**Université d'Avignon et des Pays du Vaucluse**
**IUP Génie Mathématique et Informatique**
**DESS Traitement Automatique de l'information sur Internet**

# STRUCTURATION ET COMPOSITION DE DOCUMENTS

**Notes de cours :**

**L. Médini,**

**Septembre – octobre 2001.**

# Introduction (transparents issus du cours)



## Plan du cours

- Introduction aux techniques de composition documentaire
  - Objectifs du cours
  - Notion de document
  - Cycle de vie de l'information
  - Aperçu des langages de structuration et de composition
- Structuration des données : XML
  - Spécifications
  - Applications
- Des données aux documents : XSL
  - XPATH
  - XSLT





## Introduction

- Objectifs du cours
  - But
    - Génération de documents à partir de structures de données prédéfinies.
  - Orientation du cours
    - Traitement automatique de l'information textuelle
    - Utilisation des « NTIC » (technologies W3)
  - Technique prédominante
    - Documents Virtuels Personnalisables (DVP)

## Introduction

- Le document en tant que flux informationnel
  - Communication = échange d'information
  - Information
    - Définition dynamique (Shannon, 1948)

      Flux circulant entre un émetteur et un récepteur

      Exemples : signal électrique, son
    - Définition statique (sémiologique, De Saussure, 1916)

      Caractéristiques physiques : tension, amplitude

      Contenu sémantique : « élément de connaissance susceptible d'être représenté à l'aide de conventions pour être conservé, traité ou communiqué » (Larrousse)





## Introduction

- Le document en tant que flux informationnel
  - Cas des opérateurs humains dans les entreprises
    - Entités informationnelles échangées = documents
      - Le document est un tout
      - Le document comporte des sous-parties
    - Caractéristiques physiques
      - Critères de forme : support, format, nb de pages…
      - Critères de fonds : mots-clés, résumé, domaine

## Introduction

- Le document en tant que flux informationnel
  - Parties du document (méta-données)
    - Découpage en unités informationnelles
    - Organisation sémantique des parties
  - Contenu du document (données)
    - Information à transmettre
  - Présentation du document (informations de formatage)
    - Notion de page (physique ou d'écran)
    - Traitées différemment par l'outil d'accès à l'information utilisé





## Introduction

- Cycle de vie de l'information dans les entreprises
  - Production (sc. de l'information et de la communication)
  - Stockage et transformations (ingénierie)
  - Accès (interaction homme-machine)

## Introduction

- Production documentaire
  - Type d'information produite : information textuelle
  - Outils : éditeurs
    - Texte (ASCII)
    - WYSIWYG
    - Mixtes
  - Destinataire : opérateur humain

    (à l'inverse d'un programme ou d'un fichier de données)





## Introduction

- Stockage et transformations
  - Format : dépendant de l'information produite
  - Outils : appropriés (serveurs, BD, filtres, parsers…)
  - Problème des conversions de formats

    ® utilité d'un format pivot

## Introduction

- Accès à l'information
  - Opérateur : individu ayant un besoin d'information
  - Moyens : stratégies d'accès à l'information
    - Stratégies analytiques : recherche d'information
    - Stratégies de parcours : navigation hypermédia
    - Stratégies intermédiaires…
  - Outils :
    - « Visualisateurs »
      - Outils spécifiques aux formats (.doc, .pdf…)
      - Navigateurs W3
      - Outils génériques (Alchemy)
    - Moteurs d'indexation et de recherche / navigateurs





## Introduction

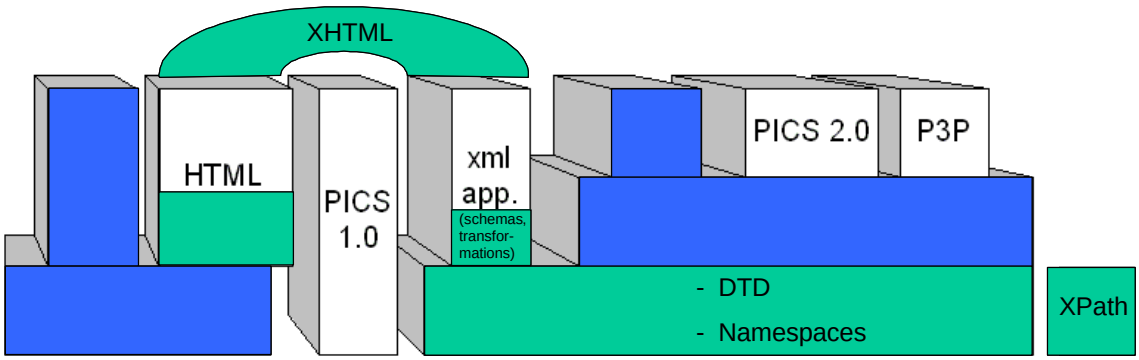
- Objectif de la production d'information
  - Rendre l'information produite accessible

    Prise en compte des techniques d'accès à l'information
    - indexation par des moteurs de recherche
    - Mise au format hypertexte
  - Rendre l'information produite compréhensible

    Facilitation de l'interprétation
    - Structuration des documents (plans, vue générale)
    - Composition « lisible » des documents

## Historique et positionnement des principaux langages de structuration

La figure suivante est une actualisation d'un schéma proposé sur le site du W3C datant d'octobre 1997 et présentant les différents langages à balises et leurs applications. En vert (gris clair) figurent les éléments présentés dans ce cours et en bleu (gris plus sombre) ceux présentés lors des exposés.




Les paragraphes suivants présentent les principales caractéristiques des langages à balises de cette figure et des applications de ces langages.

### *Langages à balises*

***SGML :***

- début des spécifications dans les années 60,
- norme ISO depuis 1986 ; amendement en 88 (ISO 8879),
- puissant mais complexe (taille des spécifications > 500 pages).

***HTML :*** langage de visualisation

- 1ère version : 1992,
- Version 4.01 : décembre 1997.

***XML :***.

- Syntaxe et documents XML bien formés : version 1.0 : février 98 ; deuxième édition : octobre 2000 (mais toujours V1.0)
- Nombreuses extensions et applications (présentées plus loin),

***XHTML :***

Version de HTML respectant la syntaxe XML : 1999

Il existe également d'autres langages à balises (CompactHTML, WirelessML, MathML, Scalable Vector Graphics…) qui ne sont pas mentionnés sur cette figure et qui sont destinés à

des applications spécifiques. Pour ces langages comme pour toutes les applications en rapport avec le web, le site du W3C (http://www.w3.org) propose en ligne soit les recommandations ayant valeurs normatives, soit au moins des informations et des liens vers les sites concernés.

### *Applications*

***RDF :*** Resource Description Framework : langage de description générique des métadonnées, permettant diverses opérations telles que le stockage ou l'échange, indépendamment du type de données.

***PICS :*** Platform for Internet Content Selection : permet aux parents et aux enseignants de déterminer les contenus visibles par les enfants sur Internet. Utilisé dans les outils de filtrage.

***P3P :*** Platform for Privacy Preferences Project : projet d'aide au contôle par les utilisateurs des informations personnelles qu'ils fournissent sur les sites web qu'ils visitent.

# XML : Extensible Markup Language

## Définition

XML est un méta-langage de description des données, c'est-à-dire qu'il permet de définir des langages de description d'informations structurées, encore appelés langages de structuration, par opposition aux langages de transformation (i.e. de programmation). Par abus de langage, on dira souvent que XML est lui-même un langage de structuration, car les descriptions des données engendrées utilisent la syntaxe XML.

En soi, XML ne sert à rien (au sens applicatif du terme). En revanche, il est possible, à partir d'une de ces descriptions des données, d'utiliser les nombreuses applications de XML pour « faire quelque chose » avec ces données. Par exemple : affichage en HTML, traitements automatiques des données (extraction de données, tri , transformation…)

XML permet de faire apparaître clairement les données (l'information) dans une structure de données (méta-information).

En cela, XML est également indiqué pour l'échange des données entre les applications (format pivot). Il peut aussi être utilisé pour l'échange entre les individus, du fait qu'il est au format texte et que sa structure est « lisible ». Mais dans le cas d'informations textuelles, nous considérons que les individus échangent des documents mis en forme (données + méta-données + informations de formatage), plutôt que XML (données + méta-données uniquement).

## Positionnement de XML par rapport à SGML

XML est fondé sur SGML. Il découle d'une volonté de simplification des spécifications de ce langage, qui, pour mémoire, ont été rédigées entre les années 60 et 1986.

SGML est un méta-langage, c'est-à-dire un langage servant à décrire des langages à balises (ex : il a permis de décrire HTML, qui n'est qu'une des nombreuses applications de SGML).

En SGML, ces applications ne sont pas clairement identifiées. On s'y réfère uniquement par le nom de leur DTD. En XML, la notion d'espaces de noms permet d'identifier les applications. Ces applications sont appelées *langages de balisage* depuis XML.

SGML s'appuie sur les DTD pour définir des *modèles d'information*. La DTD est le cœur s'une application SGML. La modification d'une DTD entraîne souvent celle des données qu'elle définit. D'où l'intérêt de porter une attention spéciale à la mise au point de la DTD.

SGML a été conçu pour durer. Il affirme prendre en compte les évolutions futures des techniques et des logiciels. Cette évolutivité a été mise en œuvre au prix d'une grande complexité.

Finalement, SGML est la solution industrielle : chère et compliquée, mais extrêmement puissante. Par conséquent, même s'il pourrait servir de format d'échange de données pour le web, SGML n'est pas utilisé (à quelques rares exceptions près), car il nécessite un investissement trop lourd en temps, en outils et en formation.

C'est en partie pour cela qu'a été créé XML. XML est un sous-ensemble de SGML, possédant les mêmes objectifs que SGML : la structuration des données, mais de manière plus simple et avec des spécifications plus compactes (44 pages pour la version française de la recommandation XML contre environ 500 pour SGML)[1]. Le premier document de travail du W3C date de novembre 1996, et la première édition de la version 1.0 a été publiée le 10 février 1998. La deuxième (et dernière) édition de cette recommandation date du 2 octobre 2000. Elle est disponible à l'adresse : http://www.w3.org/TR/REC-xml.

## Les différents composants de XML

Nous avons vu que XML est un méta-langage de balisage. Plusieurs « entités » lui sont associées pour permettre la mise au point, la lecture ou la visualisation de l'information structurée.

**XML 1.0** est la spécification de la syntaxe du langage de structuration utilisé pour la description des données. Cette spécification permet d'écrire des *documents XML bien formés*. Ces documents XML comprennent l'information et la méta-information à structurer.

**Les *déclarations de type de document* (DTD) ou les *schémas XML*** permettent de définir la structure de l'information décrite de façon plus ou moins approfondie. Lorsqu'un document XML est associé à une DTD ou à un schéma XML et qu'il est conforme à la description de l'information indiquée, on dit qu'il est *valide*.

**Un processeur (ou *parser*) XML** est une application qui permet d'analyser et de traiter l'information contenu dans les fichiers XML. Il existe des parsers validants et non validants (on dit aussi validateurs et non validateurs), selon qu'ils analysent ou non la conformité du document avec une DTD ou un schéma XML éventuellement associés. La simplicité de la spécification d'XML par rapport à SGML rend également l'écriture d'un parser plus simple pour XML que pour son prédécesseur. La plupart des parsers sont écrits en C/ C++ ou en Java. Pour des raisons pratiques, nous utiliserons dans les TD le parser MSXML de MicroSoft™ qui est inclus dans Internet Explorer depuis la version 5.00. Ce parser permet d'analyser et d'afficher directement les documents XML dans le navigateur, même sans qu'aucune feuille de style ne leur soit associée.

**Le modèle objet de document (*DOM*)** permet à l'application d'accéder aux données d'un document XML. Il s'interface entre le parser et l'application. Les spécifications du DOM sont définies par niveaux. Celles des niveaux 1 et 2 sont définies, et le W3C travaille actuellement sur DOM niveau 3. Le DOM définit des *interfaces* qui permettent d'accéder aux objets (éléments) d'un document XML. Une interface propose des propriétés et des méthodes pour chaque type d'élément. Il existe un DOM commun pour toutes les applications, le *DOM Core*) et des extensions : DOM XML, DOM HTML[2], DOM CSS… La combinaison langages de scripts / DOM / HTML constitue le Dynamic HTML (*DHTML*).

---

[1] Par conséquent tout ce qui respecte les spécifications d'XML respecte celles de SGML, mais la réciproque n'est pas vraie. Par exemple, SGML définit la notion de balise de façon beaucoup plus large que XML : elles sont en général encadrées par des chevrons ("<" et ">"), mais cela n'est pas imposé dans la spécification du langage. Il pourrait s'agir d'accolades ("{" et "}") comme de toute autre paire de caractères. XML n'accepte pas autre chose que les chevrons pour encadrer les balises.

[2] Le DOM HTML est utilisé pour accéder (et éventuellement modifier) dynamiquement les éléments d'un document HTML. L'implémentation du DOM est cependant très dépendante des navigateurs (ex : *document.balise* sous Netscape™ ; *document.all.balise* sous IE).

**L'API simplifiée pour XML (Simple API for XML), ou *SAX*** est une alternative au DOM. Elle est adaptée à l'analyse de documents XML volumineux et est plus complexe à mettre en place, car elle nécessite la connaissance de la programmation en Java et l'installation d'un parser (xp) et d'un environnement de développement (jdk) spécifiques. SAX ne sera pas utilisée dans ce cours.

**Les espaces de noms (*namespaces*)** permettent de définir des « catégories » de vocabulaires XML. Ils sont notamment utilisés pour éviter les conflits entre des termes identiques ayant des significations différentes dans des domaines différents. L'utilisation d'espaces de noms permet de traiter ces termes sans connaître la structuration ni les méta-données du document.
Par exemple, sur une carte de visite, le titre d'une personne correspond à une fonction professionnelle (docteur, ingénieur…) ; pour un document, le titre est la désignation générique du document. Un document de type carte de visite peut comporter ces deux types de titres, qu'il est important d'arriver à différencier.
Il existe des « espaces de noms qualifiés » déterminés par la recommandation *Qname* du W3C.

***XLink* (XML Linking Language) et *XPointer* (XML Pointing Language)** permettent de définir des mécanismes de liens vers d'autres ressources pour XLink et vers des parties de documents pour XPointer. Le concept de lien avec XLink est plus puissant que celui d'HTML. XLink permet de définir des liens simples (comme en HTML), étendus (qui associent plusieurs ressources) ou des arcs (qui définissent les règles de passage entre les ressources). La syntaxe de XPointer pour XML est la même que celle utilisée en HTML : le nom du fichier est suivi du caractère '#' (dièse) et de l'identifiant de la ressource vers laquelle pointe le lien. Ces deux langages ne sont pas détaillés dans ce cours.

**Le langage de feuilles de styles associé à XML, *XSL*** (Extended StyleSheet Language) permet la mise en forme de documents XML. Il se compose des langages *XPATH*, qui permet la localisation des éléments et des parties dans un document XML, et *XSLT* (XSL Transformations), qui définit le format de sortie du document. L'association d'XSL à un document XML permet de compléter le couple données / méta-données défini dans le document en lui associant des informations de formatage. Il existe aussi un troisième langage, *XSL-Formatting Objects*, qui permet d'aller plus loin dans la composition documentaire, en décrivant notamment la structure physique des pages des documents générés. XSL-FO n'est pas détaillé dans ce cours.

## Description d'un document XML

### *Déclaration et instructions de traitement*

**Déclaration simple d'un document XML** : `<?xml version="1.0"?>` Une déclaration XML n'est *a priori* pas obligatoire, le serveur étant capable d'indiquer qu'il envoie du XML. Cependant, il est recommandé de l'inclure, notamment pour y faire figurer le numéro de version.
Remarque : si cette déclaration est incluse, l'attribut « `version` » doit nécessairement y figurer.

**Encodage** : `<?xml version="1.0" encoding="UTF-8"?>`. UTF-8 et UTF-16 sont les formats ASCII qui sont gérés en interne par tous les processeurs XML. Ces encodages sont détectés par défaut et n'ont donc pas besoin d'être spécifiés. En revanche, la présence de

caractères incompatibles avec l'un de ces deux formats sans précision du type d'encodage utilisé entraîne une erreur du processeur.

Remarque : d'autres encodages classiquement utilisés pour inclure des caractères français (accents, cédilles…) dans des documents XML sont « `windows-1252` », qui n'est pas accepté par tous les parsers, ou « `ISO-8859-1` », qui lui est très semblable et plus souvent reconnu. Certains éditeurs permettent également d'enregistrer du texte directement en unicode.

**SDD (Standalone Document Declaration)** : dans la déclaration `<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>`, le dernier attribut (standalone) permet de déterminer si le document est *autonome* (on trouve aussi *autodescriptif*). Si ce n'est pas le cas, c'est qu'il fait référence à une DTD (ou un schéma XML) externe. Par défaut, un document XML est considéré comme autonome, puisque la DTD peut être omise.

Remarque : les attributs version, encoding et standalone qui figurent dans la déclaration doivent y figurer dans cet ordre.

**Instructions de traitement** : d'une façon générale, les lignes encadrées par "<?" et "?>" sont appelées *instructions de traitement* (Processing Instructions, ou *PI*). Ces instructions n'ont aucune signification pour le document XML, et sont transmises directement à l'application par le parser pour être traitées par elle. La syntaxe générale d'une PI est : `<?Application_destinatrice Texte_de_l'instruction(avec ou sans espaces)?>`.

Remarque : officiellement, la déclaration XML n'est pas une instruction de traitement, car elle ne concerne pas l'application qui utilise le document XML, mais est uniquement destinée au parser. Un parser n'est donc pas obligé de la transmettre à l'application. En pratique, cette distinction n'a aucune incidence sur ce cours. Nous considérons ici la déclaration XML comme une instruction de traitement, puisqu'elle respecte la syntaxe de ces instructions.

### *Document bien formé*

Un document est dit bien formé s'il respecte la syntaxe XML. Il n'a pas besoin d'être conforme à un quelconque modèle de document.

**Un document XML** est composé d'un *prologue* et d'un *élément* (appelé élément racine).

**Un prologue** est composé éventuellement d'une déclaration XML et éventuellement d'une *déclaration de type de document*.

**Un élément** est composé soit d'une *balise ouvrante*, éventuellement d'un *contenu* et d'une *balise fermante*, soit d'une balise d'*élément vide*.

**Une balise ouvrante** est entourée de chevrons et contient un nom d'élément et éventuellement des attributs.

**Une balise fermante** est entourée de chevrons, contient un slash ('`/`') et un nom d'élément.

**Une balise d'élément vide** est entourée de chevrons, contient un nom d'élément, éventuellement des attributs et un slash.

