Technical Solutions for the Transformation-Driven Graphical Tool Building Platform METAclipse

Oskars Vilītis[1], Audris Kalniņš

Institute of Mathematics and Computer Science, University of Latvia, 29 Raiņa blvd., Rīga, LV-1459, Latvia, ph.: (+371) 6 7224 363

Oskars.Vilitis@gmail.com, Audris.Kalnins@mii.lu.lv

Abstract. The paper gives a detailed description of technical solutions developed for the implementation of a metamodel-based graphical tool building platform whose main area of application is the development of DSL editors. As opposed to the well-known static-mapping-driven approach, the implementation described here provides more flexible means for the definition of the correspondence between the domain and presentation metamodels, using model transformations. The solutions described in the paper form the basis of a newly developed Eclipse plugin METAclipse that allows an easy use of transformations and materializes the ideas of the transformation-driven tool building platform. METAclipse has proven its flexibility and efficiency in the development of a new generation graphical editor for the model transformation language MOLA.

Keywords. DSL Editors, model transformations, metamodel-based graphical tool building platform, transformation-driven, Eclipse

1 Introduction

Due to the increasing interest in the MDA approach and the growing popularity of various domain-specific languages (DSLs), various graphical tool building environments have gained continuously increasing attention in recent years. The first simple generic metamodel-based tool environments, such as MetaEdit [1], Kogge [2] and early versions of Dome [3] and [4], appeared already in the mid-nineties, but their capabilities were quite limited.

The second generation of such metamodel-based environments with much wider possibilities, such as MetaEdit+ [5], GME [6], and ATOM3 [7], appeared around 2000 (the first version of MetaEdit+ actually appeared much earlier [8]). They already had domain metamodeling facilities close to MOF [9] and more advanced graphical capabilities. Therefore the popular tool paradigm of a visual language being based on a presentation-independent domain (as it is e.g., for UML [10]) could be supported. But the presentation metamodel (the description of graphical elements) still had to be close to the domain metamodel, only relatively simple mappings between them were

[1] supported partially by ESF (European Social Fund), project 2004/0001/VPD1/ESF/PIAA/04/NP/3.2.3.1/0001/0001/0063

permitted, and everything else had to be defined by OOPL code (e.g., C++ in GME). The previous tool framework by UL IMCS, the Generic Modeling Tool environment [11], also belongs to this category.

A completely new generation of tool frameworks has emerged in recent years in response to the need of the MDA community to make DSLs an everyday software development practice. One such group of environments is based on the open-source Eclipse platform. Eclipse, together with its EMF plugin [12], is a broadly used metamodeling environment, close to MOF. In addition, the GEF plugin [13] is a basic "diagram drawing engine." Only something linking the two was required for a complete tool building environment. The first and the most popular solution is the static metamodel mapping-driven GMF platform [14]. Alternative solutions are provided by the Pounamu/Marama [15] environment and the coming GEMS project [16].

A popular alternative to Eclipse on a commercial basis is offered by Microsoft DSL Tools [17] in Visual Studio 2005; however, the logical capabilities there are quite close to GMF. The already mentioned MetaEdit+ has significantly evolved and has also become a key player in this area.

The above-mentioned solutions are quite appropriate for relatively simple cases, where the domain and presentation metamodels are close and no complicated mapping logic is required. However, DSL support frequently requires much more complicated and flexible mapping logic. Therefore a new approach has appeared: to define this mapping by model transformation languages. Model mappings in tools actually lie very close to the traditional MDA tasks, for which model transformation languages were invented. Therefore they can be considered very appropriate DSLs for metamodel-based tool building, yielding development efficiency that is an order of magnitude higher when compared to that of OOPL.

The first frameworks using this approach to a degree are the Tiger project [18] and the ViatraDSM framework [19]. Both are based on Eclipse and use GEF as a drawing engine. The Tiger project is based on the graph transformation language AGG [20]. However, a specific domain modeling notation is used there, which still forces the domain metamodel of a language to be close to the presentation metamodel. Standard editing actions (create, delete, etc.) are specified by graph transformations, which act on the domain model, and the presentation model is updated accordingly. The ViatraDSM framework is based on the Viatra2 transformation language [21]. In this framework, the domain metamodel must be close to the presentation metamodel too, but larger freedom is allowed, and the transformation approach can, to a degree, be combined with the static mapping approach. There are also plans to use the Fujaba [22] transformation language in the MOFLON framework [23].

A detailed analysis of the two approaches and their strengths and weaknesses has been done in the paper "Building Tools by Model Transformations in Eclipse" [24]. This paper concentrates on a thorough description of the technical solutions developed in order to implement the fully transformation-driven tool building platform METAclipse. METAclipse is partly being developed within the project "New Generation Modeling Tool Framework," which is funded by ERDF (2006–2008). Within this project, another tool implementing similar ideas, GrTP [25], is also being developed, however with a different profile: its aim is to handle various tasks related to the semantic web.

In METAclipse there are no restrictions on the correspondence between the domain and presentation metamodels. The mappings are defined dynamically by transformations in the model transformation language MOLA [26]. METAclipse is implemented as an Eclipse plugin and reuses the basic Eclipse components such as EMF and GEF, as well as parts of the GMF runtime [14]. METAclipse obeys traditional Eclipse style and behavior guidelines and therefore can be integrated in other eclipse-based development environments. Also, it is possible to integrate other Eclipse technologies like model-to-text generation. An overview of the platforms architecture and the rationale behind the METAclipse framework will be presented in section 2.

The main distinguishing feature of METAclipse is an appropriately built presentation metamodel, which is discussed in detail in section 4. It enables a clear separation of responsibilities between the METAclipse presentation engines, which handle all the low-level presentation and layout-related tasks, and transformations, which create and maintain only the domain and the logical structure of presentation.

Section 5 provides a brief sketch of transformations in METAclipse, however, they are not the main topic of this paper. The emphasis of this paper is on the structure and functionality of the METAclipse framework itself.

METAclipse is already proven to be useful in practice by creating an editor for the MOLA language itself (MOLA is a graphical model transformation language, thus being a remarkable example of a DSL). This editor is successfully being used in the European IST 6th framework project ReDSeeDS [27]. All figures containing class diagrams in the paper have been created with the MOLA metamodel editor.

2 Overall METAclipse Architecture

A graphical modeling tool must deal with many complex tasks, such as proper domain element representation; intuitive and standardized element editing; correct model modifications in response to the graphical editing events; providing a convenient way of navigating through models and a clear way of model element property representation; etc. The most complex and time consuming tasks are the ones concerned with the graphical representation and user interface handling. Luckily, a number of these tasks are common to all graphical tools (i.e., they are domain-independent) and can be handled at the tool-building platform framework level.

2.1 Basic Principles of the METAclipse Framework

In METAclipse, a well-defined framework is provided for the tool builders. The top-level view of the METAclipse architecture is very simple (see Fig. 1). METAclipse itself consists of a set of Eclipse plugins that define the framework of the tool building platform and comprise several so-called presentation engines, each of which deals with a particular set of graphical editor tasks (project tree engine, element property engine, etc.). Each of these engines will be discussed later in this paper in section 4.

METAclipse plugins contain all the common functionality needed for the tools and relieves the creator of the tool from the need to worry about many technical user interface issues. The part that defines a concrete tool and that must be written by the toolsmith is the transformation library containing all the necessary model transformations that change the model according to the user actions in the tool.

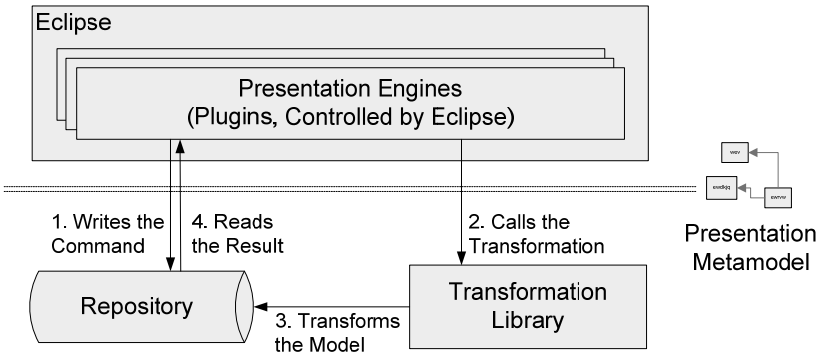

Fig. 1. High-Level view of the METAclipse architecture

In METAclipse the toolsmith must start with the creation of the domain metamodel and proceed with wiring the domain metamodel to the presentation metamodel through writing the model transformations. Thus the only items the toolsmith builds for a concrete tool are the domain metamodel and the transformation library defining the functionality. In the paper the combined metamodel of presentation and domain metamodels will be referred to as METAclipse metamodel. Accordingly, the combination of domain and presentation models will be called simply model. Manipulations with the domain model are completely the responsibility of the transformation writer. METAclipse framework provides no support for the domain model modifications.

Every framework engine exposes its features to the transformations through a strictly defined metamodel that serve as an interface between the transformations and editors. Metamodels of the engines will be discussed in more detail in later sections of this paper. Part of each engine's metamodel is also the available set of commands that could occur as a result of user actions. Commands are used to trigger the transformations and a single command instance represents one atomic user action, which constitutes the smallest piece of work in the framework. All actions that make purely graphical changes are handled directly by METAclipse framework. Only semantic actions (actions causing domain model changes or any changes in the presentation model that are specific to a concrete tool) are transformed into the commands and passed to the transformations for execution.

Together metamodels of all engines form the presentation metamodel of METAclipse. Each element displayed in the tool, created using METAclipse, corresponds to a presentation model element (an instance of some presentation metamodel class). Presentation model as well as domain model (model on which the tool actually operates) are stored in the model repository and are changed by the

transformations as a reaction to the user triggered events. Every semantic user action in METAclipse results in the following sequence of actions:

- The presentation engine that gets some user action writes the command corresponding to the action taken (right click on a project tree node, creation of an element, drawing a link between elements, etc.) to the model repository and invokes the main transformation (steps 1 and 2 in Fig. 1);
- The main model transformation recognizes the command written and makes the necessary changes to the presentation and/or domain models (step 3 in Fig. 1);
- Presentation engines read the model changes and react accordingly: show context menu, show newly created element or edge, etc. (step 4 in Fig. 1).

Such top-level view of METAclipse architecture can be compared to the traditional MVC approach: the role of the controller is played by transformations; the repository serves as the model, and the presentation engines act as the view. It must be noted that METAclipse leverages the abstraction level of the MVC approach: the controller (transformations) receives only the semantic actions.

In order to make the METAclipse architecture and functionality more clear, an example state of the project tree engine is given in Fig. 2. A visual representation of the project tree engine is given on the left. In the middle, a part of the simplified project tree engine metamodel is shown. Here one can see how the visual editor elements are represented to the transformations: ProjectTreeNode class represents one node in the project tree. The ShowMenuCommand class represents a right-click event on the tree node and expresses user request to show the context menu.

