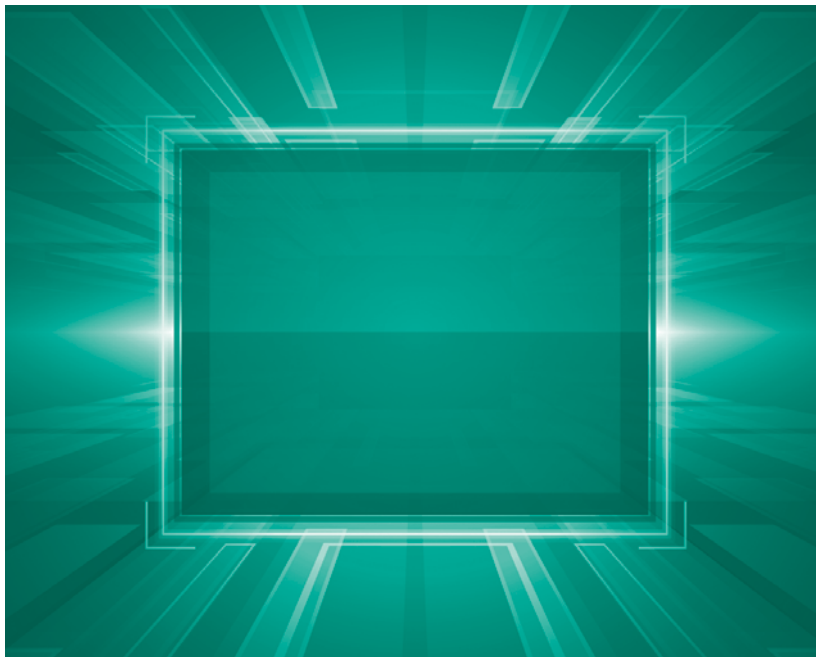


Michail Anastasopoulos

Evolution Control for Software Product Lines: An Automation Layer over Configuration Management



Editor-in-Chief: Prof. Dr. Dieter Rombach
Editorial Board: Prof. Dr. Frank Bomarius
Prof. Dr. Peter Liggesmeyer
Prof. Dr. Dieter Rombach

FRAUNHOFER VERLAG

PhD Theses in Experimental Software Engineering

Volume 49

Editor-in-Chief: Prof. Dr. Dieter Rombach

Editorial Board: Prof. Dr. Frank Bomarius
Prof. Dr. Peter Liggesmeyer
Prof. Dr. Dieter Rombach

Zugl.: Kaiserslautern, Univ., Diss., 2013

Printing:
Mediendienstleistungen des
Fraunhofer-Informationszentrum Raum und Bau IRB, Stuttgart

Printed on acid-free and chlorine-free bleached paper.

All rights reserved; no part of this publication may be translated, reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. The quotation of those designations in whatever way does not imply the conclusion that the use of those designations is legal without the consent of the owner of the trademark.

© by **Fraunhofer Verlag**, 2014
ISBN (Print): 978-3-8396-0702-2
Fraunhofer-Informationszentrum Raum und Bau IRB
Postfach 800469, 70504 Stuttgart
Nobelstraße 12, 70569 Stuttgart
Telefon +49 711 970-2500
Telefax +49 711 970-2508
E-Mail verlag@fraunhofer.de
URL <http://verlag.fraunhofer.de>

Evolution Control for Software Product Lines: An Automation Layer over Configuration Management

Beim Fachbereich Informatik
der Technischen Universität Kaiserslautern
zur Verleihung des akademischen Grades

Doktor der Ingenieurwissenschaften (Dr. Ing.)

genehmigte Dissertation
von

Dipl. Math. Michail Anastasopoulos

Fraunhofer-Institut für Experimentelles Software Engineering
(Fraunhofer IESE)
Kaiserslautern

Berichterstatter:

Prof. Dr. Dieter Rombach
Prof. Dr. Ulrich Eisenecker

Dekan:

Prof. Dr. Arnd Poetzsch-Heffter

Tag der Wissenschaftlichen Aussprache:

29.11.2013

D 386

To Caroline

Abstract

Modern software organizations increasingly aim at the development of individualized customer solutions in a cost-effective way. Product line engineering is a paradigm on the rise that addresses this goal and achieves true order of magnitude improvements in efficiency, quality and time-to-market.

In order to achieve these improvements product line engineering builds on strategic software reuse. In this regard the development lifecycle is decomposed in two major parallel running activities: family and application engineering. The former develops software assets for reuse across the product line while the latter applies reusable assets in the context of particular products. In order to keep the evolution of a product line under control family and application engineering must be continuously coordinated.

Software Configuration Management (SCM) is an established and mature technology for evolution control of software systems. However traditional SCM does not address explicitly the particular needs of product line engineering. When a product line evolves the number of product line members and variations increase significantly. In terms of scalability SCM systems can deal with the increased complexity. However, the execution of multiple configuration management operations becomes necessary in order to carry out evolution control scenarios within a product line.

Therefore, users of SCM systems can easily get overwhelmed while trying to make use of SCM mechanisms in order to keep product line evolution under control. In this context a serious amount of effort is often spent in practice to synchronize changes between family and application engineering. In some cases the synchronization is even neglected as the additional burden is not bearable. This in turn leads to unnecessary duplication of work and other serious problems. In the long term an organization possibly faces significant deficiencies in the timely delivery of software products.

Overall problem addressed by this thesis is that software configuration management requires significant effort, when used for the coordination of product line engineering processes.

Main Goal of this thesis is to reduce the software configuration management effort for the coordination of product line engineering processes.

This thesis introduces a virtual layer – called Customization Layer in the following – that bridges the gap between product line engineering and SCM. The layer offers a set of specialized evolution control operations for product lines, while conventional SCM operations are used behind the scenes and in an automated way.

The idea of the Customization Layer is accompanied by a method that enables the specification of the product line at hand and the selection of the necessary evolution control scenarios. Subsequently the method provides guidelines for the implementation of the scenarios based on the configuration management system available within an organization.

The approach has been validated through structural and usability evaluation, experimental studies as well as partially through a case study with an industrial organization. The results have shown potential for significant improvements in terms of efficiency and effectiveness of evolution control.

Acknowledgements

The process of writing a PhD is a long-term endeavor that requires solid foundations, proper guidance, great power of endurance and naturally hard work. Many people have contributed to these four requirements and I would like to take this opportunity to thank them.

I want to start with Dirk Muthig who was leading the department of product line architectures at Fraunhofer IESE when I started with this PhD. He was the one with the initial idea on the Customization Layer that was concretized during several fruitful discussions. Dirk helped a lot in overcoming the initial inertia and in getting this work on the right track.

Next, I would like to refer to the exemplary supervision of this thesis by Professor Dieter Rombach and by Professor Ulrich Eisenecker. Both have guided me through this work and gave valuable feedback as the thesis was evolving. Especially Professor Eisenecker joined the process at a later stage, yet showed a great interest and provided constant and proficient support in spite of the distance and in spite of repeated postponements. I would also like to acknowledge the contribution of Martin Becker the later head of the product line department at IESE. Martin helped in the finalization of this thesis by providing significant organizational support. Actually, all members of the product line department have helped me in the one or the other way. I want to give special mention to my former colleagues Jens Knodel, Slawomir Duszynski, Adeline Silva-Schäfer, Thiago Burgos and Vander Alves for their continuous assistance.

Regarding endurance and hard work I have to give all credits to my wife Caroline. She was there to take care of our children, when I was working overhours, she gave me all necessary faith during the difficult phases and she was patient when things were taking longer than originally planned. Without her this work would never get done and I am really grateful to her for that.

Table of Contents

Abstract	v
Acknowledgements	vii
Table of Contents	ix
List of Figures.....	xiii
List of Tables	xvii
1 Introduction	1
1.1 Software Reuse	2
1.2 Product Line Engineering.....	5
1.3 Problem Definition	7
1.3.1 Problem, Goals and Hypotheses.....	9
1.3.2 Research Questions.....	10
1.3.3 Basic Configuration Management Concepts	12
1.3.4 Computer Theoretical Contribution	13
1.3.5 Practical Contribution	14
1.4 Importance of the Problem.....	20
1.5 Solution Approach	22
1.6 Methodological Approach.....	23
1.7 Output of this thesis.....	24
1.8 Outline of this thesis	25
1.9 Research Context.....	26
2 Related Work	27
2.1 Nature of product line evolution	28
2.1.1 A model of system families	28
2.1.2 Variability Specification Language	30
2.1.3 Product Line Evolution Scenarios.....	31
2.1.4 Process dynamics	33
2.2 Implementation of product line evolution.....	35
2.2.1 Molhado SPL.....	36
2.2.2 Product Line Asset Manager	36
2.2.3 Configuration Management	38
2.2.4 The Kobra method	45
2.2.5 Software Frameworks	46
2.3 Discussion	47
2.4 Section summary.....	50

3	Conceptual Model of Evolution Control.....	51
3.1	Introduction to control theory and feedback	51
3.2	Feedback in product line engineering	52
3.3	Introduction to control loops.....	53
3.4	Control loops and configuration management	54
3.4.1	Change management	55
3.4.2	Version management.....	55
3.4.3	Status accounting	55
3.5	Control loops in product line engineering	56
3.6	Types of product lines	57
3.6.1	Individual (or independent) Products	57
3.6.2	Product Generations	58
3.6.3	Standard Application	58
3.6.4	Professional or Customizable Application.....	58
3.6.5	Standardized Infrastructure	59
3.6.6	Platform.....	59
3.6.7	Product Population	59
3.6.8	Software Product Line.....	60
3.6.9	Hierarchical Product Lines	60
3.6.10	Production Lines	61
3.6.11	Adaptive Product	61
3.7	Conceptual model.....	61
3.7.1	Validation of instance models	63
3.7.2	Code Generation	64
3.8	Refined Conceptual Model.....	64
3.9	Role of the conceptual model.....	66
3.10	Section summary.....	68
4	Data Model of Evolution Control.....	69
4.1	Basic Asset Model	69
4.1.1	Core Assets.....	70
4.1.2	Instances and product-specifics	71
4.2	Asset State Model	73
4.2.1	Core asset states.....	74
4.2.2	Product asset states	77
4.2.3	Identified operations	79
4.3	Role of the Basic Asset Model	80
4.4	Variability Management	81
4.4.1	Connection to Basic Asset Model.....	84
4.4.2	Connecting the Basic Asset Model to Decision Models...85	
4.5	Section summary.....	87
5	Process Model of Evolution Control	89
5.1	Evolution Control Scenarios for Family Engineering	90
5.1.1	Creation of core asset change requests.....	90
5.1.2	Scenarios for version management of core assets.....	92
5.2	Evolution Control Scenarios for Application Engineering	95

5.2.1	Creation of product asset change requests	96
5.2.2	Scenarios for Version Management of product assets	96
5.3	Common Status Accounting Scenarios.....	99
5.4	Change impact analysis.....	103
5.4.1	Comparing core assets with instances.....	103
5.4.2	Formal model for impact analysis	105
5.4.3	Change impact analysis activities	107
5.5	Interaction with Variability Management.....	112
5.5.1	Core asset creation	112
5.5.2	Instance creation.....	113
5.5.3	Further interactions.....	115
5.6	Section summary.....	115
6	Interaction with Configuration Management	117
6.1	Implementation Guidelines.....	118
6.1.1	Guidelines for version management scenarios.....	118
6.1.2	Guidelines for change management scenarios	127
6.1.3	Guidelines for status accounting scenarios.....	130
6.2	CMS functionality and the Customization Layer	131
6.2.1	Main Functionality Blocks.....	131
6.2.2	Structuring.....	131
6.2.3	Controlling	132
6.2.4	Versioning	133
6.2.5	Construction.....	135
6.2.6	Accounting and Auditing	137
6.2.7	Team	137
6.2.8	Process	138
6.3	Section summary.....	138
7	A Customization Layer framework.....	139
7.1	Customization Layer.....	140
7.2	User Interface.....	141
7.3	Command Parser	143
7.4	Section summary.....	144
8	Adoption process	145
8.1	Characterization.....	146
8.2	Goal definition	150
8.3	Process selection	151
8.4	Execution	151
8.5	Analyze experiences	152
8.6	Prepare experiences for reuse.....	153
8.7	Section Summary	153
9	Validation.....	155
9.1	Structural evaluation	156
9.1.1	Core Decision: Layering.....	156

9.1.2 Data Model.....	158
9.1.3 Process Model.....	160
9.2 Usability evaluation	161
9.2.1 Planning.....	161
9.2.2 Tasks.....	163
9.2.3 Interface descriptions	164
9.2.4 Analysis.....	165
9.3 Experimental validation	170
9.3.1 Customization Layer	170
9.3.2 Configuration Management.....	171
9.3.3 Experiment 1	175
9.3.4 Experiment 2	184
9.4 Case Study	187
9.4.1 Setting	188
9.4.2 Implementation and experiences.....	189
9.4.3 Results and recommendations.....	190
9.5 Section summary.....	190
10 Conclusion.....	191
10.1 Research Questions	191
10.2 Validation.....	192
10.3 Limitations	192
10.4 Future Work.....	194
References	197
Appendix	209

List of Figures

Figure 1:	Software reuse activities and interrelations.....	3
Figure 2:	Synchronization in single-system and product line engineering.....	9
Figure 3:	Refining the goals of this thesis.....	10
Figure 4:	Lifetime of an asset in terms of branches and versions	13
Figure 5:	Example product line	15
Figure 6:	Example Instantiation	15
Figure 7:	Traceability through branching.....	16
Figure 8:	Basic product line engineering coordination	17
Figure 9:	Branching in open-source development (graph obtained through mining of the GCC version history).....	18
Figure 10:	Merging in open source development (graph obtained through mining of the GCC version history).....	19
Figure 11:	Customization Layer concept	23
Figure 12:	Components leading to a Customization Layer	25
Figure 13:	Model of product lines according to Belady and Merlin.....	29
Figure 14:	Matrix of permitted product line members according to Belady and Merlin	30
Figure 15:	Representation of software reuse with process dynamics	34
Figure 16:	PLAM operations	38
Figure 17:	Configuration with build management	39
Figure 18:	Voodoo version management approach.....	41
Figure 19:	Closed loop feedback	52
Figure 20:	Closed loop in Product Line Engineering	52
Figure 21:	Control loop	53
Figure 22:	Mapping control loop concepts to configuration management	54
Figure 23:	Control loops in product line engineering	56
Figure 24:	Conceptual model of evolution control	62
Figure 25:	Conceptual model (Xtext-based)	62
Figure 26:	Refined conceptual model (Processes)	65
Figure 27:	Refined conceptual model (Family Engineering).....	66
Figure 28:	Refined conceptual model (Application Engineering).....	66
Figure 29:	Example application of the conceptual model	67
Figure 30:	Types of assets and change requests.....	69
Figure 31:	Core assets	71
Figure 32:	Instance and product-specific assets.....	72
Figure 33:	Core Asset integration	74
Figure 34:	Core Asset Release.....	75
Figure 35:	Core Asset reuse monitor.....	75
Figure 36:	Core Asset change management	76

Figure 37: Product asset characterization	77
Figure 38: Rebasing instances	78
Figure 39: Operations identified through state models.....	80
Figure 40: Role of the asset model in the Customization Layer	81
Figure 41: Decision meta-model and relation to basic asset model.....	86
Figure 42: createCoreAssetChangeRequest scenario.....	91
Figure 43: createCoreAsset scenario	92
Figure 44: removeCoreAsset scenario	94
Figure 45: modifyCoreAsset scenario	94
Figure 46: integrateCoreAsset scenario	95
Figure 47: createProductAssetChangeRequest scenario	96
Figure 48: createProductAsset scenario	96
Figure 49: removeProductAsset scenario.....	98
Figure 50: modifyProductAsset scenario	99
Figure 51: rebaseProductAsset scenario	99
Figure 52: showChangeRequests scenario	100
Figure 53: showCoreAsset scenario	100
Figure 54: showCoreAssetInstances scenario	101
Figure 55: showCoreAssetChanges scenario.....	101
Figure 56: showProductAssets scenario	102
Figure 57: showInstanceCoreAssets scenario	102
Figure 58: showProductAssetChanges scenario	103
Figure 59: Reactive analysis of core asset changes	108
Figure 60: Proactive analysis of core asset changes	109
Figure 61: Reactive analysis of product asset changes.....	111
Figure 62: Proactive analysis of product asset changes.....	112
Figure 63: Interactions between Evolution Control and Variability Management (core asset creation)	113
Figure 64: Interactions between Evolution Control and Variability Management (instance creation)	114
Figure 65: Main CMS functionality blocks.....	131
Figure 66: Continuous integration paradigm adapted from [Duv07] ..	136
Figure 67: Customization Layer prototype screenshot	139
Figure 68: Framework structure	140
Figure 69: User interface classes and dependencies	142
Figure 70: Command parser classes and dependencies	143
Figure 71: Steps of the Quality Improvement Paradigm (QIP)	145
Figure 72: Product Line Type Decision Tree	147
Figure 73: Adopting a Customization Layer (steps 1 to 3)	151
Figure 74: Experimental V-Model of this thesis	155
Figure 75: Usability Evaluation / Actions per Task.....	166
Figure 76: Usability Evaluation / Cognitive Walkthrough Questions....	168
Figure 77: Usability Evaluation: Efficiency results.....	169
Figure 78: Customization Layer of the experiment.....	171
Figure 79: SVNConnector implementation.....	174
Figure 80: Average time for task execution (Experiment 1).....	179
Figure 81: Times per Group and Role.....	180

Figure 82: Times per Group and Task	181
Figure 83: Versions and Branches produced over time	182
Figure 84: Influence of experience to efficiency	184
Figure 85: Summary results of Experiment 2	185
Figure 86: Experiment 2 task times without outliers.....	186
Figure 87: Customization Layer of the case study	188

List of Tables

Table 1:	PLE survey: Participant classification	21
Table 2:	PLE survey results	21
Table 3:	Requirements to evolution control solutions.....	28
Table 4:	Categorization of evolution control related work	48
Table 5:	Related work characterization	49
Table 6:	Types of core asset instantiation.....	73
Table 7:	Variability management approaches.....	84
Table 8:	Population of core asset attributes	93
Table 9:	Population of core asset instance attributes	98
Table 10:	createCoreAsset guideline.....	118
Table 11:	createProductAsset guideline	120
Table 12:	removeCoreAsset guideline.....	122
Table 13:	removeProductAsset guideline	123
Table 14:	modifyCoreAsset guideline	124
Table 15:	modifyProductAsset guideline	125
Table 16:	integrateCoreAsset guideline	126
Table 17:	rebaseProductAsset guideline.....	126
Table 18:	createCoreAssetChangeRequest guideline	127
Table 19:	createProductAssetChangeRequest guideline	129
Table 20:	Main evolution control goals and sample refinement	149
Table 21:	Usability Evaluation / experience profiles	162
Table 22:	Usability Evaluation / Group assignments	164
Table 23:	Usability Evaluation / Unnecessary actions in CL group	167
Table 24:	Experiment variables	176
Table 25:	Arrangement of students	177
Table 26:	Experiment tasks	177
Table 27:	Distribution of student experience.....	183
Table 28:	Evaluation based on UTAUT	187

1 Introduction

A computer-based system is a collection of components – both hardware and software – organized to accomplish a specific function or set of functions [IEEE610.12]. From the time when the microchip has been invented until today the role of software became more prominent in computer-based systems. This is due to the increasing amount of functionality that can be realized only with software in a cost-effective way.

In the lifetime of a system the complexity of the constituent software parts continuously grows. This phenomenon has been first observed in the seventies by Manny Lehman who formulated the laws of software evolution [Leh80]. In particular, the 2nd law on increasing system complexity states that: “As an evolving program is continually changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain or reduce it.” As elucidated in [Leh78], the complexity grows because changes in software primarily aim at economic gain which is achieved by enhancing a system at the lowest cost possible.

As shown in further studies, continuous system enhancement primarily involves expanding and customizing the system to new requirements and secondly involves correcting errors [LST78][McK84]. Hence, in order to maximize economic gain organizations must be able to continuously address new requirements in an effective and efficient manner. Software reuse is a promising solution into that direction. The software reuse approach that was initially proposed in 1968 by McIlroy [Mc69] enables the exploitation of existing and previously proven software assets during the construction of new software.

In modern days agile development methods (e.g. Scrum [SB02] or Extreme Programming [BA05]) also confirm that economic gain in terms of customer satisfaction is the primary goal. In order to keep growing complexity under control, agile methods propose the frequent application of software refactoring. The latter aims at preserving the function of a system while reducing its complexity. In this context refactoring a system often leads to the introduction of software reuse.

Software reuse supports the efficient realization of new requirements since effort can be saved through the usage of existing assets rather than building assets from scratch. Usage in this context may entail modification of existing assets to address new requirements but it might also entail the as-is application of existing assets. Apart from efficiency,

reuse also supports effectiveness because the quality of reused assets has been already assured. However it is necessary that efficiency and effectiveness do not fade over time. This research problem, namely assuring that the advantages of software reuse are maintained over time, is addressed in the present thesis.

In the last decades, continuous system enhancement has been combined with a shift away from mass production towards mass customization [Pine93]. Mass production aims to reduce costs by improving the production process so that many identical copies of the same product can be produced in shorter time. Mass customization goes a step further and enables to produce many customized products out of a product portfolio.

To address mass customization production steps must entail a lower amount of manual steps and a higher amount of pre-arranged or automated steps. This in turn requires that a software engineering capability is set-up accordingly. Product line engineering is a development paradigm that enables setting-up this software engineering capability for mass customization based on software reuse.

The present thesis focuses on product line engineering and aims at the sustainability of the reuse-based software engineering capability, which is created in a product line engineering context.

The following two subsections will introduce software reuse and product line engineering in more detail.

1.1 Software Reuse

Definition 1. An asset is an item, such as design, specifications, source code, documentation, test suites, manual procedures, etc., that has been designed for use in multiple contexts [IEEE1517-2010]

Definition 2. Software reuse is the use of an asset in the solution of different problems [IEEE1517-2010]

Software reuse can be generally divided into development for and to development with reuse. The former provides the reusable assets while the latter exploits them. Figure 1 shows a sequence of basic steps that should be involved during software reuse. The dotted lines illustrate bi-directional relations between development for and with reuse. These relations can be bi-directional: Development for reuse delivers assets to development with reuse. On the other hand development with reuse can give feedback to development for reuse. The feedback can be helpful in order to trace reuse cases and to evaluate the level of reusability.

Definition 3. Reusability is the degree to which an asset can be used in more than one software system, or in building other assets [IEEE1517-2010])

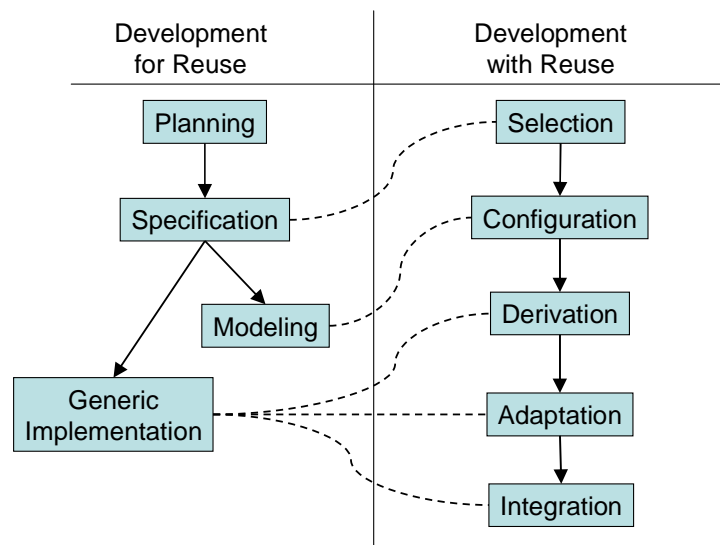


Figure 1: Software reuse activities and interrelations

Development for reuse commences with a planning step that identifies the domain in which the reusable assets will be employed as well as the corresponding reuse requirements [Ra05]. Then in the specification step the properties of the reusable assets are defined [La91]. This specification is important since it enables identifying and selecting reusable assets later. Subsequently the implementation including quality assurance of reusable assets can start. Since those assets are to be employed in various situations within a domain the implementation ought to be generic enough to allow that. In other words the implementation should be customizable for different situations. Typical example for a customizable implementation is the provision of abstract software assets that can be refined during the customization process.

In parallel to the implementation a modeling activity [Mut02] takes place that describes which elements of a reusable asset can be customized and how this can be done. This information may be contained in the implementation as well. The modeling activity captures this information explicitly. The resulting model describes customization options, effects as well as possible interrelations and enables reusers to reason about them. Put simply the modeling step creates a “configurator” for reusable assets.

Development with reuse starts with a selection of the reusable assets based on their specification. The selection may involve identification as well as evaluation of assets [Mut02]. After selection the configuration step takes place [Be04]. This step is guided by the customization model attached to every reusable asset. During configuration, reusers set-up the reusable assets for the specific context in which they are to be reused.

Identification, evaluation and configuration can be significantly facilitated if a declarative approach is followed in the modeling step. The declarative approach allows the representation of a reusable asset based on its characteristics that can be selected (i.e. declared). In [CE00] a declarative representation is compared to a procedural (or less direct representation). For example, thread synchronization can be an optional nonfunctional characteristic of an asset. With a declarative approach a reuser can declare whether synchronization is necessary or not. The declaration leads to the automated generation of the synchronization algorithm. Without a declarative approach the reuse would have to know internals of the asset and the synchronization algorithm in order to combine them.

Based on the configuration decisions the derivation step is performed [Be04]. This step yields an instance of the respective reusable asset based on the configuration decisions from the previous step. To this end reusers follow instructions in the customization model that describe the effects of customization decisions to the implementation of reusable assets. Depending on the form of the customization model the derivation step can be automated. After derivation the adaptation step possibly takes place [Be04][Kr92]. Here implementations of reusable assets are changed further – usually in a manual way – to reflect specific needs that were not considered during development for reuse. At this point it is crucial to carefully contrast local manual adaptation with a global augmentation of the reusable asset. In the former case the instance of the reusable asset is adapted locally, possible for one single customer. In the latter case the original reusable asset is augmented and the changes are available globally.

Finally the adapted asset is integrated with other assets in the respective software system [Mut02]. The integration effort may depend on the implementation technology employed during development for reuse (e.g. usage of interfaces or other module interconnection mechanisms) [Kr92].

Software reuse, however, does not succeed by default [Sch99]. There are basic prerequisites in terms of technology, methods and processes and there are also success factors like common understanding and acceptance by people. Product line engineering is addressing the basic

prerequisites by providing a well-defined and repeatable reuse-oriented development process.

1.2 Product Line Engineering

Definition 4. Product Line Engineering (PLE) aims at the systematic development of a set of similar software systems by understanding and controlling their common and distinguishing characteristics [Mut02].

Product line engineering addresses the challenge of delivering software solutions tailored to individual needs of customers or environments. In such cases, organizations typically produce a series of related software systems instead of one singular system. It is often the case that an organization produces only one singular system. However, even in such cases, the system is often customizable or available in various flavors. Having only one static instance of a software system that addresses all individual requirements is in many cases not sustainable from the economic point of view.

Hence, product line engineering proposes to consider related software systems or the different flavors of one singular system as a family of systems, namely as a product line¹. One of the well-known definitions of the term product line is given in the following.

Definition 5. **(A)** A product line is a group of products that provide a core benefit yet differ along attributes which affect the buying behavior of different customer groups. **(B)** A product line is a group of products that are closely related, either because they function in a similar manner, are sold to the same customer groups, are marketed through the same types of outlets or fall within given price ranges. [Wit96]

Product line engineering has proved to bring a series of benefits to organizations [CN02]. However there are also serious challenges. Definition 5B states that product line members share a common, managed set of features. That means that a management process is necessary that determines the common and varying features (or characteristics in general) in a product line that have good potential to satisfy the specific needs of a particular market segment or mission. This activity is usually called scoping since it determines the scope of the product line.

Product line engineering is based on strategic reuse as opposed to opportunistic reuse which may be applied in single-system development.

¹ In the context of this thesis the term product line will be often used as an abbreviation for the term software product line

Strategic reuse requires that software is made reusable according to a plan that, when followed, is expected to achieve clear improvements. In a product line context the scoping activity delivers this plan. By following this approach the challenge of continuous evolution can be better met than with a traditional single system approach. There are various scenarios where this can be illustrated. For example:

- A new requirement to a product line member can be realized efficiently by reusing the realization of a similar requirement in another member.
- Analyzing the impact of a change in a member of the family requires less effort since each member has well defined boundaries within the product line scope.
- Creating a new product line member as a customized product is accelerated by the fact that the rules and processes of becoming a product line member are clearly defined.

In order to implement the close relation between members of a product line it is necessary to develop from a common set of core assets in a prescribed way. The term core asset refers to assets that have been made reusable in order to realize the common and varying characteristics determined in the scoping phase.

Definition 6. Core asset is an asset or resource that is built to be used in the production of more than one product in a software product line [BC05]

Product line engineering involves the following parallel running engineering processes:

- The development-for-reuse process that develops according to the product line scope the core assets to be reused across the different members of the product line in a prescribed way. In the context of product line engineering this process is commonly named family engineering.
- A set of product engineering processes (one process for each member of the product line) that develop the assets which make up the final products. Those assets are either core asset instances that are created via the reuse of core assets (i.e. development for reuse) or product-specific assets that are created in another way (i.e. from scratch or third-party reuse). Core asset instances and product-specific assets constitute product assets. In the context of product line engineering this process is commonly named application engineering.

Definition 7. Core asset instance (or simply instance) is an asset or resource that is created in the context of a product via software reuse of one or more core assets

Definition 8. Product-specific asset is an asset that is created in the context of a product without reusing core assets

Definition 9. Product asset is either a product-specific asset or a core asset instance

Family and application engineering are not disconnected from each other. On the contrary there is a series of interrelations (cf. Figure 1) that must be continuously respected as a product line evolves. Hence, for a product line engineering effort to be successful over time it is necessary to coordinate and control the evolution within and between family and application engineering.

Evolution control has been defined in [Mit06] as the situation in which “change management from requirements onwards till feature integration allows planning”. In the context of this thesis this definition can be reformulated as follows:

Definition 10. Evolution control is the activity of monitoring and evaluating changes that take place on assets of an ongoing engineering process so that correcting actions can take place when necessary

This defines the overall focus of this thesis.

Focus of this thesis: Focus of the present thesis is the facilitation of evolution control in a product line engineering process.

Restriction of this thesis: The focus is restricted on the engineering processes in a product line, namely on family and application engineering. Clearly the scoping process is also involved in product line evolution. However this is not covered by the present thesis.

1.3 Problem Definition

Software configuration management is an indispensable discipline for the coordination of activities in complex development processes and it is mandatory component in several process improvement models like CMMI [ACT01] or BOOTSTRAP [KSK+94]. Software configuration management technology and experiences are available in most software organizations. This know-how is indispensable and must be taken into account when an organization is embarking on product line engineering.

As the term implies software configuration management is about managing configurations. There are several official definitions of the term configuration (including for example the definition in [IEEE610.12]). However the time-dependent nature of a configuration as it applies to software configuration management is often neglected. Therefore the following definition is preferred.

Definition 11. A configuration is a set of software assets that have been developed to coexist harmonically in the context of a system at a specific point in time.

Based on the above definition software configuration management can be defined as follows.

Definition 12. Software configuration management is the discipline of identifying the configuration of a system at discrete points in time for purposes of systematically controlling changes to this configuration and maintaining the integrity and traceability of this configuration throughout the system life cycle [BSH80]

Configuration management² supports the evolution of software systems by enabling and controlling parallelism and synchronization in joint development efforts. In single-system development this usually entails multiple engineers implementing multiple changes in a single software system. Although configuration management provides mechanisms and concepts, this kind of parallelism is generally considered difficult [Ba99]. If the means of the configuration management system are not used properly, joint development efforts can easily get off hand. In a product line context this situation becomes even more difficult since the variability dimension has to be additionally considered. The latter captures the various forms that core assets take within a product line as they progress from the family to the application engineering phases.

Figure 2 illustrates the implications of the shift from single-system engineering to product line engineering. While in the first case synchronization between engineers is required only within a single product, in the second case the synchronization links spread across the variability dimension.

² In the context of this thesis the term configuration management will be often used as an abbreviation for the term software configuration management. This is not to be confused with the traditional definition of configuration management that also includes the management of hardware parts.

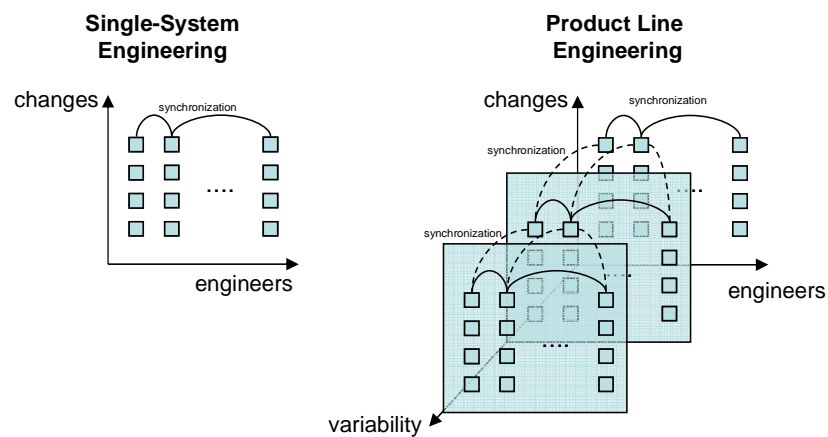


Figure 2: Synchronization in single-system and product line engineering

Given this additional complexity, engineers can easily get overwhelmed while trying to make proper use of the configuration management mechanisms and concepts. For that reason a serious amount of effort is often spent in practice for the synchronization of changes. In some cases the synchronization is even neglected as the additional burden is not bearable. This in turn leads to unnecessary duplication of work and other serious problems regarding the maintainability of the system under development und the team productivity.

1.3.1 Problem, Goals and Hypotheses

According to the above discussions the overall problem addressed by this thesis and the corresponding research goal can be formulated as follows.

Overall problem addressed by this thesis is that software configuration management requires significant effort, when used for the coordination of product line engineering processes

Main Goal of this thesis is to reduce the software configuration management effort for the coordination of product line engineering processes.

The above goal can be refined according to the Goal-Question-Metric method [BCD94] as shown in the following figure.

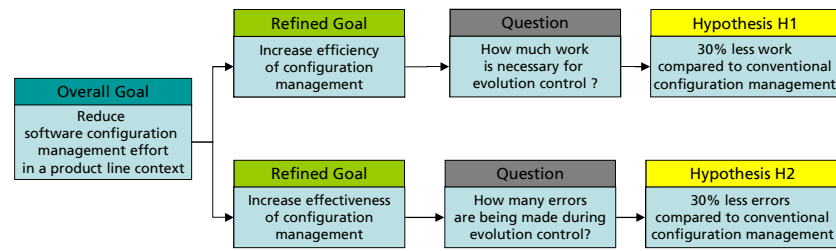


Figure 3: Refining the goals of this thesis

The effort of configuration management can be expressed in terms of efficiency and effectiveness. Efficiency can be characterized by means of the work that is necessary to perform evolution control activities in a product line. On the other hand, effectiveness is characterized in terms of errors that are produced during evolution control. As shown in Figure 3 this thesis defines two main hypotheses:

H1: The solution proposed in this thesis enables to save a significant amount of work units in product line evolution control compared to conventional configuration management

H2: The solution proposed in this thesis enables to achieve significantly less errors during product line evolution control compared to conventional configuration management

1.3.2 Research Questions

In addition to the overall goal of this thesis (effort reduction) and the two refined goals (i.e. increasing effectiveness and efficiency) described above, a set of research questions are derived in this section. These research questions characterize the context, to which the goals of this thesis apply.

As described in [DB91] there is a series of significant software configuration management questions that arise from the application of software reuse.

- Traditional software configuration management mainly controls changes at the subsystem level. This strategy may not be applicable to a reuse repository which consists of reusable assets at various granularity levels. Should software configuration management apply change control procedures to all granularity levels in this case? If not, to which ones and under what conditions?
- Successful software reuse requires assets to be reused numerous times. Should software configuration management keep track of all

reuse cases? This implies significant effort but on the other hand it is necessary for the analysis and propagation of changes between reusers and reuse repository.

- Reusable assets are characterized by the fact that they can be easily specialized to various requirements. When a reusable asset is changed it may be necessary to propagate the changes to all its specializations. Equally when a specialization of a reusable asset changes it may be necessary to propagate changes to its origin in the reusable library. How can this change propagation take place given the fact that reusable assets can significantly differ from specializations?
- Software configuration management usually includes access and ownership rules. They prescribe who has the right to perform changes on what assets and what kind of changes. How should these rules be defined in a reuse setting? Is a reusable asset owned only by the developer who put it in the reuse repository? Or is it also owned by the corresponding reusers? What kind of change operations are allowed in every case?

The above discussion and the main goal of this thesis (section 1.3.1) lead to the derivation of the following research questions for the present thesis:

Research Question 1: How can configuration management deal with core assets and core asset compositions at different granularity levels? How does this relate to the instantiation of core assets?

Research Question 2: How can configuration management keep track of numerous core asset instantiations and at the same time keep the effort under control?

Research Question 3: How can software configuration management avoid that the same changes are performed redundantly in product line members.

As discussed in the beginning of section 1.3 configuration management is an indispensable discipline for controlling the evolution of software systems. Therefore, organizations embarking on product line engineering typically have a configuration management system already in place. In most cases considerable capital and effort have been invested for system acquisition, set-up, customization and optimization. Ideally, such investments should be preserved and utilized by a configuration management solution for product lines. Moreover such a solution must be able to operate with different configuration management systems in

order to be widely applicable. This leads to an additional research question:

Research Question 4: How can we take advantage of existing configuration management systems when addressing product line evolution control?

1.3.3 Basic Configuration Management Concepts

In the upcoming sections 1.3.4 and 1.3.5 the theoretical and practical reasons are explained, which justify the problem addressed in the present thesis. To this end a series of additional configuration management concepts are introduced in the following. The majority of the definitions have been derived from the upcoming standard [OSLC10].

Definition 13. A configuration item is a software asset under the control of software configuration management.

Every configuration item undergoes various changes during its lifetime. For the purpose of recording and reproducing changes configuration management systems support the concept of versioning. Versions are usually created to denote, to persistently store and to communicate important steps in the lifetime of an asset.

Definition 14. A version is a resource representing the contents of a configuration item a particular point in time.

When an asset is to be developed jointly within a group, the usage of branches can be beneficial. Branches can be used to organize activities in a joint development effort. Different tasks can be taken over by different branches that run in parallel. Obviously, when this kind of parallelism is enabled coordination between branches becomes necessary so that the outcomes of the different parallel activities can be integrated.

Definition 15. A branch contains the consecutive versions of a configuration item that haven been created in a specific context.

For example there may be branches containing the versions created by a specific developer, the versions created during quality assurance or the versions created during the adaptation of a reusable asset.

A branch is created off another branch. That means that the first version of a configuration item in a branch is a copy of a previously existing version in another branch of the same configuration item. An exception constitutes the main branch (also called root branch or mainline) of an item. In this case the first version represents the contents of the item,

when it was first put under configuration management control. Following picture illustrates the lifetime of a configuration item in terms of branches and versions. The picture also shows relations between branches in terms of branching off and integration activities. The picture shows a directed acyclic graph, meaning that branches have always to join the branches they have been created off. This is not always the case in practice. There may be situations where branches do not join their parent branches. This applies for example to permanent branches or to throw-away prototype branches.

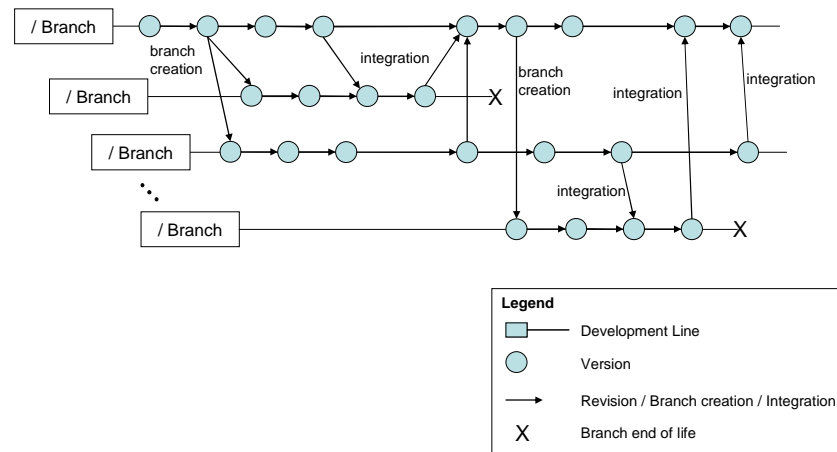


Figure 4: Lifetime of an asset in terms of branches and versions

1.3.4 Computer Theoretical Contribution

From the computer theoretical point of view there is a semantic gap between product line engineering and software configuration management. That means that concepts in the semantic domain of product line engineering are fundamentally different than the corresponding concepts in the semantic domain of software configuration management. Therefore the usage of configuration management for evolution control of a product line requires a mapping between these two semantic domains. If configuration management is seen as a programming language that helps to implement evolution control, there may be patterns in this language that are particularly useful in a product line context. Then, bridging the semantic gap involves identifying these patterns and mapping them to higher level product line operations.

As discussed in section 1.2 a product line operates on two types of assets: core assets and product assets. The latter can be further broken down to core asset instances – in short instances – namely assets created by reusing core assets, and to product-specific assets created from

scratch or through third-party reuse. Moreover, in product line engineering core assets must be associated with the instances that have been derived from them. Ideally those associations should describe how core assets are being reused.

Configuration management does not make a distinction between different types of assets and operates on configuration items. That means that configuration items must be used to manage core as well as product-specific assets. On the other hand associations between core and derived assets can be compared to the branch creation operation in configuration management that relates a child branch to a parent branch. However the semantics of branch creation is different than the semantics of derivation [Mah95]. Branch creation means that temporal parallelism is required while derivation implies software reuse.

Hence the theoretical contribution of this thesis can be defined as follows:

Computer Theoretical Contribution of this Thesis: The computer theoretical contribution of this thesis is to bridge the semantic gap between product line engineering and software configuration management.

1.3.5 Practical Contribution

In order to explain the practical problems that arise when standard configuration management is used in a product line context an example will be presented in the following.

The example is centered on the concept of a collection. In software development collections are parts of libraries that provide mechanisms for the storage and manipulation of objects. In other words collections are containers for objects during the execution of a program. Typical examples of collections are arrays, hash tables or vectors. Therefore a product line of different collections is assumed in the following.

The product line contains three products as members. Family engineering has produced a collection library (L) which contains two collections, namely array (A) and vector (V). L is the container of (A) and (V) and might add for example functionality common to A and V. The library is reused during application engineering in each of the three members. The following picture illustrates the setting.

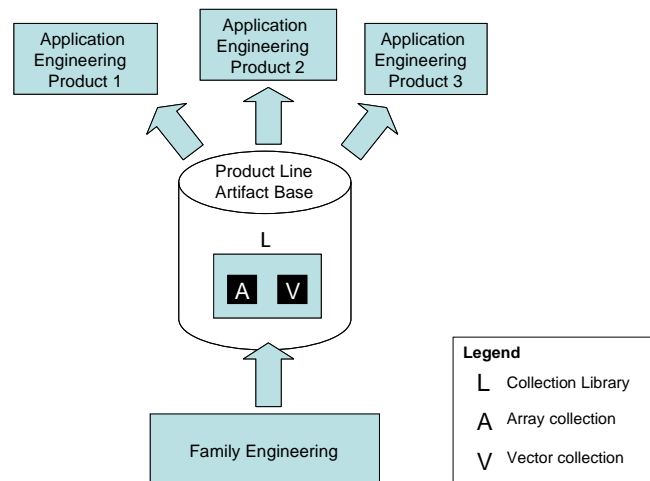


Figure 5: Example product line

During the reuse of the collection library L for the first product A must be adapted. The adaptation changes A to A' (for example the array must support a sorting algorithm in product 1) while V is left untouched. In the case of product 2 A and V are reused as-is. Finally for product 3, A is left untouched and V must be adapted to V' (for example to add multithreading support to the collection). Figure 6 illustrates the situation.

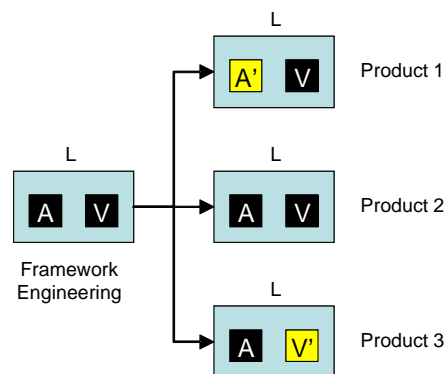


Figure 6: Example Instantiation

In order to manage the lifetime of the collection library L in a situation such the above the usage of branches at the top-level (i.e. at the level of L in this example) is typical. That means that a branch off the original L branch is created for each product. Upon branch creation the contents of L (i.e. A and V) are copied into the branch. Afterwards the changes to A and V are performed.

By following this approach configuration management usually enables traceability. That means that when L is branched components A and V are automatically branched as well. Moreover the resulting L, A and V branches can be linked to each other. Figure 7 shows all versions and branches of L, A and V for the three products. The dotted regions highlight the branches for product 1 and the arrows the corresponding traceability.

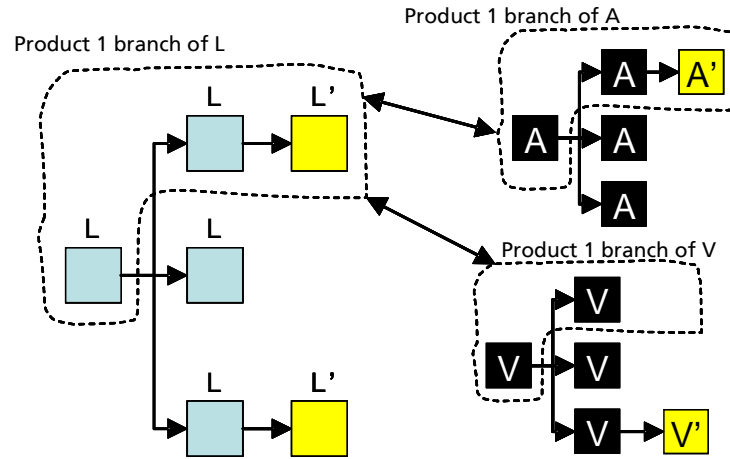


Figure 7: Traceability through branching

The problem with branches in a product line setting is that the user of the configuration management system has to combine numerous operations in order to coordinate activities in the product line engineering process. Coordination involves two main steps: monitoring and propagating changes. A basic coordination schema is given in Figure 8.

Figure 8 describes the maximum sequence of steps for the coordination of family and application engineering. There may be situations where only a part of this sequence applies. The sequence starts (step 1) with a change that has been identified in a derived asset. In step 2 the change is propagated to the core asset from which the changed asset has been derived. Such propagation is necessary because a product-specific change is possibly due to faults, weaknesses or missing characteristics of core assets. In such cases the affected core assets should be revised (step 3). After the core asset revision the changes are propagated to the corresponding derived assets (step 4) and finally the latter are possibly changed as well (step 5).

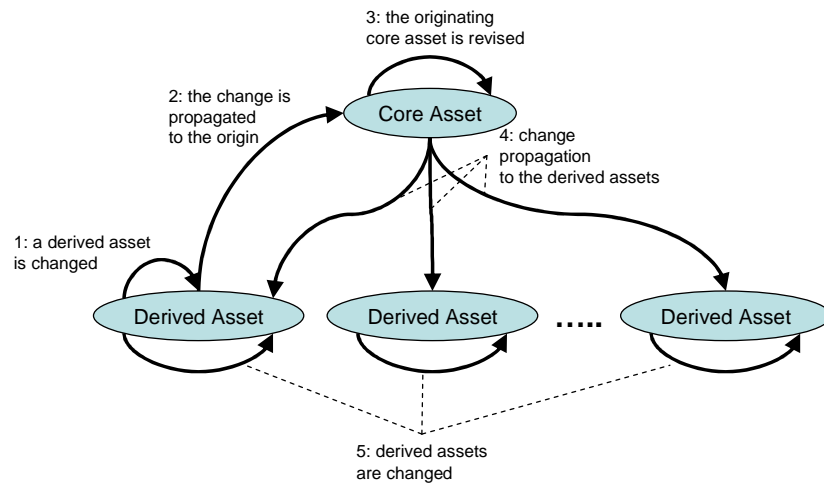


Figure 8: Basic product line engineering coordination

Steps 1 to 3 of Figure 8 represent a single evolution control scenario, namely the detection of changes in the assets derived from a core asset. In order to execute this scenario for the library example with the help of conventional configuration management (i.e. using versions and branches) a series of operations become necessary:

1. Open the version set of library L that contains all produced versions
2. Identify the product branches (there may be many other temporary branches next to the product branches)
3. Identify when the product branches have been synchronized with the family engineering branch that contains the original library L
4. For each product branch look for new L versions since the last synchronization
5. For each new version of L query the configuration management System for the changes made in that version
6. Filter out product-specific changes and identify changes that may affect L
7. Consolidate the changes between L and the product-specific instances

In many cases as a product line evolves the amount of branches increases significantly. This can lead to significant effort for the execution of configuration management operations as described in the above seven steps. Open-source systems offer an interesting possibility for the study of the branching phenomenon in development processes. Figure 9 provides the result obtained by analyzing a part of the history of the GNU Compiler Collection (gcc). In fact, only the branches have been

considered, for which merging (i.e. synchronization with other branches) information was explicitly available.

Figure 9 depicts the graph that has been obtained by mining the gcc version history. The nodes and the edges represent branches and relations (i.e. parent-to-child branch relations) respectively. In addition the size and the color depth of nodes provide a measure of the connected edges (i.e. increased size and dark color signify that many edges are connected to the particular node). The circular layout of the graph enables identifying a set of branch groups, in which the central branch is the parent for the branches in the perimeter.

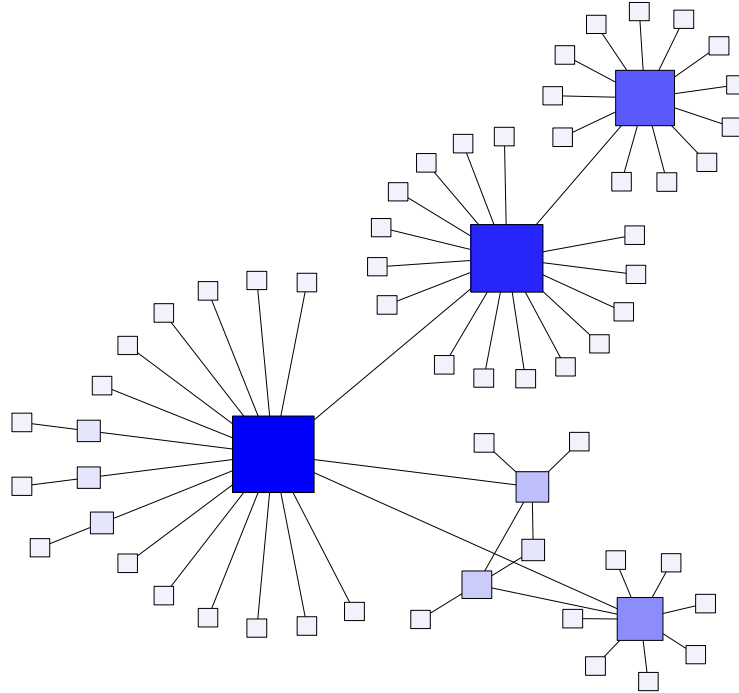


Figure 9: Branching in open-source development (graph obtained through mining of the GCC version history)

Figure 9 illustrates how branches are created in gcc. In a similar way the evolution of gcc can be analyzed with respect to the change propagation, which is often called merging, between branches. In this case a similar picture can be actually expected, in which the centers of the cycles represent the propagation targets (i.e. the branches that receive changes from branches on the perimeter). However the analysis yields another picture, shown in Figure 10.

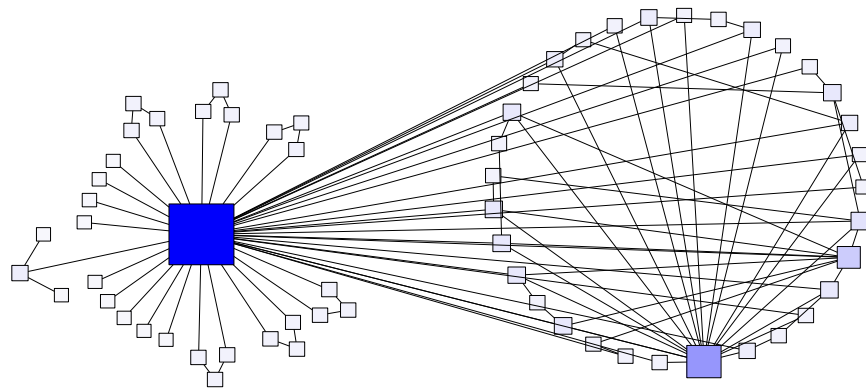


Figure 10: Merging in open source development (graph obtained through mining of the GCC version history)

Figure 10 shows the same branches as Figure 9, however the edges represent aggregated merge operations in this case. In other words multiple merges between the same two branches are depicted with one single edge.

Figure 10 shows a problematic situation than can be often encountered in a product line context. Numerous branches communicate directly with each other. The reason for that lies often in the absence of corresponding coordination mechanisms. And that usually results to reduced awareness about the software development status, to unnecessary redundancies and therefore to significant effort for the joint evolution of a system.

In a product line context branches are often mapped to different members of a product line or – more generally – to different instances of core assets. In this case the significant effort of branch coordination can lead to reduction of reusability. Changes cannot be easily propagated among branches. Possibly, new core asset versions are not exploited, core asset adaptations and bug fixes are duplicated and new core assets candidates are not identified.

Based on the above observations the practical contribution of this thesis is defined as follows.

Practical Contribution: The practical contribution of this thesis is to reduce the effort product line engineers have to spend in order to coordinate activities in a product line engineering process

1.4 Importance of the Problem

While the previous section has shown that the problem of coordinating product line engineering processes entails difficulties, this section will discuss the importance of the problem. In other words it will be shown that this problem can affect a serious amount of software developing organizations.

Coordination of product line engineering processes is particularly necessary when family and application engineering are running in parallel. There may be situations however where this parallelism does not apply. For example in smaller organizations it is possible that there is no explicit application engineering process. In such cases software reuse is restricted to the derivation of product-specific assets. The latter are considered transient products that belong to a product release. In other words the development for reuse process does not include an adaptation step. If an adaptation is necessary it is applied directly on the reusable assets.

However a recent analysis of several industrial product lines has shown that in most cases framework and application engineering are separate processes which run in parallel [LSR07]. If the interactions between these processes are not enforced and properly managed the risk of product line “erosion” grows [YR06]. The latter can be defined as the aggravating situation, in which core asset reuse diminishes while the amount of product-specific assets created in other ways than product line reuse grows. Such a situation can have several negative side-effects including the exponential growth of maintenance effort over time and finally the failure of a product line effort.

The existence of parallel running framework and application engineering processes has been also confirmed through a survey performed in the context of this work [SSF+09]. A total of 17 organizations from industry and academia took part in the survey (see Table 1). Among other things the survey participants were asked to characterize their product line engineering processes by selecting one or more answers out of seven choices. Technology providers and academic institutions typically do not have their own product line engineering processes but rather contribute to corresponding processes of customers. Therefore these participants were asked to characterize the product line engineering processes they contribute to.

Type of organization participating in the survey	Definition	Example	Number of participations in the survey
Industrial Partner	Large organizations delivering safety-critical systems	Aircraft manufacturer	10
Academic Institution	Institution that carries out fundamental research	University	3
Technology Provider	Institution that transfers research results and innovative technologies to industrial application	Solution provider for supply chain management	4
Total			17

Table 1: PLE survey: Participant classification

The following table lists the choices as well as the results in terms of the percentage of the participants that selected each choice.

Answers	Percentage
There is a family engineering activity where reusable assets are developed	65%
There is a series of application engineering activities where individual products are developed	47%
Application engineering includes reuse of assets delivered by family engineering	47%
Application engineering includes development of product-specific assets	41%
Family and application engineering are activities which run in parallel by different groups of stakeholders	35%
Family engineering propagates its changes to application engineering	18%
Application engineering provides feedback to family engineering	29%

Table 2: PLE survey results

As shown in the above table 35% of the participants agreed that family and application engineering run in parallel. From those 5 are industrial organizations and one is technology provider. One of the industrial organizations stated that in some cases there is no explicit family engineering activity but only a set of application engineering activities which run in parallel and synchronize in order to obtain common reusable assets.

Hence both the existing literature and the specialized survey have shown that the parallelism of family and application engineering is a real situation.

1.5 Solution Approach

This section describes the approach followed in this work to address the problems defined in section 1.3.

The 7 steps discussed at the end of section 1.3.5 could be subsumed under a single logical command that consolidates changes between a core asset instance and the original core asset. This is the main idea behind the solution, namely to define and realize a logical interface against which engineers can coordinate product line processes. Two approaches can be considered for the realization of such an interface:

- Defining a new configuration management system that has built-in support for the coordination of product line engineering.
- Define a virtual layer that provides the logical interface needed for product line engineering by encapsulating already existing functionality of a configuration management system.

In response to research question 4 (section 1.3.2), this work opts for the latter approach, namely for the realization of a virtual layer on top of an existing configuration management system. In the following this layer will be called **Customization Layer**. The approach aims at hiding away the complexity of a configuration management system which arises in a product line setting. Figure 11 illustrates the Customization Layer concept.

