Magnus Eriksson

An Approach to Software Product Line Use Case Modeling

LICENTIATE THESIS, 2006

DEPARTMENT OF COMPUTING SCIENCE
UMEÅ UNIVERSITY
SE-90187 UMEÅ, SWEDEN

Print & Media, Umeå University

UMINF-06.01 ISSN 0348-0542 ISBN 91-7264-023-5

Abstract

Organizations developing software intensive defense systems are today faced with a number challenges related to characteristics of both the market place and the system domain:

1. Systems grow ever more complex, consisting of tightly integrated mechanical, electrical/electronic and software components.
2. Systems are often developed in short series; ranging from only a few to a few hundred units.
3. Systems have very long life spans, typically 30 years or longer.
4. Systems are developed with high commonality between different customers; however systems are always customized for specific needs.

The goal of the research presented in this thesis is to investigate methods and tools to enable efficient development and maintenance of systems in such a context. The strategy adopted in this work is to utilize the forth system characteristic, high commonality, to achieve this.

One approach to software reuse, which could be a potential solution as it enables reuse of common parts but at the same time allow for variations, is known as software product line development. The basic idea of this approach is to use domain knowledge to identify common parts within a family of related products and to separate them from the differences between the products. The commonalties are then used to create a product platform that can be used as a common baseline for all products within such a product family.

The main contribution of this licentiate thesis is a product line use case modeling approach tailored towards organizations developing software intensive defense systems. We describe how a common and complete use case model can be developed and maintained for a whole family of products, and how the variations within such a family are modeled using a feature model. Concrete use case models, for particular products within a family, can then be generated by selecting features from a feature model. We furthermore describe extensions to the commercial requirements management tool Telelogic DOORS and the UML modeling tool IBM-Rational Rose to support the proposed approach. The approach was applied and evaluated in an industrial case study in the target domain. Based on the collected case study data we draw the conclusion that the approach performs better than modeling according to the styles and guidelines specified by the IBM-Rational Unified Process (RUP) in the current industrial context. The results however also indicate that for the approach to be successfully applied, stronger configuration management and product planning functions than traditionally found in RUP projects are needed.

Acknowledgements

There are many people involved in the completion of this licentiate thesis that I am very grateful to. First of all, I would like to thank my academic supervisor Associate Professor Jürgen Börstler, and my industrial supervisor Kjell Borg. You always make yourselves available to discuss my ideas, and your advice has significantly increased the quality of my work.

Dr. Örjan Olsson and Mats Bergström, without your support this doctorial project would never have been initiated in first place. Mats, I am really sad that you are no longer with us to see how it is starting to transform development at Hägglunds.

Henrik Morast, without your enthusiasm developing tools to support my ideas, they would have had far less influence on development at Hägglunds. Thank you also for your sanity checks of my tool ideas.

Dr. Carl-Gustav Löf and Conny Flemin, thank you for believing in me and providing me with the means to apply my ideas at Hägglunds.

All the people at Hägglunds that applied my ideas and participated in my studies, without you, none of this work would have been possible. Thank you all!

Dr. Charilaos Christopoulos, my master thesis supervisor at Ericsson Research, without your encouragement I would never have pursued a research degree in the first place.

Finally, I would like to thank my family for their support, but also apologize to them. Conducting process improvement activities is very rewarding when things work out right, but sometimes also very frustrating when they do not. Thank you for always being there for me when I needed you, and sorry for all those long hours that were required to finish this work.

Umeå, December 2005
Magnus Eriksson

Preface

This licentiate thesis consists of the following three papers and an introduction to the research area (Kappa):

I. M. Eriksson, J. Börstler & K. Borg (2004): *Marrying Features and Use Cases for Product Line Requirements Modeling of Embedded Systems*[1], Proceedings of the Fourth Conference on Software Engineering Research and Practice in Sweden, Institute of Technology, UniTryck, Linköping University, Sweden, pp. 73-82

II. M. Eriksson, J. Börstler & K. Borg (2005): *The PLUSS Approach — Domain Modeling with Features, Use Cases and Use Case Realizations*[2], Proceedings of the 9'th International Conference on Software Product Lines, LNCS, Vol. 3714, Springer-Verlag, pp. 33-44

III. M. Eriksson, H. Morast, J. Börstler & K. Borg (2005): *The PLUSS Toolkit — Extending Telelogic DOORS and IBM-Rational Rose to Support Product Line Use Case Modeling*[3], Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, Long Beach California, ACM Press, pp. 300-304

These papers have been reformatted, compared to the original publication, to have a consistent layout with the rest of the thesis. Furthermore, minor typographical errors have also been corrected.

---

[1] © Magnus Eriksson, 2004

[2] © Springer-Verlag, 2005

[3] © ACM Press, 2005

Table of Contents

1 Thesis Introduction

1.1 Background and Motivation

Organizations developing software intensive defense systems are today faced with a number challenges. These challenges, which are related to characteristics of both the market place and the system domain, include:

1. Systems grow ever more complex, consisting of tightly integrated mechanical, electrical/electronic and software components. This implies that strong means of communication between different engineering disciplines are important to achieve efficient development.
2. Systems are often developed in short series; ranging from a few to a few hundred units. This implies that it is important to achieve efficient development, since development costs are carried by only a few units.
3. Systems have very long life spans, typically 30 years or longer. This implies that it is important to develop high quality systems, and to achieve effective maintenance of these systems once developed.
4. Systems are developed with high commonality between different customers; however systems are always customized for specific needs. This implies that there is potential for high levels of reuse of development efforts between different customer projects.

1.2 Research Question

The research presented in this thesis is intended to address some of the complexity related to development and maintenance of systems such as those described above.

The research question investigated in this licentiate thesis is:

> *What methods and tools are needed to enable effective development and maintenance of complex and long-lived software intensive systems?*

1.3 Research Context

The work presented in this licentiate thesis is financed by, and performed in collaboration with, Land Systems Hägglunds AB. Land Systems Hägglunds, which is part of the Land Systems division of BAE Systems, is a leading developer and manufacturer of combat vehicles, all terrain vehicles and a supplier of various turret systems.

To address some of the complexity related to the development of such systems (as discussed in section 1.1), Land System Hägglunds has a systems engineering [31] team which is responsible for system-wide technical issues (see Fig. 1). Systems engineering is an interdisciplinary approach to enable the realization of complex systems [30]. Its focus is on defining stakeholder needs and required functionality early in the development cycle and to synthesis an overall system design that captures those requirements from a total life-cycle perspective (see Fig. 2).

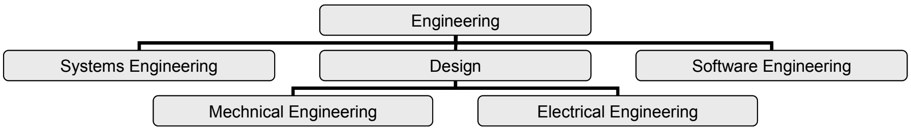


Fig. 1: A partial view of Land Systems Hägglunds organization.

Land Systems Hägglunds develops software according to a tailored version of the IBM-Rational Unified Process (RUP) [48]. RUP, which is widely used in industry, is a specific and detailed version of the more general Unified Software Development Process (USDP) [40]. RUP will be further discussed in section 5.

System development projects at Land Systems Hägglunds are often constrained by different types of standards prescribed by acquisition organizations (customers). These standards typically prescribe certain artifacts to be developed and certain processes to be executed. The organization is also certified according to the ISO 9001 [32] standard.

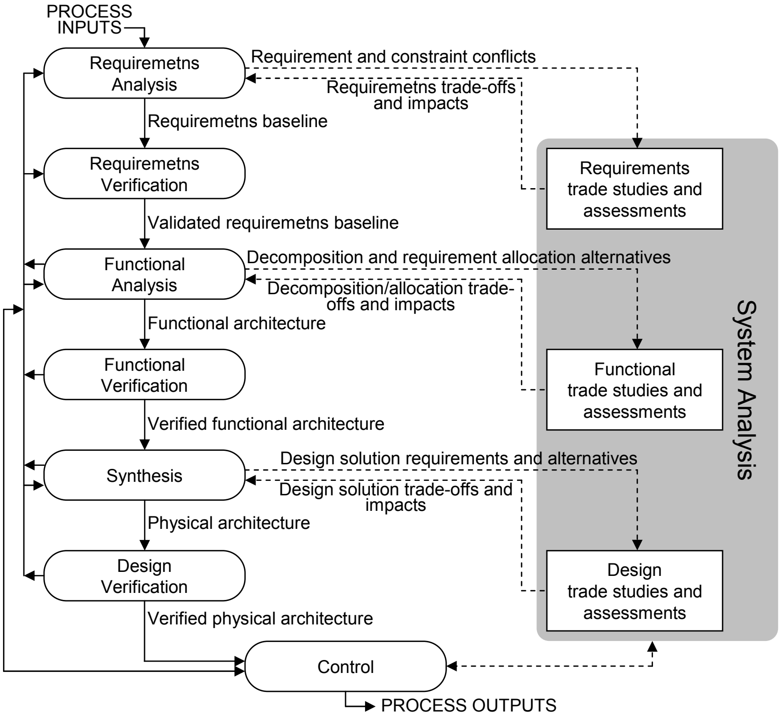

PROCESS INPUTS
Requiremetns Analysis
Requirement and constraint conflicts
Requiremetns trade-offs and impacts
Requiremetns baseline
Requiremetns Verification
Requirements trade studies and assessments
Validated requiremetns baseline
Functional Analysis
Decomposition and requirement allocation alternatives
Decomposition/allocation trade-offs and impacts
Functional architecture
Functional Verification
Functional trade studies and assessments
System Analysis
Verified functional architecture
Synthesis
Design solution requirements and alternatives
Design solution trade-offs and impacts
Physical architecture
Design Verification
Design trade studies and assessments
Verified physical architecture
Control
PROCESS OUTPUTS

Fig. 2: The Systems Engineering Process [30].

1.4 Research Hypothesis

The strategy adopted in this work was to take advantage of the forth system characteristic described in section 1.1 (high commonality) to enable effective development and maintenance of systems in the target domain. One approach to software reuse, which could be a potential solution that utilizes this characteristic, is known as software product line development. The basic idea of this approach is to use domain knowledge to identify common parts within a family of related products and to separate them from the differences between the products. The commonalties are then used to create a product platform that can be used as a common baseline for all products within the product family.

