Why and How We Should Use Graphiti to Implement PCM Editors*

Christian Stritzke, Sebastian Lehrig

University of Paderborn
Zukunftsmeile 1
33102 Paderborn
cstritzk@mail.upb.de
sebastian.lehrig@upb.de

Abstract: For the Palladio Component Model (PCM), performance engineers use graphical editors for specifying and composing components as well as usage scenarios and deployment environments. These editors are created with the Graphical Modeling Framework (GMF). The fact that GMF editors are widely considered as difficult to maintain and to customize raises the question whether it makes sense to migrate to a more modern framework, namely Graphiti. Currently, related work does not evaluate the applicability of Graphiti for PCM editors. Therefore, we analyze the advantages and disadvantages of Graphiti and GMF in the context of Palladio. Based on our findings, we suggest to use Graphiti for the further development of PCM editors. Furthermore, we suggest how these editors can be implemented based on Graphiti.

1 Introduction

Developing maintainable, reusable, and easy-to-understand graphical editors for a large meta-model like the PCM poses a great challenge to software developers. In order to meet this challenge, developers must choose appropriate editor frameworks.

For example, the Palladio-Bench comes with a variety of editors. This includes a repository editor with which components can be specified. In the composite structure editor, these components can be assembled to systems. Also, there is are editors for Service Effect Specifications (SEFF), resource environment specifications, resource allocations, and usage scenarios. All of these editors are currently implemented based on the *Graphical Modeling Framework (GMF)* [ST06]. Due to a mixture of model-driven code generation and manual adjustments as well as a generally complicated architecture of the GMF runtime, the aforementioned requirements are difficult to meet. Therefore, there is the need to analyze whether more appropriate editor frameworks exist.

A prominent alternative to GMF is the editor framework *Graphiti* [BGK+11]. Refsdal

*The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant no 317704 (CloudScale).

Proc. Kieker/Palladio Days 2013, Nov. 27–29, Karlsruhe, Germany
Available online: http://ceur-ws.org/Vol-1083/

compares GMF and Graphiti using a reference editor he implemented with both technologies [Ref11]. However, Graphiti has not been investigated in the Palladio context yet. Related work investigating editors for Palladio, compares the manual development of an editor and an equivalent GMF-based approach [KB07]. However, they lack the consideration of the Graphiti framework, thus, posing the question whether Graphiti could be a more appropriate framework.

Therefore, we evaluate whether a reimplementation of the PCM editors is rewarding. For this, we compare GMF and Graphiti regarding the existing PCM editors. For our comparison, we formulate the following requirements for an appropriate editor framework (taken from the ISO/IEC 25010 [ISO10]):

Functional Completeness: Does the editor framework provide all the necessary features for implementing the Palladio editors?

Learnability: Is the editor framework easy to learn?

Modifiability: Does the editor framework's architecture allow an easy addition and modification of features?

Reusability: Can already implemented editors be reused and extended?

Comfort: Is it possible to develop a comfortable editor with the editor framework?

The contribution of this paper is the comparison of the GMF and Graphiti frameworks for PCM editors regarding our requirements. We compare the two options based on the existing GMF editor implementation and prototyped Graphiti editors for the PCM. We come to the conclusion that Graphiti is in fact superior concerning most of the requirements. After that, we additionally suggest the architecture of Graphiti-based PCM editors based on our prototyped Graphity editors.

In the remainder of this paper, we first give an overview of the two editor frameworks in question (Section 2). After that, we compare and discuss the frameworks in Section 3. The architecture of PCM editors based on Graphiti is described in Section 2.2. Finally, we present related work (Section 5) and draw conclusions (Section 6).

2 Eclipse Editor Frameworks

Here, we give a brief overview of the editor frameworks *GMF* and *Graphiti*, describing their fundamental architectural concepts. Also, we describe *Spray*, a model-driven approach for generating *Graphiti* editors. We do not consider editor development, e.g. directly using the Eclipse editor APIs GEF and Draw2D, because the high development time and relatively low maintainability [KB07] rules this option out.

