'attribut_G dont le parent est la représentation du type d'élément vers lequel toutes les valeurs de l'attribut_xml de type `IDREF` pointent.
  L'information nécessaire pour représenter un attribut_xml de type `IDREF` (et plus précisément, pour déterminer la cible de la contrainte référentielle) n'est pas entièrement disponible dans le DTD. Elle doit être déduite de l'interprétation que l'on fait du DTD.

- **Représentation d'un attribut de type `IDREFS` :**
  Pour un même élément, l'attribut_xml de type `IDREFS` peut prendre plusieurs valeurs. Les spécifications XML n'imposent pas que ces valeurs désignent des éléments de même type. En revanche, la représentation que nous avons choisie l'impose. Par exemple, la figure 3.9 montre un exemple de fichier XML dont la structure n'est pas exprimable ici.

```
<Bibliothèque>
   <Livre numero='1'>...</Livre>
   <Livre numero='2'>...</Livre>
   <Article num='1254'>...</Article>
   <Auteur numero='C1' a_ecrit='1 1254'/>
</Bibliothèque>
```

Figure 3.9- Exemple de document XML pour lequel la structure n'est pas représentable selon les règles décrites en Page 43.

En effet, l'attribut `a_ecrit` attaché au type d'élément `Auteur` prend deux valeurs référençant chacune un élément. La première référence un élément de type `Livre`, alors que la seconde pointe vers un élément de type `Article`. L'attribut `a_ecrit` devrait selon les règles de représentation énoncées ci-dessus être représenté par un attribut_G (du type d'entité représentant `Auteur`). De plus, une contrainte référentielle doit être définie à partir de cet attribut_G. C'est là que réside l'impossibilité de représentation puisqu'il faut définir une cible à cette contrainte référentielle...
Notons que puisque les données des documents XML proviennent d'une base de données, il y a fort à parier qu'on ne soit pas confronter à une telle situation.

3.4. Le modèle XML

Si nous ne nous attachons pas à représenter tous les concepts de XML, ni même toutes les combinaisons possibles de ces concepts, nous n'utilisons pas non plus tous les concepts du modèle GER pour notre représentation. Un schéma XML, entendez un schéma représentant un DTD sous le modèle GER, est donc non seulement conforme au modèle GER, mais aussi à un sous-modèle de ce modèle. Ce sous-modèle est baptisé **modèle XML**.

Un schéma qui se conforme au modèle XML, qu'on appelle **schéma XML**, vérifie les conditions suivantes :

- un schéma XML ne contient pas de types d'association;
- un schéma XML ne contient pas de relations IS-A;
- les attributs simples ou composés d'un schéma XML qui ne sont ni identifiants, ni la source d'une contrainte référentielle, doivent avoir une cardinalité qui vaut : [0-1], [1-1], [0-N] ou [1-N];
- un type d'entité peut avoir plusieurs identifiants pour autant que ceux-ci aient tous un parent différent (par exemple, un attribut de niveau 1 et un attribut de niveau 2 peuvent être tous les deux identifiants d'un même type d'entité);
- le nom des types d'entité, attributs, schéma doit être unique au sein du schéma;

- un groupe identifiant ne peut être constitué que d'un seul attribut simple et mono-valué;
- un groupe référentiel ne peut être constitué que d'un seul attribut simple;
- il ne peut y avoir d'autre groupe que les groupes identifiants et de contraintes référentielles simples;
- les noms du schéma, de chaque type d'entité et de chaque attribut doivent se conformer à la règle suivante : ils doivent commencer par une lettre ou un *underscore* (_); les autres caractères sont des lettres, chiffres, un underscore, un double-points, une virgule ou un tiret;

Remarques

- La condition "*le nom des types d'entité, attributs, schéma doit être unique*" est trop restrictive. En effet, il faut imposer que le nom des objets du schéma correspondant à des types d'élément soient différents car un DTD ne peut contenir deux types d'élément de même nom. En revanche, en ce qui concerne le nom des attributs de XML, un type d'élément ne peut avoir deux attributs de même nom MAIS deux types d'éléments peuvent se voir attacher chacun un attribut de même nom. La condition devrait être énoncée de nouveau pour refléter correctement ce fait. Cependant, dans le cas où on voudrait automatiser la vérification des conditions décrites ci-dessus, la petite restriction que l'on soulève ici fait gagner beaucoup en simplicité.
- Nous avons omis de parler du type et de la longueur des attributs dans un schéma XML. En fait, XML n'impose aucune restriction sur la longueur du contenu de ses attributs et de ses éléments. D'autre part, tout le contenu est de type textuel. Formellement, la longueur des attributs dans un schéma XML est infinie et leur type est alphanumérique. Concrètement, nous les ignorerons.

Nous allons supposer que le DTD associé au document XML à produire est représenté sous la forme d'un schéma XML. Ceci implique que nous nous restreignons aux seuls DTD que nous sommes capables de représenter avec le modèle XML.

4. Etablissement des correspondances

A ce stade-ci, nous disposons donc de la structure de la base de données et de celle du ou des document(s) XML exprimées dans un formalisme commun de représentation. Nous sommes maintenant en mesure d'établir les correspondances structurelles entre les deux.

Ces correspondances structurelles sont particulièrement importantes pour la suite car elles permettront :

- d'établir de quel type d'entité il faut extraire les données;
- de déterminer les règles de conversion de ces données afin qu'elles répondent aux exigences de l'utilisateur.

Nous allons présenter deux façons de procéder pour établir les correspondances.

4.1. Comparaison.

La méthode la plus naturelle pour établir les correspondances entre la structure de la base de données et celle du ou des documents XML à produire est de traiter chaque construction (ou groupe de constructions) du schéma de la base de données tour à tour afin d'identifier la construction (ou groupe de constructions) qui lui correspond dans le schéma XML. Cette étape purement manuelle est extrêmement fastidieuse dans le cas de schémas de grande taille. De plus, une telle démarche est difficilement automatisable.

4.2. Approche transformationnelle

Il existe une deuxième approche, dite approche transformationnelle. A première vue, une transformation de schéma est un opérateur `T` qui remplace une construction `C` du schéma `S` par une construction `C'`, transformant ainsi le schéma `S` en un schéma `S'`.

Par exemple, nous pouvons définir une transformation de schéma qui remplace un attribut multivalué en une série d'attributs monovalués. La figure 3.10 illustre cette transformation.

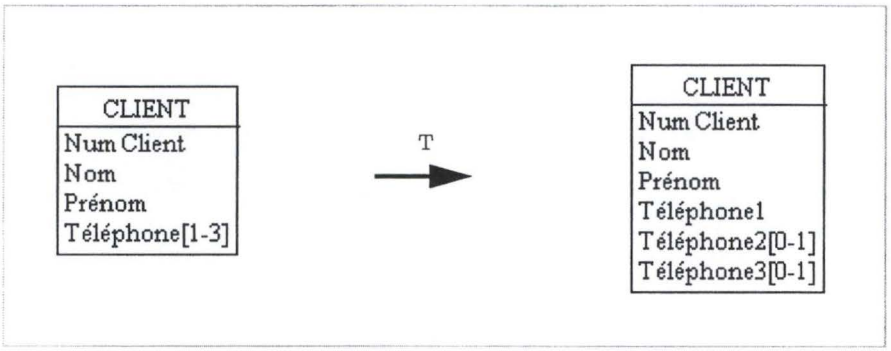


Figure 3.10- Transformation d'un attribut multivalué en une série d'attributs monovalués.

Les deux schémas apparaissant à la figure 3.10 semblent être *équivalents*, c'est-à-dire exprimer la même sémantique. On parle dans ce cas d'un opérateur de transformation *semantic-preserving*.

Selon la définition de transformation énoncée ci-dessus, on peut identifier deux situations extrêmes lorsque C ou C' est vide. Si C est vide, la transformation consiste à ajouter de nouvelles constructions au schéma. Dans ce cas, on la qualifie de *semantics-augmenting*. En revanche, lorsque C' est vide, la transformation consiste à enlever des constructions du schéma. On parle dans ce cas, de transformation *semantics-decreasing*.

La transformation illustrée à la figure 3.10 décrit la correspondance structurelle entre les types de données mais ne dit rien au sujet de la relation entre les instances de ces types de données. Cela dit, dans notre exemple, tout laisse à penser que la relation inter-instances est la suivante :
A chaque instance c du type d'entité CLIENT du schéma initial correspond une instance c' du type d'entité CLIENT du schéma cible où :

- les valeur des attributs NumClient, Nom et Prénom de c' sont les mêmes que dans c;
- les valeurs des attributs Téléphone1, Téléphone2 et Téléphone3 de c' sont respectivement la première, deuxième et troisième valeur de l'attribut Téléphone dans c.

Cependant, beaucoup d'autres relations inter-instances pourraient être définies.
Par exemple :
A chaque instance c du type d'entité CLIENT du schéma initial correspond une instance c' du type d'entité CLIENT du schéma cible où :

- les valeur des attributs NumClient, Nom et Prénom de c' sont les mêmes que dans c;
- la valeur de l'attribut Téléphone1 de c' est la concaténation des valeurs prises par l'attribut Téléphone dans c;
- les valeurs des attributs Téléphone2 et Téléphone3 de c' sont nulles.

Pour spécifier correctement une transformation de schéma, il faut donc préciser les relations inter-schémas (T) mais également les relations inter-instances (t).

Plus précisément, un opérateur de transformation de schéma Σ est complètement défini par le couple <T, t> où :

- T, appelé **mapping structurel,** remplace une construction C du schéma S par une construction C', transformant ainsi le schéma S en schéma S';
- t, appelé **mappping d'instance**, définit comment traduire les instances c de C en instances c' de C';

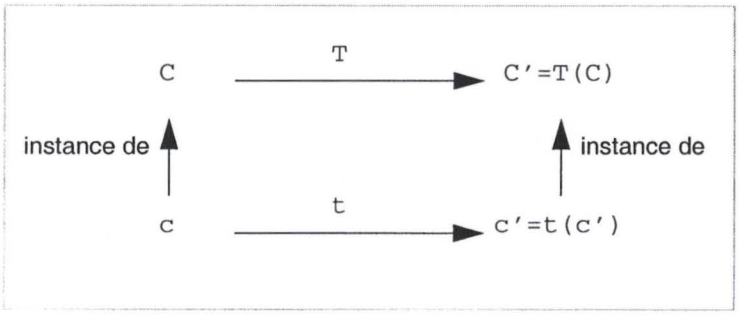


Figure 3.11- Les deux mappings d'un opérateur de transformation de schéma.

Le premier mapping décrit la syntaxe de la transformation alors que le second en décrit la sémantique.

Nous pouvons maintenant spécifier complètement la transformation d'un attribut multi-valué en une série d'attributs monovalués de la figure 3.10 (cf. Fig. 3.12).

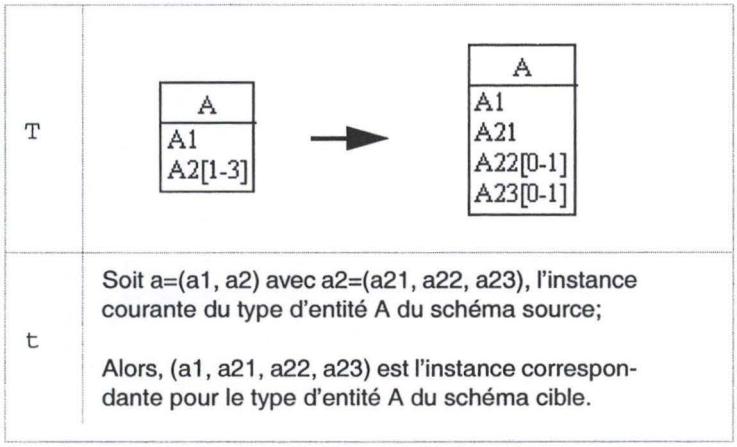


Figure 3.12- Spécification complète de la transformation d'un attribut multi-valué en une série d'attributs monovalués.

La notion d'opérateur de transformation de schéma est utile quand il s'agit d'établir les correspondances entre un schéma S et un schéma S'.
En effet, si nous disposons d'un certain nombre d'opérateurs de transformation entièrement spécifiés, des mappings structurels peuvent être appliqués sur le schéma S afin d'obtenir le schéma S'. La combinaison équivalente des mappings d'instance permet de connaître les règles de conversion de données.

Par exemple, nous pouvons définir la transformation d'un type d'entité en un attribut composé (cf. Fig. 3.13).

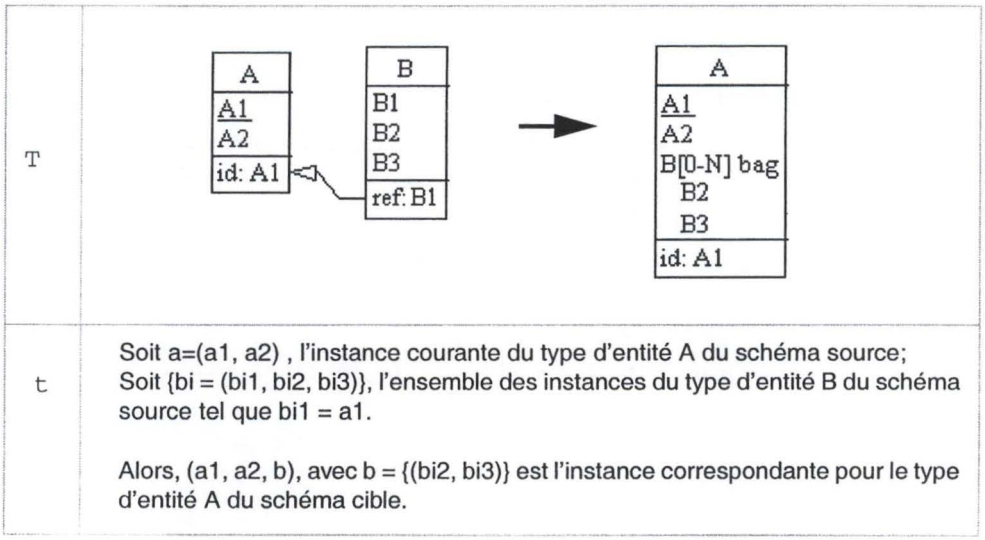


Figure 3.13- Transformation d'un type d'entité en un attribut composé.

La figure 3.14 montre les schéma S et S' entre lesquels il s'agit d'établir les correspondances.

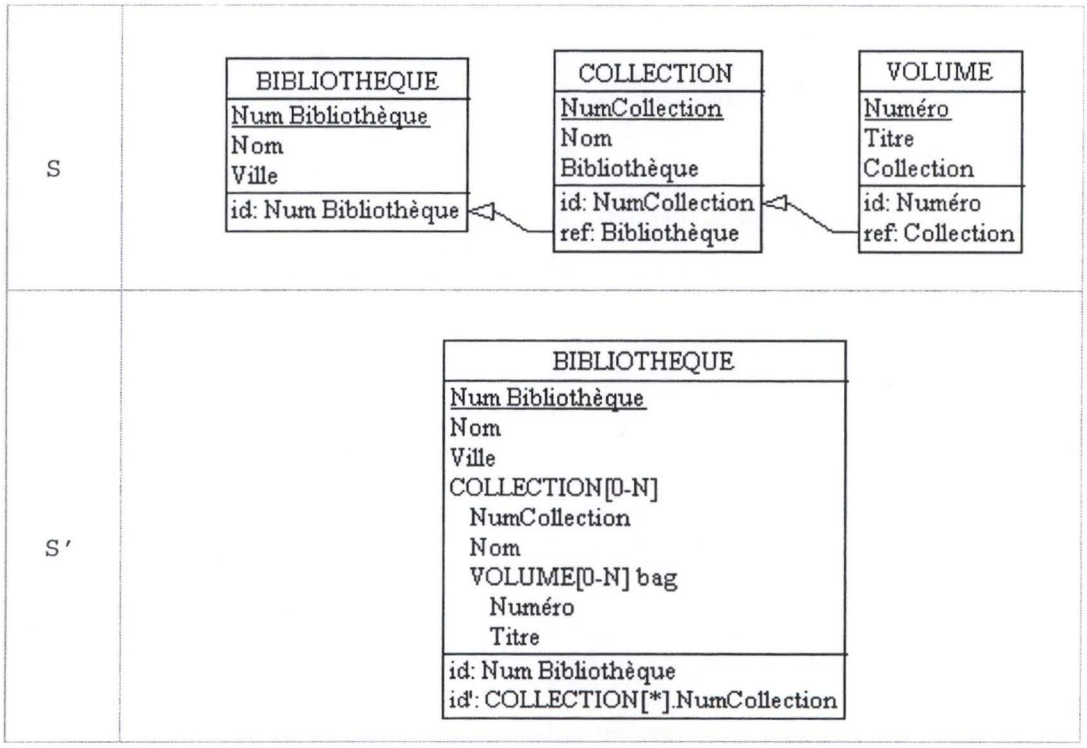


Figure 3.14- Exemple de schémas source et cible entre lesquels il faut établir les correspondances.

On peut observer qu'il suffit d'appliquer deux fois le mapping structurel T de la transformation d'un type d'entité en attribut (cf. Fig. 3.13) sur le schéma source S pour obtenir le schéma cible S' ; d'abord sur le type d'entité VOLUME et ensuite, sur le type d'entité COLLECTION.

Il reste à combiner les mapping d'instance correspondant pour connaître les règles de conversion de données.

4.3. Avantages de l'approche transformationnelle

Dans la pratique, il existe des centaines d'opérateurs de transformations. Il n'est donc pas possible de les définir tous précisément. Cependant, il est possible de définir formellement un ensemble d'opérateurs de transformations élémentaires qui peuvent être combinés afin de construire des opérateurs de transformations plus complexes. La chaîne de transformations à déterminer n'est rien d'autre qu'une combinaison de ces opérateurs de transformations (complexes ou élémentaires). Il est évident qu'une fois ces opérateurs de transformations définis formellement, ils peuvent être réutilisés pour d'autres cas.

