Projecting UML Class Diagrams from Java Code Models

Heiko Klare, Michael Langhammer, and Max E. Kramer
Chair for Software Design and Quality (SDQ)
Institute for Program Structures and Data Organization (IPD)
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany
{heiko.klare@student.kit.edu, michael.langhammer@kit.edu, max.e.kramer@kit.edu

ABSTRACT

In model-driven software development, source code and other artifacts are used to describe and develop a software system. UML class diagrams are one of the most common models that are used. A UML class diagram models classes and interfaces of a software system as well as their relations.

The usage of UML class diagrams in addition to source code can lead to drift and erosion if the models are not kept consistent with code changes and vice versa: Existing solutions solve this problem using consistency mechanisms that update the source code and UML class diagram accordingly. The development and maintenance of such consistency mechanisms can result in considerable effort and costs.

In this paper, we present a prototype for a new UML class diagram editor that is realized as a projection of a Java source code model. The editor does not use an explicit UML model. It provides another concrete syntax for a subset of the source code elements and their relations. A model represenation of the source code is used as a single underlying model (SUM) for the projective UML class diagram view. As a result, code and diagrams are updated automatically without the need for a consistency mechanism. The current minimal UML class diagram editor uses state-of-the-art model-driven software technologies and adds less than 2000 LLOC to the Eclipse IDE, Sirius and JaMoPP.

Categories and Subject Descriptors

D.2.2 [**Design Tools and Techniques**]: Object-oriented design methods; D.2.11 [**Software Architectures**]: Languages

Keywords

UML class diagram, projective view, round-trip engineering

1. INTRODUCTION

In modern model-driven software development, not only source code is used to develop a software system, but also other artifacts, such as component or class diagrams. For object-oriented software development, UML class diagrams [Obj11] are a common language to model software systems [Lan+14]. Within a UML class diagram, elements such as classes, interfaces and methods as well as the relations between them are represented. Certain details, such as the implementation of method bodies, are not shown. Hence, an UML class diagram can provide a quick overview of a software system. To have up-to-date UML class diagrams during the evolution of a software system, the UML class diagrams have to be kept consistent with the source code. If the diagrams are not kept consistent with the code, the well-known problems architecture drift and architecture erosion [PW92] arise. To solve these problems, a lot of tools, especially for UML class diagrams, already exist.

We subdivide these tools into three categories. The first category of tools generate UML class diagrams dynamically from the source code. This generated diagrams help to get an overview about the software system but can not be edited. The second category comprises forward engineering tools, which can be used to generate a blueprint of the system's source code from UML class diagrams. The third category contains tools that combine the first two categories. Hence, tools in the third category support round-trip engineering between source code and UML class diagrams. This means as soon as developers change the code the architecture is updated immediately and visa versa. A lot of tools that fall into the third category, e.g. UML Lab[1], have been developed. To our knowledge, all of these tools use an explicit consistency mechanism to keep the source code consistent with another model, which is used for the UML class diagram.

In this paper, we present a round-trip engineering approach for the third category that does not need to use an explicit consistency mechanism. Instead, we create a projective view onto the source code, which omits the problem of keeping different artifacts consistent during the development process. By creating a projective view, we achieve consistency between UML class diagrams and source code by design: all changes made in the UML class diagram editor are changes to a different representation of a subset of the source code so that there is no need for change propagation or something similar. Since we do not use a consistency mechanism, our editor should be easier to maintain in cases where either the UML metamodel or the source code metamodel changes. In such cases, we only have to update the view definition instead of changing the consistency mechanisms. Our approach can be seen as an implementation of the Orthographic Software

[1] http://www.uml-lab.com/

VAO 2016, 2 March, 2016, Karlsruhe, Germany

ISSN 2190-4782.

Modelling (OSM) approach, which was introduced by Atkinson et al. [ASB10]. OSM introduces the idea of a Single Underlying Model (SUM) that contains all artifacts, which are used to develop a specific software system. The access to the SUM is only possible via defined views. In our approach, we use the source code as the SUM. As views, we use the standard source code view and the UML class diagram view. Figure 1 exemplifies the functionality of our editor. The figure shows a class diagram with two classes, two interfaces and three methods. An edit operation is performed in step (1): A new method called *download* is added within the source code. The UML editor is updated (2) after saving the source code files containing the new information.

If one wants to show information in the UML class diagram that can not be generated from the source code solely, we propose, similar to other tools, an embedding mechanism into the source code. One important concept, for example, that can be used within the UML diagram but does not have a source code equivalent are the associations. To embed associations in the source code, we have developed annotations that allow developers to define the source and target multiplicity as well as the kind of the association. Hence, the creation of a projective view is possible because our UML class diagram does not contain information that is not contained within the source code already.

