Urbilog

Solutions Accessibles

Documentation technique et utilisateur

API PHP 5 pour la génération de document Open XML

Rédacteur : Loïc Torrez

Urbilog
Solutions Accessibles

SOMMAIRE

A. Description de l'api

Cette api permet de faire gagner du temps dans la création de documents Word et PowerPoint. En effet, elle contient un ensemble de fonctions permettant de générer automatiquement des documents en fusionnant un document template et des données provenant d'un ou plusieurs fichiers XML.

Afin d'acquérir une certaine flexibilité tout en gardant un grand nombre de fonctionnalités, l'application se découpe en trois parties : l'api, le template et CLIPS. Ce mode de fonctionnement peut être associé à un modèle MVC où le modèle serait le template, la vue CLIPS et le contrôleur l'api.

Dans la partie utilisateur, je m'efforcerai d'expliquer comment se servir de l'api et quant à la partie technique, elle sera consacrée à l'explication du fonctionnement de l'api en elle-même ainsi qu'à la façon d'ajouter facilement de nouvelles fonctionnalités.

1. Le template

Le but du template est de s'occuper de la partie présentation, c'est-à-dire stocker les styles du document, comme par exemple mettre un pied de page, un sommaire, un style particulier pour les titres de niveau deux, définir la taille des marges, en somme toutes les choses qui peuvent se faire automatiquement. Afin que l'api comprenne où et comment insérer des informations il est nécessaire d'utiliser des marqueurs. Afin de ne pas être limité à de simples remplacements, une grammaire a été élaborée permettant de réaliser des choses un peu plus complexes telles que des boucles.

2. CLIPS

CLIPS *(C Language Integrated Production System)* est un environnement et un langage de programmation créés en 1985, faisant partie du paradigme des langages déclaratifs et logiques.

Il s'agit avant tout d'un outil de construction de systèmes experts à base de règles et d'objets. Ses caractéristiques notables sont :

- La gestion de trois paradigmes à la fois : programmation objet, programmation procédurale et programmation par règles.
- Une grande flexibilité dans la représentation des connaissances, grâce à ces trois paradigmes.
- Portabilité et rapidité : il est écrit en C.

- Intégration : Clips peut être embarqué dans des logiciels écrits dans d'autres langages.
- Extensibilité : ces autres langages peuvent servir à étendre ses fonctionnalités en respectant certains protocoles.
- Interactivité avec l'interpréteur de commandes.
- Fonctionnalités poussées de validation de systèmes experts, de vérification de contraintes.

Dans notre cas, nous utilisons CLIPS afin d'améliorer, de personnaliser la façon dont sont affichées les choses.

3. L'api

L'api est écrite en PHP 5.1.3 pour plusieurs raisons, la première est que le PHP est un langage simple que la plupart des gens connaissent et qu'il est connu pour sa puissance et simplicité dans les traitements XML. Quand au choix de la version il était imposé, en effet afin d'utiliser CLIPS il faut se servir d'une version spécifique de PHP, compilé par Michel et intégrant nativement CLIPS. L'api est écrite en objet et se compose de plus de 45 classes réparties dans quatre principaux packages : transformer, document, utils et une série d'autres petits packages.

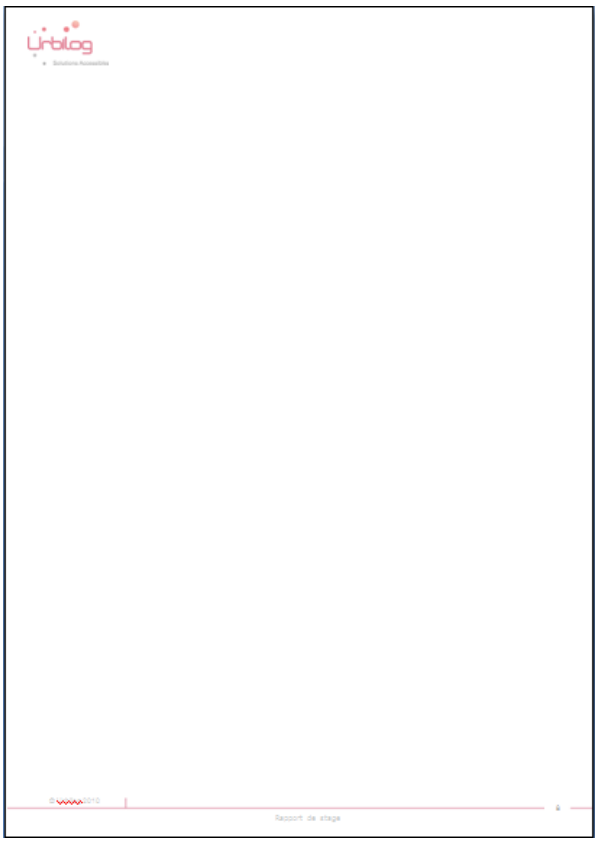
B. Documentation utilisateur

Cette partie va vous permettre de vous servir de l'api pour générer des documents Word et PowerPoint. Comme dit précédent l'api est basé sur un pseudo MCD. Le but de ce mécanisme est qu'une fois le projet fini, on touche le moins possible à l'api et qu'il soit quand même possible de changer de manière significative la façon dont sont générés les documents, soit en modifiant le template, soit les règles.

1. Création d'un template

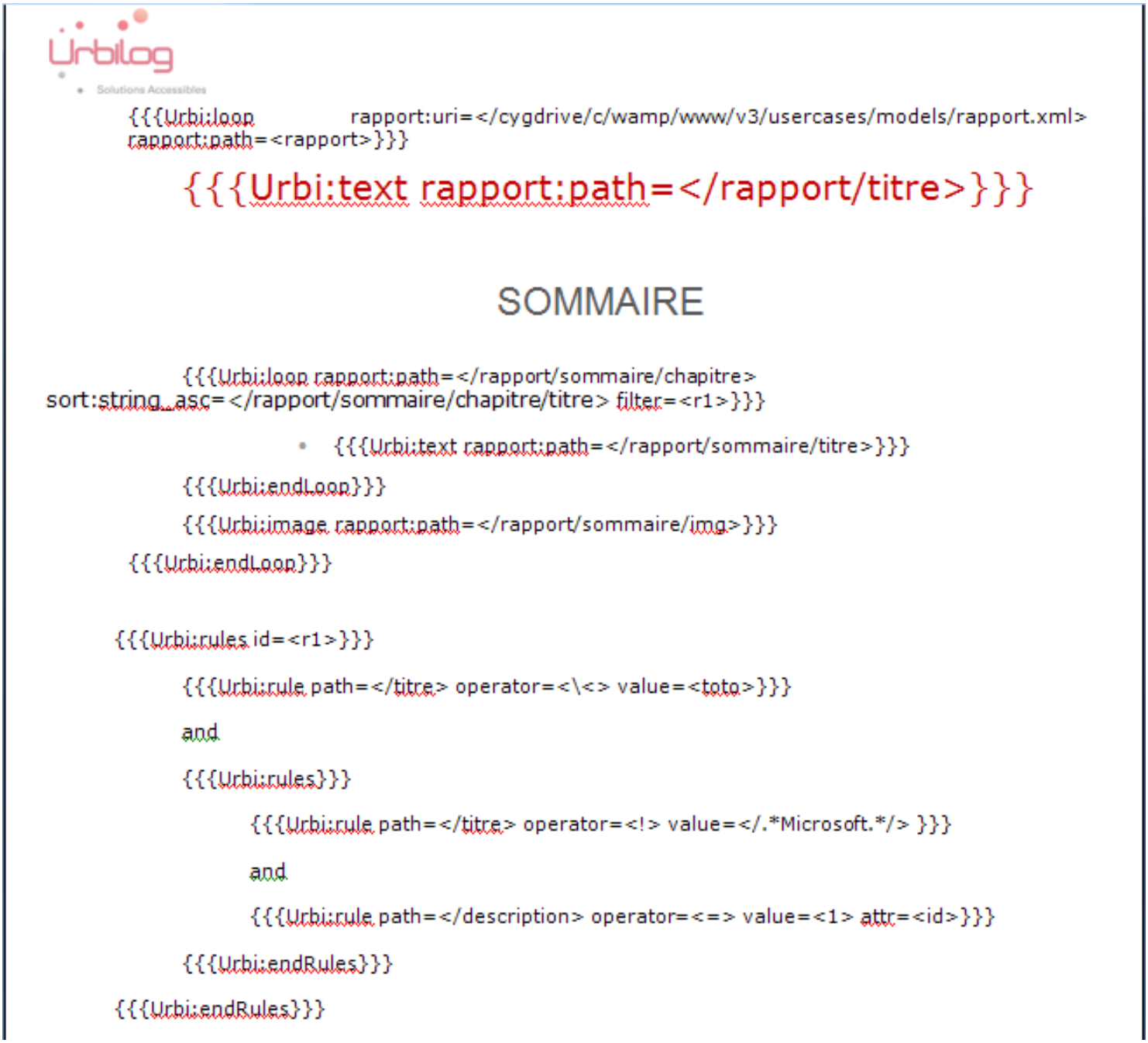
Le premier pas dans la génération d'un document est la création d'un template. Pour cela, vous pouvez utiliser soit Open Office ou Micro$oft Word. Dans notre cas, nous allons utiliser Micro$oft Word 2007.