**Un contenu** est composé de texte et / ou d'autres éléments appelés fils de l'élément courant. Le texte est appelé « données caractères analysables » ou PCDATA (pour Parsed Character DATA). Le contenu autorisé pour le PCDATA dépend du type d'encodage choisi.

Remarques :

- Le fait d'avoir un unique élément racine, décomposable en éléments fils, eux-mêmes décomposables, etc. définit une structure arborescente, assimilée à un modèle objet du document.
- L'inclusion du contenu entre les balises d'ouverture et de fermeture de l'élément ne permet pas le chevauchement de balises entre un élément père et un élément fils.
- Dans tous les cas, les caractères "<" (inférieur) et "&" (esperluète) sont interdits dans les contenus. On aura recours aux *entités* "`<`" et "`&`".
- L'usage de ">" (supérieur) ou des guillemets simples ou doubles peut également être perturbant. Dans ce cas, on a recours à &gt ;, &apos ; et &quot ;.
- Si l'on veut vraiment utiliser les caractères "<" ou "&", il est possible de définir une balise sous forme de zone de caractères non analysés, sous la forme : `<![CDATA [ texte comprenant des caractères interdits ]]>`. Ici, CDATA s'oppose à P(arsed)CDATA, qui autorise l'analyse des données textuelles.
- Le mode de traitement des espaces peut être défini grâce à l'attribut *xml-space*, qui peut prendre les valeurs *default* ou *preserve*.
- Par défaut, les éléments définis comme de type PCDATA conservent les espaces. Le mode de traitement des espaces pour les autres éléments dépend de l'application.

**Un nom d'élément** peut contenir des lettres, des chiffres, des modificateurs de lettres (si cela est autorisé par l'attribut « encoding » de la déclaration XML, et les caractères "`.`" (point), "`-`" (tiret), "_" (souligné) ou "`:`" (deux-points).

Remarques :

- Un nom doit commencer par une lettre ou par le caractère "_" et non par un chiffre ou un autre signe de ponctuation.
- Un nom ne peut pas commencer par la séquence de lettres x, m, l, quelle qu'en soit la casse.
- XML est sensible à la casse des noms d'éléments.
- Éviter les deux points dans le nom des éléments, car ils sont utilisés pour spécifier les espaces de noms (voir plus loin).

**Les attributs** possèdent un nom unique pour chaque élément et une valeur. La syntaxe employée en XML est tout d'abord le nom de l'attribut, puis le caractère "=" (égal), puis sa valeur entre guillemets simples ou doubles.

**Les commentaires** débutent par la chaîne « `<!--` » et se terminent par la chaîne « `-->` ». Ils ne peuvent être placés à l'intérieur d'une balise et ne sont pas obligatoirement transmis à l'application.

## Document valide

L'opération de validation est réalisée par le parser après l'analyse de la syntaxe (i.e. uniquement si le document est bien formé). Pour cela, il faut que le parser utilisé soit validant (et éventuellement que l'option de validation soit activée[3]). La validation d'un document peut se faire d'après une DTD ou d'après un schéma XML. Le principe d'une DTD est détaillé au paragraphe suivant. Les schémas constituent une application XML et font appel à la notion d'espace de noms. Ils sont présentés ultérieurement dans ce cours.

## Les DTD

Une DTD est une *définition de type de document*. On s'y réfère pour spécifier le modèle de données auquel appartient un document. Ce type de définition est hérité de SGML et permet de valider la structure du document. La spécification des DTD XML fait partie intégrante de la recommandation XML 1.0. Pour qu'un document soit validé par une DTD, il faut :

- qu'il contienne une (et une seule) *déclaration de type de document*,
- qu'il fasse référence, de façon interne ou externe, à une DTD décrivant sa structure.

**La déclaration de type de document** est de la forme :
`<!DOCTYPE Nom_de_l_élément_racine Type_de_source emplacement1 emplacement2 [sous-ensemble interne de DTD]>`, où :

- `Nom_de_l_element_racine` représente – contre toute attente – le nom de l'élément racine du document XML. En conséquence, il ne peut y avoir de DTD pour une partie de l'arborescence du document.
- Le type de source est associé aux mots-clé « `SYSTEM` » ou « `PUBLIC` ». `SYSTEM` correspond à une DTD employée ponctuellement pour décrire le document. Elle est associée à un ou deux emplacements indiqués sous forme d'URLs. Dans le cas de deux emplacements, le second est un « emplacement de secours », pour le cas où le premier est inaccessible. Une DTD déclarée `PUBLIC` est une DTD qui peut être partagée. Elle a donc une portée plus large qu'une DTD `SYSTEM`. Le mot-clé `SYSTEM` permet de mettre la DTD en cache et de la rendre disponible hors connexion. Pour cela, le premier emplacement indiqué est un URI, qui désigne pour le processeur l'emplacement local de la DTD.
- `[sous-ensemble interne de DTD]` désigne une éventuelle partie de la définition de la structure du document incluse dans celui-ci.

**La définition de type de document** est dérivée de SGML. Elle permet de décrire la structure du document dans un formalisme spécifique. Une DTD peut comporter un sous-ensemble externe et un sous-ensemble interne. Dans ce cas, c'est le sous-ensemble interne qui a priorité sur le sous-ensemble externe. Une seule DTD est toutefois autorisée par document.

Une DTD décrit quatre types d'éléments.

[3] Par exemple, MSXML n'effectue pas de validation de documents XML, lorsque ceux-ci sont affichés avec la feuille de style par défaut. Il existe un « flag » appelé validateOnParse qu'il faut positionner (via le DOM) pour forcer la validation.

- Les différentes entités auxquelles font référence les éléments du document, grâce au mot-clé `ENTITY`,
- les éventuelles notations, qui déclarent du contenu non-XML, comme des données graphiques ou binaires (mot-clé : `NOTATION`),
- les éléments du document XML, de façon arborescente, en partant de l'élément racine (mot-clé : `ELEMENT`),
- les attributs des différents éléments, sous forme de liste (`ATTLIST`).

**Une entité** peut être de deux types : générale ou paramètre.

- il existe deux types d'entités générales : analysables (internes ou externes à la DTD), ou non analysables (toujours situées dans un fichier externe à la DTD). Les entités analysables internes sont définies sous la forme `<!ENTITY nom "texte_de_remplacement" >`, où texte de remplacement est un texte bien formé quelconque, ne contenant pas de référence directe ou indirecte à cette entité.
- Une entité générale non analysable se trouve toujours dans un fichier ou une ressource externe. Elle est définie par forme `<!ENTITY nom SYSTEM "localisation" NDATA type_de_notation>` ou `<!ENTITY nom PUBLIC "localisation1" "localisation2" NDATA type_de_notation>`. Chaque NDATA doit correspondre au nom d'une déclaration de notation pour être valide.
- Les entités paramètres sont uniquement utilisées dans les DTD et doivent toujours être analysables. Elles sont définies sous la forme : `<!ENTITY % nom "texte_de_remplacement" >`.

Les références d'entités se font par &nom; pour les entités générales et %nom; pour les entités paramètres.

**Les notations** sont définies comme les entités, définies sous la forme : `<! nom SYSTEM "localisation" >` ou `<!NOTATION nom PUBLIC "localisation1" "localisation2" >`.

**Un élément** peut avoir cinq types de contenu :

- élément (il est composé uniquement d'éléments fils),
- `PCDATA` (du texte analysable),
- mixte (élément + texte),
- `ANY` (n'importe quel type de contenu XML bien formé) ou
- `EMPTY` ( ).

Dans les cas où il contient d'autres éléments ou du contenu mixte, ce contenu peut être décrit sous forme de liste entre parenthèses "`(`" "`)`", et séparées par une virgule "`,`" (pour signifier un ET logique entre ces éléments) ou un pipe "`|`"(pour un OU logique). Dans le cas du ET, il est impératif de respecter l'ordre des éléments dans la DTD. Dans le cas de contenus mixtes, PCDATA apparaît toujours en début de liste dans la DTD, même si dans le document, les données textuelles peuvent apparaître n'importe où dans l'élément.

Par ailleurs, il existe des *opérateurs de cardinalité* qui spécifient le nombre d'occurrences de chaque type d'élément enfant compris dans le contenu de l'élément :

- Aucun opérateur signifie que le contenu apparaît une fois et une seule.
- `?` indique 0 ou une occurrence du contenu (contenu facultatif).
- `*` indique 0 ou plusieurs occurrences du contenu (contenu facultatif)
- `+` indique une ou plusieurs occurrences (contenu obligatoire).

Les attributs peuvent avoir différents types :

- `CDATA` : chaîne textuelle simple.
- Valeurs énumérées dans une liste de choix entre parenthèses et séparés par des pipes.
- `ID` : un identifiant unique respectant la syntaxe des noms d'éléments XML.
- `IDREF` : référence à un élément ayant pour valeur de l'attribut ID la valeur indiquée.
- `IDREFS` : liste de plusieurs IDREF séparés par des esapces.
- `NMTOKEN` : jeton de nom conforme aux règles de noms XML (permet de limiter le nombre de valeurs que peut prendre l'attribut).
- `NMTOKENS` : liste de jetons de noms séparés par des espaces.
- `ENTITY` : nom d'une entité prédéfinie.
- `ENTITIES` : liste de plusieurs entités séparés par des espaces.
- `NOTATION` : type de notation déclaré ailleurs dans la DTD.

Un attribut peut avoir les valeurs par défaut « `#REQUIRED` », « `#IMPLIED` » (facultatif), « `FIXED` » (avec une valeur par défaut : s'il est présent, il doit prendre cette valeur, sinon, cette valeur par défaut lui est attribuée), ou une valeur par défaut entre double guillemets (qui peut utiliser des caractères génériques)

# Les espaces de noms XML

## Position du problème

XML permet une grande liberté dans le choix des noms de balises utilisées pour structurer l'information. Des balises identiques peuvent être utilisées pour décrire différents types de méta-information, ce qui est parfaitement valide en XML. Par conséquent, des conflits peuvent apparaître lors de l'analyse et de l'interprétation (mise en forme) du document. Par exemple, les balises `<title>`, `<commande>`, `<liste>` peuvent être interprétées de façons très diverses (balises HTML ou titre de civilité, instruction ou sélection d'articles, liste d'éléments de natures très diverses…).

## Principe général

Cette redondance est imputable à une *polysémie* des termes, et peut être levée par la donnée du contexte d'utilisation de ces termes. Les *espaces de noms XML* ont été créés pour différencier les mêmes termes utilisés dans le contexte de *vocabulaires* différents. Ces vocabulaires sont des regroupements de termes établis de manière purement conceptuelle. L'idée est bien entendu de définir ces regroupements de manière suffisamment fine pour ne pas avoir de conflits de termes à l'intérieur d'un espace de noms.

Remarque : les espaces de noms XML sont des vocabulaires et non pas des types de documents. Ils ne fournissent aucune information sur la signification ou la structure des méta-données.

## Unicité des espaces de noms

Chacun de ces vocabulaires est alors défini de façon unique en le liant avec une URI (i.e. une URL ou une URN[4]). L'idée est de particulariser les balises en leur ajoutant l'URI comme préfixe.

Par exemple, la balise <title> correspondant à l'annonce d'un titre en XHTML pourrait être définie par `<{http://www.w3.org/1999/xhtml}title>`, ce qui permettrait par exemple de la différencier de celle de votre CV, définie par `<{http://www.iup.univ-avignon.fr/etds/CV/english}title>`. Bien entendu, ces balises ne sont pas correctes, puisqu'elles contiennent les caractères '{', '}' et '/', qui sont interdits dans les balises XML.

## Syntaxe d'utilisation des espaces de noms XML

La syntaxe retenue par le W3C (disponible à l'adresse : http://www.w3.org/TR/REC-xml-names) est l'utilisation de *noms qualifiés* (Qnames). Ces noms qualifiés sont composés de deux parties : la *partie locale*, (les noms « simples » des éléments dans le fichier XML d'origine) et le *préfixe d'espace de noms*, indiquant l'espace de noms auquel l'élément appartient. Ce préfixe fait référence à un espace de noms désigné par une URI. Le préfixe est séparé de la partie locale du nom par le caractère ':'.

L'association du préfixe et de l'espace de noms est réalisée en utilisant un attribut commençant par le terme « xmlns: ». Cet identifiant doit respecter la syntaxe XML. L'attribut ainsi constitué reçoit pour valeur l'URI choisie (ici, des URL complètes, c'est-à-dire avec le nom du protocole utilisé et éventuellement les sous-répertoires du nom de domaine).

```
<?xml version="1.0"?>
<etd:CV xmlns:etd="http://www.iup.univ-avignon.fr/etds/CV/english"
        xmlns:xhtml="http://www.w3.org/1999/xhtml">
  <etd:personne>
    <etd:civil_status>
      <etd:title>Mr.</etd:title>
    </etd:civil-status>
    ...
  </etd:personne>
  <xhtml:html>
    <xhtml:head>
      <xhtml:title>CV of an IUP student</xhtml:title>
    </xhtml:head>
    <xhtml:body>
    ...
    </xhtml:body>
  </xhtml:html>
</etd:CV>
```

Remarques :

- On ne peut utiliser plusieurs URI différentes pour spécifier le même espace de noms (comme on le fait par exemple pour localiser une DTD), car cela conduirait à définir plusieurs valeurs pour le même attribut d'une balise XML.
- Les attributs XML peuvent également être associés à des espaces de noms, selon la syntaxe : `<nom_ns1:balise nom_ns2:attribut="valeur">`. Toutefois, toutes

[4] Universal Resource Name, voir plus loin.

les applications ne reconnaissent pas le nommage des attributs XML. On tentera donc de ne pas l'utiliser.

- Le préfixe en lui-même n'a pas de signification. Son remplacement par n'importe quel autre préfixe n'altèrerait en rien la signification du document.
- Au niveau interne, lors de l'analyse d'un document, le processeur remplace simplement tous les préfixes d'espaces de noms par l'espace de noms lui-même (i.e. l'URI référencée). Cette syntaxe non valide utilisée en interne par le processeur est appelée *noms pleinement qualifiés*.
- Le DOM permet cependant l'accès au préfixe. Cela est notamment utilisé par la feuille de style XML par défaut de IE5, pour l'affichage des documents XML avec les préfixes d'espaces de noms définis par leur auteur.

## Espaces de noms par défaut

Un espace de noms par défaut fonctionne comme un espace de noms normal, hormis qu'il ne lui est pas associé de préfixe. Tous les éléments du document XML ne possédant pas de préfixe lui sont alors associés.

```
<?xml version="1.0"?>
<CV xmlns="http://www.iup.univ-avignon.fr/etds/CV/english"
    xmlns:xhtml="http://www.w3.org/1999/xhtml">
  <personne>
    <civil_status>
      <title>Mr.</title>
    </civil-status>
    ...
  </personne>
  <xhtml:html>
    <xhtml:head>
      <xhtml:title>CV of an IUP student</xhtml:title>
    </xhtml:head>
    <xhtml:body>
    ...
    </xhtml:body>
  </xhtml:html>

</CV>
```

## Espaces de noms descendants

L'attribut « xmlns: » de déclaration d'espaces de noms n'est pas nécessairement situé dans l'élément racine du document XML. Cet attribut peut être associé à tout autre élément regroupant l'ensemble des termes utilisés dans le vocabulaire. Dans certains cas, cela améliore même la lisibilité du document.

```
<?xml version="1.0"?>
<CV xmlns="http://www.iup.univ-avignon.fr/etds/CV/english">
  <personne>
    <civil_status>
      <title>Mr.</title>
    </civil-status>
    ...
  </personne>

  <xhtml:html xmlns:xhtml="http://www.w3.org/1999/xhtml">
```

```
    <xhtml:head>
      <xhtml:title>CV of an IUP student</xhtml:title>
    </xhtml:head>
    <xhtml:body>
    ...
    </xhtml:body>
  </xhtml:html>

</CV>
```

Il est même possible de définir des espaces de noms par défaut différents pour les éléments enfants et pour l'élément racine.

```
<?xml version="1.0"?>
<CV xmlns="http://www.iup.univ-avignon.fr/etds/CV/english">
  <personne>
    <civil_status>
      <title>Mr.</title>
    </civil-status>
    ...
  </personne>

  <html xmlns="http://www.w3.org/1999/xhtml">
    <head>
      <title>CV of an IUP student</title>
    </head>
    <body>
    ...
    </body>
  </html>
</CV>
```

## Annulation des espaces de noms

Il est possible d'annuler l'appartenance d'un élément interne à un espace de noms en lui associant un espace de noms par défaut vide. On emploie alors l'attribut « `xmlns=""` ».

## URI, URL, URN et signification des espaces de noms

Nous avons vu qu'un espace de noms est désigné par une URI (c'est-à-dire une URL ou une URN complètes). L'objectif est de définir une ressource conceptuelle permettant d'assurer l'unicité de l'espace de noms. Plusieurs raisons peuvent être avancées pour justifier le choix de l'une ou l'autre de ces notations.

D'un point de vue théorique, le choix d'une URN est plus « libre », et correspond mieux à la définition d'un espace de noms. En effet, la structure une URN se présente sous la forme suivante : « urn:NID:NSS », où NID (namespace identifier) représente le nom de l'espace de noms et NSS (namespace specific string, à vérifier) une chaîne spécifique à l'espace de noms. Les URN permettent donc de *nommer* une ressource et non pas de *pointer vers* une ressource, ce qui est plus conforme à l'esprit de la définition d'espaces de noms, tel qu'il est défini dans la recommandation du W3C.

D'un point de vue pratique, rien ne garantit l'unicité de la NSS d'une URN. L'utilisation d'URLs est donc un moyen simple d'éviter les conflits entre les espaces de noms.

Cependant, on pourrait être tenté d'aller plus loin dans la définition d'espaces de noms. En effet, la plupart des utilisateurs voudraient donner du sens aux noms des éléments. Cela permettrait notamment, dans l'optique de la structuration documentaire, de définir la signification de la méta-information présente dans les documents XML : à quoi cette méta-information sert-elle ? Quelles sont les différentes valeurs possibles pour les balises. S'il était possible de définir une description liée à l'espace de noms, cela serait sans doute un bon moyen de valider la conformité du document en fonction des espaces de noms utilisés. Cela pourrait même permettre d'éviter l'écriture de DTDs, parfois complexes, et souvent fastidieuses.

