Comput Sci Res Dev (2014) 29:73–93
DOI 10.1007/s00450-011-0201-1

REGULAR PAPER

Model-driven software engineering: concepts and tools for modeling-in-the-large with package diagrams

Thomas Buchmann · Alexander Dotor · Bernhard Westfechtel

Received: 9 July 2010 / Accepted: 19 August 2011 / Published online: 17 September 2011

Abstract Model-driven software engineering intends to reduce development effort by generating executable code from high-level models. However, models for non-trivial problems are still very large and require sophisticated support for modeling-in-the-large—a challenge which has not yet gained sufficient attention in model-driven software engineering. In response to this need, we have developed a set of tools for modeling-in-the-large based on UML package diagrams. The tool set comprises a stand-alone tool, which serves as a graphical editor for package diagrams, an integration tool, which controls the consistency of the model with its architecture defined in the package diagram, and a metrics tool, which is used to measure the coupling of model packages. Our tools for modeling-in-the-large complement the functionality of an environment for model-driven software engineering which so far has focused on supporting modeling-in-the-small. The overall environment has been applied successfully in a project which is dedicated to model-driven engineering of a product line for software configuration management systems.

Keywords Model-driven development · Modeling-in-the-large · Package diagrams · Incremental round-trip engineering

1 Introduction

Model-driven software engineering intends to reduce development effort by generating code from high-level models. For example, many tools support the generation of code from class diagrams. However, class diagrams define only a structural model. Thus, the code generated from class diagrams merely provides implementation frames which have to be completed by programming method bodies. Behavioral models may be defined e.g. with statecharts, sequence diagrams, or activity diagrams. *Executable* code may be obtained by generating code from behavioral models. Only then is model-driven engineering supported in a full-fledged way.

So far, research on model-driven software engineering has focused on *modeling-in-the-small*, with a particular emphasis on the definition of *model transformations* [44]. However, models for non-trivial problems are still very large and require sophisticated support for *modeling-in-the-large* [4]. The problem of organizing software at the architectural level simply does not go away with model-driven software engineering [30].

We have experienced this problem in the *MOD2-SCM* project, which is dedicated to the development of a model-driven and modular product line for software configuration management [12, 19]. With MOD2-SCM, a software engineer defines the desired features of a version control system with the help of a feature model. In this way, a feature configuration is created which is used to configure an executable domain model. A major challenge of the MOD2-SCM project consists in the design of a model architecture being composed from loosely coupled modules such that orthogonal features actually can be combined independently. To make this work, the interfaces between the modules of the architecture have to be controlled carefully.

So far, modeling-in-the-large has not received sufficient attention in model-driven software engineering. For example, we have been using *Fujaba* [50], an object-oriented tool for model-driven software engineering, which provides ex-

T. Buchmann (✉) · A. Dotor · B. Westfechtel
Angewandte Informatik I, Universität Bayreuth, 95440 Bayreuth, Germany
e-mail: thomas.buchmann@uni-bayreuth.de

cellent support for modeling-in-the-small, including generation of executable code from structural and behavioral models. Unfortunately, the tool designers have not seen the need for tools for modeling-in-the-large [11]. As a consequence, we experienced severe problems in the MOD2-SCM project, in which we soon lost track of the overall organization of the domain model.

Motivated by the requirements of the MOD2-SCM project, we therefore decided to develop a tool set for modeling-in-the-large which complements the modeling-in-the-small support provided by Fujaba (see [9] for a short sketch of a preliminary version of this tool set). Using the tool set, the overall organization of a model may be represented with the help of UML 2 *package diagrams*. An *integration tool* assists the modeler in controlling the consistency of the domain model with its architecture defined in the package diagram. The integration tool supports forward engineering, reverse engineering, validation, constrained editing, and incremental synchronization. Furthermore, a *metrics tool* provides metrics for the coupling and cohesion of model packages. By applying the metrics tool, the modeler may examine the modularity of the model architecture and detect problematic parts of the architecture which require further attention.

The rest of this paper is structured as follows: Sect. 2 puts our work on modeling-in-the-large into the context of our overall research on model-driven software product line engineering. Section 3 summarizes previous work on modeling-in-the-large and related topics such as programming-in-the-large, module interconnection languages, software architectures, and architecture description languages. In our work, we have decided to use package diagrams, which are introduced briefly in Sect. 4 (along with a motivating example of its use from the MOD2-SCM project). Section 5, which constitutes the core part of this paper, describes our support tools for modeling-in-the-large. Section 6 briefly describes the realization of the tool set. Section 7 explains how we applied the tool set in the MOD2-SCM project. Section 8 discusses related work on tools for modeling-in-the-large with package diagrams. Section 9 concludes the paper.

2 Background

2.1 Model-driven software product line engineering

The work presented in this paper is embedded into overall research on *model-driven software product line engineering* [33]. In the context of this research, we developed an *environment* for model-driven software product line engineering which is depicted in Fig. 1 [7]. The environment is composed of both external, reused tools (grey rectangles) and self-developed tools (orange rectangles). In this paper, we will focus on the tools for modeling-in-the-large (see the ellipsis at the upper right corner). In the following, we will give a brief overview of the overall environment to put our work into context.

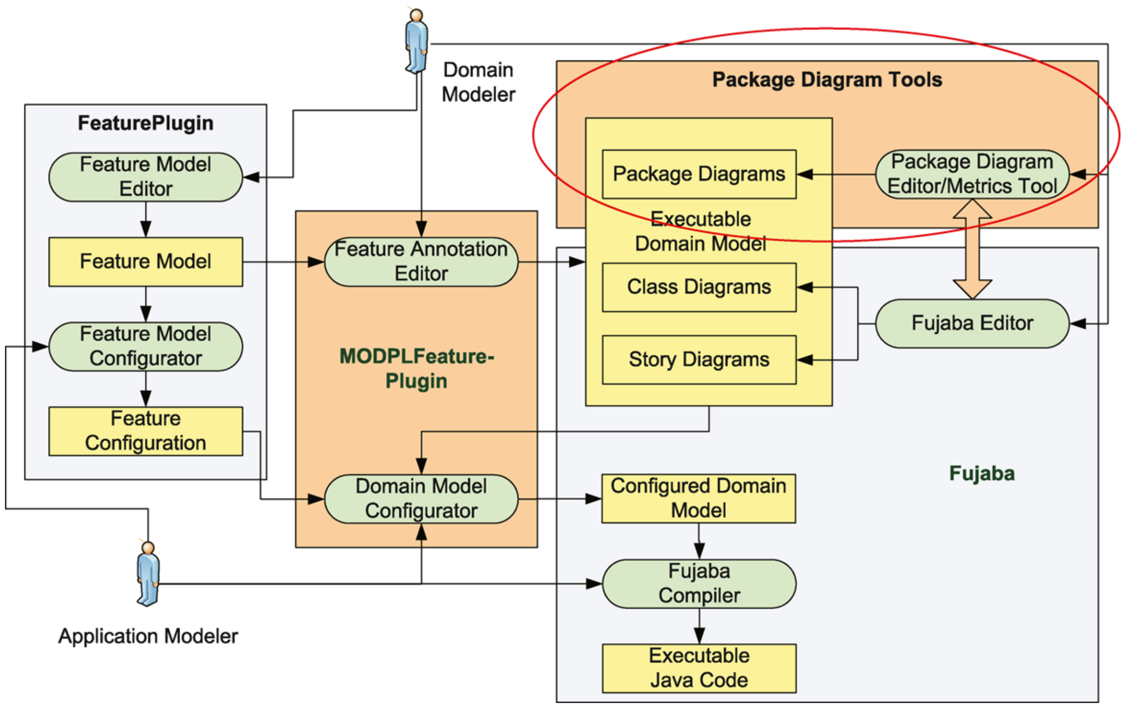


Fig. 1 Environment for model-driven engineering of software product lines

The *domain modeler* creates a *feature model* [13], which describes common and discriminating features of instances of the software product line. To this end, the tool FeaturePlugin [3] is used. Furthermore, the domain modeler produces an *executable domain model*, which covers all product variants, with the help of Fujaba and the tools described in this paper. Features are mapped onto elements of the domain model with the help of MODPLFeaturePlugin, which is described in [7].

The *application modeler* is responsible for configuring an instance of the product line. Using FeaturePlugin, a *feature configuration* is derived from the feature model by selecting those features which are required for the respective product instance. Subsequently, a *configured domain model* is created by assembling all elements of the overall domain model for the product line which contribute to the realization of the features selected in the feature configuration. Finally, the Fujaba compiler is used to transform the configured domain model into executable Java code.

2.2 Fujaba

Fujaba is an environment which supports model-driven software engineering with the help of class diagrams and story diagrams. UML-like *class diagrams* are used for structural modeling. *Story diagrams* are employed for behavioral modeling: A method defined in a class may be realized with the help of a story diagram. The Fujaba compiler generates Java code from a model being composed of class and story diagrams. For each class in the model, a corresponding Java class is generated. For each method realized by a story diagram, a method body is generated into the respective Java class. If all methods of the model have been realized by story diagrams, the generated code is fully executable.

A story diagram resembles an interaction overview diagram as introduced in UML 2. The control flow is defined by an activity diagram which contains nodes of two kinds: A *statement activity* consists of a fragment of Java code, allowing for seamless integration of textual and graphical programming. A *story pattern* is a communication diagram which is composed of objects and links. A story pattern containing only static elements constitutes a graphical query. A story pattern may also contain dynamic elements for creating/deleting objects and links, modifying attribute values, and calling methods.

As it stands, Fujaba provides virtually no support for modeling-in-the-large. A model is presented as a collection of class and story diagrams. In the case of large models, the user easily loses track of the overall model architecture. Each class diagram shows a local cutout of the overall model, but there is no way to figure out how the class diagrams are related to each other. For this reason, we have developed the tools for modeling-in-the-large described in this paper.

2.3 MOD2-SCM

In the MOD2-SCM project, the environment illustrated in Fig. 1 is used to develop a product line for software configuration management. *Software configuration management* (SCM) is the discipline of controlling the evolution of large and complex software systems. A wide variety of SCM tools and systems has been implemented, ranging from small tools such as RCS [42] over medium-sized systems such as CVS [46] or Subversion [15] to large-scale industrial systems such as Adele [22] and ClearCase [49]. These systems are based on similar concepts, e.g., in each of the cited systems version control is based on some variant of version graphs. Furthermore, implementing these systems requires significant effort.

These observations have motivated us to launch a project for developing a *model-driven and modular SCM system*:

- The system is based on an *explicit domain model* for SCM. This makes it easier to communicate and reason about the model.
- The model is *executable*. Thus, development effort is reduced by generating code from the model.
- The system is designed to support a *product line* for SCM systems. To this end, the executable domain model is composed from *loosely coupled modules* which may be configured by defining the *features* of the respective target system.

In the following we will give examples for our model-driven software engineering approach in order to demonstrate how the loose coupling is achieved both in structural and behavioral modeling with Fujaba. This approach is illustrated by the *storage model*, which is concerned with the space efficient storage of versions of product items. The storage model is independent of both the history model (version graphs representing the evolution of versioned items) and the product model (files, EMF models, etc.). Here, we consider only *directed deltas*: A directed delta is a sequence of operations which, when applied to some input version v, produces some output version v'.

Figure 2 shows a *class diagram* for the storage model. Generic interfaces are displayed at the bottom. The interfaces IDeltifiableItem and IDelta introduce methods for comparing items and applying deltas which have to be implemented by the product model. Furthermore, IStorage defines methods for storing and retrieving items. The class DeltaStorage and its subclasses implement these methods. A delta storage stores either baseline items or delta items. Its subclasses, which are annotated by features from the feature model, realize different variants of applying deltas.