De plus l'établissement des correspondances par la méthode de recherche d'une chaîne de transformations pourrait être fortement facilitée par le recours à un outil de description de schéma offrant une palette d'opérateurs de transformations qu'il est possible d'instancier et d'appliquer au gré des besoins sur les différentes constructions du schéma. Cet outil devrait permettre l'enregistrement automatique des différentes transformations effectuées sur le schéma.

5. Démarche méthodologique enrichie

Les considérations et hypothèses faites dans ce chapitre nous permettent de préciser la démarche méthodologique présentée à la figure 3.1.

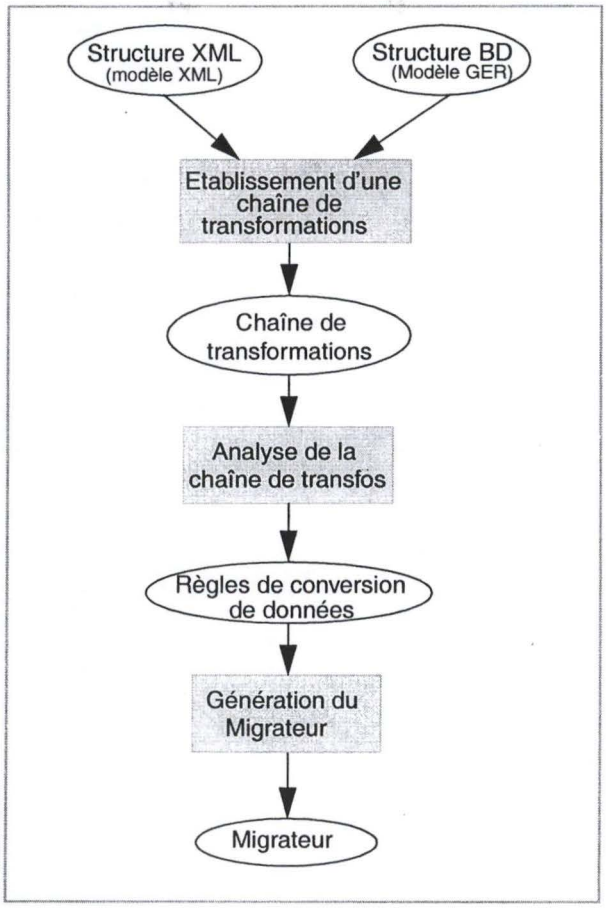


Figure 3.15- Démarche méthodologique enrichie.

CHAPITRE 4

Construction d'un prototype de migrateur BD/XML

Avec ce chapitre débute la seconde partie de ce document : celle relative à l'implémentation. Ce chapitre traite de l'implémentation d'un migrateur inspiré de l'analyse qui en est présentée au chapitre 2. Le chapitre suivant présentera la mise en pratique de la démarche méthodologique proposée au chapitre 3.

Plus précisément, le chapitre 4 traite d'abord des outils choisis pour implémenter le migrateur. L'architecture de ce dernier est fidèle à celle que nous avons présentée au chapitre 2 sous l'appellation "Architecture améliorée". Il faut évidemment fixer la stratégie à adopter pour l'extraction des données. Enfin, nous terminons en décrivant une découpe plus fine du migrateur. Une description plus complète de chaque "bloc" est reléguée en annexe.

1. Outils

1.1. Le langage de programmation

Notre choix s'est porté sur le langage de programmation JAVA pour les raisons suivantes :

- sa portabilité;
- sa compatibilité avec le module d'accès au DMS : ce module offre une interface écrite en JAVA;
- sa réutilisabilité : le prototype que nous avons développé ne couvre qu'un nombre restreint de cas. Il est donc nécessaire d'envisager des extensions. Elles seront d'autant plus faciles à implémenter si nous pouvons réutiliser des composants existants;
- sa disponibilité d'une librairie de classes et d'interfaces prêtes à l'emploi.

Pour plus de précision sur le langage JAVA, consultez [7].

1.2. Le module d'accès au DMS

Le module d'accès au DMS que nous avons choisi est un module développé dans le cadre du projet InterDB du **L**aboratoire d'**I**ngénierie de **B**ase de **D**onnées (LIBD) de l'Institut d'Informatique (FUNDP[1]-Namur).

Ce module, spécifique à une base de données particulière, permet l'accès à cette base de données (connection, requête de consultation, requête de mise à jour) à partir de son schéma logique. Cela permet à l'utilisateur d'exécuter des requêtes construites en connaissant uniquement la structure de la base de données décrite au niveau logique. En d'autres mots, le module InterDB cache les détails techniques d'implémentation et la syntaxe spécifique au DMS.
Le module InterDB suit l'architecture présentée à la figure 4.1.

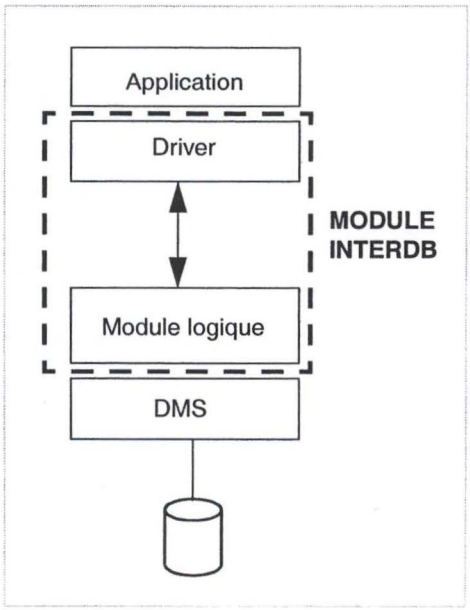


Figure 4.1- Architecture du module InterDB.

1. FUNDP (**F**acultés **U**niversitaires **N**otre-**D**ame de la **P**aix).

Le **module logique** est chargé de traduire dynamiquement les requêtes construites sur base du schéma logique en requêtes primitives. Les requêtes soumises au module logique sont exprimées dans un langage propriétaire au projet InterDB, appelé LQL[1]. Les requêtes primitives, quant à elles, sont exprimées dans le langage du DMS (ex: SQL dans le cas d'un DMS relationnel). Ce module logique a également pour rôle de composer les données physiques (renvoyées par le DMS) en données logiques (correspondantes aux structures logiques).

Le **driver**, quant à lui, offre une interface JAVA (similaire à JDBC[2]) pour le module logique.

Le module InterDB est spécifique à une base de données. Cependant, sa construction a été automatisée par le biais d'un générateur développé dans le cadre du projet InterDB.

Théoriquement, un tel module peut être construit quel que soit le DMS. Cependant, dans la pratique, le projet InterDB s'est restreint au cas des bases de données relationnelles et des bases de données COBOL. D'autre part, pour le développement du générateur de module, l'équipe InterDB impose que la structure de la base de données décrite au niveau logique soit un schéma conforme au modèle L3 décrit à la figure 4.2 [3].

Conditions pour qu'un schéma soit conforme au modèle L3
Schéma de niveau logique càd : sans type d'association; sans contrainte de hiérarchie ISA; sans collection; sans clef d'accès
Un type d'entité a au moins un attribut.
Un type d'entité a au plus un groupe identifiant primaire.
Un type d'entité ne contient pas de groupe de coexistence, d'at_least_one ou d'exclusivité, uniquement des groupes identifiant ou référentiel.
Un attribut simple et de niveau 1 est de cardinalité [i-1] ou [1-j] avec i=0,1 et j>1.
Un attribut composé est de cardinalité [i-1] avec i=0,1.
Un attribut de niveau > 1 est mono-valué (éventuellement optionnel).
Un attribut peut être de type CHAR, VARCHAR, NUMERIC ou DATE.
Un groupe est constitué uniquement d'attributs simples et monovalués et de premier niveau.
Une contrainte référentielle cible un groupe qui a le même domaine de valeurs.

Figure 4.2- Description du modèle L3.

L'utilisation de ce module comme module d'accès au DMS nous impose que :

- la base de données doit être de type relationnel ou COBOL;
- le schéma logique de la base de données doit être conforme au modèle L3.

Signalons que la présentation que nous avons faite sur le module InterDB est une vue extrêmement réductrice du projet InterDB. Pour plus d'informations, voir [8].

---

1. similaire à SQL.
2. JDBC (Java DataBase Connectivity). Pour plus de renseignements, consultez [9].
3. Le modèle L3 est en constante évolution. L'article [10] décrit le modèle à son origine tandis que la figure 4.2 le décrit tel qu'il est au moment de la rédaction de ce document.

2. Architecture du prototype

Le prototype de migrateur que nous avons construit suit l'architecture générale présentée ci-avant (cf. Section 2.3. Page 31). Le choix du module d'accès au DMS nous permet de particulariser cette architecture, comme le montre la figure 4.3.

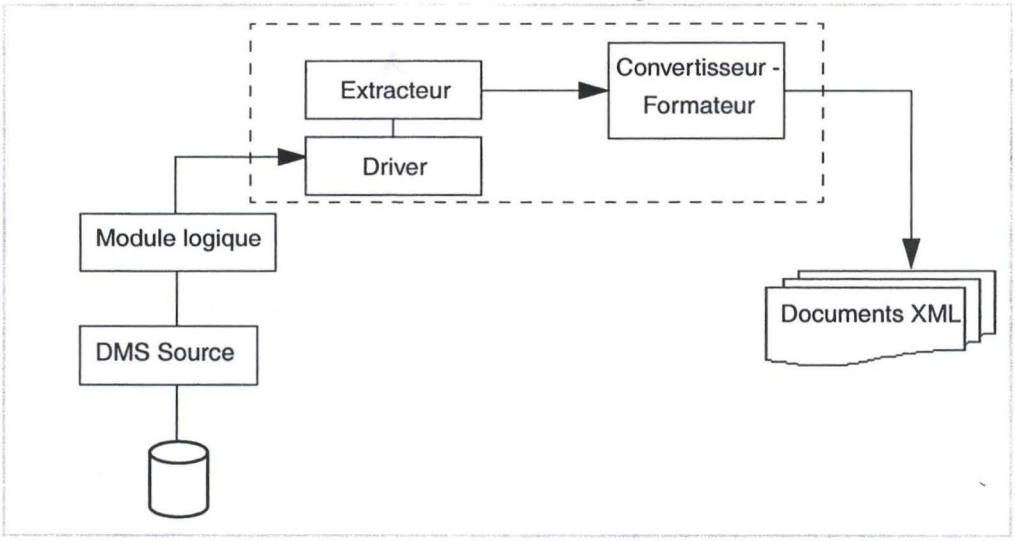


Figure 4.3- Architecture du prototype du migrateur.

Pour construire le module Extracteur, il est nécessaire de définir le nombre et la forme des requêtes d'extraction. Nous avons choisi d'associer une requête d'extraction par type d'entité de la base de données, à l'exception des types d'entité desquels aucune donnée n'est requise. La requête d'extraction porte sur la totalité des données, même si cela n'est pas nécessaire. Les données superflues seront écartées par le module Convertisseur-Formateur. La requête d'extraction peut être paramétrisée par le résultat d'une autre requête d'extraction dans le cas où elle porte sur une table qui a subi une transformation en un attribut multivalué. C'est le cas qui est illustré à la figure 2.5.

Le choix de l'algorithme du migrateur a été orienté par diverses considérations.
En premier lieu, toutes les instances d'un même type d'entité de la base de données doivent être traitées de la même façon. Il y a donc tout à gagner à grouper leur traitement.
Ensuite, nous faisons l'hypothèse que toutes les données d'un même type d'entité de la base de données seront contenues dans le même document XML. Les documents XML sont donc créés successivement.
Enfin, nous ne considérons que les deux cas suivants :

- **Cas 1** : un type d'entité A du schéma XML est construit à partir d'un seul type d'entité B du schéma de la base de données (cf. Fig. 4.4).

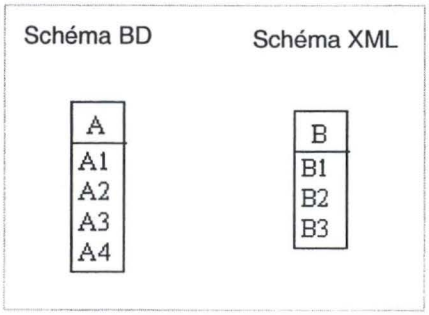


Figure 4.4- Un type d'entité du schéma XML est construit à partir d'un seul type d'entité du schéma de la base de données.

Dans ce cas, les instances de B peuvent être construites à partir des seules instances de A. On ne construit, ici, qu'une seule requête d'extraction portant uniquement sur A et indépendante de toutes les autres;

- **Cas 2** : un type d'entité A du schéma XML est construit à partir de plusieurs types d'entité Bi du schéma de la base de données (cf. Fig. 4.5).

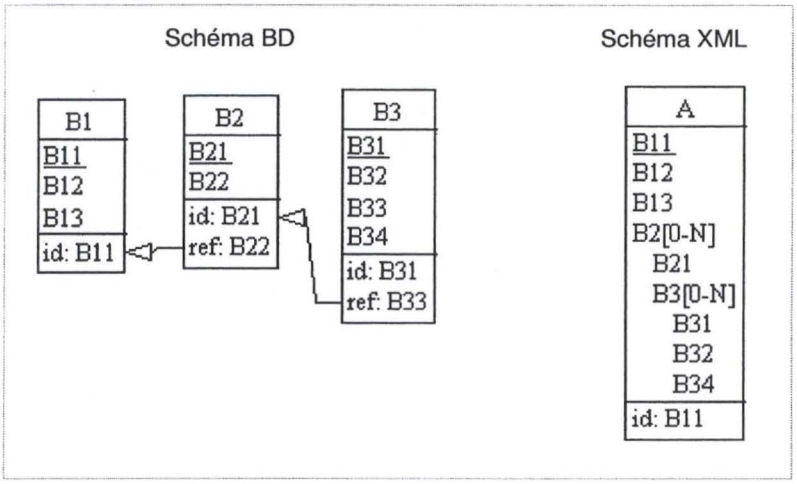


Figure 4.5- Un type d'entité du schéma XML est construit à partir de plusieurs type d'entité du schéma de la base de données.

C'est typiquement une situation obtenue dans le cas de la transformation d'un type d'entité en un attribut. On construit alors une requête par type d'entité Bi du schéma de la base de données. Ces requêtes sont dépendantes les unes des autres puisque certaines sont paramétrées par le résultat d'autres.

La figure suivante illustre l'algorithme du migrateur. Dans le cas 2, il a été particularisé à l'exemple de la figure 4.5.

```
pour chaque collection C du schéma XML
faire  pour chaque type d'entité (TE) E' de la collection C
       faire
              si cas 1 : pour chaque instance du TE E
                             correspondant à E' dans le schéma BD
                         faire  extraction
                                conversion et formatage

              si cas 2 : pour chaque instance b1 du TE B1
                         faire  extraction
                                conversion et formatage
                                pour chaque instance b2 du TE B2
                                     liée à b1
                                faire extraction
                                      conversion et formatage
                                     pour chaque instance b3 du TE
                                          B3 liée à b2
                                      faire extraction
                                            conversion et formatage
```

Figure 4.6- Algorithme du migrateur.

3. Implémentation du migrateur

L'implémentation du migrateur est calquée sur l'algorithme proposé à la figure 4.6. Chaque niveau de traitement est isolé dans une classe ou un type de classe.

Plus précisément, on a construit :

- une classe `migrateur` qui assure le traitement global en construisant tour à tour chaque document XML;
- une classe de type `CollecNom_de_Collection` par collection figurant dans le schéma XML. Celle-ci assure la production complète du document XML lui correspondant;
- une classe de type `SpecificNom_de_TE_BD` par type d'entité figurant dans le schéma de la base de données. Il contient selon les cas, une requête paramétrisée (p. ex. `B2` et `B3`, cf. Fig. 4.5) ou non (p. ex. `B1`, cf. Fig. 4.5). De même, il se limite au traitement de ses instances (p. ex. `B3`, cf. Fig. 4.5) ou il déclenche le traitement d'instances issues d'autres types d'entité (p. ex. `B1` et `B2`, cf. Fig. 4.5);
- une classe `XMLFile` indépendante des structures de la base de données et du ou des document(s) XML.

La figure 4.7 présente ces différentes classes et leurs interactions.

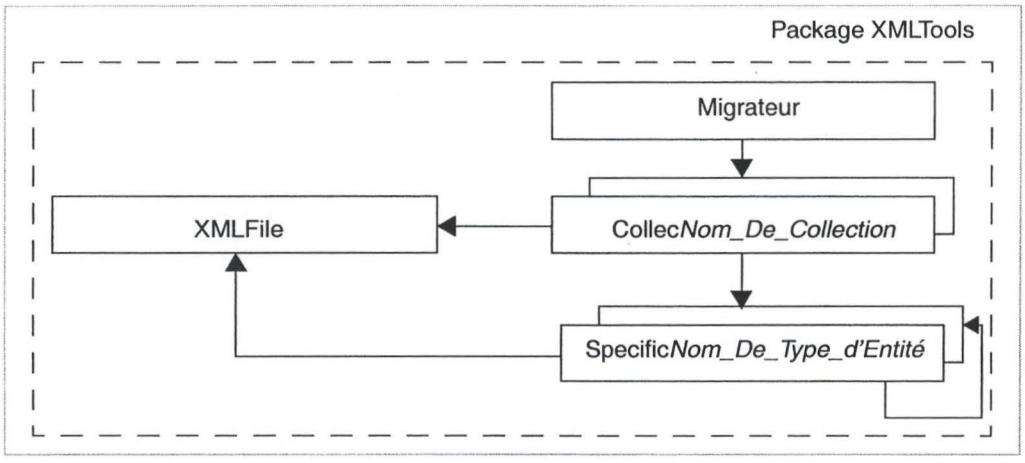


Figure 4.7- Interactions inter-classes constitutives du migrateur.

Légende :

- chaque classe est représentée par un rectangle simple;
- les rectangles doublés représentent un ensemble de classes de même type.
- les flèches noires indiquent qu'une classe en utilise une autre;
- Notons que les classes fournies par le projet InterDB (non représentées) sont utilisées par toutes les classes constitutives du migrateur à l'exception de la classe XMLFile.