The research hypothesis on which the work presented in this thesis is based on is therefore:

> *Adopting a software product line development approach enables effective development and maintenance of complex and long-lived software intensive systems.*

A formal capability assessment of Land Systems Hägglunds in accordance with the ISO/IEC 15504 standard (SPICE) [33,34,35,36,37], revealed system- and software requirements engineering to be important areas on which to focus process improvement efforts [51]. A decision was therefore made to investigate if software product line development could be introduced in the organization using a requirements-based approach.

1.5 Thesis Outline

The remainder of this thesis is structured as follows: Section 2 describes the research method adopted in this work. Section 3, 4 and 5 provides introductions to software product line development, requirements engineering and RUP respectively. Section 6 presents the main contribution of this thesis, an approach to software product line use case modeling. Section 7 summarizes the thesis contributions and section 8 presents some ongoing and future work in the area.

2 Research Method

An "Industry-as-laboratory" [62] (see Fig. 3) approach was chosen for this work. The motivation for this was to allow for frequent exchange of information from the problem domain (industry) to the academic domain and back.

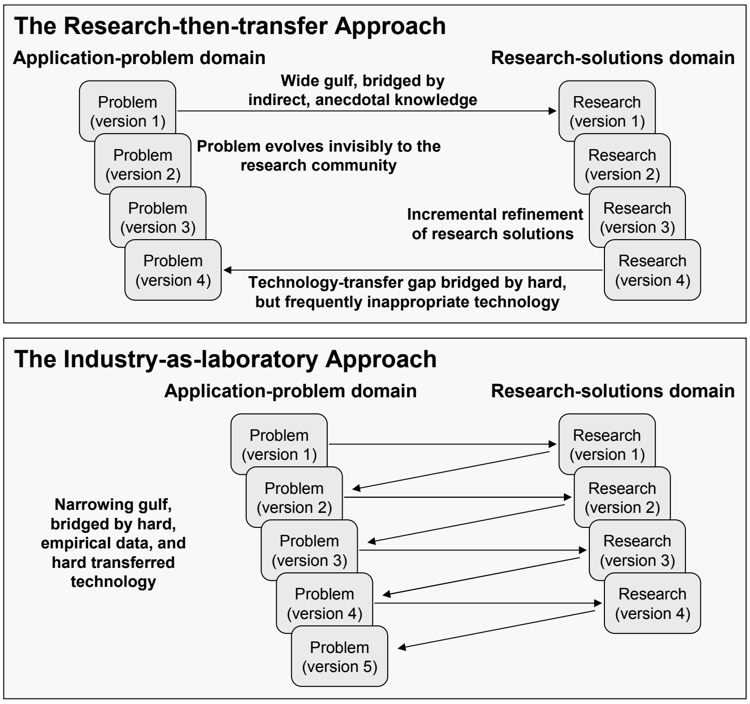

The Research-then-transfer Approach
Application-problem domain
Research-solutions domain
Problem (version 1)
Problem (version 2)
Problem (version 3)
Problem (version 4)
Research (version 1)
Research (version 2)
Research (version 3)
Research (version 4)
Wide gulf, bridged by indirect, anecdotal knowledge
Problem evolves invisibly to the research community
Incremental refinement of research solutions
Technology-transfer gap bridged by hard, but frequently inappropriate technology
The Industry-as-laboratory Approach
Application-problem domain
Research-solutions domain
Narrowing gulf, bridged by hard, empirical data, and hard transferred technology
Problem (version 1)
Problem (version 2)
Problem (version 3)
Problem (version 4)
Problem (version 5)
Research (version 1)
Research (version 2)
Research (version 3)
Research (version 4)

Fig. 3: The "Industry-as-laboratory" approach [62].

The industry-as-laboratory approach was applied as illustrated in Fig. 4, where industry expresses the initial research problem. This problem is then analyzed by academia, and a solution to the problem is proposed. Industry then executes one or more pilot projects where the proposal is applied in the problem domain. Academia supports this activity by training personnel in new methods and tools, and by assuming a mentoring role during pilots.

In parallel to, and after these pilots, academia collects data to enable an empirical evaluation of the proposal. This data is typically of qualitative type. Since qualitative data is richer than quantitative data [66], it is often a better choice when gathered from only one or a few pilot projects. Example sources of such qualitative data are document analysis, participant observation, questionnaires and interviews [50].

Based on the empirical evaluation, a decision is made to either refine/reject the proposal or to institutionalize the change and move on to other research problems. If a decision is made to refine or reject the proposal, the process returns to the problem analysis activity which is then followed by new proposals and new pilots. If a decision is made to institutionalize the change, industry will incorporate the proposal in its quality system and also apply it in future projects. This enables academia to perform follow-up studies on a larger set of projects. Evaluating proposals in such a setting typically involves collection of quantitative data. Example sources of such data are surveys, and metrics from both historical (pre-change) and new (post-change) projects.

One risk associated with this research strategy is the close involvement of the research team with the development teams. This confounding factor[4] may affect the internal validity of any empirical evaluations performed. It is therefore important to take this fact into consideration during data analysis. To minimize the effect of other confounding factors it is also important that pilots are staffed using normal procedures, subjects are given adequate training, and that subjects have sufficient experience of the new technology prior to pilots [44].

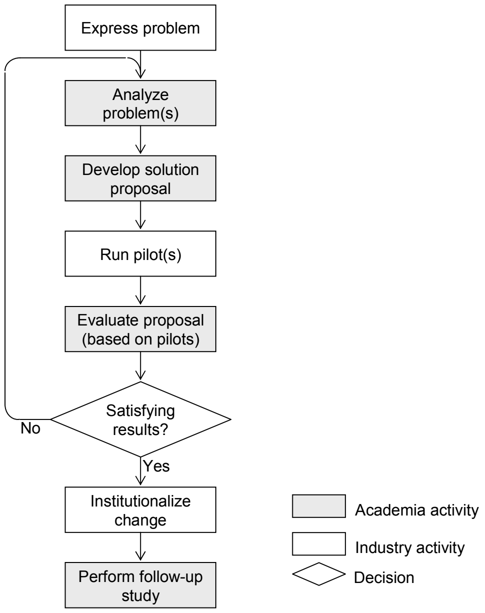


Fig. 4: An overview of the applied research approach.

3 Software Product Lines

Over the last few years a new[5] approach to software reuse has gained considerable attention both by industry and academia. This approach is known as software product line development and it supports large-grained (architecture level) intra-organization reuse. Software product line[6] development is an approach to gain organizational benefits by exploiting commonalities between a set of related products that address a particular market segment. The basic idea of the software product line approach is to

---

[4] A confounding factor is one that can not properly be distinguished form another factor measured in a study [45].

[5] The basic concepts were actually presented already in the seventies by David Parnas [61].

[6] A number of software product line development methods have been proposed in the literature. Surveys of the most important ones can be found in [71] and [25].

use domain knowledge to identify and separate common parts among a family of products from the differences between the products. The commonalties are used to create a product platform that can be used as a common baseline for all products within a product family. Studies have shown that organizations can yield considerable improvements in productivity, time to market, product quality and customer satisfaction by applying this approach [12,13,15].

In this work the term *Product Line* or *Product Family* is used to denote [15]:

> “a set of software-intensive systems sharing a common set of features that satisfy the specific needs of a particular market or mission and that are developed from a common set of core assets in a prescribed way”.

The term *Core Assets* or *Platform* is used to denote the reuse repository of a product line. These software product line core assets include, not only software components, but also often architecture, requirements, documentation, schedules, budgets, test plans, test cases, etc. [58].

3.1 Reuse

The main purpose with software reuse is to improve software quality and productivity, and thereby maximize a software development organizations profit [25]. The software engineering community has had long-standing high hopes that software reuse would be the answer to the “software crisis”[7]. A number of software reuse approaches have been presented over the years. One example of such an approach is the object oriented programming paradigm (OOP). OOP supports software reuse by techniques known as polymorphism, encapsulation and inheritance [22]. These techniques help the developer in producing highly modular and to some extent reusable code. Much research has also been done on reuse libraries[8] [25]. The basic idea of such traditional software reuse approaches is that organizations create repositories where the outputs of practically all development efforts are stored. These repositories would typically contain components, modules and algorithms that developers are then urged to use. Unfortunately, it usually takes longer to find the desired functionality and adapting it to current needs than it would to build it anew [12]. The typical programmer solution to this problem is to ignore the legacy and build most of the software from scratch. Traditional techniques, which support so called small-grained[9] reuse, have therefore proved ineffective[10] when trying to address the software crisis in practice [12].

Another and more effective approach to software reuse is known as the “clone and own” approach [15]. When a new product project is initiated using this approach, the development team tries to find another product within the organization that resembles the current product as much as possible. The organization then copies (clones) all

---

[7] See [26] for more information regarding the ‘software crisis’.

[8] See [55] for more information regarding research on reuse libraries.

[9] Also known as ‘code salvation’ or ‘code scavenging’ (see [22,12]).

[10] One exception is the “Japanese Software Factories” approach, which proved very successful in the 1970’s and 1980’s. The main weakness of this approach was however that it had too much focus on process improvement, and not enough support for product innovation [17].

project artifacts, and modify and add whatever needed to launch the new product. This approach can yield considerable savings compared to developing all products from scratch. One drawback with the clone-and-own approach is however inefficient maintenance. When “cloning” an existing product to create a new product, its maintenance trajectory is split into two separate paths. This could lead to considerable additional maintenance costs for the common parts of the products over their lifespan.

Software product lines are about strategic reuse, this means that software product lines are as much about business practices as they are about technical practices [58]. Adopting a software product line approach requires a shift in mind for an organization. An organization must move from developing single products to developing product families. During analysis several related products are envisioned together and a design that can capture the requirements of the whole family must be developed. This means that everything is developed with reuse (within the family) in mind. This in turn implies that the effort needed for customization of the reusable assets to fit a new system is largely reduced compared to traditional reuse approaches. Another benefit of software product line development compared to traditional reuse is maintenance. In software product line development, products are built on a common platform and maintenance costs of the platform can be shared by all products using the platform.

