

# Parseur de code XML à partir de la DTD

Xavier Perséguers  
xavier.perseguers@epfl.ch

Informatique

Projet de semestre

Octobre 2003 – Février 2004

**Responsable**  
Prof. Claude Petitpierre  
claude.petitpierre@epfl.ch  
EPFL / LTI

**Superviseur**  
Duy Vo Duc  
duy.voduc@epfl.ch  
EPFL / LTI



# Table des matières

<b>1</b>	<b>Introduction</b>	<b>5</b>
1.1	Méthodologie . . . . .	5
1.2	Choix de la langue . . . . .	5
1.3	Nom du projet . . . . .	6
<b>2</b>	<b>Lecture du fichier DTD source</b>	<b>7</b>
<b>3</b>	<b>Génération de classes Java</b>	<b>9</b>
3.1	Classes issues des éléments de la DTD . . . . .	9
3.1.1	Ensemble des méthodes générées pour une liste . . . . .	9
3.2	Programme exemple . . . . .	10
3.3	Support des listes d'éléments . . . . .	10
3.3.1	Exemple de bloc <i>if-then-else</i> . . . . .	11
<b>4</b>	<b>Création d'une grammaire XML</b>	<b>13</b>
4.1	Choix conceptuels . . . . .	14
4.1.1	Exploitation de l'information de la DTD . . . . .	14
<b>5</b>	<b>Problèmes rencontrés</b>	<b>15</b>
5.1	Génération de classes . . . . .	15
5.1.1	Correspondance noms d'éléments / variables de classe . . . . .	15
5.2	Génération du parseur XML . . . . .	16
5.2.1	Utilisation des classes générées . . . . .	16
5.2.2	Utilisation adéquate des classes . . . . .	16
<b>6</b>	<b>Limitations</b>	<b>19</b>
<b>7</b>	<b>Installation</b>	<b>21</b>
7.1	Installations préliminaires . . . . .	21
7.1.1	Java . . . . .	21
7.1.2	JavaCC . . . . .	22
7.2	Installation de DTD2Java . . . . .	22
7.2.1	Création d'un alias . . . . .	22

<b>8 Exemple d'utilisation</b>	<b>23</b>
8.1 DTD2Java . . . . .	24
8.1.1 Choix des options . . . . .	24
8.1.2 Génération des fichiers . . . . .	25
8.1.3 Vérification du fonctionnement et documentation . . .	25
8.1.4 Listing des personnes . . . . .	26
8.1.5 Conclusion . . . . .	27

# 1 | Introduction

Les parseurs XML disponibles sont créés sur la base d'une architecture d'événements, ce qui complique le programme principal, puisqu'il ne peut pas diriger les opérations de récupération des symboles lui-même, et qu'il doit les délèguer au parseur. Ainsi qu'on l'a montré dans le cadre de l'accès à une interface interactive, il est tout à fait possible et même très efficace d'inverser cette structure et de laisser le contrôle des opérations au programme principal.

On demande donc de créer un parseur et un générateur de code qui crée deux parties : une qui décode les lignes du texte XML et une autre qui représente le squelette du programme principal, capable de récupérer un des éléments suivants possibles selon la syntaxe.

Ce projet, qui utilise le compilateur de compilateur JavaCC <sup>1</sup>, est disponible sur Internet en libre téléchargement à l'URL :

`http://ic2.epfl.ch/~persegue/dtd2java`

## 1.1 Méthodologie

Le projet a été réalisé en quatre étapes qui seront développées au fil de ce document :

1. Lecture du fichier DTD source ;
2. Génération de classes Java correspondant aux éléments de la DTD ;
3. Création d'une grammaire XML basée sur la DTD ;
4. Améliorations diverses comme la création d'un exemple de programme utilisant le parseur et corrections des erreurs.

**Note importante :** Dans la suite de ce document, nous utiliserons une DTD et un document XML exemples (listings 1.1 et 1.2).

## 1.2 Choix de la langue

Le projet a été entièrement développé en anglais ; que ce soit dans le choix du nom des variables ou des méthodes, ou dans les commentaires.

---

<sup>1</sup><https://javacc.dev.java.net/>

Listing 1.1 – DTD exemple

```
<!ELEMENT Personnes (Personne)*>
<!ATTLIST Personnes Class CDATA #FIXED "humain">
<!ELEMENT Personne (Prenom, Nom)>
<!ATTLIST Personne Age CDATA #REQUIRED>
<!ATTLIST Personne Sexe (M | F) "M">
<!ELEMENT Prenom (#PCDATA)>
<!ELEMENT Nom (#PCDATA)>
```

Listing 1.2 – Fichier XML exemple

```
<?xml version="1.0" standalone="no"?>
<Personnes Class="humain">
  <Personne Age="23">
    <Prenom>Xavier</Prenom>
    <Nom>Perseguers</Nom>
  </Personne>
  <Personne Age="30">
    <Prenom>Jim</Prenom>
    <Nom>XMLlover</Nom>
  </Personne>
</Personnes>
```

Ceci a pour but que n'importe quel programmeur puisse en continuer le développement à sa guise. Par contre, le présent document est rédigé en français car la description du projet, également donnée en français, semblait le suggérer.

