Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche

researchportal.unamur.be

UNIVERSITÉ DE NAMUR

University of Namur

THESIS / THÈSE

MASTER EN SCIENCES INFORMATIQUES

Etude et expérimentations d'outils de publication, de transformation et de validation de document XML

Robbe, Maxime; Walgraffe, Sébastien

Award date:
2002

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 23. Jun. 2020

Facultés Universitaires Notre-Dame de la Paix, Namur

Institut d'Informatique

Etude et expérimentations d'outils de publication, de transformation et de validation de document XML
-
Study and experimentation of publication, transformation and validation tools for XML documents

-

Maxime Robbe & Sébastien Walgraffe

Année Académique 2001 – 2002

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

Résumé

Le langage XML est de plus en plus utilisé dans la conception des applications actuelles. Nous présenterons les concepts de base du langage XML ainsi que d'autres langages et outils qui interagissent avec le langage XML.
Parmi ceux-ci, nous examinerons le langage XSL et indiquerons en quoi il est intéressant dans un rôle de transformation et de publication.
Dans le cadre de la transformation de fichiers XML, XSL a attiré notre attention car il permet de générer du code qui n'est pas XML.
Dans le cadre de la publication, nous étudierons Cocoon, le framework de publication de Apache.
Nous examinerons différents outils et langages de validation (DTD, W3C XML Schema, Schematron et Xlinkit) que nous comparerons.
Nous développerons finalement plusieurs études de cas utilisant certaines des technologies développées précédemment.

Abstract

XML language is more and more used in the design of new applications.
In this document, we present basic concepts of XML language, as well as other associated XML-interactive languages and tools.
Among others, we examine XSL language, and discuss its interest in a publication and transformation goal.
XSL is of a particular interest because it allows code generation towards a language that isn't XML.
In our study of publication problem, we study Cocoon, the publication framework of the Apache XML project.
We also examine some validation tools and languages (DTD, W3C XML Schema, Schematron and Xlinkit) and we compare them.
Finally, we develop a number of case studies illustrating the practical use of the technologies introduced above.

Remerciements

A Eric Dubois;
pour sa gentillesse, ses conseils et sa patience

A Pierre Brimont, Olivier Marchand et Thibaud Latour,
pour leur aide et leur sympathie;

A Catherine Bossicart et Anne Gaspard,
pour leur bonne humeur et leur gentillesse;

A tous les membres du centre de recherche public Henri Tudor (CRP-HT),
pour leur accueil chaleureux;

A nos professeurs,
pour le savoir qu'ils nous ont transmis;

A tous nos camarades de classe,
pour ces cinq années magnifiques passées à leurs côtés;

Aux courageux relecteurs,
Pour toutes les fautes d'orthographe qu'ils ont trouvé;

Enfin tout un chacun,
qui se reconnaîtra, pour le soutien, la bonne humeur apportés durant ce travail.

Table des matières

TABLE DES MATIERES 4

TABLE DES FIGURES 7

GLOSSAIRE......... 9

1. INTRODUCTION 12

2. CONCEPTS DE BASE 14

2.1. OBJECTIF DU CHAPITRE. 14
2.2. CARACTERISTIQUES DE BASE D'UN DOCUMENT XML. 14
2.3. A QUOI RESSEMBLE UN DOCUMENT XML ? 14
2.4. LA SYNTAXE XML......... 16
2.5. BALISES ET ATTRIBUTS......... 19
2.6. GRAMMAIRE ET VALIDATION DE DOCUMENTS XML 21
2.7. LES NAMESPACES XML 23

3. TRANSFORMATIONS DE DOCUMENT XML : LA TECHNOLOGIE XSL 25

3.1. OBJECTIF DU CHAPITRE 25
3.2. INTRODUCTION 25
3.3. LA TECHNOLOGIE XSL......... 25
3.4. A QUOI RESSEMBLE UNE FEUILLE DE STYLE XSL ? 27
3.5. LE LANGAGE DE SELECTION XPATH......... 29
3.5.1. Les location path......... 29
3.5.2. Les prédicats 31
3.5.3. Sélection d'attributs avec XPath......... 33
3.5.4. Les « context-nodes » et les modes de recherche avec XPath......... 33
3.6. LE LANGAGE DE TRANSFORMATION XSLT......... 34
3.6.1. Présentation générale......... 34
3.6.2. Fonctionnement de XSLT......... 35
3.6.3. Mécanismes avancés 36
3.6.4. Références......... 40

4. PUBLICATION DE DOCUMENTS XML......... 41

4.1. OBJECTIF DU CHAPITRE 41
4.2. INTRODUCTION 41
4.3. PUBLICATION DE DOCUMENT XML PAR DEFAUT. 42
4.4. XSLFO (XSL FORMATING OBJECTS)......... 43
4.5. XSL ET HTML 45
4.6. LE « FRAMEWORK » DE PUBLICATION COCOON......... 46
4.7. LE DOCUMENT SITEMAP COCOON 47
4.7.1. Généralités......... 47
4.7.2. Utilisation des « wildcards » et des « tokens »......... 49
4.7.3. Paramètres du transformateur......... 50
4.7.4. Mécanismes avancés de sélection 51
4.7.5. Conclusion 52

5. VALIDATION DE DOCUMENTS XML 53
5.1. OBJECTIF DU CHAPITRE 53
5.2. INTRODUCTION. 53
5.3. LES DTDS 54
5.3.1. Qu'est-ce qu'une DTD ? 54
5.3.2. Les règles de construction d'une DTD 54
5.3.3. Limites des DTDs 59
5.4. LES XML SCHEMAS. 60
5.4.1. Qu'est-ce qu'un XML Schema? 60
5.4.2. Les règles de construction d'un XML Schema 60
5.4.3. Limites des XML Schemas. 67
5.5. SCHEMATRON 68
5.5.1. Qu'est-ce que Schematron 68
5.5.2. Fonctionnement 68
5.5.3. Construction du XML Schema de règles 69
5.5.4. Choix de la meta-stylesheet 71
5.5.5. Limites de Schematron 71
5.6. XLINKIT 72
5.6.1. Qu'est-ce que Xlinkit? 72
5.6.2. Fonctionnement de Xlinkit 72
5.7. TABLEAU RECAPITULATIF. 76
5.8. PERSPECTIVES D'AVENIR 77
6. ETUDE DE CAS: INTEGRATION DE LA VALIDATION DANS LE MARCHE DES STAGES. 78
6.1. PRESENTATION DE L'ETUDE 78
6.2. ENJEUX 79
6.3. ARCHITECTURE UTILISEE 79
6.3.1. Le choix des technologies 79
6.3.2. Architecture générale 80
6.4. LA RECUPERATION DES CHAMPS VIA JSP 80
6.5. LA VALIDATION PAR XLINKIT 81
6.6. TRAITEMENT DES RESULTATS ET AFFICHAGE 85
6.7. EVALUATION ET AMELIORATIONS 88
7. ETUDE DE CAS: GENERATION DE FORMULAIRE 89
7.1. PRESENTATION DE L'ETUDE 89
7.2. ENJEUX 89
7.3. ARCHITECTURE 90
7.3.1. Le choix des technologies 90
7.3.2. Architecture générale 90
7.4. OPTIONS DE PUBLICATION : LE NAMESPACE GENFORM 91
7.5. IDENTIFICATION DES VARIABLES 93
7.6. STRUCTURES SUPPORTEES 94
7.7. AMELIORATIONS POSSIBLES 94
8. ETUDE DE CAS : LA RECHERCHE ENSEMBLISTE. 95
8.1. PRESENTATION DE L'ETUDE 95
8.2. ENJEUX 96
8.3. ARCHITECTURE 96

8.3.1. Le choix des technologies *96*
8.3.2. Architecture générale *97*
8.3.3. Le fichier XML de données. *97*
8.3.4. Le système de règles *98*
8.4. LA SAISIE DE PARAMETRES 99
8.5. LE TRAITEMENT DES PARAMETRES 100
8.6. L'AFFICHAGE DES PARAMETRES 103
8.7. EVALUATION ET AMELIORATIONS POSSIBLES 105

9. CONCLUSION 106

BIBLIOGRAPHIE 107

ANNEXES 110

A.1. LES MODES DE RECHERCHE AVEC XPATH 110
A.2. EXEMPLE COMPLET D'UTILISATION DE XLINKIT 112
A.3. LES REGLES COMPLETES DE LA VALIDATION DU MARCHE DES STAGES 117

Table des figures

Figure 2.1. Exemple de document XML 15
Figure 2.2. Stricte imbrication des balises XML..... 18
Figure 2.3. Représentation arborescente d'un document XML 19
Figure 2.4. Modélisations alternatives pour les entrées de l'annuaire..... 21
Figure 2.5. Exemple d'utilisation de namespace..... 24
Figure 3.1. Fonctionnement de XSL 26
Figure 3.2. Exemple de feuille de style XSL 27
Figure 3.3. Résultat de l'application de la feuille XSL de la figure (3.2) sur le document XML de la figure (2.1)..... 28
Figure 3.4. Location Path, premier exemple..... 30
Figure 3.5. Location Path, deuxième exemple..... 30
Figure 3.6. Location Path, utilisation de "*" 31
Figure 3.7. Location path avec prédicats. 32
Figure 3.8. Location Path, utilisation de "@" 33
Figure 3.9. Exemple d'utilisation du mode :..... 37
Figure 3.10. Utilisation des variables et paramètres..... 39
Figure 4.1. Output d'un document XML sous Internet Explorer..... 42
Figure 4.2. Vue arborescente de document XML avec XMLSpy..... 43
Figure 4.3. Fonctionnement de XSLFO..... 44
Figure 4.4. Exemple de document XSLFO et la sortie correspondante 44
Figure 4.5. Schéma de fonctionnement de Cocoon..... 47
Figure 4.6. Entrées de sitemap avec document XML commun 49
Figure 4.7. Paramètres dans une entrée sitemap 50
Figure 4.8. Utilisation de <map:choose>..... 51
Figure 5.1. Fonctionnement de Schematron 68
Figure 5.2. Fonctionnement de Xlinkit..... 72
Figure 5.3. Exemple général de DocumentSet.xml..... 73
Figure 5.4. Exemple général de RuleSet.xml..... 73
Figure 5.5. Exemple général de rule.xml..... 74
Figure 5.6. Exemple général de fichier résultat 75
Figure 6.1. Exemple de formulaire pour une entreprise..... 78
Figure 6.2. L'architecture utilisée..... 80
Figure 6.3. Le fichier DocumentSet.xml utilisé..... 81
Figure 6.4. Le fichier RuleSet.xml utilisé..... 82
Figure 6.5. Exemple de règles..... 83
Figure 6.6. Exemple de résultat..... 85
Figure 6.7. Template XSL fournissant un message d'erreur..... 86
Figure 6.8. Template faisant correspondre un message d'erreur à la règle 1 87
Figure 6.9. Message d'erreur..... 87
Figure 7.1. Fonctionnement du formulaire 89
Figure 7.2. Architecture employée :..... 91
Figure 7.3. Exemple de formulaire généré automatiquement..... 93
Figure 8.1. Exemple de moteur de recherche..... 95
Figure 8.2. Architecture de notre moteur de recherche 97
Figure 8.3. Décomposition d'une règle..... 99
Figure 8.4. Formulaire de saisie..... 99
Figure 8.5. Exemple de traitement pour la sous-règle i ayant n contraintes..... 100
Figure 8.6. Feuille de style de "filtrage" 101

Figure 8.7. La feuille de style "doublons" 102
Figure 8.8. Exemple de choix de présentation. 103
Figure 8.9. Détail de la feuille de style "afficher" 104
Figure 8.10. Exemple d'affichage de résultats 105
Figure A.1. Les modes de recherche, exemple 111

Glossaire

- *API* : "Application Program Interface"

- *CGI* : "Common Gateway Interface"
  Standard de communication utilisé pour permettre à des programmes extérieurs de communiquer avec des serveurs http ([CGI99]).

- *Cocoon* :
  Framework de publication de documents XML de chez Apache XML project (ApCo02).

- *CRP-HT* : "Centre de Recherche Public Henri Tudor"
  Notre lieu de stage, Luxembourg Kirchberg ([CRP02]).

- *CSS* : "Cascading Style Sheet"
  Système de feuille de style utilisé par HTML ([CSS02]).

- *DTD* : "Document Type Definition"
  Formalisme de grammaire de document balisé ([Dtd00]).

- *FOP* : "Formatting Object Procesor"
  Convertisseur de format utilise par XSL ([ApFo02]).
  Site officiel

- *HTML* : "HyperText Markup Language"
  Langage de balises utilisé pour la publication sur le World Wide Web ([Html02]).

- *IDE* : "Interactive Development Environment".

- *Java* :
  Langage de programmation développé par Sun Microsystems ([Jav02]).

- *Javascript* :
  Langage de script côté client pour le World Wide Web.

- *JFOR* : "Open-Source Java XSL-FO to RTF converter"
  Convertisseur de format utilisé par XSL ([Jfo02]).

- *JSP* :" Java Server Page"
  Langage de script côté serveur développé par Sun Microsystems ([Jsp02])..

- *PDF* : Portable Document Format.

- *PHP* : "Hypertext PreProcessor"
  Langage de scripts ([Php02]).

- *Relax NG* :
  Langage de Schemas alternatif aux W3C XML Schemas ([Rel02]).

- *RTF* : "Rich Text Format".

- *Schematron* :
  Application de validation de documents XML ([Sch02]).

- *SGML* : "Standard Generalized Markup Language"
  Langage de balises ancêtre de XML ([Sgm00]).

- *SQL* : "Structured Query Language"

- *TOMCAT* :
  Serveur de servlets Java développé par Apache Software Fondation ([ApTo02]).

- *TREX* : "Tree Regular Expressions for XML"
  Langage de validation pour XML ([Tre02]).

- *URL* : "Universal Ressource Locator" ([Ur02]).

- *URI* : "Universal ressource Identifier" ([Ur02]).

- *W3C* : "World Wide Web consortium"
  Organisme officiel validant les technologies du World Wide Web ([W3c02]).

- *WAP* : "Wireless Application Protocol"

- *WWW* : "World Wide Web"

- *Xalan* :
  Processeur XSLT de chez Apache Software Fondation ([ApXa02]).

- *XHTML* : "eXtensible HyperText Markup Language" ([Xht02]).

- *Xlinkit* :
  Application de validation de documents XML ([Xli02]).

- *XML* : "eXtensible Markup Language"
  Langage de balises universel, objet de ce document ([Xml02]).

- *XPath* : "XML Path Language"
  Langage de requêtes utilisé par XSL ([Xpa99]).

- XSL : "eXtensible Stylesheet Language"
  Technologie de feuilles de style utilisée pour transformer des structures XML en d'autres structures ([Xsl02]).

- *XSLFO* : "XSL Formatting Object"
  Formalisme de mise en forme compris dans XSL ([Xsl02]).

- *XSLT* : "XSL Transformation"
  Langage faisant partie de XSL ([Xslt99]).

- *XVIF* : "XML Validation Interoperability Framework" ([Pxv02]).

1. Introduction

L'utilisation de l'Internet et des réseaux est en pleine expansion, il est désormais assez rare de trouver une grande entreprise ne disposant pas de tels moyens de communication.

L'essor de tels réseaux a eu un impact sur les frontières de l'entreprise. Alors qu'avant, elles vivaient dans un environnement fermé où les informations ne sortaient pas des murs de l'entreprise, elles rentrent maintenant dans un monde ouvert, où l'information peut être échangée avec des filiales délocalisées, voire d'autres entreprises (partenaires).

Cet essor a eu pour conséquence une remise en question du fonctionnement des systèmes d'information existants et une adaptation de ceux-ci, principalement au niveau de l'échange de données sur les réseaux.

Cette ***interopérabilité*** des systèmes d'information nécessite de plus en plus l'adoption de normes standard quant au format d'échange des informations. En effet, il est impératif que les applications qui sont appelées à communiquer entre elles puissent se comprendre mutuellement, récupérer les données qui leur sont envoyées mais également comprendre la structure et la sémantique de celles-ci.

Les échanges de données sont également un enjeu économique important, avec l'avènement d'un nouveau type de commerce, appelé "***E-Business***"

C'est pour combler ce besoin réel, et dans un souci de faciliter cet échange de données, que le langage XML a été imaginé.

Le langage XML est un langage simple et générique. Il permet de modéliser un grand nombre de structures de données, de type de problèmes, ceci grâce à la possibilité d'associer aux données une sémantique appropriée qui aide à les décrire et les identifier.

La technologie XML s'est développée de telle manière qu'à l'heure actuelle, c'est plus qu'un langage, c'est une famille de langages. La technologie XML regroupe un vaste ensemble de langages et d'outils qui interviennent de près ou de loin avec le langage XML.

Le but du présent mémoire n'est pas de répertorier ces différents langages et outils, mais plutôt de mettre l'accent sur ceux qui paraissent les plus prometteurs et démontrer leurs potentialités au travers du développement de plusieurs études de cas.

Le choix étant, comme on l'a dit, très vaste, il nous semblait plus pertinent d'examiner la fonction qu'ils remplissaient de manière à mieux cerner leur interaction avec le langage XML.

Ainsi, notre choix s'est porté sur le langage XSL, qui permet de transformer un document XML en un autre document. Plus précisément, nous avons examiné la possibilité de transformer un document XML en un document HTML, ce qui permet aisément d'afficher des données, mais aussi de transformer une structure XML en une autre, ce qui permet des manipulations intéressantes.

L'interopérabilité étant un des enjeux clés de XML, il nous a semblé intéressant d'approcher la manière avec laquelle l'unicité des formats de données pouvait être vérifiée. Nous avons donc abordé plusieurs validateurs, dont les spécificités sont plus ou moins différentes. La validation a un rôle capital dans l'interopérabilité, puisqu'elle permet de dire si un document XML ou son contenu, correspond bien à ce qui avait été défini par les partenaires. La validation sera traitée en détail au chapitre 5.

Nous nous sommes également intéressés aux frameworks de publication de documents XML, dont les potentialités sont grandes. En particulier, nous examinerons le logiciel cocoon de chez Apache Software Fondation au chapitre 4.

Enfin, nous présenterons une série d'études de cas, en rapport avec les technologies développées au niveau théorique :

- Le marché des stages (chapitre 6) : étude de cas dans laquelle nous illustrerons la problématique de la validation de document XML.

- La génération de formulaire (chapitre 7) : étude de cas dans laquelle nous soulignerons les mécanismes de transformation que propose la technologie XSL.

- La recherche ensembliste (chapitre 8) : étude de cas plus audacieuse où nous mettrons en évidence également les mécanismes de transformation XSL.

2. Concepts de base

2.1. Objectif du chapitre.

Ce chapitre présente les concepts de base du langage XML. A savoir, le système de balisage, la notion de document bien formé, les attributs de balises et les namespaces XML ([Mic99] et [NoHe99]).

Il introduit aussi brièvement à la validation de documents XML.

2.2. Caractéristiques de base d'un document XML.

XML est un langage de description de données « universel » (défini en [Xml02]).
XML est issu du langage SGML et utilise tout comme lui un système de balisage.
Ce balisage permet de structurer les données contenues dans un document.

XML est dit universel car il permet de décrire n'importe quel type de structure et laisse chaque auteur de document, libre de personnaliser la structure qu'il utilise pour décrire ses données. Il n'y a pas de définition de balises par défaut, chaque auteur est libre d'utiliser le nombre de balises qu'il veut, et d'y associer la sémantique de son choix. La structuration est complètement libre.

Bien sûr, certaines règles syntaxiques simples doivent être respectées, nous y reviendrons au point 2.4.

2.3. A quoi ressemble un document XML ?

Pour illustrer ce à quoi un document XML peut ressembler, nous avons choisi un exemple simple : un annuaire téléphonique dont voici une description :

- L'annuaire téléphonique est composé d'un certain nombre d'entrées.

- Chaque entrée est soit à caractère privé, soit à caractère professionnel.

- Chaque entrée de l'annuaire décrit un contact ; chaque contact a un nom et un numéro de téléphone.

Cet exemple tout simple nous permet néanmoins de dégager quelques concepts syntaxiques intéressants concernant XML.

La figure (2.1) est une des structures XML possibles pour représenter notre annuaire. L'indentation des différents niveaux et les retours à la ligne ne sont bien sûr pas obligatoires et sont utilisés uniquement dans un souci de lisibilité de l'exemple.

Figure 2.1. Exemple de document XML

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<! DOCTYPE livre SYSTEM
"http://www.tudor.lu/carnetTelephone.dtd">

<!— Modélisation de notre annuaire téléphonique -- >

<annuaire>
      <entrée type="professionnel">
      <nom>John Doe</nom>
      <téléphone>03 44 57 89</téléphone>
      </entrée>

      <entrée type="privé">
      <nom>Aline Dubois</nom>
      <téléphone>63 69 00 98</téléphone>
      </entrée>

</annuaire>
```

Faisons quelques remarques au sujet de ce document :

- Il ne s'agit que d'un choix de modélisation parmi d'autres. Il n'existe pas en XML de solution « type » pour modéliser un carnet de téléphone. La structuration du document est laissée à l'appréciation de l'auteur du document.

- Ce document est parfaitement lisible par un être humain. La sémantique de chaque élément de la structure saute aux yeux et ceci grâce à l'emploi de noms de balise explicites. L'auteur d'un document XML a en effet la liberté quant au choix du nom des balises qu'il emploie dans sa modélisation. Cette liberté ajoutée à la liberté au niveau de la structuration du document lui-même font de XML un langage de description de données universel et générique.

- L'en-tête du document permet d'annoncer qu'il s'agit bien d'un document XML. Cette en-tête précise aussi mais facultativement le jeu de caractères utilisé pour l'encodage du document.

- La balise `<! DOCTYPE ..>` fait référence à une grammaire DTD (voir le chapitre 5 : validation de documents XML).

- En XML, les commentaires sont écris entre les balises "`<!--`" et "`-- >`"

- Le type de balisage utilisé en XML n'est pas sans rappeler le langage HTML (défini en [Html02]). Ceci s'explique par le fait que XML et HTML sont tous les deux descendants de SGML qui lui-même utilise un système de balisage similaire. La principale différence entre les documents XML et les documents HTML est que HTML utilise le balisage pour décrire la

manière dont il faut afficher les données tandis que XML lui, utilise le balisage pour décrire la sémantique des données et dégager une structure de données. Alors qu'un document HTML est donc un document qui décrit la mise en page relative aux données qui y sont contenues, un document XML est une structure de données organisées.

2.4. La syntaxe XML

XML utilise un système de balisage pour structurer ses documents. Nous appellerons ***structure d'un document XML***, l'agencement des balises d'un document XML.

Une balise XML est comprise entre "<" et ">".

Chaque balise a un nom qui l'identifie. Comme déjà mentionné plus haut, l'auteur d'un document XML a toute liberté quant au choix du nom de ses balises. Certaines remarques sont quand même à faire :

- Les majuscules sont à prendre en compte. La balise `<annuaire>` est différente de `<ANNUAIRE>` ou de `<AnnUaiRe>`.

- Un nom de balise ne peut comprendre que des lettres, des chiffres et certains caractères spéciaux. Des noms de balise comme `<hello#|@#{>` ne sont dès lors pas valides.

- Un nom de balise commence toujours par une lettre ou les caractères "_" et ":". `<444719>` n'est dès lors pas un nom de balise valide.

Si aucune autre contrainte n'existe sur les noms de balise, on ne saurait que trop conseiller aux auteurs d'utiliser des noms de balises *explicites* dans leur modélisation. La liberté de choix pour les noms est en effet une des grandes forces de XML, à condition d'être bien exploitée à des fins de lisibilité du document. Un mauvais choix de nom de balise peut se révéler désastreux quant à l'intérêt d'utilisation pratique d'un document XML, aussi désastreux que n'importe quelle erreur de modélisation. De même, il est fortement recommandé d'adopter **des *conventions de notations*** dans le choix de noms pour les balises. Nous nous permettons de proposer quelques pistes :

- Les majuscules étant prises en compte, utiliser soit des minuscules partout, soit des majuscules partout pour les noms de balise (ne pas utiliser des noms de balises en majuscules à un endroit du document et utiliser des noms en minuscules à un autre endroit). Mélanger les deux nuit gravement à la clarté. Nous recommandons les minuscules, en prévision des autres pistes que nous citons plus bas.

- Utiliser des substantifs précis et des abréviations évidentes pour raccourcir le plus possible les noms de balises sans nuire à leur lisibilité. Préférer par exemple `<numTéléphone>` à `<Numéro_de_téléphone>`. Ceci évite une certaine lourdeur du document.

- Utiliser les majuscules pour indiquer les débuts de mots dans un même nom de balise, comme par exemple dans `<numTéléphone>` . Cette convention nous semble plus claire qu'une convention utilisant le caractère « _ » comme séparateur de mots.

- Utiliser des noms différents pour des balises différentes, ou s'assurer qu'il n'y a pas d'ambiguïté possible entre deux balises de même nom (par exemple, faire en sorte qu'elles soient à des endroits différents dans la structure de sorte qu'elles ne puissent être confondues). Cette convention est importante pour la lisibilité du document XML par un concepteur, mais également pour la lisibilité des programmes qui seront amenés à traiter l'information contenue dans ce document XML (notamment au niveau de la validation, voir chapitre 5 : Validation de documents XML).

- Bien que d'autres caractères soient autorisés en début de nom, utiliser uniquement des lettres. C'est beaucoup plus lisible.

Dans le même ordre d'idées, l'usage d'indentation et de retour à la ligne dans l'écriture du document n'est pas obligatoire mais met en évidence la structure du document. De même, l'emploi de commentaires pour améliorer la compréhension du texte est une très bonne pratique. C'est d'autant plus vrai dans un environnement de travail où plusieurs personnes sont amenées à travailler sur le même document. Ni l'auteur ni ses collaborateurs ne sont des machines qui analysent et interprètent le document en le lisant de manière linéaire. L'aspect visuel du document est important pour l'efficacité du travail. A noter enfin que l'utilisation d'un éditeur « intelligent » syntaxique de documents XML (ce type d'éditeurs utilise couleurs, indentations et auto-completion des structures, pour aider l'utilisateur) est un grand confort pour un auteur.

Après ces considérations d'ordre pratique, revenons à présent à l'analyse de la syntaxe proprement dite.

Toute balise XML est ouverte à un moment et doit être fermée à un autre.

La syntaxe est la suivante :

```
<annuaire>.........</annuaire>
```

Tout document XML (hors en-tête et commentaires) commence par une balise ouvrante et se termine par la fermeture de cette balise. La balise la plus extérieure d'un document XML est appelée ***balise racine*** du document XML.

Une balise XML peut entourer des données, par exemple :

```
<numTéléphone>02 23456789</numTéléphone>
```

Elle peut également entourer une sous-structure XML (cfr. par exemple la balise `<entrée>` de la figure (2.1))

Enfin, une balise peut entourer un mélange des deux, par exemple :

```
<texteLibre>ceci est un <b>texte</b> libre </texteLibre>
```

Ces considérations ont une conséquence directe sur la structure d'un document XML : l'imbrication stricte de ses composants.

Une balise ouverte qui entoure une autre balise ouverte doit nécessairement se fermer après la balise qu'elle entoure, sous peine de ne pas respecter la stricte imbrication (figure (2.2)).

Figure 2.2. Stricte imbrication des balises XML

```
<balise1>
     <balise2>
          ....
</balise1>
</balise2>

KO
```

```
<balise1>
     <balise2>
          ....
</balise2>
</balise1>

OK
```

Cette règle de modélisation est loin d'être une contrainte tant elle est logique.

Il arrive néanmoins qu'une balise se suffise à elle-même d'un point de vue sémantique, ne concerne aucune donnée spécifique du document et de ce fait n'entoure aucune autre structure. Il existe un raccourci de syntaxe pour ce type de balise :

`<annuaire/>` qui est équivalent à `<annuaire></annuaire>`

Un document XML dont la structure est entourée d'une balise racine et qui satisfait à la condition de stricte imbrication de ses composants est un document XML ***bien formé***.

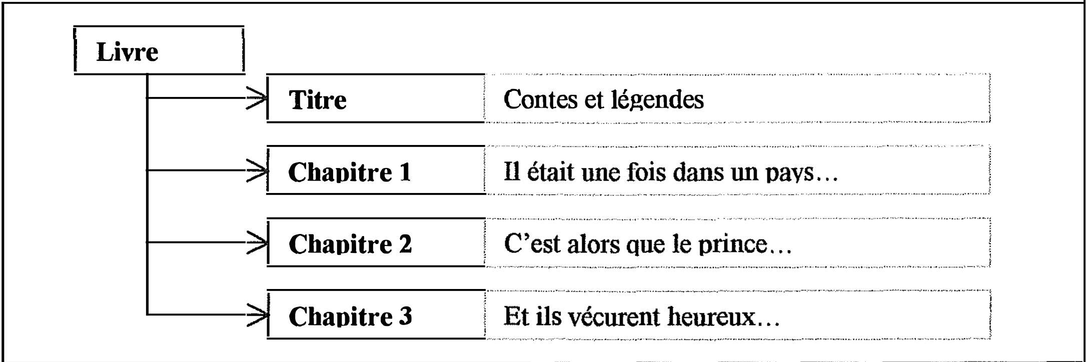
Beaucoup d'éditeurs « intelligents » de documents XML ainsi que certains programmes de traitement vérifient que ces deux conditions sont bien remplies par les documents XML qu'ils sont amenés à traiter. Seuls les documents XML bien formés sont traités par les technologies XML, d'où l'importance de faire attention lors de la rédaction de tels documents.

Un document XML bien formé peut être considéré comme une ***structure arborescente*** dans laquelle la balise racine fait office de racine de l'arborescence, les balises imbriquées font office de nœuds et les données font office de feuilles.

Ainsi, la figure (2.3) est une représentation arborescente de la structure suivante :