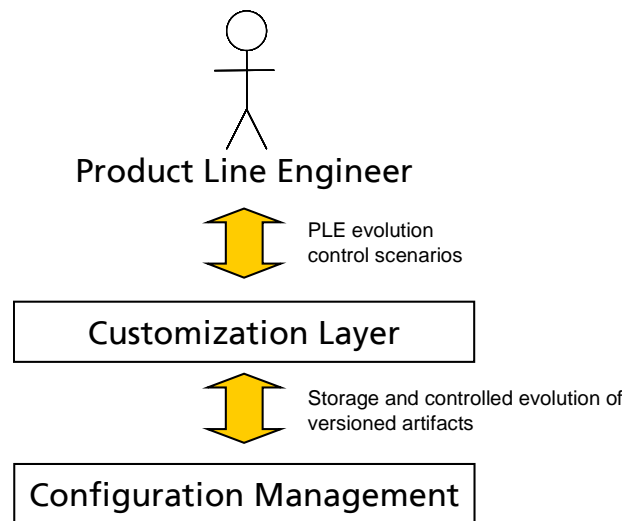


Figure 11: Customization Layer concept

By taking the Customization Layer approach organizations are given the opportunity to incrementally set-up the product line engineering virtual layer. Moreover even when the layer is set-up it can be bypassed at will and the already existing configuration management system can still be used directly.

The Customization Layer approach assumes that existing configuration management functionality is sufficient to realize coordination scenarios in a product line engineering context. As mentioned in section 1.3, configuration management will typically provide the necessary technical functionality. The problem is rather that users cannot cope with the complexity of using the technical functionality in a proper way.

1.6 Methodological Approach

The main objective of this thesis, namely to reduce configuration management effort in a product line, is addressed by a combination of three research methods, the scientific method, the engineering method, and the empirical method [WRH+00]. All of these provide different contributions to achieving the main objective.

The scientific method consists of observing the world and building a model based on the observations. The scientific aspects of the work described in this thesis encompass the investigation of the basic scenarios (section 2), concepts (section 4) and processes (section 5) necessary for product line evolution control.

The engineering method consists of the study of current solutions, as well as the proposal and following evaluation of changes to the current solutions. To this end, current practice and functionality of configuration management is investigated and guidelines are derived (section 6). Furthermore an implementation framework is proposed (section 7) and finally, the steps necessary for the introduction of the Customization Layer approach to an organization are specified in terms of an adoption process (section 8).

The empirical method evaluates a proposed model through empirical studies (section 9). The Customization Layer approach has been evaluated by means experiments, a simulation and a case study.

1.7 Output of this thesis

Main output of this thesis is a method that allows an organization to specify the Customization Layer that it needs. The method consists of four components:

- **Conceptual Model of Evolution Control** (section 3): The model enables an organization to describe the type of product line at hand. The applicable family and application engineering activities can be defined and characterized with respect to evolution control. Furthermore coordination flows can be defined between the activities.
- **Data Model of Evolution Control** (section 4): The model captures the data entities and relations that pertain to evolution control in a product line context. In other words the model describes the different of assets (core assets, instances etc.), their associations and as well as the states that can be taken by these assets.
- **Process Model of Evolution Control** (section 5): The model allows refining the high-level descriptions of the conceptual model. The refinement leads to concrete evolution control operations that are necessary for the given product line and the given coordination needs. The operations use entities of the data model as inputs and outputs.
- **Guidelines for Interaction with Configuration Management** (section 6): Capabilities of the configuration management system at hand are described and these are then mapped to evolution control operations of the process model.

Figure 12 provides an overview of the components that are combined in order to produce a Customization Layer for a given organization. These

components are also brought together in terms of an adoption process in section 8.

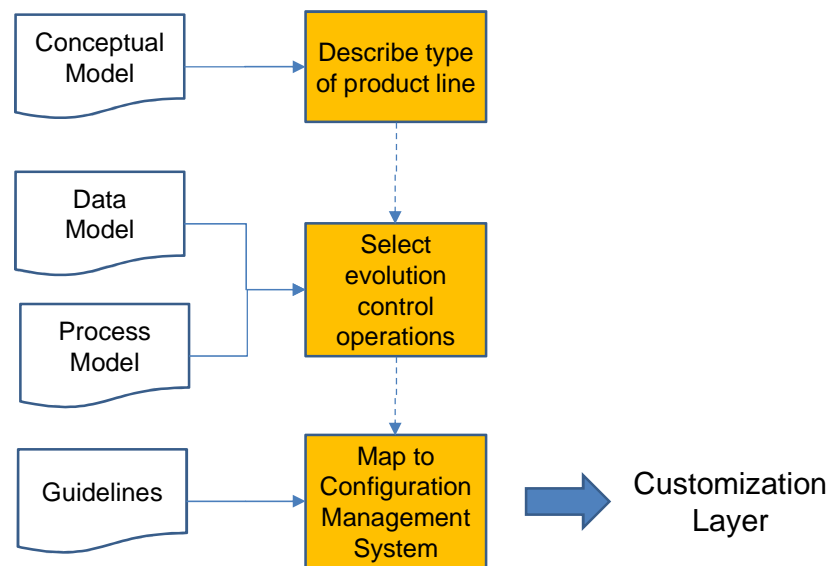


Figure 12: Components leading to a Customization Layer

1.8 Outline of this thesis

The rest of this document is structured as follows.

- **Chapter 2** *Related Work*: The chapter presents existing work that also addresses product line evolution control issues.
- **Chapter 3** *Conceptual Model of Evolution Control*: The chapter introduces the concept and basic scenarios of product line evolution control based on ideas from control theory. Subsequently different types of software product lines and the corresponding evolution control needs are discussed and consolidated in terms of a common conceptual model.
- **Chapter 4** *Data Model of Evolution Control*: This chapter provides a model of the assets pertaining to product line evolution control and defines a semantic bridge to the domain of software configuration management.
- **Chapter 5** *Process Model of Evolution Control*: This chapter refines the conceptual model of evolution control scenarios by allowing the definition of evolution control operations.

- **Chapter 6** *Interaction with Configuration Management*: This chapter defines capabilities of configuration management that can be used for evolution control of a product line. Subsequently the capabilities are mapped to evolution control operations.
- **Chapter 7** *Implementation framework*: This chapter proposes a prototypical framework for the implementation of a Customization Layer.
- **Chapter 8** *Adoption Process*: This chapter brings the different models together by showing how they are used during the introduction of evolution control in a product line.
- **Chapter 9** *Validation*: This chapter presents the two experiments and a simulation study that have been performed.
- **Chapter 10** *Conclusion*: This chapter closes the thesis by summarizing the work, by drawing the necessary conclusions with respect to the research questions and by outlining future work.

1.9 Research Context

This work has been carried out in the context of various research and technology transfer projects at Fraunhofer IESE. Initial results of this work were produced in the context of the BELAMI research project (Bilateral German-Hungarian Research Collaboration on Ambient Intelligence Systems) funded by German Federal Ministry of Education and Research (BMBF), Fraunhofer-Gesellschaft and the Ministry for Science, Education, Research and Culture (MWWFK) of Rhineland-Palatinate, in Germany.

The main part of the work has been performed in the context of the Fraunhofer Innovation Cluster "Digital Commercial Vehicle Technology" funded by the European Union and the state of Rhineland-Palatinate. The majority of the partners participating in the innovation cluster are large organizations facing the problems addressed by this work.

2 Related Work

Goal of this thesis is to support evolution control for software product lines. To this end it is necessary to understand the inherent characteristics of product line evolution control in terms of relevant concepts and scenarios. Subsequently methods, techniques and tools can be developed that address the identified concepts and scenarios. In other words it is necessary to look into product line evolution control from two perspectives [MD08]:

- Analysis of the nature of product line evolution
- Support for the implementation of product line evolution

This chapter presents related work from these areas. In order to evaluate the related work with respect to the research questions defined in section 1.3.2 a set of requirements is derived in Table 3. For simplicity the term solution refers to evolution control solutions for product line engineering in the following.

Research Question	Requirement ID	Requirement description
RQ1 "Granularity"	R1	Solutions have to be aware of different types of assets, in particular core assets, product-specific assets and instances of core assets
	R2	Solutions have to support decomposition of core assets and instances at different granularity levels
RQ2 "Keeping track of reuse"	R3	Solutions have to keep track of core assets reuse cases at different granularity levels
	R4	Solutions have to enable complexity management even with numerous core asset reuse cases
RQ3 "Decay avoidance"	R5	Solutions have to support change propagation (feed-forward) from family to application engineering
	R6	Solutions have to support change propagation (feedback) from application engineering to family engineering

Research Question	Requirement ID	Requirement description
RQ4 "Take advantage of existing configuration management systems"	R7	Solutions have to utilize capabilities of existing configuration management systems.
	R8	Solutions have to be applicable for various types of already existing configuration management systems

Table 3: Requirements to evolution control solutions

2.1 Nature of product line evolution

Research in the area of the nature of software evolution seeks to understand the phenomenon of software evolution in terms of analytic and empirical analyses. The observations are used to specify requirements for further research and also to improve current solutions.

This section reports related work on:

- Conceptual models for product line evolution
- Product line evolution scenarios
- Process dynamics in the context of product lines

Conceptual models discuss concepts and interrelations that pertain to product line evolution. For the present thesis this is relevant since it can be investigated whether current models contain the concepts of core and product assets as well as the corresponding associations.

Product line evolution scenarios analyze stimuli that lead to changes in a product line setting as well as necessary responses by the product line organization. Again this is a relevant field of investigation for the present thesis. It can be examined whether scenarios identified by the community involve continuous coordination between family and application engineering.

Finally, process dynamics use simulation in order to characterize the effects of various types of changes (e.g. organizational, technological) on development processes. For the purpose of the present thesis it can be examined whether process dynamics cover family and application engineering along with their interactions.

2.1.1 A model of system families

The seminal work of Belady and Merlin [BM77] "a model of system families" delivered a model (Figure 13) for the characterization of product lines and the corresponding evolution needs. The basic concept

in this model is the unit, which can be seen as a component in the sense of component-based development [Sz98]. A product line provides a set of units and a product line member, called system in this model, is obtained through configuration of units. In this context a special type of unit is characterized as the basis of the product line and must be included in every system.

Unit: U	Basis of Product Line: U_B
Family: $f = \langle U_B U_1 U_2 \dots U_n \rangle$	Configuration: $c \subseteq f : U_B \in c$
Modules of a Unit: $MOD(U_i) = \langle M_{i1} M_{i2} \dots M_{in} \rangle$ where M_{i1} denotes Module 1 of Unit i	
Modules of a Configuration: $MOD(U_1 U_2 \dots U_n) = MOD(U_1) \cup MOD(U_2) \cup \dots \cup MOD(U_n)$	
Versions of a Unit: $VER(U_i) = \langle \langle M_{i1}(1) M_{i1}(2) \dots M_{i1}(j) \rangle \dots \langle M_{in}(1) M_{in}(2) \dots M_{in}(k) \rangle \rangle$ where $M_{i1}(1)$ denotes Module 1 of Unit i in version 1.	
Versions of a Configuration: $VER(U_1 U_2 \dots U_n) = \langle A_1 A_2 \dots A_n \rangle$ where A_i = union of all versions of the same module	
System S of a Configuration = $\langle B_1 B_2 \dots B_n \rangle$ where B_i selects one version from A_i	

Figure 13:

Model of product lines according to Belady and Merlin

Every unit consists of modules, which in turn may be available in several versions. Therefore unit configuration involves the selection of the appropriate module versions. A challenge at this point is to select the units and versions that yield a valid system. In other words it must be possible to define the set of permitted configurations in a product line. To this end the authors proposed a matrix (Figure 14) which enables the description of units, modules and versions that yield valid systems.

In the model proposed by Belady and Merlin the process of deriving a concrete product is reduced to the selection of already existing assets, i.e. units and their modules. There is no need for a differentiation between different types of assets (R1 is not applicable). On the other hand decomposition of units and modules is not part of the model (R2 is not addressed).

Reuse cases can be tracked by looking into the defined systems (cf. Figure 13), which contain modules and versions that are used in a system (R3 is therefore addressed).

The maintenance of the configuration matrix (Figure 14) can become a complex undertaking in large systems due to the large number of units, modules and versions. Different implementations of this matrix are proposed including mechanisms that facilitate complexity reduction. Requirement R4 is therefore addressed.

Coordination activities between family and application engineering are partially necessary in this type of product line. Application engineering uses core assets as-is and without any modifications. Therefore the main coordination that is necessary is the propagation of new versions to the already defined systems, provided that the consistency of the systems is maintained. Feedback from systems to the development of units is necessary when changes on modules are required. In this case a change request is to be communicated along with the versions in use. The authors provide general guidelines for these situations (R5 and R6 are addressed).

	U ₁	U ₂	U ₃	M ₁	M ₂	M ₃	M ₄	M ₅	
	1	0	0	1	1	1	0	0	module version number
	1	1	0	1	2	1	1	1	
U ₁ U ₂ U ₃ is a permitted configuration and these are permitted systems	1	1	1	1	2	2	1	1	
	1	1	1	1	2	2	1	2	
	0	1	0	0	2	0	1	1	
1=Unit is selected 0=Unit is out	1	0	1	1	1	2	0	0	

Figure 14: Matrix of permitted product line members according to Belady and Merlin

With respect to requirements R7 and R8 the authors give guidelines for the implementation of configuration management. On the other hand there is no discussion about existing configuration management systems and how they can be utilized. There were not much configuration management systems available when this work was done, however some systems existed (e.g. SCSS [Ro75]). For that reason, requirements R7 and R8 will be considered as not addressed.

2.1.2 Variability Specification Language

The Variability Specification Language (VSL) [Be04] provides an XML dialect that enables describing, managing and resolving the variability in product line assets. The conceptual model underlying the language addresses the differentiation between family and application engineering (R1 is addressed). However asset composition is not part of the model (R2 not addressed)

The VSL model introduces the concepts of generic and specific assets, which are the equivalents of the core and product assets discussed in section 1.2. Generic and specific assets are interrelated via derived-from relationships (R3 is addressed). In addition, both types of assets have associations to a hierarchical variability model, which manages the points

of variation in the assets as well as the corresponding instantiations. The hierarchical structure of the variability model enables to tackle complexity when the amount of variations increases (R4 is addressed). Since the focus of VSL lies on variability management the evolution issues are not addressed in the conceptual model.

In the context of VSL the VAMP (Variability Management Platform) reference architecture is also proposed that specifies the capabilities that are necessary for variability management in a product line. The different capabilities are specified in terms of architectural components, interfaces and connections. The Evolver component assumes the responsibility of managing the evolution in the product line and enables change impact analysis, execution and propagation. However the details of the Evolver component were not in the focus of VSL and therefore are not specified in greater detail. As a conclusion, VSL can be considered as partially addressing requirements R5 and R6. Requirements R7 and R8 are not addressed since there is little discussion about the usage of existing configuration management systems.

There are several variability management solutions like the VSL (e.g. [CVL10], [SDN+04], [Mut02], [KCH+98], [PBL05]). In general such solutions focus on managing the decisions that have to be taken upon deriving a product and on facilitating the efficient and correct derivation of product assets. Variability management solutions do not address the issues of evolution and coordination explicitly.

2.1.3 Product Line Evolution Scenarios

Svahnberg and Bosch [SB99] studied the evolution of two industrial product lines from the domains of data storage and mobile communication. The authors observed a series of common scenarios that arose during the evolution of these product lines. The scenarios have been grouped in three interrelated categories, which will be discussed in the following.

- Requirements: In both case studies evolution started with the evolution of requirements. Creation of a new product, improvement of functionality and revision of the execution platform were some of the scenarios that have been encountered.
- Architecture: In order to realize the requirements the authors found out that several architecture evolution scenarios were necessary. For example a new product creation requirement can be fulfilled by introducing a new component in the product line architecture (i.e. generic architecture, common to all product line members). Further architectural scenarios included changing, replacing and splitting

components as well as modifying the relations between components.

- Components: Evolution requirements could be also fulfilled through direct evolution of product line components (i.e. component core assets). The latter were realized in terms of common implementation frameworks. Therefore the evolution scenarios mainly included the creation or modification of components based on the frameworks or the modification of the framework functionality, which was common to all components.

The above scenarios partially address requirement R1 since components can be considered as core assets. There is however little discussion neither about product-specific components nor about the way how components are reused. Requirement R2 is partially addressed since the authors deal with architectural decomposition of reusable assets. Regarding the tracking of reuse cases this work does not make any concrete statements (requirements R3 and R4 not addressed).

Based on the observations, Svahnberg and Bosch propose a set of guidelines for the successful evolution of product lines. The guidelines operate on the level of product line architectures and components. For example it is recommended to avoid the modification of component interfaces either by deferring it when possible or by generalizing the interfaces so that they are more resistant to change. Further recommendations address issues of separation of concerns (i.e. keep a product line intact and introduce new product lines when new application domains are to be addressed; separate common from product-specific behavior) or issues of reusability (i.e. detect and exploit common functionality; rewrite components from scratch when necessary; make design decisions explicitly visible in the architecture). In this regard the work can be considered as implicitly addressing requirements R5 and R6: feedback between family and application engineering is implied as being necessary. Requirements R7 and R8 are not addressed though, since there is only a minor discussion on the role of configuration management in the evolution scenarios.

John McGregor [McG07] derives a set of similar but more generic scenarios for product line evolution. In this case the association between family and application engineering is clearer: Different types of assets are dealt with (requirement R1 is addressed), however asset composition is not discussed (requirement R2 not addressed). The author discusses following evolution scenarios.

- Core assets are extended so that more instances can be derived out of them (i.e. the variation points, namely places in core assets where an adaptation can take place, facilitate the derivation of more instances).

- The addition of product assets in various products may lead to the identification of variations and hence to the creation or modification of core assets.
- Requests for new products and corresponding assets lead to changes in the product line scope.
- A variation (e.g. a decision in a decision model) is split into two variations. This in turn leads to modifications to the corresponding core assets.
- A product line is split into multiple product lines if sufficient new variation is identified.

The above scenarios address tracking reuse cases as well as feed-forward and feedback; hence requirements R3, R5 and R6 are addressed. Furthermore the author proposes recommendations by Mohan and Ramesh [MB06] in order to realize the above scenarios. The first recommendation is to apply modularization on variation points to avoid ripple effects after changes (requirement R4 addressed). The second is to establish traceability between core and product assets and to continuously track the changes on both sides (R3, R5 and R6 again addressed). Finally the variability of core assets must be described in a way that facilitates reuse. In other words, application engineers must be aware of the available core assets, the provided variations as well as the means and effects of resolving the variations.

Configuration management systems are not part of the discussion therefore requirements R7 and R8 are not addressed.

2.1.4 Process dynamics

The field of software process dynamics aims at understanding the phenomena that arise in software development processes. In this context simulation is used to model and observe how development processes operate under different circumstances and also to evaluate the effects of changes – for example modification of contextual factors – on the processes. Cost models are often included in the simulations in order to estimate costs.

Software reuse is considered as one type of development lifecycle, which can be also analyzed in this way. The following figure provides a process dynamics representation of software reuse [Mad08].

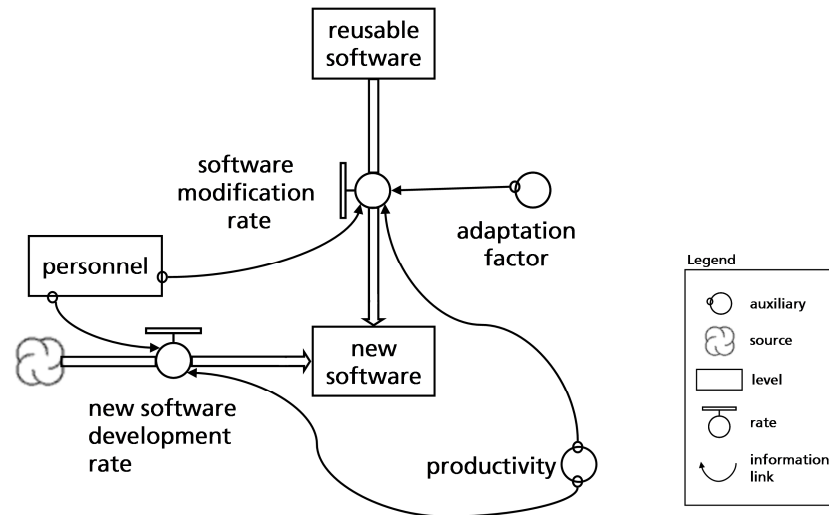


Figure 15: Representation of software reuse with process dynamics

Figure 15 depicts reusable software as a so-called level. The latter can be generally seen as an asset repository. Existing software is being modified, at a particular modification rate, in order to obtain new software. In Figure 15 this connection is modeled in terms of a so-called rate. This can be thought off as a data flow connector between levels.

The rate at which reusable software leads to completed software depends on the effort that is necessary to adapt existing software but also on the productivity of the corresponding personnel. These factors are depicted as so-called auxiliaries that link to the software modification rate. Auxiliaries are typically variables that influence rates in a process dynamics model.

New software is not only produced via reuse of existing software but also through the implementation of new software functionality. This is depicted as the new software development rate in Figure 15. A source element represents the external environment (e.g. customer requirements) that delivers the input for the development of the new software. The rate of new software development depends again on productivity as well as on the appointed personnel.

The above model can be combined with cost models such as COCOMO [BAB+00] or COPLIMO [BBM+04], which explicitly addresses product line engineering, in order to estimate the cost of software reuse in various situations (e.g. using different programming languages or different mechanisms for the implementation of generic components).

The process dynamics model of software reuse, as depicted in Figure 15, focuses on the creation of new software which can be supported through adaptation of existing software. In other words the model deals with different types of software assets (reusable and new software) (R1 is addressed). Decomposition of assets is not discussed however (R2 not addressed).

In the basic model shown in Figure 15 there is no discussion about reuse cases. However another study by [Lo99] refines this model and discusses reuse in different development phases, namely in requirements, architecture, code and also in quality assurance. In this study reuse decay is modeled in terms of components that are not reused any more due to changes of needs. Hence tracking reuse as well as feed-forward from family to application engineering is addressed (R3 and R5 is supported); on the other hand there is no feedback cycle modeled so that decay can be kept under control (R6 not supported). Furthermore the issue of handling numerous reuse cases is not discussed either (R4 not supported).

Also this work does not address configuration management systems; therefore requirements R7 and R8 are not addressed.

2.2 Implementation of product line evolution

Research in the area of the implementation of product line evolution seeks to provide practical solutions in terms of methods, techniques and tools. Among the different research results that have been investigated in the context of the present thesis, the Product Line Asset Manager (PLAM) [BCE+04] showed the biggest relevance and will be discussed in the subsequent subsection.

Another interesting field of related work is naturally the field of configuration management. Some sophisticated systems in this area provide solutions for the management of variant-rich systems.

A series of solutions from the field of software frameworks aims at ensuring the consistency between frameworks and their instantiations. In this regard such solutions are similar to the problem addressed in this work, namely the coordination between development for reuse and development with reuse. Finally, detailed methodological support to evolution control in product line is provided by the Kobra method [ABB+01]. The next sections will hence detail the related work in the aforementioned fields.

2.2.1 Molhado SPL

Molhado SPL [Th12] addresses configuration management for software product lines. It introduces core assets as components that are shared across the members of a product line. The underlying version management system enables managing configuration items along with relationships among them. These relationships are used to associate core assets (i.e. shared components) and instances (i.e. copies of shared components) and also to create composition or other types of relations between assets. Based on that, requirements R2 and R3 are addressed.

Molhado SPL aims at a similar solution as the present thesis. However the following significant differences can be identified:

- Underlying version management: Molhado SPL uses a special-purpose version management system, which is part of the Fluid framework [Boy13]. The authors mention the possibility of implementing the solution on top of conventional systems. However no concrete details are provided. Based on that requirements R7 and R8 are not fulfilled. In this regard it is also unclear if requirement R4 is fulfilled as the employed version management system appears to be a research prototype.
- Scenarios: It is unclear which evolution control scenarios are supported. It seems that the whole issue is delegated to the underlying version management system. There is a discussion on different types of change propagation that are needed; hence requirements R5 and R6 are addressed. However the authors do not elaborate on the implementation with lower-level operations.
- Core assets and instances: Reusable assets are identified through their container, a so-called core asset project. All assets contained in this project are considered as core assets. Product assets are also identified through their respective containers i.e. product line members. Instances are not identified through their type but through their relations to core assets. There is no concrete information on attribute or property-based querying against different types of assets in the product line. Therefore requirement R1 is seen as partially addressed.

2.2.2 Product Line Asset Manager

PLAM aims at the coordination of evolution activities in a product line context. To this end the PLAM tool architecture is proposed, which consists of the following elements

- Roles: PLAM introduces the roles of family and application engineer that develop core and product assets respectively (requirement R1 is

addressed). In addition the roles of programmer and product line engineer are introduced. Programmers are supervised by family or application engineers and are responsible for the implementation of assets. Product line engineers are responsible for the whole product line and coordinate the activities of family and application engineers

- **Architectural Model:** The model contains the conceptual elements supported by PLAM. The basic concept is the component, which is a container of variants, namely in parallel existing versions of assets. The latter can contain further components (requirement R2 is addressed) as well as releases. In the context of PLAM a release is set of configuration items, called objects, in a particular released version. Finally all elements of the architectural model can be associated to actions that are triggered upon execution of evolution operations on the model elements.
- **Architectural Views:** Views enable PLAM users to filter out specific information of the architectural model and to concentrate only the model elements that are relevant in a specific situation. For example an application engineer can use an application view that shows only the assets of a particular product. The view concept of PLAM can hence be used for dealing with complexity in case of numerous core assets and instances (requirement R4 is addressed).
- **Repositories:** PLAM repositories are data storages for product line assets. PLAM provides for different repositories (e.g. for core asset and product development) that can be synchronized.

PLAM specifies a set of operations that can be performed on product line assets and roles. Figure 16 depicts these operations. The instantiation of core assets is addressed by the initiate operations (i.e. the operation refers to the initiation of core asset instances). Change propagation between family and application engineering is achieved with the help of the reconfigure and promote operations (requirements R3, R5 and R6 are addressed). In this context the broadcast and notify operations facilitate the communication between family and application engineers

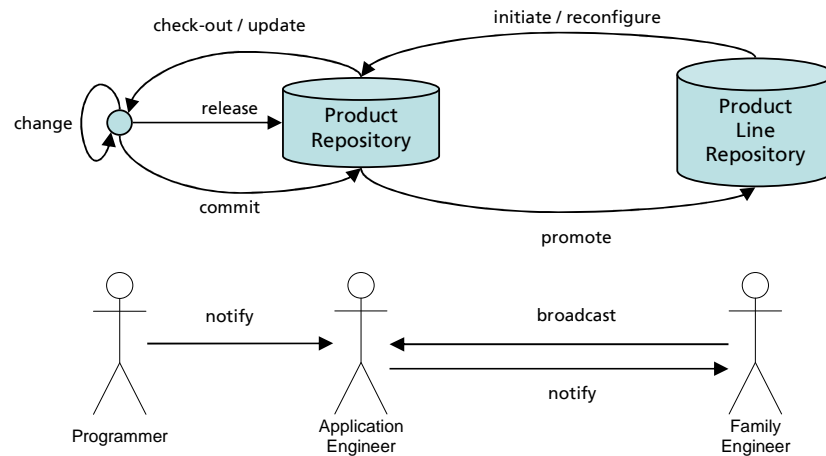


Figure 16: PLAM operations

In general the PLAM approach appears to be in a preliminary stage since most of the concepts are not elucidated in detail. Furthermore, although the authors report that configuration management is used behind the scenes the interface is not discussed any further (requirements R7 and R8 are not addressed).

2.2.3 Configuration Management

Configuration management systems address variability issues with the help of build, version and change management. Build management aims at managing and reproducing the process of compiling (i.e. building) systems based on particular configurations. Version management on the other hand enables keeping track of the evolution of configuration items, performing traceable changes as well as propagating changes. Finally, change management enables the enactment of a systematic evolution control process that entails definition, evaluation, assignment and tracking of change requests. The following sections will discuss build, version and change management in more detail.

Build Management

For the reasons described in sections 1.3.4 and 1.3.5, configuration management literature (e.g. [Mah95], [Whi00]) recommends tackling product line variation through architectural design and build management. Architecture should decompose the product line in a way so that common and product-specific functionality can be clearly separated. Subsequently build management should be setup in a way that enables building (i.e. compiling) particular members of the product line.

```

/cygdrive/c/temp/lkc-1.4
Main Options  VT Options  VT Fonts

anastaso@lap244 /cygdrive/c/temp/lkc-1.4
$ ./conf.exe motor.config
#
# using defaults found in .config
#
* Product Line Configuration
*
* Motor Configuraton
*
Motor
  1. GASOLINE (GASOLINE)
  > 2. DIESEL (DIESEL)
  3. NATURAL_GAS (NATURAL_GAS)
choice[1-3?]: 2
Drive
  1. FRONT_WHEEL (FRONT_WHEEL)
  2. BACK_WHEEL (BACK_WHEEL)
  > 3. 4_WHEEL (4_WHEEL)
choice[1-3]: 2
*
* Diesel Configuration
*
CYLINDER_CAPACITY (CYLINDER_CAPACITY) [12] 2000
Please select a diesel engine
> 1. SDI (SUCTION_DIESEL_DIRECT_INJECTION)
  2. NORMAL_DIESEL (NORMAL_DIESEL)
choice[1-2]: 2

```

Figure 17: Configuration with build management

Build management solutions can provide explicit support for the configuration of systems. Figure 17 provides an example of a configuration front-end, which has been created on the basis of the configuration menu language [URL13] and the make build automation utility [URL14]. In fact the language operates on top of the make utility (requirement R7 is addressed, requirement R8 is not addressed since only make is supported). The front end allows the user to take different decisions for a concrete system configuration. Decisions as well as constraints between decisions (e.g. incompatibility between decisions) are specified with the help of the configuration menu language. When the configuration is finished the make tool and the corresponding compilation process are invoked. Variation points in source code files are managed in terms of pre-processor directives [ISO/IEC 9899], which refer to the decisions specified with the configuration menu language. In this way the compiler generates a running system based on the user's choices.

In this regard build management relates to variability management. The output of build management is usually an executable member of a product line, which usually is not altered any further. Coordination of evolution activities between family and application engineering are therefore not an issue with build management (requirements R5 and R6 are not addressed therefore).

Build management solutions differentiate between source files that go into the compilation process and derived files that result from it. Requirement R1 is implicitly addressed; source and derived files can be seen as core assets and instances respectively. Requirement R2 is also addressed as most build management systems can manage the build process at different granularity levels (e.g. hierarchical make files). Some systems (e.g. the build management facility of the ClearCase system [Whi00]) can also keep track of numerous derived files, therefore requirements R3 and R4 can be considered addressed.

The Shape toolset [ML88] can be seen as an integrated build and version management system. As with traditional build management (e.g. make) it enables building systems. However Shape comes with its own version management system, which is tightly integrated with the build process. However, integration with version management can be also achieved with make or other conventional build management systems like ant [URL18] or maven [URL15].

The unique characteristic of Shape is the provision of a configuration rule language. The latter enables to run queries against the version management system and to select source objects based on their attributes. For example, a rule can be defined that tells Shape to compile only the latest versions of all objects that have passed all tests successfully. Moreover Shape provides explicit support for variations. Different configuration decisions (as in Figure 17) can be described along with constraints (e.g. mutual exclusion). Then, during the build process Shape users can resolve configuration decisions and run the build process accordingly.

In summary, build management solutions can be considered as partially addressing all requirements of Table 3 except R5 and R6.

Version Management

In contrast to build management, version management can coordinate evolution activities. In simpler version management systems this can be accomplished in terms of branching, which is available in different sophistication levels in the various systems available. Branches can be used for variability management in a product line, although this is generally discouraged [Mah95]. As the number of branches increase it becomes difficult to classify the branches and to map them to sensible configurations. This in turn undermines the ability to efficiently and effectively identify, select and jointly evolve configurations. This ability is however indispensable in complex systems and some version management systems provide explicit support in this regard. Four representative examples will be discussed in the following.

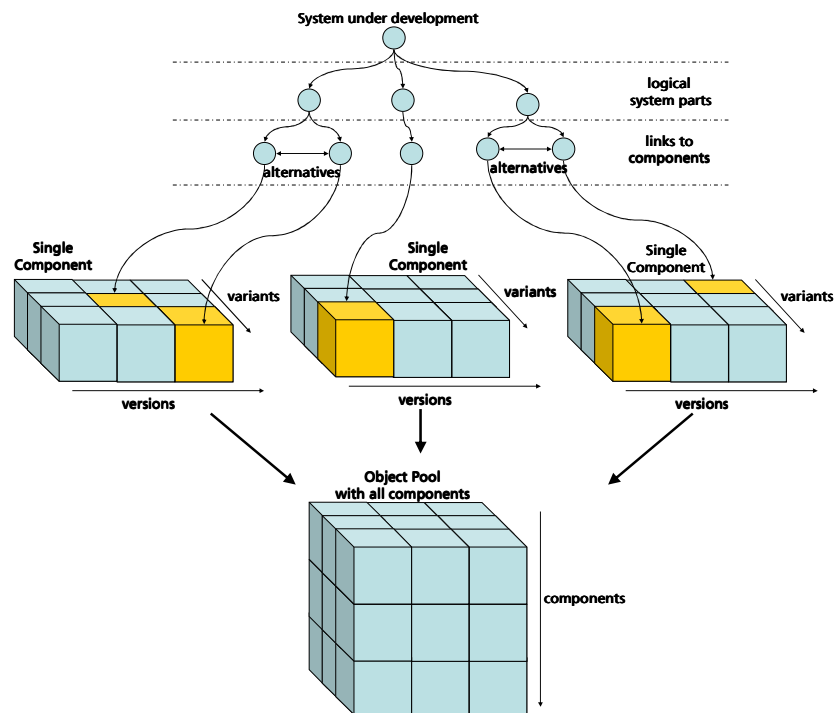


Figure 18: Voodoo version management approach

The Voodoo (Versions of Outdated Documents Organized Orthogonally) versioning management system proposed by [Rei95] stores configuration items in terms of a three dimensional space, called object pool (Figure 18). The first dimension captures the different configuration items, while the second and third dimension capture the versions and branches of the configuration items respectively. Furthermore Voodoo enables to project the object pool to concrete configurations. The projection is accomplished with the help of a logical specification on top of the (physical) object pool. This specification enables mapping configuration items to logical entities, which describe the system under development. The logical entities are grouped together in a tree-based structure that represents the system under development. In a sense this structure resembles a feature tree [KCH+98] that hierarchically structures the mandatory, optional and alternative features of a system. With Voodoo the selection of features leads to the selection of specific versions in the object pool.

The Adele configuration manager [EC95] is a similar solution that facilitates the management of a large number of configurations with the help of logical models. Adele enables the typing and attribution of configuration items. Attributes can be of simple or complex types. In the latter case the type of an attribute is another configuration item. This

provides for the association between configuration items. Configuration items can be versioned and version selection rules enable the definition of configurations. Furthermore, it is possible to assign a set of versions to an attribute. This enables the configuration item containing that attribute to vary.

The ICE system (Incremental Configuration Environment) [Ze97] uses feature logic – a description logic based on feature/value attributions – to manage versions and configurations. A central difference to Adele lies in the way versions are characterized and selected. In Adele a configuration item is characterized by conjunction of its attribute/value pairs (in case of variations, the value of an attribute is whole set of versions). Version selection is then done through Boolean terms over attribute/value terms. ICE on the other hand uses feature logic both for characterization and selection. Moreover, feature logic is the core formalism for the full spectrum of the ICE functionality (e.g. version, repository, workspace management).

While there is no evidence that Voodoo³, Adele and ICE are maintained any further, ClearCase [Whi00] is a modern and popular version management system that also provides similar mechanisms. The concept of a view plays a major role in this context. A view can be compared to a database query that is executed against the version management repository. The query delivers a configuration that is relevant for a particular evolution control scenario. Moreover a view can be related to change management activities, which can be also modeled with the system. In that way the traceability between versions and change requests is facilitated.

Version management systems like Voodoo, Adele, ICE and ClearCase provide powerful facilities for the management of different types of configuration items, numerous variations at different granularity levels, management of associations between configuration items as well as the coordination of activities. In that sense this kind of systems addresses requirement R2 and partially R1 and R3 to R6 of Table 3. Requirement R7 can be also considered as addressed since these systems integrate helpful functionality directly into version management. However requirement R8 is not addressed as the proposed solutions are compatible only with the corresponding version management systems.

Sophisticated version management systems like the above can significantly facilitate evolution control in a product line. However this does not mean that an organization that obtains one of these systems can directly start managing its product line. The necessary evolution

³ A major contribution of Voodoo was also a differencing algorithm, which has been implemented by ClearCase.

control mechanisms have still to be implemented with the available version management functionality. Although the implementation effort is smaller than with simple version management systems, there is usually no explicit guidance available.

For example with ClearCase a product line engineer can easily obtain all assets that belong to a product or to a part of a product, change them and commit them back to the repository. Yet this presupposes that assets are stored accordingly. The process of creating and storing the assets is not given by default and the engineer must ensure that core and product assets are stored in a consistent way. Similarly, feedback loops between family and application engineer have also to be set-up and are not automatically available. On the other hand ClearCase simplifies the creation of such loops through its stream concept: A stream defines the changes that should be propagated to a view. Therefore a stream can be defined for a core asset so that it automatically delivers changes to the products that use the core asset [Le03].

Sophisticated version management systems often provide workflow management support on top of their version management functionality. Adele and ClearCase are examples of such systems. With workflow support it is possible to define evolution control processes in terms of roles, activities, events, data and control flows. In a product line setting it would be for example possible to automate the process of change propagation between family and application engineering. To this end, a process could be defined denoting that when a core asset changes all instances of the core asset have to be updated automatically.

Change Management

Change management enables creating, processing and monitoring change requests, bugs and problem reports in a software development effort. Change requests (also called tickets or issues) can be categorized and interrelated. For example, in a product line context, when a change is necessary on a core asset a corresponding family engineering change request can be created. Since the change to the core asset is likely to influence instances of the core asset, a series of second-level application engineering change requests can be defined as well. The latter are to be processed after the core assets have been changed. Finally, the original change is considered as closed, after all change requests (family and application engineering) have been accomplished. The same mechanism can be also used for managing changes in composed assets: For example a first-level change request can be assigned to a subsystem and second-level change requests can be assigned to subsystem parts.

This change management approach has been described as a ticket hierarchy approach in [UKR09]. Modern change management environments like JIRA [URL9] enable the realization of ticket hierarchies.

Based on the above discussion change management's support for the requirements of Table 3 are elucidated in the following:

- R1: Different types of assets can be addressed by different types of change requests. However change management does not store different types of assets. It can only store change requests associated to assets.
- R2: Asset decomposition can be supported by change request decomposition.
- R3: Reuse can be tracked through the association of family and application engineering change requests. However change requests are transient entities that are closed when a change is accomplished. Therefore special support (e.g. a filtering mechanism) has to be implemented that gives an overview of reuse cases.
- R4: Change management system can deal with numerous change requests and their associations. Categorization, attribution and queries over change requests provide support in this direction.
- R5 and R6: Change propagation is again supported through change request associations. Creation of a change request can lead to the creation of other change request and thus the targets of change propagation can be easily identified.
- R7: Change management is often established on top of existing version management systems so that the actual changes (i.e. new versions) can be related to the corresponding change requests.
- R8: The ticket hierarchy approach can be implemented with different change management systems. Furthermore, version management systems often provide connectors to various change management systems.

In summary, pure change management is considered as partially addressing all requirements of Table 3 since it is possible to solve evolution control problems only from the point of view of change requests. The actual storage, management and evolution of assets are not subject of change management.

2.2.4 The Kobra method

The Kobra approach to component-based development in the context of product lines, described in [ABB+01], proposes a change management process for product line components and their instances. A key idea at this point is to enable a component under configuration management to be self-contained by including its dependencies to other components as first-class entities. In that way the concept of a configuration becomes obsolete since the information about compatibility of component versions are stored within the components. By this means when a set of components are checked-out from a configuration management system, it is possible to perform a consistency check in an automated way. This approach has been also developed by component-based technologies like OSGi [URL16] and Eclipse [URL1] as well as by build automation tools like Maven [URL15].

Kobra allows for the development of core assets, as a special type of components that contain a so-called decision model. The latter contains all configuration decisions that can be taken, when a reusable component is reused. During reuse the decision model is resolved and the result is a so-called resolution model. The latter captures the configuration decisions taken during reuse. Components that are not subject to reuse do not contain decision or resolution models. Therefore Kobra tackles requirements R1, R2 (components can be nested), R3. With respect to R4 Kobra does not provide any guidance. In particular there is no discussion about the implementation of the connection between decision and resolution models or about any scalability issues in case of numerous decisions and resolutions. R4 is therefore considered as not addressed.

Kobra proposes strategies for change management in a product line context. The strategies address requirements R5 and R6.

- Core Asset Change Integration: When a core asset (i.e. a component in the case of Kobra) is changed, the change is propagated to other core assets.
- Application Change Integration: When a product asset (i.e. a product component) is changed, the change is propagated to other product assets.
- Feed-forward Change Reintegration: Core asset changes are propagated to product assets
- Feedback Change Reintegration: Product asset changes are propagated to the original core assets.

KobrA then describes several activities that pertain to the execution of the above four scenarios. This includes concrete conflict resolution guidelines when dealing with such change propagations. For example when a core asset has changed and needs to be synchronized with an instance that has also changed, KobrA proposes to first create a temporary instance. The latter is obtained by applying the last (since the last synchronization) instantiation decisions on the new core asset version. Subsequently the temporary instance can be synchronized with the current instance version at hand. At this point KobrA does not provide guidance for cases, in which already taken instantiation decisions cannot be applied to new versions of core assets.

A similar conflict resolution approach is proposed when instance changes are to be propagated to core assets that have been changed in parallel. In this case it is proposed to first create a temporary core asset that integrates the changes of instances and then to synchronize that temporary core asset with the latest one.

In summary, KobrA provides specific guidance when it comes to the synchronization and conflict resolution between core assets and product assets. Apart from that however, KobrA does not describe how an existing configuration management system can be employed to realize the evolution control activities (requirements R7 and R8 are not addressed).

As a component-based approach KobrA applies mainly to the architecture, design and implementation phases in the development lifecycle. Nevertheless the proposed solutions for variability management with decision models as well as the guidelines for conflict resolution can be applied to virtually any type of asset.

2.2.5 Software Frameworks

Software frameworks can be considered as a special type of software product line. Frameworks usually consist of interfaces, classes as well as of tools and guidelines for the development of particular types of applications. Frameworks contain so-called hotspots, which can be seen as points of variation in the framework contents. Developers are able to extend or customize frameworks at the predefined hotspots according to provided guidelines.

In order to ensure that a framework fulfills its requirements, satisfies its users and constantly evolves, several approaches enable to keep track of the framework usage and to facilitate the application of new framework versions (this tackles requirements R1, R2, R3 and R4). The CatchUp! approach described in [HD05] records the refactoring operations that are performed on a framework. Subsequently the recorded refactorings can

be replayed in the context of an application that uses the framework (requirement R5 is addressed). A special-purpose replay wizard shows detail information of all changes that have been carried out on the framework and allow the application programmer to reason about the effects of the changes and to accept or reject them. However the other way, namely integration of changes back to the framework is not supported (R6 not supported).

The approach in [DMN+06] goes a step further and integrates recorded refactorings with the underlying version management system (requirement R7 and R8 are addressed). That means that refactoring operations can be connected to versions of the framework and become first-class entities of the framework history. Refactoring can be then used in merge operations of the version management system in order to synchronize new framework versions with applications.

It must however be noticed that frameworks operate only at the code level. In a product line context however evolution control spans the whole development lifecycle and hence applies to all kinds of assets produced and consumed therein. Therefore the support of frameworks for the requirements of Table 3 is considered partial.

2.3 Discussion

The analysis of related work in the area of evolution control in a product line setting yields the following categorization which is orthogonal to the categorization used at the beginning of this chapter.

- Conceptual work that does not provide a suitable implementation
- Technical solutions that do not explicitly address the particulars of evolution control in product line engineering.

Table 4 categorizes the presented related work across the two categorization schemes.

Related work presented on the nature of evolution falls into the first category. The model of system families (section 2.1.1) defines the aspects that suite a particular type (not all types) of product line but provides only general guidance for the implementation. The evolution scenarios of section 2.1.3 capture mainly software architecture-related scenarios and again provide general guidance only. The Kobra approach (section 2.2.4) on the other hand explicitly addresses evolution control scenarios for product lines and focuses on change propagation and conflict resolution. However there is no implementation provided on the basis of configuration management. Similar to Kobra, the Product Line

Asset Manager (section 2.2.1) defines a type of workflow management system that explicitly addresses product line scenarios too. However the interface to configuration management is not discussed.

	Nature of Evolution	Implementation of Evolution
Conceptual work	<ul style="list-style-type: none"> • A model of system families [BM77] • Product Line Evolution Scenarios (Svanhberg&Bosch) [SB99] • Product Line Evolution Scenarios (McGregor; Mohan & Ramesh) [McG07] [MB06] 	<ul style="list-style-type: none"> • Product Line Asset Manager [BCE+04] • The Kobra method [ABB+01]
Technical solution	<ul style="list-style-type: none"> • Variability Specification Language [Be04] • Process dynamics [Mad08] [Lo99] 	<ul style="list-style-type: none"> • Molhado SPL [Th12] • Build Management [Whi00], [URL13], [URL14], [ML88] [URL18] [URL15] • Version Management [Whi00], [Rei95], [EC95], [Ze97] • Change Management [UKR09] • Software Frameworks [HD05], [DMN+06]

Table 4: Categorization of evolution control related work

The Variability Specification Language (section 2.1.2) lies between the two categories introduced above. It provides a conceptual model that explicitly addresses product line issues as well as an implementation of the model. However the evolution aspect is neither sufficiently addressed in the model nor in the provided implementation.

Process dynamics (section 2.1.4) provide technical means of analyzing evolution control processes in terms of a special methodology that entails modeling and simulation. However current models cover only basic software reuse and do not include configuration management.

In the category of technical solutions, configuration management (section 2.2.3) is of major importance. Build management addresses most requirements but does not address change propagation. It is therefore applicable only to situations, in which no coordination between family and application engineering is necessary.

Requirements: R1: different types of assets; R2: decomposition; R3: keep track of reuse; R4: handle complexity; R5: feed-forward; R6: feedback; R7: utilize CM; R8: support different CMS								
Related Work	R1	R2	R3	R4	R5	R6	R7	R8
A model of system families [BM77]	✓	✗	✓	✓	✓	✓	✗	✗
Variability Specification Language [Be04]	✓	✗	✓	✓	?	?	✗	✗
Product Line Evolution Scenarios (Svanhberg&Bosch) [SB99]	?	?	✗	✗	?	?	✗	✗
Product Line Evolution Scenarios (McGregor; Mohan & Ramesh) [McG07] [MB06]	✓	✗	✓	✓	✓	✓	✗	✗
Process dynamics [Mad08][Lo99]	✓	✗	✓	✗	✓	✗	✗	✗
Molhado SPL [Th12]	?	✓	✓	?	✓	✓	✗	✗
Product Line Asset Manager [BCE+04]	✓	✓	✓	✓	✓	✓	✗	✗
Build Management [Whi00], [URL13], [URL14], [ML88] [URL18] [URL15]	?	✓	✓	✓	✗	✗	✓	✓
Version Management [Whi00], [Rei95], [EC95], [Ze97]	?	✓	?	?	?	?	✓	✗
Change Management [UKR09]	?	?	?	?	?	?	?	?
Software Frameworks [HD05], [DMN+06]	?	?	?	?	?	✗	?	?
The Kobra method [ABB+01]	✓	✓	✓	✗	✓	✓	✗	✗
Legend ✓: supported; ✗: not supported; ? : implicitly/partially supported								

Table 5:

Related work characterization

Sophisticated version management addresses all requirements but the solutions provided are applicable only to specific version management systems. Change management solutions on the other hand address all requirements including the flexibility requirement. Yet, change management concentrates on change requests only and does not

address evolution of assets. Furthermore, the concepts used in version and change management do not correspond to product line concepts directly.

Table 5 summarizes related work and the corresponding fulfillment of requirements derived in the beginning of this chapter.

2.4 Section summary

This chapter has presented related work in the field of evolution control for product lines. Existing results have been analyzed with respect to the type of evolution aspects they address as well as the research questions pertaining to the present thesis.

The next chapter will present the first component of the solution proposed in the present thesis, namely a model that explicitly captures the concepts of product line evolution control.

3 Conceptual Model of Evolution Control

Focus of this thesis is evolution control in the context of product line engineering. This section refines the concept of product line evolution control and elaborates on the basic scenarios that must be supported. Furthermore this section describes different types of product line engineering settings and the corresponding evolution control needs.

Controlling the evolution of assets produced in an engineering process requires that changes are continuously monitored as well as evaluated and that correcting actions take place when necessary. In other words control generally requires three basic capabilities: (a) monitoring, (b) evaluating and (c) correcting. An established area of applied mathematics that deals with such issues is control theory [AM08]. Hence, control theory will be used as foundation in the following in order to systematically obtain the basic scenarios (or scenario categories) for product line evolution control.

3.1 Introduction to control theory and feedback

Control theory provides all mathematical constructs necessary for the creation of systems that can be continuously controlled. A central concept in control theory is the concept of dynamical system. The latter is defined as a system whose behavior changes over time in response to stimulation. In that sense a product line can be seen as a dynamical system. As discussed in section 1 a product line consists of several related software products that are continually changed over time. Changes can be stimulated externally by customers, users or the overall environment in which the software runs or internally, that is within the software developing organization. According to the discussions in section 1.2 a product line can be actually seen as the aggregation of two dynamical systems, namely family and application engineering.

A further central concept of control theory is the concept of feedback. It is defined as the connection between dynamical systems in terms of influence. In other words the output of a dynamical system can influence another dynamical system which in turn can influence other systems (including the first one). The concept of closed loop is often used in this context to illustrate circular dependencies between dynamical systems as shown in Figure 19.

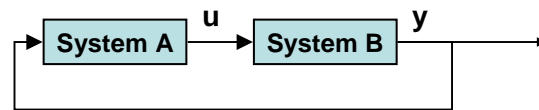


Figure 19: Closed loop feedback

In the above example *System A* produces an input u which influences *System B*. The latter then produces output y which in turn influences back *System A* and possibly other systems not shown in the figure.

3.2 Feedback in product line engineering

Based on the above discussion a product line engineering process can be seen as a dynamical system comprising two sub-systems, family and application engineering as depicted in the following figure.

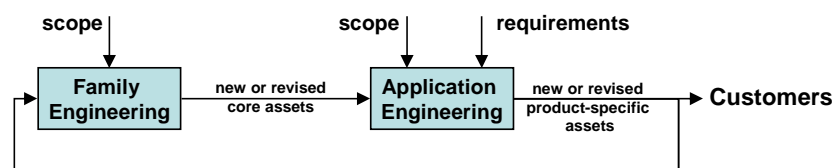


Figure 20: Closed loop in Product Line Engineering

Figure 20 illustrates Family and Application Engineering as two series connected dynamical systems. For simplicity it is assumed that there is only one single application engineering process. Family engineering initially receives one input; that is the scope of the product line. The latter defines the common and varying characteristics of product line members and it also provides information about different technical domains and the corresponding software reuse potential for the product line at hand. The scope plays a major role in family engineering since it helps deciding which core assets are made reusable and to what extent. The outputs of family engineering are new or revised core assets that are to be reused across the product line during application engineering.

Application engineering takes three inputs: (a) core assets delivered by family engineering, (b) the scope of the product line and (c) product-specific requirements. The latter are naturally the major driver of application engineering. Since application engineering delivers products (depicted as a set of product-specific assets in Figure 20) an interaction must take place between the application engineering section (i.e. the sales department) and the corresponding customers. During this interaction product-specific requirements are obtained which are possibly realized during application engineering. The decision about which requirements to realize is facilitated by the product line scope.

Based on the scope definition the customer service representatives are in position to offer different product features to customers. They are also in position to judge whether a customer wish is supported by the product line. After customer negotiations application engineers start producing the specific product line member, partially by reusing the core assets from family engineering. The result is a set of new or revised product-specific assets which on the one hand are packaged as products and delivered to customers and on the other hand are communicated to family engineering in terms of feedback.

Feedback is necessary since it enables family engineering to analyze the evolution of product line members and the extent to which software reuse is applied. The feedback enables identifying and avoiding redundant effort in different product line members. It facilitates keeping the core asset base up-to-date so that the advantages of software reuse are continuously exploited. This kind of feedback is also necessary in order to modify the product line scope according to customer needs.

3.3 Introduction to control loops

In order to control a dynamical system it is necessary to provide the three basic capabilities of monitoring, evaluating and correcting. In control theory monitoring is taken over by sensors, evaluating by controllers and corrections by the system under control. In other words a controlled dynamical system requires a closed-loop feedback system with three subsystems as shown in the following figure.

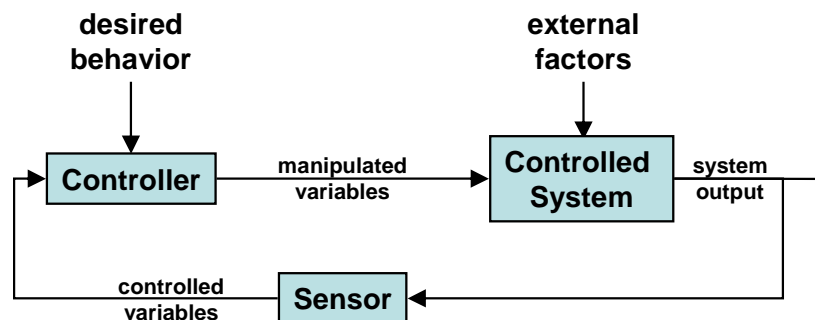


Figure 21:

Control loop

Figure 21 depicts a dynamical system under control, a sensor that monitors the system output and a controller that compares the system output with the desired behavior before taking any correcting actions. The inputs to the controller are called controlled variables [RP05]. These are the data of interest captured by the sensor that monitors the system output. The outputs of the controller are called manipulated variables

[RP05] because they are given as input to the controlled system attempting to change its behavior. The final system output is hence a function of the manipulated variables and external factors that influence the system.

A typical example of a system that works as illustrated in Figure 21 is a home automation system. Such a system needs to be controlled so that specific control variables like temperature do not exceed the desired thresholds. External factors that influence a home automation system can be user actions (e.g. operating the windows, heating) or environmental factors (e.g. outside temperature). Manipulated variables can be instructions from a controlling computer to turn off the heating or to lower the shutters. Finally sensors (e.g. temperature or humidity sensors) perform measurements and assign data to the controlled variables

3.4 Control loops and configuration management

In order to study evolution control in a product line engineering process the control loop concepts discussed in section 3.3 can be applied. In fact using control loops for the study of software engineering dynamics in general has already been proposed in the literature [Mad08].

Configuration management is the de-facto established discipline for evolution control in software and system development. Therefore, before looking into the particularities of product line engineering, it makes sense to first map the concepts of controller, system and sensor to corresponding concepts from the field of configuration management.

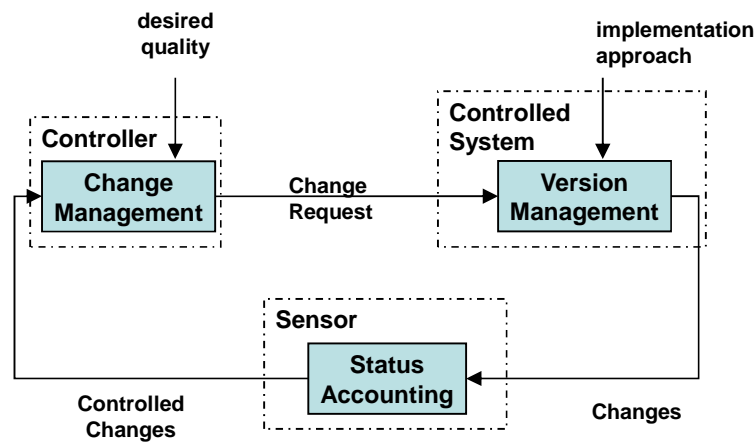


Figure 22: Mapping control loop concepts to configuration management

The corresponding concepts in configuration management are change management, version management and status accounting. The following subsections will discuss these concepts in the context of the evolution control loop.

3.4.1 Change management

In Figure 22 the controller is mapped to the activity of change management. This activity is responsible for the evaluation of changes in terms of impact to system functionality, interfaces, utility, cost, schedule and contractual requirements [Le04]. All these factors are shown as the desired quality input in Figure 22 since they influence the quality of the system under development but also the quality of the development process per se.

The output of change management is usually one or more change requests which describe items affected by changes, the nature of changes, proposals on how to approach the changes and further information about cost or schedule⁴.

3.4.2 Version management

After change management has specified the necessary changes, version management realizes them. This activity performs the actual changes and in terms of control theory it corresponds to a dynamical system that is to be controlled.

The implementation approach is a factor that influences the quality of new versions. This factor is an uncontrolled variable and can depend on the profile and experience of the engineer that performs a change or on other organizational, political or societal factors. Output of version management is a set of new versions, which contain the change requested by change management.

3.4.3 Status accounting

Configuration status accounting (CSA) consists of the recording and reporting of information needed to effectively manage a software system and its characteristics [Le04]. In other words status accounting enables to query the configuration management repository for all different kinds of information, particularly changes. Therefore it corresponds to the sensor concept in control theory.

⁴ All these parts of a change request are often captured in terms of templates, which are then also given as an input to the change management activity.

Status accounting can be setup as a filter that evaluates and selects information about new versions against predefined criteria. Examples of such criteria are the type of changes (i.e. versions), the owners of changes or the time when changes occurred.

3.5 Control loops in product line engineering

A product line engineering process consists of parallel running family and application engineering activities (see also sections 3.2 and 1.2). In the simplest case there are one family engineering activity that delivers core assets and a set of application engineering activities that deliver product line members by reusing core assets.

