EUGene 2.0

Date: 2009-11-02
Author: fdesbois (fdesbois@codelutin.com)

Contenu

Avant-propos

Ce document regroupe les nouvelles évolutions apportés à EUGene pour sa version 2.0. Certaines évolutions pourront potentiellement changés lors de leurs développements. Suivre la RoadMap . Le développement est en cours sur la branche eugene-2.0 .

Contexte de départ

L'évolution majeure d'EUGene de la 1.0.1 à la 2.0.0 concerne une simplification de la génération de fichiers Java. Pour ce faire, plusieurs évolutions sont à prendre en compte :

- Modification du processus de génération (modification du point de lecture dans le processus)
- Permettre une transformation Model To Model
- Création d'une template de génération simple et sans intelligence (sans interprétation du modèle) pour génération uniquement Java

La version 2.0 modifie donc le processus sur trois niveaux différents : Lecture, Transformation et Génération.

Contraintes de non régression

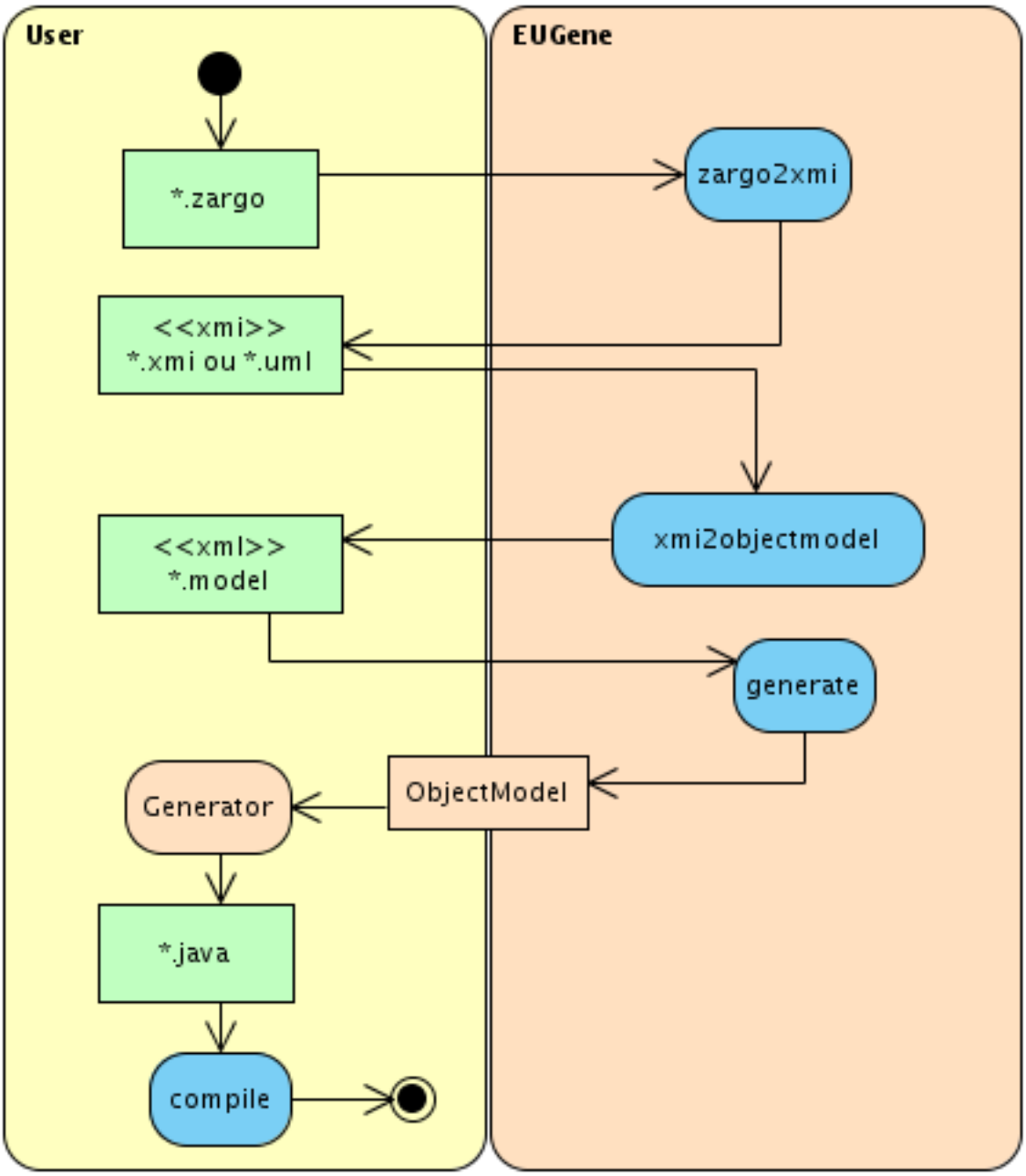
- La config du plugin maven ne doit quasimment pas changer : Les transformations seront considérés comme des templates de générations
- L'ObjectModel ne doit pas prendre en compte des spécificités Java (pas d'importsManager, pas d'attributs "synchronized", ...), le comportement doit rester identique (et indépendant du langage).
- Les templates de génération existantes doivent fonctionner toujours de la même manière : même résultat à la génération