3.2 Product Line Management

As illustrated in Fig. 5 and Fig. 6, development in a software product line organization can be divided into two main activities, *Domain engineering* and *Application engineering*:

- The purpose of the domain engineering activity is to develop the product line reusable core assets. The goal of this core asset development is to provide a production capability for products [58]. Together with these core assets, some sort of production plan [15] is also developed. The purpose of a production plan is to describe how products are to be built from a core asset. For example by describing how specific tools are to be applied in order to use, tailor and evolve assets.
- The purpose of the application engineering activity is to generate new applications utilizing the assets developed by domain engineering. The main input to this activity, besides from core assets and production plans, is product requirements.

As discussed above, in software product lines, reuse is planned, enabled and enforced [15]. This implies that management is an integral part of any successful product line effort. Both technical (project) and organizational management must be strongly committed to the product line effort [58]. Technical management oversees core asset development and enforces use of the core assets by product development teams. Organizational management must set necessary organizational structures, such as funding models, in place to ensure the evolution of core assets.

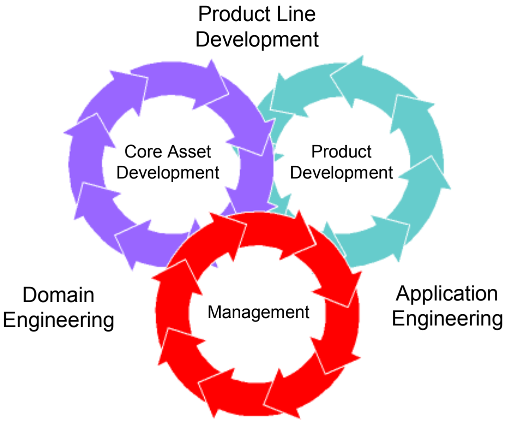


Fig. 5: Essential product line activities [58].

3.3 Essential Artifacts

As illustrated in Fig. 6, some of the key artifacts of a product line are its requirements, its architecture and its components. However, compared to single system development, a few differences exist in these product line artifacts:

- *Product line requirements* span several products. This means that some of these product-line-wide requirements must be written with variation points to be able to capture variations between individual products within a product line. Mannion classifies product line requirements to be "Non-reusable", "Directly reusable", "Variable" (see Fig. 7) or "Obsolete" [53].
- *Product line architectures* define a set of explicitly allowed variations that represent the individual products that can be built within a product line. In conventional software architectures, almost any variation is allowed as long as the product requirements are fulfilled. It is also the product line architectures' responsibility to provide the necessary mechanisms to implement these variations [15]. A number of variability mechanisms (see for example [68,70]), and product line architecture design methods (see for example COPA [4], FAST [75], FORM [43], KobrA [5] and QADA [57])[11] can be found in the literature.
- *Product lines components* can either be a part of the core assets, or they can be developed for product specific reasons. Even though software product line development employ a form of component-based development [69], a few differences exist compared to the view of components in other settings. For example:

---

[11] Further discussion of these methods is not within scope of this thesis, a summary and comparison of these methods can be found in [54].

- Product line components are typically not independently deployed. Product line components are assembled in a prescribed way specified by their production plans and the product line architecture [15].
- Product line components implement variability mechanisms specified by the product line architecture [12]. Fig. 8 shows an overview of the activities and artifacts leading up to component design and implementation in a software product line context.

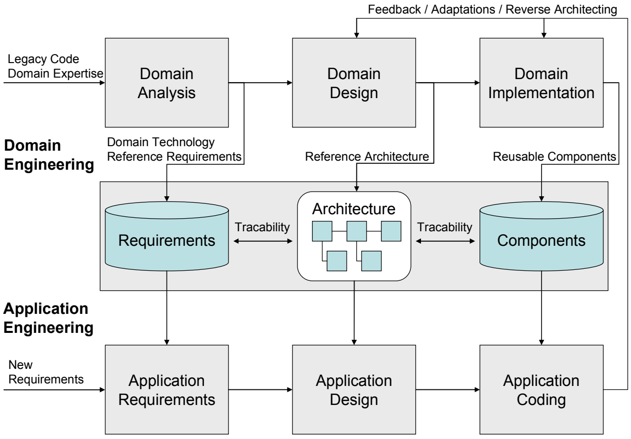


Fig. 6: The ESAPS reference process [72].

CMD-01220: It shall be possible to define up to a maximum of **@MAXNUMCMD** command sub-systems where **MAXNUMCMD** can not be greater than 255.

Fig. 7: An example of a variable (parameterized) requirement [53].

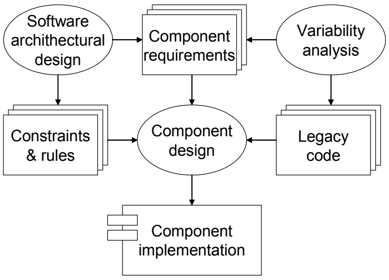


Fig. 8: Activities and deliverables in software product line component development [12].

4 Requirements Engineering

Software requirements engineering involves activities such as discovering, documenting and maintaining a set of requirements for a computer based system [67]. The purpose of the requirements engineering activities in a software project is to describe precisely what to build without describing how to build it. This seems like a simple task, however in large complex software projects, requirements are often considered to be the biggest software engineering challenge [24]. System/Software requirements can be divided into two main categories [67]:

- *Functional requirements*, which describe what the system should do.
- *Non-functional requirements*[12], which place constraints on how functional requirements are implemented.

One problem in requirements engineering is that requirements are continuously changing [48]. It is impossible to capture all requirements for a non-trivial system before development starts. As a system evolves during development, so does its requirements as the system stakeholders gain a better understanding of the system domain. It is therefore critical to keep track of the current status of each requirement throughout the project.

4.1 Traceability

A critical key to successful system development is the ability to understand relationships that exists between requirements, design, code, and tests [60]. The tool used to achieve this ability is referred to as traceability or requirements tracing. Lauesen defines four types of requirements tracing (see Fig. 9) [49]:

1. Forward tracing from demands (stakeholder needs) to system requirements (needed to verify that all demands are reflected by system requirements).
2. Forward tracing from system requirements to a system design (needed to verify that all system requirements are considered in the design).
3. Backward tracing from a system design to system requirements (needed to verify that all parts of the design are required).
4. Backward tracing from system requirements to demands (needed to see that all system requirements have a purpose).

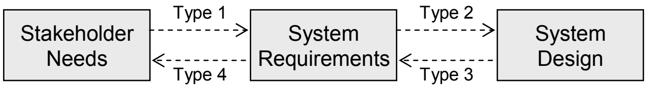


Fig. 9: Requirements traceability types.

---

[12] Also referred to as, for example, “Quality requirements” in [49] or, “Quality attributes” or “Constraints” in the systems engineering community. The term non-functional requirement is used in this work to be consistent with the RUP terminology.

4.2 Use Case Modeling

Use cases provide a semi-formal framework for modeling (mainly functional) requirements [39,1]. A use case can be described as goal that a user of a system want to accomplish by interacting with the system. These goals are depicted in UML use case diagrams [59]. Use case diagrams may contain two types of entities:

- Actors, depicted as stick figures (see Fig. 10), which represents users of the modeled system. These actors can be either human users or external systems.
- Use cases, depicted as ellipses, which can have association relationships to actors. An association relationship between an actor and a use case means that the actor can communicate with the use case. That is, either initiate or participate in the behavior specified in the use case.

These use cases are further specified by a number of use case scenarios (also referred to as use case instances). These scenarios, which describe interaction between a system and its actors, are typically described in informal natural language. However, UML Sequence diagrams and Activity diagrams [59] are other popular notations used for describing use case scenarios.

Typically, for each use case in a use case model, there is also a corresponding use case realization in a design model [7]. A use case realization is a description of how different design elements collaborate to solve a specific use case (see Fig. 10) [48]. The main purpose or a use case realization is to provide a bridge between requirements modeled as a use case and a systems' design (i.e. traceability). Use case realizations are often described using UML Sequence or Collaboration diagrams [7].

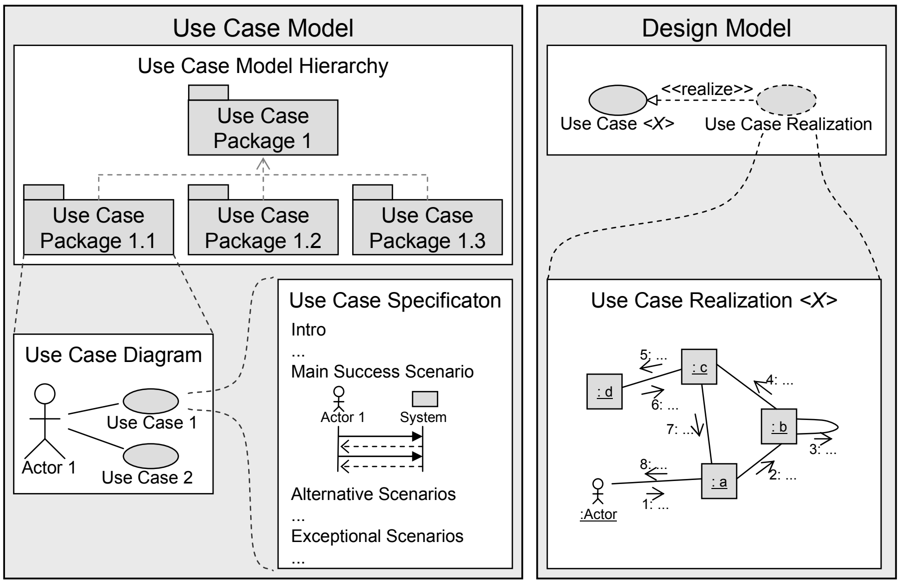


Fig. 10: An overview of use case modeling artifacts and concepts.

An interesting extension to use case modeling, from the perspective of software product lines, is known as Change case modeling. Change cases, which were proposed by Ecklund et al. at OOPSLA'96 [19], are basically use cases that specify anticipated changes to a system over its foreseeable lifetime. Change cases provide a relation "impact link" that creates traceability to use cases whose implementations are affected, if the change case is realized (see Fig. 11). Modeling change cases, allows product line designers to plan for and, more effectively, accommodate anticipated future requirements in a domain [15].

