NTNU

Norwegian University of
Science and Technology

Study of Software reuse at Skattedirektoratet

Line Ånderbakk Olsen
Thor Ånderbakk Olsen

Master of Science in Informatics
Submission date: May 2008
Supervisor: Reidar Conradi, IDI
Co-supervisor: Tore Hovland, Skattedirektoratet

Norwegian University of Science and Technology
Department of Computer and Information Science

Abstract

This master thesis is a case study on software reuse within a subset of systems at the Norwegian Directorate of Taxes, Skattedirektoratet (SKD). The systems chosen for our research are the GLD systems; legacy systems which dates back to the late 1980's and early 1990's. Because of historical reasons, these systems are copied and created over and over again in an annual cycle. There are redundancies in code and data between the annual versions, but also across the different GLD systems. The consequence of this is systems with reduced maintainability and possible inconsistencies in code and data.

Our objectives with this case study is to determine both the current level of software reuse within a subset of the GLD systems, and the emphasis on reuse in SKD's development process. After determining the status of as-is, we will continue with an investigation of the potential for software reuse within the context of SKD, and how they can achieve systematic software reuse.

The contributions of this thesis can be divided into four main themes:

- **T1:** Review of state-of-the-art literature on software reuse
- **T2:** Investigation of reuse level within selected GLD systems
- **T3:** Investigation of SKD's development process
- **T4:** Investigation of opportunities for systematic reuse in SKD

The main contributions are:

- **C1:** Review of literature in the field of software reuse
- **C2:** Measurement of the reuse maturity level within the selected GLD systems.
- **C3:** Survey of the software development process and reuse aspects at SKD
  - **C3.1:** Results from SKD
  - **C3.2:** Results from SKD combined with results from previous surveys on software developers attitude toward software reuse by NTNU
- **C4:** Process which assures reuse

Keywords: Systematic software reuse, Reuse maturity, Software development, Software engineering

Preface

This thesis was written as the concluding part of the Master of Science degree in Informatics.

First of all we would like to thank our supervisor, Reidar Conradi, for his continuous support, feedback and advice during this master thesis. Second we want to thank Tore Hovland and the developers working with the GLD system for their helpfulness during our data acquisition, and the people from Skattedirektoratet who participated in the survey and interview.

Trondheim, 30. May 2008

Line Ånderbakk Olsen Thor Ånderbakk Olsen

Contents

List of Figures

List of Tables

Abbreviations

ADF	Application Development Framework by Oracle, a commercial Java framework for creating enterprise applications
Altinn	E-government portal for electronic dialogue between industry and public authorities
BATCH	Batchwise data processing with insignificant or no user involvement, opposite to online or interactive data processing
CASE	Computer-Aided Software Engineering
CICS	Customer Information Control Systems - Program for control of transactions to IBM mainframe. Is used for interactive data processing, online entry in database
COBOL	COmmon Business-Oriented Language, which is a third generation programming language especially designed for administrative computer systems
COTS	Commercial-Off-The-Shelf
DB2	Relational Database Management System from IBM
DSB	Computer-aided tax return treatment - used locally on tax offices (Datastøttet selvangivelsesbehandling)
ESB	Enterprise Service Bus
FLT	Simplified salary and deduction statements
GLD	Assignments (basis data) from third party
GLDB	The GLD database
GUI	Graphical User Interface
HR	Human Resources
IBM	International Business Machines Corporation
ISO	International Standardization Organization
JCL	Job Control Language, for execution of for example COBOL programs
LTO	Salary and deduction
MAG	Modernization of basis data
MVC	Model-View-Controller
SG2	System group 2 in the IT-department
SKD	The Norwegian Directorate of taxes
SOA	Service Oriented Architecture
OO	Object Oriented
PhD	Doctor of Philosophy - an advanced academic degree awarded by universities
PSA	Prefilled tax return
REXX-routine	A menu made in REXX, that is used to initiate executions on basis data, print out control lists, copy disks, withdraws paper assignments and such
RMM	Reuse Maturity Model
SEI	Software Engineering Institute
SL	System for assessment (System for Likning)
SPC	Software Productivity Consortium
SPSS	Statistical Package for the Social Sciences
NTNU	Norwegian University of Science and Technology

Part I

Introduction

Chapter 1

Introduction

This chapter presents our motivation for doing this Master's thesis, followed by a definition of the problem and the research questions and goals. We briefly present our contributions and themes which will be used throughout this thesis. Finally, the outline of the thesis is presented.

1.1 Motivation

Both of the authors have been programming for five years, in projects, assignments, and as software developers at different companies. One thing we have noticed that does not cease to amaze us, is the feeling of déjà vu when programming. And it's not just a feeling, it's a fact. The module for reading and writing files, the component which accesses the database, your MVC-patterned web implementation or your first implementation of the quicksort-algorithm you were once so proud of. It has all been done before, but still we keep on writing the same code over and over again. Occasionally when we know that the exact problem has been solved before, we search our entire hard drives for that one piece of code fragment, written several years ago, located in a file we can not remember. If you are lucky enough to find the particular file and code sample, it is not necessarily as easy as copy/paste into your current project. The little piece of code might need comprehensive re-writing in order to work in your new software, and every time source code is changed new errors might be introduced. The whole task can only be described as time-consuming, and as the old saying goes: "Time is money".

What if the code or component written for a earlier system had been designed and implemented with the intention that it would be reused in a later project? And what if this component was stored in library, along with documentation and several other components created for other purposes. The idea is promising, develop a component once, and create new software systems from the reusable components. A good parallel to this is to consider a software component as a brick of Lego. The brick alone is rather useless, but it's simplicity lets you combine it with other bricks, building large, complex structures. The software community is still a far away from creating systems as easily as children puts the Lego bricks together, but we do believe that a systematic approach to reuse can contribute to more efficient development of software systems, and reduce costs, development time and errors.

1.2 Problem Outline

This master thesis is a case study on software reuse within a subset of systems at the Norwegian Directorate of Taxes, Skattedirektoratet (SKD). The systems chosen for our research are the GLD systems; legacy systems which dates back to the late 1980's and early 1990's. Because of historical reasons, these systems are copied and created over and over again in an annual cycle. There are redundancies in code and data between the annual versions, but also across the different GLD systems. The consequence of this is systems with reduced maintainability and possible inconsistencies in code and data[61].

1.3 Research Questions and Goals

Our objectives with this case study is to determine both the current level of software reuse within a subset of the GLD systems, and the emphasis on reuse in SKD's development process. After determining the status of as-is, we will continue with an investigation of the potential for software reuse within the context of SKD, and how they can achieve systematic software reuse. In addition, our research initially tried to answer if reused components are more stable than non-reused components, but we were not able to answer that question with our collected data. The research questions are:

- **RQ1:** What is the current state of software reuse in the selected GLD systems?
- **RQ2:** Do reused components have lower change and defect rate compared to other components?
- **RQ3:** What is the emphasis on software reuse in the current development process?
- **RQ4:** What is the potential for systematic reuse, and how can it be achieved?

In addition to the research question, SKD inquired that two organization-specific goals would be added to the research

- **SKD goal 1:** Propose a process which assures software reuse
- **SKD goal 2:** Propose an ideal architecture for GLD, with focus on reuse

1.4 Our Contribution

The contributions of this thesis can be divided into four main themes:

- **T1:** Review of state-of-the-art literature on software reuse
- **T2:** Investigation of reuse level within selected GLD systems (RQ1)
- **T3:** Investigation of SKD's development process (RQ3)
- **T4:** Investigation of opportunities for systematic reuse in SKD (RQ4, SKD goal 1 and 2)

The main contributions are:

- **C1:** Review of literature in the field of software reuse

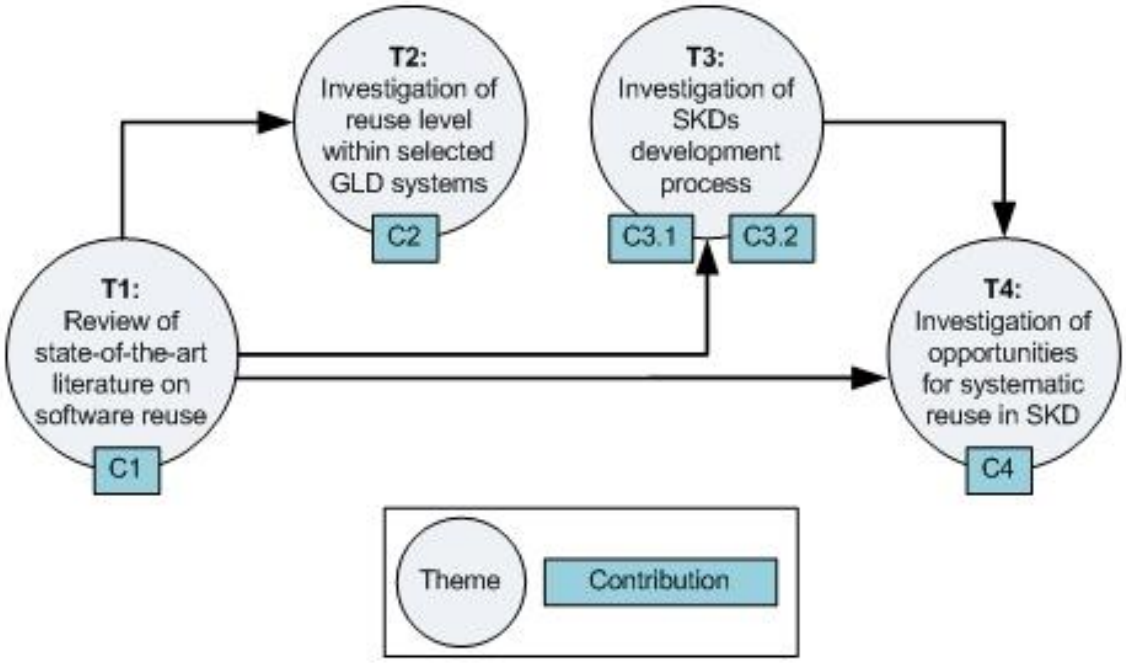

Figure 1.1: Themes and contributions

- **C2:** Assessment of the reuse maturity level within the selected GLD systems
- **C3:** Survey of the software development process and reuse aspects at SKD
  - **C3.1:** Results from SKD
  - **C3.2:** Results from SKD combined with results from previous surveys on software developers attitudes toward software reuse by NTNU
- **C4:** Process which assures reuse

Research Questions	Themes	Contributions
RQ1	T1, T2	C2
RQ2		
RQ3	T1, T3	C3.1, C3.2
RQ4	T1, T4	C4

Table 1.1: Relationship between research questions, themes and contributions

Figure 1.1 shows the different themes in our thesis in a chronological order, and how the contributions relate to them. Table 1.1 shows the relationship between our research questions, themes and contributions. RQ2 remains unanswered in this thesis, due to lack of data resources, and is therefor not covered by any of the themes or contributions.

1.5 Thesis Structure

This thesis is organized into seven parts as following:

Part I describes our motivation for this thesis, problem definition, research questions and goals, and our contribution.

Part II contains a literature review on State-of-the-Art of software reuse, where we present a definition of software reuse, terms, approaches, metrics and benefits/problems associated with reuse. We proceed with an introduction of the Norwegian Directorate of Taxes (SKD), and the GLD systems, which was the context and subject for our research.

Part III presents the activities performed in this project, and our choice of research methods. The research questions are revisited, and we also connect each research question with a corresponding theme. The themes are used throughout the report in order to organize our research and results.

Part IV contains results concerned with two of the themes: Investigation of the reuse level within the GLD systems and investigation of SKD's development process.

Part V contains discussions of our results, organized into corresponding themes.

Part VI presents our conclusion, and suggestions to further work.

Part VII contains appendices: the original task description, resumes of meetings and interview, and the survey and its results.

Part II

Literature Review and Research Context

Chapter 2

State-of-the-Art of Software Reuse (T1)

Ever since libraries of shared components were first suggested by Doug McIlroy in 1968[30], software reuse has been recognized as an attractive idea with an obvious payoff. Individuals and small groups have always practiced ad-hoc software reuse[16] by reusing ideas, abstractions and processes[43]. Nowadays, software reuse is becoming increasingly necessary for competitive reasons such as decreased time-to-market and lower costs. This mitigates towards a more organized approach to reuse[43]. This chapter contains a literature review on the field of software reuse. A great deal of research has been performed in this field in the last decades, and the existing information is comprehensive. In jeopardy of overwhelming both our readers and the size of the report, we have narrowed down to topics that are of relevance to our research.

We will start by giving a definition of systematic software reuse and why software reuse is important. Both benefits and problems will be presented. As software reuse comes in many different forms, and the terms are often mixed in the literature, we have tried to present this in a sensible way. We will also explain why changes to both the development process and organization are required in order to achieve systematic reuse. Measurement is of high importance for achieving systematic reuse, thus a presentation of metrics and models will be given. Because we have performed a survey on developers' attitude towards software reuse at SKD, we present two similar studies from NTNU. The last section has nothing to do with software reuse, but gives a presentation of empirical strategies that we have used in our project.

2.1 Software Maintenance

Increasingly more software developers are employed to maintain and evolve existing applications instead of developing new systems from scratch[36]. Software evolution is a necessary consequence of the inevitable software maintenance process[26], and it is the process of modifying a software system or a component in order to correct faults, improve performance or other attributes, or adapt to a changed environment. The maintenance of existing software can occur for over 60 percent of all effort carried out by a development organization, and the percentage continues to increase as more software is produced[74].

Reengineering existing applications to be web enabled is a issue in many software development teams today[1]. Substantial efforts are now made in rearchitecting legacy systems into more maintainable and functional system families, generally oriented at distributed applications[9].

Osborne and Chikofsky[40] have the following description of software, which we find quite suiting despite it was written almost 20 years ago:

> Much of the software we depend on today is on average 10-15 years old. Even when these programs were created using the best design and coding techniques known at the time (and most were not), they were created when program size and storage space were principal concerns. They were then migrated to new platforms, adjusted for changes in machine and operation system technology and enhanced to meet new user needs - all without enough regard to overall architecture.

These types of changes to a system can have impact on the system's internal structure and complexity. Software evolution may therefore cause decline in software quality and erosion of software architecture over time[6].

2.2 Introduction to Software Reuse

Software reuse is the process of creating new software systems from existing systems, knowledge or artifacts rather than building them from scratch[50][15][17]. Reuse can be found in many different forms from ad-hoc to systematic and from white-box to black-box reuse[50][47]. These terms will be described further in this chapter. Software reuse can also be specified in two directions: development *for* reuse and development *with* reuse [53][49]. Development for reuse relate to systematic generalization or components for later reuse, while development with reuse relates to how existing components can be reused in new applications and systems.

Software is seldom built entirely from scratch[50]. Frakes and Terry[17] gives several examples of reusable assets from software projects which can be copied and adapted to fit new requirements:

- Architecture
- Source code
- Data
- Design
- Documentation
- Estimates (templates)
- Human interfaces
- Plans
- Requirements
- Test cases

2.2.1 Ad-hoc versus Systematic Reuse

Software reuse usually takes to different forms: ad-hoc or systematic[14], where the first is the most common form[50]. Ad-hoc reuse is an informal process, where no methods for reuse are defined. This type of reuse usually happens by chance[34] and individual developers are responsible for identifying and locating reusable components. Many developers have successfully performed this type of reuse, primarily by cutting and pasting code snippets from existing programs into new ones[51]. This may work fine for a while for individual developers or small groups, but it does not scale up across business units or enterprises to provide systematic reuse[51]. The increase of productivity from this type of reuse is only marginal[34]. Ad-hoc reuse is also known as individual reuse or opportunistic reuse[50] .

Systematic reuse requires up-front efforts and investment to define guidelines and procedures [50], and to monitor and measure changes in the process at the organizational level to ensure regular reuse[34]. Systematic means that the process is consistent and repeatable, and follows a logical sequence of events[17][16]. Domain models and architecture are important concepts that are used for specifying and designing a new system in a particular domain or application area[45][16]. This requires changes to the development methods and organizational structure. Systematic reuse is also referred to as institutionalized reuse or planned reuse[50] .

The following advantages of systematic reuse are gathered from Prieto-Dìaz[44]:

- Makes software reuse an integrated part of software development
- Makes software reuse a standard practice
- Can help towards better software methodology
- Makes everybody a participant
- Promises a reuse culture

Frakes[16][14] claimed, in the mid 90s, that systematic reuse is a paradigm shift in software engineering where one went from building single systems to building families of related systems. The related systems share common parts and vary in certain regular and detectable ways. This new paradigm is domain focused, based on repeatable processes and centers around reuse of higher level life cycle artifacts such as requirements, design, subsystems etc. Sametinger[50] argues that some people might think that Frakes idea of a paradigm shift is too extravagant, but justifies it with that software reuse and software components has provoked so many changes and innovations to the way we perform software engineering.

2.2.2 Benefits from Software Reuse

Software Reuse is widely believed to be the most promising technology for significantly improving software quality and productivity [72][67][50]. The quality of a program or component increases every time the item is reused, since error fixes accumulates from reuse to reuse[50]. In the construction phase, reuse can increase productivity with 20%[5]. Effort required in other phases are also reduced; developers need to create less code, and can minimize the redundant work and enhance the reliability of their work because each reused component has already been created, tested and documented in the course of its

original development [20][16][50][27]. This is also supported by other statistics[7] which indicates that only 15% to 40% of developed code is code that defines a new purpose. The remaining 60% to 85% are most likely components that could have been created as reusable assets. Software reuse makes it easier to estimate project costs and delivery time. Accelerated development can be achieved because both development and validation time should be reduced.

Software reuse can also improve the way software systems work together, their interoperability, if several systems use the same components for the interfaces[50]. When developers reuse components over time, they will become familiar with these components. Costs are reduced as the productivity increases and time spend on training and testing is reduced. Another important cost reducing factor is that the development can be conducted by smaller teams with software reuse[50].

To summarize benefits that software reuse can achieve:

- Improved quality, since errors can be discovered every time the item is reused
- Increased productivity, since less code needs to be created. As more artifacts are reused, such as design, tests and documentation, the productivity increases even further
- Better interoperability between software systems
- Reduced costs due to less training, maintenance and production time, and smaller teams

Successful Reuse Programs

Prieto-Dìaz[44] defines a reuse program as an organizational structure and collection of support tools, which is aimed at fostering, managing, and maintaining the practice of reusing software in an organization.

Research studies have consistently shown that reusing technology has the greatest potential to reduce the cost of software [8], where some reuse programs have achieved from 30 to 80 percent reuse. Hitachi's Eagle Eye environment has shown reduced time to market, defect density, maintenance cost and overall software development costs [20]. Hewlett-Packard improved time to marked, higher quality of their systems and lower development costs. This increased as the levels of reuse and sophistication of the reuse program increased. But experience has shown that it takes time, investment and experience with software reuse in each organization to get those levels of reuse. Other organizations have achieved highly customizable products, increased market agility and consistent families of related products that provide familiar, compatible interfaces to many customers [20].

Ericsson, the Swedish telecommunication company, had great success with its reuse program. A strategic decision was made early on to architect and implement the product for substantial reuse and evolution. The need for extensive initial investment and long-term organizational commitment, a well-designed architecture, support for various configurations and structuring the development organization to match the system architecture, showed to be of vital importance[20].

2.2.3 Problems associated with Software Reuse

Software reuse has proved to be difficult to achieve[16], and far from all reuse programs succeed [8]. Earlier software reuse was seen as a technical problem, but now we see that it is not that simple [71]. Reuse must address issues such as managerial, economic, cultural, technology transfer, in addition to the technical ones[46][47]. The root cause identified by Morisio et al.[35], was a lack of commitment by top management, or non-awareness of the importance of those factors, often coupled with the belief that using the object-oriented approach or setting up a repository seamlessly is all that is necessary to achieve success in reuse. Organizations attempting to implement systematic reuse (a software reuse program), face therefore both technical and non-technical problems[16].

