SU- Master 2 DevREp **Lom Hillah**
Tewfik Ziadi

TP2 : Génération de Code Java et manipulation de modèles

NB. : Ce TP est à commencer en salle et à finir chez vous au cas où vous manqueriez de temps. Il est important que vous fassiez l'effort de répondre à toutes les questions. Cela est primordial pour la suite des TPs. Ne comptez pas sur une solution de notre part, vous serez déçus ;-)

En TP2 vous avez appris à faire un méta-modèle avec EMF ainsi que des modèles instances dynamiques à travers l'éditeur arborescent. Le tout sans générer de code Java. Aujourd'hui, vous allez apprendre à créer et à manipuler vos modèles instances à travers l'API Java que EMF va générer pour vous.

Génération de code Java

Pour générer du code Java à partir du méta-modèle .**ecore**, EMF a besoin de rajouter quelques informations nécessaires à la génération Java. Afin de ne pas polluer/annoter votre **ecore**, EMF crée un autre modèle à partir de ce dernier contenant ce qu'il faut pour Java. Ce modèle s'appelle le .**genmodel**. Pour obtenir un .genmodel à partir de votre .ecore,

1) click droit sur votre .ecore dans le Package Explorer **->new->other…->Eclipse Modeling Framework-> EMF Generator Model** puis Next
2) nommer votre genmodel « turtle**.genmodel** » puis Next
3) Laisser l'option par défaut i.e., à partir d'un Ecore Model puis Next
4) Cliquez sur **Load…**, puis sur Next
5) Puis sur Finish !
6) Parcourez le nouveau model généré, vous verrez qu'il ressemble en tout point au .ecore. Par contre, si vous ouvrez la version xmi des deux modèles, vous pourrez constater une différence au niveau des concepts/balises utilisés (un autre schéma). Si vous modifiez votre .ecore, il vous suffira de faire un click droit sur le modèle (.genmodel) puis Reload…
7) Maintenant, vous êtes prêts pour lancer la génération de code ! Ouvrez votre genmodel dans l'éditeur arborescent, puis click droit sur la racine**->Generate All** pour générer toute l'API Java pour manipuler vos modèles, un éditeur arborescent et les tests qui vont avec. Attention, si vous avez des erreurs de compilation après la génération, il suffit parfois juste de relancer la génération de code. Si ça ne marche toujours pas, demander un coup de pouce à votre enseignant.
8) Parcourez le code généré, essentiellement les packages Turtle et Turtle**.impl** (peut être un autre nom .impl). Le package Turtle contient principalement les interfaces et les factories qui vous permettront de créer des modèles instances de votre .ecore. Le package .impl contient les classes d'implémentation de vos interfaces.
9) Ouvrez le fichier **plugin.xml** puis allez dans le tab « **Dependencies** », dans la partie **Required plugin**s, cliquez sur **Add..** puis tapez **org.eclipse.emf.ecore.xmi**, rajoutez le à la liste et sauvegarder.

IMPORTANT : à chaque fois que vous modifiez le .ecore il faudra régénérer le .genmodel puis le code. Si entre temps vous avez modifié le code de l'API ou que vous avez rajouté votre propre code en implémentant Display par exemple, il faudra l'annoter avec **generated NOT** afin que EMF ne l'efface pas à la prochaine génération de code. Prenez le comme une règle, avant de modifier/implanter une méthode, je modifie son annotation de **generated** à **generated NOT**.

Manipulation de modèles instances avec l'API Java générée

A vous de jouer maintenant ! Voici ce qui est demandé :

1) On aimerait charger (en mémoire sous forme d'objets Java) le modèle instance que vous avez crée en TP1, le parcourir et afficher ce qu'il fait. En gros, on aimerait connaître les dimensions du Stage, les Choreographies qu'il contient, les actions de chaque Choreography (le type de l'action, la langueur si c'est un Forward, l'angle si c'est un Rotate et la position du pen si c'est un SetPen) ainsi que le nom de la tortue qui exécute l'action ! Pour cela, une fois que vous aurez charger le model avec la méthode load à partir du main de ModelIO, appelez la méthode **Display** de Stage. Si tout se passe bien la méthode devra juste retourner un String « End » ainsi que la chaine qui suit :

```
SetPen :DOWN
Turtle :Franklyn
Forward :100
Turtle :Franklyn
SetPen :UP
Turtle :Franklyn
End
```

Pour la méthode Load, , on vous la donne (voir aussi le poly du cours). Voir partie Aide ci-dessous.