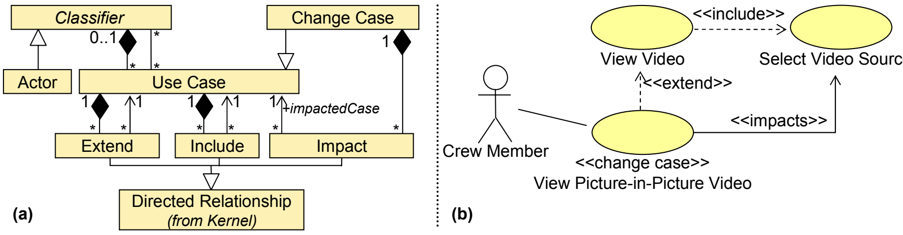


Fig. 11: (a) A change case meta-model[13] based on the UML use case meta-model, and (b) a change case example in a UML use case diagram.

4.3 Feature Modeling

The activity in which commonality and variability analysis is performed in software product line requirements engineering is commonly referred to as domain analysis. A widely used technique in domain analysis is Feature modeling [18]. Kang et al. first proposed using feature models in 1990 as part of Feature Oriented Domain Analysis (FODA) [42].

Kang et al. define a feature as: "A prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems". In feature models, system features are organized into trees of AND and OR nodes that represent the commonalities and variations within a family of related systems. General features are located at the top of the tree and more refined features are located below. Originally, FODA described "Mandatory", "Optional" and "Alternative" features that may have "requires" and "excludes" relations to other features. Mandatory features are available in all systems built within a family. Optional features represent variability within a family that may or may not be included in products. Alternative features represent an "exactly-one-out-of-many" selection that has to be made among a set of features. A "requires" relationship indicates that a feature depends on some other feature to make sense in a system. An "excludes" relationship between two features indicates that both features can not be included in the same system.

---

[13] A meta-model is an explicit model of constructs and rules needed to build a model within a specific domain (i.e. a meta-model is a model of how a specific type of model may be constructed).

Fig. 12 shows an example of a simple feature model in the FODA notation, however there are also a number of other feature modeling notations available in the literature. Robak has provided an overview of some commonly used ones in [64].

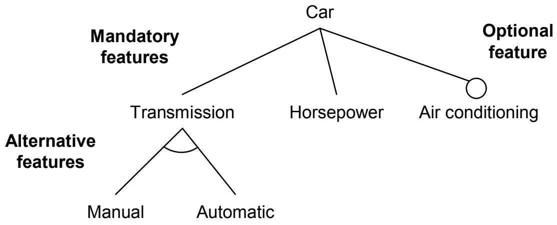


Fig. 12: A FODA Feature model [42].

5 The IBM-Rational Unified Process (RUP)

A software development process defines *who* is doing *what* and *how* to build or enhance a software product [40]. An effective process reduces risk and improves predictability by providing guidelines based on best practices for the development of quality software. The IBM-Rational Unified Process (RUP) [48] is a commercial product which provides a framework for such a software development process.

As mentioned in section 1.3, RUP is an instance of the Unified Software Development Process (USDP) framework [40]. There are also other instances of USDP available, for example the Agile Unified Process (AUP) [3] and the Enterprise Unified Process (EUP) [2]. However, further discussion of these other instances is not within the scope of this thesis.

As shown in Fig. 13, RUP has its roots in work preformed by the Ericsson Corporation in the late sixties on visual modeling of telecom systems using scenarios. This work was later refined into a process product developed by Objectory AB. At the same time (1987) the term "Use Case" was first introduced by Ivar Jacobson at OOPSLA [38], and it became a cornerstone of the developed process. In 1995 the Rational Software Corporation acquired Objectory AB [40]. This lead to further development of the process by adding ideas developed at Rational regarding for example architectural views [46] and on arranging iterative development into phases [40]. This led to the development of the Rational Objectory Process, which also adopted the Unified Modeling Language (UML) [59]. In 1998, Philippe Kruchten from Rational published the book "*The Rational Unified Process: An introduction*" [47] and thereby made the details of the proprietary process available to the general public for the first time. Today, RUP is well established and has become widely used in the software industry.

RUP has been developed based on six "*best practices*" which are adopted by many successful software development organizations [48]:

1. Develop Iteratively
2. Manage Requirements

3. Use Component Architectures
4. Model Visually
5. Continuously Verify Quality
6. Manage Change

The following sections will discuss these best practices in some more detail.

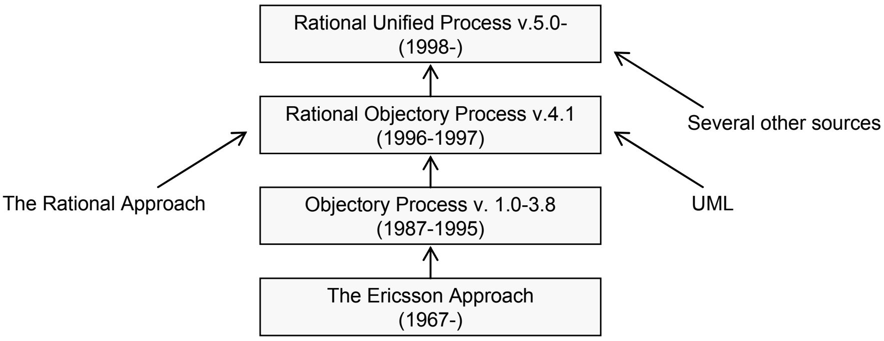

Rational Unified Process v.5.0- (1998-)
Rational Objectory Process v.4.1 (1996-1997)
Several other sources
The Rational Approach
Objectory Process v. 1.0-3.8 (1987-1995)
UML
The Ericsson Approach (1967-)

Fig. 13: History of the RUP [40].

5.1 Develop Iteratively

The iterative development approach is based on the spiral model (see Fig. 15) developed by Barry Boehm [8]. The Spiral model was intended to address shortcomings of the waterfall model [65] (see Fig. 14) which was widely accepted in industry at the time.

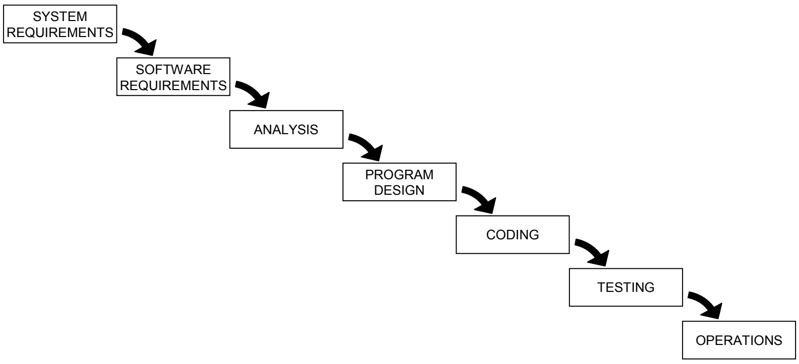


Fig. 14: The waterfall model [65].

The basic idea of iterative development is to develop systems incrementally by applying the waterfall model on portions of the system several consecutive times as illustrated in Fig. 16. These "miniature waterfalls" are referred to as iterations. This enables teams to work more risk-driven, since the most critical parts of the system can be developed and tested early in the project. It furthermore helps to find contradictions in requirements, design and implementations early, since an executable subset of the system is developed in each iteration.

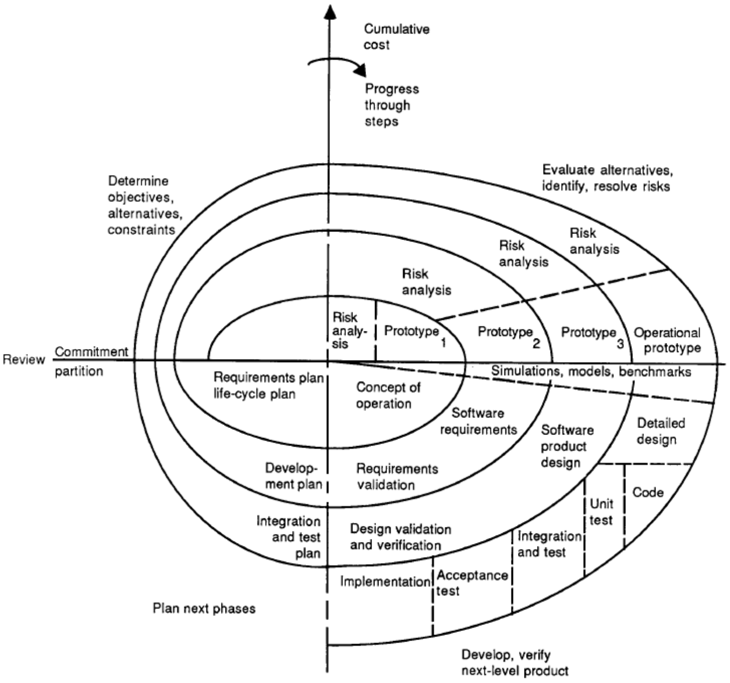


Fig. 15: The Spiral Model [8].

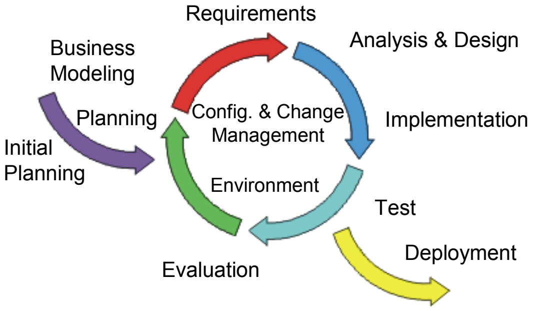


Fig. 16: Iterative and incremental development [48].

