MODEL-DRIVEN DEVELOPMENT OF SOFTWARE CONFIGURATION MANAGEMENT SYSTEMS
A Case Study in Model-driven Engineering

Thomas Buchmann, Alexander Dotor and Bernhard Westfechtel
Angewandte Informatik 1, Universität Bayreuth, D-95540 Bayreuth, Germany
firstname.lastname@uni-bayreuth.de

Keywords: Model-driven development, Software configuration management, Software product line.

Abstract: Software configuration management (SCM) is the discipline of controlling the evolution of large and complex software systems. Current SCM systems are themselves large and complex. Usually, their underlying models are hard-wired into the program code, which is written manually. In contrast, we present a modular and model-driven approach to software configuration management which (a) reduces development effort by replacing coding with creating executable models and (b) provides a product line supporting the configuration of an SCM system from loosely coupled, reusable components. In addition to improving SCM support, our intent is to use our system as a large case study for evaluating languages and tools for model-driven development.

1 INTRODUCTION

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 (Tichy, 1985) over medium-sized systems such as CVS (Vesperman, 2006) or Subversion (Collins-Sussman et al., 2004) to large-scale industrial systems such as Adele ClearCase (White, 2003). The current state of practice is characterized as follows:

- SCM systems are *large*. For example, even the code base of the GNU CVS project comprises about 300,000 lines of code. CVS is still a rather small tool compared to a commercial system for large enterprises such as ClearCase.
- SCM systems are *similar*. For example, almost all commercial and open source systems are based on *version graphs*, which are used to manage the evolution of software objects.
- The underlying *models* are defined only implicitly by the program code, i.e., the model is *hard-wired* into the respective system.
- SCM systems are *hard to adapt* to modified requirements. For example, although Subversion provides similar functionality as CVS (at least from a bird's eye view), the developers of Subversion decided to start over from scratch.

These observations have motivated us to launch a project for developing a *model-driven and modular SCM system* (*MOD2-SCM* (Buchmann et al., 2008)):

- 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 components* which may be configured by defining the *features* of the respective target system.

In addition to improving SCM support, our intent is to use our system as a large *case study* for evaluating languages and tools for model-driven development. The research is built on the hypothesis that model-driven development improves the software development process. However, this cannot be simply taken for granted. Rather, the hypothesis has to be checked carefully, and both achievements and limitations have to be identified.

2 APPROACH

In our approach, we follow a *model-driven product line engineering process* (Figure 1). In product line engineering, a distinction is made between domain and application engineering (Pohl et al., 2005). In *domain engineering*, the domain is analyzed, and software is developed to support the respective domain. In *application engineering*, a specific application, i.e., an instance of the product line, is created. While domain engineering requires a full-fledged *development process*, application engineering is reduced in our approach to a *configuration process*. The steps of the engineering process are described below:

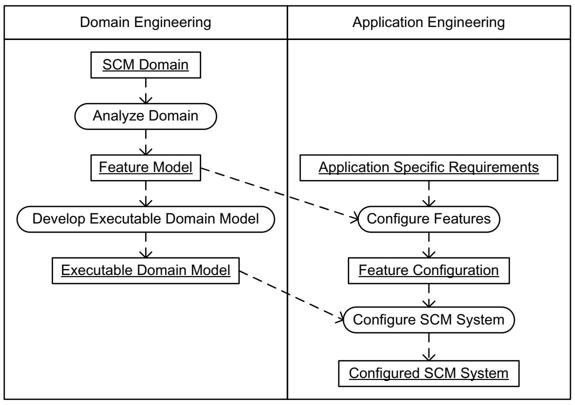


Figure 1: Engineering process.

Analyze Domain. The SCM domain is analyzed by investigating and classifying SCM systems. The result of this analysis is documented by a *feature model*, which is based on FODA (Feature-Oriented Domain Analysis (Chang et al., 1990)). The feature model describes mandatory, optional, and alternative features of the SCM systems to be built with the product line.

Develop Executable Domain Model. An executable domain model is developed for the feature model. To this end, we use *Fujaba* (Zündorf, 2001), an object-oriented modeling language and CASE tool. The domain model comprises both a structural model, defined by class diagrams, and a behavioral model, defined by story diagrams (similar to UML 2.0 interaction overview diagrams).

