

Model Engineering Lab 188.923 IT/ME VU, WS 2014/15	Assignment 1
Deadline: Upload (ZIP) in TUWEL until Monday, November 10 th , 2014, 23:55 Assignment Review: Wednesday, November 26 th , 2014	25 Points

Lab - Overview

Lab 1: In Lab 1, you will develop the *abstract syntax* of the so-called *Entity Forms Modeling Language (EFML)* using *Ecore* and *OCL*. EFML allows to define entities as well as forms for entering entity instances.

Lab 2: In Lab 2, your goal is to develop a *textual concrete syntax* and *editor support* for EFML using *Xtext*.

Lab 3: Lab 3 comprises the development of a *model-to-model transformation* using *ATL* which initializes EFML form definitions from EFML entity definitions.

Lab 4: In Lab 4, you will develop a *model-to-code transformation* that generates HTML and JavaScript code from EFML models using *Xtend*.

Metamodeling

The goal of this assignment is to develop the metamodel of the *Entity Forms Modeling Language (EFML)* using *Ecore* as well as *OCL* constraints defining additional well-formedness rules for EFML models.

Part A: Development of the Metamodel

Modeling Language Description

EFML is a modeling language for modeling entities as well as forms that enable to enter entity instances. In the following, we separately discuss EFML's modeling concepts for (1) modeling entities and for (2) modeling forms.

1.) Entity Modeling

The entity modeling concepts provided by EFML are described with the help of the example model depicted in Figure 1. For representing the model elements, we adopt the UML class diagram syntax. The example model defines the entities *Publication*, *Person*, *Proceedings*, *Journal*, *Book*, *Event*, and *PublicationVenue*. Table 1 describes the syntax and semantics of the entity modeling concepts in detail.