A l'heure actuelle, la spécification des espaces de noms XML du W3C indique que le rôle premier d'une URI d'espace de noms n'est pas de permettre la récupération d'une telle description. L'URI est uniquement un identificateur, qui n'a pas de sens en soi. En particulier, il ne signifie rien pour le processeur XML, qui le transmet tel quel à l'application.

Cependant, rien n'empêche le « propriétaire » de l'espace de noms (c'est-à-dire le premier à s'être attribué l'URI référencée) de fournir une description de cet espace qui sera utilisée pour la validation du document XML. Une telle description ne peut être fournie sous forme de DTD, car un document XML ne peut comporter qu'une seule DTD. L'utilisation de *schémas XML* est plus indiquée pour une telle description.

# Les schémas XML

Bien que les DTD soient la méthode de validation de documents prescrite par la recommandation XML 1.0, la description de documents XML par les DTD compte un certain nombre de limites. Une alternative aux DTD est proposée par le concept de schémas XML. Le document de travail concernant les schémas XML date du 10 avril 2000 ; il est devenu une recommandation officielle du W3C depuis le 2 mai 2001. Cette recommandation est disponible en trois parties.
XML Schema part 0: Primer (introduction) : http://www.w3.org/TR/xmlschema-0
XML Schema part 1: Structures : http://www.w3.org/TR/xmlschema-1
XML Schema part 2: Datatypes : http://www.w3.org/TR/xmlschema-2

Ce cours résume les principales caractéristiques de schémas XML tels qu'ils ont été définis dans cette recommandation. Il s'est également inspiré d'ouvrages traitant de ces schémas publiés avant la publication définitive de la recommandation et inspirés d'une syntaxe parfois différente de celle de la recommandation. Le maximum a été fait pour corriger cette variabilité et rendre les éléments de syntaxe présentés dans ce cours conformes à la recommandation. En tout état de cause, les concepts fondamentaux liés aux schémas sont identiques dans la recommandation et dans tous les ouvrages. Le lecteur est simplement invité à s'assurer de la conformité de la syntaxe présentée ici avec celle de la version des schémas qu'il utilise.

## Comparaison schémas et DTD

Le tableau suivant récapitule les principales différences entre les schémas XML et les DTD. Il permet de se rendre compte des différences existant entre ces deux concepts.

| Caractéristique | DTD | Schémas |
|---|---|---|
| **Syntaxe** | Notation EBNF (Extended Backus-Naur Form) + pseudo-XML | XML 1.0 |
| **Outils** | Outils SGML existants (chers et complexes) | Tous les outils XML existants et à venir (DOM, XSLT, navigateurs) |
| **Support DOM** | Non | Oui (affichage et manipulation comme pour les fichiers XML). |
| **Modèles de contenu** | - Listes : ordonnées ou de choix<br>- Cardinalité : 0, 1 ou plusieurs occurrences<br>- Pas d'éléments nommés ou de groupes d'attributs. | - Listes : ordonnées et de choix (détails de contenus mixtes)<br>- cardinalité : spécification d'un nombre exact d'occurrences possible<br>- groupes de modèles nommés |
| **Typage des données** | Faible (chaînes, jetons nominaux, ID…) | Fort (nombres, chaînes, date/heure, booléen, structures…) |
| **Portée des noms** | Globale | Globale ou locale |
| **Héritage** | Non | Oui |
| **Extensibilité** | Non (pas sans modification de la recommandation XML 1.0) | Oui (puisque fondés sur l'extensibilité de XML) |
| **Contraintes légales** | Compatibilité avec SGML | Aucune (simplement des « emprunts » aux DTD, comme pour les types de données) |
| **Nombre de vocabulaires supportés** | Une seule DTD par document | Autant que nécessaire (grâce aux espaces de noms) |
| **Dynamicité** | Aucune (les DTD sont en lecture seule) | Peuvent être modifiés en cours d'exécution (par exemple avec le DOM) |

## Caractéristiques des schémas XML

**Syntaxe :** contrairement aux DTD, dont la syntaxe particulière est héritée des spécifications de SGML, les schémas XML sont des parties de document XML. Ils respectent donc la syntaxe du langage. Cela leur confère la même extensibilité que XML, et permet de les manipuler avec les mêmes outils. En particulier, on peut utiliser le DOM pour analyser et même modifier dynamiquement un schéma XML en cours d'utilisation.

**Modèles de contenu :** Comme nous l'avons vu, les DTD ne permettent pas de spécifier des modèles de contenus précis. Tout au plus, peut ont indiquer une liste des différents types d'éléments et des indications de cardinalité peu précises. Les schémas XML permettent en revanche de spécifier avec tout le niveau de détail nécessaire les modèles de contenu. Les contenus mixtes peuvent alors être décrits et validés précisément, aussi bien pour les données XML traditionnelles (i.e. « documentaires ») que pour les applications XML récentes ou futures.

**Typage des données :** pour permettre la compatibilité avec toutes ces applications, les schémas sont capables de prendre en charge tous les types de données. Cela est réalisé par deux mécanismes distincts, qui font l'objet de deux documents distincts des spécifications du W3C : les *types de données* (datatypes) et les *structures*. Nous revenons sur ces mécanismes plus loin.

**Extensibilité :** alors que la syntaxe des DTD est figée (tout ajout à cette syntaxe nécessite une modification de la recommandation XML 1.0), l'extensibilité des schémas XML est principalement due à l'absence de limitation dans l'utilisation des types de données. De plus, elle profite du partage de vocabulaires, grâce à la prise en charge des espaces de noms XML.

**Dynamicité :** Outre la modification dynamique d'un schéma à l'aide du DOM (peu recommandée, car la capacité de prise en charge de ce type de modifications est très dépendante du processeur utilisé), il est possible de sélectionner dynamiquement (i.e. en fonction des actions de l'utilisateur), le schéma ou la partie de schéma à appliquer à un élément d'un document XML.
Par exemple, après avoir effectué une recherche bibliographique en ligne à l'aide d'un schéma approprié, un utilisateur peut décider de commander un livre, à l'aide d'un schéma totalement différent et choisi dynamiquement en fonction du pays à partir duquel il réalise sa commande.

## Principes de base des schémas

Le but d'un schéma est de définir une « classe de documents XML », également souvent désignée sous le terme *instance de document*. Ni les instances ni les schémas n'existent indépendamment les uns des autres. De plus, ces termes ne font pas directement référence à des fichiers à proprement parler. Il s'agit dans les deux cas de parties de documents XML, qui peuvent cependant constituer des fichiers entiers.

Nous avons vu que la spécification officielle du W3C propose deux mécanismes de définition des données : les types de données et les structures. Il existe également deux modes de balisage, qui correspondent – à peu près – à ces deux parties de la spécification : les *définitions* et les *déclarations*.

Bien que le W3C ait choisi de spécifier d'abord les structures et ensuite les types de données (après les avoir cependant présentés dans l'ordre inverse dans le document non normatif d'introduction), nous faisons ci-dessous l'inverse.

## Les types de données

Ce paragraphe décrit les différentes caractéristiques des types de données utilisés dans les schémas XML. Il présente les trois dichotomies utilisées pour décrire les types de données, en fonction du fait qu'ils sont *primitifs* ou *dérivés*, *intégrés* ou *dérivés par l'utilisateur* et définis de façon *atomique* ou par *listes*.

Comme dans la plupart des langages de programmation « évolués », tous les types de données des schémas sont définis de façon arborescente dans une *hiérarchie des définitions de type*. La racine conceptuelle de cette arborescence se nomme *définition d'ur-type*. La distinction entre les types de données primitifs et dérivés permet de les situer dans cette hiérarchie. Les types primitifs sont alors les premiers descendants de l'ur-type. Les types dérivés constituent tous les niveaux inférieurs de cette décomposition.

La deuxième classification distingue les types de données déterminés par les spécifications des schémas XML des types définis dans le corps des schémas. Les premiers sont les types intégrés (« built-in »). Ils sont automatiquement reconnus par le processeur XML et peuvent être utilisés dans tout schéma fondé sur la recommandation du W3C. La définition des types dérivés par l'utilisateur fait appel à la notion de structure qui est présentée plus loin.

Remarque : la relation entre ces deux premières classifications est simple : TOUS LES TYPES PRIMITIFS SONT INTÉGRÉS. La réciproque n'est cependant pas vraie : il existe des types intégrés dérivés. Comme leur nom l'indique, tous les types de données dérivés par l'utilisateur sont dérivés.

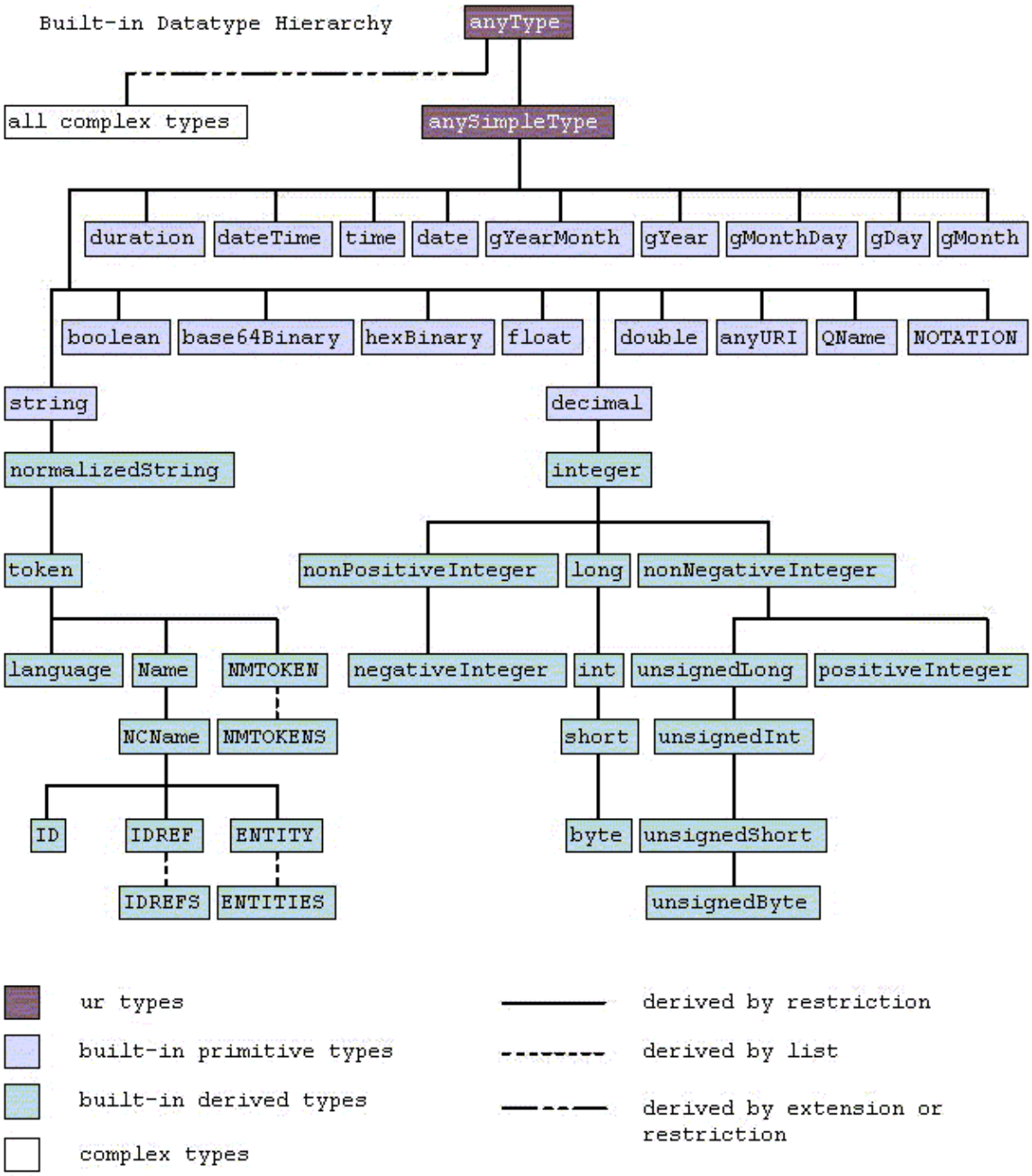
La troisième distinction sépare les types de données atomiques et listes. Les premiers sont constitués de valeurs « indivisibles » (au sens des schémas XML). Les seconds sont constitués d'une séquence délimitée de valeurs atomiques.

Remarque : la relation entre types atomiques / listes et types primitifs / dérivés n'est pas directe. LES TYPES ATOMIQUES NE SONT PAS DES TYPES PRIMITIFS. Il existe des types atomiques primitifs et dérivés. En revanche, les types listes sont une catégorie de types dérivés.

**Les types de données primitifs** peuvent être considérés comme la matière première de définition des éléments XML. Ils peuvent être utilisés pour spécifier les *valeurs* d'éléments ou d'attributs, mais ne peuvent pas avoir de *contenu* (i. e. des éléments enfants ou des attributs).

**Les types de données dérivés** sont définis à partir d'un type de données existant (le type de base). Ils peuvent dériver d'un type primitif ou d'un autre type dérivé. Un type de données dérivé peut contenir un texte XML bien formé et valide (selon la définition du type de données de base), ainsi que des attributs.

La figure ci-dessous (extraite de la recommandation du W3C) donne la liste de tous les types de données intégrés.




**Les types de données atomiques** peuvent être primitifs (comme les chaînes de caractères, puisque le type caractère n'existe pas) ou dérivés (comme les entiers, qui dérivent du type float). Les exemples suivants présentent respectivement des éléments de ces deux types atomiques.

```
<atome_primitif>Cette chaine est appropriee parce qu'elle ne peut
etre divisee en caracteres</atome_primitif>
<atome_derive>231344</atome_derive>
```

**Les types de données listes** sont toujours des dérivés de types atomiques et ne peuvent être constitués d'autres listes. Les éléments d'une liste sont toujours délimités par des espaces. De ce fait, les espaces ne peuvent intervenir à l'intérieur de l'une des valeurs de la liste (comme c'est par exemple le cas dans une chaîne de caractères). Par exemple, si l'on définit une liste

`taille`, ayant pour base le type intégré `decimal` (la syntaxe d'une telle définition est présentée plus loin), un élément qui utilise ce type de liste est :

```
<Pointures xsi:type="tailles">8 8.5 9 9.5 10 10.5 11 12
13</Pointures>
```

Remarque : de la même façon, une liste ayant pour base le type atomique `string` et présentée sous la forme : `<ListeTexte>Element un, element deux, element cent-quatre-vingt-douze</ListeTexte>` comporte cinq espaces et donc six éléments.

## Les facettes des types de données

Tous les types de données possèdent trois caractéristiques : un *espace de valeurs*, un *espace lexical* et des *facettes*.

**L'espace de valeurs** d'un type de données désigne l'ensemble des valeurs pouvant être prises par les éléments qui s'y réfèrent. Ces espaces de valeurs sont implicites dans la définition des types primitifs. Par exemple, l'espace de valeurs lié au type `float` va de moins l'infini à plus l'infini. Les types de données dérivés héritent des espaces de valeurs de leurs types de base. Par conséquent, l'espace de valeurs du type `integer` est le même que celui du type `float`.

**L'espace lexical** d'un type de données désigne la chaîne de caractères *représentant* la valeur de l'élément. Ces chaînes comportent toujours du texte valide en XML, en fonction de l'encodage choisi. Les chaînes de caractères n'ont par exemple qu'une seule représentation lexicale, alors que d'autre types peuvent en avoir plusieurs. Par exemple, dans l'espace de valeurs «nombres à virgule flottante », 100, $1^E2$ et $10^2$ représentent le même nombre. Ces représentations sont cependant différentes au niveau lexical.

**Les facettes** désignent les propriétés définitionnelles des types de données, permettant de les différencier les uns des autres. Il existe des propriétés abstraites, nommées *facettes fondamentales*, et des limites facultatives sur l'espace de valeurs des types de données, appelées *facettes de contrainte*.

Il existe cinq **facettes fondamentales** :

- *égalité* : cette propriété s'applique à tous les types de données, numériques ou non, et peut dépendre de la casse et de l'encodage des valeurs,
- *ordre* : les relations d'ordre sont intrinsèques pour les valeurs numériques et textuelles (pour ces dernières, elles dépendent également de l'encodage utilisé),
- *bornes* : les valeurs d'un type de données peuvent avoir une borne supérieure ou inférieure ou les deux ; dans ce dernier cas, l'espace de valeurs est considéré comme borné,
- *cardinalité* : tous les types de données sont concernés par cette propriété ; la cardinalité d'un espace de valeurs peut être finie (comme dans le cas d'une liste de valeurs énumérées), infinie dénombrable ou infinie indénombrable,
- *numérique / non numérique* : cette propriété est vraie si les valeurs prises par le type de données appartiennent à un système de nombres.

Il existe 12 **facettes de contraintes** :

- `length`, `minlength`, `maxlength` : traitent le nombre d'unités de longueur d'un type de données en nombres de « points de code Unicode » (i.e. sur 8, 16 ou 32 bits) pour les chaînes de caractères, en nombres d'octets pour les données de types binaires et dérivés et en nombres d'éléments pour les données de types listes,
- `pattern` : contrainte de l'espace lexical, qui limite l'espace des valeurs, via une expression régulière « regex » (langage proche de PERL),
- `enumeration` : limite l'espace de valeurs à une liste de choix énumérées,
- `whiteSpace` : s'applique uniquement aux types dérivés de string ; contraint la manière dont sont traités les caractères d'espacement ; cette facette peut prendre les valeurs `preserve` (aucune modification n'est apportée à la mise en forme), `replace` (les caractères de tabulation et de retour à la ligne sont remplacés par des espaces) ou `collapse` (effet identique à la précédente, plus remplacement des séquences de caractères d'espacement par un caractère d'espacement unique),
- `minInclusive`, `maxInclusive`, `minExclusive`, `maxEclusive` : définissent les bornes inférieures et supérieures de l'espace de valeurs, par des inégalités larges ou strictes,
- `totalDigits` : s'applique uniquement aux données de type `decimal` (ou dérivées de ce type) ; cette facette définit le nombre total de chiffres que peuvent prendre les données,
- `fractionDigits` : s'applique uniquement aux données de type `decimal` (ou dérivées de ce type); cette facette définit le nombre de chiffres de la partie décimale des données.

## Les structures

Les structures permettent de décrire des types d'éléments, des attributs d'éléments et des *modèles de contenu* d'éléments. Pour cela, elles s'appuient sur la définition de types de données, selon les mécanismes de dérivation évoqués plus haut. Il existe des types de données *simples* et *complexes*. Toutes les structures de données présentées ici correspondent donc à des types de données dérivés par l'utilisateur[5].

