



## DiaGen DiaMeta

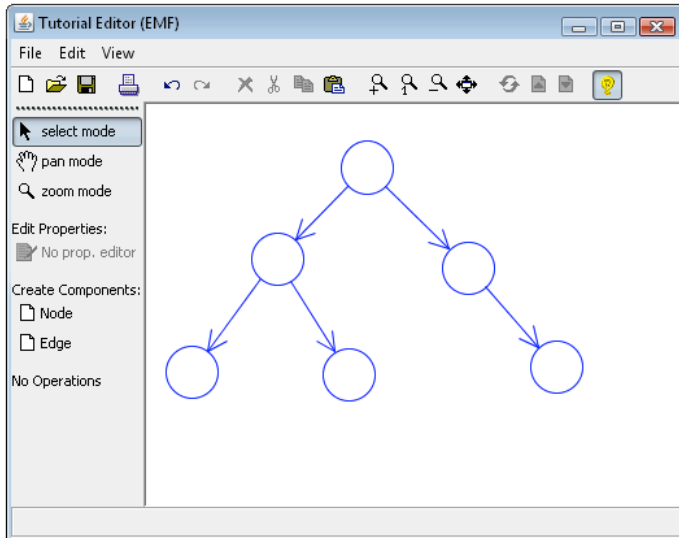
**Tutorial**  
**Version 1.5**  
**12.02.2009**

# 1. Contents

- 1. CONTENTS ..... 2
- 2. INTRODUCTION ..... 3
- 3. SOFTWARE REQUIREMENTS ..... 4
- 4. INSTALLATION ..... 4
  - Option 1: Webpage ..... 4
  - Option 2: Repository ..... 6
- 5. RUN A GENERATED EDITOR ..... 7
- 6. CREATE AN EDITOR ..... 8
  - Create a project in Eclipse ..... 8
  - Create an EMF Model ..... 9
  - Create the DiaMeta specification file ..... 14
  - Define Relations ..... 25
  - Write a Layouter (optional) ..... 30
  - Run the Editor ..... 32
  - Enable GML Output (optional) ..... 32
- 7. REFERENCES ..... 33

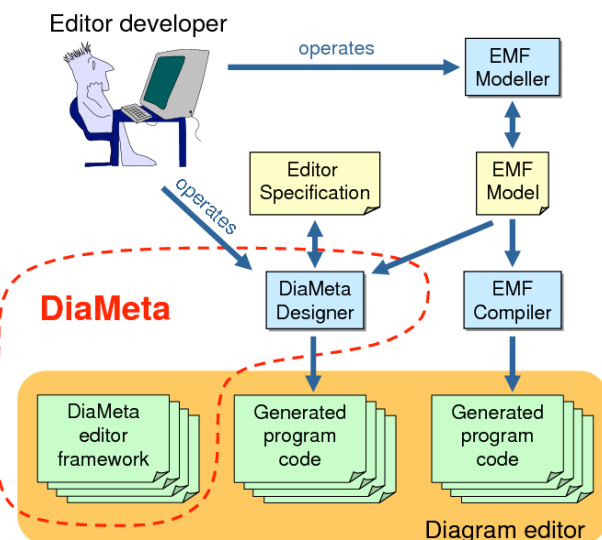
## 2. Introduction

This tutorial explains, how to use the diagram editor generator framework DiaMeta. We will explain how to install the DiaMeta diagram generator framework, and how to create an editor using the generator. The figure below shows the editor that will be created.



DiaMeta is a diagram editor generator that makes use of meta-model-based language specifications (DiaMeta) or of hypergraph grammars (DiaGen) and supports free-hand as well as structured editing. The language specification by hypergraph grammars is not described in this tutorial. (In the following we refer to DiaMeta only.)

The DiaMeta environment shown in the figure below consists of an editor framework and the DiaMeta designer. As a collection of Java classes, the framework provides the generic editor functionality which is necessary for editing and analyzing diagrams.



In order to create an editor for a specific diagram language, the editor developer has to provide two specifications: First, the abstract syntax of the diagram language in terms of its model, and second, the visual appearance of diagram components, the concrete diagram language syntax, the reducer rules and the interaction specification.

Further details can be found in the paper [Min06] "Generating Meta-Model-Based Freehand Editors".

### 3. Software requirements

You will need Eclipse Version 3.3 (or higher) and EMF plugin for Eclipse 2.3.0 [Emf07]. (Eclipse Version 3.2 might also work, but wasn't tested.) Besides that, subversion [Svn07] and maven [Mvn07] are required (if you want to work directly on our repository).

### 4. Installation

You may either use the files provided on our webpage (see paragraph "Option 1: Webpage"), or you may directly work on our repository (see paragraph "Option 2: Repository").

#### Option 1: Webpage

In this section, we will explain how to use the files that are available for download on our webpage

<http://www.unibw.de/inf2/DiaGen/download.html> .

You can download the executable jar `examples-xxx.jar` that allows you to view different examples. Start it as

```
> java -jar examples-xxx.jar
```

Besides, you can download the diagen designer as an executable jar. To start it, you need to type

```
> java -jar designer-and-runtime-xxx.jar
```

#### Editor dependencies

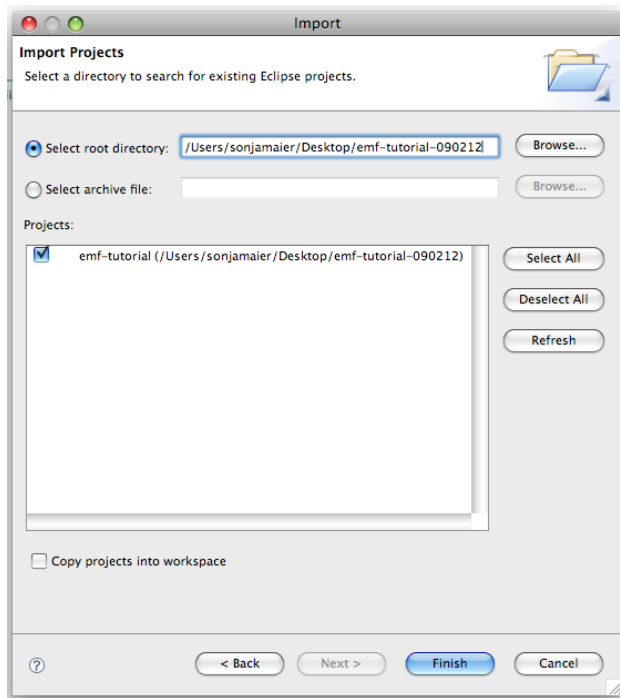
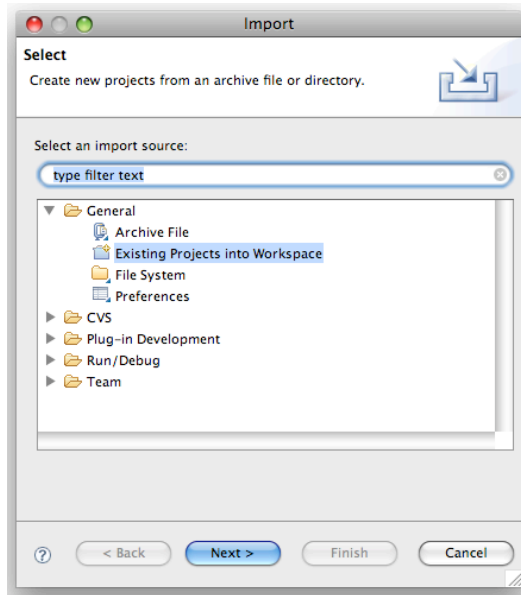
If you create an editor, the following dependency is required:

```
designer-and-dependencies-xxx.jar
```

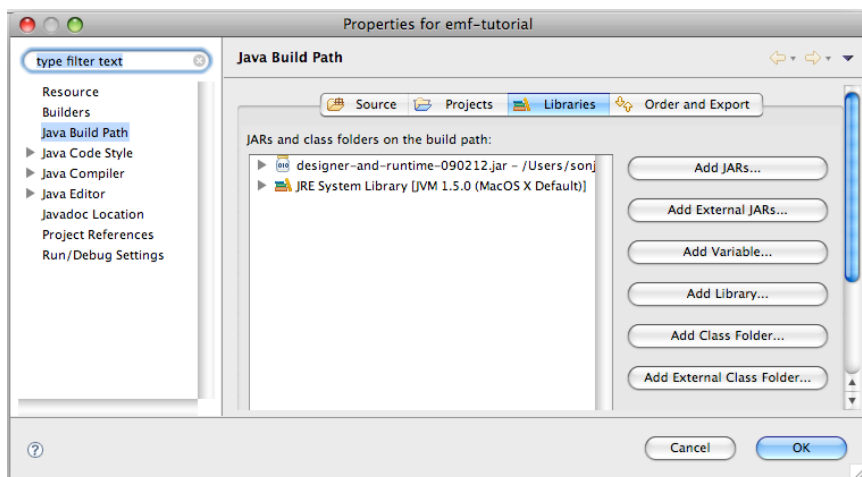
On the webpage, also the source code of the tutorial is provided as an eclipse project.

```
emf-tutorial-xxx.zip
```

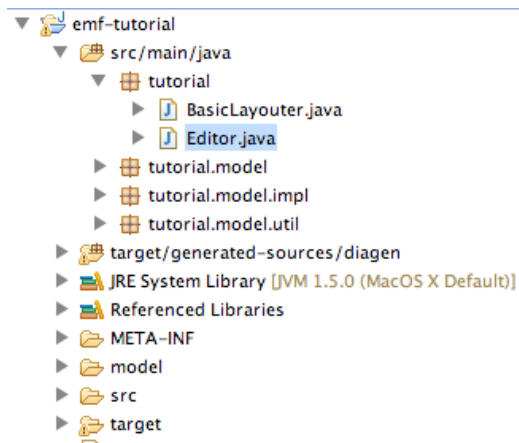
Unzip it and import it as a project into Eclipse.



