Projet Xmlia

BOIVIN Benoit
LE PHILIPPE Noé
KEGBA-SANGO-SANGO Ulrich-Chancelin
WOUTERS Stéphane

23 mai 2014

Résumé

A remplir !

Remerciements

Nous remercions tout particulièrement Michel Meynard pour nous avoir brillamment encadré et soutenu tout le long de la réalisation de ce projet.

Table des matières

Chapitre 1

Introduction

Chapitre 2

Analyse du projet

2.1 Contexte

Le langage Extensible Markup Language ou XML est utilisé à des fins de stockage de données, et est structuré par un schéma qui lui est associé, il permet de definir la structure et le type de contenu du document, en plus de permettre de verifier la validité du document.

Généralement, les fichiers XML sont générés par un programme quelconque dans le but d'échanger des données ou de les stocker, XML faisant office de plateforme commune. Mais on peut également utiliser un éditeur de texte basique pour créer de toute pièce un document XML, avec des fonctionnalités propre à un éditeur de texte, sans fonctionnalités spécialement prévues pour XML.

C'est dans ce contexte que des solutions logicielles d'éditeur XML ont vu le jour : un éditeur de texte qui possède des fonctionnalités permettant une écriture d'un fichier XML beaucoup plus rapide et efficace, le tout avec un contrôle des erreurs.

2.2 Analyse de l'existant

Plusieurs solutions sont déjà proposées, certaines étant payantes et d'autres sont gratuites et open source. L'objectif est ici de fournir une solution similaire aux autres logiciels.

En se basant sur le logiciel le plus pertinent d'après les recherches autour du sujet "xml editor", le logiciel oXygen XML Editor semble être le plus présent et utilisé. C'est une solution logicielle contenant deux principaux outils : XML Developer et XML Author. Le tarif pour le package complet de XML Editor est au prix de 488$ ou 19$ par mois là ou chaque outil coûte à l'unité 349$, ce qui en fait un logiciel très onéreux. L'entreprise propose cependant une version d'essai de 30 jours pour tester le logiciel. Le programme propose un nombre très important de fonctionnalités dans le but d'être utilisable à la fois par un développeur qui connaît déjà le langage et qui voudrait optimiser la saisie de fichiers XML et par un novice de XML avec un système d'assistants de création de documents.

Par exemple, la figure suivante expose une vue spéciale du document sous forme de tableur, permettant une modification beaucoup plus aisée des attributs des nœuds.