Configure Features. From the feature model, features are selected for the SCM system to be built. This results in a *feature configuration*.

Configure SCM System. The executable domain model is configured automatically according to the selected feature configuration.

3 RELATED WORK

About a decade ago, a few research projects were dedicated to the development of a *uniform version model*. E.g., in ICE (Zeller and Snelting, 1997) version models were represented with logical expressions for expressing visibilities and constraints.

(van der Lingen and van der Hoek, 2003) proposes a component-based architecture for SCM systems. Components may be viewed as variation points offering different policies for storage, hierarchy, locking, distribution, etc.

Only a few approaches have been dedicated to *model-driven product lines of SCM systems*. In Bamboo (Whitehead and Gordon, 2003; Whitehead et al., 2004), version models are defined in an extended ER data model (Containment Modeling Framework). An SCM system is generated from a CMF model.

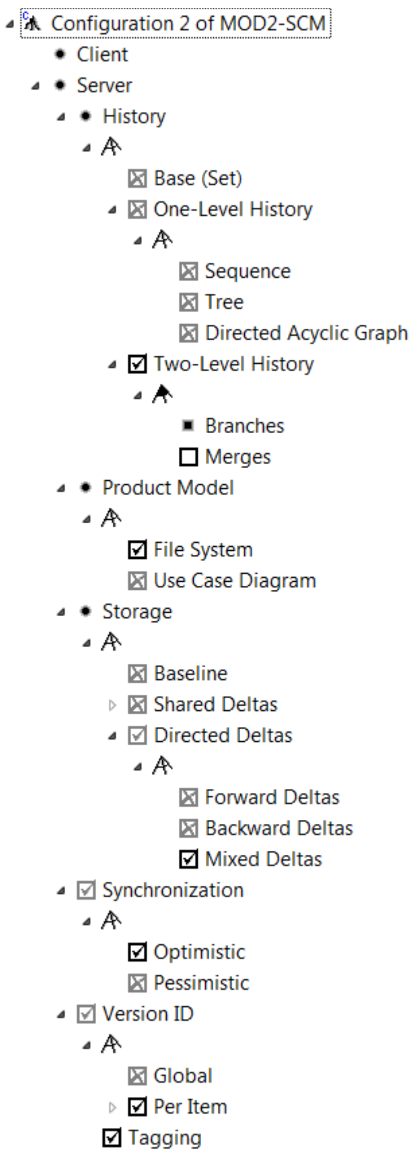
The PhD thesis of Kovŝe (Kovŝe, 2005) investigates a product line approach to the model-driven development of versioning systems. The user of the product line defines a version model as a UML profile; stereotypes and tagged values are used to parameterize the behavior of modeling elements.

Among the approaches described above, only Bamboo and the work of Kovŝe combine model-driven development and software product line engineering. MOD2-SCM is unique with respect to behavioral modeling of SCM systems: In all systems discussed above, behavior has to be programmed. In MOD2-SCM, behavior is modeled with story diagrams, from which Fujaba generates executable code.

4 FEATURE MODEL

A *feature* is a property that is relevant to some stakeholder and is used to capture commonalities or to distinguish among products in a product line. A *feature model* consists of one or more *feature diagrams*. Figure 2 shows a feature diagram for a part of the SCM domain. The diagram was created with FeaturePlugin for Eclipse (Antkiewicz and Czarnecki, 2004). Filled and unfilled circles represent mandatory and optional features, respectively. An unfilled fork denotes an exclusive-or selection, i.e., exactly one of the child features (unfilled squares) has to be selected. Finally, in the case of a filled fork at least one of the child features has to selected; here, a filled square indicates a mandatory child feature. Crosses and ticks represent a feature configuration (Section 6).

