GENERATIVE SOFTWARE DEVELOPMENT

Model-Driven Engineering Tutorial

Creating metamodels, models, transformations and workflows

The purpose of this tutorial is to get some familiarity in Model-Driven Engineering and transformation languages. Based on the following example we are going to create several models using EMF and the necessary transformations to generate the application source code.

Suppose you have been asked to develop an application for managing the MP3 songs of a music store. This application must display the songs' catalog and the information of a selected song. It must also allow adding new songs to the catalog and order them by each of the song attributes (using either bubble or insertion sort). Each song has a name, a price, a length (in minutes and seconds), a size (in MB) and a bit rate (in kbps).

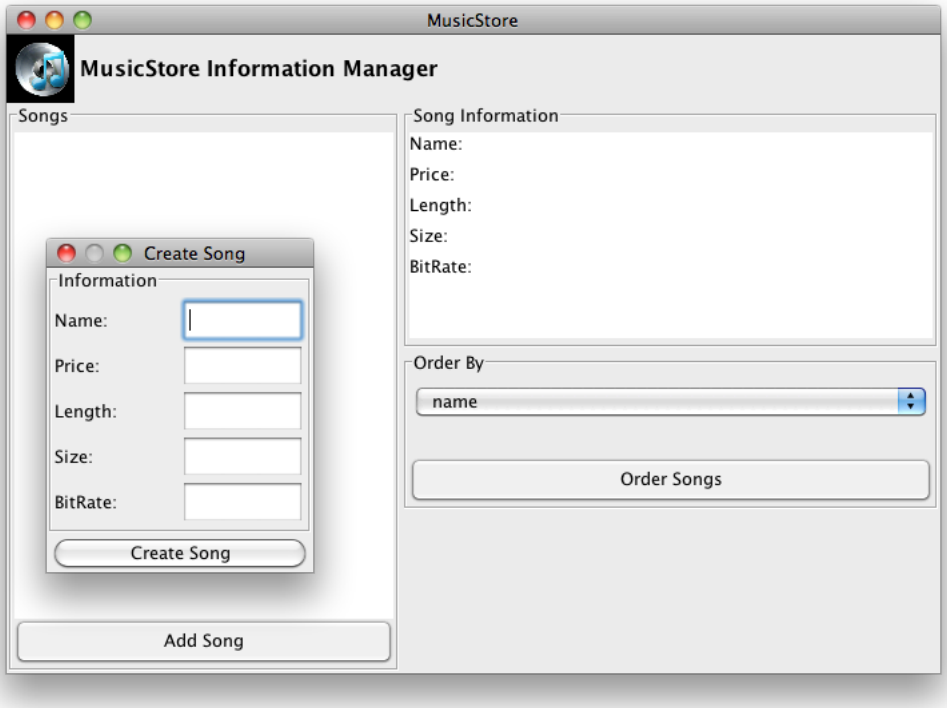

Figure 1. Music store GUI

Setup the work environment:

The first step to model-driven engineering is getting the work environment set up. This environment is how you will transform the models you create into the source code of an application. The work environment consists of Eclipse Modeling Tools, the openArchitectureWare plug-in for Eclipse, and the oAW project of the music store case study.

Download the Eclipse Modeling Tools with the oAW plug-in from:

- Windows: http://200.3.193.23/departamentos/tecnologias_informacion_comunicaciones/proyectos/desarrollo_generativo/tools/eclipse_helios_win.zip
- Mac OS X: http://200.3.193.23/departamentos/tecnologias_informacion_comunicaciones/proyectos/desarrollo_generativo/tools/eclipse_helios_mac.tar.gz

I. Metamodels, Models and the Eclipse Modeling Framework (EMF)

The Eclipse Modeling Framework (EMF) is a framework for modeling and code generation. A model represents an abstraction of the domain you want to work with. In order to do this, you need to create what is called a metamodel and the actual model. The metamodel describes the structure of the model; in other words, it contains the information about the defined concepts. The model is then an instance of this metamodel.
EMF allows creating the metamodel using Ecore, a language to define structural models. Ecore defines different elements:

- **EClass:** represents a class, with zero or more attributes and zero or more references.
- **EAttribute:** represents an attribute, which has a name and a type.
- **EReference:** represents one end of an association between two classes. It has a flag to indicate if it represents containment and a reference (target) class to which it points.
- **EDataType:** represents the type of an attribute, e.g. int, double or String.

The Ecore model shows a root object representing the whole model. This model has children, who represent the packages, and its children represent the classes. The children of the classes represent the attributes of these classes.

Define the Music Store Metamodel:

1. Import the music store openArchitectureWare Project into your Eclipse workspace (co.edu.unicesi.gendev.mde.collectionManager.basic).
2. Inside the project folder you will find three source folders named *'src'*, *'metamodels'* and *'models'*.
3. Select the *'metamodels'* folder, right click on it and select New > Other… > Eclipse Modeling Framework > Ecore Model. Give it the name *'problemSpaceMetamodel'*. This will open a tree view editor.

   Note: To generate the graphical view of the metamodel, in the Package Explorer view, right click the created Ecore Model and select the 'Initialize Ecore Diagram File…' option. You can edit the model on either view.