Our prototypical implementation of the Projective UML class diagram editor for Java (ProjUMLed4J) can be downloaded online[2] and uses state-of-the-art model-driven software development technologies: ProjUMLed4J is based on Eclipse Sirius [VMP14] and the Java Model Parser and Printer (JaMoPP) [Hei+10]. Both tools are realized using the Eclipse Modeling Framework (EMF). While our approach for a projective UML editor could be used for any object-oriented language, the implementation is currently limited to Java.

The remainder of the paper is structured as follows. In Section 2 we introduce the necessary foundations. Section 3 gives an overview about the related work. In Section 4 we explain the realization and features of our new UML editor. In Section 5 we discuss the advantages and disadvantages of our approach. Section 6 concludes the paper and gives a brief outlook on the future work.

2. FOUNDATIONS

2.1 Model-Driven Software Development

In MDSD (Model-Driven Software Development) models are the primary artifacts of the development process. This means that in contrast to model-based software development, models are not only used for some tasks such as documentation. Instead, every artifact is either a model conforming to an explicit metamodel, which describes the set of allowed instances, or it is derived from a model. Therefore, even code is either generated from models or treated as a model that is used to obtain other models. Source code can be treated as a textual representation of a code model with special printing and parsing capabilities. This makes it possible to apply techniques that were developed for arbitrary models and to abstract away from the textual nature of code. Programs that consume or produce models are called model transformations and can also be described as model instances of a transformation metamodel.

[2]https://sdqweb.ipd.kit.edu/wiki/Vitruvius/ProjUMLed4J

2.2 Class Diagrams of the Unified Modeling Language (UML)

A well-known diagram type of the UML ISO/IEC 19505-2:2012(E) standard are class diagrams. They represent classes and interfaces of object-oriented software that may be grouped into packages. Many concepts of class diagrams, such as inheritance, attributes, or operations, have direct equivalents in object-oriented programming languages, such as Java. The initial purpose of the UML was, however, not code generation or co-evolution but support for the documentation and design of software. Therefore, there is not a universal mapping to object-oriented code for all concepts that are available in UML class diagrams. Special associations between classes, such as aggregations, which represent whole-part relationships, can, for example, be realized in different ways in object-oriented code. Round-trip engineering tools for UML class diagrams and object-oriented code rely on such mappings although they do not always make them explicit.

2.3 Eclipse Modeling Framework (EMF) and Sirius

EMF is a set of tools and plug-ins for the Eclipse IDE that provides all infrastructure necessary for model-driven development. It provides the metamodelling language Ecore, which is closely aligned to the Essential Meta-Object Facility standard ISO/IEC 19508:2014(E), and is used to define metamodels and implement their instances in a variety of tools and domains. Code generation facilities of EMF provide a very convenient way to obtain code for instantiating Ecore metamodels and editing in a customizable tree-based editor.

To obtain grapical editors for Ecore-based models the Sirius [VMP14] framework can be used. With Sirius, so called representation descriptions can be defined. These descriptions specify how elements of the represented models shall be displayed and how they can be edited. A description can be instantiated for an Ecore-based model and results in a diagram for that model. The descriptions are interpreted dynamically, which means that an existing diagram is opened with the actual version of the representation description and not with the one it was created with. Each description must be assigned to a viewpoint, which summarizes descriptions that belong together and can be simultaneously activated or deactivated for a project that uses Sirius diagrams. To create diagrams with Sirius, a Sirius session must be created. A Sirius session exists for each project that uses Sirius diagrams. The session contains references to the used models and the created representations with their layout information. It is persisted in a special file in the root folder of the project.

2.4 Java Model Printer and Parser (JaMoPP)

JaMoPP [Hei+10] parses Java source code and represents it as an instance of an Ecore metamodel so that model-driven tools can be applied to it. It also supports printing generated or modified code models without any loss of information. Printing and parsing of source code is performed in a fully transparent way so that transformations and analyses of Java models do not have to consider the special serialization as Java source files. Cross-references between model elements are established after parsing so that tools that use JaMoPP can easily navigate the models and perform analyses.

