UPMC- Master 2 Module Car **Reda Bendraou**
Tewfik Ziadi

TP5 : Meta-Programmation avec Kermeta

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 ;-)

Dans ce TP vous allez apprendre un langage de méta-programmation i.e., un langage pour la manipulation de modèles (jusqu'à présent, vous utilisiez le code Java généré par EMF). Dans notre cas c'est Kermeta. Kermeta fournit un bon nombre de fonctionnalités. Dans un premier temps il sert à définir le comportement exécutable (i.e. la sémantique opérationnelle) de vos méta-modèles. Ainsi une fois vos modèles crées, ils seront directement exécutables par un simple click (à partir d'un main). Le comportement exécutable est défini dans un fichier .kmt qui sera tissé sous forme d'un aspect au niveau de vos modèles au moment de l'exécution. Il est ainsi possible de choisir entre plusieurs scénarios d'exécution possibles au moment de l'exécution (cas des lignes de produits par exemple). Avec Kermeta, il est également possible de définir des contraintes (à la OCL) sous forme de fichier .kmt (aspect). Ce dernier sera tissé au moment de l'exécution sur vos modèles. Finalement, Kermeta peut être utilisé comme langage de transformation de modèles. La transformation est définit dans un fichier .kmt, elle applique le principe du pattern visiteur afin de produire un modèle cible à partir de n'importe quel modèle source. Vous devez mettre en pratique tous ces aspects !

Tissage de la sémantique opérationnelle

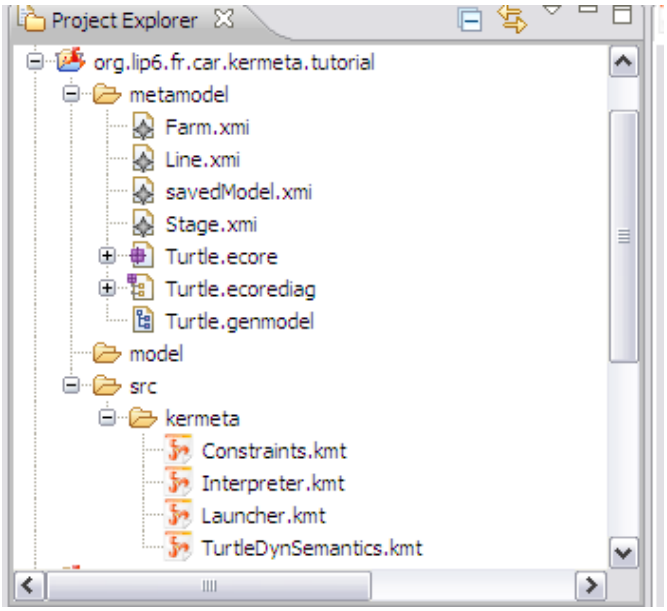
Dans cette étape on va reproduire le comportement de la tortue que vous avez déjà écrit en Java mais en Kermeta. L'idée est de voir la différence entre les deux et surtout que maintenant, vous n'avez plus besoin de générer une seule ligne de Java.

1- Télécharger le bundle Eclipse Kermeta :
http://www.kermeta.org/releases/1.4.1/eclipse_package/

2- Créez un nouveau projet Kermeta : File->new Project…->Kermeta->new Kermeta Project
3- Nommez votre projet : **org.lip6.fr.car.kermeta.tutorial** puis cliquez sur finish
4- Copiez votre méta-modèle Turtle.ecore ainsi que les modèles instances (.xmi) que vous avez crée dans vos précédents TP dans le répertoire metamodel de votre projet Kermeta (attention aux dépendances entre les différents modèles Stage et Fram par exemple)
5- Ouvrez vos modèles et vérifiez que tout est OK, sinon demandez à votre enseignant un coup de main
6- Mettez les fichiers **.kmt** suivants (présents dans le répertoire matos de votre zip de TP) sous le répertoire **src/kermeta** :

- **TurtleDynSemantics.kmt** : ce fichier contiendra la sémantique opérationnelle de vos méta-classes
- **Constraints.kmt** : contiendra les contraintes (exp. Invariants à la OCL) sur vos modèles (ce fichier est à compléter dans la partie 2 de ce TP)
- **Interpreter.kmt :** contiendra les méthodes pour charger les modèles en mémoire et pour lancer son exécution
- **Launcher.kmt :** est le main de votre programme.

