Graphiti

Development of High-Quality Graphical Model Editors

Graphiti supports the fast and easy creation of homogeneous graphical editors that visualize an underlying Domain Model using a tool-defined graphical notation and make editable. We locate Graphiti in the Eclipse Modeling arena and distinguish the framework from GMF and GEF. Then, we give an overview of the architecture and an introduction into the basic concepts of Graphiti. In the main part of the article, the framework becomes tangible by a simple example. We develop a graphical editor, which builds upon the tutorial of the Graphiti example feature. Already after a few steps we receive first payoffs. Hopefully, this inspires to dive deeper into the framework.

By Christian Brand, Matthias Gorning, Tim Kaiser, Jürgen Pasch and Michael Wenz

If one tries to build a graphical editor for a Domain Model using Eclipse, he will soon come across the Eclipse Modeling Framework (EMF) and the Graphical Editing Framework (GEF). While EMF provides the basis for modeling, GEF supports programming graphical editors. But it is immediately noticeable, that GEF is fairly complex and it needs a notable amount of work to get used to the framework. Therefore, and also to homogenize their tooling, SAP built a framework that hides GEF's complexities from the developer and bridges EMF and GEF to ease and speed up the development of graphical editors. After SAP AG decided in 2009 to donate the framework to the eclipse community, the development team was able to deliver the 0.7.0 incubation release of the Graphical Tooling Infrastructure (Graphiti) in October, 2010. We should emphasize that already the SAP-internal predecessor of Graphiti was in productive use [2], [3]. Installation instructions can be found on the project’s website [1].

Graphiti provides the following benefits for development and usage:

- low entry barrier: platform specific technologies (GEF/Draw2D) are hidden
- incremental development: fast payoffs by using default implementations, short development cycle (program, debug)
- homogeneous editors: editors on top of Graphiti look and behave similarly
- optional support of different platforms: diagrams are defined platform independently, in principle they can be rendered on any platform

Graphiti is an alternative to the Graphical Modeling Framework (GMF) that exists for some time at Eclipse. Basic differences in the paradigms of Graphiti and GMF are listed in Table 1.

	Graphiti	**GMF**
Architecture	runtime-oriented	generative
Interface	self-contained	GEF-dependent
Client Logic	centralized (feature concept)	distributed functionality
Look & Feel	standardized, defined by SAP usability experts (adaptable)	simple (adaptable in generated code)

Table 1: Graphiti vs. GMF

As paramount difference between the architectures of the frameworks we rate that GMF follows a generative approach, while in Graphiti one programs against a plain old Java interface. Graphiti's approach has the big advantage, that no generated source code must be manipulated to adapt the editor – in contrast to GMF where one has to change generated sources, which can cause headaches when regenerating.

