Cahiers GUTenberg

POLYDOC : UN EXEMPLE D'APPLICATION XML POUR LA CRÉATION PERSONNALISÉE DE POLYCOPIÉS

Michel Cubero-Castan

Cahiers GUTenberg, nº 35-36 (2000), p. 133-155.

<http://cahiers.gutenberg.eu.org/fitem?id=CG_2000___35-36_133_0>

PolyDoc : un exemple d'application XML pour la création personnalisée de polycopiés

Michel Cubero-Castan

Institut National des Sciences Appliquées de Toulouse
Département de Génie Electrique et Informatique
Michel.Cubero-Castan@insa-tlse.fr

Résumé

Le Document Object Model (DOM) de W3C est une norme qui spécifie une API permettant d'effectuer des opérations de création, de modification ou de consultation des éléments d'un document XML représenté sous forme arborescente. Des implémentations du DOM existent pour divers langages de programmation, en particulier Java.

Cet article présente PolyDoc, une application Java basée sur le DOM, et permettant la traduction d'un document XML vers HTML, LaTeX, Open e-Book, ... ou encore XML (mais avec une DTD différente). A travers PolyDoc, nous décrivons un processus de production de documents suivant trois étapes :

- écriture du contenu dans le format XML,
- mise en forme globale personnalisée en Java,
- production du résultat via HTML, LaTeX, ...

A titre d'exemple, un document type, la fiche de programmation, est traitée de manière détaillée.

1. Introduction

Souvent, lorsqu'un enseignant a la charge d'un cours, il est amené à préparer un certain nombre de documents liés à cet enseignement. Le premier peut être une fiche descriptive du cours contenant par exemple les objectifs de l'enseignement, le plan détaillé, le nombre de séances ou encore des références vers des ouvrages ou des ressources liés à cet enseignement. Le deuxième document peut être un polycopié du cours dont le plan est justement celui présenté dans la fiche descriptive. A ce polycopié, nous pourrions ajouter un recueil d'exercices éventuellement accompagnés de corrigés, ainsi qu'un recueil de travaux pra-

tiques. Enfin, l'enseignant peut créer des transparents pour illustrer un cours magistral.

Quels outils utiliser pour la réalisation de ces documents ? Avant de répondre à cette question, essayons d'examiner quelles seraient les fonctionnalités de l'outil *idéal*. La notion d'idéal est ici personnelle, elle est directement liée aux habitudes de l'utilisateur. Pour l'auteur de cet article, voici quelques éléments qu'il juge sinon nécessaires, du moins pratiques :

- séparation entre contenu et forme
- partage de fragment
- plusieurs formats de sortie
- plusieurs formats d'entrée
- personnalisation

Examinons maintenant ces différents points.

1.1. Séparation entre contenu et forme

Les utilisateurs de LaTeX ne nous contrediront pas, lorsqu'on rédige un document, il est préférable de s'intéresser tout d'abord au contenu avant la forme. D'autant plus que la forme, c'est à dire la présentation du résultat, peut dépendre du support utilisé. Beaucoup de systèmes utilisent cette approche, par exemple par le biais de feuilles de style.

1.2. Partage de fragment

Lorsqu'on rédige plusieurs documents pour un même enseignement, on est souvent amené à utiliser des fragments d'un document dans un autre. Par exemple, un transparent pourra contenir une formule mathématique déjà présente dans le polycopié. La solution du *copier-coller*, quoique très simple, est loin d'être satisfaisante : si la formule initiale doit être modifiée, on doit reporter la modification dans tous les documents contenant cette formule. Une autre solution est l'insertion de fichier : le fragment partagé se trouvera dans un fichier séparé qui sera inséré dans les différents documents par une commande d'insertion. En général une telle commande, si elle existe, est globale au niveau du fichier, c'est à dire qu'il ne peut y avoir que l'insertion du contenu complet du fichier et non un fragment. Pour profiter de cette fonctionnalité, on peut être conduit à avoir une multitude de petits fichiers, ce qui n'est guère facile à gérer.

Dans certains cas, l'approche "un fichier par fragment" n'est pas possible. Ainsi, dans un cours de programmation, on peut être amené à inclure dans le polycopié, une petite partie de programme pour illustrer tel ou tel point. En général, les langages de programmation ne permettent pas la rédaction d'une même

procédure répartie sur plusieurs fichiers. Cette difficulté conduit à insérer du texte dans la programmation, comme on le fait dans la programmation littérale, plutôt que l'inverse.

1.3. Plusieurs formats de sortie

Jusqu'à ces dernières années, tous les documents utilisaient le papier ou assimilé (transparent) comme support. L'outil LaTeX est alors un excellent moyen pour obtenir des documents offrant une qualité typographique digne des professionnels de l'édition.