Une description de l'interface de chacune de ces classes figure à l'annexe B.

CHAPITRE 5

Mise en pratique de la démarche méthodologique

Ce chapitre décrit la mise en pratique de la démarche méthodologique présentée au chapitre 3.

Il débute par la présentation des outils utilisés, à savoir l'atelier DB-MAIN et ses extensions. Ensuite, la démarche méthodologique du chapitre 3 est rappelée, non sans avoir subi quelques modifications qui découlent des considérations faites au chapitre 4. Les sections suivantes décrivent les différents processus apparaissant dans la démarche méthodologique. Enfin, la dernière section montre comment il est possible de guider l'utilisateur dans le suivi de la méthodologie proposée.

L'objet du chapitre 6 est d'illustrer la démarche méthodologique sur un cas concret. Pour une meilleure compréhension, il est conseillé de le découvrir simultanément au présent chapitre.

1. Outil : DB-MAIN

L'outil CASE[1] DB-MAIN est un outil graphique, programmable, basé sur les transformations de schémas et dédié aux applications d'ingénierie de bases de données. Il est développé dans le cadre du projet de recherche DB-MAIN, à l'Institut d'Informatique des FUNDP à Namur.

1.1. Présentation générale

L'outil CASE DB-MAIN[2] peut être considéré comme un atelier d'outils offrant un support aux activités d'ingénierie ou comme un environnement de développement d'outils.

Principales caractéristiques

- le modèle de spécification de DB-MAIN est le modèle GER, un modèle large offrant un support de représentation de schémas à différents niveaux d'abstraction (physique, logique et conceptuel) et selon différents paradigmes (Entité-Association, Objet, ...).
- l'interface de DB-MAIN offre des vues multiples des spécifications (vues graphiques et textuelles). Tous les opérateurs peuvent être activés quelque soit la vue dans laquelle l'objet est visualisé. Les modifications faites dans une vue sont immédiatement propagées au reste du schéma.
- DB-MAIN offre trois niveaux d'opérateurs de transformations de schémas : les transformations élémentaires (appliquer la transformation T à un objet C), les transformations globales (appliquer la transformation T à tous les objets satisfaisant la condition P) et les transformations conduites par un modèle (appliquer les transformations nécessaires pour que le schéma soit conforme à un modèle donné).
- DB-MAIN offre différents analyseurs de texte.
- DB-MAIN est extensible en développant des méthodologies (via le langage de développement de méthode MDL) ou en ajoutant de nouveaux concepts et de nouvelles fonctionnalités à la boîte à outils de base (via le langage Voyager)
- DB-MAIN offre des assistants de support à des classes spécifiques de problèmes tels que l'assistant de transformation, l'assistant d'analyste, ...
- DB-MAIN permet de tracer le processus d'ingénierie via l'enregistrement de l'historique.

---

1. CASE (**C**omputer **A**ided **S**oftware **E**nvironnement).
2. Une description complète de l'outil est disponible sur le site Internet de DB-MAIN (cf. [11]). Le site contient, entre autre, une version de démonstration de la version 5.1 de l'outil DB-MAIN et le manuel d'utilisation (cf. [12]).

1.2. Voyager

DB-MAIN permet l'ajout de fonctionnalités propres à l'utilisateur. C'est le langage Voyager 2 qui permet de développer des fonctionnalités supplémentaires telles que des générateurs de programmes, des extracteurs, des chargeurs, des transformations complexes, ...

Caractéristiques de Voyager :

- communication avec le repository via des requêtes navigationnelles ou prédicatives;
- les fonctions et les procédures peuvent être récursives;
- gestion automatique de la mémoire (liste extensible dynamiquement, garbage collector);
- fonctions textuelles puissantes permettant de construire des fonctions de parsage;
- tous les outils de base de DB-MAIN sont disponibles à partir de Voyager;
- code pré-compilé et interprété;
- les procédures Voyager peuvent apparaître dans un menu de DB-MAIN exactement comme les outils de base;

Vous trouverez une description complète de Voyager en [13].

1.3. Historique des activités d'ingénierie

DB-MAIN peut enregistrer dans un fichier, l'historique des activités d'ingénierie qui y ont été menées par le développeur ou par l'outil lui-même (dans le cas, par exemple, de l'exécution d'une fonction Voyager). Ce fichier, baptisé **journal**, contient l'historique des activités, décrit dans un langage formel qui peut être traité automatiquement (parser ou rejouer, via l'utilisation de fonctions Voyager) ou lisible par l'homme.

1.4. Le langage MDL

Pour faciliter le suivi d'une démarche méthodologique, l'outil CASE DB-MAIN offre une guidance par le recours à une méthode développée dans le langage MDL[1]. Quand on travaille avec une méthode MDL, l'outil DB-MAIN indique à l'utilisateur ce qu'il doit faire et comment il doit le faire.

Dans le contexte de MDL, chaque étape de la démarche méthodologique est appelée **processus**, qu'elle soit complexe ou élémentaire. Un processus peut être complètement automatique ou manuel. En fonction du type de processus en cours, la liberté de l'utilisateur est plus ou moins brimée. En d'autres mots, à certain moment, la méthode impose à l'utilisateur d'exécuter une tâche précise sans lui laisser aucune liberté; tandis qu'à d'autre moment, elle permet l'utilisation d'un certain nombre de fonctionnalités de l'atelier.

Chaque processus est considéré comme la transformation de **produits**. Un produit est un document qui est utilisé, produit ou modifié durant le suivi de la méthode. Typiquement, dans le cas des bases de données, il consiste en un schéma représentant des structures de données ou en un document texte.

1. MDL (**M**ethod **D**escription **L**anguage). Pour plus de renseignements, consultez [14].

Le langage MDL permet la définition de **modèle de produits**. Un modèle de produits définit une classe de produits en déclarant les composants qui sont admis dans le produit et les contraintes qui doivent y être satisfaites. Par exemple, dans le cas qui nous occupe, nous avons défini deux modèles de produit : le modèle L3 (cf. Fig. 4.2 Page 59) et le modèle XML (cf. Section 3.4. Page 50). Ces modèle définissent des **produits de type** L3 et d'autres de type XML. A la création d'un produit, on lui affecte un type. La vérification qu'un produit est conforme au modèle lui correspondant se déclenche dans certains cas (voir plus loin).

Chaque processus complexe est un enchaînement de processus élémentaires. Le langage MDL permet de définir des conditions sur ces enchaînements. Par exemple, il peut imposer que l'enchaînement des processus élémentaires soit séquentiel ou qu'il soit laissé au libre arbitre de l'analyste, ou encore qu'un processus élémentaire soit exécuté jusqu'à ce qu'une condition soit vérifiée. Par exemple, on peut imposer qu'un schéma soit modifié jusqu'à ce qu'il se conforme à un modèle.

Un processus élémentaire peut consister, entre autre, en :

- le déclenchement d'une fonctionalité de l'outil DB-MAIN tel que créer un nouveau schéma, copier un schéma, mener une transformation globale, ...;
- l'utilisation d'une **boîte à outils** c'est-à-dire un ensemble de fonctionnalités qui sont mises à la disposition de l'utilisateur. Par exemple, si l'objectif du processus est de construire un schéma, on peut mettre à la disposition de l'utilisateur une boîte à outils rassemblant les fonctionnalités de création, modification et suppression d'un type d'entité, d'un type d'association, d'un attribut, ... que l'utilisateur peut exploiter à son gré;
- le déclenchement d'une procédure écrite en Voyager;

Une méthode MDL vérifie automatiquement qu'un produit déclaré d'un certain type soit vraiment conforme au modèle sous-jacent. Cette vérification se déclenche entre autre, après l'utilisation d'une boîte à outils, quand l'utilisateur signale qu'il a terminé. L'atelier vérifie que le produit qui vient d'être modifié par cette boîte à outils est conforme au modèle relatif à son type. Par exemple, si on offre une boîte à outils permettant de construire un schéma de type L3 représentant la structure de la base de données, à la fin de sa construction, l'atelier vérifie que le schéma est conforme au modèle L3. Si ce n'est pas le cas, l'utilisateur ne peut pas passer au processus suivant et doit continuer à utiliser la boîte à outils afin de se conformer au modèle L3;

2. Démarche méthodologique

Avant de décrire la mise en pratique de la démarche méthodologique présentée au chapitre 3 (cf. Fig. 3.15), il convient de tenir compte d'une contrainte, imposée au chapitre 4 relative au choix du module d'accès à la base de données (module InterDB). Ce choix implique que la base de données soit de type relationnel ou COBOL. De plus, les structures de la base de données décrites au niveau logique doivent constituer un schéma conforme au modèle L3 (cf. Fig. 4.2);

Ceci nous mène à la démarche méthodologique suivante :

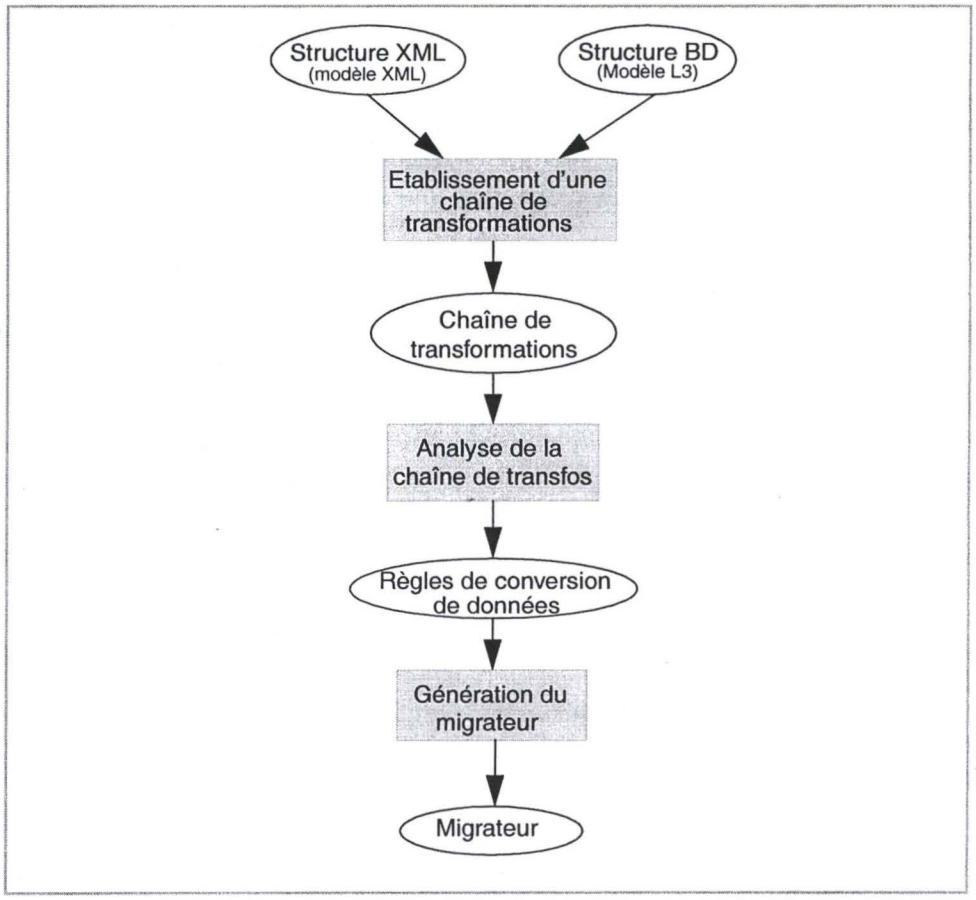


Figure 5.1- Démarche méthodologique.

Il est important de remarquer que la démarche méthodologique ne tient pas compte de l'étape de construction du module InterDB. Dans les faits, elle est construite par les membres de l'équipe InterDB selon une méthodologie qui leur est propre. Pour être complet, il faudrait insérer cette étape dans notre méthodologie.

A ce stade-ci, nous supposons qu'on dispose de :

- la structure de la base de données exprimée sous forme d'un schéma conforme au modèle L3;
- la structure du ou des document(s) XML exprimée sous forme d'un schéma conforme au modèle XML.

Le reste de ce chapitre est organisé de manière à ce que chaque processus de la démarche méthodologique fasse l'objet d'une section qui lui est propre.

3. Etablissement d'une chaîne de transformations

L'enjeu de cette étape est d'aboutir à une chaîne de transformations telle que la chaîne de mappings structurels transforme le schéma de la base de données en celui représentant la structure du ou des document(s) XML.
La chaîne de mappings d'instance correspondante permet de déduire les règles de conversion de données.

Cette étape reste principalement manuelle. La chaîne de transformations doit être déterminée par l'analyste à partir des connaissances sémantiques qu'il a de la structure de la base de données et de celle du ou des document(s) XML.

Cependant, comme nous l'avons signalé dans le chapitre 3, il peut être aidé :

- en disposant d'une palette d'opérateurs de transformation (complexe ou élémentaire) entièrement spécifiés;
- en disposant d'un outil implémentant ces opérateurs de transformation, qui permet de les appliquer au gré des besoins sur un schéma.

La première de ces conditions est remplie. Il suffit pour s'en convaincre de consulter [15]. Ces opérateurs de transformation sont implémentés dans l'atelier DB-MAIN

L'atelier DB-MAIN offre une large palette d'opérateurs de transformation. Pour le développement de notre prototype, nous avons restreint le choix aux seuls opérateurs de transformation suivants :

Opérateur de transformation permis
Création d'une collection
Destruction d'un attribut
Destruction d'un type d'entité
Destruction d'un groupe
Renommage d'un attribut
Renommage d'un type d'entité
Modification de la cardinalité d'un attribut simple
Transformation d'un attribut simple et multivalué en une liste d'attributs simples et monovalués
Transformation d'un type d'entité en attribut

Figure 5.2- Opérateurs de transformation supportés par notre prototype.

Le choix des opérateurs de transformation a été guidé par les considérations suivantes :

- Le modèle L3 n'accepte pas les collections; en revanche, le modèle XML les accepte. Il fallait donc pouvoir supporter la création de ces dernières;
- Les documents XML peuvent ne contenir qu'une partie des données de la base de données. Il faut donc pouvoir détruire des attributs et des types d'entité lorsqu'ils représentent des données non destinées à figurer dans les document(s) XML;
- Les groupes donnent lieu à des attributs dans les document(s) XML. On peut préférer que les attributs constitutifs de ces groupes correspondent plutôt à des éléments du ou des document(s) XML. Il est donc nécessaire de permettre la destruction de groupe;
- Les constructions du schéma correspondantes à des types d'élément (attribut ou type d'entité) doivent porter un nom unique dans le schéma. Pour se conformer à cette contrainte, il faut pouvoir les renommer ;
- Dans le modèle XML, la cardinalité maximale d'un attribut simple doit être égale à 1 ou à N. Pour s'y conformer, on a le choix entre deux alternatives : la modifier purement et simplement ou, si la cardinalité maximale n'est pas trop grande, il est préférable de transformer l'attribut multivalué en une liste d'attributs monovalués. C'est pourquoi ces opérateurs de transformations sont admis;
- Il est nécessaire de disposer d'une transformation de "hiérarchisation" de schéma. Par exemple, si la base de données est de type relationnel, le schéma décrivant sa structure ne contient que des attributs de premier niveau. Il faut disposer d'une transformation permettant d'obtenir des attributs de niveau plus élevé de manière à exploiter les principes de hiérarchie de XML. La transformation d'un type d'entité en attribut permet de hiérarchiser le schéma.
  Concrètement, dans l'atelier DB-MAIN, elle n'apparaît pas sous cette forme. En réalité, elle est la combinaison de deux autres opérateurs de transformation, à savoir l'opérateur de transformation d'un groupe en un type d'association et ensuite, la transformation d'un type d'entité en attribut.

Bien sûr, il n'est pas suffisant d'établir une chaîne de transformations adéquate. Il faut également pouvoir la stocker en vue d'un traitement ultérieur.
L'atelier DB-MAIN offre une fonctionnalité d'enregistrement des activités d'ingénierie menées en son sein. Il suffit à l'utilisateur de déclencher cette fonctionnalité au début de l'étape. Il en résulte un fichier décrivant les différentes opérations qui ont été appliquées. Ce fichier est exprimé dans une syntaxe formelle. Il peut donc faire l'objet d'un traitement automatisé tout en restant lisible par l'homme.

Concrètement, cette seconde étape se concrétise par la démarche suivante :

- copie du schéma de la base de données afin de garder une trace de l'original;
- activation de la fonctionnalité d'enregistrement automatique de l'historique des activités dans l'atelier DB-MAIN;
- appliquer les différents opérateurs de transformations admis jusqu'à obtenir le schéma décrivant la structure du ou des document(s) XML;
- enregistrer dans un fichier, l'historique des activités menées dans l'atelier depuis l'activation de la fonction d'enregistrement de l'historique.

4. Analyse de la chaîne de transformations

Conceptuellement, déduire les règles de conversion de données à partir de la chaîne de transformations établie à l'étape précédente ne présente pas de réelles difficultés. En effet, puisque les opérateurs de transformation sont formellement spécifiés, la chaîne de mappings d'instance est entièrement définie. C'est celle-là même qui correspond aux règles de conversion de données.

Cependant, pour construire le migrateur, nous avons besoin d'un accès rapide à l'information relative à une construction du schéma de la base de données bien particulière.

C'est ici que réside le problème : déduire les transformations menées sur une construction bien particulière nécessite d'analyser l'entièreté du journal. Le journal va devoir être passé en revue un grand nombre de fois durant la construction du migrateur.

Il semble donc plus performant de traiter intégralement le journal et de stocker l'information de manière à ce qu'elle soit rapidement et facilement accessible. Il faut garder à l'esprit que l'accès à cette information se fera à partir d'une construction particulière. Voici donc l'enjeu de cette étape.

