Université de Technologie de Troyes
DEA RACOR

Rapport de stage

Présenté en septembre 2003

par

Julien Baudry

Passerelles entre les formalismes XML Schema et la programmation par objet (JAVA)

Entreprise	INRIA
Lieu	Sophia Antipolis (06)
Date	Printemps 2003
Etudiant	Julien Baudry (UTT / GSID 6 & DEA RACOR)
Responsable entreprise	Didier Parigot (INRIA / OASIS)
Responsable UTT	Marc Lemercier (UTT / LM2S)

Remerciements

Ce rapport reflète le travail de six mois de stage au sein de l'INRIA.

Je remercie Didier Parigot pour m'avoir accueilli dans son service et m'avoir permis de réaliser ce stage. De plus, je lui suis très reconnaissant de m'avoir fait partager ses connaissances, sa passion pour la recherche, et pour la confiance qu'il m'a accordée au cours de ce stage.

Enfin, je remercie l'ensemble du personnel de l'INRIA impliqué dans le projet OASIS, merci à tous d'avoir fait de ce stage une période clé de ma vie étudiante.

Liste des acronymes

AOP	Aspect-Oriented Programming
API	Application Programming Interface
AST	Abstract Syntax Tree
BML	Bean Markup Language
BSML	Bioinformatic Sequence Markup Language
CCM	CORBA Component Model
CML	Chemical Markup Language
CORBA	Common Object Request Broker Architecture
CSS	Cascading Style Sheet
DDML	Document Definition Markup Language
DOM	Document Object Model
DSL	Domain Specific Language
DTD	Document Type Definition
EJB	Entreprise JavaBeans
HTML	Hyper Text Markup Language
MDA	Model-Driven Architecture
MathML	Mathematical Markup Language
OMG	Object Management Group
OMT	Object Modeling Technique
OOSE	Object-Oriented Software Engineering
PDA	Personal Digital Assistant
PIM	Plateform-Independant Model
PSM	Platform-Specific Model
RDF	Resource Description Framework Schema
RELAX	Regular Language Description for XML
RNTL	Réseau National de recherche et d'innovation en Technologies Logicielles
SAX	Simple API for XML
SGBD	Système de Gestion de Base de Données
SGML	Standard Generalized Markup Language
SMIL	Synchronized Multimedia Integration Language
SOAP	Simple Object Access Protocol
SOX	Schema for Object-oriented XML
SVG	Scalable Vector Graphics
TREX	Tree Regular Expression for XML
UML	Unified Modeling Language
W3C	World Wide Web Consortium
XML	Extensible Markup Language
XPath	XML Path language
XSLT	Extensible Stylesheet Language Transformation

Sommaire

1. Introduction

La qualité du logiciel et sa capacité à évoluer, ainsi que la rapidité du développement, sont des soucis majeurs pour les industriels. Un logiciel bien conçu doit pouvoir s'adapter rapidement aux demandes des clients et aux nouvelles technologies pour pouvoir lutter contre la concurrence. Il doit aussi être capable d'échanger des données très variées avec d'autres applications, particulièrement depuis l'avènement d'Internet. En quelques années, l'informatique confinée aux domaines scientifiques s'est démocratisée. Ses utilisateurs ont maintenant des besoins, des connaissances et des domaines d'activité différents. De plus, la pression du marché impose des temps de développement de logiciel plus courts et des coûts plus faibles. Cette évolution a bouleversé la manière de concevoir et de réaliser les logiciels. Il n'est plus possible de les développer entièrement sauf dans les domaines tels que la défense ou les transports où la sûreté de l'ensemble du code doit être vérifiée. Les technologies propriétaires sont à bannir car elles isolent les logiciels et freinent leurs évolutions. Les exigences vis-à-vis des logiciels ont aussi été modifiées à cause des disparités de connaissances des utilisateurs et des programmeurs, des contraintes de temps et d'argent, et des nécessités d'adaptation rapide aux besoins du marché. Les logiciels doivent satisfaire les exigences suivantes :

- **conviviaux** grâce à une interface utilisateur interactive ;
- **faciles** à **utiliser** avec peu de compétences informatiques et basés sur des techniques bien connues ou des standards ;
- **ouverts** grâce à un format d'échange de données standard utilisé pour communiquer entre les composants et avec les applications externes ;
- à implémentation **modulaire** et **flexible**, basée sur des composants génériques et réutilisables.

Pour prendre en compte ces bouleversements, de nouveaux standards pour l'échange de données et de nouvelles techniques de programmation ont émergé.

1.1. Contexte technologique

Tout d'abord, avec l'effervescence liée à la naissance du Web, il y a eu une volonté de standardiser les langages et les protocoles, ce qui a conduit à la création du W3C. Ce consortium a proposé un nouveau format de données, XML (Extensible Markup Language), issu de SGML (Standard Generalized Markup Language) afin de simplifier et d'uniformiser les échanges d'informations entre applications, indépendamment de tout langage et plate-forme.
Aujourd'hui, on constate que de nombreux langages métiers sont créés dans des domaines d'activité très variés tels que les mathématiques (MathML), la chimie (CML - Chemical Markup Language),

le commerce (CommerceXML), les applications sans fils (WML -Wireless Markup Language) …etc. Ces langages métiers ont de plus en plus tendance à être définis grâce à un formalisme de modélisation de document appelé Schema XML. Les Schemas XML permettent de définir le vocabulaire qui sera employé dans le langage ainsi que sa grammaire (règles d'écriture).
De nouvelles techniques ont aussi émergé en génie logiciel, notamment avec la programmation par objets et ses notions d'encapsulation et d'héritage propices à la modularité, la réutilisation et l'extensibilité du code. Puis, d'autres techniques sont venues s'ajouter pour rendre ce type de programmation plus efficace, comme les patrons de conception, la programmation par aspects, la programmation générative et la programmation par modèle (UML).
Au croisement de ces évolutions technologiques, SmartTools prend tout son sens dans le développement d'applications communicantes (traitement et échanges d'informations). Avec la seule description d'un langage métier, la plateforme va générer un environnement de développement à base de composants logiciels. Le développeur pourra se concentrer sur la valeur ajoutée de son application car les outils de bases qui lui permettent de travailler (manipuler, effectuer des traitements) avec son langage auront été générés auparavant.

1.2. Objectifs du stage

Le principal objectif du stage est d'essayer d'ouvrir SmartTools aux Schemas XML, afin qu'il puisse traiter de nouveaux langages métiers et donc s'ouvrir à une nouvelle communauté d'utilisateurs. Pour réussir cette ouverture, toute la problématique est de trouver des règles de correspondances entre le formalisme des Schemas XML et la programmation par objets, technologies utilisées à l'intérieur de SmartTools.

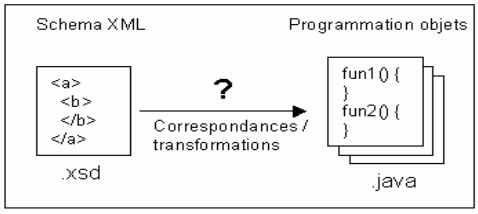


Figure 1 : Liens entre un Schema XML et les structures objets

Le travail pourra s'appuyer sur la première expérience de l'équipe qui a élaborée des passerelles entre les DTD et des structures objets (Java) pour proposer ou un ensemble de contraintes ou un ensemble de transformation pour réaliser cette correspondance dans le cadre particulier des Schemas XML. L'objectif du stage sera d'identifier plus précisément le sous-ensemble des Schemas qui permet une translation aisée vers des structures Java, sans perte d'information. Ainsi les traitements sémantiques pourront utiliser toute la puissance d'expression d'un langage de programmation par objet comme Java. Le prototype réalisé utilisera fortement les briques de bases déjà réalisées dans SmartTools.
Enfin, il faudra proposer et envisager des extensions à cette transformation de base.

A travers ce rapport nous essaierons d'apporter une réponse aux différentes problématiques soulevées par le sujet. Pour cela, dans un premier temps nous vous expliquerons le rôle de la plateforme logicielle « SmartTools » qui est au cœur du projet. Puis, nous vous présenterons les différentes technologies associées au sujet ou qui ont contribuées à atteindre les objectifs. Enfin, nous vous ferons part des solutions retenues et du travail effectué pendant ce stage.

2. Présentation générale de SmartTools

Ce chapitre permet d'avoir une vue d'ensemble de SmartTools. Il introduit brièvement quelques concepts utilisés pour sa construction et son utilisation. Ce chapitre se décompose en deux sections, la première décrit de façon générale SmartTools et la deuxième partie explique plus précisément son fonctionnement.

2.1. Description

La qualité, l'évolutivité du logiciel et la rapidité du développement sont devenues des soucis majeurs dans le développement d'applications informatiques. Le principal intérêt des efforts de spécifications du W3C est de proposer des spécifications utilisées par tous, par opposition à des formats propriétaires. Mais cet effort (en particulier le concept de DTD ou de Schema), donne surtout les moyens de décrire formellement les informations échangées. Ainsi tous les types d'informations peuvent être structurés et traités avec une approche comparable à celle utilisée pour la définition de langages de programmation. Par opposition à ces langages de programmation, ces langages, dits métiers, sont très variés, et liés à un domaine d'application (telecom mais aussi finance, assurance ou transport). Leur syntaxe et leur sémantique (les traitements associés) sont certainement beaucoup plus simples que pour des langages de programmation généralistes. D'autre part, les concepteurs et les utilisateurs de ces langages n'ont pas forcément de compétences approfondies sur les techniques liées aux langages (analyse, compilation, interprétation, etc …). Enfin la rapidité du développement, les possibilités d'intégration mais aussi la facilité d'utilisation et d'affichage multi supports sont des caractéristiques importantes pour ce type d'activités.

Les objectifs de la plateforme SmartTools s'inscrivent dans cette nouvelle problématique associée à la conception rapide de langages métiers pour l'échange et/ou le traitement d'informations. SmartTools est un générateur de composants basé sur les formalismes du W3C, qui offre un environnement pour manipuler et exploiter les langages métiers créés.

2.2. Fonctionnement

Dans cette partie nous allons expliquer pourquoi SmartTools est utile pour le développement d'applications basé sur des langages métiers. Avec une seule description (la DTD ou le Schema du

langage), la plate-forme permet de générer un environnement de développement qui contient principalement les composants logiciels suivants :

1. un **analyseur** d'une forme concrète du langage
2. un **formateur**
3. un **éditeur** syntaxique
4. un ensemble de **sources Java** pour faciliter l'écriture de traitements (analyses, transformations) ; la technique utilisée est celle des "visitor design patterns".
5. un ensemble de **vues génériques** de l'arbre (textuelle et graphique).

