Génie logiciel GLCB2 :
TP3
Génération de code Java pour le patron composite

Jean-Claude Royer

5 janvier 2011

Les objectifs de ce TP sont :
- Représenter partiellement un modèle de programme par des objets Java.
- Travailler un exemple de génération de code.
- Expérimenter pratiquement la généricité par template.

Dans un outil comme ArgoUml vous avez une partie permettant d"éditer graphiquement divers modèles UML et une partie permettant la génération de code (et d'autres parties que j'ignore ici). Vous avez déjà utilisé la génération de code Java et la partie graphique sera abordée avec le cours d'IHM. Dans ce TP vous allez acquérir une bonne idée de la génération de code en Java. Dans l'outil graphique les dessins de modèles (association, classes, méthodes, attributs, etc) sont représentés grâce à des instances de classes Java et ensuite le générateur de code prends ces instances en entrée pour produire du code source Java.

1 Liste d'entiers

Récupérez sur campus l'archive `tp3.jar` et créer un nouveau projet à partir de celle-ci. Il est utile de lire le fichier `resources/README` qui contient des explications sur la structure du projet. Vous trouverez le code Java pour des listes d'entiers dans le style du patron composite vu en cours. Il est disponible dans le paquetage `patron`. Il me semble judicieux de lire les classes dans l'ordre suivant : `List, Empty` et `NotEmpty`. Vous pouvez tester le programme en utilisant la classe `tests.TestPatron`, ne pas oublier de faire un `refresh` de votre projet pour avoir l'état courant à jour. Nous allons représenter ce patron par des classes Java et à partir de ces classes générer le code du patron. En effet le code précédent ne marche que pour des listes d'entiers, mais si on veut des listes de caractères ou de chaînes, comment faire ?

Nous allons donc réaliser un générateur de code Java, *i.e.* un programme (écrit en Java) qui prends en entrée les données (le patron des listes) et qui va produire du code Java correspondant à ce patron. Ensuite ce code pourra être compilé et exécuté comme usuellement. Il est donc facile de se mélanger les neurones, la figure 1 peut aider à comprendre les différents éléments. Les éléments que vous dessinez habituellement sont en réalité mémorisés sous forme d'objets au sens de la PPO. Pour définir le format de ces objets nous avons besoin d'une description sous forme d'un diagramme de classe (on parle de modèle). Ces objets (instances des classes du diagramme) vont servir de données d'entrée à votre programme de génération. En gros celui-ci analyse la structure du patron et va générer les fichiers avec les instructions Java représentant le patron composite.