```
<livre>
            <titre>Contes et légendes</titre>
            <chapitre1>Il était une fois dans un
            pays…</chapitre1>
            <chapitre2>C'est alors que le
            prince…</chapitre2>
            <chapitre3>Et ils vécurent heureux…</chapitre3>
</livre>
```

Figure 2.3. Représentation arborescente d'un document XML



Cette représentation est très utile pour visualiser une structure XML, à des fins de documentation, ou encore pour comprendre le fonctionnement des technologies XML et en particulier la manière dont celles-ci parcourent le document.

A noter que certains IDE XML tels que XMLspy proposent cette représentation parmi leurs formats de sortie.

2.5. Balises et attributs

Il peut arriver qu'un concepteur veuille utiliser une balise plus générique pour entourer un ensemble de données et veuille donner plus de détails en ajoutant des données supplémentaires.

Reprenons le cas des entrées d'un annuaire de téléphone tel que nous l'avons défini plus haut, à savoir :

- Chaque entrée est soit à caractère privé, soit à caractère professionnel.
- Chaque entrée de l'annuaire décrit un contact. Chaque contact a un nom et un numéro de téléphone.

Regardons le concept d'entrée. Il existe deux catégories d'entrées, les entrées à caractère professionnel et les entrées à caractère privé et toutes les deux sont structurées de la même manière.

Il semble dès lors intéressant d'utiliser une balise commune <entrée> et d'ajouter dans les données qu'elle entoure les informations relatives au caractère privé ou professionnel du contact.

Une telle structure ressemblerait à ceci :

```
<entrée>
        <type>Privé</type>
        <nom>Jean Dupont</nom>
        <numTéléphone>04 558956</numTéléphone>
</entrée>
```

XML permet un autre type de modélisation pour ce genre de cas : les attributs de balise. Un attribut de balise est une donnée qui est rattachée à une balise ouverte et qui aide à la description des données que la balise entoure. La syntaxe est la suivante :

```
<nomBalise  attribut1="…" attribut2= "…">…</nomBalise>
```

Ce qui donne pour notre exemple :

```
<entrée type="Privé">
        <nom>Jean Dupont</nom>
        <numTéléphone>04 558956</numTéléphone>
</entrée>
```

Les remarques concernant les noms de balise et les conventions de notation sont bien sûr d'application pour les noms d'attributs.

L'introduction de ce nouveau mécanisme ouvre un débat quant à la technique de modélisation à employer. Le choix quant à l'emploi d'attributs pour décrire une donnée revient au concepteur du document. Dans la pratique, les attributs sont souvent utilisés pour décrire des "méta-données" (données décrivant les données).

De par sa généricité, XML permet une infinité de modélisations différentes pour représenter un même problème. La figure (2.4) illustre quelques autres alternatives de modélisation pour les entrées de notre annuaire de téléphone.

Figure 2.4. Modélisations alternatives pour les entrées de l'annuaire

```
<entrée type="privé" nom="Jean Dupont">04 558956</entrée>

<entrée type="professionnel" nom="Jack Ryan">02 445451</entrée>


<contact_privé>
      <nom>Jean Dupont</nom>
      <numTél>04 558956</numTél>
</contact_privé>

<contact_prof>
      <nom>Jack Ryan </nom>
      <numTél>02 445451</numTél>
</contact_prof>


<entrée type="privé" nom="Jean Dupont" numTél="04 558956" />

<entrée type="prof" nom="Jack Ryan " numTél="02 445451" />
```

Le choix d'une structure par rapport à une autre revient au concepteur. Il n'y a pas selon nous de « meilleure modélisation possible » pour un problème donné. Il existe néanmoins des travaux qui vont dans ce sens et qui proposent des méthodes de modélisation mais ceci dépasse le cadre de notre travail.

2.6. Grammaire et validation de documents XML

Nous l'avons déjà mentionné plusieurs fois, le langage XML est générique. L'auteur d'un document XML est libre de choisir ses noms de balises et la structure de son document sans restriction majeure.

Cette généricité de XML pose néanmoins quelques problèmes que nous nous bornerons à citer pour l'instant :

- Difficultés de traiter les données d'un document XML de manière automatisée si on n'a de prime abord aucune garantie sur la structure du document.

- Difficultés pour plusieurs auteurs de travailler sur un même document XML si chaque auteur est libre de toute contrainte de structure et ne s'efforce pas de respecter un référentiel commun.

Ces deux difficultés sont des enjeux clés dans l'utilisation de documents XML comme messages intra ou inter-systèmes. Ce sont des enjeux de ***l'interopérabilité.***

Appelons ***grammaire d'un document XML*** une description de la structure d'un document XML. Cette définition englobe par souci de généralité aussi bien les descriptions informelles que les descriptions formelles.

XML propose tout un ensemble de formalismes de grammaire XML, nous nous contenterons dans ce chapitre de n'en citer que quelques-uns sans entrer dans les détails.

- Les ***DTD*** (document type définition) : forme la plus simple de grammaire formelle XML.

- Les ***XML Schema*** : grammaire formelle décrivant un document XML à l'aide d'un document XML.

Nous détaillerons ces deux formalismes (parmi d'autres) dans le chapitre 5 : validation de documents.

Nous appellerons ***vocabulaire d'une grammaire XML***, toute partie de structure XML qui répond aux règles de la grammaire en question.

Les technologies XML permettent alors de vérifier si la structure d'un document XML correspond bien à la structure décrite par une grammaire donnée. En plus de ceci, il semble intéressant de pouvoir instaurer des règles de cohérence de données dans un document XML, voire entre plusieurs. Le terme le plus souvent utilisé pour décrire cette problématique est la ***validation de documents XML.*** Cette problématique, ces enjeux et développements font l'objet d'un chapitre complet (chapitre 5) dans le présent travail.
La référence à un quelconque de ces deux systèmes se fait dans l'en-tête du document XML. La figure (2.1) montre un exemple de référence à une DTD.

2.7. Les namespaces XML

XML propose également un mécanisme intéressant, à savoir le mécanisme des ***namespaces.***

Le but des namespaces est le suivant :

- C'est un mécanisme qui permet de mélanger du vocabulaire de plusieurs grammaires XML distinctes.

- Ce mécanisme permet également d'identifier de manière distincte les balises d'un document XML

Il est difficile d'exprimer précisément et en quelques lignes toute l'utilité des namespaces.

Nous nous contenterons dès lors de citer un exemple dans lequel la problématique peut être résolue par l'emploi de namespaces.

Supposons qu'un document XML contienne des données utilisées par plusieurs logiciels concurrents ainsi que des données ajoutées spécifiquement par ces logiciels, sans concertation ni structure commune. On peut très bien imaginer que pour chacun de ces logiciels, les informations ajoutées par les autres ne soient pas utiles. De plus, dans le traitement des données de ce document XML de façon automatisée, la présence de données écrites par d'autres logiciels représente du bruit qui peut être très problématique. Pour résoudre ce problème, une solution possible serait que chaque logiciel utilise un namespace personnel pour ses données et ne tienne pas compte dans le traitement du document des données se trouvant dans un namespace d'un logiciel concurrent.

Cet exemple abstrait est similaire à la solution que nous utiliserons dans l'étude de cas ***génération de formulaire***, disponible au chapitre 7 : génération de formulaire.

La syntaxe pour les namespaces XML est la suivante :

- Dans la balise racine d'une structure, déclarer le namespace comme suit :

  ```
  xmlns : nomNamespace= "URI"
  ```

  Et ce pour chaque namespace en présence. Nous conseillons les mêmes conventions de notation pour les noms de namespaces que celles décrites pour le reste des identificateurs (balises et attributs). A noter que l'URI (« ***Uniform Resource Identifier*** ») est bien souvent une URL (« ***Uniform Resource Locator*** ») qui ne correspond pas nécessairement à une ressource. Le but ici est simplement d'identifier de manière unique le namespace en le liant à un identificateur de ressources. En pratique, les auteurs de document XML utilisent parfois comme URL la localisation d'une grammaire formelle (DTD, schema, ou simple texte descripteur) qui décrit la structure contenue dans le namespace mais ceci n'est pas obligatoire.

- Chaque balise ou attribut contenu dans un namespace est identifié de cette manière :

```
nomNamespace : nomBaliseOuAttribut
```

Pour fixer les idées, la figure (2.5) est un exemple de document XML contenant des éléments (en gras) de différents namespaces. Cet exemple est tiré de [Lbl01]

Figure 2.5. Exemple d'utilisation de namespace

```
<organisation
xmlns:entreprise="http://www.entreprise.org"
xmlns:personne ="http://www.personne.org">
      <entreprise:nom>Bizib inc</entreprise:nom>

      <personne:nom>
            <personne:nomDeFamille>
                  Lafargue
            </personne:nomDeFamille>
            <personne:prénom>Paul</personne:prénom>
            <personne:fonction>PDG</personne:fonction>
      </personne:nom>

</organisation>
```

Une balise peut contenir un attribut faisant partie d'un autre namespace. De plus, un attribut qui n'est pas explicitement préfixé par un nom de namespace fait partie du même namespace que la balise à laquelle il est attaché.

Une remarque enfin sur le namespace par défaut. Toute les balises et attributs dont le nom n'est pas préfixé par un identificateur suivi de « : » fait partie du namespace par défaut, ce namespace n'a pas besoin d'être déclaré.

3. Transformations de document XML : la technologie XSL

3.1. Objectif du chapitre

Ce chapitre a pour vocation d'introduire la technologie de transformation XSL. Après avoir introduit la technologie en elle-même, ce chapitre abordera tour à tour les différents sous-ensembles de XSL. Et plus particulièrement XPath, le langage de sélection, et XSLT le langage de transformation.

3.2. Introduction

Jusqu'ici, nous avons exploré avec plus ou moins de détails la syntaxe des documents XML. Il est temps d'aborder maintenant la manipulation de tels documents.

Nous n'avons pas l'intention de décrire un panorama de toutes les manipulations possibles de documents XML. Nous aborderons une manipulation particulière que nous allons souvent utiliser dans les chapitres ultérieurs : la transformation de documents XML par les technologies XSL.

Les documents XML sont utilisés comme réceptacles de données et bien souvent comme média de communication entre logiciels dans le cadre de l'interopérabilité. Dans ce cas de figure, il est parfois nécessaire de devoir récupérer les données du document en entrée et de les mettre dans une structure compréhensible par le logiciel de traitement. Il s'agit donc de convertir la structure de données du document en entrée afin qu'en sortie, la structure soit adaptée au logiciel de traitement. Nous appellerons ***transformation de document XML*** le mécanisme de conversion d'une structure XML à une autre. Cette transformation se fait par le biais de technologies spécifiques, en particulier la technologie XSL.

3.3. La technologie XSL

XSL (eXtensible Stylesheet Language) est la technologie de transformation de documents proposée par le consortium W3C (plus d'informations sur le W3C en [W3c02]). XSL comprend 3 sous-ensembles :

- XSLT : XSL-Transformation : langage de transformation de documents XML. C'est le cœur de la technologie XSL. Beaucoup de gens font usage de XSLT lorsqu'ils disent utiliser « le langage XSL ». Le langage XSLT est une recommandation du consortium W3C depuis novembre 1999.

- XPath : Langage de sélection de données d'un document XML. Ce langage est à XML ce que le SQL est aux bases de données. Il est utilisé intensément par le langage XSLT dans le cadre de son fonctionnement. Ce langage est lui aussi une recommandation du consortium W3C depuis 1999.

- XSLFO : Grammaire XML de mise en forme de données. Plutôt que de décrire la sémantique des données qu'elles encadrent, les balises XSLFO décrivent la manière dont les données doivent être affichées. Nous

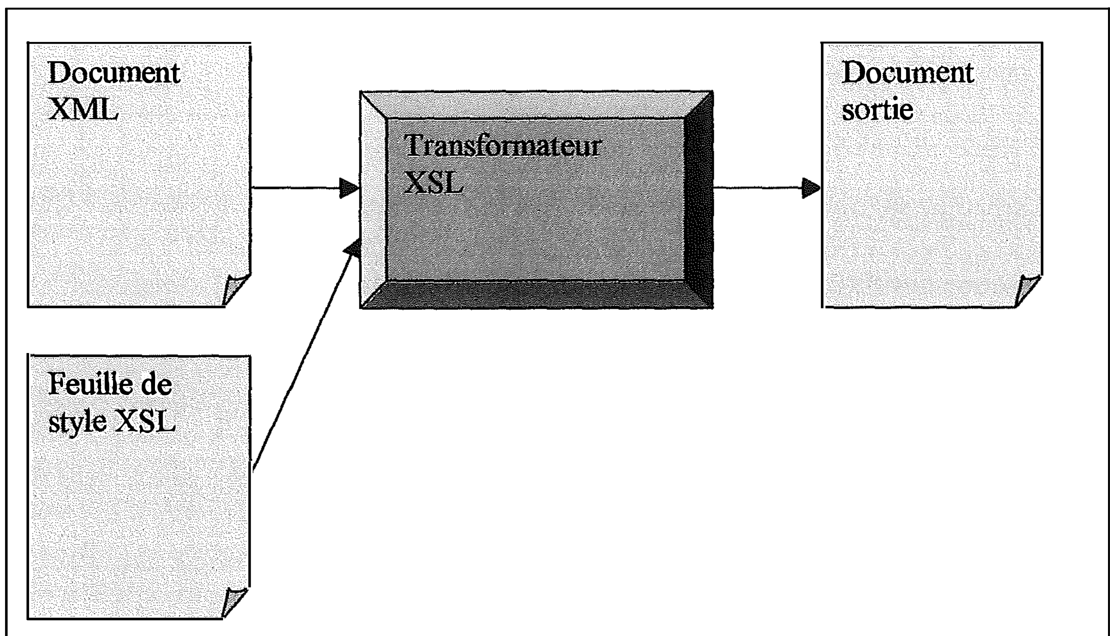
détaillerons XSLFO dans le chapitre consacré à la ***publication de document XML.***

XSL comprenant ces trois sous-ensembles est une recommandation du consortiumW3C depuis octobre 2001.

Le « transformateur » XSL prend en entrée un document XML et une « feuille de style XSL ». A partir du document XML et des instructions de transformations décrites dans la feuille de style XSL (via le langage XSLT), le transformateur génère le document de sortie.

La figure (3.1) décrit ce schéma de fonctionnement.

Figure 3.1. Fonctionnement de XSL



L'usage de XSL ne s'arrête pas à la transformation de documents XML vers un autre document XML et à la publication de documents XML. Nous avons relevé quelques autres applications intéressantes de XSL dont voici une brève description. Ces applications sont toutes des cas particuliers de l'usage normal de XSL mais qui valent la peine d'être approfondies car elles ajoutent véritablement une « corde » à « l'arc XSL » .Ces applications sont :

- ***Génération de code à partir de structures XML*** : La technologie XSL peut être utilisée pour transformer un document XML en un document texte ou apparenté. Parmi les débouchés possibles, nous analyserons (dans le chapitre 7 : Génération de formulaire) les possibilités, à partir d'un document XML, de générer de manière automatique du code dans un langage de programmation quelconque.

- ***Sélection de données dans un document XML*** : La technologie XSL sert à transformer un document XML en un autre document XML. En particulier, XSL peut être utilisé pour « filtrer » un document XML et produire en sortie un document XML de même structure mais dans lequel on a enlevé une partie des données qui ne remplissent pas certains critères.

3.4. A quoi ressemble une feuille de style XSL ?

Ainsi que nous l'avons fait dans notre approche des documents XML, voici un exemple de feuille de style XSL (figure (3.2)). Nous retrouverons cet exemple tout au long du chapitre.

Comme toujours, l'emploi d'indentations et retours à la ligne est facultatif mais fortement conseillé pour la lisibilité du document.

Figure 3.2. Exemple de feuille de style XSL

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

<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:fo="http://www.w3.org/1999/XSL/Format">

<xsl:template match="/">
      <téléphone>
            <xsl:apply-templates select="child::*"/>
      </téléphone>
</xsl:template>

<xsl:template  match="entrée">
      <xsl:element name="contact">
            <xsl:attribute name="typeContact">
                  <xsl:value-of select="@type"/>
            </xsl:attribute>
            <xsl:element name="nomContact">
                  <xsl:value-of select="./nom"/>
            </xsl:element>
      </xsl:element>
</xsl:template>

</xsl:stylesheet>
```

Figure 3.3. Résultat de l'application de la feuille XSL de la figure (3.2) sur le document XML de la figure (2.1)

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

<téléphone>
     <contact typeContact="profesionnel">
          <nomContact>John Doe</NomContact>
     </contact>

     <contact typeContact="privé">
          <nomContact>Aline Dubois</NomContact>
     </contact>

</téléphone>
```

Quelques remarques à propos de cet exemple :

Il s'agit bien d'une transformation d'une structure XML à une autre. La figure (3.3) reprend une partie des informations de la figure (2.1) mais dans une structure différente de la première. Les numéros de téléphone des contacts ne sont pas repris dans cette nouvelle structure.

Pour que la transformation se fasse, il faut mettre en relation d'une manière ou d'une autre la feuille de style XSL au document XML sur lequel elle travaille. Plusieurs méthodes sont possibles

- Faire mention de la feuille de style XSL à utiliser dans l'en-tête du document XML. Tout comme pour la validation, il est possible d'indiquer dans l'en-tête d'un document XML la feuille de style XSL à lui appliquer. Il suffit ensuite de lire le document XML avec un logiciel de transformation. La syntaxe est la suivante :

  ```
  <?xml-stylesheet type="text/xsl"
  ref="http://urltoxsl.com"?>
  ```

- Utiliser un logiciel de transformation qui prend en entrée de manière séparée un document XML et une feuille de style XSL. C'est le cas de Xalan chez Apache [ApXa02].

Une feuille de style XSL est elle-même un document XML ! Les balises XSL sont des balises XML tout à fait normales utilisant un namespace appelé `xsl`. Cette particularité implique plusieurs conséquences intéressantes :

- Une feuille de style XSL peut être validée.
- Une feuille de style XSL peut être transformée par la technologie XSL.
- Une feuille de style XSL peut être générée à partir de la technologie XSL

Nous n'aurons pas l'occasion d'explorer ces pistes en profondeur mais nous détaillerons une application (Schematron, décrit au point 5.5) qui exploite la génération de feuille XSL par XSL..

Le code présent dans la feuille de style XSL est du code XSLT. XSLT utilise XPath pour sélectionner des parties de document à traiter. Il semble logique de nous intéresser donc à XPath en premier lieu.

3.5. Le langage de sélection XPath

Comme annoncé plus haut (au point 3.3), XPath est un langage de requêtes qui permet de sélectionner des données dans un document XML (il est défini en [Xpa99]). Son rôle est crucial dans la technologie XSL car c'est XPath qui permet à XSLT de sélectionner les parties de structure sur lesquelles il va ensuite travailler.

Pour comprendre XPath, il est utile de considérer l'aspect «arborescence» des documents XML; la figure (2.3) est un modèle graphique de cet aspect «arborescence».

XPath utilise deux concepts principaux que nous allons introduire : les location path et les prédicats.

3.5.1. Les location path

Le concept de location path est le concept principal de XPath. Un location path est une expression qui décrit la position de sous-structures XML appartenant au document. La notion de location path est entièrement basée sur l'aspect «arborescence» de XML.

La sélection décrite par un location path est appelée « node-set ».

Pour introduire les location path, nous utiliserons deux concepts de relation entre balises :

Une balise A est fille/enfant d'une autre balise B si la structure attachée à A est entièrement contenue dans la structure attachée à B. De manière inverse, on dit que B est le parent de A

La syntaxe utilisée pour décrire la position où chercher dans l'arborescence est très facile et rappelle la syntaxe utilisée pour les URL ou encore les arborescences des systèmes de fichiers :

```
nomBalise1/nomBalise2/…
```

Ainsi, le location path `/document/élément1/élément2` décrit bien la position d'une balise `élément2`, fille d'une balise `élément1`, elle-même fille d'une balise `document`.

La figure (3.4) donne un exemple de location path et la structure sur laquelle il pointe.

Figure 3.4. Location Path, premier exemple

Le location path `/document/élément1/élément2` pointe sur la structure en gras.

```
<document>
      <élément1>
            <élément2/>
      </élément 1>
      <élément2/>
</document>
```

A noter qu'un location path peut pointer sur plusieurs structures qui remplissent ses critères. Ainsi, la figure (3.5) montre un location path qui pointe sur plusieurs balises `<élément2>`.

Figure 3.5. Location Path, deuxième exemple

Le location path `/document/élément1/élément2` pointe sur les structures en gras.

```
<document>
      <élément1>
            <élément2/>
      </élément1>
<élément1>
            <élément2/>
      </élément1>
      <élément1>
            <élément2/>
      </élément1>
      <élément2/>
</document>
```

Enfin, la notation `//élément1` permet de sélectionner toutes les balises élément1, quelles que soient leurs positions.

Les location path cités ci-dessus peuvent s'avérer utiles si l'on connaît exactement la structure du document. Mais parfois il pourrait être intéressant de pouvoir sélectionner un élément sans pour autant en connaître le nom. En effet, il peut arriver que l'on ne connaisse pas forcément tous les éléments du document, ou alors que leur nombre soit trop élevé. Cette possibilité permet beaucoup de généricité.

Ainsi XPath met à notre disposition le symbole "*" à cet effet.
"*" remplace n'importe quel nom de balise présent dans le document (figure (3.6)).

Figure 3.6. Location Path, utilisation de ""*

Le location path `document/élément1/*` pointe sur tous les éléments qui ont comme ancêtres `document` et `élément1`.

```
<document>
      <élément1>
            <élément2/>
      </élément1>
      <élément1>
            <élément2/>
      </élément1>
      <élément1>
            <élément2/>
      </élément1>
      <élément2/>
</document>
```

On aurait pu également l'écrire de la façon suivante :
`/*/*/élément2`
Ce qui correspond à tous les éléments `élément2` qui ont 2 ancêtres.

Enfin, notons que le location path « / » désigne la balise racine d'un document.

3.5.2. Les prédicats

Nous avons vu que les location path peuvent sélectionner plusieurs structures qui correspondent à leurs critères, les prédicats permettent d'affiner cette sélection en posant des conditions supplémentaires sur les balises à sélectionner. Les conditions peuvent porter sur la position de la balise dans le document (1ère occurrence de balise avec ce nom,...), la valeur des attributs qui lui sont attachés, la valeur des données qu'elle entoure, ...

Parmi toutes les possibilités de prédicats possibles, et dans le cadre de notre travail, nous indiquerons la syntaxe générale pour l'utilisation des prédicats et nous l'illustrerons par des exemples courts.

Les prédicats s'écrivent directement dans les location path. Un prédicat est écrit entre crochets [et] à côté de n'importe quel nom de balise compris dans le location path et affecte celle-ci dans son interprétation.

La figure (3.7) montre deux exemples de location path contenant des prédicats.

Figure 3.7. Location path avec prédicats.

Exemple 1 :

Location path : `//livre[position()>1]/auteur[.="Camus"]`

Cette requête sélectionnera tous les éléments de type « livre », dont la position dans l'arbre (numérotation des sous-éléments) est supérieure à 1 et dont l'auteur est Camus. Un seul élément répond à ce critère, c est le troisième livre. (Le premier livre de Camus dans le document a la position 1 et donc ne satisfait pas au prédicat `[position() > 1]`

```
<bibli>
      <livre>
            <auteur>Camus</auteur>
      </livre>
      <livre>
             <auteur>Albert</auteur>
      </livre>
      <livre>
             <auteur>Camus</auteur>
      </livre>
</bibli>
```

Exemple 2 :

Location path : `//*[string-length(name()) = 3]`

Cette requête sélectionnera tous les fils dont la balise a une longueur égale à 3.

```
<aaa>
      <b/>
      <cccc>
            <ddd/>
      </cccc>
</aaa>
```

3.5.3. Sélection d'attributs avec XPath

Jusqu'ici nous n'avons parlé que d'éléments. Or, dans un document XML, nous avons vu qu'il était possible de définir des attributs aux éléments. Il est également possible de sélectionner les attributs via XPath ou encore de faire une sélection des éléments suivant leurs attributs.
La sélection d'attributs est possible grâce à l'utilisation du symbole "@" (figure (3.8)).

Figure 3.8. Location Path, utilisation de "@"

Le location path `//élément1[@nom]` pointe sur tous les `élément1` ayant un attribut `nom`.

```
<document>
      <élément1 nom="Camus"/>
      <élément2 tél ="1234"/>
      <élément3 chien="oui" >
            <élément5 nom="toutou"/>
      </élément3>
</document>
```

Le location path `//@nom` pointe sur tous les attributs `nom` peu importe où ils sont.