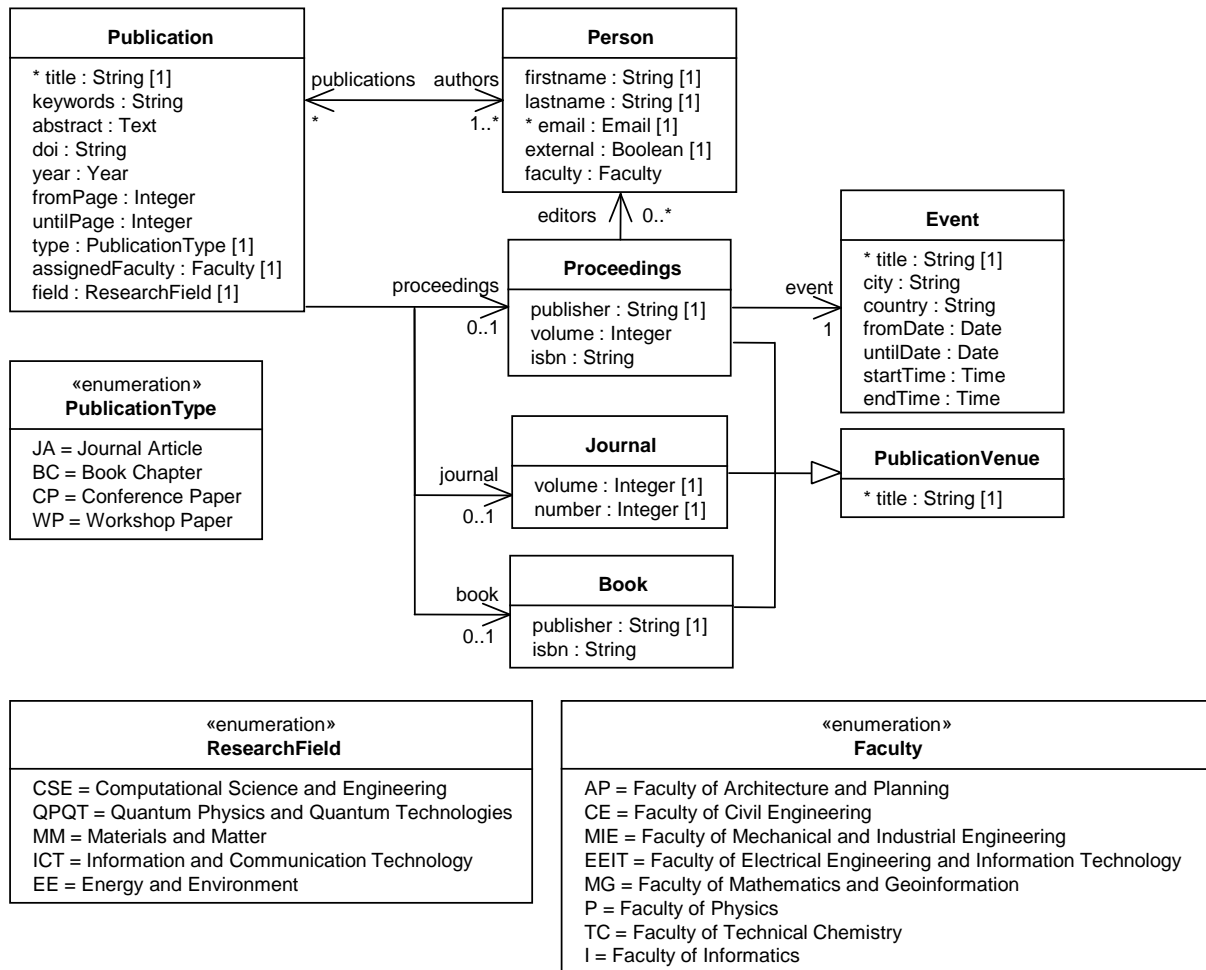


Figure 1: Entity modeling example

Syntax	Semantics
<div> <div>Publication</div> <div> * title : String [1] keywords : String abstract : Text doi : String year : Year fromPage : Integer untilPage : Integer type : PublicationType [1] assignedFaculty : Faculty [1] field : ResearchField [1] </div> </div>	<p><i>Entities</i> have a <i>name</i> and can contain <i>attributes</i>. One attribute must be set as the identifier (<i>id</i>) of the entity. <u>Example</u> The example shows the entity named “Publication” containing ten attributes. The attribute named “title” is defined as id (in the graphical syntax used for representing the example model, id attributes are indicated by the ‘*’ in front of the attribute’s name).</p> <p><i>Attributes</i> have a <i>name</i>, may be <i>mandatory</i>, and have a <i>type</i>. There exist nine predefined <i>AttributeTypes</i>: <i>String</i>, <i>Text</i>, <i>Integer</i>, <i>Date</i>, <i>Time</i>, <i>Year</i>, <i>Email</i>, <i>Boolean</i>, and <i>None</i>. Besides these predefined <i>AttributeTypes</i>, an attribute can also define an enumeration as type. In this case, the <i>AttributeType None</i> has to be set as <i>type</i> and the attribute has to reference the respective enumeration as <i>enumerationType</i>. <u>Example</u> The attribute with the name “keywords” is of type <i>String</i> and not mandatory. The attribute with the name “type” has the enumeration named “PublicationType” set as <i>enumerationType</i> and is mandatory (in the graphical syntax used for representing the example model, mandatory attributes are indicated by ‘[1]’ after the attributes’ type).</p>

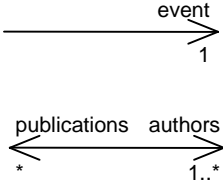
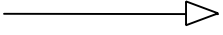
<div data-bbox="204 230 483 427"> <div>«enumeration» PublicationType</div> <div>JA = Journal Article BC = Book Chapter CP = Conference Paper WP = Workshop Paper</div> </div>	<p><i>Enumerations</i> have a <i>name</i> and consist of one or more <i>literals</i>.</p> <p><i>Literals</i> have a <i>name</i> and a <i>value</i>.</p> <p><u>Example</u></p> <p>The example shows an enumeration named “PublicationType” consisting of four literals, e.g., the first literal has the name “JA” and the value “Journal Article”.</p>
	<p><i>Relationships</i> are unidirectional references from one entity to another entity. A relationship has a <i>name</i>, a <i>target</i> (i.e., the referenced entity), and a multiplicity defined by a <i>lowerBound</i> and <i>upperBound</i> (the unbounded <i>upperBound</i> indicated by “*” in the graphical syntax must be represented by the Integer “-1”; furthermore the <i>lowerBound</i> has to be greater than or equal to zero and the <i>upperBound</i> has to be greater than or equal to the <i>lowerBound</i>).</p> <p>Bidirectional references between two entities are created with two distinct relationships pointing in opposite direction, where each of the relationships maintains a pointer to the corresponding <i>opposite</i> relationship.</p> <p><u>Example</u></p> <p>The relationship named “event” defines a unidirectional reference from the entity “Proceedings” to the (target) entity “Event” and has defined a lowerBound and upperBound of “1”.</p> <p>The relationships named “authors” and “publications” define a bidirectional reference between the entities “Publication” and “Person”. Thereby, the relationship named “authors” defines the entity “Person” as target, the lowerBound “1”, the upperBound “-1”, and the relationship named “publications” as opposite relationship. Similarly, the relationship named “publications” defines the entity “Publication” as target, the lowerBound “0”, the upperBound “-1”, and the relationship named “authors” as opposite relationship.</p>
	<p>EFML supports single inheritance between entities. Thereby, an entity can define at most one entity as <i>superType</i>.</p> <p><u>Example</u></p> <p>In the example model, the entities named “Proceedings”, “Journal”, and “Book” define the entity named “PublicationVenue” as super type.</p>