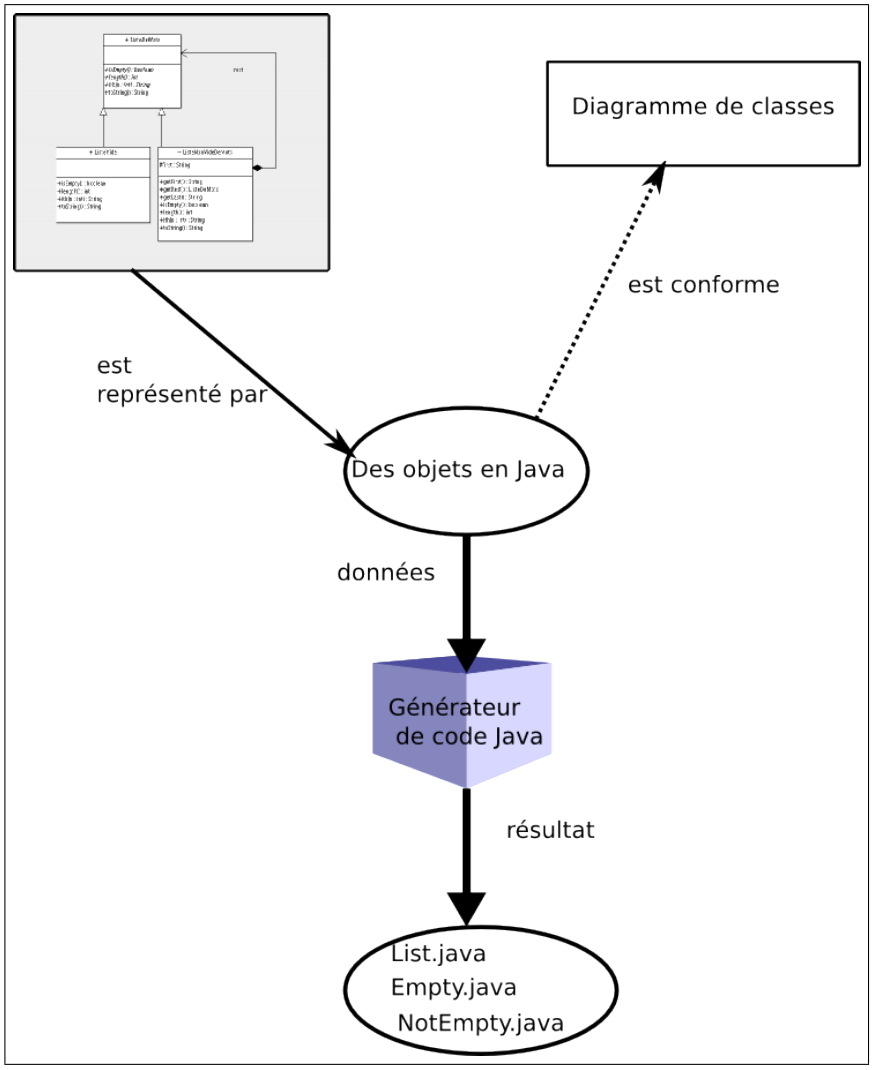

FIGURE 1 – Le principe de la génération de code

2 Représentation Java du patron composite

Le diagramme de classes ci-dessous (disponible dans `resources/diagramme`) représente notre modèle pour la représentation du patron. Il s'agit de définir des classes Java pour représenter les éléments constitutifs d'une classe, donc des attributs, des méthodes etc. Faire un modèle complet est un problème plutôt complexe et du domaine de l'API réflexive Java, voir par exemple `http://www-igm.univ-mlv.fr/~dr/XPOSE2004/ramassamy/api.php` ou le JDT pour les curieux. Ici nous allons considérer un cadre restreint mais suffisant pour notre problème :

- Les classes, les attributs et les méthodes sont publiques
- Une classe concrète définit un unique constructeur complet, par exemple si la classe a deux attributs `a:A` et `b:B` on va générer le constructeur suivant :

```
public LaClasse( A a, B b)  {
    this.a = a;
    this.b = b;
```

```
}
```

Dans le paquetage `input` vous trouverez une implémentation partielle des classes pour représenter le patron et générer le code. Nous avons ajouté une classe `Write` dont le rôle est de faciliter l'écriture dans un fichier. La classe `tests.TestGeneration` permet de tester la génération de code à partir de ce modèle. La génération n'est pas complète et votre tâche va être d'améliorer et de compléter celle-ci. On ne se souciera pas trop de la présentation du programme généré, par exemple de l'indentation des lignes.

