A quoi sert XSLT ?

XSLT (Extensible Style Language Transformations) est, comme son nom l'indique, un langage destiné à *transformer* un fichier XML en quelque chose d'autre. Ce quelque chose d'autre sera le plus souvent un fichier XML ou HTML. Mais ce pourra être tout aussi bien un fichier d'un autre format : par exemple du texte pur, ou du *Rich Text Format*...

Les avantages de XSLT

Ils sont énormes. Par la grâce de *XML* les fichiers de données d'une part, et les documents d'autre part, deviennent une seule et même chose. Par la magie de *XSLT* les uns et les autres peuvent être manipulés à volonté de façon automatique, et ce grâce à un langage certes complexe mais néanmoins accessible au non programmeur, puique seulement *déclaratif*. Ce qui veut dire que nous en avons désormais fini avec les tâches répétitives effectuées manuellement sur nos documents ! Et que tout fichier "hérité", quel que soit son format d'origine -- sous réserve qu'il puisse être d'abord transformé au format HTML (la transformation en XML "clone" n'étant ensuite qu'une formalité), ou au format "comma separated values" (voir notre feuille de style de transformation CSV -> XML) -- va pouvoir :

1. être transformé en XML "propre" (c'est-à-dire reflétant la structure *intrinsèque* de l'information qu'il contient et non plus une présentation plus ou moins arbitraire de cette information)
2. être secondairement, selon les besoins, transformé en fichiers affichables sur quelque support que ce soit (papier, microordinateur, téléphone portable...)

Comme il vient d'être dit, XSLT est en lui-même un langage très puissant et accessible au non programmeur. Mais, dans la mesure où ce langage comporte encore quelques déficiences, ou ne traite pas certains cas particuliers, les programmeurs pourront continuer à se faire plaisir en profitant des extensions propriétaires proposées par les moteurs de transformation du marché qui en élargissent encore les possibilités -- en particulier en y ajoutant des possiblités de scriptage...

Les caractéristiques de XSLT

Les deux caractéristiques principales de XSLT sont les suivantes :

- c'est un langage *déclaratif* et non *procédural*. Ce qui revient à dire qu'à la différence d'un langage de programmation classique, il ne spécifie pas le *comment ?* (les algoritmes) : il se contente de déclarer le *quoi ?* Par exemple :
  - que tout ou partie des balises <para> présentes dans le XML source sont à remplacer dans le HTML cible par des balises <p>
  - que telle partie de l'arbre XML source doit être reproduite telle quelle dans l'arbre XML résultat, ou bien déplacée, ou bien encore dupliquée...
- il est lui-même écrit en XML. Ce qui veut dire qu'il pourra être à son tour transformé par une nouvelle feuille de style XSLT, et ainsi de suite, à l'infini ! Ou bien encore qu'il pourra être manipulé à l'aide de tout langage de programmation qu'on voudra, pourvu que ce langage implémente l'interface *Document Object Model* (DOM)...

A côté de sa syntaxe propre, XSLT fait aussi appel à un second langage, déclaratif lui aussi : XPath. XPath sert à spécifier des chemins de localisation à l'intérieur d'un arbre XML (ainsi que desexpressions booléennes, numériques ou "chaîne de caractères" construites à partir de ces chemins), et fait l'objet d'une spécification distincte du W3C.

Un peu d'histoire : de XSL à XSLT et XSL-FO

Langages "orientés-contenu" et "orientés-présentation"

XML, comme chacun sait, est, de même que son grand ancêtre SGML, un langage de balisage universel. Il peut donc, comme SGML, servir à encapsuler toutes sortes de données -- à la seule condition qu'elles soient représentables sous forme d'arborescence. En particulier il peut parfaitement servir à encapsuler des données relatives à la manière de présenter des informations sur un support. C'est donc un raccourci un peu inexact de dire que XML est orienté-contenu et non pas orienté présentation -- puisque l'orienté-présentation est seulement un cas particulier de l'orienté-contenu ! Rappelons au passage que le langage de présentation favori du Web, HTML, est lui-même une application particulière de SGML (ce qui le rend à quelques détails près conforme à la syntaxe XML -- son successeur XHTML le sera complètement). Et qu'une myriade de nouveaux langages de présentation sont en train d'apparaître (XHTML, XSL-FO, SVG, X3D...) qui seront conformes à la syntaxe XML.

XML et feuilles de style

Ces clarifications apportées, il reste qu'un fichier XML n'est pas, en général, un fichier affichable/présentable en l'état. Il faut donc lui ajouter quelque chose pour que cet affichage soit possible. Ce quelque chose a été appelé "feuille de style", par une analogie un peu boiteuse avec les feuilles de styles stricto sensu -- comme les feuilles de style CSS ou les styles de MS Word -- qui servent à associer (de manière centralisée) des caractéristiques typographiques (marges, alignements, polices et tailles de caractères, couleurs, etc.) à un contenu déjà orienté-présentation.

