Cahiers GUTenberg

LE DOCUMENT OBJECT MODEL (DOM)

François ROLE, Philippe VERDRET

Cahiers GUTenberg, nº 33-34 (1999), p. 155-171.

<http://cahiers.gutenberg.eu.org/fitem?id=CG_1999___33-34_155_0>

Le Document Object Model (DOM)

François ROLE

Université Paris 8, 22 rue Sibuet, 75012 Paris, France
<Francois.Role@inria.fr>

Philippe VERDRET

Sonovision-Itep, 86 rue Regnault, 75652 Paris CEDEX 13, France,
<pverdret@sonovision-itep.fr>

Résumé. Cet article décrit le *Document Object Model* (DOM), une hiérarchie d'interfaces normalisées proposée par le consortium W3, permettant aux logiciels applicatifs d'accéder à la structure des documents XML et d'en manipuler le contenu. Après une description théorique du DOM nous donnons des exemples d'utilisation pratique dans trois langages (Java, Perl et JScript). Cette mise en parallèle des trois langages est destinée à donner un aperçu des fonctions offertes par le DOM ainsi que de son caractère de neutralité vis-à-vis des langages de programmation. Nous terminons cet article par une discussion des limites et des évolutions prévisibles du DOM.

Abstract. *The present article gives an overview of the* Document Object Model *(DOM), a hierarchy of standard interfaces proposed by the W3 Consortium. It allows application programs to access the structure of* XML *documents and manipulate their content. We start with a brief theoretical description of the* DOM. *Then we have a look at a few use cases expressed in three languages (Java, Perl and JScript). The parallel treatment in these three languages should allow you to get an idea of the functionality offered by the* DOM, *as well as emphasize its programming language neutral character. At the end of the article we discuss the present limitations of the* DOM *and its foreseeable future evolution.*

Mot-clés : DOM, XML, modèle objet d'un document, structure d'un document.

1. Qu'est-ce que le DOM?

XML tend à devenir le méta-format universel de représentation des données et des documents. XML est défini dans une spécification publiée par le consortium W3 [4].

D'autres spécifications du même consortium [1] définissent des aspects complémentaires des documents XML et de leur traitement :

- XLink [7] et XPointer [8] qui couvrent les aspects hypertextuels des documents ;
- XSL [6] qui traite du rendu graphique des documents ainsi que de leur transformation ;
- le DOM qui fait l'objet de cet article.

Le DOM [1] peut schématiquement être décrit comme une interface de programmation indépendante des langages et des plate-formes permettant aux programmes applicatifs de naviguer dans la structure des documents XML et d'en manipuler le contenu.

Dans un système utilisant le DOM, un parseur analyse (et valide éventuellement) un fichier balisé en XML pour en construire une représentation sous forme d'une collection d'objets (éléments, attributs, fragments de texte, etc.) organisée hiérarchiquement. Les méthodes définies sur ces objets peuvent alors être invoquées par des logiciels applicatifs (par exemple un navigateur internet, un éditeur structuré, etc.) voulant accéder au document XML ou le manipuler. Ce scénario typique est décrit à la figure 1.

On notera que tous les parseurs XML disponibles actuellement ne construisent pas forcément une représentation arborescente. C'est notamment le cas des parseurs dits « évènementiels » qui exposent les constituants d'un document les uns après les autres, de manière séquentielle et sans construire explicitement un arbre, ce qui est suffisant pour de nombreux traitements. On aura par contre intérêt à utiliser le DOM chaque fois que le traitement à réaliser nécessite un accès aléatoire et/ou en contexte aux constituants élémentaires du document.

De façon plus précise, le DOM a pour fonction d'étendre les possibilités de manipulation des documents XML. Sur un navigateur, il peut permettre de filtrer un document, de l'afficher seulement à un certain niveau de détail (en extrayant la liste des auteurs dans un tableau, en créant des tables des matières avec dépliement/repliement des niveaux, etc.). Il peut également être utilisé pour enrichir des données, par exemple, par une pose d'annotations. Comme pour toute interface normalisée, l'avantage est que les méthodes (au sens de ce terme en programmation orientée objet) d'accès utilisées pour accéder à l'arbre XML sont les mêmes quels que soient les applications et les langages utilisés [2].

1. Les documents publiés par le consortium W3 ont, en fonction de leur degré de validation, des statuts divers. On distingue par exemple les *working drafts*, les *proposed recommendations*, les *recommendations*. Les documents ayant le statut de *recommendation* sont les documents officiellement approuvés par le consortium. XML et le DOM sont des *recommendations*.

2. Pour une description plus détaillée des principes et des motivations ayant présidé à la conception du DOM, le lecteur peut se référer à [9].