Each activity (family or application engineering) is subject to evolution control. Therefore the control loop shown in Figure 22 applies to each of the activities. This yields a set of parallel running control loops. Following figure illustrates these loops in a product line engineering setting containing one family and three application engineering processes.

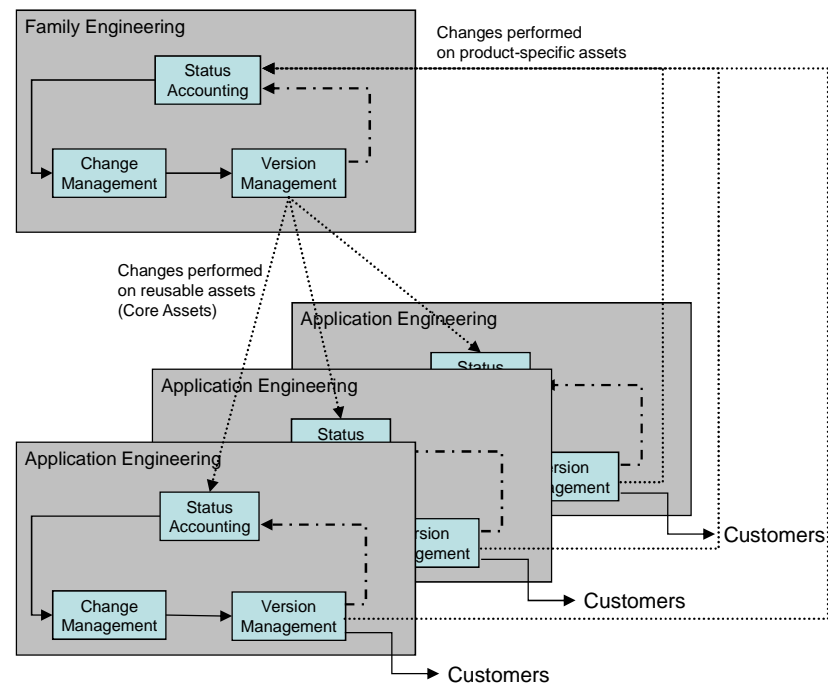


Figure 23: Control loops in product line engineering

Figure 23 illustrates internal and external feedback connections between activities. Internal feedback is applied within a family or application engineering process, while external feedback represents data flow across different processes. As shown in the figure, version management in

family engineering provides feedback to status accounting in application engineering. This makes sense since new versions of core assets have to be communicated to the users of core assets, namely the corresponding application engineering activities. Status accounting can capture this information. Similarly, new versions of product assets can be interest to family engineering; therefore there are connections between version management in application engineering and status account in family engineering.

3.6 Types of product lines

Figure 23 has shown an example of a product line type, in which one family engineering activity interacts with three application engineering activities through feedback between the corresponding version management and status accounting activities. However there are other types of product lines, in which the situation looks differently.

It is for example conceivable that a change management activity in family engineering directly interacts with a change management activity in application engineering [UKR09]. Similarly, a version management activity in family engineering may involve version management activities in application engineering: A change in a core asset may automatically lead to an update in the corresponding application engineering activities.

There is a series of different product line types and according product line engineering processes identified thus far in the community [Mut02][Bo02]. The following sections will elucidate these different types as well as the corresponding evolution control needs.

3.6.1 Individual (or independent) Products

It is possible that an organization delivers custom created products only. Each time a customer requires a product it is built from scratch or via ad-hoc software reuse. In such a situation all individual products are maintained independently of each other and feedback is performed in an opportunistic manner.

In general development of an individual product can be characterized as product-specific. There may be however cases in which the development for reuse appears sensible for some assets. Therefore the development process as well as the corresponding asset base can be seen as hybrid, combining aspects from both family and application engineering. The decision to make an individual product asset reusable is typically made in an opportunistic way. The same applies to the application of reusable assets as well as to feedback relationships between different individual products.

3.6.2 Product Generations

A product generation is defined as a set of related versions of a singular product that have been released by an organization in a specific time period (e.g. releases 3.1 and 3.5 of the popular Firefox browser). In some cases organizations opt for the continuous maintenance of the various generations. A product line can hence be defined in which the product generations can be seen as the product line members.

During the parallel development of different product generations backward and possibly forward compatibility must be usually guaranteed. Therefore feedback and control mechanisms must be established. On the other hand software reuse is not necessarily part of the development process and is applied in ad-hoc manner.

3.6.3 Standard Application

Standard applications often arise as a natural evolution of individual product development. The ad-hoc application of software reuse may gradually lead to a set of assets that make up a standard product or standard application delivered by the organization. This application will usually capture a unique selling point or a major competence field of an organization (e.g. complex process simulations). Hence every time a customer requires a product the standard application is individually extended to address the needs of the customer.

In terms of evolution control the common core captured by the standard application is maintained in parallel to the individual instances of the applications that are delivered to customers. Feedback loops are necessary to communicate possible problems with the customization or execution of the standard application and also for the delivery of new versions of the standard application. The feedback loops can be internal or external depending on the size of the standard application and the organization.

Software reuse is generally applied in an opportunistic way. The common core of the standard application is generally developed independently of any reuse potential. On the other hand experiences from the instantiation of the common core may lead to development of some reusable assets.

3.6.4 Professional or Customizable Application

This type is similar to the standard application. The difference lies in the range of features and properties offered by the application. In this case the professional application aims at comprising as much functionality as

possible in an application domain while the standard application focuses on the most important functionality.

To reduce the tailoring effort a professional application must be made customizable. There are different possibilities and maturity levels to this customizability. Typically a professional application will provide for its customization during installation or at run-time. For example configuration dialogs may come into play that will let the user select the functionality to be executed. More advance customizability may involve the adaptation of the application even in an earlier stage (for example during compilation or linking).

In any case the customized instances of a professional application are not maintained any longer after their delivery to customers. Therefore feedback control, if applied, is usually internal.

3.6.5 Standardized Infrastructure

A standardized infrastructure has also many similarities to a standard application. Actually a standardized infrastructure may comprise a standard application along with necessary infrastructure for production of product line members. Such an infrastructure typically consists of an operating system, an integrated development environment and possibly a set of external (commercial or open source) systems, tools, libraries and frameworks. The evolution control needs are similar to the needs of the standardized application.

3.6.6 Platform

The platform follows the same idea as the standardized infrastructure. The difference here is that a platform (or a framework) is developed that captures the commonality in the product line. Product line members are hence built by reusing and extending the platform at well-defined places, also known as hotspots [FSJ99]. In other words a platform also addresses the variability (cf. Figure 2, section 1.3) in a product line but does not provide a detailed variability specification. Both the platform and the derived products need to be maintained in parallel. Feedback from the products to the platform is possibly supported [DMN+06] [HD05].

3.6.7 Product Population

This type is similar to the platform type introduced above. Here the platform is realized as a collection of components in the sense of component-based development [Sz98]. In other words components have well defined incoming, outgoing and configuration interfaces.

Components may be also nested but usually there are no predefined compositions of components. Product line members are therefore created through the composition (possibly including configuration and adaptation) of preexisting components. The different components are maintained individually. The same applies to the different component compositions. This approach has been introduced in [Om02].

3.6.8 Software Product Line

This is that traditional setting that corresponds to the discussions in sections 1.2 and 3.2. A Family Engineering activity is established that delivers reusable assets based on a strategic plan. The latter is the result of the scoping activity. In addition a set of Application Engineering activities are set-up for each of the members of the product line. Therein assets that make up the final products are being developed by reusing available core assets and by creating product-specific assets. Family and Application Engineering activities are executed in parallel and must be continuously synchronized in order to avoid erosion of the core asset base.

3.6.9 Hierarchical Product Lines

Hierarchical product lines [TH03] extend the idea of the basic software product line with additional framework and application engineering activities. This can become necessary in bigger organizations that involve different internal or external units (i.e. through subcontracting). In case for example an organization delivers a set of large-scale systems that consists of various subsystems, each subsystem may follow a product line approach. In the family engineering phase the subsystem will provide for the different variations the subsystem may be influenced from. Such a family engineering phase can be then embedded in the family engineering phase of the enclosing system. On the other hand a subsystem-specific application engineering process will enable the instantiation of a subsystem during the instantiation of the enclosing system and therefore can be also embedded in the enclosing application engineering phase. Hierarchical product lines can be also employed in cases where the product portfolio of an organization is large so that products are assigned to different product categories, possibly at different granularity levels. In this case a product category can be seen as the result of application engineering, which however needs to be further refined in order to become a product. In other words product categories are created in hybrid processes that have elements of both family and application engineering.

3.6.10 Production Lines

Production lines have been introduced in [Kr02] and are similar to the typical software product lines regarding family engineering. Production lines also contain a family engineering activity that creates flexible and reusable assets. Furthermore family engineering is equipped with special infrastructure for the explicit management of the variability. The different options, alternatives and parameters that must be decided on during the creation of a product as well as their interdependencies are managed explicitly in this variability management infrastructure.

However production lines consider the result of the instantiation process, which is the set of assets that make up a product line member, as a transient work product. Therefore product assets are not part of the product line and consequently there is also no feedback loops usually planned from the products back to the product line.

In that sense production lines can be compared to professional or customizable applications (cf. 3.6.4). The major difference is the presence of the variability management infrastructure.

3.6.11 Adaptive Product

An adaptive product can be seen as specialization of a professional application. Adaptive products are able to dynamically change according to changing conditions in the execution environment. The changes in this case can go further than with customizable products. In the case of adaptive products new code can be integrated or existing code can be reflectively adapted during execution. The concrete mode of adaptation may vary but ideally an adaptive system is able to sense the environment and trigger a self-adaptation accordingly. Given a set of adaptive systems running in a field feedback cycles can come into play in order to inform an organization about the status of the running systems. Thereby the types of changes from the field and the triggered self-adaptation operations can be communicated.

3.7 Conceptual model

In order to support evolution control in different types of product lines a generalized conceptual model is necessary. A basic model is depicted in Figure 24 in terms of UML notation [UML03]. At this time, connections between processes (i.e. feedback loops) are not modeled; they will be discussed as the model is refined in later sections.

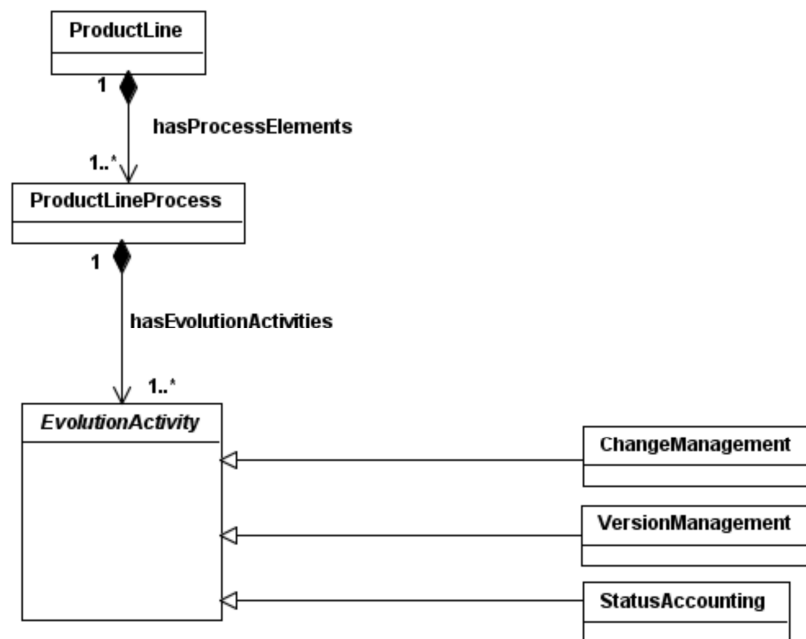


Figure 24: Conceptual model of evolution control

Figure 25 below shows the same model as in Figure 24 in terms of a textual representation, which will be used in the remainder of this work for the description of structural models. This representation enables the creation of a Domain-specific Language (DSL) for product line evolution control and is based on the Xtext language development framework [URL19].

```

ProductLine:
    name=ID
    (processElements+=ProductLineProcess)+;

ProductLineProcess:
    name=ID
    (evolutionActivity+=EvolutionActivity)+;

EvolutionActivity:
    ChangeManagement |
    VersionManagement |
    StatusAccounting;
  
```

Figure 25: Conceptual model (Xtext-based)

Xtext provides for the creation of textual DSLs as context-free grammars. Hence, the different concepts (e.g. product line etc.) that appear as classes in the UML model are specified as grammar rules (e.g.

ProductLine followed by a colon) with Xtext. UML class attributes in turn are mapped to assignments (e.g. name=ID) within rules. The left part of an assignment represents an attribute or feature (e.g. a product line has a name) and the right part of the assignment can be seen as a type (i.e. ID is a type that classifies all uniquely identified attributes). Multiplicities are handled in terms of Extended Backus-Naur expressions (i.e. a '+' refers to the one-or-more multiplicity etc.)

Compared to the UML-based representation a DSL based on Xtext offers a serious advantage: it enables to exactly address the concepts needed for the specification of product line evolution control. Xtext takes a grammar as an input and generates a specialized textual editor, an editor for the description of product line evolution control. The editor focuses only on the necessary concepts and facilitates significantly (e.g. through auto-completion, syntax highlighting etc.) the description. On the contrary a UML-based solution imposes the usage of the UML meta-model, which is extensive and does not necessarily correspond to the specification problem at hand. There are two further advantages of a DSL-based solution, which will be explained in the following subsections.

3.7.1 Validation of instance models

The UML model in Figure 24 can be thought off as a meta-model that captures various types of product lines. A concrete instance of that model would describe a particular type of product line. In UML such a model instance can be realized in terms of an object diagram or in terms of another class diagram (for that it would be necessary to turn the model of Figure 24 into a UML profile). Yet, in both cases and with current UML tools it is difficult to control the creation of instance models. As a result invalid instance models that do not adhere to the meta-model can arise. The Object Constraint Language [OCL01] can support the validation of such models; however the creation of OCL scripts is an error-prone task that requires considerable effort. Moreover, different tools implement the OCL specification differently. Finally, some tools enable the creation of custom validation programs (e.g. through development of specialized validation plug-ins), which have full access to the UML model and can check against various constraints. The development of such validator programs is again not an easy task and requires deep knowledge of the UML tool at hand. Moreover, such validators are tool-specific and depend on the particular tool version as well as on the UML version at hand.

On the other hand Xtext facilitates significantly the creation of validators as it automatically generates two types of validators out of a grammar definition. These validators can be invoked continuously, that is while a grammar instance is edited or separately, through selection of the corresponding menu item for validation.

- Java-based validator: Xtext generates Java classes representing the rules of the grammar as well as a template for a validator class. The latter allows full programmatic access to the grammar elements. Therefore, it is possible to implement a broad spectrum of validation rules, which can be applied to instances of the grammar. Moreover, Xtext runs as a plug-in for the Eclipse development environment [URL1]. Hence, various features of Eclipse come into play and facilitate the development activities significantly.
- Check-based validation: Xtext also generates templates for validation scripts based on the Check language [oAw]. The latter is a declarative constraint language that significantly facilitates the validation of grammar instances.

For example, the initial model shown in Figure 25 has the constraint that an evolution process cannot have two identical evolution activities (e.g. a family engineering process cannot have two identical change management processes). This constraint has been implemented in terms of the Java-based validator.

3.7.2 Code Generation

Xtext provides facilities for the creation of code generators out of a grammar definition. It actually generates a code generator project for Eclipse that can be extended at will. The project uses the specialized Xpand template language [oAw]. The latter enables the creation of code templates that can be filled in with input from grammar instances. In so doing, it is possible to generate parts of a Customization Layer based on the specified evolution control needs.

3.8 Refined Conceptual Model

In order to refine the basic model shown in Figure 25 the following refinement or aggregation levels will be considered:

- Processes: Enables to specify the contents of a product line in terms of family engineering, application engineering or hybrid processes (i.e. a combination of family and application engineering)
- Activities: Enables to specify the contents of a process in terms of change management, version management and status accounting activities
- Scenarios: Enables to specify the contents of an activity in terms of scenarios to be supported.

This chapter will address the first two refinement levels. The third refinement level, i.e. the scenarios, will be covered in detail in chapter 5.

Figure 26 provides the first level of refinement. A product line process is modeled as an abstract concept, which can be instantiated as family engineering, application engineering or hybrid process. The latter can arise in situations, in which an activity develops for reuse and with reuse at the same time.

```

ProductLine:
    name=ID
    (processElements+=ProductLineProcess)+;

ProductLineProcess:
    FamilyEngineeringProcess |
    ApplicationEngineeringProcess |
    HybridProcess;

FamilyEngineeringProcess:
    'FamilyEngineering' name=ID
    (evolutionActivity+=EvolutionActivityFE)+;

ApplicationEngineeringProcess:
    'ApplicationEngineering' name=ID
    (evolutionActivity+=EvolutionActivityAE)+;

HybridProcess:
    'Hybrid' name=ID
    (evolutionActivity+=EvolutionActivityFE)+
    (evolutionActivity+=EvolutionActivityAE)+;

```

Figure 26: Refined conceptual model (Processes)

As discussed in section 3.7 each product line process (i.e. family engineering, application engineering or hybrid) has one or more evolution activities. The content of these activities, that is the scenarios to be supported, can differ depending on the type of the activity. In other words a family engineering process has other change management, version management and status accounting activities than an application engineering process.

Hence, in the next step family engineering, application engineering and hybrid process are refined in order to enable the specification of the corresponding evolution activities. Each activity (i.e. change management, version management or status accounting) will be further refined later (chapter 5) in terms of scenarios to be supported.

```

FamilyEngineeringProcess:
  'FamilyEngineering' name=ID
  (evolutionActivity+=EvolutionActivityFE)+;

EvolutionActivityFE:
  ChangeManagementFE |
  VersionManagementFE |
  StatusAccountingFE;

ChangeManagementFE:
  'ChangeManagementFE' name=ID
  (scenario+=ChangeManagementScenarioFE)+;

VersionManagementFE:
  'VersionManagementFE' name=ID
  (scenario+= VersionManagementScenarioFE)+;

StatusAccountingFE:
  'StatusAccountingFE' name=ID
  (scenario+= StatusAccountingScenarioFE)+;

```

Figure 27: Refined conceptual model (Family Engineering)

```

ApplicationEngineeringProcess:
  'ApplicationEngineering' name=ID
  (evolutionActivity+=EvolutionActivityAE)+;

EvolutionActivityAE:
  ChangeManagementAE |
  VersionManagementAE |
  StatusAccountingAE;

ChangeManagementAE:
  'ChangeManagementAE' name=ID
  (scenario+=ChangeManagementScenarioAE)+;

VersionManagementAE:
  'VersionManagementAE' name=ID
  (scenario+= VersionManagementScenarioAE)+;

StatusAccountingAE:
  'StatusAccountingAE' name=ID
  (scenario+= StatusAccountingScenarioAE)+;

```

Figure 28: Refined conceptual model (Application Engineering)

3.9 Role of the conceptual model

The conceptual model can be used to describe a particular type of product line as well as the corresponding evolution control needs in a semi-formal way in terms of a domain-specific language.

In the context of this thesis the description of a product line is the first step towards setting-up an evolution control system, namely a Customization Layer (cf. section 1.5). In that sense the conceptual model can be first used to describe the set of processes (i.e. family and application engineering) that have to be controlled as well as the evolution control activities therein. This is the first step towards the derivation of appropriate structure for the Customization Layer as well as the access to the underlying configuration management system. The next step, which will be elucidated in chapter 5, is to specify the scenarios to support in each activity and then to map this to configuration management (chapter 6).

Figure 29 illustrates an example usage of the conceptual model⁵. The grammar instance corresponds to the library example introduced in section 1.3.5. The example product line delivers a reusable library of vectors. Two product line processes are modeled: *VectorDevelopment* as instance of *FamilyEngineering* and *vectorApplication* as instance of *ApplicationEngineering*.

```

ProductLine VectorProductLine{
    FamilyEngineering VectorDevelopment{
        ChangeManagementFE VectorDevelopmentCM,
        //{
        //scenarios to be added later
        //}
        VersionManagementFE VectorDevelopmentVM,
        StatusAccountingFE VectorDevelopmentSA
    }
    ApplicationEngineering VectorApplication{
        ChangeManagementAE VectorApplicationCM,
        VersionManagementAE ProductPopulationVM,
        StatusAccountingAE ProductPopulationSA
    }
}

```

Figure 29: Example application of the conceptual model

Since a Customization Layer will possibly be employed within a distributed team persistence of the basic asset model have to be considered carefully. A strategy to cope with that is to consider entities of the data as transient. In this case the model instantiated in memory from the underlying CMS every time it is needed. If however there is a continuous integration facility available it is possible to continuously update the model based on the feedback from the continuous

⁵ The example uses some syntactical extensions (e.g. commas, brackets), which can be modeled with an Xtext grammar. For simplicity reasons these extensions will not be shown in the text. The full grammar including the syntactical extensions is given in the appendix.

integration server. In this case however a database management solution should be considered in order to cope with the concurrent access by multiple users.

3.10 Section summary

In this chapter concepts of control theory such as dynamical system, control loop and feedback have been mapped to product line engineering. In so doing, evolution control activities in a product line context have been identified. Then a model has been elaborated that enables to specify these activities. The model has been derived based on the control theory concepts as well as on the observation of different types of product lines. Instantiation of the model is the first step towards the creation of a Customization Layer for a product line. In the next section the basic data model that underlies the Customization Layer will be discussed.

4 Data Model of Evolution Control

This chapter introduces a data model for the assets pertaining to product line evolution control. Subsequently a mapping is presented between product line evolution control and configuration management that closes the semantic gap between the two areas.

Variability management is a crucial component of product line engineering. It enables the description and management of the variability that is inherent in every product line effort. Therefore this section also discusses the relations between product line evolution control and variability management.

4.1 Basic Asset Model

As discussed in sections 1.2 and 1.3.4 there are two basic types of assets that appear in a product line engineering context:

- Core assets
- Product assets

Core assets represent the output of family engineering and are considered as reusable assets. On the other hand product assets are the output of application engineering and may be core asset instances (i.e. the result of reusing core assets) or product-specific assets that were not developed with planned reuse. Figure 30 formalizes the different types of assets in terms of Xtext grammar.

```
Asset:
    CoreAsset | ProductAsset;

ProductAsset:
    CoreAssetInstance | SpecificAsset;

ChangeRequest:
    name=ID
    item=ID
    state="open" | "closed" |
        "inprogress" | "fixed" | "approved" |
        "reviewed" | "verified"
    assets+=[Asset]*;
```

Figure 30: Types of assets and change requests

In this case Xtext is not meant to be used directly by product line engineers in order to describe assets. Xtext is used to elucidate entities and relations, on which a Customization Layer operates. It is the responsibility of a Customization Layer to instantiate these entities and relations based on the evolution control operations (or scenarios) that will be invoked.

The usage of Xtext for the specification of the basic asset model enables the generation of all necessary Java classes that correspond to the entities of the model. This enables a Customization Layer to programmatically access instances of the model during execution.

Change requests can be assigned both to core and product assets. Therefore Figure 30 also introduces the concept of a change request. Each request has a unique name and a unique item identifier. The latter refers to the persistent location that physically contains the change request. Such a location can be described in terms of a uniform resource identifier (URI) of a change management system. Furthermore a change request contains references⁶ to assets. Finally, a change request moves across various states during its lifecycle. Figure 30 uses the states proposed by the OSLC specification [OSLC10].

4.1.1 Core Assets

A core asset can be uniquely identified by its name as well as by its configuration item, i.e. file or directory. A core asset refers to a family engineering process from the conceptual model (see section 3.7). This reference aims at specifying the process that produces the core asset at hand.

Furthermore a core asset may refer to a reuse contract [Me99]. The latter formally describes the way the core asset is to be reused. The reuse contract may also establish a connection to the product line scope and describe the products for which the core asset is planned for. Finally, a core asset can refer to zero or more instantiation objects, which describe how the asset is being reused. Figure 31 provides the grammar extract for core assets.

⁶ References in Xtext grammars are expressed in terms of squared brackets, e.g. [CoreAsset] represents a reference to a core asset object, which is described in a grammar instance.

```

CoreAsset:
  name=ID
  item=ID
  process=[ConceptualModel::
            FamilyEngineeringProcess]
  (reuseContract=[ReuseContract])?
  instantiations+=[Instantiation]*;

```

Figure 31: Core assets

4.1.2 Instances and product-specifics

A core asset instance as well as a product-specific asset is also identified in terms of a name and a configuration item. And, similarly to core assets there are references to application engineering processes from the conceptual model. An instance also contains references to one or more instantiation objects. The latter relate core assets with instances and arise when a core asset is reused. As counterpart to reuse contracts an instantiation object refers to signed contracts. The latter describe the decisions that have been taken while reusing a core asset. Figure 32 provides the grammar extract for instances and product-specifics.

As shown in the grammar, an instantiation can relate many core assets to many instances. There are three main cases to be considered, which can be also combined with each other:

- Multiple core asset instantiation
- Core asset composition
- Core asset decomposition

The first case is conceivable if there are many possibilities for the derivation of a specific instance out of a core asset. This can happen for example if the instantiation is a configuration procedure. In this case an instance is obtained by configuring a core asset and it is possible that different configuration decisions result to the same instance. The sets of the different final configurations can so be captured in terms of instantiation objects.

```

CoreAssetInstance:
    name=ID
    item=ID
    process=[ConceptualModel::
        ApplicationEngineeringProcess]
    instantiations+=[Instantiation]+;

SpecificAsset:
    name=ID
    item=ID
    process=[ConceptualModel::
        ApplicationEngineeringProcess];

Instantiation:
    name=ID
    coreAsset+=[CoreAsset]+
    signedContract+=[SignedContract]
    instance+=[CoreAssetInstance]+;

SignedContract:
    name=ID
    item=ID
    reuseContract=[ReuseContract];

```

Figure 32: Instance and product-specific assets

The second case may come into play if an instance is the result of combining multiple core assets. There may be situations where core assets can be used in products only in combination with other core assets. For example a core asset may contain a placeholder that needs to be filled by another core asset. Hence during instantiation the application engineer has to decide with which core asset to fill the placeholder. The result of the instantiation is a single entity, an instance asset, that resulted from the instantiation of two core assets, namely of the core asset that contained the placeholder and the core asset the filled-up the placeholder.

In the third case one or more core assets are combined to produce many instances. This can happen for example if a particular instance is required many times. This can also happen if a core asset reuse requires decomposition of the core asset. In this case the instantiation output is a set of instances obtained from the decomposition of the core asset. Table 6 provides a graphical overview over the three types of instantiations.

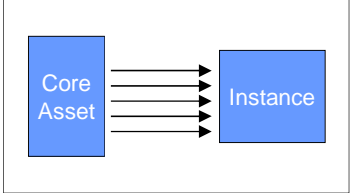
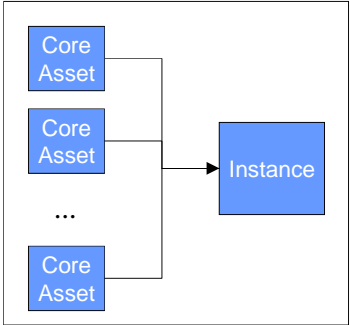
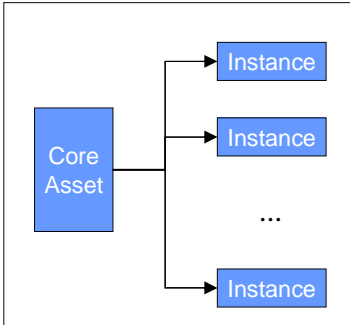
<div><p>Multiple instantiation</p></div>	<p>The same instance can be produced in different ways out of the same core asset</p>
<div><p>Core Asset Composition</p></div>	<p>Many core assets are composed to produce one instance</p>
<div><p>Core Asset Decomposition</p></div>	<p>One core asset is decomposed to produce many instances</p>

Table 6: Types of core asset instantiation

4.2 Asset State Model

This section will discuss the different states that apply to core assets and instances. This discussion is useful as it allows understanding the fundamental behavior of core assets and instances. Based on this discussion and especially based on the identified triggers behind state transitions evolution control scenarios can be derived (in chapter 5). UML state charts will be used in the following as notation.

4.2.1 Core asset states

The state of a core asset can be broken down to four concurrent regions:

- Integration
- Release
- Reuse
- Change Management

The following sections will discuss these regions in detail.

Core Asset Integration

The region Integration, shown in Figure 33, monitors the state of the asset with respect to the synchronization with its instances. To avoid product line erosion it is important to ensure that new versions of instances are reflected in the corresponding core assets. Upon core asset creation the NonIntegrated state is by default entered. When the integrate operation executes it propagates changes from instances back to their core asset. Therefore this moves the core asset to the Integrated state. This indicates that the core asset has taken relevant modifications from application engineering into account.

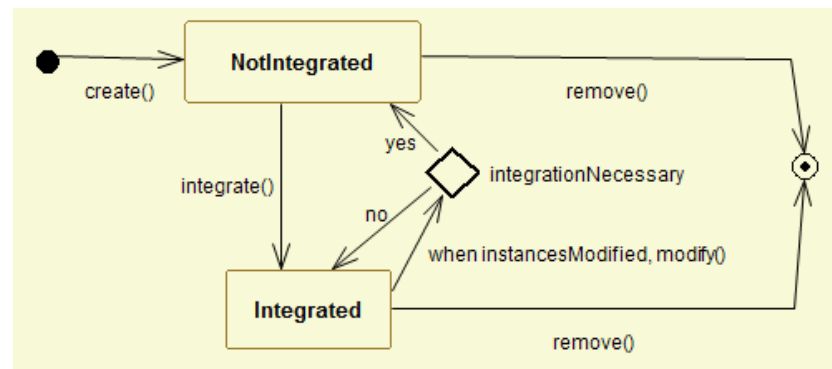


Figure 33: Core Asset integration

Core Asset Release

The second region Release (Figure 34) monitors the process of releasing an asset. The idea is that a core asset must first be released before it can be instantiated. Again, upon core asset creation the NotReleased state is entered. A series of modification may return to this state until the release

operation is executed. In this case the Released state is entered. A subsequent modification leads back to the NotReleased state. The asset must be explicitly released so that modifications become available for propagation to the instances.

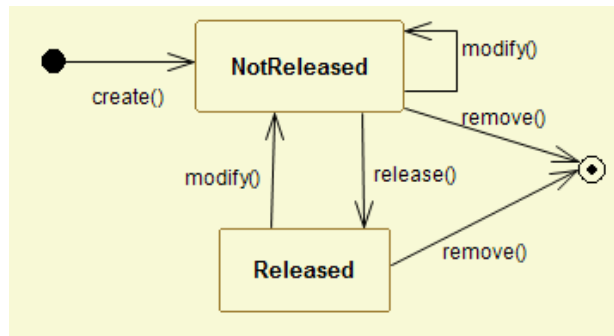


Figure 34: Core Asset Release

Core Asset Reuse

The region Reuse (Figure 35) aims at monitoring the reuse of a core asset. Upon core asset creation the state NotReused is entered. When a core asset is instantiated the region moves to the Reused state. Modification of the core asset either leads to the NotReused state, if the modifications removed all instances, or back to the reused state. Figure 35 also shows `instanceModified` as a change event. This indicates that the transition from the Reused state to the `instancesLeft` choice can also be initiated by a change on instances from the side of application engineering.

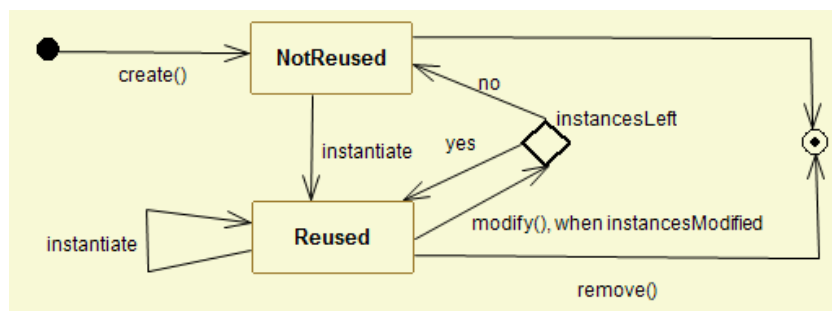


Figure 35: Core Asset reuse monitor

Core Asset Change Management

Finally the last region ChangeManagement aims at monitoring change requests related to the core asset at hand. When a core asset is first

initialized it enters the state `NoChangesPending`. Upon creation of a new change request the `ChangesPending` state is entered. When a change request is closed the core asset goes back to the `ChangesPending` state except if there are no more change requests open, in which case the `NoChangesPending` state is reentered.

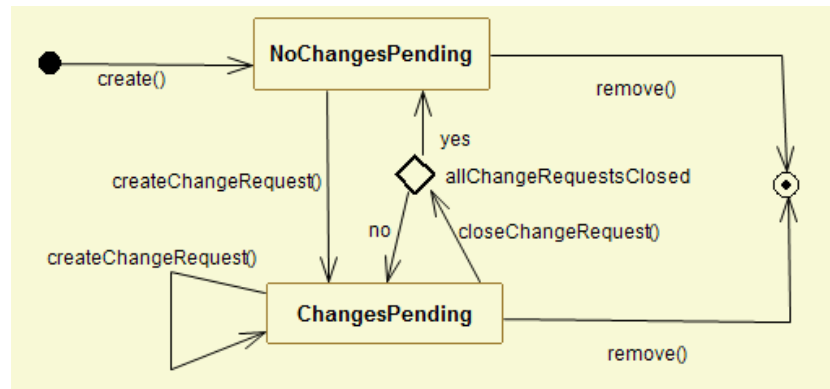


Figure 36: Core Asset change management

Correctness Constraints

At this point there are two correctness constraints that can be formulated, which ensure that undesirable effects will never be obtained:

- `always(`
`coreAsset.release_state == CoreAssetStates.RELEASED =>`
`eventually(coreAsset.reuse_state == CoreAssetStates.REUSED)`
`)`
- `always(`
`((coreAsset.reuse_state == CoreAssetStates.NotReused) AND`
`coreAsset.integration_state == CoreAssetStates.INTEGRATED))`
`== FALSE`
`)`

The first constraint indicates that if a core asset gets instantiated its reuse state will be eventually be set to *Reused*. The second constraint states that it is not possible to have core asset which is not reused but integrated at the same time. A tool as the Customization Layer is responsible for the satisfaction of these constraints.

4.2.2 Product asset states

As with core assets, states of product assets can be broken down to the following regions:

- Instance
- Rebase
- Change Management

Instance and Rebase will be explained in the following. The Change Management region is identical to the corresponding core asset region presented in section 4.2.1 and therefore will not be discussed here again.

Instance

As described previously, a product asset can be the result of core asset reuse, in which case it is an instance of a core asset, or it can be a product-specific development obtained through new development or other kinds of reuse.

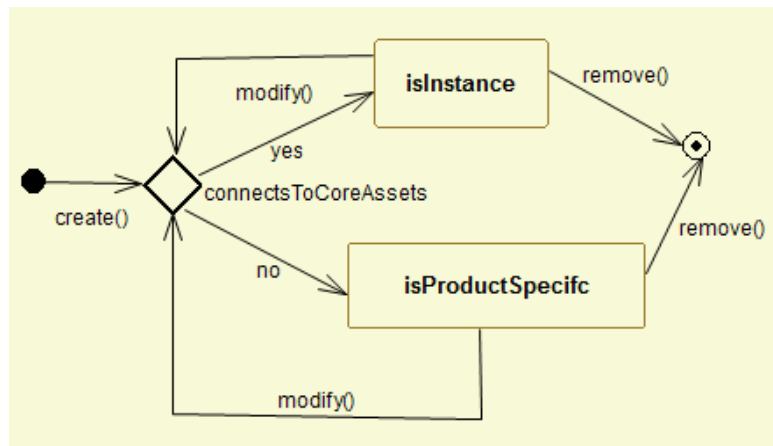


Figure 37: Product asset characterization

Therefore, the first region Instance contains two states `isInstance` and `isProductSpecific`, which denote whether a product asset is an instance or not. The condition `connectsToCoreAssets` is checked upon entering the region and upon modification. The purpose of the condition is to identify whether the given product asset relates to core assets.

Rebase

The Rebase region aims at monitoring whether a product asset, in particular an instance asset, is in sync with the core asset it originates from. This state is the equivalent of the Integrated state in the core asset state chart (section 4.2.1). When the core asset an instance originates from changes (i.e. a new release is available) it is important to consider updating the instance with the latest changes from the core asset. To this end the operation rebase of the Customization Layer can be called.

In this region the first state to be entered depends whether the asset is product-specific or not. In the former case the state Irrelevant is entered. Otherwise it is checked if changes from core assets have to be synchronized and the state is set accordingly. Changes on instances or core assets are monitored afterwards in order to continuously update the state of the Rebase region.

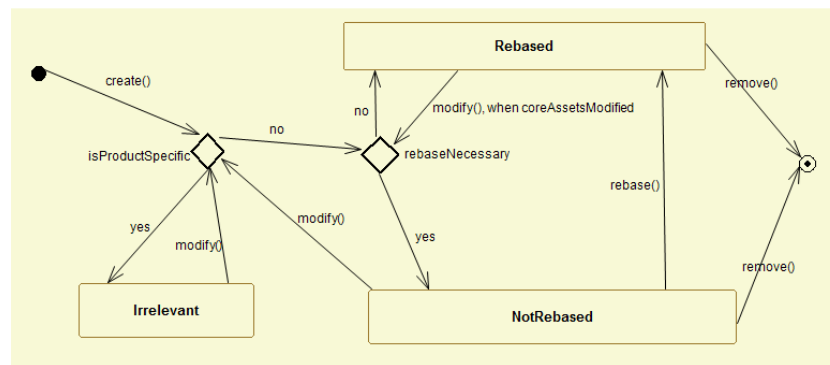


Figure 38: Rebasing instances

Correctness constraints

There are three correctness constraints that can be formulated for instances at this point. A simplified version of the domain-specific language proposed in [BH11] is used in the following. The language enables the expression of temporal logic constraints in a functional way in terms of the Scala programming language [Od10].

In the first constraint for example the set of core assets that relate to an instance are obtained by accessing the *instantiations* attribute of the instance. By accessing the *coreAssets* attribute of *instantiations* the corresponding core assets are obtained. Finally each core asset in the set is referred to by the variable *c* that can be used to check the integration state.

- ```
always(
 (instance.instance_state == InstanceStates.REBASED)
 implies(
 eventually(instance.
 instantiations.
 coreAssets.filter
 (c => c.integration_state ==
 InstanceStates.INTEGRATED).
 isEmpty == false)
)
)
)
```
- ```
always(
  (coreAsset.integration_state ==
    CoreAssetStates.INTEGRATED)
  implies(
    eventually(coreAsset.
      instantiations.
        instances.filter
          (i => i.instance_state ==
            InstanceStates.REBASED).
            isEmpty == false)
    )
  )
)
```
- ```
always(
 ((instance.instance_state ==
 InstanceStates.ISPRODUCTSPECIFIC)
 AND
 (instance.rebase_state == InstanceStates.REBASED))
 == FALSE
)
)
```

Hence, the first constraint indicates that if an instance gets rebased its corresponding core assets should eventually have the integrated state. Similarly the second constraint ensures that if a core asset is integrated its corresponding instances shall eventually be rebased.

Finally the last constraint ensures that a product-specific asset will never enter the *rebased* state as it has no connection to core assets

### 4.2.3 Identified operations

Based on the state model discussed in sections 4.2.1 and 4.2.2 a set of operations on core and product assets can be identified as shown in the simplified class diagram of Figure 39.

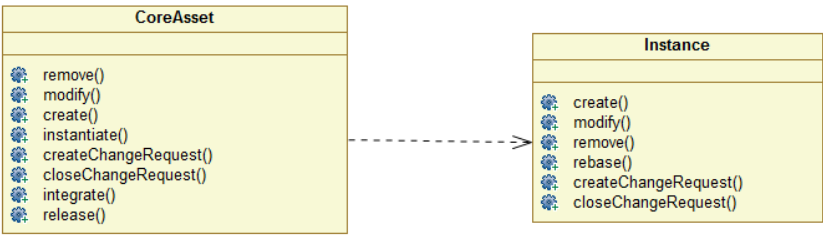


Figure 39: Operations identified through state models

Moreover two change events `instancesModified`, `coreAssetsModified` as well as conditions `isProductSpecific`, `rebaseNecessary`, `integrationNecessary`, `connectsToCoreAssets` and `allChangeRequestsClosed` have been identified.

This functionality is to be provided by a Customization Layer implementation along with its interaction with a configuration management system as it will be shown in sections 5 and 6.

### 4.3 Role of the Basic Asset Model

Assets in the basic asset model are to be considered as logical entities that may refer to physical entities. In the context of configuration management, which is the major enabler towards product line evolution control, physical entities are configuration items (cf. section 1.3.3). In order to close the gap between logical assets and physical configuration items a relationship has been defined between asset and configuration items. This relation is expressed in terms of the *items* attribute that is assigned to assets (see for example Figure 31).

The major idea of this thesis as initially discussed in section 1.5 is that an infrastructure – the Customization Layer – is set on top of configuration management that enables evolution control in terms of product line concepts, that is to say core assets and product assets. On the other hand the underlying configuration management deals with the physical configuration items that embody the logical assets. Following picture illustrates this setting.

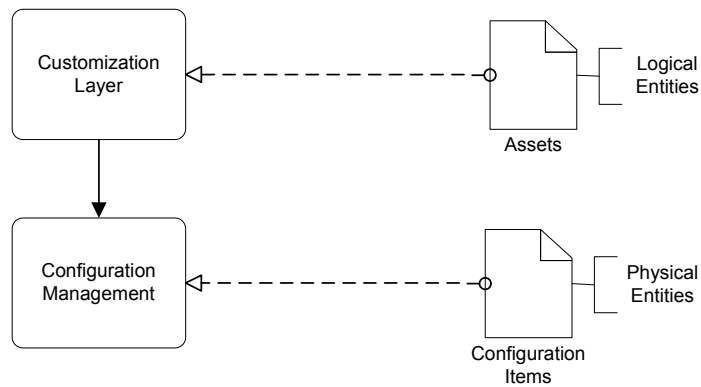


Figure 40: Role of the asset model in the Customization Layer

The basic asset model defines therefore logical concepts the Customization Layer operates on. Furthermore the state models provide a specification of the externally visible states of assets managed by the Customization Layer.

The implementation of the basic asset model entails an important parameter, the model update mode. The update mode specifies how the logical model is synchronized with the configuration management repository. Two modes are possible:

- **Dynamic update mode:** Create the necessary logical entities dynamically, every time a Customization Layer operation is called. The entities are created based on information from the repository. For example when the users invokes the operation to list all core assets, the Customization Layers looks for core assets in the corresponding repository location, collects all information necessary and delivers the result in terms of logical entities.
- **Offline update mode:** Maintain the model as a separate item and keep it up-to-date when needed, based on information from the repository. The Customization Layer holds the model as a configuration item or in terms of another technology (e.g. as a database) and checks the synchronization status every time a model operation is called.

## 4.4 Variability Management

Variability management is a central capability in every product line engineering effort. As discussed initially in section 1.3 a product line must be able to manage the variability dimension, which is not present in single system development. Variability is produced during family engineering. In fact it is an inherent characteristic of core assets that

enables them to be used in various situations, namely in the context of various products. In other words variability is a central component of reusability.

In order to become reusable core assets contain different variation points. According to [Be04] a variation point is defined as a place in a core asset, at which an adaptation can be carried out. The set of variation points in a core asset can so be subsumed as the variability of the core asset. Variability management entails the description of the core asset variability and also the description of the dependencies between variation points. Such a description shows the decisions that can be taken to adapt a core asset as well as the effects of these decisions. This description is referred to as variability model in the following.

Reuse contracts discussed in section 4.1.1 correspond to variability models. Yet reuse contracts, as a concept, are more abstract and can take various forms. Variability models can thus be seen as a concrete incarnation of reuse contracts. Variability models can be created for single core assets but they can also be composed to create variability models for whole product lines.

Variation points can have interdependencies. It can happen that an adaptation performed on a variation point requires the adaptation of another variation point. In the collection library example (section 1.3.5) the array and the vector may vary with respect to the types of objects to be held in the collection. In other words the array and the vector can be developed as parameterized classes in which the type of objects to be collected is a parameter. The parameter is the variation point in this case which might be connected to other variation points in the collection. A vector may provide for example a method that sums the elements in the vector. The concrete sum algorithm depends on the type of objects in the collection. When the vector is set to manage integers, the sum algorithm will calculate the arithmetical sum of the integers. If the vector is set to manage strings, the algorithm may perform the concatenation of the contained strings. The sum algorithm is therefore another variation point which relates to the class parameter variation point.

In the above example two variation points (i.e. class parameter and algorithm) are related to two adaptation decisions, namely the possibility to adapt the object type and the possibility to adapt the sum algorithm. These two possibilities can be naturally merged to a single possibility, namely the decision upon the type of objects to be used across the collection. Resolving the higher decision on the type to be used across the class automatically resolves the lower level decisions. Such a composition of decisions is another major responsibility of variability management. In other words variability management enables the creation of adaptation decision classifications based on part-of relations

(i.e. partonomies). The highest-level decision that can be defined in that way is the product line member to be delivered. On the other the lowest-level decision will adapt a single entity within a core asset. In addition to composing decisions towards higher-level decisions variability management also manages implications between decisions. In the above example composition automatically included implication (e.g. selecting an integer in the high-level decision for the whole class implies selecting an integer for the type of objects to be held in the collection and an integer for the sum algorithm). However this must be not always the case. In other words there may be implications between decisions that do not belong to the same part according to the decision partonomy.

Variability management does not only describe the adaptation possibilities and effects of all core assets; it also acts as a configuration frontend to this adaptation. As such it enables application engineers to select and to actuate an adaptation possibility. The goal of the actuation is to carry out the adaptation on the respective core assets and to obtain the corresponding core asset instances. To this end a variability management infrastructure has to be connected with the environment that is used for the development of core assets. In case of source code core assets such an environment is the integrated development environment (IDE), in case of architectural core assets it can be a modeling tool and in case of requirements it can be a requirements management system. In this regard variability management must also keep track of variability configurations. A variability configuration is a set of resolved decisions and corresponds to signed contracts as discussed in section 4.1.2.

Finally variability management can provide functionality that assures the quality of a variability model. For example a variability model with erroneous implications between decisions can lead to a constraint satisfaction problem and therefore to no valid core asset instance or to no valid product. Variability management can enable the discovery of such problems.

Following list summarizes the functionality that must be provided by a variability management infrastructure:

- Description of possible adaptation decisions and effects
- Classification of adaptation decisions based on part-of relations
- Management of implications between decisions
- Actuating the adaptation
- Keeping record of already performed adaptations
- Quality assurance of variability models



Different approaches to variability modeling and management have been proposed in the literature each one addressing some of the above issues. Following table provides a list of well-known approaches.

| Approach                                            | Reference |
|-----------------------------------------------------|-----------|
| Common Variability Model<br>(upcoming OMG standard) | [CVL10]   |
| COVAMOF                                             | [SDN+04]  |
| Decision Modeling                                   | [Mut02]   |
| Feature Modeling                                    | [KCH+98]  |
| Orthogonal Variability Model                        | [PBL05]   |
| Variability Specification Language                  | [Be04]    |

Table 7: Variability management approaches

In addition to the above approaches there are also commercial and open-source tools that support variability management. The underlying conceptual models are however not published in the most cases. The decision modeling approach for example has been implemented in terms of the Decision Modeler tool [YFM+08]. Other well-known tools include GEARS [URL12], pure::variants [URL7] or the free PLUM [URL11].

#### 4.4.1 Connection to Basic Asset Model

The basic asset model can be connected to variability management approaches and the respective underlying models. The advantage of the connection is two-fold.

While the asset model focuses on the relation between core assets and instances variability management technologies focus on the relation between core assets and variability decisions. With variability management a domain space is typically built-up that specifies the decisions an application engineer has to make when deriving a product or parts of a product. Decisions in the domain space are mapped to logic that actuates and delivers the assets accordingly. Hence when a product is to be derived a set of decisions are being made and then the corresponding logic is executed that selects, generates, transforms, deletes etc. assets. The result is the asset instances that make up a product or parts of a product. The first possible interaction between the asset model and variability management resides in the activity of creating the domain space. A core asset defined with the basic asset model can be connected to one or several decisions in the variability management system. In so doing the description of the core asset variability can be connected to the asset and then to the physical configuration items

(through the connection of the basic asset model to configuration management)

The second possible interaction is the derivation step. When a derivation of a core asset instance takes place in a variability management system the *Instantiation* and the *SignedContract* concepts (see Figure 32) can be used to capture the derivation steps and result respectively. This enhances the traceability of a product derivation procedure which is handled with a variability management infrastructure and the result of the derivation which is handled by evolution control and the Customization Layer.

#### 4.4.2 Connecting the Basic Asset Model to Decision Models

In order to exemplify the connection between the basic asset model and variability management the decision model approach will be used. Figure 41 shows the meta-model of a decision model [Mut02] and the connection to the basic asset model.

In the decision model approach an asset qualifies to be a reusable product line asset if it contains a decision model that manages the variability of the asset. Hence a product line asset contains two parts: a decision model and a product line asset. The latter represents the contents of the product line asset and can be broken down to a set of asset elements. Some of those elements will be varying and therefore are called variant asset elements. A variation point generalizes the concept of a variant asset element which is further generalized by the decision concept. A decision is considered to be more abstract than a variation point because a decision does not directly connect to a variant asset element. In other words decisions enable structuring a variability model in terms of partonomies and implications. The latter are represented in terms of the class *ResolutionConstraint* which in turn is related to a *ConstraintDecision*.

During instantiation of a decision model each decision must be resolved. In the simplest case resolution means deciding whether a decision is true or false (that is why *ConstraintDecision* relates to at least two instances of *ResolutionConstraint*). In more complex cases a resolution may involve selecting among a range of possible values.

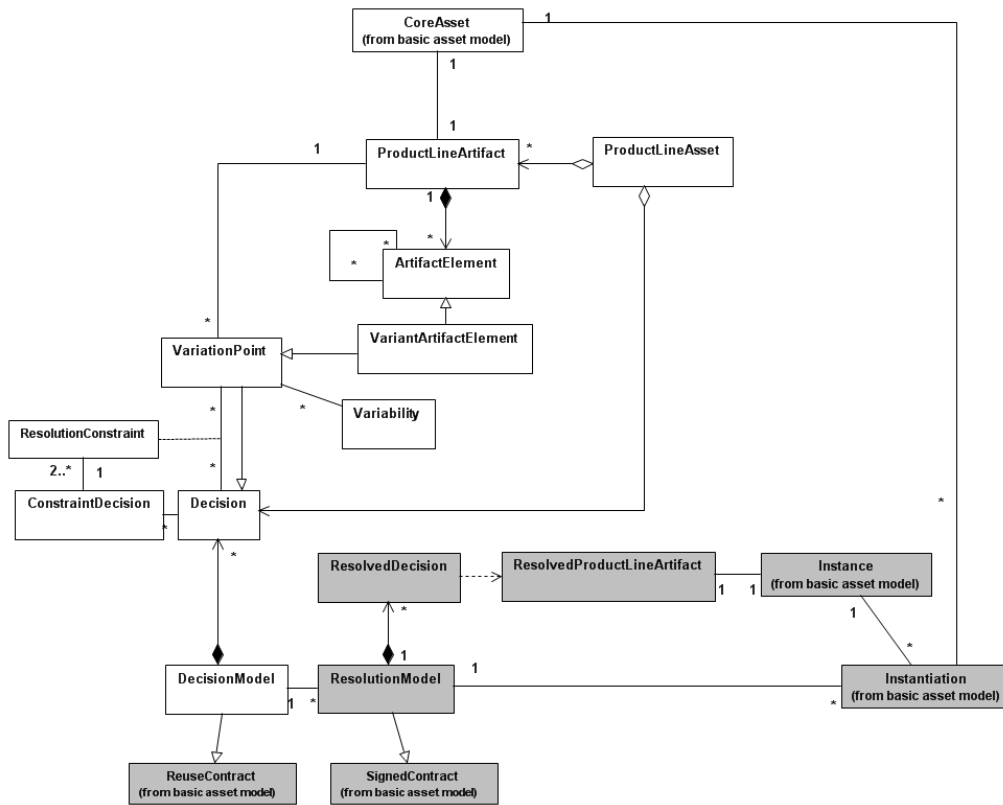


Figure 41: Decision meta-model and relation to basic asset model

The concept of *CoreAsset* from the basic asset model (Figure 31) can be seen as the equivalent of the *ProductLineAsset* from the decision meta-model. On the other hand the classes *Instance* and *Instantiation* (Figure 32) cannot be directly connected since these concepts are not explicit parts of the decision meta-model, although the concept of resolution and resolution model are parts of the approach.

Therefore the decision meta-model is extended in Figure 41 in order to enable the connection to the classes *Instance* and *Instantiation* of the basic asset model. The classes *ResolutionModel*, *ResolvedDecision* and *ResolvedProductLineAsset* are introduced to this end. A resolution model is an instance of a decision model, in which all decisions have been already resolved. Therefore a resolution model can be connected to one or more instantiations. Finally an instance of the basic asset model can be connected to a *ResolvedProductLineAsset*.

There may be various levels of abstractions between decision models and basic asset model. Based on the above mapping a top-level decision can be directly mapped to low level core assets such as single files. Ideally the

same level of abstraction should be enforced: a top-level decision should be mapped to a top-level core asset, embodied for example by a top-level directory in the configuration management repository. Such enforcement can be achieved with the help of the Customization Layer: When a new core asset is created in the Customization Layer the connection to the decision model can become active and enable the user to automatically initialize the corresponding decision model. Furthermore when an instantiation is created in the Customization Layer the decision model can be brought up in order to automate the derivation and to complete the instantiation. A similar interaction can be initiated from the decision modeler when a resolution model is to be created. Chapter 5 that discusses concrete evolution control scenarios, will provide more details on these interactions.

## **4.5 Section summary**

This section has presented the basic asset model, the data model of the Customization Layer, the solution proposed in this thesis. Different logical entities, on which the Customization Layer operates, have been defined and their externally visible states have been modeled. The section concluded with the connection between Customization Layer and variability management, in particular decision modeling. The next section will define the scenarios that operate on the basic asset model.



## 5 Process Model of Evolution Control

Controlling software evolution can be considered as a software engineering process. This applies in particular to product lines since the forces that drive evolution are significantly stronger than with single-system development. As elucidated in the 8<sup>th</sup> law of software evolution [Leh96] “evolution processes are multi-level, multi-loop and multi-agent feedback systems”. In other words there may be many types of feedback loops, many granularity levels at which feedback takes place and finally many stakeholder (agents) that participate in the process.

In the context of product lines the 8<sup>th</sup> law of software evolution becomes even more important. Due to the fact that a family of related products is managed as a whole the number of involved stakeholders increases significantly compared to single-system development. The separation between family and application engineering or even to multiple interconnected family and application engineering phases (cf. hierarchical product line, section 3.6.9) also increases the different levels of granularity.

Finally, the types of possible feedback loops also increase. In single-system development two types of feedback have been observed: positive or “reinforcing” feedback and negative or “balancing” feedback [MKL00]. In the former case the feedback usually leads to increasing the scope of a system through addition of new functional or non-functional properties. In the latter case the feedback usually results from defects in the running system and leads to system maintenance. In other words reinforcing feedback means that changes are introduced into a system while balancing feedback ensures that the introduced changes adhere with the goals of the system. In a product line context these two types of feedback can be extended along the lines of family and application engineering. In product line engineering there can be feedback that reinforces core assets, instances of core assets or other product-specific assets. Accordingly a reinforcing feedback can be accompanied by a balancing feedback for core assets, instances and product-specifics. Moreover these types of feedback can be further specialized as internal or external feedback loops (cf. section 3.5).

In order to cope with the above challenges it is necessary that activities for the evolution control of a product line are clearly defined. This chapter presents such activities in terms of evolution control scenarios that refine the conceptual model presented in chapter 2. The scenarios operate on the entities of the basic asset model discussed in chapter 3.

Each scenario will be described from two points of view:

- **Xtext:** The Xtext-based description shows the scenarios that are available and can be selected when establishing a Customization Layer for a product line. This view refines the conceptual model discussed in section 3.7
- **Interface:** Internally, a Customization Layer will realize each scenario as a subroutine. Therefore, for each scenario a Java-based signature will be shown that specifies the number, types and order of the scenario input and return parameters. The interfaces are meant for interaction between users and a Customization Layer. Therefore, the input and output parameters reflect information that can be provided by or to the user. Moreover, this chapter discusses mainly the signatures (i.e. the publicly available interfaces) of the scenarios. The interaction with configuration management will be discussed in the next chapter.

Evolution control of a product line entails monitoring and controlling changes of core assets and product assets. Based on the categorization introduced in section 3.4 and the refined conceptual model shown in Figure 27 and Figure 28 the scenarios can be grouped as follows:

- **Change management:** Entails management of change requests for assets
- **Version management:** Entails performing changes on assets.
- **Status accounting:** Entails the identification of asset changes and facilitates impact analysis

The following subsections will elucidate the scenarios for family and application engineering. Since the status accounting scenarios are applicable to family and application engineering, they will be discussed in a common section. The chapter will continue with a discussion of change impact analysis as part of evolution control. Finally, possibilities for interaction with variability management will be elucidated.

## **5.1 Evolution Control Scenarios for Family Engineering**

This section discusses change and version management scenarios for family engineering.

### **5.1.1 Creation of core asset change requests**

Regarding change management of core asset the only scenario that has to be automated by a Customization Layer is the creation of change request. Modification and removal of change requests should be

performed through direct usage of the underlying change management system. Following figure specifies the change request creation scenario.