En XML une feuille de style *stricto sensu* n'est bien entendu pas suffisante. Si votre XML contient, par exemple, une bibliographie, vous pouvez certes l'associer directement à un feuille de style CSS qui vous permettra, par exemple, d'associer à l'élément *auteur* la police Verdana 14 points et la couleur *teal*. Mais une telle feuIle de style CSS ne vous permettra pas de spécifier :

- que vous voulez que la bibliographie soit présentée sous la forme d'un tableau, ou sous la forme d'une liste ;
- qu'elle doit être classée selon tel ou tel critère ;
- que les différentes informations relatives à un même livre (auteur, titre, éditeur...) devront apparaître dans tel ou tel ordre, avec tels ou tels séparateurs, etc.

On voir par cet exemple que pour qu'un fichier XML puisse être affiché de manière réellement intéressante, il nous faut pouvoir spécifier :

- non seulement les objets de présentation *génériques,* tels que listes et tableau -- et à l'intérieur de ceux-ci l'italique, les sauts de ligne, etc -- qui vont être mis en oeuvre pour afficher son contenu ;

- mais encore, et surtout, la façon dont les parties constitutives du contenu (en l'occurence les livres et à l'intérieur de ceux-ci les auteurs, les titres etc.) vont être *distribuées* à l'intérieur de ces objets génériques -- dans quel ordre, selon quel classement, etc.

Dans le premier cas on parlera d'objets de formattage ou formatting-objects. Et nous constatons qu'HTML (ou plutôt le couple HTML + CSS) nous fournit d'ores et déjà de tels objets de formattage, à peu près suffisants tout au moins pour l'affichage sur écran.

Dans le second cas on parlera de *transformation*.

Cette analogie boiteuse qui avait été faite au départ avec les feuilles de styles *stricto sensu* explique que dans les premiers projets de spécification du W3C le langage de transformation propre à XML que nous appelons aujourd'hui XSLT a pu être mélangé, dans un projet des spécification unique baptisé à l'époque XSL (*Extensible Style Language*), à un tout autre langage. Cet autre langage étant, lui, destiné à définir des objets de formattage plus riches que ceux de HTML puisque destinés à de présenter un contenu XML sur les supports les plus variés (écran, mais aussi papier...)

Désormais les choses sont beaucoup plus claires, puisque les deux langages ont été séparés. L'un est devenu XSL Transformations (XSLT) et l'autre XSL-Formatting Objects (XSL-FO). Le premier seul est à l'heure actuelle arrivé au stade de spécification du W3C.

L'espace de nom (*namespace*) de XSL(T)

XSLT constitue un bel exemple d'utilisation de la philosophie des "espaces de nom" XML (XML namespaces). Toute feuille de style XSL(T) débute en effet (après la processing instruction xml) par une déclaration de l'espace de nom xsl :
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
ou <xsl:transform xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
(synonyme)

Deux avantages :

- Le préfixe xsl: va permettre de différentier à l'intérieur de la feuille de style les éléments qui appartiennent au langage XSLT de ceux qui devront apparaître dans le résultat final (les "éléments de résultat litéral")
- L'URI spécifiée dans la déclaration de l'espace de nom va éventuellement permettre de distinguer plusieurs implémentations de XSLT. (C'est ce que fait en particulier le nouveau moteur XSLT (MSXML3) de Microsoft et qui lui permet de traiter aussi bien des feuilles de style "IE5 natif" que XSLT 1.0)

Note. Le préfixe xsl: est le préfixe couramment utilisé mais n'est pas obligatoire. L'important est l'URI spécifiée dans sa déclaration. Ce qui suit serait donc valide :

```
<toto:transform xmlns:toto="http://www.w3.org/1999/XSL/Transform" version="1.0">
....
</toto:transform>
```

La philosophie des "règles modèle" (*templates*)

Qu'est-ce qu'une règle modèle (*template*)?

Comparaison avec les règles CSS

En dépit de ce qui vient d'être dit sur la différence entre CSS et XSLT, il est utile de partir de l'exemple des feuilles de style CSS pour bien comprendre comment fonctionnent les règles modèles XSLT.

On se rappelle qu'une feuille de style CSS se compose d'un certain nombre de **règles** (*rules*). Chacune de ces règles se composant :

1. d'un **sélecteur** (ex. p.commentaire em, qui signifie "les balises contenues dans une balise <p> de classe *commentaire*")
2. d'une ensemble de **déclarations** du type **propriété** (ex. color) - **valeur** (ex. yellow)

```
p.commentaire em {
    color: yellow
}
```

L'effet d'une règle CSS est que si un élément (une balise) du source HTML se trouve satisfaire à la condition exprimée par le sélecteur de cette règle, l'objet de formattage correspondant (paragraphe pour une balise <p>, cellule de table pour une balise <td>, etc.) se verra affecté des caractéristiques spécifiées par les déclarations de la règle (dans notre exemple la couleur jaune).
Mais il est important de comprendre qu'une feuille CSS ne fait que *décorer* un arbre HTML ; elle ne le modifie pas. Une règle CSS ne fait qu'*ajouter* des caractéristiques nouvelles (positionnement, couleurs, polices de caractères, etc.) à un objet de formattage qui aurait été généré de toutes façons, CSS ou pas. En conséquence une régle CSS vide (ne contenant aucune déclaration) sera simplement sans effet : lorsqu'une balise satisfera aux conditions exprimées par son sélecteur, l'objet de formattage associé sera généré exactement comme si la règle n'existait pas.