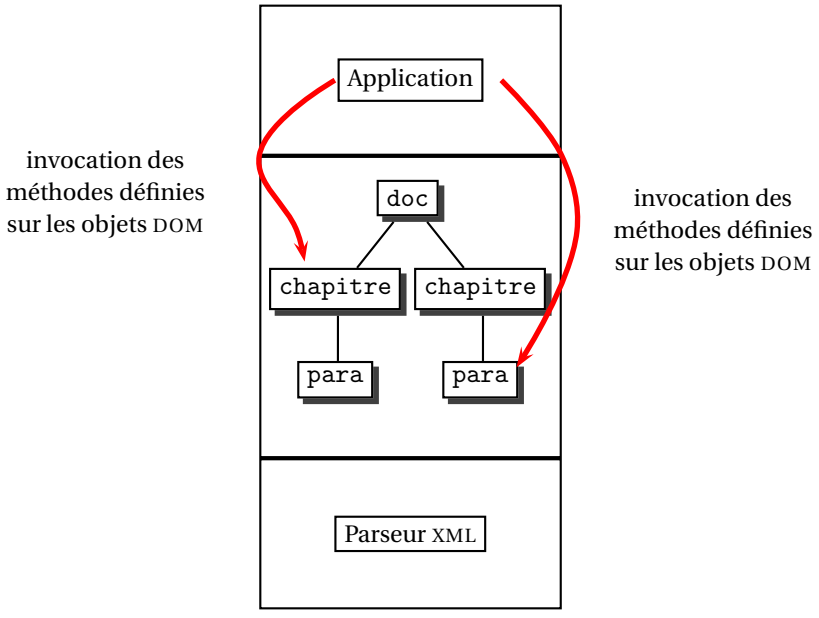

FIGURE 1 – *Les applications accèdent à la représentation* DOM *construite par le parseur* XML

Dans la suite de cet article, nous commençons par présenter les interfaces du DOM et nous donnons quelques indications sur la structure de cette spécification qui est un document complexe. Nous donnons ensuite des exemples pratiques d'utilisation dans trois langages très utilisés dans les applications Web : Java, JavaScript (dans sa version normalisée ECMAScript) et Perl. Ces exemples, même courts, doivent donner au lecteur un aperçu des possibilités de traitement offertes par le DOM. Par ailleurs, comme les langages retenus illlustrent des styles de programmation différents et sont utilisés dans des contextes applicatifs variés, ces exemples sont susceptibles de permettre de mesurer intuitivement jusqu'à quel point le DOM atteint son objectif ultime : présenter aux logiciels applicatifs une interface uniforme d'accès et de manipulation. Nous concluons cet article par une discussion portant sur les limites actuelles du DOM et sur ses perspectives d'évolution.

2. Interfaces du DOM

Comme son nom le suggère, le *Document Object Model* est un modèle objet qui représente un document XML comme une collection d'objets implémentant des interfaces.

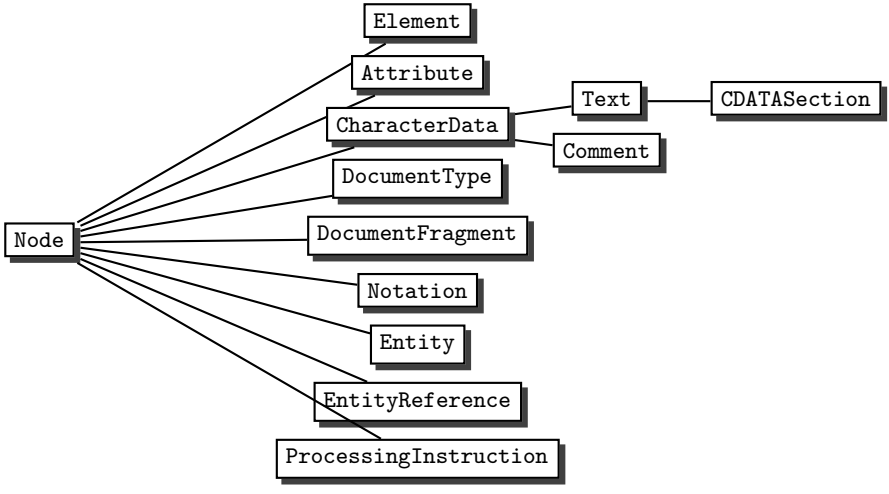

FIGURE 2 – *La hiérarchie des interfaces du* DOM

Comme dans les langages objet, ces interfaces spécifient les attributs et les comportements des objets qui les implémentent. Les interfaces DOM sont organisées en une hiérarchie d'héritage dont la racine est l'interface `Node`. Comme on peut le constater en figure 2, les interfaces DOM qui descendent de l'interface `Node` reprennent tous les constituants habituels d'un document XML. On notera que le DOM permet de représenter tout ce que l'on peut trouver dans un document XML, c'est-à-dire aussi bien ce qui relève de la structure logique *stricto sensu* (les éléments) que d'autres composantes (attributs, entités, commentaires).

L'interface `Node` regroupe des attributs généraux dont héritent tous ses descendants. Parmi ces attributs, on trouve :

- `nodeType` : un entier court non signé qui indique le type du nœud (élément, attribut, entité, etc.) ;
- `nodeName` : une chaîne qui indique le nom du nœud ;
- `nodeValue` : une chaîne qui indique la valeur du nœud ;
- `parentNode` : une nœud correspondant au père du nœud ;
- `childNodes` : la liste des nœuds fils du nœud.