```
<document>
      <élément1 nom="Camus"/>
      <élément2 tél ="1234"/>
      <élément3 chien="oui" >
            <élément5 nom="toutou"/>
      </élément3>
</document>
```

Ainsi, le `nom` attribut de l'`élément 1` et le nom attribut de l'`élément5` sont sélectionnés malgré leur position différente dans le document.

3.5.4. Les « context-nodes » et les modes de recherche avec XPath

XPath permet de sélectionner des éléments dans un contexte donné.

Appelons « context-node » ou *balise courante* une balise XML sélectionnée par une expression XPath. Nous aimerions pouvoir sélectionner grâce à XPath une série d'autres balises qui sont en relation avec notre « context-node ».

Il existe en XPath plusieurs autres modes de recherche qui permettent depuis une balise donnée de sélectionner d'autres balises que ses filles.

La syntaxe pour utiliser ces modes de recherche est la suivante :

```
Mode ::path
```

Nous citerons quelques-uns uns des modes de recherche fréquemment utilisés :

- `Child` : mode de recherche par défaut, du parent vers ses enfants.
  Exemple : `Child::*` sélectionne tous les éléments fils de l'élément actuel.

- `Self` : c'est le mode de recherche qui permet de sélectionner la balise elle-même. Le caractère "." est un raccourci de syntaxe pour `Self::*`.

- `Ancestor` : mode de recherche d'une balise vers ses ancêtres.
  Exemple : `Ancestor::élément1` sélectionne tous les ancêtres de la balise courante dont le nom est élément1.

Ces autres modes de recherches sont disponibles dans l'annexe A.1.

3.6. Le langage de transformation XSLT

3.6.1. Présentation générale

Le langage XSLT est au cœur de la technologie XSL. C'est lui qui s'occupe d'effectuer la transformation proprement dite (il est défini en [Xslt99]).

XSLT s'utilise par le biais de feuilles de styles XSL.

Toute feuille de style a pour balise racine la balise `<xsl:stylesheet>`

Une feuille de style XSL est organisée en règles appelées template. Chaque template décrit comment transformer une sous-structure particulière du document XML.

En voici un exemple simple :

```
<xsl:template match="/">
      <téléphone>
           <xsl:apply-templates select="child::*"/>
      </téléphone>
</xsl:template>
```

L'attribut match contient une expression XPath qui sert à déterminer quand appliquer la template.

Le contenu d'une template représente la structure XML que le processeur XSL va générer lorsqu'il interprétera cette template.

Ce contenu peut contenir des structures XML, du texte libre, mais aussi des instructions XSLT qui serviront à naviguer dans le document XML (`<xsl:apply-template>`), à récupérer des données du document XML initial (`<xsl:value-of>`), à contrôler le flux (`<xsl:when>,<xsl:for-each>`) ou à créer de nouvelles structures XML (`<xsl:element>,<xsl:attribute>`).

3.6.2. Fonctionnement de XSLT

Pour nos besoins, nous illustrerons le fonctionnement de XSLT par un exemple déjà introduit dans ce chapitre (figure (2.1) comme document XML initial, figure (3.2) pour la feuille de style XSL et figure (3.3) pour le fichier XML résultat.). Cet exemple contient deux templates.

Le processeur XSLT utilise le principe du « pattern matching », il recherche dans la feuille de style XSL quelles sont les templates qui correspondent le mieux au(x) balise(s) courante(s) qu'il essaye de traiter. Si plusieurs templates s'appliquent, XSLT utilise un système complexe de priorités pour choisir laquelle appliquer.

Au départ du traitement, le processeur XSLT prend la balise racine comme balise courante. Dans notre cas, la seule template qui est susceptible de s'appliquer est la suivante :

```
<xsl:template match="/">
      <téléphone>
            <xsl:apply-templates select="child::*"/>
      </téléphone>
</xsl:template>
```

Cette balise décrit une structure de donnée qu'il faut créer en remplacement de la balise racine.

Cette template est récursive, elle contient une instruction `<xsl:apply-templates>` qui sélectionne un ensemble de balises comme nouvelles balises courantes et analyse quelles templates appliquer. Dans notre exemple, l'instruction `<xsl:apply-templates>` demande à rechercher une template à appliquer pour chaque balise fille de la balise racine. La seule template qui peut être appliquée est la suivante :

```
      <xsl:template   match="entrée">
            <xsl:element name="contact">
                  <xsl:attribute name="typeContact">
                        <xsl:value-of select="@type"/>
                  </xsl:attribute>
                  <xsl:element name="nomContact">
                        <xsl:value-of select="./nom"/>
                  </xsl:element>
      </xsl:element>
</xsl:template>
```

Cette template crée une structure XML à partir de données contenues dans la balise courante qui est transformée. Elle crée un élément appelé contact qui possède un attribut `typeContact`. Cette balise contact entoure une autre balise appelée `nomContact`. Les instructions `<xsl:value-of>` permettent de récupérer les données du document XML initial et de les recopier dans la structure. Les balises `<xsl:element>` et

<xsl:attribut> sont des instructions XSLT qui permettent respectivement de créer des structures XML, des balises et des attributs. Il est parfois pratique de créer ces structures de cette manière plutôt que de les créer en les écrivant telles quelles dans le corps des templates (notamment lorsqu'il faut remplir des attributs avec des valeurs récupérées du document XML initial).

L'appel de template précédent <xsl:apply-templates> avait sélectionné toutes les balises filles de la balise racine, la template que nous venons d'analyser doit donc être appliquée à toutes ces balises filles les unes après les autres. Le processeur XSLT les appliquera dans leur ordre d'apparition dans le document.

Enfin, une fois les balises désignées dans l'appel <xsl:apply-templates> traitées, le processeur XSLT continue le traitement de la template qui était appliquée sur la racine et termine le traitement du document. Le résultat final est bien celui de la figure (3.3).

Le fonctionnement de XSLT n'est pas sans rappeler celui des langages de programmation fonctionnels.

3.6.3. Mécanismes avancés

Pour terminer cet aperçu rapide de XSLT, nous développerons deux mécanismes intéressants que nous exploiterons souvent dans nos études de cas.

Il est parfois intéressant de pouvoir traiter une même structure XML de plusieurs manières différentes dans la même feuille de style XSL. Un exemple courant serait de convertir un document XML représentant un livre en une autre structure de livre dans laquelle les noms des chapitres, en plus d'être à leur place, se retrouvent dans une section table des matières en début de document. Dans cet exemple, il faudrait pouvoir traiter la structure qui contient chaque nom de chapitre deux fois dans le processus de transformation, une fois pour générer la table des matières, une deuxième fois pour mettre ce nom au début de chaque chapitre.

XSLT permet ce type de transformation grâce au mécanisme des ***modes.***

Via l'utilisation d'un attribut mode dans les balises <xsl:template> , XSLT permet de faire suivre un mode à une template. Il permet ensuite d'associer un mode à un appel template comme <xsl:apply-templates>. Lorsqu'un tel appel est effectué, le processeur XSLT ne retient que les templates qui suivent le mode sélectionné.

Dans une feuille de style XSL, il devient ainsi possible de créer plusieurs templates qui seront appliquées à une même structure du document XML initial. Il suffit alors au moment opportun dans le processus de transformation d'indiquer au processeur quelle mode adopter et ainsi d'appliquer la bonne template au bon moment.

Le choix du nom d'un mode est laissé à l'auteur de la feuille de style. Nous recommandons les conventions de notation habituelles de notation telles que décrites au point 2.4.

L'exemple de la figure (3.9.) illustre cet emploi de mode et la syntaxe à utiliser.

Figure 3.9. Exemple d'utilisation du mode :

Feuille de style:

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

<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:fo="http://www.w3.org/1999/XSL/Format">

<xsl:template match= "/">
  <annuaire>
      <noms>
        <xsl:apply-templates select="child::*" mode="nom"/>
      </noms>
      <numéros>
        <xsl:apply-templates select="child::*" mode="num"/>
      </numéros>
  </annuaire>
</xsl:template>

<xsl:template match= "entrée" mode="nom">
      <nom>
        <xsl:value-of select="./nom"/>
      </noms>
</xsl:template>

<xsl:template match= "entrée" mode="num">
      <num>
        <xsl:value-of select="./téléphone"/>
      </num>
</xsl:template>

</xsl:stylesheet>
```

Appliquée au document XML de la figure (2.1):

```
<annuaire>
      <noms>
       <nom>John Doe</nom>
       <nom>Aline Dubois</nom>
      </noms>
      <numéros>
       <num>03 44 57 89</num>
       <num>63 69 00 98</num>
      </numéros>
</annuaire>
```

Dans cet exemple, les balises `<entrée>` du document XML de la figure (2.1) sont traitées 2 fois, une première fois pour créer la structure `<noms>`, une deuxième fois pour créer la structure `<numéros>`. Les templates qui correspondent à ces deux traitements

sont appelées dans la template qui correspond à la racine via les instructions `<xsl:apply-templates>` auxquelles on a ajouté l'attribut mode. La première indique au processeur qu'il faut rechercher toutes les templates qui peuvent être assimilées aux filles de la balise racine et qui suivent le mode « nom », la deuxième indique qu'il faut rechercher celles qui peuvent êtres assimilées aux filles et qui suivent le mode « `num` ».

Nos études de cas manipuleront ce mécanisme assez fréquemment dans les feuilles de style qu'elles utilisent.

XSLT permet enfin de manipuler ***variables*** et ***paramètres***.

Les variables et les paramètres ont une portée qui dépend de l'endroit où ils sont déclarés. Tous deux peuvent être déclarés soit avant les templates dans le corps des stylesheet, soit avant toute structure dans le corps d'une template. Dans le premier cas, les variables et paramètres ont une portée qui s'étend à la stylesheet entière, nous les appellerons ***variables et paramètres globaux***, dans le deuxième cas, uniquement à la template concernée, ce sont des ***variables et paramètres locaux***.

Les paramètres globaux reçoivent leur valeur lors de l'appel de la stylesheet (lors de l'exécution du processeur XSLT bien souvent). Nous dirons d'une feuille de style qui possède des paramètres globaux qu'elle est ***paramétrique***.

Les paramètres déclarés dans une template reçoivent leur valeur lors de l'appel de cette template via ce type d'appel :

```
<xsl:apply-templates select="  ">
 <xsl:with-param name="  " select="  "/>
</xsl:apply-templates>
```

L'attribut `name` contient le nom du paramètre dont on donne la valeur et l'attribut `select` contient la valeur à lui donner. Si il y a plusieurs attributs, il suffit d'inclure plusieurs balises `<xsl:with-param>`

La déclaration d'un paramètre se fait via la balise `<xsl:param>`. Cette structure peut prévoir une valeur par défaut à donner au paramètre.

Les variables quant à elles reçoivent leur valeur au moment de leur déclaration.

Les valeurs des variables et paramètres peuvent être récupérées au sein des expressions XPath via la notation `$nomVarOuParam`. Si une de ces valeurs devait être appelée en dehors d'une telle expression, il conviendrait d'utiliser les caractères d'échappement { et } autour de l'appel.

Pour terminer, signalons une chose importante à propos des variables et paramètres. Leur valeur est ***fixée*** au moment de leur déclaration et ne peut être modifiée par le code XSLT. Ceci tient au mode de travail de XSLT qui ne fonctionne pas comme un langage impératif. Il est donc impossible de modifier une valeur de variable comme faisant part d'une séquence d'instruction.

La figure (3.10.) donne un exemple d'utilisation des variables et paramètres.

Figure 3.10. Utilisation des variables et paramètres

Feuille de style:

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

<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:fo="http://www.w3.org/1999/XSL/Format">

<xsl:variable
name= "nomRacine">annuaireTéléphone</xsl:variable>

<xsl:template match= "/">
<xsl:variable name= "nomContact">contact</xsl:variable>

  <xsl:element name="{$nomRacine}">
       <xsl:apply-templates select="child::*" >
<xsl:with-param name="nomContact" select="$nomContact"/>
       </xsl:apply-templates>
   </xsl:element>
 </xsl:template>

<xsl:template match= "entrée">
<xsl:param name="nomContact"></xsl:param>
     <xsl:element name="{$nomContact}">
       <xsl:value-of select="./nom"/>
     </xsl:element>
</xsl:template>

</xsl:stylesheet>
```

Appliquée au document XML de la figure (2.1):

```
<annuaireTéléphone>
  <nomcontact>
     John Doe
  </nomcontact>
  <nomcontact>
     Aline Dubois
  </nomcontact>
</annuaireTéléphone>
```

Dans cet exemple, nous avons mis les noms des balises à générer dans des variables, et nous passons un de ces noms de balises en paramètre pour illustrer le mécanisme de passage. Les valeurs des paramètres et des variables sont bien récupérées grâce à l'usage du caractère "`$`", les accolades "`{`" et "`}`" étant utilisées lorsque cette valeur apparaît en dehors d'une expression XPath comme par exemple dans l'attribut `name` de la balise `<xsl:element>`.

3.6.4. Références

Ceci termine notre présentation rapide du langage XSLT. Nous espérons avoir donné un bon aperçu de ses possibilités.

Pour les lecteurs qui souhaiteraient élargir leur connaissance de ce langage, nous les invitons à consulter un des très nombreux ouvrages sur le sujet et notamment [Kay01] qui nous a servi de référence lors de nos études de cas.

4. Publication de documents XML

4.1. Objectif du chapitre

Dans ce chapitre, nous introduirons la problématique de la publication de documents XML. Nous analyserons tour à tour les technologies XSLFO et la génération de code HTML (bien formé) à l'aide de XSLT.

Enfin nous analyserons plus en détail le "framework" de publication Cocoon, de chez Apache XML Project.

4.2. Introduction

Dans les chapitres précédents, nous avons mentionné à plusieurs reprises que XML était un langage de stockage d'information. Un document XML est avant tout structuré pour contenir les données de manière ordonnée. Cette structuration des données n'est pas nécessairement plaisante pour l'œil humain non averti.

Dans ce chapitre, nous allons analyser les possibilités que nous offrent les technologies XML pour publier des données XML.

Par publication, nous voulons dire mettre en forme les données de manière à ce qu'elles soient agréablement agencées dans un souci de lisibilité.

Il convient de séparer deux types de publication de données.

- Les délivrables : ce type de publication est destiné à l'impression sur papier. Les formats utilisés pour ce type de publication sont PDF, RTF,...

- La publication "en-ligne": ce type de publication est destiné à être consulté sur le World Wide Web. Il est souvent plus fantaisiste que le type délivrable.

Nous nous intéresserons aux deux types de publication, mais nous nous attarderons plus longtemps sur les outils de transformation qui permettent de mettre des données en page pour l'un ou l'autre de ces types.

Nous mentionnerons enfin les possibilités de publication dynamiques, différentes en fonction du contexte dans lequel elles ont été demandées.

4.3. Publication de document XML par défaut.

Il est parfaitement possible en XML de créer une structure de données claire, avec des noms de balises et attributs clairs et précis, et un jeu de commentaires servant à compléter la compréhension des données d'un document XML. Par l'emploi d'indentations, par l'usage de bonnes pratiques d'écriture et de bonnes conventions de notation, il sera possible de publier un document XML facile à lire.

C'est dans cette direction que travaillent certains navigateurs Web lorsqu'on leur demande d'afficher le contenu d'un document XML. Dans ce cas de figure, ces navigateurs analysent la structure arborescente d'un document XML et se chargent d'ajouter indentations couleur et systèmes de navigations sommaires (principalement de type « tree-view ») pour donner une vue structurée et confortable au document XML.

La figure (4.1) est une capture d'une sortie à l'écran d'un document XML traité par Microsoft Internet Explorer.

Figure 4.1. Output d'un document XML sous Internet Explorer

```
  <?xml version="1.0" encoding="UTF-8" ?>
- <annuaire>
  - <entrée type="professionnel">
      <nom>John Doe</nom>
      <téléphone>03 44 57 89</téléphone>
    </entrée>
  - <entrée type="privé">
      <nom>Aline Dubois</nom>
      <téléphone>63 69 00 98</téléphone>
    </entrée>
  </annuaire>
```

Cette vue, si elle reste utilisable et assez lisible, reste néanmoins austère et repoussante pour le commun des mortels et donc peu adaptée à une utilisation grand public.

Dans le même esprit mais plus poussé, certains IDE XML proposent des vues arborescentes de documents XML. Ce type de vue ne montre plus les balises telles quelles mais analyse la structure arborescente du document et propose une représentation graphique de l'arbre.

A nouveau, des mécanismes de navigation de genre « tree-view » permettent de déployer plus ou moins l'arbre sur l'écran.

La figure (4.2) montre un exemple d'une telle vue avec XMLSpy

Figure 4.2. Vue arborescente de document XML avec XMLSpy

XML
annuaire
entrée (2)

	= type	() nom	() téléphone
1	professionnel	John Doe	03 44 57 89
2	privé	Aline Dubois	63 69 0098

Bien qu'il s'agisse déjà d'une amélioration énorme par rapport à la vue décrite plus haut, on peut déplorer le manque de personnalisation. On est toujours en sous-utilisation des possibilités que nous offrent des technologies comme le WWW en matière de publication.

La solution idéale serait d'utiliser une technologie de publication de document qui soit complètement paramétrable par le programmeur. Une technologie qui, d'une certaine manière, permettrait au programmeur de transformer son document XML en une entité publiable. On pense immédiatement à la technologie XSL dont nous avons parlé au chapitre précédent.

La technologie XSL permet de répondre à nos besoins de publications de deux manières que nous analyserons tour à tour.

4.4. XSLFO (XSL Formating Objects)

Déjà introduit dans le chapitre précédent, XSLFO est une des composantes de la technologie XSL et ce depuis que XSL est devenu une recommandation du consortium W3C en octobre 2001 [Xsl02].

XSLFO est un vocabulaire XML dont le but est d'associer aux données qu'il structure une mise en page.

Le but de XSLFO est bien la publication de documents XML. Il est voulu aussi générique que possible pour permettre ensuite à des logiciels processeurs spécialisés de transformer ces documents au format XSLFO en document publiables. De cette manière, un seul document XSLFO permet une publication dans divers formats de sorties.

Nous citerons, sans détailler davantage, quelques exemples de processeurs, FOP pour la publication PDF entre autres et JFOR pour la publication RTF.

Nous n'entrerons pas dans les détails du formalisme XSLFO ou de l'utilisation de ces processeurs.
La figure (4.3) schématise le principe de fonctionnement de XSLFO; la figure (4.4) nous donne un exemple de ce à quoi ressemble un document XSLFO (cet exemple est tiré de [Lbl01]).

Figure 4.3. Fonctionnement de XSLFO

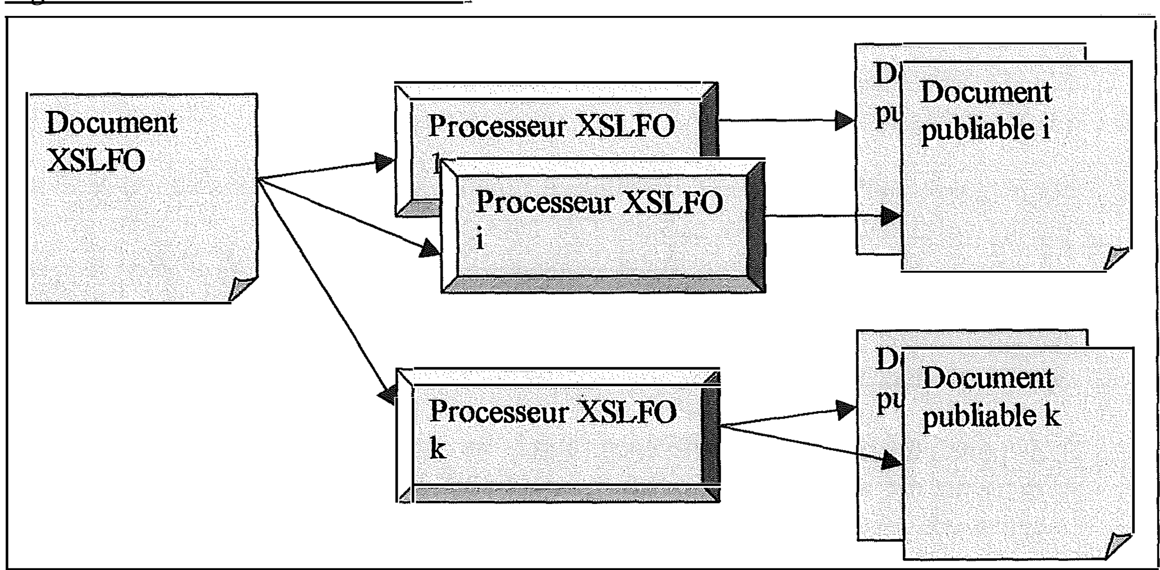


Figure 4.4. Exemple de document XSLFO et la sortie correspondante

Document :

```
<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">

     <fo:layout-master-set>
          <fo:simple-page-master>
               <fo:region-body/>
          </fo:simple-page-master>
     </fo:layout-master-set>

     <fo:page-sequence>
          <fo:flow>
               <fo:block font-size="14pt" font-weight="bold">
               Annuaire pages blanches
               </fo:block>

               <fo:block font-size="12pt">
               nom:Lorédan Larchey téléphone:0600010203
               </fo:block>

               <fo:block font-size="12pt">
               nom:Paul Lafargue téléphone:0603020100
               </fo:block>

          </fo:flow>
     </fo:page-sequence>
</fo:root>
```

Sortie :

Annuaire pages blanches
nom:Lorédan Larchey
téléphone:0600010203
nom:Paul Lafargue téléphone:0603020100

4.5. XSL et HTML

Nous avons déjà fait remarquer que les syntaxes de XML et HTML avaient beaucoup de points communs.

Pour rappel, HTML (HyperText Markup Language) est le langage de publication utilisé par le world wide web. Sa version 4.01 est une recommandation du W3C depuis le 24 décembre 1999. C'est certainement le standard de publication sur Internet le plus connu du grand public. Nous n'avons pas l'intention d'entamer une description complète de ce langage. Nous renvoyons le lecteur vers un des nombreux sites Internet, livres et travaux traitant du sujet.

Un point important de différence entre les mondes XML et HTML, au niveau syntaxique est le laxisme important existant dans le monde HTML pour tous les détails syntaxiques. Ainsi, les navigateurs et autres interpréteurs de code HTML sont conçus de manière relativement robuste pour être « fault-tolerant ». L'effet pervers de cette robustesse étant bien entendu que les mauvaises pratiques en matière de programmation HTML sont courantes et même parfois enseignées comme étant des bonnes pratiques !

Ainsi, il n'est pas rare et même très commun, de rencontrer des documents HTML avec les erreurs suivantes :

- Le document ne possède pas de balise racine.
- Certaines balises comme par exemple `<br>` ne sont pas fermées. Il faudrait écrire `<br/>`
- Certains documents HTML ne respectent même pas la stricte imbrication des éléments, les navigateurs actuels étant suffisamment « intelligents » pour détecter ces erreurs et les corriger lors de l'interprétation.
- La plupart des documents HTML ne seraient pas validés si on les confrontait à la grammaire « officielle » HTML 4.01

Appelons ***document HTML bien formé***, un document HTML qui est correct du point de vue syntaxique. Nous invitons le lecteur à se référer au chapitre 1 pour une description correcte de la syntaxe du balisage.

Il est important de constater qu'un document HTML bien formé peut être considéré comme une structure XML valide. A ce titre, il peut donc être traité comme tout document XML par la technologie XSL.

HTML étant le langage de publication le plus utilisé sur le World Wide Web, l'intérêt de cette dernière considération est immédiat : il est possible de transformer un document XML quelconque en un document HTML bien formé, publiable sur le World Wide Web, et ce via la technologie XSL.

Nous ne nous attarderons pas davantage sur les détails de cette transformation, vu que nous avons détaillé le langage XSLT dans le chapitre précédent.

Notons simplement que les navigateurs modernes possèdent un interpréteur XSL intégré, de sorte qu'un document XML dans l'en-tête duquel figure une référence à une feuille de style XSL, sera transformé avant d'être affiché à l'écran. Nous mettons toutefois en garde le lecteur à propos du fait que les interpréteurs XSL intégrés aux navigateurs ne supportent pas toujours toutes les fonctionnalités des dernières versions de XSLT.

Terminons cette réflexion par un dernier commentaire à propos de HTML. Si les documents HTML bien formés sont déjà un pas en avant dans le bon sens, le consortium W3C travaille sur un nouveau standard, le ***XHTML*** (Extensible HyperText MarkUp language). XHTML est une reformulation de HTML 4.0 dans le formalisme XML. Cette reformulation offre non seulement une solution au problème des erreurs syntaxiques, mais aussi au problème de la validation de documents HTML. Enfin, le mécanisme des namespaces XML permet d'envisager des extensions possibles à ce standard. XHTML 1.0 (seconde édition) est une recommandation du consortium W3C depuis le 1er août 2002.

4.6. Le « framework » de publication Cocoon

Nous avons déjà indiqué plusieurs méthodes pour relier des documents XML et des feuilles de style XSL. La méthode qui consiste à mettre une référence à une feuille de style XSL dans l'en-tête d'un document XML semble la plus facile à utiliser dans un environnement Web car l'association est alors transparente pour la personne qui demande à visualiser le fichier XML sur son navigateur. Cependant, cette méthode pose plusieurs problèmes :

- Le processeur XSLT utilisé dans ce cas de figure se trouve côté client. Ceci enlève donc à l'auteur de feuille de style XML le contrôle sur le rendu de sa transformation. Les processeurs XSLT présents dans les navigateurs ne supportent pas toujours toutes les fonctionnalités de XSLT, et n'interprètent parfois même pas les templates XSLT de la même façon.

- On ne peut relier à un document XML qu'une et une seule feuille de style XSL, ce qui peut restreindre les possibilités de traitement.

Il serait donc intéressant de trouver une autre solution. Le logiciel Cocoon est satisfaisant de ce point de vue.

Cocoon est un logiciel développé par Apache project [ApCo02] et qui est basé sur la technologie java, et plus précisément la technologie des servlets. C'est un « framework » de publication, un logiciel qui permet d'associer des documents XML avec des feuilles de style XSL pour les « servir » ensuite comme un serveur web classique le ferait. L'association entre XML et XSL se fait entièrement côté serveur. Dans le cadre de nos travaux, nous avons été amenés à utiliser la version 2 de ce logiciel au-dessus du moteur de servlet Apache Tomcat. C'est cette version que nous détaillerons dans le reste de ce chapitre, nous nous contenterons de dire Cocoon et non Cocoon 2 par facilité.

La figure (4.5) détaille le mode de fonctionnement de Cocoon. A partir des requêtes faites au serveur (1), Cocoon vérifie quelle feuille de style XSL est à appliquer à quel

document XML (2), récupère les deux fichiers (3), s'occupe de la transformation et renvoie le résultat au client (4). Cocoon est capable de servir des documents d'autres types que HTML. Il peut en effet interagir avec différents processeurs XSLFO et servir ainsi des documents PDF, RTF,... La mise en relation entre documents XML et feuilles de style XSL se fait via le document ***sitemap***.

Figure 4.5. Schéma de fonctionnement de Cocoon

4.7. Le document sitemap Cocoon

4.7.1. Généralités

Le document sitemap est la pierre angulaire du logiciel Cocoon. C'est ce document qui pilote véritablement tout le logiciel.

Le sitemap est lui-même un document XML. Cette constatation remarquable ouvre bien des pistes prometteuses. Comme n'importe quel document XML, le sitemap Cocoon peut être généré/transformé par la technologie XSL, voire au travers du « framework » Cocoon lui-même. Nous n'aurons malheureusement pas le loisir de développer cette constatation autant qu'elle le mériterait.

Le sitemap contient toutes les informations permettant au logiciel Cocoon de traiter les requêtes qui lui sont soumises. Nous n'avons pas l'intention d'explorer en détails toutes ses fonctionnalités, nous nous attarderons seulement sur celles qui nous paraissent les plus importantes.

Le sitemap contient tout d'abord toutes les informations techniques nécessaires au moteur Cocoon pour s'exécuter.

Ensuite, le sitemap contient une section dans laquelle il décrit comment mettre en relation les documents XML et XSL.

Nous appellerons **entrées du sitemap** les règles inscrites au sitemap qui mettent XML et XSL en relation.

Voici un exemple d'entrée présente dans cette section du sitemap.

```
<map:match pattern="entrX/entreprises.html">
      <map:generate src="docs/stages/entreprises.xml"/>
     <map:transform src="docs/stages/entrX_ent.xsl"/>
     <map:serialize type="html"/>
</map:match>
```

La signification de cette entrée est assez intuitive.

Si la requête du client est `entrX/entreprises.html`, alors il faut prendre le fichier XML `docs/stages/entreprises.xml`, le transformer avec la feuille de style XSL `docs/stages/entrX_ent.xsl` et le document final qu'il faut servir au client qui a fait la requête est un document de type HTML.

Si notre serveur Cocoon était installé sur une machine avec comme adresse de base `http://www.moncocoon.com` , la requête complète entrée par l'utilisateur qui serait assimilée à notre entrée de sitemap serait :

```
http://www.moncocoon.com/cocoon/entrX/entreprises.html
```

L'URL de base du serveur Cocoon est bien sûr paramétrable lors de son installation. Toutes les entrées du sitemap sont donc relatives à l'URL de base du serveur.

Sur la machine serveur, il n'y a pas de fichier appelé `entreprises.html`. L'URL utilisée par notre exemple d'entrée est donc fictive et sans équivalent physique. Ceci est assez logique vu que c'est précisément le but du serveur Cocoon que de générer ses sorties à partir d'autres documents. L'extension `.html` elle-même n'est absolument pas obligatoire pour la génération de documents HTML.

La balise `map:serialize` indique le type de document à générer. Dans le cas de HTML cette entrée n'apporte pas grand-chose, aucun traitement supplémentaire n'étant en effet à faire après l'application de la feuille de style XSL au document XML. En revanche, Cocoon permet, grâce à cette balise, de transformer un document XML en un document XSLFO avec XSL et d'ensuite utiliser un processeur XSLFO pour transformer ce document en un document publiable d'un type donné qui sera ensuite servi au client.

Rien n'empêche un même document XML de se retrouver comme générateur dans plusieurs entrées différentes. De cette manière, en fonction de l'URL entrée par l'utilisateur, le serveur Cocoon sert les mêmes données mais avec l'application d'une feuille de style différente. La figure (4.6) illustre ceci par un exemple pratique.

L'URL dans cet exemple est bien différente dans les deux entrées et permet à Cocoon de choisir entre deux feuilles de style XSL différentes selon la demande du client.

Figure 4.6. Entrées de sitemap avec document XML commun

```
<map:match pattern="entrX/entreprises.html">
     <map:generate src="docs/stages/entreprises.xml"/>
     <map:transform src="docs/stages/entrX_ent.xsl"/>
     <map:serialize type="html"/>
</map:match>

<map:match pattern="entrY/entreprises.html">
     <map:generate src="docs/stages/entreprises.xml"/>
     <map:transform src="docs/stages/entrY_ent.xsl"/>
     <map:serialize type="html"/>
</map:match>
```

4.7.2. Utilisation des « wildcards » et des « tokens »

Imaginons la situation suivante : nous aimerions que chaque requête de document HTML d'un répertoire donné mette en relation le document XML de même nom avec une feuille de style XSL commune à tous les documents XML. La solution simple serait de rajouter au sitemap une entrée par document XML. Malheureusement, cette solution est lourde à mettre en œuvre. Cocoon propose fort heureusement un mécanisme de « wildcard » et de « token » pour pallier à ce genre de situation.

Le caractère «* » dans un pattern du sitemap remplace un nombre quelconque de caractères jusqu'à occurrence d'un caractère structurel « / ».

Exemple :

`entreprises/entrX/entreprises .html` peut être assimilée au pattern `entreprises/*/*.html`

La chaîne de caractère « ** » dans un pattern du sitemap remplace un nombre quelconque de caractère, y compris le caractère structurel « / ».

Exemple :

`entreprises/entrX/entreprises .html peut être assimilée au pattern **/*.html`

Les chaînes de caractères associées aux wildcard peuvent être récupérées facilement afin de servir de variable dans le reste de l'entrée du sitemap. On y accède avec la syntaxe suivante {#} où # est le numéro d'ordre d'occurrence du wildcard dans le pattern.

Exemple :

Lors d'une requête `entrX/entreprises.html`, assimilée au pattern `*/*.html`, {1} prend la valeur « `entrX` » et {2} prend la valeur « `entreprises.html` ».

Ce mécanisme permet de créer des entrées génériques dans le sitemap, comme dans cet exemple:

```
<map:match pattern="entreprise/*.html">
      <map:generate src="docs/entreprises/{1}.xml"/>
 <map:transform src="docs/entreprises/entreprise.xsl"/>
 <map:serialize type="html"/>
</map:match>
```

Ce mécanisme de « wildcard » introduit la problématique de la généricité dans les sitemaps et en même temps la problématique de la sélection de règles. En effet, il devient possible que plusieurs entrées du sitemap soient des candidats possibles pour traiter une requête. Dans ce cas de figure, Cocoon utilise un mécanisme de priorité très simple, la première entrée du sitemap qui peut être utilisée est choisie. Ceci implique comme conséquence qu'il faut toujours dans un sitemap Cocoon ***mentionner les entrées les plus spécifiques avant les entrées plus génériques***, sous peine de voir les entrées spécifiques n'être jamais utilisées.

4.7.3. Paramètres du transformateur

Nous avons vu dans le chapitre précédent que les feuilles de style XSL pouvaient prendre des paramètres.

Cocoon permet bien évidemment de passer des paramètres aux feuilles de style qu'il utilise. La figure (4.7) nous montre un exemple de passage de paramètres :

Figure 4.7. Paramètres dans une entrée sitemap