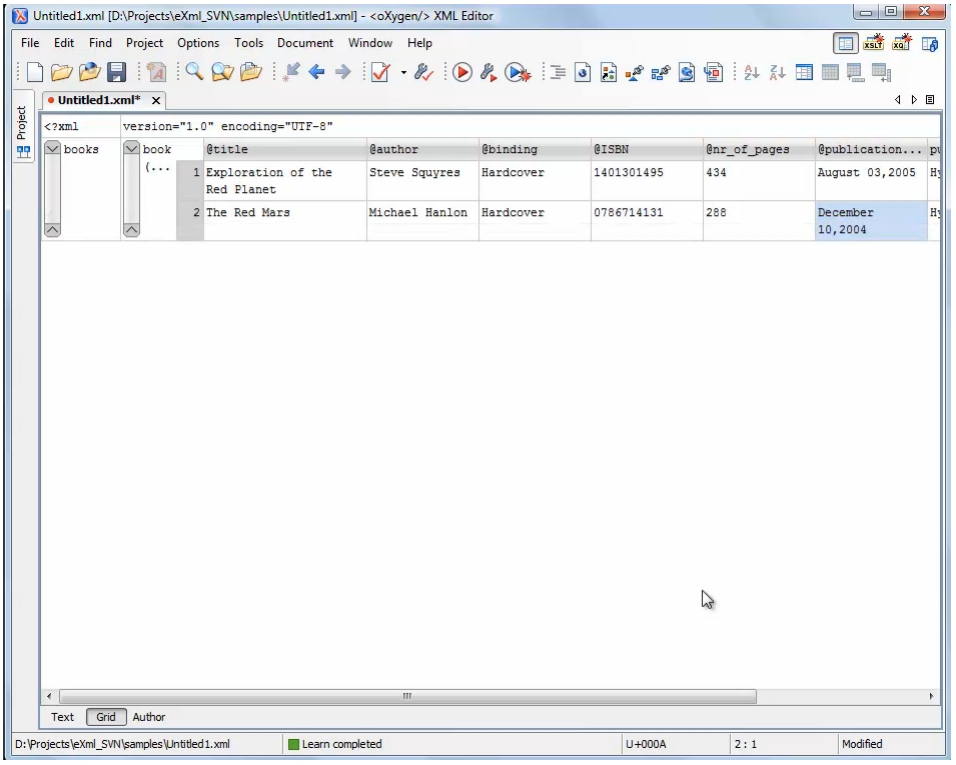

FIGURE 2.1 – Maquette qsdqdsqsdd'interface.

2.3 Analyse des besoins fonctionnels

L'objectif du projet est de développer un éditeur XML multi-vues avec différentes fonctionnalités.

Les fonctionnalités liées à un éditeur de texte simple devront être présentes : la possibilité de saisir manuellement au clavier l'intégralité du fichier, la création et la sauvegarde du fichier à manipuler ainsi que l'ouverture d'un fichier déjà existant dans le but de le modifier.

Des fonctionnalités d'éditeur de texte avancées seront aussi présentes : coloration syntaxique et indentation automatique du code permettant ainsi une lisibilité claire des fichiers manipulés et une autocomplétion du code écrit permettant un gain de temps au cours de la frappe.

Pour finir, l'éditeur proposera des fonctionnalités spécifiques au langage XML : validation syntaxique du fichier, vue arborescente du fichier XML avec possibilité de modification des données via cette vue et ajout d'un schéma sur lequel la validation se basera.

La figure suivante expose la maquette d'interface de l'éditeur avec chacune des parties annoncées : à gauche la vue arborescente affichant chaque nœud de l'arbre formé par les données, à droite l'éditeur de texte avec toutes ses fonctionnalités associées et en haut la barre d'outils avec la gestion de fichiers (nouveau fichier, ouvrir, sauvegarder) ainsi que la gestion du schéma.

2.4 Analyse des besoins non fonctionnels

2.4.1 Spécifications techniques

Le programme devra permettre de créer des fichiers XML structurés avec un respect des normes de balisage et, s'il est défini, du schéma de données. De plus, les données saisies ou modifiées à l'aide de l'éditeur doivent rester exploitables, sans corruption du fichier original. Pour terminer, l'éditeur aura à répondre dans des durées acceptables et de manière stable, dans la mesure où la taille et la complexité des données restent raisonnables.

2.4.2 Contraintes ergonomiques

Le logiciel devra être suffisamment simple pour qu'un utilisateur connaissant déjà le fonctionnement du XML puisse l'utiliser sans être bloqué par une courbe d'apprentissage trop élevée. On utilisera pour cela des icônes claires et des textes explicatifs. Un utilisateur avancé pourra augmenter sa productivité en utilisant les raccourcis clavier disponibles et pourra gagner de temps en réduisant les transitions souris/clavier.

