Donc, avant de voir une copie non conforme, voyons tout d'abord une copie conforme générique. Il évident, par ailleurs, qu'une copie conforme se fait très simplement avec `xsl:copy-of` ; mais pour évoluer vers une copie non conforme, il faut partir d'une copie conforme utilisant `xsl:copy` et non pas `xsl:copy-of`.

Ensuite nous verrons l'intérêt qu'il peut y avoir à obtenir un document *presque* identique à l'original, et comment réaliser cela.

Copie conforme générique

Commençons par considérer la règle suivante :

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

C'est une règle récursive, puisque elle s'applique à tout élément, et qu'elle sera donc sélectionnée pour traiter les nœuds rassemblés en une nouvelle liste par `<xsl:apply-templates/>`.

Une des premières choses dont il faut donc s'inquiéter, c'est de savoir si la récursion a des chances de s'arrêter, ou si elle est infinie. Clairement, la récursion s'arrête si la liste de nœuds constituée par `<xsl:apply-templates/>` est vide. Cela peut-il se produire ? Oui, puisque les nœuds en question sont les enfants directs du nœud courant. Si donc le nœud courant est une feuille de l'arbre XML, la récursion s'arrête.

La règle montrée ci dessus exprime donc la définition récursive suivante :

La recopie d'un élément, c'est :

```
la copie de cet élément
    à laquelle on accroche :
        - la recopie du premier enfant
        - la recopie du deuxième,
        - ...
        - la recopie du dernier enfant,
        - ou rien s'il n'y a pas d'enfant.
```

Ceci constitue donc l'idée de base pour réaliser une copie conforme et en profondeur d'un élément. Mais ce n'est pas suffisant, car l'élément peut avoir des attributs, qui dans la règle ci-dessus, sont ignorés. Comme en XML, un attribut n'est pas un enfant, les attributs ne sont donc pas pris en compte par cette règle.

Mais il suffit d'y penser, et de demander la recopie des attributs :

La recopie d'un élément, c'est :

```
la copie de cet élément
    à laquelle on accroche :
```

```
        - la recopie des attributs s'il y en a,
        - la recopie des enfants s'il y en a.
```

Mais du coup, si l'on fait ça, la règle est incomplète, car on y demande la recopie des attributs, alors qu'on définit la recopie d'élément, mais pas celle d'attribut. Là encore, c'est très simple de corriger :

la recopie d'un élément ou d'un attribut, c'est :

```
la copie de cet élément ou de cet attribut
    à laquelle on accroche :
        - la recopie des attributs s'il y en a,
        - la recopie des enfants s'il y en a.
```

Si cette règle s'applique à un attribut, elle dit que la recopie d'un attribut, c'est la copie de cet attribut, suivie de la recopie de ses attributs ou de ses enfants. Mais un attribut n'a ni attribut ni enfant ; la recopie d'un attribut se résume donc à la copie de cet attribut, ce qui ma foi semble assez satisfaisant pour l'esprit.

Traduite en XSLT, cette règle devient :

```
<xsl:template match="child::*|attribute::*">
    <xsl:copy>
        <xsl:apply-templates select="attribute::*"/>
        <xsl:apply-templates select="child::*"/>
    </xsl:copy>
</xsl:template>
```

Notons que `child::*` est la forme longue de `"*"`, et que `<xsl:apply-templates select="child::*"/>` sélectionne moins de types de nœuds que `<xsl:apply-templates/>`. En effet, ce sont tous les enfants du nœud courant qui sont sélectionnés par l'instruction `xsl:apply-templates`, pas seulement ceux qui sont des éléments, mais aussi les textes, les commentaires et les processing-instructions.

Mais la façon dont la règle est écrite, ci-dessus, renforce le parallèle avec la définition en langue naturelle telle qu'on l'a établie.

Note

Il faut remarquer ici que l'ordre relatif des deux instructions `xsl:apply-templates` est important, car il faut se rappeler ici qu'il est interdit d'accrocher un attribut à un nœud si on a déjà commencé à accrocher des enfants (voir *Règle XSLT typique*, page 291).

Est-on arrivé à la forme définitive de cette règle ? Pas tout à fait. Pour l'instant nous traitons les éléments et les attributs, mais il y a d'autres types de nœuds possibles : le type `text`, le type `namespace`, le type `comment`, le type `processing-instruction`, et le type `root`.

Nous avons vu (voir *Copie d'un nœud de type element*, page 326 et *Copie d'un nœud de type namespace*, page 338) qu'un nœud de type namespace est copié automatiquement par l'instruction `<xsl:copy>` appliquée à un élément, il n'est donc pas nécessaire de

s'occuper explicitement des domaines nominaux. De même (voir *Copie du nœud de type root*, page 339), la racine `root` de l'arbre XML du résultat est créée automatiquement, sans qu'il y ait besoin de s'en occuper. Restent les textes, les commentaires et les processing-instructions. Leur point commun, c'est qu'ils sont chacun nécessairement enfant d'un élément.

Si le nœud courant est un élément, l'instruction `<xsl:apply-templates select="child::*"/>` ne sélectionne que des éléments (voir *Le déterminant est une* *, page 55). Mais l'instruction `<xsl:apply-templates select="child::node()"/>` (ou plus simplement `<xsl:apply-templates"/>`), sélectionne tous les nœuds qui peuvent être des enfants du nœud courant : c'est exactement cela qu'il nous faut, puisque seront compris les commentaires, les textes, et les processing-instructions. Mais si on les sélectionne, encore faut-il que la règle les accepte en entrée ; il faut donc aussi modifier en conséquence l'attribut `match` :

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

Cette règle est maintenant correcte, mais on peut la simplifier légèrement :

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

Note

On peut toutefois s'interroger sur le bien-fondé de cette simplification. En effet on a maintenant l'union de deux node-sets, ce qui donne un nouveau node-set, dans lequel sont mélangés des attributs, des éléments, des textes, etc. Or un node-set n'est pas ordonné ; on peut donc craindre que les nœuds ne soient pas traités dans le bon ordre (i.e. les attributs avant les éléments). En fait, il n'y pas de problème ici, car `xsl:apply-templates` traite les nœuds du node-set renvoyé par le `select` dans l'ordre de lecture du document. Or l'ordre de lecture du document est parfaitement spécifié, (voir *Représentation graphique*, page 50), et stipule que pour un élément donné, les attributs viennent avant les enfants. Donc ici, nous retombons sur nos pieds, et nous sommes sûrs que les attributs (s'il y en a) seront traités avant les autres nœuds.

La dernière petite touche que l'on peut apporter à cette règle, c'est de la rendre plus sûre à utiliser dans des contextes différents :

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

Cette fois, on a une règle qui n'est activée que si on précise le mode « copie » ; elle peut donc coexister pacifiquement avec d'autres règles ayant un motif similaire, mais des transformations complètement différentes.

Mais il peut aussi arriver que l'on ait besoin que la copie soit conforme par défaut, sauf pour certains éléments que l'on équipe d'une règle spécifique. Dans ce cas, la copie ne doit pas être lancée autoritairement sur tel élément particulier, elle doit se lancer toute seule quand il n'y a aucune autre règle éligible.

Un exemple de cette idée se trouve mis en œuvre aux sections *Pattern n° 8 – Utilisation d'une structure de données auxilaire*, page 422 et *Pattern n° 18 – Localisation d'une application*, page 533.

Cela peut très simplement se faire ainsi :

```
<xsl:template match="child::node()|attribute::*" priority="-10">
    <xsl:copy>
        <xsl:apply-templates select="attribute::*" />
        <xsl:apply-templates select="child::node()" />
    </xsl:copy>
</xsl:template>
```

En affectant une priorité extrêmement faible à cette règle, on est sûr qu'elle ne sera sélectionnée que si le processeur XSLT n'a vraiment rien d'autre à se mettre sous la dent.

Terminons par une remarque concernant le degré de conformité d'une copie conforme : le point de vue est ici celui du parseur XML, non celui de l'œil humain. En particulier, il est impossible *a priori* de discerner deux documents XML qui ne diffèrent que de la façon dont sont écrits les attributs (avec des guillemets ou des apostrophes), ou les textes contenant des caractères interdits comme `"<"` ou `"&"` (avec des `<![CDATA[<]]>` ou avec des `<`).

Copie presque conforme

Nous partons ici d'un document XML contenant du XHTML généré par Docbook.

> **Note**
>
> Docbook est une DTD associée à des feuilles de style XSLT pour rédiger des articles, des livres, et plus généralement de la documentation. Les feuilles de style XSLT produisent du HTML ou du FO, lequel donne du PS ou du PDF avec des processeurs FO adéquats (voir *http://docbook.org*).

doc.xml

```
<?xml version="1.0" encoding="UTF-16"?>
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-16"/>
      <title>Génération de la documentation avec Docbook</title>
```

```
      <link rel="stylesheet" href="..css" type="text/css"/>
      <meta name="generator" content="DocBook XSL Stylesheets V0"/>
   </head>
   <body bgcolor="white" text="black" link="#0000FF" vlink="#840084" alink="#0000FF">
      <div class="article"> <div class="titlepage">
            <div> <h1 class="title">
               <a name="d0e1"></a>Génération de la documentation avec Docbook
               </h1>
            </div>
         </div>
         <p>Toutes ces modifications se trouvent dans le fichier principal
         <tt xmlns="http://www.w3.org/TR/xhtml1/transitional"
             class="filename">C:\DocBook\RunDocBook\extensions\verbatim.java</tt>,
             qui a finalement l'allure suivante:

        <table xmlns="http://www.w3.org/TR/xhtml1/transitional"
               border="1" bgcolor="#E0E0E0">
        <caption align="right" class="listingTitle">verbatim.java</caption>
        <tr><td>
            <pre class="programlisting">
// ici listing de programme Java</pre>
        </td></tr>
        </table>
        </p>
         <!-- ... suite du fichier sans importance ... -->
      </div>
   </body>
</html>
```

Le vrai problème que nous nous posons ici, c'est de modifier le listing qui se trouve entre les balises `<pre> ... </pre>` en remplaçant les tabulations initiales de chaque ligne de code par des séries de quatre espaces. Ceci parce que les navigateurs comptent en général huit espaces par tabulation, ce qui est beaucoup trop pour certains listings.

Traduit en terme de copie non conforme, cela veut dire que nous cherchons à faire une copie conforme du document, à ceci près que les éléments `<pre>` seront légèrement différents dans leur contenu.

Cependant, ce problème est curieusement difficile.

Non pas à cause de la copie en elle-même, qui repose sur ce qu'on vient de voir, ni à cause du remplacement de tabulations par des espaces, qui est simple à réaliser.

Ce problème est difficile parce que l'élément `<pre>` est un descendant de l'élément `<table>`, qui déclare un domaine nominal par défaut. Cette difficulté est donc complètement hors-sujet dans cette section, mais pour illustrer malgré tout cette notion de copie non conforme, nous allons supposer que nous voulons expurger la partie `<head>` pour ne garder que la balise `<title>`, tout le reste étant conservé intact.

copie.xsl

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version='1.0'>

    <xsl:output  method='xml' indent="yes" encoding='ISO-8859-1' />

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

    <xsl:template match="/html/head">
        <head>
            <xsl:apply-templates/>
        </head>
    </xsl:template>

    <xsl:template match="/html/head/*">
    </xsl:template>

    <xsl:template match="/html/head/title" priority="2">
        <xsl:apply-templates select="." mode="copie"/>
    </xsl:template>

    <xsl:template match="body">
        <xsl:apply-templates select="." mode="copie"/>
    </xsl:template>

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

</xsl:stylesheet>
```

La copie est lancée sur deux éléments : `<title>` et `<body>` ; le reste est reconstruit à la main, et éventuellement modifié. Ici la non conformité de la copie est due à la règle `<xsl:template  match="/html/head/*">` ; cette règle annule tous les éléments enfants de `<head>`, sauf `<title>`, qui bénéficie d'une règle à part. On remarquera d'ailleurs la priorité 2 imposée à cette règle à part, pour éviter l'ambiguïté avec la règle d'annulation.

Le résultat obtenu est bien celui qu'on attendait.

Pattern n° 11 – Détection d'un élément avec domaine nominal par défaut

Motivation

La motivation est très simple : on veut écrire une transformation XSLT (peu importe laquelle) qui nécessite d'écrire une certaine règle pour un certain élément. En principe, il n'y a rien de plus facile à faire, il suffit d'écrire par exemple :

```
<xsl:template match="truc">
        ...
</xsl:template>
```

Le problème qui peut survenir est que dans le document source XML, l'élément `<truc>` soit associé à un domaine nominal par défaut.

La règle ci-dessus n'est alors jamais activée, car le motif utilisé, `truc`, concorde avec tout élément `<truc>` sans domaine nominal, mais pas avec un élément possédant un domaine nominal, qu'il soit explicite ou par défaut.

Une solution de contournement peut éventuellement être mise en place, comme ceci :

```
<xsl:template match="*[ local-name() = 'truc' ]">
        ...
</xsl:template>
```

Cela fonctionne, car `"*"` ramasse tout ce qui se trouve sur l'axe `child::`, et ensuite on filtre pour ne garder que les éléments dont le nom est égal à la chaîne `'truc'` .

Mais ce n'est pas extraordinaire comme style, car cette règle sera sélectionnée sur tout élément, puis presque toujours rejetée à cause du prédicat. Par ailleurs, on ne fait que contourner le problème, alors qu'on pourrait le résoudre.

Solution

La solution passe évidemment par l'écriture d'un motif qui concorde avec un élément `<truc>`, dans un certain domaine nominal par défaut.

Une chose est certaine, c'est que :

```
<xsl:template match="truc">
```

concorde avec tout élément `<truc>` qui n'est dans *aucun* domaine nominal.

Si l'on veut écrire un motif qui concorde avec un élément `<truc>` dans un certain domaine nominal, il faut absolument écrire quelque chose du genre :

```
<xsl:template match="xx:truc">
```

où `xx` représente un préfixe (ou abréviation) d'un certain domaine nominal.

Mais c'est précisément là que la difficulté surgit : si on ajoute un préfixe, le domaine nominal n'est plus « par défaut », penserez vous. Certes, mais où ? Parlons-nous d'un

domaine nominal « par défaut » dans le programme XSLT, ou « par défaut » dans le document source XML ?

Un domaine nominal, cela reste un domaine nominal, qu'il soit par défaut ou non. C'est donc quelque chose du genre : `http://www.machinchose.fr/bidule`.

Si le document source se présente comme ceci :

```
<machin xmlns="http://www.machinchose.fr/bidule">
    <truc>
        ...
    </truc>
</machin>
```

l'élément `<truc>` a un domaine nominal par défaut.

Si le document source se présente comme cela :

```
<machin xmlns:xx="http://www.machinchose.fr/bidule">
    <xx:truc>
        ...
    </xx:truc>
</machin>
```

l'élément `<truc>` a un domaine nominal explicite.

Le point un peu subtil, mais ici essentiel pour obtenir la solution, c'est que dans les deux cas, *le domaine nominal peut très bien être le même, la seule différence résidant dans la façon de le dire*.

Donc une règle telle que :

```
<xsl:template match="xx:truc">
```

n'est pas une règle concordant avec

```
tout élément "truc" dont le préfixe est "xx"
```

mais avec

```
tout élément "truc" dont le domaine nominal est
http://www.machinchose.fr/bidule
```

en supposant que `xx` soit l'abréviation de ce domaine nominal.

La solution, pas très évidente, il faut bien le reconnaître, est donc de déclarer un domaine nominal explicite dans le programme XSLT, comme ceci :

```
<?xml version='1.0'?>
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"

    xmlns:xx="http://www.machinchose.fr/bidule"
```

```
    version='1.0'>

<!-- etc -->
```

puis d'écrire la règle en faisant mention du domaine nominal requis :

```
<xsl:template match="xx:truc">
        ...
</xsl:template>
```

Le document source XML est supposé être de la forme suivante :

```
<machin xmlns="http://www.machinchose.fr/bidule">
    <truc>
        ...
    </truc>
</machin>
```

Le motif de la règle ci-dessus concorde avec tout nœud `<truc>` dans le domaine nominal *http://www.machinchose.fr/bidule* ; or c'est précisément le cas de l'élément `<truc>` du fragment XML ci-dessus. Donc la règle s'appliquera bien à l'élément `<truc>` en question.

Note

Cette façon de résoudre le problème est tellement peu intuitivement évidente, qu'elle a fait l'objet d'une requête officielle d'amélioration pour la version XSLT 2.0. On pourra consulter, dans *http://www.w3.org/TR/xslt20req*, la section intitulée « 2.1 Must Allow Matching on Default Namespace Without Explicit Prefix ».

Exemple

Nous allons pouvoir revenir au problème de la copie non conforme dont il était question à la section *Copie presque conforme*, page 447, et arriver cette fois à une solution satisfaisante au problème posé. Rappelons qu'il s'agit de modifier légèrement le document XHTML produit par Docbook, en intervenant uniquement dans les balises `<pre class="programlisting">`, afin de supprimer les tabulations initiales de chaque ligne, en les remplaçant par des séries de quatre espaces.

Voici tout d'abord un extrait du document XHTML à transformer :

doc.html

```
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-16"/>
      <title>Génération de la documentation avec Docbook</title>
      <link rel="stylesheet" href="..css" type="text/css"/>
      <meta name="generator" content="DocBook XSL Stylesheets V0"/>
   </head>
   <body bgcolor="white" text="black" link="#0000FF" vlink="#840084"
         alink="#0000FF">
```

```
      <div class="article"> <div class="titlepage">
            <div> <h1 class="title">
               <a name="d0e1"></a>Génération de la documentation avec Docbook
               </h1>
            </div>
         </div>

         <div class="sect1"> <div class="titlepage">
               <div> <h2 class="title" style="clear: both">
                  <a name="d0e38"></a>1. Documentation avec docbook
                  </h2>
               </div>
            </div>

            <div class="sect2"> <div class="titlepage">
                  <div><h3 class="title">
                     <a name="d0e46">
                     </a>1.2. Description des principaux fichiers utiles
                     </h3>
                  </div>
               </div>

               <div class="sect3"> <div class="titlepage">
                     <div> <h4 class="title">
                        <a name="d0e231"></a>1.2.6. puratim.class
                        </h4>
                     </div>
                  </div>
         <p>Toutes ces modifications se trouvent dans le fichier principal
         <tt xmlns="http://www.w3.org/TR/xhtml1/transitional"
         class="filename">C:.java</tt>,
             qui a finalement l'allure suivante:

<table xmlns="http://www.w3.org/TR/xhtml1/transitional"
       border="1" bgcolor="#E0E0E0">
<caption align="right" class="listingTitle">puratim.java</caption>
     <tr>
        <td>
            <pre  class="programlisting">
import java.io.*;
public class puratim {
    public static void main( String arg[] ) {
        try {
            FileInputStream fis = new FileInputStream( arg[0] );
            InputStreamReader isr = new InputStreamReader( fis, "UTF-16" );
            BufferedReader in = new BufferedReader( isr );
            ...
            line = in.readLine(); // la 2° ligne n'est pas recopiée
                                  // (c'est la référence au graphe)

            for(;;){
```

```
                line = in.readLine();
                if ( line == null ) break;
                pw.println( line );
            }
            ...
        }
        ...
    }
}</pre>
</td>
</tr>
</table>
</p>