Le fichier **TurtleDynSemantics.kmt** est à compléter par vos soins. A ce niveau vous devait avoir la structure suivante au niveau de votre Projet :



7- Un premier pas pour tester Kemrta/les aspects et pour vérifier que ça marche :

- Rajoutez dans **TurtleDynSemantics.kmt** l'aspect suivant :

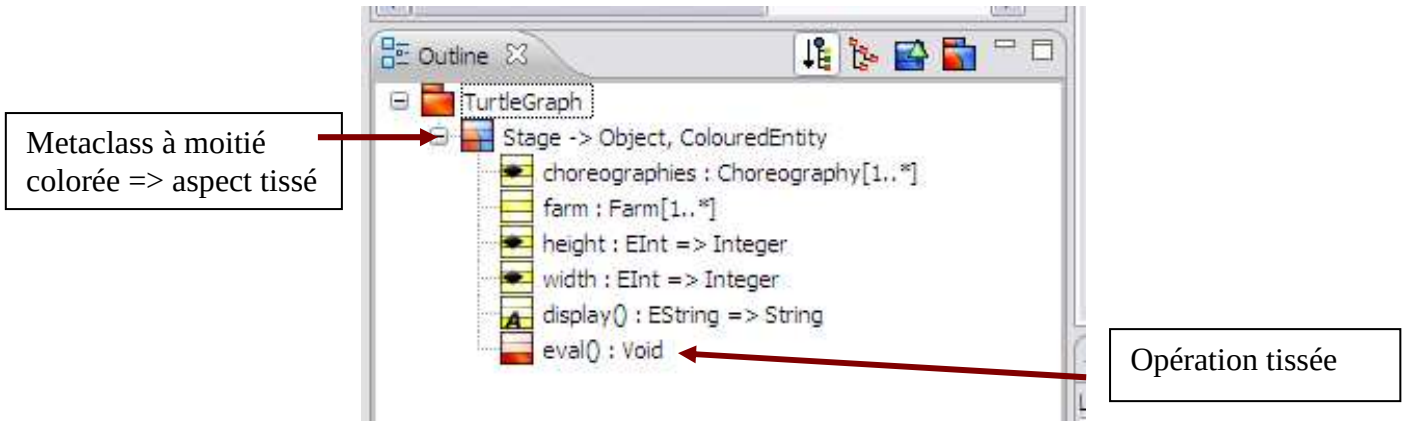
```
aspect class Stage
{
	operation eval() : Void is do

		stdio.writeln("Coucou, mon premier programme Kermeta")

	end

}
```

- Au niveau de l'Eclipse Outline, vous pouvez vérifier que votre aspect a bien été tissé :



- Allez au niveau du fichier Launcher.kmt et click droit->run as Kermeta Application
- Vérifier la console

8- A vous de jouer maintenant ! Rajoutez le comportement des metaclasses Stage, Choreography, Forward, Rotate, etc. Pour cela vous pratiquerez les opérations sur les collections (les lambda expressions), les concepts d'opération et de méthode, et les liens avec

Java (si vous décidez de nous faire un affichage dans une frame Java). **IMPORTANT** : Utilisez le Memento Kermeta joint à ce TP.

Tissage des Contraintes (sémantique statique)

On souhaite maintenant tisser des contraintes, des invariants plus exactement au niveau de vos modèles instances de Turtle. Pour cela on utilisera Kermeta. Les contraintes sont à mettre au niveau du fichier **Constraints.kmt**. On vous aidant avec le memento et les notes de cours, écrivez les contraintes suivantes au niveau de **Constraints.kmt** :

- La largeur et la hauteur d'un Stage doivent toujours être supérieures à 100
- Une Choegraphy ne doit pas contenir plus de 10 actions
- Dans une Choegraphy, il ne doit pas y avoir deux actions Rotate qui se suivent

Aide :

Voici un exemple sur comment rajouter un invariant à Stage sous forme d'aspect dans **Constraints.kmt** :

```
aspect class Stage{
	/**
	 * Testing Height & width of the Stage
	 */
	inv nomDeVotreInvariant is
	do
		Expression Booléenne ici
	end
}
```

Pour tester ces invariants, rajoutez ce bout de code au niveau d'**Interpreter.kmt**, juste après le chargement du modèle (i.e., après l'appel à loadProgram(uri)):

```
	do
		root.checkAllInvariants()
	rescue (e : kermeta::exceptions::Exception)
		stdio.writeln("Error in model "+ uri +" : " + e.message)
	end
```

À vous de jouer maintenant !

En Option : Tissage de transformation/génération (compilation)

Finalement, de la même manière et toujours en utilisant les aspects, écrivez un fichier .kmt qui contiendra l'aspect pour la génération d'XML à partir d'un modèle instance de Turtle. Pour cela vous appliquerez le pattern visiteur comme vu dans le précédent TP.

Vous pouvez télécharger la version complète (workspace Eclipse) de l'exemple présenté en cours ici : http://pagesperso-systeme.lip6.fr/Reda.Bendraou/CAR/workspaceLogo_Kermeta.zip
Pour l'exécuter (dans un Eclipse qui contient le plugin Kermeta) : allez dans le sous répertoire logo/5.Simulator/tests/ et lancer un des programmes .kmt au choix