Projet – Ingénierie des Modèles

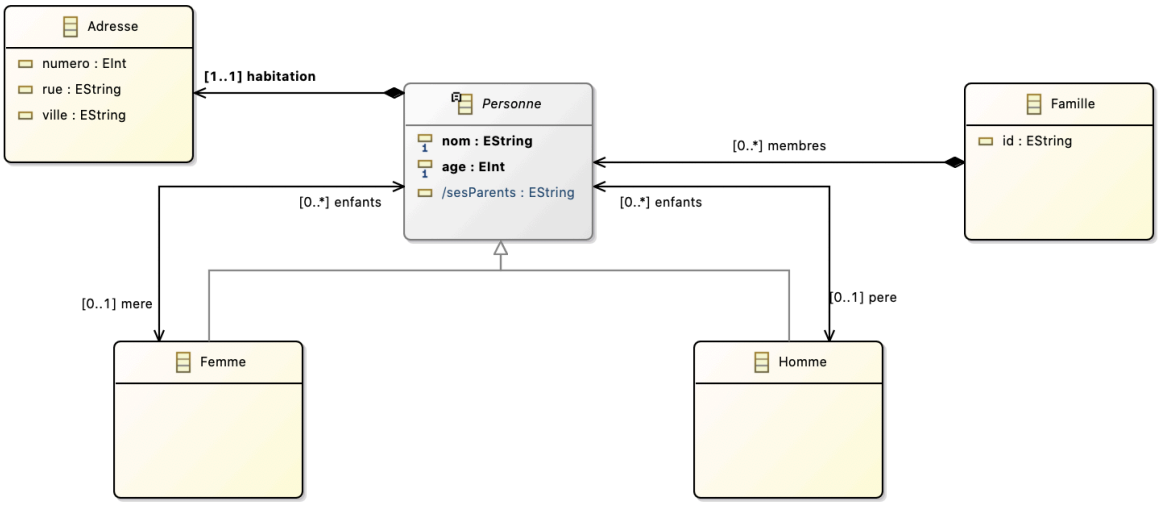
Création d'un DSL de A à Z

Objectifs : le but de ce projet est d'apprendre à implémenter un DSL (Domain Specific Language) en utilisant le framework de modélisation d'Eclipse appelé EMF. Nous couvrirons tous les aspects liés aux développements dirigés par les modèles avec les briques logiciels fournis par la plateforme Eclipse. La version Eclipse avec le package modeling sera utilisée à cet effet.

Buts pédagogiques : conception de modèles EMF, génération de code Java, manipulation d'API EMF pour instancier le modèle, manipulation du méta-modèle Ecore, sauvegarde et chargement de fichiers XMI, transformation de modèles vers textes.

Étape 1 : Création du modèle EMF d'un arbre généalogique

But : Démarrer un projet EMF, création d'un modèle EMF avec les outils Eclipse, manipuler les différents éditeurs
Description : Une famille est caractérisée par un ID contient un ensemble de personnes (membres). Une personne est identifiée par un nom et un âge. Une personne, qui peut être de type homme ou femme, contient obligatoirement une adresse « habitation ». Une adresse est identifiée par un numéro, une rue et une ville. Voir le schéma ci-dessous qui représente graphiquement la modélisation attendue par cet exercice. On s'intéresse ici à modéliser via les outils EMF le modèle présenté.



Réalisation :

1. Choisir comme nom de projet *fr.ensma.idm.projet.familydsl* puis faire directement **Finish**.
2. Choisir comme nom du fichier ecore *famillemm.ecore*
3. Choisir comme nom du package principal du métamodèle « arbregen »
4. Sélectionner le répertoire *model*, puis le nœud *famillemm.ecore* et enfin ouvrir l'éditeur de diagramme de classes.
5. Visualiser votre modèle à partir des différents éditeurs (OCLinEcore (Ecore) Editor, Sample Ecore Model Editor). Si l'éditeur OCLinEcore n'est pas installé par défaut, vous pouvez l'ajouter en installant le plugin Eclipse OCL ;

Étape 2 : Génération de codes Java (l'API du méta-modèle)

