

---

---

R T F M\*

Retour  
vers les  
Techniques Fonctionnelles  
et la  
Modélisation

---

L3 MIAGE



Université Grenoble Alpes  
Nicolas.Glade@univ-grenoble-alpes.fr

Copyright ©2024–2024 Nicolas Glade, PhD

Ce cours a été rédigé par Nicolas Glade.

La partie XML est très largement inspirée du cours de XML d’Emmanuel Promayon, PhD, enseigné à Polytech’Grenoble et du cours de Formalisation des données - Technologies XML de Céline Fouard, PhD, et Nicolas Glade, PhD, enseigné en L3 Miage.

Certaines parties sont inspirées de *XML Cours et exercices*, Alexandre Brillant, éditions Eyrolles et *XSLT, Mastering XML Transformations*, Doug Tidwell, O’Reilly editions.

Les références des exercices sont donnés dans le texte.

Si vous souhaitez utiliser ce document, merci de contacter [Nicolas.Glade@univ-grenoble-alpes.fr](mailto:Nicolas.Glade@univ-grenoble-alpes.fr)

\* RTFM : Read This Fantastic Manual !

# Table des matières

<b>Organisation du cours</b>	<b>1</b>
1 Objectifs du cours . . . . .	1
2 Pourquoi XML ? . . . . .	1
3 Pourquoi C# .Net ? . . . . .	2
4 Interface de Développement Intégrée . . . . .	2
5 Bibliothèque ReactiveX . . . . .	3
6 Contrat Pédagogique . . . . .	3
7 Horaires et Organisation . . . . .	3
8 Évaluation . . . . .	3
<b>I Introduction aux langages et à la programmation</b>	<b>5</b>
1 Programmer . . . . .	5
2 Langages informatiques . . . . .	6
<b>II Formalisation - Introduction</b>	<b>13</b>
1 Formaliser / Modéliser . . . . .	13
2 Les données et les langages de formalisation . . . . .	17
3 JSON - un langage de modélisation et stockage des données . . . . .	21
4 XML - un langage de modélisation et stockage des données . . . . .	22
<b>III Formalisation - Le "langage" XML</b>	<b>27</b>
1 Éléments et attributs . . . . .	27
2 Les 8 points clés de XML . . . . .	28
3 Document bien formé . . . . .	33
<b>IV Formalisation - But des Schemas XML</b>	<b>35</b>
1 Objectifs . . . . .	35
2 Spécifier . . . . .	36
<b>V Formalisation - XMLSchema : Schema et Instance</b>	<b>43</b>
<b>VI Formalisation - Structure d'un XMLSchema</b>	<b>49</b>
1 Conventions de nommage en vigueur dans ce cours . . . . .	49
2 Exemple . . . . .	49
3 Élément racine . . . . .	50
4 Types de Données . . . . .	51
5 Attributs . . . . .	58
6 Occurrences et Cardinalités . . . . .	58

<b>VI</b>	<b>Formalisation - Espaces de nom en XMLSchema</b>	<b>63</b>
1	Un schéma = un vocabulaire . . . . .	63
2	Exemple des températures . . . . .	63
3	Définition du vocabulaire et de son nom . . . . .	64
4	Contraindre un document XML à un schema . . . . .	67
5	A qui appartiennent les éléments/attributs ? . . . . .	69
<b>VII</b>	<b>Contraintes de cohérence : Unicité et Existence</b>	<b>71</b>
1	Contraintes d'unicité . . . . .	71
2	Contraintes d'existence . . . . .	79
<b>IX</b>	<b>XPath : Naviguer dans un document XML</b>	<b>83</b>
1	Node-set . . . . .	84
2	Les axes de recherche . . . . .	84
3	Sélectionner des parties de l'arbre . . . . .	86
4	Ecrire correctement du XPath . . . . .	88
<b>X</b>	<b>XSLT: transformations en XML</b>	<b>91</b>
1	Principes de fonctionnement . . . . .	92
2	Mécanisme <i>Push</i> . . . . .	94
3	Mécanisme <i>Pull</i> . . . . .	95
4	Mécanisme de navigation . . . . .	96
5	Programmation . . . . .	99
<b>XI</b>	<b>Mise à niveau C#</b>	<b>103</b>
1	Le framework DotNet . . . . .	103
2	Création d'un projet C# . . . . .	103
3	C# - vue d'ensemble . . . . .	104
4	Ecrire en C# . . . . .	105
<b>XII</b>	<b>De la modélisation aux traitements</b>	<b>123</b>
1	Statique/Dynamique . . . . .	123
2	Équivalence XML Schema ↔ Java / C# . . . . .	124
<b>XIII</b>	<b>Analyser un document XML - Les Parsers</b>	<b>127</b>
1	<i>Document Object Model</i> : DOM [Java et C#] . . . . .	128
2	Forward parsers . . . . .	139
3	<i>Simple Api for Xml</i> : SAX [Java] . . . . .	140
4	<i>STreaming Api for Xml</i> : StAX [Java] . . . . .	144
5	La classe <code>XmlReader</code> [C#] . . . . .	147
6	Résumé des différents <i>parsers</i> . . . . .	153
<b>XIV</b>	<b>Data Binding - Sérialisation</b>	<b>155</b>
1	Classes et schémas XML - Equivalence . . . . .	155
2	Principes du Data Binding, de la sérialisation et désérialisation . . . . .	156
3	Data Binding : <i>Design-time</i> ou <i>Run-time</i> . . . . .	157
4	Avantages et limitations . . . . .	157
5	Data Binding et Sérialisation en Java . . . . .	158
6	Data Binding et Serialisation en C# . . . . .	167
7	Personnalisation de la compilation de Schema XML . . . . .	168
8	Compilateurs XML Schema / sérialisation / persistance . . . . .	175
9	Références . . . . .	176
<b>XV</b>	<b>Programmation fonctionnelle - Introduction</b>	<b>177</b>
1	Qu'est ce que la programmation fonctionnelle ? . . . . .	177
2	Avantages et inconvénients . . . . .	177
3	Caractéristiques . . . . .	177
4	Conséquences . . . . .	177

<b>XVI</b>	<b>Programmation style fonctionnelle en C#</b>	<b>179</b>
1	Immuabilité . . . . .	179
2	Fonctions et lambdas . . . . .	179
3	Effets de bord - Fonction pure . . . . .	179
4	Fonctions d'ordre supérieur . . . . .	179
5	L'API LINQ . . . . .	179
<b>XVII</b>	<b>Programmation asynchrone</b>	<b>181</b>
1	Principe général . . . . .	181
2	En C# . . . . .	181
3	L'intérêt de la programmation fonctionnelle en asynchrone . . . . .	181
<b>XVIII</b>	<b>Programmation réactive</b>	<b>183</b>
1	Principe . . . . .	183
2	L'API ReactiveX . . . . .	183



# Organisation du cours

## 1 Objectifs du cours

Ce cours de *Formalisation des Connaissances et Programmation Fonctionnelle* s'attèle à vous enseigner comment:

- Formaliser des informations
- Organiser et décrire des données puis des objets complexes (typage et représentation objet)
- Utiliser les technologies XML pour structurer, décrire, stocker, transformer, utiliser ces données
- Ecrire un programme sécurisé, testable, en utilisant les paradigmes de la programmation fonctionnelle.

Ce cours utilise la norme XML pour stocker les données. UML et XMLSchema sont utilisés pour décrire et organiser les données. En se basant sur UML et XMLSchema, le lien avec la Programmation Orientée Objet (POO) est simplifié.

Notez que nous utiliserons .Net (DotNet) C# comme langage pour rendre opérationnels nos projets utilisant XML et implémenter les paradigmes de la programmation fonctionnelle. Ce cours ne constitue en aucun cas un cours avancé de programmation C#. Notez aussi que le cours contient des exemples d'utilisation des technologies XML en Java, son API présentant quelques différences importantes avec C#. Là encore, il ne s'agit pas d'un cours de Java. Vous aurez cependant des rappels réguliers de C# et Java au cours des CTD et TP.

## 2 Pourquoi XML ?

*Pourquoi XML plutôt que d'autres formats comme JSON, les expressions régulières ... ?.*

Le point important, est que cet enseignement vise à vous faire prendre de bonnes habitudes de programmation, à savoir de savoir modéliser les données proprement et ensuite de savoir les utiliser. Le Web est effectivement complètement centré sur le stockage et l'échange de données. Le reste "n'est qu'interface et fonctionnalisation". C'est le cas de nombreuses autres applications. Dans ce cours, nous verrons comment interagir avec ces données, les transformer et les rendre disponibles, en utilisant Java ou C#.

Maintenant, à la question de pourquoi XML et pas JSON ou autre: en fait cela dépend un peu du problème que l'on traite. Je vous renvoie pour cela à [ce lien fort clair et fort bien documenté](#). Cependant, il apparaît qu'XML est adapté à l'objectif principal de ce cours (formalisation, structuration des données, mais également navigation, usage ...). XML dispose d'atouts moins marqués ou absents en JSON :

- XML permet (en XML Schema) de décrire des types complexes, incluant des notions d'héritage ou des contraintes de cohérence, et de contrôler la validité des données par rapport à ces types ; JSON permet le typage des données mais peu complexe (voir [JSON Schema](#)).
- XML permet l'usage d'espaces de nom (namespaces) qui autorisent l'usage combiné de plusieurs langages XML différents utilisant un vocabulaire commun ; JSON ne le permet pas.

- Les documents XML sont auto documentés ; on sait à quel schéma ils doivent se conformer. En JSON, c'est le code appelant qui met en regard instance JSON et schema JSON pour s'assurer de la validité.
- La lecture directe, par le programmeur ou un utilisateur, d'un fichier XML est (en général) plus intelligible que celle d'un fichier JSON.

### 3 Pourquoi C# .Net ?

Le choix d'un langage particulier pour ce module a été compliqué. Il résulte de multiples contraintes. Il fallait que :

- le langage soit objet et assez proche de Java (pour lequel vous avez des cours)
- le langage soit suffisamment moderne pour être employé en entreprise
- le langage évite la lourdeur de l'apprentissage d'un langage très complexe comme C++ moderne
- le typage dans ce langage soit plus avancé que celui de Java
- le langage dispose d'une API performante permettant d'utiliser XML
- le langage puisse permettre une écriture dans un style fonctionnel, donc puisse correctement prendre en compte les critères nécessaires pour une écriture en langage fonctionnel : immuabilité, fonctions pures, fonctions d'ordre supérieur, notion de foncteur ... L'alternative étant l'usage de Java ou C++ combinée à un langage de programmation fonctionnelle comme Haskell, Purescript, Scheme ou Racket (du Lisp).
- le langage soit multi-plateforme
- le langage dispose d'une bibliothèque permettant le jeu vidéo (pour le projet)

Le langage C# (C Sharp) appartenant au framework DotNet développé par Microsoft, répond à toutes ces contraintes. C'est un langage moderne qui permet une écriture dans un style très élégant. Par dessus tout, dans ce même framework DotNet existe **un langage nommé F#** qui est un langage de programmation fonctionnelle pur. L'intérêt du framework DotNet est de permettre l'inter-opérabilité de ces langages : il est tout à fait possible de mélanger du code F# dans du C# !!!

### 4 Interface de Développement Intégrée

Parmi les IDE disponibles pour C#, celles qui ressortent sont :

- VS Code (sous Windows, Linux et Mac)
- Visual Studio (Windows seulement)
- JetBrains Rider

C'est ce dernier que nous utiliserons. Je déconseille fermement l'usage de VS Code. VS Code n'est pas à proprement dit un IDE, mais seulement un éditeur de texte-code avancé.

JetBrains Rider est une solution non publique mais qui autorise un usage gratuit pour les universitaires et étudiants. Vous trouverez la procédure d'obtention d'une licence et d'installation ici : **Licence et installation JetBrains** Nous utiliserons Monogame pour écrire le jeu vidéo qui constituera votre projet à rendre. Monogame est un framework et non un moteur de jeu comme Unity ou Unreal. Il permet cependant de gérer l'affichage d'objets, de gérer une boucle de jeu et ses callbacks, de gérer les contrôles (clavier, souris ...), etc.

Nous verrons son usage en TP mais vous trouverez ici un bon tutoriel, suffisamment simple et court pour rapidement utiliser Monogame : **Tutoriel Monogame [video]** de la chaîne **Coding With Sphere**.

**Le site de référence de Monogame** vous permettra d'installer le framework. Vous y trouverez également des **tutoriels** et la **doc officielle**.



## 5 Bibliothèque ReactiveX

Vers la fin du projet, nous nous essayerons à la programmation réactive (utilisant la programmation fonctionnelle) en C#, via le [framework ReactiveX](#). Il vous faudra l'installer (cloner le dépôt github).

Vous trouverez un exemple visuel d'application de ReactiveX ici : [Marbles in RX](#)

Côté tutos, je vous renvoie vers 2 sites :

- [Introduction to RX](#)
- [Introducing RXJava](#)

Le deuxième tuto traite de RXJava, mais l'API est très proche de RXC#.

## 6 Contrat Pédagogique

Le cours de *Formalisation des Connaissances et Programmation Fonctionnelle* a une forme un peu particulière. Il est composé d'un **travail préparatoire** que vous devez faire **AVANT** de venir aux sessions de Cours-TD (CTD).

En effet, chaque semaine, il vous sera demandé d'étudier (pas seulement lire rapidement) le cours pour le prochain CTD. Vous devez, pour chaque CTD:

1. **Lire et Apprendre** les chapitres de cours fournis au format pdf.
2. **Écouter / Regarder / Comprendre / Connaître** les éventuels exemples / animations fournis avec le cours
3. **Vérifier vos connaissances** en faisant l'exercice corrigé du photocopié de TD (sans regarder la correction avant de l'avoir fait) et la vérification de vos réponses
4. **Préparer** d'éventuelles questions à poser en CTD

## 7 Horaires et Organisation

La promotion est divisée en plusieurs groupes équilibrés en effectif. Il est éventuellement possible pour un étudiant de changer de groupe, mais il doit trouver un camarade de l'autre groupe qui veut bien échanger avec lui.

Les dates et horaires sont données à la rentrée et sur le site du cours.

## 8 Évaluation

La note finale de l'UE *Formalisation des données* sera composée

- pour 60% de la note de l'examen final
- pour 40% de la note de contrôle continu, elle-même composée de
  - de la notes d'éventuels QCM faits en début de CTD
  - de la note du Projet Jeu (rendu code C# + XML/XSD/... + rapport + présentation)
  - de la note du Projet Cabinet Infirmier (rendu code C# + XML/XSD/... + rapport + présentation)



# Introduction aux langages et à la programmation

## 1 Programmer

Que signifie “programmer” ?

Il s’agit évidemment de transcrire une intention humaine de calcul ou d’actions dans un langage approprié, sous la forme d’une suite d’instructions, afin qu’un ordinateur réalise ces calculs et actions.

C’est la description usuelle. Mais il faut ajouter à cela une autre intention que doit avoir le programmeur lorsqu’il écrit un programme : il faut que celui-ci soit lisible, intelligible, testable ... autant de contraintes qui relèvent de la qualité du code et qui doivent être comprises dans l’action de programmer.

Etant donné un langage de programmation, il y a toujours de très nombreuses manières d’aboutir à un programme fonctionnel. Il y en a cependant assez peu qui répondent à l’ensemble de ces contraintes de bonne pratique de programmation. Un programmeur averti utilise chaque instruction à bon escient, utilise des styles de programmation standards, utilise des schémas (design patterns) établis, spécifie au maximum chaque élément de son programme (est-ce que cet objet est constant ? est-ce que j’ai le droit de le copier ? est-ce qu’il s’agit d’une redéfinition ? ...), ... et doit choisir avec soin son langage pour l’usage qu’il lui destine.

Son objectif est de ne pas laisser la possibilité à tout programmeur (un autre comme lui-même) d’utiliser son programmes ou ses éléments pour autre chose que ce pour quoi il a/ils ont été conçu-s. Un programmeur-utilisateur qui disposerait de trop de libertés dans l’usage des éléments d’un programme risquerait de provoquer crashes ou pire, comportements indéterminés.

Nous verrons que le typage des données, des objets, lorsqu’il est bien fait, réduit ces problèmes, en particulier dans les langages impératifs. Nous verrons aussi ce qu’apporte le paradigme de la programmation fonctionnelle.

Ce petit topo a pour objectif de rappeler une chose : programmer ne signifie pas juste “apprendre un paquet d’instructions et les sortir dans le bon ordre pour réaliser quelque chose de fonctionnel”, mais aussi sécuriser un programme, s’assurer qu’il termine, que chaque fonction termine, qu’il soit intelligible en tout point, ... C’est vous programmeurs qui, par la façon dont vous présentez votre code, dont vous construisez votre programme, par les algorithmes que vous implémentez, rendez explicites les intentions que vous avez !

Cependant ... cependant, vous découvrirez avec le temps qu’il y a autant de façons de programmer que d’usages. Faire du calcul scientifique ne requiert en général pas la même exigence que concevoir un logiciel doté d’IHM ou une API. Dans le calcul scientifique par exemple, le programmeur n’allouera pas un temps considérable à l’écriture d’un code ultra-sécurisé et utilisera des langages comme Python qui autorisent de prendre de libertés avec la qualité du code. Ces bonnes pratiques que nous allons commencer à vous enseigner doivent pourtant rester une force de rappel pour éviter les problèmes.

## 2 Langages informatiques

### 2.a Types et usages des langages informatiques

Il existe de très nombreux langages de programmation informatique. Ils peuvent être classés selon leur paradigme principalement :

- **langages impératifs:** BASIC, Fortran, Pascal, C, Ada, etc.
- **langages impératifs orienté objet:** Java, C++, C#, etc.
- **langages déclaratifs et fonctionnels:** Lisp (Scheme, Racket ...), F#, Caml, Haskell, Prolog, ASP, etc.
- **langages de structuration/description:**  $\text{\LaTeX}$ , HTML, XML, JSON, etc.

#### Programmation impérative

Un programme **impératif** est une **machine à états (mutables)**. On décrit la succession d'opérations composées de séquences d'instructions modifiant les états (**variables mutables**).

Dans un langage **impératif** on dit à l'ordinateur **quoi faire (what to do)**.

Dans l'exemple ci-dessous, ce programme impératif (en Python) calcule et affiche un automate cellulaire 1D. Il implémente la règle 30. Ce programme contient (volontairement) des effets de bord (nous verrons plus tard ce que c'est, quand nous aborderons la programmation fonctionnelle). Les effets de bord dans ce programme sont l'utilisation dans 2 fonctions (qui ne sont donc pas des fonctions pures) d'une variable externe aux fonctions (`nb_x`).

```

1 import numpy as np \# pour créer un tableau 2D (espace / temps)
2 import matplotlib.pyplot as plt \# pour l'affichage graphique
3 nb_x = 300 \# nombre de colonnes = x l'espace
4 nb_t = 100 \# nombre de lignes = t le temps
5 x_t = np.zeros(shape=(nb_t,nb_x),dtype=float)
6 x_t[0,49] = 1 \# on ne fait qu'une seule modification de la ligne de 0 : un 1
   au centre
7 x_t[0,99] = 1 \# on ne fait qu'une seule modification de la ligne de 0 : un 1
   au centre
8
9 def indice(i):
10     return i%nb_x
11
12 def evol_regle30(xt_t1, xt_t2) :
13     for i in range(nb_x) :
14         if xt_t1[i]==1 :
15             if xt_t1[indice(i-1)]==1 :
16                 xt_t2[i]=0
17             else : \# si la valeur à droite est égale à 0
18                 xt_t2[i]=1
19         else :# si la cellule i est à 0
20             if xt_t1[indice(i-1)]+xt_t1[indice(i+1)]==1 :
21                 xt_t2[i]=1
22             else : \# si la somme des valeurs à droite et à gauche est diffé
   rente de 1
23                 xt_t2[i]=0
24
25
26 for t in range(0,nb_t-1) : \# calcule pour tous les temps
27     evol_regle30(x_t[t,], x_t[t+1,])
28
29 fig = plt.figure(figsize=(15, 10))
30 plt.imshow( x_t , cmap = 'magma' )
31 plt.title( "Evolution de la règle 30" )
32 plt.xlabel('espace')
33 plt.ylabel('temps')

```

Figure I.1: Exemple de programme impératif en Python ...

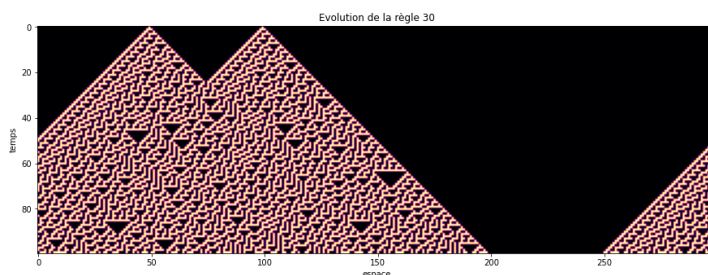


Figure I.2: ... et le résultat de son execution.

## Programmation fonctionnelle

La programmation fonctionnelle repose sur la notion fondamentale de **fonction**. Dans le cours dédié, nous verrons en détail cette notion somme toutes complexe.

Un **programme fonctionnel** est dans l'idée une **énorme fonction** (un calcul) mappant un état initial (entrée) en un état de sortie (output). Il n'y a **pas de changements d'états (immuabilité)**. Le programme décrit un calcul qui est vu comme une **évaluation de fonctions**.

Dans un langage **fonctionnel** on dit à l'ordinateur **comment faire** (how to do).

```

1 #lang racket
2
3 ;; Nombre de colonnes (x l'espace) et de lignes (t le temps)
4 (define nb-x 30)
5 (define nb-t 10)
6
7 ;; Fonction pour afficher la grille (mode texte)
8 (define (print-grid grid)
9   (for-each
10    (lambda (row)
11      (for-each (lambda (cell) (display cell) (display " ")) (vector->list row))
12      (newline))
13    (vector->list grid)))
14
15 ;; Fonction pour créer une matrice 2D initialisée à zéro
16 (define (create-matrix rows cols)
17   (make-vector rows (make-vector cols 0)))
18
19 ;; Initialiser la grille avec des valeurs spécifiques
20 (define (init-grid)
21   (let ((grid (create-matrix nb-t nb-x)))
22     (vector-set! (vector-ref grid 0) 19 1)
23     (vector-set! (vector-ref grid 0) 9 1)
24     grid))
25
26 ;; Calcul de l'indice modulo
27 (define (indice i n)
28   (modulo (+ i n) n))
29
30
31 ;; Fonction pure pour calculer la prochaine ligne
32 ;; en fonction de la règle 30
33 (define (evol-regle30 xt-t1)
34   (define (rule30 i)
35     (let ((left (vector-ref xt-t1 (indice (- i 1) nb-x)))
36           (center (vector-ref xt-t1 i))
37           (right (vector-ref xt-t1 (indice (+ i 1) nb-x))))
38       (cond
39         ((and (= center 1) (= left 1)) 0)
40         ((= center 1) 1)
41         ((= (+ left right) 1) 1)
42         (else 0))))
43   (list->vector (map rule30 (build-list nb-x values))))
44
45 ;; Fonction pour calculer toutes les lignes
46 (define (calcule-toutes-lignes grid)
47   (define (evolve grid t)
48     (let ((new-line (evol-regle30 (vector-ref grid (- t 1)))))
49       (vector-set! grid t new-line)
50       grid))
51   (foldl (lambda (t g) (evolve g t))
52         grid (build-list (- nb-t 1) (lambda (x) (+ x 1)))))
53
54 ;; Programme principal
55 (let ((x-t (init-grid)))
56   (define result-grid (calcule-toutes-lignes x-t))
57   (print-grid result-grid))

```

Figure I.3: Exemple de programme fonctionnel en Racket (une forme de Lisp) ...

```

44 (define (print-grid grid)
45   (for-each
46     (lambda (row)
47       (for-each (lambda (cell) (display cell) (display " ")) (vector->list row))
48       (newline))
49     (vector->list grid)))
50
51 ;; Programme principal
52 (let ((x-t (init-grid)))
53   (define result-grid (calcule-toutes-lignes x-t))
54   (print-grid result-grid))

```

```

Welcome to DrRacket, version 8.6 [cs].
Language: racket, with debugging, memory limit: 128 MB.
0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 1 0 0 1 0 0 0 0 0 0 1 1 0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 1 1 0 1 1 1 1 0 0 0 1 1 0 1 1 1 1 0 0 0 0 0 0 0
0 0 0 0 0 1 1 0 1 1 1 1 0 0 0 1 1 0 1 1 1 0 0 0 0 0 0 0 0 0
0 0 0 0 1 1 0 1 1 1 1 0 1 1 0 1 1 0 1 1 1 1 0 1 1 1 0 0 0 0
0 0 0 1 1 0 0 1 0 0 0 0 1 0 0 1 0 1 0 0 0 0 1 0 0 1 0 0 0 0
0 0 1 1 0 1 1 1 1 0 0 1 1 1 1 0 1 1 0 0 1 1 1 1 1 1 0 0 0
0 1 1 0 0 1 0 0 0 1 1 1 0 0 0 0 0 1 0 1 1 1 0 0 0 0 0 1 0 0
1 1 0 1 1 1 1 0 1 1 0 0 1 0 0 0 1 1 0 1 0 0 1 0 0 0 1 1 1 0
>

```

Figure I.4: ... et le résultat de son execution (affichage console seulement).

### Programmation Descriptive

Il existe de très nombreux langages de description. Les langages de description comme JSON ou XML permettent de décrire des informations, basiquement des données qu'on souhaite stocker. Mais d'autres sont spécialisés dans la description de transformations (qui sont des informations) comme XSLT (qui est un langage XML) ou encore le format USD (Universal Scene Description) de Pixar.

```

1 from pxr import Usd, UsdGeom
2 stage = Usd.Stage.Open('HelloWorld.usda')
3 hello = stage.GetPrimAtPath('/hello')
4 stage.SetDefaultPrim(hello)
5 UsdGeom.XformCommonAPI(hello).SetTranslate((4, 5, 6))
6 print stage.GetRootLayer().ExportToString()
7 stage.GetRootLayer().Save()
8 refStage = Usd.Stage.CreateNew('RefExample.usda')
9 refSphere = refStage.OverridePrim('/refSphere')
10 print refStage.GetRootLayer().ExportToString()
11 refSphere.GetReferences().AddReference('./HelloWorld.usda')
12 print refStage.GetRootLayer().ExportToString()
13 refStage.GetRootLayer().Save()

```

Figure I.5: Exemple de description de scène en langage USD (Pixar), tiré de [la documentation officielle du format USD](#)

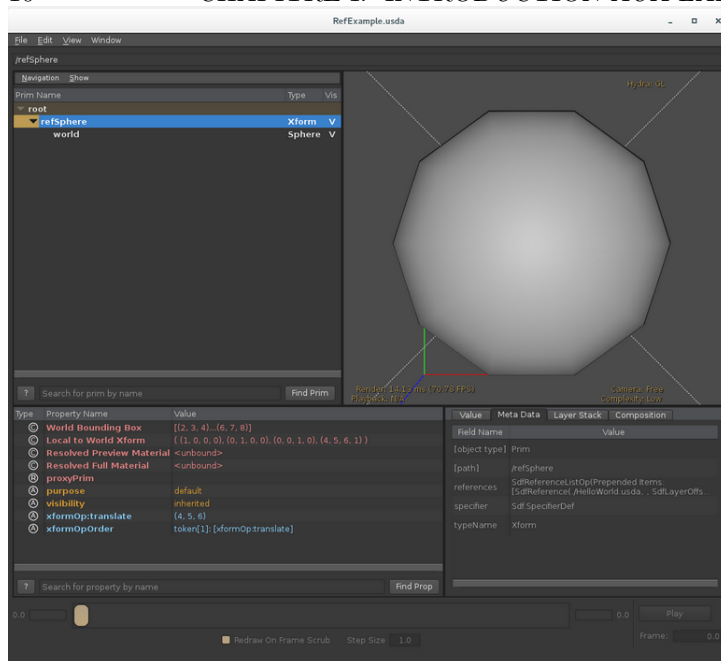


Figure I.6: ... et le résultat de son interprétation par l'API OpenUSD en Python.

Ces langages peuvent être typés (comme les instances de documents XML qui peuvent être typées par un Schema XML) ou non (comme les documents JSON qui ne sont pas contraints par un typage). Le typage de documents permet de spécifier précisément des types, d'imposer des contraintes sur les informations qui seront stockées, ne laissant pas à l'utilisateur d'ambiguïté dans ce qui peut être stocké ; Cela impose également au programmeur de penser finement la façon dont il souhaite représenter les informations, donc de les formaliser au préalable. C'est ce que nous verrons dans les cours de XML qui suit ce chapitre.

## 2.b Caractéristiques des langages informatiques

Tout comme les langages naturels, les langages informatiques vérifient systématiquement les caractéristiques suivantes:

### l'Alphabet

L'alphabet est l'ensemble des caractères qui peuvent être utilisées dans un langage (écrit ou parlé par des humains en général). Par exemple

- en Anglais : a, b, c, ..., z.
- en Français: idem qu'en Anglais, avec en plus ç, à, ê, etc.
- en Langage Machine: 0 ou 1
- en C, C++, Visual Basic: alphabet ASCII
- en Java, Python, C#: Unicode UTF-8

L'alphabet ASCII est un code qui donne une valeur numérique à chaque lettre par exemple A vaut 65, B vaut 66 etc. L'alphabet ASCII ne peut coder que 128 caractères c'est pourquoi on peut pas représenter les lettres accentuées ou autres caractères spéciaux dans cet alphabet. Pour en savoir plus sur l'alphabet ASCII, reportez-vous à [la page Wikipédia dédiée<sup>1</sup>](http://fr.wikipedia.org/wiki/American_Standard_Code_for_Information_Interchange).

L'alphabet UTF-8 aussi appelé iso 859-1 est un alphabet qui permet d'encoder plus de caractères, notamment les caractères accentués. Reportez-vous à [la page Wikipédia dédiée<sup>2</sup>](http://fr.wikipedia.org/wiki/UTF-8).

