Supporting Modeling in the Large in Fujaba

Thomas Buchmann
Angewandte Informatik 1
Universität Bayreuth
D-95440 Bayreuth
thomas.buchmann@uni-bayreuth.de

Alexander Dotor
Angewandte Informatik 1
Universität Bayreuth
D-95440 Bayreuth
alexander.dotor@uni-bayreuth.de

Martin Klinke
Xavo AG
Enterprise IT Solutions
D-95444 Bayreuth
martin.klinke@xavo.com

ABSTRACT

Model-driven software development intends to reduce development effort by generating code from high-level models. However, models for non-trivial problems are still large and require sophisticated support for modeling in the large. Experiences from our current project, dedicated to a model-driven modular software configuration management (SCM) system, confirm this claim. This paper presents a stand-alone package diagram editor that has been developed using Fujaba and discusses its integration into the Fujaba tool suite.

Keywords

package, import, package diagram, model-driven development, modeling in the large, validation

1. INTRODUCTION

In object-oriented modeling, modeling in the large is an area which has not attracted sufficient attention, so far. The focus in object-oriented modeling tools is clearly on the level of class diagrams. When it comes to modeling the architecture of a system, particularly the dependencies between certain modules, package diagrams play a crucial role. The experiences gained in our project, dedicated to the development of a model-driven product line of software configuration management systems, showed us that managing the dependencies between packages is mandatory for the overall success of the project [4].

UML 2.0 offers concepts for structuring large models [6, 1]: A model may be structured into hierarchically organized *packages*. Each model element is owned by exactly one package. *Private elements* are not accessible from other packages, while *public elements* are visible. Each package defines a *namespace* in which the names of declared model elements have to be unique. Public model elements from other packages may always be referenced through their fully qualified names. A model element from an enclosing package may be referenced without qualification, unless it is hidden by an inner declaration.

Apart from nesting, UML 2.0 introduces the following relationships between packages: *Imports* merely serve to extend the set of those elements which may be referenced without qualification. UML 2.0 distinguishes between *public* and *private* imports, which are denoted by the stereotypes <<import>> and <<access>>, respectively. A private import makes the imported elements visible only in the importing package, while a public import simultaneously adds those elements to its exported namespace. Public imports are transitive; this property does not hold for private imports. UML 2.0 also offers *package merges*, which will not be discussed in this paper.

In this paper, we present the *MODPL package diagram editor*, which is a first step of supporting UML 2.0 package diagrams, either as a stand-alone editor or within the Fujaba tool suite. Furthermore, it can be used to validate Fujaba models against a pre-defined package diagram.

2. PACKAGE DIAGRAM META-MODEL AND EDITOR

Our *MODPL package diagram editor* has the capability of creating and editing UML compliant package diagrams. Basically it is **independent** of Fujaba or Ecore. But its meta-model is designed to provide an interface for third party tools - either to **import** an already existing package structure, or to **validate** visibility constraints.

The **package diagram meta-model** itself is a subset of the meta-model described in the UML specification [6]. It contains the following elements:

- Packages
- Classes
- Package imports (Private and Public)
- Element imports (Private and Public)

Currently package merges are omitted in the meta-model. Take note that a package diagram model contains **only** information about the **visibility** and no other relationships between packages or classes.

Additionally, the meta-model provides an **interface** to *create*, *delete* and *validate* model elements which is solely based on **fully qualified names**. This way, the interface is independent of the 3rd-party meta-models the package diagram editor has to work with: *Ecore*, *Fujaba* or any other UML-based meta-model can be integrated – identity of elements is provided by the fully qualified names.

The **editor** was developed in a pure model-driven way. In particular, *Fujaba* was used to describe the meta-model – including an interface and behavior to conduct visibility checks