The feature diagram distinguishes between the Client (not considered further) and the Server. With respect to the History, the alternatives Base (no his-



- Configuration 2 of MOD2-SCM
  - ● Client
  - ● Server
    - ● History
      - ☒ Base (Set)
      - ☒ One-Level History
        - ☒ Sequence
        - ☒ Tree
        - ☒ Directed Acyclic Graph
      - ☑ Two-Level History
        - ■ Branches
        - □ Merges
    - ● Product Model
      - ☑ File System
      - ☒ Use Case Diagram
    - ● Storage
      - ☒ Baseline
      - ☒ Shared Deltas
      - ☑ Directed Deltas
        - ☒ Forward Deltas
        - ☒ Backward Deltas
        - ☑ Mixed Deltas
    - ☑ Synchronization
      - ☑ Optimistic
      - ☒ Pessimistic
    - ☑ Version ID
      - ☒ Global
      - ☑ Per Item
      - ☑ Tagging

Figure 2: Feature model and feature configuration.

tory), One-Level History (with subfeatures Sequence, Tree, Directed Acyclic Graph) and Two-Level History are provided. In the latter case, Branches are mandatory, and Merges are optional. Product Model refers to the types of items under version control, e.g., File System or Use Case Diagram. Storage is used to control the storage mechanism: Baseline if deltas are not used, Shared Deltas, and Directed Deltas as shown in Figure 3. With respect to Synchronization, we distinguish between Optimistic and Pessimistic. Finally, Version ID contains a feature group for Global or Per Item system-defined identification, and an optional feature Tagging for user-defined names.

In a product line, it is essential to provide for

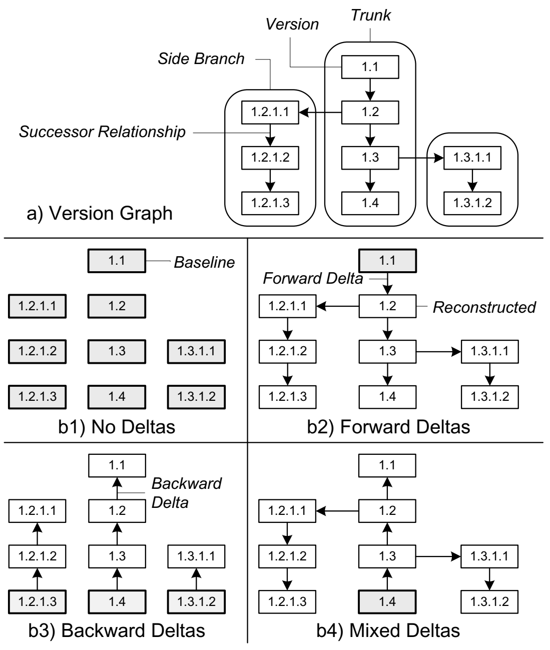


Figure 3: Logical organization of a version graph (a) and alternative physical organizations (b).

orthogonal feature combination. For example, Figure 3 illustrates that the *history model* and the *storage model* may be combined in an orthogonal way. Figure 3a displays an RCS/CVS-like version graph, which has explicit branches (two-level history). Part b of the figure illustrates alternative physical organizations (ways of storing versions). Thus, a version graph may be stored in different ways. Conversely, a given storage model may be combined with different version graphs (not shown here).

5 EXECUTABLE DOMAIN MODEL

This section presents cutouts of the executable domain model realizing the feature model defined above. The presentation demonstrates

- how the features are mapped into the executable domain model,
- how the domain model supports orthogonal combination of features, and
- how the behavioral model may be defined in a high-level graphical notation.

5.1 Package Diagram

Figure 4 displays the *model architecture* as a *package diagram*, using UML 2.0 notation. In addition