<p>A priori le service "Sécurité des réseaux" est au courant de ce problème,
qui devrait donc être résolu dans les jours qui viennent.
Lorsque ce sera le cas, on pourra supprimer ce fichier
<tt xmlns="http://www.w3.org/TR/xhtml1/transitional"
class="filename">puratim.class</tt>
</p>
               </div>
            </div>
         </div>
      </div>
   </body>
</html>
```

On voit que le problème de la détection d'un élément dans un domaine nominal par défaut va se poser ici, puisque l'élément `<pre ...>` fait partie de la descendance de `<table ...>`, qui déclare le domaine nominal par défaut `http://www.w3.org/TR/xhtml1/transitional`.

On va donc mettre en œuvre la solution indiquée plus haut ; cela consiste d'abord à déclarer un domaine nominal identique, identifié par une abréviation quelconque qui servira de préfixe. On écrira donc :

```
<?xml version='1.0'?>
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"

    xmlns:dns="http://www.w3.org/TR/xhtml1/transitional"

    version='1.0'>

<!-- etc -->
```

Ensuite, puisque l'on veut tout copier à l'identique, sauf précisément les éléments `<pre class="programlisting">`, il va falloir écrire une règle spéciale pour ces éléments :

```
<xsl:template match="dns:pre">
    <pre class="programlisting">
    <xsl:call-template name="replace_first_tabs_on_all_lines">
```

```
        <xsl:with-param name="codeSource" select="." />
    </xsl:call-template>
    </pre>
</xsl:template>
```

La règle ci-dessus semble correcte. Et pourtant ... Nous ne sommes pas au bout de nos peines avec les domaines nominaux. Telle qu'indiquée, la règle est correcte, du moins dans l'expression du motif : c'est tout l'objet de la discussion de la section précédente. Le corps de la règle, par contre, va poser à nouveau problème ; voici ce que l'on va obtenir lorsque le modèle de transformation de cette règle sera instancié :

```
<pre
    xmlns=""
    xmlns:dns="http://www.w3.org/TR/xhtml1/transitional"
    class="programlisting">
import java.io.*;
public class puratim {
    public static void main( String arg[] ) {
    ...
}</pre>
```

Dans le document résultat généré, l'élément `<pre ...>` est défini avec un domaine nominal par défaut nul (`xmlns=""`) ; cela signifie que cet élément ne fait partie d'aucun domaine nominal par défaut.

On a de plus une définition du domaine nominal `http://www.w3.org/TR/xhtml1/transitional`, déjà déclaré dans l'élément `<table ...>`, mais comme domaine nominal par défaut, alors qu'ici, il est associé au préfixe `dns`. Pourquoi pas ? Après tout, ce qui compte, c'est que l'élément soit associé au bon domaine nominal ; que ce soit par le truchement d'un domaine nominal par défaut ou explicite, peu importe.

Oui, mais malheureusement, l'élément `<pre>` n'a pas de préfixe. Et comme il ne fait désormais plus partie d'aucun domaine nominal par défaut, `<pre>` n'est donc associé à aucun domaine nominal. En cela, la copie que nous avons réalisée est trop infidèle, car nous ne voulions pas modifier quoi que ce soit aux domaines nominaux, mais seulement faire sauter les tabulations.

Cette fois, cependant, le problème n'est pas difficile à identifier et à corriger ; c'est la règle que nous avons écrite qui est incorrecte, car elle utilise l'élément littéral sans le préfixe `dns`. Or, dans le programme XSLT, *il n'y a pas de domaine nominal par défaut* ; ainsi, un élément littéral, comme `<pre>`, qui apparaît sans préfixe, est un élément sans domaine nominal. Le processeur XSLT a donc généré une déclaration d'annulation de domaine nominal (sous la forme `xmlns=""`) afin d'obéir à nos ordres.

Le plus dur est peut-être de réaliser qu'on a *effectivement* donné cet ordre.

Ayant vu cela, la correction est immédiate :

```
<xsl:template match="dns:pre">
    <dns:pre class="programlisting">
    <xsl:call-template name="replace_first_tabs_on_all_lines">
        <xsl:with-param name="codeSource" select="." />
```

```
    </xsl:call-template>
    </dns:pre>
</xsl:template>
```

Avec cette règle corrigée, on obtiendra ceci :

```
<dns:pre
    xmlns:dns="http://www.w3.org/TR/xhtml1/transitional"
    class="programlisting">
import java.io.*;
public class puratim {
    public static void main( String arg[] ) {
    ...
}</dns:pre>
```

Cette fois, c'est bon, le document résultat est identique au document d'origine du point de vue du traitement des domaines nominaux. La seule différence, c'est que dans le document d'origine, l'élément `<pre>` est associé à un domaine nominal par défaut, alors que dans le document résultat, il est associé à un domaine nominal explicite. Mais comme dans les deux cas, il s'agit du même domaine nominal, les deux documents sont équivalents du point de vue XML.

Le programme XSLT est celui-ci (les modèles nommés réalisant la suppression effective des tabulations ne sont pas montrés, car ils sont hors-sujet pour ce qui nous occupe actuellement) :

tabsToSpaces.xsl

```
<?xml version='1.0'?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:dns="http://www.w3.org/TR/xhtml1/transitional"
                version='1.0'>


    <xsl:output  method='html' encoding='ISO-8859-1' />

    <xsl:template name="replace_first_tabs_on_all_lines">
        ...
    </xsl:template>


    <xsl:template match="/">
        <xsl:apply-templates mode="copie"/>
    </xsl:template>

    <xsl:template match="dns:pre" mode="copie">
        <xsl:comment>Dans pre[@class='programlisting]'</xsl:comment>
        <dns:pre class="programlisting">
        <xsl:call-template name="replace_first_tabs_on_all_lines">
            <xsl:with-param name="codeSource" select="." />
        </xsl:call-template>
        </dns:pre>
```

```
    </xsl:template>

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

</xsl:stylesheet>
```

Et voici maintenant le résultat obtenu (la présentation a été remaniée pour les besoins de la mise en page) :

docSansTabs.html

```
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-16">
      <title>G&eacute;n&eacute;ration de la documentation avec Docbook</title>
      <link rel="stylesheet" href="..css" type="text/css">
      <meta name="generator" content="DocBook XSL Stylesheets V0">
   </head>
   <body bgcolor="white" text="black" link="#0000FF" vlink="#840084"
         alink="#0000FF">
      <div class="article">
         <div class="titlepage">
            <div>
               <h1 class="title">
               <a name="d0e1"></a>
               G&eacute;n&eacute;ration de la documentation avec Docbook
               </h1>
            </div>
         </div>

         <div class="sect1">
            <div class="titlepage">
               <div>
                  <h2 class="title" style="clear: both">
                  <a name="d0e38"></a>1. Documentation avec docbook
                  </h2>
               </div>
            </div>

            <div class="sect2">
               <div class="titlepage">
                  <div>
                     <h3 class="title">
                     <a name="d0e46"></a>
                     1.2. Description des principaux fichiers utiles
```

```
                    </h3>
                 </div>
              </div>

              <div class="sect3">
                 <div class="titlepage">
                    <div>
                       <h4 class="title">
                       <a name="d0e231"></a>1.2.6. puratim.class
                       </h4>
                    </div>
                 </div>
<p>Toutes ces modifications se trouvent dans le fichier principal
<tt xmlns="http://www.w3.org/TR/xhtml1/transitional"
    class="filename">C:.java</tt>,
    qui a finalement l'allure suivante:

 <table xmlns="http://www.w3.org/TR/xhtml1/transitional"
        border="1" bgcolor="#E0E0E0">
    <caption align="right" class="listingTitle">puratim.java</caption>
    <tr>
       <td>
<!--Dans pre[@class='programlisting]'-->
<dns:pre
        xmlns:dns="http://www.w3.org/TR/xhtml1/transitional"
        class="programlisting">
import java.io.*;
public class puratim {
 public static void main( String arg[] ) {
     try {
         FileInputStream fis = new FileInputStream( arg[0] );
         InputStreamReader isr = new InputStreamReader( fis, "UTF-16" );
         BufferedReader in = new BufferedReader( isr );
         ...
         line = in.readLine(); // la 2&deg; ligne n'est pas recopi&eacute;e
                               // (c'est la r&eacute;f&eacute;rence au graphe)
         for(;;){
             line = in.readLine();
             if ( line == null ) break;
             pw.println( line );
         }
         ...
     }
     ...
 }
}
</dns:pre>
       </td>
    </tr>
 </table>
</p>
```

```
<p>A priori le service "S&eacute;curit&eacute; des r&eacute;seaux"
est au courant de ce probl&egrave;me, qui devrait donc &ecirc;tre
r&eacute;solu dans les jours qui viennent. Lorsque ce sera le cas,
on pourra supprimer ce fichier
<tt xmlns="http://www.w3.org/TR/xhtml1/transitional"
    class="filename">puratim.class</tt>
</p>
              </div>
            </div>
          </div>
        </div>
    </body>
</html>
```

Si l'on compare maintenant le document original et celui obtenu, on constate des différences de présentation, et d'encodage des caractères non Ascii, mais aucune différence de structure XML. On a les même éléments, avec les mêmes attributs et les mêmes domaines nominaux. La suppression des tabulations n'est pas visible ici, mais ce n'est pas vraiment le sujet qui nous a donné du travail. On pourra toutefois trouver une discussion spécifique sur ce sujet à la section *Exemple*, page 402.

Finalement, on voit que ce n'est pas forcément si évident que ça de réaliser une copie presque conforme d'un document utilisant les domaines nominaux. Mais les solutions à mettre en œuvre, pour ces problèmes de domaines nominaux, sont assez génériques ; elles sont donc parfaitement réutilisables dans d'autres circonstances, sans rapport avec des copies presque conformes.

Pattern n° 12 – Références croisées inter fichiers

Motivation

Les problèmes de références croisées sont très fréquents en XSLT ; ce sont des problèmes où, pour traiter une information *A*, il faut en déduire une information *B*, puis rechercher cette information *B*, éventuellement dans un autre document, qui donne accès à une information *C*. On peut alors présenter côte à côte les informations *A* et *C*.

S'il y a beaucoup d'informations *A* à traiter, le point critique est la recherche de *B* dans tout un document, car cette recherche est multipliée par le nombre de *A* à traiter.

Une solution efficace consiste alors à indexer les informations *A* par les informations *B*, en utilisant le couple `instruction xsl:key/fonction key()`.

Nous allons montrer cela sur un exemple mettant en œuvre deux fichiers source XML.

Pour cela nous reprenons l'exemple que nous avions vu à la section *Exemple*, page 307. Nous avions un fichier `codesPlages.xml` contenant des informations sur des codes utilisés par ailleurs :

codesPlages.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<codes>

    <artistes>
        <artiste id="cplu" name="Christine Plubeau"/>
        <artiste id="nspt" name="Noëlle Spieth"/>
        <artiste id="eblq" name="Eric Bellocq"/>
        <artiste id="fmrt" name="Frédéric Martin"/>
        <artiste id="oded" name="Odile Edouard"/>
        <artiste id="fech" name="Freddy Eichelberger"/>
        <artiste id="dsmp" name="David Simpson"/>
    </artistes>

    <instruments>
        <instrument id="vdg" name="Viole de gambe"/>
        <instrument id="clv" name="Clavecin"/>
        <instrument id="thb" name="Théorbe"/>
        <instrument id="vl1" name="Violon baroque"/>
        <instrument id="vl2" name="Violon baroque"/>
        <instrument id="org" name="Orgue positif"/>
        <instrument id="vlc" name="Violoncelle baroque"/>
    </instruments>

</codes>
```

Par ailleurs, nous avions un fichier XML contenant des descriptions de plages de CD, mis en place dans ce même exemple, et modifié par la suite pour illustrer le principe de numérotation des nœuds (voir *Exemple*, page 350).

PlagesCD.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<plages>

   <plage No="1" vdg="cplu" clv="nspt"
                 thb="eblq" vl1="fmrt"
                 vl2="oded" org="fech"> Grave </plage>

   <plage No="2" thb="eblq"
                 clv="nspt" vl1="fmrt"
                 vl2="oded" vlc="dsmp"> Presto / Prestissimo </plage>

   <plage No="3" vdg="cplu" clv="nspt"
                 thb="eblq" vl1="fmrt"
                 vl2="oded"> Adagio </plage>

   <plage No="4" thb="eblq" vdg="cplu"
                 clv="nspt" vl1="fmrt"
```

```
                vlc="dsmp"
                org="fech"> Presto Récit de basse </plage>

</plages>
```

Ce fichier indique, pour chaque plage, la distribution sous forme d'attributs dont les noms sont des codes d'instruments, et les valeurs sont des codes d'artistes. Tous ces codes ont leur traduction en clair dans le fichier `codesPlages.xml`. Il s'agit maintenant d'écrire une transformation XSLT qui permette d'obtenir le fichier suivant (qui, mis en forme, pourrait figurer dans la pochette du disque, puisqu'il donne pour chaque artiste, les numéros de plages où il intervient, et l'instrument dont il joue) :

Résultat attendu

```
-------------------------------------
Christine Plubeau
1 3 4 Viole de gambe
-------------------------------------
Noëlle Spieth
1 2 3 4 Clavecin
-------------------------------------
Eric Bellocq
1 2 3 4 Théorbe
-------------------------------------
Frédéric Martin
1 2 3 4 Violon baroque
-------------------------------------
Odile Edouard
1 2 3 Violon baroque
-------------------------------------
Freddy Eichelberger
1 4 Orgue positif
-------------------------------------
David Simpson
2 4 Violoncelle baroque
```

Puisque l'on veut un fichier où les informations soient classées artiste par artiste, il semble logique de considérer que le fichier `codesPlages.xml` sera le fichier principal, et l'autre le fichier secondaire. Le cheminement peut alors se faire comme ceci :

1) Partant d'un `<artiste>`, on a son `id` (par exemple *cplu*) et son `nom` (*Christine Plubeau*).

2) Connaissant l'id *cplu* on va rechercher dans le fichier `PlagesCD.xml` toutes les `<plage>` qui ont un attribut dont la valeur est *cplu*, ce qui permet d'afficher les numéros de plages.

2) Puis ayant l'une de ces plages, il faut noter le nom de l'attribut dont la valeur correspond à la valeur cherchée (*cplu* dans l'exemple), ce qui nous donne le code de l'instrument joué (*vdg*, en l'occurrence).

2) Enfin, ayant ce code, on revient dans le document source principal, où l'on recherche un instrument dont le code correspond, et il n'y a plus qu'à afficher le nom de cet instrument.

Réalisation

La première chose à faire est de mettre en place une table d'association qui va donner des `<plage>`, connaissant des `id` d'artistes.

Or, dans le fichier `PlagesCD.xml`, les `id` d'artistes ne sont pas référencés par des attributs dont le nom est fixé ; il suffit qu'on ajoute un nouvel instrument, pour que cela donne un nouveau nom d'attribut, dont la valeur est un `id` d'artiste. La clé sera donc déclarée ainsi :

```
<xsl:key name="PlagesParCodesArtistes"
         match="plage" use="attribute::*" />
```

Comme l'attribut `use` est une expression renvoyant un node-set, chaque valeur textuelle de chaque nœud de ce node-set va jouer le rôle de valeur de clé ; une fois construite, la table va donc avoir l'allure représentée à la figure 9-1.

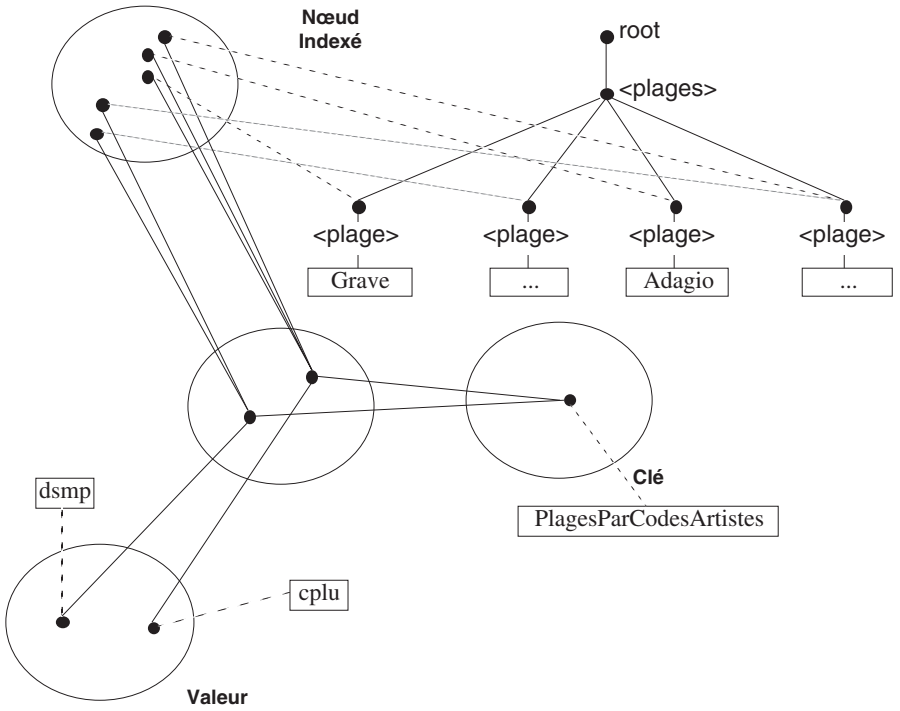


Figure 9-1

Plages par codes artistes.

De même, il va nous falloir une clé pour avoir un `<instrument>` connaissant son `id` :

```
<xsl:key name="InstrumentsParCodesInstruments"
         match="instrument" use="attribute::id" />
```

Note

On pourrait ici utiliser la fonction `id()`, qui renvoie l'élément dont l'identifiant est égal à une valeur donnée. Néanmoins, on ne le fait pas, car la manipulation d'identifiant est plus lourde : il faut déclarer l'attribut correspondant du type prédéfini ID, ce qui de fait, oblige à équiper le fichier XML d'une DTD. Une clé apporte ici beaucoup plus de souplesse, en nous affranchissant complètement de l'obligation d'une DTD.

Ceci étant, on peut commencer à coder la transformation. Le but étant de lister des artistes, on va donc partir du fichier `codesPlages.xml`, et traiter les éléments `<artiste>` :

```
<xsl:template match="artiste">
<xsl:template match="artiste">

    <xsl:call-template name="instancier-NomArtiste">
        <xsl:with-param name="nomArtiste" select="@nom"/>
    </xsl:call-template>

    <xsl:call-template name="instancier-NosPlagesAvecCetArtiste">
        <xsl:with-param name="codeArtiste" select="@id"/>
    </xsl:call-template>

    <xsl:call-template name="instancier-NomInstrumentDeCetArtiste">
        <xsl:with-param name="codeArtiste" select="@id"/>
    </xsl:call-template>

</xsl:template>

</xsl:template>
```

Il s'agit maintenant de déterminer les trois modèles nommés.

Instanciation du nom de l'artiste

Il n'y a pas de difficulté particulière ici :

