The aim of this assignment is to make us able to define models in EMF using Eclipse, also the other step is to generate code from it and in final to work upon that code. We thought that for this assignment it would be more relevant to go with something different from the standard models. Meaning that in many books we were presented with the classic UML diagrams of: Bank account, Car system, or more typical that of a Library Management. So, for this assignment we will try to work with a Tournament of Bowling. It can be adapted to any sport tournament, so bowling it was just for naming purposes. We will represent only a limited number of classes and properties since going into much detail would require more time since plugins development is new for us.

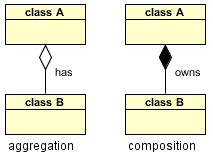
We were presented with EMF, this modeling framework that gives us the code generation facility with which we can build other tools centered on the model we built. The workflow followed is like this:

* Create a model using .ecore format steps. (Like those shown in the LAB)
* Use the built in ability of ecore model to generate code
* Use two instances of Eclipse: one for building purposes and the other for running the model instances and plugins.

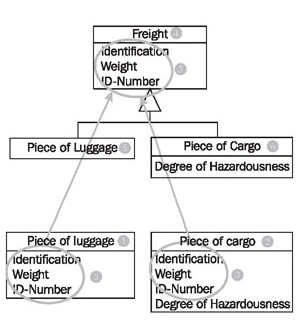
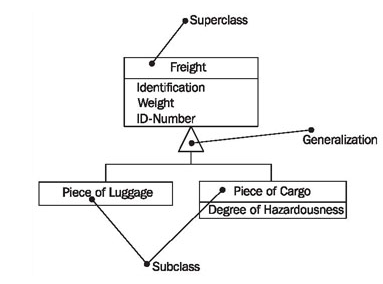
1. Create a meta model (.ecore file)

Requirements: Choose an example with at least three classes with one containment relationship and one generalization/specialization

Containment 🡪 It can be divided in two forms: aggregation (one object contains other objects), composition (the containing object has the duty to create the contained object). Talking in UML standards, respectively we are dealing with a non-filled diamond and a filled diamond, has and owns.



Generalization/specialization🡪 With generalization we have the case when shared characteristics from different classes are combined into a mother class. And in specialization, we have the case when subclasses are created from the existing class. Talking in UML, we are dealing with the triangle.



Keeping this in mind and the tournament, we can end up in these classes:

League🡪 Contains players

Tournament🡪Contains matchups and types.

Matchup🡪 Holds two games

Game🡪 Contains a list of scores assigned to a player.

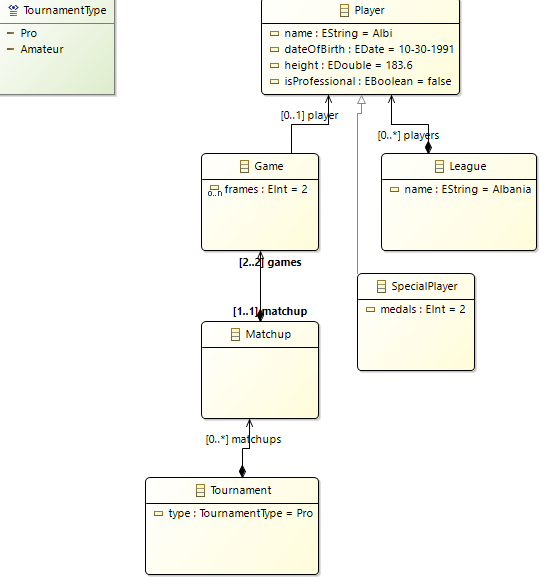


Figure 1. UML representation of our concepts.

1. Generate Java code for managing the model

Till we reach the code generation, we first have to follow the steps learned in the LAB.

We need a model in EMF first and after that the classes will be automatically generated.

* Create an empty EMF project, in New-Other.
* After we insert an ecore model from the Eclipse modeling framework
* After we finish a palette should be open, but in our case there was a problem with the Eclipse Mars or other configuration. So, we used Neon to make the model as showed in the class.
* Briefly will show the steps that were repeated for every class:
* In the model, we right click for new child-eclass and in the properties view the fillings were made accordingly.
* We add the attributes, called EAttributes. These are the ones that hold the properties.
* The multiplicity between classes was done via the EReferences. For example League and Player class, we made a right click on the League and made the EType to Player. The bounds are -1 which means many, and true for containment. A containment is the "A has B" kind of relation. Usually used for lists, e.g. "A has multiple B". ECore/EMF can then perform atomic commands on such collections, such as move all objects from one containment to another. It can also enforce constraints, such as a minimum amount of contained objects or a maximum amount of contained objects, or ensuring that the contained object is not contained in any other containment.
* Before going to the entities generation, we need to follow some steps for the so called genmodel. In the model folder, we click new-other – emf generator model. For the model importer we chose ecore model and the ecore model as workspace.
* For the generation step: We right click in the genmodel root and select the appropriate generations we want. In our case we need : model,edit,editor.
* Model: Necessary for the creation of model instances
* Edit: Needed for the labels.
* Editor: For the purpose of modification of model instances.

