TP5: Chaînes de transformations et transformations

Note et pré-requis

Le but de ce TP va être de produire au moins deux petites transformations de modèles, basées sur les méta-modèles que vous avez pu développer ainsi qu'un moteur de chaînes de transformations MDE-based.

Ce TP, un peu plus long que les autres mais est plus "guidé". Il est à faire en utilisant Python, pour des raisons techniques. Les moteurs de transformations de modèles utilisant les technologies EMF-Java (ex: ATL, QVTo, Epsilon, Acceleo), sont bien établies et fonctionnent très bien lorsqu'on les utilise de manière localisées, mais leur usage est parfois très lourd : configurations complexes, interfaçage avec divers moteurs complexes, chaînage de transformations complexes à mettre en oeuvre et usage en dehors d'Eclipse vraiment compliqué et pénible.

Pré-requis

Si vous n'avez pas travaillé en Python en premier lieux, assurez-vous d'avoir bien les méta-modèles au format `.ecore` pour ce TP, il faudra générer les méta-modèle PyEcore à partir d'eux. Aussi, créez un virtualenv dédié pour ce TPs si vous ne l'avez pas fait dans les TPs précédents (environnement virtuel Python pour installer des dépendances sans bouleverser votre système).

Génération du code des méta-modèles

Si vous avez le code des méta-modèles déjà générés en Python, vous pouvez sauter cette étape et passer directement à la prochaine section.

Si vous n'avez pas le code des méta-modèles en Python, il est nécessaire de les générer. Pour faire ça, dans un premier temps, il est intéressant de créer un environnement virtuel. Pour créer un environnement virtuel :

1. placez-vous dans votre répertoire de travail (celui de votre repository) et tapez la commande suivante : `$ virtualenv venv` (ou `$ python -m venv venv`). Cela va créer un environnement virtuel dans le répertoire `venv` que vous pourrez activez quand vous voulez.
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`

Vous pouvez maintenant installer les outils nécessaires pour méta-modéliser et travailler avec Ecore de la manière suivante :
`$ pip install pyecore pyecoregen`.

Transformations de modèles avec Motra

L'année passée vous avez manipulé des transformations de modèles type ATL ou, très certainement, QVTo. Ces langages de transformations reposent sur OCL pour la navigation dans les modèles et rajoutent des concepts pour la création/modification/suppression d'éléments dans les modèles manipulés. Dans ce TP, vous allez utiliser un nouveau type de langage qui n'est pas, comme ATL, QVT… un DSL externe, mais un DSL interne à Python nommé Motra (pour MOdel TRAnsformation). Ce DSL repose entièrement sur Python en étendant, d'une certaine manière, le langage et en proposant une sémantique proche de QVTo. Motra est toujours considéré comme expérimental, donc si vous trouvez des soucis, n'hésitez pas à nous les remonter directement ou à écrire une issue détaillée sur github.

1. Activez votre virtualenv et installez la librairie : `$ pip install motra`
2. Regardez les quelques exemples donnés directement sur le repository : https://github.com/pyecore/motra/tree/main/examples/m2m

Vous allez maintenant écrire plusieurs petites transformations (au moins 2) qui seront des transformations "sur-place" et "exogènes". Regardez bien toutes les transformations proposées et sélectionnez celles que vous voulez écrires.

NOTE	Si vous avez l'idée d'une transformation en particulier, parlez en nous et faites là.

Transformation sur votre meta-modèle de BD ou FileSystem ou Simple Java

Vous pouvez implémenter ces transformations pour le méta-modèle que vous préférez. Vous n'avez pas à écrire une seule transformation qui fonctionne avec tout les méta-modèles, ça impliquerait de faire une transformation générique et c'est un poil plus compliqué.

1. Écrivez une transformation qui transforme le nom de toutes les tables (ou fichiers ou java classes) de camel case vers snake case majuscule (ex: `MyTable` deviendra `MY_TABLE`).
2. Écrivez une transformation qui préfixe le nom de chaque éléments avec un numéro unique (vous pouvez utiliser `id(…)` pour obtenir un id unique à partir d'un objet).

Transformations sur `simplejava`

1. Écrivez une transformation qui ajoute un attribut `id` de type `String` pour chaque objets. Attention à n'introduire l'attribut que pour les classes qui n'héritent d'aucune autre. S'il existe déjà dans la classe un attribut qui commence par `id_` ou qui s'appelle `id`, alors la transformation n'ajoutera pas de nouveau l'élément.
2. Écrivez une transformation qui introduit une interface commune possédant une méthode `getUUID()` à tout les objets. Attention à n'introduire l'interface que pour les classes qui n'héritent d'aucune autre.
3. Écrivez une transformation qui détecte les hiérarchies de classes (arbres d'héritages) et qui rajoute au modèle l'équivalent d'une factory, c'est-à-dire, une classe qui possède une méthode

pour créer une instance de chaque classe non abstraite de l'arbre d'héritage découvert.