Add `designer-and-dependencies-xxx.jar` as a dependency.



The main class of the project is `Editor.java`.



## Option 2: Repository

In this section, we will explain how to checkout the repository, and how to integrate DiaMeta into Eclipse.

### Checkout the Repository

To checkout the repository, you need your own username and password. If you do not already have one, please contact us. ([mark.minas@unibw.de](mailto:mark.minas@unibw.de) or [sonja.maier@unibw.de](mailto:sonja.maier@unibw.de) )

First of all you need to checkout DiaMeta. If you want to checkout the trunk, you need to do the following.

```
svn checkout
https://argonaut.informatik.unibw-muenchen.de
/repos/trunk
```

Alternatively, you can checkout a branch (e.g. you are a student working with the system) via

```
svn checkout
https://argonaut.informatik.unibw-muenchen.de
/repos/branches/<yourBranch>
```

Now, build and install DiaMeta packages via

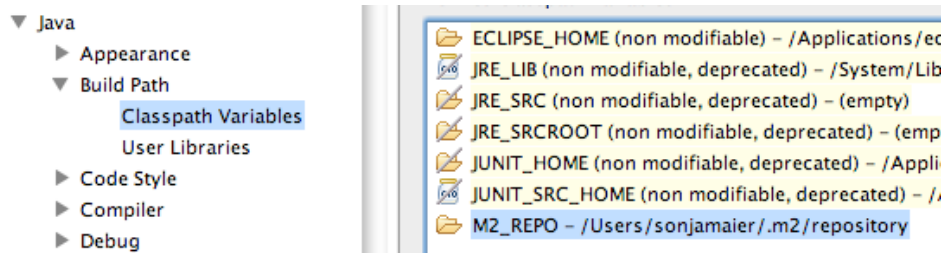
```
mvn install
```

### Integrate into Eclipse

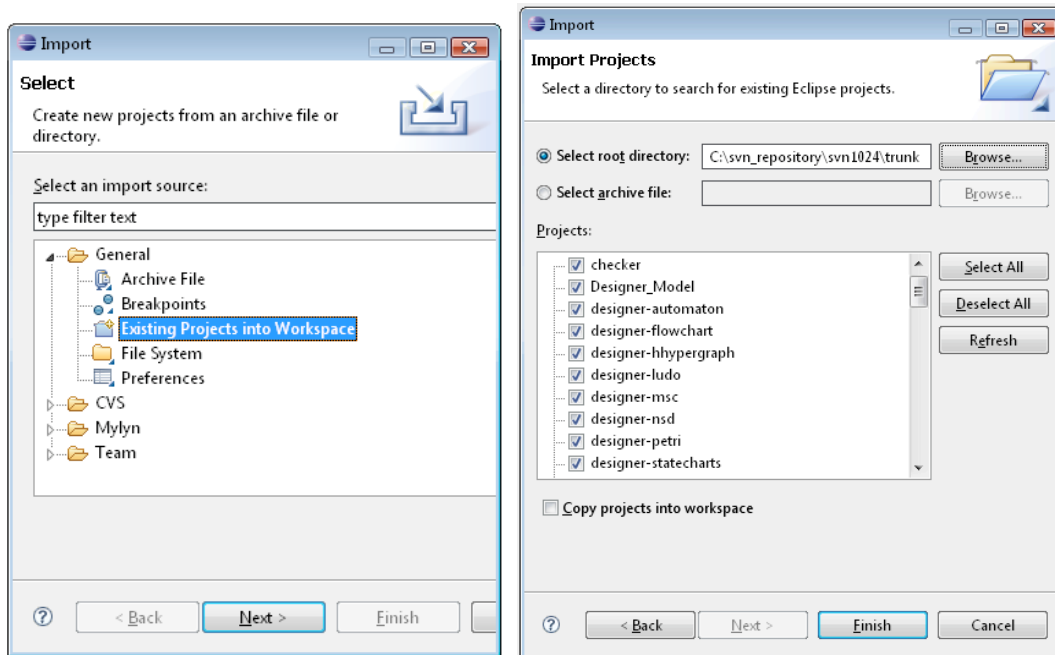
To integrate DiaMeta into Eclipse, you have to execute the following commands.

```
mvn -Declipse.workspace=<path-to-eclipse-workspace> eclipse:add-maven-repo
mvn eclipse:eclipse
```

This creates the classpath variable M2\_REPO in Eclipse that points to the .M2 folder on your computer.



In Eclipse, you need to import all projects, that can be found in the directory `./trunk` as existing projects. (Without copying!)



**Congratulations! The installation is complete.**

## Remove The Generated Files (optional)

After running the build script several times, the following command might be useful:

```
mvn clean
```

It removes all generated files, and you may start from scratch. (`mvn install && mvn eclipse:eclipse`)

**To see changes in Eclipse, you have to press F5!**

## Create An Executable Jar (optional)

With another useful command, you can create an executable jar for each example. You need to change to the directory, the example is located in.

For example, you may switch to the directory:

```
.\trunk\examples\DiaMeta-examples-EMF\EMFTutorial
```

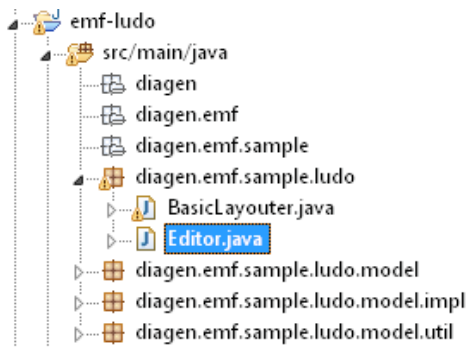
Then you may run the command

```
mvn assembly:assembly
```

The result is the jar file `emf-tutorial-1.0-SNAPSHOT-jar-with-dependencies.jar` in the directory `.\EMFTutorial\target\`.

## 5. Run a generated editor

To run a generated editor directly from inside Eclipse, you need to execute the main class. For each (example) editor, there exists the class `Editor.java`. E.g. for the sample editor `emf-ludo`, the main class is `diagen.emf.sample.ludo.Editor.java`.



## 6. Create an editor

Now we will create an editor from scratch. Basically, we will need to do the following things:

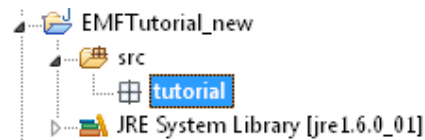
- ➔ Create new project in Eclipse.
- ➔ Create EMF Model (Tutorial.ecore) and generate the model code.
- ➔ Create specification file (spec.dds) with the DiaMeta designer and generate the editor code.
- ➔ Write the layouter BasicLayouter.java.
- ➔ Compile the generated as well as the manually written source code
- ➔ Run the editor.

### Create a project in Eclipse

Start Eclipse with an empty workspace and create a new Java project with the name EMFTutorial\_new.

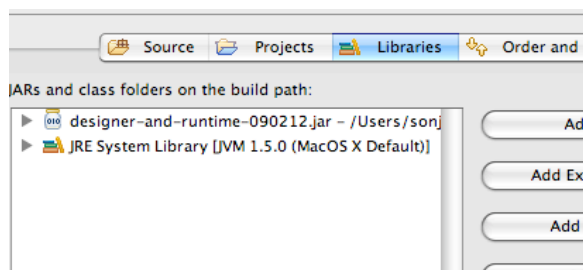
File >> New >> Java Project

Create the package tutorial.



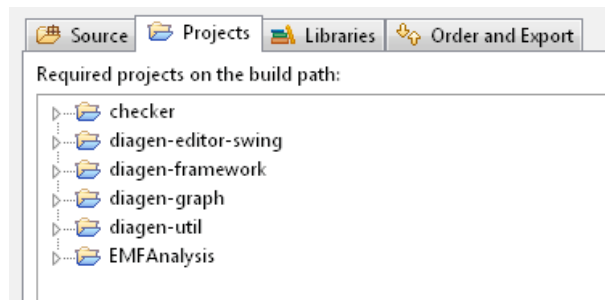
Add the required projects and libraries to the Java Build Path.

#### Option 1: Webpage

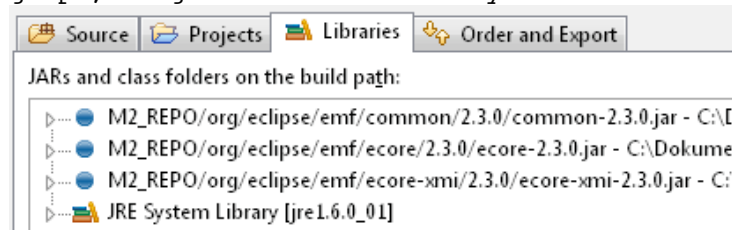


The required library is designer-and-runtime-xxx.jar. This jar includes all required libraries.

## Option 2: Repository



The required projects are checker, diagen-editor-swing, diagen-framework, diagen-graph, diagen-util and EMFAnalysis.