An example of a *story diagram* is given in Fig. 3. At the control flow level, the notation of activity diagrams is used. A statement activity (node 5) contains a text fragment of

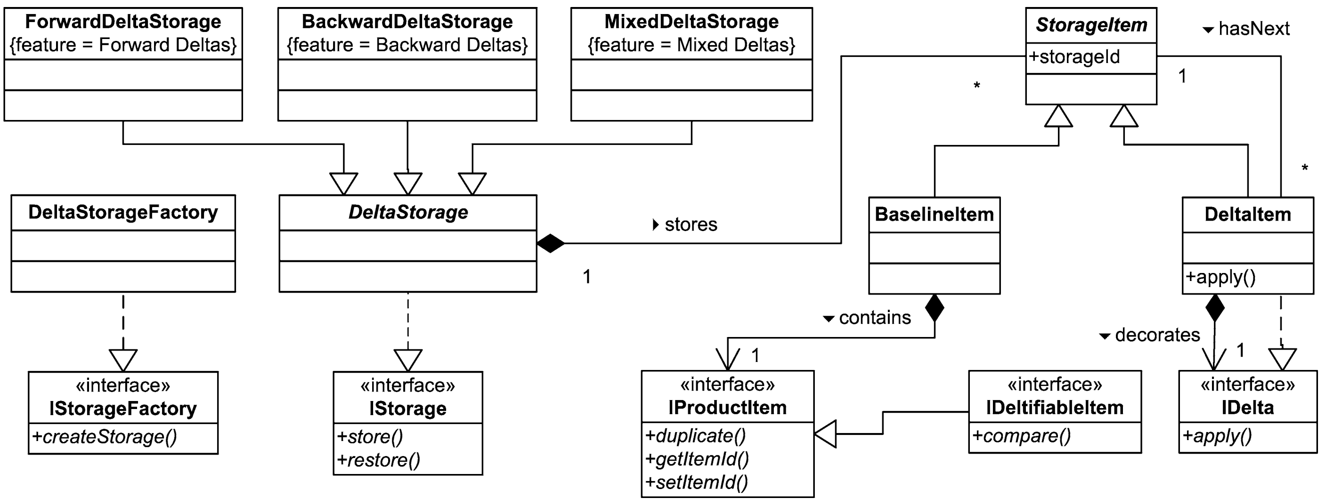


Fig. 2 Class diagram for the storage model (directed deltas)

Java code. A story pattern is composed of objects and links. Elements with dashed lines represent optional parts of story patterns. A crossed element denotes a negative application condition. In addition to method calls, a story pattern may describe structural changes: Objects and links to be created or deleted are decorated with the stereotype <<create>> (green) or <<destroy>> (red), respectively. Furthermore, := and == denote attribute assignments and equality conditions, respectively.

The diagram shows the method store() of class MixedDeltaStorage. The method may be executed only if the item to be stored is deltifiable (1). The first item is stored as a baseline (2). Otherwise, the method terminates if the supplied identifier has already been used for a stored item (3). The context parameter must identify a predecessor item relative to which the new item is stored (4). In case of success, the predecessor item is restored to prepare the comparison (5). If the predecessor item was stored as a delta (6), the new item is stored as a forward delta (7). Otherwise, the new item is stored as a baseline, and the predecessor item is stored as a backward delta (8).

3 Modeling-in-the-large

The term *modeling-in-the-large* was introduced in [4] in analogy to *programming-in-the-large*, which was coined in [17] in 1976. In the latter paper, it was argued that “structuring a large collection of modules to form a system is an essentially distinct and different intellectual activity from that of constructing the individual modules”. To support this activity, *module interconnection languages* [37] were developed which describe the composition of systems from a set of modules. Modules are considered as entities which provide and require resources (e.g., methods and data objects). Module interconnection languages focus on static relationships among modules such as nesting and import relationships.

The discipline of *software architecture* is concerned with the large-scale organization of software systems [31, 38]. Unfortunately, there is no generally accepted definition of the term “software architecture”. A useful definition is given in [5], where an architecture is defined as “the set of significant decisions about the organization of a software system, the selection of the structural elements and their interfaces by which the system is composed, together with their behavior as specified in the collaborations among those elements, the composition of these structural and behavioral elements into progressively larger subsystems, and the architectural style that guides this organization”.

For the notation of software architectures, *software architecture description languages* [14, 32] have been developed which go far beyond the capabilities of module interconnection languages. Many of these languages are built on components and connectors, as exemplified e.g. by [1]. The description encompasses not only the structure, but also the behavior, for which various formal notations have been developed. In addition, some architecture description languages provide for multiple views.

Object-oriented modeling languages have also been used to describe software architectures. For example, in the UML the structure of a software system may be described by package, class, or component diagrams. Furthermore, state diagrams may be used to describe the behavior of classes or components, and the interaction among components may be described with the help of sequence diagrams.

To some extent, architecture description languages and object-oriented modeling languages have evolved independently. However, several attempts have been made to “join

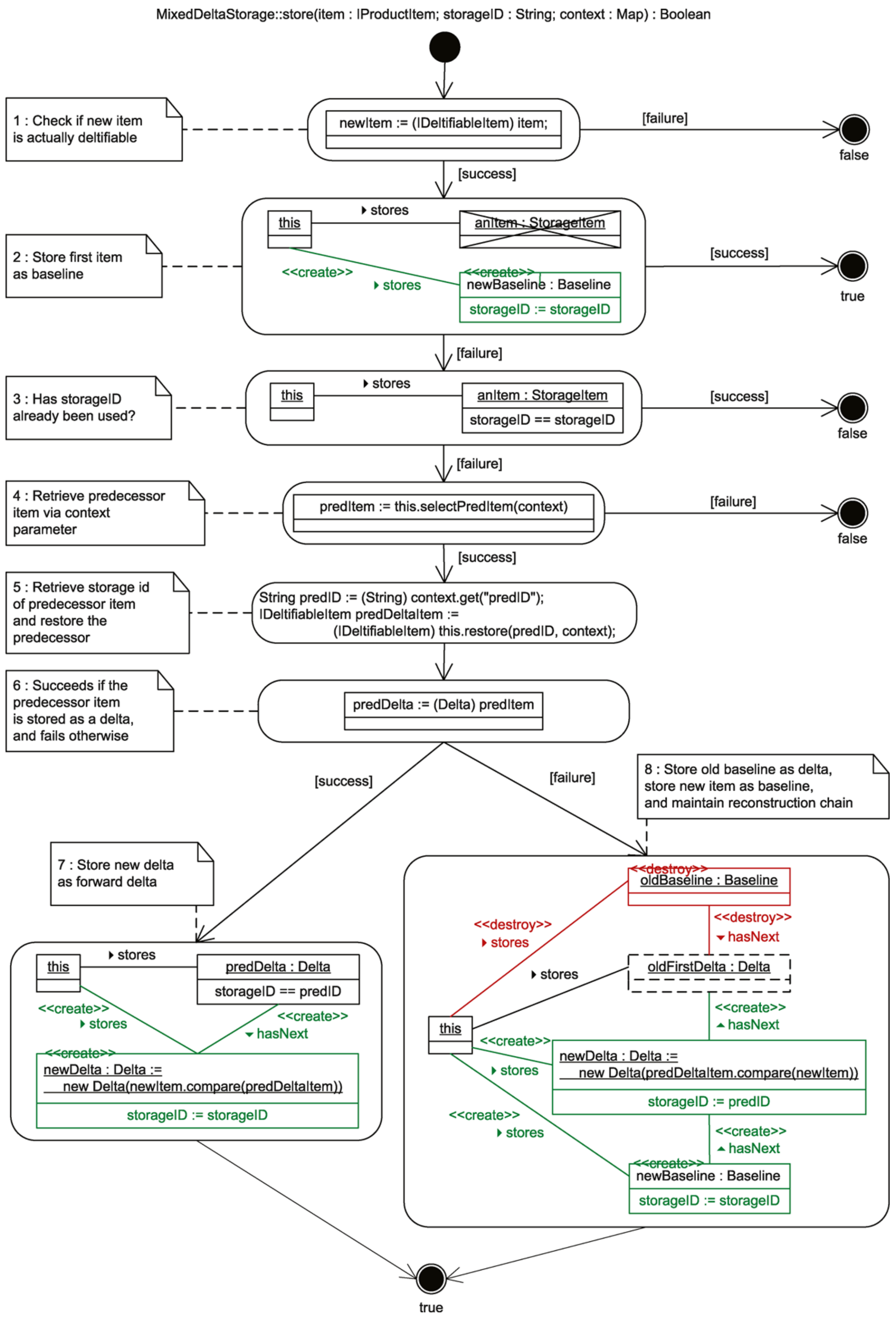

MixedDeltaStorage::store(item : IProductItem; storageID : String; context : Map) : Boolean
1 : Check if new item is actually deltifiable
newItem := (IDeltifiableItem) item;
[failure]
false
[success]
2 : Store first item as baseline
this
stores
anItem : StorageItem
<<create>>
stores
<<create>>
newBaseline : Baseline
storageID := storageID
[success]
true
[failure]
3 : Has storageID already been used?
this
stores
anItem : StorageItem
storageID == storageID
[success]
false
[failure]
4 : Retrieve predecessor item via context parameter
predItem := this.selectPredItem(context)
[failure]
false
[success]
5 : Retrieve storage id of predecessor item and restore the predecessor
String predID := (String) context.get("predID");
IDeltifiableItem predDeltaItem := (IDeltifiableItem) this.restore(predID, context);
6 : Succeeds if the predecessor item is stored as a delta, and fails otherwise
predDelta := (Delta) predItem
[success]
[failure]
8 : Store old baseline as delta, store new item as baseline, and maintain reconstruction chain
7 : Store new delta as forward delta
this
stores
predDelta : Delta
storageID == predID
<<create>>
stores
<<create>>
hasNext
<<create>>
newDelta : Delta := new Delta(newItem.compare(predDeltaItem))
storageID := storageID
<<destroy>>
oldBaseline : Baseline
<<destroy>>
stores
<<destroy>>
hasNext
stores
oldFirstDelta : Delta
this
<<create>>
hasNext
<<create>>
stores
newDelta : Delta := new Delta(predDeltaItem.compare(newItem))
storageID := predID
<<create>>
stores
<<create>>
hasNext
<<create>>
newBaseline : Baseline
storageID := storageID
true

Fig. 3 Story diagram for storing an item (mixed deltas)

the forces". For example, in [26, 29] software architectures are expressed in the UML with the help of *UML profiles*.

In the context of *model-driven software engineering*, the problem of scalability and the need for the modular organization of large models has been recognized some time ago, but only recently it has been addressed [4, 30]. Several approaches for describing *model architectures* have been developed [16, 25, 45, 47, 48]. Modeling-in-the-large still requires further investigation, both at the conceptual level and the level of tool support.

4 Package diagrams

4.1 Motivation

In the MOD2-SCM project, we lost track of the overall organization of the executable domain model very soon. Currently, the domain model contains about 80 class diagrams. Using Fujaba as it stands, these diagrams are displayed in a flat list without any indication of how these diagrams are related to each other. In this way, it is impossible to obtain an overview of the model's organization. Therefore, we were looking for an approach which allows us to structure the model into modules implementing features of the product line for version control systems, and to control the dependencies among these modules such that orthogonal features are decoupled from each other. Please notice that it was not our ambition to develop a novel language for architectural modeling. Rather, we intended to stick closely to the UML standard [35, 36]. Furthermore, we were looking for an architectural modeling language which is simple, lightweight, and easy to apply (sacrificing expressive power as provided by several architecture description languages).

Under these prerequisites, UML *package diagrams* appeared to be a natural choice. Along the lines of the previous section, the language for package diagrams is classified as a module interconnection language rather than an architecture description language (it focuses on the static structure, which is described with the help of containment and import relationships). For an informal introduction, we specifically recommend Chap. 7 of the book on the Catalysis method [20], which also provides methodical guidance in applying packages to structure large models. A brief summary of UML packages referring to the current version of the UML specification is given below.

