Mikko Raatikainen

A Research Instrument for an Empirical Study of Software Product Families

Master's thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Technology

Espoo, January 15th, 2003

HELSINKI UNIVERSITY OF TECHNOLOGY

ABSTRACT OF THE MASTER'S THESIS

Author :	Mikko Raatikainen	
Title:	A Research Instrument for an Empirical Study of Software Product Families	
Date:	January 15th, 2003	Number of pages: 95+33
Department: Professorship:	Department of Computer Science and Engineering T-86 Information Technology	
Supervisor: Instructor:	Prof. Timo Soininen Prof. Timo Soininen	

The software product family approach for software development is a promising way to achieve significant reuse and efficiency in a software development organization that produces a large number of varying products that yet embody also commonalities. In this approach, products are developed on a basis of a predefined software product family architecture and shared assets.

In this thesis, a research instrument, an initial theory and interview questions, is developed for a multiple case study for studying state-of-the-practice of software product families. It concentrates on the architecture and reusable assets in a family, but it also covers to some extend the process, organization and business issues relevant for understanding software product families. The instrument was used to carry out one case study to evaluate its utility. The studied company had succeeded excellently in developing a software product family, in which products are developed efficiently by configuring. The research instrument was successful and the case study results increase the credibility of promises of the software product family approach.

Keywords:	case study, interview, software product family, software product line

TEKNILLINEN KORKEAKOULU DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Mikko Raatikainen	
Työn nimi:	Tutkimusväline empiiriselle tutkimukselle ohjelmistotuoteperheistä	
Päiväys:	15.1.2003	Sivumäärä: 95+33
Osasto: Professuuri:	Tietotekniikan osasto T-86 Tietotekniikka	
Valvoja: Ohjaaja:	Prof. Timo Soininen Prof. Timo Soininen	

Ohjelmistotuoteperhe lähestysmistapa on lupaava keino saavuttaa merkittäviä uudelleenkäyttö- ja tehokkuusetuja ohjelmistokehitysorganisaatiossa, joka tuottaa varioituvia tuotteita, jotka sisältävät myös yhtäläisyyksiä. Ohjelmistotuoteperhe lähestysmistavassa ohjelmistot kehitetaan ennalta suunniteltua ohjelmistoarkkitehtuuria ja ohjelman osia käyttäen.

Tässä työssä on kehitetty tutkimusväline – alustava teoria ja haastattelukysymykset – ohjelmistotuoteperheiden nykytilan tapaustutkimukselle teollisuusyrityksissä, keskittyen arkkitehtuuriin ja uudelleenkäytettäviin osiin teknisestä näkökulmasta, joskin sisältäen myös liiketoiminnallisen, prosessi ja organisatorisen näkökulman. Tutkimusvälinettä on käytetty yhdessä tapaustutkimuksessa sen soveltuvuuden arvioimiseksi. Tutkittu yritys oli onnistunut erinoimaisesti ohjelmistotuoteperheen kehittämisessä, jossa tuotteet kehitettiin tehokkaasti konfiguroimalla. Tutkimusväline oli onnistunut ja tutkimuksen tulokset lisäävät ohjelmistotuoteperheisiin liitettyjen etujen uskottavuutta.

Avainsanat:	haastattelu, ohjelmistotuotelinja, ohjelmistotuoteperhe, tapaustutkimus

Acknowledgements

This work was conducted in the Software Architectures for Configurable Ubiquitous Systems (Sarcous) project of Product Data Management Group (PDMG) in Software Business and Engineering (SoberIT) Laboratory of Helsinki University of Technology. I would like to thank all people in SoberIT who have helped during this thesis.

I also am grateful to my instructor and supervisor Professor Timo Soininen.

I would like to thank Tomi Männistö, Antti Mattila and Professor Reijo “Shosta” Sulonen who also had their contribution for the research conducted in this thesis.

Finally, I am grateful to the National Technology Agency (Tekes) and our industrial partners of the financial support for this thesis.

Espoo, January 2003

Mikko Raatikainen

Table of Contents

1 Introduction

1.1 Background

The importance of software in products is increasing (Kitchenham and Pfleeger 1996). Software has entered new domains and become an integrated part of everyday life (Bosch 2000a). Systems are increasingly dependent on software and software may even be the differentiating factor between products. A trend is to increase flexibility and customization of products that is often achieved by altering software. At the same time, the size and complexity of software are increasing (Card and Comer 1994).

Software development from the scratch is complex, expensive, and laborious. One promising way to develop more efficiently products that have common properties but also differences is a software product family approach (Bosch 2000a; Clements and Northrop 2001; Knauber et al. 2000). In the software product family approach it is assumed to be advantageous to analyze and take advantage of the common properties of a set of the products before analyzing each product on its own (Parnas 1976; Weiss and Lai 1999). According to (Bosch 2000a), *a software product family* consists of a software product family architecture, a set of reusable shared assets and a set of products based on the software product family architecture and the shared assets.

The origins of the software product family approach date back to the middle of 1970's (Weiss and Lai 1999) when David Lorge Parnas (Parnas 1976) was the first to define the concept. He also noted already then that some of the same software product family approach principles had been applied in software engineering for a while. Recently, Paul Clements and Linda Northrop (Northrop 2001) predicted that the software product family approach would be the "predominant software paradigm at the beginning of the new century". They also see that the history of programming can be viewed as an "upward spiral", in which reuse of software and applicability of building blocks in software increase and each decade has its own paradigm as follows:

- 1960's subroutines
- 1970's modules

- 1980’s objects
- 1990’s software components
- 2000’s software product families

The software product family approach differs from other reuse paradigms such as component based reuse, single system development with reuse or configurable architectures. The most essential characteristics of the software product family approach that differentiate it from other reuse approaches are (Bosch 2000a; Clements and Northrop 2001):

- The software product family approach is planned, enabled and even enforced opposite of e.g. opportunistic reuse approaches in which software is only made available for reuse
- A software product family includes shared assets other than a software architecture and software components
- Software product family is designed top-down and a software architecture centric instead of being only composed of components
- A software product family consists of simultaneous variants as well as older versions of products that have market potential

Although the idea of a software product family is not new, not until in a past few years software product families have become a subject for growing amount of research (McGregor et al. 2002). Many aspects of the software product family approach require deeper understanding. Similarly, there seems to be a need for tools and methods to manage software product families (Knauber and Succi 2002), but there is a lack of rigorous empirical descriptive results on details of the software product family approach to develop them. As noted in (Bosch 1999b) there is also little research done in the small and medium size industry. It is also noted in (Northrop 2001) that small and medium sized development organizations apply the software product family approach differently from large ones, but there are not details on what respect do they differ.

This work is a part of “**S**oftware **AR**chitectures for **CO**nfigurable **U**biquitous **S**ystems“ (Sarcous) project (Sarcous 2002). The objectives of the Sarcous-project are to study and formulate methods for managing software product families and reusable software com-

ponents on the basis of modeling them as configurable software product families. A software product family case study was decided to be conducted in several companies to elicit needs and prerequisites for the method in order for the method would be applicable in practice (Yin 1994).

1.2 Objectives

The following *research topics* that specify objectives for the software product family case study were formulated in the Sarcous-project:

1. The concept and utilization of software product families
2. The methods and tools used to capture information on software product families and the information that exists on software product families
3. The processes supporting the creation, utilization, management and maintenance of software product family information
4. Business aspects of software product families

The objectives of this work are:

> *To develop and initially validate a research instrument for the software product family case study.*

1.3 Method

In this section, the conducted case study is outlined first in general. The objectives of this work cover only parts of the whole case study and these parts are described second.

1.3.1 Software product family case study

In (Yin 1994), a case study research is divided into research design, preparing for data collection, collecting evidence, analyzing the evidence, and composing case study reports.

Research design

The research design of the software product family case study is a lightweight descriptive multiple-case study in companies, that produce software products or products containing software (Yin 1994) and the software is developed applying the software product family approach. The objectives were to elicit state-of–the-practice in industry on the issues specified in the research topics (Männistö et al. 2000).

For the case study, *an initial theory* describes the preliminary knowledge illustrating relevant theoretical issues (Yin 1994). The initial theory is based on synthesis of the most relevant previous research results on the software product family approach and influenced by experience (Soininen 2000; Tiihonen et al. 1998) on configurable mechanical and electronic products.

The companies for the case study were selected purposefully (Patton 1990). The goal was to find companies that are representative for the software product family approach state-of-the-practice in Finnish industry. In order to decrease the initial number of candidates, people who were familiar with Finnish industry were asked for companies that could be suitable for the study. Then, the product portfolios of the candidate companies were analyzed and the companies were asked for willingness to participate the case study. Five companies have been studied to date.

Data collection was decided to be conducted in each company in an interview, a validation session and document analysis. It was decided to give each company a possibility to read reports before publishing them, if they were based on data gathered in the company.

Preparing for data collection

The interview approach was decided to be a semi-structured interview. There is no exact definition for a semi-structured interview (Hirsjärvi and Hurme 2001) hence it was applied such that interview questions were developed before interviews and meant to be followed as strictly as would be meaningful in the interviews. The interview questions were developed to elicit the issues specified in the research topics. The initial theory

guided the development. In addition, similar and related empirical studies were studied for guidance on what kind of questions should be used or they could even provide questions that could be reused (Kitchenham and Pfleeger 2002a).

The interview questions were evaluated using two different techniques before using them in the case study. The first evaluation resembles a focus group study (Kitchenham and Pfleeger 2002b) and it was conducted with a researcher working in the same research project. She was naturally aware of the goals, intentions and terminology used in the questions, but she had not participated in any of the phases in their design. She had been working in the industry with software development before joining the research group. The technique used was to interview her and ask her to rephrase the interview questions in her own words in order to determine if the interview questions were understood as intended (Foddy 1993).

In the second evaluation, the interview questions were piloted with an employee of another research project (Foddy 1993; Kitchenham and Pfleeger 2002b). The responder had been working in the industry with software product families and therefore possessed experiences on them. The pilot study simulated the actual research settings and hence practices for the interviews were rehearsed concurrently. The results of the both evaluation processes were used to finalize the interview questions.

Finally, for the validation session, the used approach was to construct a slide presentation to guide discussion. The slide presentation included slides on preliminary results, clarifications for unclear issues and issues that were interesting or seemed potential targets for improvements in the company.

Collecting the evidence

The case study took one day in each of the participating companies, consisting of 4-6 hours interview and 2-3 hour validation session both held in their office premises. The interviews were conducted between November 2001 and March 2002. The feedback sessions were held in May and June 2002.

In the interviews, there were at least two researchers present depending on the company. One led the interview and asked the interview questions and one made notes. The roles were rotated in the middle of interviews. In the interviews that more than two researchers were present the rest followed the interview as a whole and asked questions on topics that seemed important and vague or felt unclear. Depending on the company there were two to four responders present though not all the time and simultaneously.

At the beginning of each interview, the background, objectives and terminology of the case study were described to responders. In order to gain trust from the responders they were assured to be handled confidentially such that she or he could not be identified. In addition, it was assured that someone in the company would have a possibility to read the reports in order to avoid releasing e.g. confidential information or details on the products.

The interview questions were shown to the responders using a data-projector in order to help responders with definitions, complex questions, and questions that included pictures. Some questions were skipped and some wordings were changed especially in the case of terminology. Sometimes the order of the questions was changed and additional questions that were not in the prepared questions were asked. The additional question were mainly clarification questions, questions that deepened the topic and summarizing comments in a question format to generalize and give responder a possibility to correct misinterpretations and misunderstandings.

In the validation session, at least partially the same responders were present. The prepared slide presentation was shown and clarifications were asked. The people present were able to correct misunderstandings.

The interviews and validation sessions were tape-recorded. Notes were made in the company and later during listening the tapes. The interviews were also transcribed in detail but the recordings from the validation session were not transcribed, because the researchers did most of the talking. In addition, documentation was analyzed. The documentation included technical documentation such as software architecture design documentation and non-technical documentation such as product brochures, marketing

material and www-pages. Some documentation could be taken with for further study after the interviews and some were analyzed in the premises of the company. Notes were made on the documentation.

Analyzing case study evidence

The data from each company were first analyzed separately. Analysis was based on a short discussion on the impressions of the case with other researchers immediately after the interviews, listening the tapes, written notes, documentation, and the written transcript. After the preliminary analysis, a validation session was held in that company. For the final analysis, data from the validation session were added.

The data were analyzed using ATLAS.ti application, which is a tool designed for qualitative data analysis. The used data were transcripts, notes, and documents, which were possible to be included in ATLAS.ti. The data that could not be analyzed in ATLAS.ti, such as paper documentation or unsupported file formats, were linked in order not to forget them and then examined during the analysis. The data were marked using the codes that initially based on the initial theory and was then extended during the analysis. These coded data segments, *quotations,* made easier to handle data. The codes had structure where their relationships or meanings were clarified. In addition, own remarks, *memos* and *comments*, were used to synthesize data. (Moilanen and Ruohonen 1994)

Composing the case study report

This work is the first report of the case study.

1.3.2 This thesis

In this thesis the research instrument for the case study was developed and documented and one case study was conducted. The development of the research instrument consisted of formulation of the initial theory and interview questions. Documenting the research instrument meant in practice writing down the initial theory, interview questions and links from the initial theory to the questions. The case study was used primary to evaluate the validity and reliability of the research instrument. The research instrument

was described earlier. In this section, the conducted case study is described in more depth.

Case

The company that was selected for this thesis was the first one that agreed with a schedule of the companies in the software product family case study. Consequently, the company was in practice selected randomly from the set of the participating companies.

The interview took about four hours, excluding breaks and about an hour long introduction to the company and products given by the employees of the company. Four employees were interviewed. The first responder, the software architect of the software product family, was present all the time. The second responder, a product manager, was present the first two sections. The third responder, manager of products to be delivered the customers, was present the latter half of the second section. The fourth responder, a software engineer responsible for designing and implementing a software product family, was present for the third section. This is shown in Figure 1

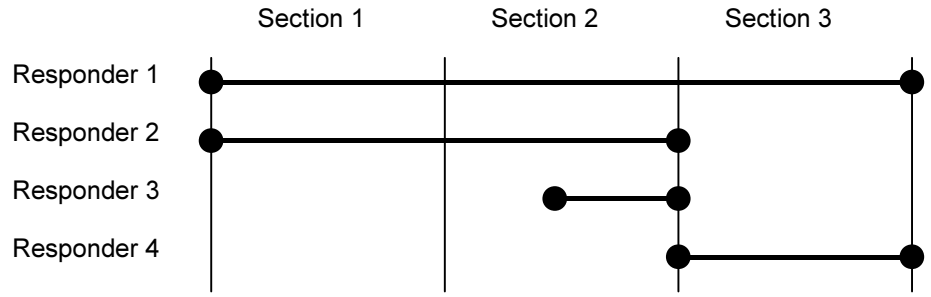

Figure 1 Responders present in the interview

Four researchers participated in the interview. In addition to the interview, technical documentation was analyzed later during a second visit to the company that took about two hours. The company gave also material that was then available also later in the analysis. The material included annual report, marketing CD-ROM and product brochures. In addition, the company's www-pages are available in the Internet and were analyzed.

1.4 Scope

The initial theory is based mostly on previous research results reported in the software product family approach related scientific literature. Closely related fields such as software reuse and component based software engineering were not reviewed thoroughly. The initial theory is presented as a synthesis of issues regarded relevant.

The developed research instrument is designed for the software product family case study, and therefore it is not validated to be necessary suitable for any arbitrary empirical research of software product families (Kitchenham and Pfleeger 2002a). This is because the research instrument is focused on software engineering perspective of software artifacts in software product families as outlined in the objectives of the software product family case study and the research design such as planned duration were taken account during the development.

The primary purpose of the case study in this thesis was to use and then validate the research instrument in practice. The case study in this work is done primary for that purpose, hence it lacks in-depth analysis of the results and comparison with other results on empirical studies.

1.5 Outline of the thesis

The thesis is organized as follows. In the Chapter 2 the software product family concept is briefly reviewed. The research instrument is described in Chapter 3. In Chapter 4 the case study is described. The case is compared to results in literature, key findings are summarized and validity, reliability, and issues related to construction of the research instrument are discussed in Chapter 5. In Chapter 6 conclusions of the thesis are drawn and future work is outlined. The interview questions follow as Appendix 1.

2 Software Product Family Concept

This chapter is organized as follows. A note on terminology on the software product family is made in Section 2.1. In Section 2.2 different software product family definitions, which seem to be the most used and widely accepted, are introduced and the definitions are then compared. A concept of a product population, closely related to the software product family concept, is described in section 2.3.

2.1 Terminology on software product families

There exists neither standard definition nor established unambiguous terminology for software product families (Di Nitto and Fuggetta 1996). Usually the term a software product line is used as a synonym for the term a software product family (Bosch 2001). According to (Linden 2002), the former is usually used in the US and the latter in Europe albeit even this is not permanently established and unambiguous. The term a software product family is used in this work.

2.2 Software product family definitions

David Lorge Parnas (1976) gives probably the first definition as he defines *a program family* as follows (Parnas 1976):

> "Program families are defined as sets of programs whose common properties are so extensive that it is advantageous to study the common properties of the programs before analyzing individual members".

