MooX

– Manuel de Conception –

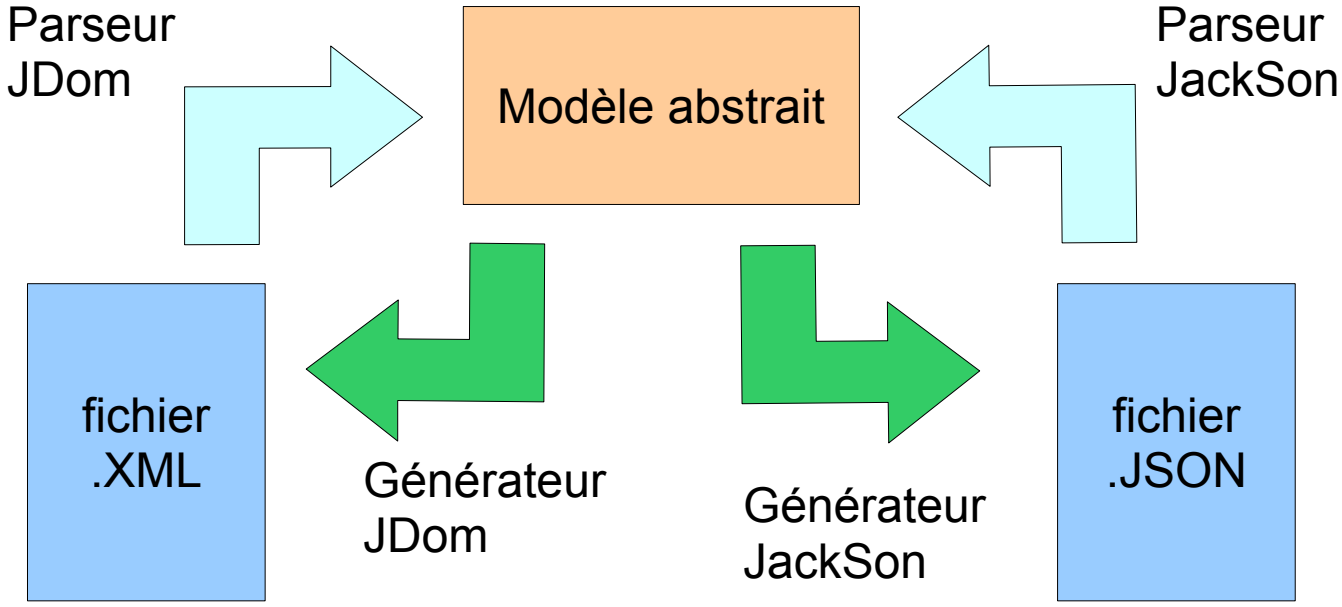
Rédigé par: SERIN Kévin

Table des matières

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 prédéfinit qui sera ensuite utiliser pour la génération du fichier cible.



De cette manière, les différents parseurs et générateurs sont indépendants. La seule chose qui doit être définit est le modèle abstrait. De plus, cette méthode permet de rajouter autant de format 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 le modèle abstrait définit conjointement).

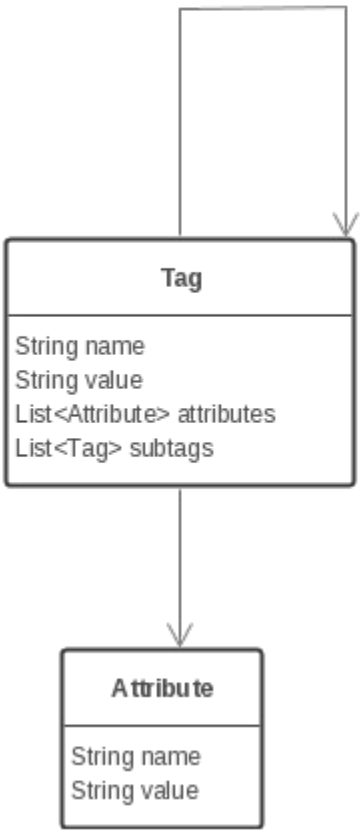
A l'heure actuelle, la gestion des fichiers XML (que ce soit pour la génération ou pour l'analyse de fichier) est faite grâce à la bibliothèque JDOM 2[1]. Cette bibliothèque à l'avantage de faire une analyse en créant directement un arbre depuis la structure XML ce qui facilite sa manipulation pour le transformer tel que définit dans le modèle abstrait.

Pour la gestion des fichiers XLK (génération et analyse), c'est la bibliothèque Jackson[2] 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 est un avantage considérable.

1 http://www.jdom.org/
2 http://jackson.codehaus.org/

I.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 un structure JSON particulière.

I.3 Le format JSON

Comme nous venons de le dire, le format JSON, bien que léger, présente certains manque par rapport eu format XML. Les deux formats ne sont donc pas strictement équivalent. Pour palier à ce problème, nous avons définit 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: <name> … </name>	Un élément est qualifié par un objet ayant également un nom: { "name": … }
Un élément peut posséder une valeur textuelle: <name>text</name>	Cette valeur textuelle sera représenté par l'attribut "text" de l'objet. (Sa valeur pouvant être nulle): { "name": { "text": "text" … } }
Un élément peut avoir des attributs: <name attribut="valeur"/>	Les attributs seront modélisés par un tableau nommé "attributes": { "name": { "text": null, "attributes": [{"attribut": "valeur" }] … } }

Un élément peut posséder des sous-éléments:
 	
<name>
<sub/>
</name>
	Les sous-éléments seront modélisés par un tableau nommé "subTags":
	{ "name":
	{ "text": null,
	"attributes": [],
	"subTags": [
	{ "sub"
	{
	…
	}
	}]
	}
	}

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”:
                  {
                     …
                  }
               }
      ]
   }
}
```

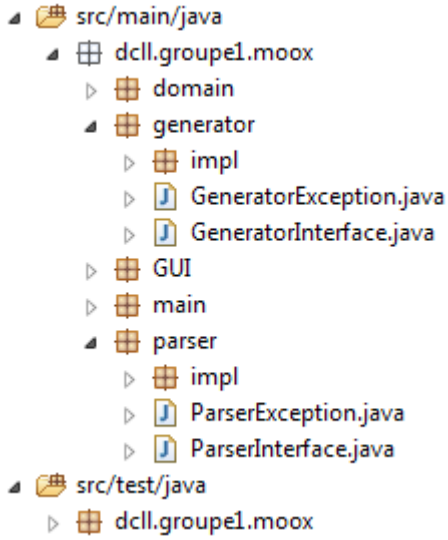
II Organisation du code

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 chaine 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-packages 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.