TECHNISCHE UNIVERSITÄT
ILMENAU

Ecore for MDE4CPP

Tutorial

For Windows

Table of contents

1. Introduction 2
   1.1. Content 2
   1.2. Reference Model 3
2. Tutorial: Creating an Ecore Model using Code 4
   2.1. Creating the project 4
   2.2. Creating Ecore-Elements using the Factory 6
   2.3. Creating Ecore-Elements using Metaclass Names 8
   2.4. Creating Ecore-Elements using Classifier IDs 10
   2.5. An Output-Method for the model 12
   2.6. Compiling the project 13
3. Tutorial: Creating an Ecore Model using EMF 15
   3.1 Creating the project 15
   3.2 Creating our Ecore-Model in EMF 17
   3.3 Writing a main.cpp for the project 21
   3.4 Generating and compiling the model code 23
4. Tutorial: Ecore Model Behavior 26
   4.1 Adding custom methods to the model 26
   4.2 Including libraries for custom methods 28
   4.3 Testing the final model 30
5. Tutorial: MDE4CPP Plugin Framework for Ecore 32
   5.1 Creating the project 32
   5.2 Including and using the model plugin 34
   5.3 Compiling the project 38

1. Introduction

The objective of the MDE4CPP project is to exploit the chances and opportunities of Model Driven Engineering (MDE) for the development of software using the programming language C++.

The idea behind MDE is to close the gap between the model level and the source code level of software development by using domain specific models for automated generation of source code during the whole development process. By that, increasingly extensive software systems which implement more and more complex issues can be abstracted and become more understandable and tangible for software developers. As a result, the amount of sources of errors during the development process should be reduced. Moreover, software could be created much faster, cheaper, more efficient and of higher quality.

1.1. Content

At the moment there are mostly Java based tools available for model driven software development. The MDE4CPP framework provides a set of tools which (in combination with the *Eclipse Modelling Framework* by the Eclipse Foundation) enables a model driven development process with C++.

Currently, the following (meta) models are available:

- *ecore* (metamodel by the Eclipse Foundation)
- *UML* (unified modelling language by the OMG)
- *fUML* (foundational UML, standardized executable UML by the OMG)

This document is an introduction to working and developing ecore models with the MDE4CPP framework. It provides a reference model (see section 1.2) which will be the basis for the step-by-step instructions on the functioning and workflow of the MDE4CPP framework (starting with creating a project through creating models, all the way to source code generation and compilation).

Important Notes:

1. In order to be able to follow this tutorial, a fully functioning installation of MDE4CPP is required. For a detailed setup and installation guide for MDE4CPP as well as all required software components, please see the setup and installation guide.

2. It is highly recommended to abide by the naming conventions (parameters, names, file/directory/project names, etc.) used in this document. Especially some identifiers that are relevant for code generation and/or compilation must have certain names.

3. Complete versions of all tutorial projects (tested and working) can be found in the directory tutorialProjects.

1.2. Reference Model

The following figure shows the reference model for the ecore tutorials in this document in the form of a class diagram.

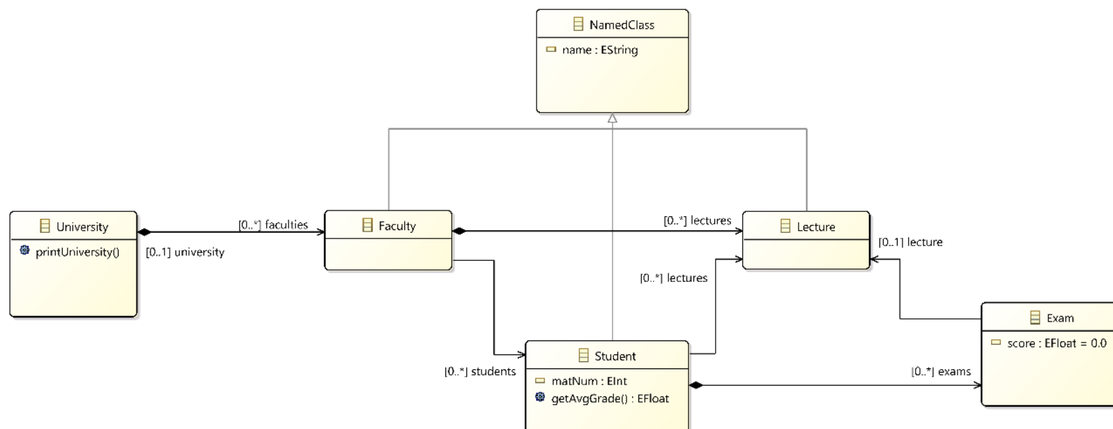


Fig. 1: Reference model "University Model"

It shows a typical university scenario. The model elements are:

- classes like “University”, “Student”, “Faculty”, etc.
- class attributes like the name of a lecture or the matriculation number of a student
- user defined class operations
- relationships like generalization and associations with different multiplicities

2. Tutorial: Creating an Ecore Model using Code

In this tutorial we will create ecore model elements (classes, attributes, references, etc.) via C++-Code. In MDE4CPP there are three different possibilities to create instances of ecore model elements: using the *EcoreFactory*, using the metaclass name or using the classifier ID.

The ecore metamodel contains elements like *EPackage*, *EFactory*, *EClass*, *EAttribute*, *EAnnotation*, *EReference*, *EDataType*, and many more. For further information on model elements and their relations, see the ecore documentation.

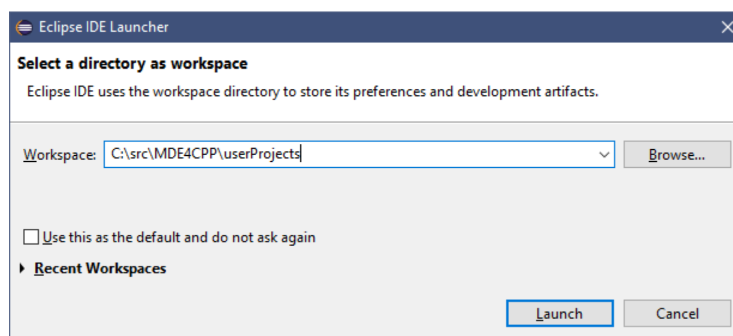
Because creating an ecore model only using code will result in lots of code, we will just create a small part of the reference model in this tutorial.

Hint: For help, find the complete and finished project here.

2.1. Creating the project

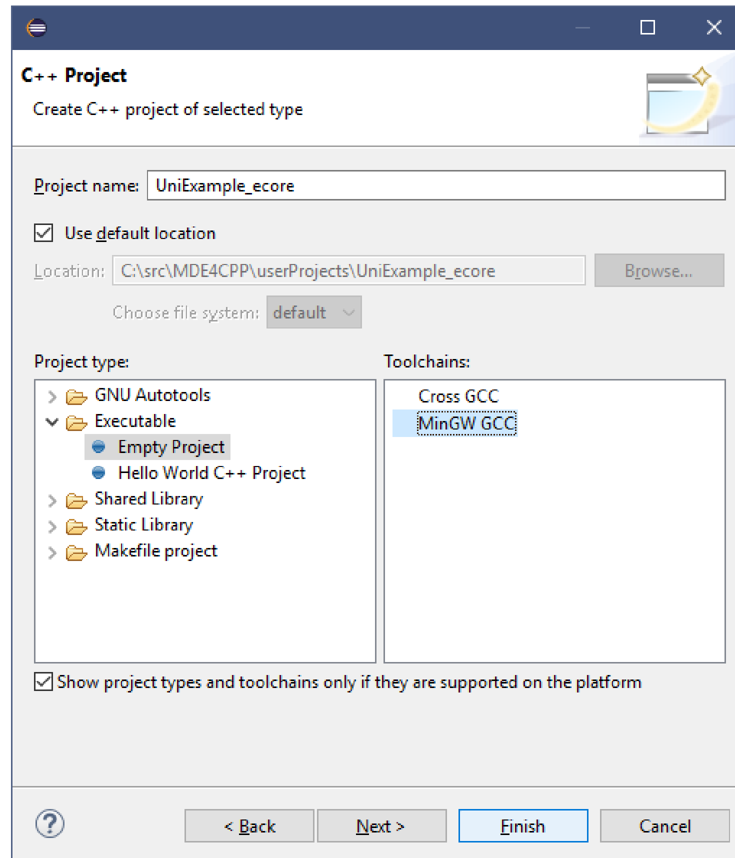
Step 1:

First we need to create the project. We will use an *Eclipse* IDE for developing C/C++-Applications, which you can download here. Extract the directory and start *Eclipse*. Choose *%MDE4CPP_HOME%\userProjects* as workspace directory, where %MDE4CPP_HOME% is the directory you installed MDE4CPP to. Click ***Launch***.



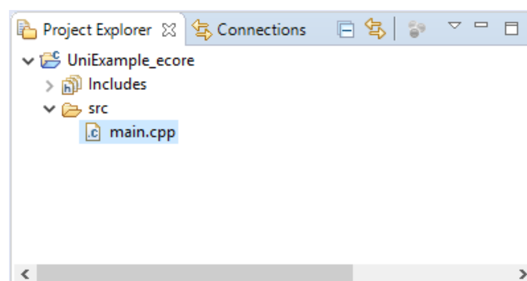
Step 2:

From the menu bar choose ***File → New → C/C++-Project***. Choose the ***C++ Managed Build*** template and click ***Next***. Name the project "*UniExample_ecore*", choose "***Empty Project***" as project type and "***MinGW GCC***" as compiling toolchain. Click ***Finish***.



Step 3:

Inside the project explorer right-click ***UniExample_ecore***, then choose ***New → Folder*** and create a folder called "*src*" inside your project. After creating the folder, right-click it and choose ***New → Source File*** and create the file "*main.cpp*" inside the source folder. The project-tree inside the project explorer should look like this:



Hint: After creating the project using *Eclipse* you can also edit the file "*UniExample_ecore\src\main.cpp*" using any other text editor, source code editor, etc. (e.g. *Notepad++*) from now on, if you wish.

2.2. Creating Ecore-Elements using the Factory

The *EcoreFactory* class provides operations to instantiate elements of the ecore metamodel (e.g. *EcoreFactory->createEClass*() returns an instance of *std::shared_ptr*<*ecore::EClass*>). You can find the full implementation of the *EcoreFactory* class in the file

%MDE4CPP_HOME%\src\ecore\ecore\src_gen\ecore\impl\EcoreFactoryImpl.cpp
where %MDE4CPP_HOME% is the directory you installed MDE4CPP to.

