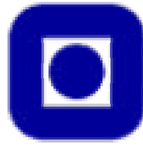
NORGES TEKNISK-NATURVITENSKAPELIGE UNIVERSITET
FAKULTET FOR INFORMASJONSTEKNOLOGI, MATEMATIKK OG ELEKTROTEKNIKK



HOVEDOPPGAVE

Kandidatens navn:	Øivind Wang
Fag:	Systemutvikling, Datateknikk
Oppgavens tittel (norsk):	En Studie av Industriell, Komponent-Basert Utvikling, Mogul
Oppgavens tittel (engelsk):	A Study of Industrial, Component-Based Software Engineering, Mogul

Oppgavens tekst:

This thesis aims at suggesting improvements to the current practice at Mogul. Through reengineering an old system Mogul has implemented a solution based on reusable components. The components are now to be reused in other systems, leading to problems related to component-based software engineering and software product lines. The suggestions should be based on a characterization of the current practice at the company, and a literature study of the relevant state-of-the-art. A survey should also be conducted to explore the motivation for reuse and component-based development, and how the current development process is viewed by the employees. In addition, a case study should be performed to see the value of component-based software engineering in an industrial context.

Oppgaven gitt:	27. januar 2003
Besvarelsen leveres innen:	23. juni 2003
Besvarelsen levert:	20. juni 2003
Utført ved:	NTNU, Gløshaugen, Trondheim / Mogul, Trondheim
Veileder:	Reidar Conradi

Trondheim, 20. juni 2003

Faglærer
Reidar Conradi

Abstract

Title: A study of industrial, component-based software engineering, Mogul.

Background: When adapting to component-based software engineering, the introduction of a software product line should be performed to optimize the level of reuse. The organization discussed here has reengineered an old monolithic architecture to a solution based on components. The components are meant to be reused in several different products, leading to considerations on how to manage the variability, changes and evolution of the software product line and its core assets. The existing development process should be modified to handle the emerging considerations.

Thesis goals: The goals have been reviewed throughout the work with this thesis. The following list presents the final goals:

1. Study the state-of-the-art in component-based software engineering and relevant extensions for the organization, such as software product lines.
2. Conduct a survey at the organization to explore the motivation for reuse and component-based software engineering among the employees and how the current development process is adapted to these issues.
3. Conduct a case study at the organization to quantify the maintenance value of component-based software engineering.
4. Based on knowledge and experience of state-of-the-art and current practice at the organization, and the results from the empirical studies the thesis suggests improvements valuable to the organization.

Competence: The presented work is not an extension of the previous fall project for the student. Therefore, the study of related literature and state-of-the-art involved a significant workload. The articles in the PhD course, Dif 8901, turned out to be valuable.

Conditions: The thesis has been conducted partly at NTNU and partly at Mogul's office in Trondheim. The organization provided a working place and access to both relevant documentation and office at all times.

In retrospect: The final goals of the thesis have emerged through the increasing knowledge not only to the state-of-the-art, but more importantly to the current practice and how mature the concepts of component-based software engineering and software product lines were at the organization. All four goals have been achieved as planned.

Preface

This master thesis is written as the concluding part of the Master of Science degree in Computer Science at the Norwegian University of Science and Technology (NTNU). The industrial contribution and practical view is derived through experience, knowledge and help from the Mogul, Trondheim.

I wish to thank my head supervisor, Professor Reidar Conradi at NTNU, PhD student Li Jingyue at NTNU, Anita Efskin and Gunnar Nordseth external supervisors at Mogul. The insightful input, valuable feedback and help throughout my work were inspiring and essential. I would also like to thank the anonymous participants who helped out with answering the questionnaire for the experiment.

Trondheim, June 20th, 2003.

Øivind Wang

Table of Contents

Abstract i
Preface v
Chapter 1. Introduction 1
1.1. Motivation 1
1.2. Context 1
1.3. Structure of Thesis 3
1.4. Reading guide 4
Chapter 2. Survey of state-of-the-art 5
2.1. Component-Based Software Engineering (CBSE) 7
2.1.1. Defining a component 9
2.1.2. Why CBSE? 11
2.1.3. Criticism of CBSE 13
2.2. Software Product Lines 14
2.2.1. The Product Line Engineering Process 17
2.2.2. Variability in Software Product Lines 19
2.2.3. Change Management and Evolution of Software Product-Lines 23
Chapter 3. Mogul Context 31
3.1. History 33
3.2. Work areas 33
3.3. The Kamelon-project 34
3.4. Analysis and results from project 34
Chapter 4. Research Focus and Methods 37
4.1. Research Problem 39
4.2. Methods 40
Chapter 5. Survey of Reuse Aspects in Software Development 41
5.1. Planning and Executing the Survey 43
5.2. Results from the Survey 44
5.2.1. Comments to the Question Categories 46
5.2.2. General Questions 48
5.3. Evaluation of the Hypotheses 50
5.4. Validity discussion 53
Chapter 6. Empirical Data on Frequency of Change Requests and Fault Reports in Relation to Components 55
6.1. Results from the Case Study 57
6.2. Evaluation of Hypotheses 58
6.3. Validity discussion 60
Chapter 7. Improvement Suggestions at Mogul 61
7.1. List of Improvement Suggestions Based on Research 63
7.2. The Introduction of Software Product Lines at Mogul 64
7.3. Change Management and Evolution of the Introduced Software Product Line 66
7.3.1. Description of Current Process 67
7.3.2. Process Adoption 68
Chapter 8. Conclusions and further work 71
Chapter 9. References 73

Appendix 79
Appendix A Confidential 81
Appendix B Questionnaire of the software development process and reuse at Mogul, Trondheim 83
Appendix C Abbreviations..........95

List of Figures

Figure 1.1 - Structure of Thesis 3
Figure 2.1 - From decomposition to composition [Ommering 2002] 8
Figure 2.2 - Conventional Process and CBSE Process [Aoyama 1998] 9
Figure 2.3 - Mapping from Product Line to Product 15
Figure 2.4 - The ESAPS Product Line Process Lifecycle [Schreiber 2001] 17
Figure 2.5 - Example of Product Line Engineering Process [Schreiber 2001] 19
Figure 2.6 - Changes within the Product Line Engineering Process [Schreiber 2001] .. 24
Figure 5.1 - General Question Q1 48
Figure 5.2 - General Question Q2 49
Figure 5.3 - General Question Q3 50
Figure 7.1 - Software Product Line at Mogul 66
Figure 7.2 - Current Change Management Process 68
Figure 7.3 - Modified Change Management Process 70

List of Tables

Table 2-1 - Variability Mechanisms on Different Levels 22
Table 5-1 - Survey questions, relation to null hypotheses, and results 46
Table 6-1 - Absolute Frequency of Changes and Errors 57
Table 6-2 – Mean value of Changes and Errors 58

Chapter 1. Introduction

1.1. Motivation

This thesis is written as the concluding part of the Master of Science degree in Computer Science at the Norwegian University of Science and Technology (NTNU). The task was chosen based on interest in this particular field of study and interest from the Norwegian industry. The Trondheim department of Mogul was contacted by the head supervisor at NTNU and the collaboration was a fact.

Mogul has reengineered an old monolithic architecture to a solution based on components for a large customer in the Norwegian bank and finance sector, and was for that reason a perfect partner in the context of this work. The original goal was to perform an in-depth study of the transition to developing components. However, an interview with the technical manager of the transition project showed that most documentation was restricted by a confidentiality agreement between Mogul and the customer. Key figures on time and cost were impossible to obtain, making it hard to perform an accurate cost-benefit analysis of component-based development. The results are still included, but much reduced. The goal was transformed into how the developed component should be used in the best manner in future products for the customer. This resulted in a study of the current practice at the organization, identification of problem areas and suggestions of improvement.

1.2. Context

The following list presents the final goals:

1. Study the state-of-the-art in component-based software engineering and relevant extensions for the organization, such as software product lines.

2. Conducting a survey at the organization to explore the motivation for reuse and component-based software engineering among the employees and how the current development process is adapted to these issues.
3. Conducting a case study at the organization to quantify the maintenance value of component-based software engineering.
4. Based on knowledge and experience of state-of-the-art and current practice at the organization, and the results from the empirical studies the thesis suggests improvements valuable for the organization.

The goals themselves present to some extent the used research methods. To specify, goal one is achieved by a thorough literature study and includes several articles from the PhD course, Dif8901. The second goal is supported through an empirical survey conducted as a questionnaire at the organization. For the third goal an empirical and quantitative case study was chosen. The forth and final goal is reached through assessment of the previous three goals, and knowledge and experience of the current practice at the organization. In addition, interviews have been carried out to gather personal experiences and information that could not be acquired anywhere else. The chapters will point out where this method has been used.

1.3. Structure of Thesis

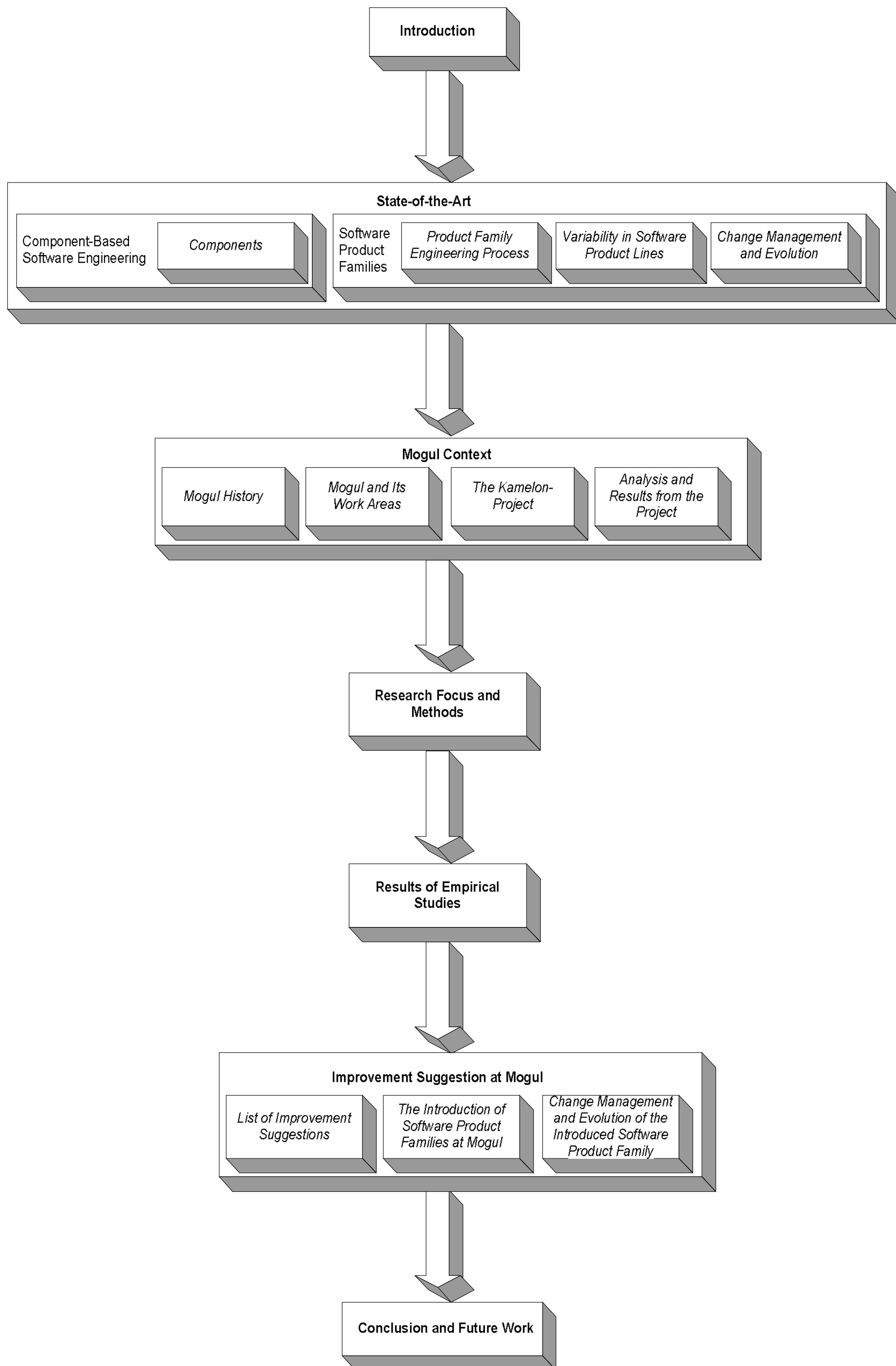


Figure 1.1 - Structure of Thesis

1.4. Reading guide

This thesis can be read in several ways.

If you are interested in the state-of-the-art of CBSE and software product lines, read chapter 2. This chapter can be skipped if you already are familiar with the concepts.

If you are interested in background information and the current situation at Mogul, read chapter 3. A more technical description of the transition to CBSE can be found in the appendix.

The rest of the thesis should not be left out as it presents the given contribution. First the research focus and methods are presented. Chapter 5 and 6 present the result from the conducted empirical studies.

If you are interested in the suggested improvements to the current practice at Mogul, read chapter 7.

The last chapter concludes the work.

Chapter 2. Survey of state-of-the-art

This chapter is designated to survey the state-of-the art for achieving goal one.

The theory discussed here can be divided into two main parts; *Component-Based Software Engineering* and *Software Product Lines.*

Subchapter 2.1 provides an introduction to component-based software engineering defines a component and discusses the pro and cons. Relevant information on components can be found in appendix B.

Subchapter 2.2 takes component-based development one step further and introduces software product lines, related process, variability, change management and evolution.

2.1. Component-Based Software Engineering (CBSE)

Ever since the term “software crisis”[1] was introduced in the late 1960’s, researchers and developers have been concerned with how to reduce development costs, maintenance costs and time-to-market, and to increase the quality of the software. Many different approaches to software system construction have been proposed to deal with the problems. One of these was the reuse of existing software through componentization [McIlroy 1968]. Until we ended up with what we today refer to as component-based software engineering, several iterations were done. During the 1970’s program modules were proposed as the unit of reuse. The 1980’s brought us object oriented programming which proposed classes as its units of reuse. [Meyer 1999] calls it absurd to present CBSE as “the next thing after objects” because object oriented-methods pursue the aim of building software from reusable components as well. However, objects are too fine-grained for assembly and reuse [Brown 2000]. CBSE aims to reuse large components, components that may make up the larger part of a system.

The following definition of CBSE is taken from an IEEE article [Capretz et al. 2001]:

> *“Component-based software development strives to achieve a set of pre-built, standardized software components available to fit a specific architectural style for some application domain; the application is then assembled using these components.”*

Many different developers claim that the established software development process needs adaptation [Ommering 2002, Sindre et al. 1995]. Traditionally, software developers start from a (single) system specification. The system is decomposed into subsystems, the subsystems into components, which again is implemented and integrated into subsystems and ultimately into the desired system. Through CBSE the components are developed to fit into some high-level architecture. These components are combined in multiple ways into subsystems, and subsystems in multiple ways into systems. In other words we want *composition* rather than *decomposition*. Figure 2.1 presents the fundamental transition from decomposition to composition.

[1] Software crisis is referring to the point in time where researchers became aware that the cost of developing software had exceeded the cost of developing hardware.