```
<xsl:template name="instancier-NomArtiste">
    <xsl:param name="nomArtiste"/>
    ------------------------------------
    <xsl:value-of select="$nomArtiste"/>
    <xsl:call-template name="instancier-sautLigne"/>
</xsl:template>
```

Instanciation des Nos de plages

Il faut maintenant construire la table associée à la clé `PlagesParCodesArtistes`, construction qui doit se faire en explorant le fichier XML auxiliaire `PlagesCD.xml`. C'est la fonction `key()` qui réalise cette tâche, et on sait que l'arbre exploré est celui qui

contient le nœud contexte de l'évaluation de l'expression contenant l'appel. Pour explorer l'arbre XML du document contenu dans `PlagesCD.xml`, il faut donc placer le nœud contexte de cet appel quelque part dans cet arbre. La racine n'étant pas un plus mauvais endroit qu'un autre, on peut donc obtenir l'effet désiré en écrivant :

```
<xsl:template name="instancier-NosPlagesAvecCetArtiste">
    <xsl:param name="codeArtiste"/>

    <xsl:for-each select="document('PlagesCD.xml')">

        <!--
            ici le nœud contexte est la racine de l'arbre XML du document
            contenu dans PlagesCD.xml, car la fonction document() renvoie
            un node-set ne contenant que la racine du document demandé.
        -->

    </xsl:for-each>

</xsl:template>
```

En mettant l'appel à `key( "PlagesParCodesArtistes", $codeArtiste )` à la place du commentaire ci-dessus, on va donc construire une table d'après le bon fichier XML.

Mais cet appel est susceptible de renvoyer plusieurs nœuds, puisque un même artiste peut intervenir pour plusieurs plages du CD. Il faut donc explorer un par un les nœuds renvoyés, avec un `xsl:for-each` , dans lequel le nœud contexte est à nouveau changé, et passe tour à tour sur chaque `<plage>` renvoyée :

```
<xsl:template name="instancier-NosPlagesAvecCetArtiste">
    <xsl:param name="codeArtiste"/>

    <xsl:for-each select="document('PlagesCD.xml')">
        <xsl:for-each select='key( "PlagesParCodesArtistes",
                                   $codeArtiste )'>
            <xsl:value-of select="./attribute::No"/>
            <xsl:text> </xsl:text>
        </xsl:for-each>
    </xsl:for-each>

</xsl:template>
```

Instanciation des noms d'instruments

Si l'on connaît le code instrument, un appel à

```
key( "InstrumentsParCodesInstruments", $codeInstrument )
```

renvoie l'`<instrument>` cherché. Ici, l'arbre XML à explorer est celui du document source principal, donc en principe, le nœud contexte, quel qu'il soit, est dans cet arbre : inutile ici d'immerger l'appel à `key()` dans un `xsl:for-each` pour placer le nœud contexte dans le bon arbre XML. Ayant l'`<instrument>` cherché, il n'y a plus qu'à prendre la valeur de son attribut `nom` :

```
    <xsl:template name="instancier-NomInstrumentDeCetArtiste">
        <xsl:param name="codeArtiste"/>
        <xsl:variable name="codeInstrument">
            <xsl:call-template name="instancier-codeInstrument">
                <xsl:with-param name="idArtiste" select="$codeArtiste"/>
            </xsl:call-template>
        </xsl:variable>

        <xsl:value-of select='key( "InstrumentsParCodesInstruments",
                                   $codeInstrument )/attribute::nom' />

        <xsl:call-template name="instancier-sautLigne"/>
    </xsl:template>
```

Le problème est donc d'obtenir le code instrument connaissant le code artiste. Un appel à

```
key( "PlagesParCodesArtistes", $idArtiste )
```

renvoie les plages où l'artiste intervient, à condition bien sûr que cet appel soit placé dans un `xsl:for-each` pour que le nœud contexte indique le bon arbre XML, comme à la section précédente (voir *Instanciation des Nos de plages*, page 463). Ayant l'ensemble des plages pour un artiste donné, on considère l'une d'elle, peu importe laquelle. La première est le choix le plus simple, car si cet ensemble n'est pas vide, on est sûr que la première existe :

```
<xsl:variable name="unePlageAvecCetArtiste"
                          select='key( "PlagesParCodesArtistes",
                                       $idArtiste )[1]' />
```

Ayant une plage, par exemple :

```
<plage No="3" vdg="cplu" clv="nspt"
              thb="eblq" vl1="fmrt"
                         vl2="oded"> Adagio </plage>
```

et connaissant le code artiste, par exemple `"cplu"`, il faut en déduire le nom de l'attribut dont la valeur est `"cplu"` (ici `vdg`). Dans l'ensemble des attributs de la plage courante, il faut donc récupérer celui dont la valeur est celle du code artiste donné :

```
$unePlageAvecCetArtiste/attribute::*[ . = $idArtiste ]
```

Ayant l'attribut (qui est un nœud de type `attribute`), il n'y a plus qu'à appeler la fonction prédéfinie `local-name` qui va renvoyer son nom, correspondant au code instrument ; ce qui nous donne le modèle nommé d'instanciation de ce code instrument

```
    <xsl:template name="instancier-codeInstrument">
        <xsl:param name="idArtiste"/>
        <xsl:for-each select="document('PlagesCD.xml')">
            <xsl:variable name="unePlageAvecCetArtiste"
                          select='key( "PlagesParCodesArtistes",
                                       $idArtiste )[1]' />
```

```
            <xsl:value-of select="local-name(
                                $unePlageAvecCetArtiste/attribute::*
                                [ . = $idArtiste ] )" />

        </xsl:for-each>

    </xsl:template>
```

Programme complet

En réunissant tous les morceaux, on obtient finalement le programme suivant :

distribution.xsl

```
<?xml version="1.0" encoding="UTF-16"?>
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="1.0">

    <xsl:output  method='text' encoding='ISO-8859-1' />

    <xsl:key name="PlagesParCodesArtistes"
             match="plage" use="attribute::*" />

    <xsl:key name="InstrumentsParCodesInstruments"
             match="instrument" use="attribute::id" />

    <xsl:variable name="racinePlages" select="document('plagesCD.xml')" />


    <!-- ======================================== -->
    <xsl:template name="instancier-NomArtiste">
        <xsl:param name="nomArtiste"/>

        <xsl:text>-----------------------------------</xsl:text>
        <xsl:call-template name="instancier-sautLigne"/>

        <xsl:value-of select="$nomArtiste"/>
        <xsl:call-template name="instancier-sautLigne"/>
    </xsl:template>


    <!-- ======================================== -->
    <xsl:template name="instancier-NosPlagesAvecCetArtiste">
        <xsl:param name="codeArtiste"/>

        <xsl:for-each select="$racinePlages">
            <xsl:for-each select='key( "PlagesParCodesArtistes",
                                       $codeArtiste )'>
                <xsl:value-of select="@No"/>
                <xsl:text> </xsl:text>
            </xsl:for-each>
```

```
        </xsl:for-each>

    </xsl:template>


    <!-- ======================================== -->
    <xsl:template name="instancier-NomInstrumentDeCetArtiste">
        <xsl:param name="codeArtiste"/>
        <xsl:variable name="codeInstrument">
            <xsl:call-template name="instancier-codeInstrument">
                <xsl:with-param name="idArtiste" select="$codeArtiste"/>
            </xsl:call-template>
        </xsl:variable>

        <xsl:value-of select='key( "InstrumentsParCodesInstruments",
                                   $codeInstrument )/attribute::nom' />

        <xsl:call-template name="instancier-sautLigne"/>
    </xsl:template>


    <!-- ======================================== -->
    <xsl:template name="instancier-codeInstrument">
        <xsl:param name="idArtiste"/>
        <xsl:for-each select="$racinePlages">
            <xsl:variable name="unePlageAvecCetArtiste"
                          select='key( "PlagesParCodesArtistes",
                                       $idArtiste )[1]' />

            <xsl:value-of select="local-name(
                                  $unePlageAvecCetArtiste/attribute::*
                                  [ . = $idArtiste ] )" />

        </xsl:for-each>

    </xsl:template>

    <!-- ======================================== -->
    <xsl:template name="instancier-sautLigne">
    <xsl:text>
</xsl:text>
    </xsl:template>



    <!-- ======================================== -->
    <xsl:template match="artiste">

        <xsl:call-template name="instancier-NomArtiste">
            <xsl:with-param name="nomArtiste" select="@nom"/>
        </xsl:call-template>
```

```
        <xsl:call-template name="instancier-NosPlagesAvecCetArtiste">
            <xsl:with-param name="codeArtiste" select="@id"/>
        </xsl:call-template>

        <xsl:call-template name="instancier-NomInstrumentDeCetArtiste">
            <xsl:with-param name="codeArtiste" select="@id"/>
        </xsl:call-template>

    </xsl:template>


    <!-- ======================================== -->
    <xsl:template match="text()"/>


</xsl:stylesheet>
```

Pattern n° 13 – Génération d'hyper liens

Motivation

Dans la section précédente (voir *Pattern n° 12 – Références croisées inter fichiers*, page 459), il s'agissait de suivre des références en passant de fichier en fichier ; ici le problème est de générer des hyper-liens. Typiquement, en HTML, ces hyper liens seront des paires `<a name="...">` `<a href="...">`. Si HTML est cité ici en exemple, c'est uniquement parce que les balises `<a>` sont bien connues ; mais le problème est exactement le même pour n'importe quel texte où l'on doit émettre des références à d'autres parties du document, et se résout de la même façon.

On se restreint ici à un seul document, mais si l'on voulait émettre des références de type `<a href="...">` vers un autre document, il suffirait de combiner les idées mises en œuvre dans cette section, avec celles de la section précédente (*Pattern n° 12 – Références croisées inter fichiers*, page 459).

L'idée de base, ici, est d'utiliser la fonction `generate-id()` pour obtenir une chaîne de caractères qui pourra servir d'identifiant pour une ancre référençable.

Une autre idée est que nous sommes ici dans un domaine où la notion de clé d'indexation a des chances d'être utile.

Prenons par exemple ce fichier XML :

Saison.xml

```
<?xml version="1.0" encoding="UTF-16"?>
<Saison>

    <Période> Automne 1999 </Période>
```

```
    <Manifestations>
        <Concert>
            <Organisation> Anacréon </Organisation>
            <Date>Samedi 9 octobre 1999 <Heure> 20H30 </Heure> </Date>
            <Lieu>Chapelle des Ursules</Lieu>
        </Concert>
        <Théâtre>
            <Organisation> Masques et Lyres </Organisation>
            <Date>Mardi 19 novembre 1999 <Heure> 21H </Heure> </Date>
            <Lieu>Salle des Cordeliers</Lieu>
        </Théâtre>
        <Théâtre>
            <Organisation> Masques et Lyres </Organisation>
            <Date>Mercredi 20 novembre 1999 <Heure> 21H30 </Heure> </Date>
            <Lieu>Salle des Cordeliers</Lieu>
        </Théâtre>
    </Manifestations>

    <Adresse>
        <Lieu>Chapelle des Ursules</Lieu>
        9, rue des Ursules - 49000 Angers
    </Adresse>

    <Adresse>
        <Lieu>Salle des Cordeliers</Lieu>
        1, rue des Prévoyants de l'avenir - 49000 Angers
    </Adresse>

</Saison>
```

On veut obtenir une version HTLM de ce document, avec un lien actif de chaque lieu vers son adresse. Donc, arrivé sur un `<Lieu>` de concert ou de théâtre, il faut référencer le `<Lieu>` correspondant, enfant de `<Adresse>`. Pour cela, il faut chercher, parmi les enfants d'éléments `<Adresse>`, un `<Lieu>` dont la valeur textuelle soit la même que celle de l'élément `<Lieu>` courant. A ce `<Lieu>` ainsi trouvé, on associera un identifiant par la fonction `generate-id()`, et cet identifiant servira ensuite de référence pour les balises `<a>`.

Si les références à émettre sont peu nombreuses, et le document à traiter peu volumineux, cela peut rester acceptable de rechercher le `<Lieu>` parmi les `<Adresse>` à chaque fois qu'on a besoin d'émettre un `<a href="...">`. Mais dans le cas contraire, il est beaucoup plus efficace de construire une clé d'indexation des `<Lieu>`.

Réalisation avec recherche par clé

La clé est ici à construire en récoltant des `<Lieu>` enfants d'éléments `<Adresse>`, en prenant la valeur textuelle du `<Lieu>` comme valeur de clé :

```
<xsl:key name="lieux" match="Adresse//Lieu" use="." />
```

Les ancres à générer sont soit de la forme `<a href="...">`, soit de la forme `<a name="...">`. Dans les deux cas, l'attribut sera égal à `#abc`, ou `abc` est une chaîne de caractères caractéristique, fournie par la fonction `generate-id()`.

Donc, connaissant la valeur littérale d'un lieu, par exemple `Chapelle des Ursules`, on recherche le `<Lieu>` correspondant par la fonction `key()`. Le résultat est nécessairement un node-set à un et un seul élément : on transmet donc cet élément à `generate-id()`, et la valeur renvoyée va constituer la fin de la référence (le début étant le caractère `'#'`). Par exemple,

```
<a name="#{generate-id(key('lieux', 'Chapelle des Ursules'))}">
```

génère la balise HTML :

```
<a name="#d0e56">
```

La valeur `d0e56` n'est pas prédictible, bien sûr ; ici, c'est un exemple obtenu avec Saxon.

Remarquez la façon dont le descripteur de valeur différée d'attribut est employé : `"#{...}`. Après le guillemet, la valeur de l'attribut commence par un `'#'`. Ensuite vient le descripteur, introduit par le caractère `'{'`. Le descripteur est évalué, et sa valeur vient s'ajouter au `'#'` déjà pris en compte. On aurait éventuellement pu tout calculer dans le descripteur :

```
<a name="{concat('#', generate-id(key('lieux', 'Chapelle des Ursules')))}">
```

Un programme possible de traitement de ce fichier XML est donc le suivant :

Saison.xsl

```
<?xml version="1.0" encoding="UTF-16"?>
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="1.0">

    <xsl:output  method='html' encoding='ISO-8859-1' />

    <xsl:key name="lieux" match="Adresse//Lieu" use="." />


    <xsl:template match="/">
        <html>
            <head>
                <title>Programme Saison
                    <xsl:value-of
                    select="/Saison/Période"/>
                </title>
            </head>
            <body bgcolor="white" text="black">
                <xsl:apply-templates/>
            </body>
        </html>
    </xsl:template>
```

```
    <xsl:template match="Saison">
        <xsl:apply-templates select="Manifestations"/>
        <H3>Adresses :</H3>
        <xsl:apply-templates select="Adresse"/>
    </xsl:template>

    <xsl:template match="Concert|Théâtre">
        <H3><xsl:value-of select="local-name(.)"/> </H3>
        <p>Date : <xsl:value-of select="Date"/> <br/>
           Lieu : <a href="#{generate-id(key('lieux', ./Lieu))}">
                   <xsl:value-of select="Lieu"/>
                   </a>
        </p>
    </xsl:template>

    <xsl:template match="Adresse">
        <p><a name="#{generate-id(key('lieux', ./Lieu))}">
           <xsl:value-of select="Lieu"/>
           </a>
           <br/>
           <xsl:value-of select="./child::text()[2]"/>
        </p>
    </xsl:template>

    <xsl:template match="text()"/>


</xsl:stylesheet>
```

On notera la façon d'obtenir le texte associé à une adresse :

```
<xsl:value-of select="./child::text()[2]"/>
```

Le texte utile est le deuxième, car le premier n'est composé que d'espaces blancs : ce sont les espaces blancs situés entre la fin de la balise `<Adresse>` et le début de la balise `<Lieu>`.

Une alternative serait d'utiliser l'instruction `<xsl:strip-space>` qui supprime tous les nœuds `text` ne contenant que des espaces blancs. Cette solution est meilleure dans la mesure où elle évite d'avoir à gérer l'invisible. Nous la mettrons en œuvre dans la variante de la prochaine section, à titre de comparaison.

Le résultat obtenu avec Saxon est donné ci-dessous (les valeurs d'ancres dépendent du processeur utilisé) ; voir aussi la figure 9-2.

Saison.html

```
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">

      <title>Programme Saison  Automne 1999 </title>
```

```
    </head>
    <body bgcolor="white" text="black">
       <H3>Concert</H3>
       <p>Date : Samedi 9 octobre 1999  20H30  <br>
                Lieu : <a href="#d0e56">Chapelle des Ursules</a></p>
       <H3>Th&eacute;&acirc;tre</H3>
       <p>Date : Mardi 19 novembre 1999  21H  <br>
                Lieu : <a href="#d0e62">Salle des Cordeliers</a></p>
       <H3>Th&eacute;&acirc;tre</H3>
       <p>Date : Mercredi 20 novembre 1999  21H30  <br>
                Lieu : <a href="#d0e62">Salle des Cordeliers</a></p>
       <H3>Adresses :</H3>
       <p><a name="#d0e56">Chapelle des Ursules</a><br>
                 9, rue des Ursules - 49000 Angers

       </p>
       <p><a name="#d0e62">Salle des Cordeliers</a><br>
                 1, rue des Pr&eacute;voyants de l'avenir - 49000 Angers

       </p>
    </body>
</html>
```

Figure 9-2

Rendu HTML du fichier Saison.html

Réalisation avec recherche par expression XPath

Il n'y a pas de difficulté particulière à utiliser une expression XPath pour rechercher un lieu de même valeur textuelle que le lieu courant ; le seul petit problème est l'écriture du prédicat où l'on a besoin à la fois du nœud contexte et du nœud courant. Nous avons déjà rencontré ce problème et vu comment le résoudre : reportez-vous à la fin de la section *Exemple*, page 187.

Le programme est donné ci-dessous ; le résultat est exactement identique au précédent.

L'instruction `<xsl:strip-space elements="*" />` permet de supprimer de l'arbre XML les nœuds `text` ne contenant des espaces blancs ; en conséquence, l'instruction `<xsl:value-of select="./child::text()"/>` est maintenant beaucoup plus naturelle que dans la version précédente.

Saison.xsl

```
<?xml version="1.0" encoding="UTF-16"?>
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="1.0">

    <xsl:output  method='html' encoding='ISO-8859-1' />
    <xsl:strip-space elements="*" />

    <xsl:template match="/">
        <html>
            <head>
                <title>
                    Programme Saison
                    <xsl:value-of select="/Saison/Période"/>
                </title>
            </head>
            <body bgcolor="white" text="black">
                <xsl:apply-templates/>
            </body>
        </html>
    </xsl:template>

    <xsl:template match="Saison">
        <xsl:apply-templates select="Manifestations"/>
        <H3>Adresses :</H3>
        <xsl:apply-templates select="Adresse"/>
    </xsl:template>

    <xsl:template match="Concert|Théâtre">
        <H3><xsl:value-of select="local-name(.)"/> </H3>
        <p>Date : <xsl:value-of select="Date"/> <br/>
           Lieu : <a href="#{generate-id(/Saison/Adresse/Lieu
                                         [ . = current()/Lieu ]
                              )}">
```

```
                    <xsl:value-of select="Lieu"/>
                    </a>
        </p>
    </xsl:template>

    <xsl:template match="Adresse">
        <p><a name="#{generate-id(./Lieu)}">
           <xsl:value-of select="Lieu"/>
           </a><br/>
           <xsl:value-of select="./child::text()"/>
        </p>
    </xsl:template>