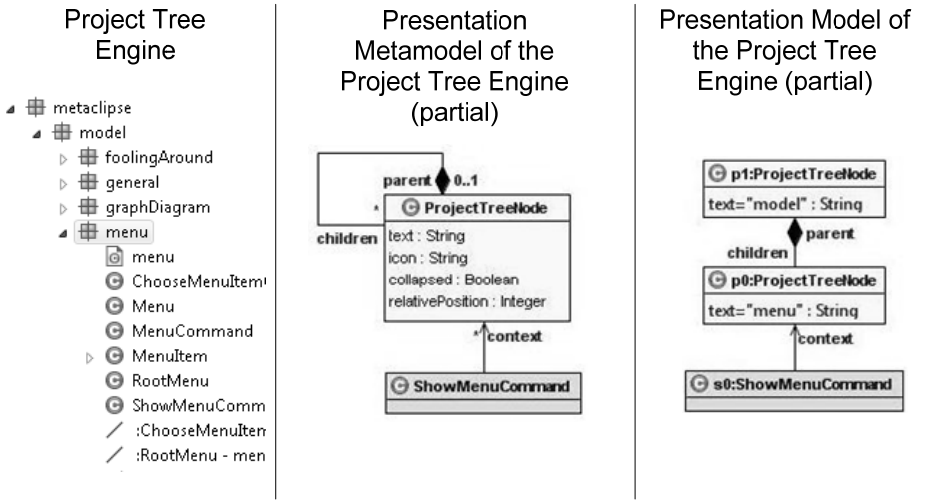

Fig. 2. Example of a project tree engine and its metamodel and model states

Let us imagine that one has right-clicked the node called “menu” in the tree and the project tree engine has written the ShowMenuCommand instance to the repository (step 1 in Fig. 1). At the right side of Fig. 2 the presentation model part is given, showing the instances involved in the handling of the right-click event. As the next step in event processing, the engine will invoke the transformation (step 2 in Fig. 1).

The transformation will find that ShowMenuCommand has been written in the repository and will create presentation metamodel instances (not shown in the Fig. 2) comprising the needed context menu (step 3 in Fig. 1). No domain model changes are needed in this example. At last, Eclipse will get back the control and presentation engines will be notified of the model elements changed. The menu engine will see that a menu has been created, so it will show the context menu for the project tree node called "menu."

2.2 Solutions Chosen for the METAclipse Implementation

METAclipse is built on top of Eclipse technologies and is packaged in the form of several Eclipse plugins. Eclipse was chosen as a mature and widely appreciated platform, providing a large number of frameworks covering many needs of the tool developers. Eclipse is also a very popular choice of a wide variety of leading production-quality software development platforms that could potentially gain from integration of modeling and DSL editor tools.

The transformation language MOLA [26, 28], developed by LU IMCS, was chosen for the implementation of transformations. MOLA has a rich set of language elements and it had already proven its performance and stability in practice, so it was a natural choice. The current implementation of MOLA is compiled to a Windows DLL file and works against the repository MIIREP (codenamed "OUR" in the paper "Towards Semantic Latvia" [29]), also developed by LU IMCS. Therefore, the choice of the repository was also clear. However, to make METAclipse more flexible, it was decided to make the access to transformations and the repository transparent so that it would be possible to switch to other transformation languages and/or repositories. The repository access solution will be described in Section 3.

As discussed in the previous section, every METAclipse presentation engine exposes its features to the transformations through its metamodel. What is actually displayed in the editor is a visual representation of the engine metamodel instances, i.e., models. In Eclipse, Java code needs to access this model information. To accomplish this, physical in-memory model storage is needed. The framework fitting these purposes already exists and is called EMF [12]—Eclipse Modeling Framework. EMF is being used in many Eclipse-based tool building platforms as the model repository.

EMF was also chosen for implementation of the METAclipse model repository, as it has several features that fit the framework needs. EMF provides a generator for the creation of Java classes that correspond to the model elements. This eases the creation of the runtime model classes. Another important feature of EMF is the model change notification mechanism implemented through model listeners that allow easy and dynamic model change transfer to various presentation engine parts. There are also some aspects of the EMF that are currently less important for METAclipse, which however could turn useful in time: XMI import/export, OCL implementation, etc.

This leads to the presentation model in METAclipse being stored in the EMF repository. Transformations, however, also need to operate on this model. As transformations work on an external repository, a challenge rises to synchronize the

EMF model instances with information in the MIIREP. More details on the non-trivial solution will be given in Section 3.

The EMF framework is not the only Eclipse framework used in METAclipse. For various METAclipse needs, others are used as well:

- The property engine uses the tabbed properties framework for dynamic generation of the element property sheets (see Section 4.5 for detailed description of the property engine);
- The project tree engine (described in Section 4.3) uses the navigator framework;
- The graph diagram engine (described in Section 4.6) uses the Graphical Editing Framework GEF [13] and parts of the Graphical Modeling Framework GMF [14] runtime.

3 Interaction with the Repository and Transformations

As already stated before, editor interaction with the repository and transformation invocation was intended to be made as generic as possible in order to maintain the possibility to change the implementation of repository or transformations if necessary. To achieve such independence, two problems had to be solved. First of all, an interface to the set of external repository operations used in METAclipse (such as find object, store object, change object property etc.) had to be defined. Transformation invocation is also part of this interface, as transformations are always related to a particular repository. Secondly, a generic mechanism to transfer the repository data to EMF object instances had to be developed in order to allow the handling of repository objects in Eclipse as if they were normal EMF objects, thus giving the access to the entire infrastructure provided by EMF.

3.1 Repository Interface

The repository interface itself is nothing special; it is a regular Java interface containing all the operations required by METAclipse. The interface contains the following sets of operations:

- Metamodel (object type) manipulations, such as creating a class, adding a class attribute, finding classes, creating associations, etc.
- Model (object) manipulations, such as finding an object of a certain class, creating objects and setting object attributes, etc.
- Transformation invocation. Only one function for this is required, as transformations have just one entry point in the METAclipse architecture.

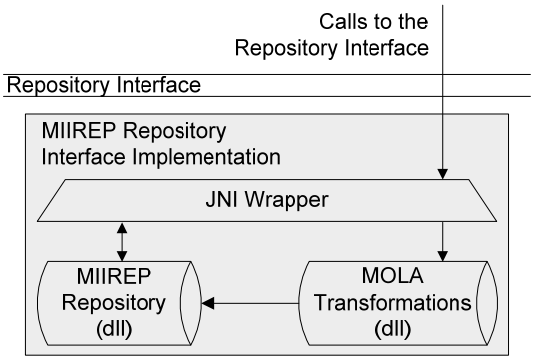

Fig. 3. MIIREP repository interface implementation

MOLA transformations currently are compiled against the MIIREP repository, which is developed in C++ and released as a Windows DLL file. MOLA transformations themselves are also compiled to a DLL file, which directly accesses the MIIREP DLL loaded in memory. This implies that the MIIREP repository interface implementation currently used in METAclipse (see Fig. 3) uses a JNI (Java Native Interface) wrapper for the repository operations (see [30] for information on JNI). The wrapper delegates all repository access operations (model and metamodel manipulations) to the appropriate MIIREP repository API functions and the invocation of transformations to the transformation library.

3.2 The Link Between Eclipse and the Repository: "Wise" Objects

As stated before, all presentation engines (Eclipse plugins) developed work with EMF runtime objects in order to gain all the benefits the EMF framework is offering. Transformations, on the other hand, work with the external repository, so synchronization between the repository and EMF is required.

The task of integrating the external repository seamlessly into the Eclipse EMF framework was quite challenging. Simple interface did not satisfy the requirement to keep Java-side code unaware that anything other than EMF is used, which is why the "wise" object mechanism was created. The main reason for such a requirement was the wish to keep the possibility to switch to a clean EMF implementation in the future (meaning that no external repository would be used, with EMF itself serving as the repository), as well as to be able to use clean EMF infrastructure.

Another aspect that had to be taken into account was performance. As every little action in the editor results in changes in the repository through the invocation of the transformation, a complete re-read of all repository data after each operation is unacceptable. Only the "dirty" or changed information has to be transferred back to EMF object instances.

To comply with the given requirements, a special mechanism was developed, consisting of alternative EMF runtime objects that conform to the EMF interfaces and externally look like normal EMF objects, but internally do all the synchronization with the repository. These objects were named "wise" objects, as they show certain

“intelligence”: though from the interface perspective they look like normal EMF objects and support all EMF framework operations, internally they know when and how it is necessary to read or write some information to the repository. The standard EMF notification mechanism is used to notify any changes occurring in the repository. “Wise” objects can be considered the second level of repository abstraction, which introduces the caching mechanism, conforms to the EMF object interfaces and uses first level abstraction—repository interface—to read and write data to the repository.

ECore, the core metamodel in EMF, is very similar to the EMOF (Essential MOF), a subset of the MOF model [9]. In fact, there are just some small, mostly notational, differences between these two. According to the MOF hierarchy, ECore is at the M3 layer, the same as MOF itself. The code generation facility provided by EMF can be used to generate Java runtime classes for a particular metamodel (M2 layer) defined by ECore. Instances of the generated runtime classes then correspond to the M1 layer in MOF.

ECore metamodel classes (ECore base classes) define the class hierarchy that forms the basis for the Java runtime. All EMF runtime classes generated for a particular metamodel extend these base classes. ECore base classes provide all the functionality to the generated classes and allow using them in EMF infrastructure by providing all the EMF framework features. Therefore, base classes are the best place where the repository synchronization should be implemented.

“Wise” Objects as an EMF Extension

Base ECore classes were extended and a set of “wise” object base classes was defined (see Fig. 4). By analogy to ECore classes, base “wise” object classes, together with some helper classes comprising the whole “wise” object concept, were called WCore. In WCore, the methods inherited from ECore for getting and setting the properties are extended with functionality of reading and writing data from and to the repository through the repository interface described in the previous section. For performance considerations, “Wise” objects keep track of the state of every object property and cache the data from the repository in the object instance, so the consequent reads of the same property will result only in one read of the property from the repository.

The fact that the parent of all ECore classes is a single class—EObject (see [12] for complete ECore structure)—simplified the extension of ECore. For “wise” object needs it was enough to extend just two ECore classes, EObject and EFactory, with the corresponding WObject and WFactory classes. WObject contains all the caching and synchronization logic and, as it is the superclass of all the other framework classes, the logic is available all across the framework. The WFactory extension of the factory class was needed, as some initialization of the “wise” object on its creation was required.