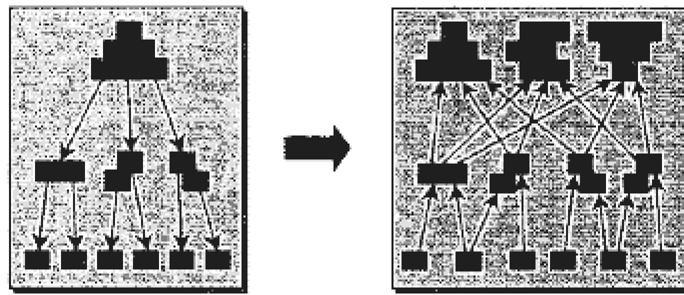


Figure 2.1 - From decomposition to composition [Ommering 2002]

One fundamental difference is that in a decomposition approach, it does not matter where the functionality is implemented as long as it is a part of the final system. In a component approach this does matter, since some components may be used in other systems [Ommering 2002]. To clarify this, figure 2.1 shows multiple subsystems composed into multiple systems. The functionality implemented in the subsystems must be of common use for all the systems using that subsystem. If some of the functionality is not interesting for one of the systems, the functionality may have to be moved down the hierarchy and implemented in one of the components.

Artifacts must be reused at all stages of the software lifecycle in a planned and systematic manner [Reifer 1997]. This includes not only reusing implemented components, but also artifacts related to analysis, design, testing and maintenance. The conventional software development process must be modified to support both the development and the integration of the artifacts [Aoyama 1998]. This is in many cases referred to as development *for reuse* and *with reuse* [Sindre et al. 1995]. These two processes can be done concurrently by two separate organizations. For that reason, third-party vendors may develop needed components for sale as COTS (commercial off the shelf) products. The decision on whether to buy-or-make depends of the core competencies of a company (i.e., a mining firm would develop its own components for mining applications if it provided its users with some advantage over COTS). Figure 2.2 illustrates the conventional waterfall process and an example CBSE process on development with reuse. A new process for component acquisition is introduced, the processes for design and implementation are modified, and the process for unit test is excluded.

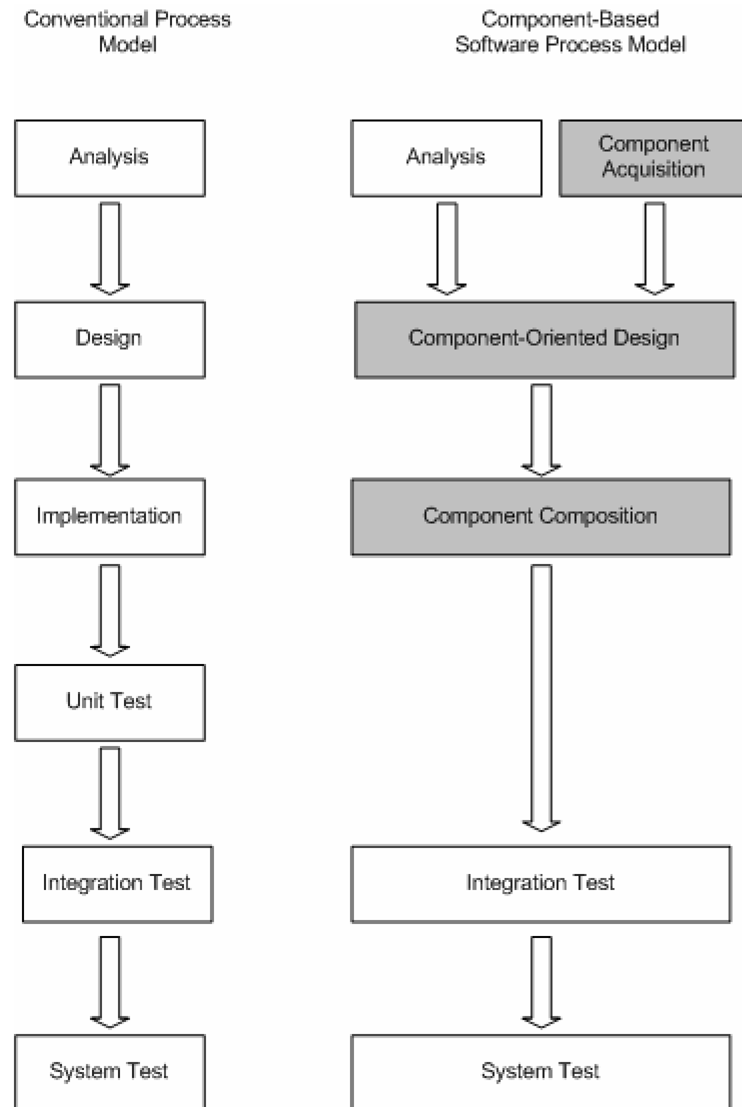


Figure 2.2 - Conventional Process and CBSE Process [Aoyama 1998]

2.1.1. Defining a component

The following elements and properties of a component is proposed by [McInnis 2000] and supported by the different definitions given in [Brown et al. 1998].

A component consists of the following three elements [McInnis 2000]:

1. **Interface:** An interface tells the developer using the component what functionality the component provides.
2. **Implementation:** An implementation is the source code that implements the functionality promised through the interface. A component may have more than

one version of the implementation. For example, a component might possess one implementation that relies on a relational database, and another that uses an object-oriented database.

3. **Deployment:** The deployment of the component is the executable file that is used to make the component run.

A component needs to have certain properties [McInnis 2000]:

1. **Encapsulation** is grouping and hiding of the implementation of the functionality. Because the developer using the component does not need to understand the implementation to use the component, access is defined through interfaces, thus providing encapsulation. This is often referred to as separating the "what" from the "how".
2. **Descriptive** means describing the component's three key elements. For a developer to be able to use the component, its interface, implementation and deployment must be described to ensure correct use.
3. **Replaceable** is related to encapsulation and description of a component. Due to these properties a component's implementation can be replaced as long as its interfaces remain unchanged or new component offers the same interfaces.
4. **Extensible** means that the provided functionality can be extended without affecting the systems using the component.

Even though there exist many different definitions of a component and the decision of which definition to choose is a very subjective one [Brereton et al. 2000], the following definition is will be used throughout in this thesis:

> *"A **component** is a language-neutral, independently implemented package of software services, delivered in an encapsulated and replaceable container, accessed via one or more published interfaces. A component is not platform constrained or application bound"* [McInnis 2000].

A component is considered to be language-neutral, which means that they are designed and deployed so that components written in different languages can work together, which effectively makes them language neutral.

A component may be implemented independently of others. This is possible due to the property of encapsulation. Each one is its own self-contained small unit of development and testing and is therefore subject to third-party composition [Brown et al. 1998].

A component is not constrained to a single platform. The deployment element makes it possible to develop different containers for a component operating on any platform.

A component is not bound to a particular system. Although many components are developed for a specific system, encapsulation and interfaces results in possible reuse in different systems. The extension property contributes to reuse with modifications as long as the interfaces remain unchanged.

2.1.2. Why CBSE?

CBSE has given a lot of promises in solving the software crisis and a large number of expected benefits have been pointed out. [ComponentGroup 2003] provides a summary of both the technical and business benefits as follows.

Technical benefits are:

1. Complexity is managed better, thereby improving the quality of the solution.
2. Components are constrained by the framework, which contains complex non-business functionality.
3. Independent design, implementing and testing allows a high degree of concurrent development.
4. Loose coupling of components stops a ripple of changes spreading throughout a system when requirements change, or enhancement is required.
5. Generic services gain leverage from fixed interface which allow rapid and powerful functionality to be added consistently in one place.

Business Benefits:

1. Higher quality product.
2. Reduced time-to-market.

3. Better utilization of human resources.
4. Ability to respond to changes.
5. Reduced costs (both development and maintenance).
6. High reuse for future projects.

The relation between the technical benefits and the business benefits is obvious. The new technical approach of developing software is introduced in order to present increased business benefits. Business factors are the driving forces in most development and research.

An analogy

The advantages of moving to a component-based model for software development can be compared to the evolution of the home stereo. Older stereos used to be built as single units. When a part required repair, the entire unit had to be taken in for servicing, and it was often difficult to locate the problem because there were no clear boundaries between its parts. It was also impossible to add components to single-unit stereos, and an entirely new system containing the desired component had to be purchased.

Newer stereos are made up of components that are connected. When a repair is required, only the faulty component must be examined. If a new component appears on the market, it can be added to the stereo by simply plugging it into the existing system. The connections are standardized in order not to exclude any actors from producing new components driving the technology forward.

In the past, software systems have been designed a lot like single-unit stereos. If there was a problem, it was hard to locate, and repairs often had an impact on other parts of the system. Adding a new function to the existing environment often meant a great deal of hard work, including recompiling entire applications to link the new software to the rest of the system. Although many software solutions are still being built this way even today, there is now an industry shift toward designing sound software architectures that are based on components. When these systems require a new component, it can be created or purchased, and plugged to the software system and the developers pray for it

to work. If one component needs to be repaired or enhanced, the other components are not affected.

2.1.3. Criticism of CBSE

If CBSE really incorporates all the benefits from subchapter 2.1.2, why are software components not utilized in more applications? In addition to the positive view presented in the previous subchapters, [Tanacea 2003] points out that some developers even claim that CBSE; saves up to half of software development costs, is easier to maintain, and incorporates a shorter development cycle.

[Tanacea 2003] claims that the answer to the slow adaptation to CBSE lies in the application development process, especially the early stages of software development - research and discovery. CBSE has been hampered due to lack of discipline and expertise, especially when applying CBSE to the application development process.

The same author identifies several driving factors:

1. Software applications need to be well defined and designed before coding begins.
   This is especially important for CBSE. Through critical tools, such as iterative prototypes, functional requirements, business process flows and transitions, and use cases, reusable components can be identified. If these tools are used correctly and rigorously, base components that are utilizable in multiple applications will emerge. Firstly, it takes knowledge and foresight to design for reuse. Secondly, there is always the urge to quickly finish designing and start coding.

2. All development benefits must be measured.
   Metrics for calculating component cost (in component development and re-use), productivity (speed of development) and quality (number of defects) must be established and tracked. Application metrics, the number of places components are used, the cost per application (total cost divided by the number of applications) and the percentage of applications using reusable components

(productivity increase) must also be established. This quantitative analysis should improve the development process.

3. Design and development are not separated.
   Different individuals must work on each phase. Both must have their own peer testing and review process.

4. Managers must ensure that the internal component assets are well organized. Developers must be able to search easily for components that meet their functional and technical requirements. Components must also be well developed and well documented.

2.2. Software Product Lines

> *"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"*
> [SEI 2003].

[Bosch 2000] points out that when combining software architecture and component-based software development, the result is software product lines. The components are developed to fit into the higher-level structure defined by a software architecture. Furthermore, the components implement the common features and are to be looked upon as the core assets of the product line.

In general, the software product line consists of the following abstract[2] elements:

- Product line requirements; the requirements, both functional and non-functional, that has to be fulfilled by all the products in the product line. This set of requirements is referred to as the commonalities.

[2] The elements are abstract because they are all specifications and not an instantiation.

- Product line architecture; the architecture is the higher-level structure incorporated by all the products in the product line.
- Component set; the set of core assets shared among the products.

For the products we can identify the same elements because the products are instantiations of the elements in the software product line:

- Product requirements; specific requirements related to the specific product. The intersection of these requirements is referred to as the commonalities and the rest are referred to as the variabilities of the product.
- Product architecture; the architecture of the product which is restricted by the product line architecture.
- Component set; for the product this will consist, not only of the core assets implementing the commonalities, but also the components implementing the variabilities.

Figure 2.3 shows the mapping between the elements in the product line to the elements for the products.

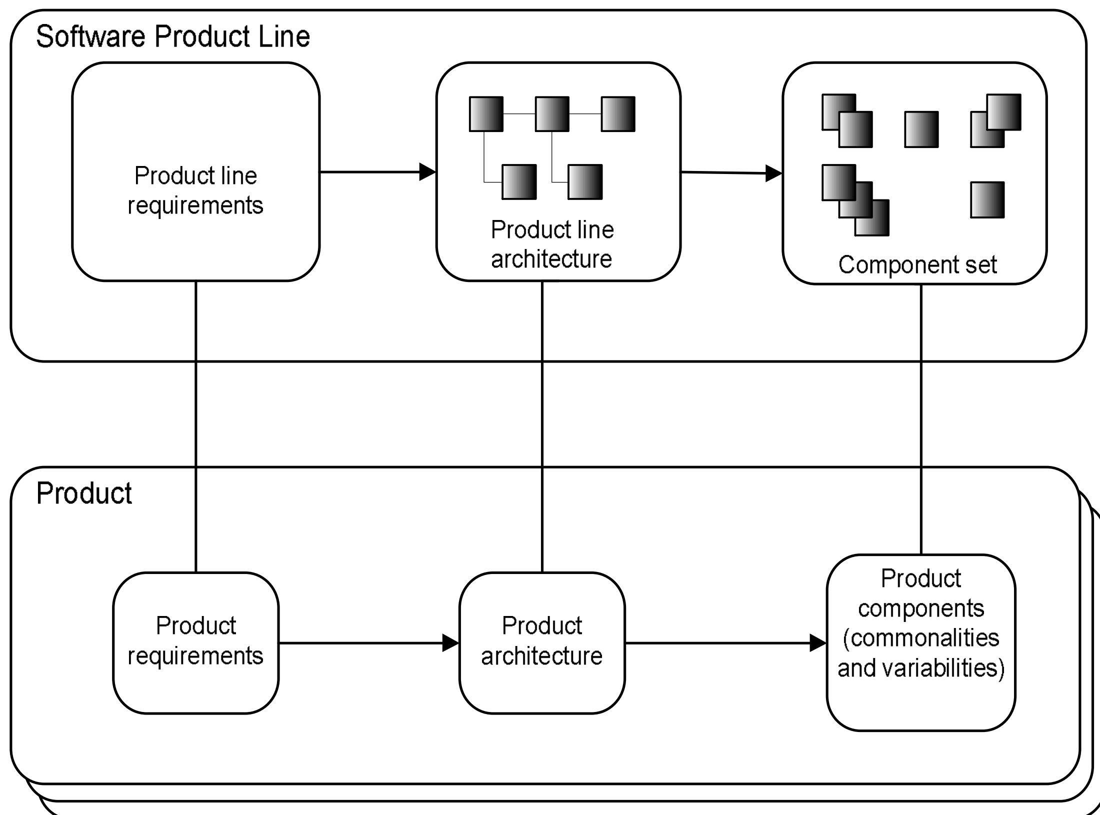


Figure 2.3 - Mapping from Product Line to Product

As building products from common assets can yield remarkable quantitative improvements in productivity, time to market, product quality, and customer satisfaction, more and more software companies are implementing their own software product line. Because every organization is different and comes to the product line approach with different goals, missions, assets, and requirements and constraints, the process of adoption and evolution of product line is different. [Schmid 2002] distinguishes the adoption strategies of product line into two categories, *big-bang* methods and *incremental* methods. [McGregor et al. 2002] denote the approaches respectively as *heavyweight* and *lightweight*. Big-bang approach is to set up a completely new product line by developing a reuse infrastructure for the whole range of products right from the start. Commonalities are identified first by domain engineering and product variations are foreseen. Because it is hard to predict future requirements, the investment analysis will be uncertain and the risks high. Usually, constraints on available resources and unpredictable future force the use of incremental approach. The first step of incremental approach is to decide a product family architecture based on the architecture of the existing products. Then components that fill the requirement for more than one product must be identified, and starting with the most important, a product's component must be generalized into core assets of product line. Each time a core asset is finished, the products that are to be a member of the product line must be adapted to use the new shared component rather that their previous individual one. By generalizing more and more components, the company gradually moves from a traditional product based approach to a software product line. In general, incremental method reduces the initial cost of implementing a software product line.