There is a significant amount of additional effort required in both the initial design and adoption of an architecture which supports the idea of software reuse. Software components must be easily retrieved from a component library, understood and sometimes adopted to work in a new environment[67]. There are considerable costs associated with understanding whether a component is suitable for reuse in a particular situation, and in testing the component to ensure dependability[67]. Technical problems therefore includes interchangeability and classification, cataloging and retrieving software components[48][47], in addition to problems with tools, standards and technology.

Nontechnical problems includes the human factor and cultural-, economic- and organizational issues. When several components should be developed and reused across a suite of applications, it must also be supported by both the inherent process and the organization[20]. We will come back to nontechnical aspects of software reuse in section 2.4 on page 20.

2.3 Technical Aspects of Software Reuse

2.3.1 Reuse Perspectives

Software reuse does not only apply to fragments of code[17] as mentioned in section 2.2, but to all life cycle products such as documentation, system specifications, architectures, functions, tests and pseudo-code. Ideas and concepts can be reused just as well as entire applications. It can be conducted by individuals, groups or entire organizations. Due to the widespread of terminologies, definitions, types and approaches in the field of software reuse, we have used a taxonomy suggested by Prieto-Dìaz[46]. He identified six perspectives, or facets, from which to view software reuse. These facets were:

- Substance - Defines the essence of the items which are to be reused
- Scope - Defines the extent of reuse
- Mode - Defines how the reuse is conducted
- Technique - Defines the approach used to implement reuse
- Intention - Defines how the items will be reused
- Product - Defines which items, or work products, are reused

Table 2.1 is based on Prieto-Dìaz's six perspectives, and shows the facets with additional examples. The first column in the table is "Substance". An example of reusable ideas

Substance	Scope	Mode	Technique	Intention	Product
Ideas, concepts, artifacts, components, procedures, skills	Vertical, Horizontal, domain-specific, general-purpose, internal, external, small-scale, large-scale	Planned, systematic, ad-hoc, opportunistic, institutionalized, individual	Compositional, generative	Black-box, as-is, white-box, modified, glass-box, by adaption	Source-code, design, specifications, objects, text, architectures, documentation

Table 2.1: Six perspectives from which to view software reuse[46]

and concepts is when a developer uses an algorithm from a book. The algorithm tells *how* the problem should be solved, but the developer must still use a specific programming language and implement the solution herself. When the component has been developed, the component also becomes a reusable asset. Concepts can also be designed so they can be configured and adapted for a range of situations such as design patterns, configurable system products and program generators[67]. Prior studies in the field of software reuse distinguishes between reuse across horizontal and vertical domains[5], as seen under "Scope" in the table. Horizontal reuse refers to software components that are used across a wide variety of application areas, such as library components, software drivers or graphical user interface functions. Commercial off-the-shelf (COTS) components are also often referred to as horizontal reuse. Vertical reuse is harder to achieve, but has a greater potential when it comes to the benefits of software reuse. Vertical reuse embraces entire functional areas, or domains, and can occur if the majority of applications are representative of a single kind of data processing activity[5] (family of systems). Internal reuse is when a component is used multiple times within the system it was developed for[50], while external reuse applies to the use of a component developed for another software system. Small-scale reuse is the "common way" of developing software, by reusing simple classes and functions.

Table 2.1 lists two techniques for developing software systems with reuse. Compositional reuse is the composing of available lower-level components into a larger system[50]. In such cases, the components are retrieved from a software repository, as described in 2.3.4. When adopting reference- or generic architectures and standard interfaces for components, this is referred to as *generative reuse*[50]. The fifth facet, intention, describes how the items will be reused. This depends on the visibility of the internals within the component[50]. If a component is white-box, we have access to all its internals, and can modify it in any way we feel appropriate[47]. Black-box is the opposite, it must be used as-is. When using components from libraries, such as in Java, these are often compiled classes and thus black-box. Glass-box is a combination of white- and black-box, their internals are visible but they can not be modified. Morisio et al.[34] states that white-box reuse entails less cost and risk, and is more feasible since it does not require specific personnel in the organization for developing and maintaining the reusable assets. The last facet in table 2.1, product, concerns all the artifacts that can be reused.

2.3.2 Approaches to Software Reuse

Krueger[25] lists 8 approaches to software reuse. These are given in the list below, and are ranked on how well they "Reduce the amount of intellectual effort required to go from the initial conceptualization of a system to a specification of the system in abstractions of the reuse technique". As the rankings shows, source code reuse (code scavenging) is not the most effective way of developing software.

1. Application Generators
2. Software Architectures
3. Transformational systems
4. Very high-level languages
5. Software schemas'
6. Source code components
7. Code scavenging
8. High-level languages

Application generators

Application generators can create customized applications based on patterns[50]. Application generators focuses on a narrow domain, and translates input specifications into executable programs. They can be used when[25]:

- Many similar systems are written
- One software system is modified or rewritten several times during its lifetime
- Many prototypes of a system are necessary to converge on a usable product

There is only a limited availability of application generators, and it can be difficult to find a generator for a specific type of software. Building a general-purpose application generator has proved to be complicated[25].

Software Architectures

Reusable software architectures are software frameworks and design that captures the global structure of a system[25]. The software architecture represents a substantial effort in both design and implementation, and if reused, it can offer a significant leverage in the development process. The scope of this type of reuse is large-scale [25], and it is a difficult task to create a reusable general-purpose software architecture.

Source Code Components

Source code is the most common reuse product[50]. Source code components are similar to code scavenging since both approaches involves that developers copy existing source code into new systems. Source code components are developed as reusable components, and

are stored in catalogs or libraries. The main problem with this approach is to retrieve the stored components. It is unreasonable to believe that a developer will spend much time searching trough a large library for an appropriate component[25], and if the process of finding the component takes too long, the developer will most likely build it herself.

Code Scavenging

Code scavenging is an ad-hoc approach of reusing design or source code, where existing software systems are scavenged for useful code fragments[25][50]. By copying existing code, the number of keystrokes and time needed for developing software are reduced. The scavenging approach separates between *code scavenging* and *design scavenging*. In code scavenging, a continuous block of code is copied from an existing system and pasted into the new system. Design scavenging is when a larger, continuous block is copied and used as a global template for a new system. There are several disadvantages with the scavenging approach. If a developer should be able to find code a fragment to scavenge, she must be able to remember or know where to find it. Most likely, the code fragment must be customized or specialized in order to resolve the conflicts between the existing and new system. This means that the developer must manually modify the fragment, and by doing so introduce new errors. Thus, the code fragment must be validated, tested and debugged as thoroughly as when it was first developed. Also, the developer is forced to become closely involved in the implementation details of the source code, since scavenging offers no abstraction[50].

High-level Languages

High-level languages applies to programming languages that are closer to human language than machine language, such as C, Lisp, Ada, Smalltalk, COBOL, C++, Java etc. High-level languages offers a relatively small numbers of reusable artifacts in the form of language constructs which a programmer can choose from[25]. Such constructs can for example be an if-statement. High-level languages are usually not treated as examples of software reuse, although their goals are similar to those of software reuse[50]. Frakes and Fox[15] conducted a survey of organizations from the U.S and Europe, and found that the choice of programming language does not affect code reuse levels. Contrary to popular belief, efforts to increase reuse levels should focus on other factors besides programming languages. Most software development organizations move to object oriented technology because engineering managers believe that this will lead to significant reuse. Unfortunately, without an explicit reuse agenda and a systematic reuse-directed software process, most of these object adoption efforts do not lead to successful large-scale reuse[20]. Practice has also been quite successful with non-object oriented languages such as COBOL and Fortran. These non-object component bases technologies reinforce the fact that successful reuse is not really about object-oriented languages or class libraries [20].

2.3.3 A Definition of Reuse Types

A definition of reuse types is given in figure 2.2[17]. Some of the terms in the table overlap in meaning. For instance, the terms public and external in the table, both describe the part of a product that was constructed externally, while private and internal describes

when a product is used multiple times within a system it was originally written for. The terms verbatim and black-box both describe reuse without modification, while leveraged and white-box both describe reuse with modifications. The last four terms describe levels of reuse that can occur in the object-oriented paradigm.

Type of reuse	Description	Facet
Ad-hoc	Reuse is conducted by individuals in an informal manner. Components are not designed for reuse. Also called: opportunistic and individual reuse.	Mode
Black-box	The components implementation is hidden from the reuser and cannot be modified.	Intention
External	The use of a component originally written for another software system.	Scope
Generative	A black-box component is created from a application generator	Technique
Glass-box	When components is used as-is like black-boxes (without modification), but their internals can be seen from outside	Intention
Grey-box	If changes to a component are only smaller, for example a few variable renaming or changes in procedure calls.	Intention
Horizontal reuse	Reuse of generic parts in different applications. Also called general-purpose reuse.	Scope
Internal	Multiple use of a component within a software system for which it was originally written.	Scope
Large-scale	Requires a consideration of many nontechnical issues, and cannot be adopted without organizational changes.	Scope
Leveraged	Reuse with modifications	Scope
Small-scale	Reusing small code components like subroutines, functions, modules and classes.	Scope
Source-code	Source code is copied from existing sources. A white-box approach, since the developer is able to modify the code as pleased. Also referred to as code/design- scavenging.	Product
Systematic	Reuse is conducted in a planned, systematic and formal way, as found in software factories.	Mode
Vertical reuse	Reuse within the same application or domain. Also called domain-specific reuse, and this field led to domain engineering.	Scope
White-box	The internals of the component is visible to the developer. The component can be modified and customized in any way appropriate. Source code is available.	Intention

Table 2.2: Definitions of reuse types

2.3.4 Software Repository

One of the most important requirements for achieving software reuse, is to be able to store and retrieve the reusable assets. A software repository is a software library or structure where these assets can be stored and classified, and later found and reused. Before developing new code and components for a specific purpose, the developer searches the repository to check if the particular solution already exists. If the solution exists, the developer can use this as-is or modify it in order to fulfill her specific needs. A repository

should have a wide variety of high-quality components, which must be properly classified in order to retrieve them[50]. Poulin[42] states that the best repositories range from 30 to 250 components, since users often find classification and retrieval methods hard to use. The repository should be managed at organization or enterprise level[5], since it is an activity that spans across different projects and application systems. The largest challenges concerned with software repositories are techniques to locate and retrieve the components, and how to integrate the components into software system[50].

2.3.5 Domain Engineering

In order for an organization to achieve systematic reuse it must have a business strategy that looks beyond the current project[8], and invest in assets that future projects will take advantage of. The requirements of future projects must be anticipated, and resources must be invested in the current project to build software that is intended for reuse. It is also important that the reuse program is marked-driven, and that it profits from the economic dimensions of reuse.

Domain engineering is a key concept in systematic reuse[16], and it is a methodical way of identifying potentially reusable assets and an architecture which enables reuse. A domain may be defined as an application area or, more formally according to Frakes and Isoda[16], as a set of systems that share design decisions. The domain can therefore function as a design space for a family of related systems[20].

Domain engineering consists of two phases; domain analysis and domain implementation. The first phase is the process of discovering and recording the commonalities and variabilities of the systems in a domain[16]. The second phase makes use of the information uncovered in the domain analysis in order to create the reusable assets and new systems. This way, domain engineering helps an organization to look beyond a single project or system. The system architecture is defined for the applications and components, and a set of appropriately generalized components is developed. This should result in reusable assets that can be cost-effectively exploited in subsequent system engineering.

2.4 Nontechnical Aspects of Software Reuse

> "Technical aspects are important prerequisites for successful reuse. However, they do not suffice to make reuse happen" (Johannes Sametinger, 1997, p.37[50]).

Non-technical problems includes changing the organizational structure, processes and culture, as well as the up-front investment to build a software reuse infrastructure[47][50]. The most important obstacles to reuse are economic and cultural, not technological[8]. In an organization, individual developers can do little about software reuse on their own. Management must institute the mechanisms needed, provide organizational support and money to finance them[27], and be committed to reuse[9].

2.4.1 The Human Factor and Cultural Issues

It is now obvious that reuse is not merely a technical issue; it is also a people issue. Morisio et al.[35] have identified three main causes of failure: not introducing reuse-specific pro-

cesses, not modifying non-reuse processes and not considering human factors. An example of human factors is that some software engineers prefer to write their own code or rewrite components because they believe that they can improve on them[67]. This has to do with trust and the fact that writing original software is seen as more challenging than reusing other people's software. This problem is called the "Not-invented here syndrome". This requires a change of attitude from "not invented here" to "we reuse her"[48]. However, Frakes and Fox[15] found that most of the respondents to a survey they conducted did not suffer from the "Not-invented here syndrome", and argued that most developers prefer to reuse rather than build from scratch.

Studies have identified that successful reuse programs must be integrated within the culture of a company's existing organizational structure[13], since cultural changes takes energy and persistence[8]. Card and Comer[8] states that there are four cultural issues have either unique or misunderstood effects on software reuse:

- Training
- Incentives
- Measurement
- Management commitment

Training is often overlooked when reuse programs are planned because people think reuse requires only minor variations on traditional software development techniques. This is not necessary the case since software reuse requires additional planning and design in order to make the developed software applicable in future applications. When Motorola implemented a software reuse program, they discovered that training was the most important activity and it was the key for gaining acceptance for reuse in the organization[21].

The introduction of awards or incentives for developers who contributes to the organizations reuse program has shown to have a considerable positive effect[21][48]. Motorola had good experience with encouraging reuse throughout the organization with a cash-reward for the developers who shared or used reuse-components[21]. Developers should be encouraged to develop software systems with reusable software, and to design components of new software systems for future reuse[48]. We will come back to measurement in section 2.5.2 on page 25. Management commitment is discussed in section 2.4.3.

2.4.2 Economic Issues

Initial investments are needed in order to install a reuse program[50], and separating the investments costs can be difficult. Producing maintainable software is often an integrated part of the development process, whereas making its components reusable is not. Realizing the full return on reuse investment takes time, and organizations must adopt a long-term perspective. This requires real commitment and strategic thinking. Joos argues that high initial costs and a slow return on investment can make software reuse hard to sell[21]. Management must understand and accept the need to invest not just in enabling technology, but also in the projects that produce reusable assets[50].

The extra costs may inhibit the introduction of reuse and can lead to that the overall costs savings are not as great as anticipated[20]. It takes funding to finance[27][20]:

- Education and training

- Vendor supplier components
- Creating or purchasing reuse work products, libraries and tools
- Implementing reuse-related processes
- Domain engineering

The costs and time involved in systematic reuse has to be justified by the organization[17][50]. Software reuse can only succeed if it can be proved as economical beneficial[44]. Potential costs and payoffs can be calculated with a cost benefit analysis model. The payoff will not come for several years later, when numerous applications are built more cost-effectively and more rapidly using the reusable components. This payoff of reduced time to marked and reduced cost builds up over time as the components are reused repeatedly and as more components are developed[20].

2.4.3 Organizational Issues

Organizational factors can have great impact on the implementation of reuse programs[50]. An organizations management structure can determine the failure or success of a reuse program[42], and organizational change will in many cases be required before the full potential of software reuse can be realized[50][6][47].

It is essential that a reuse program receives commitment from the management[34] because reuse programs require changes in the way software is developed. Current methodologies and procedures do not consider reuse as a part of their processes[44]. It is also important that a reuse program obtain resources, power, priority, visibility and support if it is to succeed, just as with any other process improvement project[18].

It is not uncommon for project managers to wanting their development projects to succeed at the expense of other development groups within the same company[50]. Successful projects can enhance project leaders' career opportunities, because it makes them better than their direct competitors. In such a corporate culture, development groups are not encouraged to build generalized software components that may be reused not only in their own future projects, but also in projects of other groups.

Developing for future reuse is expected to take at least twice as much time as when developing non reusable components [48][50]. Even though reusable components are initially more expensive, they cause downstream cost avoidance for subsequent project that can make use of the components. At Motorola[21] they found that a lot of reuse efforts existed throughout the company, and by sharing information, the different groups could benefit from each other experiences, failures and success.

2.5 Reuse Maturity Models and Measurement

> "A maturity model is at the core of planned reuse, helping organizations understand their past, current, and future goals for reuse activities"(Frakes and Terry, 1996, p.424[17]).

The concept of reuse maturity is based on the presumption that an organization wants to become more effective in reusing its software assets[12]. Maturity is intended to be an

indication on how advanced an organization is when it comes to implementing systematic reuse[17]. The assessment of an organizations reuse maturity level should act as motivation factor for taking the appropriate steps for increasing the maturity. Figure 2.1 is gathered from the book "Software engineering with reusable components"[50] on page 51, and shows two examples of reuse maturity. In the case of low reuse maturity, potential and intended opportunities do not match each other well. This means that actual reuse is limited from the start. In the case of high reuse maturity, these opportunities match each other, an facilitates higher actual reuse. Existing reuse opportunities has to be recognized and exploited systematically in order for the efforts to be fruitful.

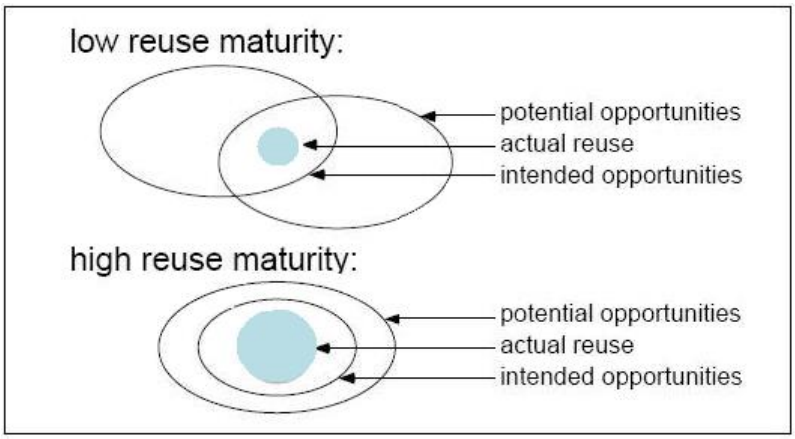

Figure 2.1: Two cases of reuse maturity(Sametinger, 1997, p.51[50])

Reuse maturity models have similarities to the Capability Maturity Model developed by the Software Engineering Institute (SEI). A five-stage maturity model was developed by the Software Productivity Consortium (SPC). This model proposes the following stages: ad-hoc reuse, repeatable reuse, portable reuse, architectural reuse and systematic reuse[50]. Another five-level maturity model was proposed by Koltun and Hudson, which we describe further in section 2.5.1. Other reuse maturity models in literature worth mentioning are:

- REBOOT's Reuse Maturity model[53]
- STARS Reuse Maturity Model[11]
- The Reuse Capability Model[12]

Why are models and measurements so important? Organizations must be able to measure their progress and identify the most effective reuse strategies as they implement systematic reuse programs[17]. This is done with reuse models and metrics. Figure 2.2 is derived from Frakes and Terry and shows metrics and models categorized into six categories. They state that organizations often encounter the need for these metrics and models in the order presented.