### ***createCoreAssetChangeRequest***

|                                                                                                                                                |
|------------------------------------------------------------------------------------------------------------------------------------------------|
| XTEXT SPECIFICATION                                                                                                                            |
| <pre>createCoreAssetChangeRequest:     'createCoreAssetChangeRequest' name=ID;</pre>                                                           |
| INTERFACE SPECIFICATION                                                                                                                        |
| <pre>String createCoreAssetChangeRequest(     String[] caID,     boolean synchronizeInstances )     throws CoreAssetCRCreationException;</pre> |

Figure 42: createCoreAssetChangeRequest scenario

`createCoreAssetChangeRequest` enables the creation of a change request for one or more core assets of a family engineering process. As shown in the Xtext view it is again necessary to assign a unique identifier to the scenario.

The signature of the scenario allows passing a set of core asset IDs. However all core assets should pertain to the same product line engineering process. Based on these IDs and the relation between core assets and family engineering processes (see Figure 31) the Customization Layer shall identify the location, on which to store the change request in the underlying change management system. The scenario returns a change request ID as a string that represents the newly created change request. A *CoreAssetCRCreationException* is thrown if the operation fails.

The signature also accepts a Boolean parameter *synchronizeInstances*. This parameter specifies whether the family engineering process at hand shall be associated with the corresponding application engineering processes. If such an association is established the creation of a core asset change request leads to the creation of change requests for each of the core asset instances. The purpose of that is to perform changes on core asset in synchronization with changes on instances. A ticket hierarchy approach [UKR09] can come into play in this case.

The ticket hierarchy approach proposes to structure change requests, referred to as tickets, hierarchically. Top-level (or main) tickets are produced during family engineering and subtickets during application engineering. Modern change management environments like JIRA [URL9] enable this kind of hierarchy. The mechanism resembles the



branching mechanism in version management. As with braches main tickets and subtickets can be interrelated and corresponding rules can be established. For example a rule can be defined, which imposes than a main ticket can only be closed when all subtickets are first closed.

This approach enables a broad propagation of change requests across the closure of all related core assets and instances. The implementation of the scenario must therefore assure that no change request duplication occurs (more on that in section 6).

5.1.2 Scenarios for version management of core assets

|                                                                                                                                                                              |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>createCoreAsset</b>                                                                                                                                                       |
| XTEXT SPECIFICATION                                                                                                                                                          |
| <pre>createCoreAsset:     'createCoreAsset' name=ID;</pre>                                                                                                                   |
| INTERFACE SPECIFICATION                                                                                                                                                      |
| <pre>void createCoreAsset(     String sourceLocation,     String targetLocation,     String templateLocation,     String depth )     throws CoreAssetCreationException</pre> |

Figure 43: createCoreAsset scenario

createCoreAsset creates a core asset based on a given source location of the asset contents. This location can be in a file system or in a configuration management repository. The Customization Layer shall perform all necessary steps in order to put the asset contents at the target location under configuration management control and to mark them as core assets. If the source location is already a repository location the target location can be omitted and the Customization Layer only marks the source location as a core asset location. The exception *CoreAssetCreationException* is thrown if the creation fails, for example due to an invalid source or target location.

The scenario method also accepts a template as input parameter. The purpose of this parameter is to tell the Customization Layer what kind of directory structure to use when creating an empty core asset (in this case *sourceLocation* is nil). This can be useful if an organization has a predefined directory structure for reusable assets. The parameter *templateLocation* refers thus to a location containing this directory

structure. In this case the Customization Layer extracts the structure and applies it upon creating the asset.

An interesting situation arises when the asset to be added already contains content, i.e. when the asset is a directory containing files and other directories. In this situation the parameter *depth* enables describing as a regular expression to which extend the contents of the assets are to be marked as core assets. If *depth* is equal to nil the Customization Layer shall only mark the asset (i.e. the directory) as a core asset. However, if a regular expression is passed the Customization Layer shall look up all directory entries matching this expression and mark them as core assets.

The creation of a core asset involves the creation of a core asset object according to the model discussed in section 4.1.1. The following table shows how the Customization Layer can populate the model based on the input parameters and possibly through additional interaction with the user and a variability management system.

| Core Asset Attribute | Population approach                                                                              |
|----------------------|--------------------------------------------------------------------------------------------------|
| name                 | Obtained through user interaction or set equal to the item attribute (next row)                  |
| item                 | Assigned automatically based on the input source or target location                              |
| process              | Assigned automatically since every scenario is related to a process (see section 3.8)            |
| reuseContract        | Depends on the availability of a connector to a variability management system (see section 5.5). |
| Instantiation        | Not populated during core asset creation                                                         |

Table 8: Population of core asset attributes

**removeCoreAsset**

|                                                                                    |
|------------------------------------------------------------------------------------|
| XTEXT SPECIFICATION                                                                |
| <pre>removeCoreAsset:     'removeCoreAsset' name=ID;</pre>                         |
| INTERFACE SPECIFICATION                                                            |
| <pre>void removeCoreAsset(String caID)     throws CoreAssetDeletionException</pre> |

Figure 44: removeCoreAsset scenario

removeCoreAsset removes the marks that indicate a particular configuration management repository location as a core asset. It expects a core asset ID as input and throws a *CoreAssetDeletionException* exception if the operation fails (e.g. invalid core asset ID, open change requests on the asset).

An interesting situation arises when the core asset to be removed has been already instantiated. In this case the Customization Layer must ensure that instances of the removed core asset maintain a consistent state. One strategy is to associate the affected instances with a special-purpose core asset that represents deleted assets. Another strategy is to transform the affected instances to product-specific assets if there are no other instantiation relations to core assets.

**modifyCoreAsset**

|                                                                                        |
|----------------------------------------------------------------------------------------|
| XTEXT SPECIFICATION                                                                    |
| <pre>modifyCoreAsset:     'modifyCoreAsset' name=ID;</pre>                             |
| INTERFACE SPECIFICATION                                                                |
| <pre>void modifyCoreAsset(String caID)     throws CoreAssetModificationException</pre> |

Figure 45: modifyCoreAsset scenario

modifyCoreAsset enables the user to modify information about a core asset. Upon invocation the Customization Layer allows modification of core asset attributes (see Figure 31) and subsequently stores the information back to the configuration management system. A *CoreAssetModificationException* is thrown if the operation fails.

***integrateCoreAsset***

|                                                                                          |
|------------------------------------------------------------------------------------------|
| XTEXT SPECIFICATION                                                                      |
| <pre>integrateCoreAsset:     'integrateCoreAsset' name=ID;</pre>                         |
| INTERFACE SPECIFICATION                                                                  |
| <pre>void integrateCoreAsset(String caID)     throws CoreAssetIntegrationException</pre> |

Figure 46: integrateCoreAsset scenario

This scenario corresponds to the integration region in the core asset state machine (see section 4.2.1) and aims at integrating a core asset with its instances. Therefore, when the scenario is invoked the Customization Layer must allow propagation of instance changes to the corresponding core assets. There are two strategies that can be followed:

- A-posteriori integration: In this case the Customization Layer assumes that changes have been already propagated. That is, core assets and instances have been changed in a way that served propagation of changes. In this case the Customization Layer only marks the performed changes as integration changes
- Session-based integration: In this case the Customization Layer shall open a session, in which the user will perform the integration. The Customization Layer has to guide the user across the different steps necessary for the integration (identification of instances, modification of instances etc.).

## 5.2 Evolution Control Scenarios for Application Engineering

This section discusses change and version management scenarios for application engineering. Since the scenarios are almost identical to the corresponding scenarios in family engineering only the major differences will be discussed in the following.

5.2.1 Creation of product asset change requests

***createProductAssetChangeRequest***

|                                                                                                                                                       |
|-------------------------------------------------------------------------------------------------------------------------------------------------------|
| XTEXT SPECIFICATION                                                                                                                                   |
| <pre>createProductAssetChangeRequest :     'createProductAssetChangeRequest' name=ID;</pre>                                                           |
| INTERFACE SPECIFICATION                                                                                                                               |
| <pre>String createProductAssetChangeRequest(     String[] paID,     boolean synchronizeCoreAssets )     throws ProductAssetCRCreationException;</pre> |

Figure 47: createProductAssetChangeRequest scenario

At this point it is worth discussing the *synchronizeCoreAssets* parameter that can be passed to the *createProductAssetChangeRequest* scenario. Similar to the *createCoreAssetChangeRequest* scenario the idea here is to keep change requests for core assets and instances synchronized. Therefore if the parameter is set to true when a change request for an instance is opened, the implementation shall create a change request for the corresponding core assets as well.

5.2.2 Scenarios for Version Management of product assets

***createProductAsset***

|                                                                                                                                                                                                                                                                              |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| XTEXT SPECIFICATION                                                                                                                                                                                                                                                          |
| <pre>createProductAsset :     'createProductAsset' name=ID;</pre>                                                                                                                                                                                                            |
| INTERFACE SPECIFICATION                                                                                                                                                                                                                                                      |
| <pre>enum InstantiationStrategy { DEEP, SHALLOW };  void createProductAsset(     String sourceLocation,     String targetLocation,     String templateLocation,     boolean isInstance,     InstantiationStrategy iStrategy )     throws ProductAssetCreationException</pre> |

Figure 48: createProductAsset scenario

The creation of a product asset may involve the creation of a product-specific asset or the creation of an instance. In the former case the scenario works in a similar way as the `createCoreAsset` scenario. In the latter case however (*isInstance* parameter set to true) the realization of the scenario must ensure that relations are established between the to-be-created instance and the corresponding core assets. In other words, when the scenario is invoked a session is to be started, in which the user is asked to provide the attributes of the newly-to-be-created instance (see section 4.1.2). Thereby the user will have to provide, among other things, the ID of core assets for which an instantiation is to be created.

If the *sourceLocation* is left empty and *isInstance* is set to true, the Customization Layer can copy the contents of the core assets to be instantiated to the location designated by the *targetLocation* attribute. On the other hand if *isInstance* is set to true and *sourceLocation* is not empty the Customization Layer has to assume that in the given *sourceLocation* there is already an instance of the corresponding core assets. The contents of this location are then put under *targetLocation* or they are left unchanged if *targetLocation* is empty and *sourceLocation* is a repository location. In every case the Customization Layer has to mark the contents as instances of the specified core assets. The *ProductAssetCreationException* is thrown if the usage of the *sourceLocation*, *targetLocation* and *isInstance* attributes is not appropriate (e.g. both *sourceLocation* and *targetLocation* are nil) or if there is a problem with the creation of the product asset (e.g. network connection problem)

The `createProductAsset` scenario also accepts an instantiation strategy as an input, which according to Figure 48 can be deep or shallow. The choice of a strategy is necessary, since core assets may form a composition hierarchy. In other words it can happen that the configuration item associated with a core asset contains other configuration items, which in turn are associated with other core assets. For example a top-level directory that contains a library, is associated to a library core asset and contains a series of other directories. The latter are associated with other core assets, which thus can be seen as parts of the library core asset. Both strategies make sense when *sourceLocation* is nil and are explained in the following:

- Deep instantiation: When a composite core asset is instantiated, all contained core assets are instantiated as well. That means, if the core asset contains other core assets, they will be instantiated as well. The containment relations can be identified in terms of the corresponding configuration items.
- Shallow instantiation: When a composite core asset is instantiated, only the root configuration item of the core asset is instantiated.

As with the createCoreAsset scenario the createProductAsset has also to populate the attributes of an instance object with values according to the model discussed in section 4.1.2. The following table discusses how these attributes can be populated.

| Core Asset Instance Attribute | Population approach                                                                                                                                                                                                                                                                       |
|-------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| name                          | Obtained through user interaction of set equal to the item attribute (next row)                                                                                                                                                                                                           |
| item                          | Assigned automatically based on the input source or target location                                                                                                                                                                                                                       |
| process                       | Assigned automatically since every scenario is related to a process (see section 3.8)                                                                                                                                                                                                     |
| Instantiation                 | Obtained through user interaction: The Customization Layer is to query the user for the core assets to be instantiated. If a connection to a variability management system is available (see section 5.5) the Customization Layer can also assign a signed contract to the instantiation. |

Table 9: Population of core asset instance attributes

### ***removeProductAsset***

|                                                                                                                        |
|------------------------------------------------------------------------------------------------------------------------|
| XTEXT SPECIFICATION                                                                                                    |
| <code>removeProductAsset:<br/>    'removeProductAsset' name=ID;</code>                                                 |
| INTERFACE SPECIFICATION                                                                                                |
| <code>void removeProductAsset(String paID, boolean<br/>detachOnly)<br/>    throws ProductAssetDeletionException</code> |

Figure 49: removeProductAsset scenario

As with the removeCoreAsset scenario, removeProductAsset aims at removing all marks that identify a configuration item as a product asset. In the case of a core asset instance the scenario allows to only detach the instance from the corresponding core asset (parameter *detachOnly* has to be set to true). In this case the scenario makes the asset a product-specific asset.

***modifyProductAsset***

|                                                                                                |
|------------------------------------------------------------------------------------------------|
| XTEXT SPECIFICATION                                                                            |
| <pre> modifyProductAsset:     'modifyProductAsset' name=ID; </pre>                             |
| INTERFACE SPECIFICATION                                                                        |
| <pre> void modifyProductAsset(String paID)     throws ProductAssetModificationException </pre> |

Figure 50: modifyProductAsset scenario

modifyProductAsset enables the user to modify information about a product asset. It is setup similarly to the modifyCoreAsset scenario.

***rebaseProductAsset***

|                                                                                          |
|------------------------------------------------------------------------------------------|
| XTEXT SPECIFICATION                                                                      |
| <pre> rebaseProductAsset:     'rebaseProductAsset' name=ID; </pre>                       |
| INTERFACE SPECIFICATION                                                                  |
| <pre> void rebaseProductAsset(String paID)     throws ProductAssetRebaseException </pre> |

Figure 51: rebaseProductAsset scenario

rebaseProductAsset is the equivalent of the integrateCoreAsset scenario (section 5.1.2) and maps to the rebase region of the product asset state machine (section 4.2.2). It aims at propagating changes from core assets back to the corresponding instances. Again, the two strategies discussed in the integrateCoreAsset scenario are applicable.

### 5.3 Common Status Accounting Scenarios

Status accounting scenarios enable retrieval of information from the configuration management repository. As it will be shown in the following, all status accounting scenarios allow filtering the result set, in order to obtain more precise information. To that end a *searchCriteria* parameter will come into play in the following.

This parameter takes the form of a predicate over the attributes and operations of the result type (for example change requests, core assets, and product assets) and the corresponding values. Such attributes can be:



- Attributes of the result type, e.g. "name == 'VectorLibrary'"
- Operations on these attributes. e.g. "name.length() > 5"
- Attributes of corresponding configuration items (in this case attributes will depend on the configuration management system at hand), e.g. "lastChange == '01.12.2010'"
- Asset states, e.g. "state == 'integrated'"
- A foreach statement as proposed by the Scala programming language specification [Od10]. This statement applies a Boolean function to each element of a collection. It can be useful for collection attributes. For example in order to obtain all change requests that correspond to core assets the following predicate could be passed:  

```
"getAssets().foreach((x:Asset) =>
CoreAsset.class.isInstance(x))"
```

**showChangeRequests**

|                                                                                                           |
|-----------------------------------------------------------------------------------------------------------|
| XTEXT SPECIFICATION                                                                                       |
| <pre>showChangeRequests:   'showChangeRequests' name=ID;</pre>                                            |
| INTERFACE SPECIFICATION                                                                                   |
| <pre>String[] showChangeRequests(String searchCriteria)     throws ChangeRequestRetrievalException;</pre> |

Figure 52: showChangeRequests scenario

showChangeRequests lists change requests associated with assets (core or product assets). It takes search criteria as input and delivers an array of strings that describes the obtained change requests.

**showCoreAssets**

|                                                                                                   |
|---------------------------------------------------------------------------------------------------|
| XTEXT SPECIFICATION                                                                               |
| <pre>showCoreAssets:   'showCoreAssets' name=ID;</pre>                                            |
| INTERFACE SPECIFICATION                                                                           |
| <pre>String[] showCoreAssets(String searchCriteria)     throws CoreAssetRetrievalException;</pre> |

Figure 53: showCoreAsset scenario

showCoreAssets delivers a list of product assets based on search criteria. The typical search criterion will be the family engineering process of interest; however further criteria are conceivable such as the current state of a core asset (see section 4.2.1).

### ***showCoreAssetInstances***

|                                                                                                                                         |
|-----------------------------------------------------------------------------------------------------------------------------------------|
| XTEXT SPECIFICATION                                                                                                                     |
| <pre>showCoreAssetInstances:     'showCoreAssetInstances' name=ID;</pre>                                                                |
| INTERFACE SPECIFICATION                                                                                                                 |
| <pre>String[] showCoreAssetInstances(     String[] coreAssets,     String searchCriteria )     throws InstanceRetrievalException;</pre> |

Figure 54: showCoreAssetInstances scenario

showCoreAssetInstances takes a set of core assets as input and provides a list of all corresponding instance assets. Again, search criteria can be used in order to refine the result. For example it may be necessary to deliver only the instances of a core asset within a given application engineering process.

This scenario is a shortcut as the same information can be provided by the showCoreAssets scenario. The latter delivers a list of core assets including their instantiations, which refer to the corresponding instances.

### ***showCoreAssetChanges***

|                                                                                                                                                                                 |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| XTEXT SPECIFICATION                                                                                                                                                             |
| <pre>showCoreAssetChanges:     'showCoreAssetChanges' name=ID;</pre>                                                                                                            |
| INTERFACE SPECIFICATION                                                                                                                                                         |
| <pre>String[] showCoreAssetChanges(     String[] coreAssets,     String searchCriteria,     boolean sinceLastSynchronization )     throws ModificationRetrievalException;</pre> |

Figure 55: showCoreAssetChanges scenario

showCoreAssetChanges shows changes performed on a set of core assets based on the given search criteria. If the underlying configuration management system supports version of change requests, the operation will also deliver changes in the change requests that correspond to the given core assets. The *sinceLastSynchronization* parameter addresses the synchronization between core assets and instances. When the parameter is set to true the scenario shall list only the core asset changes that have been performed since the last time, when the core asset was in the integrated state (see section 4.2.1).

**showProductAssets**

|                                                                                                             |
|-------------------------------------------------------------------------------------------------------------|
| XTEXT SPECIFICATION                                                                                         |
| <pre>showProductAssets:<br/>    'showProductAssets' name=ID;</pre>                                          |
| INTERFACE SPECIFICATION                                                                                     |
| <pre>String[] showProductAssets(String searchCriteria)<br/>    throws ProductAssetRetrievalException;</pre> |

Figure 56: showProductAssets scenario

showProductAssets delivers a list of product assets based on a set of search criteria (the *searchCriteria* parameter). For example, by providing the type of product asset (e.g. SpecificAsset or CoreAssetInstance see section 4.1.2) the scenario will retrieve the list of assets from corresponding type.

**showInstanceCoreAssets**

|                                                                                                                                                         |
|---------------------------------------------------------------------------------------------------------------------------------------------------------|
| XTEXT SPECIFICATION                                                                                                                                     |
| <pre>showInstanceCoreAssets:<br/>    'showInstanceCoreAssets' name=ID;</pre>                                                                            |
| INTERFACE SPECIFICATION                                                                                                                                 |
| <pre>String[] showInstanceCoreAssets(<br/>    String[] instances,<br/>    String searchCriteria<br/>)<br/>    throws CoreAssetRetrievalException;</pre> |

Figure 57: showInstanceCoreAssets scenario

showInstanceCoreAssets delivers the list of core assets that correspond to a set of instances as well as to the given search criteria. As with the showCoreAssetInstances scenario this is again a shortcut scenario as the

same information can be obtained via the `showProductAssets` or the `showCoreAssets` scenarios.

### ***showProductAssetChanges***

|                                                                                                                                                                                       |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| XTEXT SPECIFICATION                                                                                                                                                                   |
| <pre>showProductAssetChanges:     'showProductAssetChanges' name=ID;</pre>                                                                                                            |
| INTERFACE SPECIFICATION                                                                                                                                                               |
| <pre>String[] showProductAssetChanges(     String[] productAssets,     String searchCriteria,     boolean sinceLastSynchronization )     throws ModificationRetrievalException;</pre> |

Figure 58: `showProductAssetChanges` scenario

`showProductAssetChanges` shows changes performed on a set of product assets and change requests (depending on availability of change request history) based on the given search criteria. The *sinceLastSynchronization* parameter addresses again the synchronization between core assets and instances. When the parameter is set to true the scenario shall list only the product asset changes that have been performed since the last time, when the product asset (it has to be an instance actually) was in the rebased state (see section 4.2.2).

## **5.4 Change impact analysis**

Change impact analysis is an important component of change management. It aims at identifying the potential consequences of a change, or estimating what needs to be modified to accomplish a change [Bo96]. In a product line context, the status accounting and change management scenarios, as discussed above, support the identification, classification and analysis of a change impact. In this regard the `showCoreAssetChanges` and `showProductAssetChanges` scenarios address already performed changes while the `showChangeRequests` scenario addresses planned changes. However, change impact analysis requires additional capabilities, which will be discussed in the following sections.

### **5.4.1 Comparing core assets with instances**

The identification of the change impact often involves comparison of different asset versions. The latter can become a complex undertaking in

a product line setting. Core assets are usually modified during instantiation. Therefore the comparison of a new core asset version with an existing instance may not be directly possible. In order to enable this comparison the implementation mechanisms used for the creation of core assets must be taken into account. According to [Be04] there are three main mechanisms:

- **Selection:** With selection variation points in core assets (see section 4.4) are realized as a set of options. A variation point is instantiated by selecting one of the available options. Therefore the impact of variation point changes can be analyzed by retrieving the selections made in instances and by evaluating whether the variation point change affects these selections.
- **Generation:** With generation a variation point is realized in terms of a generator that can transform the variation point to an instance. Typically a generator provides a mechanism for the specification of the desired output. Such a mechanism usually allows mapping core asset elements to elements of instances. To this end the generic format of instances must be known in advance. Impact analysis can be performed in this case by retrieving the generator specification that was used in order to create a core asset instance. If the core asset change involves a modification of the specification mechanism the impact analysis checks whether the new mechanism is in conflict with the instance specification. If the core asset change involves a change of the core asset itself, impact analysis checks if parts of the instance specification refer to core asset elements that have changed.
- **Substitution:** With substitution variation points are realized as placeholders that can be filled in a prescribed way with elements relevant to a product. Impact analysis can be performed by retrieving the placeholders that are used in instances as well as the contents that were given as input to the placeholders. It can be subsequently checked whether the modifications affect the placeholders used in instances. It can for example happen that after a core asset modification the contents of an instance become invalid because they do not comply anymore with the core asset placeholders.

Selection sets actually the grounds for all three implementation mechanisms:

- Given a core asset, selection can produce the final core asset instance that consists of mandatory core asset elements, selected core asset elements and product-specifics.

- Generation operates on a selection of core asset elements. A generator transforms the selected core asset elements against the target format of instances.
- Substitution also operates on a selection of core asset elements. In this case placeholders can be seen as selected core asset elements. Substitution changes these elements by filling them with content.

#### 5.4.2 Formal model for impact analysis

In order to set the grounds for an automated change impact analysis between core assets and their instances the following formal model has been set-up. The initial observation is that a product line can be seen as tuple consisting of processes and assets:

$$PL = (PROCESSES, ASSETS)$$

The set of assets in a product line consists of a set core assets and a set of product assets. The latter is the union of product-specific assets with core asset instances:

$$ASSETS = (ASSETS_{core}, ASSETS_{product})$$

$$ASSETS_{product} = ASSETS_{specific} \cup ASSETS_{instantiated}$$

The set of all core assets can be defined as:

$$ASSETS_{core} = (asset_{core_1}, asset_{core_2}, \dots, asset_{core_n})$$

The set of instance assets can be defined as:

$$ASSETS_{instance} = (asset_{instance_1}, asset_{instance_2}, \dots, asset_{instance_n})$$

Each core asset can be seen as a set, which consists of mandatory and optional elements.

$$asset_{core} = \{elements_{mandatory} \cup elements_{optional}\}$$

Each instance asset can be defined as a set, which consists of instantiated elements (i.e. elements obtained from core assets) and product-specific elements.

$$asset_{instance} = \{elements_{instantiated} \cup elements_{specific}\}$$

Based on the above, the union of all core asset elements can be defined as follows:

$$AllAssetElements_{core} = \bigcup_{asset_{core}} \forall asset_{core} \in ASSETS_{core}$$

Similarly, the union of all instance asset elements can be defined as

$$AllAssetElements_{instantiated} = \bigcup_{asset_{instance}}^{asset_{instance}} \forall asset_{instance} \in ASSETS_{instance}$$

Applying the selection mechanism on a core asset produces an asset that contains the mandatory core asset elements and a subset of the optional elements.

$$select(asset_{core}) = \{elements_{mandatory} \cup (elements_{selected} \subseteq elements_{optional})\}$$

Selection can produce the final instance of a core asset; however it is also conceivable that a function *derive* is applied on the selection result. That function can be the identity function, a generation function, a substitution function or any composition thereof. The composition *derive*  $\circ$  *select* yields then the *instantiation* function. The latter delivers the instantiated elements of an instance asset. The function is applied on the union of all core asset elements according to the instantiation strategies discussed in section 4.1.2.

$$elements_{instantiated} = instantiate(AllAssetElements_{core}) = derive(select(AllAssetElements_{core}))$$

Changing a core asset involves changing its elements. Therefore a changed core asset consists of changed mandatory elements and changed optional elements.

$$change(asset_{core}) = \left\{ \begin{array}{l} change_{mandatory} \subseteq elements_{mandatory} \\ \cup \\ change_{optional} \subseteq elements_{optional} \end{array} \right\}$$

In order to characterize the impact of core asset changes on instances it is necessary to know if instances were produced through instantiation of elements, which have changed. A function can be defined that delivers the set of instance elements affected by the core asset change.

$$impact_{FE \rightarrow AE}(asset_{core}, asset_{instance}) = (i_1, i_2, \dots, i_k) : \left( i_j \in asset_{instance} \wedge i_j \in instantiate(change(asset_{core})) \right) \forall j \in (1, \dots, k)$$

Similarly, changing an instance asset involves changing its elements. Therefore a changed core asset instance consists of changed elements that originate from core assets and changed product-specific elements.

$$change(asset_{instance}) = \left\{ \begin{array}{c} change_{instantiated} \subseteq elements_{instantiated} \\ \cup \\ change_{specific} \subseteq elements_{specific} \end{array} \right\}$$

Again a function can be defined that delivers the core asset elements affected by the instance change.

$$impact_{AE \rightarrow FE}(asset_{instance}) = (c_1, c_2, \dots, c_n) \in AllAssetElements : \\ instantiate(c_i) \forall i \in (1, \dots, n) \subseteq change(asset_{instance})$$

The differencing approach proposed in [ALB+11] can be beneficial at this point. The approach allows to associate conflict handlers with particular asset elements. Therefore the approach can be used to associate special handlers with variation points. Upon asset comparison handlers can guide and partially automate the resolution of conflicts. However, such handlers must still have access to the instantiation function mentioned above.

### 5.4.3 Change impact analysis activities

Change impact analysis can be classified as proactive or as reactive depending on the point in time when the analysis occurs. Reactive analysis looks into already performed changes, while proactive analysis operates on change requests.

#### **Reactive analysis of core asset changes**

Change impact analysis in a product line context must pay special attention on the variability dependencies between core assets. As discussed in section 4.4 core assets typically contain variation points, which are possibly related to a variability model (e.g. a decision model). The latter in turn is possibly related to the product line scope, which specifies the spectrum of commonality and variability in the product line. Therefore the first step in change impact analysis is to examine whether a change affects variation points and the product line scope. Figure 59 summarizes the activities that have to be performed within family engineering in order to evaluate impact of internal changes (i.e. changes in the own core assets).

Variation points can have interdependencies. Taking a decision in one core asset may require taking a particular decision in another core asset. When core assets change it is possible that the enclosed variation points change as well. In such a case variation point dependencies are



influenced. For example if core asset modification removes a variation point which is being referenced by another core asset, the instantiation of the latter may not produce valid instances any more. Hence, variation point consistency must be ensured after such a modification.

Consistency means that the set of derivable instances in a core asset base remains unaffected after a core asset modification. Violation of the variation point consistency is an indicator that a change can have unwanted side effects. However there may be cases when such a violation is normal because the variability model has to change.

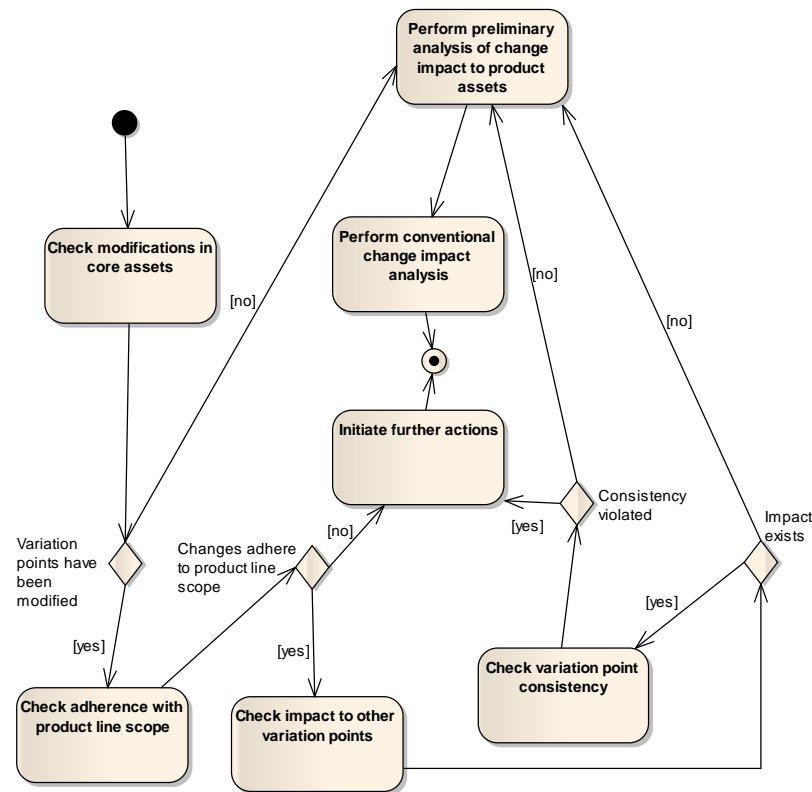


Figure 59: Reactive analysis of core asset changes

### ***Proactive analysis of core asset changes***

In the proactive case change requests on core assets are analyzed and a decision is taken, whether to accept, reject or defer the change. Change requests usually classify the requested change, characterize the priority and identify the items that have to be changed. Figure 60 illustrates the process.

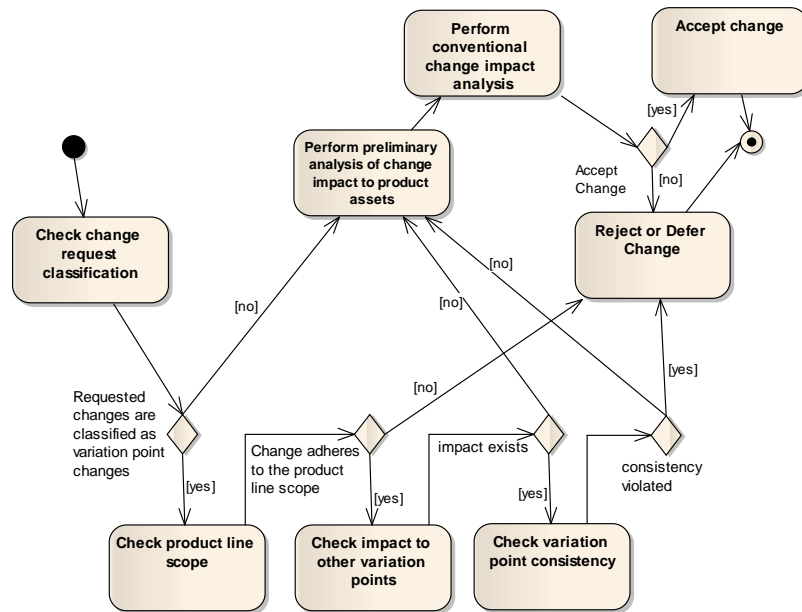


Figure 60: Proactive analysis of core asset changes

As in the reactive case, complexity of change analysis rises if variation points are to be modified or if application engineering is affected by the planned changes. In the former case the impact to the variation point consistency must be examined. When variation points are changed, impact analysis may decide to accept a change if adherence to the product line scope is given and the variation point consistency is guaranteed. If these constraints are not fulfilled the change request can be rejected and the creation of new change request may be necessary. If finally the change request is justifiable but requires a modification of the product line scope it can be deferred until the scope is modified. The final decision of whether to accept or to reject the change request also depends on the analysis of the impact on application engineering (preliminary analysis of impact on application engineering; the full impact analysis will be performed in the application engineering context) or on other core assets (conventional impact analysis)

### **Reactive analysis of product asset changes**

When product asset changes affect core assets, change impact analysis has to examine the reasons, for which product assets have been modified. If the reasons lead back to the core assets at hand, measures may have to be undertaken.

An instance may have changed for example because of removing a defect that was present in the core asset and that was passed to the instance during instantiation. Quality assurance of instances is likely to reveal defects of core assets since instances are executed in the context of real products, while core assets are often not directly executable.

Instances may also change because of reusability problems. It is possible that a core asset does not exhibit an adequate level of reusability. For that reason instances obtained from such a core asset must be often adapted in the context of products. This can be another indication that core assets must be improved. Such a problem can also arise because of poor documentation of the reusability or because of poor automation support. Application engineers may prefer to adapt core assets from scratch than to follow an inefficient software reuse process.

Another scenario arises when instance assets are changed without any assumptions about core assets. This scenario usually arises when a product asset or instance is modified because of product-specific change requirement. In this situation feedback to the core assets can be beneficial to check the degree to which this change is really product-specific. It can happen that similar product-specific changes appear in other products as well. As the family engineering keeps an overview over products and instances such a check should be performed in a regular basis. If it is indeed the case that similar changes are performed in various products, this is an indication that these changes should be taken over to the corresponding family engineering process.

Finally there may exist also situations in which changes on product-specific assets are of interest to family engineering and may lead back to changes on the core asset base. For example, if family engineering finds out that a set of application engineering activities are producing similar product-specific assets, measures can be planned to join the efforts and to create appropriate core assets. The latter can then be reused across the identified application engineering processes. Figure 61 summarizes the activities that have to be performed within family engineering in order to evaluate impact of changes in product assets (i.e. instances and product-specifics).

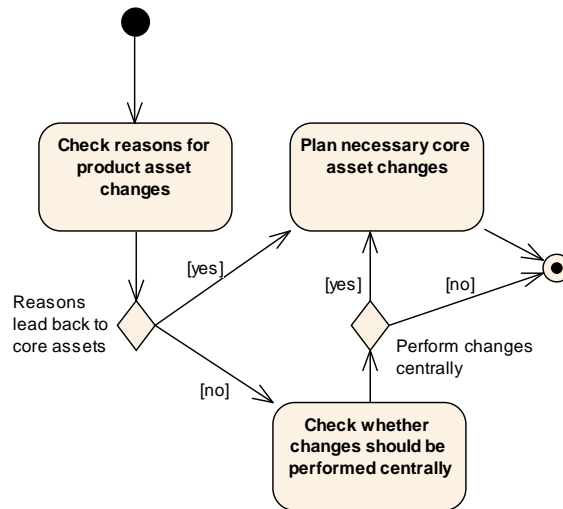


Figure 61: Reactive analysis of product asset changes

### ***Proactive analysis of product asset changes***

When product assets are changed proactively impact analysis looks into changes that are planned on instances and on product-specific assets. The goal is again to analyze the reasons for which assets are to be changed. At this point the proactive control has the advantage of the upfront decision, whether the changes are allowed or not. If for example an instance is to be changed for a reason that lies in family engineering, impact analysis may reject the change. Subsequently a procedure can be opened in order to perform that change directly in the corresponding core asset. This can be beneficial in case the planned instance change is of interest for other instances as well. When this happens it is reasonable to do the change centrally in family engineering and then to update the instances.

Requested changes can lead back to a core asset while there is no immediate need to change them. In this case it can be decided to perform the changes locally in the instances. This however depends on the specific change procedures that are in place in an organization. Figure 62 illustrates again the impact analysis process.

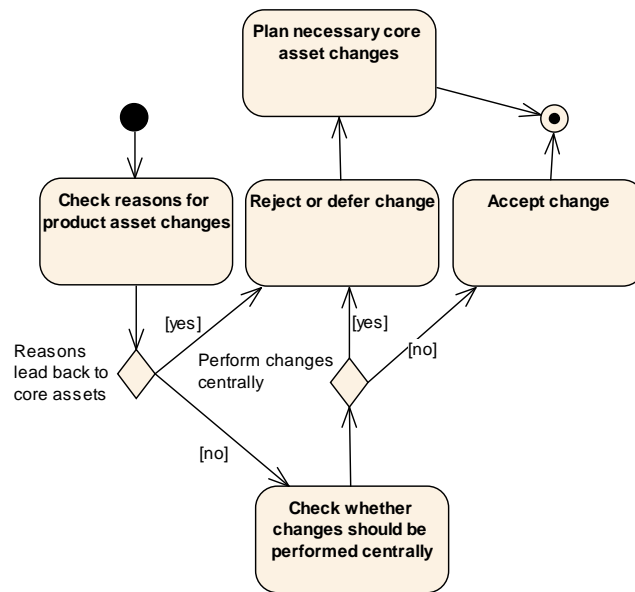


Figure 62: Proactive analysis of product asset changes

## 5.5 Interaction with Variability Management

Evolution control scenarios as described in section 5 can be combined with variability management. This section will hence discuss possible interactions between an evolution control system (i.e. a Customization Layer) and a variability management system (e.g. a decision modeling tool).

### 5.5.1 Core asset creation

The creation of a core asset may involve interaction with variability management as shown in the following figure. A decision model is used as an example of a reuse contract (see Figure 31) in this case.

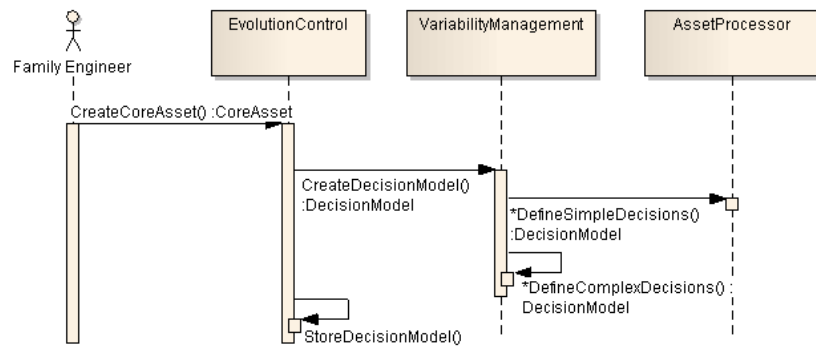


Figure 63: Interactions between Evolution Control and Variability Management (core asset creation)

The interaction starts when a family engineer initiates the creation of a core asset. Subsequently evolution control invokes variability management in order to setup the decision model of the core asset under creation. This starts a process in variability management which also involves an asset processor. This can be any tool in the development process (e.g. integrated development environment, architecture modeling tool, requirements specification tool) that can process assets. When a decision model is to be created the family engineer has to use the asset processor to specify variation points within the assets, which in turn map to decisions in the decision model. Decisions can be subsequently aggregated towards more complex decisions. The core asset creation process ends with the delivery and storage of the decision model for the core asset. In terms of a Customization Layer storage can mean the creation of configuration item that holds or refers to the contents of the decision model. That decision model may be part of a bigger decision model that involves other core assets as well. Variability management is responsible for the management of the corresponding cross references.

### 5.5.2 Instance creation

Figure 64 illustrates interaction between evolution control and variability management in the context of an instance creation.

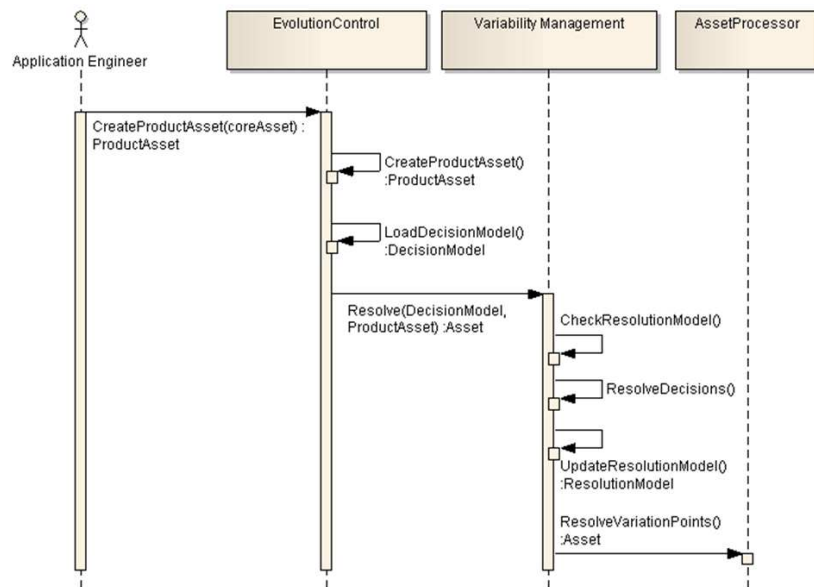


Figure 64: Interactions between Evolution Control and Variability Management (instance creation)

The interaction starts when an application engineer invokes the creation of a product asset (i.e. an instance) out of a core asset. First, this makes evolution control to internally create the product asset. Then, if a decision model is attached to the core asset it will be loaded and passed to variability management. However, it can be the case that no decision model is stored for the specific core asset. This can happen if the variability decisions for the core asset under instantiations are part of a bigger decision model that involves other core assets as well. In this case evolution control will pass a null decision model and variability management will try to locate the core asset in the overall decision model for the product line.

In the next step variability management checks whether there is already a resolution model for the product under development. If there is a resolution model it checks for the existence of decisions relating to the core asset under development which are already resolved. Subsequently it starts the resolution process (i.e. usually with the help of configuration wizards) for the remaining decisions. Given the final resolution model variability management invokes, if applicable, the corresponding asset processor (i.e. tool that can process the core asset under instantiation) in order to resolve the variation points and to obtain the core asset instance.

### 5.5.3 Further interactions

While evolution control focuses on the evolution of assets variability management focuses on the management of variations on derivation of products. With variability management a domain space is typically built up that specifies the decisions an application engineer has to make when deriving a product or parts of a product. Decisions in the domain space are mapped to logic that delivers the assets accordingly. Hence when a product is to be derived a set of decisions are being made and then the corresponding logic is executed that selects, generates, transforms, deletes etc. assets. The result comprises the assets that make up a product or parts of a product.

Another possible interaction is hence possible from the variability management towards the Customization Layer. Whenever a core asset is defined within the variability management system the `createCoreAsset` scenario can be invoked in order to store the core asset accordingly in the configuration management repository.

A further interaction from variability management towards Customization Layer can occur as part of the derivation step. For each core asset being processed by the derivation logic the `createProductAsset` scenario can be called. The interface of the scenario would have to be changed though in order to be able to directly pass assets. The scenario interfaces, as presented thus far, are meant for interaction with users and therefore they do not allow direct passing of core assets or instances. Yet with an alternative interface that would also allow programmatic access variability management could programmatically pass core asset and instance objects to a Customization Layer. The result of such an interaction is that after product derivation a Customization Layer has stored instances in a controlled way and evolution control can start.

## 5.6 Section summary

This section has described activities necessary in order to perform evolution control in a product line context. The next section will discuss how these activities can be realized internally by a Customization Layer. In particular it will be elucidated how common capabilities of configuration management systems can be used.





## 6 Interaction with Configuration Management

After a set of evolution control scenarios have been selected the next step is to realize a Customization Layer, which automates these scenarios with the help of configuration management. The effort required for this step depends on the desired degree of automation and also on the facilities of the underlying configuration management system. In general the goal should be to introduce a Customization Layer with the lowest impact possible for the organization. In that way existing investments in configuration management can be preserved.

Configuration management systems (CMS) offer a broad spectrum of functionality that can be taken into account when implementing a Customization Layer. In the one extreme case a Customization Layer can be considered as a database and process management system that on the one hand fully manages core and product assets and the other hand fully controls the evolution control processes. This extreme situation can become necessary if the underlying configuration management system offers limited or no functionality (for example if a regular file system is used as configuration management system). On the other extreme a Customization Layer may become obsolete if a special-purpose configuration management system is developed from scratch for the purposes of product line evolution control.

In the optimal case a Customization Layer will exploit and encapsulate existing CMS functionality. A Customization Layer should take over functionality for those evolution control scenarios, for which the CMS does not provide adequate support. The adoption of a Customization Layer has low impact if the functionality of existing CMS is reused. This is likely to happen if the CMS is equipped with a rich spectrum of functionality. On the other hand the impact is high if the Customization Layer takes over functionality although it could be achieved by exploiting existing CMS functionality.

In the following paragraphs a top-down approach will be followed in order to map Customization Layer scenarios to CMS functionality. The latter will be detailed in a subsequent section. For each scenario implementation guidelines will be provided addressing the following questions:

- What is the necessary CMS functionality to implement the scenario?
- What alternative CMS functionality can be used?

- What workarounds are applicable?

A table will be used for each scenario in order to break down the scenario implementation steps and to map them to CMS functionality and workarounds. Some steps will be mapped as optional in case their execution depends on the scenario input parameters or on other conditions. Furthermore for each step the corresponding interactions between Customization Layer (CL), CMS and variability management (VM) will be indicated.

As discussed in sections 3.8 and 3.9 evolution control scenarios are directly related with evolution control activities and processes. This can be supported by providing implementation guidelines in terms of Xtext. For space reasons guidelines are provided exemplarily in this format in Appendix A.

Apart from that scenarios in the following will be presented in a different order than in section 5 in order to avoid forward references.

## 6.1 Implementation Guidelines

### 6.1.1 Guidelines for version management scenarios

#### *createCoreAsset (→ section 5.1.2)*

| <pre>void createCoreAsset(String sourceLocation,                     String targetLocation,                     String templateLocation,                     String depth)     throws CoreAssetCreationException</pre> |                 |                             |                                |                      |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------|-----------------------------|--------------------------------|----------------------|
| Step                                                                                                                                                                                                                   | Interactions    | Necessary CMS functionality | Alternative CMS functionality  | Workaround           |
| «optional»<br>1. Populate target location                                                                                                                                                                              | CL → CMS        | Basic version management    | -                              | -                    |
| «optional»<br>2. Apply template                                                                                                                                                                                        | CL → CMS        | Basic version management    | -                              | -                    |
| 3. Mark core assets                                                                                                                                                                                                    | CL → CL         | Basic version management    | Properties, Naming conventions | Special-purpose file |
| 4. Set state                                                                                                                                                                                                           | CL → CMS        | State management            | Properties                     | Special-purpose file |
| «optional»<br>5. Set reuse contract                                                                                                                                                                                    | CL→VM<br>CL→CMS | Intentional Versioning      | Properties                     | Special-purpose file |

Table 10: createCoreAsset guideline

Steps 1 and 2: The first two steps consist in the creation and the population of the target repository location with the core asset contents. If the core asset to be created is not empty its contents are to be stored in the target location according to the input depth. On the other hand if the core asset is empty and a template is provided the target location is populated according to the template. Furthermore the target location should be relative to the process that is assigned to this scenario (see section 3.8). Result of this step is a set of configuration items that pertain to the input core asset. For this step standard version management functionality is sufficient.

Step 3: In the next step the newly created configuration items have to be marked as items that belong to a core asset. The way of marking items may vary according to the capabilities of the underlying CMS. A generally applicable approach is to put a mark (e.g. "[CL]" or another mark that cannot be used by accident by direct users of the CMS) in the message that is used during the commit operation. In most CMS systems obtaining the list of commit messages, which pertain to a repository or repository location, is an efficient operation. Therefore, to locate core assets with this approach, a list of commit messages can be obtained and then parsed in order to find the marks. Other possibilities to consider are properties or naming conventions. The latter refer to using specific configuration item names in order to mark core assets. As a workaround there is also the possibility to store a special file in the target location (e.g. an XML file) that can be parsed by the Customization Layer. That file can inform the Customization Layer about the types of assets stored. Disadvantage of this solution is that a common CMS user cannot obtain the information. With the other solutions, Customization Layer information is stored by means of common CMS facilities. This enables common CMS users to access the information as well.

Step 4: In the fourth step the Customization Layer sets the state of the newly created core asset to "not released" as described in section 4.2.1. State management functionality is used to this end. If state management is available the CMS has simply to assign the new state to the related configuration item. As an alternative a state property can be assigned to the configuration item. Finally a workaround would be to maintain a special-purpose file in the same location as the core asset that describes the states of all assets in that location.

Step 5: In the last step the Customization Layer interacts with a variability management system, if the latter is available, in order to obtain the reuse contract for the core asset. As described in section 5.5.1 this involves the creation of a decision model (or other equivalent reuse contract). Subsequently the decision model is to be stored in the CMS. If the latter provides intentional versioning or similar approach (see also section 6.2.4) it might be possible to associate open variability decisions

directly with the core asset. However most modern CMS do not provide such support therefore properties can be used as an alternative. Different properties can map to different open decisions of the core asset. Each property value can be set to a representation of the variability type provided in the variability management system. For example a property "region" can be set to a value "{Europe, Asia}" denoting that the given core asset can be customized for Europe or Asia. Finally, the decision model for the given core asset or for all core asset in the target location can be extracted from variability management and stored as a special-purpose file.

### ***createProductAsset (→ section 5.2.2)***

| <pre>void createProductAsset(String sourceLocation,                         String targetLocation,                         String templateLocation,                         boolean isInstance,                         InstantiationStrategy iStrategy)                         throws ProductAssetCreationException</pre> |                 |                             |                               |                      |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------|-----------------------------|-------------------------------|----------------------|
| Step                                                                                                                                                                                                                                                                                                                        | Interactions    | Necessary CMS functionality | Alternative CMS functionality | Workaround           |
| 1. Select core assets to instantiate                                                                                                                                                                                                                                                                                        | CL              | -                           | -                             | -                    |
| «optional»<br>2. Instantiate core assets                                                                                                                                                                                                                                                                                    | CL→VM           | -                           | -                             | -                    |
| 3. Create instances                                                                                                                                                                                                                                                                                                         | CL → CMS        | Branching / Copying         | Build management              | Links                |
| «optional»<br>4. Store signed contract                                                                                                                                                                                                                                                                                      | CL→VM<br>CL→CMS | Intentional Versioning      | Properties                    | Special-purpose file |
| 5. Set state                                                                                                                                                                                                                                                                                                                | CL → CMS        | State management            | Properties                    | Special-purpose file |

Table 11: createProductAsset guideline

In case the product asset is not an instance but a product-specific asset (i.e. `isInstance = false`) the implementation guideline is similar to steps 1 to 3 of the core asset creation guideline. The following steps address the case where the product asset is an instance (i.e. `isInstance = true`)

Step 1: The first step consists in the selection of the core assets to instantiate. To this end the Customization Layer has to query the user.

Step 2: In case variability management is available this step invokes it in order to resolve variability decisions on the input core assets (see also section 5.5.2). The step results in the instance to be stored subsequently under CMS.

Step 3: The third step carries out the product asset creation. At this point a new development line has to be created and associated with the development lines of the input core assets. To this end branching functionality of the underlying CMS is necessary. This can range from sophisticated streaming and sharing and streaming functionality to simple branches (see also appendix A.4). It must however be noted that branching might be applicable only if the instance is derived from a single core asset. If core asset composition is applied (see also section 4.1.2) it can be sensible to use the copy functionality available in most CMS in order to copy all core assets to the target location. In every case it makes sense to store the original core assets to the target location and then to replace them with their derived instances. Thereby the traceability between core asset and instances is maintained as CMS typically keep track of branching and copying operations.

Special attention has also to be paid with the input instantiation strategy. Branching functionality suits better the deep strategy since it creates a branch of the core asset along with all its contained core assets. On the other hand copying functionality enables to better control which items to store in the instance location.

If instances are considered as derived artifacts that are not be changed any further build management can come into play. In this case the instances are typically binary files obtained through compilation of the core assets. If neither branching nor build management are applicable instance can be stored in the target location and related via links to the core assets.

Step 4: The fourth step is performed if variability management is available. In this case the set of resolved decisions are attached to the instance using the same mechanisms as described in step 5 of the core asset creation guideline.

Step 5: The final step consists in setting the state of the involved core assets to "reused" and the resulting instance to "isInstance" (see sections 4.2.1 and 4.2.2)

**removeCoreAsset (→ section 5.1.2)**

| <pre>void removeCoreAsset(String caID)     throws CoreAssetDeletionException</pre> |              |                             |                                |                      |
|------------------------------------------------------------------------------------|--------------|-----------------------------|--------------------------------|----------------------|
| Step                                                                               | Interactions | Necessary CMS functionality | Alternative CMS functionality  | Workaround           |
| 1. Retrieve core asset based on ID                                                 | CL (→ CMS)   | -                           | -                              | -                    |
| «optional»<br>2. Remove markers                                                    | CL → CMS     | Basic version management    | Properties, Naming conventions | Special-purpose file |
| 3. Remove instance associations                                                    | CL → CMS     | Properties                  | Basic version management       | Special-purpose file |
| 4. Set state                                                                       | CL → CMS     | State management            | Properties                     | Special-purpose file |
| 5. Remove asset                                                                    | CL → CMS     | Basic version management    | -                              | -                    |

Table 12: removeCoreAsset guideline

Step 1: The first step consists in the retrieval of the core asset details including any associated instances. To this end underlying CMS functionality comes into play as described in step 3 of the core asset creation scenario.

Step 2: In the next step any markers assigned to the core asset have to be removed so that the asset is not identified as core asset any more. However depending on the mechanism chosen for the creation of the core asset removal of such markers might not be directly possible. For example if commit messages have been used, it might not be possible to change them or to remove the corresponding version. In this case additional markers might be necessary denoting the given core asset is not active anymore.

Step 3: In the third step any association between the core asset and instances have to be also removed. In the case of sharing or streaming it is usually possible to unshare items or to relocate streams. This step might become more cumbersome if conventional branching or copying functionality has been used as the history of operations cannot be easily changed. In this case it might be necessary to use again markers indicating that instances and core assets are not related any more. Such markers can be stored in terms of properties. As an alternative a new

branch version can be created that holds the marker in its commit message. Finally a workaround is again to use a special-purpose file denoting the removed associations. Furthermore, if no more core assets are related to the affected instances markers can be used to indicate that the given instance is turned into a product-specific asset.

Step 4: The rebase state of instances might have to be updated if associations with core asset were removed.

Step 5: In the last step the CMS is invoked in order to remove the configuration items of the core asset from the repository. This usually leads to a new version of the corresponding repository location that does not contain the configuration items anymore.

#### ***removeProductAsset (→ section 5.2.2)***

| void removeProductAsset(String paID, boolean detachOnly)<br>throws ProductAssetDeletionException |              |                             |                                |                      |
|--------------------------------------------------------------------------------------------------|--------------|-----------------------------|--------------------------------|----------------------|
| Step                                                                                             | Interactions | Necessary CMS functionality | Alternative CMS functionality  | Workaround           |
| 1. Retrieve product asset based on ID                                                            | CL (→ CMS)   | -                           | -                              | -                    |
| «optional»<br>2. Remove markers                                                                  | CL → CMS     | Basic version management    | Properties, Naming conventions | Special-purpose file |
| 3. Remove instance associations                                                                  | CL → CMS     | Properties                  | Basic version management       | Special-purpose file |
| 4. Set state                                                                                     | CL → CMS     | State management            | Properties                     | Special-purpose file |
| 5. Remove asset                                                                                  | CL → CMS     | Basic version management    | -                              | -                    |

Table 13: removeProductAsset guideline

Removal of a product asset follows a similar approach as the removal of core assets and hence will not be detailed any further. A difference arises from the detachOnly parameter. The latter makes the second step in the guideline optional.



**modifyCoreAsset (→ section 5.1.2)**

| void modifyCoreAsset(String caID)<br>throws CoreAssetModificationException |                 |                                                 |                                |                      |
|----------------------------------------------------------------------------|-----------------|-------------------------------------------------|--------------------------------|----------------------|
| Step                                                                       | Interactions    | Necessary CMS functionality                     | Alternative CMS functionality  | Workaround           |
| 1. Retrieve core asset based on ID                                         | CL (→ CMS)      | -                                               | -                              | -                    |
| 2. Modify core asset                                                       | CL              | -                                               | -                              | -                    |
| 3. Commit changes to core asset and affected product assets                | CL → CMS        | Basic version management, Copying functionality | Properties, Naming conventions | Special-purpose file |
| 4. Set state                                                               | CL → CMS        | State management                                | Properties                     | Special-purpose file |
| «optional»<br>5. Set new reuse contract                                    | CL→VM<br>CL→CMS | Intentional Versioning                          | Properties                     | Special-purpose file |

Table 14: modifyCoreAsset guideline

Step 1: The first step consists as with previous guidelines in the retrieval of the core asset details including any associated instances.

Step 2: In the next step the Customization Layer enables the user to change core asset attributes (name, configuration items, instances and process).

Step 3: In the third step changes are to be committed in the CMS. Different scenarios are possible: (a) The simplest scenario arises when the user changes the name of the core asset; in which case the CMS will be ask to commit the new name (if the core asset has the same name as its configuration item; in the other case only the Customization Layer model will be updated). (b) If the user changed the associated configuration item the latter has to be marked accordingly (see core asset creation guideline) (c) If the user changed the process the asset is to be copied to the repository location pertaining to the input process. To this end the CMS can be invoked to create a new core asset with the contents of the modified asset or to copy the modified asset to the new location. (d) If the user changes the associated instances the latter have also to be

marked accordingly (see create, remove product asset guidelines). It might be necessary to mark previously associated instances as product-specifics if there are no more core asset associations.

