

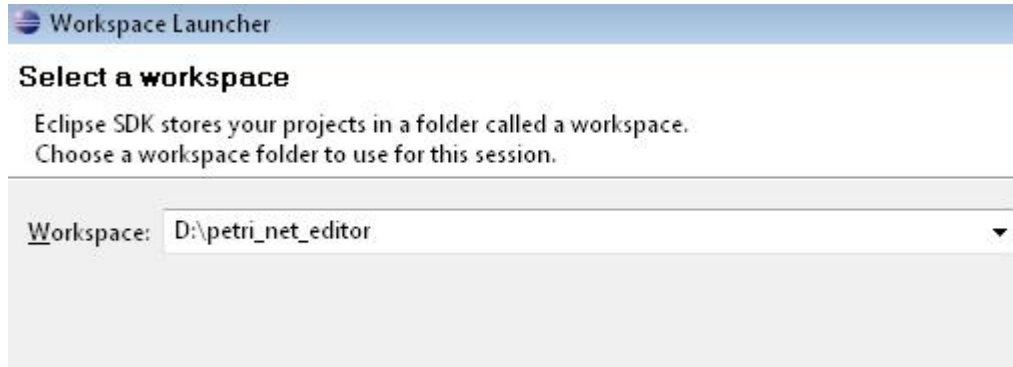
Developing a DSL with Poseidon for DSLs

This documents explains how to develop a graphical modeling language, or Domain Specific Language (DSL) using Poseidon for DSLs. It is a step by step tutorial and shows all steps needed for creating a complete modeling language. As an example we will be using the language of Petri Nets. We start from a Petri net editor metamodel and finish with a complete editor for Petri nets.

1. Developing a DSL with Poseidon for DSLs	1
1. Setting up the environment	2
2. Creation of a Metamodel for Petri Nets	6
1. Ecore Model creation with Poseidon for Ecore.....	6
2. Generating code for the metamodel	12
3. Generation of an empty editor.....	15
4. Creation of all nodes and edges using default implementations and ignoring the look of elements	16
5. Making elements look like they have to look.....	20
6. Adding rapid buttons.....	24
7. Adding edge rules.....	26
8. Customizing Add menu.....	27
9. Customizing Attributes tab in Model browser (Properties model)	29
10. Customizing the code.....	31
11. Advanced: Creating nodes with compartments	34
12. Changing splash screen	42
13. Building distribution of your editor	42
14. Conclusion.....	43

Setting up the environment

To get the environment to run, we provide a pre-configured Eclipse distribution and a zipped set of projects (so called poseidon template). Simply unzip the provided eclipse archive. Then start eclipse and open a new workspace which you are going to use for the editor. For example, "D:\petri_net_editor"



Then import the also provided template project using "File->Import->General->Existing Projects into Workspace", select the provided archive

Import Projects

Select a directory to search for existing Eclipse projects.



Select root directory:

Select archive file:

Projects:

<input checked="" type="checkbox"/> generators (generators)	<input type="button" value="Select All"/> <input type="button" value="Deselect All"/> <input type="button" value="Refresh"/>
<input checked="" type="checkbox"/> graphics (graphics)	
<input checked="" type="checkbox"/> metamodel (metamodel)	
<input checked="" type="checkbox"/> model_browser (model_browser)	
<input checked="" type="checkbox"/> Poseidon (Poseidon)	
<input checked="" type="checkbox"/> poseidon_for_ecore (poseidon_for_ecore)	
<input checked="" type="checkbox"/> poseidon_generator (poseidon_generator)	
<input checked="" type="checkbox"/> repository (repository)	
<input checked="" type="checkbox"/> resources (resources)	

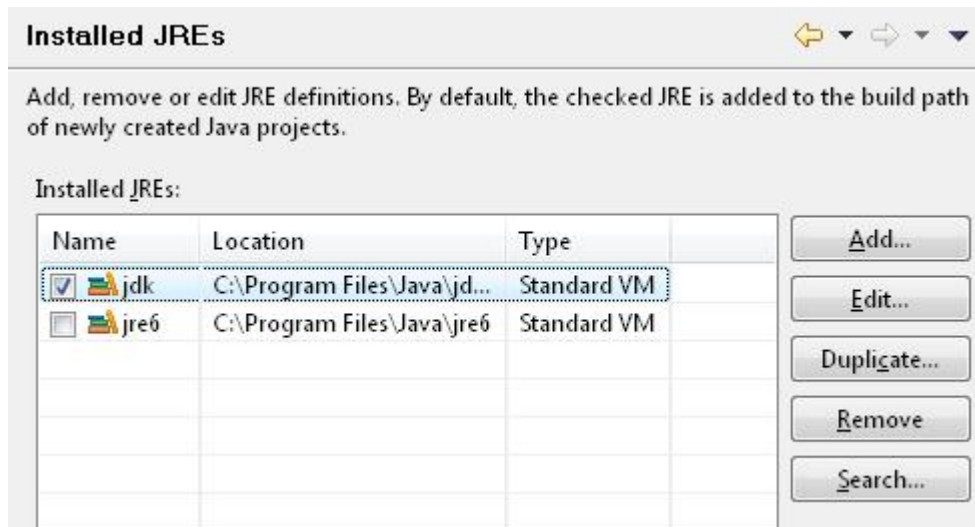
Copy projects into workspace

Working sets

Add project to working sets

Working sets:

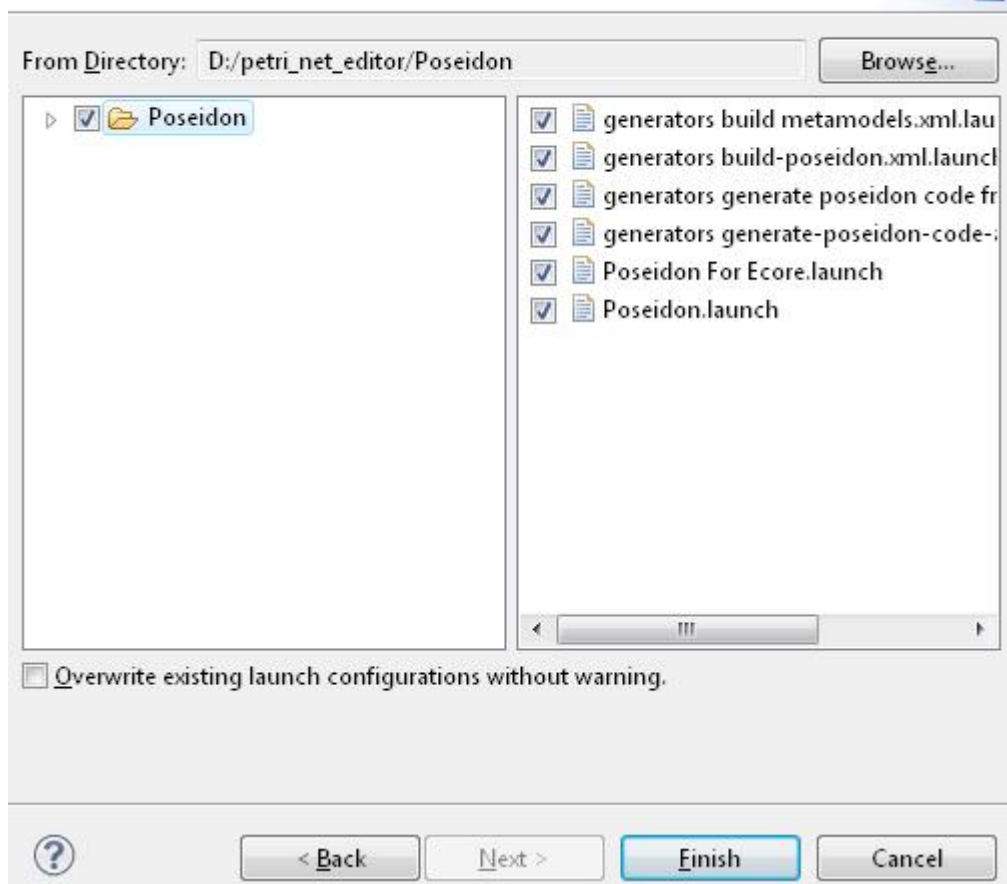
The loaded projects will be imported and built. There will be a few errors in the workspace. Don't worry, they will disappear after the first code generation. To work with Poseidon for DSLs, you need java SDK 1.5 or higher. In Eclipse, please open JRE settings (Window -> Preferences -> Java -> Installed JREs). Click "Add" button and add a Java SDK 1.5 or higher to the list of available JREs. Please be sure that you give a name ""jdk" to this jdk, it is important because this name is used in some workspace settings. Do not forget to check the checkbox for that JRE.



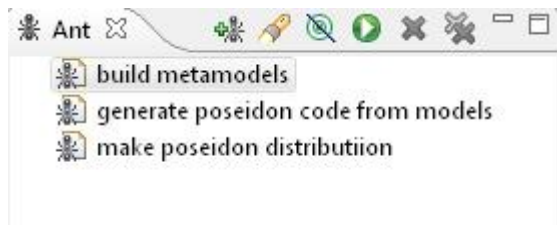
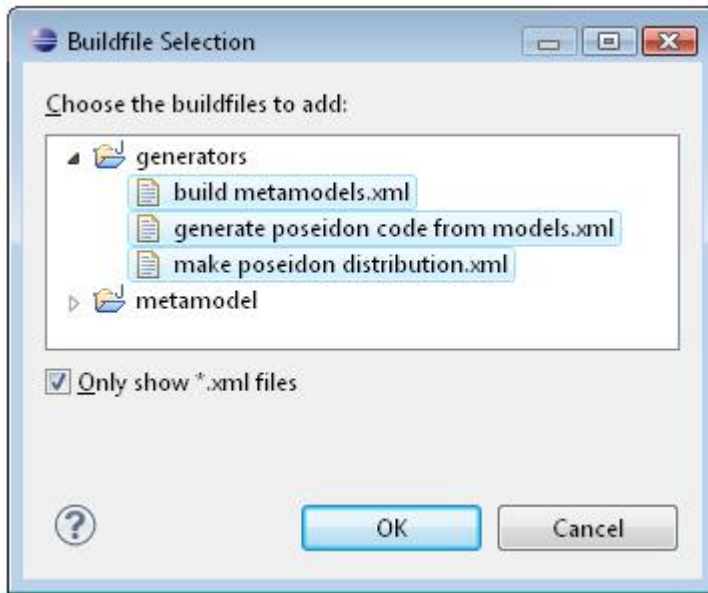
Now please import the launch configurations into your workspace. You will need them later to start code generation and different builds. So, use "File->Import->Run/Debug->Launch Configurations". Select all launch configurations from project "Poseidon":

Import Launch Configurations

Import launch configurations from the local file system



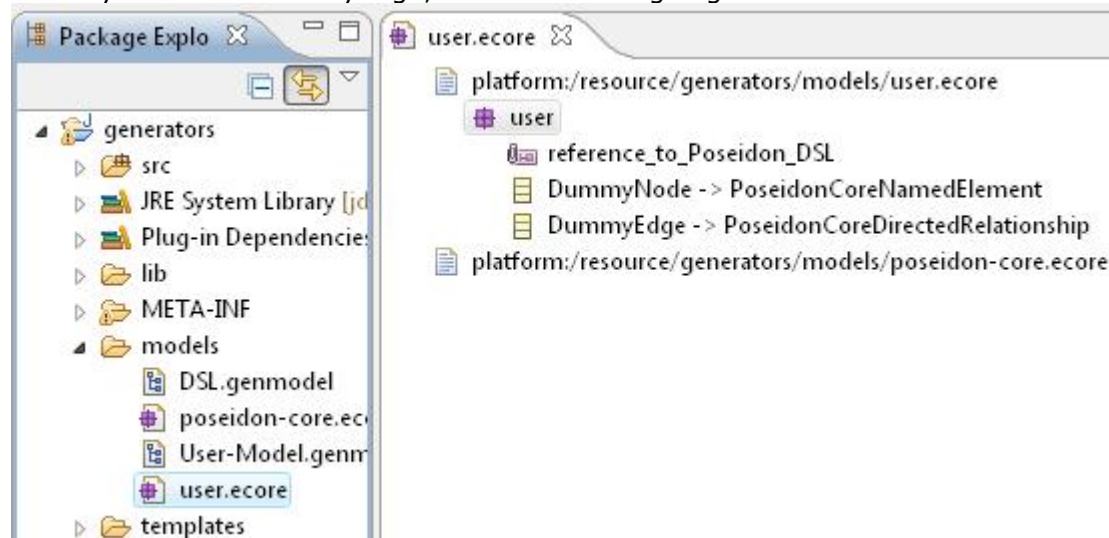
And, the last step of configuration. In the Ant view on the right of your workbench, add all ant build files from project "generators":



Creation of a Metamodel for Petri Nets

Ecore Model creation with Poseidon for Ecore

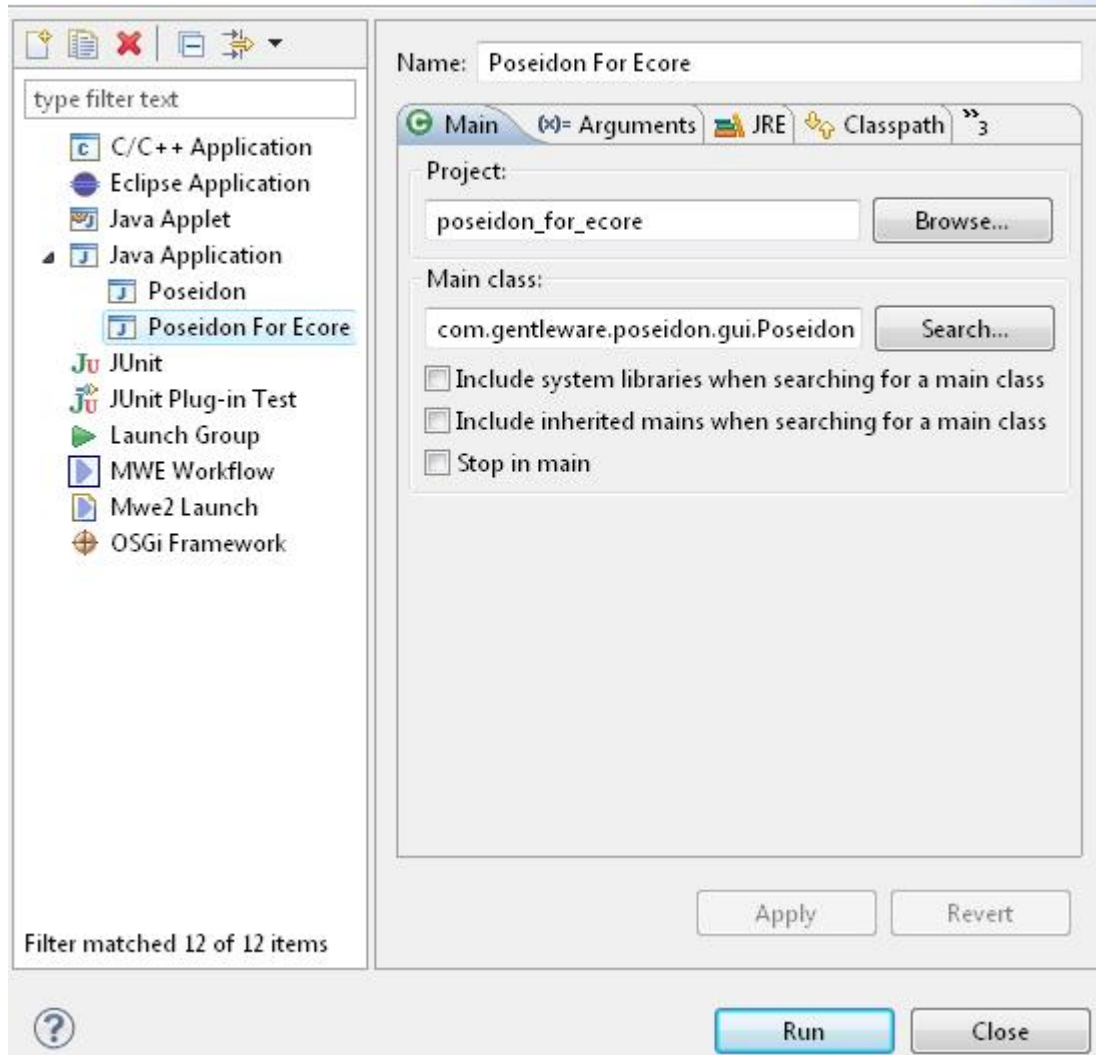
Throughout this tutorial we are going to use a simple metamodel. The metamodel defines the semantic meaning of the elements of our DSL. It is created according to the ECore metamodeling language, which is part of the Eclipse Modeling Framework (EMF). Creation of this metamodel will be the first step in this tutorial. We start from a template workspace which we provide to start a new project. This workspace contains an empty metamodel which you should start from. In your workspace, find project **generators** and open file **models/user.ecore**. This model contains 2 dummy elements already - DummyNode and DummyEdge, but we are not going to use them.