The required libraries are

M2\_REPO/org/eclipse/emf/common/2.3.0/common-2.3.0.jar

M2\_REPO/org/eclipse/emf/ecore/2.3.0/ecore-2.3.0.jar

M2\_REPO/org/eclipse/emf/ecore-xmi/2.3.0/ecore-xmi-2.3.0.jar

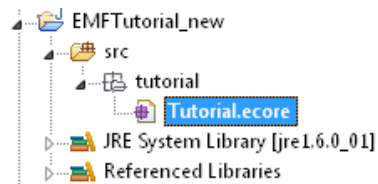
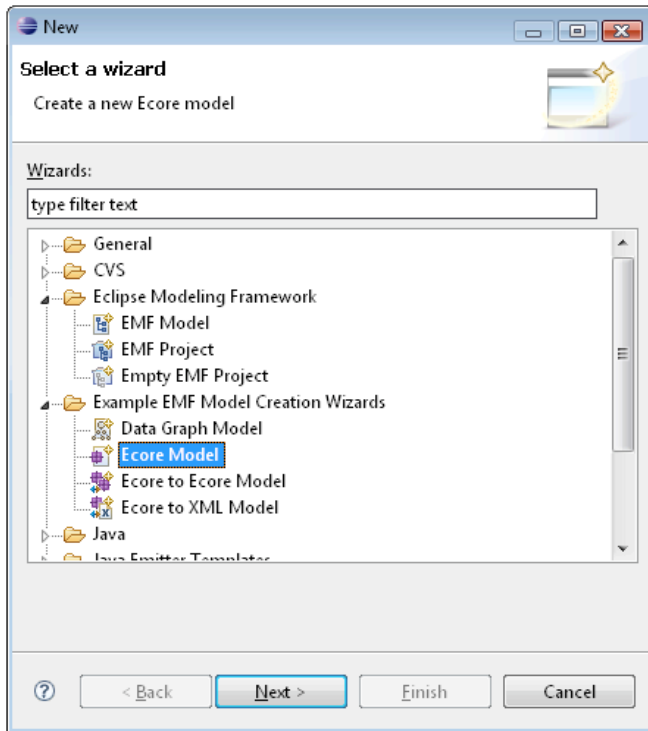
## Create an EMF Model

DiaMeta uses the Eclipse Modelling Framework EMF for specifying language models and generating their implementations.<sup>2</sup> A language model is specified as an EMF model that the editor developer creates by using the EMF modeller. Several tools are available as EMF modeler. In the following, we will use the built-in EMF model editor in the EMF plugin for Eclipse.

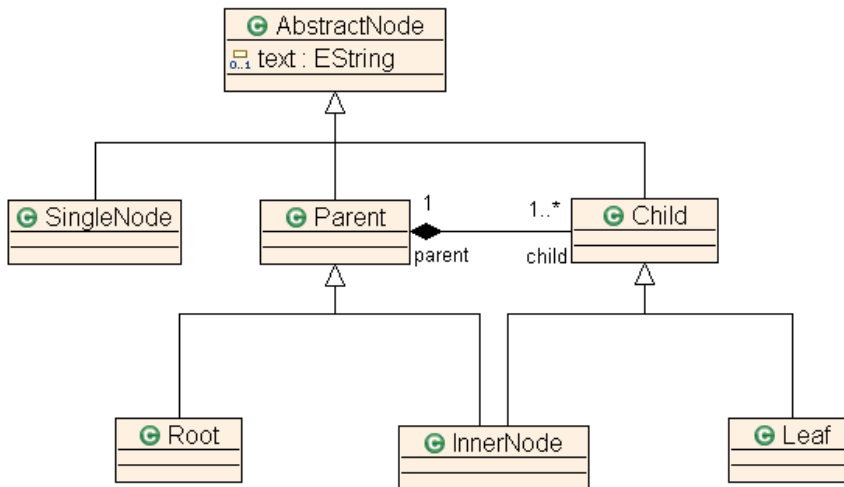
We will create an EMF model and generate the model code.

Create a new Ecore file and name it Tutorial.ecore in the package tutorial:

File >> New >> Other >> Example EMF Model Creation Wizards >> Ecore Model



First of all, we examine the structure of the ecore file that we are going to create.<sup>1</sup>

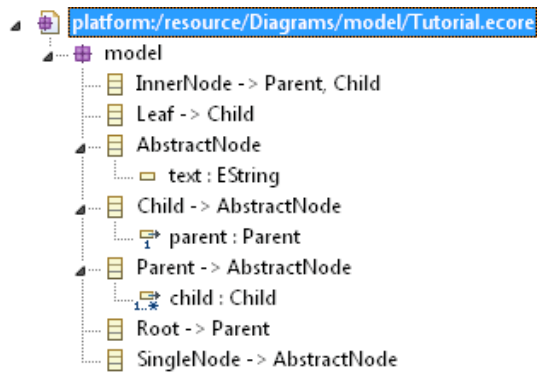


An AbstractNode can be a SingleNode, a Parent node, or a Child node. A Parent can either be a Root node or an InnerNode. A Child node can be an InnerNode or a Leaf node. Child and Parent are connected via the association with the roles child and parent.

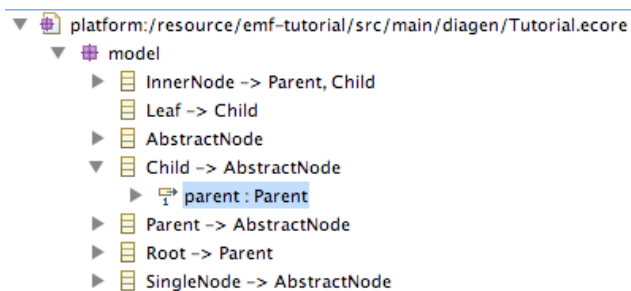
Now create the ecore file exactly as it is shown in the figure below.

<sup>1</sup> The image was drawn with the Topcased EMF editor that can be used as an alternative to the built-in EMF model editor. [Top06]

<sup>2</sup> MOF is not considered here.



You need to carefully set the properties of each element. One speciality of EMF is that you need to set the property “eOpposite” to create a binary association. Below, the properties of the association parent are shown.



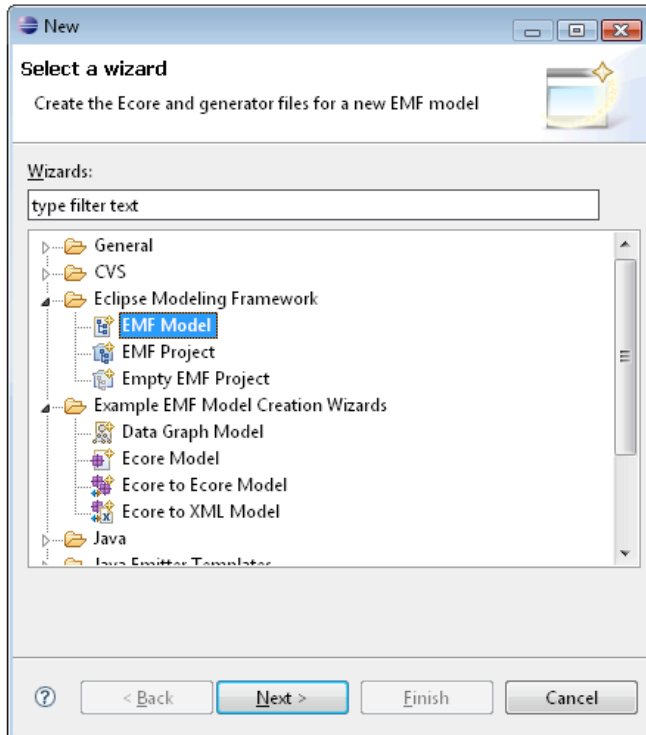
Property	Value
Changeable	<input checked="" type="checkbox"/> true
Container	<input checked="" type="checkbox"/> true
Containment	<input checked="" type="checkbox"/> false
Default Value Literal	<input type="checkbox"/>
Derived	<input checked="" type="checkbox"/> false
EKeys	
eOpposite	child : Child
EType	Parent -> AbstractNode
Lower Bound	<input type="text" value="1"/>
Name	<input type="text" value="parent"/>
Ordered	<input checked="" type="checkbox"/> true
Resolve Proxies	<input checked="" type="checkbox"/> true
Transient	<input checked="" type="checkbox"/> false
Unique	<input checked="" type="checkbox"/> true
Unsettable	<input checked="" type="checkbox"/> false
Upper Bound	<input type="text" value="1"/>
Volatile	<input checked="" type="checkbox"/> false

You can find the complete.ecore file in the project emf-tutorial in the folder ./src/main/. If you don't want to build the whole file from scratch, copy this file to your project. (package tutorial)

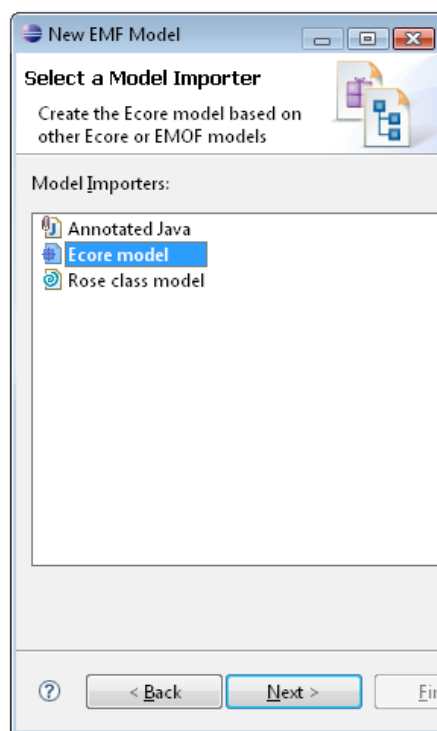
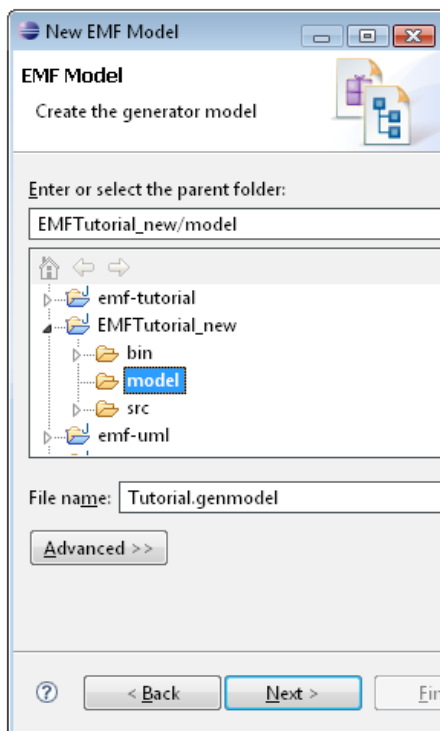
After you finished the EMF model, create the file Tutorial.genmodel in the directory ./model (You have to create this folder first.):

