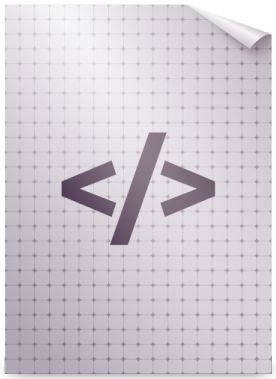
INSA Lyon
20, avenue Albert Einstein
69621 Villeurbanne Cedex

Livrable de projet

Grammaire et langage

« Analyse de fichiers XML »

du 14 Mars au 3 Mars 2014



Hexanôme :
Guillaume Abadie
Thierry Cantenot
Juliette Courlet
Rémi Domingues
Adrien Duffy-Coissard
Ahmed Kachkach

Enseignants :
Nabila Benharkat
Eric Guérin

Année scolaire 2013-2014

Sommaire

1. Introduction

L'objectif de ce projet est de réaliser un processeur xml en C++, en utilisant en plus les outils flex et bison. Ceux-ci nous permettront d'analyser syntaxiquement et sémantiquement le langage xml, afin de créer une grammaire simplifiée. Notre programme a 3 fonctionnalités principales : parser un fichier xml et l'afficher, valider un document xml par rapport à un document xsd, et enfin transformer un document xml avec une feuille de style xsl. En raison du temps qui est imparti pour ce projet, ces fonctionnalités ne couvriront pas toutes les possibilités offertes par le langage xml, afin de réduire le temps de développement de ce programme.
Nous l'utiliserons en ligne de commande de la façon suivante :

Parsage/affichage du fichier doc.xml :

```
xmltool -p doc.xml
```

Validation du fichier doc.xml par rapport au fichier doc.xsd ! :

```
xmltool -v doc.xml doc.xsd
```

Transformation du fichier doc.xml avec la feuille de style doc.xsl :

```
xmltool -t doc.xml doc.xsl
```

Voici l'architecture global de notre programme :

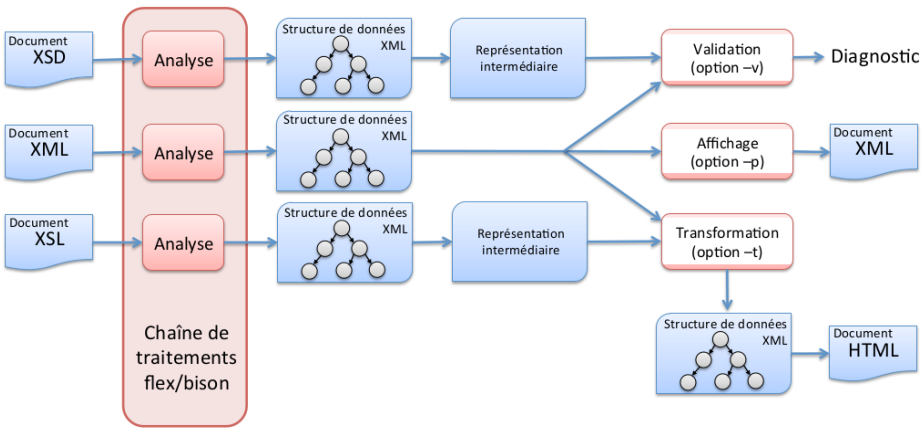


FIGURE 1.1 – Architecture globale

La conception sera détaillée plus précisément dans la suite du document.

2. Namespace XML

Cette partie décrit la représentation C++ d'un XML ainsi que les traitements associés.

Les hypothèse suivantes sont également formulées :
— Les documents XML à parser ne contiendront pas de DTD interne
— Les Processing Instruction (PI) ne contiendront que des attributs
— Aucune référence ne sera présente dans les documents XML

2.1 Diagramme de classes

Les classes ici décrites seront rattachées au *namespace Xml* défini en C++.