Table 1: Entity modeling concepts

2.) Form Modeling

EFML also provides modeling concepts for defining forms that allow to enter entity instances, e.g., forms for entering publications, persons, proceedings, etc.

A *Form* has a *name*, a *title*, and a *description*. Each form is associated with one *entity*. Exactly one form must be defined as *welcome form* constituting the main form for entering entity instances.

A form consists of a sequence of *pages*. A page has a *title* and consists of *page elements*.

Page elements have a *label* and an element identifier (*elementID*). EFML supports two kinds of page elements, namely *attribute page elements* for entering attribute values of entity instances and *relationship page elements* for entering instances of related entities.

An attribute page element refers to the *attribute* for which a value can be entered. Thereby, the attribute has to be defined or inherited by the entity associated with the form containing the attribute page element. The following types of attribute page elements are supported:

- *Text fields*: Text fields may be used to enter values for attributes of arbitrary type. Text fields may define the allowed value *format* in terms of regular expressions.
- *Text areas*: Text areas may like text fields be used to enter values for attributes of arbitrary type.
- *Selection fields*: Selection fields may only be used for attributes of type Boolean or having an enumeration set as type.
- *Date selection fields*: Date selection fields may be used to enter values representing dates for attributes of arbitrary type.
- *Time selection fields*: Time selection fields may be used to enter values representing times for attributes of arbitrary type.

A relationship page element refers to the relationship for which values (i.e., instances of the related entity) can be entered. Thereby, the relationship has to be defined or inherited by the entity associated with the form containing the relationship page element. The following types of relationship page elements are supported:

- *Lists*: Already entered values are displayed in terms of a list.
- *Tables*: Already entered values are displayed in terms of a table consisting of *columns*. Thereby, a *column* refers to the attribute of the related entity that shall be displayed.

Relationship page elements refer to a form (called *editing form*), which is associated with the related entity and may be used for entering instances of this related entity.

With EFML it is possible to define *conditions* on the visibility of pages and page elements. Conditions have an identifier (*conditionID*) and a *type* (*Hide*, *Show*, *Enable*, or *Disable*). It is distinguished between *attribute value conditions* and *composite conditions*.

An attribute value condition defines that a page or page element shall be shown, hidden, enabled, or disabled if a specific *value* was entered for a certain attribute using an attribute page element contained by the same form.

Composite conditions compose two conditions using the Boolean operators *AND* and *OR*. The composed conditions may be attribute value conditions or composite conditions.

Figure 2 depicts forms for the entities "Publication", "Person", "Journal", "Book", "Proceedings", and "Event", which can be defined using EFML. Aside the forms, additional information about the different elements is provided.

Form 1: Publication

Publication

Form for scientific publications

Publication Details

Title:

Keywords:

Abstract:

DOI:

Year:

From page:

Until page:

Publication type: v

Assigned faculty: v

Research field: v

Authors

First name	Last name	E-Mail
Firstname 1	Lastname 1	Email 1
Firstname 2	Lastname 2	Email 2
...

Add

Journal

- Journal 1

Add

Book

- Book 1

Add

Proceedings

- Proceeding 1

Add

Form (welcome form)

(Form description)

Page

Text field (format = "[a-zA-Z-]+")

Text field (format = "[a-zA-Z]+,)+[a-zA-Z]+")

Text area

Text field

Text field (format = "[0-9]+")

Text field (format = "[0-9]+")

Text field (format = "[0-9]+")

Selection field

Selection field

Selection field

Page

Table (with 3 columns showing first name, last name, and email of already added authors).

(The form Person may be used for entering additional authors.)