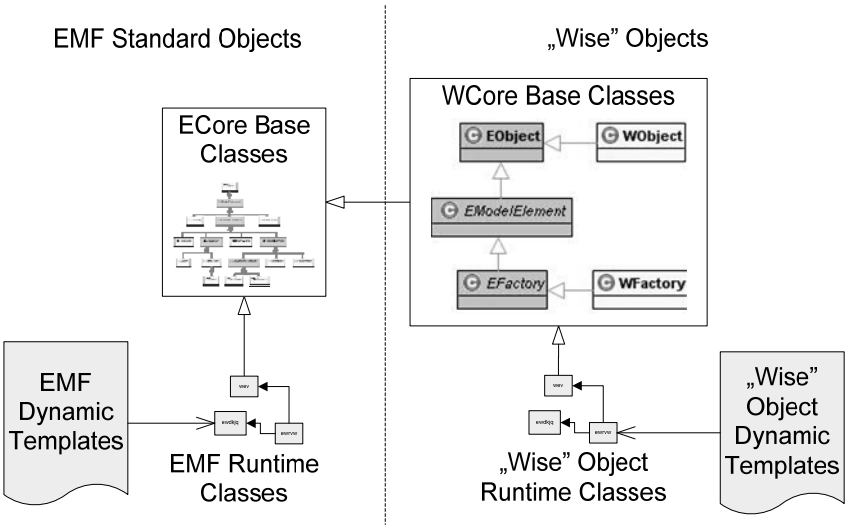

Fig. 4. "Wise" object dependencies

To put the WCore classes in action, the EMF generator also had to be extended so that it produced "wise" objects extending WCore base classes. The EMF framework uses the so-called dynamic code templates (using another Eclipse framework for the code generation—JET [31]) during the generation process of the runtime classes. The EMF generator reads the serialized form of the metamodel and then, using the set of templates, generates the runtime classes (see Fig. 4). Default templates producing EMF runtime classes were extended so that they would generate the code using WCore instead of ECore.

The complete set of classes comprising the WCore can be seen in Fig. 5. The above-mentioned extension of getter and setter methods of ECore is divided into two classes. Reading of the attributes from the repository was easiest to implement in the WObjectImpl class itself, in the inherited getter methods. Writing the attributes, however, was easier to move to a separate class WObjectChangeObserver, which implements the EMF change listener and is attached to every instance of WObject. The change observer listens to any changes done to the WObject from the engine side and if any occurs, writes the data to the repository.

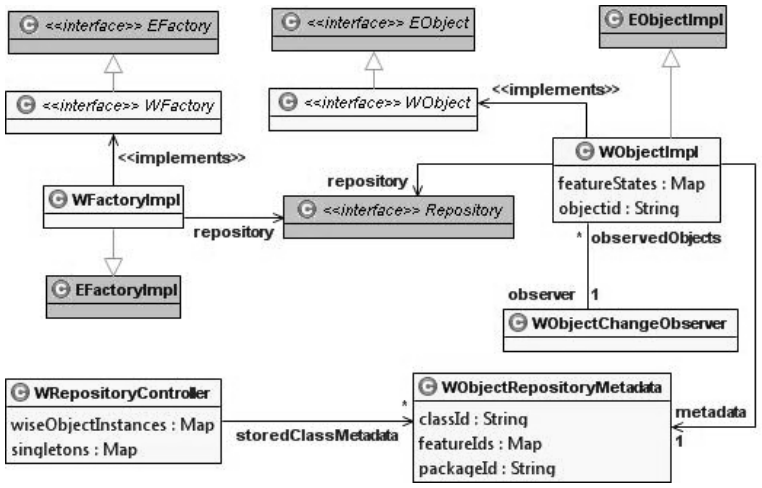

Fig. 5. WCore class diagram

To be able to read and write the repository data, "wise" objects need to have a possibility to map the classes, attributes and associations to the corresponding repository objects. Such mapping can be defined only at M2 level and thus it is necessary to have the WCore class and feature mapping to the repository metadata at the M2 layer. As it is inefficient to read these mappings every time any object is accessed, class metadata mappings are cached. The WRepositoryMetadata object represents the class metadata. The map of WCore class to repository metadata mappings is held in the WRepositoryController object and the mappings are attached to every WObject instance for convenience when instantiating it (as a reference to the cached mappings).

Repository Change Notification in METAclipse

Extending the ECore base classes covers the synchronization needs only from the METAclipse perspective, i.e., if changes to the model are done from the engines. However, the most intense model changes happen on the other side—in the transformations. Therefore, another missing piece is a change notifier back from the transformation, which would trigger the EMF change events for all objects that have been changed in the repository. In WCore, the WRepositoryController class takes care of this. There, a special method is defined for change detection, which has to be invoked after each transformation execution.

Transformation change notification is not a trivial task, as it is also constrained with tight performance requirements. It is very inefficient to detect the changes already after transformation execution, as it means inspection of all object instances in the repository. This means that a support from the side of the transformations is required in order to make an efficient implementation of the change notification. This is why WRepositoryController change notification method is designed in a way that it calls special functions of the repository interface in order to get the lists of the changed or deleted objects. Functionality of tracking changes is left to the

implementation of the interface. When changed or deleted object lists are read from the repository, WRepositoryController issues the corresponding EMF notifications and the changed features of the object instances that have changed are set "dirty," so that they are once again read from the repository instead of using the cached values from the WObject instances. The concept of "wise" objects is not trivial and is best understood on a concrete example. One such example is provided in Section 4.2.

In case of the repository and transformations currently used in METAclipse, it was very easy to track object deletions, as the MIIREP repository itself has the functionality to track such changes. However, the tracking of the changes to the existing objects had to be incorporated in the transformations. For this reason, a special class "Changes" was introduced in the presentation metamodel. Each transformation is responsible not only for making the actual changes, but also for adding a link from the "Changes" singleton object to the objects actually changed. See Section 4.1 for more information on the "Changes" object and the singleton concept.

Of course, it would be more convenient to have also detection of changes to the existing objects automated and incorporated at the repository level, but unfortunately MIIREP does not provide such a possibility. In case of MOLA, as its transformations are compiled, it is also possible to add special functionality in the MOLA compiler that automatically adds the "Changes" link. However, at the moment such functionality is not implemented.

4 Presentation Engines

As already stated before, METAclipse consists of several presentation engines. Although there are some additional smaller helper parts in METAclipse, four main presentation engines can be named that together comprise the whole tool building platform (in Fig. 6 all of them can be seen in action).

1. Project tree engine, responsible for organization of projects, models and model elements in a hierarchical tree structure;
2. Graph diagram engine: the main engine of METAclipse, providing editing capabilities to the graph diagrams;
3. Property engine: provides property editing capabilities for other engines (like properties for a selected item in the project tree or a selected diagram element);
4. Menu engine: used by other engines for the displaying of context menus (like by project tree engine for showing context menus of the tree nodes or by graph diagram engine for showing context menus on the diagram elements).

Besides these four engines, additionally there are some less important components in METAclipse responsible for common functionality like drag-and-drop, clipboard, METAclipse perspective; utility functions; transformation control etc. These will not be discussed here. In the following sections the focus will be put on the interaction between the engines and transformations, and special attention will be paid to the description of all the presentation metamodels, as they form one of the most important aspects describing the METAclipse functionality.

The look and feel and general operation principles in METAclipse engines were adopted from Eclipse standard editors so that the editors would fit smoothly in the Eclipse environment. This means that some eclipse standards were obeyed. For example, METAclipse does not use dialogs for the diagram element creation. Instead, all element properties are assigned default values, which can later be changed to the desired values through the properties view. Properties are displayed in one single view for all editors, implying that just one editor is in focus at all times.

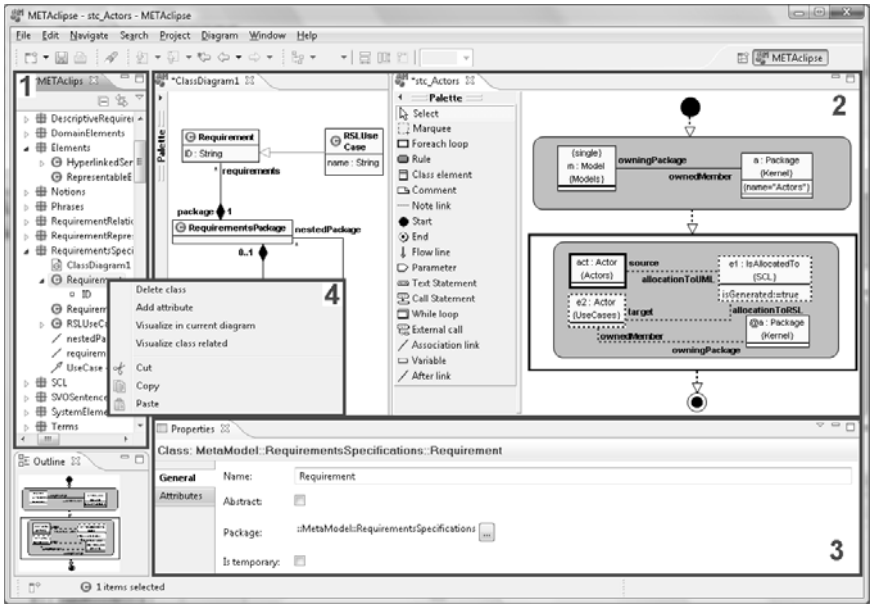

Fig. 6. METAclipse presentation engines in action

In the development of the presentation engines, one simple rule drove the splitting of functionality between the engine and transformations:

- Every task that needs any information read from the domain model, i.e., that is domain-specific, has to be done by transformation;
- All tasks that do not require any knowledge of the domain have to be done by the engines.

So, for example, the right click on the project tree node for showing the context menu needs the knowledge of what kind of node it is in order to know what menu options to offer. This means that this is a task for a transformation. Another example—the move of a diagram element within the borders of the same parent—does not require any knowledge of the domain. Such operation requires only changing of some presentation model attributes, thus it can be carried out by the engine itself. If, in contrast, the diagram element was dragged out of the borders of the parent element (for example, dragged from one sub-diagram to another), this again would require some domain model changes and thus it is a semantic operation that has to be performed by transformations.

4.1 Presentation Metamodel Structure

The transformation library is the changing part in METAclipse from one tool implementation to other. That is why transformation creation must be made as easy as possible in order to make METAclipse useful and convenient for the toolsmiths. In order to accomplish this there are several prerequisites to be met:

- A well-established set of base transformations common to all or at least most editors must be provided. This would form the base framework for transformations to be created by the toolsmith. This would allow the toolsmith to concentrate on semantic tasks for mapping of domain elements to presentation elements and would remove the need to worry about some tasks that could be done by the framework (for example, handling of the element styles, parts of copy and paste logic, building of standard menus, etc.);
- A set of helper transformations must be provided, so that the transformation creator has decent artillery at hand for handling of different kind of tasks (utility functions);
- It is very important to create a good interface to the presentation engines. In this case engine metamodels compose this interface. A proper presentation metamodel is extremely important for the transformation creators to make work with the editors easy and convenient.

A very short overview on the solutions provided by METAclipse for the first two will be given in Section 5. The focus in this paper however is on the last—proper design of the presentation metamodel. A large amount of effort and time was invested in the design of this metamodel to make it best usable from transformations. The following few sections will give a thorough description of various parts of it, i.e., of various presentation engine metamodels.

The presentation engines rely heavily on various Eclipse frameworks. Therefore, the metamodels of the engines could be partially extracted from them. It must be noted, however, that none of the used Eclipse frameworks had a metamodel already defined. Metamodel of every engine had to be synthesized from the corresponding framework API. Then it had to be amended with the METAclipse-specific classes needed for the engine.