Autre caractéritistique des feuilles de style CSS : l'ordre d'apparition des règles dans la feuille de style est indifférent.

Enfin, lorsqu'une balise satisfait aux conditions exprimées par les sélecteurs de *plusieurs* régles, l'objet de formattage associé va être affecté de *la somme* des caractéristiques spécifiées par les déclarations listées dans ces différentes règles. En cas de conflit entre des déclarations, un mécanisme de priorité entrera en jeu : la déclaration qui est contenue dans la règle la plus spécifique (ex. p.commentaire em est plus spécifique que em) va l'emporter.

A première vue les règles modèles de XSLT ressemblent fort aux règles CSS :

1. elles comportent un sélecteur (ici un chemin de localisation XPath) qui va définir à quel(s) *noeud(s)* (éléments, attributs...) la règle s'applique ;
2. leur ordre d'appartion dans la feuille de style est indifférent ;
3. il existe un mécanisme de priorités pour régler les conflits entre règles s'appliquant à un même noeud.

```
<xsl:template match="para[@type='commentaire']//important">
    <span class="insister"><xsl:apply-templates /></span>
</xsl:template>
```

Mais il y a toutefois des différences notables :

1. Une régle modèle n'a pas pour fonction d'ajouter des caractéristiques nouvelles à un résultat prédéterminé : le résultat dépend *entièrement* de la règle modèle. Ainsi une régle modèle *vide* ne va générer aucun résultat. Lorsqu'un noeud (élément, attribut...) satisfera aux conditions exprimées par son sélecteur, *aucun* contenu correspondant (ni balise, ni attribut, ni texte) ne sera généré dans le fichier résultat : la règle modèle se comportera alors comme un filtre.
2. Caractéristique très importante : pour produire un résultat une règle modèle doit impérativement avoir été *invoquée* par une autre règle modèle.
3. Enfin, en cas de conflit *une seule* règle modèle s'appliquera (par application également d'un mécanisme de priorités), à l'exclusion de toutes les autres.

Toutes ces différences aboutissent à une première conséquence remarquable : alors qu'une feuille de style CSS vide est sans effet (le résultat est le même que si aucune feuille de style n'était associée), une feuille de style XSLT véritablement vide aurait, elle, pour effet de produire un résultat nul (un fichier vide).

Note. En réalité il n'existe pas de feuille de style XSLT *véritablement* vide, puisque la spécification XSLT a prévu qu'un certain nombre de règles modèles, dites régles modèles "internes", doivent exister par défaut dans toute feuille de style. Ces règles modèles, explicitées plus bas, vont faire qu'une feuille de style "vide" va néanmoins produire un résultat. A titre anecdotique on peut rappeler que le moteur de transformation XSLT d'"IE5 natif" ne respecte pas cette exigence, ce qui fait que ce moteur produit bien un résultat nul en réponse à une feuille de style XSLT vide.

De ce qui précède on retiendra deux choses essentielles :

- La feuille de style XSLT *ressemble* à la feuille de style CSS en ce qu'elle est constituée comme elle d'une suite de déclarations, d'ordre indifférent, dont chacune comporte un *filtre* (sélecteur pour CSS, chemin de localisation XPath pour XSLT) servant à distinguer les constituants du fichier source (éléments HTML pour CSS, noeuds XML pour XSLT) auxquels la déclaration s'applique de ceux auxquels elle ne s'applique pas.
- La feuille de style XSLT *diffère* de la feuille CSS en ce que ses différentes déclarations (règles modèles) doivent *s'appeler* les unes les autres, exactement comme dans un langage de programmation procédural un programme principal peut appeller des sous-programmes, qui eux-mêmes peuvent appeler des sous-programmes, etc. C'est ce mécanisme qui va permettre à la feuille XSLT de générer un arbre résultat éventuellement très différent de l'arbre source, alors que la feuille CSS ne sait que décorer un arbre source dont elle est bien incapable de modifier la structure.