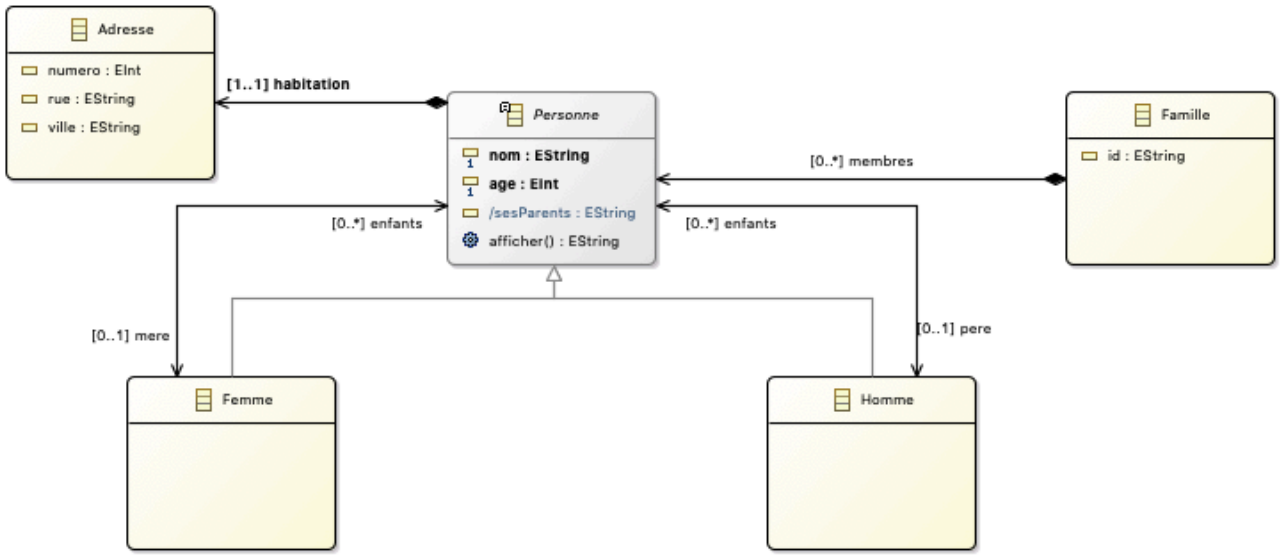
But : Créer un modèle de génération à partir d'un Ecore, paramétrer le modèle de génération de code, générer un code Java correspondant au modèle EMF, mise à jour du modèle et re-génération du code généré et modification du code généré.

Description : On s'intéresse dans cet exercice à toutes les étapes de génération de code à partir d'un modèle EMF. On s'intéresse également aux étapes de re-génération et de protection des codes modifiés explicitement par le développeur. Ce modèle contient les informations dédiées uniquement à la génération et qui ne pourraient pas être intégrées au modèle (chemin de génération, package, préfixe, …). Ce modèle appelé genmodel est également un modèle EMF et chaque classe du modèle de génération est un décorateur des classes Ecore.

Réalisation :

1. Créer le fichier *famillemm.genmodel* (en utilisant EMF Generator Model). Visualiser le genmodel avec l'outil d'édition EMF Generator. Vous noterez une ressemblance avec le contenu du fichier *famillemm.ecore* hormis le fait que le modèle décrit dans *famillemm.genmodel* ne s'appuie pas sur le même méta-modèle.
2. Modifier le contenu du fichier genmodel pour que le package de génération soit `fr.ensma.idm.projet.familydsl.model` (propriétés : *Base Package*) ;
3. Sélectionner depuis le genmodel le package racine `Arbregen` et générer le code Java correspondant au modèle (*Generate Model Code*). Un ensemble de classes Java doivent être générées dans le package `fr.ensma.idm.projet.familydsl.model.arbregen` ;
4. Examiner les classes générées et remarquer le découpage en trois catégories qui font apparaître une programmation par contrats des interfaces et des implémentations.

Nous décidons par la suite de modifier le modèle de façon à ajouter une opération *String afficher()* dans la classe *Personne* qui se chargera d'effectuer un affichage complet d'une instance de `Personne`. Le schéma ci-dessous représente graphiquement la modélisation attendue par cette modification.



5. Compléter votre modèle EMF via l'éditeur de diagrammes de classes de façon à intégrer les modifications demandées.
6. Le fichier *arbregen.ecore* est automatiquement impacté. Toutefois, le fichier genmodel peut ne pas être mis à jour. Sélectionner le fichier *arbregen.genmodel* puis cliquer sur *Reload* (via le menu contextuel). Sélectionner ensuite *Ecore model* et laisser les valeurs par défaut puis valider. Vous remarquerez que les nouveaux changements ont été pris en compte et que les anciennes valeurs de configuration de génération (*Base Package* en l'occurrence) n'ont pas été supprimées ;
7. Re-générer les codes Java (Generate Model Code) ;
8. Modifier la classe `fr.ensma.idm.projet.familydsl.model.arbregen.impl.PersonneImpl` de façon à implémenter la méthode `afficher()` (voir code ci-dessous) ;

```
    /**
     * <!-- begin-user-doc -->
     * <!-- end-user-doc -->
     * @generated NOT
     */
    public String afficher() {
           StringBuffer sb = new StringBuffer();
           sb.append("Prenom : ").append(this.getNom())
                    .append(" Age:").append(this.getAge())
                    .append(" Adresse:").append(this.getHabitation());

           return sb.toString();
}
```

