# Proposal for a Multi-View Modelling Case Study: Component-Based Software Engineering with UML, Plug-ins, and Java

Max E. Kramer, Michael Langhammer
Karlsruhe Institute of Technology, Germany
{max.e.kramer, michael.langhammer}@kit.edu

## ABSTRACT

During the design and development of complex systems, multiple modelling languages are often necessary in order to describe a system for specific tasks and users. The resulting models can show parts of the same system from different perspectives or views, which is described by the term multi-view modelling. The overlap between individual views presents fundamental challenges, e.g. for sustaining consistency among views or for the creation of new views. A common multi-view modelling case study that covers essential challenges and requirements can be used as a basis for the comparison of approaches that address these challenges.

In this paper, we propose such a case study in the context of component-based software engineering with UML composite diagrams, Eclipse plug-ins, and Java code. We explain the overlap between the different views, propose new view types that aggregate information from several sources, and discuss essential challenges of multi-view modelling that are posed by this case study. These challenges are, for example, one-to-many and partial relations between elements of different views, and the constraint combination effects of different views. Our proposal contributes to the community effort that is required to obtain a common case study that enables an efficient comparison of multi-view modelling approaches.

## Categories and Subject Descriptors

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

## Keywords

Model-Driven Software Engineering, Software Architectures

## 1. INTRODUCTION

For many complex systems, a single modelling or programming language is insufficient to represent the system and its parts in ways that facilitate all development and analysis tasks. Therefore, many tailor-made languages are used to describe and realize systems with graphical or textual models from different perspectives or views. To manage these views and their overlap, various multi-view modelling approaches [1, 2, 4, 6] have been presented and discussed so far. Many approaches focus, however, on such specific concerns, that it is difficult to compare these approaches, e.g. in terms of view generation, consistency mechanisms, or their universal applicability to other languages. The first Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling (VAO) demonstrated the need for a common case-study in order to compare modelling approaches that support different perspectives or concerns. For the community of component-based modelling, such a case study called Common Component Modeling Example (CoCoMe) [3] has already proven to be very helpful. For mechatronic engineering Lucio et al. have presented a power window case study [5]. It involves different languages and views, but focuses on a concrete system and specific languages. This makes it difficult to analyse system-independent multi-view challenges such as the control of the semantic overlap between languages and the creation of views that may combine them.

In this paper, we propose a component-based software engineering scenario for such a common multi-view modelling case study. It involves UML component diagrams for describing the software architecture, Eclipse plug-ins for realizing component dependencies, and Java code for implementing the functionality of individual components. In designing and describing the case study proposal, we have pursued two conflicting goals: On the one hand, we have integrated relevant and realistic challenges of multi-view modelling, such as support for legacy views, complex but vague standards, partial and conditional overlap, or constraint transfer. On the other hand, we have tried to keep the case study compact and easily comprehensible, so that numerous multi-view modelling approaches can be applied and efficiently compared.

The rest of this paper is structured as follows: In Section 2, we present the languages used in our case study and the relations between them. In Section 3, we propose view types to access artefacts of the different languages. In Section 4, we suggest editability rules for these view types. In Section 5, we discuss important challenges that are posed by this case study. In Section 6, we draw some final conclusions.

## 2. CONTEXT AND MAPPINGS

In this section, we introduce the application context of the case study and propose mappings between the involved languages. The goal of the scenario is to keep UML component diagrams, Eclipse plug-ins and Java source code consistent during the development of a software system. The scenario can easily be extended, for example, to engineering scenarios that also cover non-software artefacts or that involve component-based languages such as SysML or AUTOSAR.

The mapping between UML composite diagrams and the Eclipse extension mechanism is as follows: For each UML composite diagram, three plug-ins are created. The first one is the *models* plug-in, which contains the UML composite diagram itself. The second one is the *contracts* plug-in. It contains all architectural interfaces and signatures that are used in the UML composite diagram. It also contains a *data types* package that contains classes, which can be used as parameter types and return types in interfaces signatures. The third plug-in that is created, is an *assembly* plug-in, which instantiates and connects all the components that are used in the diagram. Other components or end users can use the *assembly* plug-in by accessing the exported interfaces.

