Layered Configuration Management for Software Product Lines

Master thesis

Kroon, E.

Graduation Committee

Dr. P.M. van den Broek
I. Galvão Lourenço da Silva, Msc.
Prof.Dr.ir M. Aksit

Research Group

University of Twente
Faculty of Electrical Engineering, Mathematics and Computer Science
Software Engineering

Enschede, The Netherlands
November, 2009

Abstract

Software Product Lines are becoming more and more mainstream for software development. However, standard Configuration Management techniques are not designed to work on constant branching and variability that can be necessary in Software Product Lines. This thesis proposes a novel approach to Configuration Management that is a suitable match for Software Product Line Engineering. We propose a layered approach to Configuration Management, which permits the gradual adoption of product lines and the handling of the distinct Software Product Line Engineering phases. Our contribution includes preliminary tool support that can work with the proposed layered approach.

Acknowledgements

After a good start of my master study Computer Science, I would not have thought that my graduation project would take more then a year, however it did. Luckily my family and friends are very patient and kept on supporting me. The same is true for my supervisors.

After a while I accepted a full-time job beside my graduation project and while it was not easy to work on both, it provided me with some valuable practical input on software product lines and variability mechanisms.

In the end, it is all about finishing what you started and I am really happy that I finally finished what I started, resulting in a degree in Computer Science.

Erwin Kroon

Table of Contents

List of Figures

List of Tables

List of Abbreviations

AOP Aspect Oriented Programming

CM Configuration Management

DTO Data Transfer Object

EMF Eclipse Modelling Framework

IEEE Institute of Electrical and Electronics Engineers

LCM Layered Configuration Management

OVM Orthogonal Variability Model

RCS Revision Control System

SCM Software Configuration Management

SE Software Engineering

SPL Software Product Line

SPLE Software Product Line Engineering

VCS Version Control System

VM Variation Management

One

Introduction

1.1 Introduction

Software Product Lines (SPLs) are being used more and more, they are becoming a part of mainstream Software Engineering (SE) [24]. They are being used for groups of products or applications focused on a specific domain.

In an SPL there are different features that can be selected to compose a product. Depending on the technology of the product line, the composition of a product can be done in different ways. An example is custom configurations for the compilation process, something that is quite common for *C* or *C++* software. Another example is composition of a product after compilation, by having some kind of plug-in structure in the base part of the SPL; this approach is taken by the Eclipse project.

SPLs have two benefits: there is a shorter time to market (after the delivery of the first product) and the total cost after a couple of products are lower. However, these SPLs change over time, just like traditional developed applications. When these changes are not handled properly and maintaining the product line is starting to cost more then creating single systems, then their is no incentive to use the product line anymore and the initial investment is lost.

For single-system development, Configuration Management (CM) is available within SE. CM is responsible for technical and administrative direction to the software development, for instance resolving defects and deciding on new functionality.

Applying CM to SPLs is difficult; the added complexity of domain and application engineering makes CM harder, because there are more artefacts in the design and branches in the code. Also, evolution of the product line in total is something that is more difficult than single-system evolution. Relations between components are more complex and changing components could influence other products in the same product line. When changing a component there is not one product that should be tested again; every valid combination in which the component is present should be tested. This already assumes that you know which combinations are valid. All this means that it is key to have a good understanding of the impact of the maintenance activities performed.

To aid in understanding the impact of a change, traceability links can be used. If different artefacts are traceable to each other, this will help keeping the product line stable. A couple of examples are: traces between requirements

and implementation artefacts, traces between defect tickets and commits to a code repository or traces between features of a SPL and the implementing artefacts. Current implementation of CM tools and approaches are not designed to effectively handle the added complexity introduced by SPL. In this thesis we will evaluate the problem.

1.2 Problem Statement

There is general agreement in the literature that using trace information in CM is a good solution for SPLs [22, 18, 27]. Consequently, there should be traceability support in tool-support in CM tools for Software Product Line Engineering (SPLE). However, tool-support that uses traceability is not widely spread in the literature and when there is support available it is tightly coupled to a traceability reference model[23, 21].

Just as every software product, an SPL will change over time. This means that changes have to be implemented and maybe propagated across domain engineering and application engineering phases. Krueger [20] and Yu [35] describe Variation Management (VM) and which artefacts should be used for evolutionary propagation in the domain. Where they do distinguish between custom components and variants, they do not address the fact when something should be promoted from custom component to variant. So, while every artefact is under *configuration* management, it is needed to know when something will be put under *variation* management.

In addition, Yu describes various ways of evolution propagation, including a *propagate* link, but he fails to mention, *how* to propagate. How do you inform stakeholders that they have to act on a change, for instance?

Riebisch [27, 26] states that feature diagrams can be used to reduce complexity when tracing and additionally can help to improve understanding of the system.

The state-of-the-art shows us that traditional CM is not enough for SPL. There are more dimensions for change then in traditional software development and handling evolution is as important in an SPL as in traditional development. A suggested solution is using traceability in addition to CM, but work-flow processes and tool support incorporating traceability are not available within an SPLE context. This leads to the following problems statements for our research.

Problem Statement 1. *How to adapt standard configuration management practises for enhanced variability management?*

Problem Statement 2. *Which complementary support is necessary on current practises, to support problem statement 1?*

1.3 Approach

Based on the summary of problems above, we propose an approach with subtasks to be full-filled.

- *Investigate on current approaches of CM and variability management for SPLs.*

- *Improve existing theory for CM and variability management in SPLE.*
- *Provide process and tool support for variability management.*

With the result of the investigation on current approaches we will propose an adapted and new approach. We will use this theory to design an and implement a software prototype that provides tool-support for CM of SPLs.

1.4 Outline

This thesis is structured as follows. Chapter two will introduce relevant concepts to our work, followed by a chapter on current research on the subject. Chapter three introduces our novel approach called Layered Configuration Management. The next two chapters give us requirements for tool-support for our model and a prototype implementation for these requirements respectively. In chapter seven we end with an evaluation of our work and recommendations for future work.

Two

Relevant Concepts

In this chapter we explain relevant concepts for our work. We introduce Software Product Line Engineering (SPLE) with the related concepts domain and application engineering. We introduce feature diagrams and use them to depict Software Product Lines (SPLs). Furthermore we introduce software evolution and standard Configuration Management (CM). Finally, we explain traceability and introduce to the MobileMedia product line.

2.1 Software Product Line Engineering

SPLE applies traditional SE techniques on SPLs. An SPL is defined as follows:

Definition 1 (Software Product Line)**.** A set of products sharing a common, managed set of features that satisfy the specific needs of a particular market segment (domain) and that are developed from a common set of core assets in a prescribed way [24].

This means that SPLE is responsible for the creation of software with high standardisation and high customisation, also described as software *mass customisation*. Mass customisation is the *large-scale production* of goods tailored to *individual customers* needs [24]. An example of mass customisation outside the scope of software development is car manufacturing; it is possible to purchase a certain standard model with a specific colour, interior, radio, rims and so on. The term *variability* is used to indicate the points that can change. Van Gurp et al. [13] describe this concept in more detail.

SPL is a two-phased process, the first phase is engineering the domain of the SPL, which composes the general part or baseline of the SPL. The second phase is the engineering of individual applications. We will elaborate on this in the following sub-sections.

2.1.1 Domain Engineering

The concept of mass customisation implies that there is a basis for every product line that is consistent for all derived products. This part is developed by domain engineering and the resulting artefacts are the *domain artefacts*. Regular SE activities are included into the domain engineering phase, *requirements*

engineering, *design*, *implementation* and *testing*. The result is a set of usable artefacts in the form of design documents, interfaces, software components and test coverage.

We will illustrate domain engineering with an example. Suppose we want to create a media management application for mobile phones. For this we use Java for mobile phones (J2ME), which is almost platform independent, but still needs specific libraries and compilation steps depending on the brand and model of the mobile phone required to run the application. These libraries imposes constraints on the build process of the application, depending on the targeted mobile phone.

Apart from constraints on the build process, there can be constraints on the functionality (or features as it is usually called in SPLE) imposed by the hardware. Taking photos with the built-in camera of the phone from within the application is impossible when the phone does not have a camera. Having support for playback of certain media types is impossible if the hardware in the phone does not have the required specifications.

All the requirements for components and implementation of components are created in the domain engineering phase of the product line engineering process.

2.1.2 Application engineering

The actual creation of a product is executed in the application engineering phase of a product line. To illustrate this with the example of domain engineering: for a telephone with recording capability a version of the application that can start recording videos can be build.

In case of the need of functionality that is not available from the domain engineering phase, a decision has to be made. Is this extra functionality added to the results of domain engineering, or is it only implemented for a specific instance (build) of an application. It stands to reason that adding nothing new to the domain is probably not correct and adding everything new to the domain probably also is not.

If existing functionality should be changed to satisfy requirements for an application, than this should be done with extreme caution, because this could influence already existing products.

Products

Until now we have called the result of a product line an *application* to have a relation with single-system-development. In the context of SPLE, we think it is more correct to talk about a *product* or *products* to emphasise the difference with the result of single-system-development.

2.1.3 Feature Diagrams

A feature is a characteristic of a system visible to an end-user. An SPL consists out of common and non-common features, these features can be implemented by sub-features. To display the different features of an SPL, feature diagrams can be used. There are many variations on the basis feature diagram in the

literature. Kang et al. [19] explain how to use feature diagrams for domain analysis, which is a good match with SPLE. In addition, Schobbens et al. [28] clarify the use of feature diagrams with respect to SPLs and elaborate on different possible approaches.

In Figure 2.1 we show you a very simple example feature diagram as part of an example for this whole thesis.

2.1.4 Orthogonal Variability Model

Pohl et al. [24] propose an SPL specific alternative for feature models in something they call the Orthogonal Variability Model (OVM). In this model, it is possible to link to different artefacts, such as UML use-cases or requirements. In addition, it is possible to have private features and it is possible to add constraints between variants and variation points (which is the authors terminology for features and their sub-features). This means that the OVM is able to display features and also has tracing information to other steps in the design. A problem with their approach is that there is no tool-support available, which means that drawing all the diagram and handling the traces is a lot of work.

2.1.5 Variability Mechanisms

Product lines have locations in their code-base that offer variability. We call this variability points. The technique used to implement this variability is call a variability mechanism. There is a wide variety of variability mechanisms possible; we will discuss conditional compilation and aspect oriented programming.

Conditional Compilation

With conditional compilation, the source code is generally preprocessed to include or omit certain parts of code. By having variables for different variants in the code, specific code for variants can be explicitly enabled or disabled.

Aspect Oriented Programming

With Aspect Oriented Programming (AOP) it is possible to use advices and join-point declarations for implementing variability. Examples are changing functionality of base modules by wrapping code around it or linking implementation modules to interface modules.

2.1.6 Product Line Decay

In the literature on SPLE many research is motivated by the prevention of *product line decay* or some similar concepts. The decay or degradation of a product line means that its use is getting less or non-existent. For a product line to become profitable, more products should be sold then a comparable application in single-system-development, because of higher upfront costs. In fact, the production of a single product should cost less then the production of a single comparable application, otherwise single-system-development would be more profitable. If the real-world domain of a product line changes, the domain of the product line should change also. Already produced products

should be upgraded with the changes to the domain if needed. A bug fix to one product should be propagated to other products and so on. For every action on the product line or products of the product line, the time needed to perform this changes on other parts of the product line should not increase over time. If this happens than product line decay is happening in the product line.

2.2 Software Evolution

Definition 2 (Software Evolution)**.** Software Evolution is the application of software maintenance activities and processes that generate a new operational software version with a changed customer-experienced functionality or properties from a prior operational version, where the time period between versions may last from less than a minute to decades, together with the associated quality assurance activities and processes, and with the management of the activities and processes; sometimes used narrowly as a synonym for software maintenance, and sometimes used broadly as the sequence of states and the transitions between them of software from its initial creation to its eventual retirement or abandonment. [7]

The above definition links software evolution to the maintenance of a software product. This maintenance can be subdivided into four traditional categories, as specified in the ISO/IEC 14764 [17]:

- *Corrective Maintenance*: Reactive modification of a software product performed after delivery to correct discovered problems.
- *Adaptive Maintenance*: Modification of a software product performed after delivery to keep a software product usable in a changed or changing environment.
- *Perfective Maintenance*: Modification of a software product after delivery to improve performance or maintainability.
- *Preventive Maintenance*: Modification of a software product after delivery to detect and correct latent faults in the software product before they become effective faults.

2.3 Configuration Management

Definition 3 (Configuration Management)**.** A discipline applying technical and administrative direction and surveillance to: identify and document the functional and physical characteristics of a configuration item, control changes to those characteristics, record and report change processing and implementation status, and verify compliance with specified requirements.[1]

CM helps to control software evolution and maintaining its integrity, by defining tasks that have to be performed on development artefacts. For SPLE these tasks have to be executed on the domain engineering and the application engineering stages. Domain engineering should identify development artefacts, whilst application engineering should handle identification of development artefacts used in a specific product.

The composition of a product out of components leads to added complexity in identification of development artefacts; a product could need a specific version of a development artefact (often a component). These independent requirements of products for components requires CM not only to direct the development process in time, but also in space.

CM includes a couple of tasks, which are[8]:

- *Identification*: an identification scheme is needed to reflect the structure of the product. This involves identifying the structure and kinds of components, making them unique and accessible in some form by giving each component a name, a version identification, and a configuration identification.
- *Control*: controlling the release of a product and changes to it throughout the life cycle by having controls in place that ensure consistent software via the creation of a baseline product.
- *Status Accounting*: recording and reporting the status of components and change requests, and gathering vital statistics about components in the product.
- *Audit and Review*: validating the completeness of a product and maintaining consistency among the components by ensuring that the product is a well-defined collection of components.
- *Manufacture*: managing the construction and building of the product in an optimal manner.
- *Process Management*: ensuring the carrying out of the organisations procedures, policies and life cycle model.
- *Team Work*: controlling the work and inter-actions between multiple users on a product.

The last three items above have a more organisational process focus. We think it is important to include these, because SPL oriented development implies dividing the work between teams, e.g. a domain engineering team and a team for every product.

2.4 Traceability

Definition 4 (Traceability)**.** The degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another [16].

An example of a predecessor-successor relationship is the relationship a changed source code artefact has to his original implementation. Example of a master-subordinate relationship are the link between a requirements document and the implementing artefacts or the link between a feature and the implementing artefacts.

2.5 Sample Product Line

Having introduced the relevant concepts we will introduce an example product line that we can use for examples in the rest of this thesis.

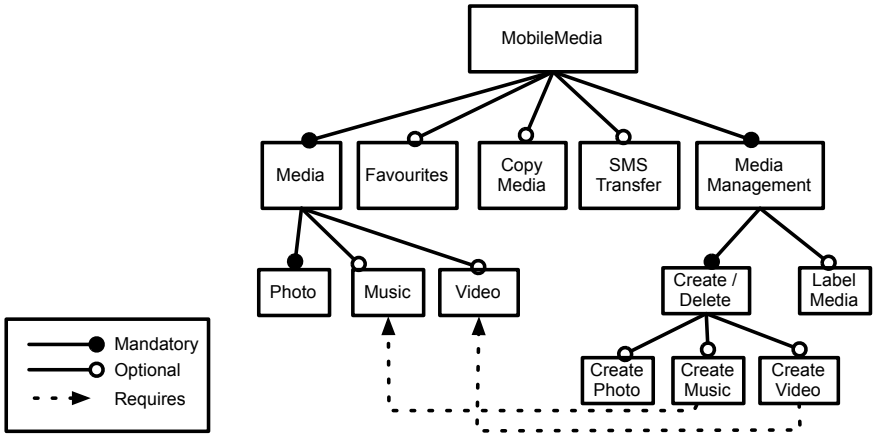

Figure 2.1: Feature diagram of the MobileMedia product line

The product line used in the examples across this thesis is called MobileMedia, and the result of this product line is a product that can work with different media files on a mobile phone. Dependent on included features, it can only view media files, edit/create them and/or send them by SMS.

Figure 2.1 shows the feature diagram of MobileMedia. The constraints in the feature diagram are as follows:

- If a feature is mandatory it should be selected if its parent is selected.
- For every selected feature, its parent should be selected.
- If a feature with an outgoing requires constraint is selected, the required feature should also be selected, still adhering to the previous constraints.

This product line uses J2ME, a Java edition intended for mobile phones. In contrast with the Java standard edition, which is platform independent, in J2ME every mobile phone brand has some specific details needed for implementation, because there is no generic Java virtual machine available for mobile phones. Therefore, depending on the mobile phone that the product should run on, certain fixed for a product should be applied.