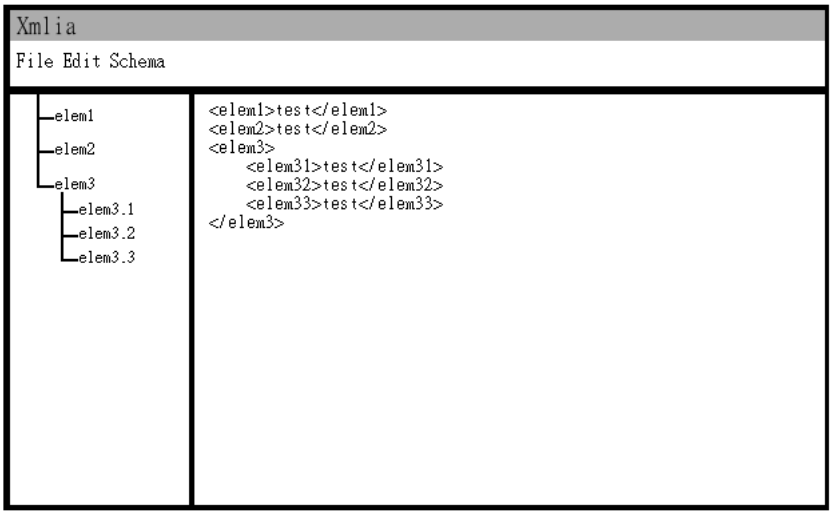

FIGURE 2.2 – Maquette d'interface.

Chapitre 3

Rapport d'activité

3.1 Organisation du travail

3.1.1 Commmunication

La communication s'est au départ faite au travers de réunions hebdomadaires au cours desquelles le cahier des charges a été défini auprès de M. Meynard.

Une fois le cahier des charges défini et rendu, le but suivant a été de déterminer quel langage et quel bibliothèque d'affichage graphique sélectionner pour le projet et ainsi être conscient des avantages et des limites des éléments choisis.

Ensuite, l'objectif suivant a été de définir une maquette du logiciel afin de savoir quelles seront les fonctionnalités retenues, la manière de les exploiter et quel organisation visuelle du logiciel sera retenue. C'est ainsi qu'a été définie la maquette qui comporte une interface simple avec une barre d'outils, une vue principale sur le côté droit avec le système d'éditeur de texte, une vue secondaire sur la gauche avec la vue arborescente du modèle XML du fichier présenté par l'utilisateur et enfin la fenêtre de log associé au différents messages liés à l'utilisation de l'éditeur, par exemple, des erreurs de schéma.

Il fallait enfin définir l'organisation du travail au sein du groupe avec des tâches définies et mettre un place un plan de travail, c'est donc Stéphane WOUTERS, le chef du projet, qui a mis en place le gestionnaire de projet, a décidé des moyens de communication et qui a défini le premier objectif de développement : l'apprentissage de la librairie.

Puis la phase d'apprentissage et de développement commença, il fallait alors découvrir et apprendre le framework utilisé, et commencer à développer pour le projet, la communication s'est donc faite au travers de messages sur Google Hangouts, de mails, de commentaires via le gestionnaire de projet et de communication orale dans une salle informatique de la faculté.

3.1.2 Répartition des tâches

3.2 Outils de développement

3.2.1 Langage et bibliothèques

Nous avons utilisé Qt comme bibliothèque d'interface graphique.

3.2.2 IDE

Nous avons naturellement utilisé Qt Creator comme IDE, c'est en effet l'IDE

3.2.3 Gestionnaire de projet

Le gestionnaire de projet utilisé est Bitbucket, un site Internet d'hébergement mutualisé supportant des projets utilisant Mercurial ou Git comme gestionnaire de versions. Dans le cas de Xmlia, Git a été retenu et utilisé.

Git a ici permis de gérer les accès et les mises à jour différents fichiers du projet, qu'ils soient du code ou du texte brut, comme par exemple pour le rapport du projet. Son utilité aura donc été de permettre une gestion des fichiers de manière formelle, avec des gestions de conflits de versions de fichier, par exemple avec un système de gestion de fichiers basique comme un FTP, si deux développeurs travaillent en accès concurrentiel sur le même fichier, chacun aura alors sa version du fichier et au moment de renvoyer le travail effectué sur le serveur FTP, un conflit de version surviendra, c'est pour cela que Git est utile en mettant en place des barrières empêchant ce genre de problèmes et proposant des solutions pour, par exemple, réunir les deux fichiers et qu'un développeur s'occupe de "merge" ces deux fichiers en un seul qui aura alors le code des deux développeurs. Un dernier point important à aborder est le fait qu'un envoi de code sur le serveur peut être annulé si par exemple une erreur d'utilisation de Git a été faite et que des fichiers auraient alors été modifiés rendant le projet non fonctionnel.

