Editing EMF models with Exeed

(EXtended EMf EDitor)

Dimitrios S. Kolovos,
Department of Computer Science,
The University of York

Introduction

So you have defined your metamodel using ECore (or if you know better using a textual syntax for ECore such as KM3 or Emfatic and then injecting to ECore) and now you want to start creating instances of it. You have four options (no! writing the XMI by hand is not an option):

- **Implement a visual editor:** You can use GMF (www.eclipse.org/gmf) to implement a visual editor for your language. However, be prepared for some reading and experimenting – which is a good thing if you have the time to do!
- **Implement a textual syntax:** You can use one of the text-to-model frameworks (e.g. the Xtext framework from oAW or TCS from INRIA). Again, be prepared to learn some form of EBNF to express the textual syntax of your language and the framework will generate plugins that automatically transform from the textual to the abstract syntax and vice versa.
- **Generate a tree-based EMF editor:** Using tools that ship with EMF you can generate plug-ins for a tree-based editor (similar to the ECore editor). You can then use Java to customize labels, icons and other aspects of the editor.
- **Use the built-in reflective editor:** This is the only approach that does not require generating any additional plugins to edit your models. You can simply right-click a meta-class, click *"Create Dynamic Instance"* and create a new .xmi file that can be then edited using the reflective editor. This is really the poor (or lazy)-man's choice, but also the most agile one, because it doesn't require generating/maintaining anything rather than the ECore metamodel itself. The price you have to pay though is editing your models with an ugly editor with non-customizable labels and icons.

Choosing an Approach to Editing Models

If you are working for a big – or not so big – company and you want to offer professional-level support for your modelling language, the option of going with GMF seems inevitable. If you also want to impress techies, go for a textual syntax as well.

Now if you are doing some kind of research in MDD, MDSE, MDA, MDE or something like that, you are writing small – or larger – metamodels for evaluation and experimentation purposes (and you probably have dozens of them by now – in many different places), and the only people that will ever see the example models you are

editing is you and – the bravest of – your colleagues. Therefore, the investment on time and effort to implement a GMF editor or a textual syntax will probably never pay off. This leaves you with the last two options: generating and customizing a tree-based editor or using the reflective editor.

I personally use many experimental MDE tools (ATL, ModelWeaver, Epsilon, MOFScript), which I run from source code. Therefore, I have to design my metamodels in the Eclipse runtime-workbench. Now if I wanted to generate tree-editor plugins for each of my metamodels, to instatiate them I would need to launch another Eclipse instance from my runtime workbench where I could use my generated editors. Moreover, if I wanted to generate an editor for all of my metamodels I'd probably have to manage around 50 more plugins in my runtime-workbench. This is too much for me personally so I decided to go for the reflective editor. However, using the relective editor has some major drawbacks:

- The icons that represent model elements in the model are picked – randomly? – by the editor and all are very similar to each other. In fact it is the same icon (✦) in different colors. This makes distinguishing between different types of model elements really difficult.
- The labels that represent the model elements in the editor are composed by the name of the meta-class (EClass) and the value of its first attribute. This is also not particularly helpful in most cases. For example, for EClasses that have only references, the label of an instance is just the name of the EClass.
- Because Eclipse binds editors to file extensions, all my models have to have the .ecore or .xmi extension – unlike with the generative approach where you can define an icon and extension for your specific type of model and make your generated editor the default one for this extension.

Wouldn't it be nice if you could *instruct* the reflective editor about how to construct labels for different elements and what icons you want for each type of model elements without having to generate code and do the customization in Java?

