WTFXSL

Wonderful Transformer For XSL

Erik Clairiot & Julien Merlin

January 2008

WTFXSL, a Wonderful Transformer For XSL

Erik Clairiot, Julien Merlin

13 janvier 2008

Table des matières

1 Introduction

1.1 Presentation du logiciel

Le langage XSLT[1] est un langage, qui respecte la specification XML[2] et les recommandations W3C[3], permettant principalement la transformation de documents XML en d'autres documents XML. XSLT utilise le langage de feuilles de style XML, qui est XSL[4]. XSL est donc un méta-langage, qui permet de définir des règles de transformation d'un fichier XML en un autre.

A quoi peut donc servir un transformateur XSLT ?

Il est désormais courant qu'une source de contenu soit utilisée par de multiples utilisateurs, destinée à de divers types d'affichages (WAP, fichiers HTML, fichiers de descriptions de ressources RDF...). Il est important dans ces cas là de ne pas dupliquer autant de fois le contenu qu'il y a d'affichages diffèrents. XSLT permet d'éviter cela, en permettant pour un unique contenu (un fichier source XML), d'être affiché de multiples manières, en utilisant pour chaque type d'affichage une feuille de style XSL adéquate.
Ainsi, on peut imaginer qu'un webmaster utilise un fichier XML de contenu mis à disposition sur le net par un site tiers, peaufine plusieurs feuilles de style XSL diffèrentes selon l'affichage qu'il désire pour ce contenu, et propose au visiteur de visualiser ce contenu selon ces diffèrentes feuilles de style.

Grâce à cela, la séparation du contenu et de l'affichage est fortement appliquée, permettant une meilleure mise à jour du contenu et une meilleure répercussion sur les visualisateurs de contenu.

Concrétement, comment la génération des visualisateurs de contenu est-elle exercée ?
Tout simplement en utilisant un processeur XSLT, qui, justement, est ce que nous vous proposons.

Un processeur XSLT est le lien entre le contenu et la feuille de style.
Notre processeur permet, en acceptant en entrée un fichier XML source et une feuille de style XSL, de génerer un contenu XML résultat de la transformation par la feuille de style. Ce document résultant peut ensuite être utilisé pour visualiser le contenu.

1.2 Auteurs

Ce logiciel est une mise à jour d'un squelette mis à disposition par Mr.Levaire, maître de conférences en informatique à l'université de Lille 1.

La poursuite du développement a été executé par Erik Clairiot et Julien Merlin, étudiants en Master 1 informatique de cette même université.
Si vous souhaitez nous joindre, pour quelques raisons, nous vous encourageons à le faire en utilisant les adresses présentées çi-dessous.
Pour optimiser vos chances de réponses, n'hésitez pas à envoyer vos requêtes sur les deux adresses.

- erik.clairiot _AT_ etudiant _DOT_ univ-lille1 _DOT_ fr
- julien.merlin _AT_ etudiant _DOT_ univ-lille1 _DOT_ fr

1.3 Version

Actuellement, le logiciel est à la version 0.2.2, qui date de Janvier 2008. Elle est la suite logique de 4 premières distributions, que vous pourrez trouver à l'adresse *http ://apps.chesstux.com/xslt*.

Ce manuel a été écrit sur la base de cette dernière version.

1.4 Que me faut-il pour utiliser ce processeur ?

Le développement de cette application a été faite sur un JDK 1.6, donc vous devez posséder la JRE 1.6 de Java pour faire tourner le processeur XSLT.

[1] eXtensible Stylesheet Language Transformation
[2] eXtensible Markup Language
[3] World Wide Web Consortium
[4] eXtensible Stylesheet Language, vous l'aurez deviné...

2 Manuel d'installation

2.1 Structure de l'archive

La distribution de ce processeur XSLT se fait via une archive disponible à l'adresse *http ://apps.chesstux.com/xslt/wtfxsl-latest.tar.gz*.
Détaillons ensemble le contenu de cette archive, que nous pourrons décompresser avec la commande *tar -xzvf wtfxsl-latest.tar.gz*.

Arborescence de l'archive :

- **dist/** Contient un executable Java, sous la forme d'un fichier Jar, qui permet d'utiliser notre processeur XSLT.
- **src/** Les sources des classes Java constituant le processeur.
- **doc/** L'ensemble de la documentation, au format HTML, generée par l'outil Javadoc.
- **samples/** Ce dossier contient plusieurs fichiers XSL et XML, qui vous permettront de vous assurer du bon fonctionnement du processeur XSLT fourni. Ils sont séparés en 2 dossiers :
  - **sheets/** Les feuilles de style XSL présentant le panel complet des fonctions supportées par ce processeur.
  - **sources/** Des fichiers XML sur lesquels vous pourrez tester les feuilles de style proposées.
- **build.xml** Un fichier Ant, qui permet de notamment de compiler le processeur XSLT avec votre machine virtuelle Java.

2.2 Comment installer le processeur XSLT

Pour pouvoir utiliser notre processeur, il suffit d'avoir à disposition l'archive Jar compilée. Il ne nécessite l'utilisation d'aucune librairie externe.

Le seul impératif est de compiler l'application, ce que nous allons voir tout de suite.