L'arrivée de nouvelles technologies (NT) amène l'enseignant à diversifier les supports des documents. Le polycopié se présentera sous la forme d'une liasse de feuilles de papier mais aussi sous forme d'un fichier au format PDF ou au format Open e-Book destiné à être consulté via un livre électronique. Il en est de même pour le recueil d'exercices. Le recueil de travaux pratiques pourra se retrouver au format HTML de manière à offrir, lors de sa consultation, les avantages de l'hypertexte pour l'accès aux divers documents annexes qui ne seraient accessibles que via le réseau. La fiche descriptive du cours peut aussi utiliser ce format pour autoriser sa consultation à partir d'internet. A l'ancien transparent, nous pourrons lui préférer la rétroprojection d'écran d'ordinateur pour apporter plus de dynamique dans le cours.

1.4. Plusieurs formats d'entrée

Les nouveaux supports d'information deviennent de plus en plus multimédia. Il faut donc pouvoir intégrer aisément ces médias dans le document source. On doit pouvoir insérer par exemple une animation graphique, une séquence sonore ou vidéo, ou encore inclure une appliquette java.

Sans aller jusqu'au multimédia, une simple illustration dans un texte doit pouvoir provenir de plusieurs sources différentes sans avoir à effectuer manuellement les conversions nécessaires pour tel ou tel support de sortie. Par exemple, on peut produire telle illustration avec le logiciel de dessin xfig et telle autre à partir de code PSTricks par exemple. Comme pour le partage de fragment de document, l'utilisateur ne doit manipuler qu'un seul fichier, la source de l'illustration. Il ne doit pas être obligé de sauvegarder explicitement son illustration au format eps pour être traité par LaTeX, ou au format png pour être visualisé par les navigateurs, ou encore dans d'autres formats.

1.5. Personnalisation

Enfin, le dernier point et non le moindre a trait à la personnalisation de la présentation du résultat. C'est une affaire de goût. Chacun doit pouvoir présenter un résultat suivant sa personnalité. Un outil de production de documents doit donc être facilement malléable de ce point de vue.

2. L'approche LaTeX

Est-ce que LaTeX correspond à l'outil idéal ? Examinons chacun des points énumérés ci-dessus.

2.1. Séparation entre contenu et forme

Le principe de macro permet à l'utilisateur de LaTeX de s'intéresser au contenu avant la forme. Le traitement de la présentation est relégué à la programmation des macros utilisables. C'est donc un bon point pour LaTeX.

2.2. Partage de fragment

LaTeX permet l'inclusion de fichier dans un document grâce à la commande `\input{`*`filename`*`}`. Cependant, l'inclusion concerne un fichier complet et non pas un fragment. C'est une manière de découpage de fichier volumineux en petits fichiers plutôt qu'un partage de fragment.

Pour certains cas particuliers, par exemple l'insertion de listing, il est possible de sélectionner et d'insérer un fragment de fichier en indiquant le numéro de la première ligne à insérer, ainsi que le numéro de la dernière. C'est une méthode qui possède le défaut de l'adressage absolue : si le fichier contenant le fragment à inclure se trouve modifié, par exemple en rajoutant une nouvelle ligne au début du fichier, il faut modifier les adresses de ligne dans la commande d'insertion. Bien sûr, il est possible de créer une extension de LaTeX réalisant ce traitement d'insertion, mais ce type de programmation relève du TeXpert.

2.3. Plusieurs formats de sortie

L'utilisateur de TeX/LaTeX dispose déjà d'une panoplie de logiciels permettant de produire le document sur des supports différents du support papier. On peut ainsi utiliser, soit latex2html pour la production de fichiers au format HTML, soit pdflatex pour obtenir un fichier au format PDF. Malheureusement, il est

difficile (sinon très difficile) d'obtenir, avec ces outils, un résultat personnalisé au niveau de la visualisation du document final.

2.4. Plusieurs formats d'entrée

LaTeX est principalement dédié à la publication sur papier. Il n'est pas question ici d'inclure des éléments issus d'autres média. Seuls des éléments de type image peuvent être utilisés. Le problème ici vient du format, principalement du PostScript Encapsulé (EPS), qui oblige à créer explicitement des sauvegardes de dessin sous ce format. Ici aussi, il est possible de créer une extension de LaTeX réalisant automatiquement une telle conversion, mais, encore une fois, cela relève du TeXpert.

2.5. Personnalisation

LaTeX permet une personnalisation de l'apparence du résultat relativement simple, grâce à une multitude d'extensions disponibles. Le manque se fait sentir dès que l'on utilise des outils complexes tels que latex2html.

3. L'approche XML

SGML apporte une première solution à tous ces problèmes. Par la création de nouvelles balises, il est possible de faire cohabiter, dans le même fichier source, différents médias. De plus, il existe beaucoup d'outils permettant le traitement de fichier SGML pour obtenir différents formats supports. On peut ainsi citer sgml2latex, sgml2html, sgml2text, etc. Cependant, souvent ces outils travaillent sur des documents respectant une DTD unique se voulant la plus générale possible, et qui, par voie de conséquence, devient complexe. Il suffit d'examiner la DTD de DocBook pour s'en rendre compte.