Démarrez en créant un nouveau fichier dans Micro$oft Word 2007 et enregistrez ce template sous le nom de votre choix. Vous pouvez alors commencer à concevoir le modèle en y insérant du texte, un sommaire, un pied de page, un logo repris sur chaque page, définir un style pour les différents éléments du document, etc.



Au bout d'un moment vous devriez avoir un template contenant tous les styles du document final.

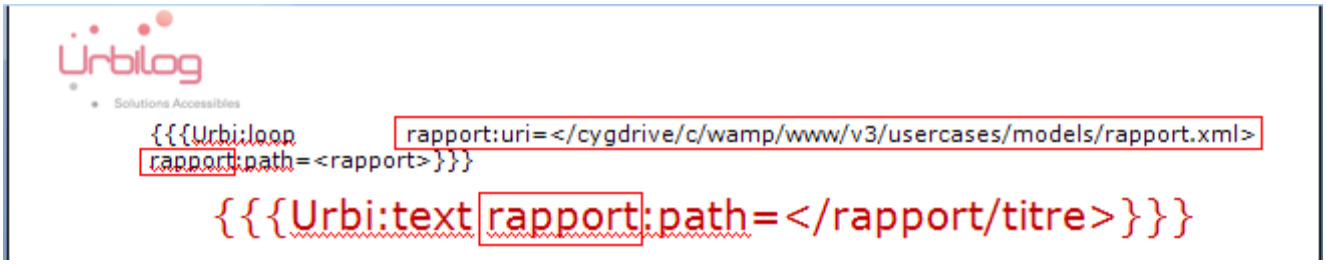
La prochaine étape est l'insertion de marqueurs qui seront remplacés par des données. La capture d'écran suivante vous montre à quoi peut ressembler un template fini, il nous servira d'exemple tout au long de cette documentation.



Ces marqueurs respectent une grammaire que nous avons inventée, pour comprendre cette grammaire il est nécessaire dans un premier temps de définir les différents marqueurs pouvant être utilisés.

Mais avant toute chose il est utile de faire quelques remarques concernant la syntaxe générale du template. En effet, vous l'aurez peut-être remarqué, mais nous n'utilisons pas les caractères " et " comme délimiteur de chaîne de caractère, mais les caractères < et >. Si vous voulez utiliser ces caractères pour autre chose que délimiter une chaîne, il suffit de mettre un « \ » devant pour le protéger. Dans la plupart des balises on doit préciser le chemin du fichier XML où se trouvent les données à insérer.

Pour éviter de réécrire le chemin complet à chaque fois, il est possible d'utiliser un alias que l'on met dans la première balise du document, la syntaxe pour cela est la suivante : **ALIAS**:uri=<**CHEMIN_DU_FICHIER**>.



Dans l'exemple ci-dessus je définis l'alias rapport qui correspond au fichier XML dont le chemin est /cygdrive/c/wamp/www/v3/usercases/models/rapport.XML puis j'utilise cet alias à deux reprises.

Et enfin, parfaite transition pour vous parler de la notation des chemins, partout dans l'api vous devez utiliser la notation cygdrive, c'est-à-dire séparer les répertoires par des « / » et les chemins doivent commencer par « /cygdrive ».

Je présenterai les marqueurs qui ont le plus d'intérêt sous la forme suivante : la syntaxe, suivie d'une brève description, la liste des options disponibles et souvent, un exemple d'utilisation. Par convention je mettrai en majuscule et en gras les choses qui ne sont pas fixes.

La majorité des marqueurs sont à la fois disponibles pour les docx et pptx, néanmoins ce n'est pas le cas de tous, soit par manque de temps soit parce que ce sont des balises spécifiques à un format. Par exemple cela n'aurait aucun sens d'avoir une balise slide pour les documents Word. Voici donc un tableau récapitulatif des types de marqueurs supportés avec leurs attributs disponibles :

Nom du marqueur	Attributs supportés pour les docx	Attributs supportés pour les pptx
Image	path	path
List	Balise non supportée	path
Loop	path, sort, filter	path, sort
loopRel	Balise non supportée	path, sort
Rules	id	Balise non supportée
Slide	Balise non supportée	id
Text	path	path

a.La balise image

Syntaxe :

{{{Urbi:image **ALIAS**:path=<**XPATH**>}}}

Description :

Comme son nom l'indique cette balise permet d'insérer des images, l'image est positionnée au même endroit que le marqueur qu'elle vient remplacer, de plus elle garde ses dimensions initiales. Seules les images au format png et gif sont supportées pour le moment. Si vous souhaitez insérer une image d'un autre format il vous faut dans un premier temps la convertir avec par exemple Paint ou Photoshop.