    <xsl:template match="text()"/>


</xsl:stylesheet>
```

Pattern n° 14 – Regroupements

Motivation

Le regroupement est un problème très fréquent dans les transformations XSLT. Il se pose à chaque fois que le document XML à traiter contient des informations disséminées ici et là, mais ayant toutes un point commun (par exemple, des villes, qui font toutes partie d'un pays ; des instrumentistes, qui jouent tous d'un certain instrument ; des pièces (de maisons), qui ont toutes une certaine surface, etc.). Il y a plusieurs sortes de regroupements, qui dépendent de ce qu'on veut obtenir, et du point de vue adopté pour décider du point commun à considérer.

Dans un regroupement par valeur, le problème consiste à produire un document résultat dans lequel par exemple les villes sont regroupées par pays, les instrumentistes par instrument, les pièces par surface, etc.

Dans un regroupement positionnel, le but est de rassembler des éléments dont les positions (au sein d'une certaine liste) sont identiques d'un certain point de vue. Par exemple, on a une liste de villes, et on veut les regrouper trois par trois.

Enfin, une dernière catégorie de regroupement est la restauration hiérarchique de documents XML. Il s'agit ici de reconstituer la structure hiérarchique d'un document XML qui a été « aplati » pour une raison quelconque. Un exemple de document plat a déjà été rencontré (voir le fichier `company.xml`, à la section *Exemple plus évolué*, page 275) ; la restauration hiérarchique d'un tel document consisterait à reconstituer un vrai document XML :

Fichier aplati

```
<?xml version="1.0" encoding="UTF-16" ?>
<table name="COMPANY">
```

```
NOACA<tab/>CHAR(6)<br/>
LSACA1<tab/>VARCHAR2(35)<br/>
CCPAA1<tab/>NUMBER(4)<br/>
LAACA1<tab/>VARCHAR2(35)<br/>
URL<tab/>VARCHAR2(40)<br/>
STATRAT<tab/>VARCHAR2(1)<br/>
</table>
```

Fichier restauré

```
<?xml version="1.0" encoding="UTF-16" ?>
<table name="COMPANY">
    <field>
        <name>NOACA</name>
        <type>CHAR(6)</type>
    </field>
    <field>
        <name>LSACA1</name>
        <type>VARCHAR2(35)</type>
    </field>
    etc.
</table>
```

Remarque

La future version 2.0 de XSLT devrait inclure des possibilités pour spécifier des regroupements, parce que tout le monde s'accorde à reconnaître que les regroupements ne sont pas simples à programmer, alors qu'ils font partie du quotidien.

Regroupements par valeur ou par position

D'une façon générale, on peut dire qu'on a identifié un problème de regroupement par valeur quand les deux conditions suivantes sont réunies :

1) L'arbre XML contient certains nœuds *N* d'un certain type pour lesquels on dispose (au moins conceptuellement) d'une certaine fonction *valeur*, telle que `valeur(N)` renvoie un résultat qu'on peut appeler la *valeur de N*. Suivant ce point de vue, la valeur d'un ville, c'est son pays ; la valeur d'un instrumentiste, c'est son instrument ; et la valeur d'une pièce, c'est sa surface.

2) Le document résultat doit contenir les nœuds *N* regroupés par valeurs identiques.

Un regroupement par position est en fait un cas particulier du précédent, où la valeur d'un élément est liée à sa position au sein d'une certaine liste. La position étant donnée par un nombre entier, la valeur d'un élément, dans ce cas, est donc numérique. Par exemple, si l'on à une liste de villes à regrouper trois par trois, on prendra pour valeur d'une ville le tiers de sa position dans cette liste, de telle sorte que chaque groupe de trois soit constitué de villes de même valeur.

Dans son principe, la solution consiste à procéder en deux étapes :

- **Etape 1 :** on commence par constituer l'ensemble *NVD* des nœuds de type *T* de valeurs toutes différentes.
- **Etape 2 :** on parcourt **(2a)** l'ensemble *NVD*, et pour chaque nœud (ayant une certaine valeur *v*) de cet ensemble, on sélectionne **(2b)** l'ensemble des nœuds de type *T* de l'arbre XML qui ont *v* pour valeur : ces nœuds sont regroupés par valeur.

L'étape 1 est donc une étape de constitution d'un ensemble de valeurs toutes différentes ; or nous avons déjà vu comment résoudre ce problème (voir *Node-set de valeurs toutes différentes*, page 438). La méthode de Steve Muench est particulièrement recommandée ici, car elle est basée sur la construction d'une clé, qui va aussi servir dans l'étape 2, en rendant sa mise en œuvre particulièrement simple.

Des pièces regroupées par surface habitable (regroupement par valeur d'attributs)

Nous allons continuer le même exemple qu'à la section *Node-set de valeurs toutes différentes*, page 438, afin de pouvoir reprendre la réalisation obtenue par la méthode de la clé.

Le fichier XML à traiter est le suivant :

maisons.xml

```
<?xml version="1.0" encoding="UTF-16" standalone="yes"?>
<maisons>
    <maison id="1">
        <RDC>
            <cuisine surface='40m2'>
                Evier inox. Mobilier encastré.
            </cuisine>
            <WC>
                Lavabo. Cumulus 200L.
            </WC>
            <séjour surface='40m2'>
                Cheminée en pierre. Poutres au plafond.
                Carrelage terre cuite. Grande baie vitrée.
            </séjour>
            <bureau surface='15m2'>
                Bibliothèque encastrée.
            </bureau>
            <garage/>
        </RDC>
        <étage>
            <terrasse>Palmier en zinc figurant le désert.</terrasse>
            <chambre surface='28m2' fenêtre='3'>
                Carrelage terre cuite poncée.
                <alcôve surface='8m2' fenêtre='1'>
                    Lambris.
                </alcôve>
```

```
            </chambre>
            <chambre surface='15m2'>
                Lambris.
            </chambre>
            <salleDeBains surface='15m2'>
                Douche, baignoire, lavabo.
            </salleDeBains>
        </étage>
    </maison>
    <maison id="2">
        <RDC>
            <cuisine surface='28m2'>
                en ruine.
            </cuisine>
            <garage/>
        </RDC>
        <étage>
            <terrasse>
                vue sur la mer
            </terrasse>
            <salleDeBains surface='15m2'>
                Douche.
            </salleDeBains>
        </étage>
    </maison>
    <maison id="3">
        <RDC>
            <séjour surface='40m2'>
                paillasson à l'entrée
            </séjour>
        </RDC>
        <étage>
            <chambre surface='28m2'>
                porte cochère.
            </chambre>
        </étage>
    </maison>
</maisons>
```

Le fichier que l'on souhaite obtenir est celui-ci :

pieces.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<piecesParSurfaces>

   <pieces surface="40m2">
      <cuisine idMaison="1">Evier inox. Mobilier encastré.</cuisine>
      <séjour idMaison="1">Cheminée en pierre. Poutres au plafond.
                           Carrelage terre cuite. Grande baie vitrée.</séjour>
      <séjour idMaison="3">paillasson à l'entrée</séjour>
   </pieces>
```

```
   <pieces surface="15m2">
      <bureau idMaison="1">Bibliothèque encastrée.</bureau>
      <chambre idMaison="1">Lambris.</chambre>
      <salleDeBains idMaison="1">Douche, baignoire, lavabo.</salleDeBains>
      <salleDeBains idMaison="2">Douche.</salleDeBains>
   </pieces>

   <pieces surface="28m2">
      <chambre idMaison="1">Carrelage terre cuite poncée.</chambre>
      <cuisine idMaison="2">en ruine.</cuisine>
      <chambre idMaison="3">porte cochère.</chambre>
   </pieces>

   <pieces surface="8m2">
      <alcôve idMaison="1">Lambris.</alcôve>
   </pieces>

</piecesParSurfaces>
```

La réalisation de l'étape 1 a déjà été faite, nous avions obtenu la clé :

```
<xsl:key name="groupesdeSurfacesParValeurs"
         match="attribute::surface"
         use="." />
```

et nous avions obtenu l'expression qui donne l'ensemble des valeurs toutes différentes :

Création d'un node-set de surfaces toutes différentes

```
//attribute::surface[
      generate-id() =
      generate-id(
         key('groupesdeSurfacesParValeurs', .)[1]
      )
  ]
```

La réalisation de l'étape 2 est très simple : il suffit de parcourir l'ensemble obtenu, ce qui va nous donner des valeurs toutes différentes, et ensuite, pour chaque valeur, d'exploiter la clé pour obtenir les nœuds possédant cette valeur.

maisons.xsl

```
<?xml version="1.0" encoding="UTF-16"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

    <xsl:output  method='xml' encoding='ISO-8859-1' indent="yes" />

    <xsl:key name="groupesdeSurfacesParValeurs"
             match="attribute::surface"
             use="." />


    <!--
```

```
Etape 1 : constitution d'un ensemble de valeurs toutes différentes
================================================================== -->

 <xsl:variable name="lesDifférentesSurfaces"
              select="//attribute::surface[
                          generate-id() =
                          generate-id(
                              key('groupesdeSurfacesParValeurs', .)[1]
                          )
                        ]" />
<!--
================================================================== -->




<xsl:template match="/">
    <piecesParSurfaces>
        <!--
        Etape 2A : parcours des valeurs de cet ensemble
        ============================================== -->
        <xsl:for-each select="$lesDifférentesSurfaces" > <!--
        ============================================== -->

          <!--
            ici le nœud courant est un nœud attribute::surface
            -->

            <xsl:variable name="valeurCourante" select="." />

            <pieces surface="{$valeurCourante}">
                <!--
                Etape 2B : parcours des nœuds ayant cette valeur
                ================================================== -->
                <xsl:for-each select="key('groupesdeSurfacesParValeurs',
                                          $valeurCourante)" >     <!--
                ================================================== -->

                  <!--
                    ici le nœud courant est un nœud attribute::surface
                    -->

                    <xsl:variable name="pieceCourante"
                                  select="./parent::node()" />

                    <xsl:element name="{local-name($pieceCourante)}">
                        <xsl:attribute name="idMaison">
                            <xsl:value-of select="
                            $pieceCourante/ancestor::maison/@id"/>
                        </xsl:attribute>
                        <xsl:value-of select="
                        normalize-space($pieceCourante/child::text())"/>
                    </xsl:element>
```

```
                    </xsl:for-each>
                </pieces>
            </xsl:for-each>
        </piecesParSurfaces>
    </xsl:template>


</xsl:stylesheet>
```

Le résultat obtenu est exactement celui montré ci-dessus (à part quelques sauts de lignes ajoutés pour la lisibilité).

Les villes regroupées par pays (regroupement par valeur d'attributs)

On suppose qu'on a un fichier XML donnant une liste de villes avec, pour chacune, son pays :

villes.xml

```
<?xml version="1.0" encoding="UTF-16" standalone="yes"?>
<Villes>
    <Ville nom="Paris" pays="France" />
    <Ville nom="Madrid" pays="Espagne" />
    <Ville nom="Milan" pays="Italie" />
    <Ville nom="Rome" pays="Italie" />
    <Ville nom="Angers" pays="France" />
    <Ville nom="Barcelone" pays="Espagne" />
    <Ville nom="Venise" pays="Italie" />
    <Ville nom="Cordoue" pays="Espagne" />
    <Ville nom="Naples" pays="Italie" />
</Villes>
```

On veut la liste des villes par pays. La méthode est exactement la même, bien que le fichier XML à traiter soit d'un structure assez différente :

villes.xsl

```
<?xml version="1.0" encoding="UTF-16"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

    <xsl:output  method='text' encoding='ISO-8859-1' />

    <xsl:key name="groupesDePaysParNoms"
             match="attribute::pays"
             use="." />

    <!--
    Etape 1 : constitution d'un ensemble de valeurs toutes différentes
    ================================================================== -->
    <xsl:variable name="lesDifférentsPays"
                  select="//attribute::pays[
                              generate-id() =
```

```
                              generate-id(
                                  key('groupesDePaysParNoms', .)[1]
                              )
                            ]" />
    <!--
    ================================================================ -->

    <xsl:template match="/">
        <!--
        Etape 2A : parcours des valeurs de cet ensemble
        ============================================== -->
        <xsl:for-each select="$lesDifférentsPays"> <!--
        ============================================== -->
          <!--
            ici le nœud courant est un nœud attribute::pays
            -->
        - <xsl:value-of select="."/> :
            <xsl:variable name="tousLesAttributsPaysDeMêmeValeur"
                          select="key('groupesDePaysParNoms', .)" />
            <!--
            Etape 2B : parcours des nœuds ayant cette valeur
            ======================================================== -->
            <xsl:for-each select="$tousLesAttributsPaysDeMêmeValeur"> <!--
            ======================================================== -->
              <!--
                ici le nœud courant est un nœud attribute::pays
                -->
                <xsl:variable name="laVilleCorrespondante"
                              select="./parent::node()" />
                . <xsl:value-of select="$laVilleCorrespondante/@nom"/>

            </xsl:for-each>
        </xsl:for-each>

    </xsl:template>

</xsl:stylesheet>
```

Résultat

```
        - France :

                . Paris
                . Angers

        - Espagne :

                . Madrid
                . Barcelone
                . Cordoue
```

```
        - Italie :

                . Milan
                . Rome
                . Venise
                . Naples
```

Les instrumentistes regroupés par instruments (regroupement par valeur de nœuds *texte*)

Dans les deux exemples précédents, les valeurs servant de valeur de clé étaient des valeurs d'attributs. Ce n'est nullement une obligation : cet exemple va le montrer.

On part d'un fichier XML comme ceci :

Concert.xml

```
<?xml version="1.0" encoding="UTF-16" standalone="yes"?>

<Concert>

    <Entête> Les Concerts d'Anacréon </Entête>
    <Date>Mardi 11 Février 2003, 20H30</Date>
    <Lieu>Chapelle des Ursules</Lieu>

    <Ensemble> Hespèrion XXI </Ensemble>

    <Interprète>
        <Nom> Jordi Savall </Nom>
        <Instrument>Dessus de viole</Instrument>
    </Interprète>

    <Interprète>
        <Nom> Wieland Kuijken </Nom>
        <Instrument>Hautecontre de viole</Instrument>
    </Interprète>

    <Interprète>
        <Nom> Sophie Watillon </Nom>
        <Instrument>Hautecontre de viole</Instrument>
    </Interprète>

    <Interprète>
        <Nom> Sergi Casademunt </Nom>
        <Instrument>Ténor de viole</Instrument>
    </Interprète>

    <Interprète>
        <Nom> Sylvia Abramowicz </Nom>
        <Instrument>Basse de viole</Instrument>
    </Interprète>
```

```
    <Interprète>
        <Nom> Marianne Muller </Nom>
        <Instrument>Basse de viole</Instrument>
    </Interprète>

    <Interprète>
        <Nom> Philippe Pierlot </Nom>
        <Instrument>Basse de viole</Instrument>
    </Interprète>

    <TitreConcert>
        Fantasias for the Viols (1680)
    </TitreConcert>

    <Compositeur>Henry Purcell</Compositeur>

</Concert>
```

Et on veut obtenir la distribution du concert, classée par instruments. Ici, les « valeurs » d'instruments sont leurs noms, qui ne sont pas des attributs, mais des nœuds texte.

distribution.xsl

```
<?xml version="1.0" encoding="UTF-16"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

    <xsl:output  method='text' encoding='ISO-8859-1' />

    <xsl:key name="groupesInterprètesParInstruments"
             match="Instrument"
             use="." />


    <!--
    Etape 1 : constitution d'un ensemble de valeurs toutes différentes
    ========================================================================= -->

     <xsl:variable name="lesDifférentsInstruments"
                  select="//Instrument[
                              generate-id() =
                              generate-id(
                                  key('groupesInterprètesParInstruments', .)[1]
                              )
                            ]" />
    <!--
    ========================================================================= -->




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

```
        <!--
        Etape 2A : parcours des valeurs de cet ensemble
        ================================================ -->
        <xsl:for-each select="$lesDifférentsInstruments" > <!--
        ================================================ -->

          <!--
            ici le nœud courant est un nœud <Instrument>
            -->

            <xsl:variable name="valeurCourante" select="." />

            - <xsl:value-of select="$valeurCourante"/>

            <!--
            Etape 2B : parcours des <Instrument> ayant cette valeur
            ================================================== -->
            <xsl:for-each select="key('groupesInterprètesParInstruments',
                                      $valeurCourante)" >     <!--
            ================================================== -->

              <!--
                ici le nœud courant est un nœud <Instrument>
                -->

                <xsl:variable name="instrumentiste"
                              select="./parent::Interprète" />
                . <xsl:value-of select="$instrumentiste/Nom"/>
            </xsl:for-each>
        </xsl:for-each>
    </xsl:template>


</xsl:stylesheet>
```

Résultat

```
            - Dessus de viole
                .  Jordi Savall

            - Hautecontre de viole
                .  Wieland Kuijken
                .  Sophie Watillon

            - Ténor de viole
                .  Sergi Casademunt

            - Basse de viole
                .  Sylvia Abramowicz
                .  Marianne Muller
                .  Philippe Pierlot
```

Les villes trois par trois (regroupement positionnel)

Le problème est très simple : on a une liste de villes, et on veut les afficher sous la forme d'un tableau de trois colonnes par ligne. Nous reprenons donc le même fichier que celui qui a servi pour regrouper les villes par pays :

villes.xml

```
<?xml version="1.0" encoding="UTF-16" standalone="yes"?>
<Villes>
    <Ville nom="Paris" pays="France" />
    <Ville nom="Madrid" pays="Espagne" />
    <Ville nom="Milan" pays="Italie" />
    <Ville nom="Rome" pays="Italie" />
    <Ville nom="Angers" pays="France" />
    <Ville nom="Barcelone" pays="Espagne" />
    <Ville nom="Venise" pays="Italie" />
    <Ville nom="Cordoue" pays="Espagne" />
    <Ville nom="Naples" pays="Italie" />
</Villes>
```

Le fichier à obtenir est le suivant :

villes.html

```
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">

      <title> Les villes </title>
   </head>
   <body text="black" bgcolor="white">
      <table>
         <tr>
            <td>Paris</td>
            <td>Madrid</td>
            <td>Milan</td>
         </tr>
         <tr>
            <td>Rome</td>
            <td>Angers</td>
            <td>Barcelone</td>
         </tr>
         <tr>
            <td>Venise</td>
            <td>Cordoue</td>
            <td>Naples</td>
         </tr>
      </table>
   </body>
</html>
```

Le principe n'est pas différent, malgré le fait que le regroupement soit ici positionnel : il suffit de se ramener à un point de vue dans lequel la valeur d'une ville est maintenant non plus son pays, mais sa position au sein de la fratrie des villes.

La première étape sera donc comme d'habitude de constituer une clé de regroupement par valeur.