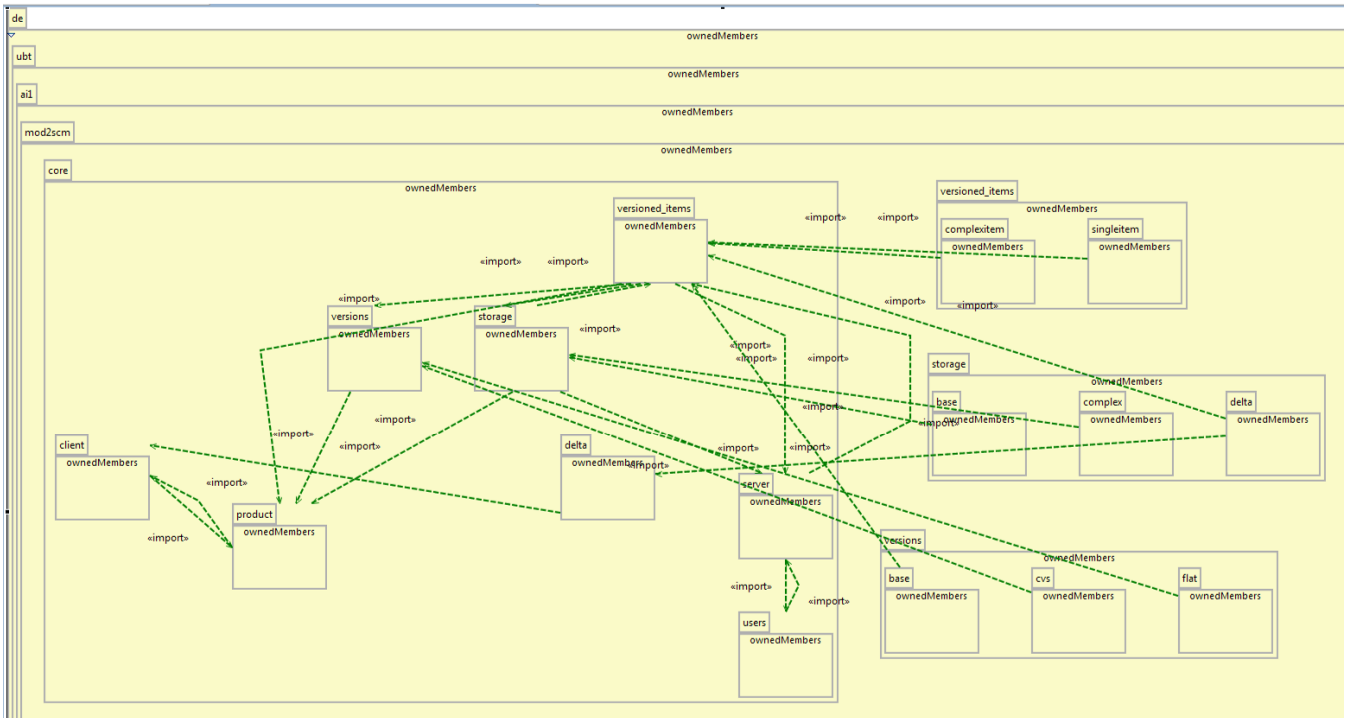

Figure 1: The package diagram of the MOD2-SCM project created with the package diagram editor.

– and exported as *Ecore* model, using Fujaba's code generation engine for the *Eclipse Modeling Framework* (EMF) [5]. Then the *Eclipse Graphical Modeling Framework* (GMF) was used to generate the graphical editor, based on our previous experiences [3]. Figure 1 shows a package diagram of our *MOD2-SCM* project.

3. INTEGRATION INTO EMF AND FUJABA

The ultimate goal of the MODPL package diagram editor is to become integrated with other tools that are based on models containing the concept of package diagrams, but lack a package diagram editor itself: like *Fujaba*. In the following sections we describe the various steps we have undertaken, so far, towards the goal of a seamless integration with Fujaba and EMF.

3.1 Step 1: Deriving package diagrams from Ecore

It is very easy to generate package diagrams on top of an already existing Ecore model definition (see the left dependency in Figure 2). The **import mechanism** reads the Ecore file and creates a corresponding package diagram file containing all packages and classifiers (Classes and Datatypes) from the imported model. Additionally, package or element imports can be deducted from the given Ecore model. After that, the graphical representation of the package diagram can be initialized.

3.2 Step 2: Validating Ecore models against package diagrams

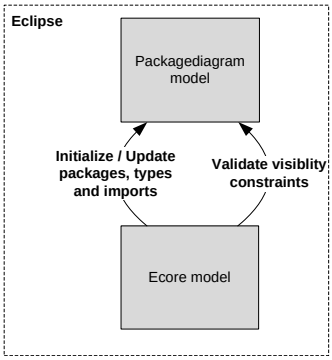

Figure 2: Step 1&2 – Dependencies between Ecore and package diagram models.

The validation checks if for each model element all referenced model elements are visible (see the right dependency in Figure 2). The validation process works as follows: The **source** and **target** element are specified by their fully qualified name, and a request is made – using our meta-model's interface – if the target is visible to the source. This distinction is necessary as visibility is not symmetric – if element A is visible to element B they are not necessarily vice versa.

