University of Latvia
Institute of Mathematics and Computer Science

OSKARS VILITIS

METAMODEL-BASED TRANSFORMATION-DRIVEN GRAPHICAL TOOL BUILDING PLATFORM

Thesis for the Degree of Doctor of Philosophy
at the University of Latvia

Field: Computer Science
Section: Programming Languages and Systems

Scientific Advisor:
Prof., Dr. Habil. Sc. Comp.
AUDRIS KALNINS

Riga – 2009

TABLE OF CONTENTS

Introduction

In software engineering the *MDSD* (*Model Driven Software Development*, [1]) approach has gained continuously increasing popularity in recent years. The basic idea of the *MDSD* technology is to drive the whole software development process by specialized models that correspond to each development phase. These models not only serve as documentation of various aspects of the system, but also become a direct constituent of the software. In practice, various specialized modeling languages are nowadays used for description of the *MDSD* models. These languages (called *Domain Specific Languages* or *DSLs*, [2]) allow describing models of some specific problem domain more clearly and effectively by using notation designed specifically for the given domain.

Along with the introduction of *MDSD* in software development processes and its application to new problem domains, a constant need for creation of new *DSLs* arises. In order for the new *DSL*s to be practically usable, it is also mandatory to develop supporting tools, i.e., editors for the programs (models) of newly developed *DSL*s. Creation of such tools is a very time consuming process, which requires a lot of effort. Therefore, there is a need for some means of speeding up and easing the development of *DSL* tools.

One way of making the development of *DSL* tools easier is using a supporting universal metamodel-based tool building platform that allows simplified tool definition by creating a correspondence between the metamodel of the *DSL* (domain metamodel) and the metamodel of the tool being built (presentation metamodel). The research leading to this thesis is devoted exactly to this topic: universal metamodel-based tool building platforms.

It must be noted that the idea of universal metamodel-based tool building platforms is not new, and the history of such platforms can be traced to more than twenty years ago. *GMT* (*Generic Modeling Tool*, [3]), developed in *UL IMCS*, is among many platforms created during this time. *GMF* (*Graphical Modeling Framework* [4]), *Microsoft DSL Tools* [5] and

MetaEdit+ [6] can be named as the most popular tools of the recent years. All of the mentioned platforms belong to the category of tools that use static mappings for defining the correspondence between domain and presentation metamodels. The means available for definition of such mappings are limited: tools provide a fixed set of mapping types that can be used to map the model elements. This way of defining the correspondence is appropriate for relatively simple cases where the domain metamodel is close to the presentation metamodel, and no advanced mapping is required.

Unfortunately, *DSL* tools often require much more complicated and dynamic mapping logic. In order to satisfy the *DSL* needs, static mapping definitions alone are not enough, and in order to implement the required functionality, it is often necessary to write a supplementing logic in one of the object-oriented programming (*OOP*) languages (*Java* for *GMF*, *C#* or other supported language for *Microsoft DSL Tools*, etc.). This approach requires the user of the platform to have deep knowledge both in the corresponding *OOP* language and the platform architecture, so the usability and efficiency of these tool-building platforms are still unsatisfactory.

In order to overcome the constraints of static mappings, a new approach has emerged, which uses model transformations for the definition of the correspondence. This approach allows the definition of very complex mapping logic in a uniform way and does not require additional usage of *OOP* languages. This transformation approach is used in such platforms as *Tiger* [7], *Tiger GMF* [8], *ViatraDSM* [9] and *MOFLON* [10]. However, these platforms still do not allow complete freedom in mapping definition.

Despite the long history of research of metamodel-based tool building platforms, one must come to the conclusion that the desired result still has not been achieved: a completely universal and efficient solution for defining the correspondence between *DSL* and tool metamodels has not been found. Therefore, the problem of creating an optimal *DSL* tool building platform is still open. This thesis offers a new solution to the given problem: a metamodel-based tool building platform that is completely driven by model transformations. The newly developed methodology has the potential of finally achieving the desired universality and efficiency in tool building arena.

One of the main results of the research carried out by the author is the methodology that allows using model transformations in building metamodel-based tools. The practical solution proposed by the author, *METAclipse* [11], demonstrates application of this methodology in developing a platform for building *DSL* editors. The methodology itself, however, can be used more generally, for creating a variety of transformation-driven metamodel-based tools. Also the formalization of the presentation mechanisms that has been created for the *METAclipse* platform (the *Eclipse* presentation metamodel) is important by itself, as it allows looking at the *Eclipse* platform [12] through a modeling perspective and driving the elements of the *Eclipse* platform through formal models. An interpreter of the presentation metamodel has also been created (it is an extension of the *Eclipse* engines, which fully utilizes their features), allowing to visualize the instances of the metamodel in corresponding *Eclipse* engines.

As an interesting side result with its own scientific value, the developed methodology for external repository integration in the *Eclipse EMF* environment must be noted. Practical application of the methodology has been demonstrated by integrating the *MIIREP* [13] repository in the *METAclipse* platform. The methodology itself, however, can be used more generally for integrating an arbitrary *MOF*-isomorphous model repository.

The most straightforward proof of the practical significance is the use of the *METAclipse* platform in the development of the *MOLA* language [29] editor. *EU 6th Framework* project *ReDSeeDS* (*Requirements-Driven Software Development System*, [14]), which is aimed to develop methodology and supporting tools for a real (in the sense of *MDA* [15]) model-based system development, bases the whole development on the newly created *MOLA* editor.

The research results presented in the thesis have achieved the desired efficiency in building complicated *DSL* editors, exceeding the capabilities of currently available tools. However, the static mapping approach is admittedly more efficient for simple *DSL*s. By combining the transformation-driven approach with the possibility to use static mappings, the developed methodology has a potential to reach or even exceed the efficiency of the existing solutions also in the development of editors for simple *DSLs*. Currently this topic is actively being researched in *UL IMCS*.

The thesis presents a newly developed *Eclipse*-based platform *METAclipse*, which practically implements the developed methodology and defines the presentation metamodel consisting of several parts, the most important of which is the graph diagram metamodel. *METAclipse* provides engines driven by these metamodels, which are incorporated in a very simple architecture allowing easy use of transformations. Following chapters give an in-depth description of the developed methodology and architecture of the *METAclipse* platform, overview of all principles and technologies utilized by the platform, and technical solutions created for *METAclipse*:

- CHAPTER 1 outlines the main ideas and basic principles of metamodel-based graphical tool building platforms. The reader is thus given the basic knowledge needed for understanding the research carried out by the author, as well as the significance of the results achieved. At the end of the chapter, two metamodel-based graphical tool building approaches are compared: the static mapping approach and the model transformation approach (*METAclipse* follows the latter). It is also explained which situation is more suitable for which of the approaches. This chapter of the thesis also gives an overview and analysis of the most popular metamodel-based graphical modeling tool building platforms of the recent years.

- CHAPTER 2 briefly describes the *Eclipse* technologies used for the implementation of the *METAclipse* platform. Justification for the suitability and use of each technology is given, along with a description of the extent to which each of them has been used in *METAclipse*. This chapter provides the reader with the basic knowledge needed for understanding the solutions described in the following chapters.
- CHAPTER 3 explains the basic principles of the developed methodology for the application of model transformations to building metamodel-based tools. It also explains the main operating principles of the *METAclipse* platform and provides description of the main components of the tool.
- CHAPTER 4 gives a detailed description of the solution for external repository integration in the *Eclipse EMF* environment. The chapter also analyzes the practical applicability of the solution.
- CHAPTER 5 gives a detailed description of technical solutions developed for *METAclipse*. It also analyzes in greater detail the metamodels of all the developed engines.
- CHAPTER 6 demonstrates the practical application of the developed tool building platform: the editor of the *MOLA* graphical transformation language.
- CHAPTER 7 lists the conclusions accumulated during the development of the thesis. Also, possible future directions of the research in metamodel-based transformation-driven tool building platforms are outlined.

CHAPTER 1

Metamodel-Based Graphical Tool Building Platforms

When introducing metamodel-based graphical tool building platforms, it is worth taking a brief look at their history. The first simple generic metamodel-based tool building platforms, such as *MetaEdit* [16], *Kogge* [17] and early versions of *Dome* [18, 19], appeared already in the mid-nineties, but their capabilities were quite limited. Tools of the next generation, such as *MetaEdit+* [6], *GME* [20] and *ATOM3* [21], appeared around year 2000 (the first version of *MetaEdit+* appeared much earlier [22]). These tools already had domain metamodeling facilities close to *MOF* [23] and more advanced graphical capabilities. They made it possible to base visual languages on a presentation-independent metamodel (such as the UML [24] metamodel). The *GMT* (*Generic Modeling Tool* [3]) tool building platform, developed at *UL IMCS*, also belongs to this category.

The popularity of *DSLs* has grown in recent years, and these languages are increasingly being used in everyday software development tasks. As a response to such increased popularity, a completely new generation of tool building platforms has emerged. One such group of platforms is based on the open-source *Eclipse* platform, which, together with its *EMF* plugin, is a broadly used, popular metamodeling environment, close to *MOF*. Historically first and still the most popular among *Eclipse*-based graphical tool building platforms is *GMF* [4]. Alternative *Eclipse*-based solutions are provided by the *Pounamu/Marama* [25] environment and the *GEMS* project [26]. A popular alternative to *Eclipse* on a commercial basis is offered by *Microsoft DSL Tools* [5], available in *Microsoft Visual Studio* starting from version 2005. The logical capabilities there are quite close to *GMF*. The already mentioned *MetaEdit+* has significantly evolved in recent years and has also become a key player in this area.

In order to define the correspondence between the metamodel of the *DSL* language and the presentation metamodel, all of the above-mentioned solutions use the so-called static mapping approach. This means that for the definition of the correspondence a fixed set of patterns is used, which describes which presentation objects correspond to a given domain object. Furthermore, this set is fixed at development-time and cannot be changed during runtime. Such approach is suitable for relatively simple cases, when domain and presentation metamodels are close and no complicated mapping logic is required. Whenever metamodels have more significant differences, the correspondence cannot be defined through static mappings and some traditional programming language has to be used in order to define the mapping. This requires a deep knowledge and understanding of the architecture of the tool building platform itself and is not easy to implement.

In order to solve this problem, a new approach has appeared: to define this mapping by model transformation languages. Model mappings in tools are actually very close to traditional *MDA* tasks, for which model transformation languages were invented. Therefore these languages can be considered very appropriate *DSL*s for metamodel-based tool building, yielding development efficiency that is an order of magnitude higher when compared to that of *OOPL*s.

The first platforms using this approach to a degree are *Tiger* [7], *ViatraDSM* [9], and *MOFLON* [10]. However, these platforms still do not provide complete freedom in the definition of mappings. So, for example, the *Tiger* platform uses a specific domain modeling notation, which still forces the domain metamodel to be close to the presentation metamodel. The other tools have similar constraints. The subject of the research for the thesis is a completely transformation-driven platform, which does not have any constraints on how the correspondence between the metamodels can be defined. The goal of the research is to develop a universal metamodel-based tool building platform that would be driven by model transformations.

Further subdivisions of the chapter will describe the basic concepts and principles used in metamodel-based graphical tool building platforms. In the beginning, basic metamodeling concepts are introduced and process of the model transformations explained. Then the model-driven approaches in software development are introduced. In chapter 1.4, two approaches for the graphical tool building platform architectures are described. The last part of the chapter concentrates on describing some of the most popular metamodel-based graphical tool building platforms. Key features and brief analysis of each platform is given.

1.1 Basics of the Metamodeling

In order to understand the subject of the thesis and the principles of the graphical tool building platforms, it is necessary to first understand the basic ideas of metamodeling. This chapter will give an explanation of the main terms of metamodeling.

Metamodeling as such is a construction of the collections of concepts (objects, phenomena, terms, events, etc) within a certain domain. In metamodeling, a model is an abstraction of phenomena in the real world. It is a set of interconnected objects, which usually is presented in the form of a graphical image, where different kinds of elements are shaped and laid out according to certain rules (graphical representation is not obligatory: a model can also be textual).

Metamodel is yet another level of abstraction, which describes the properties of the model itself. The model is said to conform to a metamodel as a program conforms to the grammar of the language in which the program has been written. So, for example, the grammar of the *UML* [24] language is the *UML* metamodel. Metamodels are also referred to as models of models.

Other examples of metamodels include the metamodels that define the elements for describing the organizational structure of a company, as well as models describing automotive engineering processes and models of physical processes, etc. In any case, a metamodel defines rules for creating new models. Models conforming to a certain metamodel are also called instances of the metamodel.

Classic metamodeling is based on an architecture that consists of four meta-layers:

- **The information layer** contains the data that we want to describe, i.e., the real-world objects.
- **The model layer** contains metadata that describe the data of the information layer. Informally these metadata are known as models.
- **The metamodel layer** contains meta-metadata, which define the structure and semantics of the metadata. Informally meta-metadata are referred to as metamodels. Metamodels are abstract languages that allow describing different kinds of data, i.e., they are languages without a concrete syntax or notation.
- **The meta-metamodel layer** contains definitions of the elements for description of the meta-metadata structure and semantics. In other words, it is an abstract language for definition of different kinds of metadata.

The most popular formalization of the four-layer architecture is *MOF* (*Meta-Object Facility* [23]). In *MOF* terminology the four layers of the architecture are called correspondingly *M0*, *M1*, *M2* and *M3* (see. Fig. 1). *MOF* defines a metamodel at the *M3* layer. This metamodeling language is being used for creating metamodels at the *M2* layer. The most prominent example of an *MOF M2* model is the *UML* metamodel: the model that describes the *UML* modeling language. *M2* models describe elements at the *M1* layer (for example, *UML* models), which, in turn, describe the *M0* layer: the real-world objects.

Because of the similarity of the *MOF M3* model and *UML* structural models, *MOF* metamodels are usually modeled as *UML* class diagrams. In fact, *MOF2* even reuses parts of the *UML* metamodel. *MOF* is a closed metamodeling architecture: it defines an *M3* metamodel that conforms to itself.

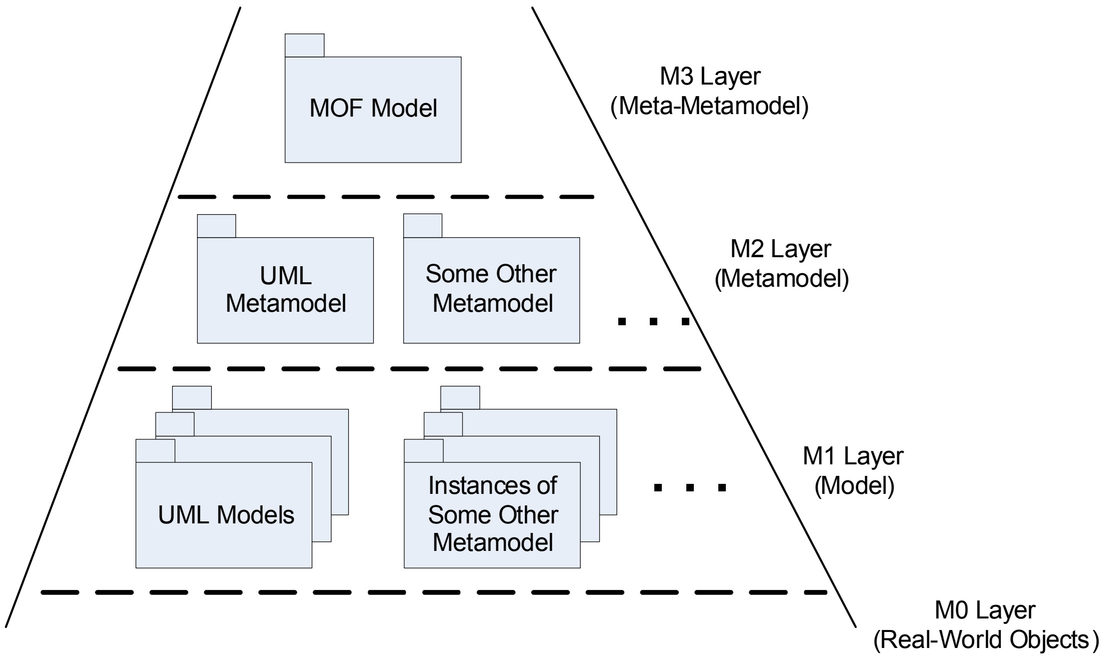


Fig. 1. *MOF* architecture

Some typical examples of the models:

- Business process model (*UML* activity diagram).
- System requirement specification (*UML* use-case and activity diagrams).
- System analysis model (*UML* class diagrams).
- Workflow definition (*BPMN* [27] or similar language).

1.2 Transformation Languages

As the focus of the research of the thesis is a transformation-driven platform, a couple of words have to be said also about model transformations. Let there be a model *Mo1* that conforms to a metamodel *MMo1* (see Fig. 2). Then a transformation of this model is a process that transforms it into another model *Mo2* by following certain rules, so that it conforms to the metamodel *MMo2* (*MMo2* can also be the same as *MMo1*). Thus, a model transformation is a set of rules that determines how a given model has to be transformed. In *MOF* terminology, the transformation process is defined at the *M2* layer and transformation itself is carried out at the *M1* layer.

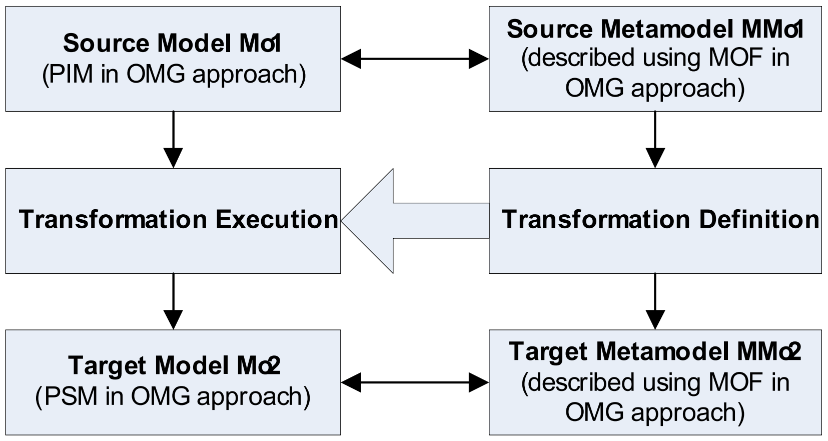


Fig. 2. Transformation execution process

In order to be able to define model transformations, a specialized programming language—a model transformation language—is required for description of the model transformation processes. A program written in a given transformation language typically takes an instance of some metamodel as input data and returns an instance of some other (or the same) metamodel as a result. In the aforementioned example, the input data of the transformation program are *Mo1*, and the result is *Mo2*.

The history of transformation languages is also quite long. In 2002 *OMG* announced a tender for the development of a standard for the transformation language *QVT* (*Queries / Views / Transformations*). Several projects were submitted for the development of the standard, most of which were either discontinued or merged over time, until just one project remained: *MOF QVT* [28]. 16 institutions participate in the development of this standard, among which are *IBM*, *Sun,* and four universities. It must be noted that the final version of this language was approved only recently.

At the same time a series of other transformation languages were developed that were not directly associated with the *OMG* tender. It is interesting that the existing transformation languages are still being extended and new languages created. Amongst them are both graphical and textual languages. Special attention has to be brought to the graphical model transformation language *MOLA* [29, 58], which is being developed in *UL IMCS* and is used also for driving the *METAclipse* tool.

1.3 Model-Driven Approaches in Software Development

In the end of the nineties, enough experience had been accumulated to introduce a new paradigm in software development. Software industry had well accustomed technologies like metamodeling, model processing, use of *UML* in software engineering, various component-based environments, object-oriented languages, etc. Still, all of this had not given the expected efficiency increase in software development. In year 2000, *OMG* started a new project: model-based architecture (*MDA*, [15]). First notable results were presented in the end of the year 2001 when *OMG* published the first version of the *MDA* guide explaining the basic ideas of *MDA* and its applications.

MDA bases the whole software development process on special models, and these models become direct development artifacts. Three kinds of models are used by *MDA*:

- Platform-Independent Model (*PIM*), a model that does not contain any platform-specific information.
- Platform-Specific Model (*PSM*), a model that supplements *PIM* by information, specific to a particular platform (*EJB*, *.NET*, *CORBA*, etc.).
- Computation-Independent Model (*CIM*), the conceptual model of a system or business model. It must be noted that *CIM* usually represents a part of the documentation rather than a formal model, understood by a computer.

In software development process these three kinds of models are used sequentially: first, *CIM* is developed, then, according to *CIM*, *PIM* is produced. And finally, *PSM* is produced from the according *PIM*. In order to transfer information from model to model,

model transformations are used. Model transformation process is the keystone in model-based system development.

Over the time there has developed multiple approaches of how models and model transformations are being applied in software engineering. In *OMG* terms, a classical *MDA* application is designed in a way that it consists of one platform-independent model and one or more platform-specific models accompanied by full implementations. During the application development, transformations (that are part of the development process) are used. It must be noted, that only means of metamodeling allowed in *MDA* are the ones provided by *OMG* (namely, *MOF* and *UML*).

Another approach, *MDSD* (Model-Driven Software Development, [1]), does not tie the development process to particular standards, but instead allows to use any formalized metamodeling architecture. In specific problem areas a widely accepted approach is to use specialized modeling languages, so-called Domain-Specific Languages (*DSL*s, [2]). These specialized languages allow using domain-specific notation for describing the domain, which is in most cases much more appropriate than the use of the plain *UML*.

Another important difference in *MDSD* comparing to *MDA* is that in *MDSD*, any two sequential models used in development process can be source and target models. The categorization in *PIM* and *PSM* models is relative to the usage of the models and abstraction level from which we look at them. So, for example, even such a specific model as *Java* program, written in its abstract syntax, can act both as a *PIM* and as a *PSM* model. Such program could be as a *PSM* model for a design class diagram or as a *PIM* model for a specific development environment platform, which in its place is a *PSM* model for the abovementioned *Java* program.

Due to the increasing interest in the *MDA* approach and the growing popularity of various domain-specific languages (*DSL*s), the need for a possibility to rapidly build editors for specific *DSL*s is increasing constantly. This is where metamodel-based tool building platforms come in to play.

1.4 Paradigms of the Metamodel-Based Graphical Tool Building Frameworks

This chapter gives a brief analysis of the well-known static-mapping-based approach and recently evolved, more flexible approach driven by model transformations, where the correspondence between the domain metamodel and the presentation metamodel is defined by transformations. Both static-mapping-based and transformation-driven approaches are compared and their applicability for different situations is analyzed.

1.4.1 General Principles of Paradigms

A visual language basically consists of two parts – the domain part and the presentation (visual) part. Sometimes they are called also the abstract and concrete syntax respectively – the terminology taken from textual languages. For visual languages the mostly used syntax specification technique is metamodels. Sometimes graph grammars are used too (this possibility will not be analyzed here).