2.1 GMF

GMF is a model-driven approach to developing editors. It consists of a runtime and a tooling environment. GMF tooling enables the developer to specify editors in the form of EMF models. From these models, GMF generates ready-to-use editor code based on GEF and Draw2D.

Aside from a domain meta-model, developers have to specify three models in order to provide the necessary data for code generation: a graphical definition to specify figures, compartments, and connections, a tooling definition to specify a palette and how the elements of a diagram are handled, and a mapping between diagram elements and domain model elements. For each of these models, GMF provides a wizard to generate an initial model.

If developers need the editor to have custom features, they can implement it in the generated code using the flag `@generated not`. Every code element marked with that flag will then be untouched during the next generation. Alternatively, developers can set up a new plug-in and implement custom features there or modify the code generator templates. The customizations can be made on the level of GEF and Draw2D using the GMF Runtime framework.

2.2 Graphiti

Graphiti is a framework for developing graphical editors. Like GMF, it is based on the Graphical Editing Framework (GEF) and Draw2D. However, programmers using Graphiti do not access these APIs directly since the Graphiti API completely hides them. Contrary to GMF, Graphiti only provides a Java API against which developers program while GMF follows a model-driven approach.

The static structure of the Graphiti architecture is depicted in Figure 1. Users interact with a Graphiti editor by using the *Interaction Component* to manipulate objects and having them displayed by the *Rendering Engine*.

The *Diagram Type Agent* consists of the developer-written code and is in charge of creating elements of the *Pictogram Model* that provide a graphical representation of elements of the *Domain Model*. The *Diagram Type Agent* also provides the means to create new elements of the *Domain Model* graphically and to link pictogram elements to domain objects. In the following, we will have a closer look at the *Diagram Type Agent*.

As shown on the left-hand side of Figure 2, the *Diagram Type Agent* contains a *Diagram Type Provider* for each diagram to be implemented. Each *Diagram Type Provider* references to a *Feature Provider* that creates instances of the Diagram's Features.

For each domain object, The most important types of features are the following:

Add: Specifies what happens when a new pictogram element is created. This creation usually contains drawing the pictogram element and linking it to the domain object.

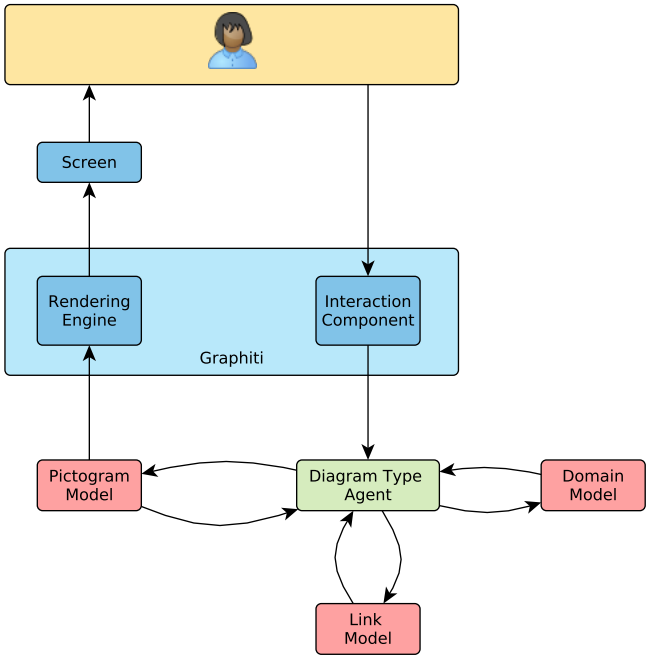

Figure 1: Static structure of Graphiti (taken from `help.eclipse.org`)

Create: Specifies what happens when a new domain object is created using the editor's palette. Usually that means creating a domain object and a pictogram element.

Delete: Specifies what happens when a domain object or a pictogram element is deleted.

Layout: Specifies how a pictogram element behaves when it is resized or moved.

Custom: Adds custom features to call in the context menu of a pictogram element.

Implementing features for each domain object can be tedious and involve a lot of classes to implement. Also it is difficult to reuse features for other domain objects using inheritance. In order to cope with that issue, newer versions of Graphiti introduce the *Pattern* concept (see Figure 2, right-hand side).