Note. Deux conséquences importantes à ce mécanisme d'appel des règles modèles entre elles :

- Il faut bien une régle modèle "principale", qui soit appelée en premier et qui puisse ensuite donner la main aux autres. Cette règle principale est celle qui est associée à la racine (notée "/" dans le langage XPath) du document XML (nous dirons que cette règle est *positionnée* sur la racine du document). Cette première règle va en général en appeler d'autres qui auront toutes pour caractéristiques d'être elles aussi positionnées sur un noeud, ou un ensemble de noeuds

précis du document XSLT. Il faut bien en effet avoir présent à l'esprit qu'à tout moment de l'exécution d'une feuille XSLT le moteur de transformation va se trouver positionné sur *deux* documents/fichiers différents :

1. dans le *programme* que constitue la feuille de style : dans une règle modèle précise, et sur une instruction précise de ladite règle modèle ;
2. dans les *données* que constitue le fichier XML source : sur un noeud (élément, attribut...) précis.

- Contrairement à ce qui se passe dans un langage procédural, on a ici un mécanisme "piloté par les données" (*data driven)*. En ce sens que l'appel n'est pas un appel direct à une règle modèle (Note. un tel mécanisme existe en XSLT mais il y joue un rôle secondaire : c'est celui des *règles modèle nommées*), mais passe par deux étapes successives:
    - Rechercher dans le fichier XML les noeuds satisfaisant une condition précise (<xsl:apply-templates select="*motif XPath*"/>). Par défaut (absence de l'attribut "select") on recherche les éléments *fils* du noeud sur lequel on se trouve actuellement positionné.

      Note. Le motif XPath consiste essentiellement en un adressage, soit absolu soit relatif au noeud XML sur lequel on se trouve actuellement positionné, assorti éventuellement de conditions (dites *prédicats*).

    - Constituer la liste de ces noeuds qui devient la *liste de noeuds courante*, et éventuellement ordonner cette liste selon certains critères.
    - **Se positionner** successivement sur chacun des noeuds de cette liste (qui devient alors provisoirement le *noeud courant*), et pour chacun de ces noeuds :
        - rechercher les règles modèles s'appliquant à ce noeud ;
        - s'il y en a plusieurs déterminer la plus prioritaire ; et enfin
        - invoquer cette règle.

Bien entendu les règles ne font pas que se passer le relai les unes aux autres. De temps en temps il leur faut bien travailler! Un travail qui consiste tout naturellement à insérer du contenu dans le fichier cible :

- à l'emplacement du fichier cible où l'on se trouvait lorsque le modèle a été invoqué
- en exploitant le fichier source -- en adressage absolu ou relatif à partir du noeud courant.

Le prélèvement d'information sur le noeud contextuel sélectionnés pourra être :

- soit direct (<xsl:value-of...>)
- soir par appel d'une éventuelle règle modèle susceptible de s'y appliquer (<xsl:apply-templates.../>)

La dualité fichier source/fichier cible est ce qu'il y a de plus difficile à comprendre lorque vous mettez au point une feuille XSLT. Le fichier source a l'avantage d'être préexistant donc physiquement visible. Le fichier cible, quant à lui, est en gestation, et vous devez donc l'imaginer mentalement, ce qui n'est pas toujours facile, surtout s'il doit différer beaucoup dans sa structure du fichier source... Une chose à bien comprendre en tous cas est que le fichier source ne change pas au cours de la transformation XSLT. Le fichier cible, par contre, se construit progressivement, et vous devez, pour la construction de chaque règle modèle, imaginer où il en sera de sa construction lorsque la règle modèle sera invoquée. Pour cette raison, il est recommandé de construire les feuilles de style de manière incrémentale, règle modèle après règle modèle, en visualisant à chaque fois le résultat obtenu.

Qu'est ce qu'un noeud XML ?

Vu de XSLT, "noeud" (node) est le terme générique qui désigne sept objet différents parmi ceux que l'on rencontre à l'intérieur d'un fichier XML. A savoir

1. la *racine* du document -- l'objet qui englobe l'ensemble du document : à ne pas confondre avec l'élément (balise) de plus haut niveau, qui en est le fils, et qui est parfois appelé *élement racine*. En XPath la racine du document est représentée par le symbole /. Si une feuille XSLT déclare des règles modèles il est obligatoire qu'elle en déclare au moins une ayant pour cible la racine (mais il est possible de construire des feuilles XSLT ne déclarant aucune règle modèle et se comportant comme une unique règle modèle).
2. les *éléments* ou balises -- de loin les noeuds les plus importants. Ce sont les "branches" de l'arbre XML. En XPath ils sont symbolisés individuellement par leur nom, tout simplement, et collectivement par * (si le spécificateur d'axe qui précède n'est pas attribute:: ou namespace::)
3. les *noeuds textuels* -- les "feuilles" de l'arbre XML, où réside l'information "de base". Les noeuds textuels sont ce qui reste à l'intérieur d'une balise quand on a retiré les balises filles et leur contenu. Ils sont eux aussi considérés comme des noeuds "fils" de la balise qui les contient. En XPath ils sont symbolisés par text(). Note. Les valeurs d'attribut ne sont pas des noeuds textuels
4. les attributs -- les informations complémentaire inscrites (sous la forme étiquette="valeur") à l'intérieur même des balises. En XPath ils sont symbolisés individuellement par leur nom précédé de attribute:: ou de @, et collectivement par attribute::node() ou attribute::*, ou @*. Ils ne sont pas considérés comme des noeuds fils de l'élément qui les contient.
5. enfin, et pour mémoire : les *instruction de traitement* ("processing instructions"), les *commentaires*, et les *espaces de nom* ("namespaces"). En XPath ils sont symbolisés par processing-instruction('*cible*'), comment() et namespace::*préfixe*

A noter qu'en XPath node() désigne collectivement tous les types de noeuds.