Architecture and basic concepts

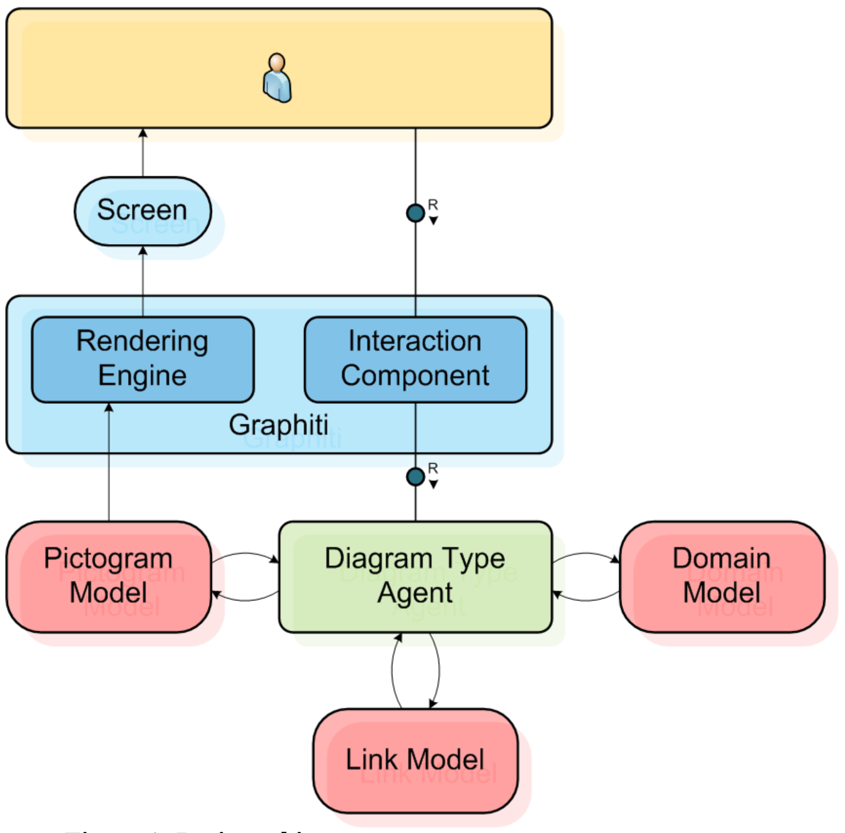

Figure 1: Basic architecture

Naturally, communication between a user and a Graphiti based tool happens via the screen as well as by mouse and keyboard (see Figure 1). An Interaction Component will receive requests such as resizing, drag-and-drop, or deleting. The actual processing of these requests happens in a so-called Diagram Type Agent, which will be discussed in the following section in more detail.

It is the objective of the Rendering Engine to display the current data on the screen. Rendering Engine and Interaction Component together form the framework components of Graphiti - and thus the actual Graphiti runtime. The technical realization of the Graphiti runtime is based on the Graphical Editing Framework (GEF) in conjunction with Draw2d. Graphiti's defaults ensure a uniform user interface (look and feel) of all Graphiti based tools.

Diagram Type Agent

As mentioned already above, the Diagram Type Agent is addressed via the Interaction Component. This happens over a standardized interface. The Diagram Type Agent has to be implemented and made available by the developer. For this implementation developers can make use of lots of various services as well as many meaningful standard implementations.

Assistance of the services and standard implementations can support a developer to create his own graphical editor really fast. For example, actions like move, resize, delete, remove, and print are available immediately. One should not regard a first version as a prototype which has to be thrown away later. Rather the initial version can be extended to a complete product incrementally with increasing Graphiti know-how and rising requirements.

In other words: In the beginning a developer comes to a functional editor very quickly. This editor can be adorned gradually (e.g. color gradients) and can be equipped with additional functionality (e.g. direct editing, extension of the context menus and context buttons).

A Diagram Type Agent’s major task is to modify the model data. On the one side we have the Pictogram Model (incl. Link Model) whose metamodel is made available by Graphiti and on the other side we have the domain specific model which comes from the developer. These models are explained in the following.

Domain Model

The Domain Model contains the data which has to be visualized graphically. A developer would for example use the Ecore metamodel for a graphical Ecore editor. An editor for BPM (Business Process Management) would use the Businesses Process Modeling Notation.

It is of advantage (however not absolutely necessary) if this model is available in EMF just like the Pictogram Model. But Graphiti can deal also with models outside EMF. In Graphiti, the data of the Domain Models are called Business Objects.

Pictogram Model

The Pictogram Model contains the complete information for representing a diagram. That implies that each diagram can be represented without the presence of the Domain Data. This requires a partially redundant storage of data, which is present both in the Pictogram Model and in the Domain Model. Imagine the class names in case of an Ecore editor. Since data redundancy always draws the problem of the synchronization, Graphiti offers a concept to update this data. Data which is out of sync can be visualized graphically and can be corrected (semi-) automatically by the use of so-called Update Features. This approach enables an editing of Domain Data with different tools and a subsequent updating of already existing diagrams.

A detailed discussion of the Pictogram metamodel would blow up the content of this article. Further information can be found in the „org.eclipse.graphiti.mm “-plugin which contains the metamodel and some diagrams. The diagrams can be viewed with the help of the Ecore tooling [4].