Here, instead of implementing a class for each feature, developers can implement the standard features (*add, create, delete, etc.*) in one class and custom features in extra classes, making inheritance a lot easier. That way, each class of a meta-model, including abstract classes, that is relevant to the graphical representation in an editor can be represented by one pattern class in the Graphiti code.

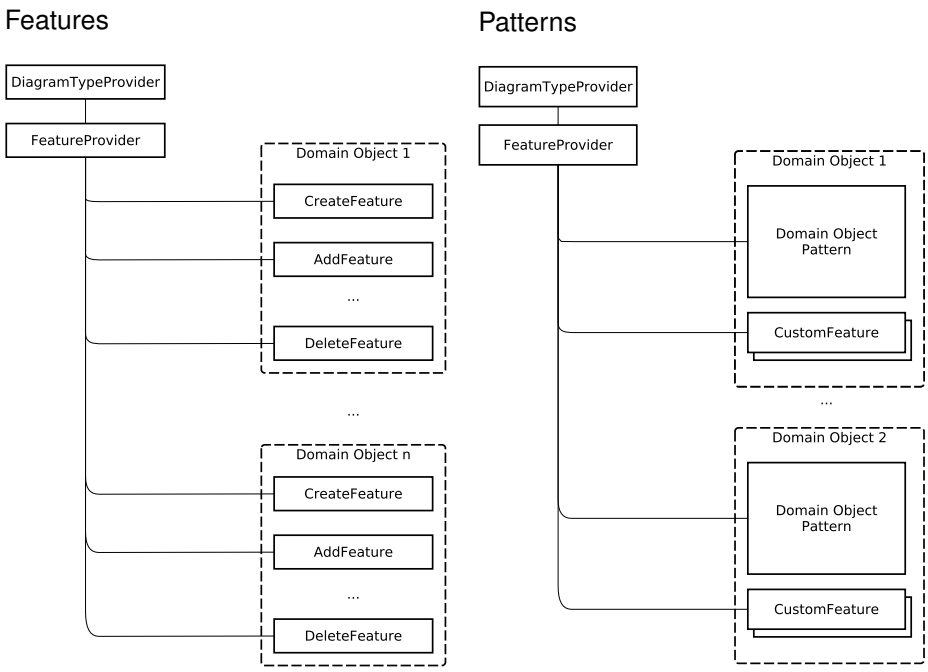

Figure 2: Features and patterns in Graphiti (taken from `help.eclipse.org`)

2.3 Spray

Spray is a generative approach to create Graphiti editors (sketched in Figure 3). It provides textual DSLs in which developers can specify pictogram elements like shapes and connections (Shape DSL), overall diagram styles and styles for specific elements (Style DSL), and the relationship between domain models and pictogram elements (Spray Core DSL).

From the specifications in the respective DSLs, Spray generates the code necessary to implement a Graphiti editor. This includes XML-files for registering Eclipse plug-ins, file and project properties, and Java classes. These Java classes implement the editor in the form of features rather than patterns.

The DSLs of Spray themselves are simple and easy-to-use, yet they lack the flexibility to create more complex graphical representations. For example, it is impossible to place a component symbol in the upper right corner of a component using only the shapes DSL. Also, reuse concepts such as shape inheritance are unsupported.

However, it is possible to overwrite the generated feature classes in order to add custom code. This allows to create more complex graphical representations of objects, however, the issue of reuse remains.

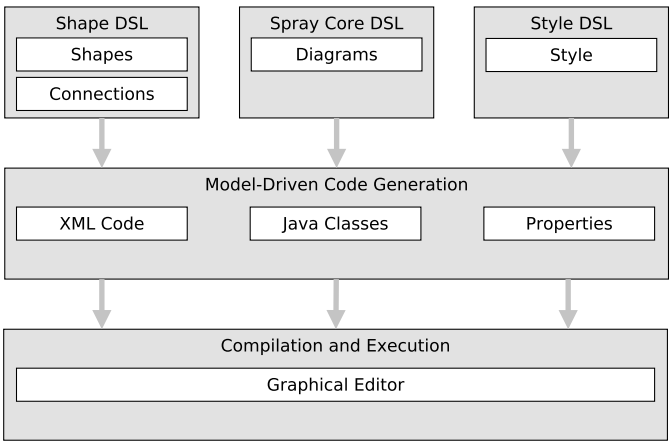

Figure 3: Artifacts and generation process in Spray.