```
<xsl:key name="lesVillesParPosition"
             match="Ville"
             use="??" />
```

Néanmoins, le problème est ici de définir la position de la ville courante, lors de l'évaluation de l'expression XPath fournie par l'attribut `use`.

Remarque importante

Il faut ici se souvenir que cette expression XPath est évaluée avec le nœud en concordance comme nœud contexte, et une liste réduite au seul nœud contexte comme liste contexte. Si bien que la fonction prédéfinie `position()` ne peut que renvoyer la valeur 1, si on l'appelle quelque part au sein de cette expression. Il faut donc définitivement abandonner l'idée d'utiliser cette fonction dans une expression pour le calcul de la valeur d'une clé, si on l'avait jamais eue : une fonction qui renvoie toujours 1 n'est pas vraiment des plus utiles.

Ici, la liste intéressante n'est donc pas la liste contexte formée lors de l'évaluation de l'expression attribut de `use`, mais le node-set des `<Ville>`, énuméré dans l'ordre de lecture du document, qui donne la numérotation (en partant arbitrairement de 0) 0 (Paris), 1 (Madrid), ..., 8 (Naples).

Pour une `<Ville>` donnée, son numéro est donc égal au nombre d'éléments contenus dans le node-set des `<Ville>` qui sont des `preceding-sibling` de la `<Ville>` courante. Si donc l'on évalue l'expression

```
preceding-sibling::Ville
```

par rapport à un nœud contexte qui est une `<Ville>` *V* quelconque, on obtient le node-set de toutes les villes situées avant *V* dans l'ordre de lecture du document.

L'expression qui donne la position d'une telle `<Ville>` *V* est donc :

```
count( preceding-sibling::Ville )
```

si toutefois l'évaluation a bien lieu par rapport au nœud contexte *V*.

Il est donc possible, maintenant, de déterminer l'expression qui va fournir la valeur de regroupement des villes :

```
<xsl:key name="lesVillesParPosition"
             match="Ville"
             use="floor( count(preceding-sibling::Ville)  div 3 )" />
```

L'opérateur `div` est la division ; les calculs se faisant en nombres réels double précision, il est nécessaire de prendre la partie entière du quotient obtenu, d'où l'appel à la fonction `floor()`.

Avec cette définition de clé :

- les villes Paris, Madrid, et Milan ont pour valeur 0 ;
- les villes Rome, Angers, et Barcelone ont pour valeur 1 ;
- les villes Venise, Cordoue, et Naples ont pour valeur 2.

L'étape 1 consiste comme d'habitude à constituer l'ensemble des éléments de valeurs toutes différentes (ici des villes), mais c'est plus simple à réaliser que dans les exemples précédents, car les villes qui vont constituer cet ensemble ont un numéro que l'on peut calculer à l'avance : ce sont les villes numérotées 0 (Paris), 3 (Rome), et 6 (Venise), qui ont respectivement pour valeur 0, 1, et 2. L'initialisation de la variable contenant le node-set des villes de valeurs toutes différentes va donc pouvoir s'écrire :

```
<xsl:variable name="lesDifférentsGroupesDeVilles"
              select="//Ville[
                         ((position() - 1)  mod 3) = 0
                        ]" />
```

Ici, la fonction `position()` peut être utilisée, car elle est appelée dans un prédicat qui filtre un node-set constitué de toutes les villes : lors de l'évaluation du prédicat, la liste contexte est constituée des éléments du node-set à filtrer, donc la fonction `position()` renvoie la position de la ville courante au sein de cette liste, ce qui donne le résultat attendu.

On peut donc maintenant écrire le programme, dont le plan reprend très exactement celui des autres exemples de regroupements que nous avons déjà vus.

villes.xsl

```
<?xml version="1.0" encoding="UTF-16"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

    <xsl:output  method='html' encoding='ISO-8859-1' />

    <xsl:variable name="facteurDeRegroupement" select="3"/>

    <xsl:key name="lesVillesParPosition"
             match="Ville"
             use="floor( count(preceding-sibling::Ville)  div 3 )" />

    <!--
    Etape 1 : constitution d'un ensemble de valeurs toutes différentes
    ================================================================== -->
    <xsl:variable name="lesDifférentsGroupesDeVilles"
                  select="//Ville[
                             ((position() - 1)  mod $facteurDeRegroupement) = 0
                            ]" />
    <!--
    ================================================================== -->
```

```
    <xsl:template match="Villes">

        <table>
        <!--
        Etape 2A : parcours des valeurs de cet ensemble
        ================================================ -->
        <xsl:for-each select="$lesDifférentsGroupesDeVilles"> <!--
        ================================================ -->
          <!--
            ici le noeud courant est un noeud Ville
            (la première de chaque ligne du tableau)
            Sa valeur de clé est donnée par position() - 1
            -->

            <xsl:variable name="valClé" select="position() - 1" />

            <tr>
            <!--
            Etape 2B : parcours des noeuds ayant cette valeur
            ========================================================= -->
            <xsl:for-each select="key('lesVillesParPosition', $valClé)"> <!--
            ========================================================= -->
              <!--
                ici le noeud courant est une ville
                -->
                <td><xsl:value-of select="@nom"/></td>

            </xsl:for-each>
            </tr>
        </xsl:for-each>
        </table>

    </xsl:template>

    <!--   -->
    <xsl:template match="/">
        <html>
            <head>
            <title> Les villes </title>
            </head>
            <body text="black" bgcolor="white">
            <xsl:apply-templates/>
            </body>
        </html>
    </xsl:template>

</xsl:stylesheet>
```

Le résultat obtenu est celui montré plus haut (fichier `Villes.html`) ; on observera le désagrément que procure l'interdiction d'utiliser une référence de variable dans l'attribut `use` de l'instruction `xsl:key`.

Regroupements hiérarchiques

Les regroupements hiérarchiques consistent à reconstituer explicitement une hiérarchie d'éléments, absente du document à traiter, mais néanmoins sous-jacente. Au lieu d'être véritablement arborescent, avec des éléments imbriqués, le document est en effet généralement réduit à une structure linéaire d'éléments juxtaposés. C'est ce qui explique qu'un regroupement hiérarchique ne soit pas sans rapport avec un regroupement positionnel, dans la mesure où la reconstitution du niveau hiérarchique d'un élément est en partie basée sur sa position dans la structure linéaire.

La valeur d'un élément sera donc ici la place hiérarchique que l'élément devra avoir dans le document résultat, ce qui est équivalent à dire que la valeur d'un élément, c'est le nœud qu'il devrait avoir pour parent dans le document source.

D'une façon générale, on peut affirmer qu'on a identifié un problème de regroupement hiérarchique quand les deux conditions suivantes sont réunies :

1) L'arbre XML contient certains nœuds *N* d'un certain type pour lesquels on dispose (au moins conceptuellement) d'une certaine fonction *valeur*, telle que `valeur(N)` renvoie l'identité du nœud qui devrait être le parent de N, si la hiérarchie était correctement respectée.

2) Le document résultat doit contenir les nœuds *N*, réorganisés (ou regroupés) de telle sorte que chaque nœud *N* ait pour parent un nœud dont l'identité est égale à `valeur(N)`.

Dans son principe, la solution consiste à procéder en deux étapes :

- **Etape 1** : on commence par une reconstitution des relations hiérarchiques. Chaque nœud `N` à traiter est associé par une clé à sa valeur hiérarchique (l'identité du nœud qui devrait être le parent de `N`). A la fin de cette opération, il est donc possible d'obtenir le node-set des enfants directs de chaque nœud `P` du document : en effet, connaissant `P`, on connaît son identité *I* (`generate-id()`), et la clé donne alors tous les éléments ayant `I` pour valeur, c'est-à-dire tous les éléments ayant `P` pour parent.
- **Etape 2** : on parcourt la hiérarchie, en partant du sommet. Connaissant le sommet, on en déduit les enfants qu'il devrait avoir, grâce à la clé obtenue à l'étape précédente, et on les construit dans le document résultat. Il n'y a plus qu'à reprendre ce processus, non plus avec le sommet, mais avec chacun des enfants, et ainsi de suite récursivement avec tout le reste de la hiérarchie.

Exemple générique

Cet exemple est un exemple simple et générique, qui revient assez souvent sous diverses formes sur la mailing-list XSLT (*www.biglist.com/lists/xsl-list/archives*).

> **Note**
> On retrouvera cet exemple avec le moteur de recherche disponible à l'adresse ci-dessus : chercher « De-flattening an XML tree ».

On part d'un fichier plat, comme ceci :

flatDatas.xml

```
<?xml version="1.0" encoding="UTF-16" ?>
<recordset>
    <record/>
        <data/>
        <data/>
    <record/>
        <data/>
    <record/>
        <data/>
        <data/>
        <data/>
</recordset>
```

et on veut reconstituer un fichier hiérarchique, comme cela :

Datas.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<recordset>
   <record>
      <data/>
      <data/>
   </record>
   <record>
      <data/>
   </record>
   <record>
      <data/>
      <data/>
      <data/>
   </record>
</recordset>
```

Première étape

Dans la première étape, il s'agit de reconstituer la hiérarchie sous-jacente. L'idée est donc de rattacher chaque élément à son parent : les `<record>` doivent être rattachés au `<recordset>`, et chaque `<data>` au premier `<record>` qui apparaît dans le sens inverse de celui de la lecture du document. Pour cela, on va affecter à chaque élément une valeur égale à l'identité de son parent potentiel, et maintenir cette information grâce à une clé, dont on peut voir la structure à la figure 9-3.

La figure 9-3 montre la structure plate du fichier XML, et comment retrouver les enfants directs d'un nœud : par exemple le `<record>` *A* correspond à la valeur *VA*, qui est la valeur du nœud `<data>` *B*, ce qui signifie que le `<record>` *A* a pour enfant l'élément `<data>` *B*. De même, le nœud `<recordset>` correspond à la valeur *VRS*, qui est la valeur des trois nœuds `<record>`, ce qui signifie que le `<recordset>` a pour enfants les trois `<record>`. Bien sûr, cette dernière valeur de clé est redondante avec la structure même de

l'arbre XML de départ, qui donne déjà cette information, mais il est plus simple de garder cette redondance pour régulariser la structure du programme, qui pourra ainsi toujours passer par la clé pour obtenir les enfants directs d'un nœud.

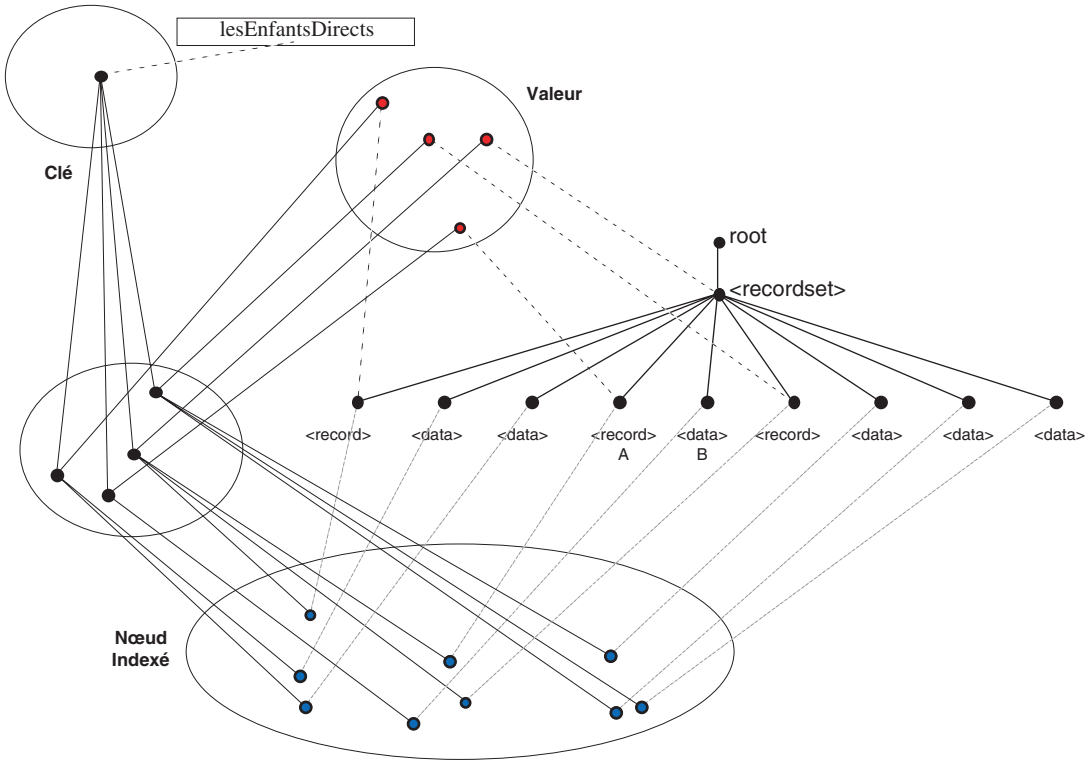


Figure 9-3

Clé pour retrouver les enfants directs d'un nœud.

Les éléments `<data>` ne correspondent à aucune valeur, parce qu'ils n'ont pas d'enfants directs.

En résumé, si l'on considère l'ensemble des valeurs, et que l'on y choisit une valeur *V* quelconque, on peut dire que :

- *V* correspond à un (et un seul) nœud *N* ;
- *V* est la valeur d'un ensemble de nœuds *E*.

On sait alors que le nœud *N* devrait avoir les nœuds *E* pour enfants. C'est pourquoi la clé est nommée `"lesEnfantsDirects"`.

Une fois qu'on a vu la structure de la clé à obtenir, il n'y a plus qu'à trouver comment l'initialiser. Partant d'un élément E quelconque dans le document original, il s'agit de déterminer son parent potentiel, et d'associer à E une valeur qui est l'identifiant de ce parent potentiel.

Mais il y a deux types d'éléments dont il faut trouver les parents : les <data> et les <record>. Pour un nœud <data>, il faut rechercher le premier <record> en remontant vers la racine du document, et la valeur associée est dans ce cas l'identifiant de ce <record>. Mais pour un nœud <record>, il faut prendre le parent <recorset>, et l'identifiant de ce <recordset> est alors la valeur du nœud <record>.

On est donc dans un cas où la clé doit être constituée en deux fois : une première déclaration pour le parent des nœuds <record>, et une deuxième pour les parents des nœuds <data>, comme ceci :

Première étape : initialisation de la clé des enfants directs

```
<xsl:key
    name="lesEnfantsDirects"
    match="record"
    use="generate-id( parent::recordset  )" />

<xsl:key
    name="lesEnfantsDirects"
    match="data"
    use="generate-id( preceding-sibling::record[1]  )" />
```

Cette façon de constituer une même clé en deux fois est parfaitement autorisée, et le résultat est conforme à ce que montre la figure 9-3.

Deuxième étape

La deuxième étape est une étape de parcours récursif de la hiérarchie mémorisée dans la clé. Un tel parcours a une structure extrêmement régulière que l'on retrouvera à l'identique dans tous les problèmes de regroupements hiérarchiques.

On commence par le sommet de la hiérarchie :

Deuxième étape : parcours hiérarchique (début)

```
<xsl:template match="recordset" >
    <recordset>
        <xsl:apply-templates
             select="key('lesEnfantsDirects', generate-id(.))"/>
    </recordset>
</xsl:template>
```

Ici, l'on construit l'élément <recordset> en lui affectant pour enfants ceux que va nous donner la clé pour la valeur correspondant à l'identifiant du nœud courant, c'est-à-dire le <recordset>.

Les enfants du <recordset> étant des <record>, il suffit maintenant d'écrire une règle pour les <record> :

Deuxième étape : parcours hiérarchique (suite)

```
<xsl:template match="record" >
    <record>
```

```
        <xsl:apply-templates
             select="key('lesEnfantsDirects', generate-id(.))"/>
    </record>
</xsl:template>
```

Les enfants d'un élément `<record>` étant des `<data>`, qui sont des éléments terminaux de la hiérarchie, on va cette fois écrire pour les éléments `<data>` une règle non récursive, qui se contentera de recopier cet élément dans le document résultat.

On peut donc maintenant donner le programme complet :

unflatten.xsl

```
<?xml version="1.0" encoding="UTF-16"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

    <xsl:output  method='xml' encoding='ISO-8859-1' indent='yes' />

    <!--
    ================================================================================
    Reconstitution des relations hiérarchiques
    ================================================================================
    -->

    <xsl:key name="lesEnfantsDirects"
             match="record"
             use="generate-id( parent::recordset  )" />

    <xsl:key
        name="lesEnfantsDirects"
        match="data"
        use="generate-id( preceding-sibling::record[1]  )" />


    <!--
    ================================================================================
    Parcours récursif de la hiérarchie
    ================================================================================
    -->


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

    <xsl:template match="recordset" >
        <recordset>
            <xsl:apply-templates
                 select="key('lesEnfantsDirects', generate-id(.))"/>
        </recordset>
    </xsl:template>
```

```
    <xsl:template match="record" >
        <record>
            <xsl:apply-templates
                 select="key('lesEnfantsDirects', generate-id(.))"/>
        </record>
    </xsl:template>

    <!--
    ============================================================================
    Eléments terminaux de la hiérarchie
    ============================================================================
    -->

    <xsl:template match="data" >
        <xsl:copy-of select="."/>
    </xsl:template>


</xsl:stylesheet>
```

Reconstitution hiérarchique d'un texte

Le programme que l'on vient de voir s'appliquait à un exemple très simple, comportant une hiérarchie presque triviale. Nous allons maintenant voir que si l'on s'attaque à la reconstruction d'une hiérarchie beaucoup plus complexe, cela ne change rien à la structure du programme, que l'on va donc pouvoir reprendre intégralement du programme précédent. Par contre, ce qui peut devenir un peu compliqué, c'est d'exprimer les concordances de motifs adéquates pour les initialisations des valeurs de la clé.

Le problème est maintenant de transformer un document XML représentant un texte (typiquement un livre ou un article), structuré en chapitres, sections, etc., pour passer d'une représentation plate à une représentation hiérarchique. Ce genre de problème peut éventuellement se poser quand on veut adapter un texte existant à une nouvelle DTD.

Ce problème revient assez souvent et sous diverses formes sur la mailing-list XSLT : faire une recherche de « transforming an incorrectly structured document » sur *www.google.fr* , par exemple.

Nous partirons de l'exemple suivant, assez représentatif du problème :

texte.xml

```
<?xml version="1.0" encoding="UTF-16" ?>
<document>
    <titreSection1> titre A </titreSection1>
        <p> para 1 </p>
        <titreSection2> titre A.1 </titreSection2>
            <p> para 2 </p>
```

```
            <p> para 3 </p>
            <p> para 4 </p>
        <titreSection2> titre A.2 </titreSection2>
            <p> para 5 </p>
            <titreSection3> titre A.2.1 </titreSection3>
                <p> para 6 </p>
                <p> para 7 </p>
        <titreSection2> titre A.3 </titreSection2>
            <p> para 8 </p>
    <titreSection1> titre B </titreSection1>
        <titreSection2> titre B.1 </titreSection2>
            <titreSection3> titre B.1.1 </titreSection3>
                <p> para 9 </p>
            <titreSection3> titre B.1.2 </titreSection3>
                <p> para 10 </p>
        <titreSection2> titre B.2 </titreSection2>
            <p> para 11 </p>
</document>
```

L'indentation permet de mieux appréhender la structure du texte, mais il faut bien voir qu'il n'y a ici aucune imbrication d'éléments.

