Projet GL – Document de conception

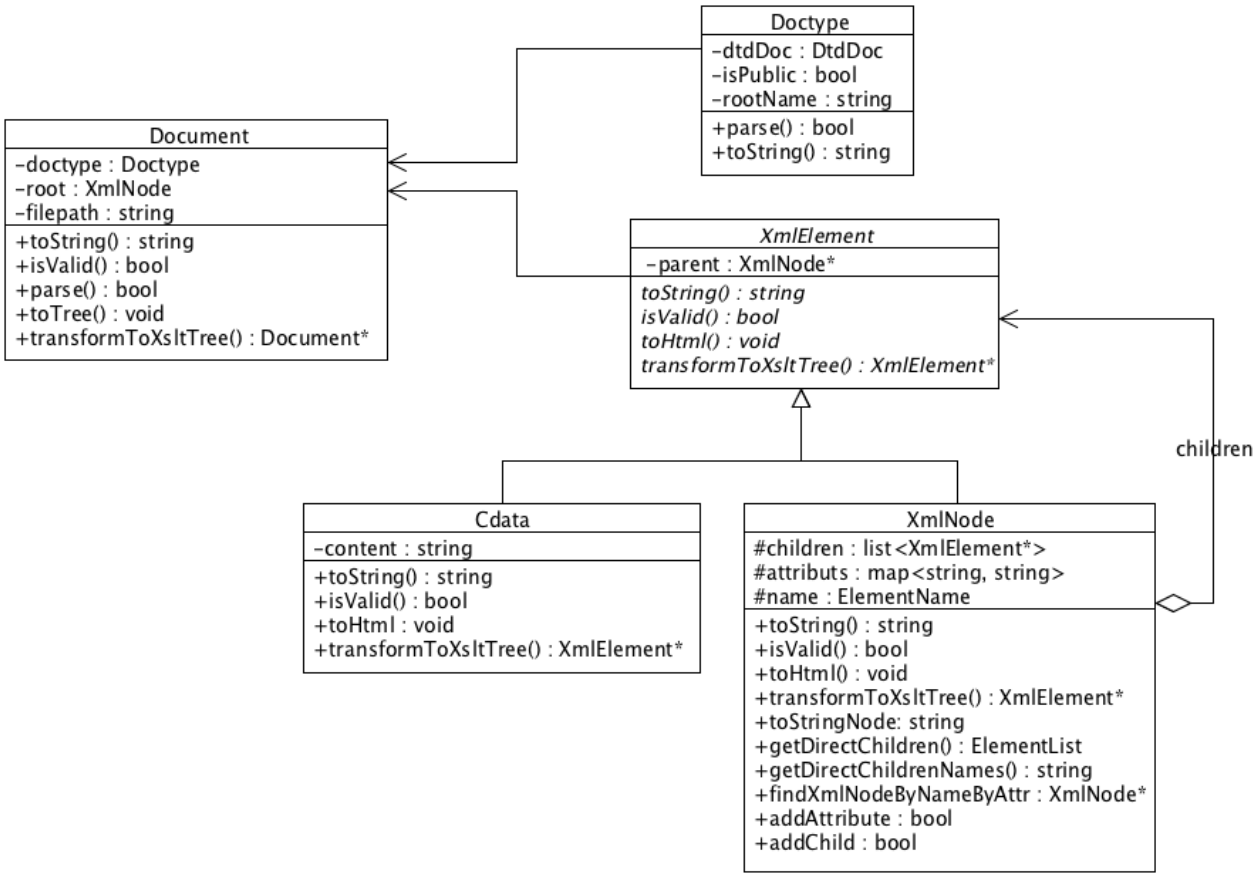
Modèles

Document XML

La représentation en mémoire d'un document XML fait appel à 5 classes :

- Document, 1er objet créé, représentant le fichier parsé
- Doctype, 1 doctype par document, qui va contenir la représentation mémoire de la DTD
- XmlElement, classe abstraite, représentant un élément XML, deux formes sont possibles :
    - XmlNode, nœud qui a éventuellement des attributs et/ou contient d'autres éléments
    - Cdata, character data, une chaîne de caractère non interprétés