9. Re-générer les codes Java (Generate Model Code) et assurez-vous que le code que vous avez saisi n'a pas été modifié.

Étape 3 : Création d'instances via l'éditeur généré

But : Générer un éditeur graphique, exécuter une configuration d'exécution, création d'instances via l'éditeur généré, validation des contraintes

Description : Nous allons maintenant générer le code correspondant à un éditeur graphique. Cet éditeur sera utilisé pour créer graphiquement des instances de notre modèle. Nous vérifierons par ailleurs la validité de notre modèle par rapport à un jeu d'instances.

Réalisation :

1. À partir du modèle de génération (genmodel) générer le code de l'éditeur (Generate Edit Code et Generate Editor Code). Deux plugins doivent être créés (*fr.ensma.idm.projet.familydsl.edit* et *fr.ensma.idm.projet.familydsl.editor*) ;
2. Passer en perspective *Java* ;
3. Créer une configuration d'exécution (*Run as -> Run Configurations…*) à partir d'un type Eclipse Application. Nommer cette configuration *FamilyDSLConfiguration*, modifier la valeur de son chemin avec cette valeur (*${workspace_loc}/runtime- FamilyDSLConfiguration*) puis ajouter vos trois plugins (*familydsl*, *edit* et *editor*). Décocher *Target Platform* puis faites *Add Required Plug-ins*. Ajouter enfin le plugin *org.eclipse.ui.ide.application* et *org.eclipse.ui.navigator.resources* et faires une nouvelle fois *Add Required Plug-ins* ;
4. Exécuter cette configuration d'exécution. Une nouvelle instance d'Eclipse s'exécute en intégrant votre éditeur de modèle de carnet d'adresse ;
5. Créer un simple projet (**File -> New -> Project … -> General -> Project**) que vous appellerez *ExempleInstanceFamille* ;
6. À partir de cette nouvelle instance, créer une instance du modèle *FamilleTest* (**File -> New -> Other … -> Example EMF Model Creation Wizards -> Arbregen Model**) que vous appellerez *FamilleTest.arbregen*. Choisir ensuite *Famille* comme classe racine ;
7. Construire les instances via l'éditeur associé à votre modèle.
8. Sélectionner le nœud racine de vos instances et valider ces instances en cliquant sur **Validate** (via le menu contextuel).

Étape 4 : Création d'instances via l'API EMF : EarlyBinding

But : Créer des instances via l'API EMF, créer un plugin (fragment) de test, sauvegarder et charger via l'API EMF des instances via un XMI (en utilisant la première instance d'Eclipse).

Description : Nous allons dans cette étape créer des instances d'un modèle de manière programmatique à partir de classes connues avant l'exécution du programme (EarlyBinding). Dans ce cas les plugins générés précédemment (Edit et Editor) ne seront pas utilisés. Nous utiliserons un plugin spécifique appelé fragment (utilisé pour enrichir un plugin existant) pour créer des classes de tests.

Réalisation :

1. Créer un nouveau plugin de type fragment (**File -> New -> Other …-> Plug-in Development -> Fragment Project**) nommé *fr.ensma.idm.projet.familydsl.test* Choisir comme plugin hôte *fr.ensma.idm.projet.familydsl* (créé à l'étape 1). Une fois le fragment créé, ajouter la dépendance (onglet Dependencies du fichier *manifest.mf*) vers le plugin org.junit ;