To make work more controlled, RUP group iterations by dividing projects into four phases: *Inception*, *Elaboration*, *Construction* and *Transition* (see Fig. 17). Each of these phases is related to a major project milestone which must be achieved before entering the next phase of the project. These RUP milestones are identical to the milestones that were proposed by Barry Boehm in 1996 [9,48]:

- The *Lifecycle Objectives Milestone*: The goal of the inception phase is to achieve concurrence among the system stakeholders on the life cycle objectives for the project. A major part of this is to determine scope and boundaries for the software to be developed. The major evaluation criteria for the lifecycle objectives milestone, which ends the inception phase, are:
    - Stakeholders agree upon system scope and project estimates.
    - Agreement exist that the right set of requirements has been captured.
    - Agreement exist that all major risk in the project have been identified, and that mitigation strategies exist for each of them.
- The *Life Cycle Architecture Milestone*: The main goal of the elaboration phase is to baseline the software architecture, to provide a stable basis for the bulk of the design and implementation work in the construction phase. The major evaluation criteria for the lifecycle architecture milestone, which ends the elaboration phase, are:
    - Requirements are stable.
    - The architecture is stable.
    - Major risk elements have been addressed by prototypes or other means.
- The *Initial Operational Capability Milestone*: The goal of the construction phase is to clarify the remaining requirements and develop the operational software based on the baselined software architecture. The major evaluation criteria for the initial operational capability milestone, which ends the construction phase, are:
    - The product is mature and stable enough to be deployed in the end-user community.
    - All stakeholders are ready for the transition to the end-users.
- The *Product Release Milestone*: The goal of the transition phase is to ensure that the software is readily available to its end-users. This includes activities such as testing and making minor adjustments based on user feedback. After the transition phase, the project lifecycle ends and the software product moves into its maintenance phase. The major evaluation criteria for the product release milestone, which ends the transition phase, is:
    - Customers have reviewed and accepted the project deliverables.

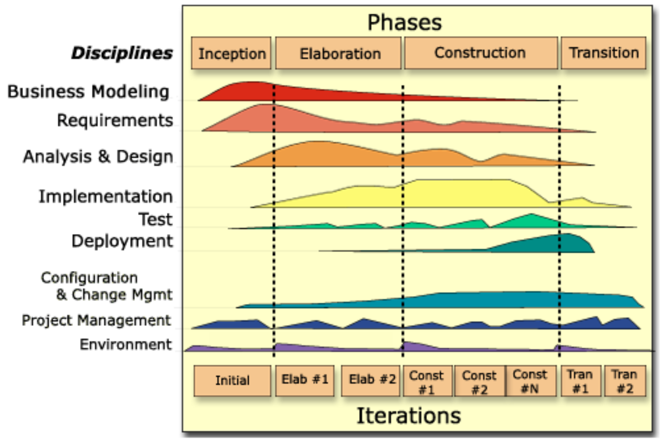


Fig. 17: An overview of RUP [48].

5.2 Manage Requirements

As discussed in section 4, managing requirements and maintaining traceability to the design are important activities in a software development project. This view has also been adopted in RUP. The most prominent requirements artifact in RUP is the use case model. RUP is often referred to as a use case driven methodology. The reason for this is that use cases form the basis for many activities in RUP, they drive [48]:

- Creation and validation of the design
- Definition of test cases and test procedures
- Project planning
- Development of user manuals
- Deployment

Non-functional requirements are managed in a natural language (text) specification called “Supplementary Specification” in RUP [48].

5.3 Use Component Architectures

The software architecture is the structure of a system, which comprises software components, the externally visible properties of those components, and the relationships among them [6]. The basic idea of component based development is that software building blocks (components) are pre-fabricated, deployed and assembled into a system. Components have clearly defined interfaces and can be (re)used independently of other components [69]. Iterative development combined with such component architectures means that a system can grow continuously. Each iteration produces an executable architecture that can be evaluated against system requirements [48]. RUP uses the “4+1 model view” [46] (see Fig. 18) to describe the software architecture.

- The *Logical View* describes the system architecture from a functional perspective.
- The *Development View* (referred to as *Implementation View* in RUP), describes how source code and other related static software modules such as data files are organized in the development environment.
- The *Process View* describes concurrency aspects within the system, such as thread management, deadlocks, fault tolerance, etc.
- The *Physical View* (referred to as *Deployment View* in RUP), describes how executable software modules are allocated to the underlying (hardware) platform.
- The *Scenario View* (referred to as *Use Case View* in RUP), has a special role since it ties the other views together. The use case view contains a number of key usage scenarios and descriptions of how the software architecture realizes these scenarios.

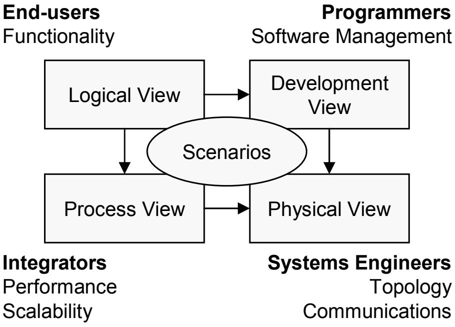


Fig. 18: The 4+1 View of Architecture [46].

5.4 Model Visually

A model is a simplification of reality that describes a system from a certain viewpoint [48]. Visual modeling helps teams to cope with system complexity by enabling abstraction. As mentioned in section 5, RUP has adopted UML [59] as its visual modeling language. UML provides a standardized graphical notation that can be used to specify, visualize, construct, and document the artifacts of software-intensive systems. UML includes language constructs to capture both system structure, behaviour and interactions.

5.5 Continuously Verify Quality

Finding and fixing a software problem is typically 5-100 times more expensive after delivery than finding and fixing it during requirements analysis or design [10]. It is therefore important that problems are found as early as possible in the system lifecycle. Adopting an incremental development approach enables testing to be performed in each iteration. This in turn means that the software quality can be continuously and quantitatively measured throughout the project.

5.6 Manage Change[14]

One of the big challenges when developing complex software intensive systems is to manage a large number of developers divided into several teams, working at the same time on several releases of project deliverables [48]. Without good guidance, this process may result in chaos. Having a formalized way of managing change to project artifacts addresses some of this complexity. It also enables metrics to be extracted from projects regarding change statistics, which then can be used for objective project status assessments.

6 The Proposed Approach

The strategy employed in this work was to extend the requirements discipline of RUP to better support software product line development. An analysis of RUP revealed that it provides little or no support for managing (or modeling) variability among members of a product family. This is unfortunate, however not surprising since the scope of RUP is a single software development project, and focus is on new development, rather than where coordination with other projects and maintenance.

One approach to address the problem of lack of commonality and variability analysis in RUP would be to transform RUP into a feature driven approach. This could be accomplished by replacing the RUP use case model with a FODA feature model. However, since use cases have such a central role in RUP, such a change would make it hard to even recognize the unified process in the result. This would in turn, lead to problems for organizations applying the resulting process. Examples of such problems could be increased training costs of new personnel and problems capturing new market segments, since features do not provide strong support for exploring new or poorly understood system characteristics [14]. Instead, the approach adopted in this work was to investigate how use case modeling could be extended to better support commonality and variability analysis.

Analysis of a number of use case models, revealed four types of variants that can exist in product family use case models as we described in *Paper I* and *Paper II*:

- The first type of variability regards the set of included use case in each product within a family.
- The second type of variability regards the set of included use case scenarios within each of these use cases.

---

[14] "Manage Change" in RUP refers to having control over changes, not to rapidly respond to changes; which is the main focus in agile software development [16]. This lack of agility is one of the most common critics of RUP, since it makes RUP unsuitable for small fast paced project [29].

- The third type regards the set of included steps within each of these use case scenarios.
- The fourth and final type of variability regards cross-cutting aspects that can affect several use cases on several levels. For example the existence of different sets of use case actors in different products.

6.1 Related Work

The UML use case meta-model provides poor support for variability modeling [74]. A number of suggestions on how to address this issue have been discussed in the literature (see Table 1 for an overview). These approaches can be divided into four main categories:

1. Approaches that structure use cases according to a feature model, and model variants in the feature model (see [28,27]).
2. Approaches that extend UML use case diagrams with variability constructs (see [27,74,41,56]).
3. Approaches that add variability mechanisms to textual use case specifications (see [39,28,23,27]).
4. Approaches that combine two or more of the different types of variability mechanisms described above.

We do, however, see a number of problems with existing approaches to product line use case modeling:

- When attempting to model variability in UML use case diagrams, diagrams tend to get cluttered to a degree where it is impossible to get an overview of the variants within (a non-trivial) product family. It is furthermore not enough to only manage variability among whole use cases (see discussion on types of variability in section 6).
- Existing approaches to manage variability within textual use case specifications do not have any means to provide a good overview of all variants within a family.
- Most existing approaches lack strong mechanisms to trace variant use case behavior to the system design.
- Most existing approaches allow *Free Selection*[15] among use cases and variants during product instantiation of the product line use case model. Adopting such an approach, instead of maintaining (and enforcing) a common system family model, is a major maintenance concern when working on extremely long lived systems. Copying documents and removing variant information is not good from this perspective, since information is being duplicated.

---

[15] Free selection means allowing single system requirements engineers to browse a product line model and simply copy requirements from the family model and pasting it into a single system model [52].

Table 1: An overview of variability mechanisms used in other published software product line use case modeling approaches.

	Variability mechanism:			
Approach	**Use case**	**Scenario**	**Step**	**Cross-cutting**
Jacobson et al. [39] (RSEB)	Using the generalization and extend relationships in UML use case diagrams by using a different use case stereotype icon for abstract use cases.	N/A[16]	N/A[16]	Only within a single use case specification using textual parameters.
Griss et al. [28] (FeatuRSEB)	Using a feature model that is linked to the use case model.	N/A[16]	N/A[16]	Only within a single use case specification using RSEB parameters.
Fantechi et al. [23] (PLUC)	N/A	N/A	N/A	Only within a single use case specification using the tags “Alternative”, “Optional” and “Parametric”.
Gomaa [27] (PLUS)	Using UML stereotypes in use case diagrams (“kernel”, “optional” or “alternative” use cases) and by modeling use cases packages as features in a feature model.	N/A[16]	N/A[16]	Only within a single use case specification using a section describing all variation points according to a variation point template.
van der Maβen & Lichter [74]	By extending the UML use case meta-model with the relations “Option” and “Alternative”.	N/A[16]	N/A[16]	N/A

---

[16] Could be managed by describing variant scenarios and variant steps as separate use cases which extends the original use case. This strategy is however likely to fragment the use case model resulting in too many and too small use cases when applied on a product line of non-trivial systems (See also [27] for further discussion of this issue).