- Reuse cost-benefits models include economic cost/benefit analysis as well as quality and productivity payoff
- Maturity assessment models categorize reuse programs by how advanced they are in implementing systematic reuse
- Amount of reuse models are used to assess and monitor a reuse improvement effort by tracking percentages of reuse for life cycle objects

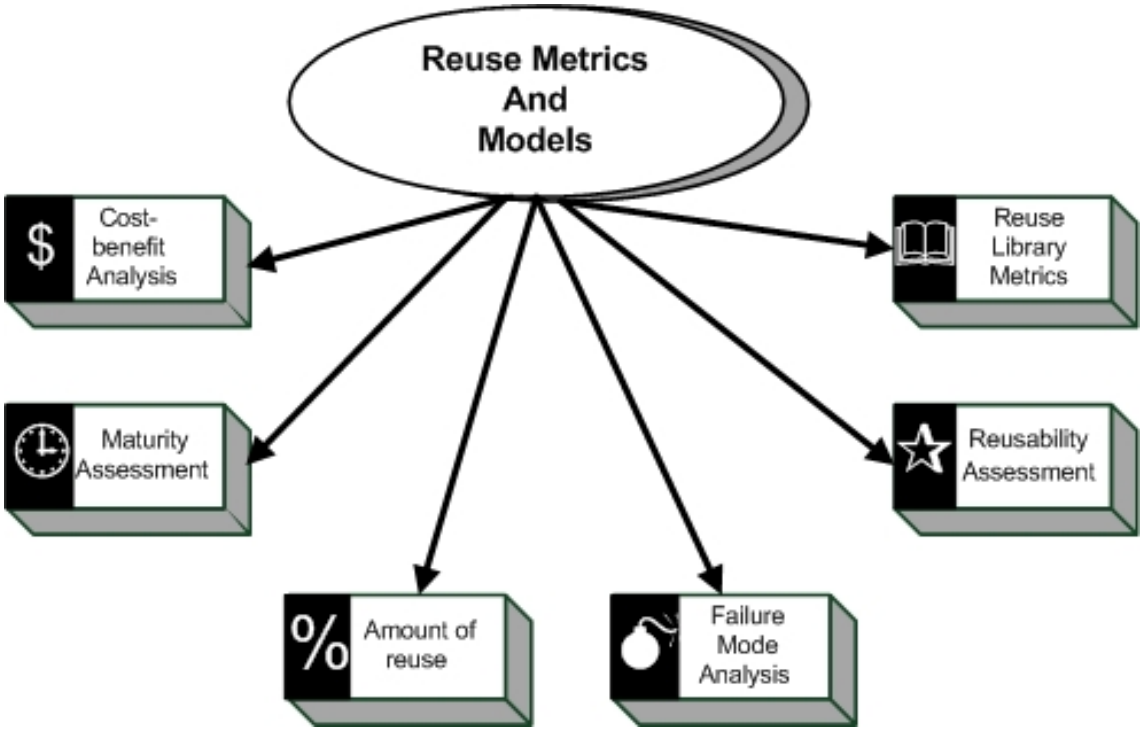

Figure 2.2: Categorization of reuse metrics and models (Frakes and Terry, 1996, p.416[17])

- Failure models are used to identify and order the impediments of reuse in a given organization
- Reusability metrics indicate the likelihood that an artifact is reusable
- Reuse library metrics are used to manage and track usage of a reuse repository

2.5.1 Koltun and Hudson's Reuse Maturity Model (RMM)

The Reuse Maturity Model was developed by Philip Koltun and Anita Hudson during a year-long system improvement project at Harrison Corporation[23]. The model is shown in table 2.3.

The columns in the model represent the five phases of reuse maturity, assumed to improve along an ordinal scale from 1 to 5 [23][17]. Ten dimensions or aspects of reuse maturity have also been enumerated. For each of these dimensions, an attribute or situation has been specified for each maturity level. In this model an organization has to assess its reuse maturity before starting with a reuse improvement program by identifying its placement on each dimension. The amount of organizational involvement and commitment for each of the ten dimensions of reuse expands as an organization advance from initial/chaotic reuse to ingrained reuse. With such characteristic descriptions, organizations should be able to find their place in the maturity model with a minimum of efforts according to Koltun and Hudson[23], although Sametinger[50] states that it is difficult to quantify reuse maturity levels.

These are the following five phases in the model:

1. Initial /chaotic

2. Monitored
3. Coordinated
4. Planned
5. Ingrained

At the start of a reuse program, most organizations are between the Initial/Chaotic and Monitored phases[17]. When an organization achieves Ingrained reuse, reuse becomes part of the business routine and will no longer be recognized as a distinct discipline. Ingrained reuse embodies fully automated support tools and accurate reuse measurement to track progress. Software reuse concerns must be institutionalized in the development life cycle if it is to make lasting progress[22].

	1 Initial/ Chaotic	2 Monitored	3 Coordinated	4 Planned	5 Ingrained
Motivation/ Culture	Reuse discouraged	Reuse encouraged	Reuse incentivized re-enforced rewarded	Reuse indoctrinated	Reuse is the way we do business
Planning for reuse	None	Grassroots activity	Target of opportunity	Business imperative	Part of strategic plan
Breadth of reuse	Individual	Work group	Department	Division	Enterprise wide
Responsible for making reuse happen	Individual initiative	Shared initiative	Dedicated individual	Dedicated group	Corporate group with division liaisons
Process by which reuse is leveraged	Reuse process chaotic; unclear how reuse comes in	Reuse questions raised at design reviews (after the fact)	Design emphasis placed on off the shelf parts	Focus on developing families of products	All software products are genericized for future reuse
Reuse assets	Salvage yard (no apparent structure to collection)	Catalog identifies language and platform specific parts	Catalog organized along application specific lines	Catalog includes generic data processing functions	Planned activity to acquire or develop missing pieces in catalog
Classification activity	Informal, individualized	Multiple independent schemes for classifying parts	Single scheme catalog published periodically	Some domain analysis done to determine categories	Formal, complete, consistent timely classification
Technology support	Personal tools, if any	Many tools, but not specialized for reuse	Classification aids and synthesis aids	Electronic library separate from development environment	Automated support integrated with development environment
Metrics	No metrics on reuse level, pay-off, or costs	Number of lines of code used in cost models	Manual tracking of reuse occurrences of catalog parts	Analysis done to identify expected payoffs from developing reusable parts	All system utilities, software tools and accounting mechanisms instrumented to track reuse
Legal, contractual, accounting considerations	Inhibitor to getting started	Internal accounting scheme for sharing costs and allocating benefits	Data rights and compensation issues resolved with customer	Royalty scheme for all suppliers and customers	Software treated as key capital assets

Table 2.3: Hudson and Koltun Reuse Maturity Model(Frakes and Terry, 1996, p.425[17])

2.5.2 Measurement

> "It is often said that we cannot manage what we cannot measure" (Johannes Sametinger, 1997, p.48[50]).

Measurement is the activity of measuring a given property of software[32]. A metric is the property of the measured software and it is a mean to measure the software product and the process by which it was developed. This makes metrics important in effective software management[41]. Software reuse in organizations can span across multiple projects and can have influence on an organizations processes and structures. This is why some kind of monitoring is required if one is to manage such enterprise-wide activities. Software metrics can be used for estimating costs, costs savings and evaluating a particular software practice[41].

Incorporating reuse measures into an organizations measurement program can be a strong incentive to reuse within the organization[8]. If the management notices that they can gain returns of investment for their investments in reuse, it is a greater chance that they will make a commitment and grant resources. This is why reuse practitioners have learned the mantra: "business decisions drive reuse!"[42].

Reuse Level

The amount of software reuse (referred to as both reuse level and reuse ration[28]) in a certain software system can be determined by the ratio of reused components (or their lines of code) to the total components of the system (or total amount of code lines)[50][17]. It can also be used to determine how similar two files are. The number of lines that are identical in file a and file b as opposed to the total number of lines (of file a) gives an indication of how much of file a has been reused in file b. The formula can be used for determining how much of file a is being reused in file b, and how much of file b originates from file a (by using the total number of lines of file b)[50].

$$\text{Line reuse percentage} = \frac{NumberOfIdenticalLines}{TotalNumberOfLines}$$

The comparison of lines and words can give a good indication about white-box reuse[50], while the comparison of components gives indication of black-box reuse. This approach is used extensively by both the industry and academia, but this metric should never be used alone when measuring results of software reuse; a high reuse ratio does not necessary denote that the time and effort expended to achieve the level is justified[28][50]. Additional lines may be required in order to incorporate the reused source code or component. Another disadvantage with reuse ratio alone is that it does not consider other important aspects in software reuse maturity, such as software repository and organizational aspects.

2.6 How to achieve Systematic Software Reuse

Systematic reuse does not just happen. It must be planned and introduces throughout an organization-wide reuse program[67]. Software reuse requires a concerted and systematic effort by both management and software developers in order to overcome the business, process, organizational and technical impediments that often hinder software reuse [20]. Without an explicit reuse agenda and a systematic approach to design and process, it is difficult to obtain the desired benefits and reuse level[3]. A proper reuse program reduces the initial risk for development, maintenance, and acquisitioning activities by using software that is already proven to be functional and reliable through prior usage[48].

2.6.1 Reuse Program

An organization without a systematic and mature development method can experience difficulties in taking advantage of reuse[8]. A high level of maturity can help the organization to understand and control their processes[34]. In section 2.2.2 we defined a reuse program as an organizational structure and collection of support tools that is aimed at fostering, managing, and maintaining the practice of reusing software in an organization. A reuse program should be part of an organizations overall process improvement program, not a standalone activity[34], in order to make a lasting progress[22]. A reuse program can give a organization the power to deliver more function, to deliver it faster, to deliver it with fewer defects and to deliver it with less cost[41]. As mentioned in section 2.4.3, systematic reuse requires changes to the development methods and organizational structure.

Morisio et al.[34] suggests that an organization follow these six key points for success:

- Change processes and roles
- Obtain management commitment
- Minimize changes
- Keep a sense of portion
- Anticipate human factors
- Acquire process maturity

Defining roles and adding new reuse processes will help clarify who is in charge of developing a company's reusable assets and when[34]. Prieto-Dìaz and Joos[44][21] both recommend that reuse programs should be implemented incremental. This helps the organization to provide immediate return on investments and build up confidence in the organization. An incremental approach also helps the organization to manage the program, tune and refine the reuse processes, and facilitate monitoring and evaluation of the program. Prieto-Dìaz also argues that the reuse program should be systematic, formal and have management support. As mentioned in section 2.4.1, management support and commitment are key ingredients in all successful reuse programs. Managers can be seen as the reuse programs sponsors; representatives from the top management who has power over resources[18]. By minimizing changes, Morisio et al.[34] recommend that one should introduce as few changes as possible at a time, and build on existing knowledge, skills, and tools in the organization. Human factors can be considered by providing education and training. This should not be underestimated since it has shown to be highly important for gaining acceptance and understanding of reuse in the organization. Joos[21] highlights the importance of finding a champion for each of the roles in the reuse organization. Champions are the driving forces behind the change processes, and it is important that these people have a strong software engineering background, are enthusiastic about reuse and have trust from their fellow workers[18].

The combination of guidelines, organizational structures, processes, methods, reuse metrics, economic aspect, technologies and a set of requirement is needed for a reuse program to be effective[3][44]. The guidelines should include how to plan, specify, model, design, implement and document components and applications in the problem domain.

2.6.2 Reuse requires Changes in Development Process

Software engineering should apply defined, repeatable processes to achieve predictable results[34]. However, since processes vary from organization to organization, and from project to project, one approach clearly cannot fit all. Morisio et al.[34] initiated a two-year study in 1997 of about two dozen European companies that were establishing reuse programs. They discovered that despite variation in business context, technical and managerial tradition, and size, companies can indeed achieve reuse, using diverse processes. Alvaro et al.[3] argues that existing methods are not flexible enough to meet the needs of various situations, or too vague, and not applicable without strong additional interpretation and support. There is therefore a need for better and more flexible methods[9] that can be customized to adapt various enterprise situations, and methods with sufficient guidance and support[34]. An organization implementing a reuse program is therefore advised to adapt processes that are suitable for doing so.

According to Jacobson et al.[20], systematic software reuse can be expressed in four concurrent processes, as seen in figure 2.3: create, reuse, support and manage processes.

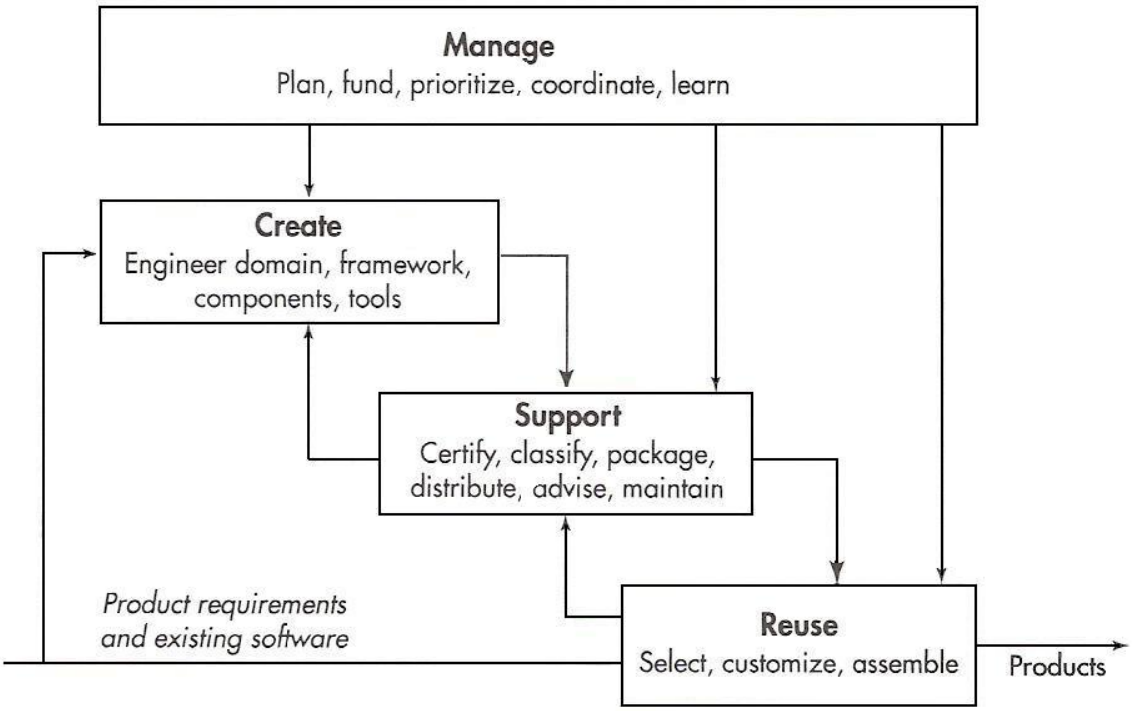

Figure 2.3: Systematic reuse involves four concurrent processes (Jacobson et al., 1997, p.16[20])

Create

The create process, as seen in figure 2.3, identifies and offers reusable assets to the reusers[20]. The process can include activities such as inventory and analysis of existing application and assets, domain analysis, architecture definition, assessment of reuser needs, reusable assets testing and packaging. The assets can include code, interfaces, architecture, test, tools an so on.

Reuse

The reuse process make use of the reusable assets to produce applications or products[20]. The process can include activities such as examination of domain models and reusable assets, the collection and analysis of end-user needs, the design and implementation of additional components, adaption of provided assets and the construction and testing of complete applications.

Support

The support process helps the overall set of processes, and manages and maintains the reusable assets collection [20]. The process may include activities such as classification and indexing the reusable assets in a library, announcing the distribution of the asset, providing additional documentation, collection feedback and defect reports from the reusers.

Manage

The manage process plans, initiates, resources, tracks and coordinates the other processes[20]. The process can include activities such as setting priorities and schedules for new asset construction, analyzing the impact and resolving conflicts concerning alternative routes when a needed asset is not available, establishing training, and setting direction.

2.6.3 Reuse requires Changes in Organization

An organizations management structure can often determine the failure or success of a reuse program[41]. A systematic reuse process involves two primary functions; domain engineering organization and application engineering organization[20]. Domain engineering aims at defining and implementing domain commonalities in a generic product as discussed in section 2.3.5 on page 20, and application engineering produces individual applications for customers starting from the generic product, according to Morisio et al.[34]. The first function involves the creator (also called producer[17] of the reusable components), and the latter involves the reuser(also called consumer of the reusable components). Companies with experience in systematic reuse generally find that a third function is of importance, namely support as seen in figure 2.4 by Jacobsen et al.[20].

Reuse activities can be divided into producer activities and consumer activities[17], each with distinct goals. Creators need to build high-quality assets that will serve the needs of many reusers[20]. The creators have to be close to the reusers to keep reusable components practical. At the same time they must be isolated from daily project pressure if they are to get reusable components designed and build. The "M" in the box, in figure 2.4 represents a local manager who has to mediate the interests of both creators and reusers, because of the different goals.

2.6.4 Pilot Projects

Several reuse strategies suggest that an organization should start the reuse program with smaller pilot projects[51][21]. As these pilot projects meet a degree of success, they are

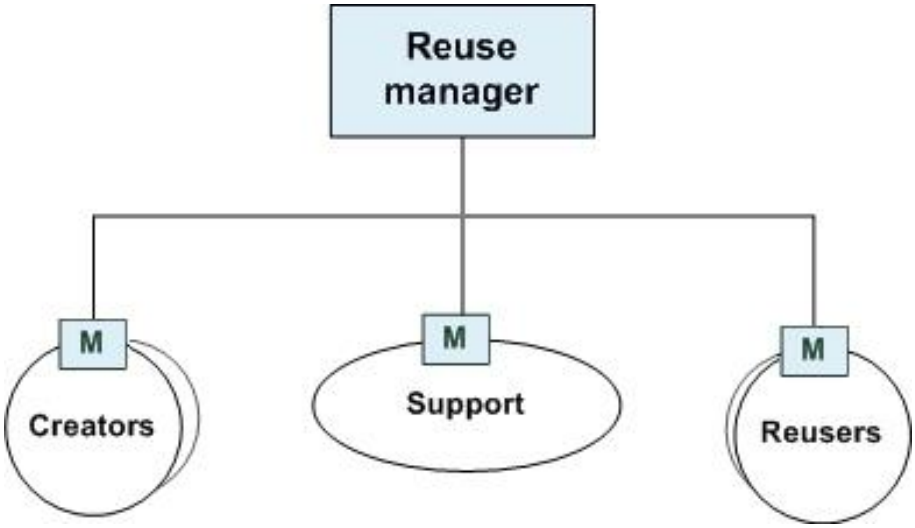

Figure 2.4: A standard reuse organization(Jacobson et al., 1997, p.20[20])

expanded incrementally which increases the reuse coverage and penetration into the organization. An incremental approach is therefore recommended, i.e., starting with small teams and representative subsystems to gain experience, skills and confidence[9][51]. Observation of the introduction of business process engineering and indeed, change management in general further reinforces this stepwise approach.

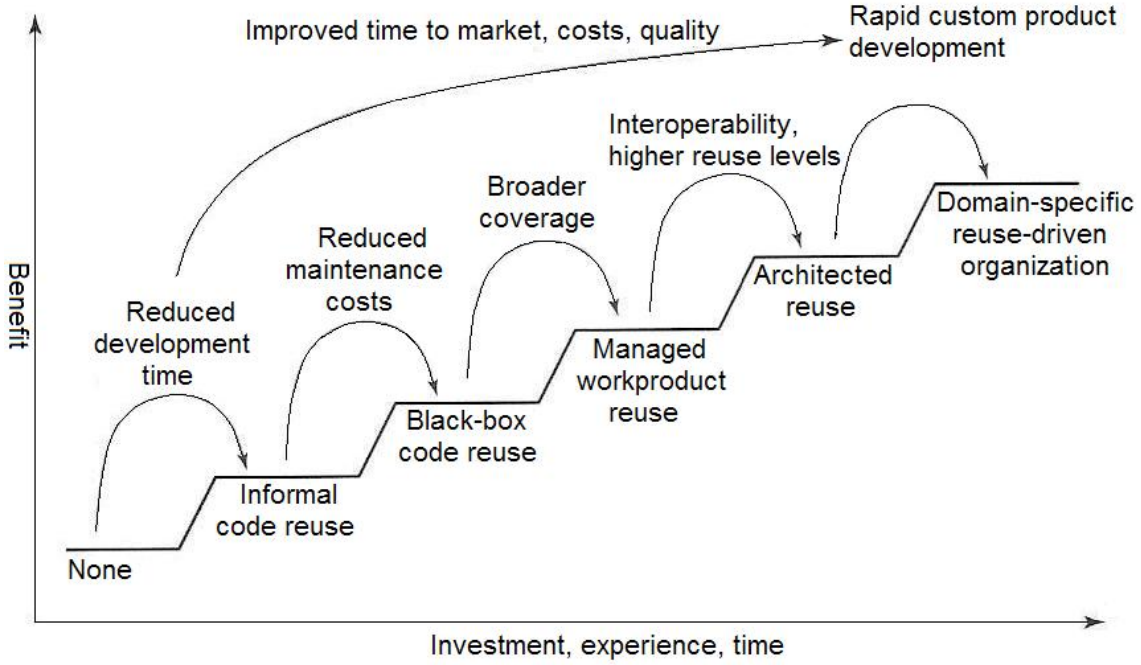

Figure 2.5: The incremental adoption of reuse (Jacobson et al., 1997, p.21[20])

2.6.5 An Example of an Incremental Transition from no Reuse

The transition from no reuse to informal reuse as shown in figure 2.5 by Jackobsen et al.[20] occurs when developers are familiar with each other's code, and trust each other. They also have to feel the need to reduce time to marked, even though they prefer to write their own code. This strategy works fine for a while. Development time is reduced and testing is often simpler than with brand new code. But as more products are developed using this approach, maintenance problems increases. Multiple copies of the software, each slightly different, has to be managed. Defects found in one copy must be found and fixed multiple times.