File >> New >> Other >> Eclipse Modeling Framework >> EMF Model

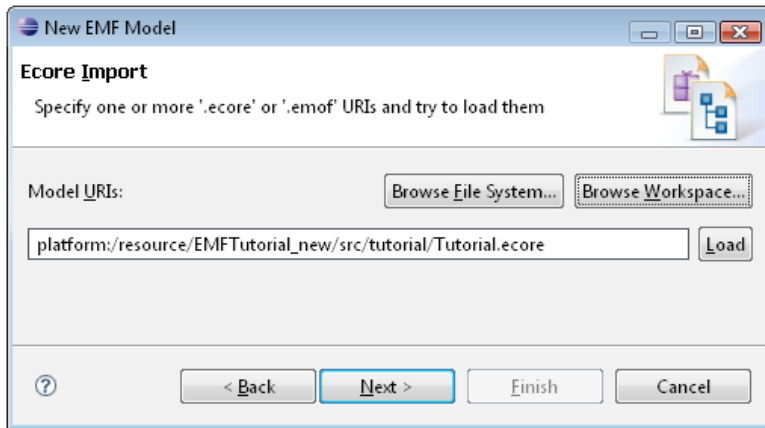
from Tutorial.ecore.



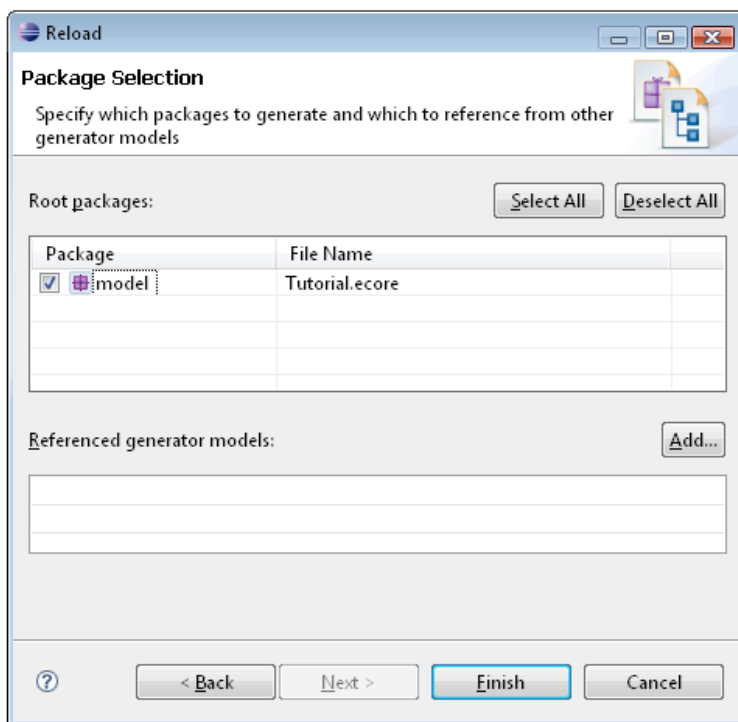
Choose “EMF Model” and push “Next”.



Choose the directory `model`, the “File name” `Tutorial.genmodel`, and push “Next”.  
 Choose “Ecore model” as the “Model Importer” and push “Next”.

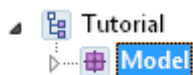


Push “Browse Workspace...” and select your Ecore file Tutorial.ecore.



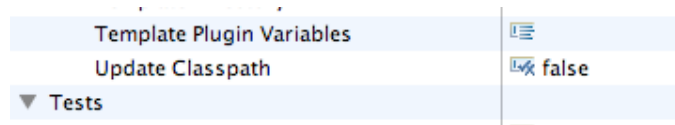
The “Root package” is model.  
Click on “Finish”. The file Tutorial.genmodel should have been created.

Now you have to change the Base Package to tutorial.

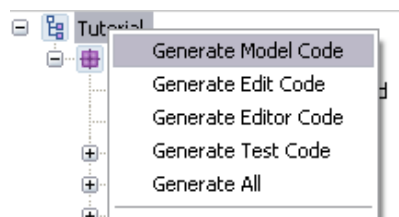


▲ All	
Base Package	tutorial
Prefix	Model
▲ Ecore	
Package	model
▲ Edit	
Disposable Provider Factory	true
▲ Editor	
Generate Model Wizard	true
Multiple Editor Pages	true
▲ Model	

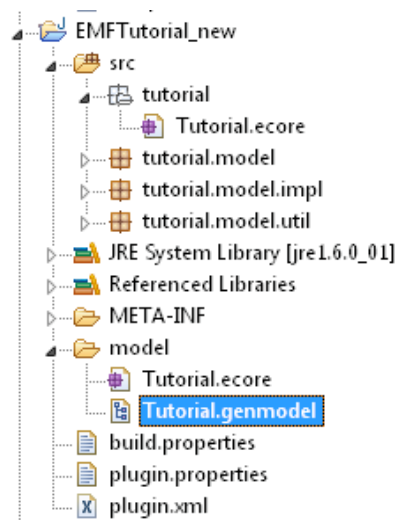
Before you generate the model code, it is important to set “Update Classpath” to false. (Otherwise all classpath entries are removed.)



In the generated EMF Model (Tutorial.genmodel), click on “Generate Model Code”. The EMF compiler creates Java classes (resp. interfaces) for the specified classes.



This should generate the three packages tutorial.model, tutorial.model.impl and tutorial.model.util as shown below.



## Create the DiaMeta specification file

We use the DiaMeta designer for specifying the concrete syntax and the visual appearance of diagram components, i.e., that tree nodes are drawn as circles and tree edges as arrows. The DiaMeta designer generates Java code from this specification. This code, together with Java code created by the EMF compiler and the editor framework, implements an editor for the specified diagram language.

### Option 1: Webpage

Run the DiaMeta designer via the executable jar file

```
designer-and-runtime-xxx.jar
```

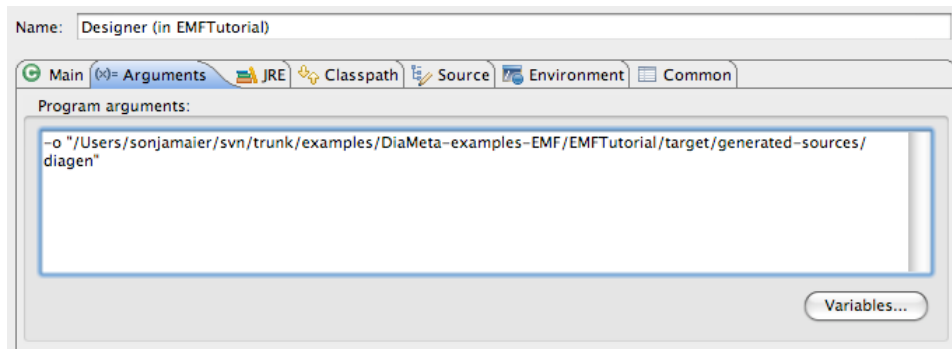
To change the output directory of the generated files, you can use the option “-o”:

```
> java -jar designer-and-runtime-090212.jar -o "./test"
```

## Option 2: Repository

Run the DiaMeta designer via its main class `Designer.java` that can be found in the project `diagen-designer`

To change the output directory of the generated files, you can use the option “-o”:

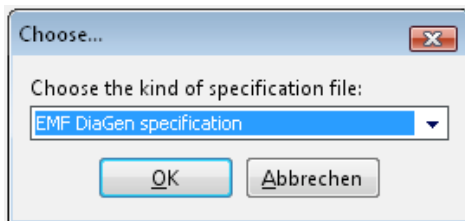


## Creation

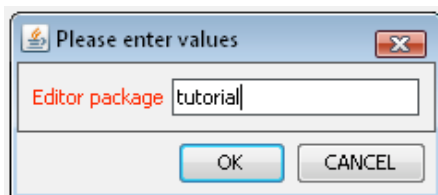
Create a new specification via

File >> New

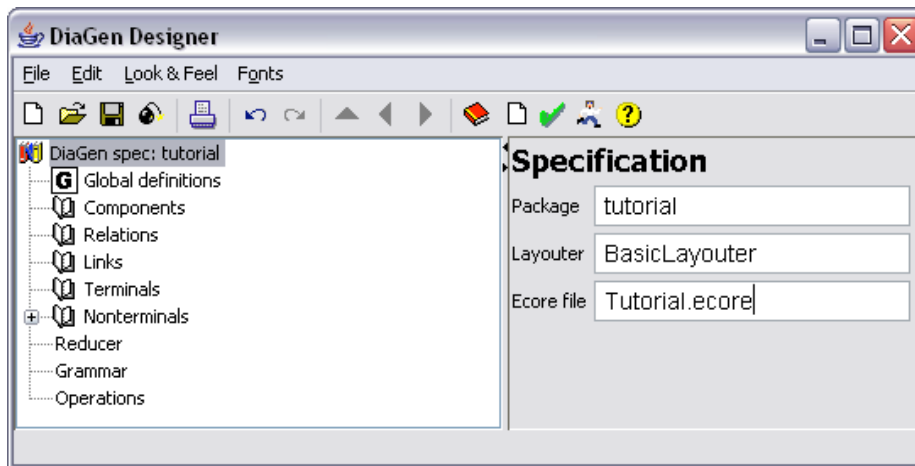
Choose “EMF DiaGen specification” in the appearing popup window and push “OK”.



