JavaBuilder

Date: 2009-10-21
Author: fdesbois (fdesbois@codelutin.com)

Contenu

Besoin/Contexte 1
Utilisation de l'ObjectModel existant 1
Mise en place du JavaBuilder 1
Principe des JavaFile 2
Autre idée pour les imports 2
Résumé 2
Pour aller plus loin 3
Travail à fournir 4
Annexe 4

Besoin/Contexte

A l'heure actuelle, les générations de code Java sont faites depuis EUGene en passant par des modeles UML (ArgoUML XMI 1.2 ou TopCased XMI 2.1). Il n'est pas possible de générer simplement un fichier java sans passer par le modèle UML. De plus, les templates de génération peuvent être fastidieuse lors de la création de fichier Java pour tout ce qui concerne les imports (cf ToPIA-persistence). Il a été initié l'idée de créer un autre générateur sans intelligence permettant de simplement généré du code Java sans contraintes particulières et de manière simple et intuitive.

Utilisation de l'ObjectModel existant

Il n'est pas nécessaire de définir un nouveau modèle spécifique au langage Java, le modèle ObjectModel d'EUGene contient les éléments nécessaires à la construction de n'importe quel fichier java. Cependant les méthodes 'setter' ne sont pas accessibles à partir des interfaces (pour éviter les modifications du modèle en cours de génération). Ainsi il doit être mis en place une classe Helper permettant de remplir l'ObjectModel directement en Java.

Ce Helper (ObjectModelBuilderHelper) permettra simplement de construire et de remplir les objets de l'ObjectModel (classes, interfaces, operations, ...) sans définir précisément le but de ce remplissage. Pour faire simple, il remplit simplement le modele sans se préoccuper de la future génération. Ainsi il reste complètement indépendant de la génération ou des futurs 'builders'.

ObjectModelBuilderHelper
addAttribute(classifier : ObjectModelClassifier,name : String,type : String,value : String,visibility : String,static : boolean,final : boolean) : ObjectModelAttribute createClass(name : String,packageName : String) : ObjectModelClass createAbstractClass(name : String,packageName : String) : ObjectModelClass createInterface(name : String,packageName : String) : ObjectModelInterface addOperation(classifier : ObjectModelClassifier,name : String,returnType : String,inputParameters : String...) : ObjectModelOperation setOperationBody(operation : ObjectModelOperation,body : String) : void

Mise en place du JavaBuilder

L'idée est de permettre aux utilisateurs voulant faire de la génération de ne pas être obliger de passer par les fichiers XML 'objectmodel'. Ainsi le développeur aura à sa charge de le faire manuellement.

Sur le même principe que les Generator, le développeur devra créer ses propres Builder pour remplir le modèle et ainsi utiliser un Generator simple qui ne souci pas de conversion et génère tel quel l'ObjectModel (également à créer). Une classe abstraite JavaBuilder doit être ainsi créer pour permettre son utilisation dans le processus de génération (à l'instar des Generator).

Le développeur va surcharger cette classe abstraite JavaBuilder pour remplir son modèle comme il le souhaite.

Note

Il sera possible d'utiliser des templates pour faciliter la mise en place du corps des operations (modification de processor nécessaire)

Le développeur devra ensuite configurer la génération (via maven bien entendu) pour ajouter ces builders dans le processus de génération. Ainsi les builders remplacent en quelque sorte la transformation du XMI en objectmodel.

Principe des JavaFile

Pour permettre l'utilisation du ImportsManager facilement, il sera nécessaire d'ajouter des méthodes dans le builder pour l'ajout des imports. En effet, l'objectmodel ne comprend pas la notion d'imports et les classes ne peuvent pas connaitre les imports nécessaire au futur fichier générer. Le développeur devra donc utiliser une autre classe intégrer au builder pour faciliter la gestion des imports : JavaFile.

En principe le développeur ne se préoccupera pas de construire un objectModel pour le remanipuler à l'aide des templates de générations comme dans ToPIA, il souhaitera directement créer des fichiers qui seront après automatiquement générés, avec les imports !

Le JavaBuilder contient donc l'ObjectModel global (contenant toutes les classes, ...) et une liste de JavaFile, contenant chacun :

- le classifier associé (classe, interface, enumeration, ...) qui sera l'élément principal du fichier : son nom et son package determinerons l'endroit ou sera généré le fichier et son futur nom.
- l'imports manager contenant les imports propre au fichier

Une idée plus simple sera peut-être d'utiliser une map contenant tous les importsManager des classifier contenu à la racine du modèle. Mais ne mélangeons pas tout (quoique), l'ObjectModel construit ne sera potentiellement pas censé servir à autre chose qu'une génération basique (donc avec un seul générateur stupide).

