XSLT Orienté Objets – Eric Bréchemier

Version mise à jour (mars 2006)
extraite de la documentation du projet Binary 4 Java (décembre 2004)

XSLT est un langage de programmation qui ne s'écrit pas sous forme simplement textuelle, comme la grande majorité des langages de programmation, mais sous forme de balises XML.

Pour comparaison, voici un extrait de programme en XSLT et l'extrait équivalent dans le langage de programmation Java :

- en XSLT, pour exécuter un traitement particulier en fonction des différentes valeurs possibles de la variable $input, on écrira :

```
<xsl:choose>
  <xsl:when test="$input = 0">
    <xsl:call-template name="handle0"/>
  </xsl:when>
  <xsl:when test="$input = 1">
    <xsl:call-template name="handle1" />
  </xsl:when>
  <xsl:otherwise>
    <xsl:call-template name="handleOther">
      <xsl:with-param name="otherValue" select="$input"/>
    </xsl:call-template>
  </xsl:otherwise >
</xsl:choose>
```

- en Java on écrira

```
switch(input) {
  case 0: {
    handle0();
    break;
  }
  case 1: {
    handle1();
    break;
  }
  default: {
    handleOther(input);
  }
}
```

XSLT définit un langage de **transformation de documents XML**. Sa première vocation est d'appliquer des feuilles des styles à des données XML provenant par exemple d'une base de données, pour permettre leur lecture sur différents types de terminaux : en HTML pour un navigateur Web, en SVG pour les graphiques ou en XSL-FO pour l'impression.

Toutefois, ses facultés ne se limitent pas aux feuilles de styles. XSLT est utilisé plus généralement pour transformer des données qui ne suivent pas la même logique, par exemple dans un dialogue entre deux applications.

Une transformation XSLT définit un ensemble de règles à appliquer au document d'entrée pour obtenir le document de sortie. Chaque règle s'applique sur un ou plusieurs éléments du document d'entrée et construit une partie du document de sortie.

Les données qui constituent les instructions d'un programme en XSLT sont elles-mêmes structurées et manipulables plus facilement par un autre programme. En effet, il existe de nombreuses librairies standard pour accéder, modifier et transformer les données qui suivent la syntaxe XML.

Ceci permet de générer plus facilement du code XSLT, ce qui était déterminant dans notre cas, et facilite par ailleurs le développement de processeurs XSLT, qui s'appuient sur des analyseurs XML standards développés indépendamment.

Ces analyseurs suivent principalement deux logiques :

- Une vision du document XML comme une arborescence que l'on peut parcourir à sa guise. Le standard DOM définit une API (Application Programming Interface, la définition d'un ensemble de fonctions dont plusieurs organisations vont pouvoir proposer des implémentations) d'accès au document sous la forme d'une arborescence de noeuds.

- Une vision plus réduite avec un curseur qui progresse toujours en avant, du début à la fin du document. Le modèle SAX définit une API événementielle : le document est parcouru en entier une seule fois, et des événements sont émis à toutes les étapes clefs de ce parcours.

De même, en XSLT, il existe deux approches principales pour aborder le problème de la transformation des données :

- une approche guidée par la logique du document d'entrée. Il s'agit de la technique la plus simple, qui consiste à traiter le document d'entrée dans l'ordre de ses éléments. C'est le mode par défaut des processeurs XSLT, qui appliquent les règles suivantes :

```
Pour traiter un élément
- si l'utilisateur a défini une règle pour cet élément, applique-la
- sinon traite chaque élément enfant dans l'ordre du document
et recopie tous les textes que tu trouves vers le document de sortie.
```

  Ce qui donne en XSLT :

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

  On utilise dans ce premier cas des règles (template) "match" qui s'appliquent à tous les noeuds qui respectent le motif défini par l'attribut match.

- une approche guidée par la logique du document de sortie. On définit dans ce cas des règles "name", qui ne se déclenchent pas directement, mais doivent être appelées explicitement. Il s'agit d'un mécanisme d'appel de fonction, utilisé classiquement dans la plupart des langages de programmation (voir le premier exemple en page précédente).

Dans le projet B4J, je me suis inspiré de l'approche objet pour définir une troisième voie. Du point de vue théorique, un objet est caractérisé par :

1. un état interne (ses données)
2. des opérations (ses méthodes)
3. une identité

Considérons une règle de transformation XSLT : elle s'applique sur des noeuds du document d'entrée qui sont d'un certain type, il s'agit de ses **données**. Elle représente une **opération** à effectuer sur ces données pour construire des données de sortie. Chaque élément d'information du document d'entrée est **identifié** par sa position dans ce document. Les briques de base des objets sont donc bien présentes.