Modification du processus de génération

Version 1.0.1

Dans la version 1.0.1, les possibilités de fichiers en entrée du processus sont assez limités. Il est possible d'utiliser différents goal du maven-eugene-plugin pour pouvoir avoir en entrée d'autres fichiers que ceux spécifiques à un des modèles supportés par EUGene (StateModel, ObjectModel, ...). Les goals existants sont : zargo2xmi, xmi2objectmodel et xmi2statemodel. Sur le diagramme ci-dessous est représenté une génération à partir de fichiers **.zargo** pour obtenir des fichiers **.java**. En beige sont représentés les points d'extension possible dans le processus de génération. Ces derniers sont très limités :

- **ObjectModel** : Modele représentant les données extraites à partir des fichiers sources (le modèle étend la classe << *Model* >>)
- **Generator** : Template de génération qui interprète le modèle pour effectuer une génération de fichiers en sorties, ici java. (la template étend la classe << *Generator* >>)



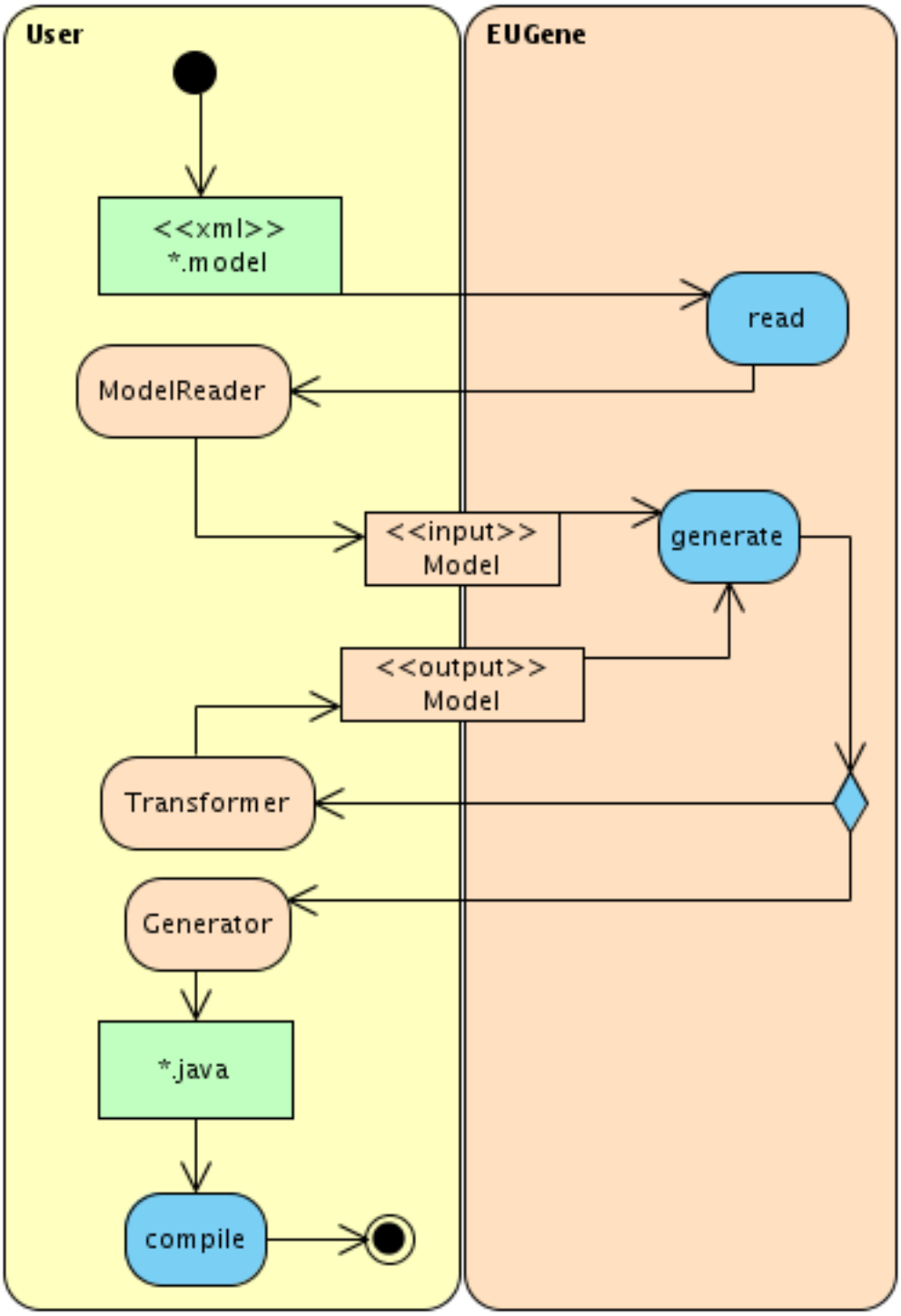
Processus de génération v1.0.1

Version 2.0

Deux nouveautés :

- Possibilité d'étendre le point de lecture en début de processus : Cela permettra ainsi d'avoir n'importe quel type de fichier en entrée, du moment qu'un Reader existe pour les interpréter et rendre un Model.
- Possibilité de transformer un modèle pour spécifier un langage (ex Java) ou tout autre transformation : Un Transformer est alors considéré comme un Generator pour simplifier le processus.