The domain part of the language is defined by means of the *domain metamodel*, where the relevant language concepts and their relationships are formalized. The domain metamodel is used also for precise definition of language semantics. Typically standard *MOF* [23] is used for the definition of domain metamodel. Some frameworks (*Microsoft DSL Tools* [5], *MetaEdit+* [6], *Tiger* [7], and *ViatraDSM* [9]) use a slightly alternative notation for domain metamodels, where those metamodel classes, which correspond to edges in diagrams, are singled out and called relationships (actually they are equivalent to *UML* association classes, which formally are not part of *MOF*). A note on terminology should be added here. Though according to general modeling principles the language domain is defined by its metamodel, several frameworks, including *GMF* and *MS/DSL Tools*, call the definition the *domain model*.

For the presentation part (concrete syntax) there is no universally accepted notation. The same metamodeling techniques typically are used, but with various semantics. During the definition of the presentation part, for example both *GMF* and *MS/DSL Tools* use a *presentation metamodel*. Instances of classes in this metamodel are *types* of diagram elements to be used in the diagram (e.g., *ClassNode*, *AssociationEdge*). A concrete set of graphical element types for a diagram definition is called the *presentation* (or graphics) *model*. During runtime *GMF* uses another metamodel (*notation* metamodel), where instances are specific graphic *elements* in a diagram (nodes, edges, etc.).

As already mentioned, metamodel-based tool building platforms can be divided in two categories: depending on how they are defining the correspondence between the domain and presentation metamodels. The most traditional is the static mapping definition approach. In this approach, during the development of a specific *DSL* tool, a presentation metamodel element (for example, a graph node type, edge type, or label type) gets associated with a specific element of the domain metamodel. This association is called mapping. By using such mappings, domain elements get visualized. In the typical case, a specific node type is mapped to a certain class of the domain metamodel, and a specific edge type is mapped to a certain association of the domain metamodel. It means that all instances of the domain class will be visualized using the given node type. The structure of the diagram elements (how elements will be included in each other, how elements will be connected etc.) is also specified by the mappings. Therefore, static mappings fully define the whole functionality of the graphical tool. Different tools provide various additional tools allowing creation of more dynamic mappings (for example, some tools allow to use *OCL* [30] constraints, while others use specialized constraint languages).

Tools driven by static mappings usually have a generation step, during which the code in some *OOP* language (*Java*, *C*++ or another) is generated, implementing logic described by the mappings. However, code generation is not mandatory and some tools work as mapping interpreters (for example, *MetaEdit+* and *GMT*). In order to allow implementing logic that cannot be described through static mappings, tools with the generation step usually provide some mechanism allowing supplementing the generated code in the target *OOP* language.

An alternative to the static mapping approach is the model-transformation-driven approach, which is the main topic of the thesis. In this approach, a correspondence between the domain and presentation metamodels is defined dynamically, through the usage of the model transformations that are described in an appropriate transformation language. These transformations describe what corresponding changes have to be carried out in the domain or presentation metamodel, if one of them is changed (in result of a user action or another internal activity). Synchronization between models can occur in both directions. The key distinguishing feature of the transformation approach, compared to the static mapping approach, is the possibility to freely define any kind of relationship between the domain and presentation metamodels. It should be noted that the transformation approach requires a different model-metamodel terminology (transformations always process models defined by metamodels). Therefore in the transformation approach by domain model we must understand a concrete set of runtime instances, and by presentation model a diagram (or several diagrams) representing this set visually. The definitions always are metamodels.

Let us conclude the chapter with some more comments on metamodeling. The defined domain metamodel always is at the layer *M2* according to the *MOF* hierarchy (it is defined according to *MOF* itself, which is at *M3*). A concrete domain model (a runtime model in a tool) is at *M1*. However, in the static mapping approach the presentation definition metamodel is at *M2* within the *MOF* hierarchy (it is not *MOF* itself, but a specific metamodel); the defined presentation model (which contains the diagram element types) is at *M1*. The runtime presentation model (also at *M1*) is not an instance of this definition model. This *MOF* layer mismatch is not a problem for the generation approach. There only one metamodel/model layer is used (metamodels for definition, corresponding models for generation of language classes). However, the transformation approach (for all transformation languages) requires a symmetric situation at both domain and presentation. Therefore the presentation definition metamodels from *GMF*, *MS/DSL Tools* or other frameworks cannot directly be used for transformations.

1.4.1 Areas of applicability

Here a brief discussion on *DSL*s is presented, in order to find out in what situations which of the mapping approaches is superior. From the tool development perspective, most of the industrial *DSL* languages (in telecommunications, process control, automotive industry etc.) are relatively simple. Usually in these sectors no metamodeling standards are used and the domain metamodel can be created from graphical notation used in practice. Examples of such *DSL*s are usually used in the documentation of different tools (see, for example, [31]). Here the mapping logic is very trivial: domain classes are associated with graph nodes; associations with edges; class attributes with labels and so on. Of course, the runtime semantics of these models is complicated, but this is a different dimension. For creation of tools for these cases, the static mapping approach is very appropriate.

However, it has to be understood that the use of static mappings is this simple only in cases when the correspondence between the domain and presentation metamodels is obvious. In the software development industry *DSL* languages usually are much more complicated. Often a given language already has certain domain metamodel standards, which are used for the definition of formal semantics and are far from the concepts used for graphical representation of the language. Furthermore, the interdependency among language elements is much more complicated, including cases when the same elements are used in multiple diagrams. When developing language editors, editing of the language elements typically has to be made syntax-oriented, i.e., the editor must prevent the user from creating incorrect constructs as much as possible. All these factors make the correspondence between the domain and presentation metamodels very complicated and make the application of static mappings less effective. For the transformation approach, on the other hand, the conditions mentioned present no obstacles.

An example class of such complicated *DSL*s could be model transformation languages, more precisely, the graphical ones. They include the *OMG* standard *MOF QVT* [28], the *MOLA* language [29] used in this thesis, *Fujaba* language [32] and some others. Let us consider a small fragment of the *MOF QVT* language. For this language, the *OMG* standard provides the domain metamodel and both graphical and textual syntax. The domain metamodel is quite far from the graphical form of the language.

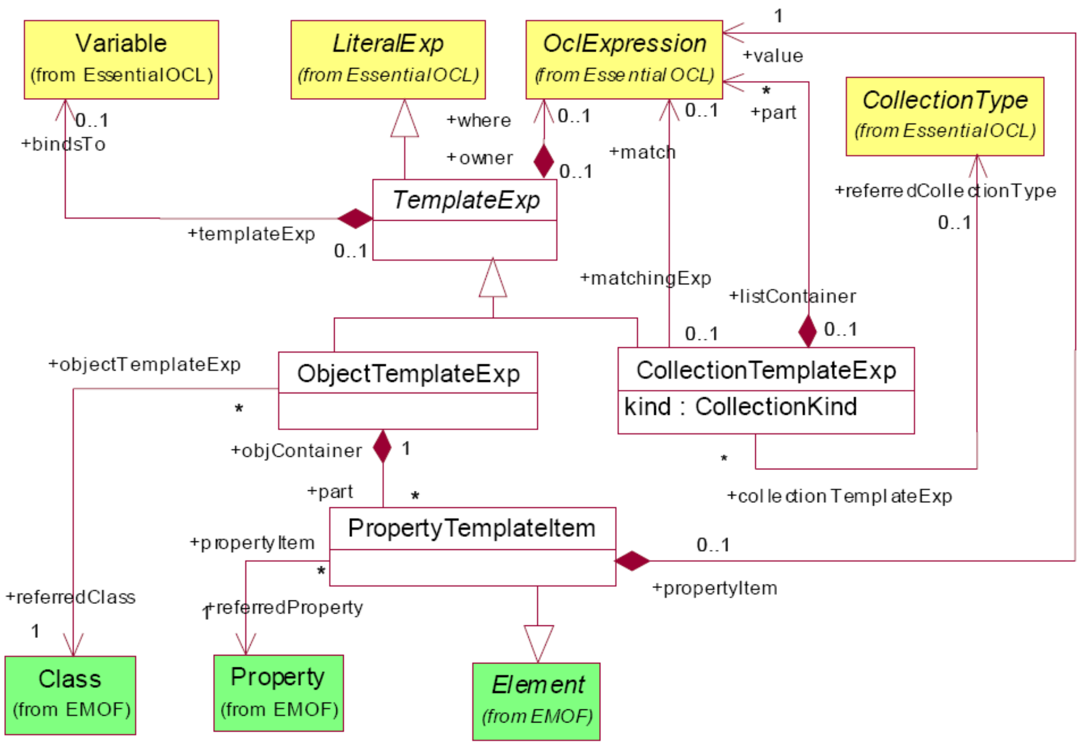


Fig. 3. *QVT* template package

For example, the *ObjectTemplateExp* (see Fig. 3 or [28], page 30) domain metamodel class has to be depicted as a pattern element (a rectangle similar to a class instance in *UML*). An instance of this domain class can contain multiple instances of *PropertyTemplateItem*. If the *value* association of the domain is pointing to a corresponding *OCL* expression, then each of these instances has to be depicted as certain slot in the pattern element (a label within a compartment in *GMF* terminology). If the *value* association is pointing to another *ObjectTemplateExp*, then each of the instances has to be depicted as a link (edge) to another pattern element. Another complicated task of *QVT* (as an example of a complicated *DSL*) is composition of various texts, as they are composed from various domain classes, and the content of the text depends on model data.

It is obvious that the described correspondence between the domain and the graphical presentation cannot be specified with any static mapping definitions (and this is not the only example of this kind in *MOF QVT*). If we tried to define this correspondence in *GMF*, this would mean vast programming in pure *Java* (with just some generated classes to be reused). On the other hand, by using transformations, the implementation of the given logic is relatively simple.

The situation of the *MOLA* transformation language itself is similar to that of the *QVT*, though the domain metamodel there is closer to the graphical presentation. However, the requirement that the graphical tool must be syntax directed (and support a lot of context-sensitive syntax constraints during diagram building) requires that the domain metamodel must be "semantic" to a great degree. Therefore the correspondence between domain and presentation is complicated in *MOLA* too. For example, each pattern element in *MOLA* domain (a concept similar to that in *MOF QVT*) has only one of several possible paths in a model to the corresponding class, the existing path must be found to display the class name as a label in the corresponding node. The composition of label texts is as complicated as in *MOF QVT*. A specific issue in *MOLA* tool (more related to property editing than mapping to presentation) is the requirement to offer only relevant associations for building a pattern link. To find the relevant list, a transitive closure of inherited associations in a class diagram must be computed. There are other such complicated situations for property editors in *MOLA* too. All the mentioned would make the implementation of *MOLA* tool by means of static mappings quite complicated. At the same time, this thesis shows in CHAPTER 6 that a *MOLA* tool satisfying all requirements has been built by reasonable efforts using the transformation approach.

One more example from a completely different area would be a tool which would visualize an *RDF* [33] data base as a simple class diagram, with update support too. This task has become popular in the context of ontology development. The domain model implied by typical *RDF* data bases [34] is a simplified metamodel for *RDF* triples [35]. To visualize *RDF* data as a class diagram (containing also instances) in a natural way, a complicated analysis of attribute values (of *String* type) for some *RDF* domain classes must be done. Only this way it may be decided whether an *RDF* property should be visualized as an attribute or association. Such a tool is apparently easier to be implemented by the transformation approach.

1.5 Existing Non-Eclipse Platforms

In order to provide a better understanding of the current situation in the world of metamodel-based *DSL* editor building platforms, this and the following section will briefly introduce some of the most popular of them. In these chapters, most characteristic features of each platform will be described. Although the focus of the thesis is on *Eclipse* environment, platforms based on other environments also are worth discussing, as ideas of the tool building are similar regardless of the implementation technology. This chapter concentrates on non-*Eclipse* solutions.

1.5.1 MetaEdit+

MetaEdit+ [6] tool is one of the most popular commercial *DSL* editor building platforms and is used widely for development of *DSL* editors in various domains (mobile communications, web application development, industrial processes, etc.). *MetaEdit+* is using a very simple metamodeling language called *GOPPRR* [22]. There are just Properties and Non-Properties (which in turn can have Properties again). Non-Property-Types are Graph, Object, Port, Role and Relationship, and for each of these types there is a creation wizard/form. See Fig. 4 on how these terms relate to an example of a data flow diagram.

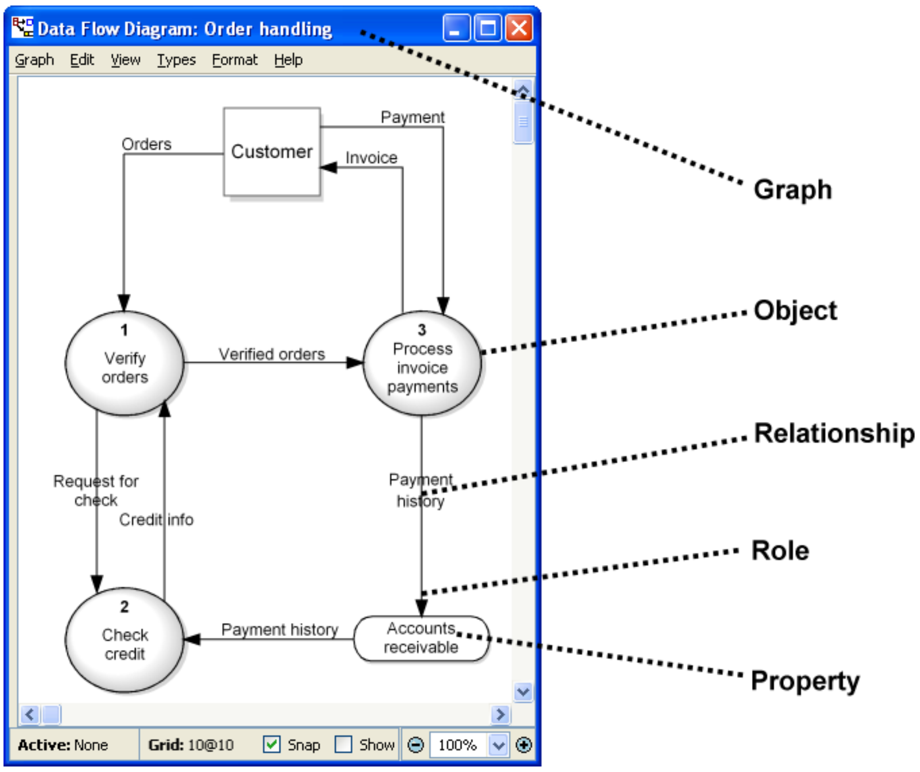


Fig. 4. A data flow diagram example in MetaEdit+

MetaEdit+ is using static mapping approach for definition of the correspondence between the domain and presentation metamodels. A feature that distinguishes *MetaEdit+* from other tools is its symbol editor (see Fig. 5). One does not have to specify the concrete syntax by hand. Rather it is possible just to draw the component in a *WYSIWYG*-style editor which makes the object creation process very simple. In particular it is not necessary to define the mapping between the domain and presentation metamodels explicitly like in *GMF*; instead it is done by the tool for you. This is possible because of the fact that mapping is very straight-forward: objects correspond to nodes and relationships to edges of the diagram. Even more: some properties of the presentation metamodel are encoded in the domain metamodel. The relationships are specified by providing a so-called binding that tells the system which object types may be linked together and in which roles.

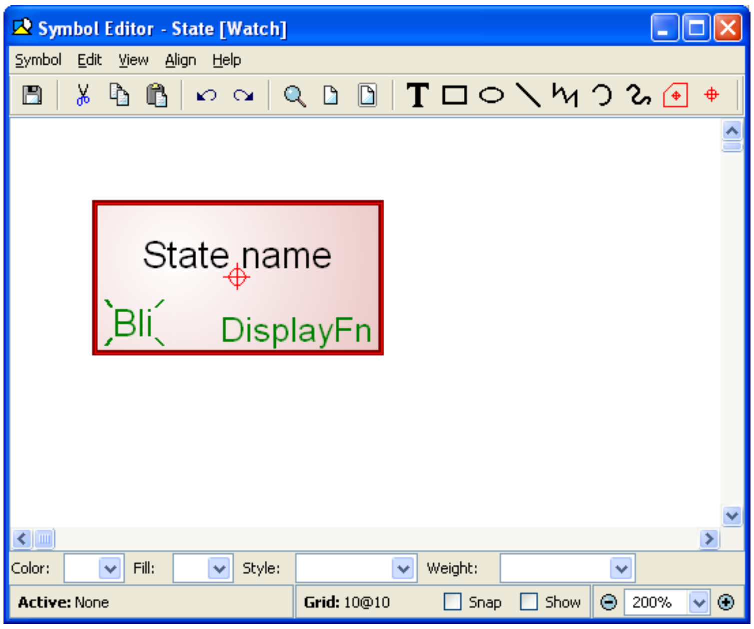


Fig. 5. Symbol editor of the *MetaEdit+* tool

Some other advantages of *MetaEdit+* are:

- feature-richness, e.g., sophisticated editors for components,
- good documentation and plenty of examples,
- high integration,
- possible *Model2Code* transformations.

Being a tool using the static mapping approach, *MetaEdit+* is suitable for creation of editors for relatively simple *DSLs* with more or less straight-forward correspondence between the domain and presentation.

1.5.2 Microsoft DSL Tools

"The *Microsoft Tools for Domain-Specific Languages* [5] is a suite of tools for creating, editing, visualizing, and using domain-specific data for automating the enterprise software development process" [36]. *MS/DSL Tools* employ the Software Factories approach that is introduced by [37] and extend on the *Microsoft*'s notation for *Domain Specific Modeling* which is a specific realization of *MDE* standards and principles (see, for example [38] for more detailed description). Also *MS/DSL Tools* suite uses the static mapping approach for tool definition. One of the main distinguishing features of the *MS/DSL Tools* platform is the sophisticated integration with other *Microsoft Visual Studio* tools, which also is the reason for its popularity among developers utilizing *Microsoft* technologies.

MS/DSL Tools use its own meta-metamodel, which offer classes, value properties, and relations such as embedding (composition), reference (aggregation) and inheritance (see Fig. 6).

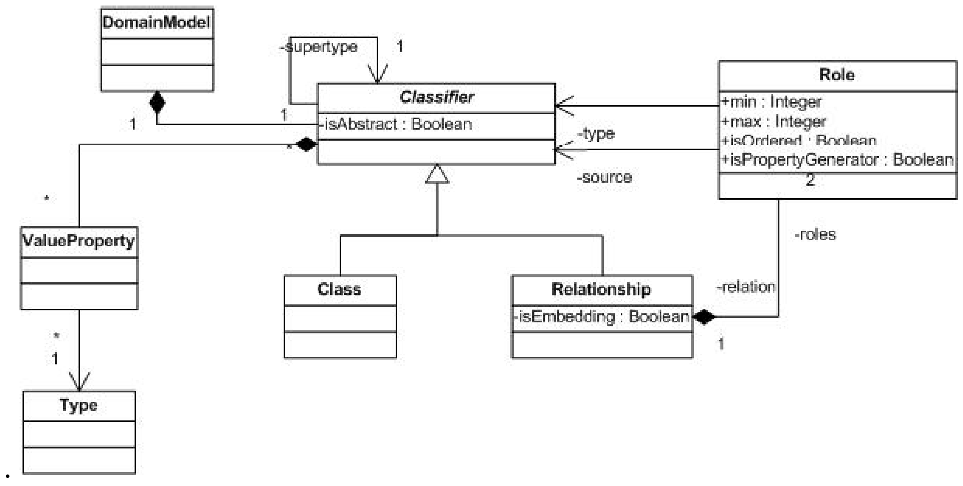


Fig. 6. Simplified version of *MS/DSL Tools* meta-metamodel

The *MS/DSL Tools* suite consists of:

- A project wizard for creating a fully configured solution. In this solution, one can define a domain metamodel that consists of a designer and a text output generator.
- A graphical designer for defining and editing domain metamodels.
- Graphical facilities for defining the mapping between the domain metamodel and presentation elements. Mappings are defined graphically by drawing special mapping lines from domain elements to presentation shapes and connectors.
- Designer definitions in *XML* (mappings). The source code for implementing designers is generated from these definitions.

- A set of code generators, which take a domain metamodel definition and a designer definition as input, validate it and produce source code that implements both components as output.
- A framework for defining template-based text output generators.

The main steps in building a *DSL* in *MS/DSL Tools* are:

- Defining the domain metamodel in .dsl file.
- Defining the notation elements such as shapes and connectors. The *XML* serializations of notation elements are automatically generated and stored in a separate file named .dsl.diagram.
- Defining visualization of domain metamodel via notation elements (mapping shapes to classes and connectors to relationships).

Following the static mapping path, *MS/DSL Tools* platform also is suitable only for relatively simple *DSL*s and straight-forward mappings (even fewer capabilities than in GMF are provided). In more complicated cases, this platform allows to manually add the required functionality by writing the code in one of the languages supported by the *Microsoft Visual Studio*.

1.5.3 Generic Modeling Tool

Another tool based on the static mapping approach is the *Generic Modeling Tool* (*GMT*, [3]), developed in *UL IMCS*. This tool can be considered as one of the pioneers of the metamodel-based modeling tool building platforms. At the time when this tool was created, it was one of the most sophisticated *DSL* editor building platforms available. It can be said that the research for this thesis is directly inspired by the results and experience of the *GMT* development and practical application, accumulated in the *Institute of Mathematics and Computer Science* at the *University of Latvia*.

GMT is based on a meta-metamodel that is very similar to *MOF*. For tool definition *GMT* provides mapping definition wizards (both for graphical elements and element property editors). Correspondence between the domain and presentation metamodels is defined, using so-called mapping links (associations) between the model elements by specifying a particular mapping type for the link. The semantics of each mapping type is predefined in *GMT*; however in some cases it is possible to use explicit *OCL* expressions in order to specify more advanced constraints.

One of the main distinguishing features of the *GMT* is a very intelligent diagram layout engine. It supports different operation types: automatic layout, semi-automatic layout and manual layout. By changing the layout style, it is possible to either quickly lay out complicated diagrams, make modifications to existing diagrams still complying to layout rules, or manually change layout with no constraints. The algorithms of the layout engine have been awarded several prizes.

First *MOLA* transformation language (used for driving the *METAclipse* transformations) editor was created, using *GMT*. All transformations for *METAclipse* version of the *MOLA* editor were initially created using this editor, following the bootstrapping approach.

1.5.4 GrTP

Graphical Tool Platform (*GrTP*, [39]) is the platform, ideas of which are the closest to ones behind the *METAclipse* platform. In fact, *GrTP* platform was developed in parallel with *METAclipse*, but it is based on different principles and is targeted at the implementation of a different kind of modeling languages. One of the main areas the *GrTP* platform is intended to address is support for the tasks related to ontologies and semantic web.