<sup>1</sup>[http://fr.wikipedia.org/wiki/American\\_Standard\\_Code\\_for\\_Information\\_Interchange](http://fr.wikipedia.org/wiki/American_Standard_Code_for_Information_Interchange)

<sup>2</sup><http://fr.wikipedia.org/wiki/UTF-8>



### le Vocabulaire

Le vocabulaire est l'ensemble des mots prédéfinis d'un langage (humain). On trouve généralement la définition de l'ensemble des mots d'un langage c'est-à-dire de son vocabulaire dans le dictionnaire. Par exemple:

- en Anglais: Oxford Dictionary
- en Français: Le Petit Robert, Le Petit Larousse, etc.
- en Langage Machine: Assembleur (dépend du processeur)
- en Java: 48 mots prédéfinis (ex: `for`, `class`, etc. )
- en C# : 78 mots prédéfinis (cf. [Mots clefs du C#](#))

### la Syntaxe

La syntaxe est l'ensemble des règles qui indiquent comment se font les phrases. Par exemple:

- en Anglais ou en Français : *Sujet Verbe Complément.* ou *Verbe Sujet Complément ?*
- en Langage Machine: *mot + données = instruction*
- en C, C++, Java, C# ... : *mot + opérateur (=, +, -) ponctuation (;)*

### la Sémantique

La sémantique est l'ensemble des règles qui indiquent quelles sont les phrases qui ont du sens. Par exemple si l'on considère les quatre phrases suivantes, elles sont toutes correctes sur le plan du vocabulaire et de la syntaxe mais deux d'entre elles n'ont aucun sens.

- Pascal lance un caillou. ==> OK
- Un caillou lance Pascal. ==> NON OK
- Il faut changer l'ampoule. ==> OK
- Je faux changer l'ampoule. ==> NON OK

La sémantique doit également être respectée dans les langages informatiques. Par exemple en C, si l'on écrit:

- `x = 5`; cela signifie que l'on met la valeur 5 dans la variable `x`.
- `5 = x`; ne veut rien dire, puisque l'on ne peut pas mettre de valeur dans le chiffre 5.

### l'organisation des phrases

Dans les langage naturels, les phrases sont organisées en paragraphes, les paragraphes sont organisés en sous-sections, sections, chapitres, tomes, volumes, etc. En langage objet comme C# ou Java par exemple les instructions sont organisées en classes, méthodes, et blocs d'instructions.



# Formalisation - Introduction

## 1 Formaliser / Modéliser

### 1.a Stocker

Le stockage de données est évidemment un problème crucial sur Internet ou localement dans nos ordinateurs.

Supposons l'énoncé suivant : “On veut stocker dans un fichier la liste des bagages d'un passager d'un vol Air France. Le passager a 2 bagages qu'il enregistre le 6 janvier 2012. Le premier bagage est un sac à dos rouge référencé AF677793 et le second une valise noire référencée AF67840. Ces bagages pèsent respectivement 9.8 et 22.5 kg”.

Comment stocker ces données ? Comment les modéliser ? Il s'agit de 2 problèmes différents.

Il y a autant de façons de stocker les données que de programmeurs. Chaque programmeur peut décider de son propre format de données, le programme qui lira ou écrira ces données étant intrinsèquement lié à la façon dont elles sont arrangées dans le document de sauvegarde.

Par exemple on pourrait avoir le document montré figure II.1. On voit là que ce document n'a pas une organisation très claire. Un autre exemple est le format CSV montré figure II.2. Cette fois, la première ligne contient le nom de champs et les autres les valeurs... pourquoi pas. Le format CSV est après tout très utilisé lorsque les données sont assez massives et destinées à faire l'objet de traitements statistiques par exemple.

```

1 Air France
2 2012-01-06
3 bagage AF677793
4 sac à dos ; rouge ; 9.8
5 bagage AF67840
6 valise ; noir ; 22.5

```

Figure II.1: Exemple de document texte stockant des données.

Compagnie	Date	RefBagage	Type	Couleur	Poids
Air France	2012-01-06	AF677793	sac à dos	rouge	9.8
Air France	2012-01-06	AF67840	valise	noir	22.5

Figure II.2: Autre exemple de document texte stockant des données au format CSV (coma separated text).

## 1.b Modéliser

Mais ces représentations ne reposent pas sur des modèles orientés objets des données. On voit bien dans ces exemple que l'ensemble représente un grand objet, l'ensemble des bagages d'un passager d'Air France dont le vol est le 6 janvier 2012, et la liste de ses bagages (chaque bagage étant un objet)... En programmation Orienté Objet, par exemple en Java ou C#, vous n'auriez aucun mal à imaginer leur **modélisation** sous forme de classes, par exemple :

```

1 package bagages;
2
3 import java.time.LocalDate;
4 import java.util.List;
5
6 public class BagagesPassager {
7
8     enum Format {sac_a_dos, valise, malle, bag, autre}
9     enum Couleur {rouge, vert, bleu, jaune, blanc, noir, orange, violet, gris,
10         marron, autre}
11
12     public class Type{
13         public Format format;
14         public Couleur couleur;
15     }
16
17     public class Bagage {
18         public String ref;
19         public Type type;
20         public float poids;
21     }
22
23     public String compagnie;
24     public LocalDate date;
25     public List<Bagage> bagage;
26 }
```

Figure II.3: Représentation objet de la liste des bagages d'un passager en Java (objet ne comportant que des données ; pas de méthodes).

Cette fois, l'espace des données est intelligemment découpé et organisé : on sait qui contient quoi et quels sont les types des valeurs qui seront stockées dans les attributs. Et plus que ça encore : on a cette fois modélisé les données *indépendamment* des valeurs associées. Le programmeur est parti du principe que les bagages, les types, les couleurs, particuliers pouvaient être représentés par des modèles  $\Leftarrow \Rightarrow$  formalisés.

On doit donc voir plus loin que la valeur et la généraliser en types ... et ensuite organiser ces types. C'est cela modéliser. C'est ce que nous ferons dans cette UE.

## 1.c UML

Mais avant d'implémenter un modèle dans un langage dédié comme XML Schema, il est utile de représenter ces types (le modèle) et valeurs (instances) sous une forme graphique. C'est à cela que sert UML (Unified Modeling Language). UML est une norme graphique permettant de représenter de nombreuses choses en informatique : diagrammes de classes, arbres d'instances, diagrammes fonctionnels ...

Nous n'allons pas faire ici un cours d'UML mais au fil de ces cours nous montrerons comment représenter Schemas XML, diagrammes de classes, instances XML ... sous la forme de diagrammes UML.

Au delà de la représentation à l'aide d'un papier et d'un crayon, ou d'un tableau et de feutres, outils principaux d'un bon programmeur, notez qu'il existe plusieurs outils utiles pour générer de tels diagrammes. [PlantUML](#) en fait partie. Pour s'en servir, il suffit de soumettre en ligne ([Web Server PlantUML](#)) votre code PlantUML. Vous trouverez de la documentation [ici](#) et [ici](#).

... et notez aussi qu'il existe des plugins intéressants comme par exemple celui qui génère des diagrammes de classes à partir de schemas XML. C'est d'ailleurs avec ce plugin que j'ai généré le code plantuml ci-dessous correspondant au Schema XML donné figure II.8, à l'aide de la commande suivante : `xsd2uml bagages/bagages.xsd -output plantuml -package bagages` (NB: en réalité, le plugin n'a pas généré les packages et quelques relations que j'ai ajoutés).

```

1 @startuml
2
3 package XMLSchema {
4     class string
5     class date
6     class decimal
7 }
8
9 package bagages {
10
11     class Bagage {
12         +ref : string
13         +type : Type
14         +poids : decimal
15     }
16
17     class BagagesPassager {
18         +compagnie : string
19         +date : date
20         +bagage : Bagage[]
21     }
22
23     enum Couleur {
24         rouge = rouge
25         vert = vert
26         bleu = bleu
27         jaune = jaune
28         blanc = blanc
29         noir = noir
30         orange = orange
31         violet = violet
32         gris = gris
33         marron = marron
34         autre = autre
35     }
36
37     enum Format {
38         sac à dos = sac à dos
39         valise = valise
40         malle = malle
41         bag = bag
42         autre = autre
43     }
44
45     class Type {
46         +format : Format
47         +couleur : Couleur
48     }
49
50     class bagagesPassagers {}
51
52     bagagesPassagers *- BagagesPassager
53     Type *-- Format
54     Type *-- Couleur
55     BagagesPassager "1" *-- "many" Bagage : list
56     Bagage *-- Type
57

```

```

58 }
59 }
60 @enduml

```

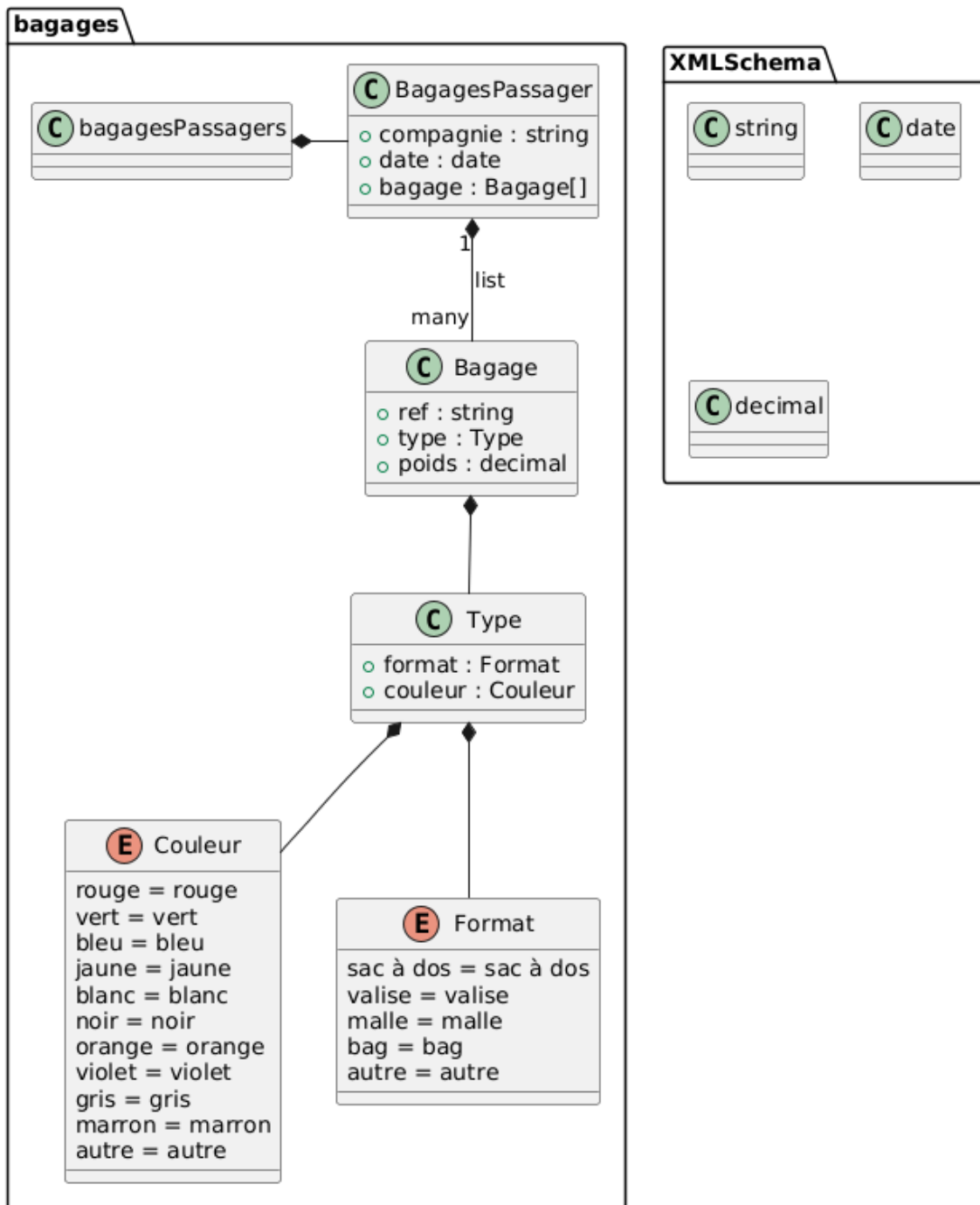


Figure II.4: Diagramme de classes généré par le code PlantUML préalablement généré et correspondant au Schema XML donné figure II.8.

<sup>2</sup>Hyper Text Transfert Protocol

```

    <meta name="viewport" content="width=device-width, initial-scale=1.0"/>
  </head>
  <body>
    <div>TODO write content</div>
  </body>
</html>

```

## Les balises

Le langage XHTML est organisé grâce à des balises.

`<nom>` est une balise ouvrante (on ouvre un bloc d'organisation). `nom` est alors le nom de la balise.

`</nom>` est une balise fermante. Une balise ouvrante et une balise fermante délimitent un bloc.

`<nom/>` est une balise ouvrante/fermante (dites encore auto-fermante).

Le XHTML est un ensemble de texte et de marqueurs délimitant des blocs:

```

<exemple> ici du texte </exemple>
           texte délimité par des balises ou-
           vantes et fermantes exemple

```

```

<exemple>
  <plus> compliqué </plus>
</exemple>

```

On a, dans ce second exemple, une structure arborescente que l'on peut représenter comme suit:

```

+ exemple
|
- + plus
  |
  - compliqué

```

## Structure HTML

HTML définit un ensemble de balises (mots) ainsi qu'une organisation (sémantique). HTML utilise la balise `html` pour englober toutes les autres balises.

```

1 <html>
2   <head>
3     <title>Exemple de Titre de Page HTML</title>
4   </head>
5   <body>
6     <h1>Premier Titre</h1>
7     <p>Texte dans un paragraphe...</p>
8   </body>
9 </html>

```

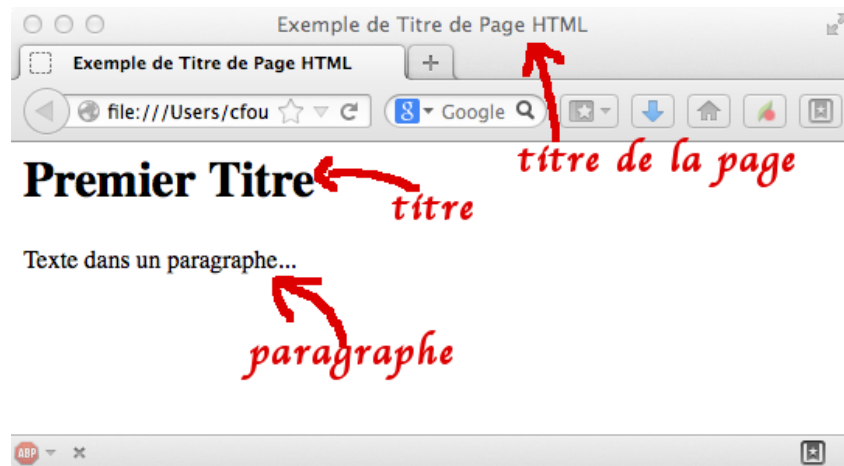
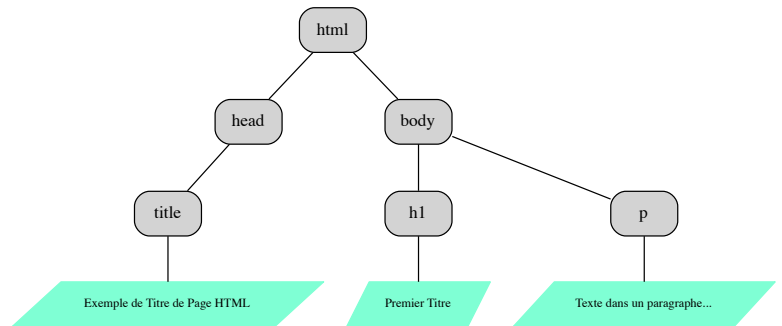
Cet exemple de page HTML peut aussi être représentée comme suit:



```

+ html
|
| - + head
| |
| | - + title
| | |
| | | - Exemple de Titre de Page HTML
| | |
| | | - + etc.
| |
| - + body
| |
| | - + h1
| | |
| | | - Premier Titre
| |
| | - + p
| |
| | - Texte dans un paragraphe...

```



### Les principales balises html

**html** Balise *racine* d'un document html. Les autres balises sont incluses dans cette balise

**head** Entête d'un document html. Contient des informations sur le document qui ne seront pas forcément affichées dans le navigateur.

**body** Indique le corps du document contenant ce qui doit être affiché sur la page web. Les balises suivantes sont incluses dans la balise **body**

#### 1. Les balises de structure

**h1** Titre de niveau 1 (gros)

**h2** Titre de niveau 2 (plus petit)

**p** Paragraphe

**br** Retour à la ligne

**hr** Ligne horizontale

#### 2. Mise en forme de texte

**i** Italique (exemple: `<i>mot</i>`)

**b** Gras (exemple: `<b>texte en gras</b>`)

#### 3. Les listes

**ol** Listes numérotées (*ordered list*)

```

1 <ol>
2   <li> Point A </li>
3   <li> Point B </li>
4   <li> C </li>
5 </ol>

```

1. Point A
2. Point B
3. C

**ul** Liste à points (*unordered list*)

```

1 <ul>
2   <li> Point A </li>
3   <li> Point B </li>
4   <li> C </li>
5 </ul>

```

- Point A
- Point B
- C

## 4. Les tableaux

**table** Début d'un tableau**tr** Ligne (*table row*)**td** Cellule d'une ligne (*table data*)

```

1 <table>
2   <tr>
3     <td>A</td>
4     <td>B</td>
5   </tr>
6   <tr>
7     <td>C</td>
8     <td>D</td>
9   </tr>
10 </table>

```

A	B
C	D

## 5. Les hyperliens

**a** élément d'hyperlien avec un attribut **href** (*anchor*)

```

1 <a href="http://google.com"> cliquer ici </a>

```

Il y a un espace entre le nom de l'élément **a** et le nom de l'attribut **href**. Si l'on clique sur le texte cliquer ici, on arrive sur la cible du lien référencé par l'attribut **href** (ici la page d'accueil de Google).

## 6. Les images

**img** élément d'insertion du image avec un attribut **src** (*image*).

```

1 

```

On peut remarquer que la balise **img** est une balise ouvrante/fermante. L'attribut **src** a pour valeur le nom du fichier image à insérer. Ce nom peut contenir un chemin relatif sur le disque ou bien une URL.

**2.c La limite d'HTML**

HTML, surtout doublé de feuilles de styles CSS, est un formidable langage pour modéliser des présentations de documents. Ses spécifications (structure, nom des balises ...) sont précises ... mais ne peuvent pas être ni changées, ni étendues, ni utilisées pour modéliser des données ! Et c'est bien ça la limite de ce langage !

HTML ne peut pas servir à représenter des données, à les typer, à les stocker. Pour faire cela, il faut utiliser un langage dédié comme JSON ou XML.

### 3 JSON - un langage de modélisation et stockage des données

JSON signifie JavaScript Object Notation. C'est un standard utilisé pour représenter des données structurées. Il est particulièrement compatible avec Javascript dont il reprend la syntaxe des objets. Il peut cependant être utilisé séparément de Javascript. Ce n'est pas particulièrement un langage "à balises", mais dans une certaine mesure, l'enchassement des différents objets et leur nommage explicite s'y apparente.

Nous n'allons pas faire un cours de JSON ici, d'autant que ce ne serait pas bien compliqué ; vous trouverez assez facilement des tutos en ligne. L'idée est de vous montrer les différences de documents XML et JSON, et vous montrer aussi les différences de typage entre XMLSchema et JSO?-Schema. JSON est plus compact que XML en écriture mais finalement assez peu. En réalité, vous découvrirez assez tôt l'intérêt majeur que représentent les technologies XML, bien plus vastes que ce que permet l'usage de JSON, ces technologies XML existant depuis plus longtemps et plus stables que les technologies JSON.

#### 3.a Structure d'un document JSON

Un objet JSON sont délimités par des accolades ouvrantes et fermantes, les valeurs numériques sont notées sans guillemets, les valeurs Booléennes également (true ou false) et les textes entre guillemets, les tableaux sont délimités par des crochets. Les champs contenant les valeurs sont nommés et les valeurs leurs sont associées par le symbole `:`. Dans un même objet, les champs sont séparés par des virgules. Par exemple :

```
1 {  
2   "compagnie" : "Air France",  
3   "date": "2012-01-06",  
4   "bagages" :  
5   [  
6     {  
7       "ref" : "AF677793",  
8       "type" :  
9       {  
10        "format" : "sac à dos",  
11        "couleur" : "rouge"  
12      },  
13      "poids" : 9.8  
14    },  
15    {  
16      "ref" : "AF67840",  
17      "type" :  
18      {  
19        "format" : "valise",  
20        "couleur" : "noir"  
21      },  
22      "poids" : 22.5  
23    }  
24  ]  
25 }
```

Figure II.6: Exemple de document JSON

Voilà, vous savez écrire du JSON ! Mais vous verrez qu'écrire du XML n'est pas plus compliqué.

#### 3.b JSON Schema

Les types stockés ici (dans le document JSON) peuvent être modélisés. Certains aspects de la modélisation ne sont cependant pas simples à implémenter en **JSON Schema**, en particulier **la notion d'héritage**.

Voilà un exemple très simplifié par rapport au schéma XML correspondant (voir plus loin) de ce qu'on pourrait écrire en JSON Schema. On voit que ce n'est pas simple à écrire et dur à lire. On pourrait l'améliorer en faisant plusieurs schémas et surtout, il manque dans ce schéma la contrainte (expression

régulière) sur la référence bagage, ainsi que les types énumérés pour la couleur du bagage et son type. On peut le faire bien entendu. Je vous laisserai faire votre choix entre les schémas XML et JSON ...

```

1  {
2  "$id": "http://www.timc.fr/nicolas.glade/bagages",
3  "$schema": "https://json-schema.org/draft/2020-12/schema",
4  "type": "object",
5  "properties": {
6    "compagnie": {
7      "description": "le nom de la compagnie",
8      "type": "string"
9    },
10   "date": {
11     "description": "la date du vol",
12     "type": "date"
13   },
14   "bagages": {
15     "description": "une liste de bagages",
16     "type": "array",
17     "items": {
18       "type": "object",
19       "properties": {
20         "ref": {
21           "description": "la référence d'un bagage",
22           "type": "string"
23         },
24         "type": {
25           "type": "object",
26           "properties": {
27             "format": {
28               "description": "le type de bagage",
29               "type": "string"
30             },
31             "couleur": {
32               "description": "la couleur d'un bagage",
33               "type": "string"
34             }
35           }
36         },
37         "poids": {
38           "description": "le poids d'un bagage",
39           "type": "number"
40         }
41       }
42     },
43     "minItems": 1,
44     "uniqueItems": true
45   }
46 },
47 "required": [
48   "compagnie",
49   "date",
50   "bagages"
51 ]
52 }
```

## 4 XML - un langage de modélisation et stockage des données

Un ensemble de technologies associées à un langage permettent de lever les ambiguïtés et de bien distinguer le fond et la forme d'un document, contrairement à HTML ou même XHTML, et de modéliser des données : les technologies XML.

XML signifie EXtensible Markup Language, c'est à dire *langage de balises qui peut être étendu*. Nous verrons par la suite que l'on peut définir nous-même le nom des balises (contrairement au HTML où, par exemple, les balises délimitant un paragraphe sont déjà définies et ne peuvent être que `<p>...</p>`). C'est en cela qu'XML est extensible. Cependant, comme son vocabulaire n'est pas défini *a priori*, il ne s'agit pas vraiment d'un langage informatique, mais d'un *méta-langage* (c'est-à-dire un langage qui permet de définir des langages, ou *dialectes*).

Il s'agit en fait simplement d'un ensemble de règles à respecter lorsque l'on veut décrire des informations structurées.

#### 4.a Exemple

On considère l'exemple de la figure II.7 :

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!-- Bagagerie -->
3 <bagagesPassagers
4   xmlns="http://www.timc.fr/nicolas.glade/bagages"
5   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
6   xsi:schemaLocation="http://www.timc.fr/nicolas.glade/bagages bagages.xsd">
7   <compagnie>Air France</compagnie>
8   <date>2012-01-06</date>
9   <bagage>
10    <ref>AF677793</ref>
11    <type format="sac à dos" couleur="rouge"/>
12    <poids>9.8</poids>
13  </bagage>
14  <bagage>
15    <ref>AF67840</ref>
16    <type format="valise" couleur="noir"/>
17    <poids>22.5</poids>
18  </bagage>
19 </bagagesPassagers>
```

Figure II.7: Exemple de document XML (*instance xml*).

On peut remarquer plusieurs choses sur ce document, par exemple:

- le nom des balises est en français, ce qui est peu commun pour un langage informatique. Il s'agit en effet d'un vocabulaire *ad hoc*, qui a été défini spécialement par la personne qui a écrit ce document pour une application donnée.
- les balises sont *indentées*. Lorsqu'une balise commence à l'intérieur d'un autre élément, elle est décalée vers la droite pour une meilleure lisibilité. Indenter correctement son document XML n'est pas obligatoire, mais cela fait partie des bonnes pratiques de programmation.
- le nom des éléments est explicite: même si l'on ne connaît pas ce langage, on peut comprendre aisément le type d'information contenu dans les balises. C'est un document lisible par tous.
- les balises ne décrivent que la sémantique (ce que sont les choses, par exemple *AF67840* est une référence de bagage d'un passager), il n'y a aucune information de présentation des données (en gras, italique, retour à la ligne, etc.).
- il se réfère à un autre document (bagages.xsd) et fait appel à plusieurs vocabulaires, ou namespaces, (ici le vocabulaire des bagages défini dans le fichier **bagages.xsd** et nommé `http://www.timc.fr/nicolas.glade/bagages`, ainsi que le vocabulaire pour l'instance de schema, nommé `http://www.w3.org/2001/XMLSchema-instance`). Nous verrons plus loin dans le cours comment sont nommés ces vocabulaires et comment on s'y réfère.

On a ici un exemple de données aéroportuaires décrites dans un langage XML spécifique. En informatique, **un exemple est aussi appelé une instance**. Cette instance est fondée sur un nouveau langage qui permet d'ordonner et de typer les informations jugées nécessaires pour une application donnée.

A cette instance de document est associée un **Schema XML** (défini dans le fichier `bagages.xsd`, avec `xsd` pour XML Schema Descriptor), autrement dit un type qui, à l'instar des classes en langage objet et de leurs attributs, modélise sous la forme de types (types complexes ou simples) les objets de données qui sont instanciés dans le document XML. Voici ce schéma (ci-dessous). Etudiez le en regardant comment il est construit, et mettez le en regard de l'instance XML ci-dessus.

```

1 <?xml version="1.0"?>
2 <schema version="1.0"
3     xmlns="http://www.w3.org/2001/XMLSchema"
4     xmlns:bg="http://www.timc.fr/nicolas.glade/bagages"
5     targetNamespace="http://www.timc.fr/nicolas.glade/bagages"
6     elementFormDefault="qualified">
7
8     <element name="bagagesPassagers" type="bg:BagagesPassager"/>
9
10    <complexType name="BagagesPassager">
11        <sequence>
12            <element name="compagnie" type="string"/>
13            <element name="date" type="date"/>
14            <element name="bagage" type="bg:Bagage" minOccurs="0" maxOccurs="
15                unbounded"/>
16        </sequence>
17    </complexType>
18
19    <complexType name="Bagage">
20        <sequence>
21            <element name="ref" type="bg:Ref"/>
22            <element name="type" type="bg:Type"/>
23            <element name="poids" type="decimal"/>
24        </sequence>
25    </complexType>
26
27    <complexType name="Type">
28        <attribute name="format" type="bg:Format" use="required"/>
29        <attribute name="couleur" type="bg:Couleur"/>
30    </complexType>
31
32    <simpleType name="Format">
33        <restriction base="string">
34            <enumeration value="sac à dos"/>
35            <enumeration value="valise"/>
36            <enumeration value="malle"/>
37            <enumeration value="bag"/>
38            <enumeration value="autre"/>
39        </restriction>
40    </simpleType>
41
42    <simpleType name="Couleur">
43        <restriction base="string">
44            <enumeration value="rouge"/>
45            <enumeration value="vert"/>
46            <enumeration value="bleu"/>
47            <enumeration value="jaune"/>
48            <enumeration value="blanc"/>
49            <enumeration value="noir"/>
50            <enumeration value="orange"/>
51            <enumeration value="violet"/>
52            <enumeration value="gris"/>
53            <enumeration value="marron"/>
54            <enumeration value="autre"/>
55        </restriction>
56    </simpleType>
57
58    <simpleType name="Ref">
59        <restriction base="string">
60            <pattern value="[A-Z]{2}[0-9]*"/>
61        </restriction>
62    </simpleType>
63 </schema>

```

Figure II.8: Exemple de Schema XML. Ce schema modélise les données qui sont instanciées dans le document `bagages.xml`.





# Formalisation - Le "langage" XML

Dans ce chapitre, on considèrera l'exemple de la figure III.1 :

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <!-- Bagagerie -->
3 <bagagesPassagers
4   xmlns="http://www.timc.fr/nicolas.glade/bagages"
5   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
6   xsi:schemaLocation="http://www.timc.fr/nicolas.glade/bagages bagages.xsd">
7   <compagnie>Air France</compagnie>
8   <date>2012-01-06</date>
9   <bagage>
10    <ref>AF677793</ref>
11    <type format="sac à dos" couleur="rouge"/>
12    <poids>9.8</poids>
13  </bagage>
14  <bagage>
15    <ref>AF67840</ref>
16    <type format="valise" couleur="noir"/>
17    <poids>22.5</poids>
18  </bagage>
19 </bagagesPassagers>

```

Figure III.1: Exemple de document XML (*instance xml*).

## 1 Éléments et attributs

XML est un langage à balises. On distingue les balises, les éléments et les attributs.

### 1.a Balise

Une balise est un marqueur. Elle commence par le signe inférieur < et se termine par le signe supérieur >. Exemples: Une balise peut être ouvrante <nom>, fermante </nom> ou ouvrante/fermante (auto-fermante) <nom/>.

