TP2: Manipulation de modèles

Notes

À partir de ce TP, vous allez devoir manipuler les modèles de façon programmatique. Cela va forcément impliquer l'utilisation d'un langage de programmation pour lequel il existe le support de librairies de méta-modélisation. Vous avez le choix entre Java et Python.

- Si vous décidez d'utiliser Java, vous devez travailler sous Eclipse, vous ne pouvez pas utiliser Intellij, les outils EMFs ne sont disponibles que sous Eclipse.
- Si vous décidez d'utiliser Python, vous pouvez utiliser l'éditeur de code que vous préférez, mais il est indispensable d'utiliser un env. virtuel pour installer les dépendances sans altérer votre système.

Comme ce TP est une prise en main des différentes librairies, des détails seront donnés pour les deux langages et les différents outils. Par contre, il est important que : * soit vous décidiez de tester les deux librairies au fil des TPs et alors, d'être bien attentif et de faire attention à la manière de manipuler les choses, * soit vous décidiez de rester avec un seul langage/lib pour éviter de multiplier la complexité, dans ce cas, jetez quand-même un oeil à comment les choses sont manipulées chez un de vos camarades pour bien voir les similitudes et différences techniques.

À vérifier : à plusieurs étapes, du code va être généré, soit en Java, soit en Python. Faîtes bien attention au nommage de vos méta-classes dans votre méta-modèle. Si vous utilisez des noms réservés ou des noms spéciaux, il est possible que cela pose problème une fois que le code sera généré. Spécialement, faites attention aux noms de méta-classes/attributs/references qui porteraient des nom comme `class` (avec une minuscule), `super` (qui clasherait en Python), `return` (qui pourrait poser problème en Java et Python),...etc Tout ces noms ne vont pas poser systématiquement problème, mais si vous vous retrouvez avec un modèle qui ne se charge pas pour une raison obscure, essayez de vérifier qu'il n'y a pas de soucis de nommage.

Aussi, prêtez une attention toute particulière à l'`URI` de vos méta-modèles. Si vous avez un modèle qui ne se charge pas bien et que vous ne comprenez pas pourquoi, vérifiez que votre modèle au format `.xmi` fait bien référence au même `prefix` et `URI` que ceux écrit pour votre méta-modèle (pour votre `EPackage` racine).

Manipulation d'instances

Vous allez manipuler ici les méta-modèles que vous avez définis au TP précédent, c'est-à-dire `simplejava`, `filesystem` et votre méta-modèle de base de donnée. Pour pouvoir manipuler des modèles, il faut être en mesure de charger le méta-modèle dont ils sont instances depuis un fichier `.ecore`. Pour charger un fichier, EMF/PyEcore passent par un système de `Resource`. Une `Resource` représente une entité logique qui contient un modèle ou un méta-modèle depuis une source en particulier (ici depuis un fichier). Ces resources sont conservées dans un `ResourceSet`, qui représente un conteneur de resources. Le `ResourceSet` fait plus qu'uniquement conserver des resources par ce qu'il va aussi permettre de lier des factories spéciales pour lire des formats de fichiers spécifiques et pour enregistrer des méta-modèles existants. Enregistrer un méta-modèle

dans un ResourceSet signifie que ce ResourceSet est en mesure de lire des instances de ce méta-modèle.

Directement, à partir de code généré

La génération du code d'un méta-modèle repose sur le fait de générer le code Java ou Python des méta-classes décrites dans le méta-modèle. Comme les librairies de méta-modélisation sont légèrement différentes, la génération de code ne se passe pas de la même manière.

En Java avec EMF

La génération de code passe par un `.genmodel` qui est un fichier de configuration de la génération de code. Pour le méta-modèle simple java, le `.genmodel` devrait être déjà existant, vous pouvez donc le réutiliser.

1. Ouvrez le fichier `.genmodel` et faites clic-droit sur la racine du fichier ouvert, puis `Generate Model Code`

Si vous n'avez pas de `.genmodel` pour votre méta-modèle

1. Faites un clic-droit sur votre `.ecore`, puis `new→Other→EMF Generator Model`, suivez les étapes pour générer le `.genmodel`
2. Ouvrez ensuite le fichier généré et faites clic-droit sur la racine du fichier ouvert, puis `Generate Model Code`

À l'issue de ces étapes, vous devriez avoir le code de votre méta-modèle entièrement généré. Maintenant, pour pouvoir ouvrir des modèles existant, il est nécessaire d'enregistrer une dépendance en plus pour votre projet "Ecore Modeling Project".