Par ailleurs, la programmation orientée objets est caractérisée par trois concepts :

1. l'**encapsulation** : un objet est comme un iceberg, sa plus grande partie est immergée, (*son état interne est protégé, on ne peut le modifier directement ce qui garantit son intégrité et la possibilité de modifier sa représentation interne indépendamment des opérations proposées*), et on ne peut accéder qu'à la partie visible de l'iceberg (ses opérations).
2. **l'héritage** : un objet d'une classe fille hérite des caractéristiques des objets de la classe mère (état interne et opérations). Il peut redéfinir ces caractéristiques, par exemple pour compléter les traitements de base proposés par la classe mère.
3. **le polymorphisme** : un objet d'une classe fille peut toujours se "faire passer" pour un objet de la classe parente : il peut être utilisé partout à la place d'un objet de la classe parente.

Les données manipulées par une transformation XSLT sont **encapsulées** au sein du document d'entrée, qui n'est jamais modifié directement par la transformation ; ses données sont seulement recopiées pour construire un autre document en sortie.

Le mécanisme d'imports de XSLT correspond au concept d'**héritage**. Il ressemble à l'inclusion d'une feuille de style dans une autre, en y ajoutant la gestion des priorités entre les règles de la transformation principale et celles de la transformation importée ; les règles de la transformation principale s'appliquent prioritairement sur celles de la transformation importée, de la même manière que les méthodes d'une classe fille sont appelées prioritairement sur celles de sa classe mère. De la même manière qu'en Java, il est possible en XSLT d'appeler la méthode de base de la classe mère à partir de la méthode plus spécialisée de la classe fille en utilisant `<xsl:apply-imports />`. La seule limite par rapport à Java est qu'il est impossible de transmettre des arguments à cette méthode (limitation corrigée dans la deuxième version de XSLT, en cours d'élaboration au W3C).

Enfin, la transformation principale peut s'appliquer sur tous les fichiers de données sur lesquels s'appliquait la transformation de base, et les règles qu'elle définit **remplacent de manière transparente** les règles correspondantes de la transformation importée.

On peut donc programmer en XSLT en suivant une philosophie objet. Le tableau suivant récapitule les correspondances entre les concepts XSLT et les concepts objets :

Objet	XSLT
Objet	Transformation (correspond à un fichier, comme en Java)
Etat interne	Données du document d'entrée
Opération	Règle de transformation
Encapsulation	Transformation des donnés
Héritage et Polymorphisme	Import et Priorités

Une transformation XSLT est typiquement constituée d'un ensemble de règles qui s'appliquent aux principaux éléments du document d'entrée. Il s'agit de la première approche évoquée, guidée par la logique du document d'entrée.

```
<xsl:template match="my:element">
  <!-- do something on my:element attributes and child nodes -->
</xsl:template>
```

Toutefois, il est parfois nécessaire d'appliquer plusieurs traitements distincts sur un même élément, à partir de contextes différents. Par exemple, on devra faire deux passes sur les données d'entrée pour construire un document et sa table des matières. La notion de **mode** permet de distinguer les traitements à effectuer suivant le contexte :

```
<xsl:template mode="index" match="my:element">
  <!-- create entry in table of contents -->
</xsl:template>
<xsl:template mode="body" match="my:element">
  <!-- create output content -->
</xsl:template>
```

Le changement de mode est effectué au niveau de la commande **apply-templates** :

```
<xsl:apply-templates mode="index" />
```

Dans le cadre de XSLT Orienté Object, la troisième voie proposée, la notion de mode prend une signification supplémentaire : chaque mode correspond à une méthode différente de nos objets. On peut ainsi mettre en relation les structures XSLT ci-dessous :

```
<xsl:template mode="print-tasks" match="node()" />
<xsl:template mode="print-tasks" match="calendar">
  <!-- ... -->
</xsl:template>

<xsl:template mode="print-meetings" match="node()" />
<xsl:template mode="print-meetings" match="calendar">
  <!-- ... -->
</xsl:template>

<xsl:template mode="print-vacations" match="node()" />
<xsl:template mode="print-vacations" match="calendar">
  <!-- ... -->
</xsl:template>
```

avec les définitions d'objets suivantes en Java :

```
public class Calendar {

  public void printTasks(){
     // ...
  }

  public void printMeetings(){
     // ...
  }

  public void printVacations(){
     // ...
  }
}
```

Dans l'exemple ci-dessus, les règles `<xsl:template mode="xx" match="node()" />` surchargent les règles par défaut. Elles jouent le rôle de filet de sécurité lorsque l'élément `calendar` est absent du document : elles stoppent net le traitement par défaut qui consiste à recopier récursivement tous les noeuds textes descendants du noeud courant.