John & Munthig [41]	Using UML stereotypes in use case diagrams ("variant"), and marking sections of diagrams as optional.	Using XML-like tags to mark scenarios as optional or alternatives.	Using XML-like tags to mark steps as optional or alternatives.	N/A
Moon et al. [56] (DREAM)	Using UML stereotypes in use case diagrams ("common" and "optional" use cases)	N/A[16]	N/A[16]	N/A

6.2 Marrying Use Case Modeling with Feature Modeling

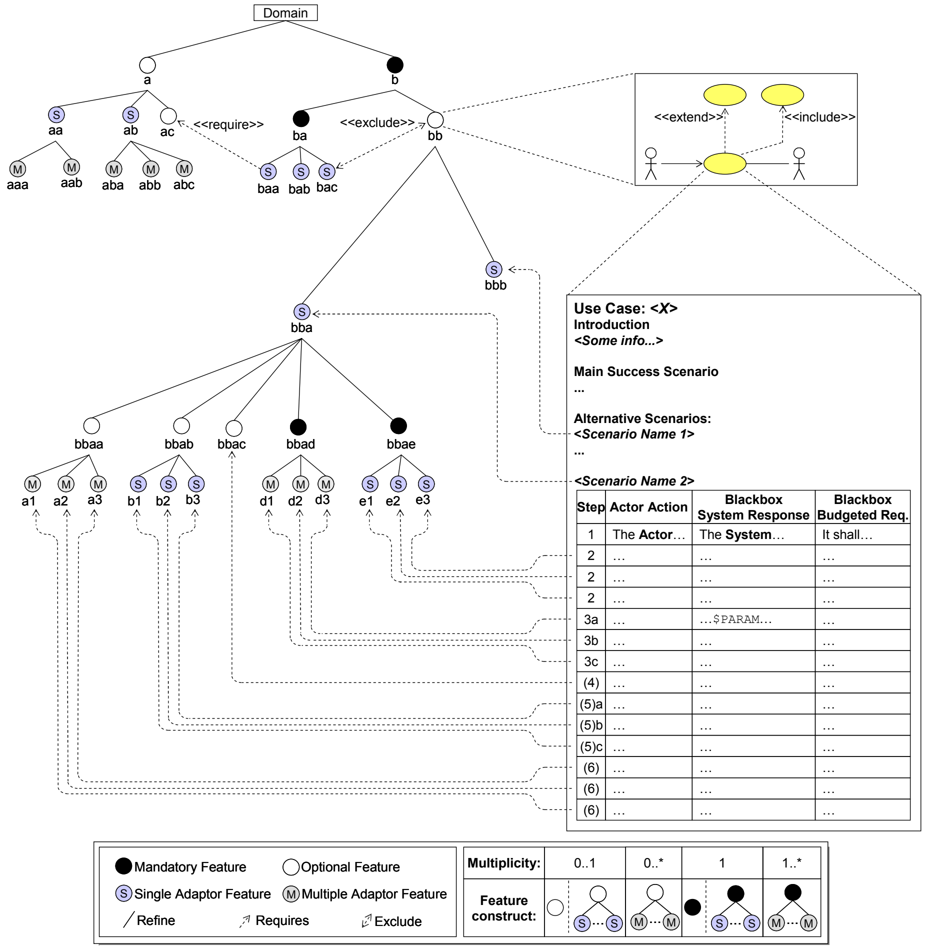
The approach for managing variability in use case models, presented in this thesis (see *Paper I* and *Paper II*), is based on the work by Griss et al., on FeatuRSEB[17] [28]. Like Griss et al. we argue that feature models are better suited for domain modeling than for example UML use case diagrams. A feature model should therefore be used as the high level view of a product family. However, in the proposed approach, the primary purpose of the feature model is not to take "center stage", but rather to be a tool for visualizing variants in our abstract product family use case models.

We use a feature model as a tool for structuring and instantiating our abstract family models into concrete product use case models for each system built within the family. We accomplish this by relating use cases, use case scenarios and use case scenario steps to features of appropriate types in a feature model as illustrated in Fig. 19 (see *Paper II*). We then select among the variants in the family model by selecting features from the feature model. To manage cross-cutting aspects, textual parameters as described by Mannion et al. in [53], are used. These parameters, which can be used anywhere in use case specifications, are linked to and visualized in the feature model as well. We also maintain use case realizations [48] and change cases [19] as part of this product family model. We utilize use case realizations to trace variant use case behavior to the system design (see Fig. 10), and change cases to mark proposed however not yet accepted functionality in a domain (see section 4.2).

Our approach is similar to Gomaa's approach [27]. Gomaa proposed to model features as use case packages. We extended this idea, saying that possibly a whole set of features compose a use case package. This has the advantage of enabling us to also visualize variants within use case specifications using a feature model. This means that a feature model provides a total overview of all variants that exist within a product family. A set of included features directly correspond to a specific set of included (concrete) use cases for a specific product within a family.

---

[17] In FeatuRSEB [28] a feature model is added to the 4+1 view model (see Fig. 18) adopted by Jacobson et al. in RSEB [39]. The feature model in FeatuRSEB takes "center stage" and provides a high-level view of the domain architecture and the reusable assets in the product family.



Use Case: <*X*>
Introduction
<Some info...>

Main Success Scenario
...

Alternative Scenarios:
<Scenario Name 1>
...

<Scenario Name 2>

Step	Actor Action	Blackbox System Response	Blackbox Budgeted Req.
1	The **Actor**…	The **System**…	It shall…
2	…	…	…
2	…	…	…
2	…	…	…
3a	…	…$PARAM…	…
3b	…	…	…
3c	…	…	…
(4)	…	…	…
(5)a	…	…	…
(5)b	…	…	…
(5)c	…	…	…
(6)	…	…	…
(6)	…	…	…
(6)	…	…	…

Fig. 19: An example of the relationship between features and use cases.

6.3 Product Instantiation

By marrying feature modeling with use case modeling, we have provided means to maintain a common and complete use case model for a whole product family. This means that product instantiation of the model is basically done by adding any new requirements to the model (which is likely to require new features to be added to the feature model as well) and then using the feature model to choose among its variants (see Fig. 20). New requirements are modeled as change cases to provide an overview the current delta in the model and to provide stronger support for change impact analysis [11]. A product use case model is generated by applying a filter to the domain model sorting out features not included in the current system. This will result in three types of reports: A "*Use Case Model Survey"* including all use cases (and possible change cases) for the product, and "*Use Case Specifications"* and "*Use Case Realizations*" for all use cases in that survey (see *Paper III* for tool support).

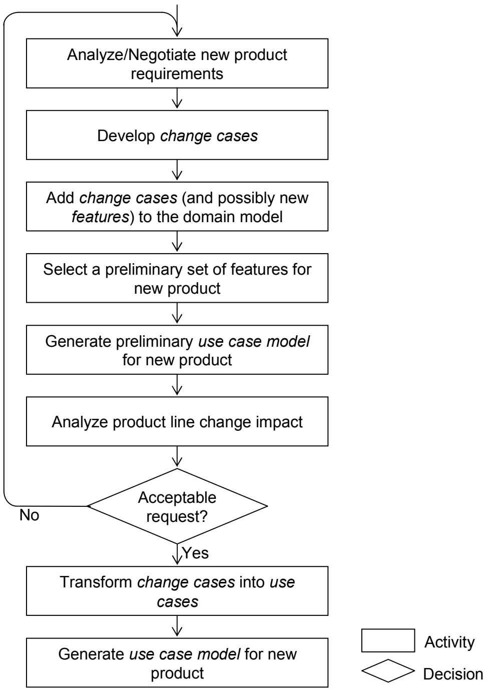
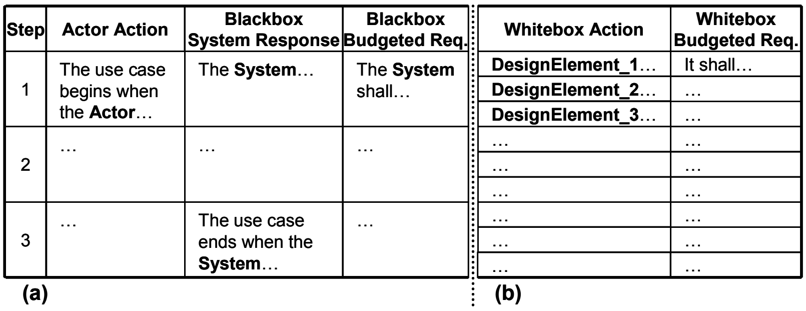

Analyze/Negotiate new product requirements
Develop change cases
Add change cases (and possibly new features) to the domain model
Select a preliminary set of features for new product
Generate preliminary use case model for new product
Analyze product line change impact
Acceptable request?
No
Yes
Transform change cases into use cases
Generate use case model for new product
Activity
Decision

Fig. 20: Adding a new product to a product line model.

6.4 A Note on Notations Used

As we described in *Paper I* and *Paper II*, we have chosen a tabular natural language description of use case scenarios and use case realizations in the proposed approach (see Fig. 21). The main motivation for this was that the industrial partner works in the embedded systems domain. This increases the number and diversity of stakeholders interested in the resulting models, including for example systems- and electrical engineering. This makes UML unsuitable for the purpose, since we believe its learning threshold is too high for wide-spread use within the organization. These natural language descriptions can however be supplemented with UML diagrams to improve understandability and precision/formality as needed.



Step	**Actor Action**	**Blackbox System Response**	**Blackbox Budgeted Req.**
1	The use case begins when the **Actor**…	The **System**…	The **System** shall…
2	…	…	…
3	…	The use case ends when the **System**…	…

(a)

Whitebox Action	**Whitebox Budgeted Req.**
DesignElement_1…	It shall…
DesignElement_2…	…
DesignElement_3…	…
…	…
…	…
…	…
…	…
…	…
…	…

(b)

Fig. 21: The (a) Blackbox flow of events used for describing use case scenarios, and (b) the Whitebox flow of events used for describing use case realizations.

6.5 Tool Support

The main tools used to support the proposed approach are the commercial requirements management tool Telelogic DOORS and the commercial UML modeling tool IBM-Rational Rose. Both tools are widely used and accepted in industry. Telelogic DOORS is utilized to manage these system family use case models and IBM-Rational Rose is used for drawing feature graphs and UML diagrams. Appropriate reports are generated from DOORS as MS Word documents, as shown in Fig. 22. A number of extensions to these tools were also developed to better support the proposed approach (see *Paper III* for details).