David M. Weiss and Chi Tau Robert Lai (1999) wrote the first and widely referred book on the software product family approach and use the term *a product line* as follows (Weiss and Lai 1999):

> "[Product line is] a family of products designed to take advantage of their common aspects and predicted variabilities."

Paul Clements and Linda Northrop (2001) (Clements and Northrop 2001) and Jan Bosch (2000) (Bosch 2000a) wrote more recent and widely referred books on the software product family approach. These books are used extensively through this thesis. The following two definitions are from these books.

Clements and Northrop (Clements and Northrop 2001) define a *software product line* as follows:

> "A software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way".

Bosch (Bosch 2000a) uses the term *software product line* as follows:

> "A software product line consists of a product line architecture and a set of reusable components that are designed for incorporation into the product line architecture. In addition, the product line consists of the software products that are developed using the mentioned reusable assets"

2.2.1 Commonalities

Three commonalities between all given definitions are as follows:

- A set of products in a software product family
- Commonality between products
- A software product family has an influence on development of products

All definitions point out that there are products or a set of products in a software product family. The word "set" or usage of a plural form of the word "product" means in practice that there is more than one product in a software product family. Other relevant issues related to the set of products are that there is no upper limit for the number of products, the number of products is not fixed and the set of products is not prescribed.

All definitions contain a notion on commonality between products. The definitions given by Bosch and Clements and Northrop emphasize assets. The two other definitions regard commonality more abstractly using terms “common aspects” and “common properties”. These, and in addition the terms shared assets, core asset, reusable assets and even components seem to be used to refer the same thing (Northrop 2001).

Third similarity in the definitions is that the software product family approach has an influence on the development of products that are in the software product family. Clement and Northrop state this most clearly in the their definition as they require that products in a software product family have to be developed in a prescribed way. In addition, the other definitions imply that developing products in a software product family differs from developing a set of products from the scratch. In fact, Weiss and Lai (Weiss and Lai 1999) introduce the following four concepts that illustrate the idea clearly

- Potential family
- Semifamily
- Defined family
- Engineered family

A potential family is a set of products that are suspected to have sufficient amount of commonality reasonable to be studied in more detail. This commonality and variability are identified in *a semifamily*. In *a defined family* the commonality is analyzed and made explicit, but not realized. A product line is then finally *an engineered family*, in which commonality is made operational in processes, tools and resources to rapidly create products in a family.

2.2.2 Differences

The differences between definitions are

- Definition of Clements and Northrop emphasizes the needs of a particular market segment or mission.
- Definition of Clements and Northrop emphasizes that products have to be developed in a prescribed way.

- Role of software architecture

The needs of a particular market segment or mission implies that a set of products in a software product family is dealt with also from other viewpoints than technical product development, such as from the user or marketing point of view, as a logically set of product. All other definitions regard a software product family to be a logical set of products only from the software construction viewpoint.

The second difference and according to (Northrop 2002) an important issue in the definition is that products have to be developed in a prescribed way. Although all definitions have an influence on the development, Clements and Northrop require that a software product family includes a production plan, which describes how the assets are used and how products are built using these assets.

Finally, Bosch emphasizes in his definition a software product family architecture that is for Clements and Northrop one of the core assets (Clements and Northrop 2001) and the two other definition do not explicitly require it, but neither exclude it.

However, the commonalties between the definitions are more significant than the differences.

2.3 Product population

Rob van Ommering (2001) defines a *software product family* (Ommering 2001):

> "[Software product family is] a (small) set of products that have many commonalities and few differences".

The difference when compared to the above definitions is that in this definition it is explicitly pointed out that the products in the family have few differences. However, more important is that he uses also the term *a product population* that is defined as follows (Ommering 2001):

> "[A product population] is a set of products with many commonalities but also many differences".

The difference between a software product family and a product population is somewhat vague. A criterion, according to (Ommering 2001), is that a software architecture in products of a software product family is the same, whereas between products in a product population the software architecture is different. In addition to the varying software architecture, a product population is also more loosely defined. According to (Ommering 2000) it is not reasonable to define a parametric architecture that embodies all possible variants for a product population. Therefore, products in product populations are constructed by combining components to this unspecified or loosely specified software architecture. In fact, a product population could be implemented as a software product family, but the different term is used to emphasize larger diversity of products and different approach in construction (Ommering 2000).

Another aspect in these definitions is that van Ommering constrains the commonality and differences in a set of products that are in a software product family. He orders the above outlined concepts with respect to spectrum of product diversity using consumer electronic as example (Ommering 2000). The spectrum is presented in Figure 2. A single product, on the left in Figure 2, is a shrink-wrap mass product. Products in software product family and product populations include variability in some extend and diversity of the products increases to the right in Figure 2.

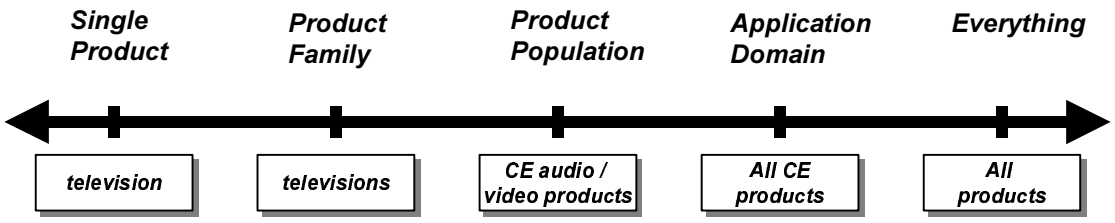

Figure 2 Spectrum of product diversity in consumer electronics (CE) products (Ommering 2000)

3 Research Instrument

In this chapter, the research instrument for the software product family case study is described. First the structure and terminology of the research instrument are described in Section 3.1. In Section 3.2 definitions of essential concepts and clarification of the used terminology are given. Then, the next four sections describe different concerns in a software product family and is organized as follows: business in Section 3.3, processes in Section 3.4, organization in Section 3.5 and product in Section 3.6.

3.1 Introduction

The research instrument for the case study consists of an initial theory and the interview questions to be asked from responders. The initial theory describes the preliminary knowledge illustrating relevant theoretical issues (Yin 1994) and it is organized as follows:

- The initial theory for the case study is decomposed to a business, process, organization, and product *concern*.
- For each concern there are identified several *aspects.*
- For each aspect there is a set of *factors.*
- For each factor there is a set of *interview questions.*

The division to concerns was done in order to take into account several points of view to a software product family. The aspects characterize the concerns from different angles. A synthesis of the literature on concerns and aspects that are relevant or specific for the research are presented in this chapter. The aspects are further decomposed to the factors from case study objectives point of view. The factors are outlined in the chapter and the interview questions, given in Appendix, measure the factors. An exception to this decomposition in the interview questions is that organization concern is handled under process concern. Figure 3 shows this decomposition to the concerns, aspects and factors that is outlined in the following sections.

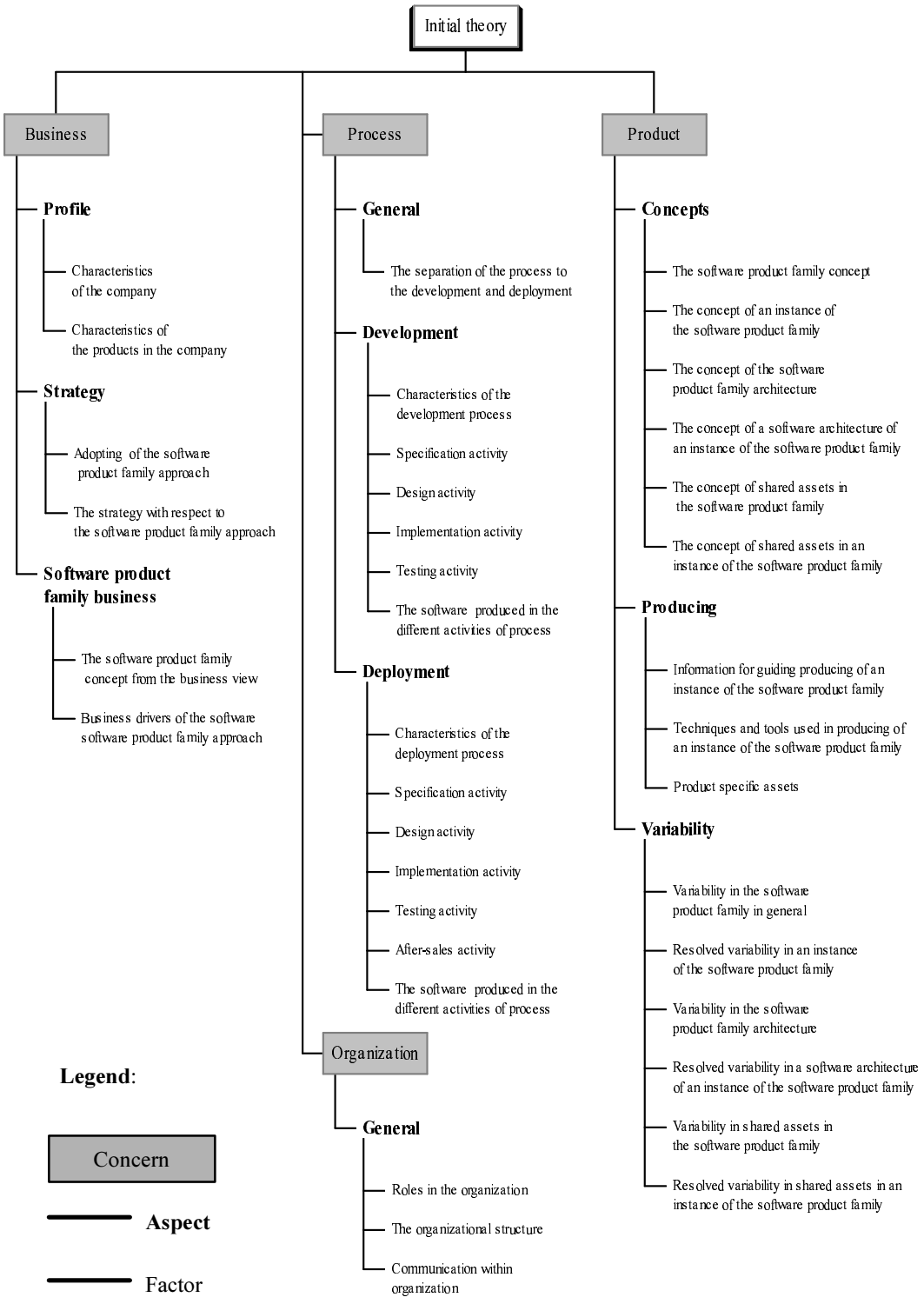

Figure 3 The concerns, aspects and factors in the research instrument.

3.2 Definitions

In this section the central definitions for the case study are given.

In this work the definition in (Basili 1992) for *software* is used:

> "Software can be viewed as a part of a system solution that can be encoded to execute on a computer as a set of instructions; it includes all the associated documentation necessary to understand, transform and use that solution".

Software is understood in this work in a rather wide meaning. Software can be e.g. a design decision, which is then encoded by implementing it as source code, which is further encoded by compiling to be executable in a computer as a set of instructions. During this encoding, more details are added and the stages are documented. In the given example, design decision is captured in design documentation.

The definition of *a software product family* is the same as it is used in (Bosch 2000a), but instead of the word "line" the word "family" is used:

> A software product family consists of a software product family architecture and a set of shared assets that are designed for incorporation into the software product family architecture. In addition, the software product family consists of the software products that are developed using the mentioned reusable shared assets.

The definition actually uses the term products, but in order to distinguish it from its more generic meaning, the term *an instance of the software product family* is used for a product that is a member of a software product family.

3.3 Business

In this section, the relevant aspects for the business concerns of a software product family are described. *The profile* aspect in Section 3.3.1 includes characteristics of companies applying the software product family approach and outlines the software product families they develop. Benefits of the software product family approach are briefly introduced and different strategies to adopt the software product family approach are enu-

merated in *the strategy* aspect in Section 3.3.2. The *software product family approach business* aspect in Section 3.3.3 introduces economic rationales for a software product family and specific characteristics of the software product family approach from the business and customer viewpoints.

3.3.1 Profile

The issues related to *the profile* aspect are the application domains in which the product family approach is applied and the characteristics of a company that applies the approach.

The software product family approach has been applied successfully in different application domains such as in consumer electronics, medical, banking and mobile telecommunications domains (Macala et al. 1996; Ommering and Bosch 2002; SEI 2002b). It seems that there are no easily identifiable application domains in which the software product family approach is not particularly applicable. The software in these application domains can also exists as a pure software or as embedded software; thus this issue does not seem to be a common denominator for the software product family approach.

According to (Bosch 1999b), the experiences in the field of the software product families are primarily from large American software companies that are often defense-related. However, the results from other studies (Bosch 1999a; Bosch 1999b; Clements and Northrop 2001; Knauber et al. 2000) show that the software product family approach can be applied also in small and medium sized enterprises and in companies that are not related to the defense industry. Therefore, there is no evidence of influences of these characteristics to the success of a software product family. However, according to (Northrop 2001) in a small organization practices related to the software product family approach are applied differently.

Profile related factors

Although the profile of a company applying the software product family approach does not seem to have an influence on the success of a software product family, it has its in-

fluences on practices. It is important to get an understanding of the general characteristics of the company and the products it develops in order to put the study into the right context. Thus, the factors related to the profile aspect are:

- Characteristics of the company
- Characteristics of the products in the company

The characteristics of the company include questions on the annual turnover, the number of employees and importance of the products compared other business areas such as consulting or education. The questions on the characteristics of the products deal with the application domain, the price of the product and other parts than software such as hardware or electronics and importance of software when compare with the other parts.

3.3.2 Strategy

The issues related to *the strategy* aspect are the strategic characteristics of the software product family approach and different adoption strategies.

According to (Card and Comer 1994) the decision for a company to adopt a reuse based approach, such as the software product family approach, has to be conscious and based on business rationales. Developing a software product family may lead to numerous benefits such as better quality (Wijnstra 2002), decrease of the development effort per product and decrease of the time to market per product (Clements and Northrop 2001). Some of the perceived benefits are tangible and they may be different for different stakeholders such as a product development manager or a software quality assurance manager (Knauber et al. 2001). In addition to the benefits, the software product family approach is a source of new risks such as effort wasted on assets that are developed for reuse but never actually used in instances of the software product family (Cohen 2001).

A company adopting the software product family approach has to decide between different adoption approaches (Bosch 2000a; Ommering and Bosch 2002):

- Evolutionary or revolutionary approach
- An existing software product family or a new software product family
- Top-down or bottom-up approach

- A component or a software architecture driven approach
- Planned or opportunistic reuse
- Software components available or to be developed
- Inter-organizational or intra-organizational

The evolutionary approach means that a company changes its business and software to the software product family approach piecemeal. The old products exists while the software product family approach is taken into use. The assets, either existing or new ones, are developed to become shared assets of the software product family. The opposite of the evolutionary approach is *the revolutionary* or "*big bang*" approach (Böckle et al. 2002). It means developing a software product family by changing all software at once. The development of new products is halted and instead the software product family is developed. All new products are based only on the developed software product family. There are different variations of these approaches such as the *incremental introduction* in which the software product family approach is introduced to one development department at a time and slowly extended to cover the whole organization (Böckle et al. 2002).

In *the existing software product family* approach, a company has products that are replaced by a software product family. In *the new software product family* approach, a software product family is developed to extend the product portfolio of the company and no products are replaced by the software product family.

The top-down or *decomposable* approach means that a higher level structure is created, parts of the software is produced in the context of the structure, and variability is built into the structure. *The bottom-up* or *compositional* approach means producing software, to e.g. a repository, and then software engineers use software to compose a product. A higher level structure, which specifies the context for the shared assets, does not exist in this approach.

The planned approach means that an organization puts explicit effort to make software engineers reuse the existing software. Opposite of the planned approach is the *opportunistic* approach, in which software engineers are assumed to pick pieces of the existing

software and reuse them. In the opportunistic approach, there are typically no incentives or obligations to reuse, whereas the planned approach encourages or even enforces reuse.

The use of software components available approach uses the existing software components typically from external but also from internal sources such as from a repository of legacy software components. In *the software components to be developed* approach it is assumed that there are no appropriate software components available, but in order to develop a software product family the components have to be developed.

The intra-organizational approach assumes that software is developed within an organization. *The inter-organizational* approach uses more externally developed assets such as the domains specific architectures (DSSA), commercial of the shelf components (COTS) or sub-contracted assets. As noted in (Ommering and Bosch 2002) also in the intra-organizational approach advantages are taken of the inter-organizationally developed assets, but it is emphasized that the core of the product family is develop internally.

Traditionally, a combination of the planned and top-down approaches is argued to be the only working solution in the software product family literature (Bosch 2000a), although the compositional and bottom-up approaches also have supporters (Böckle et al. 2002; Ommering 2000; Ommering and Bosch 2002). In addition, typically the software product family approach is regarded as a planned and intra-organizational approach (Bosch 2000a; Ommering and Bosch 2002). According to (Clements and Northrop 2001), an orthodox compositional approach without any guidelines how to compose shared assets is not a software product family approach. Nevertheless, there does not seem to be consensus on which of the approaches are the best for each circumstance

Regardless of the approach, a company that has adopted software product family approach should also institutionalize it (Böckle et al. 2002). Institutionalization of software product family approach means influences on the arrangements in a company such as policies, procedures, and conventions so that all of them support the software product family approach. Furthermore, a strong commitment is required through the organization (Clements and Northrop 2001).