Note

Le développeur devra nommé directement les classes comme il souhaite les générer, les méthodes getFilenameFrom... ne semble pas avoir leur place dans ce type de génération.

Autre idée pour les imports

Pour éviter l'utilisation de JavaFile, il serait intéressant d'ajouter directement dans l'ObjectModel racine une map contenant les importsManager lié à chaque fichier à générer

```
Map<String, ImportsManager> importsManagers; // key = classifierQualifiedName, value = associated ImportsManager to the Classifier
```

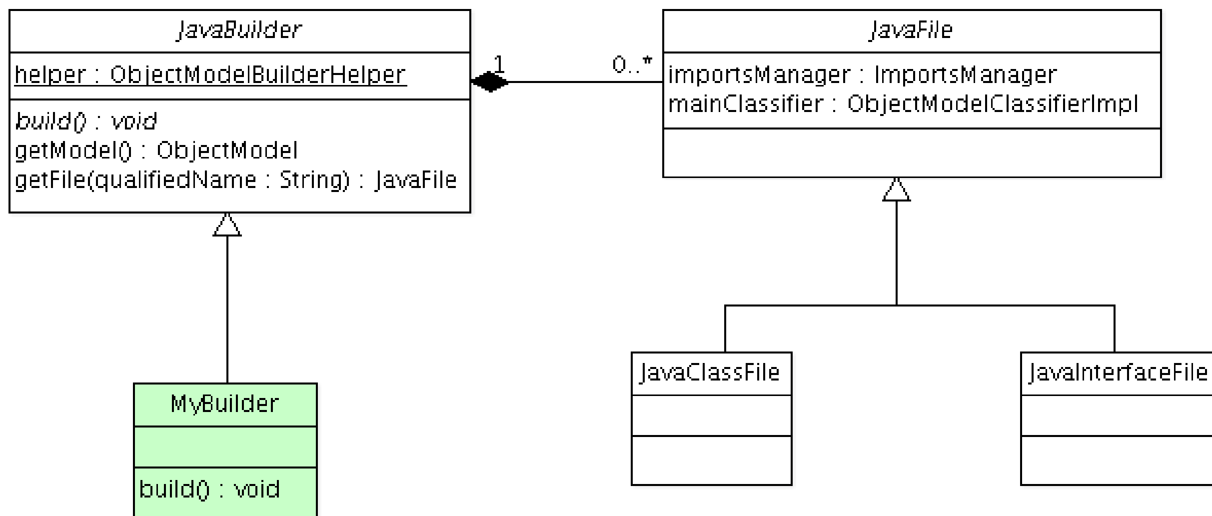
Libre ensuite au generator de prendre en compte ces manager ou non !

Note

Pour une meilleure prise en charge des enumerations, il semble nécessaire de les faires dériver d'un Classifier. A voir si cela peut poser des problèmes.

Résumé

Le développeur pourra créer plusieurs 'builder' héritant du JavaBuilder pour créer son modèle et donc les fichiers qu'il souhaitent générer basiquement.



Ex

```
public class MyBuilder extends JavaBuilder {

    @Override
    public void build() {
        ObjectModelClass clazz = helper.createClass("Personne", "org.chorem.bonzoms");
        helper.addAttribute(clazz, "nom", "String");
        helper.addAttribute(clazz, "prenom", "String");

        ObjectModelOperation setNom = helper.addOperation(clazz, "setNom", null, "String", "nom");
        helper.setOperationBody(setNom, "" /*{
             this.nom = nom;
        }*/);

        ObjectModelOperation getNom = helper.addOperation(clazz, "getNom", "String");
        helper.setOperationBody(getNom, "" /*{
             return this.nom;
        }*/);

        this.createJavaClassFile(clazz); // Creation du JavaFile et ajout à l'ObjectModel lié au Builder
    }

}
```

A noter qu'il sera intéressant de créer plusieurs Builder abstrait pour simplifier l'utilisation pour des cas spécifiques, ex : BeanBuilder qui pourrait contenir des méthodes ajoutant automatiquement les setter/getter et les property listener.

Pour aller plus loin

La question se pose : Où placer l'interpretation des builder (cad les builder.build()) dans le processus de génération ?