Step 4 and 5: After modification the states (rebase, reuse, integration) of the affected core and instance assets might have to be updated. The same applies to the reuse contract of the core asset. Corresponding steps of the create core asset guideline provide more details.

The modification operation can be also useful for relocation (i.e. move) of configuration items. In this case the relocation of configuration item can be realized by modifying the associated core asset.

### ***modifyProductAsset (→ section 5.2.2)***

| void modifyCoreAsset(String caID)<br>throws CoreAssetModificationException |              |                                                 |                                |                      |
|----------------------------------------------------------------------------|--------------|-------------------------------------------------|--------------------------------|----------------------|
| Step                                                                       | Interactions | Necessary CMS functionality                     | Alternative CMS functionality  | Workaround           |
| 1. Retrieve product asset based on ID                                      | CL (→ CMS)   | -                                               | -                              | -                    |
| 2. Modify product asset                                                    | CL           | -                                               | -                              | -                    |
| 3. Commit changes to product asset and affected core assets                | CL → CMS     | Basic version management, Copying functionality | Properties, Naming conventions | Special-purpose file |
| 4. Set state                                                               | CL → CMS     | State management                                | Properties                     | Special-purpose file |

Table 15: modifyProductAsset guideline

The modification of a product asset operates in a similar way as the modification of a core asset. Among the other product asset attributes the user has the possibility to change the associations of the given product asset to core assets. The marking functionality described previously has to be applied. Thereby a product-specific asset might turn into an instance. Finally the corresponding rebase and reuse states might have to be updated.

**integrateCoreAsset (→ section 5.1.2)**

| void integrateCoreAsset(String caID)<br>throws CoreAssetIntegrationException |              |                             |                               |            |
|------------------------------------------------------------------------------|--------------|-----------------------------|-------------------------------|------------|
| Step                                                                         | Interactions | Necessary CMS functionality | Alternative CMS functionality | Workaround |
| 1. Retrieve core asset based on ID                                           | CL (→ CMS)   | -                           | -                             | -          |
| «optional»<br>2. Merge from instances and mark integration                   | CL → CMS     | Basic version management    | -                             | -          |

Table 16: integrateCoreAsset guideline

Step 1: The first step consists in the retrieval of the core asset details including any associated instances. To this end underlying CMS functionality comes into play as described in step 3 of the core asset creation scenario.

Step 2: The implementation of the core asset integration involves two strategies as described in section 5.1.2. In the a posteriori case the CMS connector assumes the last changes as integration changes and puts a mark in the last core asset version accordingly (see previous guidelines). In the session-based case the implementation has to locate all instances of the core asset and initiate a merging. The result of the merging can be then marked as integration merge.

**rebaseProductAsset (→ section 5.2.2)**

| void rebaseProductAsset(String paID)<br>throws ProductAssetRebaseException |              |                             |                               |            |
|----------------------------------------------------------------------------|--------------|-----------------------------|-------------------------------|------------|
| Step                                                                       | Interactions | Necessary CMS functionality | Alternative CMS functionality | Workaround |
| 1. Retrieve product asset based on ID                                      | CL (→ CMS)   | -                           | -                             | -          |
| «optional»<br>2. Merge from core assets and mark rebase                    | CL → CMS     | Basic version management    | -                             | -          |

Table 17: rebaseProductAsset guideline

The implementation of the rebase product asset scenario follows the same schema as the integrateCoreAsset. In this case when a rebase session is executing the direction of the merge is the opposite, i.e. from the core assets towards the instances.

### 6.1.2 Guidelines for change management scenarios

#### *createCoreAssetChangeRequest (→ section 5.1.1)*

| <pre>String createCoreAssetChangeRequest(String[] caID,                                    boolean synchronizeInstances)     throws CoreAssetCRCreationException</pre> |              |                             |                                  |                      |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------|-----------------------------|----------------------------------|----------------------|
| Step                                                                                                                                                                   | Interactions | Necessary CMS functionality | Alternative CMS functionality    | Workaround           |
| 1. Retrieve core assets based on ID                                                                                                                                    | CL (→ CMS)   | -                           | -                                | -                    |
| 2. Create change request                                                                                                                                               | CL → CMS     | Basic change management     | -                                | -                    |
| 3. Set state of core assets                                                                                                                                            | CL → CMS     | State management            | Properties                       | Special-purpose file |
| «optional»<br>4. Find instances of core assets                                                                                                                         | CL → CL      | Basic version management    | Properties<br>Naming conventions | Special-purpose file |
| «optional»<br>5. Create change requests for instances                                                                                                                  | CL → CMS     | Basic change management     | -                                | -                    |
| «optional»<br>6. Relate instance change requests to core asset change requests                                                                                         | CL→CMS       | Ticket hierarchy            | Links                            | Properties           |

Table 18: createCoreAssetChangeRequest guideline

Step 1: The first step in this scenario involves the retrieval of the core asset details based on the input ID. This is necessary in order to obtain the product line engineering process the asset belongs to. The latter is

required in order to store the new change request in the corresponding location in the change management system. At this point it might be necessary to interact with the CMS in this step in order to generate the core asset detail data from the CMS. The CMS mechanisms used for core and product asset creation have to come into play in this case (see also section 6.1.1).

Step 2: Given the core asset details CMS is invoked to create the change request in the underlying change management system. At this point it is important to identify the right storage location. In so doing it can be ensured that Customization Layer change requests are well organized. One way of achieving that is to map product line engineering processes to corresponding entities like projects, queues or groups within the change management system. It must however be ensured that such entities can be interrelated.

The result of this creation is a configuration item that corresponds to the change request object. The latter shall then be associated with the configuration items that correspond to the input core assets. In order for users of the CMS to recognize that the new change request has been created by a Customization Layer it is recommended that the CMS connector adds a marker to the change request. There are different ways of accomplishing that depending on the CMS capabilities. Such a marker can be added for example in terms of a label or in terms of a comment.

Step 3: Upon creation of a change request the related core assets change into the state “changes pending” (see Core Asset Change Management in section 4.2).

Step 4 and 5: In the next two steps the Customization Layer is retrieving the instances of the core assets and subsequently change requests are created. Instances are identified based on the mechanism used for the creation of core and product assets (for more details on that see section 6.1.1). This is an optional step and has to be undertaken only if change requests are to be created for the instances as well (`synchronizeInstances = true`). This can be useful if the resolution of the core asset change request requires approval from the corresponding application engineering processes. In this case instance change requests have to be created and resolved before the core asset change request can be resolved.

Step 6: In the last step the instance change requests are to be connected to the core asset change request. If a ticket hierarchy approach [UKR09] is available this operation is straightforward. Alternatively the tickets can be associated using linking functionality available in most change management systems. If linking is also not possible a workaround would

be store associations in terms of custom or standard change request properties.

An interesting point in this scenario is that the creation of a core asset change request may lead to creation of instance change requests. The latter however may subsequently lead to change requests for further core assets that also relate to the instances. With such an approach the creation of core asset change request leads to a ripple effect that spreads across instances and core assets. Therefore the implementation has to query the user about the desired extent of change propagation. In every case the CMS connector shall maintain a history of all visited core assets and instances so that duplicate change requests can be avoided.

***createProductAssetChangeRequest (→ section 5.2.1)***

| <pre>String createProductAssetChangeRequest(String[] paID,                                      boolean synchronizeCoreAssets)     throws ProductAssetCRCreationException</pre> |              |                             |                               |                      |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------|-----------------------------|-------------------------------|----------------------|
| Step                                                                                                                                                                            | Interactions | Necessary CMS functionality | Alternative CMS functionality | Workaround           |
| 1. Retrieve product assets based on ID                                                                                                                                          | CL (→ CMS)   | -                           | -                             | -                    |
| 2. Create change request                                                                                                                                                        | CL → CMS     | Basic change management     | -                             | -                    |
| «optional»<br>3. Find related core assets                                                                                                                                       | CL → CL      | Basic version management    | Properties Naming conventions | Special-purpose file |
| «optional»<br>4. Create change requests for core assets                                                                                                                         | CL → CMS     | Basic change management     | -                             | -                    |
| «optional»<br>5. Relate change requests                                                                                                                                         | CL→CMS       | Ticket hierarchy            | Links                         | Properties           |

Table 19: createProductAssetChangeRequest guideline

As shown in Table 19 the creation of change requests for product assets follows a similar scheme as for core assets. Therefore the associated steps will not be detailed any further. The only differentiation arises

when the product asset to create the change request for is not an instance but a product-specific asset. In this case steps 3 and 4 are not considered.

### 6.1.3 Guidelines for status accounting scenarios

All status accounting scenarios operate on information that has been stored via the change and version management scenarios. Therefore, the implementation of the status accounting scenarios can be based on the strategies discussed above. Hence, the implementation of the status accounting scenarios is discussed in a condensed way in the following:

- `showChangeRequests`

Retrieval of the list of change requests can be achieved by querying the underlying change management systems.

- `showCoreAssets`

In order to show the available core assets a Customization Layer has to make use of the selected marking strategy. For example if commit messages have been, the Customization Layer has to query the history of the repository in order to identify this marks.

- `showCoreAssetInstances`

The implementation of this scenario also depends on the version functionality that has been used for creating instances. In every case it can be retrieved which development lines has been created off the core assets and based on the marks it can be identified whether these development lines relate to instances.

- `showCoreAssetChanges`

For this scenario it is necessary to access the history of the corresponding configuration item and possibly to identify the last synchronization point based on the used marks. Subsequently the list of changes can be delivered.

- `showProductAssets`

This scenario is also to be implemented based on the marking strategy at hand

- `showInstanceCoreAssets`

This scenario also depends on the versioning functionality. Given an instance the implementation must first retrieve for development lines the instance originates from and must then check for marks that designate core assets.

- `showProductAssetChanges`

This scenario requires a similar strategy as the `showCoreAssetChanges` scenario

## 6.2 CMS functionality and the Customization Layer

This section details CMS functionality referred to in the previous and elucidates the possible interactions with a Customization Layer.

### 6.2.1 Main Functionality Blocks

Figure 65 illustrates the main functionality blocks of configuration management systems adapted from a renowned survey [Da90]. Although this survey has been written in 1990 it still covers the range of functionality of contemporary configuration management systems. The process and team blocks are considered of major importance in the following picture and therefore they are connected to all other functionality blocks.

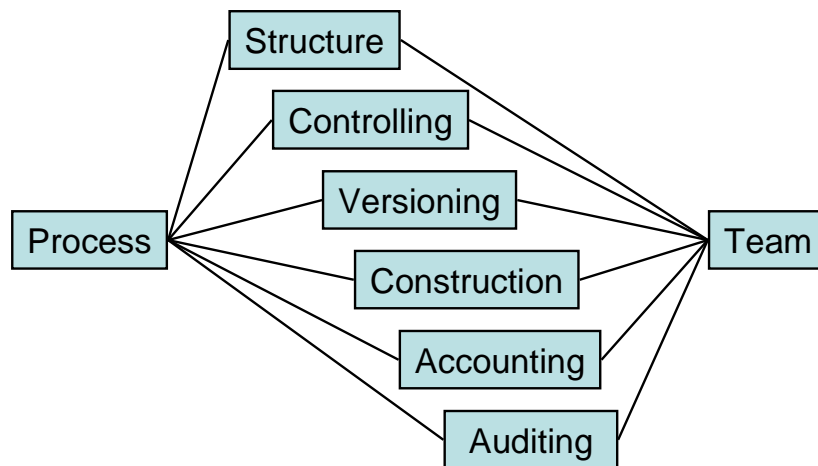


Figure 65: Main CMS functionality blocks

### 6.2.2 Structuring

Structuring functionality supports modeling configuration items and attributed interrelations at different levels of abstraction. Various types



of relations can be supported, such as hierarchy, aggregation and dependency relations or links as known from other disciplines (e.g. software architecture modeling or computer-aided manufacturing). Specialized relationships such as the consistency and compatibility relationships introduced by the Configuration Management Assistant (CMA) [PF89] are also conceivable. Two versions of different configuration items are considered consistent if they can be included in a configuration without violating its validity. On the other hand two versions of a configuration item are compatible if they can be equally used in a configuration. CMA also introduces the concept of inheritable dependencies between configuration items. When a version of configuration item is included in a configuration and the item is related via inheritable dependencies to other items, versions of the other items must be included as well. In CMA configuration items and versions are called objects and instances respectively.

With most modern configuration management systems structuring functionality is however generally weak. A solution that is proposed is to combine product management systems, which do provide powerful structuring facilities, with configuration management [CAD03].

Given the basic asset model (section 4.1) a Customization Layer establishes relationships between core assets and their instances. Structuring functionality can be used in this case in order to model these relationships in the CMS. A drawback however with many CMS can be that such relationships do not facilitate synchronization. In other words the CMS will not automatically propagate changes or notifications between related items. In this case, the usage of version management should be preferred at this point. Version management allows creation of relationships (e.g. through branches), which strongly facilitate change propagation.

The retrieval of relationship information is crucial for a Customization Layer. With versioning this retrieval requires additional effort, which can be avoided by structuring facilities. However synchronization is also crucial for evolution control. With version management synchronization is given; with structuring it might be necessary that a Customization Layer implements the synchronization. Therefore, there is tradeoff to be carefully analyzed with a given CMS regarding the structuring and synchronization facilities. In the subsequent discussion a Customization Layer opts for a solution based on version management, as this is the most common in CMS.

### 6.2.3 Controlling

Controlling functionality is always available in configuration management and according to [Da90] mainly addresses issues of change

management (cf. section 3.4.1). In other words controlling enables creating, processing and monitoring change requests, bugs and problem reports.

A Customization Layer can exploit controlling functionality in order to realize change management scenarios. The implementation of these scenarios requires basic change management functionality, which is available with every CMS. The ticket hierarchy approach as discussed in the createCoreAssetChangeRequest scenario may not be available with some (older) systems. In this case the implementation will have to operate on the basic mechanisms. For example, custom change request fields can be used in order to define associations between change requests.

#### **6.2.4 Versioning**

Versioning (cf. section 3.4.2) is a major functionality in every configuration management system. It enables creating and managing versions, branches, and configurations (section 1.3.3).

Branching can be generally employed in order to manage the synchronization between family and application engineering. Core assets can be placed in branches and instances in other branches created off the core asset branches. With the encapsulation provided by the Customization Layer, the complexity that appears when multiple branches arise (see also section 1.3) can be hidden.

Versioning and branching will be also used for conventional evolution control activities that do not directly relate to synchronization between family and application engineering. For that reason tagging and documentation functionality inherent to versioning should come into play. Tagging (or labeling) can be used to mark branches. Likewise documentation can be used to describe versions (e.g. in terms of commit messages). These features can also be used by a Customization Layer to automatically mark the results of operations performed. When for example a user initiates the creation of a core asset with the Customization Layer a version can be created with a particular commit message that can be later recognized by the layer but also by users, who directly access the CMS.

The advantage of using branches at this point is the synchronization support, which is commonly available. Configuration management systems enable propagating changes between branches and this can be used for the synchronization of family and application engineering efforts. In some cases synchronization can be even tracked. That means that the versioning subsystem knows when branches have exchanged changes without requiring the users to explicitly describe synchronization

activities in terms of commit messages. This feature can simplify the automated coordination between core assets and instances significantly.

An interesting scenario arises however when core assets and instances are to be removed (removeCoreAsset and removeProductAsset scenarios respectively). In this case it is actually necessary to change existing relationships between branches. With some configuration management systems (e.g. AccuRev [ACC10]) this is possible by replacing branches in a branch hierarchy. In other systems this can be accomplished by changing the versioning history (i.e. changing the commit message used when creating a branch can detach an instance from a core asset or indicate that a configuration item stops being a core asset). Alternatively a special attribute can be attached to a configuration item in order to indicate this kind of information.

Some configuration management systems provide specialized support for branching, which can be beneficial for branch management with the help of the Customization Layer. The following list discusses two examples from well-known modern configuration management systems.

- sharing: This functionality is offered by some systems like the Team Foundation Server [URL8] or StarTeam [URL20]. Sharing enables managing reusable configuration items, which can be “shared” across different users and projects. Sharing enables reusers to be automatically notified when new versions of reusable items are available and to obtain the changes at will. In many systems the sharing functionality comes into play when the reuse of assets is initiated. In other words it is often not necessary to mark shared assets as such during their creation.
- streams: Streams are provided by AccuRev [ACC10] and constitute a sophisticated implementation of the branching concept. Streams can be considered as branches; however change propagation is facilitated to a great extent. Hierarchies of streams can be built and modified at will. Moreover propagating changes from parent to children streams and vice versa can be configured individually.

Versioning support in configuration management can be distinguished between extensional and intentional versioning [CW98]. Extensional versioning, which is common in most implementations, enumerates the versions of a configuration item in an ascending order. Each version can be therefore identified by a unique alphanumeric representation. On the other hand intentional versioning uses logical terms for version identification. In other words a particular version of an item can be described as logical predicates (e.g. operating system = windows AND market = Europe).

When implementing a Customization Layer the usage of intentional versioning should be considered, if possible. In this case the usage of predicates provides for better traceability between variability management and the configuration management system than with extensional versioning. As discussed in section 4.4 instances of core assets are derived by resolving variability decisions and variation points inherent to core assets. Therefore an instance of a core asset can be identified by the decisions that have been made during derivation. With intentional versioning when a core is instantiated a new version of it can be created that is marked with the corresponding resolved decisions.

Although intentional versioning is not available in modern CMS, in most of the cases it is possible to implement it in combination with extensional versioning. In this case the metadata functionality, which is common in most systems, can be employed. The predicates that characterize a particular version can be attached to a version in terms of special-purpose attributes. Since however these attributes cannot be processed automatically, if intentional versioning is not available, the Customization Layer has to take over this task.

Versioning also includes managing states of configuration items. States characterize versions with respect to the stage in the development process that yielded the versions (for example under development, tested, etc.). In this regard, state management functionality should be used in order to deal with the different states of core assets and instances. If explicit state management is not available, metadata can again be used to store the states of configuration items. Some CMS also provide support for tags, i.e. special markers that can be assigned to configuration item versions. Tags can also be useful for state management. Finally, some CMS do not provide any state management at all. In this case states can be managed by special-purpose files or they be mapped to repository locations, which hold configuration item versions in specific states.

### **6.2.5 Construction**

Construction functionality is responsible for the compilation of configuration items in order to obtain executable systems. In this regard it usually involves managing, automating and monitoring the compilation process. Executable or binary items can be seen in some cases as instances of core assets. Although binary items are usually not subject to evolution control a Customization Layer might need to invoke construction functionality to instantiate core assets. This might be necessary in production lines (section 3.6.10), namely in product lines that consider instance assets as transient.

However, construction functionality can be particularly interesting for the monitoring tasks of product line evolution control. Modern build management systems often implement the concept of continuous integration [Duv07] illustrated in Figure 66,

A continuous integration server runs in parallel to a configuration management system (or server), monitors operations performed therein, notifies affected users and initiates further measures if configured so. Usually the goal is to facilitate quality assurance by invoking compilations and test runs when execution of particular operations or when transition to particular states are identified in the repository. A typical scenario is to invoke test runs when a user loads a new version of a subsystem from a local working copy to the repository and then to notify the users about test results. In that way it is ensured that changes from different users are continuously “integrated” into a consistent system.

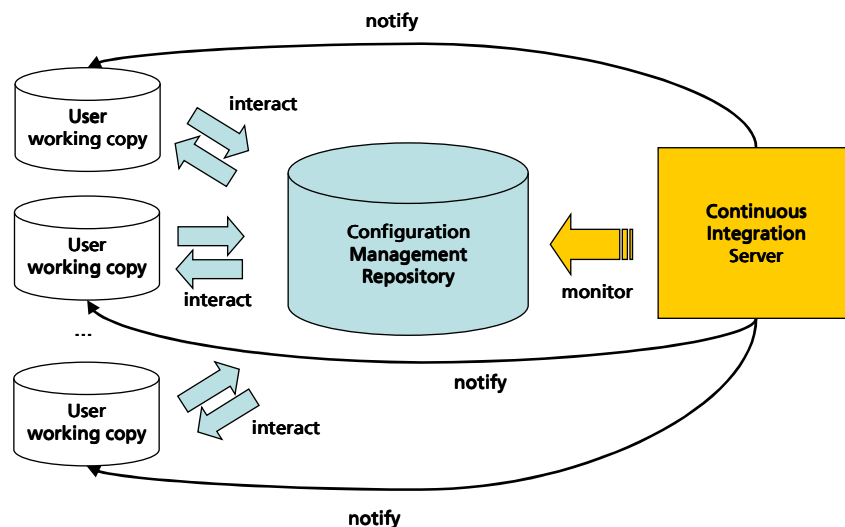


Figure 66: Continuous integration paradigm adapted from [Duv07]

In the context of a Customization Layer core and product asset monitoring tasks (i.e. status accounting scenarios in section 5.3) can be facilitated if continuous integration is in place. To this end special monitoring routines can be setup when assets are created or changed. These routines can check for instantiations, changes and continuously store the retrieved data, so that the status accounting scenarios can access them. In this case it can be made possible that a concrete incarnation of the basic asset model (section 4.1) is continuously kept under persistent storage.

Monitoring routines can also perform further analyses. For example a continuous integration server can be programmed to continuously

compare code bases of product line members to identify similarities that may indicate new candidates for core assets. Based on this information the corresponding changes can be requested. As a continuous integration server is usually a dedicated machine that does not interfere with normal development such heavy-weight analyses can be performed without any efficiency overhead.

An alternative mechanism to a continuous integration server is provided in some configuration management systems in terms of special scripts (e.g. hook scripts in Subversion [URL6]). The latter are executed in the context of configuration management operations, for example after creation of a new configuration item and enable to perform actions based on input parameters, user rights, state of items etc.

#### **6.2.6 Accounting and Auditing**

Accounting functionality (cf. section 3.4.3) accesses a configuration management repository and provides statistics, information about the system status and generates reports [Da90]. In this regard the functionality overlaps the continuous integration functionality discussed in the previous section. In every case the status accounting scenarios discussed in section 5.3 can be realized either by using continuous integration functionality or by retrieving basic evolution data from the repository. The latter can be retrieved through the auditing functionality, which is described in the next paragraph.

According to [Da90], auditing allows accessing the history of all changes and establishing traceability links between configuration items. In terms of Customization Layer traceability functionality is similar to structuring: It allows establishing explicit relationships between items. Also here it should be evaluated whether the synchronization of related items is facilitated. On the other hand, history functionality is important and can be used in an automated way by a Customization Layer. That means that a Customization Layer can navigate through the history of core assets and instances in order to draw certain conclusions. Instances of core assets can be obtained, for example, by navigating through the versions of a core asset and by finding branches that are marked accordingly. Or, given an instance asset the Customization Layer can navigate through its history in order to identify from which core asset it has been created (i.e. branched off).

#### **6.2.7 Team**

Team functionality facilitates the cooperation of engineers in a joint development effort. According to [Da90] team functionality mainly involves different sophistication levels of workspace management.

Workspaces contain private working copies of configuration items or of particular versions thereof, which can be processed in isolation. Workspace functionality possibly enables engineers to create private versions and branches and to synchronize with other workspaces or with a central repository.

For a Customization Layer the synchronization functionality is crucial but it overlaps with the versioning functionality. A Customization Layer implementation can benefit however from workspace functionality, if it is possible to define family and application engineering workspaces. In that case the workspace functionality would allow workspace owners to invoke only the particular evolution control scenarios applicable to their role. Apart from that, and for the scenarios described in section 5, no additional workspace functionality is necessary.

### 6.2.8 Process

Some configuration management systems enable modeling and enacting development processes similar to workflow systems. In that sense users are assigned roles as well as activities and the lower level configuration management operations such as versioning or construction can be traced back to the relevant activities.

The scenarios described in section 5 are isolated evolution control activities and their realization does not require workflow support. However scenarios could be also orchestrated in order to define evolution control workflows. For example, a workflow can be defined that starts with the creation of a change request, proceeds with the modification of a core asset and ends with the integration of a core asset with its instances. Process functionality can be used in order to realize such workflows. To that end it is however necessary that Customization Layer scenarios are implemented in a way (e.g. as services according to the principles of service-orientation) that is compatible with the process functionality. The latter must be able to recognize the scenarios as activities that can be orchestrated.

## 6.3 Section summary

This section has discussed configuration management functionality that can be used for the creation of a Customization Layer and implementation guidelines have been provided for each of the evolution control scenarios. Next section will discuss a framework for the implementation of Customization Layer.

# 7 A Customization Layer framework

This chapter presents a framework that facilitates the implementation of Customization Layer consoles based on Java technology. The framework supports the creation of console applications that resemble a command prompt. In the lower part of the application window users are able to issue evolution control commands. In the upper part a text field lists the output of commands and enables scrolling over the execution history. Figure 67 provides a screenshot for a Customization Layer prototype that has been implemented with the help of the framework and for a selection of evolution control scenarios. As shown in Figure 67, for simplicity the scenario names are slightly different that the scenario names used in section 5.

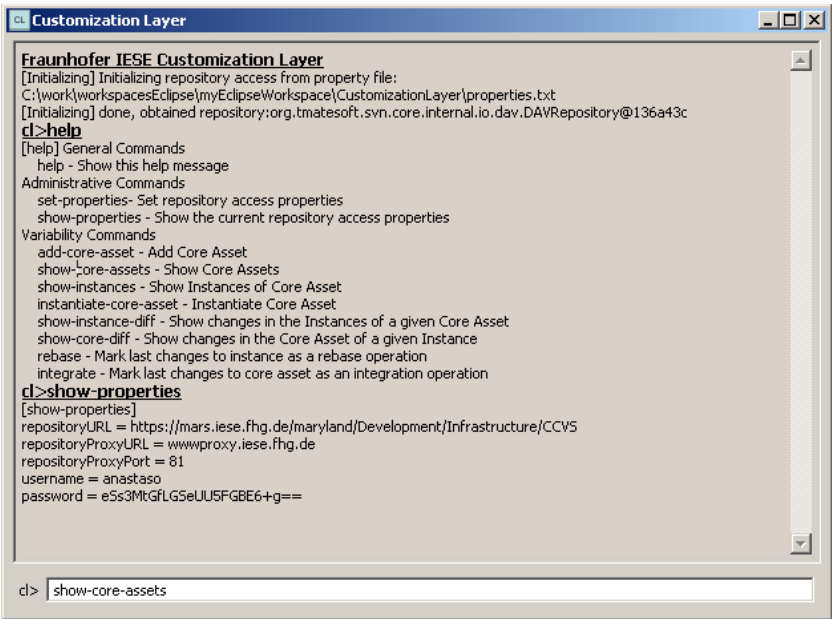


Figure 67: Customization Layer prototype screenshot

The basic structure of the framework is depicted as a UML class diagram in Figure 67. The class *CustomizationLayer* is responsible for the implementation of the required evolution control activities. To this end the class uses configuration management operations contained in the *ConfigurationManagement* package (abbreviated as *cm*). The framework also provides a user interface (*UserInterface* package, abbreviated as *gui*) that among other things enables managing the evolution control



commands issued by the product line engineers. To this end the user interface interacts with a *CommandParser* that defines a grammar for the accepted evolution control commands and parses the user input accordingly. This grammar is meant for the lexical and syntactical analysis of the console commands and is not to be confused with the Xtend grammars used thus far.

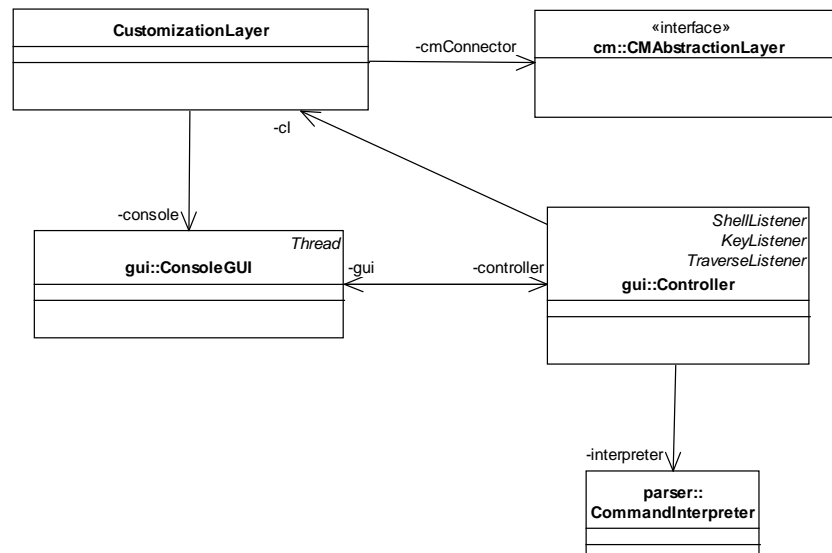


Figure 68: Framework structure

The framework has been implemented in Java and with the help of the Standard Widget Toolkit [URL4] and the JavaCC parser generator [URL10].

## 7.1 Customization Layer

The class *CustomizationLayer* implements the selected evolution control scenarios. In terms of the Model-View-Controller (MVC) pattern [Re79] it assumes the role of the model. To this end the class has two associations:

- *CMAbstractionLayer*: This is the interface that defines the configuration management operations used by the *CustomizationLayer*. The interface is realized by connectors, which take over the interaction with concrete configuration management systems. This interface along with implementation guidelines can be generated out of the selected evolution control scenarios and with the help of the Xtend tool chain (see section 6).

- *ConsoleGUI*: This is the main class of the graphical user interface and plays the role of the view in the MVC pattern. *CustomizationLayer* references this class in order to send output of evolution control tasks to the user interface as well as to interact with the user.
- *Controller*: This class plays the role of the controller according to the MVC pattern. In other words, it manages the classification and execution of the commands issued by the user.

## 7.2 User Interface

The user interface package (gui) contains classes (see Figure 69) that enable users to input and manage the execution of evolution control commands. The class *ConsoleGUI* is implementing the user interface elements shown in Figure 67. The user interface consists of two main widgets, the input and the output widget. The former is an editable text field, in which the evolution control commands are entered. The latter is a non-editable text field that shows the output of the commands and enables the user to scroll over the execution history and also to perform copy and paste text operations.

The abstract class *Command* follows some (e.g. undo or redo is not supported as this is unusual in the context of configuration management) of the ideas of the command design pattern proposed in [GHJ+95] and provides a framework for the definition, execution and management of evolution control commands. Therefore all operations described in section 7.1 are implemented as subclasses of *Command*. Furthermore some additional helper commands are available such as the help command that describes the function and syntax or the set-properties command that tells the Customization Layer to show the configuration management property dialog.

The *Command* class inherits from the Java class *Thread*. This enables commands to be executed asynchronously and also to be interrupted if necessary. This can be useful with long-running commands or in case wrong commands arguments have been passed.

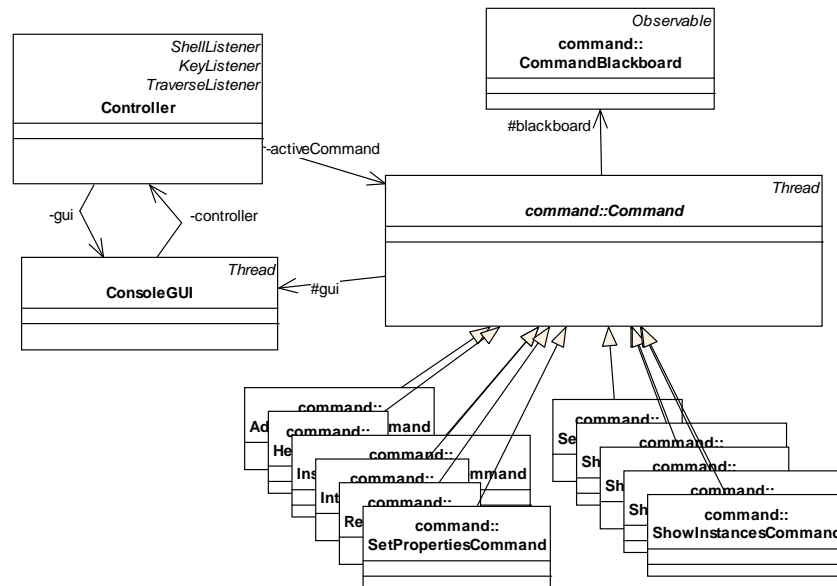


Figure 69: User interface classes and dependencies

The class *CommandBlackboard* is a simple extension of the Java class *Observable*. When a command is being executed it usually adds an observer to *CommandBlackboard*. In the current implementation the observer is contained in the configuration management connector. When the user requests a command to stop execution, the controller tells the command to stop. Subsequently the command invokes the *CommandBlackboard*, which in turn notifies the configuration management connector. The latter can then respond to the interruption requested by the user and gracefully stop any running configuration management operations.

In the case of the Subversion version management system for example, the corresponding programming interface SVNKit [URL5] provides special handler classes that can deal with such interruptions. These handlers, when registered, are invoked every time a particular Subversion event takes place. For example when a commit is about to take place the handler is being notified. Upon notification the handler has the possibility to check whether the current configuration management operation is to be cancelled. To this end a special method *checkCancelled* is provided. The method checks whether the current configuration management event should be stopped and if so it raises a special exception *SVNCancelException*. This is subsequently captured by SVNKit, which cancels the operation. In case a user requests the interruption of a command the class *CommandBlackboard* invokes the handler thereby setting an interruption flag to true. The flag is

subsequently checked by the *checkCancelled* method which initiates the command interruption.

### 7.3 Command Parser

The command parser package enables parsing the command strings passed to the graphical user interface. To this end the JavaCC parser generator comes into play, which generates a fully functional LL parser. To this end it is necessary to provide a JavaCC grammar for the desired commands and then to use the generated parser classes (see Figure 70) from the *Controller* class of the user interface package.

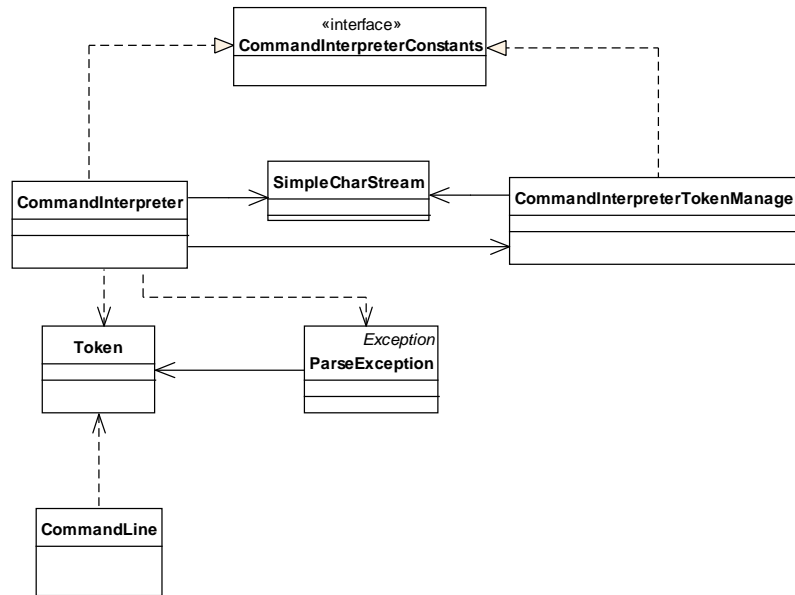


Figure 70: Command parser classes and dependencies

The class *CommandLine* holds the commands issued by the user along with all the command arguments. Hence, instances of this class are passed to the *Controller* class of the graphical user interface in order to initiate the command execution.

*CommandInterpreter* is delivering the instances of *CommandLine* by parsing the strings input by the user in the corresponding field of the user interface. To this end *CommandInterpreter* is given a *java.io.Reader* and in particular a *java.io.StringReader* of the command string input by the user. It then uses the *CommandInterpreterTokenManager* class to break down the input string to different tokens (class *Token*) and to build up instances of *CommandLine*. The class

*CommandInterpreterConstants* contains constants that are used during parsing. These constants include the strings of the allowed Customization Layer commands. The latter are passed to *ParseException*, which is thrown by the interpreter if the input command or arguments are invalid. The exception uses the constants to generate an error message that is then displayed in the output field of the user interface.

### 7.4 Section summary

This section has presented a framework facilitating the implementation of Customization Layer frontends as console applications based on Java technology. Next section will present a process for the adoption of a Customization Layer within an organization.

## 8 Adoption process

In order to ensure that a Customization Layer is transferred successfully into an organization, it is necessary to carefully examine the organizational context at hand and to adapt the approach when necessary. Hence this chapter describes a series of steps that can be followed by an organization for the adoption of a Customization layer. The Quality Improvement Paradigm (QIP) [BCR94] is used to this end. QIP is an iterative process improvement approach that describes a sequence of recommended steps for the introduction of a new process (e.g. method, technology, and tool) into an organization. Figure 71 gives an overview of the cyclic QIP.

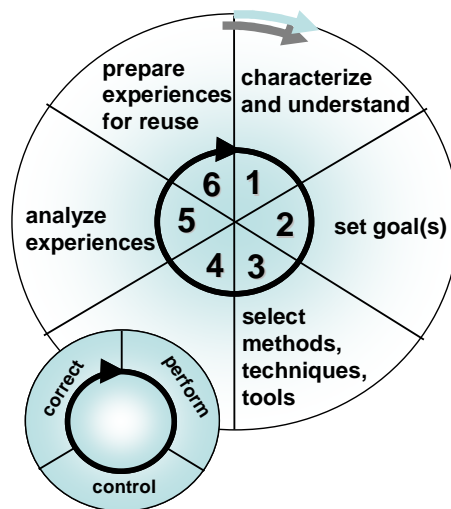


Figure 71: Steps of the Quality Improvement Paradigm (QIP)

The first step in the QIP aims at understanding the organizational context as well as the processes and setting the baseline to compare to afterwards. Subsequently issues are identified and improvement goals are set. In the next step solutions including methods, techniques or tools are selected that can contribute to the goal fulfillment. Then, in the fourth step, the selected solutions are applied. As shown Figure 71 the latter step can be seen as a subordinate cyclic process. This sub-process is broken down to a step that actually applies the selected solutions, a step that controls the extent, to which goals are addressed and finally a step that takes corrective measures.

After having applied a set of solutions QIP continues with an analysis of the experiences. That includes the identification of problems that arose during the application, the analysis of measurements that possibly took place and finally the derivation of recommendations for the next iteration. In the final iteration step QIP explicitly documents and packages the experiences so they can be easily retrieved for reuse in future iterations. The next subsections will elucidate the steps recommended by the QIP.

## **8.1 Characterization**

The Customization Layer approach supports organizations that have a product line in place and want to improve the way evolution is controlled. The first step in this direction is to clearly understand the type of product line that is to be controlled. This will normally be performed only once when the QIP cycle starts. As described in section 3.6 there are various types of product lines. In order to identify which type is relevant for an organization Figure 72 provides a decision tree. By using this tree the organization performs a first reasoning and raises the awareness regarding the product line situation at hand. This is a useful starting point for the later QIP steps.

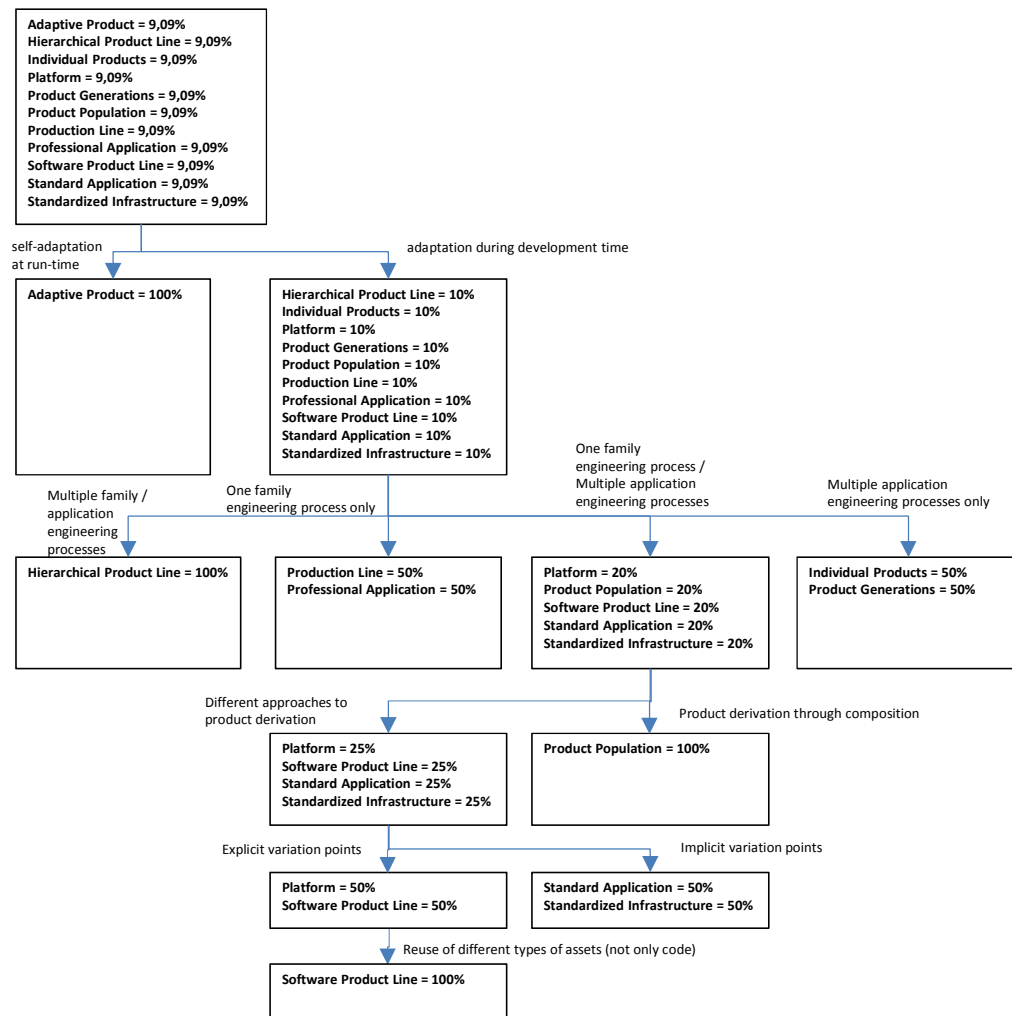


Figure 72: Product Line Type Decision Tree

Each decision in the tree contains the probabilities of having a particular type of product line given a set of criteria. The criteria that have to be considered are:

- Number of family and application engineering processes: This is the most crucial factor for the later activities. At this point the organization has to start thinking about the different product line engineering processes that are possibly established as well as about the interactions between them.

Binding time and adaptation mode: Binding time refers to the point in time in the development process when variation points in core asset are resolved [Kru03]. The adaptation mode on the other hand



refers to the way variation points are resolved. There are two main modes of adaptation: Product line members resolve variation points automatically (i.e. self-adaptation is performed based on contextual information that is acquired automatically) or the variation points are resolved by application engineers.

- Product derivation mechanisms: There are various mechanisms for the derivation of instances from core assets. The most prominent examples of such mechanisms are applied on source code assets. For example a core asset can employ conditional compilation (i.e. if-defs) to implement variation points. In this case instances are derived by declaring identifiers in macros.
- Types of reusable assets: Core assets can be produced at various stages in the development process. Hence the types of reusable assets refer to the stages in the development process in which reuse takes place.

At the top level of the decision tree all possible types have the same probability. The links between the decisions in the tree represent questions that have to be answered. When a particular question is answered positively the associated decision is selected. In that way the decision tree gradually reduces the number of possible types. For example if product line members are subject to self-adaptation during execution, decision *d3* is selected and evolution control has to deal with adaptive products.

After identifying the product line type the organization has to characterize the way evolution is controlled. To this end the conceptual model presented in section 3.8 can be used. In doing so the organization describes family and application engineering processes at hand. Subsequently the corresponding scenarios can be selected as presented in section 5.

The next step is to reason about the performance of the evolution control processes at hand. This analysis can be facilitated by taking central goals of product line evolution control into account and by reasoning whether these goals are met. Generally evolution control aims at increasing the productivity in the development process by taking optimal advantage of software reuse, by reducing maintenance effort and by ensuring the sustainability of the product line. In this regard a series of finer goals can be defined. Table 20 provides a sample refinement of these goals.

| Top-level Goal     | Sub Goals                                                                                                                                                                                                                                                                                                                                                                                        |
|--------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Software reuse     | <ul style="list-style-type: none"> <li>• Core assets are reused across products</li> <li>• Core asset development is synchronized with product development</li> <li>• Requirements for new core assets can be easily identified</li> </ul>                                                                                                                                                       |
| Maintenance Effort | <ul style="list-style-type: none"> <li>• The majority of the development effort is spent for the creation of new features and not for quality assurance</li> <li>• Efficient change impact analysis</li> <li>• Only allowed changes are actually carried out</li> <li>• Changes can be traced back to product line requirements</li> <li>• Redundancies can be efficiently identified</li> </ul> |
| Sustainability     | <ul style="list-style-type: none"> <li>• The complexity of the configuration management repository increases at an acceptable rate</li> <li>• The degree of core asset reuse remains stable or increases over time</li> <li>• Maintenance effort increases at an acceptable rate</li> </ul>                                                                                                      |

Table 20: Main evolution control goals and sample refinement

Evolution control problems can be made even more tangible by looking into concrete measures. In the area of evolution control it can be beneficial to take existing configuration management measures into account and to map them to the previously defined goals. Following list provides example measures in this regard [Le04]:

- Average time taken for the resolution of change requests
- Number of change requests (in particular problem reports)
- Percentage of approved change requests
- Number of defects found after every release
- Number of unfixed bugs in each release
- Time difference between defect reporting and removal

Furthermore it can be beneficial to make use of software reuse metrics as proposed for example in [OH92], [Pou97], [WYF03] or in [Pa10]:

- Maintainability index: A measure of the maintainability of a reusable component in terms of its complexity

- Reuse efficiency: Percentage of reused software relative to the total amount of software
- Rate of Component Observability: A measure of the understandability of a reusable component in terms of its externally visible behavior.
- Cost-benefit analysis metric: A measure of the return on investment on software reuse. Such a measure will usually take into account effort for the development of reusable components, effort for their retrieval and modification as well as effort for development of new components (i.e. without reuse)
- Temporal code churn [HM00]: A measure of modifications on lines of code over a time period.
- Number of variation points: Amount of well-defined positions within reusable assets that can be adapted in a prescribed way

Such measures enable setting a clearly comparable baseline. That means that if such measurements are performed (i.e. calculated or estimated) in the characterization step they can be easily compared with corresponding measurements of subsequent characterizations (i.e. in subsequent QIP iterations). In so doing the value of the improvement effort can be clearly assessed.

## 8.2 Goal definition

The characterization step sets the baseline of QIP iterations. The next step analyzes the results of the characterization and sets improvement goals. In this regard the first analysis to be undertaken is an evaluation of the organization-specific specialization of the conceptual model and the scenario selection. At this point the organization has to reason whether the structure of the evolution control processes is satisfactory and whether the selected scenarios cover the needs of the involved stakeholders. The analysis should look into the granularity of the defined product line processes (i.e. family and application engineering) and reason whether process decomposition is sensible. An indicator towards such decomposition might be the presence of many hybrid processes in the model.

Through the analysis of the conceptual model instance the organization can identify possible weaknesses in the baseline process structure. The next step is then to correct the process structure accordingly. Therefore the new structure represents the goal for the active iteration.

The goal definition can be further refined by analyzing the measurement results from the characterization phase as well. If the results are not acceptable further goals can be defined in terms of measurement values that have to be reached until the next characterization and with the new process structure that has been possibly modeled.

### 8.3 Process selection

The third step in the QIP cycle selects the engineering processes in terms of models, methods, techniques and tools that are expected to address the goals set in the previous step. In the context of this thesis this involves the examination of the configuration management functionality at hand and the creation or modification of a Customization Layer according to the scenario specifications and guidelines discussed in chapters 5 and 6. Furthermore the implementation framework presented in section 7 can be used, possibly as an initial proof of concept. Figure 73 provides an overview of the QIP steps thus far.

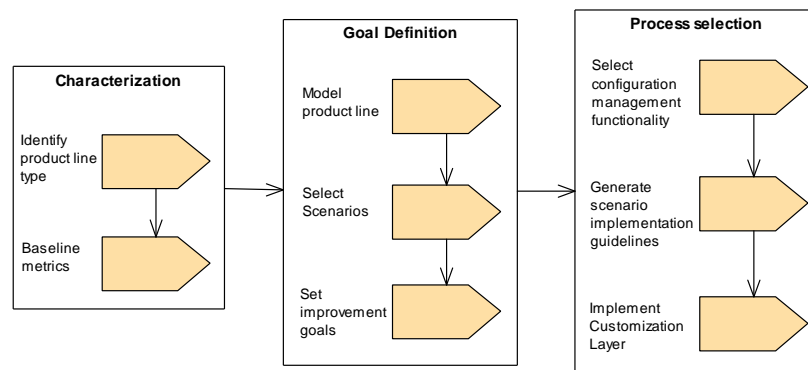


Figure 73: Adopting a Customization Layer (steps 1 to 3)

### 8.4 Execution

During the execution step the implemented Customization Layer comes into operation. The first step is to identify already existing assets items that can be put under the control of the Customization Layer. Subsequently evolution activities can start in terms of the selected evolution control scenarios.

The execution step is to be considered as an iterative process within the bigger QIP cycle. Therefore after having executed a set of scenarios an analysis step takes place. The goal is to get feedback from the use of the Customization Layer and possibly to provide usage instructions as corrective action. In particular the analysis step should look after possible

misuses of the Customization Layer. Since the Customization Layer commands automate a series of configuration management operations, which otherwise would require more effort, there is the risk that users issue more commands than necessary. This should be identified during the analysis and corrective action should be taken. For example particular commands can be blocked until more strict rules are implemented in the next QIP iteration. For the blocked commands the execution can continue with the conventional way; that is through direct usage of configuration management.

## **8.5 Analyze experiences**

This step retrospectively looks into the execution phase and aims at understanding strengths and weaknesses of the available Customization Layer implementation. The effectiveness of the corrective measures taken during the execution is assessed with respect to the success of the measures but also with respect to the information that was available.

The main issue during this step is to analyze whether the goals of the iteration (section 8.2) were met. Following list provides a set of possible reasons, for a failure of the Customization Layer in the iteration. Based on these possibilities the next iterations can be planned.

- Usability issues: Since a Customization Layer is built from scratch and on top of an existing configuration management system there may a usability gap that has led to misuses during the execution.
- Wrong input to the execution process: If the input to the execution process were core and product assets that do not need particular synchronization the Customization Layer might not show any benefit.
- Too much automation: Users do not have a clear picture of the effects of a Customization Layer. This can happen if the layer encapsulates a series of underlying configuration management operations without the users knowing about it.
- Wrong distribution of functionality: As discussed in section 6 evolution control functionality can be distributed across the existing configuration management system and a newly introduced Customization Layer. If this distribution is not adequate the acceptance of the solution loses ground.

## 8.6 Prepare experiences for reuse

In the final step the findings of the analysis phase are explicitly documented and provided as input to the next iteration. At this point it is important to capture all implementation decisions and the corresponding rationales that pertain to the Customization Layer of the iteration at hand.

In this context it can be beneficial to use a template such as the one suggested in [DP00] in order to capture the rationale behind software engineering decisions. As discussed in section 6 the implementation of a Customization Layer involves reasoning on multiple alternatives. This is due to the broad range of functionality provided by configuration management that can be used to realize the evolution control processes. Decisions and experiences that are made in this regard can be therefore particularly beneficial for future QIP iterations. Rational management enables describing the reasons behind the implementation by judging different alternatives (e.g. branches, properties, special-purpose files) that were evaluated upfront and by explaining the final decision. The judgment of the alternatives is accomplished with the help of different factors (impact, effect etc. in the example) that should relate to the respective QIP goals (QIP step 1 and 2).

## 8.7 Section Summary

This chapter has presented an adoption process based on the Quality Improvement Paradigm that guides the introduction of a Customization Layer into an organization. Next chapter presents validations carried out in the context of this thesis.



## 9 Validation

As described in section 1.3 the practical contribution of the work described in this thesis is to reduce the effort, which product line engineers have to spend, in order to coordinate activities on the basis of configuration management and in a product line engineering process. This contribution supports the avoidance of further higher-level problems that can be encountered by a product line organization. Figure 74 depicts the main problem addressed by this thesis along with higher-level problems that are related to it. The left-hand side of the figure shows the different problems, and the right-hand side shows hypotheses that can be used in order to validate possible solutions. Hypotheses H1 and H2 have been investigated in the context of this thesis.

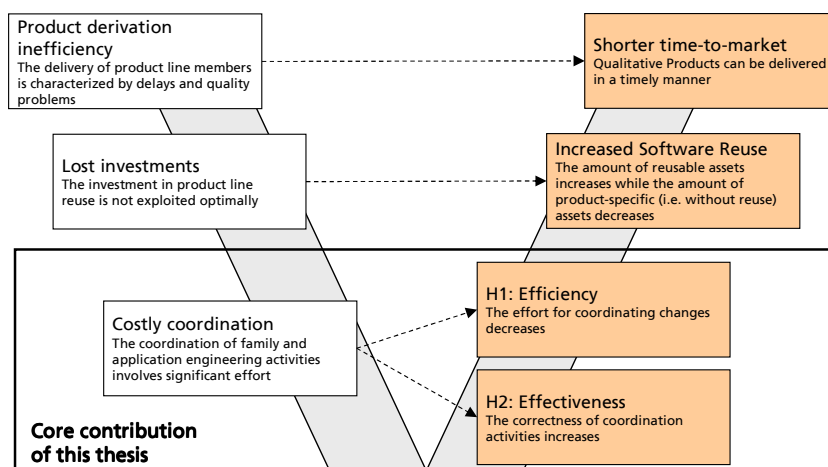


Figure 74: Experimental V-Model of this thesis

Figure 74 illustrates that the problem of costly coordination, which is addressed by this thesis, influences two higher-level problems:

- **Lost investment in software reuse:** When the coordination of family and application engineering is too costly, reusable assets cannot be evolved adequately. Changes on instances of reusable assets cannot be easily coordinated. Consequently reusable assets do not fulfill the needs of the product line and become obsolete over time. For a software organization this is a severe problem, since investments in software reuse may get lost.
- **Product derivation inefficiency:** When the applicability of software reuse gradually disappears, the effort for the delivery of products to customers increases. Instead of saving effort by applying previously



proven assets, application engineers create assets from scratch, possibly in a redundant way and spent a significant part (if not the majority) of their effort in quality assurance

The core contribution of this thesis, namely the reduction of coordination effort, has been validated by means of a usability evaluation, two controlled experiments, a simulation study and a case study with a software developing organization. The validation activities aimed at comparing the effort for evolution control with a Customization Layer against the corresponding effort with direct usage of configuration management.

Furthermore a structural evaluation that will be presented in the subsequent section, clarifies the decisions underlying the Customization Layer solution as well as the sensitivity points, trade-offs and risks that can be expected.

## 9.1 Structural evaluation

This section will clarify the benefits of the Customization Layer solution in terms of a structural evaluation, as partially proposed by the architecture tradeoff analysis method (ATAM see [KKC00]). The argumentation will produce following output.

- Design Decisions: Core decisions that were taken during the conception and development of the Customization Layer
- Rationale: The reasons and assumptions that led to the application of a particular decision
- Sensitivity Points: Parameters of a design decision that might influence a particular performance indicator of the solution
- Tradeoffs: Parameters of a design decision that affect differently more than one performance indicator. Thus, a sensitivity point can also be a tradeoff.
- Risks: Decisions, which have been left open or decisions whose consequences to the performance of the solution, are not specified.

### 9.1.1 Core Decision: Layering

The solution proposed in this thesis, the Customization Layer, is a layer on top of a conventional configuration management system (CMS). Therefore the core decision that underlies the solution is the deployment of a layer on top of configuration management. The layering approach is

strict, meaning that all Customization Layer operations are encapsulating lower-level configuration management operations.

The reason for the layering decision is the assumption that configuration management is common practice that cannot be neglected in the vast majority of systems and software developing organizations. On the other hand the bare usage of configuration management in a product line context entails complexity that can easily overwhelm users (see sections 1.3 and 1.4). Therefore the Customization Layer solution addresses the needs of product line evolution control while preserving existing investments of an organization.

The adoption of a Customization Layer can be judged in terms of the effort needed for the development of the layer and in terms of the benefits to be expected afterwards. The underlying configuration management system influences greatly these parameters. Following sensitivity points have to be considered:

- Availability of an Application Programming Interface (API) in the underlying CMS: An API can significantly simplify the implementation of the evolution control scenarios described in section 5. The lack of an API increases the complexity for the automated interaction between Customization Layer and the CMS. It might be for example necessary to invoke the command line interface and to parse the console output of the underlying system.
- Availability of particular CMS features: As discussed in section 6 the functionality provided by the chosen CMS has an important impact to the effort for the creation of a Customization Layer. In particular the availability of ticket hierarchy approaches, of marking functionality and finally of versioning functionality has to be considered. These capabilities are also tradeoffs as they influence both the effort for the establishment of a Customization Layer and the expected benefits. The latter refer in this case to the efficiency of the layer operations (i.e. response time, usage of computing resources), the resulting complexity of the CMS repository and the maintainability of the Customization Layer. Efficiency can decrease for example if the Customization Layer has to iterate over multiple versions in the history of an artifact in order to find instances created off this artifact. Repository complexity can be estimated in terms of the amount of configuration items, versions, branches, properties and custom commit messages. This complexity is increased by a Customization Layer that for example uses branches to create instances.
- Extensibility of the underlying CMS: Every CMS comes with a graphical or command line interface. For a Customization Layer to

be easily adopted by an organization it should be seamlessly integrated in the existing CMS user interface. To this end some CMS provide extensibility points. For example the free version management system Mercurial [URL21] enables implementation of extensions for its command line interface.