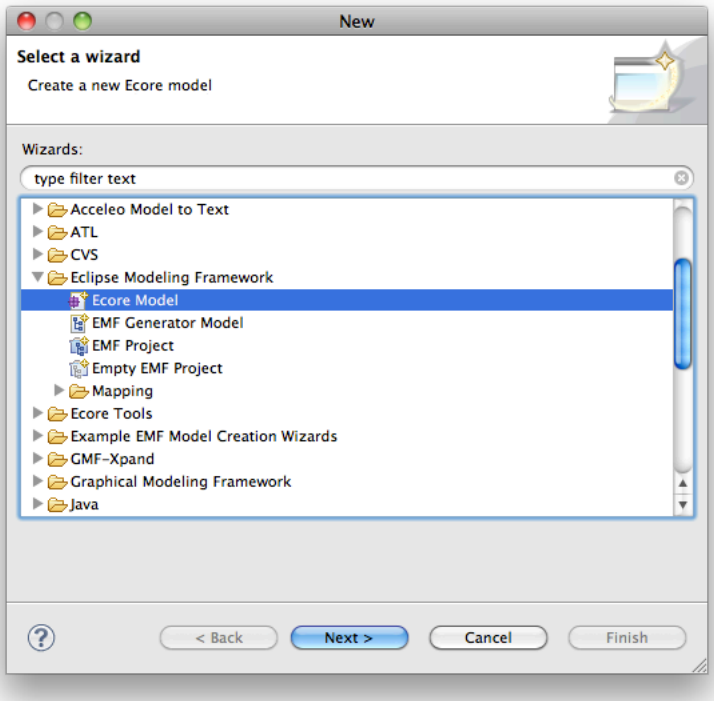

Figure 2. Create a new Ecore model

4. Open the *'Properties'* view. This view will allow you to modify the attributes of the model elements.
5. Create the EClasses, EAttributes and EReferences for the metamodel shown in Figure 3. This metamodel represents collections. The Entity is the element to be managed, and each characteristic is a property of this Entity.

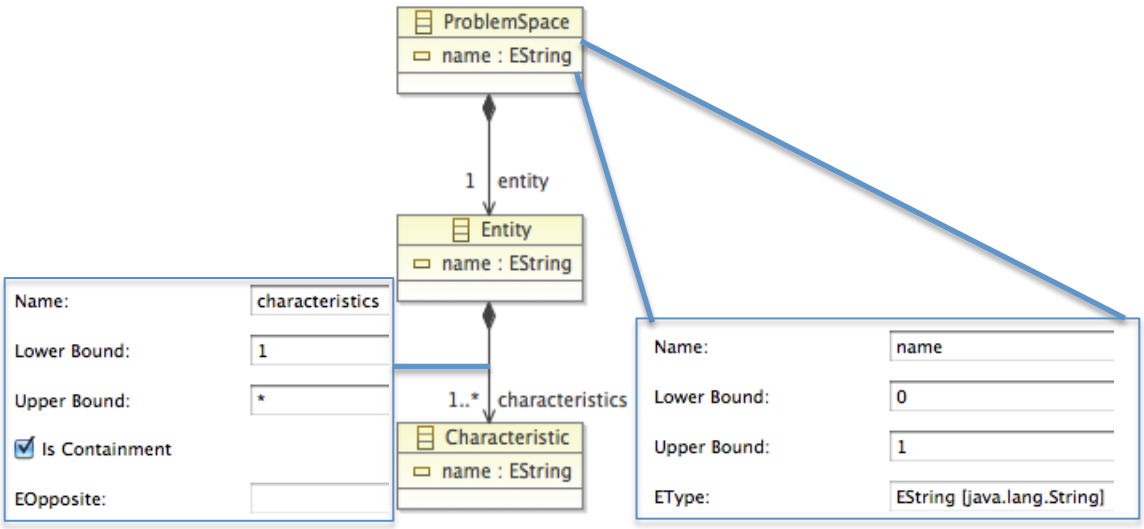

Figure 3. Problem space metamodel

Exercise: Repeat steps 3 and 5 for the following metamodels. This metamodels represent the kernel and GUI structures. Name the metamodels *'kernelMetamodel'* and *'guiMetamodel'*.