The MobileMedia SPL is available in two editions, which adopt distinct variability mechanisms: AOP and conditional compilation. These two separate implementations have the same functionality. For both versions a model-view-controller pattern is chosen as the implementation pattern[12].

Three

Research

In this chapter we evaluate the current state-of-art on product lines and CM and traceability. After that we will discuss our opinion on the start-of-the-art.

3.1 State-of-the-art

In this section we will describe the state-of-the-art on CM in relation with SPLE. In addition we investigate the applicability of traceability in combination with CM and SPLs.

3.1.1 Configuration Management and Product Lines

Section 2.3 shortly introduced that the tasks of CM have to be executed *in time* and *in space*. To clarify this a little bit more we explain the concepts *variation in time* and *variation in space* used by Krueger [20].

In traditional single-system development there is the notion of *variation in time*. This means that there are multiple versions of a software product. Maybe there is some branching in the code, but this often for refactoring or adding functionality without breaking the existing product.

In SPLE there is also *variation in space*. This means that at any fixed point in time (so not temporarily), there is a difference between different products in the product line. This is inherent to the *mass customisation* of a product line; certain features will be present in one product, but not in others and the other way around.

Variations in space make CM a more challenging task, resulting in more risks, costs, and time needed. Krueger proposes a divide and conquer system in which he extends CM with two extra tasks, resulting in the perspective of VM shown in Figure 3.1.

The first extra task, *component composition* (shown in the horizontal box left of products), is needed when there is variation in space. When every component is in the baseline of the product line (the part that is developed in sequential time, as with single-system-development), component composition is superfluous. If it is needed, it can be implemented in two ways: a *shelf* approach or a *context* approach. The shelf approach puts every available component as a reusable part on a shelf. Guidance is needed for determining correct and consistent compositions, because there may be combinations of

	Sequential Time	**Parallel Time**	**Domain Space**
Files	version management	branch management	variation point management
Components	baseline management	branched baseline management	customization management
Products	composition management	branched composition management	customization composition management

Figure 3.1: Krueger's model for SPL CM [20]

components that are not correct. The context approach will only store valid compositions of components. As the product line evolves, more compositions can become available.

The second task is *software mass customisation* (shown in the vertical box beneath domain space), which introduces abstractions for identifying variation points in development artefacts (variation point management), instantiating them in components (customisation management) and composing products (customisation composition management)

Krueger advocates that every single file from the production line should be under VM. This is needed for *reproducibility*; every product in the product line can be reproduced as it was in some point in the past, using only artefacts under VM. This means, that every file should be at least under basic CM.

Krueger also advocates to not put product instances under CM, because he states that VM is for handling the *production line* and not the *product line*. A separate engineering activity should be in place to propagate changes from the product instances back into the core assets and other products. The reason is that with a large number of products, management becomes a very time consuming and difficult process. Consequently, this means that Krueger's model only handles reproducing products, but not evolving the production/product line from changes in the resulting products.

Yu and Ramaswamy [35] adapted Krueger's model for CM and included product instances in VM. In their model, there are two options for evolution propagation. If there is a change to a core development artefact from a product, this is propagated back to the core artefacts and then to the other products. If there is a change to a custom part of a product, this is reported to the custom parts repository but not into already existing products, only new products will use the new version; it is possible that maintainers of already deployed products will use the report to improve their own product. This model allows configuration management of both the SPL as products of the product line.

3.1.2 Traceability and Software Product Lines

In [27] the authors suggest to use traceability to prevent product line decay. In his approach, traceability is used to visualise relations and constraints between user requirements and common features with their dependencies; constraints within architecture, design and implementation; and constraints between features relevant for feature configuration. Their model suggests to add traceability to the meta-model of the models, so the trace information is stored in the *original model*. For this approach the authors assume that UML models are used. How to record trace information is not in the model, it only expresses that recording design decisions is a very abstraction demanding tasks, as all the design details need to be incorporated in the definition of the traces. As a solution, Riebisch suggests to record design decisions by means of reverse engineering.

Maintaining traceability links for a product line is a difficult task for multiple reasons [26]. Firstly, links have to be created by humans, as current tool-support is not intelligent enough. Secondly, the amount of links to create can be too big. The first problem is not solved in the article, but the second can be solved by using feature diagrams as an intermediate step for traceability links. The solution is to link implementation details to features and requirements to features also. Doing this limits the number of links needed to connect implementation artefacts to requirements; m requirements belong to 1 feature and n implementation artefacts can be linked to a feature also. For one feature, the number of links will be $n+m$. Without an intermediate feature the number of links needed in the worst case will be $n * m$.

Jirapanthong and Zisman [18] propose a traceability reference model that makes tool support possible. In order to make it work they require the use of FORM (Feature-Oriented Reuse Method) [21], which means that this specific development method is required to have appropriate tool support. This, in turn, means that this solution is only a solution in projects that use the FORM method or are willing to adopt it.

Apart from the reference model of Jirapanthong and Zisman, there are more traceability reference models. It is likely that, for appropriate tool support to be viable, the tool support should be connected to a specific reference model. In turn, a traceability model could require the use of a development model.

Mohan and Ramesh [22] propose a traceability knowledge system based on a requirements traceability model proposed in [25]. They implemented tool support for this, based on a tool called *Remap*. The tool is designed to capture the rationale behind design decisions. These rationales cannot be linked to versions, as the tool is not specific for SPLs, leaving space for improvement.

3.1.3 Software Product Lines in Organisations

With the added complexity of modern SE techniques and principles, it becomes more and more important that managers, developers and engineers accept and understand the process and structure of an SE project. For SPLE, the development structure is different from single-system development. Our opinion is that the organisational structure and its processes should reflect this. Böckle et al. [6] and Mohan and Ramesh [23] address this.

Böckle et al. give an adoption plan to institutionalise an SPL in a company.

They describe a strategy that defines the stakeholders, after which the authors give some pointers to find the current state and the desired state and how to reach the desired state. Thereafter, the process should become the organisation (hence *institutionalised*), making sure the whole organisations culture is built around SPLE.

Mohan and Ramesh did a case study to see how change management can be done effectively for product lines. They concluded three things: *modularise changes and variation points*, *track scope and life of variations*, and *facilitate reuse based on knowledge sharing*. The last point is of organisational matter; if software engineers (informally) share what they are working on, then they all know what the can reuse.

3.2 Traceability in Configuration Management Tasks

Standard CM tasks do not describe explicitly how to use traceability information and source files under version control. Therefore, we should describe how to use this information in the Configuration Management tasks.

- *Identification.* By putting source files under version control, it is possible to identify them. The repository location combined with the path within the repository and the file name should be a unique identifier in most situations.
- *Control.* Traditional CM is sufficient for this when we only apply this to the baseline (or domain part) of an SPL. If changes to the baseline need to be propagated to the products, we can use a version control system for non-customised artefacts and probably also for customised artefacts (by using some patch based system if the changes are not very big). Pushing changes from the products to the baseline is something that is less trivial and this should probably solved on a technological - and organisational level.
- *Status Accounting.* With the use of trace information, change requests to products could be propagated to the baseline. The result of this propagation could be a simple aggregation of items, but the changes can be traced back to original requirements, which could help the maintainer of the product line.
- *Audit and Review.* With trace information, a tool can verify if changes in the requirements have been propagated to related artefact. Or, if bug fixes in a deployed product also have been implemented in other products.
- *Manufacture.* By linking implementation artefacts to features it is possible to select the artefacts for a build of a software product by selecting features. This, for instance, is very useful for a functional consultant, because he will discuss the software product with the customer in terms of functionality or features and not in technical details.
- *Process Management.* By maintaining a relation between features or requirements of software and implementation components it is possible to create a list of affected (other) requirements when a component needs to change for a requirement change. This results in better insights into the amount of work to be performed.

- *Team Work.* Traceability information is very useful for team work. When a developer commits his work to a repository, and thereby implements a (part of a) feature of fixes a bug that is logged in a issue management system, a link can be created between the commit and the issue. This makes it possible to review only relevant code for the issue by displaying it with the issue itself, using the available trace.

3.3 Software in State-of-the-art

FeatureMapper [14] is a tool that supports mapping features to solution artefacts expressed in EMF/Ecore-based languages. This could be a UML representation, but also a domain specific language representation. The focus is on combining SPLE and model-driven software development, but not so much on CM for SPLs.

3.4 Models in State-of-the-art

Krueger[20] takes basic CM as a core for his model and defines four sub tasks (see Figure 3.1). These sub tasks are based on sequential and parallel time activities, and files and components. We think that this division is clear and it is easy to understand the sub tasks. However, in the approach of Krueger, the addition of *Products* (as a propagation of *Files* and *Components*) and the addition of *Domain Space* (as a propagation of *Sequential Time* and *Parallel Time*) are poorly stated. We can say that *files* create *components* and *components* create *products*, so this extension makes sense. The relation of *Domain Space* with the other components on the top row is not obvious. The relation between *Sequential Time* and *Parallel Time* is clear, and one could see how the domain relates to them, but they do not relate in a similar way as *Files, Components* and *Products*. Furthermore, the current partitioning puts *Products* and *Domain Space* as opposites of each other, but in the concept of product line engineering, they relate very strongly.

Yu and Ramaswamy [35] propose an adaption of Krueger's model that also puts already released products and possibly adapted products under configuration management. This is explained in section 3.1. Yu and Ramaswamy focus more on the evolution part of SPLE and on how to keep an SPL in shape.

In their model (see Figure 3.2) it is possible to choose between including product instances and product-in-use into the CM model. The essence is that the users of the product line can choose if they keep information on already composed products and use possible bug fixes for other products also. This layered approach gives a user of the model the option to gradually support their approach.

While the model of Yu and Ramaswamy addresses the problem of changes outside the domain and we like the modular build-up of their model, the model is not very clear on specific CM activities and their order.

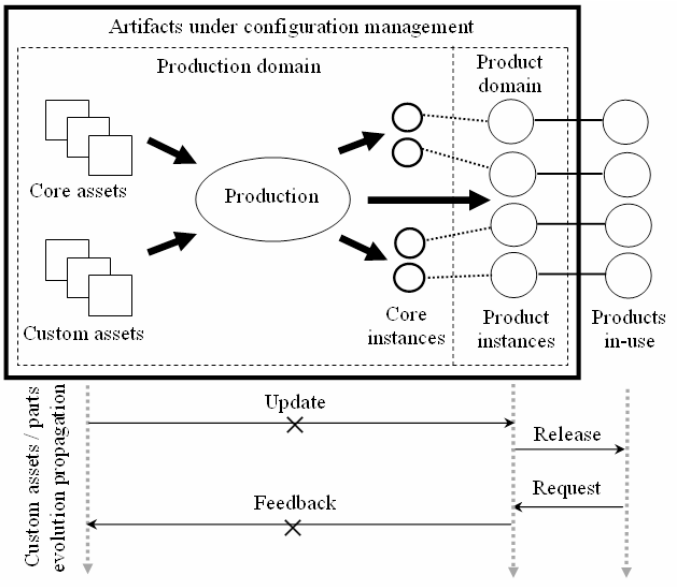

Figure 3.2: Products under Configuration Management[35].

3.5 Discussions

We propose a couple of changes in the model of Krueger. Firstly, we would like to see the activities in the model in some kind of order. This would remove the need to categorise the activities in a less then ideal way (as we expressed in the previous subsection).

Secondly, the possibility to customise a product after composition out of standard and customised components should be available. Without this ability, it is not possible to change products outside the domain. A situation where this is useful, is the creation of a user interface that is different for every customer.

Finally, including Yu and Ramaswamy's approach allows for gradual adaptation within Krueger's approach.

Four

Layered Configuration Management

This chapter will introduce a novel approach that solves the problems addressed in chapter 3, related to SPLE and CM. We mentioned missing ordering in the model of Krueger and would like to address this in our own model. From Yu and Ramaswamy we reuse their modular approach and incorporate this by adding layers in our own model. As a result Layered Configuration Management (LCM) is introduced.

For LCM, the focus is on the activities that distinguish traditional CM from CM for product lines in our layered model. As for a modular approach: it should be possible to incrementally add our model into a product line, layer by layer.

4.1 A Model for Layered Configuration Management

We will describe the activities in the LCM model and give the relation to traditional CM activities from chapter 2. The activities are grouped into four levels, numbered one to four. Figure 4.1 depicts the resulting diagram.

The list of activities in our model is created by taking the activities of Krueger and renaming them for clarification plus the addition of *product customisation management*, which is a replacement of *composition customisation management* as we think that *composition customisation management* is redundant with *customisation management* and *composition management* (see Figure 3.1).

4.1.1 Level One

The activities *version management*, *branch management*, *baseline management* and *branched baseline management* are contained in layer one. These activities are taken directly from Figure 3.1.

Version Management

Having version management for an SPL ensures that it is possible to identify different versions of a file. It is an extended version of the *identification* task in traditional CM. In addition to identification of an artefact it is possible to identify an artefact having a specific version.

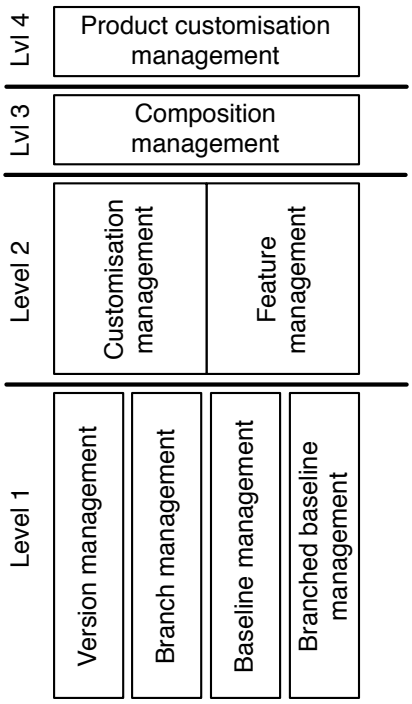

Figure 4.1: The LCM model.

Version management is needed by most of the management activities in the higher levels of our model to identify their input (see Figure 4.1).

In the MobileMedia product line, every artefact is versioned and thereby identified. For some old Nokia telephone no new version of the J2ME libraries is released, so it is not possible to update the phone with current versions of the Media management feature. Therefore an old stable version can be selected, because of version management.

Branch Management

Branch management is strongly related to version management; it makes it possible to (temporarily) have different versions of a source file, a component or a complete feature. The reasons for having a branch are numerous, but as requirements change, software needs to change and during a change it is still required to be able to create working product, which means that changes should be done in a branch. Branch management takes on the task of *controlling* the changes to (the baseline of) a product line.

Branch management without the implementation of a higher layer of the LCM would be the same as branching in a VCS such as Subversion, but when for example layer two is implemented, the branching of a feature would be possible.

For MobileMedia, when a feature is updated, a developer could branch a existing feature (e.g. create video) and develop on that. When it is finished the complete feature can be merged and used from now on.

Baseline Management

Introducing baseline management for an SPL ensures that there are mechanisms in place guaranteeing a stable baseline. Baseline management is ensuring the quality of actions performed for the domain engineering of a product line.

When comparing baseline management to traditional CM, then baseline management is responsible for the *control* and *audit and review* tasks. Consistent components in the baseline can be ensured by having testing mechanisms in place and, likewise, the complete baseline can also be tested.

An example for baseline management in the MobileMedia product line is automatic testing of the complete baseline after a change to (some of) the artefacts of the product line.

Branched Baseline Management

Branched baseline management is responsible for the branch management of the baseline of an SPL. Whereas branch management is responsible for temporary branching of features, products and customised products, branched baseline management is responsible for the management of persistent branching.

If we compare this to version management, the difference is in the development status of the involved components. In case of version management, an old version of a stable feature is needed that is not updated anymore. In case of branched baseline management, two components implementing the same functionality for different platforms that are both actively developed are needed.

For the Blackberry implementation of MobileMedia, some specific code is needed to handle the creation of a photo, because it does not use a J2ME specification but some custom libraries. Therefore a different version of the implementing component is needed. This is always needed and updated with new functionality, therefore this is not supposed to be handled with version management.

4.1.2 Level Two

On level two of LCM is *feature management* and *customisation management*. Level two introduces the concept of features into our layered model. It is on this level that we start working with features. In contrast with Krueger, we replaced his variation point management with feature management, but we reused his customisation management.

Feature Management

Feature management is a replacement of Krueger's variation point management. The term variation point management is not complete for our purposes. Its name focuses on the management of variation points in development artefacts, whereas it is a good possibility that a product line is implemented without variation points in the development artefacts. However, every product line will

have features and a relation between features and implementing source-code artefacts resulting from domain engineering.

Managing features in the product line is the first responsibility of feature management. Ensuring selection of valid combinations is another responsibility.