This often leads to a black-box code reuse strategy, in which a carefully chosen instance of code is reengineered, tested and documented for reuse[20]. All projects are then encouraged or required to use just this copy without modifications. This works well for a while too, until the issues of dealing with changes to satisfy an increasing number of reusers arise. Such issues can lead to the creation of a managed work product reuse process, in which the creation and reuse of components is explicitly managed and supported by a distinct organization.

An organization must move incrementally. Start by focusing on a narrow domain and only a part of the complete product architecture. This is usually best done via a series of reuse pilots[20]. As the reusable software components and frameworks is being developed, the organization must gain and retain management support, customize process and organization, and ensure that the reusers are involved in the discussions early.

Recommendation for Implementing a Reuse Program

Jaconsen et. al recommends the following ten principles for software reuse(Jacobson et al., 1997, p.27[20]):

1. Maintain top-management leadership and financial backing over the long term
2. Plan and adopt the system architecture, the development process and the organization to the necessities of reuse in a systematic but incremental fashion. Start with small pilot projects, and then scale up
3. Plan for reuse beginning with the architecture and an incremental architecting process
4. Move to an explicitly managed reuse organization which separates the creation of reusable components from their reuse in applications, and provides an explicit support function
5. Create and involve reusable components in a real world environment
6. Manage application systems and reusable components as a product portfolio of financial value, focusing reuse on common components in high-payoff application and subsystem domains
7. Realize that object or component technology alone is not sufficient
8. Directly address organization culture and change, using champions and change agents
9. Invest in and continuously improve infrastructure, reuse education and skills

10. Measure reuse progress with metrics, and optimize the reuse program

2.7 Previous studies of Developers Attitude towards Software Reuse at NTNU

26 developers from three Norwegian companies participated in a survey about their experience and attitude towards component reuse and component-based development to investigate the relationship between the companies reuse levels and some key factors in reusing in-house components[38]. The study was published in 2004.

Another study from Statoil ASA[49], published in 2006, characterized developers view on software reuse. The researchers used a survey followed by semi-structured interviews which investigated software reuse in relation to requirements (re)negotiation, value of component information repository information, component understanding and quality attribute specification.

2.7.1 Ericsson, EDB Business Consulting and Mogul

An empirical study was performed as part of two Norwegian R&D projects; SPIKE (Software Process Improvement based on Knowledge and Experience) and INCO (Incremental and Component-based development)[38]. The sample size of the current research is still small and is therefore still a prestudy. Data collection was carried out by NTNU PhD and MSc students. Mohagheghi, Naalsund, and Walseth performed the first survey in Ericsson in 2002. In 2003, Li, Sæhle and Wang performed the survey reusing the core parts of the questionnaire in two other companies (i.e. EDB Business Consulting and Mogul Technology)[38]. The companies were selected because they had experience on component reuse and wanted to contribute to NTNU's research.

Ericsson Norway-Grimstad started a development project in the late 90s and has successfully developed two large-scale telecommunication systems based on the same architecture and many reusable components in cooperation with other Ericsson organization. EDB Business Consulting in Trondheim (now Fundator) is an IT-consultant firm which helps its customers to utilize new technology who started to build reusable components in 2001. Mogul Technology (now Kantega) in Trondheim has large customers in the Norwegian finance- and bank sector[38].

The surveys found that developers are positive, but not strongly positive to the value of component repository. None of the developers believe that the design/code of components is well documented because the documents are either incomplete or not updated. They believe that insufficient component documentation is a problem, but they manage to get an understanding of the components from informal channels, such as previous experience and local experts.

2.7.2 Statoil ASA

Data was collected by two NTNU PhD students. 16 developers participated in the survey and filled in the questionnaire based on their experience and views on software reuse. The results from the survey showed that reuse benefits from developers view include lower

costs, shorter development time, higher product quality of the reusable components and a standardized architecture. These results support those found in the literature[27]. In terms of factors contributing towards reuse, the survey found no link to education or evidence that experience contributes towards reuse. When it comes to formal processes, the findings support the literature[15]; the formal processes are used only for software development in general, not specifically for software reuse, but they may still have an implicit positive effect. Improving documentation of reusable components would have been largely beneficial toward achieving successful reuse.

2.8 Empirical Research Methods

It is of critical importance that the research is carried out in such a way that the knowledge we establish can be used and re-examined by other researchers[4]. By using a method, one follow a determined path toward a goal. Different methods tells one how to collect, analyze and interpret data. The three most important characteristics for empirical research is that it is systematic, thorough and open. By knowing the different methods, it helps one to make the appropriate choices and increases the chances for reliable results[19].

Quantitative and qualitative methods are used for performing empirical studies, where each of these are used in different situations depending on the desired result[4]. The difference between these two approaches is that qualitative methods collect information as text, pictures or sound, while quantitative methods collect information as numbers.

2.8.1 Qualitative Methods

Collection and analyzing of qualitative data is more flexible than with quantitative approaches. Qualitative data can consist of text, pictures or sound that are processed to get a deeper meaning[4]. We chose to give a short description of the qualitative method "interview" because we have used it in our research.

Interview

A interview is a conversation with a structure and a purpose. The interviewers asks all the questions and follow up the answers that the informant gives. The epistemological starting point in using qualitative interviews is that the researcher must talk, interact, listen and ask questions in order to get peoples knowledge, views, understandings, interpretations, experiences and interactions[4].

An interview usually has a structure, and can consist of both open and closed questions. The interview can have various degrees of being structured, ranging from a conversation without strict structure or order, to a fully structured interview with predefined questions and alternatives for the answers. Advantages with standardized interviews with alternatives for the questions, is that it makes it easier to focus the interview and that it is easier to analyze and compare results. One disadvantage with interviews with strict structures is limited flexibility. Expert interview can also be performed, where people with specific knowledge about a topic is interviewed or group interview.

2.8.2 Quantitative Methods

The collection and analysis of quantitative data is characterized by lesser flexibility than with qualitative approaches. The analysis of data is done with enumeration and various statistical estimates[4]. A description of surveys is given since we have made use of it in our research.

Survey

A questionnaire usually has predefined answer options (pre-coded forms) and is standardized. This let us look at similarities and variations in how respondents answer[4]. One can, among other things, look into diffusion of a phenomenon, correlations between different phenomenons, and generalize results from sample to population. Another advantage is that standardized questionnaires lets one collect data from many respondents in a relatively short period of time.

The starting point for the formulation of a questionnaire is the research questions. It is important that the questions are formulated in such a way that they give adequate results on the research questions. One benefit with using questions from other questionnaires is that the results can be compared with other surveys[4]. It is important that questions' and answers' options are formulated in the exactly same way as the original questionnaire, in order to compare the results.

Open questions were the respondents can write down the answer with his own words is especially useful in order to get additional information beyond the predefined answer options. But such questions are hard to generalize. If many open questions are required, it may be better to use interviews instead. Few responses to a questionnaire can cause imbalance in the selection, because many of the preferred respondents are missing. The respond rate to questionnaires are often low.

2.8.3 Evaluation of Research Activities

The reliability term and the various forms for validity are used for assessing quantitative research. Guba and Lincoln[4] thinks that qualitative research has to be evaluated differently than quantitative research. They use the terms reliability, credibility, transferability and conformability as measurement in qualitative research situations. Other researchers states that it is not all or nothing, but both. Cook and Campbell[10] states that there are four threats to validity: conclusion, internal, construct and external validity. Table 2.6 shows the relationship between qualitative and quantitative situations, and is translated from Norwegian from the book "Forskningsmetode for økonomisk administrative fag" on page 227[4].

Reliability

Reliability has to do with the surveys data; what type of data is used and how it is collected and processed. Reliability is critical in quantitative research. In qualitative research it is not always practical to test data's reliability in the same manner as with quantitative research. This has to do with unstructured data collections as observations (which is

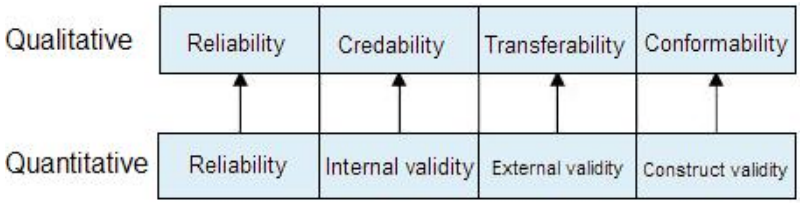

Figure 2.6: Different approaches to quality assessments of qualitative and quantitative research(Johannessen et. al, 2004, p.227 [4])

context dependent) and interviews. It is almost impossible for a researcher to duplicate another researcher's qualitative research. A researchers experience has great influence on how data is interpreted. In order to strengthen data's reliability, the researcher should give a detailed description of the context and an open and thorough presentation of the method that is used under the whole research process[4].

Credibility

Credibility in qualitative research deals with to what degree the researcher's findings on an accurate way reflects the purpose with the study and how it represents reality[4]. Continual observation and triangulation is two commonly used techniques for increasing the researcher's prospects for credible results. Credibility can also increase by letting colleagues analyze the same data material to see if they interpret the results the same, or by letting the informants verify the results. For quantitative research the term internal validity is used.

Transferability

A survey's transferability has to do with whether the results of the research is transferable to other similar situations. For quantitative research the term external validity is used.

Conformability

Conformability has to do with to what degree our empirical data really measures the theoretical terms and variables we intended to measure. Conformability is thus important for how meaningful, interpretable and generalizable the research results really are[4]. The findings have to be a result of the research, and not a result of our own subjective perception. It is therefore important to be self critical to how the research is carried out, and comment previous experience, imbalance or deviation, prejudice and other perceptions that could influence the interpretation and approach to the research. So even though one cannot achieve conformability in qualitative surveys, it is still important that the researcher's data material can be traced back to its origin (without revealing the informants identity). For quantitative research the term construct validity is used. An example of a relevant threat to the construct is that the subjects have different perceptions of the scale used in the answer alternatives. A person answering "very high" may be equal to another person's perception of "high".

2.9 Summary

We have seen that software reuse is the process of creating new software from existing systems, knowledge or artifacts rather than building them from scratch[50][15][17]. Software reuse can be found in many different forms, ranging from ad-hoc to systematic, from white-box to black-box, and from horizontal to vertical[50][47][14]. Improved software quality, increased productivity, better interoperability between software systems and reduced costs are important benefits associated with software reuse[72][67][50]. Software reuse has however, proved to be difficult to achieve, and far from all organizations succeed with systematic reuse[16]. We see that non-technical issues such as organization structure, human factors, economical issues, together with technical issues such as creating and maintaining a component library are important when dealing with software reuse. Systematic reuse does not just happen. It must be planned and introduces throughout an organization-wide reuse program[67].

Chapter 3

State-of-the-practice at Skattedirektoratet

Skattedirektoratet (SKD) is the Norwegian directorate of taxes. Its overall goal is to make sure that obliged taxes and duties are correctly settled and paid[55]. SKD is also responsible for the National Registry of the Norwegian population. The organization chart for SKD is shown in figure 3.1.

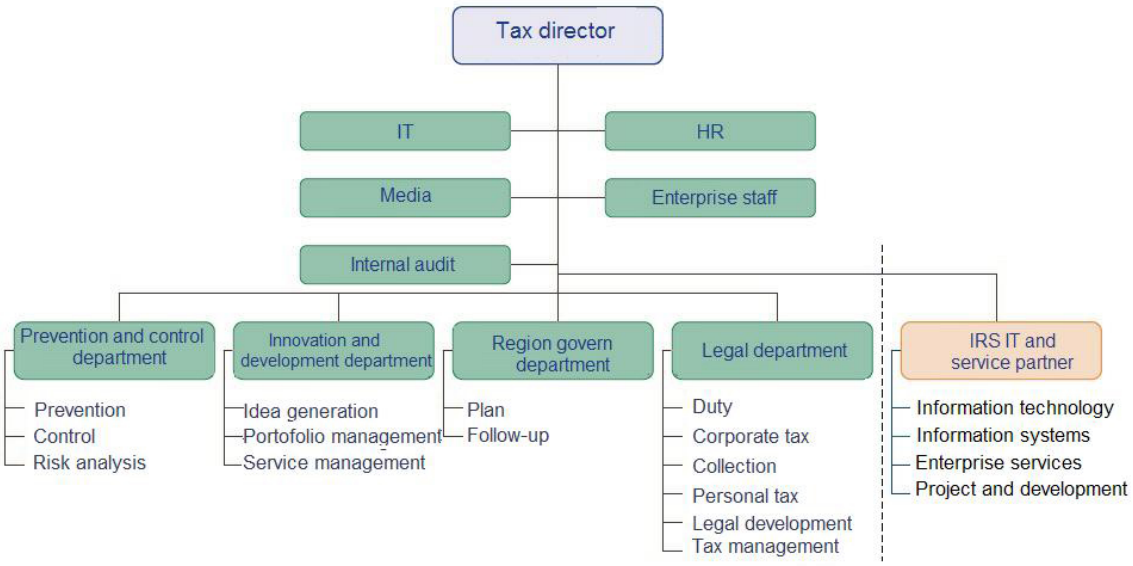

Figure 3.1: Organization chart (01.01.2008)

3.1 IT-Department

The computer systems of SKD are one of Norway's largest when it comes to amount of data and users. The IT-department is responsible for developing, managing and operating these systems. The department has about 220 employees and is organized as shown in

figure 3.2. The *Information systems* branch in the figure is responsible for developing new concepts and systems, and perform maintenance on existing systems.

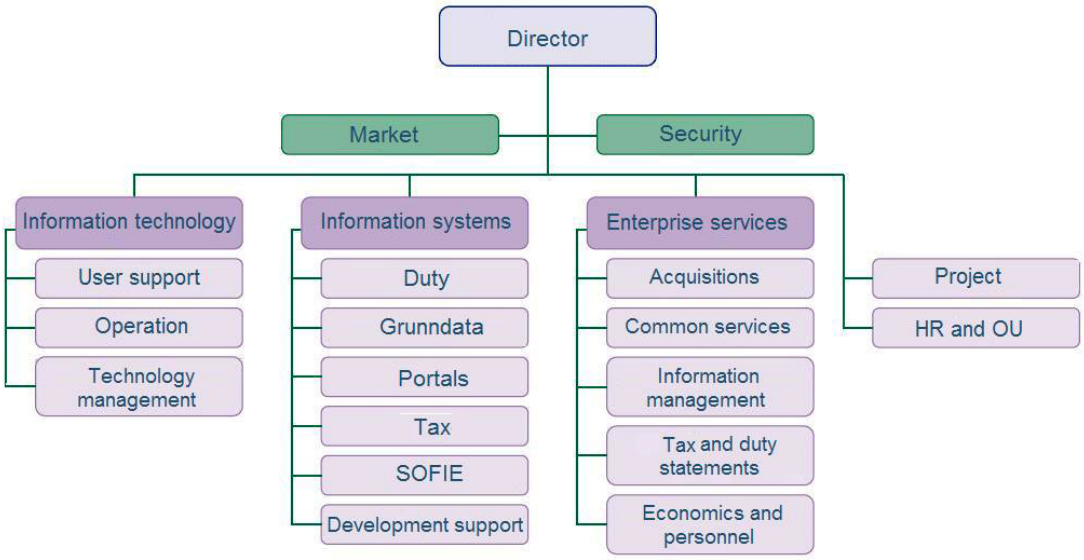

Figure 3.2: IT-department (01.01.2008)

3.1.1 System Group 2 (SG2)

At the time when our research was initiated, the group responsible for maintenance and further development of the GLD systems (see section 3.5) was named System group 2, from now on referred to as SG2. They were also responsible for the technical expertise regarding GLD and performed work related to the system. Our supervisor and contactperson within SKD, Tore Hovland, was the group leader for SG2. As of January 2008 the organizational structure of SKD was reorganized. Changes were made to both SG2 and other system groups, but we chose not to emphasize on this since it did not affect our work. SG2 would have belonged under the *Information systems* branch in figure 3.2. Although SG2 has ceased to exist, we will still refer to the group responsible of the GLD systems as SG2.

The largest and most important tasks for SG2 involves[56]:

- Program and print out new versions of forms and accompanying letters
- Develop new versions of the GLD systems each year
- Correct defects in the GLD systems
- Produce weekly statistics over received, selected and approved tax related information (basis data)
- Responsible for distributing tax related information

3.2 IT Systems

Most of the systems that SKD develops and administer are built in order to carry out and direct tax regulations [58]. Each tax year, the Norwegian parliament, Stortinget, proposes and legislates new laws and regulations. Sporadic resolutions and changes in regulations with a fixed start date can come throughout the year, which has high priority and must be implemented independent of scope, cost and any other planned or started changes.

SKD's systems are divided into two main groups of systems: *annual systems* and *continual systems*[58]. Changes to the latter can in much larger extend be planned and accomplished independent of regulated deadlines. On the basis of earlier experience, both functional and technical changes may be proposed and planned thorough, either annually or as intermediate cycles during the year.

The IT-systems are built on various technical platforms, and have high demands regarding performance, quality and functionality. The systems are based on Oracle and/or DB2 databases, and uses UNIX, PC and IBM mainframe as platforms, as seen in figure 3.3. Note that the figure shows the system groups responsibilities before the reorganization of SKD.