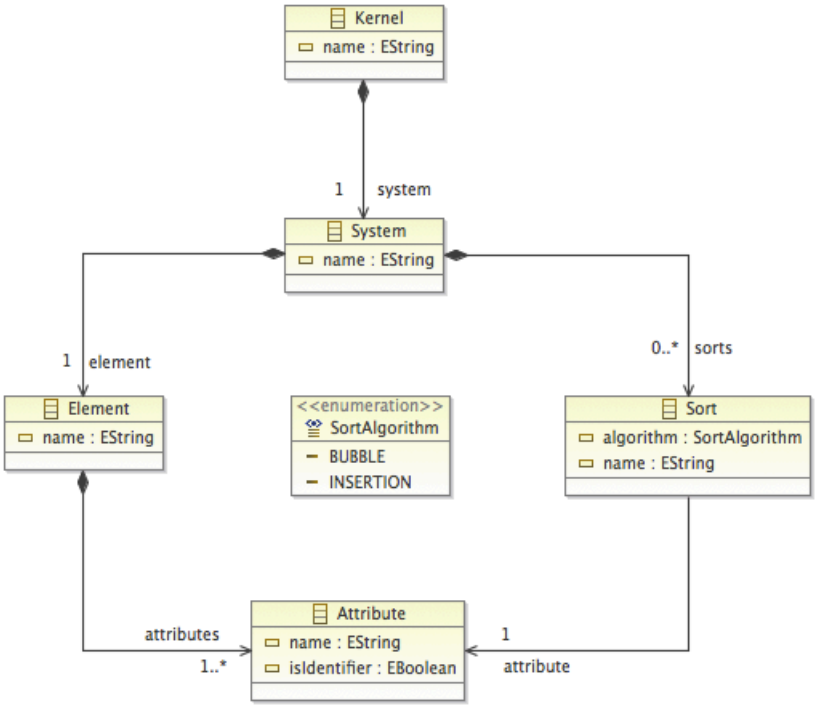

Figure 4. Kernel metamodel

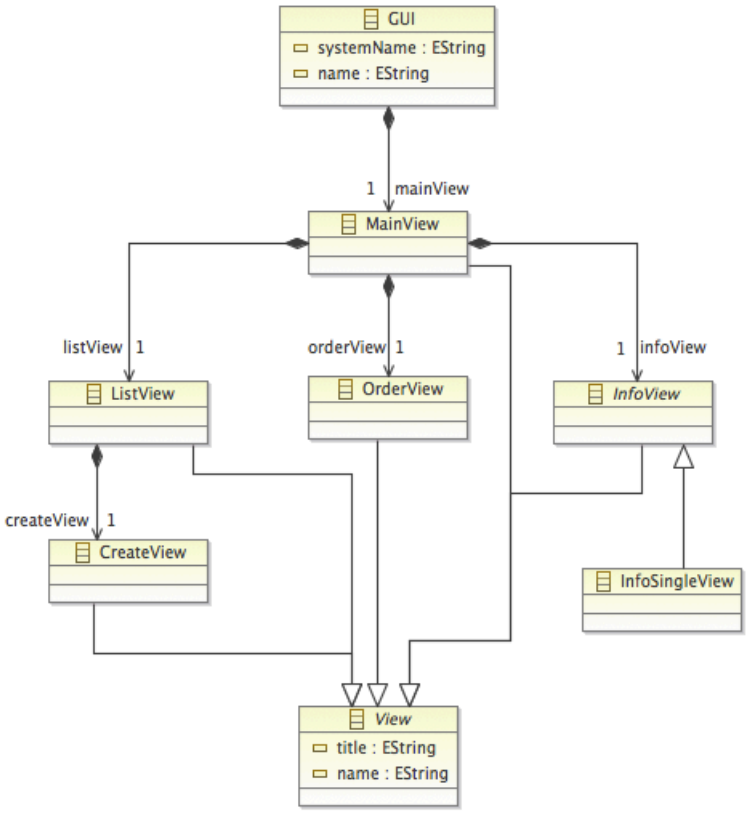

Figure 5. GUI metamodel

6. There are two ways to create a model conforming to the problem space metamodel.

Exercise: Create the problem space model both ways.

a. One way is to create a dynamic instance of the metamodel:
    i. Open the problem space metamodel, right click the ProblemSpace EClass and select 'Create Dynamic Instance…'.
    ii. Select the *'models'* folder and give the model a name (e.g. problemSpaceModel).

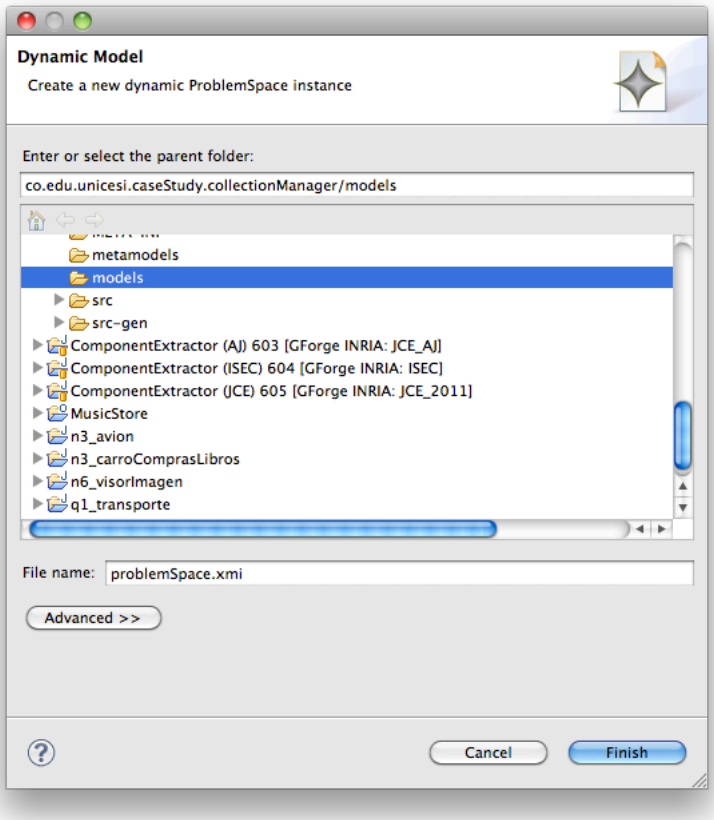

Figure 6. Create a new dynamic model instance

b. Another way is to generate an editor using EMF Model Generation. This editor allows creating models that conform to the metamodel:
    i. Inside the *'metamodels'* folder, create a generator model by selecting File > New > Other… > Eclipse Modeling Framework > EMF Generator Model and give the model a name and add the genmodel extension (e.g. problemSpace.genmodel).