En général, pour la rédaction des documents liés à un cours, l'enseignant n'a pas besoin d'utiliser un large éventail de balises, ni une DTD trop générale. Ce qu'il lui faut, c'est plutôt une DTD adaptée à son texte, avec, peut-être, des balises spécifiques.

La norme XML offre une solution très satisfaisante pour ce type de traitement. Par son langage de balises extensible, il est possible de prendre en compte tout type d'élément dans un document. Grâce aux feuilles de style XSL, la traduction simple vers d'autres formats (LaTeX, HTML) est facile. L'interface DOM permet une manipulation plus complexe du document au moyen d'une programmation Java ou C++. Ceci permet également la génération de pages

HTML à la volée, si l'on souhaite produire des pages en fonction du navigateur utilisé pour la visualisation.

3.1. Séparation entre contenu et forme

Comme SGML, XML est un langage de balisage. Un document XML correspond à un contenu structuré par des balises assimilables de ce fait aux macros de LaTeX. Il y a une parfaite séparation entre contenu et forme puisque cette dernière ne sera définie que par l'application qui traitera le document XML : soit une application C++ ou Java qui transformera le contenu pour lui donner une forme grâce à l'interface DOM, soit une feuille de style XSL qui fournira les règles de transformation à appliquer sur le document XML.

3.2. Partage de fragment

Outre l'inclusion classique de fichier grâce aux entités externes, le langage XML peut être enrichi d'objets de type lien (XLink) permettant, entre autres, un partage de fragment entre fichiers XML. Encore en stade de développement, il n'existe pas encore de moteurs XLink suffisamment stables. On peut toutefois, via la programmation Java et l'interface DOM, réaliser un sous ensemble des fonctionnalités du lien XML.

3.3. Plusieurs formats de sortie

Du fait de la séparation entre contenu et forme, il n'y a pas de restriction sur les formats de sortie possibles pour un même document XML. La recommandation décrivant le langage des feuilles de style XSL propose un format intermédaire de description de sortie : fo. Déjà des convertisseurs de ce format vers d'autres formats commencent à apparaitre (par exemple de fo vers PDF).

3.4. Plusieurs formats d'entrée

La définition du langage XML a été orientée, dès le départ, pour offrir un format d'échange de données de tout types. Par le biais des balises, on peut donc inclure, dans un fichier XML, tout type de donnée. C'est l'application XML qui sera chargée de diriger telle ou telle donnée vers le programme pouvant la traiter.

3.5. Personnalisation

Une personnalisation de l'apparence que l'on veut donner au résultat pourra se faire par l'écriture d'applications spécifiques ou de feuilles de style spécifiques.

4. L'outil PolyDoc

L'outil PolyDoc est un programme Java chargé de traduire un document XML soit vers LaTeX soit vers HTML. Le choix de Java vient de sa modularité et surtout de sa portabilité. Ajouter de nouvelles balises dans la DTD du document est une opération très simple. Du fait de la modularité de la programmation Java, il est presque aussi facile de rajouter les méthodes de traduction associées à une nouvelle balise.

4.1. Un exemple de document : la fiche de programmation

Pour décrire les capacités de l'approche PolyDoc, regardons comment créer et traiter un type de document simple : la fiche de programmation.

La fiche de programmation est un document décrivant la programmation d'un problème donné. C'est un document ne nécessitant pas une présentation très élaborée ; la DTD utilisée sera très simple.

4.1.1. La DTD de la fiche de Programmation

Une fiche de programmation sera contituée de 5 éléments de premier niveau :

titre sans commentaire !

auteur l'auteur de la fiche. Cet élément sera lui-même constitué de 4 sous éléments indiquant respectivement son nom, son prénom, son adresse électronique, et l'adresse de sa page personnelle sur internet.

date la date de mise à jour de la fiche, constituée de 3 sous éléments : le jour, le mois et l'année.

problème la description du problème posé, organisée en une suite de paragraphes.

solution la solution du problème organisée en suite de paragraphes et de sections, une section correspondant ici à une étape clé dans le processus de programmation. La section comprendra un titre et une suite de paragraphes. Il n'y aura pas d'autres subdivisions.

