Magyar Kutatók 8. Nemzetközi Szimpóziuma
8th International Symposium of Hungarian Researchers on Computational Intelligence and Informatics

A General Purpose Model Visualization Environment

Tamás Mészáros, Gergely Mezei, Hassan Charaf

{mesztam, gmezei, hassan}@aut.bme.hu

Abstract: The creation and handling of Domain-Specific Modeling Languages (DSML)is a fastly evolving research area of software development. DSMLs are able to describe the modeled problems in an abstract and expressive way, often better than general purpose modeling languages, like UML would do. DSMLs can also be used by business specialists who do not have exhaustive programming knowledge, consequently, the visualization of such models is of key importance. Visual Modeling and Transformation System (VMTS) is a general purpose metamodeling environment, which facilitates visual editing of models and metamodels. Furthermore, it also provides a powerful framework (VMTS Presentation Framework – VPF), which is suitable for designing model element appearance in a very comfortable way. The goal of this paper is to describe the design decisions of VPF and compare its capabilities with other metamodeling environments.

Keywords: Metamodeling, Domain-Specific Language

1 Introduction

Model-driven software engineering became an essential approach in software development. The usage of general purpose modeling languages is not always satisfactory, because they do not exactly fit to the modeled area. Domain-Specific Modeling Languages fulfill this requirement, though, they have to be created separately for each and every application area. Metamodeling is an approach to define the elements and the relations of such a language, however, the definition of the concrete syntax (the visualization of the elements) is also needed.

Visual Modeling and Transformation System (VMTS) is an n-layer metamodeling environment [2]. The VMTS Presentation Framework (VPF) is the graphical environment part of VMTS used for displaying and editing the models with their proprietary representation. VPF highly builds on the newest technologies provided by the .NET Framework including WPF (Windows Presentation Foundation) and XAML [3]. XAML is an XML based language developed to describe the visual appearance of Windows controls. Although the framework was written in C# and it is based on .NET technologies, the solutions discussed here are reusable in every high-level programming environment.

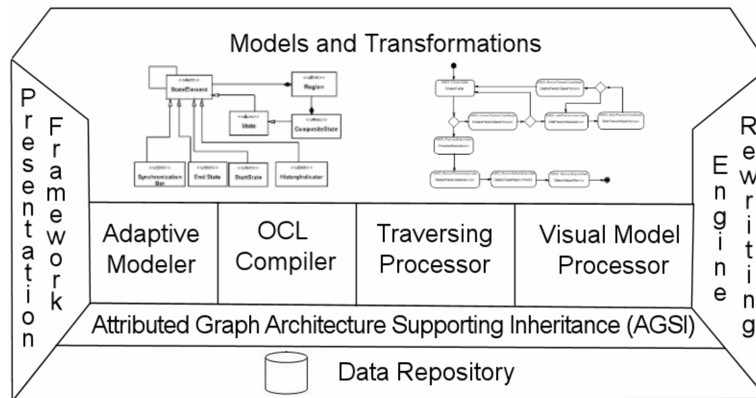

Figure 1
The schematic structure of VMTS

In Figure 1 the schematic structure of VMTS can be seen. VPF keeps contact with the models - stored in the database – through the AGSI data access layer. The modeling space of VMTS (and AGSI) distinguishes two types of model elements: nodes and edges. Nodes represent the entities in the model, while edges define the relations between the entities. However, edges are attributed in VMTS, so they can also have their own properties. Nodes and edges are organized into models (*Package* in AGSIs terminology), and the container object of several models is called *Project*.

There are several frameworks that have graphical editing support for the DSMLs in a more or less user-friendly way, but none of them is fully capable of expressing the domain-specific constraints. Although the visualization of DSMLs is not fully supported in these frameworks they use many notable solutions. Besides the introduction to metamodeling in VPF, this paper also introduces the features available in other metamodeling frameworks.

