

Graphical Modeling Framework Architecture Overview

Version 1.0, September 28, 2006

Alexander Shatalin, Borland Software Corporation, Prague, Czech Republic – alexander.shatalin@borland.com

Artem Tikhomirov, Borland Software Corporation, Prague, Czech Republic – artem.tikhomirov@borland.com

Abstract

Recent creation of Eclipse Modeling Project (EMP) should significantly simplify the usage of Model Driven Engineering (MDE) techniques in real applications. A number of open source, high quality sub-projects should be established and hopefully will reach release state as a part of this project. The Graphical Modeling Framework (GMF) project was recently assigned to EMP as a tool for the creation of Graphical Editors for user-defined meta-models. At the same time, GMF itself uses several MDE principals internally and probably could be represented as a good source of requirements and potential customer of neighboring projects. The main idea of this paper is to present an overview of the current GMF architecture and highlight potential places where other sub-projects of EMP could be reused.

1. Introduction

Currently GMF [1] is one of sub-projects in top-level Eclipse Modeling Project [2]. This project was announced at EclipseCon 2005. First version of GMF was released as a part of the Callisto Simultaneous Release [3] in June 2006 with an upcoming maintenance release at the end of September. The next major release is 2.0 and will be a part of Europa Simultaneous Release [4].

"The Eclipse Graphical Modeling Framework (GMF) provides a generative component and runtime infrastructure for developing graphical editors based on EMF and GEF." As it is specified, GMF consists of two parts – generative and runtime. The runtime part could be described as a set of plug-ins extending existing EMF [5] and GEF [6] functionality. The runtime not only allows easier integration between EMF and GEF, but provides additional services like: transactions support, extended meta-modeling facilities, notation meta-model, variability points used for runtime extensibility of generated code, etc.

The generative part mainly provides user with possibility to define future diagram using specially designed EMF models, and to generate code using this information. A GMF user will get all the advantages of a generative approach – fully functioning diagramming code automatically generated and easily re-generated upon model modification. Additional generator-specific variability points aid in customization of the generated code for the end-user. Generated code is optimized and proved by a large community. As you will recognize, the generative part mostly involves working on developing a new language for describing diagrams and fits perfectly into the Generative Programming [7] or Software Factories [16] concepts.

Below you will find an overview of the main EMF meta-models defined in both parts of GMF, along with a description of standard user workflow used while generating diagramming code.

2. GMF meta-models

As it was mentioned, GMF defines a number of specific meta-models. Some of these meta-models will be instantiated at runtime and will be used by generated diagram editor. An example of this kind of model is the notation model.

2.1 Notation model

The notation model is used to store visual information necessary for diagram drawing and is independent from the underlying semantic model. For example, it is used to store the position and size of nodes, link bend point locations, color styles, etc. The notation meta-model could be compared with Diagram Interchange meta-model [8] as it stores a similar set of visual information. The question of aligning these two meta-models was already raised and hopefully will bring to the corresponding changes in OMG specifications in future.

The notation model is mostly based on the following sub-classes of abstract class View: Diagram, Node and Edge. Each of these subclasses is used to store a corresponding set of properties specific for Diagram, Node or Edge visual representation, respectively. The notation model can be extended with some user-defined data by attaching specific Style to the existing View instance. The notation meta-model is most stable meta-model in the GMF.

Each opened diagram editor has own notation model instance attached and is actively used by the runtime as an API for diagram-specific information exchange, and to store corresponding information between editing sessions using standard EMF persistence mechanism.

2.2 Graphical Definition model

The Graphical Definition model is instantiated to gather information about a future diagram editor from a toolsmith and to feed corresponding parameters into the code generator subsystem. This model is not available in the runtime environment, as this information is used only for code generation while creating the diagram editor. GMF contains a number of such meta-models describing different aspects of future diagram editors.

The graphical definition model is used to describe composition of GEF figures forming diagram elements – Nodes, Connections, Compartments and Labels. Diagram element figures can be composed either from standard GEF figures or from custom ("legacy") figures – hand-coded classes implementing the IFigure interface. Either standard GEF layouts or custom layouts can be used to arrange nested figures in accordance with attached layout data (Custom Layout Data).

The information stored in this model is enough to generate the code of described composite figures. Generated figures can be

instantiated later on by the diagram editor and used to visualize diagram elements.

The graphical definition model can fully describe composite figure structures, but there is no place to describe behavioral aspect of generated figures, so it is impossible to generate a really complex GEF figure listening for events and performing some actions based on these events. In case such figures are needed, either custom figures should be used or generated code can be customized by user.