La compilation de l'application est extrêment facilitée par le fichier Ant proposé. Plusieurs règles permettent de contrôler la sortie de la compilation du programme ; vous pouvez notamment choisir si vous voulez ou non rassembler le résultat de la compilation dans une archive Jar.
Note : Il est possible de ne pas utiliser Ant pour compiler, les seules classes à compiler se trouve dans le dossier *src/*.

Ci-dessous, les diffèrentes règles proposées par notre fichier Ant qui agissent sur la compilation du processeur.

- **compile** Compilation des sources
- **dist** Compilation et création d'une archive Jar

Pour utiliser une de ces règles avec Ant, le plus simple est de lancer en ligne de commande la requête suivante : *ant regle*

Nous ne saurions vous recommander d'utiliser la règle *dist*, qui permet tout de même une meilleure propagation pour les fichiers compilés de l'application.

2.3 Comment lancer le processeur XSLT

Pour lancer le processeur, il faut l'avoir compilé au préalable, ce qui a été expliqué dans la section précédente. Selon les deux règles de compilation, nous vous présenterons les commandes à executer pour lancer le processeur. Notons DIR le dossier ou vous avez decompressé l'archive.

– **dist**
Dans le dossier *DIR/dist*, lancez la commande :
java -jar wtfxsl.jar options

– **compile**
Dans le dossier *DIR/bin*, lancez la commande :
java transformer.Transformer options

Attention Il est très important de lancer les commandes çi-dessus à partir du dossier spécifié, sinon, le processeur ne trouvera pas le fichier de configuration, et le lancement ne fonctionnera pas.

Maintenant que vous savez lancer le processeur, nous vous présenterons dans une prochaine partie les options que le processeur accepte.

Note Nous utiliserons par la suite l'alias *run*, qui correspondra en fait à une des deux commandes présentées çi-dessus, selon la règle que vous aurez utilisée pour compiler le processeur.

2.4 Paramètrage du logiciel

Normalement, après l'installation, vous n'avez aucune modification à faire au niveau des fichiers de configuration de l'application.
A l'execution, le transformateur est parametrable par les options que l'on peut passer par la ligne de commande.

Un unique fichier de configuration est susceptible d'être modifié, et cela uniquement si vous effectuez des modifications sur les sources de l'application.
Pour les développeurs qui voudraient s'y risquer, sachez que ce fichier est à modifier uniquement si vous modifiez des instructions, ou en ajoutez de nouvelles. Il permet principalement de lier une instruction de type XSL à une classe de type *transformer.xsl.instruction.Instruction*.
Ce fichier, présent dans le dossier *conf/* à la racine de l'archive, ou du dossier de l'archive décompressée, se nomme *instructionResolver.conf*. Chaque ligne représente le nom de l'instruction XSL, précédée de l'espace de nommage *xsl :*, suivi après un espace du nom de la classe à charger lorsqu'une telle instruction XSL est trouvée dans une feuille de style XSL.
Voici un exemple de ligne du fichier, qui permet de charger l'instruction *ElementInstruction* lorsque l'instruction *xsl :element* est rencontrée dans une feuille de style XSL :
xsl :element transformer.xsl.instruction.ElementInstruction

2.5 Tester l'installation et le fonctionnement

Pour tester le fonctionnement du logiciel, rien de mieux qu'une batterie de tests à effectuer par notre processeur. Ainsi, nous avons créé de multiples feuilles de style, aux rendus relativement diffèrents, que l'on testera sur un unique fichier source XML. Cela permettra notamment de voir le bon fonctionnement du processeur, et en même temps, de voir comment fonctionne certains règles et instructions XSL que l'on utilise dans nos feuilles de style.

Ces fichiers sont contenus dans le dossier *samples/*, avec les feuilles de style dans *samples/sheets* et les fichiers sources XML dans *samples/sources*.

Nous avons implementé, pour pouvoir tester de manière aisée le processeur, un système permettant de transformer un fichier source XML avec un ensemble de feuilles de style, contenues dans un dossier. De plus, pour pouvoir se rendre compte rapidement du rendu du résultat, il est possible de demander de créer un fichier pour chaque transformation.

Ici, nous voulons transformer des fichiers XML en de multiples fichiers HTML. Donc, une manière simple de visualiser le rendu est d'ouvrir chacun des fichiers résultats dans un navigateur Internet classique.

Comment tester

1. Déplacez vous dans le dossier *dist/* à la racine du dossier decompressé.
2. Créez un dossier ou stocker le résultat des transformations, par exemple */tmp/xslt*.
3. Executez l'instruction suivante :
   java -jar wtfxsl.jar -in ../samples/sheets/test1.xml -xsl_dir ../samples/sheets -out_dir /tmp/xslt -ext html
4. Consultez le dossier */tmp/xslt/* dans votre navigateur, et analysez chacun des fichiers produits pour vérifier le succès des transformations.

3 Manuel d'utilisation

Une fois le logiciel installé, il est temps d'apprendre à se servir du processeur XSLT. Cette partie détaillera en premier lieu le fonctionnement global du logiciel, puis nous détaillerons les fonctionnalités présentes dans le processeur. Il sera normalement aisé pour vous par la suite de vous servir du logiciel.

3.1 Fonctionnalités

Que permet de faire notre processeur XSLT ?

Nous pouvons distinguer 2 types de fonctionnalités : celles que notre application permet, et celles que le transformateur XSL à proprement dit permet, et parmi ces dernières, nous présenterons particulièrement les instructions XSL supportées par notre transformateur.

