

# Advanced Graphical Model Decoration with EMF Profiles

DIPLOMARBEIT

zur Erlangung des akademischen Grades

**Diplom-Ingenieur**

im Rahmen des Studiums

**Software Engineering & Internet Computing**

eingereicht von

**Bećir Bašić**

Matrikelnummer 0227261

an der

Fakultät für Informatik der Technischen Universität Wien

Betreuung: O.Univ.Prof. Dipl.-Ing. Mag. Dr.techn. Gerti Kappel

Mitwirkung: Univ.Ass. Dipl.-Ing. Dr.techn. Philip Langer

Wien, 19.02.2015

\_\_\_\_\_  
(Unterschrift Verfasser)

\_\_\_\_\_  
(Unterschrift Betreuung)



# Advanced Graphical Model Decoration with EMF Profiles

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

**Diplom-Ingenieur**

in

**Software Engineering & Internet Computing**

by

**Bećir Bašić**

Registration Number 0227261

to the Faculty of Informatics  
at the Vienna University of Technology

Advisor: O.Univ.Prof. Dipl.-Ing. Mag. Dr.techn. Gerti Kappel

Assistance: Univ.Ass. Dipl.-Ing. Dr.techn. Philip Langer

Vienna, 19.02.2015

\_\_\_\_\_  
(Signature of Author)

\_\_\_\_\_  
(Signature of Advisor)



# Erklärung zur Verfassung der Arbeit

Bećir Bašić  
Engelsberggasse 3/1/13, 1030 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

---

(Ort, Datum)

---

(Unterschrift Verfasser)



# Abstract

EMF Profiles is an adaptation of the well-known Unified Modeling Language (UML) profile concept to Domain Specific Modeling Languages (DSML). Profiles have been a key enabler for the success of UML by providing a lightweight language-inherent extension mechanism which is expressive enough to cover an important subset of adaptation scenarios. Thus, we believe a similar concept for DSMLs provides an easier extension mechanism that has been so far neglected by current metamodeling tools.

The Profile mechanism is based on a profile definition comprised of stereotype definitions. Stereotypes are used to annotate model elements in order to refine their meta-classes by defining supplemental information in form of additional meta attributes, also known as tag definitions. Instances of tag definitions are known as tagged values and they are used for the provision of new informations to existing models.

With EMF Profiles, users can apply profiles within graphical modeling editors that are created using the Graphical Modeling Framework (GMF). Applied stereotypes are visualized using icons that are attached to shapes that represent the model elements to which stereotypes are applied. However, in many scenarios, visualization methods going beyond simple icons are helpful for locating and grasping the applied stereotypes and to allow for more domain-specific decorations according to the domain of the applied profile. For instance, highlighting a shape by a specific background color or enriching the shape with adornments and informations from a stereotype application reflects the meaning of the stereotype application more adequately than a simple icon.

This thesis aims at providing decoration methods for applied stereotypes in EMF Profiles going beyond simple icons. Therefore, we investigate the decoration facilities in GMF and Graphiti and provide a decoration description language to allow users to define specific decorations for stereotypes. Once a specific decoration is defined, the goal is that the applications of these stereotypes are visualized using the defined decorations in any GMF-based and Graphiti-based modeling editor. The results and benefits of the extensions developed in this thesis are evaluated in the context of a case study. In particular, we assess how the runtime information of executable models can be visualized appropriately and dynamically updated during the execution with EMF Profiles.





# Kurzfassung

EMF Profiles ist eine Adaptierung von dem wohlbekannten Unified Modeling Language (UML) Profil-Konzept für domänenspezifische Modellierungssprachen (DSML). Profile haben sehr viel zu dem UML Erfolg beigetragen, da sie einen leichtgewichtigen, sprachinhärenten Erweiterungsmechanismus darstellen. Deswegen glauben wir, dass genau so ein Konzept einen nützlichen Erweiterungsmechanismus für DSML bringt, welcher bis jetzt von den Metamodellierungswerkzeugen vernachlässigt wurde.

Der Profil-Mechanismus basiert auf Profildefinitionen, die Stereotypdefinitionen beinhalten. Stereotypen erweitern Metamodellklassen mit zusätzlichen Informationen in Form von Meta-Attributen, auch bekannt als Tag-Definitionen. Instanzen der Tag-Definitionen, sogenannte Tagged Values, erweitern dadurch die Modelle mit den Informationen, die sie beinhalten.

Die Benutzer von EMF Profiles können Profile auf Modelle anwenden, indem sie dafür graphische Modellierungsedatoren verwenden, die mit dem Graphical Modeling Framework (GMF) erstellt wurden. Angewendete Stereotypen sind im Editor anhand von Icons ersichtlich, die auf dem jeweiligen Shape des Modellelements angezeigt werden, auf welchem der Stereotyp angewendet wurde. In vielen Situationen können Visualisierungen, die über einfache Icons hinausgehen, sehr hilfreich sein, um die angewendeten Stereotypen besser auffinden und erfassen zu können, sowie um zusätzliche, an die Domäne des angewendeten Profils angepasste Dekorationen zu ermöglichen. Beispielsweise könnte die Hervorhebung eines Shapes durch eine bestimmte Hintergrundfarbe oder zusätzliche Verzierungen und Informationen aus der Stereotypanwendung auf dem Shape mehr zum Verständnis des angewendeten Stereotyps beitragen als ein einfaches Icon.

Ziel dieser Diplomarbeit ist es, zusätzliche Dekorationsmechanismen, die über einfache Icons hinausgehen, zu ermöglichen. Daher werden wir die Dekorationsmöglichkeiten von GMF und dem Graphiti-Framework untersuchen und anhand der gewonnenen Informationen eine Sprache zur Beschreibung der Dekorationen bereitstellen, welche den Benutzern der EMF Profiles ermöglichen soll, die Beschreibung der Dekorationen für entsprechende Stereotypen zu definieren. Sobald eine bestimmte Dekoration definiert ist, ist das Ziel, dass die Anwendungen dieser Stereotypen mit ihren definierten Dekorationen in jedem GMF-basierten und Graphiti-basierten Modellierungsedator visualisiert werden. Die Ergebnisse und die Vorteile der in dieser Arbeit entwickelten Erweiterungen werden im Rahmen einer Fallstudie ausgewertet. Dabei wird die Laufzeitinformation von ausführbaren Modellen mit EMF Profiles adäquat visualisiert und dynamisch aktualisiert.



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Motivation . . . . .	1
1.2	Problem Statement . . . . .	3
1.3	Aim of the Thesis . . . . .	5
1.4	Methodological Approach . . . . .	6
1.5	Structure of the Thesis . . . . .	7
<b>2</b>	<b>Background and Related Work</b>	<b>9</b>
2.1	Model Driven Engineering . . . . .	9
2.2	Modeling Language Engineering . . . . .	15
2.3	Transformation Engineering . . . . .	21
2.4	Concrete Syntax Development . . . . .	22
2.5	Extending Modeling Languages . . . . .	31
2.6	Related Work . . . . .	35
<b>3</b>	<b>Running Example and Requirements</b>	<b>37</b>
3.1	Running Example . . . . .	37
3.2	Requirements . . . . .	44
<b>4</b>	<b>Decoration Description Language</b>	<b>47</b>
4.1	Decoration Description Language Engineering . . . . .	47
4.2	Grammar Specification . . . . .	48
4.3	Summary . . . . .	58
<b>5</b>	<b>Decorating Graphical Modeling Languages at Runtime</b>	<b>59</b>
5.1	Eclipse IDE Tooling Environment around EMF Profiles . . . . .	59
5.2	Decoration Description and its Manifestation . . . . .	61
<b>6</b>	<b>Architecture</b>	<b>69</b>
6.1	Architecture of the Solution . . . . .	69
6.2	Profile Application Registry . . . . .	71
6.3	GMF Decorator . . . . .	74
6.4	Graphiti Decorator . . . . .	76

<b>7</b>	<b>Evaluation</b>	<b>77</b>
7.1	Case Study . . . . .	77
7.2	Evaluation of the Extension of the Graphical Concrete Syntax based on GMF .	82
7.3	Discussion of the GMF-based GCS Extension . . . . .	88
7.4	Evaluation of the Extension of the Graphical Concrete Syntax based on the Graphiti Framework . . . . .	88
7.5	Discussion of the Graphiti-based GCS Extension . . . . .	94
<b>8</b>	<b>Conclusion and Future Work</b>	<b>97</b>
8.1	Conclusion . . . . .	97
8.2	Future Work . . . . .	98
<b>A</b>	<b>Grammar of the Decoration Description Language</b>	<b>101</b>
A.1	The Grammar Specification in Xtext . . . . .	101
A.2	Railroad Visualization of the Grammar Syntax . . . . .	106
	<b>Bibliography</b>	<b>111</b>

# Introduction

The introduction part is meant to give the reader an overall information about this work. We begin with motivating reasons for this master's thesis and then tackle the problems still encountered in the research field, after which we will present the aim of the thesis, what kind of methodological approach is used, and at the end give an overview of the structure of this work.

## 1.1 Motivation

When we think about software engineering as a discipline to create executable software artifacts, we probably think of it as a task of writing source code in one of the myriads of programming languages out there, e.g., C, Java, C#, Ruby, Haskell, JavaScript, etc. This approach can be classified as *code-centric*, where a software engineer writes the source code in a tool of his/her choice, be it a simple text editor or a full-blown integrated development environment (IDE).

The complexity of creating software artifacts can directly relate to the complexity of tasks that one tries to overcome through the usage of the information technology (IT). However, that is certainly not the only complexity software engineers have to deal with. Today, we can certainly say that working as a software engineer is not a one man job, as someone might imagine. It is usually a team work with team sizes spanning from just a few colleagues to hundreds of persons involved in the software engineering process. So, managing the process and the people involved in it can also be a very demanding task. Software engineering is a process comprised of *analyzing* the problems we need a solution for, which leads to the definition of *requirements* we *must-have* a working solution for — and those that could make the solution more appealing to work with, also known as *nice-to-have* requirements. Having the requirements, we *design* an architecture for the solution-to-be, e.g., Model-View-Controller (MVC) [10], and also make the more granular designs of the system in form of different *modeling diagrams*. For example, to design the structure of components in the system we might employ a *component diagram*; to design an interaction between the elements in the system we might model a *sequence diagram*. After the design phase, software engineers start with an *implementation* in a programming language and frameworks of their choice. Following the implementation phase is the *verification* &

*testing* phase — not necessarily after the implementation phase, because this can also be done in parallel — for verification and validation purposes of the product. At the end of the process, as the last phase, is the product *deployment & maintenance* phase, which lasts as long as the product is in use. Building upon the solution further, such as adding new functionalities to it — inspired by new ideas — would lead to repeating the process.

We could imagine the software engineering process as a continuous cycle depicted in Figure 1.1. So, doing a bad job in the beginning of the process will most certainly end in more

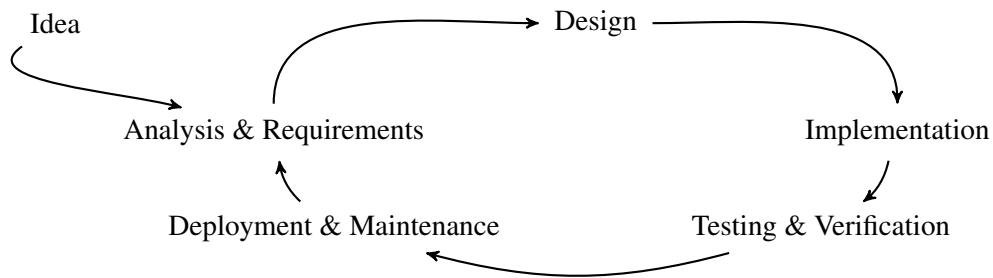


Figure 1.1: Software Engineering Process.

complexity in later steps. Bad design decisions or no design at all can make the work of a software engineer very difficult. Without a way to convey the clear and comprehensible information of what is to do, the whole undertaking of building a software product may be doomed.

In 1997 Object Management Group (OMG)<sup>1</sup> standardized Unified Modeling Language (UML) [23] a general-purpose modeling language in the field of software engineering for visualizing and documenting the design of an information system [46]. With usage of many different modeling diagrams — as, e.g., previously mentioned component and sequence diagrams, and also *class diagram*, *use case diagram*, etc. — we can model structure, behavior and interaction of the system.

The notion of Model Driven Engineering (MDE) [6,31] emerged in November 2000 with the OMG publicly announcing MDA (Model Driven Architecture) [49] initiative. The idea behind MDE was that the models are to play the central role in developing executable software, and not, as before, to provide only visualization and documentation purposes. Models, as *first-class citizens*, provide information and specification for the software which is generated by the usage of *transformation techniques*. For example, information in models is transformed into a specific programming language source code which in turn is compiled for a specific platform on which the software will be executed. MDE promotes the *model-based* practices apposed to the code-centric. For more information about the MDE go to Chapter 2.

Because UML was designed to be general purpose notation, in certain situations this brought much complexity when modeling particular *domains*, for which specialized languages could

<sup>1</sup>Object Management Group. <http://www.omg.org>

have been more appropriate. UML gained even more traction when OMG introduced UML profiles [20] which addressed exactly the afore mentioned problem. UML profiles are a lightweight language extension mechanism allowing to adapt the *metamodel* (i.e., the language definition of UML) for different technological platforms and domains without requiring modifications of the UML modeling tools.

The focus of this master thesis is exactly on the extension of modeling languages without the need for the modification of modeling tools.

## 1.2 Problem Statement

As previously mentioned, Model Driven Engineering (MDE) is a discipline of software engineering which promotes the use of software models for the development of software artifacts. The practices of MDE have proved to increase efficiency and effectiveness in software development, as stated by various quantitative and qualitative studies [2, 9].

As models build the basis in MDE for generating executable software, they have to conform to well-defined *modeling languages* in terms of an *abstract syntax* and a *concrete syntax*. To define the abstract syntax of a modeling language, we build *metamodels* as an abstraction of models that conform to the defined modeling language. This process is also known as *meta-modeling* [9]. One prominent metamodeling language is *Ecore*, from the Eclipse Modeling Framework (EMF)<sup>2</sup>, which is a core technology in Eclipse<sup>3</sup> for model driven engineering. EMF not only provides facilities to define metamodels, but also tools, such as editors and views to visualize or manipulate concrete instances of metamodels, inside the Eclipse environment. Even more, there are frameworks which support the creation of the concrete syntax — graphical or textual — for the concrete concepts of new languages, based on EMF and GEF<sup>4</sup>. Graphical concrete syntax, and the creation of editors that can support it, can be specified and built using frameworks such as the Graphical Modeling Framework (GMF)<sup>5</sup> or Graphiti<sup>6</sup>.

### Extension of modeling languages

As a rule of life, most things are susceptible to change — due to new conditions introduced with the passage of time — so are modeling languages not the exception to the rule. In some situations, modeling languages have to be extended beyond their original specification to accommodate the changes, e.g., new requirements of stakeholders or new concern-specific information must be introduced to the language, thus, modification and extension of the language, to meet the new criteria, is immanent. Consider you have a generic data modeling language that is designed to be independent from any specific database management system (DBMS) and users need to

---

<sup>2</sup>The EMF project is a modeling framework and code generation facility for building tools and other applications based on a structured data model. <http://www.eclipse.org/modeling/emf>

<sup>3</sup>Eclipse is a platform that has been designed for building integrated development tooling. It supports rapid development of integrated features based on a plug-in model. <http://www.eclipse.org>

<sup>4</sup>Graphical Editing Framework (GEF) provides technology to create rich graphical editors and views for the Eclipse Workbench UI. <http://www.eclipse.org/gef>

<sup>5</sup>Graphical Modeling Framework. <http://www.eclipse.org/modeling/gmp>

<sup>6</sup>Graphiti project. <http://www.eclipse.org/graphiti>

add DBMS-specific features, such as dedicated storage engines, to enable the full generation of all SQL scripts.

To enable the extension of modeling languages, there are two possibilities, as mentioned in [39]:

**The heavyweight extension** of modeling languages denotes the modification and extension of the modeling language's metamodel. Adapting modeling languages in this way to the changing needs of the domain they represent is a time-consuming and tedious task, because not only their abstract and concrete syntax but also all related artifacts as well as all language-specific components of the modeling environment have to be re-created or adapted each and every time. This can, indeed, be considered as a *heavyweight* approach.

**The lightweight extension** introduces new language concepts into modeling languages not by modifying and extending their metamodels, but through other dedicated extension mechanisms that do not affect the modeling language's metamodel. The lightweight extension is an approach that tries to ease the pain of extending modeling languages by circumventing many hardships common to the heavyweight approach. One such approach is based on *EMF Profiles* [39].

EMF Profiles are an adaptation of the UML profile concept to the realm of Domain-Specific Modeling Languages (DSMLs). As already mentioned, UML profiles played a major role in the success and widespread uses of UML by providing a lightweight, language-inherent extension mechanism [48], which also lead to the standardization of several UML profiles by OMG. The Profile mechanism is based on a *profile definition* comprised of *stereotypes*, which are used to annotate model elements. Stereotypes are the specialization of *metaclasses* defined in metamodels. They can also define additional *meta attributes* (tagged values) for the provision of new informations to existing models.

EMF Profiles provide an extension mechanism for existing DSMLs with following benefits [39]:

- Lightweight language extension. EMF Profiles provide the ability to systematically introduce further language elements without having to recreate the whole modeling environment, such as editors, transformations, and model APIs.
- Dynamic model extension supports dynamic extension of already existing models by introducing additional profile information, without the need to recreate extended model elements.
- Preventing metamodel pollution. Information coming from the outside of the modeling domain can be represented by additional profiles without polluting the actual domain metamodels. Also, that information is kept separated from the model, so there is no pollution of actual model instances.
- Model-based representation. Information introduced to the model, through profile application, is accessible and processable such as any other model information. Thus, model engineers can use familiar model engineering technologies to process profile information.



Although EMF Profiles provide a neat way of extending a modeling language, they support model engineers only by providing an extension mechanism for the abstract syntax, and do not offer any means, except for simple icons, to extend the concrete syntax.

### **Problem of the graphical concrete syntax extension**

At the moment the extension of the graphical concrete syntax is not supported thoroughly in EMF Profiles. With EMF Profiles, users can apply profiles within graphical modeling editors that are created in GMF. Applied stereotypes are visualized using icons that are attached to shapes representing the model elements, to which stereotypes are applied. However, in many scenarios, visualization methods going beyond simple icons are helpful for locating and grasping the applied stereotypes and to allow for more domain-specific decorations according to the domain of the applied profile. For instance, a specific background color or a dedicated shape reflects the meaning of a stereotype application more adequately than a simple icon.

The extension of the graphical concrete syntax — the visual shapes representing model elements — can span from simple highlighting of shapes, by means of, say, changing their background color, changing their border type, or attaching icons to them, and similar; to, say, a shape is changed in its form that it looks like a completely different shape. So, how feasible the extension of the graphical concrete syntax through the means of the lightweight language extension is, depends very much on the underlying technologies for the specification of the graphical concrete syntax and their extension mechanisms.

## **1.3 Aim of the Thesis**

This master's thesis has the following goal:

**Adaptation/Extension of the graphical concrete syntax of a modeling language to reflect the information added through a lightweight language extension mechanism.**

More concretely, the aim of this thesis is to provide decoration methods for applied stereotypes in EMF Profiles going beyond simple icons. Therefore, we investigate the visual decoration facilities in GMF and Graphiti and use the gained information to design a *decoration description language* allowing users to define specific decorations for stereotypes. The decoration description language also provides means of restricting the activation of decorations based on the comparison of tagged values contained in stereotypes against concretely specified values. Once a specific decoration is defined, applications of these stereotypes are visualized using the defined decoration descriptions in any GMF-based and Graphiti-based modeling editor.

The solutions elaborated in this thesis are tailored to EMF Profiles but the concept could also be adapted easily to UML profiles.

## 1.4 Methodological Approach

This master's thesis follows the guidelines from the methodological approach based on the *design science* paradigm [27]:

1. **Problem identification and its relevance.** In a nutshell — there are no known means to extend the graphical concrete syntax in the area of the lightweight language extension. Adapting the graphical concrete syntax of a model element to reflect a newly added information introduced through the lightweight extension mechanism and an easy specification of visual adaptations through the means of visual decorations would bring a lot for locating and grasping the newly added information.
2. **Literature Analysis.** We do an extensive literature analysis and the evaluation of the previous research work done in the field of heavyweight vs. lightweight language extensions. Especially, the differences of those approaches are evaluated in the way how they address the extension of the graphical concrete syntax.
3. **Design as a search process.** We look for already existing approaches for a lightweight language extension and their mechanisms for supporting the graphical concrete syntax extension. Upon that knowledge we build a new solution.
4. **Design as an artifact.** Upon the knowledge gained from the search process we build a new solution addressing our requirements. This means we do following:
  - Adopt already existing implementation of EMF Profiles and its API to introduce an adequate architecture and an implementation to accommodate our requirements.
  - Design and implement a decoration description language and an interpreter of the language.
  - Provide implementations for the decorators which address the underlying technology in which the graphical concrete syntax is defined for the particular DSML — the scope of this work is limited to GMF and Graphiti.
5. **Evaluation.** After completing the solution, we undertake the evaluation of the results and benefits of the extensions developed in this thesis in the context of a representative case study for our problem statement. In particular, we are interested to assess how the runtime information of executable models can be visualized appropriately and dynamically updated during the execution with EMF Profiles.

Finally, in the last step we provide a critical reflection and a discussion of our design choices and the functionality of our solution. Especially, this means a discussion about trade-offs and limitations of our approach against a heavyweight approach where the complete rebuild of a modeling language and its artifacts must be done.

## 1.5 Structure of the Thesis

This master's thesis is organized as follows.

Chapter 2 gives a general introduction to the topics of Model Driven Engineering (MDE), modeling language development, graphical syntax development within EMF, and the extension of modeling languages — the extension of abstract and concrete syntax respectively — as well as related work to it.

Chapter 3 provides a common ground for better grasping and understanding of informations presented in this thesis by introducing a *running example* that illustrates the need for the adaptation of the graphical concrete syntax in the case of the extension of a modeling language through the mechanism of EMF Profiles. In this chapter we also specify the requirements for our solution.

In the Chapter 4 we present a decoration description language. It is a domain specific modeling language (DSML) for describing decorations of stereotypes that influence the graphical representation of modeling language elements to which stereotypes are applied.

The Chapter 5 gives an overview on how decoration descriptions, specified in the decoration description language, manifest themselves on model elements in an editor at runtime.

In the Chapter 6 we provide an overview of the architecture of our solution and we describe few relevant components in some detail.

A case study is presented in Chapter 7 that we have used to assess our solution for which we have set research questions based on our requirements. In particular, we assess how the runtime information of executable models can be visualized appropriately and dynamically updated during the execution with EMF Profiles. We also provide critical reflection and discussion on limitations we encountered while developing decorators for our targeted technologies GMF and Graphiti.

The final Chapter 8 gives a conclusion and an outlook on future work.



# Background and Related Work

Modeling language development is a part of Model Driven Engineering (MDE) approach, which we describe in some detail. After that we concentrate on the development of modeling languages, i.e., development of their abstract and concrete syntax. We handle the development of the concrete syntax within Eclipse Modeling Framework (EMF)<sup>1</sup>, especially of graphical concrete syntax, since this work is concerned with the extension of graphical concrete syntax based on EMF technologies. Next topic is the extension of modeling languages — the extension of their abstract and concrete syntax respectively. We present different approaches for the extension of modeling languages and as part of the search for related work done for this master's thesis, we provide the comparison of these approaches, especially on how and to what extent do they address the extension of the graphical concrete syntax.

## 2.1 Model Driven Engineering

Model Driven Engineering (MDE) [6, 31] as a notion emerged in November 2000 when the Object Management Group (OMG)<sup>2</sup> publicly announced Model Driven Architecture (MDA)<sup>3</sup> initiative [49]. MDE denotes another approach in software engineering that, as opposed to code-centric approach, represents a model-centric approach in building software artifacts. This means that models in MDE play the central role and are not only used for design, communication or documentation purposes, as it is the case within the code-centric approach, where the models are used as guidelines or blueprints to implement a solution. We might say that MDE brings the models to life because it bridges the gap between the design phase and the implementation phase of the software engineering process (cf. Figure 2.1), which means that out of the models we can generate an executable software. To achieve this, MDE encompasses different kind of

---

<sup>1</sup>The EMF project is a modeling framework and code generation facility for building tools and other applications based on a structured data model. <http://www.eclipse.org/modeling/emf>

<sup>2</sup>Object Management Group. <http://www.omg.org>

<sup>3</sup>Model Driven Architecture. <http://www.omg.org/mda>

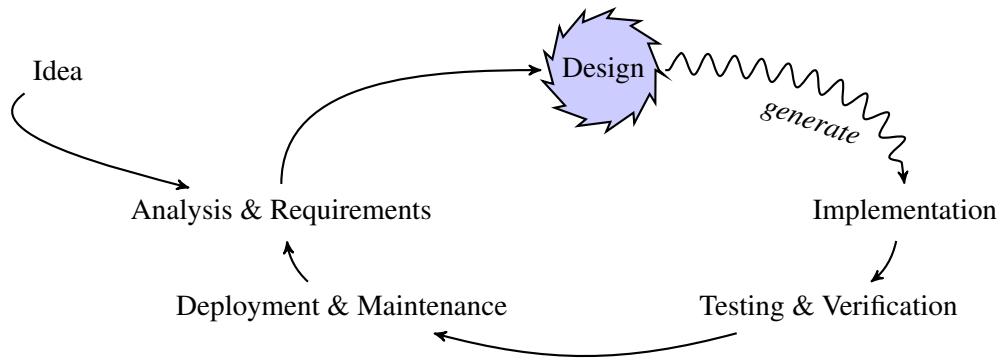


Figure 2.1: The gap in the traditional software engineering process: Generating the implementation from system designs.

techniques, such as language engineering and transformation techniques, and leverages tool sets build around them, such as generators, views and editors, to provide efficient means to use and bridge them all together.

Before we go into any details, let us also say that nowadays the MDE notion is broader in the sense of its activities because not only it is set around software development paradigm, as it is in the case of MDA, but it encompasses other model-centric tasks of a complete software engineering process, such as, the model-based evolution of the system or the model-driven reverse engineering of a legacy system [9, 28].

## Model Driven Architecture

MDA itself is not a separate OMG specification but rather an approach or a guideline to system development which is enabled by existing OMG standards and specifications [9, 42]. A formal definition of MDA was provided in the year 2001 by the OMG and current guideline is from 2003 (cf. MDA guide in [42]). Because it relies on OMG standards, which are well adopted by industry, it is seen as a very good example or the conceptual reference of the MDE approach. Some standards and specifications it combines at its core are, e.g., Unified Modeling Language (UML)<sup>4</sup>, Meta-Object Facility (MOF)<sup>5</sup>, Common Warehouse Metamodel (CWM)<sup>6</sup> and XML Metadata Interchange (XMI)<sup>7</sup> among others.

The core idea of the MDA approach is the strict separation of the operational specification of a software system from the details of the way how it is implemented for a specific platform on which it will be executed. This is done through different abstraction levels that can be specified

<sup>4</sup>Unified Modeling Language. <http://www.omg.org/UML>

<sup>5</sup>Meta-Object Facility. <http://www.omg.org/mof>

<sup>6</sup>Common Warehouse Metamodel. <http://www.omg.org/spec/CWM>

<sup>7</sup>XML Metadata Interchange. <http://www.omg.org/spec/XMI>

through modeling. By separating the specifications into a general and specific ones, the general specification can be used repeatedly to create specific specifications of the system for different platforms and execution environments. An example would be a general specification of a system in the form of models specifying business data and business logic and from that general specification of the system we can create more specific models of the system for particular platforms (e.g., Java EE, .NET) on which the system will be running.

Let us first introduce few basic definitions and assumptions that MDA defines at its core [9, 42]:

- **System** denotes the subject of any MDA specification. It represent some existing or planned system that may include anything, e.g., a program, combination of parts of different systems, people, an enterprise, etc.
- **Model** is a description of the system and its environment.
- **Model-Driven** in MDA means that models are the driving force for the system development.
- **Problem Space (or Domain)** is the context or environment where the system operates.
- **Solution Space** is the spectrum of possible solutions that satisfy the system requirements.
- **Architecture** represents the architecture of the system in the form of the specification of the parts and connectors of the system and the rules for the interactions of the parts using the connectors.
- **Platform** provides a set of subsystems and technologies that expose coherent set of functionalities through interfaces and specified usage patterns that can be used by any supported applications without the concern of knowing how those functionalities are implemented by the platform.
- **Viewpoint** represents a technique for abstraction using a selected set of architectural concepts and structuring rules to only focus on particular concerns within a system.
- **View** is a model of a system seen from the perspective of a chosen viewpoint.
- **Model Transformation** is the process of converting one model to another model of the same system.
- **Application** is a concept used to refer to a functionality being developed. A system is described in terms of one or more applications supported by one or more platforms.

MDE definitions are perfectly in line with these definitions [9]. Based on these definitions, OMG defines a modeling approach which encompasses methods, language engineering, transformation engineering and different modeling levels. As a relevant part for this master's thesis we will focus more on language engineering in the Section 2.2.

## Modeling Levels

The benefit of reusing existing software models on different platforms is an important aspect in MDA. That is why MDA defines different abstraction levels that only contain information from different viewpoints. A *computation independent viewpoint* focuses on the environment of the system and the requirements for the system, thus, leaving the details of the structure and the processing of the system undefined [42]. A *platform independent viewpoint* focuses on the operation of the system while hiding the details necessary for a particular platform, hence it represents the specification of the system which does not change regardless of any target platforms on which the system may run [42]. A *platform specific viewpoint* combines the platform independent viewpoint with an additional detailed specification of the use of a specific platform by the system [42]. For these three different viewpoints on the system MDA defines three different views or models to specify the details of the system at these abstraction levels [9, 42]:

### 1. Computation Independent Model (CIM)

This model represents the highest abstraction specification of a system. It holds the details that describe the purpose and the requirements of the system, but at the same time it hides any implementation related concerns, such as underlying data structures. The assumption is that the CIM is primarily specified by the experts on the domain of the system, which do not necessarily need to be software engineering experts. This abstraction level can be seen as a bridge between two different expert groups, namely those knowledgeable about the domain and its requirements, and those who are experts in realizing software artifacts satisfying the domain requirements. The CIM is also referred to as a business model or domain model.

### 2. Platform Independent Model (PIM)

At the next abstraction level is the model that holds all detailed information on the behavior and structure of a developed application. In other words, the requirements specified in the CIM are taken, for which a software-based solution will be built, and out of these a model is specified covering structure and operations of the components comprising the solution, but at the same time, as the name already suggests, it omits implementation details needed to run the application on a specific platform. At this level we have models that describe our system completely as a software solution independent of any specific platform. Specification of a platform independent model requires for a specific modeling language. More information about modeling languages can be found in Section 2.2.

### 3. Platform Specific Model (PSM)

A platform specific model combines the specification of a system from a PIM instance and additional information covering necessary details on how to run the system on a specific platform. This limits any PSM to one specific target platform, which means that we can have many instances of PSMs for one PIM. At this level, software engineers have all the necessary information to implement the software application out of the PSM specification.

The structure of modeling levels described in MDA is depicted in Figure 2.2.



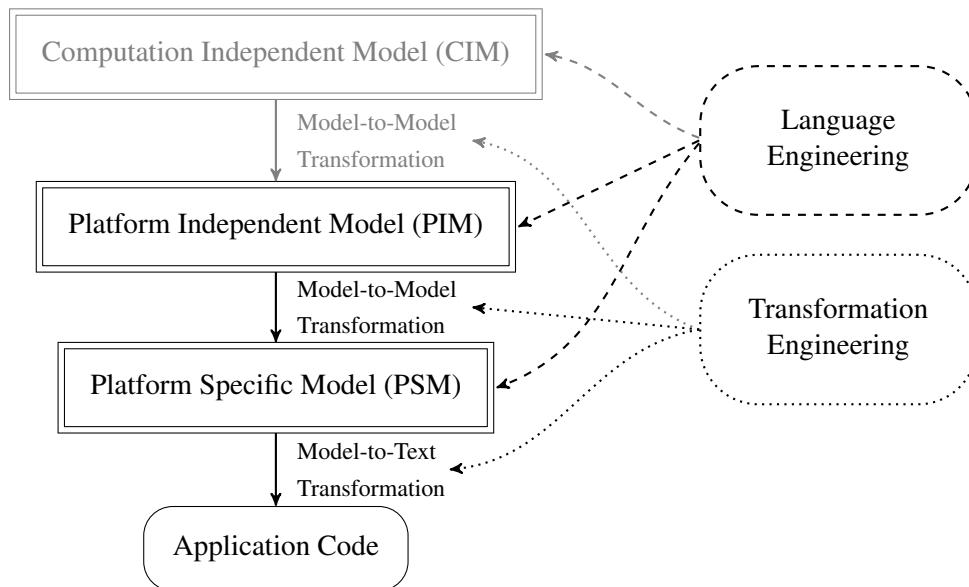


Figure 2.2: MDA modeling levels and related engineering disciplines (based on [9, 44]).

Between each abstraction level there exist a set of mappings so that each subsequent one can be defined through model transformations [9]. For example, every CIM can map to different PIMs, which in turn can map to different PSMs, and PSMs can map to application code.