2) La deuxième étape consiste à modifier votre modèle, toujours en utilisant l'API Java, et à le sauvegarder. Pour cela vous devrez créer une nouvelle Choreography, de nouvelles actions, des turtles, etc et relier le tout au Stage (du modèle précédent). **Attention** vous devez avoir le méta-modèle en tête pour ne pas oublier d'affecter les références entre les différents concepts de votre langage (Stage, Turtle, Action, etc.). Vos premières erreurs d'exécution seront souvent liées à cela. Vérifiez bien votre graph d'objets avant de le sauvegarder. On vous offre la méthode save().Voir partie Aide ci-dessous. Exécuter de nouveau votre programme après modification de votre modèle pour voir si la nouvelle Choreography ainsi que les nouvelles actions, turtles, s'affichent bien comme il le faut

3) La troisième étape consiste à afficher en HTML les instances.

La suite au prochain TP :

- Un peu de Transformation de modèles
- Un peu de génération automatique d'éditeur textuel avec JET

Aide :

```
package suniv;

import java.io.IOException;
import java.nio.file.Path;
import java.util.Collections;
import java.util.Map;

import org.eclipse.emf.common.util.URI;
import org.eclipse.emf.ecore.EObject;
```

```
import org.eclipse.emf.ecore.EPackage;
import org.eclipse.emf.ecore.resource.Resource;
import org.eclipse.emf.ecore.resource.ResourceSet;
import org.eclipse.emf.ecore.resource.impl.ResourceSetImpl;
import org.eclipse.emf.ecore.xmi.impl.XMIResourceFactoryImpl;

import fr.lip6.turtle.TurtlePackage;
import fr.lip6.turtle.impl.TurtlePackageImpl;

/**
 * Utility class to save and load model instances of Turtle from XMI files.
 * @author lmh
 */
public final class ModelIO {

	private ModelIO() {}

	/**
	 * Loads a model stored in a XMI file.
	 * @param file the path to the incoming XMI file
	 * @return the root object of the loaded model
	 */
	public static final EObject loadModel(Path file) {
		Resource.Factory.Registry reg =
Resource.Factory.Registry.INSTANCE;
		Map<String, Object> m = reg.getExtensionToFactoryMap();
		m.put("*", new XMIResourceFactoryImpl());

		// Obtain a new resource set
		ResourceSet resSet = new ResourceSetImpl();
		TurtlePackage tp = TurtlePackageImpl.init();
		EPackage.Registry ereg = resSet.getPackageRegistry();
		ereg.put(tp.getNsURI(), tp);

		Resource resource =
resSet.getResource(URI.createURI(file.toString()),  true);
		return resource.getContents().get(0);
	}

	/**
	 * Saves a model in a XMI file.
	 * @param mo the model to save
	 * @param file the path to the destination file.
	 */
	public static final void saveModel(EObject mo, Path file) {
		Resource.Factory.Registry reg =
Resource.Factory.Registry.INSTANCE;
		Map<String, Object> m = reg.getExtensionToFactoryMap();
		m.put("*", new XMIResourceFactoryImpl());

		// Obtain a new resource set
		ResourceSet resSet = new ResourceSetImpl();
		Resource resource =
resSet.createResource(URI.createURI(file.toString()));

		// Get the first model element and cast it to the right type,
in my
		// example everything is hierarchical included in this first
node
		resource.getContents().add(mo);

		// now save the content.
		try {
			resource.save(Collections.EMPTY_MAP);
```

```
            } catch (IOException e) {
                  e.printStackTrace();
            }
      }
}
```