Attributs :

alias = alias du fichier XML dans lequel se trouve les informations à insérer

xpath = xpath permettant d'aller chercher l'information dans le fichier XML précisé juste avant (vous pouvez donc utiliser toutes les règles de xpath afin de sélectionner un nœud particulier)

Exemple :

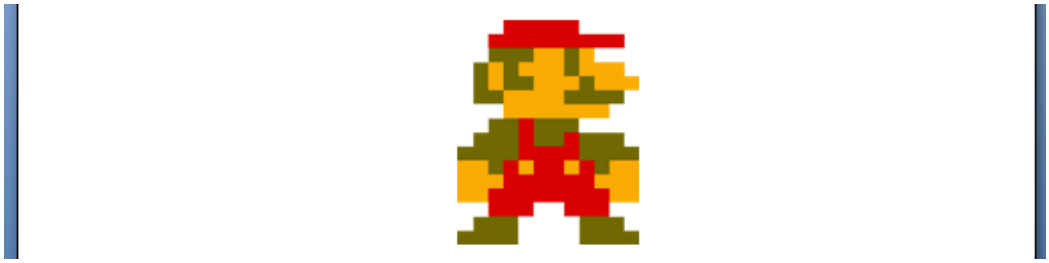
Reprenons notre exemple cité plus haut

```
{{{Urbi:endLoop}}}
{{{Urbi:image rapport:path=</rapport/sommaire/img>}}}
{{{Urbi:endLoop}}}
```

Ce marqueur sera remplacé par une image dont le chemin peut être retrouvé grâce au xpath « /rapport/sommaire/img » dans le fichier XML qui a pour alias rapport.

```
1  <?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>
2  <rapport>
3      <titre>Dcoumentation utilisateur</titre>
4      <sommaire>
5          <img src="/cygdrive/c/Documents and Settings/stagiaire/Mes documents/Mes images/pixel-mario1.png"/>
6          <chapitre id="1">
7              <titre>DESCRIPTION DE L'API</titre>
8          </chapitre>
9          <chapitre id="2">
10             <titre>DOCUMENTATION UTILISATEUR</titre>
11         </chapitre>
12         <chapitre id="3">
13             <titre>DOCUMENTATION TECHNIQUE</titre>
14         </chapitre>
15     </sommaire>
16 </rapport>
```

Produisant ainsi le morceau de document suivant :



b.La balise liste

Syntaxe :

{{{Urbi:liste **ALIAS**:path=<**XPATH**>}}}

Description :

La balise liste permet de créer … des listes. Cette balise s'utilise exactement comme la balise image, à l'exception que le xpath donné doit pointer sur un nœud dont les fils constitueront la liste. C'est une balise spécialement conçu pour les documents PowerPoint, qui créent automatiquement de nouvelles slides si la liste est trop grande pour tenir sur une seule slide.

Attributs :

alias = alias du fichier XML dans lequel se trouve les informations à insérer

xpath = xpath permettant d'aller chercher l'information dans le fichier XML précisé juste avant. Dans le cas d'une liste le xpath doit mener à un nœud dont les fils constitueront la liste. Chacun de ces fils doit posséder un attribut « lvl » représentant le niveau de profondeur du texte, si celui-ci n'est pas présent la valeur par défaut sera prise, c'est-à-dire zéro.

c.La balise loop

Syntaxe :

{{{Urbi:loop **ALIAS**:path=<**XPATH**> sort:**TYPE_DE_TRI**=<**XPATH_2**> Filter=<**ID_DU_FILTRE**>}}}

Description :

La balise loop permet de faire des boucles, c'est-à-dire répéter plusieurs fois une information. Cela peut paraitre sans grand intérêt mais c'est l'une des balises les plus complexes de l'api qui permet de générer des documents de plusieurs dizaines de pages à partir d'un template de 10 lignes. La balise permettant de marquer la fin d'une boucle s'appelle « endLoop », le principe est que l'on va prendre tout ce qui se trouve dans la boucle et le répéter un certain nombre de fois en changeant le xpath suivant notre position dans la boucle.

Attributs :

alias = alias du fichier XML dans lequel se trouve les informations à insérer

xpath = xpath permettant d'aller chercher l'information dans le fichier XML précisé juste avant. Dans le cas d'une boucle il permet de connaître la taille de la boucle et ainsi de faire varier le xpath au cours du temps.

Attributs optionnels :

TYPE_DE_TRI et XPATH_2 = il est possible de trier les groupes d'éléments répétés suivant l'ordre alphabétique ou numérique sur un nœud précisé par XPATH_2. TYPE_DE_TRI peut prendre comme valeur « string_asc », « string_desc », « numeric_asc » et « numeric_desc ».

Id_du_filtre = il est également possible de filtrer les groupes d'éléments répétés, c'est-à-dire ne pas afficher ceux qui ne respectent pas la règle dont l'identifiant est spécifié. Pour plus d'information voir les marqueurs « rules » et « rule ».

Exemple :

Reprenons notre fameux exemple

SOMMAIRE

{{{Urbi:loop rapport:path=</rapport/sommaire/chapitre>
sort:string_asc=</rapport/sommaire/chapitre/titre> filter=<r1>}}}

- {{{Urbi:text rapport:path=</rapport/sommaire/titre>}}}

{{{Urbi:endLoop}}}

Ces balises indiquent à l'api qu'il faut répéter tout ce qui se trouve entre le marqueur « loop » et le marqueur « endLoop ». Si l'on regarde le fichier XML associé, on observe que le xpath « /rapport/sommaire/chapitre » cible plusieurs nœuds, trois pour être exact.

```xml
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>
<rapport>
    <titre>Documentation utilisateur</titre>
    <sommaire>
        <img src="/cygdrive/c/Documents and Settings/stagiaire/Mes documents/Mes images/pixel-mario1.png"/>
        <chapitre id="1">
            <titre>DESCRIPTION DE L'API</titre>
        </chapitre>
        <chapitre id="2">
            <titre>DOCUMENTATION UTILISATEUR</titre>
        </chapitre>
        <chapitre id="3">
            <titre>DOCUMENTATION TECHNIQUE</titre>
        </chapitre>
    </sommaire>
</rapport>
```

Tout ce qui se trouve à l'intérieur de la boucle sera donc répété trois fois, la seule différence sera le xpath qui vaudra à tour de rôle « /rapport/sommaire/chapitre[1] », « /rapport/sommaire/chapitre[2] » et enfin « /rapport/sommaire/chapitre[3] », produisant le résultat suivant :

SOMMAIRE

- DESCRIPTION DE L'API
- DOCUMENTATION UTILISATEUR
- DOCUMENTATION TECHNIQUE

Là où cette balise devient extrêmement puissante est qu'à l'intérieur des boucles on peut mettre toutes les autres balises, y compris des boucles ou encore du contenu fixe qui sera répété comme une puce donnant l'illusion de créer une liste.

d.La balise loopRel

Syntaxe :

{{{Urbi:loopRel **FICHIER**:path=<**XPATH**> id=<**NUM_SLIDE**>}}}

Description :

Cette balise se comporte exactement comme une boucle normale à l'exception qu'au lieu de répéter ce qui se trouve à l'intérieur de la boucle celle-ci répète une slide. Pour connaître la taille de la boucle, l'api se base sur le nombre de nœuds que renvoie le xpath, et crée autant de nouvelles slides avec chacune un xpath différent.

Attributs :

alias = alias du fichier XML dans lequel se trouve les informations à insérer

xpath = xpath permettant d'aller chercher l'information dans le fichier XML précisé juste avant. Dans le cas d'une boucle il permet de connaître la taille de la boucle et ainsi de faire varier le xpath au cours du temps.

num_slide = identifiant de la slide à répéter. Pour donner un identifiant à une slide voir le marqueur slide.

e.La balise rules

Syntaxe :

{{{Urbi:rules id=<**ID_DE_LA_REGLE**>}}}

Description :

Cette balise permet d'écrire des règles et de les affecter ensuite à une boucle dans les documents Micro$oft Word. Une « rules » est constituée d'une « rule », d'un opérateur et d'une « rule ». Une « rule » est une expression qui peut être immédiatement évaluée et retourne un booléen. Ainsi la valeur d'une « rules » est l'évaluation de ces règles auxquelles on applique un opérateur. De plus une « rules » peut également être constituée de « rules ».