Main characteristic of the *GrTP* platform (and at the same time, the main difference from *METAclipse* platform) is its ideological independency of domain metamodel. In fact, *GrTP* platform does not require using the domain model at all, but instead operates (using model transformations) with the terms of presentation metamodel and events. However, if necessary, it is possible also to define the domain metamodel and create its instances upon some specific request on some user event (for example, for code generation, model import/export, etc.).

For transformations *GrTP* platform uses the *Lx* language series [40] developed in *UL IMCS* (*MOLA* transformation language also is compiled to *Lx* as a lower level language). *GrTP* uses parts of the *GMT* tool forming the base of the development; therefore it also possesses and even further enhances the graphical capabilities of *GMT*.

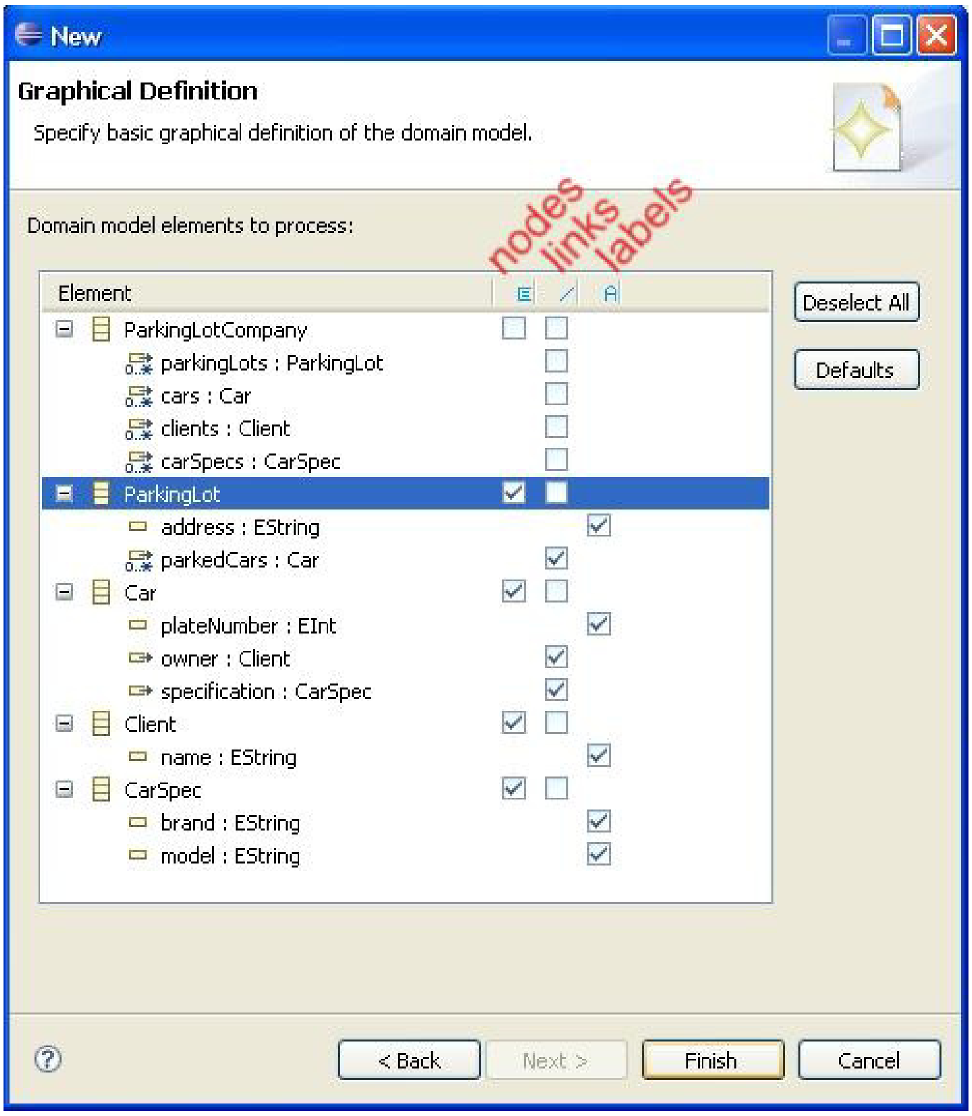
1.6 Eclipse-Based Platforms

This chapter will give a brief overview of the most prominent examples of the existing *Eclipse*-based *DSL* editor building platforms.

1.6.1 GMF

GMF is probably the most popular *DSL* editor building framework available for *Eclipse* platform. It provides a code generation component and a runtime environment for graphical editors, based on *Eclipse EMF* [41] and *GEF* [42] technologies. *GMF* uses *EMF ECore* as meta-metamodel and follows the static mapping approach for defining the correspondence between the domain and presentation metamodels. In fact, *GMF* utilizes three definition metamodels: graphical (defining types of the graphical elements), tooling (defining toolbar and menu elements) and mapping (defining the possible relations between the domain and graphical models). Additionally, *ECore* is used as domain metamodel. Instances of all these metamodels have to be created in order to define an editor in *GMF*. Code generated by *GMF* generator uses yet another metamodel: notation metamodel, though this metamodel is not visible to tool builders.

GMF provides a set of wizards for defining the mappings (see Fig. 7 for an example of graphical definition model creation wizard). Admittedly, wizards of *GMF* are not as convenient as symbol editor in *MetaEdit+* or *MS/DSL Tools* graphical facilities for mapping definition, however actual mapping possibilities in *GMF* are much broader. If *DSL* editors require more advanced logic, *GMF* allows also using *OCL* expressions, and if that is not enough, comprehensive *API*s are provided for extending the generated editors with required logic in *Java*.



Graphical Definition

Specify basic graphical definition of the domain model.

Domain model elements to process:

Element	nodes	links	labels
ParkingLotCompany	☐	☐	
parkingLots : ParkingLot		☐	
cars : Car		☐	
clients : Client		☐	
carSpecs : CarSpec		☐	
ParkingLot	✓	☐	
address : EString			✓
parkedCars : Car		✓	
Car	✓	☐	
plateNumber : EInt			✓
owner : Client		✓	
specification : CarSpec		✓	
Client	✓	☐	
name : EString			✓
CarSpec	✓	☐	
brand : EString			✓
model : EString			✓

Fig. 7. Graphical definition model wizard in *GMF*

Popularity of *GMF* lies in its integration in *Eclipse* platform, rich set of provided mapping possibilities and fact that it is an open-source product. However, even with the provided extension mechanisms, complicated *DSL*s cannot be easily built with *GMF*. If code has to be written in *Java* programming language, deep understanding of *GMF* internal architecture is required and non-trivial logic implemented. *GMF* will be discussed in more detail in chapter 2.3.

1.6.2 Tiger

Tiger [7] is historically the first framework offered for *Eclipse* platform that implements the transformation approach. Although this framework uses model transformations for definition of the correspondence between the domain and presentation metamodels, it also uses a specific domain modeling notation, which forces the domain metamodel of a *DSL* still 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 main goal of *Tiger* approach is to provide the building of syntactically correct diagrams only.

Currently there is a new version of *Tiger*, called *Tiger GMF* [8], which as the name suggests is based on the *GMF* framework. *Tiger GMF Transformation* project proposes to extend *GMF* by complex editing commands. The mapping between domain and presentation models is defined by standard *GMF* facilities, but new complex model editing commands can be defined by transformations acting only on the domain model. However, this approach does not permit to define more complicated (transformation-driven) mappings between the domain and presentation metamodels.

1.6.3 ViatraDSM

Another framework based on *Eclipse* environment implementing the transformation approach is *ViatraDSM* [9]. In fact, this framework already proposes some sort of combined solution involving both static mapping definitions and model transformations, which is a topic of further research also in case of *METAclipse* (out of the scope of this thesis). However, a lot of principal issues such as a generic mapping metamodel, seamless integration of static mappings with transformations and user-friendly mapping definition facilities are not solved there.

The *ViatraDSM* framework is based on the *Viatra2* transformation language [43]. In this framework a mapping from domain to *GEF*-level presentation concepts has to be defined. This static mapping is interpreted by the *ViatraDSM* engine. The transformation-driven mapping (defined by *Viatra2* rules) can be combined with the static mapping approach. However, the static mapping mechanisms support only very limited mapping possibilities. Only basic mapping patterns are supported. Mapping and transformation integration possibilities are very limited as well. Each object can be mapped using either transformations or mappings. Mapping definition for *ViatraDSM* framework has no adequate notation.

CHAPTER 2

Eclipse Technologies Used for the Development

As mentioned before, the *METAclipse* tool (developed as the result of the research of the thesis) is based on the *Eclipse* platform and uses a series of *Eclipse* technologies. In order to gain better understanding of the architecture and operating principles of the tool, this chapter will give an introduction to the most important of the utilized technologies.

Eclipse itself is open-source software aimed to create a highly integrated tool platform. The foundation of *Eclipse* development is formed by the so-called core project, which consists of a universal framework for tool integration and a *Java* development platform based on this framework. *METAclipse* also utilizes and extends the *Eclipse* core, but additionally it uses several *Eclipse* extensions. The most important of those are *EMF* (*Eclipse Modeling Framework* [41]), *GEF* (*Graphical Editing Framework* [42]) and *GMF* (*Graphical Modeling Framework* [4]).

2.1 Eclipse Modeling Framework

EMF is a modeling framework and code generation tool, which eases the design and implementation of structured models. Essentially, *EMF* is a runtime model repository that offers various services for operation with models, including model serialization. Originally *EMF* was created as an implementation of the *MOF* specification (to be more precise, an implementation of the *EMOF* subset of *MOF*), but its evolution took a slightly different path, based on the experience gained in the application of the *EMF* technology to a large set of tools.

EMF, similarly to *MOF*, provides a metamodel at the *M3* layer, called *ECore* (see Fig. 8). The *ECore* metamodel describes a superset of *EMOF* modeling concepts. The functionality of *EMOF* roughly matches *EMF* functionality. Differences are mainly in the naming of concepts rather than in their essence. It is even possible to read and write *EMOF* models with *EMF*.

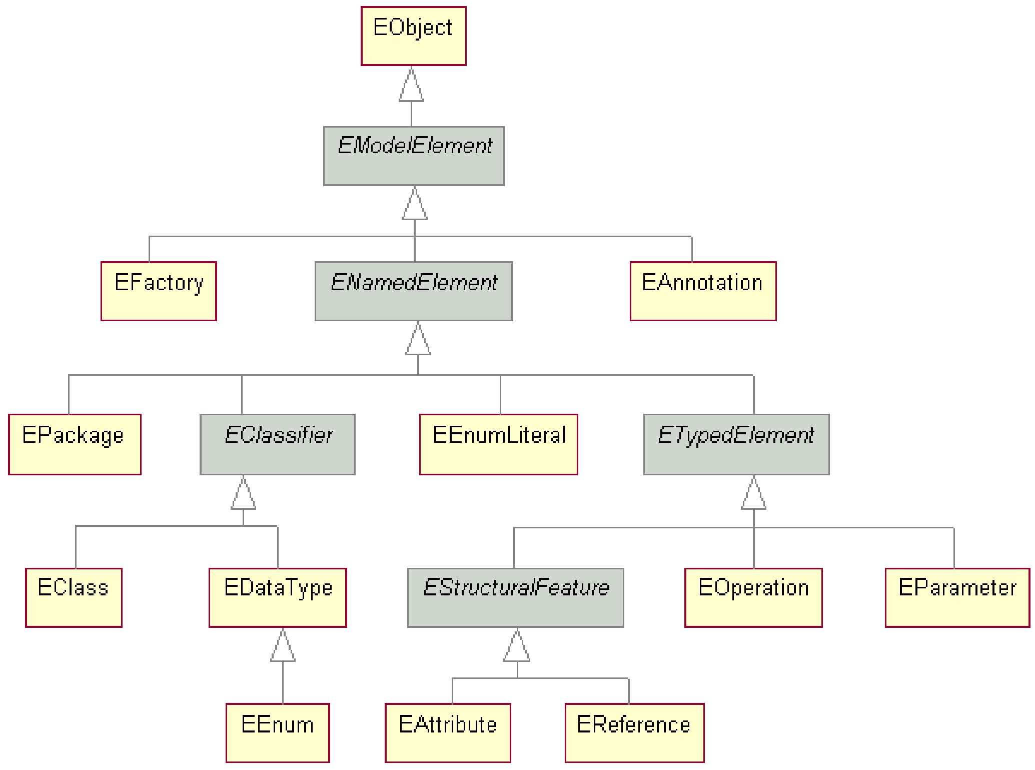


Fig. 8. Class hierarchy of the *ECore* metamodel

Fig. 8 shows a full class hierarchy of the *ECore* metamodel (gray classes are abstract). All *ECore* classes extend the base *EObject* class. *ECore* models are described using the following classes: *EClass* (class), *EDataType* (data type), *EEnum* (enumeration data type), *EEnumLiteral* (literal of an enumeration data type), *EAttribute* (attribute), *EReference* (association), *EOperation* (method), *EParameter* (parameter) and *EAnnotation* (annotation). In addition to these model description classes, *ECore* provides two additional classes: *EFactory* (helper-class for creation of model instances) and *EPackage* (helper-class containing the constants and methods for convenient access to model metadata).

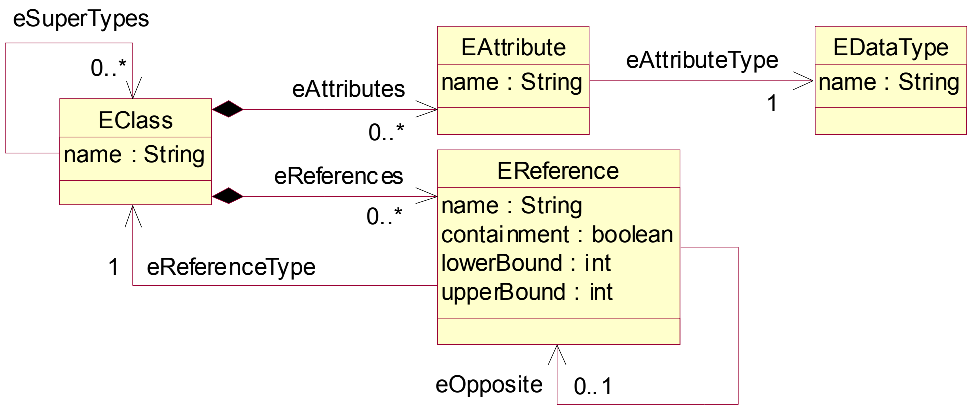


Fig. 9. Fragment of the *ECore* metamodel for describing a class

The most important *ECore* classes are shown in more detail in Fig. 9. These are the base classes used in metamodeling. As one can see, *ECore* metamodel stipulates that a model will consist of classes (*EClass*), each of which can extend an arbitrary number of other classes (*eSuperTypes* association) and contains an arbitrary number of attributes and references (*eAttributes* and *eReferences* associations). An attribute has a specific data type (*eAttributeType* association to *EDataType* class). A reference, on the other hand, has a specific type—one of the model classes—and two-way association is modeled as two references linked together with the *eOpposite* association.

Along with a clearly defined *M3* metamodel *ECore*, *EMF* offers also several supporting services:

- Code generation facility, which allows generating an efficient model implementation (a repository, specific to the runtime model) from *EMF* models. The *ECore* metamodel itself is generated using the *EMF* code generator.
- Dynamic *API* for operation on any *ECore* model (efficiency is not as high as with generated models, but it is possible to work with any model in a unified fashion).
- Model change notification mechanism allowing carrying out a simple and dynamic transfer of the model change events to various parts of the presentation engines.

- *XMI* [44] import / export.
- Additional services from other *Eclipse* projects: *OCL* [30] implementation, model validation service, transaction services etc.

All of the above-mentioned services together with clear and thorough application of well-defined standards have promoted significant popularity of *EMF* among users and developers. Therefore the most important gain from using *EMF* is compatibility and interoperability with other *EMF*-based tools. Taking into account that *EMF* fully supports all of the *METAclipse* needs, this was also the main reason why *EMF* was used for its implementation.

METAclipse is fully based on *EMF* modeling facilities and its metamodel (*ECore*), which is used as an *M3*-layer metamodeling language. In order to make the usage of *MOLA* transformations on *EMF* models possible, the author has developed a solution for external repository integration in the *EMF* environment (for more detailed explanation about the *EMF* extension developed, see CHAPTER 4 and paper [45]). This solution has allowed full application of *EMF* features for model handling in the *Eclipse* environment, additionally providing a possibility to utilize *MOLA* model transformations.

2.2 Graphical Editing Framework

GEF is a framework for the creation of graphical user interfaces, based on an existing application model. *GEF* consists of two components: *Draw2D* and *GEF API*. *Draw2D* is a component for graphical representation of the information, which provides an efficient *API* for drawing and laying out the graphical components, together with a series of services supporting various graphical tasks. *GEF API*, on the other hand, provides an infrastructure for graphical editing: *MVC* (*Model-View-Controller* pattern [46]) framework, *Undo/Redo* support, the concepts of toolbar (palette) and tools, etc.

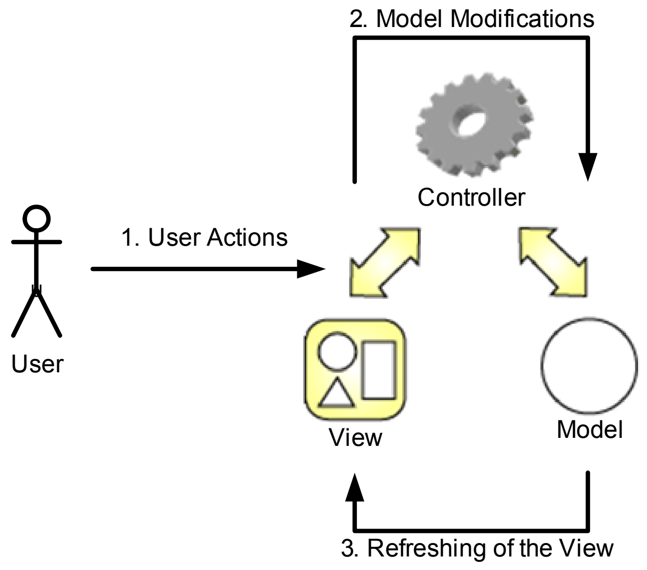


Fig. 10. Control flow in the *GEF MVC* framework

The *GEF MVC* (see Fig. 10) model and view are entirely independent from each other and can be changed separately. If the model gets changed, the controller refreshes the view (*GEF* uses *Draw2D* for the view). If the user makes any changes in the view, the controller translates them into the corresponding model changes. *GEF* ensures a unified scheme for graphical editing operations. The information being visualized is directly driven through the model, and the state of the model unequivocally defines the visual state of the view. User actions trigger changes in the model, which on turn trigger the refreshing of the view.

Because everything in *GEF* is driven through the model, the *GEF* framework is very appropriate for the creation of model-based editors. It is the main reason why it is used in the *METAclipse* platform. Additionally, *GEF* provides implementations also for such basic services as printing, overview window, tree view, toolbar, the concept of an action, undo support, etc.

From all the *GEF* instruments, *METAclipse* fully utilizes all of the *2D* graphics rendering infrastructure and the above-mentioned additional services. The author has additionally developed the necessary graphical elements for efficient drawing of graph diagrams based on the contents of the presentation models. The additions developed allow direct visualization of the presentation metamodel instances. This link between the visual representation and the presentation metamodel allows basing the driving of the tools on models and using transformations in the tools. Bare *GEF* does not offer the presentation metamodel, as it is positioned as model-agnostic technology.

2.3 Graphical Modeling Framework

As stated before, *GMF* is one of the most popular metamodel-based modeling tool building platforms in the *Eclipse* environment. *GMF* utilizes *EMF* for model handling and *GEF* for implementation of the user interface (see Fig. 11). *GMF* uses the static mapping approach for the definition of the correspondence between the presentation and domain metamodels (see chapter 1.4): at first the mapping models are defined, and then a modeling tool is generated, based on these models.

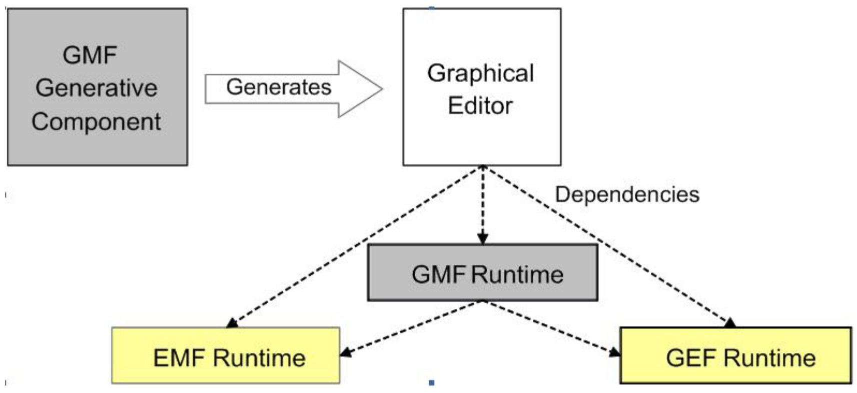


Fig. 11. Components of the *GMF*

GMF logically consists of two components: editor generator and runtime. The generative component uses several metamodels: the graphical definition metamodel, the tooling metamodel and the mapping metamodel. Additionally, *GMF* uses *ECore* as the domain metamodel (in *GMF* terminology; it is actually meta-metamodel in *MOF* terminology). The graphical definition metamodel defines the types of the graphical elements; the tooling metamodel defines the toolbar and menus, and the mapping metamodel defines the possible relations between the domain and graphical models. In order to define a tool in *GMF*, first all three models are created (four, if the domain model also has to be created), then the generative component is used and *Java* code generated (*Eclipse* plugin).

The created *Java* code (graphical editor) uses the *GMF* runtime in order to provide the basic functionality of the editor. The *GMF* runtime works on yet another metamodel: the notation metamodel. This metamodel in traditional terminology corresponds to the presentation metamodel and describes the graphical model instances during the runtime (graph nodes and edges, diagram element compartments, labels etc.). In fact, the *GMF* runtime is a graphical *Eclipse* plugin that significantly extends the *GEF* framework in the direction of diagram drawing. Additionally, this runtime provides diagramming tools with such services as export to image, printing, standardized style dialogs etc.

The *METAclipse* graph diagram engine partially uses the *GMF* runtime in order to reuse the services it provides (such as the diagram layout service). The metamodel of the graph diagram engine is created by taking into account the ideas of the *GMF* notation metamodel; however, it is made more convenient for use from transformations. In fact, *GMF* is the only technology of the utilized *Eclipse* technologies that already is based on metamodels, therefore is slightly more convenient for metamodel-based development. Unfortunately, it is not easy to use *GMF* runtime models directly in transformation-driven tools: they are created with the static mapping ideas in mind, which introduces quite significant constraints and makes development inconvenient. Therefore, for *METAclipse* the author has developed a metamodel that is much more suitable for transformations. For visualization of this metamodel, mostly pure *GEF* is used, reusing only several class libraries from *GMF* for the support of several services.