2. Créer un package *fr.ensma.idm.projet.familydsl.model.test* et créer une classe appelée `FamilleTest`;
3. Depuis la classe `FamilleTest` compléter la méthode de tests `createFamilleTest` en s'assurant que les assertions associées soient vraies. Il vous est demandé de construire des instances (via la fabrique `ArbregenFactory`) et de modifier les valeurs des attributs. **L'exécution du test unitaire doit se faire obligatoirement dans un environnement de plugins (Run As JUnit Plug-in Test) ;**

```
package fr.ensma.idm.projet.familydsl.model.test;

import org.junit.Assert;
import org.junit.Test;
import fr.ensma.idm.projet.familydsl.model.arbregen.Adresse;
import fr.ensma.idm.projet.familydsl.model.arbregen.ArbregenFactory;
import fr.ensma.idm.projet.familydsl.model.arbregen.Famille;
import fr.ensma.idm.projet.familydsl.model.arbregen.Femme;
import fr.ensma.idm.projet.familydsl.model.arbregen.Homme;

public class FamilleTest {

    @Test
    public void createFamilleTest() {
            Famille uneFamille = ArbregenFactory.eINSTANCE.createFamille();
            uneFamille.setId("la famille test");

            Homme marc = ArbregenFactory.eINSTANCE.createHomme();
            Homme claude = ArbregenFactory.eINSTANCE.createHomme();
            Femme marcelle = ArbregenFactory.eINSTANCE.createFemme();
            // A compléter
            Adresse adr1 = ArbregenFactory.eINSTANCE.createAdresse();
            adr1.setNumero(44);
            adr1.setRue("larue");
            adr1.setVille("laville");

            marc.setAge(50);
            marc.setNom("Marc");
            marc.setPere(claude);
            marc.setMere(marcelle);
            marc.setHabitation(adr1);

            // A compléter

            uneFamille.getMembres().add(claude);

            // A compléter

            Assert.assertEquals(3, uneFamille.getMembres().size());
            Assert.assertEquals("la famille test", uneFamille.getId());
            Assert.assertEquals("Marc", marc.getNom());
            Assert.assertEquals("larue", marc.getHabitation().getRue());

            // A compléter
    }
}
```

4. On s'intéresse maintenant à sauvegarder des instances depuis un fichier XMI. Compléter la méthode de tests de manière à sauvegarder les instances créées précédemment (voir code ci-dessous). Le fichier d'instances sera stocké dans le répertoire utilisé par la configuration d'exécution de l'étape 3. Ajouter dans votre plugin la dépendance vers le plugin *org.eclipse.emf.ecore.xmi* et exécuter le test unitaire ;

```
    // A compléter
ResourceSet resourceSet = new ResourceSetImpl();
resourceSet.getResourceFactoryRegistry().getExtensionToFactoryMap().put("arbregen", new XMIResourceFactoryImpl());

final String apiSamplePath = "c://votreworkspace//runtime- FamilyDSLConfiguration//ExempleInstanceFamille//";
URI uri = URI.createURI("file:/" + apiSamplePath + "APIFamilleTest.arbregen");
Resource resource = resourceSet.createResource(uri);
resource.getContents().add(uneFamille);
            try {
                    resource.save(null);
            } catch (IOException e) {
                    e.printStackTrace();
            }
            Assert.assertTrue(new File(apiSamplePath + "APIFamilleTest.arbregen").exists());
    }
    }
```

5. On s'intéresse ensuite à charger des instances depuis un fichier XMI. Compléter la méthode de tests de manière à charger les instances créées lors de l'étape 3 (fichier *FamilleTest.arbregen) ;*

```
resourceSet = new ResourceSetImpl();
uri = URI.createURI("file:/" + apiSamplePath + "FamilleTest.arbregen");
resource = resourceSet.getResource(uri, true);
uneFamille = (Famille) resource.getContents().get(0);
Assert.assertNotNull(uneFamille.getMembres());
Assert.assertEquals(3, uneFamille.getMembres().size());
```