### *Les types simples*

Ils consistent à dériver un type de données atomique[6] par restriction, par liste ou par union. La dérivation par restriction s'obtient en spécifiant un ensemble de contraintes sur les espaces de valeurs ou lexical d'un type de base. La dérivation par liste est la spécification de l'ensemble des valeurs pouvant être prises par un élément du type concerné. La dérivation par union permet de définir un type simple de données en tant que sur-ensemble d'autres types simples.

Remarque : les modes de dérivation par restriction et par liste correspondent au concept de définition par intension ou par extension. Dans le premier cas, on définit un ensemble par les propriétés de ses éléments. Dans le second, on nomme explicitement tous ses éléments.

[5] D'après les spécifications du W3C, la plupart des éléments XML décrits dans cette partie peuvent contenir un élément `<annotation>`. Les annotations sont utilisées pour transmettre des commentaires soit aux lecteurs humains du schéma, soit à l'application. Pour cela, elles contiennent respectivement des éléments enfants `<documentation>` et `<appInfo>`. Elles ne sont pas plus détaillées ici. L'inclusion de cette balise dans les éléments de définition de types est quasi-systématique et n'est plus mentionnée dans la suite.

[6] La dérivation de types listes n'est pas autorisée.

La définition de types simples est encadrée par l'élément `<simpleType>`. Comme les types primitifs dont ils héritent de plus ou moins loin, les types simples ne peuvent avoir de contenu. Il ne sont utilisables que pour spécifier des valeurs d'attributs ou de contenus d'éléments n'ayant pas d'enfants.

**Attributs** : l'élément `<simpleType>` prend les attributs suivants.

- `name` : NCName (nom du type de données) ; facultatif (dans ce cas, le type est dit *anonyme* et s'applique à l'élément, attribut ou annotation de niveau immédiatement supérieur),
- `final` : `#all`, ou (`list` ou `union` ou `restriction`) ; spécifie les modes de dérivation pour lesquels le type ne peut pas être dérivé ; facultatif (valeur par défaut : aucun mode de dérivation),

**Contenu** : le mode de dérivation est indiqué par un élément enfant de l'élément `<simpleType>` parmi l'un des trois suivants : `<restriction>`, `<list>` ou `<union>`.

`restriction` doit posséder un attribut `base` (Qname) qui a pour valeur le nom du type de données de base ; cet attribut est facultatif (valeur par défaut : ur-type). Le contenu des descriptions de types simples dérivés par restriction est constitué d'une ou de plusieurs facettes de contrainte représentées par des éléments enfants vides. Exemple :

```
<simpleType name=”EntierNegatif”>
  <restriction base=”xsi:integer”>
    <minInclusive value="unbounded" />
    <maxInclusive value="-1" />
  </restriction>
</simpleType>
```

`list` peut avoir l'attribut `itemType` qui indique le type d'items auquel appartiennent les éléments dont on va donner la liste ou un élément enfant `<simpleType>` qui définit ce type de façon anonyme ; `itemType` et `<simpleType>` sont exclusifs dans un élément `list`, mais l'un des deux est requis, comme dans l'un des deux exemples suivants ;

```
<simpleType name="tailles">
  <list itemType="decimal"/>
</simpleType>
```

```
<simpleType name="tailles">
  <list>
    <simpleType>
      ...
    </simpleType>
  </list>
</simpleType>
```

Cette définition de type permet alors l'emploi dans le schéma du type liste `tailles`, comme dans l'exemple `Pointure` présenté précédemment.

Remarque : de la même façon que pour la dérivation de types par restriction, il est possible de faire figurer des facettes de contraintes ou des annotations dans une définition de liste.

`union` peut avoir l'attribut `memberTypes` qui indique les types d'items auxquels peuvent appartenir les éléments ou un ou plusieurs éléments enfants `<simpleType>` anonymes. `memberTypes` et `<simpleType>` ne sont pas exclusifs dans l'élément `union` (contrairement à l'élément `list`) :

```
<simpleType name="tailles">
  <union memberTypes="xsi:integer">
    <simpleType>
      <restriction base="decimal">
        <fractionDigits value="1"/>
      </restriction>
    </simpleType>
  </union>
</simpleType>
```

Remarque : cette spécification de la syntaxe de définition de l'élément `<simpleType>` est extraite de la recommandation du W3C sur les schémas datant du 02 mai 2001. Elle diffère des versions précédentes en plusieurs points. En particulier, l'attribut `derivedBy` disparaît, puisque le mode de dérivation est indiqué par un élément enfant. D'autre part, il n'est plus possible de définir des types simples abstraits, comme cela était autorisé dans des versions précédentes.

### *Les types complexes*

Ils permettent de spécifier des gammes de contenus plus vastes que ceux des types simples. Ils sont créés par la dérivation d'autres types. Une définition de type complexe est :

- soit une dérivation la restriction d'un type de base complexe,
- soit une dérivation par extension d'un type de base (simple ou complexe),
- soit une dérivation par restriction de l'ur-type definition.

Les types complexes définissent :

- des attributs appartenant aux attributs du type de base,
- des contenus complexes, conformes à un *modèle de contenu* dérivé de celui du type de base.

Par rapport aux types simples, les types complexes proposent en plus une syntaxe de description des contenus. Ils sont introduits par l'élément `<complexType>` et possèdent les attributs suivants :

- *name* : NCName (nom du type de données) ; facultatif (valeur par défaut : l'élément de niveau immédiatement supérieur),
- *mixed* : booléen (l'utilisation de cet attribut n'est pas autorisée si le contenu est de type simple) ; facultatif (valeur par défaut : false)
- *abstract* : booléen (si true, le type ne peut pas être utilisé pour la validation d'un contenu ou d'un attribut ni être dérivé ; ils peuvent juste être utilisés en tant que types de base ou pour spécifier le type d'un élément dans sa déclaration) ; facultatif (valeur par défaut : false),
- *final* : « `#all` », « `extension` », « `restriction` » ou vide (indique la *finalité* du type de données pour chaque mode de dérivation ; un type d'élément final pour un mode de dérivation ne peut pas être dérivé suivant ce mode) ; facultatif (valeur par défaut : vide ; tous les modes de dérivation sont autorisés) ; cet attribut n'est pas hérité par les types dérivés,
- *block* : « `#all` », « `extension` », « `restriction` » ou vide (permet de spécifier les modes de dérivations qui ne doivent pas être utilisés) ; facultatif

(valeur par défaut : la valeur de l'attribut `blockDefault` de l'élément racine `<schema>` s'il y en a une ; sinon : vide) ; cet attribut est hérité par dérivation.

Les définitions de types complexes permettent de spécifier des modèles de contenus constitués d'attributs et de contenus.

**La spécification d'attributs** est faite grâce à l'élément `<attribute>` qui peut contenir des annotations ou un élément `<simpleType>`. Cependant, en règle générale et à l'exception des annotations, cet élément est souvent vide. Il possède les attributs suivants :

- `default` : string (valeur par défaut de l'attribut) ; incompatible avec `fixed`,
- `fixed` : string (valeur constante de l'attribut) ; incompatible avec `default`,
- `form` : « `qualified` » ou « `unqualified` » (définit si l'attribut appartient à un espace de noms ; le cas échéant, il doit être préfixé par le préfixe correspondant à l'espace de noms de niveau immédiatement supérieur) ; incompatible avec `ref`,
- `name` : string ; incompatible avec `ref`,
- `ref` : Qname (référence à un autre type d'attribut) ; facultatif (incompatible avec `form`, `name` et `type`),
- `type` : Qname (type simple de la valeur de l'attribut) ; incompatible avec `ref`,
- `use` : « `optional` », « `prohibited` » ou « `required` » ; facultative (valeur par défaut : « `optional` ») ; doit avoir pour valeur « `optional` » si une valeur par défaut est indiquée (i.e. si `default` est présent).

Remarque : la spécification d'attributs peut aussi être réalisée par l'intermédiaire de groupes d'attributs nommés, introduits par l'élément `<atrributeGroup>` ou par l'autorisation d'attributs non validés (élément `<anyAttribute>`), que nous ne détaillons pas ici.

**La spécification de contenus** se fait par l'intermédiaire de définitions d'un élément `<simpleContent>` ou d'un élément `<complexContent>`. Le premier ne peut pas contenir d'attribut et ne peut contenir qu'un élément indiquant le mode de dérivation utilisé (`<extension>` ou `<restriction>`). Le second peut avoir un attribut (`mixed`, qui indique la présence à la fois de caractères et d'éléments dans le contenu) et d'un élément `<extension>` ou `<restriction>`.

Les éléments `<extension>` et `<restriction>` permettent de spécifier des structures de contenus, introduits par les éléments `<group>`, `<all>`, `<choice>` et `<sequence>`.

## Préambule d'un schéma

Le vocabulaire utilisé dans un schéma XML est défini dans un espace de noms spécifique, identifié dans un *préambule* (i.e. l'élément racine du schéma). Un schéma conforme à la recommandation du W3C a pour racine un élément `schema`, rattaché à l'espace de noms de spécification des schémas du w3C (pour lequel il est recommandé d'utiliser le préfixe « xsd », pour XML schema definition). La syntaxe valide est :

```
<xsd:schema
      xmlns:xsd="http://www/w3.org/2001/XMLSchema"
      xsd:targetNamespace="http://www.monsite.com/monnamespace">
```

L'attribut `targetNamespace` permet d'identifier l'espace de noms que le schéma définit. Tout document XML faisant référence à cet espace de noms et se déclarant conforme à la recommandation sur les instances de schémas XML doit respecter la structure décrite dans ce schéma pour être validé.

Un schéma peut cependant ne pas se référer à un espace de noms. Dans ce cas, on peut utiliser l'attribut `noTargetNamespace` (avec n'importe quelle valeur), ou omettre cet attribut. Sans espace de noms, la déclaration précédente devient alors :

```
<xsd:schema
      xmlns:xsd="http://www/w3.org/2001/XMLSchema"
      xsd:noTargetNamespace="noTargetNamespace">
```

ou simplement :

```
<xsd:schema
      xmlns:xsd="http://www/w3.org/2001/XMLSchema">
```

Remarques :

- Un schéma n'est pas nécessairement situé dans un fichier à part. Ce peut être un sous-élément de l'arborescence d'un document XML.
- Un schéma peut faire référence à des éléments d'autres schémas, par des mécanismes qui ne sont pas détaillés ici.

## Association d'un document XML à un schéma

Pour qu'un document XML puisse être validé par un schéma, il faut déclarer ce document comme une *instance de schéma XML* (le préfixe recommandé est « xsi »). L'attribut `schemaLocation` permet la validation du document en fonction d'un schéma relatif à un espace de noms. La valeur de cet attribut est en deux parties : l'URI de référence du schéma (i.e. le « `targetNamespace` » du schéma) et le nom du fichier (avec éventuellement un chemin) contenant le schéma, séparés par une espace. Il faut aussi déclarer l'espace de noms auquel appartiennent les éléments du document XML à valider. La syntaxe correcte pour l'élément du document à valider est :

```
<ici:element
      xmlns:xsi=″http://www.w3.org/2001/XMLSchema-instance″
      xsi:schemaLocation=″http://monsite.com/monnamespace
            http://monsite.com/monnamespace/schema/MonSchema.xsd″
      xmlns:ici=″http://monsite.com/monnamespace″>
```

Cette déclaration requiert la validation de `ici:element` et de tous ses sous-éléments et attributs préfixés par « `ici:` ».

Si le schéma ne fait référence à aucun espace de noms, on utilise l'attribut `noNamespaceSchemaLocation` avec pour valeur le fichier contenant le schéma :

```
<element
      xmlns:xsi=″http://www.w3.org/2001/XMLSchema-instance″
     xsi:noNamespaceSchemaLocation=″http://monsite.com/monnamespace/
schema/MonSchema.xsd″>
```

Dans ce cas, le schéma introduit par l'élément indiqué au paragraphe précédent et situé dans le fichier indiqué permet de valider `element`, tous ses sous-éléments et leurs attributs.

# Le DOM (modèle objet de document)

## Définition et origine

XML et la plupart des langages qui sont décrits dans ce cours ne permettent que la description des données et éventuellement leur mise en forme. Le concept d'application XML suppose la possibilité de traiter ces données (i.e. pouvoir y accéder, les réorganiser et les modifier) depuis ces applications.

Le DOM a été créé pour permettre la manipulation de documents XML et HTML depuis des langages de programmation destinés au web tels que les langages de scripts ou Java. L'objectif était de rendre les applications w3 à la fois dynamiques (dans la lignée du DHTML) et portables. Le DOM définit une API (Application Programming Interface) fondée sur la notion d'arbre[7] (« tree-based »), qui prend en charge la création, la modification ou l'accès à des documents XML et HTML depuis un programme.

## Principe général

Comme son nom l'indique, le DOM est un modèle objet, qui fait correspondre, à chaque nœud d'un document XML, une *interface* possédant des méthodes et des propriétés définies. Ces interfaces peuvent être vues comme des « classes abstraites » en C++. Les objets du document XML (ou HTML) instancient ces interfaces.

Ce terme vient du fait que le DOM s'interface entre l'application et le parser qui analyse le document XML. Le programmeur qui utilise le DOM travaille donc sur ces interfaces, et non directement sur le document XML source. En particulier, il n'a jamais besoin de connaître le parser utilisé, car celui-ci se situe à un niveau inférieur au DOM et est masqué par lui.

Remarques :

- le DOM s'interfaçant entre l'application et le parser (et non le document directement), une condition essentielle à l'utilisation du DOM est que le document XML soit bien formé (et valide, si le parser utilisé est validant). De même, pour HTML, le DOM ne permet l'accès qu'à des documents valides. Un document HTML non valide est ignoré par le DOM.
- Il n'est pas tout-à-fait exact que le DOM permet d'ignorer le parser utilisé. La connaissance du fait que ce processeur est validant ou non, de son comportement par rapport aux caractères d'espacement multiples, ou des jeux de caractères qu'il prend en charge est nécessaire. Elle permet soit d'obtenir les données correspondant au document en sortie du parser, soit de prévoir la forme dans laquelle ces données seront renvoyées.

## Spécifications

Le « DOM Working group » du W3C a débuté ses activités en août 1997. Une vue de l'état d'avancement des activités de ce groupe de travail est disponible à l'adresse : http://www.w3.org/DOM/Activity. Les figures présentées dans les paragraphes suivants sont extraites de cet état d'avancement.

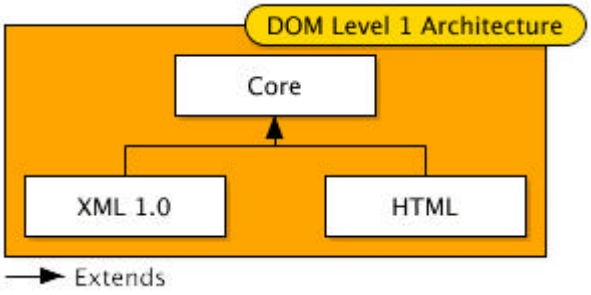
[7] Par opposition à SAX, qui s'appuie sur celle d'événement (« event-based »).

Chacune des recommandations du DOM est décomposée en *modules*. Ces recommandations successives définissent des *niveaux* (DOM Level 1, 2 et 3) qui ont pour but de :

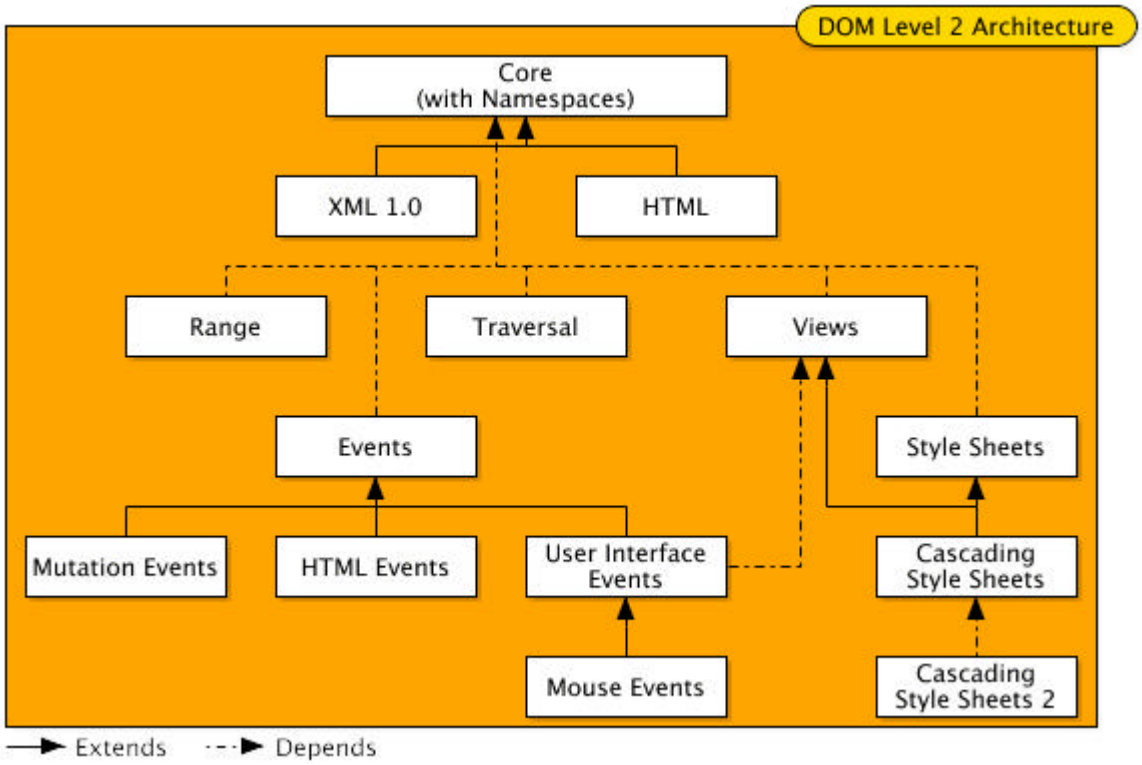
- corriger d'éventuelles erreurs ou imperfections de la version précédente,
- proposer de nouveaux modules pour la prise en charge de nouvelles propriétés.

Remarque : on peut également trouver des références au **DOM niveau 0**. Il s'agit des quelques fonctionnalités de manipulations des documents qui sont implémentées depuis la version 3.0 des navigateurs Netscape™ et Internet Explorer®. Aucune recommandation du W3C ne couvre ces fonctionnalités.