3 Comparison of Eclipse Editor Frameworks

In this section, we argue on the advantages and disadvantages of GMF and Graphiti regarding PCM editors and the requirements we named in the introduction in Section 1. We base our argumentation on the findings of Refsdal [Ref11] who evaluated the two technologies using a reference editor implemented in both ways. Also, we have implemented a prototypical Graphiti-based composite structure editor in order to test the capabilities of Graphiti[1].

Functional Completeness: In the Graphiti API, the full functionality of GEF/Draw2D is hidden behind an abstraction layer. Thus, editor customization down to the last detail will be impossible as opposed to GMF.

Because the GMF Runtime allows developers to manipulate editors on the GEF/Draw2D level, this framework can be considered as more flexible than Graphiti.

Since the Palladio editors have not been implemented with Graphiti yet, we cannot make an exact statement about the functional completeness of Graphiti for Palladio. However, our prototyped editor of the composite structure editor provides all the necessary features for our needs, thus, indicating that Graphiti is functional complete for our needs.

Modifiability: If a GMF editor is solely generated from the models via GMF Tooling, it would be more maintainable than an equivalent Graphiti implementation because only the GMF-related models are modified. Regarding Palladio editors of which the GMF implementation is highly customized using GMF Runtime, Graphiti provides a higher maintainability. The reason for this is, according to Refsdal, the simplicity of Graphiti's architecture.

[1]The implementation can be accessed via `https://svnserver.informatik.kit.edu/i43/svn/code/Palladio/Incubation/GraphitiEditors` (User: anonymous; Password: anonymous; Visited on 14/11/2013).

Graphiti's architecture makes extending, understanding and modifying an editor implementation easier than a GMF Runtime-based equivalent.

Learnability: Again, we have to differentiate between non-customized editors created only by GMF Tooling and customized editors. Developers can generate basic GMF-based editors with many ready-to-use features in little time without having to look into an API description. For that, a well-written tutorial is available. Any customization, however, requires knowledge of the GMF Runtime API that, unfortunately, is poorly documented and rather complicated [Ref11].

With Graphiti, developing a basic editor takes much more time. However, customizations are easier, because its API and architecture are far more easy to understand. Also, there is a tutorial [SAP12] that covers many aspects of the framework as well as an example project that can be used as a starting point for development.

Reusability: With the pattern-based approach, Graphiti provides the means to develop extensible and reusable diagram elements in order to quickly develop editors based on already existing editors.

Comfort: Developers will be able to develop similarly comfortable editors with either framework. However, they have to put their effort into resolving different issues. For example, it is easier to create a consistent and appealing visual style for editor elements in Graphiti. On the other hand, GMF handles automatic layouting of graphical elements better.

Concluding, we can consider Graphiti as a better alternative to GMF. Although we expect it to take a long time to get Graphiti-based editors up and running (experience from the prototypical implementation), they are far more maintainable, compared to GMF, once they are initially set up. Due to its rather simplistic architecture, Graphiti-based editor code is easier to read for new developers as well.

4 Palladio Editors in Graphiti

After evaluating the advantages and disadvantages of the different approaches of implementing Graphiti editors (Section 3), we come to the conclusion that a manual implementation using patterns is the best solution for implementing editors for the PCM. It is most convenient being able to implement one pattern class for each PCM class that should be represented in an editor such that the patterns have the same inheritance structure as their corresponding PCM classes.

In the following, we name the classes implementing the concepts from Figure 2 (right). We first name the Diagram Type Providers and the Feature Providers for each Palladio editor which are directly connected to the plug-in framework. We then zoom in on one particular Feature Provider and show an example pattern structure. Afterwards, we give an example on how a pictogram element is composed.