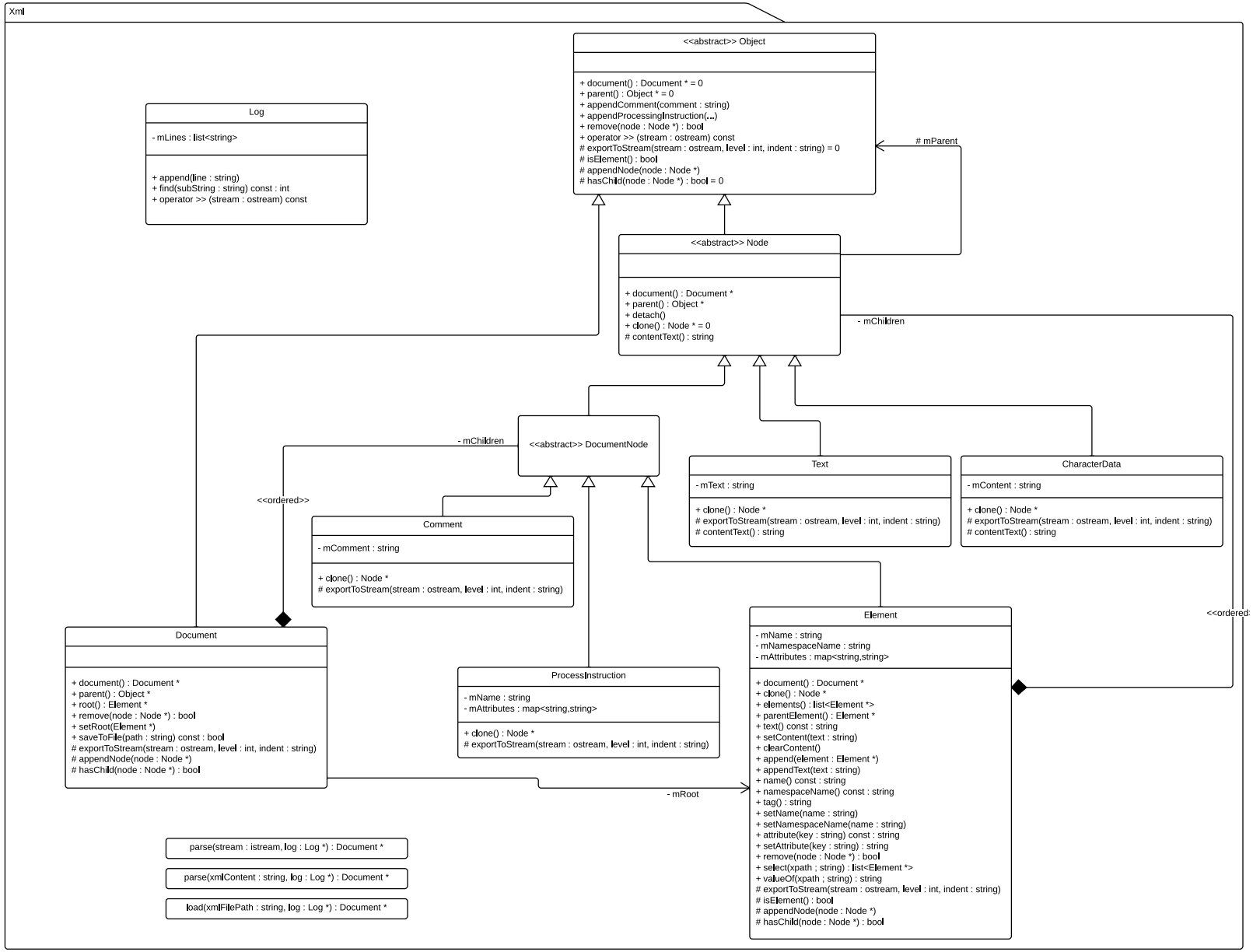


FIGURE 2.1 – Diagramme de classes de l'arborescence XML

2.2 Classes

2.2.1 Log

La class `Log` a pour unique mission de stocker l'ensemble de l'output du parseur *XML* telle que :

— Erreurs lexicales ;
— Erreurs syntaxique ;
— Erreurs semantiques.

2.2.2 Object

Pour optimiser et éviter au maximum la redondance de données dans notre arbre d'héritage de l'implementation *XML*, nous nous sommes inspirés de la très célèbre librarie *Qt* avec son `QObject`. Ainsi nous avons notre `Xml::Object` offrant les avantages que vous trouverez dans les sous-sections suivantes.

2.2.3 Node

Ainsi, nous partons du principe qu'un document XML n'est en réalité qu'un arbre ayant des noeuds (`Node`) étant des objects XML, mais de natures differents (commentaires, texte, élément ...). Certains de ces noeuds seront des feuilles de l'arbre (le noeud de commentaire par exemple).

Ici se trouve déjà un avantage du `Xml::Object` car chaque `Node` connait son `Object` parent à l'aide de `mParent` pouvant être un `Element` ou un `Document`. Ainsi, on peut à partir d'un noeud, retrouver le `Document` dans lequel il se trouve simplement en remontant l'arbre.

2.2.4 DocumentNode

Le `DocumentNode` est une spécialisation de `Node`, mais ayant seulement une particularité' sémantique : seul ses classes fillent peuvent avoir pour parent, un `Element` par hertiage de `Node`, mais aussi un `Document` au contraire de la classe `Text` ne pouvant avoir pour parent qu'un `Element`.

2.2.5 Document

Un `Document` est un `Node` composé de `DocumentNode`. Parmis ces `DocumentNode`, un seul et unique `Element` racine compose cette liste. Mais afin de pouvoir retrouver cette racine du document en complexité $O(1)$, `DocumentNode` dispose aussi d'un attributs `mRoot` dedié à cette tache.

Cette même liste de `DocumentNode` est ordonnée afin de garentir l'ordre des noeuds au chargement et à l'exportation du document *XML*. L'`Element` racine fait partie de cette liste afin d'éviter d'avoir à gerer deux listes de noeuds (ceux avant et ceux après).

2.2.6 Comment

`Comment` est un `DocumentNode` car il peut être n'importe où dans le document *XML* : dans un `Element` ou bien en dehors de l'élément racine du document.

2.2.7 ProcessingInstruction

Une `ProcessingInstruction` est aussi un `DocumentNode` d'après les spécifications officielles de *XML*. Mais en accord avec l'hypothèse énoncée précédemment, une `ProcessingInstruction` ne contient qu'un nom et une map d'attributs.