Bitbucket est un service accessible depuis une page Web permettant la gestion de projet Git. Le service propose donc un serveur Git fonctionnel avec une interface Web très performante. Cette interface permet de gérer tous les projets auxquels on est rattaché, créer de nouveaux projets et gérer les droits liés à ces projets. Un nouveau projet a donc été créé sur le site au travers de l'interface puis les droits de modification du projet ont été données aux autres membres du groupe projet qui ont alors pu commencer à utiliser le Git du projet. Ajouté à cela, Bitbucket liste l'intégralité des différentes opérations effectuées sur le Git permettant un regard global et rapide sur toutes les réalisations et offre aux différents membres du groupe la possibilité d'écrire des commentaires sur chaque opération, les membres seront donc notifiés par e-mail de ce changement, ce qui aura été un moyen de communication très présent au cours de la phase de développement. De plus, une fonctionnalité très importante de Bitbucket pour la gestion de projets est le système de tickets qui est intégré à l'interface Web où chaque membre peut ajouter soit une tâche à effectuer ou un bug à corriger et l'affecter à un autre membre, cela permet alors d'avoir une communication plus claire et concise, donnant un regard des autres membres sur les tâches effectuées par le groupe, aussi en leur donnant la possibilité de commenter ces tickets et d'ajouter des informations nécessaires à la réalisation de la tâche, donnant alors un point de vue global sur l'avancement du projet.

Par exemple, l'illustration suivante montre un ticket concernant une fonction à coder avec une description précise et des commentaires afin de mettre l'accent sur le sens de la tâche à réaliser, afin d'avoir une compréhension plus claire du problème.

FIGURE 3.1 – Exemple de ticket sur Bitbucket.

Chapitre 4

Rapport technique

4.1 Conception

4.2 Architecture de l'application

4.2.1 Le modèle

4.2.2 L'arborescence

4.2.3 L'éditeur de texte

L'éditeur de texte est décomposé en deux parties, l'éditeur de schéma et d'éditeur XML. Ce sont en réalité des spécialisation de la classe TextEditor, mais parlons d'abord de l'éditeur de texte en général.

La classe TextEditor

C'est ici que sont toutes les méthodes servant à l'édition du texte en général, telle que l'insertion de texte, l'indentation ou la coloration. L'éditeur de texte est une sous classe de QTextEdit, qui fournit toutes les fonctionnalités basiques d'une éditeur de texte riche, telles que la sélection de texte, la copie, l'annulation etc. On peut intégrer plusieurs fonctionnalités à un QTextEdit comme par exemple la coloration syntaxique. Nous parlerons dans un premier temps des fonctionnalités propres à notre éditeur de texte, puis nous verrons plus en détail chacune de ses implémentations, à savoir l'éditeur XML et l'éditeur de schéma.

Qt propose un QCompleter, il ne peut cependant pas s'intégrer et proposer l'autocompletion dans un QTextEdit. Même s'il ne s'intègre pas dans un QTextEdit, le QCompleter peut tout de même proposer une suggestion si on lui fourni un début de mot. On peut donc récupérer le mot sous le curseur, le donner au QCompleter et insérer le mot complété dans notre éditeur de texte. Cela se fait de la manière suivante :

```
//on donne le mot sous le curseur au QCompleter
completer->setCompletionPrefix(textUnderCursor());
//on recupere le mot propose
QString completion = completer->currentCompletion();
QString currentWord = textUnderCursor();

//si un mot est sous le curseur et qu'il n'est pas deja complete
if(currentWord.size() > 0 && currentWord.size() < completion.size())
{
```

```
    QTextCursor cursor = text->textCursor();
    //sauvegarde de la position actuelle du curseur
    int pos = cursor.position();
    //deplacement jusqu'a la fin du mot
    while(word.indexIn(cursor.selectedText()) == 0)
    {
      cursor.movePosition(QTextCursor::Left, QTextCursor::KeepAnchor);
    }
    //insertion de la partie qui n'est pas deja ecrite
    cursor.insertText(completion.right(completion.size() - currentWord.
        size()));
    //le curseur est replace a sa posision originelle
    cursor.setPosition(pos, QTextCursor::KeepAnchor);
    text->setTextCursor(cursor);
```