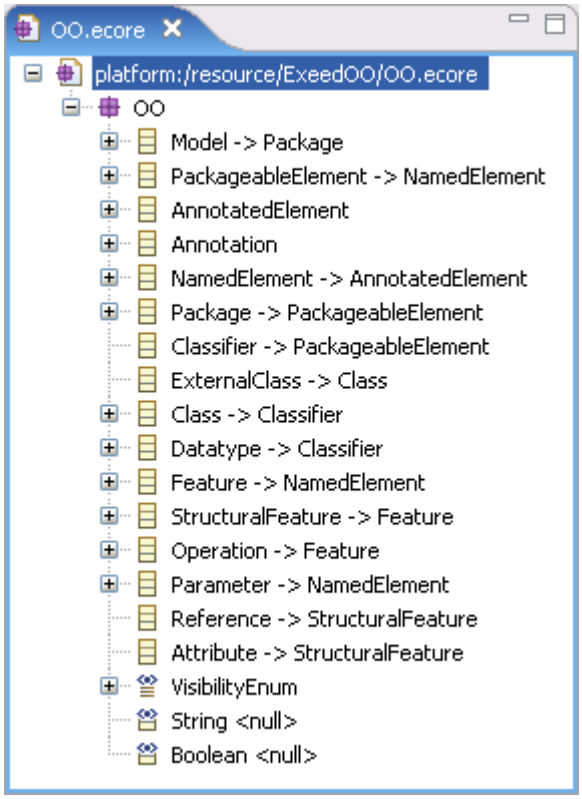
An overview of Exeed

Exeed (EXtended Emf EDitor) does exactly that. In one sentence, you can add some Exeed-specific EAnnotations to your EClasses, which provide instructions about how to format labels and icons and Exeed will then use them to visualize your models properly. You will not need to generate anything, just annotate your metamodel a bit. Let's get down to action with a real example.

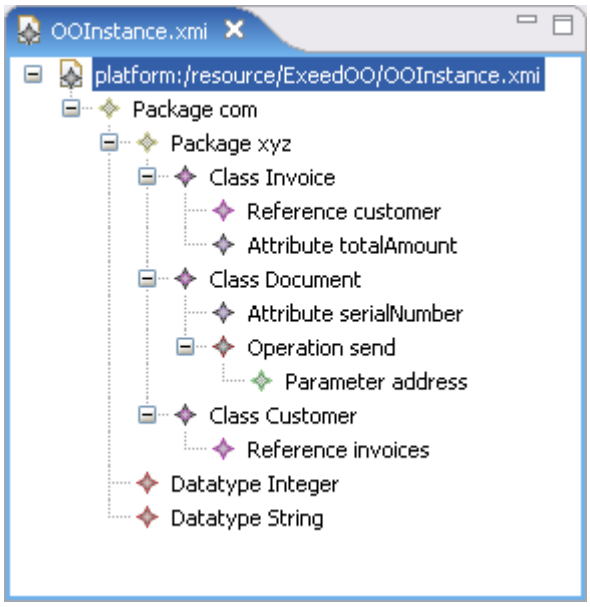
Defining an Object-Oriented Metamodel

For the purposes of this example we have defined a simplified Object Oriented metamodel (OO.ecore) with EClasses such as **Package**, **Class**, **Attribute**, **Reference**, **Operation** etc.

Dimitrios Kolovos,
Department of Computer Science,
The University of York



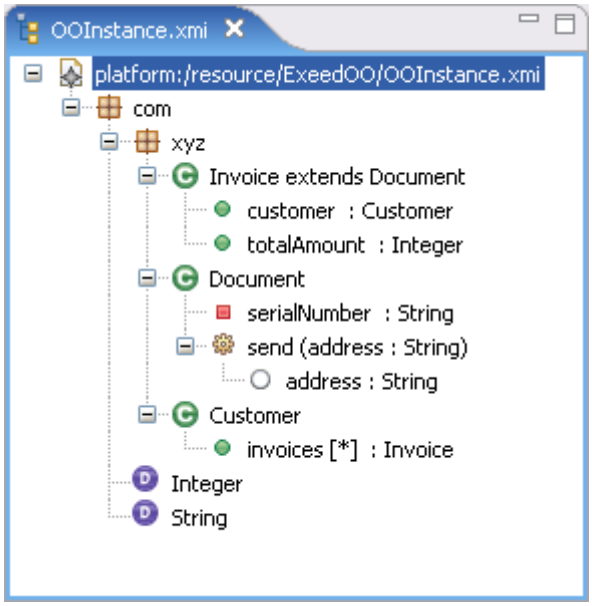
To instantiate a model that conforms to OO.ecore, you can right-click on the **Package** EClass and select *"Create Dynamic Instance"*. You can then populate the model with some model elements using the default reflective editor and get to this:



Hint: *If you close Eclipse and re-open it you will need to right-click on the OO.ecore and select "Register ECore metamodel" to open your model with either the reflective editor or Exeed.*

Dimitrios Kolovos,
Department of Computer Science,
The University of York

As you can see, this view is not very informative. By looking at it you cannot tell that Invoice extends Document, that the totalAmount attribute is an Integer, that the serialNumber is private, and that there are many invoices associated with each customer. Wouldn't it be nicer if we could have the following nicer and more informative visualization?