Pour quelques descendants de `Node`, certains des attributs indiqués ci-dessus n'ont pas de sens. Ils prennent alors une valeur conventionnelle (par exemple, la valeur de `nodeName` pour un commentaire est `#comment`) ou égale à `null` (c'est le cas de l'attribut `nodeValue` d'un nœud implémentant l'interface `EntityReference`).

```
<?xml version="1.0" ?>
<!-- Ceci est un exemple de document XML -->
<rapport>
<titre>Utilisation pratique du DOM </titre>
<texte>Un exemple de contenu.</texte>
</rapport>
```

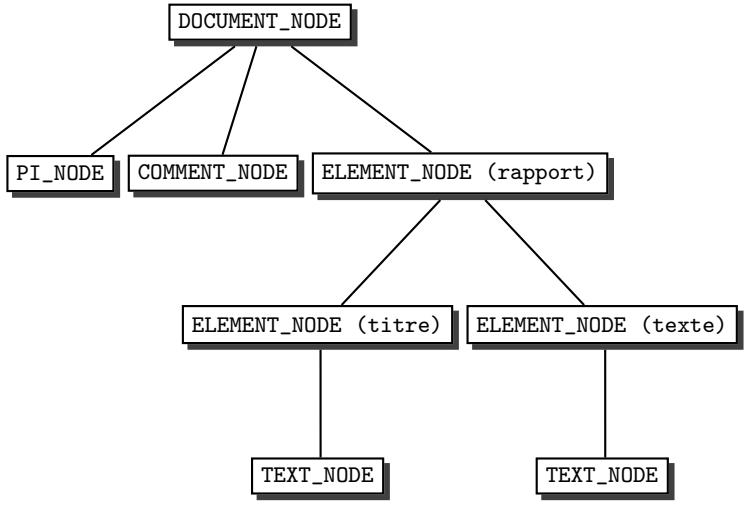

FIGURE 3 – *Exemple d'un document* XML *et de sa représentation* DOM

Le DOM utilise les interfaces dont nous venons de parler pour modéliser un document XML sous forme d'un arbre dont la racine est un nœud implémentant l'interface `Document`. Ce nœud `Document` peut avoir pour fils des nœuds implémentant les interfaces `DocumentType` (si le document est accompagné de sa DTD), `Comment`, `ProcessingInstruction` et `Element`. Un nœud de type `Document` ne peut avoir qu'un seul fils de type `Element`. Par exemple, figure 3, on voit que le nœud `Document` a trois fils :

- un nœud de type `ProcessingInstruction` (`nodeType = PI_NODE`);
- un nœud de type `Comment` (`nodeType = COMMENT_NODE`);
- un nœud de type `Element` (`nodeType = ELEMENT_NODE`).

Le DOM comporte également deux interfaces utilitaires `NodeList` et `NamedNodeMap` servant à constuire des structures de données. Ces structures sont par exemple utiles pour stocker la liste des fils ou des attributs d'un élément.

Il n'est pas possible de traiter de façon détaillée de toutes les interfaces décrites dans la spécification DOM. Avant de présenter des exemples de réalisations dans les sections suivantes, nous terminons cette introduction en fournissant quelques indications utiles sur la structure de ce document qui est une spécification complexe.

Conceptuellement, le DOM comporte deux niveaux (*Document Object Model Level 1* et *Document Object Model Level 2*), chaque niveau correspondant à un ensemble de fonctionnalités. Actuellement, seul le niveau 1 est officiellemement normalisé. Il offre les fonctionnalités de base pour accéder au contenu des documents XML et les manipuler. Des possibillités supplémentaires sont prévues pour le niveau 2, mais ce dernier est toujours en cours de définition et n'a pas de statut officiel. Nous en donnons cependant, section 4, un aperçu[3].

Au sein du niveau 1, on distingue les interfaces du noyau (*DOM Core*) et les interfaces HTML ; au sein des interfaces du noyau on distingue encore les interfaces « fondamentales » et les interfaces « étendues ». Nous n'insistons pas sur ces distinctions, mais signalons simplement ici que pour être conforme au DOM une implémentation doit prendre en compte au moins les interfaces « fondamentales » du noyau.

Ajoutons enfin, que pour des raisons de neutralité vis-à-vis des langages de programmation, les interfaces DOM sont décrites en IDL (*Interface Definition Language*), le langage de description d'interfaces défini par l'*Object Management Group* et utilisé par la communauté Corba[4].

3. Exemples d'implémentation

Comme exemple illustrant les implémentations du DOM, nous présentons d'abord un parcours classique en profondeur permettant d'examiner tous les nœuds d'un arbre DOM. Ce type de parcours est utile dans de nombreuses situations et peut être adapté pour effectuer des traitements plus intéressants.