For the next responsibility, feature management needs the activities in level one. Feature management needs to link features to development artefacts, to make it possible to correctly identify features and relating artefacts. Depending on the details of implementation, only version management could be needed. If there are some difficult platforms to support, branched baseline management could be needed.

If, for the customisation of components some extra information is needed (for example information about variation points), this is handled by feature management also. If this is needed for variation points, the actual customisation will still be the responsibility of customisation management, which we will discuss in the next section.

Now, remember Krueger's [20] two approaches for composition management. One of them was the context approach. If this approach is chosen then this is a responsibility for feature management. Valid combinations of features should be defined and for every valid combination of features extra artefacts should be linked if necessary. We will explain this with an example at the end of this section. For Krueger's shelf approach we have composition management.

If we would like to relate feature management to traditional CM, it would be related to *identification* and *control*. Identification, because with an SPL there is an extra need for identification of variation points in addition to file identification. Control, because consistent component instantiation depends on the allowed combination of features or variants.

Feature management for the MobileMedia case would be the management of the feature diagram when using a very low-tech implementation of feature management. All the artefacts could be linked in text files per feature or something alike. This, of course, is a lot of work, but would be a valid implementation of feature management for the MobileMedia product line.

Suppose that we want to use the context approach for the selection of valid products. (In fact, this is not so strange, because the number of valid feature combinations is not very high in Figure 2.1.) If the implementing technology for the product line would be Aspect Oriented Programing, then we could apply extra functionality to some basic functionality by applying advices to this basic functionality.

When we select the *Create Photo* and *Create Video* features, we apply two advices in the components of the *Media Management* feature. These advices overlap, which is not a problem when we select only one feature, however, now it is because we need to order this advice. Therefore, if we select a combination of features including both *Create Photo* and *Create Video*, we have to select an extra development artefact that declares the ordering of two already included advice artefacts.

Customisation Management.

Customisation management in the domain space is customisation management as defined by Krueger [20]. This is the instantiation of components by select-

ing and configuring the available variation points of the component and the corresponding files. The instantiation or build process uses information from feature management to select the required files and possibly generate build-files or configure the build process.

If no variation points are available, then customisation management is only responsible for gathering components from their repositories.

The link between customisation management and feature management is as follows: feature management provides the available variation points and ensures that valid combinations of variation points are selected. Customisation management is responsible for using this information and instantiating components from the SPL baseline.

MobileMedia is a J2ME application; for J2ME there are tools available for conditional compilation, where the source code gets preprocessed. This is something quite common for C or C++ programmers. If we compare this with the example given at feature management, then instead of selecting extra files for a combination of features, we can activate some variables and activate parts in the code for a specific feature.

4.1.3 Level Three

Level three consists out of one task, *composition management*. In contrast with Krueger, we do not differentiate between *customisation composition management* and *composition management*; customisation composition management is a combination of customisation - and composition management so we can provide all functionality already.

We also differentiate from Krueger with *Branched composition management*, which we do not have. There is no replacement activity for it, it is contained in *branched baseline management* on level one. Krueger has branched composition management to offer branching of compositions, whereas in normal composition management of Krueger only provides sequential versioning. We offer both by handling branching and versioning on level one. This is of course also possible for *feature management*, even if we did not mention this explicitly.

Composition Management

Composition management is the composition of a product out of components. This can be be components from the baseline or customised components. With valid compositions it is possible to implement the *shelf* approach with valid combination of features or complete products.

It is actually possible to define a subset of valid features and add extra artefacts to this composition. It is also possible to add artefacts to complete product configurations.

By defining valid combinations and products, it becomes very easy to reproduce a specific product and produce specific bug fixes of product and applying them consistently.

For MobileMedia, in case of defining all possible combinations and making them work, compositions can be created for every mobile phone (or brand). For instance, lets say that only a BlackBerry offers an API to send a file by

SMS directly. In that case only the BlackBerry compositions contain the feature *Send by SMS*.

4.1.4 Level Four

On level four we support management of customisations not possible within the domain engineering results of a product line. This is done with the activity *product customisation management*, which is an addition compared to Krueger.

Product Customisation Management

Product customisation is the customisation of a product outside the product line; changes that cannot be made by instantiation of variation points. This replaces *customisation composition management* in Krueger's model.

There are two possible customisations: customisations outside and inside the domain. When the customisation is inside the domain it would be smart to include this customisation into the product line; making it a variant for customisation and feature management. This a customisation as described by Yu and Ramaswamy [35]. Tool support for product customisation management should support promoting custom code to a variant or variability point / feature.

A customisation outside the domain of the product line is less interesting. Of course, maintaining these customisations has to be done in such a way that the are easily re-applicable; upgrading the product is easier that way.

When a customer for MobileMedia needs a specific gui, because they offer a phone with some special display, then this can be implemented as a product customisation. After an update, because this customisation is managed, it can be applied automatically.

4.2 Configuration Management Focus Points

With limited time at hands we will have to focus our time on a couple of activities within CM. The activities we will focus on, will be *identification*, *manufacture* and *audit and review*. We think that these three subjects provide the most added value. Within audit and review we want to focus on testing of the product line and products. Of course, validation of requirements is also important, but no special actions are needed in comparison with single-system-development.

In this section we will describe actions performed on every level for the selected activities. By depicting this, it is possible to extract requirements for tool support.

4.2.1 Level One

Level one focuses on identification of development artefacts. Every development artefact is handled on this level. Manufacture activities are not defined for this level. On this level only testing of standard components is possible — components that do not need customisation.

4.2.2 Level Two

On level two, identification of variation points and features in an important task. The identification information can be used for manufacture activities in customisation management. Manufacture activities on this level consist mainly out of component instantiation / customisation, but creation of build files is also possible, all dependent on the implementation of the product line.

Customisable components that need testing have to be instantiated with a specific variant before testing is possible. Instantiating for testing is an important activity here. This is something where tool-support can help.

4.2.3 Level Three

On level three, identification is identifying feature compositions and complete products. Manufacture activities are the building of products.

For testing, however, there is a task on level three. Tests that are specifically for combination(s) of features are maintained on this level. Setting up the environment for executing the tests is a responsibility for composition management.

4.2.4 Level Four

On level four, we have to distinguish to types of customisations before we can continue. There are customisations that are within the domain of the product line and there are customisations that are outside the domain of the product line. Customisations outside the domain should be easy applicable only (for upgrading reasons), therefore they should only be identified.Customisations inside the domain, on the other hand, should be promotable to variants or commonalities in the domain, if needed.

It has to be possible to identify customisations to development artefacts, this could be in the format of patch files.

Manufacturing and testing activities on level four are comparable to level four, only with a set of customisations applied.

4.3 Traceability in Layered Configuration Management

Yu and Ramaswamy [35] use evolution as a key concept to show that SPLs change based on change requests and that such a change request should be *traced* back to the product or implementing component.

This concept of tracing is important for us also. Without tracing, it is impossible to handle evolution of the product line in such a way that product line decay is prevented. This decay can be a result of numerous things: the product line does not keep op with changes in the domain, or every single adaption for clients becomes a new variant of a feature and variants and / or features that clearly do not belong to the domain still get added.

When product line decay occurs the motivators for a product line, reduced costs and shorter time-to-market, become less strong or even invisible (as using the product line would mean increased costs).

In our opinion we should not limit our model to a specific set of trace-types. It is up to the developers and engineers of a product line to decide if certain

traces are important. When it is decided that certain traces are needed, then they can be implemented for the product line.

Suppose that the requirement documents for MobileMedia are in a Subversion repository. By committing with specific commit messages mentioning the requirements file a simple trace between requirement and implementing artefacts can be created.

4.4 Conclusions

In this chapter we have introduced a novel approach for CM for SPLE. We have added three layers upon standard CM; one for the features of the product line, one for compositions of features and one for customisations of products. We propose not to use a limited set of traces within our model but to add intrinsic support for generic traces that can be customised to the needs of the product line.

Five

Requirements for Tool Support

The theoretical foundations in the previous chapter were necessary to give input for the creation of tool support for CM for SPLs. This chapter will give the requirements for tool support.

5.1 Requirements

Following from the model and activities, we can define requirements for tool support. This section contains the requirements for tool support of the layered CM model. These requirement are split into functional and non-functional requirements, where in addition the functional requirements are split into requirements for every layer of the model.

5.1.1 Functional Requirements

Basic Configuration Management

For basic CM (level 1), the tool should support version control for all the artefacts (files, for instance) in the product line. When we assume that all the files are in text-format, it should be easy to have requirements and diagrams under version control also, because we can handle them as source files.

Ideally all the functionality of a well established VCS as Subversion is be present. When we look at a very basic level, we would like to be able to handle the basic situation of check in and check out, together with simple branch management. In case of conflicts, some form of conflict resolving should be present.

Above information leads to the following requirements:

Requirement 1. *The system shall support checkin and checkout of files or directories.*

Requirement 2. *The system shall support branch management as in Subversion.*

Requirement 3. *The system shall support some (simple) kind of conflict resolving.*

Feature Management

Feature management (level 2) deals with variation points and features. Therefore, a tool has to handle features and variation points.

Variation points should be registrable and linkable to artefacts in level one. Features also need registering and linking to variation points.

Variation points and features need to have optional links to artefacts that are not instantiation related. This can be used to link requirements and so on. When developing on the product line these artefacts can also be checked-out, but when building a product with a specific feature and/or variation point, these artefacts can be omitted.

Features and/or variants can be conflicting, to support this, the tool will support feature trees to validate a specific product configuration. This information is used by customisation management.

To work on a feature one would like to checkout a feature (or multiple features). Or some code should be added to an existing / new feature. In short, some kind of CM on feature level should be possible.

Requirement 4. *The system shall support registration of features. Features should be registered in form of a feature tree.*

Requirement 5. *The system shall support linking of implementation artefacts to features.*

Requirement 6. *The system shall support linking of optional artefacts to features.*

Requirement 7. *The system shall support feature trees for constraint handling between features.*

Requirement 8. *The system shall support checkin and checkout of features, as well creating features from implementation artefacts or adding/removing implementation artefacts to/from features.*

Customisation Management

Customisation management (level 2) is responsible for initialisation of features / variants and composing complete products. Initialising is dependent on the specific implementation of the product line. Therefore, the information stored for a variation point is dependent on this.

There are cases where only variable names are needed, in case of preprocessing code, or maybe only the file name (information that is already present by the link between variation point and artefact) when using aspect oriented programming techniques.

Therefore based on the variation mechanism, the tool should have different implementations for initialising features and their components. These implementations should plug-able.

Requirement 9. *The system shall support different variability mechanisms.*

Requirement 10. *The system shall offer generic support for instantiating features and their components.*

Composition Management

Composition management (level 3) is responsible for keeping track of composed products or valid compositions. Tool support for this means that for a composed product it should be clear from which features/variants it is instantiated. A fix for certain parts of the product should be optionally propagated to the original source code and/or artefact; vice verse, propagation of artefacts to products should also be possible.

In essence, a product under composition management should be traceable; back to the original source code, design artefacts and additional configuration. When this is possible, change propagation is possible in every direction.

Requirement 11. *The system shall support maintenance (creating, deleting, modifying) of products, composed of features and/or configurations.*

Requirement 12. *The system shall support checkin and checkout of products.*

Requirement 13. *The system shall support registration of artefacts on product level.*

Requirement 14. *The system shall support propagation of changes from compositions to features.*

Requirement 15. *The system shall support propagation of changes from features to compositions.*

Product Customisation Management

Product customisation management (level 4) requires to adapt products outside the domain. This means that these changes should be traceable back to conflicting requirements, design and variation points, if present. It should be possible to "upgrade" a product customisation to a variation point, variant or feature in the system.

Requirement 16. *The system shall support registration of artefacts on customer level.*

Requirement 17. *The system shall support promotion of customisations to product or feature level.*

Requirement 18. *The system shall support integration of changes from lower levels to higher levels.*

5.2 Use-cases

For the use-cases we will make a distinction between server-side, client-side use-cases and general use-cases. The use-cases depicted in the following diagrams are available in textual form in Appendix B.

5.2.1 Server use-cases

The use-cases for the server side are depicted in Figure 5.1. These use-cases should be executed on the server itself.

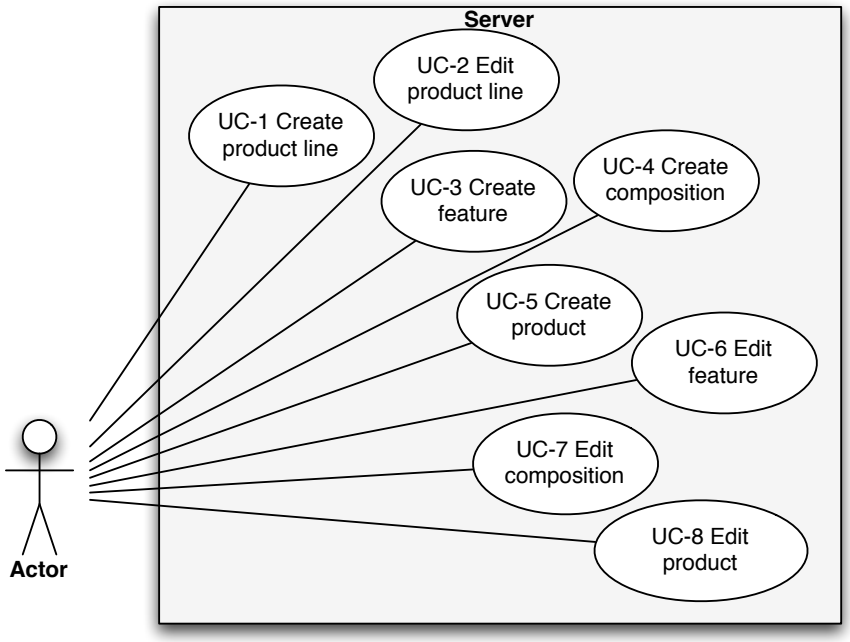

Figure 5.1: Server use-cases.

Creating and Editing Product Line (UC-1 and UC-2)

When a user wants to start with a product line, the following steps are needed. The user initialise a product line by using the server program with specific parameters. After this a directory will be created that contains the necessary layout for the product line. If necessary, the user can create a subversion repository inside the product line hierarchy (in the subversion directory).

After the creation of the subversion repository, the user has to configure the product line, by giving it the necessary parameters for accessing the Subversion repository.

Creating and Editing Features, Compositions and Products (UC-3 to UC-8)

A user can create features, compositions and products. These correspond to respectively layer two, three and four. When creating a feature a parent feature is needed – or no one if the feature is the head-feature of the product line. For a composition a set of features is needed and for a product a set of compositions is needed as input.

When a created feature, composition or artefact, needs to be edited – such as renaming, changing the parent feature – a user can do that by executing the corresponding use-case.

5.2.2 Client use-cases

Figures 5.2 depicts the use-cases for the client. These use-cases describe actions meant to be performed on the client side.

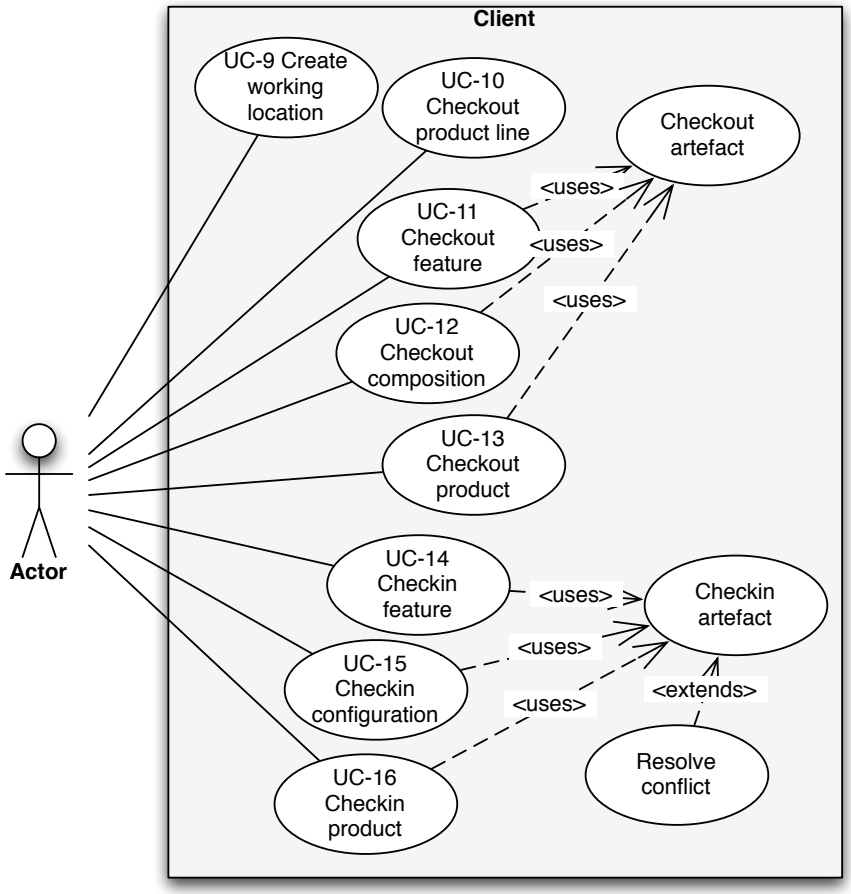

Figure 5.2: Client use-cases

Create working location (UC-9)

Before a user can start working on the artefacts of a product line, he first needs to create a working location. This is somewhat similar to creating a working location for a Revision Control System (RCS), which is often called a working copy. The software client creates a directory containing a basic setup that makes it possible to checkout the product line, features, compositions and products.