Attributs :

Id_de_la_regle = identifiant de la règle qui permet de l'utiliser dans une boucle

Exemple :

Je ne sais pas pourquoi, mais je sens que vous l'attendiez celui là. Dans l'exemple qui va suivre je définis une règle nommée r1 que j'affecte à une boucle. Pour cela il faut utiliser l'attribut « filter » des boucles et lui donner la même valeur que l'identifiant de la règle à appliquer.

```
{{{Urbi:loop rapport:path=</rapport/sommaire/chapitre>
sort:string asc=</rapport/sommaire/chapitre/titre> filter=<r1>}}}
    • {{{Urbi:text rapport:path=</rapport/sommaire/titre>}}}
{{{Urbi:endLoop}}}
    {{{Urbi:image rapport:path=</rapport/sommaire/img>}}}
{{{Urbi:endLoop}}}

{{{Urbi:rules id=<r1>}}}
Règle 1   {{{Urbi:rule path=</titre> operator=<\<> value=<toto>}}}
Opérateur and
          {{{Urbi:rules}}}
              {{{Urbi:rule path=</titre> operator=<!> value=</.*Microsoft.*/> }}}
              and
Règle 2       {{{Urbi:rule path=</description> operator=<=> value=<1> attr=<id>}}}
          {{{Urbi:endRules}}}
{{{Urbi:endRules}}}
```

Comme vous pouvez le constater la règle « r1 » est constituée de deux règles séparées par l'opérateur logique « ET ». Prenons les règles une par une, si l'on regarde la règle une, on s'aperçoit qu'elle est du type « rule » c'est donc une règle qui peut immédiatement être évaluée. Ses attributs sont :

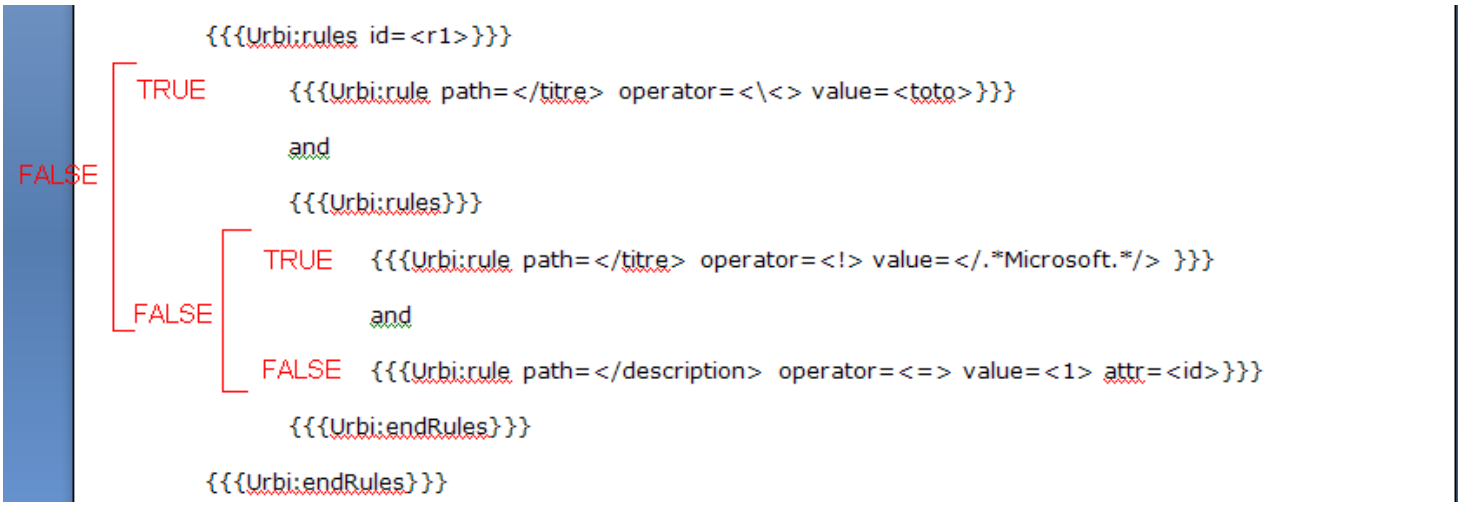
- un xpath qui doit mener à un nœud sur lequel la comparaison s'effectuera. Si vous souhaitez récupérer la valeur d'un attribut à la place de la valeur du nœud, vous pouvez soit utiliser la notation suivante dans le xpath « /description[@id] » soit utiliser l'attribut « attr » comme dans l'exemple ci-dessus.
- un opérateur de comparaison qui peut prendre les valeurs suivantes : <, >, =, !. Pour les opérateurs « = » et « ! » la valeur doit être une expression régulière.
- une valeur qui sera comparée à la valeur du nœud que le xpath cible

En français la règle une veut donc dire, je ne garde la ligne que si la valeur du nœud ciblé par « /rapport/sommaire/chapitre[NUM_DANS_LA_BOUCLE]/titre » est inférieur à la chaine « toto ».

Observons maintenant la règle deux, contrairement à la règle un, c'est une « rules », c'est-à-dire qu'elle est constituée de deux règles et d'un opérateur logique. La

valeur de cette règle sera donc l'évaluation des règles qui la composent auxquelles on applique l'opérateur logique « ET ».

Au final la règle « r1 » peut donc être traduite par : je ne garde la ligne que si la valeur du nœud ciblé par « /rapport/sommaire/chapitre[NUM_DANS_LA_BOUCLE]/titre » est inférieure à la chaîne « toto » et ne contient pas le mot « Microsoft » et dont la description possède un attribut « id » de valeur un.



Dans notre cas il n'y a pas de nœud description dans notre fichier XML, le résultat retourné vaudra donc toujours faux et aucune ligne ne sera affichée.

f.La balise slide

Syntaxe :

{{{Urbi:slide id=<**ID_DE_LA_SLIDE**>}}}

Description :

Vous avez pu avoir un aperçu de cette balise grâce au marqueur loopRel, en effet ce marqueur ne sert qu'à donner un identifiant à une boucle. Par exemple dans le cas d'une loopRel cela sert à préciser quelle slide répéter.

Attributs :

ID_DE_LA_SLIDE = une chaîne de caractère qui permet de distinguer de façon unique les différentes slides

g.La balise text

Syntaxe :

{{{Urbi:text **ALIAS**:path=<**XPATH**>}}}

Description :

Et enfin, voici le dernier marqueur supporté, mais pas des moindres, puisque c'est sans aucun doute la balise la plus utilisée, celle qui permet d'insérer du texte. Comme pour toutes les autres balises, si un style particulier (une police, une taille de police,…) est appliqué au marqueur, ce style sera également repris par le texte inséré.

Attributs :

alias = alias du fichier XML dans lequel se trouve les informations à insérer

xpath = xpath permettant d'aller chercher l'information dans le fichier XML précisé juste avant.

Exemple :

Un exemple spécifique pour la balise text n'est pas nécessaire puisqu'elle est utilisée dans les exemples cités plus haut. En revanche, voici à quoi ressemble le document complet une fois généré.

Jusqu'à présent cette documentation parle beaucoup des documents Word et peu des documents PowerPoint, pour la simple et bonne raison qu'il s'utilise presque de la même manière, à une ou deux exceptions près. La première est que lors que vous écrivez votre template PowerPoint vous devez impérativement mettre tous les marqueurs d'un slide dans le même champ texte. Un seul marqueur doit être placé dans son propre champ texte, c'est le marqueur « image ».