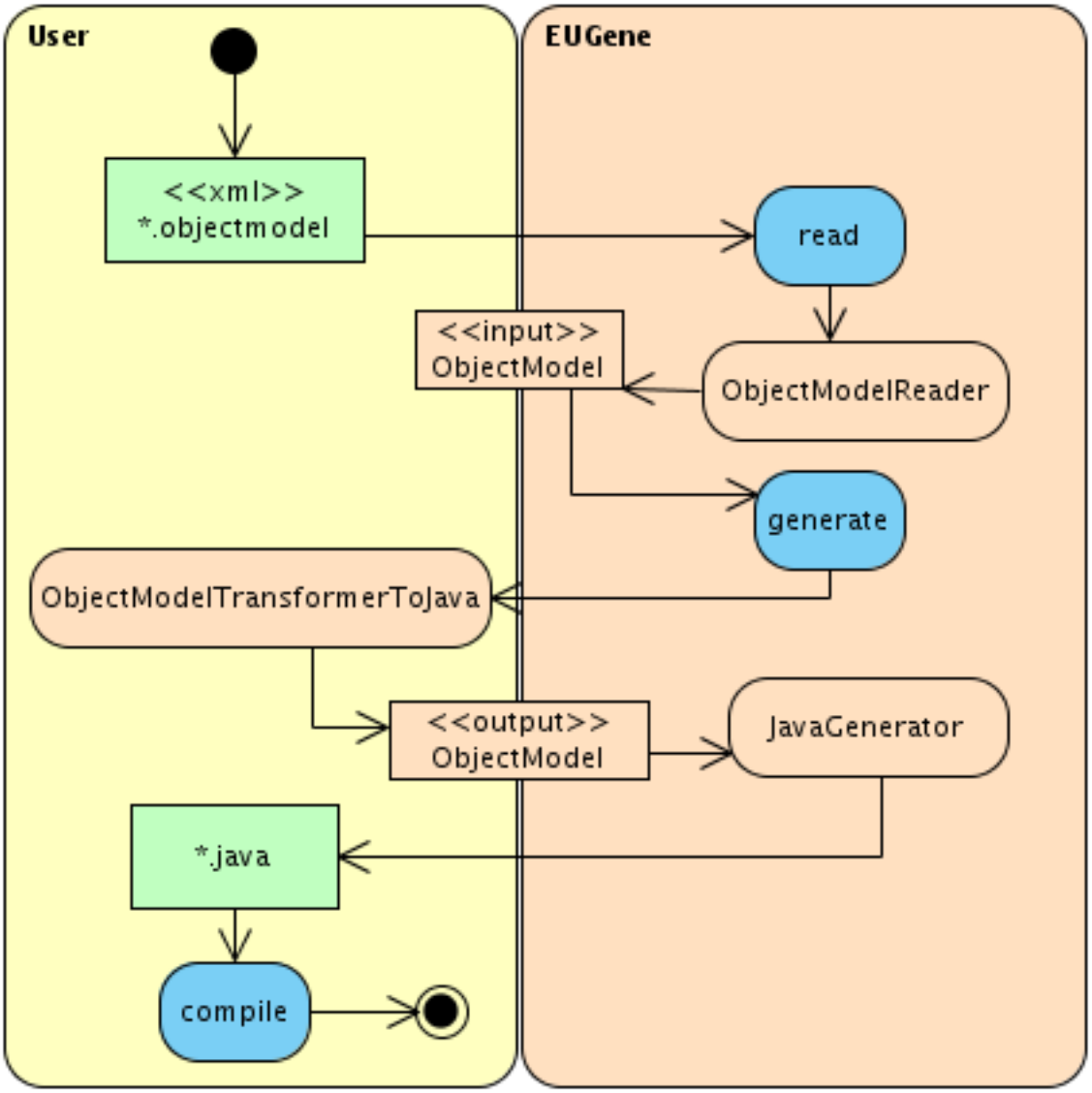
Ces deux nouveautés forment les nouveaux points d'extension du processus, à noter également la possibilité d'inclure dans le processus la transformation de n'importe quel modèle en un autre modèle.



Processus de génération v2.0

Génération spécifique à Java

Voici le cas spécifique à la génération Java. Nous voulons en entrée lire des fichiers Objectmodel (utilisation d'un ObjectModelReader), puis transformer ce modèle complet (avec notions UML) en un nouveau ObjectModel simple uniquement dédié à la génération Java (utilisation d'un ObjectModelTransformerToJava associé à la template JavaGenerator).



Processus de génération v2.0 pour Java

Le développeur se chargera d'intepréter le modèle d'entrée (lu par l'ObjectModelReader) pour le transformer en ObjectModel Java : extension de l'ObjectModelTransformerToJava. Ce transformer sera une sorte d'équivalent à un template de génération qui étend Generator.