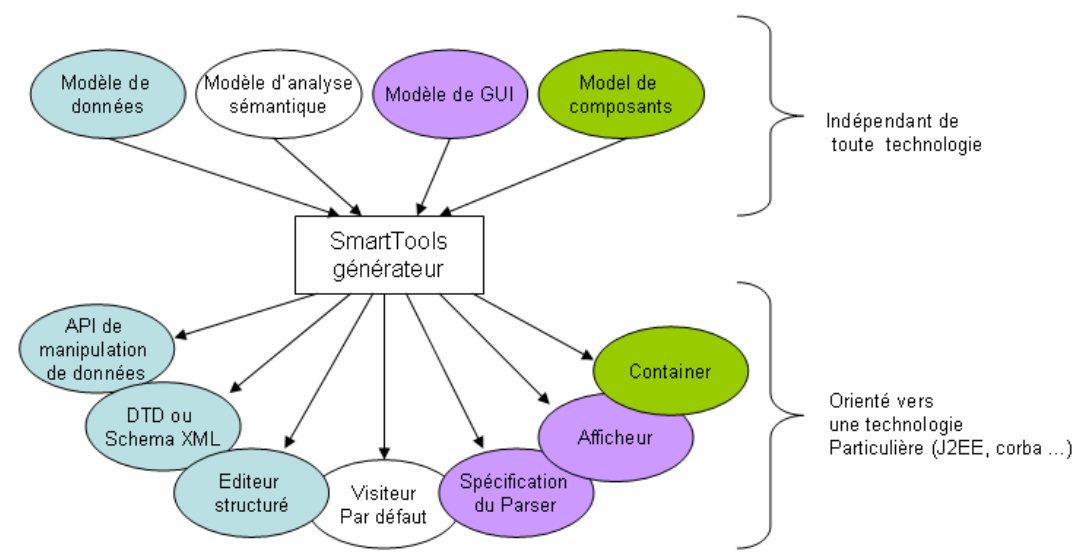


Figure 2 : Vue d'ensemble de la programmation générative mise place dans SmartTools

L'originalité et l'innovation de SmartTools peuvent être résumées en quatre points :

1. Accepter en entrée des formalismes du W3C (DTD et Schema), ce qui permet de profiter des nombreux développements autour des standards du W3C. L'innovation consiste à proposer pour des documents XML, une méthodologie de programmation (pour décrire les traitements sémantiques) fondée sur les travaux autour des "design patterns", issus de la programmation par objets.

2. Fournir une interface utilisateur conviviale. SmartTools a l'avantage de traiter tous ces aspects d'affichage sur un même modèle. Cela permet de proposer une approche homogène et uniforme avec un fort potentiel de réutilisation tant pour SmartTools que pour les environnements produits. Un autre avantage important est de pouvoir exporter les vues vers d'autres supports comme des browsers.

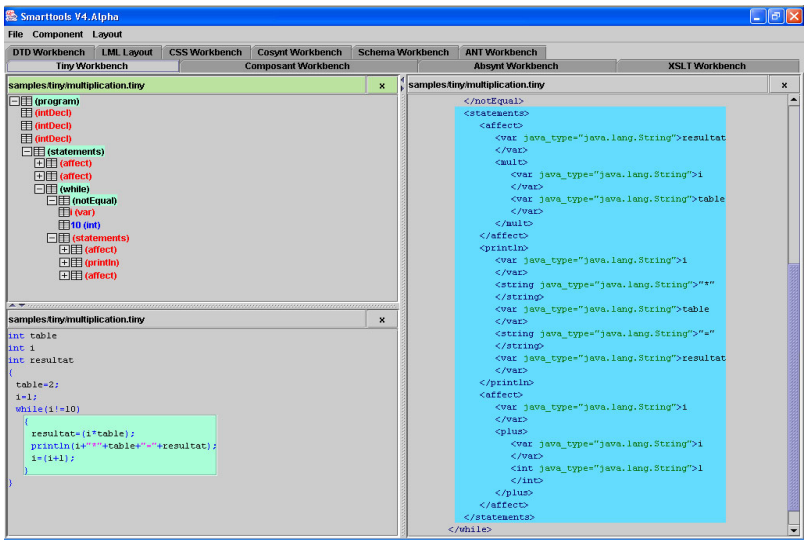


Figure 3 : L'interface graphique de SmartTools et ses différentes vues associées.

3. Pour assurer une bonne évolution de l'outil, il était nécessaire de concevoir dès le début une architecture logicielle modulaire (par composants indépendants) et extensible.

4. Pour plus de flexibilité, SmartTools utilise les techniques de programmation par aspects avec la technique de programmation par visiteurs, ce qui évite des modifications du code source. Cette technique dynamique est très simple à mettre en œuvre, et elle s'applique très bien dans le cadre d'applications d'e-commerce par exemple pour traiter les problèmes de reconfiguration, d'adaptation et de sécurité.

L'intérêt de l'approche avec SmartTools est de rendre accessible et facilement utilisable (grâce à la génération de code) des techniques avancées de programmation initialement développées pour les langages de programmation. En particulier les nouvelles approches de programmation adaptive, par aspects et par composants qui sont très utilisées et qui semblent être des atouts en terme de modularité et réutilisation de composants.
Le schéma ci-dessous montre le travail à effectuer par le développeur pour réaliser une application et l'aide fournit par SmartTools.

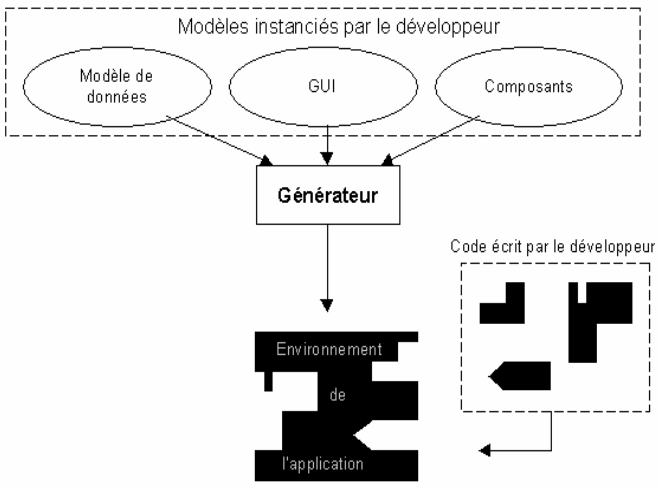


Figure 4 : modèles instanciés + génération environnement de l'application + code métier = une application

3. Aperçu des technologies

L'objectif de ce chapitre est de présenter brièvement les technologies utilisées dans SmartTools. Tout d'abord, nous présenterons XML de façon générale, puis le concept de modèle de document avec la présentation des DTD et des Schema XML. Puis, nous présenterons le formalisme utilisé dans SmartTools : AbSynt (Syntaxe Abstraite). Enfin, nous vous proposerons un aperçu des projets qui proposent d'autres solutions sur la problématique du passage de la technologie XML vers JAVA.

3.1. Introduction à XML

XML (eXtensible Markup Langage) est un langage à balise extensible créé en 1996 par le W3C (World Wide Web Consortium) comme langage pivot à tout échange d'information entre applications. Il s'agit d'un sous-ensemble de la norme SGML (Standard Generalized Markup Langage ou ISO 8879), utile pour décrire des documents mais pas adapté aux besoins engendrés par le Web. Bien qu'aussi flexible que ce méta-langage et avec la meme puissance d'expression, il est plus simple d'usage et donc aussi bien accepté que HTML (Hyper Text Markup Langage) qui lui, n'est qu'une instance de SGML. XML est un langage universel, il est aussi bien utilisé dans le monde des documents – le Web qui était son but initial, la documentation (ex : documentation technique) et la bureautique – que dans le monde des données – les SGBDs (Système de Gestion de base de Données), la gestion et les applications scientifiques.

Contrairement à HTML, XML est extensible (balise, tags non fixés), et peut donc définir une structure de données complexe (plusieurs niveaux d'imbrications). De plus, la conformité structurelle de ses documents peut être validée. Ses utilisateurs peuvent choisir librement les attributs et les noms de balises qui qualifient sémantiquement et non graphiquement leurs données. Les balises ne correspondent plus à des informations de présentation dans le navigateur de l'utilisateur mais donnent un sens à l'information contenue entre ces balises. Cette séparation entre les données/documents et les traitements/présentations donne une plus grande réutilisabilité et longévité aux données. Une des notions importantes en XML est celle du ***document bien formé***.

▪ Document bien formé

XML nous donne la possibilité de choisir nos propres types d'éléments, et d'inventer nos propres grammaires, pour créer des langages adaptés à nos besoins. Mais cette flexibilité pourrait être dangereuse pour les parseurs XML, s'ils ne disposaient pas de règles syntaxiques minimales. Un

parseur dédié à un unique langage tel que HTML peut tolérer un certain laxisme dans le balisage, parce que l'ensemble d'éléments est de petite taille et qu'une page Web n'est pas très complexe. Mais comme les processeurs XML doivent se préparer à tous les types de langages, un ensemble de règles de base est nécessaire.
Ces règles sont des contraintes syntaxiques très simples. Toutes les balises doivent utiliser des délimiteurs corrects, une balise de fin doit suivre toute balise de début, les éléments ne peuvent pas se chevaucher, … etc. Les documents qui respectent ces règles sont dits ***bien formés***.

Certaines de ces règles sont listées ci-dessous :

▪ La première de ces règles est qu'un élément contenant du texte notamment, doit avoir une balise de début et une balise de fin.

Juste

```
<list>
    <listitem>arbre</listitem>
    <listitem>table</listitem>
</list>
```

Faux

```
<list>
    <listitem>arbre
    <listitem>table
<list>
```

▪ La balise d'un élément vide doit avoir une barre oblique (/) avant le chevron fermant.

Juste

```
<graphique fichier="icon.png"/>
```

Faux

```
<graphique fichier="icon.png">
```

▪ Toutes les valeurs d'attributs doivent être entourées de guillemets

Juste

```
<graphique fichier="icon.png"/>
```

Faux

```
<graphique fichier=icon.png/>
```

Les règles ci-dessus ne sont pas exhaustives, il en existe beaucoup d'autres. Pour consulter la liste complète vous pouvez vous reporter aux spécification XLM Schema sur le site du W3C [7] .

3.2. Les modèles de documents

3.2.1. Présentation du concept

L'une des caractéristiques les plus puissantes du langage XML est qu'il nous permet de créer notre propre langage de balisage, ainsi que de définir les éléments et les attributs qui correspondent à l'information que nous voulons représenter, plutôt que de nous limiter à un langage général, prédéfini et mal adapté. Mais ce qui nous manque encore est un moyen de définir un langage de manière formelle, pour pouvoir restreindre le vocabulaire utilisable et la grammaire du langage. Ce processus de définition formelle d'un langage en XML s'appelle : la ***modélisation de document***. Dans les sections suivantes, nous allons décrire deux façons de modéliser un document : les définitions de type de documents (DTD), qui décrivent la structure du document à l'aide de règles déclaratives, et les Schemas XML, qui détaillent la structure d'un document à l'aide de motifs d'éléments.

Qu'entend-on par un "modèle de document" ? Il définit les documents qui peuvent être produits dans un langage. Pour utiliser la terminologie XML, le modèle de document précise quels documents sont conformes au langage. Un modèle de document permet de répondre à des questions telles que "Cet élément peut-il avoir un titre ?" ou encore "Doit-il y avoir un prix sur cet élément ?".