2.2.8 Text

Un noeud de texte est naturellement décrit par une chaîne de caractères `mText`. Celui-ci ne dérive pas de la classe `DocumentNode` puisque, si son contenu s'apparente à celui d'un noeud Comment, il ne peut en revanche être contenu par une instance de `Element`. Il est donc nécessaire d'ajouter le niveau de spécialisation `DocumentNode` afin d'éviter une relation ne respectant pas les spécifications standards *XML*.

2.2.9 Element

Un `Element` *XML* est un `DocumentNode`. Il possède une liste d'`Element` enfants `mChildren`. Cette classe peut aussi représenter la racine d'un `Document`.

Celui ci est defini avec un nom de balise (`mName`) mais aussi d'un espace de nom (`mNamespaceName`). La concatenation de ces deux derniers forment le tag (namespace + " :" + nom) de l'`Element`. Le nom du membre `mNamespaceName` au lieu de `mNamespace` a été effectué pour éviter la collision avec le mot clef du language C++ `namespace` avec son getter théorique `Xml::Element::namespace()` remplacer par `Xml::Element::namespaceName()`.

Un `Element` possède un ensemble non-ordonné d'attributs étant simplement une map ayant une association $clef \leftarrow valeur$. Par soucis de déterminisme, nous les exportons par ordre alphabétique des clefs.

Enfin un `Element` possède une liste ordonnée de noeuds (`Node`). Mais cette liste est inaccessible à l'utilisateur. En effet, nous ne voulons pas que l'utilisateur du namespace ait besoin de tester les différents types de noeuds. La méthode `Xml::Element::elements()` permet de récupérer l'ensemble des `Element` fils.

/* FIXME */

Mais il ne pourra pas recuperer les `Comment` car ne sont pas senser être parser. De même l'utilisateur ne pourra acceder aux `ProcessingInstruction` car sont seulement dedier au parser *XML*.

/* END FIXME */

L'utilisateur pourra accéder au contenu texte d'un `Element` à l'aide de `Xml::Element::text()` qui concatènera l'ensemble des noeuds enfants de type `Text`.

2.3 Encapsulation du namespace Xml

Au final l'utilisateur aura simplement accès aux classes suivantes :

— `Log`;
— `Object`;
— `Document`;
— `Element`.

Les autres classes sont implémentées avec des constructeurs `private` (avec des amitiés nécessaires entre elles bien entendu). Ainsi l'utilisateur peut réutiliser notre implémentation tout aussi facilement que la librarie `DOM` ayant pour avantage de ne pas avoir à gérer la présence de commentaires ou de processing instructions dans son code car ne travaillant dans l'arbre *XML* qu'avec `Document` et `Element`.

```
Xml::Document doc;

doc.setRoot(new Xml::Element("root"));
doc.root()->append(new Xml::Element("hi"));
doc.root()->appendComment("Hello world!");
doc.root()->append(new Xml::Element("bar", "foo"));

assert(doc.elements("hi").size() == 1);

doc.elements()[1]->setAttribute("attr", "myValue");

delete doc.elements("hi")[0];

std::cout << doc << std::endl;
/** stdout:
 * <root>
 *   <!--Hello world!-->
 *   <foo:bar attr="myValue"/>
 * </root>
 */
```

2.4 Algorithmes

2.4.1 Xml : :Element : :select

```
std::list<Xml::Element const *> Xml::Element::select(std::string const
& xPathQuery)const
```

Dans la suite de cette section, nous nous référerons à l'élément sur lequel nous appelons `Xml::Element::select` par "E".

Description

Cette méthode permet de récupérer la liste des éléments matchant une requête XPath.

Les requêtes XPath supportés sont les suivantes :
- *""* : Retourne une liste vide.
- *"/"* : Retourne une liste contenant la racine du document si E fait partie d'un document, une liste vide sinon.
- *"."* : Retourne une liste contenant E.
- *".."* : Retourne une liste contenant le parent de E si le parent est un élément, une liste vide sinon.
- *"bookstore"* : Retourne liste des éléments enfants de E qui ont pour tag *bookstore*.
- *"bookstore/book"* : Retourne la liste des éléments qui sont des *book* enfants de *bookstore* enfants de E.
- *"/bookstore/book"* : Retourne la liste des éléments qui sont des *book* enfants de *bookstore* qui sont enfants de la racine du document si E appartient à un document, une liste vide sinon.

Dans le cas où aucun élément ne match la requête, une liste vide est renvoyée.

Algorithme

Pour la suite, nous nous référerons à la requête XPath par "XP".

Le fonctionnement de l'algorithme `Xml::Element::select` est le suivant : On regarde si XP est égale à "", "/", "." ou ".." ce qui correspond aux 4 premiers cas triviaux cités précedemment. Si XP se trouve parmi ces 4 cas, le traitement est immédiat et retourne ce qui a été définis au-dessus.

Sinon, on regarde si XP possède un "/" :
- Si XP ne possède pas de "/", XP est un tag simple et on cherche, parmi les éléments de E, les éléments qui ont pour tag XP et on les renvoie dans une liste.
- Si XP possède moins un "/" mais ne commence pas par un "/", il s'agit d'un chemin relatif par rapport à E. Dans ce cas, on extrait la chaîne partant du début de XP jusqu'au premier "/" et on récupère tous les enfants de E qui possède ce tag. Ensuite, on appelle récursivement `Xml::Element::select` sur chacun des éléments avec pour requête XPath la sous-chaîne de XP commençant juste après le premier

"/" jusqu'à la fin de XP. On concatène ensuite toutes les listes récupérées via les appels à `Xml::Element::select`, puis on renvoie la nouvelle liste générée.
— Si XP possède au moins un "/" et commence par "/", on appelle `Xml::Element ::select` sur la racine du document contenant E avec pour requête XP privée de son premier "/". On se retrouve alors dans le cas précédent avec E étant la racine.