Checkout Product Line (UC-10)

When a developer wants to start working on a product line, he has the possibility to checkout the complete product line. This is a convenient way to get a complete overview of all the files in the product line.

Checkout Features, Compositions or Products (UC-11 to UC-13)

With the client it is possible to checkout a specific part of the software product line. There are three options, either to checkout a feature, a configuration or a complete product.

In case of the checkout of a feature, the user will get all the files related to this feature.

When a user requests a composition, he will get all the files of the corresponding features, but also possibly extra files. These will all be available in one directory structure, whereas on the server they could be stored separately.

Lastly, when a user requests a product, he will get all the features and configurations, plus possibly extra files again. This step can be repeated for the customised products for customers.

Checkin Features, Compositions or Products (UC-14 to UC-16)

When done working on a artefact or set of artefacts, the user can checkin made changes. If new artefacts are needed, the user will have to schedule the particular new artefacts for registration first. Thereafter he can checkin the feature, composition or product.

If a conflict arises, the user will get a message describing the conflict, after which the user first will have to mark the conflicts resolved before being able to checkin. In section 6.2.4 we will elaborate more on when conflicts can arise with respect to actions performed on the product line.

5.2.3 General use-cases

Figure 5.3 depicts general use-cases. These use-cases can be implemented for the server or client side. Ideally they should be implemented for both sides.

Propagation (UC-17 to UC-20)

There are several use cases for propagation of artefacts to different levels of the LCM. As said, these correspond to products, compositions and products.

Ideally a user can selectively propagate artefacts from one level to another level, however propagating the complete set of linked artefacts is also satisfactory. If there are unwanted artefacts they can be deleted from the RCS.

When propagating overwrites existing artefacts, there is a conflict. The user should be notified of the conflict and ideally the user should be helped with solving the conflict (requirement 3) by merging the existing and new artefact. If a merge is not feasible then a different solutions is that the user can resolve the conflict himself in the new artefact and overwrite everything with a forced overwrite.

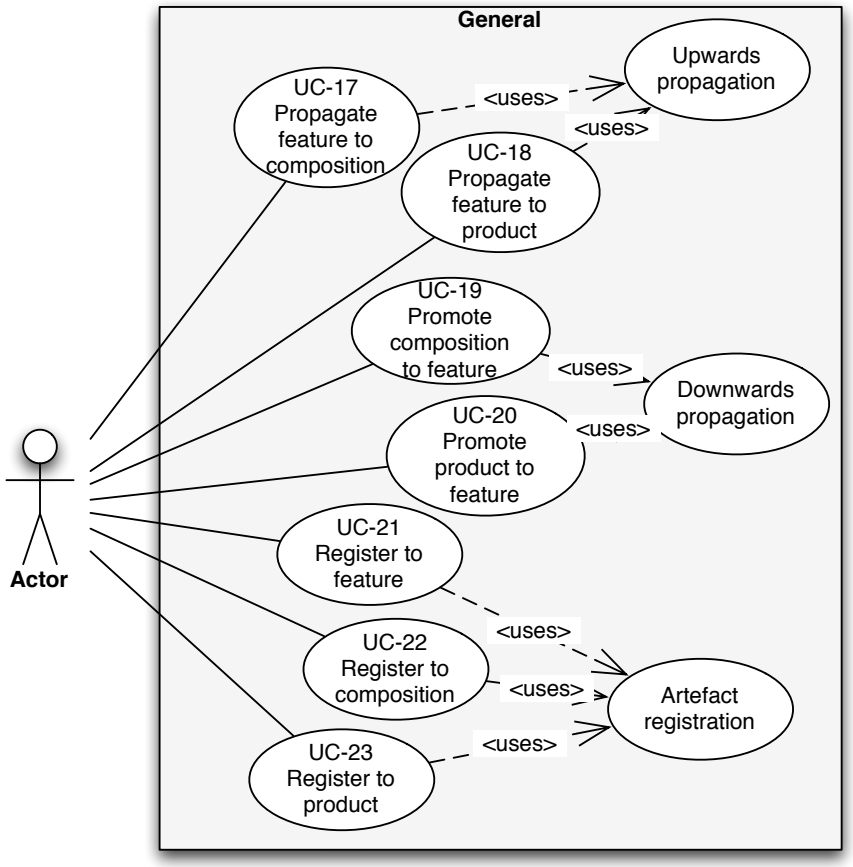

Figure 5.3: General use-cases

Register to Feature, Composition or Product (UC-21 to UC-23)

Artefacts from the RCS need to be registered to a feature, composition or product, otherwise it is not possible to retrieve them from the RCS by selecting a feature, composition or product. The user can register them by performing one of these use-cases.

5.3 Conclusions

In this chapter we introduced requirements for tool support of LCM. Use-cases were added to the requirements. In the use-case descriptions some alternative options for implementation are given for when complete implementation is not feasible. The next chapter will introduce the implementation based on this chapter.

Six

Tool-support

In this chapter we explain the tool-support design, the impact of actions on a product line and the implementation of our tool-support.

6.1 Tool-support Design

This section will describe the design of the prototype we have developed based on the requirements in chapter 5.

The name of our resulting software is *Prolicoment*; an acronym of *Pro*duct *li*ne *Co*nfiguration Manage*ment*.

6.1.1 Architectural Key Decisions

Design

We have chosen to use traceability as a key concept within LCM. All artefacts are stored individually and if there is a relation between artefacts, then a trace is created between those artefacts. In order to be able to create traces, the artefacts should be identified (the first activity). A good choice would be to use a RCS, as it is designed for CM usage, but a database would be a fine choice too for the small artefacts. In Appendix A we have reviewed a number of RCSs that can be used as a basis for CM for the rest of the tool.

When all artefacts are identified, then they can be related by a trace of certain types. While there certainly has be a trace type that relates features to implementation artefacts, the rest of the trace types are dependent on the available time creating them and the usefulness of the traces.

Pre-conditions

The design of the prototype was bound to some pre-conditions on the implementing parts. Firstly, implementation on the Eclipse platform was preferred. In the research area of SE, it is quite common to use Eclipse, so an implementation usable within Eclipse has more opportunity for reuse. Implementation on the Eclipse platform is possible by writing a plug-in that can be loaded into the Eclipse rich-client-platform. This allows the developer of the plug-in to interact with the graphical user interface of Eclipse. As as an alternative,

it is also possible to use a plug-in as the host for the application. We chose this alternative, because it gave us the possibility to create a command-line application, whilst making it possible to extend our software to be used from within the Eclipse graphical user interface. We will elaborate more on this in section 6.3 which is about the implementation of our tool-support.

Secondly, to implement level one functionality we chose to reuse existing CM software for identification of development artefacts. In Appendix A we evaluated some possibilities for this. The result of this evaluation was to use Subversion [31, 4].

Thirdly, as an implication of the second pre-condition, we assumed that a client/server application was requested. In the end this was not really the case, but in our opinion using Subversion for basic CM really makes it necessary to have a central repository, because repository to repository merging is not supported. Having a central repository means that you already need clients for committing source artefacts, therefore using the same client for changing the rest of the product line in a central location sounds logical.

Finally, the Eclipse Modelling Framework (EMF) was requested to use in the solution, because it makes it easy to define models and generate Java code from it.

6.1.2 Server Architecture

Figure 6.1 depicts the high-level design of Prolicoment. We will address every block in the diagram from bottom to top.

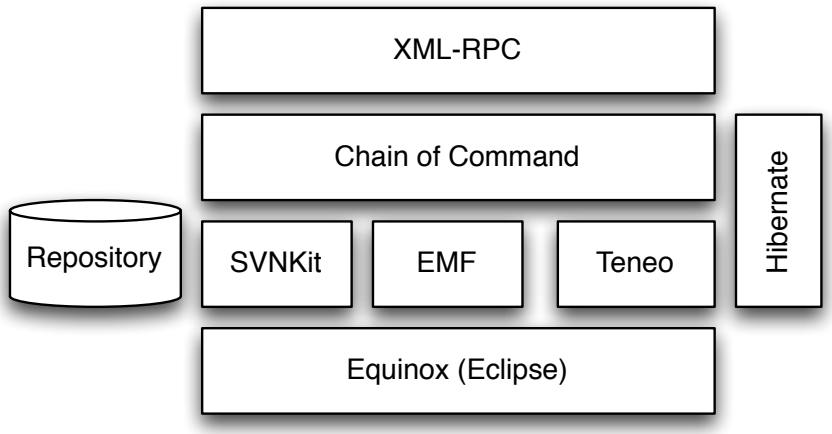

Figure 6.1: High-level server architecture

Equinox (Eclipse) The underlying framework of Eclipse is called Equinox [10]. This will be the basic starting point for our tool. By depending on Equinox and not Eclipse itself, we can create an Eclipse plug-in that can run in an environment without graphical user interface.

SVNKit SNKKit[29] is a Java Subversion implementation, it is completely compatible with the current native version of Subversion. On the server side, we use SVNKit to interact with Subversion repositories directly.

EMF As a pre-condition we use EMF[9] for the meta-model of features, compositions, products and traces. This is modelled in a modelling language called *Ecore* and in the detailed design we will specify details of the meta-model.

Teneo and Hibernate In an previous (not final) iteration of the design of our prototype we had difficulties with saving and loading of EMF resources. To fix this fast we decided to use Teneo[11], which is an automatic mapping of generated EMF classes to a database with the help of Hibernate[15]. Hibernate is an industry standard with respect to Object Relation Mapping; a Java instance – or POJO (Plain Old Java Object) – is mapped to data in a database. This makes it possible to query the database for EMF model instances without loading the necessary files first. It also saves us from writing some look-up strategy for files, because SQL queries can be used to retrieve EMF model instances. As an added advantage we can easily allow users of Prolicoment to write on queries for custom traces or something alike.

Chain of Command Chain of Command is the chain of command pattern used from Apache Commons Chain[2]. We have it prominently in our design, because it will be used to make the prototype extendable with custom traces or other extra functionality users of Prolicoment may need to support their product line. We try to be implementation-agnostic with our design, thereby not imposing any technology onto existing product lines but only a certain work-flow.

XML-RPC In order to communicate between the client and the server apart from Subversion interaction, we have chosen to use remote procedure calls over XML-RPC[3]. This is done by running a small HTTP server handling the requests. Subversion requests can also be done over HTTP, making it possible to use protocol filtering on the firewall or even only opening port 80 (by putting a proxy before Subversion and the Prolicoment server).

6.1.3 Client Architecture

In Figure 6.2 the high-level design of the client is depicted. The client has less components, we will discuss them shortly.

Equinox (Eclipse) The client is based upon Equinox, just as the server. The client is designed as an Eclipse plug-in. This creates some small problems, because Equinox filters a couple of parameters from the command line for own usage, however this is only a minor inconvenience for a big advantage; the client can be reused directly in an Eclipse RCP application or plug-in with a graphical user interface.

SVNKit SVNKit will be used to connect with the subversion repository directly on the server side.

EMF As EMF is used on the server side we want to reuse it on the client side. This is not directly possible as we only want to send parts of the EMF model instances and instances are linked to each other. Our version of EMF is in between two technologies that should allow this (being SDO and CDO). Therefore, in the detailed design we will elaborate on this some more.

XML-RPC We use XML-RPC to communicate with the server. Details about *features*, *compositions*, *product customisations* and other product line artefacts will be transferred over a XML-RPC link.

Chain of Command As with the server, the client will work by chains of command to allow for flexible customisation of the actions the client can perform.

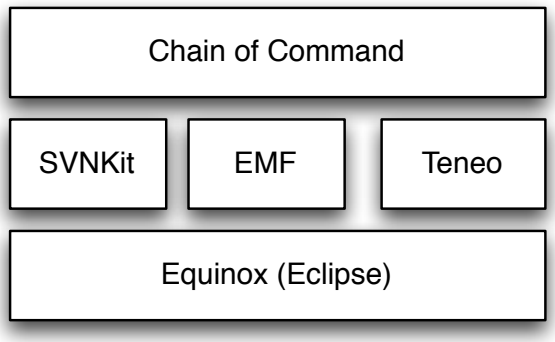

Figure 6.2: High-level client architecture

6.1.4 Detailed Design

In this section we will elaborate our high-level design into a detailed design. We will start with the shared EMF model, continue with the server side and follow-up with the client design.

Eclipse Modelling Framework Model

We have used the EMF to define the meta-model for our tool. Figure 6.3 depicts the part that defines the feature tree.

Feature extends from *Artefact* to support traces between different parts of the model. In Figure 6.4 we display the model of trace.

In the feature tree model we see a link to configurations (which is the old name for *composition*) from features. There is no link to product customisations, because those are linked from the composition / configuration itself. *Require* and *Exclude* are used to define constraints between features and *MandOptType* and *FeatureType* are used to express if a feature is mandatory or optional and if sub features are linked as an *or* choice or an *exclusive or* choice.

We can use the example product line diagram from Figure 2.1 to clarify the model. The *ProductLine* element depicts the complete diagram, with all the features and constraints. In the model, there are three types of features: mandatory/optional, xor and or. In the diagram, we only have the mandatory/optional type. The features *Media*, *Photo*, *Media Management*, and *Create/Delete* have the *MandOpt* attribute set to true. In our model we support two types of constraints: a require and an exclude constraint. In the example product line there are two require constraints and these are added as *Require* instances to the *ProductLine* instance.

Appendix C lists all the models that we used for our prototype.

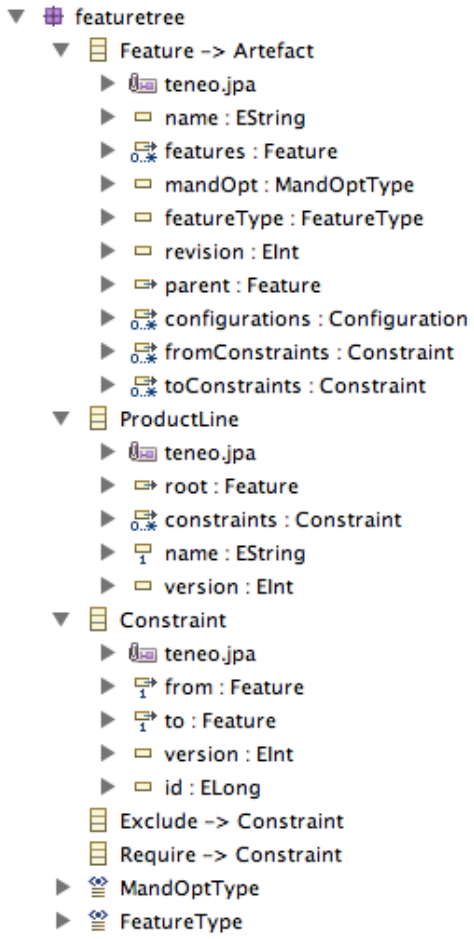

Figure 6.3: EMF feature tree

Detailed Design Server

Figure 6.5 depicts a class diagram with the packages composing the server part of our prototype. We have split the design into three parts: a part for the persistent storage (*db*) which uses the Teneo and EMF packages; an other part

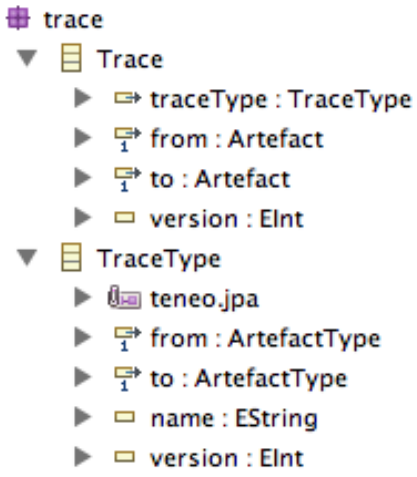

Figure 6.4: EMF trace

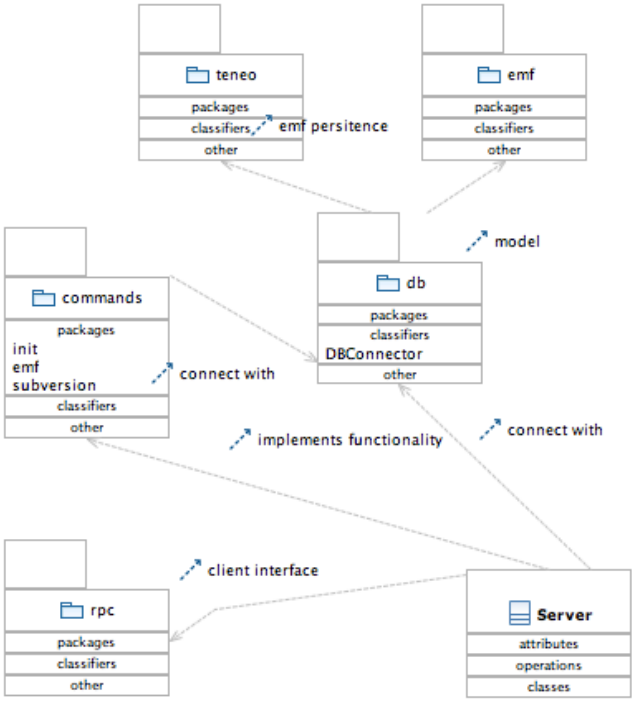

Figure 6.5: Package diagram server

that implements the possibility to work with commands (*commands*); and as third a package to work with the client (*RPC*).