En XML, **une balise ouverte doit TOUJOURS être refermée.**

### 1.b Élément

Un élément est un ensemble:

- balise ouvrante + contenu + balise fermante  
ou bien

- balise ouvrante/fermante (ou auto-fermante)

Exemples:

`<ref>AF677793</ref>` est un élément. Son contenu est AF677793 et son nom ref.

Le contenu d'un élément peut aussi être un sous élément. Par exemple

```

1      <bagage>
2          <ref>AF677793</ref>
3          <type format="sac à dos" couleur="rouge"/>
4      </bagage>

```

est un élément dont le contenu est 3 sous-éléments (`<ref>...</ref>`, `<type.../>` et `<poids>...</poids>`).

L'élément `<type format="valise" couleur="noir"/>` est une balise ouvrante/fermante (ou auto-fermante) sans contenu. Un élément auto-fermant ne contient pas de texte, mais il peut cependant contenir des attributs ; c'est le cas ici.

## 1.c Attributs

Un attribut est un couple `nom = "valeur"`. Il est associé à une balise ouvrante ou une balise ouvrante/fermante. Il a un nom et une valeur. Par exemple dans l'exemple de la figure III.1, `couleur="rouge"` est un attribut de la balise ouvrante/fermante `<type/>`

Vous remarquerez également que l'élément `bagagesPassagers` possède plusieurs attributs comme `xmlns` par exemple. Ces attributs, nous le verrons dans le chapitre suivant, servent à indiquer à quel vocabulaire appartiennent les différents mots (noms d'éléments) utilisés ici. Il s'agit ni plus ni moins de ce que l'on appelle **namespaces** (espaces de nom).

## 2 Les 8 points clés de XML

**NB** : ce qui est indiqué ici est valable pour TOUT document XML, à savoir les documents XML (instances), mais aussi les Schemas XML, les feuilles de transformation XSLT, ...

### 1. Prologue

Le prologue est la première ligne d'un document XML. Elle doit commencer au premier caractère du document (c'est à dire qu'il ne doit pas y avoir de caractères avant: ni caractères visibles, ni même de caractères d'échappement, espaces ...). Il indique au processeur du document (c'est-à-dire le logiciel qui va lire le document) qu'il s'agit d'un document XML. Il indique également la version de XML ainsi que l'alphabet (ASCII, UTF-8 ...) utilisé dans le document.

```

<?xml version="1.0" encoding="UTF-8"?>
  ^      ^           ^
espace  espace      nom de l'aphabet
instruction xml

```

Il existe plusieurs alphabets numériques (encodages) que l'on peut utiliser dans un document XML. Si l'on utilise l'alphabet *ASCII* par exemple, on ne pourra pas utiliser d'accents ni de caractères spéciaux dans le document (nom des balises, mais également contenu des éléments). Ici, on utilisera majoritairement l'alphabet *UTF-8*, mais on pourrait également utiliser *Unicode*, *latin-1*, *ISO-8859-1*, etc.

**NB** : il n'est en général pas recommandé d'utiliser d'autres encodages que l'*ASCII* de base ! L'usage d'accents et de caractères spéciaux implique que votre système est bien configuré et surtout peut poser des problèmes de compatibilité lorsque les fichiers sont utilisés par d'autres personnes.

### 2. Arborescence et balises

Un document XML est une hiérarchie d'éléments représentable sous forme d'arbre. Il n'y a qu'un seul élément racine dans un document XML. Tout élément doit être soit l'élément racine, soit inclus dans un et un seul autre élément aussi appelé élément parent. Il ne doit donc pas y avoir de recouvrement.

L'exemple de la figure III.1 peut être représenté sous forme d'arbre de la façon suivante:

Un contre-exemple est donné dans la figure suivante:

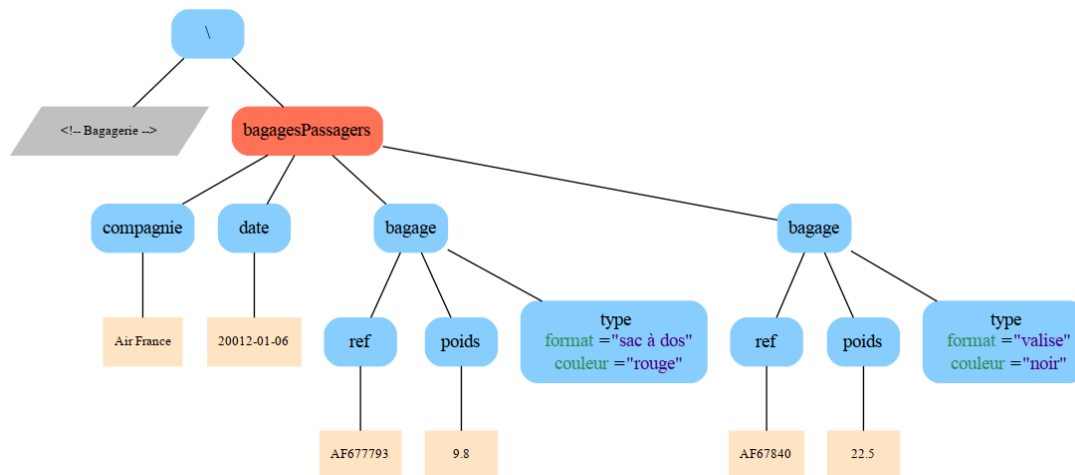
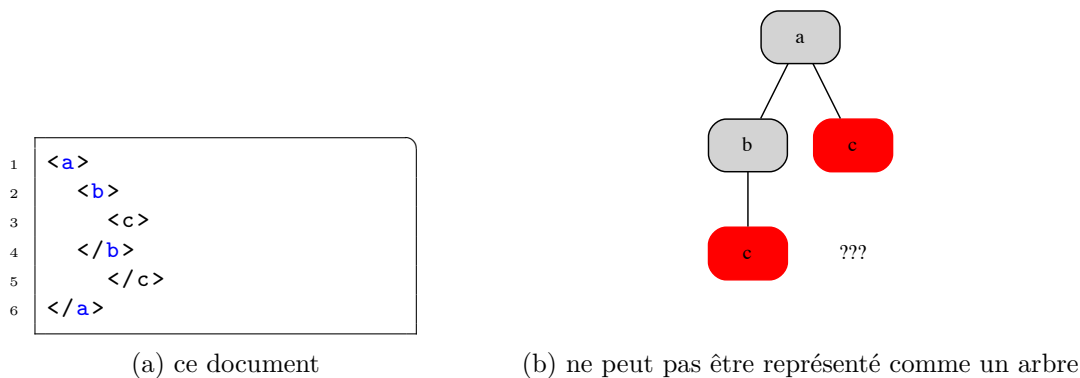


Figure III.2: représentation sous forme d'arbre du document XML de la figure II.7. .

Figure III.3: Exemple de document ne vérifiant pas l'arborescence des balises: l'élément **c** n'est pas totalement inclus dans l'élément **b**.

### 3. Complétion

Tous les éléments d'un document XML doivent être complets. C'est-à-dire que:

- toute balise ouverte doit être fermée
- soit il y a un contenu (ex. `<balise>contenu</balise>`)
- soit l'élément est vide de contenu texte (ex. `<balise/>` équivalent à `<balise></balise>`)

### 4. Règles de nommage

L'alphabet est défini dans le prologue. L'encodage (**encoding**) indique les caractères qu'il est possible d'utiliser dans le document XML (donc y compris dans le nom des balises).

Par ailleurs, il existe des règles additionnelles pour définir le nom des balises et des attributs:

- tout nom d'élément commence par un caractère alphabétique non accentué (ou bien le caractère `_` (souligné / underscore))
- les caractères qui suivent sont alphanumériques, accentués ou non (selon l'alphabet) ou le caractère `_` (souligné / underscore) ou `.` (point / dot).
- il ne peut pas y avoir d'espace dans les noms de balise ou d'attribut

- le nom d'une balise ou d'un attribut ne peut pas commencer par `xml` (quelque soit la casse (minuscules ou majuscules))

Par ailleurs, XML est sensible à la *casse*<sup>1</sup>. La balise `<nomdefamille>` est donc différente de la balise `<nomDeFamille>`

Exemples:

- `<compteClient>` → **OK**
- `<ma balise>` → **KO** : contient un espace dans le nom
- `<maBalise nom="espace">` → **OK**
- `<Tom&Jerry>` → **KO** : contient le caractère `&` qu'il faut remplacer comme nous le verrons dans le point 7
- `<12j>` → **KO** : commence par un caractère numérique et non alphabétique
- `<base16>` → **OK**
- `<prénom>` → **OK** : à condition que l'alphabet accepte les accents

NB: Encore une fois, s'il est évidemment normal de pouvoir enregistrer n'importe quel symbole de n'importe quel encodage (sauf les caractères interdits qui doivent être remplacés par leurs substituts, cf section substitutions ci-dessous) dans les données contenues dans les éléments ou dans les valeurs des attributs, il n'est vraiment pas recommandé d'utiliser d'autre alphabet que l'alphabet ASCII pour les noms des éléments et des attributs.

## 5. Attributs

Les attributs sont associés aux **balises ouvrantes** ou **balises sans contenu**. Les noms des attributs suivent les mêmes règles que les noms des éléments. La syntaxe des attributs est la suivante:

- `nomAttribut="valeur"`
- les guillemets sont obligatoires
- On peut remplacer " (double quote) par ' (single quote) si nécessaire (exemple: `<type format='valise "à roulettes"'/>`)

### Sous-élément ou attribut ?

Il y a équivalence entre

- `<type format='valise "à roulettes"'/>` et
- `<type><format>valise "à roulettes"</format></type>`

Le choix entre l'utilisation d'attributs ou de sous-éléments est laissé au concepteur. Les critères de choix sont variés: la clarté de la présentation, la longueur du document, les conséquences sur la programmation, etc. Attention cependant, on ne peut pas avoir de sous-élément dans un attribut, alors que l'on peut *empiler* les hiérarchies dans un élément.

*Remarque:* en pratique une différenciation entre éléments et attributs se fait souvent sur l'utilisation associée aux contenus. Les éléments seront davantage utilisés pour stocker les données d'intérêt, tandis que les attributs seront utilisés pour caractériser ces données et éventuellement les retrouver lors d'une recherche spécifique dans le document. Dans l'exemple des bagages ci-dessous, la donnée stockée est la référence, les attributs de couleur et de format pouvant servir à retrouver des références correspondantes (associées à un passager). Ici, une recherche faite sur les sac à dos donnera une liste de 2 références.

<sup>1</sup>XML fait la différence entre les lettres minuscules et majuscules.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <!-- Références bagagerie -->
3 <bagagesPassagers
4     xmlns="http://www.timc.fr/nicolas.glade/bagages"
5     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
6     xsi:schemaLocation="http://www.timc.fr/nicolas.glade/bagages bagages.xsd">
7     <compagnie>Air France</compagnie>
8     <date>2012-01-06</date>
9     <bagage>
10         <ref>AF677793</ref>
11         <type format="sac à dos" couleur="rouge"/>
12         <poids>9.8</poids>
13     </bagage>
14     <bagage>
15         <ref>AF67840</ref>
16         <type format="valise" couleur="noir"/>
17         <poids>22.5</poids>
18     </bagage>
19     <bagage>
20         <ref>AF48717</ref>
21         <type format="sac à dos" couleur="noir"/>
22         <poids>7.3</poids>
23     </bagage>
24 </bagagesPassagers>

```

## 6. Commentaires

Il est possible d'insérer des commentaires dans un document XML grâce à une balise spéciale:

```

1 <!-- Ceci est un commentaire
2      (possible sur plusieurs lignes)
3 -->

```

Par exemple, la ligne 2 de l'exemple III.1 il n'y a pas d'élément, mais un commentaire.

On ne peut pas utiliser -- à l'intérieur d'un commentaire. On ne peut pas insérer un commentaire à l'intérieur d'une balise (mais on peut évidemment insérer un commentaire à l'intérieur d'un élément).

## 7. Substitutions

Il existe des caractères interdits dans un document XML. par exemple, le caractère < indique le début d'une balise. On ne peut donc pas écrire

```

1 <equation> x<y </equation>

```

dans un document XML.

On substitue alors les caractères interdits par un code commençant par &.

- substitutions obligatoires:

< doit être remplacé par &lt; (*Lesser Than*)

& doit être remplacé par &amp; (*AMPersand*, en français *esperluette*)

- substitutions conseillées:

> doit être remplacé par &gt; (*Greater Than*)

" doit être remplacé par &quot; (*QUOTation mark*)

' doit être remplacé par &apos; (*APOStroph*)

## 8. Rubriques CDATA

Dans une rubrique *Character Data*, les caractères ne sont pas analysés comme du XML par le processeur qui analyse le document. Ils peuvent en revanche être utilisés par d'autres applications (nous verrons des exemples dans le projet). Dans une telle section, tous les caractères sont autorisés, y compris <, & ... La rubrique CDATA est généralement utilisée pour inclure des portions de code non (nécessairement) XML (comme du Javascript, du C++, ... ou du code XML bien entendu).

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <!-- Du XML qui contient du C ... -->
3  <code
4      xmlns="http://www.timc.fr/nicolas.glade/code"
5      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
6      xsi:schemaLocation="http://www.timc.fr/nicolas.glade/code code.xsd">
7      <script>
8          <![CDATA[
9              int comparer(int a, int b) {
10                  if ((a < b) && (a < 0)) {
11                      return b-a;
12                  } else {
13                      return a-b;
14                  }
15              }
16          ]]>
17      </script>
18  </code>

```

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <!-- Du XML qui contient du XML ... -->
3  <code
4      xmlns="http://www.timc.fr/nicolas.glade/code"
5      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
6      xsi:schemaLocation="http://www.timc.fr/nicolas.glade/code code.xsd">
7      <script>
8          <![CDATA[
9              <?xml version="1.0" encoding="UTF-8"?>
10             <!-- Références bagagerie -->
11             <bagagesPassagers
12                 xmlns="http://www.timc.fr/nicolas.glade/bagages
13                 "
14                 xmlns:xsi="http://www.w3.org/2001/XMLSchema-
15                 instance"
16                 xsi:schemaLocation="http://www.timc.fr/nicolas.
17                 glade/bagages bagages.xsd">
18                 <compagnie>Air France</compagnie>
19                 <date>2012-01-06</date>
20                 <bagage>
21                     <ref>AF677793</ref>
22                     <type format="sac à dos" couleur="rouge
23                     "/>
24                     <poids>9.8</poids>
25                 </bagage>
26                 <bagage>
27                     <ref>AF67840</ref>
28                     <type format="valise" couleur="noir"/>
29                     <poids>22.5</poids>
30                 </bagage>
31             </bagagesPassagers>
32          ]]>
33      </script>
34  </code>

```

### 3 Document bien formé

Un document XML est dit **bien formé** (*well formed*) si et seulement s'il respecte les règles de nommage, il n'a qu'un seul élément racine, il respecte les règles de complétude et de non-recouvrement.

Pour résumer, un document est bien formé, s'il respecte les 8 points clé du XML.





# Formalisation - But des Schemas XML

## 1 Objectifs

L'informatique est la science du traitement et de la manipulation de données. Dans le chapitre précédent, nous avons vu comment était formé un document XML. Dans ce cours, nous allons voir comment organiser des données dans un document XML particulier, le Schema XML.

Organiser des données XML sur la base de la description textuelle d'une situation (par exemple l'énumération des employés d'une société avec leurs relations hiérarchiques, ou encore la modélisation d'eaux minérales de compositions différentes, de conteneurs différents ...) peut être ambigu. En effet, il peut y avoir plusieurs solutions possibles qui modélisent plus ou moins bien la question.

Il est alors difficile de prévoir un traitement automatique sans savoir exactement quelles données seront utilisées, ni quelles seront leurs valeurs ou opérations possibles. C'est pourquoi nous allons utiliser des langages de modélisation.

Dans ce cours, nous utiliserons les diagrammes UML et langage XMLSchema pour la modélisation des données.

Grâce au diagramme UML (*Unified Modeling Language*), nous allons pouvoir expliquer un système d'information sans ambiguïté. Le langage XMLSchema, permet également de décrire les types de données (de quelles données ils sont composés, et quels sont les types primitifs de ces données) et leur organisation (comment ces types sont composés entre eux), sous forme textuelle. Ce format permettra par la suite de valider un document XML par rapport à son schéma.

L'intérêt de cette démarche est double :

- pour le concepteur (ou l'équipe de conception), cela permet de réfléchir *a priori* (ce qu'il faut TOUJOURS FAIRE) à la façon dont on va organiser nos données (et par extension, nos programmes ; savoir formaliser un problème orienté données est un préalable à savoir formaliser un projet de programmation plus vaste). On spécifie **à l'avance** le problème ou les données du problème = on dit comment on pense le problème / les données (comment elles seront typées et agencées). Cette réflexion préliminaire est indispensable pour mener à bien un projet sur le long terme et éviter des déconvenues à cause de mauvaises planifications. Ainsi une fois que les choses sont fixées par un tel schéma, une telle formalisation, il faut s'y conformer. S'il s'avérait que la modélisation devenait bloquante pour la suite du projet car finalement pas adaptée, cela nécessiterait de se rasseoir autour d'une table et réfléchir à comment faire évoluer tout ce qui dépend de cette structuration ; dans ce cas le "bricolage *a posteriori*" n'est pas permis.
- pour le concepteur (ou l'équipe de conception) et pour l'utilisateur, ceci sert à contenir *a posteriori* à l'échelle d'un projet les fichiers de données et les classes sérialisées dans un programme exploitant ces données. Cela permet une standardisation, par exemple pour permettre à plusieurs équipes de travailler sur un même type de données. Par exemple, à la CNAM (la Caisse Nationale d'Assurance Maladie, un éditeur de logiciels pour les caisses primaires (CPAM)), un individu est

modélisé de la même manière dans les différents services (arrêts de travail, naissance d'un enfant, maladies de longue durée ...), son modèle étant forcé par un seul et même fichier : le schéma.

## 2 Spécifier

Ce qui vient d'être dit dans la section précédente et qui a déjà été dit dans un chapitre précédent, implique que **lorsqu'on programme, on ne doit pas se contenter d'aligner des instructions de telle manière que "ça marche"** mais on doit **d'abord, planifier, typer, organiser ...**

La meilleure façon de procéder, et vous y serez confrontés dans votre projet de fin d'année (et le cas échéant, votre projet échouera !), c'est **d'abord de discuter entre vous** comment chacun voit l'organisation d'un ensemble de données, d'un programme ..., et très vite **...de faire des schémas ... sur papier**. Oui, car on pense plus vite sur papier que sur ordinateur et parce que sur un papier on peut gribouiller, gommer, ... [*disclaimer* : votre prof datant d'un autre siècle, il utilise du papier, mais bien entendu, ça marche avec une tablette et un stylet]. Ca, c'est la première étape.

La seconde, c'est d'être plus précis en spécifiant exactement ce qu'on a, ce qu'on veut ... pour chaque élément d'une donnée ou d'un programme, pour chaque étape d'un programme. Etre précis, c'est raconter une petite histoire : "ce type contient ceci et cela et leur type est ainsi ou comme ça ..." ou bien pour un programme (ou une transformation XSLT par exemple) "cette fonction/transformation utilise tel argument pour faire ça, puis fait ceci, puis fait cela, puis renvoie cette valeur dont le type est tsointsoin". Et cette histoire, on l'écrit ... **DANS LES COMMENTAIRES !!!!!!!!!!!!!!!** Oui, ça sert à ça !

**Et maintenant mon conseil** : lorsque vous écrivez une classe, un type XMLSchema, une transformation XSLT, **(1)** d'abord vous réfléchissez sur papier, **(2)** ensuite vous écrivez un long commentaire racontant ce que contient ou fait votre bout de code, **(3)** enfin et seulement après vous écrivez effectivement votre bout de code !

### Exemple 1. la modélisation des bagages

1) D'abord, on fait un schéma UML (à la main ou avec un outil numérique) de notre modèle de liste de bagages pour un vol.

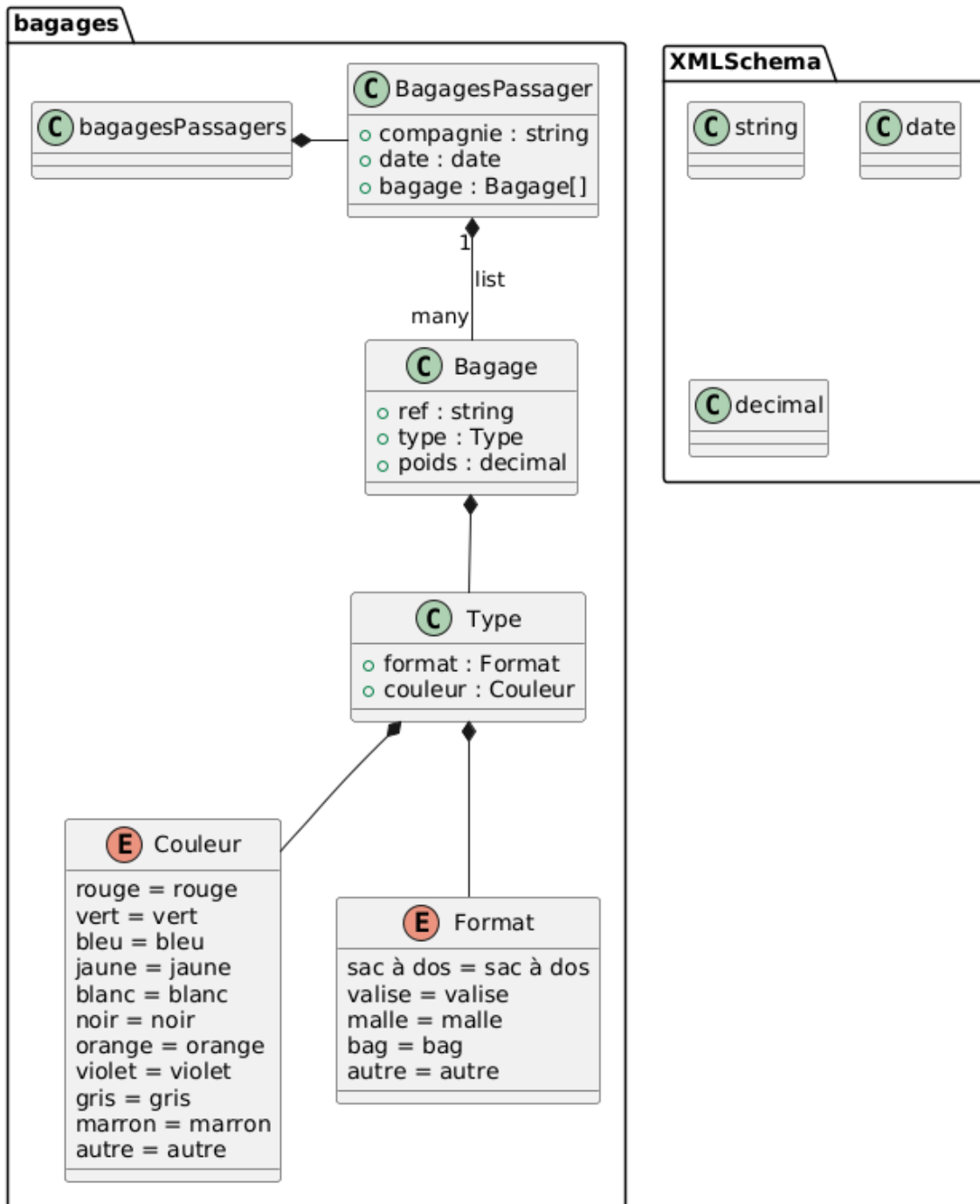


Figure IV.1: Diagramme de classes modélisant les données sur les bagages.

2) Ensuite on crée le document XMLSchema et on écrit les commentaires.

```

1 <?xml version="1.0"?>
2 <schema version="1.0"
3     xmlns="http://www.w3.org/2001/XMLSchema"
4     xmlns:bg="http://www.timc.fr/nicolas.glade/bagages"
5     targetNamespace="http://www.timc.fr/nicolas.glade/bagages"
6     elementFormDefault="qualified">
7
8     <!-- ..... RACINE ..... -->

```

```

9      <!-- On déclare un élément nommé bagagesPassager ,
10      de type BagagesPassager , qui sera l'élément racine de notre
11      instance XML -->
12
13      <!-- ..... TYPES ..... -->
14      <!-- Le type BagagesPassagers contient :
15      - une compagnie de type string
16      - une date au format date standard
17      - un ensemble illimité d'éléments bagage (de type Bagage). Il
18      peut y avoir 0 bagages -->
19
20      <!-- Le type Bagage contient :
21      - une référence de type Ref (un type simple restreint à un
22      pattern)
23      - un type de bagage de type Type (un type complexe contenant un
24      format de bagages et une couleur)
25      - un poids de type décimal -->
26
27      <!-- Le type Type contient :
28      - un format de bagages de type Format (un type simple énuméré)
29      - une couleur de type Couleur (un type simple énuméré) -->
30
31      <!-- Le type Format est une restriction de string proposant une énumération
32      de différents formats de bagages , par ex une valise ou un sac à
33      dos -->
34
35      <!-- Le type Couleur est une restriction de string proposant un
36      certain nombre de couleurs plausibles pour des bagages -->
37
38
39      <!-- Le type Ref est une restriction de string obeissant à une regex telle
40      que :
41      - la référence commence par 2 lettres en majuscule
42      - elle est suivie d'un nombre indéterminé de chiffres -->
43
44 </schema>

```

3) Le listing ci-dessous montre l'étape suivante, un exemple de Schema XML construit à partir des spécifications faites dans les commentaires. Ce schema modélise les données qui sont instanciées dans le document `bagages.xml`

```

1 <?xml version="1.0"?>
2 <schema version="1.0"
3     xmlns="http://www.w3.org/2001/XMLSchema"
4     xmlns:bg="http://www.timc.fr/nicolas.glade/bagages"
5     targetNamespace="http://www.timc.fr/nicolas.glade/bagages"
6     elementFormDefault="qualified">
7
8     <!-- ..... RACINE ..... -->
9     <!-- On déclare un élément nommé bagagesPassager ,
10     de type BagagesPassager , qui sera l'élément racine de notre
11     instance XML -->
12
13     <element name="bagagesPassagers" type="bg:BagagesPassager"/>

```

```

14      <!-- ..... TYPES ..... -->
15      <!-- Le type BagagesPassagers contient :
16          - une compagnie de type string
17          - une date au format date standard
18          - un ensemble illimité d'éléments bagage (de type Bagage). Il
              peut y avoir 0 bagages -->
19      <complexType name="BagagesPassager">
20          <sequence>
21              <element name="compagnie" type="string"/>
22              <element name="date" type="date"/>
23              <element name="bagage" type="bg:Bagage" minOccurs="0" maxOccurs="
                  unbounded"/>
24          </sequence>
25      </complexType>
26
27      <!-- Le type Bagage contient :
28          - une référence de type Ref (un type simple restreint à un
              pattern)
29          - un type de bagage de type Type (un type complexe contenant un
              format de bagages et une couleur)
30          - un poids de type décimal -->
31      <complexType name="Bagage">
32          <sequence>
33              <element name="ref" type="bg:Ref"/>
34              <element name="type" type="bg:Type"/>
35              <element name="poids" type="decimal"/>
36          </sequence>
37      </complexType>
38
39      <!-- Le type Type contient :
40          - un format de bagages de type Format (un type simple énuméré)
41          - une couleur de type Couleur (un type simple énuméré) -->
42      <complexType name="Type">
43          <attribute name="format" type="bg:Format" use="required"/>
44          <attribute name="couleur" type="bg:Couleur"/>
45      </complexType>
46
47      <!-- Le type Format est une restriction de string proposant une énumération
48          de différents formats de bagages, par ex une valise ou un sac à
              dos -->
49      <simpleType name="Format">
50          <restriction base="string">
51              <enumeration value="sac à dos"/>
52              <enumeration value="valise"/>
53              <enumeration value="malle"/>
54              <enumeration value="bag"/>
55              <enumeration value="autre"/>
56          </restriction>
57      </simpleType>
58
59      <!-- Le type Couleur est une restriction de string proposant un
60          certain nombre de couleurs plausibles pour des bagages -->
61      <simpleType name="Couleur">
62          <restriction base="string">
63              <enumeration value="rouge"/>
64              <enumeration value="vert"/>
65              <enumeration value="bleu"/>
66              <enumeration value="jaune"/>
67              <enumeration value="blanc"/>
68              <enumeration value="noir"/>
69              <enumeration value="orange"/>
70              <enumeration value="violet"/>
71              <enumeration value="gris"/>

```

```

72         <enumeration value="marron"/>
73         <enumeration value="autre"/>
74     </restriction>
75 </simpleType>
76
77 <!-- Le type Ref est une restriction de string obeissant à une regex telle
78      que :
79          - la référence commence par 2 lettres en majuscule
80          - elle est suivie d'un nombre indéterminé de chiffres -->
81 <simpleType name="Ref">
82     <restriction base="string">
83         <pattern value="[A-Z]{2}[0-9]*"/>
84     </restriction>
85 </simpleType>
86 </schema>

```

### Exemple 2. modélisation d'une transformation des données de bagages en page html

1) ce qu'on veut (une page html qu'on prototype à la main ; ici c'est le résultat qui est montré).

## Bagages du vol Air France du 2012-01-06

Référence	Description	Poids enregistré
AF48717	sac à dos de couleur noir	7.3
AF677793	sac à dos de couleur rouge	9.8
AF67840	valise de couleur noir	22.5

Figure IV.2: Page HTML que l'on souhaite voir afficher et dont on va modéliser les transformations.

2) Les commentaires pour modéliser la feuille de transformation des données de bagages en page html.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <xsl:stylesheet
3     xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0"
4     xmlns:bg="http://www.timc.fr/nicolas.glade/bagages">
5
6     <xsl:output method="html"/>
7
8     <!-- Le template principal
9          - contient tous les éléments du fichier html (entête, body ...)
10         - définit le style de la table (bordures blanches simples et cellules
11           bleu-vert) dans le header
12         - affiche un titre "Bagages du vol **compagnie** du **date**
13         - crée une table avec en première ligne (tr) l'entête contenant 3
14           cellules (th) (la "Référence", la "Description", le "Poids enregistré")
15         - applique les templates sur le champ sélectionné bg:bagage en triant
16           les bagages par numéro de référence croissant
17     -->

```

```

17     <!-- Ce template s'applique sur les éléments bg:bagages
18         Il crée une ligne de table html (tr) dans laquelle il crée 3 cellules (
19             td)
19         - la référence
20         - la description au format "**format de bagage** de couleur **la
21             couleur**"
21         - le poids du bagage
22     -->
23
24
25 </xsl:stylesheet>

```

### 3) La feuille de transformation des données de bagages en page html.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <xsl:stylesheet
3     xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0"
4     xmlns:bg="http://www.timc.fr/nicolas.glade/bagages">
5
6     <xsl:output method="html"/>
7
8     <!-- Le template principal
9         - contient tous les éléments du fichier html (entête, body ...)
10        - définit le style de la table (bordures blanches simples et cellules
11            bleu-vert) dans le header
12        - affiche un titre "Bagages du vol **compagnie** du **date**
13        - crée une table avec en première ligne (tr) l'entête contenant 3
14            cellules (th) (la "Référence", la "Description", le "Poids enregistré")
15        - applique les templates sur le champ sélectionné bg:bagage en triant
16            les bagages par numéro de référence croissant
17    -->
18    <xsl:template match="/">
19        <html>
20            <head>
21                <title>bagages.xsl</title>
22                <style>
23                    table, th, td {
24                        border: 1px solid white;
25                        border-collapse: collapse;
26                    }
27                    th, td {
28                        background-color: #96D4D4;
29                    }
30                </style>
31            </head>
32            <body>
33                <h2>Bagages du vol <xsl:value-of select="//bg:compagnie/text()"
34                    /> du <xsl:value-of select="//bg:date/text()" /></h2>
35                <table>
36                    <tr>
37                        <th>Référence</th>
38                        <th>Description</th>
39                        <th>Poids enregistré</th>
40                    </tr>
41                    <xsl:apply-templates select="//bg:bagage">
42                        <xsl:sort select="bg:ref"/>
43                    </xsl:apply-templates>
44                </table>
45            </body>
46        </html>

```

```
43 </xsl:template>
44
45 <!-- Ce template s'applique sur les éléments bg:bagages
46      Il crée une ligne de table html (tr) dans laquelle il crée 3 cellules (
47          td)
48          - la référence
49          - la description au format "***format de bagage** de couleur **la
50              couleur**"
51          - le poids du bagage
52      -->
53 <xsl:template match="bg:bagage">
54     <tr>
55         <td><xsl:value-of select="bg:ref/text()"/></td>
56         <td><xsl:value-of select="bg:type/@format"/> de couleur <xsl:value-
57             of select="bg:type/@couleur"/></td>
58         <td><xsl:value-of select="bg:poids/text()"/></td>
59     </tr>
60 </xsl:template>
61
62 </xsl:stylesheet>
```



# Formalisation - XMLSchema : Schema et Instance

Un XMLSchema (xsd) permet de définir:

- un nouveau vocabulaire (les mots clef du langage)
- un ensemble de règles (grammaire) précisant comment utiliser les mots du vocabulaire (i.e. comment ils s'articulent entre eux)

Un schéma XML (XMLSchema) sert donc de modèle pour la création de documents XML (ce que l'on nomme "instance").

## 0.a Exemple

On considère par exemple le document suivant:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <position>
3   <module>32.904237</module>
4   <angle>73.620290</angle>
5 </position>
```

Pour faciliter le traitement automatisé de ce type de document, on peut souhaiter imposer quelques *règles* sur les données de tous les documents de ce type. Ce document suggère des données sous forme de coordonnées polaires<sup>1</sup>. Un `module` devrait donc être:

- un nombre
- réel
- positif

L'`angle` quant à lui doit être compris soit entre -180 et 180 degrés, soit entre  $-\pi$  et  $\pi$  (radians). Selon ce que l'on veut exprimer par ces données (le modèle de données), on pourra éventuellement ajouter d'autres règles/restrictions.