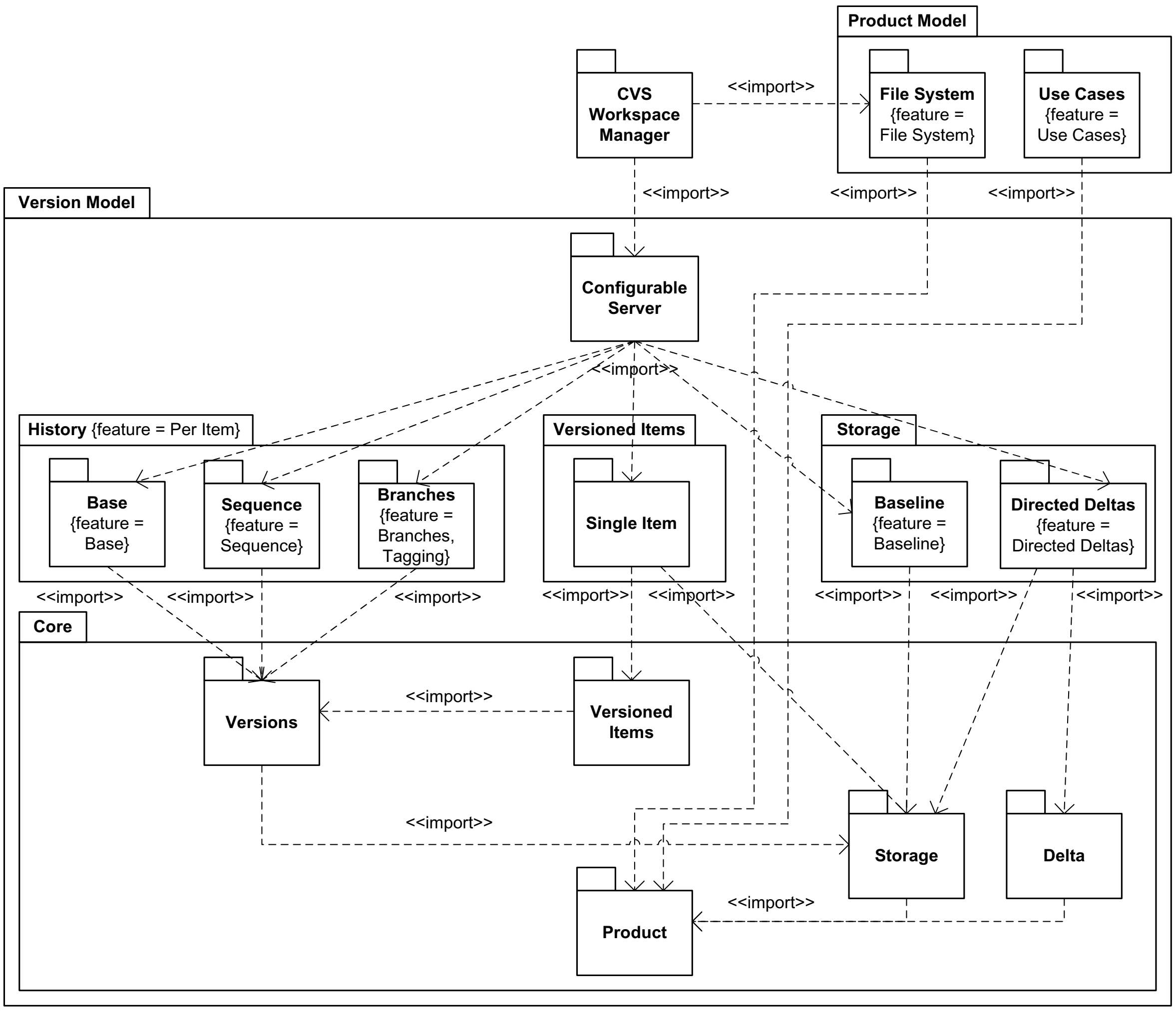

Product Model
CVS Workspace Manager
<<import>>
File System {feature = File System}
Use Cases {feature = Use Cases}
Version Model
Configurable Server
History {feature = Per Item}
Base {feature = Base}
Sequence {feature = Sequence}
Branches {feature = Branches, Tagging}
Versioned Items
Single Item
Storage
Baseline {feature = Baseline}
Directed Deltas {feature = Directed Deltas}
Core
Versions
Versioned Items
Storage
Delta
Product

Figure 4: Package diagram with feature annotations.

to *nesting*, packages are related by *public imports*, which work transitively. Packages are related to the feature model by *tagged values*: The tag feature is assigned the supported feature(s) as value(s). Some features are still missing because they have not been implemented yet.

Version Model contains the nested package Core, which provides basic, generic functionality on top of which higher-level packages are built. *Generic* means that the Core addresses common rather than discriminating features. *Basic* implies that the assumptions to be met by higher-level packages are minimized.

Discriminating features are introduced above the Core. History deals with version graphs and currently supports the features Base, Sequence, and Branches. So far, all subpackages support the identification of versions per item, i.e., identification by global numbers such as e.g. in Subversion is not supported yet. Storage provides storage of baselines and directed deltas. Versioned Items is built on top of the Core subpackage of the same name. Its subpackage Single Item adds a storage to the version set (in the Core package, versioned items are still completely independent from the storage mechanism). Finally, Configurable Server provides a uniform interface to the server managing the repository.