4.2 The package concept of UML 2

Let us briefly recall the concepts which UML 2 offers for structuring large models: A model may be structured into hierarchically organized *packages*. Each model element is owned by exactly one package. *Private elements* may be accessed only in their owning package, while *public elements* may be accessed in other packages. 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.

Imports serve to extend the set of those elements which may be referenced without qualification. An import which ends at an element of a package is called *element import*. It extends the namespace of the importing element with the imported element (e.g., a class which may be used without qualification in the importing namespace). An import which ends at a package is called *package import*. It is equivalent to the set of element imports referring to all public elements.

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. Public and private imports are denoted by the stereotypes <<import>> and <<access>>, respectively.

There are fundamental differences between imports in UML 2 and imports in *modular programming languages* such as Modula-2 and Ada. In these languages, each—separately compiled—program unit (called module in Modula-2 and package in Ada) may reference only its own local declarations unless the namespace is extended by an import. Depending on the kind of import, imported elements may be referenced with or without qualification. Furthermore, an imported element may only be *used* but not modified. In contrast, the UML 2 standard states ([35], p. 144): "An element import ... works by reference, which means that it is not possible to add features to the element import itself, but it is possible to modify the referenced element in the namespace from which it was imported". For example, an imported class may be inserted into a class diagram of the importing package and extended with attributes, methods, and associations; these extensions apply to the imported package where the imported class is declared.

Apart from nesting and imports, the UML offers *package merges*, which are heavily used in the definition of the UML metamodel. Package merge is not covered by our tool set for the following reasons: First, the semantics of package merge are complex; second, package merge is defined only for the structural model (class diagrams). For a discussion of package merge, the reader is referred to [18].

4.3 Example

Figure 4 displays a package diagram (using only public package imports) from the MOD2-SCM project. We will describe this diagram briefly below since it will serve as running example in Sect. 5. The package diagram shows a significantly simplified cutout of the actual model architecture.

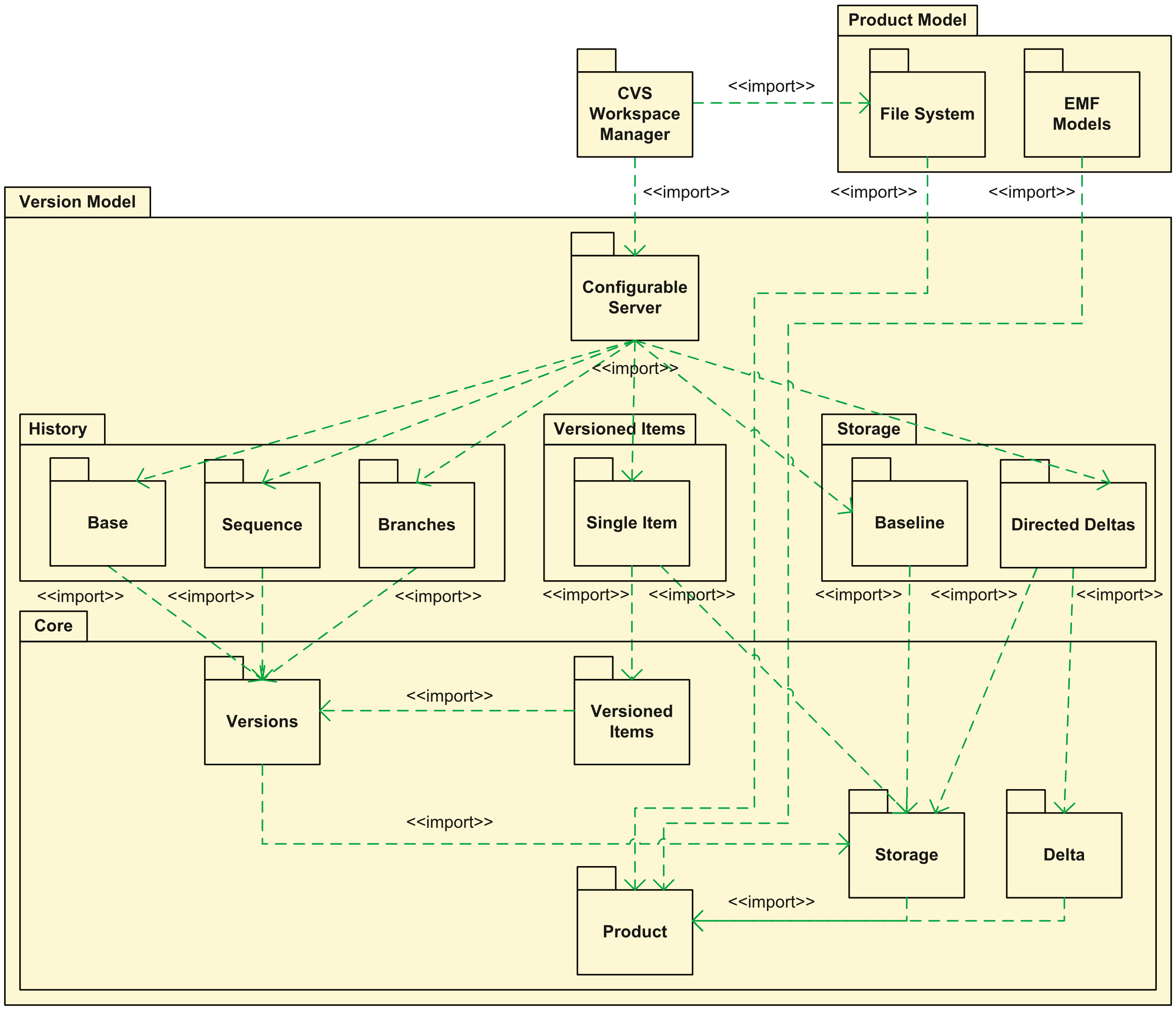


Fig. 4 Package diagram from the MOD2-SCM project (simplified)

A detailed description of the model architecture goes beyond the scope of this paper; see [19].

The package Version Model is dedicated to the version model underlying the repository of an SCM system. The nested package Core provides basic, generic functionality on top of which higher-level packages are built. Discriminating features are introduced above the Core. For the History, we may select among the alternatives Base (a set of versions without history relationships), Sequence (a single sequence of versions), and Branches. For the Storage, the alternatives Baseline (independent storage of all versions) and DirectedDeltas (storage relative to an adjacent version) are supported. Finally, Configurable Server provides a uniform interface to the server managing the repository. The package Version Model does not depend on a specific product model. In Product Model, two sample alternatives (file systems and EMF [41] models) are provided by respective subpackages. Finally, CVS Workspace Manager offers a CVS-like workspace manager, relying on the file system, branches, and mixed deltas.

While a detailed understanding of this architecture is not required in the context of this paper, it is important to note that the success of the MOD2-SCM project heavily depends on a carefully designed architecture. In particular, it is crucial to control the dependencies between different components. For example, the architecture has been designed such that the history model and the storage model may be combined orthogonally. This is reflected in the feature model, where the alternative features for the history model and the storage model may be selected independently. However, in an early version of the domain model the implementation of directed deltas depended on the history model. This dependency, which effectively made independent selection of history and storage features impossible, was revealed by architectural analysis and subsequently removed by a redesign.

5 Tool support

5.1 Overview

Figure 5 provides an overview of our tool support for modeling-in-the-large. Our package diagram editor was developed to be used as a universal tool which can either act as a pure stand-alone editor for package diagrams or that can be used in conjunction with either Ecore [41] or Fujaba [50] models. As mentioned earlier, we were interested primarily in extending the modeling tool used in the MOD2-SCM project—Fujaba—with modeling-in-the-large. However, the package diagram editor was built in such a way that it can be used independently from Fujaba in other projects relying on EMF technology.

The *package diagram editor* supports graphical editing of package diagrams with a focus on visualizing the dependencies between packages by providing all import mechanisms which are specified in the UML superstructure specification [36]. Its functionality and user interface will be described in this section.

Providing just a stand-alone editor would support the modeling process only to a severely limited extent. Therefore, it is crucial to offer an *integration tool* for maintaining consistency between modeling-in-the-large and modeling-in-the-small. Integration with the package diagram editor assists the modeler in various ways. The integration functions may be classified into six groups:

Forward engineering The skeleton of a model is generated from a package diagram. This is done as a *batch process*, i.e., the whole package diagram is compiled into a model.

Reverse engineering A package diagram is created from a model. Again, this function is performed as a batch process.

Validation A model is validated against a package diagram. Validation is performed on demand.

Constrained editing The model editor is constrained in such a way that only visible elements may be used during editing.

Synchronization In contrast to batch forward and reverse engineering, synchronization is performed *incrementally* between the model and the package diagram. Updates may be propagated in both directions (*round-trip engineering*).

Metrics calculation The degree of coupling between packages can be calculated based on metrics. The degree of coupling between package diagram elements is represented by the thickness of the line depicting an import relationship between them.

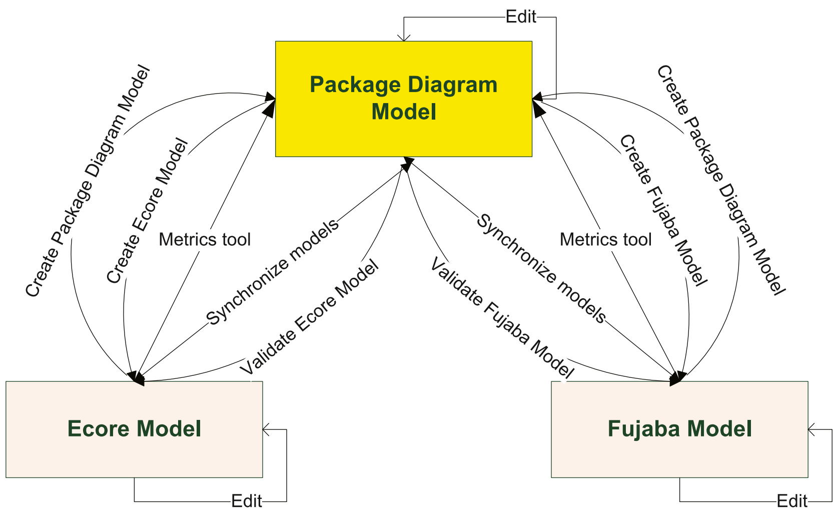
The integration tool was implemented not only for Fujaba models, but also for Ecore models. Please notice that the Ecore metamodel defines only package hierarchies. Imports are not covered by the metamodel; any element may be used in any package. Thus, using Ecore alone does not provide adequate control of dependencies among packages. In the case of Ecore, constrained editing has not been implemented (standard Ecore editors are used without modifications).

5.2 Functionality

5.2.1 Stand-alone tool

The *stand-alone tool* deals with package diagrams representing the package hierarchy as well as import relationships. Within the area for displaying a package, elements declared in this package are shown. These elements comprise nested packages and owned types (classes, data types,

Fig. 5 Tool overview



and enumerations). Please note that types are included into a package diagram at the discretion of the modeler. This means that the package diagram editor may be used at a coarse-grained level, showing only the relationships among packages as in Fig. 4, or in a more fine-grained way by also listing some or all elements declared in a package. In the latter case, only a list of elements is shown because it is not intended to replicate the functionality of a class diagram editor.

The package diagram editor was designed as a plug-in for the Eclipse platform. Figure 6 shows a screenshot of our tool. The main editing area is displayed in the center of the screen and it is used to edit the package hierarchy and to specify visibility constraints by defining public or private imports of packages or elements. To create new elements, the *palette* which is shown on the right side can be used. Several nodes (packages, classes, data types, and enumerations) can be added to the diagram. The different import types can be added as edges. Packages can be nested to build up a package hierarchy. On the bottom left side, an *overview diagram* is shown which allows easier navigation in package diagrams, especially if they describe a model for a non-trivial problem, which often results in very large models. The property view at the bottom center shows all namespaces which are *accessible* from the currently selected element, whereas the bottom right shows a tree view which displays the *package hierarchy* as a collapsible tree. Accessible packages are highlighted in blue color in this view.