Detailed Design Client

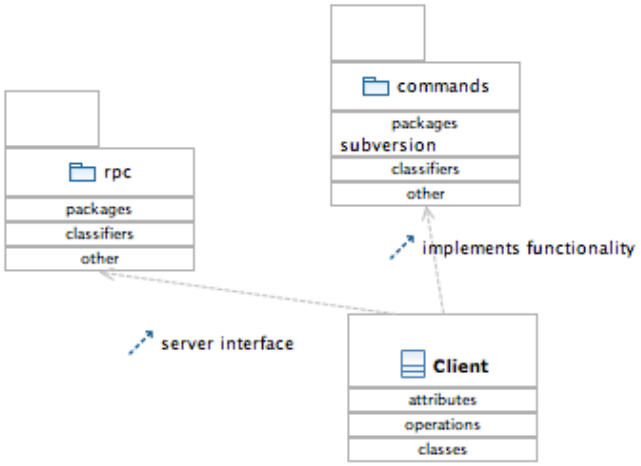

Figure 6.6: Package diagram client

Figure 6.6 depicts the class diagram of the client. It has an *RPC* package for the communication with the client and a *command* package for the commands (for the chain of command pattern). Within the *command* package is an extra package *subversion* for commands that handle subversion interaction.

When we started a detailed design we wanted to include an EMF package, but with the problems listed in architecture we omitted this and decided to send everything with hash-maps. With hindsight, we should have chosen to map the EMF classes on Data Transfer Objects (DTOs), especially because with the current version of EMF it is possible to generate these DTOs automatically – and with specific code templates, they can be generated with the version we use.

6.2 Impact of Changes in Layered CM and Implementation Suggestions

6.2.1 Introduction

In LCM a change in a specific layer can have influence on other layers. Sometimes this influence will be small, sometimes big. Nevertheless there will be an impact. This document will list the possible impact, some informative scenario's and lists algorithms that are needed on model instances of the EMF model designed for LCM.

We will start with a short illustration of the LCM model. We follow up with a list of actions possible on the model and product line represented by the model and the impact in different layers.

We will end with pseudo implementations and algorithms.

6.2.2 Layered Configuration Management

In the proposed LCM there are four levels. They will be mentioned shortly here, for more information please read back in chapter 4.

- Basic Configuration Management. Basic operations of CM are *checkin*, *checkout*, *update* and *delete*. These actions can be implemented by an available VCS, e.g. Subversion.
- Product Line Management. Customisation Management and Feature Management are the two sub-parts of Product Line Management. Feature Management is managing the feature tree of the product line, whereas Customisation Management is managing how to create instances of implementation components for the features. For impact analysis the focus will be on Feature Management mostly.
- Composition Management. Composition Management is managing valid compositions (=configuration) of features. For the impact analysis we assume an off-the-shelf approach, because that approach is more pragmatic from a developer perspective (having certain valid feature combinations opposed to having to maintain every possible combination).
- Product Customisation Management. Product Customisation Management is managing customer specific configurations of compositions in the product line. An example is custom images in the user interface.

6.2.3 Tasks

In LCM the focus is on version control, build management and testing. For an (prototype) implementation we focus on version control and providing building blocks for build management. Testing can be seen as a refinement of version control and build management, because the implementation artefacts of the prototype can be reused.

6.2.4 Actions and Impact

This section will describe actions from LCM, influencing a feature tree and link those actions to levels of LCM on which impact is visible. The actions will be related to the requirements and use-cases in chapter 5 and the levels of LCM.

Version Control

The list of version control actions is based on available options in Subversion. We describe these actions concise, for more information look in the manual of Subversion [4].

- Checkin (Ci). A commit of one (or more) file(s) already available in a repository of files. This updates the files in the repository.
- Checkout (Co). This retrieves a directory or specific files from a repository, creating a local working copy on which changes can be made.
- Add (A). Before files can be committed to a repository, they first have to be explicitly added.

- Delete (D). Removing a file in a working copy only removes it from the working copy, to remove it from the repository, it should be explicitly deleted.
- Update (U). When changes are committed from a working copy they are not automatically available in all working copies. Individual working copies have to be updated by executing an update action. The update synchronises the local files with updates from the repository (possible creating new files in the working copy).
- Export (E). An export is an action that retrieves files from a repository but does not add a relation to the repository; the resulting files are not part of a working copy. This actions is useful when the contents of a repository is needed for distribution.

These actions are performed on level one, however they can influence the levels above as it is possible to do CM actions on artefacts of a higher level. This is displayed in Table 6.1.

Product Line

Actions on the producline are executed on the model instance of a product line, which contains the features, compositions and product customisations.

- Add feature (PA.1). Adding a feature to the product line adds the new feature as a child to a parent feature (or to the product line if it is the first feature).
- Delete feature (PD.1). Deleting a feature from a product line has multiple implications. Firstly, if the feature has sub-features, what has to be done with those? Maybe they have to be deleted or they get the parent of the feature as there parent. Secondly, what has to happen with related (implementation) artefacts. Should they stay or be deleted also? Lastly, if a feature is used in a composition, that composition is invalid from now on, so should it be deleted or changed?
- Change feature (FC.1). If a feature or its relation to the product line is changed, this can have impact on other parts of the product line. If only the name is changed, this is not a problem, but if the type is changed (XOR-container (see section 6.1.4)) then compositions could become invalid. The same is true if a feature gets a constraint or requirement to another feature.
- Propagate feature (PP.1). Propagating a feature is an action that makes artefacts available on a higher level, making it possible to customise the artefacts. This adds a relation between the customised artefacts and its original ancestors.
- Add composition (PA.2). Adding a composition is creating a valid combination of features and possibly linking artefacts to it that are specific for this composition. There is a relation between the composition and composing features.
- Delete composition (FD.2). Deleting a composition can be a resulting action of a feature that is deleted or changed. Alternatively, it is just not

needed anymore. When deleting a composition, there are implications, just as with deleting a feature. Firstly, what to do with related artefact, delete or let them stay? Secondly, what to do with dependent product customisations? Remove the composition from the customisation or delete the complete customisation?

- Change composition (PC.2). Relevant actions that change a composition are the deletion or addition of a feature from / to the composition. The impact on the product line is to the product customisation level. If a feature is deleted, then the feature can stay as a customer customisation, however we would advice to only do that as a temporarily solution as it is against the product line paradigm.
- Propagate composition (PP.2). Propagating a composition makes artefacts available on an other level, allowing for them to be customised and creating a relation between the original and customised artefacts. When propagating the artefacts to a feature, ideally a feature that is also in the composition, this removes the need for the customisation, allowing the artefacts to be deleted.
- Add product customisation (PA.3). By adding product customisations it is possible to relate custom artefacts to a composition and the features of a composition. There is no impact on other parts of the product line when doing this.
- Delete product customisation (PD.3). Deleting a product customisation has no impact on other levels of the product line.
- Change product customisation (PC.3). Adding or removing an extra composition or adding or removing artefacts are relevant actions related to changing a product customisation. None of those action have impact on the rest of the product line.
- Propagate product customisation (PP.3). Customisations can be propagated to a composition or a feature, where the customisations are the ancestors of the artefacts belonging to the feature or composition.

Detailed information on related requirements and use-cases is in Table 6.1, in the *relates* column, numbers prefixed with R are for requirement, numbers prefixed with UC are for use-cases.

Impact

This section displays tables showing the trace and impact relations that exist between actions per level and the levels of the layered configuration management model. We use these tables to define algorithms for the actions.

Every table lists the part of the model to work on: features (level 2), composition (level 3) or product customisation (level 4). The abbreviations are the same as used in the previous section and impact is described using the numbers of the levels of the LCM.

The column *first impact* lists the main impact – the most important one, the column *impact propagation* lists resulting impacts or less important impacts.

	Actions	**Layers impacted**		
	Checkin	**First impact**	**Impact propagation**	**Relates**
Ci.1	Feature	$1 \rightarrow 2$	$2 \rightarrow 3 \rightarrow 1$, $2 \rightarrow 4 \rightarrow 1$	R-1,8 UC-9,14
Ci.2	Composition	$1 \rightarrow 3$	$3 \rightarrow 4 \rightarrow 1$	R-1 UC-9,15
Ci.3	Customisation	$1 \rightarrow 4$	None	R-1 UC-9,16
	Checkout	**First impact**	**Impact propagation**	**Relates**
Co.1	Feature	$1 \rightarrow 2$	None	R-1,8 UC-9,11
Co.2	Composition	$1 \rightarrow 3$	None	R-1 UC-9,12
Co.3	Customisation	$1 \rightarrow 4$	None	R-1 UC-9,13
	Add	**First impact**	**Impact propagation**	**Relates**
A.1	Feature	$1 \rightarrow 2$	$2 \rightarrow 3 \rightarrow 1$, $2 \rightarrow 4 \rightarrow 1$	R-5
A.2	Composition	$1 \rightarrow 3$	$3 \rightarrow 4 \rightarrow 1$	R-13
A.3	Customisation	$1 \rightarrow 4$	None	R-16
	Delete	**First impact**	**Impact propagation**	**Relates**
D.1	Feature	$1 \rightarrow 2$	$2 \rightarrow 3 \rightarrow 1$, $2 \rightarrow 4 \rightarrow 1$	R-5
D.2	Composition	$1 \rightarrow 3$	$3 \rightarrow 4 \rightarrow 1$	R-13
D.3	Customisation	$1 \rightarrow 4$	None	R-16
	Update	**First impact**	**Impact propagation**	**Relates**
U.1	Feature	$1 \rightarrow 2$	None	R-1
U.2	Composition	$1 \rightarrow 3$	None	R-1
U.3	Customisation	$1 \rightarrow 4$	None	R-1
	PL Add	**First impact**	**Impact propagation**	**Relates**
PA.1	Feature	None	None	R-4 UC-1,2,3
PA.2	Composition	None	None	R-11 UC-4
PA.3	Customisation	None	None	R-11 UC-5
	PL Delete	**First impact**	**Impact propagation**	**Relates**
PD.1	Feature	$2 \rightarrow 1$	$2 \rightarrow 3 \rightarrow 1$, $2 \rightarrow 4 \rightarrow 1$	R-4 UC-2,6
PD.2	Composition	$3 \rightarrow 1$	$3 \rightarrow 4 \rightarrow 1$	R-11 UC-7
PD.3	Customisation	$4 \rightarrow 1$	None	R-11 UC-8
	PL Change	**First impact**	**Impact propagation**	**Relates**
PC.1	Feature	$2 \rightarrow 3$,	$3 \rightarrow 4$	R-4,5,8 UC-2,6,21
PC.2	Composition	$3 \rightarrow 2$, $3 \rightarrow 4$	$4 \rightarrow 1$	R-11,13 UC-7,22
PC.3	Customisation	$4 \rightarrow 3$	None	R-11,16 UC-8,23
	PL Propagate	**First impact**	**Impact propagation**	**Relates**
PP.1	Feature	$2 \rightarrow 1$	$2 \rightarrow 3$, $2 \rightarrow 4$	R-15,19 UC-17,18
PP.2	Composition	$3 \rightarrow 1$	$3 \rightarrow 2$, $3 \rightarrow 4$	R-15 UC-19
PP.3	Customisation	$4 \rightarrow 1$	$4 \rightarrow 2$, $4 \rightarrow 3$	R-17 UC-20

Table 6.1: Impact relations

Clarification of Impact Relations Table Let us give a clarification for Table 6.1 on page 43. We will address the concerned artefacts and a couple of impact relations and propagations.

In the Table 6.1, we take two types of artefacts into account. The first is a Subversion repository artefact; a file. The second is a product line artefact, which are features, compositions and customisations.

When we look at the actions *Ci, Co, A, D* and *U*, we see that the artefacts handled are files, so the first impact is from level one to level two. Propagation is possible when artefacts are reused (for a customisation of the artefact on a composition or a product customisation), because when a file changes or is added / deleted, a decision on what to do with the reused files has to be made.

Then, when we look at the actions *PA, PD, PC* and *PP*, the concerned artefacts are product lines artefacts. An impact relation to level one consists of the decission whether or not to execute an action on repository level. This action is one of *Ci, Co, A, D, U*.

The *PA* actions do not have a great impact on other levels. If we add a product line artefact to the product line, then at that moment, there can be no dependency to other parts still. Therefore, the only impact we see is already addressed in the *A* actions.

In the *PP* actions list there is a first impact on level one, this is because artefacts are propagated to another level, which means that files are copied in the Subversion repository. As a result these copied files are linked to the dependent product line artefacts.

6.2.5 Algorithms

In this section we will give list algorithms needed in software to work with Layered Configuration Management.

Version Control Algorithms

The algorithms for version control are designed to reuse original commands implemented by the underlying VCS (in our case Subversion), wrapped around by extra actions that handle product line artefacts and storing trace information (which is implemented as auditing). By wrapping around version control commands, it should be possible to use any version control system as implementation.

Figure 6.7 lists an example algorithm in Java code. It displays what has to be done before and after the wrapped revision control commands. It starts by logging the fact that something has happened between a level one SVN artefact and a product line artefact. Then it executes the original Subversion command. After that we have to decide if we have to handle impact relations and when the impacts will be handled.

Product Line Algorithms

The algorithms for product line artefacts do not need to wrap around version control (Subversion) commands, except for action *PP*, where we need to propagate (or simpler: copy) artefacts from one part to another part of the repository.

```
public void sampleCommand(ProductLineArtefact artefact,
  SVNArtefact svnArtefact, SVNCommand command,
  String... options) {
  // Audit.trace(Object action, String message,
  //  Artefact objects...)
  Trace.trace(command,"Some log message", artefact,
    svnArtefact);
  SVNCommand.execute();
  if (hasImpactPropagation(artefact)) {
    // ask for impact propagation
    ...
    // and handle impact propagation
    ...
    for (a1 :
      impactPropagations(artefact).getArtefacts()) {
      SvnCommand a1SvnCommand =
        getSvnCommand(a1, command);
      sampleCommand(a1, svnArtefact, command, options);
    }
    ...
    // or log that something is needed
    for (a1 :
      impactPropagations(artefact).getArtefacts())
    Trace.futureTrace(svnCommand, "message", artefact,
      svnArtefact, a1);
  }
}
```

Figure 6.7: Version control generic algorithm

```
public void sampleProductLineCommand(Artefact artefact,
  Command command) {
  if (command instanceof PropagateCommand) {
    SVNCommand svnCommand = new SvnCopyCommand();
    // set right artefacts on svnCommand
    svnCommand.execute();
    Trace.trace(svnCommand,"message", artefact);
  }
  Trace.trace(command,"Some log message", artefact);
  command.execute();
}
```

Figure 6.8: Product line generic algorithm

Example

Let us explain Figure 6.7 with an example. We have a user that is working on feature A, which is used in product (configuration) B for a certain customer and a product customisation C is in place because some company specific information was needed in the product.

Now let us assume a *Ci.1* situation (checkin for a feature), where a user executes a checkin after some work on a feature. In Table 6.1 we look up a first impact from level 1 to level 2; the checkin is on level 1 and the checkin relates to a feature on level 2. From level 2 there is (possible) impact propagation to level 3 and 4 and from level 3 and 4 to level 1.