2. Modification des règles CLIPS

Actuellement CLIPS n'est utilisé dans l'api que dans le cas où l'on utilise le marqueur « list ». En fait l'utilisation de CLIPS s'est décidée lorsque j'ai été confronté à un problème lié au PowerPoint. En effet à la différence des documents Word, dans un document PowerPoint lorsque l'on arrive à la fin d'une slide, le texte ne continue pas sur une nouvelle slide. Et pire, il n'existe pas de moyen de savoir si l'on déborde d'une slide ou non. Ainsi dans un premier temps nous avons décidé d'utiliser CLIPS afin de poser des limites pour empêcher ces débordements et proposer une répartition plus judicieuse. Puis dans un second temps nous avons décidé d'élargir l'utilisation de CLIPS à un rôle de personnalisation, par exemple avec la possibilité de définir le style des listes à puces, en définissant un caractère pour chaque niveau de profondeur.

Pour l'heure, du point de vue utilisateur deux choses sont possibles, la première est de changer le style des listes à puces et la deuxième est de changer la façon dont est réparti, sur plusieurs slides, une liste trop grande.

Tout se passe dans le fichier def.clp du répertoire clips. Dans notre cas ce sont les premières lignes qui nous intéressent :

```
(defrule MAIN::focus
  =>
  (assert(nbLiBySlide 5))
  (assert(bulletStyle "•,o,>"))
  (focus MAIN MAKE_REPARTITION DISPLAY END)
)
```

C'est ce que l'on appelle une règle, néanmoins ne paniquez pas, il n'est aucunement nécessaire de connaître ce langage, seul quelques menus modifications sont éventuellement à effectuer. Cette règle spécifie juste le nombre maximum d'éléments d'une liste sur une slide, précise le style des listes à puces et appelle les différents modules qui composent ce fichier.

Ainsi si vous souhaitez changer le nombre maximum d'éléments d'une liste su r une slide, il suffit de remplacer dans « (assert(nbLiBySlide 5)) » le cinq par le nombre de votre choix. De même que pour changer le style des listes à puces il suffit de modifier les différents caractères de la chaîne « (assert(bulletStyle "•,o,>")) », sachant que le « • » représente le niveau zéro d'indentation et « > » le niveau deux.

Pour changer l'algorithme de répartition d'une liste il est nécessaire de comprendre le langage CLIPS (se référer à la partie technique de cette documentation), néanmoins si vous vous sentez d'attaque voici une partie de l'algorithme en question :

```
(defrule select_li_withChild
	(nbLiBySlide ?nb)
	?li <- (object(is-a TAG_LI)(childs $?clds)(num ?num&:(and (eq 0 (modulo ?num ?nb)) (neq ?num 0))))
	(test
		(> (length$ ?clds) 0)
	)
 =>
	(bind ?*stringRepartition* (str-cat ?*stringRepartition* (- ?num 1)))
	(bind ?*stringRepartition* (str-cat ?*stringRepartition* ","))
)
```

3. Utilisation de l'api

Maintenant que les règles et le template sont prêts, il ne reste plus qu'à insérer des données dans le template.

Cet exemple montre comment générer un document Word, la procédure est exactement la même pour générer un document PowerPoint à l'exception qu'il faut utiliser, non pas un objet de type DocxGenerator, mais un objet PptxGenerator.

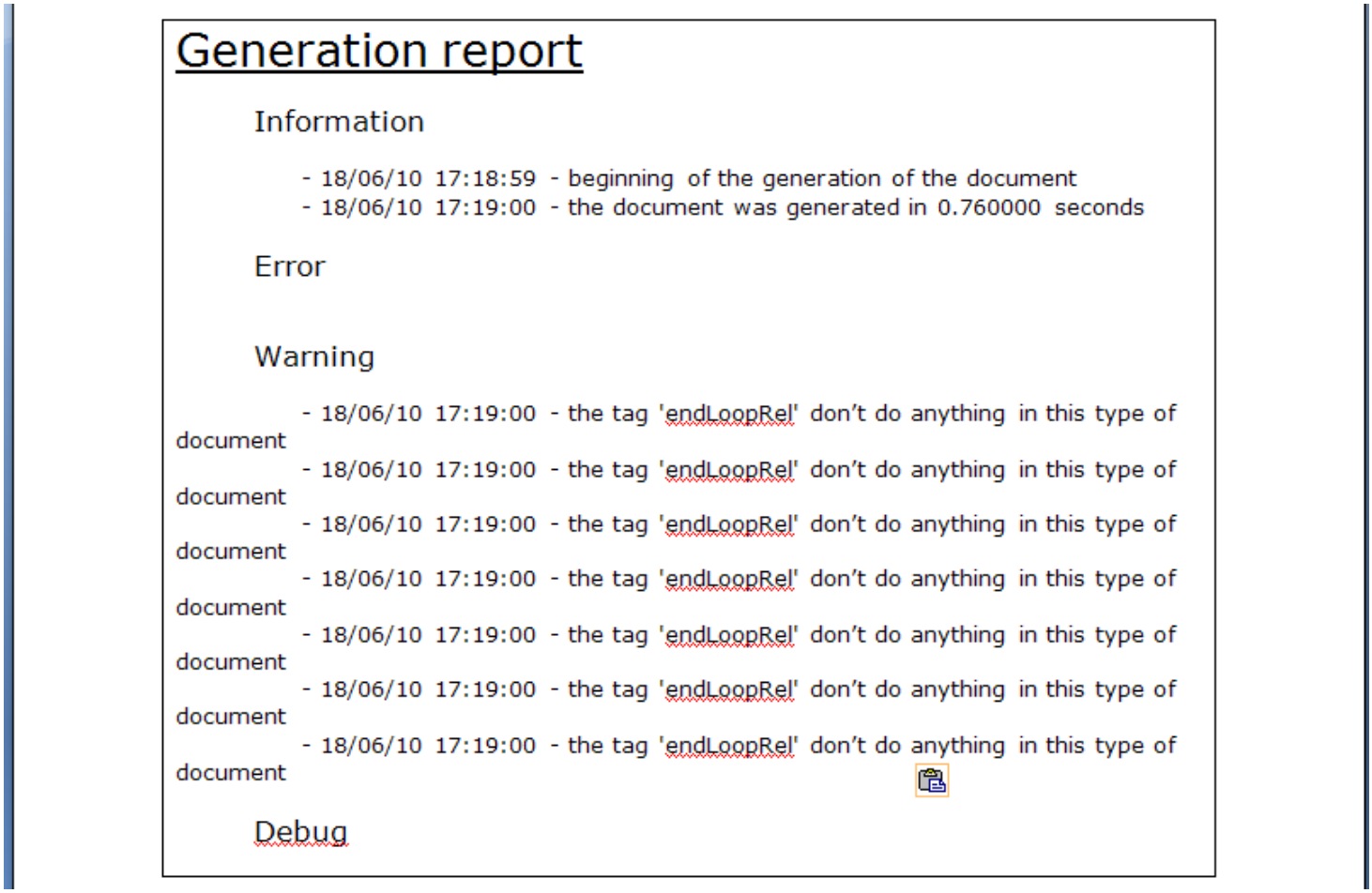
```
//$inputFile = emplacement du template
//$outputName = emplacement du fichier généré

$documentGen = new DocxGenerator() ;
$documentGen->setLocalTemplate($inputFile);
$documentGen->setOutputFile($outputName);
 $documentGen->createDocument() ;
```

Pour vous faciliter la tâche j'ai écrit un lanceur qui permet de lancer le traitement facilement, la syntaxe de la commande est la suivante :

```
php_win\php.exe –q launcher.php –i "chemin_du_fichier_entrée" -o "chemin_et_nom_fichier_sortie"
```

Quelque soit la façon que vous choisissez pou r lancer le traitement vous devez **impérativement** vous trouver à la racine du projet.