Application

- Conversion d'un fichier source XML en un fichier résultat XML, selon une feuille de style spécifiée
- Conversion d'un fichier source XML avec un ensemble de feuilles XML contenues dans un dossier
- Exportation du résultat sous un fichier XML d'extension spécifiable par l'utilisateur

Instructions XSL

Chaque résultat d'évaluation d'instruction remplace, dans le document résultat, l'instruction XSL qui aura été copié préalablement.

- **xsl :value-of**
  Permet de récupérer le texte résultat d'une évaluation XPath, close spécifiée par l'attribut *select* de l'instruction. L'évaluation XPath se fait à partir du même noeud contexte que celui de la règle contenant cette instruction.

- **xsl :apply-templates**
  Recherche les règles appliquables sur le noeud contexte passé à l'attribut *select* de l'instruction. Si cet attribut n'est pas renseigné, le noeud contexte est le noeud courant. Une fois ces règles trouvées, si règles il y a, elles se voient évaluées.

- **xsl :call-template**
  Recherche parmi les templates nommés celui qui correspond à la valeur de l'attribut *select* de cette instruction. Cette règle est ensuite evaluée, à partir du noeud contexte de la règle mère de l'instruction.

- **xsl :if**
  Permet d'effectuer des tests au sein d'une feuille de style, et de sélectionner une execution plutôt qu'une autre selon le résultat d'une évaluation. L'attribut *test* contient la clause à tester, il s'agit d'une expression XPath qui est censée retourner un booléen. Si la clause s'avère vérifiée, alors le corps de l'instruction *xsl :if* se voit executé. Sinon, rien n'est executé, et le résultat de cette instruction est donc une liste de noeuds vide.

- **xsl :choose**
  Cette instruction permet d'utiliser de manière plus puissante l'instruction *xsl :if*, pour la simple raison qu'elle offre la possibilité de tester des valeurs alternatives, à l'instar des structures conditionnelles *if/else*, ou plus précisément, du *switch*.
  Deux instructions découlent du *xsl :choose*.
  xsl :when accueille en attribut *test* une évaluation XPath, et *xsl :otherwise* n'accepte aucun attribut. L'instruction *xsl :choose* testera successivement les instructions *xsl :when*, jusqu'à obtenir une valeur vraie, et éxecutera le corps de règle correspondant à l'instruction ayant retournée vrai. Si aucune de ces instructions n'a renvoyé de valeur vraie, alors, si l'instruction *xsl :otherwise* existe, le corps de règle de celle-çi est executée.

- **xsl :element**
  Si vous voulez créer un élement XML, cette instruction est là pour vous. Elle accepte en attribut *name* le nom de l'élement à créer, et en namespace le nom de l'espace de noms auquel associer l'élement créé. Le corps de règle évalué de cette instruction est logiquement ce que contiendra l'élement créé.

- **xsl :text**
  Cette instruction est de loin la plus basique, car le résultat de son évaluation est juste un noeud de type texte. Le contenu de ce noeud est juste le contenu de l'instruction xsl :text.

- **xsl :for-each**
  Cette instruction permet d'effectuer des traitements en boucle avec l'attribut *select*, qui est une évaluation XPath retournant un ensemble de noeuds. Pour chacun de ses noeuds, le corps de l'instruction *xsl :for-each* sera evalué une fois, en prenant pour noeud contexte l'ensemble des noeuds retournés.

- **xsl :key**
  Cette instruction permet de regrouper des noeuds du document source avec l'attribut *match*, qui contient une évaluation XPath devant retourner un ensemble de noeuds, d'utiliser une expression qui permet pour chacun de ses noeuds trouvé, de générer une clé, avec l'attribut *use*, et de mémoriser cet ensemble de noeud sous un nom de clé, avec l'attribut *name*. Cela permet alors à la feuille de style, à tout instant, de demander un noeud correspond à une clé spécifique, avec la fonction *key(name, value)* associée à cette instruction.

- **xsl :sort**
  Cette instruction permet de trier les noeuds résultat d'un appel à l'instruction *xsl :for-each*. La clé du tri est l'element en attribut *select*, et l'évaluation se fait à partir du noeud contexte de l'instruction *for-each*. Il est possible de demander un ordre au tri, de manière ascendante (par défaut) ou descendante, avec l'attribut *order*. De plus, selon la clé du tri, il peut être nécessaire de demander le type de tri, sur chaine de caractères ou numérique, avec l'attribut *data-type*.

3.2 Fonctionnalités futures

Trois instructions seront implémentées prochainement, afin de rendre plus complet le processeur. Pour l'instant, il n'y a aucun moyen pour utiliser des variables ou paramètres XSL dans vos feuilles de style. Pour pallier à cela, nous devrons établir les instructions *xsl :param*, *xsl :with-param* et *xsl :variable*.

De plus, il pourrait être interessant d'implémenter l'instruction *xsl :attribute*, afin de rendre l'utilisation notamment de *xsl :element* beaucoup plus puissante.

3.3 Comment utiliser le processeur

-v **Verbosité**
  Attend un entier compris entre 0 et 5, et permet de contrôler de degré de verbosité de l'application, c'est à dire la quantité d'affichages dans les logs que l'application generera, selon les activités du processeur. Par défaut, est à 0.