In the next popup window, type in the package name `tutorial` and push “OK”.



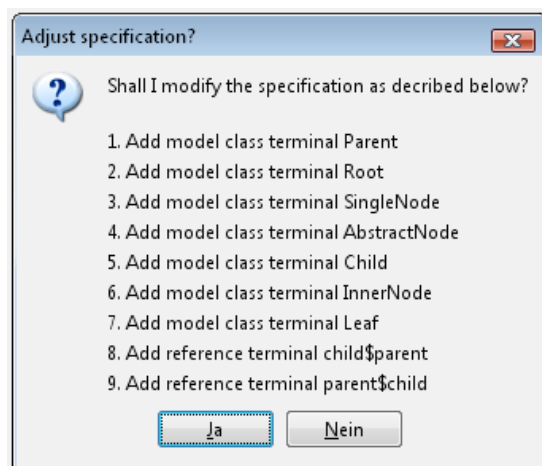
Choose as “Layouter” `BasicLayouter` and as “Ecore file” `Tutorial.ecore`.  
(Alternative: Leave the “Layouter” field blank if you you don’t want a layouter at all.)



Now save the file as `spec.dds` in `<workspace>/EMFTutorial_new/src/tutorial`. (`Tutorial.ecore` and `spec.dds` need to be in the same directory.)



Click on the check mark (green button) in the toolbar. A popup window appears, offering to automatically add component terminals. Push “Ja” and terminals are automatically added.



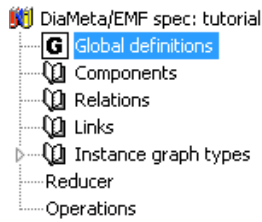
All classes in the EMF model are added as terminals. For each association that can be navigated in both ways, two terminals are added. E.g. for the reference between “Parent” and “Child”, the terminals “child\$parent” and “parent\$child” are added. The role names are separated by the sign “\$”.

(If you don’t want to create the specification file from scratch, open the file `spec.dds` that is located in the directory `./EMFTutorial/src/main/`. Save the file as `spec.dds` in your project, in the package `tutorial`.)

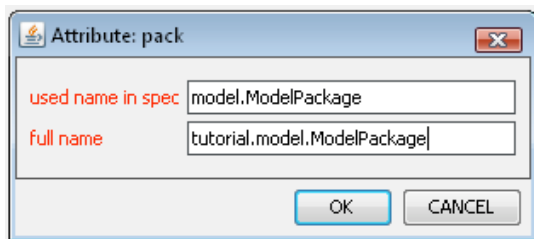
If you want to create the specification file from scratch, you have to complete the parts global definitions, components, relations and reducer.

## Global definitions

Click on Global definitions, switch to the Translations tab and create a new entry.

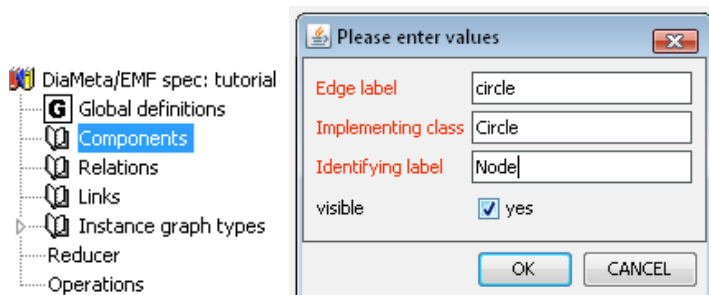


Fill in the field “used name in spec” `model.ModelPackage` and in the field “full name” `tutorial.model.ModelPackage` and click “OK”.



## Components

Now we have to create the visual components. Click on “Components” and create a new visual component.

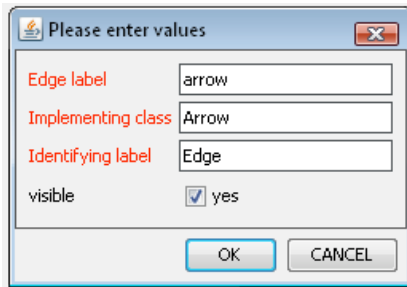


The “Edge label” is `circle`, the “Implementing class” `Circle`, and the “Identifying label” `Node`. (We will reference the edge label `circle` in the section “Reducer”.) Click on “OK” and fill in the missing field. The “EMF model class” is `tutorial.model.AbstractNode`. This class is the corresponding class of the abstract meta model. Instances of the class `Circle` (that are automatically created) are used for layout computation. The name of the class `circle` is also used as the name of nodes in the abstract syntax.

### Component

Edge type	<input type="text" value="circle"/>
Class name	<input type="text" value="Circle"/>
Create button label	<input type="text" value="Node"/>
Create button icon	<input type="text" value="images/default.gif"/>
visible	<input checked="" type="checkbox"/>
Super types	<input type="text"/>
EMF model class	<input type="text" value="tutorial.model.AbstractNode"/>

Create a second visual component.



The “Edge label” is `arrow`, the “Implementing class” `Arrow`, and the “Identifying label” `Edge`. Click on “OK” and fill in the missing field. The “EMF model class” is `tutorial.model.Child`.

Component	
Edge type	arrow
Class name	Arrow
Create button label	Edge
Create button icon	images/default.gif
visible	<input checked="" type="checkbox"/> yes
Super types	
EMF model class	tutorial.model.Child

The field “EMF model class” defines a “mapping” between the visual components and the classes of the abstract model.

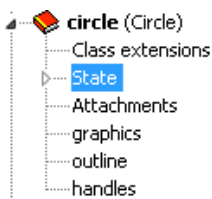
After creation, we will now fill in the missing parts in circle and arrow.

## Circle

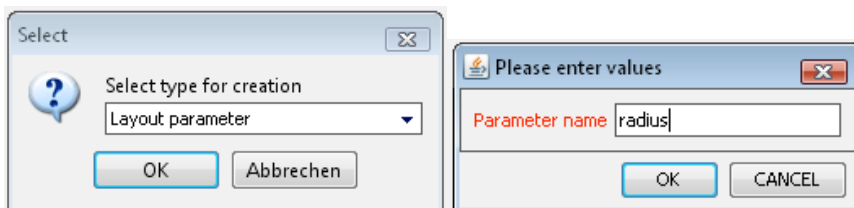
We define the state, the attachments and the graphics of a circle.

### State

We define the state of a circle. Click on “State”.



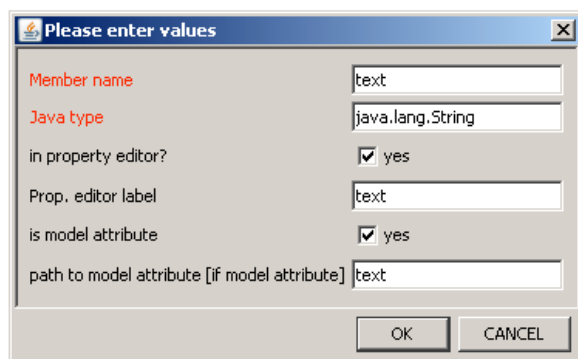
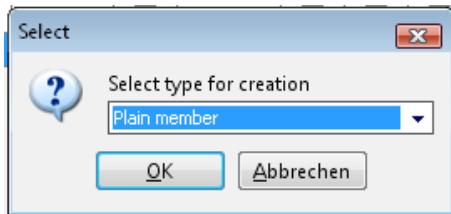
Create the two members `radius` and `text`. `radius` is a “Layout parameter” with the “Parameter name” `radius`.



`text` is a “Plain member” with the “Member name” `text`, the “Java type” `java.lang.String` and the “Prop. editor label” `text`. Besides, you need to set “is model attribute” to yes, because this attribute is contained in the EMF model. The path to this attribute is “text” (, as “text” is an attribute of

the corresponding EMF model class AbstractNode). This way, you may access attributes of the EMF model.

(If you set “is model attribute” to yes, but do not specify the path, you need to implement three methods by hand: to read the attribute (get\_ModelObject\_M\_<Member name>), to write the attribute (set\_ModelObject\_M\_<Member name>) and to update the graphical representation (get\_BaseObject\_M\_<Member name>).)



Later, when you use the editor, you will do the following, in order to create a circle: You choose “Circle” from the menu and click on the position (x[0], y[0]), where you want to create the circle.



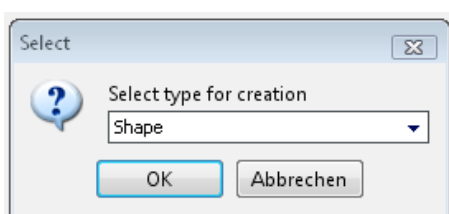
Choose “xPos” as “Ref.x”, “yPos” as “Ref.y”. Those are the parameters that are updated when the circle is moved in the editor. Set the init values as shown below.