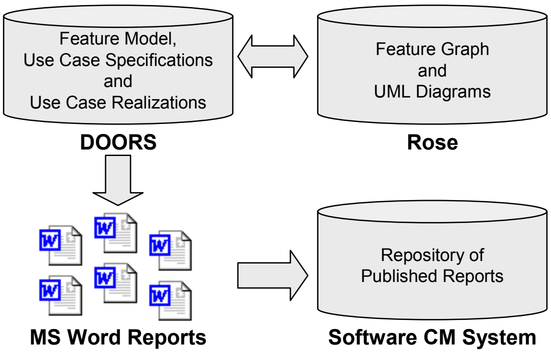


Fig. 22: An overview of the PLUSS toolkit.

7 Summary of Contributions

We have developed a simple extension to use case modeling that enable a common use case model to be developed and maintained for a whole family of products. The following sections will briefly summarize the appended papers which describe our contributions to this area of research.

7.1 Paper I – Marrying Features and Use Cases

Paper I outlines the proposed approach in terms of marrying use case modeling with feature modeling. A two-layer product family model is proposed in which concrete product use case models are derived from an abstract product family use case model.

Paper I also proposes the idea of introducing domain modeling and requirements reuse as part of the systems engineering process to provide stronger support for embedded software product line development. An approach for this, based on the RUP SE [63] “Use case flowdown“-activity, is also discussed in *Paper I* (see section 8.3 for future work in this area).

7.2 Paper II – The PLUSS Approach

Paper II extends the proposal of *Paper I* by providing means to develop and maintain a common and complete use case model for a whole product family. *Paper II* thereby removes the need to allow free selection (see footnote on page 20) in the model, which turn is likely to ease maintenance of the resulting product use case models.

Furthermore, a meta-model is presented which also includes use case realizations in the family model. The approach thereby provides strong means for tracing variant use case behavior to the system design.

Paper II also describes an extension to FODA feature models that enables modeling of “at-least-one-out-of-many”-selections. This extension was required to be able to capture all variants that can exist in use case models. Together with this extension, a new feature modeling notations was also proposed.

Finally, an industrial case study is presented where the proposed approach was applied and evaluated in the target domain.

7.3 Paper III – The PLUSS Toolkit

Paper III describes how commercial tools can be adapted and utilized to support the proposed product line use case modeling approach. A toolkit is presented, which extends the commercial UML modeling tool IBM-Rational Rose, and the commercial requirements management tool Telelogic DOORS, to better support the proposed product line use case modeling approach.

The basic idea presented in *Paper III* is to add the semantics of feature models to the heading outline of a natural language specification (in our case a *Use case model survey*). This enables sections of a specification to be included or not by a specific product in a product family, by selecting or deselecting features from a feature model.

8 Ongoing and Future Work

8.1 Test of Research Hypothesis

The study presented in *Paper II* indicates that adopting our software product line modeling approach enables more efficient development of systems in the target domain. However, how software product line development affects the maintainability of these systems is yet to be investigated. Even though it intuitively seems like a common platform would ease maintenance of systems by reducing the total amount of source code, other factors might have a negative influence. One example of such an influencing factor could be increased code complexity due to implemented variability mechanism in the platform. Another example could be the need for a more heavyweight process for change impact analysis and release management compared to single system development.

8.2 Further Development and Evaluation of the Proposed Approach

Further experience using the approach at Land Systems Hägglunds has shown the concept of local and global use case parameters (see *Paper I* and *Paper II*) is not as intuitive as initially indicated. The use of parameters has therefore been modified to only have one type. These new parameters are defined on an appropriate level in a feature model (on the same level as, or above, the use case(s) using them), and instead have scope rules similar to variable names in an imperative programming language. These new parameters can either be single-valued or multi-valued as illustrated by “PARAM_1” and “PARAM_2” in Fig. 23.

A follow-up study, as discussed in section 2, is planned to investigate if the initial positive results applying the proposed approach reported in *Paper II*, are still valid when being applied by a larger set of projects throughout the organization.

The study presented in *Paper II* indicated that the proposed approach form a good basis for early cost estimates. This area of application of the approach could be further developed. By attaching metrics to use cases and change cases in the model, powerful and highly automated cost estimates could be implemented in the presented toolkit. Examples of such metrics could be cost of development and integration of use cases in historical projects, and use case point [73] style metrics that could be added to change cases.

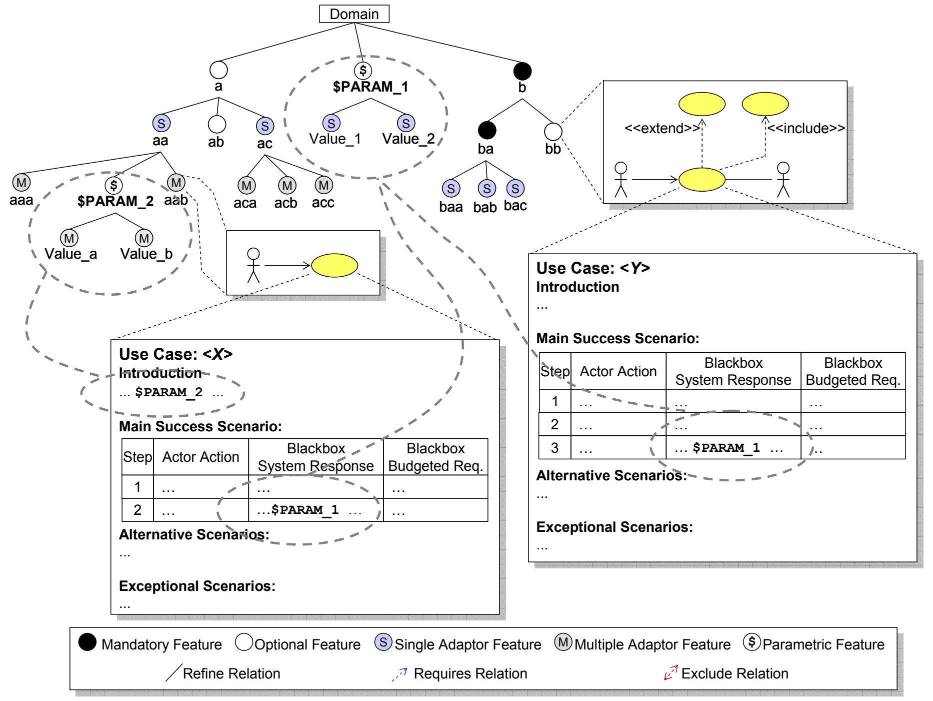


Fig. 23: Examples of the new parametric feature type.

8.3 Reuse of Systems Engineering Specifications

For successful embedded software product line development, we believe it is important that product line concepts such as domain modeling are also introduced into the systems engineering process (see Fig. 2). The reason for this is that embedded software requirements are for the most part not posed by customers or end users, but by systems engineering and the systems architecture (see Fig. 24). We have therefore developed a use case driven systems engineering method [20,21], that can be applied in accordance with the proposed product line use case modeling approach. Our assumption is that this will lead to systematic reuse of systems engineering specifications and thereby also ease the organizations' embedded software product line development. A major part of the reminder of this project will be dedicated to investigation of this area.

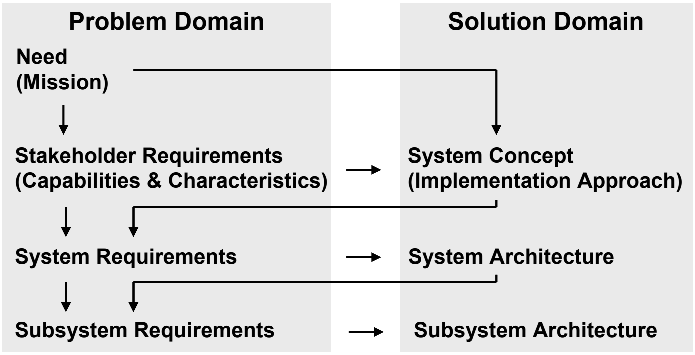


Fig. 24: System vs. subsystem (software) requirements.

8.4 Managing Variants in Design Specifications

An important supplement to the presented approach is a systematic methodology to handle variants in design specifications. One such variant example is if a single use case has different realizations in different products in a product line. At the first glance this might seem unnecessary since the basic idea of software product line development is to have a common architecture for all products in a family. This would in turn imply that if use cases are the same, so would their realizations. However, experience has shown situations where this is not the case. One example is if the architecture is very modular and enables different implementations of the same function (for example if a high-end vs. low-end product choice is possible).

We believe that similar techniques as those presented in this work for managing variability in use cases, can also be applied to other types of specifications. The idea of adding the semantics of feature models to specification heading outlines is likely to be applicable to basically any type of specification provided adequate tool support is available. This might however require a more expressive feature modeling notation than the one used in this work.