Nous donnons ensuite un petit exemple d'utilisation des *factory methods* du DOM, c'est-à-dire des méthodes permettant de créer à la volée des nœuds de type élément, attribut, etc. On peut ainsi créer des documents XML en mémoire qui, si nécessaire, peuvent ensuite être enregistrés sur disque sous forme de texte balisé.

3.1. Description des programmes d'exemple

Dans les exemples d'implémentation présentés ci-desssous, l'algorithme d'exploration est le même quel que soit le langage utilisé. On commence par obtenir une ré-

3. Il faut ajouter qu'il n'existe pas encore d'implémentations du niveau 2. Les exemples que nous donnons à la section 3 correspondent au niveau 1 du DOM.

4. On trouve cependant dans les annexes de la spécification des traductions de IDL en Java et en ECMAScript (la version normalisée de JavaScript).

férence sur la racine de l'arbre DOM correspondant au document de la figure 3. On invoque ensuite une méthode de traversée à qui l'on passe cette racine comme argument. La méthode imprime le type (attribut `nodeType`), le nom (attribut `nodeName`) et la valeur (attribut `nodeValue`) du nœud passé en argument puis s'invoque récursivement sur les fils de ce nœud s'ils existent. Ces attributs appartiennent à l'interface `Node` dont nous avons donné une rapide description à la section 2.

En ce qui concerne les exemples présentant les méthodes de l'interface `Document` qui permettent de créer des nœuds de type `Element`, `Attribute`, `Text`, `CDATA`, etc., on commence également par obtenir une référence sur la racine de l'arbre DOM correspondant au document de la figure 3. À partir de cette référence initiale, on obtient une nouvelle référence, cette fois sur l'élément racine du document XML. On invoque alors des méthodes permettant de créer un nouvel élément nommé « annexe » et de l'attacher sous la racine. On ajoute ensuite un attribut à l'élément racine pour indiquer qu'une modification a été apportée au document.

3.2. Implémentation Java

IBM a mis dans le domaine public un parseur XML nommé xml4j [11] qui est très complet et écrit en Java. Ce parseur valide les documents de façon précise et supporte le DOM au travers un ensemble de classes appartenant au paquetage `org.w3c.dom`. Plusieurs applications intéressantes ont été écrites en s'appuyant sur ce parseur, notamment un processeur XSL très complet nommé LotusXSL. Utilisée côté serveur, une implémentation Java du DOM peut servir à générer une page HTML à partir d'un document XML, en utilisant une technique de parcours de l'arbre DOM similaire à celle qui est exposée dans l'exemple ci-dessous. Utilisée côté client, une implémentation Java du DOM peut servir à générer des composants graphiques Java permettant d'afficher ou de manipuler le document XML sur le poste utilisateur. Les exemples Java donnés dans cet article ont été écrits avec les versions 1.1.14 et 2.0 du parseur xml4j[5].

3.2.1. Parcours de l'arbre DOM

On définit une classe publique dont la fonction `main()` construit un objet `Parser`. La méthode `readStream()` du parseur lit le document XML `rapport.xml` et renvoie une référence sur la racine de l'arbre DOM.

```
import com.ibm.xml.parser.*;
import org.w3c.dom.*;
import java.io.*;
```

5. Il existe d'autres implémentations intéressantes du DOM en Java, notamment celle proposée récemment par Sun dans le cadre du projet *Project X* [12]. L'implémentation IBM nous semble cependant être pour l'instant l'une des plus fidèles à la spécification DOM.

```
public class DOMDisplay {
    public static void main (String args[]) {
         String fileName="rapport.xml";
         InputStream is = null ;
         try {
               is = new FileInputStream(fileName);
         }
         catch (FileNotFoundException notFound) {
                System.err.println(notFound);
                System.exit(0);
         }
          Parser parser = new Parser(fileName);
          Document doc = parser.readStream(is);
          DOMDisplay.traverse(doc);
    }
```

On passe ensuite cette référence à la fonction de traversée, une fonction statique définie comme suit (on aurait aussi pu spécialiser la classe du parseur comme cela est fait dans l'exemple d'implémentation Perl donné plus loin) :

```
public static void traverse(Node node) {
       System.out.println(node.getNodeType() +":" + node.getNodeName()
       +":"+node.getNodeValue());
       if (node.hasChildNodes()) {
            NodeList childNodes  = node.getChildNodes();
            int size = childNodes.getLength();
            for (int i = 0; i < size; i++) {
              DOMDisplay.traverse(childNodes.item(i));
            }
       }
}
```

Cette fonction affiche des informations sur le nœud en se basant sur les méthodes comme `getNodeName()`, `getNodeType()` et `getNodeValue()` définies dans l'interface `Node`. Elle teste ensuite si le nœud passé en argument a des fils et, dans l'affirmative, utilise la méthode `getChildNodes()` pour récupérer un objet implémentant l'interface `NodeList` et contenant la liste des fils du nœud passé en argument. Le parcours se poursuit en parcourant cette liste.