Later we will show an example where data of the Pictogram Model is produced for the graphical representation.

Link Model

The Link Model is responsible for connecting data from the Domain Model and the graphical representation (that is, data from the Pictogram Model). These connections are again needed by many actions in the graphical editor. For instance, a deletion or a move of a graphical object needs also access to the associated object of the Domain Model in order to be able to make the necessary changes.

Tasks of the Diagram Type Agent

The tasks of the Diagram Type Agent can be explained most conveniently on the basis of a small example - in this case we use a create operation which is initiated by the user through the Interaction Component. The user clicks into the tools palette of the graphical editor and draws a rectangle in the diagram to indicate size and position of the object to be created. Now the Diagram Type Agent does the following:

- Create a new object in the Domain Model
- Create the graphical visualization in the Pictogram Model. E.g., create Graphics Algorithms like a rounded rectangle or a text element. Initialize them with colors and fonts and do the layout.
- Create the link between Pictogram Model object and Domain Model object (Link Model).

The Feature Concept

Figure 2 displays the internal structure of the Diagram Type Agent. The actors shown there are to be provided by the developer. This means he has to implement an amount of so-called Features which are similar to operations. Here are some typical examples for Graphiti Features:

- Add Features create the graphical representation of Domain Model elements in the Pictogram Model.
- Create Features produce new objects in the Domain Model as well as the appropriate graphical representations. Here it makes sense to reuse existing Add Features.
- Remove Features delete graphical representations in the Pictogram Model.
- Delete Features delete both, a Domain Model object and its graphical representations, in the Pictogram Model.

A Feature Provider supplies the needed features considering the given context. Processing the features leads to the modification of the appropriate data specified above.

Further diagram type specific and tool specific requests to the Interaction Component are handled by the Diagram Type Provider. This includes the selection of the appropriate update strategy, that is, the decision when synchronizations between Domain and Pictogram Model have to be done.

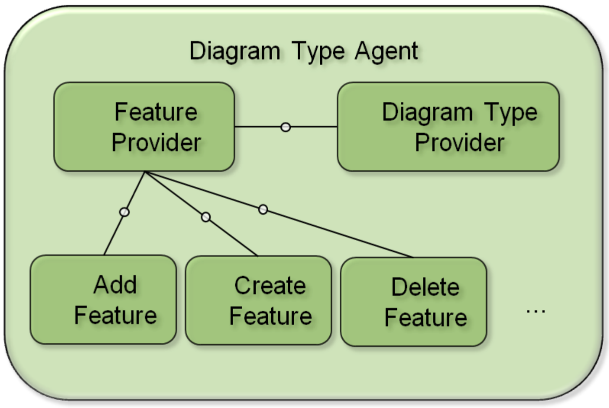

Figure 2: Construction of the Diagram Type Agent

Example

In this example we implement a simple Ecore editor for metamodels with which we can create the graphical representation of an EClass. This will be shown as a rounded rectangle that contains a separator-like horizontal line above which the name of the EClass appears as text.

We create the Diagram Type Agent in four steps: first we implement a Diagram Type Provider, and then we register it for our new Diagram Type. In the third step, a Feature Provider is created for which we implement an Add Feature in the end.

For our *MyTutorialDiagramTypeProvider* we extend the standard implementation AbstractDiagramTypeProvider:

```
package org.eclipse.graphiti.examples.tutorial.diagram;
public class MyTutorialDiagramTypeProvider extends AbstractDiagramTypeProvider {
public MyTutorialDiagramTypeProvider() {
       super();
}
```

The *second step* consists of the definition of the *mytutorial* Diagram Type. Information about a Diagram Type must be declared in the plugin.xml with the extension point *diagramTypes* (see listing 1). The assignment of *mytutorial* to *MyTutorialDiagramTypeProvider* and registering this combination in the framework is accomplished using the extension point *diagramTypeProviders*. Thus, the sample editor is registered in Eclipse.