To keep the package diagram small and readable, we require only the packages to be contained in the package diagram. Classes can be omitted and should be, as long as they are no target of an import. Therefore, four possible results can occur when source and target are validated.

1. **both** elements are contained in the package diagram
2. only the **target** is part of the package diagram
3. only the **source** is part of the package diagram
4. **both** source and target are **not** contained in the package diagram

If an element is not contained in the package diagram, the containing namespace – in most cases a package if no inner class is checked – is used for the validation, instead of the element itself. The fully qualified name of the containing namespace can be computed by simply **truncating** the element's fully qualified name before the **last dot**.

If a violation occurs, we have two possible **origins**:

1. The validated model uses an **non-visible type** and has to be fixed.
2. The package diagram **misses** one or more **imports** which have to be added.

3.2.1 Ecore validation plugin

To validate an Ecore model, we chose to use the *EMF Validation Framework* to trigger the constraint checking and the *Object Constraint Language* (OCL) to specify a constraint on each Ecore element. The OCL statement simply **delegates** source and target name to the appropriate package diagram model method – and does not specify the constraint itself. This allows us to specify the constraint checks in Fujaba without loosing the capability of the Eclipse Validation framework to mark violating model elements visually and to link them with the corresponding error message. During the validation process, the containment tree of the Ecore model is traversed, and the following meta-types of Ecore elements are validated:

- EReference
- EOperation
- EParameter
- EAttribute
- EClass

Source and target are determined as follows: The source is the owner of the Ecore element, whereas the target is its type (i.e. their fully qualified names).

3.3 Step 3: Deriving package diagrams from Fujaba

To create a package diagram for a Fujaba model, it is exported to EMF first, and the generated Ecore model used to initialize the package diagram. This way Ecore acts as **intermediate model** between Fujaba and the MODPL package diagram model (as depicted in Figure 3).

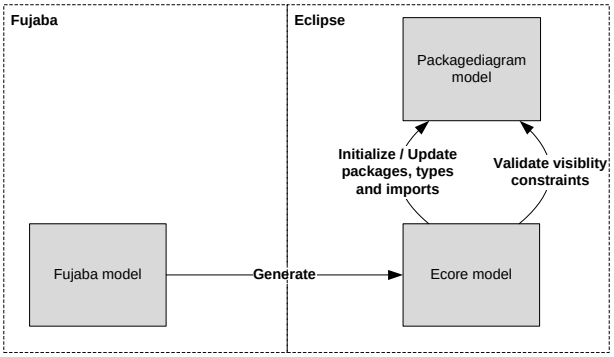

Figure 3: Step 3 – Using Ecore as intermediate model.

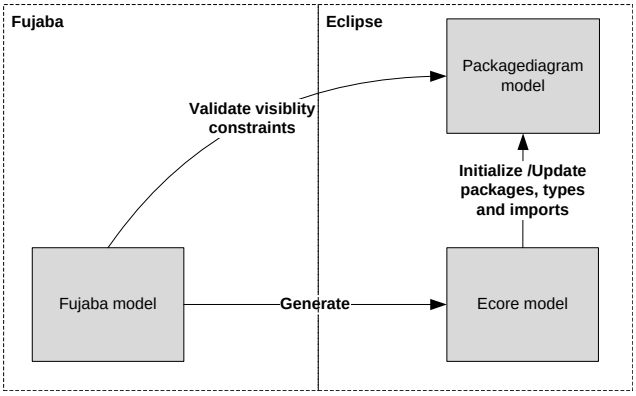

Figure 4: Step 4 – Validating Fujaba models directly.

3.4 Step 4: Validating Fujaba models against package diagrams

By passing the fully qualified names from Fujaba to the corresponding package diagram model, it is possible to check the visibility between two Fujaba elements. Figure 4 depicts the dependencies between the three models: Fujaba, Ecore and package diagram model.

3.4.1 Fujaba validation plugin

A connector has been implemented as a Fujaba plugin, which combines the package diagram editor presented in this paper and *Fujaba4Eclipse* with the *SwingUI*. In the Fujaba tool suite, an existing package diagram model (created with our package diagram editor) is loaded during the editing process. After the model has been loaded, the visibility of model elements is checked when

1. an **association** is created.
2. **attributes** are added.
3. **return types** of methods are selected.
4. **parameters** of methods are specified.
5. the type of an **object** is selected in a story diagram.

The plugin is even capable of updating the package diagram in case a visibility constraint is violated.