In the current example, the package wsmanager_cvs (on the top of the diagram) is selected in the editor. All packages that are accessible through imports are highlighted. Please note that also the enclosing packages and their public elements are visible, but they are not highlighted to increase the clarity. The package wsmanager_cvs defines two public package imports (to server and to product_model.filesystem). Outgoing imports of the selected package are visualized using solid lines, whereas all other import dependencies in the diagram are shown in dashed lines. In that way, it is easier to see which import dependencies originate from the selected element, especially when the package diagram contains a large number of packages and imports. Since only public package imports are used in the example, and due to the transitivity of public imports, all highlighted packages are visible to wsmanager_cvs as well, as they are all connected by public package imports.

As models for non-trivial problems are often very large, it is obvious that the number of imports can reach a level where it becomes very hard for the modeler to keep track of them. In the editor, the modeler may use *public imports* to reduce the number of dependencies shown in the diagram. In addition, the editor provides *reduction algorithms* which further reduce the number of public imports. All of these reduction algorithms exploit the transitivity of public imports. For example, if all subpackages of a common parent import the same target, these imports may be replaced by a single import emanating from the parent package. This is illustrated in Fig. 7, where the imports from the subpackages versions.base, versions.branches and versions.sequence to the package core.versions are replaced by a single public package import from the enclosing package versions to package core.versions. Other variants of reductions are also available. For example, imports may be moved upwards if the number or percentage of importing child packages exceeds a certain threshold.

5.2.2 *Consistency rules*

Before describing the functionality of the integration tool ordered by the functional groups introduced in Sect. 5.1, we list some *consistency rules* which serve as the foundation for deriving the tool's functionality:

1. The package hierarchies are identical in the model and the package diagram, i.e., the respective trees are isomorphic, and corresponding packages have the same name.
2. The set of elements (classes) defined in the package diagram is a subset of the elements defined in the model. Corresponding elements have the same name and are owned by the same package.
3. Each element which is used in the model is visible with respect to the package diagram.

Please note that the second rule does not require equality of the element sets because elements are considered optional in the package diagram (the user has to decide how many details are shown in the package diagram). Furthermore, applying the third rule to Fujaba is possible only when it is known where an element is used. As it stands, Fujaba merely maintains a flat list of all class diagrams defined in the model. Therefore, we have extended Fujaba by supporting the assignment of class diagrams to owning packages.

5.2.3 *Forward engineering*

The integration tool offers the possibility to create a skeleton of a Fujaba or Ecore model from an existing package diagram model. A new target model file is created, and all packages and possible types (classes, enumerations or data types) are added. The packages are created with their respective names in the package diagram and also an identical package hierarchy is established. Types are created with an identical name and are assigned to the same package.

5.2.4 *Reverse engineering*

Reverse engineering of Fujaba or Ecore models allows to either create a plain package hierarchy, or to also deduce

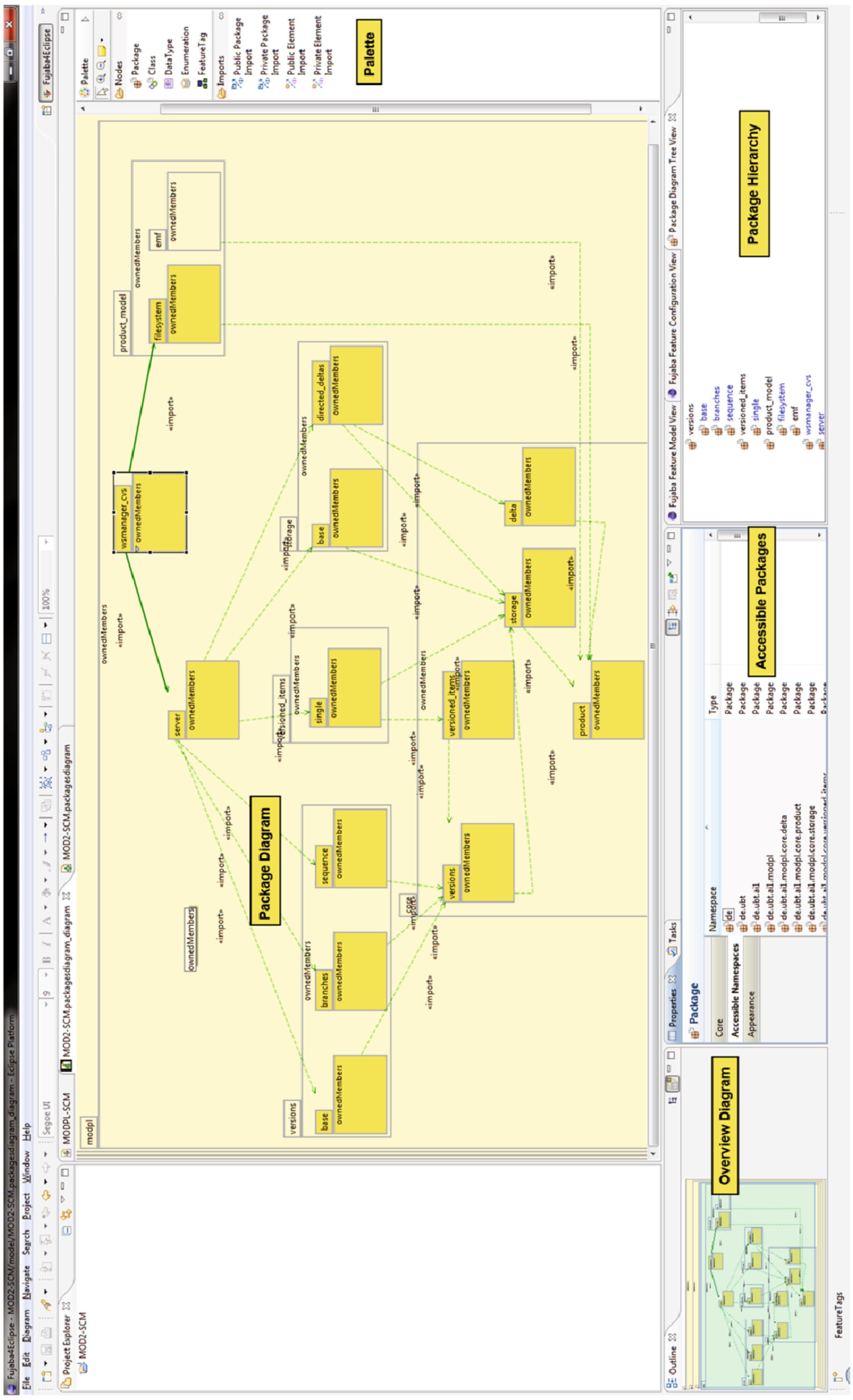


Fig. 6 Screenshot of the package diagram editor

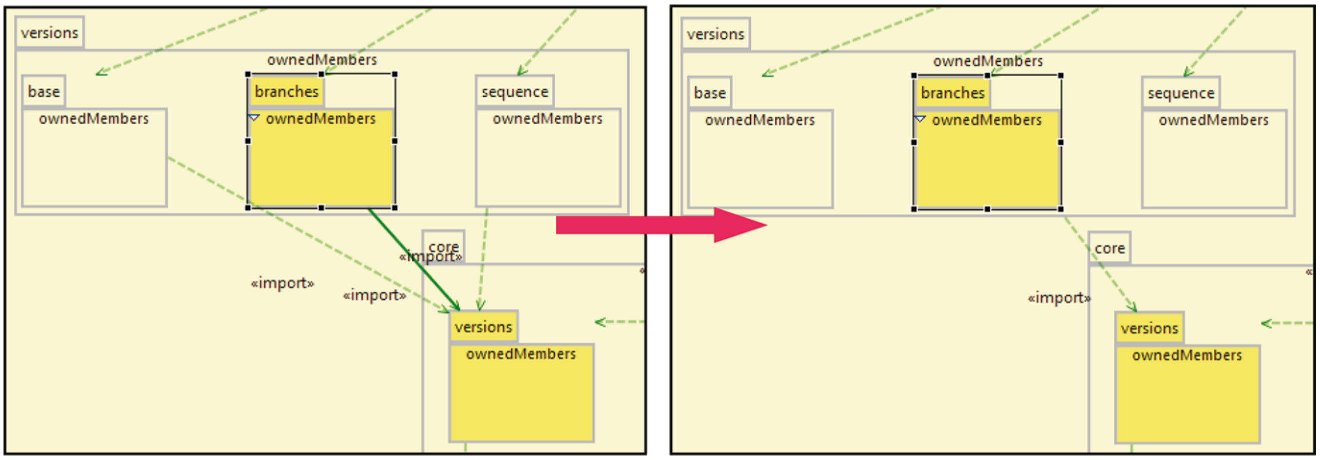


Fig. 7 Reduction of imports

all imports that are necessary to get a valid model. In order to compute the required import dependencies, the modeler has to choose the desired import type (e.g. public package imports). The required imports are determined by checking the visibility of types of superclasses, attributes, method parameters, method return values, and association roles for all classes in the source model. In addition, in the case of Fujaba applied occurrences of types in story diagrams are also taken into account (e.g., types of objects in story patterns). In case a referenced type is not visible, a corresponding import (of the previously selected type) is created in the package diagram.

5.2.5 *Validation*

As mentioned in Sect. 5.1, the model and the diagram may be edited asynchronously, resulting in inconsistencies between them. In particular, violations of the visibility constraints defined in the package diagram can occur in many ways:

– by defining generalizations to non-visible types,
– by using non-visible types as attributes in classes,
– by using non-visible types as return values of methods or as method parameters,
– by defining associations between non-visible types, and
– by defining objects of non-visible types in story patterns.

The validation mechanisms detect and highlight model elements which violate the visibility constraints. In the case of Fujaba, validations refer to both the structural model (class diagrams) and the behavioral model (story diagrams). As an example for validating the structural model, Fig. 8 shows a class diagram in Fujaba which contains several model elements violating the visibility constraints specified in the corresponding package diagram (shown in red color). In the current example, four different types of violations are shown:

1. AbstractDeltaStorage inherits the non-visible class AbstractStorage.
2. The non-visible type IDeltifiableItem is used as parameter in several methods.
3. Several methods like restore from AbstractDeltaStorage use non-visible types in their respective method implementations (specified using Fujaba's story diagrams).
4. The non-visible type IDeltifiableItem is used as a return type in method apply of class Delta.

Furthermore, the integration tool for Fujaba also provides so called *Quick Fix* functions to add new import dependencies to the package diagram based upon the user's choice in order to make the referenced element visible.

In the case of Ecore, validation refers to the structural model only (behavioral models are not defined in Ecore). At the user interface, the *problems view* shows all constraint violations in a list while the violating elements are highlighted in the Ecore tree editor.