Strategy related factors

It seems that adoption the software product family approach cannot be unconscious, but it has to be based on rationales and there are several decisions to take a stand on. The factors related to the strategy are:

- Adoption of the software product family approach
- The strategy with respect to the software product family approach

The factor adoption the software product family approach includes questions on when was it adopted, reasons for it, how intentional was it and the development approach in the company before it adopted the software product family approach. The strategy with respect to the software product family approach factor includes questions on the software product family approach related strategy, the changes in it in the course of time and strengths, weaknesses, opportunities, and threats (SWOT) of the software product family approach.

3.3.3 Software product family approach business

For the software product family approach business aspect, how the software product family approach is not only a technical product development principle is described and economic rationales for the software product family approach are given.

The software product family concept can be ambiguous or even vague in a company, it can vary depending on the company and stakeholders, and it may mean different thing for different stakeholders (Clements and Northrop 2001). In Chapter 2 it was pointed out that the software product family approach is more than just a technical software construction principle. According to the definition given by Clements and Northroup (Clements and Northrop 2001), instances of the software product family satisfy the specific needs of a particular market segment.

Economic rationale for the software product family approach is that developing instances of a software product family is more efficient than building the same set of products from the scratch. A *payback point* is the number of instances of a software

product family for which the costs caused by initial investments for the software product family and developing these instances of the software product family are the same as building the same number of products from the scratch. After the payback point, total costs of the software product family are smaller than development of the product from the scratch. The payback point is estimated in (Macala et al. 1996; Weiss and Lai 1999) to be after three or four products for some products.

Software product family business related factors

Software product family business has its own influences on the software product family concept. On the other hand, business rationales are the fundamental reasons for having a software product family. There are two factors related to the software product family business aspect:

- The software product family concept from the business view
- Business drivers of the software product family approach

The software product family concept from the business view factor includes questions on how is the software product family understood and what kind of features are there in the software product family. The features in the software product family can be classified to the important, attractive and additional features from customers' point of view. The business drivers of the software product family approach factor includes questions on the quantifiable and qualifiablebenefits and costs of the software product family

3.4 Software processes

In this section, the software process concern of the software product family approach is described. In Section 3.4.1 the general aspect of software product family processes is described. The software product family development process aspect is dealt with in Section 3.4.2 and the software product family deployment process aspect is dealt with in Section 3.4.3.

3.4.1 Software product family processes in general

The issues in general aspect of software product family processes concern characteristics and decomposition principle of a software process, and typical processes in the software product family approach and their interdependencies

Software process

The software product family approach based software development is conducted in a software process. The following definition of *a software process* is given in (Sommerville 1995):

> "[A software process is a] set of activities and associated results which produce a software product"

According to (Beck 1999; Sommerville 1995) one way to break down a software process is a decomposition to a analyses, design, implementation, testing, and operation and maintenance activity as follows:

- *Analyses*. Customer and system requirements are elicited and captured.
- *Design.* A software product architecture and a more detailed design is developed and documented.
- *Implementation.* The design is realized.
- *Testing.* The designed and implemented system is ensured to meet the requirements.
- *Operation and maintenance:* The system is in use, possible deficiencies and bugs are fixed and it is changed in the course of time to meet changing customer needs.

Organizing and timing of these activities can be different (Beck 1999) and there is not a right way to organize a software process. According to (Sommerville 1995), there exist several different process models, of which the waterfall and evolutionary or iterative models are widely used. Lately agile models (Beck 1999), such as the extreme programming, have gained increasing amount of attention. All these models deal with primarily the four first activities. In the *waterfall model*, the activities follow each other and a product development is rather straightforward. In the *iterative model*, the a product is

developed in several iterations and the product is not even meant to be finished at once or in the first iteration cycle. In the *agile models*, the duration of the activities is short or even parallel. A sketch of these three process models is presented in Figure 4.

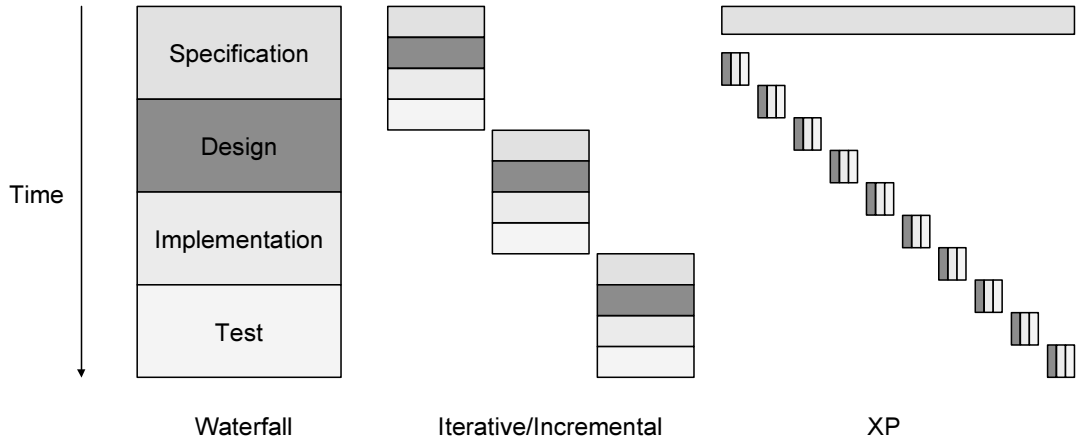

Figure 4 The specification, design, implementation, and testing activities in the waterfall, iterative/incremental and XP process model (Beck 1999).

Software product family processes

Traditionally in the software product family approach the software process is twofold (Bosch 2000a; Clements and Northrop 2001; Weiss and Lai 1999). The two processes are called in this work the *software product family development process* and *software product family deployment process* or the *development process* and *deployment process* in short (Bosch 2000a). The development process produces the software product family architecture and shared assets for reuse. Deployment process uses the created software product family architecture and shared assets and produces instances of the software product family. The processes with their associated artifacts are presented in Figure 5

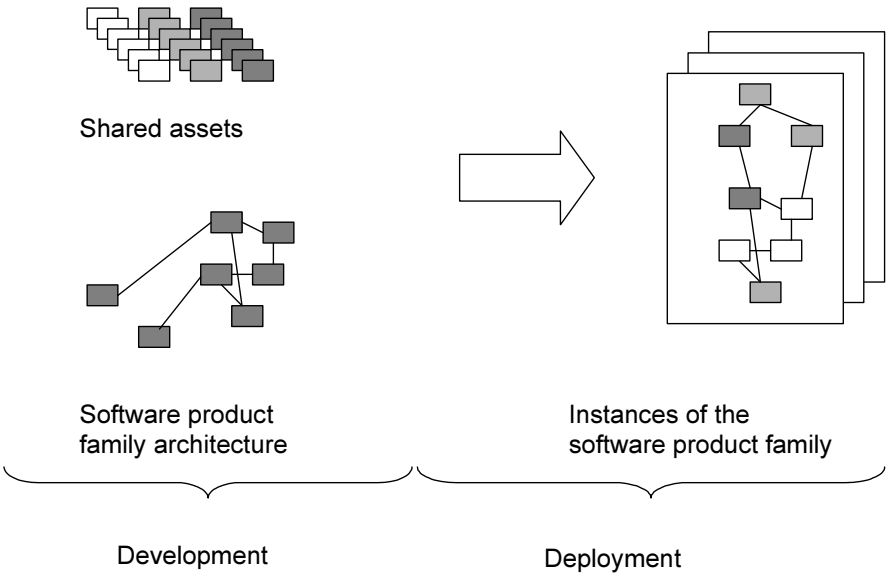

Figure 5 The two software product family processes.

The two processes, the development and deployment process, are identifiable, but they are not isolated (Clements and Northrop 2001). There is an obvious link from the development process to the deployment process as the shared assets and software product family architecture are produced in the development process and then reused in the deployment process. There is also a link in the opposite direction, i.e. in a deployment process is developed or required something that becomes a new shared asset. The nature of this link varies. In one extreme, every new requirement that is not covered by the software product family causes at least a consideration, e.g., for changing a shared asset or developing a new shared asset (Bosch 2000a). In the another extreme the software product family architecture and shared assets are never changed while instances of the software product family are produced. Changes to the software product family architecture and new shared assets are introduced only as a new version of the software product family is created (MacGregor 2000).

A third process related to software product family approach that can be separately identified is a software product family *management process*. The management process controls the development and deployment processes. According to (Clements and Northrop 2001), a management process has a critical role in a successful software product family

approach, both for technical issues such as strategic technical decisions and for organizational issues such as the organizational structure and staffing.

Software product family process related factors

Software processes in the software product family approach seem to be divided clearly at least into the development and deployment processes. This separation seems to be relevant in order to understand the processes and the software product family approach in general. Consequently, the factor related to a software product family process is:

- The separation of the process to the development and deployment

This factor includes questions on the existence and characteristics of the separation

3.4.2 Development

The issues related to the software product family development process aspect are the development process in general and the specification, design, implementation, and testing activities, in that order

In the software product family development process the software product family architecture and the shared assets are developed (Clements and Northrop 2001; Weiss and Lai 1999). The development can also be called core assets development (Clements and Northrop 2001) or domain engineering (Weiss and Lai 1999) process. In this thesis, the term *a producer* is used for the developers in a development process (Fafchamps 1994).

According to (Macala et al. 1996), software is not inherently reusable. Effort should be spent on making the software product family architecture and shared assets reusable. The shared assets should incorporate variability and they should be designed with care in order that major unpredicted changes are not necessary in the near future (Bosch 2000a). In fact, it is specific for the software product family approach that the software is not development for a specific product, but in order to be reused in an instance of the product family. In (Clements and Northrop 2001) this is called *a production capability* for instances of the software product family.

The specification is the first activity in the development and includes scoping (Bosch 2000a). S*coping* means selecting and identifying possible features and products that are to be included in the software product family *scope* (Clements 2001). Scoping is essential for a software product family development, whereas for a single-system development it can be more informal and even partially ignored. Another difference is that in the software product family approach even mutually exclusive features can be included in the scope, as long as they are not meant to be realized in a same instance of the software product family.

According to (Clements 2001), the boundary of the scope in a software product family does not have to be strict. A sub-set of the products and features that are included in the software product family scope can be clear. Yet there may be products or features that are on the boundary to be included in or excluded from the software product family scope. In fact, a software product family scope is rarely crisp.

There are three different approaches for scoping (Clements 2002; Krueger 2002):

- Proactive scoping
- Reactive scoping
- Extractive scoping

In *proactive scoping*, a scope is attempted to set as precisely and early as possible. In *reactive scoping*, a scope is not set at once but it is adjusted along as instances of the software product family are developed. In *extractive scoping*, the scope of the existing products is used as a basis for the software product family scope. Consequently, the scoping is in a close relationship with the software product family adoption approach.

A specific issue in the software product family approach in the design and implementation activities is that the software product family architecture and shared assets are not necessary developed to the level of concreteness or completeness where they would become executable software. An example of such non-executable or in other words an unfinished implementation is a framework in which details are left open to be fixed in the deployment process (Bosch 2000a).

Development related factors

Software processes can divided into the analyses, design, implementation, and testing activities as noted earlier and it seems to be applicable to use this decomposition as a basis for understanding the development process in detail. The factors related to the development process are:

- Characteristics of the development process
- Specification activity
- Design activity
- Implementation activity
- Testing activity
- The software produced in the different activities of process

The questions on the characteristics of the process factor deal with the development process model, activities in the development process, organizing of the activities and documentation on the process. Questions on the specification, design, and implementation activities deal with tasks performed in each activity, responsibilities of performing the tasks, and documentation of the activities. The software produced in the process factor includes questions on what kinds of shared assets are produced in the development process.

3.4.3 Deployment

In the software product family deployment processes aspect, deployment process in general and the specification, design, implementation, testing, and after sales activities and in that order are described.

In the deployment process, an instance of the software product family is developed using the software product family architecture and shared assets (Clements and Northrop 2001; Weiss and Lai 1999). The development can also be called the application engineering (Weiss and Lai 1999) or the instantiation (Bosch and Högström 2000) process. In this thesis, the term *a consumer* is used for the developers in a deployment processes (Fafchamps 1994).

The deployment varies in how systematic and predefined it is. According to (Clements and Northrop 2001), instances of the software product family are developed “in a prescribed way”. The software product family definition that is used in this thesis does nor require that, but requires only the use of the software product family architecture and shared assets. In (Bosch 2002) is noted that the deployment process for configurable product can be very systematic and predefined because automated tools or techniques support the development of an instance of the software product family.

The deployment process varies also in the work effort. Using generative techniques may lead to very small amounts of work because instances of the software product family can be automatically developed from the existing software components (Czarnecki and Eisenecker 2000). In other techniques, deployment process may include development of new software and therefore may need more effort (Bosch 2000a; Clements and Northrop 2001).

In the specification activity of the deployment process, the customer specific requirements for an instance of the software product family are captured (Sommerville 1995). The customer’s requirements are then compared with the requirements that the software product family covers. A lot of requirements should be that same, according to the software product family definition that states that products have commonalities and hence the requirements should be common. To fulfil the rest requirements, the software product family has to be extended or changed and additional or conflicting features has to be removed (Bosch 2000a).

The software product family architecture and shared assets are then used to implement an instance of the software product family. The implementation as well as testing work can be reused as the shared assets are used. Finally, an instance of the software product family is delivered to a customer. Later an instance of the software product family may be changed or updated. (Bosch 2000a)

Deployment related factors

Similarly, as in the development, in the deployment the software process can be divided into the specification, design, implementation, and testing activities. In addition, there is the after-sales activity. In the development process a particular customer is not taken into account, but in the deployment process, the requirements of an individual customer have an essential role. The factors relevant with respect to the deployment are similar as in the development with a few exceptions:

- Characteristics of the deployment process
- Specification activity
- Design activity
- Implementation activity
- Testing activity
- After-sales activity
- The software produced in the different activities of process

The questions related to the characteristics of the deployment process are the process model, activities in the process, organizing of the activities, work effort in the deployment process, the complexity of the process and documentation of the process. Work effort is lead-time and man-hours that the deployment takes from the customer's requirement specification to the delivery. The questions related to the activities deal with work performed in each activity and responsibilities in them. The after sales activity factor includes questions on changes to instances of the software product family after delivery. The questions related to the last factor deal with software produced in the process, which is at least partially reused from the development activity.

3.5 Organization

In this section, organization concern of the software product family approach is described. The issues related to the organizational concern are the principles for the organizational structure and the producer and consumer roles, and models for the producer and consumer relationship and organizational structures.

A common way to organize software reuse is to separate the consumers, who develop assets for reuse, from the producers, who develop the products to the customers using the developed assets (Bosch 2001; Fafchamps 1994; Macala et al. 1996). According to (Bosch 2001), several factors, such as geographical distribution, project management, organizational culture, and the type of systems produced, influence the nature of the separation between the producers and consumers.

In (Bosch 2000b) it is noted that a clear separation between the producers and consumers ensures truly reusable software because they are not developed for a particular product. On the other hand, if the producers are too separated from the consumers, the producers may produce too generic and unusable software and they may not be not able to react to changes fast enough.

Fafchamps (Fafchamps 1994) has identified four models for producer-consumer relationship:

- Lone producer model
- Nested producer model
- Pool producer model
- Team producer model

A *lone producer* develops shared assets without being clearly assigned to any specific place in the organization with respect to the consumers. More then one consumers use the developed software. A n*ested producer* is working as a member and under supervision of a consumer team. The responsibility of the nested producer is to provide the consumer team with reuse expertise, services and software. In the *pool producer* model, two or more consumer teams have assigned resources to a pool of producers that develop software for reuse. In the *team producer* model there is a producer team that resides within organizational structure in a similar position as the consumer teams. The models are presented in Figure 6

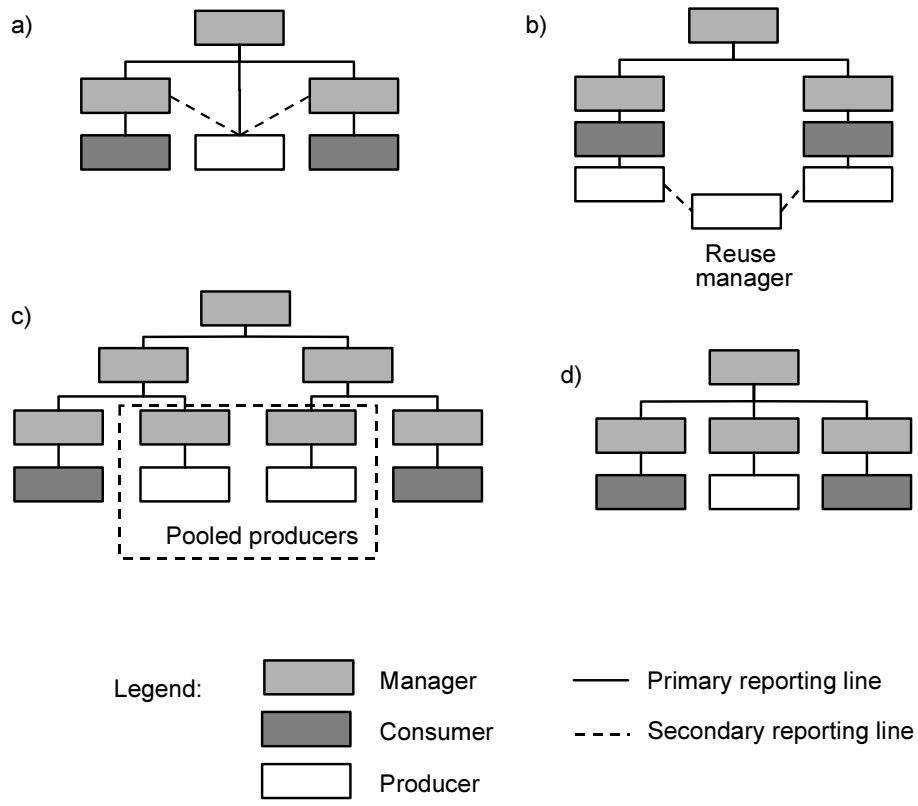

Figure 6 The four models for producer-consumer relationships. a) Lone producer, b) nested producer, c) pool producer, d) team producer (Fafchamps 1994).