### 1.3 Nom du projet

Tout projet se doit d'avoir un nom. Comme, de par sa conception, le projet se présente sous la forme d'un programme Java à lancer depuis un terminal ou une ligne de commande (selon l'OS utilisé), ce nom est assez court et explicite : DTD2Java. Il représente bien le fait qu'une DTD est « convertie » en code Java — ce qui se passe par la génération de classes — et, indirectement, par la création d'une grammaire XML qui est ensuite également transformée en code Java.

## 2 | Lecture du fichier DTD source

Le projet consiste à générer un parseur XML basé sur une DTD en utilisant le compilateur de compilateur JavaCC. Il apparaît alors comme judicieux de générer un parseur pour les fichiers DTD en utilisant également JavaCC.

Après une recherche sur Internet portant sur une grammaire BNF décrivant les DTD, il s'est avéré qu'une bibliothèque libre contenant entre autres un parseur de DTD — créé à l'aide d'une grammaire JavaCC — avait été développée par A. TOTOK dans son projet Exolab<sup>1</sup>.

Ce parseur de DTD a donc été retenu comme brique de base de ce projet. Le parseur est en fait composé de deux parseurs, le premier se charge de rechercher les déclarations d'entités paramétrées<sup>2</sup> et de substituer les références d'entités paramétrées par leur valeur correspondante. Le second est un parseur classique.

Le parseur produit un arbre hiérarchique de la DTD qui, pour l'exemple utilisé (listing 1.1), correspond à la structure de la figure 2.1.

La difficulté principale de cette étape, outre le fait de bien comprendre la façon dont sont enregistrés et référencés les éléments dans l'arbre, est de retrouver l'élément racine d'une instance XML correspondant à la DTD. Dans le cas d'une DTD incluse dans un document XML — on parle alors de DTD *locale* — il existe une déclaration `<!DOCTYPE` définissant le nom de l'élément racine. Dans le cas d'une DTD *externe*, qui est celui de ce projet, cette information n'est pas présente.

L'ordre de déclaration des éléments peut donc être quelconque, même si, en pratique, une convention tacite consiste à déclarer la racine comme premier élément de la DTD.

L'élément racine est trouvé en utilisant un graphe de dépendances et en cherchant l'unique élément non référencé par d'autres déclarations. S'il y en a plus d'un, c'est qu'il y a plusieurs racines et un message d'erreur est affiché.