The Version Model depends neither on the Storage nor on the Product model. For the latter, two sample alternatives (file systems and use case diagrams) are provided. Finally, CVS Workspace Manager offers a CVS-like workspace manager, relying on the file system, branches, and mixed deltas.

5.2 Class Diagrams

In the next step, the model architecture is refined into a set of class diagrams. As an example, Figure 5 shows a class diagram (extending over multiple pack-

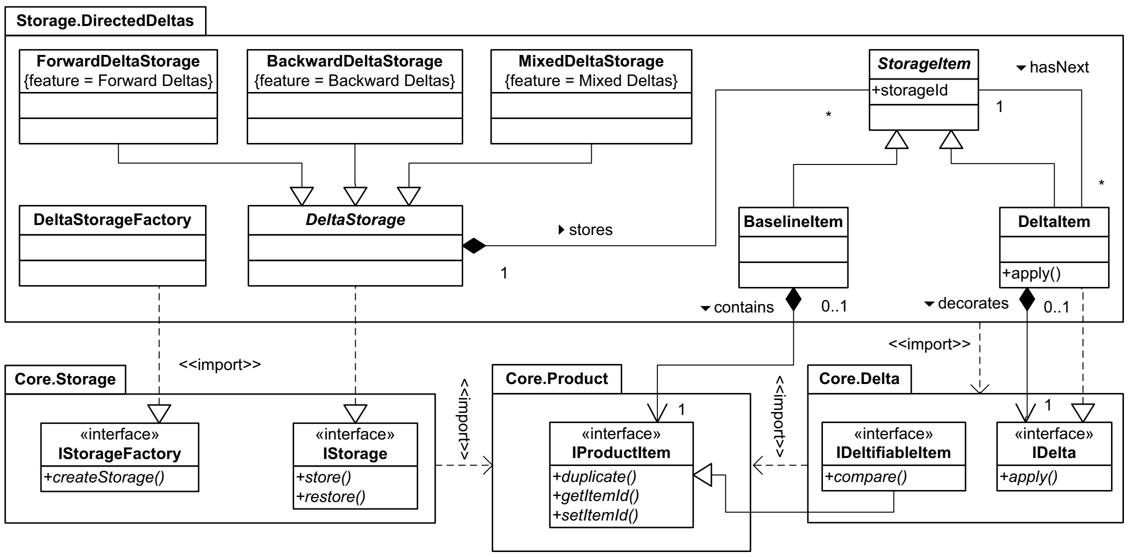


Figure 5: Class diagram for the storage model.

ages) for the *storage model*, which relies on minimal assumptions regarding the items to be stored. The interface IProductItem of package Product defines a few methods to be supported by each class of storable items: duplicate() for copying an item, and get() and set() methods for an *item identifier*, which is used to uniquely identify the item to be versioned. The subpackage Storage of package Core introduces an interface IStorage with store() and restore() methods, which refer to IProductItems, and an interface for a storage factory, offering a createStorage() method. Please note that Storage is still independent of the Delta package, which introduces an interface IDeltifiableItem extending IProductItem with a compare() method, and an interface IDelta, which is used to represent (directed) deltas between deltifiable items and offers an apply() method.

Implementations of storages are provided in the subpackage Storage of package Version Model. In the subpackage Storage.DirectedDeltas, the abstract class DeltaStorage partially implements the interface IStorage by means of some auxiliary methods to be called by its subclasses. A delta storage stores a set of storage items, each of which is identified uniquely by an externally assigned *storage identifier*. The abstract class StorageItem is refined into the subclasses Baseline, which wraps a product item, and DeltaItem, which both extends and decorates a delta. Each of the features Forward, Backward, and Mixed Deltas is supported by a respective subclass of DeltaStorage. Finally, the package offers a factory class for creating one of these storages.

The storage model is orthogonal to the history model. The package DirectedDeltas knows how to store a new item relatively to a predecessor item. Here, the term “predecessor” does not refer to a successor relationship in a version graph. Rather, it merely implies that this item must be already present when the new item is to be stored. Furthermore, the storage identifier must be supplied by the caller of the store() method. Its meaning — actually a version identifier — is not known to the storage.