On veut écrire une transformation qui aboutisse à ceci :

texteReconstruit.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<Document>
   <Section1>
      <titre> titre A </titre>
      <p> para 1 </p>
      <Section2>
         <titre> titre A.1 </titre>
         <p> para 2 </p>
         <p> para 3 </p>
         <p> para 4 </p>
      </Section2>
      <Section2>
         <titre> titre A.2 </titre>
         <p> para 5 </p>
         <Section3>
            <titre> titre A.2.1 </titre>
            <p> para 6 </p>
            <p> para 7 </p>
         </Section3>
      </Section2>
      <Section2>
         <titre> titre A.3 </titre>
         <p> para 8 </p>
      </Section2>
   </Section1>
```

```
   <Section1>
      <titre> titre B </titre>
      <Section2>
         <titre> titre B.1 </titre>
         <Section3>
            <titre> titre B.1.1 </titre>
            <p> para 9 </p>
         </Section3>
         <Section3>
            <titre> titre B.1.2 </titre>
            <p> para 10 </p>
         </Section3>
      </Section2>
      <Section2>
         <titre> titre B.2 </titre>
         <p> para 11 </p>
      </Section2>
   </Section1>
</Document>
```

Le programme général de reconstruction d'une hiérarchie étant applicable ici, nous pouvons donc tout de suite l'écrire, en laissant en blanc l'initialisation de la clé :

Ebauche de la transformation

```
<?xml version="1.0" encoding="UTF-16"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

    <xsl:output  method='xml' encoding='ISO-8859-1' indent='yes' />

    <!--
    ========================================================================
    Reconstitution des relations hiérarchiques
    ========================================================================
    -->

    <xsl:key name="lesEnfantsDirects"
             match="??"
             use="??" />

    <xsl:key ... />


    <!--
    ========================================================================
    Parcours récursif de la hiérarchie
    ========================================================================
    -->


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

```
        <xsl:apply-templates />
    </xsl:template>

    <xsl:template match="document" >
        <Document>
            <xsl:apply-templates
                 select="key('lesEnfantsDirects', generate-id(.))"/>
        </Document>
    </xsl:template>

    <xsl:template match="titreSection1" >
        <Section1>
            <titre><xsl:value-of select="."/></titre>
            <xsl:apply-templates
                 select="key('lesEnfantsDirects', generate-id(.))"/>
        </Section1>
    </xsl:template>

    <xsl:template match="titreSection2" >
        <Section2>
            <titre><xsl:value-of select="."/></titre>
            <xsl:apply-templates
                 select="key('lesEnfantsDirects', generate-id(.))"/>
        </Section2>
    </xsl:template>

    <xsl:template match="titreSection3" >
        <Section3>
            <titre><xsl:value-of select="."/></titre>
            <xsl:apply-templates
                 select="key('lesEnfantsDirects', generate-id(.))"/>
        </Section3>
    </xsl:template>

    <!--
    ================================================================================
    Eléments terminaux de la hiérarchie
    ================================================================================
    -->

    <xsl:template match="p" >
        <p>
            <xsl:value-of select="."/>
        </p>
    </xsl:template>

</xsl:stylesheet>
```

On voit combien le programme est semblable au précédent, du moins pour ce qui est de l'étape 2, où l'on doit faire un parcours récursif de la hiérarchie.

Pour établir l'initialisation correcte de la clé, il faut tout d'abord déterminer la structure hiérarchique possible, c'est-à-dire en fait la DTD du document résultat. Il n'est pas nécessaire de formaliser cela sous la forme d'une vraie DTD, il suffit ici de dire que :

- un `<Document>` peut avoir pour enfants des `<Section1>` ;
- une `<Section1>` peut avoir pour enfants des `<p>` ou des `<Section2>` ;
- une `<Section2>` peut avoir pour enfants des `<p>` ou des `<Section3>` ;
- une `<Section3>` peut avoir pour enfants des `<p>`.

Le problème étant de redonner à chaque élément le parent qu'il devrait avoir dans la future hiérarchie, il faut donc inverser la relation « a pour enfant » dans la liste ci-dessus :

- un `<titreSection1>` devrait avoir pour parent l'élément `<document>` ;
- un `<titreSection2>` ou un `<p>#1` devrait avoir pour parent l'élément `<titreSection1>[-1]` ;
- un `<titreSection3>` ou un `<p>#2` devrait avoir pour parent l'élément `<titreSection2>[-1]` ;
- un `<p>#3` devrait avoir pour parent l'élément `<titreSection3>[-1]`.

La notation `<p>#1` désigne ici un élément `<p>` qui est tel que si on remonte depuis cet élément `<p>` vers la racine du document, le premier élément rencontré qui ne soit pas un `<p>` est un `<titreSection1>`. Il en va de même avec `<p>#2` et `<titreSection2>`, avec `<p>#3` et `<titreSection3>`.

La notation `<titreSection1>[-1]` désigne le premier `<titreSection1>` rencontré en remontant depuis le nœud contexte vers la racine, ce qui s'exprime techniquement par une expression XPath très classique :

```
preceding-sibling::titreSection1[1]
```

On a le même genre d'expression avec `<titreSection2>[-1]` et `<titreSection3>[-1]`.

La seule réelle difficulté de ce programme réside dans l'écriture d'un motif exprimant la notation `<p>#n`. Mais en procédant progressivement, depuis un node-set contenant tous les éléments `<p>` sans distinction, que l'on filtre en enchaînant les prédicats adéquats, on parvient au résultat cherché.

Voyons cela sur le cas de `<p>#1` ; il faut partir de tous les éléments `<p>` :

```
p
```

On ne retient que ceux dont l'axe `preceding-sibling`, réduit aux éléments non `<p>`, n'est pas vide :

```
p[preceding-sibling::*[not(self::p)]]
```

Maintenant, on ne retient que ceux dont l'axe `preceding-sibling`, réduit aux éléments non `<p>`, n'est pas vide et possède un premier élément qui est un `<titreSection1>` :

```
p[preceding-sibling::*[not(self::p)][1][self::titreSection1]]
```

Arrivé à ce stade, on a de quoi initialiser partiellement la clé ; on reprend par exemple le deuxième item de l'énumération ci-dessus :

- un `<titreSection2>` ou un `<p>#1` devrait avoir pour parent l'élément `<titreSection1>[-1]` ;

Il suffit de transcrire cette phrase en initialisation de clé :

```
<xsl:key
    name="lesEnfantsDirects"
    match="titreSection2 |
           p[preceding-sibling::*[not(self::p)][1][self::titreSection1]]"
    use="generate-id( preceding-sibling::titreSection1[1]  )" />
```

Dès lors, on a tout ce qu'il faut pour terminer le programme complètement :

texte.xsl

```
<?xml version="1.0" encoding="UTF-16"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

    <xsl:output  method='xml' encoding='ISO-8859-1' indent='yes' />

    <!--
    ========================================================================
    Reconstitution des relations hiérarchiques
    ========================================================================
    -->

    <xsl:key name="lesEnfantsDirects"
             match="titreSection1"
             use="generate-id( parent::document  )" />

    <xsl:key
        name="lesEnfantsDirects"
        match="titreSection2 |
               p[preceding-sibling::*[not(self::p)][1][self::titreSection1]]"
        use="generate-id( preceding-sibling::titreSection1[1]  )" />

    <xsl:key
        name="lesEnfantsDirects"
        match="titreSection3 |
               p[preceding-sibling::*[not(self::p)][1][self::titreSection2]]"
        use="generate-id( preceding-sibling::titreSection2[1]  )" />

    <xsl:key
        name="lesEnfantsDirects"
        match="p[preceding-sibling::*[not(self::p)][1][self::titreSection3]]"
        use="generate-id( preceding-sibling::titreSection3[1]  )" />


    <!--
```

```
========================================================================
Parcours récursif de la hiérarchie
========================================================================
-->


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

<xsl:template match="document" >
    <Document>
        <xsl:apply-templates
             select="key('lesEnfantsDirects', generate-id(.))"/>
    </Document>
</xsl:template>

<xsl:template match="titreSection1" >
    <Section1>
        <titre><xsl:value-of select="."/></titre>
        <xsl:apply-templates
             select="key('lesEnfantsDirects', generate-id(.))"/>
    </Section1>
</xsl:template>

<xsl:template match="titreSection2" >
    <Section2>
        <titre><xsl:value-of select="."/></titre>
        <xsl:apply-templates
             select="key('lesEnfantsDirects', generate-id(.))"/>
    </Section2>
</xsl:template>

<xsl:template match="titreSection3" >
    <Section3>
        <titre><xsl:value-of select="."/></titre>
        <xsl:apply-templates
             select="key('lesEnfantsDirects', generate-id(.))"/>
    </Section3>
</xsl:template>

<!--
========================================================================
Eléments terminaux de la hiérarchie
========================================================================
-->

<xsl:template match="p" >
    <p>
```

```
            <xsl:value-of select="."/>
        </p>
    </xsl:template>


</xsl:stylesheet>
```

Le résultat est conforme à ce que l'on a montré au tout début.

Maintenant, la question que l'on pourrait se poser concerne la disproportion entre la faible variété des éléments terminaux (uniquement des `<p>`), et la complexité importante des motifs de clé. Si l'on a un texte plus riche en éléments terminaux, que va-t-il advenir de ces motifs ?

Supposons par exemple qu'il puisse y avoir des `<figure>` en plus des `<p>`, comme ceci :

texte.xml

```
<?xml version="1.0" encoding="UTF-16" ?>
<document>
    <titreSection1> titre A </titreSection1>
        <p> para 1 </p>
        <figure href="A.gif"/>
        <titreSection2> titre A.1 </titreSection2>
            <p> para 2 </p>
            <p> para 3 </p>
            <figure href="A.1.gif"/>
            <p> para 4 </p>
        <titreSection2> titre A.2 </titreSection2>
            <p> para 5 </p>
            <titreSection3> titre A.2.1 </titreSection3>
                <p> para 6 </p>
                <p> para 7 </p>
                <figure href="A.2.1.gif"/>
        <titreSection2> titre A.3 </titreSection2>
            <p> para 8 </p>
    <titreSection1> titre B </titreSection1>
        <figure href="B.gif"/>
        <titreSection2> titre B.1 </titreSection2>
            <titreSection3> titre B.1.1 </titreSection3>
                <p> para 9 </p>
            <titreSection3> titre B.1.2 </titreSection3>
                <p> para 10 </p>
        <titreSection2> titre B.2 </titreSection2>
            <p> para 11 </p>
            <figure href="B.2.gif"/>
</document>
```

L'initialisation partielle de la clé prend alors la forme suivante :

```
<xsl:key
    name="lesEnfantsDirects"
    match="titreSection2 |
```

```
            *[self::p or self::figure]
            [preceding-sibling::*
                [not(self::p)]
                [not(self::figure)]
                [1]
                [self::titreSection1]
            ]"
    use="generate-id( preceding-sibling::titreSection1[1]  )" />
```

Comparez avec l'initialisation lorsque <p> est le seul élément terminal :

```
<xsl:key
    name="lesEnfantsDirects"
    match="titreSection2 |
           p[preceding-sibling::*[not(self::p)][1][self::titreSection1]]"
    use="generate-id( preceding-sibling::titreSection1[1]  )" />
```

On peut améliorer la comparaison en récrivant cette initialisation de la même façon :

```
<xsl:key
    name="lesEnfantsDirects"
    match="titreSection2 |
           *[self::p]
           [preceding-sibling::*
               [not(self::p)]
               [1]
               [self::titreSection1]
           ]"
    use="generate-id( preceding-sibling::titreSection1[1]  )" />
```

On voit donc comment évolue le motif lorsqu'on ajoute des éléments terminaux possibles : globalement, le motif se complexifie, mais sa structure reste très régulière, ce qui permet donc d'envisager de transformer des textes plus riches que ceux que l'on a montré en exemple.

Le reste du programme (l'étape 2) reste inchangée dans sa structure : le seul endroit délicat est l'écriture des motifs.

texte.xsl

```
<?xml version="1.0" encoding="UTF-16"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

    <xsl:output  method='xml' encoding='ISO-8859-1' indent='yes' />

    <!--
    ========================================================================
    Reconstitution des relations hiérarchiques
    ========================================================================
    -->

    <xsl:key name="lesEnfantsDirects"
             match="titreSection1"
```

```
          use="generate-id( parent::document  )" />


<xsl:key
    name="lesEnfantsDirects"
    match="titreSection2 |
           *[self::p or self::figure]
           [preceding-sibling::*
               [not(self::p)]
               [not(self::figure)]
               [1]
               [self::titreSection1]
           ]"
    use="generate-id( preceding-sibling::titreSection1[1]  )" />

<xsl:key
    name="lesEnfantsDirects"
    match="titreSection3 |
           *[self::p or self::figure]
           [preceding-sibling::*
               [not(self::p)]
               [not(self::figure)]
               [1]
               [self::titreSection2]
           ]"
    use="generate-id( preceding-sibling::titreSection2[1]  )" />

<xsl:key
    name="lesEnfantsDirects"
    match="*[self::p or self::figure]
           [preceding-sibling::*[
               not(self::p)]
               [not(self::figure)]
               [1]
               [self::titreSection3]
           ]"
    use="generate-id( preceding-sibling::titreSection3[1]  )" />


<!--
================================================================================
Parcours récursif de la hiérarchie
================================================================================
-->


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

<xsl:template match="document" >
```

```
        <document>
            <xsl:apply-templates
            select="key('lesEnfantsDirects', generate-id(.))"/>
        </document>
    </xsl:template>

    <xsl:template match="titreSection1" >
        <Section1>
            <titre><xsl:value-of select="."/></titre>
            <xsl:apply-templates
            select="key('lesEnfantsDirects', generate-id(.))"/>
        </Section1>
    </xsl:template>

    <xsl:template match="titreSection2" >
        <Section2>
            <titre><xsl:value-of select="."/></titre>
            <xsl:apply-templates
            select="key('lesEnfantsDirects', generate-id(.))"/>
        </Section2>
    </xsl:template>

    <xsl:template match="titreSection3" >
        <Section3>
            <titre><xsl:value-of select="."/></titre>
            <xsl:apply-templates
            select="key('lesEnfantsDirects', generate-id(.))"/>
        </Section3>
    </xsl:template>

    <!--
    ========================================================================
    Eléments terminaux de la hiérarchie
    ========================================================================
    -->

    <xsl:template match="p" >
        <p>
            <xsl:value-of select="."/>
        </p>
    </xsl:template>

    <xsl:template match="figure" >
        <xsl:copy-of select="."/>
    </xsl:template>


</xsl:stylesheet>
```

Exemple atypique

Nous allons maintenant voir un exemple beaucoup plus simple, mais un peu atypique, afin de vérifier que la structure générale d'un programme de reconstruction hiérarchique tient debout malgré le changement de contexte déstabilisant.

Il s'agit du fichier `company.xml`, évoqué au début de cette section sur les regroupements :

company.xml

```
<?xml version="1.0" encoding="UTF-16"?>
<table name="COMPANY">
    NOACA<tab/>CHAR(6)<br/>
    LSACA1<tab/>VARCHAR2(35)<br/>
    CCPAA1<tab/>NUMBER(4)<br/>
    LAACA1<tab/>VARCHAR2(35)<br/>
    URL<tab/>VARCHAR2(40)<br/>
    STATRAT<tab/>VARCHAR2(1)<br/>
</table>
```

restoredCompany.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<table name="COMPANY">
   <field>
      <name>NOACA</name>
      <type>CHAR(6)</type>
   </field>
   <field>
      <name>LSACA1</name>
      <type>VARCHAR2(35)</type>
   </field>
   <field>
      <name>CCPAA1</name>
      <type>NUMBER(4)</type>
   </field>
   <field>
      <name>LAACA1</name>
      <type>VARCHAR2(35)</type>
   </field>
   <field>
      <name>URL</name>
      <type>VARCHAR2(40)</type>
   </field>
   <field>
      <name>STATRAT</name>
      <type>VARCHAR2(1)</type>
   </field>
</table>
```

Ce qui est atypique, ici, c'est que ce sont les éléments `<tab>` qui sont les éléments de deuxième niveau, rattachés aux éléments `<br>`, alors qu'ils sont placés avant. Bien que

cela change un peu de l'habitude, la structure générale du programme de transformation reste intacte :

restoredCompany.xsl

```
<?xml version="1.0" encoding="UTF-16"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

    <xsl:output  method='xml' encoding='ISO-8859-1' indent='yes' />

    <!--
    ========================================================================
    Reconstitution des relations hiérarchiques
    ========================================================================
    -->

    <xsl:key name="lesEnfantsDirects"
             match="br"
             use="generate-id( parent::table )" />

    <xsl:key
        name="lesEnfantsDirects"
        match="tab"
        use="generate-id( following-sibling::br[1]  )" />


    <!--
    ========================================================================
    Parcours récursif de la hiérarchie
    ========================================================================
    -->


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

    <xsl:template match="table" >
        <xsl:copy>
            <xsl:for-each select="attribute::*">
                <xsl:copy/>
            </xsl:for-each>

            <xsl:apply-templates
                 select="key('lesEnfantsDirects', generate-id(.))"/>

        </xsl:copy>
    </xsl:template>

    <xsl:template match="br" >
        <field>
```

```
            <xsl:apply-templates
                 select="key('lesEnfantsDirects', generate-id(.))"/>
            <type><xsl:value-of select="preceding-sibling::text()[1]"/></type>
        </field>
    </xsl:template>

    <!--
    ============================================================================
    Eléments terminaux de la hiérarchie
    ============================================================================
    -->

    <xsl:template match="tab" >
        <name>
        <xsl:value-of select="normalize-space(preceding-sibling::text()[1])"/>
        </name>
    </xsl:template>


</xsl:stylesheet>
```

Le résultat obtenu est le fichier `restoredCompany.xml` montré ci-dessus.

Pattern n° 15 – Génération d'une feuille de style par une autre feuille de style

Motivation

Il y a des cas (a priori assez rares) où l'on tombe sur des limitations du langage XSLT lui même : par exemple, le fait que seulement certains attributs soient susceptibles d'accepter des descripteurs de valeurs différées d'attributs ; ou bien le fait qu'une expression ne puisse pas être construite puis interprétée dynamiquement.

Une solution, lorsque l'on est confronté à cette difficulté, consiste à utiliser telle ou telle extension proposée par tel ou tel processeur. Mais cette solution, en général, rend le programme non portable, ce qui peut ne pas être acceptable dans certains cas.

Une autre solution consiste à écrire une feuille de style qui génère une autre feuille de style.

L'une des difficultés de la mise en œuvre d'une feuille de style qui en génère une autre, est que la feuille de style va contenir des instructions littérales à produire dans le document résultat ; mais il ne faut pas que le processeur XSLT s'aperçoive que ces éléments XML sont des instructions XSLT, sinon il va tenter de les exécuter. C'est là qu'intervient l'instruction `namespace-alias`, qui permet de produire dans le document résultat des éléments XML dans un domaine nominal différent de celui qu'ils avaient dans la feuille de style primitive.

Une autre difficulté peut aussi survenir, inattendue : c'est qu'une feuille de style générée, c'est bien ; mais une feuille de style générée et exécutée, c'est mieux.

Il est donc assez probable qu'en plus de générer la feuille de style, on veuille générer un fichier de commande pour lancer l'exécution de cette feuille de style générée avec les bons arguments. Se pose dès lors le problème d'être capable de produire plus d'un fichier résultat. C'est en principe infaisable en XSLT 1.0 ; mais le besoin étant réel, des extensions existent avec la plupart des processeurs XSLT pour permettre la production de plusieurs documents résultat. De son côté, le « Working Draft » XSLT 1.1 propose d'ajouter une instruction `xsl:document`, (et cette proposition a été reprise dans le Working Draft 2.0) dont l'effet sera de permettre la production de plusieurs documents.

Exemple

L'exemple que nous allons voir est la réalisation d'un petit interpréteur XPath : une feuille de style qui prend en donnée un fichier XML d'essai, et un fichier XML contenant des expressions XPath. La feuille de style à écrire devra générer une feuille de style capable d'évaluer les expressions XPath sur le fichier d'essai.

Voici tout d'abord ces deux fichiers XML :

BaseProduits.xml