As the metamodel is an interface between two parties, transformations and Java code, it has to be conveniently usable from both sides. However, more importance must be given to the transformation requirements for the metamodel. It was decided to adopt the naming and structuring standards of classes from the Java coding standards, keeping in mind not to make any transformation tasks complicated. As it turned out, it is very convenient for both sides if the metamodel is structured in strictly hierarchical and logically split packages. The whole presentation model contains the following packages:

- the *general* package contents include the base classes used by the presentation metamodel, classes common to all engines and various types used across the presentation metamodel;

- the *project* package contains all the classes needed for project handling in METAclipse and classes for steering the project tree engine (see section 4.3 for the description);
- the *menu* package contains classes for steering the menu engine (see section 4.4 for the description);
- the *properties* package contains classes for steering the properties engine (see section 4.5 for the description);
- the *graphDiagram* package contains classes for steering the graph diagram engine, excluding the classes for palette organization (see section 4.6 for the description);
- the *palette* package contains classes for creation of the editor palettes. This was created as a separate package, because palettes may be required not only by graph diagrams. Palette elements could be reused also if another kind of editor engine were created.

The Common Part of the Presentation Metamodel (*general* Package)

The *general* package defines the core classes of the METAclipse presentation metamodel (see Fig. 7). In this and following figures a special color-coding will be used. Normal metamodel classes will be shown in white. Light gray color will represent the command classes. For more information on what a command is, see Sections 2.1 and 4.1, as well as descriptions of METAclipse presentation engines. The dark gray classes will denote the singletons. The description of the term "singleton" is given below.

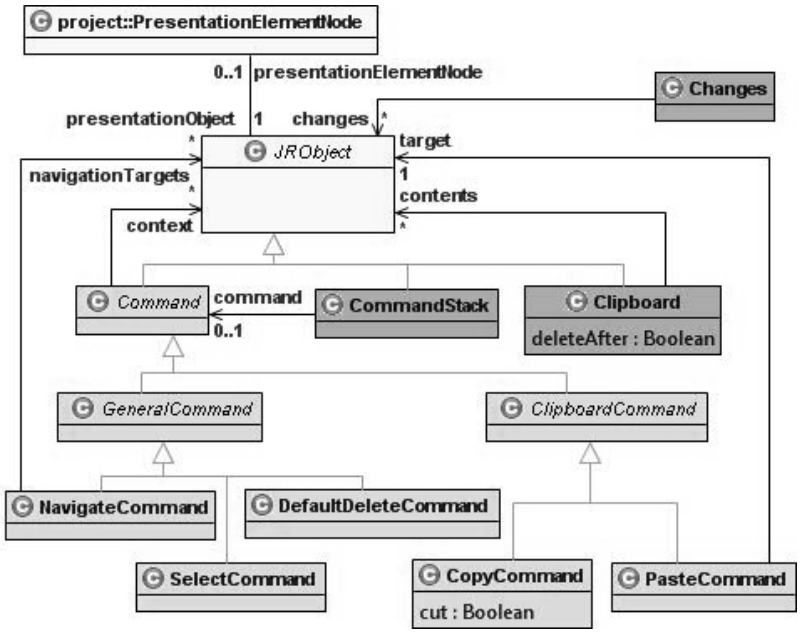

Fig. 7. The general part of the presentation metamodel

As the metamodeling practice shows, and also as the preliminary experience of METAclipse technology evaluation proved, it is very convenient to have one superclass for all classes in the metamodel and to organize all classes in strict hierarchies. Just as Java has a superclass of all classes, "Object", the METAclipse presentation metamodel also includes such a superclass, JRObject. One example of how the introduction of such a superclass helps is the case when there is a need to define a very general association to any kind of object. This can be done only if there is a superclass for every object needed to be referenced. In the *general* package this is used to model the concept that any presentation model element can be displayed in the project tree engine as a node: association between PresentationElementNode and JRObject (see Fig. 7).

A concept used across all metamodels by engines for finding the starting points of various parts of models is singletons. Singletons are classes that have exactly one instance. This fact is used by the presentation engines to find the only instance just by knowing the class name. Singleton classes are used in METAclipse engines everywhere where there is a need for an entry point in the model. In the *general* package one example of singletons is the Changes class. This class is an important singleton, which is used to find all the changed or deleted objects after the execution of a transformation.

As discussed in Section 3.2, for wise objects to work there is a need for change tracking after each transformation invocation. Current implementation of the MIIREP repository and MOLA transformations does allow automatic tracking of deletions; however changes must be tracked by each transformation manually. The Changes singleton instance must be linked through "changes" association to every presentation model object changed by the transformation. Engines will then use the singleton nature of the Changes class to find the only instance and read the list of the changed model objects.

The *general* package also contains the supporting and base classes for one of the backbones of METAclipse, namely, the command infrastructure. Commands have already been discussed before. A command in a presentation metamodel corresponds to a possible user action in the editor that requires some reaction from the engine, i.e., the invocation of a transformation. Command class in the metamodel is the superclass for all the command classes. Command base class defines the "context" association: every command can have links to some JRObject instances that form the context of the command. All commands are structured in a strict class hierarchy: for every logical set of commands, an additional superclass is defined (as GeneralCommand and ClipboardCommand in Fig. 7). This opens diverse command parsing possibilities in transformations.

The sequence of command execution in METAclipse is described in Section 2.1. After any user action, a corresponding command is written to the repository. CommandStack singleton instance is linked to the written command. Transformations then seek the command to execute by querying the "command" link of the CommandStack singleton. Currently this link points to at most one instance of a command. After execution, the transformation may write back some results to the executed command by setting some attributes or links. Finally, engines read the command after the transformation execution in order to get the transformation results, if needed.

The rest of the *general* package classes shown in Fig. 7 are common classes used by many presentation engines. This includes some common command classes and the clipboard-supporting classes. NavigateCommand is used as a response to double-clicking on some project tree node or diagram element. Such action would result in opening a diagram in the editor and possibly selecting some diagram element (or multiple elements), if the element under the cursor were a diagram or diagram element. Transformations must return the diagram to open or diagram elements to select by setting the navigationTargets link. It will be queried by the engines after the execution of the transformation to find the objects to open / select.

SelectCommand is executed if any object is selected. It must be used by transformations to generate the property sheets corresponding to the selected object. See section 4.5 for more information about the properties engine. Command DefaultDeleteCommand is executed if the delete button is pressed on any of the selected objects. As the name suggests, transformations should carry out the default delete action when processing this command. Such a command is especially useful for diagrams—usually it is possible to delete an element from the diagram while retaining the domain element or to delete both the diagram and the model element. Different tools require different default logic on such operation.

For clipboard operations, the Clipboard singleton and two commands for copying and pasting are defined. The Clipboard singleton contains links to the copied or cut objects (through "contents" association); the deleteAfter flag is used to distinguish the copy and cut operations. Copy command is executed when the selection is copied. Selected objects are linked to the command through the "context" association. Paste command is executed when users executes the paste operation in the engines.

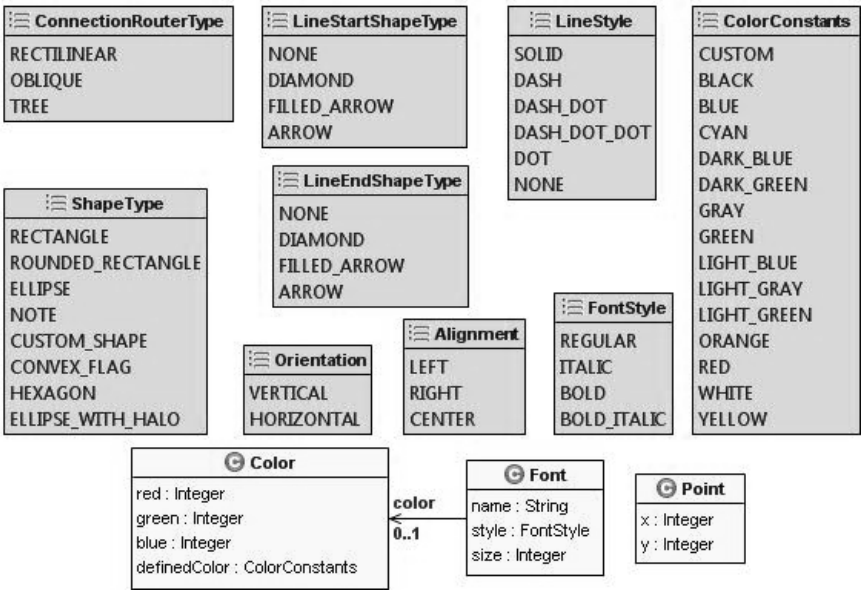

Fig. 8. General type part of the presentation metamodel

Finally, the last set of classes found in the *general* package consists of the various types used across the entire METAclipse presentation metamodel. These include definitions of enumerations like Alignment, ShapeType, Orientation, etc., as well as some type classes like Color, Font and Point.

4.2 Interaction between the Transformations and Engines

The mechanism of the interaction between the engines and transformations has already been outlined. Now, as all the concepts of the components involved in METAclipse (engines, wise objects, repository, transformations and presentation metamodel) have been introduced, it is time to put it all together. This section will give an example of how all of the METAclipse components fit together before proceeding to the detailed descriptions of the separate engines in the sections to follow. See Fig. 9 for a detailed operation schema of the opening of a new diagram from the project tree. Solid lines in the figure represent the control flow; dashed lines, simple operations like creation of objects.

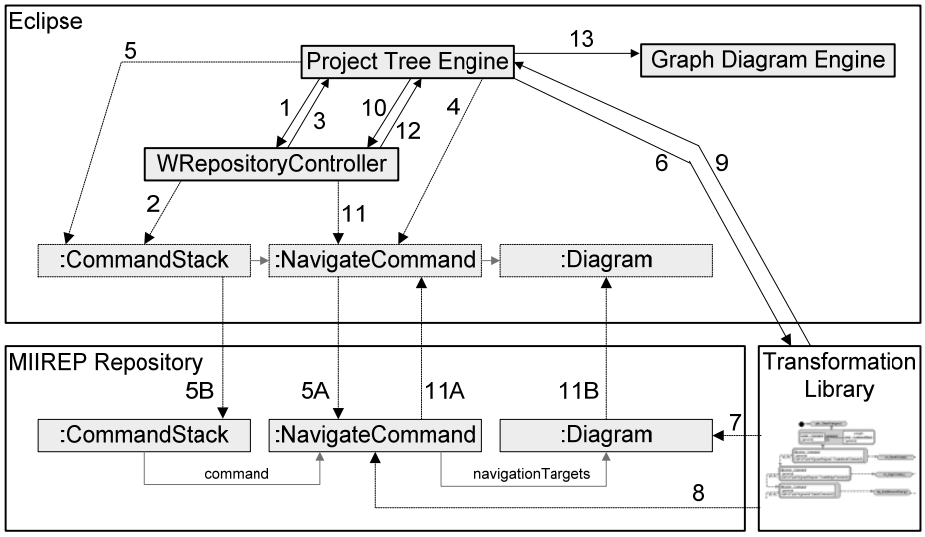

Fig. 9. Opening a new diagram from the project tree:
an example of the METAclipse component interaction

Let us imagine that a user has double-clicked a node in the project tree that represents a graph diagram. This results in invocation of the project tree engine (discussed in more detail in section 4.3). This engine must react so that a corresponding diagram is opened. Such operation includes the following steps:

- 1: The project tree engine asks WRepositoryController to find the singleton instance of the CommandStack class (see previous section for information about singletons, repository controller, and command stack).