Les informations vont être stockées dans ce que nous appelons des **méta-propriétés**.
Le repository de l'atelier DB-MAIN (accessible et modifiable par le langage Voyager) offre la notion de **méta-objet**. Un méta-objet est une classe d'objets regroupant les objets d'un même type (ex de méta-objet : classe des types d'entité, classe des types d'attribut simple, ...).
Une méta-propriété est une propriété dynamique qui est attachée à un méta-objet (ex de méta-propriété définie sur la classe des types d'entité : `nbre_att` qui est une propriété de type `integer` et qui, à chaque type d'entité précis associe le nombre d'attributs de ce type d'entité). Les méta-propriétés peuvent être créées et modifiées manuellement via des boîtes de dialogue offertes dans l'atelier ou à partir de procédures Voyager.

Le traitement du journal va s'effectuer en deux temps :

- création des méta-propriétés destinées à contenir l'information du journal;
- mise à jour de ces méta-propriétés au fur et à mesure du traitement du journal.

Le reste de cette section consiste en la description de ces étapes et des conséquences qui découlent de cette manière de procéder.

4.1. Création des méta-propriétés

Nous allons créer une méta-propriété par opérateur de transformation supporté par notre prototype (cf. Fig. 5.2). Par exemple, à l'opérateur de destruction d'un type d'entité est associée une méta-propriété baptisée DEL définie sur le méta-objet correspondant à la classe des types d'entité et prenant des valeurs réelles. Par convention, si la valeur de la méta-propriété DEL pour un type d'entité donné est nulle, cela signifie que le type d'entité n'a pas été soumis à l'opérateur de destruction.

La figure suivante liste les méta-propriétés associées à chaque opérateur de transformation admis ainsi que l'interprétation de la valeur qui leur est associée.

Transformations permises	**Nom et type de la méta-propr.**	**Convention sur la valeur de la méto-propriété.**
Destruction d'un type d'entité, d'un attribut ou d'un groupe;	DEL - numeric	DEL = 1 si l'objet a été détruit; = 0 sinon.
Renomage d'un type d'entité ou d'un attribut;	REN - varchar	REN = nouveau nom si il y a eu une renomage; ancien nom sinon.
Transformation d'un attribut simple multi-valué en liste d'attributs simples mono-valués	MTL - numeric	MTL = 1 si l'attribut a été transformé en liste de mono, 0 sinon. + Mise à jour de la m.p. DEL
Modification de la cardinalité d'un attribut simple	CARD - varchar	CARD = cardinalité sous forme i-j si il y a eu modification; 0 sinon.
Transformation d'un type d'entité en attribut	EST_CONTENU - numéric CONTIENT- list	EST_CONTENU = référence au TE dans lequel le TE est désormais contenu, 0 sinon; CONTIENT = liste des références des types d'entités contenus dans ce type d'entité;

Figure 5.3- Description des méta-propriétés et des conventions prises sur leur valeur.

4.2. Traitement du journal

Le journal est un fichier constitué d'enregistrements, à raison d'un enregistrement par transformation opérée sur le schéma de la base de données. Plus précisément, chaque enregistrement contient l'information relative à

- l'opérateur de transformation qui a été appliqué;
- la construction du schéma sur laquelle l'opérateur a été appliqué;
- le ou les constructions résultantes.

Le stockage des informations relatives à chaque enregistrement se fait en mettant à jour la méta-propriété associée à l'opérateur de transformation (cf. Fig. 5.3) pour la construction sur laquelle la transformation a été appliquée.

Par exemple, si on travaille sur un schéma qui contient un type d'entité de nom CLIENT et que le journal contient un enregistrement relatant le fait que ce type d'entité CLIENT a été soumis à la transformation de destruction, la méta-propriété DEL pour le type d'entité CLIENT se verra affecter la valeur 1.

La création des diverses méta-propriétés ainsi que le traitement du journal se fait via des procédures Voyager.

En conclusion, les informations du journal sont associées à chaque construction du schéma de la base de données et sont facilement accessibles à partir d'une construction donnée.

4.3. Contrainte résultant de cette démarche

Cette démarche sous-entend que chaque enregistrement du journal décrit une transformation appliquée sur une construction du schéma de la base de données. Autrement dit, le cas d'une transformation appliquée sur une construction n'apparaissant pas directement dans le schéma de la base de données mais résultant d'autres transformations intermédiaires ne peut pas être traité de cette façon. En effet, il n'est pas possible de stocker cette information dans le schéma de la base de données puisque la construction n'y existait pas. Dans la pratique, nous devons travailler sous l'hypothèse que chaque transformation est appliquée sur une construction apparaissant telle quelle dans le schéma de la base de données.

5. Génération du migrateur

Les étapes précédentes ont permis de disposer, au sein de l'atelier DB-MAIN :

- du schéma représentant la structure de la base de données enrichi des transformations appliquées sur chaque construction du schéma;
- du schéma représentant la structure du ou des document(s) XML.

Nous disposons en outre du langage Voyager qui permet d'accéder au repository de l'atelier DB-MAIN.

Le générateur du migrateur est très simple. Pratiquement, il est constitué d'une procédure par type de classe à construire.

Par exemple, la classe `Migrateur` est chargée d'assurer la production de tous les documents XML . Pour chaque collection présente dans le schéma XML, elle crée un instance de la classe `CollecNom_de_Collection` lui correspondante en utilisant son constructeur et elle lui applique la méthode `CollTreatment`.

Pour la génération de cette classe , on procède comme suit :

- déterminer l'ensemble des collections présentes dans le schéma XML. Ceci est tout-à-fait possible et facile puisque le langage Voyager peut accéder au repository de l'atelier DB-MAIN;
- pour chaque collection, générer le code Java correspondant.

La génération de cette classe est particulièrement simple car elle ne consiste qu'en une requête simple. De plus, elle n'a pas besoin d'accéder à l'information contenue dans les méta-propriétés. Cependant, la génération des autres classes se fait selon les mêmes principes à ceci près que les requêtes peuvent être un peu plus élaborées. Par ex : déterminer le type d'entité du schéma de la base de données dont la structure correspondante dans le schéma cible est un type d'entité donné, ...

6. Implémentation de la méthodologie

La démarche méthodologique que nous avons décrite comporte un certain nombre d'étapes élémentaires ou complexes qui doivent être exécutées selon un ordre bien défini. Certaines de ces étapes semblent naturelles dans l'objectif qu'on poursuit, alors que d'autres liées aux outils utilisés, n'ont pas d'équivalents conceptuels et ne constituent que des aménagements techniques. Ces dernières peuvent faire l'objet d'oubli de la part de l'analyste. Malheureusement, l'oubli de certaines de ces étapes peut avoir des conséquences très lourdes, allant même jusqu'à devoir recommencer le processus entièrement. C'est pourquoi notre méthodologie a été implémentée dans le langage MDL. De cette façon, l'utilisateur est guidé tout au long du processus de construction du migrateur au sein même de l'atelier DB-MAIN.

En règle générale, l'organisation des différents processus de l'implémentation de la méthodologie est calquée sur celle des étapes établies précédemment (cf. Fig. 5.1). Les seules modifications qui apparaissent sont liées à des contraintes techniques. La figure 5.4 montre la représentation graphique de la méthode MDL telle qu'elle apparaît à l'utilisateur lors de son déroulement dans l'atelier DB-MAIN.

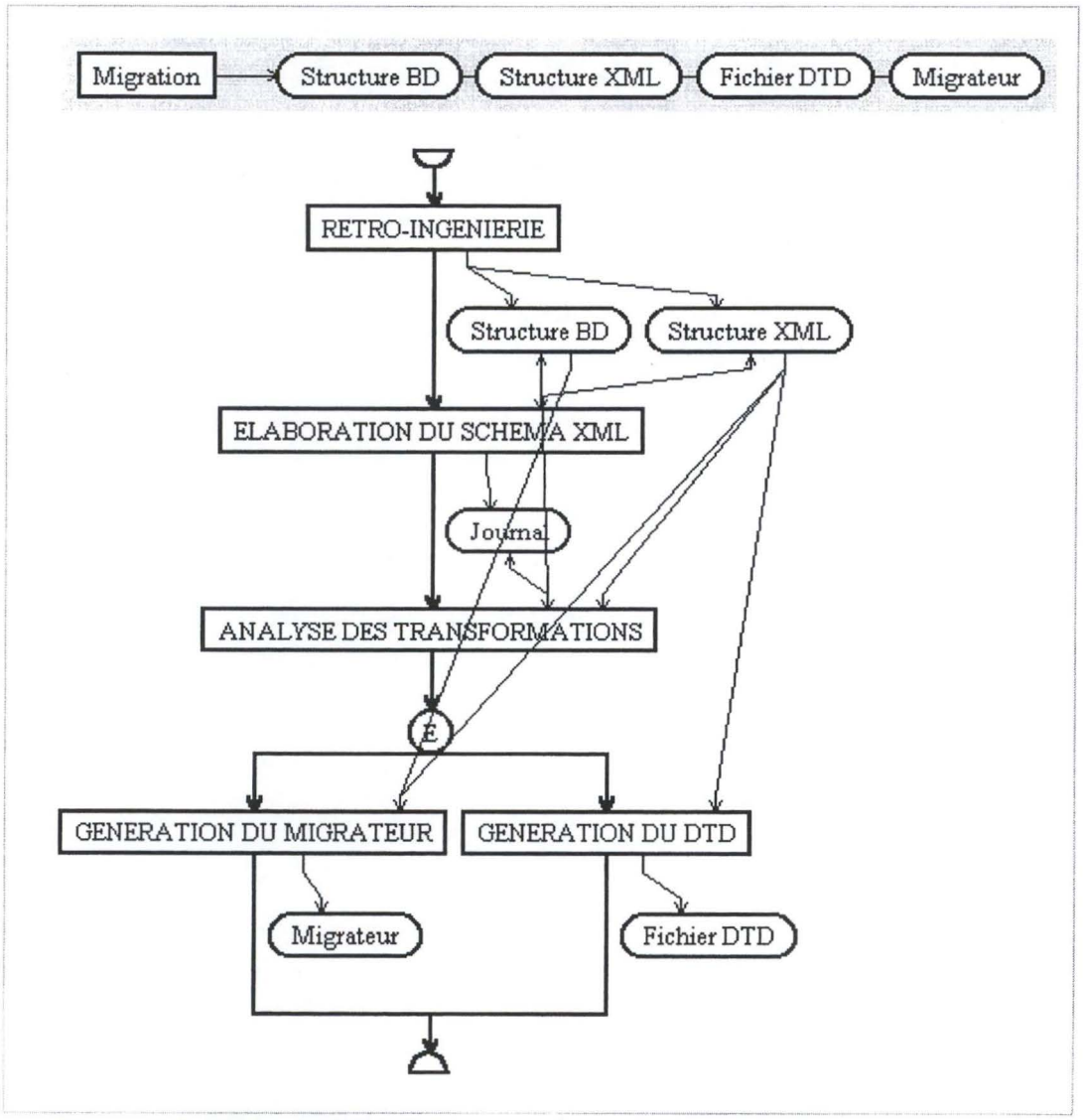


Figure 5.4- Méthode MDL de support à la migration de données d'une base de données vers un ou plusieurs documents XML.

La première étape consiste à rechercher la structure de la base de données. Elle est plus ou moins complexe en fonction de la documentation disponible concernant cette base de données. Au terme de cette étape, nous disposons non seulement du schéma de la base de données conforme au modèle L3 (cf. Fig. 4.2), mais également d'un deuxième schéma que nous avons anticipativement nommé, schéma XML, qui n'est, à ce stade-ci, qu'une copie du schéma de la base de données.

La deuxième étape consiste à modifier le schéma qu'on a appelé schéma XML afin qu'il coïncide avec celui représentant la structure du ou des document(s) XML à produire. On procède en lui appliquant une série de opérateurs de transformation disponibles au sein de l'atelier DB-MAIN.

La troisième étape consiste à enregistrer dans un fichier, l'historique de certaines activités qui ont été menées dans l'atelier DB-MAIN. Il s'agit ici des diverses transformations appliquées à l'étape précédente. De plus, cette étape prend en charge l'analyse automatique des transformations et le stockage des informations dans des méta-propriétés au sein du schéma de la base de données.

La quatrième étape prend en charge la génération du migrateur. Cette étape est complètement automatique.

Nous avons ajouté une cinquième étape par rapport à celles établies précédemment. Elle consiste en la génération des DTDs sur base de leur représentation dans le schéma XML.

CHAPITRE 6

Etude de cas

Ce chapitre présente un cas concret d'utilisation des outils et de la méthodologie que nous avons élaborée. Il présente, d'abord, le contexte d'utilisation en décrivant les structures de la base de données et des documents XML à produire. Ensuite, il montre comment chaque étape se concrétise. Enfin, il présente un extrait des documents XML obtenus. Cette étude de cas utilise la méthode de guidance MDL décrite au chapitre précédent.

1. Présentation du cas

Une entreprise spécialisée dans la vente de matériel informatique souhaite utiliser le format XML pour envoyer à ses services d'expédition et de facturation, les informations relatives aux commandes en cours.

La figure 6.1 présente la structure de la base de données de cette entreprise, décrite à un niveau logique.

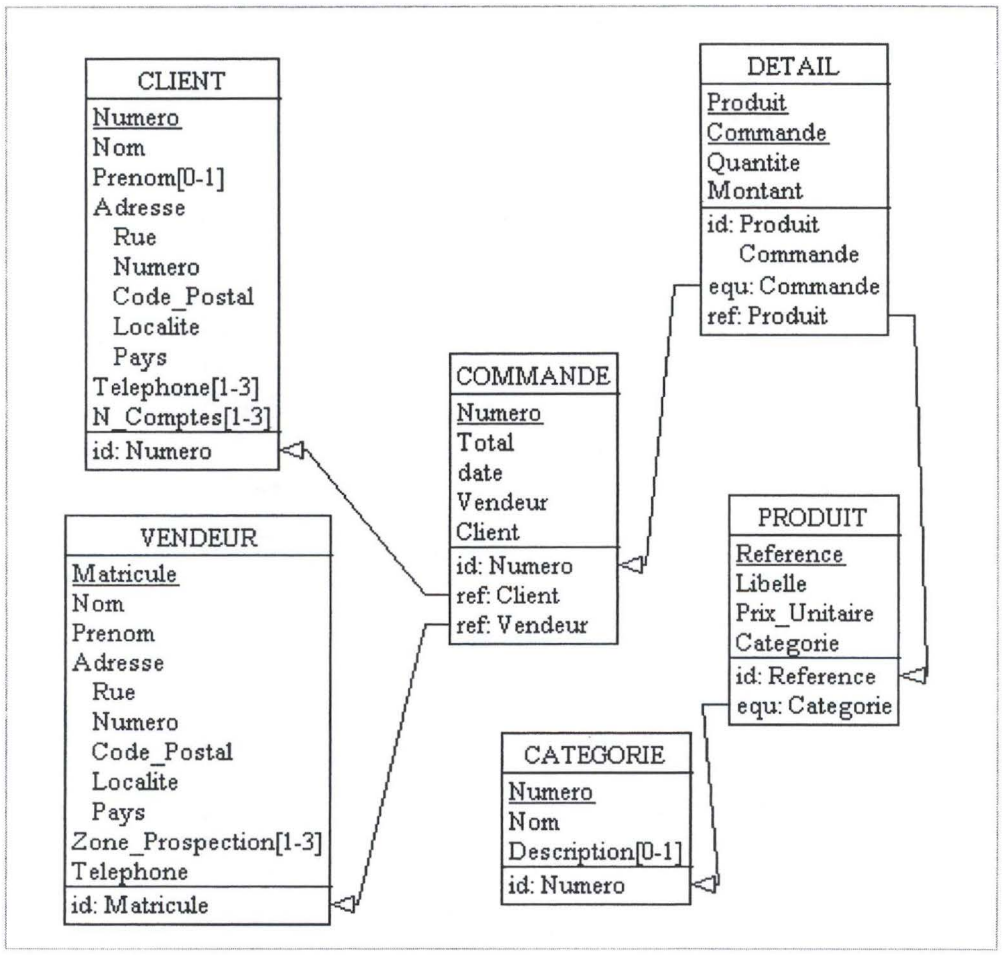


Figure 6.1- Schéma logique de la base de données.

Les documents XML à produire doivent contenir l'ensemble des informations relatives aux commandes qui ont été passées, c'est-à-dire :

- les détails de chaque commande;
- les informations complètes concernant les différents produits commandés, y compris la catégorie de laquelle ils sont issus afin de retranscrire le descriptif complet du produit sur la facture;
- les informations relatives au client;

Supposons que pour éviter les redondances, on décide de séparer les informations relatives aux clients de celles propres aux commandes dans deux documents XML.

La figure 6.2 illustre la structure des documents XML à produire.

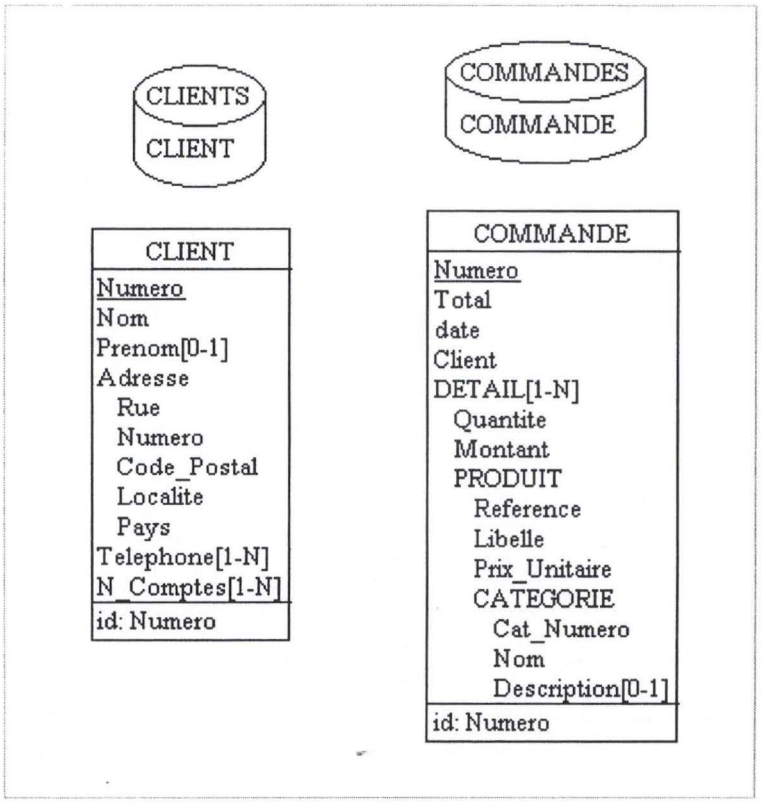


Figure 6.2- Structure des documents XML à produire.