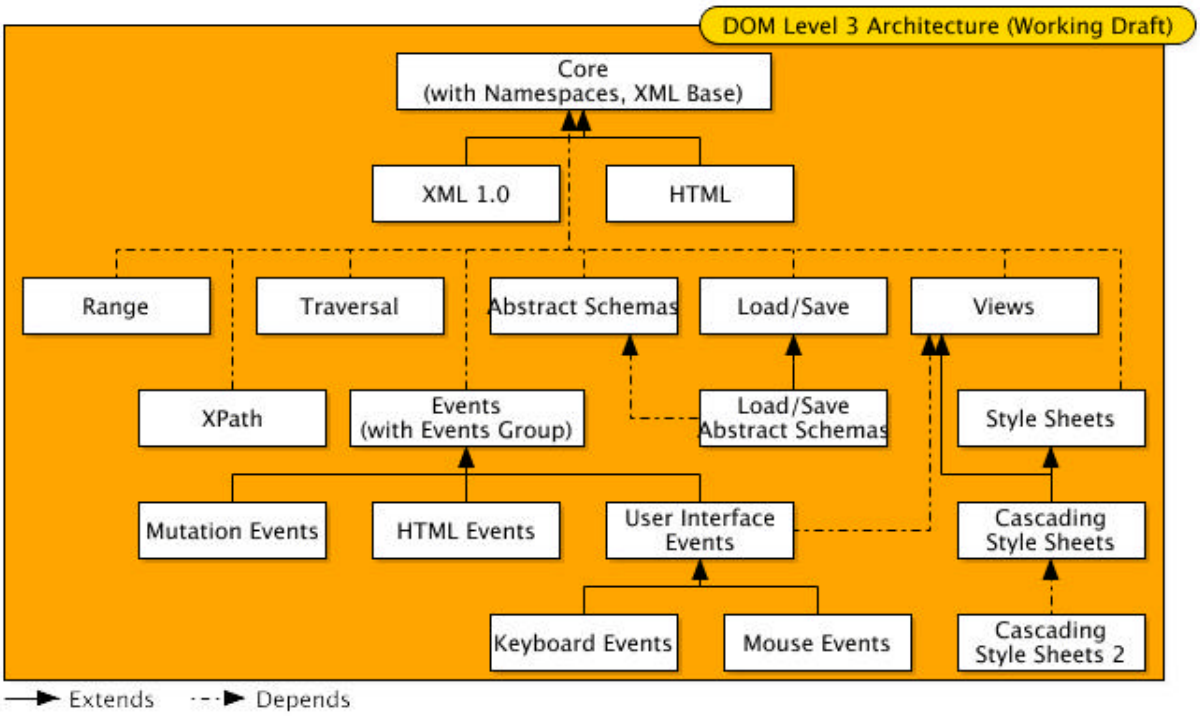
La recommandation du **DOM Niveau 1** date d'octobre 1998. Elle ne définit que trois modules. Un noyau et deux modules spécifiques à HTML et XML. Les fonctionnalités de ces modules sont décrites plus loin.




La recommandation du **DOM niveau 2** date du 13 novembre 2000. Elle est disponible à l'adresse : http://www.w3.org/TR/DOM-Level-2-Core/ (pour le module Core). Elle enrichit le niveau 1 en ajoutant notamment des modules de prise en charge d'évènements et en améliorant les mécanismes de « range » et de déplacement dans les arbres. Le module Core accepte désormais les noms qualifiés (i.e. faisant référence à des espaces de noms).



La recommandation officielle du **DOM niveau 3** était initialement prévue pour la fin de l'année 2001. À l'heure actuelle (début novembre 2001), le DOM niveau 3 est encore au stade de « working draft ». D'après le W3C, « Beginning of 2002 seems a better expectation. » Le niveau 3 comprendra une prise en charge plus complète des espaces de noms que le niveau précédent, et introduit la notion de schémas abstraits (i.e. DTD ou schémas XML) liés à un document XML. Un module « Load and Save » fait également son apparition, ainsi que la prise en charge de XPath.




Les différents modules possèdent des propriétés (*features*) spécifiques. Le tableau suivant indique les propriétés définies par les principaux modules du niveau 3. Ces propriétés ne sont pas des objets du DOM. Elles ne sont utiles que dans des cas particuliers, (par exemple, en tant qu'argument de méthodes comme `hasFeature()`[8]).

| **Nom du module** | **Nom de la propriété** |
|---|---|
| Core | Core |
| XML | XML |
| Events | Events |
| User interface events | UIEvents |
| Mouse Events | MouseEvents |
| Text Events | TextEvents |
| Mutation Events | MutationEvents |
| HTML Events | HTMLEvents |
| Load and Save | LS |
| Abstract Schemas Editing | AS-Edit |
| XPath | Xpath |

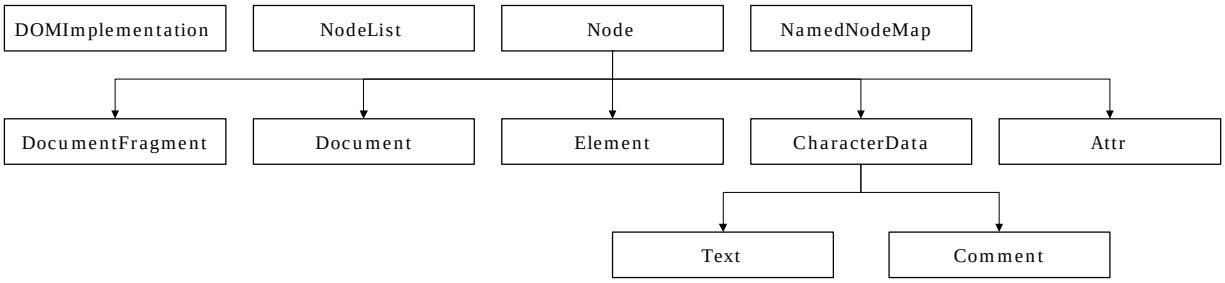
Remarque : les noms de propriétés sont insensibles à la casse (« case-insensitive »).

[8] Méthode qui renvoie `true` ou `false` si un module permet ou non l'accès à la propriété considérée.

## Les différents modules du DOM

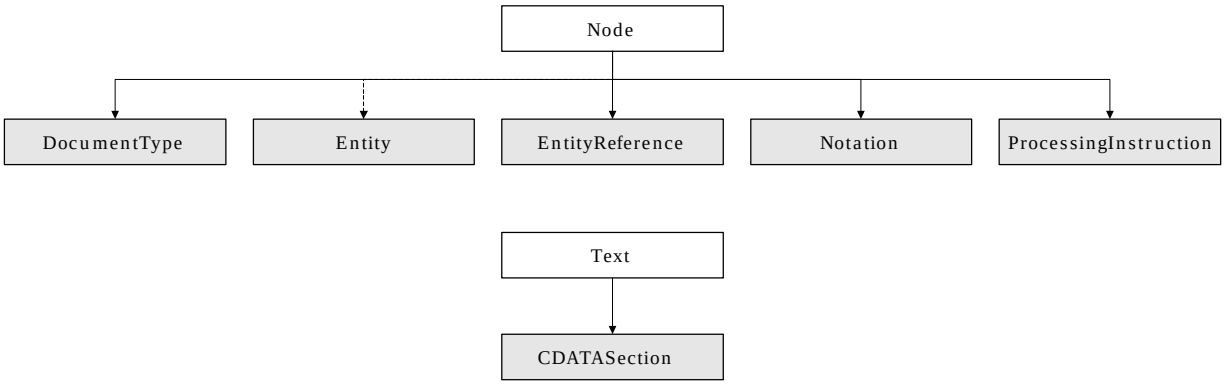
### *DOM Core*

Le module Core définit une représentation interne arborescente d'un document. Il permet les déplacements dans ce modèle de document ainsi que l'accès et la manipulation d'objets du document. Il est également possible, grâce au module Core, de réaliser des manipulations sur la structure arborescente du modèle de document. Pour cela, le DOM Core propose des interfaces *fondamentales*. Ces interfaces sont définies soit dans le module Core lui-même, soit dans les modules dont il dépend (relations en pointillés sur les figures précédentes). Les interfaces fondamentales du DOM sont les suivantes (extraites de DOM Core niveau 2).




### *Le DOM XML*

Il Le DOM propose aussi des interfaces *étendues* définies par les modules d'extension (relations en traits pleins) du module Core, c'est-à-dire les modules XML et HTML. Le DOM XML étend le module Core pour les besoins spécifiques au traitement de documents XML 1.0. Parmi ces spécificités, on compte la prise en compte des entités, des CDATA ou des instructions de traitement. Les interfaces étendues relatives au module XML sont en grisé sur la figure suivante.




### *DOM HTML*

De la même façon, le DOM HTML étend le module Core pour les fonctionnalités spécifiques à HTML. Il n'est pas présenté ici. Ses spécifications sont disponibles à l'URL : http://www.w3.org/TR/DOM-Level-2-HTML.

Il existe aussi d'autres modules qui ne figurent pas sur les diagrammes précédents, concernant la prise en charge d'applications XML spécifiques comme MathML 2.0, SVG 1.0 ou SMIL Animation.

## Interfaces du DOM

Chaque interface est un objet qui peut posséder des propriétés (*properties*[9]) et des méthodes. Celles-ci sont accessibles depuis un langage de programmation possédant une implémentation du DOM.

Remarque : un prototype peut également être associé à une interface ; dans ce cas, il définit des constantes spécifiques à cette interface, qui lui sont associées par le biais d'une propriété spécifique. Par exemple, la propriété `nodeType` de l'interface `Node` définit un ensemble de constantes représentant les différents types de nœuds existants.

### *Détail des interfaces*

Les principales interfaces du DOM sont les interfaces `Node`, `Document` et `Element`. Les deux dernières sont des types particuliers de la première. Elles possèdent donc toutes les propriétés et méthodes de l'interface `Node` en plus de propriétés et de méthodes spécifiques.

- L'interface `Node` permet d'instancier tous les nœuds de l'arborescence, qu'ils correspondent à des éléments XML, des attributs, des instructions de traitement ou d'autres types d'objets. Elle possède toutes les propriétés et méthodes élémentaires qui permettent de caractériser un nœud (comme un nom et une valeur), de se déplacer dans l'arborescence ou de manipuler celle-ci.
- L'interface `Document` est à la racine du modèle de document. Elle permet non seulement d'accéder à l'arborescence des éléments XML, mais également à l'environnement défini à l'extérieur de l'élément racine XML. Par exemple, l'interface `Document` définit les attributs `version`, `encoding` et `standalone` de la déclaration XML, ainsi que différentes parties de la DTD (ou du schéma XML) éventuellement associé(e) au document[10].
- L'interface `Element` désigne une balise du document XML ou HTML. En plus des propriétés de `Node`, elle possède la propriété `tagName`, qui permet d'accéder au nom de la balise. Les méthodes « utiles » spécifiques à l'interface Element sont relatives au traitement des attributs des balises (`getAttribute`, `setAttribute`, `removeAttribute` ou `hasAttribute`).

La liste ci-dessous indique les constantes, propriétés et méthodes des différentes interfaces utilisables depuis les langages de scripts. Elle est extraite des spécification du DOM niveau 3, et date du 13 septembre 2001. Elle est disponible à l'adresse : http://www.w3.org/TR/2001/WD-DOM-Level-3-Core-20010913/ecma-script-binding.html.
Le détail de l'utilisation de chacune des interfaces est donné dans la spécification du module Core Level 3 (http://www.w3.org/TR/2001/WD-DOM-Level-3-Core-20010913/).

Remarque : n'ayant pas encore le statut de recommandation officielle, cette liste d'interfaces peut ne pas encore avoir été entièrement implémentée selon les spécifications du W3C dans les navigateurs ou les applications XML utilisées en TP.

---

[9] Par opposition aux « features » définies pour les modules, voir plus loin.
[10] À l'heure actuelle, la dernière version des spécifications du module Core est plus ancienne que celle du module Abstract Schemas. C'est pourquoi le W3C précise dans la première qu'il n'a pas indiqué les modalités de prise en compte des schémas abstraits depuis la propriété `doctype` de l'interface `Document`.

### *Prototype Object `DOMException`*

The `DOMException` class has the following constants:

`DOMException.INDEX_SIZE_ERR`

This constant is of type **Number** and its value is **1**.

`DOMException.DOMSTRING_SIZE_ERR`

This constant is of type **Number** and its value is **2**.

`DOMException.HIERARCHY_REQUEST_ERR`

This constant is of type **Number** and its value is **3**.

`DOMException.WRONG_DOCUMENT_ERR`

This constant is of type **Number** and its value is **4**.

`DOMException.INVALID_CHARACTER_ERR`

This constant is of type **Number** and its value is **5**.

`DOMException.NO_DATA_ALLOWED_ERR`

This constant is of type **Number** and its value is **6**.

`DOMException.NO_MODIFICATION_ALLOWED_ERR`

This constant is of type **Number** and its value is **7**.

`DOMException.NOT_FOUND_ERR`

This constant is of type **Number** and its value is **8**.

`DOMException.NOT_SUPPORTED_ERR`

This constant is of type **Number** and its value is **9**.

`DOMException.INUSE_ATTRIBUTE_ERR`

This constant is of type **Number** and its value is **10**.

`DOMException.INVALID_STATE_ERR`

This constant is of type **Number** and its value is **11**.

`DOMException.SYNTAX_ERR`

This constant is of type **Number** and its value is **12**.

`DOMException.INVALID_MODIFICATION_ERR`

This constant is of type **Number** and its value is **13**.

`DOMException.NAMESPACE_ERR`

This constant is of type **Number** and its value is **14**.

`DOMException.INVALID_ACCESS_ERR`

This constant is of type **Number** and its value is **15**.

### *Object `DOMException`*

*Remarque : cet objet est à la fois une exception et une interface. Il peut être généré (« raised ») par des propriétés ou méthodes d'autres interfaces et est accessible par le DOM.*

The `DOMException` object has the following properties:

`code`

This property is of type **Number**.

*Remarque : cette propriété renvoie l'une des constantes du prototype ci-dessus.*

### *Object `DOMImplementationSource`*

The `DOMImplementationSource` object has the following methods:

`getDOMImplementation(features)`

This method returns a `DOMImplementation` object.

The `features` parameter is of type **String**.

*Remarque : cette méthode n'apparaît pas dans le document de travail dont est extraite cette liste et la définition de l'objet auquel est s'applique est incohérente avec le reste du document. Je l'ai modifiée pour la faire correspondre aux spécifications du module Core.*

### *Object `DOMImplementation`*

The `DOMImplementation` object has the following methods:

`hasFeature(feature, version)`

This method returns a **Boolean**.
The `feature` parameter is of type **String**.
The `version` parameter is of type **String**.

`createDocumentType(qualifiedName, publicId, systemId)`

This method returns a `DocumentType` object.
The `qualifiedName` parameter is of type **String**.
The `publicId` parameter is of type **String**.
The `systemId` parameter is of type **String**.
This method can raise a `DOMException` object.

`createDocument(namespaceURI, qualifiedName, doctype)`

This method returns a `Document` object.
The `namespaceURI` parameter is of type **String**.
The `qualifiedName` parameter is of type **String**.
The `doctype` parameter is a `DocumentType` object.
This method can raise a `DOMException` object.

`getInterface(feature)`

This method returns a `DOMImplementation` object.
The `feature` parameter is of type **String**.

### *Object `DocumentFragment`*

`DocumentFragment` has all the properties and methods of the `Node` object.

### *Object `Document`*

`Document` has all the properties and methods of the `Node` object as well as the properties and methods defined below.

The `Document` object has the following properties:

`doctype`

This read-only property is a `DocumentType` object.

`implementation`

This read-only property is a `DOMImplementation` object.

`documentElement`

This read-only property is a `Element` object.

*Remarque : cette propriété permet d'accéder à l'élément racine d'un document XML.*

`actualEncoding`

This property is of type **String**.

`encoding`

This property is of type **String**.

`standalone`

This property is of type **Boolean**.

`strictErrorChecking`

This property is of type **Boolean**.

`version`

This property is of type **String**.

The `Document` object has the following methods:

`createElement(tagName)`

This method returns a `Element` object.
The `tagName` parameter is of type **String**.
This method can raise a `DOMException` object.

`createDocumentFragment()`

This method returns a `DocumentFragment` object.

`createTextNode(data)`

This method returns a `Text` object.
The `data` parameter is of type **String**.

`createComment(data)`

This method returns a `Comment` object.
The `data` parameter is of type **String**.

`createCDATASection(data)`

This method returns a `CDATASection` object.
The `data` parameter is of type **String**.
This method can raise a `DOMException` object.

`createProcessingInstruction(target, data)`

This method returns a `ProcessingInstruction` object.
The `target` parameter is of type **String**.
The `data` parameter is of type **String**.
This method can raise a `DOMException` object.

`createAttribute(name)`

This method returns a `Attr` object.
The `name` parameter is of type **String**.
This method can raise a `DOMException` object.

`createEntityReference(name)`

This method returns a `EntityReference` object.
The `name` parameter is of type **String**.
This method can raise a `DOMException` object.

`getElementsByTagName(tagname)`

This method returns a `NodeList` object.
The `tagname` parameter is of type **String**.

`importNode(importedNode, deep)`

This method returns a `Node` object.
The `importedNode` parameter is a `Node` object.
The `deep` parameter is of type **Boolean**.
This method can raise a `DOMException` object.

`createElementNS(namespaceURI, qualifiedName)`

This method returns a `Element` object.
The `namespaceURI` parameter is of type **String**.
The `qualifiedName` parameter is of type **String**.
This method can raise a `DOMException` object.

`createAttributeNS(namespaceURI, qualifiedName)`

This method returns a `Attr` object.
The `namespaceURI` parameter is of type **String**.
The `qualifiedName` parameter is of type **String**.
This method can raise a `DOMException` object.

`getElementsByTagNameNS(namespaceURI, localName)`

This method returns a `NodeList` object.
The `namespaceURI` parameter is of type **String**.
The `localName` parameter is of type **String**.

`getElementById(elementId)`

This method returns a `Element` object.
The `elementId` parameter is of type **String**.

`adoptNode(source)`

This method returns a `Node` object.
The `source` parameter is a `Node` object.
This method can raise a `DOMException` object.

`setBaseURI(baseURI)`

This method has no return value.
The `baseURI` parameter is of type **String**.
This method can raise a `DOMException` object.

### *Prototype Object `Node`*

The `Node` class has the following constants:

`Node.ELEMENT_NODE`

This constant is of type **Number** and its value is **1**.

`Node.ATTRIBUTE_NODE`

This constant is of type **Number** and its value is **2**.