Since Graphical Definition model deals only with graphical elements and have no connections to domain-specific aspects, this model can be used to generate plain GEF figures for any GEF-based applications. In other words, this model could be presented as an internal model for a GEF visual editor [9]. Moreover, Graphical Definition meta-model was designed to describe generic graphical shapes with no dependencies on GEF so this model potentially could be used as a source of information to visualize described figures using any other diagram editor(s) like: Microsoft Domain-Specific Language Tools [18].

2.3 Tooling Definition model

Another important aspect of diagram editor description is definition of diagram palette (toolbar), additional main menu items, popup menus, other UI actions. This aspect of the diagram description is covered by Tooling Definition model.

Currently, the Tooling Definition is basically used to describe desired diagram palette tools set, which consists of standard tools like Selection Tool, Zoom Tool and Creation Tool. It is possible to organize tools into groups and attach appropriate icons. Based on information stored in this model, the PaletteFactory code will be generated. This model is likely to evolve in the future in order to provide user with more options.

Unlike Graphical Definition model, it is not possible to generate code only using information stored in this model. The Tooling Definition model should be referenced from Mapping Model to generate reasonable code.

2.4 Mapping model

GMF distinguishes between two independent aspects of diagram definition – Graphical and Tooling. Another important aspect of future diagram is underlying (semantic) meta-model. Since GMF was designed to create diagrams based on EMF, this aspect is covered with a user-defined EMF model (.ecore file). To fulfill the diagram description, it is necessary to bind these three aspects together, and this is responsibility of Mapping model.

While working with mapping definition, the toolsmith can create such elements as: CanvasMapping, NodeMapping, LinkMapping. These mappings are used to fully describe the diagram surface, diagram nodes, and diagram links, respectively. Each mapping connects the domain model element from EMF model with its corresponding visual representation from Graphical Definition model, and to a necessary set of tools from Tooling model. In addition, Link and NodeMappings could own LabelMappings used to visualize some attributes of underlying semantic model elements as a diagram labels. The LabelMapping keeps references to attributes from semantic model and to a Label element from Graphical Definition model. NodeMappings optionally own

CompartmentMappings, which are used to logically group a set of child nodes inside compartment figure. CompartmentMapping keep a reference to Compartment from Graphical Definition model.

Node and LinkMappings store all the necessary information to present some particular semantic model element as a diagram link, or a node with labels and compartments. However, this information is not enough to embed one diagram node into another. ChildReference instances were designed to solve this problem. Each Canvas or NodeMapping can contain arbitrary number of ChildReferences. A ChildReference used to point to another NodeMapping which could be located in this Canvas or NodeMapping. In addition, a ChildReference keeps the necessary information for getting corresponding semantic children and storing a newly created one. In other words, ChildReferences capture information about the child-parent relationship for different types of future diagram nodes.

Each Mapping model represents complete diagram description while Tooling and Graphical models are used to describe only some particular aspects of diagram definition. As a result, the same Tooling/Graphical models could be reused across different Mapping models representing different diagrams.

GMF can also be used to describe pure-design diagrams. In this mode Mapping model will not keep any references to the underlying EMF model. Generated diagram code will work only with Notation model to store all information about diagram nodes. This mode could be useful if diagram information should not be used by any other subsystems except diagram editor itself.

2.5 Generator model

Graphical, Tooling definition and Mapping model (high-level models) together with EMF meta-model description completely determine future diagram structure. These models organize a kind of specific language for diagram definition. This language could be successfully used to gather the required information from the toolsmith, but is not well-suited for code generation tasks. A code generator needs some more generation-specific details like class/package names for some particular generated classes, and some implementation-specific parameters to allow fine-tuning of generated code, etc. Moreover, code generation templates are able to be significantly simplified by introducing a specific Generator model.

This model operates with such elements like: GenPlugin (used to hold necessary parameters for generating Eclipse plug-in), GenDiagram (used to generate diagram editor), GenClass (used to generate GEF EditParts and EditPolicies), etc. The idea of this model is to reflect all of the necessary details and simplify the code generation process as much as possible by moving some parts of code generator logic to the process of transformation from high-level GMF models to the Generator model. Generator model could be compared with EMF .genmodel – both models describes the same concepts.

Like in EMF, the GMF Generator model can be automatically created from high-level models. Unlike in EMF, GMF significantly reorganizes the diagram description while creating Generator model. The transformation from high-level GMF models to Generator model is not one-to-one transformation like

in EMF, so more powerful techniques are used here. Currently this transformation is hand coded in Java classes, while some model-to-model transformation techniques could be used there in the future.

Most of the diagram code can be generated without any modifications to the Generator model that is created by automatic transformation process, or with simple modifications like: changing diagramming plug-in name, ID, package statements, etc. These changes can be done in dedicated places in the Generator model so it is not necessary to change properties of all the elements in Generator model. Corresponding changes should be preserved on the next high-level model → Generator model transformation such that a toolsmith will be able to change the high-level models later on and regenerate diagramming code for the updated description. If some particular transformation engine will be used in GMF to perform high-level → Generator model transformation in future, it will definitely require support for such functionality in order to be effectively utilized by GMF.