Le fichier contenant la DTD commencera par :

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

<!ELEMENT ficheProg (titre, auteur, date, problème, solution)>
  <!ELEMENT titre (#PCDATA)>
  <!ELEMENT auteur (nom, prénom, mél, siteweb)>
    <!ELEMENT nom (#PCDATA)>
    <!ELEMENT prénom (#PCDATA)>
    <!ELEMENT mél (#PCDATA)>
    <!ELEMENT siteweb (#PCDATA)>
  <!ELEMENT date (jour, mois, an)>
    <!ELEMENT jour (#PCDATA)>
    <!ELEMENT mois (#PCDATA)>
    <!ELEMENT an (#PCDATA)>
  <!ELEMENT problème (p*)>
  <!ELEMENT solution (p*, sect*)>
    <!ELEMENT sect (titreSect, p*)>
      <!ELEMENT titreSect (#PCDATA)>
...
```

Parmi les éléments disponibles, deux ont été rassemblés par l'entité interne `xlink` :

```
...
<!ENTITY % xlink " insertionXML | insertionFichier ">

<!ELEMENT insertionXML EMPTY>
<!ATTLIST insertionXML show CDATA #FIXED "embed">
<!ATTLIST insertionXML actuate CDATA #FIXED "auto">
<!ATTLIST insertionXML xml:link CDATA #FIXED "simple">
<!ATTLIST insertionXML href CDATA #IMPLIED>
<!ATTLIST insertionXML inline CDATA #FIXED "true">
<!ATTLIST insertionXML idref CDATA #REQUIRED>
<!ATTLIST insertionXML id ID #IMPLIED>

<!ELEMENT insertionFichier EMPTY>
<!ATTLIST insertionFichier src CDATA #REQUIRED>
<!ATTLIST insertionFichier début CDATA #IMPLIED>
<!ATTLIST insertionFichier fin CDATA #IMPLIED>
<!ATTLIST insertionFichier id ID #IMPLIED>
...
```

Le nom de l'entité, `xlink`, a été choisi pour rappeler les pointeurs XML proposés par le consortium w3. Ce ne sont pas de véritables pointeurs XML, ceux-ci n'étant pas actuellement supportés par les outils utilisables.

Le premier, `insertionXML`, correspond à un pointeur XML très limité ; parmi tous ses attributs, seuls les attributs `href` et `idref` sont utilisés par PolyDoc. Les autres sont là pour pouvoir être utilisés par des moteurs XLINK, lorsque ces derniers seront disponibles. Le fonctionnement de cet élément est simple : après la lecture d'un document, PolyDoc remplace tous les éléments de nom `insertionXML` par l'élément contenu dans le fichier référencé par la valeur de l'attribut `href`, et possédant un identificateur dont le nom est celui correspondant à la valeur de l'attribut `idref`. Si l'attribut `href` n'est pas présent, le document référencé est lui même.

L'élément `insertionFichier` permet d'insérer non pas des éléments XML mais de simples données (PCDATA : Parsed Character Data) issues d'un fichier texte (ici, un programme source java). L'attribut `src` indique le fichier source duquel ces données vont être extraites. Les attributs `début` et `fin` indiquent les marques délimitant la zone à extraire (si l'attribut `début` est inexistant, le début de la zone est le début du fichier ; l'attribut `fin` a un fonctionnement analogue). La marque correspond à la chaîne de caractères délimitée par `"/*marque="` et `"*/"`.

Les éléments restant à décrire sont ceux correspondant aux paragraphes et aux éléments contenus dans les paragraphes. Tous ces éléments sont rassemblés dans l'entité interne `texteFormaté`. Ils possèdent tous un attribut ID de manière à pouvoir être référencés par un lien XML.

```
...
<!ENTITY % texteFormaté "p | em | latex | java |  %xlink; ">
...
```

L'élément suivant permet de délimiter un paragraphe :

```
...
<!ELEMENT p (#PCDATA |  %texteFormaté; )* >
<!ATTLIST p position (centrée|aligné) "aligné">
<!ATTLIST p id ID #IMPLIED>
...
```

Le paragraphe peut contenir tout élément indiqué par l'entité interne `texteFormaté`. L'attribut `position` permet de choisir entre un paragraphe centré ou un paragraphe aligné.

L'élément `em` permet un formatage de texte :

```
...
<!ELEMENT em  (#PCDATA |  %texteFormaté; )*>
<!ATTLIST em format (italique|gras|tty) 'italique'>
<!ATTLIST em id ID #IMPLIED>
...
```

L'attribut `format` permet de choisir entre du texte *italique*, **gras** ou avec des caractères `télétype`.

Les deux derniers éléments permettent d'insérer dans le code XML, du code LaTeX ou du code Java :

```
...
<!ELEMENT latex (#PCDATA)>
<!ATTLIST latex xml:space (default|preserve) 'preserve'>
<!ATTLIST latex id ID #IMPLIED>

<!ELEMENT java (#PCDATA|insertionFichier)*>
<!ATTLIST java xml:space (default|preserve) 'preserve'>
<!ATTLIST java id ID #IMPLIED>
```

4.1.2. Une fiche de programmation XML

Regardons à présent comment on peut créer le document XML à partir de cette DTD. Nous avons choisi, comme exemple de fiche, le problème de la résolution d'une équation du second degré dans l'espace R. La version papier de la fiche est donnée en annexe, une version en ligne est accessible à l'adresse `http ://wwwdgei.insa-tlse.fr/~castan/GUT2000`. Examinons les aspects intéressants de sa rédaction.

Le sujet commence par la ligne suivante :

Soit l'équation $a \times x^2 + b \times x + c = 0.$

On constate qu'elle contient une présentation digne de ce nom pour l'équation du second degré. La DTD n'offrant pas les possibilités d'écriture d'expressions mathématiques de MathML, nous devons passer par le biais de LaTeX pour obtenir une présentation correcte. Le document sera donc rédigé de la manière suivante :

...