A UML component is represented by an Eclipse plug-in. To enable access to all architectural interfaces, data types, and utility functions, this plug-in has a dependency to the contracts plug-in of the diagram. The Eclipse plug-in also contains a component-realization class, which implements all interfaces that are provided by the component. Thus, this class is the corresponding class for the UML component. To enable instantiation of the component, the package, which contains the component-realization class, is exported from the plug-in. Hence, it is possible for the *assembly* plug-in or composite components to create instances of the component.

All provided and required interfaces in the UML diagram are considered architectural interfaces and are represented in an interface package of the *contracts* plug-in. The signatures of these interfaces are mapped to Java methods. Non-primitive data types, which are used as return or parameter types, are represented in a data types package in the *contracts* plug-in. To enable other plug-ins to implement these interfaces, an extension point is provided for each interface.

In addition, a provided interface of a component is represented by an extension for an extension point, which is defined for the interface. This implementation is done by an Eclipse plug-in that represents the component. In Java code, a provided interface is represented by the component-realization class of the Eclipse plug-in implementing the interface. A required interface of a component is mapped to a private field and a constructor parameter, with the type of the interface, in the realization class of the component.

The composition in the *assembly* is realised as follows: Like the other plug-ins, the *assembly* plug-in has a realization class. It implements the interfaces provided by the UML diagram and in its constructor it creates instances of the component-realization classes of the components used in the UML diagram. A UML composite component is mapped to two Eclipse plug-ins: The first plug-in has dependencies to the contracts plug-in and the plug-ins representing the components used in the composite component. Like the *assembly* plug-in, the composite component plug-in composes the components used in it. The second plug-in is a feature plug-in that combines all components that are instantiated in the composite component and the composite component itself.

A similar mapping could be defined for other Architecture Description Languages (ADLs), other plug-in mechanisms, or other general purpose programming languages.

To illustrate the above-mentioned mapping rules we present an exemplary MediaStore system, which can be used by end users to access media files e.g. audio files, via HTTP. We do *not* propose to make this or any other system part of the case study because this would shift the focus from the meta level to the instance level in an undesired way. The example system consists of two components: *MediaStore*, which provides the *IMediaStore* interface, and *WebGUI*, which provides the *IHTTP* interface and requires the *IMediaStore* interface. The mapping to plug-ins and source code is as follows (cf. Figure 3): A *contracts* plug-in contains two interfaces *IMediaStore* and *IHTTP* as well as a data type *File*. The two components are mapped to two Eclipse plug-ins *MediaStore* and *WebGUI*. The *MediaStore* plug-in provides an extension point *IMediaStore*. This extension point is realised by a class *MediaStoreImpl*, which implements the *IMediaStore* interface and its methods. The *WebGUI* plug-in provides an extension point *IHTTP*, which is realised by a *WebGUIImpl* class (cf. Figure 1). Both plug-ins have a dependency to the *contracts* plug-in. An *assembly* plug-in provides an *IHTTP* extension and contains a component-realization class *MediaStoreAssembly*. This class implements the provided *IHTTP* extension point by implementing the *IHTTP* interface. In the constructor of the class, instances of the *MediaStoreImpl* and the *WebGUIImpl* classes are created. To allow *WebGUIImpl* to use the *IMediaStore* interface a *MediaStoreImpl* instance is provided to the constructor of *WebGUIImpl*.

## 3. VIEW TYPES

For this case study we propose three well-known view types displaying information of a single metamodel and three additional view types, which combine information from different metamodels. The view types for UML Composite Diagrams, plug-in XML files and manifest files of Eclipse, and a Java source code editor should be provided in order to support essential standard tools for developers and architects. Additionally, view types for annotated code views, simple component realisation views, and global navigation views should be defined. These view types display relations between the different models that are not present in the models themselves. Therefore, they are crucial for developers and architects for understanding these relations. Further view types that are not predefined should also be realized by approaches that are compared with this case study in order to assess which range of view types is supported by which approach.