[Gjertsen et al. 2002] presents a list on how the product line-oriented development differs from classic contract-oriented development in many ways:

- Sharp distinction between family development (common development for the entire family) and product development (specific for each product).
- The reason for this distinction is a model of common and varying characteristics. The common base is developed as a part of the family development process and is a very central element.
- Other important elements are a common architecture model and common components.

- Applications are built according to the common architecture and combined of common components that are specific for each application.
- Common components can be modified to meet a certain products specific needs.

2.2.1. The Product Line Engineering Process

Figure 2.4 presents the high-level product line process lifecycle used by the ESAPS-project. The ESAPS-project is a European industrial-cooperation project on engineering software architectures, processes and platform for system families. As stated in [Linden 2002] the typical software development process involves separate development for each product, the product line engineering process focuses on a development process incorporating all of the line's products.

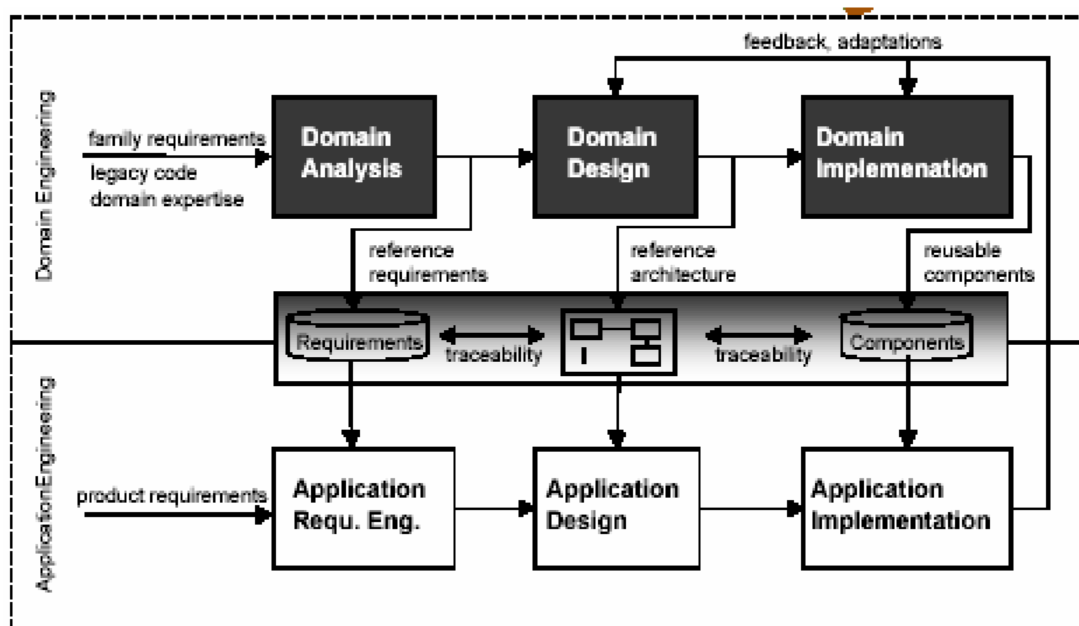


Figure 2.4 - The ESAPS Product Line Process Lifecycle [Schreiber 2001]

The process consists of two parts: *domain engineering* and *application engineering* [Schreiber 2001]. Domain engineering describes the development *for reuse* and produces core assets that the product development process will reuse. This process involves the following sub-processes:

- Domain analysis: analyzing the domain in order to define the product line. The commonality and variability analysis as a part of the domain analysis analyses

and identifies the common product line requirements, i.e. requirements that are common to all members of the product line, and the discriminating requirements, i.e. the requirements that may be different for different products.

- Domain design: development of a product line reference architecture based on the reference requirements.
- Domain implementation: building and buying components and supporting infrastructure.

Application engineering describes the development *with reuse* and produces the products as instances of the product line based on the product line platform. Sub-processes within the application engineering are:

- Application requirements engineering: determining what the product should be.
- Application design: components to make the product.
- Application implementation: combining components using the infrastructure and possibly additional product-specific code.

Figure 2.5 shows the main process steps involved in the product line engineering process and the causal dependencies of these steps, but not the temporal dependencies. The figure presents the input/output behavior, but not the time sequence of these steps. The presented process is abstract and has several different implementations depending on the organization in which it is implemented. Factors that affect the implementation are organizational structure, the type of budgeting and cost distribution, and the proprietary process of the organization. Most companies today already have an iterative process and this will of course be considered when the product line process is implemented. This will typically be reflected in the application engineering sub-processes with iterations between requirement engineering, design and implementation.

[Schreiber 2001] presents an implementation of the process that can be seen in figure 2.5.

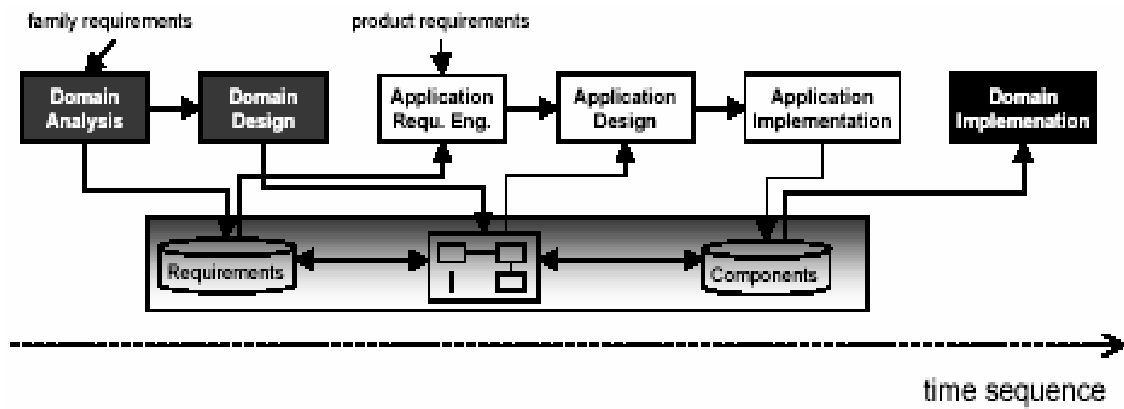


Figure 2.5 - Example of Product Line Engineering Process [Schreiber 2001]

The process starts with the family requirements as an input to the domain analysis, which results in the reference requirements for the product line. Through the domain design a reference architecture is derived. Based on the product line requirements and the product requirements the application requirement engineering begins. Subsequently, the application design is performed based on the reference architecture and the output of the previous sub-process. The application can now be implemented and the reusable components are developed and stored for further use.

2.2.2. Variability in Software Product Lines

Where no other reference is given this subchapter is based on the research done by [Svahnberg et al. 2000] and presents the different levels at which variability might occur and the available techniques to implement the variability into the product line and the components.

[Svanberg et al. 2000] defines *variability* as how the product line allows for and facilitates the difference between the products in the product line, and concludes that a cost-effective management of variability is one of the key issues for a successful product line.

[Jacobson et al. 1997] defines the *variability point* as a location in software at any level of detail at which variation can occur.

Levels of Variability

Variability occurs at different levels in the design. The following levels are presented:

- *Product Line Level* - how different products in the product line varies.
- *Product Level* - concerning the architecture and choosing the components for a particular product.
- *Component Level* – How to add new implementations of the components interfaces, and also how these evolve over time.
- *Sub-component Level* – A component consists of a set of features. In most cases these features can be grouped into a finite number of smaller feature sets often referred to as sub-components. One can therefore say that a component contains a set of these sub-components and this level is concerned with selecting the correct ones.
- *Code Level* – Where evolution and most variability between products actually takes place.

Available Techniques

To handle the variability in the different levels of a product line, [Jacobson et al. 1997] has proposed the following techniques:

- *Inheritance* is used when the variation point is a method that needs to be implemented in all products, or when some products need to extend the common functionality. The inheritance from a (possibly abstract) super class typically implements the commonalities, and the different extending sub classes implement the variabilities.
- *Extensions and extension points* is used when part of a component can be extended with additional behavior, selected from a set of variations for a particular variation point.
- *Parameterization* is used when the source code has different behavior depending on the parameters it is given. The parameters can be passed either at compile time or at run time. Two important techniques in parameterization are *templates* and *ifdefs*. Templates are a general construct used to await the decision of which types to support. Ifdefs are used at compile-time to decide to include or exclude source code in the compiled code.
- *Configuration and Module Interconnection Languages* are used to select the appropriate files and configure the connectors between the components. This is a high-level selection of components and additional techniques may be necessary.
- *Generation* of derived components is used when components are created from a task specification in a high level language.

Applicability of Techniques

Due to the fact that not all techniques are equally applicable to all levels of variability, a mapping is needed. The following table shows the relations between the techniques and the levels of variation. The results presented in the table are based on the work in [Svahnberg et al. 2000].

	Inheritance and Extensions	**Parameterization**	**Configuration**	**Generation**
Product Line Level	Significant.	Generally used as a more fine - grained configuration tool. Ifdefs used to remove unwanted product specific code (PSC).	Significant. Used to select the components and the PSC from a repository.	Not much used in industry.
Product Level	Significant.	Allows for a static way of connecting the components.	Significant. Used to connect the selected components.	Not much used in industry.
Component Level	Significant.	Parameterizations could be used to select the correct component implementation. Ifdefs could be used to remove unwanted code.	Relevant. If the components is built from several sub-components.	Not much used in industry.
Sub-component Level	Significant.	Not advised. Could be used, but increases complexity of code and configuration files.	Not relevant. Too coarse-grained.	Not much used in industry.
Code Level	Significant.	Not applicable. Increases complexity of code and configuration files.	Not relevant. Too coarse-grained.	Not much used in industry.

Table 2-1 - Variability Mechanisms on Different Levels

2.2.3. Change Management and Evolution of Software Product-Lines

A software product line is under continuous evolution and change. The main difference between evolution and change is the time range. Evolution covers a long time period, while changes may be a sudden or short time event [Schreiber 2001]. A change is just one step of evolution and may be a consequence or effect of evolution, but changes may as well cause evolution. The intention of product line engineering is to have planned evolution (evolution that follows a certain goal or strategy). A change should be a step towards the planned evolution. Through using the products, new requirements or requirement changes occur and these requests should be analyzed to see whether they are in the scope of the planned evolution [Bosch 2000].

The emerging requirements could occur in any (sub-) process. Figure 2.6 enlightens the different change requests in relation to the product line engineering process presented in subchapter 2.2.1. Four different change requests are suggested, the first two are related to the domain engineering and the last two to the application engineering:

- *Product Line Requirements Change Request:*
  Change requests of this type are related to the domain analysis. As stated in subchapter 2.2.1, the domain analysis results in the reference requirements for the product line and give input to the domain design process. Through the domain design the reference architecture could also be affected. Therefore, changes to the product line requirements could influence all the products of the product line and all stages of the process.
- *Component Improvement Change Request:*
  Change requests of this type are related to the domain implementation and will affect the components in the repository and subsequently all the products using these components through the application implementation process.
- *Product Requirements Change Request:*
  Change requests of this type are related to the application requirement engineering process and could lead to several different scenarios. First, the new requirements must be analyzed to see if the product with its changed requirements still is in the scope of the product line. The product has to be

redesign and reimplemented to incorporate the desired changes. The changes are only related to the specific product and will not affect the other products.

- *Component Improvement Change Request:*
  Change requests of this type are related to the application implementation process, which is a part of the application engineering. A decision has been made not to incorporate the changes in the components in the repository, but only in the components related to the product at hand. This will lead to having different versions of the components.

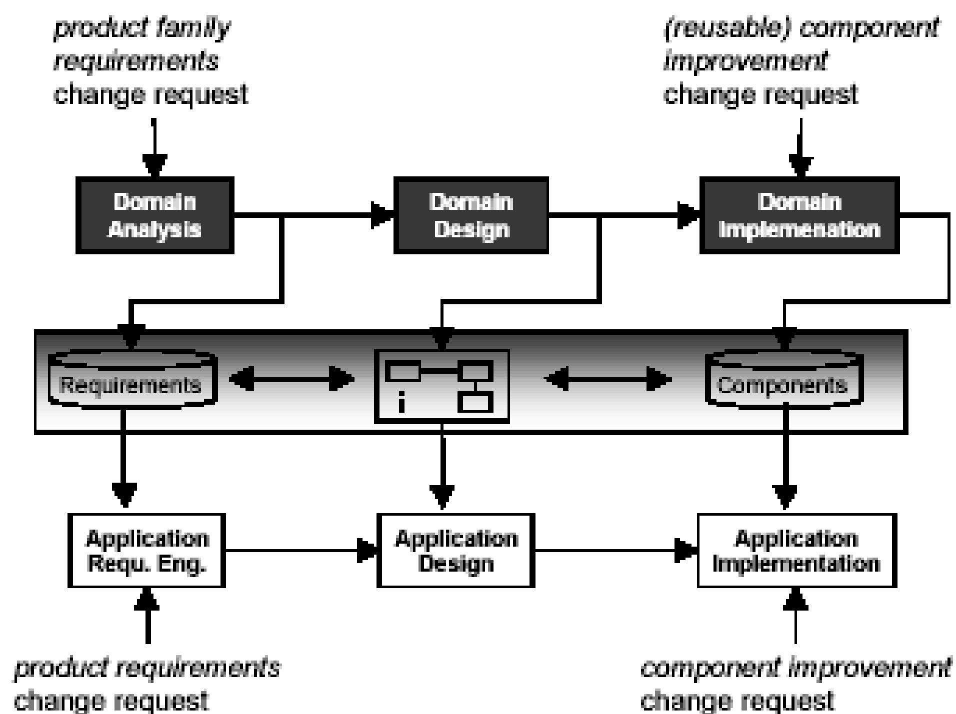


Figure 2.6 - Changes within the Product Line Engineering Process [Schreiber 2001]

The requirements and change requests affect the product line, and therefore cause evolution. In [SvBo99a] and [SvBo99b] several categories of evolution are presented:

- New product family
- Introduction of new product
- Adding new features in a product
- Extend standards support
- New version of infrastructure
- Improvement of quality attributes

All of these categories will be discussed in the following, with respect to why it occurs and the related variability techniques supporting the evolution.

New product line

The decision to introduce a new product line to satisfy the new requirements can occur due to a number of circumstances. Examples of such are; too large variability from the reference requirements in the original product line, business reasons, incorporating independent products through merging of companies, geographical distance, cultural conflicts, cloning and specialization [see Bosch 2000 for further discussion]. Typically, the change requests to the product line and the products are candidates where introducing a new product line must be considered.

A new product line be can either duplicated from an existing product line or created as a branch in a software product line. When cloning a product line it is very hard, if not impossible, to keep any functionality in common between the product families [Svahnberg 2000]. For this reason no variability techniques can be easily applied. When a branch is created all the commonalities between the two families are moved up in a product line hierarchy and new variability points are introduced to represent the differences. Inheritance and extension are applicable techniques to represent the commonalities (super classes) and variability (sub classes). In addition, configuration, parameterization and indefs can be used to select the correct classes that make up the components and finally the product.

Introduction of a new product

The introduction of a new product in a product line can originate from a number of different reasons; too large variability from the requirements in existing products, identification of market opportunity, incorporation of an independent product, and extension of product line scope.

[Bosch 2000] points out a number of steps needed to incorporate a product; identification of commonalities, matching product and product line architecture, development of variation points for product line components,

development/reengineering of product specific code, and instantiation of a product. Relating these steps to figure 2.6, the identification of commonalities is done by comparing the product requirements to the reference requirements. The product architecture is matched with the product line architecture through the application design process. And finally, in application implementation after the current components are retrieved, variation points are developed together with the product specific code and the product is instantiated.