2.4 Other Presentation Frameworks

As described in previous chapters, the model runtime for the *METAclipse* platform is provided by *EMF*, and *GEF* together with *GMF* supports diagram drawing. However, *DSL* tools additionally require rich possibilities for displaying the domain elements in a tree, as well as for diagram and model element property editing. For these purposes *METAclipse* supplements the already mentioned *Eclipse* frameworks with two more frameworks, *Eclipse Tabbed Properties* and *Navigator*. The *Tabbed Properties* framework provides a standardized approach to editing of various object properties: the so-called property sheets. The *Navigator* framework provides the basic means for the representation of model elements in the project tree. *METAclipse* extends both these frameworks by defining a metamodel for both and allowing the control of their operation through the corresponding instances of the presentation metamodel.

None of the frameworks mentioned can be used directly in model-based tools, as they are not based on metamodels. The author has extended these frameworks by defining convenient metamodels for each of them and implementing the functionality that "translates" certain presentation metamodel instances to the corresponding states of framework elements (e.g., with the created extension of the *Navigator* framework, a given instance of the project tree metamodel is visualized in the project tree).

CHAPTER 3

METAclipse: Building Tools by Model Transformations in Eclipse

A graphical modeling tool has to handle a series of complicated tasks:

- Correctly visualize the domain elements;
- Provide intuitive and standardized editing support for them;
- Perform correct modification of the domain model elements according to graphical editing events;
- Provide convenient model navigation;
- Provide unified style for editing of the model element properties, etc.

The most complex and time consuming of the tasks in the development of a metamodel-based graphical tool are those associated with graphical representation and driving of the user interface. Fortunately, most of them are common to all graphical tools and can be solved already at the level of the tool building platform framework. *METAclipse* also provides an implementation for these tasks in platform itself; therefore, when developing a new tool, it is possible to concentrate on functionality that is specific to a given graphical tool.

This chapter will give a high-level description of the developed methodology for usage of the model transformations in the metamodel-based modeling tool building platforms. Also, an introduction will be given to the architecture of the *METAclipse*, *Eclipse*-based implementation of the methodology. The description of the main ideas of model transformation application in the tool building is given in the paper "*Building Tools by Model Transformations in Eclipse*" [11].

3.1 Methodology for Using the Model Transformations in Tool Building

As mentioned before, one of the results of the thesis is the methodology developed for the application of transformations in building of metamodel-based tools. This chapter will describe the schematics of the developed transformation application solution.

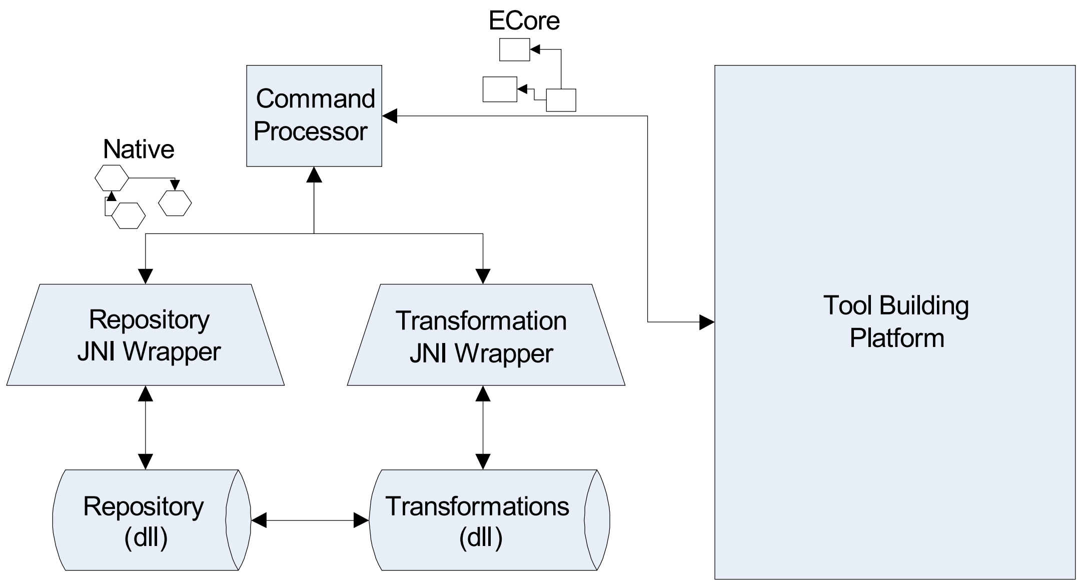


Fig. 12. Basic components of a transformation-driven tool building platform

The most important task is to define the boundary between the functionality of the graphical user interface of the tool building platform, which is common to all tools, and the logic of the tool itself, which in case of transformation-driven tools consists of the model transformations. The tool building platform consists of presentation engines, each of which is responsible for steering a separate part of the graphical user interface. Presentation engines must not depend on any functional logic specific to a certain tool. Model transformations, on the other hand, should not be forced to deal with tasks for creation of the graphical user interface. In order to define such clear separation, the developed methodology uses metamodels of the engines as an interface between the engines and transformations. To provide a functional link between the two sides, the concept of command has been introduced. Commands also are parts of engine metamodels and depict possible actions that can be received from the user (for example, the moving of a diagram element, or the request for a context menu by clicking the right mouse button on a project tree element). Such notion of a command allows defining the smallest unit of action that has to be processed by transformations. It must be noted that this is the simplest implementation scheme both from the perception and implementation points of view, which allows very clear separation of responsibilities between the presentation engines and transformations.

Therefore, in the developed methodology, one can basically speak of two main parts of the transformation-driven tool: the tool building platform, which provides the basic functionality for the assembly of the user interface and its various supporting services, and the command processor, which implements the functional logic of the tool and makes changes to the models in response to user operations performed in the user interface (see Fig. 12). Metamodels are used as an interface between the command processor and the platform: each part of the tool building platform has its own metamodel defined. All these metamodels together compose the so-called presentation metamodel. Commands also are part of the presentation metamodel. The platform provides all tools with the common functionality; however, the logic of a specific *DSL* tool is defined with the command processor written specifically for the tool. The command processor is nothing else than a library of transformations, which are invoked according to the commands passed to the processor for processing.

Commands are created as a result of any semantic user action (semantic actions are only those that trigger any changes in the domain model, for example, the creation of a new element, the change of the text in one of the displayed elements, the drawing of a new connecting line, etc.). The newly created command is passed to the command processor for processing. The command processor is responsible for changing the domain and presentation metamodels according to the command passed, therefore providing the tool with already specific functionality.

Fig. 12 shows the transformation usage scheme specifically for the *METAclipse* platform: the platform itself is based on *Eclipse,* and the presentation metamodels are defined using *EMF ECore*. Transformations, on the other hand, are written in the *MOLA* model transformation language, which operates upon an external model repository. Both the external repository and *MOLA* compiled programs are *DLLs* (*Dynamic Link Library*), which are called from the command processor through *JNI* (*Java Native Interface* [47]). In order to organize communication between the *DLL*s and the platform, all models are being translated to the model format understood by the external repository. The description of this translation can be found in CHAPTER 4.

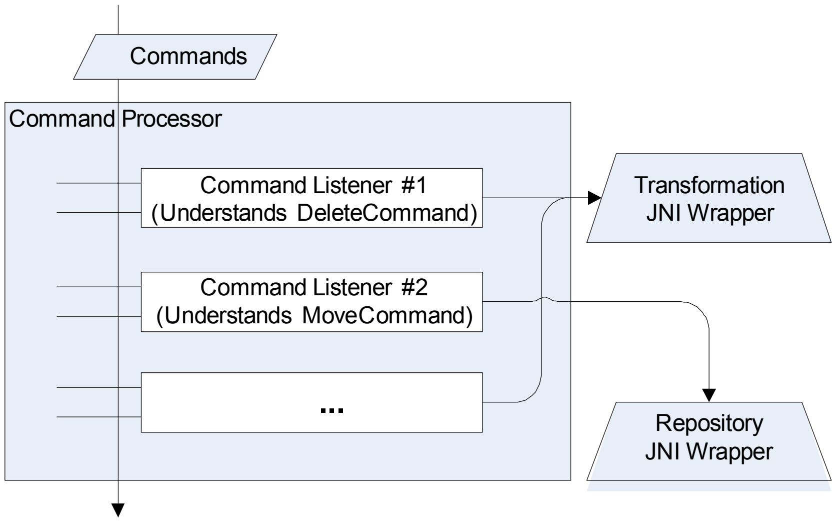


Fig. 13. Operation of the command processor

Fig. 13 demonstrates the operational scheme of the command processor: by passing a specific command to the command processor, a corresponding command listener is found and invoked. Some listeners do not require an invocation of the transformations, as they do not make any changes in the domain models (so, for example, the moving of an element, i.e., changing its coordinates, impacts only the presentation model). In case of *METAclipse* it means that instead of invoking the transformation *JNI*, changes are written directly to the repository through the repository *JNI*. Semantic commands (such as the delete command), on the other hand, require invocation of transformations, because they imply changes in the domain model. Only transformations know how the domain model has to be changed in response to such a command.

The methodology of the application of transformations in tool building platforms does not depend on the implementation: the platform could just as well be based on *C++* and use completely different metamodeling means. Similarly, the transformations could be written in other languages. The basic idea of the methodology is to define the platform interface with clearly defined metamodels and include all of the actions available to the user as part of the presentation metamodel. Interaction with transformations is then organized only through the instances of these metamodels. The command processor is processing the received commands (presentation metamodel element instances) by correspondingly changing both the presentation and domain metamodels. Transformations are completely responsible for the synchronization between the domain and presentation metamodels, because the platform itself does not know anything about the domain metamodel and does not take care of its processing.

3.2 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. 14). *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 chapter 5.2.

METAclipse plugins contain all the common functionality needed for the tools and relieves the creator of the tool of a 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.

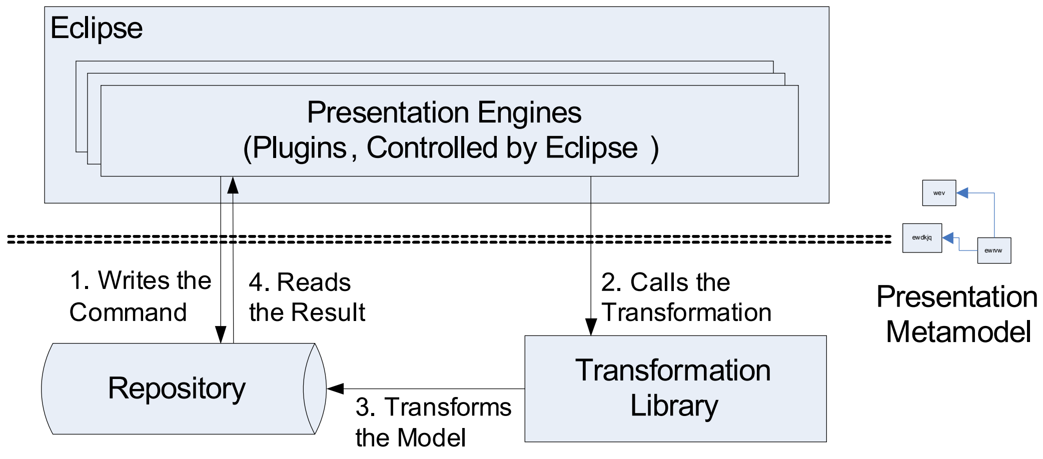


Fig. 14. High-Level view of the *METAclipse* architecture

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. In the thesis the combined metamodel of presentation and domain metamodels will be referred to as *METAclipse* metamodel. Accordingly, 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 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 chapters of this thesis. 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. 14);
- The main model transformation recognizes the command written and makes the necessary changes to the presentation and/or domain models (step 3 in Fig. 14);
- Presentation engines read the model changes and react accordingly: show context menu, show newly created element or edge, etc. (step 4 in Fig. 14).

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. 15. 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.

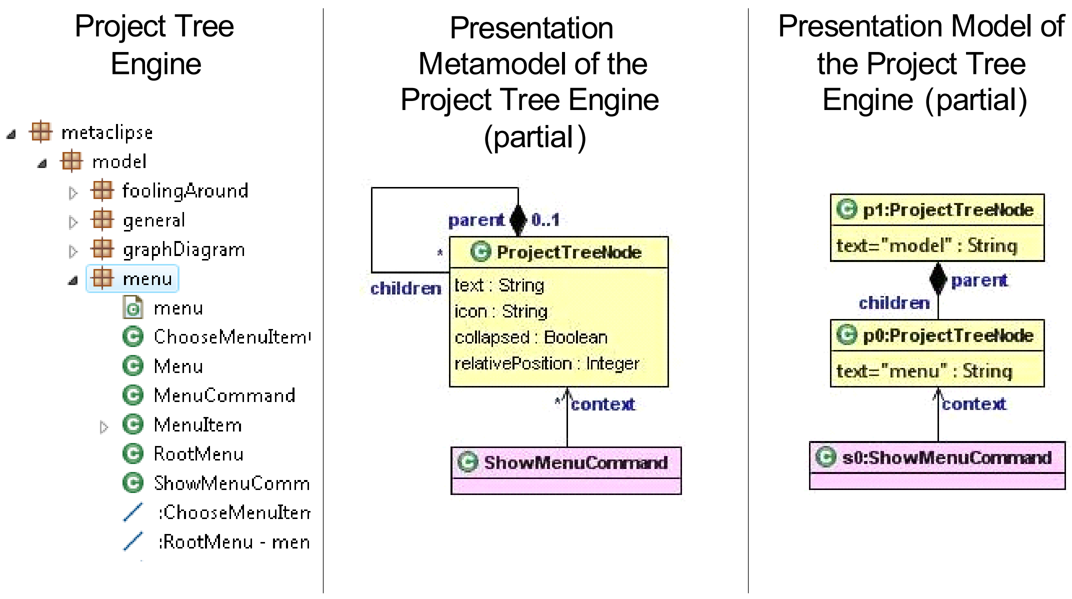


Fig. 15. Example of a project tree engine and its metamodel and model states

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. 14). At the right side of Fig. 15 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. 14). The transformation will find that *ShowMenuCommand* is written in the repository and will create presentation metamodel instances (not shown in the Fig. 15) comprising the needed context menu (step 3 in Fig. 14). 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.”

3.3 Application of Model-Transformation-Based Approach in METAclipse

Let us assume that a domain is given for which we need to build a tool, e.g., the Class domain in *UML2*. In *METAclipse* the domain metamodel is specified as an *EMOF*-compliant (*Essential MOF*, see [23]) *UML* model (similarly to the *ECore* model in *GMF*). The presentation metamodel is predefined in *METAclipse* (an equivalent to the notation metamodel in *GMF*). It describes the available graphical elements in a diagram: nodes, edges, text labels, and other elements. More precisely, an instance of this metamodel is a runtime graphical model, which is visualized and serviced by the *METAclipse* presentation engine. Transformations build instances of the presentation classes and connect them by mapping links to the corresponding domain instances. The presentation engine visualizes these presentation instances and, in addition, notifies transformations when the user has performed some action. A high-level graphical model transformation language *MOLA* [29, 58] is used in *METAclipse* for building the required transformations, but in principle other such transformation languages could be used too, e.g., *MOF QVT*. There is no generation step in *METAclipse* (certainly, *MOLA* transformations must be compiled to an executable form).

Fig. 16 shows the most interesting part of the presentation metamodel. In fact, the complete metamodel of the world transformations have to work in consists of the predefined presentation metamodel (described in greater detail in CHAPTER 5) and the required domain metamodel (of which only the *Element* class is visible in the figure). For *UML*-related domains (such as the above mentioned Class domain) and many other it is typical that there is a common ancestor for all domain classes (in *UML* it is *Element* in the *Kernel* package). The association *domain–present* here serves as a generic mapping link between domain and presentation.

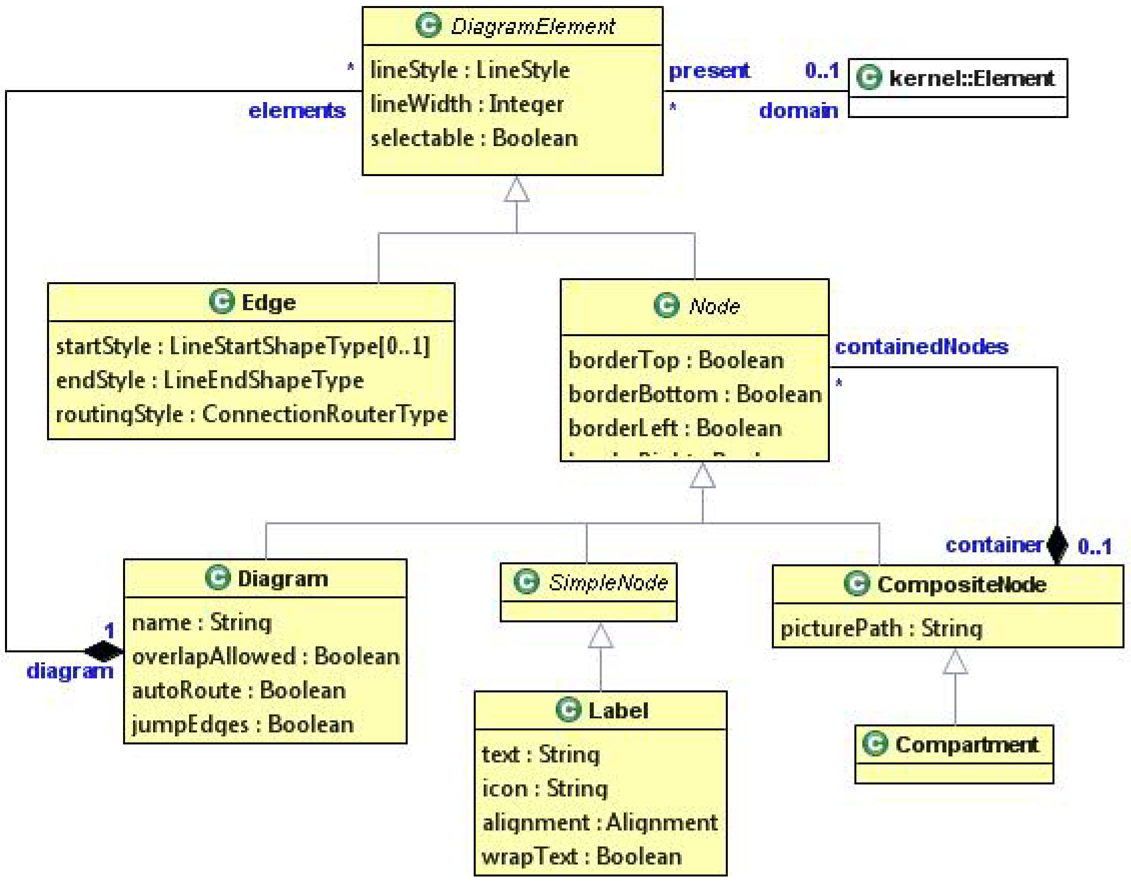


Fig. 16. Graph diagram metamodel

To implement the abovementioned schema, a transformation procedure must be built for each kind of diagram element (node, edge, or sub-diagram) representing an essential domain element. For example, such procedure has to be created for a class node (representing a domain *Class*). In particular, this transformation builds a new class node (an instance of *CompositeNode*) when a *Class* instance in the domain is to be shown in a class diagram.

Though building the transformation procedure for each diagram element may initially seem to require much more effort than just defining a static mapping in *GMF*, the job is actually quite easy in an appropriate transformation language. It should be reminded that for *DSLs* we are interested in, nearly each static mapping in *GMF* would need to be complemented by a relevant piece of *Java* code after the generation step. The transformation procedure can also collect in a cohesive block the various constraints relevant to a diagram element. These constraints become much more readable in *MOLA* than they would be in a mix of *OCL* and *Java*. The main gain, however, is flexibility.

Besides the domain-determined specific functionality for each diagram element, a lot of functionality in a tool (setting an element style, displaying menu items, moving an element etc.) is actually common to all elements, and therefore needs to be built only once, as part of the framework. The presentation engines in *METAclipse* perform all generic presentation and graphics-related jobs, such as moving an element. In order to maximize this common part in transformations, the presentation metamodel must be used in an appropriate way, especially the “scaffolding part” of it (see chapter 5.7).

CHAPTER 4

A Proxy Approach to External Model Repository Integration in Eclipse EMF Infrastructure

An important result of the thesis is the developed universal solution for external model repository integration in the *Eclipse EMF* environment. The need for the development of such a solution occurred because the *MOLA* transformations used in *METAclipse* operate on their own external repository, and there was a need for synchronization of the data between this repository and *Eclipse*. However, the developed solution is sufficiently generic to be applicable also for the integration of other external repositories; therefore the solution can be of use also in other tools. There could be an interest in the integration of a repository in *EMF* simply for the reason that *EMF* provides very wide model processing capabilities and various services (*XMI* serialization, model validation, code generation, model transformations etc.). These services can supplement the functionality of already existing tools with new possibilities. On the other hand, integration increases also the value of the *EMF* tools by giving access to services available only through the external repository.

In this chapter a direct integration of external model repositories in the *EMF* infrastructure is described. The presented approach is based on the application of the proxy pattern to extend the functionality of *EMF* base objects and provide a runtime synchronization of the model data with the repository. The approach allows existing applications to interchange the model data seamlessly with *EMF*, thus giving access to the services offered by *EMF* technologies. On the other hand, *EMF*-based applications can benefit from the services provided by the external repositories and applications (for example, efficient model-to-model transformation implementation) without a need to adjust the application code. Applicability of the introduced solution is analyzed at the end of the chapter.

4.1 Motivation for the External Repository Integration in EMF

Necessity to integrate the *MIIREP* repository in the *EMF* environment is clear. The methodology itself, however, can be used more generally for integrating an arbitrary *MOF*-isomorphous model repository. Therefore, it is important to understand the general motivation for such integration.

EMF is widely used as a tool for the implementation of a structured model. However, this is not the only functionality the family of the *EMF* technologies has to offer. By introducing an efficient and standardized approach to model handling, *EMF* has promoted the evolution of various *EMF*-based projects supporting the model-driven engineering process (like *EMF Query* [48], *EMF Transaction* [49], *OCL* implementation [50], validation component [51] and various transformation language implementations, e.g., *ATLAS* Transformation Language *ATL* [52] for model-to-model transformations and *JET* [53] for model-to-text transformations).