```
Soit l'équation
<latex id="ax2-bx-c">$a\times x^2 + b \times x + c = 0$</latex>.
</p><p>
Ecrire un programme <em format="gras">Java</em> permettant sa
résolution dans R.
...
```

L'élément `latex` ci-dessus contient un attribut id de valeur `"ax2-bx-c"`. Cet identificateur nous permettra d'utiliser, dans la suite de la fiche, un élément `insertionXML` qui sera remplacé par l'élément `latex`. C'est ce qui apparait au début de la partie **solution** du document XML :

```
...
  <solution>
    <p position="centrée"><insertionXML idref="ax2-bx-c"/></p>
    <sect>
      <titreSect>Utilisation du programme</titreSect>
      <p>Le programme que l'on va obtenir aura pour nom
...
```

Enfin le document contient des extraits d'un programme java. Ce programme est entièrement contenu dans le fichier `racine.java`. On va pouvoir récupérer des extraits de ce fichier grâce à l'élément `insertionFichier`. Par exemple, pour inclure un fragment du fichier java, la deuxième section du document contient le code XML suivant :

```
...
suivante (il convient de vérifier à ce niveau la validité des
coefficients, soit a#0).
      <java>        ...
<insertionFichier src="racine.java" début="etape1"
                                    fin="etape2"/>
        ...
</java>
...
```

Le fichier source `racine.java` doit alors contenir :

```
        ...
            System.exit(0);
        }
        /*marque=etape1*/
        /* on recupere les valeurs des coefficients */
```

```
        double a = Float.valueOf(args[0]).doubleValue();
        double b = Float.valueOf(args[1]).doubleValue();
        double c = Float.valueOf(args[2]).doubleValue();

        if (a==0.0) {
            System.out.println("a doit etre non nul");
        } else {
            /* resolution dans R */

    /*marque=etape2*/            double delta = b*b - 4.*a*c;

        ...
```

On peut remarquer que les commentaires relatifs aux marques sont éliminés dans le fichier résultat, que ces marques soient celles recherchées ou non !

4.2. Prise en compte d'une nouvelle DTD

La section 4.1 précédente a montré comment comment créer une DTD et un document respectant la structure de cette DTD. Il reste à examiner, maintenant, comment programmer en Java, les méthodes et classes qui vont permettre de traiter de tels documents pour produire différents formats de sortie (HTML, LaTeX ou autre).

4.2.1. Organisation générale

Le traitement, par PolyDoc, d'un document décrit par la DTD de nom `DTD` pour une production de document dans un format de nom `FORMAT` est réalisé par la classe PolyDoc.DTD.FORMAT. Ainsi, pour notre exemple, nous avons dû écrire une classe PolyDoc.ficheProg.HTML pour produire la version en ligne, et une classe PolyDoc.ficheProg.Latex pour produire la version papier :

```
package PolyDoc.ficheProg;

public class HTML {
    org.w3c.dom.Document docXML;
    ...
    public HTML (String nomFichierXML,
                 String répertoireDestination,
                 boolean traitementDessin) {
        ...
```

```
    }

    public void traitement() {
        ...
    }
    ...
}


package PolyDoc.ficheProg;

public class Latex {
    org.w3c.dom.Document docXML;
    ...
    public Latex (String nomFichierXML,
                  String répertoireDestination,
                  boolean traitementDessin) {
        ...
    }

    public void traitement() {
        ...
    }
    ...
}
```

Le constructeur de chaque classe est défini avec trois paramètres :

- `nomFichierXML` : donne le nom du fichier XML contenant le document source pour ce format. En effet, bien que la partie XML est commune pour tous les formats, la partie déclaration de la DTD peut être différente (par exemple, le caractère μ sera défini pour le format HTML par l'entité
  `<!ENTITY micro "&#181;">`
  et pour le format Latex par l'entité
  `<!ENTITY micro "<latex>$\mu$</latex>">`
- `répertoireDestination` : donne le répertoire dans lequel le ou les fichiers produits seront créés,
- `traitementDessin` : indique si les différentes images contenues dans le fichier source doivent être produites ou non (ainsi, dans notre exemple, cela permet d'éviter de créer les images qui correspondent à des équations écrites en LaTeX, à chaque traitement du fichier XML, de manière à obtenir un traitement, certe incomplet, mais plus rapide, lors de la phase de rédaction du document).

Un des traitements réalisés par le constructeur est la création de l'objet docXML construit à partir d'une plateforme XML (plusieurs sont disponibles, voir `http : //www.w3.org/XML`). Cet objet possède une hiérarchie arborescente où les noeuds sont les balises XML, et les feuilles, les données textuelles du document.

4.2.2. Méthode traitement()

La méthode `public void traitement()` permet la génération du ou des fichiers résultats à partir de l'objet `docXML`. Par exemple, dans le cas de la classe `PolyDoc.ficheProg.Latex`, nous pourrions avoir :