In the example situation there is impact propagation to level 4, depending on the specific files checked-in for feature A. When the checkin is for the customised files we certainly have to check the customisations on level 4 to see if they are still needed or correct and act upon that information.

When we do not have the correct information to take action immediately, then we have to postpone the decision on what to do and we have to log a reminder to take actions somewhere in the future.

As a second example, assume a *D.1* situation (delete files from a feature). When we delete files from the repository there is a first impact from level 1 to level 2. Impact propagation is from level 2 to level 3 and 4 and from level 3 and 4 to level 1. For our example there is only an impact propagation to level 4. If the deleted files are customised then the customised files can stay or be deleted also; it depends on the specific contents of the files what the most logical thing to do is. When the files need to be deleted, a nested *sampleCommand* will do a *D.3* action.

As a last example think of a *PP.3* situation (propagate a product customisation). A user developed some extra functionality for a specific customer that is very useful within the product line; so he wants to propagate that functionality to level 2 of the LCM. The first impact is copying the files in the repository. As an impact propagation, a check has to be done if the files are linked to the feature already; it is possible to link at component / package level or with wild-cards, so there can already be a link available.

Functions

In order to implement all the actions in Table 6.1 we suggest the following list of functions to be available. These functions work on the EMF model that we introduce in section 6.3.

Fun.1. GetCompositionsForFeature. This function will return all the compositions where a given feature is used in.

Fun.2. GetProductCustomisationsForFeature. This function will return all the product customisations where a given feature is used in.

Fun.3. GetProductCustomisationsForComposition. This function will retuen all the product customisations where a given composition is used in.

Fun.4. ValidateComposition. This function will validate a composition of features with the rules as defined in [32].

Fun.5. ValidateProductCustomisation. This function will validate a product customisation, which is a validation of the composing compositions.

Fun.6. GetCustomisationsForComposition. This function will return the artefacts that are used to customise certain features in a composition.

Fun.7. GetCustomisationsForProductCustomisation. This function will return the artefacts that are used to customise certain compositions in a product

customisation.

Fun.8. AddRequireConstraint. This function adds a require or constraint between two features.

Fun.9. RemoveRequireConstraint. This functions removes a require or constraint between two features.

Fun.10. AddActionForArtefact. This function can be used to log a delayed action that should be performed on a artefact of the product line. This occurs when for example a *PD.1* action invalidates a composition and the current user cannot or will not solve the invalidation.

Fun.11. GetActionsForArtefact. This function returns all the actions currently open for a artefact.

In addtions we need extra functions that do not work on product line artefacts, but are trace and CM related.

Fun.12. CreatePatch. This function creates a patch that can be used to propagate changes to customisations on another level. An implementation of this function can use functionality of the used VCS; in our case Subversion.

Fun.13. MergePatch. This function applies a created patch. As with the previous function, this can be implemented using the chosen version control system.

Fun.14. Audit. This is a set of functions which should enable the software to log messages about actions done. This implements a very generic form of tracing as it gives information about the performed actions.

For a couple interesting functions, we will give a sample implementation, in our prototype we implemented them all.

As a small remark, in this thesis we use the term *composition* for a combination of features. When starting our prototype we use the term *configuration*, but after a while we decided that in the text this leaded to confusion and changed it to composition, but in the code examples we still use configuration.

In Figure 6.9, 6.10 and 6.11 we display three functions we think are useful to explain. These functions work on our EMF model as explained in section 6.1.4 in chapter 6.3.

6.3 Tool-support implementation

This section elaborates on the implementation of our tool-support. Apart from implementation details it will also list the implemented and the not implemented requirements.

6.3.1 Implementation overview

We describe our implementation in three steps: first the shared implementation between server and client, followed by a part for the server and a part for the client.

```
public static boolean validateConfiguration(
  Configuration configuration) {
  for (Feature feature : configuration.getFeaturesList()) {
    while (feature.getParent() != null) {
      if (!configuration.getFeaturesList().contains(
          feature.getParent())) {
        return false;
      }
      for (Constraint constraint :
        feature.getFromConstraintsList()) {
        if (constraint instanceof Require) {
          if (!configuration.getFeaturesList().contains(
              constraint.getTo())) {
            return false;
          }
        } else if (constraint instanceof Exclude) {
          if (configuration.getFeaturesList().contains(
              constraint.getTo())) {
            return false;
          }
        }
      }
    }
  }
  return true;
}
```

Figure 6.9: Code sample, validate composition

```
public static void addRequireConstraint(Feature from,
  Feature to, Session session) {
  // session is a hibernate session used to persist
  // the constraint to a database
  for (Constraint constraint :
    from.getFromConstraintsList()) {
    if (constraint instanceof Require
        && constraint.getFrom().equals(from)) {
      return;
    }
  }
  Constraint constraint =
    FeaturetreeFactory.eINSTANCE.createRequire();
  constraint.setFrom(from);
  constraint.setTo(to);
  session.save(constraint);
}
```

Figure 6.10: Code sample, add require constraint

```
public static void addActionForArtefact(Artefact artefact,
  String action, Session session) {
  Action action_ = ActionFactory.eINSTANCE.createAction();
  action_.setAction(action);
  action_.setArtefact(artefact);
  artefact.getActionsList().add(action_);
  session.save(action_);
}
```

Figure 6.11: Code sample, add action for artefact

Shared implementation

In the shared implementation of the tool-support are all the classes that can be reused within the complete tool. These are configuration support, the EMF model and generated code with small adaptations, the basic setup for the chain of command and the interface for the RPC calls between the client and server.

For the chain of command, catalog files are needed, both for the client as the server. In Figure 6.12 we display a sample catalog file. Notice the possibility to create a named chain or a named command. This allows replacing a single command for a chain and vice-verse. A chain or a single command gets executed with a context that gets passed to the command(s) that can modify it and place results in it.

```
<?xml version="1.0" ?>
<catalog name="shared">
  <chain name="sample-chain-command">
    <command className="CommandOne" />
    <command className="CommandTwo" />
  </chain>
  <command name="single-command" className="CommandOne"/>
</catalog>
```

Figure 6.12: Sample catalog file

In the shared implementation is a configuration store that has defaults for all configuration variables. When there is a configuration file it will parse it and overwrites the available defaults. The configuration file is a default Java XML property file.

Server implementation

The server exists out of two big parts: a local part with a text console a user interface and an RPC server, that can be used by a client.

The local part of the server is basically one big loop that parses command from the console interface. Before it is started it initialises the internal database and sets up a connection to the Subversion repository.

Figure 6.13 depicts a sequence of the execution of a single command. The server retrieves the command by name from the catalog, it creates an execu-

tion context and run the retrieved command with it. The context passed to the command always contains the configuration store and a database connector. The rest that is available in the context depends on the specific textual command executed from the console.

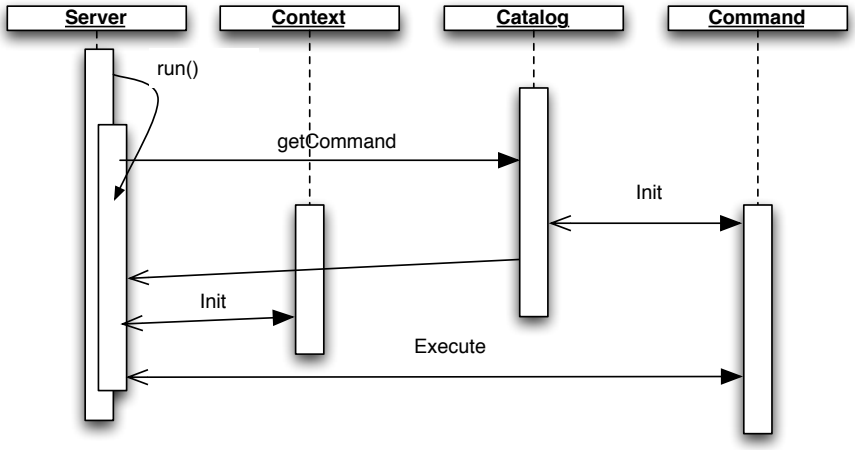

Figure 6.13: Sequence Diagram Server

For the RPC calls, the server implements the methods from the interface. These are also implemented with commands and this works similar to Figure 6.13, but without a run method in our own code, the listening is implemented in Apache XML-RPC.

Client implementation

The client implementation is running on Equinox by executing the Equinox jar-file of Eclipse. This introduces some limitations, as the Equinox framework uses some console parameters for its own input. An example is the *-clean* parameter, which is used for the cleanup of cached Eclipse plug-ins.

Command execution works the same as in the server, only now there is no database connection. The client also connects to the Subversion repository by itself; it only asks the server for the address.

The client has different locations reserved on the file system for the product line, features, compositions and products. They all get checked out to a specific location with their name and type in it: *configurations\ConfigurationA*, *features\FeatureA*.

6.3.2 Implementation completeness

In this section we will address the implementation completeness of our prototype with respect to the requirements of chapter 5.

Basic Configuration Management

With respect to basic CM we have only implemented checkout from requirement 1. We do not support checkin of files and branch management from the tool itself. It is possible to use a Subversion client and perform a checkin or create a branch from the working copy of a feature for example. We cannot really say we support conflict resolving, as we only indicate if there is a conflict. After that the user can force the requested action if he wants, but the user should make sure the result is going to be satisfactory.

Feature Management

In the group of requirements for feature management we implemented almost every requirement, with a few remarks.

We focused mostly on the registration of features and not so much on variation points. It is possible to link a property to a feature, which in turn can be retrieved by the client, that can use that information for a variation point in a linked artefact.

We provide support for optional artefacts in the EMF model, but we did not focus on the specific commands and user interface implementation needed for that. Our focus was on the linking of required artefacts as they are more important for the product line than optional artefacts.

As last, we do not provide the possibility to checkin features from the client itself, this is for the same reasons as mentioned at the basic CM requirements. The workaround for this missing implementation is to checkin the work on a feature and adjust the feature on the server thereafter if necessary. After that, delete the working copy of the feature and do a new checkout when some work has to be done on it later. This work-strategy prevents getting out of sync with the product line.

Customisation Management

For customisation management we kept the implementation simple; the setup of our prototype is very generic, so it can support every variability mechanism. We have implemented a process command for the client that executes a simple copy command and that command could be replaced with a command that performs the required actions for the variability mechanism at hand.

Composition Management

With respect to requirements for composition management we did not implement checkin of artefacts in the client itself, and the work around is the same as with feature management. We implemented propagation (requirement 14 and 15), however when a composition or feature already has an artefact, it will fail with an error as we have not implemented the functionality that can resolve such a conflict in the tool itself. Again, using the Subversion client directly solves this problem.

Product Customisation Management

For product customisation management we only implemented registration of artefact. Propagation is not implemented, however it is very similar to the propagation implemented for composition management. In general, product customisation management is on technical level very similar to composition management, which makes implementation of functionality on this level very easy.

6.3.3 User Guide

A user guide of the tool can be found in Appendix D.

6.4 Conclusions

In this chapter we described the design of the tool-support for LCM. Our design is based on extendability and generic traces. We listed impact of changes in LCM and suggested how to implement the functionality in the design. Finally, we described how we implemented our prototype and summarised which requirements were implemented in the prototype and which were not.

Seven

Evaluation & Conclusions

In this chapter we evaluate our work. We will evaluate our theory, tool-support and compare our work to related work.

7.1 The LCM for SPLs

We think that our theoretical model creates a paradigm that is very useful for product lines. By having a layered approach, it gives more grip on the implementation of the product line. When a developer is working on a composition, he knows that he is working on a specific customisation for a set of features that otherwise would not work together, and when the developer is working on a product, he knows he is making customisations specific for a customer. When a new feature for a customer is becoming more important in the domain of the product line, it is possible to make the feature available in a lower layer and make it part of the base-line of the product line.

Composing a product is as follows: all the composing features will be selected, together with extra artefacts needed for the composing configuration and possible another set of extra artefacts needed for the product itself.

On a downside of our approach; it is hard to write tool-support for a very generic an broad approach. A theoretical model that is focused more on a specific set of product lines would have been easier to implement in a tool.

7.2 Tool

We think that with creating the tool we could have made some decisions that were better for a prototype implementation. To start, a client-server model was too much work; we chose this model because we use Subversion, which has a client-server architecture itself. However, only a client (or server) would have been a much better approach we would not have had problems with sending the EMF instances over the network.

Secondly, it would have been better if we had abstracted more from the underlying RCS. At the moment we are very dependent on Subversion, whereas we should have been dependent on the specific CM functionality Subversion provides.

Thirdly, we think the chain of command pattern is very useful, however we should have created a layer around it that creates more stability and requires less type casting. Currently, Java's compile-time type-safety is completely deleted in our implementation with respect to the commands and boilerplate code to work with the commands.

7.3 Future Work

As a recommendation for future work we would like to see the remaining requirements implemented. In the process of that implementation it would be wise to review the architectural design and see if some refactoring is necessary. As an example we would like an extra abstraction layer around the command-pattern and possibly add some standardisation to the information available for every command, which would make it easier to develop custom commands.

Validation of the tool with a case study would be the next step, to see if the LCM approach works in an industry setting. We would expect that some changes are needed. As a proposition we would like to see how Prolicoment could be applied to OSGi-type applications, which is the underlying machinery of Eclipse plugins. OSGi allows for a plugin-a-like design, but keeping track of all the dependencies between the different OSGi bundles is not easy, at this point LCM and Prolicoment can be used to make dependency handling easier.

As an extension and to add value, an investigation on the possibility to integrate Prolicoment with FeatureMapper and/or a traceability framework would be very useful. Our opinion is that handling of feature trees and trace management can be done with existing tool-support, but there is no link to CM. FeatureMapper makes it possible to map features to EMF/Ecore artefacts, and in Prolicoment we use EMF artefacts to link to Subversion artefacts. This makes us think that it should be possible to integrate these two approaches and we would like to see some work in that direction.

7.4 Conclusions

In this thesis, we have worked toward a novel approach for CM for SPLs. The result is LCM and a prototype called Prolicoment. LCM focuses on layered applicability of CM to SPLE. On level one there is basic support CM, on level two there is CM support for features and customisation of features. Level three provides CM support for compositions of features and level four provides CM support for products. To allow for flexibility, every relation between artefacts is expressed explicitly as a trace. This is visible in Prolicoment itself.

At the moment the user has to switch between multiple tools as he sometimes needs to use Subversion to do certain tasks on the artefacts of the product line. Whether this is a problem or just logical is open for discussion. One could say that LCM is a complete approach and needs a tool providing all the functionality. On the other hand, for basic CM there is already a lot of tool-support (see Appendix A), and there is only extra tool-support needed to be able to apply the LCM principles to an SPL.

Section 6.3.2 illustrates how to work with this partial tool-support in combination with basic CM, as we did not provide complete tool-support, and it proved to be a viable solution.

Concluding, this thesis successfully introduces a novel layered approach to CM for SPL and has results that provide more insight in how to design and implement tool-support for a layered approach.

Appendices

Appendix A

Evaluation of Configuration Management Tools

A.1 Introduction

In this Appendix we will address the selection of a CM tool that we will extend for usage in CM for SPLE. We will start with describing why we need the tool and what we want to achieve by using it. Secondly, we will describe the differences between centralised and distributed CM tools. Next, we will list our requirements for the final tool. Thereafter, a comparison of several tools is given, and finally a couple of tools are compared in more detail, after which a final choice for a tool is made.

A.2 Configuration Management Tools

CM tools are available with different levels of *feature completeness*. There are tools that only support versioning of the information, the so-called VCSs, and the tools that also support (part of) the process around the versioning (e.g. build management, auditing and so on).

We are mostly interested in the basic variant; the VCSs. The reason for this is simple: the open-source and free tools are mostly of this kind. Furthermore, we intend to extend the tool our self, so we do not need all the extra functionally apart from basic version management, but we (probably) need access to the source-code itself.

A.3 Centralised versus Decentralised

Currently, there are two major paradigms for CM or version control in general; centralised and distributed. In this section we will explain the differences as we see it and will evaluate the consequences for us.

A.3.1 Overview

In a centralised solution, there is a client-server model, thus making a server necessary. In a large product line environment, this is probably not a problem;

there will be a server anyway for project files and so on. Additional scripts are only needed on the server, which limits the distribution of code. However, the GUI is still client-side which could make this advantage a disadvantage.

When using a decentralised solution there is not necessarily a central server, however there can be one. When the central server has no functionality (so it only is a storage), then extra care is needed to prevent a corrupt state of the system, because an invalid client could break things. A clear advantage of a decentralised model is the option for everyone to introduce a new branch, with a proper gatekeeper (someone that handles changes to the main branch), this could prevent product line decay. In a centralised model, e.g. Subversion, it is not possible to start an own branch locally, making it harder to (keep) using version management on new branches.