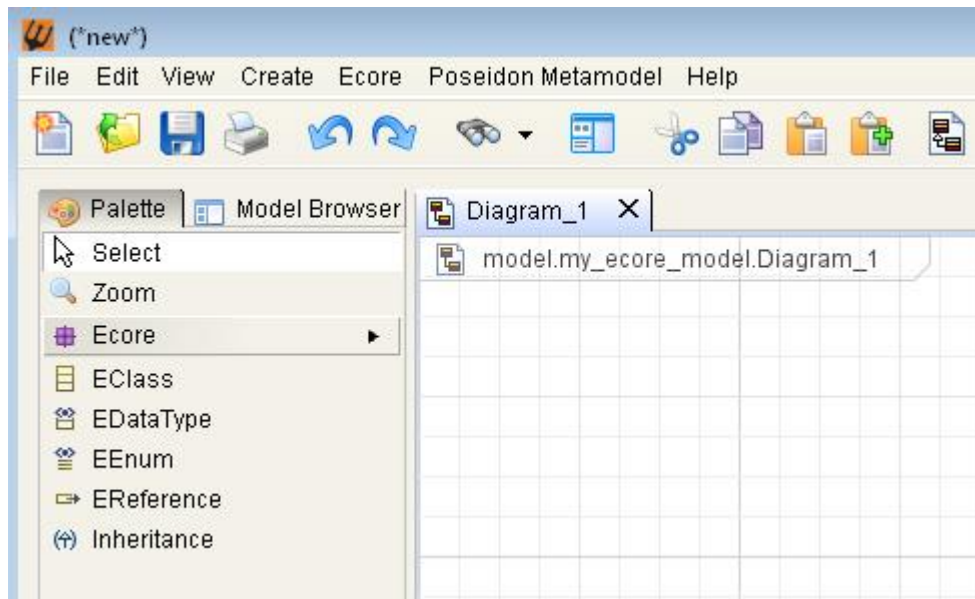
Now we are going to create our own metamodel instead of this dummy model. We will use a graphical editor **Poseidon for Ecore** to do that. This is a graphical editor for ECore models created with **Poseidon for DSLs** framework. It also has special tools for creation of ecore models which can be used as a metamodel for **Poseidon for DSLs** editors. Start **Poseidon for Ecore** from your Eclipse workspace. To do that, in the main menu select "Run -> Run Configurations...", then expand "Java Application" and find "Poseidon For Ecore" configuration there.

Create, manage, and run configurations

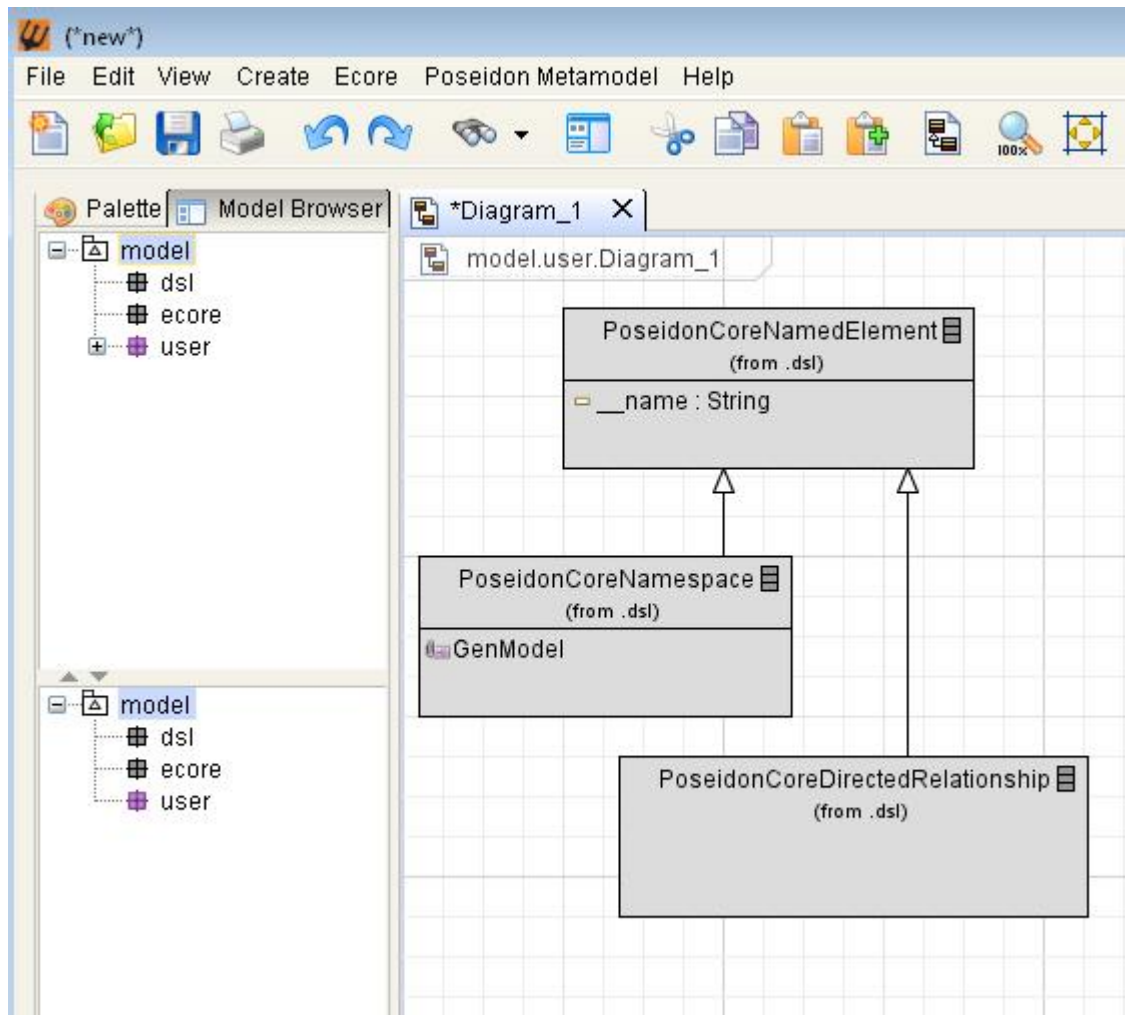
Run a Java application



Now start **Poseidon For Ecore** by double-clicking on the configuration or using button "Run". Here is how the **Poseidon For Ecore** should look like:



Poseidon For Ecore is a fully-functional graphical Ecore editor, which allows you to create and work with your own ecore models. It also has an extension which makes it easier for you to create an ecore model which can be used as a metamodel in **Poseidon for DSLs**. So, now we are going to use it to create an ecore metamodel, which will be used in the "**Poseidon for Petri nets**" editor. In the main menu of **Poseidon For Ecore**, choose "Poseidon Metamodel -> Start creating metamodel for Poseidon". This action imports Poseidon for DSLs core metamodel (package "dsl") into the project and also creates a new diagram for you, with 3 elements already on the diagram.



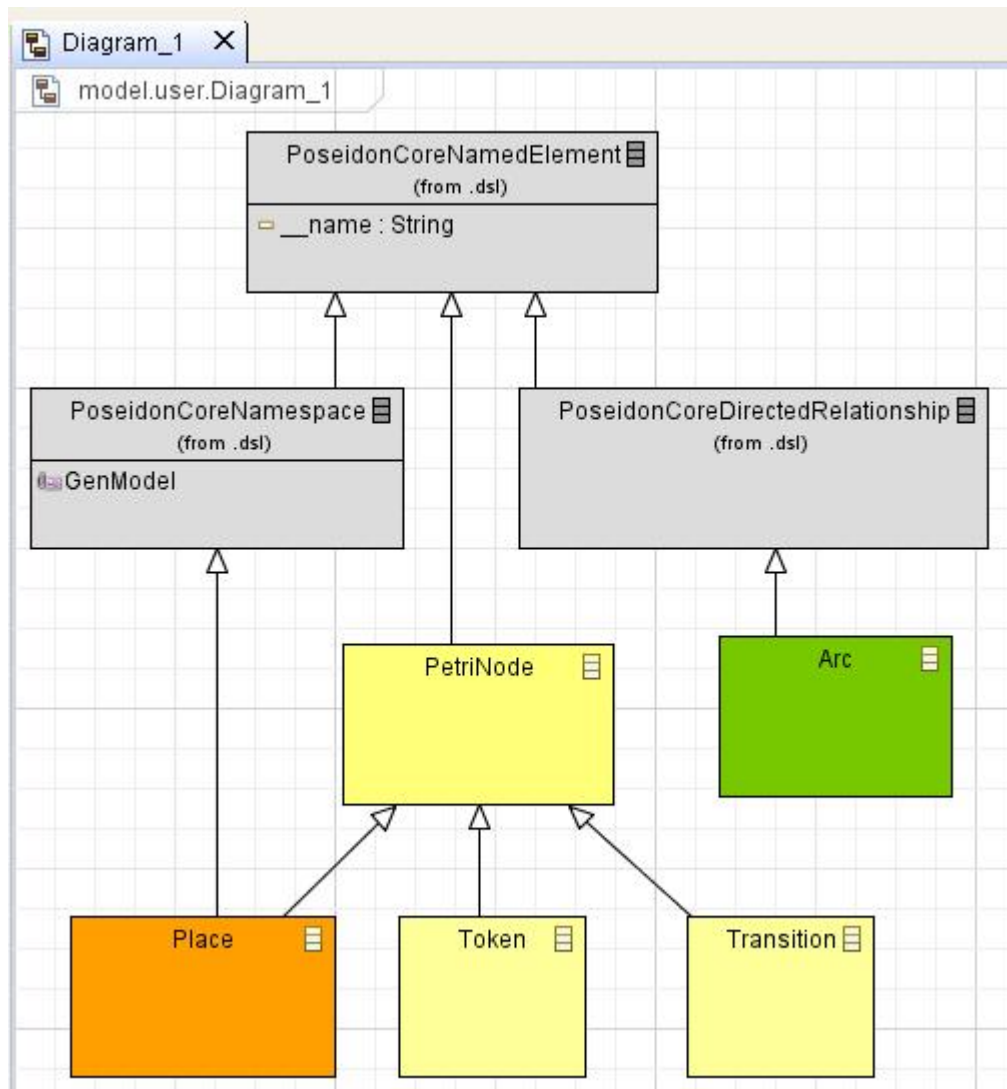
Those 3 are the basic elements of **Poseidon for DSLs** core metamodel. PoseidonCoreNamedElement represents a simple node in the future graphical editor, PoseidonCoreNamespace is used for the nodes which are capable of containing another nodes inside, and the PoseidonCoreDirectedRelationship represents an edge. In the palette you will now find a category called "**Poseidon Metamodel**".

For the Petri nets editor we need only several graphical elements :

1. Node for **Token**.
2. Node for **Transition**.
3. Edge for **Arc**.
4. Node for **Place**. This node should be capable of holding another nodes (**Tokens**) inside it, this is called a namespace.

So, let us create those metamodel elements in **Poseidon for Ecore**! First, let's create an element which will be a parent of all nodes in the metamodel. Use tool "**Poseidon Node**" to create a new element on the diagram. Call the new element "**PetriNode**". This element will be a parent of all nodes on the Petrin Nets diagram. Then create 2 Ecore Classes "**Token**" and "**Transition**" using tool "EClass". The next ecore class to create is "**Place**". Use tool "**Poseidon Namespace Node**" to create it. Now we want to inherit the "**Token**", "**Transition**" and "**Place**" from "**PetriNode**", so select the tool "Inheritance" from the palette and create 3 inheritances going from the "**Token**", "**Transition**" and "**Place**" nodes to the "**PetriNode**".

Next, use tool "**Poseidon Edge**" to create a new element called "**Arc**". In the Petri Net editor Arc will be an edge which connects nodes. The diagram should look like this now:



Poseidon for Ecore is a free tool, but you have to register before you can use any export features (such as export or save). So, in Poseidon for Ecore, click "Help -> License Manager", select the embedded serial key and click "Register". Provide your registration details in the pop-up dialog, click "Next".

Register your Gentleware Product!

User Information
 The license key will be issued for this user. Values in bold are required.

Salutation Mr

First Name Best

Last Name Customer

E-Mail best.customer@bestcustomers.com

Subscribe to Gentleware announcements (approx. one mail per month)

Preferred E-Mail format HTML plain text

Company Company Inc.

Country United Kingdom

See <http://www.gentleware.com> for Gentleware's privacy policy.

Previous Next Cancel

On the next screen, click "Finish":

Register your Gentleware Product!

Send Registration Data

Register Online: Gets your license key immediately from Gentleware's license server

Web Registration: Opens a web browser to get your license key.