1. Ouvrez le fichier `META-INF/MANIFEST.MF`, puis switchez sur l'onglet `Dependencies`
2. À partir de cet onglet, cliquez sur le `Add···` de la frame de gauche `Required plug-ins` et ajoutez le plugin suivant :
    - `org.eclipse.emf.ecore.xmi`

Vous pouvez maintenant charger un modèle instance du méta-modèle enregistré comme vue précédemment en utilisant le `ResourceSet` correctement paramétré.

le méta-modèle que vous voulez utiliser dans un `ResourceSet`. Cela se fait avec le code ci-dessous. Notez bien que ce snippet créer en premier lieux un `ResourceSet` et le configure ensuite en enregistrant les méta-modèles dont il aura besoin pour lire votre modèle. Pour chaque nouveau méta-modèle dont vous générez le code, vous devez aussi l'enregistrer dans le `ResourceSet`.

```
ResourceSet rset = new ResourceSetImpl();
rset.getResourceFactoryRegistry().getExtensionToFactoryMap().put("ecore", new
XMIResourceFactoryImpl());
rset.getResourceFactoryRegistry().getExtensionToFactoryMap().put("xmi", new
XMIResourceFactoryImpl());
rset.getPackageRegistry().put(SimplejavaPackage.eNS_URI, SimplejavaPackage.eINSTANCE);

Resource resource = rset.getResource(URI.createFileURI("path_vers_un_modele"), true);
JavaModel model = (JavaModel) resource.getContents().get(0);
// JavaModel est le nom du concept racine du méta-modèle simple java,
// chez vous il peut avoir un nom différent ! Pensez bien à le modifier !
```

En Python avec PyEcore

La génération de code avec PyEcore passe par `pyecoregen` un outil dédié utilisant PyEcore pour générer le code Python des concepts du méta-modèle. Dans un premier temps, créez un env. virtuel :

1. placez-vous dans votre répertoire de travail (celui de votre repository) et tapez la commande suivante : `$ virtualenv venv`. Cela va créer un environnement virtuel dans le répertoire `venv` que vous pourrez activez quand vous voulez. Si la commande `virtualenv` n'est pas connue directement, utilisez à la place `$ python3 -m venv venv`, cela va créer un environnement virtuel `venv`, exactement comme la première commande.
2. Avant d'activer votre environnement virtuel, ajoutez le répertoire `venv` à votre `.gitignore` pour ne pas le commiter
3. Activez votre environnement virtuel de la manière suivante : `source venv/bin/activate`. **Attention** Si vous êtes sous windows, pour activer l'environnement virtuel, il est possible que la commande soit différente et qu'il faille aller chercher le `venv\bin\activate.bat`, si vous utilisez le sous-système linux sous windows, ça devrait être la même chose.

Vous pouvez maintenant installer les dépendances qui vous seront utiles :

1. Installez `pyecoregen`, dans votre virtualenv, tapez `pip install pyecoregen`
2. Générez le code de votre méta-modèle, ceci se fait avec la commande `pyecoregen -e chemin_vers_ecore -o .` (il s'agit bien d'un `.` en fin de ligne). Cette commande génère le code de votre méta-modèle dans le répertoire courant.

Maintenant que le code de votre méta-modèle est généré, il est possible de l'utiliser et d'ouvrir un modèle existant : Dans un premier temps, enregistrez le métamodèle dans un `ResourceSet` comme ceci :

```
from pyecore.resources import ResourceSet
import simplejava  # pour le méta-modèle simplejava

rset = ResourceSet()
rset.metamodel_registry[simplejava.nsURI] = simplejava  # enregistre le méta-modèle
simplejava
# si vous générez le code d'autres méta-modèles, n'oubliez pas de les enregistrer
aussi
resource = rset.get_resource('path_vers_un_modele_simplejava')
racine = resource.contents[0]
```

Questions générales

Voilà quelques questions relatives à simple java :

1. Écrivez une fonction qui prend en paramètre l'équivalent d'un `JavaModel` et qui retourne la liste de toutes les classes d'un de vos modèles `simplejava` (toutes les classes qui ont été modélisées dans le modèle, donc toute les instances de `JClass` si vous les avez nommé comme ça). Testez votre fonction avec plusieurs modèles. Si vous êtes certains qu'un de vos collègues à exactement le même méta-modèle que vous (mêmes nommage, relations, URI, prefix...etc), vous pouvez échanger des modèles pour vérifier que tout fonctionne correctement.
2. Écrivez une fonction qui affiche à l'écran le nom de tous les attributs (equivalent `JAttribute` contenues par les `JClass`) de votre modèle java.
3. Écrivez une fonction qui retourne la profondeur d'héritage d'une `JClass` classe en particulier (par exemple, en considérant que l'on a une instance de class `JObject` nommé `Object` dans notre méta-modèle, 0 sera retourné pour cette instance de `JClass`, 1 pour les classes qui héritent de `Object`, 2 pour les classes qui héritent des précédentes et ainsi de suite)
4. Écrivez une fonction qui retourne le nom pleinement qualifié d'une instance de `JClass` (le nom de tous ses packages contenant séparés par un `.` suivi du nom de la classe)

Voilà quelques questions relatives à filesystem :

1. Écrivez une fonction qui retourne le nombre de fichiers de votre modèle
2. Écrivez une fonction qui retourne le poids de tous les fichiers contenus dans un répertoire
3. Écrivez une fonction qui retourne le poids de tous les fichiers contenus dans le modèle

Et quelques questions pour la manipulation directe de vos méta-modèles :

1. Écrivez une fonction qui prend un `EPackage` en paramètre et retourne le nombre de méta-classes d'un de vos méta-modèles
2. Écrivez une fonction qui prend une `EClass` en paramètre et retourne les méta-classes qui héritent d'elle dans votre méta-modèle.