The present thesis aims at providing a solution that can be used with different CMS. Therefore the selection of a concrete CMS is left open. This can be clearly seen as a risk of the current approach, as there is plethora of CMS available in the market and the Customization Layer approach might not be applicable to all of them. However this risk has been mitigated by studying the CMS spectrum of functionality, by identifying varying CMS features and by providing corresponding implementation guidelines.

Another risk of the layering approach arises when switching the underlying CMS is a realistic scenario. If a Customization Layer is been established on top of a given CMS, changing the CMS bears the risk that the product line information is lost. This risk can be mitigated by choosing the offline basic asset update mode (section 4.3), which holds a backup of the product line information in parallel to the CMS repository.

Concluding, the Customization Layer opts for a layering approach on top of a CMS. The selection of a concrete CMS, which is also the central influencing factor of the solution, is left open. Yet, typical variations in CMS functionality and their impact to the Customization Layer are captured. Alternative approaches would either opt for a concrete CMS or create a Customization Layer independently of a CMS (i.e. the Customization Layer would come with its own repository of product line artifacts). These approaches however would limit the easiness of adoption of the Customization Layer approach and this would contradict one of the research questions in this thesis (Research question 4, section 1.3.2). Given this argumentation the layering decision is considered as viable in the context of this thesis.

### **9.1.2 Data Model**

The second decision in the present thesis is the structure of the data model discussed in section 4. This model captures the entities of product line evolution control (section 4.1), namely core assets (that contain variability), instances (that resolve variability) and product-specific assets (developed without reuse in a product). Core assets and instances play the central role because they have to be synchronized over time. A state model (section 4.2) is also defined to this end, which explicitly specifies the different synchronization states of core assets and instances.

A tradeoff that can be identified at this point is the decision to separate the management of assets from the management of configuration items (section 4.3). Assets are logical entities managed by the Customization Layer whereas configuration items are physical entities (i.e. change requests, files or directories) in the configuration management repository. Furthermore, logical entities can be related to their corresponding physical entities (see for example section 4.1.1). The choice of the update mode influences the performance of the Customization Layer:

- The dynamic update mode reduces the efficiency of operations, as the Customization Layer has to instantiate the model dynamically. However, in so doing the obtained model is up-to-date at the time of its creation.
- The offline mode increases efficiency in particular when multiple logical entities are to be processed. In this case the model contents are available and do not have to be created dynamically through interaction with the repository. On the other hand the implementation complexity increases since the Customization Layer must enable synchronization with the repository. Furthermore some information will be kept redundantly. In particular, associations between core assets and instances will be kept in the logical model as well as in the repository. Although this makes the usage of additional computing resources necessary it enables replacing the underlying configuration management system, if necessary, as the basic asset model is available offline.

The data model capture entities and relations pertaining to product line evolution control. A risk that can be seen at this point is that some of these model elements may not apply to all types of product lines. For example product populations (section 3.6.7) do not require controlling the evolution of asset instances. This risk has been mitigated by enabling the optionality of model elements by considering different types of product lines that can arise (section 3.6). Associations between core assets and instances are for example optional.

The data model discussed has been derived from common product line engineering definitions and is applicable to various types of product lines. Through the (optional) association between core assets and instances it explicitly addresses the issue of product line erosion (section 1.4). Given also the fact that the model does not bear any uncontrolled risks, it can be seen as a solid foundation for the Customization Layer solution.

### 9.1.3 Process Model

Another decision taken in the present thesis is the selection of the evolution control operations (section 5) to be offered by a Customization Layer. These operations enable engineers to plan, perform and monitor changes in a product line context. The focus thereby resides on the relation between core assets (that contain variability) and their instances (that resolve the variability).

The evolution control operations have been specified in term of a domain-specific language. The goal is to enable users to select operations to name them at will and thus to tailor a Customization Layer according to their needs. This is a tradeoff to the implementation of a Customization Layer. The more operations are selected the more complex is the implementation. On the other hand a rich set of operations increases the usefulness of the layer.

A risk that can be identified arises from the question whether the set of proposed evolution control operations is complete. There are indeed operations (for example moving configuration items associated to core assets) that are not explicit part of the current set. The currently selected operations are based on common control theory scenarios and capture the main evolution primitives, namely creation, modification and removal of assets. This is the strategy used to mitigate the completeness risk. The assumption is that additional operations will be achievable through combination of the existing ones (moving for example is realizable through modification of core assets).

Another risk at this point arises from the fact that a Customization Layer identifies changes but does not explicitly enable to reason about these changes. In other words the Customization Layer will indicate when changes have to be propagated from core assets to instances and vice versa, but it will not support the actual merging activity. The latter has to compare in detail the affected assets, to identify the differences of interest and to finally change the assets accordingly. There is currently no approach that explicitly supports differencing core assets and instances. Next to the traditional approaches that compare files textually, there are some approaches for semantic [Me02] or structural merging [ALB+11]. But also these approaches do not consider the fact that instances are obtained from core assets through resolution of variability. In order to mitigate the risk of lacking tool support at this point this thesis introduced a formal model (section 5.4.2) that can serve as a foundation of a “variability-aware” differencing mechanism.

## 9.2 Usability evaluation

Goal of the usability evaluation was to estimate the effort of using the Customization Layer solution as opposed to the effort for the direct usage of Version Management. As discussed in the introduction of this chapter, effort is considered as the combination of efficiency with effectiveness.

The first step was to decide upon the usability evaluation method to apply. Out of the various methods available in the literature, the study was restricted to heuristic evaluation, cognitive walkthrough and usability test as proposed in [BEM+12]. The final decision was to select the cognitive walkthrough method [WRL+93] for two reasons:

- **Conceptual:** Cognitive walkthroughs are task-based. Tasks are defined in advance and the usability inspectors judge the system by estimating the effort for the execution of the tasks. In so doing, the inspectors put themselves in the position of employing the running system. The estimation is then done by answering pre-defined questions on each task. The cognitive walkthrough method was seen as most appropriate since different evolution control scenarios could be directly mapped to tasks. Due to the task-based character cognitive walkthroughs are more concrete than heuristic evaluation. The latter judge a system based on common heuristics. Thus they require more experience and more effort from the evaluators. In the context of this thesis the participants were students and software engineering professionals with no experience in usability evaluations.
- **Organizational:** Cognitive walkthroughs do not require a running software application to be inspected. The evaluation is based on interface documentation of the system. This simplified the organizational setup. A usability test on the other hand would require a running Customization Layer to be installed. Given the number of participants (17) this posed organizational difficulties. Apart from that, a direct evaluation of the running software took part in the experimental studies presented later in this chapter.

### 9.2.1 Planning

The usability evaluation involved 17 participants. 6 of them were software engineering professionals with good experiences in the field of product line engineering (PLE) and average to strong experience in version management (VM). The remaining 11 participants were students in the product line engineering class at the University of Kaiserslautern. For the students the evaluation was associated to the version management exercise, which every year is part of the class.

As expected no student had experience with product lines, while some of the students had a good version management background. Following table shows the experience profiles of the participants. The profiles were used to achieve an even distribution of tasks to the participants.

| Participant    | Version Management Experience                                                        | Product Line Engineering Experience                                                    |
|----------------|--------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------|
| Participant 1  | low                                                                                  | low                                                                                    |
| Participant 2  | strong                                                                               | low                                                                                    |
| Participant 3  | low                                                                                  | low                                                                                    |
| Participant 4  | low                                                                                  | low                                                                                    |
| Participant 5  | strong                                                                               | strong                                                                                 |
| Participant 6  | strong                                                                               | low                                                                                    |
| Participant 7  | strong                                                                               | low                                                                                    |
| Participant 8  | average                                                                              | strong                                                                                 |
| Participant 9  | strong                                                                               | low                                                                                    |
| Participant 10 | average                                                                              | strong                                                                                 |
| Participant 11 | low                                                                                  | low                                                                                    |
| Participant 12 | low                                                                                  | low                                                                                    |
| Participant 13 | low                                                                                  | low                                                                                    |
| Participant 14 | strong                                                                               | average                                                                                |
| Participant 15 | low                                                                                  | average                                                                                |
| Participant 16 | low                                                                                  | strong                                                                                 |
| Participant 17 | strong                                                                               | low                                                                                    |
|                | <b>VM experience summary:</b><br>Total low: 8<br>Total average: 2<br>Total strong: 7 | <b>PLE experience summary:</b><br>Total low: 10<br>Total average: 2<br>Total strong: 4 |

Table 21: Usability Evaluation / experience profiles

The experience profiles were collected before the evaluation in terms of a pre-briefing. The latter was also used to prepare the participants to the context and goals of the evaluation and also to introduce the participants to the cognitive walkthrough method. To this end corresponding information sheets (see appendix) have been distributed one week before the actual evaluation took place.

The usability evaluation workshop consisted of two sessions:

- Introduction and refreshing session (~10 minutes): Discussion of the evaluation goals and the overall context of product line evolution control
- Main session (~90 minutes): Task processing and documentation of results in corresponding usability evaluation forms (see appendix).

### 9.2.2 Tasks

Goal of the cognitive walkthrough was to investigate the Customization Layer solution as opposed to regular version management with respect to specific tasks. These tasks are listed in the following:

1. Find items marked as Core Assets
2. For a given Core Asset (e.g. MyLibrary.java) find where it is being reused in products. In other words, find items marked as instances of the Core Asset.
3. For a given Instance (e.g. MyLibrarySpecialized.java) find from which Core Asset it comes from.
4. Imagine a core asset has been changed. Propagate the changes to its instances
5. Imagine an instance has been changed. Propagate the changes to the core assets, from which the instance comes from
6. Find assets of a product line member, which are not instances of core assets (in other words, find product-specific assets)
7. Find items marked as Instances

The above tasks were distributed across three groups as shown in Table 22.



- Customization Layer (CL): This group investigated the usability of the Customization Layer
- Version Management / Family Engineering (SVN\_FE): This group investigated the usability of conventional version management regarding family engineering
- Version Management / Application Engineering (SVN\_AE) : This group investigated the usability of conventional version management regarding application engineering

|               | Participant         |
|---------------|---------------------|
| <b>CL</b>     | 1, 3, 5, 10, 12, 16 |
| <b>SVN_FE</b> | 4, 6, 7, 11, 14, 15 |
| <b>SVN_AE</b> | 2, 8, 9, 13, 17     |

Table 22: Usability Evaluation / Group assignments

The CL group participants processed tasks 1 to 6. The two other groups processed tasks 1, 2, 4 and 7, 3, 5 respectively. It would have been however possible to create only one version management group that also processes the full amount of tasks. The reason for not doing so was that the version management participants had to deal with a more complex interface and had to combine several operations in order to fulfill a task. Therefore, in order to allow all participants to finish in the same time frame it has been decided to split version management in two groups and to reduce the number of tasks accordingly. The version management groups were then assigned the most typical task sequences for the respective product line engineering process. This sequence includes finding entities (core assets or instances), finding related entities (instances or core assets) and finally propagating changes.

### 9.2.3 Interface descriptions

The cognitive walkthrough method does not require a running software system to be investigated. Thus interface descriptions (see appendix) in terms of a simplified application programming interface have been distributed to the participants at the beginning of the evaluation. For the CL group the description reflected the XText specifications described in section 5. For the version management groups the descriptions were based on the SVNKit programming interface [URL5].

#### 9.2.4 Analysis

According to the cognitive walkthrough method participants were asked to decompose each task to necessary actions and then to judge the resulting complexity. For example a participant of the SVN\_FE group decomposed task 2 (i.e. finding instances of core assets) to two actions. These were to first call the directory listing operation (dir) and then to call the log operation.

Hence, in the context of this usability evaluation efficiency is a function of the necessary actions per task, the effort of identifying these actions and finally the effort of the actions themselves. On the other hand effectiveness is seen as the correctness of decomposition of tasks to actions. That means, a task is defined as correct if the selected actions indeed lead to the fulfillment of the task. The assumption is that a system with low usability might give the user the impression that a task was fulfilled through a series of actions, while this is actually not the case.

In order to judge the effort per action the cognitive walkthrough method proposes to answer following questions:

- Was it easy to understand, that you had to do this action?
- Was it easy to associate the correct action with the effect you are trying to achieve?
- Will you see that progress is being made toward solution of your task?

In order to judge the correctness it is necessary to analyze task decompositions and to examine whether the corresponding tasks are indeed fulfilled.

In total there were 100 actions identified by the participants and captured in usability evaluation forms. After the workshop the majority of the results have been digitalized. To this end, a web-based data entry application has been developed (see appendix B.6). The latter enabled to store all results in a relational data base management system, which in turn allowed querying the result set in various ways.

##### ***Efficiency evaluation***

In average the Customization Layer participants required 1.3 actions per task. This was mostly expected, since the Customization Layer aims at encapsulating underlying version management operations and therefore at reducing the number of actions. On the other hand, the version

management participants required 1,588 actions per task in average. This difference of only 18% was surprising. It was expected that the version management groups would need significantly more actions per task. Figure 75 shows in detail how many actions were required per task.

The analysis first looked in the Customization Layer results, in order to understand why some participants required more than one action for some tasks. It was actually expected that the operations offered by the CL interface could be mapped one-to-one to tasks. The analysis showed that 4 (out of 6) CL participants required more than one action for 4 (out of 6) tasks. By looking in detail into the corresponding usability forms it was identified that in most cases the additional actions were not necessary (see Table 23). The participants selected the additional actions possibly because they misunderstood the task specification. Another possible explanation is that in some cases additional actions were selected in order to confirm the results of a preceding action.

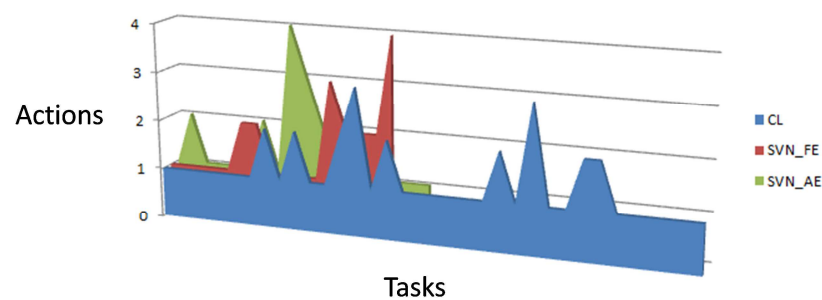


Figure 75: Usability Evaluation / Actions per Task

In one case the participant included actions, which were not defined in the CL interface description. The participant possibly assumed that the rebasing operation of the Customization Layer would not first update the working copy of the instance. Furthermore he possibly assumed that after the rebasing operation the Customization Layer would not commit the changes on the core assets. Therefore he identified updating and committing as necessary actions. It is indeed the case that the CL interface description did not detail what happens during the rebase operation in terms of updating and committing. Therefore the participant's assumption has to be considered as reasonable. On the other hand, the rebase operation's purpose was in fact to encapsulate these actions as well.

| Task                                                                   | Actions identified by participants                                                                                                                                                                                                                            | Analysis                                                                                                                                                                               |
|------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Find instances of a given core asset                                   | <ul style="list-style-type: none"> <li>• <code>showCoreAssets</code></li> <li>• <code>showCoreAssetInstances</code></li> </ul>                                                                                                                                | <code>showCoreAssets</code> was not necessary: <code>showCoreAssetInstances</code> expects the name of the core asset, whose instances are to be found                                 |
| For a given instance find the core assets it originates from           | <ul style="list-style-type: none"> <li>• <code>showCoreAssets</code></li> <li>• <code>showCoreAssetInstances</code></li> </ul>                                                                                                                                | <code>showCoreAssets</code> was not necessary: <code>showCoreAssetInstances</code> returns <code>Instance</code> objects with all information necessary                                |
| Propagate changes of a core asset to its instances                     | <ul style="list-style-type: none"> <li>• <code>showCoreAssets</code></li> <li>• <code>integrateCoreAsset</code></li> </ul>                                                                                                                                    | <code>showCoreAssets</code> was not necessary: <code>integrateCoreAsset</code> fulfills the task                                                                                       |
| Propagate changes of an instance to the core assets it originates from | <ul style="list-style-type: none"> <li>• <code>showCoreAssetInstances</code></li> <li>• <code>update instance</code></li> <li>• <code>rebaseInstance</code></li> <li>• <code>showProductAssets</code></li> <li>• <code>commit merge results</code></li> </ul> | <code>showCoreAssetInstances</code> and <code>showProductAssets</code> were not necessary: <code>rebaseInstance</code> fulfills the task                                               |
|                                                                        |                                                                                                                                                                                                                                                               | <code>update</code> and <code>commit</code> were not part of the CL interface description; <code>rebaseInstance</code> did not detail what happens in terms of updating and committing |

Table 23: Usability Evaluation / Unnecessary actions in CL group

The next step in the analysis was to look into the version management groups. It was discovered that the version management participants did not process a series of tasks correctly. In these error cases, the identified actions would not achieve the corresponding task goal. Correctness will be discussed in detail in the next subsection. For the purpose of the efficiency evaluation it was reasonable to analyze the results without error cases as well. Hence, by factoring out the error cases the average amount of necessary actions drops without significance to 1,571. The difference to the Customization Layer is further reduced to 17%. There were no unnecessary actions in the set of correctly processed tasks.

Concluding, the minor improvement of 17% to 18% of the Customization Layer as opposed to version management can be explained only by the fact, that the Customization Layer participants selected a series of unnecessary tasks. Assuming that these participants would need - as intended - 1 action per task the improvement would be 36%.

Apart from the amount of actions per task the efficiency was also evaluated in terms of the answers given to the cognitive walkthrough questions (mentioned in the beginning of section 9.2.4). Figure 76 summarizes the results. The Customization Layer received 89% positive answers compared to 68% of the other groups. This is an indicator for a better usability of the Customization Layer: The participants had fewer problems in understanding the user interface, could better relate actions to effects and perceived more clearly progress towards achievement of their goals.

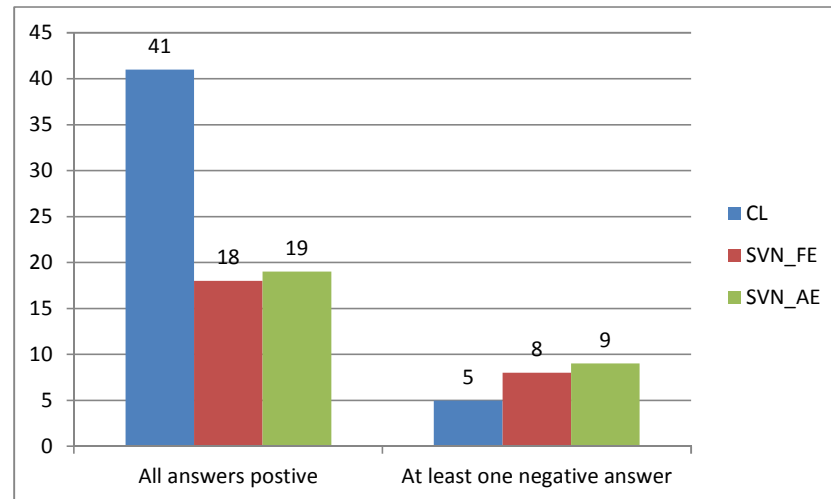


Figure 76: Usability Evaluation / Cognitive Walkthrough Questions

### **Effectiveness evaluation**

For the purpose of this evaluation effectiveness was defined as the number of tasks that were correctly decomposed to actions. If for a task the selected actions would not achieve the task goal the task was considered as wrong. The examination of the tasks was carried out after the usability workshop by the author of this thesis.

Figure 77 summarizes the effectiveness of the Customization Layer as opposed to conventional version management. As shown in the figure the Customization Layer group achieved significantly more correct tasks. The correctness ratio for the CL group was 94% as opposed to 70% of the other groups.

For the Customization Layer the expected correctness was 100% as the layer fully automates underlying operations. Nevertheless 2 tasks were not processed correctly. Again, this is possibly related to confusion due to the task specifications or due to the interface description.

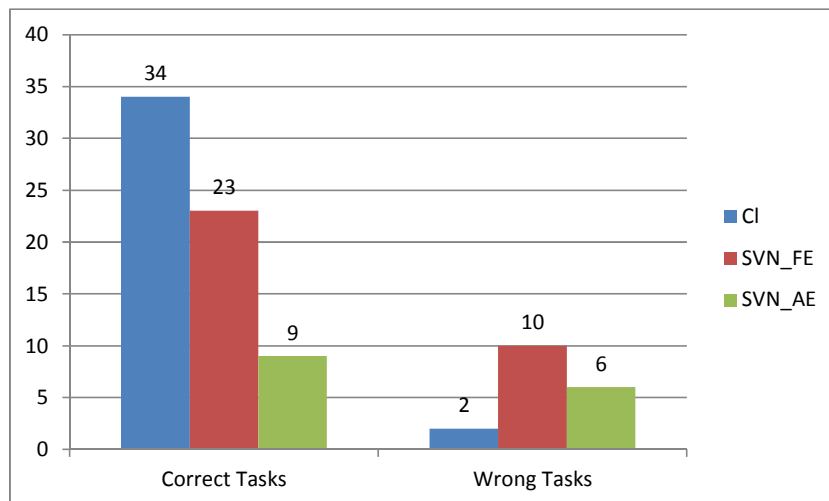


Figure 77: Usability Evaluation: Efficiency results

For version management wrong tasks were mainly due to the complexity of the user interface. Participants had to combine a series of operations, sometimes even recursively, in order to achieve results. The version management experience did not play any role in this context. Even experienced participants gave wrong answers.

### ***Improvement suggestions***

The cognitive walkthrough method also enables capturing improvement suggestions in cases where usability is perceived as low. Hence, during the main session participants were asked to provide suggestions every time they gave a negative answer to the three questions mentioned in the beginning of section 9.2.4.

In general, participants did not provide many suggestions. The most interesting suggestion came from the version management groups. Participants proposed 9 times to eliminate the requirement of performing the corresponding actions. This was interesting for the analysis provided that the Customization Layer aims exactly at eliminating such version management actions. There was also one Customization Layer member who proposed to eliminate the operations for showing core assets and instances. Given the fact that these operations map one-to-one to tasks the rationale behind this suggestion was not clearly understood.

## 9.3 Experimental validation

This section presents experimental validation activities including two controlled experiments and a simulation study. Before presenting the experiment results this section describes the Customization Layer that has been implemented beforehand and used in the experiments.

### 9.3.1 Customization Layer

Figure 78 specifies the Customization Layer that has been used for the experimental validation. This layer has been implemented upfront by the author of this thesis. Given the good knowledge of the underlying CMS and the availability of a well-documented programmatic interface this implementation has required approximately 4 person days. The reason was the relatively high complexity for the automated branch management, which will be explained in the following.

The specification is based on the models of Figure 27, Figure 28 and Figure 29. It contains the product line engineering processes to be controlled as well as the desired evolution control scenarios. Finally it specifies that branching functionality is available with the CMS at hand. As shown in the figure the product line under development follows the collection example discussed in section 1.3.5 and uses Subversion [URL6] as the underlying CMS.

The selected scenarios constitute a subset of the evolution control scenarios of section 5. The selection has been undertaken based on the following criteria:

- Selection of common scenarios: Creation of core assets and instances is a scenario that can be considered common in a product line context
- Selection of complex scenarios: In order to exemplify the benefits of a Customization Layer scenarios have been selected that require significant effort with conventional configuration management. These scenarios are `getInstanceDiff` and `getCoreDiff` and involve change propagation between family and application engineering.

```

ProductLine CollectionProductLine{
 FamilyEngineering LibraryDevelopment{
 VersionManagementFE LibraryDevelopmentVM{
 createCoreAsset addCoreAsset,
 integrateCoreAsset integrate
 },
 StatusAccountingFE LibraryDevelopmentSA{
 showCoreAssetInstances getInstances,
 showProductAssetChangesSinceLastSynchronization
 getInstanceDiff
 }
 }
 ApplicationEngineering LibraryApplication{
 VersionManagementAE LibraryAppVM{
 createProductAsset instantiateCoreAsset
 rebaseProductAsset rebase
 }
 StatusAccountingAE LibraryAppSA{
 showCoreAssets getCoreAssets,
 showCoreAssetChangesSinceLastSynchronization
 getCoreDiff
 }
 }
}

```

Figure 78: Customization Layer of the experiment

### 9.3.2 Configuration Management

For the experimental validation the well-known CMS Subversion has been used and a corresponding connector has been realized. The latter provided following method implementations:

- *addDir*: The method creates a new directory as a configuration item in the configuration management system. It is used by the Customization Layer when an asset (core or product asset) is created out of directory in the file system of the Customization Layer client. The method arguments are a string with the target repository location and a string with a commit message to be passed to the configuration management system.
- *commitItem*: The method commits changes to a configuration item. It is used by the corresponding method of the Customization Layer. The method arguments is a file or directory (instance of the Java class *File*) containing the latest changes, a Boolean value that when true locks the item after commit and a string with the commit message.
- *copyItemWithTag*: This method creates a copy of a configuration item and marks the copy with a specific mark. In other words the method creates a named branch of a configuration item. This operation is used when creating instances of core assets. Therefore



the instances are stored in branches of core assets. The method arguments are a string identifying the source configuration item, a number identifying the version number to branch off, a string with the copy destination, a commit message, a name for the branch and a Boolean denoting whether the copy operation is to be considered as a move. In some configuration management systems copy operation differ from move operations with respect the versioning history. Copy operations maintain the history from branched items back to the main configuration items. On the other hand move operations create a new versioning history that starts from the first branch version.

- *getBranchedItemsWithTag*: This method is used to retrieve configuration items that reside in branches that are marked with a particular tag. The method is used when retrieving core assets and instances. The method expects a repository location to start the search from, optionally a version number in order to obtain a branched item off a particular version number and finally a branch tag for the name of the branch.
- *getLatestItemRevisionSinceTag*: The method obtains the first version number in a configuration item branch. The information is used to check the synchronization status of core assets and instances: When instances are created through a branching operation the first version in this branch also describes the branch origin. In our case this is the core asset version the branch has been created from. This version can be compared with the latest core asset version to identify if the instance needs to be rebased. The method expects a repository location to base the query on and a tag in order to perform the query of specifically named branches.
- *getLatestRepositoryRevision*: This method delivers the latest repository revision, when global repository versioning is applied. This revision is used by other operations in order to define the version to start from in queries.
- *importItem*: This method imports a file or directory from the local file system to the configuration management repository. Therefore it accepts an instance of the Java class *File*, a string with the target repository location, a tag representing the commit message and a Boolean parameter that specifies
- *itemHasChanged*: The method counts the versions that succeeded a specific version of a configuration item. To this end it expects a repository location identifying the configuration item of interest as well as a version number to base the search on.

- *prepareRepoPath*: The method formats a string that represents a repository location according to the format used by the configuration management system at hand. This method simplifies the input of evolution control commands as simple strings can be used (e.g. relative paths for repository locations) that are automatically adapted to fully-qualified paths.
- *traverseHistoryUntilFirstTag*: The method traverses the version history of a given configuration item until it finds a specific commit message. If the corresponding version has been branched off another version the method returns the repository location of the branching source. Given an instance this method enables finding the core asset the instance originates from. The method expects a string with the name of the item to traverse and a string containing the commit message of interest.

The connector also provides a series of helper methods such as *setup*, *setRepository* and *dispose*. These methods are responsible for the initialization of the layer upon system start or to gracefully close network connections and to release resources upon system shut down.

The class *SVNConnector* is the implementation of the *CMAbstractionLayer* interface for Subversion. Figure 79 depicts the structure of the subversion connector (no public methods of *SVNConnector* are shown in the picture since they are inherited from *CMAbstractionLayer*). The class *CommitEventHandler* implements a series of handler interfaces, which are defined in *SVNKit* (i.e. the API for Subversion [URL5]). These handlers enable the *SVNConnector* to monitor various versioning operations and to interfere if necessary. For example the method *handleEvent* is invoked when a commit takes place while the *handleLogEntry* method is invoked when the version history is queried. Handlers can be useful for the interruption of operations as discussed in section 7.2.

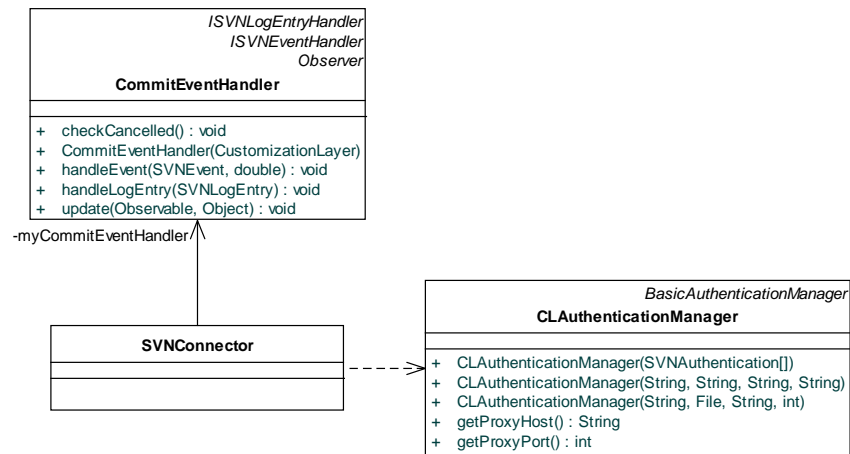


Figure 79: SVNConnector implementation

The class *CLAuthenticationManager* is a helper class supporting the authentication of the Customization Layer user with the subversion repository. The class is being instantiated during start-up with the user credentials and network connection properties (i.e. http proxy) at hand. It is then passed to *SVNKit* and provides the stored credentials upon connection with the repository.

The current implementation of *SVNConnector* relies fully on commit messages and the branching support of Subversion to keep track of core assets and instances. When a core asset is instantiated a branch off the core asset is created and a special marker is used in order to distinguish such a branch from other branches that may be created. Accordingly when a core asset is put under version control a special commit message is used to mark the commit operation accordingly. Hence when the user issues the *getCoreAssets* command *SVNConnector* queries Subversion for all branches or commits that are labeled with the specific marker. The current prototypical implementation obtains all log entries from the version history and then searches for the log entries that indicate the creation of instance branches. Such an operation may require significant time to return if called at top level (i.e. for all versioned items) in a long-lived repository (i.e. containing many versions). For example in a Subversion repository with 34.000 versions the *getCoreAssets* command requires 208 seconds when called at the root level. The same operation requires 23 seconds when invoked on a part (i.e. subdirectory) of the repository.

An alternative implementation at this point could combine the subversion functionality with a set of special-purpose files that store indexing information. For example a file can be used that saves the

repository version when a core asset or an instance is created. This would accelerate the execution of commands like *getCoreAssets* significantly. Since the current prototype was applied in experimental repositories these performance issues were not relevant and therefore the indexing functionality was not necessary.

### 9.3.3 Experiment 1

The goal of the first experimental study was to show in a controlled environment that the Customization Layer approach is significantly better than the state of the practice, which is the direct usage of configuration management for evolution control of a product line. The experiment was defined as follows [WRH+00]:

- Object of the study: The investigated objects are (a) the Customization Layer approach proposed in this work and (b) the direct usage of configuration management for product line evolution
- Purpose: The purpose was to evaluate each approach, in particular with respect to different user profiles
- Quality Focus: The quality focus is the efficiency (i.e. required effort) and the effectiveness (i.e. correctness) of the approaches
- Perspective: The perspective is from the researcher's point of view
- Context: The experiment was run with the help of students who were randomly asked to perform a set of tasks on a given lab setting. Each task contained a question the students had to answer by performing evolution control operations.

#### ***Experiment planning***

The experiment took place as a practical exercise complementing a lecture on software product lines. 14 undergraduate students participated in the experiment. There are two main hypotheses underlying the experiment

H1: The proposed method significantly reduces the effort in terms of the time necessary to perform evolution control operations

H2: The proposed method significantly increases the effectiveness of evolution control in a product line

These hypotheses can be refined as follows:

H1.1: The proposed method significantly reduces the time needed to perform evolution control operations during Family Engineering

H1.2: The proposed method significantly reduces the time needed to perform evolution control operations during Application Engineering

H2.1: The complexity of the underlying configuration management repository increases at a lower rate with the proposed method than with the state of the practice approach. Complexity can be estimated in terms of files, folder, versions and branches being created.

H2.2: The proposed method significantly reduces the errors made while performing evolution control operations

The following table describes the experiment variables.

| Variable            | Type        |
|---------------------|-------------|
| The approach in use | Independent |
| Students experience | Independent |
| Necessary time      | Dependent   |
| Complexity increase | Dependent   |
| Correctness         | Dependent   |

Table 24:

Experiment variables

### ***Experiment design***

The experiment consisted of two groups, the group CL that used the Customization Layer and the group SVN that directly used the Subversion version management system with the help of the TortoiseSVN client [URL17], which is available as a Microsoft Windows explorer extension.

Moreover there were two roles, family (FE) and Application Engineers (AE). The following table shows the arrangement of students in terms of student ids.

|            | <b>Family Engineer (FE)</b> | <b>Application Engineering (AE)</b> |
|------------|-----------------------------|-------------------------------------|
| <b>CL</b>  | 2, 7, 9, 10                 | 1, 6, 8, 13                         |
| <b>SVN</b> | 3, 5, 11                    | 4, 12, 14                           |

Table 25: Arrangement of students

Each student was asked to perform 4 evolution control tasks. The goal was to analyze the effect to the task performance of manipulating the “approach in use” variable. The tasks are summarized in the following table.

|               | <b>Family Engineer (FE)</b>   | <b>Application Engineering (AE)</b>                                          |
|---------------|-------------------------------|------------------------------------------------------------------------------|
| <b>Task 1</b> | Create core assets            | Create instances                                                             |
| <b>Task 2</b> | Change core assets            | Change instances                                                             |
| <b>Task 3</b> | Find instances of core assets | Find the origins of instances (i.e. core assets they have been derived from) |
| <b>Task 4</b> | Find changes in the instances | Find changes in the origins                                                  |

Table 26: Experiment tasks

### **Validity evaluation**

The complexity increase variable has a confounding factor, the existing repository complexity. In other words the rate, at which complexity increases, is expected to depend on the already existing complexity. In order to analyze this dependency a further experimental study is necessary. In the current study this potential internal validity threat has been addressed by using the same repository for all students and tasks. Hence both groups had to face the same repository complexity during the experiment.

Another internal validity threat was due to the different student profiles that participated in the experiment. There were students with different product line and configuration management experiences. This threat was addressed by randomization. That resulted to a reasonable distribution of student profiles across control and experimental group.

The most important (conclusion validity) threat though was the statistical significance of the observed effects. The small number of students and the restricted duration gave not enough data points to achieve general

significance (a t-test was not applicable and the Wilcoxon test did not show a significance). To address that the experiment must be ran again with more subjects (e.g. students or engineers) and higher duration.

There was also a social threat in the experiment arising from the split in two groups. The SVN group – as control group – had to work according to the state of the practice and did not have the chance to try out the Customization Layer. To reduce this effect the students have not been informed in advance about the different group settings. The latter have been presented only after the experiment finished.

Finally the experiment has an external validity threat arising from the maturity of the Customization Layer prototype. The current implementation is in a prototypical phase and entails some user-friendliness issues.

### **Operation**

The experiment ran in the context of a software product lines class. Therefore the first step was to refresh the concepts and challenges of product line infrastructures. Afterwards the tasks and the roles have been presented and the experiment started.

The repository had been already populated with core assets right from the beginning. So the AE role did not have to wait for the FE role to produce any core assets. Moreover, in order to simplify Task 1 for FE a core asset was already available on each student's machine. So the Family Engineers did not have to create any core assets from scratch; they only had to put the core assets, which were located on their machine, under configuration management control.

The data was primarily collected through feedback forms that were filled out after experiment execution. The students were asked to count the time needed for the execution of the tasks and then to fill-in that information in the feedback forms. In addition the forms allowed collecting data about the student profile (i.e. existing PLE and configuration management experiences) as well as general feedback data (the complete feedback forms are available in the appendix)

Furthermore, the students were asked to create text documents, in which they could enter the detail answers to their tasks (e.g. list of instances). Finally, further data has been captured by means of log files that are created automatically by the Customization Layer prototype but also by the underlying configuration management system. The detail answers that were put in the text files were used during the experiment interpretation in order to evaluate their correctness. To this end the

answers given by the students have been compared to the data provided by the log files.

The duration of the experiment was one hour. Most students managed to perform the tasks within that period. Since some students finished earlier they used the remaining time for further experimentation with the respective tool. The data produced from this experimentation has been factored out though.

### Analysis

Figure 80 shows a box-and-whisker diagram with the dispersion of the time that each student took in average in order to perform a task. As it can be seen the CL students needed less time in average: CL students needed 7.1 minutes and SVN students 9.9 minutes. In other words CL brought an efficiency improvement of 28% in average. This was an indicator that hypothesis H1 is supported.

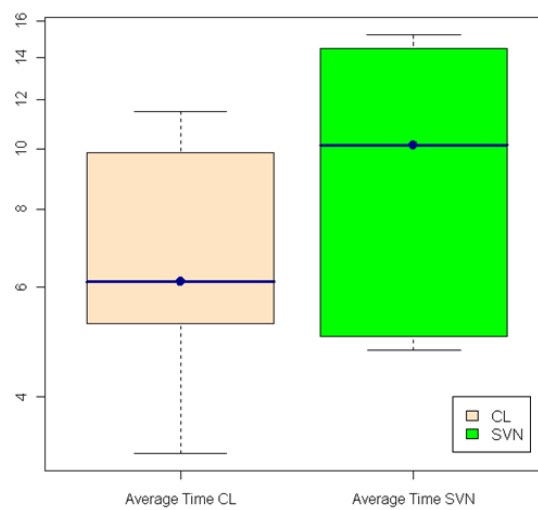


Figure 80: Average time for task execution (Experiment 1)

The efficiency improvement was also confirmed for each of the roles as shown in Figure 81. CL was 33% and 20% respectively more efficient than SVN. Family Engineering took for both groups more time than Application Engineering. In the experiment framework engineering was only about finding the core assets existing on each student's machine and putting them under evolution control. However the involved commands both for the CL and the SVN group were slightly more complicated than the respective commands for Application Engineering and this caused the additional effort.



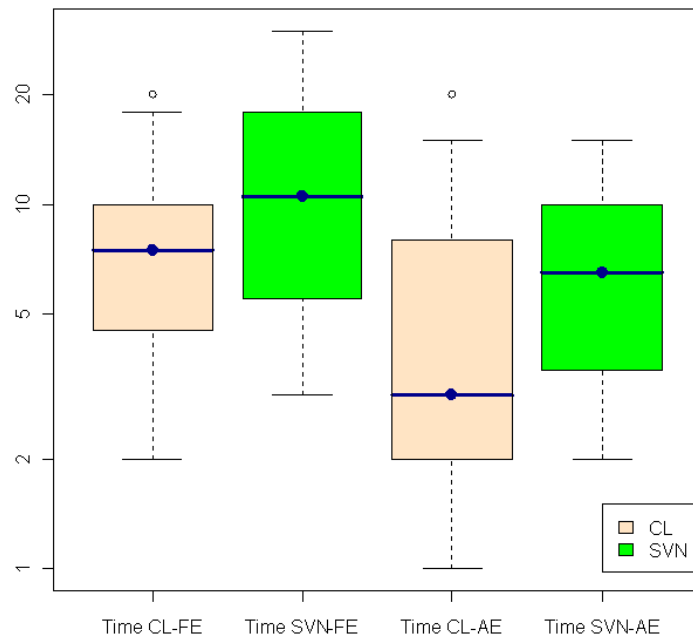


Figure 81: Times per Group and Role

The better performance of the CL group against the SVN group was mainly due to Tasks 3 and 4 only as it can be shown in Figure 82. These tasks entail by nature more complexity than the other tasks and the automation brought by the Customization Layer paid-off in these cases.

As depicted in Figure 82 the CL group was generally slower with Tasks 1 and 2. These tasks were relatively easy to accomplish. Therefore the SVN group had an advantage in this case because it used TortoiseSVN which has a much higher usability than the prototypical Customization Layer. The answers in the feedback forms showed indeed that some students had difficulties at the beginning of the experiment with the command line interface of the prototypical Customization Layer. The difficulties were concentrated on the correct input of the evolution control commands as well as on the interpretation of the command output.

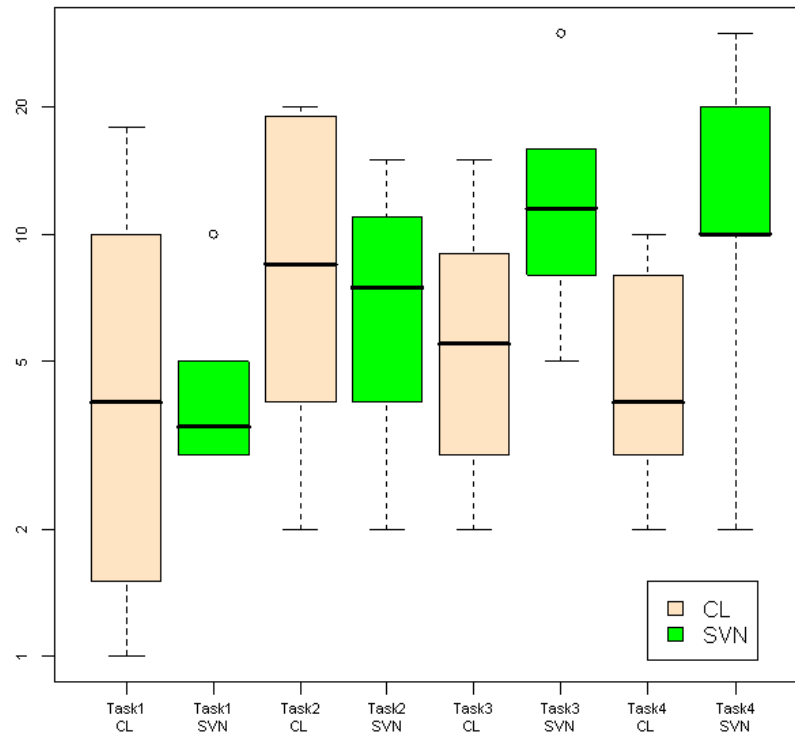


Figure 82: Times per Group and Task

In order to evaluate hypothesis H2 the number of versions and branches produced by each group has been first collected from the log files. The results are shown in Figure 83. CL generated significantly more versions and revisions than SVN. Therefore hypothesis H2.1 is not supported.

One minor reason for the increased complexity was that the CL group was bigger than the SVN group. However the main reason was the automation brought by the Customization Layer, which simplifies the creation of versions and revisions. Given this simplification the students experimented with the tool and created more versions and branches than necessary. On the other hand the SVN students created only the versions and branches as required by the tasks. For the Customization Layer approach this is an indication that a Customization Layer implementation must realize a set of versioning or change management rules so that the creation of unnecessary versions and branches can be avoided. The current prototypical implementation does not implement such rules and evolution control commands can be issued at will.

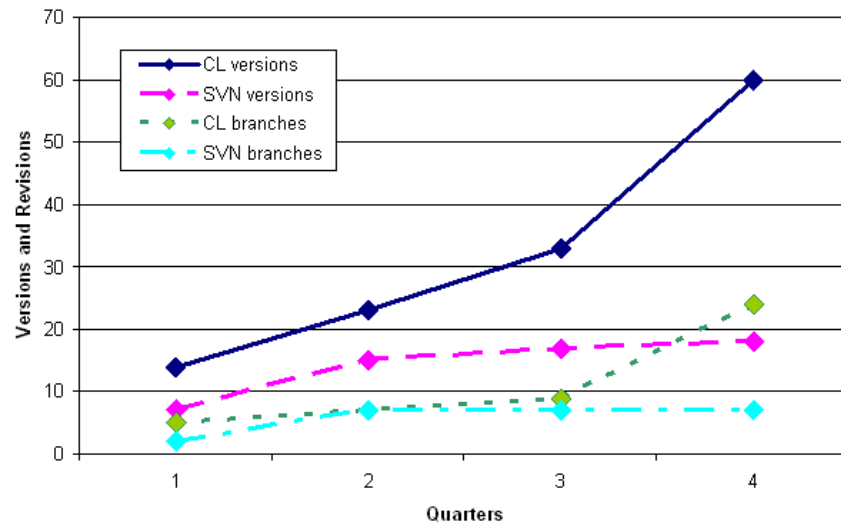


Figure 83: Versions and Branches produced over time

For the evaluation of sub-hypothesis H2.2 the detail answers have been analyzed, which the students gave for Tasks 3 and 4, the more difficult tasks of the experiment. 80% of the SVN answers were wrong. In other words SVN students gave wrong answers with respect to the number of changes in core assets and instances. This was an indication that H2.2 is supported. As the complexity of the repository grew during the experiment, the students had increasing difficulties in tracking the changes. On the other hand this did not apply to the CL group since the Customization Layer performs change tracking automatically and enables filtering the results when necessary.

At this point it was also important to observe the influence of student's product line engineering (PLE) and configuration management (CM) experiences to the dependent variables. Following table shows the distribution of experience across students and groups.

| Student ID | CM Experience | PLE Experience | Group |
|------------|---------------|----------------|-------|
| 1          | Substantial   | Little         | CL    |
| 2          | Little        | No data        | CL    |
| 3          | Average       | Little         | SVN   |
| 4          | Professional  | Little         | SVN   |
| 5          | Little        | Little         | SVN   |

| Student ID | CM Experience | PLE Experience | Group |
|------------|---------------|----------------|-------|
| 6          | Average       | Average        | CL    |
| 7          | None          | Little         | CL    |
| 8          | Professional  | Substantial    | CL    |
| 9          | Substantial   | Little         | CL    |
| 10         | None          | None           | CL    |
| 11         | Average       | Little         | SVN   |
| 12         | Substantial   | Substantial    | SVN   |
| 13         | Substantial   | Substantial    | CL    |
| 14         | Average       | Little         | SVN   |

Table 27: Distribution of student experience

The PLE experience did not have any influence on the dependent variables, since a basic understanding of product line engineering concepts were sufficient for the execution of the experiment. On the other hand CM experience could play an important role. It was reasonable to assume that SVN students with professional CM experience would perform as good as CL students or even better than them. Regarding hypothesis H1 and as shown in Figure 84 experienced SVN students had in total a slightly better performance than experienced and inexperienced CL students. However in Tasks 3 and 4 the CL group performed again better. Regarding hypothesis H2 no deviations could be observed. Even the SVN students who claimed professional configuration management experience did not provide fully correct answers.

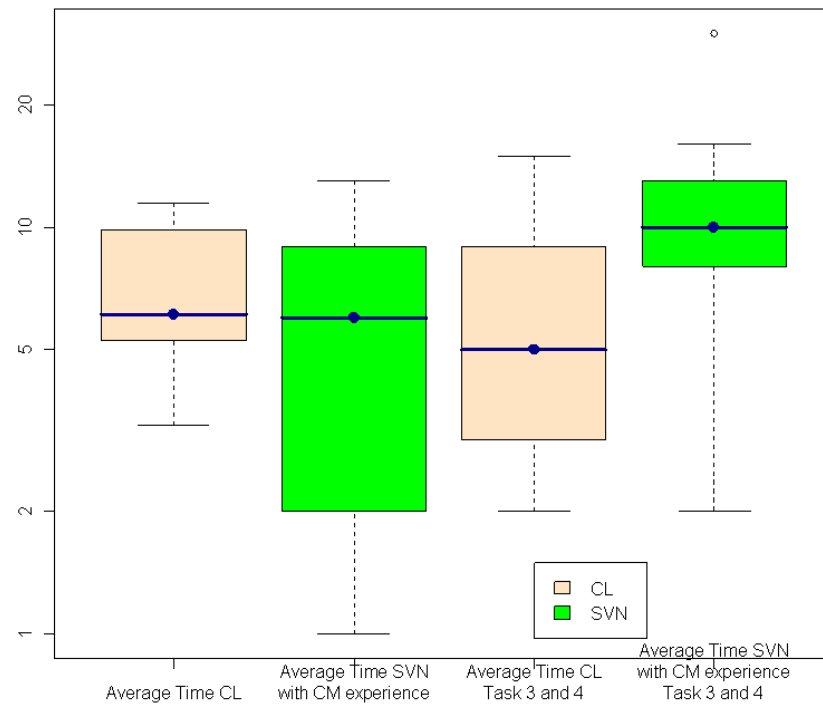


Figure 84: Influence of experience to efficiency

### 9.3.4 Experiment 2

The second experiment that was performed in the context of this thesis aimed at obtaining further data about the hypothesis H1 and was set-up identically to the first experiment. There were however less subjects in this case (9 students have participated in the experiment). The result was less positive in this case. The coordination effort required by users of the Customization Layer was 22% less in average than the corresponding effort of the direct version management usage.

However the second experiment showed a different picture with respect to the distribution of the effort across the tasks. In this experiment most savings, obtained through the usage of the Customization Layer, were achieved in the first task (creation of assets). For the rest of the tasks the Customization Layer achieved similar or worse results compared to the direct usage of subversion. Figure 85 provides the corresponding box-and-whisker diagrams.

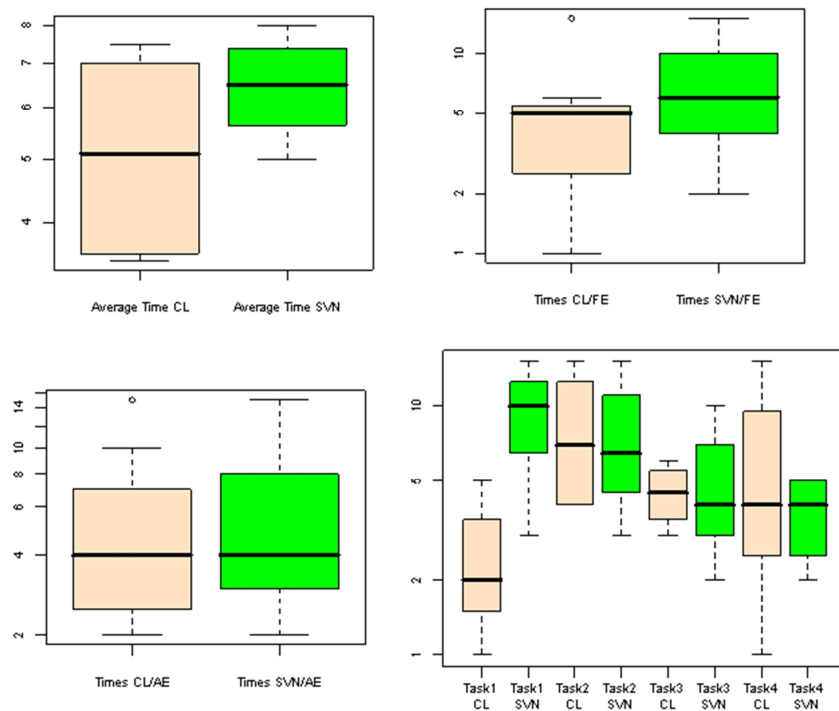


Figure 85: Summary results of Experiment 2

The analysis of the feedback forms revealed that the usability issues of the Customization Layer prototype had a much bigger impact on this experiment than on the first one. In particular, two students had several difficulties in issuing the commands correctly and in interpreting the results. In this regard the outputs of the two students can be considered as outliers. By filtering out these outliers the situation changes positively, as shown Figure 86.

Nevertheless the usability issue of the current prototype remains important and has to be improved in further versions of the tool. At this point it must be also noticed that in both experiments the subversion users (i.e. members of the SVN group) were not using a command line interface to subversion, although this would have been possible. The reason was educational. One of the goals of the experiment was also to show to the students the possibilities of the graphical TortoiseSVN client. Given this fact, it is expected that a comparison of the Customization Layer prototype against a command line interface to subversion would cause more positive results.

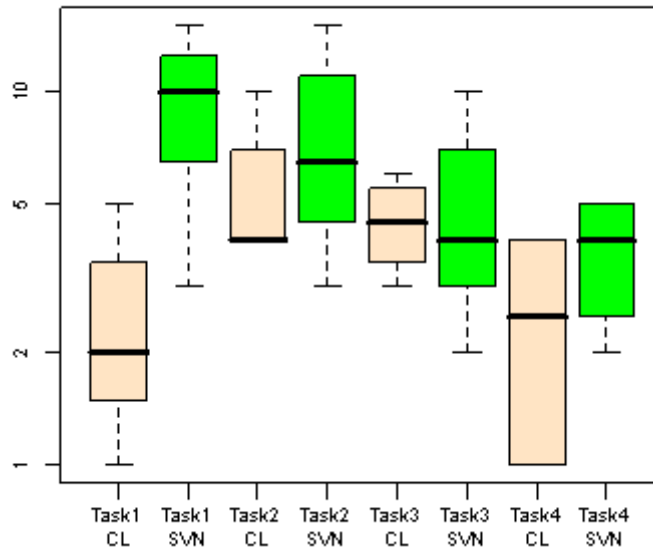


Figure 86: Experiment 2 task times without outliers

Another difference of the second experiment compared to the first one lied in the feedback forms that were used. The second experiment enhanced the feedback forms with additional questions from the UTAUT (Unified Theory of Acceptance and Use of Technology) [VMD+03]. UTAUT provides a set of questions that can be used for the evaluation of a technological solution. Based on these questions the students were asked to evaluate the Customization Layer as well as TortoiseSVN. The questions were grouped in five categories (the technology can be replaced by Customization Layer or TortoiseSVN in the following):

- **Performance expectancy** is defined as the degree, to which users believe that using the technology will help him or her to attain gains in job performance.
- **Effort expectancy** is defined as the degree of ease associated with the use of the technology
- **Attitude** toward using the technology is defined as the overall affective reaction to using the technology
- **Social influence** is defined as the degree to which a user perceives that important others believe he or she should use the technology

- **Facilitating conditions** are defined as the degree to which a user believes that an organizational and technical infrastructure exists to support the use of the technology

The students were requested to answer to a total of 20 questions from the above categories (all details of the UTAUT form are provided in the appendix). Each question could be answered with a numerical value ranging from 1 to 7. A low value corresponded to high acceptance of the technology (i.e. 1 = “fully agree”) and a high value corresponded to low acceptance (i.e. 7 = “fully disagree”). Table 28 provides the summary of the UTAUT evaluation.

| Category                | Customization Layer | TortoiseSVN |
|-------------------------|---------------------|-------------|
| Performance expectancy  | 1,91                | 2,08        |
| Effort expectancy       | 1,66                | 3,33        |
| Attitude                | 2                   | 2,75        |
| Social influence        | 3,25                | 3,33        |
| Facilitating conditions | 1,8                 | 2,58        |

Table 28: Evaluation based on UTAUT

The Customization Layer generally received a better evaluation than TortoiseSVN in spite of the usability problems during the operation. This is an indicator of the good potential of the Customization Layer that was perceived by the users. Most importantly with respect to hypothesis H1 is the effort category, which in the case of the Customization Layer received a significantly better evaluation. Although users had difficulties in the interaction with some of the Customization Layer commands, they generally perceived the interaction with the tool as clear and understandable. An influencing factor at this point was the explanation that the students received during the experiment and when the problems with the Customization Layer commands arose.

## 9.4 Case Study

A further validation of the Customization Layer approach has been undertaken in terms of a case study with an international manufacturer of agricultural machinery. In this context the organization also provides software-intensive agricultural management solutions that are subject to significant variation due to the diversity of the underlying embedded



systems. Therefore the adoption of a product line engineering approach is being considered at the time of writing.

#### 9.4.1 Setting

Figure 87 provides the Customization Layer specification for the case study in terms of evolution control processes and scenarios. The scenarios haven been obtained through interaction with the corresponding stakeholders and involve a pair of family and application engineering processes. The LibraryDevelopment process maps to the development of a reusable software library while the LibraryApplicationToGS2630 corresponds to the process that applies the library to a particular product. In this context four stakeholders have been initially asked to fill a questionnaire (see Appendix C.2) with the goal to capture current problems and expectations. The answered questionnaires confirmed partially the problems addressed by this thesis and indicated the lack of adequate tooling that addresses them.

Mercurial [URL21] is used in this case as the underlying CMS. As a distributed version management system Mercurial allows each developer to have an own repository of configuration items. Each repository can manage a series of branches and furthermore changes in branches can be propagated between repositories.

```

ProductLine DisplayProductLine{
FamilyEngineering LibraryDevelopment{
 VersionManagementFE LibraryDevelopmentVM{
 createCoreAsset makeFileShared,
 removeCoreAsset removeLibraryAsset
 },
 StatusAccountingFE LibraryDevelopmentSA{
 showCoreAssetInstances findSharesOfFiles,
 showProductAssetChangesSinceLastSynchronization
 allChangesSinceLastMergeReturn
 }
}
ApplicationEngineering LibraryApplicationToGS2630{
 VersionManagementAE LibraryVMIn2630,
 StatusAccountingAE LibrarySAIn2630 {
 showCoreAssets findSharedFiles,
 showCoreAssetChangesSinceLastSynchronization
 allChangesSinceLastMergeOneWay
 }
}
}

```

Figure 87: Customization Layer of the case study

### 9.4.2 Implementation and experiences

A Mercurial connector implementation has been performed by the author of this thesis (a class diagram of the implemented connector can be found in appendix C.1. In this case 2 person weeks were approximately required. Good support for marking facilitated the implementation significantly. On the other hand, at the time of writing it is officially recommended to use the command line interface even for the programmatic access to Mercurial. To this end the connector had to provide a set of operations (depicted as private operations in the following) to issue commands to the command line interface and also to interpret the corresponding results. This slowed down the implementation process significantly.

For the purpose of this case study it was not necessary to provide an executable Customization Layer in terms, for example, of a console application. In a first step it was sufficient to show how product line scenarios can be realized in an automated way with Mercurial. To this end a test suite has been provided that performs various unit tests on the evolution control scenarios. Appendix C.1 provides an overview of the Mercurial connector implementation and the test suite.