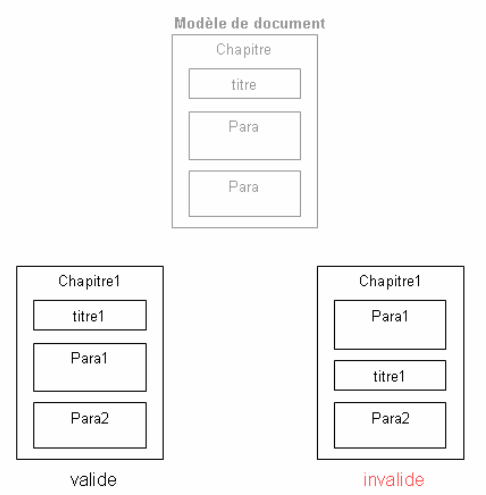


Figure 5 : Notion de documents valide ou invalide avec XML

3.2.2. Introduction aux DTD

Le type de modèles de documents le plus utilisé est la DTD (Définition de Type de Documents). Les DTD sont en fait antérieures à XML, puisqu'elles sont un héritage de SGML dont la syntaxe a été conservée presque intacte. Une DTD définit un type de documents de la façon suivante :

- Elle déclare un ensemble d'éléments autorisés. On ne peut pas utiliser d'autres éléments que ceux de cet ensemble. Il faut imaginer cet ensemble comme étant le « vocabulaire » du langage.
- Elle définit un modèle de contenu pour chaque élément. Le modèle de contenu est un motif qui indique quels éléments ou quelles données peuvent apparaître à l'intérieur d'un élément, dans quel ordre, dans quelle quantité, et si ce contenu est optionnel ou requis. Il faut imaginer l'ensemble de ces contraintes comme étant la « grammaire » du langage.
- Elle déclare un ensemble d'attributs autorisés pour chaque élément. Chaque déclaration d'attribut définit un nom, un type de données, une valeur par défaut (éventuellement), et un comportement (par exemple l'attribut est requis ou optionnel).
- Elle a aussi la possibilité d'importer certaines parties d'un fichier externe.

ⓘ Vous pouvez consulter l'Annexe 2 pour voir un exemple d'utilisation d'une DTD et visualiser une instance de fichier XML conforme à cette DTD.

3.2.3. Présentation des Schemas XML

L'objectif de cette section est de comprendre le principe général d'écriture d'un Schema et aussi de mettre en relief les différences par rapport aux DTD.

Certains se sont plaints que les DTD utilisent une syntaxe inflexible et qu'elles ne sont pas suffisamment expressives pour leurs besoins. De plus les DTD respectent un format particulier et XML un autre, les modèles de contenu des éléments et les listes d'attributs sont difficiles à lire et à comprendre, et il est frustrant de ne pas pouvoir préciser plus de contraintes sur leurs contenus. C'est pour toutes ces raisons que le W3C à développé un nouveau standard en matière de modélisation de document : les Schemas XML.
Contrairement à la syntaxe des DTD, la syntaxe de Schema XML relève du XML bien formé, ce qui rend possible l'utilisation d'un éditeur XML pour les éditer. Il fournit également un contrôle beaucoup plus strict sur les types de données et les motifs de texte, ce qui en fait un langage très attractif pour assurer une cohérence très stricte des données.

Voici quelques éléments fondamentaux des Schema XML :

▪ **Déclaration des éléments et des attributs**
La définition d'un Schema XML se réalise par l'intermédiaire de plusieurs balises « éléments » chargés de représenter l'arborescence et les informations d'un document XML.

```
<xsd:element name="expéditeur" type="adresseFrance"/>
```

▪ **Types d'éléments simples et complexes**
L'élément « complexType » définit un type de données complexe pour des éléments XML. Un type de données complexe peut être composé d'autres éléments, attributs, group … etc.

```
<xsd:complexType name="USAddress">
  <xsd:sequence>
   <xsd:element name="name"   type="xsd:string"/>
   <xsd:element name="street" type="xsd:string"/>
   <xsd:element name="city"   type="xsd:string"/>
   <xsd:element name="state"  type="xsd:string"/>
   <xsd:element name="zip"    type="xsd:decimal"/>
  </xsd:sequence>
  <xsd:attribute name="country" type="xsd:NMTOKEN"
     fixed="US"/>
</xsd:complexType>
```

▪ **Restriction sur le modèle de contenu**
Si une suite d'éléments simples ne fournit pas suffisamment d'informations, XML Schema propose d'autres possibilités. Par exemple, les attributs minOccurs et maxOccurs fixent le nombre de fois ou un élément peut apparaître. Dans l'exemple ci-dessous, MaxExclusive correspond à la borne supérieure de l'entier (« positiveInteger »).

```
<xsd:simpleType>
  <xsd:restriction base="xsd:positiveInteger">
    <xsd:maxExclusive value="100"/>
  </xsd:restriction>
</xsd:simpleType>
```

▪ **Type de données**
Toute déclaration d'élément et d'attribut a un attribut « type », comme dans la première déclaration. Certains types sont prédéfinis par XML Schema, comme String et Décimal. Le type String représente les données ordinaires, comme le type CDATA dans les DTD. Le type décimal représente les nombres. Si l'on veut déclarer un « âge », on peut déclarer un élément de type « integer » et restreindre sa valeur à toutes les valeurs positives.

▪ **Facettes**
Les facettes sont des propriétés utilisées pour préciser le type de données, en imposant par exemple des limites sur les valeurs des données. Si on définissait une balise <age> dont le type serait « integer-positive », on pourrait limiter sa valeur à 200, grâce à la facette « max-inclusive ». Il existe 13 autres facettes dans les XML Schema.

ⓘ Vous pouvez consulter l'Annexe 1 pour voir un exemple d'utilisation d'un Schema XML et visualiser une instance de ce Schema dans le cadre de la création d'un bon de commande.
ⓘ Pour plus d'information sur la spécification des Schemas XML, vous pouvez consulter le site du W3C et les ressources [15] [16] [17] .

Après avoir vu les éléments les plus importants permettant d'écrire un Schema XML, nous allons découvrir le formalisme utilisé dans SmartTools : **AbSynt** (**Ab**stract **Synt**axe). En effet, il est important d'avoir une vue sur les deux formalismes pour comprendre le passage de l'un à l'autre.

3.3. AbSynt dans SmartTools

Dans la communauté des langages de programmation, la notion d'arbre de syntaxe abstraite est usuellement employée pour représenter un programme en mémoire. Il existe une multitude de formalismes pour décrire cette notion, souvent liée à un langage de programmation sous-jacent. C'est la raison pour laquelle ces formalismes présentent généralement de nombreuses similitudes.
Dans cette section nous allons expliquer pourquoi un langage d'arbre de syntaxe abstraite nommé AbSynt, a été utilisé dans SmartTools et quelles sont ses caractéristiques techniques.

3.3.1. Pourquoi utiliser le formalisme AbSynt

Dans cette section nous allons énoncer les principes de la démarche et expliquer pourquoi un modèle de données propre à SmartTools a été créé :

1. Avoir un formalisme de description de structure de haut niveau adapté à aux besoins de la plateforme et indépendant de toute technologie et langage d'implémentation. Ce modèle abstrait est la clé de voûte de SmartTools, utilisé par tous ses outils. A partir d'un tel modèle abstrait de données d'un langage, on peut générer le code nécessaire à la construction des arbres de syntaxe abstraite.

2. Établir des passerelles avec les formalismes de description de documents standardisés du W3C afin d'obtenir un outil ouvert. De cette façon, l'outil accepte en entrée, non seulement son format propriétaire de description de structure, mais aussi presque toutes les DTDs et tous les XML Schemas. Ces derniers sont convertis dans le modèle pivot (AbSynt) avec, parfois, quelques pertes d'information de structure. De plus, tout modèle abstrait peut aussi être traduit en DTD ou XML Schema. Ces passerelles ouvrent de nouveaux champs d'applications avec les langages métiers définis avec ces formalismes et facilitent aussi leur création.

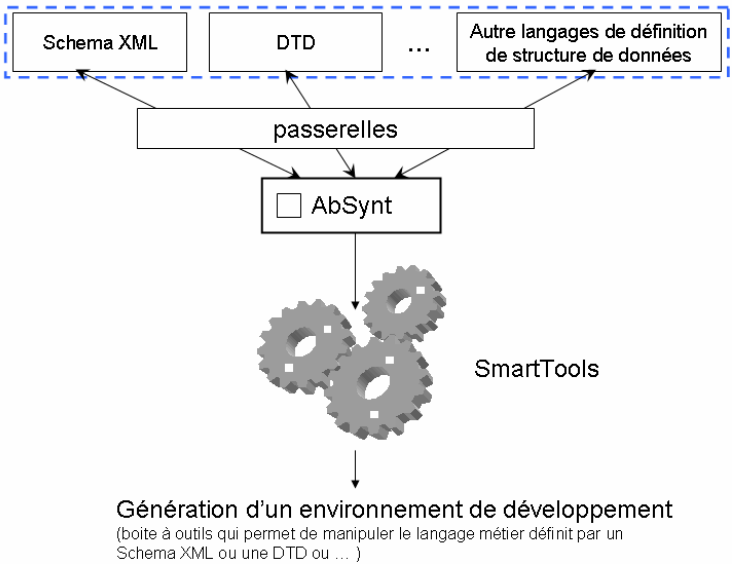


Figure 6 : Place d'AbSynt dans SmartTools par rapport aux langages de modélisation de documents

Afin d'accepter un plus large sous-ensemble des langages définis avec les formalismes du W3C, notre modèle de données (AbSynt) a dû être étendu (possibilité de définir des constructeurs ayant des fils optionnels ou tableaux à n'importe quel emplacement). Ces spécificités n'existent pas dans les autres formalismes.

3. Avoir des interfaces de programmation, pour manipuler les arbres, indépendantes de l'implémentation adoptée pour le représenter. Cette surcouche de méthodes d'accès et de modifications (get et set) est générée en fonction des informations contenues dans le modèle. De cette manière, l'implémentation des arbres peut être changée sans qu'aucun des traitements de l'utilisateur ne soit modifié. A partir d'une DTD, d'un XML Schema ou d'une spécification AbSynt, l'outil génère les classes Java (l'API) associées à ce langage, utiles pour représenter ses documents/programmes en mémoire. De plus il est possible d'effectuer la transformation inverse, passer de classes JAVA vers un Schema XML. Ces possibilités sont illustrées par la figure ci-dessous.

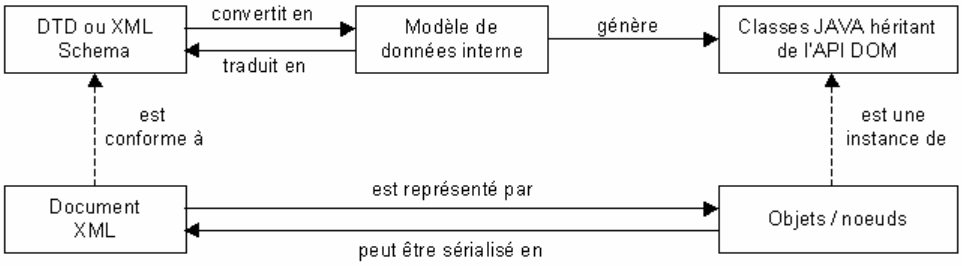


Figure 7 : Correspondance entre un document XML et les objets Java

A partir d'une spécification AbSynt, on peut générer, comme le montre la figure ci-dessous, l'API du langage, la DTD ou le XML Schema correspondant et des visiteurs de base (utilisés pour parcourir le langage métier), ces derniers peuvent être personnalisés. Les arbres de syntaxe abstraite peuvent aussi être affichés avec une forme plus lisible. Pour cela SmartTools propose une spécification qui permet de générer des afficheurs écrits à l'aide de transformations XSLT.