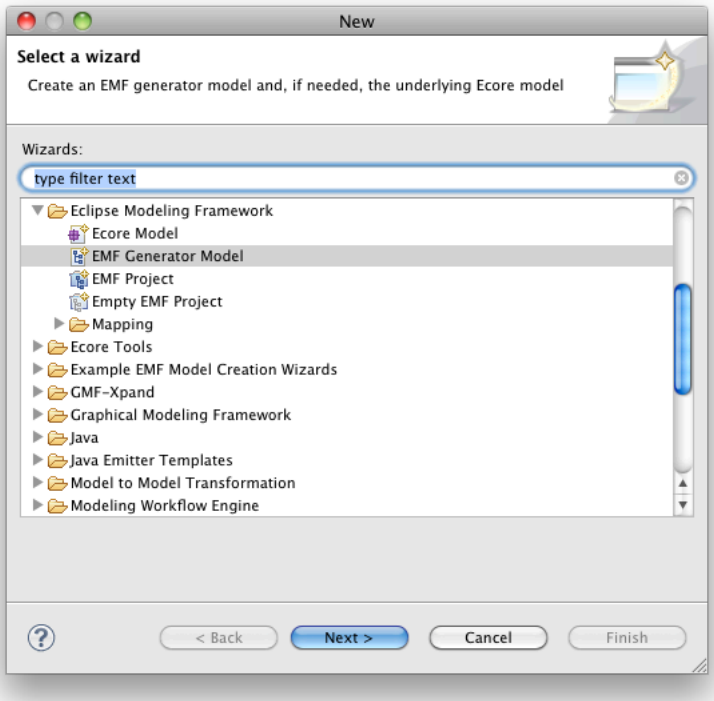

Figure 7. Create an EMF generator model

ii. Then, select as model importer the created problemSpaceMetamodel.ecore.

iii. The model editors are automatically generated as Eclipse plug-ins by using an EMF facility. For this, open the previously created genmodel, right click the root element and select the *'Generate All'* option.

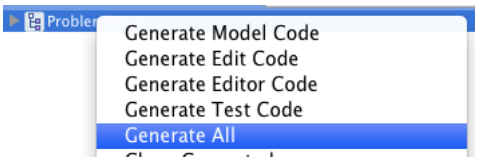

Figure 8. Generator model contextual menu

iv. For creating models that conform to the created metamodels it is necessary to execute the created plug-in. For this, right click on the project and select Run As > Eclipse Application. Another Eclipse instance is launched.

v. In the new Eclipse instance, Import the music store openArchitectureWare Project into the new Eclipse instance workspace.

vi. Create a new ProblemSpaceMetamodel Model inside the *'models'* folder. Right click on the project and select New > Other… > Example EMF Model Creation Wizards > ProblemSpaceMetamodel Model. Give the model a name (e.g. problemSpaceModel).

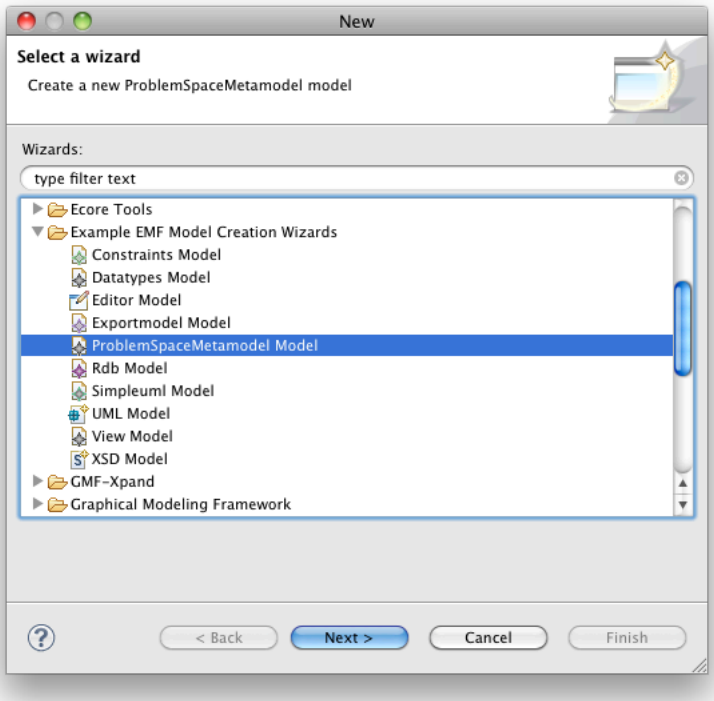

Figure 9. Create a new model

vii. As Model Object select the defined main container EClass. In this case *'ProblemSpace'*. The model is created and different model elements can be added.

Note: If you created the problem space model using EMF Model Generation (step 6.b), the following steps are done in the imported project of the second Eclipse instance.

7. Create the elements as shown in Figure 10. In the next section, this model will be transformed to a model conforming to the kernel metamodel and to another model conforming to the GUI metamodel.