Liste des Evolutions

1 - Evol #112 : Ajout d'un point d'entrée dans le processus de génération : ModelReader

2 - Evol #114 : Ajout d'extensions à l'ObjectModel

3 - Evol #115 : Remplissage de l'ObjectModel directement a partir du code : Builder

4 - Evol #113 : Possibilité de faire du Model To Model : Transformer

5 - Evol #116 : Modification hiérarchie des Generator pour prise en charge des Transformer

6 - Evol #117 : Création d'une template de génération sans intelligence pour Java : JavaGenerator

7 - Evol #107 : Remplacer les attributs (static, abstract, visibility, ...) par une liste de "modifier"

Evolution #112 : ModelReader

Besoin/Contexte

Les types de fichier en entrée du processus de génération sont limités : .xmi, .objectmodel, .statemodel, .zargo ou .uml sont les différentes possibilités. Il serait intéressant de pouvoir modifier le point d'entrée pour pouvoir par exemple gérer un autre type de fichier (un autre fichier xml que du objectmodel par exemple). De plus cela permettra de simplifier les goal maven en créant des Reader en entrée pour chaque type de fichier supporté (zargo, xmi, uml, ...).

Mise en place

Extraction de la lecture des fichiers d'entrée depuis les Generator. Un Reader est créé pour prendre en entrée une liste de fichiers et obtenir en sortie un Model qui sera par la suite interprété par un Generator. Un Reader par type de model supporté dans EUGene sera créé : ObjectModelReader et StateModelReader.

Extension

Il est possible d'étendre un Reader pour pouvoir gérer soit un autre format d'entrée de fichiers, soit un autre modèle de sortie (ou potentiellement les deux).

- extension d'ObjectModelReader pour gérer d'autres fichiers d'entrée (xmi, zargo, uml, ...)
- extension du Reader<M extends Model> pour gérer un autre modèle de sortie.

Evolution #114 : Extensions ObjectModel

Besoin/Contexte

La génération Java nécessite une manipulation simple pour l'ajout des imports qui seront rajoutés en début de fichier. Il existe dans la 1.0.1, la classe ImportsManager qui remplit cette tâche : Elle permet de lui ajouter différents imports à effectuer sous forme de String ou de Class. Ces imports sont ensuite récupérer suivant un nom de package. Pour simplifier la génération Java, il suffirait d'ajouter les ImportsManager (un par classifier) directement dans l'ObjectModel racine. Cependant, pour ne pas ajouter de notion spécifique à Java, cet ajout sera considéré comme une extension à l'ObjectModel.

Mise en place

Une *Map<String, Object> extensions* est ajoutée dans l'ObjectModel racine. La méthode getExtension(String reference, Class extensionClass) permettra de récupérer l'extension souhaité (avec son type) et sera disponible dans l'interface ObjectModel. Cette méthode créera automatiquement l'extension si elle n'est pas encore définie dans l'ObjectModel.

Cas des ImportsManager

Pour les ImportsManager, il est nécessaire d'avoir une Map<String, ImportsManager>, la clé étant le nom complet (full qualified name) d'un classifier, et la valeur l'ImportsManager associé à ce classifier. Pour simplifier l'utilisation des ImportsManager, une classe ImportsManagerExtension est créé comprenant la map des managers et des méthodes accesseurs :

- **ImportsManager getManager(ObjectModelClassifier classifier)** : Permet de récupérer (ou créer s'il n'existe pas) l'ImportsManager lié à un classifier. Méthode utilisée pour les transformations du modèle (ObjectModelTransformer, JavaBuilder)
- **List<String> getImports(ObjectModelClassifier classifier)** : Permet de récupérer tous les imports liés à un classifier. Méthode utilisée pour la génération (ObjectModelGenerator, JavaGenerator)

L'ImportsManagerExtension forme donc l'extension spécifique aux imports ajouté à l'ObjectModel:

```
// Utilisation  dans Transformer ou Builder
ImportsManagerExtension managers = model.getExtension(ImportsManagerExtension.OBJECTMODEL_EXTENSION, ImportsManagerExtension.class);
ImportsManager maClassManager = managers.getImportsManager(maClass);
maClassManager.addImport("java.util.List");

// Utilisation dans Generator
ImportsManagerExtension managers = model.getExtension(ImportsManagerExtension.OBJECTMODEL_EXTENSION, ImportsManagerExtension.class);
managers.getImports(maClass);
```

Extension

Il est désormais possible d'utiliser les extensions comme moyen d'enrichir l'ObjectModel lors des transformations et des générations.