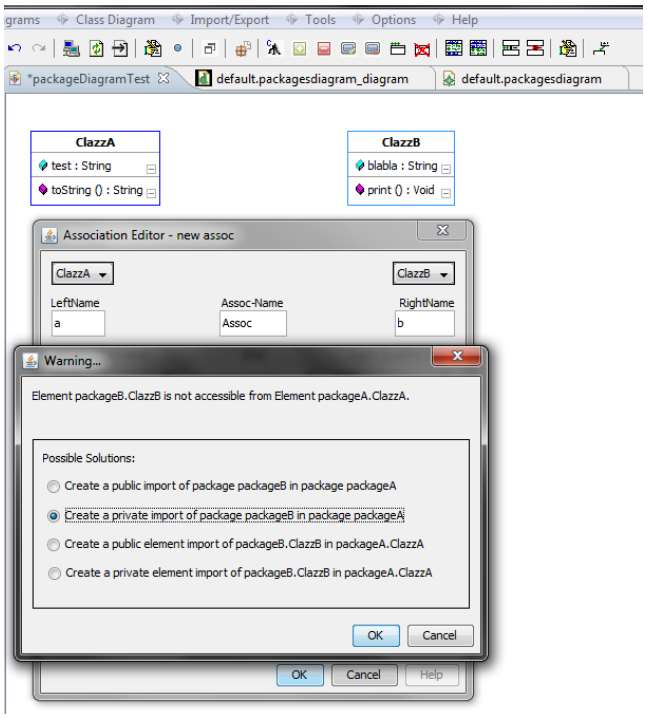

Figure 5: A simple example for the integration of the package diagram editor in Fujaba.

3.5 Step 5: Updating the package diagram from Fujaba

In case a visibility constraint is violated in Fujaba (i.e. a selected element is not visible), the **user** can decide whether he wants to **cancel** the operation in order not to violate the import dependencies specified in the package diagram, or to **add** the appropriate kind of **import** automatically to the package diagram (public or private package import or public or private element import). A simple example is given in Fig. 5: The example consists of two classes, `ClazzA` and `ClazzB`, both defined in package `packageA` and `packageB`, respectively. The corresponding package diagram does not define any imports between the two packages. Now an association is added between `ClazzA` and `ClazzB`. Figure 6 shows the package diagram after the association was added and the user selected the kind of import that should be added automatically to the package diagram (in this case a private package import was used).

The current dependencies between Fujaba and the package diagram model are depicted in Figure 7. Please note that the package diagram editor can be used on existing Fujaba models - but these have be imported using the editor's Ecore import mechanism. Therefore, the Fujaba model has (still) to be exported to Ecore using the EMF code generation. Once the import has been completed (and the package diagram has been associated with the Fujaba model), the validation plugin is used to keep the package diagram up-to-date.

4. RELATED WORK

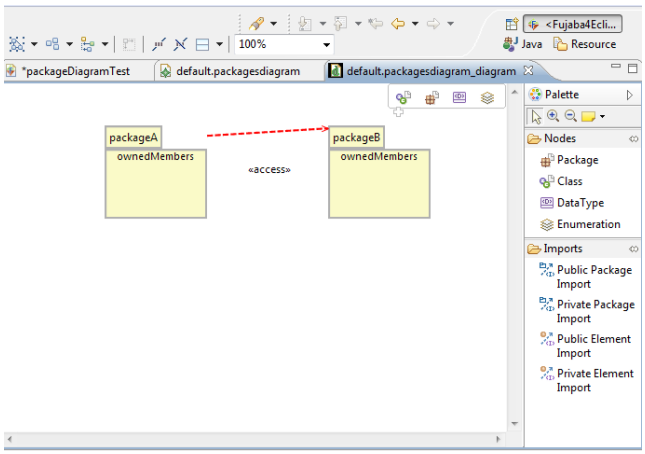

Figure 6: The package diagram after the association was added in Fujaba.

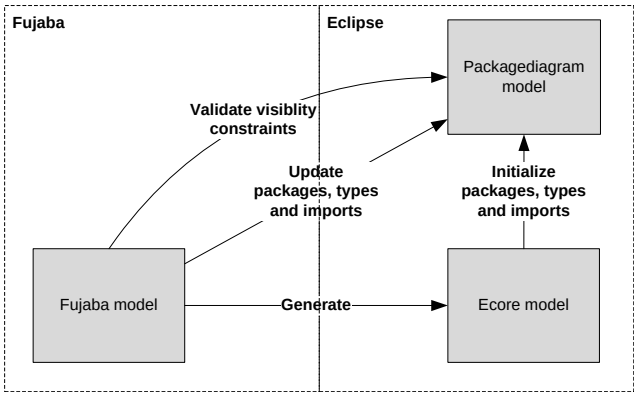

Figure 7: Step 5 – Current state: Updating package diagram models from Fujaba.