Dans le cas où aucun élément ne match la requête, une liste vide est renvoyée.

2.4.2 Xml : :Element : :valueOf

```
std::string Xml::Element::valueOf(std::string const & xPathQuery)const
```

Dans la suite de cette section, nous nous référerons à l'élément sur lequel nous appelons `Xml::Element::valueOf` par "E".

Description

Cette méthode permet de récupérer la valeur d'une requête XPath.

Les requêtes XPath supportés sont les suivantes :
— *""* : Retourne une chaîne vide.
— *"/"* : Retourne le texte contenu récursivement de la racine.
— *"bookstore"* : Retourne le texte contenu récursivement du premier élement enfant de E qui a pour tag *bookstore*.
— *"bookstore/book"* : Retourne le texte contenu récursivement du premier élement des *book* enfants de *bookstore* enfants de E.
— *"/bookstore/book"* : Retourne le texte contenu récursivement du premier élement des *book* enfants de *bookstore* qui sont enfants de la racine du document.
— *"@attr"* : Retourne la valeur de l'attribut *attr* de E.
— *"/@attr"* : Retourne la valeur de l'attribut *attr* de la racine du document ou une chaîne vide.
— *"bookstore/@attr"* : Retourne la valeur de l'attribut *attr* du premier élement enfant de E qui a pour tag *bookstore*.
— *"bookstore/book/@attr"* : Retourne la valeur de l'attribut *attr* du premier élement des *book* enfants de *bookstore* enfants de E.
— *"/bookstore/book/@attr"* : Retourne la valeur de l'attribut *attr* du premier élement des *book* enfants de *bookstore* qui sont enfants de la racine du document.

Dans le cas où un attribut ou un élément n'existe pas, une chaîne vide est renvoyée.

Algorithme

Pour la suite, nous nous référerons à la requête XPath par "XP".

Le fonctionnement de l'algorithme `Xml::Element::valueOf` est le suivant :

On regarde si XP est de la forme "", "/", "tag", "tag1/tag2" ou "/tag1/tag2". Dans ces cas là, on effectue un `Xml::Element::select` avec XP comme requête. On récupère

ensuite tout le texte contenu récursivement dans le premier élément du résultat de la requête.

Dans les autres cas, on cherche à récupérer la valeur d'un attribut. Notre approche est la suivante :

- Si XP est de la forme "@attr", on retourne la valeur de l'attribut *attr* de E.
- Si XP est de la forme "/@attr", on retourne la valeur de l'attribut *attr* de la racine du document.
- Dans les 3 cas restants ("tag/@attr", "tag1/tag2/@attr" et "/tag1/tag2/@attr"), on effectue un `Xml::Element::select` avec XP privé de "/@attr" comme requête. On récupère ensuite tout le texte contenu récursivement dans le premier élément du résultat de la requête.

Dans le cas où un attribut ou un élément n'existe pas, une chaîne vide est renvoyée.

2.4.3 Xml : :Element : :matches

```
bool Xml::Element::matches(std::string const & xPathQuery)const
```

3. XSD

Dans cette partie, nous expliquerons notre conception de la partie validation d'un document XML à partir d'un document XSD.

3.1 Hypothèses simplificatrices

Afin de simplifier l'implémentation du parseur XSD, les hypothèses suivantes ont été posées :

— Les types simples seront de type *string* ou *date*
— Les types complexes seront de type *sequence* ou *choice* et pourront contenir des attributs

3.2 Fonctionalités implémentées

Les points suivants sont pris en charge par le validateur XSD :

— Choix d'éléments racine multiples
— Présence et type des attributs
— Éléments de types *mixed*
— Types complexes contenant des éléments *sequence* et *choice* imbriqués
— Nombre d'occurences minimum et maximum par élément
— Références sur des attributs, éléments et types

3.3 Diagramme de classes

Les classes ici décrites seront rattachées au `namespace Xsd` défini en C++.

