

**MooX**

**– Manuel de Conception –**

## Table des matières

Introduction.....	3
I. Conception globale.....	3
I.1 Fonctionnement.....	3
I.2 Le modèle abstrait.....	4
I.3 Le format JSON.....	5
II Organisation du code.....	7
II.1 Les générateurs et les analyseurs.....	7
II.2 Organisation des packages.....	8

## Introduction

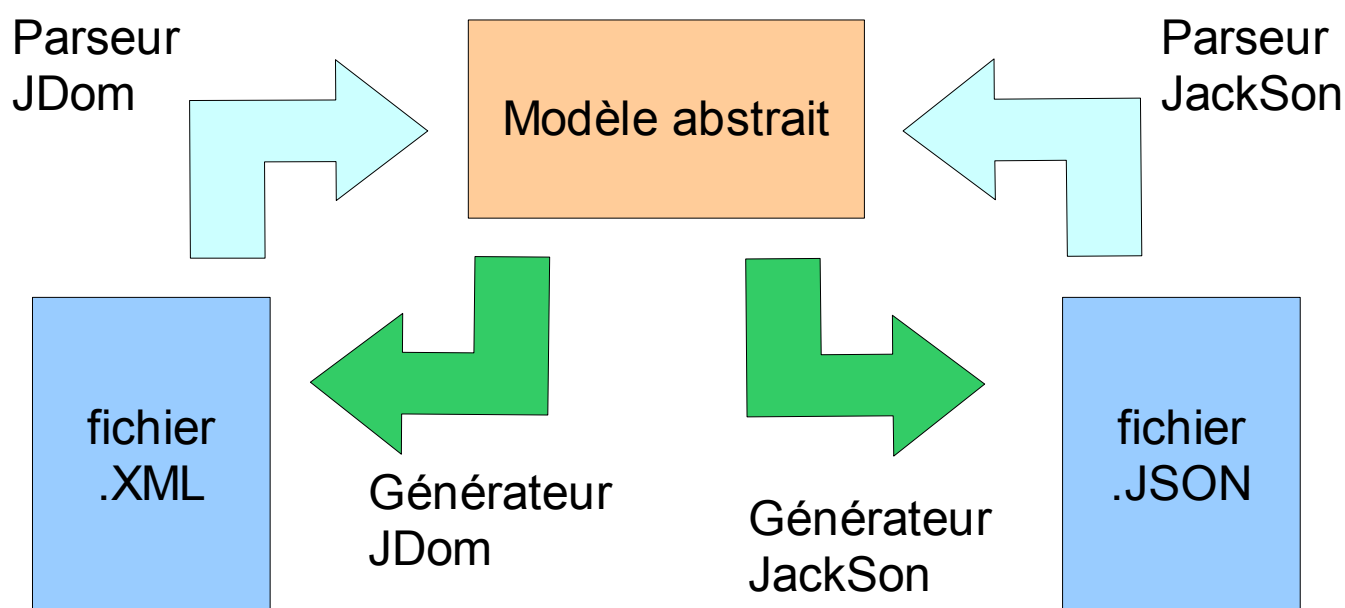
Le logiciel a pour vocation à transformer les fichiers des quiz de moodle. Ces fichiers sont originellement au format XML et MooX doit les convertir au format JSON. L'opération inverse doit également être possible.

L'objectif de ce document est de présenter la manière dont a été conçue le logiciel MooX afin que des développeurs puissent maintenir et faire évoluer le logiciel.

## I. Conception globale

### I.1 Fonctionnement

La conception du logiciel MooX sépare la partie d'analyse du fichier source à transformer de la partie génération du fichier cible. Pour relier ces deux fonctionnalités, le logiciel transforme le fichier source en un modèle abstrait prédéfini qui sera ensuite utilisé pour la génération du fichier cible.



De cette manière, les différents parseurs et générateurs sont indépendants. La seule chose qui doit être définie est le modèle abstrait. De plus, cette méthode permet de rajouter autant de formats de fichier que l'on veut. Enfin, d'un point de vue répartition des tâches, chaque personne peut travailler sur un module différent sans avoir aucune dépendance aux autres modules (mis à part au modèle abstrait défini conjointement).