```
<map:match pattern="entrX/entreprises.html">
     <map:generate src="docs/stages/entreprises.xml"/>
     <map:transform src="docs/stages/entreprises.xsl">
          <map:parameter name="entreprise" value="entrX"/>
     </map:transform>
     <map:serialize type="html"/>
</map:match>
```

La feuille de style XSL `entreprises.xsl` recevra un paramètre appelé « entreprise » dont la valeur sera dans ce cas précis « `entrX` »

En associant les mécanismes de wildcards et de paramètres, il est possible de récupérer des paramètres venant d'une URL et de les envoyer à la feuille de styles XSL.

4.7.4. Mécanismes avancés de sélection

Le mécanisme de « matching » utilisé par Cocoon permet déjà pas mal d'applications. Néanmoins, il reste certaines situations où ce mécanisme ne suffit pas à résoudre le problème de manière satisfaisante.

Imaginons le problème suivant : nous avons un document XML contenant des données que nous désirons publier. Nous souhaitons profiter au maximum des possibilités offertes des différents navigateurs Internet. Malheureusement, le niveau de compatibilité entre ces différents navigateurs est tel qu'il semble nécessaire d'utiliser des feuilles de style différentes selon le navigateur faisant la requête. Ce type de problème ne peut être résolu par le mécanisme de matching classique, il faudrait utiliser un mécanisme de détection/sélection plus puissant.

Cocoon propose un mécanisme de sélection permettant de résoudre ce problème. Via l'utilisation d'une nouvelle balise `<map:choose>`, Cocoon permet de tester certains paramètres d'environnement (navigateur mais aussi support des cookies, informations dans header,...) et en fonction de ces tests permet de choisir un transformateur plutôt qu'un autre Il est évidemment possible de choisir le document XML de base avec ce système (c'est-à-dire, en fonction de paramètre d'environnement, choisir un document XML spécifique.)

La figure (4.8) donne un exemple d'utilisation de ce mécanisme, l'exemple est tiré de [ApCo02]. Cet exemple attribue une feuille de style XSL différente selon le type de navigateur employé. Ce mécanisme comprend également une clause `<map:otherwise>` qui permet de déterminer un choix par défaut.

Figure 4.8. Utilisation de `<map:choose>`

```
<map:match pattern="docs/*.html">
  <map:generate src="xdocs/{1}.xml"/>

  <map:select type="browser">
     <map:when test="netscape">
          <map:transform src="stylesheets/netscape.xsl" />
      </map:when>
     <map:when test="explorer">
           <map:transform src="stylesheets/ie.xsl" />
     </map:when>
      <map:when test="lynx">
          <map:transform src="stylesheets/text-based.xsl" />
     </map:when>
     <map:otherwise>
           <map:transform src="stylesheets/html.xsl" />
      </map:otherwise>
  </map:select>

  <map:serialize/>

</map:match>
```

Faisons remarquer que ce mécanisme permet également à un serveur Cocoon de renvoyer différents résultats selon la plate-forme (OS ou même Hardware comme le WAP) . Ce mécanisme promet beaucoup de choses. Il est souvent appelé le "***multi-devicing***".

4.7.5. Conclusion

Le "framework" de publication Cocoon est un outil très complet. Nous n'avons fait ici qu'effleurer l'éventail de ses possibilités en nous concentrant sur les mécanismes qui nous semblaient les plus intéressants. Selon nous, les mécanismes présentés dans ce travail permettent une prise en main assez rapide du logiciel dans son exploitation la plus courante. Une étude plus approfondie de Cocoon pourrait faire l'objet d'un travail complet.

De part ses mécanismes de sélection et son architecture qui supporte la généricité, Cocoon se présente comme une alternative très attractive aux mécanismes de contenu dynamique dits classiques (base de donnée et script côté serveur). En séparant la mise en page des données des données elles-mêmes, Cocoon permet des économies d'échelle qui peuvent être importantes. Plutôt que d'abandonner complètement le système de base de données, Cocoon peut également devenir un complément aux mécanismes classiques, avec l'introduction d'une couche XML intermédiaire générée par les mécanismes classiques et une couche XSLT/cocoon pour la mise en page.

Enfin, nous n'avons fait que mentionner le mécanisme de "multi-devicing". Ce mécanisme à lui seul est une des originalités qui font le succès de Cocoon auprès des webmasters.

Pour plus d'informations à propos de Cocoon, nous renverrons le lecteur vers [ApCo02].

5. Validation de documents XML

5.1. Objectif du chapitre

Ce chapitre introduit la problématique de la validation. Il aborde tour à tour différents validateurs tels que les DTDs, W3C XML Schemas, Schematron et Xlinkit. Il détaille les mécanismes de chacun, explique leurs avantages et limites et en quoi ils complètent les autres.

5.2. Introduction.

Il semblait donc utile de pouvoir associer à ces technologies des mécanismes de validation des données contenues dans les fichiers XML, de façon à pouvoir s'assurer que ces données soient cohérentes. Ceci étant d'ailleurs un des points clés de l'interopérabilité actuelle.

Qu'entend-on par validation?
La validation est le processus qui permet de vérifier si un fichier XML correspond bien à un format ou à des règles données. Pour ce faire, nous allons aborder les standards qui définissent les structures des fichiers XML, ainsi que quelques validateurs. La validation peut avoir lieu sur la structure du fichier, sur son format, au niveau du typage de ses éléments et enfin, grâce à des règles que l'on a définies et qu'on voudrait vérifier.

La qualité des validations examinées grandit de manière progressive.
On commence par les DTD, qui sont simples mais très incomplètes, car ne permettant pas de typage précis (notamment le type PCDATA qui correspond aux chaînes de caractère, sans pour autant différencier les entiers, dates,...). Mais il ne faut pas oublier que les DTD étaient un des premiers formalismes.

La plupart des lacunes des DTD sont comblées par les XML Schemas, qui apparurent un peu plus tard. Ils permettent de typer les éléments de façon très précise (± 40 types). Et dans le cas où ce typage serait insuffisant, il est possible de définir ses propres types. Les XML Schemas permettent aussi de dénombrer de façon complète les éléments.
Mais cette validation est encore plus poussée par Schematron et Xlinkit. En effet ils proposent de définir des règles.

Schematron propose un système de règles qui sont des prédicats basés sur la logique booléenne. Ces règles sont écrites dans des fichiers XML Schemas qui sont ensuite traités grâce à des feuilles de style XSL pour obtenir un résultat.
Xlinkit est le dernier validateur examiné. Il est assez différent de Schematron, car permet de faire une validation entre plusieurs documents, ce qui est non négligeable quand on parle d'interopérabilité. Il est basé sur une technologie Java et permet d'exprimer des règles plus riches, car basées sur la logique des prédicats du 1er ordre.

5.3. Les DTDs

5.3.1. Qu'est-ce qu'une DTD ?

DTD (*Document Type Definition*)

La norme DTD a été émise par le W3C (World Wide Web Consortium) dans le but de pouvoir définir la grammaire des documents XML (W3C Recommandation du 6 octobre 2000 disponible sur [Dtd00])
Une DTD définit quelles balises (*tags*) et attributs (*attributes*) peuvent être utilisés pour décrire le contenu d'un document XML. C'est ce qu'on appelle la grammaire du document. Elle permet de savoir quelles balises sont permises, mais également de définir la structure du document.
Exemple : Une DTD pourrait autoriser qu'un élément OBJET soit à l'intérieur d'un élément LISTE, alors qu'une autre autoriserait le contraire.
Cela permet donc de s'assurer que l'information des « documents instances » aura le même format, la même structure, si on utilise la même DTD.
Une validation a lieu si on vérifie qu'un document XML respecte une DTD.
Néanmoins les limites des DTDs sont grandes, puisque le typage des éléments contenus dans les balises est très restreint.
Les DTDs sont également écrites dans un langage différent du XML (car elles dérivent d'une technologie liée au SGML), ce qui peut poser des difficultés de manipulation et de transformations.

5.3.2. Les règles de construction d'une DTD

Déclaration

Une DTD (interne ou externe) doit toujours commencer par un en-tête contenant le nom de l'élément qui est à la racine. Toutes les autres déclarations seront faites à l'intérieur de cet élément.

Exemple (pour une DTD interne) :

```
< !DOCTYPE racine [
...  ]>
```

Dans le cas d'une déclaration externe (la DTD se trouve dans un fichier propre `.dtd`), il conviendra de déclarer la DTD utilisée dans le fichier XML par l'utilisation de la ligne suivante :

```
< !DOCTYPE racine SYSTEM "chemin/fichier.dtd">
```

avec `racine` qui est le nom du premier élément de la DTD, aussi appelé élément racine (*root*).
Dans le cas d'une DTD externe, `chemin` est le chemin d'accès à la DTD et `fichier.dtd` le fichier DTD.

Eléments

Les éléments (correspondant aux balises XML) sont définis comme suit :

```
<!ELEMENT nom type>
```

Où `nom` est le nom de l'élément, et `type` est soit un des types de base disponibles (se reporter au tableau (5.1)) soit la définition de la structure arborescente de l'élément (définition de sous-éléments grâce à l'utilisation d'opérateurs).

Tableau 5.1 : Types disponibles

Type	Description
ANY	L'élément peut contenir tous type de données.
EMPTY	L'élément est vide.
#PCDATA	(Parsed Character Data) Ce type définit une chaîne de caractères qui sera "parsée" (analysée) lors de la validation.

Déclarer un élément comme ayant un contenu #PCDATA nécessite une attention particulière. En effet, le contenu étant examiné lors de la validation, il se pourrait que certains symboles soient mal interprétés.

Exemple :

```
<!ELEMENT contrainte #PCDATA>
```

Instance XML :

```
<contrainte> <5 et >3 </contrainte>
```

Alors qu'on voudrait exprimer deux contraintes en utilisant les symboles "<" et ">", l'élément `contrainte` ayant été défini comme `#PCDATA`, l'intérieur de la balise sera traité comme une balise XML. Pour éviter ceci, il convient d'utiliser des caractères d'échappement, comme le montre le tableau (5.2).

Tableau 5.2 : Caractères d'échappement

Caractères d'échappement	Correspond à
<	<
>	>
&	&
"	"
'	'

Exemple d'élément :

```
<!ELEMENT numéro #PCDATA>
```

On définit un élément `numéro` qui est une chaîne de caractères.

```
<!ELEMENT paire (e1,e2)>
<!ELEMENT e1 #PCDATA>
<!ELEMENT e2 #PCDATA>
```

On définit un élément paire, qui est composée de deux sous-éléments e1 et e2. Ceci est possible grâce à l'utilisation de l'opérateur ",".
Comme e1 et e2 sont également des éléments, il faut les définir de la même manière.

<u>Les opérateurs</u>

Les DTD permettent d'utiliser quelques opérateurs pour préciser la cardinalité des éléments rencontrés. Le tableau (5.3) liste les différents opérateurs.

Tableau 5.3 : Opérateurs

<table>
<tr><th>Opérateur</th><th>Description</th><th>Exemple</th></tr>
<tr><td>+</td><td>L'élément doit être présent au moins une fois.</td><td>A+ : A présent 1-n fois.</td></tr>
<tr><td>*</td><td>L'élément peut être présent un nombre indéfini de fois.</td><td>A : A présent 0-n fois.</td></tr>
<tr><td>?</td><td>L'élément peut être omis ou apparaître une fois.</td><td>A? : A présent 0-1 fois.</td></tr>
<tr><td>|</td><td>Permet de laisser le choix entre plusieurs éléments.</td><td>A|B : A ou B (mais pas les 2).</td></tr>
<tr><td>,</td><td>Permet de définir l'ordre des éléments.</td><td>A,B : A suivit de B.</td></tr>
<tr><td>()</td><td>Permet de regrouper des éléments pour appliquer les autres opérateurs.</td><td>(A,B)+ : (A,B) présent 1-n fois.</td></tr>
</table>

Exemple :

```
<!ELEMENT population (individu)+ >
<!ELEMENT individu (nom, prénom, hobby*, job?, sexe) >
<!ELEMENT sexe (homme | femme) >
<!ELEMENT homme #PCDATA>
<!ELEMENT femme #PCDATA>
```

Ce qui signifie que la population est composée d'individus (au moins 1).
Chaque individu possède un nom un prénom et éventuellement un ou plusieurs hobbys (0-n). Il peut avoir un job (0-1) et appartient à un sexe.
Le sexe est défini comme étant un choix entre homme et femme (qui sont eux aussi des éléments).

Attributs

Les attributs (correspondant aux attributs des balises XML) sont déclarés comme suit :

```
<!ATTLIST
nom-élément nom-attribut  type valeur-par-défaut
nom-élément nom-attribut2 type valeur-par-défaut>
...
```

Où `nom-élément` est l'élément auquel l'attribut `nom-attribut` se rapporte, `type` est un des types présents dans le tableau (5.4).
Il est également possible (c'est facultatif) de définir une valeur par défaut `valeur-par-défaut` à l'attribut.
Le tableau (5.5) montre qu'il est possible d'utiliser certaines options aux attributs.

Tableau 5.4 : Types des attributs

Type	**Description**
CDATA	La valeur est une chaîne de caractères.
(en1|en2|..)	La valeur doit être une de celles de la liste.
ID	La valeur est un identifiant unique.
IDREF	La valeur est l'identifiant d'un autre élément.
IDREFS	La valeur est une liste d'autres IDs.
NMTOKEN	La valeur est un nom XML valide.
NMTOKENS	La valeur est une liste de noms XML valides.
ENTITY	La valeur est une entité.
ENTITIES	La valeur est une liste d'entités.
NOTATION	La valeur est une notation.
xml:	La valeur est une valeur xml prédéfinie.

Remarques :

- Le type ID permet de définir un nom unique pour l'attribut. En effet, un nom qui est de type ID ne peut apparaître qu'une seule fois dans le document comme valeur de ce type.

- Le type IDREF permet de faire référence à un des noms qui sont définis dans ID.

Exemple :

```
<!DOCTYPE FAMILLE [
      <!ELEMENT FAMILLE (PERSONNE)*>
      <!ELEMENT PERSONNE (NOM, EPOUSA)>
      <!ATTLIST PERSONNE ID ID #REQUIRED>
      <!ELEMENT EPOUSE EMPTY>
      <!ATTLIST EPOUSA IDREF IDREF #REQUIRED>
      <!ELEMENT NOM (#PCDATA)> ]>
```

Instance XML :

```
<FAMILLE>
      <PERSONNE ID="p1">
            <NAME>Robert Durant</NAME>
                  <EPOUSA IDREF="p2"/>
      </PERSONNE>
      <PERSONNE ID="p2">
            <NAME>Emilie Durant</NAME>
                  <EPOUSA IDREF="p1"/>
      </PERSONNE>
</FAMILLE>
```

Tableau 5.5 : Options des attributs

Option	Description
Value	La valeur par défaut de l'attribut.
#DEFAULT value	La valeur par défaut de l'attribut.
#REQUIRED	La valeur de l'attribut doit être incluse dans l'élément.
#IMPLIED	La valeur de l'attribut ne doit pas être obligatoirement incluse.
#FIXED value	La valeur de l'attribut est fixée.

Exemple :

```
<!ATTLIST élément1 id1 CDATA #FIXED "1">
```

Instance XML :

`<élément1 id1="1">` : ok
`<élément1 id1="2">` : ko

L'utilisation de `#FIXED "1"` permet de déclarer la valeur de l'attribut id1 comme étant fixée et ayant toujours la valeur 1. Toute valeur autre que 1 sera rejetée.

Exemple :

```
<!ATTLIST
élément1 attr1 CDATA #REQUIRED
élément1 attr2 CDATA #IMPLIED
élément1 attr3 ("valeur1" | "valeur2") "valeur1">
```

Instance XML :

```
<élément1 attr1="val" attr2="val2" attr3="valeur1"> ok
<élément1 attr1="val" attr3="valeur2"> ok
<élément1 attr2="val2" attr3="v"> ko
```

l'attribut `attr3` est défini de la manière suivante :
Les seules valeurs qu'il peut prendre sont `valeur1` ou `valeur2`.
La valeur par défaut sera `valeur1`

Entités

Les entités sont des variables qui sont utilisées comme des raccourcis pour des morceaux de texte souvent utilisés.
Elles peuvent être déclarées en interne comme en externe (du fichier `.dtd`).
Elles sont donc automatiquement substituées lors de leur utilisation dans le document.

5.3.3. Limites des DTDs

Les DTD étant les premiers formalismes permettant une validation, il est évident qu'elles présentent certaines lacunes. Comme on l'a vu, le typage des DTD est très limité. PCDATA et CDATA reprennent toutes les informations des chaînes de caractères, sans qu'on puisse différencier les strings, dates, entiers, ...

De plus, il n'est pas possible de dénombrer précisément les éléments qu'on définit.

Un des problèmes importants est aussi que la validation par une DTD d'un fichier XML peut échouer dans le cas où il contiendrait plusieurs balises portant le même nom et placées à des niveaux d'arborescence différents.

Il va de soi que les DTD suffisent pour définir la syntaxe d'un document XML simple, mais qu'elle sera assez vite trop simpliste, s'il s'agit de documents où l'on doit pouvoir faire la différence entre du texte ou un entier. De plus, le fait que les DTD ne soient pas écrites en XML peut poser certains problèmes (on pense notamment à la transformation XSL qui devrait être totalement repensée, puisque le XSLT n'est pas adapté à cette structure).

Quand les DTD furent créées, elles étaient utilisées dans le cadre de XML "document-oriented" qui prévalait avec le SGML. Il semble que maintenant, les XML Schema, initiative du W3C soient plus adaptés aux besoins actuels, plus centrés sur l'interopérabilité et "data-oriented").

5.4. Les XML Schemas.

5.4.1. Qu'est-ce qu'un XML Schema?

Il existe plusieurs normes XML Schemas. Celle que nous utiliserons dans ce chapitre fait référence au W3C XML Schema. Du fait qu'elle émane du W3C, elle a suivit un long processus de standardisation.
Il en existe néanmoins d'autres, comme Relax NG [Xvif02].
Convention de notation : XML Schema fait référence à la norme W3C XML Schema (définie en [XmlS02])

Le but des XML Schemas est exactement le même que celui des DTD. La différence se situe dans la précision et la complétude des XML Schemas.
Un Xml Schema permet de définir:

- Les éléments qui peuvent apparaître dans le document.
- Les attributs qui peuvent apparaître dans le document.
- La structure arborescente du document.
- L'ordonnancement au sein du document.
- Le nombre de sous-éléments.
- Le type de contenu des éléments et attributs.

En quoi les XML Schemas présentent-ils des avantages sur les DTD? Les XML Schemas :

- Sont écrits directement en XML.
- Supportent les types de données.
- Permettent de dériver de nouveaux types de ceux déjà existants (par restriction ou extension)
- Supportent les "namespaces".
- Permettent un dénombrement précis des éléments.

5.4.2. Les règles de construction d'un XML Schema

Déclaration

Tout XML Schema est compris dans l'élément suivant (appelé élément racine) :

```
<xsd:schema >
...
</xsd:schema>
```

Cette balise peut contenir des attributs

Exemple :

```
<xsd:schema xmlns:xsd="http://www/w3.org/2001/XMLSchema">
```

Cet attribut, obligatoire, sert à indiquer que les éléments et types de données utilisés dans le XML Schema viennent du "namespace" http://www/w3.org/2001/XMLSchema Cela indique aussi que ces mêmes éléments doivent être préfixés avec "xsd".

L'attribut `targetNamespace` est utilisé pour définir le XML Schema correspondant à ce XML Schema. Et c'est ce XML Schema qui sera utilisé pour valider des XML Schema contenant des éléments ou attributs de ce namespace.

Exemple :

```
<xsd:schema xmlns:xsd="http://www/w3.org/2001/XMLSchema"
targetNamespace="http://www.exemple.com">
```

Définit le XML Schema qui serait utilisé pour valider des documents tels que

```
<xsd:schema xmlns:xsexemple="http://www.exemple.com">
```

utilisant le namespace " http://www.exemple.com" et dont les éléments doivent être préfixés par `xsexemple`.

La référence du fichier XML Schema (`.xsd`) dans un fichier XML (`.xml`) se fait via la ligne suivante :

```
<Elément-racine xmlns:xsi=http://www.w3.org/2001/XMLSchema-
instance xsi:noNamespaceSchemaLocation="chemin d'accès au
schema">
```

Le préfixe `xsi` étant utilisé pour faire référence au namespace "`XMLSchema-instance`", nécessaire à la définition de l'attribut "`noNamespaceSchemaLocation`", qui est un mot clé réservé par le W3C et reconnu par l'ensemble des parseurs et validateurs W3C XML Schema compatibles.

Eléments

Les XML Schemas permettent de définir deux sortes d'éléments. Il y a les éléments simples et les éléments complexes.

a. Les éléments simples (simple element)

Ce sont ceux qui ne contiennent que du texte, ils ne contiennent donc pas de sous-éléments ou d'attributs.

Exemple :

```
<xsd:element name="nom" type="type">
```

Avec `type` qui est un des types de données permis par le langage W3C XML Schema. Les types les plus communs sont : `xsd:string, xsd:decimal, xsd:integer, xsd:boolean, xsd:date, xsd:time` (pour plus d'informations sur le typage, se référer au paragraphe qui lui est réservé).
Il est possible de définir des valeurs fixes ou par défaut pour les éléments simples en utilisant les attributs "`default`" et "`fixed`"

Exemple :

```
<xsd:element name="compteur" type="xsd:integer" default="0">
```

Ce qui permet de définir un élément XML, dont le nom est compteur, qui aura comme valeur par défaut 0 dans le document instance.

b. Les éléments complexes (complex element)

Ce sont ceux qui contiennent des sous-éléments et/ou des attributs.

Exemple :

```
<xsd:element name="personne">
  <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="nom" type="xsd:string"/>
        <xsd:element name="prénom" type="xsd:string"/>
        <xsd:element name="age" type="xsd:integer"/>
      </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

Instance XML :

```
<personne>
 <nom>Robert</nom>
 <prénom>Dupont</prénom>
 <age>40</age>
</personne>
```

Remarque: On voit très bien que l'élément personne a une structure arborescente, par opposition aux éléments simples, qui ne contiennent pas de sous-éléments. On remarque également, au sein de la structure arborescente de l'élément personne, l'existence d'une structure « complexType » (voir Typage).

Attributs

Les attributs sont déclarés de la même manière que les éléments simples. Tous les éléments contenant des attributs sont de type complexe.

```
<xsd:attribute name="id" type="xsd:integer/>"
```

On peut également déclarer des valeurs par défaut, comme dans le cas des éléments simples (utilisation de l'attribut fixed et default).
Mais il est aussi possible de spécifier si un attribut est optionnel ou obligatoire, grâce à l'utilisation de l'attribut `use` :

```
<xsd:element name="personne">
  <xsd:complexType>
   <xsd:attribute name="id" type="xsd:integer"
   use="optional/required">/>
  </xsd:complexType>
</xsd:element>
```

Typage

a. Les types simples.

Dans la norme XML Schema, le W3C met à la disposition de l'utilisateur plus de 40 types de base. Ce qui représente un ensemble assez complet de types de données.

Les plus courants sont les types de base comme : `string, integer, date, CDATA, token, byte,...` (pour les détails concernant ces types, se référer à la norme XML Schema disponible sur [XmlS02]).
L'utilisation de ces types est assez simple, et aucune déclaration préalable n'est nécessaire (sauf, néanmoins, déclarer le namespace W3C pour les types).

Reprenons le même exemple que pour le paragraphe concernant les éléments simples :

```
<xsd:element name="compteur" type="xsd:integer">
```

Le type de base utilisé est le type `xsd:integer`, ce qui permet d'affirmer que la valeur de l'élément compteur doit être un entier.

b. Les types complexes

Il se peut que les types simples proposés ne suffisent pas à pouvoir définir tout ce que l'on voudrait. L'introduction des types complexes comble cette lacune, puisqu'ils permettent à l'utilisateur de définir son propre type.

Reprenons l'exemple développé pour expliquer les éléments complexes :

```
<xsd:element name="personne">
 <xsd:complexType>
      <xsd:sequence>
           <xsd:element name="nom" type="xsd:string"/>
           <xsd:element name="prénom" type="xsd:string"/>
           <xsd:element name="age" type="xsd:integer"/>
      </xsd:sequence>
 </xsd:complexType>
</xsd:element>
```

On aurait pu déclarer un type complexe "`tpersonne`" hors de l'élément personne.

```
<xsd:element name="personne" type="tpersonne"/>

<xsd:complexType name="tpersonne">
 <xsd:sequence>
      <xsd:element name="nom" type="xsd:string"/>
      <xsd:element name="prénom" type="xsd:string"/>
      <xsd:element name="age" type="xsd:integer"/>
 </xsd:sequence>
</xsd:complexType>
```

Le résultat est exactement le même, au détail près que le type "`tpersonne`" pourra être réutilisé par la suite pour d'autres éléments.

Tout comme dans les DTD (grâce aux opérateurs) il est possible de définir l'ordre des éléments et le choix entre plusieurs éléments :

```
<xsd:sequence>
[sous-éléments]
</xsd:sequence>
```

Cet élément permet de définir l'ordre des sous-éléments.

```
<xsd:choice>
 <xsd:element name="professeur" type="xsd:string" />
 <xsd:element name="étudiant" type="xsd:string"/>
</xsd:choice>
```

Cet élément permet définir un choix entre deux possibilités.

Substitution

Le mécanisme de substitution est comparable à l'élément `choice` que nous avons vu.

```
<xsd:element name="name" type="xsd:string"/>
<xsd:element name="nom" substitutionGroup="name"/>
```

Ceci permet aux instances suivantes d'être valides :

```
<name>John Smith</name>
```

et

```
<nom>John Smith</nom>
```

Un XML Schema utilisant pleinement les substitutions pourrait donc être déclaré de façon à supporter plusieurs langues.

Dérivation

Il peut arriver que l'on veuille modifier un type déjà existant dans le document (sous forme de type simple ou complexe) pour pouvoir le réutiliser sans devoir définir un type propre.
C'est possible grâce au mécanisme de dérivation (par restriction ou par extension).

La dérivation par restriction, permet de modifier légèrement le type utilisé en lui ajoutant des contraintes. Ainsi, on ne fait que restreindre l'ensemble de valeurs possibles de ce type. Reprenons l'exemple des types simples :

```
<xsd:element name="compteur" type="xsd:integer">
```

On pourrait décider que les valeurs que peut prendre l'élément compteur soient limitées. Cela peut être fait par l'utilisation d'une dérivation par restriction. :

```
<xsd:element name="compteur">
   <xsd:simpleType>
     <xsd:restriction base="xsd:integer">
       <xsd:minInclusive value="0"/>
       <xsd:maxInclusive value="10"/>
     </xsd:restriction>
   </xsd:simpleType>
</xsd:element>
```

La dérivation par restriction permet ici de borner inférieurement et supérieurement les valeurs possibles de l'élément compteur. Ici la valeur de l'élément compteur sera comprise entre 0 et 10 (non strictement).

Il existe beaucoup d'autres possibilités de dérivation par restriction, qui permettent de définir le contenu attendu d'un élément très précisément. Ces possibilités sont appelées "facets" et sont variables suivant le type de base subissant une dérivation par restriction.
Il est notamment possible de définir la longueur de la balise; la liste complète des facets du type `xsd:integer` se trouve dans le tableau (5.6).

Tableau 5.6 : Facets pour le type de base xsd:integer

Contrainte	Description
Enumeration	Définit la liste des valeurs acceptables
FractionDigits	Spécifie le nombre maximum de chiffres décimaux permis. (doit être plus grand ou égal à zéro)
Length	Spécifie le nombre exact de caractères, ou liste d'objets permis. (doit être plus grand ou égal à zéro)
MaxExclusive	Spécifie la limite supérieure des valeurs numériques. (la valeur doit être inférieure à cette valeur)
MaxInclusive	Spécifie la limite supérieure des valeurs numériques. (la valeur doit être inférieure ou égale à cette valeur)
MaxLength	Spécifie le nombre maximum de caractères, ou liste d'objets permis. (doit être plus grand ou égal à zéro)
MinExclusive	Spécifie la limite inférieure des valeurs numériques. (la valeur doit être supérieure à cette valeur)
MinInclusive	Spécifie la limite inférieure des valeurs numériques. (la valeur doit être supérieure ou égale à cette valeur)
MinLength	Spécifie le nombre maximum de caractères, ou liste d'objets permis. (doit être plus grand ou égal à zéro)
Pattern	Définit la séquence exacte des caractères qui sont acceptables.
TotalDigits	Spécifie le nombre exact de chiffres permis. (doit être plus grand ou égal à zéro)
WhiteSpace	Spécifie comment les espaces (line feeds, tabs, spaces, and carriage returns) sont manipulés.

(Source : [Fac02])

Remarque : Il est intéressant de noter que la contrainte Pattern permet de définir au caractère près (par l'utilisation d'expressions régulières) la valeur que peut prendre le type ainsi modifié.

On peut également étendre un type déjà déclaré pour qu'il réponde à nos besoins. Cela est possible grâce à la balise xsd:extension. Cette balise permet de dériver par extension un type déjà existant. Ce principe peut être assimilé à l'héritage du paradigme orienté objet.