2.5 Software Architecture Views

The ISO/IEC/IEEE 42010:2011(E) standard mentions two

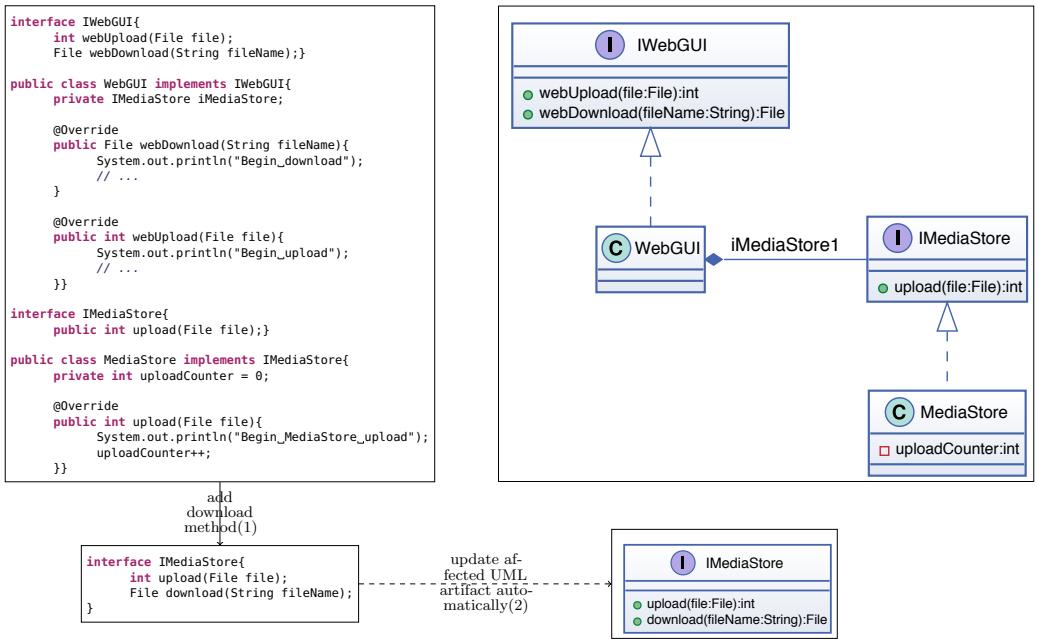

Figure 1: Example of the UML editor's functionality: After adding the *download* method to the interface *IMediaStore* the UML editor is automatically updated.

different approaches for architectural views that can be applied to any kind of view: *Synthetic* approaches integrate views with each other and have to manage correspondences and updates. *Projective* approaches use a central repository from which all views are derived respectively projected.

3. RELATED WORK

UML class diagrams are a common and established representation of software fragments. In the last years, many tools for generating, editing and viewing class diagrams have been developed. As we already introduced in Section 1, we subdivide these tools into three categories, which are *read-only views* generated directly from the source code, *forward engineering tools* that generate code stubs from the UML class diagram, and *round-trip engineering tools* providing a synchronization mechanism for code and the diagram. Since our tool falls into the third category, we compare our approach with others in the same category.

UML class diagram editors providing round-trip engineering with the source code can again be subdivided into three different categories based on the way they represent the additional semantic information, which a class diagram contains as opposed to source code. Most of the existing tools fall into the first category. They either use a separate UML model that stores the whole information that is needed for the diagram or an artifact containing the additional information compared to the source code. Tools of the second category use a central model that stores the whole information of all related artifacts. Thus, the model contains the UML information as well as implementation and further details. The third and last category aggregates tools which use no additional artifact to store semantic information but extract it from the code or, if necessary, store it directly in it.

The list of approaches that fall in the first category is long. Popular tools are ArgoUML [Ram+03], IBM Rational Software Architect [Cla10], MagicDraw [No 12] and UML Lab, which arose from FuJaBa [Nic+00]. The first three approaches use explicit synchronization mechanisms that must be triggered by the user. While an explicit update in one direction, from diagram to code or vice versa, mostly tries to integrate the changes into the other artifact, some tools simply generate the other artifact again. In the case of ArgoUML, this means that changes in a UML class diagram can only be transformed into a new and empty code template but not be integrated into existing code. MagicDraw also provides an explicit synchronization mechanism of diagram and code. Changes are integrated into the other artifact. Nevertheless, even simple modifications such as the renaming of a field are not detected. Instead, the field exists twice after the synchronization of the renaming operation. On the contrary, UML Lab uses an approach combining implicit and explicit synchronization. If an artifact that is affected by a change is currently opened, it is automatically updated. For instance, if a class is opened in a Java source code editor and modified in a UML class diagram, this modification is automatically synchronized with the code. To ensure consistency with closed artifacts, an explicit synchronization can also be triggered. UML Lab also stores the additional semantic information of UML class diagrams inside the code using formalized code comments. This allows the sharing of these information without sharing the diagram itself. The Rational Software Architect also provides a primarily implicit synchronization mechanism. Modifications in one view are transferred to the other. However, modifications of associations inside the code are not represented in the UML diagram.

A popular tool that falls into the second category is the Enterprise Architect [Spa14]. It provides a toolset for the design of software architectures containing several UML diagram types. A central model contains the information that is necessary for all diagrams and for the generation of code for different programming language from the architectural design. Modifications in a UML class diagram are persisted in this central model. However, the synchronization with the source code has to be called explicitly. This also means that concurrent modifications in the diagram and the code are not possible because one overwrites the other.

The last category covers tools that use only the source code as the model for the UML class diagram. Because these tools do not use an additional model for storing further information, there is no mechanism for the synchronization of semantic elements. A popular tool that falls into this category is Together from Borland [Bor05]. The central feature of Together is the so called LiveSource mechanism, which synchronizes the class diagram with the source code. The semantic information that is represented by the UML class diagram is either already stored in the source code or persisted in formalized code comments, for example, for multiplicities of associations. Together is a quite unintrusive tool since it does not influence an existing Java project apart from the mentioned code comments. The layout information of the class diagrams is stored in a separate folder of the project.