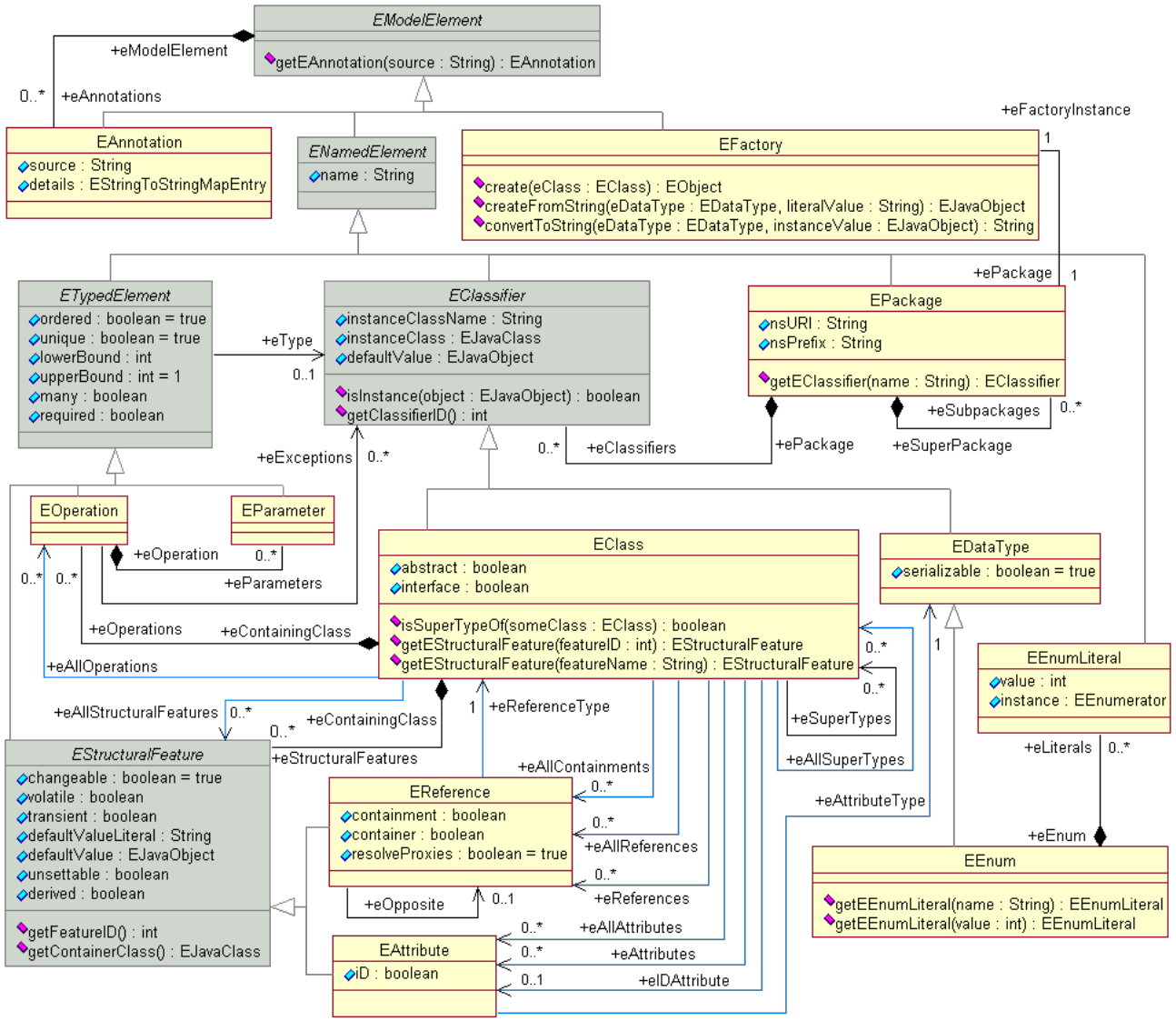
6. Compléter finalement la méthode de façon à ajouter pour les instances de type `Famille` un écouteur sur les changements. Réaliser un affichage qui donne les anciennes et les nouvelles valeurs.

```
@Test
public void createFamilleTest() {
        Famille uneFamille = ArbregenFactory.eINSTANCE.createFamille();

        uneFamille.eAdapters().add(new EContentAdapter() {
                @Override
                public void notifyChanged(Notification notification) {
                        System.out.print("Ancienne Valeur : " + notification.getOldValue());
                        System.out.println(" Nouvelle Valeur : " + notification.getNewValue());
                }
        });
        ... // Identique au précédent
}
```

Étape 5 : Manipulation en liaison différée (LateBinding = instanciation dynamique)

But : Manipuler le méta-modèle, création d'instances via le méta-modèle, création d'instances via les outils d'Eclipse

Description : Nous allons dans cet exercice manipuler le méta-modèle Ecore afin de connaître la structure de notre modèle (puisque le modèle famillemm est une instance du méta-modèle Ecore). Nous allons également créer et modifier des instances de notre modèle via le méta-modèle Ecore. Finalement nous sauvegarderons et chargerons ces instances afin d'obtenir un fichier XMI identique à l'étape 4. L'intérêt de cette étape est d'utiliser une API de type LateBinding où les classes ne sont pas connues avant l'exécution du programme. Nous ferons donc appel explicitement au méta-modèle Ecore. Nous présentons à titre indicatif le méta-modèle Ecore.