In practice, and as stated in [44], it is not always possible to clearly distinguish between a CIM and PIM. Furthermore, current transformation tools mostly provide standardized solutions for transformations from a PIM to a PSM but no mechanisms for transforming a CIM into a PIM [30, 44]. There are also attempts and proposals on how to automate the transformation of the specification from a CIM into a PIM, cf. [13, 32, 53], but these approaches mostly work only in certain scenarios and are not widely accepted yet. This means that because of the difficulty to automate the CIM to PIM transformation, and this resulting in specifying them both manually, would lead to more additional effort without clear benefits. To be more efficient, this mostly leads to omitting the modeling of the system specification in a CIM and doing it in a PIM right away, treating it as the highest abstraction level.

In order to specify aforementioned models the OMG recommends its technologies. For example, the CIM and PIM can be specified with the usage of the UML. Transforming the specification from one model into another and mapping the related elements between each other can be done with the usage of a standard set of languages for model transformation defined by the OMG under the name Query/View/Transformation (QVT)<sup>8</sup>. However, this is only a suggestion and not a requirement, which means that there are also other relevant modeling languages and

<sup>8</sup>OMG Query/View/Transformation. <http://www.omg.org/spec/QVT>

technologies — not related to the OMG organization — that we can use to specify and develop our system following the MDE approach.

## Aims of the Model Driven Engineering

Now that we have introduced MDE and some basic concepts of MDA, we will list few aspects of software engineering process that MDE aims to improve [7, 35, 44]:

- **Software development at high level of abstraction**

Using models as a driving force in Model Driven Engineering in general and in Model Driven Architecture in particular guarantees for the higher level of abstraction. For example, the PIM, PSM and application code represent artifacts from different steps in the software engineering process. But, more importantly, they represent the same system specification at different abstraction levels. The ability to transform a high level PIM into a PSM raises the level of abstraction at which a software engineer can work, thus, enabling him to cope with more complex systems with less effort.

- **Productivity**

If we consider traditional way of software engineering process where we use our designs — from the design phase — only as guidelines or blueprints to manually implement the solution, and comparing that to the generation of the implementation out of the designs, as emphasized in the MDE approach, we can clearly recognize the productivity boost.

- **Portability and Reusability**

Having an abstraction level where the specification of a system is independent of any underlying technology or any platform, as it is the case with the PIM, gives us the means to use the same specification and transform it into specifications tailored for specific technologies and platforms. For example, having already the solution for our system realized with the Java programming language and the Java EE platform as its execution environment, we could port our solution to the C# programming language and .NET platform by reusing the PIM of our solution and writing new dedicated transformations.

- **Interoperability**

Nowadays, it is common to have software systems as the conglomerate of smaller systems built on top of diverse technologies, old and new. Many systems are dependent on each other and to function properly the communication and interoperability is of essence. MDE approaches the interoperability issue in the way that if we have PIMs of the systems and their related PSMs then we have all the information needed to generate so-called *interoperability bridges* [35] between the systems.

- **Documentation and Maintenance**

In a traditional software engineering process, creating and maintaining the documentation is a tedious task — from the software engineer's point of view, of course. But, as hard as it is to accept, providing up-to-date documentation is essential part of every software engineer's job. Luckily, there are tools to generate documentation out of the code. This,

consequently, provides only the low level documentation. The high level documentation (text, diagrams) of the system still must be maintained by hand.

In the MDE approach, however, changes to the specification of a system is done not on the code level but on the higher level of abstraction, e.g., in the PIM. Naturally, this means that the documentation at the high level of abstraction is always available.

## 2.2 Modeling Language Engineering

Providing a concrete solution, in the field of computer science, for a problem at hand would traditionally mean implementing or developing the solution in some kind of a programming language, or simply a language that allows us to give instructions to a machine. There are many languages to choose from because over the time the languages in computer science have been designed to address different approaches on how to provide instructions for a machine, but also some language designs are more concentrated around the spectrum of a problem domain, providing language constructs that can be used generally or only in a specific context. We can say that the languages in computer science are tools to specify the solution for a problem, and in most cases they have a textual representation.

Same way, as an integral part of the MDE approach we have *modeling languages* as tools to define a concrete representation of a conceptual model for a solution, which may have graphical representations or textual representations, or both [9].

A language, be it for example, a programming language or a modeling language can be classified into [9]:

**Domain Specific Language (DSL)** is specifically designed to address a specific problem domain, context, or company with the goal to ease the task of people that need to describe things in that domain. For example, HTML is a DSL to define the structure of a web page, CSS is a DSL to define the styling of elements of a web page, or in a modeling realm Object Constraint Language (OCL)<sup>9</sup> is a DSL to specify constraints on objects of a modeling language.

**General Purpose Language (GPL)** is designed to provide concepts which can be applied to any particular domain, which makes it generally applicable — as the name suggests. For example, in computer science we have GPLs, such as Java, C#, Ruby, Python, Haskell — just to name the few — and the most popular modeling language, which falls under the GPL, is the Unified Modeling Language (UML).

All languages — generally speaking this involves also languages that we use to communicate with each other — have certain characteristic that are common to them all.

### Language Characteristics

Every language has some kind of a structure and also some kind of a representation in order to convey a meaning. That is also the case with modeling languages. The structure of a modeling

---

<sup>9</sup>Object Constraint Language. <http://www.omg.org/spec/OCL/>

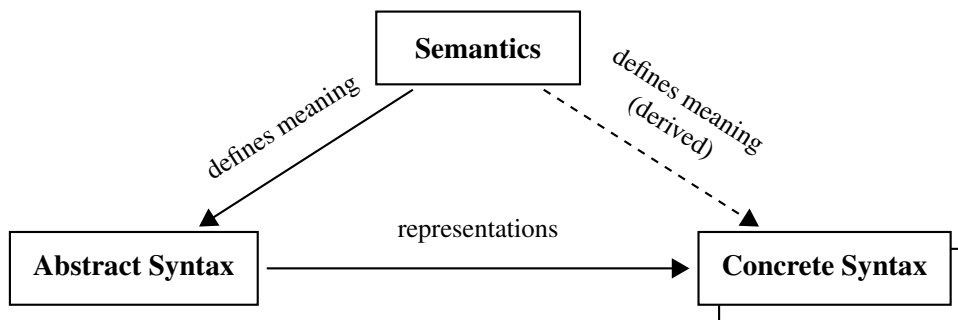


Figure 2.3: The three core ingredients of a modeling language and their relationships [9].

language, its representation and the meaning of its concepts can be characterized as the three core ingredients of a modeling language as depicted in the Figure 2.3 and described as following [9, 34, 44]:

1. **Abstract Syntax** is the definition of the structure of a modeling language. The language that is defined is an artificial one and the designer of the language needs to define its valid phrases and how they can be combined with each other. Thus, an abstract syntax describes the elements of a language and the relationships between them, in other words, it is the grammar of the language.
2. **Concrete Syntax** is the visual representation or notation of the language elements defined in an abstract syntax. A user of a modeling language is usually only involved in working with the concrete syntax. For one modeling language, i.e., for its abstract syntax, we can create an arbitrary number of concrete syntaxes.  
 We can differentiate between graphical and textual concrete syntax.  
 Graphical concrete syntax means that the language elements are visualized as graphical shapes in an graphical (diagramming) editor, where language elements are created by placing the corresponding shapes on a diagram, and connections between them are usually visualized by drawing connection paths between shapes accordingly.  
 Textual concrete syntax is created by specifying the keywords for the elements defined in an abstract syntax. The connections between elements, in a textual syntax, can be specified, for example, by referencing elements by their name or an ID. The keywords are usually highlighted — color or font style highlighting — in a dedicated textual editor in order to easily differentiate them from other parts of the textual syntax. Most programming languages have a textual concrete syntax.
3. **Semantics** is a part of a language definition describing the meaning of the language elements and the meaning of different ways of combining those elements. Precise specification of the language semantics is very important in order to understand what the language constructs imply. In order to successfully use a language the user must familiarize himself with its semantics.

What we see in the Figure 2.3 is that the semantics defines the meaning of the abstract syntax and, indirectly also, the meaning of the concrete syntax, whereas the concrete syntax is a representation of the abstract syntax — to note, there can be many concrete syntax representations for the same abstract syntax.

These three core ingredients are mandatory for every modeling language, be it a GPL or a DSL, and if any of them is missing or incomplete then a modeling language is not well defined [9].

## Meta language

In order to create an artificial language, be it a textual or model-based, we need to have means for the precise definition of it. Having a language which exact purpose is to define another language is generally referred to as a *meta language* [44]. In very early days of computer science (in the middle 1950's), computer scientist were all about designing high-level programming languages as formal languages and building compilers for them [43]. To formalize the syntax of ALGOL programming language John Backus devised a notation that was simple, powerful and could be used to specify the syntax of any programming language, which was later revised and popularized by Peter Naur. Using such a notation, both a programmer and compiler can determine whether a program is syntactically correct - whether it adheres to the grammar and punctuation rules of the specified programming language [43]. In their honor, this notation is known as Backus-Naur Form (BNF). Widely-used variant of that notation (or meta language) is known as Extended Backus-Naur Form (EBNF). With EBNF we specify the grammar and the syntax of a language by using so-called *production rules* which describe how a *non-terminal symbol* expands to a valid combination of non-terminal and *terminal* (alphanumeric characters) symbols. Through the specification of the combination of production rules we specify valid sentences of a programming language. Consequently, in EBNF there is no strict separation between the abstract and concrete syntax of the language because the EBNF specification combines both of them, which means that we can use it only to specify textual languages — as most programming languages are. An example of an EBNF grammar specification for a Mini-Java programming language can be found in [47].

However, in MDE, as we know it by now, we use models at different abstraction levels to develop our software systems. For this, MDE employs modeling languages anchored at different abstraction levels and for different purposes. In this work we concentrate on a *metamodel-centric language design* (cf. Figure 2.4) concerning the task of developing a new modeling language. The term *meta language* is in MDE also known as the *metamodeling language*. Terms such as *model*, *metamodel* and *meta-metamodel* will be explained in the following.

## Model, Metamodel and Meta-Metamodel

Models play the key role in Model Driven Engineering. They are the driving force behind it and in the MDE ideology “*everything is a model*” [9]. So, what is a model?

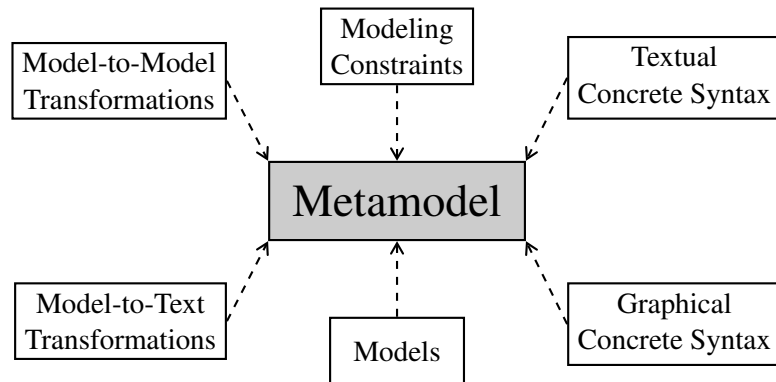


Figure 2.4: Metamodel-centric language design [9].

## Model

The simplest definition of the term would be “a model is an abstraction of reality”. Abstracting reality is nothing new for us humans, we do it all the time. We do it, for example, in order to simplify our cognitive processes.

The term model has many definitions, we will name the few, such as one from [38] defining it as following “*a model is an abstraction of a (real or language-based) system allowing predictions or inferences to be made*”, and another definition found in [29] (based on [3]) goes as following “*a model is a description or specification of a system and its environment, intended for a specific purpose, and is represented by a combination of drawings and text*”. There are many more definitions, but in the context of software engineering, simply put, a model is used to describe the software system under development and its environment.

Models may have also different purposes. In early days of using models in software engineering, they were used as sketches or blueprints to describe the software systems under development. In MDE models are first-class citizens, as we described it in the Section 2.1, and are used to produce executable software artifacts.

The model holds only relevant parts of the system definition and is therefore regarded as the abstraction of the system under development [44]. The OMG’s *four-layer metamodeling stack* illustrated in Figure 2.5 shows different modeling layers. The layers are ordered from lowest to highest and named M0, M1, M2 and M3. The instances of a model or snapshots of a system are found on the lowest layer M0. Artifacts found on M0 level conform to their abstract specification or models found, consequently, on the next higher level M1. The variety of elements that comprise a model must also conform to a more abstract formal specification of them, found on the next higher layer M2, and that specification is known as a *metamodel*.

## Metamodel

Among different ways to specify languages one way is by means of a metamodel, which represents the abstract syntax of a language [33] — we already described what abstract syntax is, above in the Language Characteristics. A metamodel is exactly the starting point for language

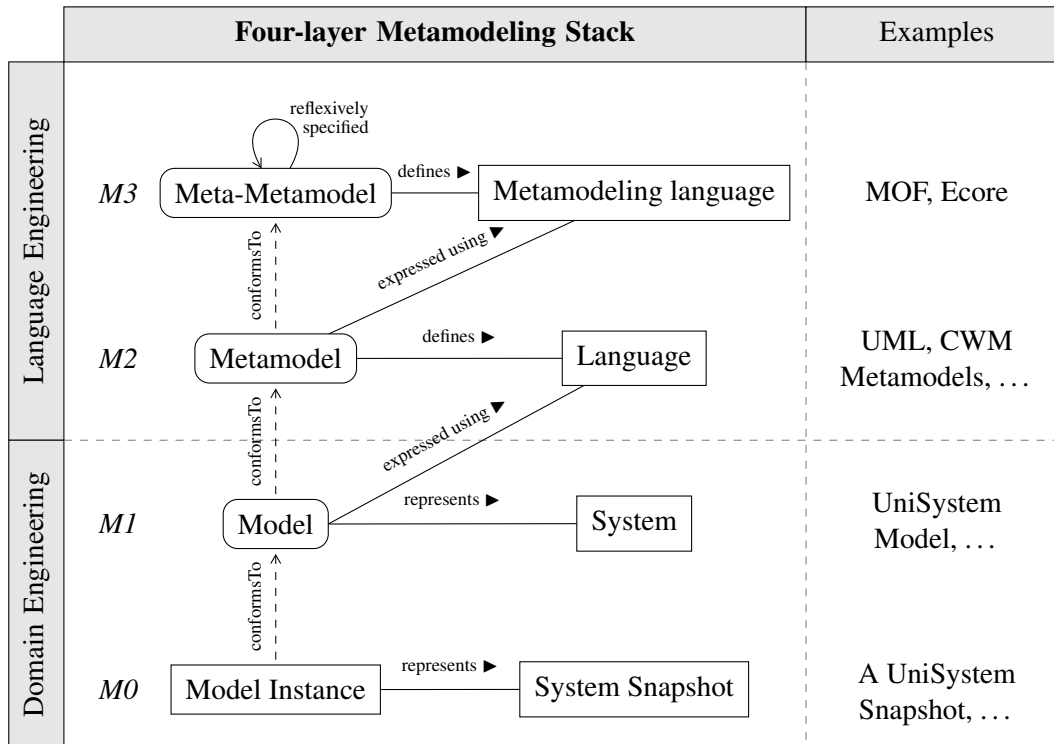


Figure 2.5: The four-layer metamodeling stack (based on [9,44]).

designers to begin their work on specifying a new modeling language. Within a metamodel we specify language concepts, their properties and their interconnections. This process is also known as *metamodeling* [9]. Therefore, a model is an instance of a metamodel, consequently, the elements of a model are instances of the elements of a metamodel [4]. Metamodels are defined with the usage of so-called *metamodeling languages*, as illustrated in Figure 2.5. Metamodels bring certain advantages as the means for designing modeling languages, as described in [9,44]:

1. Metamodels, as already mentioned, hold the concise definition of model elements. This means that only these language constructs, which are defined in a metamodel, are valid in derived models. Consequently, this leads to better common understanding of the specified modeling domain.
2. Similarly, because metamodels define model elements, models can be checked for syntactic validity as they have to conform to their respective metamodels. This makes metamodels together with corresponding constraint specifications — e.g., in OCL — an efficient control mechanism in specifying valid models for a software system.

3. Another important aspect of a metamodel is that it is not necessarily a rigid construct that can not be modified or extended at a later stage. There are different kinds of mechanism to extend or adapt an already existing modeling language. Since this master's thesis is concerned with exactly these tasks in MDE, we provide more information about the extension of modeling languages in Section 2.5.

As it is the case with a metamodel being the abstraction of a model, providing the formal specification of the model, also, one can only expect to find an abstraction of the metamodel as a formal specification of it too. Indeed, there is one, and its name is the *meta-metamodel*.

### Meta-metamodel

Looking at the Figure 2.5 we can recognize that the same way a metamodel specifies the abstract syntax of a model, so does a meta-metamodel specify the abstract syntax of the metamodel. The meta-metamodel can be found on the highest level M3 of the metamodeling stack. The Figure 2.5 also illustrates that the meta-metamodel defines a metamodeling language, in other words it defines a modeling language for defining another modeling languages. The elements of the meta-metamodel specification are used to specify the elements of a metamodel. Consequently, this means that the model elements of a metamodel conforms to the model elements of a meta-metamodel.

The two most widely acceptable metamodeling languages are:

- **MOF**

Meta Object Facility (MOF) Specification [25] is the OMG's standard metamodeling language. MDA principles were mostly influenced by MOF specification. It is the basis for all modeling concepts done by OMG. For example UML is specified with MOF.

- **Ecore**

The metamodeling language that is part of the Eclipse Modeling Framework (EMF) [50]. It is based on the essential subset of MOF (Essential MOF or EMOF), which is designed to match the capabilities of object oriented programming languages and mappings to interchangeable formats [25] such as XMI<sup>10</sup>. Most notably, Ecore provides a Java-based implementation of most important concepts from EMOF. Ecore and EMF in particular are very well integrated within Eclipse technologies and provide to a language designer tools such as editors, views, and generators. Ecore is well suited and used mostly for the specification of DSLs.

We concentrate our work around EMF technologies and modeling languages specified with Ecore.

Both MOF and Ecore are heavily based on the core of structural object-oriented modeling languages. In other words, the syntax of these languages is similar to that of an UML class diagram. So, most notable concepts of these languages are *classes*, *attributes* and *associations*. These concepts are very simple in their nature which makes them generally applicable, even

---

<sup>10</sup>OMG's XML Metadata Interchange (XMI). <http://www.omg.org/spec/XMI/>



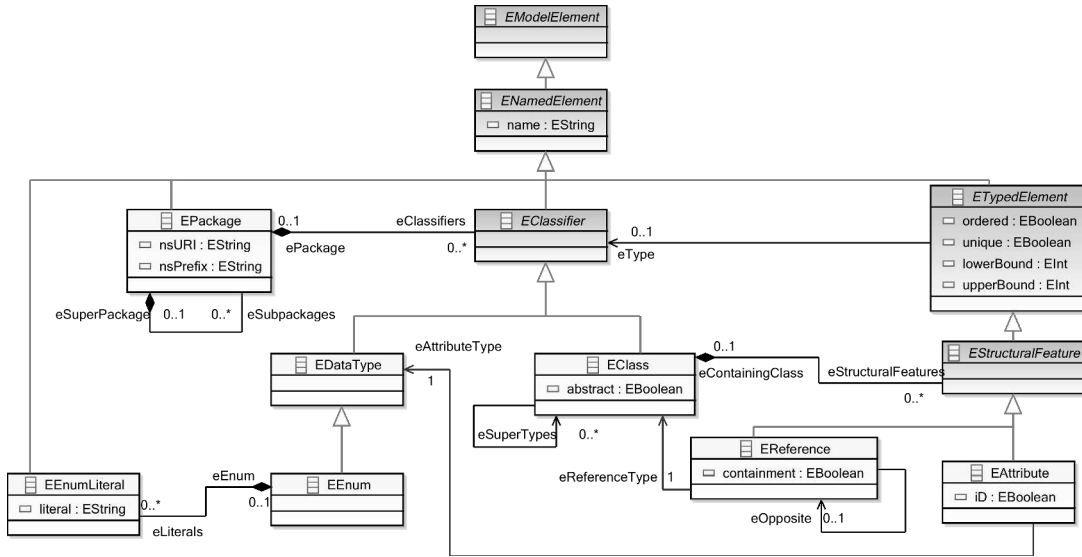


Figure 2.6: Overview of Ecore’s main language concepts [9].

for their own specification. That is exactly the reason why M3 level in the metamodeling stack is the highest one (cf. Figure 2.5). A meta-metamodel is reflexively specified with its own language constructs, in other words it defines itself and renders any higher level of abstraction unnecessary. An overview of Ecore modeling concepts can be viewed in Figure 2.6.

## 2.3 Transformation Engineering

The fact that in MDE a model of a software system is not only a pretty representation of the system in form of a picture but rather a formal specification of the system that conforms to a modeling language formal specification, known as abstract syntax or metamodel, allows us to leverage that formal information in order to transform the elements of a model conforming to one modeling language into the elements of another model conforming to another modeling language. Transforming the elements of a model into a formal syntax of some programming language is also possible.

Transformation engineering is concerned with exactly these tasks, and it plays the pivotal part in MDE and modeling languages based on the metamodel-centric language design (cf. Figure 2.4). The information from models and their metamodels is usually all the information we need to specify transformation rules. A transformation, in general sense, is a program that takes as input one or more models, does transformations based on some logic, and produces one or more output models. We can differentiate between two types of transformations [40]:

**Model-to-Model (M2M) transformation** has different use cases, such as, transforming models from one modeling language into models of another modeling language, which is also

known as *exogenous* transformation, and transformation of models into the models within the same modeling language — merging different models, refactoring models, etc. — which is known as *endogenous* transformation.

**Model-to-Text (M2T) transformation** is about generation of textual fragments populated with informations provided in models. These transformations are usually used for automating several software engineering tasks such as the generation of programming code, documentation, deployment scripts, and so forth.

Transformation engineering and writing transformation rules involves formal languages commonly known as *transformation languages*. For example, for M2M transformations we have OMG's Query/View/Transformation (QVT)<sup>11</sup> standard set of transformation languages, or for models constructed within EMF using modeling languages specified with Ecore we can use ATLAS Transformation Language (ATL)<sup>12</sup>. In general, these languages provide declarative or imperative language constructs — or both — for definition of transformation rules.

Transformation engineering is very crucial to the MDE approach, but since this master's thesis is not involved in transformation engineering we will not go into any more details on the subject. The work done in [44] has more information, if you are interested to learn more.

## 2.4 Concrete Syntax Development

In order to effectively use a newly designed language we need to specify an adequate concrete syntax for that language. The concrete syntax must be intuitive, concise and suited to the domain concepts for which the language was build. To be successfully employed, the concrete syntax of a language must support its user by providing a way to easily spot and grasp the most relevant domain concepts. As we already mention it in the Language Characteristics section, we can specify a *Textual Concrete Syntax* (TCS) or a *Graphical Concrete Syntax* (GCS). Which one is more suitable to represent domain concepts of the language is probably the one that provides more expressiveness and ease the creation and manipulation of model elements — but, mostly it is a matter of a user's personal opinion. Sometimes it is also suited to provide both concrete syntax specifications for a language, leaving the choice of which one is preferable to the user.

For a user of a language the textual concrete syntax provides a one-dimensional view on a model. A model is specified with textual notations describing or specifying elements of the model in a top-down fashion — a vertical view, from top to bottom. In contrast to the textual concrete syntax, a graphical concrete syntax provides a spacial view on the elements of a model — a two-dimensional view. We can, for example, order the graphical representations of the model elements in arbitrary horizontal and vertical position on a canvas, which allows, for example, to structure and place the elements of the model based on some special meaning in grouping or layering them together in order for user to easily grasp what the model specifies. The graphical concrete syntax mostly looks like graphs, with graphical nodes representing model elements and edges between them to describe the relationships between model elements.

---

<sup>11</sup>OMG's Query/View/Transformation. <http://www.omg.org/spec/QVT>

<sup>12</sup>ATLAS Transformation Language. <http://www.eclipse.org/at1>

The process of specifying concrete syntax for a language, regardless of the kind of concrete syntax, is defined by mapping the modeling concepts described in the metamodel of the language to their *visual notation*. In other words, visual notations introduce symbols, be it textual or graphical symbols, which allow to visualize models as textual or graphical elements in a diagram [9] (cf. Figure 2.7).

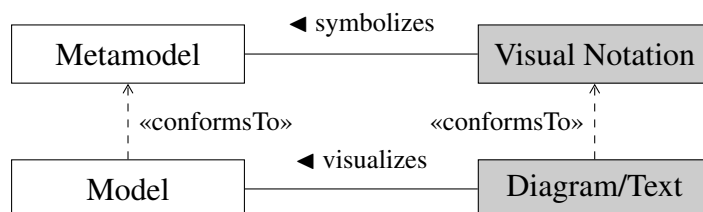


Figure 2.7: Visual notation: introducing symbols for modeling concepts [9].

The end product of a concrete syntax specification that a user of a language uses and that a language designer builds is a tool commonly known as an *editor* — a textual editor or a graphical editor, depending on which concrete syntax it is created for. In the following we will go more into detail about specification of the concrete syntax for a modeling language. We concentrate here on technologies and frameworks that are part of the Eclipse Platform, and designed to work in conjunction with EMF framework and technologies based on it.

## Textual Syntax Development

Modeling languages, especially in the beginning of MDE, were considered to be solely graphical languages, mainly influenced by precursors of UML and “*a picture is worth a thousand words*” reasoning. In the meantime we know that having a textual concrete syntax (TCS) for a modeling language increases its usefulness, especially if its users are more familiar working with textual documents to specify domain information. Due to powerful frameworks, emerged in the last few years, specifying TCS and creating textual editors for it has become much user-friendly, which changed the view at modeling languages (being solely graphical) fundamentally.

Informations described with textual languages are constructed by a collection of textual symbols. Not every symbol or wording constructed out of them, as well as their placement in the text is valid — surely we recognize this from programming languages. To define valid symbol sequences and their grouping *grammar specifications* are used. The grammar of the TCS enables domain data in textual form to be parsed into model data, and also the other way around, rendering model data into textual representation.

Readability and usability of the textual concrete syntax can be much improved if the syntax specification supports the designer of TCS in, for example, specifying language-specific keywords or some other syntactic sugar. In order to enable this, textual languages must support the language designer in specification of following concepts [9]:

- **Model information:** That the textual concrete syntax should support model information such as model element name or attribute type, goes without saying.

- **Keywords:** They are special words in the syntax conveying a concise meaning of the language constructs, and are used, for example, to denote different elements of a modeling language.
- **Scope borders:** All element descriptions that are conceptually part of another element description should be grouped or syntactically enclosed between so-called scope borders. A good example of symbolizing scope borders are curly brackets, known from programming languages, such as, e.g., C or Java.
- **Separation characters:** They are needed, for example, to separate language constructs from each other when they are syntactically grouped as part of another language construct. An example, also coming from programming languages, would be using semicolons to separate programming expressions or comma to separate the elements of a collection.
- **Links:** Linking or referencing a model element from another one is in textual syntax achieved by referencing the unique name (an ID) specified for that element. This is exactly how it is done in, e.g., object oriented programming languages, where every class must have a unique name in its name space.

From the point of view at the development of TCS we can differentiate between two different approaches, namely having either a generic TCS or a language specific TCS.

The generic TCS, as it is in the case of the specification for serialization of model data into its textual form, and vice versa (e.g., XMI specification) the generic TCS is well suited to be consumed by a machine, but not so well by a human.

For language specific TCS engineering approaches, also two approaches can be distinguished [9]:

- **Metamodel first** approach is about specifying the metamodel of a modeling language first, and considering it as the central language artifact, upon it the textual production rules are defined for the elements of the metamodel. Thus, the concrete syntax is specified on top of the abstract syntax. This approach is followed by different projects such as Textual Concrete Syntax (TCS) project<sup>13</sup> — uses text production rules similar to EBNF — and EMFText<sup>14</sup>.
- **Grammar first** approach is inspired by EBNF approach in the way that the language designer first specifies the grammar of the language — the abstract and concrete syntax are specified together as textual production rules in the grammar. Following grammar specification, the metamodel of the language is automatically inferred from the grammar by dedicated metamodel derivation rules. A very powerful framework designed with this approach in mind is Xtext<sup>15</sup>. Actually, Xtext can be used both ways, either by starting with grammar specification or metamodel specification for which we can generate default grammar specification and further refine.

With either approach, at the end we have the same language artifacts — the metamodel of the language and its TCS specification.

<sup>13</sup>Textual Concrete Syntax (TCS) project. <http://www.eclipse.org/gmt/tcs>

<sup>14</sup>EMFText project <http://www.emftext.org>

<sup>15</sup>Xtext Eclipse project. <http://www.eclipse.org/Xtext>

## Graphical Syntax Development

In contrast to text-based language engineering and also the specification of the abstract syntax of a language — as these processes are supported by formal definitions, e.g., EBNF grammar; or a standard such as OMG’s MOF standard — in the case of specifying graphical concrete syntax there is only one OMG standard available, namely the Diagram Definition (DD) [24] specification. DD specification provides basis for modeling and interchanging graphical notations — specifically node and arc style diagrams. This specification replaced the OMG’s Diagram Interchange (DI) [22] specification, which goal was to provide smooth and seamless exchange of graphical concrete syntax model representations between different software tools. For example, the Graphical Modeling Framework implements DI specification. Having the formal definition of the graphical concrete syntax allows us to leverage other advanced tools such as automatic generation of editor code, for example. Without formal definition we would most probably have to manually implement editors supporting visual notations of the language, which would require much effort and high investments in realizing a MDE environment [9].

Nowadays, there are frameworks, which leverage formal definitions and provide, for example, their own domain specific languages for the specification of the graphical concrete syntax. Not all frameworks adhere to the OMG’s specification but have rather their own specifications for visual notations representing modeling concepts. This may lead to different visual notations of the same modeling concepts at runtime depending on which framework we used to specify graphical concrete syntax. We will introduce some frameworks later on, but first, let us elaborate on the anatomy of the graphical concrete syntax, as described in [9].

**Anatomy of graphical languages.** Typically, a graphical concrete syntax is composed of following parts: (1) **graphical symbols** such as lines, shapes or complete figures, and labels representing textual information (e.g., the name of the modeling element); (2) **compositional rules** that define how the graphical symbols can be combined to create more complex visual representation for a modeling element. These rules also define, for example, positioning and styling of different graphical symbols inside the compartment of a figure; (3) **mapping** of the graphical symbols to their counterparts found in the abstract syntax, which they represent. For example, a rectangle shape could represent some key modeling element, and a label placed inside of that rectangle might be mapped to the name of that modeling element or any other of its attributes.

An editor built to support GCS provides a modeling canvas that allows positioning of model elements in a two-dimensional raster. For this purpose, every visual notation has an assigned  $x, y$  coordinate as the mean of specifying the position of its top-left corner from the top-left corner of the visual notation that contains it. Modeling elements are usually arranged in form of a graph contained within the modeling canvas. This arrangement, in form of a graph, is usually called a *diagram*, and it represents the graphical view on the model. In order not to overload the diagram with every detailed information of modeling elements, but rather to provide a more concise view on the model, only most relevant information can be visualized in the diagram. Other information is then visualized and accessible in dedicated tools known as *views*.

To further elaborate on the anatomy of GCS and its specification, Figure 2.8 illustrates a small excerpt of a generic metamodel for graphical concrete syntaxes [9]. We can see that a `Diagram` is a collection of different `DiagramElements`, which can be nodes, edges, labels

and compartments. Most modeling languages only need these diagram elements to provide sufficient visual representation of its models. Nodes are visualized as shapes, edges as lines, compartments are usually shapes that allow nesting of other diagram types, and labels allow to annotate, for example, nodes and edges with additional information. Actually, all these are concrete parts of the illustration itself.

To state the relationship between the modeling concepts and their visual representations we employ a mapping between abstract syntax and concrete syntax elements.

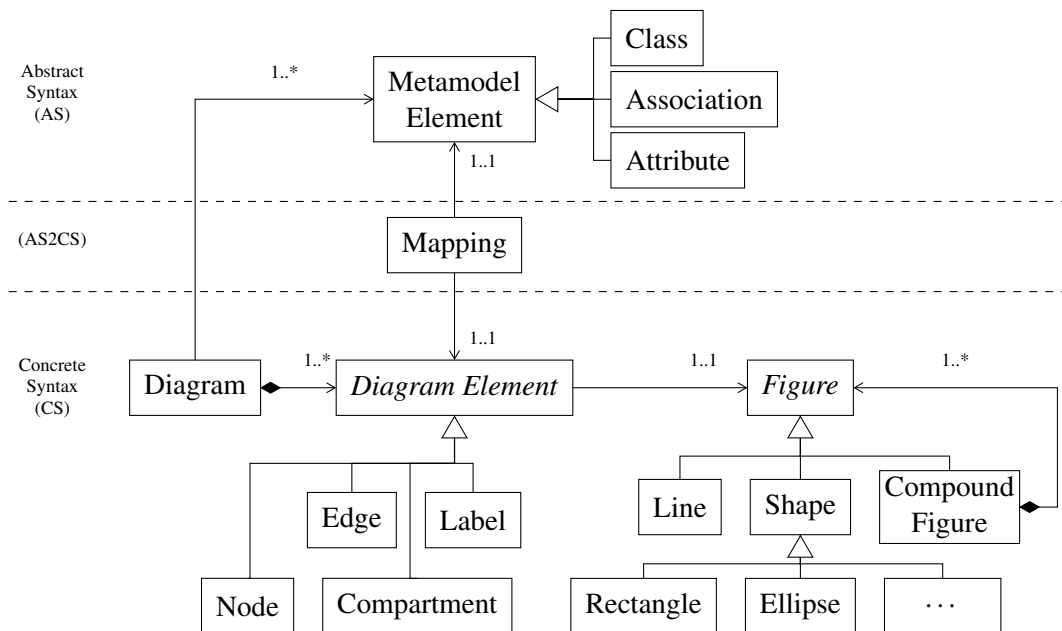


Figure 2.8: A generic GCS metamodel [9].