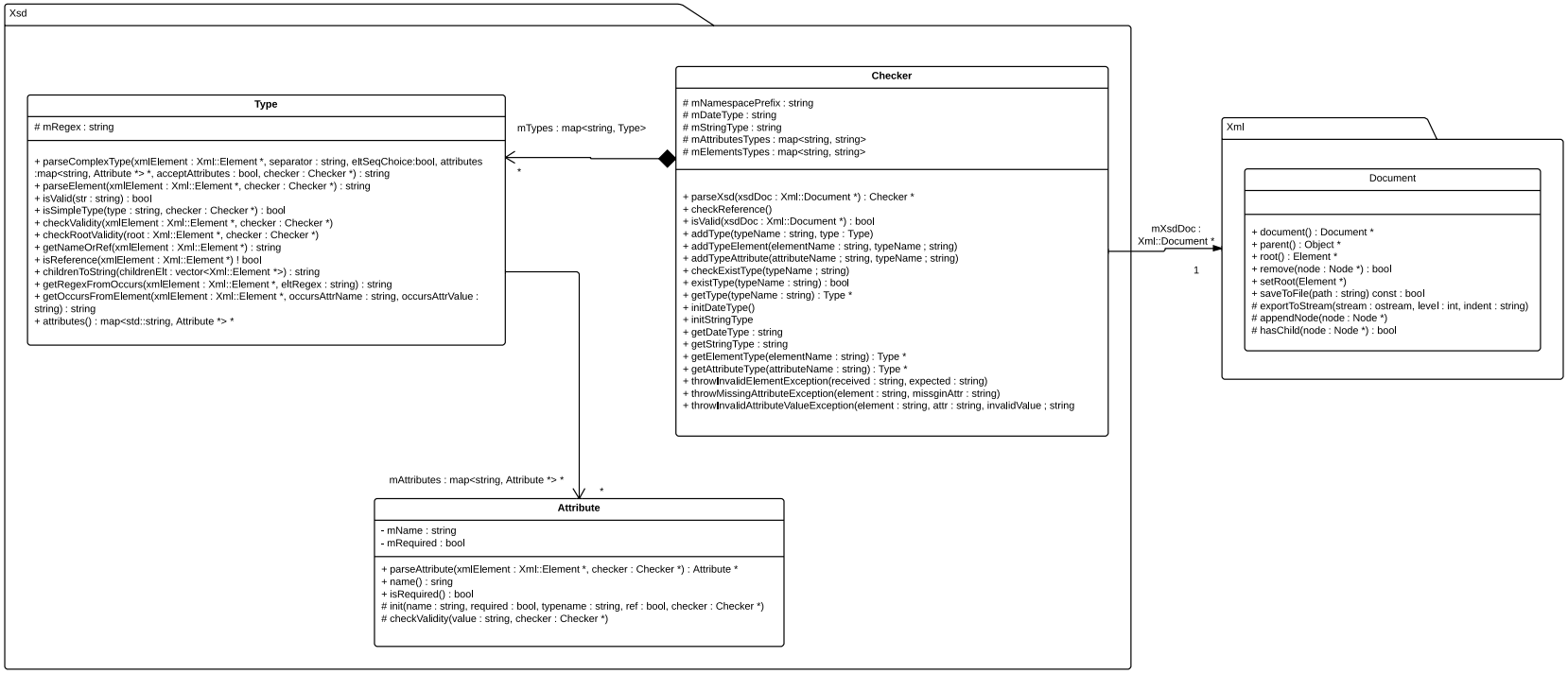


FIGURE 3.1 – Diagramme de classe de la validation xsd

3.4 Entités

3.4.1 Checker

Un objet Checker décrit les règles présentes dans un document XSD. Les méthodes principales sont les suivantes :

— **parseXsd** : méthode statique retournant un objet de type *Checker*, construit à partir du document XSD passé en paramètre
— **isValid** : retourne *true* si le document XML passé en paramètre est en accord avec les contraintes inscrites dans l'objet *Checker* courant, *false* sinon

La structure intermédiaire s'appuie sur 3 tables de hachages :

— **mTypes** : associe le nom (chaîne de caractères) d'un type avec son objet de type *Type*
— **mElementsTypes** : associe le nom d'un élément avec le nom de son type
— **mAttributesTypes** : associe le nom d'un attribut avec le nom de son type

3.4.2 Element

Un élément est une chaîne de caractères, associé [1] à un type dans la table de hachage *mElementsTypes* d'un objet de type *Checker*.

3.4.3 Type

On objet *Type* correspond à un type complexe XSD. Il est défini par deux éléments : *Expression régulière* : Cette chaîne de caractères détermine l'ensemble des fils directs que l'élément rattaché au type peut recevoir. Elle spécifie leur ordre et nombre d'occurences. **Exemple** : *(<fromage>)4|(<poisson>)1)* sera l'expression régulière du type associé à l'élément *pizza*. Elle indique qu'une pizza peut contenir 4 fromages ou un poisson. *Liste d'attributs* : Cette liste décrit les attributs que contiendra l'élément rattaché au type courant, l'ordre des attributs n'ayant pas d'importance.

3.4.4 Attribute

Un attribut est décrit par son nom et un attribut *required* à *true* ou *false*. Il est également associé à un type dans la table de hachage *mAttributesTypes*.

1. Dans la suite de cette partie, le terme *associer* correspondra à l'insertion d'une paire clé/valeur dans une table de hachage

3.5 Algorithme

3.5.1 Construction de la structure intermédiaire

L'algorithme présent construit un objet Checker, lequel contiendra un ensemble de types et attributs. Cet algorithme parcours en une passe l'arbre XML obtenu lors du parsing préalable du document XSD. Il s'agît d'un algorithme récursif, le parcours de l'arbre XML pouvant être assimilé à un parseur de type SAX sur le document XSD.