At first, we implement one *Diagram Type Provider* class for each PCM editor. These classes inherit from the abstract class *AbstractDiagramTypeProvider* and are registered as

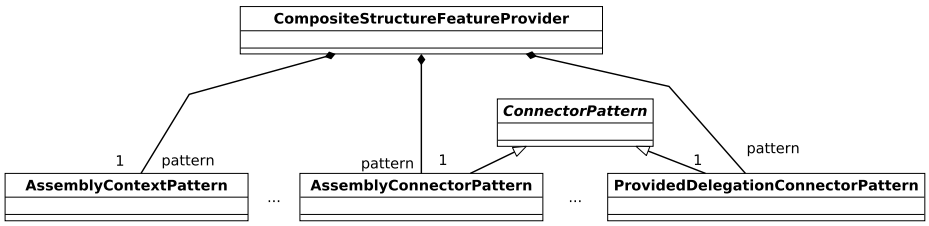

Figure 4: Patterns for some of the PCM Classes and their corresponding Classes of the PCM

diagram types in the *plugin.xml* file.

Each of the Diagram Type Providers contain a respective Feature Provider that inherits from the class *DefaultFeatureProviderWithPatterns* provided by Graphiti. Each of these Feature Providers contains a couple of patterns. One of these Feature Providers and its pattern classes are depicted in Figure 4.

In this example, we show a feature provider for the elements of Composite Structure diagrams. As this is a pattern-based Feature Provider, it is composed of several pattern classes, each named after the PCM class they represent. For example, the class *AssemblyContextPattern* contains methods to draw a component and its provided and required interfaces when an *AssemblyContext* is added to the diagram. Also, it invokes a prompt to choose a component from the repository when creating a new *AssemblyContext*. Other methods layout the internal elements of a component figure when resized, and many more.

Now, we have a closer look at the graphical representation of an Assembly Context. This representation consists of a composition of *Shape* Objects. In order to specify how a shape is drawn, each shape object refers to a graphics algorithm (*Ellipse, Rectangle*, etc.). For graphic algorithms, we can set parameters like position, size, rendering style, etc. As an example, we take the graphical representation of an instance of the class *AssemblyContext* which has one provided operation interface.

In the example, the outermost shape is realized by the object *ContainerShape*. Its graphical algorithm draws an invisible rectangle, meaning that the shape merely sets the borders and positions of the inlying shapes, but is invisible itself.

The inlying objects are several *Shape* objects with different kinds of graphics algorithms. For the component itself, we create a rectangle (*AssemblyComponentShape*). For each provided role of the component, we draw an ellipse (*ProvidedRoleShape*) and connect it with the component shape with a connector object (*ProvidedRoleConnector*). The text inside the component (*TextShape*) as well as the component symbol (three instances of *SymbolShape_n*) are represented by a shape object that has the shape object representing the component set as its parent shape.

This section should give a basic overview on how a pattern-based PCM editor is implemented. However, there are many more features of a Graphiti editor such as property views and look-and-feel specifications that we do not refer to here, because they are basically implemented the same way in every approach.

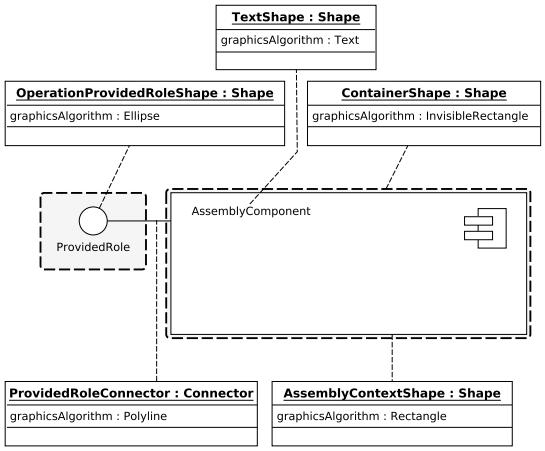

Figure 5: Classes implementing a graphical representation for an assembly component

5 Related Work

PCM editors. Krogmann and Becker evaluate two different approaches of PCM editor development [KB07]. On the one hand, there is a manually implemented PCM editor based on the .NET framework and the free graphical library Netron. On the other hand, there is a PCM editor with GMF. Both editors are implemented in the course of student projects. Krogmann and Becker conclude that using GMF results in dramatically less developing time and a better maintainability. However, this work lacks the consideration of Graphiti that also follows a manual approach but is more tailored to developing editors than .NET and Netron.