**Approaches to GCS development.** The Eclipse platform and its graphical environment are governed by toolkits (as described in [45]) such as Standard Widget Toolkit (SWT)<sup>16</sup> and JFace. SWT provides portable access to user interface facilities of the underlying operating system and JFace provides helper classes for SWT to solve common UI problems which are implemented following the Model-View-Controller (MVC) paradigm.

For drawing and layout of graphical components on the two-dimensional SWT canvas the Draw2d<sup>17</sup> framework is responsible. Draw2d also provides facilities for event handling such as listening to focus, keyboard or mouse events.

<sup>16</sup>Standard Widget Toolkit for Java. <http://www.eclipse.org/swt/>

<sup>17</sup>Draw2d is a layout and rendering toolkit building on top of SWT <http://www.eclipse.org/gef/draw2d/>

Building upon them we have the Graphical Editing Framework (GEF)<sup>18</sup> — nowadays also considered to be low-level — for developing graphical editors. GEF is designed with MVC architecture in mind, separating data model from its respective graphical representations and linking them together through the controller. Furthermore, GEF controller — also known as *EditPart* in GEF jargon — besides mapping both model and view together also has the responsibility of updating, e.g., the view if the data model changes, or the other way around if user makes changes on the view it updates the data model. GEF does not set any restrictions about the models to be used with it, which makes it well suited for GCS development of a modeling language.

Currently there are few different Eclipse projects for specification of GCS tailored for modeling languages based on EMF, and are built upon GEF. They can be differentiated in their approach on how to specify and develop the graphical concrete syntax of a modeling language, as stated in [9]:

**API-centric GCS.** This approach is about providing more powerful and easier to learn and use API than what GEF offers. The user programs against the dedicated API in order to develop GCS of a modeling language. The most notable protagonist of this approach is Graphiti<sup>19</sup> project. More about Graphiti later on.

**Mapping-centric GCS.** The protagonists of this approach provide dedicated modeling languages for the GCS specification and mapping between abstract and concrete syntax elements [9]. The most notable protagonist of this approach is the Graphical Modeling Framework (GMF)<sup>20</sup>. GMF provides different modeling languages and graphical editors for them in order to specify GCS and mapping. From the created descriptions in form of models GMF provides code generation facility which transforms these descriptions into editor code. Later we provide more details on GMF.

Another notable protagonist of the mapping-centric approach is the Spray<sup>21</sup> project. It also provides different modeling languages for GCS and mapping specification. The only difference compared to GMF is that it uses textual editors (textual modeling languages). The goal of Spray project is to simplify description of a modeling language GCS against the Graphiti runtime, and provide code generation to create the boilerplate code for realizing the implementation against the Graphiti framework.

In both GMF and Spray, realizing graphical modeling editors is achieved by applying MDE techniques.

**Annotation-centric GCS.** Annotating the metamodel of a modeling language with descriptions of visual representations for the model elements is what this approach propagates. The most notable protagonist of this approach is EuGENia<sup>22</sup>. EuGENia is all about abstracting and simplifying the GCS specification as it can be done with GMF. The GCS information necessary for the implementation of a graphical editor is captured by embedding

---

<sup>18</sup>Graphical Editing Framework (GEF) provides technology to create rich graphical editors and views for the Eclipse Workbench UI. <http://www.eclipse.org/gef>

<sup>19</sup>Graphiti project. <http://www.eclipse.org/graphiti>

<sup>20</sup>Graphical Modeling Framework. <http://www.eclipse.org/modeling/gmf>

<sup>21</sup>Spray project. <http://eclipselabs.org/p/spray>

<sup>22</sup>EuGENia. <http://www.eclipse.org/epsilon/doc/eugenia/>

high-level annotations in the Ecore based metamodel of a modeling language [36]. Then, by model transformations EuGENia produces GMF models and reuses GMF facilities to generate editor code. Consequently, by abstracting upon GMF EuGENia hides the complexities of GMF.

EuGENia as well as aforementioned Spray project are very useful in helping a developer to jump-start the specification of GCS of a modeling language and development of GMF-based and Graphiti-based editors, respectively.

In fact, we use these frameworks for our case study in Chapter 7.

In following we provide more information on GMF and Graphiti framework.

### Graphical Modeling Framework

The Graphical Modeling Framework is an open source framework for development of diagram editors. It is the prime example of MDE (and MDA of course) approach because it leverages modeling languages and strictly separates PIMs, PSMs and code [26], as can be seen in Figure 2.9. GMF is a composition of three different components [21]:

1. *The GMF Tooling* provides a model-driven approach for generating graphical editors. This is the component that as a language designer we are most interested with.
2. *The GMF Runtime* is an industry proven application framework for creating graphical editors using EMF and GEF. It provides a set of common features such as printing, actions, toolbars, image export, and more. It bridges command frameworks, such as those of EMF and GEF, and provides extendibility that allows graphical editors to be open and extendable at runtime.
3. *The GMF Notation* provides the standard EMF notation metamodel with standard means and purpose of persisting diagram information separately from the domain model. It is based on OMG's DI specification and therefore can also be used as basis for diagram interchange.

GMF brings a set of four modeling languages, a transformator that maps PIMs to PSMs, a code generator that turns PSMs into code, and the runtime platform on which the generated code relies [26]. As a designer of the graphical concrete syntax for a modeling language we would usually only work with modeling languages from GMF Tooling. For a *domain model* — the metamodel of our modeling language — we use the *gmfgraph* language for describing graphical elements such as nodes and edges in the diagram, the *tooldef* language to describe tools available to the user of the diagram, and the *mappings* language to produce a *mapping model* that combines the other three views to an overall view which maps the graphical elements from the *graphical definition model* and the tools from the *tool definition model* onto the domain model elements from the domain model. From the mapping model the transformator generates a *diagram generation model* that conforms to the *gmfgen* language. This generated model can further be augmented by the designer and at the end it holds enough information that can eventually be consumed by a set of model-to-text transformations in order to generate the Eclipse plug-ins that contain the actual Java code that implements the editor.



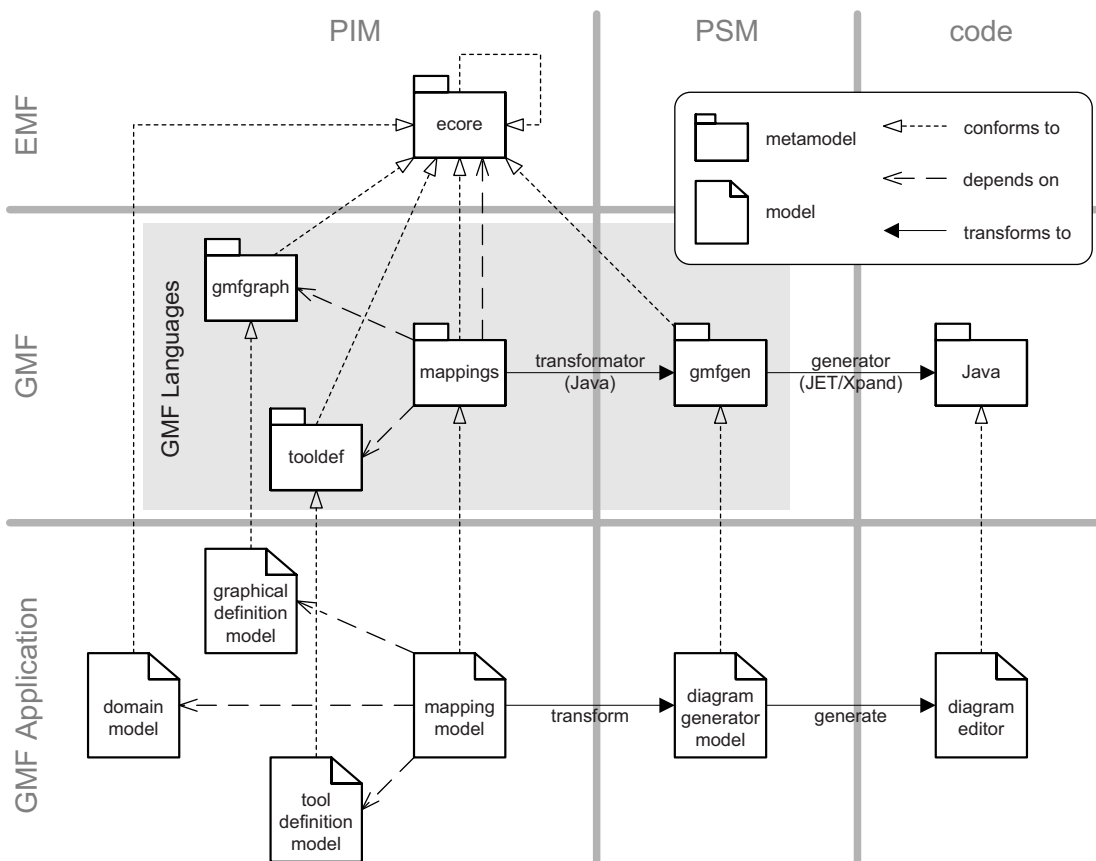


Figure 2.9: Languages involved in the Graphical Modeling Framework [26].

## Graphiti

As opposed to GMF, Graphiti does not use any code generation facilities for creating diagram editors. It solely provides its API for definition of the graphical concrete syntax, as well as for EMF-based domain models as for Java-based domain models. For its usage and development of GCS, Graphiti provides following benefits [16]:

- *Low-entry-barrier*: it hides platform specific technologies such as GEF and Draw2d.
- *Incremental development*: provides fast-payoffs by using default implementations, thus shorter development cycles.
- *Homogeneous editors*: editors on top of Graphiti look and behave similarly.
- *Optional support of different platforms*: diagrams are defined platform independently, thus if GEF and Draw2d would be replaced with other underlying platform technologies, the same diagram definitions could be rendered on different platforms without any necessary adaptations.

Graphiti has a big advantage as opposed to GMF, and the reason is while in Graphiti one programs against plain old Java interfaces and all adaptations for an editor are done directly in the code, in GMF one must manipulate generated sources to adapt the editor, which can cause headaches when regenerating [16]. The architecture of Graphiti is solely runtime oriented in contrast to GMF having also a generative component.

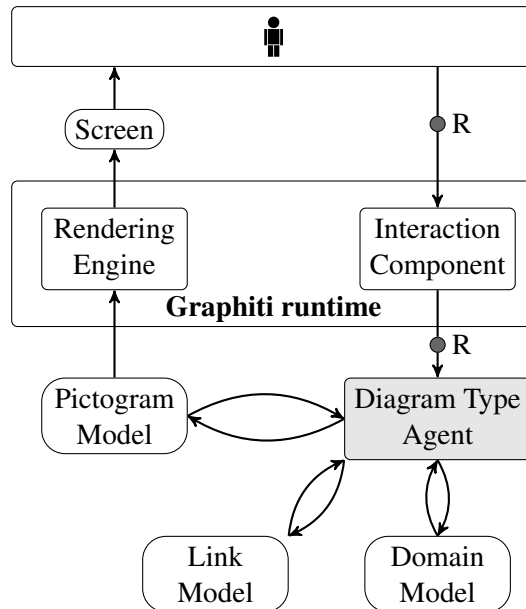


Figure 2.10: Graphiti architecture (based on [16]).

Graphiti architecture, depicted in Figure 2.10, shows that the communication between a Graphiti-based tool and a user happens over the screen and the user generates events, e.g., keyboard or mouse events, that are registered by the `Interaction Component` as specific user requests. The `Rendering Engine` is the part of the framework responsible for rendering the current data on the screen. `Interaction Component` and `Rendering Engine` form together the Graphiti runtime. Technically the runtime is based on GEF and Draw2d, but the Graphiti programming interfaces are self-contained, meaning no knowledge of or access to underlying technologies is necessary. Similarly, Graphiti defaults ensure similar look and feel of all Graphiti based tools.

Handling of the registered user requests is done by a `Diagram Type Agent`. This is the component that a developer must manually provide by building it upon various services and default implementations that the framework provides.

**Domain model** is the metamodel of our modeling language. **Pictogram model** contains the complete data from domain model as well as information how these data is represented in a diagram. As a consequence the pictogram model contains redundant data which allows simultaneous manipulation of domain model by Graphiti and any other tools. Redundant data is then synchronized with usage of so-called *Update Features*. **Link model** connects the elements

of the domain model with the elements of their graphical representations in pictogram model.

GMF and GEF are based on Model-View-Controller paradigm, however Graphiti is not. It employs so-called *features* which encompass all concepts found in MVC such as creation, deletion and changing of domain model elements, as well as creation, deletion and updating of visual elements.

Graphiti, at the moment of writing this thesis, is still in its incubation phase as an Eclipse project, meaning it is still in the phase of becoming a fully-functioning open-source project.

## 2.5 Extending Modeling Languages

In Chapter 1 we already gave an introduction to the topic of the extension of modeling languages, as well as to two possible ways on how to approach it, namely the *heavyweight* vs. *lightweight* language extension. In this section we go straight to describe the concepts of EMF Profiles — as they are at the core of our work in this master’s thesis — and in the subsequent *related work* Section 2.6 we introduce other approaches for extending modeling languages in the lightweight manner.

### EMF Profiles

EMF Profiles, a solution to the extension of modeling languages based on EMF technology was proposed in the work of Langer et al. in [39]. EMF Profiles are an adaptation of the UML profile concept to the realm of Domain-Specific Modeling Languages (DSMLs) and they provide a lightweight language extension mechanism.

The Profile mechanism is based on a *profile definition* comprised of *stereotype definitions*. *Stereotypes* are used to annotate model elements in order to refine their meta-classes by defining supplemental information in form of additional *meta attributes*, also known as *tag definitions*. Instances of tag definitions are known as *tagged values* and they are used for the provision of new informations to existing models.

The metamodel of the EMF Profiles definition language is illustrated in package *Standard EMF Profile* depicted in Figure 2.11. The precise definitions of the terms profile, stereotype and tagged value are given in the following.

### Profile

A Profile is a package of related and coherent extensibility elements in form of stereotypes and tagged values [9]. These extensions are usually covering some particular requirements or purposes and profiles are used for grouping them together into their domain-specific extensions. A profile is usually denoted in a diagram by the keyword *«profile»* followed by its name. In the EMF Profile language definition (cf. Figure 2.11) we can see that the class `Profile` inherits from the Ecore meta-metamodel class `EPackage`. The `Profile` class has a `base` reference to an `EPackage`, which at the instantiation time is set to point to the package of a modeling language which elements will be extended through stereotyping. The EMF Profiles implementation uses for instantiation and application of a profile the *Profile Application* metamodel (cf. Figure 2.11). Its purpose is to weave the necessary concepts for a profile application into a profile

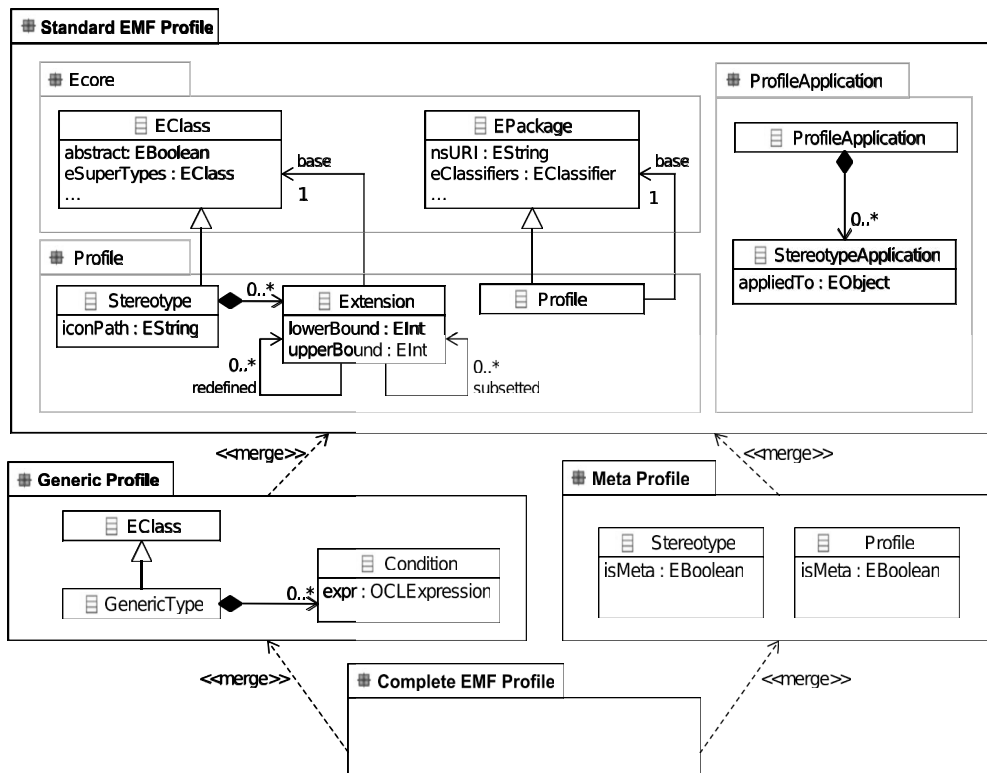


Figure 2.11: EMF Profile Metamodel [39].

model [39]. The class `ProfileApplication` denotes the root element for all stereotype applications in a profile application model.

## Stereotype

Stereotypes play the main part in EMF Profiles and, as already said, they are used to annotated elements of a modeling language in order to specialize their meta-classes with respect to some domain, platform or some specific functionality [5, 44]. By applying a stereotype to a metaclass, the metaclass gets bound to a defined purpose or a usage context [44]. In a diagram a stereotype definition is usually denoted by the keyword `«stereotype»`. In the EMF Profile language definition (cf. Figure 2.11) we can see that the class `Stereotype` is a specialization of the `EClass`. Consequently, that brings a nice side effect with it, namely reusing the `EAttribute` and `EReference` elements to represent tagged values. To specify the applicability of stereotypes to metaclasses, the class `Stereotype` has a reference to the class `Extension`. Hence, the class `Extension` is used to denote the base metaclass and also the applicability bounds. For example, setting the `lowerBound` of the `Extension` class to 1 would mean that the stereotype must be applied to each instance of the base metaclass, in order to obtain a valid profile application. Stereotypes, when applied, need to have a reference to the model elements

to which they are applied. As in the case of profiles, the EMF Profiles implementation uses the *Profile Application* metamodel to instantiate and apply a stereotype to a base class. For this purpose the *Profile Application* metamodel contains the `StereotypeApplication` class which holds the `appliedTo` reference pointing to an arbitrary `EObject`. Consequently, whenever a profile is saved, the `StereotypeApplication` is automatically added as the superclass to each specified stereotype — this is possible because the `Stereotype` is an `EClass` and the `EClass` can have a superclass. Due to the fact that the `Stereotype` inherits from the `EClass`, the stereotype can be defined as the specialization of a different stereotype, thus we can use inheritance when defining stereotypes. To visualize the application of a stereotype usually an icon is placed as an adornment in the upper-left corner of the visual representation of the model element to which the stereotype is applied. The `iconPath` property in the class `Stereotype` is meant to be used for this purpose.

### Tagged Value

Tagged values are nothing else then a tag-value pair that can be attached as an attribute or a reference to a stereotype [9], hence a property of a class. Consequently, we can specify the name and the type of the tagged value. The type can be a simple data type, such as `String`, `Integer` or `Boolean`; or it can be a complex data type, such as a user defined data type. Their purpose, as already said, is to allow the designer to specify additional information to the base metaclasses, which may be necessary to accomplish another modeling tasks such as model transformations, execution of a model, or adding project management data to model instances.

Besides the *Standard EMF Profile* the EMF Profile Metamodel also contains the *Generic Profile* and *Meta Profile* specifications. These represent the two novel techniques introduced in the EMF Profiles [39], for reusing profile definitions. *Generic profiles* are defined independently of any specific modeling languages in mind and may be bound at later point to several user-defined modeling languages. *Meta profiles* fall in the category of those profile definitions that can be immediately reused for all modeling languages — of course, the requirement is that these modeling languages are based on the `Ecore` metamodeling language.

An example of how EMF Profiles implementation technically represent profiles and their profile application can be found in Figure 2.12. The example depicts an Entity-Relationship (ER) model [11] and a simplified version of an well-known EJB profile — more information on EJB can be found in Chapter 3. The illustration in Figure 2.12(a) depicts the definition of EJB profile with the stereotype `Bean`, which extends the metaclass `Entity` from the ER metamodel, and two concrete subclasses of the `Bean` called `SessionBean` and `EntityBean`. Furthermore, the profile defines another stereotype `IDAttribute` extending the metaclass `Attribute` to denote the ID of an `Entity`.

Figure 2.12(b) depicts internal representation of the profile application. At the moment of instantiation of (i.e., applying) the EJB profile, a root element of the type `ProfileApplication` is created, which contains possible stereotype applications as depicted in Figure 2.12(c).

Also, very important to note is that the *EJBProfileApplication* model resides in a separate file and not in the original ER model file denoted with *BaseModel* in Figure 2.12. This keeps the original model unchanged and allows for arbitrary profile applications at the same time.

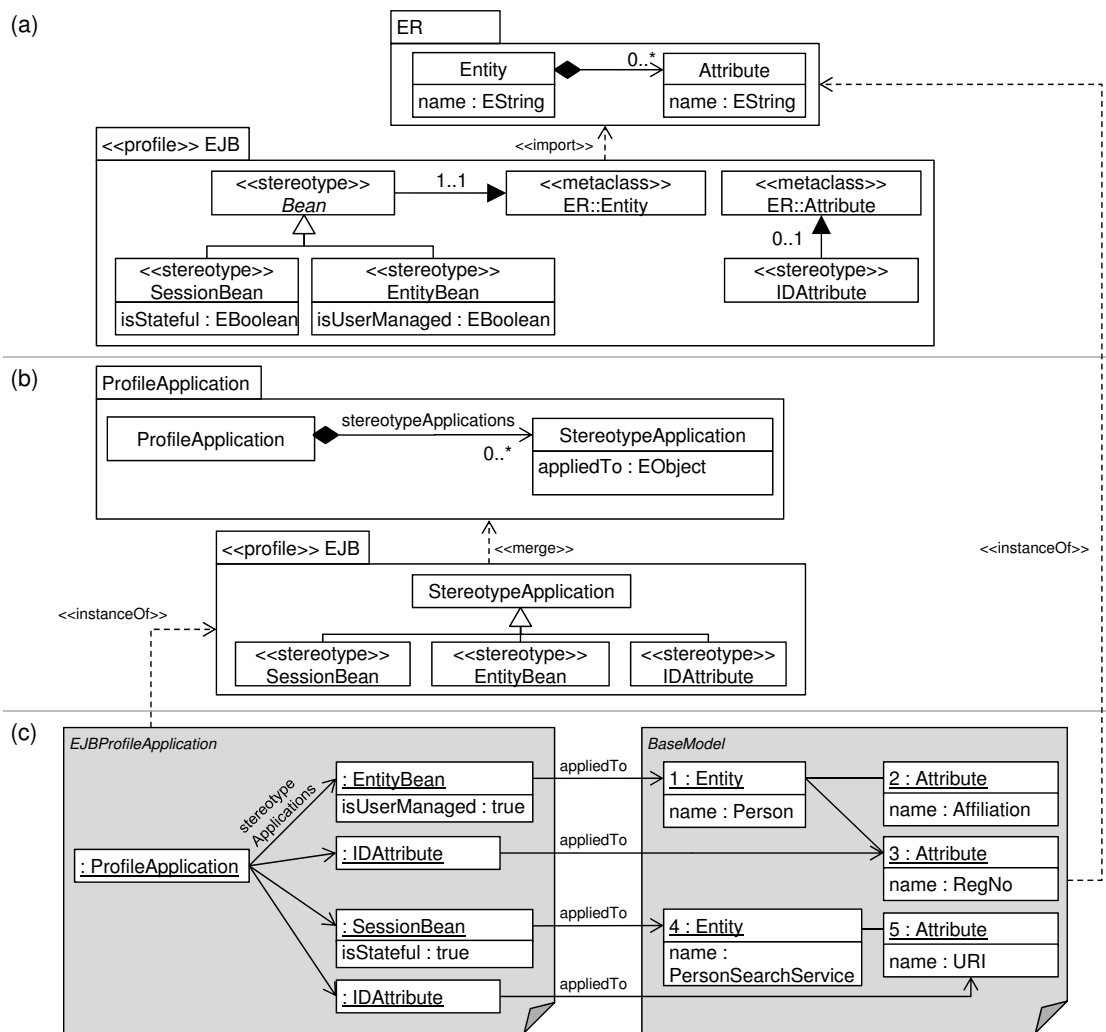


Figure 2.12: EMF Profiles by Example: (a) Profile Definition User-View, (b) Internal Profile Representation, (c) Profile Application [39].

The EMF Profiles, as the acronym suggests, is an Eclipse project. The implementation brings a dedicated graphical editor for specifying new profiles. It also ships with a dedicated *Profile Application View* to visualize applied profiles and their stereotype applications on the model that is currently displayed in the modeling editor. Furthermore, in order to manipulate the tagged values of applied stereotypes the implementation provides a dedicated implementation of the Eclipse properties view. Besides these mentioned components the solution implements also other tools, such as for the purpose of managing profiles and profile applications, a helper tools for creating a profile definition project or applying a profile to a model instance.

At the moment, addressing the extension of the graphical concrete syntax of a model diagram with EMF Profiles does not go beyond the mentioned icon decoration.

## 2.6 Related Work

For extending a modeling language in a lightweight manner by means of an annotation mechanism there are several alternative approaches besides the EMF Profiles.

For example, in Fritzsche et al. an approach is presented that leverages the model weaving mechanism [19]. With such a mechanism it is possible to compose different separated models, and thus, could be used to compose a core model with a concern-specific information model in a non intrusive manner. However, as mentioned in [39], although this is a powerful mechanism, it is also counter-intuitive for manually annotating models with weaving models, because this was not their intended purpose.

Another approach, called *Model Decorations* [37], addresses a very similar goal as EMF Profiles [39]. Additional information is attached (or *decorated*) by means of text fragments embedded in GMF's *diagram notes*. In order to extract or inject decorations from or into a model, a user must write transformations for each target model to enable the translation of the text-fragments in the notes into a separate model and vice versa. Important to note is that this approach can only be applied on models visualized in GMF diagrams. EMF Profiles do not have this restriction.

*EMF Facet* [14] represents a complementary extension direction compared to EMF Profiles. With EMF Facet a user can dynamically extend models with additional transient information computed from existing model elements by model queries expressed in, e.g., Java or Object Constraint Language (OCL) [52]. In contrast to EMF Facet, with EMF Profiles a user can add new information — not only derived one as in case of EMF Facet — and persist it in a separate file.

Meta-packages [12] is a concept for the lightweight extension of the structural modeling language XCore, which is based on packages, classes and attributes. Compared to meta-packages, EMF Profiles are more generic, because not only one modeling language may be extended, but any Ecore-based modeling language [39].

Although these approaches provide a lightweight language extension, they are centered around the extension of the abstract syntax. None of them addresses the extension of the concrete syntax in a way we would like to have.

In our search for related work, we could not find any work that tackles the problem of *extending* not only the abstract syntax but also the concrete syntax of modeling languages. To the best of our knowledge, there are only approaches for specifying the concrete syntax from scratch, which are summarized in the following:

- Graphical Modeling Framework (GMF) and Graphiti framework from the Eclipse Graphical Modeling Project (GMP) [15] provide a set of generative components and runtime infrastructure for developing graphical editors based on EMF and GEF. GMF and Graphiti framework have been introduced in Section 2.4.
- EuGENia [36] builds upon GMF by raising the level of abstraction. Their approach is annotating the domain model elements in the metamodel with the information of the concrete syntax representation. This is, however, contradictory to the separation of concern paradigm, because the domain model is polluted with the concrete syntax description.

- YTUNe [51] is an approach very similar to EuGENia. In YTUNe the elements of the concrete syntax are included in the abstract syntax.
- Spray [1] aims to provide one or more Domain Specific Languages (DSL) to describe Visual DSL Editors against the Graphiti runtime, and provide code generation to create the boilerplate code for realizing the implementation against the Graphiti framework.

All of them have the same goal, which is to provide a simpler way for defining the graphical concrete syntax for DSMLs, with respect to how it can be done with GEF only.

Although, our approach is not about defining the concrete syntax for a DSML — but rather the extension of an already defined one — information provided in these related works can be beneficial to our work, because we also need to define language concepts for describing the extension of the graphical concrete syntax.



## Running Example and Requirements

In this chapter we establish a common ground for better grasping and understanding the information we provide in this work later on. Topics addressing difficulties that may occur when extending a modeling language, or how the specification of a decoration in *decoration description language* will manifest itself graphically in an editor, and so on. So, whenever needed we use the following example as our *running example* throughout this work.

In the second section of this chapter we specify the requirements for our solution.

### 3.1 Running Example

The Eclipse Modeling Framework (EMF)<sup>1</sup> contains in its arsenal the meta-modeling language *Ecore*, which in itself is nothing more than a subset of the *UML class diagram* for modeling the structure of a domain in the form of packages, classes, attributes and relationships. EMF brings also a default implementation of the graphical concrete syntax — an editor — for *Ecore*, which is realized with GMF<sup>2</sup>. We are going to reuse that editor to model our running example, because the class diagram is the most used modeling diagram out there [8] and most users are familiar with its graphical concrete syntax.

Please take a look at the model of our running example depicted in Figure 3.1. It is a model of a *Simple Blog* application. The model is of a very naive design and incomplete — this is on purpose, as you will see later why — but sufficiently simple for us to show you few interesting cases why we would like to extend the domain language with supplementary information through the simple annotation mechanism rather than extending the language itself (its meta-model) and adapting its tools to reflect introduced changes to the language.

First, let us examine the contents of the running example model. We see two packages containing classes. The *data* package contains our data model, i.e., *User*, *Adress*, *Post* and

---

<sup>1</sup>The EMF project is a modeling framework and code generation facility for building tools and other applications based on a structured data model. <http://www.eclipse.org/modeling/emf>

<sup>2</sup>Graphical Modeling Framework. <http://www.eclipse.org/modeling/gmp>

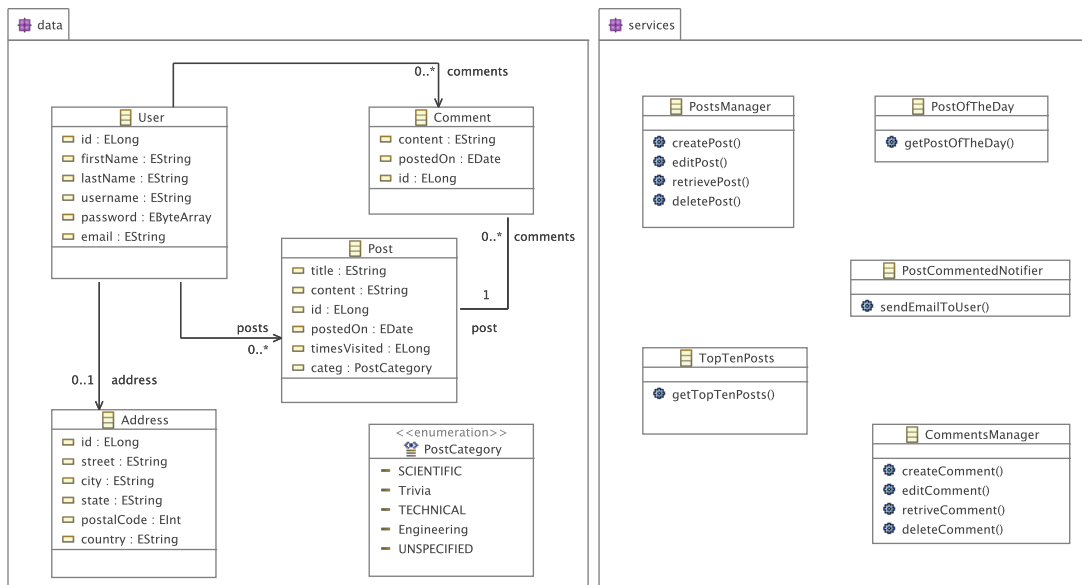


Figure 3.1: Simple Blog application – the model of our running example.

Comment classes — modeled as simple POJOs<sup>3</sup> — hold within their properties business data which is relevant for the Simple Blog application. The `services` package contains classes which implement the business logic of the application, such as creation, manipulation and retrieval of the business data. `PostsManager` and `CommentsManager` hold the business logic for creation and manipulation of posts and comments by Simple Blog users. `PostOfTheDay` class is responsible for retrieval of a *post of the day* previously specified through some criterion or by an administrative user. If the creator of a post wants to be notified, say, by email, when some other user has left a comment on the post, he can be notified by the service provided in the `PostCommentedNotifier` class when such an event occurs. And the `TopTenPosts` class will provide an implementation for retrieval of the ten most visited posts.