   Note: There is no graphical view support for XMI models.

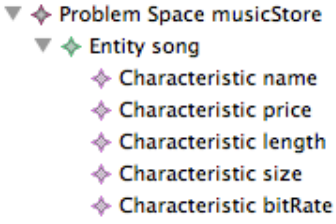

Figure 10. Problem space model

Note: For a more detailed tutorial about creating an Ecore model refer to the Modeling Tutorial in the *docs/reference* folder.

II. Model-to-Model Transformation

Note: Before you continue reading, review the product derivation process explanation. For this refer to the Process Explanation file in the *docs* folder.

Model-to-model transformations aim to provide a mean to specify the way to produce target models from a number of source models. This target models are more specialized than the source models; for example, the target model may include architectural details. Figure 11 shows a simple model transformation Ma2Mb that defines the way to generate a model *Mb*, conforming to a metamodel *MMb*, from a model *Ma*, conforming to a metamodel *MMa*. The source model elements must be matched and navigated in order to initialize the target model elements.

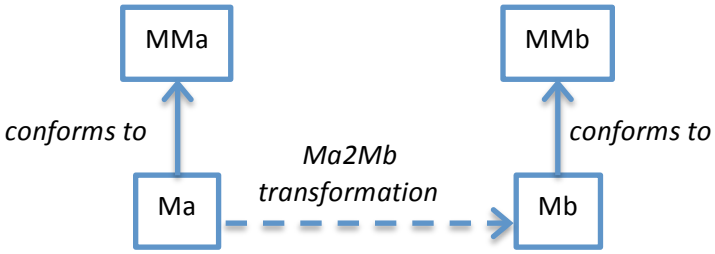

Figure 11. Model to model transformation

As mentioned earlier, the music store problem space model from step 6.b is going to be transformed to a model conforming to the kernel metamodel and to another model conforming to the GUI metamodel. There are several model-to-model transformation languages that can operate on the Ecore models. Among these languages are Xtend and ATLAS Transformation Language (ATL).

Define the Music Store Model-to-Model Transformations:

8. Expand the *'src'* folder, you will find a package for the kernel transformations (*'transformations.kernel'*) and another for the GUI transformations (*'transformations.gui'*).
9. Select the *'transformations.kernel'* package and open the *'problemSpace2kernel.ext'* file. This Xtend file contains the transformations to generate a kernel model from the problem space model created in the previous section.

Note: To create a new Xtend file, go to New > Other… and search for the Xtend folder as shown in Figure 12.

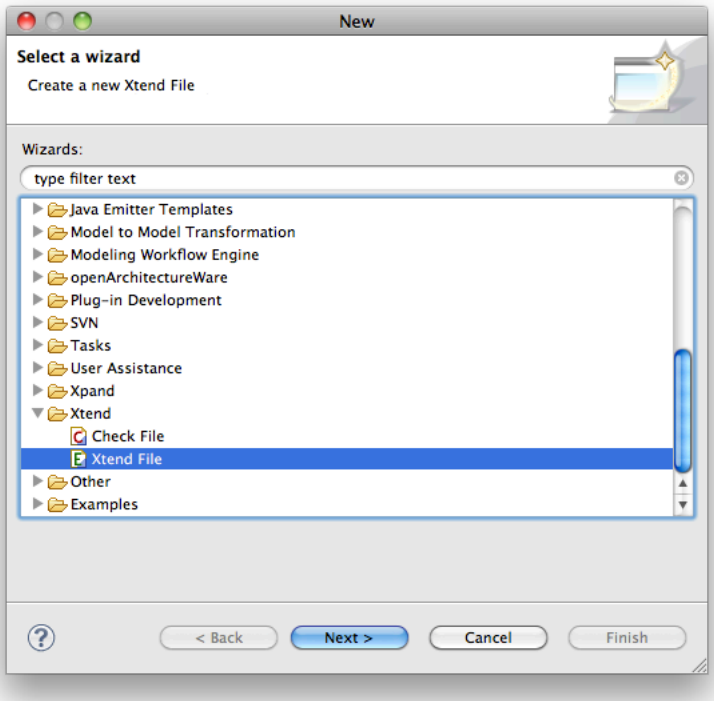

Figure 12. Create a new Xtend file

10. In the Xtend file you opened, notice first the imports for the problem space metamodel, our source metamodel, and the kernel metamodel, our target metamodel.

```
import problemSpaceMetamodel;
import kernelMetamodel;
```

Listing 1. Imports

11. Following is the root transformation that will begin the element matching and model navigation. Figure 13 shows the element matching between the problem space and kernel metamodels.