Pour garantir l'efficacité d'un algorithme sur ces données, on va vouloir valider de manière automatique la conformité du document XML (des données) par rapport aux règles (modèle sous forme XMLSchema) comme illustré Figure V.1

<sup>1</sup>[http://fr.wikipedia.org/wiki/Coordonn%C3%A9es\\_polaires](http://fr.wikipedia.org/wiki/Coordonn%C3%A9es_polaires)

## 0.b Validité d'un document XML

Un document XML est dit **valide** par rapport à un modèle de données décrit dans un document XMLSchema (XSD) si et seulement si:

- il est bien formé
- il est conforme au modèle défini par le XSD, à savoir:
  - l'élément racine porte le bon nom
  - l'ensemble des éléments est structuré, les éléments organisés dans le bon ordre
  - les données sont du type attendu

## 0.c Modèle et Instance

Un **XMLSchema** est aussi appelé

- un **modèle** de données
- un vocabulaire/langage XML
- un **espace de noms XML**<sup>2</sup>
- un document XSD (pour *Xml Schema Description*). xsd est également l'extension des noms de fichiers XMLSchema

Un **document XML** valide *par rapport à un XMLSchema* est aussi appelé une **instance** du schéma/vocabulaire XSD.

### Remarque 1:

XMLSchema est lui-même un langage XML. Un Schéma XML est donc également un document XML valide par rapport au XMLSchema des XMLSchemas...<sup>3</sup>. On peut ainsi résumer la validation de documents XML par rapport à un schéma grâce à la figure V.2

### Remarque 2:

En pratique, la validation se fait par l'intermédiaire d'un programme tiers (en Java, C++, ...) qui (i) analyse le XMLSchema, (ii) en construit un modèle interne, (iii) analyse le document XML dont on doit vérifier la validité et (iv) vérifie la correspondance entre le schéma XML et le document XML.

<sup>2</sup>nous reviendrons sur cette notion au chapitre suivant

<sup>3</sup>nous reviendrons sur cette notion en TP...

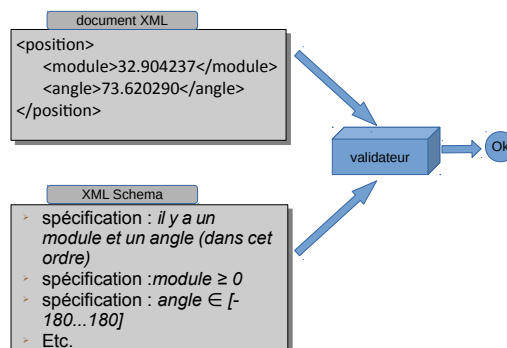


Figure V.1: Exemple de Validateur XMLSchema.

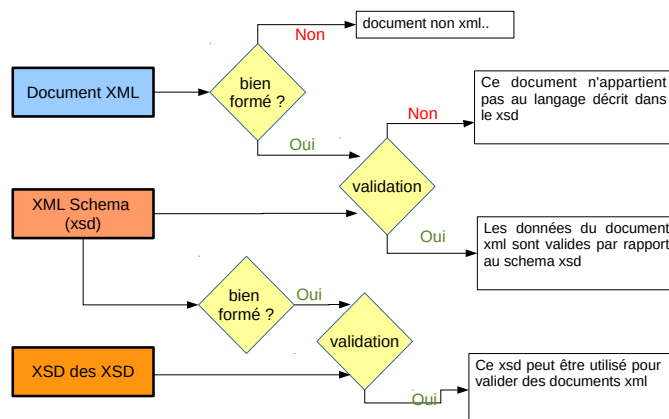


Figure V.2: Étapes de validation d'un document et d'un schéma XML.

### Remarque 3:

On peut faire un vrai parallèle entre le Schéma et l'Instance XML, et la Classe et l'Instance d'objet en programmation objet. Ce parallèle peut vous aider à vous rappeler le rôle de chaque partie du code.

Par exemple, si l'on considère le schéma (faites attention aux commentaires) suivant :

```

1 <?xml version="1.0"?>
2 <schema version="1.0"
3     xmlns="http://www.w3.org/2001/XMLSchema"
4     xmlns:dst="http://www.timc.fr/nicolas.glade/distance"
5     targetNamespace="http://www.timc.fr/nicolas.glade/distance"
6     elementFormDefault="qualified">
7
8     <!-- Déclaration de deux éléments racine différents nommés "distance"
9          et "dist". Ces éléments seront les racines de 2 instances
10         différentes évidemment ! -->
11     <!-- Ceci est équivalent en C#/Java à la déclaration d'une variable
12          dans le code. Notez que tp, le namespace, correspond au
13          package/namespace en Java/C# :
14          /* déclaration d'une variable distance et d'une variable dist de type
15             Distance la classe Distance appartenant au package/namespace dst */
16             dst.Distance distance;
17             dst.Distance dist ;
18     -->
19     <element name="distance" type="dst:Distance"/>
20     <element name="dist" type="dst:Distance"/>
21
22     <!-- Définition du type Distance -->
23     <!-- Equivalent à la définition d'une classe C#
24         namespace dst;
25         enum Unite {km, m, mm}
26         class Distance {
27             public float Value { get; set; }
28             Unite unite;
29         }
30     -->
31     <complexType name="Distance">
32         <simpleContent>
33             <extension base="nonNegativeInteger">
34                 <attribute name="unite" type="dst:Unite" use="required"/>
35             </extension>

```

```

36     </simpleContent>
37 </complexType>
38
39 <!-- Définition du type restreint Unite -->
40 <!-- Equivalent à une énumération en Java/C# -->
41 <simpleType name="Unite">
42     <restriction base="string">
43         <enumeration value="km"/>
44         <enumeration value="m"/>
45         <enumeration value="mm"/>
46     </restriction>
47 </simpleType>
48
49 </schema>

```

et une instance possible :

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <dst:dist
3     xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
4     xmlns:dst='http://www.timc.fr/nicolas.glade/distance'
5     xsi:schemaLocation='http://www.timc.fr/nicolas.glade/distance Distance.xsd'
6     unite="km">380</dst:dist>

```

... ou bien celle-ci :

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <d:distance
3     xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
4     xmlns:d='http://www.timc.fr/nicolas.glade/distance'
5     xsi:schemaLocation='http://www.timc.fr/nicolas.glade/distance Distance.xsd'
6     unite="mm">0.31</d:distance>

```

Faisons le parallèle avec un programme en C# en commençant par la définition du type Distance :

```

1 namespace dst;
2
3 public class Distance {
4     public enum Unite { mm, m, km}
5     public float _value { set; get; }
6     public Unite _unite { init; get; }
7
8     public Distance(float value, Unite unite) {
9         _unite = unite;
10        _value = value;
11    }
12 }

```

ou en style plus C# en faisant de Distance un constructeur primaire :

```

1 namespace dst;
2
3 public class Distance(float value, Distance.Unite unite) {
4     public enum Unite { mm, m, km}
5     public float _value { set; get; } = value;
6     public Unite _unite { init; get; } = unite;
7 }

```

et le programme appelant dans lequel on déclare une variable de type distance, puis on l'instancie :

```

1 using dst;
2
3 internal class Program {
4     public static void Main(string[] args) {
5         Distance dist; // déclaration

```

```
6         dist = new Distance(380, Distance.Unite.km); // instantiation +  
           affectation  
7     }  
8 }
```

Le comparatif est immédiat :

- La définition de classe C# correspond à la définition de type complexe (complexType) en XMLSchema
- Le namespace C# nommé `dst` correspond au namespace (xmlns) XMLSchema nommé `http://www.timc.fr/nicola` et préfixé `dst`
- La déclaration de la variable C# `distance` de type `dst.Distance` correspond à la déclaration d'un élément racine dans le Schema XML `<element name="distance" type="dst:Distance"/>`
- L'instanciation en C# (`new Distance(380, Distance.Unite.km)`) correspond au contenu du fichier `Distance.xml` avec ses valeurs de distance et d'unité particulières.
- L'affectation en C# (`dist = ...`) correspond à l'existence du fichier `Distance.xml` dans le système.



# Formalisation - Structure d'un XMLSchema

## 1 Conventions de nommage en vigueur dans ce cours

En XMLSchema, les noms des éléments et des types d'éléments doivent seulement respecter les conventions de nommage des noms des éléments en XML. Nous ajouterons à ces règles des *conventions de nommage* propres à ce cours, mais communément admises notamment en Java ou C#.

- les noms des éléments commencent par une minuscule et peuvent comporter des majuscules pour exprimer plusieurs mots
- les noms des *types simples* et des *types complexes* commencent par une majuscule suivie d'une minuscule et comportent une majuscule au début de chaque mot.

## 2 Exemple

Dans la suite de ce chapitre, nous allons créer le XMLSchema de l'instance XML suivante:

```

1 <?xml version="1.0" encoding="UTF-8"?>
2
3 <étudiant
4   xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
5   xmlns='http://www.uga.fr/etudiant'
6   xsi:schemaLocation='http://www.uga.fr/etudiant Etudiant.xsd'>
7
8   <nom>Kuzbidon</nom>
9   <prénom>Alex</prénom>
10  <sexe>H</sexe>
11  <dateNaissance>1991-06-15</dateNaissance>
12  <téléphone>+33 6 00 00 00 00</téléphone>
13  <adresse>
14    <numéro>42</numéro>
15    <rue>impasse de l'Univers</rue>
16    <codepostal>42420</codepostal>
17    <ville>Lunivers</ville>
18  </adresse>
19  <l3Validée>true</l3Validée>
20  <ue nom="FDD" coeff="1">
21    <note>18.5</note>
22    <validée>true</validée>
23  </ue>
24  <ue nom="IHM" coeff="1">
25    <note>15.0</note>
26    <validée>true</validée>
27  </ue>
28  <ue nom="Anglais" coeff="1">
29    <note>5.0</note>
30    <validée>false</validée>
31  </ue>
32 </étudiant>

```

Figure VI.1: Exemple d'instance XML.

### 3 Élément racine

Un XMLSchema est aussi un document XML. Il commence donc par un prologue `<?xml version='1.0' encoding='UTF-8' ?>` et a pour racine l'élément `<schema>`.

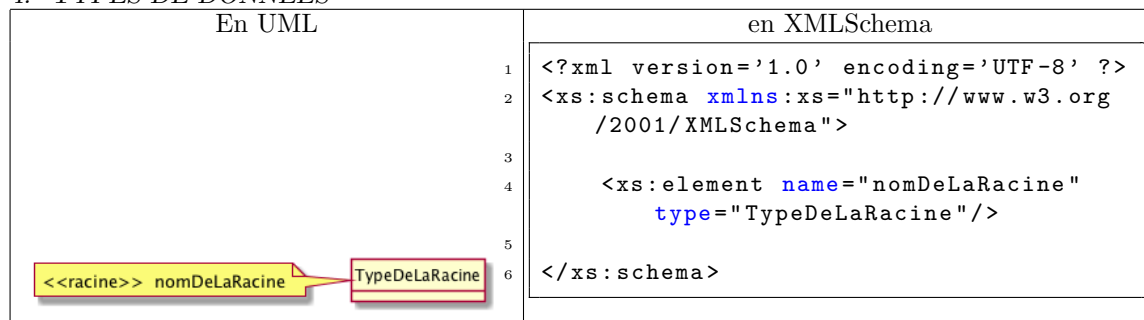
Pour désigner l'élément racine du nouveau vocabulaire<sup>1</sup> en XMLSchema, on utilise l'élément `element` avec l'attribut `name` qui désigne le nom de la racine et l'attribut `type` qui désigne son type (nous y reviendrons plus tard).

La racine, `<schema>` comporte des attributs, en particulier les attributs `xmlns`. Dans les exemples ci-dessous, la racine est préfixée `xs`, se notant ainsi `<xs:schema>` et l'attribut `xmlns` est suffixé `xs` et a pour valeur `http://www.w3.org/2001/XMLSchema`. Il s'agit de l'espace de nom définissant le vocabulaire XMLSchema ; celui-ci est lié aux mots de son vocabulaires par le préfixe `xs`. Ce point fait l'objet de la section finale sur les espaces de nom. Pour le moment, ces précisions (suffixes, ...) sont données dans les exemples afin que ces derniers soient valides ; vous pouvez cependant les ignorer pour le moment.

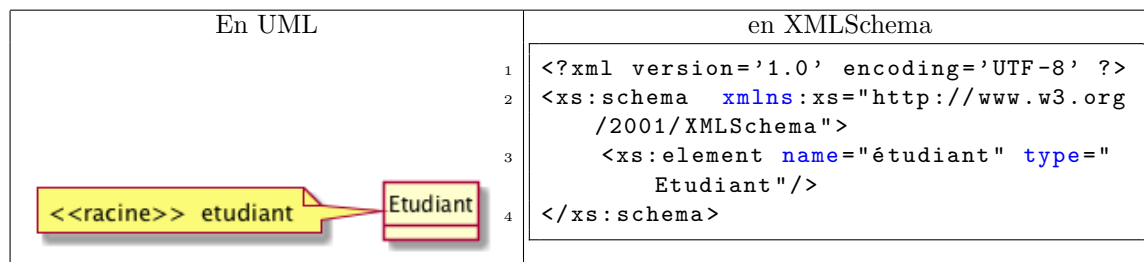
La racine s'exprime donc de la manière suivante en UML et XMLSchema:

<sup>1</sup>i.e. des instances XML qui seront valides par rapport à ce schéma





Par exemple, dans l'exemple de la figure VI.1 on obtiendrait l'UML et le XMLSchema suivant:



#### Remarque 1:

La déclaration d'un élément par `<element name="nomElement" type="TypeElement"/>` correspond à signifier qu'il est possible de créer un document XML (une instance XML) ayant pour racine l'élément `nomElement`. C'est une déclaration d'instance... au même titre qu'une déclaration d'instance en Java (ou tout autre langage) qu'on ferait en écrivant `TypeElement nomElement;`. Voir la remarque 3 faite dans la section *Schéma et Instance*.

#### Remarque 2:

Dans un fichier XMLSchema, il est tout à fait possible

- de ne déclarer aucune instance d'élément (le fichier XMLSchema ne contenant alors que des définitions de types), auquel cas il ne sera pas possible de créer d'instances XML (de document XML) ; il faut pour cela une racine. Cela peut se faire lorsqu'on écrit des types dans un schéma XML qu'on appelle (par inclusion ou importation) depuis un autre schéma.
- de déclarer plusieurs instances d'éléments (le fichier XMLSchema contenant alors des définitions de types et 1 à plusieurs déclarations d'instances d'éléments), auquel cas il sera possible de créer plusieurs instances de documents XML différentes, ayant chacune une racine différentes.

## 4 Types de Données

L'objectif d'un XMLSchema est de définir les types des données. On distingue 2 types de types de données:

- les types basiques
- les types complexes (`complexType`)

On appelle un type *basique* un type de données non décomposable, c'est-à-dire qu'un élément de type basique n'aura pas de sous-élément (exemple `<nom>Toto</nom>`) ni d'attributs. Un type basique va de plus expliciter les valeurs possibles pour la donnée (par exemple un entier positif).

On appelle type *complexe* un type de données qui contient d'autres données sous la forme d'éléments et/ou d'attributs (exemple `<étudiant><nom>Toto</nom></étudiant>` ou `<étudiant nom="Toto"/>`). Un type complexe décrit l'organisation d'une données, c'est-à-dire les sous-éléments et attributs qui la composent.

## 4.a Types basiques

Un type *basique* peut être soit

- un type prédéfini
- un type simple (`simpleType`)

### Type prédéfini

Il s'agit de types de base directement utilisables dans le langage XMLSchema (il n'y a rien à redéfinir).

Les types prédéfinis d'XMLSchema que nous utiliserons le plus souvent sont les suivant

**string** *chaîne de caractères*

**description** : Succession de caractères. Tous les caractères définis dans l'alphabet (précisé dans l'encodage) sont autorisés.

**note** : C'est le type par défaut c'est-à-dire que si aucun type n'est précisé, c'est que le type est **string**.

**int** *entiers*

**description** : Entiers (positifs et négatifs)

**note** : Attention aux limites : en XML, un entier est stocké sur 64 bits. Les valeurs possibles des entiers sont donc dans l'intervalle [-2147483648...+2147483647].

**double** *réels*

**description** : Nombres à virgule

**note** : Encodage sur 256 bits.

**date** *date*

**description** : Expression de la date au format : yyyy-mm-dd

**note** : exemple 2014-04-21

**boolean** *booléens*

**description** : Un booléen a 2 valeurs possibles : vrai (**true**) ou faux (**false**).

Il existe de nombreux autres types prédéfinis en XMLSchema que vous pouvez retrouver dans la figure VI.2 ou aux URL suivantes du site <https://www.w3schools.com/> :

- **Strings**
- **Dates et temps**
- **Valeurs numériques**
- **Autres**

Les types prédéfinis des éléments s'expriment de la manière suivante en UML et XMLSchema:

En UML	En XMLSchema
- nomElement : typePrédéfini	<element name="nomElement" type="typePrédéfini"/>

Par exemple, dans l'exemple de la figure VI.1 on obtiendrait, sur la partie XML suivante, l'UML et le XMLSchema du tableau:

```

1 <nom>Kuzbidon</nom>
2 <prénom>Alex</prénom>
3 <dateNaissance>1991-06-15</dateNaissance>
4 <l3Validée>true</l3Validée>
5 <note>18.5</note>

```

### 3 Built-in datatypes

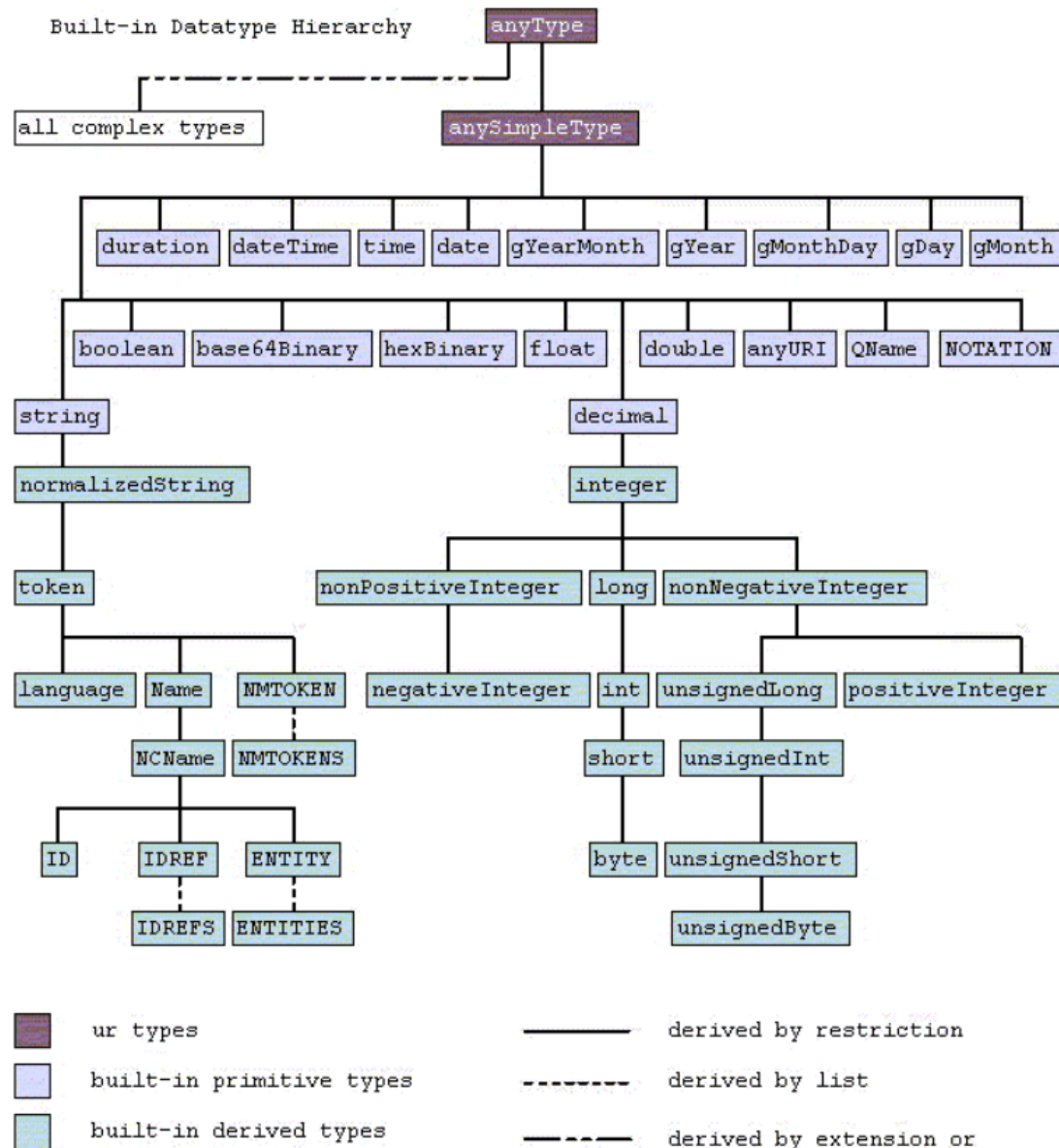


Figure VI.2: Les types prédéfinis en XMLSchema (à titre indicatif).

En UML	En XMLSchema
- nom : string	<element name="nom" type="string"/>
- prénom	<element name="prénom"/>
- dateNaissance : date	<element name="dateNaissance" type="date"/>
- l3Validée : boolean	<element name="l3Validée" type="boolean"/>
- note : double	<element name="note" type="double"/>

*Remarque:* On peut observer dans la deuxième ligne que pour l'élément **prénom**, le type n'a pas été précisé. C'est qu'il s'agit du type par défaut, c'est-à-dire **string**.

#### Types simples

Un type prédéfini permet de définir des données non décomposables en spécifiant des valeurs dans des ensembles prédéfinis (**int**, **double**, **string**, etc.). Ces ensembles sont très généraux et non dédiés à une application ou à un domaine particulier. Par exemple, si l'on choisit le type **double** pour un élément **note**, des notes de -45.7 ou 592 pourront être valides.

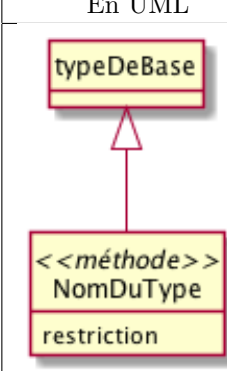
Les *types simples* permettent de réduire le champ des valeurs possibles pour un type prédéfini.

Par définition, un *type simple* (*Simple Type*) est **basé** sur un type prédéfini et le **restreint** à un sous-ensemble de valeurs à l'aide d'une *méthode de restriction*.

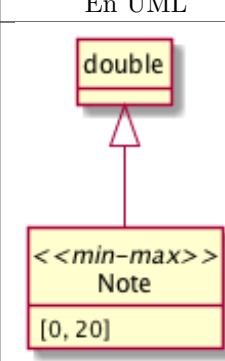
Les méthodes de restriction possibles sont exprimées dans le tableau suivant:

Type prédéfini de base	Méthode de restriction	En XMLSchema
int	«min»	<minInclusive value="..."/> <minExclusive value="..."/>
double	«max»	<maxInclusive value="..."/> <maxExclusive value="..."/>
date	«enumeration»	<enumeration value="..."/>
string	«pattern» «enumeration»	<pattern value="exp. régulière"/> <enumeration value="..."/>

En XMLSchema, un type simple est exprimé à l'aide de l'élément `simpleType` et un sous-élément `restriction` précisant le type de base et la restriction:

En UML	En XMLSchema
	<pre> 1 &lt;simpleType name="NomDuType"&gt; 2   &lt;restriction base="typeDeBase"&gt; 3     &lt;méthode value="..." /&gt; 4     ... 5   &lt;/restriction&gt; 6 &lt;/simpleType&gt; </pre>

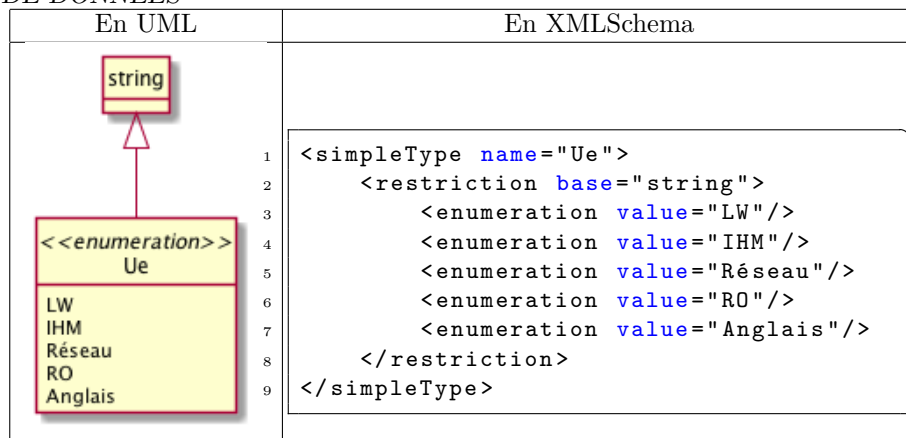
Par exemple, pour exprimer un élément `note` entre 0 et 20, on pourrait utiliser:

En UML	En XMLSchema
	<pre> 1 &lt;simpleType name="Note"&gt; 2   &lt;restriction base="double"&gt; 3     &lt;minInclusive value="0.0" /&gt; 4     &lt;maxInclusive value="20.0" /&gt; 5   &lt;/restriction&gt; 6 &lt;/simpleType&gt; </pre>

Autre exemple, si l'on doit choisir des noms d'UE parmi les UE suivantes:

- LW
- IHM
- Réseau
- RO
- Anglais

On peut alors créer le type simple suivant:



### Expression régulière

Une expression régulière est une expression mathématique qui utilise des symboles prédéfinis pour représenter un ensemble de valeurs. Le tableau suivant résume les principaux symboles que nous serons amenés à utiliser.