Il n'est pas possible d'intégrer directement l'auto-completion, mais on peut en revanche facilement lier un colorateur syntaxique à un QTextEdit. Qt propose une classe abstraite QSyntaxHighLighter qu'il faut implémenter en y indiquant nos règles de coloration. Le QTextEdit appelle automatiquement la méthode *highlightBlock* du QSyntaxHighLighter dans laquelle on aura indiqué nos règles.

```
void TextHighLighter::highlightBlock(const QString &text)
{
  setCurrentBlockState(previousBlockState());
  //l'ordre est important
  //il ne faut pas appliquer de coloration a l'interieur
  //il faut donc colorer les commentaires en premier
  for (int i = 0; i < text.length(); ++i)
  {
    //colore les commentaires
    if(cComment(last, text, i));
    //colore le texte entre quotes
    else if(cQuote(last, text, i));
    //colore les attributs
    else if(cInMarkupAttr(last, text, i));
    //colore les balises
    else if(cMarkup(last, text, i));
  }
}
```

Voici par exemple la méthode qui permet de colorer les commentaires :

```
bool TextHighLighter::cComment(int &last, const QString &text, int i)
{
  //si on se trouve deja dans un commentaire
  if(currentBlockState() == COMMENT_STATE)
  {
    //si on trouve la fin du commentaire
    if (text.mid(i, 3) == "-->")
    {
      //on colore entre last et la fin du commentaire
      setTextColor(last, i + 4, Qt::gray);
      setCurrentBlockState(DEFAULT_STATE);
    }
    setTextColor(last, i + 1, Qt::gray);
    return true;
  }
  //si on trouve le debut d'un commentaire
  else if (text.mid(i, 4) == "<!--")
  {
    //on sauvegarde la position du debut de commentaire
```

```
      last = i;
      //on marque que l'on est dans un commentaire
      setCurrentBlockState(COMMENT_STATE);
      return true;
    }
    return false;
  }
```

La classe XmlEditor

C'est donc une sous classe de TextEditor. Nous avons fait le choix de représenter les données de l'éditeur de texte simplement par une QString, pas de référence vers la position d'un noeud ou d'information supplémentaire comme sa taille ou la délimitation de son contenu. Un noeud étant identifié par son chemin depuis la racine, il faut reconstruire l'arborescence XML à partir de la QString. Cela se fait à travers la classe QXmlStreamReader qui s'utilise de la manière suivante :

```
  //creation d'un QXmlStreamReader affecte du texte du QTextEdit
  QXmlStreamReader xml(text->toPlainText());
  while(!xml.atEnd())
  {
    //detection d'une balise ouvrante
    if(xml.isStartElement())
    {
      //traitement
    }
    //detection d'une balise fermante
    else if(xml.isEndElement())
    {
      //traitement
    }
  }
```

On utilise une structure de pile lors de parcours de l'arborescence. On empile l'indice du sommet lorsque l'on rencontre une balise ouvrante et on dépile lorsque l'on rencontre une balise fermante. On parvient ainsi à se déplacer et à se repérer dans la QString.