Parsing du document XSD : *parseXsd*

```
   Parsing du document XSD sous forme d arbre XML
Parsing du type de l element racine (c.f. parseComplexType)
[2] Association du "ROOT_TYPE"avec le type recu
Association du nom d element "ROOT"avec le type recu
Verification des references (c.f. checkReferences)
```

Parsing d'un élément *complexType* : *parseComplexType*

```
   SI l element complexType a un attribut mixed a true
   Ajout de ".*"au separateur d elements de la regex

   Parcours des fils de l element complextype
   SI c est un element sequence
      Appel recursif
   SI c est un element choice
      Appel recursif avec le separateur "|"
   SI c est un element element
      on ajoute a la regex de l element complexType celle de l element
retournee par parseElement()+ le separateur
   SI c est un element attribute
      Parsing de l attribut (c.f. parseAttribute)
      Ajout de l attribut a la map d attributs du type courant

   On retourne la regex associee au type
```

Parsing d'un élément *element* XSD : *parseElement*

```
   Construction d une regex decrivant l element en fonction de ses
occurences
```

2. L élément *schema* est implicitement de type complexe, ses éléments fils sont traités comme s ils étaient dans un élément *choice*

[3] SI ce n est pas une reference
SI l element a des enfants (c est un type complexe)
Construction d un type avec le premier fils de l element
[4] Association du nom du type avec le type construit
Association du nom de l element avec le nom du type construit
On retourne la regex associee a l element

Parsing d'un élément *attribute* : *parseAttribute*

SI les attributs nom et type sont definis, et que c est un type simple
Association du nom de l attribut avec le nom du type
On retourne un nouvel attribut a partir d un nom et d un booleen required

Vérification des références : *checkReferences*

Pour chaque element
On verifie que l element est rattache a un type dans mElementsTypes
Pour chaque type
Pour chaque attribut du type
On verifie que l attribut est rattache a un type dans mAttributesTypes

3. Par exemple, pour <xsd :element name="parfum" type="xsd :string" minOccurs="1" maxOccurs="2"/> on generera l expression (<pizza>)1((<pizza>) ?)1
4. Le constructeur de type appelle *parseComplexType*

3.5.2 Validation d'un document XML

Validation d'un type : *checkValidity*

Cette méthode prend un élément XML en paramètre, et valide les attributs qu'il contient et l'architecture de ses fils directs. L'appel initial est donc fait sur le type associé au nom de l élément root XML

```
   On verifie que l ensemble des attributs de l element XML existent dans
 la map des attributs du type XSD
Pour chaque attribut XSD du type XSD
   Si l attribut XSD existe
      On verifie que sa valeur valide l expression reguliere de son type
   Si l attribut XSD n existe pas
      On verifie qu il n est pas requis


   Generation d une chaine de caracteres correspondant a la concatenation
 des balises de ses elements fils
On verifie que l expression reguliere du type valide cette chaine de
caracteres


   Pour chaque element fils de l element XML recu
   Recuperation du type associe au nom du fils
   Appel recursif a la fonction checkValidity sur le type obtenu
```

4. XSL

4.1 Conception

Dans un premier temps, nous avions choisi de suivre une conception orientée objet, et c'est donc naturellement que nous avons proposé une classe dédiée aux documents *XSL*, proposant les fonctionnalités de transformation d'une feuille *XSL*, et de même pour les instructions *XSL*.