```
public void traitement() {
    try {
        String nomFichLatex =
             (new File(nomFichierXML)).getName();
        nomFichLatex =
             nomFichLatex.substring
                  (0, nomFichLatex.indexOf("."));

        File fichierLatex = new File(répertoireDestination,
                                     nomFichLatex+".tex");
        FileOutputStream strFich =
             new FileOutputStream(fichierLatex);
        WriterLatex out =
             new WriterLatex(new DataOutputStream(strFich));

        // Génération de l'en-tête du fichier

        // Génération de la partie Problème

        // Génération de la partie Solution

        // Génération de la fin du fichier latex

        out.close();
        strFich.close();
        System.out.println("Terminé");
    } catch (Exception e) {
        System.err.println("Problème durant la création "+
                           "des pages Latex "+e);
```

```
        }
    }
```

La classe `WriterLatex` est une sous-classe de `java.io.PrintWriter` rassemblant des méthodes d'usage général, comme la génération de l'en-tête d'un fichier LaTeX, pour rendre la programmation plus lisible. Entre autre, cette classe redéfinira la méthode `public void print(String str)` de `java.io.PrintWriter` de manière à effectuer des transformations des caractères pouvant être interprétés comme des commandes LaTeX, par exemple :

- `'~'` doit être remplacé par `"\\verb+~+"`
- `'$'` doit être remplacé par `"\\$"`
- ...

Une autre méthode `WriterLatex.printLatex(String str)` permettra justement d'éviter cette transformation lors de la production de texte contenant des commandes LaTeX.

Examinons maintenant comment programmer une partie de la méthode `PolyDoc.ficheProg.Latex.traitement`, par exemple celle qui produit la partie solution, partie contenant des sections :

```
// Génération de la partie Solution
  Element racine = docXML.getDocumentElement();
  Node nodeSolution = racine.getFirstChild()
                            .getNextSibling()
                            .getNextSibling()
                            .getNextSibling()
                            .getNextSibling();
     printTexteFormaté(nodeSolution.getFirstChild(),
                       out);

  NodeList lstSect =
      ((Element) nodeSolution)
            .getElementsByTagName("sect");
  for (int noSect=0; noSect&lt;
       lstSect.getLength();
       noSect++) {
      Node noeudSection = lstSect.item(noSect);
      out.println();
      out.printLatex("\\section{");
      out.print(noeudSection.getFirstChild()
                            .getFirstChild()
                            .getNodeValue());
```

```
                out.println("}");
                printTexteFormaté(
                        noeudSection.getFirstChild()
                                    .getNextSibling(),
                        out);
                out.println();
            }
```

La méthode `getDocumentElement()`, méthode standard de l'interface DOM, permet d'obtenir la racine de l'arbre du document (c'est à dire, ici, l'élément `ficheProg`). Les méthodes `getFirstChild()` et `getNextSibling()` permettent d'obtenir respectivement, le premier élément d'un noeud de l'arbre et l'élément suivant de même niveau. Ainsi, il est facile d'atteindre le noeud de la partie solution à partir de la racine, grâce à ces deux méthodes.

L'accès aux différentes sections se fait à partir de la méthode `getElementsByTagName(String tag)` qui renvoie une liste de noeuds correspondant à des éléments dont le nom est donné en paramètre à la méthode. `getLength()` permet alors de connaître le nombre de noeuds dans cette liste, tandis que `item(int i)` permet d'obtenir le i^e noeud de la liste.

La transformation du graphe en code LaTeX est alors simple. Elle utilise la méthode `printTexteFormaté(Node nd, WriterLatex out)` qui traite une succession de paragraphes, validée par la DTD. Regardons le codage Java de cette méthode :

```
public void printTexteFormaté(Node nd, WriterLatex out) {
    switch (nd.getNodeType()) {
    case Node.CDATA_SECTION_NODE :
        out.println(nd.getNodeValue());
        break;
    case Node.TEXT_NODE :
        out.print(nd.getNodeValue());
        break;
    case Node.ELEMENT_NODE :
        String tag = ((Element) nd).getTagName();
        try {
            Class[] paramètresFormels = {
               Class.forName("org.w3c.dom.Element"),
               Class.forName("PolyDoc.ficheProg.WriterLatex")
               };
            Object méthode = getClass().getMethod(
                                    "TAG_"+tag,
```