-in **Fichier XML source**
  Attend le nom d'un fichier XML qui sera le fichier sur lequel on appliquera la feuille de style XSL.

-out **Exporter le résultat**
  Nom d'un fichier XML résultat qui sera créé, et qui contiendra le résultat de la transformation.

-xsl **Feuille de style à appliquer**
  Nom de la feuille de style XSL à appliquer sur le fichier source.

-ext **Extension du fichier résultat**
  Extension à ajouter au fichier exporté. Si cette option n'est pas spécifié, il n'y a aucun ajout d'extension en fin de nom de fichier.

-xsl_dir **Dossiers de feuille de style**
  Lorsque vous voulez appliquer sur un même fichier source XML plusieurs feuilles de style, la meilleure méthode est de réunir ces feuilles dans un dossier, et d'utiliser cette option. Elle permet d'appliquer toutes les feuilles contenues dans le dossier.

-out_dir **Dossiers des fichiers résultats**
  Lorsque vous utilisez de multiples feuilles de style, avec l'option précedente, il est préferable d'exporter le résultat dans des fichiers distincts. Cette option permet de spécifier le dossier qui contiendra ces feuilles résultats. Les fichiers résultats se nommeront à de la même manière que les feuilles de style qui les ont créées, à la diffèrence de l'extension qui sera supprimée. L'option -ext permet de personnaliser les extensions de ces fichiers.

Attention, ces options sont parfois exclusives, et le fait d'en employer certaines ignorent certaines options.
Par exemple, lorsque vous souhaitez utiliser plusieurs feuilles de style à transformer sur un fichier source XML, il est préférable de mettre toutes ces feuilles de style dans un dossier, et de la passer en valeur à l'option *xsl_dir*. Par conséquent, si l'option *xsl* est passée en même temps, celle-çi sera normalement ignorée.
Les options *xsl_dir* et *out_dir* vont de paires, tout comme les options *xsl* et *out*.

3.4 Tutoriel

Nous supposerons que vous avez correctement installé le processeur XSLT, que son fonctionnement a été verifié. Si vous n'avez pas passé cette étape, nous vous recommandons de retourner à ces étapes précédentes, puisque le tutoriel ne vous servirait à rien.
Pour plus de commodités, toutes les commandes se feront relativement au dossier courant. Libre à vous d'adapter ensuite les commandes selon vos necessités.
Note : La commande *run* correspond à la commande à éxécuter pour lancer le processeur, selon que vous ayez décidé de compiler l'application dans un fichier Jar externe ou non.

Etape 1 : Création d'un document source XML
Une transformation XSL se fait sur un document XML source. Nous supposerons originalement que ce fichier s'intitule *source.xml*.

Etape 2 : Création de feuilles de style XSL
Une transformation XSL se fait en suivant les règles précisées dans un fichier XSL. Supposons un dossier nommé *styles* dans le dossier courant. Nous imaginerons avoir dans ce dossier plusieurs feuilles de style nommées *style.xsl*, *style2.xsl* ...

Etape 3 : Application de la transformation par le processeur XSLT
C'est ici que le tutoriel prend de l'ampleur. Nous présenterons succintement les options que l'on peut demander au processeur.

- Transformation
  Pour effectuer une transformation, il faut utiliser les paramètres *-in* et *-xsl* pour respectivement spécifier le fichier source et la feuille de style.

  run -in source.xml -xsl style.xsl

- Transformation et exportation
  Pour exporter le résultat de la transformation, par exemple dans un fichier nommé *sortie.xml*, il faut ajouter le paramètre *-out*.

  run -in source.xml -xsl style.xsl -out sortie.xml

- Transformation et exportation dans un format spécifique
  Pour préciser explicitement le format du fichier dans lequel la transformation sera exportée, il est possible d'utiliser le paramètre *-ext*. Cependant, nous conviendrons que cette option n'a aucun interêt dans le cas d'une spécification explicite du fichier de sortie (et d'ailleurs ignorée), cette option prenant toute son ampleur dans le cas de transformation multiples, que nous verrons çi-après ; ici, une exportation à l'extension *html*.

  run -in source.xml -xsl style.xsl -out sortie.xml -ext html

- Transformations multiples sur un fichier source unique
  Pour effectuer des transformations multiples en un appel au processeur, il est nécessaire de placer toutes les feuilles de style dans un dossier, pour pouvoir le spécifier en paramètre avec *-xsl_dir*. Une transformation sera effectuée par feuille de style, et cela successivement.
  Attention, cette option ne permet pas de traiter de multiples fichiers source, mais uniquement de multiples feuilles de style.

  run -in source.xml -xsl_dir styles

- Transformations multiples et exportations
  Il peut être utile d'exporter chaque transformation dans des fichiers differents, pour cela, l'option *-out_dir* vous satisfera. Le résultat de chaque feuille de style sera donc exporté dans un fichier dans le dossier passé en paramètre. Le nom du fichier résultat sera le même que celui de la feuille de style appliquée, seule

l'extension sera diffèrente, par défaut *xml*.

run -in source.xml -xsl_dir styles -out_dir exports

– Transformations multiples et exportations dans un format spécifié
Pour spécifier une extension aux fichiers créés, l'option *-ext* est utile. Ici, par exemple, l'extension sera *html*.

run -in source.xml -xsl_dir styles -out_dir exports -ext html

3.5 Bugs

A notre grand désarroi, nous n'avons pas (encore) trouvé de bugs. Nous attendons donc impatiemment tout retour d'utilisation afin de remplir cette section.