Of course, it is possible to use a (distributed) VCS locally and a centralised VCS on the server. However, effectively this means that you have a peer-to-peer model with one peer being the server, which favours a distributed VCS.

A.3.2 Implementing Extra Functionality

As mentioned in the overview, when using a centralised model, extending the selected VCS is only necessary for the server platform. If the functionality is distributed, then it should be implemented for all the client platforms. Depending on the implementation language, this could mean a lot of work or not so much (e.g. C as an implementation language is probably more work then Python or Java as an implementation language, because the latter two are designed with cross platform compatibility in mind).

A.3.3 Repository Contents

With a centralised VCS it is often possible to have multiple projects and/or branches in the same repository. A decentralised solution often only gives you the possibility to have a single branch in a repository. This means effectively that creating a product from a decentralised VCS means checking out numerous different repositories, whilst it is only one action when using a centralised VCS.

On the other hand, this is probably only the case when checking out the default product, otherwise it would mean checking out specific folders multiple times. Consequently, when having some kind of wrapper tool in place (which we are going to do anyway), both solutions would be equivalent.

A.4 Requirements

This section will list the VCS requirements for our CM tool.

A.4.1 Binary Files

Design artefacts could be in a (proprietary) binary format. Therefore, the selected system should support binary files in a transparent way. We do not expect the files that get inserted into the system to be excessively big, probably below 25 Mega bytes.

Centralised version control is in an advantage here, because that limits the amount of data that needs to be stored on the client. A small change to the

contents of a binary file could mean that the patch from the original version to the next version is very big, increasing the storage requirements for the client in a distributed setting.

A.4.2 Branching

Working within an SPL means working with variants (a variant is a certain feature that provides a variable feature, e.g. implementing authentication with a pin code or biometrics). Variants can be implemented in different ways, namely: branching your variants and complete products, using code structure (e.g. using the C preprocessor for C-code), or duplication of code in a branch.

From the above three options, two use branches. The second option has a somewhat primitive usage of branches, but nevertheless uses them. We think that using branches in some way is a good way to have CM of variants that is not dependent on the tools available for the selected programming language. Therefore, we feel that branching is a key concept for our solution — the selected CM tool should support easy branching and importing branches into a code base to prevent code duplication as much as possible, while allowing for easy implementation of variants.

We assume that every VCS is able to handle branches, but there are probably different functionality levels. E.g. Subversion (one of the evaluated VCSs) can copy a file with its history to another branch, something that certain other systems cannot do (they can only move a file with its history). Another example is the feature of Subversion to import a repository into an other repository; that is useful for selecting variants (however this is dependent on the specific implementation of the SPL).

As branching is an important feature for SCM in combination with software product lines, it is important that handling branches is not only available but easily usable. An important counterpart of branching is merging — this should be intuitive and easy, preferably the VCS should now if a certain merge from a branch is already done.

A.4.3 Change Propagation

Our focus is on extending the selected tool with traceability information. This means that based on the trace information, some other variant, product or something else should be notified. To implement this, ideally, there would be a possibility to link directories within a repository to each other or repositories to each other. In the less ideal case, we should have the possibility to know if certain changes are already present in a repository or directory of the repository (so this is related to the merging part of the subsection about branching).

A.4.4 Extendability

As said before, it is relatively certain that we will have to extend the selected tool in some way to address our needs. If the tool provides means to insert functionality before or after the execution of a certain command, this would mean that there is a good change we do not have to change the source code of the tool. This, of course, has our preference, because this will mean that we

will remain compatible with the tool itself, making it easier and less work to maintain our code-base.

A.4.5 Implementation Language

It is assumable that we have to extend the existing tool with some functionality, because want to save trace information and probably additional files that are separately stored from the "normal" repository. If the tools extend points are not sufficient, we will have to adapt the tool itself.

A.4.6 Licensing

As said in the previous section; our funds are limited. However, we are allowed to open source our code, and therefore, it is not a problem if the license of the CM tool obliges us to open-source the resulting prototype tool.

A.4.7 Multi-platform

The selected tool should be usable Microsoft Windows and *nix/Linux. These are the operating systems available within the Computer Science department. It also adds nicely to the cross platform property of Eclipse.

A.4.8 Performance

The performance of the selected tool should be sufficient to handle big systems; it should be production ready. Without proper benchmarking or even case studies it is hard to do this scientifically, so we have a guideline here: if it really easy to find remarks (on the Internet, from other people) about the being slow, then it probably is. Also, if enough people and businesses use the tool it is probably good enough and fast enough.

A.4.9 Usable with Eclipse

Within the SE department of the University of Twente, the main development of prototypes is done in Eclipse; because it is easy to write a plug-in for it that does the things you want. Therefore, for our project we will be using Eclipse to implement our prototype. It would be helpful if the CM tool already has an existing plug-in for in Eclipse, as this would save us the work of implementing it our self.

A.4.10 Compatibility with project SCM

As this thesis is executed within the context of a bigger project, we should take this into account also. Withing this project, Subversion is already used, so we need to take this into account for our choice.

A.5 Evaluation

There are a lot of VCSs available [34]. However, a lot of them are proprietary software and, therefore, are not an option for us. Using the information of

[5, 33], we could get an general overview of our options. Especially the license information helped us.

A.5.1 Overview

Table A.1 shows an overview of different VCSs with respect to our requirements. When minus and plus is used, this means we used some relative order between the different tools; plus means that a certain requirement is satisfied in a positive way, whereas minus means that a requirement is not satisfied or very poorly with respect to the other tools. A *D* between brackets behind the name means that it is a distributed VCS.

A.5.2 Elaboration on the Tools

Bazaar

Bazaar-NG (it is a reimplementation of an earlier tool called Bazaar) is a distributed VCS that is used by the Ubuntu Linux distribution for its CM. It is written fully in Python (which makes it the most cross platform, but the performance is somewhat worse than the other tools) and a plug-in for Eclipse is available.

CVS

As the table shows: CVS is worse than every other tool. We knew that already, but for reference purpose it is in the table. It is not a serious contender however.

Git

Git is a distributed RCS that is used to maintain the Linux kernel. It is free and open-source and is POSIX compliant, meaning that is will work on Windows, but only when using Cygwin; which is not ideal. There are efforts to make it run on Window without Cygwin, but then there still are requirements to the file system used (Git uses symbolic and hard links), which made us decide that Git is not a very good cross platform option at the moment.

If it would run on Windows it would be our best option, because it is very fast and it consists of separate tools that are easy to extend and combine in various ways.

Darcs

Darcs is a distributed RCS written in Haskell, which is seen as one of a kind within the CM tools, because of its unique ability to *cherry pick* patches (which is choosing only one change from a repository) and a very good merging algorithm. However, there are no GUIs available for it (so no Eclipse support) and the performance is really bad when the repository gets somewhat bigger in size.

	Binary files	Branching	Change propagation	Extendability	Language	Licencing	Multi-platform	Performance	Eclipse	Project SCM
Bazaar (D)	+	++	+	+	Python	+	+++	-	+	-
CVS	-	-	-	+	C	+	++	+	-	-
Git (D)	+	+++	+	+	C, Perl	+	+	+++	-	-
Darcs (D)	+	+++	+	+	Haskell	+	+	-	-	-
Mercurial (D)	+	++	+	+	Python, C	+	++	++	+	-
Monotone	+	+++	+	+	C++	+	++	-	-	-
Perforce	++	++	+	-	N/A	+/-	+++	++	+	-
Subversion	++	+	+/-	+	C	+	+++	+	++	+
SVK (D)	+	+	-	+	Perl	+	++	+	-	-

Table A.1: Overview of numerous VCSs with respect to our requirements.

Mercurial

Mercurial is a distributed VCS written in Python, and partially in C for performance reasons, after the authors discontent with the Git tool. It is used by a lot of projects (e.g. Mozilla, NetBeans) and a plug-in for Eclipse is available.

Mercurial has the possibility to bundle patches, which can be sent to some maintainer, this can be a handy for change propagation. Also, there is the possibility to export and import changes, without synchronising two whole repositories

Perforce

Perforce is a commercial tool that is free to use non-commercially; this makes it an option for us. Perforce is more than only a RCS system, it is a full software CM tool.

The GUI of Perforce has a lot of features with respect to branching and so on, which is important for us. There is also a plug-in for Eclipse available, however it has not all the features of the normal client (and there is no source code available for the plug-in).

There is an API available to use the core features of the command line tools of Perforce, making it less of a problem that the source-code is not available.

Perforce is a client-server solution with some caching possibilities client-side, but it is still a full client-server solution; no connection to the server means no possibility to edit files.

There is one thing making Perforce stand out of the rest, namely the possibility to do change propagation out of the box to other branches. This is integrated in the software, meaning that there is already some tracing going on for this. Of course, you can get this functionality in any other tool, simply by having some additional process in place — a tracing process that knows which are the related branches.

Monotone

Monotone is a distributed VCS that focuses on a distributed work-flow; it is possible to comment on the quality of a certain revision and there is native support for public / private key pairs to collaborate more easily (even on sub parts of a repository). It has to be seen how useful the second part is within a company, where probably other (authentication) policies are already in place. Performance issues made us decide that is not a viable choice for us.

Monotone was a partial inspiration for numerous other tools, namely Bazaar, Git and Mercurial. They all adopted the cryptographic hash (SHA-1) mechanism for revision identification.

Subversion

Subversion is probably the best known centralised VCS to date. It is designed as an improvement over CVS and runs on both Windows and Linux. It handles branching very well, however merging a branch is somewhat painful at times; it requires you to manually keep track of merges that already have been done, otherwise it could be possible to merge a change twice, which could be erro-

neous. The latest version of Subversion tries to solve this however, but that is not a production candidate yet.

There is a Eclipse plug-in available for Subversion, which is stable and works fine. It is possible to extend functionality by adding scripts to the repository, without the need to update something client-side.

Subversion is written in C-code, but extensions can be written in any language that the server understands.

SVK

SVK is an add-on for Subversion that allows users to have distributed version control in combination with an Subversion server. We do not think it is widespread and there is no tool support other than the command line.

A.5.3 Selection

For the selection of a single tool, we have used the input of Table A.1 for Analytic Hierarchy Process, which is a well know multi-criteria decision analysis method. To calculate everything for us, we used an evaluation version of RightChoice [30]. Unfortunately, the evaluation version allowed only for six alternatives to be compared. Therefore, we dropped Darcs, Monotone and Perforce as alternatives; Darcs, because it uses Haskell, the only functional programming language in the list of used languages; Monotone, because the performance is not very good and it is written in C++ and finally Perforce, because it is not possible to extend the functionality.

The relative importance of the requirements is in A.5.3. As shown, a match with the projects chosen SCM is very important to us.

The results of the complete analysis are depicted in Figure A.1. The number one result of the analysis is Subversion, which will be the SCM that we within our own tool-support.

Requirement	**Priority**
Project SCM	27.81%
Eclipse	13.17%
Change propagation	13.08%
Multi-platform	12.27%
Central or distributed	9.01%
Language	6.84%
Branching	6.32%
Extendability	5.77%
Performance	3.16%
Licencing	2.57%

Table A.2: Relative ordering of requirements

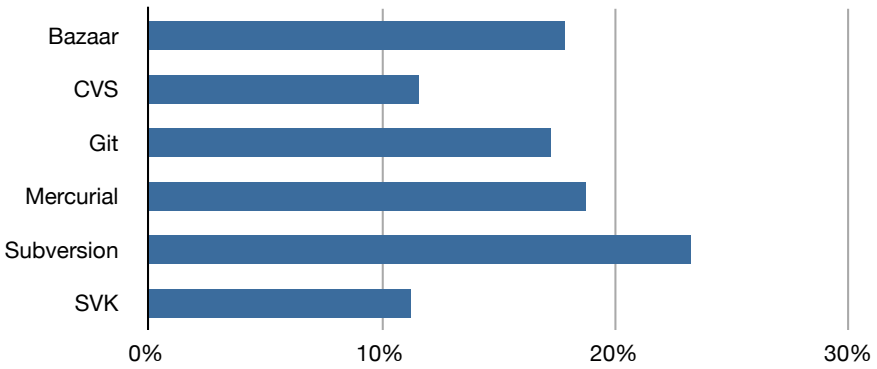

Figure A.1: Results of SCM tool selection

Appendix B

Use-cases

This Appendix lists the use cases for chapter 5.

Use Case: UC-1	Create product line
Related Use-Cases:	none
Actor Actions 1. Type *create-product-line <name>*	**System Responses** 2. Responds with message that product line is created

Table B.1: Use-case create product line

Use Case: UC-2a	Update product line, add product line constraints
Related Use-Cases:	Update product line, delete constraint
Actor Actions 1. Type *update-product-line add-constraint <fromFeature> <toFeature> <exclude|require> ...*	**System Responses** 2. Responds with message that constraint is created.

Table B.2: Use-case add product line constraints

Use Case: UC-2b	Edit product line, delete product line constraints
Related Use-Cases:	none
Actor Actions 1. Type *update-product-line delete-constraint <constraintId> <exclude|require> ...*	**System Responses** 2. Responds with message that constraint is deleted.

Table B.3: Use-case delete product line constraints

Use Case: UC-3	Create feature
Related Use-Cases:	none
Actor Actions 1. Type *create-feature <name> [parent-name]*	**System Responses** 2. Responds with message that feature is created.

Table B.4: Use-case create feature

Use Case: UC-4	Create composition
Related Use-Cases:	Create feature
Actor Actions 1. Type *create-configuration <name> <feature> <feature> ...*	**System Responses** 2. Responds that composition is created, optional listing features that did not exist.

Table B.5: Use-case create composition

Use Case: UC-5	Create product
Related Use-Cases:	Create composition
Actor Actions 1. Type *create-product <name> <configuration> [configuration] ...*	**System Responses** 2. Responds that product is created, optional listing compositions that did not exist.

Table B.6: Use-case create product

Use Case: UC-6a	Edit feature, rename feature
Related Use-Cases:	none
Actor Actions 1. Type *update-feature <featureName> rename <newFeatureName <exclude|require> ...*	**System Responses** 2. Responds with message displaying old and new feature name.

Table B.7: Use-case rename feature

Use Case: UC-6b	Edit feature, update type
Related Use-Cases:	none
Actor Actions 1. Type *update-feature <featureName> <set-mandopt-type|set-or|set-xor|set-mandatory|set-optional>*	**System Responses** 2. Responds with message that action succeeded.

Table B.8: Use-case update type

Use Case: UC-7a	Edit composition, rename
Related Use-Cases:	none
Actor Actions 1. Type *update-configuration <configurationName> rename <newConfigurationName>*	**System Responses** 2. Responds that configuration is renamed to new name.

Table B.9: Use-case rename composition

Use Case: UC-7b	Edit composition, update features
Related Use-Cases:	none
Actor Actions 1. Type *update-configuration <configurationName> <remove-feature|add-feature> <feature-Name>*	**System Responses** 2. Responds that configuration has feature removed or added.

Table B.10: Use-case update composition features

Use Case: UC-8a	Edit product, update name
Related Use-Cases:	none
Actor Actions 1. Type *update-product <productName> rename <newProduct-Name>*	**System Responses** 2. Responds that product is renamed to new name.

Table B.11: Use-case rename product

Use Case: UC-8b	Edit product, update configurations
Related Use-Cases:	none
Actor Actions 1. Type *update-configuration <configurationName> <remove-configur|add-feature> <feature-Name> <newConfigurationName>*	**System Responses** 2. Responds that product has configuration removed or added.

Table B.12: Use-case update product configurations

Use Case: UC-9	Create working location
Related Use-Cases:	none
Actor Actions 1. Type *client checkout <serverUrl>* in a empty directory	**System Responses** 2. Creates a working directory containing the necessary files to work with.

Table B.13: Use-case create working location

Use Case: UC-10	Checkout product line
Related Use-Cases:	none
Actor Actions 1. Type *client checkout productline* in a sub directory of the working directory	**System Responses** 2. Creates a checkout in working directory sub folder *checkout/productline.*

Table B.14: Use-case checkout product line

Use Case: UC-11,12,13	Checkout
Related Use-Cases:	none
Actor Actions 1. Type *client checkout <feature|configuration|product> <name>* in a sub directory of the working directory	**System Responses** 2. Creates a checkout in working directory sub folder *checkout/<feature | configuration | product>/<name>*.