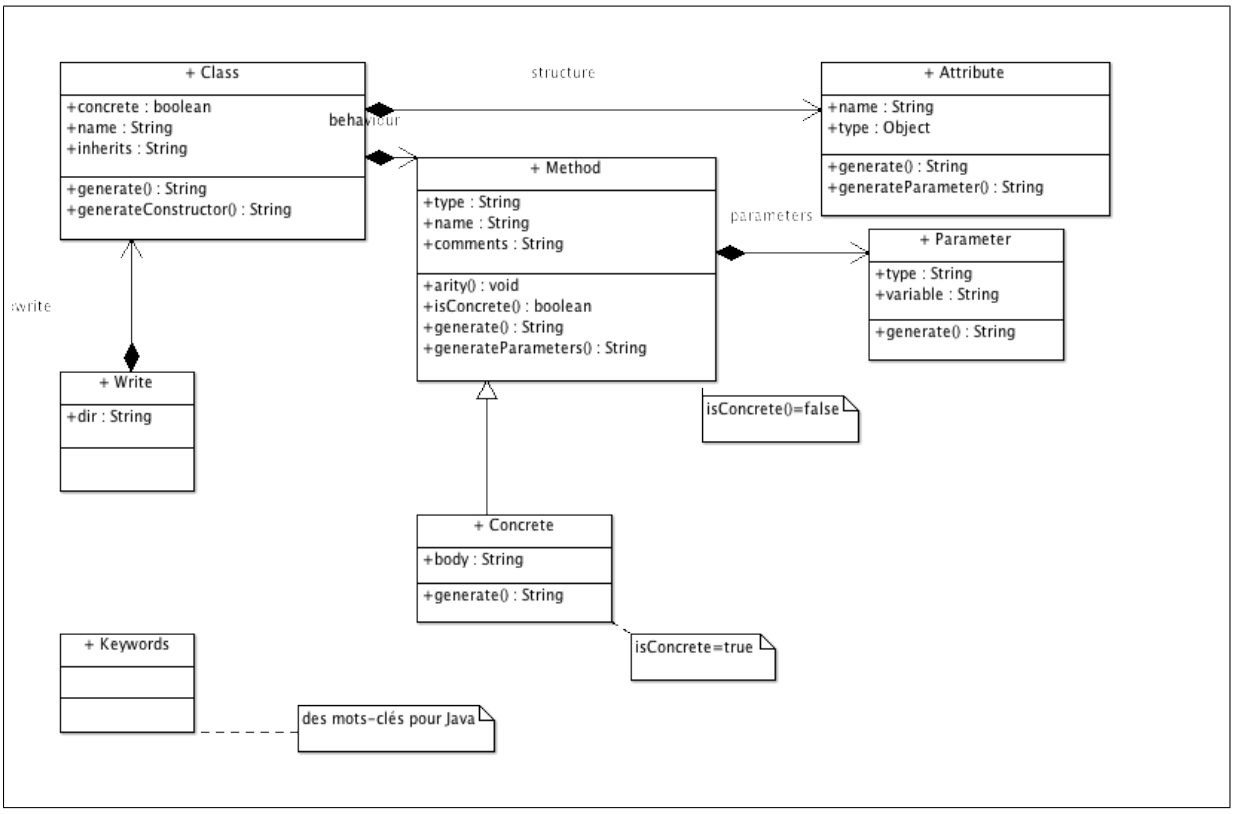

FIGURE 2 – Diagramme des classes

Dans le répertoire de test vous avez un programme `TestGeneration` celui-ci est important car vous allez devoir le compléter. Tout d'abord il faut comprendre ce qu'il contient et ceci se fait en lisant le diagramme de classe. Par exemple dans `TestGeneration` la première méthode de la classe `List` est définie par :

```
new Method(Keywords.BOOLEAN, "isEmpty", Method.NOPARAMETER, "teste si empty ou pas")
```

Le diagramme de classe (classe `Method`) permet de comprendre qu'il s'agit de la création d'une méthode retournant un boolean, sans paramètre, s'appelant `isEmpty` et avec un commentaire. La classe représentant les listes non vides est définie par :

```
Class notempty = new Class(composite, att, beh2, true, root);
```

C'est-à-dire (voir la définition de `Class.java` ou la javadoc), elle s'appelle "NotEmpty", elle a des attributs `att`, des méthodes (`beh2`), elle est concrète et hérite de la racine du patron (la classe `List`). Cette compréhension n'est pas immédiate, il faut lire attentivement le fichier `TestGeneration`, aller voir

dans la javadoc ou le code source pour effectivement s'assurer que la plupart les informations du patron composite sont là. Noter les informations manquantes car il faudra compléter la description du patron.

Votre plan de travail est le suivant :

1. Tester la génération de code et noter ce qu'il manque dans les fichiers qui sont générés dans `output`. En particulier si vous regardez les erreurs et utilisez les options proposées comme `add unimplemented methods` vous verrez vite ce qui manque. Des `TODO` sont placés dans le code pour vous aidez, mais il manque d'autres bouts de code.
2. Vous devez compléter `tests.TestGeneration` car il manque des méthodes et des attributs, rappelez vous que le code généré doit être le même que celui du patron.
3. Ensuite compléter la génération de code de façon à obtenir le même résultat que le patron.
   (a) Génération du code des attributs `Attribute.generate`. Vous devez générer la déclaration d'un attribut publique avec un commentaire.
   (b) Génération du code des constructeurs complets `Class.generateConstructor`. La forme d'un tel constructeur a été décrite avec un exemple précédemment.
4. Petit test d'implémentation : essayer d'importer sous `ArgoUML` votre résultat de génération et comparer visuellement le diagramme que vous obtenez avec l'initial ...
5. Étendre votre programme pour qu'au lieu de générer des listes de `int` il puisse générer des listes de `String`, de `char` ou de n'importe quel type dont la classe existe ou est connue de Java. Vous devez identifier les portions de code qui doivent variées en fonction de l'argument de type utilisé. Vérifier que la génération de liste pour les chaînes, caractères et entiers fonctionne avant de passer à la suite.
6. **Question subsidiaire :** modifier le programme de génération pour que les noms de classes et de fichiers fassent apparaître le paramètre. Par exemple nous suggérons d'utiliser les caractères `_` et `$`, ainsi vous devriez créer des classes `List_String$`, `Empty_String$` et `NotEmpty_String$`. Normalement si vous avez menez à bien ce travail vous devez pouvoir générer des listes de liste de caractères, ou liste de liste de liste de String.
7. Rendre un zip de votre projet avec les sources et la documentation.

3 Pour les curieux

Les listes sont connues pour être *générique* par rapport au type des éléments. Ceci est souvent noté `List<T>` ou `T` désigne une variable de type, *i.e.* prenant ses valeurs dans l'ensemble des types ({`int, char, String, double, ...` }). Java 1.5 introduit en fait une facilité de ce genre, voir par exemple `http://www.dil.univ-mrs.fr/~gispert/enseignement/coursJava/12Java15.html#typeGen`. En fait vous venez de résoudre une forme de généricité assez élémentaire dite par "template" et qui est utilisée par C++. Ce point sera plus développé en GS1 pour les chanceux ... Un programme (le préprocesseur) est chargé de réécrire le code des `List<T>` à partir de la définition des `List` et de la valeur de `T`. Puis ensuite le compilateur C++ fait son travail habituel mais sur ce nouveau code. Java 1.5 intègre une forme de généricité un peu différente dans son principe mais la syntaxe d'utilisation est très voisine de celle utilisée ici.

Du temps, une bibliothèque ou un explorateur web et les mots-clés :

- Généricité
- Compilation

– Programmation générative

vous permettrons de creuser cette question.