2 Related Work

The Generic Modeling Environment (GME) [4] is a highly configurable metamodeling tool supporting two layers: a metamodel-, and a modeling layer. GME uses a plugin-based architecture. Plugins can be defined using a COM interface. In GME the basis of the modeling is the modeling paradigm. Model paradigms act as the metamodel for the particular domain specific language. GME is a graphical metamodeling environment that supports the basic requirements for editing metamodels. Moreover, it can be used only for modeling with the MOF-based metamodeling hierarchy (modeling space defined by OMG, it does not

support n-layer metamodeling, like VMTS does). GME is the metamodeling tool from which VMTS has borrowed its base concepts.

Eclipse [5] is possibly the most popular, highly flexible, open source modeling platform that supports metamodeling. Eclipse is based on plugins, that grants the required flexibility. Eclipse Modeling Framework (EMF) can generate source code from models defined using the Class Diagram definition of UML[1]. EMF definitions contain the abstract syntax (the metamodel) only, the concrete syntax (the visualization) cannot be defined this way. The generated code contains base classes for editing the models, but the appearance is not customized. Graphical Editing Framework (GEF) is a part of the Eclipse project that provides methods for creating visual editors. EMF does not support code generation for GEF. Graphical Modeling Framework (GMF) is a new Eclipse project that is under validation. The goal of GMF is to form a generative bridge between EMF and GEF, whereby a diagram definition will be linked to a domain model as input to the generation of a visual editor.

The GenGed (Generation of Graphical Environments for Design) [6] environment is suitable for creating visual language definitions. It is rather presentation oriented: instead of metamodeling, it specifies graphical symbols, constraints and their connections; from this information, graph rewriting rules (Alphabet Rules) are generated, which serve as the graph grammar used to parse the visual language. GenGed uses AGG [6] as the internal graph transformation engine. For the editing features, a graphical editor is also generated to support the newly created visual languages. Transformation-Based Generation of Modeling Environments (TIGER) [7] is the successor of GenGed. It uses precise visual language (VL) definitions and offers a graphical environment based on GEF.

JKOGGE [8] is a tool for generating CASE tools. The tools built with JKOGGE consist of three parts: a base system, components, and documents. Documents are represented as distributed graphs. Components are realized with plugins that perform a well-defined task, e.g. editors.

Another framework is the Diagram Editor Generator (DiaGen) [9] that uses its own specification language for defining diagrams. The specification is checked and structurally analyzed using hypergraph transformations and grammars.

MetaEdit+ [10] offers a tool suite for defining a domain-specific modeling language with CASE support. The tool offers a full CASE support for the defined language, allowing developers to model using concepts that represent the product domain. MetaEdit+ allows viewing the design data in diagrams, tables and matrices. It offers an API for accessing components and enhances debugging.

3 Architecture

The VMTS Presentation Framework is based on the Model-View architectural design pattern [11]. The Model classes maintain the view-independent data and encapsulate the methods which operate on the data. The model elements themselves are handled by using the so called AGSI Interface. Each model object wraps the appropriate AGSI object. The AGSI Interface facilitates low-level model operations, while the Model classes provide high-level, domain-specific operations. There can be only one instance of Model classes for each item in our model. However, we may have several Views for the same Model object. The View objects are responsible for visualizing the model elements on the screen. The Views also receive and process user input events such as mouse clicks, mouse movements, key presses and all the general windows input events. Based on the user inputs, the appropriate modifications are performed on the Model object. The Model and View classes together realize the Observer pattern, consequently each View is immediately updated after the modification of the Model object.

The different aspects can be configured with the help of external plugins. The plugins are assigned to a specific metamodel through assembly level .NET attributes. Similarly, the metamodel-elements are assigned to the specific visualization with attributes. The architecture supports multiple views for the same model, and also supports the usage of different plugins for different views. The AGSI objects maintain two different fields for model-related and view-related information, these fields are referred to as PropertyXML and InfoXML fields. The InfoXML document of the model describes the views assigned to the model, including plugin and view identifiers. The InfoXML document of a model element describes the different views assigned to the element.

In Figure 2 the architecture of model-view classes is shown.