`Node.TEXT_NODE`

This constant is of type **Number** and its value is **3**.

`Node.CDATA_SECTION_NODE`

This constant is of type **Number** and its value is **4**.

`Node.ENTITY_REFERENCE_NODE`

This constant is of type **Number** and its value is **5**.

`Node.ENTITY_NODE`

This constant is of type **Number** and its value is **6**.

`Node.PROCESSING_INSTRUCTION_NODE`

This constant is of type **Number** and its value is **7**.

`Node.COMMENT_NODE`

This constant is of type **Number** and its value is **8**.

`Node.DOCUMENT_NODE`

This constant is of type **Number** and its value is **9**.

`Node.DOCUMENT_TYPE_NODE`

This constant is of type **Number** and its value is **10**.

`Node.DOCUMENT_FRAGMENT_NODE`

This constant is of type **Number** and its value is **11**.

`Node.NOTATION_NODE`

This constant is of type **Number** and its value is **12**.

`Node.TREE_POSITION_PRECEDING`

This constant is of type **Number** and its value is **0x01**.

`Node.TREE_POSITION_FOLLOWING`

This constant is of type **Number** and its value is **0x02**.

`Node.TREE_POSITION_ANCESTOR`

This constant is of type **Number** and its value is **0x04**.

`Node.TREE_POSITION_DESCENDANT`

This constant is of type **Number** and its value is **0x08**.

`Node.TREE_POSITION_SAME`

This constant is of type **Number** and its value is **0x10**.

`Node.TREE_POSITION_EXACT_SAME`

This constant is of type **Number** and its value is **0x20**.

`Node.TREE_POSITION_DISCONNECTED`

This constant is of type **Number** and its value is **0x00**.

### *Object `Node`*

The `Node` object has the following properties:

`nodeName`

This read-only property is of type **String**.

`nodeValue`

This property is of type **String**, can raise a `DOMException` object on setting and can raise a `DOMException` object on retrieval.

`nodeType`

This read-only property is of type **Number**.

`parentNode`

This read-only property is a `Node` object.

`childNodes`

This read-only property is a `NodeList` object.

`firstChild`

This read-only property is a `Node` object.

`lastChild`

This read-only property is a `Node` object.

`previousSibling`

This read-only property is a `Node` object.

`nextSibling`

This read-only property is a `Node` object.

`attributes`

This read-only property is a `NamedNodeMap` object.

`ownerDocument`

This read-only property is a `Document` object.

`namespaceURI`

This read-only property is of type **String**.

`prefix`

This property is of type **String** and can raise a `DOMException` object on setting.

`localName`

This read-only property is of type **String**.

`baseURI`

This read-only property is of type **String**.

`textContent`

This property is of type **String**, can raise a `DOMException` object on setting and can raise a `DOMException` object on retrieval.

*Remarque : cette propriété renvoie la totalité des contenus textuels du nœud courant et de ses enfants. Dans Internet Explorer, cette propriété s'appelle* `text` *et non* `textContent`.

The `Node` object has the following methods:

`insertBefore(newChild, refChild)`

This method returns a `Node` object.
The `newChild` parameter is a `Node` object.
The `refChild` parameter is a `Node` object.
This method can raise a `DOMException` object.

`replaceChild(newChild, oldChild)`

This method returns a `Node` object.
The `newChild` parameter is a `Node` object.
The `oldChild` parameter is a `Node` object.
This method can raise a `DOMException` object.

`removeChild(oldChild)`

This method returns a `Node` object.
The `oldChild` parameter is a `Node` object.
This method can raise a `DOMException` object.

`appendChild(newChild)`

This method returns a `Node` object.
The `newChild` parameter is a `Node` object.
This method can raise a `DOMException` object.

`hasChildNodes()`

This method returns a **Boolean**.

`cloneNode(deep)`

This method returns a `Node` object.
The `deep` parameter is of type **Boolean**.

`normalize()`

This method has no return value.

`isSupported(feature, version)`

This method returns a **Boolean**.
The `feature` parameter is of type **String**.
The `version` parameter is of type **String**.

`hasAttributes()`

This method returns a **Boolean**.

`compareTreePosition(other)`

This method returns a **Number**.
The `other` parameter is a `Node` object.
This method can raise a `DOMException` object.

`isSameNode(other)`

This method returns a **Boolean**.
The `other` parameter is a `Node` object.

`lookupNamespacePrefix(namespaceURI)`

This method returns a **String**.
The `namespaceURI` parameter is of type **String**.

`lookupNamespaceURI(prefix)`

This method returns a **String**.
The `prefix` parameter is of type **String**.

`normalizeNS()`

This method has no return value.

`isEqualNode(arg, deep)`

This method returns a **Boolean**.
The `arg` parameter is a `Node` object.
The `deep` parameter is of type **Boolean**.

`getInterface(feature)`

This method returns a `Node` object.
The `feature` parameter is of type **String**.

`setUserData(key, data, handler)`

This method returns a `Object` object.
The `key` parameter is of type **String**.
The `data` parameter is a `Object` object.
The `handler` parameter is a `UserDataHandler` object.

`getUserData(key)`

This method returns a `Object` object.
The `key` parameter is of type **String**.

### *Object `NodeList`*

The `NodeList` object has the following properties:

`length`

This read-only property is of type **Number**.

The `NodeList` object has the following methods:

`item(index)`

This method returns a `Node` object.
The `index` parameter is of type **Number**.
**Note:** This object can also be dereferenced using square bracket notation (e.g. obj[1]). Dereferencing with an integer `index` is equivalent to invoking the `item` method with that index.

*Remarque : cette note apparaît plusieurs fois dans ce document. Elle signifie qu'il est possible de passer le numéro de l'item recherché soit comme argument de la fonction* `item()` *entre parenthèses (*`objetNodeList.item(n)`*), soit comme index de l'objet* `nodeList` *entre crochets (*`objetNodeList[n]`*).*

### *Object `NamedNodeMap`*

The `NamedNodeMap` object has the following properties:

`length`

This read-only property is of type **Number**.

The `NamedNodeMap` object has the following methods:

`getNamedItem(name)`

This method returns a `Node` object.
The `name` parameter is of type **String**.

`setNamedItem(arg)`

This method returns a `Node` object.
The `arg` parameter is a `Node` object.
This method can raise a `DOMException` object.

`removeNamedItem(name)`

This method returns a `Node` object.
The `name` parameter is of type **String**.
This method can raise a `DOMException` object.

`item(index)`

This method returns a `Node` object.
The `index` parameter is of type **Number**.
**Note:** This object can also be dereferenced using square bracket notation (e.g. obj[1]). Dereferencing with an integer `index` is equivalent to invoking the `item` method with that index.

`getNamedItemNS(namespaceURI, localName)`

This method returns a `Node` object.
The `namespaceURI` parameter is of type **String**.
The `localName` parameter is of type **String**.

`setNamedItemNS(arg)`

This method returns a `Node` object.
The `arg` parameter is a `Node` object.
This method can raise a `DOMException` object.

`removeNamedItemNS(namespaceURI, localName)`

This method returns a `Node` object.
The `namespaceURI` parameter is of type **String**.
The `localName` parameter is of type **String**.
This method can raise a `DOMException` object.

### *Object `CharacterData`*

`CharacterData` has all the properties and methods of the `Node` object as well as the properties and methods defined below.

The `CharacterData` object has the following properties:

`data`

This property is of type **String**, can raise a `DOMException` object on setting and can raise a `DOMException` object on retrieval.

`length`

This read-only property is of type **Number**.

The `CharacterData` object has the following methods:

`substringData(offset, count)`

This method returns a **String**.
The `offset` parameter is of type **Number**.
The `count` parameter is of type **Number**.
This method can raise a `DOMException` object.

`appendData(arg)`

This method has no return value.
The `arg` parameter is of type **String**.
This method can raise a `DOMException` object.

`insertData(offset, arg)`

This method has no return value.
The `offset` parameter is of type **Number**.
The `arg` parameter is of type **String**.
This method can raise a `DOMException` object.

`deleteData(offset, count)`

This method has no return value.
The `offset` parameter is of type **Number**.
The `count` parameter is of type **Number**.
This method can raise a `DOMException` object.

`replaceData(offset, count, arg)`

This method has no return value.
The `offset` parameter is of type **Number**.
The `count` parameter is of type **Number**.
The `arg` parameter is of type **String**.
This method can raise a `DOMException` object.

### *Object `Attr`*

`Attr` has all the properties and methods of the `Node` object as well as the properties and methods defined below.

The `Attr` object has the following properties:

`name`

This read-only property is of type **String**.

`specified`

This read-only property is of type **Boolean**.

`value`

This property is of type **String** and can raise a `DOMException` object on setting.

`ownerElement`

This read-only property is a `Element` object.

### *Object `Element`*

`Element` has all the properties and methods of the `Node` object as well as the properties and methods defined below.

The `Element` object has the following properties:

`tagName`

This read-only property is of type **String**.

*Remarque : cette propriété n'est valable que pour le DOM HTML. Elle associe le type de balise auquel est rattaché un élément.*

The `Element` object has the following methods:

`getAttribute(name)`

This method returns a **String**.
The `name` parameter is of type **String**.

`setAttribute(name, value)`

This method has no return value.
The `name` parameter is of type **String**.
The `value` parameter is of type **String**.
This method can raise a `DOMException` object.

`removeAttribute(name)`

This method has no return value.
The `name` parameter is of type **String**.
This method can raise a `DOMException` object.

`getAttributeNode(name)`

This method returns a `Attr` object.
The `name` parameter is of type **String**.

`setAttributeNode(newAttr)`

This method returns a `Attr` object.
The `newAttr` parameter is a `Attr` object.
This method can raise a `DOMException` object.

`removeAttributeNode(oldAttr)`

This method returns a `Attr` object.
The `oldAttr` parameter is a `Attr` object.
This method can raise a `DOMException` object.

`getElementsByTagName(name)`

This method returns a `NodeList` object.
The `name` parameter is of type **String**.

`getAttributeNS(namespaceURI, localName)`

This method returns a **String**.
The `namespaceURI` parameter is of type **String**.
The `localName` parameter is of type **String**.

`setAttributeNS(namespaceURI, qualifiedName, value)`

This method has no return value.
The `namespaceURI` parameter is of type **String**.
The `qualifiedName` parameter is of type **String**.
The `value` parameter is of type **String**.
This method can raise a `DOMException` object.

`removeAttributeNS(namespaceURI, localName)`

This method has no return value.
The `namespaceURI` parameter is of type **String**.
The `localName` parameter is of type **String**.
This method can raise a `DOMException` object.

`getAttributeNodeNS(namespaceURI, localName)`

This method returns a `Attr` object.
The `namespaceURI` parameter is of type **String**.
The `localName` parameter is of type **String**.

`setAttributeNodeNS(newAttr)`

This method returns a `Attr` object.
The `newAttr` parameter is a `Attr` object.
This method can raise a `DOMException` object.

`getElementsByTagNameNS(namespaceURI, localName)`

This method returns a `NodeList` object.
The `namespaceURI` parameter is of type **String**.
The `localName` parameter is of type **String**.

`hasAttribute(name)`

This method returns a **Boolean**.
The `name` parameter is of type **String**.

`hasAttributeNS(namespaceURI, localName)`

This method returns a **Boolean**.
The `namespaceURI` parameter is of type **String**.
The `localName` parameter is of type **String**.

### *Object `Text`*

`Text` has all the properties and methods of the `CharacterData` object as well as the properties and methods defined below.

The `Text` object has the following properties:

`isWhitespaceInElementContent`

This read-only property is of type **Boolean**.

`wholeText`

This read-only property is of type **String**.

The `Text` object has the following methods:

`splitText(offset)`

This method returns a `Text` object.
The `offset` parameter is of type **Number**.
This method can raise a `DOMException` object.

`replaceWholeText(content)`

This method returns a `Text` object.
The `content` parameter is of type **String**.
This method can raise a `DOMException` object.

### *Object `Comment`*

`Comment` has all the properties and methods of the `CharacterData` object.

### *Prototype Object `UserDataHandler`*

The `UserDataHandler` class has the following constants:

`UserDataHandler.CLONED`

This constant is of type **Number** and its value is **1**.

`UserDataHandler.IMPORTED`

This constant is of type **Number** and its value is **2**.

`UserDataHandler.DELETED`

This constant is of type **Number** and its value is **3**.

### *Object `UserDataHandler`*

The `UserDataHandler` object has the following methods:

`handle(operation, key, data, src, dst)`

This method has no return value.
The `operation` parameter is of type **Number**.
The `key` parameter is of type **String**.
The `data` parameter is a `Object` object.
The `src` parameter is a `Node` object.
The `dst` parameter is a `Node` object.

### *Prototype **`Object`** `DOMError`*

The `DOMError` class has the following constants:

`DOMError.SEVERITY_WARNING`

This constant is of type **Number** and its value is **0**.

`DOMError.SEVERITY_ERROR`

This constant is of type **Number** and its value is **1**.

`DOMError.SEVERITY_FATAL_ERROR`

This constant is of type **Number** and its value is **2**.

### *Object `DOMError`*

The `DOMError` object has the following properties:

`severity`

This read-only property is of type **Number**.

`message`

This read-only property is of type **String**.

`exception`

This read-only property is a `Object` object.

`location`

This read-only property is a `DOMLocator` object.

### *Object `DOMErrorHandler`*

The `DOMErrorHandler` object has the following methods:

`handleError(error)`

This method returns a **Boolean**.
The `error` parameter is a `DOMError` object.

### *Object `DOMLocator`*

The `DOMLocator` object has the following properties:

`lineNumber`

This read-only property is of type **Number**.

`columnNumber`

This read-only property is of type **Number**.

`offset`

This read-only property is of type **Number**.

`errorNode`

This read-only property is a `Node` object.

`uri`

This read-only property is of type **String**.

### *Object `CDATASection`*

`CDATASection` has all the properties and methods of the `Text` object.

### *Object `DocumentType`*

`DocumentType` has all the properties and methods of the `Node` object as well as the properties and methods defined below.

The `DocumentType` object has the following properties:

`name`

This read-only property is of type **String**.

`entities`

This read-only property is a `NamedNodeMap` object.

`notations`

This read-only property is a `NamedNodeMap` object.

`publicId`

This read-only property is of type **String**.

`systemId`

This read-only property is of type **String**.

`internalSubset`

This read-only property is of type **String**.

### *Object* `Notation`

`Notation` has all the properties and methods of the `Node` object as well as the properties and methods defined below.

The `Notation` object has the following properties:

`publicId`

This read-only property is of type **String**.

`systemId`

This read-only property is of type **String**.

### *Object* `Entity`

`Entity` has all the properties and methods of the `Node` object as well as the properties and methods defined below.

The `Entity` object has the following properties:

`publicId`

This read-only property is of type **String**.

`systemId`

This read-only property is of type **String**.

`notationName`

This read-only property is of type **String**.

`actualEncoding`

This property is of type **String**.

`encoding`

This property is of type **String**.

`version`

This property is of type **String**.

### *Object* `EntityReference`

`EntityReference` has all the properties and methods of the `Node` object.

### *Object* `ProcessingInstruction`

`ProcessingInstruction` has all the properties and methods of the `Node` object as well as the properties and methods defined below.

The `ProcessingInstruction` object has the following properties:

`target`

This read-only property is of type **String**.

`data`

This property is of type **String** and can raise a `DOMException` object on setting.

### *Structuration des interfaces*

Le DOM présente tout document comme une hiérarchie d'objets de type `Node`. Ces nœuds implémentent différentes interfaces, qui peuvent chacune avoir différents types de nœuds enfants. La liste suivante décrit les relations de filiation autorisées (extraite de DOM Level 3).

| | |
|---|---|
| `Document` | `Element` (maximum : 1),<br>`ProcessingInstruction`,<br>`Comment`,<br>`DocumentType` (maximum : 1) |
| `DocumentFragment` | `Element`,<br>`ProcessingInstruction`,<br>`Comment`,<br>`Text`,<br>`CDATASection`,<br>`EntityReference` |
| `DocumentType` | pas d'enfant |
| `EntityReference` | `Element`,<br>`ProcessingInstruction`,<br>`Comment`,<br>`Text`,<br>`CDATASection`,<br>`EntityReference` |
| `Element` | `Attr`,<br>`Element`,<br>`Text`,<br>`Comment`,<br>`ProcessingInstruction`,<br>`CDATASection`,<br>`EntityReference` |
| `Attr` | `Text`,<br>`EntityReference` |
| `ProcessingInstruction` | pas d'enfant |
| `Comment` | pas d'enfant |
| `Text` | pas d'enfant |
| `CDATASection` | pas d'enfant |
| `Entity` | `Element`,<br>`ProcessingInstruction`,<br>`Comment`,<br>`Text`,<br>`CDATASection`,<br>`EntityReference` |
| `Notation` | pas d'enfant |

## Utilisation du DOM avec MSXML

Ce paragraphe donne un aperçu plus pratique de l'utilisation qui peut être faite du DOM pour manipuler un document XML avec Internet Explorer® (version 5.0 et suivantes). Il présente quelques opérations de base permettant d'utiliser le DOM, notamment dans des fonctions JScript.

### *Chargement du fichier*

La première chose à faire est de déclarer le document accédé par le DOM dans une variable (ou un objet) JScript. Internet Explorer® utilise la technologie ActiveX (anciennement OLE) pour traiter ces objets. Comme il s'agit d'un document du DOM accédé par le parser XML, la syntaxe est :

```
var XMLDoc = new ActiveXObject("MSXML.DOMDocument");
XMLDoc.async=false;
```

Remarques :

- Le mot-clé `var` est optionnel dans la déclaration ci-dessus.
- La seconde ligne évite le chargement du document en tâche de fond. L'accès au document via le DOM n'étant possible que lorsque celui-ci est entièrement chargé, cela dispense d'employer une temporisation (`window.setTimeout()`) et un test de la propriété `readyState` du contrôle ActiveX[11]. L'inconvénient est la perte de temps que peut occasionner le chargement d'un document volumineux.
- Il est possible de spécifier si l'on souhaite ou non que le document soit validé (par exemple par rapport à une DTD). Pour cela, il faut utiliser la propriété booléenne `validateOnParse` de l'objet `XMLDoc`. La valeur par défaut de cette propriété est `true` lorsque l'on accède à un document XML par le DOM (alors qu'elle est `false` lorsque l'on affiche directement ce document dans Internet Explorer®. Cette méthode ne permet pas la validation par les schémas XML. Celle-ci n'est pas encore correctement implémentée dans Internet Explorer® et n'est pas abordée ici.