Exemple :

```
<xsd:element name="étudiant">
  <xsd:complexContent>
      <xsd:extension base="tpersonne">
       <xsd:sequence>
        <xsd:element name="faculté" type="xsd:string"/>
        <xsd:element name="carte étudiant"
        type="xsd:integer"/>
       </xsd:sequence>
     </xsd:extension>
  </xsd:complexContent>
</xsd:element>
```

Ceci permet d'étendre le type "`tpersonne`" de façon à pouvoir l'exploiter pour des étudiants. Cette extension consiste à rajouter au type "`tpersonne`" deux éléments indiquant la faculté et le numéro de carte d'étudiant de l'individu.

5.4.3. Limites des XML Schemas.

Comme on l'a vu, les XML Schemas complètent les DTD en de nombreux points. Cela vient sûrement du fait que les XML Schemas ont été conçus bien après les DTD et qu'ils se devaient d'être plus complets.
Les XML Schemas sont résolument mieux adaptés aux besoins actuels et sont plus centrés sur l'interopérabilité. Cela est visible grâce au typage très complet, la possibilité de dériver les types, la précision de dénombrement des éléments, etc.

Mais là où quelques lignes suffisaient à définir la structure d'un document avec une DTD, on est amené à rédiger près d'une page avec les XML Schemas.
On a donc gagné en précision pour perdre en concision.

5.5. Schematron

5.5.1. Qu'est-ce que Schematron

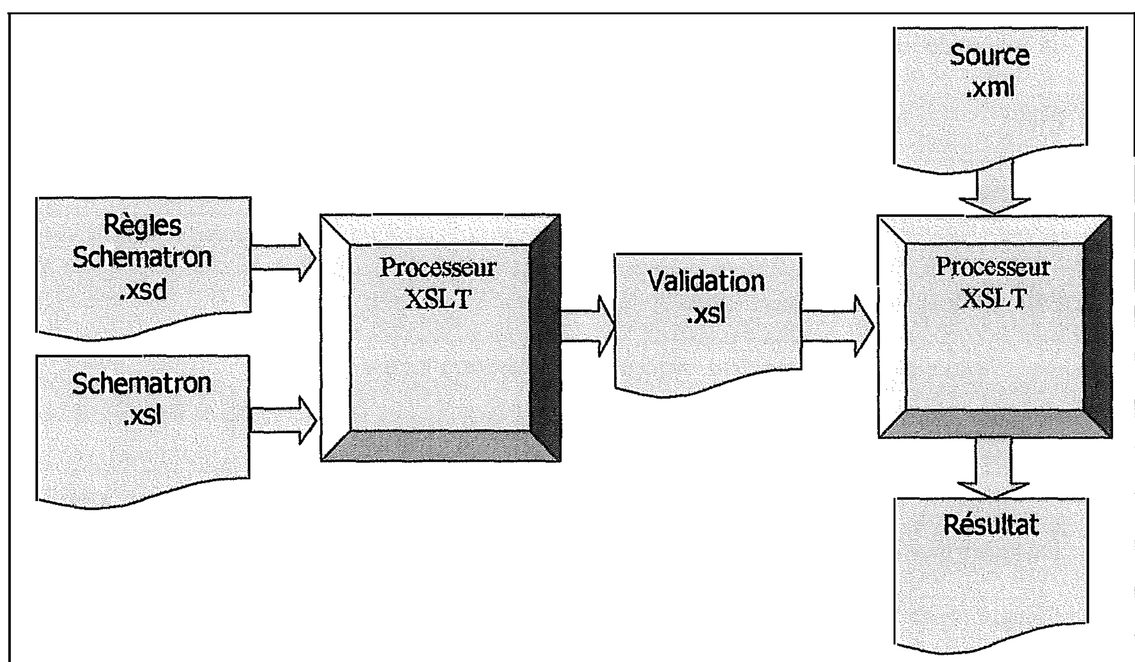
Schematron est un langage de schémas structurels (Structural Schema Language), simple et puissant. C'est un projet de Rick Jelliffe, Academia Sinica Computing Centre, Taibei. Ce projet est opensource et totalement gratuit. ([Sch02])

Ce langage est indépendant de la grammaire du fichier XML (ensemble des balises du fichier) car basé sur XPath.

Schematron permet de faire une validation basée sur les schemas. Il associe un langage de prédicats booléens avec des fonctions XPath.

5.5.2. Fonctionnement

Figure 5.1. Fonctionnement de Schematron



Le fonctionnement, comme illustré avec la figure (5.1), est assez simple. Dans un premier temps, on définit un schema contenant les règles. Ces règles sont produites grâce à l'utilisation de six éléments de base tirés de [Sch02] (détaillés au point 5.5.3).

On associe ensuite une feuille de style que l'on peut trouver sur le site de Schematron, qui va déterminer le type de sortie attendu (txt, XML, HTML,... Il y en a cinq disponibles.). Cette feuille de style est aussi appelée meta-stylesheet, car c'est une feuille de style qui en produit une autre.

La feuille de style, ainsi associée au XML Schema contenant les règles via un processeur XSLT, va produire une feuille de style contenant les règles.

Celle-ci, appliquée à n'importe quel document, permettra de vérifier les règles. Les résultats de cette vérification seront contenus dans le fichier en sortie (nommé ici Résultat, et dont le type dépend de la feuille de style initialement choisie).

5.5.3. Construction du XML Schema de règles

Le XML Schema utilisé par Schematron étant un XML Schema basique, il sera déclaré de la façon habituelle (voir le point 5.4 traitant des XML Schema). Seul le namespace de référence changera.

```
<sch:schema xmlns:sch="http://www.ascc.net/xml/schematron" >
```

Ce qui permet aux balises que l'on utilisera pour déclarer les règles de ne pas être rejetées. Tous les éléments déclarés dans le but d'être utilisés pour la validation de Schematron devront donc être préfixés de « `sch` ».

Il y a six éléments de base qui peuvent être utilisés pour construire les règles.
Pour notre exemple, tous ces éléments doivent être préfixés par « `sch` », mais pour plus de clarté, nous omettrons le préfixe.

<u>title</u>

Cet élément sert à donner un titre au document de règles que l'on construit. Cet élément est optionnel.

Exemple :

```
<title>Schéma de validation</title>
```

<u><ns prefix="?" uri="?"></u>

Cet élément permet de donner les namespaces et préfixes utilisés pour le XPath. Il peut y avoir zéro ou plusieurs éléments de ce genre.

<u>assert</u>

```
<sch:assert test="x">y</sch:assert>
```

Cet élément permet de décrire le résultat observé dans le cas où le test (expression XPath) effectué est négatif. C'est l'équivalent au CHOOSE du XSLT (avec résultat négatif) :

```
<xsl:choose>
   <xsl:when test="x"/>
   <xsl:otherwise>y</xsl:otherwise>
</xsl:choose>
```

Exemple :

```
<sch:assert test="count(roue)=4">
   Ce véhicule n'a pas 4 roues
</sch:assert>
```

<u>report</u>

```
<sch:report test="x">y</sch:report>
```

Cet élément permet de décrire le résultat observé dans le cas où le test (expression XPath) effectué est positif. C'est l'équivalent au IF du XSLT :

```
<xsl:if test="x">y</xsl:if>
```

Exemple :

```
<sch:report test="count(toit)=1">
   Ce véhicule a un toit, ce n'est pas un cabriolet
</sch:report>
```

<u>rule</u>

```
<sch:rule context="x" id="z">y</sch:rule>
```

Cet élément permet de définir des ensembles de contraintes en regroupant plusieurs éléments de type `assert` et/ou `report`. L'attribut `context` permet de définir un contexte commun à toutes les règles à appliquer. Il contient une expression XPath. C'est l'équivalent du TEMPLATE du XSLT :

```
<xsl:template match="x">y</xsl:template>
   <sch:rule context="véhicule" id=""verifvoiture>
         <sch:assert test="count(roue)=4">
         Ce véhicule n'a pas 4 roues
         </sch:assert>
         <sch:report test="count(toit)=1">
         Ce véhicule a un toit, ce n' est pas un cabriolet
         </sch:report>
   </sch:rule>
</xsl:template>
```

<u>pattern</u>

```
<sch:pattern name="x">y</sch:pattern>
```

Cet élément permet de grouper plusieurs règles, de façon à les regrouper si elles sont en rapport avec le même sujet. C'est l'équivalent du `<xsl:apply-templates>` de XSLT .

```
<xsl:apply-templates select="/"/>
```

Exemple :

```
<sch:pattern name="Contraintes des voitures"
   <sch:rule context="...">
     ...
   </sch:rule>
</sch:pattern>

<sch:pattern name="Autres contraintes"
   <!—autres règles -->
</sch:pattern>
```

Une fois l'ensemble des règles défini et rassemblé en pattern, on dispose du schema final qui pourra être utilisé pour la validation.

Il existe quelques autres éléments, mais ceux-ci ne sont pas aussi importants pour la rédaction de règles. Le lecteur pourra trouver plus d'informations à ce sujet en consultant [Sch02]

5.5.4. Choix de la meta-stylesheet

Une fois que nous disposons du schema de règles définitif, il nous reste à choisir la meta-stylesheet qui nous convient, car c'est elle qui déterminera le format de la sortie. Nous allons donc en détailler quelques-unes.

Schematron-basics génère une stylesheet de validation qui renvoie simplement le texte résultat de la validation (texte contenu dans les éléments `assert` et `report`). C'est bien évidemment la plus simple des stylesheet Schematron.

Schematron-message génère une stylesheet de validation qui peut être utilisée avec un processeur XSLT qui peut manipuler les éléments `xml:message` de façon à les envoyer en sortie standard. Cette stylesheet est principalement utilisée en coordination avec des éditeurs interactifs comme Emacs, XED pour valider une instance XML qui serait éditée.

Schematron-report et *schematron-pretty* stylesheets génèrent des stylesheets de validation qui produisent des messages en format HTML.

Schematron-xml génère une stylesheet de validation qui produit des messages en XML. Les éléments résultats ont un attribut `location` contenant les expressions XPath qui ont servi à leur évaluation. Cette stylesheet Schematron permet une meilleure interaction avec les applications utilisant la logique XML.

5.5.5. Limites de Schematron

Malgré les possibilités qu'offrent Schematron, les règles que l'on peut construire sont uniquement composées d'expressions booléennes du type :

Si A, alors B ou Si non A, alors C

Ce qui peut s'avérer insuffisant pour exprimer des contraintes plus compliquées, comme des expressions du premier ordre.

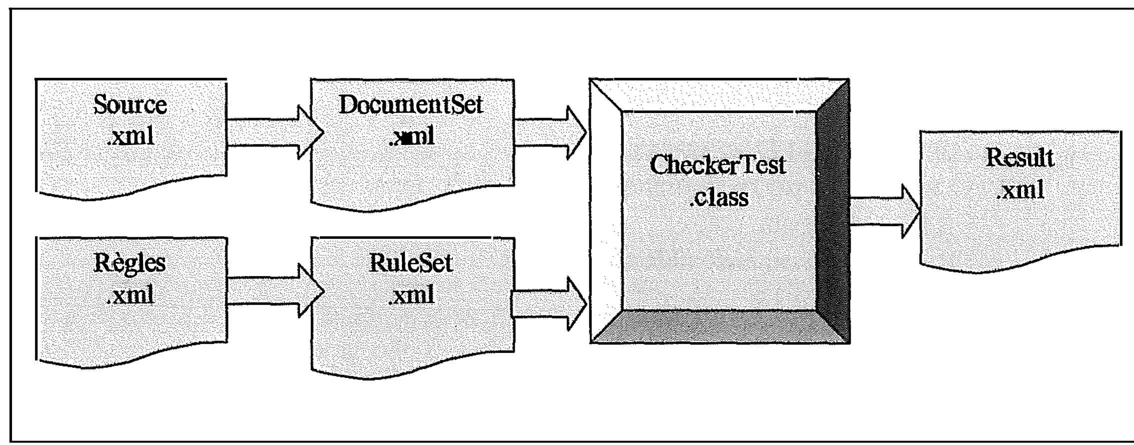
5.6. Xlinkit

5.6.1. Qu'est-ce que Xlinkit?

Xlinkit est une application issue d'un projet opensource mené par l'université de Londres (University College London). ([Xli02])
Ce projet a pour but la validation de fichiers XML à partir de règles définies par l'utilisateur dans un autre fichier XML. La spécificité de Xlinkit est que la validation qu'il permet est possible tant pour un seul fichier qu'entre plusieurs fichiers (c'est donc une validation multi-documents).
Les règles que l'on peut définir sont construites sur une logique de prédicats du 1er ordre en alliant la syntaxe des règles fournies par Xlinkit et l'utilisation d'expressions XPath.

5.6.2. Fonctionnement de Xlinkit

Figure 5.2. Fonctionnement de Xlinkit



Le fonctionnement de la validation effectuée par Xlinkit est assez simple (il est illustré par la figure (5.2)).

Dans un premier temps, on dispose du fichier source à valider (`source.xml`). Pour que le validateur `CheckerTest.java` puisse déterminer où se trouve ce fichier source, on le référence dans le fichier `DocumentSet.xml`, ainsi que tous les autres fichiers qui prendront part à la validation (on peut en effet faire une validation entre plusieurs fichiers).

Une fois le fichier `DocumentSet` établi, on peut passer à l'élaboration du fichier de règles. Le fichier `RuleSet.xml` est du même type que le fichier `DocumentSet.xml`, il ne contient pas directement les règles à appliquer, mais des références à des fichiers contenant les règles.

Après avoir défini ces deux fichiers, la validation peut alors réellement avoir lieu.
Il suffit d'exécuter le module `CheckerTest.java`. Il prend les fichiers sources, leur applique les règles et produit un fichier résultat.

Ce fichier contient toutes les informations relatives aux éléments ayant ou n'ayant pas respecté les règles qu'on avait définies.

Nous allons maintenant voir exactement le contenu détaillé des fichiers utilisés (un exemple complet de validation multi-documents se trouve en annexe A.2.)

Le fichier DocumentSet.xml

Ce fichier contient la localisation des fichiers de ressources dont le module `CheckerTest.java` pourrait avoir besoin. Cela comprend le(s) fichier(s) source(s), au(x)quel(s) on doit appliquer les règles, mais aussi les fichiers qui pourraient intervenir lors de la validation (dans le cas d'une validation multi-documents).

Figure 5.3. Exemple général de DocumentSet.xml

```
<?xml version="1.0" standalone="no"?>
   <!DOCTYPE DocumentSet SYSTEM "DTD/DocumentSet.dtd">
   <DocumentSet name="source">                        (1)
    <Description>Sources</Description>                (2)
    <DocFile href="source.xml"/>                      (3)
    <Set href="autres_sources.xml"/>                  (4)
   </DocumentSet>
```

(1) Ouverture du DocumentSet.
(2) Description du DocumentSet (facultatif).
(3) Référence à une source.
(4) Référence à un ensemble de sources.

La différence entre `<DocFile href="file.xml">` et `<Set href="file.xml">` est que dans le premier cas, `file.xml` contient directement les informations, alors que dans le second cas, il ne contient pas d'informations, mais un ensemble de références à d'autres fichiers.

Le fichier RuleSet.xml

Ce fichier contient la localisation des fichiers de règles qui seront appliquées lors de la validation effectuée par le module `CheckerTest.java`.

Figure 5.4. Exemple général de RuleSet.xml

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE RuleSet SYSTEM "../../DTD/RuleSet.dtd">
<RuleSet name="règles">                               (1)
 <Description>Règles à appliquer</Description>        (2)
 <RuleFile href="rule1.xml"/>                         (3)
 <RuleFile href="rule2.xml"/>                         (4)
</RuleSet>
```

(1) Ouverture du RuleSet.
(2) Description du RuleSet (facultatif).
(3) Référence à un fichier de règles (.xml).
(4) Référence à un fichier de règles (.xml).

Pourquoi y a-t-il plusieurs déclarations de fichiers de règles ? On peut en effet mettre toutes les règles dans un seul fichier. Mais l'utilité peut se faire sentir si on désire faire différentes validations utilisant des règles communes. Dans ce cas, il pourrait être plus facile de faire un fichier contenant les règles communes, et un autre contenant les règles spécifiques pour éviter les redondances.

Le fichier rule1.xml (idem pour rule2.xml si on définit plusieurs fichiers de règles)

Ce fichier contient les règles que l'on souhaite appliquer. Chaque règle est identifiée par un attribut « id » qui permettra par la suite de pouvoir analyser les résultats.

Figure 5.5. Exemple général de rule.xml

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

<!DOCTYPE consistencyruleset SYSTEM
'../../DTD/consistencyrule.dtd'>

<consistencyruleset>                                    (1)
  <globalset id="ensemble1" xpath="/ens1/chemin"/>      (2)
  <globalset id="ensemble2" xpath="/ens2/chemin"/>
  <consistencyrule id="r1">                             (3)
     <description>                                      (4)
     Les codes de l'ensemble2 doivent êtresupérieurs
     à ceux de l'ensemble1.
     </description>

     <forall var="a" in="$ensemble2/code[text()]">      (5)
        <not>
          <exists var="b" in="$ensemble1/code[$a/text()
          &lt;= text()]"/>                              (6)
        </not>

     </forall>

  </consistencyrule>

</consistencyruleset>
```

(1) Ouverture du consistencyruleset.
(2) Déclaration d'un globalset.
(3) Ouverture d'une consistencyrule.
(4) Description d'une consistencyrule (facultatif).
(5) Forall (obligatoire) à un fichier de règles.
(6) Expressions qui permettent de définir la règle souhaitée.

Le <globalset> permet de sélectionner des parties d'arborescence des fichiers qui nous intéressent (sortes d'ensembles de variables). Ici, on sélectionne dans la 'variable' ensemble1 tous les fils qui correspondent au chemin XPath /ens1/chemin.
Chaque règle doit avoir un forall (du type: pour tout x de mon ensemble a, vérifier une propriété).

La syntaxe des règles se trouve dans la DTD associée (ici `consistencyrule.dtd`).

A l'intérieur des balises, on fait appel à des expressions XPath pour définir des contraintes supplémentaires.

XPath est nécessaire car il permet de définir des contraintes non modélisables uniquement avec les balises disponibles, grâce par exemple, à l'utilisation de fonctions.

Le fichier Result.xml

Ce fichier contient le résultat de la validation. Il peut être mis en forme ensuite grâce à une stylesheet qui permettra de visualiser le résultat.

Figure 5.6. Exemple général de fichier résultat

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE xlinkit:LinkBase SYSTEM
"http://www.xlinkit.com/DTD/LinkBase.dtd>
<?xml-stylesheet type="text/xsl" href="linkbase.xsl"?>

<xlinkit:LinkBase date="Fri Oct 19 13:39:30 CEST 2001"     (1)
      docSet="path_to/DocumentSet.xml
      ruleSet="path_to/RuleSet.xml«
      xmlns:xlink=http://www.w3.org/1999/xlink
      xmns:xlinkit="http://www.xlinkit.com">

<xlinkit:ConsistencyLink
      ruleid="path_to\rule.xml#/consistencyruleset
      /consistencyrule[n]">                                   (2)

         <xlinkit:State>consistent</xlinkit:State>            (3)
         <xlinkit:Locator                                     (4)
            xlink:href="path_to\result.xml #/chemin_XPath1"
            xlink:label="" xlink:title=""/>

         <xlinkit:Locator
            xlink:href="path_to\result.xml #/chemin_XPath2"
            xlink:label="" xlink:title=""/>

     </xlinkit:ConsistencyLink>
</xlinkit:LinkBase>
```

(1) path_to/DocumentSet.xml et path_to/RuleSet.xml sont les chemins des DocumentSet et RuleSet qui ont été utilisés pour obtenir le fichier résultat. On déclare également les namespaces utilisés.
(2) Cette ligne permet de savoir quelle règle a été utilisée pour le résultat obtenu au point (7), elle permet aussi de savoir dans quel fichier se trouvait cette règle (le fichier RuleSet peut contenir des références à d'autres fichiers de règles.
(3) Résultat de la validation : *consistent/inconsistent* suivant la réussite/l'échec de la validation.
(4) Localisation de l'élément/des éléments ayant respecté/violé la règle.

5.7. Tableau récapitulatif.

Validation	*Technologie*	*Multi-documents*	*Type de validation*
DTD	DTD	Non	Porte sur la structure Physique du fichier XML (grammaire du fichier XML et place des balises et attributs) Assez peu sur le contenu des balises (peu de typage et d'options de validation à ce niveau)
XML Schema	XML	Non	Porte sur la structure Physique du fichier XML (grammaire du fichier XML, place des balises et attributs et cardinalité avancée) Typage assez avancé, mais toujours peu d'options de validation du contenu.
Schematron	XML Schema XSLT	Non*	Porte sur le contenu des balises essentiellement. La validation est effectuée en considérant les règles définies. Ces règles sont sous forme d'expressions booléennes.
Xlinkit	XML XPath Java	Oui	Porte sur le contenu des balises essentiellement. La validation est effectuée en considérant les règles définies. Ces règles sont sous forme d'expressions du 1er ordre.

Les plus grandes différences entre Xlinkit et Schematron sont d'une part, la possibilité d'effectuer une validation multi-documents avec Xlinkit et pas directement avec Schematron, d'autre part la différence d'expression des règles entre les deux langages, puisque Xlinkit permet de définir des règles qui suivent la logique des expressions du 1er ordre.

On ne peut donc pas dire que l'un est meilleur que l'autre, on peut tout simplement dire qu'ils sont mieux appropriés à des problèmes différents. Ainsi si l'on souhaite simplement appliquer des règles à un fichier, alors Schematron suffit, si par contre ces règles portent sur un autre fichier également, alors Xlinkit sera choisi.

* Schematron n'est pas conçu pour effectuer une validation multi-documents. Néanmoins, on peut remarquer que dans la syntaxe XPath, il existe une fonction : document() permettant de faire référence à un autre fichier.

Remarque générale :
Il se peut bien évidemment que d'autres méthodes de validation existent, mais nous n'avons eu l'occasion d'approcher que les quatre qui sont reprises dans le tableau ci-dessus.

5.8. Perspectives d'avenir

L'utilisation de validateurs au niveau sémantique ouvre de nouvelles portes derrière lesquelles on peut imaginer de nouvelles améliorations.

Au niveau international, on remarque une diversification de langages de validation. En effet, on trouve de plus en plus de langages de validation alternatifs aux W3C XML Schemas (comme Relax NG et TREX,... [Xvif02]).

Des efforts sont également faits au niveau des langages alliant validation et transformation (micro-pipes) avec des langages tels que XVIF (*XML Validation Interoperability Framework*).

6. Etude de cas: Intégration de la validation dans le marché des stages.

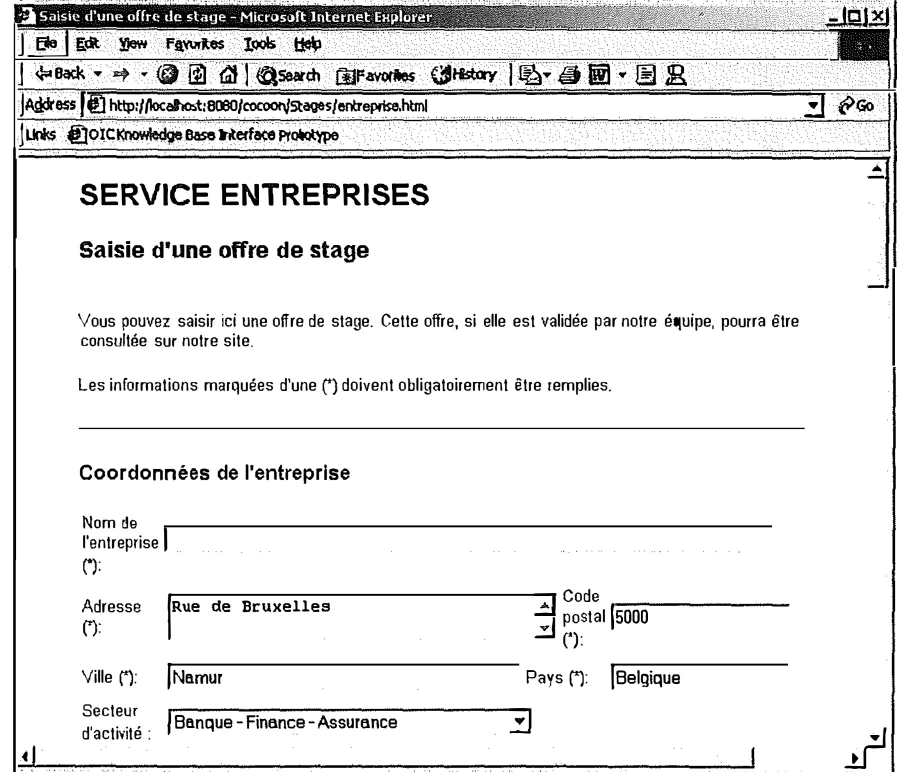
6.1. Présentation de l'étude

Le marché des stages est un service qui a pour but de faciliter l'échange d'informations entre les entreprises qui proposent des stages, les étudiants qui en cherchent et enfin les établissements d'enseignement supérieur qui envoient le profil type de leurs candidats stagiaires.

Les entreprises peuvent proposer des stages en remplissant simplement un formulaire qui est hébergé sur le site du CRP-HT. De cette façon, elles peuvent mettre leurs propositions à la disposition des étudiants.
Les étudiants peuvent faire de même en remplissant un formulaire de demande de stage, assez similaire à celui proposé aux entreprises.

Les formulaires proposés pour la saisie des informations sont très simples (format HTML, voir figure (6.1)) et permettent de récupérer les informations, mais ne vérifient pas le contenu de ces informations. La seule vérification déjà présente consiste à examiner si les champs munis d'une étoile (obligatoires) ont bien été remplis (via un script pearl).
Le résultat de cette saisie est ensuite acheminé vers un responsable via courrier électronique de façon à ce qu'il puisse en vérifier le contenu.
Une fois le contenu accepté, les informations sont ajoutées à un fichier XML contenant toutes les données récoltées.

Figure 6.1. Exemple de formulaire pour une entreprise



6.2. Enjeux

Les possibilités d'améliorations du service étaient vastes.
On peut notamment citer :

- Utilisation d'une base de données à la place d'un fichier XML.
- Ajout de critères de recherche côté publication.
- Ajout de fonctionnalités de validation plus poussées côté saisie.

C'est sur ce dernier point que s'est orientée une partie du travail durant le stage.

Une fois l'objectif choisi, il fallait déterminer quelles technologies seraient employées pour effectuer ce travail.

Comme on l'a vu, il existe plusieurs outils de validation de fichiers XML. Et ces outils peuvent être séparés en deux niveaux :

- Validation syntaxique et structurelle (DTD et XML Schema).
- Validation sémantique (Schematron et Xlinkit).

Le choix s'est porté pour une validation sémantique. En effet, ce type de validation permettrait d'automatiser davantage le processus de saisie (qui jusqu'à présent nécessitait encore l'accord d'un responsable).

Xlinkit fut enfin choisi car il permettait de définir des règles basées sur une logique de prédicats du premier ordre. De ce fait, certaines règles n'auraient pu être exprimées via Schematron (puisque celui-ci se limite aux expressions de type booléennes). De plus, Xlinkit permettait une validation entre plusieurs documents. .

6.3. Architecture utilisée

6.3.1. Le choix des technologies

- On dispose d'un formulaire HTML classique permettant de saisir les données entrées par les utilisateurs. Celles-ci sont une première fois analysées par du JSP (plus d'informations sur le JSP sont disponibles en [Jsp02]) qui se charge de vérifier si les champs obligatoires sont remplis ou non.
- Le choix du JSP est justifié par le fait que Xlinkit avait besoin d'appeler un module Java, ce qui permettait de rester dans la même technologie lors des 2 validations.
- Le choix du JSP est aussi justifié par le fait que la plupart des applications orientées XML que nous avons utilisées (Cocoon et ici Xlinkit) sont écrites en Java et utilisent le même support que JSP. Nous utilisons donc JSP car nous

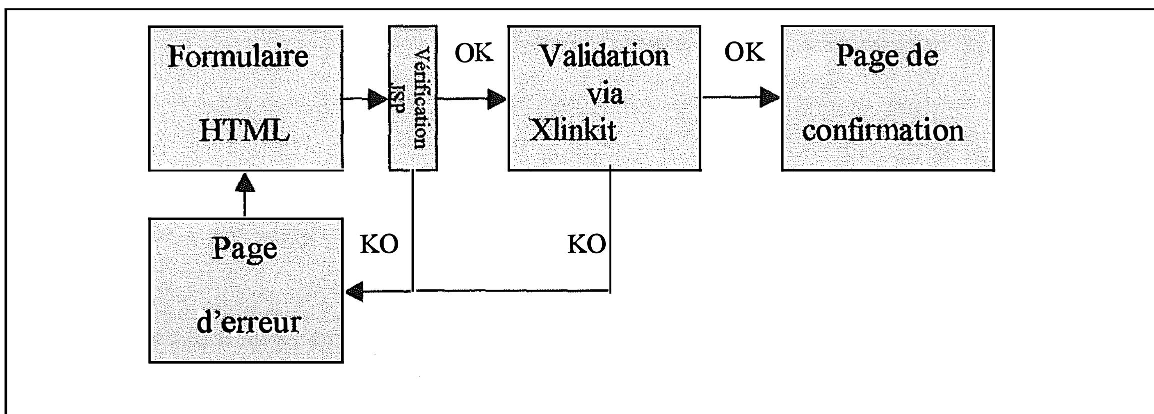
avons toutes les infrastructures nécessaires pour son fonctionnement à notre disposition.

La validation JSP aurait pu être effectuée par Xlinkit, mais pour des soucis d'efficacité, il était préférable d'utiliser JSP pour savoir si les champs étaient correctement remplis. Ensuite, Xlinkit se charge de vérifier si ces données correspondent bien aux règles qu'on avait définies.

6.3.2. Architecture générale

La figure (6.2) reprend l'architecture utilisée, telle que décrite dans les points précédents.

Figure 6.2. L'architecture utilisée



6.4. La récupération des champs via JSP

Le JSP se charge de récupérer les informations que l'utilisateur a fournies via le formulaire HTML et de les transmettre au bon format à Xlinkit.

Dans un premier temps, après avoir récupéré toutes les valeurs, le JSP vérifie si celles qui sont obligatoires ont bien été remplies et génère, dans le cas contraire, une page d'erreur en indiquant la cause à l'utilisateur.

Ensuite, après s'être assuré que les valeurs sont valides, le JSP transforme celles-ci en fichier XML portant le nom de l'entreprise. Ce fichier est nécessaire à la validation via Xlinkit.

Un autre fichier est également nécessaire à Xlinkit : le fichier `DocumentSet.xml`.
Il ne peut pas être construit à l'avance, car il contient la référence au fichier de données dont le nom est aléatoire (nom de l'entreprise).
Le JSP se construit donc également ce fichier.

6.5. La validation par Xlinkit

Nous allons maintenant examiner exactement le contenu détaillé des fichiers utilisés.

Le fichier DocumentSet.xml

Ce fichier (figure (6.6)) contient la localisation des fichiers de ressources dont le module `CheckerTest.java` pourrait avoir besoin. Il comprend le fichier XML contenant les données (construit par le JSP), ainsi que le fichier ENTREPRI.XML qui est le fichier contenant toutes les soumissions de stage approuvées. Ces fichiers sont ceux auxquels on appliquera les règles.

Figure 6.3. Le fichier DocumentSet.xml utilisé