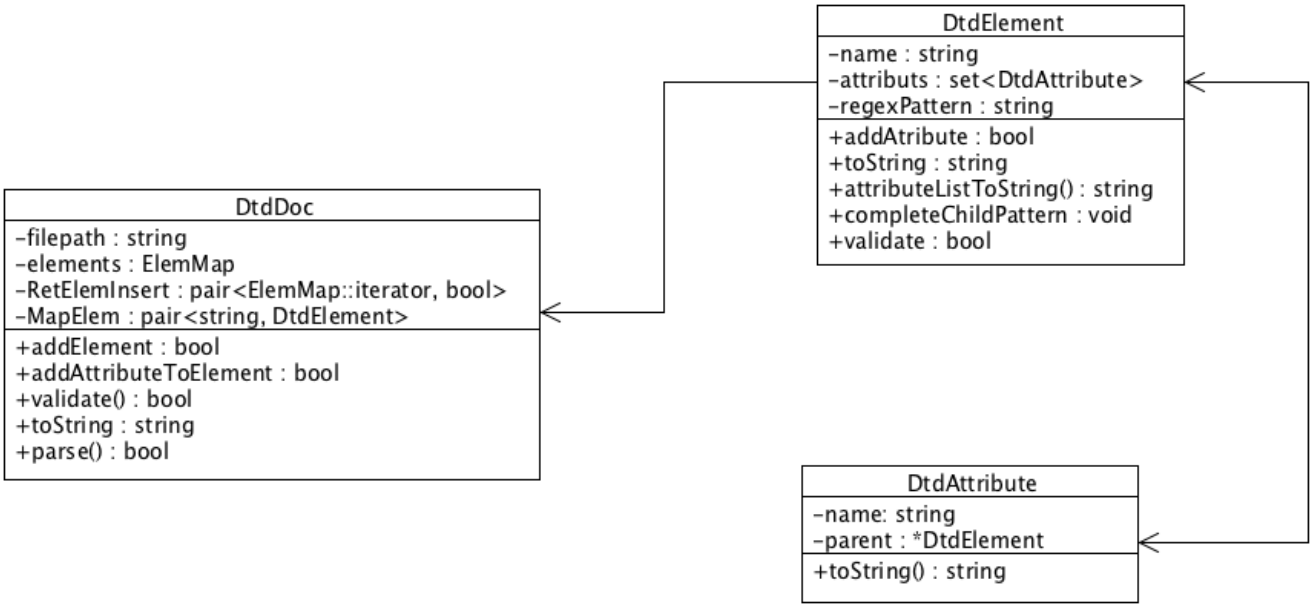
Afin d'optimiser notre code, nous avons donc utilisé le design-pattern du composite.
Il est possible de représenter nos classes ainsi :



Document DTD

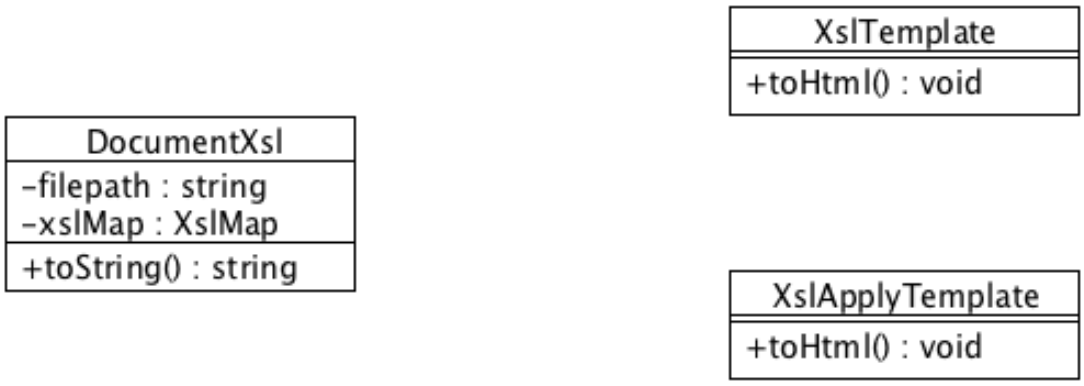
La représentation d'un document DTD est beaucoup plus simple. Nous avons ainsi :

- un objet DtdDoc représentant le document, racine, qui est pointé par l'attribut dtdDoc du Doctype du document XML. Il contient une liste de DtdElement.
- des éléments DtdElement, représentant des éléments (!ELEMENT), qui ont des liste d'attributs et ont leur contenu stocké sous forme d'une regex
- des éléments DtdAttribute, qui ont un nom, représentant les attributs (!ATTLIST)



Document XSLT

La représentation d'une telle document est plus simple : nous avons un élément « DocumentXsl », correspondant au fichier, et les différents templates, qui peuvent être appliqués par une méthode toHtml().