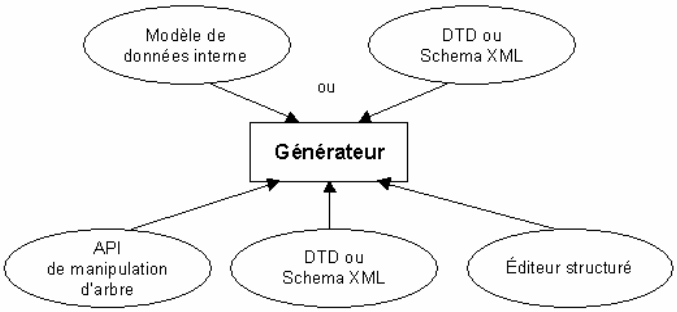


Figure 8 : Spécifications générées à partir d'une spécification AbSynt d'un Schema XML ou d'une DTD.

3.3.2. Description

Cette section décrit le langage AbSynt, l'implémentation des arbres manipulés, les passerelles réalisées pour importer d'autres formats de définition d'arbre de syntaxe abstraite et enfin les différents outils générés.

▪ Notions de base : constructeur, type et attribut

Comme les autres langages de définition de syntaxe abstraite, AbSynt possède les notions de constructeur (opérateur), de type et d'attribut (annotation). Les constructeurs représentent les nœuds des arbres et les attributs les décorations des nœuds. Afin de représenter les structures de documents XML, notre langage a été étendu avec les notions de fils optionnels et de fils tableaux (liste de

nœuds d'un même type). Les types, regroupant des constructeurs en ensembles, indiquent les différents constructeurs possibles des fils. AbSynt contient aussi des définitions de données supplémentaires, utiles pour les calculs sémantiques. Une définition d'un langage en AbSynt se décompose donc en trois parties :

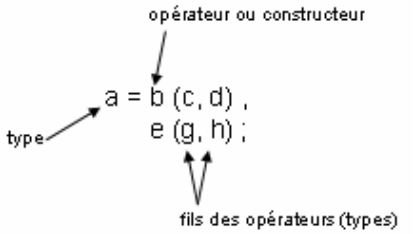


Figure 9 : Motif de base du formalisme AbSynt

▪ **Définitions de types et de constructeurs**

Cette partie contient les définitions de types et de constructeurs. Il existe deux catégories de constructeurs :

– les constructeurs atomiques qui correspondent aux feuilles de l'arbre (représenté par *opAtomique* dans la portion de code ci dessous).

– les constructeurs ayant des fils (par exemple *opVariable*).

Ces constructeurs sont regroupés en ensembles, les types, pouvant aussi contenir d'autres types (types inclus). Les constructeurs non atomiques ont des fils typés et nommés afin de faciliter la manipulation des sous arbres. Pour les feuilles, les seuls types possibles sont String et les types de base de Java.

```
Formalism of monLangage is
Root is Type;
Operator and type definitions {
Type =   opAtomique as java.lang.String,
         opVariable(Type1? filsOptionnel, Type2[] filsTableau, Type3 filsRequis),
         %TypeInclus ;
TypeInclus = opVariable2() ;
}
```

ⓘ Pour en savoir davantage sur le langage AbSynt, veuillez vous reporter à l'annexe 3 « Complément d'information sur le formalisme / langage AbSynt ».

Exemple : illustration simple montrant l'utilisation du langage AbSynt.

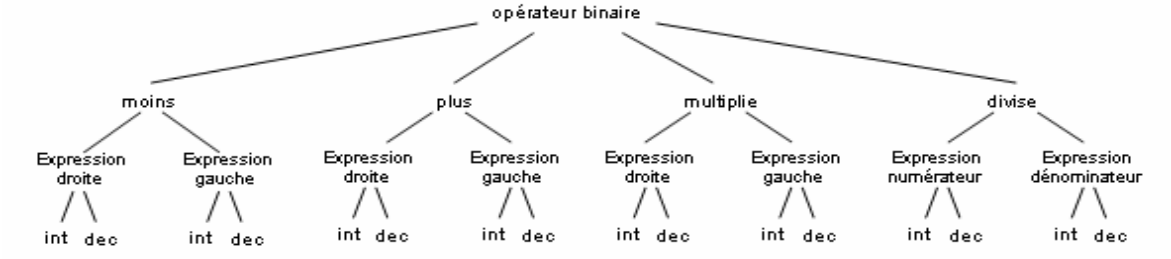
Cet exemple modélise un petit langage d'édition de formules mathématiques (plus, moins, multiplié, divisé).

Spécification partielle du langage en AbSynt

```
OperateurBinaire =   plus (Expression gauche, Expression droite) ,
                     moins (Expression gauche, Expression droite) ,
                     multiplie (Expression gauche, Expression droite) ,
                     divise (Expression numérateur, Expression dénominat) ;

Expression =   int as java.lang.integer ,
               dec as java.lang.decimal ;
```

Représentation partielle du langage sous forme d'arbre



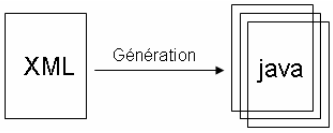
➡ AbSynt est un formalisme propriétaire à SmartTools, qui permet de représenter la structure de données de modèles de documents sous forme d'arbre abstrait. La notion « d'arbre abstrait » signifie qu'AbSynt est indépendant de toute technologie. Enfin, le principal avantage d'AbSynt est d'être adapté aux besoins de SmartTools dans la manipulation de structures de données assimilables à des arbres.

3.4. Technologies permettant la liaison : XML-JAVA

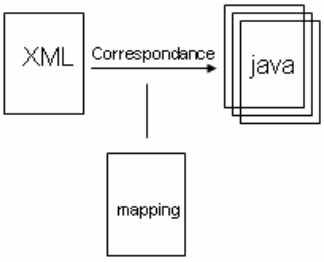
Dans SmartTools, l'utilisateur écrit un Schema XML qui est convertit d'abord dans un formalisme pivot AbSynt, ce dernier génère ensuite les classes Java (cf. Figure 7)D'autres projets existent et traitent avec plus ou moins d'exhaustivité la problématique du passage d'une structure XML à JAVA.
Voici une liste des principaux projets répertoriés :

- **JAXB** de sun (Java Architecture XML Binding) http://java.sun.com/xml/jaxb/ [9]
- **JIXB** de Sonoski software Solution http://jibx.sourceforge.net/ [10]
- **Castor** de Exolab http://www.castor.org/ [11]
- **JBind** de Apache Software http://sourceforge.net/projects/jbind/ [12]
- **Quick** de JXQuick http://sourceforge.net/projects/jxquick/ [13]
- **Zeus** de Zeus project http://zeus.enhydra.org/ [14]

Parmi ces projets on peut distinguer deux approches différentes pour traiter la relation entre les données XML et les structures objets avec JAVA : le binding et le mapping.



Binding [JAXB, JIXB, JBind, Castor]



Mapping [Quick, Zeus]

Le *binding* consiste en la génération de classes java directement à partir d'une définition de la structure de données en XML (Schema ou DTD). Le *mapping* a une approche différente, on part du principe que les structures XML et JAVA existent déjà, la technique consiste à créer un fichier qui fait la correspondance entre les deux structures, ainsi c'est ce fichier de mapping qui permet de sérialiser des objets java en XML ou de faire la transformation inverse. Ces deux techniques sont complémentaires, elles ont chacune leurs avantages et inconvénients.
Les projets vus précédemment se distinguent de part leur approche (mapping vs binding) mais aussi par leur complexité, souplesse d'intégration, avancement technologique, possibilité d'utilisation (licence GPL, commercial) et par leur environnement d'exécution.

▪ Pourquoi développer notre propre passerelle Schema XML–JAVA ?
Nous avons choisi de développer notre propre passerelle car pour le moment aucune technologie ne répond complètement à nos besoins. Cela signifie que nous sommes ouverts à toutes technologies qui répondraient à nos besoins, qui utiliseraient la génération de code (binding) et qui seraient basées sur des standards ouverts.
Actuellement, le projet JAXB de Sun est le plus prometteur et le plus proche des besoins de SmartTools car il bénéficie de l'expertise de Sun avec JAVA, d'un support rigoureux, d'une

ouverture vers d'autres standards de Sun (J2EE) et du W3C (WebServices). Par conséquent c'est une technologie à surveiller, l'évolution de ses spécifications doit être analysée régulièrement.
Le développement de notre propre solution logicielle n'est pas une perte de temps, c'est un contraire une étape riche d'enseignement :
- Tout d'abord notre passerelle Schema XML – Java nous rend service dans l'immédiat et répond à nos besoins.
- Ce travail nous confronte directement aux problèmes et nous permet de mieux les comprendre et d'avoir une vision sur les solutions futures beaucoup plus fine, on pourra choisir une technologie plutôt qu'une autre en connaissance de cause, en se posant les bonnes questions dès le début.
- Enfin, le développement actuel permet de faire évoluer SmartTools au delà de cette passerelle Schema XML–Java et de le préparer à de futures évolutions technologiques (comme JAXB par exemple).

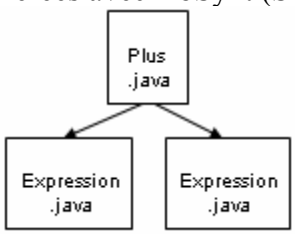
▪ **Analogies et différences avec JAXB**

JAXB n'a pas la même approche que nous dans le traitement des Schemas XML. La notion de type n'est pas la même. L'exemple ci-dessous met en relief ces différences, à partir d'un Schema XML regardons les classes java qui seraient générées avec AbSynt puis avec JAXB.

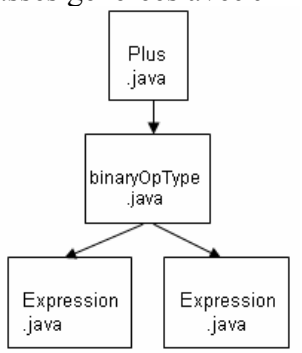
Schema XML

```
<xsd:element name="plus" ype="binaryOpType"/>
     <xsd:complexType name="binaryOpType">
          <xsd:sequence>
             <xsd:element name="expression" />
             <xsd: element name="expression" />
          </xsd:sequence>
     </xsd:complexType>
```

Classes générées avec AbSynt (SmartTools)



Classes générées avec JAXB



Avec AbSynt deux classes seraient générées (Plus.java et Expression.java) alors qu'avec JAXB trois classes seraient générées (Plus.java, BinaryOpType.java et Expression.java). Il n'y a pas de bonnes ni de mauvaises solutions mais des approches différentes qui répondent à des besoins différents. L'approche de JAXB permet de factoriser certains éléments des Schema XML, mais cela engendre une plus grande complexité dans l'architecture des classes.

AbSynt possède des caractéristiques précises qui répondent aux contraintes de modélisation de langages métiers et offre par exemple la possibilité de définir des tableaux particuliers (définition d'éléments optionnels, requis … etc). Ces contraintes ne sont pas encore prises en compte dans JAXB, c'est pourquoi nous sommes pour le moment obligés de construire notre propre outil.