Nous allons décrire les différentes étapes permettant d'aboutir à la construction d'une application de production de tels documents XML. Ce processus est facilité par le recours à la méthode MDL décrite à la section précédente (cf. Fig. 5.4).

Au début du projet, l'atelier DB-MAIN se présente tel illustré à la figure 6.3.

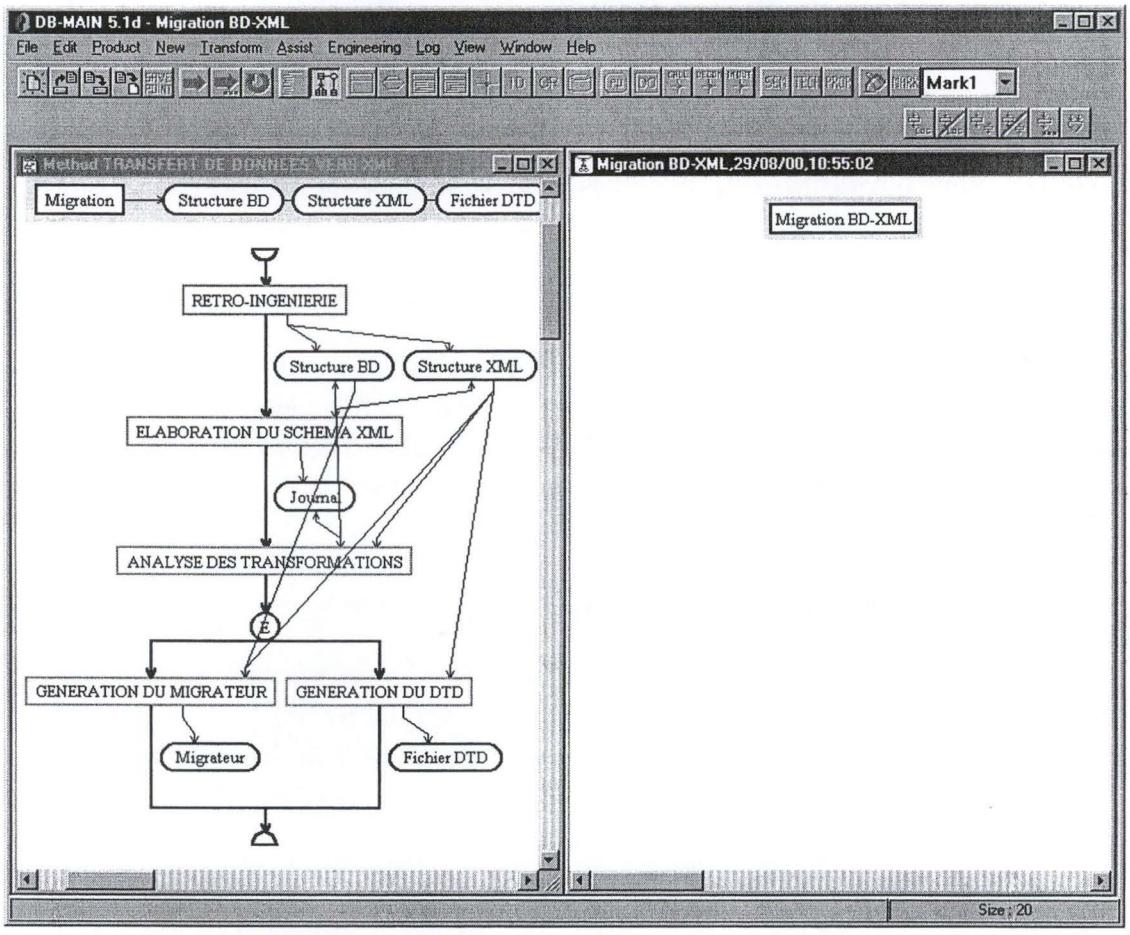


Figure 6.3- Aperçu de l'atelier DB-MAIN au début du projet de migration.

La fenêtre de gauche sert à indiquer le degré d'avancement dans la méthode. La fenêtre de droite, quant à elle, contient l'historique du projet en cours, c'est-à-dire toutes les informations relatives aux processus dont l'exécution est terminée. Elle indique également leur interaction avec les différents produits existants.

2. Rétro-ingénierie

L'objet de cette étape est de retrouver une spécification complète de la structure de la base de données. Dans le cadre de ce mémoire, nous n'en dirons pas plus.

Au terme de cette étape, nous disposons de la structure de la base de données décrite à un niveau logique. De plus, pour des raisons liées à l'utilisation du module fourni par le projet InterDB, nous devons imposer que ce schéma soit conforme au modèle L3 (cf. Fig. 4.2).

Cette étape est elle-même constituée de deux étapes. Elles sont visibles sur la figure 6.4.

La première sous-étape, baptisée ELABORATION DU SCHEMA BD, correspond au processus complet de rétro-ingénierie aboutissant à l'élaboration du schéma représentant la structure de la base de données. Au terme de cette étape, nous disposons d'un schéma baptisé COMMANDES conforme au modèle L3. C'est en partie ce que montre l'historique dans la fenêtre de droite.

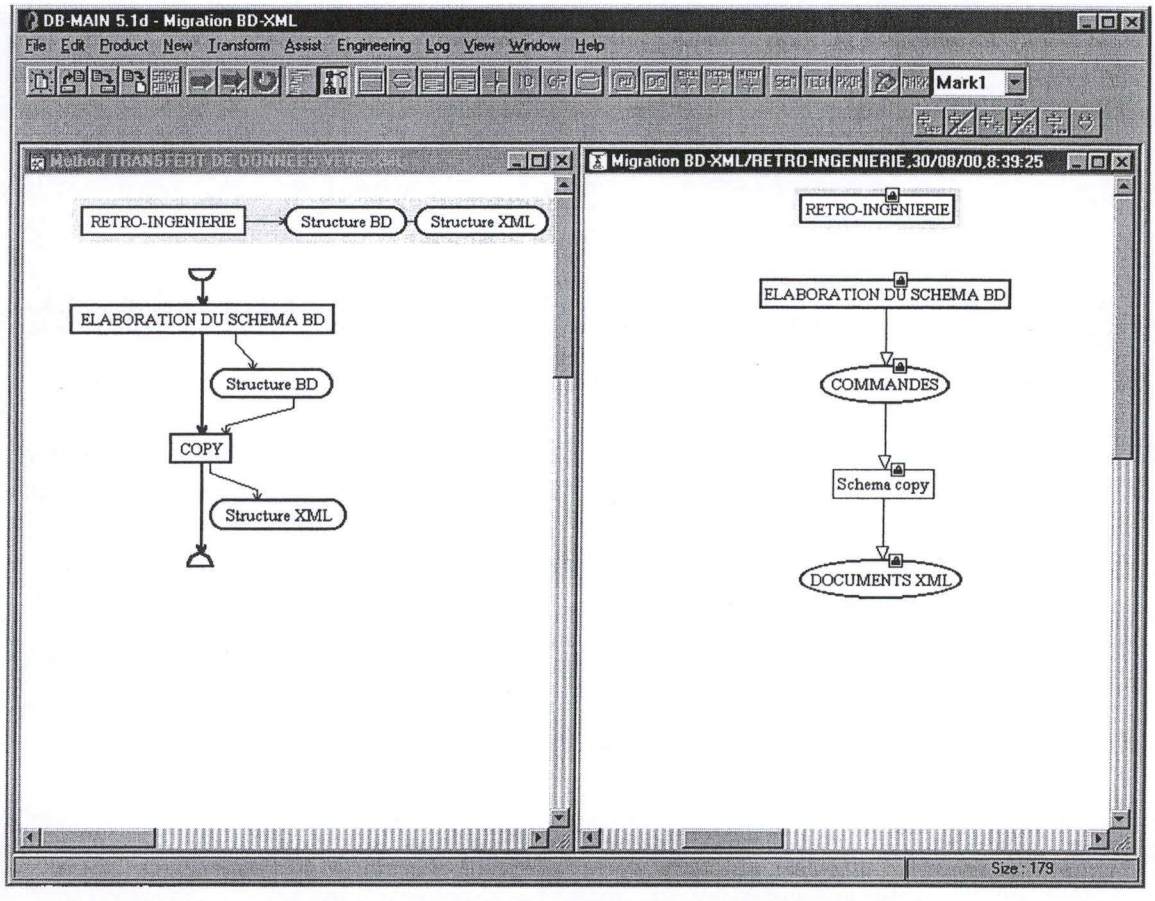


Figure 6.4- Aperçu de l'atelier DB-MAIN au terme de l'étape nommée RETRO-INGENIERIE.

La seconde sous-étape, désignée par le terme COPY, est destinée à préparer à l'étape suivante. Elle consiste à copier le schéma COMMANDES dans un autre schéma, que nous avons choisi d'appeler ici DOCUMENTS XML. C'est sur cette copie que nous allons travailler de manière à garder une trace du schéma décrivant la structure de la base de données.

3. Elaboration du schéma XML

L'objectif de cette étape est d'établir une chaîne de transformations qui (ou plutôt la chaîne de mappings structurels correspondante) transforme le schéma décrivant la structure de la base de données en le schéma décrivant la structure du ou des document(s) XML à produire.

Cette étape, comme l'illustre la figure 6.5, se décompose en trois étapes plus élémentaires.

La sous-étape PRE_TRAITEMENT prend, entre autres, en charge la création des méta-propriétés destinées à contenir les informations relatives aux transformations (cf. 5.3). Concrètement, elle est implémentée par une fonction Voyager qu'il faut simplement déclencher. L'utilisateur n'a pas d'autre alternative.

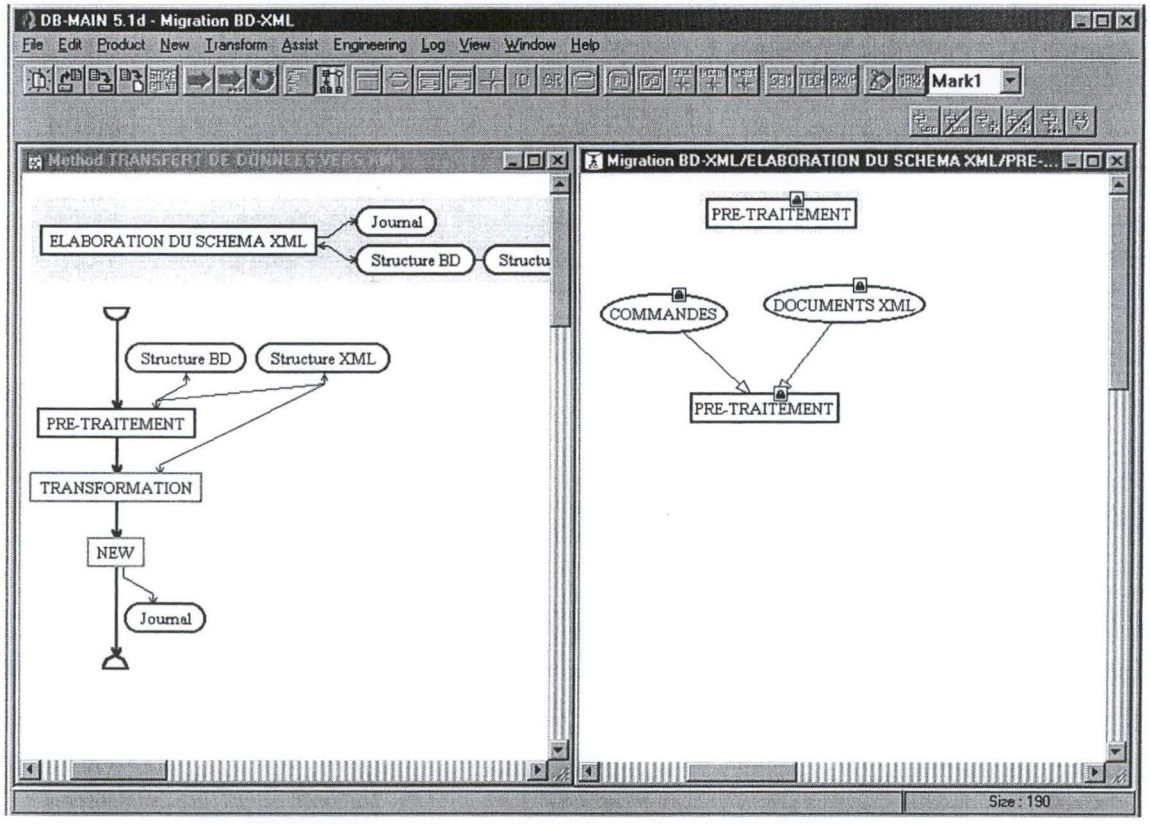


Figure 6.5- Aperçu de l'atelier DB-MAIN à la fin du processus baptisé PRE-TRAITEMENT.

La seconde sous-étape, nommée TRANSFORMATION, consiste à élaborer la chaîne de transformations permettant de passer de la structure de la base de données à celle des documents XML (entièrement spécifiée). Pour atteindre cet objectif, l'utilisateur dispose d'un certain nombre d'opérateurs de transformations (cf. Fig. 5.2) qu'il peut appliquer sur les différentes constructions du schéma DOCUMENTS XML. Ce sont des fonctionnalités du menu "Transform" de l'atelier DB-MAIN qui sont proposées à l'utilisateur.

La figure 6.6 donne un aperçu de l'atelier DB-MAIN au cours de l'exécution de cette étape. Plus précisément, c'est la transformation du type d'entité CATEGORIE en attribut composé du type d'entité PRODUIT qui est représentée sur cette figure. Comme on l'a dit, elle

n'est pas disponible telle quelle dans l'atelier DB-MAIN. Il faut passer par une étape de préparation au cours de laquelle on transforme le groupe référentiel en type d'association. C'est au cours de cette étape préliminaire que le type d'association Categorie a été créé (cf Fig. 6.6).

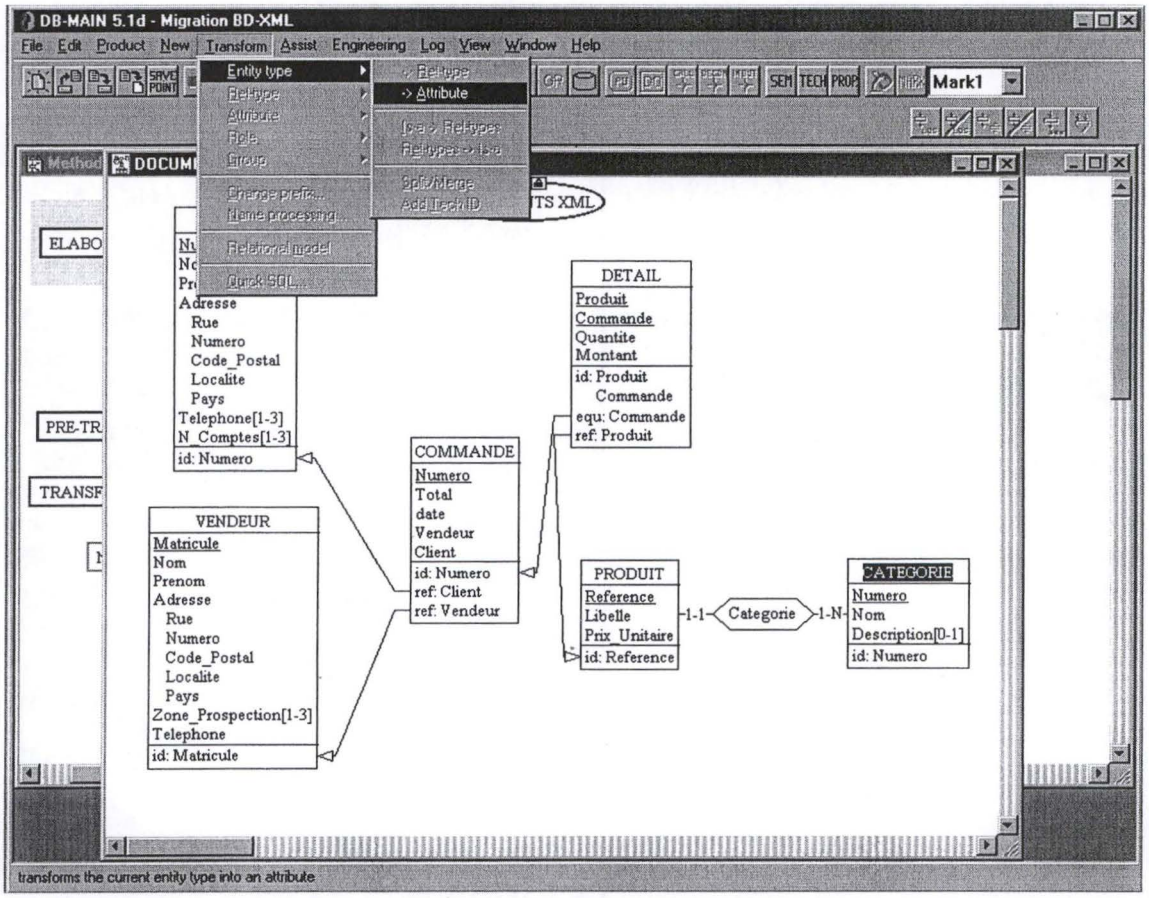


Figure 6.6- Aperçu de l'atelier DB-MAIN au cours de l'exécution du processus TRANSFORMATION.