---

<sup>1</sup><http://www.exolab.org>

<sup>2</sup><http://www.w3c.org/TR/2000/REC-xml-20001006#dt-PERef>

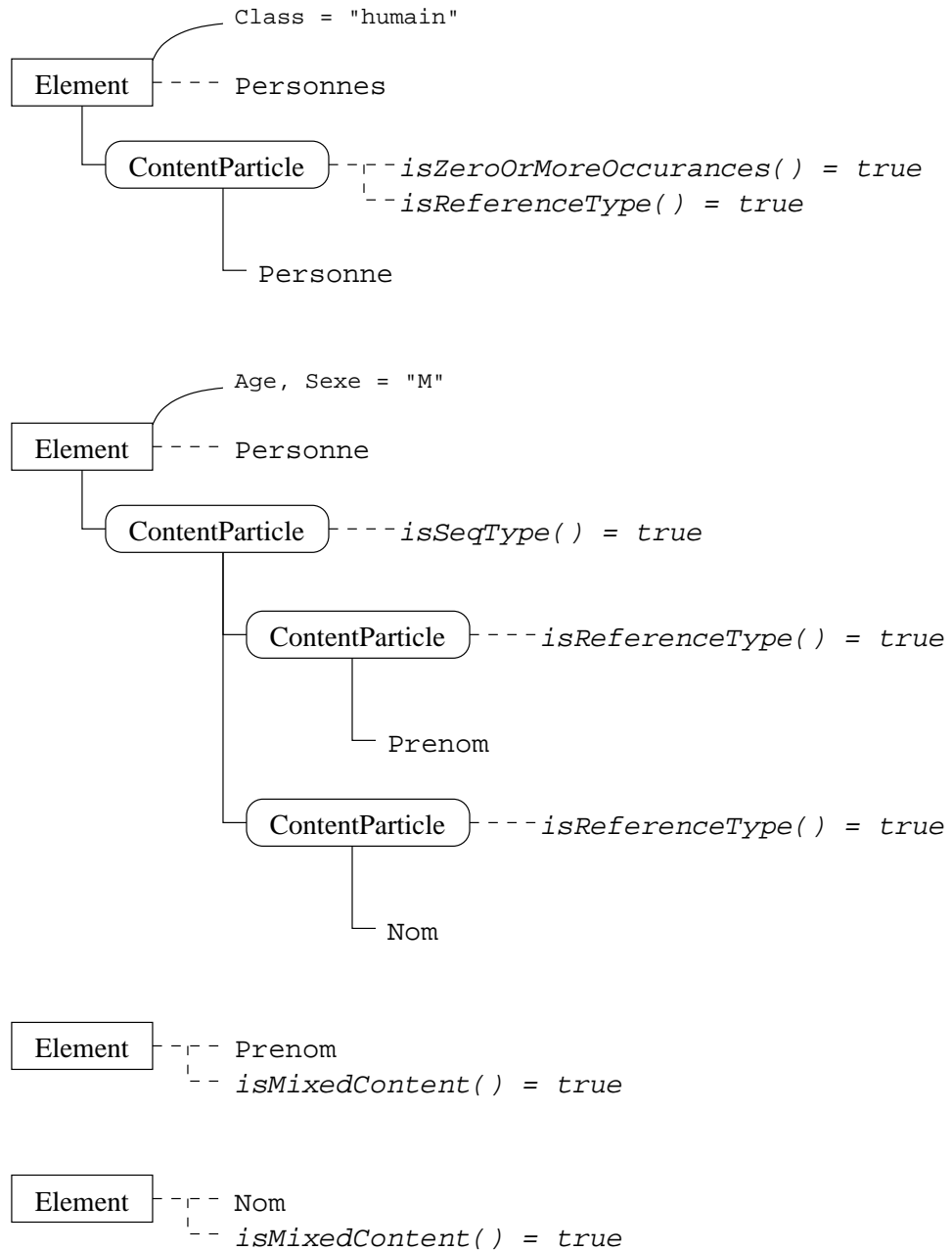


FIG. 2.1 – Arbre hiérarchique simplifié de la DTD exemple

# 3 | Génération de classes Java

Les classes générées sont certainement la partie centrale du projet, au moins au même titre que le parseur XML. C'est la façon qui a été retenue pour présenter l'information issue des fichiers que l'utilisateur choisit de lire. Il faut que cette représentation soit très facilement compréhensible, aisément accessible et plus agréable que les autres méthodes courantes d'accès aux fichiers XML, à savoir DOM et SAX.

## 3.1 Classes issues des éléments de la DTD

L'idée maîtresse de la génération de classes est de créer une classe par élément de la DTD, et de donner un accès hiérarchique aux sous-éléments logiques d'un élément donné, d'après l'information issue de la DTD. On désire qu'un attribut `Age`, par exemple, soit accessible par des méthodes standard `getAge()` et `setAge()` ; que les sous-éléments d'unicité zéro ou 1 soient accessibles d'une façon similaire et que les sous-éléments pouvant apparaître plusieurs fois aient des méthodes d'accès similaires aux listes de Java.

### 3.1.1 Ensemble des méthodes générées pour une liste

Le cas de sous-éléments d'unicité zéro ou 1 étant identique aux attributs, nous allons plutôt décrire les différentes méthodes générées pour une liste d'éléments ou un élément pouvant apparaître plusieurs fois, que ce soit un ou plusieurs fois (symbole `+`), ou zéro ou plusieurs fois (symbole `*`).

**getNomÉlément()** Cette méthode retourne une énumération des éléments `NomÉlément`. C'est la façon standard de parcourir une liste d'éléments.

**getArrayOfNomÉlément()** Seconde méthode d'accès à la liste complète des éléments, cette méthode retourne un tableau contenant les éléments, en maintenant l'ordre qu'ils avaient dans le fichier XML.

**getNomÉlémentAt()** Utilisé pour accéder à un élément d'après sa position.

**insertNomÉlémentAt()** Première méthode d'ajout d'un élément, à une position donnée.

**setNomÉlémentAt()** Remplacement d'un élément par un nouvel objet.  
**addNomÉlément()** Seconde méthode d'ajout d'un élément. Cette fois à la fin de la liste.  
**removeNomÉlément()** Suppression d'un élément donné.  
**removeAllNomÉlément()** Suppression de tous les éléments de la liste.  
**numberOfNomÉlément()** Fonction retournant le nombre d'éléments contenus dans la liste.

### 3.2 Programme exemple

Le programme exemple proposé lors de la création des classes de la DTD donne un exemple simple d'utilisation de celles-ci. Il commence par lire un fichier XML soit depuis un fichier — si un argument est passé en paramètre — soit depuis l'entrée standard (*standard in*). Il propose ensuite un bloc dans lequel l'utilisateur peut « travailler » sur les éléments lus ; *ie.* les afficher, les trier, les modifier... puis donne la syntaxe d'enregistrement des modifications. Par défaut, c'est un programme complet qui se contente de charger un fichier XML. Pour de plus amples informations sur l'utilisation des classes générées, un exemple complet pas-à-pas est proposé au chapitre 8, en page 23.

### 3.3 Support des listes d'éléments

Les énumérations ou les choix d'éléments *répétitifs* sont stockés dans des listes communes à l'unité hiérarchique de niveau supérieur. Si la DTD contient une déclaration du type

```
<!ELEMENT Vehicules (Voiture | Vélo | Moto)*>
```

alors une liste `childElements` est créée pour les regrouper. Ce choix de conception entraîne à la fois des avantages et des inconvénients par rapport à l'utilisation de listes séparées :

**Avantage :** L'ordre des éléments est conservé par rapport au fichier XML source. L'accès est unifié (une seule méthode quel que soit le type d'élément).

**Inconvénient :** L'énumération et le traitement des éléments d'une liste commune se révèlent compliqués : il n'est pas possible de faire un *cast* dynamique, car ça reviendrait à recréer le comportement qu'a un compilateur dans ses transformations. Il faut donc écrire un bloc *if-then-else* un peu compliqué pour séparer les différents cas. Ce problème est en partie résolu par l'utilisation d'une interface commune aux éléments. L'accès aux méthodes `save` et `isValid` est donc possible très simplement ; il suffit de faire un *cast* vers l'interface.