An annotated code view (Figure 1) should show the complete Java source code and Java-annotations that establish a link to the elements of the UML Component Diagrams and the Eclipse plug-ins. An according type annotation should be added to every Java class that is the main class of a UML component (Line 1) and to every Java interface that realizes an interface of a UML component. Furthermore, every implements relation between a Java class that realizes a UML component and a Java interface that realizes an interface that is provided by this component has to be expressed with another type annotation (Line 2). Similarly, every field of a Java class that realizes a UML component and that represents a requires relation between the component and an architectural interface has to be annotated (Line 4). The same annotation is necessary for every constructor parameter that represents such a requires relation (Line 6).

A simple component realisation view (see Figure 2) should show UML components with a link to their main realizing Java class and UML interfaces with links to their Java coun-

```
@ADLRealizes("WebGUI")
@ADLProvides("IHTTP")
public class WebGUIImpl implements IHTTP {
    @ADLRequires("IMediaStore")
    private IMediaStore iMediaStore;
    public WebGUIImpl (@ADLRequired("IMediaStore") {
      IMediaStore iMediaStore)
        this.iMediaStore = iMediaStore;
    }
    @ADLProvides("IHTTP")
    public int upload(File file){
        //...
```

**Figure 1: Example view of annotated code view type**

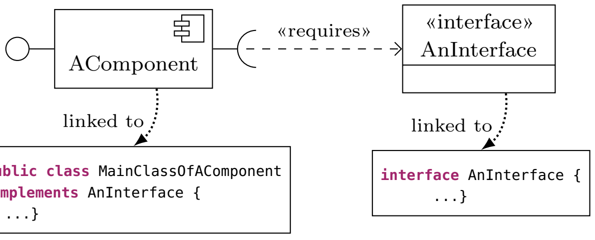
terparts. The view type for global navigation (see Figure 3) is a supertype of the simple component realisation view type. It should give architects and developers the possibility to navigate to all code and plug-in configuration artefacts from UML component diagram elements. Composite components should also be linked to the XML file of the corresponding Eclipse feature project. This project groups all plug-ins of all components that are instantiated in the composite component. Furthermore, connectors should be linked to the connected UML interfaces and to the corresponding code and plug-in artefacts. The requiring connector end should also link to the resulting dependency in the manifest of the requiring component and to the corresponding extension point provided by the plug-in of the requiring component. Similarly, the providing connector end should link to the corresponding extension provided by the plugin-in of the providing component. These links are only slightly different for the two connector kinds: Assembly connectors, which link ports of components, and delegation connectors, which link a port of a composite component to a port of a contained component.

## 4. SUPPORTED CHANGE OPERATIONS

The legacy view types for UML components, Eclipse plug-ins, and Java code should support all change operations for all elements in order to provide downward compatibility. In this section, we propose editability restrictions for new view types in order to avoid consistency conflicts but we do not discuss how views should be recomputed after external updates.

For the annotated code view type, we propose to allow additions, changes, and deletions of annotations. This way developers can express directly in the code which component or architectural interface is realized by a class or Java interface. A developer can influence which architectural interfaces are provided and required by changing the implements relation between Java classes and Java interfaces together with the fields and constructor parameters for required interfaces. If a field is changed but the corresponding parameter is not or the other way round, then we propose not to restrict editability: It is sufficient if the requires relation from a component to an architectural interface is changed as soon as one of the two redundant code locations is changed.

For the simple component realisation and global navigation view type, we propose to provide full editability for UML elements and no editability for the links to plug-in and code artefacts. If a user wants to change these links, he would have to change the code and plug-in configuration directly



```
public class MainClassOfAComponent
 implements AnInterface {
  ...}
```

```
interface AnInterface {
    ...}
```

**Figure 2: Simple component realisation view type**

so that the links in these views can be recomputed.