The new product differs from the previous members of the line by providing more (or less) functionality or the same functionality in a different way (discovered in the application requirements engineering process). The changes in functionality are achieved by either adding a new component, by changing an existing component in some way, or by replacing an existing component. In addition the product specific code can be modified to incorporate the added functionality. The latter case will not affect the other product and can be implemented in a traditional manner. In the other cases important techniques to handle the variability will be inheritance and extension, parameterization and ifdefs, and configuration. When a new component has been introduced or replaced, configuration can be used to select the component to the new product. When a component has been changed inheritance and extension, or parameterization and ifdefs can be used to separate the commonalities from the variability and therefore provide different implementations of the component used by different products. A disadvantage of using parameterization, in this case, is the problem with "dead code"[3]. The parameters used when instantiating a component will not exclude any code from the compiled binary. This is however the case when using ifdefs and is therefore preferred in many situations.

Adding new features to a product

The desire to add new features to a product line emerges when the change request is related to one specific product's requirements. If this functionality is valuable to the product line in general, the feature is implemented in the core assets. Typically, the new feature is implemented in the product specific code for a single product before it is

[3] Code that never is executed.

generalized in the core assets. This can occur due to the following causes: market investigations discover new needs, new technological opportunities, competitors implement a new feature.

The techniques to incorporate the new functionality will be the same as the ones presented in the previous subchapter, but (possibly) at a smaller scale. The new functionality will affect only a subset of the components (or even sub-components) or the product specific code for the particular product.

Extending standard support

When the products in a product line initially are developed, the associated standards have to be decided. However, in most cases only the critical parts of the standards are developed in the first release and the other parts are developed in an evolutionary fashion. The most common standards are: communication protocols, components communication standards, and file systems.

Extending standard support is similar to adding new features and the techniques were discussed in the previous subchapter. One difference is that standard support in general is less product specific than adding new features and for that reason it is desirable to incorporate the changes in the product line immediately. Hence, the process of extending standard support is more related to the domain engineering process.

New version of infrastructure

The infrastructure referred to here is hardware, operating systems and third-party components. The introduction of new versions of infrastructure is concerned with incorporating infrastructure that provides functionality originally implemented by the product line. Two different scenarios arise: the new infrastructure implements all the functionality earlier provided by a component or, more complex, implements a part of the functionality of a component. In both cases, the problem is to decrease the functionality of the product line. One could decide not to include the new infrastructure and continue letting the product line assets provide the functionality. However, this is not recommended because then the maintainers of the product line assets will have the

responsibility of evolving this functionality in the future. If one decides to incorporate the infrastructure a third-party company has the responsibility of keeping the implementation complete and up to date. This is also a matter of whether the infrastructure is a part of the core competence of the company in question and the cost of integrating the new version.

If the decision has been to include the new version and the functionality of the infrastructure is equivalent to a particular component, then the redundant component can be removed. In most cases the interface of the removed component and the new infrastructure are not equal. This can be overcome by either creating a proxy component that converts the method calls on the old interface to the new interface, or calling the new interface directly which may result in changing several component implementations. A more complex situation is when the newly provided functionality is implemented as a part of a component or as several components in the current product line. New components implementing the functionality not provided by the infrastructure must be developed and redundant functionality must be removed. The effort required is often substantial when e.g. the involved components exist in different versions.

Useful techniques will be configuration both on the component and sub-component level. Functionality covered both by the infrastructure and the components could be left out when the product is compiled. On a more fine-grained level ifdefs could be used to exclude code.

Improvement of quality attributes

This type of evolution of the product line is driven by the demands on the quality attributes of the products. [Bosch 2000] points out typical examples related to the demand for improved run-time attributes, such as performance and reliability, and design-time quality attributes, such as flexibility and variability. Changes done to the product line architecture are immensely more expensive when implemented at this stage than done in the design stage. The most used and cheapest techniques to avoid architecture redesign are: caching, memory management, indirect calls and wrapping.

[Schreiber 2001] states that improvement of a quality attributes most often results in new, changed or replaced components and can be handled the same way as described when introducing a new component. Besides, quality attributes tends to be scattered throughout the software, making it difficult to identify where changes have to be made.

Chapter 3. Mogul Context

In this chapter the collaborating organization is presented. The information found here provides necessary background of the current practice and how the organization got to the point where it is today.

Subchapter 3.1 gives a brief look into the history of Mogul, how and when it came to birth. This information is based on an interview with A. Efskin and M. Collin at Mogul.

Subchapter 3.2 presents some background information on work areas.

Subchapter 3.3 provides a description of the Kamelon-project, a reengineering project to develop components. A more detailed and technical description can be found in appendix A.

Subchapter 3.4 summarizes the analysis and results from the project.

3.1. History

Numerica came to life in 1986 and was founded by 4 students from the Norwegian Technology College. Among these 4 were G. Nordseth and T. I. Byrkjeland. Numerica was a company that always strived to be ahead of technology and was market leader in areas such as Windows applications and UML modeling, and became one of few preferred partners of Rational Rose in Norway. During a project for Gjensidige NOR, a Norwegian bank, Numerica came in contact with two other strong technology companies, Internet Aksess (founded in 1995) and Taskon (1986). The project made the involved companies aware of the complementary knowledge of technology and together they formed NumericaTascon. In 2000 a Swedish company, OptoSoft, bought NumericaTascon and three other companies, WinHlp (1993), Mogul Media (1994) and Grape (1997). The result of this fusion was Mogul. As with most software companies in the last couple of years, Mogul has also gone through some tough times. In Norway the number of employees has dropped from as high as 193 in 4th Quarter of 2000 to 71 in 3rd Quarter 2002. The key figures in the annual report show a significant decrease in revenue over the same period [Mogul 2003]. Today, Mogul is divided into two divisions, Trondheim and Oslo. This master thesis has been written in relation to the Trondheim office.

3.2. Work areas

The Trondheim office of Mogul has a large customer in the Norwegian finance- and bank-sector. The main responsibility is maintenance and development of the customer's Internet bank application. The application was originally developed by another IT company called Infovision and was taken over by Mogul in 2001. The original solution was a monolithic system called NAPP (See abbreviations). After several years in production the customer itself took initiative to reengineer the old system to a new component-based solution. The project was called Kamelon.

3.3. The Kamelon-project

As mentioned above, the Kamelon-project was initiated by the customer. The IT-department of the customer wanted to develop components in order to reduce cost and time when the customer wanted to reach more clients through new channels. The resources spent in reengineering were meant to be gained in reduced effort in developing new channels. Examples of such channels are, in addition to the Internet bank, mobile bank and offices. The components should implement general business functionality and support a common architecture that could be reused in new channels. The detailed technological aspect of the project and how the migration to a new architecture was performed can be found in Appendix 1.

3.4. Analysis and results from project

This section is based on a discussion with E. Angelvik, the technical project manager, held 9th of April 2003 at Mogul's Trondheim office. The analysis presented here is partly based on [Sneed 1994] and partly based on the experiences and knowledge of Mr. E. Angelvik.

[Sneed 1994] provides a five-step reengineering planning process by which the Kamelon-project will be evaluated. The five major steps are:

1) *Project justification.*
   Justifying the reengineering project for the managers. This step requires an analysis of the existing software product, the maintenance process, and the business value of the application. The results of this analysis are used to convince the managers the benefit of software renovation. When the managers have agreed on renovating the system, an analysis of reengineering versus rebuilding from scratch has to be done. Reengineering is a compromise typically reached when new development is too expensive.
2) *Portfolio analysis* .
   Portfolio analysis plots applications, according to their need for reengineering in a 2-dimentional matrix. The applications are then evaluated in technical quality

on one axis and business value on the other. Candidates with low technical quality and high business value are primary candidates for reengineering.

3) *Cost estimation.*

   Cost estimation is somewhat more reliable in reengineering because the application already exists. It is possible to count the exact lines of code, the executable statements, and the database accesses.

4) *Cost-benefit analysis.*

   The costs estimated in the previous step must now be compared with the estimated benefits – not only the benefits of reengineering, but redevelopment and doing nothing at all as well.

5) *Contracting.*

   Outsourcing one or more parts of the reengineering to one or more companies must be considered to avoid bottlenecks.

Due to reasons of confidentiality towards Mogul's customer the process is not applicable in all steps. In these cases, the comments of the technical project manager are the only source of information.

In the following the Kamelon-project is evaluated in each of the steps presented above.

1. **Project justification.**

   Mogul played a big role in the development of the components and had the technical project manager. The customer was sitting on the moneybag and had the main project manager. As long as the customer was the initiator of the project, there was no need to convince the customer's managers into developing the components, which the Internet bank is utilizing. However, to decide whether to reengineer or rebuild from scratch, an analysis was done. The analysis was based the lines of code in the old solution and showed that reengineering was three times less expensive than development from scratch.

2. **Portfolio analysis.**

   The customer, being the initiator, chose the application to renovate. The analysis done by the IT-department of the customer on this subject is out of the scope of this thesis.

3. **Cost estimation.**

   As pointed out under project justification, the cost of engineering had already been calculated based on the lines of code in the old version. In addition, analysis on number of executable statements and database access were performed to provide a more accurate estimation.

4. **Cost-benefit analysis.**

   Cost-benefit analysis through business cases was done. The analysis showed positive net present value indicating that the project should be initialted.

Results from the project

During the project, the customer had the responsibility to measure progress and productivity. The project was meant to last one year, but ended up lasting one year and nine months with between 2 and 14 developers contributing. The cost of the project was 35% higher than the highest estimate from the analysis.

The project manager points out the following reasons for the overrun:

- A large change was taken into the project at the ending phases of the project, causing a need for an additional 6 months of work.
- The presentation layer was also ported from WebSphere servers to BEA WebLogic servers, through a project run by another company. It resulted in a lot of latent problems related to the application servers. One of the problems was how the different servers handle database-connections. The concept of connection pooling is handled differently in BEA WebLogic than WebSphere causing an increasing number of connections on a stable workload.
- The project also lacked non-functional requirements testing in the elaboration phase due to the difficulties of using a fully functional production environment. Some simulations were done, but these were of no use since the requirements are totally dependent on the backbone systems.

Chapter 4. Research Focus and Methods

This chapter presents the research focus and the methods for this thesis.

Subchapter 4.1 gives a short summary of the current situation of the organization and the related problems to how it needs to adapt to the future. The research problems are represented through four hypotheses that will be evaluated in later chapters.

Subchapter 4.2 describes the methods used to evaluate the hypotheses.

4.1. Research Problem

So far the major concern of this thesis has been to present the state-of-the-art of both component-based software engineering and software product lines. The focus has been to justify the decision to incorporate CBSE in an organization, clarify terminology and discuss the pro and cons for CBSE. In addition, software product lines have been discussed with respect to the product line engineering process, and how to manage variability, changes and evolution. A Norwegian software company was presented to see its current state in light of the state-of-the-art. The company has reengineered an old monolithic architecture to a solution based on components, and this was used as a practical example to enlighten the theoretical aspects and considerations of CBSE.

The problem which will in focus throughout the rest of this thesis is how these components should be managed, in the context of the company, to exploit the full potential of CBSE. The first consideration is to survey the motivation among the employees of reuse through CBSE. To be able to exploit the full potential of CBSE the employees must see the importance and relevance of reuse. The second consideration is to explore how the current process is viewed by the employees. The focus will be on how the process includes reuse and if any improvements could be made. The third consideration is related to maintenance costs. One of the main reasons for implementing components for the company was to reduce maintenance costs. If the new component based solution is cheaper to maintain then the overall fault and error rate in the components must be lower than in the product specific code. The final consideration is to suggest some improvements to the current practice at the company.

The following hypotheses are deduced from the considerations:

$H1_A$: Reuse in software development is advantageous.

$H2_A$: Better guidelines and workflows for development of the products are needed.

$H3_A$: Change rate in reusable components is less than in the product specific code.

$H4_A$: Fault rate in reusable components is less than in the product specific code.

4.2. Methods

In the research of finding conclusions to the hypotheses different research methods were used.

The hypotheses $H1_A$ and $H2_A$ are supported through a both qualitative and quantitative survey at the company. The reason for choosing survey as the empirical strategy was fairly straight forward. Qualitative research is concerned with studying objects in their natural setting and a survey is an investigation performed in retrospect, whereas quantitative research is mainly concerned with quantifying the relationship or to compare two or more groups [Wohlin et al. 2000]. The survey itself was performed as a questionnaire and was preferred to an interview as the latter is more time consuming and does not necessarily give more precise answers. The questionnaire provides both quantitative and qualitative answers. Analysis and interpretation is based on the work done in [Naalsund et al. 2002]. Chapter 5 treats the survey, results, evaluation of hypotheses, and validity threats.

The hypotheses $H3_A$ and $H4_A$ are supported through a quantitative case study. The data were collected from the change and fault reports at the company and traced back to the source code to establish the needed relations. Case studies are normally aimed at tracking a specific attribute [Wohlin et al. 2000]. In this case the attribute is the number of changes and faults. Analysis and interpretation is based on the chi-square test and central tendency. Chapter 6 presents a further discussion of the case study, results, evaluation of hypotheses, and validity threats.

Chapter 7 suggests improvement to the current practice at the company. The suggestions are based on the results from the two empirical investigations, familiarization with the theory and state-of-the-art of CBSE and software product families (chapter 2), and the knowledge of the company acquired through working at the company's Trondheim office and the presentation given in chapter 3.

Chapter 5. Survey of Reuse Aspects in Software Development

In this chapter the results from the conducted survey can be found.

Subchapter 5.1 gives additional information on the planning and execution of the survey.

Subchapter 5.2 presents the answers from the survey. Depending on the nature of the questions, some results are shown as bar charts and other in a table format.

Subchapter 5.3 evaluates the related hypotheses based on the answers found in subchapter 5.2.

Subchapter 5.4 discusses the validity threats to the survey and the drawn conclusions.

5.1. Planning and Executing the Survey

The participants in the survey were a small group of software developers at Mogul. Seven developers from different teams were selected by the test manager at Mogul based on their relation to the transition project and their current work tasks. Two of the subjects have worked with the development of the components and the five others are currently working on maintenance activities on the component-based system. However, one person has worked in both areas. The participants have been working in average 4 years and 9 months in the organization, ranging from 1 year and 9 months to 10 years. Their average programming experience was 15 years, ranging from 4 years to 18 years and all having at least a Bachelor's degree.

The questions were written in English and associated with the hypotheses, $H1_A$ and $H2_A$, from the previous subchapter (4.1). The first category was personal information. Even though the survey was anonymous, some information was still required to gain better understanding on what the candidates based their answers on. The next category was general reuse questions, formulated to discover how important reuse is related to different benefits, other technologies and tools, and how affected they feel by reuse in their work. The following category concerned the developed components and how these are perceived by the developers. Three questions on requirements were given to see to the opinion of the requirement renegotiation process. The next questions were associated with architecture and the knowledge of the component-based system. The final section concerned the current development process and was formulated to ascertain potential areas of improvement. The complete questionnaire from the survey can be found in Appendix B.

The execution of the survey was performed in the following manner. The final questionnaire was sent by email to the test manager at Mogul. The test manager had the responsibility of finding sufficient and relevant candidates that had spare time in their current work to participate. Originally, more than seven participants were desirable, but due to the workload and holidays this was the maximum number achievable. The questionnaire was conducted by the participants in available moments, either

electronically or on paper. The replies were gathered by the test manager, printed and delivered at Mogul's office.

5.2. Results from the Survey

In general, the participants had no trouble in understanding the questionnaire. However, in cases of confusion assumptions were taken and pointed out. These assumptions were correct and are therefore accounted for in the results. The first three general questions are shown as a bar chart in subchapter 5.2.2, whereas the rest are presented in the common table 5.1. The 'x' character shows relation between the null hypotheses and the questions. In some cases, the general answer alternatives are not applicable and are therefore added at the end of the table. Comments are given on each question category to discuss the results, and additional comments from the developers are presented.