### 3.3.1 Exemple de bloc *if-then-else*

Dans le cas de la déclaration précédente pour les véhicules, les classes `Vehicules`, `Voiture`, `Velo` (sans accent) et `Moto` sont créées. Si l'on souhaite parcourir la liste des véhicules, et leur appliquer un traitement particulier, comme afficher certaines de leurs caractéristiques (partie non détaillée ici), il est possible d'utiliser le modèle de squelette du listing ci-dessous.

Listing 3.1 – Squelette d'accès aux listes d'éléments

```
Enumeration e = vehicules.getChildElements();

while (e.hasMoreElements()) {
    Object obj = e.nextElement();
    String className = obj.getClass().getName();

    if (className.equals("Voiture") {
        Voiture voiture = (Voiture)obj;

        // Traitement voiture

    } else if (className.equals("Velo") {
        Velo velo = (Velo)obj;

        // Traitement vélo

    } else if (className.equals("Moto") {
        Moto moto = (Moto)obj;

        // Traitement moto

    }
}
```



## 4 | Création d'une grammaire XML

La première étape est de se rendre sur le site du *World Wide Web Consortium*<sup>1</sup> pour y rechercher une définition exacte du langage XML. La spécification retenue est la version 1.0 seconde édition, recommandation du 6 octobre 2000<sup>2</sup>. Il faut ensuite créer une grammaire JavaCC qui corresponde à la recommandation, au moins dans la partie qui nous intéresse pour le projet. La créer *ex nihilo* est assez compliqué — on s'en rend compte par la suite — car la grammaire proposée n'est pas définie pour une machine mais bien plutôt pour un être humain. On remarque par exemple qu'il existe un *token* « Names » dont la définition est la suivante :

```
5. Names      ::= Name (S Name)*
```

Or la définition d'un élément est :

```
39. element   ::= EmptyElemTag | STag content ETag
40. STag      ::= '<' Name (S Attribute)* S? '>'
41. Attribute ::= Name Eq AttValue
```

Ce qui donne en remplaçant la définition 41 dans la définition 40 :

```
40. STag      ::= '<' Name (S Name Eq AttValue)* S? '>'
```

Le parseur JavaCC ne sait donc pas faire la différence entre un *token* « Names » et le début d'un élément contenant un attribut. Cet exemple n'est malheureusement pas unique. Une solution a été trouvée en étudiant une grammaire JavaCC pour XML disponible sur *The JavaCC Grammar Repository*<sup>3</sup> avec le parseur *XSilfide*<sup>4</sup>. Cette grammaire utilise une fonction mal documentée de JavaCC qui permet de demander au parseur de changer le mode dans lequel il se trouve pour adopter, par exemple, celui qui reconnaît les données textuelles plutôt que les attributs. Cette fonction est la méthode *SwitchTo*<sup>5</sup> des éléments *Token*.

---

<sup>1</sup><http://www.w3c.org>

<sup>2</sup><http://www.w3.org/TR/REC-xml>

<sup>3</sup><http://www.cobase.cs.ucla.edu/pub/javacc>

<sup>4</sup><http://www.loria.fr/projets/XSilfide>

<sup>5</sup><https://javacc.dev.java.net/doc/tokenmanager.html>

## 4.1 Choix conceptuels

La grammaire produite se veut aussi proche que possible de la définition des fichiers XML. De ce fait, elle préfère utiliser des méthodes génériques d'accès aux éléments, attributs et autres briques de base du format XML plutôt que de produire un code très spécifique à une grammaire DTD donnée. Ce choix permet une grande réutilisation du code d'accès aux fichiers XML et, de ce fait, une façon plus simple d'étendre les possibilités de DTD2Java à l'avenir.

### 4.1.1 Exploitation de l'information de la DTD

Une grammaire DTD définit un certain nombre d'informations complémentaires sur les fichiers XML à traiter. Elle permet par exemple de spécifier si un attribut est optionnel ou requis, si les valeurs qu'il peut prendre sont libres ou contraintes par un ensemble fini, etc. Le parseur XML créé par DTD2Java ne tient pas compte de certaines informations comme le fait qu'un attribut est requis ou que son ensemble de valeurs est soumis à certaines contraintes pour quelques raisons pratiques :

- la vérification de l'intégrité des données ne doit pas être faite uniquement lors de l'ouverture d'un document XML, mais pendant toute la durée de vie de la représentation mémoire de celui-ci, afin de garantir une réécriture correcte d'un fichier en sortie ;
- il n'est pas envisageable de mettre toutes les règles dans une grammaire JavaCC ; le fait d'avoir plusieurs attributs pour un même élément, certain optionnels, d'autres requis imposerait d'énumérer toutes les combinaisons d'entrelacements possibles (ordre factoriel).

Ces vérifications sont effectuées par les classes représentant les éléments eux-mêmes. Ce contrôle est donc à la charge du générateur de classes :

- un attribut à valeur fixe ne peut prendre d'autre valeur sous peine de générer une erreur ;
- un attribut requis doit être défini au moment de l'enregistrement de la représentation mémoire sur disque. Il n'est pas possible d'effectuer cette vérification en continu puisqu'il faut autoriser une violation temporaire des contraintes lors de la création ou la modification d'un élément ;
- un attribut ne pouvant prendre qu'un ensemble fini de valeurs est vérifié à chaque accès en écriture sur celui-ci. Une valeur non autorisée provoque le déclenchement d'une erreur ;
- un ensemble d'éléments devant apparaître au moins une fois (le « + » de la grammaire DTD) est autorisé à rester vide aussi longtemps que la représentation est gardée en mémoire pour la même raison qu'invoquée au point précédent.