Réflexivement, sans code généré

En Java avec EMF

La première étape pour pouvoir créer des modèles relatifs à un méta-modèle, il vous faut charger en mémoire un méta-modèle existant à partir d'un fichier `.ecore`. Voilà le fragment de code que vous pouvez utiliser pour charger un modèle ecore/xmi directement en Java :

```
ResourceSet rset = new ResourceSetImpl();
rset.getResourceFactoryRegistry().getExtensionToFactoryMap()*
                                 .put("ecore", new XMIResourceFactoryImpl());
rset.getResourceFactoryRegistry().getExtensionToFactoryMap()*
                                 .put("xmi", new XMIResourceFactoryImpl());
Resource resource = rset.getResource(URI.createFileURI("path_vers_ecore"), true);

// À partir de là "pack" contient le EPackage de votre méta-modèle
EPackage pack = (EPackage) resource.getContents().get(0);

// Puis pour l'enregistrer
rset.getPackageRegistry().put(pack.getNsURI(), pack);
```

Contrairement au snipet Java précédent, vous pouvez noter que cette fois, le code du méta-modèle n'est pas généré, on vient charger le méta-modèle à partir d'un fichier (on le charge en mémoire) et qu'ensuite, on enregistre cette représentation mémoire dans le `ResourceSet`. Vous pouvez maintenant charger un modèle instance du méta-modèle enregistré comme vue précédemment en utilisant ce `ResourceSet` correctement paramétré.

En Python avec PyEcore

L'ouverture d'un modèle `.ecore` via PyEcore se fait de plusieurs façons, mais la façon la plus simple est d'utiliser un `ResourceSet` :

```
from pyecore.resources import ResourceSet

rset = ResourceSet()
resource = rset.get_resource('path_vers_ecore')
pack = resource.contents[0]
rset.metamodel_registry[pack.nsURI] = pack
```

Contrairement au snipet Python précédent, vous pouvez noter que cette fois, le code du méta-modèle n'est pas généré, on vient charger le méta-modèle à partir d'un fichier (on le charge en mémoire) et qu'ensuite, on enregistre cette représentation mémoire dans le `ResourceSet`. Vous pouvez maintenant charger un modèle instance du méta-modèle enregistré comme vue précédemment en utilisant ce `ResourceSet` correctement paramétré.

Création, modification de modèles

Une fois que votre méta-modèle est chargé en mémoire à partir de son `.ecore`, vous allez l'utiliser pour créer des petits modèles dans les questions suivantes.

1. Écrivez une fonction qui prend en paramètre le nom d'une des méta-classes de votre méta-modèle et qui retourne la méta-classe associée ou `null` si la méta-classe n'existe pas.
2. Écrivez une fonction qui permet de créer une instance d'un méta-classe dont le `EPackage` et le nom de la méta-classe serait passé en paramètre.
3. Écrivez une fonction qui prend un `EObject` quelconque, une chaine de caractère et qui initialise l'équivalent du nom de l'`EObject` à la valeur passée en paramètre si l'objet en question possède un attribut `name`.
4. Testez votre solution avec des instances de votre méta-modèle java, votre méta-modèle de système de fichier et votre méta-modèle de BD. Pour faire ceci :
    a. chargez les trois `EPackage` de chaque méta-modèles depuis leur `.ecore` dans des variables `sjava`, `fsystem` et `bd` en utilisant le code fournit plus haut relatif à votre langage (celui pour charger un métamodèle depuis un `.ecore`).
    b. créez une instance d'un des concepts de vos métamodèles en utilisant votre fonction de création à partir d'un nom de méta-classe
    c. initialisez le nom des instances que vous avez créé avec une valeur
    d. recherchez reflexivement la valeur du `name` de votre objet pour vérifier que la valeur a bien été initialisé pour l'attribut.

Autre utilisation de la couche réflexive

Dans cette partie du TP, vous allez utilisez plus en détail la couche réflexive pour commencer à prendre plus d'aisance avec la manipulation du méta-niveau.

1. Écrivez une fonction qui prend un `EObject` quelconque et qui affiche à l'écran le nom de chacun de ses attributs/références ainsi que la valeur de chacun des attributs/références.
2. Écrivez une fonction qui recherche dans un objet le premier attribut de type chaîne de caractère (type `EString`) et qui le modifie pour que le nom devienne `id_VALEURAVANT` (ex : pour `egg` stocké dans l'attribut `nom`, la nouvelle chaîne attribué à `nom` sera `id_egg`).