Table B.15: Use-case checkout

Use Case: UC-14,15,16	Checkin
Related Use-Cases:	none
Actor Actions 1. Type *client checkin <feature|configuration|product>* in a sub directory of the working directory	**System Responses** 2. Responds with message that checkin succeeded

Table B.16: Use-case checkin

Use Case: UC-17,18	Propagate feature
Related Use-Cases:	none
Actor Actions 1. Type *propagate-feature <featureName> <configuration|product> <name>*	**System Responses** 2. Responds with message if propagation succeeded. If succeeded the traces and artefacts of the feature are available for the composition or product.

Table B.17: Use-case propagate feature

Use Case: UC-19	Propagate composition
Related Use-Cases:	none
Actor Actions 1. Type *propagate-composition <compositionName> feature <featureName>*	**System Responses** 2. Responds with message if propagation succeeded.

Table B.18: Use-case propagate composition

Use Case: UC-20	Propagate product
Related Use-Cases:	none
Actor Actions 1. Type *propagate-product <productName> feature <featureName>*	**System Responses** 2. Responds with message if propagation succeeded.

Table B.19: Use-case propagate composition

Use Case: UC-21,22,23	Register
Related Use-Cases:	none
Actor Actions 1. Type *register <feature | configuration | product> <name> <HEAD|revision> <patterns>*	**System Responses** 2. Responds with message for every pattern if it can be found and if it is registered to the feature or configuration or product.

Table B.20: Use-case propagate composition

Use Case: UC-24	Resolve conflict
Related Use-Cases:	Checkin
Actor Actions 1. Type *client checkin <feature|configuration|product> <name>* in a sub directory of the working directory 3. Resolve conflict and checkin again with *–force* as argument	**System Responses** 2. Responds with message checkin has a conflict. 4. Responds with message that checkin succeeded.

Table B.21: Use-case resolve conflict

Use Case: UC-25	Delete trace
Related Use-Cases:	none
Actor Actions 1. Type *delete-trace <traceId>*	**System Responses** 2. Responds with message that trace is deleted if found.

Table B.22: Use-case delete trace

Appendix C

EMF models

This appendix lists all the emf models as used in our prototype for Layered Configuration Management. To have a complete list, also look at Figure 6.3.

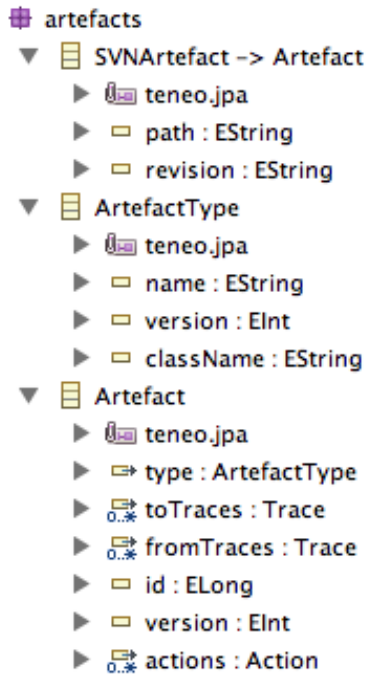

Figure C.1: EMF artefact

Figure C.1 depicts the EMF model of the artefacts package. There is a parent model artefact that can be used as a base class for models implementing artefacts. An artefact is linked to an artefacttype, which is used to identify the artefacts. As we store the model instances in a database, there is a svnartefact, used to be able to store representations of Subversion artefacts.

Figure C.2 depicts the configuration package, which contains the models used for storing configurations and products. These models are simple, they

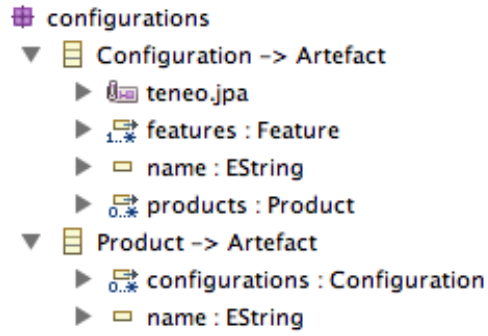

Figure C.2: EMF configuration

have a name and can store features and configurations.

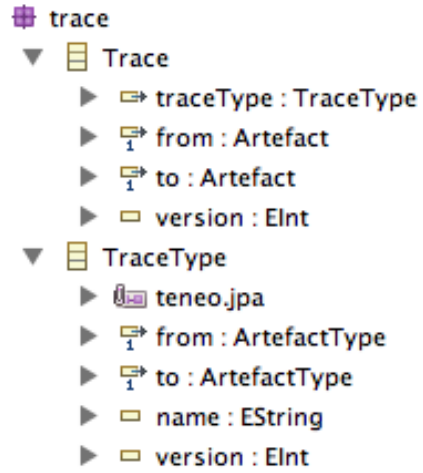

Figure C.3: EMF trace

Figure C.3 depicts the package containing the trace model. As with the artefacts, there is a model for traces and a model for tracetypes. The trace model works with the the artefact model, as it can only link artefacts to each other.

Appendix D

User Guide Prolicoment

D.1 Installation

Installing Prolicoment is just unpacking a zip-file [1]. However, there is a pre-requisite of Java. Any recent Java run-time [2], for example version 1.6.x will do, please make sure that the java binary is on the executable path of your operating system.

To run the server (from Eclipse) the easiest way to a correct configuration is to download following Eclipse Ganymede SR2 packages [3]: Eclipse IDE for Java EE Developers and Eclipse Modeling Tools. Copy the plugin and features from one of the two to the othere installation and after that install Subversive 1.6.x with all the optional packages (SVNKit, JavaHL)[4]. The resulting Eclipse installation also allows to develop on the Prolicoment client and server.

D.2 Running the Server

To run the server, start up Eclipse and open the ProlicomentJ.shared and ProlicomentJ.server project. Create a directory somewhere on your file system and place a *.prolicoment* file in it. Start *nl.k3n.prolicoment.server.Server* as a normal Java application with a working directory set to the directory just created. From now on the chosen directory will be called the working directory.

Starting the server will create a new directory in the working directory called *config* with in it a db directory. This db directory contains the Derby database used by the server. To initialise the server with basic trace types, please type in the command *_init-initial-trace-types*.

Now it is possible to create the product line to work on. This is done by typing the command *create-product-line ProductLineName*. It is only possible to have one product-line, so if this command is executed twice, then it will give an error.

The server also needs a Subversion repository. The easiest way to create one is to type in the command *create-repository*, which will create a directory

[1] `http://www.k3n.nl/Prolicoment.zip`
[2] `http://java.sun.com`
[3] `http://www.eclipse.org/downloads/packages/release/ganymede/sr2`
[4] `http://subclipse.tigris.org`

config/subversion in the working directory containing an empty Subversion repository. The server will automatically use this Subversion repository.

D.3 Running the Client

To run the client, unpack the zip file containing the client and create an environment variable *PROLICOMENT_CLIENT* containing the location. To work with the client, put *client.bat* in the root of the unpack location on the system path.

For a first checkout from the server, type *client checkout http://<serverurl>:8080* in an empty directory. After this you can issue all the commands from the use-cases in Appendix B and some more. We address all commands in the next section which contains an example usage scenario of the tool.

D.4 Example Usage

This example will be oriented around Figure 2.1. We will start by working on the server-side and configuring everything and follow by using that for the client-side.

D.4.1 Adding Features to Server

To add features to the server, first create a product line with *create-product-line MobileMedia*. After that, start adding the features:

- create-feature MobileMedia
- create-feature Media MobileMedia
- create-feature Photo Media
- ...
- create-feature CreateVideo CreateDelete

After this setup set the necessary types for every feature:

- update-feature MobileMedia set-mandopt-type
- update-feature Media set-mandatory
- update-feature Media set-mandopt-type
- ...
- update-feature CreateVideo set-optional

As last we need to add the requires constraints:

- update-product-line add-constraint CreateMusic requires Music
- update-product-line add-constraint CreateVideo requires Video

Now we can check if everything is all right by querying for some information:

- query from Feature
- query from Constraint

The result of these two commands is a list with features and constraints.

D.4.2 Adding Subversion Artefacts to Features

To add Subversion artefacts to features, there must be artefacts in Subversion first. This can be done by checking out from the client by running *client checkout productline* and add files in *checkout/productline/HEAD* and checking them in, but also by working directly on the Subversion repository on the server, which is in the *config/subversion* directory.

After this we can register artefacts to the features:

- register feature Media HEAD /trunk/path/to/artefacts/
- register feature Photo HEAD /trunk/path/to/artefacts/*.java

In the second bullet point we see the option to only add certain artefacts by wild card.

Maybe we need to delete certain artefacts, we can do this by deleting the relation between the Feature and the Subversion artefact; the trace. Query for the trace and find the id and delete the specific trace:

- query from Trace
- delete-trace id

D.4.3 Checkout Features on Client

To checkout features on the client, type *client checkout feature <name>*. To get the list of available features, type *client query from Feature*. The checked out feature is in *checkout/feature/<name>*. For every revision used from the Subversion repository there is a directory, with HEAD for the head revision of the repository.

D.4.4 Add Configurations to Server

Compositions can be added by listing the features that should be in the composition. In Prolicoment we call compositions configurations:

- create-configuration ConfigurationOne MobileMedia Media Favourites MediaManagement CreateDelete
- update-configuration ConfigurationOne add-feature Photo
- update-configuration ConfigurationOne remove-feature Favourites

It is possible to add invalid configurations, in fact the first bullit point will create an invalid configuration if we look at the sample product line in Figure 2.1. To check if a configuration is valid, issue this command: *validate-configuration ConfigurationOne*. This will list if a configuration is valid and if not, it will list what is not correct.

Just as with features we can add Subversion artefacts to a configurations. If we add files of the configuration to the Subversion repository at */configuration/<configurationName>* then we can copy overwrite files from features from a configuration automatically on the client.

D.4.5 Checkout Configurations on Client

As with features, we can checkout configurations on the client by typing *client checkout configuration ConfigurationOne*. This will checkout specific files for the configuration in addition to all the files of the features. Now, if we need to combine the artefacts of the features and the configuration, we can do the following: *client process configuration ConfigurationOne*. This will copy all the checked-out artefacts for the features first and after that it will copy the artefacts of the configuration. The results are in *parsed/configuration/ConfigurationOne*.

D.4.6 Working with Products

Working with products is the same as with configurations; please interchange *configuration* with *product* in the above commands.

D.4.7 Propagate artefacts

In LCM there is downward and upward propagation, but in Prolicoment we only have implemented upward propagation. This means that we can copy the artefacts of a feature to a configuration with *propagate-feature Photo configuration ConfigurationOne*. This will copy the registered artefacts to *configuration/-ConfigurationOne/* and will register them to the configuration.

D.4.8 Working with Updates

In Prolicoment we do not have very good support for updating features, configurations and products with new artefacts. What we recommend is to work on something and then do a checkin. If new artefacts are added to Subversion add them to the server also and after that delete the checked out data. When it is needed to work on something again, this can be done by checking out the needed feature, configuration or product again.

References

[1] James W. Moore Alain Abran, editor. *SWEBOK: Guide to the Software Engineering Body of Knowledge*. IEEE Computer Society, 2004.

[2] Apache.org. Commons chain. `http://commons.apache.org/chain/`, 2008.

[3] Apache.org. Apache xml-rpc. `http://ws.apache.org/xmlrpc/`, 2009.

[4] C. Micheal Pilato Ben Collins-Sussman, Brian W. Fitzpatrick. Version control with subversion, 2008.

[5] Berlios. Better SCM Initiative: Comparison. `http://better-scm.berlios.de/comparison/comparison.html`, March 2008.

[6] Günter Böckle, Jesús Muñoz, Peter Knauber, Charles Krueger, Do, Frank van der Linden, Linda Northrop, Michael Stark, and David Weiss. Adopting and institutionalizing a product line culture. *Software Product Lines*, pages 1–8, 2002.

[7] N. Chapin, J.E. Hale, K.M. Khan, J.F. Ramil, and W.G. Tan. Types of software evolution and software maintenance. In *Journal of Software Maintenance and Evolution: Research and Practice*, volume 13, pages 3–30. Wiley, 2001.

[8] Susan Dart. Concepts in configuration management systems. In *Proceedings of the 3rd international workshop on Software configuration management*, pages 1–18, New York, NY, USA, 1991. ACM.

[9] Eclipse.org. Eclipse modeling framework project (emf). `http://www.eclipse.org/modeling/emf/`, 2009.

[10] Eclipse.org. Equinox project. `http://www.eclipse.org/equinox/`, 2009.

[11] Elver.org. Emf hibernate. `http://www.elver.org/hibernate/index.html`, 2008.

[12] Ralph Johnson John Vlissides Erich Gamma, Richard Helm. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 2007.

[13] Jilles Van Gurp, Jan Bosch, and Mikael Svahnberg. On the notion of variability in software product lines. *Working IEEE/IFIP Conference on Software Architecture*, pages 45–54, 2001.

[14] Florian Heidenreich, Ilie Şavga, and Christian Wende. On controlled visualisations in software product line engineering. In *Proceedings of the 2nd International Workshop on Visualisation in Software Product Line Engineering (ViSPLE 2008), collocated with the 12th International Software Product Line Conference (SPLC 2008)*, September 2008. To appear.

[15] Hibernate.org. Hibernate. community documentation. `http://docs.jboss.org/hibernate/stable/core/reference/en/html/`, 2008.

[16] IEEE. *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*. ISO, New York, NY, 1990.

[17] IEEE. International standard - iso/iec 14764 ieee std 14764-2006. *ISO/IEC 14764:2006 (E) IEEE Std 14764-2006 Revision of IEEE Std 1219-1998)*, pages 1–46, 2006.

[18] W. Jirapanthong and A. Zisman. Supporting product line development through traceability. *Software Engineering Conference, 2005. APSEC '05. 12th Asia-Pacific*, pages 506 – 514, 15-17 Dec. 2005.

[19] K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak, and A.S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical report, Software Engineering Institute, Carnegie Mellon University, 1990.

[20] Charles W. Krueger. Variation management for software production lines. In *SPLC 2: Proceedings of the Second International Conference on Software Product Lines*, pages 37–48, London, UK, 2002. Springer-Verlag.

[21] K. Lee, K.C. Kang, W. Chae, and B.W. Choi. Feature-based approach to object-oriented engineering of applications for reuse. *Software Practices and Experiences*, pages 1025 – 1046, 2000.

[22] K. Mohan and B. Ramesh. Managing variability with traceability in product and service families. In *System Sciences, 2002. HICSS. Proceedings of the 35th Annual Hawaii International Conference on*, pages 1309–1317, 2002.

[23] K. Mohan and B. Ramesh. Change management patterns in software product lines. *Commun. ACM*, 49(12):68–72, 2006.

[24] K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering*. Springer, 2005.

[25] Balasubramaniam Ramesh and Matthias Jarke. Toward reference models for requirements traceability. *IEEE Trans. Softw. Eng.*, 27(1):58–93, January 2001.

[26] M. Riebisch. Supporting evolutionary development by feature models and traceability links. *Engineering of Computer-Based Systems, 2004. Proceedings. 11th IEEE International Conference and Workshop on the*, pages 370–377, 24-27 May 2004.

[27] Matthias Riebisch and Ilka Philippow. Evolution of product lines using traceability. In *In: Proceedings of Workshop on Engineering Complex Object-Oriented Systems for Evolution at OOPSLA 2001. (2001) Published online at: http://www.dsg.cs.tcd.ie/ecoose/oopsla2001/papers.shtml. Program Understanding - Issues, Tools, and Future Directions 15*, 2001.

[28] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. Generic semantics of feature diagrams. *Computer Networks*, 51(2):456–479, February 2007.

[29] TMate Software. Svnkit. [sub]versioning for java. `http://svnkit.com`, 2009.

[30] Tier3. RightChoiceDSS, Helping organizations make objective decisions. `http://tier3-inc.com/pdf/RightChoice.pdf`, September 2009.

[31] Tigris.org. Subversion website. `http://subversion.tigris.org`, april 2008.

[32] P.M. van den Broek and I. Galvao Lourenco da Silva. Analysis of feature models using generalised feature trees. In *Third International Workshop on Variability Modelling of Software-intensive Systems*, number 29 in ICB-Research Report, pages 29–35, Essen, Germany, January 2009. Universität Duisburg-Essen.

[33] Wikipedia. Comparison of revision control software. `http://en.wikipedia.org/wiki/Comparison_of_revision_control_software`, March 2008.

[34] Wikipedia. List of revision control software. `http://en.wikipedia.org/wiki/List_of_revision_control_software`, March 2008.

[35] L. Yu and S. Ramaswamy. A configuration management model for software product line. In *INFOCOMP, Journal of Computer Science*, volume 5, number 4, pages 1–8, December 2006.