Quand on exécute ce programme, on obtient l'affichage suivant :

```
9:#document:null
8:#comment: Ceci est un exemple de document XML
1:rapport:null
3:#text:
```

```
1:titre:null
3:#text:Utilisation pratique du DOM
3:#text:

1:texte:null
3:#text:Un exemple de contenu.
3:#text:
```

Dans la spécification DOM, la constante 9 correspond à un nœud `DOCUMENT_NODE`, la constante 8 à un nœud `COMMENT_NODE`, la constante 1 à un nœud `ELEMENT_NODE`, etc. Notons que, curieusement, le nœud `PROCESSING_INSTRUCTION_NODE` qui correspondant à `<?xml version="1.0" ?>` n'est pas indiqué par le parseur. On voit aussi que, par défaut, le parseur conserve tous les retours à la ligne présents dans le fichier XML. Par exemple, la première occurrence d'un nœud de type Texte, signalée par `3:#text:`, correspond à la ligne qui sépare la balise `<rapport>` de la balise `<titre>`. Ce type de retour à la ligne, ajouté pour rendre le balisage plus lisible, n'est pas pris en compte de la même manière par toutes les implémentations. Enfin, si l'on exécute le programme en ajoutant au début du fichier XML une référence à une DTD externe, on obtient la ligne supplémentaire `10:rapport:null` où 10 correspond bien à un `DOCUMENT_TYPE_NODE`.

3.2.2. Création de nœuds

Comme dans l'exemple précédent, on commence par obtenir une référence sur la racine de la représentation. À partir de cette référence initiale, on obtient une référence sur la racine du document XML proprement dite (c'est-à-dire le nœud correspondant à l'élément XML `rapport`) par la méthode `getDocumentElement()`. On crée ensuite un élément `appendixElement` en utilisant la *factory method* `createElement()`. Cet élément est alors ajouté comme fils du nœud correspondant à l'élément `rapport` par la méthode `appendChild()`. À la fin on ajoute un attribut au nœud `rapport`.

```
Parser parser = new Parser(filename);
Document doc = parser.readStream(is);
Element root = (Element)doc.getDocumentElement();
Element appendixElement = doc.createElement("annexe");
root.appendChild(appendixElement);
root.setAttribute("date-modification", new Date().toString());
```

Concernant les aspects de validation, il est important de signaler que l'insertion de l'élément `annexe` comme fils de l'élément `rapport` est acceptée même en présence d'une DTD qui n'autorise pas cette insertion. L'implémentation IBM donne cependant accès à des classes modélisant la DTD qui accompagne éventuellement le document. S'il veut être sûr de ne construire que des documents valides par rapport à cette DTD initiale, le programmeur peut consulter ces classes préalablement à l'utilisation de toute *factory method*.

3.3. Implémentation ECMAScript/JScript

À l'heure où nous écrivons cet article, Microsoft Explorer 5 est le seul navigateur grand public à supporter complètement le DOM via un langage de script (plus précisément le langage JScript très proche de ECMAScript auquel fait référence la spécification DOM). La prochaine version de Netscape supportera également bientôt le DOM dans un langage proche d'ECMAScript, mais n'est pas encore stabilisée[6]. Les exemples que nous donnons dans cette section correspondent à des traitements DOM côté client dans le cadre du navigateur. Cependant, comme dans le cas du langage Java, les traitements DOM écrits en JScript pourraient tout aussi bien être effectués côté serveur. Précisons enfin que le parseur intégré dans Microsoft Explorer 5 est un parseur validant en présence d'une DTD.

3.3.1. Parcours de l'arbre DOM

Comme dans le cas de l'implémentation Java présentée ci-dessus, on commence par charger le document XML et obtenir une référence sur la racine de l'arbre DOM. Dans Microsoft Explorer 5, une des façons de parvenir à ce résultat est d'insérer au début d'un document HTML un élément réservé nommé `XML`[7].

Quand le navigateur rencontre la balise `<XML>`, ceci provoque le chargement en mémoire du document référencé par l'attribut `SRC` et la construction de sa représentation DOM. La racine de l'arbre DOM correspond à la valeur de l'attribut `ID` de l'élément `XML` (ici «`doc` »). On peut ensuite vérifier dans un élément `SCRIPT` si le chargement s'est bien passé et, si c'est le cas, déclencher le parcours en invoquant la fonction `traverse()`[8]. Le code complet de la page HTML est donné ci-dessous.

```
<HTML>
<XML  ID="doc" SRC="rapport.xml"></XML>
<BODY>
<SCRIPT>
 if(doc.parseError.reason!="")
      {alert(doc.parseError.reason + "  " + doc.parseError.line );}
 else {traverse(doc);}

 function traverse(node) {
    document.write(node.nodeType + ":" + node.nodeName +
                                   ":" + node.nodeValue);
    document.writeln("<BR>");
    if (node.hasChildNodes()) {
       var x=node.childNodes;
```

6. Pour des informations sur l'état d'avancement de l'implémentation du DOM dans Netscape voir [3].
7. Ce mécanisme de chargement connu sous le nom de « XML island » est spécifique à Microsoft.
8. En cas d'échec l'attribut `parseError`, spécifique à Microsoft, affiche un message d'erreur.

```
       var size =x.length;
          for(var i=0;i<size;i++) {
             traverse(x(i));
          }
    }
 }
</SCRIPT>
</BODY>
</HTML>
```

Quand on exécute ce programme, on obtient l'affichage suivant dans le navigateur :

```
9:#document:null
7:xml:version="1.0"
8:#comment: Ceci est un exemple de document XML
1:rapport:null
1:titre:null
3:#text:Utilisation pratique du DOM
1:texte:null
3:#text:Un exemple de contenu
```

On peut remarquer que, à la différence de ce qui se passe dans l'implémentation Java, l'instruction de traitement initiale est signalée par un nœud de type `PI_NODE` et que par ailleurs les retours à la ligne entre les balises ont été filtrés.