4 Maintenance

4.1 Architecture de l'application

Nous allons détailler dans cette partie l'architecture de notre application, afin de pouvoir faciliter une reprise éventuelle du code par des développeurs externes à notre équipe.

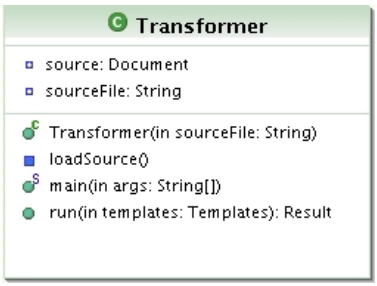
Notre application est codée en Java, et est divisée en plusieurs paquetages. Le paquetage commun pour l'application est le paquetage *transformer*.

4.1.1 Paquetage transformer

Ce paquetage contient (en plus de ses sous-paquetages) une unique classe, *Transformer*, qui est la classe qui contient le code à exécuter au lancement de l'application. Elle gère principalement la gestion des options, du parametrage de la transformation et de son lancement.

Notons que le transformateur n'est aucunement lié à une feuille de style XSL, mais uniquement au document source à transformer. Ainsi, à partir d'un transformateur, il suffit d'instancier, à partir d'une feuille de XSL, un ensemble de règles de type *transformer.xsl.Templates*, et de les passer en paramètre à la méthode *Transformer.run(Templates)*, pour obtenir en retour un document résultat de la transformation.

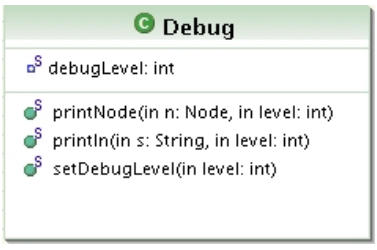
Exemple Diagramme UML du paquetage



4.1.2 Paquetage transformer.misc

Contient les classes utilitaires à l'application, principalement ici la classe *transformer.misc.Debug*, qui sert à contrôler la verbosité de l'application selon la demande de l'utilisateur. L'echelle de verbosité peut être reglée, avec la méthode *Debug.setDebugLevel(int)*, sur une échelle de 0 à 5, allant d'une verbosité très faible, voire nulle, à une verbosité très, voire trop, détaillée.
Selon le niveau de verbosité, les demandes d'affichages sur la sortie standard par la méthode *Debug.println(String, int)* se verront réalisées ou non.

Exemple Diagramme UML du paquetage

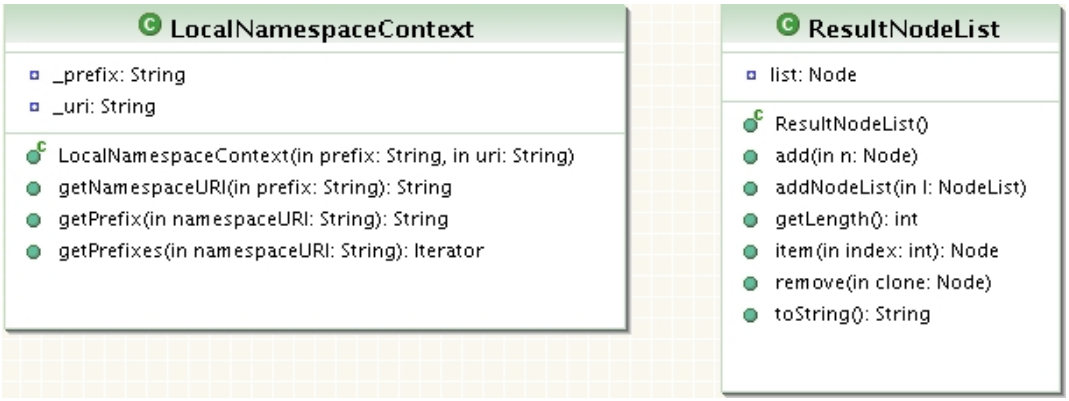


4.1.3 Paquetage transformer.xml

Contient des classes permettant une utilisation optimisée des documents XML.

Nous proposons donc une classe qui permet de facilement gérer les espaces de noms pour les noeuds XML (*transformer.xml.LocalNameSpaceContext*), et une autre classe permettant de facilement gérer une collection de noeuds XML (*transformer.xml.ResultNodeList*).
Cette dernière implémente naturellement l'interface *org.w3c.dom.NodeList* et utilise une simple *java.util.ArrayList* pour stocker les noeuds. A noter que nous avons agrémenté cette classe d'une méthode permettant d'ajouter directement une liste de noeuds à ce conteneur (méthode *ResultNodeList.addNodeList(ResultNodeList)*).

Exemple Diagramme UML du paquetage



4.1.4 Paquetage transformer.xsl

Ce paquetage contient uniquement les classes qui ne servent que pour le processeur XSLT.

Définition d'un noeud contexte
Les noeuds contexte sont essentiels en XSL : ils permettent de définir un point de démarrage pour les recherches et évaluations XPath. Tout simplement, un noeud contexte est un noeud à partir duquel les transformations s'appliqueront. Cette classe (*transformer.xsl.Context*) est donc une simple couche pour un noeud XML.