Voici le pseudo code de l'algorithme permettant de parcourir l'arbre pour se rendre sur le noeud désiré.

```
//noeud que l'on doit trouver dans la QString
var noeudCible
var pile

//Parseur xml de Qt
var xml

Tant que l'on a pas parcouru tout l'arbre
    Si on rencontre une balise ouvrante
        Si la dernière balise rencontrée est une balise ouvrante
            Empiler 0 dans pile
        Sinon
            //c'est que l'on a atteint le fils suivant
            Incrementer le sommet de la pile de 1
        Fin Si
    Sinon Si on rencontre une balise fermante
```

```
        Dépiler pile
    Fin Si

    Si noeudCible = pile
        Appeler la fonction qui traitera le noeud
        //on arrête le parcours de l'arbre
        retourner
    Fin Si

    Sauvegarder la derière balise renconcrée
    Aller à la balise suivante
Fin Tant Que
```

Cette méthode de parcours de l'arbre est appelée lorsqu'un noeud est inséré, déplacé ou supprimé dans l'arborescence. Elle est utilisée de la manière suivante :

```
//methode appelee lorsque l'utilisateur renomme un noeud dans l'
    arborescence
//on passe en parametre un pointeur de fonction
xmlEditor->parseDom(n, n.nodeName(), QString(newName), &XmlEditor::
    updateNodeName);
```

Voici l'implémentation de l'appel de fonction dans le pseudo code du parcours de document :

```

//si le chemin du node courant est le meme que celui
//du sommet passe en parametre
if(cmpVectors(path, nodePath))
{
  //on appelle la fonction passee en parametre
  //elle traitera le node avec des effets de bord
  if((this->*function)(nbFound, begin, end, c, oldName, newName, xml))
  {
    //on arrete le parcours de l'arbre si la fonction retourne true
    return;
  }
}
```

Voici le prototype de la fonction *parseDom* avec la fonction qu'elle prend en paramètre et les fonctions de manipulation de noeud qu'elle peut prendre en paramètre.

```
//parse le dom jusqu'a trouver le node target
void parseDom(QDomNode &target, QString oldName, QString newName, bool
     (XmlEditor::*function)
(int &nbFound, int &begin, int &end, QTextCursor &c, QString oldname,
    QString newname, QXmlStreamReader &xml));

//remplace le nom de la balise oldName par newName
bool updateNodeName(int &nbFound, int &begin, int &end, QTextCursor &c
    , QString oldName, QString newName, QXmlStreamReader &xml);
//supprime le noeud
bool deleteNode(int &nbFound, int &begin, int &end, QTextCursor &c,
    QString oldName, QString newName, QXmlStreamReader &xml);
//sauvegarde les donnees du noeud dans le cas d'un deplacement
bool saveNodeData(int &nbFound, int &begin, int &end, QTextCursor &c,
    QString oldName, QString newName, QXmlStreamReader &xml);
//insere un noeud
bool insertNodeText(int &nbFound, int &begin, int &end, QTextCursor &c
    , QString oldName, QString newName, QXmlStreamReader &xml);
```

Ces fonctions manipulent le texte à travers un QTextCursor qui permet de naviguer dans un QTextEdit. Un QTextCursor permet notamment de se déplacer de mot en mot, d'aller à la fin de la ligne ou d'aller à la ligne suivante. Un outil donc indispensable dans la manipulation du texte.

La liaison vers un schéma se fait à l'aide d'expressions régulières. Le XML étant standardisé, les données auront toujours la même forme, une expression régulière semble donc le plus simple pour trouver une information dans du texte.

```
void XmlEditor::removeSchema()
{
  QString s(text->toPlainText());
  //regex du lien vers un schema
  QRegExp r("\n*xmlns:xsi.*=\".*\.xsd\"\n*");
  suppression de la regex dans le texte
  s.remove(r);
  text->setText(s);
}
```