A tool that follows an approach that is similar to the one we present here was published by the developers of JaMoPP. Their GMF-based editor [Hei+09] uses JaMoPP to present the classes inside a package and their relationships graphically. The editor is not UML compliant and is not compatible with the latest version of Eclipse but is conceptually also a graphical editor that uses the source code as the underlying model. The UML Aid Explorer[3] is another tools that can be assigned to the third category of UML class diagram tools. It provides the generation of read-only class diagrams based on the source code as the underlying model. Due to this mechanism, the diagram is always in sync with the source code. Additional information of UML class diagrams is extracted from the model, if possible, but cannot be modified due to the missing editability of the diagram contents.

The language workbench mbeddr [Voe+13] is built on top of the Meta Programming System (MPS) [Voe13] and allows projective editing of Java source code [PSV13] and many other languages. It uses an Abstract Syntax Tree (AST) instead of a textual serialization from which all views are projected. All manipulations in the views are translated to manipulations of the central AST. Java code is edited in a projective editor that offers similar functionality to editors that have to parse the code. Currently, it is however only possible to visualize and navigate UML class diagrams in mbeddr but no edit operations are supported.

4. UML CLASS DIAGRAM EDITOR

The editor that we introduce in this paper provides functionality for the viewing and editing of classes in a UML class diagram that builds on Java source code as the underlying model. A single diagram represents one package of an existing Java project since a package represents a semantically isolated set of classes. However, the concepts we present can be easily extended to arbitrary sets of classes.

[3]http://www.objectaid.com

The implementation of our approach uses the EMF tools Sirius and JaMoPP. JaMoPP allows us to treat Java source code as an Ecore-based model. For the graphical representation of the model as a UML class diagram we chose Sirius. Certainly, any other tools providing these functionalities can be used for these tasks. The use of state-of-the-art model-driven software development technologies allows us to implement our minimal prototype in less than 2000 LLOC. The remainder of this section explains how our diagram generation and diagram editor work, and what the limitations of the implementation currently are.

4.1 Persisting semantic UML information

UML class diagrams are highly abstract representations of software fragments. Thus, most of their elements can also be found in more precise artifacts such as source code. Only few UML elements do not have a clear equivalent in the code. Especially association properties, such as multiplicities, belong to these elements that can be implemented in code in many different ways, if at all, and thus cannot be unambiguously extracted from code. This information has to be stored separately.

In Section 3, we summarized several tools that synchronize program code and UML class diagrams and therefore use some mechanism to store the additional semantic information of class diagrams. While most of them use a second artifact to store some kind of code-independent model that is supplemented with this information, some of them, such as UML Lab, embed the information in the program code.

Since we rely on the capabilities of Sirius and JaMoPP, the easiest way to provide the UML specific information is to integrate it into the source code. This decision has several further advantages regarding consistency, which we will discuss later. Since the information increases the code size, we have decided to use Java annotations for storing it. Annotations are compact and cannot be easily corrupted like comments, for instance, through accidental modifications in the Java code view, because they are a feature of the programming language that is syntactically and semantically checked by the compiler.

An association is stored by writing the annotation *@Association* next to a field that represents the association in one direction. Default values for all necessary association properties are specified in the annotation definition and can be overwritten in each case. Currently, supported values are the multiplicities and the type of an association, which can be an aggregation or a composition.

4.2 Diagram generation process

Our process of generating a UML class diagram for a specific Java package can be generally separated into two steps: Initially, an optional preprocessing step extracts semantic UML information out of the program code and embeds this information in the code to generate a more expressive diagram. The second step is the diagram generation itself, that again consists of several steps that complete in a UML class diagram that is opened in an editor.

Source code preprocessing

Some of the information in a UML class diagram that cannot be obviously extracted from the program code can be approximated conservatively. We assume that a field with a type that is defined in the same package shall be presented

as an association. Therefore, an association annotation is created for all the affected elements.

An interesting property of associations are the multiplicities. The common multiplicities *0..1* and *0..** can be reliably reproduced from code. Consider the following listing. During the generation of a UML diagram we generate the annotation *@Association* for the attribute *myStringList* if *MyString* is in the same package as *MyClass*. The multiplicity values are represented by annotation attributes and are set to *1* by default. Since the *myStringList* references an arbitrary number of *MyString* objects, the target multiplicity is overwritten with *0..**, whereas the upper bound is represented by *-1*.

```
public MyClass {
      @Association(targetLowerMultiplicity=0,
       targetUpperMultiplicity=-1)
      private MyString[] myStringList; }
```

More precise multiplicities, especially limited ranges, are hard to assure in code and even more hard to extract from code. Generally, there are two types of associations. If they are single-valued, the code contains a field with the type of the association target. Furthermore, if a field is declared as *final*, its value cannot be changed and thus the multiplicity can be set to *1*. By contrast, if the association is multi-valued, the field is a collection of the association target type, which can be any class implementing the *Collection* interface, an array, or a user-defined container class. While the first two can be easily determined by investigating the inheritance hierarchy respectively check whether the type is used within an array, the latter one cannot be recognized as a multi-valued association generally.