Néanmoins, certaines spécifications de notre développement se sont basées sur celles de JAXB, comme par exemple la conversion des types XML en type JAVA (cf. Annexe 4 :). Il existe aussi des correspondances entre les deux structures XML et JAVA communes à SmartTools et JAXB.

ⓘ Pour plus d'information sur les différents projets existants, veuillez vous reporter à l'étude de comparative sur la programmation générative de Dennis Sonoski [18] .

4. Passerelle entre les Schemas XML et la programmation par objet via AbSynt

De nombreux langages métiers ou vocabulaires sont créés dans des domaines d'activité très variés tels que les mathématiques (**MathML** - Mathematical Markup Language), les applications sans fils (**WML** -Wireless Markup Language), les présentation multimédia (**SMIL** - Synchronized Multimedia Integration Language), le dessin vectoriel (**SVG** - Scalable Vector Graphics), etc.
Tous ces langages ont de plus en plus tendance à être définis par des Schema XML, par conséquent si l'on veut ouvrir SmartTools à ces langages et à leurs utilisateurs il faut pouvoir lire et traiter les Schema XML.

4.1. Pourquoi ouvrir SmartTools aux Schemas XML ?

Adopter des formats de données standardisés facilite l'échange d'informations entre logiciels. Les efforts pour accepter les formalismes du W3C sont certes motivés par le souci d'élargir le champ d'applications de SmartTools mais aussi par la volonté de faciliter son utilisation. De cette manière, les utilisateurs ne sont pas contraints à apprendre les langages internes de SmartTools. L'outil accepte, en entrée, aussi bien notre propre langage de définition d'arbre de syntaxe abstraite (AbSynt) qu'une DTD ou qu'un XML Schema. L'intérêt est de proposer pour ce type d'applications (langages) les outils d'édition, d'affichage et/ou de description sémantique de SmartTools. L'approche de génération automatique du couple analyseur syntaxique et afficheur semble envisageable pour des langages métiers simples. Elle serait certainement trop complexe pour des langages de programmation généraux (C++, Java). Cette génération devrait rendre de grands services dans notre contexte de petits langages métiers.

4.2. Méthodologie utilisée

Comme les DTDs dont ils sont les successeurs, les XML Schemas décrivent les structures de documents et ont aussi la possibilité de définir des éléments très complexes de structure incompatible avec le formalisme AbSynt. Ces éléments, comme pour les DTDs, devront être convertis en éléments plus simples pour être traduits en AbSynt. De plus, comme le formalisme

XML Schema est plus riche que celui de DTD ou AbSynt, la traduction est plus complexe et la définition de structure résultante plus permissible.
Avant de définir des règles de traduction, nous avons essayé de rapprocher les notions de base des deux formalismes (Schema XML & AbSynt).

4.2.1. Rapprochement de notions

Equivalences des notions constructeur et type
Tout d'abord, la notion d'élément correspond à notre notion de constructeur, comme pour les DTDs. Tout élément global ou local, de nom unique est transformé en un constructeur. Par contre, la notion de type des XML Schemas est complètement différente de la nôtre. En effet, leur notion de type complexe se rapporte au modèle de contenu d'un élément alors que la notre au contenu (ensemble des constructeurs possibles) d'un fils.
Par exemple, l'opérateur « plus » serait spécifié, avec un XML Schema, par un élément de nom plus ayant un type complexe décrivant son contenu (type réutilisable pour les opérateurs binaires moins, div et mult).

```
<xsd:element name="plus" type="binaryOpType"/>
   <xsd:complexType name="binaryOpType">
      <xsd:sequence>
         <xsd:group ref="Expression"/>
         <xsd:group ref="Expression"/>
      </xsd:sequence>
   </xsd:complexType>
```

Alors que dans notre formalisme, il serait spécifié par un constructeur nommé « plus » ayant deux fils de type « Expression ».

```
plus(Expression gauche, Expression droite)
```

Notre notion de type est équivalente à deux notions en XML Schema : les choix et les ensembles d'éléments substituables. Tout choix est donc traduit en un type. Nos constructeurs peuvent appartenir à plusieurs types.

La section ci-dessous explique comment traduire les différents modèles de contenu des éléments.

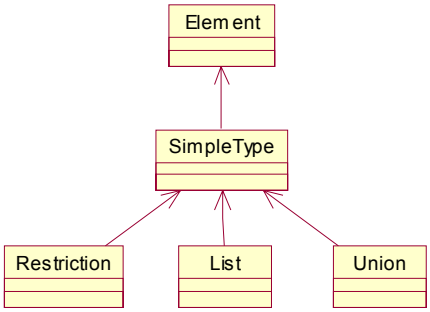
4.2.2. Modèles de contenu des constructeurs

Notre volonté de définir précisément les informations que l'on va devoir traiter permet d'identifier toutes les structures d'information que l'on va aborder. Dans notre contexte où ces structures d'information peuvent être très hétérogènes ce recensement est très utile pour la suite.

▪ **Types simples** (simpleType)
Le modèle de contenu d'un type simple correspond à une feuille et est donc équivalent pour nous à un constructeur atomique. Il faut juste établir une table de correspondance entre les types prédéfinis de XML Schema et les types Java.

La Figure 10 (ci-dessous) permet de représenter les relations entre les différentes balises d'un Schema XML. En effet, il permet d'avoir une vision des différentes balises impliquées dans la définition de types simples. Par exemple ce graphique signifie que la balise « SimpleType » définit le type de données accepté par un « Elément ». La définition du type simple peut être affinée par les balises « restriction », « list » ou « union ». On peut aussi exprimer cette hiérarchie en terme de père et de fils. SimpleType a un père nommé élément et peut avoir des fils nommés restriction, list ou union.

Structure



Implémentation en XML (exemple)

```
<xsd:element name="quantity">
 <xsd:simpleType>
  <xsd:restriction base="xsd:positiveInteger">
    <xsd:maxExclusive value="100"/>
  </xsd:restriction>
 </xsd:simpleType>
</xsd:element>
```

Figure 10 : Structure de données issue d'un SimpleType

Pour les unions de plusieurs types, il faut trouver le type parent le plus englobant (très souvent string). Par exemple, pour l'union des types byte et int, ce type serait int.

XML Schema

```
<element name="el1" type = "string">
```

AbSynt

```
el1 = el1 as java.lang.string
```

▪ **Types complexes** (complexType)
Ce schéma est plus dense que le précédent, les possibilités d'imbrication des balises sont beaucoup plus grandes.

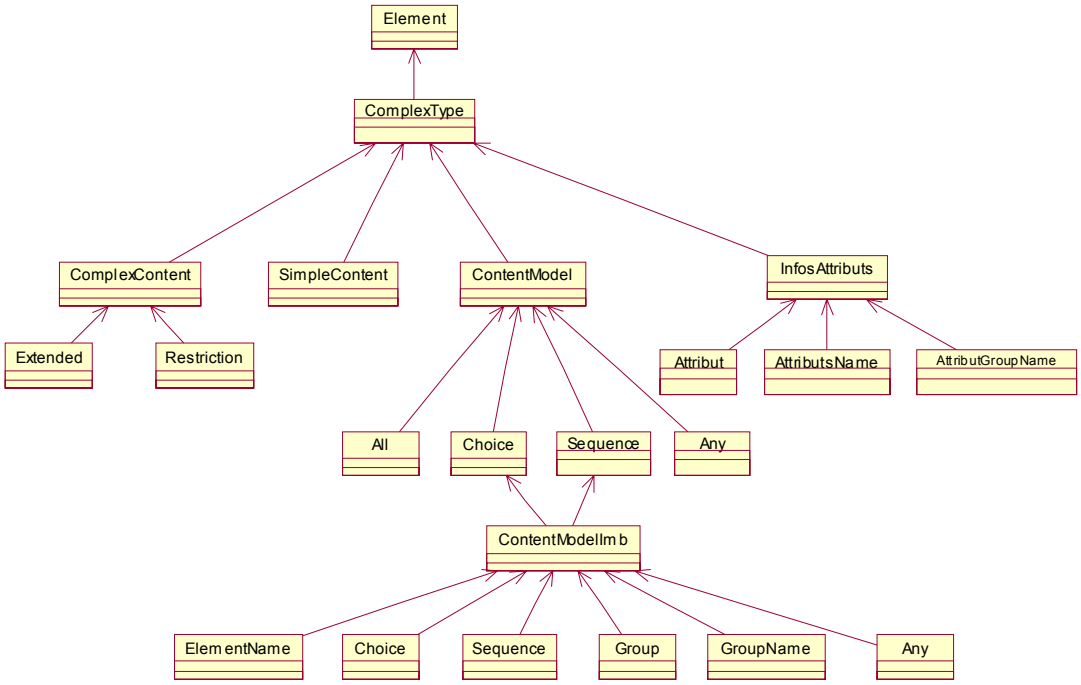


Figure 11 : Structure de données issue d'un compleType

Contrairement aux types simples, les modèles de contenu des types complexes sont plus difficiles à gérer car ils peuvent avoir des motifs très hétérogènes. Vous trouverez ci dessous la façon dont sont traités les principales balises et motifs des XML Schema lors de la traduction vers AbSynt. Pour chaque motif (éléments, sequence …), un exemple permet d'illustrer le traitement effectué.

<Sequence>

Une séquence d'éléments requise (et placée comme modèle de contenu d'un élément) correspond exactement au contenu d'un opérateur. Chaque élément de la séquence se traduit par un type et devient un fils requis, optionnel ou tableau selon sa cardinalité.

XML Schema

```
<complexType>
  <sequence>
    <element name="el1">
    <element name="el2" maxOccurs = "3">
    <element name="el3">
  </sequence>
</complexType>
```

AbSynt

```
nomType = nomOperateur (el1, el2, el3[3]);
```

<all>

La structure « All » peut contenir zéro ou une fois chacun des éléments qui peuvent apparaître dans n'importe quel ordre. De plus, cette structure ne peut contenir que des éléments ayant une cardinalité inférieure ou égale à 1.

Comme il n'existe pas de correspondant au « all » dans AbSynt, ce modèle de contenu est transformé en un seul fils tableau de cardinalité maximale au plus égale au nombre des éléments inclus dans le all. Cette transformation implique une très légère perte d'information de structure puisque les éléments pourraient être répétés (ce qui est contraire à la définition d'un all).

XML Schema

```
<complexType>
  <all>
    <element name="el1">
    <element name="el2">
    <element name="el3">
  </all>
</complexType>
```

AbSynt

```
nomType = nomOperateur (elTemp[3]) ;
elTemp =  el1 ,
          el2 ,
          el3 ;
```

<choice>

Tout choix d'éléments correspond à un type regroupant tous ces éléments. Pour connaître la cardinalité minimale (vs. maximale) du fils correspondant, il faut calculer le minimum (vs. maximale) des cardinalités minimales (vs. maximales) des éléments et du choix.

XML Schema

```
<complexType>
  <choice>
    <element name="el1" maxOccurs = "1">
    <element name="el2" maxOccurs = "3">
    <element name="el3" maxOccurs = "1">
  </choice>
</complexType>
```

AbSynt

```
nomType = nomOperateur (elTemp[3]) ;
elTemp = el1    ,
         el2[3] ,
         el3    ;
```

<group>

Tout groupe n'est qu'une factorisation du modèle de contenu. Les groupes dont le contenu est un choix sont conservées et sont à traiter comme des choix. Les autres sont à remplacer par leurs valeurs.