Transformation de `simplejava` vers votre méta-modèle de BD

Vous allez maintenant écrire une transformation qui permet de passer d'un modèle simple Java vers un modèle de base de donnée. Voilà les questions à vous poser :

- Quels sont les éléments à traduire ?
- Quelle va être la traduction d'une classe ?
- Quelle va être la traduction d'un attribut ?
- Comment gérer l'héritage ?
- Comment déterminer une clef primaire depuis une classe Java ?

Puis, une fois que vous avez des réponses plutôt précises à ces questions :

1. Écrivez la transformation pour passer d'un modèle Java à un modèle de base de données.

Transformation de `simplejava` vers votre méta-modèle de système de fichiers

Comme précédemment, les mêmes questions se posent :

- Quels sont les éléments à traduire ?
- Quelle va être la traduction d'un package ?
- Quelle va être la traduction d'une classe ?

Puis, une fois que vous avez des réponses plutôt précises à ces questions :

1. Écrivez la transformation pour passer d'un modèle Java à un modèle de système de fichiers.

Un moteur de chaîne de transformations

Vous allez développer ici un moteur de chaîne de transformations et expérimenter avec les transformations que vous avez écrites avant. Pour modéliser le moteur de chaîne, vous allez modéliser dans un premier temps un méta-modèle Ecore de chaîne de transformations.

Une chaîne simplifiée de transformations

Une chaîne de transformations est composée de plusieurs opérations. Ces opérations peuvent être des transformations de modèles (M2M et M2T) ou une opération de sauvegarde de modèle vers un chemin en particulier. Chaque opération possède une relation vers la chaîne qui la contient. Chaque opération possède une méthode `execute` qui prend exactement un paramètre, le modèle sur lequel va s'appliquer l'opération et qui retourne le nouveau modèle. Chaque transformation est liée

à une transformation Motra (modélisée par un attribut vers un `EJavaObject` qui sera traduit vers une relation vers un objet Python natif lors de la génération). Finalement, la chaîne en elle-même possède une méthode `run` qui prend en paramètre un chemin vers un fichier et qui retourne le résultat de la dernière opération exécutée.

1. Modélisez le méta-modèle de chaîne de transformations au format Ecore en utilisant le "ecore diagram tool" sous Eclipse. Pour simplifier les prochaines opérations, utilisez ces valeurs pour le package racine de votre méta-modèle : `name=chainengine`, `nsPrefix=chainengine`, `nsURI=http://chainengine/1.0`
2. Une fois que vous avez modélisé votre méta-modèle de chaîne, utilisez la ligne suivante pour générer le code du méta-modèle :
   `$ pyecoregen -e votre_metamodel.ecore -o . --user-module chainengine.mixins`
3. Copiez le fichier de mixins généré avec cette commande : `$ cp chainengine/chainengine_mixins.py.skeleton chainengine/mixins.py`. Vous allez écrire le code des méthodes `execute` et `run` directement dans ce fichier `chainengine/mixins.py`, cela évitera, si vous générez plusieurs fois le code de votre méta-modèle, d'écraser le code que vous avez écrit.

Voici un exemple de comment vous pouvez décrire maintenant une chaîne de transformations (pour un méta-modèle, si vous avez modélisé les choses autrement, vous aurez un autre code) :

```
from chainengine import *
from transfos import t1, t2, t3  # on suppose plusieurs transformations

chain = Chain(name="first_chain")
chain.operations.append(M2M(transformation=t1))
chain.operations.append(M2M(transformation=t2))
chain.operations.append(SaveModel(path='intermediaire.xmi'))
chain.operations.append(M2M(transformation=t1))
chain.operations.append(M2M(transformation=t3))
chain.operations.append(SaveModel(path='final.xmi')))
chain.run('mon_modele.xmi')
```

Actuellement, si vous exécutez ce code, vous allez récupérer une exception vous disant que votre code pour les méthodes `run` et `execute` ne sont pas encore implémentées. C'est ce que vous allez faire maintenant, mais avant ça, il est nécessaire de comprendre un peu comment Motra lance les transformations, ce qu'il attend en paramètre et ce qu'il retourne.

Motra, paramètre d'entrée, de sorties et un `ResourceSet` unique

Lorsque vous avez codé une transformation avec Motra, vous pouvez la lancer de la manière suivante :

```
from pyecore.resources import ResourceSet
from matransfo import t  # import de la transformation

# En considérant que "t" est définie comme t [in_model, out_model]
rset = ResourceSet()
result = t.run(in_model='path_to_model.xmi', resource_set=rset)
```

Il est important de noter que pour lancer la transformation, le nom du modèle d'entrée spécifié lors de la transformation doit-être utilisé. Il faut aussi noter que tout au long de la transformation, il faudra gérer une même instance de `ResourceSet`.