Réalisation

1. Dans la classe `FamilleTest` du fragment *fr.ensma.idm.projet.familydsl.test* ajouter une méthode appelée `queryArbreGenStructure`. L'objectif de cette méthode est d'afficher la structure complète de votre modèle en interrogeant le méta-modèle. Le résultat attendu est donné par le listing ci-dessous :

```
Personne
  nom(EString) age(EInt) sesParents(EString)
  Références : mere(Femme[0..1])pere(Homme[0..1])habitation(Adresse[1..1])
  Opérations : EString afficher

Homme

  Références : enfants(Personne[0..-1])
  Hérite de : Personne
Femme

  Références : enfants(Personne[0..-1])
  Hérite de : Personne
Famille
  id(EString)
  Références : membres(Personne[0..-1])
Adresse
  numero(EInt) rue(EString) ville(EString)
```

Le point d'accès au méta-modèle se fait par l'intermédiaire du package `AddressbookPackage`. Compléter le code donné ci-dessous afin d'obtenir le résultat escompté.

```
@Test
public void queryArbreGenStructure() {
	ArbregenPackage lePackageRacine = ArbregenPackage.eINSTANCE;
	EList<EClassifier> eClassifiers = lePackageRacine.getEClassifiers();

	for (EClassifier eClassifier : eClassifiers) {
		System.out.println(eClassifier.getName());


		System.out.print("  ");

		if (eClassifier instanceof EClass) {

	// A Compléter
}
```

Pour l'instant nous avons vu que pour créer des instances du modèle nous devions utiliser les classes générées approche dite EarlyBinding car les classes sont connues avant l'exécution. Par réflexivité, il est possible de créer et modifier des instances du modèle sans avoir à manipuler explicitement les classes générées.

2. Construire un projet EMF vide (***File -> Project … -> Eclipse Modeling Framework -> Empty EMF Project***) nommé *fr.ensma.idm.projet.familydsl.latebiding* Ouvrir le fichier MANIFEST.MF et ajouter la dépendance vers les plugin *org.junit* (4.12.0) et *org.eclipse.emf.ecore.xmi.* Créer ensuite le package `fr.ensma.idm.projet.familydsl.latebiding` et finalement créer à l'intérieur la classe `ArbreGenLateBinding` ;
3. Copier votre fichier *famillemm.ecore* réalisé à l'étape 1 dans le répertoire *model* de votre nouveau projet. À noter que ce nouveau plugin ne contient aucune dépendance vers les plug-ins créés précédents,
4. Créer une méthode de test appelée `queryArbreGenStructureWithoutGeneratedCode` et dont l'objectif est 1) de charger le fichier *famillemm.ecore* afin de charger le modèle (accessible via le package racine) et 2) d'afficher la structure du modèle comme précisée dans 5.1 ;

```
@Test
public void queryArbreGenStructureWithoutGeneratedCode() {
	Resource.Factory.Registry reg = Resource.Factory.Registry.INSTANCE;
	Map<String, Object> m = reg.getExtensionToFactoryMap();
	m.put("ecore", new XMIResourceFactoryImpl());
	ResourceSet resourceSet = new ResourceSetImpl();
	URI fileURI = URI.createFileURI("model/famillemm.ecore");
	Resource resource = resourceSet.getResource(fileURI, true);

	EPackage ePackage = (EPackage) resource.getContents().get(0);
	EList<EClassifier> eClassifiers = ePackage.getEClassifiers();

	for (EClassifier eClassifier : eClassifiers) {
		System.out.println(eClassifier.getName());
		System.out.print("  ");

		if (eClassifier instanceof EClass) {
			// A Compléter
		}
	}
}
```

5. Créer une méthode appelée `createAndSaveArbreGenWithMetaModel` dont l'objectif est 1) de charger le fichier *famillemm.ecore* afin de charger le modèle (accessible via le package racine) 2) de créer des instances identique à celles créées pendant les étapes 3 et 4) sauvegarder les instances dans un fichier XMI;