L'observation de la structure de la base de données (cf. Fig. 6.1) et de celle des deux documents XML à produire (cf. Fig. 6.2) nous permet de déduire les transformations nécessaires à appliquer sur le schéma DOCUMENTS XML :

- le type d'entité VENDEUR du schéma initial n'a pas de correspondant structurel. Il doit donc être détruit;
- le type d'entité CATEGORIE du schéma initial correspond à l'attribut composé du même nom dans le type d'entité COMMANDE du schéma XML. Il y a donc lieu de mener deux transformations : la première transforme le groupe défini sur l'attribut Categorie en type d'association alors que la seconde transforme l'attribut CATEGORIE en attribut composé;
- de même, le type d'entité PRODUIT correspond à l'attribut composé de même nom dans le type d'entité COMMANDE. Il doit être transformé de la même façon que le type d'entité CATEGORIE. C'est également vrai pour le type d'entité DETAIL;
- le groupe référentiel portant sur l'attribut Client doit être détruit. Le cas échéant, il donnerait lieu à un attribut de type IDREF attaché au type d'élément Commande dont les valeurs référenceraient des éléments de type CLIENT contenus dans un autre document XML. La contrainte IDREF ne serait pas respectée et le document XML ainsi produit ne serait pas conforme au DTD correspondant à la figure 6.2;
- la cardinalité maximale des attributs Telephone et N_Comptes doit être modifiée;
- l'attribut Vendeur dans le type d'entité COMMANDE n'a pas de structure lui correspondant dans le schéma XML. Il doit donc être détruit;
- l'attribut Numero dans le type d'entité CATEGORIE doit être renommé. Dans le cas échéant, il y auraient deux attributs de même nom dans le type d'entité COMMANDES dans le schéma XML. Ceci correspond à deux types d'éléments de même nom dans un même DTD, ce qui n'est pas permis.

L'atelier DB-MAIN offre la possibilité de sauvegarder les activités qui ont été menées en son sein. L'activation de cette fonctionnalité est automatique et est déclenchée au début de ce processus TRANSFORMATION.

Lorsque l'utilisateur indique qu'il a terminé cette étape, une vérification de la conformité du schéma DOCUMENTS XML avec le modèle XML est déclenchée automatiquement. Si cette condition n'est pas respectée, l'utilisateur ne peut pas passer au processus suivant et doit faire les démarches nécessaires pour se conformer au modèle XML.

Le dernier processus, baptisé NEW, est une préparation au processus suivant. Il permet de créer un fichier destiné à contenir l'historique des activités menées au sein de l'atelier depuis l'activation de la fonctionnalité d'enregistrement de cet historique. Comme le montre la figure 6.7, nous avons choisi de nommer ce fichier journal.log.

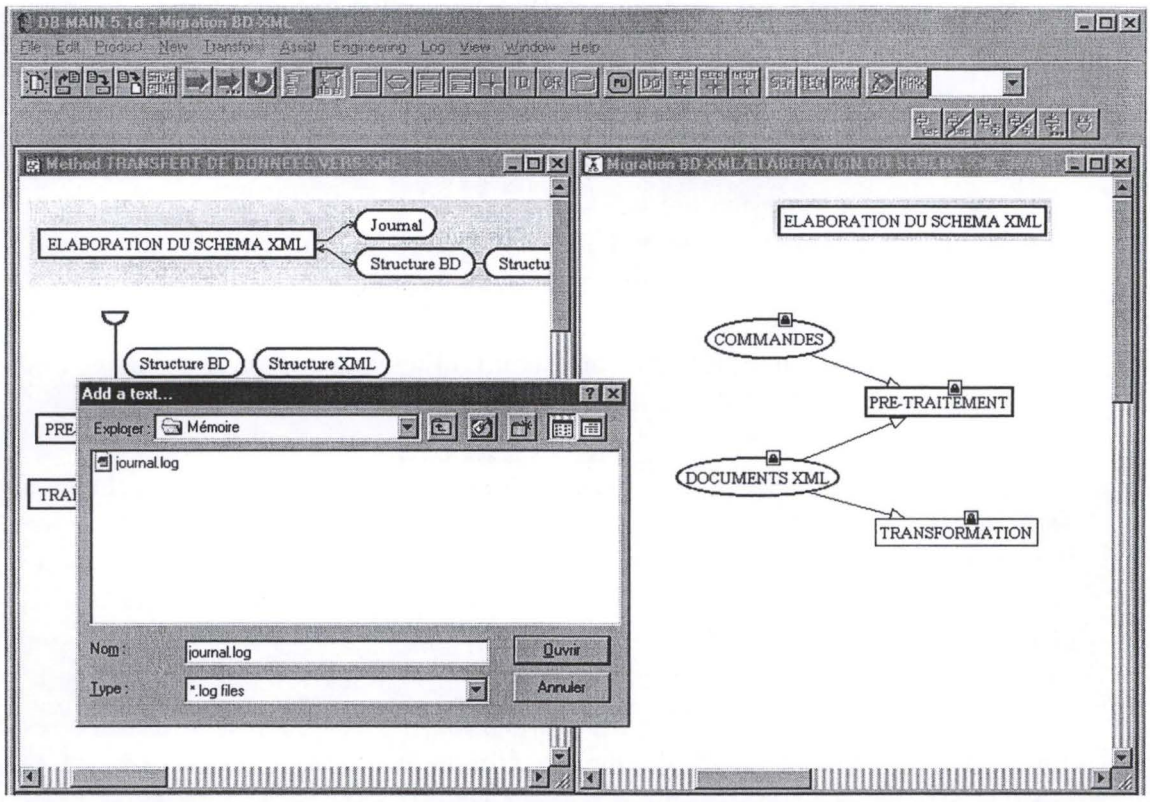


Figure 6.7- Aperçu de l'atelier DB-MAIN au cours de l'exécution de l'étape NEW.

Au terme de cette étape, nous disposons de :

- un schéma COMMANDES conforme au modèle L3;
- un schéma DOCUMENTS XML conforme au modèle XML;
- un fichier journal.log vide;

Avec cette étape se clôture l'intervention de l'analyste. A ce stade du projet, les étapes restantes sont entièrement automatisées.

4. Analyse des transformations

L'objectif de cette étape est d'enregistrer dans le fichier journal.log, la chaîne de transformations élaborée au cours de l'étape précédente. C'est une fonction Voyager qui prend en charge cet enregistrement.

D'autre part, il s'agit d'analyser entièrement le journal afin de stocker les informations qui y sont contenues dans les méta-propriétés du schéma COMMANDES. Ici aussi, l'étape est entièrement implémentée par une fonction Voyager.

Cette étape est donc composée de deux étapes élémentaires, comme l'illustre la figure 6.8.

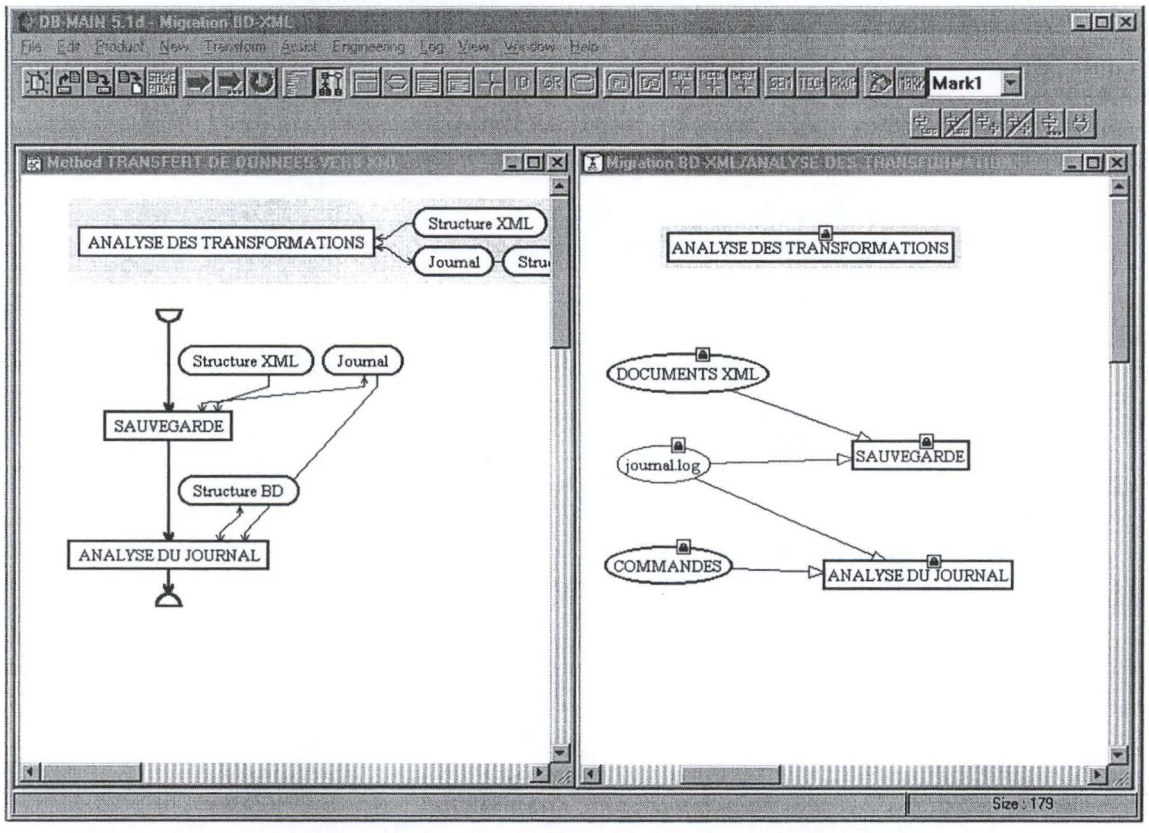


Figure 6.8- Aperçu de l'atelier DB-MAIN au terme du processus ANALYSE DU JOURNAL.

Au terme de cette étape, nous disposons :

- du schéma COMMANDES conforme au modèle L3 et enrichi des informations relatives aux transformations, stockées dans les méta-propriétés;
- du schéma DOCUMENTS XML conforme au modèle XML ;
- du fichier journal.log contenant les informations relatives à la chaîne de transformations.

5. Génération du migrateur et génération du DTD

L'ordre de l'exécution des étapes GENERATION DU DTD et GENERATION DU MIGRATEUR est laissée au choix de l'utilisateur. Ce sont deux étapes entièrement automatisées et implémentées via des fonctions Voyager. La figure 6.9 donne un aperçu de l'atelier à la fin du processus de génération du DTD.

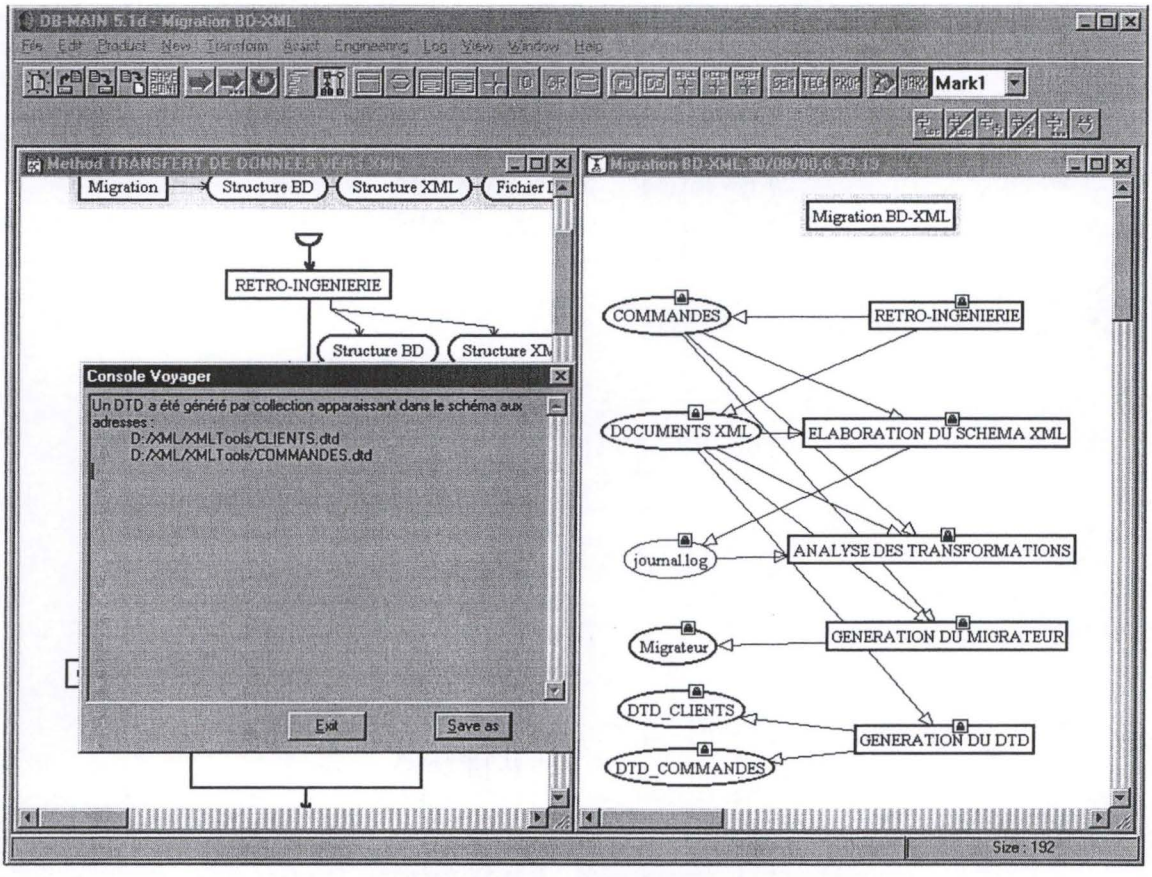


Figure 6.9- Aperçu de l'atelier DB-MAIN au terme du projet.

Enfin, nous disposons maintenant outre des produits disponibles à la fin de l'étape précédente de :

- un certain nombre de fichiers écrits dans le langage java et constituant le migrateur (cf. Page 62). Pour plus de facilités, le générateur produit un fichier de commande permettant de déclencher automatiquement la compilation et l'exécution du migrateur. C'est celui-là même qui est repris dans l'historique;
- un DTD par document XML à produire, identifiés dans l'historique par les noms DTD_COMMANDES et DTD_CLIENTS;

Le processus complet est maintenant arrivé à son terme. On peut observer dans la fenêtre de droite que l'historique contient les produits finaux.

6. Documents XML

La compilation et l'exécution du migrateur permet d'aboutir à la production des documents XML conforme aux exigences initiales. La figure 6.10 illustre un extrait du document XML COMMANDES, tandis que la figure 6.11 montre le DTD relatif à ce même document XML.

```
<?xml version="1.0"?>
<!DOCTYPE COMMANDES SYSTEM "COMMANDES.dtd">
<COMMANDES>
   <COMMANDE Numero ="COMMANDE1">
      <Total>1992441</Total>
      <date>Tue Feb 20 00:00:00 GMT+01:00 2001</date>
      <Client>2</Client>
      <DETAIL>
         <Quantite>1</Quantite>
         <Montant>104353</Montant>
         <PRODUIT>
            <Reference>6</Reference>
            <Libelle>Intel PIII X?on 550 </Libelle>
            <Prix_Unitaire>104353</Prix_Unitaire>
            <CATEGORIE>
              <Numero>2</Numero>
              <Nom>Processeurs</Nom>
              <Description>Tous les processeurs</Description>
            </CATEGORIE>
         </PRODUIT>
      </DETAIL>
      <DETAIL>
         <Quantite>1</Quantite>
         <Montant>88088</Montant>
         <PRODUIT>
            <Reference>10</Reference>
            <Libelle>Ecran Sony 24''</Libelle>
            <Prix_Unitaire>88088</Prix_Unitaire>
            <CATEGORIE >
              <Numero>5</Numero>
              <Nom>Graphiques</Nom>
               <Description>Cartes graphiques, écrans</Description>
            </CATEGORIE>
         </PRODUIT>
      </DETAIL>
</COMMANDE>
[...]
</COMMANDES>
```

Figure 6.10- Extrait du document XML Commandes.xml produit automatiquement par le migrateur.