Evolution #115 : Builder

Besoin/Contexte

Il est intéressant de pouvoir, dans certains cas, remplir un ObjectModel vide. L'utilisation se fait directement à partir du code d'une autre application qui souhaite utiliser l'ObjectModel ou à partir d'un Reader pour remplir l'ObjectModel. Cependant la version 1.0.1 ne permet pas de faire cela car les interfaces des classes de l'ObjectModel ne comprennent pas de setter. Il est donc nécessaire de créer une nouvelle classe permettant de remplir un ObjectModel vide avec potentiellement une finesse d'interprétation spécifique à Java (gestion des imports, doublon sur les méthodes, ...).

Mise en place

Création d'une classe ObjectModelBuilder qui initialise à l'instanciation un ObjectModel vide (A noter que le nom du modèle est indispensable à la génération). Ce builder permet de remplir complètement l'ObjectModel (stéréotypes, multiplicités, ...)

Création d'une classe JavaBuilder qui initialise à l'instanciation un ObjectModel vide. Le JavaBuilder propose un panel de méthodes permettant :

- Ajout de classes/interfaces au modèle
- Ajout d'une superclass
- Ajout d'interface à une classe
- Ajout de méthodes à une classe/interface
- Ajout du corps des méthodes
- Ajout de paramètres à une méthode
- Ajout d'exception à une méthode
- Ajout d'attributs à une classe/interface
- ...

Ces méthodes comprendront une gestion automatique des imports (types des paramètres/attributs, retour des méthodes, exceptions, ...).

Evolution #113 : Transformer

Besoin/Contexte

Dans le cas d'une génération java, l'écriture des templates de génération actuelles (v1.0.1) peut être fastidieuse et difficile à maintenir. Il est donc préférable d'utiliser une tranformation du modèle ObjectModel source (provenant d'un diagramme UML) en un ObjectModel représentant basiquement le Java qui sera généré. Ce qui implique pour ce cas, la suppression des éléments spécifiques à UML (mutliplicités, stéréotypes, ...) pour un ObjectModel épuré spécifique à une génération Java.

Mise en place

Un Transformer est considéré comme un *Generator* pour être intégré plus facilement dans le processus de génération (templates). Cependant il possède un modèle en entrée et un en sortie, donc **aucune sortie fichiers**. Pour ce faire, il est obligatoirement associé à un *Generator*. Ainsi un *Transformer* prendra en entrée un modèle, le transformera en un nouveau (potentiellement de type différent) et utilisera le generator de sortie pour générer ce nouveau modèle. Il est donc nécessaire d'instancier un *Transformer* en lui fournissant son *Generator* de sortie compatible avec le modèle de sortie.

Conversion existant

Pour une génération Java, il est nécessaire d'étendre l'ObjectModelTransformerToJava qui utilise un JavaBuilder pour la construction d'un ObjectModel spécifique à Java.

Ex : BeanGenerator permet la génération de bean (stéréotype <<bean>>) en leurs ajoutants les getter/setter adéquat et des listeners. Cette template de génération existe dans ToPIA. Elle sera remplacée par un BeanTransformer qui étend ObjectModelTransformerToJava. Au lieu d'écrire la template du fichier généré, le transformer remplira un nouveau modèle avec les classes, opérations, attributs à générer.

Note

Les noms complets des éléments ajoutés au modèle devront être impérativement utilisés pour la gestion des imports. (ex : java.io.Serializable, org.chorem.bonzoms.Role, java.util.List<java.lang.String>, ...)

Extension

Les Transformer sont étendables pour transformer un modèle en un autre comme on le souhaite. Plusieurs transformer sont disponibles dans EUGene :

- abstract Transformer<Input extends Model, Output extends Model>
- abstract ObjectModelTransformer<Output extends Model> extends Transformer<ObjectModel, Output>
- abstract ObjectModelTransformerToJava extends ObjectModelTransformer<ObjectModel>

Il est nécessaire d'étendre l'un de ces Transformer pour créer une transformation spécifique à son modèle métier.