A l'heure actuelle, la gestion des fichiers XML (que ce soit pour la génération ou pour l'analyse de fichier) est faite grâce à la bibliothèque JDOM 2<sup>1</sup>. Cette bibliothèque a l'avantage de

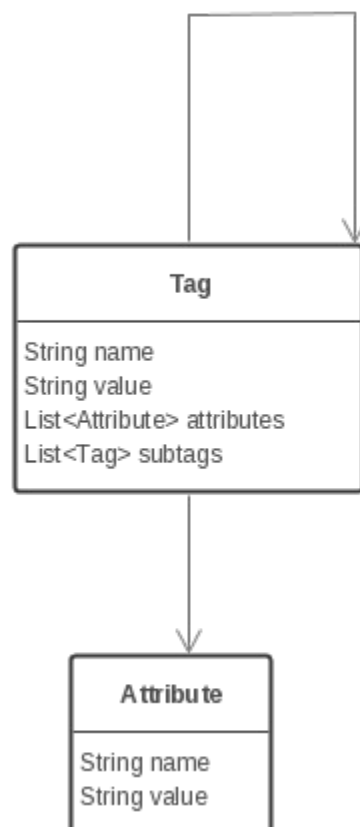
<sup>1</sup> <http://www.jdom.org/>

faire une analyse en créant directement un arbre depuis la structure XML ce qui facilite sa manipulation pour le transformer tel que défini dans le modèle abstrait.

Pour la gestion des fichiers XML (génération et analyse), c'est la bibliothèque Jackson<sup>2</sup> que nous utilisons. Cette bibliothèque fonctionne à la fois pour la génération de fichier JSON et pour l'analyse et reste simple d'utilisation. De plus, elle est sous licence LGPL ce qui nous permet de rester compatible avec la licence de notre logiciel (GPL).

## 1.2 Le modèle abstrait

Voici une représentation en UML du modèle abstrait utilisé comme jonction entre les analyseurs et les générateurs:



Comme on peut le voir, ce modèle est très simple. En fait, c'est la manière la plus simple de représenter une structure arborescente (à ceci près que les éléments peuvent posséder des attributs). Nous avons donc un nœud (Tag) qui possède un nom, parfois une valeur, une liste d'attributs et la liste de ses fils.

Cette représentation est suffisante pour représenter un document XML ou JSON. On notera que le concept d'attribut n'est pas présent dans le format JSON et qu'il a donc fallu trouver un moyen combler ce manque en créant une structure JSON particulière.

<sup>2</sup> <http://jackson.codehaus.org/>

### 1.3 Le format JSON

Comme nous venons de le dire, le format JSON, bien que léger, présente certains manques par rapport au format XML. Les deux formats ne sont donc pas strictement équivalents. Pour palier à ce problème, nous avons défini la structure de ce que devait être un format JSON qui correspondrait à un fichier XML quelconque.

Correspondance XML / JSON	
XML	JSON
Un élément est identifié par son nom: <code>&lt;name&gt; ... &lt;/name&gt;</code>	Un élément est qualifié par un objet ayant également un nom: <code>{ "name": ... }</code>
Un élément peut posséder une valeur textuelle: <code>&lt;name&gt;text&lt;/name&gt;</code>	Cette valeur textuelle sera représentée par l'attribut "text" de l'objet. (Sa valeur pouvant être null): <code>{ "name":   { "text": "text"   ...   } }</code>
Un élément peut avoir des attributs: <code>&lt;name attribut="valeur"/&gt;</code>	Les attributs seront modélisés par un tableau nommé "attributes": <code>{ "name":   { "text": null,     "attributes": [       { "attribut": "valeur" }     ]   ...   } }</code>
Un élément peut posséder des sous-éléments: <code>&lt;name&gt;   &lt;sub/&gt; &lt;/name&gt;</code>	Les sous-éléments seront modélisés par un tableau nommé "subTags": <code>{ "name":   { "text": null,     "attributes": [],     "subTags": [       { "sub"         {           ...         }       } ]     } }</code>

Exemple:

```
<tagName1 att1="value" att2="value">
  <tagName2>value</tagName2>
  <tagName3>
    ...
  </tagName3>
</tagName1>
```