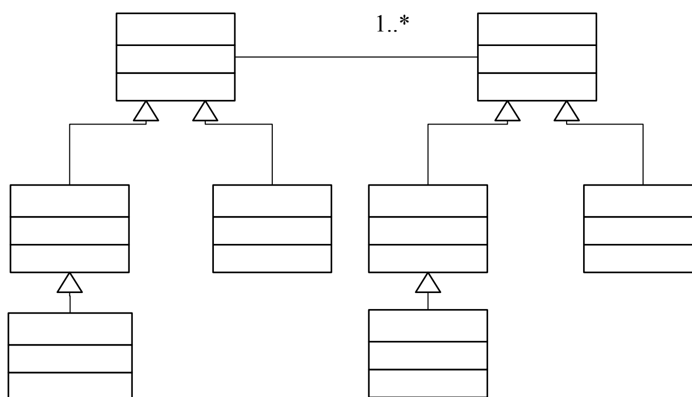

Figure 2
Model-View architecture

All model classes derive from the BaseModel class. Basically we distinguish two types of model elements: shapes and lines. Shapes represent nodes in the model, they can be the container of other model elements. Lines represent connections between two nodes in the model. The AGSI model itself is encapsulated by a DiagramModel object which is a ShapeModel descendant as DiagramModel provides all the features which are also provided by a Shape, e.g. containment of other elements, private coordinate system. We have also defined a View class for each of the basic Model classes. *ShapeView* and *LineView* classes are used to present shape and line-based model elements. The *DiagramView* class is used as a canvas, on which the elements of a model are placed. Each *DiagramView* instance has exactly one plugin assigned to. This plugin is used to instantiate the specific views for the model elements.

3.1 Model Classes

Default Model classes provide several indispensable features to handle model elements. Loading and saving element properties is performed with the help of these classes. Furthermore, the Model classes have the responsibility to instantiate the appropriate View class for an element. Custom model classes can be defined in external plugins as well. These plugins can be attached to a specific metamodel using .NET attributes. However, as we can have at most one Model object for each element in our model, we can define at most one custom Model class for each of the model elements. The plugin developer should take care of handling all the Views are handled by a common Model object.

3.2 View Classes

When visualizing a model, we do not have to strictly follow the containment hierarchy defined by AGSI and Model level. Instead, we have the possibility to skip several levels in the containment chain, e.g. we can place an element into the container of its model-level container. However, we do not have the possibility to contradict to the metamodel hierarchy: a contained object on model level cannot be the container of its model-level container in a view. As a consequence, the visual containment hierarchy is not necessarily the same as the model hierarchy. We can navigate towards the root element (*DiagramView*) using the HostShape property of the View in the visual tree, while – at the same time – the Container property is used to reach the container in the model (AGSI) tree.

In order to support the aforementioned features, we distinguish two types of model Views: *primary* and *secondary*. Changes performed on the primary view initiate the modification of the model hierarchy. Consequently, the primary view must always reflect the exact model level containment hierarchy. We can have at most one primary view and several secondary views.

Visual information is stored in the InfoXML field of an element. The same field is used for all the existing Views. Each *DiagramView* has a unique identifier, which is referenced by the View elements on the *DiagramView*.

Element views are based on windows controls. Their appearance is defined by the designer of the plugin. Since views are inherited from windows controls, all the element views have their own event handling, including mouse and other input events. We provide two possibilities to define the appearance of an element: (i) we can either derive a new class from the *ShapeView* (or *LineView*) class, where we have to define visualization and behavior using C#. (ii) We can describe visualization using XAML excusively. By using XAML and databinding, we can create a new domain specific plugin without manual coding and deep programming knowledge. We can use *Control Templates* and *Styles* to completely redefine the appearance of a plugin element. Model dependent visualization can be performed using the built-in databinding features of .NET.

If there is no appropriate plugin for a model, a default plugin (the so called Abstract Syntax Plugin) is applied. The Abstract Syntax plugin provides the basic features to visualize essential properties and containment. In Figure 3 the same model can be seen presented with both a plugin of a control flow language and the abstract syntax plugin.

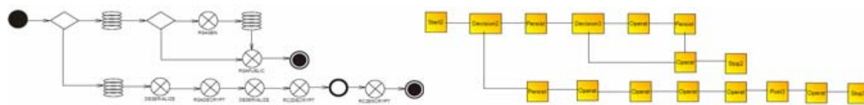

Figure 3
A model visualized with and without a plugin

3.2.1 Workspaces

Numerous modeling languages support the visualization of containment relationship between elements. Probably the simplest example is the well-known UML Statechart (Fig. 4) diagram. States can behave as composite states which means that they contain several other states. The containment is possible in an arbitrary depth.