3.3.2. Création de nœuds

La création de nœuds en ECMAScript s'effectue de la même façon qu'en Java. Notons seulement que les méthodes du DOM sont parfois traduites sous forme de propriétés, comme par exemple, ici, `documentElement` qui n'est pas suivi d'une parenthèse.

```
var root = doc.documentElement;
appendixElement = doc.createElement("annexe");
root.appendChild(appendixElement);
root.setAttribute("date-modification",
new Date().getTime());
```

3.4. Implémentation Perl

Pour le traitement de documents XML (l'analyse, la production et l'extraction), Perl[9] offre de nombreuses ressources. Parmi toutes celles disponibles sur le CPAN (*Comprehensive Perl Archive Network*), mentionnons :

9. La version la plus récente de Perl (5.005) est UTF-8. Les chaînes de caractères peuvent être UTF-8 et ces caractères peuvent être également utilisés dans les expressions régulières.

- `XML::Parser` qui est l'interface Perl avec le parseur non-validant « Expat » de James Clark.
- `XML::DOM` est un module qui s'appuie sur `XML::Parser` pour proposer une API conforme au niveau 1 du DOM. Le module ajoute un certain nombre de méthodes à celles définies dans la spécification du niveau 1 du DOM (elles permettent, par exemple, d'éditer la section du `DOCTYPE` et ajouter des attributs, des notations, des entités références, etc.).
- `XML::XQL` s'appuie sur le DOM pour effectuer des extractions d'informations. Les réponses sont des nœuds `XML::DOM`. Les réponses peuvent donc être traitées au moyen de l'API du DOM.
- `XML::Grove` à un objectif similaire au DOM : proposer un ensemble de méthodes pour le parcours et l'édition d'un document représenté par un arbre. Ce module ne se conforme pas à l'API du DOM et permet au programmeur de développer des application à base d'itérateurs et de visiteurs.

En termes d'architecture, l'implémentation Perl utilisée ci-dessous peut être un outil de choix pour des traitements DOM effectués dans le cadre de scripts CGI.

3.4.1. Parcours de l'arbre DOM

Dans l'exemple qui suit, on commence par définir la classe `XML::DOM::Displayer` qui hérite de la classe `XML::DOM::Parser`. La classe `XML::DOM::Displayer` comporte deux nouvelles méthodes `traverse()` et `trace()` qui implémentent le parcours de l'arbre DOM et l'affichage des nœuds. En présence du pragma `use strict`, nous devons déclarer toutes les variables, ce que nous avons fait au moyen de l'instruction `my` qui déclare des variables à portée lexicales. Perl définit deux contextes d'appel des procédures : le contexte liste et le contexte scalaire. Dans notre exemple `node->getChildNodes` est placé dans un contexte liste et retourne de ce fait directement la liste des nœuds fils de `$node`.

On remarquera que, comme dans les autres langages, l'on obtient une référence sur la racine de l'arbre DOM en utilisant une méthode spécifique à l'implémentation, ici `parsefile()`. Cet aspect n'est en effet pas normalisé par le DOM.

```
use strict;
package XML::DOM::Displayer;
use XML::DOM;
@XML::DOM::Displayer::ISA = ('XML::DOM::Parser');
sub trace {
  my $self = shift;
  print "@_\n"
}
```

```
sub traverse {
  my ($self, $node) = @_;
  $self->trace($node->getNodeTypeName, ":", $node->getNodeName, ":",
  $node->getNodeValue);
  foreach my $kid ($node->getChildNodes) {
    $self->traverse($kid);
  }
}

my $displayer = XML::DOM::Displayer->new();
my $doc = $displayer->parsefile("rapport.xml");
$displayer->traverse($doc);
```

En ce qui concerne la façon d'itérer sur les fils d'un élément, il aurait été plus conforme au DOM d'utiliser la boucle ci-dessous :

```
my $childNodes = $node->getChildNodes;
 if ($childNodes) {
        my $number_of_nodes = $childNodes->getLength;
        for (my $i = 0; $i < $number_of_nodes; $i++) {
        $self->traverse($childNodes->item(i));
        }
  }
```

Mais cette forme d'écriture n'est pas très « perlienne », et cet exemple illustre la tension qui existe nécessairement entre le style de programmation imposé par le DOM et les caractéristiques des langages d'implémentation. Nous aurions pu faire la même remarque à propos de l'exemple correspondant en ECMAScript qui n'utilise pas non plus la méthode `item()` recommandée par le DOM.