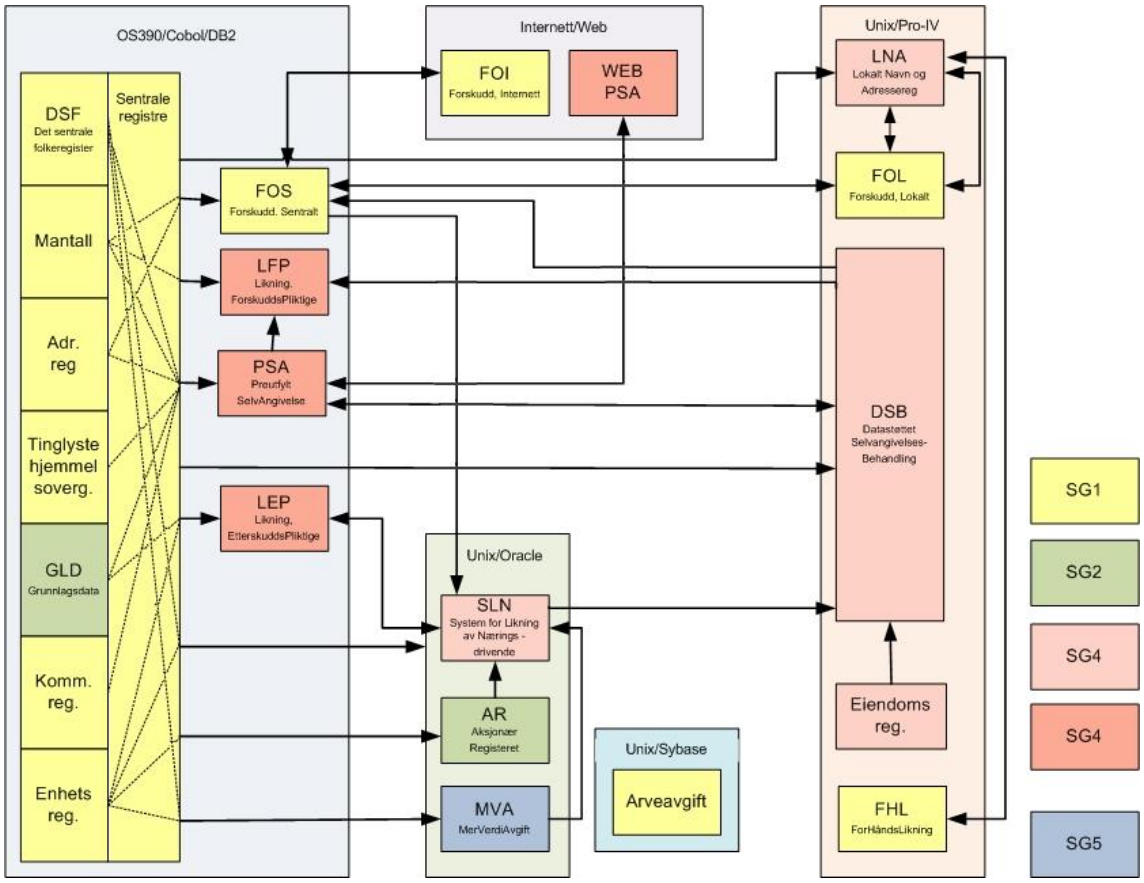

Figure 3.3: System chart for SKD, version 1.3

3.3 Electronic Cooperation in the Public Sector

The government has launched a modernization program which intends to make life easier for its citizens and industry by building down bureaucracy and transferring resources to service production[39]. The modernization program proposes strategies and initiatives which goals is to make public sector more efficient and utilize public resources more efficiently. In order to achieve these goals, the government's IT-systems must be able to interact with each other, and other IT-systems in the industry[39], and they must provide both industry and public with several and better services[24]. Electronic interaction can contribute to:

- Improved and more related services for citizen and industry
- Improved utilization of common basis data (key information for identifying persons, business, properties etc.) for citizen and industry
- Improved possibilities for establishing services across sectors and administration levels, and on new areas
- Establish new solutions in public sector, based on self-service solutions
- Greater competition and diversity in the supplier marked

The government possesses different registers, and many of these do not provide sufficient solutions. In the public sector there are for example many local variations of the National Registry that both collects and administers data, instead of using the *central national registry*. Different registers are developed over a long time period and administered by different organizations. The evolution has lead to that information which was only meant to be used by an individual organization, now wants to be used by several other organizations, and for additional purposes. The National Registry, the Employer/Employee Register, and the Property Register are examples of registers that are used by several, both in and outside of the public sector.

In order to achieve the goals of the government's modernization program, a common electronic infrastructure that public authorities, industry and citizen can connect to is required[24]. The work with an architecture for electronic cooperation is expected to result in better and more cost effective cooperation within the public sector, private users and industry[39].

A proposed, new IT-strategy[66] for SKD suggest the use of open standards and open source. This way, the work will ensure competition between suppliers in the IT-marked[66]. There is also a need for closer integration between the electronic services in use by the general public and the underlying branch systems[39]. The systems and registers in public sectors are developed over a long time period and often based on different technology, and it can be challenging to make these different technologies work together. SKD's IT-strategy suggests that middleware and common exchange solutions should be used in order to use the existing legacy systems. This way one can avoid making too much changes to well-established systems. When new software systems are requested, existing solutions should be reused if possible. New components should be designed and implemented generically, and with focus on software reuse. Software systems developed by governmental agencies, such as SKD, are expected to have a rather long life cycle. One must therefore expect considerable costs concerned with the maintenance of these systems, and the systems should be created cost-effective in a long-term perspective. It is also a request that the public

sector work together on the use of common software components and services, and that a common software library for communication within the public sector is established[24].

3.4 Framework for System Maintenance

SKD has its own framework for system maintenance. The system maintenance process describes phases, activities, guidelines, supporting material, and how the maintenance organization is organized. This process makes SKD capable of receiving, managing and effectuating requirements for both new and existing systems[57]. The process is controlled by a system maintenance organization, which consists of several units within SKD. Each unit is participating in, or managing one or more activities. Communication and collaboration is important in order for the process to be completed within a given time, and with the expected result and quality.

Each phase consist of several activities, which different units can be responsible for. The framework describes what to ground the activity on, and which support materials and roles are recommended to get the expected results. Support material ranges from checklists, routines, technical descriptions, templates, guidelines and standards. Under routine, we find routines for how to collect experiences, evaluate the system maintenance process, configuration management, development etc.

The system maintenance process consists of six generic phases, as seen in figure 3.4; mapping, analysis, design, development, test and production.

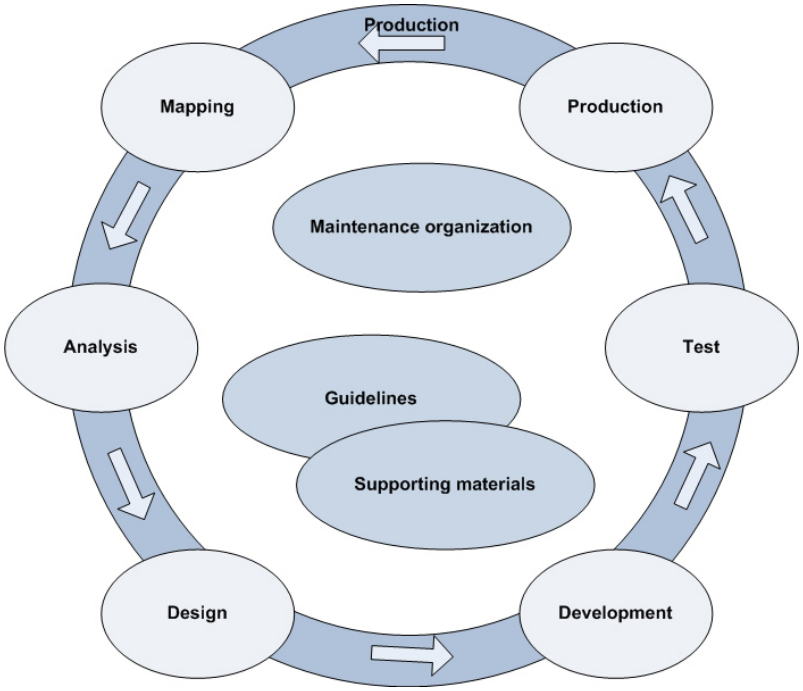

Figure 3.4: SKD's Framework for System Maintenance

Based on last years version of the system and changes in laws and regulations, new requirements and proposals for amendments must be considered when creating a new annual version. Annually version will be discussed further in section 3.5.4. All of these requirements and amendments are collected and registered in the mapping phase. The artifact

of this phase is a report which describes the experience gained from the execution of the phase. All the requested changes found in the mapping phase is then specified more thoroughly in the analysis phase. This phase also produces proposals for how to solve the requested changes, and then estimates them by using cost/benefit analysis. Based on these estimations, decisions are made regarding which changes should be implemented.

The foundation for the development is performed in the design phase. This involves the creation of a detailed and complete design for each of the changes, and planning of unit tests. All implementation and unit testing are performed in the development phase. When a decision is made, the artifacts from earlier phases are updated, so that the actual implementation is a true reflection of the proposed solution. In the test phase, all new and modified code is integrated into the existing system, and thoroughly tested. The Framework for System Maintenance includes a strategy for testing which describes how the testing is performed, and how to verify that the system is robust and fulfills its requirements. When all tests has been approved, the system is ready for production. In this phase, the system is distributed into the operating environment, and made accessible for all users.

3.5 The GLD System

The GLD systems are legacy systems, which dates back to the late 1980's and early 1990's, running on SKD's mainframe. The systems run in batch on SKD's mainframe, and are mainly made up from COBOL-programs, in addition to some assembler and REXX routines[59]. The system uses IBM's relation database DB2. COBOL developed programs over the database and CICS applications on top. The CICS applications are used for entries, control and quality assurance of the admitted data in the systems.

In Norwegian "GLD" is short for *basis data (GrunnLagsData)*. The main purpose of the systems are to receive and store tax related information from third-parties. This information is then transferred to the DSB system, and is also used for the prefilling of tax returns. DSB is a system for processing tax returns, and is used at the local tax assessment offices.

3.5.1 List of GLD Systems

There are 15 different GLD systems as listed below[64]. The original names for these systems has been translated from Norwegian, and can be found in appendix B.

- **GA/LTO** - Salary and Deduction
- **GB** - Credit and Interest
- **GC** - Shareholding
- **GD** - Building Association
- **GE** - Gifts for research and volunteer organizations
- **GF** - Life Insurance
- **GG** - Casualty Insurance
- **GH** - Holding of Share- and Bond Funds

- **GJ** - Vehicle and agriculture
- **GK** - Day Care Center
- **GL** - Housing Co-operative
- **GM** - Alimony
- **GN** - IPA, Deposit and Company Pension Scheme
- **GP** - Account Data
- **GS** - BSU, Youth Saving for Habitation

3.5.2 GLD System Description

Figure 3.5 shows an illustration of the GLD systems. Each GLD system has a part that receives information from third-parties. These third-parties range from banks, insurance companies, employers to daycare centers etc. The received information can be delivered to the GLD systems both by paper and electronically. For the latter, information is copied from Altinn, floppy disks, CD's, streamers, tapes and cassettes to the mainframe.

The received information (both on paper and electronic), is processed and treated in the corresponding GLD system[59]. This process is called "Load". During this stage, all information is automatically controlled. If the delivery has too many errors or is in an incorrect format, the delivery is stopped. In such situations, controls are performed manually by SKD personnel. After validating the information, the systems return a receipt to the submitter, confirming that the information has been received. Errors found in the received information, either by the system or manual by a SKD employee, will be listed in an attachment together with the receipt. The submitter of the information will then have to correct the errors, and send the information all over again. The systems must also keep track of the third-parties who have, or have not delivered information within the deadline. When information is put into a GLD system, it is regarded as quality assured and approved. The quality of the information is of critical importance because of SKD's mass management of information. The information should have minimal manual treatment, and also ensure accurate fiscal treatment.

The received information is stored in individual databases, one for each GLD system, as seen in the figure. The information is also replicated from the individual databases to the GLDB database throughout the whole production year. What figure 3.5 does not show, is the extraction of information from the GLD systems to the DSB system and tax accounts.

The GLD Systems Functions

The GLD systems functions can be divided into the following main parts[59]:

- Production of information material (accompanying, guidance, and information letters, and paper forms) to third-parties
- Reception of tax-related information
- Control of tax-related information
- Generation and distribution of receipts and error-lists

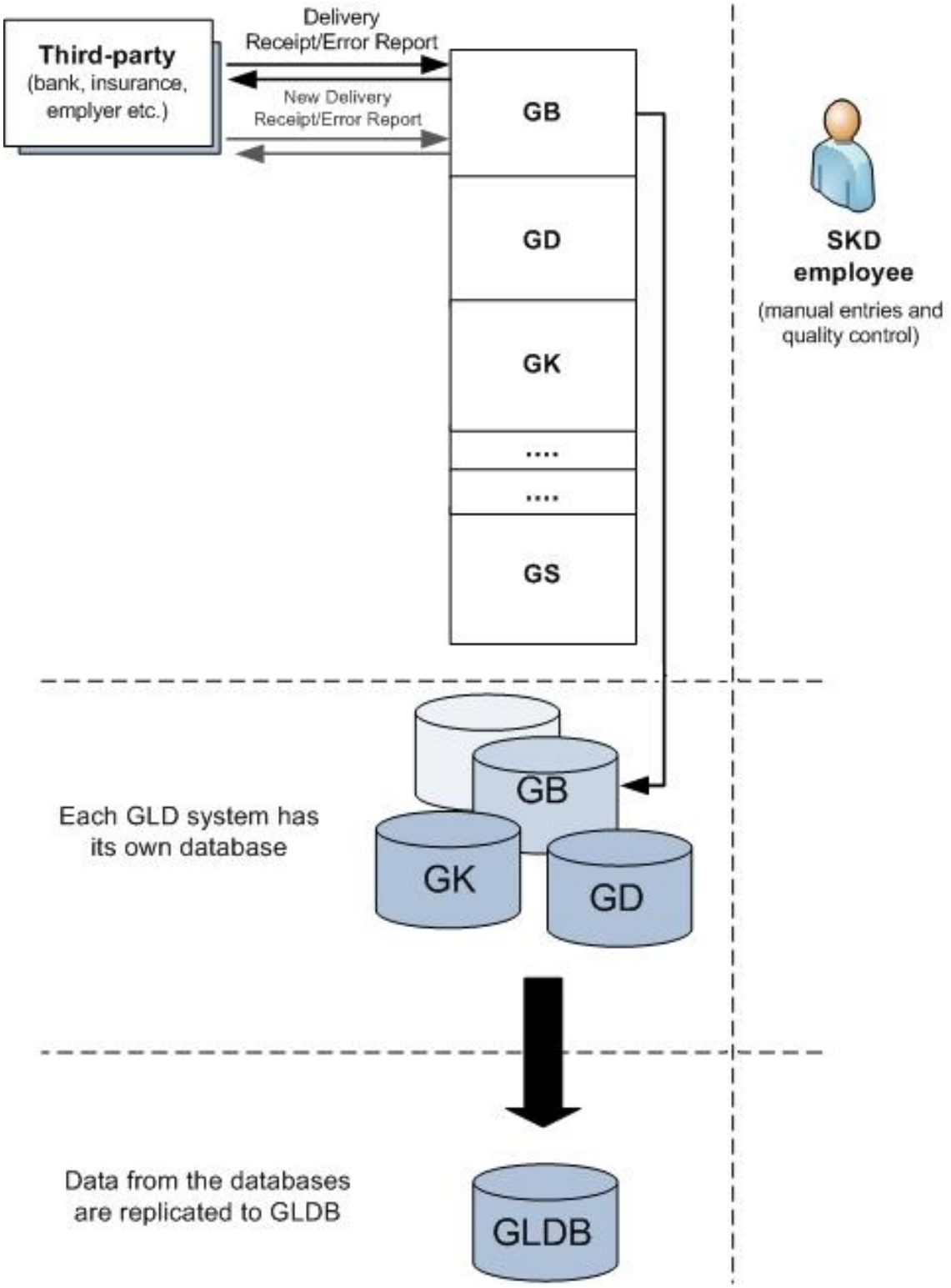

Figure 3.5: GLD system description

- Processing of tax-related information
- Communication of tax-related information

3.5.3 Batch and CICS

The GLD systems consist of several batch programs, in addition to CICS programs, which are written in COBOL. The difference between CICS and batch programs, is that the latter normally don't require any user interaction. Most of the batch programs are initiated by the system itself and set to run once a day, preferably at night. The batch programs groups similar jobs together, and carries them out.

CICS is a teleprocessing monitor, and is used for online transactions on a mainframe[2]. The CICS programs in the GLD systems are used for entries, control and quality control of the information admitted into the systems. The main flow of control is outside the program, and is controlled by the user. The user has to type in the entry she is looking for on a keyboard. Navigation in and between different displays happens with the help of given menu options, function keys (F-keys) or simple commands as plotting in organization or birth number. CICS is not a programming language, but can be written using several languages, such as COBOL. About 5000 access points are connected to SKD's mainframe.

Figure 3.6 shows the main menu to the GLD systems. By typing in a number from 1 to 22, the user can access the different GLD systems, register paper assignments and make queries. By typing the number 1, the GA/LTO system will start; by typing the number 2 will the GB system start etc.

```
SKD CICS ---------------------- GRUNNLAGSDATA ----------------------- 06/07/05
         ********************************************************** 09:54:12
VALG ===>                                                           SK10L153
                                                                    0157
 1 LTO - SAMLEOPPGAVER             2004 12 UNDERHOLDSBIDRAG             2004
 2 SALDO OG RENTER                 2004 13 PENSJONSFORSIKRING           2004
 3 AKSJER / OBLIGASJONER           2004 14 AVREGNINGSDATA               2004
 4 LIVSFORSIKRING                  2004 15 REGISTRERING AV PAPIROPPGAVER 2004
 5 SKADEFORSIKRING                 2004 16 LANDBRUKSOPPGAVER            2004
 6 BOLIGSELSKAP                    2004 17 IDENT. AV LIKNINGSOPPGAVER   2004
 7 BOLIGSPARING UNGDOM             2004 18 OPPSLAG I TRANS.OPPL.- VPS   2004
 8 VERDIPAPIRFOND                  2004 19 OPPSLAG I FORMULAR 2 - VPS   2004
 9 BOLIGSAMEIE                     2004 20 OPPSLAG I AKSJELISTE
10 BARNEHAGER                      2004 21 ELDRE GRUNNLAGSDATA
11 GAVER TIL FORSKNING             2004 22 GRUNNLAGSDATA DEL 2


  F1=HJELP      F3=TILBAKE      F9=AVSLUTT
```

Figure 3.6: GLD main menu

3.5.4 Annual Versions

As mentioned in the introduction of this chapter, the first GLD systems were developed in the late 1980's and early 1990's. At that time, the performance of storage devices was considered to be insufficient for the amount of data needed by all the GLD systems. Also, the storage capacity of the available devices was not considered to be good enough for the amount of data required by the GLD systems year after year. These two problems were

solved using a duplex solution. First, every GLD system got its own database. This took care of the performance problem. Second, each GLD system would be replicated each year, thus creating a new database for each annual version of each GLD system. This way the information stored in each database would be kept to a minimum, thereby solving capacity problems.

Figure 3.7 illustrates how these systems are copied year after year. In order to create a new annual version of a GLD system, the first step is to set up new databases for each system. After creating the tables for the database, it is time to set up views, generate declarations etc., before inserting data into internal tables. When the programs are copied, all references to year, databases and views have to be modified. New laws and legislations requires changes in the systems, and are implemented in the new annual versions. We chose not to go further into detail, since it is a very extensive process.