3. Generating diagram code with GMF

The typical GMF user scenario contains several steps to get diagram code generated. Some of these steps should be performed by the toolsmith, and some will be done by GMF. The purpose of this part of paper is to highlight the steps in diagram code generation process, which could be simplified by using some MDE techniques developed as a part of EMP.

1. Define Graphical and Tooling Definitions, Mapping model. This step requires user interaction.
2. Transform high-level GMF models to Generator model. Currently plain Java code is used to perform this transformation. It is planned to use some model-to-model transformation engine for this step in the future.
3. The Generator model may be modified by the toolsmith to fine-tune generated code.
4. Generate diagramming plug-in from Generator model. Currently, the JET [10] template engine is used to generate code. This template engine is not well suited for GMF's purposes. To fully cover all GMF requirements and make templates simple, this engine should be replaced by a more flexible framework in future.
5. Store all GMF models together with generated code in version control, change high-level models and continue from step-2. This step is important because in case of concurrent development there could be collisions which lead us to the question of merging two GMF models. Currently, default EMF persistence mechanism is used to store all GMF models and there is no usable merge functionality for these types of files. This problem could be solved by utilizing some EMF model merge functionality or by using textual notation as a way to persist GMF models.

From this overview of the diagram editor generation process it is clear that there are a number of places in GMF workflow where other sub-projects of EMP could be used.

4. Modeling sub-projects currently used in GMF

The following components are used by the generated GMF diagram code, in addition to the GMF runtime:

- EMF OCL [11]
- EMF Validation [12]
- EMF Query [13]
- EMF Transaction [14]

These projects will be moved to EMP in near future. With this movement, GMF will start using Model Query, Model Transaction and Validation Framework components of EMF within the EMP. It sounds natural to utilize more newly established EMP sub-projects to fulfill the gaps highlighted in this paper.

5. Modeling sub-projects not used in GMF yet

GMF is not the only project developed as a part of EMP. As you can see from this paper, this is a project actively using MDE techniques internally. Beside currently using other sub-projects of EMP, it is planned to introduce some more dependencies. This will help GMF to focus on its primary task – providing more suitable environment for generating diagram editors. At the same time other EMP sub-projects can benefit from this decision because GMF developers will provide them with the set of requirements based on some real usage experience, participate in communities, and perform some testing activities.

As it was specified above, the current architecture GMF can benefit from the following projects:

- Model to Model Transformation [15]
- Model to Text Transformation
- Textual Modeling Framework

6. Conclusion

This paper contains a brief description of current GMF architecture clarifying that the GMF is a project actively utilizing a number of MDE techniques. Some set of corresponding techniques are currently represented as a separate EMP sub-projects. Another set was developed as a part of GMF. It is clear that future development of corresponding MDE tools lies out of the scope of GMF project. At the same time, GMF developers are ready to contribute ideas, and serve as customers for these corresponding projects. This scenario looks reasonable and beneficial for GMF and all the mentioned modeling sub-projects.

Acknowledgements

Thanks to Richard Gronback for his review of this paper and valuable input.

References

1. Graphical Modeling Framework (GMF)
<http://www.eclipse.org/gmf>
2. Eclipse Modeling Project (EMP)
<http://www.eclipse.org/modeling>
3. Callisto Simultaneous Release
<http://www.eclipse.org/callisto>
4. Europa Simultaneous Release
<http://www.eclipse.org/projects/europa.php>

5. Eclipse Modeling Framework (EMF)
<http://www.eclipse.org/emf>
6. Graphical Editor Framework (GEF)
<http://www.eclipse.org/gef>
7. Generative Programming
<http://www.generative-programming.org>
8. Diagram Interchange meta-model
http://www.omg.org/technology/documents/modeling_spec_catalog.htm#UML_DI
9. GEF visual editor
<http://www.eclipse.org/vep>
10. Java Emitter Templates (JET)
<http://www.eclipse.org/emft/projects/jet/#jet>
11. EMF OCL
<http://www.eclipse.org/emft/projects/ocl/#ocl>
12. EMF Validation
<http://www.eclipse.org/emft/projects/validation/#validation>
13. EMF Query
<http://www.eclipse.org/emft/projects/query/#query>
14. EMF Transaction
<http://www.eclipse.org/emft/projects/transaction/#transaction>
15. Model to Model Transformation
<http://www.eclipse.org/proposals/m2m>
16. Software Factories
<http://msdn.microsoft.com/vstudio/DSLTools/default.aspx?pull=/library/en-us/dnbda/html/softfact3.asp>
17. Microsoft Domain-Specific Language Tools
<http://msdn.microsoft.com/vstudio/DSLTools/>