```
<?xml version="1.0" standalone="no"?>

<!DOCTYPE DocumentSet SYSTEM " c:\jakarta-tomcat-
3.2.3\webapps\cocoon\Stages\hello\DTD\DocumentSet.dtd">

<DocumentSet name="Stages">
     <Description>Validation du stage</Description>

     <DocFile href=" c:\jakarta-tomcat-
     3.2.3\webapps\cocoon\Stages\fichiers\sabena.xml"/>   (1)

     <DocFile href=" c:\jakarta-tomcat-
3.2.3\webapps\cocoon\Stages\fichiers\ENTREPRI.XML"/>      (2)
</DocumentSet>
```

(1) Référence au fichier de l'entreprise dont on vient de saisir les données (dont le nom est repris pour créer le fichier).
(2) Référence au fichier contenant l'ensemble des entreprises ayant déjà soumis un stage.

Le fichier RuleSet.xml

Ce fichier (figure (6.4)) contient la localisation du fichier de règles qui seront appliquées lors de la validation.

Figure 6.4. Le fichier RuleSet.xml utilisé

```
<?xml version="1.0" standalone="no"?>

<!DOCTYPE RuleSet SYSTEM "c:\jakarta-tomcat-
3.2.3\webapps\cocoon\Stages\RuleSet.dtd">

<RuleSet name="Regles">
     <Description>Règles relatives à la validation
     du marché des stages</Description>
     <RuleFile href="c:\jakarta-tomcat-3.2.3
     \webapps\cocoon\Stages\rule.xml"/>                  (1)
</RuleSet>
```

(1) Référence au fichier de règles.

Le fichier rule.xml

Ce fichier (figure 6.5)) contient les règles que l'on souhaite appliquer. Chaque règle est identifiée par un attribut « id » qui permettra par la suite de pouvoir analyser les résultats.
Par un souci de clarté, seules quelques règles seront analysées (pour une analyse détaillée, voir annexe A.3.)

Figure 6.5. Exemple de règles

```
<consistencyrule id="r1">                                       (1)
     <description>                                              (2)
     Le code postal ne doit contenir que des chiffres !
     </description>

     <forall var="a" in="$entr/ADRESSE/CODE_POSTAL">            (3)
          <exists var="b" in="$a[normalize-
          space(translate(text(),'1234567890',''))='']"/> (4)
     </forall>
</consistencyrule>
```

(1) Ouverture d'une consistencyrule. Cette balise indique juste qu'on commence une nouvelle règle, elle possède un attribut 'id' (ici r1) qui identifie la règle.
(2) Chaque règle peut avoir une balise Description (elle est facultative) qui permet de décrire de façon compréhensible ce que fait la règle.
(3) Chaque règle doit avoir un Forall , car la structure type d'une règle est « pour tout x alors… »
(4) Expressions qui permettent de définir la règle souhaitée. Ici on teste une existence.
Explication de la règle :
- On sélectionne dans 'a' le code postal de l'entreprise qui nous intéresse (grâce au forall).
- On crée 'b' qui est un sous-ensemble de 'a', mais qui contient les éléments de 'a' qui sont différents de '' (la chaîne de caractères vide) une fois les caractères 1234567890 enlevés.

Cela revient en fait à regarder s'il n'y a que des chiffres dans le code postal, puisqu'on prend le code postal, on retire les caractères 1234567890 (tous les chiffres) et on regarde si le résultat est la chaîne de caractères vide, dans le cas contraire le code postal contient un autre caractère !

Les règles sont composées à partir de balises XML propres à Xlinkit, qui sont définies dans la DTD associée (ici `consistencyrule.dtd`).
On peut citer notamment la possibilité d'utiliser des opérateurs tels que :

- Et
- Ou
- Implication
- Egalité
- Sous-ensemble
- …

L'autre partie constitutive des règles est la possibilité de faire appel à des expressions XPath pour définir des contraintes supplémentaires.
XPath est nécessaire car il permet de définir des contraintes non modélisables uniquement avec les opérateurs disponibles, grâce par exemple, à l'utilisation de fonctions.

Quelques exemples de règles :

```
<consistencyrule id="r6">

     <description>
     Le nr de téléphone doit avoir au moins 8 caractères...
     </description>

     <forall var="a" in="$entr/NUM_TEL">
          <exists var="b in="$a[string-length
          (normalize-space(text())) > 7]"/>                (1)
     </forall>

</consistencyrule>
```

(1) Ici on utilise une fonction Xpath : string-length qui permet de mesurer la longueur d'un champ. On vérifie donc la longueur du numéro de téléphone, qui doit être supérieure à 7.

```
<consistencyrule id="r15">
     <description>
     La date de début du stage minimum doit être
     inférieure à sa date de début maximum
     </description>

     <forall var="a" in="$stage/DATES/VDATE_DEBUT/ANNEE">
          <exists var="b" in="$a/../../DATE_DEBUT_MAX/
          ANNEE[text()>$a/text()]"/>                       (1)
          </forall>

     </consistencyrule>

</consistencyruleset>
```

(1) Ici on vérifie que l'année de début de stage minimum est inférieure à l'année de début de stage maximum.

6.6. Traitement des résultats et affichage.

Le fichier Result.xml

Ce fichier contient le résultat de la validation. Il peut être mis en forme ensuite grâce à une stylesheet qui permettra de visualiser le résultat.

Ci-dessous (figure 6.6), un exemple de fichier résultat.

Figure 6.6. Exemple de résultat

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

<!DOCTYPE xlinkit:LinkBase SYSTEM
"http://www.xlinkit.com/DTD/LinkBase.dtd>

<?xml-stylesheet type="text/xsl" href="linkbase.xsl"?>

<xlinkit:LinkBase date="Fri Oct 19 13:39:30 CEST 2001"           (1)
docSet=" c:\jakarta-tomcat-3.2.3\webapps\cocoon\Stage\DucumentSet.xml"
ruleSet=" c:\jakarta-tomcat-3.2.3\webapps\cocoon\Stage\RuleSet.xml"
xmlns:xlink="http://www.w3.org/1999/xlink"
xmlns:xlinkit="http://www.xlinkit.com">

<xlinkit:ConsistencyLink
ruleid=" c:\jakarta-tomcat- 3.2.3\webapps\cocoon\Stages\
rule.xml#/consistencyruleset/consistencyrule[1]">                 (2)

    <xlinkit:State>consistent</xlinkit:State>                     (3)
    <xlinkit:Locator                                              (4)
      xlink:href=" c:\jakarta-tomcat-
      3.2.3\webapps\cocoon\Stages\RuleSet.dtd\result.xml#/entr
      /ADRESSE[1]/CODE_POSTAL[1]"
     xlink:label="" xlink:title=""/>

  </xlinkit:ConsistencyLink>

</xlinkit:LinkBase>
```

(1) Le chemin des DocumentSet et RuleSet qui ont été utilisés pour obtenir le fichier résultat sont indiqués.
(2) Cette ligne permet de savoir quelle règle a été utilisée pour le résultat obtenu au point (3), ici la règle utilisée était la règle 1. On peut également voir le chemin du fichier de règles.
(3) Résultat de la validation: *consistent* la règle a bien été respectée.
(4) Localisation de l'élément ayant respecté/la règle, ici entr/ADRESSE[1]/CODE_POSTAL[1].

Remarque :

A la base, Xlinkit fournit un fichier résultat portant un nom aléatoire... Il suffit donc de modifier le code source (en Java) de façon à lui passer en argument le nom du fichier résultat.

Le fichier résultat porte donc le nom de l'entreprise qui a rempli le formulaire.

Affichage

Le fichier résultat étant une feuille XML de base, il suffisait de lui associer une feuille de style XSL de façon à afficher les résultats de manière lisible.
On opta pour une sortie standard en HTML

Figure 6.7. Template XSL fournissant un message d'erreur

```
<xsl:template match="xlinkit:LinkBase">
<html>
<head>
     <title>Vous avez rempli un champ incorrectement! </title>
</head>

<body bgcolor='#FFFFFF' text='#336699' link='#6699CC'
vlink='#FF9933'>

<h1 align='center'>
<font face='Arial' size='5' color='#CC0000'>
LE MARCHE DES STAGES : SERVICE ENTREPRISE</font>
</h1>

<h2 align='center'><font face='Arial' size='4' color='#336699'>
<b>- Attention -</b></font>
</h2>

<hr width='90%'/>

<xsl:apply-templates/>

<font face='Arial' size='2' >
<p align='center'>Veuillez revenir à la page précédente et
remplir l'espace correspondant. <br></br> Merci ! </p>
</font>

</body>

</html>

</xsl:template>
```

La première template (figure (6.7)) construit la page d'erreur elle-même.
On distingue au sein du code XSL, des balises HTML permettant d'afficher un message d'erreur.

Figure 6.8. Template faisant correspondre un message d'erreur à la règle 1

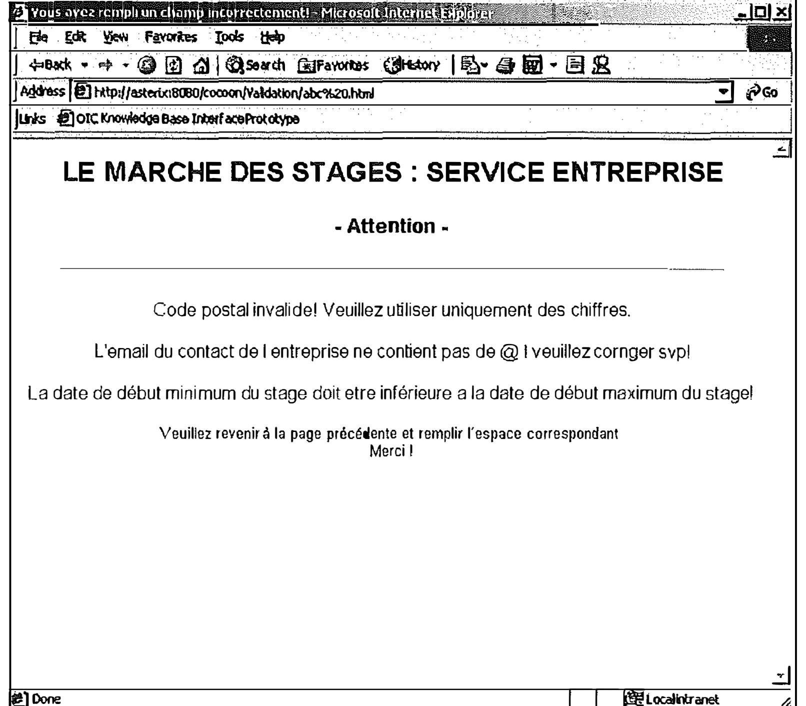
```
<xsl:template  match="xlinkit:ConsistencyLink[contains(@ruleid,'consistencyrule[1]')]">
        <p>
                <xsl:for-each select="xlinkit:State">
                        <xsl:choose>
                                <xsl:when test="text()!='consistent'">
                                        <p align='center'>
                                        <font face='Arial' size='3'>
                                        Code postal invalide! Veuillez utiliser uniquement des chiffres.
                                        </font>
                                        </p>
                                </xsl:when>
                        </xsl:choose>
                </xsl:for-each>
        </p>
</xsl:template>
```

Cette template (figure (6.8)) permet d'afficher un message d'erreur pour la règle 1 (on peut faire de même pour toutes les règles). Le message indique la cause de l'erreur.

Un exemple de message d'erreur est illustré avec la figure (6.9).

On teste si on est bien dans le cadre de la première règle : consistencyrule[1]
On affiche un message d'erreur dans le cas où la règle n'est pas respectée, donc si on a inconsistent (ici <xsl:when test="text()!='consistent'">)

Figure 6.9. Message d'erreur



La mise en forme de Cocoon

Cocoon permettait d'associer dynamiquement et automatiquement les feuilles de style aux fichiers résultats.

Le problème lié à la mise en forme par Cocoon était que le nom du fichier résultat changeait à chaque fois. Il était dès lors impossible d'associer une stylesheet à un fichier dont on ne connaissait pas le nom à l'avance (ceci étant fait dans le Sitemap)

Or, après quelques recherches, nous avons découvert qu'il existait un mode d'association générique (wildcard), en remplaçant le nom du fichier le caractère « * »

Par les quelques lignes ci-dessous (du Sitemap), nous associons à tous les fichiers HTML dont l'adresse HTTP du navigateur est `Validation` (si on tape Validation/Sabena.html par exemple), le fichier XML qui leur correspond.

Le fichier HTML sera généré par le fichier XML portant le même nom, en lui associant la stylesheet qui traite les erreurs.

```
<map:match pattern="Validation/*.html">
    <map:generate src="Stages/resultat/{1}.xml"/>
    <map:transform src="Stages/linkbase.xsl"/>
    <map:serialize type="html"/>
</map:match>
```

6.7. Evaluation et améliorations

Les résultats de l'intégration de Xlinkit dans le marché des stages étaient à la hauteur de ceux escomptés. La validation au niveau sémantique avait eu lieu, et pourrait être développée sur d'autres applications.

La prise en main de Xlinkit étant assez simple, il faut cependant souligner la nécessité de bien connaître XPath, qui est un pré-requis indispensable.
Néanmoins, il faut signaler une certaine lenteur au niveau de l'affichage des résultats, ceci étant dû à l'utilisation de Java pour la validation et Cocoon pour la publication.

L'utilisation de validateurs au niveau sémantique permet de concevoir de toutes nouvelles méthodes de validation.

On songe notamment à la validation multi-documents, qui permettrait de vérifier préalablement que les données que l'on souhaite saisir ne se trouvent pas déjà dans la base de données déjà existante, ou que la cohérence est respectée vis-à-vis de ces mêmes données.

On pense également à la possibilité de vérifier dans un fichier XML reprenant l'ensemble de données si certaines règles (basées sur une périodicité par exemple) ne sont pas violées, de façon à pouvoir enlever des données obsolètes.
Il y a bien sûr de nombreuses autres pistes.

7. Etude de cas: Génération de formulaire.

7.1. Présentation de l'étude

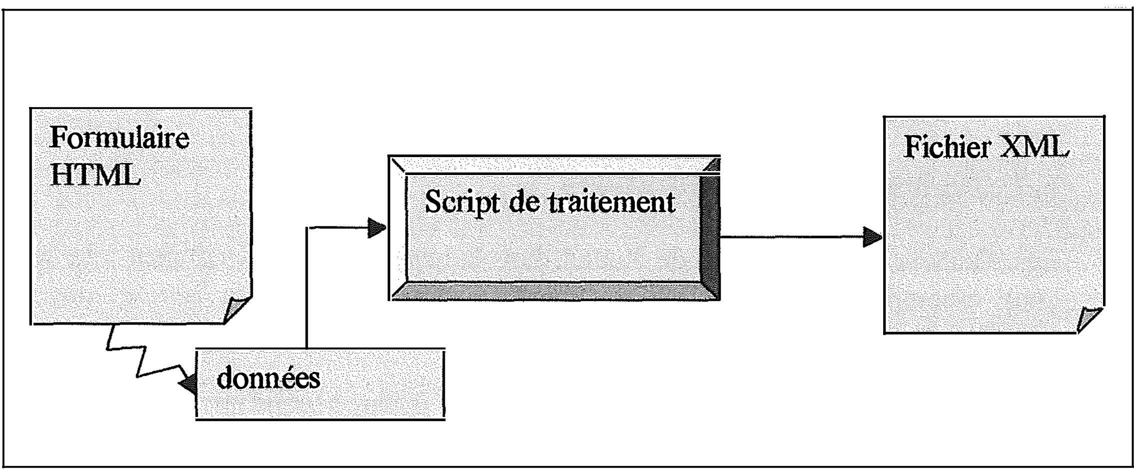
Dans cette étude de cas, nous nous intéressons aux formulaires de saisie.

En HTML, un formulaire est un mécanisme qui permet à un utilisateur d'une page web de saisir des données et de les envoyer à un serveur comme paramètres d'une requête.

La requête qui contient les données en paramètres "pointe" vers un script de traitement (CGI, PHP, JSP,...) qui exploite ces données pour un traitement particulier.

Dans notre cas, nous souhaiterions créer un formulaire où les données qui y sont saisies seraient envoyées à un script de traitement qui se chargerait de les stocker dans un fichier XML; la figure (7.1) illustre ce que nous cherchons à faire.

Figure 7.1. Fonctionnement du formulaire



Plutôt que de créer une application qui serait liée à une structure de données déterminée, il serait plus intéressant de trouver un mécanisme qui permettrait de générer une telle application à partir de la description de n'importe quelle structure de données XML.

7.2. Enjeux

Cette étude de cas toute simple nous permettra d'illustrer quelques idées que nous avons mentionnées dans les précédents chapitres. A savoir :

- Utilisation de XSLT pour générer du code HTML. Nous utiliserons ce mécanisme pour générer le formulaire HTML proprement dit.

- Utilisation de XSLT pour générer du code. Nous utiliserons ce mécanisme pour générer le code JSP.

- Utilisation du fait remarquable que les W3C XML schemas sont des documents XML. Nous utiliserons donc un document XML schema comme document XML d'entrée pour nos transformations XSLT.

- Développement d'une application possible du concept de namespace XML

Le but de l'étude n'étant pas de concevoir un générateur supportant tous les types de structures que permettent de modéliser les W3C XML-Schema, mais bien de démontrer que de tels Schemas peuvent servir de base à une transformation XSL.

7.3. Architecture

7.3.1. Le choix des technologies

Notre choix de technologie s'est porté sur HTML pour la partie visible (le formulaire proprement dit) et JSP (Java Server Page) pour la partie invisible de traitement des données du formulaire. De plus, notre code JSP nécessite la génération d'une classe bean Java.

Ce choix de technologie n'est qu'un choix parmi d'autres. La raison pour laquelle nous avons choisi JSP est la même que celle invoquée au point (6.3.1.).

7.3.2. Architecture générale

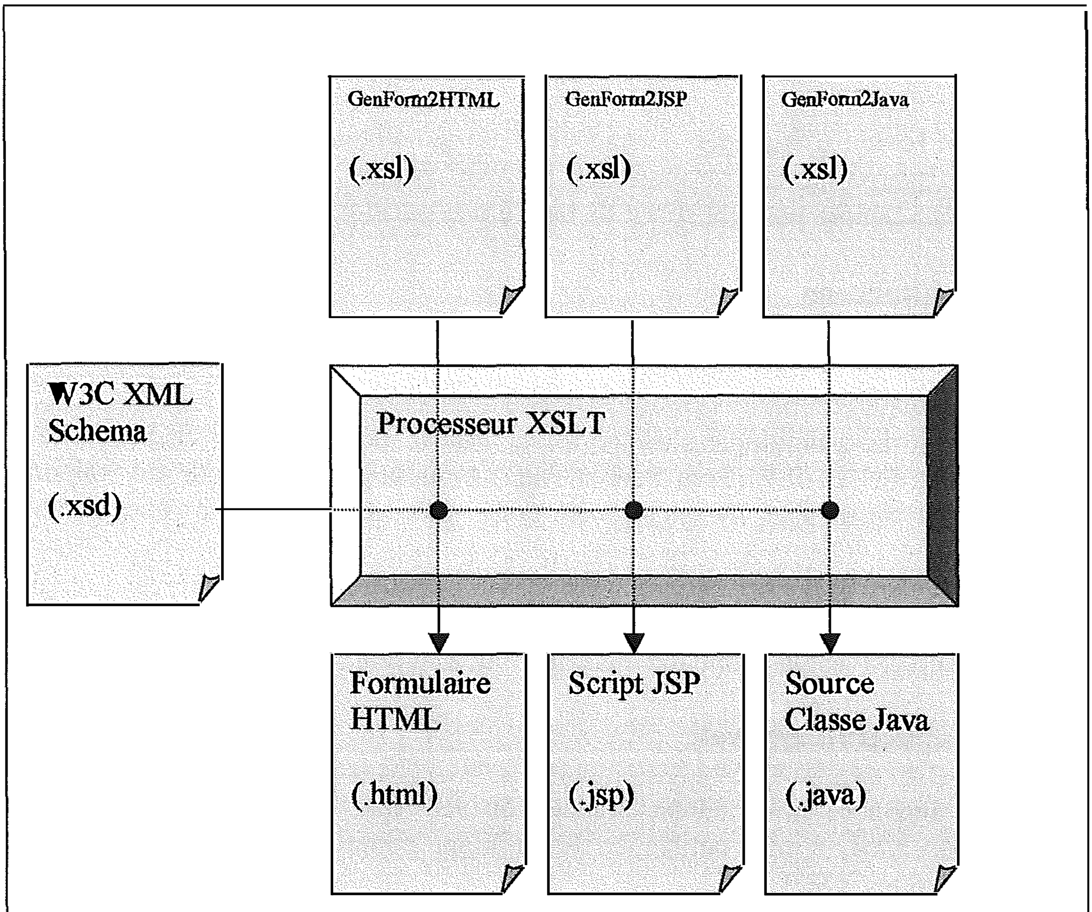
Comme annoncé, nous allons profiter du fait que les XML-Schemas soient des documents XML à part entière en les utilisant comme documents en entrée de nos transformations XSLT.

De plus, XSL pouvant générer du texte libre, rien ne nous empêche de générer du code source JSP ou Java.
La figure (7.2) résume bien l'architecture que nous emploierons, les fichiers XSL se nomment respectivement :

- ***GenForm2HTML*** : construction du formulaire HTML
- ***GenForm2JSP*** : construction du script JSP.
- ***GenForm2Java*** : construction de la source Java pour le bean.

Ces fichiers sont disponibles en [Gen02].

Figure 7.2. Architecture employée :



7.4. Options de publication : le namespace genform

Posons le problème suivant : nous souhaiterions qu'il soit possible de personnaliser le look du formulaire généré par nos feuilles de styles XSL.

Tout d'abord signalons les améliorations qui ne nécessitent aucun effort particulier. Il serait intéressant d'exploiter les possibilités des formulaires HTML pour la saisie de données de types déterminés. Le bouton radio convient très bien aux données de type booléennes et les listes déroulantes aux types énumérés. XML-Schema typant les données, la bonne exploitation de ces possibilités est facile à mettre en œuvre.

Entrons maintenant dans les difficultés avec d'autres options de présentations qu'il nous semblait intéressant d'inclure :

- ***Taille des champs texte*** : il serait intéressant de pouvoir préciser la taille (colonnes et lignes) d'un champ de texte donné afin de laisser suffisamment de place à l'utilisateur du formulaire pour rentrer ses données.
- ***Descriptif des champs de saisie*** : par défaut, notre générateur utilisera le nom de balise approprié (c'est-à-dire étant associé à ce champ de saisie) comme

descriptif d'un champ de saisie. Il serait néanmoins intéressant de pouvoir remplacer ce nom de balise par un descriptif en langage courant, pour la lisibilité du formulaire.

- ***Valeur par défaut*** : HTML autorise la mention d'une valeur par défaut pour les champs de ses formulaires. La valeur par défaut d'un champ sera inscrite dans celui-ci lors de l'affichage initial du formulaire. Cette valeur peut aider les utilisateurs à s'y retrouver dans le formulaire.

Ces trois options de publication peuvent être ajoutées en tant qu'attributs au niveau des éléments dans le XML schema. C'est en effet l'endroit le plus pertinent où les mettre vu qu'elles s'appliquent toutes à un élément particulier. Cet ajout risque néanmoins de poser un problème.

Si le XML-Schema n'est utilisé que dans le but de générer le formulaire, alors il est envisageable de rajouter les informations de mise en forme du formulaire directement dans le schema sans gêner qui que se soit. Si par contre le XML-Schema sert aussi à d'autres applications (ne fut-ce qu'être utilisé pour valider des documents XML), alors ajouter des données sur la mise en forme peut poser des problèmes de bruit et le document XML-Schema risque d'être inutilisable.

Pour résoudre ce problème, il suffit de mettre les attributs de présentations dans leur propre namespace. Les logiciels de validation les ignoreront alors et le problème est résolu. Ceci est une des applications intéressantes possibles du mécanisme des namespaces.

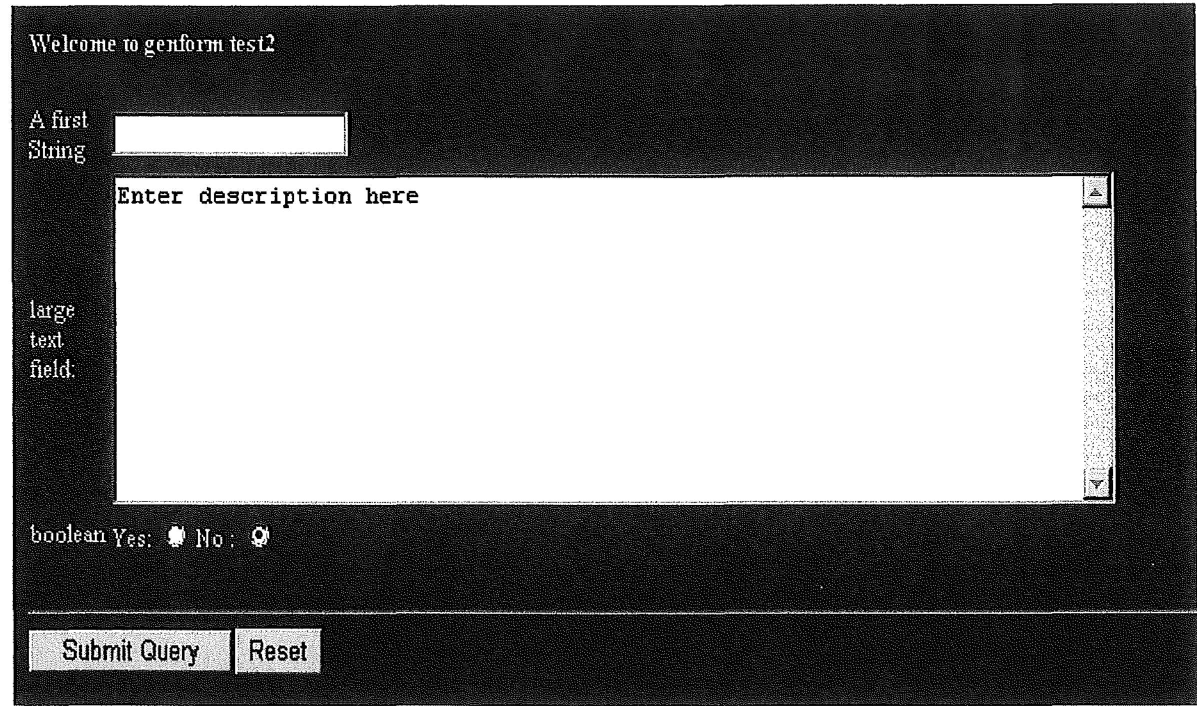
Dans notre implémentation, le namespace s'appelle ***genform***, les attributs deviennent alors :

- ***genform :cols*** et ***genform :rows*** respectivement pour le nombre de colonnes et de lignes d'un champ de texte associé à une donnée particulière
- ***genform :caption*** : pour le descriptif du champ de texte associé à cette donnée dans le formulaire
- ***genform :default*** : pour la valeur par défaut à mettre dans ce champ.

Enfin, pour ce qui est des couleurs du formulaire, elles sont regroupées dans une feuille de style CSS standard telle qu'elles sont utilisées en HTML.