Le programme ci-dessus affiche la sortie :

```
DOCUMENT_NODE : #document :
COMMENT_NODE : #comment :  Ceci est un exemple de document XML
ELEMENT_NODE : rapport :
TEXT_NODE : #text :

ELEMENT_NODE : titre :
TEXT_NODE : #text : Utilisation pratique du DOM
TEXT_NODE : #text :

ELEMENT_NODE : texte :
TEXT_NODE : #text : Un exemple de contenu
TEXT_NODE : #text :
```

On retrouve en clair les types de nœuds de la spécification alors que, dans l'implémentation Java donnée plus haut, seule la valeur numérique de ces types est affichée. Comme dans le cas de l'implémentation en Java le nœud `PI_NODE` est omis et tous les retours à la ligne génèrent un nœud de type Texte.

3.4.2. Création de nœuds

La création de noeuds en Perl ne suscite pas d'observations particulières. Remarquons simplement que le programme comporte plusieurs commentaires. En l'absence d'informations de typage explicite des variables en Perl, ces commentaires sont utiles pour rappeler au programmeur quel type précis d'interface DOM il est en train d'utiliser.

```
use XML::DOM;
use strict;
my $parser = new XML::DOM::Parser();
# XML::DOM::Document instance
my $doc = $parser->parsefile("rapport.xml");
# XML::DOM::Element instance
my $root =  $doc->getDocumentElement();
my $appendix = $doc->createElement("annexe");
$root->appendChild($appendix);
$root->setAttribute("date-modification", scalar(localtime()));
```

Ici, `scalar(localtime())` force un contexte scalaire et retourne la date de modification sous la forme d'une chaîne de caractères.

3.5. Conformité et performances des implémentations utilisées

Faute de place, nous ne pouvons procéder ici à une évaluation détaillée des diverses implémentations en termes de conformité et de performance. Sur le premier point, notons cependant que l'implémentation Java, riche d'informations de typage, est celle qui est la plus facile à comparer à la spécification qui, comme nous l'avons dit à la section 2, utilise des descriptions en IDL plus proches de la syntaxe Java que de celles de Perl ou ECMAScript[10]. Globalement, quels que soient que soient les langages considérés, la plupart des implémentations du DOM disponibles semblent fidèles à la spécification. La mise en parallèle que nous avons effectuée semble montrer qu'il est assez facile de comprendre un programme utilisant le DOM même si ce programme est écrit dans un langage que l'on n'utilise pas couramment. Certains

10. Dans le cas de ce dernier langage, on notera que les informations sur le nom, le type et la valeur de chaque nœud sont obtenus via des attributs et non des méthodes, ce qui est parfois perturbant.

TABLE 1 – *Tests de performance*

Taille du fichier (ko)	xml4j 1.1.14 (s)	xml4j 2.0 (s)	`XML::DOM` 1.19 (s)
10	0,5	0,5	0,5
100	4,1	2,6	3
1000	43,3	26,4	36

La première colonne donne la taille du fichier XML en kilo-octets, alors que les autres colonnes montrent les temps de traitement pour chaque analyseur exprimés en secondes.

pourront même trouver que le DOM « réussit trop bien » en imposant un style de programmation uniforme quel que soit le langage considéré. C'est le prix à payer quand on utilise une interface normalisée, mais on peut penser que des évolutions du DOM permettront à chaque langage de mieux utiliser son style propre.

En ce qui concerne les performances, nous pouvons donner à titre indicatif quelques ordres de grandeur concernant le chargement d'un fichier XML et la construction de sa représentation DOM. La table 1 contient le résultat de tests effectués avec, d'une part, l'implémentation Java du DOM diffusée par IBM (dans ses versions 1.1.14 et 2.0) et, d'autre part, le module Perl `XML::DOM` dans sa version 1.19. Les test de l'implémentation Java ont été effectués sur un ordinateur PC AMD K6 à 233 MHZ sous Linux RedHat 5.1 et doté de 64 Mo, en utilisant le JDK 1.1.6 (portage Linux). Les tests Perl ont eux été effectués sur une Sparc ultra sous Solaris 2.5 avec Perl 5.005.

Dans le cas de Java, on peut constater qu'on obtient un gain important en passant simplement d'une version à l'autre d'une même implémentation. Globalement, on peut dire que, sur des configurations matérielles modestes et avec des documents très volumineux, l'utilisation du DOM peut être pénalisante pour des applications interactives. Ces considérations de performance ont, bien sûr, une influence directe sur le choix de l'architecture à adopter (traitements DOM effectués côté client ou côté serveur).

4. Limites actuelles et perspectives d'évolution

4.1. Performance

Nous avons déjà signalé des problèmes de performance : le temps de construction de l'arbre et la mémoire consommée par cette opération. Pour résoudre ces problèmes les solutions proposées tournent autour d'un stockage persistant d'une représentation dérivée de l'arbre. On utilise un format qui permet de construire l'arbre en mémoire plus rapidement que la représentation XML et on indexe l'arbre de manière à

pouvoir retrouver rapidement ses constituants sans être obligé de le charger intégralement.