Other *Eclipse* frameworks and tools are built to operate on *EMF* models allowing rich graphical editing of the models (like *GMF* [4]). There are number of practical *Eclipse* applications built that even further extend the *EMF* model handling possibilities and applicability of *EMF* models. Among those are various persistency and *O/R* mapping solutions (like *CDO* [54] and *Teneo* [55]), *MDSD* supporting framework *openArchitectureWare* [56] and even commercial development and design tools like *IBM Rational Software Architect* [57], etc. The stack of the technologies in the *EMF* family and tools operating on *EMF* models is growing continuously.

Having such a rich set of services available, *EMF* is an appealing environment for model handling. Existing applications can benefit from allowing their models to be transferred to *EMF* and back. For example, such interoperability could add missing features to existing model environments when needed, like *XMI* [44] model serialization (provided by *EMF* as default serialization mechanism), possibility to validate the model against defined rule set, possibility to use code generation functionality or model-to-model transformations, possibility to develop graphical model editors, etc.

Not only existing applications can benefit from being integrated with *EMF*. Another very important aspect of external repository integration in *EMF* is the possibility to offer additional specialized services to *EMF*-based tools. In fact, this is the case of the *MIIREP* [13] repository integration in *EMF* in order for *Eclipse* and *EMF*-based tool *METAclipse* to gain the access to *UL IMCS* model-to-model transformation engine. This allowed using the transformation languages *MOLA* [29, 58] and *Lx* [40] in *Eclipse* environment. Transformations are compiled to *C++* code and work on *MIIREP* repository that is specialized particularly for efficient execution of the operations needed by transformations. By this integration the performance was gained that was needed for transformations to work on huge models in a very efficient way.

The alternative to integration would have been the transferring of the *MOLA* transformation language to *Java* so that it worked on *EMF* objects directly. This, however, would mean massive work on new implementation of *MOLA*, and would not guarantee that it would be possible to measure up with the efficiency of the *C++* implementation. So, integration of the existing repository was most reasonable choice. Also, it would not be sufficient to have only the model import/export functionality, as in case of *METAclipse* the interaction with repository is very dynamic. Every user interaction with *METAclipse* results in the execution of some transformation, so a very rapid access and change of repository objects is required from both *Eclipse* editor and *MOLA* transformations.

Another motivation for integration worth mentioning is the possibility to unite the *EMF* with different other model-handling frameworks like *MDR* [59], *MS/DSL Tools* [5], *Generic Modeling Environment GME* [20], *Fujaba* [32]. All these frameworks are meta-model-based and their meta-meta-models provide similar capabilities to *EMOF*. They all can handle models similar to *EMF* and each provides distinctive features for model handling. For example, *GME* provides advanced facilities for building model-based simulators and debuggers, while *MS/DSL Tools* provides easy integration with *Microsoft* technologies, etc. The features of each framework can turn useful for *EMF* models and thus there are good reasons for uniting them. There already exists such an attempt: *Eclipse* project *GEMS* [26] binds the *GME* to *EMF*.

4.2 Integration Solutions

In general, the tool integration problem has been a topic of discussions and publications for quite a long time. A survey [60] shows that the tool integration topic is very wide. Most of the covered papers discuss the integration problem generically. The presented solution for external repository integration in the *EMF* environment is comparatively specific and suitable specifically for the *EMF* architecture.

In order to make models of existing applications accessible in *EMF*, there is a need to map the meta-metamodel (*M3*) concepts of the application to the meta-metamodel of *EMF*, namely *ECore*. If the *M3* layer of the external modeling environment can be mapped to *ECore*, it is possible to transfer the model and metamodel data between the external application and *EMF*.

There already are some examples of model interchange between *EMF* and other technologies by providing the import and export of models. In the simplest cases it is done through some external format supported by both *EMF* and the external repository (like *XMI*), but other solutions make use of the native repository *API*s. Some of them are integration of *ARIS* [61], *MS/DSL Tools* [36], *GME* [62] and *EMT* [63]. All of the mentioned solutions map the concepts of the named technologies to the concepts of *EMF* as their *M3* layer is close to *EMOF*.

There is one significant problem with the import / export approach. In this process, the model is first exported, resulting in a copy of the model. Then changes to the copy are made, after which the modified copy is imported back in the model. If the whole model is transferred, this process is not complicated. However, usually models are big and it is inefficient to transfer them in their entirety. Normally, only a subset of the entire model needs to be exported for external modification. In this case, a huge problem is the merging of the transformed sub-model back into the original model. The main problem is that there can be references from the unmodified parts of the model to some parts of the model that have been deleted or changed. These references need to be traced and modified; sometimes perhaps redirected to newly created elements. This is not an easy task and requires knowledge of both the original and the modified models, and sometimes even about model transformation logic.

Import/export approach can support the integration needs if the model transfer from one technical space to another is relatively infrequent (for the batch processing of the models). If more rapid model data interchange is needed, other integration solutions should be considered. Also in the case of *METAclipse* and the *MOLA* external repository it was not possible to rely on complete export and import of the models, because normal operation of the platform requires a high speed of data transfer, which means that maximally optimized information volume has to be transferred between the repository and *EMF*.

The developed solution integrates the external repository directly in the *EMF* environment, by using the *API* of the repository. This solution uses lazy data loading and synchronization (only the relevant data is transferred to *EMF* and back) and dynamically integrates with *EMF* (operations with external models are performed during the runtime). This solution does not pose any problems with model merging, as the changes are made directly in the original model and model export, import or merging is not required.

The main idea behind this approach is to alter the original implementations of the core *EMF* objects in the way that they start acting as proxies to the external repository and every operation on the *EMF* model is redirected to the corresponding operation(s) on the external repository. Any changes done to the model at the runtime outside the *EMF* are properly notified to listeners through the *EMF* notification *API*. Additionally, a small change is made to the code generation facility of *EMF* in order for the generated code to use the proxy implementations of *EMF* objects instead of original ones.

Summing it up, the presented integration approach allows existing applications to gain the benefits of the services offered by the *EMF* tools and vice versa. For example, in the context of transformation languages, applications not offering transformation languages can use the transformation languages operating on *EMF* models. On the other hand, *EMF* tools can use the transformation languages offered by external applications in order to gain the efficiency and performance.

4.3 Objectives of the Integration

The goal to be achieved with the proposed repository integration solution is to provide a bridge between the external repository and *EMF* that would possess the same characteristics as import/export bridging solutions (possibility to transfer the model data from the external repository to *EMF* and back), but at the same time would provide some more sophisticated features. Main additional feature wanted is the ability to carry out the transfer of the model data dynamically as the models are changing during the runtime. There has to be a possibility to synchronize models between the external repository and *EMF*, propagating changes done in either of sides to other in real time. It must be possible to carry out the synchronization in both directions:

- If the change is done to the synchronized model directly in the external repository by some external application, it must be transferred to the *EMF* and proper *EMF* notifications should be called;

- If the change is done to the model by the *EMF*, it must be transferred also to the external repository.

That being said, it must be noted that no objective has been established to allow simultaneous changing of models by external applications and by *EMF* – the presented solution presumes that if there will be changes on both sides, they will be sequential, but never parallel and no concurrency is supported.

Another aspect to be considered is that we do not want to impose any additional requirements to the applications using the *EMF* code. This means, *EMF* interfaces has to remain intact and applications already using the *EMF* classes should not have to change significantly if it was required for them to synchronize their model data with an external repository.

4.4 Applying the Proxy Pattern

Taking into account the aforementioned goals, an appropriate method for the implementation of the external repository integration in *EMF* is the proxy pattern, as it will be shown in this chapter. The following figure (Fig. 17) shows the structure of the proxy pattern as defined in the *Design Patterns* [64].

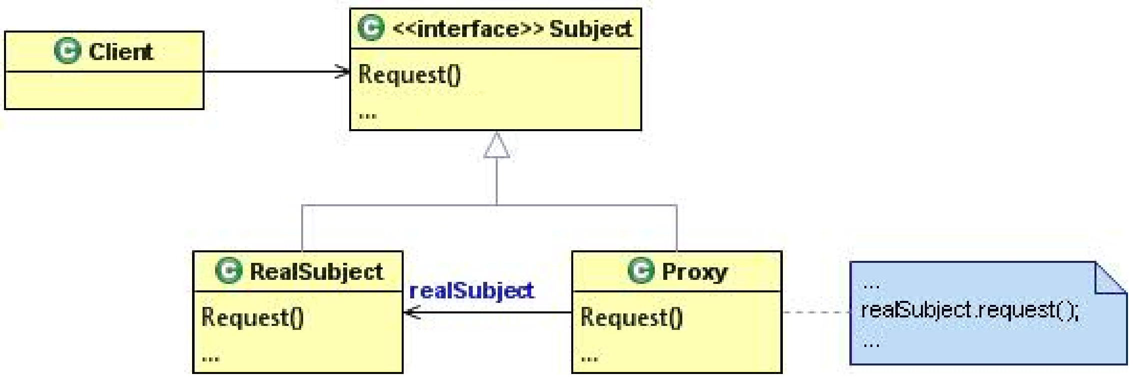


Fig. 17. Structure of the proxy pattern

The basic idea of the proxy pattern is to provide a façade for another object in order to control the access to it. Demonstrating the proxy approach sketched in Fig. 17, let us say a client is accessing some objects method. Let us call this object a "subject" (class *RealSubject*) and method called, a request. The access to the object methods is organized through an interface *Subject*. Now, if we want to control the access to the *RealSubject* object, proxy pattern suggests to add another object—*Proxy*, which implements the *Subject* interface and delegates the request method calls to the *RealSubject* class, adding necessary pre- and post-processing. When client will call the request method of the interface, the *Proxy* object will be called instead of *RealSubject* and the necessary control will be injected before calling the *RealSubject* request method.

As *Design Patterns* book suggests, most typical cases when proxy pattern is used are when:

- It is necessary to provide a local representation of a remote object (*remote proxy*);
- Objects are expensive to create and should be created on-demand, or in other words lazy-loaded (*virtual proxy*);
- Additional checks or tasks have to be performed upon access of the object (*protection proxy* and *smart reference* respectively).

Relating this to the goals, we want the *EMF* to act as a façade to the external repository and delegate the calls to the external repository *API*. It is quite natural to apply the proxy pattern for these needs. To be more specific, what we need is a remote proxy with the features of the virtual proxy. The utilization of the remote proxy is obvious. The virtual proxy features are needed as models tend to be very big and we want to transfer to *EMF* only those objects that are really needed. Additionally, for increased performance, the caching mechanism needs to be implemented so that subsequent access to the object properties would result just in one call to the repository *API* functions. See the Fig. 18 for the class diagram of the proxy pattern adjusted to integration needs.

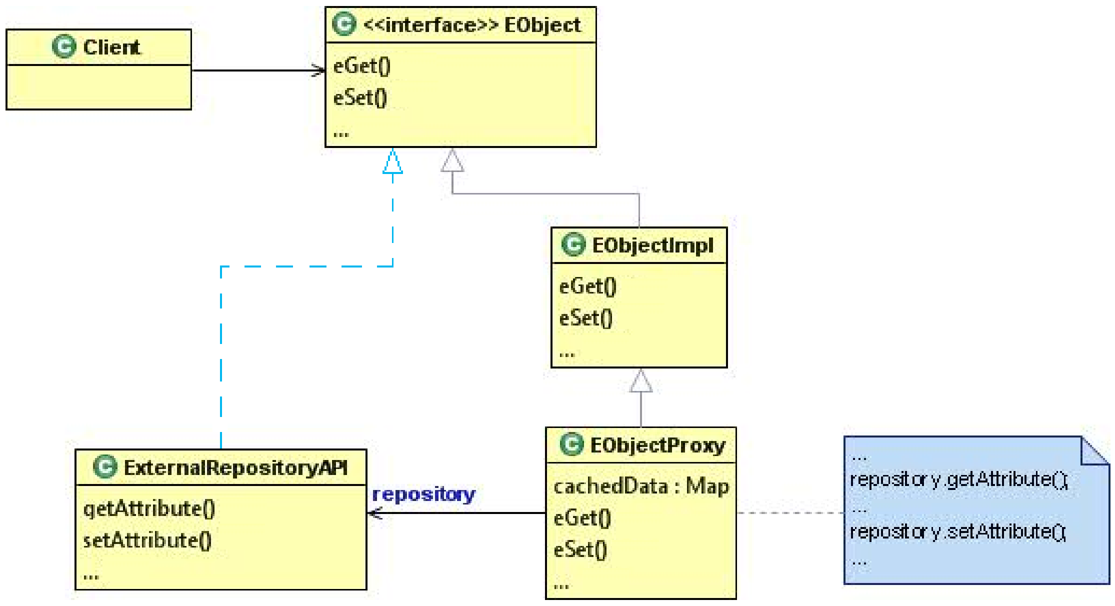


Fig. 18. Structure of the proxy pattern applied for the bridging of *EMF* and the external repository

The figure depicts only high-level structural elements and next chapter will discuss the implementation in more details. In the figure, the client is accessing the *EMF* object interfaces (only the root interface *EObject* is displayed with basic exemplary methods *eGet* and *eSet*, but it could be any sub-interface of *EObject* in *ECore* metamodel or any generated *EMF* class interface). The *EMF* interfaces correspond to the *Subject* interface in the basic proxy pattern.

Application of the proxy pattern is eased a lot because of the flexible *EMF* architecture. As *EMF* has a top-level object defined in its metamodel, namely *EObject*, it is enough to provide the proxy implementation for this object to get the proxy functionality spread throughout all *EMF* metamodel implementation classes. Extension *EObjectProxy* of the *EMF EObject* interface implementation *EObjectImpl* acts as a proxy to the external repository *API* and corresponds to the *Proxy* class in the basic proxy pattern. This class implements the “remote” and “virtual” features of the proxy pattern by delegating the calls to the *API* of the external repository and providing a *cachedData* map that is consulted before calling the actual repository *API* functions.

Finally, *ExternalRepositoryAPI* class corresponds to the *RealSubject* class. The difference between the variation shown in Fig. 18 and original proxy pattern is that the external repository *API* does not implement the same interface as proxy (*EObject* interface). It is possible that some calls to the *EObject* will result to multiple calls to *ExternalRepositoryAPI*, possibly even with some model data transformation involved. However, external repository *API* cannot be absolutely arbitrary. It must operate with the same concepts as *EMF*, i.e. its capabilities must be isomorphous to *EMOF*. Therefore, it can be said that it "virtually" or "isomorphically" still implements the same interface as proxy.

By applying plain proxy pattern we can solve the synchronization problem just in one direction—from *EMF* to the repository, but not the other way around. However, changes done in the external repository by external applications must be transferred back to the *EMF*, as both sides can actively change the models. For this reason, proxy pattern has to be augmented to incorporate some change notification mechanism. Such mechanism will be described in the next chapter (particularly, chapter 4.5.3) together with more technical details of application of the proxy pattern.

4.5 Implementation of the Proxy for EMF: "Wise" Objects

Now, when we have established how to apply the proxy pattern, we can proceed to the technical details how the actual proxy to the external repository has been implemented. The description is given, based on the experience gained while integrating repository *MIIREP* [13] with the *EMF*-based graphical model editing tool *METAclipse* [11], where the proxy approach to integration is already successfully implemented and working. The actual implementation of the *EMF* proxy will differ from repository to repository, as there will be differences in repository *API*s, still the concepts of the integration will remain the same. Technical description of *METAclipse*, including some specific details about *MIIREP* and *EMF* integration is given in CHAPTER 5.

In case of integration of *MIIREP* in *EMF*, the changes to the model can be done in both model transformations working as external applications directly with the repository *API* and *METAclipse* tool working with *EMF* representation of the model. Therefore, both kinds of the synchronization are involved—from the repository to *EMF* and vice versa.

4.5.1 External Repository API

The prerequisite for being able to integrate a particular repository with the *EMF* is the existence of a *Java API* for the repository. This *API* will be used by the *EMF* proxies to perform the synchronization operations with the repository. Only repositories that provide *API* capabilities similar to *EMOF* can be integrated with *EMF*. Therefore, the repository *API* must cover 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.

In case of the *MIIREP* integration, the *Java API* was not originally available, as the repository is implemented in *C++*. But, from the functional point of view it provided all necessary operations that were required. In order to provide a *Java API*, a *JNI* [47] wrapper of the *MIIREP* repository was created.

4.5.2 "Wise" Objects as an EMF Extension

EMF 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. So, base classes are the best place where the repository synchronization should be implemented and, as it has already been roughly sketched in chapter 4.4, *EMF* proxies are implemented as extension of original *EMF ECore* objects, providing an alternative *EMF* runtime.

New proxy objects 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. For *EMF* tools "wise" objects can be considered a 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.

Base *ECore* classes were extended and a set of "wise" object base classes was defined (see Fig. 19). 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 chapter. 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.

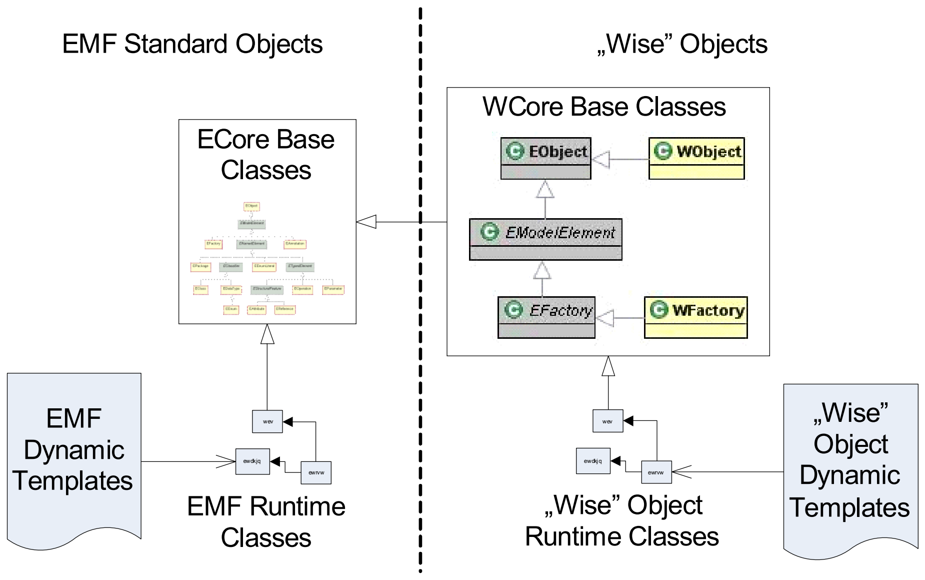


Fig. 19. "Wise" object dependencies

The fact that the parent of all *ECore* classes is a single class—*EObject* (see [41] 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.

To put the *WCore* classes in action also for the generated code, the *EMF* generator had to be extended so that it produced "wise" objects extending *WCore* base classes. The *EMF* framework uses so-called dynamic code templates (using another *Eclipse* framework for the code generation—*JET* [53]) 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. 19). Default templates producing EMF runtime classes were extended so that they would generate the code using *WCore* instead of *ECore*. The *JET* template extension is not required if only runtime *EMF* objects are used.

The complete set of classes comprising the *WCore* can be seen in Fig. 20. 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 *EMF* side and if any occurs, writes the data to the repository.

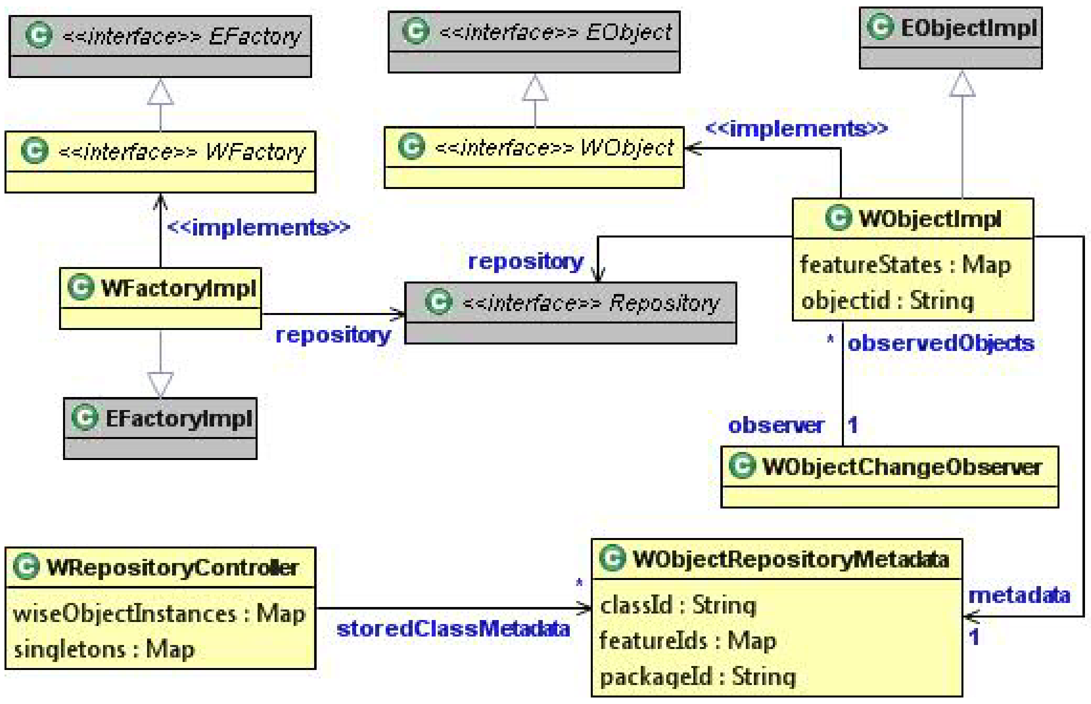


Fig. 20. *WCore* class diagram

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* layer and thus it is needed 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).

The two objects directly responsible for the synchronization of the model in repository and its representation in *EMF* (*WObject* and *WObjectChangeObserver*) act on the events of reading or changing the model information through the *EMF API*. When any operation on the model is performed, it translates the *EMF API* call to the corresponding call(s) to the repository *API*. It is easy to do this if the repository relies on the metamodel that is very close to the *EMOF*. However, the less the repository *API* resembles *EMOF*, it becomes harder to map the *EMF* calls to it and more intelligent transformations are necessary.

4.5.3 Repository Change Notification

Extending the *ECore* base classes covers the synchronization needs only from the *EMF* perspective, i.e., if changes to the model are done from the environment working with *EMF* classes (wise objects). However, model changes can happen also on the other side (in case of the *MIIREP* integration in *METAclipse*, most intense changes to the model are done by the transformations in the repository directly). So, besides the proxy pattern applied to *EMF* objects, another missing piece is a change notifier back from the repository, which would trigger the *EMF* change events for all objects that have been changed.

The 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 they have been carried out, as it means inspection of all object instances in the repository. This means that a support from the side of the repository or the tool performing the changes is required in order to make an efficient implementation of the change notification.