```
<?xml version="1.0" encoding="UTF-16"?>
<BaseProduits>

    <LesProduits>

        <Livre ref="vernes1" NoISBN="193335" gamme="roman" media="papier">
            <refOeuvres>
                <Ref valeur="20000lslm"/>
            </refOeuvres>
            <Prix valeur="40.5" monnaie="FF"/>
            <Prix valeur="5" monnaie="£"/>
        </Livre>

        <Livre ref="boileaunarcejac1" NoISBN="533791" gamme="roman"
                                                    media="papier">
            <refOeuvres>
                <Ref valeur="liatlc.bn"/>
            </refOeuvres>
            <Prix valeur="30" monnaie="FF"/>
            <Prix valeur="3" monnaie="£"/>
        </Livre>

        <Enregistrement ref="marais1" RefEditeur="LC000280"
                                          gamme="violedegambe" media="CD">
            <refOeuvres>
                <Ref valeur="marais.folies"/>
```

```
            <Ref valeur="marais.pieces1685"/>
        </refOeuvres>
        <Interprètes>
            <Interprète nom="Jonathan Dunford">
                <Role xml:lang="fr"> Basse de viole </Role>
                <Role xml:lang="en"> Bass Viol </Role>
            </Interprète>
            <Interprète nom="Sylvia Abramowicz">
                <Role xml:lang="fr"> Basse de viole </Role>
                <Role xml:lang="en"> Bass Viol </Role>
            </Interprète>
            <Interprète nom="Benjamin Perrot">
                <Role xml:lang="fr"> Théorbe et guitare baroque </Role>
                <Role xml:lang="en"> Theorbo and baroque guitar </Role>
            </Interprète>
            <Interprète nom="Stéphane Fuget">
                <Role xml:lang="fr"> Clavecin </Role>
                <Role xml:lang="en"> Harpsichord </Role>
            </Interprète>
        </Interprètes>
        <Titre
        xml:lang="fr"> Les Folies d'Espagne et pièces inédites </Titre>
        <Titre
        xml:lang="en"> Folies d'Espagne and unedited music </Titre>
        <Prix valeur="140" monnaie="FF"/>
        <Prix valeur="13" monnaie="£"/>
    </Enregistrement>

    <Matériel ref="HarKar1" refConstructeur="XL-FZ158BK" gamme="lecteurCD"
                                                          marque="HarKar">
        <refCaractéristiques>
            <Ref valeur="caracHarKar1"/>
        </refCaractéristiques>
        <Prix valeur="4500" monnaie="FF"/>
        <Prix valeur="400" monnaie="£"/>
    </Matériel>

</LesProduits>

<LesOeuvres>
    ...
</LesOeuvres>

<LesAuteurs>
    ...
</LesAuteurs>

<LesGammes>
    ...
</LesGammes>
```

```
    <LesMarques>
        ...
    </LesMarques>

    <LesCaractéristiques>
        ...
    </LesCaractéristiques>

</BaseProduits>
```

XPathExpressions.xml

```
<?xml version="1.0" encoding="UTF-16"?>

<Expressions sourceFile="BaseProduits.xml">

    <XPath
        id="1"
        contextNode="Enregistrement"
        expression="descendant::Interprète[attribute::nom =
                                        'Sylvia Abramowicz']/child::Role"
    />

    <XPath
        id="2"
        contextNode="Enregistrement"
        expression="descendant::Interprète[position() = 2] /attribute::nom"
    />

    <XPath
        id="3"
        contextNode="BaseProduits"
        expression="descendant::Livre[attribute::ref = 'vernes1']/
                                                child::Prix/attribute::valeur"
    />

    <XPath
        id="4"
        contextNode="Enregistrement"
        expression="descendant::Titre[attribute::xml:lang = 'en']"
    />

    <XPath
        id="5"
        contextNode="BaseProduits"
        expression="descendant::Livre[attribute::ref = 'vernes1']/
                    child::Prix[attribute::monnaie = 'FF']/attribute::valeur"
    />

</Expressions>
```

Le but est donc d'écrire une feuille de style `XPathInterpretor.xsl` telle que le lancement :

Ligne de commande

```
xpath XPathExpressions.xml
```

produise le résultat suivant :

Résultat

```
=== id = 1 ===

{
    -- contextNode="Enregistrement"
    -- expression="descendant::Interprète[attribute::nom =
                                            'Sylvia Abramowicz']/child::Role"
         Basse de viole
        --
         Bass Viol
        --

}

=== id = 2 ===

{
    -- contextNode="Enregistrement"
    -- expression="descendant::Interprète[position() = 2] /attribute::nom"
        Sylvia Abramowicz
        --

}

=== id = 3 ===

{
    -- contextNode="BaseProduits"
    -- expression="descendant::Livre[attribute::ref = 'vernes1']/
                                          child::Prix/attribute::valeur"
        40.5
        --
        5
        --

}

=== id = 4 ===

{
    -- contextNode="Enregistrement"
    -- expression="descendant::Titre[attribute::xml:lang = 'en']"
```

```
         Folies d'Espagne and unedited music
        --

}

=== id = 5 ===

{
    -- contextNode="BaseProduits"
    -- expression="descendant::Livre[attribute::ref = 'vernes1']/
              child::Prix[attribute::monnaie = 'FF']/attribute::valeur"
        40.5
        --

}
```

La commande de lancement que nous venons de voir ne prend en donnée que le nom du fichier XML contenant les expressions à interpréter (qui lui même fournit le nom du fichier XML à utiliser comme jeu d'essai pour les évaluations XPath). Cette commande est en fait un script qui lance le processeur XSLT (Saxon, en l'occurrence), comme ceci :

script Xpath

```
java -classpath "C:\Program Files\JavaSoft\SAXON\saxon.jar;"
                               com.icl.saxon.StyleSheet -o XPathExpressions.xsl
                                   XPathExpressions.xml ../XPathInterpretor.xsl

call temp.bat
```

Dans ce script, seul le nom de fichier en gras est un argument ; tout le reste est constant.

La feuille de style `XPathInterpretor.xsl` génère la feuille de style `XPathExpressions.xsl` ainsi que le script `temp.bat` appelé à la fin.

script temp.bat

```
java -classpath "C:\Program Files\JavaSoft\SAXON\saxon.jar;"
                                          com.icl.saxon.StyleSheet  -o out.memo
                                          BaseProduits.xml XPathExpressions.xsl
```

Dans ce script, seul le nom de fichier en gras est un argument ; tout le reste est constant.

Quant à la feuille de style générée, la voici, légèrement retouchée dans sa présentation pour les besoins de la mise en page :

XPathExpressions.xsl

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<aaa:stylesheet xmlns:aaa="http://www.w3.org/1999/XSL/Transform" xmlns:xsl=
  "http://www.w3.org/1999/XSL/Transform" version="1.0">
```

```
   <aaa:output method="text" encoding="ISO-8859-1"/>

   <-- ======================== 1 ========================== -->
   <aaa:template match="/">
=== id = 1 ===
                <aaa:apply-templates mode="M1"/>
=== id = 2 ===
                <aaa:apply-templates mode="M2"/>
=== id = 3 ===
                <aaa:apply-templates mode="M3"/>
=== id = 4 ===
                <aaa:apply-templates mode="M4"/>
=== id = 5 ===
                <aaa:apply-templates mode="M5"/>
   </aaa:template>
   <-- ======================== 1 ========================== -->


   <-- ======================== 2 ========================== -->
   <aaa:template match="text()" mode="M1"/>
   <aaa:template match="text()" mode="M2"/>
   <aaa:template match="text()" mode="M3"/>
   <aaa:template match="text()" mode="M4"/>
   <aaa:template match="text()" mode="M5"/>
   <-- ======================== 2 ========================== -->



   <-- ======================== 3 ========================== -->
   <aaa:template match="Enregistrement" mode="M1">
{
    -- contextNode="Enregistrement"
    -- expression="descendant::Interprète[attribute::nom =
                                          'Sylvia Abramowicz']/child::Role"
        <aaa:for-each select="descendant::Interprète[
                           attribute::nom = 'Sylvia Abramowicz']/child::Role">
        <aaa:value-of select="."/>
        --
        </aaa:for-each>
}
   </aaa:template>

   <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->

   <aaa:template match="Enregistrement" mode="M2">
{
    -- contextNode="Enregistrement"
    -- expression="descendant::Interprète[position() = 2] /attribute::nom"
        <aaa:for-each select="descendant::Interprète[
                                        position() = 2] /attribute::nom">
        <aaa:value-of select="."/>
```

```
        --
        </aaa:for-each>
}
   </aaa:template>

   <!-- - - - - - - - - - - - - - - - - - - - - - - -->

   <aaa:template match="BaseProduits" mode="M3">
{
    -- contextNode="BaseProduits"
    -- expression="descendant::Livre[attribute::ref = 'vernes1']/
                                            child::Prix/attribute::valeur"
        <aaa:for-each select="descendant::Livre[
                    attribute::ref = 'vernes1']/child::Prix/attribute::valeur">
        <aaa:value-of select="."/>
        --
        </aaa:for-each>
}
   </aaa:template>

   <!-- - - - - - - - - - - - - - - - - - - - - - - -->

   <aaa:template match="Enregistrement" mode="M4">
{
    -- contextNode="Enregistrement"
    -- expression="descendant::Titre[attribute::xml:lang = 'en']"
        <aaa:for-each select="descendant::Titre[attribute::xml:lang = 'en']">
        <aaa:value-of select="."/>
        --
        </aaa:for-each>
}
   </aaa:template>

   <!-- - - - - - - - - - - - - - - - - - - - - - - -->

   <aaa:template match="BaseProduits" mode="M5">
{
    -- contextNode="BaseProduits"
    -- expression="descendant::Livre[attribute::ref = 'vernes1']/
                     child::Prix[attribute::monnaie = 'FF']/attribute::valeur"
        <aaa:for-each select="descendant::Livre[attribute::ref =
         'vernes1']/child::Prix[attribute::monnaie = 'FF']/attribute::valeur">
        <aaa:value-of select="."/>
        --
        </aaa:for-each>
}
   </aaa:template>

   <-- ======================== 3 ========================== -->
</aaa:stylesheet>
```

Comme nous l'avons déjà dit, l'un des problèmes pour générer une telle feuille de style est la présence d'instructions littérales XSLT à émettre dans le document résultat. Par exemple, la section 2 ci dessus est générée par le fragment de code suivant :

```
<!-- 2 -->
<xsl:for-each select="XPath">
    <aaa:template>
        <xsl:attribute name="match">text()</xsl:attribute>
        <xsl:attribute name="mode">M<xsl:value-of select="@id"/></xsl:attribute>
    </aaa:template>
</xsl:for-each>
<!-- /2 -->
```

On voit que l'instruction `template` littérale est dans un domaine nominal préfixé par `aaa`, et différent de celui d'XSLT, afin qu'elle ne soit pas prise pour une instruction XSLT à exécuter.

Ceci implique bien sûr de déclarer le domaine nominal `aaa`, comme ceci :

```
<xsl:stylesheet
    xmlns:xsl = "http://www.w3.org/1999/XSL/Transform"
    xmlns:aaa = "http://machin"
    version   = "1.0">
```

Le domaine nominal préfixé par `aaa` n'a aucune importance, n'importe quoi convient pourvu que ce ne soit pas `"http://www.w3.org/1999/XSL/Transform"`. Le choix de `"aaa"` peut paraître un peu bizarre, mais on a intérêt à choisir un préfixe qui s'oppose visuellement à `"xsl"`, sinon la feuille de style devient inextricable à relire.

Il faut de plus que dans le document résultat, les instructions XSLT (préfixées par `aaa`), soient dans le domaine nominal d'XSLT. C'est là qu'intervient l'instruction `namespace-alias`, qui permet la transposition :

```
<xsl:namespace-alias stylesheet-prefix="aaa" result-prefix="xsl"/>
```

Encore une fois, notons que les domaines nominaux sont référencés par leur préfixes, et donc que cette instruction ne demande pas à changer le préfixe dans le résultat, mais à changer le domaine nominal associé au préfixe. En clair, on demande que dans le résultat, le domaine nominal associé à `aaa` ne soit plus soit `"http://machin"`, mais celui qui est actuellement associé à `xsl`.

Le résultat est visible ci-dessus : la feuille de style générée contient des instructions XSLT préfixées par `aaa`, mais ce préfixe est bien celui de `"http://www.w3.org/1999/XSL/Transform"`, tout est donc correct pour que la feuille de style soit réellement exécutable.

Cela peut sembler désagréable que cette feuille de style n'utilise pas le préfixe `xsl` ordinaire et traditionnel, mais il faut bien voir qu'elle est générée, et qu'elle n'est pas destinée à être lue ou maintenue par un être humain.

L'autre problème à résoudre est la génération d'un document auxiliaire, dans un fichier à part. Pour cela nous utilisons l'instruction `xsl:document`, qui ne fait encore partie d'aucun standard, mais qui figure dans les Working Drafts 1.1 et 2.0. Cette instruction

utilise les mêmes attributs que `xsl:output`, à part un attribut supplémentaire, `href`, dont la valeur donne l'URI du fichier de destination du résultat de l'instanciation de son modèle de transformation. Ici, nous l'utilisons pour générer le contenu du fichier `temp.bat` :

```
<xsl:document href="temp.bat" method="text">
    <xsl:text>java -classpath "C:Files.jar;" </xsl:text>
    <xsl:text>com.icl.saxon.StyleSheet  -o out.memo </xsl:text>
    <xsl:value-of select="/Expressions/@sourceFile"/>
    <xsl:text> XPathExpressions.xsl</xsl:text>
</xsl:document>
```

Nous avons fait le tour des problèmes à résoudre pour obtenir une feuille de style qui en génère une autre ; le résultat final est montré ci-dessous :

XPathInterpretor.xsl

```
<?xml version="1.0" encoding="UTF-16"?>
<xsl:stylesheet
    xmlns:xsl = "http://www.w3.org/1999/XSL/Transform"
    xmlns:aaa= "http://machin"
    version   = "1.1"> <!-- compatibilité Saxon 6.5 -->


    <xsl:output  method='xml' encoding='ISO-8859-1' indent='yes' />

    <xsl:namespace-alias stylesheet-prefix="aaa" result-prefix="xsl"/>


    <xsl:template match='/'>
        <aaa:stylesheet
            xmlns:xsl = "http://www.w3.org/1999/XSL/Transform"
            version   = "1.0">

        <aaa:output method='text' encoding="ISO-8859-1"/>


            <xsl:document href="temp.bat" method="text">
                <xsl:text>java -classpath "C:Files.jar;" </xsl:text>
                <xsl:text>com.icl.saxon.StyleSheet  -o out.memo </xsl:text>
                <xsl:value-of select="/Expressions/@sourceFile"/>
                <xsl:text> XPathExpressions.xsl</xsl:text>
            </xsl:document>

            <xsl:apply-templates/>

        </aaa:stylesheet>

    </xsl:template>


    <xsl:template match='Expressions'>
```

```
        <!-- 1 -->
        <aaa:template>
            <xsl:attribute name="match">/</xsl:attribute>
            <xsl:for-each select="XPath">
=== id = <xsl:value-of select="@id"/> ===
                <aaa:apply-templates>
                    <xsl:attribute name="mode">M<xsl:value-of select="@id"/>
                    </xsl:attribute>
                </aaa:apply-templates>
            </xsl:for-each>
        </aaa:template>
        <!-- /1 -->

        <!-- 2 -->
        <xsl:for-each select="XPath">
            <aaa:template>
                <xsl:attribute name="match">text()</xsl:attribute>
                <xsl:attribute name="mode">M<xsl:value-of select="@id"/>
                </xsl:attribute>
            </aaa:template>
        </xsl:for-each>
        <!-- /2 -->

        <!-- 3 -->
        <xsl:for-each select="XPath">

            <aaa:template>
                <xsl:attribute name="match">
                    <xsl:value-of select="@contextNode"/>
                </xsl:attribute>
                <xsl:attribute name="mode">M<xsl:value-of select="@id"/>
                </xsl:attribute>
{
    -- contextNode="<xsl:value-of select="@contextNode"/>"
    -- expression="<xsl:value-of select="@expression"/>"
            <aaa:for-each>
                <xsl:attribute name="select"><xsl:value-of select="@expression"/>
                </xsl:attribute>
                <aaa:value-of>
                    <xsl:attribute name="select">.</xsl:attribute>
                </aaa:value-of>
            --
            </aaa:for-each>
}
        </aaa:template>
        </xsl:for-each>
        <!-- /3 -->

    </xsl:template>

</xsl:stylesheet>
```

Pattern n° 16 – Génération de pages HTML dynamiques

Motivation

Avec l'avènement des applications Internet, il a fallu mettre au point des systèmes capables de générer des pages HTML dynamiques. En effet, l'envoi de simples pages HTML statiques est en général très insuffisant pour une application qui doit extraire ou calculer des informations (typiquement en provenance d'une base de données) et les présenter à l'internaute. La génération de pages dynamiques, en général, vient en complément des pages statiques, car il est bien rare que tout soit dynamique dans une page : il y a très souvent un fond statique, sur lequel on plaque les données dynamiques. Ce pattern va montrer comment faire pour construire une page dynamique à partir d'un fond statique et de données auxiliaires.

Remarque

Nous avons déjà vu un exemple de génération de pages dynamiques, au tout début, pour donner un avant-goût du langage XSLT (voir *Un avant-goût d'XSLT*, page 9). Mais cette page dynamique était réalisée avec une feuille de style simplifiée (Simplified Stylesheet), dont les possibilités en matière de transformation et de traitement sont très limitées. Ici nous allons voir l'équivalent d'une façon plus évoluée, permettant par exemple d'envisager de la traduction « à la volée » (voir *Pattern n° 18 – Localisation d'une application*, page 533).

La première chose à faire est de définir le fond statique, car c'est lui qui va piloter l'appel des données dynamiques aux bons endroits. Le plus simple est ici de prendre un exemple. Nous allons supposer que nous voulons afficher une page d'annonce du prochain concert ; le fond statique aura l'allure suivante :

fond.xml

```
<?xml version="1.0" encoding="UTF-16"?>
<html>
    <head>
        <title>Les Concerts Anacréon</title>
    </head>
    <body>

        <H1 align="center"> Concert le <dateConcert/> </H1>
        <H4 align="center"> <lieuConcert/> </H4>
        <H2 align="center"> Ensemble <ensemble/> </H2>

        <listeMusiciens>
            <p>
                <musicien/>, <instrument/>
            </p>
        </listeMusiciens>

        <H3>
            Oeuvres de <listeCompositeurs/>
```

```
        </H3>

    </body>
</html>
```

Le fond est donc constitué majoritairement de balises HTML ; cependant, là où on a besoin de valeurs dynamiques, un élément non HTML est là pour réclamer une valeur (par exemple, `<dateConcert/>`).

Une des difficultés pouvant se apparaître dans un tel contexte, c'est la présence de données à répéter un certain nombre de fois, nombre qui est naturellement inconnu à cet endroit. Nous avons illustré ceci avec la liste des musiciens à faire figurer : on ne sait pas combien il y en a, mais ce qu'on sait, c'est que chaque musicien doit être mentionné avec son nom et son instrument. On doit donc ici se contenter de définir comment doit apparaître une des lignes d'affichage de musicien :

```
<p> unNomDeMusicien, leNomDeSonInstrument </p>
```

C'est ce qui est exprimé par le bloc :

```
<listeMusiciens>
    <p>
        <musicien/>, <instrument/>
    </p>
</listeMusiciens>
```

Avant de voir comment mettre au point une transformation XSLT adéquate, remarquons tout de suite que cette façon de procéder est conforme à l'idée générale de la séparation des compétences que l'on essaye toujours d'obtenir dans la mise au point d'une application Internet ; ici un graphiste sans compétence particulière en programmation XSLT pourra parfaitement écrire le fond statique, avec les appels de données dynamiques.