Algorithme de validation

Cet algorithme est utilisé pour valider le fichier XML par rapport à sa DTD.
Le parse du fichier XML est lancé, si le fichier est bien formé (par d'erreur de caractère, tag mal fermé…), il est représenté en mémoire. Le doctype sera détecté, et la DTD correspondante parsée.
La fonction document->isValid() est ensuite lancée manuellement.

Fonction Document->isValid() :

```
// on vérifie que le 1er nœud est bien du type défini par le Doctype
Si document->root.getName() != doctype.getRootName()
        retourne FAUX
FinSi

document->root->isValid() // on s'appelle sur le XmlElement racine
```

Fonction XmlElement->isValid() :

Si s'agit d'un XmlNode :

```
// fonction validate(attMap) : on vérifie si les attributs son autorisés
Pour (unAttribut dans attMap)
    Si unAttribut introuvable dans DtdElement->attributs
            retourne FAUX
    FinSi
FinPour

// fonction validate(getDirectChildren()) : on vérifie si les éléments-fils sont autorisés
// on compare via Regex si le nom des éléments-fils est dans la liste (enum ou choice) du DTD

Pour (XmlElement élément dans children)
    Si élément->isValid() == false // appel récursif
            retourne FAUX
    FinSi
FinPour

retourne VRAI
```

S'il s'agit d'un Cdata :

```
retourne VRAI // character data, si bien délimité forcément valide
```

Algorithme de transformation

Cet algorithme, de transformation d'arbre repose sur deux structures XML : la structure XML à transformer en structure HTML et la structure XLST permettant de donner les informations de traitements (mise en page html, mise en forme conditionnelle…).

Après avoir posé les bases du fonctionnement du traitement XSLT ainsi que la délimitation du cadre de notre projet, nous avons mis en place un algorithme travaillant sur les deux structures XML pour pouvoir réaliser le même traitement que l'application directe du XSLT.

La transformation de la structure XML en arbre HTML commence par le parcours de l'arbre XML. Pour chaque noeud rencontré, nous allons vérifier dans la structure XSLT si un modèle ("template") existe pour ce noeud.

Plutôt que de réécrire un algorithme de validations pour s'assurer que le motif d'acceptation des enfants était bien respecté, nous avons simplement transposé le motif décrit dans la syntaxe DTD en une regex compatible avec un moteur de regex traditionnel (dans notre cas, Boost::Regex).
Ces syntaxes étant très proches, nous avons simplement eu à remplacer les virgules séparant les noms des enfants, par des espaces. Il ne reste ainsi plus qu'à générer une liste des enfants directs d'un noeud, la transformer en chaîne (séparée par des espaces), et de tenter d'appliquer la regex générée avec cette liste d'enfants. Si le match est positif, le motif est respecté.
En utilisant cette méthode, nous avons pu diminuer la complexité du programme, tout en se reposant sur des bibliothèques portables et répandue (Boost a été intégré dans C++ 11).

Pseudo-code

Méthode toTree() du Document *(toTree() pour transformation d'arbre)*

On ouvre le fichier de sortie

Si l'ouverture est OK Alors
    On lance la fonction toHtml() de la racine de notre structure xml
FinSi

Méthode toHtml() des noeuds XML

Si un template correspond à ce nœud dans la structure XSL Alors
    Si le nœud n'est pas la racine XML Alors
        On applique toHtml() pour chaque fils du noeud XML
    FinSi
Sinon *// (pas dans la structure XSL)*
    On récupère le noeud XSL correspondant (template)
    On lance la fonction toHTML() de ce noeud
FinSi

Méthode toHtml() du noeud XSL "Template"

On lance la fonction toHTML de tous les fils de ce noeud XSL *(pouvant être des noeuds XML normaux ou XSL)*

Méthode toHtml() du noeud XSL "Apply-Template"

Récupérer noeud XML pour lequel on a appliqué le template

Pour chaque enfant de ce noeud Faire
    Fonction toHtml() */* Chaque enfant est un noeud de la structure xml que l'on va afficher */*
FinPour

Méthode toHtml() de l'élément XML Cdata

On écrit dans le fichier en sortie le contenu