- 2: If this is the first time CommandStack singleton is used, WRepositoryController searches the repository for the single instance of the class with the name "CommandStack." As it is a singleton, there will be exactly one instance. The repository controller loads the CommandStack wise object instance and caches it, so that the next time the CommandStack is queried, it would be retrieved from the cache.
- 3: The CommandStack wise object is returned to the project tree engine.
- 4: The project tree engine creates a new instance of NavigateCommand wise object and links it to the project tree node wise object, on which the double-click was performed (not shown in the figure). As the NavigateCommand has not been yet saved to the repository, for the time being no synchronization with repository is carried out.
- 5: The project tree engine links the newly created command to the CommandStack. At this moment CommandStack wise object notices that a new link has occurred. As the linked object is not yet saved to the repository, it asks the NavigateCommand instance to save itself to the repository (5A). Then the CommandStack wise object links the repository instance of CommandStack to the newly created instance of NavigateCommand (5B).
- 6: Now, when the command is written to the repository, the transformation library is invoked.
- 7: Transformation detects the NavigateCommand instance linked to the CommandStack and finds which project tree node was double-clicked. Then it searches for the corresponding diagram to be opened.
- 8: Transformation links the Diagram instance found to the NavigateCommand as a result of the execution. Additionally, it puts a link from the Changes singleton (see previous section) to the NavigateCommand in order to signal that NavigateCommand instance has changed.
- 9: Control is given back to the project tree engine.
- 10: The project tree engine calls the WRepositoryController in order to invoke the repository change notification process and synchronize the wise object state with the repository.
- 11: WRepositoryController reads the Changes singleton to detect that the wise object instance of NavigateCommand has changed. It then notifies the NavigateCommand wise object that it must read its contents from the repository instead of its cached data (11A). This also causes the instantiation of the linked Diagram object (11B).
- 12: Control is given back to the project tree engine.
- 13: Finally, the project tree engine delegates control to the graph diagram engine and passes the Diagram wise object to be displayed. Graph diagram engine then uses the Diagram object as the root for reading all the contents to be displayed on the diagram.

All engines operate similarly and the wise object technology is used throughout all METAclipse for synchronization with the repository. This ensures consistent interaction with the transformations. It must be noted that only one transformation at a time can be executed. This, however, does not cause any problems, because in the graphical editors the user makes just one action at a time and actions are sequential.

We could continue describing property generation for the element that is currently selected. However, the operations for that would be very similar to the ones already described. The only additional operation for building of the properties would be the querying and modification of the domain model. This, however, is hidden from the METAclipse framework, as only transformations are responsible for the operations with it and only transformations can access the domain model.

4.3 Project Tree Engine

Every graphical tool needs some means of organizing the model objects in a hierarchical tree structure to enable the navigation through models—similarly to how files and folders are organized on the computer hard drive. At the minimum, it is required to display the diagrams as a list, so that the user could choose the one he/she desires to edit.

Eclipse defines the notion of "project" as the highest level of organization. Different tools built on Eclipse provide different kinds of projects: for example, Java, C++, GMF and others. METAclipse also defines a separate kind of project, the METAclipse project. A METAclipse project corresponds to one repository instance, which is created together with the project. All elements of the project model are stored in this repository, e.g., if there are several diagrams in one METAclipse project, they all will be stored in the same repository instance.

For organization of project artifacts, Eclipse provides the so-called navigator framework, which provides a view for displaying of items in a tree. The METAclipse project tree engine is built using this framework and implements its own view (see Fig. 6, part 1). The Eclipse navigator framework already provides all the functionality required to manage the project tree. The only thing needed to implement a specific project tree is the implementation of Navigator interfaces for the retrieving of the model data (or the so-called provider-interfaces, which is a concept used also in other Eclipse frameworks). This is an easy task, as the interfaces require an implementation of a few very simple methods like one for getting the children of a given node and another for getting the parent of a given node. METAclipse provides the implementations of these interfaces for reading the project tree data from the repository. This implementation was very easy to create: just about 100 LOC was required, which was clearly less than would be needed if all functionality had to be created from scratch.

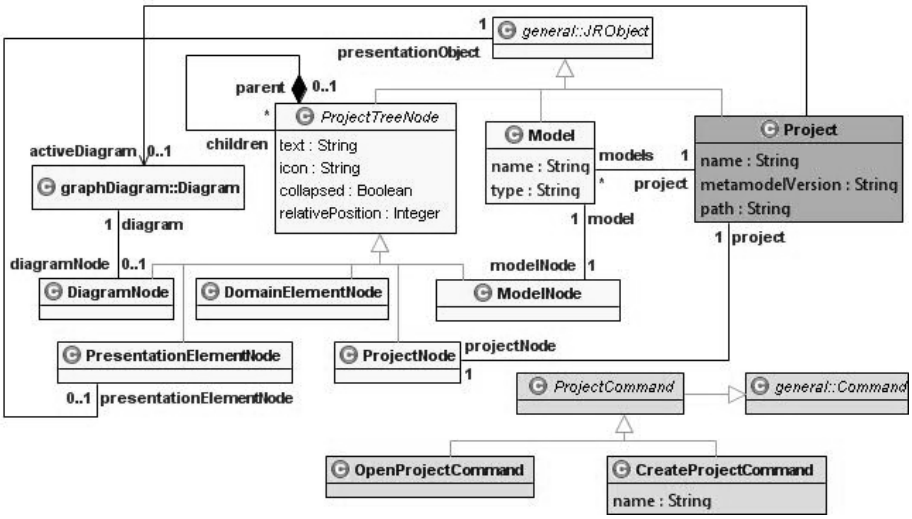

Fig. 10. Project part of the presentation metamodel

Fig. 10 shows the metamodel of the project tree engine. When a METAclipse project is opened, first the Project singleton is used to find the ProjectNode instance, which then is interpreted as the root of the project tree. Every METAclipse project will always have exactly one ProjectNode. Project is a singleton that represents the METAclipse project opened in the platform (recall that there is one-to-one correspondence between a METAclipse project and a repository instance).

The ProjectTreeNode class is the superclass of all kinds of tree nodes, ProjectNode included. This class allows defining the hierarchical structure of the tree through the parent-children association. Every instance of one of its subclasses will appear in the project tree engine as a separate node with the given text and icon and ordered by the relativePosition. Transformations are free to define any kind of project tree structures, using the ProjectTreeNode building blocks. There are five kinds of nodes at their disposal, each with a slightly different support from the engine's side:

- *ProjectNode*. Interpreted by the engine as the root project node;
- *ModelNode*. Interpreted as the node defining the boundaries of one model. The model term is introduced to allow further grouping of project items in smaller pieces of work. On possible use of the ModelNode and Model classes could be for the demarcation of the nodes that correspond to the packages in the domain or, if the domain metamodel provides the term of model (like UML domain model [10]), to the models;
- *DiagramNode*. Interpreted as a node that can be opened and represents a diagram. Transformations must make sure that tree nodes of this kind are linked to a corresponding Diagram instance;
- *PresentationElementNode***.** Interpreted as a node that represents some diagram presentation element. Can be used for navigation;
- *DomainElementNode*. Interpreted as a node that corresponds to an element from the domain model.

ProjectTreeNode is the only class that represents the original metamodel of the Navigator framework according to its API. METAclipse project tree engine also does not really need all the various subclasses of the ProjectTreeNode. The subclasses have been introduced in order to ease the creation of the transformations.

There are only two commands specific to the project tree engine that can occur. One is CreateProjectCommand, which is invoked when a METAclipse project is created. It must be interpreted by transformations to initialize the models with some startup data—for example, to initialize the singletons, to set up the default context menus and property editors, to initialize styles, etc. Second is OpenProjectCommand, which is invoked when the project is opened in METAclipse. It can be interpreted by the transformations to carry out some initialization routines required for the opening of the project.

4.4 Menu Engine

The menu engine is the simplest engine of all and provides just the functionality needed for the creation of context menus (see Fig. 6, part 2). It uses the standard Eclipse infrastructure for the generation of the menus. Therefore the implementation of the menu engine in METAclipse was even easier than the implementation of the project tree engine.

The menu engine metamodel defines one singleton class, RootMenu (see Fig. 11), which points to the root of the active menu through the "menu" association. If the RootMenu instance does not have this property set, it means that no menu will be displayed. Menu structure is defined by the Menu and MenuItem classes. The Menu class is interpreted by the engine as a menu container (like the root of the context menu or any submenu popping out when an item containing the submenu is selected). Menu consists of menu MenuItem classes, which correspond to the items displayed in the menu. Submenus are shown by the engine only for those MenuItem instances that have the submenu property set.

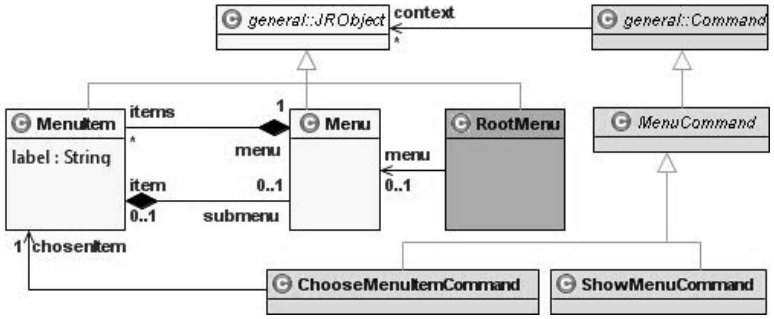

Fig. 11. Menu part of the presentation metamodel

Only two specific commands can occur in the menu engine. ShowMenuCommand is invoked when the user right-clicks any node in the project tree or any element in the diagram. Selected JRObject instances (whether tree nodes or diagram elements)

will be linked to the ShowMenuCommand through the context association defined in the general Command class. Transformations must react to this command by building the context-sensitive menu (using the context information from the context association) and setting the RootMenu singleton "menu" association to it. The menu engine then will consult the RootMenu singleton to read the menu to be shown.

ChooseMenuItemCommand is written to the repository if the user chooses an item from the menu. Then transformations must carry out the corresponding action. Action can be anything necessary for the chosen menu item, starting from creation of some element up to very complicated tasks like model simplification, compiler invocation for visual DSL languages and so on.

4.5 Properties Engine

A very important part of the tools is the properties editor. This editor is used to display and edit various properties of elements displayed in editors. For example, in the UML class diagram editor there is a need to edit the properties of a class or association. In Eclipse property editing is done through a special properties view, which is common to all editors and can be seen at all times (see Fig. 6, part 3). Any time the selection in Eclipse changes, the contents of the properties view are also updated to reflect the properties of the currently selected item. Properties can be arranged in the so-called tabs for better structuring.