After explaining the contents of the model, let us introduce two cases when and why we would like to extend such a modeling language.

**Case One: Annotating model elements with Java EE specific information.** The Java platform Enterprise Edition (Java EE) extends the standard edition of the Java programming language with API and runtime environment for developing and running enterprise applications. These applications have to satisfy the needs of larger user groups, such as organizations, rather than a single user. To properly develop a software that will be used by many users, the software has to satisfy different characteristics — reliability, scalability, accessibility, security, etc., just to name the few — common to all enterprise applications. Rather than reinventing the wheel each and every time, developers can concentrate on the business data and logic of a developed application and at a later time introduce, say, security or reliability to the application by the means of

<sup>3</sup>Plain Old Java Object. [http://en.wikipedia.org/wiki/Plain\\_Old\\_Java\\_Object](http://en.wikipedia.org/wiki/Plain_Old_Java_Object)

configuration. For more detailed information about the Java EE platform we recommend reading the Java EE technical documentation<sup>4</sup> where you can find introduction to the Java EE platform, tutorials, and API documentation.

Among all Java EE specifications we are going to concentrate only on Enterprise JavaBeans (EJB) — a server-side component architecture for specifying the components that encapsulate the business logic of an application — and on the Java Persistence API (JPA) for specifying which classes hold the business data, also known as *Entities*. The properties of an Entity class are mapped by the underlying framework to a relational database schema. This process is also known as Object/Relational Mapping (ORM). EJB components can be differentiated into: *Session Beans* and *Message-Driven Beans*. Session Beans can in turn be differentiated between three types:

- **Stateful** – Track the state of a client through a session. One instance of the business object per client.
- **Stateless** – Do not track the state of a client. The client’s requests can be handled by different instances of the business object.
- **Singleton** – For all clients there is only one instance of the business object.

Message-Driven Beans just allow the communication between a client and a business object to be message-driven rather than through method calls as it is the case with session beans.

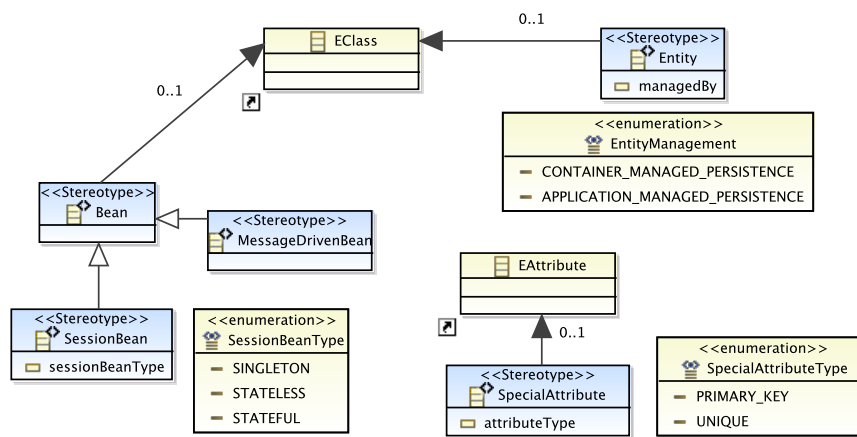


Figure 3.2: EJB\_Profile.

Now, we are going to extend the Ecore language so that we can introduce these new concepts into the language. For this purpose we have created the *EJB\_Profile*, as depicted in Figure 3.2,

<sup>4</sup>Java Platform, Enterprise Edition (Java EE) Technical Documentation. <http://docs.oracle.com/javasee/>

containing stereotype specifications for aforementioned concepts, and also an additional stereotype specification for *special attributes* such as primary keys and attributes having unique values. Applying these stereotypes to the running example model would produce the graphical concrete syntax as depicted in Figure 3.3.

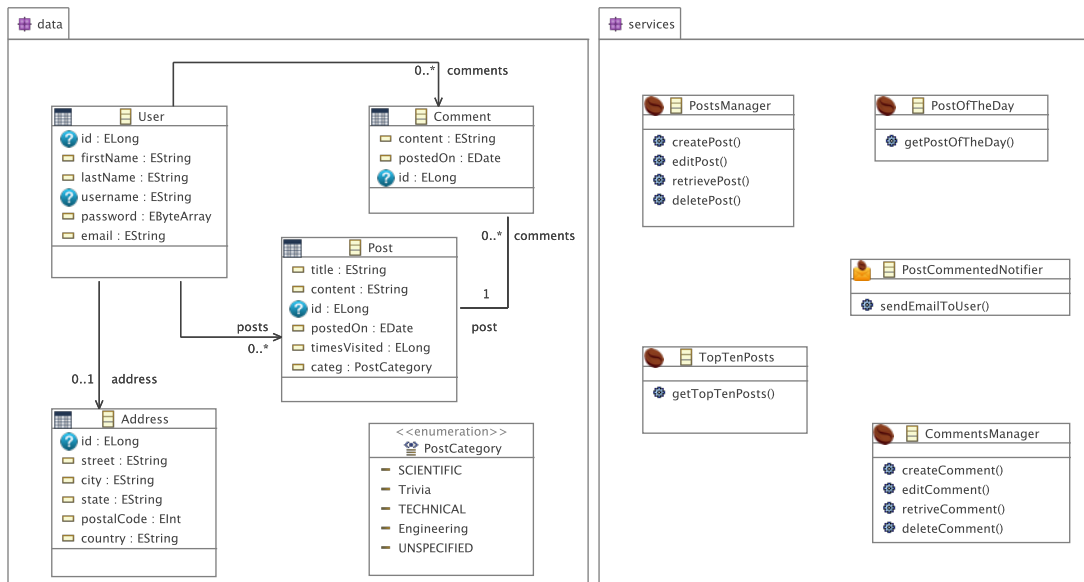






Figure 3.3: Running example model annotated with stereotypes from EJB\_Profile.

A stereotype applied to a model element is visualized by an icon decoration, which is still the case in the current implementation of EMF Profiles. For example, the icon  on User, Address, Post and Comment denotes that these are entities. Some attributes of the entities are decorated with the icon  meaning that they are special attributes. PostsManager, CommentsManager, TopTenPosts and PostOfTheDay are Session Beans  and PostCommentedNotifier is a Message-Driven Bean .

But, in order to know if the persistence of an entity is managed by a *container* — an application server, also known in JEE as the container — or by the application itself is not clear from the look at the extended graphical concrete syntax. What kind of specialization is meant for the annotated attributes? Are they meant to represent primary keys in a relational database schema or should a database system only insure uniqueness amongst the values of an attribute? It is unclear. We also do not know, for example, if our session beans are stateless, stateful or singleton — it all depends on the type assigned in the tagged value of a stereotype application.

In this concrete setting, the extension of the graphical concrete syntax, with icon decorations, does not help much to quickly grasp what model elements belong to the one or the other specific specialization. For small models, such as our running example, we might be able to comprehend it by looking at the structure and naming of model elements, but if we consider models with more than few dozen elements, this becomes an issue. In order to know it for sure what kind of

specialization a model element has we would have to look at the tagged values of a stereotype application. For larger models this would be very cumbersome and time consuming, because to list the tagging values we would have to select the stereotype application in its Eclipse IDE View and look for them in the Eclipse IDE Properties View.

The Figure 3.4 depicts the extension of the graphical concrete syntax with advanced decorations that we would like to have. With first glimpses at the model diagram we can much quicker grasp and comprehend the special information introduced to the model elements through stereotype applications. For example, looking at the `User` element we know it to be an Entity class and now we also see that its persistence will be managed by the JEE container (*CMP* in the upper left corner of the class visualization stands for Container-Managed-Persistence). Also the background color helps in quickly identifying other classes with the same specialization information. It is also clearer that all `id` attributes of entity classes are primary keys and that the `User.username` is a unique attribute. In *services* package, we know now much clearer which session beans are stateless, stateful or singleton.

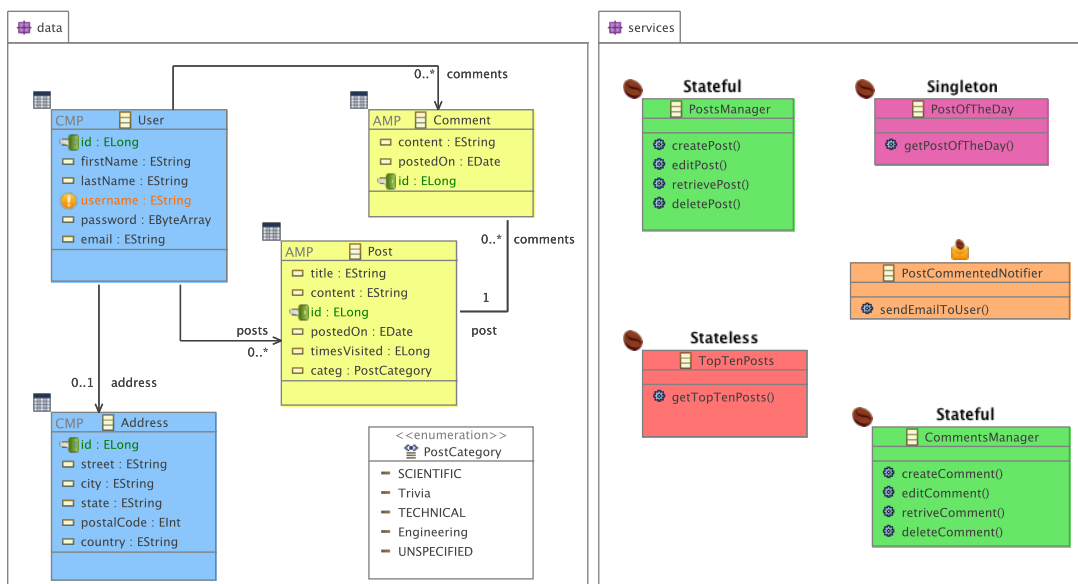


Figure 3.4: Running example model annotated with EJB\_Profile showing advanced decorations.

One way to accomplish advanced decoration, as emphasized in this thesis, is through extension mechanism provided by the EMF Profiles. But, besides our approach there are also other possibilities to achieve a similar goal, such as:

- A naive approach, where we just add additional attributes to the domain model elements to hold the technology specific information needed to generate the platform specific code later on. This is a very bad approach, because it pollutes the model with the technology specific information. It also does not really provide a more comprehensible look at the model, rather the opposite. What happens if we want to use the same model to generate

software artifacts for completely different platform and technology? Adding even more attributes to the model? This might be the worst approach you can choose to add platform specific information to the model.

With profiles you could just swap the profile definition tailored for another specific technology, say, .NET platform. This way the model stays generic and additional information can be attached as needed.

- Another approach would be adapting the Ecore language by introducing new concepts in the meta-model of the language. Doing this, we would have adapted the class diagram modeling language with new concepts and constructs to express the technology specific information, e.g., the EJB specific data for the Java EE platform. For that we need to adapt the default Ecore editor implementation in order to be able to create and visualize these new concepts in the editor. Well, doing it once may be not a big problem. But, what happens if you at a later point in time decide you want to use your modeling language for some other specific platform. Suddenly, you are faced with even more adaptations — not only to the meta-model and the editor but all dependent tooling.

**Case two: Annotating model elements with model review information.** We have already talked about the design phase in Chapter 1. We said that it is an integral part of the software engineering process in which we design our application by modeling its components, their behavior and interaction, and also how the user interacts with the application and vice versa. So, the models we design will certainly undergo some revisions and refinements. It would be very convenient, for example, to be able to add review information directly to the model elements in the editor that we use to make our design. That way we could communicate the information to our fellow designers more precisely — considering it that the review information is attached directly to a visual representation of a model element.

But, we would like to keep that information in a separate file from our model file. Thus, making it possible to visualize the review information on the model simply by loading it from a resource file, do revisions, and hide them by unloading them from the model.

One solution can be found in an EMF profile that can be used with any domain specific language (DSL) — in our case created with Ecore and EMF — also known as the *meta profile* [39].

The Figure 3.5 shows the *Model Review* meta profile. The `ReviewDecision` stereotype extends the `ENamedElement` from the Ecore meta-model and that makes it possible to apply review decision stereotypes to model elements that are subtypes of it. Many model elements are, e.g., packages, classes, attributes, operations, references, enumerations, etc. The `ReviewDecision` is an abstract stereotype definition having two tagging values (`reviewer` and `reviewDate`) common to all review decisions and from there we have three concrete stereotype specializations: `Approved`, `Rework` and `Declined`; where some of them contain additional tagging values.

Applying Model Review profile to our running example model with the current implementation of EMF Profiles would produce similar graphical concrete syntax as in Figure 3.3 extending the syntax only by icon decorations. An example of the graphical concrete syntax extension we would like to have can be seen in Figure 3.6.

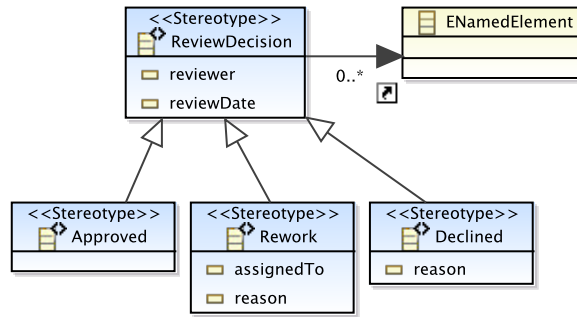


Figure 3.5: Model Review meta profile.

The icon and the solid line indicate *Approved* review decision, attached to the model element, whereas the icon and the dashed line indicate *Declined* review decision, and the icon and the dotted line indicate *Rework* review decision.

Another useful feature would be to see the detailed information of the review decision attached to a model element in the editor. For example, placing the mouse pointer over the icon decoration indicating applied stereotype would show a tooltip with the information containing text combined with actual tagging values of the stereotype application. The Figure 3.7 exemplifies such a feature.

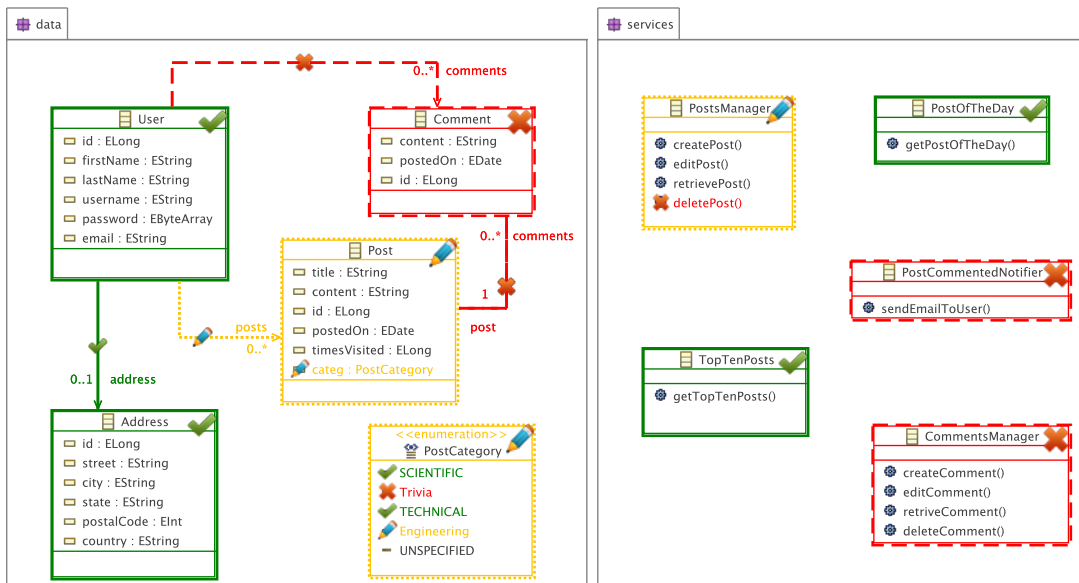


Figure 3.6: Running example model annotated with Model Review profile showing advanced decorations.

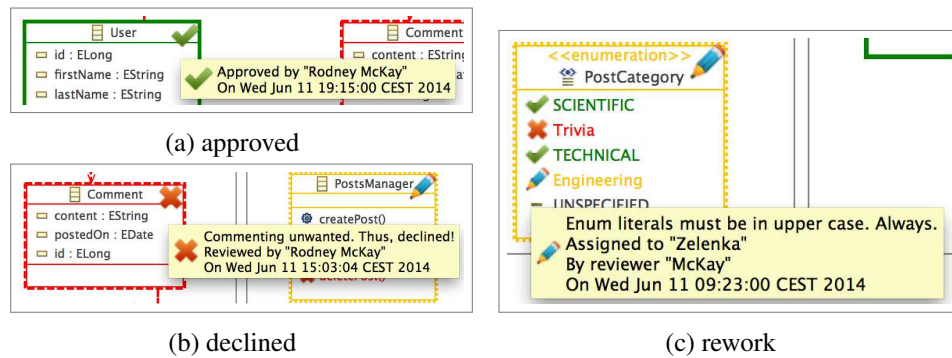


Figure 3.7: Model review decision tooltip examples.

## 3.2 Requirements

We devised following requirements:

1. It must be possible to specify decorations that only change visual properties of the model element visual shape to which a stereotype is applied. Properties such as background and foreground colors. And if the visual shape is bordered or constructed as a line (such as connections between elements), then changing its properties, such as color, width and style, must be possible.
2. Specification of decorations must be possible to add additional visual adornments to a model element visualization such as images (icons) and boxes (text-boxes).
3. Specification of arbitrary number of image and box decorations for the same stereotype must be possible, as well as their arbitrary placement relative to the visual shape of the model element they decorate.
4. For additional adornments, such as, image and box decorations, it must be possible to create tooltips that are shown when the user hovers with the mouse pointer over such a decoration. Tooltip is constructed from a simple text combined with tagging values of the stereotype application for which the decoration is specified. The actual content of a tooltip is dynamically created and updated at runtime.
5. It must be possible to specify *activation rules* for all decoration descriptions of a stereotype, or any single decoration description to provide us with control mechanism to make decisions when and which decoration should be visualized or not. Activation rules are expressions, which when evaluated return boolean values `true` or `false`. They are constructed from tagging values of simple data types, such as `String`, `Number`, `Boolean`, and `Enumeration`, compared against concrete specific values. It should also be possible to specify activation rules using OCL<sup>5</sup> invariant expressions in the context of the applied stereotype.

<sup>5</sup>Object Constraint Language. <http://www.omg.org/spec/OCL/>



6. As we want to evaluate our solution in a *model execution* case study, every change/update of a tagged value at runtime should trigger the reevaluation of the activation rules of that stereotype application and refresh its visual decorations in the editor where they are visualized.
7. The graphical concrete syntax extension must be transient. Which means that any change to the graphical concrete syntax of a modeling language at runtime will not be persisted when we save the model diagram to a file.
8. The graphical concrete syntax extension solution must be independent from the underlying technology in which the original graphical concrete syntax of a modeling language was created, e.g., GMF, Graphiti.
9. The specification of the graphical concrete syntax extension, i.e., decoration description specifications, must be kept separately from the profile definition and the extended modeling language. That means the decoration descriptions must be persisted in a separate file.
10. A domain specific language must be devised in order to simplify the process of creation and description of graphical decorations for applied stereotypes. For this language an editor must be constructed which supports the syntax of the language and provides features such as syntax highlighting and content assist.



# Decoration Description Language

One of our requirements, as specified in Chapter 3, is about devising a domain specific language with the goal to simplify the process of creation and description of graphical decorations for applied stereotypes. In this chapter we give the most relevant informations on how we approached this task and also describe few components that were created in the process of specifying the *decoration description language* and building the tools to support it.

## 4.1 Decoration Description Language Engineering

In order to accommodate our requirements on decoration description specifications from Chapter 3 we have decided to build a *textual modeling language*. This means we would create a modeling language that has the textual concrete syntax (TCS) — for more information about modeling language engineering and concrete syntax development, please take a look at Chapter 2. To build our language we have decided to employ the Xtext<sup>1</sup> framework. To specify the metamodel and TCS of our language we have decided to follow the *grammar first* approach — Xtext supports language designers with both *grammar first* as well as *metamodel first* approach.

The grammar specification in Xtext is similar to that of EBNF<sup>2</sup> but with additional features to achieve more expressiveness with respect to metamodeling language Ecore [9]. Out from the grammar specification Xtext generates the metamodel for the language, a model-to-text serializer, a text-to-model parser, and a text-based editor that supports features such as syntax highlighting, content assist for code completion, jump to declaration, and reverse-reference lookup across multiple files [17, 18]. The generated tooling can be further customized, for example to adjust linking of elements, code formatting, syntax coloring, and static error checking enhanced by custom validation checks and quick fixes.

Xtext-grammars are comprised of so-called *text production rules* and in particular three types of production rules are distinguished [9]:

---

<sup>1</sup>Xtext Eclipse project. <http://www.eclipse.org/Xtext>

<sup>2</sup>Extended Backus–Naur Form (cf. [43])

- **Type rules** in the grammar represent classes in the metamodel, which can be seen as their counterparts. They are used for defining modeling concepts. Hence, in the generated metamodel the contained classes will be produced from the type rules. The names of the generated classes in the metamodel correspond to the names of the type rules in the grammar. Type rules are constructed of *terminals* and *non-terminals*. Terminals represent the keywords, scope borders, and separation characters of the language syntax. Non-terminals are differentiated between *assignments* and *cross-references*. Assignments are mapped to attributes or containment references of the corresponding class in the metamodel, and cross-references are mapped to non-containment references. In other words, non-terminals of a type rule represent the features of the corresponding metamodel class. To define multiplicities of assignments in Xtext, the grammar syntax specifies several different assignment operators for setting the multiplicities of features in the metamodel.
- **Terminal rules** simply return a specific value, i.e., a sequence of characters — similar to EBNF terminal rules except that Xtext terminal rules may have an assigned return type such as `EString` or `EBoolean`.
- **Enum rules** are used, as the name suggests, for defining value enumerations. Consequently, when the metamodel is generated the corresponding `EEnums` are generated out of them.

Let us now describe how we specified the grammar for the decoration description language.

## 4.2 Grammar Specification

The complete grammar specification of the decoration description language can be found in Appendix A, which contains the listing of the complete grammar (cf. Listing A.1) and the railroad visualization of the grammar syntax in Section A.2. The diagram of the generated metamodel is depicted in Figure 4.1.

We are now going to tackle few type rules of the language grammar, which we find most relevant to comprehend how the language is designed and meant to be used. Afterwards, we also mention few customizations on the generated editor that we have undertaken to achieve more user-friendliness.

The first 5 lines of the grammar code are instructions to Xtext framework, which specify the name of our language, and inclusion of other languages and their element types, such as the Ecore metamodel and the EMF Profiles metamodel (cf. Listing 4.1).

Listing 4.1: Specifying language name and importing metamodels of supported languages.

```

1 grammar org.modelversioning.emfprofile.decoration.
   EMFProfileDecorationLanguage with org.eclipse.xtext.common.Terminals
2
3 generate decorationLanguage "http://www.modelversioning.org/emfprofile/
   decoration/EMFProfileDecorationLanguage"
4 import "http://www.eclipse.org/emf/2002/Ecore" as ecore
5 import "http://www.modelversioning.org/emfprofile/1.1" as profile

```



The *DecorationModel* type rule in Listing 4.2 represents the root element of our language, containing different expressions that comprise the rule. The line 8 specifies the import resource statement for the *profile definition model*, which is mandatory because we must specify where the specific profile definition model is located. If our *decoration description model* is at the same location as the profile definition model then simply specifying the resource name will suffice. The line 9 specifies the optional *namespace* declaration. If we consider that the profile definition model contains one root element, say `EJB_Profile` (it would normally be an instance of an `EPackage`), which in turn contains stereotype definitions, such as `Entity` and `SessionBean`, specifying the namespace in the decoration description model would mean that we do not need to specify the complete *qualified name* to reference a stereotype but simply stating the name of the stereotype would suffice. Otherwise, not specifying the namespace, e.g., in the following form `profile EJB_Profile`, would mean that we would need to provide the complete qualified name when referencing a stereotype, e.g., `EJB_Profile.SessionBean` instead of simply `SessionBean`. After that, an arbitrary number of decoration descriptions is possible — actually we created the restriction that only one decoration description per stereotype instance is allowed, as shown in Listing 4.13.

Listing 4.2: The root element of our language.

```

7 DecorationModel:
8   "import resource" importURI = STRING
9   (namespace=Namespace)?
10  decorationDescriptions+=DecorationDescription*
11 ;

```

The *DecorationDescription* type rule in Listing 4.3 represents the declaration of decoration descriptions for a stereotype and it begins with the keyword `decoration` which is followed by the name of a stereotype — content assist can be activated to get stereotype names for which no decoration description declaration exists. After the stereotype name is given, the opening of scope border is specified with the left curly bracket (end of line 18), and closed with right curly bracket (line 23). Enclosed within scope border definition, the statement at line 20 specifies that at least one concrete decoration description is expected, and the statement at line 21 specifies an optional *activation specification* — activation specifications are explained later on — that will be evaluated at runtime in order to infer if all decorations of the stereotype should be activated or not.

Listing 4.3: DecorationDescription type rule.

```

17 DecorationDescription:
18   'decoration' stereotype=[profile::Stereotype|QualifiedName] '{'
19   (
20     decorations+=(AbstractDecoration)+
21     & (activation=Activation)?
22   )
23   '}'
24 ;

```

We defined several concrete decoration description type rules, which are explained in the following.

## Concrete Decoration Description Type Rules

Let us go straight to the most complex type rule, and introduce the *box decoration*, because it combines many features other decoration descriptions have too.

### Box Decoration

The *BoxDecoration* type rule, found in Listing 4.4 begins with the simple `box` keyword and encompasses all its features between curly brackets — to note, every decoration description specification has its scope borders denoted with curly brackets. There are three mandatory features of the box decoration, namely the *width* (line 53), *height* (line 54), and *text* (line 52). Width and height are specified as integer numbers denoting the size of the box decoration in pixels. The text is specified by the user, and can be constructed of simple user text (String) combined with tagged values of the corresponding stereotype. Text fragments are combined with + sign between them. The text is mandatory because the box decoration resembles a *text box*, but because it can contain more than just a text, it is simply called a box.

Listing 4.4: BoxDecoration type rule.

```
47 BoxDecoration :
48   {BoxDecoration}
49   'box' '{'
50     (
51       // mandatory
52       ('text' '=' text=Text)
53       & ('width' '=' width=INT)
54       & ('height' '=' height=INT)
55
56       // optional
57       & (image=BoxImage)?
58       & ('border' '{' border=Border '}')?
59       & ('foregroundColor' '=' foregroundColor=Color)?
60       & ('backgroundColor' '=' backgroundColor=Color)?
61       & (direction=Direction)? // default value for node = NORTH_WEST; for
62         edge = CENTER
63       & (margin=Margin)? // default value -1 on nodes, 50 for connection (
64         for connection margin is interpreted as percentage)
65       & ('contentDirection' '=' contentDirection=Directions)? // default
66         is CENTER
67       & ('tooltip' '=' tooltip=Text)?
68       & (activation=Activation)?
69     )
70   '}'
71 ;
```

An optional image is possible (line 57). The special type rule for *BoxImage* is found in Listing 4.5. The user must provide a valid URI<sup>3</sup> to the image resource, and specify the placement of the image relative to the text. Specification of the image placement inside the box is optional.

---

<sup>3</sup>Uniform Resource Identifier.

The default value is WEST, which means the image is placed on the left side relative to the text. Other possible values for placement are: NORTH, EAST, and SOUTH.

Listing 4.5: BoxImage type rule.

```
128 BoxImage :
129     {BoxImage}
130     'image' '{'
131     (
132         ('uri' '=' location_uri=STRING)
133         & ('placement' '=' placement = BoxImageOrientation)? // default
134         value is WEST
135     )
136     '}'
137 ;
```

The box decoration can have an optional *border* (line 58), based on the *Border* type rule. An explanation on how a border visualization can be specified is given further in text, in the description of the *BorderDecoration* type rule.

The text inside the box decoration is per default in black color, but we can change the color of the text by setting the *foreground-color* (line 59). Different ways on how to specify a concrete color are described further in text, in the description of the *ColorDecoration*.

The box decoration is normally placed over the shape that is decorated. The *background* of the box decoration is normally transparent. However, we can make its background opaque and color it by setting the *background-color* (line 60).

The *direction* (line 61) of the box is the direction relative to the decorated shape where the box decoration is going to be placed. If the decorated shape is a node shape, then the default direction is NORTH\_WEST. However, if the decorated shape is an edge shape, the box decoration is per default centered over the shape, and setting any different value would not change the position. Decorations on node and edge shapes behave somewhat differently, as we will see when specifying the margin value.

The *margin* (line 62) is the space, in himetric<sup>4</sup> units, between the edge of a decorated shape and the decoration. A positive margin places the figure outside the shape in the north-west direction from the top-left corner of the shape, and a negative margin places the decoration inside the shape in the south-east direction, also from the top-left corner of the shape. However, this only applies if the decorated shape is a node shape. For an edge shape the margin is interpreted as the percentage of the distance from the starting-point to the end-point of the edge. For example, the margin value of 0 places the decoration at the starting point of the edge; the value of 50 at the middle distance, between start and end point; and the value of 100 at the end-point of the edge.

The *content-direction* (line 63) is the specification of the content placement inside the boundaries of the box decoration — just to clarify, text and image represent the content of the box decoration. Per default the content is centered. All possible values for content direction are: CENTER, NORTH, SOUTH, WEST, EAST, NORTH\_EAST, NORTH\_WEST, SOUTH\_EAST, and SOUTH\_WEST.

---

<sup>4</sup>Himetric is a metric measurement of length, used in computing, which is independent of the display or screen resolution.



The *tooltip* (line 64) for the box decoration can be specified, which is constructed from simple text and tagged values of the corresponding stereotype application. Indeed, the specification of the tooltip is the same as the mandatory text specification (line 52). The only difference is that the tooltip is visualized only when the user hovers with the mouse over the box decoration.

Finally, the last feature of the box decoration that can be set is the *activation* specification (line 65). Each individual decoration of a stereotype can have its own activation specification, which is evaluated at runtime. Normally, the activation specification of the *DecorationDescription* rule supersedes the individual ones only if its evaluation is negative, otherwise the individual activations are evaluated.

## Image Decoration

The *ImageDecoration* type rule, found in Listing 4.6, is nothing more than a stripped-down box decoration. It has the mandatory specification of the location of an image resource, and optional *direction* and *margin* specification for the placement of the image decoration relative to the decorated shape. Also, a tooltip and an activation specification is possible. The same rules described for box decoration apply also here, thus, no need to repeat what was already said.

Listing 4.6: ImageDecoration type rule.

```
34 ImageDecoration:
35   {ImageDecoration}
36   'image' '{'
37     (
38       ('uri' '=' location_uri=STRING)
39       & (direction=Direction)? // default value for the node = NORTH_WEST,
40         and for edge = CENTER
41       & (margin=Margin)? // default value -1 on nodes, 50 for connections
42         (for connections margin is interpreted as a percentage)
43       & ('tooltip' '=' tooltip=Text)?
44       & (activation=Activation)?
45     )
46   '}'
47 ;
```

## Color Decoration

The *ColorDecoration* type rule, found in Listing 4.7, is the decoration description that tries to change visual color properties of the decorated shape. The *background* color (line 85) and the *foreground* color (line 86) of the decorated shape can be influenced.

The color can be specified based on type rules in Listing 4.8. The user can either specify a constant color or provide a concrete color specification. Constant colors are an enumeration of values such as ORANGE, YELLOW, and CYAN (cf. enum *Colors* specification in Listing A.1, lines 240 – 244). The concrete color can be specified as an *RGB* or *HexColor*. The RGB specification has the syntax such as RGB(255,255,255), which translates to white color. The HexColor uses different syntax to specify red, green, and blue color components, expressed

as the hexadecimal numbers, preceded by the # character. For example, #00FF00 translates to green color.

Listing 4.7: ColorDecoration type rule.

```
81 ColorDecoration:
82   {ColorDecoration}
83   'color' '{'
84     (
85       ('background' '=' background=Color)?
86       & ('foreground' '=' foreground=Color)?
87       & (activation=Activation)?
88     )
89   '}'
90 ;
```

Listing 4.8: Color specification type rules.

```
154 Color:
155   {Color}
156   value=ColorConstant | concrete = ConcreteColor
157 ;
158
159 ConcreteColor:
160   RGB | HexColor
161 ;
162
163 RGB:
164   'RGB' '(' red=INT ',' green=INT ',' blue=INT ')'
165 ;
166
167 HexColor:
168   hexCode = HEX_COLOR
169 ;
170
171 ColorConstant:
172   value=Colors
173 ;
```

## Border Decoration

The *BorderDecoration* type rule, found in Listing 4.9, besides the optional activation specification, must have the *border* specification, described by the *Border* type rule in Listing 4.10. Border decoration influences direct visual properties of the decorated shape, which means that it only can have an effect on node shapes — or shapes that have a border.