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT COMMANDES (COMMANDE*)>

<!-- COMMANDE section -->
<!ELEMENT COMMANDE (Total, date, Client, DETAIL+)>
<!ELEMENT Total (#PCDATA)*>
<!ELEMENT date (#PCDATA)*>

<!-- DETAIL section -->
<!ELEMENT DETAIL (Quantite, Montant, PRODUIT)>
<!ELEMENT Quantite (#PCDATA)*>
<!ELEMENT Montant (#PCDATA)*>

<!-- PRODUIT section -->
<!ELEMENT PRODUIT (Reference, Libelle, Prix_Unitaire, CATEGORIE)>
<!ELEMENT Reference (#PCDATA)*>
<!ELEMENT Libelle (#PCDATA)*>
<!ELEMENT Prix_Unitaire (#PCDATA)*>

<!-- CATEGORIE section -->
<!ELEMENT CATEGORIE (Numero, Nom, Description?)>
<!ELEMENT Numero (#PCDATA)*>
<!ELEMENT Nom (#PCDATA)*>
<!ELEMENT Description (#PCDATA)*>

<!ATTLIST COMMANDE Numero ID #REQUIRED>
```

Figure 6.11- Document DTD relatif au document XML Commandes.xml.

Conclusion

Nous voici arrivés au terme de ce travail dont l'objectif était de supporter la production de document(s) XML à partir d'une base de données via une application dédiée à cette tâche.

Dans un contexte d'échange de données, par exemple entre des entreprises d'un même domaine, la base de données est propre à chaque entreprise mais la structure du ou des document(s) XML à produire est commune à toutes. Nous voulions donc permettre la production de document(s) XML qui se conforment à une structure donnée. Pas question donc de produire des documents XML dont la structure serait déduite de celle de la base de données d'une manière prédéfinie.

Pour atteindre cet objectif, il fallait un modèle de représentation des structures XML. Nous avons donc défini un modèle appelé **modèle XML**.

D'autre part, le contexte d'utilisation ne nécessitait pas de s'adapter dynamiquement à un changement structurel tant du côté de la base de données que de celui de la structure du ou des document(s) XML. Nous pouvions donc travailler en construisant une application spécifique à une base de données particulière et à une structure de données fixée, pour autant que l'on propose une **démarche méthodologique générale** permettant d'automatiser, autant que faire se peut, sa construction.

Notre démarche méthodologique exploite les résultats obtenus dans le contexte des transformations de schéma. Elle est constituée principalement de trois étapes :

- la recherche d'une chaîne de transformations permettant de faire évoluer le schéma représentant la structure de la base de données en un schéma représentant la structure du ou des document(s) XML. Cette démarche nécessite l'intervention humaine puisqu'elle exploite la connaissance sémantique que l'analyste a de ces deux structures;
- l'analyse de cette chaîne de transformations afin d'en déduire les règles de conversion de données. Les transformations étant formellement spécifiées, l'étape est entièrement automatisée;
- la génération proprement dite de l'application. Cette étape finale est également automatique.

Enfin, nous voulions étendre quelque peu le contexte d'utilisation premier, c'est pourquoi nous avons proposé des pistes de conception des structures du ou des document(s) XML.

Au niveau de l'implémentation, l'application de production de document(s) XML est de conception simple. Elle est indépendante du SGDB sous-jacent en utilisant un médiateur développé dans le cadre du projet InterDB.

Le suivi de la démarche méthodologique est facilité par le développement d'une méthode permettant de guider l'utilisateur tout au long du processus. Pratiquement, l'analyste n'est sollicité que pour établir le schéma décrivant la structure du ou des document(s) XML par l'application successive d'opérateurs de transformation sur la structure de la base de données. Le reste du processus se fait de manière automatisée.

L'implémentation qui a été faite limite le nombre d'opérateurs de transformation dont on peut faire usage. Cependant, une extension de la palette des transformations disponibles ne semble pas poser de problèmes.

De plus, le modèle XML défini est celui-là même qui a été implémenté. Il ne permet pas la représentation de toutes les structures de document(s) XML existantes. Il semble opportun de mener une étude sur les modifications à faire sur ce modèle afin de supporter un ensemble plus large de DTDs.

Enfin, nombreuses sont les situations qui requièrent l'utilisation d'une application de production de document(s) XML à partir d'une base de données où les données pourraient être sélectionnées non pas uniquement sur base de la structure à laquelle elles correspondent mais également sur base de conditions sur leur valeur. Citons en guise d'illustration, le cas d'une entreprise de commerce électronique qui souhaiterait remettre quotidiennement à jour son site web suivant les produits disponibles en stock. Dans une perspective à plus long terme, il serait intéressant d'étudier les éventualités d'extension de notre travail afin de prendre en charge ce genre de cas.

Bibliographie

[1]. Fernandez M., Tan W., Suciu D., "SilkRoute : Trading between Relations and XML", technical report, AT&T Labs, Novembre 1998.
[2]. van Herwijnen E., "Practical SGML", Second Edition, Kluwer Academic Publishers, 1997.
[3]. Musciano C., Kennedy B., Lukide M., "HTML : The Definitive Guide", O'Reilly&Associates, Septembre 1998, ISBN 1565924924.
[4]. http://www.w3.org/TR/WD-xml
[5]. Martin D. et al., "Professional XML", Wrox Press, 2000.
[6]. Hainaut, J.-L., "Database Reverse Engineering", Research Publication, Namur, 1999, 133 pp.
[7]. Arnold K., Gosling J., "Le langage JAVA", Traduction française, International Thomson Publishing France, Paris, 1996.
[8]. Hainaut J.-L., Thiran P., Hick J.-M., Bodart S., Deflorenne A., "Methodology and case tools for the developpement of federated databases", International Journal of Cooperative Information Systems, 1999.
[9]. Hamilton G., Cattell R., Fisher M., "JDBC Database Access with Java, a Tutorial and Annotated Reference", Addison-Wesley, 1997.
[10]. Hainaut J.-L., "Etude des modèles canoniques et des architectures d'interface", 1996.
[11]. http://www.info.fundp.ac.be/~dbm/
[12]. J.-M. Hick , V. Englebert , J. Henrard, D. Roland, J.-L. Hainaut, "The DB-MAIN Database Engineering CASE Tool (version 5) - Function Overview", DB-MAIN Technical manual, December 1999, public. Institut d'informatique, FUNDP.
[13]. V. Englebert, "Voyager2 (version 5.0) - Reference manual", DB-MAIN technical manual, December 1999, public. Institut d'informatique, FUNDP
[14]. Roland D., Hainaut J.-L., "Database Engineering Process Modeling", Proceedings of The First International Workshop On The Many Facets Of Process Engineering, Gammarth, Tunisia, May 1999.
[15]. Hainaut J.-L., "Entity-generating Schema Transformation for Entity-Relationship Models", Proc. of the 10th Entity Relationship Approach, San Mateo (CA), North Holland, 1991.
[16]. Hainaut, J.-L., "A Generic Entity-Relationship Model", in Proc. of the IFIP WG 8.1 Conf. on Information System Concepts: an in-depth analysis, North-Holland, 1989.

Annexes

A. Le modèle GER

1. Présentation générale

Le modèle GER est un modèle de représentation de structures de données dont l'objectif est de couvrir la plupart des paradigmes de modélisation de données à différents niveaux d'abstraction.
Concrètement, il peut être spécialisé afin de coïncider avec la plupart des modèles basés sur la notion d'objets ou d'entités. On peut citer, entre autre :

- toutes les variantes du modèle Entité/Association;
- les modèles conceptuels binaires (p. ex. NIAM);
- les modèles orienté-objet (p. ex. OMT, UML);
- les modèles de structure de fichiers (p. ex. COBOL).

Le modèle GER se décline sous deux formes :

- une vue abstraite dans laquelle on donne à chaque construction du modèle, une interprétation relationnelle. Elle est utilisée comme base formelle par exemple, lorsqu'il s'agit de définir rigoureusement des transformations sur ces constructions ou d'en démontrer des propriétés;
- une vue concrète dans laquelle on donne à chaque construction du modèle, une représentation graphique. Elle est plutôt utilisée par les utilisateurs du modèle GER. C'est sous cette forme qu'on va décrire le modèle GER.

2. Description des constructions du modèle GER

2.1. Type d'entité

Une **entité** est un objet concret ou abstrait appartenant au réel perçu à propos duquel nous voulons enregistrer des informations. Cependant, dans un processus de description, nous ne nous intéressons pas à chaque objet individuel mais nous envisageons les classes d'objets ou **type d'entité** (TE). Dans un domaine commercial, nous pourrions identifier des objets concrets tels que des clients, des commandes, des produits, des fournisseurs, ... Nous définissons donc naturellement quatre types d'entité : `Client`, `Commande`, `Fournisseur` et `Produit`.

La figure A.1 montre comment les types d'entité sont représentés graphiquement dans le modèle GER.

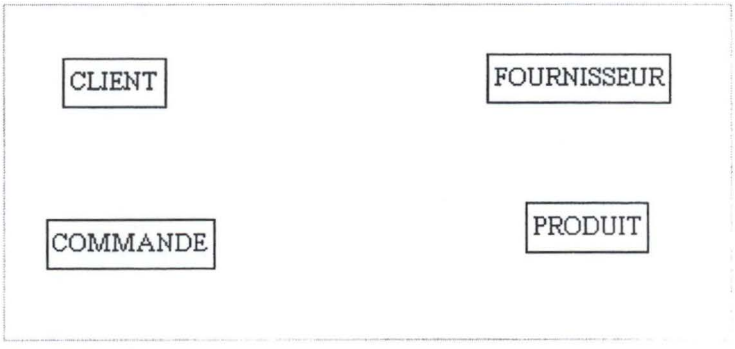


Figure A.1- Représentation graphique de quatre types d'entité.

2.2. Type d'association

Souvent, les différentes entités du domaine d'application sont liées entre elles par des **associations**. Par exemple, les clients *passent* des commandes, les fournisseurs *proposent* des produits, chaque commande *est assignée* à certains fournisseurs pour certains produits.

Les associations vérifiant les mêmes propriétés constituent un **type d'association**. Ainsi, pour notre exemple, nous pouvons définir :

- un type d'association, nommé `passe`, entre les types d'entité `Client` et `Commande`;
- un type d'association, nommé `propose` entre les types d'entité `Fournisseur` et `Produit`;
- un type d'association, nommé `assignée` entre les types d'entité `Commande`, `Fournisseur` et `Produit`.

Un type d'association est représenté graphiquement par un hexagone (cf. Fig. A.2).

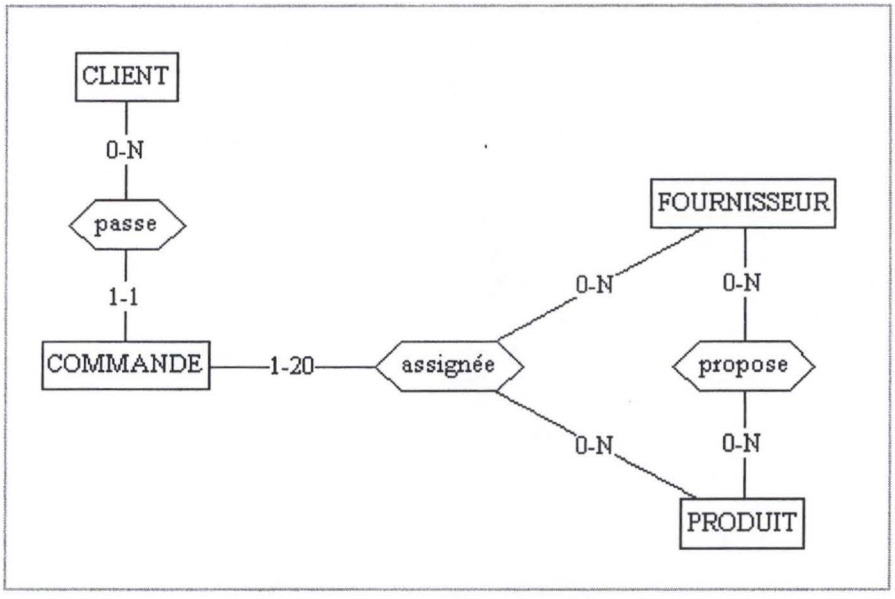


Figure A.2- Représentation graphique de types d'association.

Un type d'association peut relier un certain nombre de types d'entité, non nécessairement distincts. Le nombre de ces types d'entité est appelé le **degré du type d'association**. Dans notre exemple, seul le type d'association `assignée` est de degré 3, les autres sont tous de degré 2. Les types d'association de degré deux sont appelés **type d'association binaires**.

2.3. Attribut

L'**attribut d'un type d'entité** représente une propriété commune à toutes les entités du même type. Par exemple, chaque client est caractérisé par un *numéro*, un *nom*, une *adresse* et par un *téléphone*.

Graphiquement, un attribut est représenté par son nom inscrit dans la partie inférieure du rectangle représentant le type d'entité lui correspondant (cf. Fig. A.3).

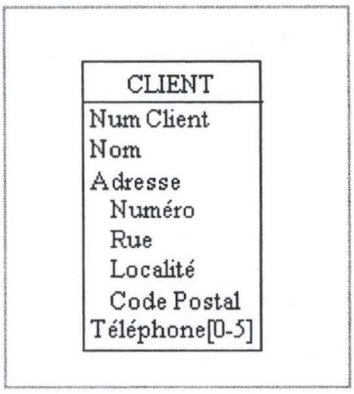


Figure A.3- Représentation graphique des attributs d'un type d'entité.

De manière similaire, un **attribut d'un type d'association** représente une propriété commune à toutes les associations de ce type. Par exemple, chaque association entre un fournisseur et un produit est caractérisée par un *prix* et un *délai de livraison*.

La figure Fig. A.4 illustre la représentation graphique d'attributs d'un type d'association.

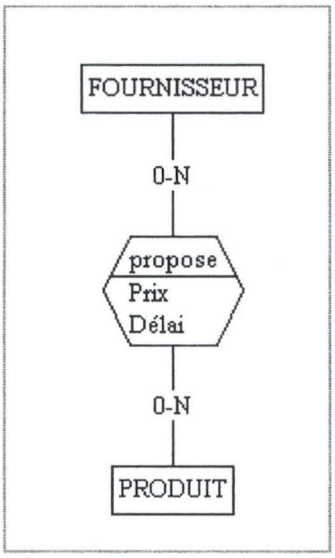


Figure A.4- Représentation graphique des attributs d'un type d'association.

Une **valeur** d'un attribut désigne l'état de la propriété que l'attribut représente pour une entité (association) particulière.

Un attribut est **atomique** si sa valeur ne peut être fragmentée en composants significatifs. En revanche, un attribut est **décomposable** si une valeur de cet attribut est constituée de valeurs plus élémentaires ayant chacune une signification précise dans le domaine d'application. Par exemple, dans la figure A.3, l'attribut `Adresse` du type d'entité `Client` est

décomposable alors que l'attribut `Nom` de ce même type d'entité est atomique.

On désigne par **parent** d'un attribut, le type d'entité, type d'association ou attribut décomposable duquel dépend directement cet attribut. Par exemple, dans la figure A.3, l'attribut `Nom` a comme parent le type d'entité `Client`, alors que l'attribut `Rue` a comme parent l'attribut décomposable `Adresse`.

Un attribut est **multivalué** si pour une occurence de son parent, il peut prendre plusieurs valeurs. Dans le cas contraire, on le qualifie de **monovalué**. Un attribut est caractérisé par un couple de valeurs `[i-j]` où `i` (respectivement `j`) représente le nombre minimum (respectivement maximum) de valeurs que cet attribut peut prendre pour une occurence donnée de son parent. Les nombres `i` et `j` sont appelés respectivement **cardinalité minimale** et **cardinalité maximale**.

Par exemple, dans la figure A.3, l'attribut `Téléphone` du type d'entité `Client` a une cardinalité minimale nulle et une cardinalité maximale de 5. Il est donc multivalué.
Graphiquement, la cardinalité par défaut est [1-1]. Seules les autres cardinalités sont représentées. Pour une cardinalité aussi grande que l'on veut, on utilise la variable `N`.

De plus, on parle d'attribut **facultatif** si la cardinalité minimale est nulle. En d'autre mot, un attribut facultatif n'a pas de valeur pour toutes les occurences de son parent. Par exemple, dans la figure A.3, l'attribut `Téléphone` du type d'entité `Client` est facultatif.
Un attribut non facultatif est appelé **obligatoire**.

2.4. Domaine d'un attribut

Un attribut tire ses valeurs d'un ensemble de références que l'on nomme **domaine de valeurs**. On distingue les **domaines de base**, les **domaines utilisateur** et les **domaines entité**.

Les premiers correspondent aux principaux types de données des SGDB. Par exemple, le domaine des valeurs numériques, des chaînes de caractères, des valeurs booléennes, des dates, ...

Un domaine utilisateur est spécifique au domaine d'application. Contrairement au domaine de base, il est doté d'une signification et d'une définition précise. Il représente un type d'information standard de l'organisation. Par exemple, l'attribut `Téléphone` peut avoir un domaine utilisateur appelé `téléphone` qui rassemble tous les numéros de téléphone des clients de la société. Ce domaine est dynamique, il peut évoluer au fil du temps.

Enfin, un type d'entité peut être utilisé comme domaine pour les attributs. C'est dans ce cas qu'on parle de domaine entité. Les attributs qui ont un domaine entité sont désignés par le terme **attribut objet**.