We generate this information for each field of classes in the considered package investigating and modifying the JaMoPP model of the code. This process could also be performed based on the Abstract Syntax Tree to avoid the complexity of JaMoPP. To improve the approximation results, the process could also be realized semi-automatically. In reasonable cases, the user could be asked for more specific property values. An example could be the check of a collection size for a limited value inside a method, which might be an indicator for the upper bound of the association multiplicity. To reduce the number of annotations, our approach could be easily adapted to only persist annotations for non-default values and to implicitly handle default values in the view.

In contrast, an association cannot be automatically identified as an aggregation or composition since the information can even not be extracted using static Java code analysis.

Diagram generation

The generation of a UML class diagram consists of three major steps. At first, a Sirius session has to be created and prepared. After that, the needed resources have to be added to the session and finally the diagram has to be instantiated. The diagram generation process is shown in Figure 2.

Sirius sessions are persisted in special files saving the resources and diagrams the session contains. The implementation of Sirius assumes such a file for every project that diagrams shall be created for. Thus, the diagram generation initially creates a session for the affected project if it does not already exist. Otherwise, the existing session is opened, which means that the persisted state, that contains the resources and the already created diagrams with their layout information, is read and resources of the session are loaded.

A Sirius session may contain different types of diagrams. These types are organized in viewpoints, which can be enabled or disabled. To enable the creation of UML class diagrams, its containing viewpoint is activated in the Sirius session.

For each diagram, Sirius needs a single root element whose content is displayed. Therefore, we cannot simply add the Java source files we want to display but need an element containing them. Since we decided to present a single Java package in each diagram, we use the package as the root element. Packages are not persisted but implicitly defined through the package declaration of Java classes. Thus, we have implemented a virtual package EMF resource that generates a package element instance of the JaMoPP metamodel. Afterwards, the Java classes inside the package are added as compilation units to this package element. This package resource is added to the Sirius session to be used as the root element for the diagram that is about to be created.

Sirius has an integrated mechanism for synchronizing the used resources with the diagram contents. It observes the persisted resources for modifications and reloads them in case of a change. Since the added package is a virtual element and has no persisted equivalent that can be monitored for changes, in this state no synchronization between code and the diagram would happen. To achieve this, the resources of the compilation units inside the virtual package element must be added to the Sirius session as well. Using the resources that are contained in the package element, Sirius reloads the Java classes on changes and implicitly updates the classes inside the virtual package that is displayed as well.

The mechanics of JaMoPP require a further adjustment of the previous process. When loading a class with JaMoPP, a proxy resolution, loading all classes which the currently loaded one depends on, is performed by default. These dependencies are treated as special proxy objects to avoid that their dependencies are resolved as well. If, in our scenario, a class of the package that is currently loaded depends on a class of the same package that will be loaded later, a proxy object gets created first. Although the class is completely loaded afterwards, the proxy object stays in the resource set. This way the Sirius session could potentially load classes twice, which will cause synchronization problems. If a class gets modified in the diagram, this change gets written back to the code. The proxy objects of the same class recognize that change of their resource and, therefore, this change is tried to be written back to the model. This violates the read-only state of the transaction, which is founded in the fact that the model should only be read to write the change to the Java file. Because of that, we need to ensure that the Sirius session contains each class only once. We achieve this by disabling the proxy resolution mechanism of JaMoPP while adding the classes to the Sirius session and reactivate it afterwards.

Finally, we can create an instance of our Sirius UML class diagram definition for the virtual package element. The diagram is automatically generated for the contents of the given package element with a default layout determined by Sirius and is opened in a Sirius diagram editor.

4.3 UML class diagram usage

After creating and opening the UML class diagram, a state-of-the-art Eclipse Sirius editor is shown. This editor shows the classes, interfaces and methods with parameters and return types as well as the visibility of the elements. Furthermore, it shows the relations of the elements in terms of inheritance and associations.