Kind	Name	Type	Model attr	Model path	Editable	Edit label	Array	Ref. x	Ref. y	init
Layout parameter	xPos		<input type="checkbox"/>		<input type="checkbox"/>		<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	x[0]
Layout parameter	yPos		<input type="checkbox"/>		<input type="checkbox"/>		<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	y[0]
Layout parameter	radius		<input type="checkbox"/>		<input type="checkbox"/>		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	20
Plain member	text	java.lang.String	<input checked="" type="checkbox"/>	text	<input checked="" type="checkbox"/>		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	""

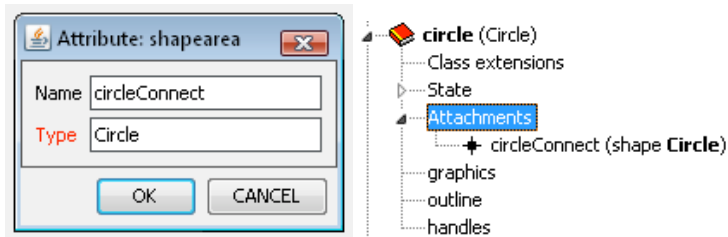
The init value for (xPos, yPos) is (x[0], y[0]), the point on the screen we clicked. “radius” is set to the fixed value “20” and “text” to an empty String.

## Attachments

We define the attachments of a circle. Click on “Attachments”. Create a new attachment of type Shape.

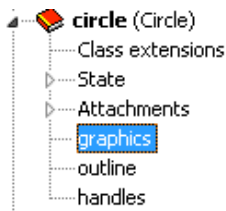


Name the attachment `circleConnect`, and give it the type `Circle`. We will reference the attachment in “Relations”.

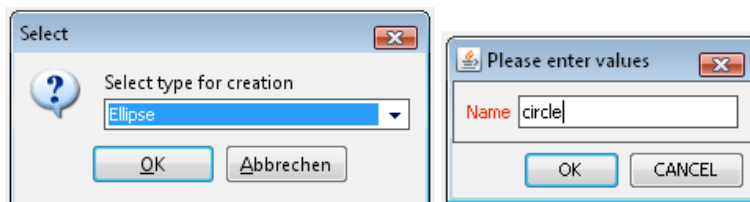


## graphics

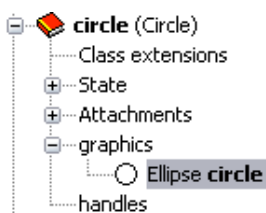
Click on “graphics” and create a new graphic.



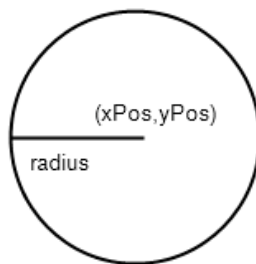
Click on the button above. Select the type “Ellipse” and click on “OK”. Name the ellipse “circle” and click on “OK”.



Click on the newly created ellipse and fill in values for “x”, “y”, “width” and “height”.



Ellipse	
Name	circle
Border line color	feedback-color
Border line stroke	default
Fill color	not-filled
Local code	
x	xPos-radius
y	yPos-radius
width	2*radius
height	2*radius



The circle has the radius “radius” and the center (“xPos”, “yPos”). (The point you clicked in order to create the circle.)

## Arrow

Now we define the state, the attachments, the graphics and the handles of an arrow.

### State

We define the state of an arrow. Click on “State”.



Create the four “Layout Paramters” xPos1, yPos1, xPos2 and yPos2. They represent the start position and the end position of an arrow.

Later, when you use the editor, you will do the following, in order to create an arrow: You choose “Arrow” from the menu. Then you click on the position (x[0], y[0]), where the arrow should start and then on the position (x[1], y[1]), where the arrow should end.

Add the “Creation prompts” “StartPosition” and “EndPosition”. (The two clicks we perform when we create an arrow.)



Create the layout parameter “xPos1”, “yPos1”, “xPos2” and “yPos2”. Choose “xPos1” as “Ref.x” and “yPos1” as “Ref.y”. This is the point that is updated when the arrow is moved. Set the init values and fill in the two lines in “when repoint is moved”.

Creation prompt

Text \*

StartPosition

EndPosition

Initially open property editor

Members

Kind	Name	Type	...	Edit...	...	...	...	init
Layout par...	xPos1		<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	x[0]
Layout par...	yPos1		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	y[0]
Layout par...	xPos2		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	x[1]
Layout par...	yPos2		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	y[1]

where

when repoint is moved

```
yPos2=yPos2+y-yPos1;
xPos2=xPos2+x-xPos1;
```

The init value for (xPos1, yPos1) is (x[0], y[0]), the first point on the screen we clicked. The init value for (xPos2, yPos2) is (x[1], y[1]), the second point on the screen we clicked.

When you move the arrow, the point (xPos1, yPos1) is automatically updated. The point (xPos2, yPos2) needs to be updated by hand via the two lines

```
yPos2=yPos2+y-yPos1;
xPos2=xPos2+x-xPos1;
```

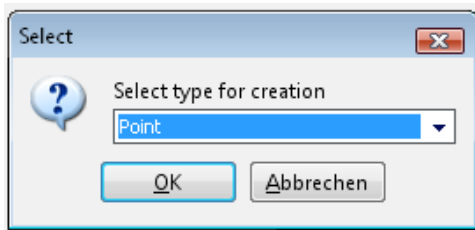
## Attachments

We define the attachments of an arrow. Click on “Attachments”, and add two attachments: The start point of an arrow and the end point.

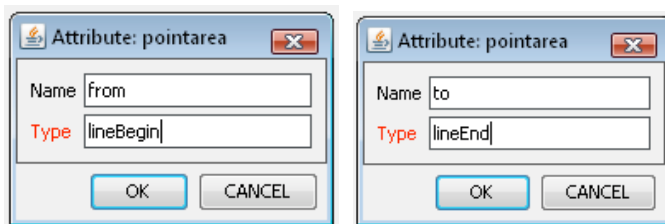




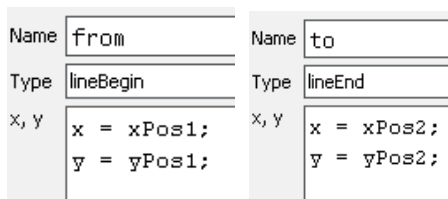
Click on the button above. Select the type “Point” and click on “Ok”.



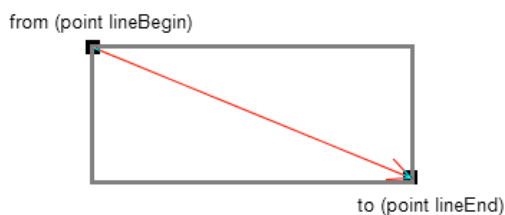
Choose the Name `from` and the Type `lineBegin` for the first attachment, and the Name `to` and the Type `lineEnd` for the second attachment.



Fill in the following lines.



The attachment “from” is exactly at the position where the arrow starts, and the attachment “to” exactly at the position where the arrow ends. We will reference these attachments in “Relations”.

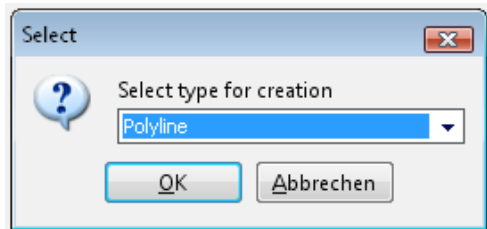
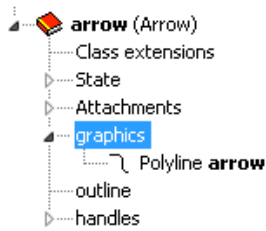


## graphics

Click on “graphics” and **fill in the following line**.

```
drawPreview (Graphics2D g2d, EditorPane panel, Point2D[] pos, int num, Point2D cur )
diagen.editor.lib.Polyline.drawPreview(g2d, pos, num, cur);
```

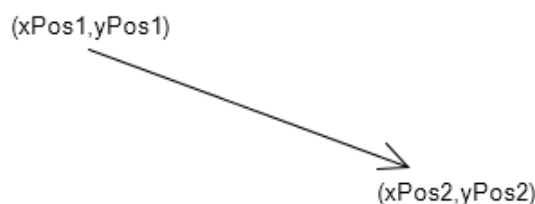
Create a new graphic. Select the type “Polyline” and click on “OK”. Choose the Name `arrow` and click on “OK”.



Click on the newly created polyline and fill in values for “xStart”, “yStart”, “xEnd” and “yEnd”. Choose as end line head “plain”.

The arrow has the start point (“xPos1”, “yPos1”) (The first point you clicked in order to create the arrow.) and the end point (“xPos2”, “yPos2”). (The second point you clicked.) The arrow has a plain arrowhead. (The default setting is that the arrow has NO arrowhead.)

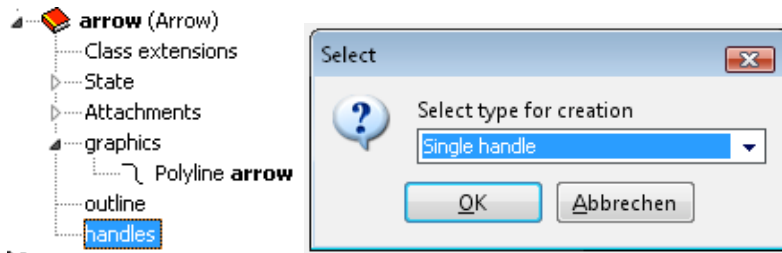
Polyline	
Name	arrow
Line color	feedback-color
Line stroke	default
start line head style	none
end line head style	plain
Local code	
xStart	xPos1
yStart	yPos1
#inner points	
xInner[i]	
yInner[i]	
xEnd	xPos2
yEnd	yPos2



## handles

We want to provide three alternatives to move an arrow. You may move the arrow as a whole, move the start point of an arrow or move the end point of an arrow. Click on “handles”. The first alternative is created, if you select “has move handle?”. The other alternatives have to be created by hand.