	Null hypotheses		Answer alternatives			
Questions	**H01**	**H02**	**Yes**	**Other**	**No**	**Blank**
General on reuse						
Q1a-e: Benefits of reuse: Lower developments costs, shorter development time, higher product quality, standard architecture and lower maintenance costs.	x		See figure 5.1			
Q2a-f: Importance of approaches/activities: Reuse /CBSE, OO development, testing, inspections, formal methods and CM.	x		See figure 5.2			
Q3a-e: What is important to be reused: Requirements, use cases, design, code, test data/documentation.	x		See figure 5.3			
Q4: How would you describe the current level of reuse?	x		See below - extra table for Q4			
Q5: To what extent do you feel affected by reuse in your work?	x		See below - extra table for Q5			
Components						
Q6: The Kamelon-components are reused in some products.	x		6		1	
Q7: A reusable component is less affected by change request.	x		2	4	1	
Q8: A reusable component is less affected by errors.	x		6	1		
Q9a: The components are sufficiently documented.	x			6	1	
Q9b: If 'sometimes' or 'no': Is this a problem?	x		5		1	
Q10: Integration when reusing a component may cause some problems.	x		5			2
Q11: Is any extra effort put into testing/documenting potentially reusable components?		x	5		2	
Q12: Would the construction of a reuse repository be worthwhile?	x	x	5		2	
Q13: Errors in a reusable component are handled by a separate team.		x	1		4	2
Q14: Components are usually reused: as-is, configured, modified.			See below - extra table for Q14			
Q15: How would you decide how a component should be reused?		x	See below - extra table for Q15			
Q16: Do you feel the process of finding, assessing and reusing components is functioning?		x	4		2	1

Requirements						
Q17: Is the requirements renegotiation process working efficiently?		x	2		4	1
Q18: In a typical project, requirements are usually flexible.			4	3 - sometimes		
Q19: Are requirements of changed / renegotiated during a project?			3	4 - sometimes		
Architecture						
Q20: Do you know the product architecture well?			7			
Q21: Do you know the source code of the components well?			See below - extra table for Q21			
Q22a: Do you know the existing architecture of the components well?			6		1	
Q22b: Do you know the interfaces of the components well?			6		1	0
Q22c: Do you know the design rules of the components well?			3		4	
Q23a: Is the architecture sufficiently documented?			3		4	
Q23b: If 'no', how would you prefer the documentation?			See below - extra table for Q23b			
Q23c: If 'no', what is the problem with the documentation?			See below - extra table for Q23c			
Q24: What is your main source of guideline information during implementation?		x	See below - extra table for Q25			
Q25: Criteria for design regarding non-functional requirements are well defined?		x	1		6	
Q26: Do you test a component after modification for non-functional requirements before integration with other components?				6		1
Q27: Where you involved in the Kamelon-project?			3		4	
Development Process						
Q28: How would you describe the current development process?		x	See below - extra table for Q28			
Q29: The development process is evaluated periodically.		x	2		1	4
Q30: The processes of developing and maintaining software products are coordinated across the organization.		x	1	1	2	3
Q31: The current development process supports finding new potential reusable components/assets.		x	4		1	2
Q32: The current development process supports using existing components/assets.			6			1
Q33: The current development process supports maintenance of reusable components/assets.		x	6			1
Q34: The development process is modified after the Kamelon-project.		x	2	1	2	2
Q35: Including effects on other products will improve the development process.		x	5		2	
Q36: Introducing reuse activities in the development process will have positive effects.		x	6			1

Questions that do not fit into the common answer alternatives

	very high	high	medium	little	no
Q4: How would you describe the current level of reuse?			4	3	

	very high	high	medium	little	no
Q5: To what extent do you feel affected by reuse in your work?		1	1	5	

	as-is	configured	modified
Q14: Components are usually reused: as-is, configured, modified.	1	2	4

	by consulting experts	by following guidelines	not clearly defined
Q15: How would you decide how a component should be reused?	3	1	3

	very high	high	medium	little	no
Q21: Do you know the source code of the components well?	1	3	2	1	

	text	web pages	described through models	other
Q23b: If 'no', how would you prefer the documentation?		1	2	1- JavaDoc

	not up to date	not complete	difficult to understand	other
Q23c: If 'no', what is the problem with the documentation?		4		

	other developers	previous work	other
Q24: What is your main source of guideline information during Implementation?		6	1 - Software Architecture Docs

	formal and well documented	have a formal process, but an informal not documented is used	there exists no process	does not apply
Q27: How would you describe the current development process?	6	1		

Table 5-1 - Survey questions, relation to null hypotheses, and results

5.2.1. Comments to the Question Categories

Comments on the General Questions

All candidates feel that the current level of reuse ranges from 'medium' to 'little'. Only one developer feel highly affected by reuse and could be explained by that he is currently working on a development project. The others feel 'medium' or 'little' affected by reuse in their work.

Comments on the Component Questions

The developers are aware of the fact that the components are reused in other products. They also find the reusable components more stable and reliable (with respect to change requests and errors) than the product specific code. More effort should be put into the documentation of the components, and the combination with a reuse repository could lead to more and easier reuse, especially concerning integration of the components which most developers consider a problem today. The components are most often reused with modification, and guidelines for modification should be revised.

Comments on the Requirement Questions

The answers suggest that the requirement renegotiation process is not working sufficiently and should be revised because requirements are often / sometimes changed during a project. In addition, the requirements are considered fairly flexible and adjustable by some developers. Thus, suggesting improvement of the process.

Comments on the Architecture Questions

Even though it depends on the experience of the developers, they feel comfortable with the source code, architecture and interfaces of the components. However, the knowledge of the design rules of the components is depending on participation in the Kamelon-project. About half of the candidates state that the documentation of the architecture is incomplete and would like more information through web pages, models and javadoc. Implementation is based on their previous work, which should be made available throughout the organization in a knowledge base. Finally, the developers feel that non-functional requirements need more attention.

Comments on the Development Process Questions

The candidates perceive the current development process as formal and well documented. It also supports activities such as finding, using, and maintaining the reusable components. However, there seems to be a lack of knowledge concerning the evaluation, coordination and modification of the process among the developers. An interesting observation is the motivation for including additional reuse activities, and changes made to a component related to other products.

5.2.2. General Questions

General Question Q1 a-e:

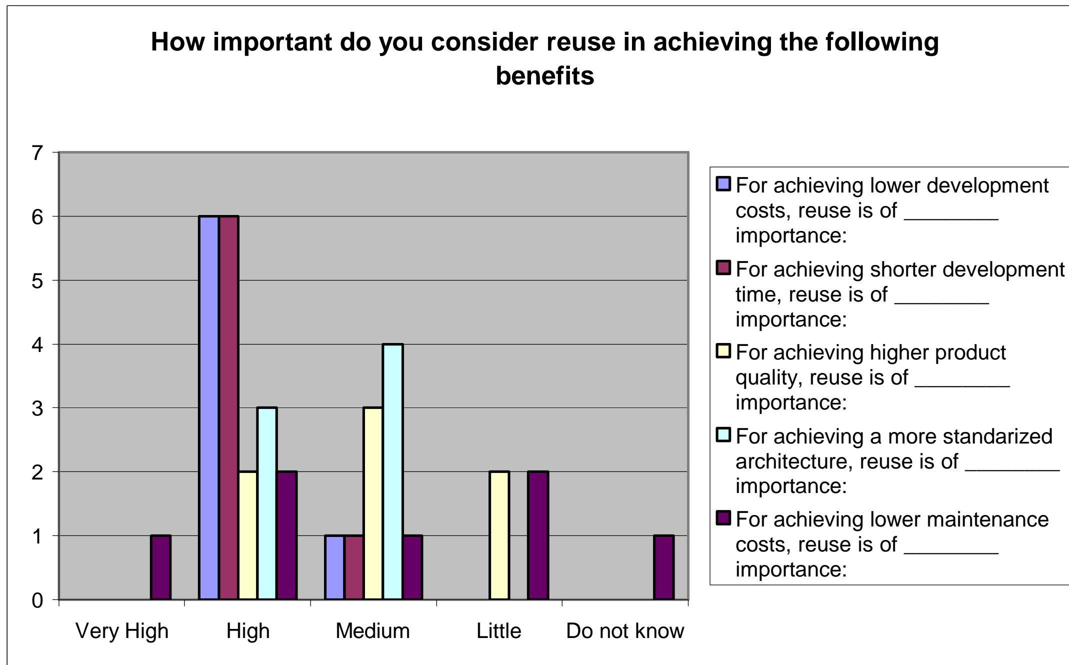


Figure 5.1 - General Question Q1

Comments: As shown in figure 5.1, the participants consider shorter development time and lower development costs as the most important advantages of reuse. An interesting aspect is the answers on lower maintenance costs, ranging from 'very high' to 'do not know'. If we take into account that most participants are working on maintenance activities one could expect a more unanimous result. Three developers comment that the benefits of reuse depends on the provided functionally and the ease of integration of the component.

General Question G2 a-f:

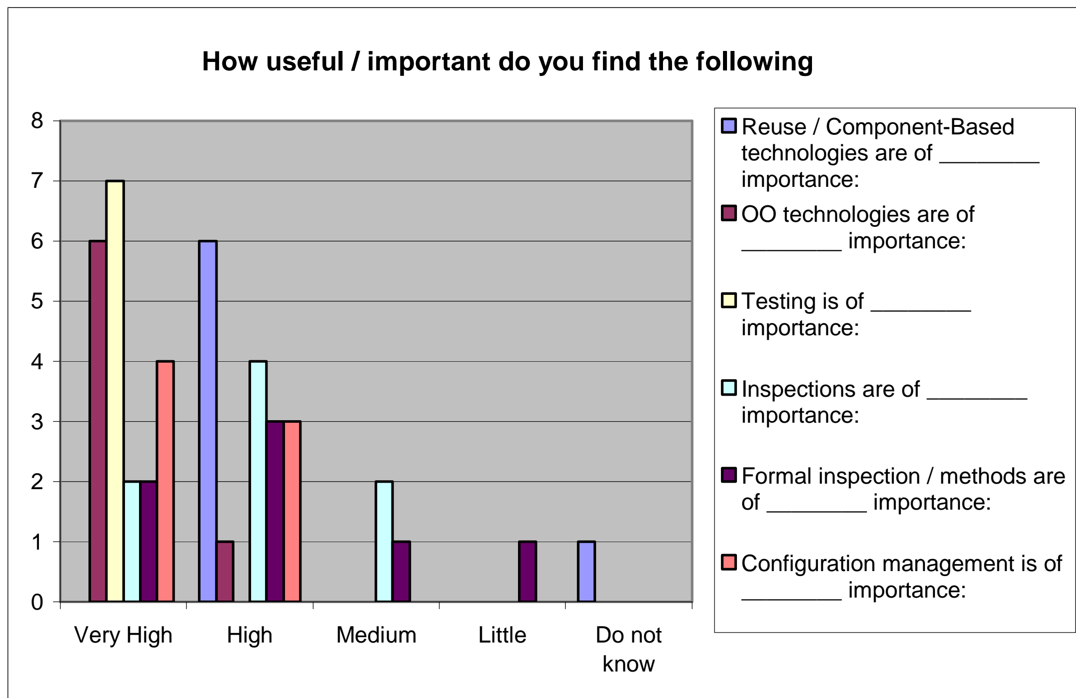


Figure 5.2 - General Question Q2

Comments: As shown in figure 5.2, reuse and component-based technologies are considered highly important by six developers. One developer does not know how important these techniques are. Testing, OO-technology and configuration management are considered to be most important, ranging from 'very high' to 'high'.

General Question G3 a-e:

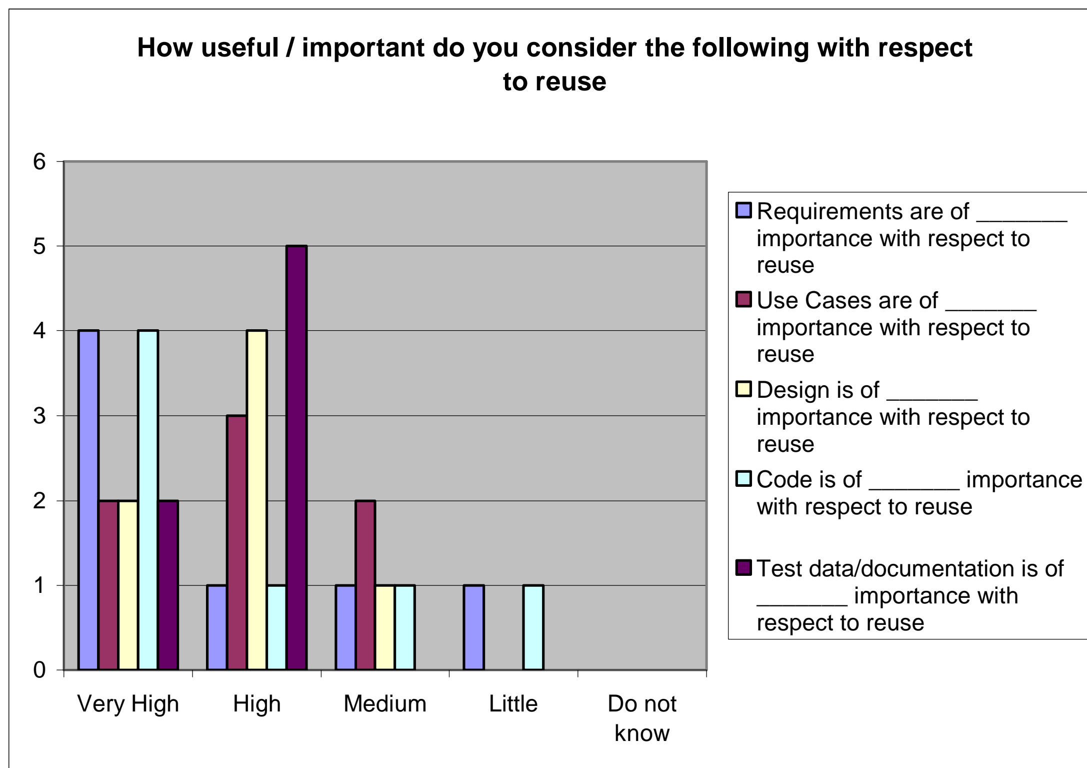


Figure 5.3 - General Question Q3

Comments: All categories are important to reuse according to the developers.

5.3. Evaluation of the Hypotheses

The null hypotheses to be evaluated are:

$H1_0$: Reuse in Software Development gives no significant advantages.

$H2_0$: The current guidelines and workflows for development of the products work well.

The alternative hypotheses are:

$H1_A$: Reuse in Software Development is advantageous.

$H2_A$: Better guidelines and workflows for development of the products are needed.

The questionnaire was formed to support/reject the hypotheses $H1_0$ and $H2_0$. Given the relatively small size of the survey, thus providing insufficient data material for

scientifically valid conclusions, only general comments will be given on the hypotheses. The origin of the hypotheses can be found in chapter 4 and table 5.1 in the previous subchapter shows the connection between the questions and the hypotheses.

Evaluation of $H1_0$:

The questions considered in the evaluation of $H1_A$ can be seen in table 5.39 and are related either to the general questions on reuse or component questions. $H1_0$ states that reuse gives no significant advantages.

In support of the alternative hypothesis $H1_A$.