Evolution #116 : Generator

Besoin/Contexte

Des modifications sont nécessaires sur les Generator racines (abstraits) pour permettre les autres évolutions. Les générateurs de la version 1.0.1, ne permettent pas la modification du modèle d'entrée. De plus les générateurs sont spécifiques à un type de fichier d'entrée (.objectmodel pour ObjectModelGenerator, .statemodel pour StateModelGenerator).

Mise en place

Extraction de la lecture en entrée des générateurs. Les générateurs ne prennent désormais qu'un modèle en entrée (qui peut potentiellement provenir de plusieurs fichiers sources à la charge du Reader). Le type du modèle est également connu à l'instanciation : *ObjectModelGenerator extends AbstractGenerator<ObjectModel>*. Un générateur est donc obligatoirement associé à un Model d'entrée. Voir diagramme en annexe pour plus de précision sur les modifications d'héritage.

Extension

Un Generator est associé à un modèle. La création d'un nouveau modèle implique la création d'un Generator abstrait associé.

- super parent : Generator<M extends Model>
- classe abstraite commune aux templates actuelles : AbstractGenerator<M extends Model> extends Generator<M>
- cas du ObjectModel en entrée : ObjectModelGenerator extends AbstractGenerator<ObjectModel>
- idem pour un autre modèle MonModel : MonModelGenerator extends AbstractGenerator<MonModel>

Evolution #117 : JavaGenerator

Besoin/Contexte

Les templates de génération (Generator) doivent être écrites pour générer des fichiers. Il est possible grâce à l'ajout de Transformer de pouvoir désormais remplir un modèle spécifique au langage Java (utilisation d'ObjectModel). Cependant il n'existe aucune template de base permettant d'interpréter un ObjectModel simplement sans prise en compte de spécificités du au métier de l'application (entités, daos, ...) qui seront interprétés dans les Transformer.

Mise en place

Création d'une template JavaGenerator toute simple qui parcourt l'ObjectModel, génère les classes, méthodes interfaces sans prise en compte des spécificités UML (stéréotypes, multiplicités, ...).

Note

JavaGenerator extends ObjectModelGenerator

Evolution #107 : ObjectModelModifier

Besoin/Contexte

En java, les modifier concernent certains paramètres pouvant être mis sur les attributs, méthodes comme "static", "abstract", "public", ... Dans l'ObjectModel de la version 1.0.1, ces modifier sont représentés par des attributs ce qui implique un attribut par modifier : static, visibility, ... Pour simplifier la génération et la manipulation de ces modifiers, il serait intéressant d'avoir simplement une liste d'ObjectModelModifier. De plus cela permettra d'ajouter plus facilement certains modifier comme "synchronized". Cela permettra également de simplifier l'écriture de certaines méthodes des builder

```
public ObjectModelOperation addOperation(String operationName, String returnType,
        ObjectModelModifier... modifiers) {
        ...
}
```

exemple

```
ObjectModelOperation setter = addOperation("setName", "void", ObjectModelModifier.PUBLIC);
// correspond a : public void setName(...)
```

Contraintes

- Les méthodes isStatic, isAbstract, etc... seront maintenus pour connaitre précisemment les modifier associés à la classe, méthode, attribut ou autre.
- Les nouveaux modifiers auront un nom générique indépendant du langage.