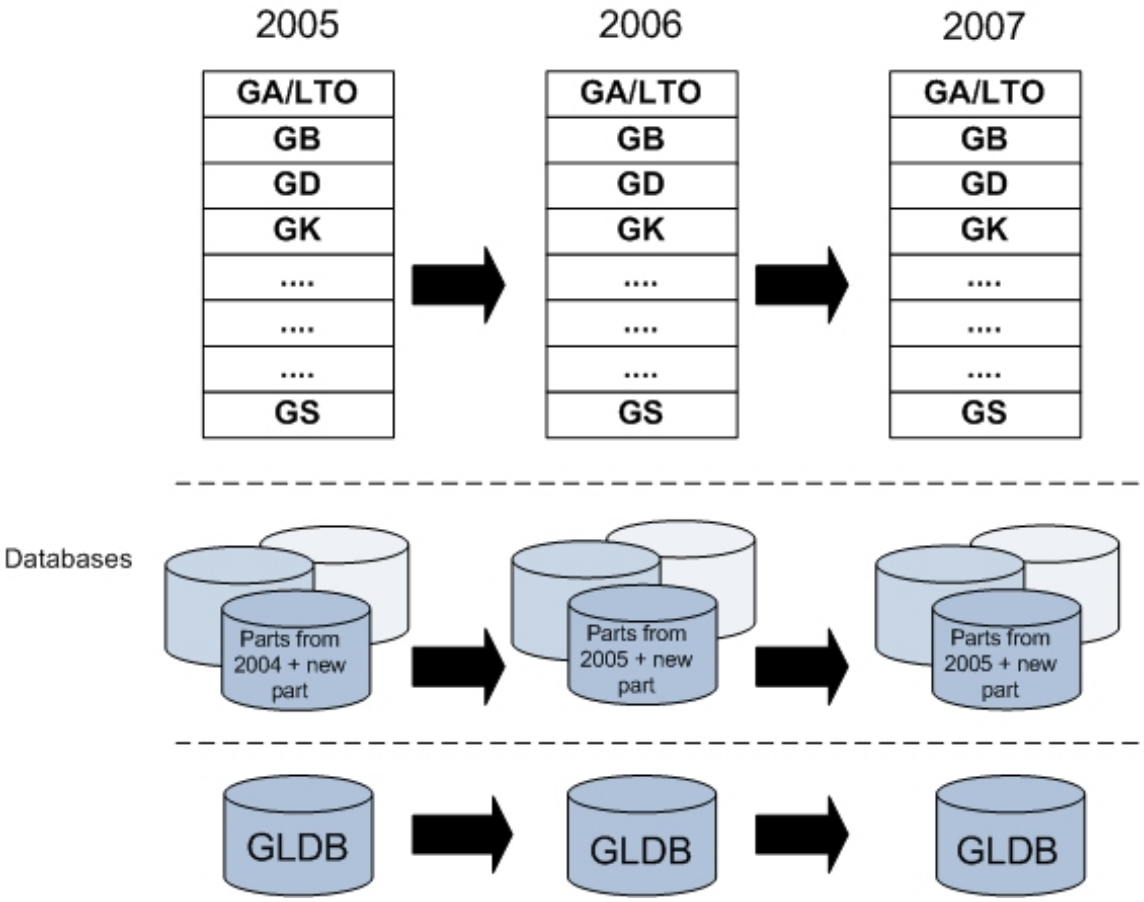

Figure 3.7: Annual versions of the systems

3.5.5 Maintenance

Development, maintenance and management are overlapping processes within SKD's systems. This is especially true for the GLD systems since they have annual versions[54]. The development and maintenance of the GLD systems are performed in cooperation between system owners and SG2. The system owners have the superior responsibility for the systems and must order changes and fault corrections, and further development of the

systems. SG2 is responsible for the actual development.

The GLD systems follows annual cycles concerning changes to the systems. The new annual versions must be ready before receiving tax assessments in January. Large or system critical faults that are discovered under production, or changes that is urgent, is performed continuously if the result of the risk assessment that is done is within an acceptable level. The work with the fault correction and change request starts in March. Last year's experiences with the systems are recorded in system owners experience report *Erfaringsrapport med tiltaksforslag 20nn*[59]. This report together with any changes to the regulations, form the basis for next year's changes to the system. The system owners defines the changes in an initiative list after priority and develops suggestions for suitable solutions. This list contains all changes and faults that have to be corrected, before next year's version of the systems. This list has three classifications of changes and/or error changes:

1. Law- and regulations that must be implemented in the systems
2. Changes and faults that must be corrected before next income year
3. Changes that is desirable to have implemented to the system

At the same time as working with the initiative list, the developers begins to test solutions and describe the test cases in the program Test Director. The figure 3.8 is a simplified graphical representation of the course of events.

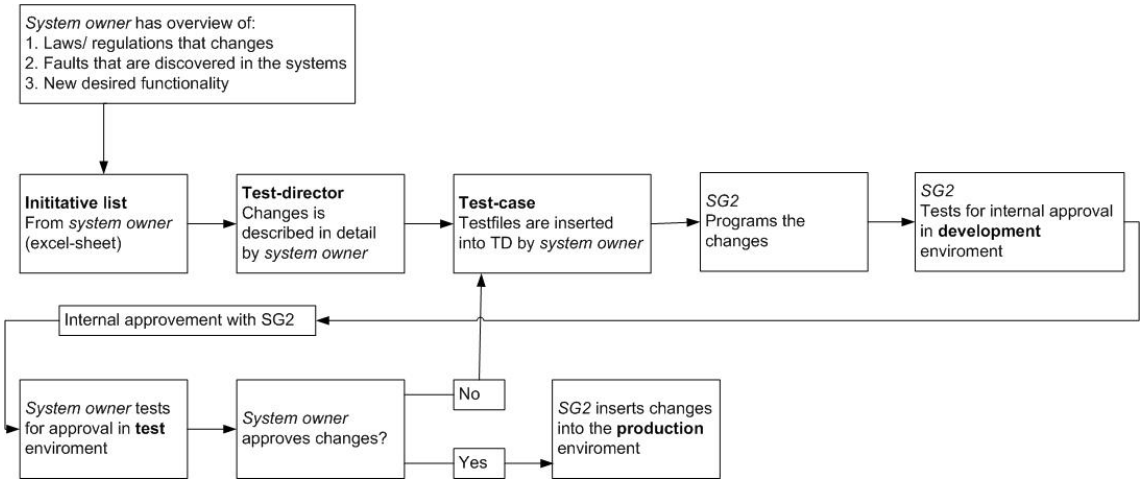

Figure 3.8: System maintenance

Previous annual versions must still be available when new versions arrive, due to after treatment, handling of complaints etc. Some of these systems must be available up to ten years[54]. Previous versions are not prioritized when it comes to correction of errors, although necessary changes are performed quickly if needed.

3.5.6 Selected GLD Systems

The GLD systems was the main source of empirical data for one of our research questions; *What is the current level of software reuse in selected GLD systems?*. In collaboration with SKD, we selected four GLD systems which would be further analyzed. Three of the

systems where chosen because they are representative for all GLD systems, and they are also quite similar in functionality and implementation. This was an important criteria since we where to compare source code for these systems. GA/LTO on the other hand, was chosen because it's implementation differs from the rest of the GLD systems. The selected systems are:

- GA/LTO
- GB
- GD
- GK

GA /LTO, Salary and Deduction System

The GA/LTO system is responsible for retrieving, validating and storing salary and deduction statements received from employers, and it provides this information to the prefilled tax returns. The system also handles the distribution of tax deductions from taxpayers and employers to the appropriate local government. All information in GA/LTO can be entered and accessed through CICS, and the system has several users such as tax offices- and collectors, employers and county revenue offices. The number of employers who reports data to this system is somewhere around 220 000, and these delivered 7 million statements electronically in 2004 [70]. Also, 600 000 statements was received on paper. It required 30 million transactions to process these statements. The GA/LTO system has a different structure and system owner than the other 14 GLD systems.

Figure 3.9 shows the CICS main menu for GA/LTO.

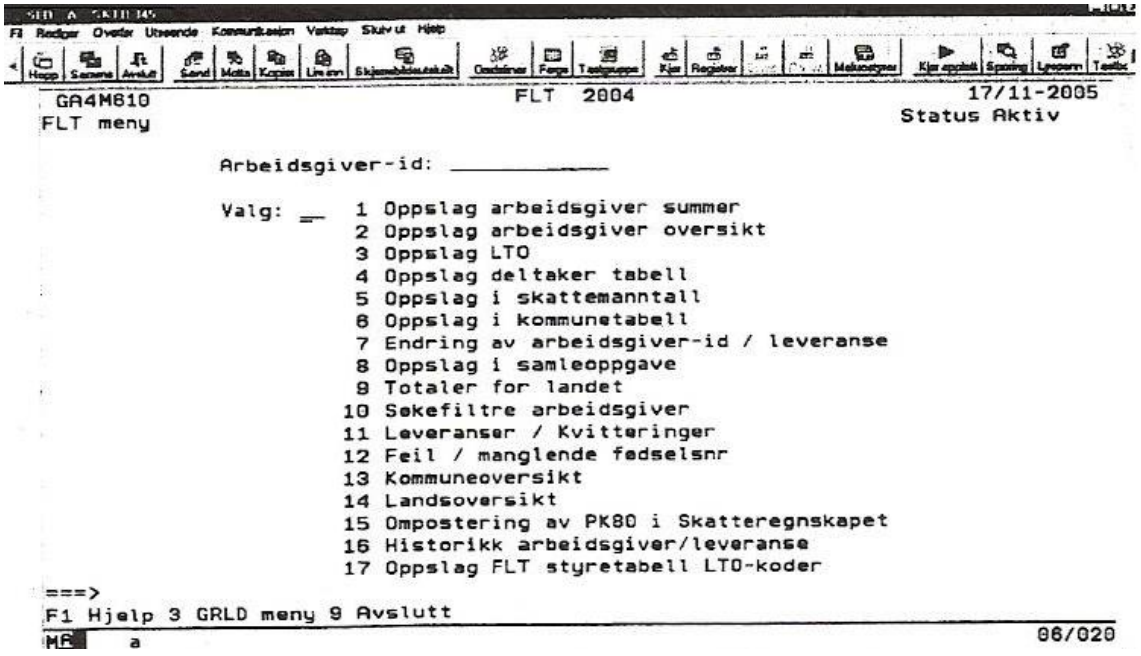

Figure 3.9: GA/LTO CICS main menu

GB, Credit and Interest System

The GB system, Credit and interest, is used as basis for the account number table. This table is used to select which accounts that are recommended as refund of outstanding taxes. It is also used by the Governments' Loan Fund for education by payment of loan and scholarship. The Governments Collection office and the National Insurance Service gets credit and interest information from the basis data register by making queries on personal identification numbers. In the period January to March, the GB systems availability is of critical importance. This is because of high activity in the systems with registration and replication of information. Load, replication, paper withdraws, identification and other jobs runs on evening and at nights. Figure 3.10 shows the CICS main menu for the GB system.

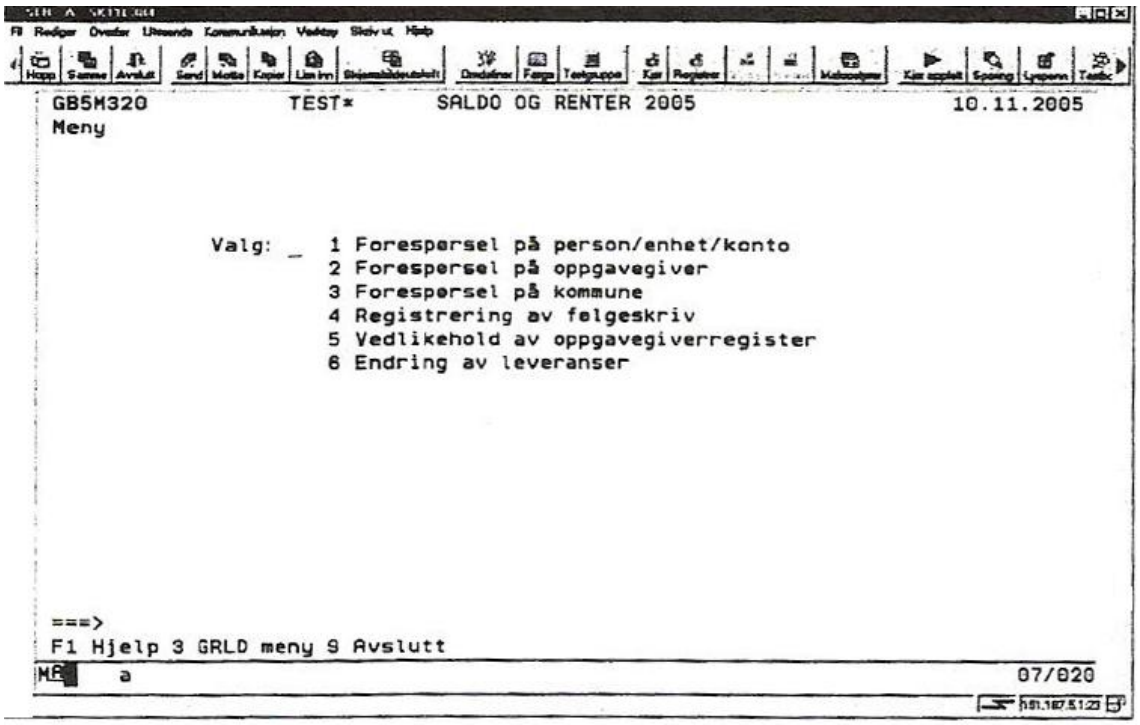

Figure 3.10: GB CICS main menu

GD, Building Association and GK, Day Care Center

There where no textual descriptions of the GD and GK system available. In short, the GD system handles the retrieval, validation and storage of statements from building associations. Figure 3.11 show the CICS main menu for the GD system. The GK system handles the retrieval, validation and storage of statements from day care center. Figure 3.12 show the CICS main menu for the GK system.

3.5.7 The MAG Project

When we executed our case study of software reuse within the GLD systems, SKD was at the same time working on their own modernization project. The project focuses on the

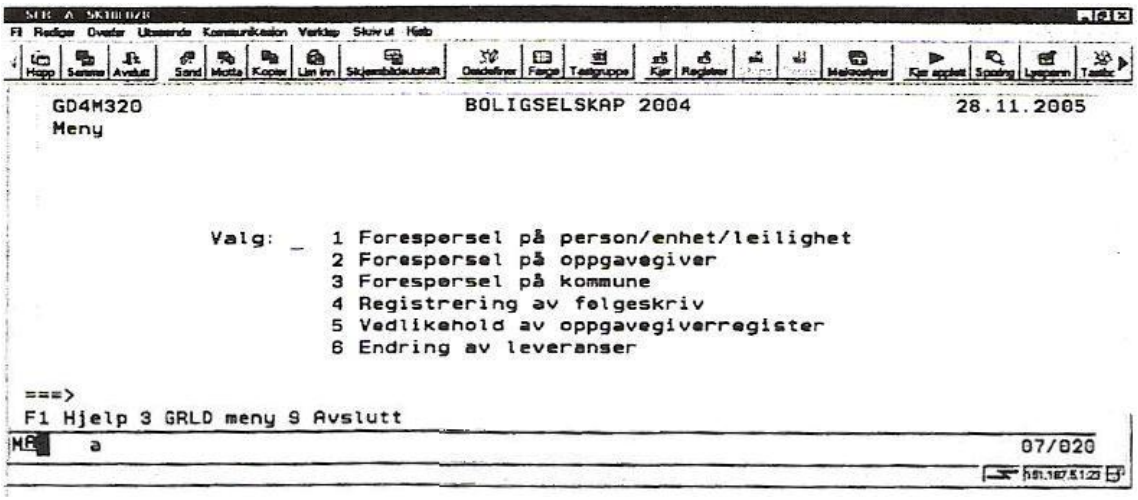

Figure 3.11: GD CICS main menu

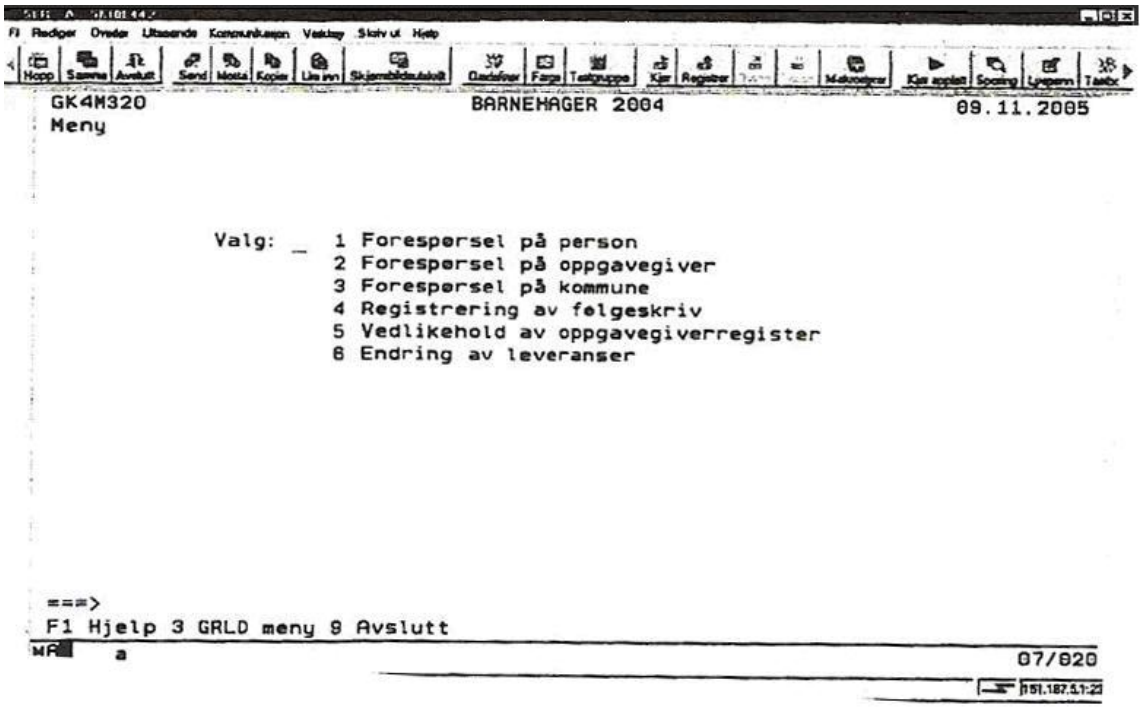

Figure 3.12: GK CICS main menu

modernization of the GLD systems, and is referred to as the *MAG project*[69]. The aim of the project was to select and develop a new platform, and to create and develop a new architecture for the GLD systems.

The project addresses the adaption required to meet the requirements of a new system for employees and senior citizens (a system referred to as SL). On page 40, *Electronic cooperation in the Public Sector*, we discussed several aspects which has lead to the MAG project. Industry and commerce is to have access to complete, electronically services, with around-the-clock availability and short response time. The services will be accessible from the existing websites, and should be designed in such a way that they can be used by other vendors and applications without additional effort from SKD.

The MAG project is divided into two phases. In the first phase, adjustments will be made to the GLD systems so that they can function with the new SL system. Also, the basis data (data provided by the GLD systems) will be made available through a web-application. GLDB will not be modified in phase 1. The goal of phase 2 is the establishment of one, common system for all areas of GLD, and that this system replaces the previously mentioned annual versions. All new systems should be implemented on the new platform, as services available by web.

3.5.8 Summary

To summarize, the GLD systems consist of 15 different systems which collect tax related information from third-parties, and are used for the prefilling of tax returns. For historical reasons, the 15 systems are copied and created over and over again in an annual cycle. This have caused the GLD systems to have reduced maintainability and possible inconsistencies in code and data. In addition, some of the systems must be available for up to ten years, which means that up to 150 systems are kept alive or stored on tape.

Part III

Design of Empirical Investigation

Chapter 4

Research Agenda

This chapter will start by giving a presentation and schedule over the main activities in our master project. Then the research questions, themes and SKD's goal with our project will be presented. Please observe that the empirical strategies were presented in section 2.8 on page 33.

List over main activities in our project:

- Literature review
- Meetings with the developers in SG2
- Review of GLD systems documentation
- Source code analysis
- Review of SKD's Framework for System Maintenance
- Interview about SKD's Framework for System Maintenance
- Survey of the software development process and reuse aspects at SKD

4.1 Schedule and Description over Research Activities

Table 4.1 shows a schedule over our research activities from January 2007 to June 2008. Based on the initial assignment description given by Reidar Conradi and SKD, as seen in appendix A, we conducted a brief literature search of software reuse before our very first meeting with Tore Hovland and SG2 at SKD in Oslo. We had an open mind about the assignment since we had no prior experience with software reuse, legacy or COBOL systems.

The meetings at SKD with SG2 has been very important for our data collection, and is furtherer presented in section 4.4. These meetings have directed our work with the research questions, and provided us with valuable insight, knowledge and feedback.