The properties view is driven by yet another Eclipse framework, the tabbed properties framework [32], which is used by the properties engine of METAclipse. When the development of METAclipse began, the tabbed properties framework did not provide all the capabilities needed for the tool building platform. Particularly, it was not possible to define the structure of the property sheets at runtime. The framework allowed only definition of what should be displayed in the property sheets during the time of development, and this information had to be compiled in the released plugins.

Because of this, in the beginning the tabbed properties framework was extended to add this functionality. Later, however, the functionality of the framework was widened to include the possibility to define the property sheets dynamically at runtime. This allowed switching to a clean tabbed properties framework without the need to extend its classes. Tabbed properties with dynamic property support will be released in Eclipse 3.4 M3, which is not yet available at the time of this writing. However, Eclipse 3.4 M2 nightly builds already include the new dynamic tabbed property capabilities, so this is what is being used for the time being.

General Part of the Properties metamodel

In METAclipse transformations are responsible for building of the property sheets. The select command is issued by editors so that transformations could carry out this task (already introduced in Section 4.1 and shown in Fig. 7). The main part of the property engine metamodel can be seen in Fig. 12.

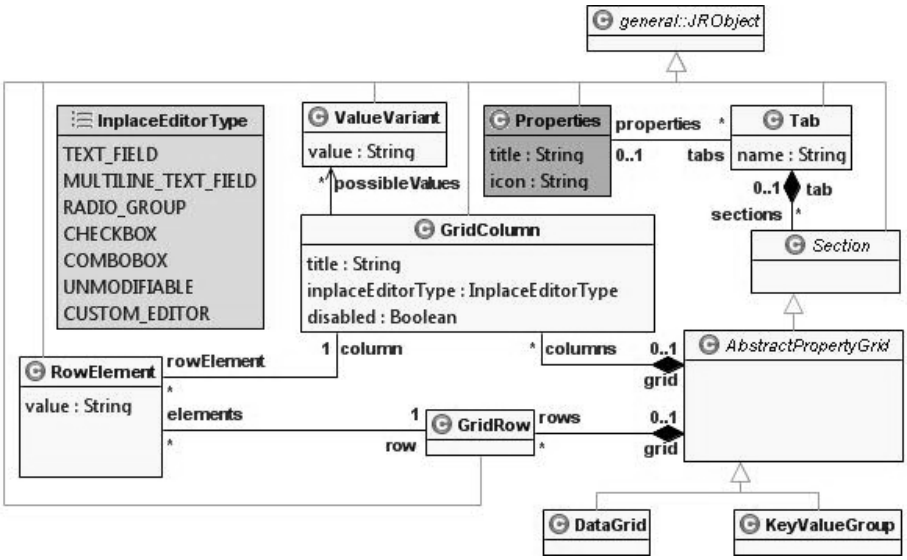

Fig. 12. Property part of the presentation metamodel: main classes

The properties singleton is queried every time after the selection of any element and execution of the SelectCommand to read the current state of the properties view. Through this singleton the whole structure describing the contents of the property page can be read. The title and icon attributes of the Properties singleton are used for the title of the properties view. The class Tab represents one property sheet tab and is linked to the Properties singleton through the "tabs" link. The attribute name is the title shown on the tab and is used to name the contents of the tab. For example, both properties views in Fig. 13 consist of three tabs: "General," "Attributes," and "Style."

Every tab in the tabbed properties framework consists of the so-called sections. Sections group the properties shown in the tab in logical groups. The corresponding class in the metamodel is the abstract Section class. The Tab class has a composite association with Section. As many section implementations as necessary could be provided in Eclipse. Two implementations turned out to be most useful in practice:

- A data grid that shows the properties in the form of a table with headers. Such a section can be used for the representation of properties that have one-to-many relationship with the element owning them. An example could be the list of attributes for a class in the UML class diagram (see Fig. 13, bottom);
- A group of key-value pairs that can be used for the representation of properties that have one-to-one relationship with the element owning them. An example application of this can be seen in Fig. 13, at the top, where the "General" tab of the class properties contains various values describing the class—such as "abstract" flag, name of the class, etc.

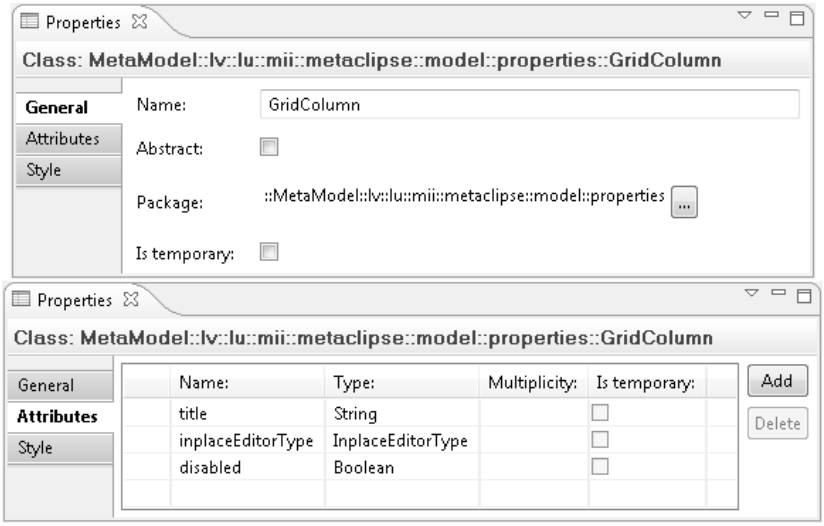

Fig. 13. KeyValueGroup properties section implementation (at the top) and DataGrid implementation (at the bottom) in action

These two kinds of section are implemented as part of the properties engine in METAclipse. DataGrid and KeyValueGroup classes in the metamodel (see Fig. 12) correspond to the data grid and key-value pair group section implementations, respectively. Both section implementations use the same metamodel structure for the description of their contents. This turned out to be particularly useful for the development of transformations, as it allowed a uniform design of the property-building transformations.

The structure used for the two section implementations consists of three main classes: GridColumn, GridRow, and RowElement. In case of the DataGrid section implementation, GridColumn corresponds to the table column. The title attribute will be shown as the header of the table. Attribute inplaceEditorType denotes the kind of editor that will be used for editing of the data found in the column. Possible values are defined by the InplaceEditorType enumeration and include such editors as text field, combo-box, checkbox, radio group etc. A special kind of editor is CUSTOM_EDITOR, which means that an external dialog has to be shown instead of in-place editor. This will be discussed in more detail below. For editing of the combo-box or radio group fields, additionally a set of possible values must be defined. This is done through the possibleValues association from the GridColumn class to the ValueVariant class.

The GridRow class corresponds to one row in the grid. The DataGrid class will hold an ordered reference to all row classes through “rows” association. Actual data of the table cells is represented by the RowElement class. The association “column” of this class will define what column the row element belongs to, while the association “row” will indicate in which row it should be displayed.

As stated before, the KeyValueGroup section implementation uses the same model. To understand how the structure is applied to the KeyValueGroup implementation, we can imagine that this implementation is nothing more than DataGrid with one row,

which is displayed vertically instead of horizontally. So, there will be exactly one GridRow instance and each GridColumn instance will correspond to the label of one key-value pair in the KeyValueGroup section (for example, “name” or “abstract” at the property view shown at the top of Fig. 13). RowElement instances correspond to the value part of key-value pairs, i.e., the values of the properties that can be edited.

Property Editors and Commands

Not all properties can be edited directly in the properties view—some require more advanced editing capabilities. For example, editing of a property denoting a color or a font requires a proper color dialog to be shown. Also properties that must be chosen from a list with lots of entries are inconvenient to be edited with a simple combo-box. The metamodel of the properties engine contains an additional set of classes for the definition of external editors (see Fig. 14).

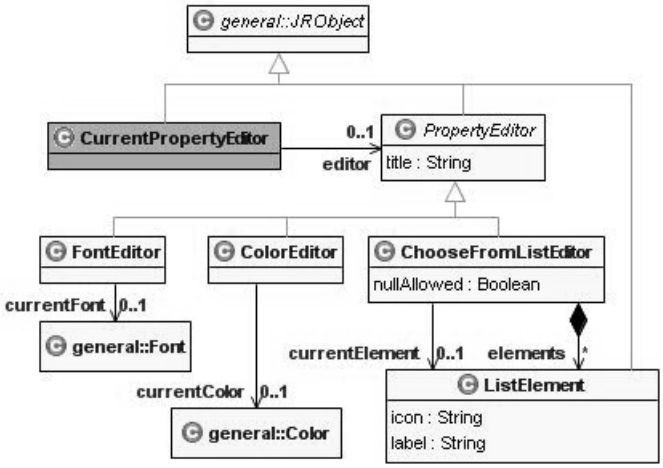

Fig. 14. Property part of the presentation metamodel: editor classes

Theoretically it would also be possible to create a universal dialog engine, so that any kind of dialogs could be constructed. However, it would require very large effort to build such engine. Therefore, it was decided to build concrete dialogs for different tasks. In the metamodel, a common superclass PropertyEditor is introduced for all dialogs. Three implementations are provided by the engine: the FontEditor class representing the font dialog, the ColorEditor class representing the color dialog and the ChooseFromListEditor representing the dialog for showing large lists.

If an external dialog is needed for a particular column, the inplaceEditorType attribute of the GridColumn instance must be set to CUSTOM_EDITOR. The engine will then display a button for invoking the external editor. If the button is pressed, the ShowEditorCommand (see Fig. 15) will be invoked and transformations will have to construct the dialog to be shown. The editor constructed then has to be linked to the CurrentPropertyEditor singleton, because the engine will consult this singleton to find which editor to show.

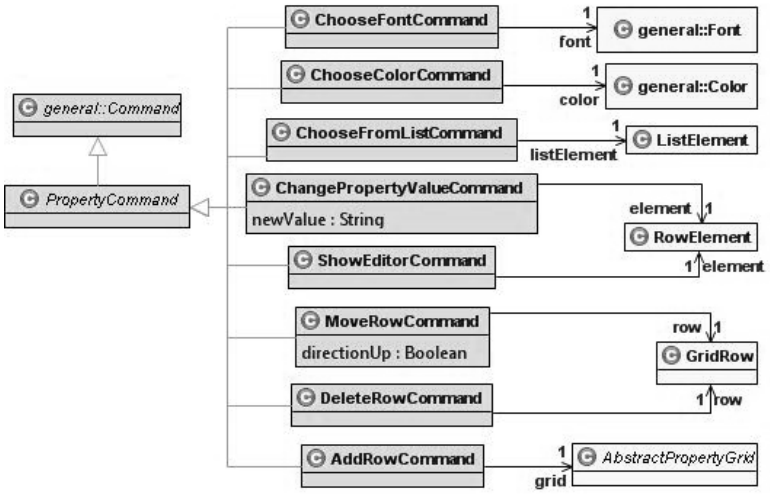

Fig. 15. Property part of the presentation metamodel: command classes

After showing the dialog and having the user choose something, the corresponding command is executed, containing the information about user actions in the dialog. Thus, for the font dialog, ChooseFontCommand is executed with the chosen font attached through the font association. Similarly, ChooseColorCommand is executed after choosing any color from the color dialog and ChooseFromListCommand, after choosing some list item from the list dialog.