For the implementation two main features of Mercurial connector have been used:

- Distributed repositories: Each product line engineering process (i.e. family and application engineering) has been mapped to a corresponding Mercurial repository. Therefore there has been a repository holding core assets and various other repositories that held instances and product-specifics. Change propagation between repositories can be done mainly in two ways: through pulling or through pushing. In the pull mode the repository interested in changes has to actively retrieve them from another repository. In the push mode on the other hand, a repository that performs changes can notify others by submitting
- Tagging: Mercurial provides good support for tagging and particular configuration item versions. Therefore tagging was used to mark and to retrieve Customization Layer information. The creation of a core asset for example leads to a configuration item, whose first version carries a tag that can be recognized by users and also by the Customization Layer. Since however tags have to be unique each tag was enriched with a time stamp.

### 9.4.3 Results and recommendations

The significance of this case study is undoubtedly questionable. Only 4 software developers of the manufacturer participated in the case study survey. Although the corresponding findings were positive they cannot be generalized due to the small number of participants. The next weakness of this case study lies clearly in the implementation part. The implementation was undertaken by the author of this thesis. The plan was to deliver the resulting Customization Layer connector to the manufacturer in order to obtain usage data. Yet, upon finalization of the layer the contact persons at the manufacturer's side were unavailable due to other activities. Finally, the delivery of the layer was suspended due to a shift of priorities in the manufacturer. Hence the experiences that can be extracted of this case study are restricted to the initial interactions with the software developers and to the implementation of the Mercurial connector.

In order to avoid such situations it is recommended to keep the contact with the organization participating in a case study constantly active. Surveys should be performed in terms of live interviews and distribution of questionnaires should be avoided. Early versions of prototype tools have to be delivered as soon as possible in order to obtain feedback and to avoid unnecessary effort. The most important factor is however the presence of a contact person in the participating organization that is interested in the case study and also has enough influence to deal with internal developments that might have a negative impact to the study. That contact person can be surely assisted in this role if the added value expected by the case study is constantly illustrated in terms of clear examples that are gaining importance as the prototypical tools evolve. These recommendations could not be satisfied in the context of this case study and this is the reason for the reduced significance of the results.

## 9.5 Section summary

This section presented activities that have been performed in order to validate the Customization Layer approach. A structural evaluation first assessed the Customization Layer approach with respect to its design decisions and the related tradeoffs and risks. Subsequently a usability evaluation, two controlled experiments and a case study have been described. The usability evaluation showed clear advantages of the Customization Layer approach as opposed to conventional version management. These advantages were supported by the two experiments and a case study with an industrial partner although the significance of these studies can be put under question. Next section summarizes this thesis.

## 10 Conclusion

In this thesis the Customization Layer approach has been described, which enables evolution control in a product line context on the basis of configuration management. The approach consists of six components:

- **Conceptual Model** (section 3): The model encapsulates the evolution control concepts that pertain to product line engineering and enables description of evolution control processes within an organization.
- **Data Model** (section 4): The model describes the entities and relations that are produced and controlled during the evolution of a product line.
- **Process Model** (section 5): The model specifies the scenarios necessary for evolution control of a product line.
- **Interaction with Configuration Management** (section 6): A set of guidelines facilitate the implementation of evolution control on the basis of configuration management
- **Implementation framework** (section 7): An implementation framework facilitates the implementation of a console application for evolution control.
- **Adoption Process** (section 8): The process specifies steps based on the Quality Improvement Paradigm that are necessary to introduce evolution control to an organization

### 10.1 Research Questions

Section 1.3.2 introduced a series of research questions. The following paragraphs discuss how these questions were answered in the context of this thesis.

**Research Question 1 “Granularity”:** In the context of this thesis the term core assets was used to refer to reusable assets. The Basic Asset Model introduced in section 4 specified the internal structure of core assets. According to the model a core asset can contain a reuse contract that guides development with reuse for the respective asset. Such a reuse contract can be realized in term of variability management

approaches as discussed in section 4.4. Moreover sections 4.1.2, 5.1.2 and 5.2.2 discussed strategies to deal with granularity of core assets and its meaning to core asset instantiation.

**Research Question 2** *“Tracking software reuse”*: In order to track software reuse this thesis proposes to relate core assets to core asset instances. While core assets are developed for reuse across a complete product line, instances are derived out of core assets by development with reuse. Instances and product-specific assets constitute the elements of a product, member of a product line. The basic asset model (section 4) specifies the association between core assets and instances. In this way the rationale behind the derivation of an instance, i.e. the decisions that lead to an instance, is also captured.

**Research Question 3** *“Avoiding product line decay”*: In order to ensure that reusable assets (i.e. core assets) are continuously reused in a product line context, this thesis proposes activities for the identification and propagation of changes between core assets and their instances. Section 5 introduces the scenarios necessary in order to avoid this kind of decay.

**Research Question 4** *“Take advantage of existing configuration management systems”*: The Customization Layer approach proposed in this thesis does not neglect an existing configuration management system. On the contrary it sets-up an automation layer on top of existing systems. Section 6 discusses interaction with different types of configuration management functionality and section 9 discusses concrete implementations in terms of two well-known version management systems.

## **10.2 Validation**

The work described in the present thesis has been validated by means of a structural evaluation, a usability evaluation, two experiments and a case study. The validations indicated that the overall effort for the management of product line evolution, with the help of configuration management, can be reduced up to about 30% by means of a Customization Layer. Efficiency of evolution control operations can be reduced and the other hand effectiveness of the operations can be increased. However the validations also indicated that the user interface friendliness of such an automation layer is an important factor that influences effort savings.

## **10.3 Limitations**

The evolution control method introduced in this thesis addresses the coordination of family and application engineering processes. However a

product line engineering process also included the process of scoping. The latter defines the product portfolio to be supported by a product line and defines commonality and variability accordingly. Moreover scoping analyzes the potential of investing in software reuse within the product line. A product line scope can be defined as the set of assets that comprises the output of this kind of activities.

Scoping plays a central role in the early stages of a product line development, as it sets the ground for the further activities. However during the evolution of a product line, the scope must be also evolved. This applies in particular when the members of a product line encounter new requirements. The latter might be implemented in a product-specific manner within application engineering or in reusable manner within family engineering. The product line scoping process has to be involved in this case in order to judge the potential for software reuse.

Therefore scoping also involves a set of evolution control activities. The current work however does not address them explicitly. The basic asset model (section 4.1) proposes the application of reuse contracts in order to establish an explicit connection between core asset and the product line scope. Furthermore section 5.4 provides guidelines for change impact analysis of core assets and describes how the product line scope can be taken into account when change requests emerge. However a detailed description of the scoping activities and the relation to the family and application engineering activities is not provided in the current work.

In a product line context it is possible that product-specific assets need to be made reusable. This can happen when properties of such assets become beneficial for the whole product line. The Customization Layer approach enables creating a core asset out of a product-specific asset by selecting the repository location of the asset as source location of the creation operation. However the current work does not provide support towards identification of product-specific assets that are good candidates for becoming core assets. To this end a Customization Layer needs to be coupled with reverse engineering techniques that are able to detect such assets.

Another limitation of the current work relates to the issue of asset comparison (i.e. differencing). The coordination between core asset and instances requires comparing these two different types of assets. However since instances are derived from core assets traditional comparison mechanisms (e.g. the traditional diff utility) can be applied but do not provide useful results. Section 5.4.2 introduced a formal model in this regard, however a detailed specification or implementation of the necessary algorithms is not provided in the current thesis.

Finally, this thesis defines a set of evolution control scenarios for product lines. However, this work does not define processes or workflows entailing these scenarios. If product line evolution control can be thought off as a software component, scenarios are the services that it offers. The latter should be combined in higher-order workflows that invoke the scenarios in order to achieve a concrete value to the organization.

## **10.4 Future Work**

The work described in the present thesis can be continued in two main directions: tool support and industrial validations. The latter involves carrying out additional long-term experiments and usability evaluations with complete user interfaces. The goal of these activities would be to observe the effects of a Customization Layer over several iterations in the QIP process.

Tool support on the other hand can address the following activities:

- Implementation of evolution control scenarios for additional or new configuration management systems. Future work should consider further tools in this area such as git [URL22], which comes with a broad set of commands and simplifies branching and merging activities significantly.
- Extensions of the Xtend/Xpand infrastructure so that the implementation activities are further facilitated. This also should take the upcoming SCM Specification in [OSLC10] into consideration as it will possibly serve as a standard for future configuration management systems.
- Improvement of the current implementation framework including its connection to the Xtend/Xpand infrastructure
- Implementation of a full-fledge Customization Layer application including visualizations of product line evolution
- Implementation of the differencing model described in section 5.4.2.

Product line engineering involves among other things management of core assets. To this end various variability implementation mechanisms can be applied [AG01]. Preprocessor directives, aspect-oriented programming or template meta-programming are examples of such mechanisms. Configuration management is also a mechanism in that direction.

Figure 9 in section 1.3.5 depicts groups of branches or development lines. When configuration management is used as a variability implementation mechanism such a group can be seen as a core asset. That means that the derivation of an instance involves the identification and selection of a core asset configuration. This yields an editable instance and is not to be confused with build management that creates executable (i.e. compiled) instances out of core assets. The selected core asset configuration can be subsequently used and evolved in the context of a product, for example in a different configuration management repository than the original core asset. In other words a core asset implementation may offer a portfolio of already defined instances, stored in a set of branches. Some of these instances may in fact be employed in products. The Customization Layer approach can be extended in this case in order to be used as a variant authoring environment [Mah95]. That means that a product line engineer may issue queries to the Customization Layer to evolve a particular configuration of a core asset. Similar queries can be issues by application engineers to obtain configurations of core assets to be used in products.





## References

- [ABB+01] Atkinson, C.; Bayer, J.; Bunse, C.; Kamsties, E.; Laitenberger, O.; Laqua, R.; Muthig, D.; Paech, B.; Wüst, J. & Zettel, J. (2001), *Component-based Product Line Engineering with UML*, Addison-Wesley, London.
- [ACC10] Damon Poole, *Stream-Based Architecture for SCM*, AccuRev, Inc., whitepaper retrieved in April 2010 from [http://www.accurev.com/whitepaper/stream\\_based\\_architecture.htm](http://www.accurev.com/whitepaper/stream_based_architecture.htm)
- [ACT01] Ahern, D. M.; Clouse, A. & Turner, R. (2001), *CMMI Distilled. A Practical Introduction to Integrated Process Improvement*, Addison-Wesley, Boston.
- [AG01] Anastasopoulos, M. & Gacek, C. (2001), 'Implementing Product Line Variabilities', *ACM SIGSOFT Software Engineering Notes* 26(3), 109-117.
- [AKH00] Colin Atkinson, T. K. & Henderson-Sellers, B. (2000), 'To Meta or Not to Meta — That Is the Question', *Journal of Object-Oriented Programming*, SIGS Publications Vol. 13, No. 8, pp. 32–35.
- [AKS+10] Anastasopoulos, M.; Keuler, T.; Silva, A.; Wanisch, S. & Höh, M. (2010), *Architecture-centric configuration management; Controlling the evolution of large software systems*, *Commercial Vehicle Technology 2010; Proceedings of the 1st Commercial Vehicle Technology Symposium (CVT 2010)*, Shaker Verlag
- [ALB+11] Apel, S.; Liebig, J.; Brandl, B.; Lengauer, C. & Kästner, C. (2011), *Semistructured merge: rethinking merge in revision control systems*, in 'Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering', ACM, New York, NY, USA, pp. 190--200.
- [AM08] Åström, K. J. & Murray, R. M. (2008), *Feedback Systems: An Introduction for Scientists and Engineers*, Princeton University Press.
- [BA03] Berczuk, S. P. & Appleton, B. (2003), *software configuration management Patterns. Effective Teamwork, Practical Integration*, Addison-Wesley, Boston.
- [BA05] Beck, K. & Andres, C. (2005), *Extreme Programming Explained. Embrace Change*, Addison-Wesley, Boston.

- [Ba99] Bays, M. E. (1999), *Software Release Methodology*, Prentice Hall PTR, Upper Saddle River.
- [BAB+00] Boehm, B. W.; Abts, C.; Brown, W. A.; Chulani, S.; Clark, B. K.; Horowitz, E.; Madachy, R.; Reifer, D. J. & Steece, B. (2000), *Software Cost Estimation with COCOMO II*, Prentice Hall PTR, Upper Saddle River.
- [BBM+04] Boehm, B.; Brown, A. W.; Madachy, R.; & Yang, Y. (2004) 'A Software Product Line Life Cycle Cost Estimation Model', *Proceedings of the International Symposium on Empirical Software Engineering (ISESE 2004)*. Redondo Beach, CA, August 19-20, 2004. Los Alamitos, CA: IEEE Computer Society
- [BC05] F. Bachmann & P. C. Clements: *Variability in Software Product Lines*. Technical Report CMU/SEI-2005-TR-12, Software Engineering Institute, 2005
- [BCD94] Basili, V., Caldiera & G., Rombach D. 1994. Goal/question/metric paradigm. In *Encyclopedia of Software Engineering*. Vol. 1, J. C. Marciniak, Ed. John Wiley and Sons, New York, 528-532.
- [BCE+04] Bellon, S.; Czeranski, J.; Eisenbarth, T. & Simon, D. (2004), *A Product Line Asset Management Tool*, in 'Workshop on Software Variability Management: Software Product Families and Populations'.
- [BCR94] Basili, V.R, C. Caldiera, H.D. Rombach, 'Experience Factory', *Encyclopaedia of Software Engineering* (Marciniak, J.J., editor), Volume 1, John Wiley & Sons, 1994, pp. 469 – 476
- [BD09] Thomas Buchmann and Alexander Dotor: *Towards a Model-Driven Product Line for SCM systems*, in: *Proceedings of the 13th International Software Product Line Conference (SPLC)*, Vol. 2, San Francisco, USA, August 24-28, 2009.
- [Be04] Becker, M. (2004), *Anpassungsunterstützung in Software-Produktfamilien*, Technische Universität Kaiserslautern, Kaiserslautern.
- [BEM+12] Bordag, S., Eisenecker, U., Müller, R., Schorp, K. (2012) *Evaluierung alternativer Bedienkonzepte für Tablets und Co*, In: *iX 1/2012*, S. 66-71.
- [BH11] Barringer, H. & Havelund, K. (2011), *TraceContract: A Scala DSL for Trace Analysis*, in Michael Butler & Wolfram Schulte, ed., 'FM 2011: Formal Methods', Springer Berlin Heidelberg, , pp. 57-72.
- [BM77] Belady L., Merlin P.M. (1977), *Evolving Parts and Relations – A model of system families*, IBM Research Report RC6677

- [Bo02] Bosch, J. (2002), Maturity and Evolution in Software Product Lines: Approaches, Artefacts and Organization, in 'SPLC 2: Proceedings of the Second International Conference on Software Product Lines', Springer-Verlag, London, UK, pp. 257--271.
- [Bo96] Bohner, S.A. and R.S. Arnold, Eds. (1996). Software Change Impact Analysis. Los Alamitos, California, USA, IEEE Computer Society Press.
- [BSH80] Bersoff, E. H.; Siegel, S. & Henderson, V. (1980), Software Configuration Management: An Investment in Product Integrity, Prentice Hall Professional Technical Reference.
- [CAD03] Crnkovic, I.; Asklund, U. & Dahlqvist, A. P. (2003), Implementing and Integrating Product Data Management and Software Configuration Management, Artech House, Boston.
- [CE00] Czarnecki, K. & Eisenecker, U. (2000), *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley Professional.
- [Chr99] Christensen, H. B. (1999), 'Ragnarok: An Architecture Based Software Development Environment,' PhD thesis, Department of Computer Science, University of Aarhus, Aarhus.
- [CN02] Clements, P. & Northrop, L. (2002), software product lines. Practices and Patterns, Addison-Wesley, Boston.
- [CVL10] (2009), 'Common Variability Language (CVL), Request For Proposal'(Document: ad/2009-12-03), Object Management Group.
- [CW98] Conradi, R. & Westfechtel, B. (1998), 'Version Models for Software Configuration Management', ACM Computing Surveys 30(2), 232--282.
- [Da90] Dart, S. (1990), 'Spectrum of Functionality in Configuration Management Systems'(CMU/SEI-90-TR-11), Technical report, Software Engineering Institute; Carnegie Mellon University, Pittsburgh, Pennsylvania 15213.
- [DB91] Davis, A. M. & Bersoff, E. H. (1991), 'Impacts of life cycle models on software configuration management', *Commun. ACM* **34**(8), 104--118.
- [DES] National Institute of Standards and Technology, Federal Information Processing Standard 46-3: The official document describing the DES standard, retrieved in May 2011 from <http://csrc.nist.gov/>
- [DMN+06] Dig, D.; Manzoor, K.; Nguyen, T. N. & Johnson, R. (2006), 'Refactoring-aware software merging and configuration management', SIGSOFT Softw.

- Eng. Notes 31(6), 1--2.
- [DP00] Dutoit, A.; Paech, B. & München, T. U. (2000), 'Rationale Management in Software Engineering'.
- [Duv07] Duvall, Paul M. (2007). Continuous Integration. Improving Software Quality and Reducing Risk. Addison-Wesley. ISBN 0-321-33638-0.
- [EC95] Estublier, J. & Casallas R. (1995), The Adele configuration manager, John Wiley & Sons, Inc., New York
- [FSJ99] Fayad, M. E.; Schmidt, D. C. & Johnson, R. (1999), Implementing Application Frameworks. Object-Oriented Frameworks at Work, John Wiley & Sons, New York.
- [GCC+03] Garg, A.; Critchlow, M.; Chen, P.; Westhuizen, C. V. d. & Hoek, A. v. d. (2003), An Environment for Managing Evolving Product Line Architectures, in 'ICSM '03: Proceedings of the International Conference on Software Maintenance', IEEE Computer Society, Washington, DC, USA, pp. 358.
- [GHJ+95] Gamma, E.; Helm, R.; Johnson, R. & Vlissides, J. (1995), Design Patterns. Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading.
- [HB01] Herrejon, R. E. L. & Batory, D. (2001), A Standard Problem for Evaluating Product-Line Methodologies, in 'Proc. 2001 Conf. Generative and Component-Based Software Eng', Springer, , pp. 10--24.
- [HD05] Henkel, J. & Diwan, A. (2005), CatchUp!: capturing and replaying refactorings to support API evolution, in 'ICSE '05: Proceedings of the 27th international conference on Software engineering', ACM, New York, NY, USA, pp. 274--283.
- [HH07] Hendrickson, S. A. & van der Hoek, A. (2007), Modeling Product Line Architectures through Change Sets and Relationships, in 'ICSE '07: Proceedings of the 29th international conference on Software Engineering', IEEE Computer Society, Washington, DC, USA, pp. 189--198.
- [HM00] G.A. Hall, J.C. Munson: Software Evolution: Code Delta and Code Churn. JSS 54(2): 111-118, October 2000
- [IEEE1517-2010] (2010), 'IEEE Standard for Information Technology--System and Software Life Cycle Processes--Reuse Processes', *IEEE Std 1517-2010 (Revision of IEEE Std 1517-1999)*, 1 -51.

- [IEEE610.12] (1990), 'IEEE standard glossary of software engineering terminology', IEEE Std 610.12-1990
- [ISO/IEC 9899] (2005), 'Programming languages - C'(ISO/IEC 9899), International Standards for Business, Government and Society.
- [KCH+98] Kang, K. C.; Cohen, S. G.; Hess, J. A.; Novak, W. E. & Peterson, A. S. (1998), 'Feature-Oriented Domain Analysis (FODA) Feasibility Study.'
- [KKC00] Rick Kazman, Mark Klein, Paul Clements, ATAM: Method for Architecture Evaluation, Technical Report CMU/SEI-2000-TR-004, August 2000, Software Engineering Institute Carnegie Mellon University
- [Kr02] Krueger, C. W. (2002), Variation Management for Software Production Lines, in 'SPLC 2: Proceedings of the Second International Conference on Software Product Lines', Springer-Verlag, London, UK, pp. 37--48.
- [Kr92] Krueger, C. W. (1992), 'Software reuse', ACM Comput. Surv. 24(2), 131--183.
- [Kru03] C.W. Krueger: Towards a Taxonomy for Software Product Lines. PFE-5: 323-331 (LNCS3014), Springer-Verlag, 2003
- [KSK+94] Kuvaja, P.; Similä, J.; Krzanik, L.; Bicego, A.; Saukkonen, S. & Koch, G. (1994), Software Process Assessment and Improvement. The BOOTSTRAP Approach, Blackwell, Cambridge.
- [La91] Latour, L. (1991), A methodology for the design of reuse engineered Ada components, in 'SETA1: Proceedings of the first international symposium on Environments and tools for Ada', ACM, New York, NY, USA, pp. 103--113.
- [Le03] Jason Leonard (2003), Simplifying Product Line Development using UCM Streams, retrieved November 2011 from <http://www.ibm.com/developerworks/rational/library/1748.html>
- [Le04] Leon, A. (2004), Software Configuration Management Handbook, Second Edition, Artech House, Inc., Norwood, MA, USA.
- [Leh78] Lehman, M. (1978), 'Laws of Program Evolution - Rules and Tools for Programming Management,' Proceedings Infotech State of the Art Conference, Why Software Projects Fail?, 11/1-11/25
- [Leh80] Lehman, M. (1980), 'Programs, life cycles, and laws of software evolution', Proceedings of the IEEE 68(9), 1060-1076.

- [Leh96] Lehman, M. M. (1996), Laws of Software Evolution Revisited, in 'EWSPT '96: Proceedings of the 5th European Workshop on Software Process Technology', Springer-Verlag, London, UK, pp. 108--124.
- [Lo99] Kam Wing Lo (1999), Simulation Report Reuse and High Level Languages, University of South Carolina, report CS599, retrieved November 2010 from [http://sunset.usc.edu/classes/cs599\\_99/projects/reuse.pdf](http://sunset.usc.edu/classes/cs599_99/projects/reuse.pdf)
- [LSR07] Linden, F. v. d.; Schmid, K. & Rommes, E. (2007), Software Product Lines in Action : The Best Industrial Practice in Product Line Engineering, Springer-Verlag, Berlin.
- [LST78] Lientz, B. P.; Swanson, E. B. & Tompkins, G. E. (1978), 'Characteristics of application software maintenance', Commun. ACM 21(6), 466--471.
- [Mad08] Madachy, R. J. (2008), Software Process Dynamics, John Wiley & Sons, Hoboken.
- [Mah95] Mahler, A. (1995), Variants: keeping things together and telling them apart 'Configuration management', John Wiley & Sons, Inc., pp. 73--97.
- [MB06] Mohan K. & Ramesh B. (2006), 'Change Management Patterns in Software Product Lines', Communications of the ACM, v. 49, n. 12.
- [Mc69] McIlroy, D. (1969), Mass-produced Software Components, in J. M. Buxton; P. Naur & B. Randell, ed., 'Proceedings of Software Engineering Concepts and Techniques', pp. 138--155.
- [McG07] John D. McGregor (2007), 'CM - Configuration Change Management', Journal of Object Technology, vol. 6, no. 1, January-February 2007, pp. 7-15
- [McK84] McKee, J. R. (1984), 'Maintenance as a function of design', American Federation of Information Processing Societies:1984 National Computer Conference, 9-12 July 1984, Las Vegas, Nevada, USA, 187-193.
- [MD08] Mens, T. & Demeyer, S., ed. (2008), Software Evolution, Springer-Verlag, Berlin Heidelberg.
- [Me02] T. Mens. 2002. *A State-of-the-Art Survey on Software Merging*. IEEE Trans. Softw. Eng. 28, 5 (May 2002)
- [Me99] Mens, T. (1999), 'A formal foundation for object-oriented software evolution', PhD Thesis, Vrije Universiteit Brussel
- [Mit06] Mittermeir, R. T. Facets of Software Evolution Software Evolution and

- Feedback, John Wiley & Sons, Ltd, 2006, 71-93
- [MKL00] Mm, G. K.; Kahen, G. & Lehman, M. M. (2000), A Brief Review of Feedback Dimensions in the Global Software Process, in '2000 Workshop: Feedback and Evolution in Software and Business Processes', pp. 44--49.
- [ML88] Mahler, A. & Lampen, A. (1988), 'An integrated toolset for engineering software configurations', *SIGSOFT Softw. Eng. Notes* **13**, 191--200.
- [Mut02] Muthig, D. (2002), A Light-weight Approach Facilitating an Evolutionary Transition Towards software product lines, Fraunhofer IRB Verlag, Stuttgart.
- [oAw] openArchitectureWare User Guide, Version 4.3.1, retrieved in February 2011 from <http://www.openarchitectureware.org>
- [OCL01] (2001), 'Object Constraint Language'(06-05-01), Technical report, Object Management Group.
- [Od10] Martin Odersky, The Scala language specification, Version 2.8, November 2010, retrieved April 2011 from <http://www.scala-lang.org/>
- [OH92] P. Oman & J. Hagemester (1992) "Metrics for Assessing a Software System's Maintainability," 337-344. Conference on Software Maintenance 1992. Orlando, FL, November 9-12, 1992. Los Alamitos, CA: IEEE Computer Society Press.
- [Om02] van Ommering, R. (2002), Building product populations with software components, in 'ICSE '02: Proceedings of the 24th International Conference on Software Engineering', ACM, New York, NY, USA, pp. 255--265.
- [OSLC10] (2010) Open Services for Lifecycle Collaboration, SCM Specification, retrieved in October 2010 from <http://open-services.net/>
- [Pa10] Patzke, T. (2010), The Impact of Variability Mechanisms on Sustainable Product Line Code Evolution, in 'Software Engineering 2010 - Proceedings', GI - Gesellschaft für Informatik, Bonn, , pp. 189-200.
- [Pa62] Parzen, E. (1962), 'On Estimation of a Probability Density Function and Mode', The Annals of Mathematical Statistics 33(3), 1065--1076.
- [PBL05] Pohl, K.; Böckle, G. & Linden, F. v. d. (2005), Software Product Line Engineering : Foundations, Principles, and Techniques, Springer-Verlag, Berlin.



- [Pe98] Perry, D. E. (1998), Generic Architecture Descriptions for Product Lines, in 'Proceedings of the Second International ESPRIT ARES Workshop on Development and Evolution of Software Architectures for Product Families', Springer-Verlag, London, UK, pp. 51--56.
- [PF89] Ploedereder, E. & Fergany, A. (1989), 'The data model of the configuration management assistant (CMA)', SIGSOFT Softw. Eng. Notes 14(7), 5--14.
- [PK05] Page, B. & Kreutzer, W. (2005), The Java Simulation Handbook: Simulating Discrete Event Systems with UML and Java, Shaker Verlag.
- [Pou97] Poulin, J. S. (1997), Measuring Software Reuse: Principles, Practices, and Economic Models, Addison-Wesley, Reading.
- [Ra05] Ramachandran, M. (2005), 'Software reuse guidelines', SIGSOFT Softw. Eng. Notes 30(3), 1--8.
- [Re79] Trygve Reenskaug (1979), Models - Views – Controllers, Xerox PARC technical note, retrieved May 2011 from <http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html>
- [Rei95] Reichenberger, C. (1995), 'VooDoo a tool for orthogonal version management', Software Configuration Management, 61—79, Springer Berlin / Heidelberg.
- [Ro75] M. J. Rochkind, (1975), The Source Code Control System, IEEE Transactions on Software Engineering SE-1:4 pages 364–370.
- [RP05] Romagnoli, J. A. & Palazoglu, A. ( 2005 ), Introduction to process control / José A. Romagnoli, Ahmet Palazoglu, Marcel Dekker/CRC, New York
- [SB02] Schwaber, K. & Beedle, M. (2002), Agile Software Development with Scrum, Prentice Hall PTR, Upper Saddle River.
- [SB99] Svahnberg, M. & Bosch, J. (1999), 'Evolution in software product lines: Two cases', Journal of Software Maintenance 11(6), 391--422.
- [Sch99] Schmidt, D. C. (1999), 'Why Software Reuse has Failed and How to Make It Work for You', C++ Report.
- [SDN+04] Sinnema, M.; Deelstra, S.; Nijhuis, J. & Bosch, J. (2004), COVAMOF: A Framework for Modeling Variability in Software Product Families, in 'In Proceedings of the Third International Software Product Line Conference (SPLC).

- 
- [SPEM] (2008), 'Software & Systems Process Engineering Meta-Model Specification', (Document formal/2008-04-01), Object Management Group.
- [SSF+09] Anastasopoulos, M.; Avrasoglou, A.; Graubmann, P.; Gross, T.; Ibanez, V.; Jaeger, M.; Mansell, J.; Patzke, T.; Priggouris, N.; Schmid, R. & Schyr, C. (2009), State-of-the-art survey for Product Line, Deliverable D\_SP1\_R6.1\_M, Artemis Joint Undertaking Call 2008, Seventh Framework Programme of the European Commission
- [Sz98] Szyperski, C. (1998), Component Software. Beyond Object-Oriented Programming, Addison-Wesley, Harlow.
- [TH03] Thompson, J. M. & Heimdahl, M. P. E. (2003), 'Structuring product family requirements for n-dimensional and hierarchical product lines', Requirements Engineering 8(1), 42--54.
- [Th12] Cheng Thao, A Configuration Management System for Software Product Lines, Theses and Dissertations. Paper 14, University of Wisconsin-Milwaukee, 2012
- [Boy13] J. Boyland, A. Greenhouse, and W. L. Scherlis, "The Fluid IR: An internal representation for a software engineering environment.", retrieved in November 2013 from <http://www.fluid.cs.cmu.edu>
- [UKR09] Uelschen, D. M.; Kniep, T. & Rodenbach, T. (2009), 'Konfigurationsmanagement, Fehlerverfolgung und Test', OBJEKTspektrum 02/2009, 24-32.
- [UML] (2009), 'Unified Modeling Language (UML), Version 2.2'(Document: formal/2009-02-02), Object Management Group.
- [UML03] (2003), 'UML 2.0 Infrastructure Specification' (ptc/03-09-15), OMG Adopted Specification, Object Management Group.
- [URL1] Homepage of the Eclipse IDE, retrieved in April 2010 from <http://www.eclipse.org/>
- [URL10] Homepage of the Java Compiler Compiler, retrieved April 2010 from <https://javacc.dev.java.net/>
- [URL11] Homepage of PLUM, retrieved April 2010 from <http://www.esi.es/plum/>
- [URL12] Homepage of Biglever, retrieved April 2010 from <http://www.biglever.com/>

- [URL13] Homepage of CM2 (configuration menu language), retrieved in April 2010 from <http://catb.org/~esr/cml2/>
- [URL14] Homepage of GNU make build automation utility, retrieved in April 2010 from <http://www.gnu.org/software/make/>
- [URL15] Homepage of Maven, retrieved in May 2010 from <http://maven.apache.org/>
- [URL16] Homepage of OSGi, retrieved in May 2010 from <http://www.osgi.org/>
- [URL17] Homepage of TortoiseSVN, retrieved in May 2010 from <http://tortoisesvn.tigris.org/>
- [URL18] Homepage of Ant, retrieved in November 2011 from <http://ant.apache.org/>
- [URL19] Homepage of the Xtext language development framework, retrieved in February 2011 from <http://www.eclipse.org/Xtext/>
- [URL2] Homepage of the Eclipse Modelling Framework, retrieved in April 2010 from <http://www.eclipse.org/emf/>
- [URL20] Homepage of StarTeam, retrieved April 2011 from <http://www.borland.com/us/products/starteam/index.aspx>
- [URL21] Homepage of Mercurial, retrieved in May 2011 from <http://mercurial.selenic.com/>
- [URL22] Homepage of Mercurial, retrieved in January 2013 from <http://git-scm.com/>
- [URL3] Homepage of the EMFText generator of textual EMF-based syntaxes, retrieved in April 2010 from <http://www.emftext.org>
- [URL4] Homepage of the Standard Widget Toolkit, retrieved April 2010 from <http://www.eclipse.org/swt/>
- [URL5] Homepage of SVNKit, retrieved April 2010 from <http://svnkit.com/>
- [URL6] Homepage of Subversion, retrieved April 2010 from <http://subversion.apache.org/>
- [URL7] Homepage of pure-systems, retrieved April 2010 from <http://www.pure-systems.com/>

- 
- [URL8] Homepage of Microsoft team system, retrieved in April 2010 from <http://www.microsoft.com/Visualstudio/products/teamsystem>
- [URL9] Homepage of JIRA, retrieved in April 2010 from <http://atlassian.com/software/jira>
- [VMD+03] Venkatesh, V.; Morris, M. G.; Davis, G. B. & Davis, F. D. (2003), 'User Acceptance of Information Technology: Toward a Unified View', *MIS Quarterly* 27(3), 425--478.
- [Whi00] White, B. A. (2000), *Software Configuration Management Strategies and Rational ClearCase. A Practical Introduction*, Addison-Wesley, Boston.
- [Wit96] Withey, J. (1996), 'Investment Analysis of Software Assets for Product Lines'(CMU/SEI-96-TR-010), Technical report, Software Engineering Institute, Carnegie Mellon University.
- [WRH+00] Wohlin, C.; Runeson, P.; Höst, M.; Ohlsson, M. C.; Regnell, B. & Wesslen, A. (2000), *Experimentation in Software Engineering. An Introduction*, Kluwer Academic Publishers, Boston.
- [WRL+93] Cathleen Wharton, John Rieman, Clayton Lewis, and Peter Polson, The cognitive walkthrough method: A practitioner's guide. Technical report #93-07, Institute of Cognitive Science, University of Colorado, Boulder
- [WYF03] Washizaki, H.; Yamamoto, H. & Fukazawa, Y. (2003), A Metrics Suite for Measuring Reusability of Software Components, *in 'In Metrics'*, pp. 211--223.
- [YFM+08] Yoshimura, K.; Forster, T.; Muthig, D. & Pech, D. (2008), 'Model-based Design of Product Line Components in the Automotive Domain', 12th International Software Product Line Conference, SPLC 2008 – Proceedings.
- [YR06] Yu, L. & Ramaswamy, S. (2006), 'A Configuration Management Model for Software Product Line.' *INFOCOMP Journal of Computer Science* 5(4), 1--8.
- [Ze97] A. Zeller, (1997) "Configuration management with version sets - a unified software versioning model and its applications," Ph.D. dissertation, Technische Universitaet Braunschweig



## Appendix



## Appendix A Implementation with XText/Xpand

This section provides examples of implementation guidelines in the Xtext format. These examples show how the guidelines can be specified in a more technical manner so that they can be directly related to evolution activities and scenarios as described in sections 3.8 and 3.9.

Obtaining a fully functioning implementation is not possible at this point, since this requires detailed information (at the level of an application programming interface) about the CMS at hand. With the rise of the OSLC standard [OSLC10] this may however change in the future.

In order for the guidelines to be more informative, pseudo code with the CL prefix is to be realized in terms of a Customization Layer; that is by operating on the conceptual model and on the basic asset model (section 3.7 and 4.1). Pseudo code with the CMS prefix is to be realized in terms of a connector to the underlying CMS. The VM prefix means that pseudo code is to be realized through a connector to a variability management tool. Finally, no prefix requires standard programming language statements.

Code marked with angle quotes (« ») marks statements of the Xpand language [oAw]. In the context of this work Xpand enables creating pseudo-code templates, which are specialized according to the selected CMS functionality. After specialization the Xpand statements are removed. Angle quotes will be used in the following only in the cases, in which the selection of an implementation strategy is clear. There will be however cases (i.e. putting markers) in which the decision depends on the concrete CMS at hand and is therefore left open.



```

«DEFINE createCoreAssetChangeRequest(String givenNameOfScenario) FOR ProductLine-»
String «givenNameOfScenario»(String caID, boolean synchronizeInstances)
 throws CoreAssetCRCreationException;

/*{
 CMS/retrieve core asset object based on caID
 CL/assign core asset to the change request
 CMS/create the core asset change request with the CMS
 «EXPAND StateManagement("changes pending")-»
 if (synchronizeInstances) {
 CMS/find instances of core asset
 for each instance {
 CL/ask user whether to propagate to_
 CL/_further core assets
 CL/invoke createProductAsset__ChangeRequest
 CL/_scenario accordingly
 «IF hasTicketHierarchy(this)-»
 CMS/assign core asset_
 CMS/_change request as parent
 «ELSE-»
 CMS/create custom field 'parent CR'
 CMS/set value of field to the just_
 CMS/_created core asset change request
 «ENDIF-»
 }
 }
 return change request creation result from CMS
}*/
«EXPAND File("CoreAssetCRCreationException")-»
«ENDDDEFINE»

```

```

«DEFINE StateManagement(String s) FOR ProductLine-»
«IF hasExplicitStateManagement(this)-»
 CMS/change state to «s-»
«ELSE-»
 CMS/use other strategy to set state
«ENDIF-»
«ENDDDEFINE»

```

```

«DEFINE createProductAssetChangeRequests(String givenNameOfScenario) FOR ProductLine-»
String «givenNameOfScenario»(String paID, boolean synchronizeInstances)
 throws ProductAssetCRCreationException;

/*{
 CMS/find product asset with the given ID
 CMS/create the product asset change request with the CMS
 if (synchronizeCoreAssets) && the product asset is an instance) {
 CMS/find core assets that correspond to instance
 for each core asset {
 CL/ask user whether to synchronize_
 CL/_with all its other instances
 CL/invoke createCoreAssetChangeRequest_
 CL/_scenario accordingly
 «IF hasTicketHierarchy(this)-»
 CMS/assign instance_
 CMS/_change request as parent
 «ELSE-»
 CMS/create custom field 'parent CR'
 CMS/set value of field to the just_
 CMS/_created instance change request
 «ENDIF-»
 }
 }
 return change request creation result from CMS
}*/
«EXPAND File("ProductAssetCRCreationException")-»
«ENDDDEFINE»

```

```

«DEFINE createCoreAsset(String givenNameOfScenario) FOR ProductLine-»
 void «givenNameOfScenario» (String sourceLocation,
 String targetLocation,
 String templateLocation,
 String depth
) throws CoreAssetCreationException;
/*{
 if (source location has contents) {
 CMS/load core asset contents from _
 CMS/_source to target location
 }
 else
 if (templateLocation != null) {
 CMS/create core asset in target_
 CMS/_location according to template
 }
 CMS/mark resulting configuration_
 CMS/_item as core asset
 CMS/mark all other configuration_
 CMS/_items according to depth as core assets
 CMS/set item states to "not released"
 if (connector to Variability Management available) {
 CL/obtain reuse contract from Variability Management
 «IF hasIntentionalVersioning(this)-»
 CMS/set reuse contract as predicate
 «ELSE-»
 CMS/set reuse contract as user-defined property
 «ENDIF-»
 }
}*/
«EXPAND File("CoreAssetCreationException")-»
«ENDEDEFINE»

```

```

«DEFINE removeCoreAsset(String givenNameOfScenario) FOR ProductLine-»
 void «givenNameOfScenario»(String caID) throws CoreAssetDeletionException;
/*
 CMS/find core asset based on ID
 CMS/find instances of core asset
 for each instance
 CMS/remove instance markers
 CMS/delete core asset
*/
«EXPAND File("CoreAssetDeletionException")-»
«ENDEDEFINE»

```

```

«DEFINE modifyCoreAsset(String givenNameOfScenario) FOR ProductLine-»
 void «givenNameOfScenario»(String caID) throws CoreAssetModificationException;
/*
 CMS/retrieve core asset object based on caID
 CL/ask user for new name
 CL/ask user for new configuration item
 CL/ask user for new instances
 if (connector to Variability Management available) {
 VM/check for new reuse contract in variability management
 CL/update core asset upon user confirmation
 }
 CMS/submit changes to change management system
*/
«EXPAND File("CoreAssetModificationException")-»
«ENDEDEFINE»

```

```
«DEFINE integrateCoreAsset(String givenNameOfScenario) FOR ProductLine-»
 void «givenNameOfScenario»(String caID) throws CoreAssetIntegrationException;
/*
 CMS/locate core asset based on caID
 /** a posteriori integration
 CL/obtain user confirmation
 CMS/mark last change on asset as integration
 /** session-based integration
 CMS/locate instances of core asset
 for each instance{
 CMS/merge last change into core asset
 CMS/mark merge as integration merge
 }
 «EXPAND StateManagement("integrated")->»
*/
«EXPAND File("CoreAssetIntegrationException")->»
«ENDDDEFINE»
```

```
«DEFINE setReleaseStateOfCoreAsset(String givenNameOfScenario) FOR ProductLine-»
 void «givenNameOfScenario»(String caID, String state)
 throws CoreAssetReleaseException
/*
«EXPAND StateManagement("state")->»
*/
«EXPAND File("CoreAssetReleaseException")->»
«ENDDDEFINE»
```

```

«DEFINE createProductAsset(String givenNameOfScenario) FOR ProductLine-»
 enum InstantiationStrategy { DEEP, SHALLOW };
 void «givenNameOfScenario»(String sourceLocation,
 String targetLocation,
 String templateLocation,
 boolean isInstance,
 InstantiationStrategy iStrategy)
 throws ProductAssetCreationException;

/*
if (not isInstance)
 CMS/create product-specific asset
else{
 CL/ask user for core assets as origins_
 CL/_of the instance
 if not all core assets released
 exit
 if (only compilation) {
 if (connector to variability management available)
 VM/perform core asset instantiation
 CMS/compile core assets
 CMS/store binaries in targetLocation
 CMS/mark binaries as instances
 else{
«IF hasSharing(this)-»
 CMS/share core assets in targetLocation
 CMS/mark share as instantiation share
«ELSE-»
«IF hasBranches(this) || hasStreams(this)-»
 CMS/create branches/streams off the core assets_
 CMS_in targetLocation
 CMS/copy core asset contents in targetLocation_
 CMS/_according to instantiationStrategy
 CMS/mark branches or streams accordingly
«ENDIF-»
«ENDIF-»
 if (connector to variability management available){
 VM/perform core asset instantiation
 VM/obtain resulting instance and signed contract
 «IF hasIntentionalVersioning(this)-»
 CMS/store signed contract as predicate
 «ELSE-»
 CMS/store signed contract as attribute
 «ENDIF-»
 CMS/store resulting instance in targetLocation
 CMS/mark change
 }
«EXPAND StateManagement("reused")-»
 }
}
*/
«EXPAND File("ProductAssetCreationException")-»
«ENDEDEFINE»

```

```

«DEFINE removeProductAsset(String givenNameOfScenario) FOR ProductLine->
void «givenNameOfScenario»(String caID) throws ProductAssetDeletionException
/*
 CMS/create product asset object based on ID
 if (not detachOnly) {
 CMS/remove product asset marks
 if (product asset is an instance)
 CMS/remove instantiation marks
 }

 if (detachOnly && product asset is an instance)
 CMS/remove instantiation marks

 if (product asset is an instance)
 for each core asset of instance {
 check if core asset has other instances
 if not
 «EXPAND StateManagement("not reused")->
 }
*/
«EXPAND File("ProductAssetDeletionException")->
«ENDDDEFINE»

```

```

«DEFINE modifyProductAsset(String givenNameOfScenario) FOR ProductLine->
void «givenNameOfScenario»(String caID) throws ProductAssetModificationException
/*
 CMS/retrieve product asset object based on paID
 CL/ask user for new name
 if (product asset is instance) {
 CL/ask user for new core assets
 if (connector to Variability Management available) {
 CL/ask user whether to re-instantiate
 VM/re-instantiate upon confirmation
 CL/update product asset
 }
 }
 CL/ask user for new configuration item if necessary
 CMS/submit changes to change management system
*/
«EXPAND File("ProductAssetModificationException")->
«ENDDDEFINE»

```

```

«DEFINE rebaseProductAsset(String givenNameOfScenario) FOR ProductLine->
void «givenNameOfScenario»(String caID) throws ProductAssetRebaseException
/*
 CMS/create product asset object based on paID
 if (product asset is not an instance)
 exit
 /** a posteriori integration
 CL/obtain user confirmation
 CMS/mark last change on instance as integration
 /** session-based integration
 CMS/locate core assets of instance
 for each core asset{
 CMS/merge last change into instance
 CMS/mark merge as rebase merge
 }
 «EXPAND StateManagement("rebased")->
*/
«EXPAND File("ProductAssetRebaseException")->
«ENDDDEFINE»

```

## Appendix B Usability Evaluation

### B.1 Pre-briefing document

#### Product Line Exercise February 17<sup>th</sup> 2012 “Version Management”

##### Pre-Briefing document

Thank you for participating in the Product Line exercise on “Version Management”.

This document introduces you to the method of the “**Cognitive Walkthrough**” that we will use during the exercise.

##### **The Cognitive Walkthrough method**

The Cognitive Walkthrough (often abbreviated as CW) is a Usability Evaluation method.

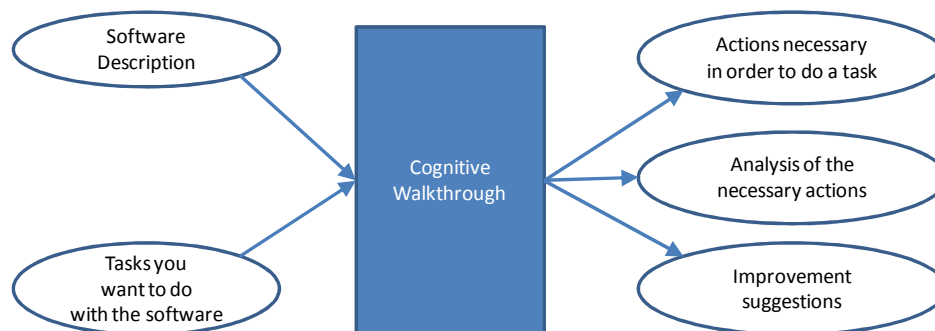
We use this method in order to understand how easy it will be to use a software product.

The CW method does not require running software. It can be also applied to interface specifications, mock-ups, screenshots or other descriptions of the software that we want to analyze

When you use a piece of software you expect it to support a set of tasks. For example, an e-mail program should help you send e-mails. Hence, when you apply the CW method you concentrate on typical tasks that you want to do with the software. For each task you look into the software description and try to understand necessary actions that you have to do. For example, in order to send an e-mail you first have to open the e-mail editor, then you have to type the message, then you have to select a recipient and so forth. **Goal of the CW method is to analyze, how easy it will be to identify and execute all the actions necessary to accomplish a task.**

In other words, when you use the CW method you imagine that you are the user of the software that you analyze. During the analysis you walk through typical tasks that you would do as user. For each task you ask yourself the question “will it be easy for me to do this task”. You answer this question by breaking down the task into smaller actions that you have to do with the software.

The following picture depicts inputs and outputs of the Cognitive Walkthrough method. In the next page you will find additional clarifications on these inputs and outputs.



| Input / Output                                      | Clarification                                                                                                                                                                                                                                                                                                                                                                                |
|-----------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Software Description (Input)                        | This is usually a document that describes what the software does. It can be an interface specification of actions available in the software.<br><br><i>The software description will be given to you at the beginning of the exercise.</i>                                                                                                                                                   |
| Tasks that you want to do with the software (Input) | This is a list of tasks that you require from the software. Some of these tasks may directly match actions in the Software Description. For some others tasks there may be no direct match. In this case you will have to identify the sequence of actions that is necessary to accomplish the tasks.<br><br><i>The list of tasks will be given to you at the beginning of the exercise.</i> |
| Actions necessary in order to do a task (Output)    | Based on the Software Description and the Tasks you identify and write down the sequence of actions necessary.                                                                                                                                                                                                                                                                               |
| Analysis of the necessary actions (Output)          | For each action that you identify you answer a set of questions, which aim at evaluating the action. One question for example will be "When you apply this action do you see any progress in the accomplishment of your task?"<br><br><i>The questions to answer will be given to you at the beginning of the exercise.</i>                                                                  |
| Improvement suggestions (Output)                    | If you answer a question with NO you are able to provide improvement suggestions.<br><br><i>A template that will be given to you at the beginning of the exercise will facilitate providing suggestions</i>                                                                                                                                                                                  |

### Additional Reading including examples

Cathleen Wharton, John Rieman, Clayton Lewis, and Peter Polson

*The cognitive walkthrough method: A practitioner's guide.*

Technical report, Institute of Cognitive Science, University of Colorado, Boulder

Downloadable under <http://ics.colorado.edu/techpubs/pdf/93-07.pdf>

## B.2 Usability Evaluation Form

### Usability Evaluation Form

Your name: .....

Your group: .....

|                                                                                                                                                                                                                                                                                                                                                                                                                                                |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Task Number :</b> .....                                                                                                                                                                                                                                                                                                                                                                                                                     |
| <b>Action Number:</b> .....<br><b>Description/Rationale:</b><br>.....<br>.....<br>.....<br>Multiple Execution of this action <input type="checkbox"/>                                                                                                                                                                                                                                                                                          |
| <b>Was it easy to understand, that you had to do this action?</b>                                                                                                                                                                                                                                                                                                                                                                              |
| YES <input type="checkbox"/>                                                                                                                                                                                                                                                                                                                                                                                                                   |
| NO <input type="checkbox"/>                                                                                                                                                                                                                                                                                                                                                                                                                    |
| <u>If NO</u> please provide improvement suggestions to the tool developers:<br>• Eliminate the requirement to do this action manually, the system should it <input type="checkbox"/><br>• Provide a message that tells me that I have to do this action <input type="checkbox"/><br>• Change the task, so that I can better understand what actions to do <input type="checkbox"/><br>Please provide any additional suggestions here:<br>..... |
| <b>Was it easy to associate the correct action with the effect you are trying to achieve?</b>                                                                                                                                                                                                                                                                                                                                                  |
| YES <input type="checkbox"/>                                                                                                                                                                                                                                                                                                                                                                                                                   |
| NO <input type="checkbox"/>                                                                                                                                                                                                                                                                                                                                                                                                                    |
| <u>If NO</u> please provide improvement suggestions to the tool developers:<br>• Improve the interface description <input type="checkbox"/><br>• Use a different name for the action <input type="checkbox"/><br>Please provide any additional suggestions here:<br>.....                                                                                                                                                                      |
| <b>Will you see that progress is being made toward solution of your task?</b>                                                                                                                                                                                                                                                                                                                                                                  |
| YES <input type="checkbox"/>                                                                                                                                                                                                                                                                                                                                                                                                                   |
| NO <input type="checkbox"/>                                                                                                                                                                                                                                                                                                                                                                                                                    |
| <u>If NO</u> please provide improvement suggestions to the tool developers:<br>Provide better feedback <input type="checkbox"/><br>Please provide any additional suggestions here:<br>.....                                                                                                                                                                                                                                                    |



## B.3 Experience questionnaire

**Product Line Exercise February 17<sup>th</sup> 2012**  
**“Version Management”**
Experience Questionnaire

**Thank you for participating in the Product Line exercise on “Version Management”.**  
**Please take a moment to fill out this questionnaire in advance.**  
**It will help us organize the exercise better.**

Your name:

## EXPERIENCE WITH PRODUCT LINE ENGINEERING

1. How would you describe your experience with product lines?

None

☐

Little

☐

Average

☐

Substantial

☐

Professional

☐

Please provide clarifications (e.g. where do you have this experience from):

2. How well have you understood the challenges of version management in a product line context? Multiple answers are possible.

I understood nothing

☐

I know it is difficult to do version management in product lines,  
but don't ask me why

☐

I understood the challenges and I can explain the reasons

☐

I have already experienced these challenges in my work

☐

I understood the challenges and also the possible solutions

☐

I understood there are solutions, but I do not know how to apply

☐

I understood there are solutions and I can apply them

☐

Any comments?

## EXPERIENCE WITH VERSION MANAGEMENT

1. How would you describe your experience version management?

None

☐

Little

☐

Average

☐

Substantial

☐

Professional

☐

Please provide clarifications (e.g. where do you have this experience from):

2. Which Version Management tools have you already used?

Subversion

☐

CVS

☐

Mercurial

☐

Git

☐

Other

☐

Any comments (e.g. list other tools you have used):

3. To which extend have you already used **branching** and **merging** functionality?

Never

☐

Almost never

☐

Occasionally

☐

Often

☐

Every day

☐

Please provide clarifications (e.g. describe how you used branching and merging):

## B.4 Task specifications

### Product Line Engineering Class

### Group CL

#### Version Management Exercise

**Goal of the exercise: Evaluate the usability of version management tools for Product Lines**

Welcome to the Product Line Engineering exercise. Subject of today's exercise is Version Management. As you learned in the lecture, Version Management with Product Lines is much more complex than with Single Systems.

There are different tools to Version Management in Product Lines. Goal of this exercise is to evaluate the usability of such tools.

Imagine that you are developing a product line and that you have to do version management. In particular you are interested in performing the following tasks on items available in your Version Management repository:

| Version Management Tasks                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ol style="list-style-type: none"><li>1. Find items marked as <b>Core Assets</b></li><li>2. For a given <b>Core Asset</b> (e.g. <code>MyLibrary.java</code>) find where it is being reused in products. In other words, find items marked as instances of the <b>Core Asset</b>.</li><li>3. For a given <b>Instance</b> (e.g. <code>MyLibrarySpecialized.java</code>) find from which <b>Core Asset</b> it comes from.</li><li>4. Imagine a core asset has been changed. Propagate the changes to its instances</li><li>5. Imagine an instance has been changed. Propagate the changes to the core assets, from which the instance comes from</li><li>6. Find assets of a product line member, which are not instances of core assets (in other words, find product-specific assets)</li></ol> |

Now, for each of the above tasks, imagine that you have a tool that can help you: **CL**.

You will find subsequently a description of the tool interface. The interface describes **actions** that you can do with the tool and the corresponding **effects** that you achieve.

For each of the above tasks please do the usability evaluation described on the next page

| Usability Evaluation                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        | Group CL |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------|
| <ul style="list-style-type: none"><li>– Study the CL tool interface and list the actions necessary in order to perform the tasks<ul style="list-style-type: none"><li>▪ Please describe how you want to use the actions and what you expect to achieve</li></ul></li><li>– For each action that you have listed, please answer following questions using the attached form:<ul style="list-style-type: none"><li>a. You selected this action because you wanted to achieve an effect. How easy was it for you to identify and select this action?<br/>(In other words: was it easy to understand, that you had to do this action?)</li><li>b. Was it easy to associate the correct action with the effect you are trying to achieve?<br/>(In other words: is the interface description clear to you?)</li><li>c. After performing the action in the real tool, will you see that progress is being made toward solution of your task?</li></ul></li><li>– If you answer a question with NO, please provide improvement suggestions as shown in the attached form.</li></ul> |          |

## Product Line Engineering Class

Group SVN\_FE

### Version Management Exercise

**Goal of the exercise: Evaluate the usability of version management tools for Product Lines**

Welcome to the Product Line Engineering exercise. Subject of today's exercise is Version Management. As you learned in the lecture, Version Management with Product Lines is much more complex than with Single Systems.

There are different tools to Version Management in Product Lines. Goal of this exercise is to evaluate the usability of such tools.

Imagine that you are developing a product line and that you have to do version management. In particular you are interested in performing the following tasks on items available in your Version Management repository:

| Version Management Tasks                                                                                                                                                                                                                                                                                                                           |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ol style="list-style-type: none"><li>1. Find items marked as Core Assets</li><li>2. For a given Core Asset (e.g. MyLibrary.java) find where it is being reused in products. In other words, find items marked as instances of the Core Asset.</li><li>3. Imagine a core asset has been changed. Propagate the changes to its instances.</li></ol> |

Now, for each of the above tasks, imagine that you have a tool that can help you: **SVN**. You will find subsequently a description of the tool interface. The interface describes **actions** that you can do with the tool and the corresponding **effects** that you achieve.

For each of the above tasks please do the usability evaluation described on the next page

| Usability Evaluation                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              | Group SVN_FE |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------|
| <p>– Study the SVN tool interface and list the actions necessary in order to perform the tasks with SVN.<br/>Please describe how you want to use each action and what you expect to achieve.<br/>Consider the following hints</p> <ul style="list-style-type: none"> <li>▪ An item is marked as Core Asset by placing it in a designated location named “CoreAssets” in the repository</li> <li>▪ You create an instance of a core asset by creating a branch (copy) of the core asset. You then place the copy in a designated location in the repository. Each product line member has such a designated location. For example if you have 3 products in your product line then there are three locations that are designated for instances “Product A Instances”, “Product B Instances” and “Product C Instances”</li> </ul> <p>– You might need to perform the same action several times. If this is the case please check the multiple execution box.</p> <p>– For each action that you have listed, please answer following questions using the attached form:</p> <ol style="list-style-type: none"> <li>a. You selected this action because you wanted to achieve an effect. How easy was it for you to identify and select this action?<br/>(In other words: was it easy to understand, that you had to do this action?)</li> <li>b. Was it easy to associate the correct action with the effect you are trying to achieve?<br/>(In other words: is the interface description clear to you?)</li> <li>c. After performing the action, will you see that progress is being made toward solution of your task?</li> </ol> <p>– If you answer a question with NO, please provide improvement suggestions as shown in the attached form.</p> |              |

**Product Line Engineering Class****Group SVN\_AE****Version Management Exercise**

**Goal of the exercise: Evaluate the usability of version management tools for Product Lines**

Welcome to the Product Line Engineering exercise. Subject of today's exercise is Version Management. As you learned in the lecture, Version Management with Product Lines is much more complex than with Single Systems.

There are different tools to Version Management in Product Lines. Goal of this exercise is to evaluate the usability of such tools.

Imagine that you are developing a product line and that you have to do version management. In particular you are interested in performing the following tasks on items available in your Version Management repository:

| Version Management Tasks                                                                                                                                                                                                                                                                                                   |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ol style="list-style-type: none"><li>1. Find items marked as Instances</li><li>2. For a given Instance (e.g. MyLibrarySpecialized.java) find from which Core Asset it comes from.</li><li>3. Imagine an instance has been changed. Propagate the changes to the Core Assets, from which the instance comes from</li></ol> |

Now, for each of the above tasks, imagine that you have a tool that can help you: **SVN**. You will find subsequently a description of the tool interface. The interface describes **actions** that you can do with the tool and the corresponding **effects** that you achieve.