Page¹

List (shows already added journal)

(The form Journal may be used for entering a journal)

Page²

List (shows already added book)

(The form Book may be used for entering a book)

Page³

List (shows already added proceeding)

(The form Proceedings may be used for entering proceedings)

Form 2: Person

Person

Person Details

First name:

Last name:

E-Mail:

Faculty external:

v

Faculty:

v

Form

Page

Text field (format = “^[a-zA-Z-]+\$”)

Text field (format = “^[a-zA-Z-]+\$”)

Text field (format = “^\\w+ @[a-zA-Z-]+\\. [a-zA-Z]{2,3}\$”)

Selection field

Selection field⁴

Form 3: Journal

Journal

Journal Details

Title:

Volume:

Number:

Form

Page

Text field (format = “^[a-zA-Z-]+\$”)

Text field (format = “^[0-9]+\$”)

Text field (format = “^[0-9]+\$”)

Form 4: Book

Book

Book Details

Title:

Publisher:

ISBN:

Form

Page

Text field (format = “^[a-zA-Z-]+\$”)

Text field

Text field

Form 5: Proceedings

Proceedings

Proceedings Details

Title:

Publisher:

Volume:

ISBN:

• Editor 1

Add

• Event 1

Add

Form

Page

Text field (format = “^[a-zA-Z-]+\$”)

Text field

Text field (format = “^[0-9]+\$”)

Text field

List (shows already added editor)

(The form Person may be used for entering additional editors)

List (shows already added event)

(The form Event may be used for entering an event)

Form 6: Event		
Event		<i>Form</i>
Event Details		<i>Page</i>
Title:	<input type="text"/>	<i>Text field (format = "[a-zA-Z-]+")</i>
City:	<input type="text"/>	<i>Text field</i>
Country:	<input type="text"/>	<i>Text field</i>
From date:	<input type="text" value="DD-MM-YYY"/>	<i>Date selection field</i>
Until date:	<input type="text" value="DD-MM-YYY"/>	<i>Date selection field</i>
Start time:	<input type="text" value="hh:mm"/>	<i>Time selection field</i>
End time:	<input type="text" value="hh:mm"/>	<i>Time selection field</i>

Figure 2: Form modeling example

¹ The page *Journal* is only shown, if the value "JA" was entered for the attribute *type* (selection field labeled "Publication type").

² The page *Book* is only shown, if the value "BC" was entered for the attribute *type* (selection field labeled "Publication type").

³ The page *Proceedings* is only shown, if the value "CP" OR the value "WP" was entered for the attribute *type* (selection field labeled "Publication type").

⁴ The selection field labeled "Faculty" is hidden, if the value "true" was entered for the attribute *external* (selection field labeled "Faculty-external").

Task Description

Develop the metamodel for EFML with EMF's metamodeling language Ecore and define at least 8 OCL constraints for ensuring the well-formedness of EFML models.

- Make sure that you specified the metamodel as precisely as possible, i.e., the metamodel has to contain all described language concepts.
- Use EMF's "Sample Reflective Ecore Model Editor" or the "Sirius Diagram Editing" editor (new graphical editor for Ecore models in Eclipse Luna) to create the metamodel.
- Constraints regarding the well-formedness of EFML models, which cannot be ensured by the metamodel only, should be defined using OCL. You have to implement at least 8 OCL constraints. Make sure that you address all well-formedness rules defined in the modeling language description.
- Use the "OCLinEcore Editor" to add the OCL constraints to your metamodel.
- **IMPORTANT:** Do NOT translate the example models depicted in Figure 1 and Figure 2 to Ecore, but develop a modeling language (i.e., a metamodel) to define the language concepts as described above.

Part B: Metamodel Testing

Task Description

In Part A you developed the metamodel for EFML using Ecore and additionally the 8 required OCL constraints. They have to be tested in this part of the lab.

- Use the functionalities of EMF to create a tree-based model editor.
- Model the example entities and forms depicted in Figure 1 and Figure 2 with the help of this editor.
- For each defined OCL constraint create one small example model which fulfills the constraint and one small example model which does not fulfill the constraint (because 8 constraints are required, you have to prepare 16 small example models).

Submission & Assignment Review

At the assignment review you will have to present your metamodel for EFML, the additionally defined OCL constraints, the generated modeling editors for EFML, as well as the example models for testing the metamodel and the OCL constraints. The metamodel has to contain all described concepts and it must be possible to model the example entities and forms depicted in Figure 1 and Figure 2. Furthermore, the correctness of the defined OCL constraints must be shown.