The remaining commands not yet discussed are ChangePropertyValueCommand, which is invoked when any of in-place property editors is used to change the value of some property; MoveRowCommand, which is used to change the order of the DataGrid rows; DeleteRowComand, which deletes DataGrid rows; and AddRowCommand, which creates new DataGrid rows.

4.6 Graph Diagram Engine

The most important of all engines is the graph diagram engine. This engine is used for visual graph diagram editing (see Fig. 6, part 4). Eclipse technologies used for the graph diagram engine are the Graphical Editing Framework GEF [13] and the Graphical Modeling Framework GMF [14]. GMF is the most popular metamodel-based graphical tool building platform for Eclipse. GMF utilizes EMF (Eclipse Modeling Framework) and GEF (Graphical Editing Framework) technologies. EMF is used for model management and GEF, for graphical user interface.

GMF uses a static-mapping-driven approach. It defines a set of metamodels: graphical (presentation), tooling and mapping metamodels. In addition, it uses ECore as the domain metamodel. The graphical metamodel defines the graphical element types. The tooling metamodel defines the palette and menus. The mapping metamodel defines the mapping possibilities between the models. To build an editor in GMF, the

domain, graphical, tooling and mapping models are defined, then generation is performed and manual code in Java added. An analysis of the GMF and a comparison of the static-mapping-driven approach as such to the transformation-driven approach described here are given in the paper "Building Tools by Model Transformations in Eclipse" [24].

The graphical (presentation) metamodel is well adapted to the generation step in GMF, but cannot be used directly by the transformation approach. The same situation is true for the tooling metamodel. Therefore, nothing of the GMF definition part can actually be reused in the proposed METAclipse approach. As a consequence, there are no explicit graphical element types to be used by transformations.

Fortunately, the GMF runtime [34] uses another metamodel—the notation metamodel. This metamodel describes graphical instances in the runtime—nodes, edges, compartments and labels (exactly, the layer required by transformations to build graphical objects dynamically). In fact, the GMF runtime is a graphical engine for Eclipse, significantly extending GEF in the direction required for diagram building. This allows at least partial reuse of the GMF runtime in METAclipse.

The created graph diagram engine does not fall back from professional Eclipse-based tools like RSA [35] in its diversity of features and graphical quality. The developed metamodel, presented further, allows relatively simple control of quite advanced graphical structures and behavior. Although the graph diagram engine was the most difficult to implement, the reuse of GMF runtime and GEF components allowed keeping the required effort for building it reasonably low.

The General Part of the Graph Diagram Engines Metamodel

The main part of the graph diagram engines metamodel in METAclipse is quite similar to the GMF notation metamodel. However, it is not the same. It has been made more accessible for the transformations and more easily usable in various contexts of METAclipse (see Fig. 16).

The root element corresponding to the actual diagram is the Diagram class. It consists of DiagramElement class instances, which can be either Node or Edge. Node class instances correspond to the graph diagram nodes and Edge instances correspond to edges. Note that Diagram itself is also a kind of node. This allows the use of sub-diagrams. The Diagram element defines the general attributes of all elements, such as line style and width. Node defines the general attributes of all kinds of nodes. The Edge class defines the routing of the edges via the routing style attribute and association with Bendpoint instances. Routing style defines how the line should be laid out on the diagram and Bendpoint instances define the layout constraints.

Besides Diagram itself, the nodes are divided into two categories—SimpleNode and CompositeNode. SimpleNode denotes the nodes that may not contain any children. CompositeNode, on the other hand, may contain children. Theoretically, Diagram also is a composite node. However, because of its specific nature, it is not in the class hierarchy of the composite nodes.

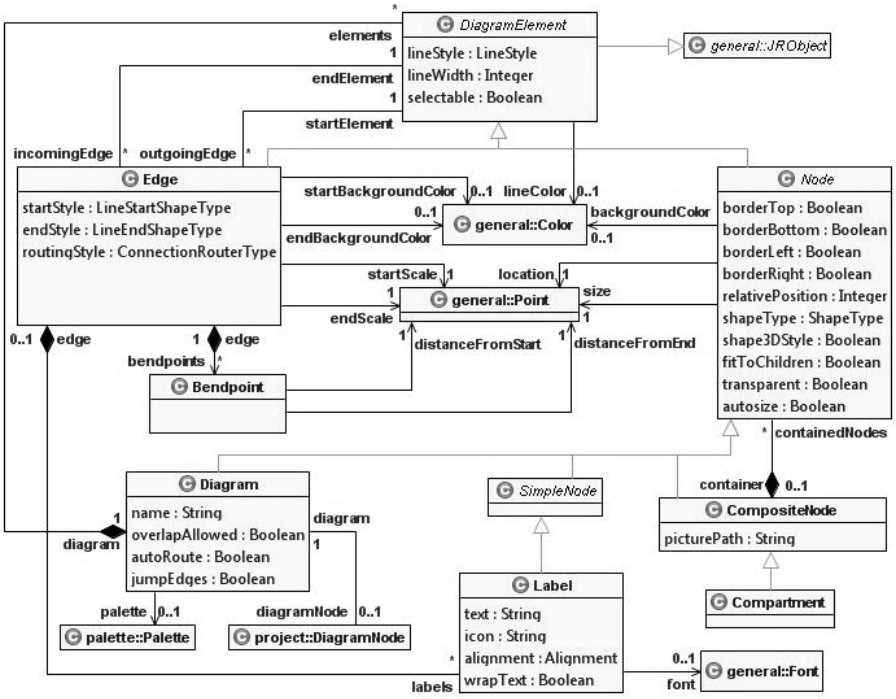

Fig. 16. Graph diagram part of the presentation metamodel without commands and palette

There is just one kind of SimpleNode type—the Label class. Labels are static text elements that may also display an icon. CompositeNode is not abstract, thus it may be instantiated itself, but there is also one special type of the composite node, i.e. Compartment. Compartment is a kind of grouping, used, for example for class diagrams in UML [10].

Just as an example, let us consider the UML class Diagram (like the one in Fig. 16). Diagram itself is represented with the Diagram class instance. It consists of CompositeNode-s, which in turn consist of one label for class icon and name, one compartment with labels for attributes, and one compartment with labels for operations (operations not shown in the figure). Associations are edges with different sets of attribute values for different kinds of associations. These are the bricks for building class diagrams in the METAclipse framework.

In Fig. 17 the command part of the graph diagram engines metamodel is shown. There are just four commands specific to the graph diagram engine:

- CreateEdgeCommand, used for creation of the edges;
- CreateNodeCommand, used for the creation of the nodes;
- MoveNodeCommand, used for the semantic moving of the nodes (in case the node is dropped in another node, for example);
- RedirectEdgeCommand, used for relocating the edge start or end to a different node.

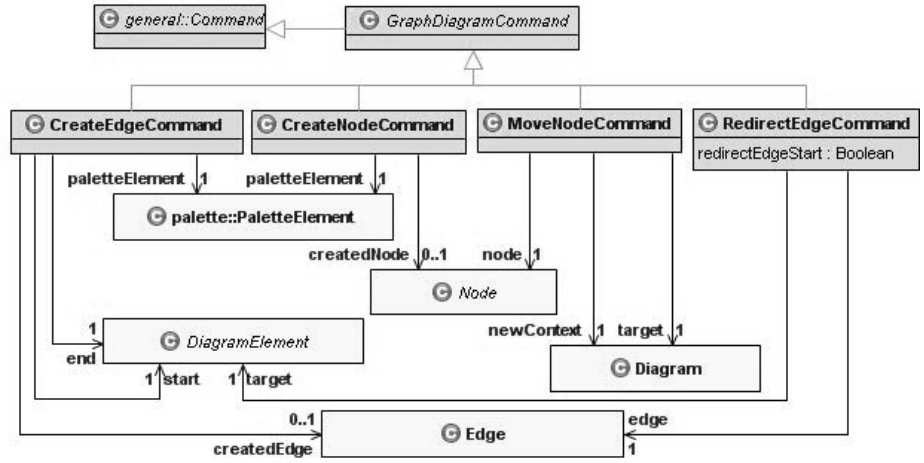

Fig. 17. Graph diagram command part of the presentation metamodel

Additionally, there are some already discussed common commands accessible in graph diagram engine, like NavigateCommand, SelectCommand, etc. These are used for the tasks that are common to more than just one METAclipse engine.

Palette Part of the Graph Diagram Engines Metamodel

The metamodel for description of the palettes has been separated from the graph diagram metamodel as it could be reused also for other diagram kinds. Fig. 18 shows the palette part of the graph diagram engines metamodel.

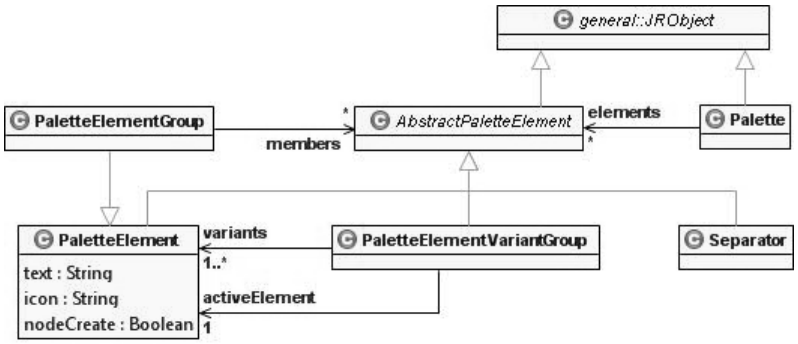

Fig. 18. Palette part of the presentation metamodel

The structure of the palette metamodel represents the possibilities to build palette in Eclipse. The Palette class represents the palette itself. It consists of AbstractPaletteElement instances. There are four kinds of palette elements that can be used:

- PaletteElement—a simple palette element with an icon and an label;
- Separator—a separating line;
- PaletteElementGroup—a container for similar palette elements grouped together. Groups cannot be nested and may be shown or hidden on user request;
- PaletteElementVariantGroup—a special kind of palette element group used for displaying the variants of the same palette element. Visually this group is shown as a normal palette element; however, it allows the switching to another palette element variant upon user request.

5 Transformation Structure

Describing the transformation part of the framework is not the objective of this paper. Therefore transformations will be discussed very briefly. As already stated, transformations in METAclipse are written in the MOLA model transformation language [28]. The MOLA compiler uses another model transformation language developed at UL IMCS, i.e. Lx language series [33]. Lx then is compiled to efficient C++ code, which is able to work with large models in fractions of a second. Only by accomplishing such performance is it possible to satisfy all needs of METAclipse, as every semantic user operation results in non-trivial transformations.