In *WCore*, the *WRepositoryController* class takes care of the repository model change tracking. There, a special method is defined for change detection, which has to be invoked after each change done to the model at the repository directly (who calls this method depends on how the integration of the external repository is used). The implementation of the *WRepositoryController*, however, is strongly dependent on the possibilities offered by the repository being integrated.

For each repository the change tracking mechanism will be different, as the possibilities of detecting changes will differ from one to another. Worst case would be if the external change source would do unpredictable changes in the repository and the repository itself did not provide any change tracking mechanism. In this case there is only two options: introduce a layer between the external tool and repository that will implement the change tracking mechanism or, if the performance requirements allows it, do the full re-scan of all model elements residing in the repository and detect which elements and how have been changed. One possibility for the implementation of the change-tracking layer would be to use aspect-oriented programming (*AOP*) in order to execute the change tracking code before or after the repository *API* function calls.

Slightly better situation would be if some kind of an algorithm existed that could limit the number of the model elements to consider while detecting the changes. Best, however, is when it is possible to rely on repository-native service that would allow us to explicitly detect or monitor the changes by either defining the listeners on the repository objects or calling some method that would return us the set of the changes.

The various scenarios how the change detection can happen 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* then 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.

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. Each transformation is responsible for maintaining the lists of the changes to be returned through the repository interface to the *WRepositoryController*.

4.6 Applicability of the Presented Approach

As already mentioned, main force that drove the development of the presented approach, was the necessity for the use of an external repository in the *METAclipse* tool. This demonstrates the case when the integration approach discussed here is applied in order for *EMF* to gain some extra features provided by an external repository (i.e., the possibility to invoke *MOLA* transformations on *EMF* models). *METAclipse* editors are driven by model transformations that are executed on every user action in the editor (even a mouse click on some model element invokes a transformation). This and the fact that models being edited with the *DSL* editors tend to become fairly large (even millions of instances) raises very high efficiency requirements to the transformation engine. For transformations to work efficiently, it is important to have an appropriate repository with operations that used by transformations fine-tuned to give maximum performance. *EMF* itself lacks the functionality required for efficient implementation of operations like pattern matching. Therefore, for an efficient transformation engine implementation, it is required to extend the *EMF* to add the functionality for efficiency of the pattern matching.

In similar situation, *Tiger* project [7] team has chosen to redesign their graph transformation language *AGG* [65] and transfer it to *EMF*. In case of *METAclipse*, there already was a very efficient repository *MIIREP* [13], specialized for transformation languages and capable of handling huge models, and a stable and efficient transformation language *MOLA* [29, 58], working on this repository. It was more natural to integrate the named repository into *EMF* rather than taking the road of redesigning the *MOLA* language to work in the *EMF* environment and extending the functionality of *EMF*.

Another example, similar to the case of *METAclipse*, would be integration of the external simulation engine functionality (such as available in *GME* framework) into *EMF*. For example, if the graphical plugins of *Eclipse* are used for visualization and animation and external libraries for computation, there is a need of rapid model data interchange between the *EMF* and the external model storage.

Another applicability domain where presented solution would be useful is for augmentation of the possibilities of the existing tools with the features provided by the *EMF* technology family. Papers [36], [61] and [62] demonstrate that there is a real need for such integration. Mentioned papers use the import/export approach with transformations involved in metamodel mapping from one technical space to other. This approach was natural for the problems addressed, as all three are examples of typical batch transfers of model data.

Things, however, get more complicated if there is a need to transfer only a part of the model and merge the changes back. For example, if a *MDSD* transformation is applied to a sub-model, the results must be integrated in the common design model of a system. In this case the integration of the results requires some non-trivial reasoning how to preserve the integrity of the complete model. The approach presented here could be adapted to fit the needs of this use-case. The practical benefits in this domain, however, still have to be investigated, as the application of presented approach for enrichment of the external applications has not yet been practically verified.

It must be noted that the applicability of the proposed solution is constrained with the need for the external repository *API* to provide functionality that would cover all the capabilities of the *EMOF*. If the concepts behind the repository are not compatible with *EMOF* (meta-meta-models at *M3* layer are not close enough), it is not possible to apply the presented approach. Also, there is no real need of using the introduced solution, if there is no use for the runtime dynamic synchronization and lazy model handling, and all that is needed are some batch updates. In such cases probably it will be easier to implement the import/export features.

The proposed solution does not provide an absolutely universal implementation that could fit all the repositories. Because of the differences in the *API*s of various repositories and variations in their capabilities, presented approach has to be adjusted slightly differently for each of them. However, most part of the implementation (detailed description of which is given in the chapters 4.5.2 and 4.5.3) can be reused and must not change.

CHAPTER 5

Technical Solutions of the METAclipse

As already stated, *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. Used *Eclipse* technologies are already described in CHAPTER 2.

The transformation language *MOLA* [29, 58], developed by *UL IMCS*, was chosen for the implementation of model transformations. *MOLA* has a rich set of language elements and 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" [13]), also developed by *UL IMCS*. So, the choice of the repository also was 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 is described in CHAPTER 4 and 5.1.

This chapter gives a detailed description of technical solutions forming the basis of a newly developed *Eclipse* plugin *METAclipse* that allows easy use of model transformations and materializes the ideas of the transformation-driven tool building platform. Chapter is based mainly on the paper [66].

5.1 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*.

5.1.1 Repository Interface

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

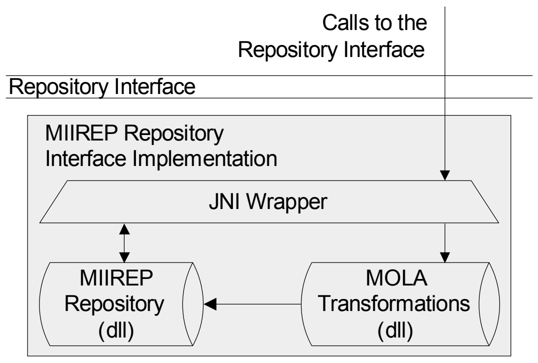


Fig. 21. *MIIREP* repository interface implementation

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. 21) uses a *JNI* (*Java Native Interface*) wrapper for the repository operations (see [47] 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.

5.1.2 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. "Wise" object mechanism is described in CHAPTER 4 in a great detail 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.

5.2 General Description of Presentation Engines

As already stated before, *METAclipse* consists of several presentation engines. Though 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. 36 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 chapters 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 taken over 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.

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 has 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 need some domain model changes and thus is a semantic operation needed to be performed by transformations.

5.2.1 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. Engine metamodels in this case 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 chapter 5.7. The focus in this thesis 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 chapters 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*. It had to be amended then 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 take over 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 chapter 5.3 for the description);
- the *menu* package contains classes for steering the menu engine (see chapter 5.4 for the description);
- the properties package contains classes for steering the properties engine (see chapter 5.5 for the description);
- the *graphDiagram* package contains classes for steering the graph diagram engine, excluding the classes for palette organization (see chapter 5.6 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.

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

The *general* package defines the core classes of the *METAclipse* presentation metamodel (see Fig. 22). In this and following figures a special color-coding will be used. Normal metamodel classes will be shown in yellow (or lightest in black-and-white printouts). Pink (or slightly darker in black-and-white) will represent the command classes described later in this chapter. See also chapter 3.2 and descriptions of *METAclipse* presentation engines for more information on what a command is. Blue (or darkest in black-and-white) classes will denote the singletons. The description of the term "singleton" is given below.

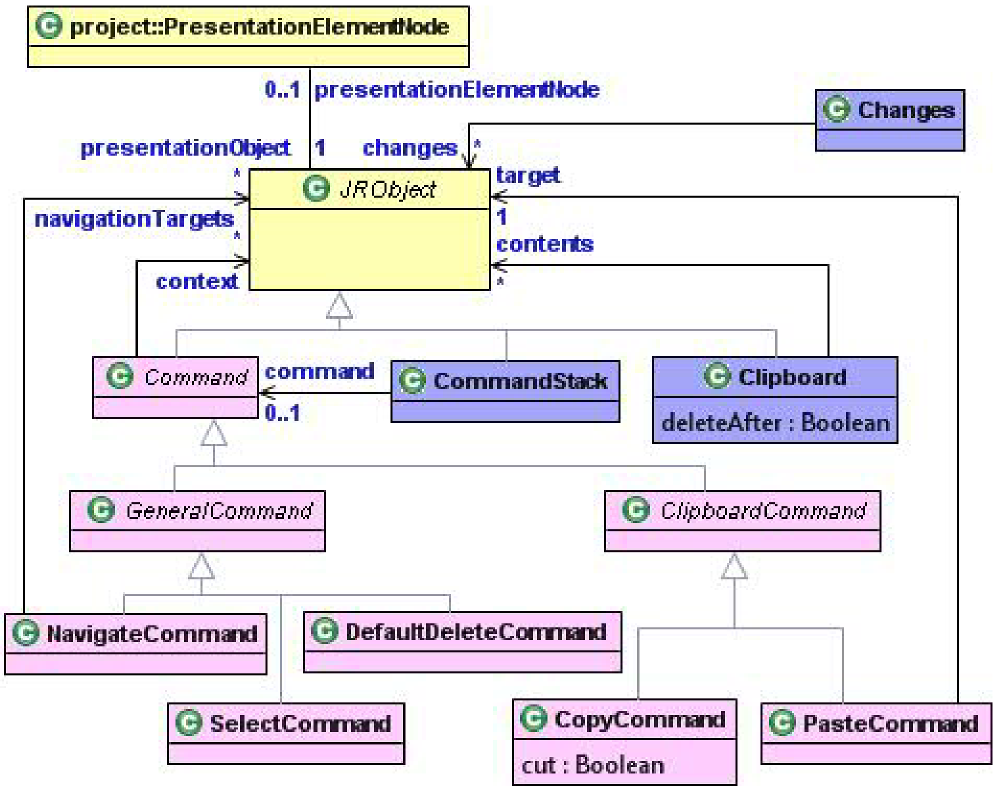


Fig. 22. The general part of the presentation metamodel

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. 22).

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 chapter 4.5.3, for wise objects to work there is a need of 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 contains also the supporting and base classes for one of the backbones of *METAclipse*, namely, the command infrastructure. Commands are already 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. 22). This opens diverse command parsing possibilities in transformations.

The sequence of command execution in *METAclipse* is described in chapter 3.2. 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. 22 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 chapter 5.5 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.

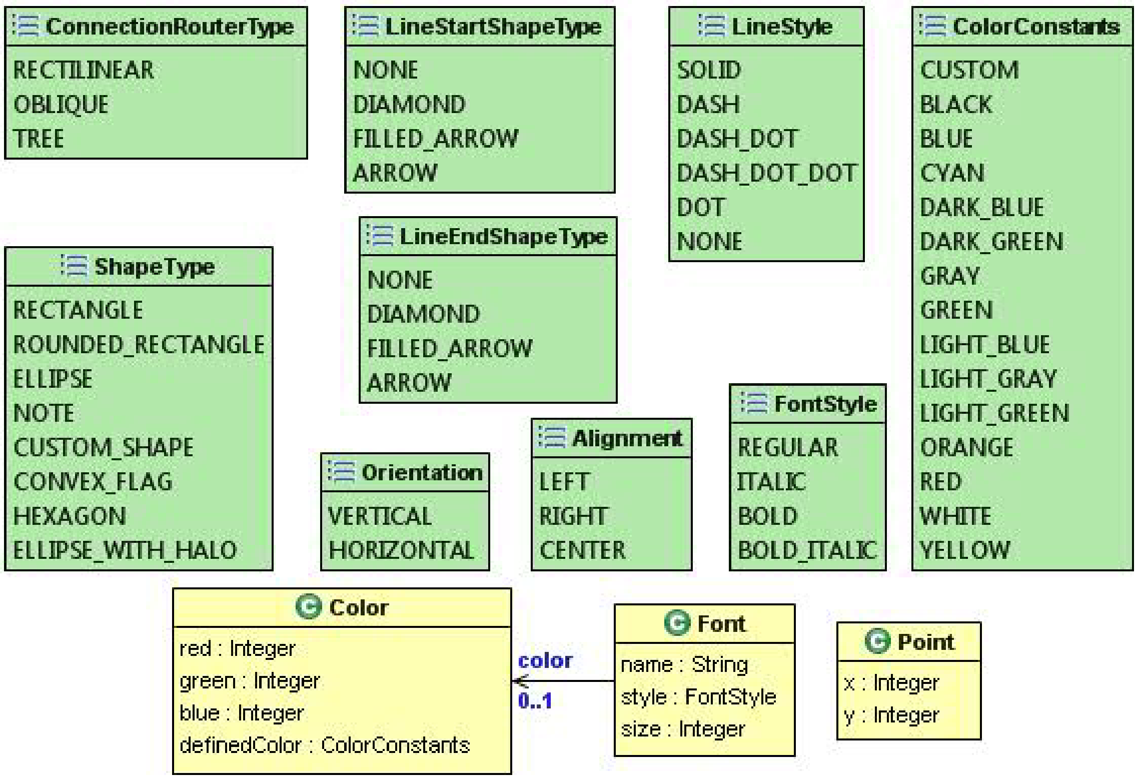


Fig. 23. General type part of the presentation metamodel

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*.

5.2.3 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 chapter 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 chapters to follow. See Fig. 24 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.

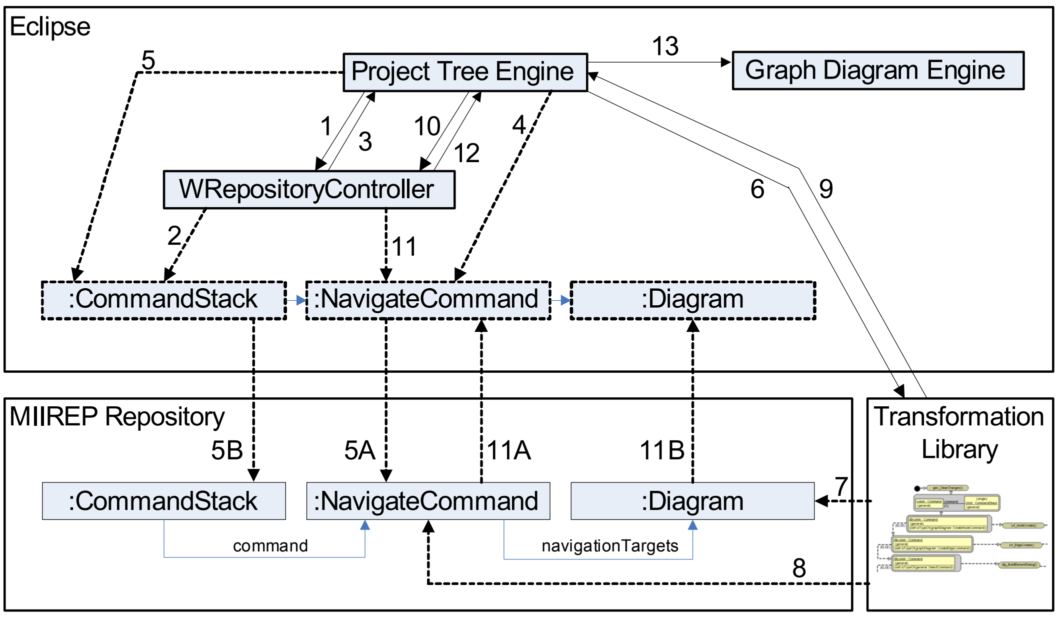


Fig. 24. Opening a new diagram from the project tree: an example of the *METAclipse* component interaction

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 chapter 5.3). 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 chapter 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* is not 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 the result of the execution. Additionally, it puts a link from the *Changes* singleton (see previous chapter) 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 causes also 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 on to describing property generation for the element that is currently selected. However, the operations for that would be very similar to the ones described already. 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.

5.3 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. 36, part 1). The *Eclipse* navigator framework already provides all the functionality required to manage the project tree. Only thing needed to implement a specific project tree is an implementation of *Navigator* interfaces for the retrieving of the model data (or 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.

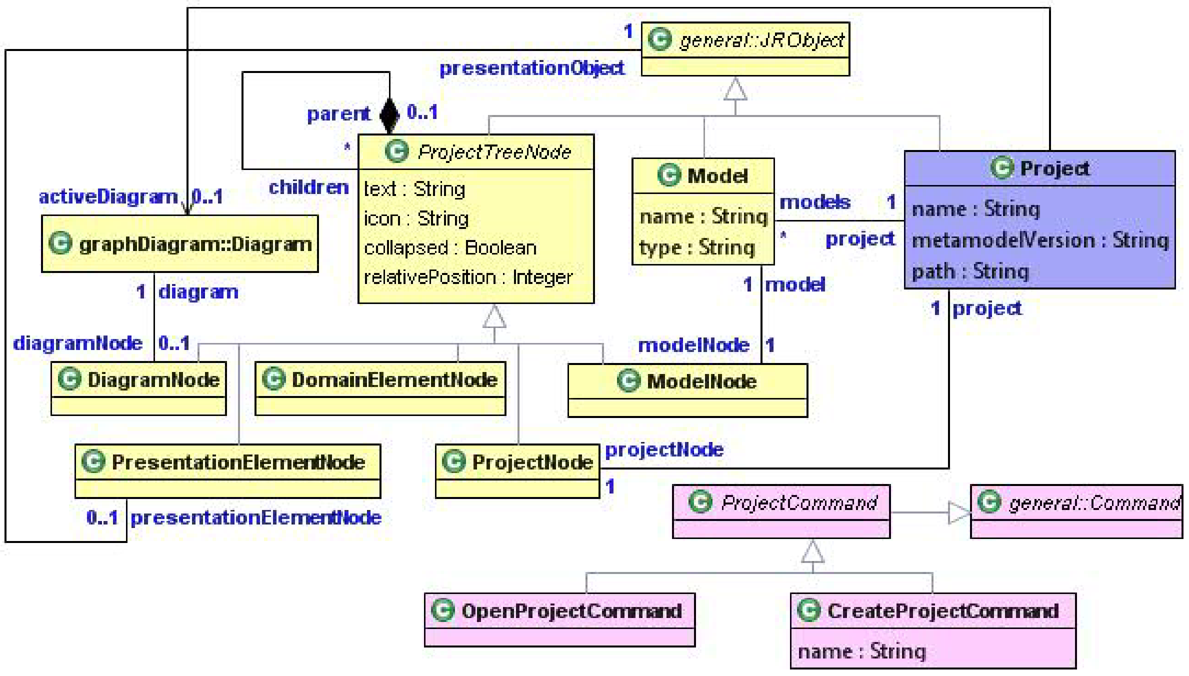


Fig. 25. Project part of the presentation metamodel

Fig. 25 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 [24]), 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.

5.4 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. 36, 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. 26), 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.

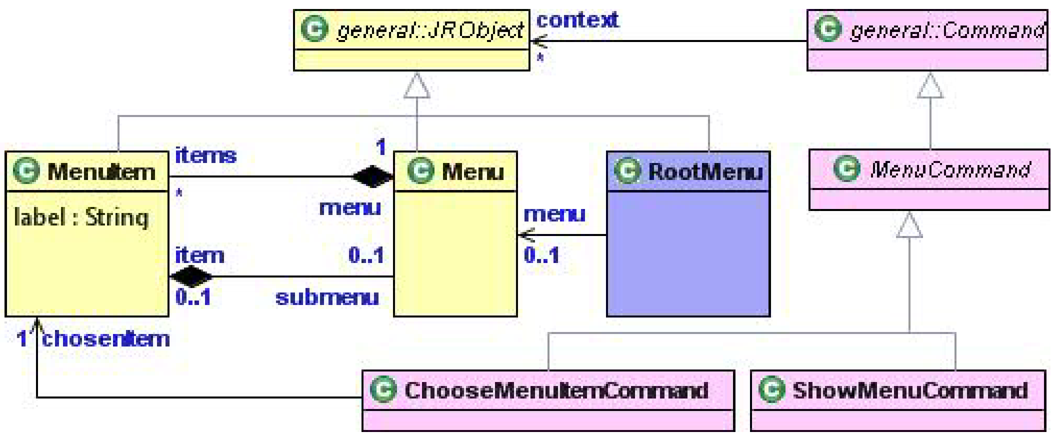


Fig. 26. Menu part of the presentation metamodel

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. Transformations must carry out the corresponding action then. Action can be anything necessary for the menu item chosen starting from creation of some element up to very complicated tasks like model simplification, compiler invocation for visual *DSL* languages and so on.

5.5 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. 36, part 3). Any time the selection in *Eclipse* changes, the contents of the properties view is also updated to reflect the properties of the currently selected item. Properties can be arranged in so-called tabs for better structuring.

The properties view is driven by yet another *Eclipse* framework, the tabbed properties framework [67], 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, at 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.

5.5.1 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 chapter 5.2.1 and seen in Fig. 22). Main part of the property engine metamodel can be seen in Fig. 27.

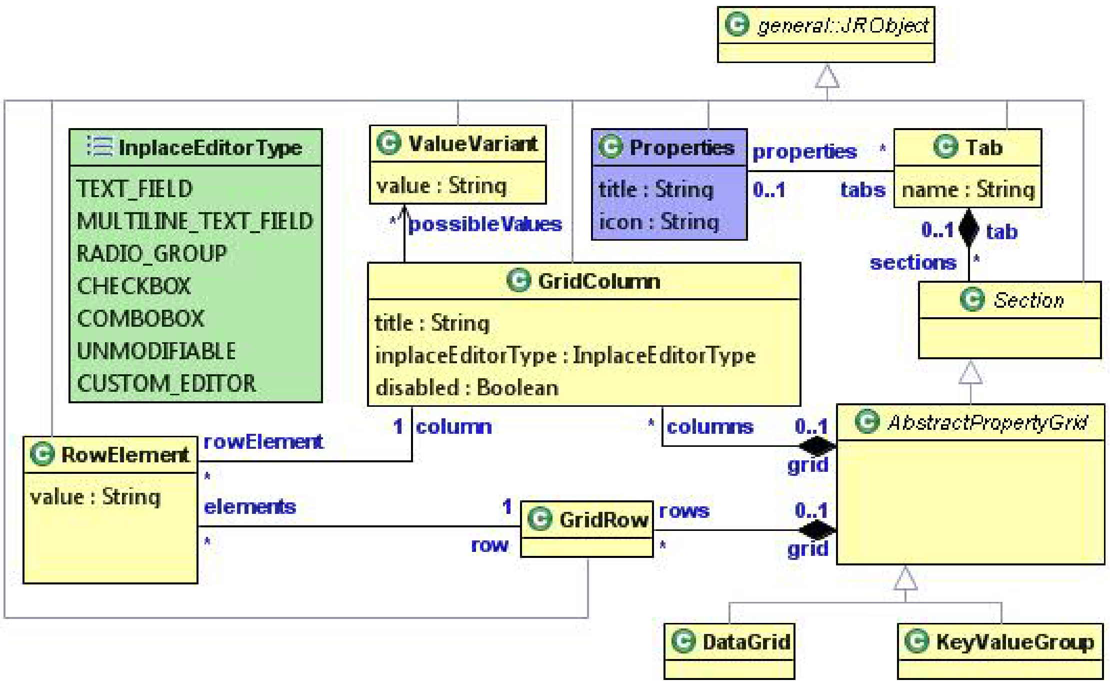
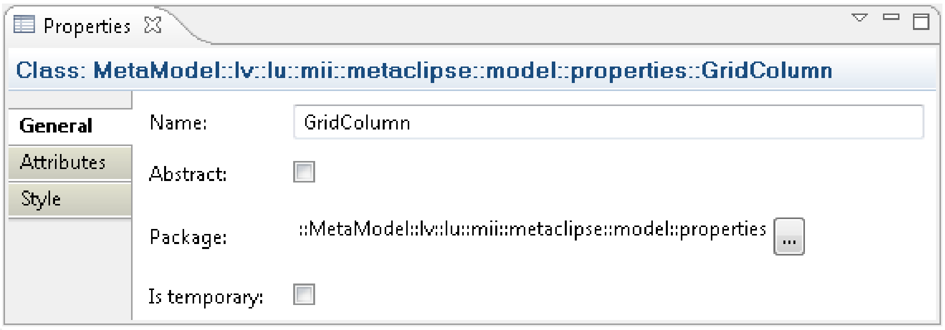


Fig. 27. Property part of the presentation metamodel: main classes

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. 28 consist of three tabs: "General," "Attributes," and "Style."