Bosch (Bosch 2001) has identified four different models for software product family development organization that applies the software product family approach:

- Development department model
- Business unit model
- Domain engineering unit model
- Hierarchical domain engineering unit model

In the *development department* model there is no clear and permanent separation in the organizational structure for the producers and consumers, but everyone does both. In the *business unit* model, each business unit is specialized in developing a subset of the instances of the software product family. There is no single unit developing only the software product family architecture and shared assets, but they are owned together and de-

veloped and evolved by the business units according to their needs; Consumers may occasionally be in producer role. In the *domain engineering unit* model, producers are organized to one organizational unit, called a domain engineering unit that is responsible for developing and evolving the software product family architecture and shared assets. Consumers are in the rest of the units, called business units, developing instances of the software product family. In the *hierarchical domain-engineering unit* model, there are several domain-engineering units that are organized hierarchically. If two or more domain engineering units have commonality in their software, the commonality between the software is developed in another higher-level domain-engineering unit. In addition, business units develop instances of the software product family. The models are presented in Figure 7.

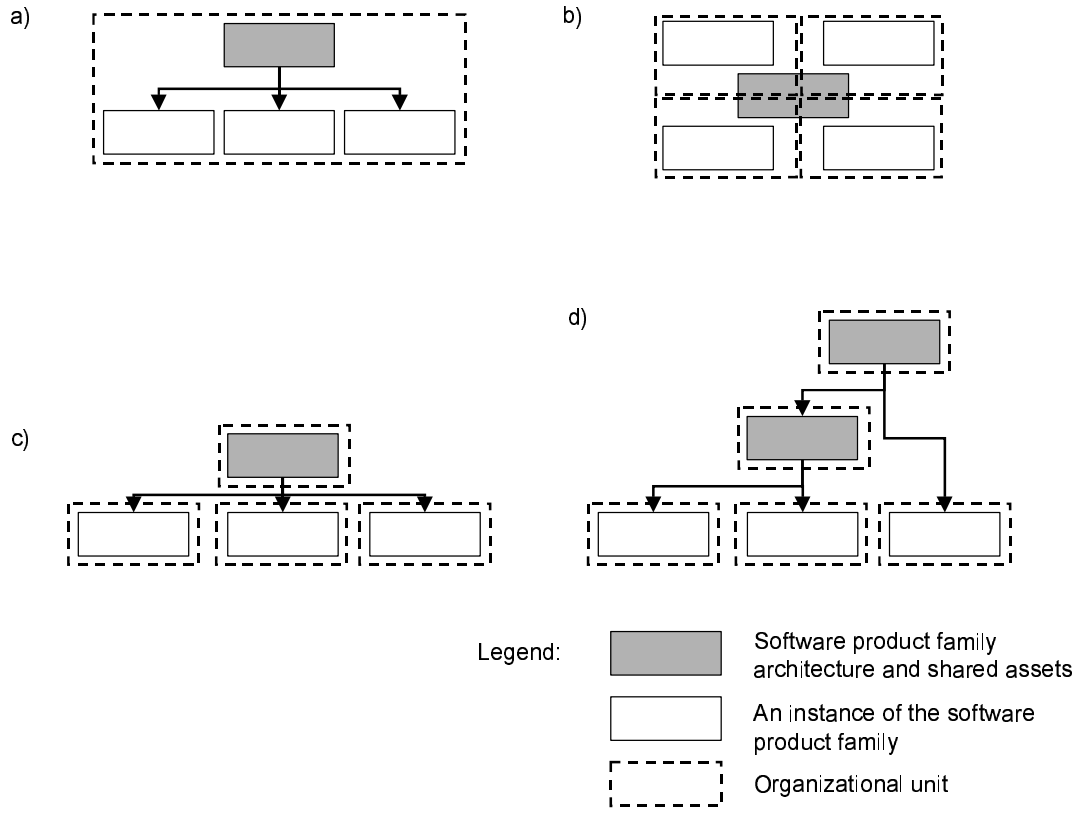

Figure 7 Organizational models for software product family. a) development department, b) business unit, c) domain engineering unit, d) hierarchical domain engineering unit (Bosch 2000a).

Organization related factors

Several aspects in the organizational concern seem to be important and specific for the software product family approach. The factors related to the organization are:

- Roles in the organization
- The organizational structure
- Communication within organization

The questions on the roles in the organization deal with the roles themselves, the dominating roles in the decision making and strictness of the roles. The questions related to the organizational structure factor deal with what kind of organizational structure is there and the changes in the organizational structure at the time the software product family approach was adopted. The questions with respect to communication factor concern how do people communicate and the form and scheduling of meetings, the people present in the meetings and the organizational roles of those people.

3.6 Product

In this section, the product related concerns of the software product family approach are described. In Section 3.6.1 the concepts aspect related to the software product family artifacts is described. In Section 3.6.2 producing aspect, i.e. the artifact point of view to the deployment process, is outlined. Variability aspect is dealt with in Section 3.6.3.

3.6.1 Concepts

The concepts aspect is dealt with in a top-down manner from larger and more general artifacts to more concrete and fine-grained ones. The concepts covered include the software product family concept, the concept of a software product family architecture, and the concepts of shared assets.

Software product family

A software product family is an abstract concept on its own. According the used definition, it consists instances of the software product family, a software product family architecture and shared assets (Bosch 2000a).

In Chapter 2, it was explained that instances of a software product family have commonalities. The commonalities are based on the use of the software product family architecture and shared assets. Differences between instances of the software product family are realized in a deployment process. According to (Bosch 2000a), the differences between the instances of the software product family are realized through either changes to the instantiated shared assets and software product family architecture or through software that is developed for that particular instance of the software product family.

Software product family architecture

A software product family architecture has a special role in a software product family (Clements and Northrop 2001) and it is sometimes, as in this thesis, treated separately from the shared assets (Bosch 2000a). The importance of the software product family architecture is that it specifies the context for developing the shared assets (Clements and Northrop 2001).

In this work the following definition of a *software architecture* is used (Bass et al. 1998):

> "The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them".

A software product family architecture is further defined as follows (Bosch 2000a):

> "[Software product family architecture is a] common architecture for a set of related products or systems developed by an organization"

In the definition, “a set of related products or systems developed by an organization” refers to a set of instances of a software product family. It also emphasizes that the software product family architecture is developed by an organization, and it is not therefore necessary, e.g., a reference architecture.

The above definition of a software architecture is probably the most widely used in software engineering. In (SEI 2002a) it is noted that there is not standard definition for a software architecture, instead there are dozens of different definitions.

The above definition of a software product family architecture emphasizes that a software product family architecture is developed to allow instances of the software product family, which differ from each other, to be based on it (Bosch 2000a; Clements and Northrop 2001). A software architecture is an abstract design that describes the design of the software product family independently from its implementation. Therefore, even a single system software architecture permits to some extend different products based on it, but for a software product family architecture this is a key issue

There are several software product family adoption approaches, as described in Section 3.3.2. Some of these approaches emphasize more the role of the software product family architecture such as the top-down approach and some other emphasize it less such as the bottom-up approach (Bosch 2000a). Consequently, the selected software product family adoption approach has its influence on the role of a software product family architecture.

An issue in developing a software product family architecture is how similar is it designed to be in the instances of the software product family or how remarkable changes are planned to it. The two extreme approaches, according to (Bosch 2000a), are the use of a software product family architecture “as-is” or only partially. In the former, changes to a software product family architecture are rarely made or at least the changes are very small. In the latter case, a software product family architecture is used to cover only a part of the software architecture of an instance of the software product family.

A closely related concept to a software product family architecture is *a platform* that forms a basis on top of which instances of the software product family are developed.

The platform does not cover whole instance of the software product family, but it is the a part that is included in all instances of the software product family. A more precise definition of *a platform* is (Meyer and Selinger 1998):

> "[A platform is a] set of subsystems and interfaces that form a common structure from which a stream of derivative products can be efficiently developed and produced"

According to (Meyer and Selinger 1998), instances of the software product family are created in the best case by adding customized modules to the platform without additional integration costs or tailoring the platform. In (Meyer and Lehnerd 1997) is noted that as add-in modules are developed, they are included in the software product family and they become shared assets.

Shared assets

In this work the term *shared assets* is understood as follows (Northrop 2001):

> "[Shared assets] often include, but are not limited to, the architecture, reusable software components, domain models, requirements statements, documentation and specifications, performance models, schedules, budgets, test plans, test cases, work plans, and process descriptions".

The terminology related to the shared assets is often used non-uniformly. The terms reusable assets or core assets are also used as synonyms for the term shared assets.

The term *a component* is widely used though its meaning is rather ambiguous. According to the above definition of the shared assets, a software component is one of the shared assets, but not the only one. Probably the most used definition of *a software components* is (Szyperski 1999):

> "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.".

Product related factors

It seem to be clear that artifacts that a software product family consists of appear in two roles. First, they are in the software product family for reuse and, second, they are used to produce instances of the software product family. This categorizes artifacts in a software product family from one point of view. On the other hand, there are artifacts in three levels of abstraction: a software product family in general, a software product family architecture and shared assets. Combining these two dimensions, results in the following six factors:

- The software product family concept
- The concept of an instance of the software product family
- The concept of the software product family architecture
- The concept of a software architecture of an instance of the software product family
- The concept of shared assets in the software product family
- The concept of shared assets in an instance of the software product family

The software product family concept includes questions on what is meant by it, why is there a software product family in the company from the technical point of view and how complex is it to develop a software product family. The concept of an instance of the software product family includes questions on what is meant by it and commonalities and differences between instances of the software product family. The concept of the software product family architecture related questions deal with how the concept is understood in the company, the number of entities in it, its documentation, and how complex is it to develop the software product family architecture. The questions with respect to the concept of a software architecture of an instance of the software product family deal with how does it and information on it differ from the software product family architecture. The concept of shared assets in the software product family includes questions on what is meant by them, what are the sizes of them, what information is there of them and how the information is documented and how complex is it to develop them. The questions with respect to the concept of shared assets in an instance of the software product family concern the number of them.

3.6.2 Producing

The producing aspect deals with the software product family architecture and shared assets from the technical viewpoint as they constitute in a deployment process an instance of the software product family. It is not strictly speaking an artifact, but is introduced here to clarify the relationship between processes and artifacts.

According to (Bosch 2000a), producing an instance of the software product family is done by one or more of the following:

- Using the shared assets, and configuring and developing instance of the software product family specific extensions to them
- Developing product specific assets
- Product integration code

Using the shared assets means that they are taken, e.g. from a repository, and used as part of an instance of the software product family. *Configuring* means adapting a shared asset e.g. by parameters, to meet the requirements of the customer or environment. *The instance of the software product family specific extensions* means that a shared asset are modified, e.g. by adding, removing or changing code, if they cannot be configured to meet the requirements of the customer or environment by configuring. Product specific assets are developed, because in the software product family there are not shared assets that would meet the given requirements. Finally, *product integration code* means code that is developed to form a complete instance of the software product family.

In (Clements and Northrop 2001) it is pointed out that shared assets developed in a software product family should include an attached process and a production plan. *The attached process* describes how to use the shared asset and *the production plan* describes how to combine the attached processes and is a general guide how to build instances of the software product family from the software product family architecture and shared assets. However, as noted earlier, not all software product family definitions require such production plan, and in particular, this is not included in the software product family definition used in this work.

Producing related factors

The following factors are related to the producing of an instance of the software product family:

- Information for guiding producing of an instance of the software product family
- Techniques and tools used in producing of an instance of the software product family
- Product specific assets

The questions on the information for guiding producing factor deal with existence of such information, contents of the information and its documentation. The techniques and tools used in producing of an instance of the software product family factor includes questions on what techniques and tools are used and how are they used. The product specific assets factor includes question on how often are such assets developed, the number of them in an instance of the software product family and the reasons why they have to be developed.

3.6.3 Variability

The variability aspect includes definitions for variability, a variability point and a variant, description of types of variability of variants, relations between variants, variability resolving in software process, variability at different levels of abstraction and implementation mechanisms of variability.

The definition in (Gurp and Bosch 2001) for *variability* is used in this work:

> "Variability is an ability to change or customize a system"

Another point of view to variability is evolution, which is variability in time, i.e. the software changes in the course of time.

According to (Gurp and Bosch 2001), variability is located at variation points. A more precise definition for a *variation point* is (Jacobson et al. 1997):

> "A variation point identifies one or more location at which the variation will occur".

One definition for a variant is (Jacobson et al. 1997):

> "[A variant is] a type-like construct, typically use case or object type or class, intended to be inserted at an appropriate variation point to specialize an abstract type or class".

Consequently, variability is *resolved* by attaching a variant to its corresponding variation point.

Relationships of variants may have influence or constraints on each other. Two such relationships are introduced in (Anastasopoulos and Gacek 2001):

- Mutually exclusive
- Mutually inclusive

Mutually exclusive variability means that one choice among the alternatives in a variation point causes that some other choice cannot be made. Mutually inclusive variability means that if one variant has selected then also some other variant needs to be selected.

In (Bachmann and Bass 2001) three types of variability are identified:

- Optional
- An instance of alternative variants
- A set of variants of alternative variants

Optional variability means that in order to resolve the variability, one variant has to be attached to the corresponding variation point or the variation point has to be left empty. An instance of alternative variants means that in a variation point there are variants of which one has to be selected in order to resolve variability in that particular variation point. A set of variants of alternative variants means a set of variants have to be selected for a variation point in order to resolve variability in that variation point.

In (Gurp and Bosch 2001), variability in software development is considered as a transformation from higher abstraction levels to more concrete and detailed descriptions, in which each step constraints the system more and resolves variability. An example of such transformation is the implementation process where design documentation is trans-

formed to source code and at least some variability is resolved during the process. In each step more variation points become resolved. In Figure 8 is two funnels that illustrate the variability resolving are illustrated. The left funnel represents *early* variability, in which the variation points are resolved immediately and the right funnel represents *delayed* variability.

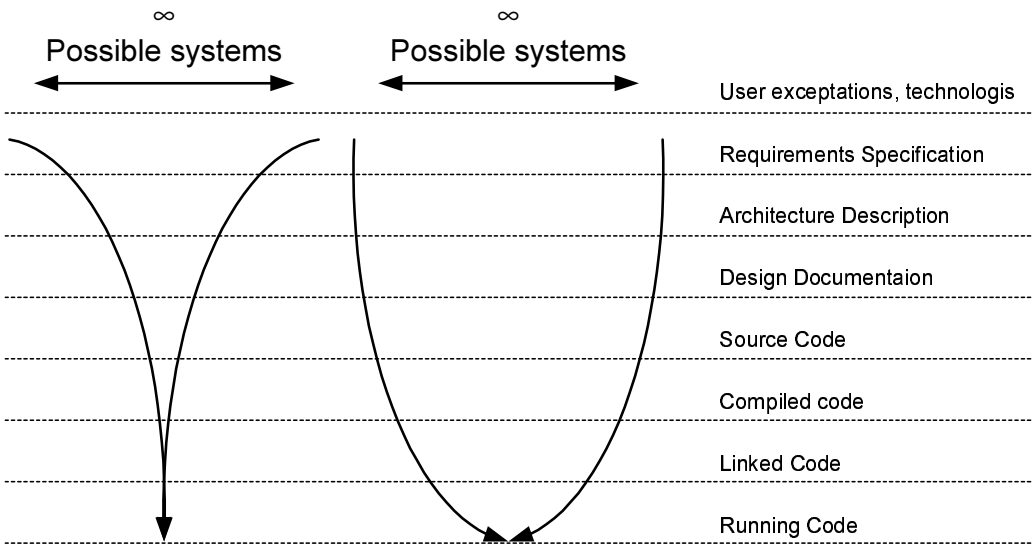

Figure 8 Variability in early and delayed variability resolving. Designs process constraints systems such that in the end one possible system remains (Gurp and Bosch 2001).

In the above variability resolving was regarded as a process. Another view, identifiable from Figure 8 is to think of variability as related to the levels of abstraction that the activities in the process concerns to. For example, at the architectural level many details with respect to sorting such as whether to sort increasingly or decreasingly are not decided yet. At the design are source code level they become designed and implemented, but can be left unbound such that both alternatives are still possible to select because both alternatives are implement. The binding can be done later such as in the runtime, i.e. the sorting order can be changed while software is in use.