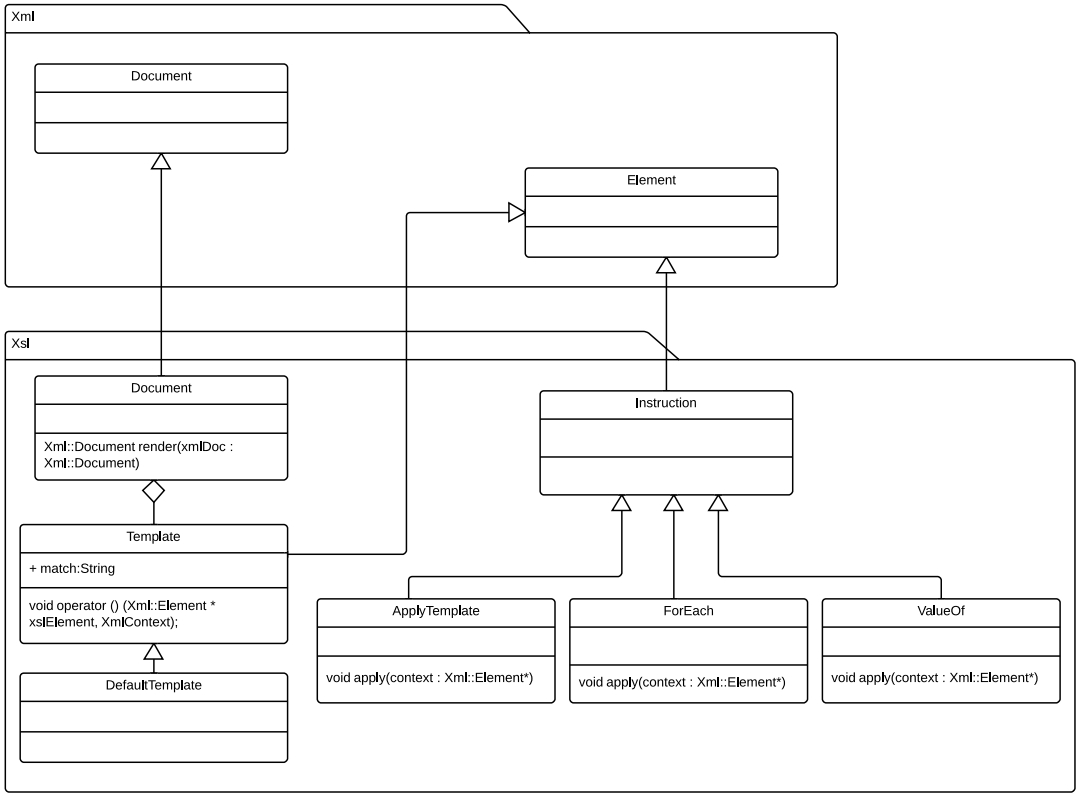


FIGURE 4.1 – Ancienne conception du module XSL

Néanmoins, cette conception présente de nombreux inconvénients : En plus d'ajouter de nombreuses classe sans vraie valeur ajoutée, elle nous forçait à reconstruire l'arbre

XML afin de convertir les tags XSL en objets dédiés. On ne profitait pas réellement d'un quelconque polymorphisme non plus puisque cela demandait de toucher aux classes XML, alors que celles ci n'ont pas comme responsabilité de gérer toutes les applications possibles de ce langage de représentation de données. Il aurait aussi fallu tester le namespace des éléments *XML* parcourus avant de les caster à la bonne classe, donc pas grand d'intérêt au "polymorphisme".

Nous avons donc choisis de passer à une architecture plus simple : pas d'héritage, enfin si. Mais d'aucune classe du `namespace Xml`. Simplement une map constante faisant officie de virtual table, la liaison entre le nom des instructions *Xsl* et des foncteurs qui génèrent des nœuds *XML* à partir d'un contexte donné.

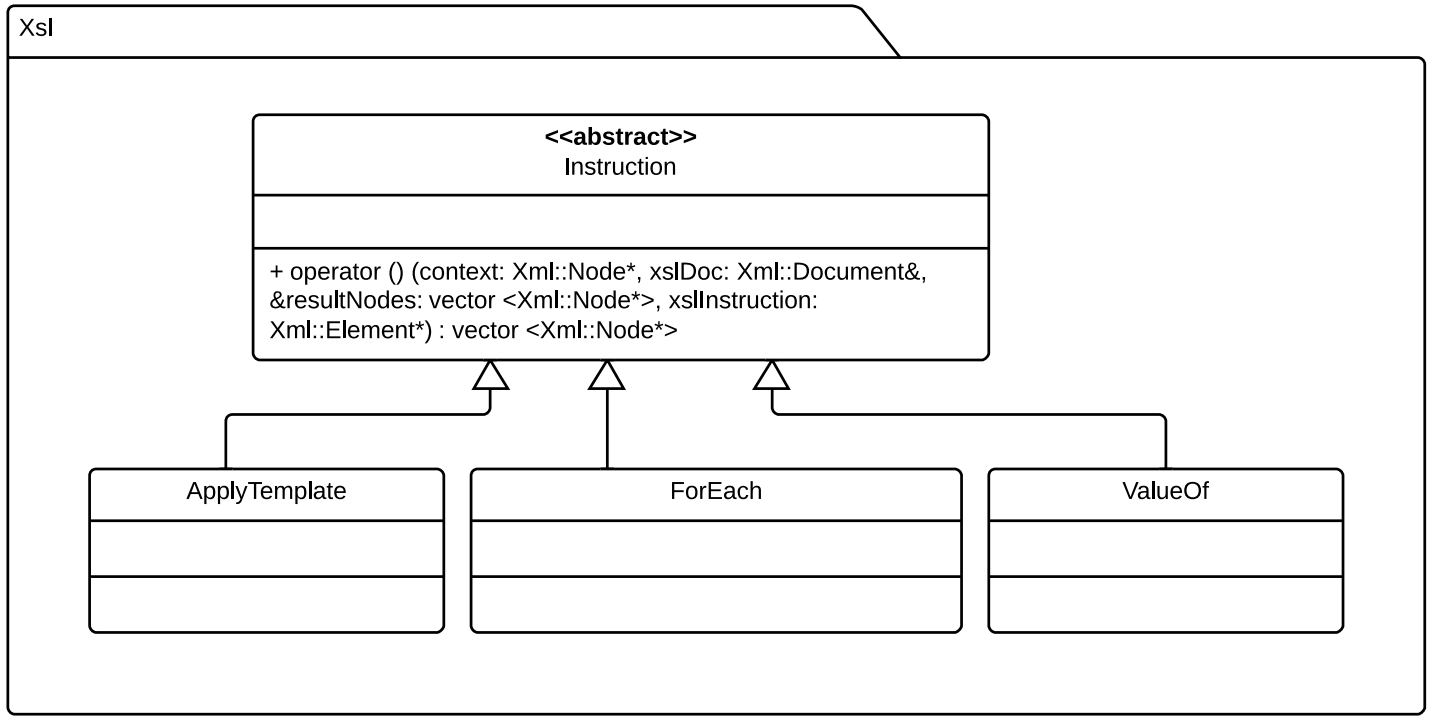


FIGURE 4.2 – Diagramme de classe des instructions XSL

4.2 Vue globale de l'algorithme

L'algorithme de transformation *XSL* s'exécute de manière récursive et bascule continuellement entre le document *XSL* et le document *XML* à transformer.