You can also start your web browser manually and paste the registration data into the form at our web site to get your license key:
<http://www.gentleware.com?redirect=register>

Registration Data

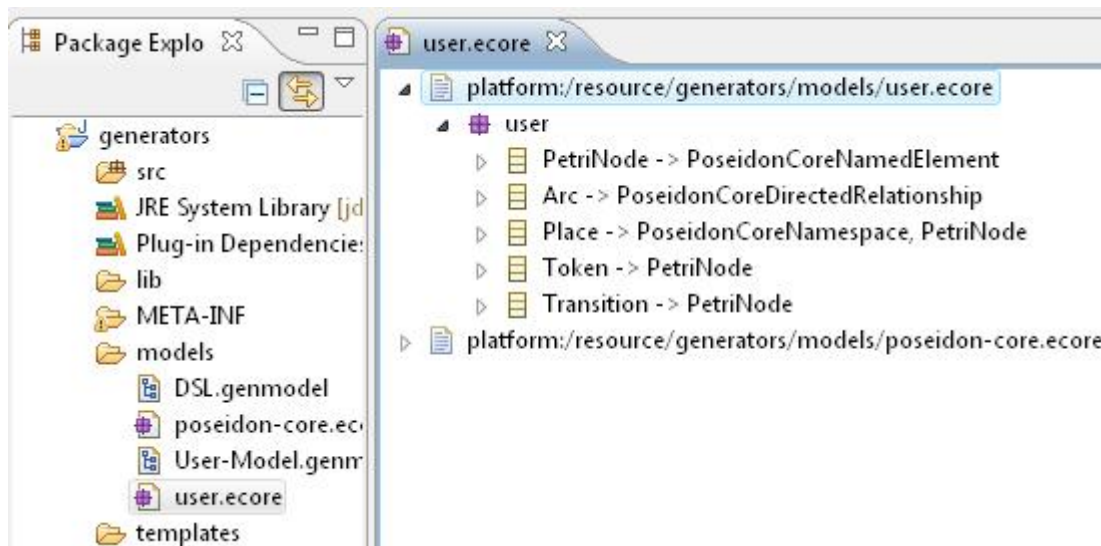
```
Um0gaXBsCvRWycHqyFyIIcu4nV21a1CIQP1IRdL1TGDQUppmWttI30csPTE
/1sQnq6HvX8+xHVF91Y4mXsCPX4Zun2Z00sAlGugvBBwzgbNEuszTfsg3DBP
ucxG1Rs1sXQB6bLQ8rIwB8/Sn0sgQKbJJzi8NHilz+K0sMVQRk+YF66uJ00Q
DS73x6G22eMI1EqziE7M7eSIP&u6MnPeKg1Q+dMvKfeYxnojkek7ajksTC2q
96xJ5S3hJrNrWQt2MgUDyYXe0Sjw/i2vby84DJNOV78mc+3t41VQIx9BUmY
```

Copy to Clipboard

Previous Finish Cancel

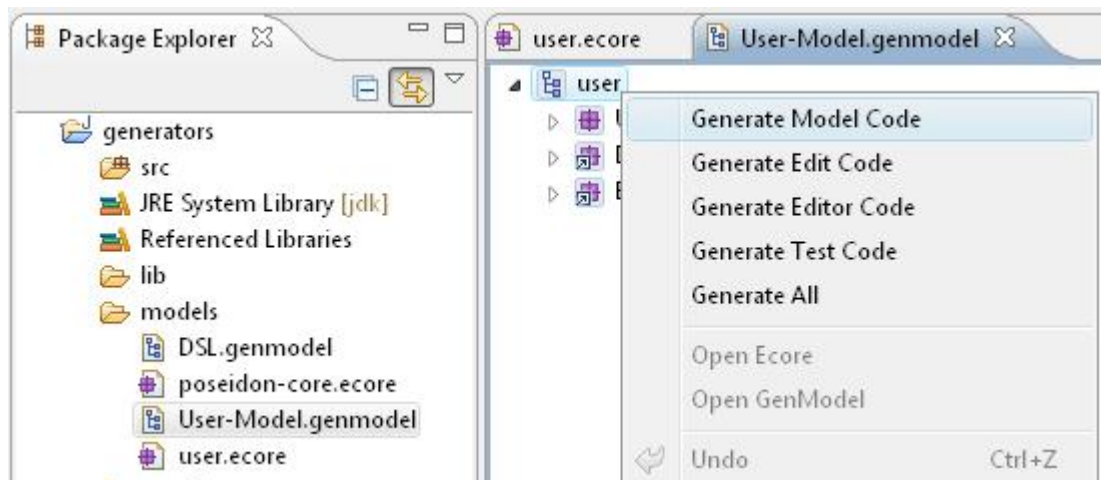


Ok, now you are ready to generate an ECore model. Go to main menu, select "Poseidon Metamodel->Generate Poseidon Ecore Metamodel". It generates an ecore model file into your workspace. Please go to the eclipse workspace, refresh the project **generators**, then open file user.ecore in the "**models**" folder of that project. It now contains the model which we just designed and generated in the **Poseidon For Ecore**. The **user.ecore** file should look like this:

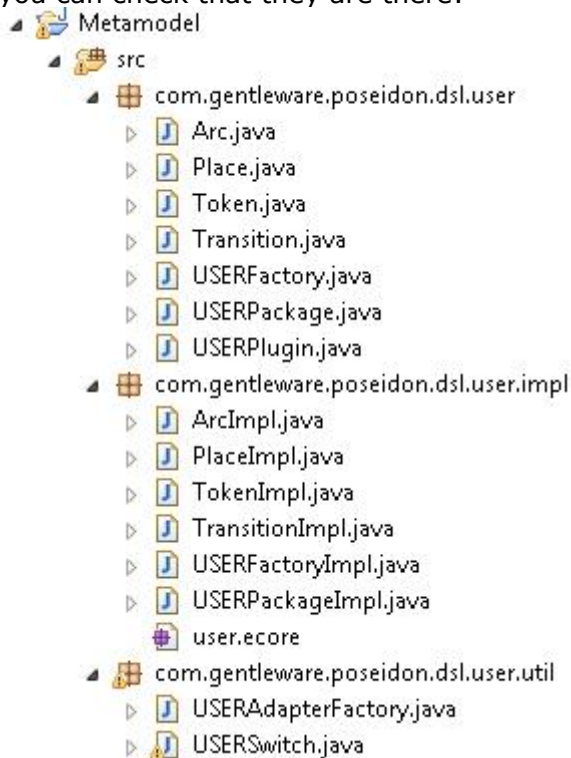


Generating code for the metamodel

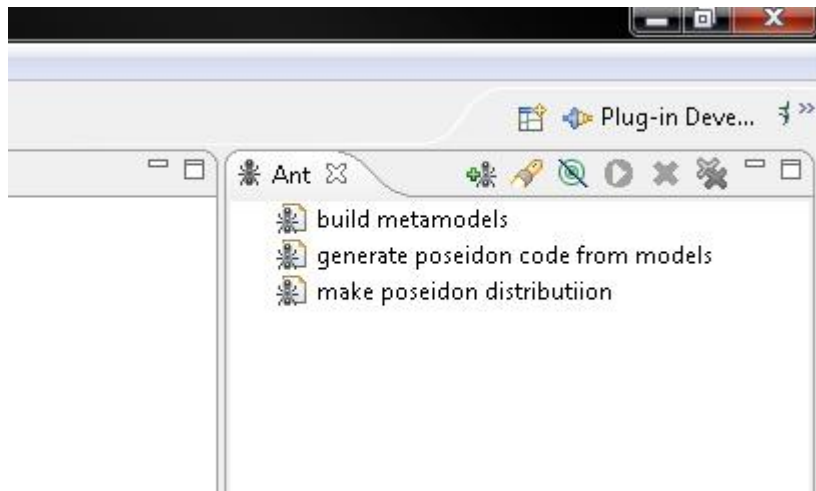
Now we are ready to generate java code for the metamodel. Open file **User-Model.genmodel** from the same **models** folder of the project **generators**. Right-click on the root element of the model and then click "**Generate Model Code**".



Wait several seconds while the code is being generated. The code is generated into project "**metamodel**". After the generation there will be several packages in **src** folder, you can check that they are there:



Then we use a Poseidon ant task to generate some additional stuff and make libraries (jar files) from the generated code to use them later. By default, the Ant view is already opened in the top-right corner of your workspace and all necessary Ant build files are added there for you.



If, for some reason, it is not there, then please open an Ant view in Eclipse (Window -> Show View -> Ant) then, in Package Explorer, browse to project **generators** and drag the file **build metamodels.xml** to the Ant view.

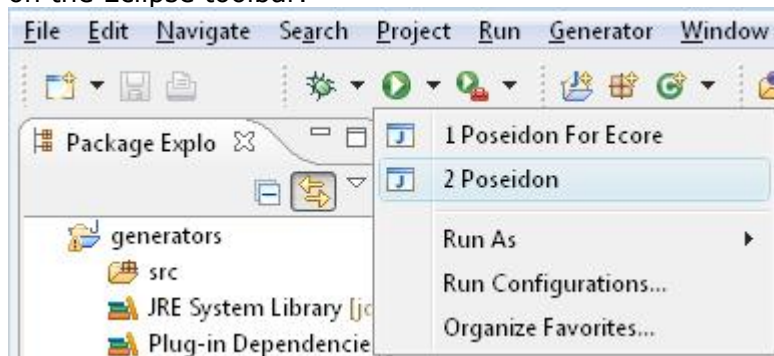
Now start this Ant build "build metamodels" from the Ant view by double-clicking on it or using the "Run" button.



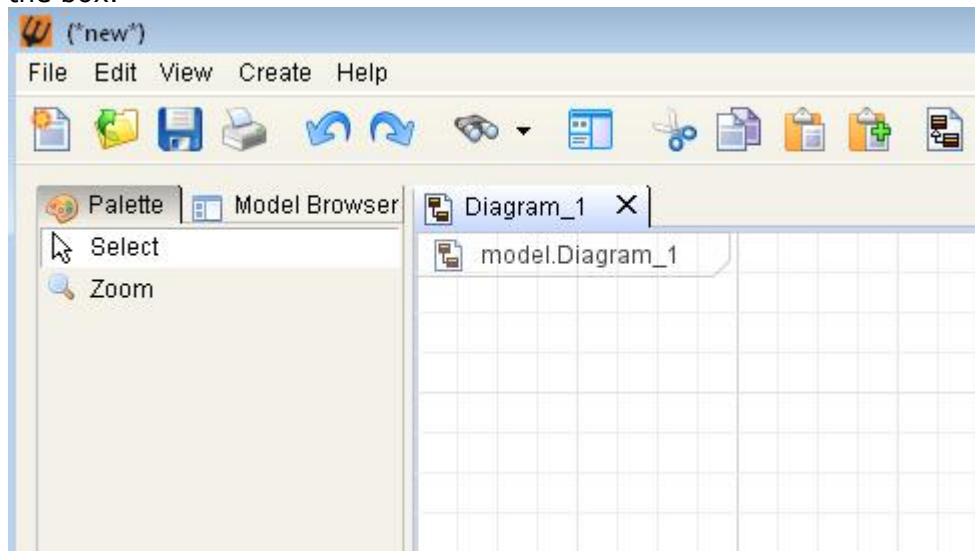
If the generation was successful (which you will see in the console) then our metamodel is ready to be used. If it did not work, check the version of your JDK. It works with Java 1.5 and jdk1.6.0_18 as external JDK. This can be changed under Window->Preferences->Java->Installed JREs. Let's move to the next step.

Generation of an empty editor

The template workspace we are working with does not contain the generated Poseidon editor code yet. We will use another ant build to generate the code. The Ant view contains the Ant task for generating the code. In the ant view it is called "**generate poseidon code from models**" (if it is not in the Ant view, please browse to project **generators** and drag the file **generate poseidon code from models.xml** from the package explorer to the Ant view). Start this Ant build. This will generate the code and the workspace will be compilable after that. Now you can start Poseidon. Use Run button on the Eclipse toolbar.



This will start Poseidon with an "empty" DSL, there are no elements on the Palette, so nothing can be created, but it already looks like a complete modeling environment out of the box.



Creation of all nodes and edges using default implementations and ignoring the look of elements

Now we can add a first element to our graphical modeling editor. Poseidon is generated from a set of textual DSLs. To add a new modeling element, we need to define it in two of these, the diagram and the tools model. Let's first look at the diagram model. You can find it in the project **Poseidon** under the folder **models** (**Poseidon/models/PoseidonDiagramModel.dgm**). Open the model file and add the following lines to it, inside the "diagram elements" space marked by comments.

```
node Place {
  metamodel_element: Place
  icon: "circle"
}

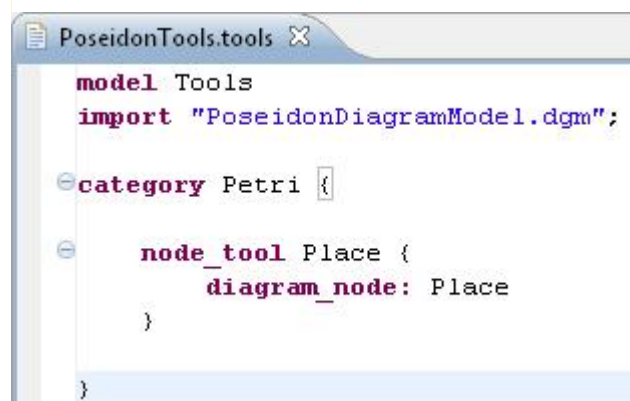
// diagram elements

node Place {
  metamodel_element: Place
  icon: "circle"
}

// end of diagram elements
```

Then add the following lines to the tools model (**Poseidon/models/PoseidonTools.tools**).

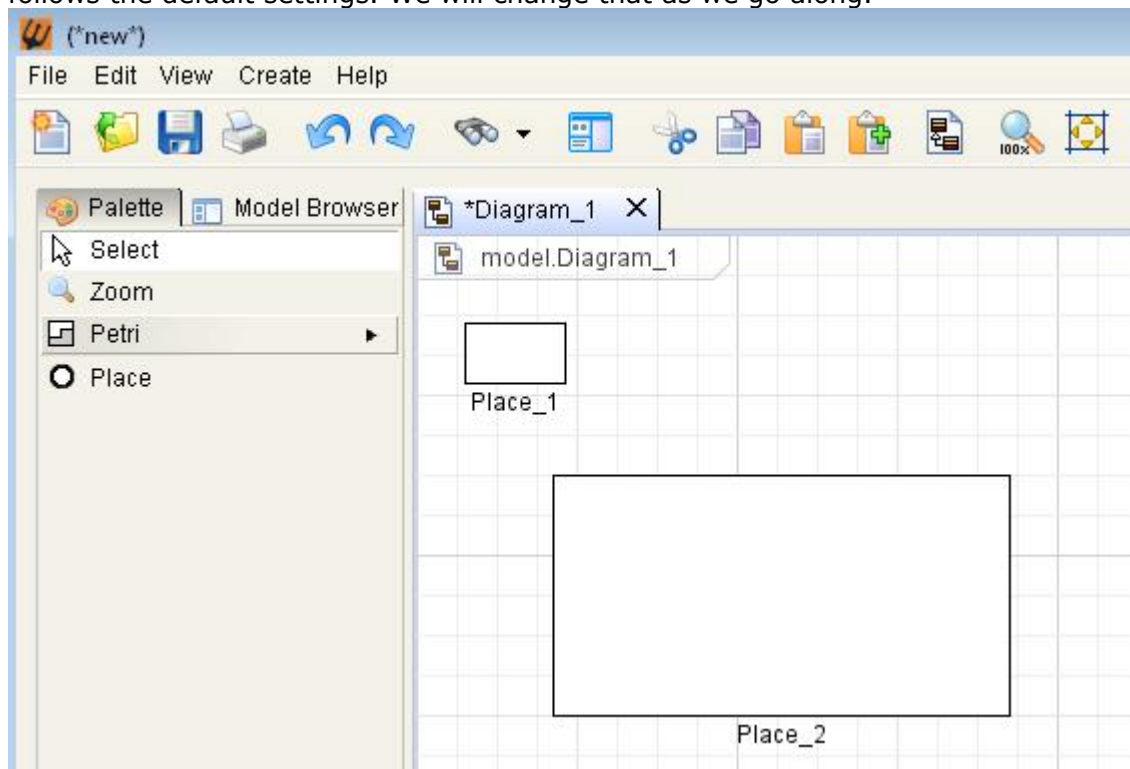
```
category Petri {
  node_tool Place {
    diagram_node: Place
  }
}
```



Now save the changed files and re-generate Poseidon code using the same Ant build in the Ant view ("**generate poseidon code from models**"). The framework will generate default implementations of the new diagram element and tools for it.