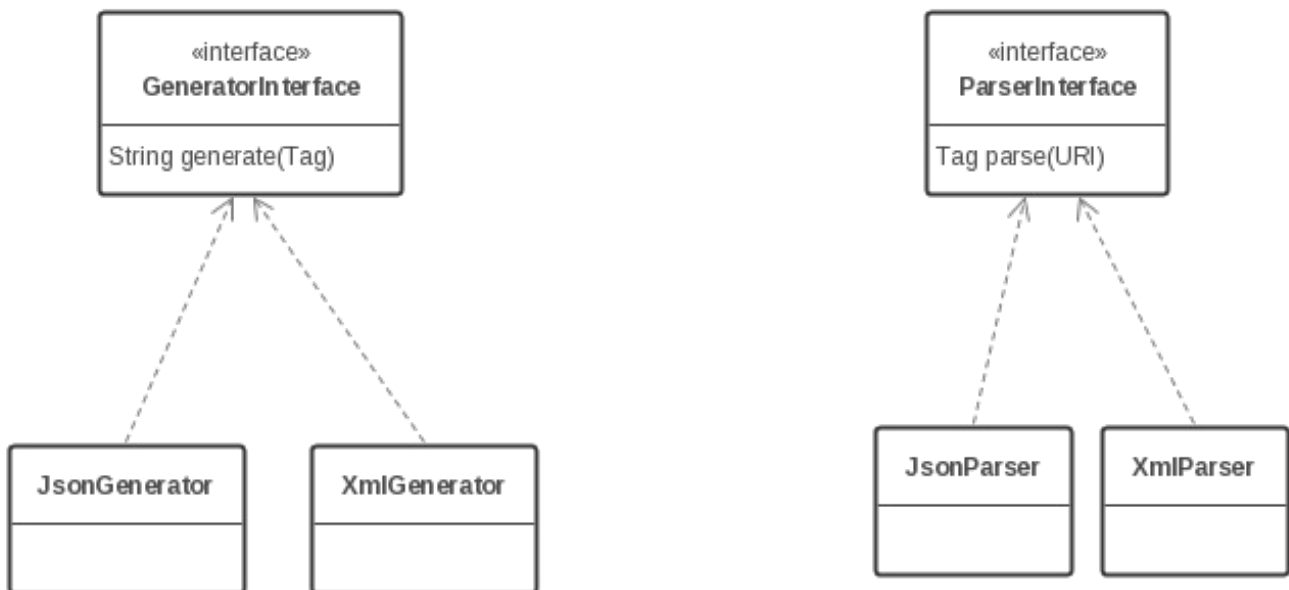
Exemple:

```
{
  "tagName1":
  {
    "text": null,
    "attributes":
    [
      { "att1": "value" },
      { "att2": "value" }
    ]
    "subTags":
    [
      { "tagName2":
        {
          "text": "value",
          "attributes":
          [
          ]
          "subTags":
          [
          ]
        }
      },
      {
        "tagName3":
        {
          ...
        }
      }
    ]
  }
}
```

## II Organisation du code

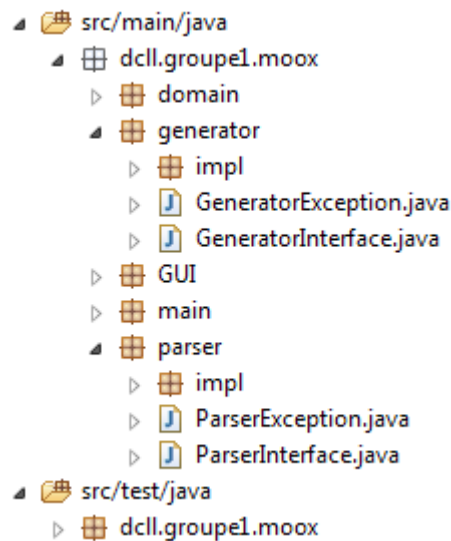
### II.1 Les générateurs et les analyseurs

Pour plus d'harmonie, les générateurs et les analyseurs implémentent une interface qui leur est propre :



Rien de plus simple, la méthode `generate()` sert à générer une chaîne représentant le fichier cible et prend en argument la racine du modèle abstrait. La méthode `parse()` quant à elle génère un objet représentant la racine dans le modèle abstrait à partir d'une URI.

## II.2 Organisation des packages



Le package principal contient 5 sous-packages:

- `domain`: qui contient le modèle abstrait tel que décrit précédemment.
- `Generator`: qui contient l'interface des générateurs et un sous-package des implémentations des générateurs.
- `GUI`: qui contient le code concernant l'IHM.
- `Main`: qui ne contient que la classe principale du programme.
- `Parser`: qui contient l'interface des parseurs et un sous-package des implémentations des parseurs.