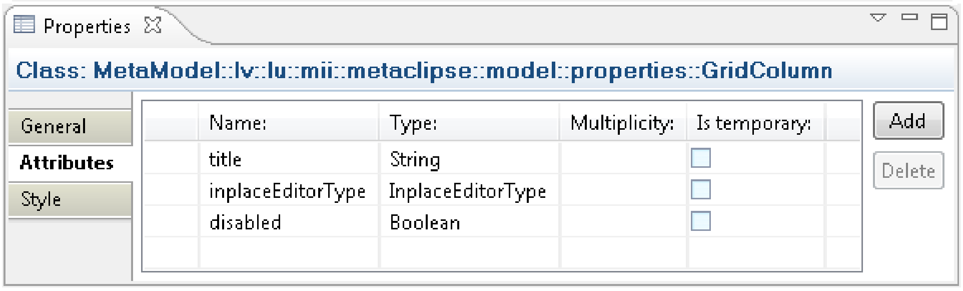
Every tab in the tabbed properties framework consists of so-called sections. Sections group the properties shown in the tab in logical groups. The corresponding class in the metamodel is the abstract *Section* class. The *Tab* class has a composite association with *Section*. As many section implementations 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. 28, 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. 28, 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.


Properties

Class: MetaModel::lv::lu::mii::metaclipse::model::properties::GridColumn

Name:	GridColumn
Abstract:	☐
Package:	::MetaModel::lv::lu::mii::metaclipse::model::properties
Is temporary:	☐


Properties

Class: MetaModel::lv::lu::mii::metaclipse::model::properties::GridColumn

Name:	Type:	Multiplicity:	Is temporary:
title	String		☐
inplaceEditorType	InplaceEditorType		☐
disabled	Boolean		☐

Fig. 28. *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. 27) 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. 28). *RowElement* instances correspond to the value part of key-value pairs, i.e., the values of the properties that can be edited.

5.5.2 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. 29).

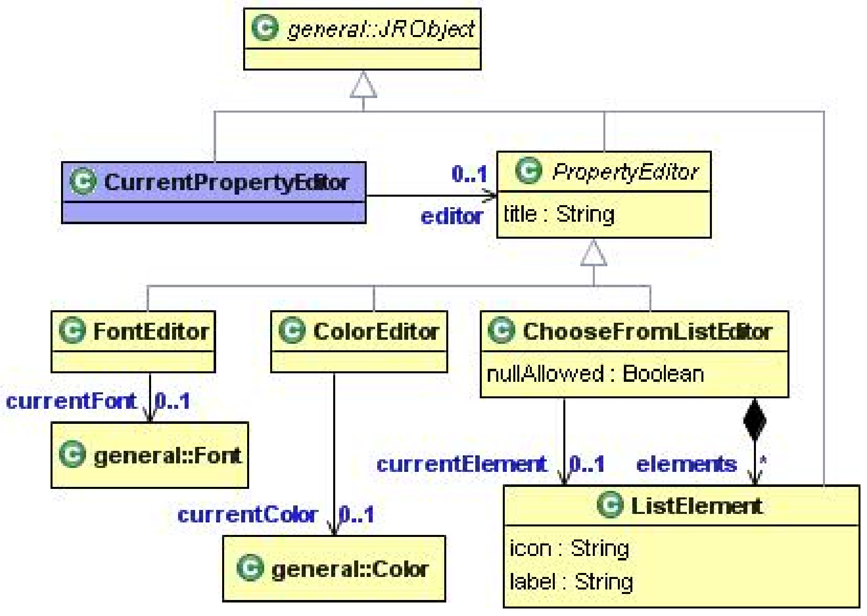


Fig. 29. Property part of the presentation metamodel: editor classes

Theoretically it would be possible also 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. 30) 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.

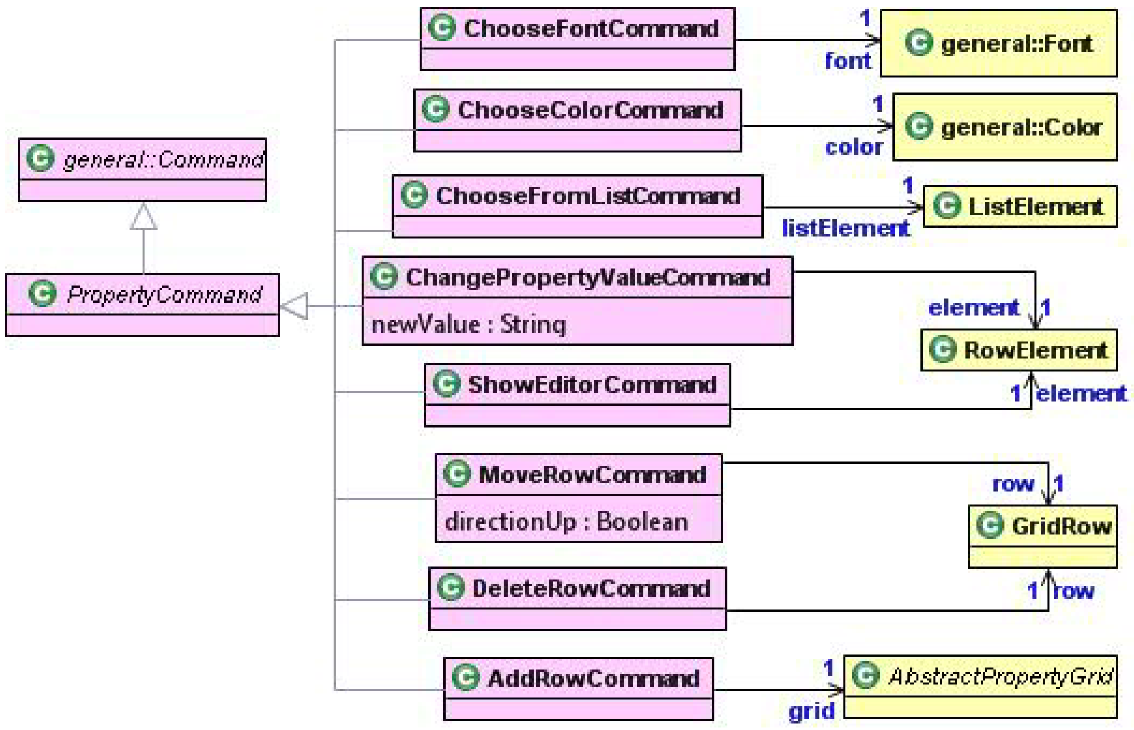


Fig. 30. Property part of the presentation metamodel: command classes

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.

5.6 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. 36, part 4). *Eclipse* technologies used for the graph diagram engine are the *Graphical Editing Framework GEF* [42] and the *Graphical Modeling Framework GMF* [4]. *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 (see CHAPTER 2).

GMF is using 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 is given in the paper "*Building Tools by Model Transformations in Eclipse*" [11] and in chapter 1.4 of this thesis.

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 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* [57] 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.

5.6.1 The General Part of the Graph Diagram Engines Metamodel

The main part of the graph diagram engines metamodel in *METAclipse* slightly resembles 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. 31).

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.

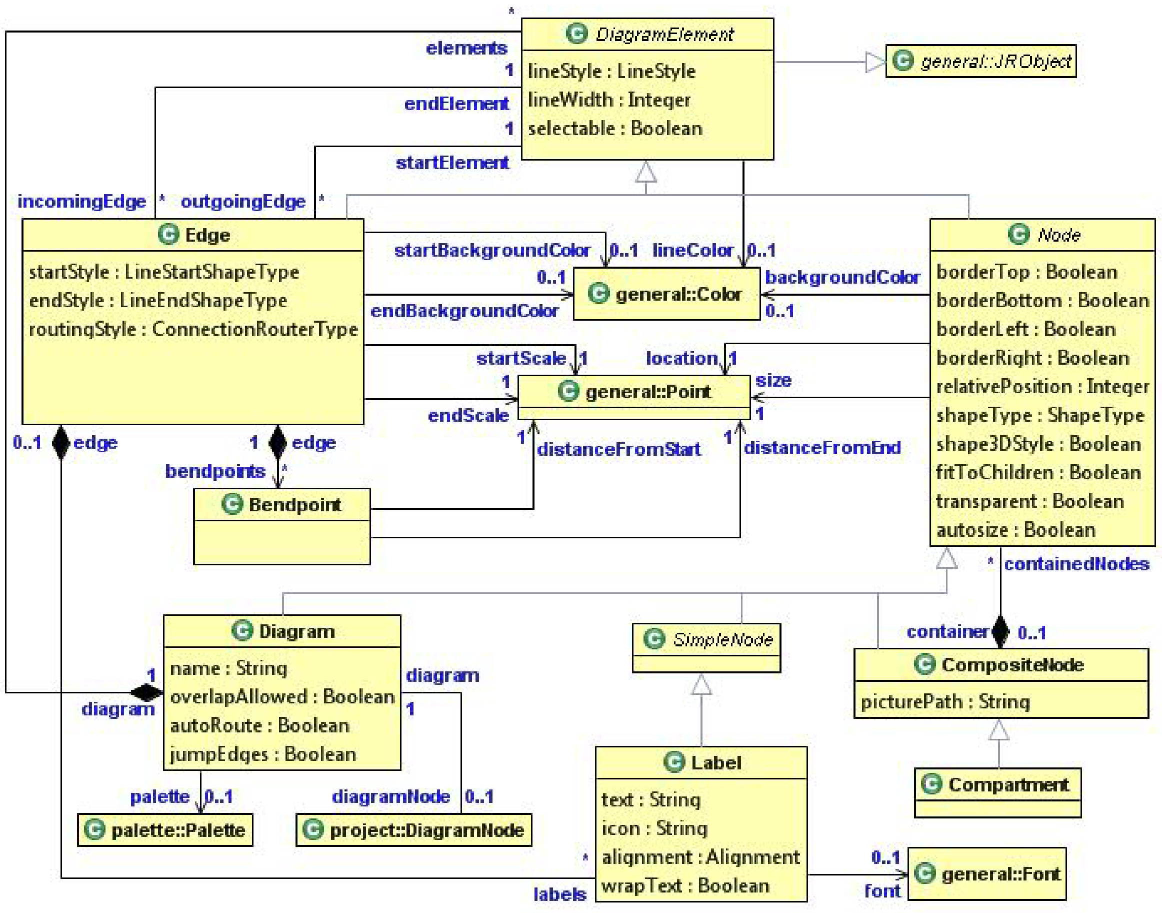


Fig. 31. Graph diagram part of the presentation metamodel without commands and palette

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* [24].

Just as an example, let us consider the *UML* class diagram (like the one in Fig. 31). 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. 32 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.

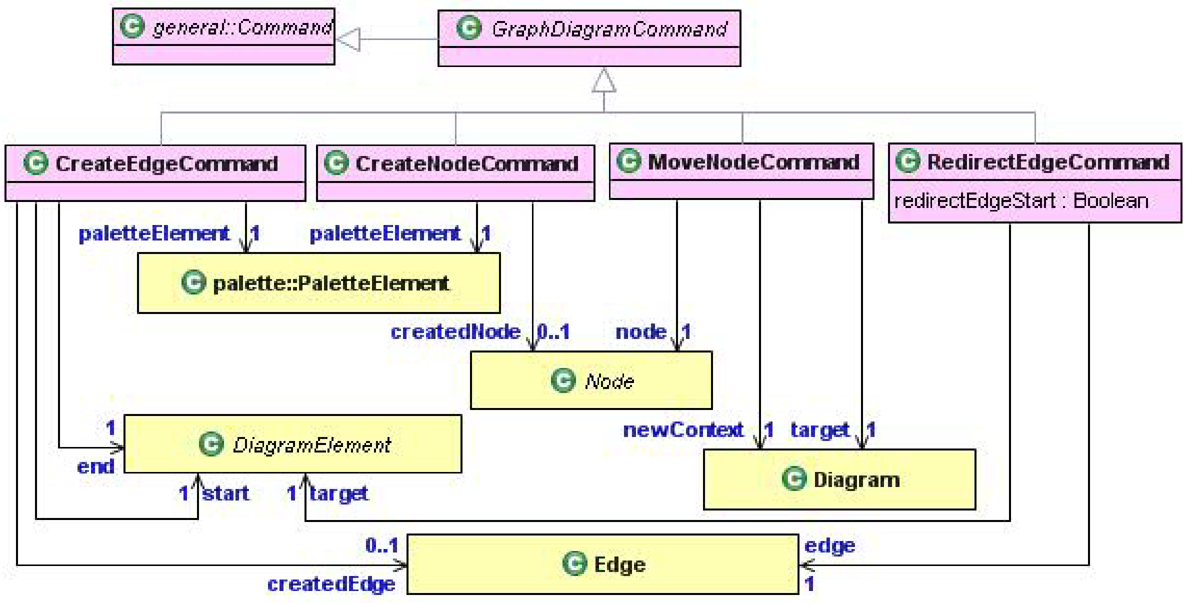


Fig. 32. Graph diagram command part of the presentation metamodel

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.

5.6.2 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. 33 shows the palette part of the graph diagram engines metamodel.

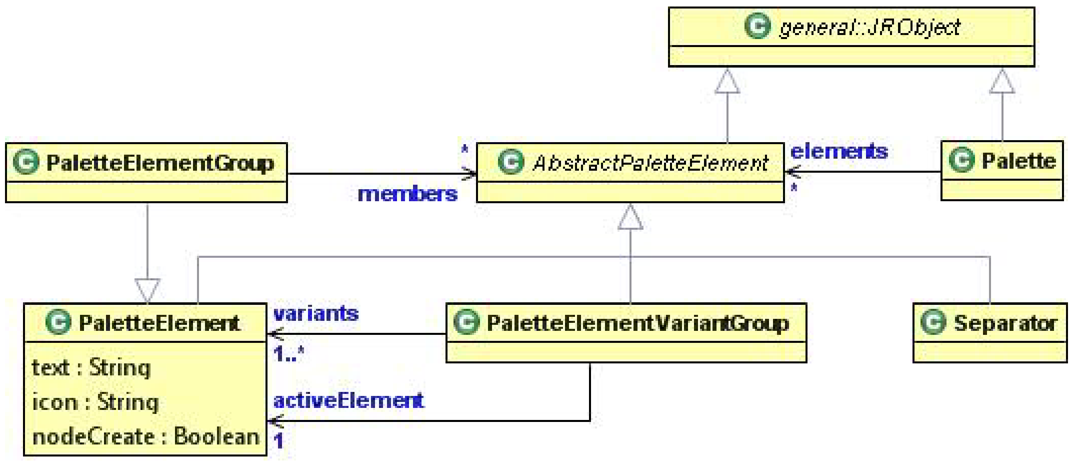


Fig. 33. Palette part of the presentation metamodel

Structure of the palette metamodel is representing 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.

5.7 Transformation Structure

Describing the transformation part of the framework is not the objective of this thesis. Therefore transformations will be discussed here very briefly. As already stated, transformations in *METAclipse* are written in the *MOLA* model transformation language [29, 58]. The *MOLA* compiler uses another model transformation language developed at *UL IMCS*, i.e. *Lx* language series [40]. *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.

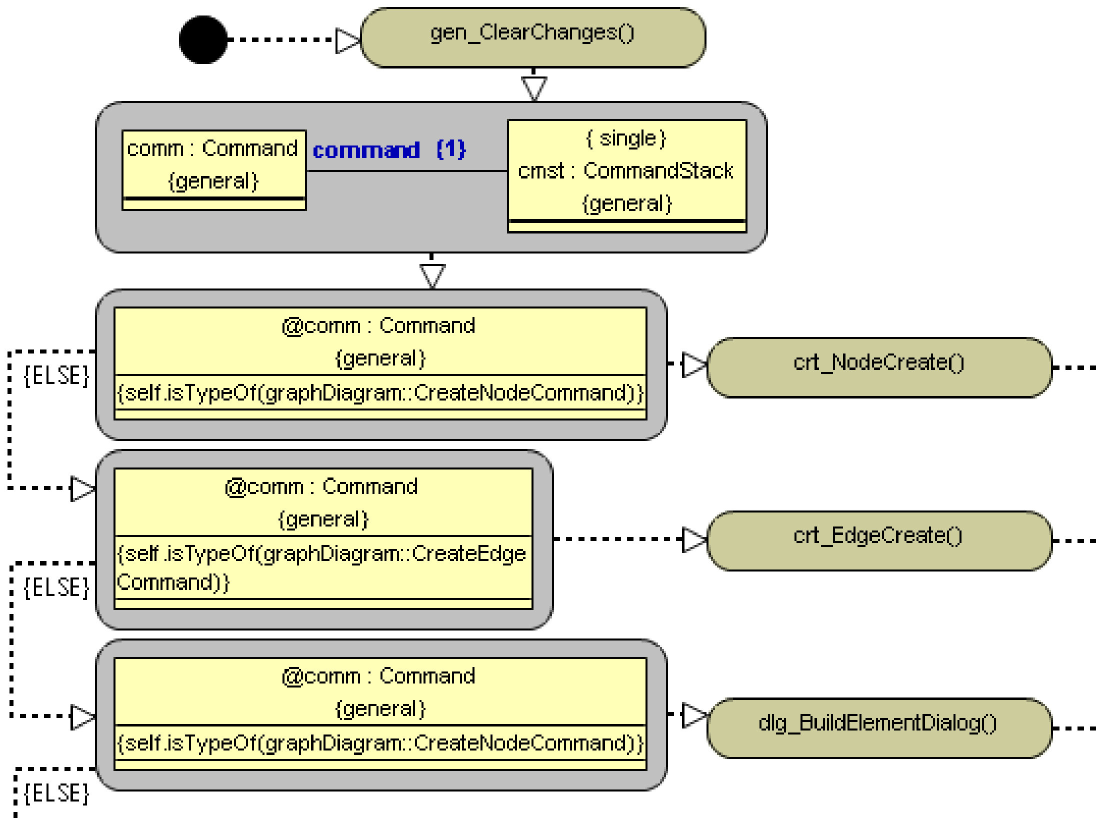

gen_ClearChanges()
comm : Command {general}
command {1}
{ single} cmst : CommandStack {general}
@comm : Command {general} {self.isTypeOf(graphDiagram::CreateNodeCommand)}
crt_NodeCreate()
{ELSE}
@comm : Command {general} {self.isTypeOf(graphDiagram::CreateEdgeCommand)}
crt_EdgeCreate()
{ELSE}
@comm : Command {general} {self.isTypeOf(graphDiagram::CreateNodeCommand)}
dlg_BuildElementDialog()
{ELSE}

Fig. 34. 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. 34, 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 chapter 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.

METAclipse metamodel consists of the static "presentation" part, fixed by the *METAclipse* framework, and a domain part, varying for each tool. In order to simplify the transformation building in *METAclipse* and imitate the "static mapping paradigm" of *GMF* as far as possible, there is another static part in the predefined metamodel, which is used by the transformations only: the scaffolding part. *METAclipse* itself does not force to use this part of the metamodel, however it makes writing the transformations much easier. The scaffolding part is used to define the structure of a specific diagram (e.g., a *UML* class diagram) in a static way, and by use of links to "hook up" all static parts of the metamodel, such as palette and menus, on this "static frame". Fig. 35 shows this "static" part of the metamodel and some of its links to other ("non-static") components. The scaffolding consists of two subparts. One is the diagram structure definition by means of instances of *DiagramType*, *NodeType*, *EdgeType* etc., which constrain what types may be inside/under other types in a valid diagram. This structure definition is a semantic equivalent to the static presentation definition in *GMF*, but in a form transformations can benefit from. The other subpart is style definitions, which specify the graphical styles for diagram elements. Instances of the static part are built only once, at the start of a new project.

Also the transformation set in the tool's transformation library consists of two parts: the domain-independent part, which actually belongs to the framework itself, and the domain-dependent part, which implements functionality specific to the graphical editor for some domain and should be built for each new domain.