Il est intéressant de noter que l'on utilise une fois de plus la classe QXmlStreamReader pour se déplacer dans l'arborescence XML. Le lien vers le schéma se trouve dans la balise racine, le plus simple pour la trouver est donc de parcourir le document jusqu'à la trouver.

```
while(!xml.atEnd())
{
  //lorsque l'on trouve la premiere balise ouvrante
  if(xml.isStartElement())
  {
    //on deplace le curseur jusqu'a la position de la fin de cette
        balise
    moveCursorToLineAndColumn(c, xml.lineNumber() - 1, xml.
        columnNumber() - 1, false);
    //construction du lien vers le schema
    QString link("\nxmlns:xsi=\"http://www.w3.org/2001/XMLSchema-
        instance\"\nxsi:noNamespaceSchemaLocation=\"");
    link.append(XmlFileManager::getFileManager()->getSchemaName()).
        append("\"");
    //insertion du lien dans le document
    c.insertText(link);
    text->setTextCursor(c);
    return;
  }
  xml.readNext();
}
```

En plus de créer et supprimer un lien vers un schéma, il est également possible de l'extraire, pour la validation par exemple. Cela se fait une fois de plus grâce à une expression régulière. Il est intéressant de noter dans l'exemple suivant que nous ne gérons que les liens locaux vers un schéma et pas les lien vers un schéma distant.

```
QString s(text->toPlainText());
//split du document pour se placer directement a l'endroit du lien
QStringList l(s.split("xsi:noNamespaceSchemaLocation=\""));
//si un lien est present dans le document
if(l.size() > 1)
{
  //split selon le caractere " pour ne garder que le lien
  s = l.at(1).split("\"").at(0);
  //on retourne le chemin courant du fichier XML auquel on concatene
      le lien que l'on vient de trouver
```

```
    return XmlFileManager::getFileManager()->getCurrentFilePath().append
        ("/").append(s);
  }
  //on cherche s'il n'y a pas de lien distant
  l = s.split("xsi:SchemaLocation=\"");
  if(l.size() > 1)
  {
    //on notifie qu'on ne le gere pas encore s'il existe
    emit log("Cannot process http requests yet, XML will not be checked"
        , QColor("orange"));
  }
  return "";
```

La classe SchemaEditor

La classe SchemaEditor ne rajoute pas vraiment de fonctionnalités à la classe TextEditor mise à part la génération de schéma à partir d'un document XML. Elle fonctionne de manière très simple, les balises sont extraites du document XML et sont ajoutées au schéma. L'extraction des balises se fait à nouveau grâce à la classe QXmlStreamReader, on parcourt le document et ajoute chaque balise dans une liste si elle n'est pas déjà présente

```
  while(!xml.atEnd())
  {
    //on utilise une hashmap pour simplifier la non duplication
    h.insert(xml.name().toString(), 0);
    xml.readNext();
  }
  //on appelle la methode de generation de schema
  schemaEditor->genSchema(h.keys());
```

Voici l'implémentation de la méthode de génération de schéma, on peut remarquer qu'elle est assez basique et laisse à l'utilisateur le soin de compléter la valeur et le type de chaque élément.

```
  void DtdEditor::genSchema(QList<QString> l)
  {
    //ajout des meta data
    text->setText("<?xml version=\"1.0\" encoding=\"UTF-8\" ?>");
    text->append("<xs:schema xmlns:xs=\"http://www.w3.org/2001/XMLSchema
        \">");

    //ajout de chaque element
    for(QString s : l)
    {
      QString toAppend("<xs:element name=\"");
      toAppend.append(s).append("\"/>");
      text->append(toAppend);
    }
    text->append("</xs:schema>");
    indent();
  }
```

La validation

Parlons maintenant de la validation syntaxique et sémantique. La validation syntaxique se fait très simplement à l'aide du QXmlStreamReader, qui permet de détecter une erreur et d'indiquer la ligne, ainsi que le type de l'erreur.

```
while(!xml.atEnd())
{
  xml.readNext();
  //si le parseur XML rencontre une erreur
  if(xml.hasError())
  {
    //affichage de la ligne de l'erreur avec son type
    emit log("Erreur ligne " + QString::number(xml.lineNumber()) + " :
         " + xml.errorString(), QColor("red"));
  }
}
```