To achieve this visualization with Exeed, we have to annotate the ECore metamodel with information on what is a proper icon and label of each element in the model. This information is added in the form of EAnnotations attached to EClasses of the OO.ecore. More specifically, under the EClasses that need to be customized, an annotation with the value of **source** set to **exeed** must be added. Under the **exeed** annotation, three annotation details are currently supported:

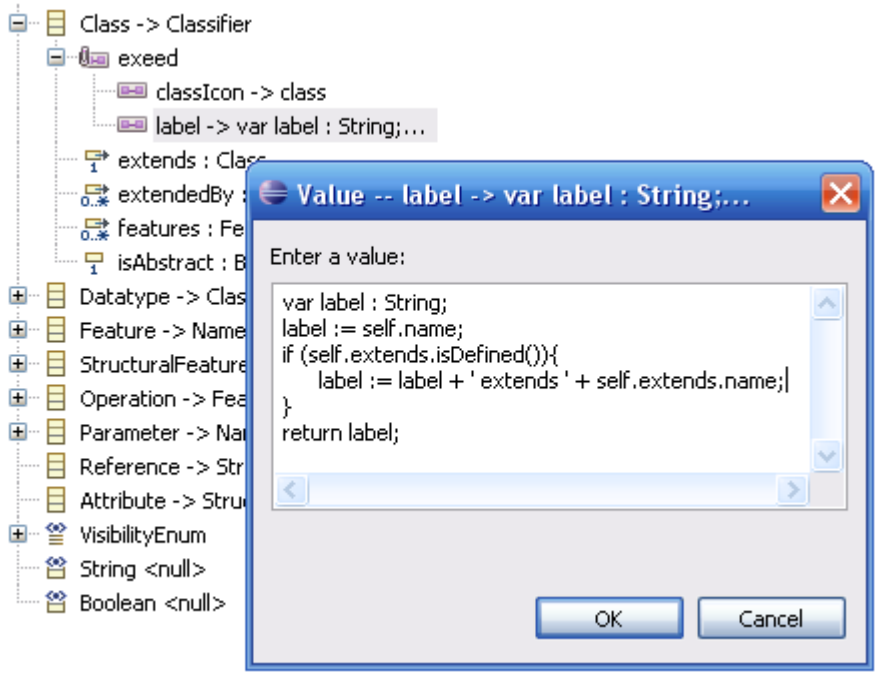
label: EOL script
icon: EOL script
classIcon: String

The label and icon values are EOL scripts that can access the current model element via the **self** built-in variable. As you will see from the examples, EOL is a simple language (something like JavaScript extended with OCL) so don 't get scared away – at least not yet. Simple EOL scripts are straightforward to write. However, for more complex scripts you'll probably need to refer to the documentation, available at www.eclipse.org/gmt/epsilon/doc.

The label annotation detail

In the value of the label detail we specify how we want the label to be calculated. For example for **Class** we specify that the label consists of the name of the class - followed by "extends" and the name of its superclass if a superclass is defined. This is shown in the following figure:

Dimitrios Kolovos,
Department of Computer Science,
The University of York



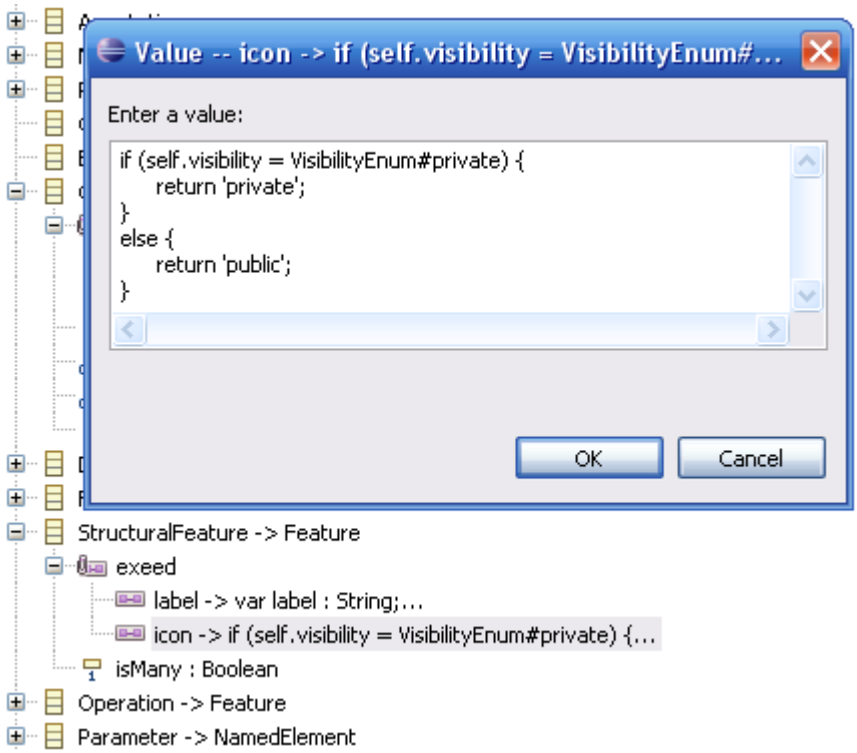
The classIcon annotation detail

In the **classIcon** label we define 'class' as the value. This means that the icon of instances of Class as well as the icon in the *New Child*, *New Sibling* menus will be set to <eclipse-installation>/plugins/org.epsilon.eclipse.dt.exeed/icons/**class**.gif. To add new icons that can be accessed by Exeed you must place them in this directory with a .gif extension (16x16 maximum).

The icon annotation detail

With the classIcon detail we can define type-level images. However, we may also want to define instance-level images. For instance, in our example we want to assign a different icon to private and public attributes (e.g. serialNumber vs totalAmount). To achieve this, Exeed also supports the **icon** annotation detail. Where both **icon** and **classIcon** details exist, the former overrides the latter for visualizing model elements. However the latter is still used in the *New Child* and *New Sibling* menus. For example, for EClass StructuralFeature we specify the following:

Dimitrios Kolovos,
Department of Computer Science,
The University of York



Here it is worth noting that Exeed respects inheritance in the metamodel. Thus, since both Attribute and Reference are subclasses of StructuralFeature and none of them defines its own icon annotation detail, they both use the specification from StructuralFeature.

Source Files for the Example

The annotated OO.ecore metamodel and the exemplar instance model can be obtained at:
http://dev.eclipse.org/viewcvs/indextech.cgi/org.eclipse.gmt/epsilon/examples/ExeedOO/ExeedOO.zip

Downloading/Installing Exeed

Exeed is redistributed as part of Epsilon. Epsilon is a platform of integrated languages for model management developed in the context of the ModelWare (www.modelware-ist.org) and ModelPlex (www.modelplex-ist.org) EU projects. Epsilon is also a component of the Eclipse GMT project. Instructions on installing Epsilon (including Exeed) and obtaining the full source code are provided at www.eclipse.org/gmt/epsilon/doc/.

Troubleshooting

For questions and/or bug reports please send a message to the Eclipse GMT newsgroup at news://news.eclipse.org/eclipse.modeling.gmt

Dimitrios Kolovos,
Department of Computer Science,
The University of York

Summary

Exeed supports a lightweight approach of defining a more human-friendly view for the built-in reflective EMF editor. Using Exeed you can customize the labels and the icons in the editor by writing simple (or more complex) scripts in a model-oriented language (EOL). Since Exeed is an extension of the default reflective editor, in case the ECore metamodel does not define annotations for some (or all) EClasses, Exeed will visualize their instances exactly as the reflective editor would. Moreover, you still have support for all the nice features of the reflective editor (e.g. refresh, drag-n-drop).

Dimitrios Kolovos,
Department of Computer Science,
The University of York