## 5 | Problèmes rencontrés

Le présent chapitre donne une description aussi détaillée que possible de certains des problèmes rencontrés lors des phases de conception et d'implémentation du projet. La liste n'est évidemment pas exhaustive ; tout projet en comportant nécessairement. En outre, la plupart des « problèmes » se sont le plus fréquemment manifestés suite à des réflexions sur ce qu'un utilisateur *risquait* de proposer dans ses grammaires DTD.

### 5.1 Génération de classes

#### 5.1.1 Correspondance noms d'éléments / variables de classe

Le choix d'un nom pour la classe correspondant à un élément semble assez pragmatique : utiliser le nom de l'élément lui-même.

La première source de problèmes apparaît réellement avec le choix du nom des variables utilisées pour stocker les attributs et les sous-éléments d'un élément. Considérons l'exemple de DTD utilisé tout au long de ce document (listing 1.1, p. 6) et intéressons-nous à l'élément racine **Personnes**. Il possède un ou plusieurs éléments **Personne** que l'on peut aisément stocker dans un vecteur nommé **Personne** avec des méthodes d'accès `getPersonne()`<sup>1</sup> retournant une énumération des éléments du vecteur et une méthode `addPersonne()` permettant d'ajouter une personne à cet élément.

Qu'en est-il de l'attribut **Class** ? De prime abord rien ne nous empêche de déclarer une variable chaîne de caractères **Class**, mais dès que l'on génère les méthodes d'accès `getClass()` et `setClass()`, on se rend compte que `getClass()` est une méthode réservée dans Java. Si l'attribut s'était nommé **class**, la définition elle-même aurait été invalide pour cause d'utilisation d'un mot réservé du langage.

Autre problème pouvant apparaître : comment stocker le contenu PC-DATA d'un élément ? Certainement pas en créant une nouvelle classe ; l'intérêt serait assez limité. Mais alors comment nommer la variable utilisée pour stocker le texte correspondant ? Quelle que soit la méthode utilisée *à priori*, on risque d'utiliser un nom correspondant à un attribut de cette même classe.

La solution retenue est de créer une liste de mots réservés, tant par

---

<sup>1</sup>On ne peut malheureusement pas le mettre au pluriel étant donné que la langue utilisé pour la définition des éléments est inconnue

le langage lui-même que par les décisions prises par DTD2Java. Au début de la génération de chaque classe, cette liste est réinitialisée avec les mots réservés de Java ainsi que quelques noms comme `Text` et `Entities` qui sont nécessaires à la génération des classes. Le fait d'utiliser l'un de ces mots réservés dans une DTD n'affecte en rien les performances puisqu'un autre nom, basé sur le nom réservé est automatiquement généré pour éviter les collisions de noms de variables.

Cette démarche conduit à résoudre un autre problème. Il faut en effet se souvenir des décisions qui ont été prises pour que le parseur puisse assigner les valeurs des éléments des fichiers XML lus aux bonnes variables. Pour ce faire, une table de hashage est créée contenant le nom de l'élément selon la DTD comme clé et le nom effectif utilisé dans la classe comme valeur. Le nom des méthodes principales d'accès à ces variables est également enregistré pour faciliter le travail du parseur et éviter d'autres conflits possibles.

## 5.2 Génération du parseur XML

### 5.2.1 Utilisation des classes générées

Le parseur que l'on génère doit stocker les valeurs lues dans la DTD en mémoire dans une structure utilisant les classes générées précédemment. Cependant, si dans le cas des éléments l'on peut facilement définir des méthodes spécialisées à chaque type d'éléments dans le parseur, il n'en est pas de même pour les attributs. Les attributs peuvent en effet apparaître dans n'importe quel ordre. Comme le codage en dur de la méthode d'assignation d'une valeur donnée à une classe n'est pas envisageable, il apparaît clairement qu'il serait élégant d'utiliser le principe de *réflexion*. Cette méthode de programmation permet d'interroger une classe sur « ce qu'elle sait faire » et d'en déduire la méthode à utiliser pour une tâche donnée. Connaissant le nom que doit avoir une méthode pour définir un attribut donné, il ne reste plus qu'à demander à la classe de nous retourner un point d'accès sur la méthode dont on lui donne le nom, puis de l'invoquer en lui passant les paramètres requis.

### 5.2.2 Utilisation adéquate des classes

D'après la section 5.1.1, un attribut `class` sera transformé en `_class` tandis qu'un autre attribut `_class` sera transformé en `__class`. Mais s'ils avaient été définis dans l'ordre inverse, `_class` serait resté inchangé alors que `class` aurait été renommé en `__class`. Il n'y a donc pas de règle autre que l'ordre d'apparition des noms d'éléments ou d'attributs. Que se passe-t-il dans le cas d'attributs `class` et `_class` ? Si l'on n'y prend garde, on risque de copier la fonction de recherche d'un mot-clé valide dans le parseur pour qu'à chaque détection de *token* de type `name`, le parseur recherche le nom de

variable ayant été utilisé dans la classe. Or dans notre cas, celui d'attributs, rien n'empêche l'utilisateur d'utiliser ses deux attributs dans l'ordre inverse par rapport à la définition de la DTD.