5.3 Story Diagrams

The next step consists in the realization of the methods defined in class diagrams. In Fujaba, a method may be programmed by a *story diagram*, from which executable Java code is generated. A story diagram is an activity diagram with 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 composed of objects and links; objects may be decorated with method calls. 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.

In the following, we present one example to illustrate the style of programming with story patterns. The store() method of class MixedDeltaStorage (Figure 6) is supplied with the item to be stored and an externally assigned storage identifier. Please note that

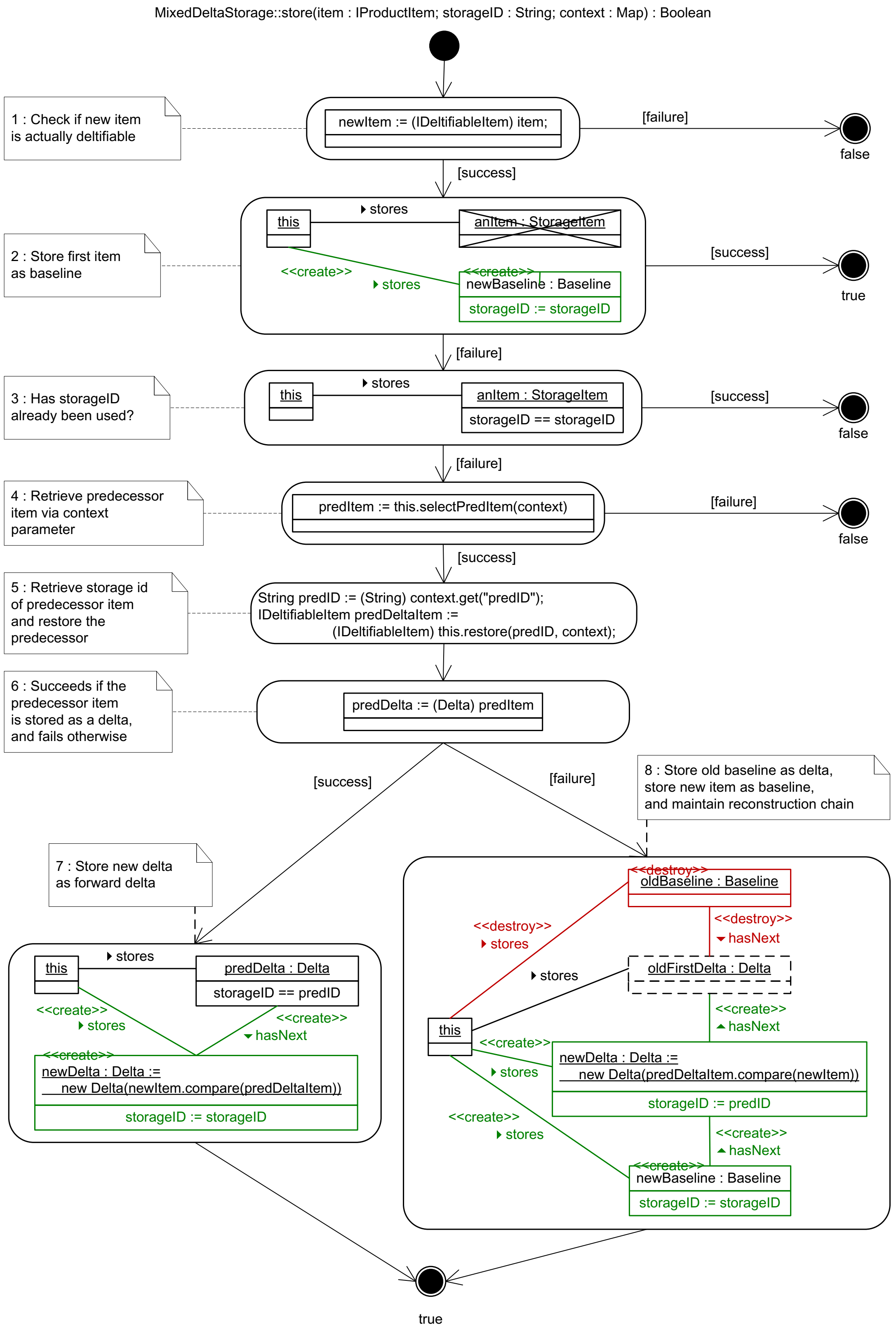

MixedDeltaStorage::store(item : IProductItem; storageID : String; context : Map) : Boolean
1 : Check if new item is actually deltifiable
newItem := (IDeltifiableItem) item;
[failure]
false
[success]
2 : Store first item as baseline
this
stores
anItem : StorageItem
<<create>>
stores
<<create>>
newBaseline : 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
<<create>>
hasNext
this
<<create>>
stores
newDelta : Delta := new Delta(predDeltaItem.compare(newItem))
storageID := predID
<<create>>
stores
<<create>>
hasNext
<<create>>
newBaseline : Baseline
storageID := storageID
true

Figure 6: Story diagram for storing an item (mixed deltas).

only deltifiable items can be stored; furthermore, the storage identifier must not have been used before. If the item is the first one to be stored, it is stored as a baseline. Otherwise, the item is stored as a forward delta if its predecessor is a delta, and as a backward delta it its predecessor is a baseline.

The method store() does not assume a specific history model. Rather, it calls the method selectPredItem(), which retrieves the predecessor item (if any) from the context parameter. This is a minimal assumption in the case of directed deltas: The context relative to which the new item is to be stored has to be known. Conversely, the method for adding a new version (not shown) does not depend on the storage model. Rather, it merely calls the store() method of the generic interface IStorage.

6 CONFIGURATION PROCESS

In the first step of the configuration process shown on the right-hand side of Figure 1, the application engineer describes a *feature configuration* for the feature model of Section 4. Ticks and crosses in Figure 2 represent selected and disabled features, respectively. The figure displays a CVS-like configuration (two-level history without merge relationships, file system as product model, mixed deltas, optimistic synchronization, version identification per item, and tagging).

The second step – *configuring the SCM system* – is performed automatically. When a server is created, the selected features are used to instantiate respective factories supporting these features. For example, for the sample feature configuration instances of BranchedHistoryFactory (package Branches) and DeltaStorageFactory (package DirectedDeltas) are created. These specific factories implement generic interfaces on which the rest of the code depends.