The general questions Q1-Q3 reveal a positive attitude towards reuse and component-based technologies. The participants see many important advantages of reuse (Q1), and many possible artifacts to reuse (Q3). In Q2 six of seven participants answered that reuse/component-based techniques are highly important. The component questions Q6-Q8 discover that the developers are aware of the fact that the components are reused in other products, and find the reusable components more stable and reliable (with respect to change requests and errors) than the product specific code. This will motivate the developers to continue their work on reuse on component-based technologies. In addition, question Q12 presents motivation for improving the current reuse situation by introducing a reuse repository.

In support of the null hypothesis $H1_0$.

Questions Q4 and Q5 suggest that the developers do not feel too affected by reuse in their work which may result in lack of motivation. However, most of the candidates are working on maintenance activities and for that reason feel less affected by reuse. The component questions Q9 and Q10 reveal that most candidates agree that the documentation of the components is insufficient and integration of components may cause problems.

Conclusion H_A / $H1_0$.

It is little doubt that the candidates see the potential benefits of reuse and component-based development. Even though they do not feel too affected by reuse in their current

work, they are motivated to improve and optimize the level of reuse. The answers in the questionnaire point in the direction of $H1_A$. The null hypotheses may be discarded.

Evaluation of $H2_0$:

The questions supporting the evaluation of $H2_A$ can be seen in table 5.39 and are related to the component, requirements, architecture, or development process questions. $H2_0$ states that the current guidelines and workflows for development of the products work well.

In support of the new hypothesis $H2_A$.

Questions on the development process Q29, Q30 and Q34, revealed a lack of knowledge to the evaluation, coordination and modification of the process among the developers. This could be a sign of not having sufficient knowledge to the current process. Questions Q35 and Q36 show a positive attitude towards introducing reuse activities, and how changes made to one component affect all the products using the component. Component question Q13 reveals that more guidelines on how to reuse a component is needed. Q16 gives contradictory answers to Q28, and Q31-Q33. Requirement question Q17 shows the requirement renegotiation process needs to be reviewed as four developers feel that it is not working well enough. The architecture question Q25 discovers that more effort is needed when considering non-functional requirements.

In support of the null hypothesis $H2_0$.

Through questions Q28 and Q31-Q33, the candidates seem to perceive the current development process as formal and well documented. It also supports activities such as finding, using, and maintaining the reusable components, suggesting that the current process is working sufficiently. Component question Q11 suggests that the current process is already including activities to test the reusable components more rigorously than the product specific code.

Conclusion $H2_A$ / $H2_0$.

Even though some of the answers are in the favor of the null hypothesis, several aspects must be improved. There is some divergence in the given answers on how well the development process works. For that reason the null hypothesis is rejected.

5.4. Validity discussion

Threats to experimental validity are classified and elaborated in [Wohlin et al. 2000]. Threats to the validity of this survey are:

Conclusion validity: Due to the small number of participants in the survey the results cannot be proven to be normal distributed. For that reason no statistical analysis was performed. This could affect the drawn conclusions.

Construct validity: A relevant threat to the construct is that the subjects have different perception of the scale used in the answer alternatives. A person answering 'very high' may be equal to another person's perception of 'high'.

Internal validity: Most participants (about 70%) are from the maintenance unit of the organization which could lead to single group threats. A maintenance unit is generally less affected by reuse than a development unit, which can be reflected in lower motivation for reuse. Hence, influence questions Q1-Q5, Q12, Q14, Q15, Q16, Q31 and Q32. In addition, previous knowledge and experience on some approaches of software development may have impact on questions Q1-Q3.

External validity: It is difficult to generalize the results to other organizations as all the participants of the experiment were from the same organization. However, a similar survey has been conducted at Ericsson and will be performed at EDB Business Consulting [Naalsund et al. 2002]. These surveys could together provide sufficient data to generalize the results to the Norwegian industrial landscape.

Chapter 6. Empirical Data on Frequency of Change Requests and Fault Reports in Relation to Components

In this chapter the results from the conducted case study can be found.

Subchapter 6.1 presents the data from the case study and is shown in a table format.

Subchapter 6.2 evaluates the related hypotheses based on the data found in subchapter 6.1.

Subchapter 6.3 discusses the validity threats to the case study and the drawn conclusions.

6.1. Results from the Case Study

All the change requests and fault reports that emerge from the different actors, users or developers, are handled through Rational ClearQuest [Rational 2003]. From ClearQuest, which is based on a SQL database at Mogul, it is possible to retrieve the desired data through simple queries. In the context of this thesis we were interested in tracing the change requests and fault reports to see how these related to the Kamelon-components and the application specific code. The desired result would be that the components were less exposed to requests.

In this case one could not make direct queries to the database to sort the result set to components and application code. This had to be done manually and with the help of a developer with very good knowledge, not only to the data in ClearQuest, but also to the architecture of the system. Another problem was that not all the data in ClearQuest was suitable for this in depth tracing. The change requests and fault reports go through a lifecycle from their birth to their solution. Only the data that have been analyzed in detail are candidates for further study. The reason is obvious, if the data are not analyzed to some extent it is impossible to trace it to some part of the system. The extra work effort to do this is too large to fit into the scope of this thesis. However, there are sufficient data to draw conclusions only considering the data that fulfill this requirement.

The data we started with was 55 change requests and 78 fault reports. Of these data only 16 change requests and 34 fault reports were analyzed to a level were they could be traced back to the source code. The following table presents the data based on the retrieved data, both traceable and no-traceable, in its absolute frequency.

	Components	Product Specific Code	Total
Total Faults	-	-	78
Traceable Faults	14	20	34
Total Changes	-	-	55
Traceable Changes	2	14	16

Table 6-1 - Absolute Frequency of Changes and Errors

In order to draw any significant conclusions from the data material, the frequency of the data has to be relative to the code size of the components and the product specific code. Taken into account that the size of the components in number of lines is 135621 and the size of the product specific code is 168308, we get the following table of mean values per kilo lines of code (kloc).

	Components	**Product Specific Code**
Traceable Faults	0,103	0,119
Traceable Changes	0,015	0,083

Table 6-2 – Mean value of Changes and Errors

6.2. Evaluation of Hypotheses

The null hypotheses to be evaluated are:

$H3_0$: Change rate in reusable components is equal than in the product specific code.

$H4_0$: Fault rate in reusable components is equal than in the product specific code.

The alternative hypotheses are:

$H3_A$: Change rate in reusable components is less than in the product specific code.

$H4_A$: Fault rate in reusable components is less than in the product specific code.

Given the nature of the data material and the complexity of decomposing the errors and changes to a specific component, hypothesis tests such as t-test and ANOVA are not applicable. Since the errors and faults are not traced back to a specific component, neither the data on the components nor the product specific code has any variance. However, the chi-square test is applicable as a nonparametric test of significance. In addition, the central tendency will be used to evaluate the hypotheses.

The methodology for calculating the chi-square can be found in [Cooper et al. 2003] and follows the formula:

$$?2 = ?_{i=0}^{k} (O_i - E_i)^2 / E_i$$

In which O_i is the observed number of cases categorized in the *i*th category, E_i is the expected number of cases in the category under H_0, and k is the number of categories. Furthermore, the number of degrees of freedom (d.f.), is defined as (k-1).

Chi-square test of $H3_0$:

The null hypothesis is defined as $H3_0 : O_i = E_i$. In other words that the change rate of the components and the product specific code is equal. To evaluate the hypothesis, the degrees of freedom and the number of categories must be decided. This case study has two categories, components and product specific code. Thus, leading to

$$d.f. = 2 - 1 = 1 \text{ and } k = 2$$

The E_i can be calculated from table 6.1 as

$$E_1 = (2 + 14) / 2 = 8$$

Choosing a significance level of

$$a = 0,05.$$

Calculating the Chi-square

$$?2 = (2 - 8)^2 / 8 + (14 - 8)^2 / 8$$

$$= 9$$

The critical test value found in a chi-square test table is

$$3,84$$

Since 3,84<9 it is possible to *reject $H3_0$*. It should be pointed out that most change requests were related to cosmetic changes in the graphical user interface.

Chi-square test of $H4_0$:

The null hypothesis is defined as $H4_0 : O_i = E_i$. In other words that the fault rate of the components and the product specific code is equal. The number of degrees of freedom and categories are the same as the in the evaluation of $H3_0$.

The E_i can be calculated from table 6.2 as

$$E_2 = (14 + 20) / 2 = 17$$

Choosing a significance level of

$$a = 0,05.$$

Calculating the Chi-square

$$?2 = (14 - 17)^2 / 17 + (20 - 17)^2 / 17$$

$$= 1,06$$

The critical test value found in a chi-square test table is

3,84

Since 3,84>1,06 it is not possible to *reject* $H4_0$. However, by evaluating the null hypothesis by the central tendency it is possible to reject it.

6.3. Validity discussion

Threats to experimental validity are classified and elaborated in [Wohlin et al. 2000]. Threats to the validity of this case study are:

Conclusion validity: Since the expected frequencies in the case study are above 5 for all values the test fulfills the required assumptions. “Fishing” is not applicable either since the developer responsible for categorizing the request had not been informed about the involved hypotheses.

Construct validity: No threats are identified.

Internal validity: The developer’s previous knowledge and experience could be a threat to the categorizing. Not only with respect to his experience, but also because he could have been responsible for correcting some of the reports.

External validity: It is difficult to generalize the results from this experience to other companies since the data material is gathered from one particular company. Similar experiences could be performed in other companies and compared.

Chapter 7. Improvement Suggestions at Mogul

This chapter is where the thesis proposes some suggestions of improvement to the current practice at Mogul.

Subchapter 7.1 provides a list of suggested improvements based on the conducted research.

Subchapter 7.2 gives a current state on software product lines at Mogul and proposes a possible introduction and organization of this in their own company.

Subchapter 7.3 is about how the evolution and variability management can be done in the introduced software product line. The subchapter proposes a process supporting this.

7.1. List of Improvement Suggestions Based on Research

Through the research conducted in chapter 5 and 6, several aspects that need improvement are identified. Most of these aspects are deduced from the conducted questionnaire and are based on the participants' perception of the current practice at the organization. The suggestions are based on state-of-the-art, knowledge of the organization and comments from the developers, and includes a wide range of improvements, some are fairly trivial and others more complex. Only some of the improvements will be discussed in detail in this thesis.

1. *Introduce software product line* to optimize the level of reuse over more products. A description of the adaptation can be found in subchapter 7.2.
2. *Improvement of the development process.* Several answers from the subject in the survey suggest possible areas of improvement. The process needs to account for reuse across several products when software product lines are introduced. Furthermore, analysis and testing of non-functional requirements must be introduced as most developers feel this is not covered by the current practice. The process should also be coordinated throughout the organization and made available for all employees to stay up to date with changes and evolution of the process. Finally, guidelines for reusing the components must be modified. Additional activities regarding reuse over several products and effects of changes for more products must be introduced. A detailed discussion of this subject can be found in subchapter 7.3.
3. Put more effort into the *documentation of the components*. The developers feel that the components have incomplete documentation. Thus, making it more difficult to retrieve and use the components in other products.
4. Put more effort into the *documentation of the architecture*. Several developers would prefer more documentation of the architecture in the form of web pages, models and javadoc.
5. *Consider implementing a simple reuse repository.* The developers find the effort of introducing a reuse repository worthwhile, making the process of finding and using the components easier. However, some research contradicts this suggestion [Morisio et al. 2002] and needs more evaluation.

6. *Review the requirements renegotiation process.* The combination of having several developers feeling that the current process is not working, and that requirements change often or sometimes during a project, is a frustrating situation. This must be addressed, as neither the customer, nor the organization is profited with this.
7. *Introduce a knowledge base* to gather the knowledge and experience of previous work. The developers state that the main source of information during implementation is previous work. In order to access not only own work, but also the work of others, the work should be made available to all employees in the organization in a systematic manner.

7.2. The Introduction of Software Product Lines at Mogul

Several Norwegian small and medium sized software companies are starting to implement their own software product lines. This chapter introduces the term software product line at Mogul and proposes an adaptation based on the knowledge of their organization. To do this we need to analyze the current situation and see where improvement has to be made and what has to be done.

Looking closer at the definition from section 2.5 one can see that a software product line requires two necessities;

- That the products in a product line share a common, managed set of features.
- That the products are developed from the common set of core assets in a prescribed way.

If we look at this in context of the current situation at Mogul, we discover two issues. First, that the domain, in which Mogul and its customer work, is related to the software intense systems needed in the bank- and finance-sector. For Mogul the product line will consist of the necessary software systems for the customer. Currently, Mogul and its employees talk about different *"channels"*, and through these channels the clients of the customer can be reached. Example of a channel is the current Internet banking system. In addition, one can imagine possible future channels, for example a mobile bank system and the different software systems needed at the customer's offices. It is obvious that all of these systems are related to the bank- and finance-sector, and could possibly

share a set of features. A trivial example of this is when somebody is interested in accessing her or his account information. Hence, the first statement of the definition is fulfilled. Secondly, the common features have to be implemented as core assets from which the channels will be developed. In the previous example, a core asset could implement the feature of retrieving the information from the underlying backbone systems. On top of this different presentation layers could be used to realize the channels. The work done through the Kamelon-project resulted in components that encapsulate business functionality for potential reuse in different channels. To summarize the current state, there exist some components that implement business functionality and we can see the possibility for future commonality through different channels.

If we now substitute the term channels with products in the software product line representing the bank- and finance-domain, and the term components with core assets shared among the products, then we discover a single software product line with only one product. In light of the two different strategies presented in section 2.5, the *big-bang* and the *incremental* approach, it is possible to characterize the evolution at Mogul. As long as this process has happened over a certain period of time, the approach has been an incremental one and the first increment included the development of core assets, implementation and description of the architecture, and creation of requirements to the product line.

The following figure proposes a way to organize the software product line at Mogul.

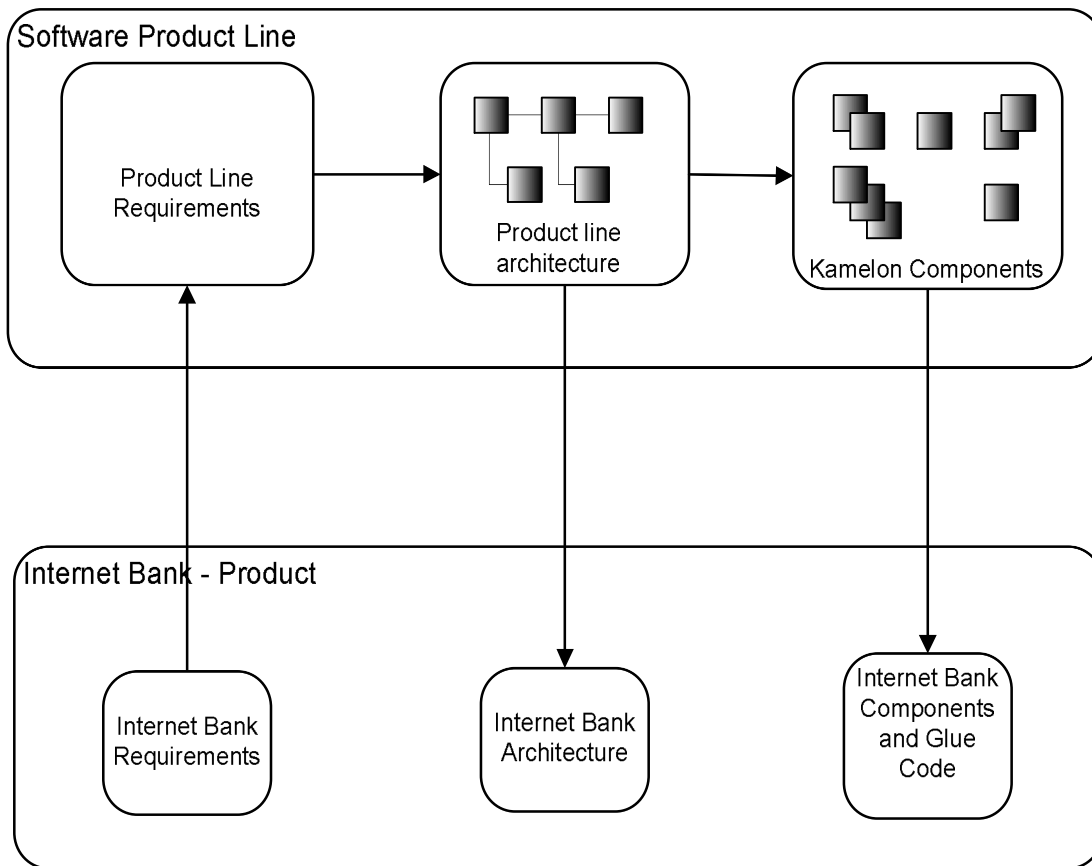


Figure 7.1 - Software Product Line at Mogul

Comparing figure 2.3 and figure 7.1 reveals that all factors in the product line are present. However, the definition reveals that the only part missing at Mogul is the process of developing products from the core assets. This will be the subject of discussion in section 7.3.