Tout dépend de comment sera manipulé l'ObjectModel résultant ! Il peut être imaginé également de passer en paramètre au JavaBuilder un ObjectModel existant (provenant d'un autre Builder ou de fichiers XML), qui se chargera de faire une manip comme créer les JavaFile, ajouter automatiquement les imports et permettre au développeur d'ajouter/modifier des éléments au modèle. Ceci dans un soucis de réutilisabilité des templates de génération existant. L'ObjectModel résultant du builder pourra être passé à un Generator existant (et pas celui stupide) pour ajouter des éléments.

En principe il sera possible de :

- créer un modèle UML/XMI qui sera généré en XML objectModel puis en ObjectModel mémoire
- utiliser un Builder pour rajouter des éléments à cet ObjectModel
- utiliser un Generator existant avec ses templates pour générer les classes finales (potentiels conflits ?)

Le cas simple sera :

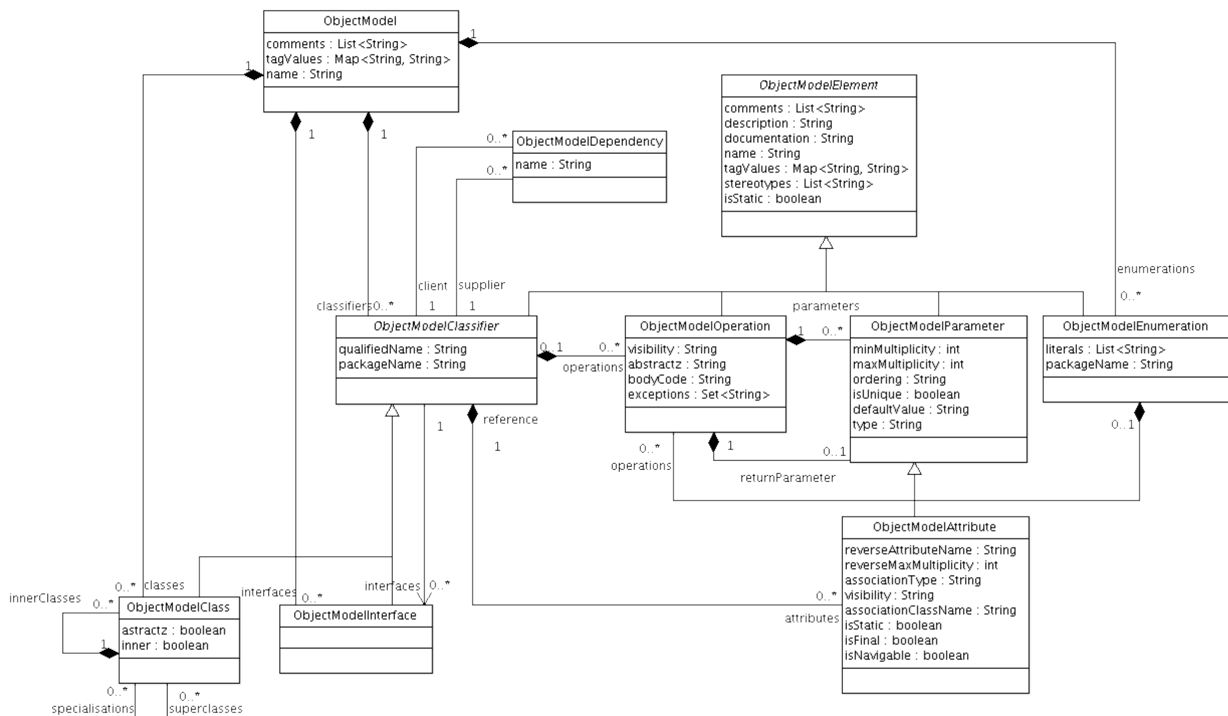
- utiliser un Builder pour créer et remplir un ObjectModel
- utiliser le Generator stupide pour générer le contenu de l'ObjectModel

A voir si d'autres cas de figures peuvent être intéressants.

Travail à fournir

- Créer un ObjectModelBuilderHelper exhaustif des possibilités de remplissage de l'ObjectModel (y compris les éléments UML comme les stéréotypes, etc...) ou peut être deux Helper : un simple spécifique au remplissage pour le générateur stupide et un plus complet pour le remplissage manuel de l'ObjectModel
- Déterminer les limites du passage de l'ObjectModel de builder en builder en generator en builder en ce qu'on veut !!?!
- Définir précisement les éléments à apporter au plugin maven pour prendre en compte les Builder
- Créer une template de génération basique pour faire une simple génération Java
- Fournir des Builder un peu plus intéressant pour faciliter leurs utilisations (BeanBuilder, DAOBuilder, ...) qui remplaceront a terme les templates existantes

Annexe



Metamodel ObjectModel