Note. On lit parfois que node() désignerait collectivement tous les types de noeud à l'exception des attributs. C'est inexact. La confusion vient du fait que dans une étape de localisation node() non précédé d'un spécificateur d'axe est une abréviation de child::node(), et désigne donc tous les noeuds *fils* du noeud contextuel. Or les attributs n'en font pas partie. Pour les désigner collectivement il convient donc d'écrire attribute::node() ou attribute::* ou @*

Comment les règles modèles accèdent aux données XML : les *chemins de localisation* ou *motifs* XPath

Les règles modèles passent leur temps à recherche des noeuds dans le fichier source, on vient de le voir. Pour ce faire elles ont à leur disposition un outil de recherche puissant : le "chemin de localisation" (location path), parfois appelé aussi "motif" (pattern).

Qu'est-ce que le chemin de localisation/motif ? C'est, si vous voulez, l'objet que vous donnez à flairer à votre limier pour qu'il piste un gibier bien particulier... Ou si vous préférez des lunettes dont vous chaussez votre moteur de transformation et qui le rendent aveugle à tous les noeuds à l'exception de ceux, bien spécifiques, qui devront être sa proie unique. Le chemin de localisation s'écrit à l'aide du langage XPath déjà mentionné.

La syntaxe des chemins de localisation : les "étapes de localisation"

Un *chemin de localisation* XPath est constitué de :

- une ou plusieurs *étapes de localisation* XPath, séparées entre elles par le symbole /. A son tour chacune de ces étape de localisation est constituée de, dans l'ordre :

    1. un *axe* (par défaut l'axe implicite est child), suivi du séparateur :: , suivi de
    2. un *test de noeud*, suivi de
    3. de zero à *n prédicats*, encadrés chacun par [ ].

Un chemin de localisation XPath, lorsqu'il est écrit en syntaxe abrégée, ressemble beaucoup au chemin qui nous sert à spécifier l'emplacement d'un fichier sur le disque dur de notre ordinateur. Ils désigne une position (relative ou absolue) à l'intérieur d'une hiérarchie, en l'occurence une hiérarchie de noeuds, en faisant appel éventuellement à des "jokers".

- Exemple de chemin de localisation absolu : //biblio/*/livre/auteur va rendre le moteur de transformation aveugle à tout ce qui n'est pas une balise <auteur>, fille d'une balise <livre>, elle-même petite fille d'une balise <biblio>", laquelle pourra se trouver n'importe où à l'intérieur de l'arbre XML.
- Exemple de chemin relatif : ../adresse va limiter les recherches aux balises <adresse> qui sont filles de la balise mère de la balise "contextuelle" (celle sur laquelle on est actuellement positionné) -- autrement dit les soeurs de la balise contextuelle qui seraient des balises <adresse>, et la balise contextuelle elle-même au cas où elle serait une balise <adresse>.

Symboles et abréviations utilisés dans la syntaxe XPath des chemins de localisation

Symbole	Valeur
/	Séparateur d'étapes de localisation. Si ce symbole est en tête d'un chemin le chemin part de la racine du document (adressage absolu). Si ce symbole n'est pas suivi d'un spécificateur d'axe, il est équivalent à un séparateur parent-enfant (l'axe child étant alors implicite)
//	"Joker vertical". Abréviation de /descendant-or-self::node()/. Si ce symbole est en tête d'un chemin le chemin part de la racine du document (adressage absolu)
.	Désigne le noeud contextuel. Abréviation de self::node().En tête d'un chemin ./ est facultatif.
::	Sépare un spécificateur d'axe d'un test de noeud
..	Désigne l'élement parent du noeud contextuel. Abréviation de parent::node()
*	"Joker horizontal". Peut désigner l'ensemble d'une fratrie d'éléments (*) ou d'attributs (@* ou attribute::*), ou l'ensemble des espaces de nom en vigueur (namespace::*).