Le document résultant est enregistré sur le disque à l'emplacement spécifié et peut être ouvert dans votre lecteur favori, tel que le Micro$oft Word. A noter qu'à la fin du document fraichement créé se trouve un rapport dans lequel se trouvent toutes les remarques générées lors du traitement. Si le template contient trop d'erreurs, l'insertion des données sera interrompue et le document ne contiendra que ce rapport afin de vous aider à corriger le template utilisé.



Generation report

Information

- 18/06/10 17:18:59 - beginning of the generation of the document
- 18/06/10 17:19:00 - the document was generated in 0.760000 seconds

Error

Warning

- 18/06/10 17:19:00 - the tag 'endLoopRel' don't do anything in this type of document
- 18/06/10 17:19:00 - the tag 'endLoopRel' don't do anything in this type of document
- 18/06/10 17:19:00 - the tag 'endLoopRel' don't do anything in this type of document
- 18/06/10 17:19:00 - the tag 'endLoopRel' don't do anything in this type of document
- 18/06/10 17:19:00 - the tag 'endLoopRel' don't do anything in this type of document
- 18/06/10 17:19:00 - the tag 'endLoopRel' don't do anything in this type of document
- 18/06/10 17:19:00 - the tag 'endLoopRel' don't do anything in this type of document

Debug

Vous l'avez peut-être remarqué, mais le rapport de génération est en anglais, en effet il est possible de changer la langue de l'api. Pour cela il suffit de modifier le fichier « config.php » présent dans le répertoire « inc » et de changer la ligne « define('lang','en'); » par la ligne « define('lang','fr'); ». Dans ce fichier il est également possible de modifier tout un tas de paramètres tels que l'emplacement des fichiers temporaires, l'emplacement des fichiers de log, etc.

C. Documentation technique

L'objectif de cette partie est d'expliquer le fonctionnement de l'API afin de vous permettre de la modifier facilement. Ainsi dans un premier temps seront détaillés les différents packages qui constituent l'api, il vous sera ensuite expliqué comment ajouter de nouvelles fonctionnalités, comment effectuer des modifications sous CLIPS et une liste des différentes améliorations possibles conclura cette documentation.

1. Analyse

Le format docx et pptx sont très semblables, à part une petite différence au niveau du nom des namespaces utilisés par l'un ou par l'autre, on retrouve la même structure, les mêmes schémas. Dans la plupart vous trouvez donc une classe abstraite contenant des fonctions communes aux deux formats et dans les classes qui en hérite, tout ce qui diffère. En réalité la plus grosse différence réside dans le fait que le contenu d'un document Micro$oft Word est stocké dans un seul fichier, alors que dans le cas des documents Micro$oft PowerPoint il y a un fichier XML par slide.

L'API est constituée de quatre packages principaux : transformer, document, utils, une série d'autre petit package. Pour détailler les différentes classes de ces packages, j'expliquerai d'abord les classes liées à la conversion des marqueurs mis en WYSIWYG.

a. Le package transformer

Les classes liées à la conversion des marqueurs ont été les premières classes écrites. Le XML généré par Micro$oft Word est loin d'être épuré, ainsi les marqueurs mis sous Micro$oft Word, sont dans l'absolu difficilement manipulables puisque chaque marqueur se retrouve généralement réparti dans une multitude de balises XML, une première étape de conversion est donc nécessaire.

Dans l'exemple qui suit, le marqueur « {{{Urbi:text path=rgaa/directive/pc/evaluation/test}}} » se retrouve réparti dans deux runs différents.

```
- <w:r w:rsidRPr="00B66EF6">
  - <w:rPr>
      <w:sz w:val="20" />
      <w:szCs w:val="20" />
      <w:lang w:val="en-US" />
    </w:rPr>
    <w:t>{{{Urbi:text path=rgaa/directive/pc/evaluation/test/</w:t>
  </w:r>
- <w:r>
  - <w:rPr>
      <w:sz w:val="20" />
      <w:szCs w:val="20" />
      <w:lang w:val="en-US" />
    </w:rPr>
    <w:t>titre}}}</w:t>
  </w:r>
```

Le rôle des classes de ce package est donc de réunir chaque marqueur en une seule balise XML dont la syntaxe est la suivante :

<Urbi type='TYPE_DU_MARQUEUR' NOM_ATTRIUBUT='VALEUR_ATTRIBUT'}}}

Dans l'exemple ci-dessus cela donnerait :

<Urbi type='text' path='rgaa/directive/pc/evaluation/test/>

b. Le package document

Ce package regroupe principalement les fonctions permettant l'insertion des données dans le template de départ, c'est sans aucun doute la plus grosse partie de l'API. Le principe est simple, la méthode parsing du document va venir découper le document en une foultitude de document fils, chacun de ces fils peut avoir également des documents fils etc. Ces documents sont reliés entre eux par les managers présents dans le sous package manager et stockent les documents fils dans un tableau. L'avantage de cette méthode est que l'on peut maintenant trier et filtrer ces documents facilement, on peut par exemple trier par ordre alphabétique les sous documents sur leur titre (toutes ces informations doivent être précisées dans le template).

Une fois que la création des documents fils est finie, il suffit d'appeler la méthode «regenerate» qui va rassembler tous les documents entre eux en remplaçant au passage les marqueurs par les données en provenance des différents fichiers XML. En fait je me suis basé sur le principe du toString, mais en l'adaptant, c'est-à-dire que le premier document va appeler sa propre méthode toString qui va appeler la méthode toString de ses fils et ainsi de suite, jusqu'à tomber sur des feuilles c'est-à-dire un document qui ne possède pas de fils. Ensuite cela va remonter récursivement une énorme chaine de caractère par laquelle on va venir remplacer le contenu du document. Les classes s'occupant de faire ce remplacement s'appellent de la même manière que leur marqueur associé et sont chargés automatiquement par une factory.

c. Le package utils

Comme son nom l'indique ce package est composé de classes contenant une pléthore de fonctions statiques utiles tout au long de l'API. Comme par exemple la classe ZipUtils qui permet de créer et décompresser des archives au format zip.