The server is independent from the product model since it depends only on generic interfaces. For example, when a version is added to the repository, an item is passed to the server which is accessed through generic interfaces. Therefore, the server need not be configured with respect to the product model.

7 DISCUSSION

7.1 Executable Domain Model

The main reason for using Fujaba consists in its support for executable models. The added value compared to other CASE tools is provided by *story diagrams* and their compilation into executable code. In contrast, code generation from class diagrams is supported by numerous other CASE tools, as well, e.g., EMF (Steinberg et al., 2009).

More specifically, *story patterns* are those elements of story diagrams which significantly raise the level of abstraction above conventional programming languages such as Java. In contrast, the *control flow* does not raise the level of abstraction. Some developers may prefer graphical over textual notation of control flow. On the other hand, it is easy to lose orientation in large story diagrams with complex control flow. Thus, a modeler using Fujaba should keep the principles of structured programming in mind.

7.2 Model Architecture and Design

The executable domain model has to be designed for reuse and change. To this end, we have applied established *object-oriented design principles and patterns* to minimize the coupling of components. In addition, we have invested significant effort in the design of a *model architecture*, which defines the overall structure at a larger scale than the rather fine-grained level on which design patterns operate. In this paper, we have used a UML package diagram to represent the model architecture. With the help of packages, the coupling between architectural units cannot be controlled adequately for the following reasons:

- Public imports work transitively. Thus, the diagram does not tell which packages in the transitive closure are actually used.
- A public element may always be referenced through its full qualified name. These dependencies are not reflected in the package diagram, which cannot constrain the use of elements.
- An imported element may even be modified in an importing namespace ((Object Management Group, 2007), p. 143). Thus, the definition of an element may be spread over multiple packages.

7.3 Feature Modeling and Configuration

We have applied feature modeling to define common and discriminating features of SCM systems. The underlying concepts are simple to understand; a feature model is an intuitive and useful means to define the capabilities of a product line. So far, we have defined no constraints on the combination of features (apart from those constraints which are expressed directly in the feature diagram itself). In fact, an essential goal of our project consists in building a product line for SCM systems where features may be combined as freely as possible.