## 5. CHALLENGES AND CRITERIA

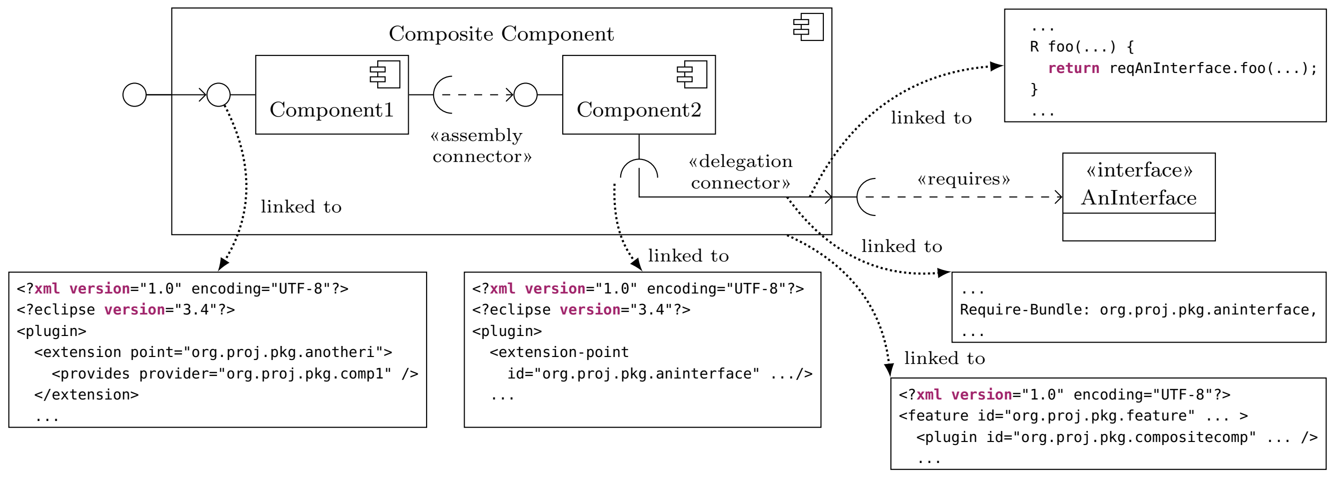
The proposed scenario and view types are particularly suited as a common multi-view modelling case study because they pose, amongst others, the following five challenges:

i) The first challenge is the support of the view types we explained in Section 3. Here we have identified three sub-challenges: The first sub-challenges is to integrate the views for Java source code, UML composite diagrams, and Eclipse plug-ins. The second sub-challenge is to create the new view types for annotated code, simple component realisation, and global navigation, which have to combine elements from different artefacts. The third sub-challenge is to allow all modifications in previously existing views and only some modifications in new views as described in Section 4.

ii) The second challenge is to sustain the semantic relation and consistency between the different languages. This challenge consists of three sub-challenges: The first sub-challenge is that changes in a view have to be reflected in other views according to the mapping rules we proposed in Section 2. The second sub-challenge is the correct realization of one-to-many relations, e.g. between a component on one side and a plug-in as well as a Java class on the other side. The third sub-challenge are the partial relations that result in straightforward consistency in one but not in the other direction. Interfaces that are provided or required by UML components, for example, should always lead to Java interfaces. But, not every Java interface that is implemented by a class that realizes a component has to be an architectural interface of a UML composite diagram.

iii) The third challenge is to handle ambiguous changes that have unclear effects on corresponding models. Such effects can occur, for example, if not all elements that are involved in a many-to-one relation are changed. For instance, if a developer removes a private field in a class that results from a required architectural interface but does not remove the corresponding constructor parameter, then the effect to the corresponding UML model is unclear. If this ambiguous change is possible, it has to be decided whether the requires relation between the component realized by the modified class and the architectural interface should also be removed.

iv) The fourth challenge are the different constraint types, constraint representations, and validation techniques of the different modelling languages. UML composite diagrams, e.g. have numerous but vague constraints defined in natural language and OCL in the UML superstructure. Once these constraints were correctly encoded in a tool, they can be directly checked without further input from other models. The constraints for plug-ins extensions and manifest dependencies, however, can only be fully validated if the Java code



```
...
R foo(...) {
  return reqAnInterface.foo(...);
}
...
```

```
...
Require-Bundle: org.proj.pkg.aninterface,
...
```