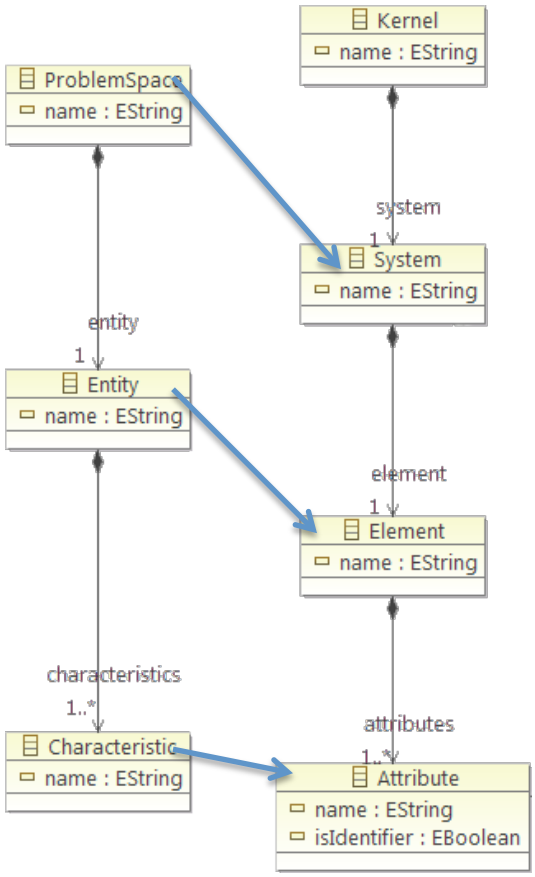

Figure 13. Problem space to kernel transformations

```
create kernelMetamodel::Kernel
  problemSpace2kernel(problemSpaceMetamodel::ProblemSpace element):
        createSystem(this, element) ->
        this;
```

Listing 2. Root transformation

Each transformation is a so-called '*create*' extension and creates an instance of the type found after the `create` keyword. The previous transformation creates an instance of the Kernel concept found in the kernel metamodelo, our target metamodel, and is referred inside the transformation as `this`. The only parameter in this transformation is a ProblemSpace concept instance referred by element. The body of the transformation calls the createSystem transformation with the Kernel concept and ProblemSpace concept instances as parameters. A template for a model-to-model transformation with parameters is as follows:

```
create <target_metamodel>::<target_concept> <transformation_name>
  (<source_metamodel>::<source_concept> <parameter_name>):
        <instruction> ->
        this;
```

Listing 3. Template for model-to-model transformation with parameters

A template for a model-to-model transformation without parameters is as follows:

```
create <target_metamodel>::<target_concept> <transformation_name>:
        <instruction> ->
```

```
this;
```

Listing 4. Template for model-to-model transformation without parameters

Note: Instructions must end with `->`, and the transformation must end with `this;`.

12. The 'createSystem' transformation creates an instance of the System concept found in the kernel metamodel based on the Element concept from the problem space metamodel.

```
create kernelMetamodel::System createSystem(kernelMetamodel::Kernel kernel,
  problemSpaceMetamodel::ProblemSpace element):
        setName(element.name) ->
        entity2element(this, element.entity) ->
        kernel.setSystem(this) ->
        this;
```

Listing 5. Example transformation

Exercise: Finish the following kernel model-to-model transformations:

- Entity-to-element transformation.
- Characteristic-to-attribute transformation.
- Bubble sort transformation.
- Insertion sort transformation.

Note: The GUI model-to-model transformations can be found in the *'problemSpace2gui.ext'* file.

Note: For additional information, and syntax and expression reference about the Xtend language see the *Check / Xtend / Xpand Reference* (Xtend section) of the *openArchitectureWare User Guide* in the *docs/reference* folder.

III. Model-to-Text Transformation

Finally the kernel and GUI models are transformed to source code. There are several model-to-text transformation languages that can operate on the Ecore models. Among these languages are Xpand and Acceleo.

Define the Music Store Model-to-Text Transformations:

13. In the *'src'* folder, you will also find a package for the kernel templates (*'templates.kernel'*) and another one for the GUI templates (*'templates.gui'*).
14. Select the *'templates.kernel'* package and open the Xpand file *'kernel2text.xpt'*. This Xpand file contains the template to generate the kernel source code.

Note: To create a new Xpand file, go to New > Other… and search for the Xpand folder as shown in Figure 14.