GMF vs. Graphiti. In his Master's thesis, Refsdal [Ref11] evaluates the differences between GMF and Graphiti using the PREDIQT method. For that, he uses a reference implementation. It remains to be seen whether his findings are also applicable in the context of the PCM. We intend to close this gap.

Other editor frameworks. Besides GEF, GMF, and Graphiti, there exist several other editor development frameworks for Eclipse. The Piccolo2D [Shn96] toolkit leverages hierarchical scene graph models, known from 3D environments, for editor development. We use similar ideas because we suggest to organize classes for Graphiti editors according to the (hierarchical) PCM structure. Papyrus [Pro13] can also be used for editor development, even though the documentation for such development is sparse. According to Schneider et al. [SSvH13], these frameworks suffer from cumbersome extension mechanisms, thus, indicating a low maintainability. Furthermore, motivated by the focus on user editing of previously frameworks, Schneider et al. [SSvH13] introduce the KLighD [SSvH13] framework allowing to synthesize diagrams fully automatically. We do not investigated such automatisms in this paper and leave it as an interesting future work.

6 Conclusions

We argue about benefits and flaws of the graphical editor frameworks GMF and Graphiti. After presenting both technologies, we evaluate them in the context of the PCM and different requirements, namely functional completeness, learnability, modifiability, reusability, and comfort. We also show how Graphiti can be applied to PCM editor development.

We find Graphiti to be superior to GMF regarding most of the requirements and that Graphiti-based editors can replace the current GMF-based implementation of the PCM editors. We come to the conclusion that rewriting the PCM editors using Graphiti will increase their general maintainability and extensibility for PCM editor developers. This is especially helpful when new editors are introduced or altered in the future. Furthermore, our application of the Graphiti framework to PCM editor development helps editor developers to have an easier start when developing such editors.

In future work, we plan to complete the prototyped composite structure editor and provide Graphiti-based alternatives for the remaining PCM editors as well. Also, we will make use of the extensibility of the new editors to implement a comparison editor to compare two composite structures as introduced by Stritzke in his Master's thesis [Str13].

References

[BGK+11] C. Brand, M. Gorning, T. Kaiser, J. Pasch, and M. Wenz. Graphiti - Development of High-Quality Graphical Model Editors. *Eclipse Magazine*, 2011.

[ISO10] ISO/IEC. ISO/IEC 25010 - Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models. Technical report, 2010.

[KB07] K. Krogmann and S. Becker. A Case Study on Model-Driven and Conventional Software Development: The Palladio Editor. In W. Bleek, H. Schwentner, and H. Züllighoven, editors, *Software Engineering 2007*, volume 106 of *LNI*, pages 169–176. GI, March 27 2007.

[Pro13] Eclipse Modeling Project. *Papyrus UML*. 2013. Visited on 14/11/2013.

[Ref11] I. Refsdal. Comparison of GMF and Graphiti based on experiences from the development of the PREDIQT tool. Master's thesis, University of Oslo, November 2011.

[SAP12] SAP AG. Graphiti Tutorial. `http://help.eclipse.org`, 2012.

[Shn96] B. Shneiderman. The eyes have it: a task by data type taxonomy for information visualizations. In *IEEE Symposium on Visual Languages*, pages 336–343, 1996.

[SSvH13] C. Schneider, M. Sponemann, and R. von Hanxleden. Just model! Putting automatic synthesis of node-link-diagrams into practice. In *VL/HCC*, pages 75–82, 2013.

[ST06] A. Shatalin and A. Tikhomirov. Graphical modeling framework architecture overview. In *Eclipse modeling symposium*, 2006.

[Str13] C. Stritzke. Considering Architectural Knowledge in the Reverse Engineering Process of Component-Based Software Systems. Master's thesis, University of Paderborn, 2013.