Symbole	Valeur
@	Préfixe des attributs. Abréviation de attribute::.
:	Séparateur de préfixe d'espace de nom (dans les noms d'élément ou d'attribut).
( )	regroupe des opérations qu'il rend prioritaires
[ ]	Encadre un prédicat
[ ]	Encadre un indice à l'intérieur d'une collection (= cas particulier de prédicat)
|	Opérateur booléen. Entre deux étapes de localisation, spécifie l'une **ou** l'autre de ces étapes de localisation

Les spécificateurs d'axe

Axe	Valeur	Abréviation correspondante
ancestor	les ancêtres du noeud contextuel	aucune
ancestor-or-self	idem, plus le noeud contextuel	aucune
attribute	les attributs du noeud contextuel	@ (équivaut strictement à attribute::)
child	les enfants du noeud contextuel	rien (axe par défaut)
descendant	les descendants du noeud contextuel	aucune
descendant-or-self	idem, plus le noeud contextuel	// (équivaut en fait à /descendant-or-self::node()/)
following	les éléments qui suivent le noeud contextuel (dans l'ordre du document)	aucune
following-sibling	idem, limité à la même fratrie	aucune

Axe	Valeur	Abréviation correspondante
namespace	les noeuds "espace de nom" du noeud contextuel	aucune
parent	le parent du noeud contextuel	.. (équivaut en fait à parent::node())
preceding	les éléments qui précèdent le noeud contextuel	aucune
preceding-sibling	idem, limité à la même fratrie	aucune
self	le noeud contextuel	. (équivaut en fait à self::node())

Les tests de noeud

Le test de noeud sert à spécifier un noeud, ou un ensemble de noeuds dans la collection de noeuds désignée par l'axe (explicite ou implicite) qui le précède. Le test de noeud peut être un nom (d'élément, d'attribut, ou d'espace de nom) ou un joker :

Joker	Valeur
*	tous les noeuds du *type de noeud principal* de l'axe (explicite ou implicite) qui précède. Autrement dit tous les noeuds de type : • *attribut* pour l'axe attribute ; • *espace de nom* pour l'axe namespace ; • *élément* pour tous les autres axes
node()	tous les noeuds quel que soit leur type
text()	tous les noeuds textuels
comment()	tous les noeuds de type commentaire
processing-instruction()	tous les noeuds de type instruction de traitement

Les filtres ou prédicats

Les prédicats vont ajouter des restrictions supplémentaires aux chemins. Ils vont rendre les motifs encore plus sélectifs. Ainsi //biblio/*/livre[@sujet="xml" and datepub="2000"]/auteur va restreindre la recherche précédente aux auteurs de livres dont le sujet (désigné par un attribut de la balise livre) est XML et l'année de publication (désignée par une balise fille du nom de "datepub") est l'an de grâce 2000.

Quelle est la syntaxe d'une règle modèle ?

Elle est la suivante :
<xsl:template match="*motif*"> <!-- Spécifie à quels noeuds la règle est applicable -->

<!-- Insertions dans le fichier cible de données prélevées/calculées à partir des données du fichier source. Appel éventuel d'autres règles modèles -->

</xsl:template>

Que fait une règle modèle une fois qu'elle est activée ?

Par défaut elle ne fait rien -- ce qui revient à dire que ce sur quoi elle est positionnée (et qui peut être un sous-arbre) sera absent (ne sera pas reproduit) dans le fichier cible. Ainsi, si vous avez écrit une règle modèle qui capture toutes les balises <para> et qui est vide, les balises <para> et tous leurs contenus (qui sont peut-être des sous-arbres) ne seront pas exploités/reproduits dans le fichier cible. (Si l'on accepte l'idée que celui-ci est une image, plus ou moins déformée, du fichier source, on peut dire que les balises <para> et tout leur contenu auront été effacés.)

Si vous souhaitez que le modèle fasse quelque chose, ne serait-ce que rendre la main à d'autres modèles, eh bien il faut le lui demander explicitement !

Vous pouvez, par exemple lui demander de simplement remplacer les balises <para> et tout ce qu'elles contiennent par le texte "trouvé !". En ce cas il vous suffira d'écrire :

<xsl:template match="//para">

trouvé !

</xsl:template>

Si vous désirez que ce texte "trouvé !" apparaisse à l'intérieur d'une nouvelle balise, par exemple , alors vous écrirez :
<xsl:template match="//para">

trouvé !

</xsl:template>

Si vous voulez que le texte qui se trouvait précédemment à l'intérieur de la balise <para> apparaisse maintenant à l'intérieur de la balise , vous écrirez :
<xsl:template match="//para">

<xsl:value-of select=".">

</xsl:template>

Mais ceci ne va reproduire que les noeuds textuels descendants de la balise <para>. Que se passera-t-il si la balise <para> contient des balises "descendantes" -- par exemple une balise <important>? Eh bien, ces balises descendantes seront ignorées et seul leur contenu textuel apparaîtra dans le résultat final.

Alors, comment faire ? C'est ici qu'apparaît tout l'intérêt de l'instruction "reine" de XSLT : <xsl:apply-templates /> .

Que fait cette instruction ? Eh bien, on l'a vu, elle permet à la règle modèle active de demander à la cantonade si, par hasard, d'autres règles modèles ne voudraient pas reprendre le travail là où elle l'a laissé... Et si il ne s'en trouve pas ? Eh bien tant pis, elle s'en désintéresse !