Un même algorithme s'exécute pour chaque nœud (à quelques exceptions citées ci-dessous) :

1. Si le nœud n'est pas un élément, on le rajoute directement au résultat.
2. Si un *template* correspond à ce nœud, on l'applique (plus sur l'application des templates ci-dessous) en prenant ce nœud comme *contexte* et on ajoute le résultat de cette application au nœud en cours de génération.
3. Sinon, on ré-applique le même algorithme sur tous ses enfants, en ayant comme contexte l'élément actuel et en concaténant les résultats des applications de l'algorithme à tous le fils.

La transformation *XSL* débute en appliquant cet algorithme à la racine du document *XML* à transformer, et se propage par récursions successives à la totalité du document.

4.3 Templates

Les *templates* sont les seuls fils de la racine (*stylesheet*) d'un document *XSL* et se présentent de la manière suivante :

```
<xsl:template match="cd/title">
    <tagxml></tagxml>
    <xsl:uneinstruction select="untag/unautretag"/>
</xsl:template>
```

Ils peuvent contenir des tags *XML* "normaux", ainsi que des instructions XSL. On dit qu'un template *match* un élément si l'élément est compatible avec la valeur de l'attribut *match* du template, par exemple :

Si un élément XML a comme nom "unautretag" et est le fils d'un élément qui a comme nom "untag", il match le template vu ci-dessus.

Si un élément *match* deux templates différents, on choisira celui qui est le plus spécifique. Par exemple : s'il y a deux templates *"catalog/cd"* et *"cd"*, c'est le premier qui sera appliqué pour les éléments "cd" fils de "catalog".

Quand on applique un template, on va renvoyer une liste de nœuds résultants (qui vont en général être ajoutés comme fils d'un document ou d'un élément). Cette liste est générée de la manière suivante :

1. Les nœuds qui ne sont pas des éléments *XML* sont rajoutés directement.
2. Les nœuds qui sont des éléments *XML*, mais pas *XSL*, sont clonés. On applique alors puis considérés comme des templates (car pouvant contenir des instructions

XSL) et appliqués avec comme contexte le nœud d'application de la transformation *XSL*.

3. Les éléments *XSL* sont appliqués avec comme contexte le nœud d'application de la transformation *XSL*. Tous les nœuds résultants de cette application sont ajoutés à la liste des nœuds générés par l'application du template.

4.4 Instructions XSL

4.4.1 apply-template

Apply-template consiste à appliquer un template au noeud correspondant au chemin indiqué par l'expression XPath de l'attribut "select". Ce template doit se trouver dans la feuille de style Xsl que l'on est en train de parcourir pour transformer le document Xml. Si l'attribut select vaut "X" et qu'il existe dans le Xml le noeud correspondant au chemin de X dans le contexte courant, il doit donc exister dans le Xsl :

```
<xsl:template match="X">
    ..........
    ..........
</xsl:template>
```

. L'idée est alors de réappliquer le template trouvé à l'élément correspondant à X depuis le contexte courant de lecture du xml. Le template est appliqué et relance le traitement du Xml par l'appel des fonctions d'application de template habituelles. L'attribut "select" est optionnel, dans le cas de son absence, il faut alors chercher et appliquer les templates correspondants à tous les Noeuds Xml du contexte Xml courant, non récursivement. C'est à dire que si ces enfants ont des enfants, apply-templates ne forcera pas l'application des templates leurs correspondants

4.4.2 value-of

Value-of est une instruction Xsl permettant de retourner la valeur textuelle des éléments contenus au chemin indiqué par l'expression XPath requise comme paramètre de "select". Cette instruction procède par récursion, c'est à dire que même les éléments textuels des éléments enfants sont affichés. Tout les textes retournés sont concatainés à l'emplacement de l'instruction Value-of. Value-of permet également d'afficher la valeur d'un attribut d'un noeud. En effet, en précisant @nomdelattribut en bout de chemin de l'expression XPath, la valeur de l'attribut appartenant à l'élément correspondant à l'expression XPath sera affichée.

4.4.3 for-each

For-each permet d'appliquer le templates contenus dans l'élément "for-each" à tous les noeuds dont le chemin est celui de l'expression XPath indiqué dans l'attribut "select" de

for-each. Pour cela, il faut récupérer dans le document Xml tous les noeuds correspondants au XPath, puis appliquer à chacun le template contenu entre les balises for-each, comme ci c'était un template normal dans le déroulement de nos algorithmes. On retourne ensuite tout cela pour que ce soit ajouté à notre document transformé.

4.5 Limites

Bien qu'elle couvre les instructions XSL les plus courantes, avec un support extensif de nombreux cas spéciaux, notre implémentation reste relativement limitée pour les raisons suivantes :

1. Spécifications assez massives et parfois ambiguës (sur le concept)
2. Grand nombre de cas particuliers / cas limites
3. Quelques incohérences (comme le pseudo-XPath utilisé pour l'attribut match)
4. Transformations complexes

Parmi ces limitations :

1. Pas de support pour des instructions XSL assez utiles, comme xsl :copy-of
2. Algorithme faillible à des boucles infinies causées par des appels infinis d'apply-template
3. Mauvais support des opérations de sélection depuis la racine (parce que le pseudo-XPATH utilisé en XSL considère "/" comme le document XML, et non sa racine)