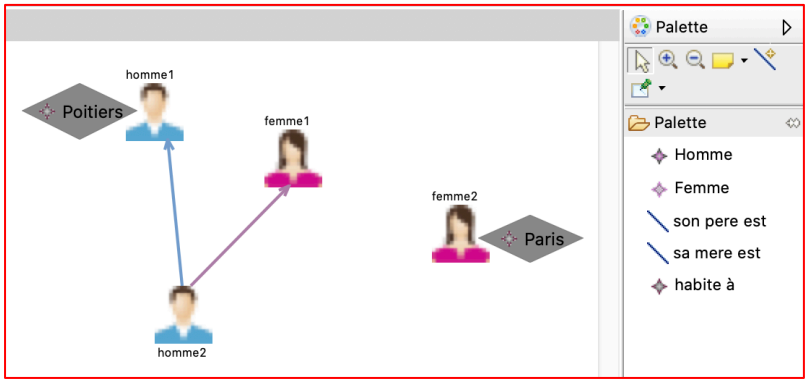
```
@Test
public void createAndSaveArbreGenBookWithMetaModel() throws IOException {


	// Objectif 1 : charger le fichier famillemm.ecore
	Resource.Factory.Registry reg = Resource.Factory.Registry.INSTANCE;
	Map<String, Object> m = reg.getExtensionToFactoryMap();
	m.put("ecore", new XMIResourceFactoryImpl());

	ResourceSet resourceSet = new ResourceSetImpl();
	URI fileURI = URI.createFileURI("model/famillemm.ecore");
	Resource resource = resourceSet.createResource(fileURI);

	resource.load(null);
	EPackage ePackage = (EPackage) resource.getContents().get(0);

	// Objectif 2 : créer des instances
	EClass famille = (EClass) ePackage.getEClassifier("Famille");
	EAttribute idDeFamille = (EAttribute) famille.getEStructuralFeature("id");
	EReference membresDeFamille = (EReference) famille.getEStructuralFeature("membres");
	EObject uneFamille = ePackage.getEFactoryInstance().create(famille);
	uneFamille.eSet(idDeFamille, "La famille test");
```

```
        // A compléter : création de certaines personnes

        EClass personne = (EClass) ePackage.getEClassifier("Homme");
        EAttribute nomDePersonne = (EAttribute) personne.getEStructuralFeature("nom");
        EAttribute ageDePersonne = (EAttribute) personne.getEStructuralFeature("age");
        EReference habiationdePersonne = (EReference) personne.getEStructuralFeature("habitation");
        EObject marc = ePackage.getEFactoryInstance().create(personne);
        marc.eSet(nomDePersonne, "Marc");
        marc.eSet(ageDePersonne, 30);

      // A compléter : création des adresses

        EClass adresse = (EClass) ePackage.getEClassifier("Adresse");
        EAttribute numeroDeAdresse = (EAttribute) adresse.getEStructuralFeature("numero");
        EAttribute rueDeAdresse = (EAttribute) adresse.getEStructuralFeature("rue");
        EAttribute villeDeAdresse = (EAttribute) adresse.getEStructuralFeature("ville");
        EObject marcAdr = ePackage.getEFactoryInstance().create(adresse);
        marcAdr.eSet(numeroDeAdresse, 44);
        marcAdr.eSet(rueDeAdresse, "larue");
        marcAdr.eSet(villeDeAdresse, "laville");

      // A compléter : ajout des adresses aux personnes

        marc.eSet(habiationdePersonne, marcAdr);

        // A compléter : ajout de la personne à l'instance uneFamille

        List<EObject> membresList = new ArrayList<EObject>();
        membresList.add(marc);
        uneFamille.eSet(membresDeFamille, membresList);


        resourceSet = new ResourceSetImpl();
        resourceSet.getResourceFactoryRegistry().getExtensionToFactoryMap()
                        .put("xmi", new XMIResourceFactoryImpl());

        URI uri = URI.createURI("model/familleViaDynamicInstanciation.xmi");
        resource = resourceSet.createResource(uri);
        resource.getContents().add(uneFamille);
        resource.save(null);
    }
```

Étape 6 : Création d'une syntaxe concrète graphique

But : L'objectif de cette étape est de créer une syntaxe concrète graphique pour le DSL.

Description : grâce à l'utilisation du plugin Sirius, vous devez pouvoir instancier les arbres généalogiques graphiquement à travers un diagramme doté d'une palette.



Réalisation :

1. Installer le plugin Sirius à travers le marketplace ;
2. Créer un projet du type «*Viewpoint Specification Project*» et choisir comme com *fr.ensma.idm.projet.familydsl.design*
3. Définir la syntaxe graphique en créant et enrichissant un «*Viewpoint Specification Model*» nommé *familydsl.odesign*

NB : Inspirez-vous des tutoriels suivants et adaptez les fichiers sources à votre besoin :
https://wiki.eclipse.org/Sirius/Tutorials/BasicFamily

Étape 7 : Génération de texte avec Acceleo

But : Génération de texte à partir des données du modèle en utilisant Accelo.
Description : Dans cette étape nous allons générer du texte à partir des instances de notre modèle. Nous utiliserons pour cela un projet de la fondation Eclipse appelé Acceleo. Ce projet fournit un langage pour écrire des templates. Les instances du modèle sont alors injectées dans ce template pour obtenir un texte conforme à ce template.