7.3. Change Management and Evolution of the Introduced Software Product Line

In order to meet the new demands of the organization emerging from the introduction of the concept of software product lines a process to manage the evolution and changes made to the product line is needed. Section 7.3.1 presents the current process for change management. Section 7.3.2 presents a modified process supporting the new demands.

7.3.1. Description of Current Process

Figure 7.2 shows the current change management process at Mogul. This process is related to the old monolithic system and shows how to handle change requests to this system.

From figure 7.2 we can see that the new requirements of changes lead to a business evaluation in order to find out whether or not to implement the requested changed. From the business evaluation there are four possibilities;

- Start implementation immediately – when business evaluation show for example a very positive cost-benefit value or when change is critical.
- Start technical analysis – for example when the change is complex or more evaluation is needed to establish reliable economical figures.
- Close – when the change is not economical profitable, or the change is insignificant.
- Postpone – when the change is neither critical nor prioritized.

When the changes have been accepted for technical analysis, the next step is to decide how the changes should be implemented in the system at hand. In the old process this is quite straight forward as long as only one system exists. The only considerations were where to implement the changes and how to do it. When the technical analysis is finished the analysis has to be evaluated. From this stage we see that there are three possible routes;

- Reject analysis – if the analysis and solution is not satisfactory from the system owner's point of view, resulting in further analysis.
- Close – the changes could be ignored for example if they are technologically unachievable.
- Postpone – if the changes are put on hold.
- Implement – technical analysis proposed a satisfactory way of implementing the changes.

Some of the changes are implemented according to the guidelines from the technical analysis. The changed system is tested and if it passes it will be put into production. If the test fails modifications has to be done until the test passes.

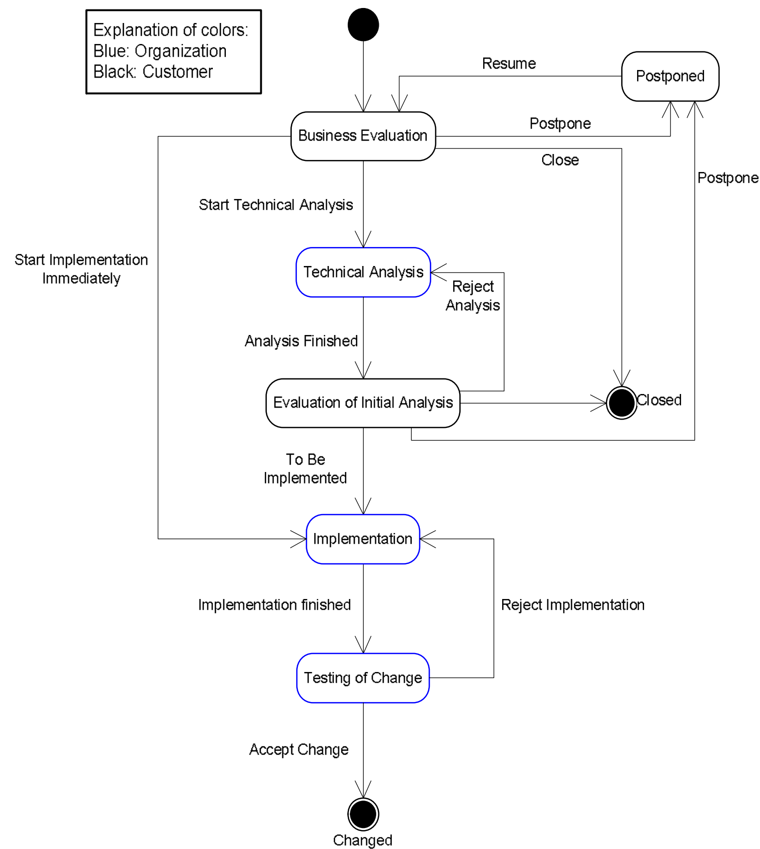


Figure 7.2 - Current Change Management Process

7.3.2. Process Adoption

This section will modify the process presented in the previous section to meet the new demands resulting from the introduction of a software product line. The process presented in figure 7.2 is inadequate because it does not cover the relations between different products. After introducing a software product line it is no longer sufficient to look at one product as an isolated unit. The process needs to be generalized to cover the evolution of the product line as a whole and the additional change management mechanisms involved when more products share core assets.

The main modifications required are related to the allowance of different change requests, additional business evaluation, and additional technical analysis. The modified process is presented in figure 7.3.

The first modification is associated to the issues discussed in section 2.2.3. The change requests and new requirements do not necessary lead to changes made to one product. The first step when changes arrive is to analyze how it will affect the product line. Several options are presented, but for the company in question most of the requirements are either related to some desired changes to one particular product, or to introduce a new product. Hence, the most typical evolution categories will be introduction of a new product and adding new features to an existing product. Of course, the requirements could lead to a new product line or improvement of quality attributes. The most typical cases would be if the new requirements differ too much from the current product line or the performance is not sufficient. In figure 7.3 all six possibilities discussed in section 2.6 are included.

The second modification is to the *business evaluation* sub-process. It is no longer sufficient to evaluate business aspects only in the context of one specific product. Changes made to core assets may not be cost-effective for the product receiving the request. However, the implemented changes could have positive effects for all products using the changed component, and also lead to increased reusability of that specific asset and therefore be economically profitable for the product line as a whole. Additional mechanisms to calculate the cost-benefit of changing core assets are needed. See [Barnes et al. 1991] for more information on reuse and cost-effectiveness.

The third modification is to the *technical analysis* sub-process. Additional analysis is needed to handle the requests that change the core assets. When some products have commonalities, the changes done to this intersection will affect all the products sharing the involved core assets. The variability mechanisms presented in subchapter 2.2.2 must be considered in this activity. The artifact from this process should be a extensive description of not only where changes have to be made and how to do it, but also the products that are affected by the changes. As a result all related products could be tested before the implementation has been accepted.

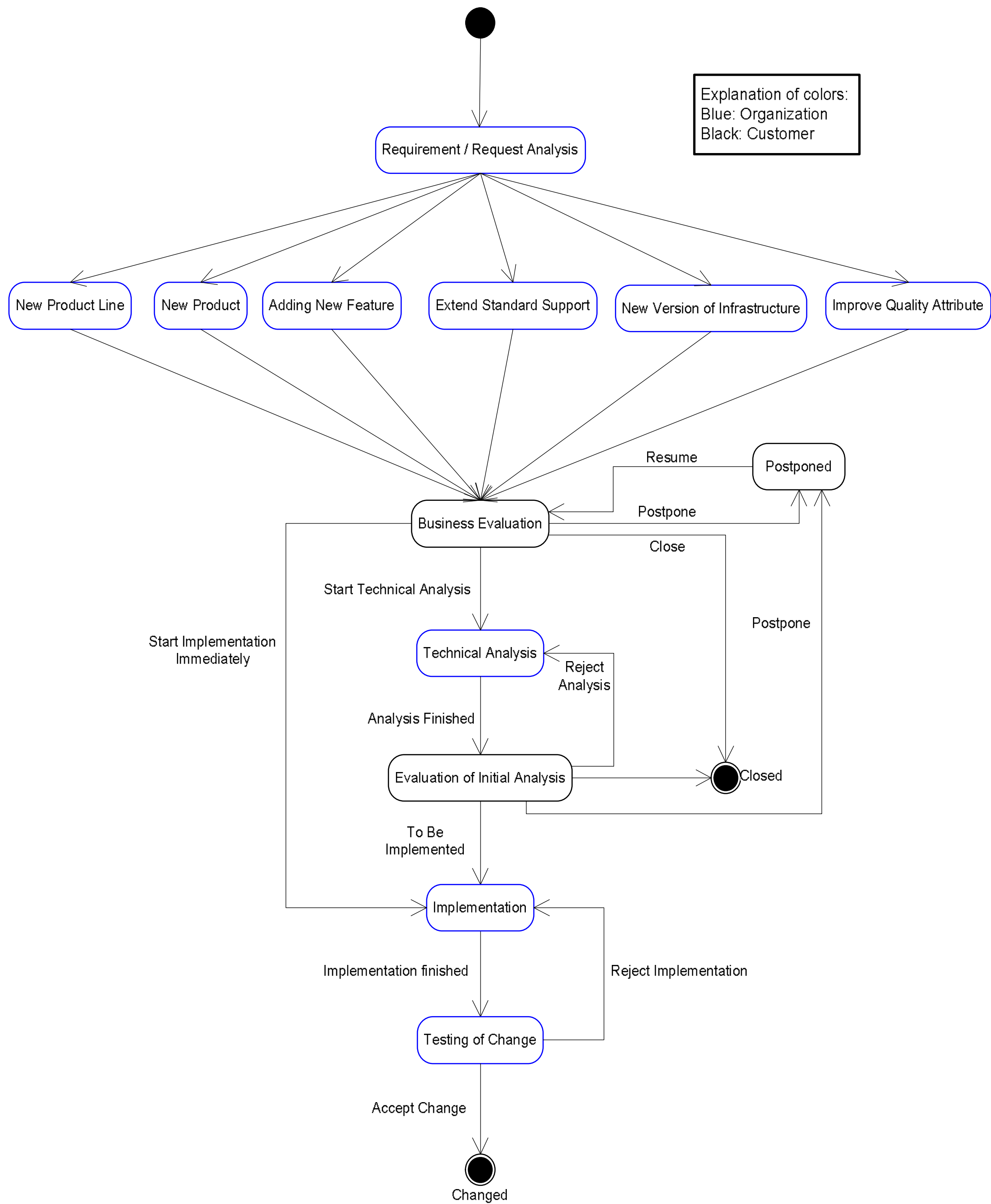


Figure 7.3 - Modified Change Management Process

Chapter 8. Conclusions and further work

This thesis studied industrial, component-based software engineering. Mogul, a software development company in Trondheim, was contacted by the head supervisor, and provides the industrial fundament of this thesis.

Mogul has performed a reengineering project to develop a component-based system from an old monolithic architecture. The developed components are reused in some products for the same customer in the interest of reducing development time and costs, increase the quality, and the other promised benefits of incorporating component-based software engineering in an organization.

In context of the current practice at the organization, state-of-the-art in both component-based software engineering and software product lines was presented to enlighten the possible development and adaptation at Mogul in the future.

Empirical work was conducted to discover areas of improvement to the current practice. A survey of the reuse benefits in software development and evaluation of the current development process was performed through a questionnaire handed out to seven developers at Mogul. In addition, a case study of the current value of developing components with respect to change and error rate was conducted. Together with the increasing knowledge of both state-of-the-art and the company, this provided a fundament from which improvement suggestion was deduced.

Based on the goals of this thesis presented in the introduction (subchapter 1.2) we can claim that this thesis has achieved all four goals. The thesis presents the relevant state-of-the-art for the company, conducted both a survey and a case study, and suggests areas that need to be improved. However, none of the suggestions were implemented and tested at the organization and it is therefore difficult to determine the value of these until a pilot project has been performed.

As for the hypotheses presented in chapter 4 – Research Focus and Methods, all hypotheses were evaluated through the conducted empirical work. The first three null hypotheses were rejected, but the forth was kept. The conclusion is that the employees at Mogul find reuse advantageous in software development and are motivated to continue working in a component-based software engineering manner. However, the current development process needs to focus more on reuse activities and incorporate the extensions needed when introducing a software product line. A suggested modification of the process was presented in subchapter 7.3.2. It was also verified that the developed components are less affected change requests than the product specific code. However, it could not be rejected that the developed components were equally affected by faults.

We believe that the suggested changes will improve the current practice and therefore prove valuable for the organization. However, the suggestions are to some extent based on the results from survey, which includes only seven test persons. The organization should perform a more extensive survey to test the validity of the obtained results.

Future work for the organization will include evaluating the suggestions and implement the valuable changes. Only two of the suggestion areas of improvement have detailed guidelines on how to do the implementation. The technical accomplishment is also left as future work for the organization. Future work for academia is related to conducting more surveys to form a broader data material of the industrial landscape in Norway. Thus, being able to apply more scientific conclusions to the aspects considered in this thesis. The case study should also be extended to trace the change requests and errors to a specific component. The amount of modified, removed and added code between different versions could be included to

Chapter 9. References

[Aoyama 1998] Aoyama M.,
New Age of Software Development: How Component-Based Software Engineering Changes the Way of Software Development,
ICSE'98: International Conference on Component-Based Software Development, http://www.sei.cmu.edu/cbs/icse98/papers/pdf.files/p14.pdf,
Last Accessed 12th of June 2003.

[Barnes et al. 1991] Barnes B. H., Bollinger T. B.,
Making Reuse Cost-Effective,
IEEE Software, Technical Article 0740-7459-91 , January1991.

[Bergey et al. 2000] Bergey J., Fischer M., Gallagher B., Jones L., Northrop L.,
Basic Concepts of Product Line Practice for the Do,
Software Engineering Institute, Carnegie-Mellon University,
Technical Note CMU/SEI-2000-TN-001, 2000.

[Bosch 2000] Bosch J.,
Software Architecture and Reuse,
Addison Wesley, ISBN: 0-201-67494-7, 2000.

[Brereton et al. 2000] Brereton P., Budgen D.,
Component-Based Systems: A Classification of Issues,
IEEE Software, 0018-9162/00 ,p.54-62, November 2000.

[Brown et al. 1998] Brown W.A., Wallnau K.C.,
The Current State of CBSE,
IEEE Software, p.37-46, September/October 1998.

[Brown 2000] Brown W.A.,

Large-scale Component-Based Development,

Prentice Hall, ISBN 0-13-088720-X, 2000.

[Capretz et al. 2001] Capretz L.F., Capretz M.A.M., Li D.,

Component-Based Software Development,

IECON'01: The 27th Annual Conference of the IEEE Industrial Electronics Society, 0-7803-7108-9/01, 2001.

[ComponentGroup 2003] ComponentGroup,

Component-Based Development - an Overview,

http://www.componentgroup.com/whitepapers/overview.html,

Last Accessed 11th of June 2003.

[Cooper et al. 2003] Cooper D.R., Schindler P.S.,

Business Research Methods,

McGraw-Hill Irwin, ISBN 0-007-24-9870-6, 2003.

[Gjertsen et al. 2002] Gjertsen S., Dahle H.P., Pettersen R., Hallstein S.,

Familier Prosjektbeskrivelse,

Bransjeprosjekt fra IKT-Norge, October 2002.