XML Schema

```
<group name="group1">
  <choice>
    <element name="el1" maxOccurs = "1">
    <element name="el2" maxOccurs = "3">
    <element name="el3" maxOccurs = "1">
  </choice>
</complexType>
```

AbSynt

```
group1 = el1    ,
         el2[3] ,
         el3    ;
```

<element>

Tout “élément” dans un XML Schema correspond à un opérateur dans Absynt.

4.3. Implémentation de la solution

Dans cette section, nous allons implémenter les solutions théoriques expliquées précédemment. Dans un premier temps, nous présenterons la structure de données qui aura pour rôle de stocker les informations issues du Schema XML. Cette première phase préparatoire permettra d’initier le processus de transformation de cette structure de données en d’autres structures de données pour finalement arriver à la transformation d’un Schema XML en AbSynt. Dans un deuxième temps, nous détaillerons l’algorithme de transformation étape par étape.

L’architecture de la structure de données utilisées correspond aux Figures 10 et 11.

4.3.1. Les structures de données

▪ **Structures de données auxiliaires**
Les sept définitions suivantes correspondent à des structures de données secondaires mais nécessaires à l’algorithme d’importation des XML Schemas.

Cette structure, « informations attributs », est utilisée dans les groupes d’attributs et les types complexes.

$$\textit{informations attributs} \triangleq \begin{cases} \textit{(attribut / nom d'un attribut / nom d'un groupe d'attributs)*} \\ \textit{(nom d'un espace de nom)*} \end{cases}$$

Cette structure, modèle de contenu, est utilisée dans les types complexes pour indiquer le modèle de contenu et ses cardinalités qui sont optionnelles et par défaut à 1.

$$\textit{modèle de contenu} \triangleq \begin{cases} \textit{all / choix / séquence / groupe / nom de groupe} \\ \textit{(cardinalité minimale) ?} \\ \textit{(cardinalité maximale) ?} \end{cases}$$

Ces quatre structures modélisent respectivement le contenu d’un all, d’un choix, d’une séquence et d’un `any`.

$$\begin{aligned} \textit{all} &\triangleq \textit{(nom d'élément)*} \\ \textit{choix} &\triangleq \textit{modèle de contenu imbriqué} \\ \textit{séquence} &\triangleq \textit{modèle de contenu imbriqué} \\ \textit{any} &\triangleq \textit{(nom de l'espace de nom)+} \end{aligned}$$

Cette structure représente le contenu possible d’un composant d’un choix, d’une séquence ou d’un groupe.

$$\textit{modèle de contenu imbriqué} \triangleq \begin{cases} \textit{(nom d'élément / choix / séquence / groupe / nom de groupe / any)*} \\ \textit{(cardinalité minimale) ?} \\ \textit{(cardinalité maximale) ?} \end{cases}$$

▪ **Structures de données principales**
Les six structures suivantes sont utilisées dans des tables de hachage pour stocker les informations nécessaires à la construction du formalisme. L’algorithme d’importation envisagé requiert cinq tables de

hachage : une pour les éléments, une pour les attributs, une pour les groupes d'attributs, une pour les types simples ou complexes et enfin une pour les groupes de modèle de contenu.

élément ≜ { *nom*, *nom du type*, *abstrait ou non*, *nul possible ou non*, *(nom d'un élément substituable)**

attribut ≜ { *nom*, *nom du type*, *requis | optionnel | interdit*, *valeur fixe ou par défaut*

groupe d'attributs ≜ { *nom*, *informations attributs*

type simple ≜ { *nom*, { *nom du type de base restreint* / *nom du type des items de la liste* / *(nom d'un type de l'union)+*

type complexe ≜ { *nom*, *global ou local*, *(nom d'un type le dérivant par extension)**, *mélange texte possible ou non*, { { { { *nom du type complexe de base étendu* / *(modèle de contenu) ?* } / *nom du type simple de base dérivé* / *(modèle de contenu) ?* } *informations attributes* } / *nom du type complexe de base restreint*

groupe ≜ { *nom*, *all | choix | séquence*

4.3.2. Algorithme

L'algorithme utilisé pour effectuer la transformation d'un Schema XML vers AbSynt peut être décomposé en six phases distinctes. Le schéma ci-dessous présente une vue synthétique de l'algorithme, chaque phase est expliquée plus en détail à la suite de ce schéma.

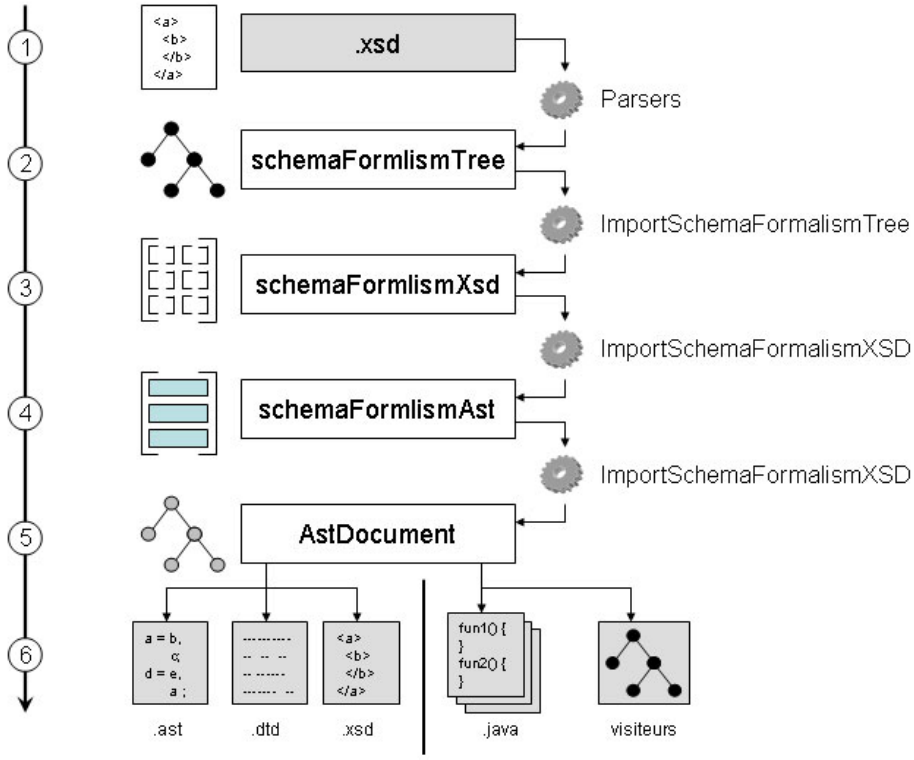


Figure 12 : Etapes de transformations d'un schema XML en AbSynt

Sur la partie gauche du schéma, on peut visualiser les différentes étapes numérotées de 1 à 6. Puis les représentations graphiques décrivent l'état des structures de données à chaque étape de l'algorithme (arbre, tableau … etc). « .xsd », « schemaFormalismTree », « schemaFormalismXsd », « schemaFormalismAst », et « AstDocument » correspondent aux noms des objets qui stockent ces structures de données. Enfin, , « ImportSchemaFormalismTree », « ImportSchemaFormalismXSD », « ImportSchemaFormalismAst », et « Parsers » correspondent aux fonctions qui réalisent les traitements sur les structures de données et qui permettent le passage d'une structure à une autre.

L'algorithme se décompose en cinq phases :

Etape 1 à 2
Lecture du XML Schema (fichier avec l'extension .xsd) en utilisant l'API DOM.
Cette API permet de représenter un document XML sous forme d'arbre, cela facilite la manipulation du Schema XML pour effectuer des traitements ou des transformations plus compliquées. Cet arbre est non typé, toutes les informations sont de type string (chaîne de caractères).

Etape 2 à 3
Stockage des différents éléments de l'arbre dans les structures de données primaires et secondaires. Ces structures de données correspondent à des tables de hachages. Par exemple, les éléments, complexType, simpleType … sont stockés dans des tables de hachages séparées, des références entre tous ces éléments permettent de conserver la structure du document.

- Tout choix, séquence, `all`, groupe ou élément de cardinalité maximale égale à 0 est supprimé (ainsi que les références à un tel groupe ou à un groupe ayant un tel choix, séquence ou `all`).
- Dans la table des types, toute référence à un attribut ou à groupe d'attributs est remplacé par sa valeur ainsi que toute référence à un groupe de modèle de contenu séquence ou `all` ; si cette dernière avait des informations de cardinalité, il faut les reporter au modèle de contenu.

▪ Seuls les groupes de modèle de contenu choix sont conservés car leurs noms pourront être utilisés comme noms de type dans notre formalisme (choix=type).
▪ Après cette phase, les tables des attributs et des groupes d'attributs ne servent plus et la table des groupes de modèle de contenu ne devrait contenir que des choix (au premier niveau).

Etape 3 à 4
Le passage de l'étape 3 à l'étape 4 est celle qui est la plus délicate car c'est à ce moment que l'on passe d'une logique de document XML à celle de document Absynt avec toutes les hypothèses et pertes d'informations que cela comporte.

▪ Tout élément est transformé en constructeur.
▪ Tout choix, tout ensemble d'éléments substituables, tout élément fils direct d'une séquence et l'ensemble des éléments globaux constitue un type.

Etape 4 à 5
Cette étape se déroule sans réelle difficulté, une simple fonction permet de passer d'une structure de données à l'autre. A l'issue de cette dernière étape on obtient un objet AbSynt.

Etape 5 à 6
Avec cet objet AbSynt, on peut imaginer différents scénarios en foncion des besoins de l'utilisateur :
- Générer des classes JAVA qui seront utilisées dans SmartTools, le but est d'obtenir une représentation sous forme d'arbre de notre langage métier mais contrairement à l'étape 2 cet arbre serait typé. Cela signifie que certains éléments (ou nœuds de l'arbre) peuvent être manipulés comme des entiers, string ou des booléens.
- Générer la DTD correspondante ou le XML Schema.
- Générer les « visiteurs » qui seront utilisés pour parcourir notre langage.

AbSynt propose de nombreux outils adaptés à SmartTools, le plus difficile étant d'arriver à convertir la description de notre langage métiers d'un formalisme vers AbSynt.

4.4. Problèmes rencontrés

Au cours de ce stage, différents problèmes sont survenus. Des problèmes liés à l'environnement même du projet et des difficultés plus techniques liées directement au sujet.

4.4.1. Problèmes généraux

▪ Compréhension de la problématique du sujet
Selon le degré d'avancement du projet, de la complexité de celui-ci, il faut plus ou moins de temps pour bien comprendre les problématiques liées au sujet. En général on aborde le sujet avec une vision partielle des difficultés qui lui sont inhérentes. Il est un peu frustrant de ne pas pouvoir anticiper ces difficultés et de les découvrir que lorsque l'on y est confronté. Cela engendre une perte de temps car on développe une solution puis on se rend compte que ce n'est pas réalisable et qu'il faut recommencer. Cela fait aussi parti de l'apprentissage.