#### Fragment de DTD

```
<!ELEMENT voiture EMPTY>
<!ATTLIST voiture class CDATA #REQUIRED>
<!ATTLIST voiture _class CDATA #REQUIRED>
```

#### Fragment de classe Voiture

```
public class Voiture {
    private String _class;
    private String __class;
    ...
}
```

#### Fragment XML correspondant à la DTD

```
<voiture class="vehicule" _class="4 roues"/>
<voiture _class="4 roues" class="vehicule"/>
```

Voici le contenu des variables `_class` et `__class` dans le cas des deux éléments `voiture` de l'exemple proposé.

**Premier cas :** `_class="vehicule"` et `__class="4 roues"`. L'ordre a été respecté et l'utilisateur de la classe est satisfait.

**Second cas :** `_class="4 roues"` et `__class="vehicule"`. Les valeurs des deux attributs ont été *logiquement* interchangées, ce qui n'est pas acceptable.

La solution retenue est de stocker les décisions d'assignation de noms de variables aux attributs lors de la génération des classes de façon telle que le parseur puisse retrouver le nom exact sans avoir besoin de refaire une démarche de réflexion dont il est nécessairement incapable.



## 6 | Limitations

Le projet possède quelques limitations qui ne l'empêchent pas de travailler avec la plupart des documents mais qui peuvent lui donner un comportement bizarre dans certains cas.

**Attributs obligatoires :** Certains attributs sont, de par leur définition dans la DTD, obligatoires. Ce qui veut dire qu'ils doivent être spécifiés pour chaque instance de document XML. Dans sa version actuelle, le parseur ne fait aucune vérification qu'un tel élément a bien été défini. Par contre, les attributs ayant une valeur fixe ou un choix de valeurs prédéfinies sont contraints de n'accepter que les valeurs autorisées. Cette vérification est faite directement au niveau de la classe correspondant à l'élément père. De cette façon, la vérification est faite pendant toute la vie de l'objet, y compris dans le programme utilisateur et *au moment de l'enregistrement* éventuel de la structure mémoire.

**Commentaires :** Tous les commentaires éventuellement présents dans le document XML lu par le parseur sont perdus si l'arborescence des objets résultante est enregistrée à la suite d'une modification éventuelle. Ce qui veut dire qu'un document XML devant être utilisé par un programme utilisateur ne devrait jamais être modifié manuellement.



# 7 | Installation

L'installation peut se faire de deux façons différentes :

- à partir des fichiers sources ;
- à partir d'une distribution.

L'avantage de la première sur la seconde est qu'elle permet de modifier le code même de `DTD2Java` ou de n'installer que ce qui vous intéresse, par exemple sans la documentation de `DTD2Java`. L'avantage d'utiliser une distribution est surtout pour les utilisateurs de Microsoft Windows ne disposant pas de l'environnement Cygwin. Cygwin<sup>1</sup> donne accès à un ensemble d'utilitaires issus du monde Unix sous Windows, et en particulier l'outil *make*<sup>2</sup>. Il faut cependant remarquer que `DTD2Java` ne sait produire que des fichiers *Makefile* mais aucun programme *batch* pour la compilation des classes générées. Vous devrez donc, le cas échéant, compiler manuellement les classes générées.

## 7.1 Installations préliminaires

Avant de pouvoir utiliser `DTD2Java`, et quel que soit le type d'installation choisi, vous devez posséder une version de Java et JavaCC correctement installées et configurées.

### 7.1.1 Java

`DTD2Java` a été développé avec et pour la version 1.4.x de la machine virtuelle Java. Téléchargez la dernière version à l'URL

<http://java.sun.com/j2se/downloads.html>

puis installez-la en suivant les instructions du programme d'installation. En ouvrant un terminal ou une ligne de commande (selon l'OS utilisé), vous devriez pouvoir taper `java` et obtenir la description des options utilisables. Si ce n'est pas le cas, vérifiez que votre variable d'environnement `PATH` contienne le chemin d'accès au répertoire `/bin` de Java.

---

<sup>1</sup><http://www.cygwin.com>

<sup>2</sup><http://www.gnu.org/software/make/make.html>

### 7.1.2 JavaCC

La version choisie de JavaCC ne devrait poser aucun problème. DTD2Java a néanmoins été développé en utilisant la version 3.2. Téléchargez la dernière version à l'URL

```
https://javacc.dev.java.net/servlets/ProjectDocumentList
```

puis décompressez simplement l'archive dans le répertoire de votre choix. N'oubliez pas ensuite de référencer le répertoire `/bin` de JavaCC pour être en mesure de l'exécuter de n'importe quel répertoire. Pour tester que l'installation est bien effectuée, ouvrez un terminal ou une ligne de commande et tapez `javacc`. Vous devriez, comme dans le cas de Java, obtenir la description des options utilisables.

## 7.2 Installation de DTD2Java

Commencez par télécharger la version que vous désirez à l'URL

```
http://ic2.epfl.ch/~persegue/dtd2java/download.html
```

Dans le cas d'une distribution, il vous suffit de décompresser l'archive dans le répertoire de votre choix. Si vous choisissez une installation à partir des fichiers sources, décompressez l'archive dans le répertoire de votre choix et exécutez le script `install.sh`. Ce script compilera automatiquement DTD2Java et générera la documentation. N'hésitez pas à modifier le script d'installation pour satisfaire vos besoins spécifiques.