Règles XSL et leur gestion
En XSL, deux types de règles existent : les règles nommées et les règles à motifs. Toutefois, elles sont toutes deux implémentées par la même classe (*transformer.xsl.Template*).
Une règle XSL doit pouvoir être executée sur un contexte, ce que fait clairement la méthode *Template.run(Context, Result)*. Globalement, l'execution d'une règle va passer par l'évaluation de son corps de règle, c'est à dire, des noeuds fils du noeud de la règle XSL.
Ce que l'on attend d'une règle en XSL est de pouvoir être explicitement appelée, dans le cas d'une règle nommée, ou d'être trouvée selon un noeud contexte, dans le cas d'une règle à motif. Les méthodes *Template.getName()* et *Template.getMatchString()* permettent respectivement cela. Mais cela n'est pas suffisant, il est nécessaire de disposer d'un gestionnaire de règles, pour avoir une vue d'ensemble sur la feuille XSL et pour pouvoir à partir de n'importe de quel règle appeler une autre règle.

La gestion des templates se fait grâce à la classe *transformer.xsl.Templates*, qui contient donc la liste de tous les templates, nommés et à motif. Cette classe permet notamment de chercher, sur un noeud contexte, la liste des templates appliquables sur un noeud fils du contexte, et ceci grâce à la méthode *Templates.findTemplates(Node)*. Notons aussi la méthode *Templates.loadTemplates()*, qui permet de charger les templates présents dans la feuille de style XSL, dont le nom du fichier aura été passé en paramètre à l'instanciation de l'objet.

Résultat d'une transformation XSL
Un résultat de transformation XSL est globalement un ensemble de noeuds XML. Comme les noeuds transformés proviennent de la feuille de style XSL et du document XML source, nous devons instancier un document de type DOM afin de pouvoir créer des noeuds de type XML. Ceci est la raison de l'attribut Document dans la classe *transformer.xsl.Result*.
A la création d'un document résultat, un élement est créé, prêt à accueillir tous les élements résultants de transformations. Tous les noeuds que l'on transforme progressivement se voient donc ajoutés à partir de cet élement, avec la méthode *Result.addNode(Node)*, qui évolue progressivement au gré des transformations.
Comme à tout instant, dans une feuille de style XSL, il est possible de quérir une recherche de règles instanciables sur un noeud contexte, il est nécessaire d'avoir constamment accès à une liste de règle valide. Comme ce document résultat est accessible à tout instant de la transformation, et que les règles sont conjointement liées au résultat, cette classe est naturellement dotée d'un attribut *transformer.xsl.Templates*.

La méthode *Result.export(String)* permet d'exporter le résultat de la transformation dans un fichier.

Corps de règle
L'évaluation d'une règle est, dans un transformateur XSL, générique. C'est à dire que quelque soit les instructions XSL qui s'y trouvent, l'évaluation restera similaire, le plus spécifique étant justement présent dans les instructions XSL. Ainsi, la classe *transformer.xsl.RuleBody* est une classe très simple, qui a principalement une méthode *RuleBody.runRuleBody(ResultNodeList, Context, Result)* qui fait ce travail d'évalution de corps de règle. Elle retourne une liste de noeuds, qui est le résultat de cette évaluation.

Profitons en pour détailler comment fonctionne l'évaluation d'un corps de règle :
Le corps d'une règle est composé de tous les noeuds fils d'une instruction XSL. Lorsqu'on évalue une règle, on copie le corps de cette règle dans le document résultat, en les créant à partir du document contenu dans *transformer.xsl.Result*. Puis, on recherche dans ce corps toutes les instructions de type XSL de premier niveau, c'est à dire qu'une instruction XSL ayant un ancètre de l'espace de nom XSL ne satisfait pas la recherche, et on évalue successivement ces instructions trouvées. Les évaluations se font à partir du noeud contexte passé en paramètre à la règle.

Exemple Diagramme UML du paquetage

4.1.5 Paquetage transformer.xsl.instruction

Contient les implémentatins d'instructions XSL.

Instructions XSL
La classe *transformer.xsl.instruction.Instruction* est la classe que toute implémentation d'instruction XSL doit étendre. Elle est relativement basique, car la seule chose que l'on peut demander à une instruction, est de s'executer.
Une instruction est caracterisée par le noeud XSL qui l'instancie, ce noeud devant toujours être spécifié au

moment de la construction de l'instruction.
L'execution d'une instruction est effectuée en spécifiant un noeud contexte (*instruction.xml.Context*), qui définit le noeud du document XML source sur lequel l'instruction va effectuer son traitement ; l'algorithme d'execution de l'instruction étant contenu dans la méthode *instruction.run(Context, Result)*.
Toute execution d'instruction renvoie un ensemble de noeuds XML (*instruction.xml.ResultNodeList*) à la fin de son execution, noeuds qui correspondent au traitement de l'instruction sur contexte passé en paramètre. Cependant, si quelque chose se passe mal lors du traitement, il se peut que l'instruction retourne une valeur nulle à la place d'une liste de noeuds.