Listing 1: Registering Diagram Type Provider and Diagram Type

```
<extension point="org.eclipse.graphiti.ui.diagramTypes">
  <diagramType
    description="This is the diagram type for my Graphiti tutorial"
    id="org.eclipse.graphiti.examples.tutorial.diagram.MyTutorialDiagramType"
    name="My Graphiti Tutorial Diagram Type"
    type="mytutorial">
  </diagramType>
</extension>
<extension point="org.eclipse.graphiti.ui.diagramTypeProviders">
  <diagramTypeProvider
    class="org.eclipse.graphiti.examples.tutorial.diagram.
           MyTutorialDiagramTypeProvider"
    description="This is my editor for the Graphiti tutorial"
    id="org.eclipse.graphiti.examples.tutorial.diagram.
        MyTutorialDiagramTypeProvider"
    name="My tutorial editor">
    <diagramType
      id="org.eclipse.graphiti.examples.tutorial.diagram.MyTutorialDiagramType">
    </diagramType>
  </diagramTypeProvider>
</extension>
```

End

In the *third step* we are again using an existing standard implementation for our Feature Provider.

```
public class TutorialFeatureProvider extends DefaultFeatureProvider {
    public TutorialFeatureProvider(IDiagramTypeProvider dtp) {
            super(dtp);
        }
    }
```

Finally, we must create and set *TutorialFeatureProvider* in our *MyTutorialDiagramProvider* (see Figure 2) with the method *setFeatureProvider* :

```
    public MyTutorialDiagramTypeProvider() {
           super();
           setFeatureProvider(new TutorialFeatureProvider(this));
    }
```

At this point we are already able to create a diagram and open it in the editor.

The first three steps took us quickly out of hand, for the remaining Add Feature we have to buy a little more effort. Figure 3 illustrates how the EClass will look like in the editor (Result). The upper part of the figure shows how the Pictogram Elements of the Add Feature, which are necessary for the graphical representation, are wired with the Domain Object.

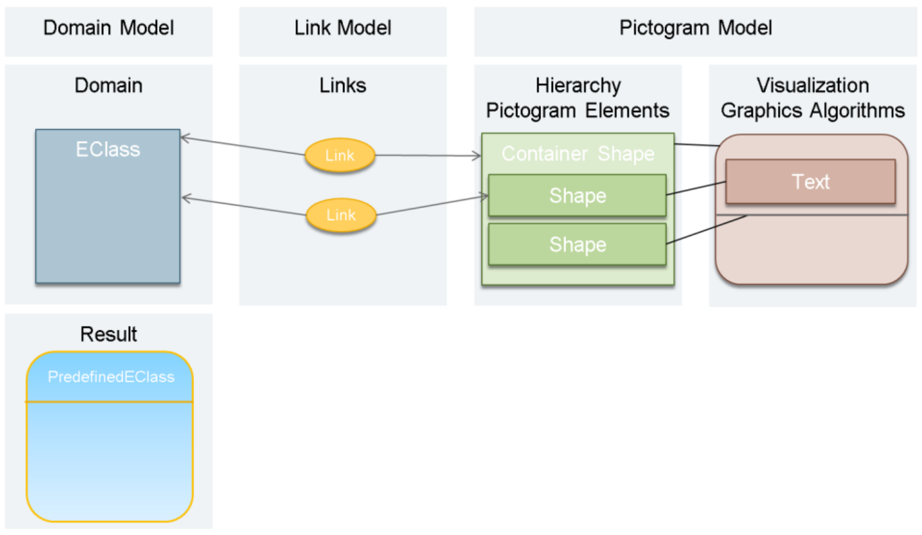

Figure 3: Linkage of the involved artifacts

In the left column of *the Pictogram Model* a Container Shape aggregates two child Shapes, which are responsible for the Text Label and the separation line. All three Shapes aggregate specific Graphics Algorithms providing the complete information to allow the rendering of the EClass in the editor. Examples for such information are positions, fore- and background colours or a gradient, visibility, and line width as well as a line style. In our example we have a *Rounded Rectangle* providing an additional corner radius, a *Polyline* having defined end- and bend points, and a Text *Label* containing a text value.