Step 1:

First we need the code 'frame' (required ecore libraries includes, namespace declarations, *main()* function). Copy the following code to the (empty) ***main.cpp*** file of the ***UniExample_ecore*** project:

```
// Include standard libraries
#include <iostream>

// Include header files for ecore
#include "ecore/EcoreFactory.hpp"
#include "ecore/EcorePackage.hpp"

using namespace ecore;
using namespace types;

int main()
{
        return 0;
}
```

Step 2:

Before we can create any model elements, we need instances of *EcorePackage* and *EcoreFactory* (Package and Factory for the ecore metamodel). Add the following first instantiations to your *main()* function:

```
// Create an instance of an ecore package and an ecore factory
// (both are needed to create an ecore model)
std::shared_ptr<EcorePackage> ecorePackage = EcorePackage::eInstance();
std::shared_ptr<EcoreFactory> ecoreFactory = EcoreFactory::eInstance();
```

Step 3:

Now we can use the ***ecoreFactory*** object to create instances of *ecore* model elements. First we need a root/main package that will later contain our model classes. We will name it “UniExample_ecore”.

```
// Create an instance of EPackage (using the ecore factory)
std::shared_ptr<EPackage> rootPackage = ecoreFactory->createEPackage();
rootPackage->setName("UniExample_ecore");
```

As an example, let's create an *EClass* object that will represent the *Student* class (see section 1.2). First include the required header in your ***main.cpp*** file.

```
#include "ecore/EClass.hpp"
```

By calling the *createEClass_in_EPackage()* function, the *EClass* instance is automatically added to the root package.

```
// Create an instance of EClass (which will become our UniModel classes)
// in root package that we just created (using ecore factory)
std::shared_ptr<EClass> eClass_student =
ecoreFactory->createEClass_in_EPackage(rootPackage);
```

Further we create an instance of *EAttribute* (e.g. matriculation number of a student) and assign an *Integer* data type to it. First include the required headers in your ***main.cpp*** file.

```
#include "ecore/EAttribute.hpp"
#include "ecore/EDataType.hpp"
#include "types/TypesPackage.hpp"
```

By calling the *createEAttribute_in_EContainingClass()* function, the *EAttribute* instance is automatically added to our student *EClass* object.

```
// Create an instance of EAttribute for the EClass that we just created
// (using the ecore factory)
std::shared_ptr<EAttribute> eAttribute_student_matNum =
ecoreFactory->createEAttribute_in_EContainingClass(eClass_student);
```

Before we can assign a data type, we need to create an instance of *TypesPackage*. We use the *TypesPackage* object to get an *EDataType* object representing an Integer type.

```
std::shared_ptr<TypesPackage> typesPackage = TypesPackage::eInstance();
//Assign data type "Integer" to the matriculation number EAttribute
eAttribute_student_matNum->
setEType(typesPackage->getInteger_Class());
```

Step 4:

Finally, let's set some properties for our model elements. For example we could set names regarding the reference model.

```
// Set properties of our ecore model elements
eClass_student->setName("Student");
eAttribute_student_matNum->setName("matNum");
```

Hint: There are many more properties that can be manipulated (depending on the type of model element). For more information, see the ecore documentation and select a page about a specific model element.

You can find the implementation files of the model elements (some Setter functions may be found in super classes of model elements) in directory

%MDE4CPP_HOME%\src\ecore\ecore\src_gen\ecore\impl
where %MDE4CPP_HOME% is the directory you installed MDE4CPP to.

2.3. Creating Ecore-Elements using Metaclass Names

We will continue to create our university model by creating instances of model elements using the name of their metaclasses. The *EcoreFactory* class provides an implementation of its *create()* method, which we pass a parameter of type *String*. Respectively the name of a metaclass (e.g. “EClass”, “EAttribute”, … as *Strings*).

The basic idea is to instantiate an instance of *EObject*[1] using the name of a metaclass, then casting the *EObject* to an instance of the actual subclass (model element) that we wanted to create.

Step 1:

First include the required header in your ***main.cpp*** file.

```
#include "ecore/EStringToStringMapEntry.hpp"
```

Create an instance of *EObject* with metaclass name “EClass” and declare the actual *EClass* object (let’s create the *Faculty* class) that will be instantiated later.

```
std::shared_ptr<EObject> eObject_1 = ecoreFactory->create("EClass");

//Forward declaration
std::shared_ptr<EClass> eClass_faculty;
```

Step 2:

We need to check if our *EObject* instance was created successfully. If the *create()* function is passed an unknown metaclass name, then the ***eObject_1*** object will be ***NULLPTR***. E.g. *ecoreFactory->create(“someUnknownMetaClass”)* will return ***nullptr***.

If ***eObject_1*** was instantiated successfully, we can use it to instantiate the ***eClass_faculty*** object. We perform the same null pointer check on our *EClass* instance.

```
if (eObject_1 != nullptr) {

	// Cast eObject_1 to an EClass object
	eClass_faculty = std::dynamic_pointer_cast<EClass>(eObject_1);
	if (eClass_faculty != nullptr) {

		eClass_faculty->setName("Faculty");

		// !!Code for Steps 3 and 4!!

	}
}
```

[1] *EObject* is (similar to the *Object* class in Java) the root super class of all ecore model elements

Step 3:

Because we have two different classes in our model now, we can create a reference between students and lectures. First include the required header in your ***main.cpp*** file.

```
#include "ecore/EReference.hpp"
#include "abstractDataTypes/SubsetUnion.hpp"
```

In this case we will create a “hasStudents”-like reference in our newly created ***eClass_faculty*** object using *EReference’s* metaclass name.

```
//Creating an EReference via EObject using metaclass name
std::shared_ptr<EObject> eObject_2 = ecoreFactory->create("EReference");

//Forward declaration
std::shared_ptr<EReference> eReference_faculty_student;


        if(eObject_2 != nullptr){
               // Cast eObject_2 to a new EReference instance
               eReference_faculty_student =
               std::dynamic_pointer_cast<EReference>(eObject_2);

               if(eReference_faculty_student != nullptr){

                    eReference_faculty_student->setName("students");

                    //Reference will be of type "Student"
                    eReference_faculty_student->setEType(eClass_student);

                    //-1 is used to set a [0..*] multiplicity
                    eReference_faculty_student->setUpperBound(-1);

                    //Add our new reference to our lecture class
                    //In MDE4CPP Bag<> is used as container class
                    std::shared_ptr<Bag<ecore::EReference> > references =
                    eClass_faculty->getEReferences();
                   references->add(eReference_faculty_student);
               }
        }
```

Step 4:

Finally, we add the *Faculty* class to our root package. First include the required header in your ***main.cpp*** file.

```
#include "ecore/EClassifier.hpp"
```

The following code adds the ***eClass_faculty*** object to our model’s root package.

```
// Add our new class to our package
//In MDE4CPP Bag<> is used as container class
std::shared_ptr<Bag<ecore::EClassifier> > classifiers =
rootPackage->getEClassifiers();

classifiers->add(eClass_faculty);
```

2.4. Creating Ecore-Elements using Classifier IDs

The third possibility to create an ecore model element is using its metaclass's ID. The basic idea is to retrieve the element's metaclass information and creating an instance of it by the metaclass ID (integer value).

Step 1:

First we will create an *EClass* object using the ***eObject_1*** object from section 2.1.3. Using the *EObject->eClass()* method, we can retrieve the metaclass information of an *EObject* instance (remember that ***eObject_1*** was instantiated using the metaclass name of *EClass*).

```
// Since eObject_1 was instantiated by metaclass name "EClass"
// this will return metaclass information for EClass
std::shared_ptr<EClass> eClass_metaclass = eObject_1->eClass();
```

Step 2:

We can create a new *EObject* instance by retrieving the *ClassifierID* parameter of our new ***eClass_metaclass*** object. Then we will cast the new *EObject* instance to a new *EClass* instance that we actually want to create.

```
// Create a new instance of EObject using the classifier ID of
// metaclass EClass
std::shared_ptr<EObject> eObject_3 =
ecoreFactory->create(eClass_metaclass->getClassifierID());

//Forward declaration
std::shared_ptr<EClass> eClass_exam;
```

Because the classifier ID could be invalid (which would result in a null pointer object), we have to do the same check like in section 2.1.3 before casting our instances and creating a new *EClass* object.

```
if (eObject_3 != nullptr) {
       // Cast eObject_3 to a new EClass instance
       eClass_exam = std::dynamic_pointer_cast<EClass>(eObject_3);

          if (eClass_exam != nullptr) {
                    eClass_exam->setName("Exam");

                    // !!Code for Steps 3 and 4
          }
      }
```

Step 3 (optional):

We will add some attributes and references (to output them later) to our model using the *EcoreFactory* (see section 2.1.2). This step is not necessary for further steps.

```
// Create an additional attribute in our new class
std::shared_ptr<EAttribute> eAttribute_exam_score =
ecoreFactory->createEAttribute_in_EContainingClass(eClass_exam);

eAttribute_exam_score->setName("score");
//Assign data type "Real"(=Float) to the exam score EAttribute
eAttribute_exam_score->setEType(typesPackage->getReal_Class());


//Add an additional EClass
std::shared_ptr<EClass> eClass_lecture =
ecoreFactory->createEClass_in_EPackage(rootPackage);
eClass_lecture->setName("Lecture");

// Add an additional reference between Exam class and Lecture class
std::shared_ptr<EReference> eReference_exam_lecture =
ecoreFactory->createEReference_in_EContainingClass(eClass_exam);
eReference_exam_lecture->setName("lecture");
// Reference will be of type "Lecture"
if(eClass_lecture != nullptr){
eReference_exam_lecture->setEType(eClass_lecture);}

//Add another reference between "Faculty"-class and "Lecture"-class
std::shared_ptr<EReference> eReference_faculty_lecture =
ecoreFactory->createEReference_in_EContainingClass(eClass_faculty);
eReference_faculty_lecture->setName("lectures");
eReference_faculty_lecture->setUpperBound(-1);
// Reference will be of type "Lecture"
if(eClass_student != nullptr){
eReference_faculty_lecture->setEType(eClass_lecture);}
```

Step 4:

Finally, we add the *Exam* class to our root package.

```
// Add our new class to our package
//In MDE4CPP Bag<> is used as container class
std::shared_ptr<Bag<ecore::EClassifier> > classifiers =
rootPackage->getEClassifiers();

classifiers->add(eClass_exam);
```

2.5. An Output-Method for the model

The following code will print a sample output for the created model. You can either use the sample code or create your own output method.