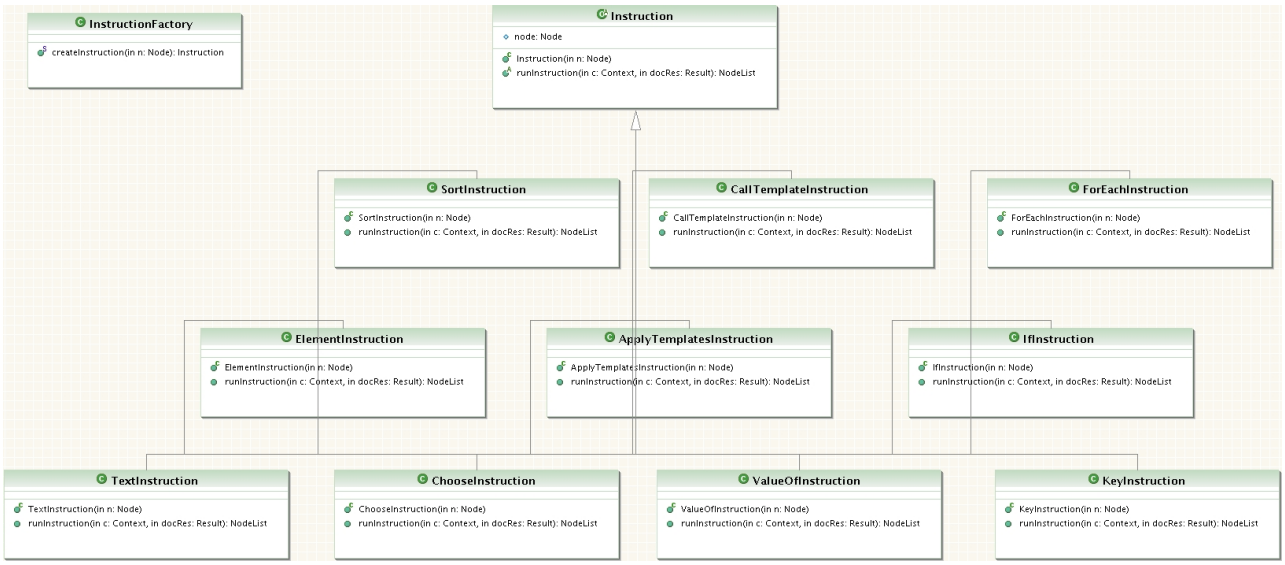
Ainsi, ce paquetage est déjà composé de plusieurs classes, toutes étendant cette superclasse présentée juste au-dessus, qui sont les implémentations des instructions supportées par le processeur XSLT.

Si vous désirez ajouter de nouvelles instructions au processeur XSLT, nous vous conseillons de vous inspirer de l'implémentation des instructions basiques, par exemple *transformer.xsl.instruction.ValueOfInstruction*.

Générateur d'instructions XSL
Une classe est présente dans ce paquetage, qui est assez importante toute de même, puisqu'elle permet notamment d'instancier à partir d'un noeud XSL un objet de la classe d'instruction correspondante. Cependant, il n'y aucune raison que vous y touchiez, puisque l'association entre une instruction XSL et sa classe Java se fait dans un fichier texte externe, et que le générateur fonctionne sur la base d'un chargeur dynamique d'objets.

Important ! A chaque nouvelle instruction implementée, vous devez modifier ce fichier texte afin de faire correspondre l'association XSL / Java. Sinon, jamais votre instruction ne sera chargée et executée. Référez-vous à la partie *2.4. Paramètrage du logiciel*, qui explique l'utilisation de ce fichier.

Exemple Diagramme UML du paquetage



4.1.6 Paquetage transformer.xsl.function

Ce paquetage recense des implémentations d'instructions XSL que l'on pourrait appeler des fonctions, comme par exemple une demande de tri, ou encore une demande de mémorisation de clé..

4.2 Recuperer la documentation

A partir de l'archive que vous avez recuperée, qui ne contient pas encore la documentation, il est très aisé de la génerer. Le plus simple est de disposer de l'outil Ant ; normalement, vous disposez déjà de l'outil Javadoc avec l'environnement de développement Java.

A la racine du dossier decompressé, là ou se trouve un fichier *build.xml*, il suffit d'executer la commande *ant doc*. Cette commande aura pour effet de créer un dossier *doc* dans le répertoire courant, et de générer la documentation dans ce dossier.
Après la génération, ouvrez dans votre navigateur favori la page HTML *doc/index.html*.

NB : La documentation est aussi accessible à l'adresse *http ://apps.chesstux.com/xslt/doc/*.

5 Historique de développement

5.1 Versions existantes du logiciel

Quatre versions du logiciel existent, marquées par des évolutions que nous pouvons considérer comme majeures. Ces versions sont disponibles à l'adresse *http ://apps.chesstux.com/xslt/*.

0.2.2 **Version finale**
Cette version est le processeur XSLT fonctionnel, ne présentant plus de bogues apparents, et permettant un contrôle plus aisé du processeur par un utilisateur lambda.

0.2.1 **Refonte du processeur**
Remise en forme du code, afin de permettre une meilleure évolution de développement et une analyse plus aisée des problèmes que nous avions rencontré.

0.1.2 **Processeur partiellement fonctionnel**
Le processeur en l'état implémente quelques instructions XSL, mais ne les execute pas toutes de manière efficace et correcte.

0.1 **Squelette de processeur**
Fichiers proposés par Mr.Levaire, qui contient du code fonctionnel et du code à implémenter, pour assurer un fonctionnement très basique du processeur. Contient notamment un vérificateur de feuilles de style XSL.

Evidemment, nous ne saurions que vous conseiller de prendre la version le plus récente à votre disposition.