The Container Shape and likewise the Shape containing the Text Label are related to a corresponding EClass from the Domain Model. The relationship is realized through special *Link* Objects which contain both ends.

In listing 2, we find the complete implementation of the Add Feature that can be easily understood. As basis serves the standard implementation *AbstractAddShapeFeature* from which the methods *canAdd* and *add* are overwritten.

Due to the given Context, *canAdd* has to check whether the given Business Object can be used. The method *add* produces the above described Pictogram Elements, Graphic Algorithms, Links and establishes the linkage to the Business Object. Using *IPeCreateService* and *IGaService,* Pictogram Elements and Graphics Algorithms are easily created by extracting the positions from the Context.

Listing 2: The Add Feature

```
public class TutorialAddEClassFeature extends AbstractAddShapeFeature {
    private static final IColorConstant CLASS_TEXT_FOREGROUND =
        new ColorConstant(51, 51, 153);
    private static final IColorConstant CLASS_FOREGROUND =
        new ColorConstant(255, 102, 0);
    public TutorialAddEClassFeature(IFeatureProvider fp) {
        super(fp);
    }
    public boolean canAdd(IAddContext context) {
        // check if user wants to add a EClass
        if (context.getNewObject() instanceof EClass) {
            // check if user wants to add to a diagram
            if (context.getTargetContainer() instanceof Diagram) {
                return true;
            }
        }
        return false;
    }
    public PictogramElement add(IAddContext context) {
        EClass addedClass = (EClass) context.getNewObject();
        Diagram targetDiagram = (Diagram) context.getTargetContainer();
        // CONTAINER SHAPE WITH ROUNDED RECTANGLE
        IPeCreateService peCreateService = Graphiti.getPeCreateService();
        ContainerShape containerShape =
             peCreateService.createContainerShape(targetDiagram, true);
        // define a default size for the shape
        int width = 100;
        int height = 50;
        IGaService gaService = Graphiti.getGaService();
        {
            // create and set graphics algorithm
            RoundedRectangle roundedRectangle =
                gaService.createRoundedRectangle(containerShape, 5, 5);
            roundedRectangle.setForeground(manageColor(CLASS_FOREGROUND));
            //set color gradient
            roundedRectangle.setRenderingStyle(
                    PredefinedColoredAreas.getBlueWhiteGlossAdaptions());
            roundedRectangle.setLineWidth(2);
            gaService.setLocationAndSize(roundedRectangle,
                context.getX(), context.getY(), width, height);
            // if added Class has no resource we add it to the resource
            // of the diagram
            // in a real scenario the business model would have its own resource
            if (addedClass.eResource() == null) {
                     getDiagram().eResource().getContents().add(addedClass);
            }
            // create link and wire it
            link(containerShape, addedClass);
        }
        // SHAPE WITH LINE
        {
            // create shape for line
            Shape shape = peCreateService.createShape(containerShape, false);
            // create and set graphics algorithm
            Polyline polyline =
                gaService.createPolyline(shape, new int[] { 0, 20, width, 20 });
            polyline.setForeground(manageColor(CLASS_FOREGROUND));
            polyline.setLineWidth(2);
        }
        // SHAPE WITH TEXT
        {
            // create shape for text
            Shape shape = peCreateService.createShape(containerShape, false);
            // create and set text graphics algorithm
            Text text = gaService.createDefaultText(shape,addedClass.getName());
            text.setForeground(manageColor(CLASS_TEXT_FOREGROUND));
            text.setHorizontalAlignment(Orientation.ALIGNMENT_CENTER);
            text.setVerticalAlignment(Orientation.ALIGNMENT_CENTER);
            text.getFont().setBold(true);
            gaService.setLocationAndSize(text, 0, 0, width, 20);
            // create link and wire it
            link(shape, addedClass);
        }
        return containerShape;
    }
}
```