Hint: In MDE4CPP *Bag*<> is used as container class.

```
//Output contents of our model package
    std::cout<<std::endl<<"-" << rootPackage->getName()
    << " package content-"<<std::endl<<"*************"<<std::endl;
//Create container of all classifiers of our package
    std::shared_ptr<Bag<ecore::EClassifier> > classifiers =
    rootPackage->getEClassifiers();

    Bag<ecore::EClassifier>::const_iterator endIt = classifiers->end();
//Loop over all classifiers of the package
for (Bag<ecore::EClassifier>::const_iterator it = classifiers->begin();
    it != endIt; ++it) {
    std::cout << "class name: " << (*it)->getName() << std::endl;

    //Cast iterator to EClass
    std::shared_ptr<EClass> currentClassifier =
    std::dynamic_pointer_cast<EClass>(*it);

    std::cout << "  Features:"<< std::endl;
    //Create container of all features (attributes, references, ...) of the
    //current classifier
    std::shared_ptr<Bag<ecore::EStructuralFeature> > structuralFeatures =
    currentClassifier->getEAllStructuralFeatures();

    Bag<ecore::EStructuralFeature>::const_iterator endItFeature =
    structuralFeatures->end();
    //Loop over all features of the current classifier

    for    (Bag<ecore::EStructuralFeature>::const_iterator itFeature =
           structuralFeatures->begin(); itFeature != endItFeature;
           ++itFeature) {

           std::string featName = (*itFeature)->getName();
           // eClass() returns the metaclass information of eClass_student,
           // eClass()->getName() returns the name of the metaclass
           std::string metaClassName = (*itFeature)->eClass()->getName();
           // in case the current feature does not have a type
           std::string typeName = "unknown";

           int upperBound = (*itFeature)->getUpperBound();
           int lowerBound = (*itFeature)->getLowerBound();

           std::shared_ptr<ecore::EClassifier> featClassifier =
           (*itFeature)->getEType();

           if(featClassifier != nullptr) {typeName =
           featClassifier->getName();}

           std::cout << "       - '"<< featName << "': " << typeName << "["
           << lowerBound << ".." << upperBound <<"]\t\tof metaclass " <<
           metaClassName << std::endl;

    }
    std::cout << "_____________" << std::endl;
}


std::cout << "Finished..." << std::endl;
```

2.6. Compiling the project

To be able to compile the project that you created in this tutorial, two more files need to be added to your project:

- a *CMakeLists*-File (required for *CMake*)
- a *GRADLE*-File (required for *Gradle*)

For this tutorial you can use prepared files and copy them to your project. Important parameters inside these files are described with comments in case you want to reuse and adapt them for your own projects.

Step 1:

Copy the prepared *CMakeLists.txt* file for this project to

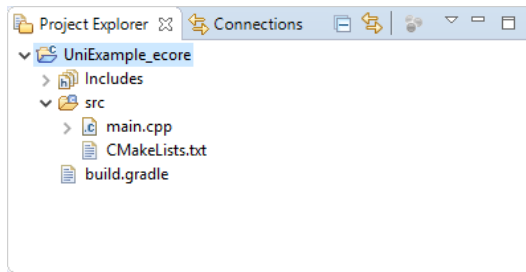
%MDE4CPP_HOME%\userProjects\UniExample_ecore\src
where %MDE4CPP_HOME% is the directory you installed MDE4CPP to.

Step 2:

Copy the prepared *build.gradle* file for this project to

%MDE4CPP_HOME%\userProjects\UniExample_ecore
where %MDE4CPP_HOME% is the directory you installed MDE4CPP to.

After step 1 and step 2, your project folder should now look like this:



Step 3:

Navigate to your %MDE4CPP_HOME% directory and open up a command prompt. Type in the following commands:

setenv.bat

gradlew tasks

Under task group ***UniExample_ecore tasks*** you should find a task called ***compileUniExample_ecore***.

```
C:\WINDOWS\system32\cmd.exe
UmlReflection tasks
-------------------
buildUmlReflection - build umlReflection
compileUmlReflection - compile umlReflection
generateUmlReflection - generate C++ code of umlReflection.uml model

UniExample_ecore tasks
----------------------
compileUniExample_ecore - compiles uniExample_ecore

Util tasks
----------
deliverUtil - deliver util header to %MDE4CPP_HOME%/application/include/util

Verification tasks
------------------
check - Runs all checks.
test - Runs the unit tests.

To see all tasks and more detail, run gradlew tasks --all

To see more detail about a task, run gradlew help --task <task>

BUILD SUCCESSFUL in 15s
1 actionable task: 1 executed
C:\src\MDE4CPP>_
```

Step 4:

Execute the task by typing in the following command:

gradlew compileUniExample_ecore

Your project will be compiled. When the compilation process has finished, navigate to the directory:

%MDE4CPP_HOME%\application\bin

Where %MDE4CPP_HOME% is the directory you installed MDE4CPP to. You should find the files "*App_uniExample_ecore.exe*" (compiled with release options) and "*App_uniExample_ecored.exe*" (compiled with debug options). You might also find just one of those files depending on how you configured the debug/release options in the file "*%MDE4CPP_HOME%\setenv.bat*".

```
C:\WINDOWS\system32\cmd.exe

C:\src\MDE4CPP\application\bin>App_uniExample_ecore.exe

-UniExample_ecore package content-
************
class name: Student
        Features:
                - 'matNum': Integer[0..1]               of meta class EAttribute
____________
class name: Faculty
        Features:
                - 'students': Student[0..-1]            of meta class EReference
                - 'lectures': Lecture[0..-1]            of meta class EReference
____________
class name: Lecture
        Features:
____________
class name: Exam
        Features:
                - 'score': Real[0..1]           of meta class EAttribute
                - 'lecture': Lecture[0..1]              of meta class EReference
____________
Finished...

C:\src\MDE4CPP\application\bin>
```

3. Tutorial: Creating an Ecore Model using EMF

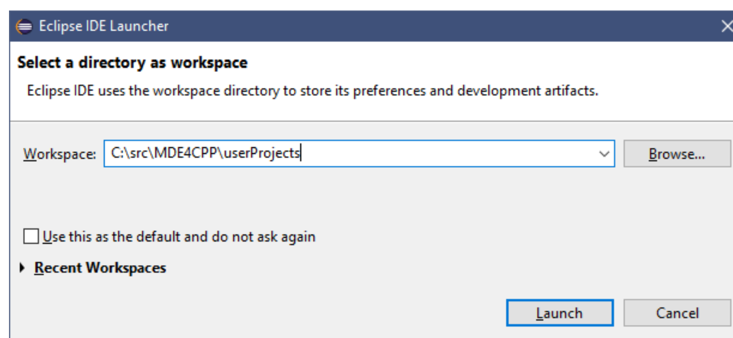
In this tutorial we will create an ecore model of our reference model (see section 1.2) using the *Eclipse Modeling Framework (EMF)*. Then we will generate and compile the model code of the created ecore model using MDE4CPP.

Hint: For help, find the complete and finished project here.

3.1 Creating the project

Step 1:

First we need to create the project. Open the *Eclipse Modeling Tools* and set the workspace directory. Choose ***%MDE4CPP_HOME%\userProjects*** as workspace directory, where %MDE4CPP_HOME% is the directory you installed MDE4CPP to. Then click ***Launch***.



Step 2:

From the menu bar choose ***File → New → Other → General → Project → Next***. Name the project "*UniModelExample_ecore*" and click ***Finish***.

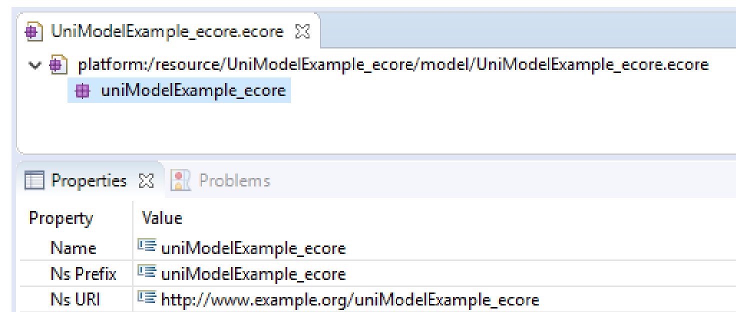
Right-click on the newly created project and choose ***New → Folder***. Name the new folder "*model*" and click ***Finish***.

Right-click the ***model*** folder and choose ***New → Other → Eclipse Modeling Framework → Ecore Model*** and click ***Next***. Name the new ecore model "*UniModelExample_ecore.ecore*" and click ***Finish***.

Step 3:

Now we will set the properties for our *EPackage*. Double click on the newly created ecore model to open it in the editor. Unfold the model and select the (nameless) *EPackage*. In the ***Properties View***, set:

- "*uniModelExample_ecore*" as ***Name***
- "*uniModelExample_ecore*" as ***Ns Prefix***
- "*http://www.example.org/uniModelExample_ecore*" as ***Ns URI***

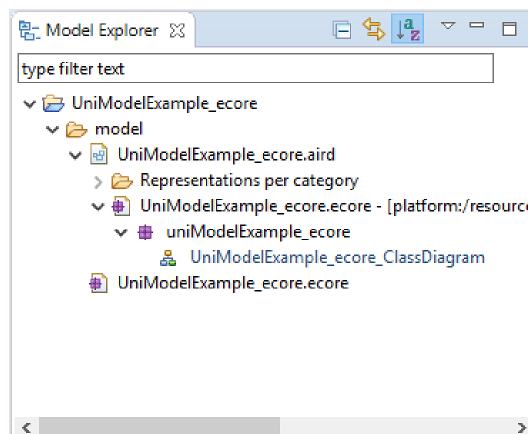


If you cannot see the ***Properties View***: From the menu bar choose ***Window → Show View → Properties***.

Step 4:

We will now add a class diagram of our ecore model to the project. Right-click on the ecore model and choose ***Initialize Ecore Diagram …***, name the new representation file "*UniModelExample_ecore.aird*". Click ***Next***, choose ***Entities in a Class Diagram*** and click ***Next***. Name the representation "*UniModelExample_ecore_ClassDiagram*" and click ***Finish***.