The *size* of the border can be specified as a positive integer number, which corresponds to the border line thickness, expressed in pixel units. Also, the *color* and the *style* of the border line can be specified. Possible values for the line style are: SOLID, DOTS, DASH, DASHDOT, and DASHDOTDOT. The default value for the size is 1; for the style is SOLID; and for the color is BLACK.

Listing 4.9: BorderDecoration type rule.

```
70 BorderDecoration:
71   {BorderDecoration}
72   'border' '{'
73     (
74       (border=Border)
75       & (activation=Activation)?
76     )
77   '}'
78 ;
```

Listing 4.10: Border type rule.

```
120 Border:
121   {Border}
122   ( (size=Size)? // default value is 1
123     & ('color' '=' color=Color)? // default value is BLACK
124     & (style=Style)? // default value is SOLID
125   )
126 ;
```

## Connection Decoration

The *ConnectionDecoration* type rule, found in Listing 4.11, as its name suggests, has an effect only on edge shapes. In the same way the border line of a node shape can be influenced, so can the edge shape line be influenced by specifying its *size* (line 96), *style* (line 97), and *color* (line 98). The same default values from the border decoration apply to the connection decoration.

Listing 4.11: ConnectionDecoration type rule.

```
92 ConnectionDecoration:
93   {ConnectionDecoration}
94   'connection' '{'
95     (
96       (size=Size)?
97       & (style=Style)?
98       & ('color' '=' color=Color)?
99       & (activation=Activation)?
100     )
101   '}'
102 ;
```

## Activation Specifications

Activation specification is the mean to govern over the set of decorations specified for a stereotype. With activation specifications we can control which decorations are visualized and which are not.

The Listing 4.12 shows the type rules that comprise the activation specification. The user specifies so-called *condition expressions* which are evaluated at runtime, and return boolean

values `true` or `false` — which stand for *activated* or *deactivated*. Conditions can also be logically combined into groups by the logical operators `ALL` and `ANY` (cf. *CompositeCondition* type rule, lines 192 – 193) — the *LogicalOperator* is an enum rule found in Listing A.1, lines 230 – 232. `ALL` means that all condition expression must evaluate to `true` in order to activate the corresponding decoration. Similarly, `ANY` operator requires only one expression that evaluates to `true`.

Listing 4.12: Activation specification type rules.

```

175 Activation:
176   'active when' condition=AbstractCondition
177 ;
178
179 AbstractCondition:
180   Condition | CompositeCondition | OclExpression
181 ;
182
183 OclExpression:
184   'ocl' '(' expression = STRING ')'
185 ;
186
187 Condition:
188   attribute=[ecore::EAttribute|QualifiedName] operator=ComparisonOperator
189   value=Type
190 ;
191 CompositeCondition:
192   operator=LogicalOperator '(' conditions += (AbstractCondition )+ ')'
193 ;

```

Condition expression can be specified as an *OclExpression* (lines 183 – 185) or as a concrete *Condition* (lines 187 – 189). OCL<sup>5</sup> expression is simply provided as a `String`, which will be evaluated at runtime. The current editor implementation does not provide syntax validation for OCL expressions. If during the evaluation, the OCL syntax is found wrong, the expression will be excluded from the evaluation. Therefore, the user must be sure that he has provided syntactically correct OCL expression.

The concrete *Condition*, on the other hand, supports the user by syntax validation based on the type of the tagged value that is part of the condition expression. At the moment, the current implementation supports only following tagged value types: integer and real numbers, `String`, `Boolean`, and `Enumeration` type. Also, these tagged values can only be compared against the concrete user specified values. If these values do not fit to the data type of the tagged value, then a syntax validation error is displayed in the editor.

The *ComparisonOperator* is an enum rule (cf. Listing A.1, lines 225 – 228). The comparison in a condition expression can be based on: equality `==`, inequality `!=`, greater than `>`, greater or equal than `>=`, lower than `<`, and lower or equal than `<=`. If the comparison operator makes no sense, based on the comparing data type, then a syntax validation error is issued. For example,

<sup>5</sup>Object Constraint Language. <http://www.omg.org/spec/OCL/>

all these comparison operators make sense when comparing numbers. However, comparing Strings, only *equal* or *unequal* operators make sense.

Important to note is, that condition expressions are reevaluated each time the change in the underlying data model occurs.

## Customizations

Just to exemplify few editor customizations that are possible through Xtext customization facilities, we describe some validation checks, specified to validate the concrete syntax of the decoration description language.

Xtext generates the so-called *Validation Provider* that can be augmented to introduce custom language syntax validation checks. Validation checks that can not be adequately specified in the grammar — e.g., some semantic validations — must be provided as additional custom validations. The Validation Provider is generated as an Xtend<sup>6</sup> class, and the validation checks are specified as the new methods of that class. Validation methods must be annotated with @Check annotation and implemented in the Xtend syntax.

The Listing 4.13 shows an implementation of the validation method, which purpose is to check that there is only one declaration of the decoration descriptions for the same stereotype. If there are more than one, then a validation error is produced, highlighting the error occurrence in the editor.

Listing 4.13: Validating that only one decoration description per stereotype instance exists.

```
1 @Check
2 def checkThatThereIsOnlyOneDecorationDescriptionForSameStereotype(
3     DecorationDescription decorationDescription) {
4     if ((decorationDescription.eContainer as DecorationModel).
5         decorationDescriptions.exists[
6             it != decorationDescription && it.stereotype == decorationDescription.
7             stereotype])
8         error("""Decoration description already defined for the «
9             decorationDescription.stereotype.name»""",
10            decorationDescription, DecorationLanguagePackage.Literals.
11                DECORATION_DESCRIPTION__STEREOTYPE)
12 }
```

Another example of a custom validation, as shown in Listing 4.14, insures that the user of the language provides valid values for *red*, *green*, and *blue* color components of the *RGB* model element (cf. type rule definition in Listing A.1, lines 163 – 165). The values must be in range between 0 and 255.

---

<sup>6</sup>Xtend, a flexible and expressive dialect of Java, which compiles into readable Java 5 compatible source code. <http://www.eclipse.org/xtend>

Listing 4.14: Validating that RGB color values are in range.

```
1 @Check
2 def checkRGB( RGB color) {
3     if (color.red < 0 || color.red > 255) {
4         error( '''Color values must be in range 0 - 255''' , color,
5               DecorationLanguagePackage.Literals.RGB__RED)
6     } else if (color.green < 0 || color.green > 255) {
7         error( '''Color values must be in range 0 - 255''' , color,
8               DecorationLanguagePackage.Literals.RGB__GREEN)
9     } else if (color.blue < 0 || color.blue > 255) {
10        error( '''Color values must be in range 0 - 255''' , color,
11              DecorationLanguagePackage.Literals.RGB__BLUE)
12    }
13 }
```

We have done more customizations on the generated editor than here indicated. Customizations such as adaptations to the so-called *Scope Provider* for specifying the linking of model elements, customizations to the content assist, and also additional validation checks. However, we will not go here into any detailed explanation of them. The reader may find more information on how to customize generated editor code on the Xtext documentation site<sup>7</sup>.

### 4.3 Summary

Let us note that these decorations will take effect on the visual representations of model elements only if it is possible to access the right visualization properties of the elements. In other words, if the visual representations are constructed in the way that deviates a lot from how the creators of graphical frameworks suggest it, then they might not influence the right visualization parts. This mostly considers decorations that try to change direct properties of the visual representation such as background and foreground color, border style and its thickness. Other decorations that add new adornments, such as box and image decorations, are not affected by this. In any case, if the decoration can not take the right effect on the visualization, it will fail gracefully.

Another important thing to note is that our decoration descriptions are independent of the underlying technologies that are used for the specification of the graphical concrete syntax and for building editors that support it. Let us here just say that decoration descriptions are interpreted by dedicated *graphical decorators*, constructed for each supported graphical editor technology. How this really works is explained in the architecture overview of the most important components in our solution (cf. Chapter 6).

---

<sup>7</sup>Xtext documentation site. <http://www.eclipse.org/Xtext/documentation.html>

# Decorating Graphical Modeling Languages at Runtime

In this chapter we describe how decoration descriptions are specified in the concrete syntax of the decoration description language, and how they are manifested in the graphical model editor at runtime. We base this description on our running example from Chapter 3.

## 5.1 Eclipse IDE Tooling Environment around EMF Profiles

Let us first describe the Eclipse IDE tooling environment centered around using EMF Profiles. The documentation on how to install EMF Profiles can be found on the project's website <https://code.google.com/a/eclipselabs.org/p/emf-profiles>.

After following the instructions on how to create a new EMF profile project, e.g., to create an *EJB\_Profile*, the Eclipse presents us with the new project in the Eclipse Project Explorer, and with the project structure as illustrated in Figure 5.1a.

The file *profile.emfprofile\_diagram* when opened presents us with the diagramming editor to model our profile — as we already said, we use our running example, so the EJB\_Profile diagram looks like in Figure 3.2.

The file *profileapplication.decoration* contains our decoration descriptions for the stereotypes of the EJB\_Profile. When opened it presents us with the dedicated textual editor understanding the textual concrete syntax of the decoration description language, as depicted in Figure 5.1b. The file is not created empty. It already contains few code fragments such as mandatory *import resource* statement pointing to the profile definition model file, and an optional profile *namespace* specification — in our case importing the root element of the profile definition model, which is the package `EJB_Profile`, so that its stereotypes can be directly referenced in the decoration description declarations without the need to prefix the stereotype name with the package name.

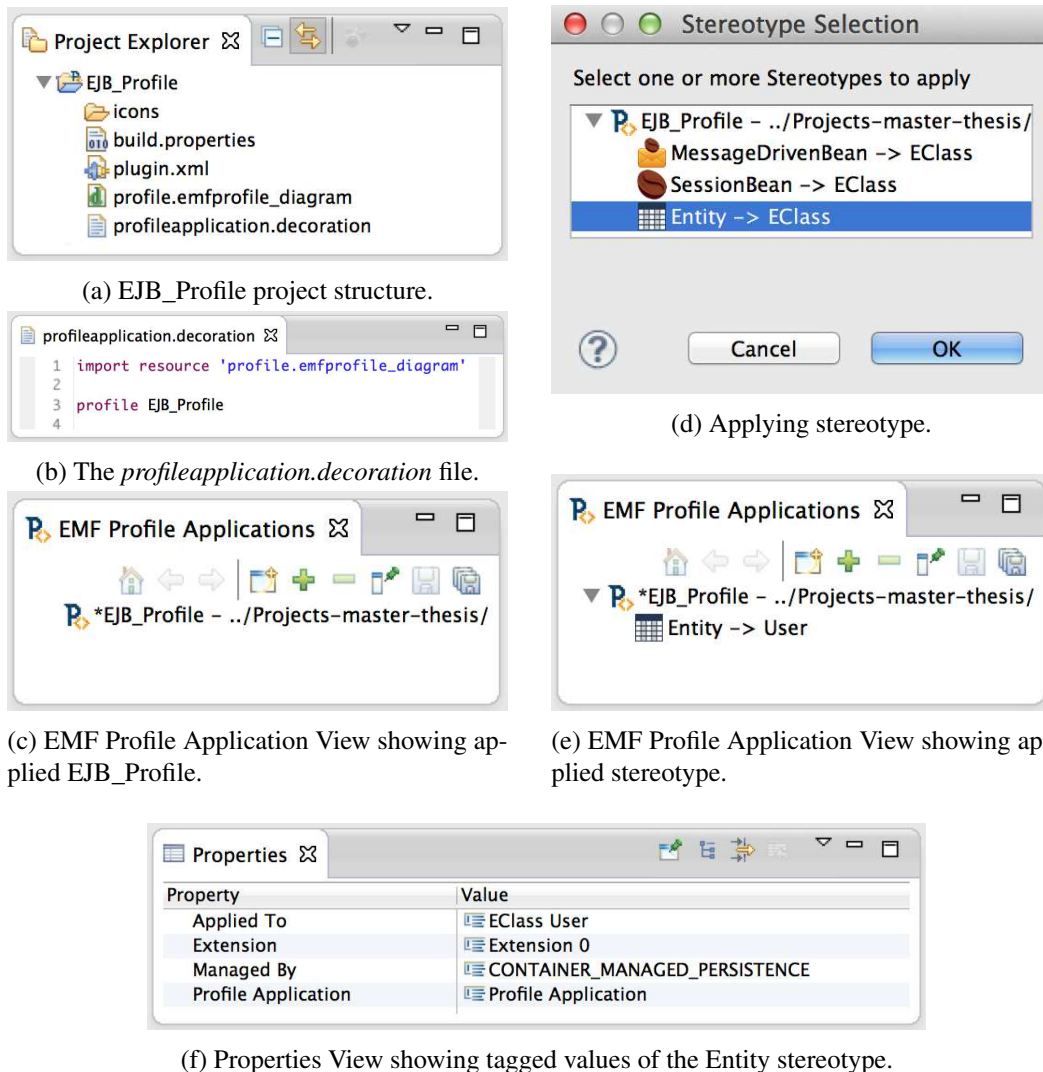


Figure 5.1: Eclipse IDE tooling environment around EMF Profiles.

When we apply the EJB\_Profile to our *Simple Blog* running example (cf. Figure 3.1), the profile application can be visualized in the dedicated *EMF Profile Application View* as depicted in Figure 5.1c.

Now we are ready to apply the stereotypes from the EJB\_Profile to the model elements of the Simple Blog. For example, we select the `User` element in the diagram and in the context menu we click at *Apply Stereotype*. This opens the pop-up window, as depicted in Figure 5.1d, with the purpose to select the desired stereotype and apply it to the model element. Selecting the `Entity` stereotype and clicking on `OK` applies the stereotype. The application of the stereotype is visible in the EMF Profile Application View, as depicted in Figure 5.1e

By selecting the applied stereotype in the EMF Profile Application View we can display its



tagged values in the Eclipse Properties View, as depicted in Figure 5.1f. The properties view allows the manipulation of tagged values.

These are the tools that the user gets to work with when extending a modeling language through the mechanism of EMF Profiles.

## 5.2 Decoration Description and its Manifestation

To demonstrate how decoration descriptions manifest themselves graphically in an editor when a stereotype is applied to a model element we now present the decoration description specifications, in the concrete syntax of our decoration description language, that we created for our running example for *EJB\_Profile* and *Model Review* profile. Furthermore, we give few concrete illustrations on how some of these decoration descriptions extend the graphical concrete syntax of a model element when the corresponding stereotype is applied to it.

### Decoration Descriptions of the EJB\_Profile

The complete decoration description specifications for the EJB\_Profile is found in Listing 5.1. Based on this specification the Figure 3.4 exemplifies how the graphical concrete syntax of the Simple Blog running example model is influenced when stereotypes are applied to it.

Listing 5.1: Decoration descriptions for stereotypes of the EJB\_Profile.

```
1 import resource 'profile.emfprofile_diagram'
2
3 profile EJB_Profile
4
5 decoration Entity {
6
7     // All Entity Beans
8     image {
9         uri = "platform:/resource/EJB_Profile/icons/db_table_16.png"
10        margin = 0
11    }
12
13    // BEGIN Container Managed Persistence
14    box {
15        active when managedBy == CONTAINER_MANAGED_PERSISTENCE
16        text = "CMP"
17        foregroundColor = GRAY
18        width = 26
19        height = 13
20        margin = -2
21        tooltip = "Container-Managed-Persistence"
22    }
23    color {
24        active when managedBy == CONTAINER_MANAGED_PERSISTENCE
25        background = #8AC6FB //RGB(245, 245, 245) // White Smoke 245-245-245
26    }
27    // END Container Managed Persistence
28
```

```

29 // BEGIN Application Managed Persistence
30 box {
31     active when managedBy == APPLICATION_MANAGED_PERSISTENCE
32     text = "AMP"
33     foregroundColor = GRAY
34     margin = -2
35     width = 26
36     height = 13
37     tooltip = "Application-Managed-Persistence"
38 }
39 color {
40     active when managedBy == APPLICATION_MANAGED_PERSISTENCE
41     background = #F5FF88 //RGB(250, 235, 215) // Antique White
42                 250-235-215
43 // END Application Managed Persistence
44 }
45
46 decoration SpecialAttribute {
47
48     // BEGIN Primary Key Decoration
49     image {
50         active when attributeType == PRIMARY_KEY
51         uri="platform:/resource/EJB_Profile/icons/key_16x16.png"
52     }
53     color {
54         active when attributeType == PRIMARY_KEY
55         foreground = GREEN_DARK
56     }
57 // END Primary Key Decoration
58
59 // BEGIN Unique Attribute Decoration
60 image {
61     active when attributeType == UNIQUE
62     uri="platform:/resource/EJB_Profile/icons/unique_16.png"
63 }
64 color {
65     active when attributeType == UNIQUE
66     foreground = #FF6C00
67 }
68 // END Unique Attribute Decoration
69 }
70
71
72 decoration SessionBean {
73
74     // All Session Beans
75     image {
76         uri = "platform:/resource/EJB_Profile/icons/coffee_bean_16.png"
77         margin = 0
78     }
79
80 // BEGIN Singleton Decoration

```

```

81  image {
82      active when sessionBeanType == SINGLETON
83      uri = "platform:/resource/EJB_Profile/icons/Singleton_16.png"
84      direction = NORTH
85      margin = 3
86  }
87  color {
88      active when sessionBeanType == SINGLETON
89      background = #E667AF
90  }
91  // END Singleton Decoration
92
93  // BEGIN Stateless Decoration
94  image {
95      active when sessionBeanType == STATELESS
96      uri = "platform:/resource/EJB_Profile/icons/Stateless_16.png"
97      direction = NORTH
98      margin = 3
99  }
100 color {
101     active when sessionBeanType == STATELESS
102     background = #FF7373
103 }
104 // END Stateless Decoration
105
106 // BEGIN Stateful Decoration
107 image {
108     active when sessionBeanType == STATEFUL
109     uri = "platform:/resource/EJB_Profile/icons/Stateful_16.png"
110     direction = NORTH
111     margin = 3
112 }
113 color {
114     active when sessionBeanType == STATEFUL
115     background = #67E667
116 }
117 // END Stateful Decoration
118
119 }
120
121 decoration MessageDrivenBean {
122     image {
123         uri = "platform:/resource/EJB_Profile/icons/mdb_16.png"
124         margin = 3
125         direction = NORTH
126         tooltip = "Message-Driven Bean"
127     }
128     color {
129         background = #FFB273
130     }
131 }

```

For example, if we take the `User` model class from the running example and apply the Entity stereotype to it — `CONTAINER_MANAGED_PERSISTENCE` is set to the `managedBy` tagged value — and apply `SpecialAttribute` stereotype to the `id` and `username` attributes — the `attributeType` tagged value for the first is set to `PRIMARY_KEY` and for the latter to `UNIQUE` — the graphical concrete syntax is extended as illustrated in Figure 5.2. The decoration descriptions responsible for the graphical changes of the `User` class are found in Listing 5.1 on lines 5 – 26, and for its `id` and `username` attributes on lines 46 – 69.

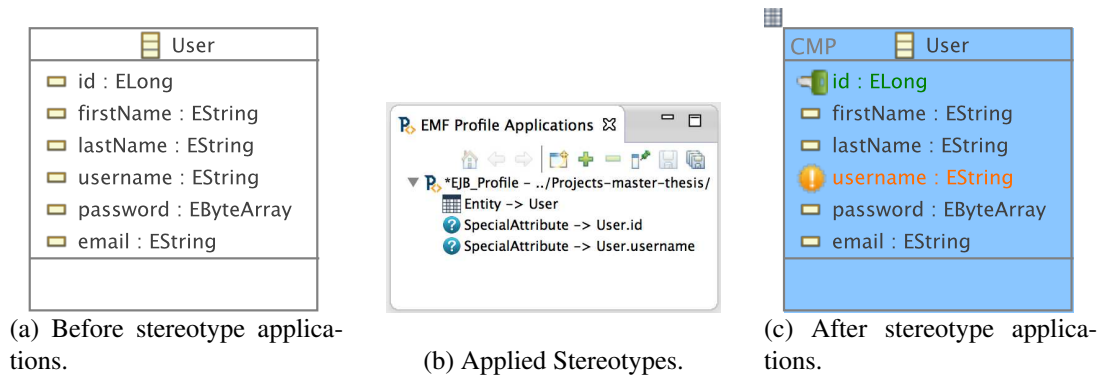


Figure 5.2: Stereotype applications to the `User` model class.

## Decoration Descriptions of the Model Review Profile

The complete decoration description specifications for the Model Review profile (cf. Figure 3.5) is found in Listing 5.2. Based on this specification the Figure 3.6 and Figure 3.7 exemplify how the graphical concrete syntax of the Simple Blog running example model is influenced when stereotypes are applied to it.

Listing 5.2: Decoration descriptions for stereotypes of the Model Review profile.

```

1 import resource 'profile.emfprofile_diagram'
2
3 profile ModelReview
4
5 decoration Approved {
6   image {
7     active when ocl ("let o:ecore::EObject = appliedTo in o.ocIsKindOf(
8       ecore::ETypedElement) or o.ocIsKindOf(ecore::EEnumLiteral)")
9     direction = WEST
10    uri="platform:/resource/ModelReview/icons/yes_16.png"
11    tooltip = "Approved by \" + reviewer + "\"\nOn \" + reviewDate
12  }
13  image {
14    active when ocl ("appliedTo.ocIsKindOf(ecore::EClassifier)")
15    direction = NORTH_EAST
16    uri = "platform:/resource/ModelReview/icons/yes.png"
17    tooltip = "Approved by \" + reviewer + "\"\nOn \" + reviewDate
18  }
19 }

```

```

18     border {
19         color = GREEN_DARK
20         size = 2
21     }
22     color {
23         foreground = GREEN_DARK
24     }
25     connection {
26         color=GREEN_DARK
27         size = 2
28     }
29 }
30
31 decoration Declined {
32     image {
33         active when ocl ("let o:ecore::EObject = appliedTo in o.ocIsKindOf(
34             ecore::ETypedElement) or o.ocIsKindOf(ecore::EEnumLiteral)")
35         direction = WEST
36         uri="platform:/resource/ModelReview/icons/no_16.png"
37         tooltip = reason + "\nReviewed by \"" + reviewer + "\"\nOn " +
38             reviewDate
39     }
40     image {
41         active when ocl ("appliedTo.ocIsKindOf(ecore::EClassifier)")
42         direction = NORTH_EAST
43         uri="platform:/resource/ModelReview/icons/no.png"
44         tooltip = reason + "\nReviewed by \"" + reviewer + "\"\nOn " +
45             reviewDate
46     }
47     border {
48         color=RED
49         lineStyle=DASH
50         size = 2
51     }
52     color {
53         foreground = RED
54     }
55     connection {
56         color=RED
57         lineStyle=DASH
58         size = 2
59     }
60 }
61
62 decoration Rework {
63     image {
64         active when ocl ("let o:ecore::EObject = appliedTo in o.ocIsKindOf(
65             ecore::ETypedElement) or o.ocIsKindOf(ecore::EEnumLiteral)")
66         direction = WEST
67         uri="platform:/resource/ModelReview/icons/wip_16.png"
68         tooltip = reason + "\nAssigned to \"" + assignedTo + "\"\nBy reviewer
69             \"" + reviewer + "\"\nOn " + reviewDate
70     }
71 }

```

```

66  image {
67      active when ocl ("appliedTo.oclIsKindOf(ecore::EClassifier)")
68      direction = NORTH_EAST
69      uri="platform:/resource/ModelReview/icons/wip.png"
70      tooltip = reason + "\nAssigned to \"" + assignedTo + "\"\nBy reviewer
71              \"" + reviewer + "\"\nOn " + reviewDate
72  }
73  border {
74      color=ORANGE
75      lineStyle=DOTS
76      size = 2
77  }
78  color {
79      foreground = ORANGE
80  }
81  connection {
82      color=ORANGE
83      lineStyle=DOTS
84      size = 2
85  }

```

If we take, for example, the `Comment` model element and apply `Declined` stereotype to it, we would change its appearance as illustrated in Figure 5.3.

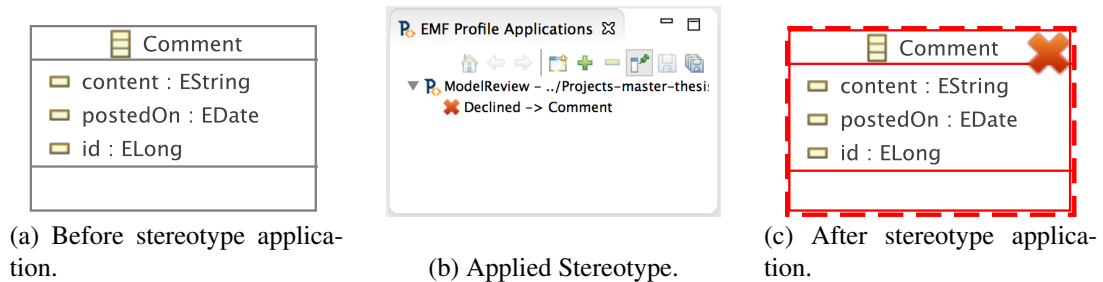


Figure 5.3: Stereotype application to the `Comment` model class.

The responsible decoration description for `Declined` stereotype is found in Listing 5.2, on lines 31 – 57. The first *image decoration* (lines 32 – 37) is activated only in the case a stereotype is applied to a model element which is a feature or enumeration literal — expressed as an OCL constraint. The second *image decoration* (lines 38 – 43) is activated only if a model element is of type `EClassifier` — also expressed as an OCL constraint. The reason for differentiation is that we wanted to display a smaller image on features such as attributes, references, and enumeration literals — which is the case with the first — and a bigger image on classes — which is the case with the latter.

For that reason, we provide another example of applying `Approved` stereotype to the reference address between `User` and `Address` model classes. The process of visual change is illustrated in Figure 5.4.

The decoration descriptions responsible for visual change are those bound to the declaration for `Approved` stereotype (cf. Listing 5.2, line 5). The *image decoration* (lines 6 – 11) places

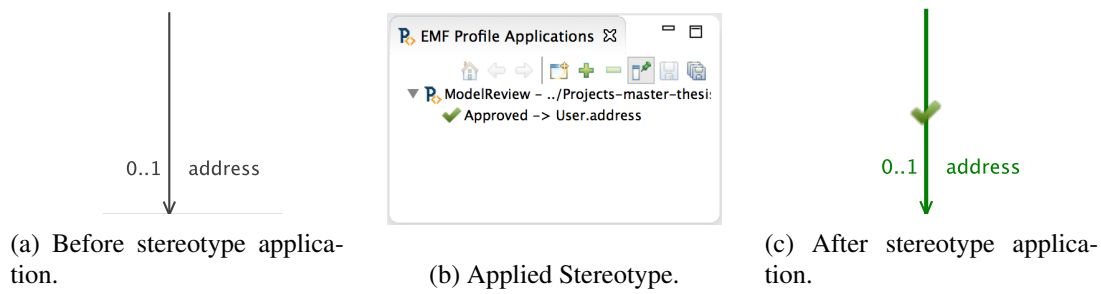


Figure 5.4: Stereotype application to the address reference between User and Address model classes.

the image on the connection line, and *connection decoration* (lines 25 – 28) specifies the change in line thickness and line coloring to the dark green color.

As already mentioned in Chapter 3, our running example is based on the default Ecore editor. Hence, the responsible component, for the interpretation and the manifestation of decoration descriptions, is the *GMF Decorator*. If it were the case that the graphical editor was based on Graphiti, then the *Graphiti Decorator* would have been the responsible component for the interpretation.

The overview of the most important components and the architecture of our solution can be found in Chapter 6.





# Architecture

The previous chapters have provided you with enough information about extending a modeling language. We have presented the new means on how to extend its graphical concrete syntax. By now you already know that this is done by specifying the decoration descriptions for stereotypes, which are used to annotate model elements. We also provided concrete examples of decoration descriptions and their manifestations.

However, what is still missing is the description of the technical solution. What kind of components are there, involved in applying stereotypes, and extending and adapting the graphical concrete syntax of the model elements in a diagram?

Well, this chapter gives exactly that, the description of the most important components that comprise the solution. In other words, the description of the architecture of the solution.

## 6.1 Architecture of the Solution

The overview of the architecture is illustrated in Figure 6.1, and is comprised of following components:

- **EMF Profile** is the component that contains the metamodel specification for profile definition and profile application, together with their generated Java interfaces and implementation classes. Of course, EMF generates other components such as the *Edit* component that contains helper classes for better integration of model data within the Eclipse tooling, and *Editor* component that contains the implementation of a simple tree-based editor. However, such components that are not relevant to better grasp the solution were excluded from the architecture overview.
- **Profile Registry** component exercises the role of a “register office” for profile definitions. The Eclipse projects can be tagged by a specific *project natures*, as it is the case, for example, with Java projects, or profile definition project. At runtime, Profile Registry searches for profile definition projects in Eclipse’s plug-ins directory and in the workspace

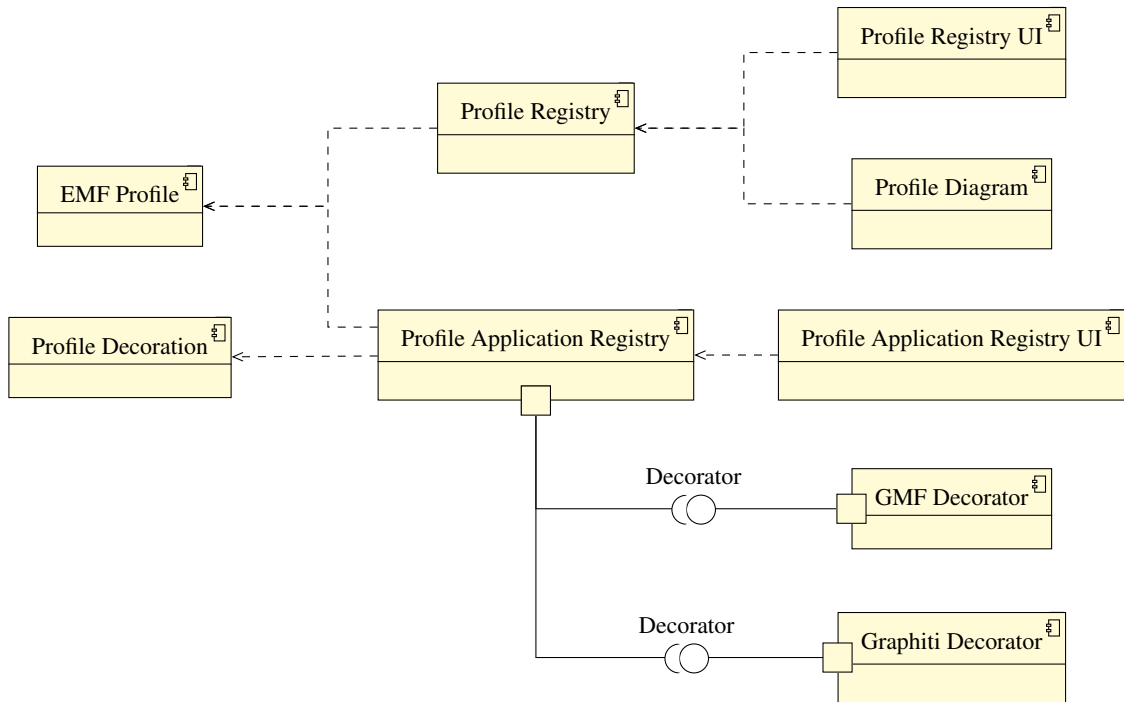


Figure 6.1: Architecture of the solution — the overview of the most important components.

directory. For every profile definition found, a record of it is created by the registry. This makes it very convenient to use the registry to query for profile definitions and to get an instance of them.

- **Profile Registry UI** component provides the implementation of a simple tree-view user interface for displaying profile definitions that the Profile Registry has a record of. It has the dependency on — in other words, it uses — the Profile Registry to get the information on profile definitions that are displayed in the view — dependency is indicated by the dashed line pointing from the Profile Registry UI to the Profile Registry.
- **Profile Diagram** component is the implementation of the graphical concrete syntax for profile definition modeling language. It is implemented with GMF and has the concrete syntax, with which we are already familiar with, illustrated in profile definitions created for our running example in Chapter 3 (cf. Figure 3.2 and Figure 3.5).
- **Profile Decoration** component comprises the specification of the *decoration description language* and the implementation of the components that realize its tools, e.g., the textual editor that supports its textual concrete syntax. The Chapter 4 can be consulted to get the informations on how this language was engineered and realized with Xtext, and Chapter 5 for description of its usage.

- **Profile Application Registry UI** component provides, similarly to the Profile Registry UI, the implementation of the tree-view user interface (cf. Figure 5.1e). However, in this case the view displays profile definitions that are applied at runtime to the model instance currently opened in an active editor. In other words, it displays profile applications and stereotype applications that the user applied to the model and its model elements. Furthermore, this component provides the implementation of the Eclipse *Properties View* for displaying and manipulating tagged values of applied stereotypes. The component depends on the *Profile Application Registry* which is explained in the next section.

The components *Profile Application Registry*, *GMF Decorator*, and *Graphiti Decorator* are explained in more detail in following sections.

## 6.2 Profile Application Registry

As in the case of the Profile Registry, this component also acts as a “register office”. It keeps the record of the profile applications that a user has created or loaded for the models that are currently opened in corresponding editors in Eclipse. The excerpt of the Profile Application Registry metamodel is depicted in Figure 6.2. The classes from the metamodel represent the application programming interface (API) of the Profile Application Registry, which is visible outside the component (the plug-in in Eclipse terms) and can be used by client programs to apply a profile to a model and to apply the stereotypes of the profile definition to the elements of the model. An example of the client that uses the API is the Profile Application Registry UI component.