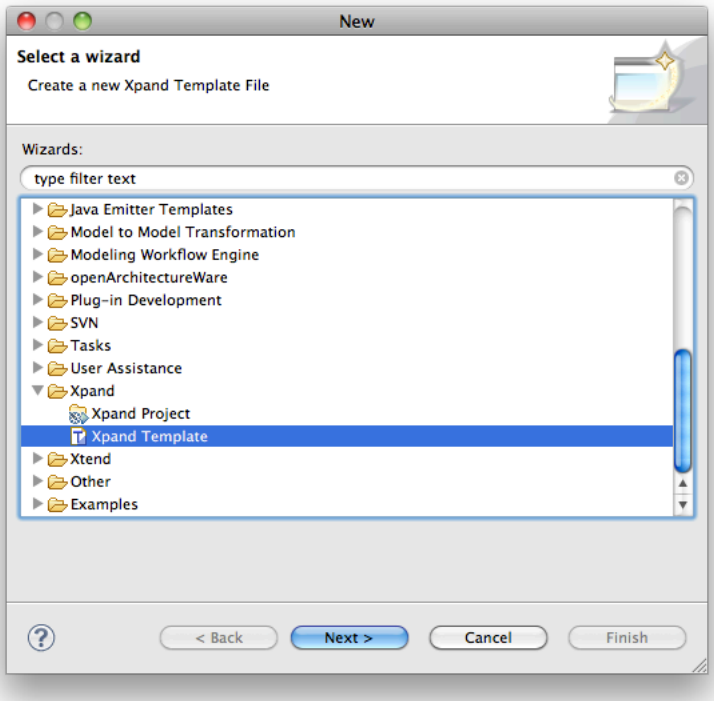

Figure 14. Create a new Xpand file

15. In the Xpand file you opened is the root template that will begin the element matching and model navigation.

```
«DEFINE kernel2text FOR kernelMetamodel::Kernel»
	«LET system.name.trim().toFirstUpper() AS systemName»
		«LET system.element.name.trim().toFirstUpper() AS elementName»
			«FILE "/unicesi/driso/" + systemName.toLowerCase() +
		"/kernel/" + systemName + ".java"»
			package unicesi.driso.«systemName.toLowerCase()».kernel;

			import java.util.Observable;
			import java.util.ArrayList;
			import java.util.Iterator;
			import java.util.Arrays;

			public class «systemName» extends Observable
			{

			...java source code...


			}
			«ENDFILE»
		«ENDLET»
	«ENDLET»
	«EXPAND element::element(system.name) FOR system.element»
«ENDDEFINE»
```

Listing 6. Root transformation

A model-to-text transformation, also called template section, in Xpand has a name and is always assigned to a specific element type or metamodel concept. The previous template defines a transformation called kernel2text that is applied to an instance of the Kernel concept in the kernel metamodel. The **EXPAND** directive is used to execute a **DEFINE** block for the specified type. The **LET** operator is used to declare variables and their content. The **FILE** operator is used to create a file in the specified path. The « and » characters, or guillemots, are Xpand's escaping characters, in other words, they separate the template expressions from the static text fragments for the output.

A template for a model-to-text transformation without parameters is as follows:

```
«DEFINE <transformation_name> FOR <target_metamodel>::<target_concept>»
	«LET <variable_value> AS <variable_name>»
		«FILE <relative_path>/<file_name>.<file_extension>»
			output text«<variable_name.<service_name>()»output text

			...
		«ENDFILE»
	«ENDLET»
	«EXPAND <xtend_file_name>::<transformation_name>[(<parameters>)] FOR
<target_concept>»
«ENDDEFINE»
```

Listing 7. Template for model-to-text transformation without parameters

A template for a model-to-text transformation with parameters is as follows:

```
«DEFINE <transformation_name>(<Type> <parameter_name>) FOR
<target_metamodel>::<target_concept>»
	«LET <variable_value> AS <variable_name>»
		«FILE <relative_path>/<file_name>.<file_extension>»
			output text«<variable_name.<service_name>()»output text

			...
		«ENDFILE»
	«ENDLET»
	«EXPAND <xtend_file_name>::<transformation_name>[(<parameters>)] FOR
<target_concept>»
«ENDDEFINE»
```

Listing 8. Template for model-to-text transformation without parameters

Exercise: Finish the following kernel model-to-text transformations:

- Bubble sort template.
- Insertion sort template.

Note: The Xpand file *'gui2text.xpt'* contains the GUI model-to-text transformations.

Note: For an overview of the Xpand language see Xpand: A Closer Look at the model2text Transformation Language (http://www.bar54.de/benjamin.klatt-Xpand.pdf). For additional information,

and syntax and expression reference about the Xpand language see the *Check / Xtend / Xpand Reference* (Xpand section) of the *openArchitectureWare User Guide* in the *docs/reference* folder.

IV. openArchitectureWare (oAW) Workflow

The oAW workflow engine is a declarative configurable generator engine. It provides a simple, XML-based configuration language that consists of a number of components executed sequentially. A workflow description consists of a list of configured WorkflowComponents. Next is the workflow description to generate the music store application source code. This workflow consists of different workflow components, indicating source and target models, and transforms, indicating the root transformations and their respective source and target models. The order of the declaration is important because the workflow engine will execute the components in the specified order.

Define the Music Store Application Generator Workflow:

16. In the *'src'* folder, you will find an openArchitectureWare workflow file named *'applicationGenerator.oaw'*.

Note: To create a new Workflow file, go to New > Other… and search for the openArchitectureWare folder as shown in Figure 15.

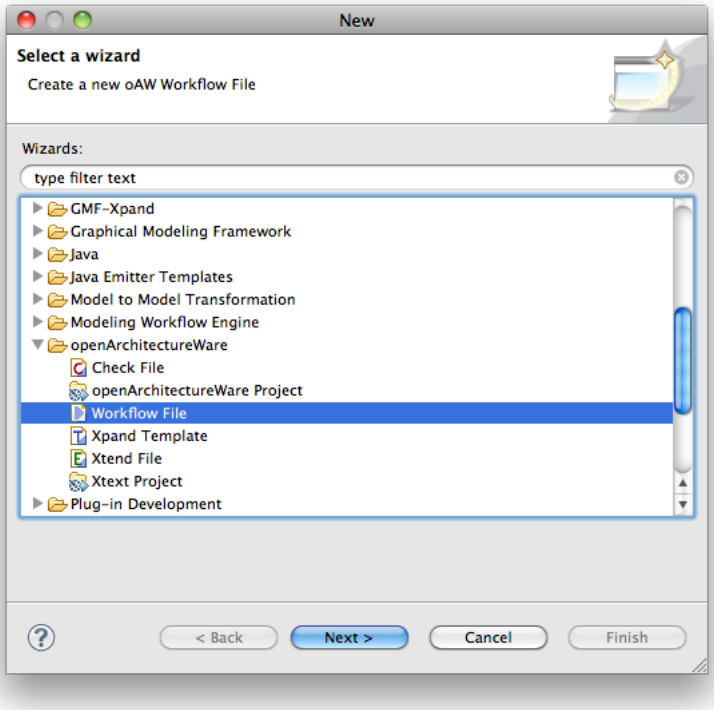

Figure 15. Create a new oAW workflow file

The workflow file you opened contains the following content.

```
<?xml version="1.0"?>
<workflow>
```

Listing 9.