The project-tree inside the project explorer should look like this:

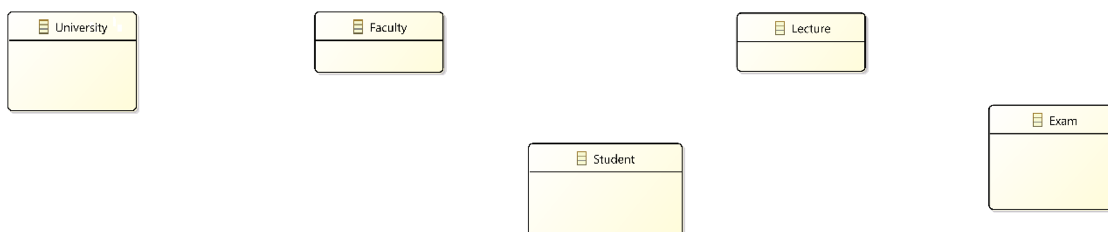


3.2 Creating our Ecore-Model in EMF

Step 1:

Double click the model class diagram to open it in the editor. From the ***Palette*** select and drag a new *Class* into the editor to create a new model class in your ecore model. To change properties of a model class you can either use the properties view or open a new properties dialog by double clicking it.

Name your new class "*University*". Continue to create more model classes and name them "*Faculty*", "*Student*", "*Lecture*" and "*Exam*".

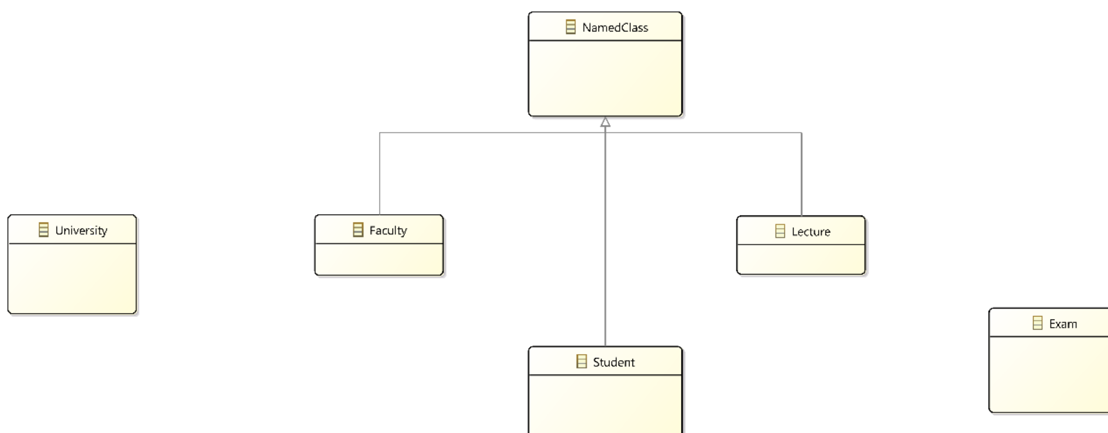


Step 2:

In modelling we can use a generalization relationship to generalize common properties of multiple subtypes in a single super type. As the classes ***Faculty***, ***Student*** and ***Lecture*** will have a common attribute (their names), we will introduce a super type called ***NamedClass***. First create the new class.

Now we can introduce the generalization relationships between the super type and the sub types. In the ***Palette*** under ***Relation → SuperType*** you can find the generalization relationship. Select it and connect the sub type and super type (for example, draw a connection between the ***Student*** class and the ***NamedClass*** super type).

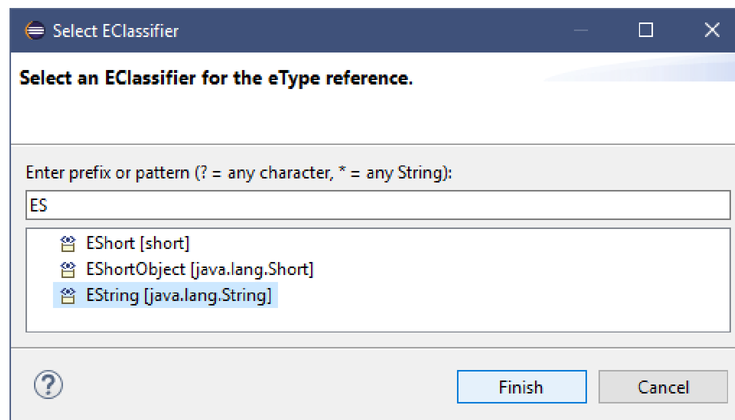
Continue with the generalizations of the ***Faculty*** and ***Lecture*** classes.



Step 3:

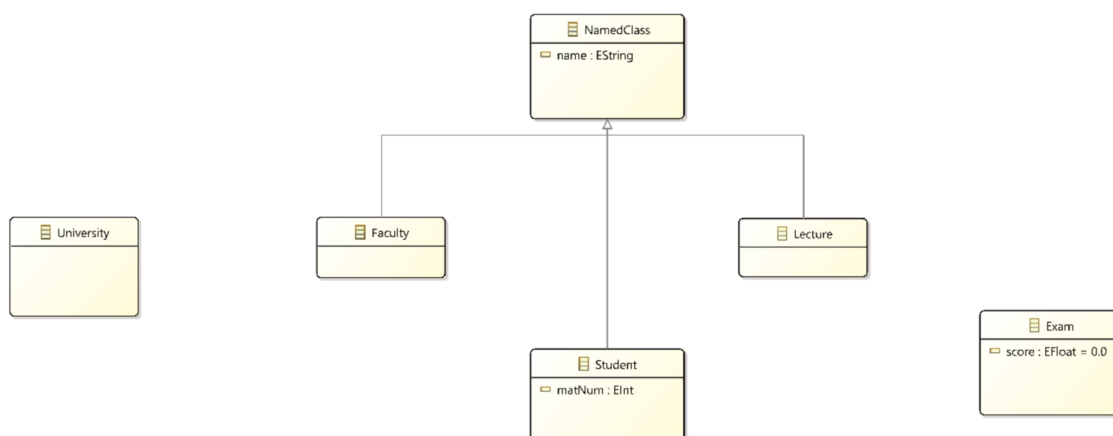
The super type does not have any attribute so far. In the ***Palette*** under ***Feature → Attribute*** you can find the attribute model element. Select it and drag it onto the ***NamedClass*** class. Name the new attribute "*name*".

We must now assign an *EType* (data type) to our attribute. Select the attribute, find the ***EType*** property in the ***Properties View → Ecore → EType*** and click the ***"…"-Button*** to add a data type. Choose *EString* as the data type.



Continue to add the following attributes to the model:

– ***matNum*** of type *EInt* to class ***Student***
– ***score*** of type *EFloat* to class ***Exam***

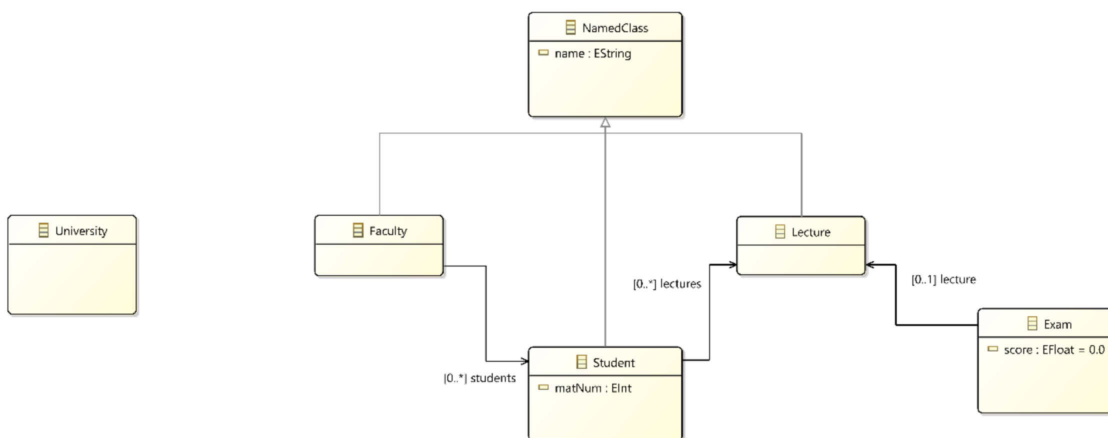


Step 4:

To model relationships between our model classes, we will add references to the model. First we will add simple references. For example, a faculty has multiple students. In the ***Palette*** under ***Relation → Reference***, find the reference model element. Select it and draw a reference between the ***Faculty*** class and the ***Student*** class. Name the reference "*students*" and set its *Upper Bound* property to "*" in the ***Properties View → Ecore***.

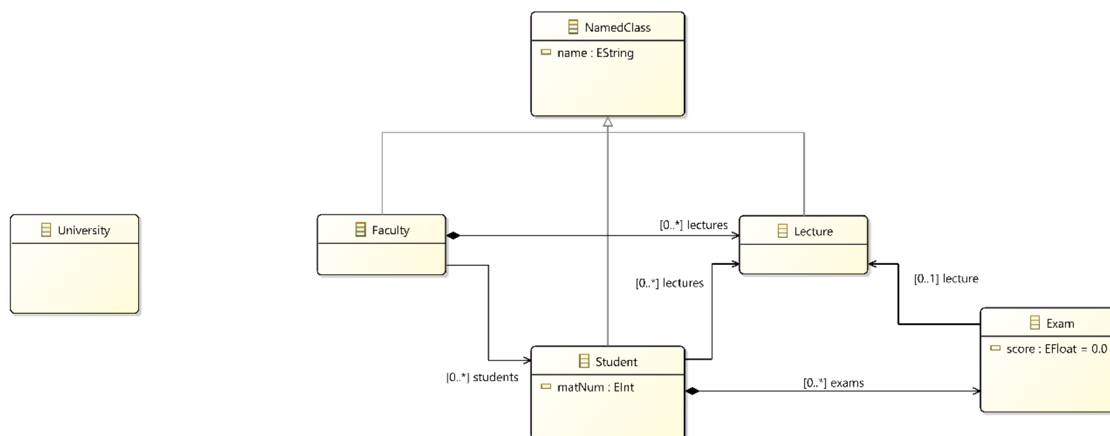
Continue to add the following references:

- ***lectures[0..*]*** reference between the classes ***Student*** and ***Lecture***
- ***lecture[0..1]*** reference between the classes ***Exam*** and ***Lecture***