Fujaba itself provides only a package tree classes can be assigned to, but no package diagrams. *MOFLON* [2], which has been built on top of Fujaba, offers package diagram capabilities based on the MOF 2.0 meta-model. The MOFLON editor filters the elements based on the visibility constraints. However, it supports, currently, neither the update of the package diagram if a visibility constraint is violated nor the integration of other meta-models. *UML2Tools* for Eclipse provides a mature package diagram editor for UML2 models. These models can be used to generated EMF models so it is possible to use these tools for large EMF models. However, neither does the graphical UML2 editor support package imports (although the metamodel does) nor does the metamodel validation check the visibility constraints. The latter is common behavior for UML package diagram editors: If an element is not visible the fully qualified name is used to access the element – without any warning or error.

5. FUTURE WORK

The next step is the complete integration of our package diagram editor into the Fujaba tool suite (as depicted in Figure 8). The validation plugin has to traverse the Fujaba model and to **create** a corresponding **package diagram** model.

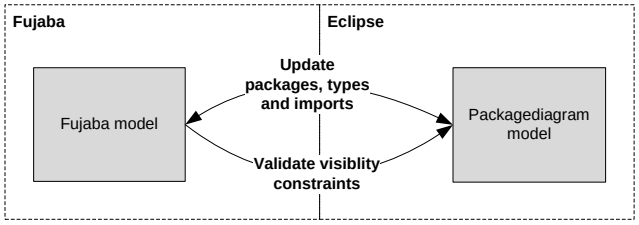

Figure 8: Step 6 – Future work: Removing the intermediate Ecore model.

Further extensions affect the incorporation of the visibility information into the **user interface**: It can be used to limit the referencing of model elements only where they are visible. For example, the class diagram editor can only show elements which are visible from the package the class diagram belongs to. Furthermore, a slight change to the Fujaba meta-model is recommended: not only model elements such as classes and associations, but also class and story diagrams should be assigned uniquely to one package. The package diagram editor itself can also be extended in various ways. For example, besides the view with nested packages, different visualizations could be chosen to provide a more flexible view of packages and their interdependencies to the user.

6. CONCLUSION

In this paper, we presented a package diagram editor developed with Fujaba and GMF. It allows the user to validate Fujaba and Ecore models against package diagrams with import dependencies – and keep them up-to-date. The package diagram can be either created from scratch, or it can be initialized from already existing Ecore models. In the last section, we discussed possible integration points of the package diagram editor into the Fujaba tool suite.

7. ACKNOWLEDGMENTS

We would like to thank our supervisor, Prof. Dr. Bernhard Westfechtel, for the invaluable input during the development of the package diagram editor and his feedback as a reader of this paper.

8. REFERENCES

[1] *OMG Unified Modeling Language (OMG UML), Infrastructure, V 2.1.2*, Nov. 2007. http://www.omg.org/spec/UML/2.1.2/Infrastructure/PDF.

[2] C. Amelunxen, A. Königs, T. Rötschke, and A. Schürr. MOFLON: A standard-compliant metamodeling framework with graph transformations. In A. Rensink and J. Warmer, editors, *Model Driven Architecture - Foundations and Applications: Second European Conference*, volume LNCS 4066, pages 361–375, Genova, Italy, October 2006 2006.

[3] T. Buchmann, A. Dotor, and B. Westfechtel. Model-driven development of graphical tools - fujaba meets GMF. In J. Filipe, M. Helfert, and B. Shishkov, editors, *Proceedings of the Second International Conference on Software and Data Technologies (ICSOFT 2007)*, pages 425–430, Barcelona, Spain, July 2007. INSTICC Press, Setubal, Portugal.

[4] T. Buchmann, A. Dotor, and B. Westfechtel. Experiences with modeling in the large with fujaba. In U. Assmann, J. Johannes, and A. Zündorf, editors, *Proceedings of the 6th International Fujaba Days*. University of Dresden, University of Dresden, 2008.

[5] L. Geiger, T. Buchmann, and A. Dotor. EMF code generation with fujaba. In L. Geiger, H. Giese, and A. Zündorf, editors, *Proceedings of the 5th International Fujaba Days*, 2007.

[6] OMG. *OMG Unified Modeling Language (OMG UML), Superstructure*. OMG, November 2007. Version 2.1.2.