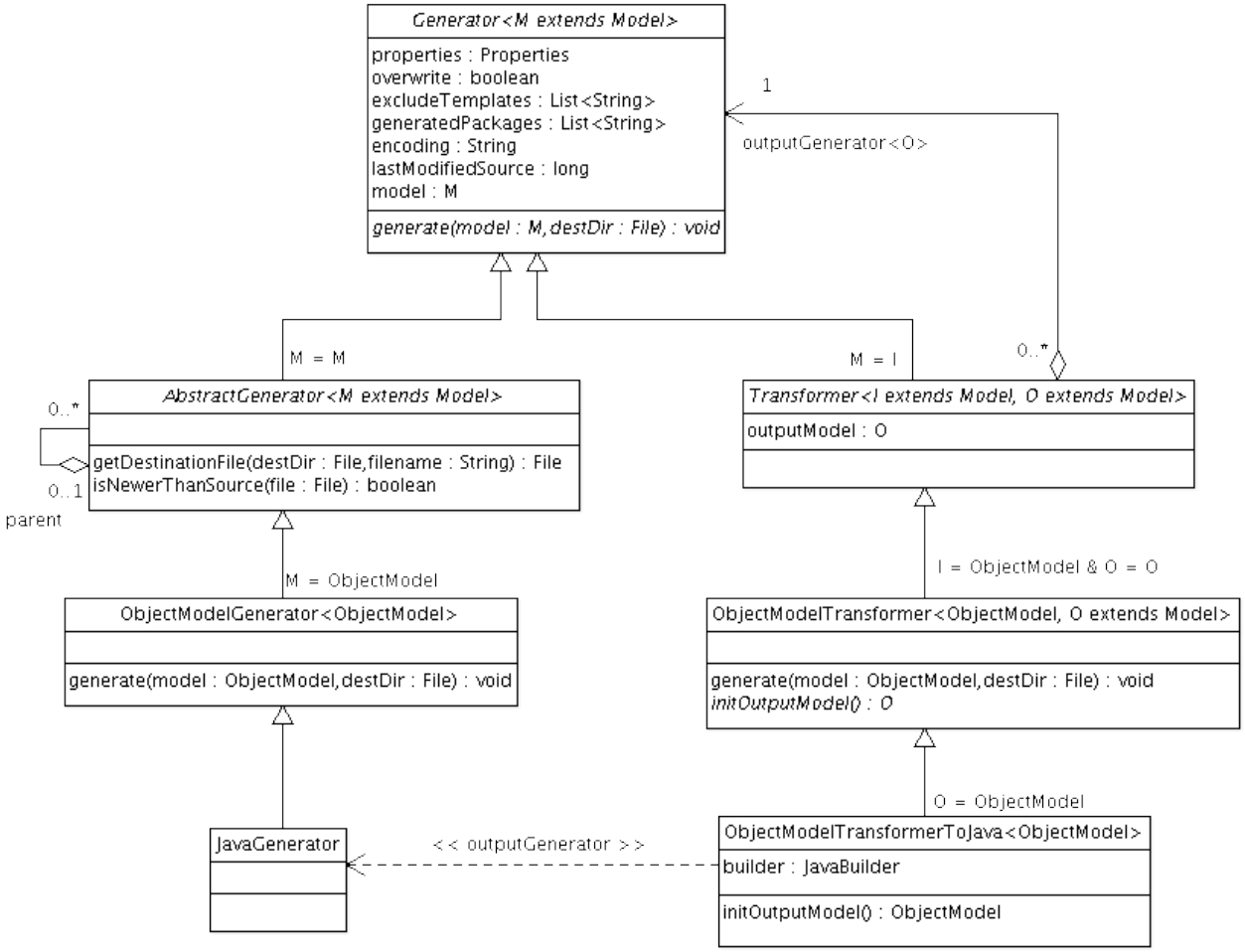
Mise en place

Création d'une enumération ObjectModelModifier comprenant les différents modifiers existants (STATIC, ABSTRACT, PUBLIC, PRIVATE, ...). Il sera possible de récupérer la valeur chaîné (String) de chaque ObjectModelModifier possible.

Conversion existant

Les attributs correspondant aux modifiers seront supprimés pour mettre en place une List<ObjectModelModifier> sur les objets qui le nécessitent (classifier, classe, interface, attribute, operation, ...).

Annexe : Hiérarchie des Generator/Transformer



Hiérarchie des Generator/Transformer