How a client uses the API to create a profile application and to apply stereotypes to model elements can be explained as the process described in the following steps:

- **Step 1:** Provided that the client has specified on which components it depends, the client gets hold of the singleton instance of the `ProfileApplicationRegistry` class.
- **Step 2:** Using the `ProfileApplicationRegistry` instance, the client then requests an instance of the `ProfileApplicationManager` by calling the method `getProfileApplicationManager()`. The client must provide an instance of the `ResourceSet` and the *Editor ID* as its parameters. Both can be obtained from the editor in which the model is opened. The Editor ID is used to easily identify on which technology is the editor implementation based on, e.g., GMF, Graphiti, or some other technology. The `ProfileApplicationManager` object is bound to the model instance and exists as long the model is in use by the user.
- **Step 3:** The `ProfileApplicationManager` is used to create new profile applications or to load existing ones for the model. In any case, the profile application manager returns an instance of the `ProfileApplicationWrapper` for each applied profile or loaded profile application. The `ProfileApplicationWrapper` essentially wraps an instance of the `ProfileApplication` to provide additional functionalities for convenient usage that go beyond offered functionality of the `ProfileApplication` object.

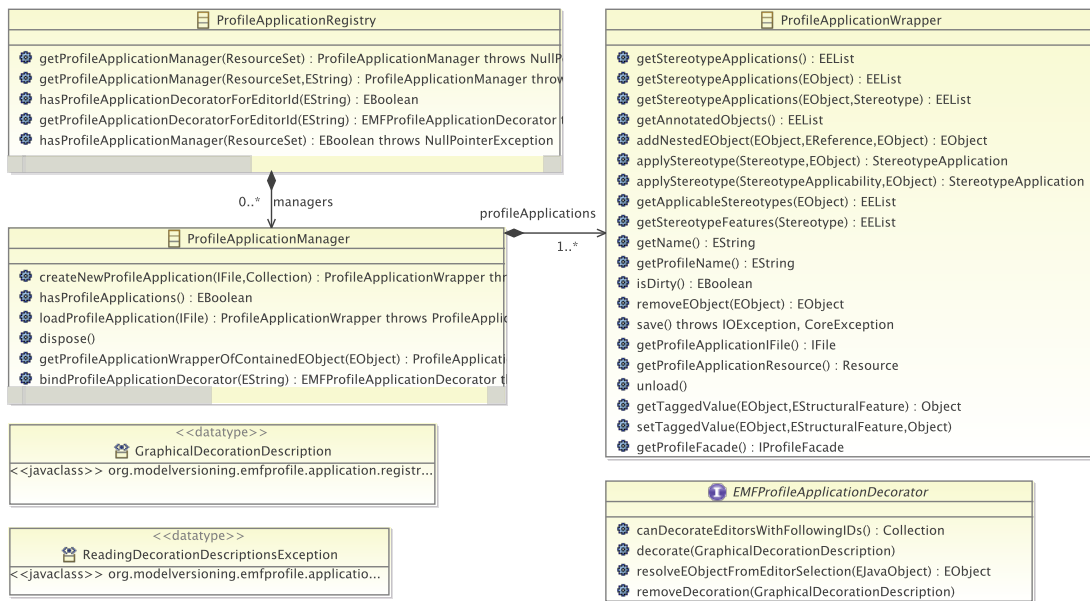


Figure 6.2: Excerpt of the Profile Application Registry metamodel.

- **Step 4:** The `ProfileApplicationWrapper` is then used to apply stereotypes to model elements, set tagged values, remove stereotype applications, and so forth. Furthermore, the user can inquire the status of the model resource, e.g., to see if it is modified, and also save the current state of the profile application to the file.
- **Step 5:** When the user is finished working with the model, for example, the latest moment would be when the user closes the model editor, the client program should dispose the `ProfileApplicationManager` in order to release all resources that it holds, hence to perform a cleanup.

These steps describe how a client can extend a model by applying profiles and stereotypes. However, the explanation on how and when the graphical decoration of model elements occurs, is still missing.

When we look at the architecture overview in Figure 6.1, we see that the Profile Application Registry uses the Profile Decoration component for some purpose, and that it also specifies the *required interface* `Decorator`. The `Decorator` interface is, in Eclipse terms, the specification of an *extension point*, which is used to make a component open for extensibility in terms of specifying the required interface that other plug-ins can provide an implementation for — in jargon of the UML Component Diagram the implementation of required interface is known as the *provided interface*. Moreover, there can be many provided interfaces for a required one, and they can be used interchangeably. `GMF Decorator` and `Graphiti Decorator` components contribute the implementation for the `Decorator` interface. The concrete Java interface

that decorator components must implement is the `EMFProfileApplicationDecorator` interface (cf. Figure 6.2).

As it is obvious by now, the Profile Application Registry is the component that manages not only the profile applications for models but also the graphical decorations of their model elements, which are based on decoration description specifications. How does the component manage graphical decorations is described in the following:

- At the startup of the Eclipse platform, the Profile Application Registry component requests from the platform all registered components that provide the implementation of the Decorator interface. After that, the Profile Application Registry inquires the decorator components for Editor IDs that they support. Out of that information an internal mapping is created to easily obtain the instance of the decorator for a specific Editor ID.
- When a client program request an instance of the `ProfileApplicationManager`, as previously described in Step 2, the client must also provide an Editor ID. The Profile Application Registry uses that Editor ID to bind an adequate instance of a decorator, which is used consequently to add or remove graphical decorations from model elements.
- At the moment the client sets on to apply a profile definition or to load an already existing profile application, which corresponds to previously described Step 3, the *decoration descriptions model* resource is looked up at the same location the profile definition model resource is located. When found, the component then uses the parser and the validator of the decoration description language, found in the Profile Decoration component, to get an instance of the decoration descriptions model.
- Now, for every stereotype application its decoration descriptions are collected, and in the next step, their activation conditions are evaluated in order to determine if the corresponding graphical decorations should be visualized or not.
- The `GraphicalDecorationDescriptions` (cf. Figure 6.2) are created for each stereotype application containing all relevant informations, previously collected and calculated, to be relayed to the concrete decorator component that can handle graphical decorations in the editor of the model.

Concrete decorator components provide the implementation to interpret the informations provided in the `GraphicalDecorationDescription` objects, and also use the facilities of their supported graphical frameworks to influence the graphical concrete syntax of model elements in the editor.

Furthermore, the Profile Application Registry observes the model data for changes such as that of a profile application model or a decoration descriptions model. For every change event that is observed, adequate measures are taken in order to update graphical decorations of model elements in the editor. This normally leads to reevaluating the activation conditions of decoration descriptions to reflect the changes caused by the model modification.

This allows, for example, for instant feedback of decoration descriptions manifestation in the editor. The user only needs to apply a profile and stereotypes to the model, and then can start

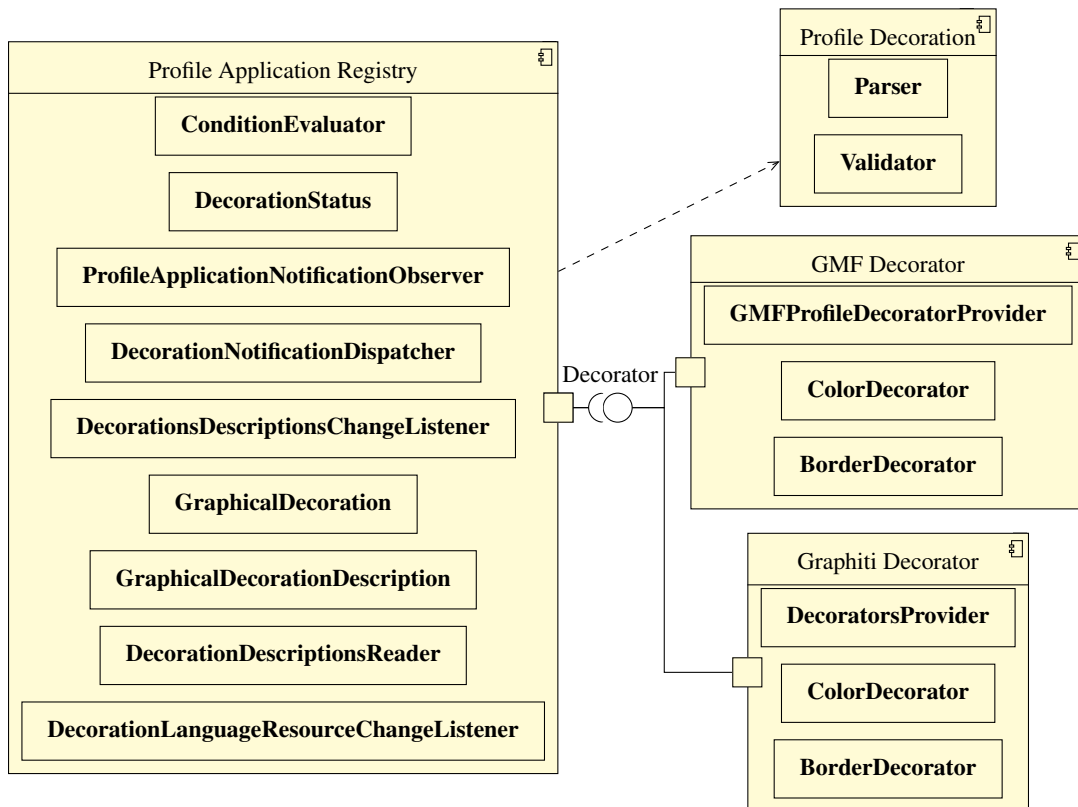


Figure 6.3: Excerpt of the classes involved in processing decoration descriptions.

to specify decoration descriptions for the applied stereotypes — or modify existing ones until the desired graphical decoration is achieved. Every time the decoration descriptions model is saved in its editor, that triggers a notification to the observer, which then relays the notification to the responsible handler class.

To portray some of aforementioned features, Figure 6.3 depicts an excerpt of the implementation classes involved in processing decoration descriptions and relaying the relevant information, for graphical decoration of model elements, to responsible graphical decorators.

### 6.3 GMF Decorator

The GMF Runtime framework offers different kind of diagram extension services such as the *Layout Service*, the *Palette Service*, and — most important to us — the *Decoration Service*.

The Decoration Service is designed to provide a simple way of annotating an existing shape, where the provider of the decoration does not even need to know the implementation of that specific shape. This is true if we just want to adorn the shape with some other graphical elements such as an icon for example. However, if we want to change some visual properties of the

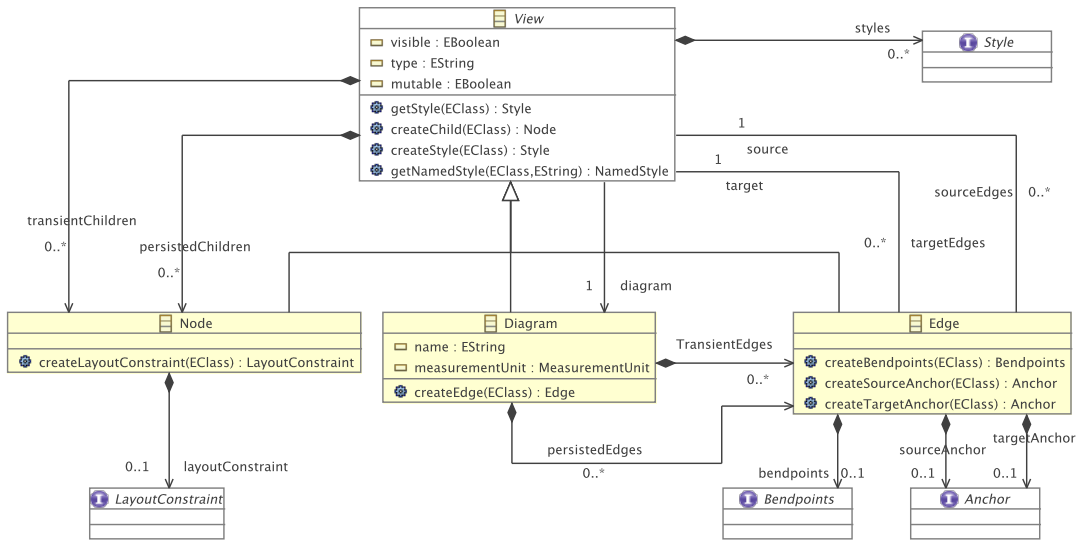


Figure 6.4: Excerpt of the GMF Notation metamodel.

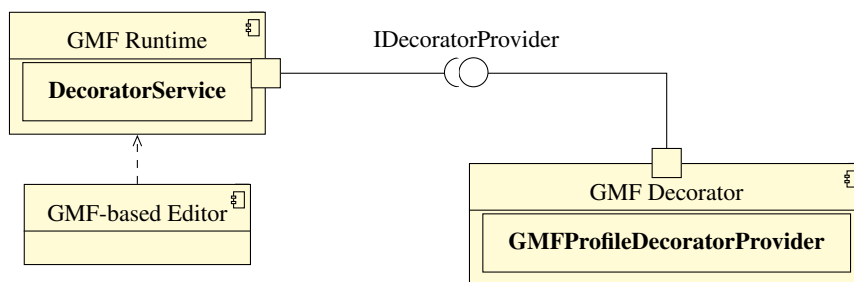


Figure 6.5: GMF DecorationService and IDecoratorProvider contribution.

shape, then we have to go deeper and take appropriate steps, which are shape-implementation dependent. Luckily, the *Notation Metamodel*, as illustrated in Figure 6.4, is used to describe graphical figures, and it only has few different shape definitions, namely, the *Node*, the *Edge*, and the *Diagram*. The *Diagram* contains nodes and edges, which correspond to, e.g. a rectangular shapes denoting model elements and line shapes denoting relations between model elements.

As previously said, the GMF Runtime framework offers the Decoration Service, which utilizes the Eclipse extension point and extension contribution mechanism. Thus, the GMF runtime specifies the required interface `IDecoratorProvider` and our GMF Decorator provides an implementation for it, as illustrated in Figure 6.5. This is exactly what enabled us to provide a general solution to extending the graphical concrete syntax (GCS) of modeling languages, which GCS is based on GMF framework.

## 6.4 Graphiti Decorator

The Graphiti framework, sadly, does not provide a decoration service as GMF does. Thus, no general solution to the extension of the graphical concrete syntax based on Graphiti framework is possible. Each newly created editor, based on Graphiti, must be modified. We tried to keep this modification minimal as possible. The dependency of components is illustrated in Figure 6.6. The editor must specify the dependency to the Graphiti Decorator and use its API — the `DecoratorsProvider` class — to register every graphical element that is directly related to a model element, so that at later time when a stereotype is applied to a model element the Graphiti Decorator can decorate its graphical representation according to decoration descriptions. This approach was designed to mimic the approach realized in the GMF Decoration Service.

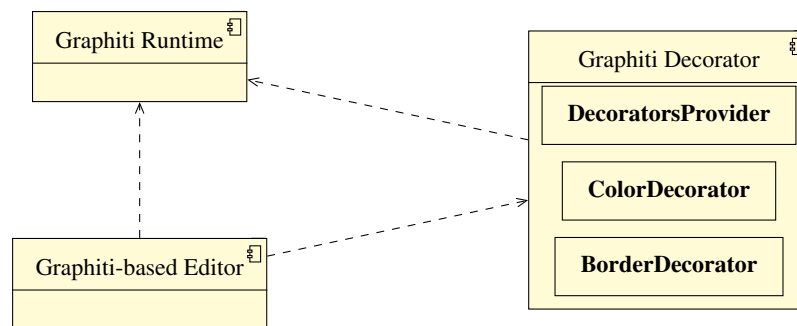


Figure 6.6: The Graphiti-based editor dependencies for supporting GCS extension.

Graphiti framework supports the lightweight extension of the graphical concrete syntax by providing following decoration facilities:

- **Border Decorator** enables adding a border to the visualization of a shape. The border can be styled by color, line-style and line-thickness.
- **Color Decorator** can be used to modify foreground and background color of a shape.
- **Image Decorator** adds an image to the visualization of a shape.
- **Text Decorator** can be used to add a text to the visualization of a shape.

In order to collect the decorators of the visual shape of a model element, the Graphiti runtime places the call to the `IToolBehaviorProvider.getDecorators(PictogramElement)` method of the class implementing it. Normally, every Graphiti-based editor must provide an implementation of the `IToolBehaviorProvider` interface. Consequently, this is exactly the place where that call must be redirected to the `DecoratorsProvider` class, from the Graphiti Decorator component, in order to enable the graphical concrete syntax extension in the editor.



# Evaluation

In this chapter we evaluate how well our solution is applicable to the graphical concrete syntax extension of a modeling language. The evaluation is based on the case study of visualizing model execution. Thus, for the purpose of the evaluation, we create the language for modeling Petri nets [41], and provide two graphical concrete syntax specifications, based on both GMF and the Graphiti framework respectively. Then, we provide the *Petri Net Execution Profile* for extending Petri net model elements with the information on the model execution state, and build a simple *Execution Simulator* that uses the EMF Profiles API to apply the stereotypes from the execution profile, in order to simulate the execution. Finally, we run the execution simulation on a Petri Net model visualized first in its GMF-based GCS and then run it again for the same model visualized in its Graphiti-based GCS. For each of them, the evaluation of the solution is discussed.

## 7.1 Case Study

The purpose of the case study is to evaluate if the requirements, specified in Chapter 3, are satisfied by our solution for the extension of modeling languages — the special focus here is on the extension of the graphical concrete syntax of a modeling language. In Chapter 1, we said that we will base the evaluation of our solution on a representative case study, and that we are particularly interested to assess how the runtime information of executable models can be visualized appropriately and dynamically updated during the execution with EMF Profiles. Hence, the following research questions can be deduced:

- Is it possible, at runtime, to place decorations on model elements so that they reflect the information on the execution of the model? Or in a more granular form:
  - Can we change the color of an element, highlight it somehow, in order to grasp where the execution is actually occurring?

- Can we add additional visual elements, that are part of the execution visualization and not of the modeling language itself?
- Can we extend the graphical concrete syntax of a model element so that concrete values from the profile can also be visualized?
- Another very important question: Can these decorations be dynamically changed and updated based on the runtime information of the execution model?

We will ascertain the answers to these research questions, but first let us introduce our case study modeling language.

## Petri Net Modeling Language

Petri nets are a graphical and mathematical modeling tool that can be applied to different modeling domains [41]. They are well suited for describing and studying the information systems that are characterized, for example, as being concurrent, asynchronous, distributed, parallel, and nondeterministic [41]. The modeling concepts of the Petri net language are illustrated in Figure 7.1. A Petri net consists of *places*, *transitions*, and *arcs*. Arcs are used to connect the places with transitions, and vice versa. The connections between places, and between transitions are not possible. The places from which an arc is pointing to a transition are called the *input places* of the transition, and the places to which an arc is pointing from a transition are called the *output places* of the transition. Furthermore, places in the Petri net diagram can hold any discrete number of *tokens*. Their distribution over the places represent the configuration of the net and it is called a *marking* of the net.



Figure 7.1: Petri net modeling concepts.

The transitions of a Petri net may *fire* if they are enabled. This firing act, which is atomic<sup>1</sup> in its nature, may occur if all input places of a transition contain sufficient tokens, hence enabling the transition. If the firing of a transition occurs, it consumes the required tokens from the input places and creates tokens in the output places. The execution policy of the Petri net is nondeterministic, which means that if many transitions are enabled at the same time, any of them may fire. This makes Petri nets very useful for modeling the concurrent and parallel behavior of distributed systems.

Our Petri net model instance, which we use to evaluate our solution, describes exactly such a scenario — in parallel computing known as the *fork-join model* — where an execution process

<sup>1</sup>**Atomic** refers to a single non-interruptible step in the execution process.

branches off (forks) in parallel and concurrent executions that are *merged* (joined) at a subsequent point, whereafter the process resumes sequential execution. The scenario is illustrated in Figure 7.5. The model instance also describes a concurrent execution scenario where two *threads* of execution are competing for access to mutually shared data. The access to the mutually shared data should be exclusive to only one thread at the same point in time — this is denoted as the *critical section* place in the model. To ensure mutual exclusion a *locking mechanism* is employed that allows entrance to only one thread and blocks all other competing threads entering the critical section, as long it is occupied.

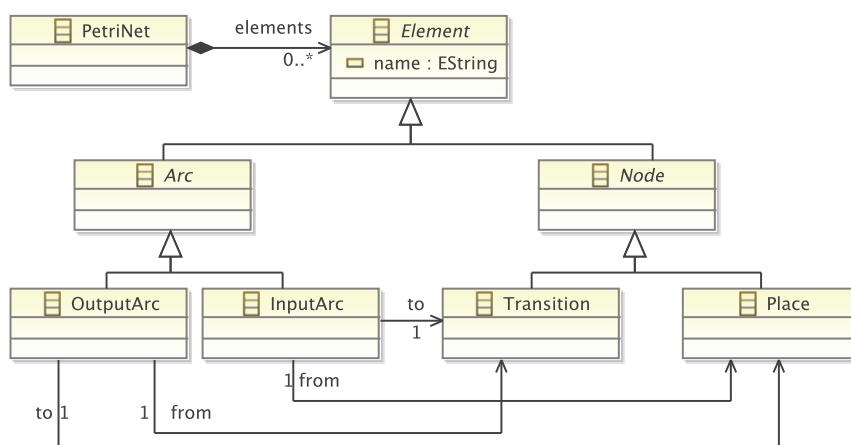


Figure 7.2: Petri net metamodel.

The metamodel of our case study *Petri net* modeling language is depicted in Figure 7.2. To note is that this Petri net language specification does not foresee the marking of the net. With this language we can model a Petri net, but we can not specify its configuration because there is no language concept for tokens. Tokens are introduced as the extension of the language through profiling — of course, by using EMF Profiles.

### Execution Profile and the Model Execution Simulator

The *execution profile* for the Petri net modeling language is depicted in Figure 7.3. From the profile specification we can see that the `Place` modeling element can be extended with the `Token` stereotype, when applied to it. The `Token` stereotype contains the amount tagged value for specifying discrete number of tokens residing in a place. Likewise, we see that modeling elements `Transition` and `Arc` can have the `Activation` stereotype applied to them. The `Activation` stereotype contains `activated` tagged value (of Boolean type) that can be set to `true` or `false`.

When stereotypes are applied to model elements, the graphical concrete syntax of the model elements should be extended based on the decoration descriptions specification listed in List-

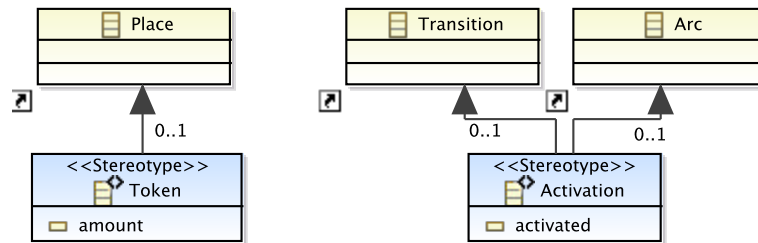


Figure 7.3: Execution profile for our Petri net modeling language.

ing 7.1. For example, when the `Token` stereotype is applied to a place, it should be highlighted by changing its foreground color to dark green, and the token image should be placed in the middle of it with the current token `amount` displayed above the token image. Also, when the `Activation` stereotype is applied to a transition or an arc, they should be highlighted in red or green color, based on the current value of the `activated` tagged value.

Listing 7.1: Decoration descriptions for stereotypes of the Petri net execution profile.

```

1 import resource 'profile.emfprofile_diagram'
2
3 profile PetriNet_ForkJoinProfile
4
5 decoration Token {
6   box {
7     image {
8       uri = "platform:/resource/ForkJoin_PetriNet_ExecutionProfile/icons/
9         token.png"
10      placement = SOUTH
11    }
12    direction = CENTER
13    width = 20
14    height = 30
15    text = amount
16    foregroundColor = BLUE
17    tooltip = amount + " Token(s) "
18  }
19
20  color {
21    foreground = GREEN_DARK
22    active when ALL // just to demonstrate the compsite condition type
23      rule
24      (
25        amount > 0
26        amount < 4
27      )
28  }
29  color {
30    active when amount == 0
31    background = ORANGE
  
```

```

30     }
31 }
32
33 decoration Activation {
34     color {
35         active when activated == true
36         background = GREEN_DARK
37         foreground=GREEN_DARK
38     }
39     color {
40         active when activated == false
41         foreground=RED
42         background = RED
43     }
44
45     connection {
46         active when activated == true
47         color = GREEN_DARK
48     }
49     connection {
50         active when activated == false
51         color = RED
52     }
53 }

```

As previously said, we have created a simple *model execution simulator* to evaluate the applicability of our solution to the visualization of model execution. The simulator was designed to emulate a model execution engine, however, it is only applicable to the model instance illustrated in Figure 7.5. The simulator, of course, uses the EMF Profiles API in order to apply the execution profile to the model, and likewise to extend model elements by applying stereotypes from the profile.



Figure 7.4: Model Execution Simulator user interface.

The simulator is implemented as an Eclipse plug-in with the user interface contribution depicted in Figure 7.4. The *Start/Stop Simulation* button applies or removes the application of the execution profile to or from the model instance currently opened in the graphical editor. The *Next Step* button, when clicked repeatedly, plays through the execution steps of the scenario that

is specific to our model instance.

## 7.2 Evaluation of the Extension of the Graphical Concrete Syntax based on GMF

In order to create the GMF-based graphical concrete syntax for our Petri net modeling language (cf. Petri net metamodel in Figure 7.2) we decided to employ the EuGENia<sup>2</sup> project, as already mentioned in Chapter 2. In short, EuGENia is very useful to jump-start the creation of the GCS for a modeling language. The developer annotates the metamodel elements with concrete syntax descriptions in form of textual annotations, as we can see it in the Listing 7.2. The listing shows the metamodel in the textual notation of the *Emfatic*<sup>3</sup> language. From the textual specification, by the means of model transformations, EuGENia generates GMF models that otherwise would have to be manually specified by the developer. After that, GMF code-generation facilities are used to generate the implementation of the editor supporting the graphical concrete syntax of our Petri net modeling language. Figure 7.5 exemplifies the GMF-based GCS.

Listing 7.2: Petri net metamodel with graphical concrete syntax specifications in Emfatic textual notation.

```
1 @namespace(uri="http://petrinet", prefix="PetriNet")
2 package petrinet;
3
4 @gmf.diagram(foo="bar")
5 class PetriNet {
6     val Element[*] elements;
7 }
8
9 abstract class Element {
10     attr String name;
11 }
12
13 abstract class Node extends Element {
14 }
15
16 @gmf.node(label.icon="false", label="name", figure="rectangle", color="
17     0,0,0", label.placement="external",size="40,15")
18 class Transition extends Node {
19 }
20
21 @gmf.node(label = "name", label.icon="false", figure = "ellipse", border.
22     width="2", border.style="solid",label.placement="external",size="40,40
23     ", color="20,20,20")
24 class Place extends Node {
25 }
26
27 abstract class Arc extends Element {
28 }
```

<sup>2</sup>EuGENia. <http://www.eclipse.org/epsilon/doc/eugenia/>

<sup>3</sup>Emfatic - A textual syntax for EMF Ecore (meta-)models. <http://www.eclipse.org/emfatic/>

```

26
27 @gmf.link(target.decoration="arrow", source="from", target="to", incoming=
    "false", label.icon="false",label = "name", target.decoration="
    filledclosedarrow", color="20,20,20")
28 class OutputArc extends Arc {
29     ref Transition[1] from;
30     ref Place[1] to;
31 }
32
33 @gmf.link(target.decoration="arrow", source="from", target="to",incoming="
    false", label.icon="false", label = "name", target.decoration="
    filledclosedarrow")
34 class InputArc extends Arc {
35     ref Place[1] from;
36     ref Transition[1] to;
37 }

```

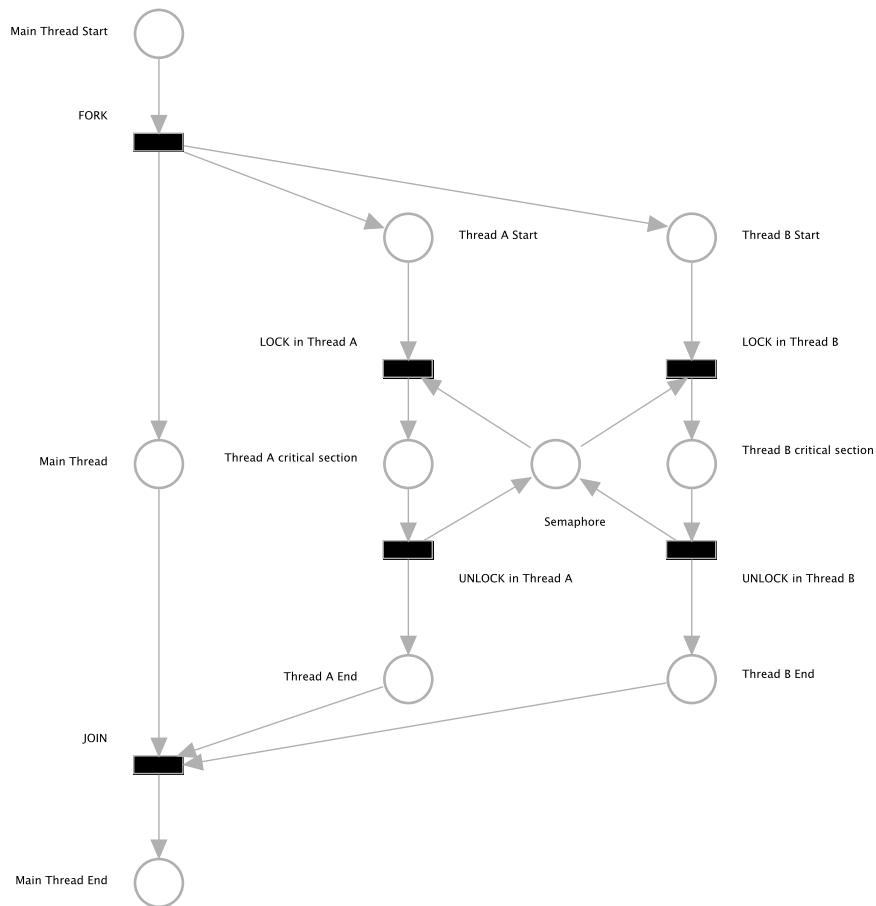


Figure 7.5: Fork-Join model in Petri net language.

## Performing Execution Simulation

In the following, we are going to illustrate the changes in the GMF-based graphical concrete syntax of our Petri net language that come into effect when model execution simulation progresses. We are not going to illustrate all the execution steps, since this would take too much space, but still enough of them so that the execution steps can be comprehended.

The visual changes in the editor can be reasoned as following: When a transition and arcs that connect it with its input and output places are colored in red, it means that the transition is enabled and that it might fire in the subsequent step; Otherwise, if they are colored in green, it means that the transition has fired, which as a consequence consumes tokens from input places and creates tokens in output places, which is visible in the subsequent execution step.

The illustrations that show few execution steps of the simulation begin with Figure 7.6 and end with Figure 7.12.