17. The *Reader* reads the problem space model and stores the content in a slot named *problemSpace*.

```
<component class="org.eclipse.mwe.emf.Reader">
	<useSingleGlobalResourceSet value="true" />
	<uri value="models/problemSpaceModel.xmi" />
	<modelSlot value="problemSpace" />
</component>
```

Listing 10.

18. Invokes the root transformation for problemSpace-to-kernel transformation and stores the output in a slot named *kernelModel*.

```
<transform id="problemSpace2kernel">
	<metaModel
class="org.openarchitectureware.type.emf.EmfMetaModel"><metaModelFile
value="metamodels/problemSpaceMetamodel.ecore" /></metaModel>
	<metaModel
class="org.openarchitectureware.type.emf.EmfMetaModel"><metaModelFile
value="metamodels/kernelMetamodel.ecore" /></metaModel>

	<invoke
value="transformations::kernel::problemSpace2kernel::problemSpace2kernel(problemSpace
)" />
	<outputSlot value="kernelModel" />
</transform>
```

Listing 11.

19. The *XmiWriter* writes the content of the slot *kernelModel* in a model file (xmi).

```
<component class="org.openarchitectureware.emf.XmiWriter">
	<inputSlot value="kernelModel" />
	<modelFile value="models/kernelModel.xmi" />
</component>
```

Listing 12.

20. Steps 17, 18 and 19 are repeated for problemSpace-to-gui transformation.

```
<component class="org.eclipse.mwe.emf.Reader">
	<useSingleGlobalResourceSet value="true" />
	<uri value="models/problemSpaceModel.xmi" />
	<modelSlot value="problemSpace" />
</component>

<transform id="problemSpace2gui">
	<metaModel
class="org.openarchitectureware.type.emf.EmfMetaModel"><metaModelFile
value="metamodels/problemSpaceMetamodel.ecore" /></metaModel>
	<metaModel
class="org.openarchitectureware.type.emf.EmfMetaModel"><metaModelFile
value="metamodels/guiMetamodel.ecore" /></metaModel>
```

```
            <invoke
value="transformations::gui::problemSpace2gui::problemSpace2gui(problemSpace)" />
            <outputSlot value="guiModel" />
      </transform>

      <component class="org.openarchitectureware.emf.XmiWriter">
            <inputSlot value="guiModel" />
            <modelFile value="models/guiModel.xmi" />
      </component>
```

Listing 13.

21. The *Generator* invokes the kernel-to-text and gui-to-text transformations and writes the generated files to the *src-gen* folder.

```
      <component id="kernel2text" class="org.openarchitectureware.xpand2.Generator">
            <metaModel
class="org.openarchitectureware.type.emf.EmfMetaModel"><metaModelFile
value="metamodels/kernelMetamodel.ecore" /></metaModel>

            <expand value="templates::kernel::kernel2text::kernel2text FOR
kernelModel" />
            <outlet path="src-gen">
                  <postprocessor
class="org.openarchitectureware.xpand2.output.JavaBeautifier" />
            </outlet>
      </component>

      <component id="gui2text" class="org.openarchitectureware.xpand2.Generator">
            <metaModel
class="org.openarchitectureware.type.emf.EmfMetaModel"><metaModelFile
value="metamodels/guiMetamodel.ecore" /></metaModel>
            <metaModel
class="org.openarchitectureware.type.emf.EmfMetaModel"><metaModelFile
value="metamodels/kernelMetamodel.ecore" /></metaModel>

            <expand value="templates::gui::gui2text::gui2text(kernelModel) FOR
guiModel" />
            <outlet path="src-gen">
                  <postprocessor
class="org.openarchitectureware.xpand2.output.JavaBeautifier" />
            </outlet>
      </component>
</workflow>
```

Listing 14.

22. Right click the workflow file and select *'Run as'* > *'oAW Workflow'* to run the workflow and generate the music store application source code.

Note: For additional information, and syntax and expression reference about the oAW Workflow see the *openArchitectureWare User Guide* in the *docs/reference* folder.

References

1. Universidad de los Andes, Understanding Models and Metamodels: Music Store Case Study.
2. Lars Vogel, Eclipse Modeling Framework (EMF) – Tutorial. Retrieved April 12, 2011, from http://www.vogella.de/articles/EclipseEMF/article.html.
3. Hugo Arboleda and Rubby Casallas, Model Driven Engineering: Modeling Tutorial, 2007.
4. Markus Voelter and Iris Groher, Handling Variability in Model Transformations and Generators.
5. Benjamin Klatt, Xpand: A Closer Look at the model2text Transformation Language, 2007.
6. openArchitectureWare User Guide version 4.3.1, 2008.