The definition in this work for a software product family does not specify in which level the variability takes place. Variability can take place e.g. in software product family architecture as well as in assets as long as, according to the definition of variability, the

developed system is changed or customized. When a software product family architecture and shared assets are developed some variability is left unresolved. This enables then instances of the software product family that differ from each other.

In (Bosch 2000a) it is noted that shared assets can be implemented at source code level such that there is:

- No implementation
- One implementation
- More implementations

In case when there is no implementation, a shared asset is so diverse that for every instance of the software product family the implementation has to be done individually. In the case of only one implementation, different kinds of variability mechanisms can be used to adjust an shared asset to meet requirements of the customer and environment. The last case, more than one implementation, takes place e.g. when the implementations are such diverse that it is advantageous to have multiple of them. This is a case when there are several implementations corresponding to the different alternatives in a variation point

There are numerous different mechanisms to achieve variability (Clements and Northrop 2001). In (Svahnberg and Bosch 2000) at the architectural level following mechanisms are enumerated: inheritance, extensions and extension points, parameterization, configuration, generation and “if-defs”. In (Anastasopoulos and Gacek 2001) the following implementation approaches for code level variability mechanisms are outlined: aggregation or delegation, inheritance, parameterization, overloading, (Delphi) properties, static libraries, dynamic class loading, dynamic link libraries, conditional compiling, frames, reflection, and aspect oriented programming.

Variability related factors

Similarly as in the case of the concept factors in Section 3.6.1 there are six factors that are based on combining the three levels of abstraction with artifacts of a software prod-

uct family for reuse and an instance of the software product family that uses them. Consequently, the factors related to variability are:

- Variability in the software product family in general
- Resolved variability in an instance of the software product family
- Variability in the software product family architecture
- Resolved variability in a software architecture of an instance of the software product family
- Variability in shared assets in the software product family
- Resolved variability in shared assets in an instance of the software product family

Variability in the software product family includes questions on what varies in the software product family and reasons for the variability. The questions on resolved variability in an instance of the software product family deal with the number and distribution of different instances of the software product family and the number of variability resolving decisions necessary to produce an instance of the software product family.

Variability in the software product family architecture factor includes questions on what varies in the software product family architecture, the reasons for variability, types of variability, variation points and the relations between them. In addition it includes question on evolution of the software product family architecture. For the factor resolved variability in an instance of the software product family architecture there are questions on the number and distribution of different variants of the software product family architecture. In addition, it includes questions at architectural level on designing and the number of variation points and variants in a variation point, and the techniques for and stage at which variability is resolved variability

The variability in shared assets in the software product family factor includes questions on how the shared assets vary and the reasons for variability and evolution in the shared assets. The Resolved variability in shared assets in an instance of the software product family factor includes questions on the implementation of variability, the number of variants and shared assets that do not vary, the used techniques to resolve variability and stage when the is resolved.

4 Case

In this chapter, a description of the case study of the software family related practices in a company producing automation-related software is presented. The Section 4.1 the essential terminology is introduced. The software product family business aspects are outlined in the Section 4.2. In Section 4.3 the software product family processes is described, and in Section 4.4 the software product family architecture and shared assets is described.

4.1 Background

The product family in the company includes electronics, mechanics, and software parts. In this chapter, the terms *a product family* and *a product* are used to refer the whole product, including mechanics, electronics, and software. The terms *a software product family* and *an instance of the software product family* refer only to the software part of the products.

The *software product family* is understood similarly as used in (Bosch 2000a) and could be defined as follows:

> The software product family consists of a set of instances of the software product family for a particular market segment that are produced using a common software product family architecture and a set of shared assets. In a deployment process, an instance of the software product family is produced by configuring the software product family architecture and shared assets and in some cases tailoring.

Configuring means customizing an instance of the software product family and it is done without modifying source code. *Tailoring* is customer specific modifications in which source code is modified or totally new code is produced. Tailoring is done approximately for one third of the instances of the software product family.

The products operate in the customer's existing business environment tightly integrated with other *existing systems*. The product is an intermediate between the existing systems and it manages them and cooperation between them. The instances of the software product family are installed on a PC and require commercial desktop software and Microsoft Windows-operating system. The size of the software product family is about half million lines of code that is programmed largely in C++ and Visual Basic.

For the customers the basic functionality and especially the reliability are the most important properties of the software product family. The customers and users do not buy software systems but rather systems that also include software. In addition, there is not especially hard demand for new features that are not in the core set of features of the software product family. Another important aspect for the customers is an ability to integrate an instances of the software product family to the existing systems. Parameterization and configuration are important and integral part of the software product family.

4.2 Business

The company operates on business-to-business markets. The latest annual turnover of the product family is tens of millions euros. The share of the software product family cannot be separated from the total turnover because it is not sold separately, although it is an important part of the products. The company develops about 50 instances of the product family in a year. The company employs a couple of hundreds people and about one fourth of them work with the analyzed product family. About 25 employees are working with the software product family. In fact, using the software product family approach the company has been able to double the number of deliveries per year using the same number of employees.

The history of the software product family dates back to the middle of the 1990's. From the 1980's to the middle of the 1990's, the business, with respect to software, based on customer specific projects. At that time, in order to produce the software to a new customer, the company took a copy of the software from an earlier project and tailored it. In the end of the 1990's, the company made a decision to change its software to be more product-oriented and it started to develop a software product that would replace the pro-

ject deliveries and become the analyzed software product family. The time-effort that was invested on the initial development was, according to an estimation, 6000 person-hours during 1,5 years.

The decision to adopt the software product family approach was strongly intentional. The main reason was that the company saw no possibility to continue profitably in the project-oriented business. It wanted to increase the number deliveries without extensively increasing the number of employees, which was not possible in project deliveries. The responders pointed out that using the project-based software a peak in volume of orders, such as 20% increase, would have caused a need to recruit people in order to meet the demand. If the increase had been only temporary, the company would have been in trouble with too many employees. In the software product family approach, employees from product family development process could be assigned to instances of product family deployment projects and vice versa. Another mentioned reasons to have a software product family were to produce products more efficiently and to be able to handle the complexity in the domain and in the products better.

At the present, the company aims to maintain the software product family that is as configurable as reasonably possible. The strategy has remained rather stable the past few years. To carry on the strategy, the company has needed and succeeded to make a conscious decision in favor of the software product family approach and to establish a strong commitment to the software product family approach through the company

The project that shifted the company away from the project-oriented business, although it was not called a software product family adoption project at that time, had typical software product family objectives. The goal was to produce a product, which would embody variability and enable efficient delivery. Hence, the software product family adoption approach resembles, although was not orthodoxly, the revolutionary approach for existing products.

Before the analyzed software product family, the company had produced another software product family where it introduced the software product family approach. In that

software product family they saw the benefits of the software product family approach and rehearsed utilization.

The application domain of the software product family at the present is the same as before in the project deliveries in great extends. Therefore, although the application domain and the company have evolved in the course of time, the business could be characterized rather stable and the software product family has been a successful with respect to its scope and selected development decisions.

4.3 Processes

The software processes in the company are clearly twofold as typically in the software product family approach. There are identifiable and separated a product family development process and an instance of the software product family deployment process. The deployment process is from the software engineering viewpoint rather lightweight because an instance of the software product family is developed by configuring. Tailoring may cause extra work, but it is avoided if possible. Therefore, much of the work is done in other areas than strictly software engineering such as sales and education.

4.3.1 Development process

The software product family development process consisted of a scoping, prototyping, design, implementation, and testing activity. The developed assets were not even planned, though not prohibited, to be used in other software products or product families. The first deployment project, using the software product family, was scheduled for a certain point of time. Hence, the development process was time-constrained.

The development process was not explicitly documented as a policy document. The principle is, as the responders pointed out, to follow appropriate practices, not to have or follow a policy. The responsibilities in the process are divided such that certain people have certain areas of responsibility. For example one person is main responsible for architecture, whereas other person is responsible for configurability. The responsibilities were not formally defined, but informally they seemed to be clear.

The company had the waterfall process model as a basis for the development process. In practice, the process model was a mixture of the waterfall and iterative models. The specification was done rather straightforward. The development process started to iterate in the design activity. The iterations took place between the design and implementation activities. The development ended up to the iterations because it was facing a threat not to be completed due to the scheduled delivery date.

In the first activity of the software product family development process, the company loosely set the scope and some architectural guidelines. It saw no need for heavyweight process, especially with respect to scoping, because the software product family was developed to the same application domain as the projects earlier, hence the company possessed exhaustive domain knowledge. In addition, the company had already experience on the software product family development.

The concept of the scope is understood in the company as what is included in the software product family. The company sees a strong link between the software product family strategy, scope, and architecture. As the responders pointed out, a strategic decision has to be made on the software product family scope because all possible customers' requirements in the application domain cannot be covered. The software product family scope sets then guidelines for developing the software product family architecture. After the software product family architecture is developed, the situation is the opposite. In practice, the software product family architecture limits the changes in the software product family scope and even in as far as in the strategy. A change in the software product family scope or strategy could cause troublesome or, at least, laborious changes to the software product family architecture. Therefore, if the software product family strategic guidelines, scope or architecture are not correct, do not support each other or communicate with each other, the software product family won't succeed.

Although software product family scoping had succeeded well for the time being, the responders pointed out the software product family scope as one of the most important issues in the software product family approach. This is because they saw that it is hard and expensive to re-scope. They saw that success of scoping was based on the exhaus-

tive informal application domain knowledge. However, they saw that there is always a risk to fall behind changes in application domain or new relevant application domains may emerge. Another concern that came up related to scoping was how to motivate the employee who work with the customers and the customers themselves to give more feedback and influence on the software product family scope.

The prototyping activity followed the scoping activity. In the prototyping activity, the user interface was developed and tested to find out what type of user interface would be the best for the software product family. In the design activity, the software product family architecture and database design were frozen. In the implementation, they then were implemented. The design and implementation were typical software development and driven by the deadline of the delivery.

In the development of the software product family, testing was done during the first delivery. Later testing has been done as new shared assets or revisions of the software product family are released similarly as in the traditional software development. In addition, the prototyping after the scoping is a kind of user interface design testing. A guideline is that every employee is responsible for testing his or her own work and software testing, such as unit, module, sub-system and system testing is done as much as possible in development process.

The software product family is currently in the maintenance. Only some minor changes, bugs and deficiency fixes are done and new interfaces to the existing system in the customer's business environment are developed. The responders pointed out that if the software product family does not work well there is risk that maintenance may become burdensome. They see that in the maintenance, preventing erosion in the software product family architecture, such that architectural elements become depended each other, is the biggest challenge.

4.3.2 Deployment process

In an instance of the software product family deployment process, a customer specific instance of the software product family is developed. Each deployment process is a pro-

ject on its own. In the company, the deployment process is routines in most of the cases. The deployment process takes time from a few weeks to months in total. The time includes implementation of mechanics, electronics and software that are all done parallel as much as possible, but e.g. final testing cannot be done until all parts are completed. The time that the software product family takes is a few days in total, if there is not tailoring. Therefore, it is not a bottleneck in the total process.

The deployment process begins in the sales. The sales representatives negotiate a deal with the customer and capture the customer's requirement based on descriptions of the product. They try to avoid selling products that need tailoring to the software product family. If the customer has requirements that need tailoring to the software product family, the employees in the software development department always review them before accepting the deal. This ensures that nothing is sold, such that it would need laborious changes to the software product family, unless the price corresponds with the work effort needed. The review can take only a moment and one person or it may need more time and a team for thorough analysis.

After both, the customer and the company, agree on the order, the deployment process continues with buying a PC and installing the operating system and other needed auxiliary software. An instance of the software product family is tailored, if needed, and installed on the PC. Finally, an instance of the software product family is configured using a specific configuration tool. For one person it takes about one day to get all software installed and configured, if there is not tailoring. The configuring process itself takes typically three to four hours. Tailoring is usually rather simple. In minimum it can take only 100 person-hours, whereas typically it takes 300-400 hundred person-hours. Figure 9 represents the described deployment process.

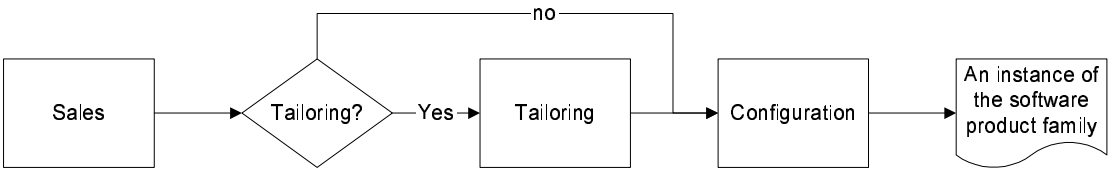

Figure 9 The deployment process in the company.

In testing of an instance of the product family, reuse of software is taken advantages of. An instance of the software product family is based on shared assets that are tested in the development process and hence there is no need to test them again. If there is no tailoring, the company tests only the instance of the software product family with mechanics and electronics. The tests are performed both in the facility of the company and in the customer's facilities. Tailoring assets are tested as they are developed.

All deployment projects are done in the software development department, although the deployment project could be done without programming. In fact, the complexity of the deployment process lies in the understanding the customer's processes and the application domain well enough in its entirety in order to be able to resolve unexpected problems even in customers' facilities. The responders estimated that for a new employee it takes a few months to learn the deployment process and application domain well enough to complete deployment project successfully.

The deployment process is documented in detail. The guide specifies what has to be done starting from installation of the operation system to description of the choices in the configuration. A customer order specification describes customer's order for the configuration and a checklist is used in the configuration to ensure that all values in the configuration are set.

4.4 Organization

The software development department was reorganized at the time the company adopted the software product family approach. The software product family development was separated from deployment at the role level. Software development was still in one organizational unit but earlier all employees had done only deployment due to nature of

project oriented business. Consequently, the reorganization meant establishing roles dedicated to software product family development. The separation of development and deployment is not strict. The employees assigned to software product family development do once in a while deployment projects and the employees assigned to deployment projects also develop the software product family.

There is one clear and formal organizational structure and procedure with respect to the software product family development, *a council* that defines the software product family scope and general guidelines for the software product family development. In the council, there are present different stakeholders such as sales and management in addition of software developers in order to influence to the software product family.

Experienced and committed employees were pointed out to be the most important resources of the software product family. Another important group of people are people who sustain and develop the organization as a pleasant working environment. The responders pointed out that in the software product family based development the most important issues are domain knowledge, ability to work in a team instead of to do things on his or her own and cooperation and communication between people. Other issues are appreciated such as experience in the software engineering, though they are not as important.

The software product family approach had changed the need for resources. First, it is possible to take advantage of generalists and specialists. There are a couple of people, the generalists, who understand the whole domain and the software product family. Another people, the specialist, are expert in some specific area, but do not necessary understand the whole software product family thoroughly. Second, before the software product family there was a need for proportionally greater number of experienced people, whereas currently experience is more balance to the average. Especially in the deployment process, software product family have made experience more balanced. The company needs only a few very experienced employees. The experienced employees can do more challenging work and thus feel their work meaningful and challenging as well as new, novice and inexperienced employees can do more routines work. Figure 10 illus-

trates the idea, but does not represent the real distribution, but a sketch of the situation. It seems that the software product family development benefits the arrangements also such that some people have time to commit themselves to some special area of expertise such as usability.

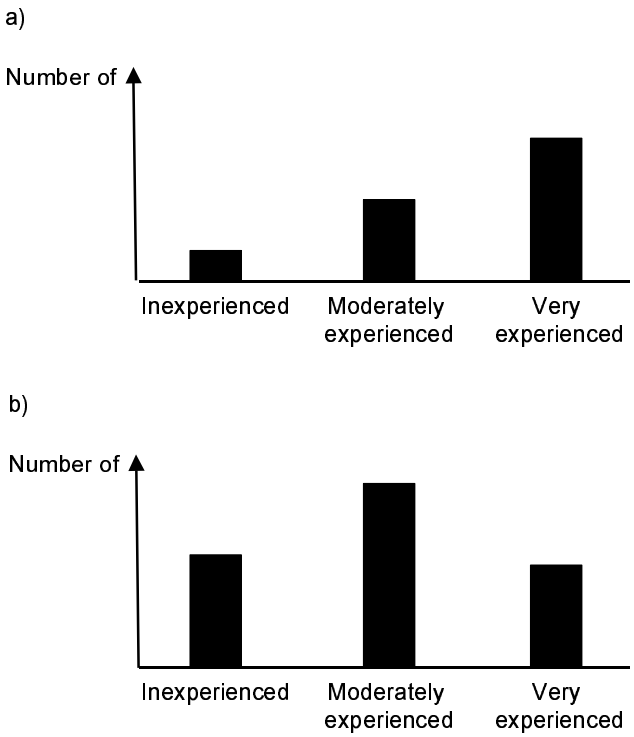

Figure 10 Proportional experience distribution a) before adoption software product family approach, b) after adoption software product family approach

The maintenance of the application domain knowledge is regarded important. The company has tried to maintain knowledge assigning employees who are mainly involved in the software development to the deployment projects regularly to maintain their overall understanding of the software product family and application domain. This procedure also lowers the risk that the product family development alienates from the customers, since the development is not done as close to customer as before in the project business.