Create two new handles. Choose the type “Single handle” for both handles and name them “lineB” and “lineE”.



Fill in values for “Point” and “When changed...”.

**Handles**

has move handle?

Type	Name	Point	When changed...
Single handle	lineB	x = xPos1; y = yPos1;	xPos1 = x; yPos1 = y;
Single handle	lineE	x = xPos2; y = yPos2;	xPos2 = x; yPos2 = y;

The first handle is placed at the point, the arrow begins, and the second handle at the point, the arrow ends.

x = xPos1; y = yPos1;

resp.

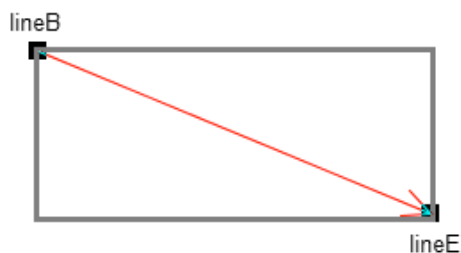
x = xPos2; y = yPos2;

When the handle is moved, the location of the arrow is updated.

xPos1 = x; yPos1 = y;

resp.

xPos2 = x; yPos2 = y;

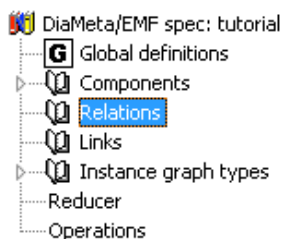


**Reload the file “spec.dds” now!** (Otherwise, the attachment types are not updated, and you may not define relations in the following.) (Because of a known bug. ;-)

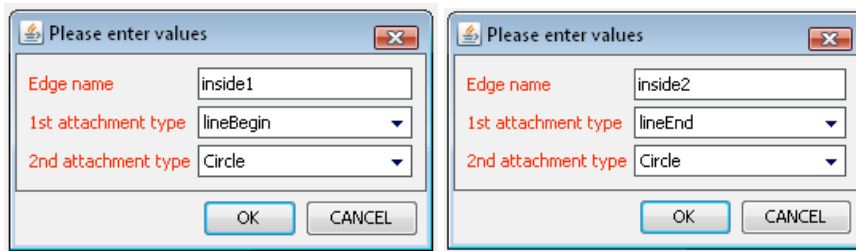
File >> Reopen >> 1...spec.dds

## Define Relations

Click on “Relations” and create the two relations “inside1” and “inside2”.

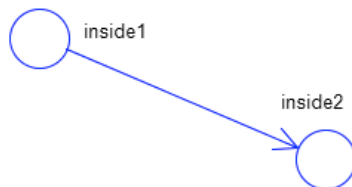


The relation “inside1” is between the attachment “lineBegin” and the attachment “Circle”. The relation “inside2” is between the attachment “lineEnd” and the attachment “Circle”. (The attachment “Circle” was automatically created, the attachments “lineBegin” and “lineEnd” were added earlier.)



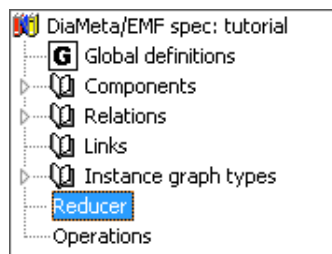
Edge type	1st attachment type	2nd attachment type
inside1	lineBegin	Circle
inside2	lineEnd	Circle

With the relation “inside1” we are able to decide, whether an arrow starts in a circle, or not. With the relation “inside2” we are able to decide, whether an arrow ends in a circle, or not. We will use these relations in “Reducer”.



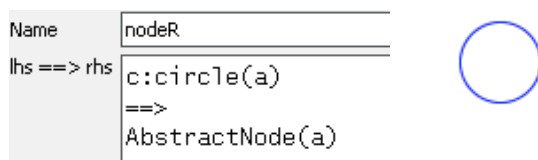
## Reducer

We define the reducer rules. Click on “Reducer” and create the rules “nodeR” and “edgeR” to reduce the components “circle” and “arrow”.

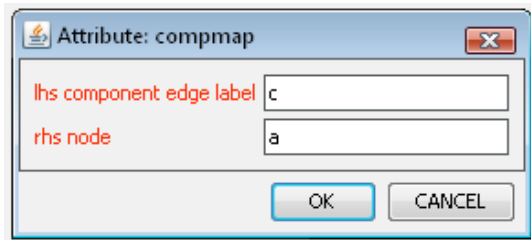


Add the following lines to the rule “nodeR”:

```
c:circle(a)
==>
AbstractNode(a)
```



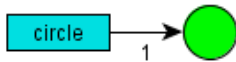
Add in the tab “Comp map” the following entry.



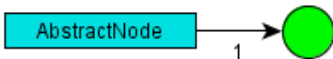
Condition	Action	Constraints	Comp map
lhs component edge label			rhs node
c			a

Here, a mapping between the model object of the diagram component of “ar” (the arrow) and the abstract syntax object (here: the AbstractNode that corresponds to “d”) is created. (For each abstract syntax object, a unary hyperedge is created.)

Each circle will be transformed internally as follows.<sup>2</sup>



The component “circle” is transformed into the terminal “AbstractNode”. The terminal represents the class “AbstractNode”.

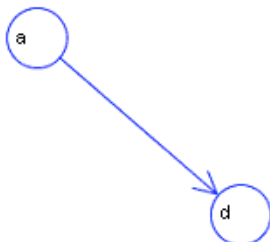


Internally, “AbstractNode” is the node . This is visualized here in a different way. (Due to the fact that DiaMeta is based on hypergraphs.)

Add the following lines to the rule “edgeR”. (In the rule, `circle(d)` is **not** required.)

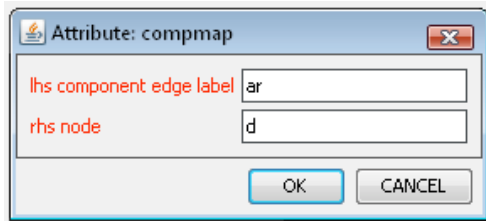
```
ar:arrow(b,c) inside1(b,a) inside2(c,d) circle(d)
==>
child$parent(a,d)
```

Name	edgeR
lhs ==> rhs	ar:arrow(b,c) inside1(b,a) inside2(c,d) circle(d) ==> child\$parent(a,d)



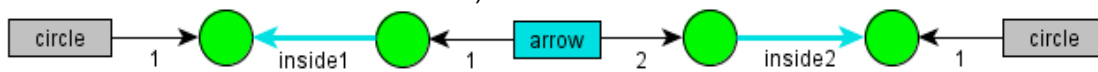
Add in the tab “Comp map” the following entry .

<sup>2</sup> Here, the two files `hgm.gml` and `reduced.gml`, that were automatically generated, are visualized. The images were drawn with the yED Graph Editor. [yFi06]



Condition	Action	Constraints	Comp map
lhs component edge label			rhs node
ar			d

Hence, an arrow that connects two circles will be transformed internally as follows. (Arrows that do not connect two circles are not transformed.)



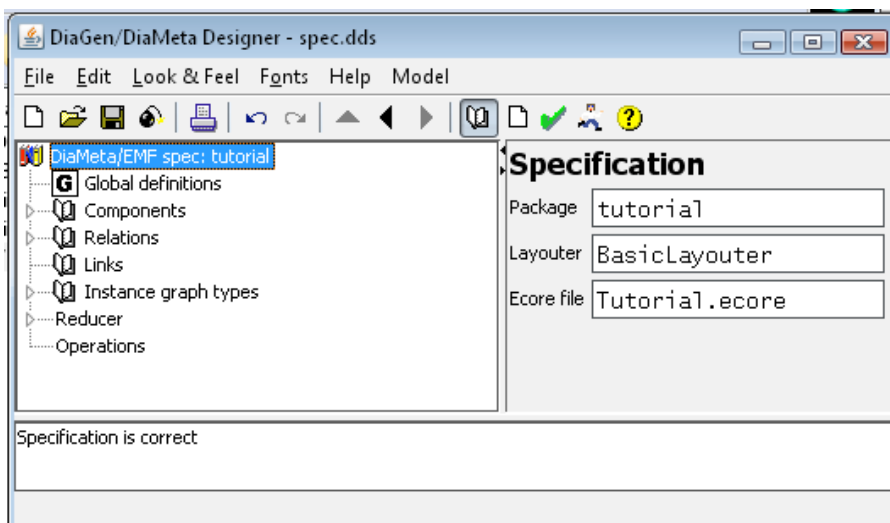
The gray components are translated via the rule “nodeR”. The rule “edgeR” transforms the component “arrow” that starts inside “circle a” and ends inside “circle d” into the terminal “child\$parent”, that connects “Circle a” and “Circle d”.



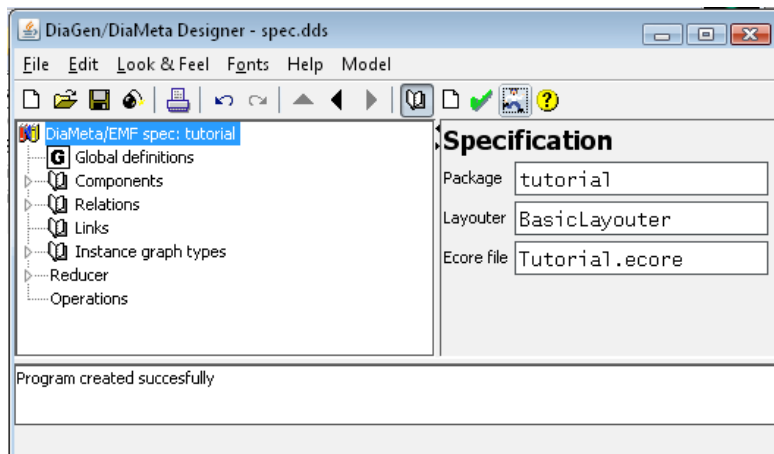
No all options for defining rules are described in this tutorial. You may for instance add a negative application condition (NAC) to your rule. E.g., the following rule expresses that a circle is only reduced if there is no arrow attached:

```
c:circle(a)
- {inside(_, a)}
==>
AbstractNode(a)
```

After the specification file is completed, click on the check mark (green button) again. The status window of the DiaMeta designer should now show “Specification is correct”.