L'API ne manipule presque que des données XML, ainsi il fallait trouver un moyen efficace de traiter les fichiers XML. Cela passe par la classe FileUtils qui s'occupe de charger un document XML et de renvoyer un objet DomDocument permettant de manipuler ce fichier. Le traitement n'est malheureusement pas instantané, me servant plusieurs fois des mêmes fichiers XML, une astuce pleine de bon sens a été de mettre dans un tableau tout les documents ouverts. Ainsi lorsque l'on demande à la fonction de parser le fichier XML elle regarde d'abord s'il n'est pas déjà chargé. Un gros travail d'optimisation a été réalisé afin de garder des

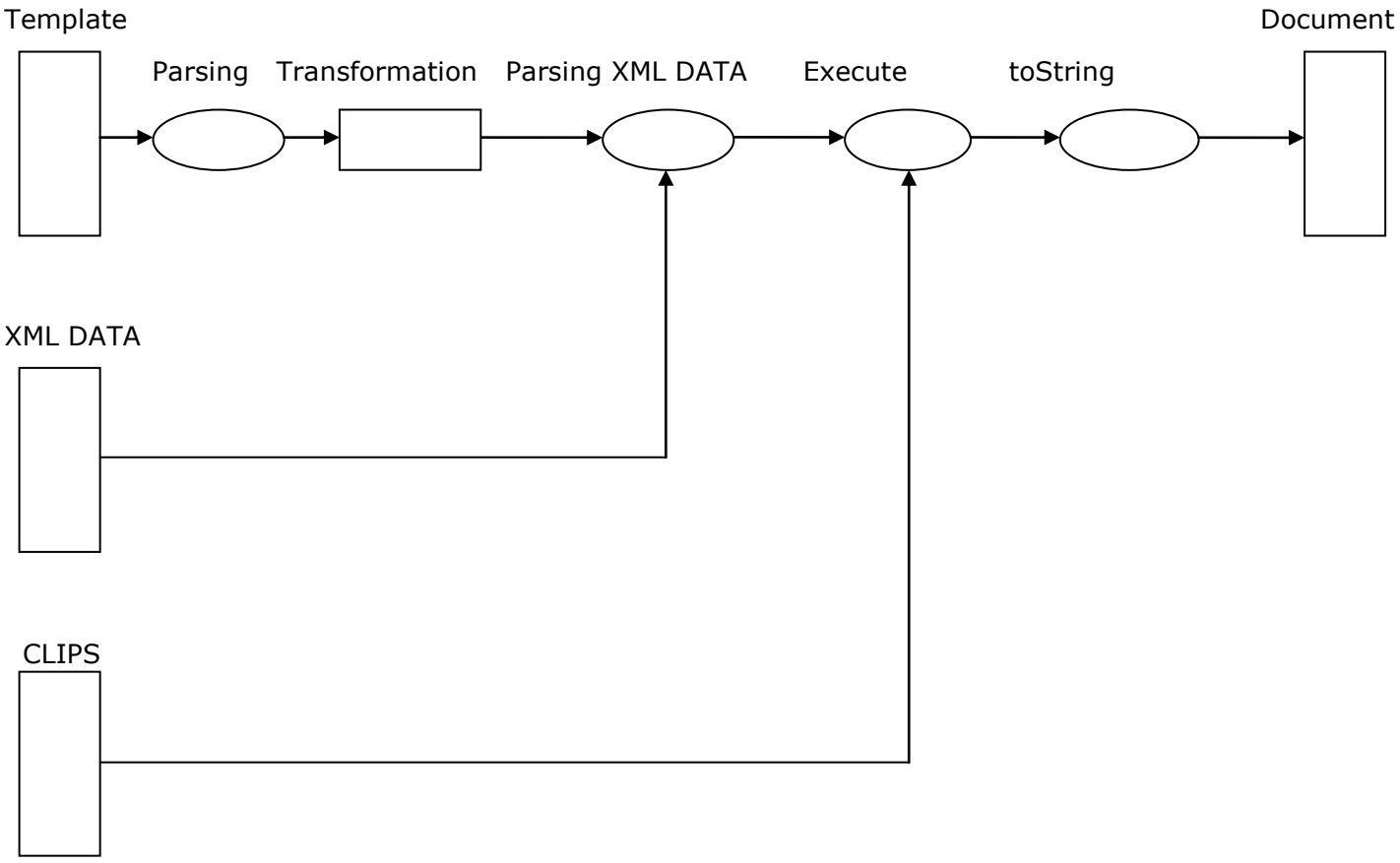
performances respectables, en effet la moindre fonction mal écrite peut provoquer une perte de performances importantes sur les gros documents.

Enfin, la classe AlgoUtils contient toutes sortes de fonctions qui effectuent des traitements algorithmiques spécifiques.

d. Les autres packages

Ce sont des packages qui contiennent peu d'objets, mais qui sont très utiles, comme le package log qui permet de créer des fichiers de log aussi bien dans le format html, dans un fichier texte ou encore dans un document Open XML, afin par exemple d'insérer à la fin d'un document généré grâce à l'API, un rapport de génération avec différentes remarques. La classe Clips permet de faire le pont entre le moteur d'inférence et PHP et TxtToXML permet de transformer un fichier texte en un fichier XML, afin de patienter en attendant que le projet Assigma soit fini.

Afin de résumer tout cela, voici un schéma récapitulatif, montrant l'enchainement des différentes étapes clefs.



2. L'api

Dans cette partie il vous sera expliqué comment ajouter le support de nouveaux marqueurs ainsi que la façon de modifier les algorithmes de comparaison pour les balises « rules » et « rule ».

a. Ajouter de nouveaux marqueurs

L'API a été écrite de façon à ce qu'il soit facile de rajouter de nouvelles balises, notamment grâce à l'utilisation de design patterns. Ainsi dès qu'un marqueur est utilisé dans un template, une factory essaye de charger une classe portant le même nom et implémentant l'interface « IType ». C'est une interface que tout marqueur doit implémenter et qui spécifie que ces classes doivent posséder deux fonctions particulières : « toDocx » et « toPptx ». Chacun de ces deux méthodes à pour objectif de définir le comportement du marqueur suivant son format.

Pour rendre cela un peu plus concret voici un exemple détaillé de A à Z un qui explique comment gérer un nouveau marqueur. Dans cet exemple un marqueur censure est créé, c'est un marqueur qui agit comme la balise « text » sauf que le texte inséré est filtré, c'est-à-dire que les mots qui ne nous plaisent pas seront remplacés par des « **** ».

La première étape est de créer une classe « censure » dans le dossier « document/type » et de faire implémenter cette classe l'interface « IType ». Cela devrait ressembler à cela pour l'instant :

```
class censure implements IType{


    public function __construct(){}


    public function toDocx($nodeToReplace,$document){}
    public function toPptx($nodeToReplace,$document){}


}
```

Dans les fonctions « toDocx » et « « toPptx », $nodeToReplace est l'instance d'un DomNode représentant notre marqueur « cencure », quant à $document, c'est un objet de type Document contenant toutes les informations relatives au document courant.

La prochaine étape est bien sûr d'écrire ces fameuses fonctions « toDocx » et « toPptx ». Dans notre cas ces fonctions doivent :

- Récupérer la valeur à insérer grâce au marqueur représenté par $nodeToReplace
- Filtrer ce message
- Créer un nouveau nœud texte avec le message filtré à l'intérieur

- Remplacer $nodeToReplace par le nœud texte fraichement créé

Commençons par récupérer la valeur à insérer, pour cela il nous faut un XPath nous menant à cette valeur. La plupart du temps le XPath est créé en concaténant le XPath du document, dans lequel nous nous trouvons, et le XPath du marqueur. La fonction permettant cela ressemblerait donc à :

```
private function getValue($tagPath,$documentPath,$inputDom){

        $path = ereg_replace("\\[[0-9]*\\]","",$documentPath);
        $subject=substr($tagPath,strlen($path));

        if($subject){
                $xPath = $documentPath."/".$subject;
        }else{
                $xPath = $documentPath;
        }

        $domXPath = FileUtils::loadDomXPath($inputDom);
        $emps = FileUtils::executeQuery($domXPath,$xPath);

        //if it's not text we don't take it
        if($emps->item(0)->firstChild->nodeName != '#text'){
                return '';
        }
        return $emps->item(0)->nodeValue;
 }
```