[INCO 2002] *INCO – Incremental and Component-Based Development*,

http://www.idi.ntnu.no/grupper/su/inco.html,

Last Accessed 11th of June 2003.

[Jacobson et al. 1997] Jacobson I., Griss M., Jonnson P.,

Software Reuse: Architecture, Process and Organization for Business Success,

ACM Press, 1997.

[Lim 1994] Lim W. C.,

Effects of Reuse on Quality, Productivity, and Economics,

IEEE Software, 0740-7459/94, September 1994.

[Linden 2002] Linden F.v.d.,
Software Product Families in Europe: The Esaps and Café Projects,
IEEE Software, 0740-7459/02, 2002.

[McGregor et al. 2002] McGregor J.D., Northrop L.M., Jarred S., Pohl K.,
Initiating Software Product Lines,
IEEE Software, 19(4):24-27, July-August 2002.

[McIlroy 1968] McIlroy M. D.,
Mass produced software components,
Proceedings of the NATO Conference on Software Engineering, 1968.

[McInnis 1998] McInnis K.,
Component-Based Development: The concepts, Technology and Methodology,
http://www.cbd-hq.com/PDFs/cbdhq_000901km_cbd_con_tech_method.pdf,
Last Accessed 13th of June 2003.

[Meyer 1999] Meyer B., Mingins C.,
Component-Based Develoment; From Buzz to Spark,
IEEE Software, 0018-9162/99, 1999.

[Mogul 2003] *Homepage of Mogul*,
http://www.mogul.com/ir/en/info/nyckeltal.xls,
Last Accessed 11th of June2003.

[Morisio et al. 2002] Morisio M., Ezran M., Tully C.,
Success and Failure Factors in Software Reuse,
IEEE Trans Software, vol.28, no.4, p. 340-357, April 2002.

[Ommering 2002] Ommering R. v.,
Building Product Populations with Software Components,
ICSE'02, 1-58113-472-X/02/005, May 2002.

[Rational 2003] *Homepage of Rational ClearQuest*,
http://www.rational.com/products/clearquest/index.jsp,
Last Accessed 11th of June 2003.

[Reifer 1997] Reifer D.J.,
Practical Software Reuse; Strategies for Introducing Reuse Concepts in Your Organization,
John Wiley & Sons, Inc., ISBN: 0-471-57853-3, 1997.

[Schmid 2002 a] Schmid K.,
A Comprehensive product line scoping approach and its validation, Proceedings of the 24th international conference on software engineering, 2002.

[Schmid 2002 b] Schmid K., Verlage M.,
The Economic Impact of Product Line Adoption and Evolution,
IEEE Software, 0740-7459/02, July/August 2002

[Schreiber 2001] Schreiber A.,
Change Management and Evolution Support,
Eureka ? ! 2023 Programme, ITEA project 99005, 2001.

[SEI 2003] Carnegie Mellon University, Software Engineering Institute,
A Framework for Software Product Line Practice Version 4.1,
http://www.sei.cmu.edu/plp/frame_report/introduction.htm,
Last Accessed 11th of June 2003.

[Sindre et al. 1995] Sindre G., Conradi R., Karlsson E-A.,
The REBOOT Approach to Software Reuse,
Journal of Systems and Software, Vol. 30 No.3, p.201-212, September1995.

[Sneed 1994] Sneed H.M.,
Planning the Reengineering of Legacy Systems,
IEEE Software, 0740-7459/94, p.24-34, January 1995.

[Svahnberg et al. 2000] Svahnberg M., Bosch J.,

Issues Concerning Variability in Software Product Lines,
IW-SAPF-3, LNCS 1951 pp.146-157,
Springer-Verlag Berlin Heidelberg, 2000.

[Tanacea 2003] Tanacea D.,
Component-Based Software Development: Why Hasn't the Vision Met Reality?,
http://www.computerworld.com/developmenttopics/development/story/0,10801,
78077,00.html, Last Accessed 11th of June 2003.

[Wohlin et al. 2000] Wohlin C., Runeson P., Höst M., Ohlsson M.C., Regnell B.,
Wesslén A.,
Experimentation in Software Engineering, An Introduction,
Kluwer Academic Publishers, ISBN: 0-7923-8682-5, 2000.

Appendix

The appendices present relevant information to this thesis.

Appendix A gives a technical description of the migration from the monolithic to the component-based system.

Appendix B is the questionnaire given to the developers at Mogul.

Appendix C shows a table of the abbreviations used in this thesis and particularly in appendix A.

Appendix A Confidential

This appendix has been excluded from the public version due to reasons of confidentiality.

Appendix B Questionnaire of the software development process and reuse at Mogul, Trondheim

The purpose of this questionnaire is to map how employees at Mogul view their current development process – how change requests and errors are handled. The emphasis is on reuse, not only in current development but also how the process ensures reuse across future products.

The questionnaire consists of multiple-choice questions. Please mark the choice **(-1-one)** you feel is most correct. The responses of this questionnaire will be handled anonymously and confidential.

The questions are enumerated by the category they belong to, and a number. The components category has questions C1, C2, C3 and so on. If a question is depends on a particular answer from the prior question, these two questions are enumerated by an “a” and “b”. C4a and C4b are two such questions. Feel free to add comments.

Estimated time to complete: 15 minutes.

Terminology

Throughout this questionnaire some terms will be used. The following specification is provided for clarifying purposes.

Component – The reusable components developed through the Kamelon-project (e.g. AccountInfo).
Development Process – The current workflow related to how the change requests and errors are handled.
Product – An application using the components (e.g. the Internet bank).

Personal Information – text answers:

P1: What is your current role?

____

____

P2: How long have you been working in the organization?

____

____

P3: On what problem do you currently work (e.g. error, change)?

____

____

P4: For how long have you been programming?

____

____

P5: What is your education degree?

____

____

General - check one option in this part

G1: *There are several benefits to reuse components (definition of components given in the guidelines). How important do you consider reuse in achieving the following benefits: [check one option]*

G1a: *For achieving lower development costs, reuse is of ______ importance:*
[ ] very high [ ] high [ ] medium [ ] little [ ] no [ ] don't know

G1b: *For achieving shorter development time, reuse is of ______ importance:*
[ ] very high [ ] high [ ] medium [ ] little [ ] no [ ] don't know

G1c: *For achieving higher product quality, reuse is of ______ importance:*
[ ] very high [ ] high [ ] medium [ ] little [ ] no [ ] don't know

G1d: *For achieving a more standardized architecture, reuse is of ______ importance:*
[ ] very high [ ] high [ ] medium [ ] little [ ] no [ ] don't know

G1e: *For achieving lower maintenance costs (including technology updates), reuse is of ______ importance:*
[ ] very high [ ] high [ ] medium [ ] little [ ] no [ ] don't know

Comments:

____
____
____
____
____
____

G2: How useful / important do you find the following: [check one option]

G2a: Reuse / component based technologies are of ______ importance:

[ ] very high [ ] high [ ] medium [ ] little [ ] no [ ] don't know

G2b: OO technologies (java, UML, CORBA) are of ______ importance:

[ ] very high [ ] high [ ] medium [ ] little [ ] no [ ] don't know

G2c: Testing is of ______ importance:

[ ] very high [ ] high [ ] medium [ ] little [ ] no [ ] don't know

G2d: Inspections are of ______ importance:

[ ] very high [ ] high [ ] medium [ ] little [ ] no [ ] don't know

G2e: Formal specifications / methods are of ______ importance:

[ ] very high [ ] high [ ] medium [ ] little [ ] no [ ] don't know

G2f: Configuration management is of ______ importance:

[ ] very high [ ] high [ ] medium [ ] little [ ] no [ ] don't know

Comments:

____

____

____

____

____

____

G3: How useful / important do you consider the following artefacts with respect to reuse: [check one option]

G3a: Requirements are of ______ importance with respect to reuse:

[ ] very high [ ] high [ ] medium [ ] little [ ] no [ ] don't know

G3b: Use cases are of ______ importance with respect to reuse:

[ ] very high [ ] high [ ] medium [ ] little [ ] no [ ] don't know

G3c: Design is of ______ importance with respect to reuse:

[ ] very high [ ] high [ ] medium [ ] little [ ] no [ ] don't know

G3d: Code is of ______ importance with respect to reuse:

[ ] very high [ ] high [ ] medium [ ] little [ ] no [ ] don't know

G3e: Test data/documentation is of ______ importance with respect to reuse:

[ ] very high [ ] high [ ] medium [ ] little [ ] no [ ] don't know

G4: How would you describe the current level of reuse in the organization?

[ ] very high [ ] high [ ] medium [ ] little [ ] no [ ] don't know

G5: To what extent do you feel affected by the reuse in your work?

[ ] very high [ ] high [ ] medium [ ] little [ ] no [ ] don't know

Comments:

____

____

____

____

____

____

Components – check one answer in this part

C1: *How widely reused are the components developed in the Kamelon-project?*

[ ] reused in several products

[ ] reused in some products

[ ] not reused at all

[ ] don't know

C2: *A reusable component is usually:*

[ ] less affected of change requests than a component that is created from scratch

[ ] about equal to a component created from scratch

[ ] inferior to a component created from scratch

C3: *A reusable component is usually:*

[ ] less affected of errors than a component that is created from scratch

[ ] about equal to a component created from scratch

[ ] inferior to a component created from scratch

C4a: *Are the components sufficiently documented?*

[ ] yes [ ] sometimes [ ] no [ ] don't know

C4b: *If 'Sometimes' or 'No': Is this a problem?*

[ ] yes [ ] no

C5: *Integration when reusing a component – your experiences through your work?*

[ ] usually works well (the components usually fit easily into the architecture)

[ ] may cause some problems

[ ] is difficult (hard to fit component into architecture)

C6: *Is any extra effort put into testing/documenting reusable components when modifications have been made?*

[ ] yes

[ ] no

C7: Would the construction of a reuse repository, with extra component documentation etc:

[ ] not be worthwhile: the current system works sufficiently

[ ] be worthwhile: make finding/reusing components easier

C8: How are errors in reusable components handled?

[ ] they are reported to asset supporter and fixed by him

[ ] they are fixed in the application

[ ] they are fixed in the application and the framework by the application developer

[ ] don't know

[ ] other:

C9: How is components usually reused?

[ ] reused as-is [ ] configured [ ] modified

C10: How would you decide to reuse a component 'as-is', reuse with modification, or to make a new component from scratch?

[ ] by consulting experts

[ ] by following guidelines

[ ] not clearly defined

C11: Do you feel that the process of finding, assessing and reusing existing code / design components is functioning?

[ ] yes

[ ] no

Comments:

____

____

____

____

____

____

Requirements – check one answer in this part

In most software development projects, the initial set of requirements may change during the course of the projects. The customers may think of new features they want to add, the developers may find some requirements unfeasible (or possibly easier) to fulfil and so on. In such cases it may be necessary for the stakeholders to renegotiate requirements.

R1: *Is the organization's requirements renegotiation process working sufficiently?*

[ ] yes

[ ] no

R2: *In a typical project:*

[ ] requirements are usually flexible, and may often be adjusted.

[ ] no particular trend (sometimes rigid, sometimes flexible).

[ ] requirements are usually very rigid, little or no change can be negotiated

R3: *Are requirements often changed / renegotiated during a development project*

[ ] often

[ ] sometimes

[ ] seldom

Comments:

____

____

____

____

____

____

Architecture – check one answer in this part

A1: Do you know the product architecture (definition of a product is given in the guidelines) well?

[ ] yes

[ ] no

A2: How would you describe your knowledge of the components' source code?

[ ] very high [ ] high [ ] medium [ ] little [ ] not at all [ ] don't know

A3a: Do you know the existing architecture of the components well?

[ ] yes

[ ] no

A3b: Do you know the interfaces of the components?

[ ] yes

[ ] no

A3c: Do you know the design rules of the components?

[ ] yes

[ ] no

A4a: Is the architecture sufficiently documented?

[ ] yes

[ ] no

A4b: If 'No', how would you prefer the documentation?

[ ] text

[ ] web pages

[ ] described through models

[ ] other: ________________________________

A4c: If 'No', what is the problem with the documentation?

[ ] it is not up to date

[ ] it is not complete

[ ] it is difficult to understand

[ ] other: ________________________________

A5: *What is your main source of guideline information during Implementation?*

[ ] other developers

[ ] previous work

[ ] other: ________________________________

A6: *Are criteria for design regarding non-functional requirements (especially performance and stability) well defined?*

[ ] yes

[ ] no

A7: *Do you test a component after modification for non-functional properties before integration with other components (or are only the functional requirements are tested?)*

[ ] yes

[ ] sometimes

[ ] no

A8: *Were you involved in the Kamelon-project?*

[ ] yes

[ ] no

Comments:

____

____

____

____

____

____

Development Process – check one answer in this part

The current process of handling change requests and errors does not evaluate the effects of the modification of a component on other possible products using that component.

DP1: How would you describe your organization's development process?

[ ] formal and well documented

[ ] the organization have a formal process description, but an informal, not documented is used

[ ] there exists no process

[ ] does not apply

DP2: Is your organization's software development process evaluated periodically?

[ ] yes [ ] no [ ] does not apply [ ] don't know

DP3: Are the processes of developing and maintaining software products coordinated across the organization?

[ ] yes [ ] no [ ] does not apply [ ] don't know

DP4: Do you think that the current development process support finding new potential reusable component/assets?

[ ] yes [ ] no [ ] does not apply [ ] don't know

DP5: Do you think that the current development process support using existing components/assets?

[ ] yes [ ] no [ ] does not apply [ ] don't know

DP6: Does the current development process support maintenance of reusable components/assets?

[ ] yes [ ] no [ ] does not apply [ ] don't know

DP7: Is the development process modified after the Kamelon-project?

[ ] yes [ ] no [ ] does not apply [ ] don't know

DP8: Will improving the development process to include the effects on other products

______:______

[ ] have little or no impact on the development process.

[ ] have positive effects / improve development process.

[ ] have negative consequences (result in more work / give a more complex and less helpful development process / have other negative effects)

DP9: Will introducing reuse activities in the development process have positive effect?

[ ] yes [ ] no [ ] don't know

Comments:

____

____

____

____

____

____

Thank You for Your Time and Effort in Completing the Questionnaire!

Appendix C Abbreviations

CBSE	Component-Based Development.
TD2	TouchDown version 2. Platform for Internet bank before Kamelon.
TDE	TouchDown Europe. Architecture and guidelines for the development of distributed applications in Sparebank 1.
Kamelon	The project of migration of the Internet bank to a distributed J2EE-based architecture.
NAPP	Internet bank application. The definition is used on the WEB-application Infovention developed on the TD2 platform.
KNAPP	Internet bank application in a Kamelon context. Used about the components (mainly JSP and Servlets) that is the WEB-application in the Internet bank.
Kamelon-components	The EJB-components developed in the Kamelon project. The Kamelon-components are a subset of the TDE-components.
TDE- components	The EJB-components buildt on a TDE-like architecture. Today, only the C4 in addition to the Kamelon-components.
KL	Communication layer, Components inherited from TD2 that communicate with systems at Fellesdata, BBS, Odin and Accept. Communicating with Fellesdata og BBS through mechanisms in Java, while JOLT/TUXEDO server are used for the others.
C4	Common Credit Card Components. Family of components that implement access to credit card information i PRIME at EnterCard.

JOLT/TUXEDO Server	Implementing communication against systems without Java compatible interfaces (Odin and Accept).
Transfer Objects	Architecturemechanism from TDE to transfer data between distributed komponents. It was evaluated as a part of the migration.