End

At the end of our small example, we only need to override *getAddFeature* from our Feature Provider to deliver our new Feature:

```
    @Override
    public IAddFeature getAddFeature(IAddContext context) {
            //is object for add request a EClass?
            if (context.getNewObject() instanceof EClass) {
                return new TutorialAddEClassFeature(this);
            }
            return super.getAddFeature(context);
    }
```

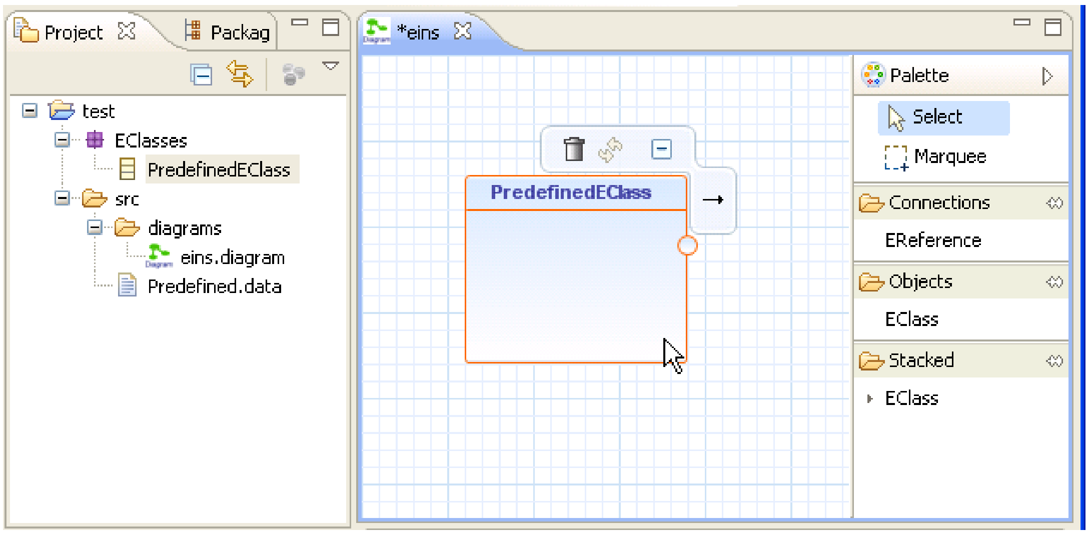

Figure 4: The EClass editor

Using the Project Wizard (NEW … → GRAPHITI DIAGRAM) we can create a new diagram and open it in the editor. Drag-and-drop gets the *PredefinedEClass* on the diagram, as it is shown in Figure 4. Without any further implementations the EClass Shape can be moved, resized, and even deleted.

Conclusion and Outlook

This article has given a first impression on writing graphical tools using the Graphiti framework. After a short introduction to the programming paradigm and the framework itself, a first version of a ready-to-run graphical editor for Ecore metamodels was developed. The amount of code necessary for this was very low, which was due to the rich standard implementations that are already part of the framework.

The Graphiti framework version 0.7.0 is part of the Eclipse GMP project in the Modeling track and available for use. For the next release, which will appear as part of the Indigo release train in June 2011, the developer team plans to improve specific details in the framework and further simplify the programming model.

The authors are working for SAP AG in the area of Java tool development and modeling. They were able to gain rich experience in a variety of Eclipse technologies over the last years.

Links & Literature

[1] Project Graphiti: http://www.eclipse.org/graphiti/

[2] Volker Stiehl: *Business Process Modeling Notation – Vermittler zwischen den Welten*, Teil 1-4, Java Magazin, Ausgaben 9.2009-12.2009
[3] Ulf Fildebrandt: *Erweiterbare Toolentwicklung – Wie man erweiterbare Werkzeuge auf Basis von Eclipse baut*, Eclipse Magazin, Ausgabe 01.2010
[4] Ecore Tooling: http://www.eclipse.org/modeling/emft/?project=ecoretools