Il faut ensuite charger le fichier dans cet objet. Deux méthodes peuvent être utilisées : `load()` et `loadXML()`. L'exemple suivant utilise la première, qui permet l'utilisation de noms de fichiers et de chemins relatifs.

```
XMLDoc.load("fichier.xml");
```

L'objet `XMLDoc` obtenu est alors une instance de l'interface `Document` du DOM.

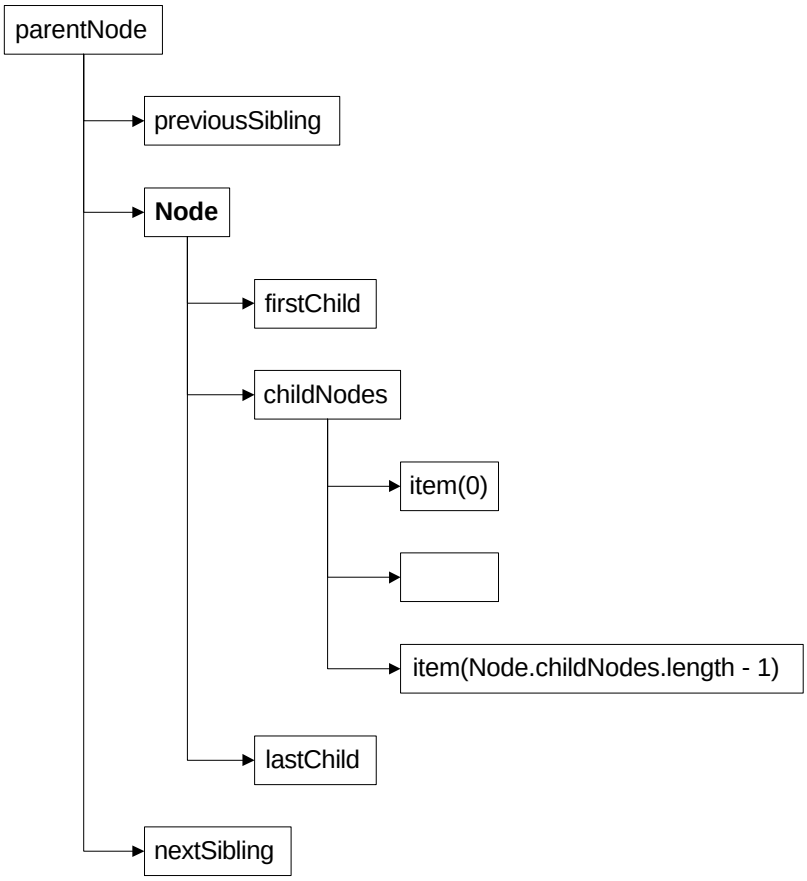
### *Accès à l'élément racine*

La propriété `documentElement` d'un objet `Document` (ici, `XMLDoc`) donne accès à l'élément racine du document XML. Cette propriété renvoie un objet `Element`, qui est un type particulier de `Node`. L'élément racine peut alors être traité comme tous les nœuds de l'arborescence, notamment pour accéder aux autres éléments du document.

[11] Cette propriété renvoie un entier positif entre 0 et 3 tant que le document n'est pas totalement chargé et 4 ensuite.

### *Déplacement dans l'arborescence des éléments*

Le schéma suivant est issu des spécifications du DOM du W3C. Il indique les propriétés de l'interface `Node` permettant, depuis un nœud représentant un élément du document, d'accéder aux autres éléments. Ces propriétés permettent d'accéder aux nœuds représentant des éléments XML ou HTML. Sur ce schéma, on suppose être positionné sur l'interface `Node` indiquée en gras.



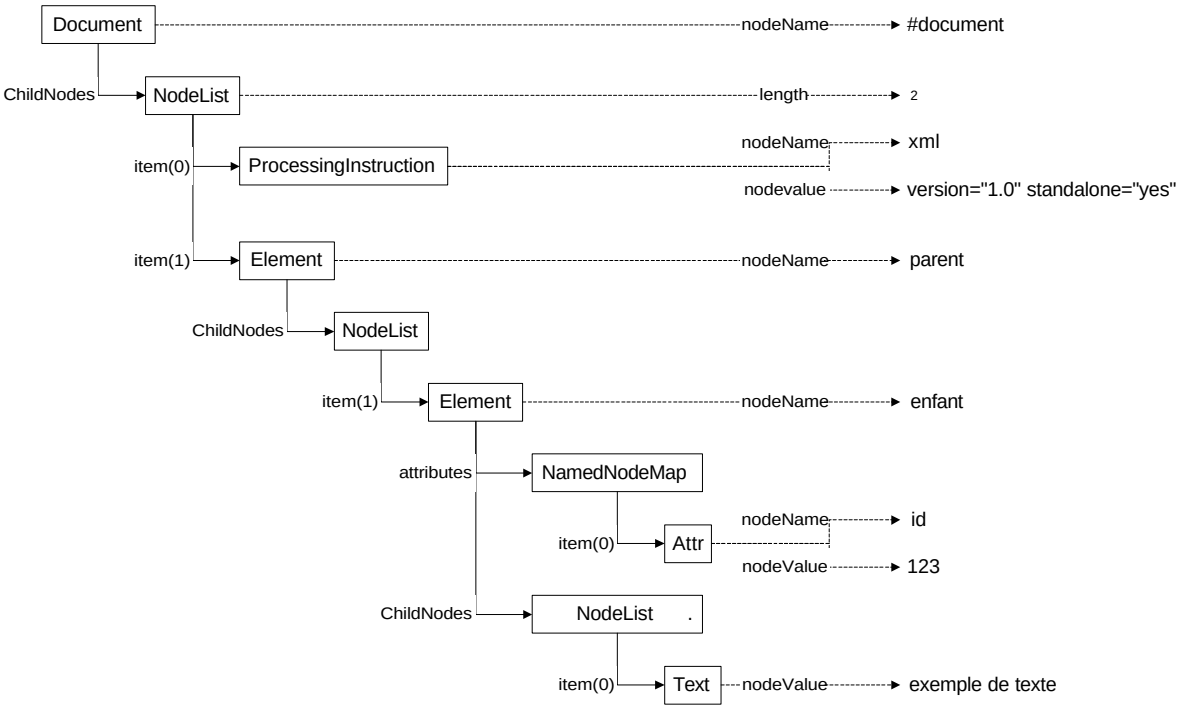

Remarques : le déplacement se fait selon l'arborescence du modèle que le DOM renvoie du document XML, qui peut différer sensiblement de l'arborescence du document lui-même. Par exemple, dans le DOM, les attributs sont représentés par des nœuds enfants du nœud représentant l'élément du document auquel ils se rapportent. Un exemple de cette différence de structures est donné au paragraphe suivant.

## Représentation d'un document XML par le DOM

Considérons le document XML suivant :

```
<?xml version="1.0" standalone="yes"?>
<parent>
     <enfant id="123">exemple de texte</enfant>
</parent>
```

Le schéma suivant montre l'arborescence de nœuds qui permet de représenter ce document avec le DOM.




Légende : les rectangles représentent les nœuds du modèle objet associé au document par le DOM. Les textes à l'intérieur de ces rectangles indiquent les types des nœuds (correspondant aux constantes accessibles par la propriété `nodeType`). Les propriétés permettant d'accéder à ces nœuds sont indiquées par les textes à gauche des flèches. Le contenu de ces nœuds est indiqué à droite, ainsi que les propriétés qui permettent d'y accéder.

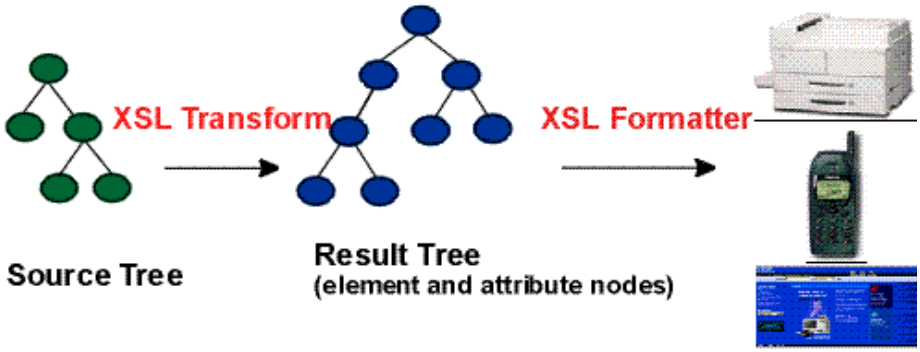
Remarques :

- La valeur de la propriété `nodeName` appliquée au nœud racine du document est empirique. Elle est attribuée à ce nœud par Internet Explorer 5.0.
- Comme décrit au paragraphe précédent, il est possible de « passer » directement du nœud `Document` au premier nœud `Element` par la propriété `documentElement`.
- Il est possible d'aller plus loin dans la décomposition de l'interface `ProcessingInstruction`. Les différentes parties de la déclaration XML peuvent également être obtenues séparément via des attributs de cette interface.
- `NodeList` et `NamedNodeMap` sont des interfaces « *vivantes* » (*live*). Cela signifie que toutes les manipulations de la structure du document sont immédiatement répercutées sur les interfaces de ce type. Par exemple, l'ajout d'un élément fils à un élément existant provoque l'apparition du nœud correspondant dans l'interface `NodeList` relative à ce nœud.
- Sur ce schéma, on voit que pour obtenir le contenu textuel d'un nœud, il faut utiliser la propriété `nodeValue` du nœud `Text` associé. Avec Internet Explorer®, un tel nœud a pour nom (propriété `nodeName`) : "`#text`".

# XSL (Extended Stylesheet Language)

## Processus de composition documentaire

XML et les autres langages abordés dans les paragraphes précédents abordent les différents aspects de la structuration documentaire. Lorsque l'on a créé un document XML bien formé et valide, on est en présence d'une structure de données incluant informations et méta-informations. Il s'agit alors de modifier et de mettre en forme cette structure de données dans un format documentaire adapté à l'utilisation envisagée. Le processus de production documentaire est alors illustré par la figure ci-dessous, extraite de la recommandation XSL du W3C du 15 octobre 2001 (http://www.w3.org/TR/xsl/).




Dans ce cours, il s'agit de générer des documents textuels lisibles par des opérateurs humains. Les formats visés peuvent être du texte simple (TXT) ou mis en forme (RTF), des formats documentaires structurés (PS, PDF) ou des formats adaptés à un média particulier, comme les pages web (HTML) ou les documents sonores (pages aurales). Rien n'empêche cependant de réaliser une telle mise en forme pour d'autres applications qui prendront en entrée des fichiers d'échange de données en XML dont la structure sera différente de celle des fichiers d'origine.

Le but de ce cours n'est pas de traiter tous ces formats. Il se limite aux cas simples de génération de documents textuels ou XHTML. Les techniques « standard » sont présentées dans le cadre d'un exemple d'application : la composition de pages web virtuelles. Il existe bien entendu d'autres techniques qui ne sont pas abordées ici. Notamment, la génération de documents complexes (comme le format pdf d'Adobe™), via le langage de description de pages XSL *Formatting Objects* (XSL-FO) associé à XSLT, dépasse le cadre de ce cours. Nous nous contentons de réaliser l'opération de formatage par l'application de feuilles de style CSS.

## Un exemple de technique de composition documentaire : les DVP

L'objectif de cette partie est de présenter un ensemble de techniques de composition documentaire, connu sous le nom de *Documents Virtuels Personnalisables* (DVP). Ces techniques permettent de générer à la volée des fichiers XHTML visualisables dans un navigateur W3 (ici : Internet Explorer® 5). Ces fichiers correspondent à la notion de document, car ils possèdent une structure de données et des informations de formatage spécifiques. Ils sont virtuels, car générés à la volée et non stockés sur un support physique. Ils sont personnalisables, car la structure de données ainsi que les informations de formatage

peuvent être générées en fonction d'actions des utilisateurs, ce qui permet de créer des documents prenant mieux en compte leurs besoins que les documents « statiques ».

L'ensemble des opérations de mise en forme nécessite trois types d'opérations. Chacune d'elles est réalisée par un outil spécifique. Les outils indiqués ici sont ceux utilisés pour la composition de DVP.

**Récupération des informations et méta-informations :** cette opération permet d'inclure des éléments de la structure de données source dans la mise en forme du document final. Dans le cas d'une structure de données en XML, le langage utilisé pour parcourir cette structure s'appelle *XPath* (XML Path Language).

**Transformation de la structure de données :** il s'agit de convertir les informations et méta-informations sources (i.e. les parties de contenu du document XML d'origine, obtenues par l'opération précédente) en une structure de données de sortie correspondant au format choisi. Techniquement, cette étape s'apparente plus à la structuration qu'à la composition documentaire. Cependant, elle est intégrée au processus de composition car elle n'est pas mise en œuvre si ce processus n'est pas initié. Le langage utilisé pour transformer ces structures de données s'appelle *XSLT* (Extended Stylesheet Language Transformation).

**Application des styles :** c'est l'opération de composition documentaire proprement dite. La technique utilisée est très variable selon l'application (par exemple, dans les milieux de l'édition le langage utilisé est DSSSL, car il permet de spécifier avec une grande précision les caractéristiques des pages). Pour la composition de documents au format W3 (et *a fortiori* de documents XHTML), le standard recommandé par le W3C est *CSS* (Cascading Style Sheets Language). Il existe plusieurs versions du langage CSS. CSS1, dont la recommandation du 17 décembre 1996, révisée le 11 janvier 1999, est disponible à l'adresse http://www.w3.org/TR/REC-CSS1 spécifie la syntaxe de base de la sélection et de la déclaration de règles de style et des mécanismes d'héritage entre ces règles (« mise en cascade ») utilisé depuis HTML. CSS2 (recommandation du 12 mai 1998, http://www.w3.org/TR/REC-CSS2) ajoute à CSS1 la spécification de feuilles de style dépendantes du support (par exemple, les feuilles de styles d'impression ou aurales) et la prise en charge des polices de caractères téléchargeables, du positionnement d'éléments, des tables… CSS3 est en cours de spécifications (adresse du working draft du 23 mai 2001 : http://www.w3.org/TR/css3-roadmap/). L'utilisation des CSS (jusqu'à la version 2) est supposée connue, et nous n'y revenons pas dans ce cours.

Remarques :

- L'association de XPath et de XSLT[12] constitue le langage de feuilles de styles « officiellement » rattaché à XML : *XSL* (Extended Stylesheet Language).
- La séparation des opérations de génération de documents XHTML visualisables avec XSLT et d'application de feuilles de style avec CSS permet de se conformer à la recommandation XHTML 1.0 stricte. Si des informations de style sont incluses dans le document XHTML généré en XSLT (i.e. dans la feuille de style XSLT), on obtient du XHTML transitionnel, et donc imparfait…

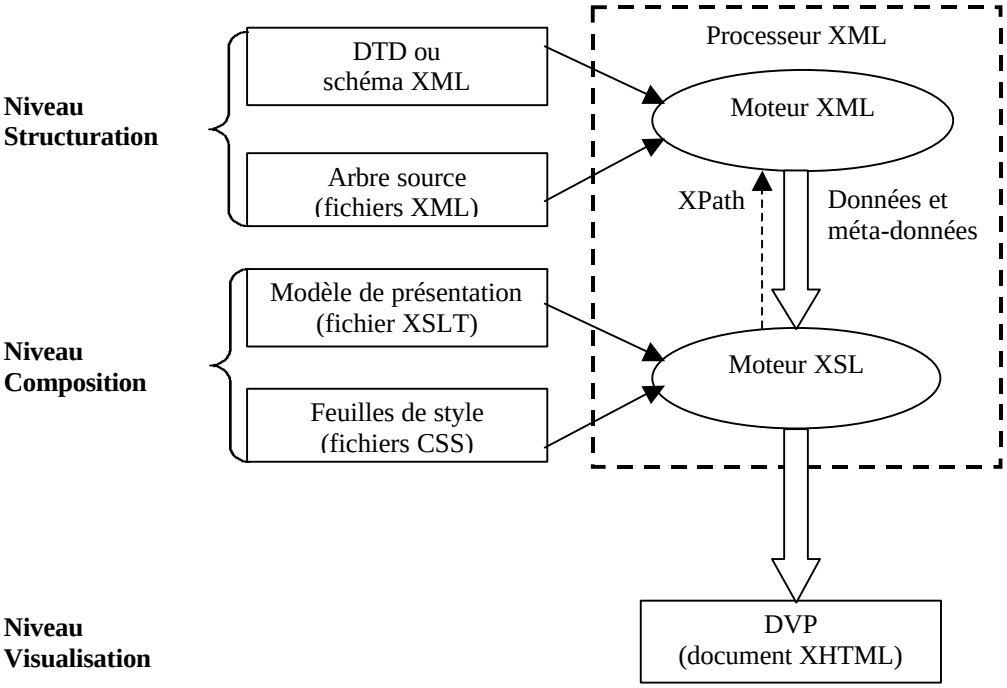
[12] Plus XSL-Formatting Objects.

- Les navigateurs récents (IE5…) prennent en charge l'affichage direct de documents XML reliés à des feuilles de style CSS2, par l'instruction de traitement : `<?XML:stylesheet type="text/css" href="Feuille_De_Style.css"?>`
- La prise en charge des feuilles de style diffère selon l'implémentation qui en est faite dans le navigateur, c'est-à-dire en fonction du navigateur et de sa version.

Chacune des opérations ci-dessus n'est théoriquement pas obligatoire. Le processus de composition documentaire privé d'une de ces étapes aboutit à des résultats plus ou moins utiles. Par exemple, l'opération de récupération des données de l'arbre XML source n'est théoriquement pas toujours nécessaire. On peut cependant s'interroger sur l'intérêt d'utiliser des techniques aussi complexes ici pour générer des documents XHTML figés. Il est également possible de ne pas faire subir de transformations au document XML source. Par exemple, un fichier XHTML est un document XML directement visualisable avec un navigateur W3. Enfin, la génération de fichiers textes ou XML ne requiert pas l'application de feuilles de style.

Les langages de composition sont pris en charge par des applications spécifiques : les *moteurs XSLT*. Ces moteurs sont inclus dans certains processeurs XML. Ci-dessous, on distingue – artificiellement – le moteur XML du moteur XSL inclus dans un processeur XML. Par chance (!), MSXML contient un tel moteur. Un document XML associé à une feuille de style XSL peut donc être directement visualisé dans Internet Explorer™ 5.

Le processus global de génération de DVP est illustré dans la figure ci-dessous.