1. Au début de la méthode `run` de votre chaîne, ajouter un attribut d'instance `resource_set` initialisé avec une nouvelle instance de `ResourceSet`. Cette instance pourra être récupérée par toutes les opérations si elles en ont besoin.
2. À la fin de la méthode `run` de votre chaîne, affectez la valeur `None` à votre attribut d'instance `resource_set`. Cela marque le fait qu'une fois que l'exécution de la chaine est terminée, cet attribut n'a plus de raison d'avoir une valeur non-nulle.

Vous allez coder maintenant les méthodes `execute` pour chacune des opérations.

1. Écrivez la méthode `execute` de l'opération de sauvegarde de modèle. Pour rappel, cette méthode reçoit en paramètre la resource représentant le modèle qui doit être sauvegardé, doit sauvegarder le modèle (en utilisant la méthode `save(output=⋯)` de la resource), puis retourner la resource passée en paramètre.
2. Écrivez la méthode `execute` de l'opération d'exécution d'une transformation de modèle. Cette méthode doit :
    a. prendre en paramètre un modèle,
    b. demander à la transformation le nom de son paramètre d'entrée (récupérable par en utilisant le premier élément de la collection `input_defs[0]`),
    c. construire un dictionnaire avec comme clé le nom du paramètre d'entrée et en valeur associé le paramètre d'entrée de la méthode,
    d. récupérer le `ResourceSet` de la chaîne,
    e. exécuter la transformation associé à l'opération avec en paramètre, le resource set récupéré et le dictionnaire que vous avez construit expand avec l'opérateur `**`. Cela signifie que si votre dictionnaire s'appelle `input_dict` et votre resource set stocké dans une variable `rset`, les paramètres passés à l'appel devront être `(resource_set=rset, **input_dict)`,
    f. récupérer le résultat de l'exécution
    g. retourner le premier modèle de sortie de la transformation, récupérable via l'attribut `outputs[0]` du résultat de l'exécution

À l'issue de ces 2 méthodes, vous avez presque fini l'écriture de votre moteur de chaîne, il ne vous reste plus qu'à coder le corps de la fonction qui va vous servir à lancer le tout.

1. Écrivez la méthode `run` de votre chaîne. Cette méthode va tout simplement itérer sur toutes les opérations de votre chaîne et passer le résultat de chaque appels comme paramètre de la

prochaine opération.

2. Utilisez votre moteur de chaîne pour décrire plusieurs chaînes de transformations en composant avec les transformations que vous avez codé précédemment, ou en utilisant les transformations codées par vos camarade, si vous êtes certains que vous avez des méta-modèles compatibles

Vers une fluent-API pour la création de chaînes

Avec votre moteur de chaînes, vous êtes en mesure de créer des chaînes complexes et d'observer comment les transformations peuvent-être réutilisés et l'impact du chaînage sur les modèles. Cependant, vous pouvez remarquer que la description d'un modèle de chaînes est un peu pénible. Ici, vous allez implémenter une fluent-API qui va vous permettre de modéliser plus facilement des chaînes.

Les fluent-API sont des APIs proposés pour simplifier l'utilisation de bibliothèques en donnant une abstraction qui permet de très facilement chaîner plusieurs appels pour écrire une "longue phrase" plutôt que plusieurs lignes. C'est, par exemple, le cas de l'API de stream de Java et, plus généralement, pour implémenter les patterns builder.

Pour faire ceci, vous allez implémenter, dans un module Python séparé (un autre fichier Python), une classe qui servira de point d'entrée pour la création de chaînes. Vous nommerez cette classe comme vous voulez, mais voilà à quoi devrait ressembler ce que vous pourrez exprimer une fois que votre fluent-API aura été codée (ici nommé ChainBuilder):

```
# Les parenthèses de début et fin ne sont là que pour pouvoir écrire le code sur
# plusieurs lignes
(ChainBuilder(name='first_chain')
  .m2m(t1)
  .m2m(t2).save('intermediaire.xmi')
  .m2m(t1)
  .m2m(t3).save('final.xmi')
  .run('mon_modele.xmi'))
```

1. Écrivez une classe permettant de définir une Fluent-API pour votre moteur de chaine.
2. Utilisez votre nouvelle API pour décrire plusieurs chaines de transformations en composant avec les transformations que vous avez codé précédemment, ou en utilisant les transformations codées par vos camarade, si vous êtes certains que vous avez des méta-modèles compatibles