Upload the following components in TUWEL:

- Entire EMF projects exported as archive file, containing:
 - Ecore file and Ecore diagram file of the metamodel (forms.ecore, forms.aird)
 - Models of the example entities and forms depicted in Figure 1 and Figure 2 (publicationEntityModel.xmi and publicationFormModel.xmi)
 - Models for testing the OCL constraints
 - Model-, edit- and editor-plugin-ins

All group members have to be present at the assignment review. The registration for the assignment review can be done in TUWEL. The assignment review is divided into two parts:

- Submission and **group evaluation**: 20 out of 25 points can be reached.
- **Individual evaluation**: Every group member is interviewed and evaluated separately. The remaining 5 points can be reached. If a group member does not succeed in the individual evaluation, the points reached in the group evaluation are also revoked for this student, which results in a negative grade for the entire course.

Note

1. **Metamodeling with Ecore:** For the lab part, we provide a description of **how to set up an Eclipse version** that contains all necessary plug-ins for all assignments. This Eclipse setup guide may be found at TUWEL:
<https://tuwel.tuwien.ac.at/mod/page/view.php?id=193744>

Once you have installed Eclipse and all necessary plug-ins, you can **create Ecore models**. Therefore, select *New → Eclipse Modeling Framework → Ecore Modeling Project*, click *Next >*, enter a *Project name* on the next page and confirm it with *Next >*, enter the *Main package name* "forms" and complete the wizard by clicking *Finish*. The project contains already some auto-generated files, which can be used to define an Ecore model.

The **Ecore diagram** (.aird file) provides a graphical view on the abstract syntax of the metamodel defined in the Ecore model (.ecore file). For opening the Ecore diagram file make sure that you are in the *Model* perspective (it should be selected in the upper-right corner of Eclipse; if, for instance, the *Resource* or *Java* perspective is opened, select *Window → Open Perspective → Other... → Modeling*). In the *Modeling* perspective you can expand the .aird until the diagram "forms class diagram" is visible. By double clicking you can open this diagram and start defining your metamodel by dragging metaclasses onto the canvas (*Palette Classifier → Class*).

You can also modify the **Ecore model** (.ecore file) directly with a **tree-based editor** (right click on the .ecore file and select *Open with → Sample Reflective Ecore Model Editor*). However, many changes cannot be automatically updated in the graphical view. In such case, the graphical view (Ecore diagram) must be rebuilt by right clicking on the .ecore file and selecting *Initialize Ecore Diagram ...* from the context menu.

You can decide, whether you create the metamodel with the graphical editor in the concrete syntax or with the tree-based editor in the tree-based abstract syntax. If you decide to model using the abstract syntax, generate a graphical view after creating the metamodel.

To **test the metamodel** you can right click on the created .ecore file and select *EPackages registration → Register EPackages into repository*. Now, open the .ecore file in the tree-based editor, right click on the metaclass you want to test and select *Create Dynamic Instance*. Specify a name for the new model (e.g., *test.xml*). To determine whether the whole model is correct or not right click on the root element of the model and select *Validate*.

2. **Creating a model editor:** To create a tree-based editor from your metamodel you have to open the **Generator model** (.genmodel file), right click on the root element and select *Generate All*. As a result, three additional projects with the postfix *edit*, *editor*, and *test* are created.

To **test your editor**, open the *MANIFEST.MF* file of the previously generated *editor* project. In the *Overview* of the *MANIFEST.MF* click *Launch an Eclipse application*. In the new opened Eclipse instance you can create an EMF model by selecting *New → Example EMF Model Creation Wizards → Forms Model* (depending on the settings in your .genmodel file).

For more information about EMF visit <http://www.eclipse.org/emf>.

3. **OCL in Ecore:** You can add OCL constraints to your metamodel by opening the .ecore file with the OCLinEcore editor by right clicking on your .ecore file and selecting *Open With* ➔ *OCLinEcore Editor*.

The *OCL Console* is very helpful for defining and testing *OCL constraints*. To make use of the *OCL Console* open a test model (e.g., *test.xml*) with the *Sample Reflective Ecore Model Editor*, right click on any model element and select *OCL* ➔ *Show OCL Console*.

Further information about OCLinEcore can be found at

<http://help.eclipse.org/luna/topic/org.eclipse.ocl.doc/help/Tutorials.html#OCLinEcoreTutorial>