Now start Poseidon again to see the result!

In the palette you now see a category called "Petri" with a model element "Place". Go ahead and try using this model element in the diagram pane. You can already create model elements with these first initial steps. The shape is not as we want it yet, so far it follows the default settings. We will change that as we go along.



Now lets add a Token. In the diagram model, add these lines:

```
node Token {  
  metamodel_element: Token  
  icon: "start-state"  
}
```

In the tools model, add the following lines (inside Petri category):

```
node_tool Token {  
  diagram_node: Token  
}
```

Also, we want to make some changes to the Place, so that it can contain the Token. And while we are at it, lets also change the default and minimum size of it. Sizes of model elements are defined in terms of the grid size of the Poseidon editor. We want it rectangular and not too small, so 3 by 3 grid cells as default should work fine.

```
node Place {  
  metamodel_element: Place  
  icon: "circle"
```

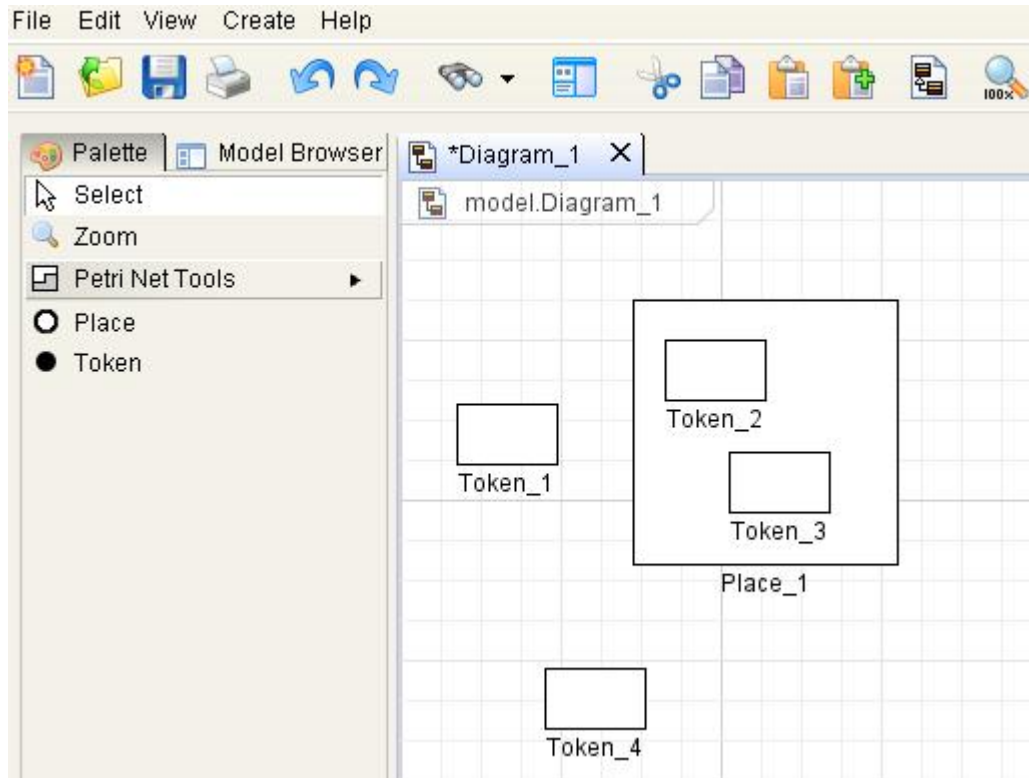
```

default_size: 4 * 4
minimum_size: 3 * 3
can_contain: Token
}

```

Poseidon also provides localization options for you. The "PetriGroup" is not a good name for the category in the palette, let's change that name in the property file. Go to project "**Poseidon**" and open file "**src/com/gentleware/poseidon/gui/palette/main/impl/DslGenMainPaletteResourceBundle.properties**". Add the following line there: PetriCategory = Petri Net Tools

Now we re-generate the code and start Poseidon again and to see our changes.



Now, lets add the other model elements, Transition and Arc. First we add following lines to the diagram model.

```

node Transition {
  metamodel_element: Transition
  icon: "join-node-horizontal"
}
edge Arc {
  sources: Place Transition
  targets: Place Transition
  icon: "arrow-right-solid"
  metamodel_element: Arc
}

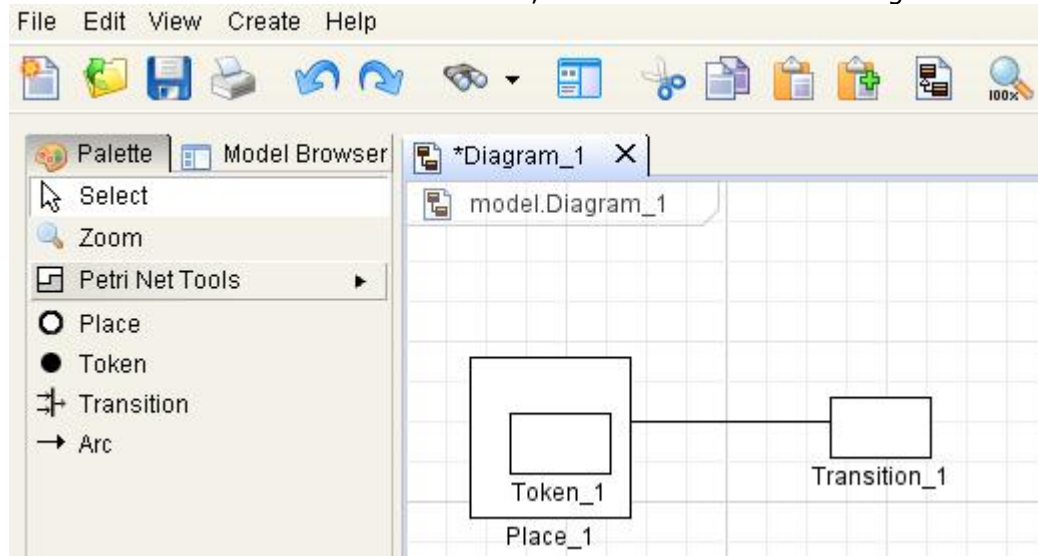
```

And then these lines to the tools model:

```
node_tool Transition {  
  diagram_node: Transition  
}  
  
edge_tool Arc {  
  diagram_edge: Arc  
}
```

Then re-generate code and start Poseidon.

As a result of the changes we have done to the models, you can now create places and transitions and connect them with arcs, as shown in the next image.



We are done with the first part of the tutorial, we have created all necessary elements. Now, we will work on their appearance.

Making elements look like they have to look

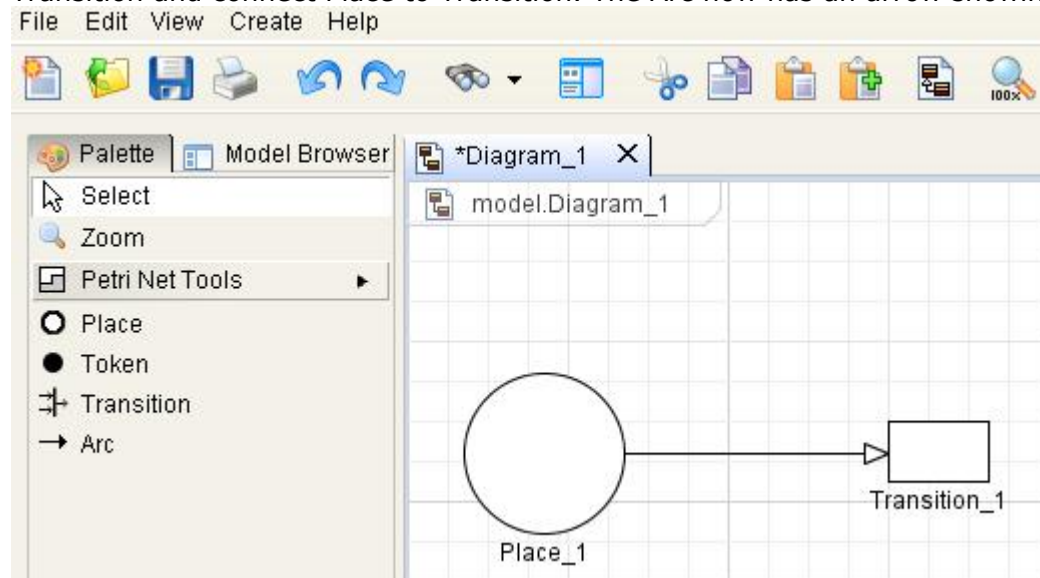
First of all, let us make Place look like a circle. We will add attribute "shape" to Place definition in the diagram model

```
node Place {  
  metamodel_element: Place  
  icon: "circle"  
  default_size: 4 * 4  
  minimum_size: 3 * 3  
  can_contain: Token  
  shape: ELLIPSE  
}
```

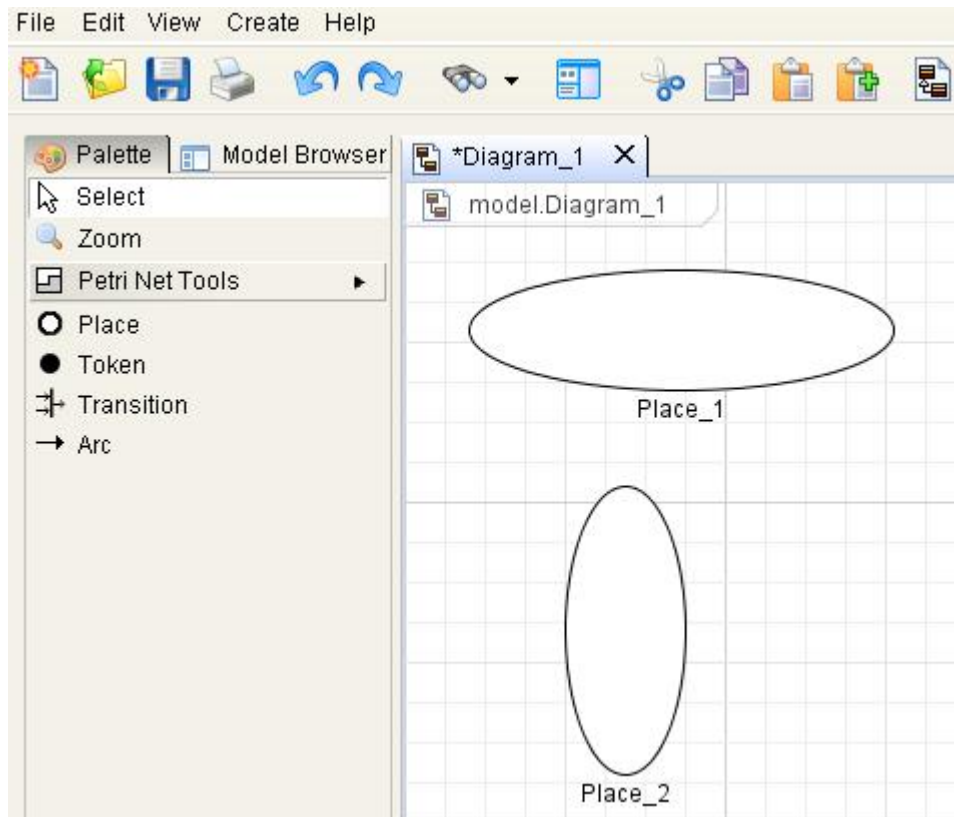
We also want the Arc to end with an arrow, so that we know the direction of the Arc. To do that, we will "tell" the Arc to draw an arrow at the target end:

```
edge Arc {  
  sources: Place Transition  
  targets: Place Transition  
  icon: "arrow-right-solid"  
  metamodel_element: Arc  
  at_target_draw: CLOSED_ARROW  
}
```

Re-generate code, start Poseidon and create a Place. It is now a circle. Create a Transition and connect Place to Transition. The Arc now has an arrow showing direction.



However, if you change the size of the Place by dragging it, it doesn't keep the proportions, so the circle actually becomes an ellipse and the Place looks not like it should look according to the notation. Like this, for example:



Poseidon diagram model allows you to choose if you wish the element to keep its' proportions. Use attribute "keep_proportions" in the definition of the Place

```
node Place {
  metamodel_element: Place
  icon: "circle"
  default_size: 4 * 4
  minimum_size: 3 * 3
  can_contain: Token
  shape: ELLIPSE
  keep_proportions: true
}
```

Re-generate the code and start poseidon. Now it will not allow you to change the proportions of the node on the diagram, so it will always be a circle.