Par ailleurs, si le DOM est déjà d'un grand intérêt pratique, il souffre encore de l'absence d'un certain nombre de fonctionnalités, ce qui incite la plupart des implémenteurs à développer leurs propres extensions. L'implémentation d'IBM propose ainsi un certain nombre de classes et de méthodes supplémentaires permettant d'efffectuer des opérations qui ne sont pas possibles dans le cadre du DOM stricto sensu. À titre d'exemple, les implémentations Perl et Java présentées dans cet article définissent de nombreux types de nœuds qui n'existent pas dans le DOM comme `AttDef`, `AttlistDecl`, `ElementDecl`, `XMLDecl`, etc. ainsi que de nombreuses méthodes supplémentaires. Ces extensions spécifiques sont souvent utiles et il est tentant de les utiliser, mais ce faisant, on va à l'encontre de l'esprit du DOM : les applications redeviennent dépendantes des parseurs sous-jacents.

4.2. Validation

Des problèmes de validation sont également à signaler. Les implémentations testées tolèrent l'ajout à un élément d'un fils ou d'un attribut qui n'est pas autorisé par la DTD, et la spécification ne dit pas très clairement le comportement qui doit être adopté par les implémentations dans ce cas.

La situation est cependant susceptible de s'améliorer sur certains des points évoqués ci-dessus avec la parution de la spécification DOM niveau~2. À la section~2 nous avons donné des indications sur l'organisation de la spécification DOM et sa division en niveaux. Ce document, actuellement disponible sous forme d'un *working draft*~[2], est destiné à compléter les interfaces définies dans le noyau du niveau 1. Il traite notamment des aspects suivants :

- gestion des espaces de noms ;
- utilisation des feuilles de style ;
- gestion des événements ;
- système d'adressage (interface `Range`).

Les deux derniers points de la liste peuvent clairement servir de support au développement d'éditeurs structurés. Quand les interfaces du niveau 2 seront effectivement implémentées, on disposera d'un ensemble d'interfaces normalisées très complet pour la manipulation des documents structurés.

Signalons en conclusion que le DOM recouvre fonctionnellement certaines autres normes « satellites » de XML, même si ces dernières ont officiellement une vocation très différente. Par exemple, le DOM peut, comme XSL, permettre d'effectuer des transformations d'arbre complexes. Dans le cas de XSL, la transformation s'effectue via un langage déclaratif, alors que le DOM relève d'une approche procédurale.

Bibliographie

[1] World Wide Web Consortium, Vidur APPARAO *et al.* (rédacteurs). *Document Object Model (DOM) Level 1 Specification. W3C Recommendation 1 October, 1998.*
`http://www.w3.org/TR/REC-DOM-Level-1/`.

[2] World Wide Web Consortium, Vidur APPARAO *et al.* (rédacteurs). *Document Object Model (DOM) Level 2 Specification. W3C Working Draft 04 March, 1999.*
`http://www.w3.org/TR/REC-DOM-Level-1/`.

[3] Vidur APPARAO et Tom PIXLEY. *The NGLayout Document Object Model (DOM) Roadmap.* Mozilla Organization, 1998.
`http://www.mozilla.org/newlayout/dom-roadmap.html`.

[4] World Wide Web Consortium, Tim BRAY, Jean PAOLI, et C. M. SPERBERG-MCQUEEN (rédacteurs). *Extensible Markup Language (XML) 1.0. World Wide Web Consortium Recommendation 10-February-1998.*
`http://www.w3.org/TR/REC-xml-19980210.html`.

[5] Elaine BRENNAN. « The DOM for Non-Programmers ». *TAG*, 11(10):1–4, October 1998.

[6] World Wide Web Consortium, James CLARK et Stephen DEACH (rédacteurs). *Extensible Stylesheet Language (XSL). World Wide Web Consortium Working Draft 16-December-1998.*
`http://www.w3.org/TR/WD-xsl`.

[7] World Wide Web Consortium, Steve DEROSE et Eve MALER (rédacteurs). *XML Linking Language (XLink). World Wide Web Consortium Working Draft 3-March-1998.*
`http://www.w3.org/TR/WD-xlink`.

[8] World Wide Web Consortium, Steve DEROSE et Eve MALER (rédacteurs). *XML Pointer Language (XPointer). World Wide Web Consortium Working Draft 03-March-1998.*
`http://www.w3.org/TR/1998/WD-xptr`.

[9] Document Object Model Requirements.
`http://www.w3.org/TR/WD-DOM/requirements-19980318/`.

[10] FAQ Perl XML. `http://www.perlxml.com/perl-xml-faq.html`.

[11] IBM Alphaworks. `http://www.alphaWorks.ibm.com/tech/xml4j`.

[12] Java Project X.
`http://developer.java.sun.com/developer/earlyAccess/xml/index.html`.