### 7.2.1 Création d'un alias

L'utilisation de DTD2Java est grandement simplifiée si vous créez un *alias* pour exécuter le programme. La syntaxe est propre à chaque shell (`sh`, `bash`, `csh`, ...) et l'alias devrait être ajouté au script de démarrage du shell (`.cshrc` pour `csh`, `.bashrc` pour `bash`, etc.). En supposant que vous ayez installé DTD2Java dans le répertoire `/usr/bin/`, un exemple d'alias serait :

```
alias dtd2java="java -jar /usr/bin/dtd2java/dtd2java.jar"
```

Vous pouvez dès lors exécuter DTD2Java depuis n'importe quel répertoire en tapant simplement le nom de l'alias.

## 8 | Exemple d'utilisation

Nous allons à présent montrer comment utiliser ce projet sur un exemple pratique de document XML contenant des informations sur des personnes. La DTD correspondante, un peu différente de la DTD exemple utilisée jusqu'alors, est la suivante :

Listing 8.1 – step-by-step.dtd

```
<!ELEMENT Personnes (Personne)*>
<!ELEMENT Personne (Nom, Prénom, Profession)>
<!ATTLIST Personne Age NMTOKEN #REQUIRED
                Sexe (M | F) "F">
<!ELEMENT Nom (#PCDATA)>
<!ELEMENT Prénom (#PCDATA)>
<!ELEMENT Profession (#PCDATA)>
```

Nous avons une instance particulière de fichier XML contenant nos personnes et nous souhaiterions y accéder de façon simple :

Listing 8.2 – step-by-step.xml

```
<Personnes >
  <Personne Age="35" Sexe="M">
    <Nom>Dupont </Nom>
    <Prénom>Jean </Prénom>
    <Profession>Carreleur </Profession>
  </Personne >
  <Personne Age="24">
    <Nom>Prunelier </Nom>
    <Prénom>Julie </Prénom>
    <Profession>Etudiante </Profession>
  </Personne >
  <Personne Age="10">
    <Nom>Quattre </Nom>
    <Prénom>Emilia </Prénom>
    <Profession>Designer </Profession>
  </Personne >
</Personnes >
```

## 8.1 DTD2Java

Si, lors de l'installation, nous avons fait un alias sur DTD2Java, nous pouvons simplement taper `dtd2java --help` pour obtenir de l'aide sur les options de lancement, sinon, il faut indiquer le chemin d'accès complet à l'archive jar : `java -jar projects/dtd2java.jar --help` par exemple.

### 8.1.1 Choix des options

Avec l'option de lancement `--help`, DTD2Java affiche l'aide suivante :

```
Usage: <DTD2Java> [OPTION]... [DTDNAME[.dtd]]
```

```
Run DTD2Java on DTDNAME, usually creating a parser DTDNAME.jj and Java
classes corresponding to the elements of the input DTD.
```

```
If no arguments nor options are specified, prompt for input.
```

```
-d --output-directory <directory> specify where to place generated files.
                                Default is current directory
-e --example <file>             create an example (<file>.java) on how
                                using the generated grammar and classes.
-m --makefile                   create a Makefile for all generated files
  --no-save                     do not generate method save for classes
-o <file>                       place the output parser into <file>
-p --package-name <package>    make generated files be part of package
                                <package>. Default is none
-P --parser-class <name>       name the class XML parser be <name>.
                                Default is 'XMLParser'
-v --verbose                   explain what is being done
  --help                       display this help and exit
  --version                    output version information and exit
```

Voici notre « cahier des charges » :

1. un parseur pour nos fichiers XML que nous voulons avoir dans une classe *XMLParser*, ie. le nom proposé par défaut pour les parseurs XML ;
2. un exemple d'utilisation, *Example.java*, du parseur généré ;
3. un fichier Makefile pour être en mesure de compiler correctement les classes et le parseur générés ;
4. finalement nous souhaitons que DTD2Java nous indique où il en est de la création des fichiers.

### Options à utiliser

**Point n° 1.** Étant donné que l'option par défaut est satisfaisante, nous n'avons rien à spécifier.

**Point n° 2.** `-e Example.java`

**Point n° 3.** `-m` ou `--makefile`

**Point n° 4.** `-v` ou `--verbose`

### 8.1.2 Génération des fichiers

Il suffit désormais de lancer DTD2Java sur notre fichier `step-by-step.dtd` :

```
bash-2.05b$ dtd2java -e Example.java -m -v step-by-step.dtd
```

DTD2Java affiche alors ce qu'il fait :

```
*****
DTD2Java version "1.0.0"
*****
Input DTD:          step-by-step.dtd
Output JavaCC:     ./step-by-step.jj
Output Example:    ./Example.java
Output Directory:  ./
Parser Class:      XMLParser

Loading DTD...Done

Generating file Personnes.java...Done
Generating file Personne.java...Done
Generating file Profession.java...Done
Generating file Prenom.java...Done
Generating file Nom.java...Done
Generating file Entities.java...Done
Generating file IElementCommonInterface.java...Done
Generating file step-by-step.jj...Done
Generating file Example.java...Done

Generating Makefile...Done
```