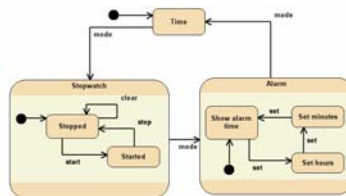

Figure 4
UML Statechart diagram

In other cases, it may be necessary to arrange the contained elements based on a predefined strict order. To visualize a program or protocol stack (Fig. 5), the levels of the stack should be arranged uniformly from the bottom towards the top of the stack.

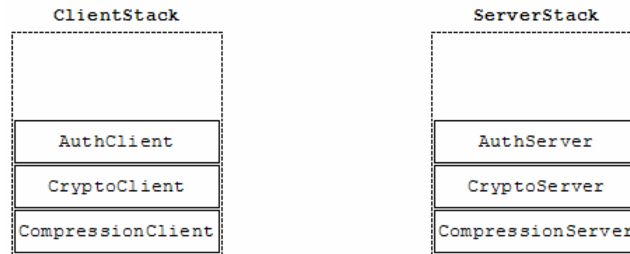

Figure 5
Protocol stack plugin

To fullfill the presented requirements, we have introduced a general component, called *Workspace* which serves as a container for other View objects. Each *ShapeView* descendant object may contain an arbitrary number of *Workspace*s providing thus visual separation of contained element, although, this kind of separation is not present in the model hierarchy. Each *Workspace* has its own coordinate system including offset, rotation and zoom. Furthermore, *Workspace*s can apply custom layout and order contained items automatically as well. The class hierarchy of the *Workspace* and the connecting classes is illustrated in Fig. 6.

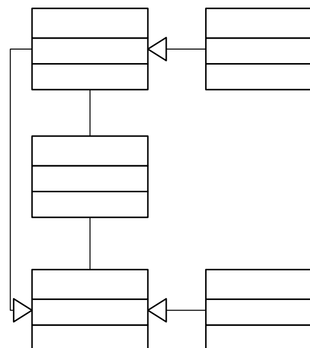

Figure 6
Container and contained classes of *Workspace*

3.3 The Framework

The *Framework* is a singleton object which provides a connection point between the Model-View layer and the modeler application (Adaptive Modeler). The different views, windows of the application has to be updated after the editing of the model elements. Subscribing the appropriate updater methods of the application to the events of each model object would result in a significant performance loss. *Framework* behaves as an event proxy, but from the applications point of view the Framework is an event source. These events are fired when the model hierarchy or relevant properties of the model change. From the Model-View layers point of view the *Framework* provides notifier methods which are called during changes in the Model-View layer. As a reaction, the notifier methods initiate the firing of the relevant events.

3.4 Persistence

When designing the persistence of View classes, the two main goals were being independent from the database representation format and providing a declarative solution to define persistable data. To simplify the work of plugin developers, we have introduced the *Externalizable* .NET attribute. Object level fields and properties marked with this attribute are automatically serialized (deserialized) during the save (load) process. Custom serialization can be performed by overriding the Externalize and Internalize methods of View classes.

The serialization process consists of two parts: (i) A generic store (*ExternalizerStore*) is filled with the externalizable fields and properties of the objects. ExternalizerStores contain tag-value pairs. The value can be of several built-in types (including numeric types, strings and basic geometric types like *Point* and *Vector*), an other ExternalizerStore or an arbitrary user object which implements the IExternalizable interface. If the value is of the type *ExternalizerStore*, we can build a tree of tag-value pairs. (ii) The *ExternalizerStore* is transformed into the final data-representation format using a format specific *Externalizer* implementation. We support currently a default Xml Externalizer only.

4 Environment

4.1 Browser Windows

In addition to the possible Diagram Views, model hierarchy and model elements can also be inspected using the Model Browser and the View Browser windows. These views present the model-based and the view-based containment hierarchy.