▪ Prise en main de SmartTools
La première difficulté concerne la compréhension du projet sur lequel on travaille. Le projet OASIS a débuté il y a 4 ans, et c'est au cours de ce projet que le logiciel SmartTools a été développé. Contrairement à un logiciel commercial qui est généralement accompagné d'une documentation et de ressources facilement exploitables, SmartTools est un prototype de recherche et même s'il existe

une documentation, il n'est pas facile à prendre en main, en effet ce type de logiciel est en évolution permanente. Le premier travail a été de comprendre comment il fonctionnait.

4.4.2. Problèmes techniques

▪ Contraintes initiales

Il n'existe pas de solutions parfaites pour résoudre le problème de passage d'XML Schema vers AbSynt, car les Schemas XML n'ont pas été créés pour modéliser l'architecture d'applications utilisant la technologie objet mais pour modéliser la structure de documents. Par conséquent la nature même du sujet et des technologies utilisées, impliquaient des contraintes de base.

▪ Gestion des noms

L'écriture de Schemas XML est relativement souple, et cela est d'autant plus difficile à gérer car un Schema peut être écrit de plusieurs façons pour finalement aboutir au même résultat.

```
<xsd:element name="purchaseOrder" type="PurchaseOrderType"/>

 <xsd:complexType name="PurchaseOrderType">
  <xsd:sequence>
   <xsd:element name="shipTo" type="USAddress"/>
   <xsd:element name="billTo" type="USAddress"/>
  </xsd:sequence>
</xsd:complexType>
```

=

```
<xsd:element name="purchaseOrder">
 <xsd:complexType>
  <xsd:sequence>
   <xsd:element name="shipTo" type="USAddress"/>
   <xsd:element name="billTo" type="USAddress"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:element/>
```

Les deux Schemas XML ci-dessus signifient la même chose en terme de contraintes pour l'utilisateur, mais le premier est plus détaillé que le deuxième. En effet, la balise complexType possède un nom dans le premier cas et pas dans l'autre. Cela pose des problèmes lorsque l'on veut stocker un complexType dans une table de hachage où le nom est indispensable car c'est une clé dans la table. La première solution qui vient à l'esprit est de se dire qu'il suffit de donner un nom fictif à ce complexType, mais pour cela il faut vérifier que ce nom n'est pas employé dans le reste du document. Cela devient tout de suite plus lourd. Pour plus de clarté, il convient d'écrire le Schema comme dans le premier exemple.

▪ Mise en place de l'algorithme

Il y a plusieurs façon d'aborder un problème et donc dans notre cas plusieurs algorithmes susceptibles de répondre à notre problématique. Néanmoins, certains se révèlent plus efficaces que d'autres. La première approche pour traiter le Schema XML a été de « l'éclater complètement », c'est-à-dire stocker tous les éléments du Schema dans des tables de hachages séparées. Mais cette solution n'était pas réaliste car les traitements à effectuer sur les tables de hachages étaient bien trop compliqués. La deuxième solution a été de définir une structure de données avec un périmètre plus précis et limité dès le départ, ainsi les traitements à effectuer pour la conversion du Schema en AbSynt ont été plus faciles à mettre en place. Cette structure de données est représentée par les figures 10 et 11.

▪ Pertes d'information

Certains problèmes sont liés directement au langage AbSynt. Comme il a été dit précédemment, les Schema XML sont beaucoup plus riches que le formalisme AbSynt, cela engendre nécessairement une perte d'information lors de la traduction. La seule solution revient à faire évoluer le formalisme

AbSynt si la perte d'information est trop préjudiciable. SmartTools est un générateur de composant, si on lui fournit des données erronées en entrée elles le seront nécessairement en sortie.

4.5. Perspectives

SmartTools est basé sur les standards du W3C qui sont amenés à évoluer. Pour garder des outils adaptés aux besoins des utilisateurs, il est nécessaire de s'adapter à ces standards. Voici quelques pistes possibles susceptibles d'améliorer le passage d'un Schema XML vers AbSynt.

▪ **Suivre l'évolution des spécifications XML Schema et AbSynt**
L'élaboration de passerelles entre les Schemas XML et AbSynt dépend directement des contraintes et des limites de ces deux formalismes. Par conséquent, une évolution de l'une des deux technologies peut engendrer de nouvelles contraintes ou de nouvelles possibilités quant à la définition d'une passerelle d'une technologie à une autre.

▪ **Analyser la technique d'écriture des Schemas des utilisateurs potentiels**
Les Schemas XML ont été créés pour modéliser la structure de documents, néanmoins ils sont aussi utilisés pour structurer des langages métiers. Il existe plusieurs utilisations des Schemas et mais aussi plusieurs manières de les écrire. Il serait intéressant de comprendre comment les utilisateurs potentiels de SmartTools écrivent leur Schema, cela permettrait de faire des hypothèses plus précises sur leur lecture, de focaliser notre attention sur certains aspects des Schemas très utilisés et à l'opposer de ne pas tenir compte de certaines contraintes.

▪ **Gestion des « substitutions group »**
La balise « substitution group » n'est pas encore gérée, mais cette balise devrait être gérée comme la balise « choice » vue précédemment. Cela permettrait de prendre en compte une plus grande variété de Schema.

▪ **Importations de Schemas XML**
Les XML Schemas ayant des importations ne pourront pas être traités car la notion d'importation n'existe pas encore dans notre formalisme de définition de langages.
L'ajout de cette notion qui semble, de prime abord, simple pour la génération des classes des types et les constructeurs implique aussi des modifications dans les parties sémantiques (plus particulièrement sur les parcours des arbres).

5. Conclusion

L'objectif de ce stage était d'élaborer une passerelle entre les formalismes XML Schema et la programmation par objet (JAVA). Cet objectif a été en partie atteint même si des améliorations sont réalisables, notamment en terme de performance et d'architecture logicielle. De plus les standards ou les solutions propriétaires sur lesquels nous nous sommes basés sont susceptibles d'évoluer ce qui offriraient des possibilités d'améliorer la traduction de Schema XML vers la technologie Objet Java. Mais au-delà de la réalisation de ce module logiciel, ce stage a permis d'approfondir une problématique dans la perspective d'une évolution de SmartTools vers une nouvelle communauté d'utilisateurs. Par exemple cette étude nous a amené à établir un bilan des technologies existantes dans notre domaine de recherche, et d'identifier les analogies et les différences par rapport à notre projet.

SmartTools est un prototype de recherche, il est donc par définition en perpétuelle évolution. Les solutions d'aujourd'hui ne seront certainement pas celles de demain. Des technologies répondant complètement ou partiellement à nos problématiques émergent où disparaissent régulièrement. Par conséquent, il faut savoir se préparer à accueillir ces nouvelles technologies, et comprendre l'évolution du domaine de recherche afin de choisir des solutions pérennes, efficaces et évolutives.
Dans la continuité de ce stage, plusieurs possibilités sont à envisager. Tout d'abord, il faudrait pouvoir prendre en compte une plus grande variété de Schema XML et donc être plus proche de ses spécifications. Ensuite, il faudrait réaliser une veille technologique pour surveiller l'évolution de solutions et anticiper l'émergence de standards répondant à nos besoins présents ou futurs. Enfin, il est toujours intéressant de confronter les solutions développées aux utilisateurs, cela permet de savoir s'il on est en phase avec leurs besoins et cela permet aussi de valider le travail accompli.

Dans le cadre de ma formation, ce stage m'a permis d'aborder différentes technologies (Schema XML, JAVA) et méthodes de conception en génie logiciel (Design patterns). J'ai eu l'occasion d'évoluer dans un institut de recherche ce qui m'a permis de compléter ma découverte du monde professionnel débutée lors de mon premier stage. De plus, il a été très intéressant de découvrir le monde de la recherche et de l'innovation, et de prendre conscience des liens entretenus avec le monde industriel.

Table des figures

Table des Annexes

Annexe 1 : Bon de commande en XML et Schema XML correspondant

Extrait de « XML Schema Part 0: Primer » disponible sur le site du W3C[7] .

▪ Instance d'un bon de commande en XML

```
<?xml version="1.0"?>
<purchaseOrder orderDate="1999-10-20">
    <shipTo country="US">
        <name>Alice Smith</name>
        <street>123 Maple Street</street>
        <city>Mill Valley</city>
        <state>CA</state>
        <zip>90952</zip>
    </shipTo>
    <billTo country="US">
        <name>Robert Smith</name>
        <street>8 Oak Avenue</street>
        <city>Old Town</city>
        <state>PA</state>
        <zip>95819</zip>
    </billTo>
    <comment>Hurry, my lawn is going wild!</comment>
    <items>
        <item partNum="872-AA">
            <productName>Lawnmower</productName>
            <quantity>1</quantity>
            <USPrice>148.95</USPrice>
            <comment>Confirm this is electric</comment>
        </item>
        <item partNum="926-AA">
            <productName>Baby Monitor</productName>
            <quantity>1</quantity>
            <USPrice>39.98</USPrice>
            <shipDate>1999-05-21</shipDate>
        </item>
    </items>
</purchaseOrder>
```

▪ Schema XML correspondant au fichier XML ci dessus

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

 <xsd:annotation>
  <xsd:documentation xml:lang="en">
   Purchase order schema for Example.com.
   Copyright 2000 Example.com. All rights reserved.
  </xsd:documentation>
 </xsd:annotation>

 <xsd:element name="purchaseOrder" type="PurchaseOrderType"/>

 <xsd:element name="comment" type="xsd:string"/>

 <xsd:complexType name="PurchaseOrderType">
  <xsd:sequence>
   <xsd:element name="shipTo" type="USAddress"/>
   <xsd:element name="billTo" type="USAddress"/>
   <xsd:element ref="comment" minOccurs="0"/>
   <xsd:element name="items"  type="Items"/>
  </xsd:sequence>
  <xsd:attribute name="orderDate" type="xsd:date"/>
 </xsd:complexType>

 <xsd:complexType name="USAddress">
  <xsd:sequence>
   <xsd:element name="name"   type="xsd:string"/>
   <xsd:element name="street" type="xsd:string"/>
   <xsd:element name="city"   type="xsd:string"/>
   <xsd:element name="state"  type="xsd:string"/>
   <xsd:element name="zip"    type="xsd:decimal"/>
  </xsd:sequence>
  <xsd:attribute name="country" type="xsd:NMTOKEN"
     fixed="US"/>
 </xsd:complexType>

 <xsd:complexType name="Items">
  <xsd:sequence>
   <xsd:element name="item" minOccurs="0" maxOccurs="unbounded">
    <xsd:complexType>
     <xsd:sequence>
```

```
      <xsd:element name="productName" type="xsd:string"/>
      <xsd:element name="quantity">
       <xsd:simpleType>
        <xsd:restriction base="xsd:positiveInteger">
         <xsd:maxExclusive value="100"/>
        </xsd:restriction>
       </xsd:simpleType>
      </xsd:element>
      <xsd:element name="USPrice"  type="xsd:decimal"/>
      <xsd:element ref="comment"   minOccurs="0"/>
      <xsd:element name="shipDate" type="xsd:date" minOccurs="0"/>
     </xsd:sequence>
     <xsd:attribute name="partNum" type="SKU" use="required"/>
    </xsd:complexType>
   </xsd:element>
  </xsd:sequence>
 </xsd:complexType>