The product manager has the final responsibility of the software product family in the company. The emphasis of the responsibilities in the company in general is to make employees committed to their own work. In the development of the software product fam-

ily, the council has the final authority and responsibility to decide about new features and main guidelines.

The success of the software product family in the company did not seem to need as much skilled software engineers as skilled application domain experts. During the interviews the responders never pointed out a need for special skills, tool or methods in the field of software engineering, but pointed out several times that the application domain knowledge and organizational culture are critical.

4.5 Software product family architecture and shared assets.

4.5.1 Concepts

The software product family architecture is understood in the company as a structure, a set of concepts and a set of interfaces that are used in order to unify the understanding among the different stakeholders. The software product family architecture is a mean to communicate the design decisions using the same terms and concepts and to make people understand same things similarly. The responders agreed upon the definition of a software architecture in Bass et. al (Bass et al. 1998), already given in Section 3.6.1, and put emphasis on the internal interfaces and relations inside and between the internal elements in the software product family architecture. The used software architecture style is blackboard architecture. One person is the main responsible for designing software product family architecture, though a few other employees have participated to the design as well.

The software product family is not regarded as component-based software in the company. The coarse-grained building blocks in software product family architecture are called modules and handled as source code. The modules are used without modifications to source code. The modules are developed one software product family in mind, though their usage is not prevented in other products or contexts.

There are about 20 modules in the software product family in total. The size of the modules are typically 30 000 -40 000 lines of code, ranging from the smallest one about 10

000 lines of code to the biggest one about 100 000 lines of code. In total, the size of the software product family is a half million lines of code. Other shared assets are interface descriptions, feature descriptions, user manuals, and other associated documentation.

The software product family architecture documentation does not follow a special procedure or method. Different methods are used flexibly. The requirement specification is documented using natural language. The documentation of the features for customers is in documents using natural language that each describe a feature and its relationships. Technical architecture is documented using natural language, OMT++ use cases, and some parts using UML state machines, sequence diagrams and class diagrams.

The modules are documented using OMT++ use cases, UML state machines, sequence, and class diagrams. The UML state machines are used especially in the documentation of the interfaces. The use cases are used for documenting the user interface. The documentation is done, as the development in general, keeping in mind that it is not essential to follow a policy as long as the things that are needed to be done are done.

As noted earlier, a sound software product family architecture is required for a successful software product family. In the software product family architecture the interfaces between modules has to be proper and stable. The responders pointed out that the stability of the interfaces is more important than stability of the components. They use e.g. wrapping to keep the interfaces unchanged. A challenge is that the software product family architecture is enough extensible and flexible.

4.5.2 Producing

The producing of an instance of the software product family, with respect to software artifacts, starts with taking a copy of the last approved software product family architecture and shared assets from the network drive. This copy becomes an instance of the software product family. First possible tailoring is done to the instance of the software product family, then the instance of the software product family is configured to meet requirements set by customer’s requirements and existing environment. Disabled components are delivered as dead code to customer. Consequently, the software architecture

and shared assets are the same in all instances of software product family such that two instances of the software product family vary only in their configuration unless tailoring has changed them.

4.5.3 Variability

An ability to variability in instances of the software product family is extremely important. The reason for variability is the general diversity of the application domain and diversity of the customers' environments and needs. The coverage of the variability is large ranging from high level functionality to low level parameterization. Quality attributes such as performance or security are not direct subject to variability. In practice, every instance of the software product family differs from each other.

In the software product family, variability can be categorized to predicted and unpredicted. The predicted variability has been designed and in deployment project of an instance of the software product family, this predicted variability is resolved configuring. Unpredicted or unprepared variability means customizations, which have to be made by tailoring. Tailoring is always customer specific, and cannot be configured. However, tailoring does not affect on the configurability of an instance of the software product family such that the same designed variability by configuring remains to be resolved regardless of the tailoring. Therefore, it is also possible to describe configurability separately from the tailoring. The term variability is used to mean variability that is resolved configuring and the term tailoring is used in the context of the variability that cannot be resolved configuring.

The variability in the software product family is divided to horizontal and vertical variability. Variability related to the set of existing systems in customers' business environment is called the *horizontal variability*. The horizontal variability concerns the number of external systems and the types of the external systems. The software product family does not set in practice limitations for the set of external systems, though the physical world does, and therefore allows arbitrary selection of them. The reason for the horizontal variability is that each customer has in practice unique environment of existing systems.

The *vertical variability* concerns parameterization and variability at high level functionality. *Parameterization* customizes an instance of the product family to fit into customers' environment. The parameterization ranges from configuring features such as existence of alarm for errors to atomic parameters such as numeric values specifying physical dimensions. The vertical *variability of high level functionality* is organized to stacks of entities. There are a base, which is in all instance of the software product family, and five stacks each containing a few high level functional entities from which customers can select for their own purposes a suitable combination. Each entity in the stacks adds high level functionality to the system and therefore costs more for the customer. Selection of the functional entities from the stacks is constrained. The order of the functional entities in the stacks specifies a legal combination. If an entity is selected a legal combination is achieved, if and only if also all functional entities below it in the same stack are selected. The company has tried to prevent the dependencies between stacks, in order that selection from each stack would be independent but there is one dependency between stacks.

The idea of the horizontal and vertical variability of high level functionality in the software product family is illustrated in Figure 11. Horizontal variability, a set of external systems, is in the bottom of the picture. The vertical variability of high level functionality is in stacks of entities. A customer can select entities from each stack independently from the other stacks. Only the vertical dimension in the stacks is relevant. If a customer selects for example the entity 1.3, he or she has to select also the entity 1.2 and entity 1.1 from the stack 1. Selection of entity 1.3 does not affect on stacks 2 or 3. He or she can for example select only the entity 2.1 and nothing from the stack 3 as long as the rule concerning each stack is followed.

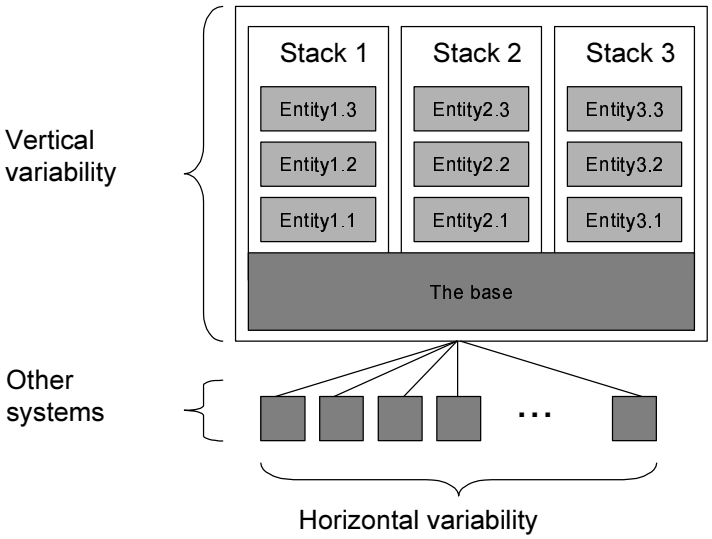

Figure 11 Horizontal and vertical variability in the software product family.

At the architectural level, the variability means that that the higher level functionality or some parts of the software product family architecture are enabled or disabled. In minimum seven, typically about 10, and in maximum all of the 20 modules in the software product family architecture are active in an instance of the software product family. The higher level functionality is associated with modules although there is not one to one relation in the stack so that one module would mean one higher level functionality. The representation of the stacks is not the same as the modules in the software product family architecture. Figure 12 illustrates a simplified picture of the software product family architecture. In instances of the software product family, some of the architectural modules are disabled, but they are as dead code is in instances of the software product family. In the instances of the software product family are also sets of the external systems.

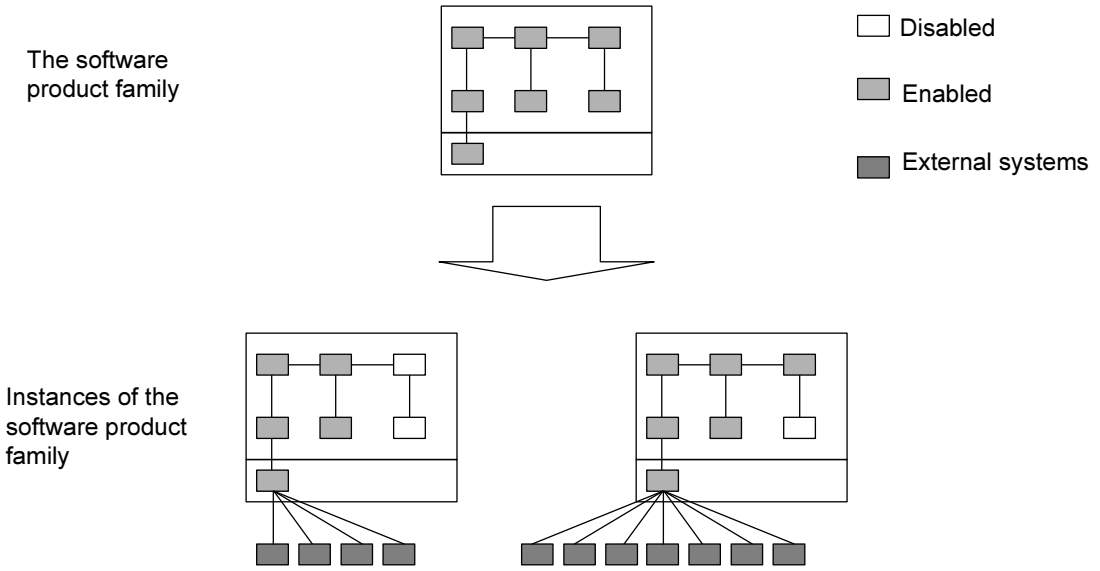

Figure 12 Software product family architecture and instances of the software product family architecture.

Decisions in the variability resolving are optional, alternative and numeric. Optional variability means selecting an item to be included or left out. Alternative means selecting one or more alternatives from a set of alternatives to be included and used in an instance of the software product family. In the alternative case, there are typically about five choices from which the selection has to be made. There is about equal number of both these types of variability. Numeric variability are fields where a value, such as an integer, has to be defined. There is usually a default value for all types of variability and in addition an allowed range defined for the numeric values.

The responders estimated that in variability resolving there are typically 200 atomic, such as yes/no, decisions to be made, even in minimum it takes as much as 100 atomic decisions. Because of the rather larger number of these decisions, practically every instance is different, at least in some value in parameters.

All resolved variability is stored in a configuration database. The higher level vertical variability as well as module parameters are in the same database. Although variability can be distinguished logically as presented earlier to higher and lower level variability, in practice they are all equal in the configuration database. The data in the database is

e.g. Boolean values true or false, values that specifies the selected alternatives or numeric values. When an instance of the product family application is started it reads its configuration from the database and initializes itself accordingly.

The company has developed in-house a configuration tool to resolve variability. In the tool, there is a graphical user interface where is a list of choices and open fields to make variability resolving decisions and explanation of each choice. The tool then writes the configuration the configuration database. The tool does check some, but not entire validity of resolved variability. The person configuring is responsible for making the correct choices on the basis of experience and testing the configuration. The responders had an intuition that there are actually not many dependencies, which would not be obvious from the context and as noted earlier a checklist ensures that all variability is resolved. Dependencies have not been a problem so that a legal and operating configuration would have been hard to make or illegal combination would have been not noticed before it led to troubles.

The differences of the instances of the software product family are in the contents of the configuration database and possible tailoring. Hence, the instances of the software product family are in practice documented in the configuration database. In addition there is a specification that specifies the customer requirements, order and possible tailoring.

Tailoring

As noted, the another type of variability is tailoring that is made based on customers' special requests. Tailoring often concerns interfaces to other customer systems, other than specified in the horizontal variability, in order to exchange data or a report data to a customer specific reporting system. It is usually very simple and routines,

4.5.4 Evolution

Currently the software product family is in the maintenance phase. Changes to the software product family are bug and deficiency fixes, adding interfaces to support other systems in customers' business environment and the software product family is further de-

veloped in some extend. The core technology does not change much in the domain and the core functionality is in the base, hence no large modifications to the base are not needed. Currently the software product family faces larger demands for changes from the architecture of the underlying operating systems than from the application domain.

The council, which was described in Section 4.4, sets guidelines for larger changes. The council makes decision of e.g. new versions of the software product family architecture and shared assets and all significant changes to them. New interfaces to support existing systems in customers' business environment are implemented whenever a need for a new interface raises in a deployment project without approval of the council. The implemented interfaces are put to the software product family as a new shared assets. Another typical smaller evolution takes place as a similar feature has been tailored for several customers. Someone notices that, generalizes the feature, and makes it a part of the software product family. The process is informal and relies on the employees' own initiative.

A delivered instance of the software produce family can change. Possible deficiencies and bugs are corrected in instances of the software product family. Instance of the software product family can be reconfigured at the customer's site or remotely through network connection, but a reconfiguration is rarely done.

Evolution of the software product family is controlled in versions. There is not a version management tool or a repository tool in use. The software product family is versioned and the shared assets and instances of the software product family are associated with a version number of the software product family. The versioning is such that after a shared asset is developed it is maintained compatible with the current and all following versions of the software product family.

In deployment is always used the newest approved version of software product family. The company encourages customers to use the newest version and does not necessarily guarantee to support and deliver new features to the older versions. The customer can get the required feature by updating the product to the same main version as the required feature is developed.

5 Discussion

In this chapter the research instrument and the case study are discussed. In Section 5.1 the case study is compared with software product family definitions and general characteristics of the software product family approach in order to put the study in right context for later discussion. The results of the case study are summarized in Section 5.2. Other empirical research is outlined and compared with the case study in Section 5.3. Validity of the research instrument, research design principles and case study are discussed in Section 5.4 and their reliability in section 5.5. Finally, in Section 5.6 the research is evaluated.

5.1 Context

In this section, the case study is briefly compared with some of the most influential result presented in the scientific literature on the software product family approach. This is done primarily in order to the evaluate whether the results are comparable with other results of empirical studies of software product families.

Above all, it needs to be noted that the studied company does have a software product family, in the sense of the software product family definitions in Chapter 2. There is a set of products that are developed usually by configuring from the software product family architecture and shared assets. The software product family architecture and shared assets realize the commonality in the software product family.

Further, the development of the software product family was planned and considered as an entire family, not only as a set of shared assets to be composed. The development approach was top-down and software product family architecture-centric that are identified to be the way to achieve success (Bosch 2000a; McGregor et al. 2002). The software product family includes shared assets, other than source code or compiled components, which is characteristics of the software product family approach, and there are also three rather clearly separated processes, the development, deployment and management proc-

ess, which is another primary characteristic of the software product family approach (Clements and Northrop 2001).

To summarize, the case seems to be have characteristics that are pointed out to be specific for the software product family approach. Hence, it is reasonable to compare the case company in this thesis with the other results of the other studies on software product families.

5.2 Case study

The most important result in the case study in this thesis has been a detailed description of the state-of-the-practice of the software product family approach in a Finnish company. The studied company had successfully developed a software product family, in which the instances of the software product family are developed very efficiently. The software product family has been in operation for years proving that the success has not been only temporary. The long history of the software product family approach in the company also shows that practice makes masters and the success does not necessary come at once, but needs months or even years of hard work and commitment. The company has not needed any special tools, methods, or processes to develop the software product family, but foremost strong application domain knowledge. In fact, the company used state-of-the-practice methods and tools that are commonly used in the field of software engineering such as UML.

In the company, the management is committed to the software product family approach, but it seemed to be at least equally important to establish and institutionalize the software product family culture through all levels of the organization. This means that the employees in the company understand the special needs of the software product family approach from the software development not only vertically to the management, but also horizontally to the sales and hardware design and construction.

In an important role for the success of the software product family approach has been the software product family architecture that has been scoped successfully and is stable, but enables enough variability. Variability is simplified and constrained effectively yet

flexibly and it has been visualized brilliantly in stacks to the customers in order to show them the customized functionality they are going to get.

The company has been able, in the market pressure, to avoid the pitfall of ending up to fulfil all customers' wishes by changing source code. Instead, the software product family embodies variability that is resolved in an instance of the software product family deployment process by configuring without modifications to source code. Consequently, the company has been able to develop a systematic and automated method for deployment of instances of the software product family that needs rarely and even in those few cases typically only little tailoring to the source code.

5.3 Other empirical studies

In this section the results of the most closely related published empirical studies on software product families are compared with the results of the case study in this work. These include the first Jan Bosch' case studies in (Bosch 1999a; Bosch 1999b), Software Engineering Institute's (SEI) case studies in CelsiusTech Systems (Brownsword and Clements 1996) and Cummins Inc (Clements and Northrop 2001) and Fraunhofer Institut Experimentelles Software Engineering (IESE) Market Maker case study (Clements and Northrop 2001). In addition, the results in this thesis are compared to two software reuse studies (Rine and Nada 2000; Rine and Sonnemann 1998) and (Morisio et al. 2002).

In addition to these studies, there are also other studies that are not included in the comparison. These are later studies that Jan Bosch has conducted with his research group such as product instantiation (Bosch and Högström 2000) and variability (Gurp and Bosch 2001) case studies and the SEI's United States National Reconnaissance office (NRO) Control channel toolkit case study.

Some research has focused on different aspects of the software product family approach than the case study in this thesis and for brevity, these results are omitted from comparison. These include an exploratory case study at Nortel on organizational principles that were believed to be critical for success of software architecture(Dikel et al. 1997) and a

study on product platforms in Philips conducted by Herman Postema and Henk Obbink (Postema and Obbink 2002).