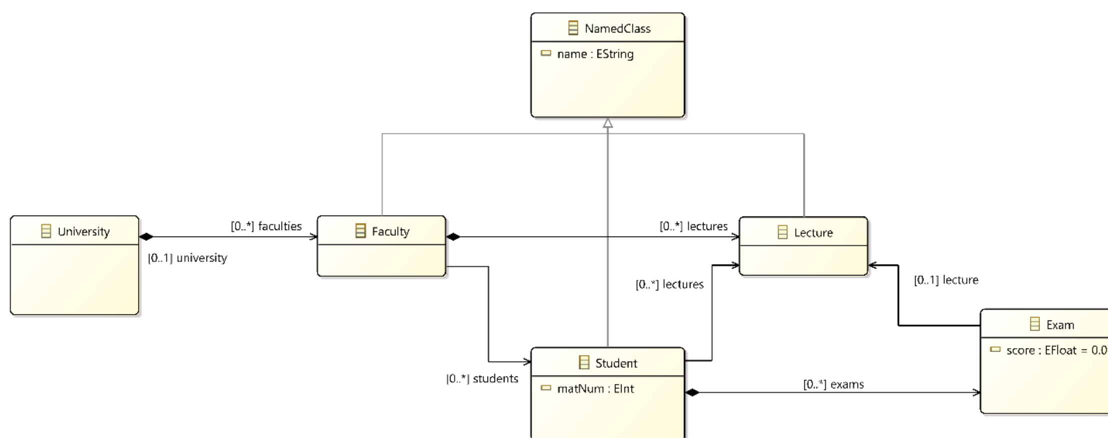
We will now introduce *Compositions* to our model. For example, an exam that is not taken by any student doesn't make sense. You can find the composition model element in the ***Palette*** under ***Relation → Composition***. Add the following compositions to the model:

- ***exams[0..*]*** between the classes ***Student*** and ***Exam***
- ***lectures[0..*]*** between the classes ***Faculty*** and ***Lecture***



In our reference model (see section 1.2) a university has multiple faculties but a faculty also has at most one university. To model this relationship, we will add a bidirectional reference to our model. In the ***Palette***, find the model element under ***Relation → Bi-directional Reference***.

Our bidirectional reference has two endpoints. Open the ***Properties View → Ecore***. Change to following properties for the ***Faculty*** endpoint: name the endpoint "*faculties*", set its *Upper Bound* property to "*" and check the *Containment* check-box (which makes the reference a composition for this endpoint).



3.3 Writing a main.cpp for the project

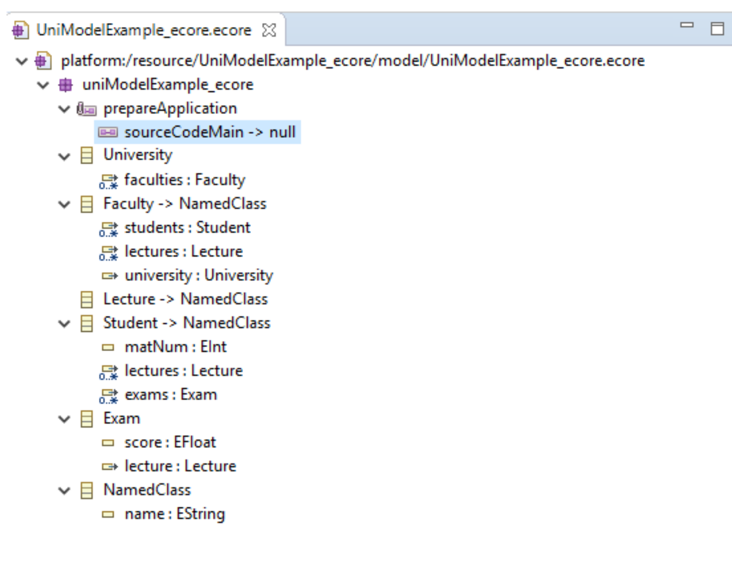
Before generating and compiling our model code, we will add a simple main.cpp (where we use our model) to the project.

Step 1:

Open the ***UniModelExample_ecore.ecore*** model by double-clicking it. Right-click the ***uniModelExample_ecore*** package and choose ***New Child → EAnnotation → EAnnotation***. In the ***Properties View*** choose the *Source* property and type in "*prepareApplication*".

Right-click the newly created *EAnnotation* and choose ***New Child → Details Entry***. In the ***Properties View*** choose the *Key* property and type in "*sourceCodeMain*".

Your model should now look like this:



Step 2:

We will now add the source code to our *main*()-function. We only have to add the function body. All other code (required includes, function declaration, instantiation of model package and factory objects) will automatically be added during code generation.

In the ***Properties View*** of the ***sourceCodeMain*** details entry choose the *Value* property. You can copy the following sample code, extend it and/or write your own *main*()-function.

```
//Creating a new instance of University via model factory
   std::shared_ptr<University> university =
   factory->createUniversity();
//Creating a new instance of Student via model factory
   std::shared_ptr<Student> student_1 = factory->createStudent();
//Creating a new instance of Faculty via model factory
   std::shared_ptr<Faculty> faculty_1 =
   factory->createFaculty_in_University(university);
//Creating two Lecture instances in Faculty 1 via model factory
   std::shared_ptr<Lecture> lecture_1 = factory->createLecture();
   std::shared_ptr<Lecture> lecture_2 = factory->createLecture();

//Setting parameters for Faculty 1
   faculty_1->setName("Computer Science");

//Setting parameters for Lectures 1 and 2
   lecture_1->setName("Software Design");
   lecture_2->setName("Model Driven Development");

//Setting parameters for Student 1
   student_1->setName("John Doe");
   student_1->setMatNum(12345);


//Output some data about our concrete university model
   std::cout<<"Student name, matr. num: "<<student_1->getName()
   <<", "<<student_1->getMatNum()<<std::endl;
   std::cout<<"Faculty name: "<<faculty_1->getName()<<std::endl;
   std::cout<<"Lecture names:\n\t"
   <<lecture_1->getName()<<"\n\t"<<lecture_2->getName()<<std::endl;
```

3.4 Generating and compiling the model code

Now we are ready to generate and compile the model code for the project.

Step 1:

Navigate to your %MDE4CPP_HOME% directory and open up a command prompt. Type in the following commands:

setenv.bat

gradlew generateModel -PModel=" %PATH_TO_MODEL%"

Where %PATH_TO_MODEL% is the complete path to our ***UniModelExample_ecore.ecore*** file. In our case this would be "*C:\src\MDE4CPP\userProjects\UniModelExample_ecore\model\UniModelExample_ecore.ecore*".

The ***generateModel*** task is a generic *Gradle* task for when you want to generate code from a model for the first time. It generates all "*.hpp*" and "*.cpp*" files and moreover creates new model/project specific *Gradle* tasks.

After successful code generation of the project, you can find:

- Header files and implementation files for all model elements at "*…\UniModelExample_ecore\src_gen\uniModelExample_ecore*"
- the ***main.cpp*** file (see section 2.2.3) at "*…\UniModelExample_ecore\application\src*"

Step 2:

When the ***generateModel*** task has finished successfully, type in the following command:

gradlew tasks

Under task group ***UniModelExample_ecore tasks*** you should find four new tasks.

```
C:\WINDOWS\system32\cmd.exe

UniExample_ecore tasks
---------------------
compileUniExample_ecore - compiles uniExample_ecore

UniModelExample_ecore tasks
--------------------------
buildUniModelExample_ecore - build uniModelExample_ecore
compileApplicationForUniModelExample_ecore - compile ApplicationForuniModelExample_ecore
compileUniModelExample_ecore - compile uniModelExample_ecore
generateUniModelExample_ecore - generate C++ code of uniModelExample_ecore.ecore model

Util tasks
----------
deliverUtil - deliver util header to %MDE4CPP_HOME%/application/include/util

Verification tasks
------------------
check - Runs all checks.
test - Runs the unit tests.

To see all tasks and more detail, run gradlew tasks --all

To see more detail about a task, run gradlew help --task <task>

BUILD SUCCESSFUL in 11s
1 actionable task: 1 executed
C:\src\MDE4CPP>
```

- the ***generateUniModelExample_ecore*** task will regenerate the model code in case the model was modified since the last code generation
- the ***compileUniModelExample_ecore*** task will create static libraries (*.hpp* files) as well as dynamic libraries (*.dll* files) of your model elements
- the ***compileApplicationForUniModelExample_ecore*** task will compile an application (*.exe* file) of your model
- the ***buildUniModelExample_ecore*** task will execute all of the three preceding tasks one after another

Step 3:

To execute the ***buildUniModelExample_ecore*** task, type in the following command:

gradlew buildUniModelExample_ecore

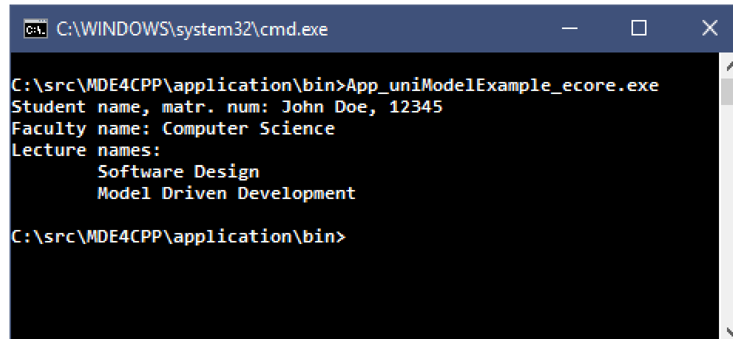
When the compilation process has finished, you can find:

- static libraries for your model at "*%MDE4CPP_HOME%\application\include\uniModelExample_ecore*"
- dynamic libraries ***uniModelExample_ecore.dll*** (compiles with release options) and ***uniModelExample_ecored.dll*** (compiled with debug options) for your models at "*%MDE4CPP_HOME%\application\bin*"
- applications ***App_uniModelExample_ecore.exe*** (compiled with release options) and ***App_uniModelExample_ecored.exe*** (compiled with debug options) for your model at "*%MDE4CPP_HOME%\application\bin*"

You might also find just one of those files each depending on how you configured the debug/release options in the "*%MDE4CPP_HOME%\setenv.bat*" file.