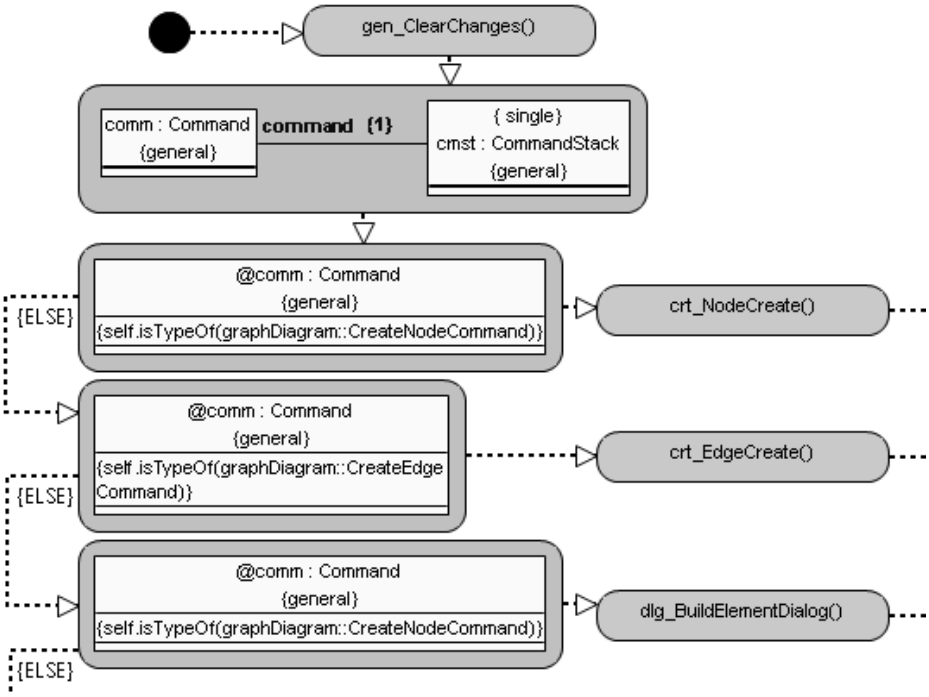

Fig. 19. Example of a MOLA transformation: a small excerpt of command handling procedure

In METAclipse there is only one entry point for the transformations, i.e., it is always the same transformation that gets called when executing a command. It is then the task of the transformation to call different procedures that implement model transformations that correspond to the particular command. In Fig. 19, one small part of the command parsing or main transformation is shown. It serves as an example of what MOLA transformations look like visually and at the same time displays how the single main transformation calls the sub-transformations in order to react to particular commands.

The transformation library is actually the key component that finally defines a concrete DSL tool created with METAclipse. Different tools built in METAclipse will have different transformation libraries. In order to build a tool, the toolsmith must first define the domain metamodel. Then he/she must link the domain metamodel to the presentation metamodel described in the previous section through model transformations. The presentation metamodel may be augmented for the transformation needs with new links or attributes. The only restriction is that existing classes, attributes and associations must remain intact. Finally, if necessary, the toolsmith must implement various functions through transformations that are needed for a particular tool.

6 Future Work

Currently METAclipse already has all the functionality needed for successful building of rich DSL editors. So, for example, the MOLA editor, built with METAclipse, has proved to be a powerful tool for editing MOLA transformations and is being successfully used. There is still a lot of work to be done in order to make the creation of transformations easier, so that tools could be built with much less effort. This would include generalization of common transformations, creation of reusable transformation frameworks (small frameworks for properties, styles, etc.), incorporation of the static mapping approach, definition of helper-functions, etc. Analysis of the transformation part, however, is beyond the scope of this paper.

Of course, there are also tasks to be done in order to make the METAclipse presentation framework (engines) more convenient and easier to use. Additional features could be implemented to enable more functionality for the tools. Some of these tasks are:

- Creating a more advanced property engine in order to allow building of more customized property pages. Currently the layout and contents of property sheets are very rigid and only a limited number of various controls can be used. There are cases when it is necessary to have richer property editors;
- Introducing the possibility for transformations to impact the engines, meaning that some special commands could be issued from transformations, which then would be interpreted by engines. This would be necessary, for example, to provide interactive debugging support for DSL editors.
- Adding possibilities to include animations. This would also be particularly useful for debugging.

- Implementing XMI import/export for domain part of the models. EMF already has the functionality needed for serialization and de-serialization of the models to XMI, however, currently only the presentation model is loaded via wise objects.
- Enhancement of the current graph diagram engine to allow more advanced constructs, such as swimlanes and pins used in UML activity diagrams.
- Creation of new engines for editing of other kinds of diagrams.

The named tasks represent just several areas in which it is already thought of to extend the METAclipse framework. The effort needed to implement the features listed above is relatively small compared to what has been already invested to provide the basic functionality of METAclipse, and all these tasks can be considered as "extras." Of course, the number of new features that could be added and that could be useful for the toolsmiths, as well as for tool users, is virtually unlimited.

References

1. Smolander, K., Martiin, P., Lyytinen, K., Tahvanainen, V-P.: Metaedit—a flexible graphical environment for methodology modeling. Springer-Verlag, 1991.
2. Ebert, J., Suttenbach, R., Uhe, I.: Meta-CASE in Practice: a Case for KOGGE. Proceedings of the 9th International Conference, CAiSE'97, Barcelona, Spain, 1997, pp. 203–216.
3. DOME Users Guide, http://www.htc.honeywell.com/dome/support.htm
4. Karsai G.: A Configurable Visual Programming Environment: A Tool for Domain-Specific Programming, IEEE Computer Society Press, pp. 36–44, 1995.
5. MetaEdit+, http://www.metacase.com/
6. Ledeczi, A., Maroti, M., Bakay, A., Karsai, G., Garrett, J., Thomason IV, C., Nordstrom, G., Sprinkle, J., Volgyesi, P.: The Generic Modeling Environment. Workshop on Intelligent Signal Processing, Budapest, Hungary, May 17, 2001.
7. de Lara, J., Vangheluwe, H., Alfonseca, M.: Meta-Modeling and Graph Grammars for Multi-Paradigm Modeling in AToM3. Software and System Modeling, 3(3), 2004, pp. 194–209.
8. Steven Kelly, Kalle Lyytinen, Matti Rossi: MetaEdit+ A fully configurable multi-user and multi-tool CASE and CAME environment Lecture Notes in Computer Science, Volume 1080, Proceedings of the 8th International Conference on Advances Information System Engineering, pp. 1–21, Springer-Verlag, 1996.
9. Meta-Object Facility (MOF), http://www.omg.org/mof/
10. OMG, Unified Modeling Language: Superstructure, version 2.0, http://www.omg.org/docs/formal/05-07-04.pdf
11. Celms, E., Kalnins, A., Lace, L.: Diagram definition facilities based on metamodel mappings. Proceedings of the 18th International Conference, OOPSLA'2003, Workshop on Domain-Specific Modeling, Anaheim, California, USA, October 2003, pp. 23–32.
12. Eclipse Modeling Framework (EMF, Eclipse Modeling subproject), http://www.eclipse.org/emf/
13. Graphical Editor Framework (GEF, Eclipse Tools subproject), http://www.eclipse.org/gef/
14. Graphical Modeling Framework (GMF, Eclipse Modeling subproject), http://www.eclipse.org/gmf/

15. N. Zhu1, J. Grundy and J. Hosking. Pounamu: a meta-tool for multi-view visual language environment construction. Proc. IEEE Symposium on Visual Languages and Human Centric Computing (VLHCC'04), pp. 254–256, 2004.
16. The Generic Eclipse Modeling System (GEMS), http://www.eclipse.org/gmt/gems/
17. S. Cook, G. Jones, S. Kent and A. C. Wills. Domain-Specific Development with Visual Studio DSL Tools. Addison-Wesley, 2007.
18. Ermel, C., Ehrig, K., Taentzer, G., Weiss, E.: Object Oriented and Rule-based Design of Visual Languages using Tiger. Proceedings of GraBaTs'06, 2006, pp. 12.
19. I. Rath, D. Varro. Challenges for advanced domain-specific modeling frameworks. Proc. of Workshop on Domain-Specific Program Development (DSPD), ECOOP 2006, France.
20. Taentzer, G: AGG: A Graph Transformation Environment for Modeling and Validation of Software. Application of Graph Transformations with Industrial Relevance (AGTIVE'03), Vol. 3062, Springer LNCS, 2004.
21. Visual Automated Model Transformations (VIATRA2), GMT subproject, Budapest University of Technology and Economics, http://dev.eclipse.org/viewcvs/indextech.cgi/gmt-home/subprojects/VIATRA2/index.html
22. Fujaba. Universitat Paderborn, Institut fur Informatik. http://wwwcs.uni-paderborn.de/cs/fujaba/documents/user/manuals/FujabaDoc.pdf
23. C. Amelunxen, A. Königs, T. Rötschke, A. Schürr: MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. Model Driven Architecture—Foundations and Applications: Second European Conference, Lecture Notes in Computer Science, Vol. 4066, pp. 361–375, Springer 2006.
24. Kalnins, A., Vilitis, O., Celms, E., Kalnina, E., Sostaks, A., Barzdins, J.: Building Tools by Model Transformations in Eclipse. Proceedings of DSM'07 workshop of OOPSLA 2007, Montreal, Canada, Jyväskylä University Printing House, 2007, pp. 194–207.
25. Barzdins, J., Zarins, A., Cerans, K., Kalnins, A., Rencis, E., Lace, L., Liepins, R., Sprogis, A.: GrTP: Transformation Based Graphical Tool Building Platform. Proceedings of MODELS 2007, MDDAUI 2007 workshop, Nashville, Tennessee, USA, September 30–October 5, 2007, pp. 4.
26. Kalnins, A., Barzdins, J., Celms, E.: Model Transformation Language MOLA. Proceedings of MDAFA 2004, Vol. 3599, Springer LNCS, 2005, pp. 62–76.
27. ReDSeeDS. Requirements Driven Software Development System. European FP6 IST project. http://www.redseeds.eu/, 2007.
28. UL IMCS, MOLA pages, http://mola.mii.lu.lv/
29. Barzdins, J., Barzdins, G., Balodis, R., Cerans, K., Kalnins, A., Opmanis, M., Podnieks, K.: Towards Semantic Latvia. Proceedings of Seventh International Baltic Conference on Databases and Information Systems, Communications, Vilnius, Lithuania, O. Vasileckas, J. Eder, A. Caplinskas (Eds.), Vilnius, Technika, 2006, pp. 203–218.
30. Java Native Interface Specification, http://java.sun.com/j2se/1.5.0/docs/guide/jni/spec/jniTOC.html
31. Eclipse Model To Text project, http://www.eclipse.org/modeling/m2t/
32. The Eclipse Tabbed Properties View, http://www.eclipse.org/articles/Article-Tabbed-Properties/tabbed_properties_view.html
33. Lx Transformation Language Set, http://Lx.mii.lu.lv/, 2007.
34. R. Gronback, Build Better Graphical Editors with the Graphical Modeling Framework, Slides, Eclipseworld 2006, http://wiki.eclipse.org/images/0/08/Gronback_EclipseWorld2006_GMF.ppt.zip
35. Rational Software Architect (RSA), http://www-306.ibm.com/software/awdtools/architect/swarchitect/