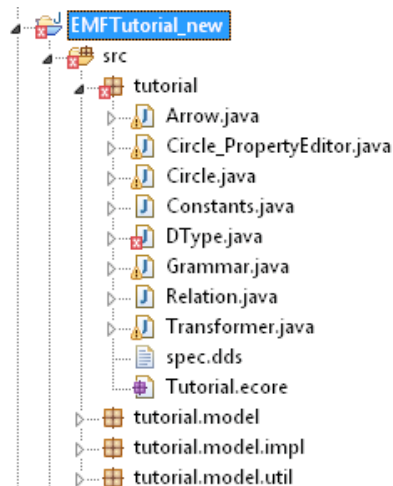


Click on the button with the small stickperson. The status window of the DiaMeta designer should now show “Program created successfully”.



The DiaMeta designer generates the classes that are responsible for visualizing diagram components, interacting with them in the editor, and language specific classes for diagram analysis. You can find the generated Java files in the package `tutorial`.

**!! Press F5 (in Eclipse) to refresh the workspace. !!** (Otherwise the editor will not work later.)



The project still contains an error. This is because the layouter is still missing. (Earlier we specified that we are going to define one.)

Now, copy the main class `Editor.java` from the project `emf-tutorial` to your project in the package `tutorial`. Right now it is not generated automatically. It contains the following lines.

```
package tutorial;

import java.util.prefs.Preferences;

public class Editor extends diagen.editor.ui.swing.Editor {

    private Editor ( Preferences prefs ) { super(prefs); }

    public static void main ( String[] args ) {
        Preferences prefs =
            Preferences.userNodeForPackage(DType.INSTANCE.getClass());
        Editor editor = new Editor(prefs);
        editor.startEditor(DType.INSTANCE,
            "Tutorial Editor (EMF)", false);
    }
}
```

## Write a Layouter (optional)

You can skip this step if you left blank field “Layouter” when you created the file “spec.dds”. The generated editor then does not provide automatic layout.

Otherwise, only one thing is missing. The file `BasicLayouter.java` (package `tutorial`). This class is responsible to do the layout for the editor. Copy the file `BasicLayouter.java` from the project `emf-tutorial` to your project in the package `tutorial`. It includes the following code.

```
package tutorial;

import java.awt.geom.Point2D;

public class BasicLayouter implements MetaModelLayouter<EObject> {

    public void layout(LayoutData<EObject> data) {

        // iterate over all components of the current diagram
        for (DiagramComponent c : data.getComponents())

            // in case it is an arrow, do some adjustments
            if (c instanceof Arrow) {
                Arrow arrow = (Arrow) c;

                // retrieve the corresponding model object
                final Child child = arrow.getModelObject();

                // navigate in the model
                if (child == null || child.getParent() == null)
                    continue;
                final Parent parent = child.getParent();

                // retrieve the corresponding diagram component
                Circle to = data.getComponent(child, Circle.class);
                Circle from = data.getComponent(parent, Circle.class);

                if (to == null || from == null)
                    continue;

                // retrieve the values of the layout attributes
                // Do NOT use (e.g.) from.get_P_xPos().getVal() !
                double x1 = data.getVal(from.get_P_xPos());
                double y1 = data.getVal(from.get_P_yPos());
                double x2 = data.getVal(to.get_P_xPos());
                double y2 = data.getVal(to.get_P_yPos());

                // do some calculations
                double d = Point2D.distance(x1, y1, x2, y2);
                double r1 = data.getVal(from.get_P_radius()) / d;
                double r2 = data.getVal(to.get_P_radius()) / d;

                // update attribute values
                // ( only if arrow wasn't changed by the user)
                if (!(data.isChanged(arrow.get_P_xPos1()) || data.isChanged(arrow.get_P_yPos1()))) {
                    data.setVal(arrow.get_P_xPos1(), r1 * x2 + (1 - r1) * x1);
                    data.setVal(arrow.get_P_yPos1(), r1 * y2 + (1 - r1) * y1);
                }
                if (!(data.isChanged(arrow.get_P_xPos2()) || data.isChanged(arrow.get_P_yPos2()))) {
                    data.setVal(arrow.get_P_xPos2(), r2 * x1 + (1 - r2) * x2);
                    data.setVal(arrow.get_P_yPos2(), r2 * y1 + (1 - r2) * y2);
                }
            }
    }
}
```

The following statements might be interesting:

```
data.getComponents()
```

Returns a collection of all diagram components of the current diagram.

```
arrow.getModelObject()
```

Returns the model object (instance of EMF model class) that corresponds to the diagram component `arrow`. The mapping was created via the reducer rules (in the tab “comp map”).

```
data.getComponent(child, Circle.class)
```

Returns the diagram component of type “Circle” that corresponds to the model object `child`. (There might be attached more than one diagram component. E.g., `child` also has an “Arrow” object.)

```
data.getVal(from.getP_xPos())
```

Returns the value of the layout attribute `xPos` of the diagram component `from`.

```
data.setVal(arrow.getP_xPos1(), newVal)
```

Updates the value of the layout attribute `xPos1`. The new value is `newVal`. (Subsequent invocation of `data.getVal(from.getP_xPos())` will return `newVal`.)

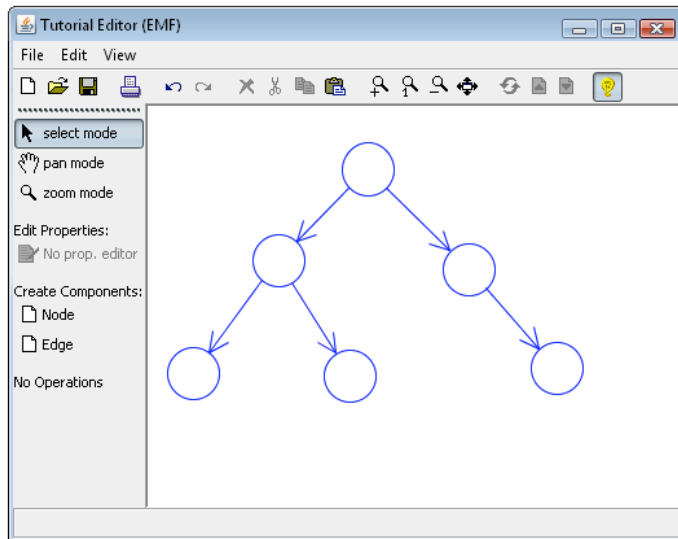
```
data.isChanged(arrow.getP_xPos1())
```

Returns true if the attribute `xPos1` was changed. The attribute may have been changed by the user (e.g. he moves the arrow) or by the layouter. The method returns false if the attribute is unchanged.

## Run the Editor

At this point, no more errors should be listed for the project. That means we created the editor successfully. Now we are able to run the editor.

To run an editor, you need to execute the main class `Editor.java`. The figure below shows a screenshot of the editor we created.



## Enable GML Output (optional)

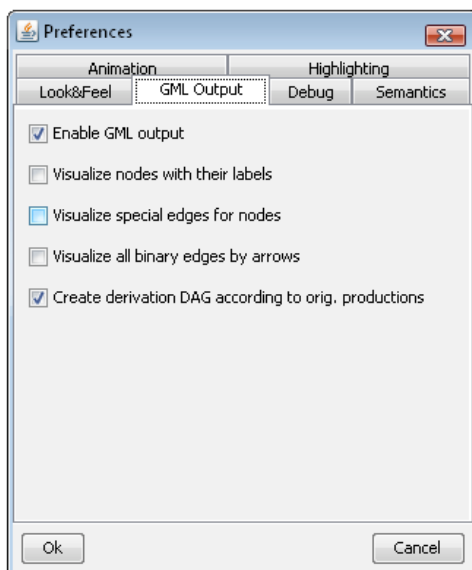
To enable GML output, open the Preferences

Edit >> Edit Preferences >> GML Output

and set the check mark "Enable GML output". The files

`hgm.gml` and `reduced.gml`

are now automatically created in the folder `.\src\EMFTutorial\` each time you change the diagram.



**Congratulations! You just finished the tutorial.**

## 7. References

- [Min06] Mark Minas. Generating Meta-Model-Based Freehand Editors, appears in Electronic Communications of EASST, Volume 1, Proc. Of 3rd International Workshop on Graph Based Tools,2006
- [yFi06] yFiles [http://www.yworks.com/en/products\\_yfiles\\_about.htm](http://www.yworks.com/en/products_yfiles_about.htm)
- [Top06] Topcased <http://www.topcased.org/>
- [Bat06] Batik SVG Toolkit <http://xmlgraphics.apache.org/batik/>
- [Gml06] The GML File Format <http://www.infosun.fmi.uni-passau.de/Graphlet/GML/>
- [Emf07] EMF <http://www.eclipse.org/modeling/emf/>
- [Svn07] Subversion <http://subversion.tigris.org/>
- [Mvn07] Maven <http://maven.apache.org/>