We have annotated the domain model manually with features such that elements of the domain model may be traced back to the feature model. These annotations are performed at a coarse-grained level, i.e., coarse-grained units such as packages or classes are decorated with features rather than fine-grained units such as attributes, associations, methods, parameters, story patterns, etc. This approach keeps the multi-variant architecture manageable.

Currently, code is generated for the complete domain model supporting all features. A feature configuration is used to fix the parameters passed to the method which is responsible for creating a server (see Section 6). While this approach, which has been realized on top of Fujaba, supports flexible selection of features even at runtime, it also requires to deliver the code for the complete product line to each individual customer. This can be avoided by modifying the Fujaba compiler such that only the code for the selected features is generated (Buchmann and Dotor, 2009).

8 CONCLUSIONS

We presented a model-driven and modular approach to the development of SCM systems. Furthermore, we discussed the experiences we have gained so far in applying model-driven product line engineering to this application domain. Development of the product line is still under way. Currently, the domain model consists of 87 classes and interfaces with 283 methods. The code base comprises 19,284 lines of (almost exclusively generated) Java code.

REFERENCES

Antkiewicz, M. and Czarnecki, K. (2004). FeaturePlugin: Feature modeling plug-in for Eclipse. In Burke, M. G., editor, *Proceedings of the 2004 OOPSLA Workshop on Eclipse Technology eXchange (ETX 2004)*, pages 67–72, British Columbia, Canada. ACM Press.

Buchmann, T. and Dotor, A. (2009). Constraints for a fine-grained mapping of feature models and executable domain models. In *Proceedings of the First International Workshop on Model-Driven Product Line Engineering (MDPLE 2009)*, Twente, The Netherlands. http://www.feasiple.de/workshop_en.html.

Buchmann, T., Dotor, A., and Westfechtel, B. (2008). MOD2-SCM: Experiences with co-evolving models when designing a modular SCM system. In *Proceedings of the 1st International Workshop on Model Co-Evolution and Consistency Management*, Toulouse, France.

Chang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., and Peterson, A. S. (1990). Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania.

Collins-Sussman, B., Fitzpatrick, B. W., and Pilato, C. M. (2004). *Version Control with Subversion*. O'Reilly & Associates, Sebastopol, California.

Kovŝe, J. (2005). *Model-Driven Development of Versioning Systems*. PhD thesis, University of Kaiserslautern, Kaiserslautern, Germany.

Object Management Group (2007). *OMG Unified Modeling Language (OMG UML), Infrastructure, V 2.1.2*. Needham, Massachusetts, formal/2007-11-04 edition.

Pohl, K., Böckle, G., and van der Linden, F. (2005). *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, Berlin, Germany.

Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E. (2009). *EMF Eclipse Modeling Framework*. The Eclipse Series. Addison-Wesley, Upper Saddle River, NJ, 2nd edition.

Tichy, W. F. (1985). RCS – A system for version control. *Software: Practice and Experience*, 15(7):637–654.

van der Lingen, R. and van der Hoek, A. (2003). Dissecting configuration management policies. In (Westfechtel and van der Hoek, 2003), pages 177–190.

Vesperman, J. (2006). *Essential CVS*. O'Reilly & Associates, Sebastopol, California.

Westfechtel, B. and van der Hoek, A., editors (2003). *Software Configuration Management: ICSE Workshops SCM 2001 and SCM 2003*, LNCS 2649, Portland, Oregon. Springer.

White, B. A. (2003). *Software Configuration Management Strategies and Rational ClearCase*. Object Technology Series. Addison-Wesley, Reading, Massachusetts.

Whitehead, E. J., Ge, G., and Pan, K. (2004). Automatic generation of hypertext system repositories: a model driven approach. In *15th ACM Conference on Hypertext and Hypermedia*, pages 205–214, Santa Cruz, CA. ACM Press.

Whitehead, E. J. and Gordon, D. (2003). Uniform comparison of configuration management data models. In (Westfechtel and van der Hoek, 2003), pages 70–85.

Zeller, A. and Snelting, G. (1997). Unified versioning through feature logic. *ACM Transactions on Software Engineering and Methodology*, 6(4):397–440.

Zündorf, A. (2001). Rigorous object oriented software development. Technical report, University of Paderborn, Germany.