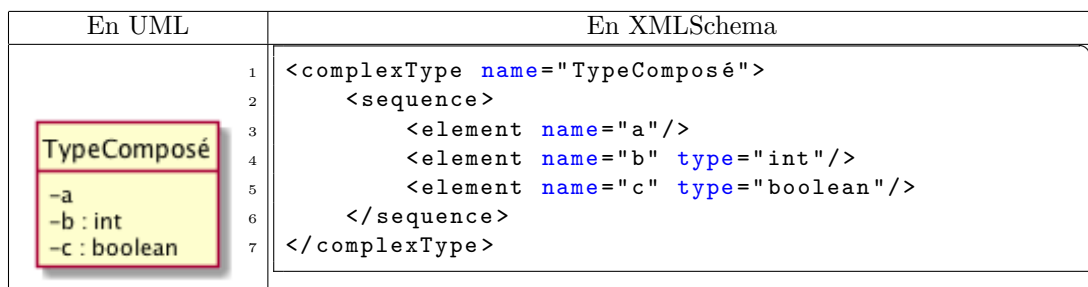
Symbole	Signification
\d	<i>digit</i> = chiffre de 0 à 9
\s	caractère de séparation (espace ou tabulation)
{m}	L'élément précédent est répété m fois. Par exemple \d{m} équivaut à \d\d\d\d\d
{x..y}	L'élément précédent est répété de x à y fois.
[A..E]	Toutes les lettres entre A et E
[1..8]	Tous les chiffres entre 1 et 8

Si l'on souhaite par exemple modéliser un numéro de téléphone sous la forme +33 4 56 52 00 12, on aurait une expression régulière du type `\+\d\d\s\d(\s\d\d){4}`.

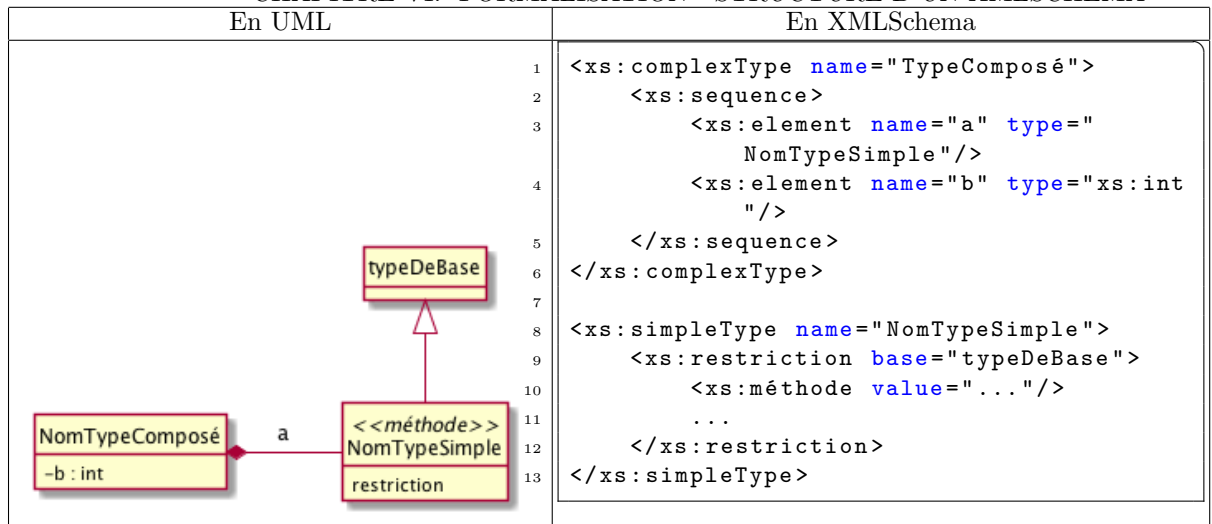
### 4.b Types complexes

La composition permet d'utiliser des types basiques ou complexes dans d'autres types complexes.

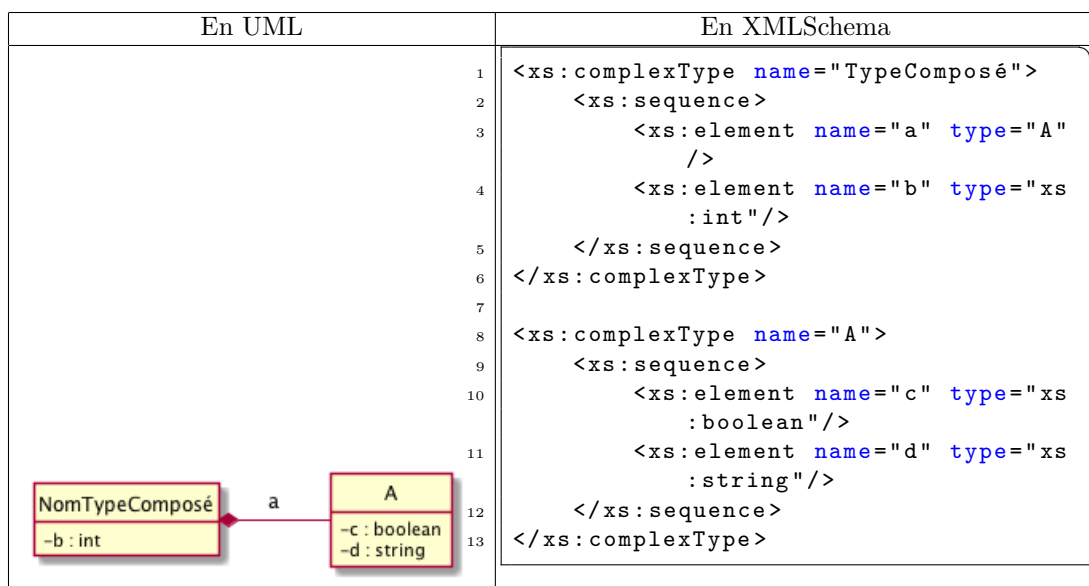
- exemple avec des types prédéfinis:



- exemple avec un type simple



- exemple avec un autre type complexe



Dans les exemples précédents, l'élément `sequence` du langage XMLSchema indique que les sous-éléments devront apparaître dans le même ordre dans les instances XML valides par rapport au schéma. On aurait également pu utiliser l'élément `choice` qui permettrait d'en utiliser 1 parmi les éléments cités ou `group` qui permet d'utiliser tous les éléments cités dans un ordre indifférent.

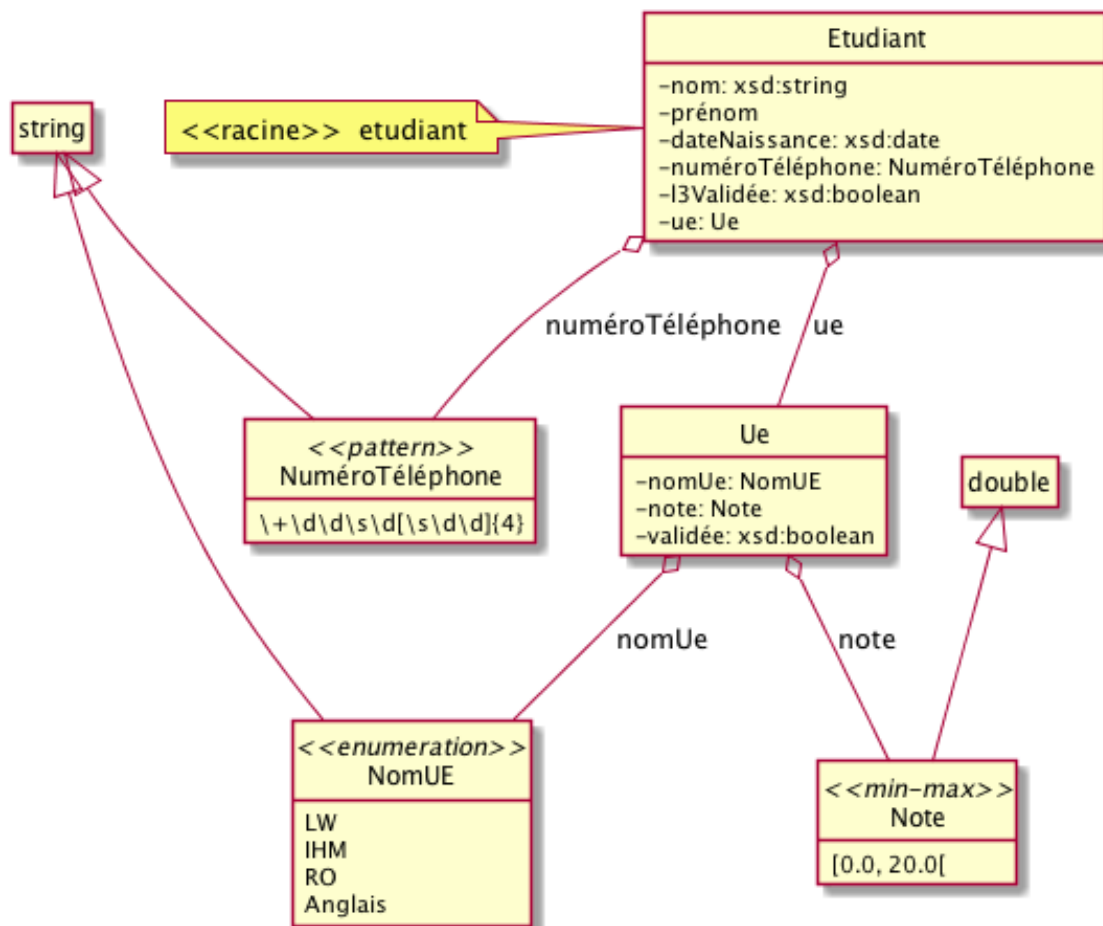
Si l'on reprend la partie l'exemple de la figure VI.1 ci-dessous:

```

1 <?xml version='1.0' encoding='UTF-8' ?>
2 <etudiant>
3   <nom>Kuzbidon</nom>
4   <prénom>Alex</prénom>
5   <dateNaissance>1991-06-15</dateNaissance>
6   <numéroTéléphone>+33 6 00 00 00 00</numéroTéléphone
7   >
8   <l3Validée>true</l3Validée>
9   <ue nom="LW">
10     <note>18.5</note>
11     <validée>true</validée>
12   </ue>
13 </etudiant>

```

On obtient alors le diagramme UML et le XMLSchema suivants:



```

1 <?xml version='1.0' encoding='UTF-8' ?>
2 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
3
4   <xs:element name="etudiant" type="Etudiant"/>
5
6   <xs:complexType name="Etudiant">
7     <xs:sequence>
8       <xs:element name="nom" type="xs:string"/>
9       <xs:element name="prénom"/>
10      <xs:element name="dateNaissance" type="xs:date"/>
11      <xs:element name="numéroTéléphone" type="NuméroTéléphone"/>
12      <xs:element name="l3Validée" type="xs:boolean"/>
13      <xs:element name="ue" type="UE"/>
14    </xs:sequence>
15  </xs:complexType>
16
17  <xs:simpleType name="NuméroTéléphone">
18    <xs:restriction base="xs:string">
19      <xs:pattern value="[+] \d {2} \s \d ( \s \d \d ) {4}"/>
20    </xs:restriction>
21  </xs:simpleType>
22
23  <xs:complexType name="UE">
24    <xs:sequence>
25      <xs:element name="note" type="Note"/>
26      <xs:element name="validée" type="xs:boolean"/>
27    </xs:sequence>
28    <xs:attribute name="nomUE" type="NomUE"/>
  
```

```

29 </xs:complexType>
30
31 <xs:simpleType name="NomUE">
32   <xs:restriction base="xs:string">
33     <xs:enumeration value="LW"/>
34     <xs:enumeration value="IHM"/>
35     <xs:enumeration value="R0"/>
36     <xs:enumeration value="Anglais"/>
37   </xs:restriction>
38 </xs:simpleType>
39
40 <xs:simpleType name="Note">
41   <xs:restriction base="xs:double">
42     <xs:minInclusive value="0"/>
43     <xs:maxExclusive value="20"/>
44   </xs:restriction>
45 </xs:simpleType>
46
47 </xs:schema>

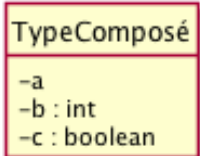
```

## 5 Attributs

Nous avons vu dans le chapitre 2 page 30 qu'il est possible de décrire des notions sous forme de sous-éléments ou bien d'attributs. Dans l'exemple précédent, le choix a été fait de n'utiliser que des sous-éléments.

Si l'on souhaite ajouter un attribut à un élément, alors cet élément devient automatiquement de type complexe. Attention toutefois à garder à l'esprit que le type de l'attribut lui même est forcément *basique*, c'est-à-dire non décomposable.

En UML, la notation des attributs est la même que pour les sous éléments.

En UML	En XMLSchema
 <pre> 1  TypeComposé 2  -a 3  -b : int 4  -c : boolean </pre>	<pre> 1 &lt;complexType name="TypeComposé"&gt; 2   &lt;attribute name="a"/&gt; 3   &lt;attribute name="b" type="int" default="42"/&gt; 4   &lt;attribute name="c" type="boolean" use="required"/&gt; 5 &lt;/complexType&gt; </pre>

*Note* : Si le type composé contient des sous-éléments *et* des attributs, les attributs sont déclarés dans le `complexType` après l'élément `sequence` (ou `choice` ou `group`).

On peut spécifier qu'un attribut est obligatoire en utilisant l'attribut `use` à la valeur `required`. De même, on peut donner une valeur par défaut si l'attribut n'est pas spécifié.

Par contre, on ne peut pas imposer un ordre pour les attributs. Ils pourront apparaître dans le document XML dans n'importe quel ordre.

## 6 Occurrences et Cardinalités

Dans tous les exemples précédents, lorsqu'un type complexe comportait un sous élément `<element name="..." type="..." />`, il en comportait par défaut 1 et 1 seul.

XMLSchema va permettre de préciser le nombre d'occurrences minimum et/ou maximum de chaque élément.

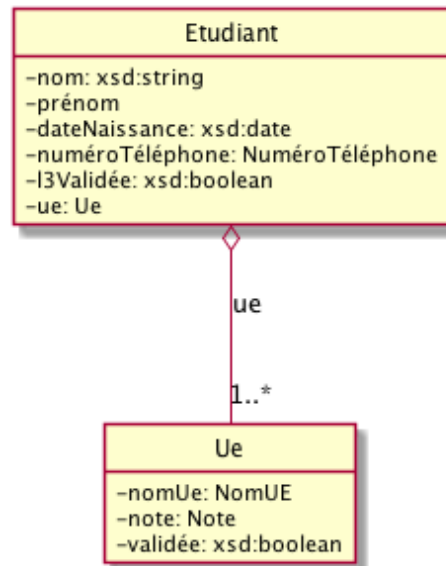
En UML, on va utiliser les nombres et le symbole `*` pour préciser *autant de fois que l'on souhaite*. Ainsi, on aura:

- `0..1` signifie que l'élément peut apparaître au minimum 0 fois et au maximum 1 fois. Cela signifie en fait que cet élément est optionnel.



- 0..\* signifie que l'élément peut apparaître au minimum 0 fois et autant de fois que nécessaire.
- 1..\* signifie que l'élément doit apparaître au moins une fois.
- 1..1 signifie que l'élément doit apparaître une et une seule fois (c'est l'option par défaut).

Par exemple, pour que dans l'exemple VI.1 un étudiant puisse choisir plusieurs options, on utilisera:



En XMLSchema, ce sont les attributs `minOccurs` et `maxOccurs` de l'élément `element` de XMLSchema qui vont permettre de préciser le nombre d'occurrences d'un élément.

```
<element name="nom" type="Type" minOccurs="..." maxOccurs="..." />
```

On utilisera le terme `unbounded` pour préciser *autant de fois que l'on souhaite*.

Par exemple,

- `<element name="ue" type="Ue" minOccurs="0" maxOccurs="1" />` signifie que le champ `ue` est optionnel (présent une fois ou absent).
- `<element name="ue" type="Ue" minOccurs="0" maxOccurs="unbounded" />` signifie que le champ `ue` est optionnel et peut apparaître autant de fois que nécessaire
- `<element name="ue" type="Ue" minOccurs="1" maxOccurs="unbounded" />` signifie que le champ `ue` est obligatoire (doit apparaître au moins une fois) et peut apparaître plusieurs fois.

## 6.a Types complexes - Extensions

Dans les langages de programmation orientés objet, on utilise l'héritage pour définir des objets de plus en plus complexes et la transmission de caractéristiques des objets, les objets fils héritant des caractéristiques (attributs et méthodes) des objets parents. XMLSchema permet de réaliser une forme d'héritage : les types enfants héritant des éléments et attributs des types parents.

En XMLSchema, nous avons vu les restrictions. Nous allons maintenant voir les extensions.

Le type complexe que l'on souhaite étendre et que l'on notera *type complexe étendu* comporte une extension. Une extension a une base, c'est à dire un type qui peut être un type simple (ou prédéfini) ou un type complexe. De la même manière qu'une restriction contenait les contraintes à apporter à la base, l'extension contiendra ce qui étendra le type, par exemple une séquence d'éléments et/ou des attributs.

Lorsque le contenu du type complexe étendu a une base correspondant à un type simple et que l'extension consiste qu'à rajouter des attributs, on dira que ce type complexe étendu a un contenu simple noté `simpleContent`. Dans ce cas, le type complexe étendu prendra la forme suivante :

```

1 <xs:complexType name="TypeComplexeEtendu">
2   <xs:simpleContent>
3     <xs:extension base = "NomTypeSimple">
4       <xs:attribute name="attr1" type="TypeAttribut1"/>
5       <xs:attribute name="attr2" type="TypeAttribut2"/>
6       ...
7     </xs:extension>
8   </xs:simpleContent>
9 </xs:complexType>

```

Lorsque le contenu du type complexe étendu a une base formée d'un type complexe contenant des éléments et/ou qu'on ajoute une séquence d'éléments, alors on dira que ce type complexe étendu a un contenu complexe noté **complexContent**. Dans ce cas, le type complexe étendu prendra la forme suivante :

```

1 <xs:complexType name="TypeComplexeEtendu">
2   <xs:complexContent>
3     <xs:extension base = "NomTypeSimpleOUComplexe">
4       <!-- sequence obligatoire si base = type simple -->
5       <xs:sequence>
6         <xs:element name="nomElement1" type="TypeElement1"/>
7         <xs:element name="nomElement2" type="TypeElement2"/>
8         ...
9       </xs:sequence>
10      <!-- attributs facultatifs -->
11      <xs:attribute name="attr1" type="TypeAttribut1"/>
12      <xs:attribute name="attr2" type="TypeAttribut2"/>
13      ...
14    </xs:extension>
15  </xs:complexContent>
16 </xs:complexType>

```

Voyons maintenant les deux cas de figure, à commencer par les types complexes à contenu simple. Parfois, on souhaite stocker du texte directement dans un élément doté d'attributs, sans que cet élément ne contienne de sous-éléments. Par exemple, considérons `<note coeff="2">12.5</note>`. Comme un tel élément dispose d'attributs (un seul dans l'exemple), il s'agit d'un type complexe. Pour autant, il ne contient pas de séquence. Pour permettre à cet élément de contenir du texte, nous allons indiquer que cet élément est un *contenu simple*, autrement dit un **simpleContent**. Ce contenu simple dont la base sera un type simple, sera en revanche étendu (extension) d'un ou plusieurs attributs. Là où une restriction ajoute des contraintes à une base (un type prédéfini), une extension permet l'ajout de caractéristiques à cette base.

```

1 <xs:simpleType name="ValeurCoeff">
2   <xs:restriction base="xs:double">
3     <xs:minInclusive value="0.0"/>
4   </xs:restriction>
5 </xs:simpleType>
6
7 <xs:complexType name="Note">
8   <xs:simpleContent>
9     <xs:extension base = "xs:double">
10      <xs:attribute name="coeff" type="ValeurCoeff"/>
11    </xs:extension>
12  </xs:simpleContent>
13 </xs:complexType>

```

Remarquons que le type simple qui sert de base à l'extension dans un **simpleContent** n'est pas nécessairement un type prédéfini de XMLSchema. On peut en effet tout à fait utiliser une type simple (comportant des restrictions) comme base pour une telle extension. Par exemple :

```

1 <xs:simpleType name="ValeurCoeff">
2   <xs:restriction base="xs:double">
3     <xs:minInclusive value="0.0"/>

```

```

4      </xs:restriction>
5 </xs:simpleType>
6
7 <xs:simpleType name="ValeurNote">
8     <xs:restriction base="xs:double">
9         <xs:minInclusive value="0.0"/>
10        <xs:maxInclusive value="20.0"/>
11    </xs:restriction>
12 </xs:simpleType>
13
14 <xs:complexType name="Note">
15     <xs:simpleContent>
16         <xs:extension base="ValeurNote">
17             <xs:attribute name="coeff" type="ValeurCoeff"/>
18         </xs:extension>
19     </xs:simpleContent>
20 </xs:complexType>

```

Finissons par un exemple très complet pour illustrer les types complexes étendus à contenu complexe. Considérons l'extension d'une personne pour en faire un étudiant de L3 Miage :

```

1 <?xml version='1.0' encoding='UTF-8' ?>
2 <xs:schema version="1.0"
3     xmlns:xs="http://www.w3.org/2001/XMLSchema"
4     xmlns:etu="http://www.uga.fr/etudiant"
5     targetNamespace="http://www.uga.fr/etudiant"
6     elementFormDefault="qualified">
7
8     <xs:element name="étudiant" type="etu:Etudiant"/>
9
10    <xs:complexType name="Etudiant">
11        <xs:complexContent>
12            <xs:extension base="etu:Contact">
13                <xs:sequence>
14                    <xs:element name="l3Validée" type="xs:boolean"/>
15                    <xs:element name="ue" type="etu:UE" maxOccurs="unbounded"
16                        />
17                </xs:sequence>
18            </xs:extension>
19        </xs:complexContent>
20    </xs:complexType>
21
22    <xs:complexType name="Contact">
23        <xs:complexContent>
24            <xs:extension base="etu:Personne">
25                <xs:sequence>
26                    <xs:element name="téléphone" type="etu:NuméroTéléphone"/>
27                    <xs:element name="adresse" type="etu:Adresse"/>
28                </xs:sequence>
29            </xs:extension>
30        </xs:complexContent>
31    </xs:complexType>
32
33    <xs:complexType name="Adresse">
34        <xs:sequence>
35            <xs:element name="numéro" type="xs:int"/>
36            <xs:element name="rue"/>
37            <xs:element name="codepostal" type="xs:int"/>
38            <xs:element name="ville"/>
39        </xs:sequence>
40    </xs:complexType>
41
42    <xs:simpleType name="NuméroTéléphone">
43        <xs:restriction base="xs:string">

```

```

43     <xs:pattern value="[+]\d{2}\s\d(\s\d\d){4}"/>
44   </xs:restriction>
45 </xs:simpleType>
46
47 <xs:complexType name="Personne">
48   <xs:sequence>
49     <xs:element name="nom"/>
50     <xs:element name="prénom"/>
51     <xs:element name="sexe" type="etu:Sexe"/>
52     <xs:element name="dateNaissance" type="xs:date"/>
53   </xs:sequence>
54 </xs:complexType>
55
56 <xs:simpleType name="Sexe">
57   <xs:restriction base="xs:string">
58     <xs:enumeration value="H"/>
59     <xs:enumeration value="F"/>
60   </xs:restriction>
61 </xs:simpleType>
62
63 <xs:simpleType name="ValeurCoeff">
64   <xs:restriction base="xs:double">
65     <xs:minInclusive value="0.0"/>
66   </xs:restriction>
67 </xs:simpleType>
68
69 <xs:simpleType name="ValeurNote">
70   <xs:restriction base="xs:double">
71     <xs:minInclusive value="0.0"/>
72     <xs:maxInclusive value="20.0"/>
73   </xs:restriction>
74 </xs:simpleType>
75
76 <xs:complexType name="UE">
77   <xs:sequence>
78     <xs:element name="note" type="etu:ValeurNote"/>
79     <xs:element name="validée" type="xs:boolean"/>
80   </xs:sequence>
81
82   <xs:attribute name="nom" type="etu:NomUE"/>
83   <xs:attribute name="coeff" type="etu:ValeurCoeff"/>
84 </xs:complexType>
85
86 <xs:simpleType name="NomUE">
87   <xs:restriction base="xs:string">
88     <xs:enumeration value="FDD"/>
89     <xs:enumeration value="IHM"/>
90     <xs:enumeration value="RO"/>
91     <xs:enumeration value="Anglais"/>
92   </xs:restriction>
93 </xs:simpleType>
94
95 </xs:schema>

```

# Formalisation - Espaces de nom en XMLSchema

## 1 Un schéma = un vocabulaire

Un XMLSchema (schema XSD) définit :

- un nouveau vocabulaire
- un ensemble de règles (syntaxe + sémantique) qui précise comment on doit utiliser les mots du vocabulaire (i.e. comment ils s'organisent les uns par rapport aux autres)



un XSD permet de définir un nouveau langage de description d'un système d'information (de la même façon que le diagramme de classe en UML).

Il existe d'autres façons de définir des langages XML (DTD, RelaxNG) ou autre (grammaire BNF).

XML signifie eXtensible Markup Language. Une des significations de l'extensibilité est qu'XML facilite les mélanges de vocabulaires. Cela le rend extrêmement puissant (par exemple : mélange de XHTML et de SVG). En contrepartie, il doit supporter un mécanisme universel permettant de conserver la cohérence entre les vocabulaires. On doit à tout moment connaître le vocabulaire qui est en train d'être utilisé. Par exemple, l'élément `set` représente un ensemble mathématique dans le langage XML **MathML** et fixe la valeur d'un attribut à une durée de temps précise dans le langage XML **SVG**. Or on souhaite pouvoir utiliser les langages **MathML** et **SVG** dans une même page **XHTML**.

Dans ce chapitre, on va voir :

- comment donner un nom à notre nouveau langage
- comment utiliser plusieurs langages au sein d'un même document XML sans entraîner de confusion.

## 2 Exemple des températures

### 2.a Exemple XML

On considère l'instance XML de la figure [VII.1](#) sur les relevés des températures à un jour donné.

### 2.b XMLSchema

On souhaite décrire ce langage XML sous forme d'un XMLSchema en précisant que les valeurs des températures devront être entières et comprises entre -70 et +80 degrés Celsius.