1. After all these were finalized successfully, in the first project we right click as run as Eclipse application, which creates the runtime eclipse instance. In this case, we now use the new-general-project and then we add the model via the new-other-example emf model creation wizard and select our model as an instance, as we can create as many instances as we want and many model object.

The connection between Tournament-Matchup, Matchup-Game will be explained. As we know a Matchup is composed of two Games. Like this vs. that. For this reason we make the bounds as 2. This is set as bi-directional for the reason as it can be traveled from the both sides. This is why we need an EReference going from Game to Matchup.

Here we are introduced to the concept of cross-EReference. This is a type of reference where the model elements do not have each other. This is the case of Game-Player. As many games can be assigned to a player, while the player is still member of a league.

The plugin:

We went with using pop-up menus as the way to integrate the actions.

Thanks to the generated Java Code, we can now have:

* Interface classes for every modeled object
* Get/Set for all object attributes
* Implementation classes for every interface
* A factory class to create instances of our model
* A package class that is the key to the access to the model metadata.

We should see the generated code in two perspectives:

1. Edit is needed as it provides items
2. Editor as a model editor

From different sources [1] [2] [3], it was stated that the generated classes are also Notifiers, which means they send out notifications when references or attributes are changed. This allows EMF objects to update views for example and labels.

Using the adapter factory we are able to make various implementations. For this, each EObject has an adapter as observers. The framework of edit uses the editing domain in order to manage the commands that act upon the model, meaning that the edit model interface is the door for gaining access to the model.

We have some dialog classes. There lies the method that gets info from the file and displays it in the respective view. Before we go with the command creation, we needed an editing domain. This last is responsible for managing models and commands into them. After that it comes the main core of this assignment. A command will come in help for this case. We use factory to create Matchup for example (holds true for the others). Afterwards the command is loaded and run.

AdapterFactoryEditingDomain🡪 Create an instance from the adapter factory, and the specialized command stack.

createResourcce🡪 This is a convenience method to create a resource, you could use the resource set returned by getResourceSet() directly.

We also tried to implement a treeview, which basically is a displayer. There we will need as a main thing the ContentProvider which makes possible the define of the structure via the getChildren(). This provider depends on EClass and here we use the AdapterFactory to get it as an input.

In the main method of this class, that of insertion to the model, we first use the EInstance, which is the singleton instance of the factory, as we need this factory for the model as it provides a create method for each non-abstract class of the model. We need an AdapterFactoryEditingDomain as going for definition: This class implements an editing domain by delegating to adapters that implement. And the addition of the command is made via the add command which logically acts upon an owner object that has a collection-based feature to which other objects can be added. And the last code to make the command execute needs the getComamndStack method that as per definition returns a command queue for executing commands.

When the new eclipse instance is created we just need to right click in the project and the actions will show. Each action was thought to be related to a model, based on the relationship that exists between classes. The fulfillment of the request: add one object (of each class) to an existing model if it does not already exist in the model, it was done with this line of code (it changes depending on the classes): EList<Game> g = matchup.getGames(); **if** (g.size()==0), if this requirement is passed than the adding code is executed, meaning the adding is done only once, as it says: if it does already exist, so if it exists than the adding is not done anymore.

# Bibliography

Eclipse. (2006). *Introduction to the Eclipse Modeling Framework.* Retrieved from http://www.eclipse.org/modeling/emf/docs/presentations/EclipseCon/EclipseCon2006\_EMF\_Intro.pdf

Eclipse. (2016). Retrieved from EMF Doc: http://download.eclipse.org/modeling/emf/emf/javadoc/2.8.0/org/eclipse/emf/edit/domain/AdapterFactoryEditingDomain.html

Steinberg, D., Budinsky, F., Paternostro, M., & Maerks, E. (2009). *EMF: Eclipse Modeling Framework.* Addison-Wesley.