The research method in the some studies is different, hence they are not included in comparison. Peter Knauber, Dirk Muthig, Klaus Schmid, and Tanya Widen (Knauber et al. 2000) from IESE applied in practice, instead of being descriptive study, the PuLSE (IESE 2002) software product family methodology, which they had developed, in six small and medium-size companies. Niemelä and Ihme (Niemelä et al. 2000; Niemelä and Ihme 2001) made a survey on component software. Finally it needs to be noted that there are industry experience reports such as the ESAPS-project (ESAPS 2001), Philips consumer electronics (Ommering 2000; Ommering and Bosch 2002), Boeing (Macala et al. 1996) and Lucent (Coplien et al. 1998).

Jan Bosch' studies

The most significant results from the point of view of the case study in this thesis are probably those produced by Jan Bosch in the case study in Sweden. The case study (Bosch 1999a; Bosch 1999b) was conducted in Axis Communications AB and Securitas Larm AB. The former company produces network based products such as printer-, scanners-, camera-, and storage servers. The products can contain up to 500.000 lines of code. The latter develops security- and safety related product such as fire-alarm, intruder-alarm and passage control systems. The study was an open unstructured interview, though a questionnaire was used to guide the process and, in addition, documentation was analyzed.

In the case study several topics that can be problematic for companies developing a software product family are identified (Bosch 1999a; Bosch 1999b):

- A lack of in-depth understanding of the whole software product family
- Weaknesses of the documentation techniques for reuse oriented software development
- Dependencies between the shared asset such that they are not anymore reusable alone
- A lack of tool support for the software product family approach

- Effort estimation for developing a software product family
- Information distribution in software product family development
- Multiple versions of the same shared asset hence one implementation can be used only in few products
- Use of shared assets in other context besides they were originally defined or
- Lack of management support for the software product family approach

A lack of in-depth understanding of the whole software product family was identified at least as a challenge, if not an actual problem by the responders in the case study in this thesis. In (Bosch 1999b), documentation that is more extensive and carefully designed interfaces for the software product family architecture are recommend to solve the problem. The responders did not saw a need for more extensive documentation, but agreed with the need for proper and stable interfaces. The responders pointed out the weaknesses of the documentation techniques to support reuse, variability and evolution in the software product family, but they did not see it as a big problem. In the case company, there were dependencies between the shared assets that were argued to be primarily causes of evolution and nature of the products. The dependencies were not regarded especially harmful, but rather inevitable to some degree in evolving software. The responders did not directly point out the tool support for a software product family as a problem except that the imperfect documentation techniques naturally also lacked a tool support. However, the company had developed its own tool for configuration, which is indirect evidence for the need for tool support. Although the company was satisfied with the tool, the company might not have needed to develop such a tool in-house if there were better general purpose tool support for software product families.

The problem with the effort estimation in software product family development process was avoided in the company in this thesis probably because they knew what they were developing due to the extensive application domain knowledge and the evolutionary software product family adoption approach. Although the company ended up to iterations in the development process, it seemed to have succeeded well in essential parts of the development process, such as scoping, and was capable to finish the development on time. In addition, the company had practiced the software product family approach in the

proceeding software product family that probably gave realistic impression on the effort needed.

The information distribution was not a problem for the company probably due to the rather small number of employees. Nevertheless, the responders admitted that the company seemed to be on the threshold that the information distribution might become a problem if it recruits more employees and the number of employees would increase. The shared assets were adjusted to instances of the software product family only by configuring and there was only one implementation of them, which was used in all instances of the software product family instead of being replaced by another implementation. Hence, there were not multiple versions of the shared asset that could even have caused problems. The company had used the shared asset only in the software product family and did not have experiences of their use in other context and this was not a problem. Finally, the software product family did not lack the management support; rather it had strong support and commitment from the management.

The identified characteristics of the software product family architecture and shared asset in the case study in this thesis are similar as in the industry in general according to (Bosch 1999a). The characteristics are:

- Mostly conceptual understanding of architecture. Minimal explicit definition.
- The of the software architecture understanding is informal
- The software architecture is not documented in detail, but only as much as is needed and felt important
- The shared assets are large and developed internally.

Software Engineering Institute

Software Engineering Institute (SEI) at the Carnegie Mellon University has conducted a case study on software product families at CelsiusTech Systems in Sweden (Brownsword and Clements 1996). A products in CelsiusTech are integrated systems unifying all weapon systems, command-and-control, and communications functions on a warship. A product consists 1-1.5 millions lines of Ada code. The number of employees

has varied in CelsiusTech between 20 and 200 such that in the end of the case study there were about 60 employees.

The data for the case study were based on the researchers' previous experiences on the company, previously published reports of the company, documentation analysis, and interviews. The interviews took three days and included more than two dozen employees. The study ended on a seminar of the major findings on the fourth day. CelsiusTech reviewed the publications before they were published to ensure that confidential information was not published.

The results of the CelsiusTech case study (Brownsword and Clements 1996) resemble largely the results of the case study in this thesis. In both studies, the development approach had been earlier more project-oriented that was then changed to the software product family approach. In both companies the software product family approach emphasize strongly the software product family architecture but also the business, process and organizational concerns were pointed out to be important. The most remarkable difference is probably the systematic deployment process and configurability of the software product family that the company in this case study has been able to establish. The software product family in CelsiusTech includes 3000-5000 parameters that are implemented as symbolic values to adjust the components. In practice, the parameters are often left unchanged and there is no systematic management for them. The case study in this thesis, although its software product family does not include as many parameters, has isolated these kinds of parameter values to a separate database and developed a configuration tool to manage them. In addition, many parameters are actually changed for instance of the software product family.

The results of Cummins Inc case study (Clements and Northrop 2001) show that success factors for their software product family are e.g. right organizational culture, a software product family champion, focus on the software product family architecture and application domain knowledge. These are the same as characteristics identified in the case company in this thesis. Similarly, in both companies, at the time the software product family approach was adopted, they did not see any other possibility to continue business

in a profitable way. Of the success factors and characteristics at Cummings the most striking difference when compared with this case, is that in Cummins Inc. the shared assets are kept under a disciplined configuration management. In the case company in this thesis, the shared assets management is rather informal. This approach is successful probably because there are few developers working on same assets, that the developers work close together at same floor, regularly communicate and are able for co-operation, and each developer have own part of the software product family, such as user interface, that they alone, or in maximum with only a few other developers, develop.

Fraunhofer Institut Experimentelles Software Engineering

Fraunhofer Institut Experimentelles Software Engineering (IESE) did a case study on product families at Market Maker (Clements and Northrop 2001). Market Marker is a small German company that develops products specialized in stock market data. The software product families in the case study in this work and in Market Maker are about the same size measured in lines of source code. The architectural styles in the software product family architectures of both are very different. Market maker software architecture that resembles three tier style and the company in this study has a blackboard style software architecture. Market Maker uses a lot legacy code unlike the company this thesis, which does not use at all legacy code. Creating the right organizational culture in both companies has been a central success factor. In Market Maker, instances of the software product family are developed in as little as three days by changing property files, but there was no tool for that. The software product family in this work exhibits similar configurability, and in addition a tool support that can be used to configure and deploy an instance of similar complexity even faster. In fact, it can be done in a day compared with the three days in Market Maker.

In the Marker Maker case study it was concluded that the following five ingredients are necessary for a successful software product family approach:

- A software product family champion, who has a vision of the software product family, communication skills and management commitment

- The current software development approach seems not to be successful and there is a pressure for a change
- Long expertise in the application domain
- Focus on the software product family architecture
- Management commitment

These ingredients exist also in the company studied in this thesis. In both companies, the champion was the software architect. The pressure for the software product family approach in both companies was that they could not continue the business profitable and reasonably by the traditional way. Market Maker has a decade long expertise and the case company in this thesis had several decades long expertise in the application domain. Both companies emphasize the importance of the software product family architecture. In Market Maker, the management is so close to software development and it was the management that championed the software product family. In the case company in this thesis there was clearly management support for the software product family but a clear evidence on how that was achieved originally was not clarified. At the present, the software product family approach shows benefits such as faster lead time and better quality that the management commitment seems to evident.

Software reuse

David Rine with his research colleges have conducted two studies on the software reuse. In the first (Rine and Sonnemann 1998), they defined potential success factors based on a literature, downsized the number of success factors and developed a questionnaire. 3750 copies of the questionnaire were distributed and the data were then analyzed statistically. In the follow-up study (Rine and Nada 2000), they developed a reuse reference model, of which correctness was then empirically studied. The study consisted of 24 case studies and a survey consisting of 34 questions. The leading indicators for software reuse capability in (Rine and Nada 2000) are:

- Product family approach
- Software architecture which standardizes interfaces and data formats
- Common software architecture across the product family

- Design for manufacturing approach, i.e. develop parts that can be reused
- Domain engineering, management which understands reuse issues
- Software reuse advocates in senior management
- State-of-the-art tools and methods
- Precedence of reusing high level software artifacts such as requirements and design versus just code reuse
- Trace end-user requirements to the components that support them

The only contradiction, compared with the result of the case study in this work, is the argument that one of the leading indicators for software reuse capability is the state-of-the-art tools and methods. The case company had not started to use the state-of-the-art tools and methods, though it was well aware of them. It used tools that could rather be characterized as the state-of-the-practice tools. There are not enough detailed data to evaluate whether the indicator "trace end-user requirements to the components that support them" hold true.

Recently, Morrisio *et al.* (Morisio et al. 2002) made a study on success and failure factors in software reuse. The study was conducted as a structured interview in 24 projects in 19 companies, which had participated software process improvement experiments funded by European Commission. The researchers read the reports on the experiments and other material available of the companies. In the interviews, the interview questions were given to a responder and the interviewer read the questions and took notes on the answers. The interviews took between two and three hours. After the interviews, the interviewer wrote a report that was given to the responders in order to be validated. Finally the data were coded and analyzed.

According to (Morisio et al. 2002), successful reuse requires initiation of reuse processes, modification to non-reuse processes and addressing of human factors. In order to initiate successfully reuse it has to be considered as a technology transfer that requires top management commitment. These all are similar to the events that had taken place in the company studied in this thesis. At the time the software product family approach was adopted, the company changed its organizational structure and initiated software product

family development process. The software product family had management support as it was quite large initial investment and a conscious change. The current software product family approach shows that the processes throughout the organization have been modified. An example of this is the sales process, which is constrained such that they are not allowed to sell such products that cause troublesome changes to the software product family. Finally, the importance of the human factors was emphasized several times during the interview.

5.4 Validity

The validity of the case study is discussed in this section. Because developing the research instrument was one of the main objectives of this thesis, the aspects of the validity related to it are given special attention. According to (Yin 1994), the following validity tests are commonly used in empirical research:

- Construct validity
- Internal validity
- External validity

5.4.1 Construct validity

The *construct validity* means that the developed operational measures are correct and valid abstractions for the real world under study (Yin 1994). The objectives of this thesis were to construct a research instrument, and therefore this is the most important aspect of the validity. The following issues with respect to construct validity are discussed in more depth:

- The structure and content of the initial theory
- Constructing the interview questions
- Multiple sources of evidence
- The validation session
- Responders' review on results

The structure and content of the initial theory

To ensure that the preliminary understanding of the subject of the study was right, the results published in literature related to software product families in the most influential and established sources were investigated.

For the structure of the initial theory, there are different established decomposition principles to divide the software product family approach. In (Bosch 2000a) three such decomposition principles are identified:

- System, architecture and component
- Development, deployment and evolution
- Business, architecture, process and organization

The first principle divides a software product family on the basis of the artifacts it consist of. The second principle decomposes a software product family with respect to the lifecycle of the software in a software product family. Software can be considered to be first developed for reuse, then deployed and reused and finally it may change in the course of time i.e. evolve. The third principle decomposes a software product family according to different concerns or views in the organization.

The third decomposition is probably the most used in other research such as (America et al. 2000). It was also applied as the highest level decomposition principle for the research instrument in this study with the following three exceptions

- The term “architecture” was replaced with the term “product”
- The process and organization concerns were combined and handled under the process concern in the interview questions
- The order of the concerns was business, process and organization, and product.

The term “architecture” was changed to the term “product” in order to distinguish it from the software product family architecture and to cover all the artifacts related to the product family. The process and organization concerns were combined, because they seemed to be closely related. The order was selected to be such that business concern was dealt with first in order get an overview understanding of the company, products

and its operation, on the basis of which a more detailed investigation could be carried out. Second, process and organization concerns were dealt with in this order. It was assumed that the processes are more important and have to be understood before the organization can be understood. Third, the product concern was dealt with most in-depth. Consequently, the order follows a top-down principle, such that concerns are investigated with respect to the first concern more cursory and later in more depth as the understanding of the software product family as a whole deepened.

The other decomposition principles in (Bosch 2000a) are used to further decompose the business, process and organization and product concerns as follows:

- The process concern was further decomposed according to development, deployment, evolution dimension
- The product concern was first decomposed according to system, architecture, component dimension, and these were divided according to the development, deployment and evolution dimension

Consequently, to ensure the content of the initial theory it was based on the most influential literature to form the theoretical background for the study. Similarly, the structure followed results on software product families. This was done to ensure that the study covered the right issues on the software product family approach extensively enough.

Constructing the interview questions

The means to acquire the data were primarily interviews. Hence, the interview questions were in practice the operational measures to measure and abstract the phenomena of the real world under study

The interview approach was semi-structured interview that was applied such that the interview questions were developed beforehand and were to be followed as strictly as would be meaningful during the interviews. There seems not have been an empirical study from the point of view taken in this study and neither the precisely same case study approach seems not been not used in other research on software product families or closely related fields such as software reuse or component software. The other em-

pirical results on the software product family approach are primarily based on the studies conducted in more loosely structure interviews and experience reports that representatives of companies, from which the results are, have written. Structured interviews and mail surveys were used in the reuse and component oriented studies. Hence, constructing a new research instrument was reasonable and the only possibility as the author was not aware of a similar research instrument that would have been available and could have been taken as a starting point. In fact, there was not available such research instruments even for cursory comparison. Similarly, comparison with the other research instruments is not reasonable, because of different coverage, duration, methodological approach, and size of samples of the other research (Kitchenham and Pfleeger 2002a).

During the interviews, clarifications on unclear questions were possible done and the terms were sometimes changed to match the terminology used in the company. The flexibility of this approach proved to be important because of the lack of a common and standardized terminology and common understanding of software product family concepts in the companies. Hence, the structured interview approach would probably have failed. Other possible approaches would have been e.g. an unstructured interview approach or a semi-structured interview approach using only a sketch of the themes (Hirsjärvi and Hurme 2001). Applying these approaches would probably have resulted that more shallow and less extensive data would have been captured in the time frame of the interviews. It did not seem to be possible to spend more time in the company for the interviews. It was hard to schedule the interview with the responders. Consequently, the selected interview approach seems to best suitable for the study.

The developed interview questions were mostly open-ended and qualitative. The responses to the few quantitative interview questions were usually based on estimations and mostly it seemed that there was no measured data to be had significance to the results. The open-ended interview provided means for responders to explain their answers that they also did. Therefore, the selected approach to use qualitative and open-ended questions seems to have been best suitable. In fact, although the information from the close-ended and quantitative interview questions was valuable, the reasoning that the responders used to come at the answer and their explanations provided an essential

background and a context for them. Finally, the interview questions were evaluated, as described in Chapter 1, in the evaluations in order to ensure that they were understandable and the responses were correct.

The issues described above to construct and use the interview questions seemed to be successful and it seems reasonable to assume that the questions succeeded in measuring the subject under study properly. Hence, it seemed that the construct validity of the study should be ensured with respect to design principles for the research instrument and its use.

Multiple sources of evidence

The semi-structured interview was combined with an analysis of documentation. This approach increased the construct validity as being a form of triangulation (Yin 1994). On the other hand, the used division to the business, process and organization and product concerns encouraged and enabled responders of different profiles and reflected different views on the software product family. This increased the validity as multiple sources described the same issues and widened the perspective of the research.

The interviews were group interviews, in which there was more than one person present at a time. The group interview approach was used in order to get maximal amount of data given the limited time for interviews from the very busy key persons in the company. In addition, it was also evident that the responders answered the questions in more depth as they could comment and correct its other's answers. However, there is unfortunately a threat that the responders may have not been inclined to reveal some issues such as own mistakes, own critical opinions or the facts were presented in a better light than was actually the case. In order to mitigate the effects, an effort to create a trusting and open atmosphere was made.

Validation session

The validation session was held in the company in order to validate and assure that the important preliminary findings were correct. Relatively little was collected during vali-

dation session, but the acquired knowledge on the software product family approach applied in the company enabled a more in-depth and open discussion. The validation session also showed that the understanding gained is mostly correct, as relatively few misunderstandings were corrected.

Responders' review on results

The case report was also sent to the company in order to be read. Similarly as in the validation session, the company had an opportunity to correct misunderstandings, though they did not point any.

5.4.2 Internal validity

The internal validity considers that the established causal relationships are correct. The case study design in this work was descriptive and it was not even meant to establish such causal relationships that in the prevailing conditions lead from one circumstance to another circumstance. Consequently, it is not relevant to discuss the internal validity with respect to this research as noted also in (Yin 1994).