La figure (7.3) montre un exemple de formulaire généré par notre système :

Figure 7.3. Exemple de formulaire généré automatiquement



7.5. Identification des variables

XML est un langage générique où l'auteur de document a tout loisir de choisir le nom de ses balises XML. En particulier, il arrive qu'un auteur de document utilise plusieurs fois le même nom de balise pour des balises différentes. Cette liberté nous pose un problème dans notre application.

Les formulaires HTML utilisent un moyen très simple pour transmettre les informations qu'ils aident à saisir. Chaque champ du formulaire est associé à un nom. Le nom de ce champ sert ensuite à récupérer la donnée qui a été saisie dedans.

Si nous utilisions les noms de balises comme noms de champs dans le formulaire HTML, une structure XML dans laquelle plusieurs balises ont le même nom ne pourraient pas être transformée correctement en formulaire. En effet, il y aurait confusion entre les données des balises de même nom.

Pour résoudre ce problème, nous avons décidé d'identifier les données en fonction de leur position dans l'arborescence de la structure. Notre notation est illustrée par ces exemples :

`el0el1el2` représente dans le code la valeur du 2ème fils du 1er fils de la racine.

`el0attName` : représente dans le code, la valeur de l'attribut Name de la balise racine.

Cette notation est purement conventionnelle.

Grâce aux prédicats XPath et aux paramètres de template XSLT, ces identificateurs de données sont générés automatiquement et de manière unique lors de la transformation avec XSL.

Il suffit simplement que les feuilles de style s'occupant de la mise en forme du formulaire et celles s'occupant des scripts de traitement parcourent le document XML-Schema dans le même sens (en utilisant la même structure de template et le même processeur XSLT) et nous avons les garanties que :

- les données sont toutes identifiées de manière unique.
- les identificateurs de données générés par une des feuilles de style XSL sont les mêmes que ceux des autres feuilles et identifient bien les mêmes données.

7.6. Structures supportées

Notre générateur de formulaire ne supporte pas tous les types de structure que proposent les W3W XML-Schemas. Ce n'était pas le but de l'étude de cas.

Il se contente des suivantes:

- éléments/balises simples
- attributs de balises
- les structures `<xsd:choose>` (une structure contient soit une sous structure , soit une autre)

7.7. Améliorations possibles

Il existe un certain nombre d'améliorations à apporter à notre système.

La première est bien sûr d'élargir le nombre de structures que notre générateur peut gérer, parmi celles qui sont les plus intéressantes, les répétitions de balises (une balise X contient 0 à n balises Y).

Enfin, il serait intéressant de pousser le typage un peu plus loin et d'inclure des vérifications des données du formulaire avant de créer le document XML. Cette vérification peut se faire simplement avec Javascript ou bien plus en détail avec l'utilisation des technologies de validation présentées dans ce travail.

8. Etude de cas : La recherche ensembliste.

8.1. Présentation de l'étude

Dans cette étude de cas, nous nous intéressons aux moteurs de recherche.

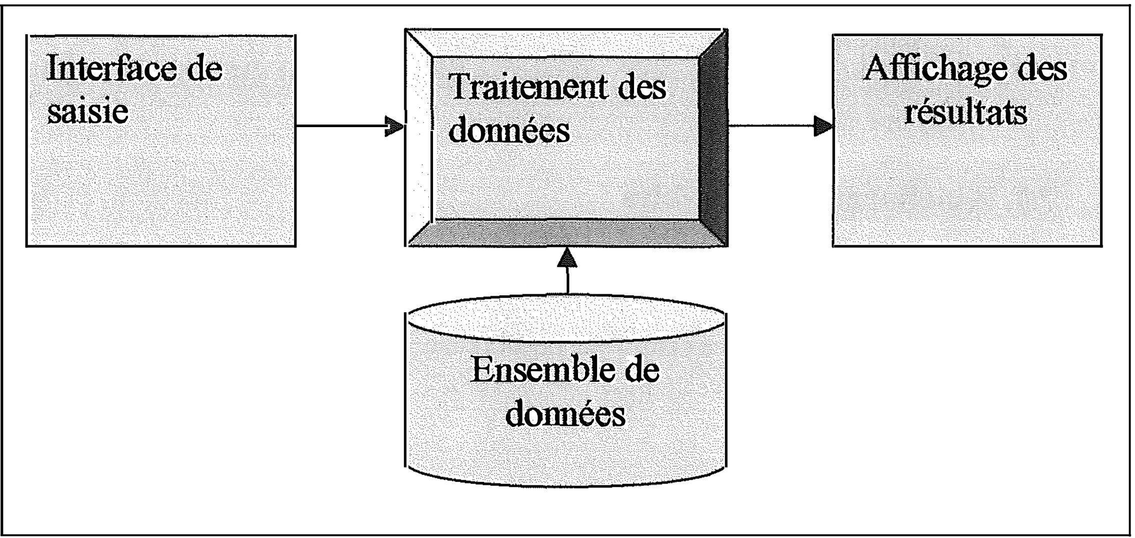
Qu'est-ce qu'un moteur de recherche?
Un moteur de recherche est une application permettant de sélectionner des données particulières dans un ensemble de données.

Il fonctionne de la manière suivante :

L'utilisateur établit, via l'interface de l'application de recherche, une liste de contraintes que devront respecter les données sélectionnées.
Le moteur de recherche examine alors l'ensemble de données et sélectionne parmi celles-ci celles qui répondent aux contraintes définies par l'utilisateur.
Enfin, il fournit à l'utilisateur le résultat de sa recherche.

La figure (8.1) illustre ce mécanisme.

Figure 8.1. Exemple de moteur de recherche



Dans le cadre de notre étude, nous allons nous intéresser à l'opportunité d'utiliser les technologies XML pour réaliser un tel moteur de recherche. En particulier :

- L'ensemble des données sera modélisé par un document XML.
- Le moteur de recherche utilisera XSLT pour sélectionner les données qui respectent les contraintes définies par l'utilisateur.
- La publication se fera via XSLT et éventuellement en utilisant un framework de publication.

8.2. Enjeux

L'enjeu principal est de montrer qu'il est possible d'utiliser un fichier XML comme une base de données dans laquelle on pourrait sélectionner des données.

Le but de cette étude n'étant pas la conception d'un moteur de recherche permettant de définir les contraintes les plus complexes possibles (il ne s'agit pas de modéliser un langage de requêtes comme SQL), mais bien de montrer qu'il est possible de définir un tel système en utilisant des technologies XML. Nous simplifierons donc le problème à l'extrême, en utilisant un système de règles très rudimentaire et une structure de données XML qui se prête facilement aux traitements qui nous intéressent.

Nous nous attarderons sur la paramétrisation de la publication des résultats. A savoir, permettre à l'utilisateur de choisir la manière dont ses données seront publiées.

Ceci passe par l'utilisation des concepts et technologies que nous avons introduits dans les chapitres précédents. A savoir :

- Au niveau du traitement des requêtes, utilisation de XSLT pour filtrer les données pour ne garder que celles qui sont intéressantes et pour optimiser les résultats de requêtes en éliminant les informations redondantes.

- Au niveau de la publication des résultats, utilisation de XSLT pour afficher le résultat de la recherche en tenant compte des options de présentation choisies par l'utilisateur. Ceci se fait grâce à la possibilité de passer des paramètres à une feuille de style XSL.

8.3. Architecture

8.3.1. Le choix des technologies

Nous avons choisi les technologies suivantes :

- HTML : Choix évident pour la publication et pour les formulaires d'entrée.

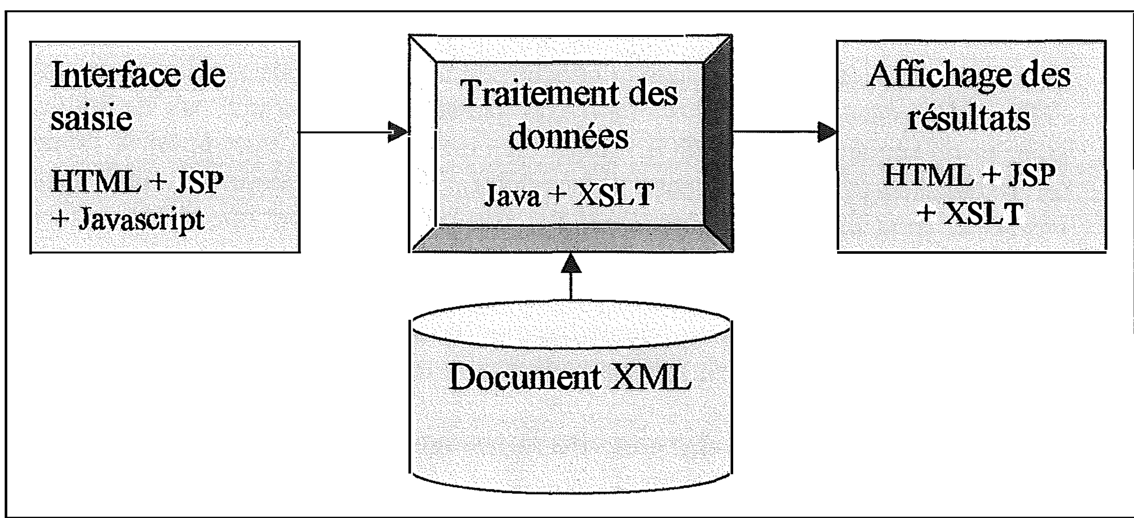
- Javascript : Utilisé dans le cadre des formulaires d'entrée.

- Java/JSP : Utilisé pour les mêmes raisons que celles invoquées dans le chapitre 6 : le marché des stages. En outre, le processeur XSLT que nous avons utilisé le plus souvent (Xalan, utilisé notamment par Cocoon) est écrit en Java, ce qui nous conforte dans ce choix. Le JSP étant utilisé au-dessus d'un serveur Apache-Tomcat (version 3.2.3)

- XML et XSLT

La figure (8.2) illustre ce choix de technologies.

Figure 8.2. Architecture de notre moteur de recherche



8.3.2. Architecture générale

L'interface de saisie des contraintes utilise HTML ainsi qu'un peu de Javascript. JSP se charge de récupérer les contraintes ainsi saisies.

Le moteur de traitement des données est un moteur Java/JSP. Ce moteur décompose les contraintes qu'il reçoit en sous-tâches qu'il soumet à Xalan, le processeur XSLT que nous avons choisi. Xalan applique ensuite des feuilles de style XSLT au fichier de données XML. Les documents XML ainsi générés sont récupérés par le moteur Java/JSP qui crée le fichier contenant les résultats correspondant à la requête.

La publication utilise la combinaison des technologies HTML, Javascript et JSP pour présenter à l'utilisateur un formulaire lui permettant de choisir la manière dont ses résultats seront affichés. Il utilise ensuite XSLT pour concrétiser ce choix.

8.3.3. Le fichier XML de données.

La structure de données avec laquelle le moteur de recherche va travailler est une structure très simple, dont voici la DTD :

```
<!ELEMENT child (#PCDATA)>
<!ATTLIST child name  #REQUIRED>
<!ELEMENT document (entry+)>
<!ATTLIST document name CDATA #REQUIRED>
<!ELEMENT entry (child+)>
<!ATTLIST entry id #REQUIRED>
```

Chaque élément "`entry`", doit être identifié par un attribut "`id`" unique.

Voici un exemple de document XML qui répond à notre DTD

```
<document name="exemple">
      <entry id="122">
            <child name="valeur1">15</child>
            <child name="valeur2">1</child>
            <child name="valeur3">12</child>
            <child name="valeur4">20</child>
            <child name="valeur5">36</child>
      </entry>
      <entry id="123">
            <child name="valeur2">1</child>
            <child name="valeur3">20</child>
            <child name="valeur5">47</child>
            <child name="valeur6">10</child>
      </entry>
</document>
```

Cette structure XML n'est certainement pas la manière la plus optimale pour représenter des données, ce n'est pas non plus la plus lisible. Elle a cependant le mérite d'être très générique et de convenir à nos besoins.

Rappelons qu'il est toujours possible de convertir une structure XML quelconque vers notre structure via l'utilisation de XSL.

8.3.4. Le système de règles

Notre système de règles est lui aussi très simple.

La syntaxe des règles est la suivante:

```
<règle générale> ::= UNION( <sous-règles>*)
<sous-règle> ::= INTERSECTION (<contraintes>*)
<contraintes> ::= SELECTIONER LES <attribut:name> = <valeur>
```

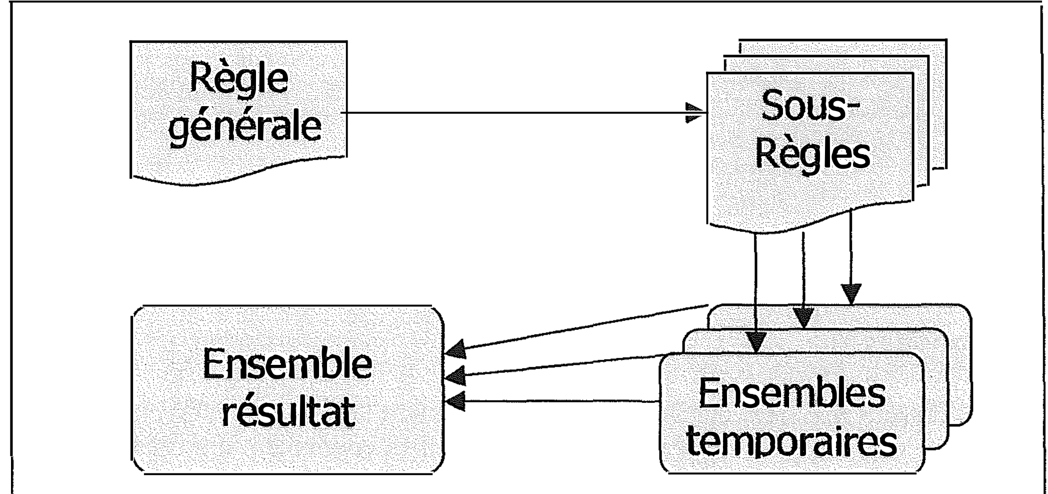
ce choix est motivé par le fait qu'il nous permet de garder un moteur de traitement assez simple. En effet, autoriser le mélange de ET et de OU dans une contrainte nous forcerait à devoir analyser les contraintes et déterminer les priorités relatives des sous-règles. En restreignant la syntaxe comme nous l'avons fait, l'algorithme de fonctionnement de notre moteur de recherche se présente comme suit (figure (8.3)).

1) Décomposer la règle en une union de sous-règles.

2) Pour chaque sous-règle, appliquer les contraintes l'une après l'autre sur le fichier de données. Ceci revient bien à en faire l'intersection. En effet, une donnée déjà éliminée par l'application d'une contrainte ne peut plus être sélectionnée par la suivante. Les données restantes à la fin du processus sont donc bien les données qui satisfont à toutes les contraintes. Cette étape de l'algorithme fournit un sous-ensemble de données provenant de l'ensemble des données de départ.

3) Une fois les sous-ensembles correspondants à chaque sous-règle définis, le moteur n'a plus qu'à les réunir dans un unique ensemble qui deviendra alors le résultat final.

Figure 8.3. Décomposition d'une règle

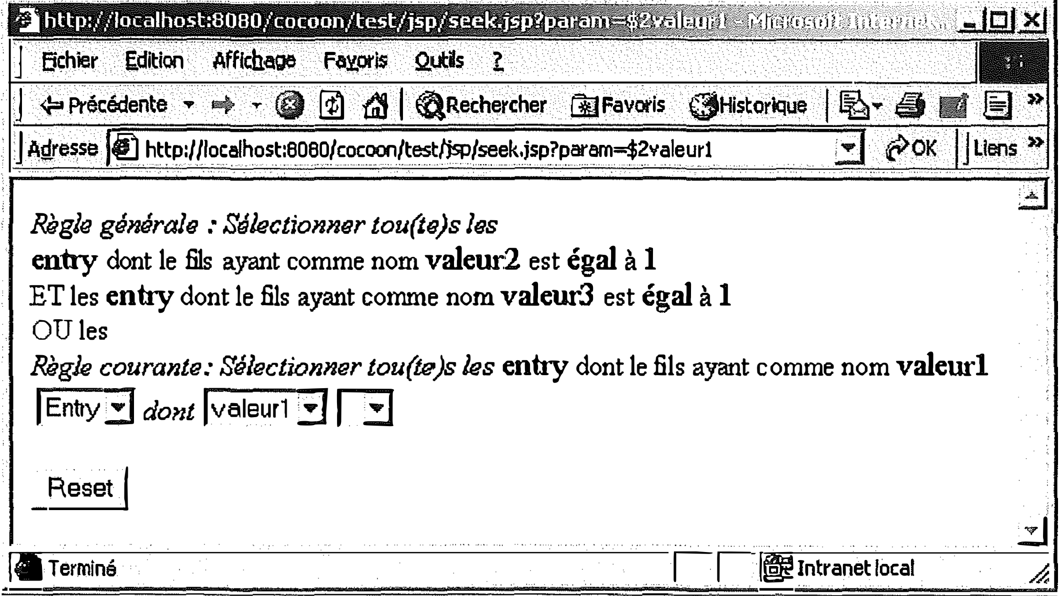


8.4. La saisie de paramètres

La saisie se fait via une interface web qui se charge de définir la requête à sélectionner, sous forme de règles alliant des ET des OU.
Cette page web est une feuille JSP qui récupère pas à pas les éléments composant la règle.

La figure (8.4) est une capture d'écran de notre formulaire de saisie.

Figure 8.4. Formulaire de saisie



L'affichage de la règle générale, ainsi que de la règle courante, est mis à jour dynamiquement grâce à l'utilisation du JSP, ce qui permet à l'utilisateur de suivre l'évolution des règles qu'il définit.

Les menus permettant de générer les règles sont gérés dynamiquement grâce à l'utilisation de Javascript. Ainsi, certains boutons apparaissent uniquement quand leur utilisation est possible.

8.5. Le traitement des paramètres

La règle générale est obtenue après la saisie, via une variable JSP.
Elle est stockée de manière à être facilement décomposable en sous-règles. Ceci est possible par l'utilisation de séparateurs qui jouent le rôle de ET.

Chaque sous-règle est ensuite décomposée en contraintes. Ceci est possible par l'utilisation de séparateurs qui jouent le rôle de OU.

Ainsi, après avoir décomposé les règles en sous-règles, et les sous-règles en contraintes, il reste à transformer le fichier XML de données de façon à obtenir un fichier XML résultat correspondant à la règle définie initialement.

Comme on l'a expliqué dans le système de règles, il suffit de faire l'intersection des contraintes d'une sous-règle pour obtenir le fichier qui correspond à la sous-règle.
Pour ce faire, on a défini une feuille de style XSL (figure (8.6)), qui permet de filtrer un document XML de façon à ne garder que les éléments qui respectent la contrainte.
Il suffit de passer comme paramètre à la feuille de style, les éléments de la contrainte :

- Le nom de l'élément concerné
- La valeur recherchée

Ces deux paramètres sont réunis en un seul et sont séparés par le caractère "/".

La figure (8.5) montre un exemple concret de traitement de règles en termes de fichiers.

Figure 8.5. Exemple de traitement pour la sous-règle i ayant n contraintes

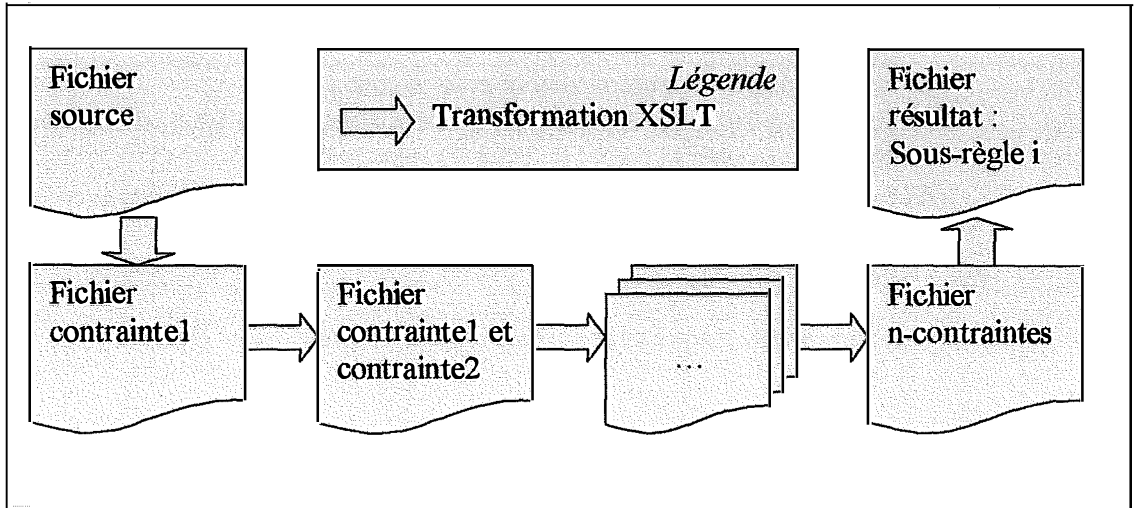


Figure 8.6. Feuille de style de "filtrage"

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:fo="http://www.w3.org/1999/XSL/Format">

<xsl:param name="garder">valeur par défaut</xsl:param> (1)

<xsl:param name="nom">
<xsl:value-of select="substring-efore($garder,'/')"/>
</xsl:param>                                           (2)
<xsl:param name="valeur">
<xsl:value-of select="substring-after($garder,'/')"/>
</xsl:param>                                           (2)

<xsl:template match="*">
  <xsl:element name="{name()}">
     <xsl:for-each select="@*">
          <xsl:attribute name="{name()}">
               <xsl:value-of select="."/></xsl:attribute>
     </xsl:for-each>

     <xsl:for-each select="child::*">
          <xsl:for-each select="child::*">
               <xsl:if test="@*[name()]=$nom">          (3)
               <xsl:if test="text()=$valeur">          (3)
                    <xsl:copy-of select="./.."/>       (4)
               </xsl:if>
               </xsl:if>

          </xsl:for-each>
     </xsl:for-each>

  </xsl:element>

</xsl:template>

</xsl:stylesheet>
```

(1) La valeur initiale du paramètre sera écrasée lors du passage de paramètre par Xalan
(2) On extrait du paramètre le nom et la valeur à tester
(3) On teste si l'attribut "name" de l'élément correspond avec celui recherché, ainsi que sa valeur.
(4) La contrainte étant respectée, on recopie la totalité de l'élément se trouvant au niveau supérieur (ici entry)

Le passage de paramètre est possible lors de l'appel du processeur Xalan.

Exemple :

```
java org.apache.xalan.xslt.Process -IN donnees.xml -XSL
filtrage.xsl -OUT resultat.xml -Param garde nom3/7
```

L'exemple donné ici transformerait la feuille XML "`donnees.xml`" en feuille XML "`resultat.xml`" en utilisant la feuille XSL "`filtrage.xsl`" avec comme paramètre "`garder`" qui prend la valeur "`valeur3/7`".

La feuille de style vérifiera pour tout élément "`entry`" si un de ses fils a un attribut dont le nom est "`valeur3`" et si la valeur de celui-ci est "7".

Si c'est le cas, alors l'élément "`entry`" sera entièrement recopié dans le fichier résultat.

Les fichiers XML résultats de toutes les sous-règles sont alors rassemblés en un seul fichier. Ceci est possible en utilisant la méthode java `getRootElement()` qui permet de récupérer l'élément racine. Et la méthode `getChild()` qui permet de récupérer les sous-éléments.(JDOM API [Jdom00])

Néanmoins, le fichier pouvant contenir des redondances, on lui applique une feuille de style XSL (figure 8.7) permettant de retirer les doublons.

Figure 8.7. La feuille de style "doublons"

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:fo="http://www.w3.org/1999/XSL/Format">

<xsl:template match="*">
  <xsl:element name="{name()}">
     <xsl:for-each select="child::*">
       <xsl:variable name="id" select="@id"/>          (1)
          <xsl:if test="(count(following-
          sibling::*[@id=$id])=0)">                    (2)
               <xsl:copy-of select="."/>               (3)
          </xsl:if>

     </xsl:for-each>

  </xsl:element>

</xsl:template>

</xsl:stylesheet>
```

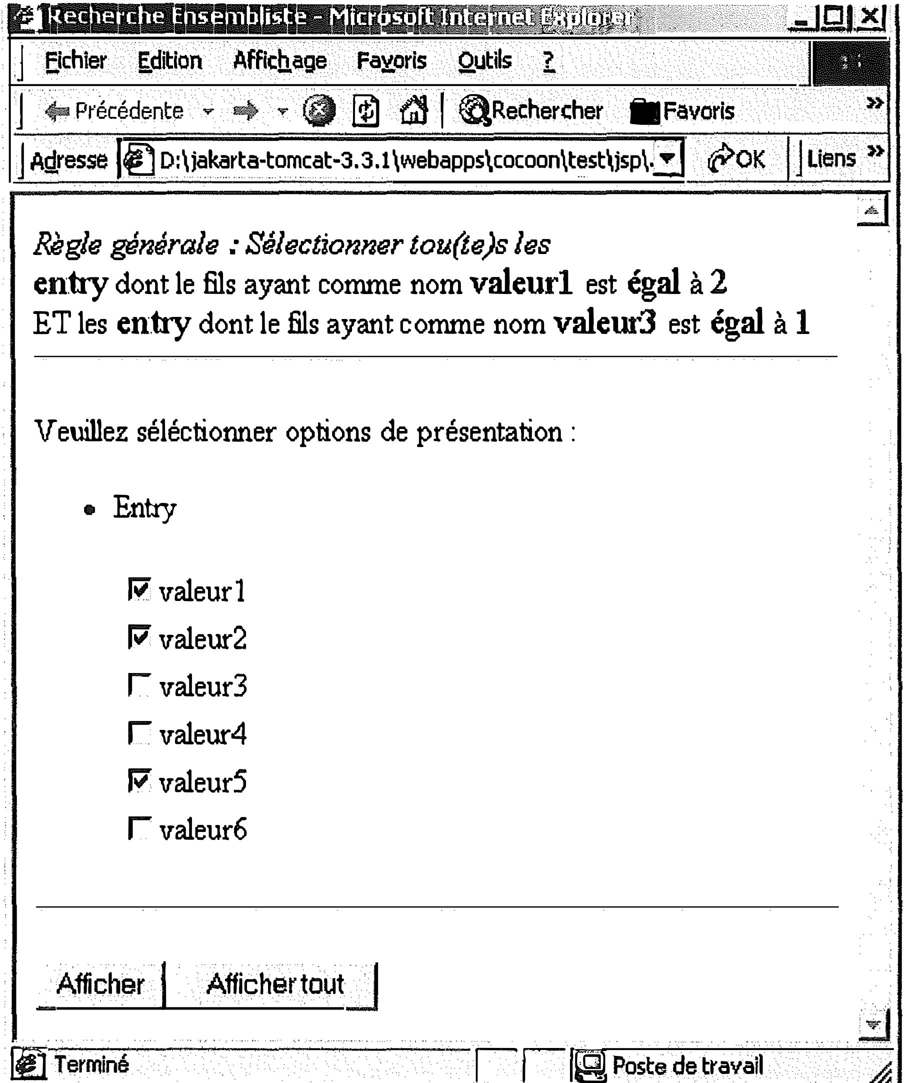
(1) On utilise une variable pour stocker le numéro d'attribut
(2) Comme chaque élément "entry" a un attribut "id" unique, il suffit de vérifier qu'il n'existe pas, dans les éléments suivants, d'autre attribut ayant le même numéro.
(3) Si on ne trouve pas d'autre élément ayant le même numéro, on recopie l'élément tel quel, car il est unique.

Le fichier résultat contient donc le résultat de la règle générale.

8.6. l'affichage des paramètres

L'affichage des résultats utilise le même principe que la saisie des paramètres.
L'utilisateur a le choix des options de présentation (figure (8.8)), sous forme d'une feuille JSP.

Figure 8.8. Exemple de choix de présentation.



Une fois le choix validé, les paramètres de présentation sont fournis à une feuille de style XSL (figure (8.9)) qui se charge d'afficher uniquement ce qui a été demandé.