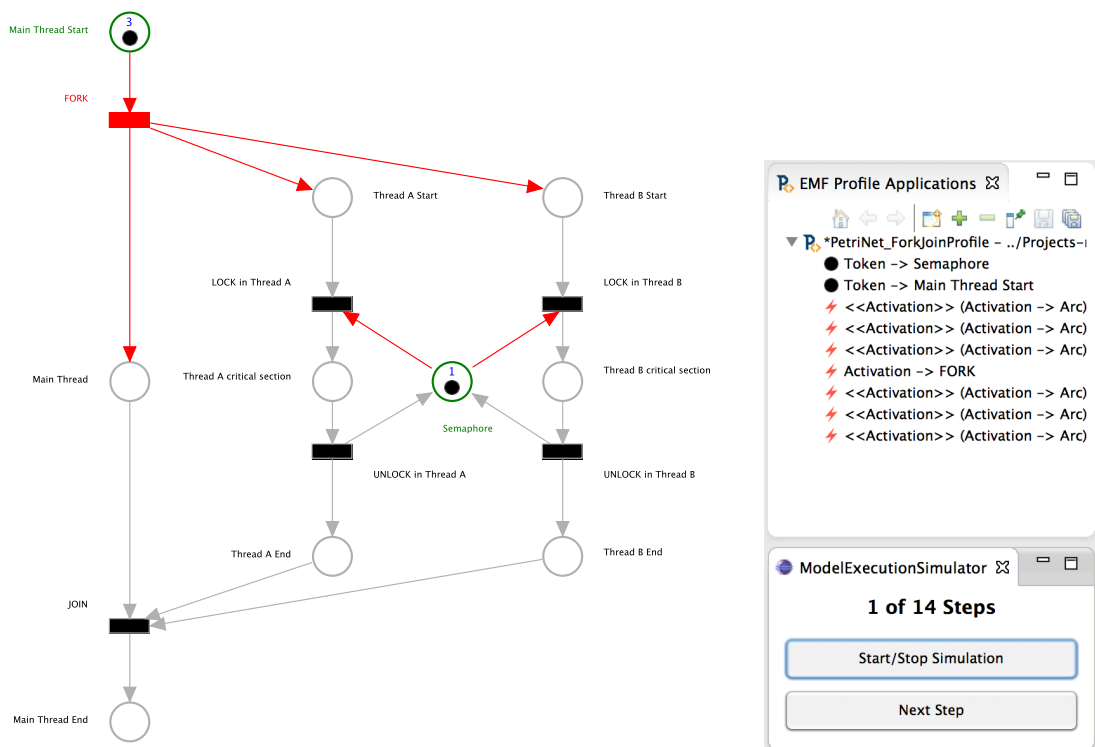
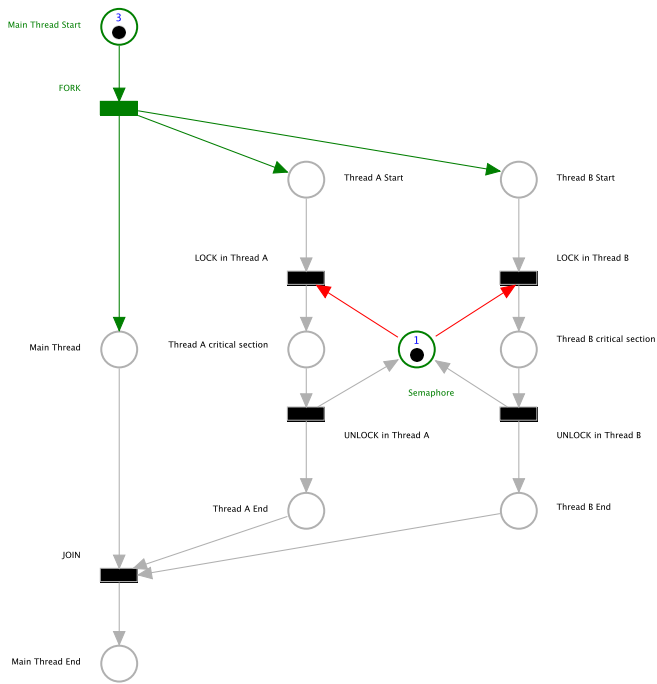


Figure 7.6: Execution simulation on GMF-based GCS – Step 1.

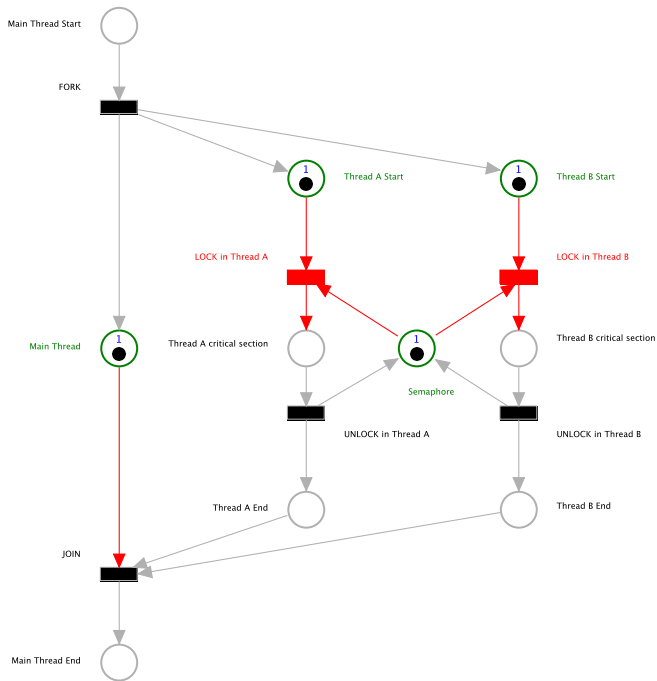




(a) Diagram.

(b) Applied Stereotypes.

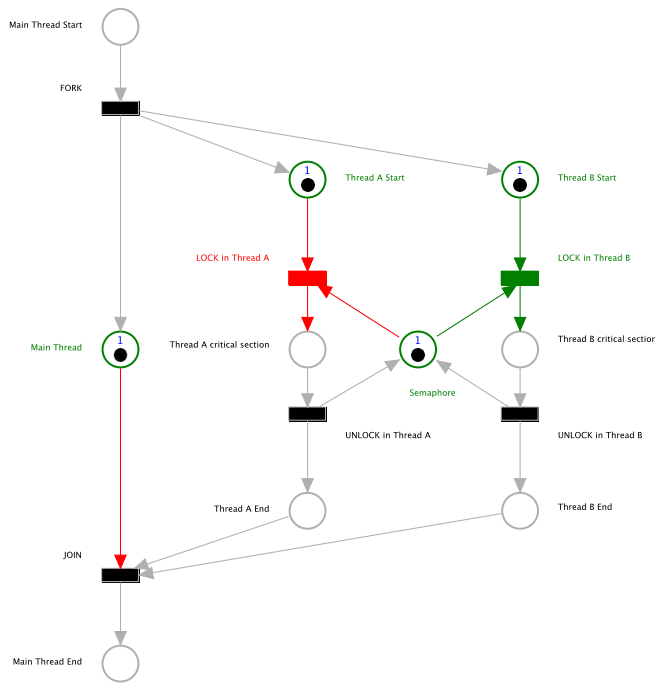
Figure 7.7: Execution simulation on GMF-based GCS – Step 2.



(a) Diagram.

(b) Applied Stereotypes.

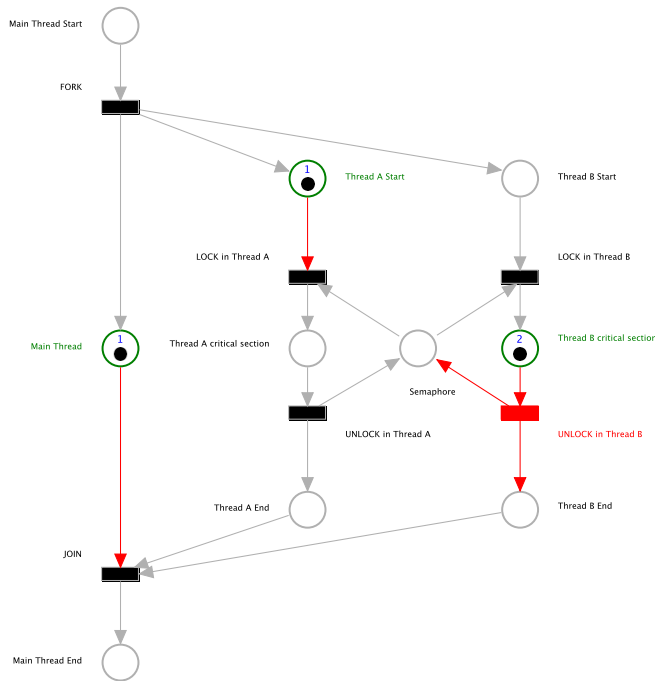
Figure 7.8: Execution simulation on GMF-based GCS – Step 3.



(a) Diagram.

(b) Applied Stereotypes.

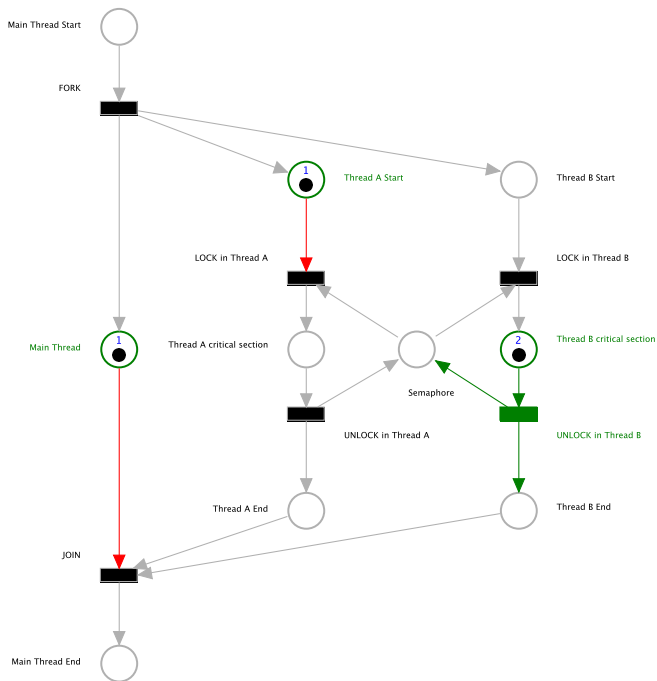
Figure 7.9: Execution simulation on GMF-based GCS – Step 4.



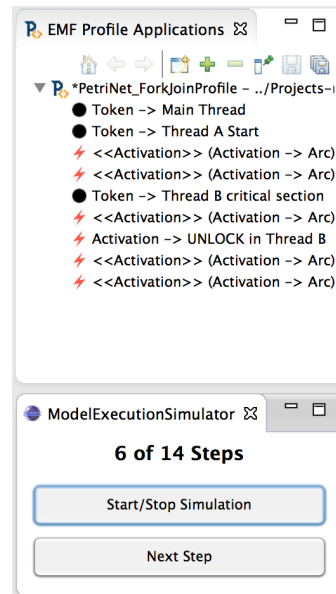
(a) Diagram.

(b) Applied Stereotypes.

Figure 7.10: Execution simulation on GMF-based GCS – Step 5.

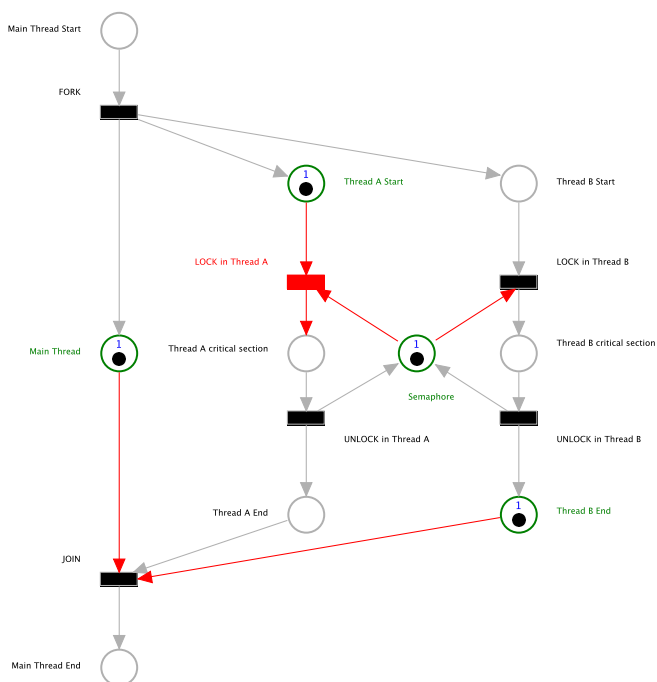


(a) Diagram.

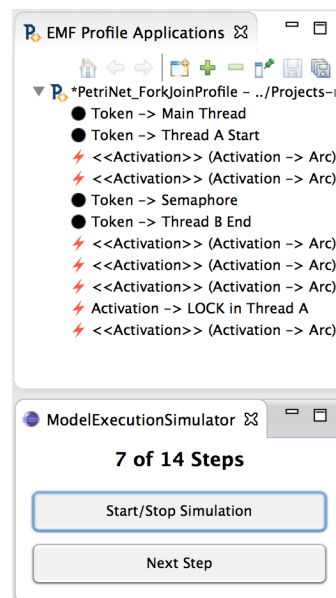


(b) Applied Stereotypes.

Figure 7.11: Execution simulation on GMF-based GCS – Step 6.



(a) Diagram.



(b) Applied Stereotypes.

Figure 7.12: Execution simulation on GMF-based GCS – Step 7.

### 7.3 Discussion of the GMF-based GCS Extension

From the illustrations of the model execution simulation (cf. Figure 7.6 to Figure 7.12) we can say that all our research questions can be answered with *yes*, and that all our requirements, defined in Chapter 3, that we set to accomplish, were indeed fully satisfied. The Graphical Modeling Framework (GMF) has been well designed to support GCS extensibility. The only thing that we might criticize is the lack of or insufficient documentation on how to use the framework — this we experienced while we were implementing the solution for our problem domain.

### 7.4 Evaluation of the Extension of the Graphical Concrete Syntax based on the Graphiti Framework

Although, creating the graphical concrete syntax for a modeling language in Graphiti is API-centric, we have decided — as already mentioned in Chapter 2 — to use the mapping-centric approach to get an editor implementation that uses the Graphiti runtime. Therefore, we have decided to employ Spray<sup>4</sup> project for the specification of the Graphiti-based graphical concrete syntax for our Petri net modeling language. Spray project aims to provide few textual DSLs for the description of GCS, and the mapping between model elements and their visualization representations. Based on GSC description and mapping specification, Spray generates the boilerplate code for realizing the implementation against the Graphiti framework [1].

GCS description is known as the *Shape model*, and the mapping specification is known as the *Spray model*. The Shape model for our Petri net modeling language is listed in Listing 7.3, and respective Spray model is listed in Listing 7.4

Listing 7.3: Definition of graphical figures for Petri net language using Spray’s Shapes language.

```
1  shape PlaceShape {
2    stretching (horizontal=false, vertical=false) proportional=true
3    ellipse {
4      position (x=0, y=0)
5      size (width=40, height=40)
6      style (line-color=RGB (20, 20, 20) line-width=1)
7    }
8  }
9
10 shape TransitionShape {
11  stretching (horizontal=false, vertical=false) proportional=true
12  rectangle {
13    position (x=0, y=20)
14    size (width=40, height=15)
15    style (background-color=black)
16  }
17 }
18
19 connection ArrowConnectionShape {
```

<sup>4</sup>Spray project. <http://eclipselabs.org/p/spray>

```

20 style (line-width=1)
21 placing {
22   position (offset=1.0)
23   polygon {
24     point (x=-15, y=-10)
25     point (x=0, y=0)
26     point (x=-15, y=10)
27     style (background-color=black)
28   }
29 }
30 placing {
31   position (offset=0.1, radius=10, angle=0)
32   text {
33     position (x=0, y=0)
34     size (width=40, height=14)
35     id = sourceText
36   }
37 }
38 placing {
39   position (offset=0.5, radius=10, angle=180)
40   text {
41     position (x=0, y=0)
42     size (width=40, height=14)
43     id = arcName
44   }
45 }
46 placing {
47   position (offset=0.9, radius=10, angle=180)
48   text {
49     position (x=0, y=0)
50     size (width=40, height=14)
51     id = targetText
52   }
53 }
54 }

```

Listing 7.4: Mapping definition between Petri net metamodel elements and their graphical representations using Spray language.

```

1 import petrinet.*
2
3 diagram Mypetrinet for PetriNet style PetrinetDefaultStyle
4
5 class Place {
6   shape PlaceShape {
7   }
8   behavior {
9     create into elements palette "Elements" askFor name;
10  }
11 }
12
13 class Transition {
14   shape TransitionShape {

```

```

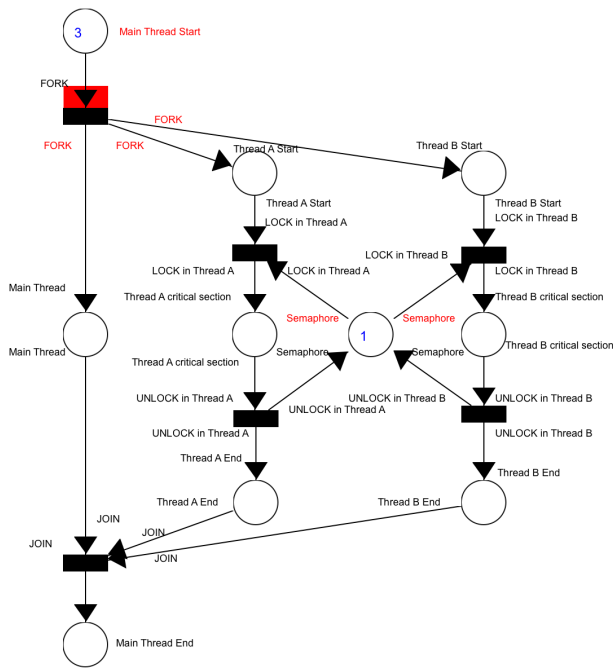
15     }
16     behavior {
17         create into elements palette "Elements" askFor name;
18     }
19 }
20
21 class OutputArc {
22     connection ArrowConnectionShape {
23         name into arcName
24     }
25     {
26         from ^from;
27         to ^to;
28     }
29     behavior {
30         create into elements palette "Arcs";
31     }
32 }
33
34 class InputArc {
35     connection ArrowConnectionShape {
36         name into arcName
37     }
38     {
39         from ^from;
40         to ^to;
41     }
42     behavior {
43         create into elements palette "Arcs" ;
44     }
45 }

```

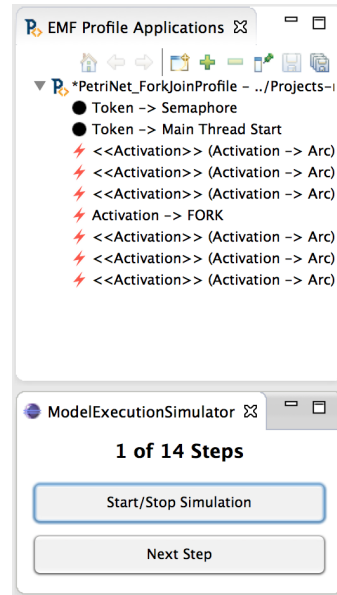
## Performing Execution Simulation

The following illustrations (cf. Figure 7.13 to Figure 7.18) show the manifestations of the Graphiti-based graphical concrete syntax changes, which occur when we run through few steps of our execution simulator on a model instance. The model instance resembles the model instance in Figure 7.5 in every respect. The only difference is in the graphical concrete syntax, which in this case is Graphiti-based.

The visual changes in the editor are to be reasoned in the same way as previously described in the model execution simulation performed for the model instance opened in the GMF editor.

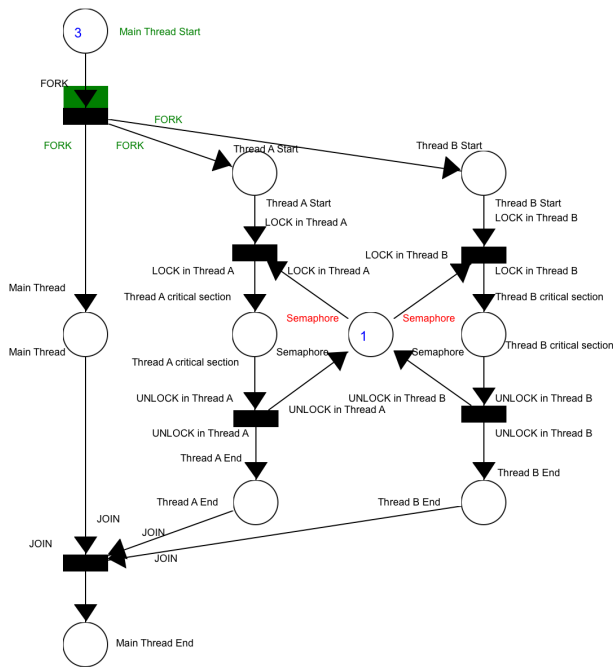


(a) Diagram.

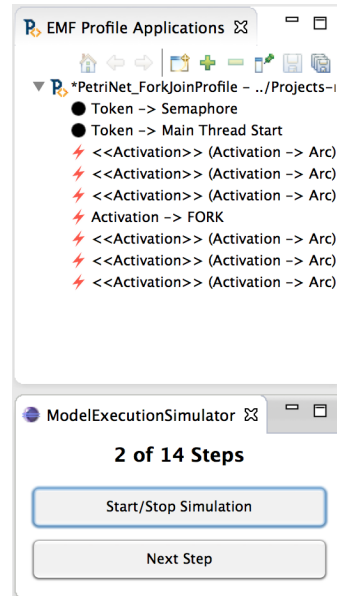


(b) Applied Stereotypes.

Figure 7.13: Execution simulation on Graphiti-based GCS – Step 1.

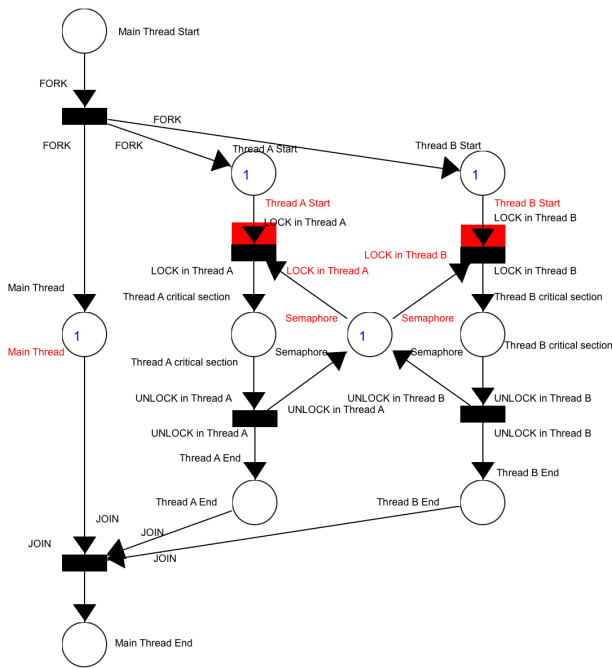


(a) Diagram.



(b) Applied Stereotypes.

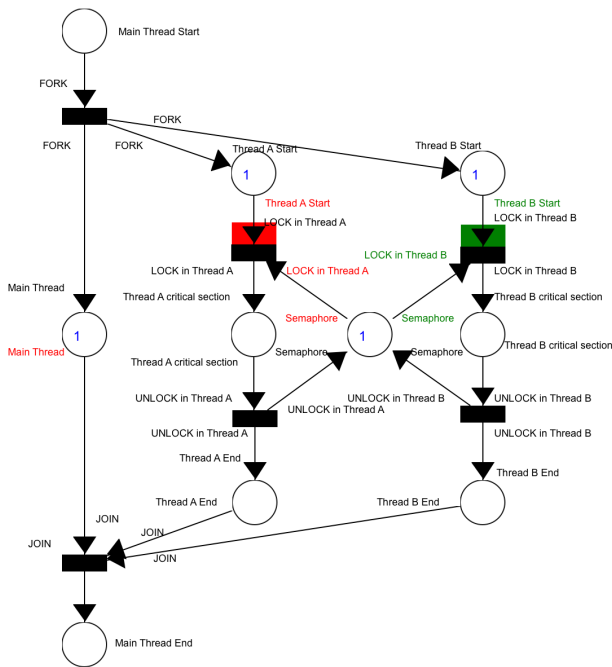
Figure 7.14: Execution simulation on Graphiti-based GCS – Step 2.



(a) Diagram.

(b) Applied Stereotypes.

Figure 7.15: Execution simulation on Graphiti-based GCS – Step 3.

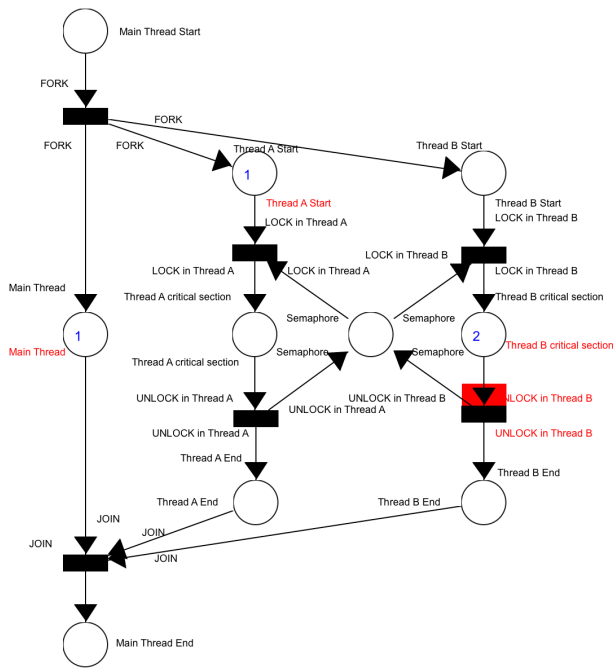


(a) Diagram.

(b) Applied Stereotypes.

Figure 7.16: Execution simulation on Graphiti-based GCS – Step 4.

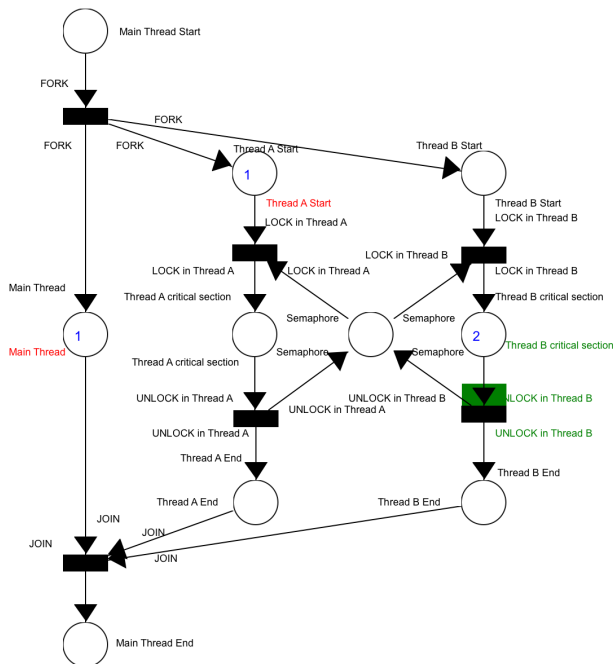




(a) Diagram.

(b) Applied Stereotypes.

Figure 7.17: Execution simulation on Graphiti-based GCS – Step 5.



(a) Diagram.

(b) Applied Stereotypes.

Figure 7.18: Execution simulation on Graphiti-based GCS – Step 6.

## 7.5 Discussion of the Graphiti-based GCS Extension

We see from the illustrations of the execution steps (cf. Figure 7.13 to Figure 7.18) that, for example, token images are missing — only the token amount is visible on a place model element. The reason for missing token images is due to a shortcoming of the Graphiti framework, which will be explained shortly.

The color decorations on the connection lines do not have the desired effect. We see that the connection lines do not change their color, based on `ConnectionDecoration` values. However, the text color of connection names do change their color. It seems that decorating connection lines is not thoroughly implemented in the Graphiti framework.

Furthermore, we see that the color decorations on the place and transition model elements do not show the desired effect. Indeed, they are not coloring the correct figure. This is, in fact, due to how Spray generates figure objects for shape definitions. Instead of generating the code against the Graphiti runtime that instantiates a figure object resembling the figure form, Spray generates pictures of the shapes and draws them in a containing graphical node element at runtime. That is why, for example, the background color on a transition model element looks like it is leaking from behind — the visual properties of the container figure are changed. This is an excellent example of the problem we discussed in the summary of Chapter 4.

**Based on our experience, we came to the following conclusion:** Extending the graphical concrete syntax of any modeling language, implemented with Graphiti, through a general implementation of a decorator, as we intended to provide, has proven to be impossible with the current implementation of the Graphiti framework. The reason is mainly in few shortcomings of the framework we encountered as we started to implement our Graphiti decorator.

The first problem is that the Graphiti framework does not provide the necessary means of extending the original editor implementation through additional plug-ins. Normally, this is done in Eclipse platform through so-called *extension points*, where one plug-in defines them — analogically this would be equivalent to defining an interface in Java without an implementation — and another plug-ins may provide an implementation for them. The original plug-in then gets the hold of an extension point implementation through the means that the Eclipse platform provides. This is obviously one very convenient feature, and as a matter of fact, this is exactly how GMF addresses the decoration of graphical elements, but as already said, Graphiti does not provide such a feature.

Despite of that shortcoming, we went on and tried to see what would be the minimal code extension of the original editor implementation to use our Graphiti decorator. We found that, although with only few additional code lines and few small changes to the original implementation, decorating graphical elements did not function as intended or expected. One problem, for example, was that as soon as we changed some graphical properties of an element they were immediately overwritten with original values by the Graphiti runtime. We found that one way to prevent this from happening was by deactivating the so-called *Update Adapter* of a diagram behavior. At least that way it was possible for us to see the extension of the graphical concrete syntax in the graphical editor.

Another shortcoming of the framework is in supporting image decorations. Graphiti uses

so-called *Image Providers* with which all images that one wants to display in an editor needs to be registered at the editor initialization time and must physically reside in the plug-in where image provider is implemented. An editor gets one instance of an image provider through the means of an extension point that the Graphiti framework defines, however, dynamically adding additional set of images to the image provider of the editor at runtime, from other plug-ins, is not possible. The fact is that this is exactly the way how profiles contribute their additional set of images, when they are applied to a modeling language. Hence, the way how Graphiti handles images is just not compatible with a lightweight language extension mechanism introduced by EMF Profiles.

Consequently, this means that, at the moment, the only feasible way to extend the concrete graphical syntax, implemented with Graphiti framework, is by adapting the original code implementation to reflect the new information added to a modeling language, hence, the heavyweight language extension approach.

We would like to note that Graphiti is still in its incubation phase, as an Eclipse project, and new features and improvements are to be expected with future releases. The version we experimented with is 0.10.2.



# Conclusion and Future Work

## 8.1 Conclusion

This work describes an effort to address the extension of modeling languages, in particular the extension of their graphical concrete syntax (GCS).

We based our work on employing the following technologies:

- Eclipse Modeling Framework (EMF): is the framework that supports the definition and creation of new domain specific modeling languages (DSMLs).
- Graphical Modeling Framework (GMF) and Graphiti: are frameworks designed to support the specification of the graphical concrete syntax for a modeling language (created with EMF), and the creation of an editor implementation that supports it.
- EMF Profiles: is a lightweight language extension approach for extending EMF-based DSLs by a mechanism known as profiling.

Due to the fact that, until now, EMF Profiles supported model engineers only by providing an extension mechanism for the abstract syntax, and did not offer any means, except for simple icons, to extend the graphical concrete syntax, we took upon the task to broaden the realm of the EMF Profiles extension mechanism to include this missing part.

This was done by executing the following tasks:

- Specification and development of the decoration description language, which is a DSL with the textual concrete syntax designed for the description of graphical decorations that are visualized when the corresponding stereotype is applied to a model element. These decoration descriptions are designed to be independent of any specific underlying technologies that are used for the specification of the graphical concrete syntax and for building editors that support it. Furthermore, the language supports the specification of conditions that govern the activation state of the corresponding decorations.

- Adapting the EMF Profiles architecture to support the pluggable contributions to the framework by the provision of the dedicated *decorators* for specific graphical editor technologies. This means that developers only need to provide the contribution implementation for our Decorator extension point in order to enable the use of the EMF Profiles extension mechanism for modeling languages with GCS created in another yet unsupported graphical editor technology.
- In order to support dynamic (more reactive behavior) of the decoration descriptions in the editor, we had to revamp the EMF Profiles core implementation to support event-driven notifications that are fired based on model data changes.
- Implementation of the dedicated decorators for the GMF-based and Graphiti-based graphical modeling editors.

We evaluated the result of our work with a case study. The case study was designed to test the support of the dynamic behavior, and, of course, the extension of the GCS for our targeted graphical editor technologies, namely the Graphical Modeling Framework and the Graphiti framework. The case study was performed in the form of a model execution scenario.

Considering our requirements defined in Chapter 3, and the evaluation based on the case study, we can conclude that our aspiration to provide a general solution for the extension of the graphical concrete syntax of any GMF-based and Graphiti-based modeling editor was, in the case of the first, successfully accomplished, and for the latter, not possible. It is due to the GMF maturity and its excellent support for the extension that we were successful in providing a general solution here. In the case of the Graphiti framework the failure to provide the same was primarily due to the lack of the needed extensibility of the framework. So, Graphiti-based modeling editors, at the moment, can only be extended in a heavyweight manner, by adapting the original code implementation. However, since the Graphiti framework is still in its incubation phase this might change with future releases.

When it comes to the extent of the graphical concrete syntax extension itself, based on our experience, we can argue that as long as the GCS extension does not go beyond adornments or changes to the simple visualization properties of model elements the lightweight extension of the graphical concrete syntax, in its general application to any modeling language, is feasible. Desiring more extensive decoration facilities, such as changing the form of a shape and adapting its structure in order to better reflect the newly added information to a model element, proves to be a complicated task when considering a general solution for the GCS extension. In such a case, we might argue that this is the time when one must decide to consider the adaptation to the original editor implementation, hence, to perform the heavyweight language extension.

## 8.2 Future Work

During the writing of this thesis on the 26th of June, 2014, a new version of Eclipse, codename Luna, was released. With it also, a new technology, known under the name *Sirius*<sup>1</sup>, for the spec-

<sup>1</sup>Sirius project. <http://projects.eclipse.org/projects/modeling.sirius>

ification of the graphical concrete syntax for modeling languages. The project leverages Eclipse Modeling technologies, including EMF and GMF. The graphical concrete syntax is specified through configuration descriptions that are dynamically interpreted at runtime by the Eclipse IDE, so no code generation is involved. This is a big leap forward in creation of the graphical concrete syntax, because the user does not need to have comprehensive technical knowledge of the Eclipse, and the instant feedback support while specifying the syntax simplifies the creation tremendously. The standard Ecore editor was replaced with a new editor implementation created with Sirius. In order to support the extension of the graphical concrete syntax created with this new technology, a new decorator must be implemented. Since Sirius is based on GMF we anticipate that decorating models should be technically very similar to the GMF decorator implementation.