Vous l'aurez compris, XSLT est basé sur la division du travail et la coopération. En XSLT comme dans la vie, la division du travail et la coopération sont un peu difficiles à mettre en place et à faire fonctionner. Mais, toujours comme dans la vie, elles se révèlent une fois en place extrêmement efficaces et d'un fonctionnement très souple.

Prenez par exemple l'instruction que nous avons écrite ci-dessus dans notre modèle :

<xsl:value-of select=".">

Eh bien dans la philosophie coopérative qui est celle de XSLT, elle est inutile. On peut la remplacer par <xsl:apply-templates /> et créer par ailleurs une règle modèle générique unique chargée de la reproduction des noeuds textuels de tout un fichier XML. Cette règle modèle générique "reproductrice de feuilles" (qui fait fort opportunément partie des règles modèle internes de XSLT) aura l'allure suivante :

```
<xsl:template match="text()">
    <xsl:value-of select=".">
</xsl:template>
```

L'avantage de cette façon de travailler est que le <xsl:apply-templates /> en question va pouvoir faire face à tous les cas possibles :

- le cas où la balise <para> ne contient que du texte
- le cas ou elle contient des balises filles -- auquel cas il faudra bien entendu que des règles modèles spécifiques aient été définies pour traiter ces balises, du genre :

```
<xsl:template match="important">
    <em><xsl:apply-templates /></em>
</xsl:template>
```

Note. Les règles modèles capturant des balises petites-filles ne prendront pas la main. A moins que :

- des règles modèles capturant spécifiquement les balises filles existent et leur passent la main en faisant à leur tour <xsl:apply-templates />
- ou bien que l'on ait défini une règle modèle générique qui, faute de de modèles plus spécifique, va capturer les balises filles et leur faire passer la main à leurs propres filles et ainsi de suite. Cette règle modèle générique "parcoureuse d'arbre" à l'allure suivante :

```
<xsl:template match="*">
    <xsl:apply-templates />
</xsl:template>
```

Une telle règle modèle est vraiment utile. Si elle n'existe pas et que vous voulez effectuer un traitement pour un noeud donné, disons changer le nom des balises <toto> en <tata>, il va vous falloir :

- soit avoir des règles modèles qui ciblent tous les noeuds intermédiaires entre la racine du document source et ce noeud particulier, même si vous n'avez aucun traitement particulier à effectuer sur ces noeuds, auquel cas ces règles modèles ne contiendront que le fameux <xsl:apply-templates />
- soit effectuer un saut direct à partir d'une autre règlemodèle, en utilisant l'instruction <xsl:apply-templates select="*chemin*" />

Les règles modèles "internes"

Cette règle modèle générique "parcoureuse d'arbre" est tellement utile que la spécification XSLT a décidé de vous dispenser d'avoir à l'écrire en spécifiant que les moteurs de tranformation XSLT devront la considérer comme existant toujours par défaut, sous la forme encore plus générique suivante :

```
<xsl:template match="/ | *">
    <xsl:apply-templates />
</xsl:template>
```

Cette règle modèle spécifie que, à defaut de spécification contraire, l'arbre XML source doit être parcouru dans son intégralité, à partir de sa racine / et en traversant, dans l'ordre hiérarchique, tous les éléments (*) -- et non pas les attributs (@*)

La spécification XSLT demande encore que deux autres règles modèles génériques soient considérée elles aussi comme existant toujours par défaut : il s'agit d'une part d'une forme encore plus générique (puisqu'elle reproduit aussi bien les valeurs d'attributs que les noeuds textuels) de la règle modèle "reproductrice de feuilles" évoquée plus haut :

```
<xsl:template match="text() | @*">
    <xsl:value-of select=".">
</xsl:template>
```

Et d'autre part d'une règle modèle assurant la non reproduction des instructions de traitement et des commentaires :

```
<xsl:template match="processing-instruction() | comment()" />
```

A elles trois ces règles modèles par défaut, dites "règles modèle internes", assurent que une feuile de style XSLT vide va reproduire tout le contenu du fichier XML source, dans l'ordre. Autrement dit elle va dépouiller le fichier de tout ce qui est purement XML (balises -- y compris les attributs et leurs valeurs, processing instructions, commentaires).

Note 1. Les valeurs d'attribut ne seront pas reproduites parce que la première règle ne fait pas parcourir les attributs. Il y faudrait un <xsl:apply-templates select="@* | node()" />. Puisque :

1. <xsl:apply-templates /> est équivalent à <xsl:apply-templates select="node()" /> ;
2. node() est en fait l'abréviation de child::node() ;
3. les attributs ne sont pas considérés comme "enfants" de l'élément qui les comporte.