Pour la validation sémantique, qui ne peut se faire que si un schéma est rattaché au document XML, on doit passer par une autre classe de Qt, le QXmlSchemaValidator et sa méthode validate. Nous l'utilisons le la manière suivante :

```
QXmlSchema schema;
//extraction de l'url du schema dans le document XML
QString url(extractSchemaUrl());

//si le xml contient une url valide vers un schema
if(url.size() > 1)
{
  schema.load(QString("file://").append(url));
  //on verifie au passage la validite du schema en lui meme
  if (schema.isValid())
  {
    emit log("Schema XSD valide", QColor("green"));
    QXmlSchemaValidator validator(schema);
    validator.setMessageHandler(mh);
    if (validator.validate(this->getText().toUtf8(), QUrl(
        XmlFileManager::getFileManager()->getCurrentSchema())))
    {
      //notification que la semantique est valide
      emit log("Semantique XML valide", QColor("green"));
    }
  }
  return true;
}
//si le schema n'a pas pu etre trouve
emit log("Schema XSD invalide, est-il manquant ou invalide ?", QColor(
    "orange"));
```

4.2.4 Le logger

Le logger est la partie que reçoit tous les messages pour les afficher à l'utilisateur. Il repose fortement sur le principe de signaux et de slots de Qt. Le fonctionnement est les suivant : Qt propose de connecter un objet émetteur, qui enverra des messages avec la macro *emit* sur le slot d'un objet receveur, qui est en réalité une fonction qui sera appelée au moment du *emit*. Voici par exemple le fonctionnement de l'envoi de messages à afficher :

Tout d'abord l'interface de notre Logger, pour montrer que les slots ne sont rien d'autres que des fonctions.

```
class Logger : public QWidget
{
  Q_OBJECT
  public:
  explicit Logger();
```

```
    signals:
    //les signaux, il n'y en a aucun dans notre Logger

    public slots:
    //notre slot log qui affichera dans la zone de texte la QString de
        la couleur QColor
    void log(QString s, QColor c);

    private slots:
    //slots prives, on ne pourra connecter que des signaux de la classe
        sur des slots de cette meme classe

    private:
    //attributs prives
};
```

On connecte les objets entre eux grâce à la fonction *connect(sender, signal, receiver, slot)* de Qt.

```
//on connecte l'editeur de texte avec le logger sur le slot log
connect(notepad, SIGNAL(log(QString,QColor)), logger, SLOT(log(QString
    ,QColor)));
```

Et lorsque qu'il y aura un *emit* du signal *log*, la fonction correspondante, dans notre cas *log* sera appelée.

```
//l'emission du signal dans la methode de validation semantique
emit log("Semantique XML valide", QColor("green"));
//appelera automatiquement
void Logger::log(QString s, QColor c)
{
  //changement de la couleur d'affichage en la couleur desiree
  logArea->setTextColor(c);
  QTextCursor cursor = logArea->textCursor();
  //insertion du texte dans le QTextEdit qui sert de zone d'affichage
      au logger
  cursor.insertText(s);

  logArea->setTextCursor(cursor);
  //affichage du Logger s'il etait masque
  show();
}
```

Ce système de signal/slot est énormément utilisé pour faire communiquer nos widgets les uns avec les autres, et permet d'une part de les séparer, c'est à dire que l'arborescence ne possède pas de pointeur vers l'éditeur de texte et inversement, mais également de ne pas avoir non plus de pointeur vers le contrôleur, une fois que les connexions sont faites dans le contrôleur, nos widgets peuvent communiquer de manière totalement indépendante.

4.3 Résultat

Chapitre 5

Manuel d'utilisation

Chapitre 6

Perspectives et conclusions

6.1 Perspectives

6.2 Conclusions

Table des figures