5.4.3 External validity

The external validity concerns how well results of an empirical study can be generalized (Yin 1994). The issues with respect to external validity are:

- The research is not designed for a particular application domain, country or type of company and product
- The research has been made in five companies and the selection of the companies was done purposefully

The software product family approach was noted to be independent of application domain, country or type of company and product in Chapter 3. In the research, it was presumed only that the company develops products that include software and the development approach at least resembles the software product family approach although it necessary was not based on the principles outlined in the literature. The selection of the companies for the research was done purposefully to find out contemporary state-of-the-

practice, rather than state-of-the-art, in the industry. The selection did not focus on certain types of products, companies or particular application domains. Therefore, the result of this research should not present only e.g. best the practices or a market niche.

The first limitation is that the study is limited, at least heretofore, to the Finnish industry where are quite few very large software development organizations and the software development organization in the case study was rather small. The second limitation is that as the research concentrated mostly on contemporary phenomena, it lacks in-depth description of the adoption and evolution of a software product family.

5.5 Reliability

According to (Hirsjärvi and Hurme 2001), there are three ways to ensure reliability:

- Two times conducted same research outcomes the same result
- Two researchers end up to the same result
- Two methods outcome the same result

No one of these was orthodoxly done, primary because it was not reasonable because of the given resources for the study. However, there were several researchers that participated the study although they did not analyze or conduct the study independently. The data was gathered in three phases, in the interview, in the validation session and as the company read the report but these were not independent of each other and the latter two produced only little data.

In (Yin 1994) is noted that reliability is best achieved if each step of the research is carefully documented and the data are collected to a case study database. In this thesis was described the interview, validation session and analyzing the case study data. The interview was transcribed, separated from the interpretations and conclusions, stored and attached with meta-data to be easier manageable in the analysis and latter checks for accuracy of the results. The research instrument that was used to capture the data was evaluated in two different phases before the actual research in order that it assure that it covers the right issues and the interview questions are understandable and are organized logically.

5.6 Research evaluation

5.6.1 Evaluation of objectives

The objectives of the case study were outlined in the research topics in Section 1.3. These objectives and the results of the case study are compared in order to evaluate the success of the case study and construction of the research instrument with respect to its ability in eliciting the wanted information.

The first research topics was “The concept and utilization of software product families”. It was found out in the case company how the concept of the software product family was understood, characteristics of the software product family and use of it. Consequently, the research instrument succeeded well with respect to this topic.

The second research topic was “The methods and tools used to capture information on software product families and the information that exists on software product families”. The research topic was clarified rather well. However, a weakness was that although the used methods became clear, the specific use of them and the reasons why they were used in the specific way were not clarified extensively. This was due to limited time for interviews and document analysis.

The third research topic was “The processes supporting the creation, utilization, management and maintenance of software product family information”. The nature of the twofold process and the management process were found out relatively well but the research instrument succeeded only partly in finding out the process aspects. The weakness was that the understanding of the maintenance process remained somewhat shallow. This was probably was due to limited time for interviews and document analysis.

The fourth research topic was “business aspects of software product families”. The business aspects were clarified to some extend in the case. Some of the business aspects such as quantitative analysis of reuse economics would have required time consuming estimations and calculations that seemed to be hard or impossible to be obtained using the se-

lected research approach. However, it is reasonable to claim that this research topic was clarified well enough with respect to research objectives.

To summarize, the developed research instrument can be considered successful. Although some of the topics did not become as clear as could be hoped, already one case of the multiple case study gave valuable information. The case confirmed the findings of several other empirical studies while bringing into light new interesting issues with respect to a small software development organization in Finland. A conclusion is that more time, e.g. a half or even an additional whole day, should be used for the interviews and document analysis to elicit more precise information.

5.6.2 Summary of validity and reliability

To ensure construct validity most influential literature was used as the preliminary theory of the research. Existing research instrument could not be used. Data were based on interview, validation session and documentation analysis. The responders had two possibilities to review to results during the validation session and while reading the case study report. External validity was ensured in the design phase as no special limitation were not defined. Internal validity is not relevant for this kind of descriptive case study. Documenting in detail how the case study was conducted, the initial theory and the interview question, evaluation of the research design and in-depth description of the company case was done to ensure reliability.

6 Conclusions and Future Work

In this work, a research instrument for a descriptive multiple case study of the state-of-the-practice of software product families was developed and one of five case companies studied using the research instrument. A software product family is understood to consist of a software product family architecture and a set of shared assets that are designed for incorporation into the software product family architecture. In addition, the software product family consists of products that are called instances of the software product family and developed using the software product family architecture and shared assets.

The research instrument consists of an initial theory that illustrates relevant theoretical issues and interview questions. It is decomposed into business, process, organization, and product viewpoints to software product family development. The interview questions are mostly qualitative and open-ended. The research instrument was evaluated in two phases, in an evaluation that resembled a focus group study and in a pilot study, before it was used in case study.

The studied case company had adopted the software product family approach. The instances of the software product family that satisfy the requirements of an individual customer are produced in most cases by configuring the software product family using the shared assets without modifications to the source code. A sound software product family architecture, the right software product family scope, commitment through organization, and strong application domain knowledge seemed to be the main success factors for the software product family approach in this company.

The construction of the instrument succeeded rather well. The selected design approach for the research was successful in eliciting the intended information. It seems that an extensive understanding of the important software product family related issues by using the constructed the constructed research instrument was able to be captured.

In developing the research instrument and conducting the research in the case company, it became clear that a software product family is a complicated and diverse concept, in

which several points of view have to be taken account. A software product family is not just a technical software engineering paradigm but influences many aspects on how operations in a company are organized and run. The software product family approach, when applied successfully and efficiently, influences many stakeholders in the company in addition to software engineers such as sales persons and managers.

The case study results increase the credibility of promises the software product family approach has made. However, there is a need for more empirical research, especially in-depth explanatory case studies to properly understand the applicability of this approach and evolution in a software product family. It also seems that there are needs for tools and methods for defining and modeling the software product family scope and to manage variability and evolution in a software product family.

.

7 References

America, P., Obbink, H. J., Ommering, R. van and Linden, F. van der "CoPAM: A Component-Oriented Platform Architecting Method Family for Product Family Engineering", in *Proceedings of the First Software Product Line Conference*. Kluwer Academic Publishers. 2000.

Anastasopoulos, M. and Gacek, C. "Implementing product line variabilities", in *Proceedings of Symposium on software reusability, SSR'01*. ACM Press. 2001.

Bachmann, F. and Bass, L. "Managing Variability in Software Architectures", in *Proceedings of Symposium on software reusability, SSR'01*. ACM Press. 2001.

Basili, V. R. "The Experimental Paradigm in Software Engineering", *Lecture Notes in Computer Science*, vol. 706. 1992.

Bass, L., Clements, P. and Klein, D. V. *Software architecture in practice*. Addison-Wesley. 1998.

Beck, K. "Embracing Change with Extreme Programming", *Computer*, vol. 32(10) 1999.

Bosch, J. "Evolution and Composition of Reusable Assets in Product-Line Architectures: A Case Study", in *Proceedings of the First Working IFIP Conference on Software Architecture*. 1999a.

Bosch, J. "Product-Line Architectures in Industry: A Case Study", in *Proceedings of International conference on software engineering, ICSE'99*. ACM, Los Angeles. 1999b.

Bosch, J. *Design and use of software architectures - adopting and evolving a product-line approach*. Addison-Wesley. 2000a.

Bosch, J. "Organizing for Software Product Lines", *Lecture Notes in Computer Science*, vol. 1951. 2000b.

Bosch, J. "Software Product Lines: Organizational Alternatives", in *Proceedings of the International Conference on Software Engineering, ICSE2001, Toronto, Canada*. IEEE computer society, Toronto, Canada. 2001.

Bosch, J. "Maturity and Evolution in Software Product Line: Approaches, Artifacts and Organization", in *Lecture Notes in Computer Science 2379, Proceedings of the second software product line conference*. Springer-Verlag, Berlin Heidelberg. 2002.

Bosch, J. and Högström, Ms "Product Instantiation in Software Product Line: A Case Study", in *Proceedings of the Second International Symposium on Generative and Component-Based Software Engineering (GCSE 2000), October 2000.* 2000.

Brownsword, L. and Clements, P. *A case study in successful product line development.* Report Technical report CMU/SEI-96-TR-016 1996

Böckle, G., Muñoz, J. B., Knauber, P., Krueger, Charles W., Leite, J., Linden, F. van der, Northrop, L. M., Stark, M. and Weiss, D. "Adopting and Institutionalizing a Product Line Culture", in *Lecture Notes in Computer Science 2379,* Springer-Verlag, Berlin Heidelberg. 2002.

Card, D., Comer, E. "Why do so many reuse programs fail?", *IEEE Software*, vol. 11(5) 1994.

Clements, P "On the importance of product line scope", in *Proceeding of the 4th international workshop on product family engineering*. European software institute. 2001.

Clements, P. "Being Proactive Pays Off", *IEEE Software*, vol. 19(4) 2002.

Clements, P. and Northrop, L. M. *Software Product Lines : Practices and Patterns*. Addison-Wesley Pub Co. 2001.

Cohen, S. "Predicting when product line investment pays", in *Proceeding of the 2nd International Workshop on Software Product Lines: Economics, Architectures and Implications*. 2001.

Coplien, J., Hoffman, D. and Weiss, D. "Commonality and Variability in Software Engineering", *IEEE Software*, vol. 16(6) 1998.

Czarnecki, K. and Eisenecker, U. W. *Generative Programming*. Addison-Wesley. 2000.

Di Nitto, Elisabetta and Fuggetta, Alfonso "Product lines: what are the issues?", in *Proceeding of teh 10th International Software Process Workshop (ISPW '96). Process Support of Software Product Lines.*. 1996.

Dikel, D., Kane, D., Ornburn, S., Loftus, W. and Wilson, J. "Applying software product-line architecture", *Computer*, vol. 30(8). 1997.

ESAPS Experiments. European Software Institute, Report ESI-WP4-0105-01 2001

Fafchamps, D. "Organizational Factors and Reuse", *IEEE Software*, vol. 11(5). 1994.

Foddy, W. *Constructing questions for interviews and questionnaires*. Cambridge University Press. 1993 .

Gurp, J. van and Bosch, J "On the Notion of Variability in Software Product Lines", in *Proceedings of WICSA 2001*. 2001.

Hirsjärvi, S. and Hurme, H. *Tutkimushaastattelu*. Yliopistopaino, Helsinki. 2001.

IESE. *PuLSE homepage.* Fraunhofer Institut Experimentelles Software Engineering. Referred 2002-12-18, Available at http://www.iese.fhg.de/PuLSE/. 2002

Jacobson, I., Griss, M. and Jonsson, P. *Software Reuse*. ACM Press. 1997.

Kitchenham, B. A., Pfleeger, S. L. "Software Quality: The elusive target", *IEEE Software*, vol. 13(1). 1996.

Kitchenham, B. A., Pfleeger, S. L. "Principles of Survey Research Part 3: Constructing a Survey Instrument", *ACM SIGSOFT Software Engineering Notes*, vol. 27(2). 2002a.

Kitchenham, B. A., Pfleeger, S. L. "Principles of Survey Research Part 4: Questionnaire evaluation", *ACM SIGSOFT Software Engineering Notes*, vol. 27(2). 2002b.

Knauber, P., Bermejo, J., Böckle, G., Leite, J, Linden, F. van der, Northrop, L. M., Stark, M. and Weiss, D. "Quantifying Product Line Benefits", in *Proceeding of the 4th international workshop on product family engineering*. European software institute. 2001.

Knauber, P., Muthig, D. and Widen, T. "Applying Product Line Concepts in Small and Medium-Sized Companies", *IEEE Software*, vol. 17(5). 2000.

Knauber, P., Succi, G. "Perspective on Software Product Lines", *ACM SIGSOFT Software Engineering Notes*, vol. 27(2). 2002.

Krueger, C. W. "Eliminating the Adoption Barrier", *IEEE Software*, vol. 19(4). 2002.

Linden, F. v. d. "Software Product Families in Europe: The Esaps and Café projects", *IEEE Software*, vol. 19(4). 2002.

Macala, R. R., Stuckey, L. D. Jr. and Gross, D. C. "Managing domain-specific, product-line development", *IEEE Software*, vol. 13(3). 1996.

MacGregor, J "A Product Line Process for Production of Platform Software at Bosch", in *Proceedings of Software Product Lines: Economics, Architectures, and Implications*, 2000.

McGregor, J., Northrop, L. M., Jarrad, S. and Pohl, K. "Initiating Software Product Lines", *IEEE Software*, vol. 19(4). 2002.

Meyer, M. and Lehnerd, A. P. *The power of product platforms: building value and cost leadership*. The Free Press. 1997.

Meyer, M., Selinger, R. "Product Platforms in Software Development", *Sloan Management Review*, vol. 40(1). 1998.

Moilanen, T. and Ruohonen, S. *Kvalitatiivisen aineiston analyysi ATLAS/ti-ohjelman avulla*. Kuluttajatutkimuskeskus, Menetelmäraportteja ja käsikirjoja 2/1994, 1994

Morisio, M., Ezran, M. and Tully, C. "Success and Failure Factors in Software Reuse", *IEEE Transactions on software engineering*, vol. 28(4). 2002.

Männistö, T., Soininen, T. and Sulonen, R. "Configurable software product families", in *Proceedings of ECAI 2000 Configuration Workshop*. 2000.

Niemelä, E. and Ihme, T. "Product Line Software Engineering of Embedded Systems", in *Proceedings of Symposium on software reusability, SSR'01 2001*. ACM Press. 2001.

Niemelä, E., Kuikka, S., Vilkuna, K., Lampola, M., Ahonen, J., Forssel, M., Korhonen, R., Seppänen, V. and Ventä, O. *Teolliset komponenttiohjelmistot.* Report Teknologiakatsaus 89/2000 2000

Northrop, L. M. *A Framework for Software Product Line Practice - Version 3.0.* SEI - The Software Engineering Institute at the Carnegie Mellon University. Referred 2002-12-12, Available at http://www.sei.cmu.edu/plp/framework.html.

Northrop, L. M. "SEI's Software Product Line Tenets", *IEEE Software*, vol. 19(4). 2002.

Ommering, R. van. "Beyond product families: Building a product population?", *Lecture Notes in Computer Science*, vol. 1951. 2000.

Ommering, R van "Configuration Management in Component Based Product Populations", in *Proceeding of the Tenth International Workshop on Software Configuration Management (SCM-10), Toronto, Ontario, Canada*. 2001.

Ommering, R. van and Bosch, J. "Widening the Scope of Software Product Lines - From Variation to Composition", in *Lecture Notes in Computer Science 2379, Proceedings of the second software product line conference*. Springer-Verlag, Berlin Heidelberg. 2002.

Parnas, D. L. "On the Design and Development of Program Families", *IEEE Transactions on software engineering*, vol. 17(4). 1976.

Patton, M. Q. *Qualitative evaluation and research methods*, 2 ed. SAGE. 1990.

Postema, H., Obbink, H. J. "Platform based product development", *Lecture Notes in Computer Science*, vol. 2290: 2002.

Rine, D. C., Nada, N. "Three empirical studies of a software reuse reference model", *Software - Practice and Experience*, vol. 30(6). 2000.

Rine, D. C., Sonnemann, R. M. "Investment in reusable software. A study of software reuse investment success factors", *Journal of Systems and Software*, vol. 41(1). 1998.

Sarcous. *Sarcous project www-pages.* Sarcous-project. Referred 2002-12-20, Available at http://www.soberit.hut.fi/sarcous/english/index.html. 2002

SEI. *How Do You Define Software Architecture?* SEI - The Software Engineering Institute at the Carnegie Mellon University. Referred 2002-12-12, Available at http://www.sei.cmu.edu/architecture/definitions.html. 2002a

SEI. *Software Product Line Hall of Fame*. SEI - The Software Engineering Institute at the Carnegie Mellon University. Referred 2002-12-16, Available at http://www.sei.cmu.edu/SPLC2/SPLC2_hof.html. 2002b

Soininen, T. *An approach to knowledge representation and reasoning for product configuration tasks*, Helsinki University of Technology. Acta Polytechnical Scandinavica, No. 111. 2000.

Sommerville, I. *Software Engineering*, 5th ed. Addison Wesley. 1995.

Svahnberg, M. and Bosch, J. "Issues Concerning Variability in Software Product Lines", in *Proceedings of the Third International Workshop on Software Architectures for Product Families*. Springer Verlag. 2000.

Szyperski, C. *Component Software*. ACM Press. 1999.

Tiihonen, J., Soininen, T., Männistö, T. and Sulonen, R. "Configurable products - Lessons learned from the Finnish industry", in *In Proceedings of second International Conference on Engineering Design and Automation (ED&A '98)*. 1998.

Weiss, D. and Lai, C. T. R. *Software product-line engineering: a family based software development process*. Addison Wesley. 1999.

Wijnstra, J. G. "Critical Factors For a Successful Platform-Base Product Family Approach", *Lecture Notes in Computer Science* , vol. 2379: pp. 68-89. 2002.

Yin, R. K. *Case study Research*, 2 edn. Sage. 1994.

Appendix

The appendix contains the interview questions for the software product family case study