Ok, now let's make Token look like it should. We want Token to be a small black circle. And we don't want the name of the Token on the diagram since we don't really need it. So, let's go to diagram model and make some changes. Add these lines to node Token:

```
default_size: 1 * 1
minimum_size: 1 * 1
shape: ELLIPSE
name_position: NO_NAME
keep_proportions: true
property FillColor: BLACK
```

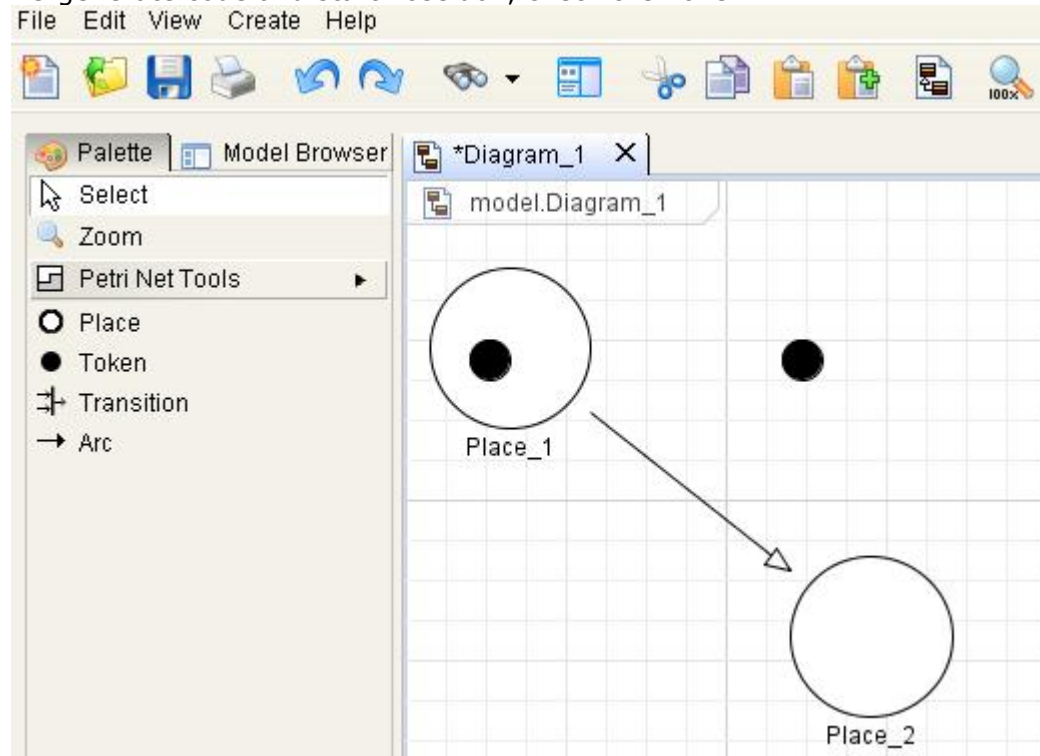
Now the node Token description should look like this:

```

node Token {
  metamodel_element: Token
  icon: "start-state"
  default_size: 1 * 1
  minimum_size: 1 * 1
  shape: ELLIPSE
  name_position: NO_NAME
  keep_proportions: true
  property FillColor: BLACK
}

```

Re-generate code and start Poseidon, check the Token.



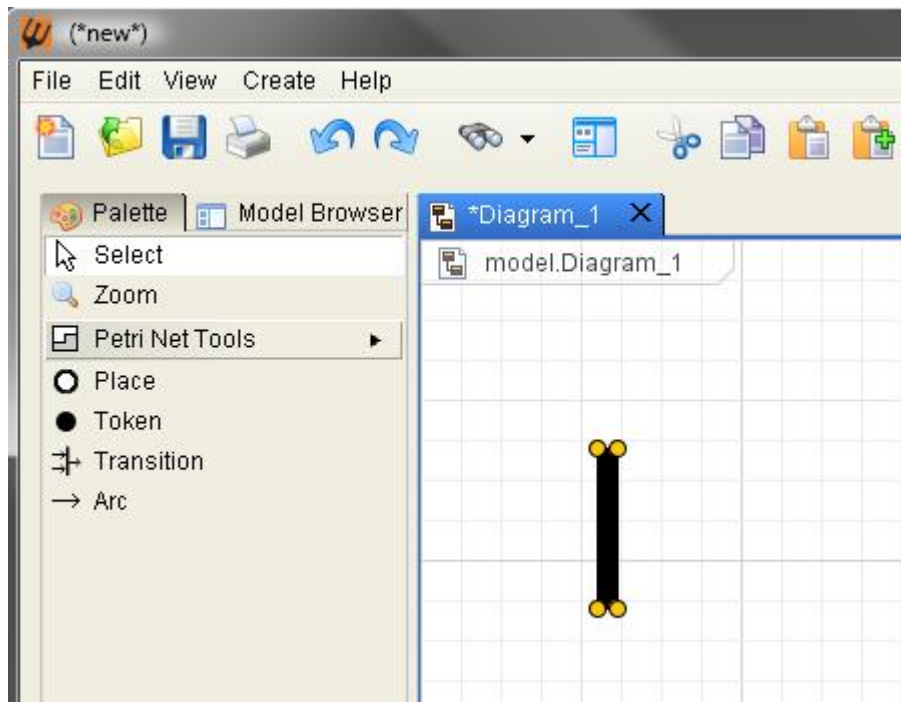
So, now Place and Token look like they have to, but Transition is still just a default rectangle. Transition should be black and the rectangle dimensions should be different. Let us change that by adding some lines to Transition node in diagram model

```

node Transition {
  metamodel_element: Transition
  icon: "join-node-horizontal"
  default_size: 0.5 * 4
  minimum_size: 0.3 * 0.3
  shape: RECTANGLE
  name_position: NO_NAME
  property FillColor: BLACK
}

```

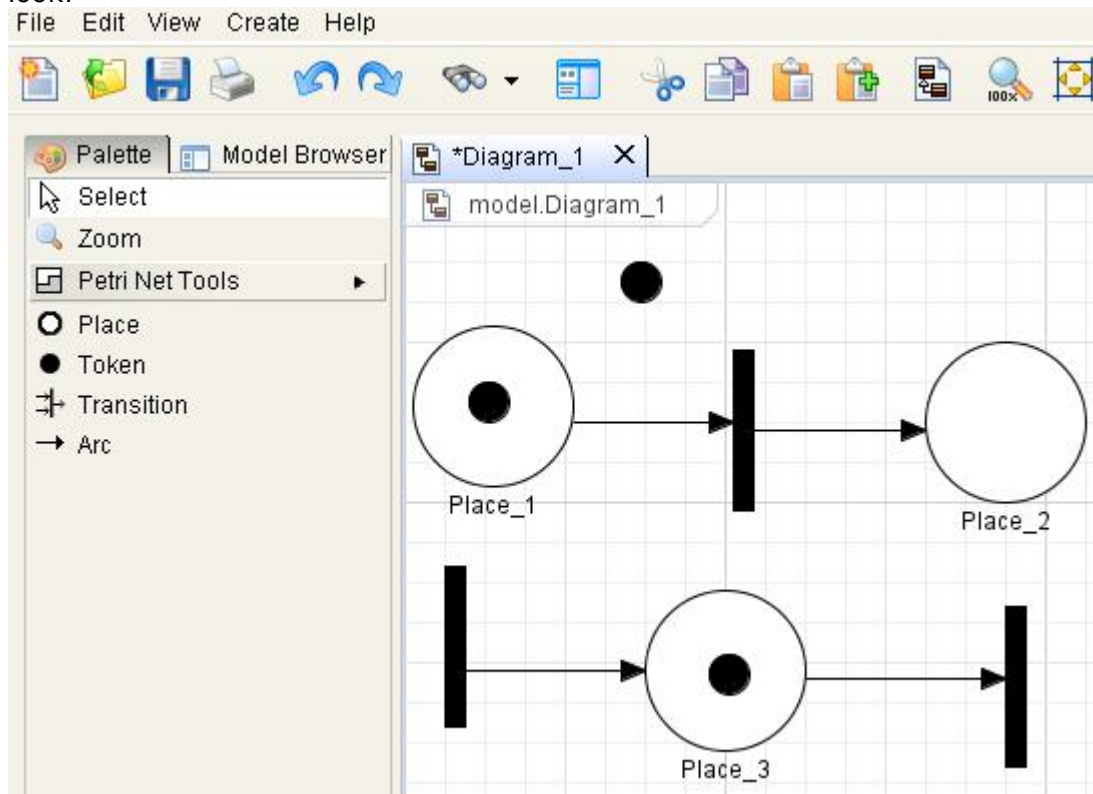
Again, re-generate code and start Poseidon to check the effect:



Now let us make one last touch to make our Petri Net editor look like it should. Go to diagram model and set the fill color of Arc to black:

property FillColor: BLACK

Now re-generate code and start Poseidon. The editor now looks like we wanted it to look!



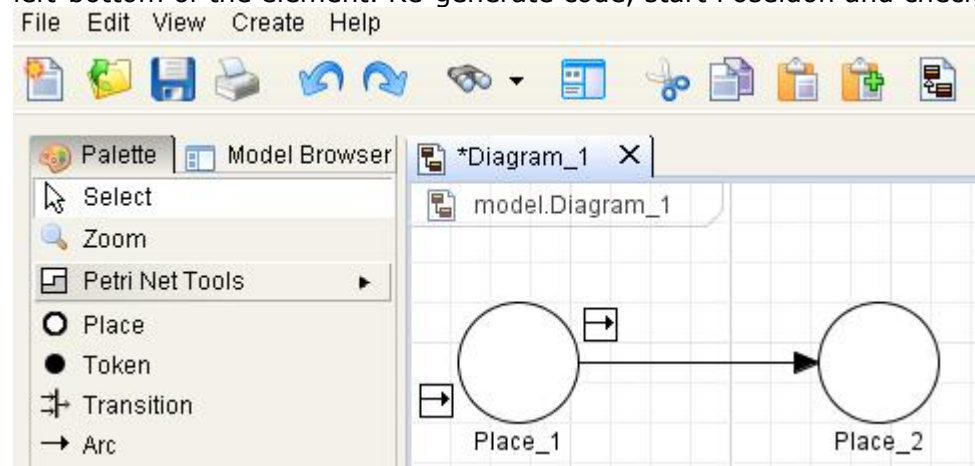
Adding rapid buttons

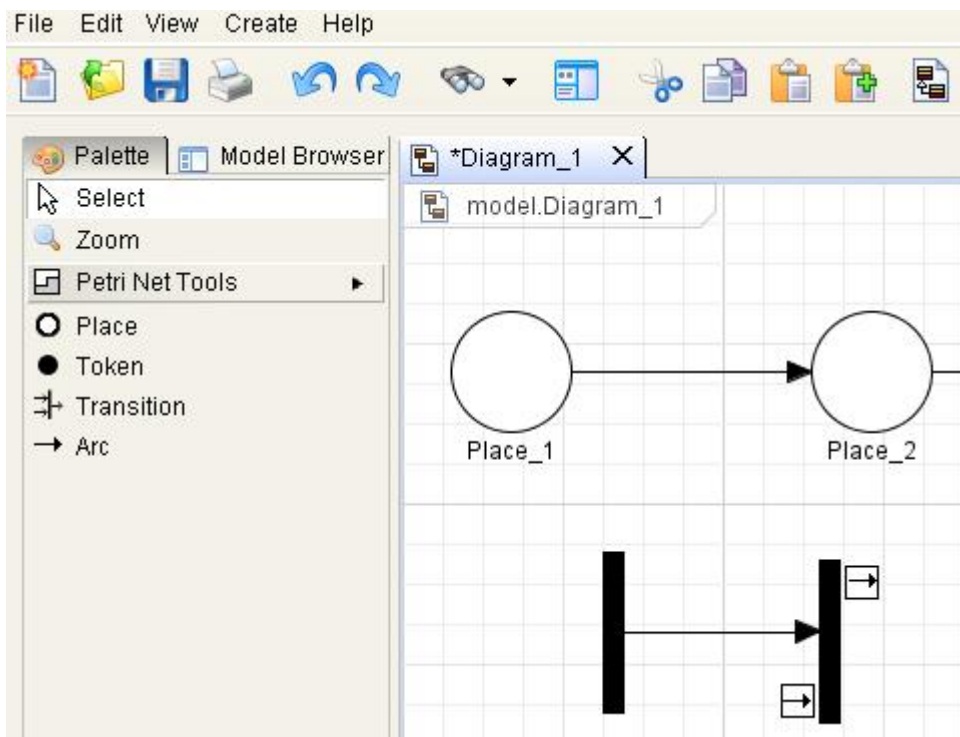
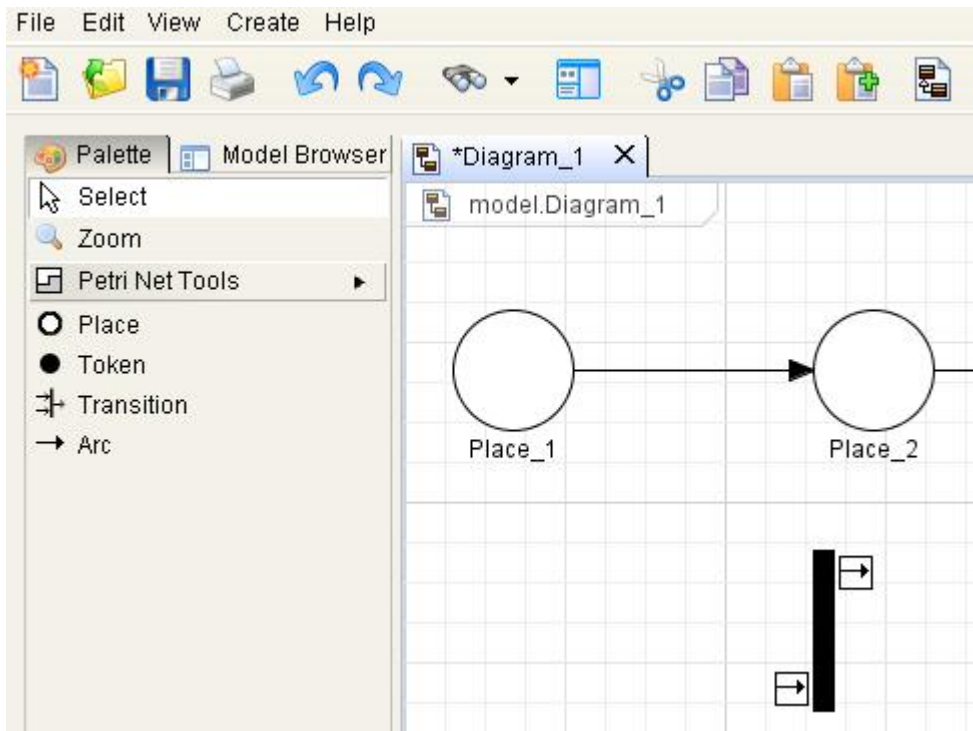
Rapid buttons are a neat little mechanism that make editing diagrams faster and more intuitive. Rapid buttons pop up in the diagram next to a node and provide the tools to create edges in the right context. Rapid buttons can appear in 8 different positions around a node, with names like RIGHT_TOP or RIGHT_BOTTOM.

Adding rapid buttons to a diagram element is very simple. This is configured in a separate model file. Open rapid buttons model (**Poseidon/models/RButtons.rbtn**). Add the following lines there:

```
button Arc {  
  anchors: Place Transition  
  edge: Arc  
  position: RIGHT_TOP LEFT_BOTTOM  
  icon: "arrow-right-solid"  
}
```

This will add rapid buttons which create Arc (edge: Arc) to Place and Transition diagram elements (anchors: Place Transition). Rapid buttons should appear on the right-top and left-bottom of the element. Re-generate code, start Poseidon and check how it works.