```
                                        paramètresFormels);
                Object[] arguments = new Object[2];
                arguments[0] = nd;
                arguments[1] = out;
                ((java.lang.reflect.Method) méthode)
                            .invoke(this, arguments);
            } catch(Exception e) {
                if (tag.equals("sect")) break;
                System.out.println("Tag inconnu : "+tag);
            }
        }
    }
```

Le fonctionnement de cette méthode est plus simple qu'il n'y parait. Un noeud de l'arborescence est passé en paramètre. Si ce noeud correspond à une donnée textuelle (noeud texte ou noeud CDATA), on envoie, grâce à l'instance de `WriterLatex` passée en second paramètre, le texte associée sur le fichier en cours de création. Sinon, si ce noeud est un noeud élément, on fait appel à la méthode dont le nom est le nom de l'élément préfixé par la chaîne "TAG_" et dont les paramètres sont les mêmes que ceux de `printTexteFormaté`. Si cette méthode n'existe pas, c'est que nous avons atteint la fin d'une section (en fait le noeud section suivant). Le traitement s'arrête alors normalement.

Examinons maintenant quelques unes des méthodes préfixées par la chaîne "TAG_".

4.2.3. Traitement des éléments

La première méthode que l'on va traiter est celle associée à la balise `java` :

```
public void TAG_java(Element nd, WriterLatex out) {
    out.printlnLatex("\\nopagebreak\n");
    out.printlnLatex("\\begin{lstlisting}{}");
    printTexteFormatéLatex(nd.getFirstChild(), out);
    out.println();
    out.printlnLatex("\\end{lstlisting} ");
}
```

Son travail est de générer le début et la fin du code contenu dans la balise (on utilise ici l'extension LaTeX `lstlisting`. La suite du travail sera fait par les éléments contenus dans l'élément `java` grâce à la méthode `printTexte-FormatéLatex` (cette méthode est différente de la méthode `printTexteFormaté` car elle n'opère aucune transformation sur les caractères transmis).

Examinons maintenant l'élément `em` qui, par rapport à l'élément précédent, possède un attribut :

```
public void TAG_em(Element nd, WriterLatex out) {
    String format = nd.getAttribute("format");
    if (format.equals("italique"))
              out.printLatex("{\\itshape ");
    else if (format.equals("tty"))
              out.printLatex("{\\ttfamily ");
    else out.printLatex("{\\bfseries ");
    printTexteFormaté(nd.getFirstChild(), out);
    out.printLatex("}");
}
```

L'acquisition de la valeur de l'attribut est possible via la méthode `getAttribute (String nom)` de la classe `org.w3c.dom.Element`. Le reste se passe de commentaires.

4.3. Un groupement de documents PolyDoc

Nous venons de présenter un exemple de document traité par PolyDoc. Nous avons vu au début de cet article qu'en général, pour un enseignement donné, on doit générer plusieurs documents. Par exemple, avec cet article, nous avons le résumé utilisé lors de la phase de soumission, puis l'article lui-même, enfin l'exemple complet développé.

Comment regrouper tous ces documents ? La solution que nous avons choisi dans PolyDoc est d'y faire référence dans un autre document XML grâce à la balise `docPolyDoc` suivante :

```
<!ELEMENT docPolyDoc (description, format*)>
<!ATTLIST docPolyDoc id ID #REQUIRED>
<!ATTLIST docPolyDoc type CDATA #REQUIRED>

<!ELEMENT description (#PCDATA)>

<!ELEMENT format EMPTY>
<!ATTLIST format type CDATA #REQUIRED>
<!ATTLIST format src CDATA #REQUIRED>
<!ATTLIST format repDst CDATA #REQUIRED>
```

Un élément `docPolyDoc` permet d'identifier un document traité par PolyDoc. L'attribut `id` donne l'identificateur du document tandis que l'attribut `type` indique le type de document (article, polycopié, présentation, ...). Cet élément peut lui-même contenir un ou plusieurs éléments associés à un format de sortie du résultat. Chaque élément format dispose de trois attributs :

- `type` : le type de sortie (HTML, LaTeX, fo, XML, ...)
- `src` : le chemin d'accès au fichier XML du document, pour le format désiré
- `resDst` : le répertoire de destination des fichiers produits

Par exemple, le document correspondant à l'exemple sera désigné par l'élément suivant :