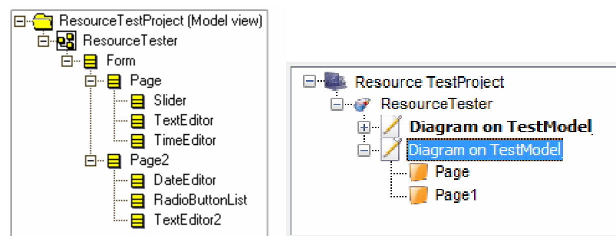

Figure 7
Model and View browser windows

The Property Grid is used to provide editing facility for the properties of the View objects. The Property Grid always visualizes the common properties of all selected elements. This feature is achieved by the runtime generation of a composite object, which has exactly the properties which are contained by all the selected objects, and forwards any events to the wrapped objects. With the help of the wrapper object we are also able to insert the appropriate Undo commands right before editing a property.

4.2 Undo-Redo

Actions performed by the user should be able to be undone and redone. For this purpose, we use the fusion of the Command and the Composite design patterns [11]. Any changes in the model are encapsulated into a *Command* object which contains all the necessary information to roll back the changes or commit them again. The *Command* objects are collected in an Undo and a Redo stack, so we can always undo and redo the last operation.

There are cases when several Commands should be threatened as a single user event. For example deleting a View on the Primary View indicates the deletion of the Model object (and the AGSI object) as well. Two separate *Command* objects are placed onto the stack this case, but we would like to roll them back in one step of course. For this purpose we have extended the *Command* objects so that they can contain several other commands and undo/redo them in one step.

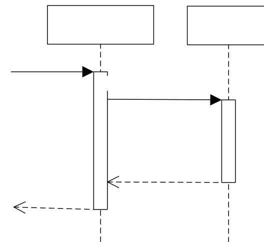

Figure 8
Processing composite undo-commands

Conclusions

The main ideas and architectural decisions made by designing VPF were presented in the paper. We have separated the model and the view logic, the separation has taken place at class level as well. Several views can be maintained for a model. The views do not necessarily reflect the proper model containment hierarchy, but can emphasize an aspect of the model. The actual model hierarchy can be seen in the Model Browser window and in the Primary View. The Primary View is also used to edit the model, however, Secondary Views are only used for visualization. Views can be fully customized with plugins which can be configured either using traditional C# code and class inheritance, or with the help of XAML documents. Object properties are serialized with the help of a powerful externalization mechanism. The externalizer engine is independent from the target data format, but it is flexible enough to process arbitrary objects.

Future work includes automatic element placing and ordering, furthermore an intelligent line routing method is required in connecting two shapes.

References

[1] UML – Unified modeling language: http://www.omg.org/uml

[2] Visual Modeling and Transformation System: http://vmts.aut.bme.hu

[3] XAML - Extensible Application Markup Language: http://msdn2.microsoft.com/en-us/library/ms752059.aspx

[4] Lédeczi Á, Bakay Á, Maróti M, Vőlgyesi P, Nordstrom G, Sprinkle J, Karsai G: Composing Domain-Specific Design Environments, *IEEE Computer 34(11), November,* 2001, pp. 44-51

[5] Graphical Editing Framework, http://www.eclipse.org/gef/

[6] Taentzer G: AGG: A Graph Transformation Environment for Modeling and Validation of Software, In J. Pfaltz, M. Nagl, and B. Boehlen (eds.),

Application of Graph Transformations with Industrial Relevance (AGTIVE'03), volume 3062, Springer LNCS, 2004

[7] Erhig, K., Ermel, C., Hansgen, S., Taentzer, G.: Generation of Visual Editors as Eclipse Plug-Ins, http://www.tfs.cs.tu-berlin.de/~tigerprj/papers/

[8] JKOGGE http://www.uni-koblenz.de/FB4/Institutes/IST/AGEbert/Projects/MetaCase

[9] Minas M.: Specifying Graph-like diagrams with DIAGEN, Science of Computer Programming 44:157–180, 2002

[10] MetaEdit+, http://www.metacase.com/

[11] Gamma E., Helm R., Johnson R., Vlissides J.: Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional Computing Series