The decoration description language at the moment does not have a dedicated *syntax formatter*. An implementation for it could improve user-friendliness of the editor, thus, helping the user to format the textual specification of decoration descriptions most accordingly to the language specifics.

When specifying activation conditions for decorations, the current implementation of the decoration description language does not provide the syntax validation for OCL expressions, which could be improved. Also, the specification of a concrete condition could be improved by allowing a comparison between the tagged values of a stereotype, and extending the support for more data types. Moreover, if the OCL syntax gets good editor support, one might also decide to use only OCL expressions for specifying the activation conditions.

Adding animations to the set of decoration descriptions is also pursuable. At the moment this might be a complicated task since the underlying technology (e.g., GEF and Draw2D, which are based on SWT) does not provide dedicated API for animations, hence, the algorithms for the animations would have to be manually implemented. However, work is currently being invested to develop JavaFX-based core of GEF. When this is done, a new set of APIs, including those for animations, will be at disposal to technologies that are based on GEF, such as GMF and Graphiti.

The fact that, in our opinion, specifying decorations that are designed to change the form of the original shape presents a complicated task (keeping it in mind that a general solution is required), and that in such cases one should opt for heavyweight extension approach, nevertheless, this is just an opinion and not based on actual research. Thus, it needs to be properly assessed.





# Grammar of the Decoration Description Language

## A.1 The Grammar Specification in Xtext

Listing A.1: The Xtext Grammar of the Decoration Description Language.

```
1 grammar org.modelversioning.emfprofile.decoration.  
   EMFProfileDecorationLanguage with org.eclipse.xtext.common.Terminals  
2  
3 generate decorationLanguage "http://www.modelversioning.org/emfprofile/  
   decoration/EMFProfileDecorationLanguage"  
4 import "http://www.eclipse.org/emf/2002/Ecore" as ecore  
5 import "http://www.modelversioning.org/emfprofile/1.1" as profile  
6  
7 DecorationModel:  
8   "import resource" importURI = STRING  
9   (namespace=Namespace)?  
10  decorationDescriptions+=DecorationDescription*  
11 ;  
12  
13 Namespace:  
14   'profile' profile=[profile::Profile|QualifiedName]  
15 ;  
16  
17 DecorationDescription:  
18   'decoration' stereotype=[profile::Stereotype|QualifiedName] '{'  
19   (  
20     decorations+=(AbstractDecoration)+  
21     & (activation=Activation)?  
22   )  
23   '}'  
24 ;  
25
```

```

26 // In Decorations that have optional values some default value will be
    // used. See comments below.
27
28 AbstractDecoration:
29     ImageDecoration | BoxDecoration | ColorDecoration // has an effect on
        both nodes and edges
30     | BorderDecoration // has an effect only on nodes
31     | ConnectionDecoration // has an effect only on edges
32 ;
33
34 ImageDecoration:
35     {ImageDecoration}
36     'image' '{'
37     (
38         ('uri' '=' location_uri=STRING)
39         & (direction=Direction)? // default value for the node = NORTH_WEST,
            and for edge = CENTER
40         & (margin=Margin)? // default value -1 on nodes, 50 for connections
            (for connections margin is interpreted as a percentage)
41         & ('tooltip' '=' tooltip=Text)?
42         & (activation=Activation)?
43     )
44     '}'
45 ;
46
47 BoxDecoration :
48     {BoxDecoration}
49     'box' '{'
50     (
51         // mandatory
52         ('text' '=' text=Text)
53         & ('width' '=' width=INT)
54         & ('height' '=' height=INT)
55
56         // optional
57         & (image=BoxImage)?
58         & ('border' '{' border=Border '}')?
59         & ('foregroundColor' '=' foregroundColor=Color)?
60         & ('backgroundColor' '=' backgroundColor=Color)?
61         & (direction=Direction)? // default value for node = NORTH_WEST; for
            edge = CENTER
62         & (margin=Margin)? // default value -1 on nodes, 50 for connection (
            for connection margin is interpreted as percentage)
63         & ('contentDirection' '=' contentDirection=Directions)? // default
            is CENTER
64         & ('tooltip' '=' tooltip=Text)?
65         & (activation=Activation)?
66     )
67     '}'
68 ;
69
70 BorderDecoration:
71     {BorderDecoration}

```

```

72     'border' '{'
73         (
74             (border=Border)
75             & (activation=Activation)?
76         )
77     '}'
78 ;
79
80
81 ColorDecoration:
82     {ColorDecoration}
83     'color' '{'
84         (
85             ('background' '=' background=Color)?
86             & ('foreground' '=' foreground=Color)?
87             & (activation=Activation)?
88         )
89     '}'
90 ;
91
92 ConnectionDecoration:
93     {ConnectionDecoration}
94     'connection' '{'
95         (
96             (size=Size)?
97             & (style=Style)?
98             & ('color' '=' color=Color)?
99             & (activation=Activation)?
100        )
101    '}'
102 ;
103
104 /**
105  * The text which we can include in the visualization, e.g., the hover
106  * text over an image decoration.
107  * It can be build out of Strings and tagged values of the applied
108  * stereotype.
109  */
110
111 Text:
112     SimpleText | ComplexText
113 ;
114
115 SimpleText:
116     text=STRING | attribute=[ecore::EAttribute|QualifiedName]
117 ;
118
119 ComplexText:
120     left=SimpleText '+' right=Text
121 ;
122
123 Border:
124     {Border}
125     ( (size=Size)? // default value is 1

```

```

123     & ('color' '=' color=Color)? // default value is BLACK
124     & (style=Style)? // default value is SOLID
125     )
126 ;
127
128 BoxImage :
129     {BoxImage}
130     'image' '{'
131     (
132         ('uri' '=' location_uri=STRING)
133         & ('placement' '=' placement = BoxImageOrientation)? // default
134             value is WEST
135     )
136     '}'
137 ;
138 Style:
139     'lineStyle' '=' value=LineStyle
140 ;
141
142 Size:
143     'size' '=' value=INT
144 ;
145
146 Direction:
147     'direction' '=' value=Directions
148 ;
149
150 Margin:
151     'margin' '=' value=SignedInteger
152 ;
153
154 Color:
155     {Color}
156     value=ColorConstant | concrete = ConcreteColor
157 ;
158
159 ConcreteColor:
160     RGB | HexColor
161 ;
162
163 RGB:
164     'RGB' '(' red=INT ',' green=INT ',' blue=INT ')'
165 ;
166
167 HexColor:
168     hexCode = HEX_COLOR
169 ;
170
171 ColorConstant:
172     value=Colors
173 ;
174

```

```

175 Activation:
176   'active when' condition=AbstractCondition
177 ;
178
179 AbstractCondition:
180   Condition | CompositeCondition | OclExpression
181 ;
182
183 OclExpression:
184   'ocl' '(' expression = STRING ')'
185 ;
186
187 Condition:
188   attribute=[ecore::EAttribute|QualifiedName] operator=ComparisonOperator
189   value=Type
190 ;
191
192 CompositeCondition:
193   operator=LogicalOperator '(' conditions += (AbstractCondition )+ ')'
194 ;
195
196 Type:
197   SignedInteger | SignedDouble | STRING | BOOLEAN | ID
198 ;
199
200 SignedDouble returns ecore::EDouble :
201   '-'? INT '.' INT
202 ;
203
204 SignedInteger returns ecore::EInt:
205   '-'? INT
206 ;
207
208 QualifiedName returns ecore::EString:
209   ID ('.' ID)*
210 ;
211
212 terminal BOOLEAN returns ecore::EBoolean:
213   'true' | 'false'
214 ;
215
216 terminal HEX_COLOR:
217   '#'
218   (('a'..'f'|'A'..'F'|'0'..'9') ('a'..'f'|'A'..'F'|'0'..'9') ('a'..'f'|'A'
219     ..'F'|'0'..'9'))?
220   ('a'..'f'|'A'..'F'|'0'..'9') ('a'..'f'|'A'..'F'|'0'..'9') ('a'..'f'|'A'
221     ..'F'|'0'..'9')
222 ;
223
224 enum BoxImageOrientation:
225   WEST | NORTH | EAST | SOUTH
226 ;

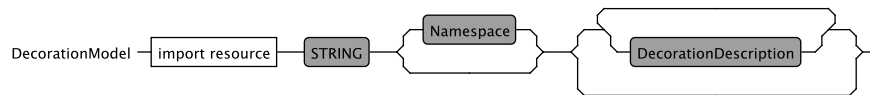
```

```

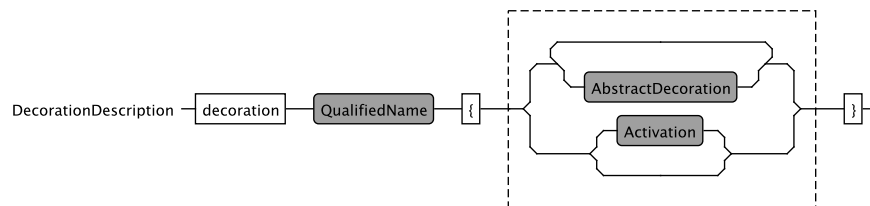
225 enum ComparisonOperator:
226     EQUAL='==' | UNEQUAL!='' | GREATER='>' |
227     GREATEROREQUAL='>=' | LOWER='<' | LOWEROREQUAL='<='
228 ;
229
230 enum LogicalOperator:
231     ALL | ANY
232 ;
233
234 enum LineStyle:
235     LINE_SOLID = 'SOLID' | LINE_DOT = 'DOTS' |
236     LINE_DASH = 'DASH' | LINE_DASHDOT = 'DASHDOT' |
237     LINE_DASHDOTDOT = 'DASHDOTDOT'
238 ;
239
240 enum Colors:
241     RED | BLACK | WHITE | GREEN | GREEN_LIGHT | GREEN_DARK |
242     BLUE | BLUE_LIGHT | BLUE_DARK | GRAY | GRAY_LIGHT |
243     GRAY_DARK | ORANGE | YELLOW | CYAN
244 ;
245
246 enum Directions:
247     CENTER | NORTH | SOUTH | WEST | EAST | NORTH_EAST |
248     NORTH_WEST | SOUTH_EAST | SOUTH_WEST
249 ;

```

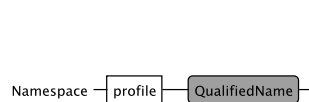
## A.2 Railroad Visualization of the Grammar Syntax



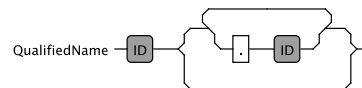
*DecorationModel rule (lines 7 – 11)*



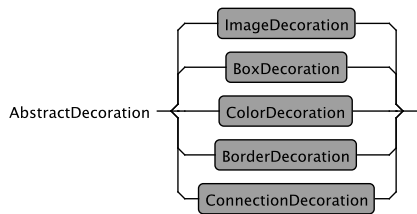
*DecorationDescription rule (lines 17 – 24)*



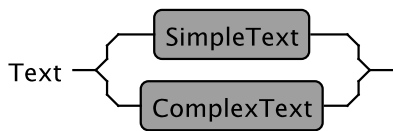
*Namespace rule (lines 13 – 15)*



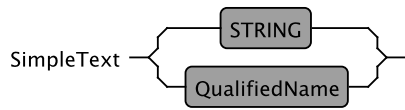
*QualifiedName rule (lines 207 – 209)*



*AbstractDecoration rule (lines 28 – 32)*



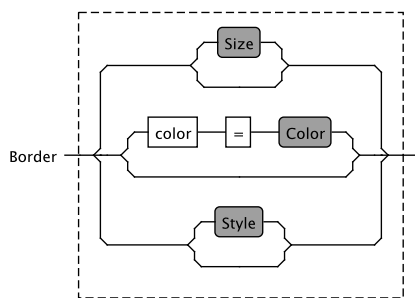
*Text rule (lines 108 – 110)*



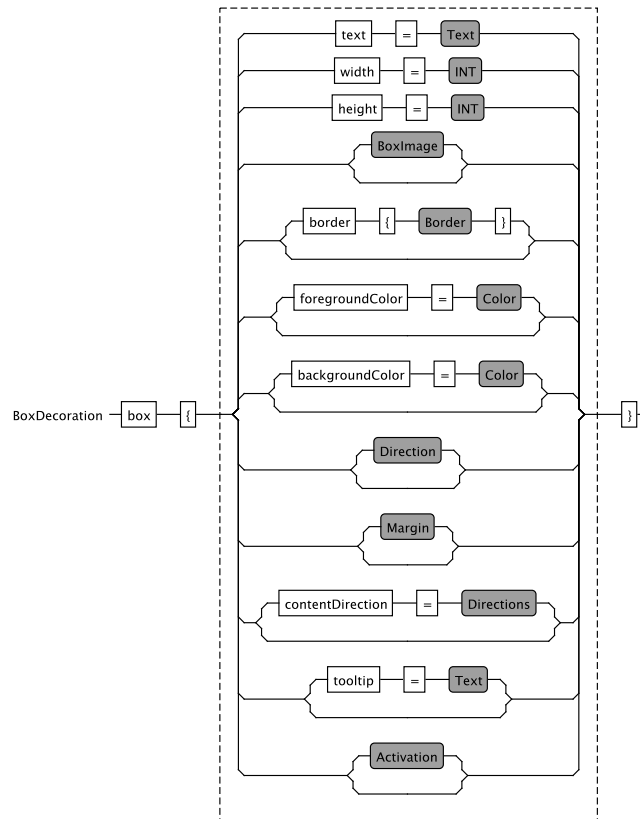
*SimpleText rule (lines 112 – 114)*



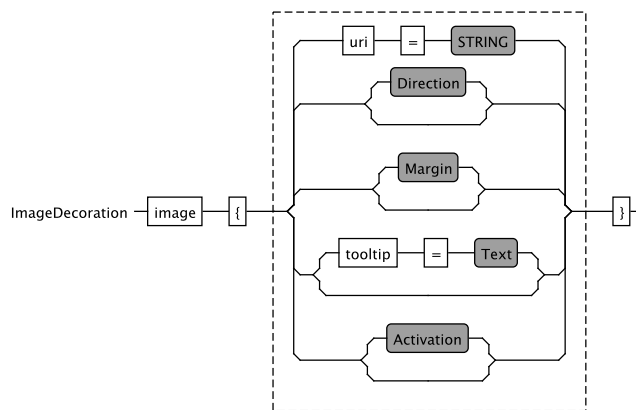
*ComplexText rule (lines 116 – 118)*



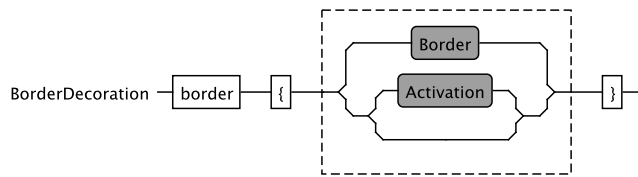
*Border rule (lines 120 – 126)*



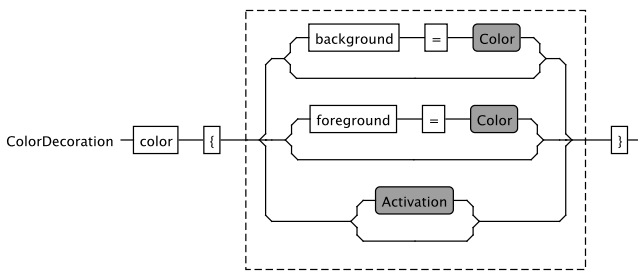
*BoxDecoration rule (lines 47 – 68)*



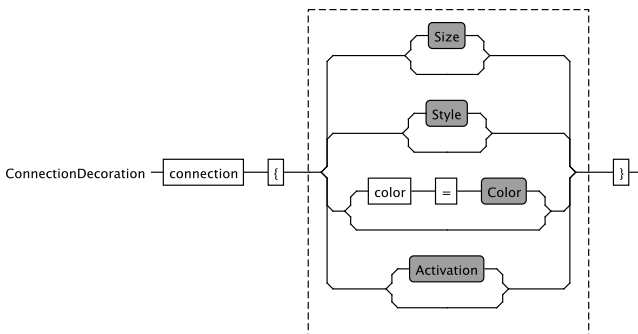
*ImageDecoration rule (lines 34 – 45)*



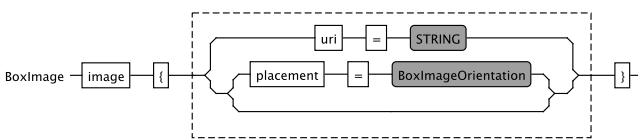
*BorderDecoration rule (lines 70 – 78)*



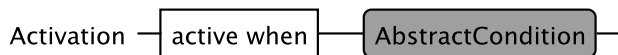
*ColorDecoration rule (lines 81 – 90)*



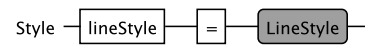
*ConnectionDecoration rule (lines 92 – 102)*



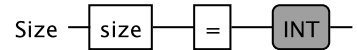
*BoxImage rule (lines 128 – 136)*



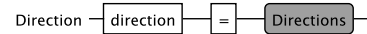
*Activation rule (lines 175 – 177)*



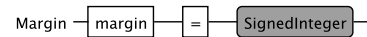
*Style rule (lines 138 – 140)*



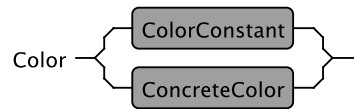
*Size rule (lines 142 – 144)*



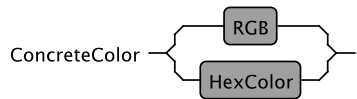
*Direction rule (lines 146 – 148)*



*Margin rule (lines 150 – 152)*



*Color rule (lines 154 – 157)*



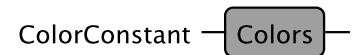
*ConcreteColor rule (lines 159 – 161)*



*RGB rule (lines 163 – 165)*

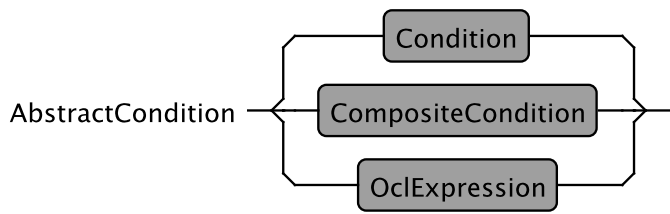


*HexColor rule (lines 167 – 169)*

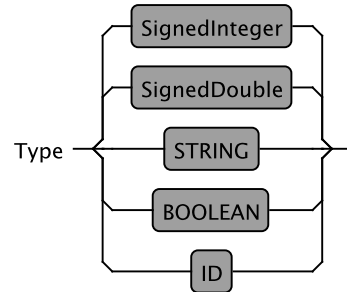


*ColorConstant rule (lines 171 – 173)*

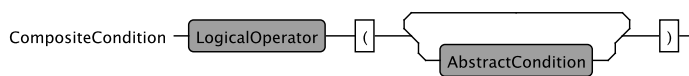




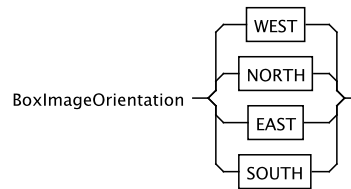
*AbstractCondition rule (lines 179 – 181)*



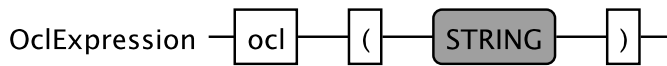
*Type rule (lines 195 – 197)*



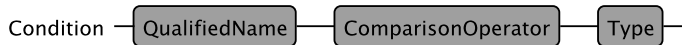
*CompositeCondition rule (lines 191 – 193)*



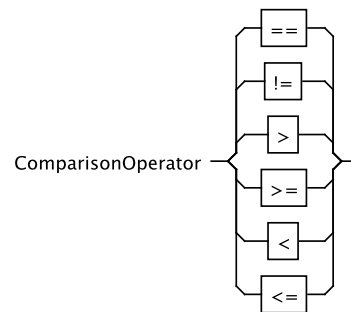
*BoxImageOrientation rule (lines 221 – 223)*



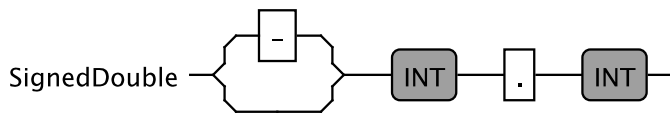
*OclExpression rule (lines 183 – 185)*



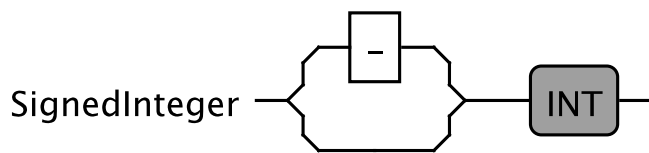
*Condition rule (lines 187 – 189)*



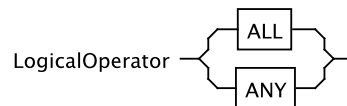
*ComparisonOperator rule (lines 225 – 228)*



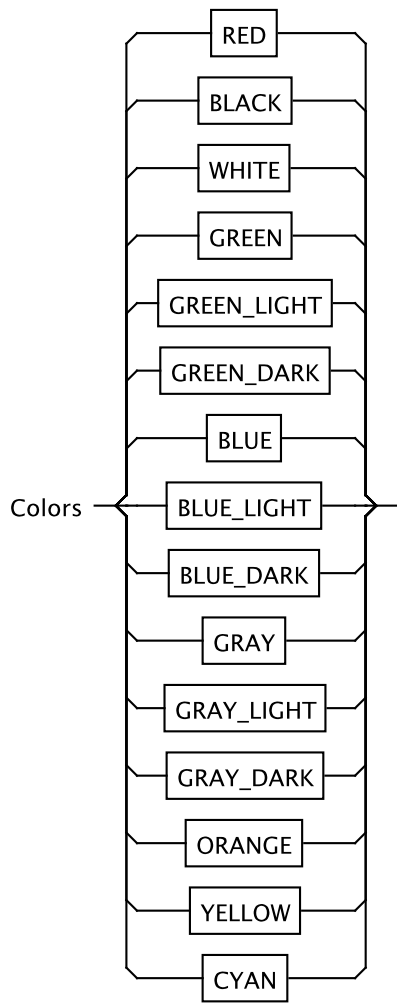
*SignedDouble rule (lines 199 – 201)*



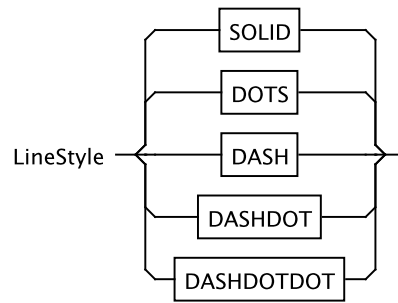
*SignedInteger rule (lines 203 – 205)*



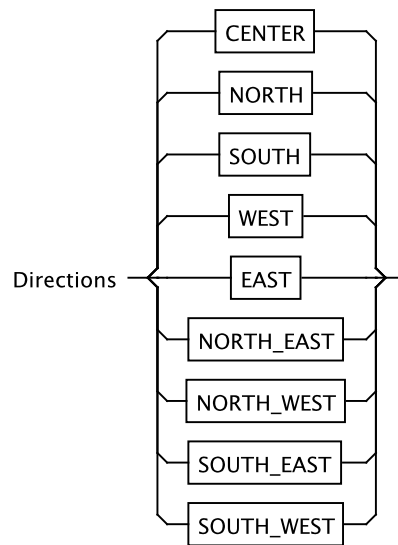
*LogicalOperator rule (lines 230 – 232)*



*Colors rule (lines 240 – 244)*



*LineStyle rule (lines 234 – 238)*



*Directions rule (lines 246 – 249)*

# Bibliography

- [1] Spray. Project website: <http://code.google.com/a/eclipselabs.org/p/spray/> [Online, accessed: 2014-09-09].
- [2] Roberto Acerbis, Aldo Bongio, Marco Brambilla, Massimo Tisi, Stefano Ceri, and Emanuele Tosetti. Developing eBusiness solutions with a model driven approach: The case of Acer EMEA. In *Web Engineering, 7th International Conference, ICWE 2007, Como, Italy, July 16-20, 2007, Proceedings*, volume 4607 of *Lecture Notes in Computer Science*, pages 539–544, 2007.
- [3] Margarida Afonso, Regis Vogel, and Jose Teixeira. From Code Centric to Model Centric Software Engineering: Practical case study of MDD infusion in a Systems Integration Company. In *Model-Based Development of Computer-Based Systems and Model-Based Methodologies for Pervasive and Embedded Software, 2006. MBD/MOMPES 2006. Fourth and Third International Workshop on*, pages 10–pp, 2006.
- [4] Colin Atkinson and Thomas Kühne. The Role of Metamodeling in MDA. In *Proceedings of the International Workshop in Software Model Engineering @ UML'02*, 2002.
- [5] Colin Atkinson, Thomas Kühne, and Brian Henderson-Sellers. Systematic Stereotype Usage. *Software and Systems Modeling*, 2(3):153–163, 2003.
- [6] Jean Bézivin. On the unification power of models. *Software & Systems Modeling*, 4(2):171–188, 2005.
- [7] Jean Bézivin and Olivier Gerbé. Towards a Precise Definition of the OMG/MDA Framework. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE'01)*, pages 273–280, 2001.
- [8] Alexander Bohn. Towards an Understanding of the Practical Use of UML. Master's thesis, Faculty of Informatics at the Vienna University of Technology, 2013.
- [9] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, 2012.
- [10] Steve Burbeck. Applications programming in Smalltalk-80 (tm): How to use Model-View-Controller (MVC). *Smalltalk-80 v2*, 5, 1992.

- [11] Peter Pin-Shan Chen. The Entity-Relationship Model—Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, March 1976.
- [12] Tony Clark, Paul Sammut, and James Willans. *Applied Metamodelling, a Foundation for Language Driven Development*, volume Second Edition. 2008.
- [13] Valeria De Castro, Esperanza Marcos, and Juan Manuel Vara. Applying cim-to-pim model transformations for the service-oriented development of information systems. *Information and Software Technology*, 53(1):87–105, 2011.
- [14] Eclipse Foundation. EMF Facet. Project website, <http://www.eclipse.org/facet/> [Online, accessed: 2014-09-09].
- [15] Eclipse Foundation. Graphical Modeling Project (GMP). Project website: <http://www.eclipse.org/modeling/gmp/> [Online, accessed: 2014-09-09].
- [16] Eclipse Magazine. Graphiti - Development of High-Quality Graphical Model Editors. Online article: <http://www.eclipse.org/graphiti/documentation/files/EclipseMagazineGraphiti.pdf> [Online, accessed: 2014-09-09].
- [17] Sven Efftinge and Markus Völter. oAW xText: A framework for textual DSLs. In *Workshop on Modeling Symposium at Eclipse Summit*, volume 32, page 118, 2006.
- [18] Moritz Eysholdt and Heiko Behrens. Xtext: implement your language faster than the quick and dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 307–309, 2010.
- [19] Mathias Fritzsche, Wasif Gilani, Michael Thiele, Ivor T. A. Spence, T. John Brown, and Peter Kilpatrick. A scalable approach to annotate arbitrary modelling languages. In *Modellierung 2010, 24.-26. März 2010, Klagenfurt, Österreich*, volume 161, pages 301–318, 2010.
- [20] Lidia Fuentes-Fernández and Antonio Vallecillo-Moreno. An introduction to UML profiles. *UML and Model Engineering*, 2, 2004.
- [21] Richard C Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Pearson Education, 2009.
- [22] Object Management Group. OMG Diagram Interchange (DI) specification, Version 1.0, 2006. <http://www.omg.org/cgi-bin/doc?formal/06-04-04> [Online, accessed: 2014-09-09].
- [23] Object Management Group. UML Specification: OMG Unified Modeling Language (OMG UML), Superstructure Version 2.4.1, 2011. <http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF> [Online, accessed: 2014-09-09].

- [24] Object Management Group. OMG Diagram Definition (DD), Version 1.0, 2012. <http://www.omg.org/spec/DD/> [Online, accessed: 2014-09-09].
- [25] Object Management Group. MOF Specification: OMG Meta Object Facility (OMG MOF), Version 2.4.2, 2014. <http://www.omg.org/spec/MOF/2.4.2/> [Online, accessed: 2014-09-09].
- [26] Markus Herrmannsdoerfer, Daniel Ratiu, and Guido Wachsmuth. Language Evolution in Practice: The History of GMF. In *Software Language Engineering*, pages 3–22. Springer, 2010.
- [27] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. Design science in information systems research. *MIS Quarterly*, 28(1):75–105, March 2004.
- [28] M. Hitz, G. Kappel, E. Kapsammer, and W. Retschitzegger. *UML @ Work, Objektorientierte Modellierung mit UML 2*. dpunkt.verlag, 3. edition, 2005 (in German).
- [29] Melanie Kapellner. How to Efficiently Collaborate in Model Versioning. A Guideline to Reduce and Resolve Conflicts. Master’s thesis, Faculty of Informatics at the Vienna University of Technology, 2010.
- [30] Milan Karow, Andreas Gehlert, Jörg Becker, and Werner Esswein. On the Transition from Computation Independent to Platform Independent Models. In *Proceedings of 12th Americas Conference on Information Systems (AMCIS’06)*, page 469, 2006.
- [31] Stuart Kent. Model Driven Engineering. In *Integrated Formal Methods*, pages 286–298, 2002.
- [32] Samir Kherraf, Éric Lefebvre, and Witold Suryn. Transformation from cim to pim using patterns and archetypes. In *Software Engineering, 2008. ASWEC 2008. 19th Australian Conference on*, pages 338–346, 2008.
- [33] Anneke Kleppe. A Language Description is More than a Metamodel. In *Proceedings of the 4th International Workshop on Software Language Engineering (SLE’07)*, 2007.
- [34] Anneke Kleppe. The Field of Software Language Engineering. In *Software Language Engineering*, pages 1–7. Springer, 2009.
- [35] Anneke G Kleppe, Jos B Warmer, and Wim Bast. *MDA explained: the model driven architecture: practice and promise*. Addison-Wesley Professional, 2003.
- [36] Dimitrios S. Kolovos, Louis M. Rose, Saad bin Abid, Richard F. Paige, Fiona A. C. Polack, and Goetz Botterweck. Taming EMF and GMF using model transformation. In *Model Driven Engineering Languages and Systems - 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010, Proceedings, Part I*, volume 6394 of *Lecture Notes in Computer Science*, pages 211–225. Springer, 2010.

- [37] Dimitrios S. Kolovos, Louis M. Rose, Nikolaos Drivalos Matragkas, Richard F. Paige, Fiona A. C. Polack, and Kiran Jude Fernandes. Constructing and navigating non-invasive model decorations. In *Theory and Practice of Model Transformations, Third International Conference, ICMT 2010, Malaga, Spain, June 28-July 2, 2010. Proceedings*, volume 6142 of *Lecture Notes in Computer Science*, pages 138–152, 2010.
- [38] Thomas Kühne. Matters of (meta-) modeling. *Software & Systems Modeling*, 5(4):369–385, 2006.
- [39] Philip Langer, Konrad Wieland, Manuel Wimmer, and Jordi Cabot. EMF Profiles: A Lightweight Extension Approach for EMF Models. *Journal of Object Technology*, 11(1):1–29, 2012.
- [40] Tom Mens and Pieter Van Gorp. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006.
- [41] Tadao Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.
- [42] OMG. Model Driven Architecture Guide: OMG MDA Guide Verision 1.0.1, 2003. <http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF> [Online, accessed: 2014-09-09].
- [43] Richard E Pattis. EBNF: A notation to describe syntax. *While developing a manuscript for a textbook on the Ada programming language in the late 1980s, I wrote a chapter on EBNF*, 1980.
- [44] Andrea Randak. ATL4pros: Introducing Native UML Profile Support into the ATLAS Transformation Language. Master’s thesis, Faculty of Informatics at the Vienna University of Technology, 2011.
- [45] Dan Rubel. *The Eclipse Graphical Editing Framework (GEF)*. Addison-Wesley Professional, 2011.
- [46] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The*. Pearson Higher Education, 2004.
- [47] Ryan Stansifer. EBNF Grammar for Mini-Java. [http://cs.fit.edu/~ryan/cse4251/mini\\_java\\_grammar.html](http://cs.fit.edu/~ryan/cse4251/mini_java_grammar.html) [Online, accessed: 2014-09-09].
- [48] Bran Selic. A Systematic Approach to Domain-Specific Language Design Using UML. In *Object and Component-Oriented Real-Time Distributed Computing, 2007. ISORC’07.*, pages 2–9, 2007.
- [49] Richard Soley et al. Model Driven Architecture. *OMG white paper*, 308:308, 2000.
- [50] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: Eclipse Modeling Framework*. Pearson Education, 2008.

- [51] Suzy Temate, Laurent Broto, Alain Tchana, and Daniel Hagimont. A high level approach for generating model's graphical editors. In *Information Technology New Generations (ITNG)*, pages 743–749. IEEE, 2011.
- [52] Jos B Warmer and Anneke G Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Professional, 2003.
- [53] Wei Zhang, Hong Mei, Haiyan Zhao, and Jie Yang. Transformation from CIM to PIM: A feature-oriented component-based approach. In *Model Driven Engineering Languages and Systems*, pages 248–263. Springer, 2005.