On décide donc d'écrire un type simple `intDeg` dans le schéma de la figure [VII.2](#) pour décrire une valeur possible de température.

```

1 <?xml version="1.0" encoding="utf8" ?>
2 <température>
3   <min>-6</min>
4   <max>2</max>
5 </température>

```

Figure VII.1: Instance XML de relevé de température.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <schema>
3   <element name="température" type="Température"/>
4   <complexType name="Température">
5     <sequence>
6       <element name="min" type="IntDeg"/>
7       <element name="max" type="IntDeg"/>
8     </sequence>
9   </complexType>
10
11   <simpleType name="IntDeg">
12     <restriction base="int">
13       <minInclusive value="-70"/>
14       <maxInclusive value="80"/>
15     </restriction>
16   </simpleType>
17 </schema>

```

Figure VII.2: Schema XML de relevé de température pour valider l'instance VII.1.

## 2.c Espaces de noms

Dans l'exemple précédent, nous avons regroupé par couleurs les éléments de la figure VII.2. En effet, nous pouvons distinguer 2 types d'éléments dans ce schéma :

- les éléments propres au vocabulaire XMLSchema (aussi appelé xsd), comme la racine du document : **schema**
- les éléments propres au vocabulaire que l'on est en train de définir et qui seront les noms des éléments de nos instances, comme **température**.

Espace de noms XMLSchema Défini par le W3C	Espace de noms de notre langage Défini par nous
<b>schema</b>	<b>température</b>
<b>element, sequence</b>	<b>min, max</b>
<b>complexType, simpleType</b>	<b>Température IntDeg</b>
<b>int</b>	



On peut remarquer que dans les 2 vocabulaires, on a des noms d'éléments (**schema**, **element**, **complexType**, **simpleType**, **température**, **min**, **max**) ainsi que des noms de types (**int**, **Température**, **IntDeg**).

## 3 Définition du vocabulaire et de son nom

### 3.a Noms de vocabulaires

Nous avons vu dans l'exemple précédent que nous avons affaire à 2 vocabulaires/espaces de noms/langages. Pour les reconnaître, nous devons les nommer.

Par convention, et pour garantir l'extensibilité universelle du XML, on choisi d'utiliser, comme nom de vocabulaire/langage XML des URI. URI signifie *Uniform Resource Identifier*. les URLs (*Uniform Resource Locator*) que vous connaissez pour accéder à un site web via un navigateur sont un sous-ensemble des URI. On a donc souvent des noms de vocabulaires/langages XML qui *ressemblent* à des URLs.

Une URI peut être fictive (si vous tapez une URI nom de vocabulaire XML dans une page web, il n'y a souvent rien au bout), mais fait souvent référence à l'organisme garant du vocabulaire en question. Par exemple, de nombreux langages XML développés par le W3C ont un nom qui commence par <http://www.w3.org>. Par exemple, pour le nom de vocabulaire du xhtml est <http://www.w3.org/1999/xhtml>, le vocabulaire du XMLSchema, aussi appelé XSD a pour nom <http://www.w3.org/2001/XMLSchema>, etc.

Dans ce cours, nous utiliserons souvent des URI qui commencent par <http://www.ujf-grenoble.fr/>....

### 3.b En UML

En UML, les espaces de noms sont représentés comme des *packages* autour des éléments correspondants.

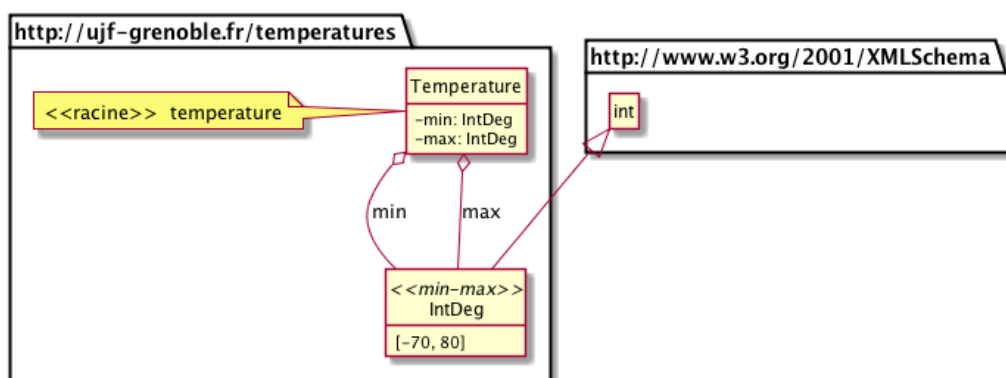


Figure VII.3: Exemple de diagramme UML avec espaces de noms

### 3.c En XMLSchema

On attribue un nom au vocabulaire, au même endroit où on le définit : dans le schema. Dans l'exemple VII.2, la racine du schéma devient:

```

1 <schema targetNamespace="nomDuNouveauVocabulaire" (1)
2   xmlns="nomDuNouveauVocabulaire (ParDéfaut)" (2)
3   xmlns:xsd="http://www.w3.org/2001/XMLSchema" (3)
4   elementFormDefault="qualified"> (4)
5   ...
6 </schema>
```

ou

```

1 <schema targetNamespace="nomDuNouveauVocabulaire" (1)
2   xmlns:ns="nomDuNouveauVocabulaire" (2 bis)
3   xmlns:xsd="http://www.w3.org/2001/XMLSchema" (3)
4   elementFormDefault="qualified"> (4)
5   ...
6 </schema>
```

- (1) `targetNamespace` définit le nom du vocabulaire, c'est-à-dire du nom du nouveau langage que l'on est en train de décrire. Les éléments qui seront définis par ce schéma (par exemple `température`, `min`, `max`) appartiennent à cet espace de noms. Dans le cas de l'exemple de la figure VII.2, `nomDuNouveauVocabulaire` deviendrait <http://www.ujf-grenoble.fr/temperatures>.

- (2) Nom du langage par défaut, c'est-à-dire du langage auquel appartiendront tous les mots qui ne seront pas préfixés. Le langage par défaut est souvent (mais pas toujours) choisi identique au `targetNamespace`. Dans le cas de l'exemple de la figure VII.2, il s'agirait également de `http://www.ujf-grenoble.fr/temperatures`
- (2 bis) Nom du nouveau vocabulaire, ici préfixé. Il est identique au `targetNamespace`. Dans le cas de l'exemple de la figure VII.2, il s'agirait également de `http://www.ujf-grenoble.fr/temperatures`
- (3) Définition du préfixe `xsd`. De manière générale, `xmlns` signifie *XML Name Space* et `xmlns:pref` signifie la définition d'un préfixe `pref`. Ici, `xmlns:xsd="http://www.w3.org/2001/XMLSchema"` signifie que tous les mots qui seront préfixés par `xsd:` appartiendront au vocabulaire XMLSchema (i.e. à l'URI `http://www.w3.org/2001/XMLSchema`).
- (4) `elementFormDefault` est une directive pour tous les documents instances voulant se conformer à ce schéma. Elle force ne effet les documents instances à qualifier (préfixer) l'espace de nom de tous les éléments de ce vocabulaire.

**Remarque :** Votre Schema XML doit TOUJOURS contenir au moins :

- 1 `targetNamespace`. `targetNamespace` signifie "voici le nom du vocabulaire que je définis"
- 2 `xmlns` (dont 1 seul peut ne pas être préfixé). Il peut y en avoir davantage. `xmlns` signifie "voici le nom du vocabulaire que j'utilise"

### 3.d Retour à l'exemple

Le schéma présenté figure VII.2 devient alors:

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <xsd:schema targetNamespace="http://www.ujf-grenoble.fr/temperatures"
3     xmlns="http://www.ujf-grenoble.fr/temperatures">
4     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
5     elementFormDefault="qualified">
6     <xsd:element name="température" type="Température"/>
7     <xsd:complexType name="Température">
8         <xsd:sequence>
9             <xsd:element name="min" type="IntDeg"/>
10            <xsd:element name="max" type="IntDeg"/>
11        </xsd:sequence>
12    </xsd:complexType>
13
14    <xsd:simpleType name="IntDeg">
15        <xsd:restriction base="xsd:int">
16            <xsd:minInclusive value="-70"/>
17            <xsd:maxInclusive value="80"/>
18        </xsd:restriction>
19    </xsd:simpleType>
20 </xsd:schema>

```

Figure VII.4: Schema XML de relevé de température pour valider l'instance VII.1 avec espaces de noms.

### 3.e Cas où le vocabulaire par défaut n'est pas le vocabulaire défini

On a vu plus haut que le vocabulaire défini par défaut, c'est-à-dire sans préfixe (par `xmlns="nomDuVocabulaire"` était souvent le vocabulaire du `targetNamespace`. Mais ce n'est pas obligatoire. Par exemple, on peut définir XMLSchema comme vocabulaire par défaut. En réalité, cette situation est souvent plus pratique car cela évite d'avoir à écrire le préfixe associé au vocabulaire du schéma de partout. Comme il ne peut y avoir qu'un seul vocabulaire par défaut, on doit alors préfixer le vocabulaire `temperature`. Ceci donnerait par exemple le schéma de la figure VII.5:

On peut remarquer dans ce schéma:



```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <schema targetNamespace="http://www.ujf-grenoble.fr/temperatures"
3       xmlns:ttmp="http://www.ujf-grenoble.fr/temperatures">
4       xmlns="http://www.w3.org/2001/XMLSchema"
5       elementFormDefault="qualified">
6       <element name="température" type="ttmp:Température"/>
7       <complexType name="Température">
8         <sequence>
9           <element name="min" type="ttmp:IntDeg"/>
10          <element name="max" type="ttmp:IntDeg"/>
11        </sequence>
12      </complexType>
13
14      <simpleType name="IntDeg">
15        <restriction base="int">
16          <minInclusive value="-70"/>
17          <maxInclusive value="80"/>
18        </restriction>
19      </simpleType>
20 </schema>

```

Figure VII.5: Schema XML de relevé de température pour valider l'instance VII.1 avec espaces de noms.

- les éléments du vocabulaire `http://www.w3.org/2001/XMLSchema` (`schema`, `element`, `complexType`, etc.) appartiennent au vocabulaire par défaut et ne sont donc plus préfixés par `xsd:`.
- de même le type `int` ligne 15 appartient au vocabulaire par défaut et n'est donc plus préfixé par `xsd:`.
- la définition du nom des types (complexes ou simples) du vocabulaire en cours de définition n'a pas besoin d'être préfixé. En effet, lignes 7 et 14, on sait que l'on est en train de définir des types du vocabulaire en cours de définition et par conséquent, `Température` ligne 7 et `IntDeg` ligne 14 appartiennent forcément au `targetNamespace` et donc au vocabulaire `http://www.ujf-grenoble.fr/temperature`.
- de même, le nom des éléments qui sont en train d'être définis (lignes 6, 9 et 10) n'ont pas besoin d'être préfixés car ils appartiennent obligatoirement au `targetNamespace`.

### 3.f Peut-il n'y avoir aucun vocabulaire par défaut dans un XMLSchema?

Oui, bien sûr ! On peut décider de tout préfixer. Mais comme dit précédemment, il est souvent astucieux de poser le vocabulaire XMLSchema comme vocabulaire par défaut et préfixer le vocabulaire que l'on définit.

## 4 Contraindre un document XML à un schema

On souhaite ici contraindre un document XML à un schéma, c'est-à-dire lier le document XML à un schéma de façon à ce que sa validité par rapport au schéma puisse être testée automatiquement.

```

1 <racine xmlns="nomDuVocabulaireParDéfaut" (1)
2     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" (2)
3     xsi:schemaLocation="NomDuVocabulaireUtilisé schema.xsd"> (3)
4     ...
5 </racine>

```

- (1) `xmlns="nomDuVocabulaireParDéfaut"` signifie ici dans l'instance, la même chose que dans le schéma, c'est-à-dire qu'il désigne le vocabulaire par défaut, celui auquel appartiennent les éléments non préfixés. Il s'agit en général du vocabulaire défini dans le schéma.
- (2) `xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"` définit le préfixe `xsi` pour désigner le langage XMLSchema-Instance (note, c'est un langage différent de XMLSchema). Ce préfixe sert en fait juste pour la ligne d'après étant donné que l'attribut `schemaLocation` de la racine n'a pas été défini dans notre vocabulaire mais appartient à XMLSchema-Instance
- (3) L'attribut `schemaLocation` (appartenance au langage XMLSchema-Instance) permet de spécifier le fichier `xsd` dans lequel est défini le vocabulaire `NomDuVocabulaireUtilisé`. On peut noter que le nom du vocabulaire et le nom du fichier sont dans les mêmes guillemets de valeur de l'attribut `schemaLocation` séparés par un espace. Le nom du fichier est en réalité un chemin relatif pour accéder au schéma depuis l'instance. Par exemple, si le schéma était situé dans le répertoire parent de celui dans lequel l'instance était situé, on aurait: `xsi:schemaLocation="NomDuVocabulaireUtilisé ../schema.xsd"`.

#### 4.a Retour à l'exemple

Supposons que le schéma de la figure VII.4 (ou de la figure VII.5 qui est équivalent) est situé dans le même répertoire que l'instance `temperatures.xml` sous le nom `temperatures.xsd`.

On aurait alors le document suivant:

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <temperature xmlns="http://ujf-grenoble.fr/temperatures"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xsi:schemaLocation="http://ujf-grenoble.fr/temperatures
5         temperatures.xsd">
6     <min>-6</min>
7     <max>2</max>
8 </temperature>

```

Ce document va pouvoir être validé automatiquement par rapport au schéma du fichier `temperatures.xsd` par NetBeans par exemple.

#### 4.b Cas où le vocabulaire par défaut n'est pas le vocabulaire utilisé

On peut souhaiter ne pas utiliser de vocabulaire par défaut. A ce moment, on utilise un préfixe en utilisant `xmlns:pref="NomDuVocabulairePréfixé"` pour définir le préfixe `pref`.

On obtient alors cet exemple d'instance XML:

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <tmp:temperature xmlns:tmp="http://ujf-grenoble.fr/temperatures"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xsi:schemaLocation="http://ujf-grenoble.fr/temperatures
5         temperatures.xsd">
6     <tmp:min>-6</tmp:min>
7     <tmp:max>2</tmp:max>
8 </tmp:temperature>

```

#### 4.c Peut-il n'y avoir aucun vocabulaire par défaut dans l'instance de document XML ?

Oui, encore une fois, bien sûr ! On peut décider de tout préfixer. Cette stratégie est même ABSOLUMENT OBLIGATOIRE lorsqu'on écrit des feuilles de transformation XSLT : il FAUT préfixer le vocabulaire XSLT **et** tous les autres vocabulaires utilisés (ceux du ou des documents XML sur lesquels s'appliquent la transformation).

## 5 A qui appartiennent les éléments/attributs ?

Voici à présent quelques règles pour identifier les espaces de noms (vocabulaires) des éléments:

- Les éléments sans préfixe sont dans le vocabulaire par défaut (défini par `xmlns="NomDuVocabulaireParDéfaut"`).
- Si aucun vocabulaire par défaut n'est défini, les éléments sans préfixe sont sans vocabulaire (sans espace de nom ou *namespace*).
- Les attributs sans préfixe sont **sans espace de nom**, même s'ils appartiennent à une balise qui a un espace de nom.

Nous verrons que certains préfixes sont canoniques (c'est-à-dire qu'ils sont *réservés* à certains vocabulaires) comme `xsd`, `rdf`, `svg`, etc.

### 5.a Règles pour connaître l'espace de noms d'un élément

- Si l'élément a un préfixe,
  - On cherche la définition du préfixe en remontant vers les ancêtres de l'élément.
  - S'il n'y a pas de définition du préfixe dans les ancêtres, c'est qu'il y a une **erreur**.
- Si l'élément n'a pas de préfixe,
  - On cherche la définition du vocabulaire par défaut en remontant vers les ancêtres de l'élément.
  - S'il n'y a pas de définition du vocabulaire par défaut, l'élément n'a **pas d'espace de nom**.

### 5.b Règles pour connaître l'espace de noms d'un attribut

- Si l'attribut a un préfixe
  - On cherche la définition du préfixe en remontant vers les ancêtres
  - Si le préfixe n'est pas défini, il y a une **erreur**
- Si l'attribut n'a pas de préfixe, **il n'a pas d'espace de nom**.



# Contraintes de cohérence : Unicité et Existence

XMLSchema permet de spécifier des contraintes de cohérence qui s'appliquent globalement sur un document (instance XML). Un tel document utilisant un schéma contraint devra non seulement être valide par rapport à son schéma, mais en plus vérifier des contraintes de cohérence.

En XMLSchema v1.0, les contraintes de cohérence sont de deux types :

- les contraintes d'unicité
- les contraintes d'existence

Mais qu'est ce qu'une contrainte de cohérence ? Le plus simple est de prendre un exemple concret : Supposons que nous modélisons un centre de soin. Il paraît évident que les personnels de soin doivent avoir un identifiant (l'**existence** de l'identifiant doit être vérifiée) et que cet identifiant est unique (l'**unicité** de cet identifiant doit être respectée) de façon à ce qu'aucune personne ne puisse avoir le même identifiant.

NB : En XMLSchema v1.1, il existe en plus de cela les **assertions**. Ce sont des composants de XML Schema 1.1 qui permettent de contraindre l'existence et les valeurs relatives à des éléments et/ou des attributs. Un autre exemple pourrait être de vérifier que la valeur d'un attribut, par exemple **temperatureMin**, est bien inférieure ou égale à celle d'un autre attribut (respectivement **temperatureMax**). Dans ce cours, nous ne traiterons pas des composants au delà de la version v1.0, la première raison étant que la prise en charge de cette version 1.1 est assez compliquée et demande l'installation du processeur XML Saxon, la deuxième étant que son installation est payante !

## 1 Contraintes d'unicité

Une contrainte d'unicité spécifie qu'il ne peut exister qu'un seul élément ou attribut d'une propriété fixée dans un élément donné. Cette notion correspond à celle des clefs des bases de données.

En XMLSchema, l'unicité est obtenue en utilisant les éléments `xsd:key` ou `xsd:unique`.

### 1.a Définir une contrainte d'unicité

#### Un exemple complet

Commençons par détailler un exemple complet de schéma contraint : la modélisation d'une matrice mathématique. En mathématiques, une matrice est un tableau de coefficients spécifiant une transformation applicable sur des vecteurs. On peut modéliser une matrice comme une collection de lignes (rows) contenant chacune le même nombre de cases (cells). Notons qu'un vecteur est une matrice à une seule dimension. Le schéma XML montré figure [VIII.1](#) modélise une matrice.

Dans ce schéma XML, 2 contraintes d'unicité sont appliquées : une sur les numéros de lignes, l'autre sur les numéros de cases dans chaque ligne.

- Les cellules doivent avoir un numéro `cellNo` unique défini par une clef `cellNoKey`. Cette clef est déclarée et définie lors de la déclaration de l'élément `row` : `mat:Row`. C'est en effet quand une ligne (`row`) est déclarée que la portée de la clef est définie et s'applique aux cellules de la ligne.
- De même, les lignes ont un numéro `rowNo` unique défini par une clef `rowNoKey`. Cette clef est déclarée et définie lors de la déclaration de l'élément `matrix` : `mat:Matrix`. C'est en effet quand une matrice (`matrix`) est déclarée que la portée de la clef est définie et s'applique aux lignes de la matrice.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <!-- SHEMA 1 -->
3 <xsd:schema version="1.0"
4     targetNamespace="https://www.timc.imag.fr/matrix"
5     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
6     xmlns:mat="https://www.timc.imag.fr/matrix"
7     elementFormDefault="qualified">
8
9     <!-- Déclaration de l'élément racine 'matrix:Matrix' -->
10    <xsd:element name="matrix" type="mat:Matrix">
11        <xsd:unique name="rowNoKey">
12            <xsd:selector xpath="mat:row" />
13            <xsd:field xpath="@rowNo" />
14        </xsd:unique>
15    </xsd:element>
16
17    <!-- Définition du type complexe 'Cell' (cellules, c'est à dire les cases
18         de la matrice) -->
19    <xsd:complexType name="Cell">
20        <xsd:attribute name="cellNo" type="xsd:int" use="required"/>
21        <xsd:attribute name="value" type="xsd:int" use="required"/>
22    </xsd:complexType>
23
24    <!-- Définition du type complexe 'Row' (ligne de la matrice) -->
25    <xsd:complexType name="Row">
26        <xsd:sequence>
27            <xsd:element name="cell" type="mat:Cell" maxOccurs="unbounded" />
28        </xsd:sequence>
29        <xsd:attribute name="rowNo" type="xsd:int" use="required"/>
30    </xsd:complexType>
31
32    <!-- Définition du type complexe 'Matrix' (une matrice) -->
33    <xsd:complexType name="Matrix">
34        <xsd:sequence>
35            <xsd:element name="row" type="mat:Row" maxOccurs="unbounded">
36                <xsd:unique name="cellNoKey">
37                    <xsd:selector xpath="mat:cell" />
38                    <xsd:field xpath="@cellNo" />
39                </xsd:unique>
40            </xsd:element>
41        </xsd:sequence>
42        <xsd:attribute name="matName" type="xsd:string"/>
43    </xsd:complexType>
44 </xsd:schema>

```

Figure VIII.1: Modélisation d'une matrice avec 2 contraintes d'unicité (Schema XML Matrix.xsd).

Un instance de document XML valide par rapport à ce schéma pourra être celle montrée figure VIII.2.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <!-- INSTANCE 1 : une matrice 3x3 -->
3 <matrix
4   xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
5   xmlns='https://www.timc.imag.fr/matrix'
6   xsi:schemaLocation='https://www.timc.imag.fr/matrix Matrix.xsd'
7   matName="m0">
8   <row rowNo="0">
9     <cell cellNo="0" value="1"/><cell cellNo="1" value="0"/><cell cellNo="2
      " value="0"/>
10  </row>
11  <row rowNo="1">
12    <cell cellNo="0" value="0"/><cell cellNo="1" value="1"/><cell cellNo="2
      " value="0"/>
13  </row>
14  <row rowNo="2">
15    <cell cellNo="0" value="0"/><cell cellNo="1" value="0"/><cell cellNo="2
      " value="1"/>
16  </row>
17 </matrix>

```

Figure VIII.2: Document XML contraint par le schéma XML `Matrix.xsd`. Une matrice 3x3 est instanciée. Elle contient 3 lignes (row) numérotées de 0 à 2, contenant chacune 3 cellules (cell) numérotées de 0 à 2 également, contenant les valeurs des coefficients de la matrice.

### Où déclarer une contrainte d'unicité

Les éléments `xsd:unique` ou `xsd:key` doivent être *éléments fils* d'un l'élément ; cet élément contient ceux sur lesquels va s'appliquer la contrainte d'unicité.

Dans l'exemple donné figure VIII.1, on voit effectivement que la contrainte `cellNoKey` s'appliquant sur les cellules d'une ligne, est élément fils de l'élément `row` : `mat:Row` qui contient les éléments de type `mat:Cell` sur lesquels s'applique cette contrainte. De même, la contrainte `rowNoKey` est élément fils de l'élément `matrix` : `mat:Matrix` qui contient les éléments de type `mat:Row` sur lesquels s'applique cette contrainte.

### Attributs de la contrainte

Les éléments `xsd:key` ou `xsd:unique` contiennent :

- un attribut `id` qui est optionel. Cet identifiant de clef est unique.
  - un attribut `name` qui est requis
  - deux types d'éléments:
    - `xsd:selector` qui est requis. Une seule occurrence doit être présente.
    - `xsd:field`. Au moins une occurrence doit être présente, mais il peut y en avoir plusieurs.
- qui ont chacun un seul élément `xpath`.

Ainsi, les éléments définissant une contrainte d'unicité s'écrivent :

```

1   <xsd:unique name="nom_de_la_clef" id="identifiant_de_la_clef">
2     <xsd:selector xpath="éléments sur lesquels porte la contrainte" />
3     <xsd:field xpath="éléments ou attributs spécifiant l'unicité" />
4     <xsd:field xpath="autres éléments/attributs ...." />
5     ...
6   </xsd:unique>

```

ou

```

1      <xsd:key name="nom_de_la_clef" id="identifiant_de_la_clef">
2          <xsd:selector xpath="éléments sur lesquels porte la contrainte" />
3          <xsd:field xpath="éléments ou attributs spécifiant l'unicité" />
4          <xsd:field xpath="autres éléments/attributs ...." />
5          ...
6      </xsd:unique>

```

Figure VIII.3: Structure d'une clef d'unicité.

L'attribut `name` est utilisé uniquement par les contraintes d'existence `xsd:keyref` (voir la section 2. *Contraintes d'existence* plus bas). Si seule l'unicité est appliquée, alors cet attribut peut être ignoré (mais doit être présent avec une valeur de nom quelconque).

L'élément `xsd:selector`, via un chemin XPath spécifié dans son attribut `path`, spécifie sur quels éléments s'applique la contrainte d'unicité. Ces éléments, de même type, peuvent être multiples : il peut s'agir d'un seul élément comme d'une liste d'élément. Par exemple, dans la figure VIII.1, le sélecteur de la clef `cellNoKey` s'applique à la liste des noeuds (éléments ici) obtenue par le chemin xpath `mat:cell`, autrement dit toutes les cellules contenues dans une ligne.

L'élément `xsd:field` permet, via son chemin XPath spécifié dans son attribut `path`, de spécifier la valeur qui doit être rendue unique. Dans le même exemple de la clef `cellNoKey` (figure VIII.1), c'est l'attribut `cellNo` des cellules d'une ligne qui est ciblé et qui doit être unique. Plusieurs champs `xsd:field` peuvent être présents pour forcer l'unicité de plusieurs valeurs en même temps (voir la section dédiée ci dessous).

### 1.b `xsd:unique` et `xsd:key`

Ces deux éléments permettent de définir une contrainte d'unicité sur des valeurs. La différence entre les deux est un critère d'existence:

- une clef de type `xsd:key` est unique mais exige que le champs (`field`) existe pour chaque résultat du sélecteur dans le schéma et dans le document XML. La réciproque est que chaque élément dans le selecteur doit avoir une clef.
- ce n'est pas requis pour une clef de type `xsd:unique` : elle ne nécessite pas que le sélecteur contienne le champ à contraindre.

Dans le document d'exemple, figure VIII.1, les deux contraintes sont définies avec `xsd:unique`. Si maintenant on utilise un élément `xsd:key` pour définir la contrainte sur les numéros de cellules (`cellNo`), comme dans la figure VIII.4, la présence de l'attribut `cellNo` dans les éléments `mat:Cell` sera obligatoire (en supposant que le schéma, pour cet attribut, ne spécifie pas `use="required"` comme dans la figure VIII.1).

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <!-- SHEMA 1 -->
3  <xsd:schema version="1.0"
4      targetNamespace="https://www.timc.imag.fr/matrix"
5      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
6      xmlns:mat="https://www.timc.imag.fr/matrix"
7      elementFormDefault="qualified">
8
9      <!-- Déclaration de l'élément racine 'matrix:Matrix' -->
10     <xsd:element name="matrix" type="mat:Matrix">
11         <xsd:unique name="rowNoKey">
12             <xsd:selector xpath="mat:row" />
13             <xsd:field xpath="@rowNo" />
14         </xsd:unique>
15     </xsd:element>
16
17     <!-- Définition du type complexe 'Cell' (cellules, c'est à dire les cases
18         de la matrice) -->
19     <xsd:complexType name="Cell">

```



```

19         <xsd:attribute name="cellNo" type="xsd:int"/>
20         <xsd:attribute name="value" type="xsd:int" use="required"/>
21     </xsd:complexType>
22
23     <!-- Définition du type complexe 'Row' (ligne de la matrice) -->
24     <xsd:complexType name="Row">
25         <xsd:sequence>
26             <xsd:element name="cell" type="mat:Cell" maxOccurs="unbounded" />
27         </xsd:sequence>
28         <xsd:attribute name="rowNo" type="xsd:int"/>
29     </xsd:complexType>
30
31     <!-- Définition du type complexe 'Matrix' (une matrice) -->
32     <xsd:complexType name="Matrix">
33         <xsd:sequence>
34             <xsd:element name="row" type="mat:Row" maxOccurs="unbounded">
35                 <xsd:key name="cellNoKey">
36                     <xsd:selector xpath="mat:cell" />
37                     <xsd:field xpath="@cellNo" />
38                 </xsd:key>
39             </xsd:element>
40         </xsd:sequence>
41         <xsd:attribute name="matName" type="xsd:string"/>
42     </xsd:complexType>
43
44 </xsd:schema>

```

Figure VIII.4: Utilisation de `xsd:key` au lieu de `xsd:unique` pour définir la contrainte d'unicité sur les numéros de cellules.

Dans ce cas, tous les éléments `mat:cell` présents dans le document XML doivent contenir un attribut `cellNo`, comme montré dans l'instance VIII.5. Par contre, les attributs `rowNo` des éléments `mat:row` ne sont pas requis : il n'est pas spécifié `use='required'` et la contrainte d'unicité qui s'applique sur eux est définie avec un élément `xsd:unique` qui n'impose pas de conditions d'existence.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <!-- INSTANCE 2 : une matrice 3x3 -->
3 <matrix
4     xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
5     xmlns='https://www.timc.imag.fr/matrix'
6     xsi:schemaLocation='https://www.timc.imag.fr/matrix Matrix.xsd'
7     matName="m0">
8     <row>
9         <cell cellNo="0" value="1"/><cell cellNo="1" value="0"/><cell cellNo="2
10             " value="0"/>
11     </row>
12     <row>
13         <cell cellNo="0" value="0"/><cell cellNo="1" value="1"/><cell cellNo="2
14             " value="0"/>
15     </row>
16     <row>
17         <cell cellNo="0" value="0"/><cell cellNo="1" value="0"/><cell cellNo="2
18             " value="1"/>
19     </row>
20 </matrix>

```

Figure VIII.5: Document XML contraint par le schéma XML `Matrix.xsd`. Il est semblable à celui montré figure VIII.2 au détail que les attributs `rowNo`, non requis, ne sont pas présents ; leur unicité était spécifiée, pas leur existence. Ce document est tout à fait valide.

En revanche, le document présenté figure VIII.6 n'est pas valide car il manque un attribut `cellNo` dont l'existence est imposée par la contrainte définie avec un élément `xsd:key`.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <!-- INSTANCE 3 : une matrice 3x3 -->
3 <matrix
4   xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
5   xmlns='https://www.timc.imag.fr/matrix'
6   xsi:schemaLocation='https://www.timc.imag.fr/matrix Matrix.xsd'
7   matName="m0">
8   <row>
9     <cell value="1"/><cell cellNo="1" value="0"/><cell cellNo="2" value="0"
10    />
11   </row>
12   <row>
13     <cell cellNo="0" value="0"/><cell cellNo="1" value="1"/><cell cellNo="2
14     " value="0"/>
15   </row>
16   <row>
17     <cell cellNo="0" value="0"/><cell cellNo="1" value="0"/><cell cellNo="2
18     " value="1"/>
19   </row>
20 </matrix>

```

Figure VIII.6: Document XML contraint par le schéma XML *Matrix.xsd*. Il est semblable à celui montré figure VIII.5 mais cette fois, **il manque un attribut `cellNo` requis** (celui de la première cellule). **Ce document n'est pas valide.**

### 1.c Portée de la contrainte d'unicité

De la même manière qu'une variable ou un attribut de classe en Java dispose d'une portée, une contrainte d'unicité a une portée : la contrainte d'unicité ne pourra s'appliquer que sur les valeurs contenues dans l'élément

Revenons à notre exemple de matrice (figure VIII.2) : dans une même ligne, les cellules doivent avoir des numéros différents, mais des cellules de lignes différentes peuvent avoir des numéros égaux car la portée d'unicité sur les numéros de cellules ne s'applique que pour chaque ligne. Pour s'en convaincre, il suffit de voir que cette contrainte d'unicité pour les numéros de lignes est définie lors de la déclaration de l'élément `row` : `mat:Row` : autrement dit, pour chaque occurrence d'élément `row` : `mat:Row` qui sera créée dans la matrice, une nouvelle contrainte d'unicité pour les numéros de cellules sera définie.

### 1.d Contrainte d'unicité sur plusieurs valeurs

Dans une définition de contrainte d'unicité, il peut y avoir plusieurs éléments `xsd:field`. Cela permet de définir une clef d'unicité en utilisant plusieurs valeurs à la fois.

Attention ! Deux valeurs sont considérées différentes si elles diffèrent sur au moins 1 champ. Il n'est donc pas nécessaire que les deux valeurs soient uniques pour définir l'unicité. C'est la combinaison des deux valeurs qui fait l'unicité. Par exemple, si l'on considère les deux champs *nom* et *prénom* qui définissent l'identité d'une personne, la combinaison { *Nicolas*, *Glade* } diffère de la combinaison { *Nicolas*, *Cage* }.

Un exemple plus détaillé est donné figure VIII.7. Dans ce schéma XML, on modélise une liste d'opérations matricielles. Ces opérations sont de type 'addition' (add), soustraction (sub), multiplication (mul). Une opération `Operation` est donc définie par un type d'opération (attribut `opType`), mais possède aussi deux autres attributs contribuant à l'identifier de façon unique, son identifiant `opID` et son nom `opName`. De plus, chaque opération se réfère, sous forme d'attributs, à deux matrices par leur nom dans deux attributs supplémentaires, `left` et `right`, qui indiquent la matrice à gauche de l'opérateur et la matrice à droite. Enfin, une opération spécifie le nom de la matrice résultante de l'opération dans un dernier attribut `result`. Au total, une opération a 6 attributs.

On remarque qu'une clef unique nommée `matUnique` est définie comme élément fils de l'élément `matrices`, une liste de matrices. Elle s'applique sur tous les éléments `mop:matrix` contenus dans cette liste, rendant ainsi chaque matrice unique par son nom `matName`.

De plus, une clef unique nommée `opUnique` est définie comme élément fils de l'élément `operations`.

Cette clef s'applique sur l'ensemble des opérations (`mop:operation`) et son unicité est définie grâce à trois champs : l'identifiant de l'opération `opID`, son nom `opName` et le type de l'opération `opType`. L'usage de ces trois valeurs à la fois permet de garantir l'unicité d'une opération. Il est tout à fait possible d'avoir 2 opérations de type multiplication (*mul*) mais avec des identifiants différents ou des noms différents, comme il est possible d'avoir 2 opérations de type différent (par exemple multiplication et soustraction), mais d'identifiants et/ou de noms égaux (voir par exemple l'instance de document XML figure VIII.8.