### 8.1.3 Vérification du fonctionnement et documentation

Compilons le projet puis exécutons le programme exemple en lui passant un fichier en argument, en l'occurrence, notre fichier `step-by-step.xml`. Si nous ne passons aucun fichier en argument, le programme exemple lira un fichier XML depuis l'entrée standard, ce qui peut toujours s'avérer pratique.

```
bash-2.05b$ make
...
bash-2.05b$ java Example step-by-step.xml
bash-2.05b$
```

Le programme s'est donc terminé normalement. Rien de très impressionnant pour l'instant ! Essayons maintenant d'utiliser les classes issues de notre DTD. Pour cela, commençons par générer leur documentation :

```
bash-2.05b$ make javadoc
```

Il ne nous reste plus qu'à ouvrir le fichier `docs/index.html` qui a été créé pour avoir une aide complète sur les classes et le parseur XML (figure 8.1). Il faut également noter qu'il est possible de modifier les premières lignes du fichier `Makefile` pour personnaliser l'aide produite avec le nom de notre société, l'année de copyright...

#### 8.1.4 Listing des personnes

Imaginons que nous souhaitons un listing du nom et du prénom de chaque personne se trouvant dans notre fichier `step-by-step.xml`. Commençons par ouvrir le fichier `Example.java` et ajoutons une méthode `showListing()` à la classe :

```
1 private void showListing() {
2
3 }
```

En lisant la description de la classe `Personnes`, qui est la classe racine de notre DTD, nous trouvons une méthode `getPersonne()` qui retourne une énumération de `Personne`, classe représentant une personne dans notre DTD. Nous pouvons donc écrire :

```
1 private void showListing() {
2     for (Enumeration e = personnes.getPersonne();
3         e.hasMoreElements(); ) {
4
5         Personne p = (Personne)p.nextElement();
6     }
7 }
```

Il ne reste plus qu'à afficher le nom et le prénom des personnes. La documentation nous indique les méthodes `getPrenom()` et `getNom()` de la classe `Personne`. Une méthode standard d'accès au contenu PCDATA d'un élément est disponible avec `getText()` et nous obtenons donc :

```
1 private void showListing() {
2     for (Enumeration e = personnes.getPersonne();
3         e.hasMoreElements(); ) {
4
5         Personne p = (Personne)e.nextElement();
6         System.out.println(p.getNom().getText() + " "
7             + p.getPrenom().getText());
8     }
9 }
```

Ajoutons encore un appel à la méthode `showListing()` dans la méthode `main`.

Il ne reste plus qu'à recompiler le projet et réexécuter la classe `Example` :

```
bash-2.05b$ make
...
bash-2.05b$ java Example step-by-step.xml
Dupont Jean
Prunelie Julie
Quatre Emilia
bash-2.05b$
```

### 8.1.5 Conclusion

L'accès aux fichiers XML se trouve grandement facilité par l'utilisation de `DTD2Java`. Il faut cependant remarquer que cette méthode ne s'applique que si l'on a une structure clairement définie de DTD. Ce petit tour d'horizon ne demande maintenant qu'à s'élargir avec vos propres expériences.

Nom ([YOUR COMPANY] - API Specification) - Mozilla Firefox

file:///home/tp/projects/dtd2java/tmp/docs/index.html

Mozilla Firefox Help Mozilla Firefox Discu... Plug-in FAQ

All Classes

Entities  
Example  
[IElementCommonInterface](#)  
Nom  
ParseException  
Personne  
Personnes  
Prenom  
Profession  
SimpleCharStream  
Token  
TokenMgrError  
XMLParser  
XMLParser.JJCalls  
XMLParserConstants  
XMLParserTokenManager

Package **Class** Use Tree Deprecated Index Help XMLParser API

PREV CLASS NEXT CLASS  
SUMMARY: NESTED | ENUM  
CONSTANTS | FIELD | [CONSTR](#) | [METHOD](#)  
DETAIL: FIELD | [CONSTR](#) | [METHOD](#)

## Class Nom

java.lang.Object  
└ Nom

All Implemented Interfaces:  
[IElementCommonInterface](#)

public class Nom  
extends java.lang.Object  
implements [IElementCommonInterface](#)

Implementation of the XML element Nom specification, used to define the content of this element.

Version:  
\$Revision: 1.0.0.0 \$ \$Date: 2004/01/20 20:59 \$

Author:  
DTD2Java Class Generator

### Constructor Summary

[Nom\(\)](#)  
Constructor, initializing all attributes to null-length string or the default value defined in the corresponding DTD, populating lists of attributes with authorized values, and setting all items to null.

### Method Summary

java.lang.String	<a href="#">getText()</a>	Return the content (PCDATA) of this element.
boolean	<a href="#">isValid()</a>	Return TRUE if this XML element is valid against its definition, otherwise return FALSE.
void	<a href="#">save(java.io.PrintWriter out, int indent)</a>	Save this element in a XML file
void	<a href="#">setText(java.lang.String value)</a>	Set the content (PCDATA) of this element.

Methods inherited from class java.lang.Object

file:///home/tp/projects/dtd2java/tmp/docs/Nom.html

FIG. 8.1 – Documentation des classes et du parseur générés

# Listings

1.1	DTD exemple . . . . .	6
1.2	Fichier XML exemple . . . . .	6
3.1	Squelette d'accès aux listes d'éléments . . . . .	11
8.1	step-by-step.dtd . . . . .	23
8.2	step-by-step.xml . . . . .	23