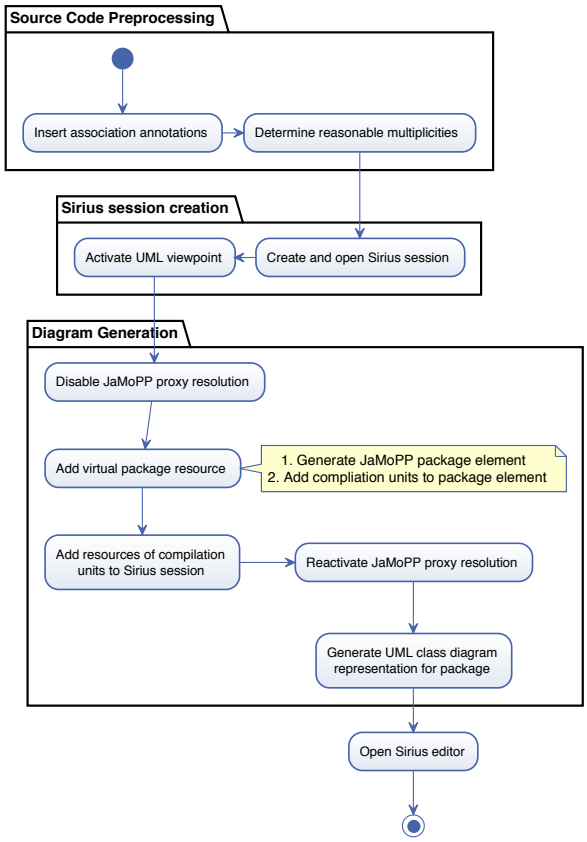

Figure 2: UML class diagram generation process for a specific package

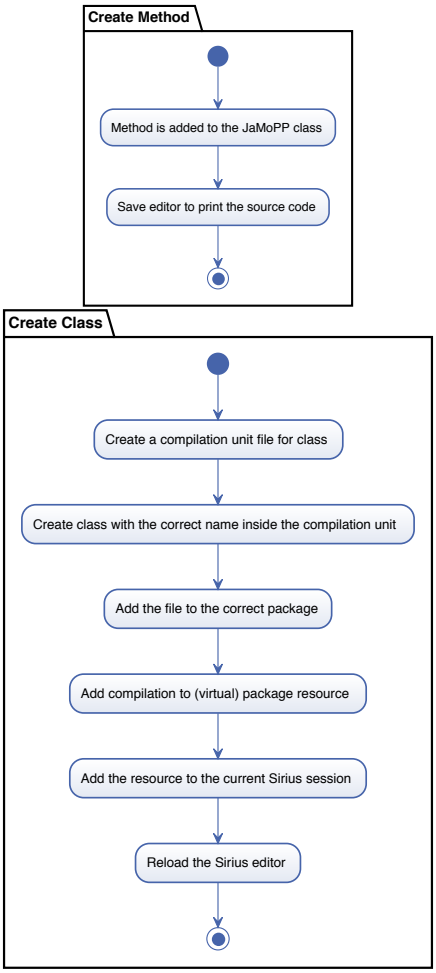

Figure 3: Steps that are necessary to create a method in comparison to steps that are necessary to create a class. To create a class first we have to create a resource and a new compilation unit.

As mentioned above, the editor allows users to change the model elements. For example, a name of the method can be changed by clicking on the method and typing the new name. Since the editor is just another view onto the source code, the source code is updated automatically after saving the editor. For rename operations, the editor automatically supports refactoring for the classes that are currently displayed. The reason for that is that each object of the model instance is created and instantiated once by JaMoPP. Other occurrences of the objects in the same model are only references to the original model. Moreover, the toolbox of the editor allows users the creation of fields, methods, classes and interfaces.

4.4 Features

We already introduced the functionality of the editor in Section 1. Further features are presented in this section. Within our editor, the navigation from the UML class diagram to the source code is possible by double clicking on the class.

Figure 3 shows the steps that are necessary to create a method or a class. The creation of a method is rather simple since we only have to create a new method element in the JaMoPP model. It is automatically synchronized to the code because Sirius calls the save operation of the JaMoPP model. The creation of a class is far more complicated: First, we have to create a file for the compilation unit. Within this compilation unit, we save the contents of the new class as text. After that, the file has to be added to the correct package. To get the new compilation unit into the Sirius session, we have to add it to the current Sirius session and to the virtual package resource. As the last step, we reload the Sirius editor to show the newly created class within the editor. This approach is complicated since Sirius, like other editor frameworks for EMF models, are optimized for only one resource containing one root element. In our case, we have many resources (the Java source code files), with each of them containing one root element.

In the future, a support of existing UML tools is planned. Therefore, we need an importer and an exporter from and to the UML Diagram Interchange format. This can be done using model-to-model transformations between the UML class diagram metamodel and JaMoPP.

4.5 Limitations

The current realization of our UML class diagram edi-

tor proves the feasibility of the concept of a projective and editable class editor based on Java code. Nevertheless, the implementation is limited in the elements it shows and the modifications it allows. Currently, no stereotypes and generics are shown and associations are always unidirectional. The persistence of bidirectional associations requires an extension of the current annotation mechanism for associations.

The editability of elements inside the editor currently covers a set of elementary modifications, that are essential for the usability of the editor. Some features which we plan to support in the future are the modification of multiplicities, switching between the presentation of a field as an attribute or as an association, the modification of class names and the modification of attribute, parameter and return types.