5.2.6 *Constrained editing*

A Fujaba model may be edited at any time in an unconstrained way, ignoring the constraints imposed by the package diagram. In addition, another benefit of our tool is the seamless integration into Fujaba to support a *constrained editing mode*, where the visibility constraints specified in the package diagram are observed at any time. As soon as a package diagram is loaded, the integration tool checks the visibility constraints on each of Fujaba's modeling operations, like creating inheritance hierarchies, specifying class attributes and methods or defining associations. Furthermore object instantiations in story diagrams are checked as well. In case a constraint violation is detected, the editing operation is suspended, and the modeler has to choose an appropriate import (public/private package/element) that will be added to the package diagram in order to successfully

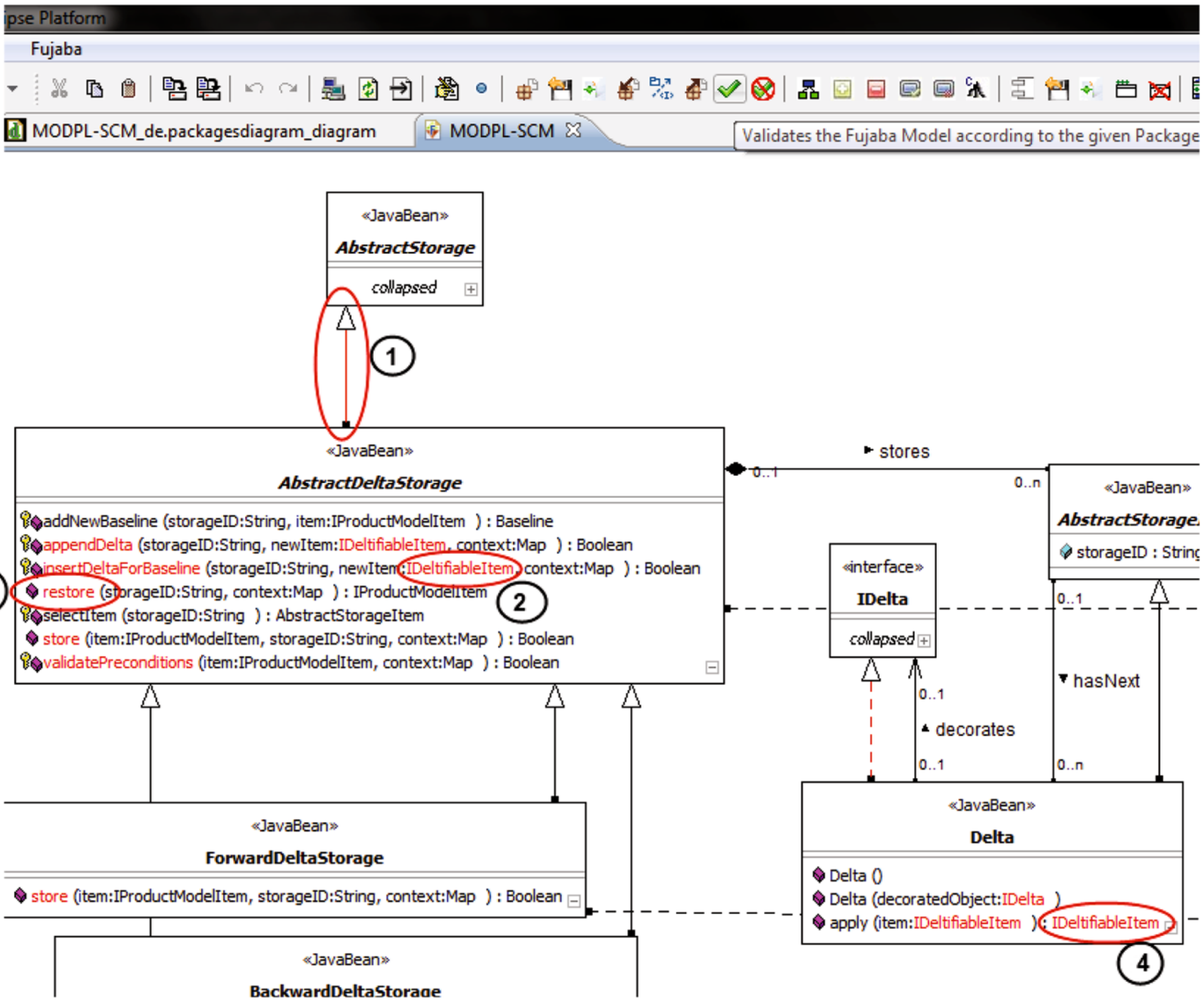

ipse Platform
Fujaba
MODPL-SCM_de.packagesdiagram_diagram
MODPL-SCM
Validates the Fujaba Model according to the given Package
«JavaBean»
AbstractStorage
collapsed
1
«JavaBean»
AbstractDeltaStorage
addNewBaseline (storageID:String, item:IProductModelItem) : Baseline
appendDelta (storageID:String, newItem:IDeltifiableItem, context:Map) : Boolean
insertDeltaForBaseline (storageID:String, newItem:IDeltifiableItem, context:Map) : Boolean
restore (storageID:String, context:Map) : IProductModelItem
selectItem (storageID:String) : AbstractStorageItem
store (item:IProductModelItem, storageID:String, context:Map) : Boolean
validatePreconditions (item:IProductModelItem, context:Map) : Boolean
2
3
stores
0..1
0..n
«JavaBean»
AbstractStorage
storageID : String
«interface»
IDelta
collapsed
0..1
hasNext
0..1
decorates
0..1
0..n
«JavaBean»
ForwardDeltaStorage
store (item:IProductModelItem, storageID:String, context:Map) : Boolean
«JavaBean»
Delta
Delta ()
Delta (decoratedObject:IDelta)
apply (item:IDeltifiableItem) : IDeltifiableItem
4
«JavaBean»
BackwardDeltaStorage

Fig. 8 Validating the current model against the visibility constraints specified in the package diagram

complete the editing operation. Figure 9 shows a suspended operation where an association should be created between two classes from different packages. The target of the association is not visible from the source and therefore the user is asked to choose the desired kind of import which will be added to the package diagram. After the import was created, the target element is visible from the source, and the operation can complete successfully. The same mechanism is also used when:

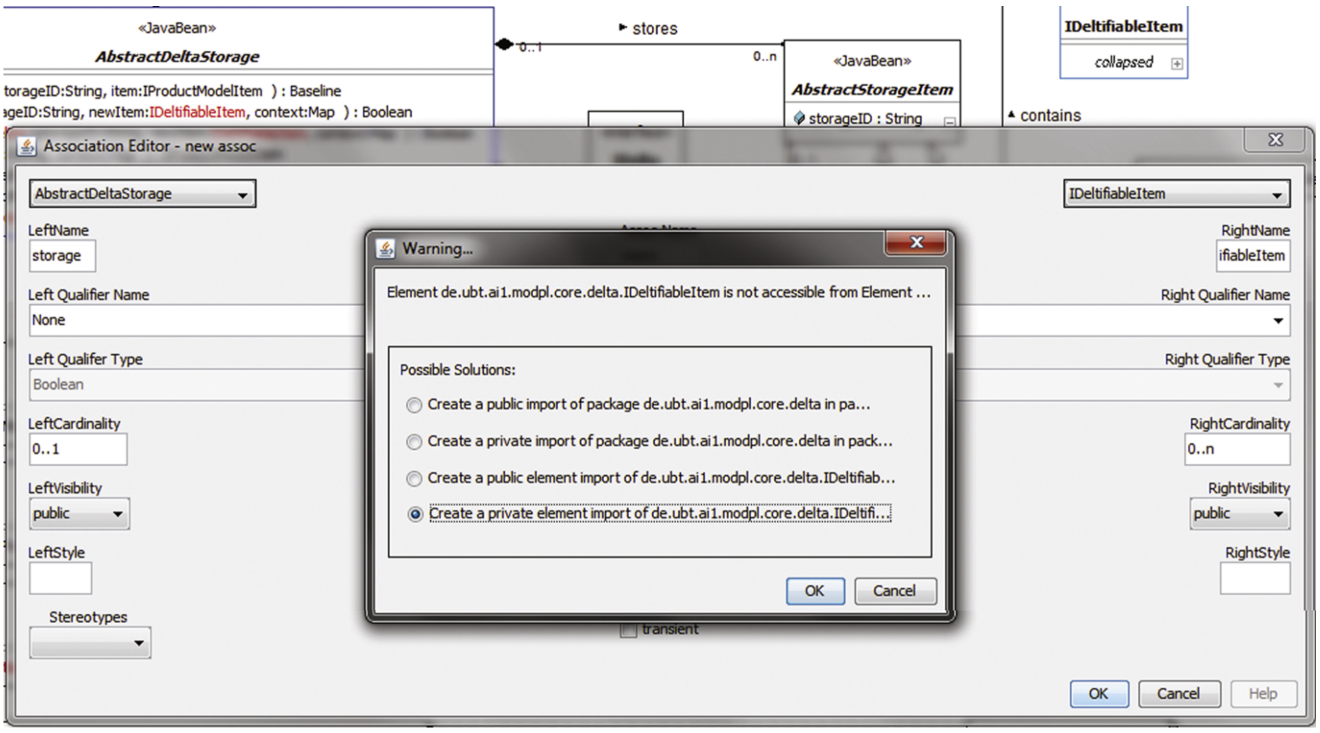
- superclasses are specified,
- attributes are added,
- methods are defined (method types, return values and exception types are checked), and
- objects are instantiated in story diagrams.

Furthermore, the integration tool supports the modeler by optionally restricting types which are displayed in selection dialogs, to types which are visible to the source element according to the constraints specified in the package diagram model. In that case, a constraint violation cannot occur, and there is no need to add an import as a side effect of a constrained editing operation.

Please note that the constrained editing mode may be switched on and off after a package diagram has been loaded. When constrained editing is switched off, additional constraint violations may be introduced which are detected by subsequent validations. Without a loaded package diagram, editing is always performed in unconstrained mode.

5.2.7 *Synchronization*

The package diagram model and the Fujaba model can be kept in synch at any time in the modeling process. With that mechanism, incremental round-trip engineering is realized, as updates in the model and the package diagram can be propagated in both directions. Synchronization is used



torageID:String, item:IProductModelItem) : Baseline
ageID:String, newItem:IDeltifiableItem, context:Map) : Boolean
storageID : String

AbstractDeltaStorage — IDeltifiableItem

LeftName: storage
Left Qualifier Name: None
Left Qualifer Type: Boolean
LeftCardinality: 0..1
LeftVisibility: public
LeftStyle:
Stereotypes:

RightName: ifiableItem
Right Qualifier Name:
Right Qualifer Type:
RightCardinality: 0..n
RightVisibility: public
RightStyle:

Warning...

Element de.ubt.ai1.modpl.core.delta.IDeltifiableItem is not accessible from Element ...

Possible Solutions:

- ○ Create a public import of package de.ubt.ai1.modpl.core.delta in pa...
- ○ Create a private import of package de.ubt.ai1.modpl.core.delta in pack...
- ○ Create a public element import of de.ubt.ai1.modpl.core.delta.IDeltifiab...
- ● Create a private element import of de.ubt.ai1.modpl.core.delta.IDeltifi...

OK Cancel

Fig. 9 Checking the visibility constraints when new associations are created

to reestablish consistency according to the rules given in Sect. 5.2.2. Depending on the selected direction, the *master* of the synchronization operation is determined. Now the source model is traversed, and missing elements are created in the target model. When it is crucial to strictly observe the visibility constraints specified in the package diagram, or when the overall architecture changes, the package diagram is selected as the master of the synchronization operation. In case the model has been updated and classes have been added which only use visible elements according to already specified visibility constraints, the package diagram is not affected by this change. Changes concerning the visibility constraints imply changes in the package diagram in case the model was the master of the synchronization operation though.

It is not only important to synchronize elements that have been added to one model, but also elements that have been deleted. To this end, the integration tool searches for elements which are present in the target model but not in the source model. These elements can now be deleted based upon the user's choice. Please note that elements are not deleted automatically to avoid inadvertent loss of work. This is very important due to the fact that both models can be edited separately and independently, possibly by different users. In that case, the model can be extended by additional packages including class diagrams as well as story diagrams while the package diagram was changed only in terms of visibility. If a synchronization operation is performed with the package diagram as the master, an automatic deletion would now delete the newly introduced packages in the model including all of their child elements.

While synchronization may be performed in both directions, the rules of Sect. 5.2.2 define the consistency of the model with the package diagram. According to these rules, the package diagram plays the master role, and the model is the dependent. In this sense, the dependency is directed. Nevertheless, either the package diagram or the model may be selected as the master with respect to a specific synchronization operation in order to support round-trip engineering.