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.4"?>
<plugin>
  <extension point="org.proj.pkg.anotheri">
    <provides provider="org.proj.pkg.comp1" />
  </extension>
  ...
```

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.4"?>
<plugin>
  <extension-point
    id="org.proj.pkg.aninterface" .../>
  ...
```

```
<?xml version="1.0" encoding="UTF-8"?>
<feature id="org.proj.pkg.feature" ... >
  <plugin id="org.proj.pkg.compositecomp" ... />
  ...
```

**Figure 3: Example view of global navigation view type linking architecture, code and plug-in artefacts**

is also taken into consideration. The annotated Java code can be checked without further input, but the syntax check should be further restricted by the typed annotations, for example with restricted code targets and name matching for UML components and interfaces.

v) The fifth challenge are the three serialization formats of the case study. The first format is a graphical UML composite diagram that may be stored, for example, in an XMI file as an instance of a metamodel defined with Ecore (a variant of the Meta Object Facility ISO standard). The second format is plain text for manifest files and Java source code. The third format is XML for Eclipse plug-in descriptions.

On the one hand, multi-view modelling approaches that are compared based on this case study should solve all these challenges to show their power and maturity. On the other hand, the degree to which these challenges are solved will help in comparing the approaches. In addition to the challenges induced by the scenario, there are some other properties of the case study that will ease the comparison: Well-known languages make it easy to assess what kind of additional view types are supported and how view definitions can be reused. The subtype relation between the simple component realization view type and the global navigation view type, for example, will permit an evaluation of view reuse mechanisms: Approaches might reuse the simple component realization view type for populating a global navigation view. Additionally, the subset relation between UML components and Java code might help in evaluating support for a different concrete syntax using the same metamodel: Is it possible to realize UML component diagrams as a direct view, i.e. different concrete syntax, for Java code in order to ease consistency management? Can a separate abstract syntax, i.e. metamodel, be used in order to decouple transformations and views? Furthermore, the used languages make it possible to evaluate whether an approach supports slightly different metamodel relations for the same views: Is it, for example, possible to use the same UML component diagram views if provided and required interfaces are realized differently in the Java code, e.g. based on the context or broker pattern? Finally, the combination of large standard languages like UML or Java, and small languages that are not standardized, such as manifest and plug-in descriptions, will show whether an approach can deal with both types of languages.

## 6. CONCLUSIONS

In this paper, we have proposed an extensible case study for multi-view modelling approaches in the context of component-based software engineering. The case study is based on UML composite diagrams, the Eclipse plug-in mechanism, and Java source code. We have presented mapping rules from UML composite diagrams to Eclipse plug-ins and Java code. In addition to three previously existing view types, we have introduced three new multi-language view types together with the change operations that should be supported by them. Furthermore, we have discussed various challenges of the case study, which represent task-specific evaluation criteria for multi-view modelling approaches. Therefore, this paper should be a solid basis for the development of a common multi-view modelling case study that uses some of the presented component languages and view types or further ones.

## References

[1] C. Atkinson et al. “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. Berlin/Heidelberg: Springer, 2010, pp. 206–219.

[2] A. Cicchetti et al. “A hybrid approach for multi-view modeling”. In: *Electronic Communications of the EASST* 50 (2011).

[3] S. Herold et al. “CoCoME - The Common Component Modeling Example”. In: *The Common Component Modeling Example.* Vol. 5153. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, pp. 16–53.

[4] M. E. Kramer et al. “View-centric engineering with synchronized heterogeneous models”. In: *Proceedings of the 1st Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling.* VAO ’13. Montpellier, France: ACM, 2013, 5:1–5:6.

[5] L. Lucio et al. *An overview of model transformations for a simple automotive power window.* Tech. rep. McGill University, Tech. Rep. SOCS-TR-2012.2, 2012.

[6] J. R. Romero et al. “Realizing Correspondences in Multiviewpoint Specifications”. In: *Enterprise Distributed Object Computing Conference, 2009. EDOC ’09. IEEE International.* 2009, pp. 163–172.