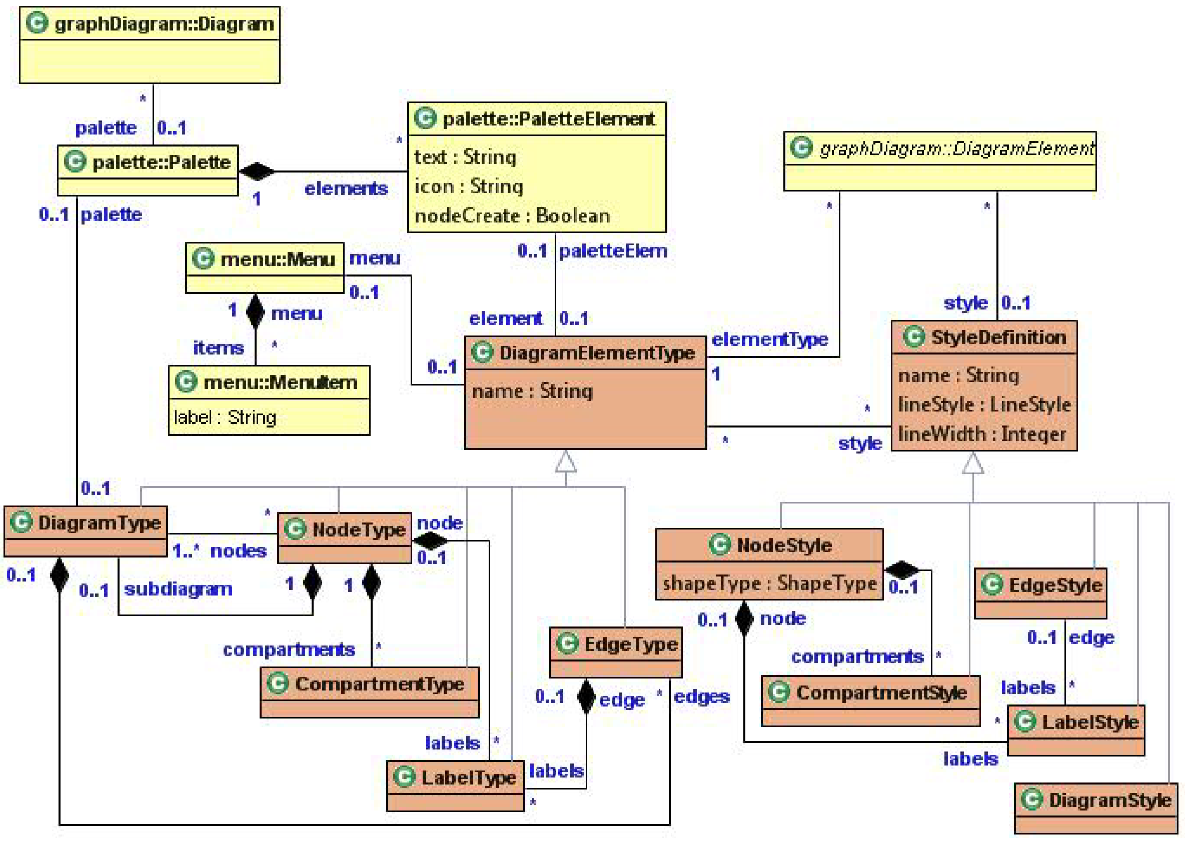


Fig. 35. Static part of the metamodel

As it was explained in the previous chapters, transformations in general have to execute the current command (only one at a time) generated by some presentation engine. The domain-independent part of transformations includes an interpreter-like structure, which recognizes the current command and checks its overall validity. But the main volume of domain independent procedures is based on the static metamodel part. The static part of the metamodel (Fig. 35) is created by special initialization transformations, which are invoked once per project, immediately after its physical creation.

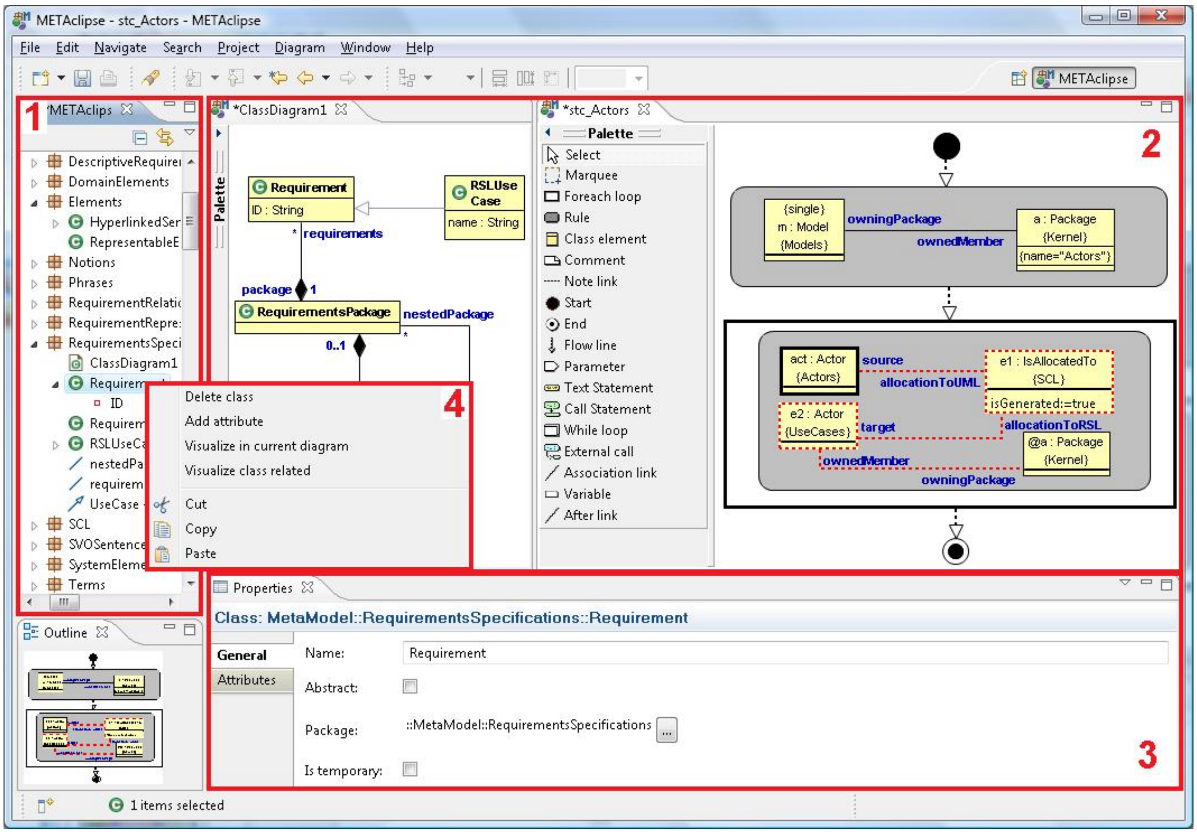
When instances of the static part are in place, many transformation tasks can be implemented in a domain-independent way, as generic interpreters based on static instances. The first such example is style-setting transformations, which set default styles for new diagram elements or perform data-dependent style modifications when domain instance data have been modified (such as the appropriate style for an association end). Certainly, all possible style instances must be built during the initialization. A similar schema can be used, for example, to build a universal procedure for displaying menu items for the selected diagram element.

The domain-dependent transformation procedures do domain-dependent semantic jobs (build domain and presentation instances, build details of a property dialog, update object properties and so on). These actions cannot be performed by universal domain-independent procedures, since a lot of specific semantic checks are to be performed during such editing. Nevertheless, these specific procedures can use a lot of domain-independent subroutines for standard jobs, such as a universal "context validity" checker for a new element (on the basis of the static structure definition), a presentation element remover (used after a domain element was deleted, which may involve nontrivial semantic checks), and others.

CHAPTER 6

Practical Application of METAclipse: The MOLA Tool

The MOLA transformation language editor created with *METAclipse* was the main testbed for the developed *METAclipse* platform and for the transformation application methodology in modeling tool building as such. As *MOLA* is a typical example of a complicated *DSL*, the transformation-based approach described in the thesis is definitely superior to the mapping approach in this case. Therefore *MOLA* is a very appropriate example for testing of *METAclipse* possibilities.



Class: MetaModel::RequirementsSpecifications::Requirement

Name: Requirement

Abstract: ☐

Package: ::MetaModel::RequirementsSpecifications

Is temporary: ☐

Fig. 36. *METAclipse* presentation engines in action: the *MOLA* tool

The *MOLA* tool has been created through the so-called bootstrapping method, for initial implementation using the prototype of the *MOLA* editor developed with *Generic Modeling Tool* [3]. The new editor implements a series of model validations and an intelligent population of menus with context-dependent values. The *MOLA* editor consists of a *UML* class editor for processing of metamodels and a procedure editor for processing of transformation programs. The procedure editor depends on much more complicated domain-specific logic for creation and modification of procedure elements.

In the window of the graph diagram shown in Fig. 36 (denoted with number 2), both editors can be seen: the class editor to the left and the procedure editor to the right. Both editors are interrelated: for example a modification of a class name in the metamodel has to be reflected in all class elements in *MOLA* procedures that use the given class. Furthermore, if the class is used in multiple diagrams, changes have to be propagated to all of them.

After the development of the first *METAclipse* version (which included approximately 180 domain-independent *MOLA* procedures), the development of the new *MOLA* editor required approximately one man-month for implementation and tests (the editor consists of approximately 120 *MOLA* procedures in the domain-specific part and about 30 main domain metamodel classes). The *MOLA* tool along with the editor provides also the *MOLA* compiler.

Currently the source procedures of the *MOLA* editor have been completely transferred from *Generic Modeling Tool* environment to the *MOLA* editor implemented with the *METAclipse* platform (the *MOLA* version has also been upgraded to the most recent one). The editor has been extended with various functions that make the creation of transformations even easier. The current functionality of the editor is spread across 435 *MOLA* procedures and its metamodel consists of 50 classes.

Transformation programs of the editor itself are a good performance test for the *MOLA* editor, as the data volume that has to be processed by the editor is equivalent or even exceeds typical use cases. The programs of the *MOLA* editor contain about 100 000 metamodel class instances (around 70 000 in the presentation and 30 000 in the domain part). Even with such a high data volume the performance of the editor is good: the time that is spent by the editor for model processing is short and it does not slow down user actions. Of course, execution of some of the functions is time consuming (for example, procedure compilation or other semantically complicated operations); however, it is so because of their complexity and the time spent working on them is adequate.

The developed *MOLA* editor has been approbated in real-world environment: it is being successfully used as the main "*MDA* instrument" in the European Union *IST* 6th framework project *ReDSeeDS* [14]. The typical data volume in this project is around 80 *MOLA* procedures and 4000 domain object instances (approximately 10 000 together with the instances of the presentation classes).

CHAPTER 7

Conclusions and Future Work

The main goal of the research was to develop a universal metamodel-based tool building platform that would be driven by model transformations. In order to achieve this goal, the following problems were solved:

- A new methodology and architecture have been developed for a universal metamodel-based modeling tool that is being driven by model transformations. In essence, it is a platform that is completely based on metamodels and that allows defining the correspondence between domain and presentation models using exclusively model transformations. Comparing to other similar platforms, the main benefit of this platform is that there are virtually no limitations on the way mappings can be defined. Also, in the case of complicated *DSL* logic, no additional knowledge of any *OOP* language is required in order to create an editor, nor is it required to have deep architectural understanding of the platform itself. Therefore, in case of complicated *DSLs*, the new solution is more efficient than existing ones.
- The developed methodology and architecture have been practically applied by implementing a metamodel-based transformation-driven tool building platform, *METAclipse*, which has been based on the *Eclipse* platform. The *METAclipse* platform is voluminous original software containing more than 50 000 code lines in the *Java* programming language. The implemented platform has been successfully applied in subsequent research and also used for the development of a new editor for the *MOLA* transformation language (a typical example of a complicated *DSL*). *EU 6th Framework* project *ReDSeeDS* (*Requirements-Driven Software Development System*), which is aimed to develop methodology and supporting tools for a real (in the sense of *MDA*)

model-based system development, bases the whole development on the newly created *MOLA* editor. *METAclipse* demonstrates application of the methodology in developing a platform for building *DSL* editors. The methodology itself, however, can be used more generally, for creating a variety of transformation-driven metamodel-based tools.

- Another important result of the thesis is the development of an easy-to-use presentation metamodel (although the *Eclipse* platform offers many frameworks for tool building, they do not have a good and convenient metamodel available). The metamodel formalizes several *Eclipse* presentation mechanisms: engines for drawing graph diagrams, project tree management, object property editing, as well as context menu assembly. Such formalization is important by itself, as it allows looking at the *Eclipse* platform through a modeling perspective and driving the elements of the *Eclipse* platform through formal models.
- An interpreter of the metamodel has also been created (it is an extension of the *Eclipse* engines, which fully utilizes their features), allowing to visualize the instances of the metamodel in corresponding *Eclipse* engines. One can say that a set of enhanced and extended presentation engines has been developed, which not only supplements the existing *Eclipse* engines with new functionality, but also allows driving them through a special interface, namely, a metamodel. The formalization allows using these engines also outside the *METAclipse* platform in various model-based solutions, as it allows driving them through instances of the corresponding metamodels. Therefore, the solution itself can be considered of value to the developers in the *Eclipse* environment.
- A solution for external repository integration in the *Eclipse EMF* environment has been developed, which is essentially a universal solution with its own scientific value and can be used as a standard solution for repository integration. The newly created tool uses the *MOLA* transformation language for the definition of mappings and

operates on a specialized model repository, *MIIREP*. The repository integration solution was created in order to enable the use of the *MOLA* language in the *METAclipse* platform.

The practical result of the thesis—the *METAclipse* platform—provides the implementation of a completely transformation-based approach to the building of *DSL* editors. The thesis shows that the chosen approach is effective in cases when the correspondence between domain and presentation metamodels is relatively complex. This statement is confirmed by the *MOLA* tool that was developed using *METAclipse*. However, in more simple cases, when domain and presentation metamodels are close to each other, it would be easier to use the static mapping approach (see chapter 1.4). Even in the case of complex *DSL*s, often some parts of the domain metamodel are very similar to the corresponding presentation metamodel parts. Therefore, combining the static mapping and transformation approaches would allow efficient creation of DSL editors in simple cases, and also would even further improve the development efficiency of complex *DSL*s. By combining the transformation-driven approach with the possibility to use static mappings, the developed methodology has a potential to reach or even exceed the efficiency of the existing solutions also in the development of editors for simple *DSLs*. The stated combination of the approaches is a very wide source for research. Currently other authors are working on doctoral theses focusing on the development of an efficient unified solution. The research is based on the results of this thesis, both theoretical and practical.

At the moment *METAclipse* already contains all the necessary functionality for successful development of *DSL* editors. For example, the *MOLA* tool that was created using *METAclipse* has proven itself as a powerful tool for editing *MOLA* model transformations and is being successfully used in practice. In order to further test the capabilities of the *METAclipse* platform, currently also a master's thesis is being developed at *UL IMCS*, which focuses on the development of an editor for the graphical form of the *MOF QVT* using

METAclipse. The *METAclipse* platform can be used also for implementation of complex workflows.

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 thesis. 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. Current implementation of the *METAclipse* allows only request-response type of communication between the engines and transformations (engines issuing commands and transformations changing the state of the models). For debugging it would be necessary also to allow transformations to send commands to the engines.
- 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 tasks listed above illustrate only several aspects that could improve the *METAclipse* platform. The amount of the work to be spent for the implementation of the features mentioned above is relatively small in comparison to the work that has already been invested to provide the basic functionality of *METAclipse*. Of course, the number of different functions that could be implemented in order to improve the tool and that would be useful for both *DSL* developers and the users of *DSL* editors is virtually infinite.

It must be noted also that currently the author of the thesis is working on the development of a new version of *METAclipse*, which is going to use a *MOLA* transformation language runtime version that works already in a clean *EMF* environment and uses *EMF* objects directly. In this version there is no need to use the external repository anymore, which allows creating a more homogenous *METAclipse* architecture, with purer *Eclipse*- and *Java*-based implementation. This will allow adding the new version of *METAclipse* as an *Eclipse* project in "*incubator*" state.

REFERENCES

1. Stahl, T., Völter, M., Czarnecki, K.: Model-Driven Software Development: Technology, Engineering, Management. John Wiley & Sons, Ltd., 2006.
2. Kelly, S., Tolvanen, J-P.: Domain-Specific Modeling: Enabling Full Code Generation. John Wiley & Sons, Ltd., 2008.
3. 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.
4. Graphical Modeling Framework (GMF, Eclipse Modeling subproject), http://www.eclipse.org/gmf/
5. Cook, S., Jones, G., Kent, S. and Wills, A. C.: Domain-Specific Development with Visual Studio DSL Tools. Addison-Wesley, 2007.
6. Karsai, G.: A Configurable Visual Programming Environment: A Tool for Domain-Specific Programming, IEEE Computer Society Press, pp. 36–44, 1995.
7. 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.
8. Taentzer, G., Crema, A., Schmutzler, R., Ermel, C.: Generating Domain-Specific Model Editors with Complex Editing Commands. Proceedings of AGTIVE 2007, Universität Kassel, Germany, October 2007.
9. Rath, I., Varro, D.: Challenges for advanced domain-specific modeling frameworks. Proceedings of Workshop on Domain-Specific Program Development (DSPD), ECOOP 2006, France.
10. Amelunxen, C., Königs, A., Rötschke, T., Schürr, A.: 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.
11. 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.
12. Eclipse Platform, http://www.eclipse.org/
13. 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.
14. ReDSeeDS. Requirements Driven Software Development System. European FP6 IST project. http://www.redseeds.eu/, 2007.
15. OMG, Model Driven Architecture, http://www.omg.org/mda/
16. Smolander, K., Martiin, P., Lyytinen, K., Tahvanainen, V-P.: Metaedit—a flexible graphical environment for methodology modeling. Springer-Verlag, 1991.

17. 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.

18. DOME Users Guide, http://www.htc.honeywell.com/dome/support.htm

19. Karsai G.: A Configurable Visual Programming Environment: A Tool for Domain-Specific Programming, IEEE Computer Society Press, pp. 36–44, 1995.

20. 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.

21. 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.

22. Kelly, S., Lyytinen, K., Rossi, M.: 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.

23. Meta-Object Facility (MOF), http://www.omg.org/mof/

24. OMG, Unified Modeling Language: Superstructure, version 2.0, http://www.omg.org/docs/formal/05-07-04.pdf

25. Zhu1, N., Grundy, J. and Hosking, J.: 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.

26. The Generic Eclipse Modeling System (GEMS), http://www.eclipse.org/gmt/gems/

27. OMG, Business Process Modeling Notation, http://www.bpmn.org/

28. OMG, MOF QVT Final Adopted Specification, http://www.omg.org/docs/ptc/05-11-01.pdf

29. Kalnins, A., Barzdins, J., Celms, E.: Model Transformation Language MOLA. Proceedings of MDAFA 2004, Vol. 3599, Springer LNCS, 2005, pp. 62–76.

30. OMG, Object Constraint Language, version 2.0, http://www.omg.org/docs/formal/06-05-01.pdf

31. MetaCase, The S60 Phone Example, Version 4.5, http://www.metacase.com/support/45/manuals/S60%20Phone%20Example.pdf

32. Fujaba. Universität Paderborn, Institut für Informatik. http://wwwcs.uni-paderborn.de/cs/fujaba/documents/user/manuals/FujabaDoc.pdf

33. Resource Description Framework (RDF), http://www.w3.org/RDF/

34. Sesame, http://www.openrdf.org, 2007.

35. OMG Ontology Definition Metamodel (ODM), Final Adopted Specification, http://www.omg.org/docs/ptc/06-10-11.pdf

36. Bézivin, J., Hillairet, G., Jouault, F., Kurtev, I., Piers, W.: Bridging the MS/DSL Tools and the Eclipse Modeling Framework. Proceedings of the International Workshop on Software Factories at OOPSLA 2005, San Diego, California, USA, 2005.

37. Greenfield, J., Short, K.: Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. John Wiley and Sons, 2004.

38. Özgür, T.: Comparison of Microsoft DSL Tools and Eclipse Modeling Frameworks for Domain-Specific Modeling in the Context of the Model-Driven Development. Thesis submitted to the School of Engineering at Blekinge Institute of Technology in partial fulfilment of the requirements for the degree of M.Sc. in Software Engineering, 2007.

39. 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.

40. Lx Transformation Language Set, http://Lx.mii.lu.lv/, 2007.

41. Eclipse Modeling Framework (EMF, Eclipse Modeling subproject), http://www.eclipse.org/emf/

42. Graphical Editor Framework (GEF, Eclipse Tools subproject), http://www.eclipse.org/gef/

43. Visual Automated Model Transformations (VIATRA2), GMT subproject, Budapest University of Technology and Economics, http://dev.eclipse.org/viewcvs/indextech.cgi/gmthome/subprojects/VIATRA2/index.html

44. OMG, MOF 2.0 / XMI Mapping Specification, v2.1.1 http://www.omg.org/technology/documents/formal/xmi.htm

45. Vilitis, O., Kalnins, A.: A Proxy Approach to External Model Repository Integration in Eclipse EMF Infrastructure. Proceedings of the ECMDA Workshop on Model Driven Tool and Process Integration, Berlin, Germany, Fraunhofer IRB Verlag, 2008, pp. 67–78.

46. Bergin, J.: Building Graphical User Interfaces with the MVC Pattern, http://csis.pace.edu/~bergin/mvc/mvcgui.html

47. Java Native Interface Specification, http://java.sun.com/j2se/1.5.0/docs/guide/jni/spec/jniTOC.html

48. EMF Query Project, http://www.eclipse.org/modeling/emf/?project=query

49. EMF Transaction Project, http://www.eclipse.org/modeling/emf/?project=transaction

50. Model Development Tools (MDT) Project, http://www.eclipse.org/modeling/mdt/5.1.1

51. EMF Validation Project, http://www.eclipse.org/modeling/emf/?project=validation

52. Atlas Transformation Language (ATL) Project, http://www.eclipse.org/m2m/atl/

53. JET Project, http://www.eclipse.org/modeling/m2t/?project=jet

54. CDO Project, http://www.eclipse.org/modeling/emft/?project=cdo

55. Teneo Project, http://www.eclipse.org/modeling/emft/?project=teneo

56. openArchitectureWare, http://www.openarchitectureware.org/

57. Rational Software Architect (RSA), http://www-306.ibm.com/software/awdtools/architect/swarchitect/

58. UL IMCS, MOLA pages, http://mola.mii.lu.lv/

59. Metadata Repository (MDR), http://mdr.netbeans.org/

60. Tool Integration within Software Engineering Environments: An Annotated Bibliography, http://www.macs.hw.ac.uk:8080/techreps/build_table.jsp?id=0041

61. Kern, H., Kühne, S.: Model Interchange between ARIS and Eclipse EMF. Proceedings of DSM’07 workshop of OOPSLA 2007, Montreal, Canada, Jyväskylä University Printing House, 2007.
62. Bézivin, J., Brunette, C., Chevrel, R., Jouault, F., Kurtev, I.: Bridging the Generic Modeling Environment (GME) and the Eclipse Modeling Framework (EMF). Proceedings of the Best Practices for Model Driven Software Development at OOPSLA'05, San Diego, California, USA, 2005.
63. Biermann, E., Ehrig, K., Koehler, C., Kuhns, G., Taentzer, G., Weiss, E.: Graphical Definition of In-Place Transformations in the Eclipse Modeling Framework. Proceedings of MoDELS’06, Genova, Italy, October 2006.
64. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley, Boston, MA, 1995.
65. Taentzer, G: AGG: A Graph Transformation Environment for Modeling and Validation of Software. AGTIVE’03, Vol. 3062, Springer LNCS, 2004.
66. Vilitis, O., Kalnins, A.: Technical Solutions for the Transformation-Driven Graphical Tool Building Platform METAclipse. Computer Science and Information Technologies, Acta Universitatis Latviensis, 2008, pp. 179–212.
67. The Eclipse Tabbed Properties View, http://www.eclipse.org/articles/Article-Tabbed-Properties/tabbed_properties_view.html