Réalisation :

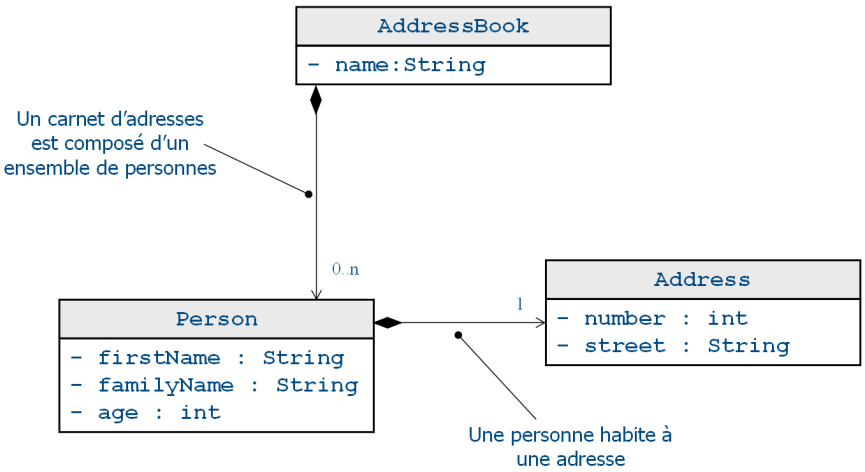
1. Démarrer un nouveau projet de type Acceleo Project (**New -> Other -> Acceleo Model to Text -> Acceleo Project**) ;
2. Pour le nom du projet saisir la valeur *fr.ensma.idm.projet.familydsl.acceleo* puis faire **Finish** ;
3. Créer depuis ce nouveau projet un répertoire *model* puis copier le fichier modèle EMF *famillemm.ecore* et le fichier instance *FamilleTest.arbregen* créé à l'étape 3 ;
4. Créer un nouveau module Acceleo (**New -> Other -> Acceleo Model to Text -> Acceleo Module File**), dans le champ *Module Name* placer la valeur *vcardsGenerate*, dans le champ *Metamodel URIs* chercher l'URI du modèle Ecore http://www.example.org/arbregen (effectuer la recherche en mode Runtime Version), dans le champ *Type* choisir la classe englobante `Famille`, puis finalement cocher les cases *Generate File* et *Main Template* puis faire **Finish** ;
5. Editer le fichier de template appelé *generate.mtl* de façon à générer pour chaque personne de l'arbre (âgée de 18 ans au moins) un fichier Vcard (inspirez-vous de l'extrait du template ci-dessous) ;

```
[comment encoding = UTF-8 /]
[module generate('http://addressbook/1.0')]

[template public generateElement(uneFamille : Famille)]
[comment @main/]
[for (current : Person | uneFamille.membres)]
   [file (current.nom.concat(('.vcard')), false, 'UTF-8')]
Création d'un VCards pour la person [current.nom/]
   [/file]
[/for]
[/template]
```

6. Créer une configuration d'exécution basée sur une application Acceleo que vous nommerez *ArbreGenAcceleoConfiguration*. Dans le champ *Project* choisir le projet *fr.ensma.idm.projet.familydsl.acceleo,* le champ *Main Class fr.ensma.idm.projet.familydsl.acceleo.VcardsGenerate,* le *champ Model /fr.ensma.idm.projet.familydsl.acceleo/model/ FamilleTest.arbregen* et finalement le champ Target */fr.ensma.idm.projet.familydsl.acceleo/model*;
7. Exécuter et vérifier que le contenu généré correspond à un VCard.

Étape 8 : Génération d'un carnet d'adresses avec ATL

But : transformation des données vers une autre structure d'un autre méta-modèle
Description : Dans cette étape nous allons transformer les des instances de notre modèle afin d'obtenir un carnet d'adresses. Nous utiliserons pour cela un projet de la fondation Eclipse appelé ATL. Ce projet fournit un langage pour écrire des règles de transformation Model-to-Model. Les instances du modèle sont alors soumises aux règles de transformation pour obtenir des instances conformes au méta-modèle suivant.



Réalisation :

1. Installer le plugin ATL à travers le marketplace ;
2. Créer un EMF projet pour mettre en place le méta-modèle *addressbook.ecore*. Choisir comme nom de projet *fr.ensma.idm.projet.addressbook* Un carnet d'adresses est identifié par un nom et contient une liste de personnes (*contains*). Une personne est identifiée par un prénom, un nom et un âge. Une personne contient obligatoirement une adresse « location ». Une adresse est identifiée par un numéro de rue et un nom de rue.
3. Créer un projet du type «*ATL Project* » et choisir comme nom *fr.ensma.idm.projet.familydsl.design*
4. Au sein du projet ATL, créer un fichier « ATL file » afin d'implémenter les règles de transformation permettant d'effectuer la transformation souhaitée.