5.2.8 *Metrics tool*

The metrics tool is used to determine the degree of coupling between packages based on the corresponding import dependencies. Existing metrics that are used to determine the coupling between modules are usually based on classes and their dependencies [6]. But in our modeling approach the dependencies for modeling-in-the-large are described using packages and private import relationships.[1] Thus, metrics may be calculated analogously to the derivation of private imports since they are based on the classes which are contained in the corresponding package. There are several scenarios that imply a dependency between class A and class B:

[1] Public imports are too imprecise to serve as the base for metrics calculation; see Sect. 7.

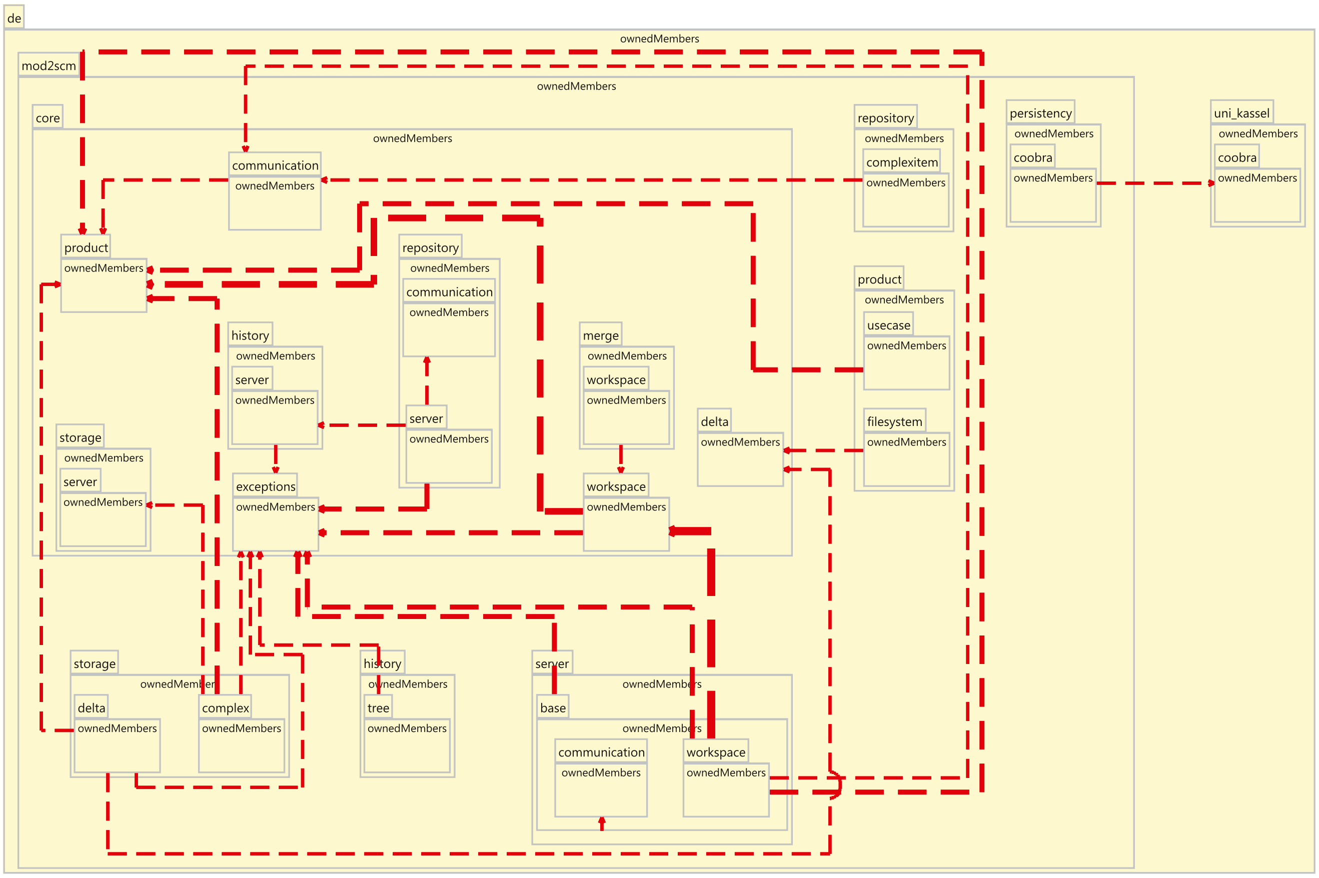


Fig. 10 Degree of coupling between packages, depicted by different thickness of the lines representing import relationships

– B is an attribute type of A,
– B is a return type of a method defined in A,
– B is a parameter type of a method defined in A,
– B is a direct superclass of A, or
– B is used as an object type in a method implementation of A.

The *coupling-strength* $k(A; B)$ between a class A and a class B is computed as follows:

$$\begin{aligned} k(A, B) = {} & |B \textit{ attribute type in } A| \\ & + |B \textit{ return type in } A| \\ & + |B \textit{ parameter type in } A| \\ & + |B \textit{ superclass of } A| \\ & + |B \textit{ object type in } A| \end{aligned}$$

Accordingly, the coupling strength between two packages is defined as the sum of the coupling strengths for each pair of their owned classes.

Figure 10 shows an excerpt of a package diagram and the different degrees of coupling between the single packages. The thickness of the lines depicting package import relationships represents the degree of coupling. To this end, the coupling strength is mapped onto the interval [1 : 10], i.e., 10 degrees of coupling may be distinguished in the diagram. The mapping has been defined heuristically, based on examples from the MOD2-SCM project.

5.3 Methods of use

The integration tool may be used in different ways. Generally, we intend to support the modeling process, but not to enforce a specific way of using the tool. Some examples of *methods of use* are given below:

Strict forward engineering First the package diagram is created, from which a model skeleton is generated. Subsequently, the model is edited in constrained mode only. If the package structure needs to be modified, the modification has to be performed in the package diagram editor and propagated into the model.

Strict reverse engineering First the model is created, from which the package diagram is derived. Subsequently, editing is performed only in the model, either in constrained or unconstrained mode, and the package diagram is synchronized on demand. This functionality is required to import existing models.

Incremental round-trip engineering Going back and forth, the modeler may opportunistically select the most convenient way of working. The model and the package diagram are synchronized frequently, with updates being propagated in both directions.

6 Realization

According to our previous experiences with model-driven development of graphical editors [8, 10, 23], we decided to build the package diagram editor also in a model-driven way using Fujaba to define the metamodel and GMF [24] to build a graphical user interface on top of it.

The meta-model was designed in Fujaba according to the UML specification. Additionally, methods that check the visibility constraints were modeled using Fujaba's story diagrams [9]. The final model was exported to Ecore using Fujaba's EMF code generation engine [23] which is available as a plugin. It exports the static structure of the model into an Ecore model file, and the dynamic part, which is described by the story diagrams, into executable Java code. Afterwards, the code generator of the EMF framework was used to generate the missing Java code fragments (e.g., set() and get() methods). Using this approach, modifications or extensions to both the editor's metamodel and the graphical user interface can be done very quickly, as it often only requires regeneration of the executable code.

GMF allows to generate graphical editors for Ecore models in a model driven way, by providing three different types of models that describe the design and appearance of the graphical elements, the tools that are part of the editor's palette, and a model which maps the elements of the Ecore model with their graphical representation and the appropriate palette entries. The GMF code generation uses these models to generate code for an editor which allows to graphically edit instances of the underlying model. For a more detailed description of how to use Fujaba and GMF to generate graphical editors, please refer to [10].

The forward and reverse engineering capabilities for EMF models have been implemented manually as pure Java code as well as the automatic layout function and the highlighting capabilities for visible packages. Manual coding was preferred because working with external references (in that case classes from the Eclipse runtime and the EMF framework) is not very convenient in Fujaba, especially because in that case the Fujaba story patterns would have consisted mainly of statement activities containing hand written Java code.

The integration tool for Fujaba on the other hand was developed using a mix of manual programming and model-driven development. The parts that concern extensions of Fujaba's graphical user interface were realized using manual coding whereas all operations working on the Fujaba model were specified directly in Fujaba (with the help of story diagrams) and were transformed to executable code by the Fujaba compiler.

7 Application

In this section, we describe how we applied the package concept and the supporting tools in the MOD2-SCM project. We present some *design principles* and *methods of use* which we have considered beneficial in the context of the MOD2-SCM project. In this way, it is illustrated how the package concept and the support tools may be employed for modeling-in-the-large. However, it should be noted that the methods of use presented below constitute only one possible way of applying package diagrams. In particular, our tools deliberately do not enforce these methods of use.

7.1 Architectural design

Figure 11 presents a package diagram which was prepared with the help of our package diagram editor and contains most of the packages of the MOD2-SCM domain model. The diagram is much larger than the strongly simplified version which was presented in Fig. 4 and served as running example in Sect. 5. In the diagrams, all *modules* of the MOD2-SCM domain model are displayed. The term "module" denotes a logically coherent set of packages. To keep the figure legible, some modules were not completely refined to the level of leaf packages.

The overall model architecture is structured into two parts:

– The *core* (shown on the top[2]) provides reusable interfaces and (mainly) abstract classes shared by all instances of the product line. Each of the packages defined on the top level of the core package constitutes a core module.
– The *module library* exploits the core to realize the variable features of the product line. Each top-level package displayed below the core package groups a set of modules which contribute a logically coherent set of features. Modules are decorated with features from the feature model (indicated by white boxes). Each module realizes a single feature. For example, history.tree realizes a tree-structured history (feature tree).

In contrast to Fig. 4 in which public imports were used (green arrows), Fig. 11 displays private imports (red arrows). However, only a small subset of these imports is shown.[3] *Vertical imports* correspond to realization relationships; for

[2]In Fig. 4, an inverted layout was used, where the core was placed at the bottom of the figure.

[3]The complete diagram contains about 250 private imports.