## XPath

XPath est un outil souple permettant de pointer vers les différents morceaux d'un document XML. La syntaxe de ce langage permet de définir des *expressions XPath* qui sont utilisées :

- pour le parcours de documents XML,
- pour le test de valeurs associées aux contenus ou aux attributs d'éléments.

La syntaxe de ce langage est volontairement différente de celle de XML. Cela est dû au fait que les expressions XPath sont principalement utilisées dans des attributs d'éléments XSLT (qui eux respectent la syntaxe XML). La syntaxe XPath a donc été définie pour ne pas « perturber » l'analyse des feuilles de style XSLT par le parser XML. En particulier, elle ne fait – pratiquement – pas appel au caractère de début de balise '<'.

La recommandation du W3C date du 16 novembre 1999. Elle est disponible à l'URL : http://w3.org/TR/REC-xpath/.

### *Syntaxe élémentaire des expressions de parcours*

Dans un premier temps, XPath permet de spécifier des *chemins de localisation* de parties de document. Pour cela, ce langage utilise le concept de *nœud*.

**Un nœud XPath** désigne une partie quelconque de l'arborescence d'un document, qu'il s'agisse d'un élément, d'un contenu ou d'un attribut. Une expression permet également de désigner des *ensembles de nœuds*, par des propriétés qu'ils ont en commun (par exemple, l'ensemble de tous les éléments possédant un attribut « id »). Des nœuds particuliers sont la *racine du document* et le *nœud contextuel*.

**La racine du document** désigne en XPath une entité qui ne pointe vers aucun des éléments du fichier XML et qui a pour enfant l'élément racine du document. Cette entité est désignée par le caractère '/' (slash) en début d'expression.

**Le nœud contextuel** est la position courante dans le document. La sélection d'un nœud contextuel est présentée plus loin. Ce nœud contextuel est désigné par '.' (point).

**Un chemin de localisation XPath** peut être absolu, relatif ou récursif. Dans le premier cas, il part de la racine du document. Dans le deuxième, il part du nœud contextuel. Dans le troisième cas, le moteur XSLT parcourt la totalité du document à la recherche de l'expression cherchée. L'opérateur de descente dans l'arborescence de l'arbre est désignée par le caractère '/' (slash). L'opérateur de descente récursive dans l'arborescence est désignée par les caractères "//" (double slash). Par défaut, un chemin de localisation est considéré comme relatif.

Remarque : l'opération de descente récursive a tendance à dégrader sérieusement les performances des feuilles de style. Quand cela est possible, il est préférable d'éviter de l'utiliser.

**Un attribut** est désigné en XPath par le caractère '@'. En XPath, un attribut est considéré comme un sous-nœud de l'élément XML auquel il se rapporte. Bien entendu, un chemin de localisation XPath comportant un opérateur de descente (récursive ou non) suivant ce caractère est invalide.

Il existe aussi un **caractère générique** "*" qui désigne n'importe quelle valeur ou élément non vides.

Exemples :

`/` désigne l'ensemble du document XML.

`./Etat_civil/Nom` (ou `Etat_civil/Nom`) désigne les éléments `<Nom>` enfants des éléments `<Etat_civil>`, eux-mêmes enfants du nœud contextuel.

`Etat_civil/*` désigne tous les éléments enfants des éléments `<Etat_civil>`, eux-mêmes enfants du nœud contextuel.

`/racine/@xmlns` désigne l'attribut `xmlns` (rappel : espace de noms par défaut) de l'élément racine du document.

`//xhtml:div/@id` désigne les attributs `id` de tous les éléments `<html:div>` où qu'ils soient situés dans l'arborescence du document XML.

`.//enfant` désigne tous les éléments `<enfant>` descendant du nœud contextuel.

### *Filtrage des chemins de localisation*

Les exemples précédents permettent de sélectionner des nœuds en fonction de leur ascendance. Il est possible de définir des règles de sélection prenant également en compte la descendance des nœuds. Pour cela, la syntaxe XPath permet d'utiliser des *filtres de localisation*, encadrés par les caractères '`[`' et '`]`' (crochets). Ces filtres peuvent porter sur l'existence d'un nœud « enfant » dans la descendance du nœud à sélectionner ou sur la valeur d'un tel enfant.

Exemples :

`Etat_civil[Nom]` désigne tous les éléments `<Etat_civil>` enfants du nœud contextuel et ayant au moins un élément enfant `<Nom>`.

`Etat_civil[Nom = 'toto']` désigne tous les éléments `<Etat_civil>` enfants du nœud contextuel et ayant au moins un élément enfant `<Nom>` dont la valeur est « toto ».

`Etat_civil/Nom[. = 'toto']` désigne tous les éléments `<Nom>` dont la valeur est « `toto` », enfants d'éléments `<Etat_civil>`, eux-même enfants du nœud contextuel.

`//xhtml:div[@id]` désigne tous les éléments `<html:div>` où qu'ils soient situés dans l'arborescence du document XML, pourvu qu'ils aient un attribut « id ».

`//xhtml:div[@id = 'div1']` désigne l'élément `<html:div>` où qu'il soit situé dans l'arborescence du document XML, dont la valeur de l'attribut `id` est « `div1` » (rappel : un attribut `id` a par définition une valeur unique dans le document).

### *Les fonctions XPath*

À des fins de puissance et d'efficacité, XPath possède un certain nombre de *fonctions* pouvant être utilisées dans des expressions XPath. Comme une expression XPath doit toujours

renvoyer un nœud, les fonctions qui n'ont pas ce type de valeur de retour sont toujours utilisées à l'intérieur de filtres de localisation. Ces fonctions sont de la forme :
`Nom_De_Fonction ( [Arguments] )`

Il existe plusieurs types de fonctions.

Les fonctions de nœuds, qui permettent de travailler avec des nœuds en général. Ces fonctions sont : `name()`, `node()`, `processing-instruction()`, `comment()` et `text()`.

Les fonctions de position, qui permettent de déterminer ou d'identifier la position d'un nœud dans un ensemble de nœuds. Ces fonctions sont : `position()`, `last()` et `count()`.

Les fonctions numériques, qui ne sont pas énumérées ici. La plus couramment utilisée est `number()`, qui permet de convertir la valeur d'un nœud (PCDATA) en nombre. Par exemple : `article[number(prix) < 1000]` sélectionne tous les éléments `<article>` dont un élément enfant `<prix>` a pour valeur un nombre inférieur à 1000.

Les fonctions booléennes, dont les plus courantes sont `boolean()`, `not()`, `true()` et `false()`.

Les fonctions de traitement de chaînes, parmi lesquelles `string()`, `string-length()`, `concat()`, `contains()`, `substring()` ou encore `starts-with()`.

### *Axes de navigation*

XPath ne permet pas seulement de se déplacer dans l'arborescence du document XML par des opérations de descente et de remontée. L'utilisation d'*axes XPath* permet des déplacements plus complexes dans cette arborescence. Les axes XPath sont la généralisation de la notion de chemin de localisation. Certains de ces axes possèdent des abréviations, qui sont celles que nous avons déjà vues dans la description des chemins de localisation. La syntaxe appropriée pour leur utilisation dans des expressions XPath est : `Nom_D'Axe::Nom_De_Nœud`.

Il existe 13 axes XPath.

| **Nom d'axe** | **Description** | **Exemple d'utilisation / syntaxe abrégée** |
|---|---|---|
| `self` | Nœud contextuel | `self ::node()` ou `./node()`ou `.` |
| `child` | Enfants du noeud contextuel (par défaut) | `child::Etat_civil` ou `Etat_civil` |
| `descendant` | Tout enfant, petit-enfant etc. du nœud contextuel | `descendant::Etat_civil` |
| `descendant-or-self` | Comme descendant + le nœud contextuel lui-même | `descendant-or-self::Etat_civil` ou `.//Etat_civil` |

| | | |
|---|---|---|
| `parent` | Parent du noeud contextuel | `//Nom/parent::Prenom` ou `//Nom/../Prenom` |
| `ancestor` | Tout parent, grand parent etc. du noeud contextuel | `ancestor::Prenom` |
| `ancestor-or-self` | Comme parent + le noeud contextuel lui-même | `ancestor-or-self::Prenom` |
| `following-sibling` | Tous les frères suivants du noeud contextuel (vide si le noeud est un attribut) | `following-sibling::Nom` |
| `preceding-sibling` | Tous les frères précédents du noeud contextuel (vide si le noeud est un attribut) | `preceding-sibling::Prenom` |
| `following` | following–sibling + descendants de tous les noeuds frères suivants. | `following::Nom` |
| `preceding` | preceding–sibling + descendants de tous les noeuds frères précédents. | `preceding::Prenom` |
| `attribute` | Attributs du noeud contextuel | `attribute::id ou ./@id` |
| `namespace` | Tous les noeuds appartenant au même espace de noms que le noeud contextuel | `namespace::xhtml:div` |

## XSLT

XSLT est un langage de programmation déclarative qui permet de décrire directement le contenu des fichiers de sortie, issus de la « transformation » des données et des méta-données XML. Sur le principe, une feuille de style XSLT est un document XML rattaché à un espace de noms spécifique qui définit des *balises de transformation* du contenu d'un document XML source à l'intérieur d'une série de *modèles* (templates) imbriqués. La recommandation du W3C concernant la version 1.0 de ce langage est datée du 16 novembre 1999 et est disponible à l'adresse : http://www.w3.org/TR/xslt.

L'élément racine d'une feuille de style XSLT est un élément `<stylesheet>`, qui a pour attributs la déclaration du numéro de version de XSLT utilisé et l'identification de l'espace de noms correspondant. Le préfixe recommandé pour cet espace de noms est « `xsl` ». Une déclaration « standard » de feuille de style XSLT est la suivante.

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

Remarques :

- Les feuilles de style étant des documents XML, si elles sont dans des fichiers à part, elles n'échappent pas à la déclaration XML traditionnelle.
- L'élément `<stylesheet>` est destiné à contenir la totalité des modèles XSLT de la feuille de style. Il n'est toutefois pas obligatoire, dans le cas où un seul modèle est

défini dans cette feuille. Dans ce cas, il est possible d'indiquer la version de XSLT utilisée et l'espace de noms directement à la racine de l'arbre destination :

```
<racine_dest     xsl:version="1.0"
                 xmlns:xsl=”http://www.w3.org/1999/XSL/Transform”>
```

Bien entendu, ce genre de simplification n'est pas recommandé pour la « compréhensibilité » du code.

- En TP, le processeur XML utilisé (i.e. de MicroSoft™) ne reconnaît pas l'espace de noms ci-dessus comme indicatif de l'activation du moteur XSLT. Ce moteur est activé par la déclaration suivante :

```
<xsl:stylesheet xmlns:xsl=”uri:xsl”>
```

Les éléments enfants directs de `<xsl:stylesheet>` sont appelés *éléments de niveau supérieur*. En théorie, il ne peut y avoir que deux types d'éléments de niveau supérieur : une balise `<xsl:output>` et des modèles. En pratique, un modèle peut être implicite, si bien que des éléments enfants du modèle peuvent être enfants d'une balise `<xsl:stylesheet>`.

`<xsl:output>` est une balise vide qui permet de spécifier le type de document généré en sortie de la feuille de style. Cela est réalisé par l'attribut `method` de cette balise, qui peut prendre « en standard » les valeurs `xml`, `html` ou `text`. D'autres valeurs spécifiques peuvent être acceptées par certains processeurs. En fonction de la valeur de l'attribut `method`, d'autres attributs peuvent être spécifiés. Ces attributs sont : `version`, `encoding`, `omit-xml-declaration`, `standalone`, `cdata-section-elements` ou `indent`. De tels attributs permettent par exemple de générer la déclaration XML dans le cas où ce format de sortie a été choisi par l'attribut `method`.

Remarque : le processeur MSXML étant destiné à visualiser les documents XML à l'intérieur d'Internet Explorer©, celui-ci n'admet qu'une seule valeur de l'attribut `method` : `html`.

### *Les modèles XSLT*

Hormis la balise `<xsl:output>`, une feuille de style XSLT est constituée d'une succession de modèles XSLT. Chaque modèle spécifie ce qui doit être recherché dans l'arbre source et ce qui doit être placé dans l'arbre cible. Les définitions de modèles ne peuvent s'imbriquer les unes dans les autres. En revanche, il est possible d'appeler un modèle depuis la définition d'un autre.

Le contenu d'un modèle est encadré par une balise `<xsl:template>` qui possède un attribut `match`. La valeur de cet attribut est une expression XPath qui sert à sélectionner un ensemble de nœuds dans le modèle XPath de l'arbre source. Le modèle décrit peut alors s'appliquer à chacun de ces nœuds. Dans la description du modèle, on considère tout nœud correspondant à la valeur de l'attribut `match` comme le nœud contextuel à partir duquel est défini le modèle.

Le « modèle de départ » de toute feuille de style XSLT est associé à la racine XPath du document. En d'autres termes, la racine du document transformé doit être décrite à l'intérieur d'une balise `<xsl:template match="/">`. Lorsqu'aucun modèle n'est spécifié, le processeur utilise le *modèle caché par défaut* défini dans la balise ci-dessus.

Lorsque l'attribut match ne suffit pas à déterminer quel modèle doit être appliqué, il existe un attribut mode, permettant de différencier des modèles s'appliquant aux mêmes types de contenus. Dans ce cas, il faut préciser, lors de l'application du modèle, le mode d'application choisi. Il est également possible de laisser le processeur choisir un modèle à appliquer. Par défaut, lorsque deux modèles sont applicables, le processeur applique le dernier dans l'ordre de la feuille de style, sauf si :

- l'un est plus spécifique que l'autre ; c'est notamment le cas quand un filtre XPath correspondant à un élément de l'arbre source a été utilisé dans l'un des attributs `match` des balises de définition des modèles,
- une priorité est explicitement spécifiée par un attribut `priority` ; dans ce cas, c'est le modèle de plus forte priorité qui est choisi.

Remarque : l'avantage de la seconde solution est que la valeur de cet attribut est modifiable dynamiquement, comme celle de tout document XML, par exemple via le DOM.

**L'application de modèles XSLT** se fait avec la balise `<xsl:apply-templates>`. Cette balise permet alors d'appeler un ou plusieurs modèles depuis la description d'un autre modèle. Pour cela, la sélection du ou des modèles à appeler se fait grâce à une expression XPath dans un attribut `select`. Cette expression est évaluée, et le résultat est utilisé comme nœud contextuel s'il correspond à l'attribut `match` d'une balise `<xsl:template>`. L'attribut `mode` de la balise `<xsl:apply-templates>` peut être utilisé pour spécifier un modèle particulier, comme indiqué plus haut.

Remarques :

- le contenu de l'attribut `select` est plus précis que celui de l'attribut match ; il permet de spécifier un chemin d'accès à un nœud, alors que celui de `match` détermine la condition d'application du modèle,
- l'attribut `select` est facultatif ; dans le cas où il est omis, le processeur applique les modèles sur les balises, les PCDATA, les commentaires et les instructions de traitement contenus dans la balise courante,
- dans tous les autres cas, l'appel de modèles peut être récursif ; les processeurs doivent être capables de gérer cette récursivité, et de détecter d'éventuels problèmes de boucles infinies.

Exemple :

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/ Transform">
     <xsl:template match="/">
          ...
          <xsl:apply-templates select="/CV/Etat_civil/Nom"/>
     ...
     </xsl:template>

     <xsl:template match="Nom">
          Nom trouvé
     </xsl:template>
</xsl:stylesheet>
```

L'exemple ci-dessus affichera le texte « Nom trouvé » à chaque fois qu'il trouvera une balise <Nom>, enfant d'une balise <Etat_civil>, enfant de <CV>, lui-même enfant de la racine XPath du document source.

**Les modèles nommés** permettent l'appel d'un modèle indépendamment de son contenu. Pour cela, il suffit d'utiliser l'attribut name dans la balise de déclaration du modèle <xsl:template> et dans l'instruction d'appel <xsl:call-template>.

### *Le traitement des contenus*

La principale balise permettant d'accéder au contenu textuel d'un élément XML est la balise <xsl:value-of>. Elle s'emploie avec l'attribut select, qui définit le nœud XPath dont le contenu sera affiché. En remplaçant la ligne : « Nom trouvé » dans l'exemple ci-dessus par la ligne : « Nom : <xsl:value-of select="."/> », on obtient, pour chaque élément <Nom> correctement situé dans l'arbre source, le texte « Nom : » suivi du contenu de cet élément.

L'élément <xsl:copy-of> permet de recopier des parties entières de l'arbre source (y compris les attributs et les enfants). Il s'utilise avec l'attribut select, suivi comme il se doit d'une expression XPath. Il est principalement utilisé lorsque le format de sortie de la feuille de style est XML.

L'élément <xsl:copy> s'utilise à partir du nœud contextuel et permet un traitement plus riche qui n'est pas détaillé ici.

### *Les structures de contrôle XSLT*

Les balises de contrôle XSLT permettent de réaliser plusieurs types de tests sur le contenu de l'arbre source. En fonction des résultats de ces tests, elles permettent de spécifier différents blocs de contenus pour l'arbre transformé.

L'élément <xsl:if>, employé avec l'attribut test permet de spécifier un contenu conditionnel, comme dans l'exemple suivant :

```
<xsl:if test="./Nom">
     Nom : <xsl:value-of select="Nom"/>
</xsl:if>
```

Cet exemple affichera « Nom : » et la valeur du nœud <Nom> uniquement si un tel nœud existe parmi les enfants du nœud contextuel.

L'élément <xsl:choose> permet de définir plusieurs alternatives. Il contient des éléments <xsl:when> définissant chacun une condition dans un attribut test et éventuellement un élément <xsl:otherwise> qui indique un contenu valide si aucun des tests des éléments <xsl:when> précédents n'a été évalué comme true.

L'élément <xsl:for-each> suivi de l'attribut select permet de définir une mise en forme à appliquer à tout élément correspondant à l'expression XPath. En d'autres termes, il est équivalent à l'élément <xsl:apply-templates>, mais rend la structure de la feuille de style moins lisible. On lui préfèrera donc ce dernier élément.

# Conclusion (transparents issus du cours)



## Conclusion

- Aspects abordés dans le cours
  - Structuration documentaire
    - Données et méta données : XML et les espaces de noms
    - Description des structures de données : DTD et schémas
  - Composition documentaire
    - Transformation des structures de données : XSLT et XPath
    - Mise en forme des documents : CSS
    - Visualisation : HTML et XHTML





## Perspectives

- Maintenant que vous avez les documents…
  - Stockage de l'information
  - Accès à l'information
    - Recherche d'information
    - Hypertextes et hypermédias