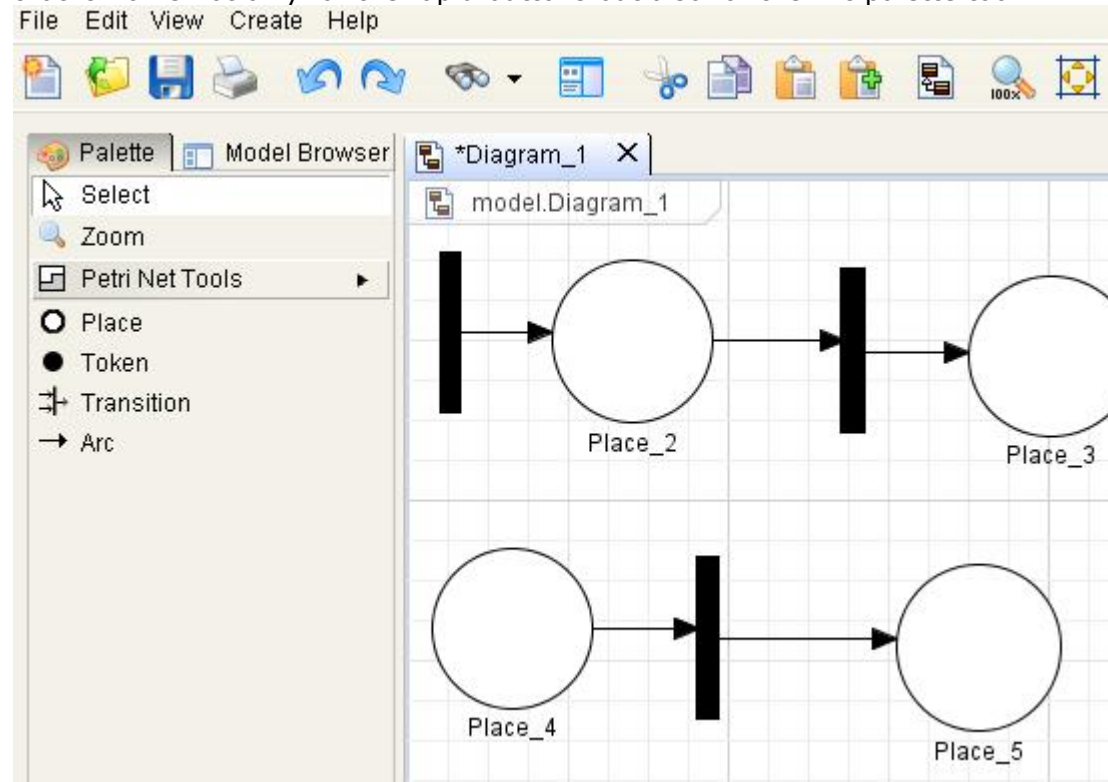


Adding edge rules

Petri net notation assumes that Arcs go from Place to Transition or from a Transition to a Place, but not from a Place to Place or from a Transition to Transition. So the default behavior of the rapid buttons we created is not as wanted. Let us change that. This is controlled by another Poseidon model file. Open **Poseidon/models/EdgeRules.erule** and add this rule in the model:

```
rules_for Arc {  
  from Place create Transition  
  from Transition create Place  
}
```

Re-generate code and start Poseidon. Now the Arc is created correctly by default! Note that it works not only for the rapid buttons but also for the Arc palette tool.



Customizing Add menu

Poseidon has another nice feature to increase productivity. In the context menu of the diagram, there is a menu item called "add" that allows to add new model elements in the given context. But there is a short cut for this. Just hit the space bar and the add context menu will appear in the position of the mouse. If you invoke it in the empty space of a diagram, it will allow you to create elements in that position. By default the add menu is empty. To add new elements to this add menu, we have to add the attribute "shown_in_add_menu_of" to the tools model.

So, let us add this entry in the right location. Place and Transition should be in the add menu of the diagram, so go to the definition of Place and Transition and add the line

```
node_tool Transition {
  diagram_node: Transition
  shown_in_add_menu_of: Diagram
}

node_tool Place {
  diagram_node: Place
  shown_in_add_menu_of: Diagram
}
```

After the code is re-generated, the add menu for the diagram will contain Place and Transition.

Next, we want the Token to be available only in the add menu of a Place. For this we need a little preparation in the settings of Place, add the following:

```
node_tool Place {
  diagram_node: Place
  shown_in_add_menu_of: Diagram
  add_menu_role: "place"
}
```

Now change the line for Token to

```
node_tool Token {
  diagram_node: Token
  shown_in_add_menu_of: "place"
}
```

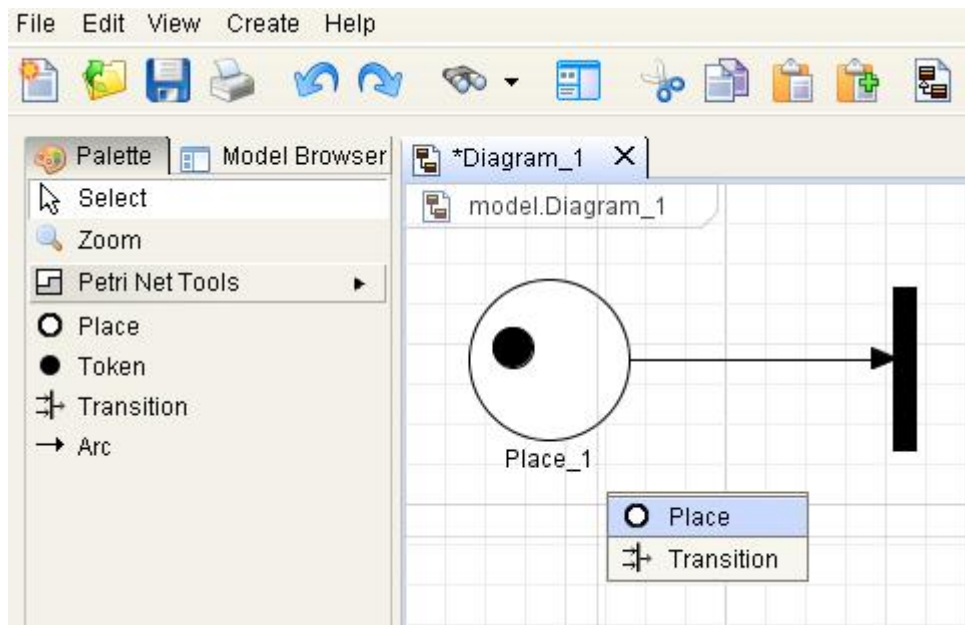
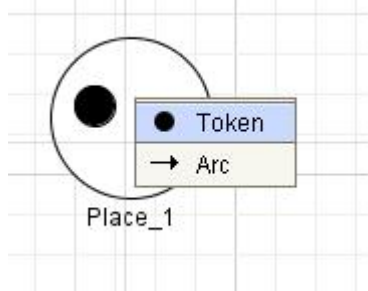
The Arc creation tool should not be available on the diagram directly, but only for existing elements like Place and Transition. So, go to Arc definition in the tools model and add the line

```
edge_tool Arc {
  diagram_edge: Arc
  shown_in_add_menu_of: element "place"
}
```

The first item in the list of "shown_in_add_menu_of" values is **element** which means that the Arc tool will be shown for all nodes by default. The second value, "place" means

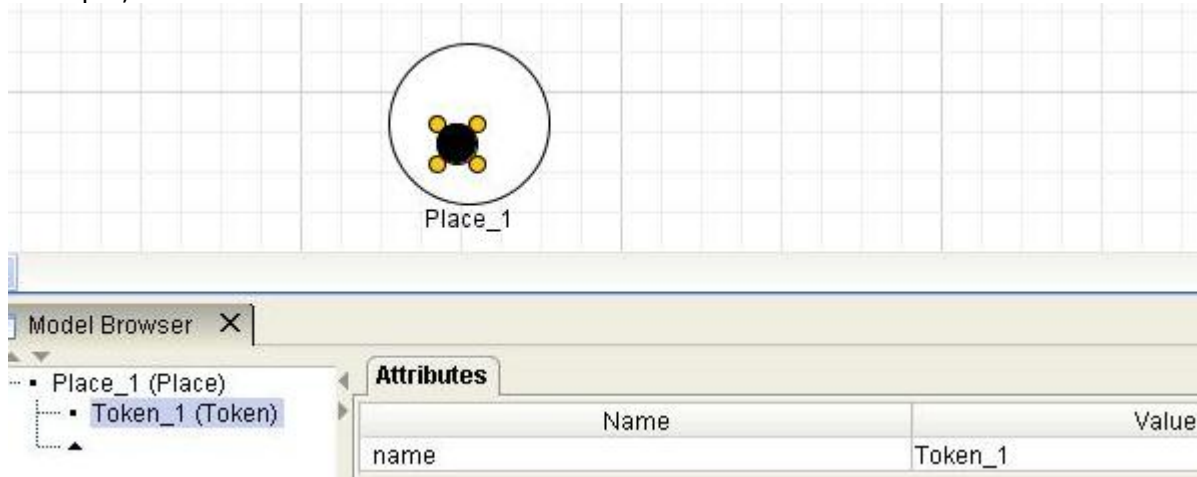
that it will be also shown for element **Place** which now has an **add_menu_role** "place."

Re-generate the code and start Poseidon to see the effect.



Customizing Attributes tab in Model browser (Properties model)

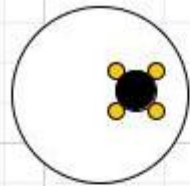
Let us take a look at Attributes tab of the Model Browser. By default, almost all attributes are shown there, but you can change the default behavior if you want to. For example, a token has a name and it is visible in the attributes tab:



On the other hand, the name of the token doesn't make sense in Petri nets, so we do not really want to show it. Open the Properties model (**Poseidon/models/Properties.prt**) and add the following definition there:

```
attribute {  
  name: "__name"  
  hidden_elements: Token  
}
```

This definition states that property "name" should be hidden for Token. Re-generate code, start Poseidon and check - the name should not be in the Attributes tab for Token any more:



Place_1

Model Browser X

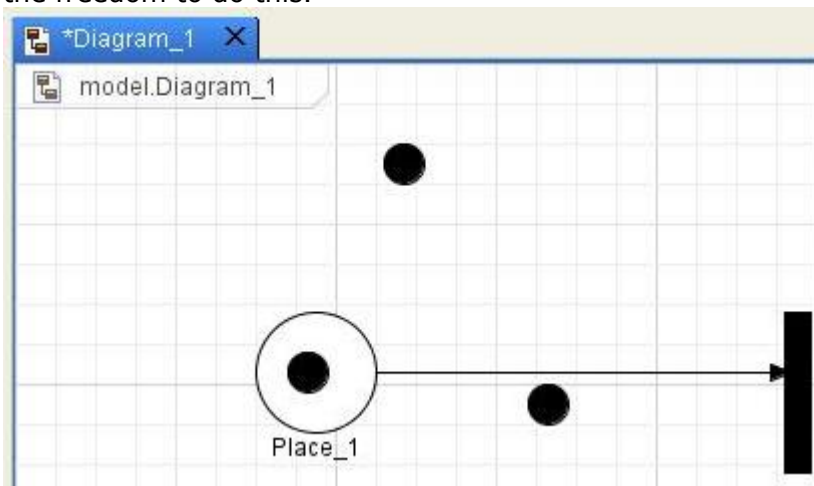
- Place_1 (Place)
 - Token_1 (Token)

Attributes

Name	Value
------	-------

Customizing the code

The editor is already quite nice. There are just a few details we would like to adjust. For example, Token should only be allowed inside a Place and not somewhere in the open space of the diagram. For this we would like to show how to override the generated code with custom code. You will not have to do this very often, but the architecture gives you the freedom to do this.



This is the first point of this tutorial where you have to add some code. The appearance and other aspects of an element on the diagram is handled by classes that have the ending Gem, as in Gemstone. Those Gem classes are generated for you when you create new elements in Poseidon, but then you can customize them according to your needs. We need to change something for the Token element, so please open class **TokenNodeGem** (project **Poseidon**, package **com.gentleware.poseidon.custom.diagrams.node.impl**). This class was generated the first time you added element Token to the diagram model, but it is NOT overwritten each time you regenerate, so you can change the code. As you see, **TokenNodeGem** is inherited from class **DslGenTokenNodeGem**. This **DslGenTokenNodeGem** IS regenerated each time you generate code, so you can't change it. Now please go to class **DslGenTokenNodeGem** and copy method **createBasicNodeAppearanceFacet()** from there. We are going to override this method. Paste it into **TokenNodeGem** class.

```
public NodeAppearanceFacet createBasicNodeAppearanceFacet() {
    return new BasicNoNameNodeAppearanceFacet(initialFillColor, initialFillColor, figureFacet,
figureName, texttableFacet, subject, true) {
        public ShapeAppearanceFacet createShapeAppearanceFacet() {
            return new EllipseShapeAppearanceFacet();
        }
    };
}
```

This method creates an Appearance facet for the Token. We now come to the notion of Appearance facets. Each **Gem** class has a number of facets. Each of the facets is responsible for some aspect of the diagram element's behavior. For example, we are now going to change the behavior of Token's Appearance facet, which is responsible for

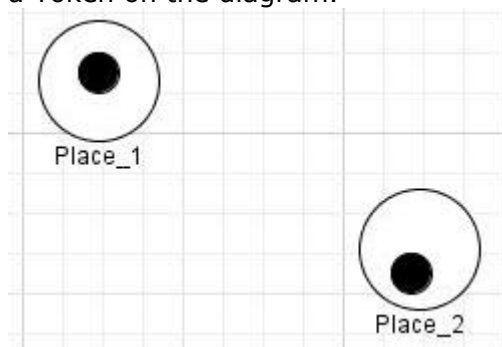
drawing and displaying the Token on the diagram. So, we took the default generated code which creates the Token's Appearance facet from the generated non-customizable parent class **DslGenTokenNodeGem** and we are going to modify that code in the generated, but customizable class **TokenNodeGem**. Now let us add some new lines to the method **createBasicNodeAppearanceFacet** in the **TokenNodeGem** class:

```

public NodeAppearanceFacet createBasicNodeAppearanceFacet() {
    return new BasicNoNameNodeAppearanceFacet(initialFillColor, initialFillColor, figureFacet,
figureName, texttableFacet, subject, true) {
        public ShapeAppearanceFacet createShapeAppearanceFacet() {
            return new EllipseShapeAppearanceFacet();
        }
        public boolean acceptsContainer(ContainerFacet container) {
            if (container == null) { //means that the container is diagram
                return false;
            }
            return super.acceptsContainer(container);
        }
    };
}

```

Save the changes we have done to the **TokenNodeGem** class and start Poseidon. Now try to create Tokens - you can now only create Token inside a Place, you cannot create a Token on the diagram.