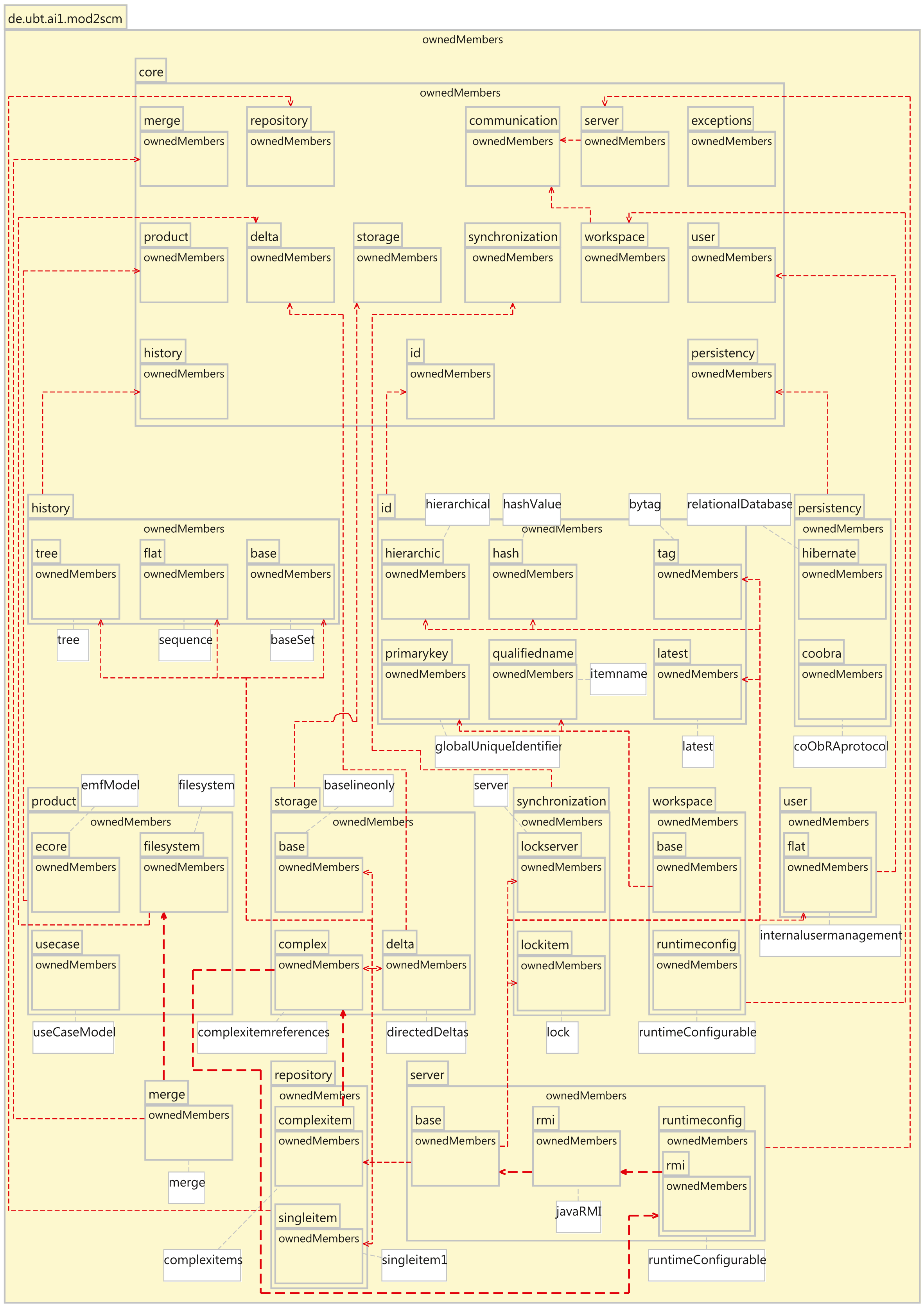


Fig. 11 Package diagram for the MOD2-SCM domain model

example, the module storage.delta realizes the interfaces provided by core.delta. *Horizontal imports* are shown only within the module library. Each import indicates a coupling which needs further investigation since it may prevent orthogonal feature selection. Apart from a few "hot spots", the modules of the module library are coupled only loosely; see [19] for a detailed analysis.

Altogether, the model architecture is organized into multiple dimensions. As suggested in [20], we distinguish between *vertical layering* and *horizontal layering*. Vertical layering decomposes the architecture into the core (base layer) and the module library. Horizontal layering is performed according to groups of functions, as introduced in the feature model. The third dimension is defined by the underlying *client/server architecture*, which requires to structure modules into subpackages for the client, the communication, and the server.[4]

7.2 Modularity

The package concept may be applied in multiple ways. In the MOD2-SCM project, we have established some *design rules* which in particular address the *modularity* of the domain model. If two features in the feature model are orthogonal (i.e., they may be selected independently), the modules which realize these features must not depend on each other. Therefore, we considered it crucial to control the modularity of the domain model. This goal was achieved by the following design guidelines.

Define only one class diagram per package This simple rule has proved useful for controlling the size of packages. If the elements of the structural model cannot be displayed in a single, manageable class diagram, the package should be decomposed into subpackages of adequate size.

Avoid qualified names Referencing model elements through fully qualified names (path names) is not only inconvenient. In addition, a reference of this kind does not require an import; thus, it results in a *hidden dependency*. Therefore, we avoid references through qualified names. Please notice that in general qualified names are required to resolve name clashes on elements of imported packages. However, in the MOD2-SCM domain model name clashes did not occur.

Avoid statement activities For the same reason, we avoid statement activities in story diagrams. In Fujaba, a statement activity is treated as an opaque text fragment which is copied into the generated Java code. Since reverse engineering operates on the Fujaba model rather than on the generated code, statement activities are avoided, if possible (e.g., by replacing statement activities with story patterns containing method calls). There are only a few cases where statement activities cannot be replaced, e.g., when they handle exceptions.

Use rather than modify imported elements As explained in Sect. 4.2, the UML permits that an imported element may be modified in the importing package. For example, attributes, operations, and associations may be added to an imported class. This flexibility may be exploited to define multiple *views*, where each view defines view-specific properties of a shared base element. However, a modification in the importing package introduces a hidden dependency in the opposite direction (the imported package becomes dependent on the importing package). For this reason, we have generally attempted to avoid modifications of imported elements. When using external Java classes, such modifications may not be performed anyway. In the case of classes defined in the domain model, we did not add attributes and operations to imported classes. However, in some cases it was necessary to add bidirectional associations, which modifies an imported class because it adds an association end.

Prefer private over public imports Public imports have been introduced to solve an annoying problem which occurs in modular programming languages such as e.g. Java: When a class is imported, the types used in its interface have to be imported, as well. With transitively working public imports, the "derived" imports may be eliminated, which may significantly reduce the number of required imports. For example, in the package diagram of Fig. 12b (public imports) the imports from P to R and S in Fig. 12a (private imports) are not required.

Nevertheless, we use private rather than public imports in the MOD2-SCM project since they allow to control dependencies more precisely. For example, from the diagram in Fig. 12b it is not evident which packages below Q are

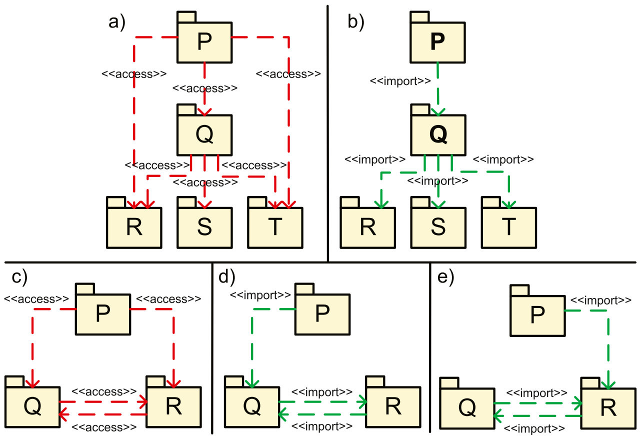


Fig. 12 Reduction of imports

[4]Please notice that Fig. 11 was not refined to this level.

actually used in P. In fact, public imports are a convenient means to access elements owned by other packages. But a public import may grant access to a large number of elements, from which only a small subset is required. For example, in Fig. 6 an import of wsmanager_cvs provides access to nearly all packages displayed in the diagram.

Furthermore, reverse engineering a domain model does not yield a unique result with public imports. For example, reverse engineering using private imports may produce the (unique) diagram of Fig. 12c. If public imports are employed, the reverse engineering tool may produce the diagrams shown in Figs. 12d, e, depending on the order in which the domain model is traversed.

7.3 Use of support tools

As explained in Sect. 5.3, the integration tool for consistency maintenance between the domain model and the package diagram supports, but does not enforce different methods of use. In the MOD2-SCM project, we had to start with a strict reverse engineering phase because the support tools for package diagrams were implemented only after the development of the domain model had already started. Subsequently, we applied incremental round-trip engineering. Thus, changes to the domain model and the package diagram were propagated in both directions, relying on incremental synchronization. Before performing a synchronization step, the domain model was usually validated against the package diagram, revealing inconsistencies which were introduced after the last synchronization step. We did not apply strict forward engineering because this mode of operation was considered too restrictive and inconvenient. However, this mode might still be used in other projects, where a "chief designer" may want to enforce conformance of the domain model with its architecture defined in the package diagram.

8 Related work

In this section we compare the functionality of our tool with a set of modeling tools supporting packages. *MagicDraw* [34], *Rational Rose* [27], *Rational Software Architect (RSA)* [28], *eUML2* [39], and *Sparx Enterprise Architect (EA)* [40] are commercial tools. *Topcased* [43], *Eclipse UML2* [21], and *MOFLON* [2, 45] are available in the public domain. With the exception of MOFLON, which is based on the MOF standard, all tools are based on UML. Furthermore, all tools including MOFLON are based on the UML 2 package concept. Finally, all tools support code generation for structural models. In addition, MOFLON is built on top of Fujaba and thus inherits Fujaba's capabilities of generating code from storage diagrams.

Table 1 Tool comparison: language features

	Package hierarchy	Public package imports	Private package imports	Element imports
MagicDraw	+	+	+	+
MOFLON	+	+	∘	∘
Topcased	+	+	–	–
Rational Rose/RSA	+	+	+	–
Eclipse UML2	+	–	–	+
eUML2	+	–	–	–
Sparx EA	+	+	–	–
our tool	+	+	+	+

Table 1 compares the *language features* supported by these tools and our tool. As is evident from the table, only the package hierarchy is provided by all tools. In contrast, imports have been implemented only partially. eUML2 even does not support imports at all. Apart from our tool, only MagicDraw and MOFLON cover package, element, private, and public imports. In the case of MOFLON, tool support is restricted in the following respects: First, MOFLON allows the user only to create public package imports in the package diagram editor. Once these public imports are created, the visibility attribute can be changed to private in a separate property dialog. Second, although MOFLON uses element imports internally to access non-visible elements, these element imports are never shown to the user in the package diagram.

A core contribution consists in the *integration functions* for controlling consistency between the model and the package diagram (Table 2). A key difference between our approach and the tools compared in this section concerns the relationship between the package diagram and the model. In our approach, we have considered the package diagram and the model as two distinct entities which have to be kept consistent with each other. In all other tools, the package diagram is considered as the skeleton of one integrated model.

As a consequence, in the compared tools the term *forward engineering* has a different meaning than in our own work. In the compared tools, forward engineering means that each model element has to be defined in a package. Thus, the package hierarchy is used for organizing the model from the very beginning. In our tool, forward engineering means that the package hierarchy is rebuilt internally in Fujaba—as mentioned earlier, Fujaba does not support editing of the package hierarchy—and the elements declared in packages are rebuilt in Fujaba, as well (attaching each element to its owning package).

In our tool, *reverse engineering* means that a package diagram is constructed from the Fujaba model by building a

Table 2 Tool comparison: integration functions

	Forward engineering	Reverse engineering	Synchro-nization	Constr. editing	Validation	Metrics calc.
MagicDraw	+	○	○	–	–	–
MOFLON	+	–	–	+	–	–
Topcased	+	–	–	–	–	–
Rational Rose/RSA	+	–	–	–	–	–
Eclipse UML2	+	–	–	–	–	–
eUML2	+	○	–	–	–	–
Sparx EA	+	–	–	–	–	–
our tool	+	+	+	+	+	+

package hierarchy such that all classifiers may be attached to their owning package (the owning package is specified in Fujaba mainly for Java code generation). Furthermore, imports are generated such that the visibility rules of the UML 2 package concept are satisfied. A comparable functionality does not exist in any other tool. Generally, the term "reverse engineering" applies to tool support at a different level: In MagicDraw and eUML2, a model may be reverse engineered from *Java code*. Here, the focus lies on constructing class diagrams. Packages are used for organizing models hierarchically, but the hierarchy is not populated with imports. In contrast, in our tool reverse engineering is performed on a *model* rather than on Java code, and all required imports are created to represent the model architecture.

Similarly, *synchronization* means in our tool that the Fujaba model and the package diagram—which may have been created asynchronously by different users—are kept consistent with each other by propagating changes back and forth. In MagicDraw, synchronization refers to round-trip engineering between the model and the code. In tools which are based on one integrated model, synchronization *within* the model would mean that imports in the package diagrams are kept consistent with the uses of model elements in other than their owned packages. Synchronization of this kind is not supported by any of the compared tools.

Apart from our tool, only MOFLON provides *constrained editing*. To this end, MOFLON restricts the types displayed in selection dialogs to visible ones. In contrast to our tool, this restriction can not be switched off. According to the methods of use mentioned in Sect. 5.3, MOFLON supports strict forward engineering only.

Our tool is unique with respect to its *validation*, which checks whether used elements are visible. MOFLON does not support validation since it does not allow the use of non-visible elements anyway. Thus, in MOFLON the model is always consistent with the package diagram. We consider this approach too restrictive, in particular if the model and the package diagram are modified asynchronously by different users (which is not possible in MOFLON).