```

1 <?xml version="1.0"?>
2 <xsd:schema version="1.0"
3     targetNamespace="https://www.timc.imag.fr/matrix"
4     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
5     xmlns:mat="https://www.timc.imag.fr/matrix"
6     elementFormDefault="qualified">
7     <xsd:include schemaLocation="./Matrix.xsd"/>
8
9     <!-- Elément racine 'matrixComputation' -->
10    <xsd:element name="matrixComputations">
11        <xsd:complexType>
12            <xsd:sequence>
13
14                <!-- liste de matrices -->
15                <xsd:element name="matrices" maxOccurs="unbounded">
16                    <xsd:complexType>
17                        <xsd:sequence>
18                            <xsd:element name="matrix" type="mat:Matrix"
19                                minOccurs="1" maxOccurs="unbounded"/>
20                        </xsd:sequence>
21                    </xsd:complexType>
22                    <!-- Unicité des matrices -->
23                    <xsd:key name="matUnique">
24                        <xsd:selector xpath="mat:matrix"/>
25                        <xsd:field xpath="@matName"/>
26                    </xsd:key>
27                </xsd:element>
28
29                <!-- liste d'opérations -->
30                <xsd:element name="operations" maxOccurs="unbounded">
31                    <xsd:complexType>
32                        <xsd:sequence>
33                            <xsd:element name="operation" type="mat:Operation"
34                                minOccurs="1" maxOccurs="unbounded"/>
35                        </xsd:sequence>
36                    </xsd:complexType>
37                    <!-- Unicité des opérations -->
38                    <xsd:key name="opUnique">
39                        <xsd:selector xpath="mat:operation" />
40                        <xsd:field xpath="@opID" />
41                        <xsd:field xpath="@opName" />
42                        <xsd:field xpath="@opType" />
43                    </xsd:key>
44                </xsd:element>
45            </xsd:sequence>
46        </xsd:complexType>
47    </xsd:element>
48
49    <!-- Définition du type complexe 'Operation' (une opération matricielle) -->
50    <xsd:complexType name="Operation">
51        <xsd:attribute name="opID" type="xsd:int" use="required"/>
52        <xsd:attribute name="opName" type="xsd:string" use="required"/>
53        <xsd:attribute name="opType" type="mat:MatOpType" use="required"/>
54        <xsd:attribute name="left" type="xsd:string" use="required"/>
55        <xsd:attribute name="right" type="xsd:string" use="required"/>

```

```

54     <xsd:attribute name="result" type="xsd:string" use="required"/>
55 </xsd:complexType>
56
57 <!-- Définition du type complexe 'MatOpType' (types possibles pour une opé
58 ration matricielle) -->
59 <xsd:simpleType name="MatOpType">
60     <xsd:restriction base="xsd:string">
61         <xsd:enumeration value="add"/>
62         <xsd:enumeration value="sub"/>
63         <xsd:enumeration value="mul"/>
64     </xsd:restriction>
65 </xsd:simpleType>
66 </xsd:schema>

```

Figure VIII.7: Schema XML (MatrixOperations.xsd) modélisant une liste d'opérations matricielles entre deux matrices.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <!-- INSTANCE 4 : des opérations sur des matrices -->
3 <matrixComputations
4     xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
5     xmlns='https://www.timc.imag.fr/matrix'
6     xsi:schemaLocation='https://www.timc.imag.fr/matrix MatrixOperations.xsd'>
7     <matrices>
8         <matrix matName="translate">
9             <row rowNo="0">
10                 <cell cellNo="0" value="1"/><cell cellNo="1" value="0"/><cell
11                     cellNo="2" value="2"/>
12             </row>
13             <row rowNo="1">
14                 <cell cellNo="0" value="0"/><cell cellNo="1" value="1"/><cell
15                     cellNo="2" value="-1"/>
16             </row>
17             <row>
18                 <cell cellNo="0" value="0"/><cell cellNo="1" value="0"/><cell
19                     cellNo="2" value="1"/>
20             </row>
21         </matrix>
22         <matrix matName="colVect1">
23             <row rowNo="0">
24                 <cell cellNo="0" value="1"/>
25             </row>
26             <row rowNo="1">
27                 <cell cellNo="0" value="0"/>
28             </row>
29             <row>
30                 <cell cellNo="0" value="1"/>
31             </row>
32         </matrix>
33         <matrix matName="rowVect">
34             <row rowNo="0">
35                 <cell cellNo="0" value="1"/><cell cellNo="1" value="-1"/><cell
36                     cellNo="2" value="2"/>
37             </row>
38         </matrix>
39     </matrices>
40     <operations>
41         <operation opID="1" opName="translation" opType="mul" left="translate"
42             right="colVect1" result="colVect2"/>
43         <operation opID="2" opName="scalarProduct" opType="mul" left="rowVect"
44             right="colVect1" result="scalar"/>
45     </operations>
46 </matrixComputations>

```

```

39     </operations>
40 </matrixComputations>

```

Figure VIII.8: Document XML contraint par le schéma XML `MatrixOperations.xsd`.

### 1.e Les expression XPath des sélecteurs et des champs

Notez que les expressions XPath des sélecteurs et des champs (`xsd:selector` et `xsd:field`) sont restreintes:

- Avec le sélecteur, on ne peut sélectionner que des éléments descendants de l'élément dans lequel la clefs d'unicité a été déclarée.
- Le chemin XPath défini dans le champ est relatif aux éléments sélectionnés dans le sélecteur. Ce sont uniquement des éléments ou attributs descendants.
- Les seuls opérateurs disponibles sont l'opérateur de chemin `/` et l'opérateur d'union `|`.
- Les axes autorisés sont les axes enfants `/`, d'attribut `@` et l'axe descendant `..`
- Aucun filtre n'est autorisé.



Attention à bien respecter les préfixes associés aux espaces de nom dans ces chemins XPath !

## 2 Contraintes d'existence

Une contrainte d'existence spécifie qu'il doit exister un élément ou attribut d'une propriété fixée dans un élément donné. Pour être plus précis, une contrainte d'existence permet de lier une contrainte d'unicité (ayant des propriétés de contrainte d'existence dans le cas des éléments `xsd:key`) à une valeur dont on requiert l'existence.

C'est l'élément `xsd:keyref` qui permet de déclarer et définir une contrainte d'existence.

L'exemple typique est celui d'une commande faite en ligne à partir d'un catalogue. Une contrainte d'existence peut imposer le fait qu'un produit commandé existe dans le catalogue en liant la référence du produit commandé à l'existence de sa référence dans le catalogue.

### 2.a Où déclarer une contrainte d'existence

Les éléments `xsd:keyref` doivent être *éléments fils* d'un élément ; cet élément contient ceux sur lesquels va s'appliquer la contrainte d'existence. Comme pour les contraintes d'unicité, la portée de la contrainte d'existence est définie à partir du niveau où elle est déclarée.

### 2.b Attributs de la contrainte

Les éléments `xsd:keyref` contiennent :

- un attribut `name` qui est requis
- un attribut `refer` qui est requis.
- deux types d'éléments:
  - `xsd:selector` qui est requis. Une seule occurrence doit être présente.
  - `xsd:field`. Au moins une occurrence doit être présente, mais il peut y en avoir plusieurs.

qui ont chacun un seul élément `xpath`.

Ainsi, les éléments définissant une contrainte d'existence s'écrivent :

```

1      <xsd:keyref name="nom_de_la_clef" refer="nom_clef_d'unicité">
2          <xsd:selector xpath="éléments sur lesquels porte l'existence" />
3          <xsd:field xpath="éléments ou attributs spécifiant l'existence" />
4          <xsd:field xpath="autres éléments/attributs ...." />
5          ...
6      </xsd:unique>

```

Figure VIII.9: Structure d'une clef d'existence.

L'attribut **name** donne son nom à la contrainte d'existence.

La valeur de l'attribut **refer** contient le nom (attribut **name**) d'une clef d'unicité associée.

L'élément **xsd:selector**, via un chemin XPath spécifié dans son attribut **path**, spécifie sur quels éléments s'applique la contrainte d'existence. Ces éléments, de même type, peuvent être multiples : il peut s'agir d'un seul élément comme d'une liste d'élément.

L'élément **xsd:field** permet, via son chemin XPath spécifié dans son attribut **path**, de spécifier la valeur servant de clef, dont on doit vérifier l'existence par rapport à une clef d'unicité. Plusieurs champs **xsd:field** peuvent être présents pour forcer la vérification de l'existence de plusieurs valeurs en même temps.



La contrainte d'existence implique que pour chaque élément sélectionné (**selector**), il existe un élément sélectionné par la contrainte d'unicité qui a la même valeur.



Attention, la cardinalité des champs **field** doit être la même entre la clef d'existence et la clef unique à laquelle elle se réfère !



Attention, les clefs uniques générées appartiennent à l'espace de nom dans lequel elles ont été définies. Si, dans le schéma XML dans lequel elles sont définies les éléments sont préfixés, alors la clef unique à laquelle la clef d'existence se réfère (avec **refer**) doit être préfixée !



Attention, la contrainte d'unicité à laquelle se réfère la contrainte d'existence doit être contenue dans le même élément ou dans un de ses descendants.

## 2.c Un exemple complet

Reprenons le schéma XML spécifiant les opérations sur les matrices. Nous avons vu qu'une opération possédait 2 attributs **left** et **right** permettant de se référer au nom des matrices. En réalité, dans ce schéma montré figure [VIII.7](#), rien ne garanti qu'une matrice dont le nom est donné dans les attributs **left** et **right** d'une opération existe réellement dans la liste des matrices instanciées ! Il faut donc ajouter une contrainte d'existence liant les valeurs de chacun des attributs **left** et **right** à l'existence des matrices, donc à leur nom donné par l'attribut **matName** présent dans chaque matrice.

Considérons d'abord la partie de schéma XML donnée ci-dessous dans la figure [VIII.10](#). Supposons que nous disposions d'une clef d'unicité nommée **matUnique** qui garanti l'unicité des noms des matrices présentes dans la listes de matrices **mat:matrices**, alors nous pouvons nous servir de cette clef pour

vérifier l'existence du nom d'une matrice.

Il suffit alors de définir deux clefs d'existence que nous nommerons `leftMatExist` et `rightMatExist`. Chacune de ces clefs se réfère à la clef d'unicité `mat:matUnique` (attention au préfixe ! voir note ci-dessus) qui lui permet de connaître l'existence du nom d'une matrice. Leur sélecteur applique la contrainte d'existence aux opérations `mat:operation` sur les valeurs `@left` et `@right` respectivement.

Il n'est cependant pas nécessaire de définir une clef d'existence pour vérifier que le nom de la matrice résultat existe dans la liste des matrices ; celle-ci, en tant que résultat, n'existe pas nécessairement encore.

L'exemple complet montrant l'inclusion de ces contraintes dans le schéma XML est donné plus bas, figure VIII.11.



On remarque (1) que la contrainte d'unicité `matUnique` est définie dans `mat:matrices`, un élément descendant de `mat:matrixComputations` dans lequel se trouvent définies les contraintes d'existence `leftMatExist` et `rightMatExist` et (2) l'usage du préfixe `mat` pour se référer à la contrainte d'unicité `mat:matUnique` dans les contraintes d'existence.

```

1      <!-- Unicité des matrices -->
2      <xsd:key name="matUnique">
3          <xsd:selector xpath="mat:matrix"/>
4          <xsd:field xpath="@matName"/>
5      </xsd:key>

1      <!-- Existence des matrices gauche et droite appelées dans les opé
2          rations -->
3      <xsd:keyref name="leftMatExist" refer="mat:matUnique">
4          <xsd:selector xpath="mat:operations/mat:operation"/>
5          <xsd:field xpath="@left"/>
6      </xsd:keyref>
7      <xsd:keyref name="rightMatExist" refer="mat:matUnique">
8          <xsd:selector xpath="mat:operations/mat:operation"/>
9          <xsd:field xpath="@right"/>
10     </xsd:keyref>

```

Figure VIII.10: Définition de deux clefs d'existence se référant à la clef d'unicité `matUnique`.

```

1  <?xml version="1.0"?>
2  <xsd:schema version="1.0"
3      targetNamespace="https://www.timc.imag.fr/matrix"
4      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
5      xmlns:mat="https://www.timc.imag.fr/matrix"
6      elementFormDefault="qualified">
7      <xsd:include schemaLocation="./Matrix.xsd"/>
8
9      <!-- Elément racine 'matrixComputation' -->
10     <xsd:element name="matrixComputations">
11         <xsd:complexType>
12             <xsd:sequence>
13
14                 <!-- liste de matrices -->
15                 <xsd:element name="matrices" maxOccurs="unbounded">
16                     <xsd:complexType>
17                         <xsd:sequence>
18                             <xsd:element name="matrix" type="mat:Matrix"
19                                 minOccurs="1" maxOccurs="unbounded"/>

```

```

19         </xsd:sequence>
20     </xsd:complexType>
21     <!-- Unicité des matrices -->
22     <xsd:key name="matUnique">
23         <xsd:selector xpath="mat:matrix"/>
24         <xsd:field xpath="@matName"/>
25     </xsd:key>
26 </xsd:element>
27
28 <!-- liste d'operations -->
29 <xsd:element name="operations" maxOccurs="unbounded">
30     <!-- code inchangé (voir précédent Schema XML) -->
31 </xsd:element>
32
33 </xsd:sequence>
34 </xsd:complexType>
35
36 <!-- Existence des matrices gauche et droite appelées dans les opé
37     rations -->
38 <xsd:keyref name="leftMatExist" refer="mat:matUnique">
39     <xsd:selector xpath="mat:operations/mat:operation"/>
40     <xsd:field xpath="@left"/>
41 </xsd:keyref>
42 <xsd:keyref name="rightMatExist" refer="mat:matUnique">
43     <xsd:selector xpath="mat:operations/mat:operation"/>
44     <xsd:field xpath="@right"/>
45 </xsd:keyref>
46 </xsd:element>
47
48 <!-- Définition du type complexe 'Operation' (une opération matricielle) --
49     >
50 <!-- code inchangé (voir précédent Schema XML) -->
51
52 <!-- Définition du type complexe 'MatOpType' (types possibles pour une opé
53     ration matricielle) -->
54 <!-- code inchangé (voir précédent Schema XML) -->
55 </xsd:schema>

```

Figure VIII.11: Schema XML (MatrixOperation.xsd) *modifié* modélisant une liste d'opérations matricielles entre deux matrices et intégrant des clefs d'existence.



## *XPath : Naviguer dans un document XML*

Modéliser en XMLSchema et stocker des données au format XML c'est bien ; pouvoir les manipuler, c'est mieux ! Dans les chapitres qui suivront celui-ci, nous apprendrons à écrire des feuilles de transformation XSLT et à lire et écrire des documents XML. Dans les deux cas, il est nécessaire de naviguer dans le document, c'est-à-dire de sélectionner des éléments ou attributs particuliers, ou même des listes d'éléments, de passer d'un élément à un autre, ... de la même manière que l'on navigue dans l'arborescence des fichiers d'un disque dur.

Le langage XPath offre un moyen d'identifier un ensemble de nœuds dans un document XML. Toutes les applications (écrites en Java, Javascript, C++, C#...) ayant besoin de récupérer un fragment de document XML peuvent utiliser ce langage. Nous verrons dans des chapitres ultérieurs de nombreux exemples d'usage avec des technologies utilisant XPath (XSLT, DOM) ; d'autres existent comme XQuery, etc.

XPath est fondé sur la représentation d'un document XML sous forme d'arbre, comme illustré dans les exemples du chapitre précédent: [II.7](#) et [III.2](#).

Une expression XPath (encore appelée chemin XPath ou encore *location path*) est **une expression non XML**, avec une syntaxe compacte. Elle s'évalue sur un nœud courant dans un contexte donné.

Il s'agit d'un ensemble d'étapes de navigation (*location steps*) séparées par des '/'. Par exemple, l'expression `/bagagesPassagers/bagage/ref` appliquée sur le document XML des figures [II.7](#) et [III.2](#) donne le résultat expliqué dans le tableau suivant<sup>1</sup>

---

<sup>1</sup>voir également l'animation proposée sur la plateforme de cours.

<div style="border: 1px solid red; padding: 2px; display: inline-block;"> <code>/bagagesPassagers/bagage/ref</code> </div> <div style="display: flex; justify-content: space-around; width: 100%; margin-top: 5px;"> <span>1</span><span>2</span><span>3</span><span>4</span> </div>	
1- <code>/</code> sélectionne l'arbre entier (il s'agit d'un chemin absolu)	2- <code>/bagagesPassagers</code> sélectionne tout l'élément racine <code>bagagesPassagers</code> (c'est-à-dire l'élément racine et ses sous-éléments).
3- <code>bagage</code> sélectionne les nœuds <code>bagage</code> , fils direct de la racine <code>bagagesPassagers</code> .	4- A partir des nœuds <code>bagage</code> sélectionnés, <code>ref</code> sélectionne les nœuds fils <code>ref</code> .

Une expression XPath désigne un ou plusieurs *chemins* dans l'arbre à partir du nœud courant. Elle a pour résultat soit un ensemble de nœuds (*node-set*), soit une valeur (numérique, booléenne ou alphanumérique).

*Remarque:* comme vous le verrez en TD et TP, déboguer une application dépendant d'une expression xpath peut s'avérer compliqué. Fort heureusement, il existe de nombreux sites sur Internet permettant de tester des expressions xpath sur des documents XML valides, par exemple <http://www.utilities-online.info/xpath>.

## 1 Node-set

Un nœud est un élément<sup>2</sup> + ses attributs.

A partir d'un nœud courant, une étape de chemin XPath (*location step*) renvoie **un ensemble de nœuds**, c'est-à-dire une *node-set*. Chaque nœud de ce *node-set* devient à son tour le nœud courant pour l'évaluation du *location-step* suivant.

On commence par la racine dénotée par `/`.

## 2 Les axes de recherche

Un axe est un ensemble de nœuds relativement à un autre nœud. On distingue les *axes verticaux* et les *axes horizontaux*.

### 2.a Axes verticaux

Parmi les axes verticaux, les axes *avant* permettent de descendre dans l'arbre:

<sup>2</sup>Rappel: un élément est soit une balise ouvrante + un contenu + une balise fermante, soit une balise ouvrante/fermante.

**self** désigne le nœud courant. Par exemple `/bagagesPassagers/bagage/type/self::type[format="sac à dos"]` désigne le seul nœud `type` dont l'attribut `format` a pour valeur `sac à dos`.

**child** désigne les fils du nœud courant (c'est l'axe par défaut). L'expression `bagage/child::type` est équivalente à l'expression `bagage/type`.

**descendant** désigne les descendants (les fils, et les fils des fils et les fils des fils des fils, etc.) du nœud courant. par exemple l'expression `/bagagesPassagers/descendant::ref` désigne les 2 éléments `ref` de l'arbre.

**descendant-or-self** désigne les descendants du nœud courant ainsi que le nœud courant lui-même.

Les axes *arrière* permettent de remonter dans l'arbre:

**parent** désigne le nœud parent du nœud courant

**ancestor** désigne les ascendants du nœud courant. Par exemple si le nœud courant est le premier élément `ref`, l'élément `ancestor::bagagesPassagers/compagnie` fait référence à l'élément `compagnie`.

**ancestor-or-self** désigne les ascendants du nœud courant ainsi que le nœud courant lui-même

## 2.b Axes horizontaux

**following-sibling** désigne les nœuds frères (i.e. issus du même parent) placés **après** le nœud courant. Par exemple, `/bagagesPassagers/date/following-sibling::node` désigne les 2 éléments `bagage`

**preceding-sibling** désigne les nœuds frères placés **avant** le nœud courant.

Par exemple, `/bagagesPassagers/date/preceding-sibling::node` désigne l'élément `compagnie`

## 2.c Axe pour les attributs

**attribute** désigne les attributs du nœud courant

## 2.d Abréviation des Axes

Afin d'éviter une trop grande lourdeur, les simplifications suivantes sont autorisées:

abréviation	Axe
(rien)	<code>child::</code>
@	<code>attribute::</code>
.	<code>self::node()</code>
//	<code>desendant-or-self::node()/</code>
//X	<code>desendant-or-self::node()/child::X</code>
..	<code>parent::node()</code>
../X	<code>parent::node/child::X</code>

## 2.e Remarque sur les Chemins

On peut faire un parallèle entre les chemins XPath et les chemins sur les disques durs des ordinateurs. Par exemple, sur l'arborescence d'un ordinateur, `/` désigne la racine du disque. De même, en XPath, `/bagagesPassagers` désigne la racine de l'arbre.

De même, sur un disque dur, `/cours/XML` désigne le fichier XML dans le répertoire `cours` à la racine du disque ; le chemin `/bagagesPassagers/date` désigne l'élément `date`, fils de l'élément racine `bagagesPassagers`.

Dans les 2 cas, si le chemin commence par `/` il s'agit d'un chemin *absolu* (qui part de la racine), sinon, il s'agit d'un chemin *relatif* (qui part de l'endroit/du nœud courant).

Attention cependant, cette comparaison a des limites, car dans le cas d'un chemin XPath, un chemin absolu (ou relatif) peut sélectionner plusieurs nœuds !

## 3 Sélectionner des parties de l'arbre

### 3.a Les filtres

**:: (nom)** le filtre *nom* sélectionne les nœuds de l'axe qui portent ce nom.

Par exemple, `/bagagesPassagers/child::bagage` sélectionne les 2 nœuds `bagage` fils de `bagagesPassagers`.

**\*** le filtre `*` sélectionne les nœuds de l'axe qui ont un nom.

Par exemple, `/bagagesPassagers/*/type/attribute::*` sélectionne les 4 attributs des 2 éléments `type` de l'arbre.

**text()** le filtre `text()` sélectionne les nœuds de type texte.

Par exemple, `/bagagesPassagers/date/text()` renvoie "2012-01-06".

**comment()** le filtre `comment()` renvoie les nœuds de type commentaire de l'axe.

Par exemple, `/comment()` renvoie "Bagagerie".

### 3.b Les critères

On peut ajouter des critères de sélection en ajoutant des crochets aux *location-steps*. `Chemin[critère]` ne sélectionne que les nœuds du `Chemin` qui correspondent à la sélection par critère. On peut lire ce chemin xpath ainsi : "renvoie les noeuds du Chemin *tel que ce critère est vrai*".

#### Critères d'attributs

On peut sélectionner tous les nœuds de l'arbre ayant/n'ayant pas un certain attribut:

`//*[@format]` sélectionne tous le nœuds qui ont un attribut `format`.

`//*[not(@format)]` sélectionne tous les nœuds de l'arbre qui n'ont pas d'attribut `format`.

`//bagage/*[@*]` sélectionne tous les nœuds fils de `bagage` qui ont un attribut (quel qu'il soit)

`//bagage/*[not(@*)]` sélectionne tous les nœuds fils de `bagage` qui n'ont aucun attribut

`//type[@couleur='noir']` sélectionne tous les nœuds `type` qui ont un attribut `couleur` de valeur `noir`

#### Critères d'ordre

`/bagagesPassagers/bagage[1]` sélectionne le premier nœud `bagage`.

`//bagage[2]` sélectionne le deuxième nœud `bagage`.

`/bagagesPassagers/bagage[last()]` sélectionne le dernier nœud `bagage`.

### 3.c Expressions arithmétiques

On peut également utiliser des expressions arithmétiques pour les sélections:

Opérations	
+	addition
-	soustraction
*	multiplication
div	division (attention, ce n'est pas /)
mod	modulo (reste de la division entière)
=	égalité
!=	inégalité
Opérateurs Booléens	
and	<i>et puis</i>
or	<i>ou bien</i>
not()	négation
Opérateurs de comparaison	
<	strictement inférieur à
<=	inférieur ou égal à
>	strictement supérieur à
>=	supérieur ou égal à

En XPath, une expression est vraie s'il a au moins un nœud sélectionné. Par exemple, l'expression `//type[@couleur='noir']` sélectionne tous les nœuds `type` dont l'attribut `couleur` a la valeur `noir`. L'expression `//type[@couleur!='noir']` sélectionne tous les nœuds `type` dont l'attribut `couleur` n'a pas la valeur `noir`. Ces deux expressions sont *vraies*. En effet, il existe un nœud dont l'attribut `couleur` a la valeur `noir` et un nœud dont l'attribut `couleur` n'a pas la valeur `noir`.

### 3.d Fonctions

#### Fonctions sur les chaînes de caractères

<code>string-length(s: string) : int</code>	renvoie la longueur (en nombre de caractères) de la chaîne de caractères <code>s</code> Exemple <code>string-length('bonjour')</code> renvoie 7.
<code>concat(s1: string, s2: string, ...): string</code>	renvoie la chaîne composée de <code>s1</code> , <code>s2</code> , ... à la suite Exemple: <code>concat('bonjour', 'toi')</code> renvoie la chaîne <code>'bonjourtoi'</code> .
<code>contains(s1: string, s2: string): boolean</code>	renvoie vrai si la chaîne <code>s1</code> contient la chaîne <code>s2</code> Exemple: <code>contains('bonjour', 'njo')</code> renvoie <code>true</code> .
<code>starts-with(s1: string, s2: string): boolean</code>	renvoie vrai si la chaîne <code>s1</code> commence par la chaîne <code>s2</code> Exemple: <code>starts-with('bonjour', 'njo')</code> renvoie <code>false</code> .
<code>substring(s: string, d:int, [, f:int]): string</code>	renvoie la sous-chaîne de <code>s</code> à partir de l'indice <code>d</code> (éventuellement jusqu'à l'indice <code>f</code> ) Exemple: <code>substring('bonjour', 3)</code> renvoie <code>'jour'</code> .

#### Fonctions sur les node-set

<code>count([node-set]) : int</code>	nombre de nœuds [contenu dans le node-set].
<code>last(): int</code>	taille du node-set courant.
<code>name([node-set]): string</code>	Nom du nœud courant [1er du node-set].
<code>position(): int</code>	Position du nœud courant.

## 4 Ecrire correctement du XPath

Vous connaissez la syntaxe XPath ? C'est bien ! Vous découvrirez vite cependant que l'écriture de chemins complexes en XPath peut vite devenir un casse-tête sans fin.