For each of the above tasks please do the usability evaluation described on the next page

| Usability Evaluation                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              | Group SVN_FE |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------|
| <p>– Study the SVN tool interface and list the actions necessary in order to perform the tasks with SVN.<br/>Please describe how you want to use each action and what you expect to achieve.<br/>Consider the following hints</p> <ul style="list-style-type: none"> <li>▪ An item is marked as Core Asset by placing it in a designated location named “CoreAssets” in the repository</li> <li>▪ You create an instance of a core asset by creating a branch (copy) of the core asset. You then place the copy in a designated location in the repository. Each product line member has such a designated location. For example if you have 3 products in your product line then there are three locations that are designated for instances “Product A Instances”, “Product B Instances” and “Product C Instances”</li> </ul> <p>– You might need to perform the same action several times. If this is the case please check the multiple execution box.</p> <p>– For each action that you have listed, please answer following questions using the attached form:</p> <ul style="list-style-type: none"> <li>a. You selected this action because you wanted to achieve an effect. How easy was it for you to identify and select this action?<br/>(In other words: was it easy to understand, that you had to do this action?)</li> <li>b. Was it easy to associate the correct action with the effect you are trying to achieve?<br/>(In other words: is the interface description clear to you?)</li> <li>c. After performing the action, will you see that progress is being made toward solution of your task?</li> </ul> <p>– If you answer a question with NO, please provide improvement suggestions as shown in the attached form.</p> |              |



## B.5 Interface descriptions

**CL - Interface Description****showCoreAssets**

```
public CoreAsset[] showCoreAssets(String nameFilter)
 throws CoreAssetRetrievalException;
```

**Effect:**

This operation lists items, which are marked as core assets in the repository

**Parameters:**

`nameFilter` – This is a filter. It enables listing only the core assets whose names start with the contents of `nameFilter`. Leave empty to retrieve all core assets.

**Returns:**

An array of `CoreAsset` objects. Each object contains the name of a core asset that matches the `nameFilter`. Each object also contains additional attributes, such as the available instances of the core asset.

**Throws:**

`CoreAssetRetrievalException` – if the operation could not be performed

**showCoreAssetInstances**

```
Instance[] showCoreAssetInstances(
 CoreAsset[] coreAssets,
 String nameFilter
)
 throws InstanceRetrievalException;
```

**Effect:**

This operation lists items, which are marked as instances of core assets

**Parameters:**

`coreAssets` – the set of core assets, for which we want to find the instances  
`nameFilter` – This is a filter. It enables listing only the instances whose names start with the contents of `nameFilter`. Leave empty to retrieve all instances.

**Returns:**

An array of `Instance` objects. Each object contains the name of an instance item that matches the `nameFilter`. Each object also contains additional fields such as the core assets, from which the instance comes from.

**Throws:**

`InstanceRetrievalException` – if the operation could not be performed

**showProductAssets**

```
ProductAsset[] showProductAssets(
 String productName,
 Boolean showOnlyProductSpecifics
)
 throws InstanceRetrievalException;
```

**Effect:**  
This operation lists items, which are marked as parts of product line member (i.e. a product)

**Parameters:**  
 productName - the name of the product to query for assets  
 showOnlyProductSpecifics - a Boolean value which determines whether to list product-specific items (i.e. not instances of core assets) only .

**Returns:**  
An array of ProductAsset objects.

**Throws:**  
InstanceRetrievalException - if the operation could not be performed

**integrateCoreAsset**

```
State integrateCoreAsset(String coreAssetName)
 throws CoreAssetIntegrationException
```

**Effect:**  
This operation propagates pending changes of a core asset to its instances,

**Parameters:**  
coreAssetName - the name of the core asset, of which the changes should be propagated

**Returns:**  
the operation returns the state of the core asset, possible states are Integrated or Not\_Integrated

**Throws:**  
CoreAssetIntegrationException - if the operation could not be performed

**rebaseInstance**

```
State rebaseInstance(String instanceName)
 throws RebaseException
```

**Effect:**  
This operation propagates pending changes of an instance to the core assets the instance comes from,

**Parameters:**  
instanceName - the name of the instance, of which the changes should be propagated

**Returns:**  
the operation returns the state of the instance, possible states are Rebased or Not\_Rebased

**Throws:**  
CoreAssetIntegrationException - if the operation could not be performed

## SVN - Interface Description

### checkout

|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>public long <b>checkout</b>(URL url,                       File destinationPath,                       Revision revision,                       Depth depth)     throws SVNException</pre>                                                                                                                                                                                                                                                                                                                                                                                                           |
| <p><b>Effect:</b></p> <p>This operation checks-out a directory from the Version Management Repository to a local directory path.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <p><b>Parameters:</b></p> <p>url – the repository location of the directory to be checked-out<br/> destinationPath – the local path where the directory will be stored<br/> revision – the revision of the item to be checked-out. Leave empty to retrieve the last revision<br/> depth – determines how many items to check-out from the directory in the input url:<br/> EMPTY<br/> Just the named directory, no entries in that directory.<br/> FILES<br/> Named directory and its file children, but not subdirectories.<br/> INFINITY<br/> Named directory and all descendants (full recursion).</p> |
| <p><b>Returns:</b></p> <p>the revision number of the Working Copy</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| <p><b>Throws:</b></p> <p>Exception – url refers to a file, not a directory; destinationPath already exists but it is a file, not a directory; destinationPath already exists and is a versioned directory but has a different URL (repository location against which the directory is controlled)</p>                                                                                                                                                                                                                                                                                                     |

### update

|                                                                                                                                                                                                                                   |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>public long <b>update</b>(File path,                    Revision revision,                    Depth depth)     throws SVNException</pre>                                                                                     |
| <p><b>Effect:</b></p> <p>This operation updates a local copy of an item with changes from the repository.</p>                                                                                                                     |
| <p><b>Parameters:</b></p> <p>path – the file system path of the local copy<br/> revision – the revision to update to. Leave empty to update to the last revision<br/> depth – tree depth to update (depth semantics as above)</p> |
| <p><b>Returns:</b></p> <p>the revision number to which the item was updated to</p>                                                                                                                                                |
| <p><b>Throws:</b></p> <p>Exception – if the update operation could not be performed</p>                                                                                                                                           |

**commit**

```
public CommitInfo commit(File[] paths,,
 String commitMessage,
 Properties revisionProperties,
 Depth depth)
 throws Exception
```

**Effect:**

This operation commits (i.e. loads) changes from local files to the repository.

**Parameters:**

`paths` – the paths of the local files to commit the changes from  
`commitMessage` – a log message to use for the commit operation  
`revisionProperties` – the properties to set on the committed files. Each property is a key-value pair

**Returns:**

Commit information containing the revision number after commit, the revision author and a time stamp.

**Throws:**

Exception – if the commit operation could not be performed

**dir**

```
public String dir(URL url)
 throws Exception
```

**Effect:**

This operation lists the items available at a given repository location

**Parameters:**

`url` – the repository location to list the items from

**Returns:**

A textual list of items (similar to the `dir` or `ls` commands in Windows and Linux)

**Throws:**

Exception – if the directory information could not be obtained

**log**

```
public LogEntry[] log(File path,
 Revision startRevision,
 Revision endRevision)
 throws Exception
```

**OR**

```
public LogEntry[] log(URL path,
 Revision startRevision,
 Revision stopRevision)
 throws Exception
```

**Effect:**

This operation obtains the list of operations (e.g. commits, branches) that have been performed on an item

**Parameters:**

path – the path in which the item resides. This can be local or a remote repository path, in the latter case a URL is used.  
startRevision – revision number to start from. Leave empty if you want to get a list of operations starting from the first revision.  
endRevision – revision number to stop at, leave empty if you want to stop at the last revision

**Returns:**

A set of log entries. Each entry represents operations performed on an item. Each entry contains following information:

- Author
- Date
- Log Message
- Files affected

**Throws:**

Exception – if the log information could not be obtained

**merge**

```
public void merge(URL url1,
 Revision r1,
 URL url2,
 Revision r2,
 File destination,
 Depth depth)
 throws Exception
```

**Effect:**

This operation merges differences between two items in the repositories into a local copy item

**Parameters:**

url1 – the URL of the first item  
r1 – the revision of the first item to consider during comparison  
url2 – the URL of the second item  
r2 – the revision of the second item to consider during comparison  
depth – the depth to use during comparison  
destination – the path to the local file that will receive the changes

---

|                                                                |
|----------------------------------------------------------------|
| <u>Returns:</u><br>no return value                             |
| <u>Throws:</u><br>Exception - if the operation did not succeed |

## B.6 Data entry application

[Home](#) [Action List](#)

---

Create Action

---

Usability Evaluation Form

---

Subject \* CL:Participant ▾

Task \* Find items marked as Core Assets ▾

---

Description

---

Multiple Execution ☐

---

Was it easy to understand that you had to do this action ☐

Eliminate the requirement to do this action manually, the system should it ☐

Provide a message that tells me that I have to do this action ☐

Change the task, so that I can better understand what actions to do ☐

Additional Suggestions

---

Was it easy to associate the correct action with the effect you are trying to achieve? ☐

Improve the interface description ☐

Use a different name for the action ☐

Additional Suggestions

---

Will you see that progress is being made toward solution of your task? ☐

Provide better Feedback ☐

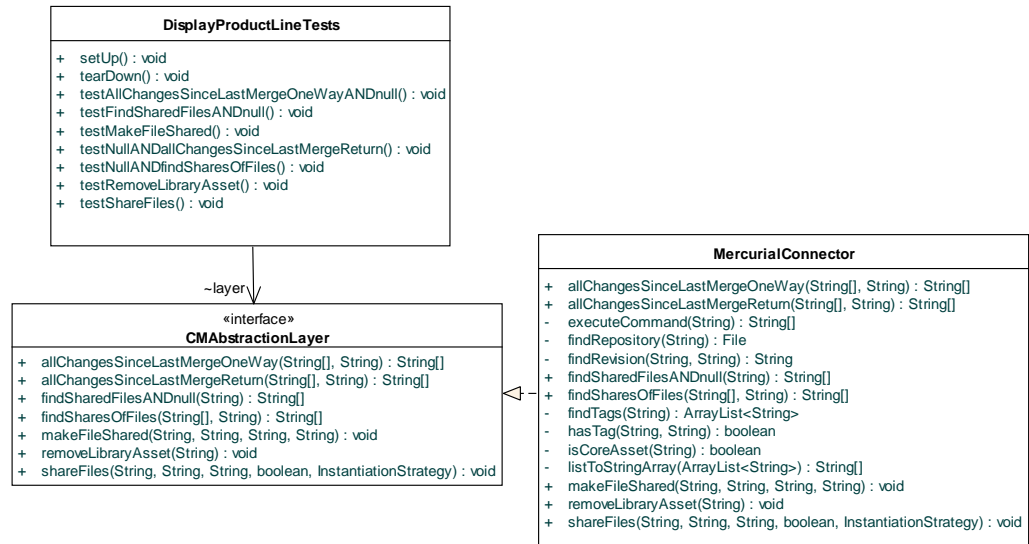
Additional Suggestions

---

[Create](#)

## Appendix C Case Study Material

### C.1 Implementation of a Mercurial connector





## C.2 Pre-briefing document

## Configuration Management Scenarios

### Pre-Briefing Questionnaire

---

**Purpose of the questionnaire:** This questionnaire aims at capturing current problems and challenges with configuration management and to estimate the benefits of a solution that automates particular configuration management scenarios.

---

#### 1 Creation of shared items

##### 1.1 How often do you have to create reusable items (i.e. files or directories) that are supposed to be shared across various different projects?

| Never                    | Very rarely              | Not often                | Often                    | Very often               | All the time             |
|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|
| <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |

Please provide any additional remarks here:

##### 1.2 How do you create shared items with the help of your version management system (i.e. StarTeam or Mercurial)?

- ☐ I put all shared items in a well-known location that contains only shared items
- ☐ I put the shared items in various locations; however there is central file that lists all shared files
- ☐ I use a naming convention for shared items
- ☐ I use a particular commit comment
- ☐ I define a user-defined property (this is possible with my version management system) to mark the item as shared
- ☐ I use the tagging mechanism of my version management system to mark the item as shared

☐ I use another mechanism that is:

Please provide any additional remarks here:

**1.3 Do you think that everybody in the organization uses the same mechanism to create shared items?**

☐ Yes they all use this mechanism:

☐ No

**1.4 Do you think that the creation of shared items is a complex operation?**

| Piece of cake            | Easy                     | Not complex              | Complex                  | Very complex             | Hardly possible          |
|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|
| <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |

Please provide any additional remarks here:

**1.5 Do you think that the creation of shared items is an error-prone operation?**

| Never heard of an error  | Errors are very rare     | Errors occur from time to time | Errors occur often       | Errors occur very often  | Errors occur all the time |
|--------------------------|--------------------------|--------------------------------|--------------------------|--------------------------|---------------------------|
| <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/>       | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/>  |

Please provide any additional remarks here:

**1.6 Do you think that a script that automates the creation of shared items would be helpful?**

☐ Yes that would save us at least 25% of effort and errors

☐ No, this is not necessary because

## 2 Finding shared items

### 2.1 How often do you have to look for shared items (e.g. a library, a reusable component, a document template) that you can use in your project?

| Never                    | Very rarely              | Not often                | Often                    | Very often               | All the time             |
|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|
| <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |

Please provide any additional remarks here:

### 2.2 The first step is to identify the item you want to reuse. How do you identify reusable items?

- ☐ I look into well-defined repository locations, which hold shared items
- ☐ I look into a special file that lists shared items
- ☐ I look for shared items with a particular naming convention
- ☐ I analyze the commit comments
- ☐ I look for shared items with particular user-defined properties
- ☐ I look for shared items with particular tags
- ☐ I use another mechanism that is:

### 2.3 The next step is to copy the shared file to your project. How do you copy?

- ☐ I copy the shared item into my project. The version management system does not notice the copy operation (i.e. later I cannot see where the shared item came from)
- ☐ I copy the shared item into my project. The version management system notices the copy operation and tracks where the item came from.

Please provide any additional remarks here:

### 2.4 Which of the following apply to the way you copy shared items to your project?

- ☐ I put the copied shared items in a well-known location that contains only copied shared items

- ☐ I put the shared items in various locations; however there is central file that lists all copies
- ☐ I use a naming convention for copies of shared items
- ☐ I use a particular commit comment to indicate that I copied a shared item
- ☐ I define a user-defined property (this is possible with my version management system) to mark the copy of a shared item
- ☐ I use the tagging mechanism of my version management system to mark the copy of a shared item
- ☐ I use another mechanism that is:
- Please provide any additional remarks here:

## 2.5 Do you think that everybody in the organization employs the same mechanism to reuse shared items?

- ☐ Yes they all use this mechanism:
- ☐ No

## 2.6 Do you think that reusing a shared item is a complex operation?

| Piece of cake            | Easy                     | Not complex              | Complex                  | Very complex             | Hardly possible          |
|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|
| <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |

Please provide any additional remarks here:

## 2.7 Do you think that reusing a shared item is an error-prone operation?

| Never heard of an error  | Errors are very rare     | Errors occur from time to time | Errors occur often       | Errors occur very often  | Errors occur all the time |
|--------------------------|--------------------------|--------------------------------|--------------------------|--------------------------|---------------------------|
| <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/>       | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/>  |

Please provide any additional remarks here:

### 3.4 Do you think that finding out about reuse is a complex operation?

| Piece of cake            | Easy                     | Not complex              | Complex                  | Very complex             | Hardly possible          |
|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|
| <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |

Please provide any additional remarks here:

### 3.5 Do you think that finding out about reuse is an error-prone operation?

| Never heard of an error  | Errors are very rare     | Errors occur from time to time | Errors occur often       | Errors occur very often  | Errors occur all the time |
|--------------------------|--------------------------|--------------------------------|--------------------------|--------------------------|---------------------------|
| <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/>       | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/>  |

Please provide any additional remarks here:

### 3.6 Do you think that a script that automates reuse of shared items would be helpful?

☐ Yes that would save us at least 25% of effort and errors

☐ No, this is not necessary because

## 4 Synchronizing changes

### 4.1 How often do you have to synchronize changes between shared items (e.g. library project) and applications (e.g. projects that use libraries)

| Never                    | Very rarely              | Not often                | Often                    | Very often               | All the time             |
|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|
| <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |

Please provide any additional remarks here:

#### 4.2 What type of changes occur during development

|                                                                                  |                          |
|----------------------------------------------------------------------------------|--------------------------|
| Shared items (e.g. libraries) change                                             | <input type="checkbox"/> |
| Specific items (e.g. project-specific files) change                              | <input type="checkbox"/> |
| Shared items that are used within projects change in the context of the projects | <input type="checkbox"/> |

#### 4.3 In which direction do you synchronize changes?

|                                                                                       |                          |
|---------------------------------------------------------------------------------------|--------------------------|
| From shared items (e.g. libraries) to applications (e.g. projects that use libraries) | <input type="checkbox"/> |
| From applications (e.g. projects that use libraries) to shared items (e.g. libraries) | <input type="checkbox"/> |

Please provide any additional remarks here:

#### 4.4 How do you synchronize changes?

- ☐ Everyone has a copy of the complete repository (includes all shared items and all applications) and everyone updates his copy frequently
- ☐ I have a repository of shared items and I synchronize with other repositories that use my libraries
- ☐ I have a repository for my application I synchronize with other repositories that have shared items

Please provide any additional remarks here:

#### 4.5 Do you think that everybody in the organization employs the same mechanism to synchronize change?

- ☐ Yes they all use this mechanism:
- ☐ No

#### 4.6 Do you think that synchronizing changes is a complex operation?

| Piece of cake            | Easy                     | Not complex              | Complex                  | Very complex             | Hardly possible          |
|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|
| <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |

Please provide any additional remarks here:

#### 4.7 Do you think that synchronizing changes is an error-prone operation?

| Never heard of an error  | Errors are very rare     | Errors occur from time to time | Errors occur often       | Errors occur very often  | Errors occur all the time |
|--------------------------|--------------------------|--------------------------------|--------------------------|--------------------------|---------------------------|
| <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/>       | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/>  |

Please provide any additional remarks here:

#### 4.8 Do you think that a script that automates synchronization would be helpful?

☐ Yes that would save us at least  of effort and errors

☐ No, this is not necessary because

## Appendix D Experiment Material

### D.1 Task specifications

|                            |                                     |
|----------------------------|-------------------------------------|
| <b>Task Specifications</b> | <b>Group: CL</b><br><b>Role: AE</b> |
|----------------------------|-------------------------------------|

#### Introduction

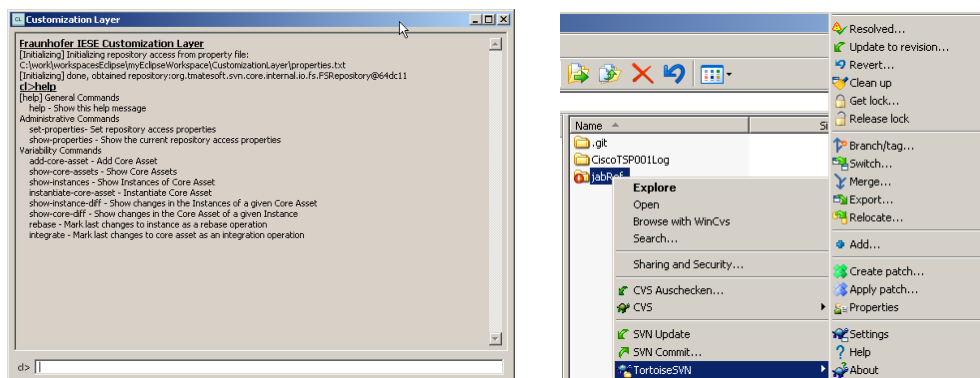
As you can see above you are in the CL group. CL stands for Customization Layer, a tool that facilitates the management of a Product Line Infrastructure. You can see a screenshot of the tool in the following picture.

The Customization Layer provides a series of commands like **add-core-asset** or **show-instances** that will help you complete your tasks.

You can use the **help** command for getting information about the other commands. By typing for example **help add-core-asset** you will get information about the function and the arguments of the **add-core-asset** command. Arguments enclosed in brackets are meant to be optional.

The Customization Layer is to be used in combination with Subversion. In fact the Customization Layer uses Subversion behind the scenes. However for some operations we have to use Subversion directly. The Subversion server for your group is available at <http://ksi/PLEvolution/CL/>

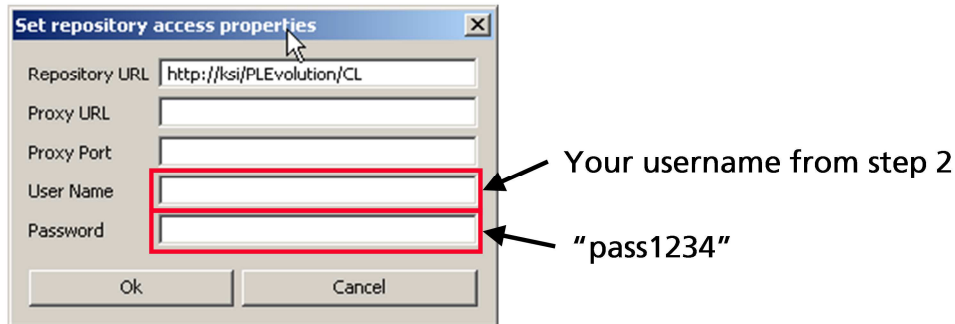
For committing and checking-out assets from the repository please use the Tortoise Subversion Client which is integrated into the Windows explorer. For all other operations please use the CL.





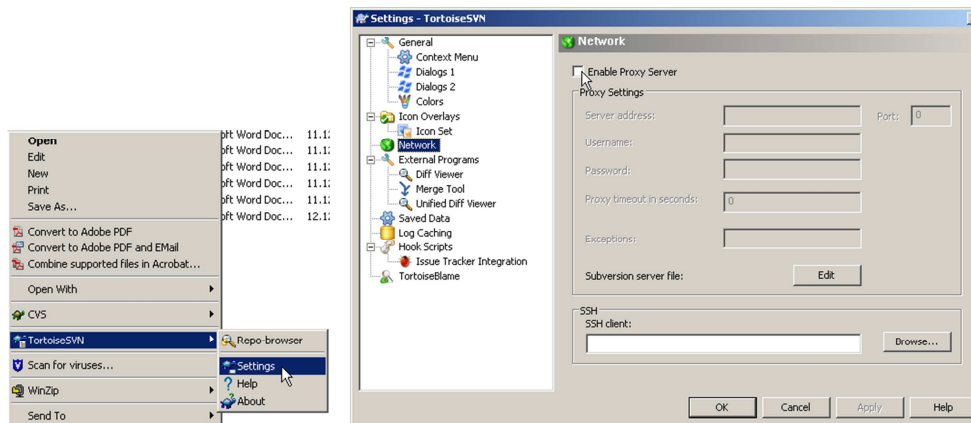
## Preparations

0. Please create a text file in your desktop for putting there the answer to the questions you will find in the tasks.
1. Open a Command Prompt (Start→Run...→Enter `cmd` and press OK)
2. Type `echo %USERNAME%`. This will display your current username (for example `stud01`).
3. Open the Customization Layer, you can find it in your desktop. Call `java -jar CL.jar`
4. Call the `set-properties` command. A dialog is displayed. Please set the properties as follows:



Please use the same user name and password with the Tortoise Subversion client in the following.

5. Ensure that Tortoise does not use a proxy server, as shown in the following screen shots:



**Task 1: Find Core Assets**

In this task we want to find existing core assets (directories and files) in the Subversion repository. Please use the CL (Customization Layer) command **show-core-assets** to get a list of all available core assets.

Please answer the following questions:

1. How many core assets are available?
  2. How many Java classes are available as core assets?
  3. How many directories are available as core assets?
- Please write down how much time you needed for this task

**Task 2: Create Instances of Core Assets**

In this task we want to create an instance of a core asset for the purpose of a product.

Please select one of the directory core assets you found in the previous task. It does not matter which one.

Imagine that you are the Application Engineer of ProductA.

Use the CL command

**instantiate-core-asset [the the directory from above] ProductA** for creating an instance of this core asset in the repository. The command will place a copy of the selected core asset in the directory <http://ksi/PLEvolution/CL/branches/ProductA>

→ Please write down how much time you needed for this task

### **Task 3: Adapt instances to the needs of a product**

During Task 2 we created an instance of a core asset, which up to now is simply a copy of that core asset.

Now we want to adapt this instance for the needs of our ProductA.

Please use the Tortoise Subversion client, which is integrated into the Windows Explorer, to check-out (load) the instance from the repository to a local directory in your hard disk (for example c:\temp).

To that end point the Tortoise Subversion Client to the directory <http://ksi/PLEvolution/CL/branches> where you added the instance in Task 2 and perform the **SVN Checkout** operation.

Perform a simple change in the instance you just checked-out. You can for example enhance a little bit the methods of the Java class contained therein. Perform the **SVN Commit** operation with Tortoise when you are finished.

→ Please write down how much time you needed for this task

### **Task 4: Find Changes in the Core Assets**

In this task we want to find out whether the core assets, we have seen during Task 1, have changed in the meanwhile.

Please use the CL command **show-core-diff** for finding out about changes.

Are there any changes in the core assets? How many?

→ Please write down how much time you needed for this task

## Task Specifications

**Group: CL**  
**Role: FE**

### Introduction

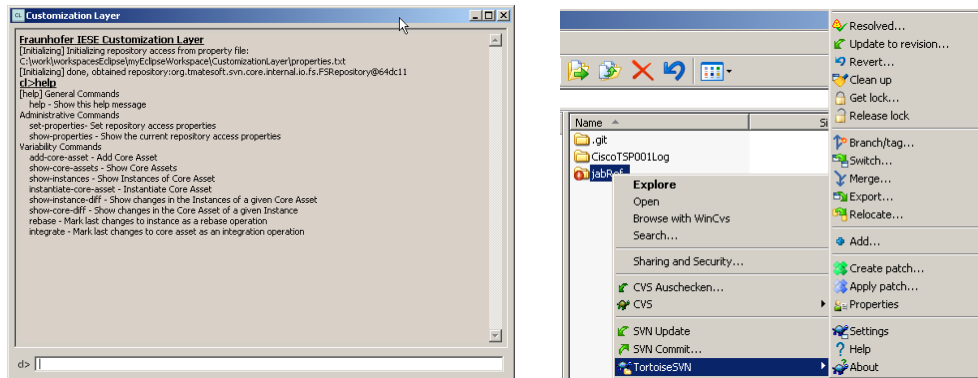
As you can see above you are in the CL group. CL stands for Customization Layer, a tool that facilitates the management of a Product Line Infrastructure. You can see a screenshot of the tool in the following picture.

The Customization Layer provides a series of commands like **add-core-asset** or **show-instances** that will help you complete your tasks.

You can use the **help** command for getting information about the other commands. By typing for example **help add-core-asset** you will get information about the function and the arguments of the **add-core-asset** command. Arguments enclosed in brackets are meant to be optional.

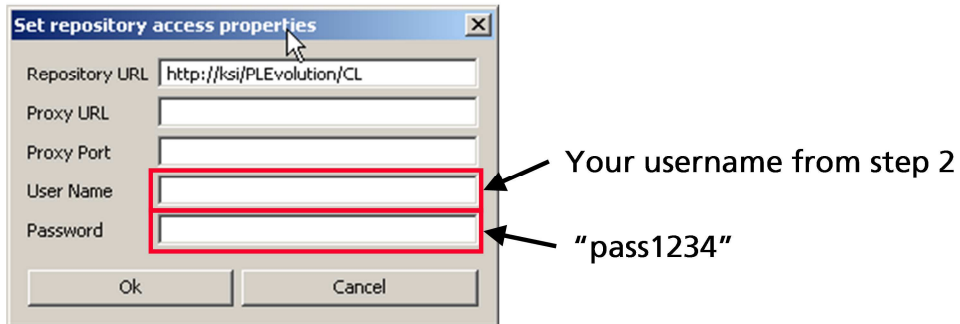
The Customization Layer is to be used in combination with Subversion. In fact the Customization Layer uses Subversion behind the scenes. However for some operations we have to use Subversion directly. The Subversion server for your group is available at <http://ksi/PLEvolution/CL/>

For committing and checking-out assets from the repository please use the Tortoise Subversion Client which is integrated into the Windows explorer. For all other operations please use the CL.



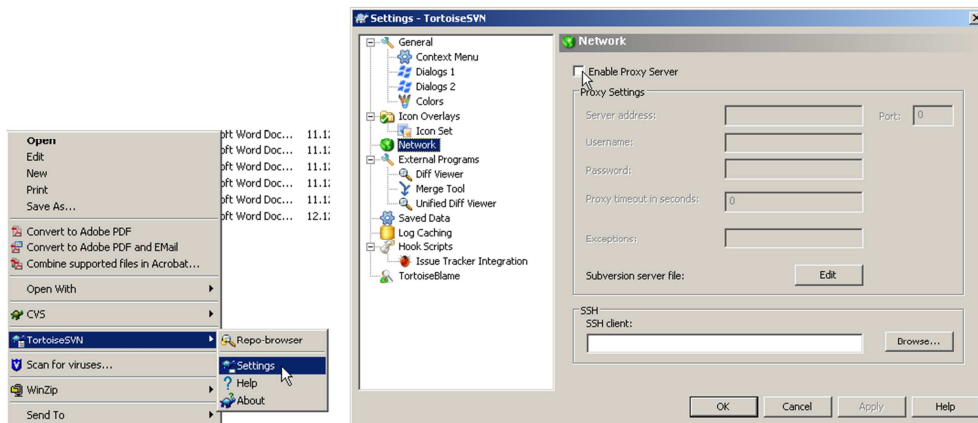
### Preparations

0. Please create a text file in your desktop for putting there the answer to the questions you will find in the tasks.
1. Open a Command Prompt (Start→Run...→Enter “cmd” and press OK)
2. Type `echo %USERNAME%`. This will display your current username (for example `stud01`).
3. Open the Customization Layer, you can find it in your desktop. Call `java -jar CL.jar`
4. Call the `set-properties` command. A dialog is displayed. Please set the properties as follows.



Please use the same user name and password with the Tortoise Subversion client in the following.

5. Ensure that Tortoise does not use a proxy server, as shown in the following screen shots:



**Task 1: Add New Core Assets**

In this task we want to add a new core asset into the Subversion repository.

You will find in your local desktop a folder named "hashtable". This folder contains a Java class, which is supposed to be used as core asset (i.e. it is reusable across the product line).

Please use the CL (Customization Layer) command **add-core-asset** to add the hashtable folder in the repository.

The CL will automatically select the directory <http://ksi/PLEvolution/CL/trunk> for storing the core asset.

→ Please write down how much time you needed for this task

**Task 2: Change Core Assets**

In this task we want to change a core asset that is already in the Subversion repository.

Please use the Tortoise Subversion client, which is integrated into the Windows Explorer, to check-out (load) the hashtable (of Task 1) from the repository to a local directory in your hard disk (for example c:\temp).

To that end point the Tortoise Subversion Client to the directory

<http://ksi/PLEvolution/CL/trunk>/hashtable where you added the hashtable in Task 1 and perform the **SVN Checkout** operation.

Now, we want to change the core asset that we just checked-out from the repository. To that end we must consider the following change request:

|                       |                                                                                                                                                                                                                                                     |  |        |                                     |        |                                     |
|-----------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--|--------|-------------------------------------|--------|-------------------------------------|
| Application Name:     | hashtable                                                                                                                                                                                                                                           |  |        |                                     |        |                                     |
| Brief Change Summary: | Provide debugging information                                                                                                                                                                                                                       |  |        |                                     |        |                                     |
| Change type           | Add                                                                                                                                                                                                                                                 |  | Change | <input checked="" type="checkbox"/> | Delete |                                     |
| Priority              | Low                                                                                                                                                                                                                                                 |  | Medium |                                     | High   | <input checked="" type="checkbox"/> |
| Detailed Change Info: | Change the methods of MyHashtable.java so that debugging information is printed out every time the methods are being called. The information should be printed to the standard output by using the <code>System.out.println(String x)</code> method |  |        |                                     |        |                                     |

Please perform the above change and use the Tortoise Subversion Client for committing your changes in the repository (use the **SVN Commit** menu with Tortoise)

→ Please write down how much time you needed for this task

### **Task 3: Find Instances**

Now we want to find out how many instances have been already created during Application Engineering. To this end please use the **show-instances** command of the Customization Layer.

Please answer the following questions:

1. Which instances have been created?
2. From which core assets have these instances been created?
3. Are there any instances of your core asset?

→ Please write down how much time you needed for this task

### **Task 4: Find Changes in the Instances**

Now we also want to find out if the Application Engineers changed the instances after they created them.

To this end please use the **show-instance-diff** command of the Customization Layer.

Are there any changes in the instances?

→ Please write down how much time you needed for this task

## Task Specifications

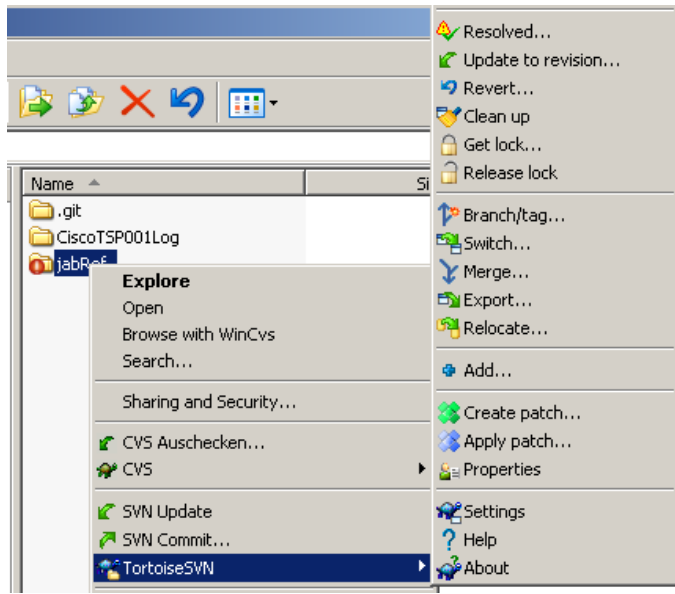
**Group: SVN**  
**Role: AE**

### Introduction

As you can see above you are in the SVN group. SVN stands for Subversion, the configuration management tool we will use for managing the Product Line Infrastructure.

The Subversion server for your group is available at <http://ksi/PLEvolution/SVN/>

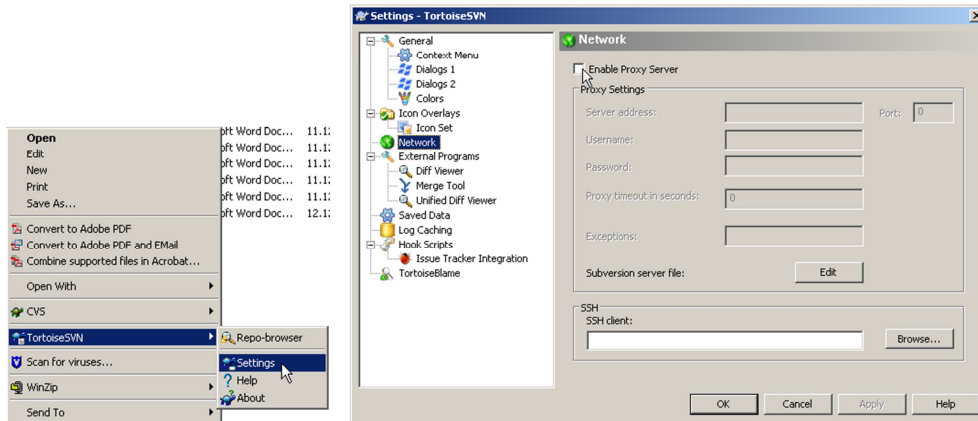
For working with Subversion we will use Tortoise Subversion Client which is integrated into the Windows explorer. Here comes a screenshot of Tortoise:





## Preparations

1. Please create a text file in your desktop for putting there the answer to the questions you will find in the tasks.
2. Open a Command Prompt (Start→Run...→Enter “cmd” and press OK)
3. Type `echo %USERNAME%`. This will display your current username (for example `stud01`).
4. Please use this user name and the password “pass1234” with the Tortoise Subversion client in the following.
5. Ensure that Tortoise does not use a proxy server, as shown in the following screen shots:



### **Task 1: Find Core Assets**

In this task we want to find existing core assets (directories and files) in the Subversion repository.

Please use the Tortoise Subversion client (you can use the **Repo-Browser** or you can do an **SVN Checkout** for loading the files from the server to your local hard disk, for example in the c:\temp directory) to get a list of all available core assets.

The core assets should be available under <http://ksi/PLEvolution/SVN/trunk> and they should be identifiable as core assets through a common convention. The convention says that when a core asset is added to the repository the commit message should be set to “adding a core asset” or to something similar. You can see the commit messages by using the **Show Log** feature of Tortoise.

Please answer the following questions:

1. How many core assets are available?
  2. How many Java classes are available as core assets?
  3. How many directories are available as core assets?
- Please write down how much time you needed for this task

### **Task 2: Create Instances of Core Assets**

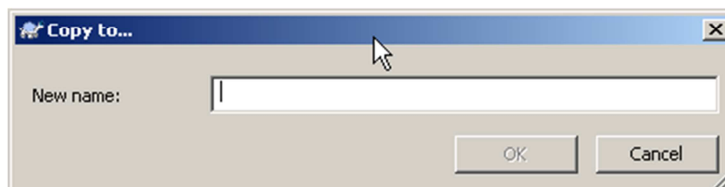
In this task we want to create an instance of a core asset for the purpose of a product.

Please select one of the directory core assets you found in the previous task. It does not matter which one.

Imagine that you are the Application Engineer of ProductA.

Use the Tortoise repository browser (**Repo-Browser**) and then the **Copy to** command for creating an instance of this core asset in the repository. This command creates a branch off the main development of the core asset. The idea is that this branch will manage the development of our instance.

For storing the branch in the right place the “New Name” text field should point to <http://ksi/PLEvolution/SVN/branches/ProductA> :



→ Please write down how much time you needed for this task

### **Task 3: Adapt instances to the needs of a product**

During Task 2 we created an instance of a core asset, which up to now is simply a copy of that core asset.

Now we want to adapt this instance for the needs of our ProductA.

Please use Tortoise check-out (load) the instance from the repository to a local directory in your hard disk (for example c:\temp).

To that end point the Tortoise Subversion Client to the directory

<http://ksi/PLEvolution/SVN/branches/ProductA> where you added the instance in Task 2 and perform the **SVN Checkout** operation.

Perform a simple change in the instance you just checked-out. You can for example enhance a little bit the methods of the Java class contained therein. Perform the **SVN Commit** operation with Tortoise when you are finished. Remember to use an appropriate commit message.

→ Please write down how much time you needed for this task

### **Task 4: Find Changes in the Core Assets**

In this task we want to find out whether the core assets, we have seen during Task 1, have changed in the meanwhile.

Please use Tortoise for finding out about changes. Combine the **Repository Browser**, **Revision Graph** and **Show Log** features of Tortoise.

Are there any changes in the core assets? How many?

→ Please write down how much time you needed for this task

## Task Specifications

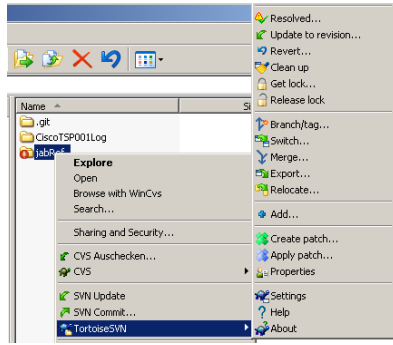
**Group: SVN**  
**Role: FE**

### Introduction

As you can see above you are in the SVN group. SVN stands for Subversion, the configuration management tool we will use for managing the Product Line Infrastructure.

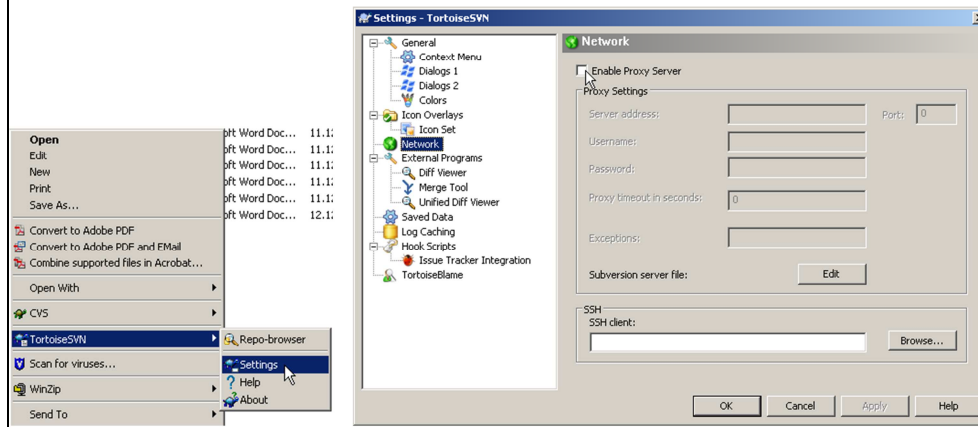
The Subversion server for your group is available at <http://ksi/PLEvolution/SVN/>

For working with Subversion we will use Tortoise Subversion Client which is integrated into the Windows explorer. Here comes a screenshot of Tortoise:



### Preparations

1. Please create a text file in your desktop for putting there the answer to the questions you will find in the tasks.
2. Open a Command Prompt (Start→Run...→Enter "cmd" and press OK)
3. Type `echo %USERNAME%`. This will display your current username (for example `stud01`).
4. Please use this user name and the password "pass1234" with the Tortoise Subversion client in the following.
5. Ensure that Tortoise does not use a proxy server, as shown in the following screen shots:



**Task 1: Add New Core Assets**

In this task we want to add a new core asset into the Subversion repository. The path <http://ksi/PLEvolution/SVN/trunk> is supposed to hold the core assets in the repository.

You will find in your local desktop a folder named “set”. This folder contains a Java class, which is supposed to be used as core asset (i.e. it is reusable across the product line).

Please use the Tortoise Subversion client (use the **Import** command), which is integrated into the Windows Explorer to add the set directory to the Subversion repository under

<http://ksi/PLEvolution/SVN/trunk/set>

→ Please write down how much time you needed for this task

**Task 2: Change Core Assets**

In this task we want to change a core asset that is already in the Subversion repository.

Please use the Tortoise Subversion to check-out (load) the set (of Task 1) from the repository to a local directory in your hard disk (for example c:\temp).

To that end point the Tortoise Subversion Client to the directory

<http://ksi/PLEvolution/CL/trunk/set> where you added the set in Task 1 and perform the **SVN Checkout** operation.

Now, we want to change the core asset that we just checked-out from the repository. To that end we must consider the following change request:

|                       |                                                                                                                                                                                                                             |  |        |          |        |          |
|-----------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--|--------|----------|--------|----------|
| Application Name:     | set                                                                                                                                                                                                                         |  |        |          |        |          |
| Brief Change Summary: | Provide debugging information                                                                                                                                                                                               |  |        |          |        |          |
| Change type           | Add                                                                                                                                                                                                                         |  | Change | <b>X</b> | Delete |          |
| Priority              | Low                                                                                                                                                                                                                         |  | Medium |          | High   | <b>X</b> |
| Detailed Change Info: | Change the methods of MySet so that debugging information is printed out every time the methods are being called. The information should be printed to the standard output by using the System.out.println(String x) method |  |        |          |        |          |

Please perform the above change and use the Tortoise Subversion Client for committing your changes in the repository (use the **SVN Commit** menu with Tortoise)

→ Please write down how much time you needed for this task

**Task 3: Find Instances**

Now we want to find out how many instances have been already created during Application Engineering.

To this end please use the Tortoise Subversion (e.g. the repository browser and the revision graphs) for finding out about instances. Instances will be normally stored as branches in the repository path <http://ksi/PLEvolution/SVN/branches>

Please answer the following questions:

1. Which instances have been created?
2. From which core assets have these instances been created?
3. Are there any instances of your core asset?

→ Please write down how much time you needed for this task

**Task 4: Find Changes in the Instances**

Now we also want to find out if the Application Engineers changed the instances after they created them.

To this end please use again the Tortoise Subversion.

Are there any changes in the instances?

→ Please write down how much time you needed for this task

## D.2 Feedback form

## Feedback Form

Thank you for participating in the Software Product Line practical exercise.

Please take a moment to fill out the following feedback form.

**Your user name (stud01, stud02 etc):**

**Your group (CL or SVN):**

**Your role (FE or AE):**

**Your name (optional):**

**Your e-mail (optional):**

### EXPERIENCE

1. How would you describe your experience with Configuration Management?

None

Little

Average

Substantial

Professional

☐☐☐☐☐

Any Comments?

2. How would you describe your experience with Product Line Engineering?

None

Little

Average

Substantial

Professional

☐☐☐☐☐

Any Comments?

### SPENT EFFORT

3. How much effort (in minutes) did you spend with Task 1

Any Comments?

4. How much effort (in minutes) did you spend with Task 2

Any Comments?

5. How much effort (in minutes) did you spend with Task 3

Any Comments?

6. How much effort (in minutes) did you spend with Task 4

Any Comments?

7. How much effort (in minutes) did you spend with Task 5

Any Comments?

## UNDERSTANDABILITY

8. If you were in the SVN group please describe how well you understood the functionality of Subversion (e.g. branching functionality).

Not at all

Poorly

Fairly

Well

Highly

☐
☐
☐
☐
☐

Any Comments?

9. If you were in the CL group please describe how well you understood the functionality of the Customization Layer

Not at all

Poorly

Fairly

Well

Highly

☐
☐
☐
☐
☐

Any Comments?

## GENERAL FEEDBACK

10. Please include any other comments or suggestions that you'd like to share



## D.3 UTAUT forms

**PERFORMANCE EXPECTANCY**

| <b>Performance expectancy</b><br>Performance expectancy is defined as the degree to which you believe that using the "Provided Tooling" will help him or her to attain gains in job performance. | Agree                    |                          |                          | Disagree                 |                          |                          |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|
| I would find the "Provided Tooling" useful in my work.                                                                                                                                           | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| Using the "Provided Tooling" enables me to accomplish tasks more quickly.                                                                                                                        | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| Using the "Provided Tooling" increases my productivity.                                                                                                                                          | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| If I use the "Provided Tooling", I will increase my chances of getting a raise. (e.g., by being faster)                                                                                          | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |

**EFFORT EXPECTANCY**

| <b>Effort expectancy</b><br>Effort expectancy is defined as the degree of ease associated with the use of the "Provided Tooling". | Agree                    |                          |                          | Disagree                 |                          |                          |
|-----------------------------------------------------------------------------------------------------------------------------------|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|
| My interaction by using the "Provided Tooling" would be clear and understandable.                                                 | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| It would be easy for me to become skilful by using the "Provided Tooling".                                                        | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| I would find the "Provided Tooling" easy to use.                                                                                  | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| Learning to use the "Provided Tooling" was easy for me.                                                                           | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |

**ATTITUDE TOWARD USING TECHNOLOGY**

| <b>Attitude toward using technology</b><br>Attitude toward using technology is defined as your overall affective reaction to using the "Provided Tooling". | Agree                    |                          |                          | Disagree                 |                          |                          |
|------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|
| Using the "Provided Tooling" is a good idea.                                                                                                               | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| The "Provided Tooling" makes work more interesting.                                                                                                        | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| Using the "Provided Tooling" is fun.                                                                                                                       | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| I like using the "Provided Tooling".                                                                                                                       | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |

## SOCIAL INFLUENCE

| <b>Social influence</b><br>Social influence is defined as the degree to which you perceive that important others believe he or she should use the "Provided Tooling". | <b>Agree</b>             |                          |                          | <b>Disagree</b>          |                          |                          |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|
| People who influence my behavior think that I should use the "Provided Tooling".                                                                                      | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| People who are important to me think that I should use the "Provided Tooling".                                                                                        | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| The senior management has been helpful in the use of the "Provided Tooling".                                                                                          | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| In general, the organization has supported the use of the "Provided Tooling".                                                                                         | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |

## FACILITATING CONDITIONS

| <b>Facilitating conditions</b><br>Facilitating conditions are defined as the degree to which you believe that an organizational and technical infrastructure exists to support the use of the "Provided Tooling". | <b>Agree</b>             |                          |                          | <b>Disagree</b>          |                          |                          |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|
| I have the resources necessary to use the "Provided Tooling".                                                                                                                                                     | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| I have the knowledge necessary to use the "Provided Tooling".                                                                                                                                                     | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| The "Provided Tooling" is compatible with other methodologies I use.                                                                                                                                              | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| A specific person (or group) is available for assistance with "Provided Tooling" difficulties.                                                                                                                    | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |



---

# Lebenslauf

|                            |                        |                                                                                                      |
|----------------------------|------------------------|------------------------------------------------------------------------------------------------------|
| <b>Name</b>                | Michail Anastasopoulos |                                                                                                      |
| <b>Wohnort</b>             | Hamburg                |                                                                                                      |
| <b>Geburtsdatum</b>        | 19.10.1974             |                                                                                                      |
| <b>Geburtsort</b>          | Athen, Griechenland    |                                                                                                      |
| <b>Familienstand</b>       | Verheiratet            |                                                                                                      |
| <b>Staatsangehörigkeit</b> | Griechisch             |                                                                                                      |
| <b>Schulbildung</b>        | 1980-1987              | Grundschule                                                                                          |
|                            | 1987-1992              | Gymnasium                                                                                            |
|                            |                        | Abschluss: Abitur                                                                                    |
| <b>Wehrdienst</b>          | 2002-2003              | Grundwehrdienst griechisches Militär                                                                 |
| <b>Studium</b>             | 1992-1997              | Universität Patras, Griechenland                                                                     |
|                            | 1997-1999              | Technische Universität Dresden                                                                       |
| <b>Berufstätigkeit</b>     | 1999-2011              | Wissenschaftlicher Mitarbeiter am<br>Fraunhofer-Institut für<br>Experimentelles Software Engineering |
|                            | 2011-heute             | System Engineer bei Kuehne+Nagel<br>(AG & Co.) KG                                                    |

Hamburg, den 15. April 2014



# PhD Theses in Experimental Software Engineering

- Volume 1**      **Oliver Laitenberger** (2000), *Cost-Effective Detection of Software Defects Through Perspective-based Inspections*
- Volume 2**      **Christian Bunse** (2000), *Pattern-Based Refinement and Translation of Object-Oriented Models to Code*
- Volume 3**      **Andreas Birk** (2000), *A Knowledge Management Infrastructure for Systematic Improvement in Software Engineering*
- Volume 4**      **Carsten Tautz** (2000), *Customizing Software Engineering Experience Management Systems to Organizational Needs*
- Volume 5**      **Erik Kamsties** (2001), *Surfacing Ambiguity in Natural Language Requirements*
- Volume 6**      **Christiane Differding** (2001), *Adaptive Measurement Plans for Software Development*
- Volume 7**      **Isabella Wieczorek** (2001), *Improved Software Cost Estimation A Robust and Interpretable Modeling Method and a Comprehensive Empirical Investigation*
- Volume 8**      **Dietmar Pfahl** (2001), *An Integrated Approach to Simulation-Based Learning in Support of Strategic and Project Management in Software Organisations*
- Volume 9**      **Antje von Knethen** (2001), *Change-Oriented Requirements Traceability Support for Evolution of Embedded Systems*
- Volume 10**     **Jürgen Münch** (2001), *Muster-basierte Erstellung von Software-Projektplänen*
- Volume 11**     **Dirk Muthig** (2002), *A Light-weight Approach Facilitating an Evolutionary Transition Towards Software Product Lines*
- Volume 12**     **Klaus Schmid** (2003), *Planning Software Reuse – A Disciplined Scoping Approach for Software Product Lines*
- Volume 13**     **Jörg Zettel** (2003), *Anpassbare Methodenassistenz in CASE-Werkzeugen*
- Volume 14**     **Ulrike Becker-Kornstaedt** (2004), *Prospect: a Method for Systematic Elicitation of Software Processes*
- Volume 15**     **Joachim Bayer** (2004), *View-Based Software Documentation*
- Volume 16**     **Markus Nick** (2005), *Experience Maintenance through Closed-Loop Feedback*

- Volume 17**     **Jean-François Girard** (2005), *ADORE-AR: Software Architecture Reconstruction with Partitioning and Clustering*
- Volume 18**     **Ramin Tavakoli Kolagari** (2006), *Requirements Engineering für Software-Produktlinien eingebetteter, technischer Systeme*
- Volume 19**     **Dirk Hamann** (2006), *Towards an Integrated Approach for Software Process Improvement: Combining Software Process Assessment and Software Process Modeling*
- Volume 20**     **Bernd Freimut** (2006), *MAGIC: A Hybrid Modeling Approach for Optimizing Inspection Cost-Effectiveness*
- Volume 21**     **Mark Müller** (2006), *Analyzing Software Quality Assurance Strategies through Simulation. Development and Empirical Validation of a Simulation Model in an Industrial Software Product Line Organization*
- Volume 22**     **Holger Diekmann** (2008), *Software Resource Consumption Engineering for Mass Produced Embedded System Families*
- Volume 23**     **Adam Trendowicz** (2008), *Software Effort Estimation with Well-Founded Causal Models*
- Volume 24**     **Jens Heidrich** (2008), *Goal-oriented Quantitative Software Project Control*
- Volume 25**     **Alexis Ocampo** (2008), *The REMIS Approach to Rationale-based Support for Process Model Evolution*
- Volume 26**     **Marcus Trapp** (2008), *Generating User Interfaces for Ambient Intelligence Systems; Introducing Client Types as Adaptation Factor*
- Volume 27**     **Christian Denger** (2009), *SafeSpection – A Framework for Systematization and Customization of Software Hazard Identification by Applying Inspection Concepts*
- Volume 28**     **Andreas Jedlitschka** (2009), *An Empirical Model of Software Managers' Information Needs for Software Engineering Technology Selection  
A Framework to Support Experimentally-based Software Engineering Technology Selection*
- Volume 29**     **Eric Ras** (2009), *Learning Spaces: Automatic Context-Aware Enrichment of Software Engineering Experience*
- Volume 30**     **Isabel John** (2009), *Pattern-based Documentation Analysis for Software Product Lines*
- Volume 31**     **Martín Soto** (2009), *The DeltaProcess Approach to Systematic Software Process Change Management*
- Volume 32**     **Ove Armbrust** (2010), *The SCOPE Approach for Scoping Software Processes*

- Volume 33**     **Thorsten Keuler** (2010), *An Aspect-Oriented Approach for Improving Architecture Design Efficiency*
- Volume 34**     **Jörg Dörr** (2010), *Elicitation of a Complete Set of Non-Functional Requirements*
- Volume 35**     **Jens Knodel** (2010), *Sustainable Structures in Software Implementations by Live Compliance Checking*
- Volume 36**     **Thomas Patzke** (2011), *Sustainable Evolution of Product Line Infrastructure Code*
- Volume 37**     **Ansgar Lamersdorf** (2011), *Model-based Decision Support of Task Allocation in Global Software Development*
- Volume 38**     **Ralf Carbon** (2011), *Architecture-Centric Software Producibility Analysis*
- Volume 39**     **Florian Schmidt** (2012), *Funktionale Absicherung kamerabasierter Aktiver Fahrerassistenzsysteme durch Hardware-in the-Loop-Tests*
- Volume 40**     **Frank Elberzhager** (2012), *A Systematic Integration of Inspection and Testing Processes for Focusing Testing Activities*
- Volume 41**     **Matthias Naab** (2012), *Enhancing Architecture Design Methods for Improved Flexibility in Long-Living Information Systems*
- Volume 42**     **Marcus Ciolkowski** (2012), *An Approach for Quantitative Aggregation of Evidence from Controlled Experiments in Software Engineering*
- Volume 43**     **Igor Menzel** (2012), *Optimizing the Completeness of Textual Requirements Documents in Practice*
- Volume 44**     **Sebastian Adam** (2012), *Incorporating Software Product Line Knowledge into Requirements Processes*
- Volume 45**     **Kai Höfig** (2012), *Failure-Dependent Timing Analysis – A New Methodology for Probabilistic Worst-Case Execution Time Analysis*
- Volume 46**     **Kai Breiner** (2013), *AssistU – A framework for user interaction forensics*
- Volume 47**     **Rasmus Adler** (2013), *A model-based approach for exploring the space of adaptation behaviors of safety-related embedded systems*
- Volume 48**     **Daniel Schneider** (2014), *Conditional Safety Certification for Open Adaptive Systems*
- Volume 49**     **Michail Anastasopoulos** (2013), *Evolution Control for Software Product Lines: An Automation Layer over Configuration Management*



Software Engineering has become one of the major foci of Computer Science research in Kaiserslautern, Germany. Both the University of Kaiserslautern's Computer Science Department and the Fraunhofer Institute for Experimental Software Engineering (IESE) conduct research that subscribes to the development of complex software applications based on engineering principles. This requires system and process models for managing complexity, methods and techniques for ensuring product and process quality, and scalable formal methods for modeling and simulating system behavior. To understand the potential and limitations of these technologies, experiments need to be conducted for quantitative and qualitative evaluation and improvement. This line of software engineering research, which is based on the experimental scientific paradigm, is referred to as 'Experimental Software Engineering'.

In this series, we publish PhD theses from the Fraunhofer Institute for Experimental Software Engineering (IESE) and from the Software Engineering Research Groups of the Computer Science Department at the University of Kaiserslautern. PhD theses that originate elsewhere can be included, if accepted by the Editorial Board.

Editor-in-Chief: Prof. Dr. Dieter Rombach

Executive Director of Fraunhofer IESE and Head of the AGSE Group of the Computer Science Department, University of Kaiserslautern

Editorial Board Member: Prof. Dr. Peter Liggesmeyer

Scientific Director of Fraunhofer IESE and Head of the AGDE Group of the Computer Science Department, University of Kaiserslautern

Editorial Board Member: Prof. Dr. Frank Bomarius

Deputy Director of Fraunhofer IESE and Professor for Computer Science at the Department of Engineering, University of Applied Sciences, Kaiserslautern

ISBN 978-3-8396-0702-2