Please note also that we have written some custom code which overrides some generated code, so now this custom code is in control of the appearance of the Token on the diagram. This means, for example, that now if you change the shape of the Token in the diagram model to RECTANGLE and regenerate the code - it will not have an effect, the shape will still be a circle because it is defined in the overridden custom code you have written. So now if you want to change the shape of the Token to a rectangle, for example, you have to open **TokenNodeGem** class and change one line there from

```
return new EllipseShapeAppearanceFacet();
```

to

```
return new RectangleShapeAppearanceFacet();
```

Now the method **createBasicNodeAppearanceFacet** should look like this:

```

public NodeAppearanceFacet createBasicNodeAppearanceFacet() {
    return new BasicNoNameNodeAppearanceFacet(initialFillColor, initialFillColor, figureFacet,
figureName, texttableFacet, subject, true) {
        public ShapeAppearanceFacet createShapeAppearanceFacet() {

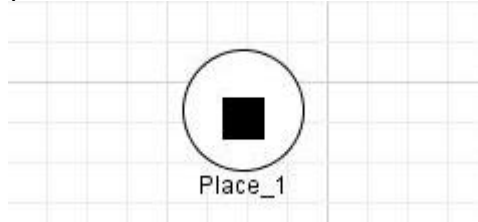
```

```

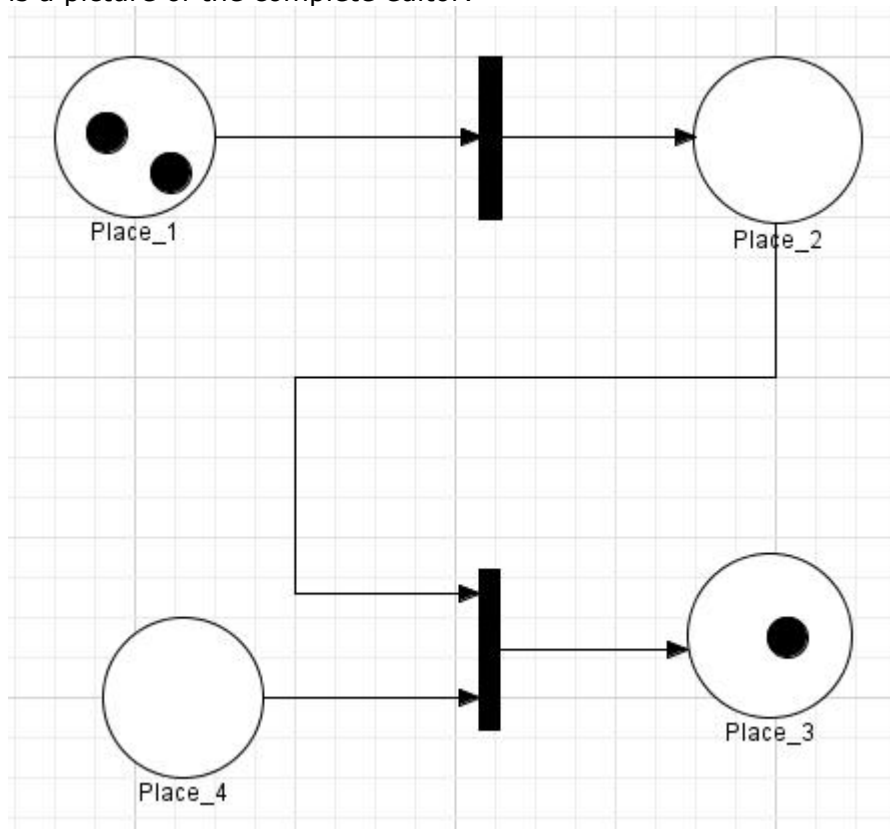
return new RectangleShapeAppearanceFacet();
}
public boolean acceptsContainer(ContainerFacet container) {
    if (container == null) { //means that the container is diagram
        return false;
    }
    return super.acceptsContainer(container);
}
};
}
}

```

Now you can start Poseidon and check that Token is drawn as a rectangle. If that is what you wanted. Otherwise undo the last step.

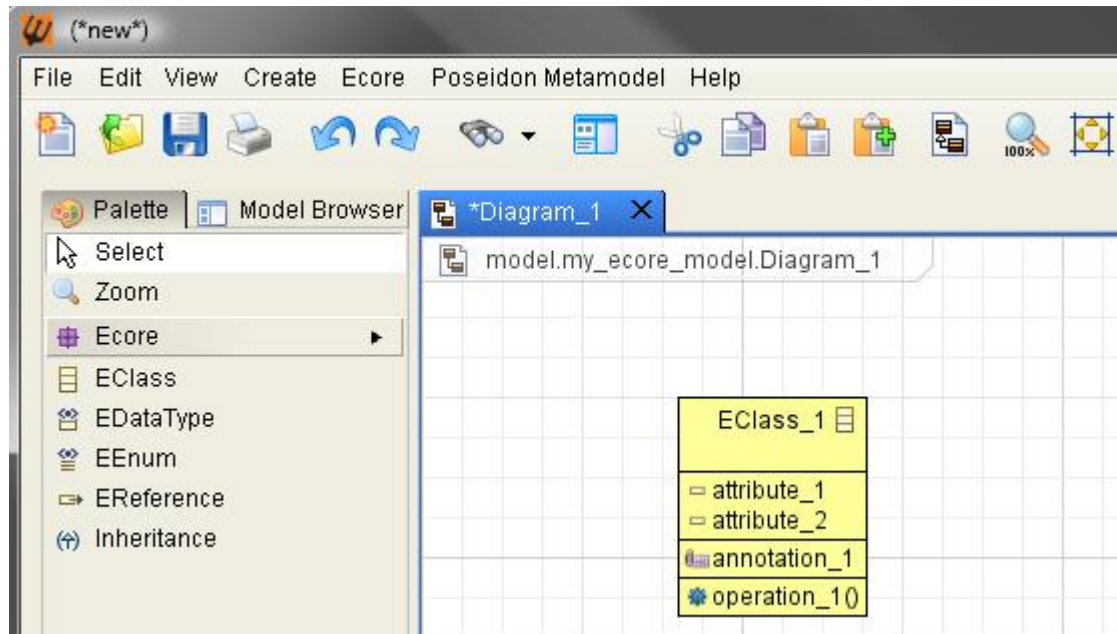


Now the DSL is complete. Models are saved to EMF and can readily be loaded into Eclipse and processed in model transformation tools such as openArchitectureWare. Here is a picture of the complete editor:

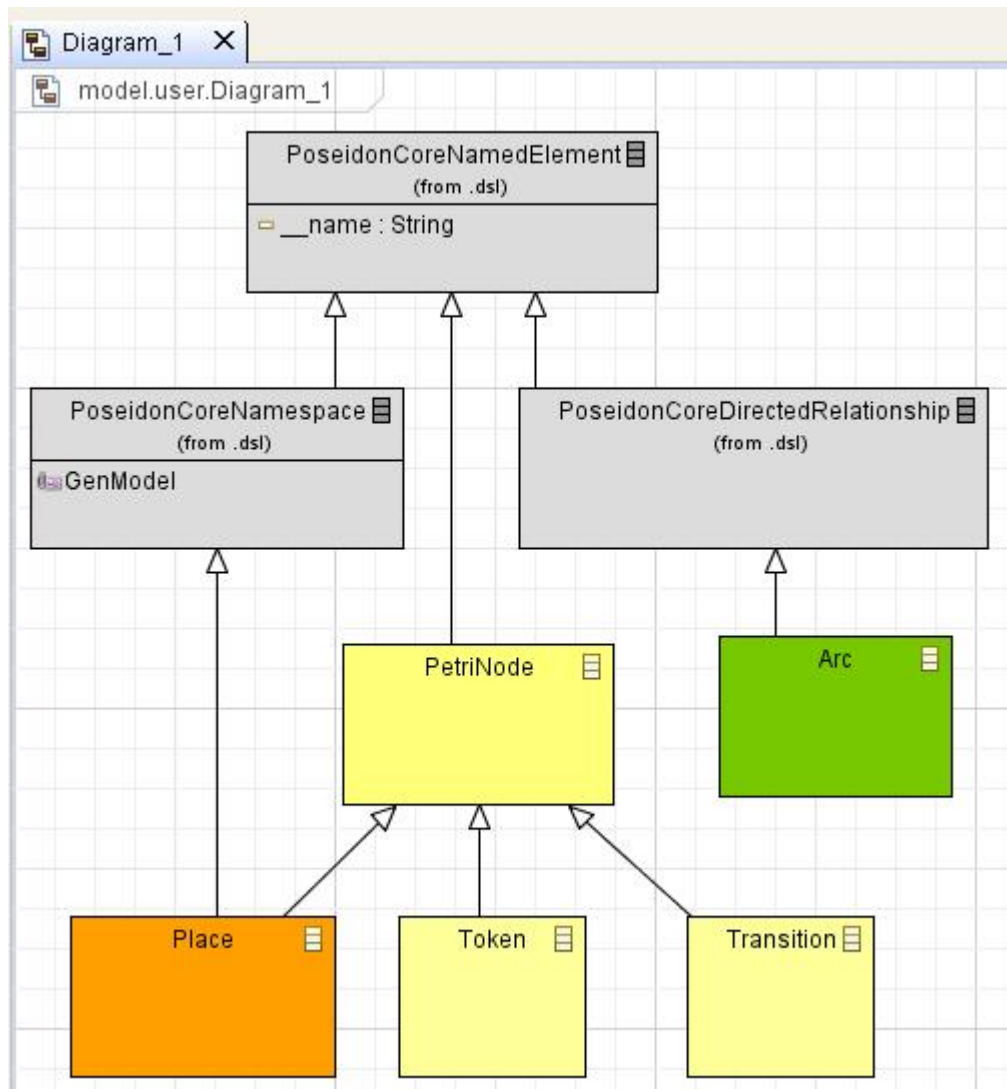


Advanced: Creating nodes with compartments

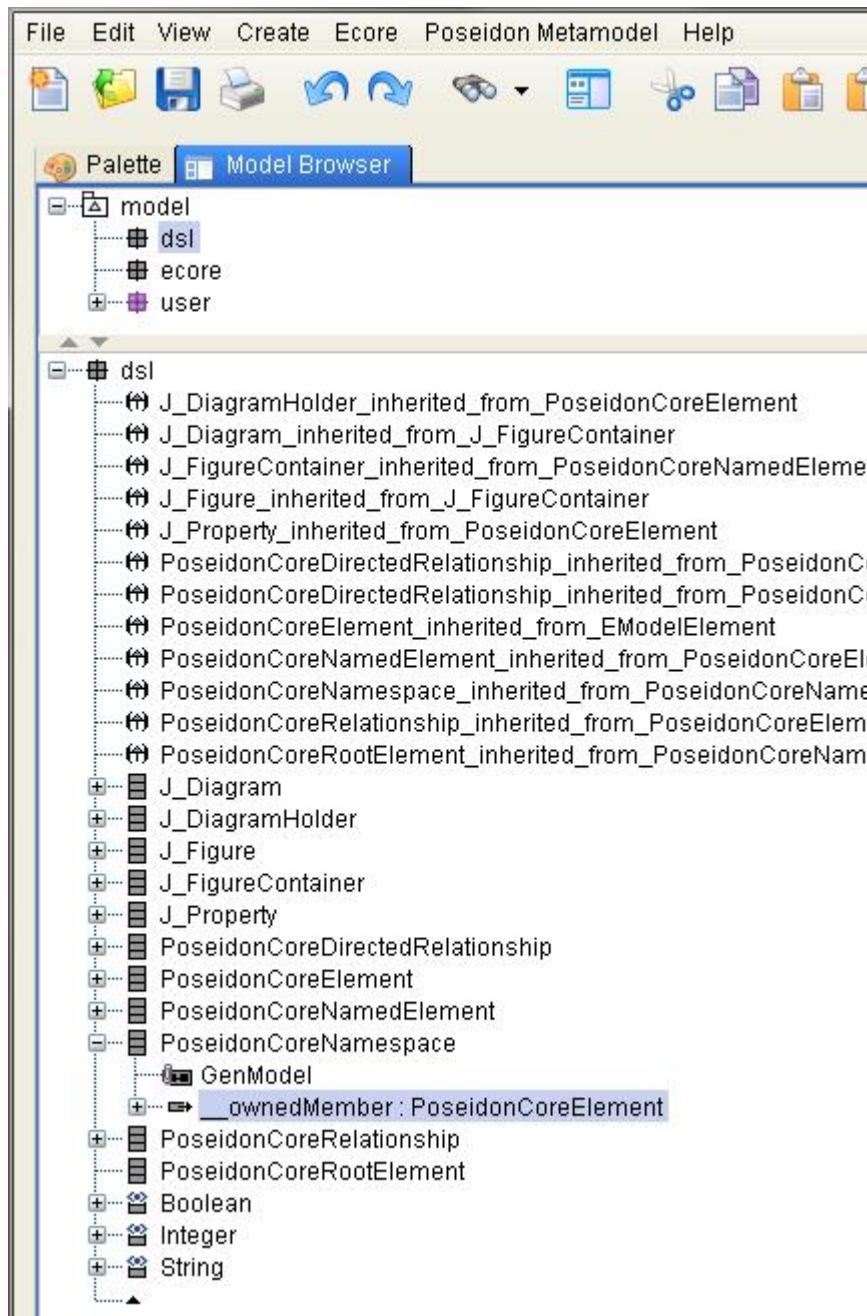
Start "**Poseidon for Ecore**" from your workspace. Create an EClass on the diagram. You can add attributes, operations and annotations to any EClass (hit space when mouse pointer is over the EClass on the diagram, or right-click the mouse and use "Add" menu in context menu).