From the project startup in January to October 2007, we reviewed system documentation handed out by SKD, which ranged from GLD user manuals, Framework for System Maintenance, experience reports, to lists over tables in the GLD databases, in order to familiarize ourself with the GLD systems and their context. At the meeting with SG2 at

October 15. 2007, the research questions were defined and it was decided on a subset of systems within the GLD systems which we would focus on. After this meeting at SKD, we started working on how to collect data for our research questions. In order to find the current state of reuse, we reviewed the system documentation for the selected GLD systems in order to identify reused programs. This process, together with the source code analysis is elaborated in chapter 5. Existing routines that ensures reuse in the organization were mapped with the help of an interview with the person responsible for SKD's Framework for System Maintenance and meetings with the developers at SG2. Reidar Conradi told us about a survey that was performed by prior master students at NTNU about developers attitude towards software reuse, and in February 2008 we performed a similar survey at SKD. During March to June we prepared and analyzed the results from the survey, and

Period	Research activity
January 2007	**Project start-up.** Conducted a brief literature search before the first meeting with SKD in Oslo. We got a short introduction to the GLD system and a selection of system documentation was handed out to us.
May 2007	**Meeting with SKD in Oslo, 25. May** At the end of the month we delivered a report to SKD and Reidar Conradi with our preliminary study. It also included an outline for the report with proposed research objectives.
June 2007 – August 2007	**Summer vacation**, where we worked as software developers. - Thor worked on a SOA-project concerning pension in Trondheim. - Line worked on a SOA-project concerning bonds in Oslo.
August 2007	**Meeting in Trondheim, 7. August** Tore Hovland visited us at NTNU for a meeting with Reidar Conradi. We continued to elaborate us in the GLD systems.
October 2007	**Meeting with SKD in Oslo, 15. October** Determined research goals in collaboration with SKD and Reidar Conradi. We were assigned which GLD sub-systems we would focus on; GA/LTO, GB, GD and GK. Documentation for the systems was handed out to us. After the meeting we were occupied with reviewing the system documentation.
November 2007	**Meeting with SKD in Oslo, 26. November** We presented our findings from the documentation review and got feedback from SKD. They also introduced us to CICS and batch, before handing out source code for some of the programs in the selected GLD systems.
December 2008 – February 2008	These months we were occupied with the source code analysis and preparation to the survey and interview.
February 2008	**Meeting with SKD in Oslo, 15. February** We interview the person in charge of the Framework for Software Maintenance, discussed the MAG Project with SKD and carried out the survey.
March 2008 – June 2008	The results from the survey was prepared and analyzed, and we started on the discussion of the research questions. The literature search was picked up again, since we discovered many new articles on software reuse.
May 31. 2008	**Delivery of report**

Table 4.1: Schedule over research activities performed from January 2007 to June 2008

proceeded with the writing of this master thesis. We also audited our literature study because we discovered new research and articles on the field of software reuse, which we did not find earlier.

4.2 Research Questions and Themes

We decided to take three of the topics from the initial assignment description (see Appendix A) as a basis for our research, and came up with the following research questions:
RQ1: What is the current state of software reuse in the selected GLD systems?
RQ2: Do reused components have lower change and defect rate compared to other components?
RQ3: What is the emphasis on software reuse in current development process?
RQ4: What is the potential for systematic reuse, and how can it be achieved?

As we stated in the introduction of this thesis, our research can be divided into four themes. The connections between the different themes, research questions, and contributions are illustrated in table 1.1 on page 7. This way of structuring the report was recommended to us by our supervisor, Reidar Conradi. The themes are as following, with the corresponding research questions are shown in parenthesis:

- T1: Review of state-of-the-art literature on software reuse
- T2: Investigation of reuse level within selected GLD systems (RQ1)
- T3: Investigation of SKD's development process (RQ3)
- T4: Investigation of opportunities for systematic reuse in SKD (RQ4)

RQ2 is not covered by any of the themes, and remains unanswered in this thesis. When our research was initiated in January 2007, RQ2 was originally one of the questions we planned to answer. However, this research question is inconclusive of several reasons. As our knowledge about the GLD systems and SKD increased, we realized that this research question was not clearly defined. The programs selected by SKD and us, which we would analyze, were all developed with source-code reuse. In fact, all program within the GLD systems are created annually, thus there were no empirical data (components) to compare the reused components against. Also, we were not able to find any rates or measurements of defects within the GLD systems. During our meetings we were told by the developers at SG2 that the GLD systems have low change density, and faults are seldom discovered. Annual changes to the systems are documented in experience reports after each year, but this data alone was inadequate to answer the research question.

4.3 SKD's Goal with our Research

Tore Hovland and SG2 proposed to specific goals with our research.

SKD goal 1 Propose a process which assures software reuse
SKD's existing software development process should be extended to take software reuse into account. This is to ensure that new components are developed with reuse in mind. This goal 1 will be answered through our work on RQ4.

SKD goal 2: Propose an ideal architecture for the GLD system, with focus on reuse
When we first initiated our research, SKD had originally asked us to propose an ideal GLD architecture for the new platform. Later in 2007, SKD had taken upon this task themselves with the MAG project, although we were not informed of this until the meeting in February 2008. We were disappointed since this research goal became obsolete, and because of the late notice. Due to this, SKD suggested that our contribution would be a review of the proposed architecture, with focus on software reuse. We used one week in March 2008 to read and evaluate the suggested GLD architecture, but we soon discovered that this new goal was not within the scope of our project. SKD was also not clear about their evaluation criteria for design, requirements and architecture, which made it difficult for us to contribute with constructive critique and suggestions. We also felt that we had neither the required knowledge about the systems internal structure nor the experience with this type of work.

4.4 Meetings with the Developers from SG2 and its Limitations

During this project we had five meetings with employees at SG2, as described in section 4.1. In addition to the meetings, we also had continuous contact with SG2 on e-mail and telephone. In general we prepared an agenda and a set of questions before each meeting, which we sent to Tore Hovland. This way he could get a hold of relevant informants and help us organize the meetings. At these meetings we usually gathered two to six developers from SG2 for an hour where we went through the agenda for the meeting and updated them on our results. The questions we had prepared and a draft of our report was often discussed in plenum with the developers. This type of meeting have similarities to a "focus group interview", which is an interview performed in a group of six to twelve people[4].

We did not use a tape recorder at the meetings, but both of us took thorough notes of what was said. At the end of each meeting we went through our notes and registered them in a text document which we sent to both Reidar Conradi and Tore Hovland. This was to increase the credibility of our notes. It also resulted in feedback, corrections and new ideas, which we used as a basis for further work. A resume of the main parts of the meetings is found in appendices C to F.

These meetings are examples of a qualitative research activity. As mentioned, we increased the credibility of the data from the meetings by transferring it back to its informants. The results have no transferability because the topics of discussion and context was specific to SKD and the GLD systems. A source of errors to this type of data sampling is whether or not we asked the right type of questions, and if the informants provided us with correct answers. Informants may give wrong answers if they misinterpret, don't know the answer, acts as they knows the answer or are lying to the questions. Focus group interviews may be problematic because the persons in a group have different personalities[4]. Some people may have great influence on the rest of the group, other may be to dominating, shy or simply jabber to much. This type of behavior can cause us to miss important information from people who might have something important to say.

Chapter 5

Investigation of Reuse Level in the GLD Systems (T2)

This chapter covers the research performed in order to answer our first research question, RQ1: *"What is the current state of software reuse within the selected GLD systems?"*. We used three different approaches in this part. First, documentation and program descriptions [64][65][62][63] of all GLD systems where reviewed in order to find components used across different applications. Our second approach was to analyze source code on a subset of the GLD programs in order to find similarities between these programs. This was done by using a textual differencing tool combined with some self-developed programs. Findings from our literature review of software reuse states that it is insufficient to find an organizations maturity level solely based on comparison of source code lines or components [28][50]. Thus, our last approach is one described by Morillo et al.[33], which determines the maturity level in Koltun and Hudsons reuse maturity model.

5.1 Reviewing the GLD Documentation

The documentation for the GLD systems contained listings of all programs in each GLD system (a GLD system consists of several programs). In the documentation the programs were grouped according to their purpose. SKD referred to the grouping of Batch-programs as "procedures", and the CICS programs as "applications". For consistency, we will refer to both of these as applications, thus one application can have one or more programs. The relationship between what we referrers to as GLD systems, applications and programs are shown in figure 5.1. Our motivation for performing this analysis was to investigate the amount of black-box reuse across the different applications in the GLD systems, since programs used by several applications would be used as-is.

We read through all of the program descriptions from the documentations, and created complete lists of all the programs used by the GLD systems, both Batch and CICS applications. Four lists were created, two for GA/LTO and two for the remaining 14 GLD systems (GB, GC, GD, GE, GF, GG, GH, GJ, GK, GL, GM, GN, GP and GS):

- GA/LTO, Batch programs
- GA/LTO, CICS programs

- GB to GS, Batch programs
- GB to GS, CICS programs

We then proceeded with a textual search for each program in the lists, and documented which application the program was used by. We crosschecked the lists to see if any programs appeared in more than one application. Programs used across different applications could indicate the presence of black-box reuse. The result was documented in a spreadsheet, as the example in table 5.2 shows.

GA/LTO is quite different from the rest of the GLD systems. The difference is so considerable that GA/LTO has its own documentation, while all other GLD systems are described in one document. For this reason we chose to analyze it separately. GA/LTO is made up of 29 applications and 124 programs. 40 of these programs are CICS, while the remaining are BATCH programs. The other 14 GLD systems share the same documentation. The results from GA/LTO was entered in a separate spreadsheet. The results for all the remaining GLD systems was entered in a single spreadsheet, which counted 163 programs (111 CICS- and 52 BATCH programs). This gives a total of *287 cross-checked programs*, covering all GLD systems.

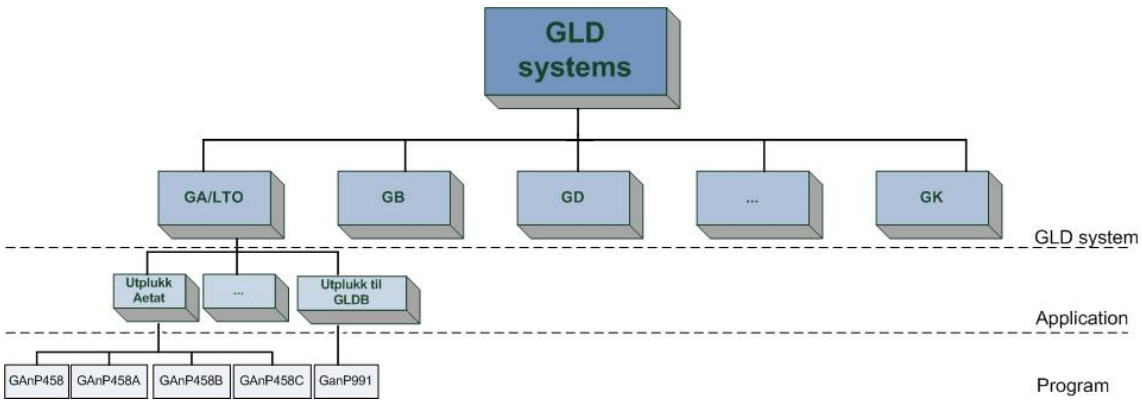

Figure 5.1: There are several GLD systems. A GLD system consists of one or more applications, and applications uses one or more programs

Naming of the Programs

The naming of the programs in the GLD systems follows a standard set by SKD: GZnbyyy[59]. An explanation of the different characters is given in table 5.1. An example on the use of the naming standard, is GB6P301. From the first two digits, we see that the program belong to the GB system. The next characters tells us that the program belong to year 2006 and the environment is production. The three last characters tells us that the number of the program is 301. Programs with identical numbering performs basically the same task but for a different GLD system.

5.2 Analyzing the Source Code

The most common way of measuring the amount of reuse within a software system is by counting the number of reused components or lines, divided by the total number of

GZnbyyy	
G	The main system
Z	The specific GLD system
n	The last digit in of the year of revenue
b	Tells which environment the program is in: P for production, T for testing and D for development
yyy	The numbering of the program

Table 5.1: The naming of the programs

Programs

Applications	GAnP035	GAnP052	GAnP054	GAnP093	GAnP098	GAnP099	GAnP200	GAnP210	....	....
FLT online										
FLT ladefase	1						1	1	1	1
FLT utplukk DSB				1						
FLT utlukk utland										
FLT utplukk Svalbard										
FLT utplukk riksarkivet										
Utplukk ENG/PK80					1	1				
Utplukk nullte termin										
Gen av ugyldige fødselsnr										
Utskrift av ugyldige fødselsnr										
Utskrift av samleoppgave										
Gen av deltakerregisteret										
Gen av adressefeil										
Utplukk av selvstendig næringsdrivende										
Gen av T-link-reg		1	1							
Genererer skattemantallet i test										
Utplukk til GLDB										
Tilbakeføring fra PSA										
Utplukk til barnehagesystemet										
Behandling av fil fra Livsforsikring/IPA										
Utplukk Aetat										
Utplukk SI										
Utplukk RTV-daglig kjring										
FLT utplukk til SKDs datavarehus										
Utplukk SKARP										
Retteprogram for kommunetabellene										
Utplukk SSB										
Utplukk RTV - ERGO										
Altinn-programmene										

Table 5.2: Sample from spreadsheet, which shows how the cross-checking was performed. Applications are aligned vertically and the programs horizontally.

components or lines[50][17]. We discussed this approach in 2.5.2 on page 26. The first approach we described focused on measuring the amount of black-box reuse, while the approach covered in this section measures the amount of white-box reuse, or code/design scavenging. SKD handed out source code for three of the of the GLD systems, namely GB, GD and GK. We were given three CICS programs (301, 302 and 304) from each of these systems and for two different years (2006 and 2007). This gives a total of 18 programs. The programs were chosen because we wanted to establish how much of the code is identical between different programs in GLD systems, and how much is changed between the annual versions. This gives us two forms of software reuse which we have focused on. The first one is the difference, or similarity, between two programs from *different GLD systems* with identical years and numbering (for example GB6P301 and GD6P301). The other form of reuse we wanted to analyze was the similarity between *two annual versions* of the same program (for example GB6P301 and GB7P301). Since the GA/LTO system differs in structure from the 14 other GLD systems (and the numbering of the programs are not the same as in the other GLD systems), we decided not to compare any programs from GA/LTO. To do this we would have had to select random programs from GA/LTO for comparisons with programs from GB, GD and GK, and this would not be suitable.

The programs were written in COBOL370, a programming language which neither of us had any previous experience with. In our first set of code to analyze, the 2007 version of the three systems, there where no line breaks present in the files. The result of this was that all code appeared on a single line in the source file. To solve this, we wrote a Java-program which separated the single line of code into lines according to the COBOL-syntax. Since the files would be compared using a "diff" or "merge" tool, we also wanted to remove all comments in order to get a more precise result. We modified our tool to remove all comments from the source code. Also, in order to make the source files more identical, we replaced all multiple blank lines with a single blank line. Without comments and multiple blank lines, the files varied in size from 879 to 2455 lines. The total number of lines was almost *30 000*.

5.2.1 Generating reports

The source code was compared using a tool called WinMerge[73], a visual differencing and merging tool for text files. The program compares two text files by showing which lines are modified, added or deleted in each of the text files. It generates this informations into a report, where added lines are marked with a '+', removed lines '-' and modified lines with an '!'. The report is similar to the one generated by the Linux/Unix diff-command, as seen in figure 5.2. There are both pros and cons when using a textual based merge tool for this purpose. First, it threats the source code as pure text, thus disregarding the programming language it was written in[31]. Considering that our subject of research was written in COBOL, which is a rather old language, this is clearly a benefit. The detection of changes are however limited to line-level granularity[29]; it shows where a change was made and what it was. It gives no context of the changes, and structural and syntactic information in the source code is ignored.

When we generated these reports, we noticed that slight differences would sometimes appear when comparing the same files. These differences occurred when we shifted the left and right side of the comparison (see figure 5.3). It seemed like the differencing tool would, for example, in the first comparison recognize a line as modified, but when shifting

```
220  !            05 W02-TILB-MIS-Z              PIC S9(11)V COMP-3 VALUE 0.
221  !            05 W02-BET-MIS-Z               PIC S9(11)V COMP-3 VALUE 0.
222
223           01  W05-GODKJENNINGS-TAB.
224               05 W05-REG-TID                 PIC X(26) OCCURS 50 TIMES.
225  --- 157,172 ----
226               05 SEKUND                      PIC 99 VALUE 0.
227               05 MAP-AAR                     PIC 9999 VALUE 0.
228
229  +        01  W01-TIMESTAMP                 PIC X(26) VALUE SPACES.
230  +
231           01  W02-HJELPE-BELOP.
232               05 W02-ANT-OPPGAVER-T          PIC S9(9)V  COMP-3 VALUE 0.
233  !            05 W02-INNB-BELOP-T            PIC S9(11)V COMP-3 VALUE 0.
234               05 W02-ANT-OPPGAVER            PIC S9(9)V  COMP-3 VALUE 0.
235               05 W02-ENDRING                 PIC S9(9)V  COMP-3 VALUE 0.
236  !            05 W02-INNB-BELOP              PIC S9(11)V COMP-3 VALUE 0.
237               05 W02-ANT-OPPGAVER-Z          PIC S9(9)V  COMP-3 VALUE 0.
238  !            05 W02-INNB-BELOP-Z            PIC S9(11)V COMP-3 VALUE 0.
239
240           01  W05-GODKJENNINGS-TAB.
241               05 W05-REG-TID                 PIC X(26) OCCURS 50 TIMES.
242  ****************
243  *** 269,456 ****
244               05 W06-REG-TID-11              PIC X(26) VALUE SPACES.
245               05 W06-REG-TID-12              PIC X(26) VALUE SPACES.
246
247  !        01  TEST-FELT                      PIC X(10)
248  !                                           VALUE '->KOMMAREA'.
249
250  -        01  TEST-FELT2                     PIC X(10)
251  -                                           VALUE '->TS-OPPGA'.
```

Figure 5.2: An extract from a report generated by Winmerge. Modified lines are shown with an '!', added lines with a '+' and removed lines with a '-'.

the left and right side it would be considered an added line. Because of this, we decided that we would generate two reports for each time we compared two files with source code. When one change is made to, for example, a variables name, this change will be counter as several changes since the new name will be used in the rest of the program. One could argue that a change like this should only be counted once. We chose to threat this scenario as several changes, mainly since a developer at SKD would have to alter all lines where this is required. By treating it as several changes, it is easy to see how much work effort is required when developing new annual versions, or even a new program in for a GLD system.