Pour éviter au maximum de se retrouver dans une situation inextricable, **il FAUT que vous travailliez méthodiquement**, à savoir :

### 4.a Formuler correctement le problème

Prenons tout de suite un exemple avec le document XML suivant qu stocke les données de la bagagerie d'un aéroport pour les bagages hors format :

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <bagagerie
3   xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
4   xmlns='http://www.timc.fr/nicolas.glade/bagages'
5   xsi:schemaLocation='http://www.timc.fr/nicolas.glade/bagages bagagerieHF.
   xsd'>
6   <passagers>
7     <passager nom="Gator" prenom="Ali" id="AG18300427"></passager>
8     <passager nom="Nancière" prenom="Sophie" id="SF23890311"></passager>
9     <passager nom="Kass" prenom="Oscar" id="OK17401023"></passager>
10  </passagers>
11  <bagages>
12    <bagage idPassager="OK17401023">
13      <ref>AF677793</ref>
14      <type format="autre" couleur="blanc"/>
15      <poids>12.1</poids>
16      <description>skis</description>
17    </bagage>
18    <bagage idPassager="OK17401023">
19      <ref>AF677793</ref>
20      <type format="autre" couleur="bleu"/>
21      <poids>8.7</poids>
22      <description>skis</description>
23    </bagage>
24    <bagage idPassager="AG18300427">
25      <ref>AF48717</ref>
26      <type format="autre" couleur="marron"/>
27      <poids>6.4</poids>
28      <description>tam-tam</description>
29    </bagage>
30    <bagage idPassager="SF23890311">
31      <ref>AF67840</ref>
32      <type format="autre" couleur="blanc"/>
33      <poids>10.4</poids>
34      <description>skis</description>
35    </bagage>
36  </bagages>
37 </bagagerie>

```

et le schéma associé :

```

1 <?xml version="1.0"?>
2 <schema version="1.0"
3   xmlns="http://www.w3.org/2001/XMLSchema"
4   xmlns:bg="http://www.timc.fr/nicolas.glade/bagages"
5   targetNamespace="http://www.timc.fr/nicolas.glade/bagages"
6   elementFormDefault="qualified">
7   <include schemaLocation="bagages.xsd"/>
8
9   <element name="bagagerie" type="bg:Bagagerie"/>
10
11  <complexType name="Bagagerie">

```

```

12      <sequence>
13          <element name="passagers">
14              <complexType>
15                  <sequence>
16                      <element name="passager" type="bg:Passager" maxOccurs="
17                          unbounded"/>
18                  </sequence>
19              </complexType>
20          </element>
21          <element name="bagages">
22              <complexType>
23                  <sequence>
24                      <element name="bagage" type="bg:BagageHF" maxOccurs="
25                          unbounded"/>
26                  </sequence>
27              </complexType>
28          </element>
29      </sequence>
30  </complexType>
31  <complexType name="Passager">
32      <attribute name="nom" type="string" use="required"/>
33      <attribute name="prenom" type="string" use="required"/>
34      <attribute name="id" type="bg:IDPassager" use="required"/>
35  </complexType>
36
37  <complexType name="BagageHF">
38      <complexContent>
39          <extension base="bg:Bagage">
40              <sequence>
41                  <element name="description" type="string"/>
42              </sequence>
43              <attribute name="idPassager" type="bg:IDPassager"/>
44          </extension>
45      </complexContent>
46  </complexType>
47
48  <simpleType name="IDPassager">
49      <restriction base="string">
50          <pattern value="[A-Z]{2}[0-9]{8}"/>
51      </restriction>
52  </simpleType>
53
54 </schema>

```

Supposons que l'on souhaite connaître le nom de famille de tous les passagers dont les bagages hors format sont des skis.

D'abord, on commence par reformuler cette phrase sous cette forme : [Ce qu'on veut] [tel que] [la ou les conditions]. Cela donne dans notre cas : [Le nom de famille des passagers] [tel que] [les bagages sont des skis].

Mais si l'on regarde mieux, on voit que ce que l'on doit faire, c'est relier l'identifiant des bagages qui sont des skis, à l'identifiant passager. Donc on peut reformuler la condition ainsi : [les identifiants des bagages qui sont des skis]  $\Rightarrow$  [les identifiants des bagages] [tel que] [ce sont des skis]

On voit donc qu'on va devoir mettre en relation un 1er chemin (les noms des passagers) avec un deuxième chemin (les identifiants des bagages)

#### 4.b Exprimer ce que l'on veut et ce que sont les conditions

Que veut-on finalement ?  $\Rightarrow$  les noms des passagers ... donc on commence par écrire le chemin XPath qui donne les noms de tous les passagers : `//passager/@nom`. Comme attendu, on obtient la liste de tous les noms du document : { Gator, Nancière, Kass}

Pour les bagages, que veut-on ?  $\Rightarrow$  **les identifiants des bagages** ... donc on écrit cela sous la forme d'un autre chemin XPath : `//bagage/@idPassager` . Cette fois, on obtient la liste de tous les identifiants passagers dans les bagages : { OK17401023, OK17401023, AG18300427, SF23890311}. Vous remarquerez que l'on peut obtenir plusieurs fois le même identifiant.

Ensuite on se pose la question de comment réunir les deux. Ce qui lie les bagages aux passagers, ce sont les identifiants des passagers. Mais avant de réunir les deux chemins, on va tester des choses simples, par exemple des identifiants fixes. Reprenons les noms des passagers et ajoutons une sélection liée à l'identifiant : je veux les noms des passagers dont l'identifiant est X : `//passager[@id='AG18300427']/@nom` ce qui se lit *les noms des passagers tel que l'identifiant passager est AG18300427*. On obtient la liste contenant 1 seul nom (mais il s'agit bien d'une liste) : {Gator}

De même, pour les bagages, récupérons les identifiants des bagages qui sont des skis : `//bagage[contains(./description/text(),'skis')]/@idPassager` . On obtient la liste d'identifiant suivante : { OK17401023, OK17401023, SF23890311 }

Ensuite seulement, on peut relier les 2 chemins :

`//passager[@id=//bagage[contains(./description/text(),'skis')]/@idPassager]/@nom`, ce qui renvoie bien la liste de noms attendus { Nancière, Kass} qui sont les deux seules personnes à partir au ski en avion.

#### 4.c Procéder par petites étapes qu'on teste

De la même manière qu'on n'écrit pas un programme entier avant de le tester, on n'écrit pas une ligne complexe de XPath sans procéder par petites étapes. Chacune des étapes écrite au dessus doit être testée individuellement soit avec un outil en ligne (par exemple <http://xpather.com/> qui marche bien) soit avec un programme appelant en Java ou autre.

Autrement dit, vous devez tester dans l'ordre les expressions suivantes :

- `//passager/@nom`
- `//bagage/@idPassager`
- `//passager[@id='AG18300427']/@nom`
- `//bagage[contains(./description/text(),'skis')]/@idPassager`
- `//passager[@id=//bagage[contains(./description/text(),'skis')]/@idPassager]/@nom`



# XSLT: transformations en XML

Nous avons vu que les langages XML et XMLSchema permettent de structurer et d'organiser des données. XSLT (*eXtensible Stylesheet Language Transforms*) va nous permettre de définir des transformations pour *modifier* et/ou **présenter** les données des documents XML (cf Figure X.1). Par exemple, on peut

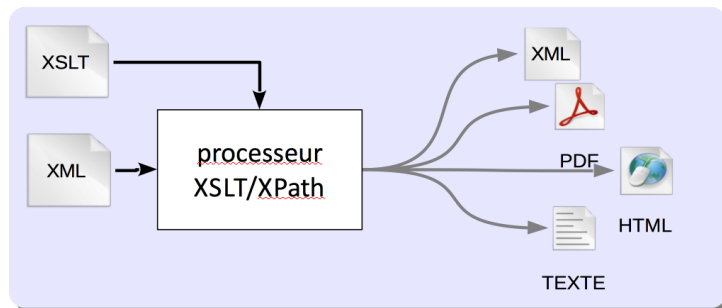


Figure X.1: XSLT permet de transformer un document XML en un document mis en page.

imaginer, comme illustré figure X.2 une transformation XSLT qui transforme un XML en un livre au format pdf.

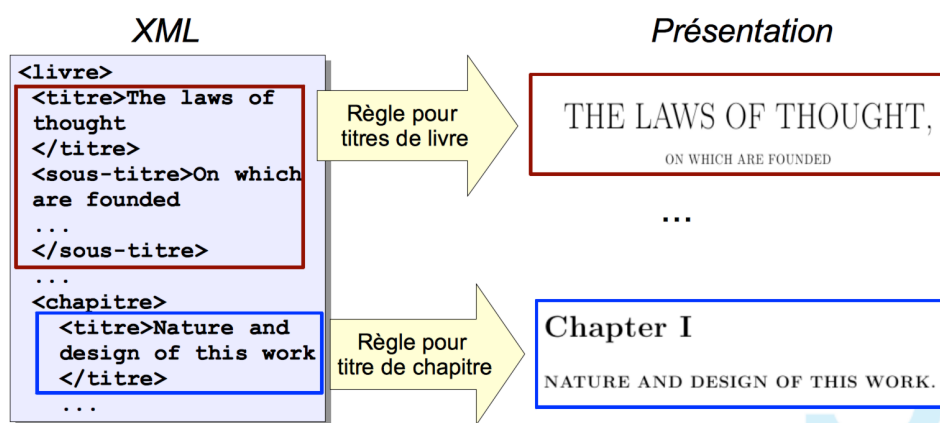


Figure X.2: Exemple de transformation d'un document XML en un livre au format pdf.

XSLT fait partie de l'ensemble XSL (*eXtensible Stylesheet Language*). XSL est composé d'XPath, XSLT et XSL-FO (*Formating Objects* XSL-FO donne la définition de la sémantique de présentation utilisée principalement dans les documents pdf).

XSLT ne permet pas vraiment d'écrire des algorithmes, mais plutôt de spécifier des transformations à appliquer. Il est basé sur un paradigme récursif.



XSLT est beaucoup plus complet que ce que nous allons voir ici.

## 1 Principes de fonctionnement

Pour transformer un document XML (i.e. une instance XML) en un autre document, XSLT le *transforme* en arbre d'instance (le même arbre que nous avons vu au chapitre III). Il commence par traiter la racine du document puis applique des transformations récursives nœud par nœud. Chaque transformation (aussi appelée *template*) s'appuie sur 3 mécanismes pour écrire le document de sortie:

- *push* permet au processeur d'écrire directement dans le fichier de sortie
- *pull* permet au processeur d'utiliser le fichier d'entrée (XML) pour écrire dans le fichier de sortie
- *navigation* permet au processeur de naviguer dans l'arbre du document d'entrée (XML)

### 1.a Anatomie d'un document XSLT

XSLT est un langage XML dont la racine est l'élément **stylesheet** (qui contient autant d'attributs **xmlns** que de vocabulaires utilisés, tous préfixés) et le premier sous-élément est **output** (qui permet de préciser le type du fichier de sortie).

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <xsl:stylesheet version="1.0" ...
3     xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
4     xmlns:ns1="http://www....namespace1...."
5     xmlns:ns2="http://www....namespace2...."
6     .....>
7     <xsl:output method="xml"/> <!-- ou html, ou text, ou autre... -->
8     <!-- Modules de transformation -->
9 </xsl:stylesheet>

```

Dans une feuille XSLT, après la racine, et au même niveaux les uns des autres (i.e. fils directs de la racine) se trouvent les modules. Un module, aussi appelé *template* (modèle) est la description de la transformation à appliquer à un nœud donné de l'entrée XML. Le nombre de modules/*templates* n'est pas limité.

La définition d'un *template* à appliquer sur des cibles d'un document XML s'écrit de la manière suivante:

```

1 <xsl:template match="cible-xpath">
2     ...
3     <!-- code xsl -->
4     ...
5 </xsl:template>

```

Le code défini à l'intérieur du *template* va être appliqué à tous les nœuds du document XML d'entrée qui seront sélectionnés par le chemin *cible-xpath*.

Le premier *template* à être appliqué sur un document doit s'appliquer à sa racine. C'est pourquoi, pour démarrer la transformation, le processeur applique le *template* suivant:

```

1 <xsl:template match="/">
2     <!-- point d'entrée de la transformation -->
3     ...
4 </xsl:template>

```

ici, la *cible-xpath* est / qui désigne la racine, donc l'ensemble du document.

## 1.b Mécanisme Général

Le document XML d'entrée est traité récursivement à partir de la racine (parcours en profondeur d'abord, comme le sens de lecture du texte). On commence par sélectionner la racine (/), puis le processeur XSLT recherche une règle de transformation pour chaque nœud sélectionné. Lorsqu'un nœud est examiné, le processeur XSLT

- recherche le *template* correspondant au nœud examiné (autrement dit, la règle de transformation qui *match* le nom du nœud).
- si un *template* correspond
  - le nœud examiné devient le nœud contexte (i.e. le nœud courant)
  - le *template* sélectionné est "exécuté" (utilisé comme modèle de transformation pour le nœud contexte).
- sinon XSLT applique le traitement par défaut, c'est-à-dire recopie la donnée texte d'entrée (i.e. le texte contenu dans les éléments) vers la sortie.

Lorsque le *template* est terminé pour le nœud courant, le processeur passe au nœud suivant et ainsi de suite jusqu'à ce que le document XML ait été complètement exploré.

Note : Si aucun *template* n'est défini dans la feuille de style, la transformation XSLT va enlever toutes les balises d'un document XML et recopier le texte dans le document de sortie.

Dans les *templates*, on peut activer les 3 mécanismes *push*, *pull* et *navigation*.

## 1.c Exemple

Dans la suite du chapitre, on considère l'exemple XML simpliste présenté figure X.3.

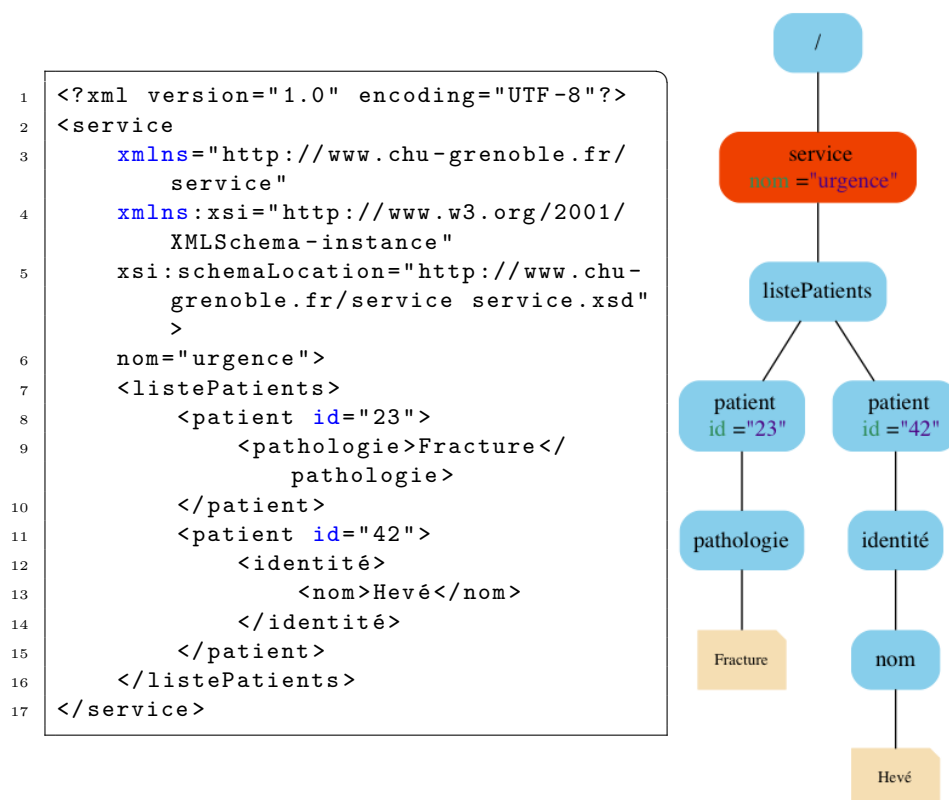


Figure X.3: Exemple de document XML simpliste et arbre d'instance correspondant.

## 2 Mécanisme *Push*

Le mécanisme *push* sert à écrire dans le document de sortie le texte écrit dans le template. Cela est utile pour écrire du code libre comme du xml, du xhtml, etc.

Par exemple, si l'on applique la transformation X.4 sur l'exemple X.3, on obtient le document X.5.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <xsl:stylesheet
3   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
4   xmlns:chu="http://www.chu-grenoble.fr/service"
5   version="1.0">
6
7   <xsl:output method="html"/>
8
9   <xsl:template match="/">
10     <html>
11       <head>
12         <title>Test</title>
13       </head>
14       <body>
15         Petit texte...
16       </body>
17     </html>
18   </xsl:template>
19
20 </xsl:stylesheet>

```

Figure X.4: Exemple d'un *template* XSLT qui utilise le mécanisme *push* (en rouge).

```

1 <html>
2 <head>
3 <META http-equiv="Content-Type" content="text/html; charset=UTF-8">
4 <title>Test</title>
5 </head>
6 <body>
7   Petit texte...
8 </body>
9 </html>

```

Figure X.5: Résultat obtenu en appliquant le *template* X.4 sur l'exemple X.3.

Le *template* de la feuille de style X.4 (lignes 4 à 13) sélectionne la racine (via /) puis écrit dans le document de sortie (mécanisme *push*) le texte html contenu dans la feuille de style. On peut donc mettre en forme et écrire directement dans le document de sortie grâce au mécanisme *push*.

Le mécanisme *push* s'applique également lorsque l'on appelle les sous-*templates*. Par exemple, si l'on applique la transformation X.6 sur l'exemple X.3, on obtient le document X.7.

On peut noter dans le résultat X.7 que la feuille X.6 contient, en plus des instructions *push*, de la navigation. En effet, le *template* `<xsl:template match="patient">` lignes 16 à 18 sélectionne les nœuds *patient*.

Le *template* `<xsl:template match="/">` lignes 4 à 14 écrit le texte dans la sortie, puis appelle les *templates* sur les sous-nœuds avec l'instruction `<xsl:apply-templates/>` ligne 11. Le *template* `<xsl:template match="patient">` lignes 16 à 18 s'applique sur chaque nœud *patient* et écrit Trouvé un patient ! à chaque fois qu'il est appliqué.

**Remarque.** Attention, il est très important :

- d'inclure tous les espaces de nom (les vocabulaires) utilisés dans la feuille XSLT.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <xsl:stylesheet
3   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
4   xmlns:chu="http://www.chu-grenoble.fr/service"
5   version="1.0">
6
7   <xsl:output method="html"/>
8
9   <xsl:template match="/">
10     <html>
11       <head>
12         <title>Test</title>
13       </head>
14       <body>
15         Petit texte...
16         <xsl:apply-templates/>
17       </body>
18     </html>
19   </xsl:template>
20
21   <xsl:template match="patient">
22     Trouvé un patient !
23   </xsl:template>
24
25 </xsl:stylesheet>

```

Figure X.6: Exemple d'un *template* XSLT appelant un sous-*template*.

```

1 <html>
2 <head>
3 <META http-equiv="Content-Type" content="text/html; charset=UTF-8">
4 <title>Test</title>
5 </head>
6 <body>
7     Petit texte...
8
9
10
11     Trouvé un patient !
12
13
14     Trouvé un patient !
15
16
17 </body>
18 </html>

```

Figure X.7: Résultat obtenu en appliquant le *template* X.6 sur l'exemple X.3.

- de préfixer TOUS les espaces de nom (pas de vocabulaire par défaut en XSLT)

### 3 Mécanisme *Pull*

Le mécanisme *pull* récupère (pioche) des données dans le fichier XML source en utilisant des expressions XPath. Il est utilisé pour transformer/analyser/présenter les données et est inclus dans les *templates*.

La récupération des données se fait principalement à l'aide de l'instruction `xsl:value-of`. L'instruction `<xsl:value-of select="expression xpath"/>` sélectionne la valeur (ensemble de nœuds ou bien valeur textuelle/chiffre/booléen, etc. de l'expression XPath. Cette expression peut être soit absolue (partir de

la racine de l'arbre), soit relative et s'appuyer sur le nœud courant, c'est-à-dire le nœud qui est en train d'être traité par le *template*.

Par exemple, si l'on applique la transformation X.8 sur l'exemple X.3, on obtient le résultat X.9.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <xsl:stylesheet
3   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
4   xmlns:chu="http://www.chu-grenoble.fr/service"
5   version="1.0">
6
7   <xsl:output method="html"/>
8
9   <xsl:template match="/">
10    <html>
11      <head>
12        <title>Test</title>
13      </head>
14      <body>
15        <h1>Service <xsl:value-of select="service/@nom"/></h1>
16        Il y a <xsl:value-of select="count(//patient)"/> patients.
17      </body>
18    </html>
19  </xsl:template>
20
21 </xsl:stylesheet>

```

Figure X.8: Exemple de transformation avec le mécanisme *pull*.

```

1 <html>
2 <head>
3 <META http-equiv="Content-Type"
4   content="text/html; charset=UTF
5   -8">
6 <title>Test</title>
7 </head>
8 <body>
9 <h1>Service urgence</h1>
10  Il y a 2 patients.
11 </body>
12 </html>

```

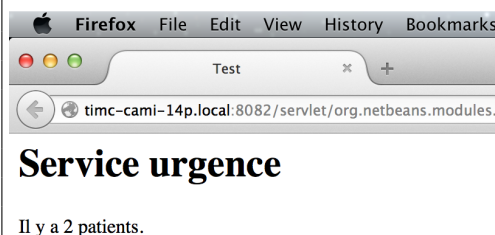


Figure X.9: Résultat obtenu en appliquant le *template* X.8 sur l'exemple X.3.

Dans ce document X.8, la cible de la *template* lignes 4 à 14 cible la racine. Le nœud courant concerne donc tout le document. L'expression *pull* ligne 10 `<xsl:value-of select="service/@nom"/>` utilise un chemin XPath relatif à la racine qui désigne l'attribut `nom` de l'élément `service`. La valeur écrite dans le document résultat est donc la valeur de cet attribut, soit **urgence**.

L'expression *pull* ligne 11 `<xsl:value-of select="count(//patient)"/>` utilise la méthode XPath *count* qui renvoie le nombre de nœuds sélectionné dans une expression XPath. L'expression XPath en question `//patient` est une expression absolue qui sélectionne tous les éléments `patient` du nœud courant. En l'occurrence, il y en a 2.

## 4 Mécanisme de navigation

Le langage XSLT parcourt le document selon l'arbre d'instance, en profondeur d'abord (dans l'ordre des balises du texte). On peut cependant guider cette navigation pour être plus efficace en utilisant les instructions

- `xsl:apply-templates`
- `xsl:call-template`

`xsl:apply-templates` permet d'appliquer récursivement des *templates* sur des sous-nœuds choisis. Par exemple, la feuille de transformation [X.10](#) saute de la racine aux nœuds `patient`<sup>1</sup>.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <xsl:stylesheet
3   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
4   xmlns:chu="http://www.chu-grenoble.fr/service"
5   version="1.0">
6
7   <xsl:output method="html"/>
8
9   <xsl:template match="/">
10     <html>
11       <head>
12         <title>Test</title>
13       </head>
14       <body>
15         <h1>Service <xsl:value-of select="service/@nom"/></h1>
16         Il y a <xsl:value-of select="count(//patient)"/> patients.
17         <xsl:apply-templates select="service/listePatients/patient"/>
18       </body>
19     </html>
20   </xsl:template>
21
22   <xsl:template match="patient">
23     <p>patient id=<xsl:value-of select="@id"/></p>
24   </xsl:template>
25
26 </xsl:stylesheet>

```

Figure X.10: Sélection des nœuds `patient` et appel de la *template* correspondante.

```

1 <html>
2 <head>
3 <META http-equiv="Content-Type"
4   content="text/html; charset=UTF
5   -8">
6 <title>Test</title>
7 </head>
8 <body>
9 <h1>Service urgence</h1>
10   Il y a 2 patients.
11   <p>patient id=23</p>
12   <p>patient id=42</p>
13 </body>
14 </html>

```

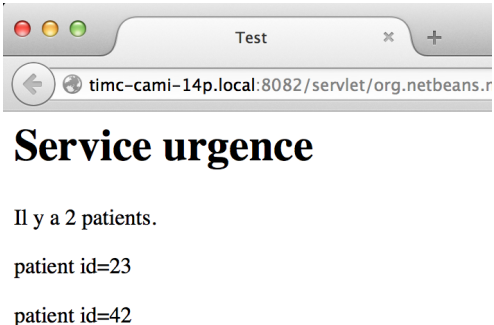


Figure X.11: Résultat obtenu en appliquant la *template* [X.10](#) sur l'exemple [X.3](#).

On peut également faire référence à un *template* sans passer par XPath, en l'appelant par son *nom*. On utilise alors l'instruction `<xsl:call-template name="nomDeLaTemplate"/>`. On déclarera alors la *template* grâce à l'expression `<xsl:template name="nomDuTemplate"/>`.

<sup>1</sup>Le résultat de cette transformation est présenté Figure [X.11](#)

#### 4.a Application/Exécution des *templates*

Par défaut, les *templates* ne sont pas exécutés (appliqués). Le *template* principal est généralement `<xsl:template match="/">` qui est le seul qui est exécuté automatiquement. En effet, lors de la transformation d'un document, le processeur XSLT commence implicitement par `<xsl:apply-templates select="/">`. Ensuite, tout dépend du code du *template* principal et du contenu du document source.

L'application/exécution d'un *template* dépend donc

- de la correspondance `select/match`
- des instructions d'appel direct

#### 4.b Paramètres de *template*

Un *template* peut recevoir des paramètres

```
1 <xsl:template match="...">
2   <xsl:param name="p1"/>
3   <xsl:param name="p2"/>
4   ...
5 </xsl:template>
```

Les paramètres sont passés dans le *template* appelant en utilisant l'élément `xsl:with-param`.

```
1 <xsl:apply-templates select="...">
2   <xsl:with-param name="p2" select="val/fonc"/>
3   <xsl:with-param name="p1" select="expression xpath"/>
4   ...
5 </xsl:apply-templates>
```

Le type des variables peut être un *Node-Set* (ensemble de nœuds) ou bien un type prédéfini (*string*, *number*, *boolean*).

#### 4.c Note sur le format de sortie

Sur la figure X.7, on peut remarquer de nombreux espaces dans le texte de sortie du XSLT figure X.6. Certains de ces espaces viennent de la mise en page du document XSLT, notamment sur le mécanisme *Push*, mais également du mécanisme de *Navigation*.

Par exemple, si l'on applique la même transformation X.6 sur le document X.12 dans lequel on a ajouté du texte pour les retours à la ligne, on obtient le document X.13.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <service
3   xmlns="http://www.chu-grenoble.fr/service"
4   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5   xsi:schemaLocation="http://www.chu-grenoble.fr/service service.xsd"
6   nom="urgence">retour à la ligne numéro 1
7   <listePatients>retour à la ligne numéro 2
8     <patient id="23">retour à la ligne numéro 3
9       <pathologie>Fracture</pathologie>retour à la ligne numéro 4
10    </patient>retour à la ligne numéro 5
11    <patient id="42">retour à la ligne numéro 6
12      <identité>retour à la ligne numéro 7
13        <nom>Hevé</nom>retour à la ligne numéro 8
14      </identité>retour à la ligne numéro 9
15    </patient>retour à la ligne numéro 10
16  </listePatients>retour à la ligne numéro 11
17 </service>
```

Figure X.12: Exemple X.3 sur lequel on a ajouté du texte pour les retours à la ligne.



```

1 <html>
2 <head>
3 <META http-equiv="Content-Type" content="text/html; charset=UTF-8">
4 <title>Test</title>
5 </head>
6 <body>
7     Petit texte...
8     retour à la ligne numéro 1
9     retour à la ligne numéro 2
10
11     Trouvé un patient !
12     retour à la ligne numéro 5
13
14     Trouvé un patient !
15     retour à la ligne numéro 10
16     retour à la ligne numéro 11
17 </body>
18 </html>

```

Figure X.13: Résultat obtenu en appliquant le *template* X.6 sur l'exemple X.12.

Ceci s'explique par le fait que la balise `<xsl:apply-templates/>` appelle récursivement les *templates* de la feuille XSLT sur tous les *nœuds* du document XML et pas seulement sur les *éléments*. C'est-à-dire que les nœuds texte sont aussi sélectionnés. Or, comme ici, aucun *template* n'est défini pour les nœuds texte, c'est le comportement par défaut qui s'applique, c'est-à-dire que le texte des nœuds texte est simplement recopié dans le fichier de sortie.

Pour éviter cela, deux stratégies sont possibles:

- Redéfinir le comportement par défaut des nœuds texte, par exemple en définissant le *template* `<xsl:template match="text()"/>` qui du coup ne va rien écrire sur le fichier de sorties dans le cas de tous les nœuds texte
- Forcer la navigation à n'appliquer les *templates* que sur les sous-éléments et non sur tous les nœuds en remplaçant l'appel `<xsl:apply-templates/>` par `<xsl:apply-templates select="unCheminXPathQuiNeSélectionneQueDesSousElements"/>`

## 5 Programmation

Il existe d'autres instructions XSLT qui permettent de manipuler les données dans les *templates*. Ces instructions sont similaires à des instructions algorithmiques

- variable
- instructions conditionnelles
- boucles de parcours
- branchements multiples



Le langage XSLT est récursif. Plus exactement, le processeur XSLT (qui est dans une API du langage appelant (Java, C++, C#, JS ...)) fonctionne de manière récursive ; c'est lui qui va exécuter les boucles pour vous ; vous n'avez donc pas à demander explicitement des boucles (comme `foreach`). Le fonctionnement qui doit être adopté lors de la rédaction d'une feuille XSLT est le fonctionnement récursif d'appels de *templates* via XPath = vous avez juste à spécifier les templates pour les nœuds que vous voulez voir pris en charge. Les instructions *algorithmiques* ne doivent être utilisées qu'en dernier recours !

nom	description	exemple
variable	une variable a un nom et un type (string, number, boolean ou node-set). Sa valeur est donnée par <code>\$nomDeLaVariable</code> .	<pre>&lt;xsl:variable name="nb"&gt;15&lt;/xsl:variable&gt; &lt;xsl:variable name="p2" select="patient[2]"/&gt;</pre>
tri	On peut également trier la sélection des éléments lorsque l'on applique des <i>templates</i> . <code>&lt;xsl:sort select="critère xpath de tri"/&gt;</code>	<pre>&lt;ul&gt; &lt;xsl:apply-templates select="//patient"&gt; &lt;xsl:sort select="@id" order="descending"/&gt; &lt;/xsl:apply&gt; &lt;/ul&gt;</pre>
condition	Une condition permet d'exécuter une partie de <i>template</i> seulement lorsqu'une condition est vérifiée (Attention, il n'y a pas de <code>else</code> ). <code>&lt;xsl:if test="..."&gt;</code> ... <code>&lt;/xsl:if&gt;</code>	<pre>&lt;xsl:if test="age &lt; 6"/&gt; ... &lt;/xsl:if&gt;  &lt;xsl:if test="\$var != 5"/&gt; ... &lt;/xsl:if&gt;</pre>
boucle	Une boucle permet d'exécuter le même traitement pour un ensemble de nœud. Attention, cette instruction ne doit être utilisée qu'en dernier recours, l'instruction <code>xsl:apply-templates</code> doit être utilisée en priorité. <code>&lt;xsl:for-each select="expression xpath"&gt;</code> ... <code>&lt;/xsl:for-each&gt;</code>	<pre>&lt;xsl:for-each select="patient[@id&gt;50]"&gt; ... &lt;/xsl:for-each&gt;</pre>
branchements	Lorsqu'une condition comporte plusieurs possibilité, on peut utiliser un branchement multiple. <code>&lt;xsl:choose&gt;</code> <code>&lt;xsl:when test="..."&gt;...&lt;/xsl:when&gt;</code> <code>&lt;xsl:when test="..."&gt;...&lt;/xsl:when&gt;</code> <code>&lt;xsl:otherwise&gt;...&lt;/xsl:otherwise&gt;</code> ... <code>&lt;/xsl:choose&gt;</code>	<pre>&lt;xsl:choose&gt; &lt;xsl:when select="@id &lt; 20"&gt;...&lt;/xsl:when&gt; &lt;xsl:when select="@id &gt; 20 and @id &lt; 100"&gt;...&lt;/xsl:when&gt; &lt;xsl:otherwise&gt;...&lt;/xsl:otherwise&gt; ... &lt;/xsl:choose&gt;</pre>

On peut également réduire la dépendance au document d'entrée des instructions *push* grâce à l'élément `xsl:text` qui permet d'écrire du texte sans tenir compte du format d'entrée.

Enfin, l'élément `xsl:element` permet de créer facilement des élément (xml) dans le document de sortie. Par exemple,

```
1 <xsl:element name="maBalise">
2   Contenu de l'élément
3 </xsl:element>
```

donne en sortie le code suivant:

```
1 <maBalise>
2   Contenu de l'élément
3 </maBalise>
```

On peut également ajouter des attributs à cet élément grâce à l'élément `xsl:attribute`. Par exemple

```
1 <xsl:element name="img">
2   <xsl:attribute name="src">
3     <xsl:value-of select="photofile"/>
4   </xsl:attribute>
5   <xsl:attribute name="alt">
6     <xsl:value-of select="nom"/>
7   </xsl:attribute>
8 </xsl:element>
```

appliqué sur le morceau de document suivant:

```
1 <facebook>
2   <name>Golade Larry</name>
3   <photofile>fgl123.jpg</photofile>
4 </facebook>
```

donne le résultat suivant:

```
1 
```