Ce fond statique va donc appeler des données dynamiques ; dans la pratique, avec un serveur d'application comme Web Logic de BEA ou d'autres du même genre, il est assez peu probable que ces données dynamiques soient placées dynamiquement dans un fichier. En fait, les données sont en mémoire sous forme d'objets, et le processus va consister à construire un arbre DOM (Document Object Model) avec ces objets.

Note

Un arbre DOM n'est rien d'autre qu'une implémentation particulière de l'arbre XML d'un document XML.

Une fois l'arbre DOM construit, il est strictement équivalent à un document XML, du point de vue du processeur XSLT. Le serveur d'application (la servlet en cours) va donc activer le thread du processeur XSLT résident (chargé avec les autres servlets par le serveur d'application) en lui transmettant l'URI du fichier statique à traiter ainsi que l'arbre DOM des données dynamiques, ce qui est faisable avec l'API TrAX (Transformation API for XML), reprise dans JAXP de SUN (Java API for XML). On se retrouve alors, du

point de vue XSLT, avec deux sources XML, une source principale, et une source auxiliaire accessible par un appel à la fonction `document()`.

Mais pour simplifier, nous supposerons ici que nous avons deux sources XML sous forme de fichiers, le seul problème étant de réaliser la transformation qui va donner la page dynamique.

Le fichier auxiliaire des données est le suivant :

Annonce.xml

```
<?xml version="1.0" encoding="UTF-16"?>

<Annonce>

    <Date>
        <Jour>Jeudi</Jour>
        <Quantième>17</Quantième>
        <Mois>janvier</Mois>
        <Année>2002</Année>
        <Heure>20H30</Heure>
    </Date>
    <Lieu>Chapelle des Ursules</Lieu>

    <Ensemble>A deux violes esgales</Ensemble>

    <Interprète>
        <Nom> Jonathan Dunford </Nom>
        <Instrument>Basse de viole</Instrument>
    </Interprète>

    <Interprète>
        <Nom> Sylvia Abramowicz </Nom>
        <Instrument>Basse de viole</Instrument>
    </Interprète>

    <Interprète>
        <Nom> Benjamin Perrot </Nom>
        <Instrument>Théorbe</Instrument>
    </Interprète>

    <Interprète>
        <Nom> Freddy Eichelberger </Nom>
        <Instrument>Clavecin</Instrument>
    </Interprète>

    <Compositeurs>
        M. Marais, D. Castello, F. Rognoni
    </Compositeurs>

</Annonce>
```

Réalisation

Le fichier principal est le fichier statique, car il est fait pour piloter l'appel des données dynamiques. Le fichier résultat (la page dynamique) est la page statique, dans laquelle les appels de valeurs sont remplacés par leur valeur : nous sommes donc dans un cas typique d'utilisation du pattern « copie non conforme ».

L'idée de base, ici, est que par défaut, les éléments doivent être recopiés sans modification ; nous allons donc adopter la règle avec priorité basse vue à la section *Copie conforme générique*, page 444 :

```
<xsl:template match="child::node()|attribute::*" priority="-10">
    <xsl:copy>
        <xsl:apply-templates select="attribute::*" />
        <xsl:apply-templates select="child::node()" />
    </xsl:copy>
</xsl:template>
```

Cette règle est à contredire uniquement pour les éléments qui constituent un appel de valeur dynamique. Dans notre cas, cela concerne les éléments `<dateConcert/>`, `<lieuConcert/>`, `<ensemble/>`, `<listeMusiciens>`, `<musicien/>`, `<instrument/>`, `<listeCompositeurs/>`. Il va donc falloir une règle spécifique pour chacun d'eux.

Les éléments `<dateConcert/>`, `<lieuConcert/>`, `<ensemble/>`, et `<listeCompositeurs/>` ne posent pas de problème particulier et reposent tous sur le même modèle ; leur traitement nécessite d'avoir accès à la source XML secondaire :

```
<xsl:param name="annonceFileRef">Annonce.xml</xsl:param>
<xsl:variable name="Annonce" select="document($annonceFileRef)/Annonce"/>

<xsl:template match='dateConcert'>

    <xsl:value-of select="$Annonce/Date/Jour" />
    <xsl:text> </xsl:text>

    <xsl:value-of select="$Annonce/Date/Quantième" />
    <xsl:text> </xsl:text>

    <xsl:value-of select="$Annonce/Date/Mois" />
    <xsl:text> </xsl:text>

    <xsl:value-of select="$Annonce/Date/Année" />
    <xsl:text> </xsl:text>

    <xsl:value-of select="$Annonce/Date/Heure" />

</xsl:template>

<xsl:template match='lieuConcert'>
    <xsl:value-of select="$Annonce/Lieu" />
</xsl:template>
```

```
<xsl:template match='ensemble'>
    <xsl:value-of select="$Annonce/Ensemble" />
</xsl:template>

<xsl:template match='listeCompositeurs'>
    <xsl:value-of select="$Annonce/Compositeurs" />
</xsl:template>
```

L'élément `<listeMusiciens>` est un peu plus compliqué à mettre au point. Il faut copier son modèle de transformation, à savoir :

```
<p>
    <musicien/>, <instrument/>
</p>
```

autant de fois que l'on va trouver l'élément `<Interprète>` dans le document `Annonce.xml`, puisqu'il faudra bien qu'il y ait autant de `<p> ... </p>` que de musiciens. Donc la règle va commencer par un `<xsl:for-each>` pour traiter le node-set des `<Interprète>` du document auxiliaire :

```
<xsl:template match='listeMusiciens'>
    <xsl:for-each select="$Annonce/Interprète">
        <xsl:variable name="current-Interprète" select="."/>
        ...
    </xsl:for-each>
</xsl:template>
```

La variable `current-Interprète` conserve le nœud courant, c'est-à-dire l'`<Interprète>` courant, car l'expérience montre qu'on a fréquemment besoin de se repérer par rapport au nœud courant dans une instruction `<xsl:for-each>` ; et même si cette variable n'est pas indispensable, elle permet de mieux s'y retrouver pour construire la règle.

Mais n'oublions pas que la règle doit instancier *n* fois le modèle de transformation indiqué plus haut ; l'une de ces instanciations peut se faire par un `<xsl:copy>`, à condition que le nœud contexte soit placé sur l'élément `<p>`. Or là où il y a les points de suspension, dans la règle ci-dessus, le nœud contexte n'est pas placé sur `<p>`, parce que l'instruction `<xsl:for-each>` déplace le nœud contexte sur le nœud en cours. Le nœud contexte, à cet endroit, est donc placé dans l'arbre XML du document auxiliaire, sur le nœud `<Interprète>` courant, celui qui précisément, est référencé par la variable `current-Interprète`.

Pour remettre le nœud contexte sur l'élément `<p>` afin de permettre la copie, il n'y a qu'une seule solution : utiliser à nouveau un `<xsl:for-each>`, puisque c'est la seule instruction capable de déplacer le nœud contexte. Pour cela, il faut sauvegarder le nœud courant à l'entrée de la règle (1), puis sélectionner son enfant direct `<p>` (2) :

```
<xsl:template match='listeMusiciens'>
    <xsl:variable name="current-listeMusiciens" select="."/> <!-- (1) -->
```

```
    <xsl:for-each select="$Annonce/Interprète">
        <xsl:variable name="current-Interprète" select="."/>
        <xsl:for-each select="$current-listeMusiciens/p"> <!-- (2) -->
            <xsl:copy>
                ...
            </xsl:copy>
        </xsl:for-each>
    </xsl:for-each>
</xsl:template>
```

Nous avons maintenant une règle presque complète ; l'instruction `<xsl:copy>` va copier l'élément `<p>`, mais les éléments enfants de `<p>` sont des éléments non HTML, qui doivent faire l'objet d'une règle spécifique, du même genre que celle de `<lieuConcert/>`, par exemple. Il est donc nécessaire d'avoir un `<xsl:apply-templates>` là où se trouvent les points de suspension, ci-dessus :

```
<xsl:template match='listeMusiciens'>
    <xsl:variable name="current-listeMusiciens" select="."/>

    <xsl:for-each select="$Annonce/Interprète">
        <xsl:variable name="current-Interprète" select="."/>
        <xsl:for-each select="$current-listeMusiciens/p">
            <xsl:copy>
                <xsl:apply-templates/>
            </xsl:copy>
        </xsl:for-each>
    </xsl:for-each>
</xsl:template>

<xsl:template match='musicien'>
    <xsl:value-of select="???" />
</xsl:template>
```

L'idée est bonne, car après avoir copié l'élément `<p>`, l'instruction `<xsl:apply-templates>` va effectivement sélectionner ses enfants directs (c'est-à-dire l'élément `<musicien/>`, le texte `', "'` (virgule, espace, guillement) et l'élément `<instrument/>`, (plus éventuellement des nœuds texte ne contenant que des espaces blancs sans intérêt), et donc la règle `<xsl:template match='musicien'>` va être sélectionnée.

Le problème, c'est que dans cette règle, on est censé instancier le nom d'un musicien ; malheureusement, le musicien en question, on l'a perdu. Il était dans la variable `current-Interprète` qui est désormais inaccessible. Inaccessible ? Pas tout à fait : l'instruction `<xsl:apply-templates>` autorise la transmission d'arguments. C'est assez rare d'avoir à utiliser cette possibilité, mais là, c'est le moment où jamais :

```
<xsl:template match='listeMusiciens'>
    <xsl:variable name="current-listeMusiciens" select="."/>

    <xsl:for-each select="$Annonce/Interprète">
        <xsl:variable name="current-Interprète" select="."/>
        <xsl:for-each select="$current-listeMusiciens/p">
```

```
            <xsl:copy>
                <xsl:apply-templates>
                    <xsl:with-param name="interprete"
                                    select="$current-Interprète" />
                </xsl:apply-templates>
            </xsl:copy>
        </xsl:for-each>
    </xsl:for-each>
</xsl:template>

<xsl:template match='musicien'>
    <xsl:param name="interprete"/>
    <xsl:value-of select="$interprete/Nom" />
</xsl:template>
```

Du coup, la règle spécifique pour l'`<instrument/>` va pouvoir être faite sur le même modèle, puisque elle sera sélectionné elle aussi par le même `<xsl:apply-templates>` :

```
<xsl:template match='instrument'>
    <xsl:param name="interprete"/>
    <xsl:value-of select="$interprete/Instrument" />
</xsl:template>
```

Mais que va-t-il advenir du nœud `text` contenant `", "` ? Le processeur va chercher une règle à lui appliquer, ne va pas en trouver, et va donc appliquer la règle par défaut (voir *Règles par défaut pour un nœud de type text ou attribute*, page 120) pour les nœuds `text`. Or cette règle par défaut ne prend pas de paramètre en donnée, alors que l'instruction `<xsl:apply-templates>` responsable de l'activation de cette règle en a transmis un. Ceci n'est pas une erreur (voir la section *Sémantique*, page 234), et heureusement, parce que sinon, cela compliquerait diablement les choses s'il fallait éviter cette situation.

On peut donc maintenant rassembler les morceaux pour obtenir le programme complet :

AnnonceConcert.xsl

```
<?xml version="1.0" encoding="UTF-16"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">

    <xsl:output  method='html' encoding='ISO-8859-1' />

    <xsl:param name="annonceFileRef">Annonce.xml</xsl:param>

    <xsl:variable name="Annonce" select="document($annonceFileRef)/Annonce"/>

    <xsl:template match="child::node()|attribute::*" priority="-10">
        <xsl:copy>
            <xsl:apply-templates select="attribute::*" />
            <xsl:apply-templates select="child::node()"/>
        </xsl:copy>
```

```
</xsl:template>

<xsl:template match='dateConcert'>

    <xsl:value-of select="$Annonce/Date/Jour" />
    <xsl:text> </xsl:text>

    <xsl:value-of select="$Annonce/Date/Quantième" />
    <xsl:text> </xsl:text>

    <xsl:value-of select="$Annonce/Date/Mois" />
    <xsl:text> </xsl:text>

    <xsl:value-of select="$Annonce/Date/Année" />
    <xsl:text> </xsl:text>

    <xsl:value-of select="$Annonce/Date/Heure" />

</xsl:template>

<xsl:template match='lieuConcert'>
    <xsl:value-of select="$Annonce/Lieu" />
</xsl:template>

<xsl:template match='ensemble'>
    <xsl:value-of select="$Annonce/Ensemble" />
</xsl:template>

<xsl:template match='listeMusiciens'>
    <xsl:variable name="current-listeMusiciens" select="."/>

    <xsl:for-each select="$Annonce/Interprète">
        <xsl:variable name="current-Interprète" select="."/>

        <xsl:for-each select="$current-listeMusiciens/p">
            <xsl:copy>
                <xsl:apply-templates>
                    <xsl:with-param name="interprete"
                                    select="$current-Interprète" />
                </xsl:apply-templates>
            </xsl:copy>
        </xsl:for-each>
    </xsl:for-each>
</xsl:template>

<xsl:template match='musicien'>
    <xsl:param name="interprete"/>
    <xsl:value-of select="$interprete/Nom" />
```

```
    </xsl:template>

    <xsl:template match='instrument'>
        <xsl:param name="interprete"/>
        <xsl:value-of select="$interprete/Instrument" />
    </xsl:template>

    <xsl:template match='listeCompositeurs'>
        <xsl:value-of select="$Annonce/Compositeurs" />
    </xsl:template>

</xsl:stylesheet>
```

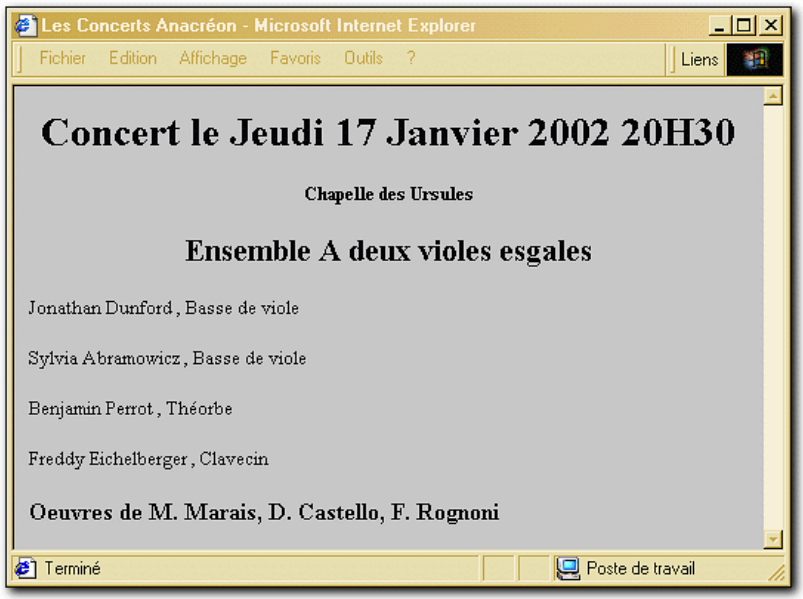
Et voici le résultat obtenu :

Annonce.html

```
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
      <title>Les Concerts Anacr&eacute;on</title>
   </head>
   <body>
      <H1 align="center"> Concert le Jeudi 17 janvier 2002 20H30 </H1>
      <H4 align="center"> Chapelle des Ursules </H4>
      <H2 align="center"> Ensemble A deux violes esgales </H2>
      <p>
                        Jonathan Dunford , Basse de viole
      </p>
      <p>
                        Sylvia Abramowicz , Basse de viole
      </p>
      <p>
                        Benjamin Perrot , Th&eacute;orbe
      </p>
      <p>
                        Freddy Eichelberger , Clavecin
      </p>
      <H3>
                    Oeuvres de
                M. Marais, D. Castello, F. Rognoni
      </H3>
   </body>
</html>
```

Dans le fichier ci-dessus, les lignes blanches ont été enlevées pour gagner de la place ; le rendu HTML est montré à la figure 9-4.

Figure 9-4
Aspect de la page dynamique obtenue.



Conclusion

Le pattern que nous venons de voir permet de générer des pages HTML dynamiques, mais il ne faut pas croire que ce soit la seule application possible. Par exemple, sur le même principe, on peut construire un générateur de classes Java, qui utilise des fonds statiques de ce style :

Measure.tmpl

```
<?xml version="1.0"?>
<template>
//
//
//------------------------------------------------------------------------------
// stored Measure <MeasureName/>
//------------------------------------------------------------------------------
//

    protected <MeasureType/> <MeasureName/>;

    public final <MeasureType/> get<MeasureNamewithCapital/>() {
        return <MeasureName/>;
    }

    public final void set<MeasureNamewithCapital/>(
                                            <MeasureType/> <aMeasureName/> ){
```

```
        super.setAttribute("<MeasureName/>", <aMeasureName/>.toString() );
        <MeasureName/> = <aMeasureName/>;
        Assertion.ensure( get<MeasureNamewithCapital/>() == <aMeasureName/> );
    }

</template>
```

Le générateur construit une classe Java en assemblant un certain nombre de petites pièces de puzzle comme celle montrée ci-dessus (l'élément `<template>`), où les éléments XML sont des appels de valeurs, exactement comme dans le fichier `fond.xml` utilisé en exemple pour la génération de pages HTML dynamiques. Les « valeurs » sont obtenues dans des fichiers XML source auxiliaires qui dérivent des différentes phases antérieures de modélisation, comme on peut en avoir un aperçu dans l'extrait ci-dessous :

Entity.xml

```
...
<Entity name="Contract" persistance="simple" IHMconnected="true">
    <MeasureRef id="beginningDate" />
    <MeasureRef id="contractNbr" isPartOfKey="yes" />
    <MeasureRef id="type" />
</Entity>

<Entity name="Country" persistance="simple" >
    <MeasureRef id="countryCode" isPartOfKey="yes" />
    <MeasureRef id="zoneEuro" />
</Entity>

<Measure name="contractNbr" type="String" access="stored"  />
<Measure name="type" type="String" access="stored"  />
<Measure name="coefficient" type="String" access="stored"  />
<Measure name="countryCode" type="String" access="stored"  />
<Measure name="zoneEuro" type="Boolean" access="stored"  />
<Measure name="beginningDate" type="BusinessDate" access="stored" />
...
```

Pattern n° 17 – Génération de pages HTML dynamiques pour un portail

Le pattern que nous allons voir ici reprend le précédent, en le modifiant légèrement pour montrer comment générer une page dynamique pour un portail. Ce qui va changer, c'est le fait qu'il ne va plus y avoir un seul fichier XML auxiliaire de description des données, mais plusieurs. En effet, dans un portail, il y a multitude d'informations rassemblées dans une seule page, et ces informations ne proviennent pas toutes du même fichier, évidemment. Il y a donc de nombreuses sources XML à exploiter et à synthétiser dans la génération dynamique de la page. Pour les besoins de l'exemple, nous aurons deux sources XML, et ce n'est plus le programme XSLT qui connaît d'avance le nom des fichiers