5.2 Bilan du développement

WTFXSL est une application qui résulte d'un projet mené dans le cursus universitaire que nous suivons actuellement, étant en première année de Master Informatique à l'université scientifique de Lille 1, en cette année 2007.

L'objectif était de créer un processeur XSLT permettant d'effectuer des transformations à partir de feuilles de style XSL. Cependant, nous n'étions pas engagés à amener notre processeur à supporter toutes les instructions que XSL fourni, mais uniquement les plus usitées, et permettant une utilisation tout de même puissante et utile du transformateur.
Ainsi, il était demandé de supporter toutes les instructions que WTFXSL implémente actuellement, et en plus de celà, les instructions permettant d'utiliser des variables et paramètres au sein des feuilles de style XSL.
Vous l'aurez compris, WTFXSL ne gère pas les instructions *xsl :param* et *xsl :variable*, pour la triste raison qui est que nous ne maîtrisons actuellement pas de manière assez efficace l'évaluateur XPath pour pouvoir sortir un code permettant l'usage de ces instructions.

Pour pallier à ces implémentations non réalisées, nous avons ajouté des fonctions telle que l'exportation de transformation ou les transformations multiples, pour ne citer qu'elles, afin d'enrichir notre processeur.

5.3 Evolutions futures

La première évolution à faire serait évidemment l'implémentation des instructions *xsl :variable* et *xsl :param*. De plus, il ne serait pas malvenu d'implémenter par la même un plus large panel d'instructions XSL, afin de pouvoir satisfaire les feuilles de style XSL les plus incongrues.
Il serait imaginable aussi de pouvoir effectuer le traitement de transformation de manière multi-threadée.

Vous l'aurez compris, il n'y a point de limite à l'évolution que WTFXSL peut avoir.

6 Conclusion des développeurs

Le développement de cette application nous aura fortement enrichi nos connaissances sur un ensemble de technologies que nous ne connaissions peu, ou pas.

La phase d'introduction à XSL et à XSLT fut pour nous une découverte, tout autant que pour XPath. De la découverte à la maitrise de ces technologies, un temps s'est écoulé. Il a fallu que nous developpions nos connaissances sur ces sujets avant que nous puissions saisir tout le principe de fonctionnement des transformations XSL. Cependant, tout au long du développement, notre compréhension sur l'évaluateur XPath s'est accrue, la puissance que peut fournir XSL s'est révélée encore plus fortement, et l'interêt pour le développement de ce processeur s'en est donc vu grandi.

Pour conclure, nous dirions que nous ne regrettons aucunement ce développement, pour la simple raison qu'il nous aura permis de connaître deux technologies, XSL et XPath, qui pourraient nous servir ultérieurement, que ce soit pour un développement de projet ou pour une utilisation personnelle.

7 HOWTO [English]

7.1 Presentation

This application is an XSL processor, designed to apply transformations on an XML source file. A transformation is described by an XSL file, which contains instructions and templates. The result of a transformation is also an XML file.
WTFXSL supports some XSL instructions, but not all of those that XSL procures.

7.2 Installation

WTFXSL is coded in Java 1.6, so you have to have JRE & JDK 1.6 installed on your machine to use this software.

The first thing to do to use WTFXSL is to compile it, by using Ant tool. It's also possible to not use Ant to compile, but this howto file only treats compilation with Ant.
There are two Ant rules available to compile WTFXSL :

- **compile**
  Compile all sources to *bin/* directory.

- **dist**
  Compile all sources to *bin/* directory, and exports compilation into a jarfile named *wtfxsl.jar* in directory *dist/*.

To launch a rule with Ant, you just have to execute *ant rule*, where *rule* is the choosen rule.

7.3 Utilisation

The method to launch this processor varies with the rule used to compile WTFXSL.
If you compiled with the *compile* rule, you have to launch from *bin/* directory the command *java transformer.Transformer*.
Else, if you compiled with the *dist* rule, it's from the *dist/* directory that you have to execute *java -jar dist/wtfxsl.jar* command.
In futures explanations, we will use an alias named *run* which corresponds to the command to use to launch the application, regarding the way you compiled it.

To use this processor, you have to know options that WTFXSL purposes. You can have a brief presentation of those options by launching *run*, without any options specified. Another way is to read what follows.

-v **Verbosity**
  Defines the quantity of information the processor has to give during transformation. You have to specify an integer ranged between 0 and 5. Per default, it is sets to 0.
-in **XML source file**
  Name of the file on which the processor wil apply transformation.
-out **Exporting the result**
  Name of the file which will contains the result of the transformation.
-xsl **XSL stylesheet to apply**
  Name of the stylesheet which will described the transformation to apply on the source file specified in parameter.
-ext **Exported file extension**
  Extension of the exported file. Not necessary, but forces the extension. Particulary powerful when using multiples stylesheets on an unique source file.
-xsl_dir **Stylesheets directory**
  Directory that contains all stylesheets to apply on an unique XML source file.
-out_dir **Exported files directory**
  If you are using previous option, which applys multiples stylesheets on an unique source file, it's very convenient to export each transformation in distinct files, instead of displaying transformations in the terminal. This option exports each transformation in a file into specified directory. Each file is named the same way as the stylesheet which was used for the file. Don't hesitate to use this option with *-ext*, which allows you to specify an extension for created files.