```
<docPolyDoc id="idExemple"
            type="ficheProg">
  <description>L'exemple de l'article</description>
  <format type="HTML"
          src="ProgEquationHTML.xml"
          repDst="/home/castan/public_html/GUT2000doc/"/>
  <format type="Latex"
          src="ProgEquationLatex.xml"
          repDst="Latex/"/>
</docPolyDoc>
```

Un document XML contenant cette balise pourrait être celui qui permettra de générer la page d'accueil du groupe de documents sur le Web. Le traitement d'un élément `docPolyDoc` sera identique au traitement de la balise `url`, balise permettant de créer un lien hypertexte sur le document.

L'outil PolyDoc utilisera ce document "page d'accueil" pour lancer le traitement de tel ou tel élément `docPolyDoc`, soit via une interface graphique, soit par la ligne de commande de lancement de l'application. Ainsi, la ligne de commande suivante démarrera le traitement de l'exemple pour une sortie LaTeX (GUT2000doc.xml étant le fichier contenant des balises `docPolyDoc`, et en particulier, une ayant `idExemple` pour valeur de l'attribut `id`) :

```
java PolyDoc.Main GUT2000doc.xml idExemple latex
```

5. Conclusion

Nous avons voulu montrer, grâce à un exemple simple, comment on peut exploiter les avantages de XML et l'apport de l'interface standard DOM en Java

pour traiter facilement un document, en suivant un processus de production en 3 étapes :

- écriture du contenu dans le format XML,
- mise en forme globale personnalisée en Java,
- production du résultat via HTML, LaTeX, ...

Une approche comme celle de PolyDoc ne rejette donc pas l'énorme contribution de la communauté TeX/LaTeX, mais peut correspondre à une évolution naturelle comme ce fut le cas pour LaTeX vis à vis de TeX.

La version en ligne de cet article est accessible à l'adresse

http ://wwwdgei.insa-tlse.fr/~castan/GUT2000doc/index.html

L'outil PolyDoc est accessible à l'adresse

http ://wwwdgei.insa-tlse.fr/~castan/PolyDoc/index.html

Des exemples de documents générés avec PolyDoc sont accessibles à l'adresse

http ://wwwdgei.insa-tlse.fr/~castan/Enseignements/index.html

Appendix

Résolution d'une équation du second degré

Michel Cubero-Castan

Soit l'équation $a \times x^2 + b \times x + c = 0$. Ecrire un programme **Java** permettant sa résolution dans R.

Rappel : les solutions sont données par la formule $\frac{-b \pm \sqrt{b^2 - 4 \times a \times c}}{2 \times a}$

Solution

$$a \times x^2 + b \times x + c = 0$$

1. Utilisation du programme

Le programme que l'on va obtenir aura pour nom `racine`. Les valeurs des coefficients seront données sur la ligne de lancement du programme. Par exemple, si on veut résoudre l'équation $x^2 - 7 \times x + 12.25 = 0$

on utilisera la ligne de commande :

```
java racine 1 -7 +12.25
```

On se contentera d'écrire une seule méthode, la méthode `main`. L'aspect général du programme sera donc :

```
/**
 * Programme permettant de resoudre dans R une equation du
 * second degre ax2+bx+c=0.<br>
 * Les coefficients a b et c sont donnes sur la ligne de
 * lancement du programme: 'java racine 3 1 5' pour la
 * resolution de 3x2+bx+5=0.
 */
public class racine {

    public static void main(String[] args) {
        if (args.length!=3) {
            System.out.println
                ("usage: java racine 1 -7 +12.25");
            System.exit(0);
        }

        ...
}
```

2. Traitement des coefficients

Les valeurs des coefficients sont fournies sous forme de chaînes de caractères. Il faut donc les convertir en valeurs de type double, ce qui se fait en deux étapes : l'obtention d'une instance de la classe `Float`, puis l'obtention d'une valeur de type `double` grâce à la méthode `doubleValue()`. On obtient donc la partie de

code suivante (il convient de vérifier à ce niveau la validité des coefficients, soit a#0).

```
        ...
        /* on recupere les valeurs des coefficients */
        double a = Float.valueOf(args[0]).doubleValue();
        double b = Float.valueOf(args[1]).doubleValue();
        double c = Float.valueOf(args[2]).doubleValue();

        if (a==0.0) {
            System.out.println("a doit etre non nul");
        } else {
            /* resolution dans R */

        ...
```

3. Résolution du problème

Nous pouvons maintenant résoudre le problème. Suivant la valeur de delta= $\sqrt{b^2 - 4 \times a \times c}$, on obtiendra 0, 1 ou 2 solutions dans R. La partie du programme réalisant cette résolution est :

```
        ...
          double delta = b*b - 4.*a*c;
          if (delta<0.0)
              System.out.println("Pas de solution reelle!");
          else if (delta==0.0)
              System.out.println("Une solution double : "
                                 +(-b/(2.*a)));
          else
              System.out.println("Deux solutions : "+
                    ((-b+Math.sqrt(delta))/(2.*a))+" et "+
                    ((-b-Math.sqrt(delta))/(2.*a)));
      }
  }
}
```