 <!-- Stock Keeping Unit, a code for identifying products -->
 <xsd:simpleType name="SKU">
  <xsd:restriction base="xsd:string">
   <xsd:pattern value="\d{3}-[A-Z]{2}"/>
  </xsd:restriction>
 </xsd:simpleType>

</xsd:schema>
```

Annexe 2 : Tableau d'horaires de trains en XML et DTD correspondante

Extrait de « SELFHTML XML/DTD Représentation de données XML » disponible sur le site de SELFHTML [8] .

▪ Un tableau d'horaires de trains en XML

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE plan route SYSTEM "plan route.dtd">
<?xml-stylesheet type="text/css" href="plan route.css" ?>
<plan route>
<gare>
 <nom>Marseille Saint Charles</nom>
  <evenement>
   <depart>
    <heure>17.03</heure>
    <type train classe=ic>IC</type train>
    <numero train>58483</numero train>
    <route>Arles 17.59, Avignon-Centre 18.20
     <destination>Lyon-Part-Dieu à 20.43</destination>
    </route>
   </depart>
  </evenement>
  <evenement>
   <arrivee>
    <heure>18.38</heure>
    <type train class="tgv">TGV</type train>
    <numero train>23</numero train>
    <route><demarrage>Nîmes 19.40</demarrage>
     Montpellier 20.07
    </route>
   </arrivee>
  </evenement>
 </gare>
</plan route>
```

▪ DTD correspondante au fichier XML ci dessus

```
<!ELEMENT collection (description,recipe*)>
<!ELEMENT plan route  (gare)>
<!ELEMENT gare   (nom,(evenement)*)>
<!ELEMENT nom      (#PCDATA)>
<!ELEMENT evenement  (depart | arrivee)>
<!ELEMENT depart   (heure,type train,numero train,route)>
<!ELEMENT arrivee   (heure,type train,numero train,route)>
<!ELEMENT heure      (#PCDATA)>
<!ELEMENT type train    (#PCDATA)>
<!ELEMENT numero train     (#PCDATA)>
<!ELEMENT route     (#PCDATA | demarrage | destination)*>
<!ELEMENT demarrage     (#PCDATA)>
<!ELEMENT destination      (#PCDATA)>
```

Annexe 3 : Complément d'information sur le formalisme / langage AbSynt

▪ Définitions d'attributs

Cette partie contient les définitions d'attributs des constructeurs et des types. Chaque attribut a un nom, peut être requis, constant ou facultatif, avoir une valeur et être de type String ou d'un type de base de Java. Il peut être défini sur plusieurs types et/ou constructeurs.

```
Attribute definitions {
REQUIRED attributRequis as java.lang.String in opAtomique, opVariable2;
FIXED attributConstant="2" as java.lang.Integer in %TypeInclus;
IMPLIED attributFacultatif="ardèche" as java.lang.String in opVariable;
}
```

▪ Définitions d'informations supplémentaires utiles pour les traitements sémantiques.

Cette partie contient les définitions de données additionnelles, de type complexe, pouvant être manipulées lors des traitements sémantiques. Chaque donnée est nommée, typée et peut être définie sur plusieurs constructeurs et/ou types. Contrairement aux constructeurs et aux attributs, ces informations n'apparaissent pas au niveau du document XML; elles sont volatiles.

```
Extra data definitions {
vecteur as java.util.HashMap in opAtomique, %TypeInclus;
}
```

▪Contraintes

- Tout constructeur ou type utilisé doit être défini de manière unique (il peut être utilisé avant d'avoir été défini). Un même nom peut être utilisé pour un type, un constructeur, un attribut et une information supplémentaire.
- Les noms des fils d'un constructeur doivent être différents.
- Tout attribut (ou information supplémentaire) ne peut être défini qu'une fois.
- Toute définition récursive ou circulaire de type est interdite.
- Toute définition de type doit contenir au moins un constructeur, directement ou d'un type inclus, ayant le premier fils de type différent ou non requis.

Il existe d'autres contraintes liées à l'utilisation de ce langage dans certains cas particuliers.

Annexe 4 : Table de correspondances entre les types XML et JAVA

Extrait de *JAXB Specification, final V1.0,* chapitre « Binding XML Schema to Java representation » p.58. Sun Microsystem.

XML Schema Data type	Java Data Type
xsd:string	**java.lang.String**
xsd:integer	**java.math.BigInteger**
xsd:int	**int**
xsd.long	**long**
xsd:short	**short**
xsd:decimal	**java.math.BigDecimal**
xsd:float	**float**
xsd:double	**double**
xsd:boolean	**boolean**
xsd:byte	**byte**
xsd:QName	**javax.xml.namespace.QName**
xsd:dateTime	**java.util.Calendar**
xsd:base64Binary	**byte[]**
xsd:hexBinary	**byte[]**
xsd:unsignedInt	long
xsd:unsignedShort	int
xsd:unsignedByte	short
xsd:time	java.util.Calendar
xsd:date	java.util.Calendar
xsd:anySimpleType	java.lang.String

Bibliographie

[1] COURBIS C. *Contribution à la programmation générative. Application dans le générateur SmartTools : technologies XML, programmation par aspects et composants.* Université de Nice Sophia Antipolis, December 2002, 166 p.

[2] Attali I., Courbis C., Degenne P., Fau A., Fillon J., Parigot D., Pasquier C., Sacerdoti Coen C. SmartTools : a development environment generator based on XML technologies. In *XML Technologies and Software Engineering*, Toronto, Canada. ICSE'2001, ICSE workshop proceedings.
http://www-sop.inria.fr/oasis/Didier.Parigot/publications/Parigot01b.ps.gz.

[3] Courbis C., Degenne P., Fau A., Parigot D., Variamparambil J. *Un modèle de composants pour l'atelier de développement SmartTools*. In : Systèmes à composants adaptables et extensibles, Octobre 2002.

[4] Entreprise Java Beans (EJB). http://java.sun.com/products/ejb.

[5] Objectweb. http://www.objectweb.org/.

[6] UML - Unified Modeling Language. http://www.uml.org.

[7] The World Wide Web Consortium (W3C). http://www.w3.org/.

[8] SELFHTML. *Représentation de données XML*. http://selfhtml.selfhtml.com.fr/xml/representation/.

[9] JAXB de sun - Java Architecture XML Binding. http://java.sun.com/xml/jaxb/.

[10] JIXB de Sonoski software Solution. http://jibx.sourceforge.net/.

[11] Castor de Exolab. http://www.castor.org/.

[12] JBind de Apache Software - http://sourceforge.net/projects/jbind/.

[13] Quick de JXQuick - http://sourceforge.net/projects/jxquick/.

[14] Zeus de Zeus project - http://zeus.enhydra.org/.

[15] XSD Part 0 - XML Schema Part 0: Primer, W3C Recommendation 2 May 2001. Disponible sur http://www.w3.org/TR/xmlschema-0/

[16] XSD Part 1 - XML Schema Part 1: Structures, W3C Recommendation 2 May 2001. Disponible sur http://www.w3.org/TR/xmlschema-1/

[17] XSD Part 2 - XML Schema Part 2: Datatypes, W3C Recommendation 2 May 2001. Disponible sur http://www.w3.org/TR/xmlschema-2/

[18] XML and Java technologies: Data binding, Part 1: Code generation approaches -- JAXB and more. SOSNOSKI D., 1 January 2003. Disponible sur http://www-106.ibm.com/developerworks/xml/library/x-databdopt/.

Résumé

La qualité du logiciel et sa capacité à évoluer, ainsi que la rapidité du développement, sont des soucis majeurs pour les industriels. Un logiciel bien conçu doit pouvoir s'adapter rapidement aux demandes des clients et aux nouvelles technologies pour pouvoir lutter contre la concurrence. Il doit aussi être capable d'échanger des données très variées avec d'autres applications, particulièrement depuis l'avènement d'Internet. Les exigences vis-à-vis des logiciels ont aussi été modifiées à cause des disparités de connaissances des utilisateurs et des programmeurs, des contraintes de temps et d'argent, et des nécessités d'adaptation rapide aux besoins du marché. Les logiciels doivent être conviviaux, faciles à utiliser, ouverts, modulaire et flexibles. Pour prendre en compte ces bouleversements, de nouveaux standards pour l'échange de données et de nouvelles techniques de programmation ont émergé.

Tout d'abord, avec l'effervescence liée à la naissance du Web, il y a eu une volonté de standardiser les langages et protocoles, ce qui a conduit à la création du W3C. Ce consortium a proposé un nouveau format de données, XML (Extensible Markup Language), issu de SGML (Standard Generalized Markup Language) afin de simplifier et d'uniformiser les échanges d informations entre applications, indépendamment de tout langage et plateforme.

Aujourd'hui, on constate que de nombreux langages métiers sont créés dans des domaines d'activité très variés tels que les mathématiques (MathML), la chimie (CML - Chemical Markup Language), la génétique (BSML - Bioinformatic Sequence Markup Language), le commerce, (BizTalk ou CommerceXML), les applications sans fils (WML -Wireless Markup Language) …etc. Ces langages métiers ont de plus en plus tendance à être définis grâce à un formalisme de modélisation de document appelé Schema XML. Les Schemas XML permettent de définir le vocabulaire qui sera employé dans le langage ainsi que sa grammaire (règles d'écriture).

De nouvelles techniques ont aussi émergé en génie logiciel, notamment avec la programmation par objets et ses notions d'encapsulation et d'héritage propices à la modularité, la réutilisation et l'extensibilité du code. Puis, d'autres techniques sont venues s'ajouter pour rendre ce type de programmation plus efficace, comme les patrons de conception, la programmation par aspects, la programmation générative et la programmation par modèle (UML).

Au croisement de ces évolutions technologiques, SmartTools prend tout son sens dans le développement d'applications communicantes (traitement et échanges d'informations). Avec la seule description d'un langage métier, la plateforme va générer un environnement de développement à base de composants logiciels. Le développeur pourra se concentrer sur la valeur ajoutée de son application car les outils de bases qui lui permettent de travailler (manipuler, effectuer des traitements) avec son langage auront été générés auparavant.

L'objectif du stage a été d'essayer d'ouvrir SmartTools aux Schemas XML, afin qu'il puisse traiter tous ces nouveaux langages métiers et donc s'ouvrir à une nouvelle communauté d'utilisateur. Pour réussir cette ouverture, toute la problématique a été de trouver des règles de correspondances entre le formalisme des Schemas XML et la programmation par objets, technologie utilisée à l'intérieur de SmartTools. Trouver ces règles de correspondance n'a pas été trivial car les Schemas XML n'ont pas été créés pour modéliser l'architecture d'applications utilisants la technologie objet mais pour modéliser la structure de documents, ils sont néanmoins de très bon outils pour modéliser des langages métiers. Afin d'élaborer un solution acceptable, un des aspects de ce stage a été de poser des hypothèses pour prendre en compte un large éventail de Schemas qui puisse être fidèlement transposable dans la technologie objet. En effet, le but de la plate-forme n'est pas de prendre en compte de façon universelle tous les Schemas XML mais de garantir un traitement fiable pour ceux qui respecteraient certains principes d'écriture. Pour valider les solutions retenues, une implémentation a été réalisée au sein de SmartTools.