Note 2. Les contenus des noeuds textuels seront, eux, bien reproduits, puisque :

1. comme rappelé ci-dessus, <xsl:apply-templates /> est équivalent à <xsl:apply-templates select="child::node()" /> ;
2. les noeuds textuels sont considérés comme "enfants" de l'élément qui les comporte.

Ce ne serait pas le cas si dans la *première* règle, on avait un <xsl:apply-templates select="*" />. En ce cas la main ne serait pas donnée aux noeuds textuels ; la *deuxième* règle ne pourrait donc pas leur être appliquée ; donc leur contenu ne serait pas reproduit.

La règle modèle de transformation "à l'identique"

En revanche une règle modèle particulière, dite règle modèle de transformation "à l'identique" (voir aussi http://www.chez.com/xml/astuces/index.htm#ie5xslt), va permettre de reproduire l'intégralité du fichier XML source, y compris les balises et leurs attributs, les processing intructions, les commentaires, etc. Cette régle modèle est la suivante :

```
<xsl:template match="@* | node()">
        <xsl:copy>
            <xsl:apply-templates select="@* | node()" />
        </xsl:copy>
</xsl:template>
```

Cette règle modèle garantit que tout les noeuds de l'arbre XML source sont parcourus (y compris les attributs), que leurs étiquettes sont copiées (par la grâce de l'instruction <xsl:copy>) et leurs valeurs reproduites également (par l'instruction <xsl:copy> également lorsque la règle modèle arrive au niveau des feuilles)

Question. Pourquoi "@* | node()" et non pas seulement select="node()" ? Réponse ci-dessus.

Noeud courant, liste de noeuds courante, et noeud contextuel

Lorsqu'une règle modèle "appelante" passe le relai "à la cantonnade" en spécifiant un chemin de localisation XPath (<xsl:apply-templates select="..."/>), l'ensemble des noeuds qui satisfont à l'ensemble des conditions spécifiées par ce chemin de localisation va devenir pour le moteur de transformation XSLT la liste de noeuds courante. Le moteur va devoir s'intéresser successivement à chacun des noeuds de cette liste qui deviendra alors provisoirement le noeud courant (caractérisé par une certaine position à l'intérieur de la liste de noeuds courante), et pour chacun tâcher de trouver une règle modèle qui lui soit applicable, et passer le relai à cette règle. Ce noeud courant va constituer ensuite le premier contexte de travail de la règle modèle "appelée". Autrement dit, toutes les instructions de travail qui figurent à l'intérieur de cette règle modèle vont utiliser l'emplacement du noeud courant comme base d'adressage à chaque fois qu'elles feront appel à une adresse relative dans le fichier source.

Rappelons que ces instructions "de travail" peuvent être de deux sortes :

- Instructions d'*exploitation directe* du fichier cible. Ainsi, dans l'exemple précédent, l'instruction <xsl:value-of select="."> va reproduire les noeuds textuels *descendants* de la balise <para>trouvée par le modèle. Ces instructions ne modifient en général pas la liste de noeuds courante et le noeud courant. Une exception est l'instruction de répétition <xsl:for-each...>. Cette instruction va avoir pour effet de créer provisoirement une nouvelle liste de noeuds courante, dont les éléments vont devenir, chacun à leur tour, le nouveau noeud courant. A la fin de ce processus la liste de noeud courante et le noeud courant vont redevenir ce qu'ils étaient au début.
- Instruction d'invocation d'autres règles modèles. De même, si dans l'exemple précédent, l'instruction <xsl:value-of select="."> est remplacée par l'instruction <xsl:apply-templates />, celle-ci va passer la main à d'autres règles modèles à la condition que celles-ci capturent (s'appliquent à) des noeuds *fils* de la balise <para> trouvée par le modèle. Chaque règle modèle ainsi invoquée va bien entendu, à son tour, constituer sa propre liste de noeuds courants et itérer à l'intérieur de celle-ci.

Dans l'un et l'autre cas, l'utilisation de chemins de localisation XPath peut amener à considérer provisoirement, dans les différentes étapes de localisation, des noeuds distincts du noeud courant, et éventuellement à les comparer à celui-ci. Par exemple, supposons que nous soyons dans la régle modèle qui traite chacun des noeuds livre d'une bibliothèque et que nous voulions, pour chaque livre, établir une liste des autres livres du même auteur. Nous écririons une instruction du type de :
<xsl:apply-templates select="//biblio/livre[auteur/nom = current()/auteur/nom]/titre[. != current()/titre]" mode="liste" />

Dans le chemin de localisation XPath utilisé par cette instruction, nous considérons tour à tour tout une kyrielle de noeuds sur lesquels nous pratiquons éventuellement des tests. Pour distinguer du noeud courant chacun de ces noeuds au moment où nous nous intéressons à lui, nous l'appelons le noeud contextuel. Dans l'exemple qui précède, la fonction current() nous permet de garder la mémoire du noeud courant et de comparer le titre et le nom de l'auteur du noeud contextuel avec le titre et le nom de l'auteur du noeud courant.