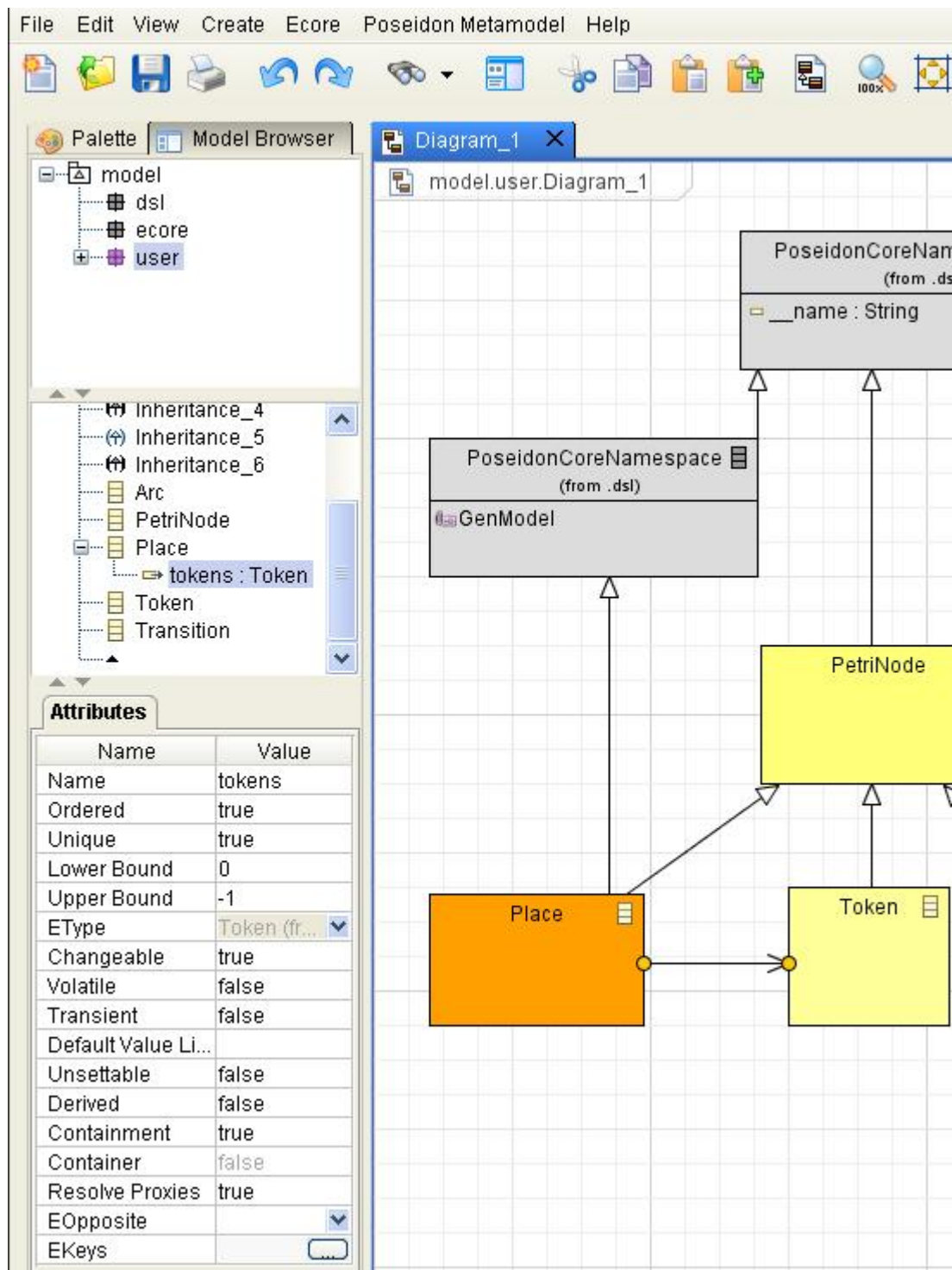
So, node for the EClass has 3 compartments for elements which are contained in it: attribute compartment, operation compartment and annotation compartment. The "compartment" feature is a part of the framework, so we can do the same thing for our Poseidon for PetriNets editor. Let us make another representation option for the Place. Let it be a rectangle, which contains a list of tokens represented by token names inside the Place node (same as attributes inside EClass node). We have to start our changes from the metamodel. Start "**Poseidon From Ecore**" from your workbench and open your metamodel project. So far it looks like this:



So far, when you added a new Token in a Place - it was stored in a default list where Poseidon namespaces store its' members. By default it is a list defined by a reference "**__ownedMember**" of EClass "**PoseidonCoreNamespace**" (which Place is inherited from). You can find it in the ecore package "dsl":



Now, if we want to have a Token compartment for Place node, we have to define our own list where Tokens will be stored. Create a new reference from Place to Token. Call it "tokens". Set the "Upper Bound" attribute to -1. Set "Containment" attribute to true.



Now we are going to repeat the steps which we did the first time we created the metamodel:

1. Generate ecore model file from **Poseidon For Ecore**. To do that, select Poseidon Metamodel-> Generate Poseidon Ecore Metamodel
2. In the Eclipse workspace, refresh project "**generators**", then check file "user.ecore" in "**models**" folder of this project. The Place EClass should now have a reference "tokens".

3. Generate java code for the metamodel. To do that, open file "User-Model.genmodel" which is next to "**user.ecore**", right-click on the root element and select "**Generate Model Code**".
4. Run Ant build "build metamodels" to build and replace metamodel libraries and some other stuff.

Now we are ready to define a new node for a Place, which will have compartments for Tokens. Open diagram model and add the following definition to it:

```
node PlaceWithCompartments{
  metamodel_element: Place
  minimum_size: 3 * 3
  compartments: Token
}
```

That is a definition of a new node for Place. This node has compartment for Tokens. To make it work, we also have to update the definition of Token by adding the following lines:

```
metamodel_container: Place_tokens
```

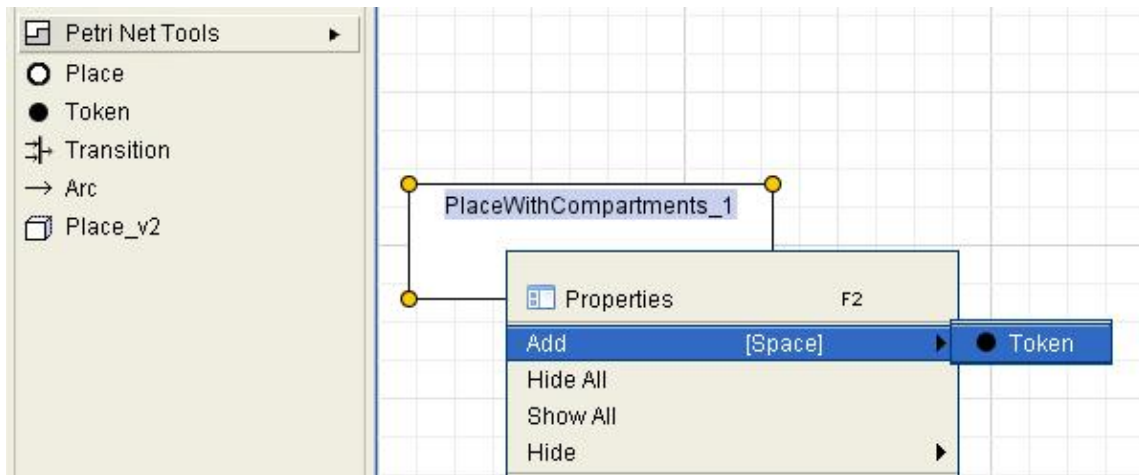
The attribute "metamodel_container" "tells" Poseidon that Tokens must be stored to the "tokens" reference which we defined one step above. Now the definition of Token should look like this:

```
node Token {
  metamodel_element: Token
  icon: "start-state"
  metamodel_container: Place_tokens
  default_size: 1 * 1
  minimum_size: 1 * 1
  shape: ELLIPSE
  name_position: NO_NAME
  keep_proportions: true
  property FillColor: BLACK
}
```

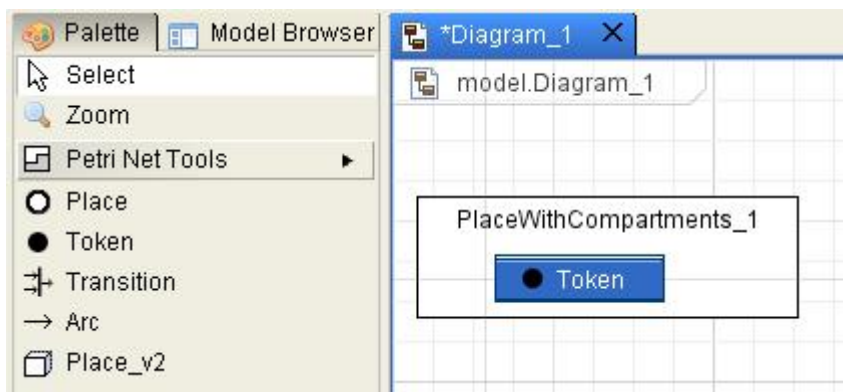
Now we have to create a tool on the palette for that new node. Open Tools model (PoseidonTools.tools) and add the following definition there:

```
node_tool Place_v2 {
  diagram_node: PlaceWithCompartments
}
```

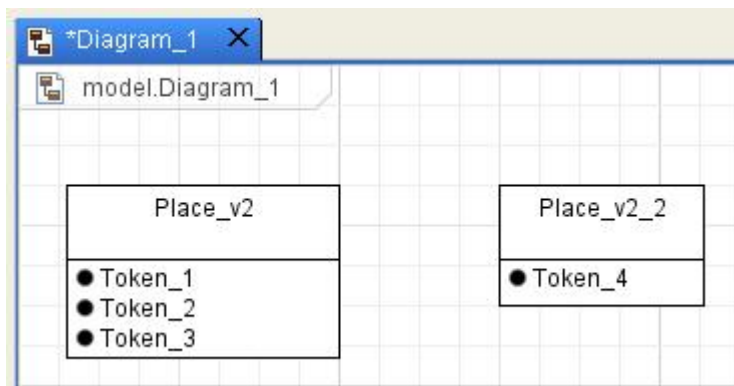
Re-generate poseidon code using Ant build "generate poseidon code from models". Start Poseidon. Now the "version 2" of the node "Place" allows you to manage Places and Tokens in a different way. You can create Tokens inside Places via context menu (invoked by right-clicking the mouse).



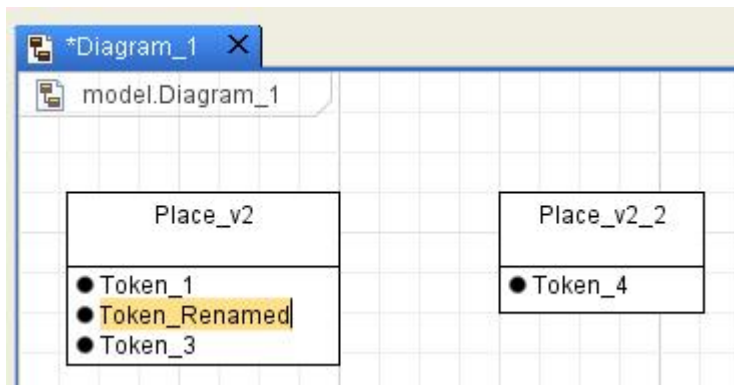
Or, you can hit Space bar when the mouse is over the node:



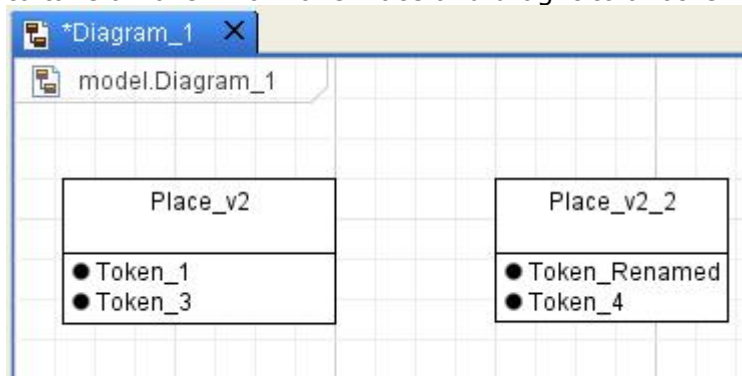
All the Tokens which you create in a Place, are listed inside the Place node by name:



You can move rename Tokens directly on the diagram:



And finally, you can move Tokens from one Place to another Place. Just use the mouse to take a Token from one Place and drag it to another Place:



Now, some last touches to make the new Place node fully functional. We have to allow Arcs to be connected to it. So, open diagram model and add node "PlaceWithCompartments" to the list of Arcs' sources and targets:

```
edge Arc {
  sources: Place Transition PlaceWithCompartments
  targets: Place Transition PlaceWithCompartments
  icon: "arrow-right-solid"
  metamodel_element: Arc
  at_target_draw: CLOSED_ARROW
  property FillColor: BLACK
}
```

Then open the rapid buttons model (RButtons.rbtn) and add node "PlaceWithCompartments" to the list of Rapid Button's anchors:

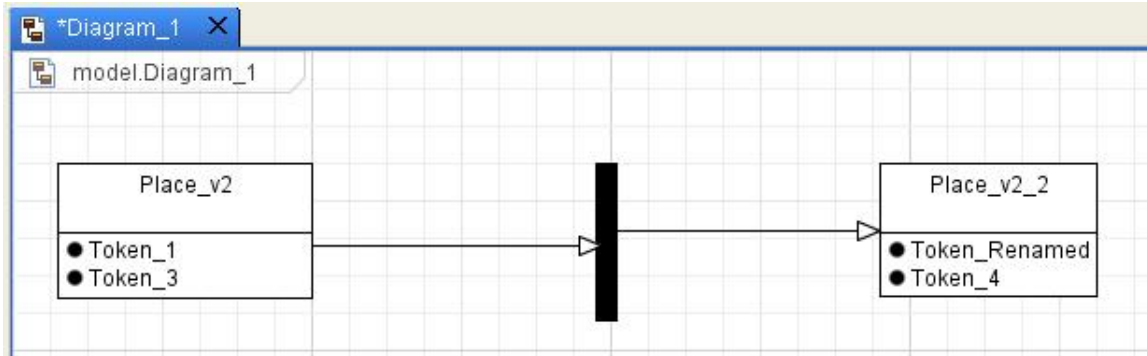
```
button Arc {
  anchors: Place Transition PlaceWithCompartments
  edge: Arc
  position: RIGHT_TOP LEFT_BOTTOM
  icon: "arrow-right-solid"
}
```

Now open edge rules model (EdgeRules.erule) and add a rule to create Transition on the target end of the Arc if the source is a "PlaceWithCompartments":

```
rules_for Arc {
  from Place create Transition
  from PlaceWithCompartments create Transition
  from Transition create Place
}
```

}

Re-generate Poseidon code and start Poseidon. Now you can use the new node to create Petri Nets:

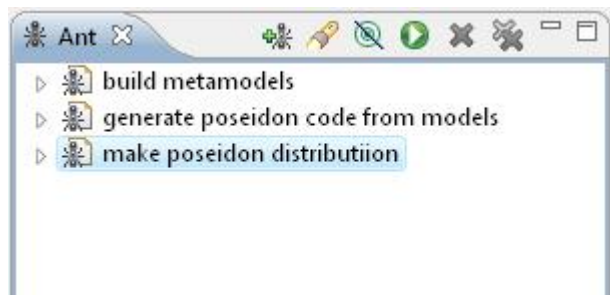


Changing splash screen

By default, Poseidon uses the "Poseidon for DSLs" splash screen when it starts. You can use your own splash screen, just replace icon "splash/splash.png" in project "**resources**" with your image.

Building distribution of your editor

When you finish working on your editor and you are satisfied with it, you want to give it to the users. There is an ant script "**make poseidon distribution**" in your ant view (if it is not there, add it, the ant build file "**make poseidon distribution.xml**" is located in project "**generators**"). This Ant build file builds Poseidon distribution.



Your Poseidon will be compiled and built, you will find the distribution in folder "target/dist" in your workspace folder.

Conclusion

During this tutorial we have developed a complete modeling tool for a graphical DSL from scratch. We started out from creating an ECore definition of the metamodel, then added the nodes and edges to the diagram model and the tools model. With just that, we had a working modeling tool but with default looks and default behavior. We then changed the look of the elements to our likings. We added rapid buttons to make developing models faster and adjusted the default behavior of edge creation. Then we fine tuned the behavior of some of the elements and took out some attributes we did not want to show. Finally we created an alternative representation for Places that contains compartments.

All in all, it is a simple and fast procedure to create a graphical DSL. The mechanisms are powerful enough to create something as complex as a complete UML tool, but at the same time they are simple enough to apply them to a small DSL that you might want to quickly assemble to speed up your development. It can be applied to a large variety of model types, such as regarding state, structure, process or composition. If you need more information, please contact Gentleware at info@gentleware.com.