The modification of values, such as types, multiplicities and names, that are presented in the diagram can usually be achieved with the capabilities of Sirius. An actual limitation of Sirius is the editability of attributes of a relationship between elements, such as the multiplicities of an association. Thus, it is currently not possible to realize the modification of multiplicities using Sirius. Realizing the modification of types with Sirius features would also be limited since Sirius only allows the selection of types that are available in the Sirius session. Although it is theoretically possible to add all potentially used types as resources to the Sirius session, this is impossible in practice. The set of potentially used types is very large, for instance, the whole Java API would have to be available. The memory consumption of JaMoPP would be too high due to the large models and the loading process would take too long for an acceptable usability. One option for the realization of these modification is the use of the Extensible Editing Framework (EEF) to realize a two-step import mechanism that does not need to load all classes prior to their usage.

Although our editor provides a limited set of displayed elements and possible modifications at the moment, it can be easily extended by all the missing features to provide a full UML class diagram. Merely the integration of advanced UML elements, such as annotation classes or qualifiers, would require major modifications. Indeed, even established tools with UML editors ignore these elements. Finally, it is just more difficult to implement such elements with our approach than the already existing features but not impossible.

5. DISCUSSION

In this section we discuss our approach in comparison to other approaches and give an outlook to what can be done with projective views on source code. Until now, we executed some tests of our prototypic editor on some example projects (e.g. the simple project that can be seen in Figure 1). Part of our future work is to provide a case study where we apply our approach to open source projects. Since we use JaMoPP to parse and print Java source code into an EMF model representation and Sirius for creating the view, our approach is embedded into the Eclipse IDE.

Compared to most other tools, our editor only depends on the source code. This makes it easy to integrate our UML class diagram editor into existing development tool chains. In fact, there is no integration effort needed to use our tool with other tools. This means that developers can stick with tools they use to change the source code. The UML class representation is updated automatically if any changes are made to the source code. Since we embed the associations in the source code and thus do not need any other artifacts, our approach can be easily used in collaborative software development processes. Simultaneous editing of model diagrams, however, is not supported. If another artifact stores the additional semantic UML information, it has to be shared as well. In the case of a conflict due to concurrent changes by different developers, both artifacts have to be merged ensuring their consistency. Using our approach, the risk of a faulty merge is minimized due to the fact that the UML information is stuck to the element it describes in the source code. The layout information of the diagrams is not saved in the source code, but in the Sirius session file that exists for each project. Thus, if diagrams and their layouts shall be shared between developers, this file has to be shared between them as well.

Compared to other tools, we do not need a consistency mechanism. Ensuring consistency is one of the most difficult parts of the approaches using an additional artifact to store the UML diagram information. Each tool shows a faulty behavior in at least one situation. Just to give some example, UML Lab does not handle the moving of a class into another package correctly; MagicDraw does not recognize a renaming of a field, which leads to a duplication of the field after a synchronization. Enterprise Architect displays a field as an attribute as well as an association without ensuring consistency between them. Renaming the association leads to an additional attribute with the old name, and both elements are synchronized to the code that afterwards contains two fields instead of one. Since our approach uses the source code as a SUM and UML diagrams are only projected from it, it does not need an explicit synchronization mechanism and it cannot exhibit a faulty behavior in these situations as long as the generic synchronization logic of Sirius works well.

The missing necessity of a consistency mechanism also brings an advantage if one of the involved metamodels changes. If the Java metamodel changes, e.g. if a new Java version is released, only JaMoPP has to be updated. If the changes are not affecting the UML class diagram, which was the case from Java 1.5 until the current Java Version 1.8, the class editor and the view still work. In this case, our approach does not differ much from other approaches. They also have to adapt their source code parser and printer or generator if the Java language changes.

Since we are not using a dedicated UML metamodel, our projective view still works if the UML metamodel changes. However, in this case, our UML class diagram does show the new features that were added in the UML metamodel changes. To get an adapted UML class diagram, the view definition has to be updated. After doing that, the view shows the new UML model and the source code is kept consistent automatically. Tools that use a consistency mechanism need to update the view and the consistency mechanism, e.g., the transformations that are used to keep the UML class diagram consistent with the code.

To exemplify this, we consider a change that could be introduced into the UML metamodel: A new *EventInterface* class is introduced into the UML metamodel, which should be displayed in UML class diagrams. In the source code, an *EventInterface* is represented as a normal interface. To figure out whether a normal interface or a new event interface should be generated for an event interface, there are two possibilities: Either an event interface has to provide a specific method, or we can annotate standard code interfaces with a new *EventInterface* annotation. To adapt our editor, we have to

specify the representation of the new *EventInterface* in the editor. To enable the creation of an event interface within our editor, we have to add a new tool in the tool section and specify the meta class of JaMoPP (in this case *Interface*) that should be created. Tools that use a consistency mechanism would have to adapt at least the consistency mechanism as well as the view to reach the same functionality.