Step 4:

Navigate to "*%MDE4CPP_HOME%\application\bin*". When executing one of the compiled application files, you should see the following output (that was implemented in the *main*()-function of the model in section 2.2.3):



4. Tutorial: Ecore Model Behavior

In this tutorial we will extend the model that was created in section 2.2 and add custom user-defined functions to the model classes, as well as learn how to include project-specific and non-project-specific libraries.

4.1 Adding custom methods to the model

First we will add a simple function to the *Student* class of our model. The function will calculate the average grade of a student and return the result as a float type.

Step 1:

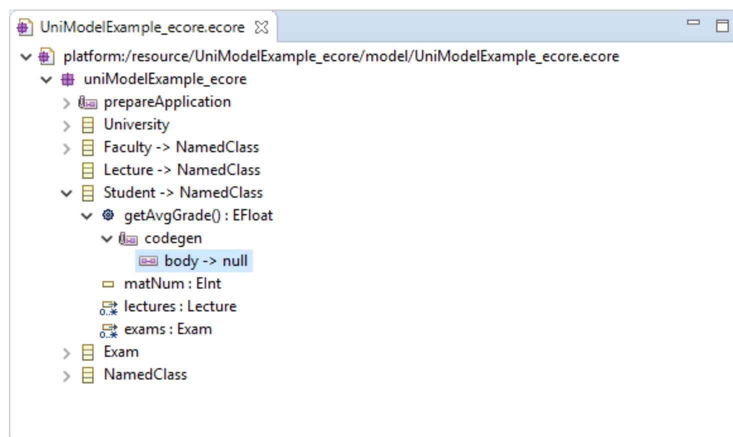
Open the ***UniModelExample_ecore*** project in the *Eclipse Modelling Framework* and open the ***UniModelExample_ecore.ecore*** model for editing. Right-click the ***Student*** class and choose ***New Child → EOperation***.

In the ***Properties View*** set the *Name* property to "*getAvgGrade*". Choose *EFloat* as *EType* property (this is the function's return type).

Right Click the newly created ***getAvgGrade()*** operation and choose ***New Child → EAnnotation → EAnnotation***. In the ***Properties View*** set the *Source* property to "*http://sse.tu-ilmenau.de/codegen*".

Right click the newly created *EAnnotation* instance and choose ***New Child → Details Entry***. Set the *Key* property to "*body*".

Your model should now look like this:



Step 2:

Now we have to insert the source code of our function body. Navigate to the *Value* property of the ***body*** entry and copy the following function code:

```
float avgGrade = 0;

//In MDE4CPP Bag<> is used as container class
//Get container of all Exams of student
std::shared_ptr<Bag<uniModelExample_ecore::Exam>> exams =
this->getExams();

//If there are no Exams for this student, return 0
if(exams->size() == 0){return avgGrade;}

//Loop over all Exam objects
//As we don't need to modify the container, we use const_iterator
for(
Bag<uniModelExample_ecore::Exam>::const_iterator it = exams->begin();
it!=exams->end();
it++)
{
	avgGrade += (*it)->getScore();
}

avgGrade /= getExams()->size();

return avgGrade;
```

4.2 Including libraries for custom methods

As an example, we will now implement the custom output function ***printUniversity()*** in the *University* class of our model.

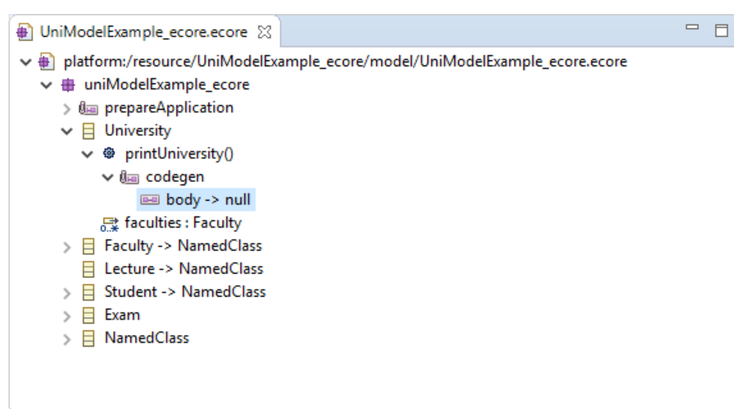
Step 1:

Open the ***UniModelExample_ecore.ecore*** model for editing. Right-click the ***University*** class and choose ***New Child → EOperation***.

In the ***Properties View*** set the *Name* property to "*printUniversity*". Leave the *EType* property as is, as the return type of this method will be *Void*.

Create an *EAnnotation* object and *Details Entry* for the ***printUniversity()*** function as explained in section 2.3.1.

Your model should now look like this:



Step 2:

Insert the following function body as explained in section 2.3.1.

```
//Get container of all faculties of university
std::shared_ptr<Bag<uniModelExample_ecore::Faculty>> faculties =
this->getFaculties();

//Loop over all faculties
for(Bag<uniModelExample_ecore::Faculty>::const_iterator it_fac =
faculties->begin();
it_fac != faculties->end();
it_fac++) {

  //Get container of all students in current faculty
  std::shared_ptr<Bag<uniModelExample_ecore::Student>> students =
  (*it_fac)->getStudents();

  //Loop over all students
  for(Bag<uniModelExample_ecore::Student>::const_iterator it_stud =
  students->begin();
  it_stud != students->end();
  it_stud++) {

       std::cout<<std::endl<<"Student " << (*it_stud)->getName()
       <<" of faculty " <<(*it_fac)->getName()<<std::endl;
```

```
            std::cout<<"He/She took the following courses:"<<std::endl;

            //Get container of all lectures of current student
            std::shared_ptr<Bag<uniModelExample_ecore::Lecture>> lectures =
            (*it_stud)->getLectures();

            //Loop over all lectures
            for(Bag<uniModelExample_ecore::Lecture>::const_iterator it_lec =
            lectures->begin();
            it_lec != lectures->end();
            it_lec++) {

                    std::cout<<"\t- "<<(*it_lec)->getName()<<std::endl;
            }

            std::cout<<"His/Her average grade is: "<<std::setprecision(2)
            <<(*it_stud)->getAvgGrade()<<std::endl;
        }

    }
```

Step 3:

In our reference model (see section 1.2) there is neither a direct reference between the *University* class and the *Student* class, nor the *Lecture* class. As we want to call member functions of both *Student* and *Lecture* inside the ***printUniversity()*** function, we have to include header files for both in the *University* class.

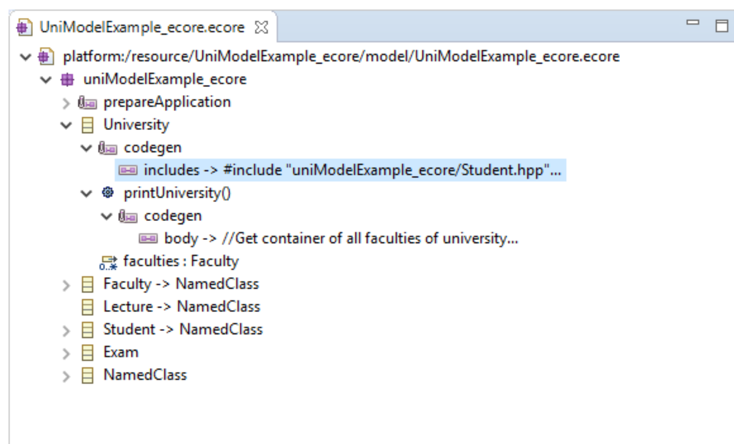
We are also using the ***std::setprecision()*** function which is implemented in the C++ standard header *iomanip*.

Right-click the ***University*** class of your model and create a new *EAnnotation* object as explained before. Create a new *Details Entry* for the newly created *EAnnotation* object and set its *Key* property to "*includes*". Insert the following includes into the *Value* property:

```
#include "uniModelExample_ecore/Student.hpp"
#include "uniModelExample_ecore/Lecture.hpp"
#include <iomanip>
```

Hint: In general, include directives for model class header files follow the pattern: *#include "PACKAGE_NAME/CLASS_NAME.hpp"*

Your model should now look like this:



4.3 Testing the final model

To test the final model, this tutorial provides the following extended example source code for the models *main*()-function. You can replace your model's current *main*()-function with the provided source code or implement your own source code for testing.

```
// Creating a new instance of University via model factory
std::shared_ptr<University> university = factory->createUniversity();

// Creating a new instance of Faculty via model factory
std::shared_ptr<Faculty> faculty_1 =
factory->createFaculty_in_University(university);

// Creating a new instance of Student via model factory
std::shared_ptr<Student> student_1 = factory->createStudent();

// Creating two Lecture instances in Faculty 1 via model factory
std::shared_ptr<Lecture> lecture_1 = factory->createLecture();
std::shared_ptr<Lecture> lecture_2 = factory->createLecture();

// Creating two Exam instances in Student 1
std::shared_ptr<Exam> exam_1 = factory->createExam();
std::shared_ptr<Exam> exam_2 = factory->createExam();



// Setting parameters for Faculty 1
faculty_1->setName("Computer Science");
faculty_1->getStudents()->add(student_1);

// Setting parameters for Student 1
student_1->setName("John Doe");
student_1->setMatNum(12345);
student_1->getLectures()->add(lecture_1);
student_1->getLectures()->add(lecture_2);
student_1->getExams()->add(exam_1);
student_1->getExams()->add(exam_2);

// Setting parameters for Lectures 1 and 2
lecture_1->setName("Software Design");
lecture_2->setName("Model Driven Development");

// Setting parameters for Exams 1 and 2
exam_1->setLecture(lecture_1);
exam_1->setScore((float)1.7);
exam_2->setLecture(lecture_2);
exam_2->setScore((float)1.0);



// Custom output function printUniversity()
university->printUniversity();
```

After rebuilding the project (see section 2.2.4) and executing the compiled application, you should get the following output:

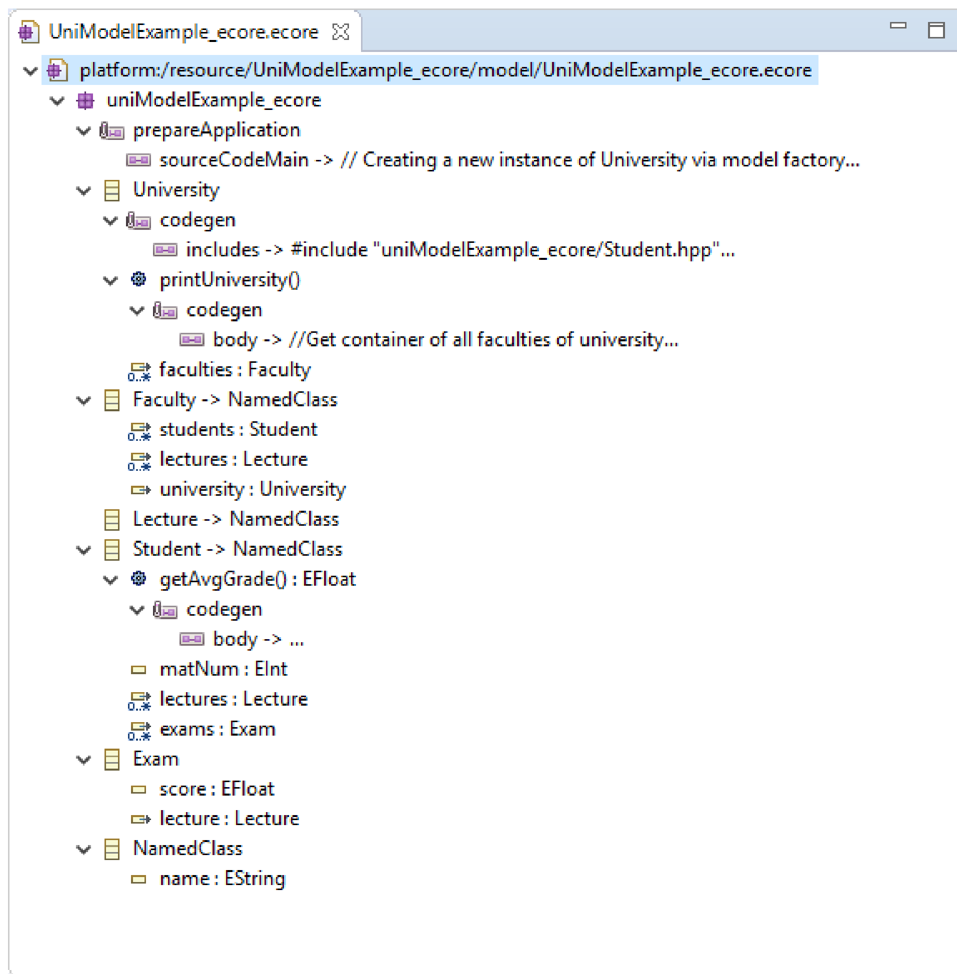
```
C:\WINDOWS\system32\cmd.exe

C:\src\MDE4CPP\application\bin>App_uniModelExample_ecore.exe

Student John Doe of faculty Computer Science
He/She took the following courses:
        - Software Design
        - Model Driven Development
His/Her average grade is: 1.4

C:\src\MDE4CPP\application\bin>
```

The final ecore model should look like this:



5. Tutorial: MDE4CPP Plugin Framework for Ecore

In sections 3 and 4 we learned how to create an ecore model, add behavior and custom methods to the model, generate and compile model code, and obtaining a model application as well as a static and dynamic library of our model.

But how can we use our model in an “external” program? MDE4CPPs *Plugin Framework* serves this purpose. The “Plugin” of our model is nothing more than the previously mentioned dynamic library (*.dll*-file) which we obtain after compiling a model. The purpose of the *Plugin Framework* is not only to be able to comfortably include model elements and methods in a “self-written” program, but also to use the model on the metamodel-level. This is realized by reflection (the model has the ability to “know and modify” its own structure and behavior).

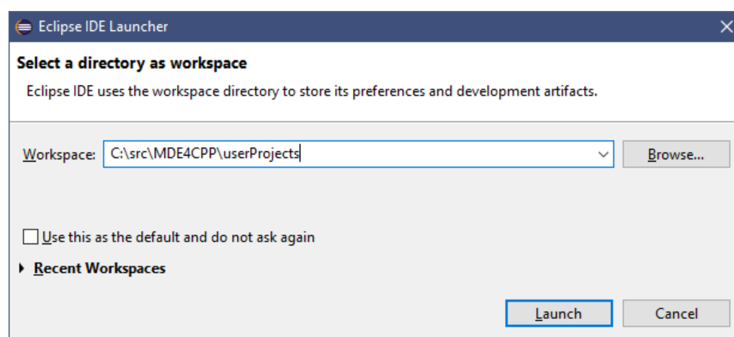
The goal of this tutorial is to use the model plugin compiled in section 3.4 by including the *Plugin Framework* in an external project, and give a basic insight on the possibilities of the *Plugin Framework*.

Hint: For help, find the complete and finished project here.

5.1 Creating the project

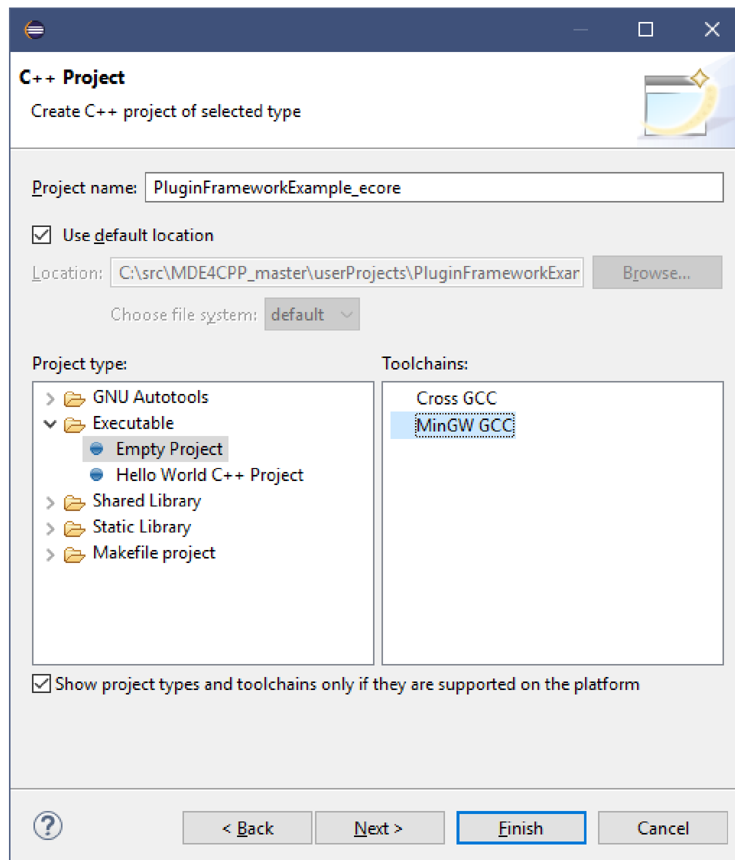
Step 1:

First we need to create the project. We will use an *Eclipse* IDE for developing C/C++-Applications (for download link see section 2.1). Choose *%MDE4CPP_HOME%\userProjects* as workspace directory, where %MDE4CPP_HOME% is the directory you installed MDE4CPP to. Click ***Launch***.



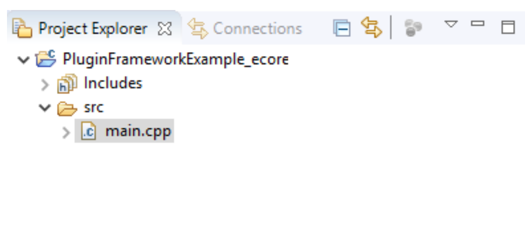
Step 2:

From the menu bar choose ***File → New → C/C++-Project***. Choose the ***C++ Managed Build*** template and click ***Next***. Name the project “*PluginFrameworkExample_ecore*”, choose “***Empty Project***” as project type and “***MinGW GCC***” as compiling toolchain. Click ***Finish***.



Step 3:

Inside the project explorer right-click ***PluginFrameworkExample_ecore***, then choose ***New → Folder*** and create a folder called "*src*" inside your project. After creating the folder, right-click it and choose ***New → Source File*** and create the file "*main.cpp*" inside the source folder. The project-tree inside the project explorer should look like this:



Hint: After creating the project using *Eclipse* you can also edit the file "*PluginFrameworkExample_ecore\src\main.cpp*" using any other text editor, source code editor, etc. (e.g. *Notepad++*) from now on, if you wish.

5.2 Including and using the model plugin

Step 1:

First we need the code 'frame' (required libraries includes, *main()* function). Copy the following code to the (empty) ***main.cpp*** file of the ***PluginFrameworkExample_ecore*** project:

```
//Incude standard libraries
#include <iostream>

//Include header files for plugin framework
#include "pluginFramework/PluginFramework.hpp"

int main()
{
	return 0;
}
```

Step 2:

Before we can use our model, we have to initialize the Plugin Framework. First, include the required header in your ***main.cpp*** file:

```
#include "pluginFramework/MDE4CPPPlugin.hpp"
```

By creating an instance of the *PluginFramework* class, the Plugin Framework will search for all plugin files in the directory "*%MDE4CPP_HOME%\application\bin*".

```
//Create an instance of the plugin framework
std::shared_ptr<PluginFramework> pluginFramework =
PluginFramework::eInstance();
```

We can now instantiate a *MDE4CPPPlugin* object by retrieving a plugin by its name (in our case, the plugins name is uniModelExample_ecore → our models name from section 3). Afterwards, we will check if the object was created successfully.

```
//Search for our model plugin and create an MDE4CPPPlugin object for it
std::shared_ptr<MDE4CPPPlugin> plugin =
pluginFramework->findPluginByName("uniModelExample_ecore");

//Check for nullptr
if (plugin){
	std::cout << "- MDE4CPPPlugin '" << plugin->eNAME() << "' found"
	<< std::endl;

	// !! Further code of Step 2 !!

}
```

MDE4CPPPlugin is just an interface to use for any possible plugin in MDE4CPP (e.g. ecore or UML plugins). To be able to retrieve our models Package and Factory, we need to cast it to an *EcoreModelPlugin* object. First, include the required headers in your ***main.cpp*** file:

```
#include "pluginFramework/EcoreModelPlugin.hpp"

//Include header files for ecore
#include "ecore/EFactory.hpp"
#include "ecore/EPackage.hpp"

using namespace ecore;
```

Now we can instantiate an *EcoreModelPlugin* object and get our models Package and Factory.

```
//Cast MDE4CPPPlugin to EcoreModelPlugin
std::shared_ptr<EcoreModelPlugin> ecorePlugin =
std::dynamic_pointer_cast<EcoreModelPlugin>(plugin);

//Check for nullptr
if(ecorePlugin){
        std::cout<<"- EcoreModelPlugin created from MDE4CPPPlugin"
        <<std::endl<<std::endl;

        //Get the models package
        std::shared_ptr<EPackage> uniModelPackage =
        ecorePlugin->getEPackage();

        //Get the models factory
        std::shared_ptr<EFactory> uniModelFactory =
        ecorePlugin->getEFactory();

        // !! Code for Steps 3-6 !!
  }
```

Step 3:

To create model elements, we can use the ***ecorePlugin***s *create()*-Method which returns an instance of *shared_ptr<EObject>*. This way we will not have to include any of our model specific libraries in our project. We only have to include some *ecore* header files to be able to call methods of metamodel elements.

```
#include "ecore/EObject.hpp"
```

Now we can create instances of our model classes (e.g. *University* and *Faculty* objects) by their class names.

```
//Create University and Faculty objects
std::shared_ptr<EObject> uniObj = ecorePlugin->create("University");
std::shared_ptr<EObject> facObj = ecorePlugin->create("Faculty");
```

Step 4:

We will now set some of our model class instances properties. As we handle these instance on the metamodel-level (remember, we instantiated them as type *EObject*), we cannot simply call the member functions of our model classes. For example, the model class *Faculty* does have a *setName()*-method, whereas *EObject* does not.