References

1. Adolph S., Bramble P., Cockburn A., Pols A.: Patterns for Effective Use Cases, Addison-Wesley (2003)
2. Ambler S., Nalbone J., Vizdos M.: The Enterprise Unified Process – Extending the Rational Unified Process, Prentice Hall PTR (2005)
3. Ambler S.: The Agile Unified Process v0.9, Available at: http://www.ambysoft.com/unifiedprocess/agileUP.html (January 2006)
4. America P., Obbink H., Muller J., van Ommering R.: COPA: A Component-Oriented Platform Architecting Method for Families of Software Intensive Electronic Products, Proceedings of the First Conference on Software Product Line Engineering, (2000)

5. Atkinson C. et al.: Component-based Product Line Engineering with UML, Addison-Wesley (2002)
6. Bass L., Clements P., Kazman R.: Software Architecture in Practice (1999) Addison-Wesley
7. Bittner K., Spence I: Use Case Modeling, Addison-Wesley (2003)
8. Boehm B.: A Spiral Model of Software Development and Enhancement, IEEE Computer (May 1988) 61-72
9. Boehm B.: Anchoring the Software Process, IEEE Software (July 1996) 73-82
10. Boehm B., Basili V.: Software Defect Reduction Top 10 List, IEEE Computer (January 2001) 135-137
11. Bohner S., Arnold R.: Software Change Impact Analysis, IEEE Computer Society Press, Los Alamitos , CA, US (1996)
12. Bosch J.: Design & Use of Software Architectures, Addison-Wesley (2000)
13. Brownsword L., Clements P.: A Case Study in Successful Product Line Development, CMU/SEI-96-TR-016), Pittsburgh, PA: Carnegie Mellon University, Software Engineering Institute (1996)
14. Chastek G., Donohoe P., Kang K.: Product Line Analysis: A Practical Introduction, Technical Report CMU/SEI-2001-TR-001, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (2001)
15. Clements P., Northrop L.: Software Product Lines, Practices and Patterns, Addison-Wesley (2001)
16. Cockburn A.: Agile Software Development, Addison-Wesley (2002)
17. Cusumano M.: The Software Factory: A Historical Interpretation, IEEE Software, vol. 6, no. 2 (March 1989) 23-30
18. Czarnecki K., Eisenecker U.: Generative Programming – Methods, Tools, and Applications, Addison-Wesley (2004)
19. Ecklund E., Delcambre L., Freiling M.: Change Cases - Use Cases that Identify Future Requirements, Proceedings of OOPSLA 96, San Jose, Ca (October 6-10 1996) 342-358
20. Eriksson M., Borg K., Börstler J.: The FAR Approach - Functional Analysis/Allocation and Requirements Flowdown Using Use Case Realizations, Submitted to the Sixteenth Annual International Symposium of the International Council on Systems Engineering (2006) Available on request
21. Eriksson M., Börstler J., Borg K.: Performing Functional Analysis/Allocation and Requirements Flowdown Using Use Case Realizations – An Empirical Evaluation, Submitted to the Sixteenth Annual International Symposium of the International Council on Systems Engineering (2006) Available on request
22. Ezran M., Morisio M., Tully, C: Practical Software Reuse, Springer (2002)
23. Fantechi A., Gnesi S., Lambi G., Nesti E.: A Methodology for the Derivation and Verification of Use Cases for Product Lines, Proceedings of the International Conference on Software Product Lines, Lecture Notes in Computer Science, Vol. 3154, Springer-Verlag (2004) 255-265
24. Faulk R.: Software Requirements: A Tutorial, Software, Requirements Engineering, IEEE Computer Society Press (1997)128-149
25. Frakes W, Kang K.: Software Reuse Research: Status and Future, IEEE Transactions on Software Engineering, vol. 31, no. 7 (July 2005) 529-536
26. Gibbs W.: Software's Chronic Crisis, Scientific American 271, 3 (1994) 72-81
27. Gomaa H.: Designing Software Product Lines with UML – From Use Cases to Pattern-Based Software Architectures, Addison-Wesley (2004)
28. Griss M., Favaro J., d'Alessandro M.: Integrating Feature Modeling with the RSEB, Proceedings of the Fifth International Conference on Software Reuse, Vancouver, BC, Canada (June 2-5 1998) 76-85

29. Hirsch M.: Making RUP Agile, Proceedings of OOPSLA 02, Seattle, Ca (November 2002).
30. IEEE 1220-1998, Standard for Application and Management of the Systems Engineering Process, ISBN 0-7381-1543-6 (January 1999)
31. International Council on Systems Engineering (31), Systems Engineering Handbook Version 2a, 31-TP-2003-016-02 (June 2004)
32. ISO 9001:2000, Quality Management Systems – Requirements
33. ISO/IEC 15504-1, Information technology – Process assessment – Part 1: Concepts and vocabulary
34. ISO/IEC 15504-2, Software Engineering – Process assessment – Part 2: Performing an assessment
35. ISO/IEC 15504-3, Information technology – Process assessment – Part 3: Guidance on performing an assessment
36. ISO/IEC 15504-4, Information technology – Process assessment – Part 4: Guidance on use for process improvement and process capability determination
37. ISO/IEC 15504-5, Information technology – Process assessment – Part 5: An exemplar Process Assessment Model
38. Jacobson I.: Object Oriented Development in an Industrial Environment, Proceedings of OOPSLA'87, Orlando, FL (4-8 October 1987) 183-191
39. Jacobson I., Griss M., Jonsson P.: Software Reuse – Architecture, Process and Organization for Business success, Addison-Wesley (1997)
40. Jacobson I., Booch G., Rumbaugh J.:The Unified Software Development Process, Addison-Wesley (1999)
41. John I., Muthig D.: Product Line Modeling with Generic Use Cases, SPLC-2 Workshop on Techniques for Exploiting Commonality Through Variability Management, Second Software Product Line Conference, San Diego, USA (August 2002)
42. Kang K. Cohen S., Hess J., Novak W., Peterson A.: Feature Oriented Domain Analysis (FODA) Feasibility Study, Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (1990)
43. Kang K., Kim S., Shin E., Huh M.: Form: A Feature-Oriented Reuse Method with Domain Specific Reference Architectures, Annals of Software Engineering, vol. 5 (1998) 143-168
44. Kitchenham B., Pickard L., Pfleeger S.: Case Studies for Method and Tool Evaluation, IEEE Software, Vol. 12 Nr. 45 (1995) 52-62
45. Kitchenham B., Pickard L.: Evaluating Software Eng. Methods and Tools Part 10: Designing and Running a Quantitative Case Study, Software Engineering Notes, vol. 23, no. 3 (May 1998) 20-22
46. Kruchten P.: The 4+1 Model View of Architecture, IEEE Software (November 1995) 42-50.
47. Kruchten P.: The Rational Unified Process: An Introduction, Addison-Wesley (1998)
48. Kruchten P.: The Rational Unified Process: An Introduction, Second Edition, Addison-Wesley (2000)
49. Lauesen S.: Software Requirements – Styles and Techniques, Addison-Wesley (2002)
50. Lethbridge T., Sim S., Singer J.: Studying Software Engineers: Data Collection Techniques for Software Field Studies, Empirical Software Engineering Journal, vol. 10 (2005) 311-341
51. Linqvist M., Johansson G.: Processutvärdering vid Alvis Hägglunds AB i Örnsköldsvik, Uppdragsrapport 03/31, Internal Report (October 20, 2003)
52. Mannion M., Lewis O., Kaindl H., Wheadon J.: Reusing Single System Requirements from Application Family Requirements, Proceedings of the International Conference on Software Engineering ICSE'99, Los Angeles, CA, USA, (1999) 453-462

53. Mannion M., Lewis O., Kaindl H., Montroni G., Wheadon J.: Representing Requirements on Generic Software in an Application Family Model, Proceedings of the International Conference on Software Reuse ICSR-6 (2000) 153-196
54. Matinlassi M.: Comparison of Software Product Line Architecture Design Methods: COPA, FAST, FORM, KobrA and QADA, Proceedings of the 26th International Conference on Software Engineering ICSE'04 (2004) 127-136
55. Mili A., Mili R., Mittermeir R.: A Survey of Reuse Libraries, Annals of Software Engineering, vol. 5 (1998) 349-414
56. Moon M., Yeom K., Chae H.: An approach to Developing Domain Requirements as a Core Asset Based on Commonality and Variability Analysis in a Product Line, IEEE Transactions on Software Engineering, vol. 31, no. 7 (July 2005) 551-569
57. Niemelä E.: QADA – Quality-driven Architecture Design and Architecture Analysis, Available at: http://www.vtt.fi/qada/, (January 2006)
58. Northrop L.: SEI's Software Product Line Tenets, IEEE Software (July/August 2002) 32-40
59. Object Management Group: The Unified Modeling Language, Version 2.0, Available at: http://www.uml.org (2005)
60. Palmer J.: Traceability, In Software Requirements Engineering, Ed. Thayer H. and Dorfman M.), IEEE Computer Society Press, Los Alamitos, CA (1996)
61. Parnas D.: On the design and development of product families, IEEE Transactions on Software Engineering, vol.2, no. 1 (1976) 1-9
62. Potts C.: Software-Engineering Research Revisited, IEEE Software (September 1993) 19-28.
63. Rational Software: The Rational Unified Process for Systems Engineering Whitepaper, Ver. 1.1, Available at: http://www.rational.com/media/whitepapers/TP165.pdf (2003)
64. Robak S.: Feature Modeling Notations for System Families, Proceeding of the International Workshop on Software Variability Management, Co-located with ICSE'03 in Portland, Oregon (May 2003) 58-62
65. Royce W.: Managing the Development of Large Software Systems: Concepts and Techniques, Proceedings of IEEE Wescon (August 1970) 1-9
66. Seaman C.: Qualitative Methods in Empirical Studies of Software Engineering, IEEE Transactions on Software Engineering (July/August 1999) 557-572
67. Sommerville I., Sawyer, P.: Requirements Engineering, A good practices guide (1997) Wiley
68. Svahnberg, M., Gurp, J., Bosch, J.: On the Notation of Variability in Software Product Lines, Proceedings of the Working IEEE/IFIP Conference on Software Architecture (2001) 45-55
69. Szyperski C., Gruntz D., Murer S.: Component Software — Beyond Object-Oriented Programming, Second Edition, Addison-Wesely (2002)
70. Thiel S., & Hein A.: Systematic Integration of Variability into Product Line Architecture Design, Proceedings of the Second International Conference on Software Product Lines (2002) 130-153
71. Trigaux J., Haymans P.: Software Product Lines: State of the Art, Technical Report EPH3310300R0462/215315, PLENTY project, Available at: http://www.info.fundp.ac.be/~jtr/PLENTY/Files/productline0309.pdf (September 2003)
72. van der Linden F.: Engineering Software Architectures, Processes and Platforms for System Families - ESAPS Overview, Proceedings of the Second International Conference on Software Product Lines, San Diego, CA, USA (August 2002) 383-397
73. Vinsen K., Jaimieson D., Callender G.: Use Case Estimation – The Devil is in the Detail, Proceedings of the 12th IEEE International Requirements Engineering Conference (2004)

74. von der Maßen T., Lichter H.: Modeling Variability by UML Use Case Diagrams, Proceedings of the International Workshop on Requirements Engineering for Product Lines, (2002) 19-25
75. Weiss D., Lai C., Tau R.: Software product-line engineering: a family-based software development process, Addison-Wesley (1999)