2.5. Rôle d'un type d'entité

Chaque type d'entité participant à un type d'association joue un **rôle** dans le cadre de ce type d'association. Par exemple, le rôle d'un client est de *passer* une commande, tandis que le rôle d'une commande est d'*être passée*. Graphiquement, le nom du rôle est indiqué sur le segment qui relie le type d'entité au type d'association (cf. Fig. A.5).

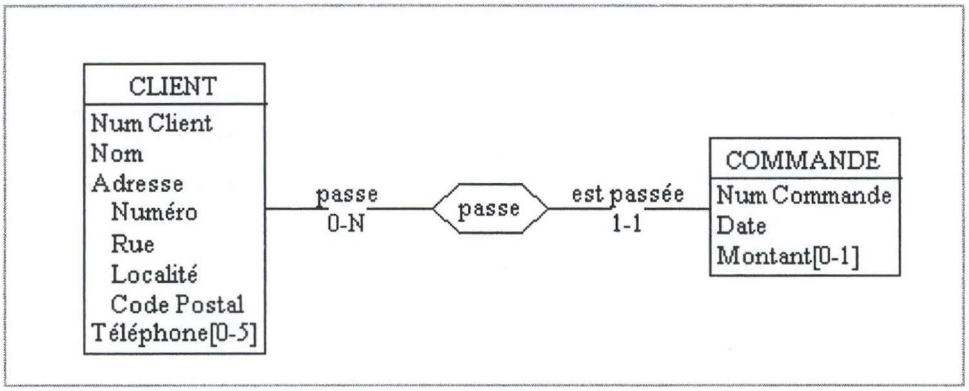


Figure A.5- Représentation graphique du rôle d'un type d'entité quand il participe à un type d'association.

Le rôle qu'un type d'entité joue dans un type d'association est fixé par une **contrainte de cardinalité**. Il s'agit d'un couple d'entiers `[i-j]` indiquant qu'une entité de ce type doit participer à un certain nombre d'association de ce type (compris entre `i` et `j`). Par exemple, la cardinalité du rôle `est passée` de [1-1] indique que chaque commande doit être passée par un et un seul client. La cardinalité d'un rôle est indiquée en dessous du nom de ce rôle.

2.6. Groupe

Un **groupe** est constitué de *composants*, qui sont :

- des attributs;
- des rôles;
- d'autres groupes.

Un groupe est attaché à une construction parente qui peut être un type d'entité, un type d'association ou un attribut décomposable.

Chaque groupe se voit assigné une fonction qui peut être :

- **identifiant primaire** : c'est l'identifiant principal du parent (voir paragraphe 2.7. pour la notion d'identifiant). Pour se voir assigner cette fonction, le groupe ne peut pas avoir de composant facultatif. On utilise le mot clef `id` pour le représenter;
- **identifiant secondaire** : c'est un identifiant du parent. On utilise le symbole `id'` pour le représenter;
- **coexistence** : les composants du groupe doivent avoir simultanément une valeur nulle ou non-nulle pour n'importe quelle occurence du parent. Tous ces composants doivent donc être optionnels. C'est le mot clef `coex` qui est utilisé pour l'identifier;

- **exclusive** : dans ce groupe, il y a au plus un composant qui peut être présent pour une occurence du parent. Tous les composants doivent donc être optionnels. C'est le mot `excl` qui permet sa représentation graphique;
- **at-least-one** : dans ce groupe, au moins un composant doit être présent pour chaque occurence du parent. Tous ses composants doivent être optionnels. C'est le mot `at-lst-1` qui est utilisé au niveau graphique pour indiquer cette fonction;
- **exactly-one** : parmi les composants du groupe, exactement un composant doit être présent pour chaque occurence du parent. Tous ses composants doivent être optionnels. C'est le mot `exact` qui représente cette situation;
- **access key** : les composants du groupe forment un mécanisme d'accès aux instances du parent. Ce concept est une abstraction aux constructions telles que les indexes, les chemins d'accès, ... Il est indiqué par le mot `acc`;
- **contraintes définies par les utilisateurs** : en fonction de ses besoins, l'utilisateur peut définir de nouvelles fonctions qui n'apparaissent pas ici.

2.7. Identifiant d'un type d'entité

Chaque type d'entité peut posséder un ou plusieurs **identifiant**(s) qui permet de repérer univoquement chaque entité de ce type. Par exemple, les entités de type `Client` sont identifiées par leur numéro unique (`Num Client`). Cela signifie qu'à tout instant il n'existe pas deux entités `Client` qui possède la même valeur de `Num Client`.

En toute généralité, un identifiant d'un type d'entité peut être constitué de :

- un ou plusieurs attribut(s);
- un ou plusieurs rôle(s) assumé(s) par le type d'entité;
- un ou plusieurs groupe(s).

Les constituants d'un identifiant d'un type d'entité sont indiqués graphiquement sous le rectangle représentant le type d'entité avec la notation "`id : ...`" (cf. Fig. A.6).

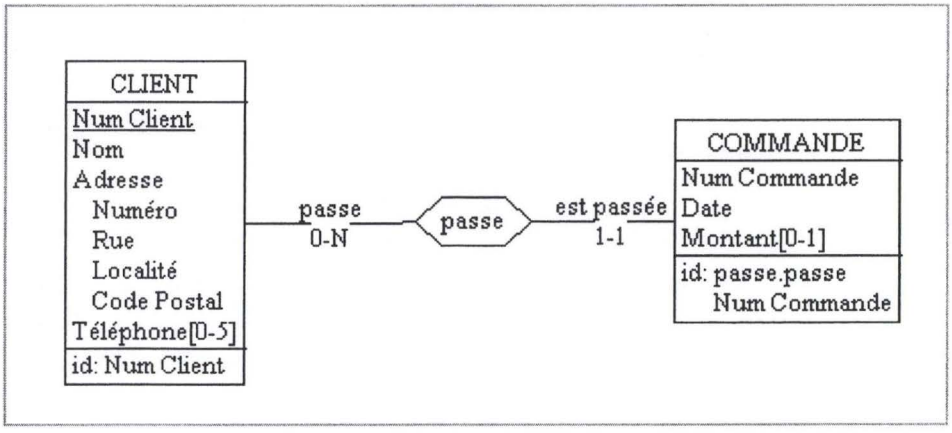


Figure A.6- Représentation graphique d'un identifiant d'un type d'entité.

De manière similaire, on peut définir la notion d'identifiant d'un type d'association.

2.8. Relation IS-A

Supposons que dans notre exemple, il existe deux types de produits, les produits périssables et les non-périssables. Ce sont donc deux catégoriers distinctes de produits. Nous définissons donc deux nouveaux types d'entité : `Denrées Périssables` et `Denrées non Périssables`. Dans ce contexte, on appelle `PRODUIT` **le type générique** (ou le super-type) et, `Denrées Périssables` et `Denrées non Périssables` des **types spécifiques** (ou le sous-type). Ces derniers héritent des propriétés (attributs, identifiants, types d'association) du type générique.
Ainsi, par exemple, si le type d'entité `PRODUIT` était identifié par un attribut `Code Produit`, il en va de même pour les types d'entités `Denrées Périssables` et `Denrées non Périssables`. De même, si le type d'entité `PRODUIT` participe au type d'association `propose`, c'est également vrai pour les types d'entité `Denrées Périssables` et `Denrées non Périssables`.

En plus, les types spécifiques peuvent également posséder des propriétés qui leur sont propres. Par exemple, le type d'entité `Denrées Périssables` a un attribut `Date de Péremption`.

Chaque entité spécifique doit obligatoirement appartenir à une et une seule entité générique. Un type générique est qualifié de **disjoint** si, à tout instant, toute entité de celui-ci est en relation avec au plus un de ses sous-types. Il est qualifié de **total** si, à tout instant, toute entité de celui-ci est en relation avec au moins un de ses sous-types. L'ensemble formé par les sous-types est une **partition** si l'ensemble est total et disjoint.

Une relation super-type/sous-type est appelée une **relation IS-A**.

La relation de sous-type se représente graphiquement par un triangle relié au sur-type par un segment de droite épais et à chaque sous-type par un segment de droite (cf. Fig. A.7). Le triangle peut contenir un caractère : T pour Total, D pour Disjoint et P pour Partition.

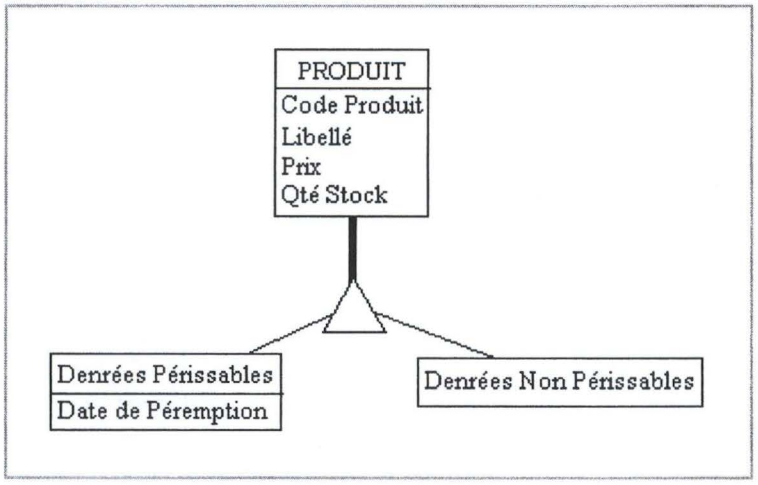


Figure A.7- Représentation graphique d'une relation de sous-type.

2.9.Attributs multivalués

La valeur d'un attribut multi-valué est en réalité une **collection** de plusieurs valeurs de même type. Il est possible de préciser cette notion de collection.

On distingue 6 types de collection distincts :

- **ensemble** : collection non-ordonnée de valeurs distinctes. Graphiquement, c'est le type de collection par défaut;
- **amas** : collection non ordonnée de valeurs non nécessairement distinctes. Graphiquement, il est représenté par le mot clef `bag`;
- **liste unique** : collection ordonnée de valeurs distinctes. Graphiquement, il est représenté par le mot clef `u-list`;
- **liste** : collection ordonnée de valeurs non nécessairement distinctes. Graphiquement, il est représenté par le mot clef `list`;
- **tableau** : collection indexée de cellules pouvant contenir des valeurs non nécessairement distinctes. Graphiquement, il est représenté par le mot clef `array`;
- **tableau unique** : collection indexée de cellules pouvant contenir des valeurs distinctes. Graphiquement, il est représenté par le mot clef `u-array`.

2.10. Contraintes inter-groupe

Indépendamment de la fonction qui leur est assignée, les groupes qui ont des composants compatibles (au niveau de leur structure et de leur domaine) peuvent être reliés entre eux à travers une relation qui exprime une contrainte d'intégrité inter-groupe.

On dénombre 3 contraintes d'intégrité inter-groupe :

- **référence** : le premier groupe ne peut prendre ses valeurs que parmi celles prises par le second groupe. Par exemple, sur la figure A.8, l'attribut `Client` du type d'entité `COMMANDE` doit avoir une valeur appartenant à l'ensemble des valeurs prises par l'attribut `Num Client` dans le type d'entité `CLIENT`. Le groupe qui est référencé doit être un identifiant. Le groupe de référence est symbolisé par le mot `ref` et une flèche montre quel est le groupe référencé;

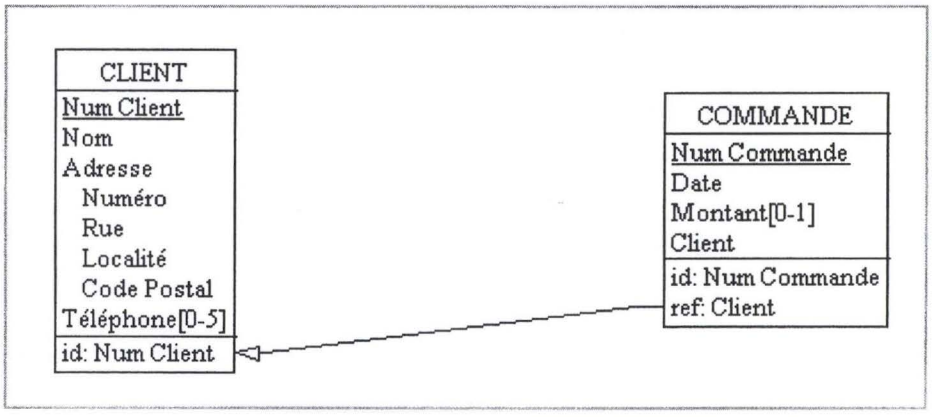


Figure A.8- Représentation graphique d'un groupe de référence.

- **inclusion** : chaque valeur prise par le premier groupe doit appartenir à l'ensemble des valeurs prises par le second groupe. La seule différence avec le cas précédent est que le premier groupe n'est pas nécessairement un identifiant. C'est le terme `incl` qui permet de représenter cette contrainte;
- **référence égale** : chaque valeur prise par le premier groupe doit appartenir à l'ensemble des valeurs prises par le second groupe. Ce second groupe est un identifiant. De plus, une contrainte d'inclusion est définie du second groupe vers le premier. Ces deux contraintes forment une contrainte de référence égale. Elle est symbolisée par le terme `equ`.

2.11.Collection

Une collection est un repository concret ou abstrait d'entités. Une collection peut contenir des entités de types différents et les entités d'un type donné peuvent être stockées dans plusieurs collections. Ce concept sert par exemple, à représenter des fichiers.

La figure A.9 montre comment représenter une collection graphiquement.

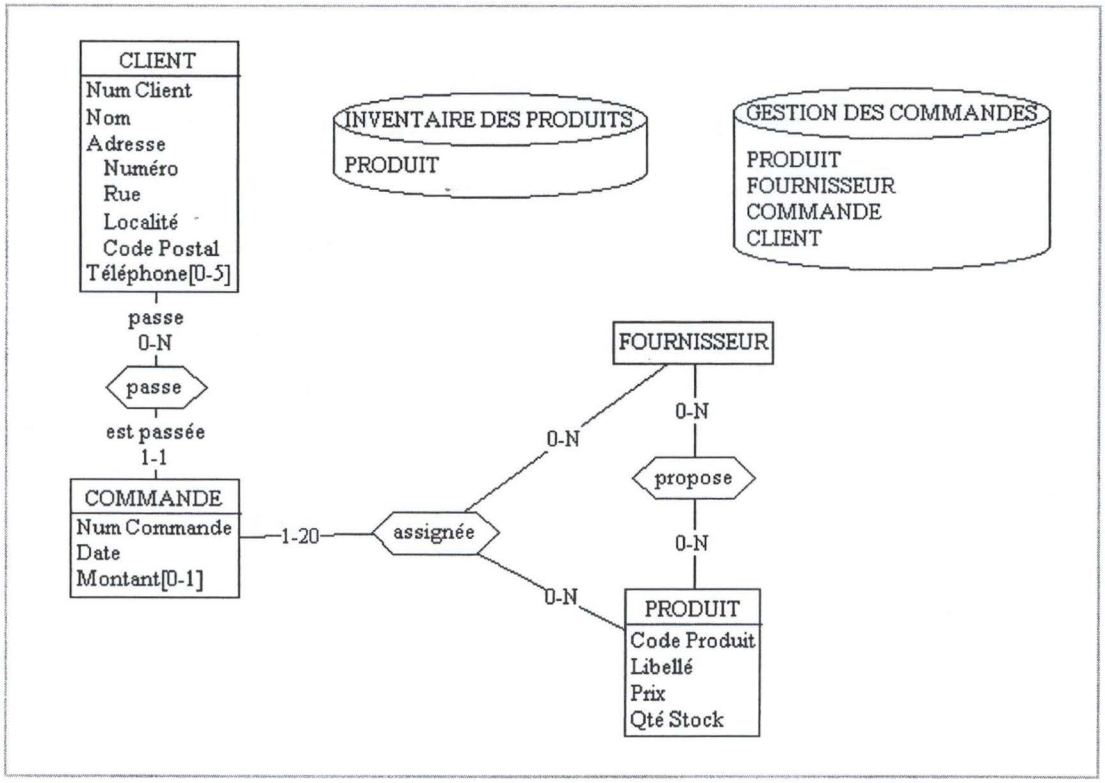


Figure A.9- Représentation graphique de collections.

Nous voici au terme de cette présentation sommaire du modèle GER. Pour plus de renseignements, consultez [16].

B. Description de l'interface des classes du migrateur

Cette section présente l'interface des différentes classes constitutives du migrateur (cf. Fig. 4.7). Avant cela, nous commençons par décrire les différentes classes qui nous sont offertes par le driver InterDB. Elles ne font pas partie en tant que telles du migrateur mais elles y sont largement utilisées.

	Driver InterDB
Classes	**Méthodes**
DriverManager	*Connection* GetConnection(*String* url, *String* login, *String* password) Cette méthode permet d'établir une connection avec le serveur logique.
Connection	*Statement* createStatement() permet de créer une requête SQL statique. *void* close() libère la connection avec le serveur logique.
Statement	*ResultSet* executeQuery(*String* lql) initialise et exécute la requête LQL dont le texte figure en paramètre d'entrée.
ResultSet	*boolean* next() renvoie *True* si le *ResultSet* contient encore des résultats, *False* sinon. *Object* getObject(*String* AttributeName) renvoie un objet contenant la valeur courante de l'attribut dont le nom figure en paramètre d'entrée. *void* close() ferme le *ResultSet*.
Instance	Une classe Instance est créée par type d'entité de la base de données. Les objets de cette classe permettent de contenir une instance du type d'entité correspondant. La structure de la classe est donc calquée sur celle de ce type d'entité comme le montre l'exemple ci-dessous. *public class CLIENT implements Serializable {* *public Integer Numero;* *public String Nom;* *public String Prenom;* *public oAdresse Adresse;* *public Vector Telephone;* *public Vector N_Comptes;}* La classe *Vector* est contenue dans le package java.util de Java. La classe *oAdresse* est décrite ci-dessous.
Attribut composé	A chaque attribut composé des structures de la base de données correspond une classe dont les objets permettent la manipulation des valeurs de cet attribut. *public class oAdresse implements Serializable {* *public String Rue;* *public Integer Numero;* *public Integer Code_Postal;* *public String Localite;* *public String Pays;}*

	Class XMLFile
Héritage	*java.io.FileWriter*
Description	Un objet XMLFile est un document XML auquel est attaché un DTD. NON GENEREE.
Propriété	*String* File_Name : nom du document XML (chemin complet du fichier); *String* DTD_Nom : nom du DTD qui lui est associé (chemin complet du fichier); *String* First_Nom : nom du type d'élément racine du document XML.
Constructeur	XMLFile (*String* File_Name, *String* DTD_Nom, *String* First_Nom) throws *java.io.IOException*
Méthodes	*Void* WriteHeader() écrit l'en-tête du document XML, y compris la référence au DTD et la balise ouvrante relative à l'élément racine. *Void* WriteTailer() écrit la balise fermante relative à l'élément racine à la fin du document XML. *Void* Close() throws *java.io.IOException* ferme le document XML.

	Class Specific*Nom_De_TE_BD*
Description	Un objet Specific*** représente l'opération d'extraction, de conversion et de formatage d'un ensemble d'entités. La requête d'extraction porte sur la totalité des instances de ce type ou sur une partie (déterminée par le résultat d'une autre requête : instsup). La méthode *Write* peut déclencher le traitement d'un autre groupe d'instances. GENEREE.
Propriété	*XMLFile* fw : nom du document XML dans lequel le stockage va être effectué;
Constructeur	Specific*** (*XMLFile* fw, *Connection* con)
Méthode	*Void* WriteXML(*Object* inst) L'instance d'entrée issue du type d'entité correspondant à cette classe va être transformée et stockée selon le format adéquat dans le *XMLFile* associé à cette classe. *Void* Write(Object instsup) exécute la requête d'extraction portant sur la table correspondante au TE considéré; traite tour à tour chaque résultat de la requête (conversion et formatage) et déclenche des requêtes paramétrisées par ce résultat si nécessaire.

	Class Collec*Nom_De_Collection*
Description	Cette classe prend en charge le traitement complet d'un document XML. GENEREE
Propriété	*DriverInterDB.Connection* con : un objet représentant la connection à la base de données; *XMLFile* fw : un objet représentant le document XML que l'on veut créer.
Constructeur	Collec***(*String* file, *String* DTD, *String* FirstElem, *Connection* con)
Méthodes	*Void* CollTreatment() prend en charge la production complète du document XML modélisé par la collection considérée ici. Cela se fait en utilisant les méthodes *Write* des classes de type *SpecificNom_De_TE_BD*.

	Class Migrateur
Héritage	*java.lang.Object*
Description	Cette classe gère la connection à la base de données et le traitement complet aboutissant à l'élaboration de différents documents XML à partir des données de la base de données. GENEREE
Méthode	Main() se connecte à la base de données; génère tour à tour chaque document XML à partir de la méthode offerte par la classe Coll***; libère la connection à la base de données.