We will use *EObject*s *eSet()*-method which we pass the *EStructuralFeature* for which we want so set a value, as well as the value itself. As the metamodel-level does not know anything about concrete data types, the value parameter is passed as the *Any* type defined in MDE4CPP. For further information on the implementation of *Any*, see the file:

%MDE4CPP_HOME%\src\common\abstractDataTypes\src\abstractDataTypes\Any.hpp

Now, let’s set the *name* attribute of the ***facObj*** object. First include the required header in your ***main.cpp*** file.

```
#include "ecore/EClass.hpp"
```

We will call *EObject*s *eClass()*-method to get the metamodel information of ***facObj*** as *EClass*. Then we can call the *getEStructuralFeature()*-method of *EClass* to get an *EStructuralFeature* object by its name. The *eAny()*-method returns an *Any* type object that contains our value.

```
//Set EAttribute for multiplicity [0..1]
std::string facName = "Computer Science";
//Use eSet to set the value for attribute "name" of faculty
facObj->eSet(facObj->eClass()->getEStructuralFeature("name"),
eAny(facName));
```

To set a reference between our *Faculty* object and our *University* object (respectively, the *university* parameter of ***facObj***), we can do the exact same thing. The ***uniObj*** instance that we want so pass has to be encapsulated in an *Any* object like before.

```
//Set EReference for multiplicity [0..1]
facObj->eSet(facObj->eClass()->getEStructuralFeature("university"),
eAny(uniObj));
```

Step 5:

Let’s add the *Faculty* object to the *faculties* container of the *University* object. In other words, we want to manipulate a reference (member attribute) with multiplicity [0..*]. For that, we have to retrieve the *faculties* container. First include the required header:

```
#include "abstractDataTypes/SubsetUnion.hpp"
```

To get the *faculties* container we will use the *eGet()*-method of *EObject* which we pass the *EStructuralFeature* that we are interested in (like in step 4). *eGet()* returns an *Any* object as well. Because of that, we have to retrieve the content of the returned *Any* object before we can use it. As we are working only with *EObject* instances, we use *Any*s *get()*-method to cast and return its content as *Bag<EObject>*.

```
//Set EReference for multiplicity [0..*]
//Use eGet to create container of all faculties of our university
Any anyAllFaculties =
uniObj->eGet(uniObj->eClass()->getEStructuralFeature("faculties"));

//Casting the "Any" type to container of EObjects
std::shared_ptr<Bag<EObject> > allFaculties =
anyAllFaculties->get<std::shared_ptr<Bag<EObject> > >();
```

We can now add ***facObj*** and use *eSet()* to pass back the new container ***allFaculties*** to the ***uniObj*** object.

```
allFaculties->add(facObj);
//Set new container for universitys "faculties"
//(value passed as an "Any" type)
uniObj->eSet(uniObj->eClass()->getEStructuralFeature("faculties"),
eAny(allFaculties));
```

Create a new EObject instance of a student and set some of its attributed and references just like in steps 3-5.

```
std::shared_ptr<EObject> studObj = ecorePlugin->create("Student");
//Set students name attribute
std::string studName = "John Doe";
studObj->eSet(studObj->eClass()->getEStructuralFeature("name"),
eAny(studName));
//Set students matriculation number attribute
int studMatNum = 98765;
studObj->eSet(studObj->eClass()->getEStructuralFeature("matNum"),
eAny(studMatNum));

//Add the student to all students of our faculty
//Get container of all students as "Any" type and cast it
Any anyAllStudents =
facObj->eGet(facObj->eClass()->getEStructuralFeature("students"));
std::shared_ptr<Bag<EObject> > allStudents =
anyAllStudents->get<std::shared_ptr<Bag<EObject> > >();
//Add the student to container and pass it back using eSet()
allStudents->add(studObj);
facObj->eSet(facObj->eClass()->getEStructuralFeature("students"),
eAny(allStudents));
```

Step 6:

Finally we want to test the program. Use the *eGet()*-method to retrieve the attributes that were set during the previous steps. The following code implements an output functionality that will print out the contents of the concrete university model that we created in this tutorial.

```
//Get all faculties of our model
anyAllFaculties =
uniObj->eGet(uniObj->eClass()->getEStructuralFeature("faculties"));
allFaculties = anyAllFaculties->get<std::shared_ptr<Bag<EObject> > >();
std::cout<<"Your University:" <<std::endl<<std::endl;

//Loop over all faculties
for(Bag<EObject>::const_iterator facIt = allFaculties->begin();
    facIt != allFaculties->end(); facIt++){
       //Use eGet to get attribute "name" of the current faculty
 Any anyFacultyName =
(*facIt)->eGet((*facIt)->eClass()->getEStructuralFeature("name"));
       //Casting the "Any" type to std::string
std::cout <<"Faculty "<<anyFacultyName->get<std::string>()<<std::endl;

       //Get all students of the current faculty
anyAllStudents =
(*facIt)->eGet((*facIt)->eClass()->getEStructuralFeature("students"));
allStudents = anyAllStudents->get<std::shared_ptr<Bag<EObject> > >();

  //Loop over all students
  for(Bag<EObject>::const_iterator studIt = allStudents->begin();
  studIt != allStudents->end(); studIt++){
   Any anyStudentName =
  (*studIt)->eGet((*studIt)->eClass()->getEStructuralFeature("name"));
  Any anyStudentMatNum =
  (*studIt)->eGet((*studIt)->eClass()->getEStructuralFeature("matNum"));
  //Casting the "Any" types to std::string and int
  std::cout<<"\t-Student "<<anyStudentName->get<std::string>()
  <<", "<<anyStudentMatNum->get<int>()<<std::endl;
 }
}
```

5.3 Compiling the project

To be able to compile the project that you created in this tutorial, two more files need to be added to your project:

- a *CMakeLists*-File (required for *CMake*)
- a *GRADLE*-File (required for *Gradle*)

For this tutorial you can use prepared files and copy them to your project. Important parameters inside these files are described with comments in case you want to reuse and adapt them for your own projects.

Step 1:

Copy the prepared *CMakeLists.txt* file for this project to

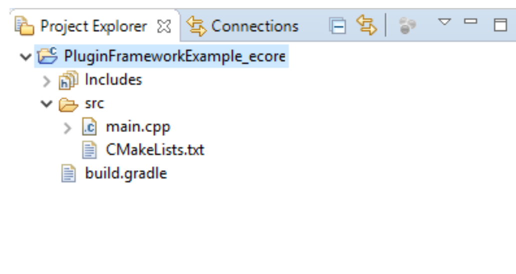
%MDE4CPP_HOME%\userProjects\PluginFrameworkExample_ecore\src
where %MDE4CPP_HOME% is the directory you installed MDE4CPP to.

Step 2:

Copy the prepared *build.gradle* file for this project to

%MDE4CPP_HOME%\userProjects\PluginFrameworkExample_ecore
where %MDE4CPP_HOME% is the directory you installed MDE4CPP to.

After step 1 and step 2, your project folder should now look like this:



Step 3:

Navigate to your %MDE4CPP_HOME% directory and open up a command prompt. Type in the following commands:

setenv.bat

gradlew tasks

Under task group ***PluginFrameworkExample_ecore tasks*** you should find a task called ***compilePluginFrameworkExample_ecore***.

```
C:\WINDOWS\system32\cmd.exe
PluginFramework tasks
---------------------
compilePluginFramework - compile pluginFramework
deliverPluginFrameworkInterface - deliver pluginFramework interface header to %MDE4CPP_HOME%
/application/include/pluginFramework

PluginFrameworkExample_ecore tasks
----------------------------------
compilePluginFrameworkExample_ecore - compiles PluginFrameworkExample_ecore

PrimitivetypesReflection tasks
------------------------------
buildPrimitivetypesReflection - build primitivetypesReflection
compilePrimitivetypesReflection - compile primitivetypesReflection
generatePrimitivetypesReflection - generate C++ code of primitivetypesReflection.uml model

Source tasks
------------
buildEcoreModels - generate and compile all ecore models
buildReflectionModels - generate and compile reflection models
compileEcoreModels - compile all ecore models
compileReflectionModels - compile all reflection models
generateEcoreModels - generate all ecore models
generateReflectionModels - generate all reflection models
```

Step 4:

Execute the task by typing in the following command:

gradlew compilePluginFrameworkExample_ecore

When the compilation process has finished, navigate to the directory:

%MDE4CPP_HOME%\application\bin

You should find the files "*App_pluginFrameworkExample_ecore.exe*" (compiled with release options) and "*App_pluginFrameworkExample_ecored.exe*" (compiled with debug options). You might also find just one of those files depending on how you configured the debug/release options in the file"*%MDE4CPP_HOME%\setenv.bat*".

```
C:\WINDOWS\system32\cmd.exe
C:\src\MDE4CPP\application\bin>App_pluginFrameworkExample_ecore.exe
- MDE4CPPPlugin 'uniModelExample_ecore' found
- EcoreModelPlugin created from MDE4CPPPlugin

Your University:

Faculty Computer Science
        -Student John Doe, 98765

C:\src\MDE4CPP\application\bin>_
```