In all other tools, packages are primarily used for organizing models in a hierarchical way (in a way which resembles the file system organization). Dependencies among the packages are not taken into account. Imports in the package diagrams have no effect. It is neither possible to constrain the use of elements according to the visibilities defined in the package diagram, nor may the model be validated against the package diagram. Altogether, package diagrams are not seen as *model architectures*. Thus, our tool differs significantly since it provides support for *modeling-in-the-large* and uses UML 2 package diagrams as a *module interconnection language*.

Finally, our *metrics tool* was developed to improve support by modeling-in-the-large for measuring cohesion and coupling on the level of packages. This tool provides unique functionality which is not offered by any of the compared tools.

9 Conclusion

In this paper, we have argued that model-driven software engineering urgently needs powerful languages and tools for modeling-in-the-large. As a step towards this goal, we have presented a set of tools based on UML package diagrams. The tool set comprises a graphical editor, a metrics tool, as well as integration tools for maintaining consistency of Ecore or Fujaba models with package diagrams. The functions provided by the tool set include batch-mode forward and reverse engineering, model validation, model synchronization, and constrained model editing. The tool set is designed to cover a wide spectrum of methods of use, including strict forward engineering, strict reverse engineering, and incremental round-trip engineering.

Furthermore, we have demonstrated how the package concept and the supporting tools were applied in a project which is dedicated to model-driven engineering of a product line for software configuration management systems. In the MOD2-SCM project, support for modeling-in-the-large played a crucial role: The dependencies among modules realizing different features had to be controlled carefully to guarantee that features may be selected in an orthogonal way. The model architecture is elaborated in detail in [19], where the design rationales for all modules are given and the dependencies among these modules are discussed and justified.

Our analysis of the current state of tool support demonstrates that modeling-in-the-large still does not receive sufficient attention. This observation applies to Fujaba, EMF, and the tools discussed in Sect. 8. We claim that long-term *model maintenance* will require to spend more effort on modeling-in-the-large. Otherwise, models will turn out to be unmaintainable because their components are coupled tightly and these couplings were not considered carefully during model development. It seems that many developers still structure models somehow into package hierarchies but do not consider the dependencies among these packages. We hope that this paper may contribute in the long run to improving this development process.

Acknowledgements The valuable comments of the unknown reviewers are gratefully acknowledged.

References

1. Allen R, Garlan D (1997) A formal basis for architectural connection. ACM Trans Softw Eng Methodol 6(3):213–249
2. Amelunxen C, Königs A, Rötschke T, Schürr A (2006) MOFLON: A standard-compliant metamodeling framework with graph transformations. In: Rensink A, Warmer J (eds) Model driven architecture—foundations and applications: second European conference, ECMDA-FA 2006. Lecture notes in computer science, vol 4066. Springer, Bilbao, pp 361–375
3. Antkiewicz M, Czarnecki K (2004) FeaturePlugin: feature modeling plug-in for eclipse. In: Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange (eclipse'04). ACM Press, New York, pp 67–72
4. Bézivin J, Jouault F, Rosenthal P, Valduriez P (2005) Modeling in the large and modeling in the small. In: Model driven architecture, European MDA workshops: foundations and applications, MDAFA 2003 and MDAFA 2004. Lecture notes in computer science, vol 3599. Twente, The Netherlands, pp 33–46
5. Booch G, Rumbaugh J, Jacobson I (1999) The unified modeling language user guide. Object technology series. Addison-Wesley, Amsterdam
6. Briand LC, Daly JW, Wüst J (1998) A unified framework for cohesion measurement in object-oriented systems. Empir Softw Eng 3(1):65–117
7. Buchmann T (2010) Modelle und Werkzeuge für modellgetriebene Softwareproduktlinien am Beispiel von Softwarekonfigurationsverwaltungssystemen. PhD Thesis, University of Bayreuth, Germany
8. Buchmann T, Dotor A (2006) Building graphical editors with GEF and Fujaba. In: Giese H, Westfechtel B (eds) Fujaba days 2006. Bayreuth, Germany, pp 47–50
9. Buchmann T, Dotor A, Klinke M (2009) Supporting modeling in the large in Fujaba. In: van Gorp P (ed) Proceedings of the 7th international Fujaba days. Eindhoven, The Netherlands, pp 59–63
10. Buchmann T, Dotor A, Westfechtel B (2007) Model-driven development of graphical tools—Fujaba meets GMF. In: Filipe J, Helfert M, Shishkov B (eds) Proceedings of the second international conference on software and data technologies (ICSOFT 2007). INSTICC Press, Barcelona, pp 425–430
11. Buchmann T, Dotor A, Westfechtel B (2008) Experiences with modeling in the large in Fujaba. In: Aßmann U, Johannes J, Zündorf A (eds) Fujaba days 2008—6th international Fujaba days, Dresden, Germany, pp 5–9
12. Buchmann T, Dotor A, Westfechtel B (2009) Model-driven development of software configuration management systems—a case study in model-driven engineering. In: Proceedings of the 4th international conference on software and data technologies (ICSOFT 2009), vol 1. INSTICC Press, Sofia, pp 309–316
13. Chang KC, Cohen SG, Hess JA, Novak WE, Peterson AS (1990) Feature-oriented domain analysis (FODA) feasibility study. Tech Rep CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA
14. Clements P (1996) A survey of architecture description languages. In: Proceedings of the 8th ACM/IEEE international workshop on software specification and design (IWSSD 1996). IEEE Computer Society Press, Schloss Velen, pp 16–25
15. Collins-Sussman B, Fitzpatrick BW, Pilato CM (2004) Version control with subversion. O'Reilly, Sebastopol
16. Cuadrado JS, Molina JG (2009) Modularization of model transformations through a phasing mechanism. Softw Syst Model 8(3):325–345
17. DeRemer F, Kron H (1976) Programming-in-the-large versus programming-in-the-small. IEEE Trans Softw Eng 2(2):80–86
18. Dingel J, Diskin Z, Zito A (2008) Understanding and improving UML package merge. Softw Syst Model 7(4):443–467
19. Dotor A (2011) Entwurf und Modellierung einer Produktlinie von Software-Konfigurations-Management-Systemen. PhD thesis, University of Bayreuth, Germany
20. D'Souza DF, Wills AC (1998) Objects, components, and frameworks with UML—the Catalysis approach. Addison-Wesley, Upper Saddle River
21. The Eclipse Foundation: Eclipse UML2. http://www.eclipse.org/uml2
22. Estublier J, Casallas R (1994) The Adele configuration manager. In: Tichy WF (ed) Configuration management. Trends in software, vol 2. Wiley, New York, pp 99–134
23. Geiger L, Buchmann T, Dotor A (2007) EMF code generation with Fujaba. In: Geiger L, Giese H, Zündorf A (eds) Fujaba days 2007, Kassel, Germany, pp 25–29
24. Gronback RC (2009) Eclipse modeling project: a domain-specific language (DSL) toolkit, 1st edn. The eclipse series. Addison-Wesley, Boston
25. Hemel Z, Kats LC, Groenewegen DM, Visser E (2010) Code generation by model transformation: a case study in transformation modularity. Softw Syst Model 9(3):375–402
26. Hofmeister C, Nord RL, Soni D (1999) Describing software architecture with UML. In: Donohoe P (ed) Proceedings of the TC2 first working IFIP conference on software architecture (WICSA 1999). IFIP conference proceedings, vol 140. Kluwer, San Antonio, pp 145–160
27. IBM Corp.: Rational Rose. http://www.ibm.com/software/awdtools/developer/rose/java/index.html
28. IBM Corp.: Rational Software Architect. http://www.ibm.com/software/awdtools/swarchitect/standard
29. Kandé MM, Strohmeier A (2000) Towards a UML profile for software architecture descriptions. In: Evans A, Kent S, Selic B (eds) Proceedings of the third international conference on the Unified Modeling Language (UML 2000). Lecture notes in computer science, vol 1939. Springer, New York, pp 513–527

30. Kolovos DS, Paige RF, Pollack FA (2008) The grand challenge of scalability for model driven engineering. In: Chaudron MR (ed) Models in software engineering, workshops and symposia at MODELS 2008. Lecture notes in computer science, vol 5421. Springer, Toulouse, pp 48–53
31. Kruchten P, Obbink H, Stafford J (2006) The past, present, and future of software architecture. Commun ACM 23(2):22–30
32. Medvidovic N, Taylor RN (2000) A classification and comparison framework for software architecture description languages. IEEE Trans Softw Eng 26(1):70–93
33. Mezini M, Beuche D, Moreira A (eds) (2009) Proceedings 1st international workshop on model-driven product line engineering (MDPLE 2009). In: CTIT proceedings, Twente, The Netherlands
34. No Magic Inc.: MagicDraw. http://www.magicdraw.com
35. Object Management Group, Needham MA (2010) OMG Unified Modeling Language (OMG UML), Infrastructure, Version 2.3, formal/2010-05-03 edn
36. Object Management Group, Needham MA (2010) OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.3, formal/2010-05-05 edn
37. Prieto-Diaz R, Neighbors JM (1987) Module interconnection languages. J Syst Softw 6(4):307–334
38. Shaw M, Clements P (2006) The golden age of software architecture. Commun ACM 23(2):31–39
39. Soyatec—Open Solution Company: eUML2. http://www.soyatec.com/euml2
40. Sparx Systems: Sparx Enterprise Architect. http://www.sparxsystems.eu
41. Steinberg D, Budinsky F, Paternostro M, Merks E (2009) EMF eclipse modeling framework, 2nd edn. The eclipse series. Addison-Wesley, Upper Saddle River
42. Tichy WF (1985) RCS—a system for version control. Softw Pract Exp 15(7):637–654
43. TOPCASED Consortium: TOPCASED: The open-source toolkit for critical systems. http://www.topcased.org
44. Tratt L, Gogolla M (eds) (2010) Theory and practice of model transformations—third international conference, ICMT 2010. In: Lecture notes in computer science, vol 6142. Springer, Malaga
45. TU Darmstadt Fachgebiet Echtzeitsysteme: MOFLON. http://www.moflon.org
46. Vesperman J (2006) Essential CVS. O'Reilly, Sebastopol
47. Wagelaar D, van der Straeten R, Deridder D (2010) Module superimposition: a composition technique for rule-based model transformation languages. Softw Syst Model 9(3):285–309
48. Weisemöller I, Schürr A (2008) Formal definition of MOF 2.0 metamodel components and composition. In: Czarnecki K, Ober I, Bruel JM, Uhl A, Völter M (eds) Model driven engineering languages and systems—11th international conference, MODELS 2008. Lecture notes in computer science, vol 5301. Springer, Toulouse, pp 386–400
49. White BA (2003) Software configuration management strategies and Rational ClearCase. Object technology series. Addison-Wesley, Reading
50. Zündorf A (2001) Rigorous object oriented software development. Tech Rep, University of Paderborn, Germany

Thomas Buchmann received his diploma degree in mathematics in 2002 from the University of Bayreuth. For the following three years he was employed as manager of the software engineering department at a medium sized company. Since 2005 he works as a research assistant at the software engineering chair. In 2010 he received his doctoral degree from the University of Bayreuth. His research interests include graph transformations, model-driven engineering, software product line engineering, software configuration management and software architecture.

Alexander Dotor finished 2005 his diploma in computer science at the Technical University of Darmstadt. In 2011 he received the doctoral degree from the University of Bayreuth for his research about a model driven product line for software configuration management systems (MOD2SCM). His research interests are modular software architecture, model driven development and software product lines. He is currently working as developer for software architecture transformations at Senacor Technologies.

Bernhard Westfechtel received his diploma degree from University of Erlangen-Nuremberg in 1983 and his doctoral as well as his habilitation degree (all in computer science) from RWTH Aachen University in 1991 and 1999, respectively. Since 2004, he has been a full professor of computer science (in software engineering) at University of Bayreuth. His research interests include graph transformations, model-driven engineering, software product line engineering, software configuration management, software process modeling, software architecture, and reengineering.