Our approach of having a projective view and using the source code as SUM could be extended to further UML views. For example, it is possible to create a UML package diagram view. A UML package diagram shows all packages of a software system and its dependencies. The necessary information to create such a model is already contained in the source code, and there is no need to extend the source code with additional annotations or comments. If one wants to use the approach of creating a projective view with the source code as SUM to create, for example, an activity diagram, the challenge arises that additional information is required that is not contained in the source code already. This information could be added by creating additional annotations or comments for the affected source code element, e.g., classes, methods or even statements within the method bodies. However, if too much additional information is included in the source code, it gets bloated and confusing for developers. To circumvent that, a new source code view could be created that shows the code without the annotations. However, this view is not easy to create since Sirius can not be used for the view creation. Hence, the approach of having projective views seems promising if only information that is contained in the source code must be displayed. To investigate this claim, the creation of additional views will be part of our future work.

6. CONCLUSION

In this paper we presented a projective UML class diagram view using the source code as SUM. The view is created using Sirius and JaMoPP. It gives an overview about the classes within a package and supports editing of the class diagram. Furthermore, we support associations and their multiplicities using Java annotations. Since we use JaMoPP, changes made in the view are automatically saved within the source code. Compared to existing tools we do not need a consistency mechanism to keep source code and the UML class diagram consistent during the software evolution.

In future work, we will extend our preprocessing to infer more association properties, such as *ordered* and *unique* for lists and set, as well as qualifiers for maps. Furthermore, technology-specific annotations, such as "OneToMany" of the Java Persistence API (JPA), could be supported. We will investigate whether it is possible to use our approach to create further projective views using the source code as SUM. We also plan to evaluate the maintainability of our approach in comparison to the maintainability of other tools.

References

[ASB10] C. Atkinson, D. Stoll, and P. Bostan. "Orthographic Software Modeling: A Practical Approach to View-Based Development". In: *Evaluation of Novel Approaches to Software Engineering*. Vol. 69. Communications in Computer and Information Science. Springer, 2010, pp. 206–219.

[Bor05] Borland Software Corporation. *Borland Together UML 2.1 Guide Version 2008 R3*. 2005.

[Cla10] Claire Liu. *Round Trip Engineering Scenario using Rational Software Architect and ClearCase Remote Client*. 2010.

[Hei+09] F. Heidenreich et al. *Jamopp: The java model parser and printer*. Tech. rep. 2009.

[Hei+10] F. Heidenreich et al. "Closing the Gap between Modelling and Java". In: *Software Language Engineering*. Vol. 5969. LNCS. Springer Berlin Heidelberg, 2010, pp. 374–383.

[ISO11] ISO/IEC/IEEE 42010:2011(E). *Systems and software engineering – Architecture description*. International Organization for Standardization, Geneva, Switzerland, 2011, pp. 1–46.

[ISO12] ISO/IEC 19505-2:2012(E). *Information technology – Object Management Group Unified Modeling Language (OMG UML), Superstructure*. International Organization for Standardization, Geneva, Switzerland, 2012, pp. 1–758.

[ISO14] ISO/IEC 19508:2014(E). *Information technology – Object Management Group Meta Object Facility (MOF) Core*. International Organization for Standardization, Geneva, Switzerland, 2014.

[Lan+14] P. Langer et al. "On the Usage of UML: Initial Results of Analyzing Open UML Models." In: *Modellierung*. Vol. 19. 2014, p. 21.

[Nic+00] U. A. Nickel et al. "Roundtrip engineering with FUJABA". In: *Proceedings of the 2nd Workshop on Software-Reengineering (WSR), August*. 2000.

[No 12] No Magic, Inc. *MagicDraw Technical Overview*. 2012.

[Obj11] Object Management Group (OMG). *Unified Modeling Language (UML), Infrastructure Specification – Version 2.4.1*. 2011.

[PSV13] V. Pech, A. Shatalin, and M. Voelter. "JetBrains MPS As a Tool for Extending Java". In: *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. PPPJ '13. ACM, 2013, pp. 165–168.

[PW92] D. E. Perry and A. L. Wolf. "Foundations for the Study of Software Architecture". In: *ACM SIGSOFT Software Engineering Notes* 17.4 (1992), pp. 40–52.

[Ram+03] A. Ramirez et al. *ArgoUML User Manual A tutorial and reference description*. 2003.

[Spa14] Sparx Systems. *Enterprise Architect User Guide*. 2014.

[VMP14] V. Viyovic, M. Maksimovic, and B. Perisic. "Sirius: A rapid development of DSM graphical editor". In: *Intelligent Engineering Systems (INES), 2014 18th International Conference on*. IEEE. 2014, pp. 233–238.

[Voe+13] M. Voelter et al. "mbeddr: instantiating a language workbench in the embedded software domain". In: *Automated Software Engineering* 20.3 (2013), pp. 339–390.

[Voe13] M. Voelter. "Language and IDE Modularization and Composition with MPS". In: *Generative and Transformational Techniques in Software Engineering IV*. Vol. 7680. LNCS. Springer Berlin Heidelberg, 2013, pp. 383–430.