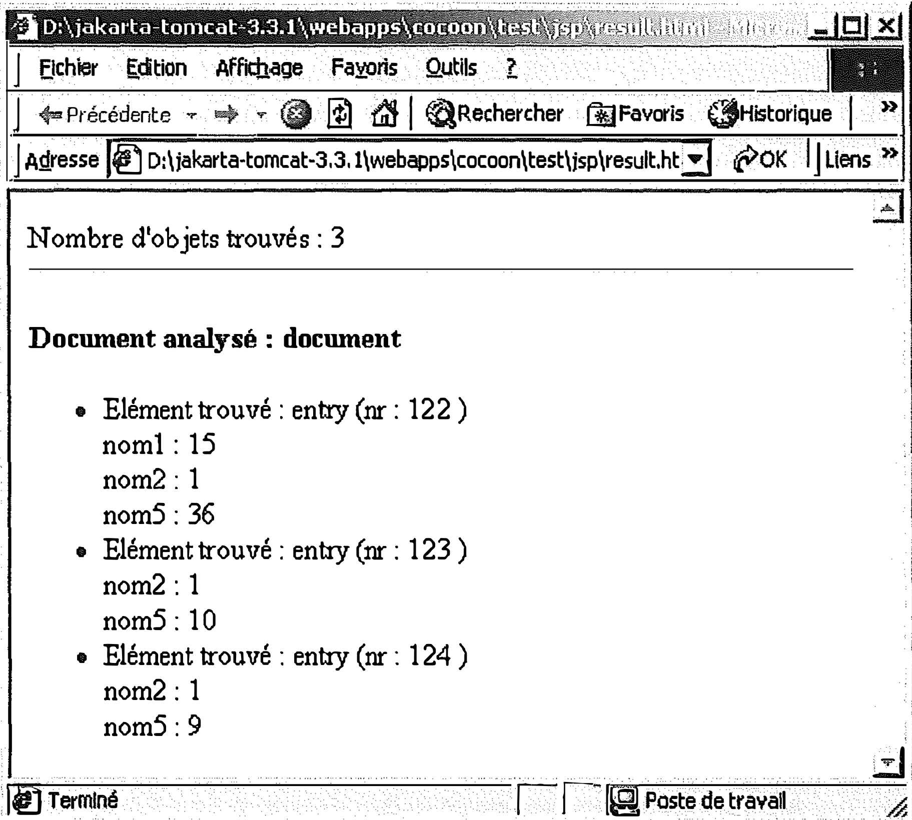
Le résultat obtenu est du même type que celui présenté avec la figure (8.10)

Figure 8.9. Détail de la feuille de style "afficher"

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:fo="http://www.w3.org/1999/XSL/Format">

<xsl:param name="affich">valeur1/valeur2/valeur5      (1)
</xsl:param>

<xsl:template match="*">
<html>
<body>
Nombre d'objets trouvés :
<xsl:value-of select="count(child::*)"/>                (2)
      <hr></hr><br></br>
      <b>Document analyse :
      <xsl:value-of select="name()"/></b><br></br>
      <ul>
      <xsl:for-each select="child::*">

      <li>Element trouve :
      <xsl:value-of select="name()"/> (nr :
      <xsl:value-of select="@id"/> )<br></br>
        <xsl:for-each select="child::*">
        <xsl:if test="contains ($affich,@*[name()])"> (3)
           <xsl:value-of select="@*[name()]"/> :
           <xsl:value-of select="text()"/> <br></br>  (4)
        </xsl:if>

        </xsl:for-each>

      </li>
      </xsl:for-each>
      </ul>

</body>

</html>

</xsl:template>

</xsl:stylesheet>
```

(1) Le paramètre `affich` contient les choix de l'utilisateur
(2) Cette ligne permet de compter le nombre d'éléments trouvés.
(3) On teste si l'élément doit être affiché (si le nom de son attribut est dans la liste fournie par le paramètre `affich`).
(4) On recopie le nom et la valeur de l'élément traité.

Figure 8.10. Exemple d'affichage de résultats



8.7. Evaluation et améliorations possibles

La recherche ensembliste était un défi d'une certaine envergure. En effet, c'est une application à part entière qu'il fallait concevoir. Malheureusement, par manque de temps, nous n'avons pu arriver au bout de celle-ci.
Néanmoins, tous les éléments que nous avons développés précédemment fonctionnent correctement.. C'est le temps de les intégrer tous ensemble qui nous a manqué.

Les améliorations possibles sont très nombreuses.

Tout d'abord, il pourrait être intéressant d'intégrer le générateur de formulaire à la recherche ensembliste. De cette façon, les formulaires de saisie et d'affichage seraient générés de manière totalement automatisée à partir d'un XML Schema.

Il serait intéressant de pouvoir élargir le champ d'action de la recherche (pour le moment très simple).

Cela pourrait être fait en étoffant le fichier XML normalisé, de manière à supporter une gamme de données plus étendue, mais aussi en élargissant la définition des règles, afin de pouvoir traiter des expressions variées (pour le moment on a juste attribut = valeur)

Il serait également intéressant, au niveau de l'affichage des résultats, d'utiliser le framework de publication Cocoon, qui se chargerait d'associer automatiquement la feuille de style avec le fichier résultat. Cela nécessite d'approfondir le passage de paramètres à une feuille de style par une URL.

9. Conclusion

Le langage XML et les technologies qui l'entourent n'ont pas fini de se développer.
Il paraît certain que ces technologies vont continuer à se diffuser et, probablement, devenir incontournables dans les années à venir.

Au travers des différents chapitres que nous avons présentés, nous espérons avoir donné un aperçu de la puissance de la généricité du langage XML.

Le langage XSLT, associé à XPath, ouvre énormément de possibilités aux développeurs XML que ce soit au niveau de la publication de données provenant de documents XML, de la génération de code, ou encore du filtrage d'informations (par transformations successives).

Le langage XSLFO quant à lui, élargit encore plus les possibilités qui sont offertes au développeur en matière de publication.

L'utilisation de frameworks de publication comme Cocoon permet une association dynamique et intelligente entre les données d'un document XML et les feuilles de style XSL qui dictent leur mise en forme. Cette association dynamique permet de résoudre bon nombre de problèmes auxquels les "webmasters" sont confrontés. Des économies d'échelle assez importantes peuvent être réalisées par l'utilisation de tels frameworks, en complément de technologies de base de données et de scripts classiques.

La problématique de la validation de documents est un enjeu capital pour l'interopérabilité entre systèmes d'information. Dans un premier temps, la validation à un niveau purement syntaxique, grâce à l'utilisation de formalismes comme les DTDs et les W3C XML Schemas, permet de s'assurer de la structure et de la syntaxe du document, ensuite, la validation au niveau sémantique, par le biais de technologies comme Schematron et Xlinkit, permet un contrôle du contenu des données.

Ces technologies pourraient se substituer aux mécanismes de contrôle habituels (c'est-à-dire devoir écrire des morceaux de code qui seront chargés de vérifier les données au cas par cas), sans pour autant demander d'efforts supplémentaires. De plus l'adoption de ces mécanismes de validation, simples à mettre en œuvre, permettrait à bon nombre d'applications, ne disposant pas de validation, d'éviter des erreurs.

Enfin, signalons que la plupart des technologies connexes au langage XML utilisent elles-mêmes la syntaxe XML. C'est le cas, par exemple, des feuilles de style XSL, des W3C XML Schemas ou encore du sitemap Cocoon. Il est donc possible de traiter, générer ou transformer ce type de document via les technologies XML elles-mêmes. Dans le cadre de notre travail, nous avons illustré ce mécanisme en présentant Schematron et dans l'étude de cas traitant de la génération de formulaire. Nous sommes très enthousiastes à l'idée de pouvoir encore ultérieurement approfondir cette perspective.

Bibliographie

Ouvrages

[Kay01] M. Kay, "XSLT Programmer's reference, second edition", Wrox Press 2001.

[Mic99] A. Michard, "XML, Langage et Applications", Eyrolles, 1999 1999, Paris.

[NoHe99] S. North et P. Hermans, "XML"(le programmeur), CampusPress, 1999, Paris.

Sites

[ApCo02] The Apache Software Foundation, "Apache Cocoon", http://xml.apache.org/cocoon/index.html, last updated July 2002.

[ApFo02] The Apache Software Foundation, "FOP" http://xml.apache.org/fop/index.html, last updated 2002.

[ApTo02] The Apache Software Foundation,"The Jakarta Site - Apache Tomcat", http://jakarta.apache.org/tomcat/index.html, last updated 2002.

[ApXa02] The Apache Software Foundation, "Xalan-Java version 2.4.D1", http://xml.apache.org/xalan-j/index.html, last updated 2002.

[CGI99] Connolly, "CGI - Common Gateway Interface", http://www.w3.org/CGI/, last updated Oct-13-1999.

[CRP02] "CRP - Tudor Online", http://www.tudor.lu, last updated 2002.

[CSS02] B. Bos, "Cascading Style Sheets" http://www.w3.org/Style/CSS/, last updtated Aug-14-2002.

[Dtd00] T. Bray, J. Paoli, C. M. Sperberg-McQueen and E. Maler, "Extensible Markup Language (XML) 1.0 (Second Edition)", http://www.w3.org/TR/REC-xml#dt-doctype, last updates Oct-06-2000.

[Fac02] W3Schools, "XSD Restrictions/Facets", http://www.w3schools.com/schema/schema_facets.asp, last updated 2002.

[Gen02] M. Robbe and S. Walgraffe, "fichiers de Genform", http://www.info.fundp.ac.be/~swalgraf/memoire, last updated Aug-2002.

[Html02] M. Ishikawa, "W3C HTML Home Page", http://www.w3.org/MarkUp/, last updated Aug-24-2002.

[Jdom00] Jason Hunter and Brett McLaughlin, "Easy Java/XML integration with JDOM, Part 1", http://www.javaworld.com/javaworld/jw-05-2000/jw-0518-jdom_p.html, May 2000.

[Jsp02] Sun Microsystems, Inc., "JavaServer Pages(TM) Technology", http://java.sun.com/products/jsp/ , last updated Aug-28-2002

[Jav02] Sun Microsystems, Inc., "The Source for Java(TM) Technology", http://java.sun.com/, last updated Aug-30-2002.

[Jfo02] "jfor - Open-Source Java XSL-FO to RTF converter - XML to RTF publishing", http://www.jfor.org/, last updated July-11-2002.

[Lbl01] D. Girard, T. Crusson, "XML pour l'entreprise", http://www.application-servers.com/livresblancs/xml/, version provisoire 0.90, last updated 2001.

[Php02] The PHP Group, "PHP: Hypertext Preprocessor", http://www.php.net/, last updated Aug-15-2002.

[Rel02] "OASIS Technical Committee: RELAX NG", http://www.oasis-open.org/committees/relax-ng/, last updated July-17-2002.

[Sch02] R. Jelliffe, "Academia Sinica Computing Centre's Schematron Home Page", http://www.ascc.net/xml/resource/schematron/schematron.html, last updated June 2002.

[Sgm00] D. Connolly, "Overview of SGML Resources", http://www.w3.org/MarkUp/SGML/, last updated Aug-23-2000.

[Tre02] J. Clark, "TREX", http://www.thaiopensource.com/trex/, last updated 2002.

[Ur02] D. Connolly, "Web Naming and Addressing Overview (URIs, URLs, ...)", http://www.w3.org/Addressing/, last updated July-09-2002.

[W3c02] W3C® (MIT, INRIA, Keio), "The World Wide Web Consortium", http://www.w3.org, last updated Aug-30-2002.

[Xht02] W3C HTML Working Group, "XHTML 1.0: The Extensible HyperText Markup Language (Second Edition)", http://www.w3.org/TR/xhtml1/, last updated Aug-01-2002.

[Xli02] Systemwire, "Xlinkit main page", http://www.xlinkit.com/, last updated 2002 .

[Xml02] W3C® (MIT, INRIA, Keio), "Extensible Markup Language (XML)" http://www.w3.org/XML/, last updated Aug-13-2002.

[XmlS02] C. M. Sperberg-McQueen and H. Thompson, "W3C XML Schema", http://www.w3.org/XML/Schema, last updated Aug-16-2002.

[Xpa99] J. Clark and S. DeRose, "XML Path Language (XPath)", http://www.w3.org/TR/xpath, last updated Nov-16-1999.

[Xsl02] M. Froumentin, team contact for the XSL Working Group, " The Extensible Stylesheet Language (XSL)", http://www.w3.org/Style/XSL/, last updated Aug-23-2002.

[Xslt99] J. Clark, "XSL Transformations (XSLT)", http://www.w3.org/TR/xslt, last updated Nov-16-99

[Xvif02] C. Chiaramonti, "xvif pour le mapping", http://xmlfr.org/actualites/decid/020715-0001, last updated July-15-2002.

[Pxv02] E. van der Vlist, "Project info for xvif", http://www.advogato.org/proj/xvif/, last updated Jun-20-2002.

ANNEXES

A.1. Les modes de recherche avec XPath

Self : la balise elle-même.

Child : mode par défaut, de parent vers enfants directs.

Parent : des enfants vers leur parent direct.

Descendant : d'une balise vers toutes les balises qu'elle entoure (en ce compris ses filles directes mais également ses « petites-filles »,...)

Ancestor : d'une balise vers toutes les balises qui l'entourent (en ce compris le parent direct, mais aussi la balise « grand parent »,...)

Descendant-or-self : d'une balise vers ses descendants ou elle-même.

Ancestor-or-self : d'une balise vers ses ancêtres ou elle-même.

Following : d'une balise vers toutes celles qui la suivent dans le document mais qui ne sont pas ses descendants.

Preceding : d'une balise vers toutes celles qui la précèdent dans le document mais qui ne sont pas ses ancêtres.

Following-Sibling : d'une balise vers toutes celles qui la suivent et ne sont pas ses descendants et qui partagent le même ancêtre.

Preceding-Sibling : d'une balise vers toutes celles qui la précèdent dans le document, qui ne soient pas un de ses ancêtres et qui partagent le même ancêtre.

Les deux axes « sibling » sélectionnent dès lors des balises qui se trouvent au « même niveau d'indentation » dans le document.

Pour fixer les idées, la figure (A.1) montre un exemple de fichier XML à plusieurs niveaux et le tableau (A.1) reprend les différents modes de recherche et les résultats correspondant à l'application de ce mode de recherche aux différentes balises du document.

Figure A.1. Les modes de recherche, exemple

```
<root>
<a>
      <b/>
           <c>
                  <d/>
            </c>
      <e/>
</a>
<f>
      <g>
            <h/>
            <i/>
      </g>
      <j/>
</f>
<k/>
</root>
```

Tableau A.1 : Eléments sélectionnables selon les différents modes de recherche :

self	child	parent	Ancestor	descendant	ancestor-or-self
a	b,c,e	root	root	b,c,d,e	root,a
b	-	a	root,a	-	root,a,b
c	d	a	root,a	d	root,a,c
d	-	c	root,a,c	-	root,a,c,d
e	-	a	root,a	-	root,a,e
f	g,j	root	root	g,h,i,j	root,f
g	h,i	f	root,f	h,i	root,f,g
h	-	g	root,f,g	-	root,f,g,h
i	-	g	root,f,g	-	root,f,g,i
j	-	f	root,f	-	root,f,j
k	-	root	root	-	root,k

Descendant-or-self	preceding	Following	preceding-sibling	following-sibling
a,b,c,d,e	-	f,g,h,i,j,k	-	f,k
b	-	c,d,e,f,g,h,i,j,k	-	c,e
c,d	b	e,f,g,h,i,j,k	b	e
d	b	e,f,g,h,i,j,k	-	-
e	b,c,d	f,g,h,i,j,k	b,c	-
f,g,h,i	a,b,c,d,e	k	a	k
g,h,i	a,b,c,d,e	j,k	-	j
h	a,b,c,d,e	i,j,k	-	i
i	a,b,c,d,e,h	j,k	h	-
j	a,b,d,e,g,h,i	k	g	-
k	a,b,c,d,e,f,g,h,i, j	-	a,f	-

En plus de ces modes directs, il en existe deux autres *:* ***attribute*** qui sélectionne les attributs d'une balise et ***namespace*** qui sélectionne les namespace attachés à une balise.

A.2. Exemple complet d'utilisation de Xlinkit

La gestion d'une bibliothèque (validation multi-documents)

1. Le fichier **bibliotheque**.xml

(fichier contenant les livres déjà répertoriés)

```
<?xml version="1.0" encoding="UTF-8" ?>
  <bibliothèque>
     <livre>
       <code>1</code>
       <titre>livre1</titre>
       <auteur>auteur1</auteur>
     </livre>
     <livre>
       <code>2</code>
       <titre>livre2</titre>
       <auteur>auteur2</auteur>
     </livre>
     <livre>
       <code>3</code>
       <titre>livre3</titre>
       <auteur>auteur3</auteur>
     </livre>
     <livre>
       <code>4</code>
       <titre>livre4</titre>
       <auteur>auteur4</auteur>
     </livre>
     <livre>
       <code>5</code>
       <titre>livre5</titre>
       <auteur>auteur5</auteur>
     </livre>
     <livre>
       <code>6</code>
       <titre>livre6</titre>
       <auteur>auteur6</auteur>
     </livre>
  </bibliothèque>
```

2. Les fichiers ***livre1***.xml, ***livre2***.xml, ***livre3***.xml

(fichiers contenant les informations sur les nouveaux livres)

livre1.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
     <livre>
       <code>7</code>
       <titre>livre5</titre>
       <auteur>auteur7</auteur>
     </livre>
```

livre2.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
     <livre>
       <code>5</code>
       <titre>livre8</titre>
       <auteur>auteur8</auteur>
     </livre>
```

livre3.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
     <livre>
       <code>1</code>
       <titre>livre5</titre>
       <auteur>auteur9</auteur>
     </livre>
```

3. Les fichiers ***livres***.xml, ***DocumentSet***.xml

(fichiers contenant les informations sur la localisation des données nécessaires au module de validation.)

livres.xml

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE DocumentSet SYSTEM "DocumentSet.dtd">
     <DocumentSet name="livres">
          <Description>Document regroupant l'ensemble des
          nouveaux livres
          </Description>
          <DocFile href="livre1.xml"/>
          <DocFile href="livre2.xml"/>
          <DocFile href="livre3.xml"/>
    </DocumentSet>
```

DocumentSet.xml

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE DocumentSet SYSTEM "DocumentSet.dtd">
     <DocumentSet name="documents">
          <Description>Localisation des différents
          livres</Description>
     <DocFile href="bibliothèque.xml"/>
     <Set href="livres.xml"/>
</DocumentSet>
```

4. Le fichier **rule**.xml

(fichier contenant la règle à valider)

rule.xml

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE consistencyruleset SYSTEM 'consistencyrule.dtd'>
     <consistencyruleset>
          <globalset id="bibli"
          xpath="/bibliothèque/livre"/>
          <globalset id="livre" xpath="/livre"/>
          <consistencyrule id="r1">
          <description>
          Pour tout livre de code x, x doit être plus
          grand que tout code déjà présent dans la
          bibliothèque.
          </description>
          <forall var="a" in="$livre/code[text()]">
               <not>
                    <exists var="c"
                    in="$bibli/code[$a/text() &lt;=
                    text()]"/>
               </not>
          </forall>
     </consistencyrule>
</consistencyruleset>
```

5. Le fichier ***RuleSet.xml***

(fichier contenant la localisation de la règle à valider)

RuleSet.xml

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE RuleSet SYSTEM "RuleSet.dtd">
     <RuleSet name="Bibliorègles">
          <Description>Règles relatives à la
          bibliothèque...</Description>
          <RuleFile href="rule.xml"/>
     </RuleSet>
```

6. Le fichier **result**.xml

(fichier contenant les résultats de la validation)

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE xlinkit:LinkBase SYSTEM
"http://www.xlinkit.com/DTD/LinkBase.dtd">

<?xml-stylesheet type="text/xsl" href="linkbase.xsl"?>
<?cocoon-process type="xslt"?>

<xlinkit:LinkBase date="Wed Dec 19 14:12:11 CET 2001"
    docSet="c:/jakarta-tomcat-3.2.3/webapps/Xlinkit/DocumentSet.xml"
    ruleSet="c:/jakarta-tomcat-3.2.3/webapps/Xlinkit/RuleSet.xml"
    xmlns:xlink="http://www.w3.org/1999/xlink"
    xmlns:xlinkit="http://www.xlinkit.com">
```

ci-dessous : résultats pour le livre1 : OK!

```
<xlinkit:ConsistencyLink
ruleid="rule.xml#/consistencyruleset/consistencyrule[1]">
  <xlinkit:State>consistent</xlinkit:State>
  <xlinkit:Locator xlink:href="livre1.xml#/livre/code[1]"
  xlink:label="" xlink:title=""/>
</xlinkit:ConsistencyLink>
```

résultats pour le livre2 : KO, les codes des livres 5 et 6 de la bibliothèque contredisent la règle !

```
<xlinkit:ConsistencyLink
ruleid="rule.xml#/consistencyruleset/consistencyrule[1]">
  <xlinkit:State>inconsistent</xlinkit:State>
  <xlinkit:Locator xlink:href="livre2.xml#/livre/code[1]"
  xlink:label="" xlink:title=""/>
  <xlinkit:Locator xlink:href="bibliothèque.xml#/
  bibliotheque/livre[5]/code[1]" xlink:label="" xlink:title=""/>
</xlinkit:ConsistencyLink>

<xlinkit:ConsistencyLink
ruleid="rule.xml#/consistencyruleset/consistencyrule[1]">
  <xlinkit:State>inconsistent</xlinkit:State>
  <xlinkit:Locator xlink:href="livre2.xml#/livre/code[1]"
  xlink:label="" xlink:title=""/>
  <xlinkit:Locator xlink:href="bibliothèque.xml#/
  bibliotheque/livre[6]/code[1]" xlink:label="" xlink:title=""/>
</xlinkit:ConsistencyLink>
```

résultats pour le livre3 : KO, les codes des livres 2,3,4,5 et 6 de la bibliothèque contredisent la règle !

```
<xlinkit:ConsistencyLink
ruleid="rule.xml#/consistencyruleset/consistencyrule[1]">
  <xlinkit:State>inconsistent</xlinkit:State>
  <xlinkit:Locator xlink:href="livre3.xml#/livre/code[1]"
  xlink:label="" xlink:title=""/>
  <xlinkit:Locator xlink:href="bibliothèque.xml#/bibliothèque/
  livre[2]/code[1]" xlink:label="" xlink:title=""/>
</xlinkit:ConsistencyLink>

<xlinkit:ConsistencyLink
ruleid="rule.xml#/consistencyruleset/consistencyrule[1]">
  <xlinkit:State>inconsistent</xlinkit:State>
  <xlinkit:Locator xlink:href="livre3.xml#/livre/code[1]"
  xlink:label="" xlink:title=""/>
  <xlinkit:Locator xlink:href="bibliothèque.xml#/bibliotheque/
  livre[3]/code[1]" xlink:label="" xlink:title=""/>
</xlinkit:ConsistencyLink>

<xlinkit:ConsistencyLink
ruleid="rule.xml#/consistencyruleset/consistencyrule[1]">
  <xlinkit:State>inconsistent</xlinkit:State>
  <xlinkit:Locator xlink:href="livre3.xml#/livre/code[1]"
  xlink:label="" xlink:title=""/>
  <xlinkit:Locator xlink:href="bibliothèque.xml#/bibliotheque/
  livre[4]/code[1]" xlink:label="" xlink:title=""/>
</xlinkit:ConsistencyLink>

<xlinkit:ConsistencyLink
ruleid="rule.xml#/consistencyruleset/consistencyrule[1]">
  <xlinkit:State>inconsistent</xlinkit:State>
  <xlinkit:Locator xlink:href="livre3.xml#/livre/code[1]"
  xlink:label="" xlink:title=""/>
  <xlinkit:Locator xlink:href="bibliothèque.xml#/bibliotheque/
  livre[5]/code[1]" xlink:label="" xlink:title=""/>
</xlinkit:ConsistencyLink>

<xlinkit:ConsistencyLink
ruleid="rule.xml#/consistencyruleset/consistencyrule[1]">
  <xlinkit:State>inconsistent</xlinkit:State>
  <xlinkit:Locator xlink:href="livre3.xml#/livre/code[1]"
  xlink:label="" xlink:title=""/>
  <xlinkit:Locator xlink:href="bibliothèque.xml#/bibliotheque/
  livre[6]/code[1]" xlink:label="" xlink:title=""/>
</xlinkit:ConsistencyLink>
</xlinkit:LinkBase>
```

A.3. Les règles complètes de la validation du marché des stages

Règle 1 : Le code postal ne doit contenir que des chiffres !

```
<consistencyrule id="r1">
     <description>
     Le code postal ne doit contenir que des chiffres !
     </description>
     <forall var="a" in="$entr/ADRESSE/CODE_POSTAL">
          <exists var="b" in="$a[normalize-
     space(translate(text(),'1234567890',''))='']"/>
     </forall>
</consistencyrule>
```

Règle 2 : La ville ne peut contenir que des lettres !

```
<consistencyrule id="r2">
     <description>
     La ville ne peut contenir que des lettres !
     </description>
     <forall var="a" in="$entr/ADRESSE/VILLE">
          <exists var="b" in="$a[normalize-
          space(translate(text(),
          'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTU
          VWXYZ',''))='']"/>
     </forall>
</consistencyrule>
```

Règle 3 : Le pays ne peut contenir que des lettres !

```
<consistencyrule id="r3">
     <description>
     Le pays ne peut contenir que des lettres !
     </description>
     <forall var="a" in="$entr/ADRESSE/PAYS">
          <exists var="b" in="$a[normalize-
          space(translate(text(),
          'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTU
          VWXYZ',''))='']"/>
     </forall>
</consistencyrule>
```

Règle 4 : La télécopie ne peut contenir que certains symboles.

```
<consistencyrule id="r4">
     <description>
     La télécopie ne doit contenir que des chiffres (et
     éventuellement + / \ . )!
     </description>
     <forall var="a" in="$entr/NUM_FAX">
          <exists var="b" in="$a[normalize-
          space(translate(text(),'1234567890+/\.',''))='']
          "/>
     </forall>
</consistencyrule>
```

Règle 5 : Le numéro de téléphone ne peut contenir que certains symboles.

```
<consistencyrule id="r5">
     <description>
     Le nr de téléphone ne doit contenir que des chiffres
     (et éventuellement + / \ . )!
     </description>
     <forall var="a" in="$entr/NUM_TEL">
          <exists var="b in="$a[normalize-
          space(translate(text(),'1234567890+/\.',''))='']
          "/>
     </forall>
</consistencyrule>
```

Règle 6 : Le numéro de téléphone doit avoir au moins 8 caractères…

```
<consistencyrule id="r6">
     <description>
     Le nr de téléphone doit avoir au moins 8 caractères…
     </description>
     <forall var="a" in="$entr/NUM_TEL">
          <exists var="b in="$a[string-length(normalize-
          space(text())) > 7]"/>
     </forall>
</consistencyrule>
```

Règle 7 : Le nom du contact ne peut contenir que des lettres.

```
<consistencyrule id="r7">
     <description>
     Le nom du contact ne peut contenir que des lettres !
     </description>
     <forall var="a" in="$contact/NOM_CONTACT">
          <exists var="b" in="$a[normalize-
          space(translate(text(),
          'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTU
          VWXYZ',''))='']"/>
     </forall>
</consistencyrule>
```

Règle 8 : Le prénom du contact ne peut contenir que des lettres.

```
<consistencyrule id="r8">
     <description>
     Le prénom du contact ne peut contenir que des
     lettres !
     </description>
     <forall var="a" in="$contact/PRENOM_CONTACT">
          <exists var="b" in="$a[normalize-
          space(translate(text(),
          'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTU
          VWXYZ',''))='']"/>
     </forall>
</consistencyrule>
```

Règle 9 : L'e-mail d'un contact doit contenir un symbole @.

```
<consistencyrule id="r9">
     <description>
     L'e-mail d'un contact doit contenir un symbole @
     </description>
     <forall var="a" in="$contact/EMAIL_CONTACT">
          <exists var="b" in="$a[contains(text(),'@')]"/>
     </forall>
</consistencyrule>
```

Règle 10 : Le sujet d'un stage ne peut avoir plus de 100 caractères…

```
<consistencyrule id="r10">
     <description>
     Le sujet d'un stage ne peut avoir plus de 100
     caractères…
     </description>
     <forall var="a" in="$stage/SUJET">
          <exists var="b" in="$a[string-length(normalize-
          space(text()) &lt ; 101]"/>
     </forall>
</consistencyrule>
```

Règle 11 : Le descriptif d'un stage ne peut avoir plus de 500 caractères…

```
<consistencyrule id="r11">
     <description>
     Le descriptif d'un stage ne peut avoir plus de 500
     caractères…
     </description>
     <forall var="a" in="$stage/DESCRIPTIF">
          <exists var="b" in="$a[string-length(normalize-
          space(text()) &lt ; 501]"/>
     </forall>
</consistencyrule>
```

Règle 12 : Le pays d'un stage ne peut contenir que des lettres !

```
<consistencyrule id="r12">
     <description>
     Le pays d'un stage ne peut contenir que des lettres !
     </description>
     <forall var="a" in="$stage/LIEU/PAYS">
          <exists var="b" in="$a[normalize-
          space(translate(text(),
          'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTU
          VWXYZ',''))='']"/>
     </forall>
</consistencyrule>
```

Règle 13 : La ville d'un stage ne peut contenir que des lettres !

```
<consistencyrule id="r13">
     <description>
     La ville d'un stage ne peut contenir que des
     lettres !
     </description>
     <forall var="a" in="$stage/LIEU/VILLE">
          <exists var="b" in="$a[normalize-
          space(translate(text(),
          'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTU
          VWXYZ',''))='']"/>
     </forall>
</consistencyrule>
```

Règle 14 : L'année du stage doit être supérieure à 1999 et inférieure à 2002

```
<consistencyrule id="r14">
     <description>
     L'année du stage doit être supérieure à 1999 et
     inférieure a 2002
     </description>
     <forall var="a" in="$stage/DATES/VDATE_DEBUT/ANNEE">
          <exists var="b" in="$a[text()>1999 and
          2002>text()]"/>
     </forall>
</consistencyrule>
```

Règle 15 : La date de début du stage minimum doit être inférieure à sa date de début maximum.

```
<consistencyrule id="r15">
     <description>
     La date de début du stage minimum doit être
     inférieure à sa date de début maximum
     </description>
     <forall var="a" in="$stage/DATES/VDATE_DEBUT/ANNEE">
          <exists var="b"
          in="$a/../../DATE_DEBUT_MAX/ANNEE[text()>$a/text
          ()]"/>
     </forall>
</consistencyrule>
```