Maintenant que nous possédons la valeur à insérer, nous pouvons la censurer. Pour cela une simple fonction dans ce style suffit :

```
private function censure($string){
        $censoredWords = array('Microsoft','Apple') ;
        foreach($censoredWords as $cw){
                $string = ereg_replace ($cw,'***',$string);
```

```
            }
            return $string;
        }
```

Il ne reste plus qu'à remplacer $nodeToReplace par un nouveau nœud texte contenant cette chaine filtrée.

```
public function toDocx($nodeToReplace,$document){

        //we create the Xpath and get the value
        $inputDom=$this->getInputDom($nodeToReplace,DocxDocument::$uriList);
        $value=$this->getValue($nodeToReplacebutes->attributes
->getNamedItem("path"->nodeValue,$document->getXpath(),$inputDom);

        //we censure the value
        $value = $this->censure($value);

        //we replace the tag but a new node containing our censored value
        $dom = $document->getDom();
        $node = $dom->createElement("w:t");
        $nodeText = $dom->createTextNode($value);
        $node->appendChild($nodeText);
        $nodeToReplace->parentNode->replaceChild($node,$nodeToReplace);
}
```

Et voilà notre marqueur censure est prêt et peut, dès à présent, être utilisé dans des templates. Et bon appétit bien sûr.

b. Changer les algorithmes de comparaison des règles

Comme expliqué plus haut dans la partie utilisateur, il existe un marqueur « rules » qui permet de filtrer les éléments d'une boucle, c'est-à-dire ne pas afficher les lignes qui ne respectent pas la règle associée. Cela permet par exemple de ne pas afficher les lignes qui contiennent le mot « Microsoft ». Lorsque l'on écrit une règle on doit spécifier un opérateur de comparaison qui peut prendre les valeurs suivantes : <, >, =, !. Chacun de ces opérateurs possède une fonction PHP associé dont il est possible de redéfinir le comportement. Pour cela il suffit de changer

l'implémentation des fonctions « sup, inf, equals, dif » de la classe « rule » qui se situent dans le dossier « /document/Type ».

3. CLIPS

Avant toute chose, si vous ne connaissez pas CLIPS vous devriez d'abord vous rendre sur cette page : http://clipsrules.sourceforge.net/OnlineDocs.html.

En règle général CLIPS s'utilise de façon interactive, mais comme vous vous en doutez il est hors de question d'engager une personne pour taper les règles et faits dès qu'un programme fait appel à CLIPS. Heureusement il est possible de stocker ces règles et ces faits dans des fichiers, puis de les faire charger par CLIPS. Ainsi la méthode que nous utilisons pour passer des faits à CLIPS est de créer un fichier, contenant nos faits, en PHP puis de le charger dans CLIPS grâce à la fonction clips_load. Ce principe a donné naissance à la classe « Clips » du répertoire « /clips » qui permet simplement de charger un fichier de règle ainsi qu'un fichier de faits grâce au constructeur.

C'est bien beau tout cela me direz vous, mais avant de pouvoir charger un fichier de règle, il faudrait d'abord le créer. Actuellement les règles CLIPS écrites concernent uniquement les listes et permettent de calculer une répartition judicieuse de cette liste sur plusieurs slides. Pour cela j'ai modélisé sous CLIPS une liste en objet, qui lors de l'utilisation est instanciée grâce au fichier créé en PHP, sur laquelle on peut écrire toutes sortes de règles.

Voici la définition de la liste :

```
(defclass TAG (is-a USER)
        (multislot childs (create-accessor read-write))
        (slot num (create-accessor read-write) (type INTEGER))
)


(defclass TAG_LI (is-a TAG)  (role concrete) (pattern-match reactive)
        (slot tag_name  (create-accessor read) (default "li"))
        (slot att_lvl (create-accessor read-write) (type INTEGER))
        (slot att_value (create-accessor read-write) (default ""))
)
```

Et voici à quoi ressemble une liste instanciée :

```
(definstances LIST_INST (list_1 of TAG_LI  (att_lvl  1)(num  1))(list_2  of  TAG_LI
(att_lvl 1)(num 2)(childs [list_3] [list_4] ))(list_3 of TAG_LI  (att_lvl  2)(num
3))(list_4 of TAG_LI  (att_lvl 2)(num 4))(list_5 of TAG_LI  (att_lvl  1)(num  5)(childs
[list_6] [list_7] [list_8] ))(list_6 of TAG_LI (att_lvl 2)(num 6))(list_7 of TAG_LI  (att_lvl
2)(num 7))(list_8 of TAG_LI  (att_lvl 2)(num 8)))
```

A partir de là, si les règles de réparation écrites ne vous plaise pas, ou que vous souhaitez ajouter de nouvelles règles pour gérer des cas particuliers, il vous suffit de modifier le fichier def.cpl dans le répertoire « /clips ».

Enfin, la méthode que j'utilise pour récupérer les messages envoyés par CLIPS est la suivante. Je récupère dans un tableau statique les messages envoyés par CLIPS sur la sortie "wclips", puis j'écris des fonctions récupérant les différentes informations qui m'intéressent.

Dans l'ordre j'utilise cette ligne dans les règles CLIPS pour envoyer un message sur la sortie « wclips ».

```
printout wclips ?param "=" ?msg "___EOF___" crlf
```

Puis cette fonction callback sert à récupérer puis ajouter ces messages dans le tableau statique $results :

```
function message($router,$msg) {
        if($router == "wclips"){
                if($msg != "___EOF___"){
                        Clips::$results[]=$msg;
                }
        }
}
```

Et enfin j'utilise une fonction dans ce style pour renvoyer l'information souhaitée :

```
public function getRepartition(){
        return split(",",self::$results[array_search('repartition',self::$results)+2]);
}
```

4. Améliorations

Voici la liste des améliorations qui pourraient être apportées à l'API avec éventuellement des pistes :

- Écriture de nouvelles règles CLIPS afin d'augmenter l'impact des règles sur la façon dont sont générés les documents
- Gérer de façon plus judicieuse les erreurs et être plus explicite dans les messages
- Ecriture du marqueur « Liste » pour les documents Word
  - Pas forcement nécessaire puisque l'on peut toujours utiliser une boucle et un marqueur texte, devant lequel on met un « • », pour faire une liste
- Avant de commencer à manipuler le template faire une première étape d'analyse syntaxique pour vérifier qu'il n'y a pas d'erreur dans le template.
  - automate à pile ?

- La balise « image » ne peut insérer que des images du type png et gif pour l'instant.
    - Word et PowerPoint ne gère que les images png et gif, la seule solution est donc de convertir, dans un de ces formats l'image que l'on nous donne en entrée, par exemple avec la librairie php GD.
- Gérer les « rules » et « rule » pour les documents PowerPoint.
- Pour les documents PowerPoint, écrire un fonction toPpt() dans la classe « simpleLog » qui insère à la fin d'un document un rapport contenant les différentes remarques que le traitement a provoqué.
- Écriture d'une fonction qui vérifie que le document généré est valide
    - Dans un premier temps, déjà vérifier qu'il n'y a plus de marqueur Urbi
- Gérer la balise commentaire pour les documents PowerPoint qui permetrait d'insérer des commentaires sur une slide.
    - Fichiers a modifier :
        - [content_types].xml
        - notesSlideX.xml
        - notesSlideX.xml.rel
        - slideX.xml.rels