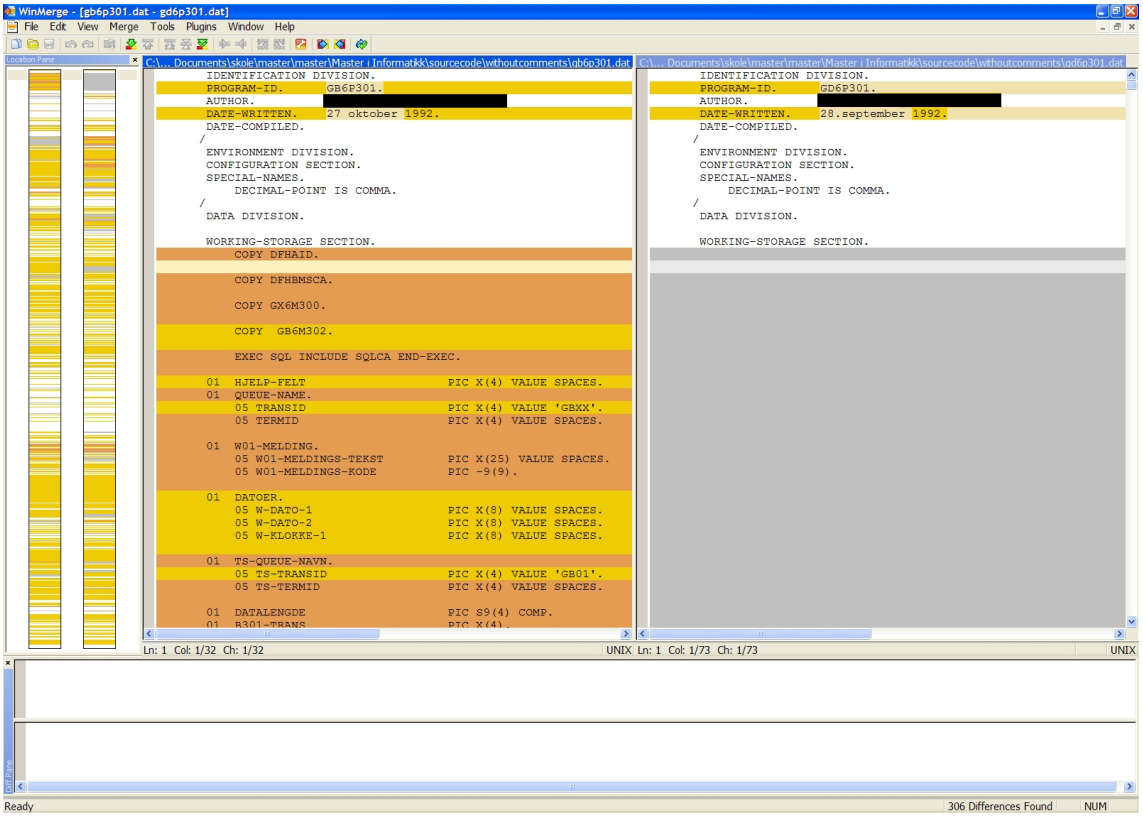

Figure 5.3: Winmerge. The program compares two documents, and highlights modified, added and removed lines.

We first compared programs with the same numbering and year, but from different GLD systems. This produced a total of (GB, GB and GK) * (2006 and 2007) * (301, 302 and 304) = 3 * 2 * 3 = 18 reports. Since we had decided to double check each report by switching the left and right side of the comparison, this gave us a total of *36 reports*.

Next when we compared two annual versions of the same program, but from different years, the numbers of reports became (GB, GD and GK) * (301, 302 and 304) = 9 reports. This was also validated by making a second, inversed comparison, and produced a total of *18 reports*.

The 56 reports was approximately 6MB of pure text, so to manually review them seemed like a very time consuming task. We wrote a simple program in Java which read the reports and counted the number of times each marker occurred (!, + or -), and it also identified which of the two files in the comparison the marker was connected to. All findings was then entered into a spreadsheet, which contained the total length in code lines for each

program, number of code lines when comments was removed, number of added, removed and modified lines. The spreadsheet then calculated the number of common lines by subtracting modified and added lines from the number of lines without comments. The results from the spreadsheets are presented in chapter 71.

5.3 Approach for Classifying with the Reuse Maturity Model

In the previous sections we discussed two approaches used to estimate the current state of software reuse within selected GLD systems. Findings from our literature review of software reuse stated that it is insufficient to find an organizations maturity level solely based on comparison of source code lines or components (software reuse ratio) [28][50].

We used the Reuse Maturity Model as seen in figure 2.3 (page 25) developed by Koltun and Hudson[17] to provide a quantitative indicator of the current level of reuse within the selected GLD systems. The columns in the maturity model represents different phases of reuse maturity; Initial/chaotic, Monitored, Coordinated, Planned and Ingrained. Quantifying reuse maturity levels can be difficult[50]. The Reuse Maturity Model has ten dimensions, and each of these can be in a different phase. So, how would we identify a maturity level when the dimensions where in various phases? For this reason, we adopted an approach described by Morillo et al.[33] to determine which maturity level was appropriate for the GLD systems. In their paper, they use three reuse factors for determining the maturity level:

- Repository structure (r)
- Software development architecture (s)
- Administrative management (g)

Each reuse factor consists of several activities grouped together. Each group has a maximum of three points, and in order to obtain three points in a group all activities must be present within the organization. The three factors r, s and g can each give a maximum of 9 points. The formula for calculating the maturity level is expressed as:

$$M_1 = \sqrt{r^2 + s^2 + g^2} \tag{5.1}$$

The value of M_1 determines the maturity level:

- 0 - 3, Initial/Chaotic
- 3 - 6, Monitored
- 6 - 9, Coordinated
- 9 - 12, Planned
- 12 ->, Ingrained

All the activities contained in each reuse factor was listed in three tables (one table for each reuse factor). We sent the tables to our contact person at SKD, Tore Hovland, along with an explanation for each activity. Next to each activity there was a field which had to be filled as either *true* or *false*, indicating if the specific activity was present or not in the development of the GLD systems. When the tables were returned to us, we started

converting the true and false values to the equivalent score, according to the paper by Morillo et al.[33]. These tables are shown in the figures 7.4, 7.5 and 7.6 in chapter 7.

Figure 7.4 contains the table of activities in the "Repository structure" factor. A repository is capable of storing artifacts which may be reused, and is therefor one of the most important requirements for software reuse. Figure 7.5 contains activities found in the "Software development architecture" factor. This factor evaluates the existing software architectures orientation concerning reuse. The final factor, "Administrative Management", is evaluated in figure 7.6. This factor covers administrative aspects of software reuse, specially those concerned with human resources allocated to reuse. The last activity in this factor concerns reuse obtained by previous projects, and the score is given by $3 * averageR_1$, and can be in the range of 0 to 3. $averageR_1$ is the average of the reuse rates, but the paper by Morillo et. al. [33] did not describe how to obtain these rates. We calculated the reuse rate by comparing the annual versions we investigated in 5.2, thus $\frac{R}{T}3$, where R is the total amount of reused lines and T is the total number of lines. This is the most frequently used approach when measuring the level of software reuse[28][50] and is used by both the industry and academia, as stated in section 2.5.2.

Chapter 6

Investigation of SKD's Development Process (T3)

We used three different sources in order to answer RQ3: *What is the emphasis on software reuse in the current development process?* These were:

- A survey among 25 developers at SKD
- Meetings with the developers in SG2
- Review of SKD's Framework for Software Maintenance
- Interview with the person in charge of SKD's Framework for Software Maintenance

A summary of SKD's Framework for Software is presented in section 3.4 on page 41. In this chapter we present the preparations for the survey and interview.

6.1 Planning the Survey of the software development process and reuse aspects at SKD

The survey was performed using a questionnaire, which was developed by two diploma students at NTNU in 2002[37], and had previously been answered by developers at Ericsson AS in 2002. Core parts of that questionnaire was used at EDB Business Consulting[52] and at Mogul in 2003[75]. The three companies were selected to participate in the survey since they had experience on component reuse and wanted to cooperate with NTNU in this research[38]

The questions were divided into categories. We removed three of the categories and modified some of the answers because they were not relevant for our purpose. The first category was Personal Info. Even though the survey was done anonymously, some information was still required to get better understanding of what the candidates based their answers on. Then a category of general reuse questions followed. They were formed as how important reuse is in relation to different assertions, and how important different technologies and tools are. Next was a category dealing with components and reuse to see what the software developers thought about the current component reuse situation. The three last questions were about requirements and change of requirements during the course of a project. To

increase the reliability of the survey, the questionnaire included a definitions of the different concepts used. The respondents were free to add comments after each category of questions. The questionnaire is found in appendix H, and the results is found in chapter 8 and in Appendix I.

Originally each of the questions in the questionnaire was used to study one or more research questions in the master thesis of Naalsund and Walseth[37]. We had no assumptions of what the results of the survey would be. Our primary goal with the survey was to use it as a starting point for research questions RQ3 "*What is the emphasis on software reuse in the current development process*" and RQ4 "*What is the potential for reuse*".

We chose to make use of someone else's questionnaire because we wanted the possibility to combine our results with the ones from the previous studies. The studies from Ericsson AS, EDB Business Consulting and Mogul had 26 respondents in all and together with our survey, it was possible to combine the results from a total of 51 respondents. This way we participated in the research already performed by NTNU on developers attitude towards software reuse. We also believed that it was a good thing to use a survey that was already tested out and quality assured.

6.2 Planning the Interview about SKD's Framework for System Maintenance

After reviewing SKD's Framework for Software Maintenance, we prepared for an interview with the person in charge of the framework. The Framework for Software Maintenance gave no mentioning on software reuse, so we assumed that it did not take reuse into account. We wanted to confirm this assumptions with the interview, and we also wanted feedback on how the current process could be altered to support reuse. The questions had no predetermined answer options and was sent to the interviewee on week in advance so she would have time to prepare herself. A resume of the interview can be found in appendix G.

Part IV

Results

Chapter 7

Results from Investigation of Reuse Level(T2)

In this chapter we present the results from the research associated with T2, the investigation of reuse level in the GLD systems. First we present the results from the system documentation review. We then proceed with the results from the source code analysis, and in the last section we present our classification with the reuse maturity model. This theme covers, as previously mentioned, research activities concerned with RQ1:"What is the current state of software reuse in selected GLD systems?".

7.1 Results from Review of GLD Documentation

As we described in section 5.1 on page 59, we created lists of all programs (and which application it was connected to). The lists were crosschecked in order to see if any programs appeared in more than one application. Our motivation for this review was to see if we could find programs or components which were being reused across applications.

7.1.1 GA/LTO System

GA/LTO is made up of 29 applications and 124 programs. 40 of these programs are CICS, while the remaining are BATCH programs. In all, we found four programs used by more than one application, of which all were BATCH programs. Table 7.1 shows the programs and the applications which used them.

7.1.2 Remaining GLD Systems

As stated previously, the other 14 GLD systems (GB, GC, GD, GE, GF, GG, GH, GJ, GK, GL, GM, GN, GP and GS) share the same documentation. The documentation contained descriptions of 163 programs; 111 CICS- and 52 BATCH programs.

As with the GA/LTO analysis, none of the CICS programs in this examination appeared in more than one application. By doing a closer investigation of the spreadsheet, we noticed repeatedly series of programs often occurred in the same applications. By series we

Programs ⟶

Applications ↓

	GAnP428	GAnP430	GAnP456	GAnP510
FLT utplukk DSB	1	1	1	
FLT utlukk utland	1	1	1	1
FLT utplukk Svalbard	1	1	1	1
FLT utplukk riksarkivet	1		1	
Utskrift av samleoppgave				1
FLT utplukk til SKDs datavarehus	1	1	1	
Utplukk SKARP	1			
Utplukk SSB			1	
Utplukk RTV - ERGO			1	
	6	4	7	3

Table 7.1: Results from documentation review of the GA/LTO system. Four programs were reused across applications

mean that the name of the program ends with the same identification number. Example of a series is GBnP301, GDnP301, GEnP301, GGnP301, GKnP301 and GLnP301. We identified seven such series. These were (presented with an description):

- **GznP301**: Checks all the requests that are received from the main menu, then calls for the right transaction (six programs: GBnP301, GDnP301, GEnP301, GGnP301, GKnP301 and GLnP301)
- **GznP302**: Online registration of different paper assignments (six programs: GBnP302, GDnP302, GEnP302, GGnP302, GKnP302 and GLnP302).
- **GznP303**: Terminates the update which is initiated by transaction XX01 (six programs:GBnP303, GDnP303, GEnP303, GGnP303, GKnP303 and GLnP303).
- **GznP304**: Online control of different paper assignments (six programs: GBnP304, GDnP304, GEnP304, GGnP304, GKnP304 and GLnP304).
- **GznP305**: Online control of different paper assignments for one delivery (six programs: GBnP305, GDnP305, GEnP305, GGnP305, GKnP305 and GLnP305).
- **GznP380**: Main program - for building association, gifts, damages, kindergarten and housing co-operative (five programs: GDnP380, GEnP380, GGnP380, GKnP380, and GLnP380).
- **GznP393**: Sub routine - gets name/address on national identity number or organization number for the current for building association, gifts, damages, kindergarten or housing co-operatives (five programs: GDnP393, GEnP393, GGnP393, GKnP393, and GLnP393).

We chose the top four programs in the list above as subjects for the source code analyze, because of their identical numbering and description. This way we could analyze them for similarities both between the different GLD systems, and between annual versions.

Among the 52 BATCH programs, we found one program that was used in two applications; this was the program GXnP002, used in both the Load- and Identification routine.

7.2 Results from Code Analysis

We performed source code analysis on 18 programs, as described in 5.2 on page 60. We focused on identifying two forms of software reuse. The first one is the difference, or similarity, between two programs from different GLD systems, with the same year and numbering (for example GB6P301 and GD6P301). The second one was the similarity between two annual versions of the same program (for example GB6P301 and GB7P301).

Table 7.2 shows the percentage of modified, added and common lines from our first analysis, where only the numbering and year of the programs name are equal. The first thing we noticed when rearranging our findings into a single table, was that there was an equally amount of changes made both years. For example, GBnP301 and GDnP301 has the same percentage, 50%, of common lines in both 2006 and 2007. This is true for every row in the table. GKnPyyy and GDnPyyy had the highest amount of common lines, up to 79%. The median, or middle value, was 60% common code lines, and the minimum was 36%.

Program for 2006	Modified	Added	Common
GB6P301 - GD6P301	44 %	6 %	50 %
GB6P301 - GK6P301	47 %	7 %	46 %
GD6P301 - GB6P301	34 %	2 %	65 %
GD6P301 - GK6P301	30 %	1 %	69 %
GK6P301 - GB6P301	29 %	2 %	69 %
GK6P301 - GD6P301	21 %	0 %	79 %
GB6P302 - GD6P302	46 %	15 %	39 %
GB6P302 - GK6P302	47 %	17 %	36 %
GD6P302 - GB6P302	47 %	4 %	49 %
GD6P302 - GK6P302	33 %	6 %	61 %
GK6P302 - GB6P302	41 %	2 %	57 %
GK6P302 - GD6P302	21 %	1 %	78 %
GB6P304 - GD6P304	52 %	1 %	47 %
GB6P304 - GK6P304	52 %	1 %	47 %
GD6P304 - GB6P304	38 %	3 %	59 %
GD6P304 - GK6P304	35 %	2 %	63 %
GK6P304 - GB6P304	25 %	1 %	73 %
GK6P304 - GD6P304	21 %	0 %	79 %

Program for 2007	Modified	Added	Common
GB7P301 - GD7P301	44 %	6 %	50 %
GB7P301 - GK7P301	46 %	8 %	46 %
GD7P301 - GB7P301	34 %	2 %	65 %
GD7P301 - GK7P301	30 %	1 %	69 %
GK7P301 - GB7P301	29 %	2 %	69 %
GK7P301 - GD7P301	21 %	0 %	79 %
GB7P302 - GD7P302	50 %	11 %	39 %
GB7P302 - GK7P302	51 %	14 %	36 %
GD7P302 - GB7P302	47 %	4 %	49 %
GD7P302 - GK7P302	31 %	9 %	61 %
GK7P302 - GB7P302	41 %	2 %	57 %
GK7P302 - GD7P302	21 %	1 %	78 %
GB7P304 - GD7P304	52 %	1 %	47 %
GB7P304 - GK7P304	52 %	1 %	47 %
GD7P304 - GB7P304	38 %	3 %	59 %
GD7P304 - GK7P304	35 %	2 %	63 %
GK7P304 - GB7P304	25 %	2 %	73 %
GK7P304 - GD7P304	21 %	0 %	79 %

Table 7.2: Common lines in programs from different GLD systems, given in percentage. The total for each row should add up to be 100%, but may vary -1 or 1 % due to rounding of decimals

Table 7.3 shows findings from our second analysis, where annual versions was compared against each other. The amount of common lines was much greater between the annual versions. According to table 7.3, between 97% and 99% of the lines are identical.

7.3 Classification with the Reuse Maturity Model

In section 5.3, we described our approach of classifying the current state of reuse within the GLD systems. The result of this classification is presented in this section.

As previously stated, we chose the Koltun and Hudson Reuse Maturity Model for classification (figure 2.3 on page 25). In order to select an appropriate level of reuse, we adopted

Program	Modified	Added	Common
GB6P302 - GB7P302	1 %	0 %	99 %
GD6P302 - GD7P302	1 %	0 %	99 %
GK6P302 - GK7P302	1 %	0 %	99 %
GB6P304 - GB7P304	1 %	0 %	99 %
GD6P304 - GD7P304	2 %	0 %	98 %
GK6P304 - GK7P304	2 %	0 %	98 %
GD6P301 - GD7P301	2 %	0 %	98 %
GK6P301 - GK7P301	2 %	0 %	98 %
GB6P301 - GB7P301	3 %	0 %	97 %

Table 7.3: Common lines in annual programs, given in percentage.

an approach described by Morillo et al.[33]. Their approach uses three reuse factors; repository structure (r), software development architecture (s) and administrative management (g). The factors consists of several activities organized in three tables, one for each reuse factor. The tables 7.4, 7.5 and 7.6 shows all activities, along with a score specific for the development of the GLD systems. The tables were filled in by SG2 themselves with true or false values before we converted these values into an equivalent score. The score which each activity gives is shown under *Max points* in the tables.

Table 7.4 shows the activities in the *Repository structure* factor. It shows that the GLD systems are not supported by a software repository. However, SKD stated that components concerning user interface are stored in a repository structure. Components are not automatically indexed, and thus there are no mechanisms for retrieving them. The total sum for repository structure (r) was 1.1.

Activities concerned with the *Software development architecture* factor are shown in table 7.5. The software architecture deployed in the GLD systems is not based on any structured techniques, nor is it created using object-oriented techniques. However, the systems are created by reusing source code (copy/paste). Analysis, design and other documentation are also currently reused. Testing is highly prioritized at SKD, as the table indicates. The total sum for "Software development architecture" (s) was 4.4.

The final factor, *Administrative Management*, is evaluated in table 7.6. The table shows that no human resources (HR) are dedicated to the development or management of reusable components. Incentives for developers who creates or makes use of reusable components are non-existing, as well as any planning for reuse. The last activity in the "Administrative Management" factor concerns reuse obtained by previous projects. We calculated the average percentage of reused lines from table 7.3, giving us an average of 98,33%. This gives $g = \frac{98.33}{100}3 = 2.95$

To summarize, we ended up with the following coefficients; r = 1.1, s = 4.4 and g = 2.95. This gives the following equation:

$$M_1 = \sqrt{r^2 + s^2 + g^2} = \sqrt{1.1^2 + 4.4^2 + 2.95^2} = 5.41 \tag{7.1}$$

Figure 7.1 shows that the value of M_1 corresponds to "Level B, Monitored" in the Reuse Maturity Model.

1 - Repository Structure = r			
1-1 Representation of Information		**SKDs points**	**Max points**
1.1.1	Repository structure to store source code components	0	0,5
1.1.2	Repository structure to store user interface components	0,5	0,5
1.1.3	Repository structure to store executable components	0	0,5
1.1.4	Repository structure to store analysis and design components	0	0,5
1.1.5	Repository structure to store to store documentation components	0	0,5
1.1.6	Repository structure to store project management components	0	0,5
1-2 Classification and Recovery Techniques		**SKDs points**	**Max points**
1.2.1	Manual indexing of components (and) classification team	0	0,6
1.2.2	Recovery of components through characterizing attributes	0	0,6
1.2.3	Automatic indexing of components that accept codification	0	0,6
1.2.4	Automatic indexing of binary components	0	0,6
1.2.5	Automatic indexing of components of design, analysis, documentation and project management	0	0,6
1.2.6	Recovery of components through documentary query	0	0,6
1-3 Management and Automation Tools		**SKDs points**	**Max points**
1.3.1	System of authorization, rejection and modification of components	0,6	0,5
1.3.2	Existence of statistics calculation module on component reuse	0	0,5
1.3.3	Automated announcement of component incorporation or modification	0	0,5
1.3.4	Techniques for automatic creation of repository from previous projects	0	0,5
1.3.5	Techniques for automatic integration of components in new projects	0	0,5
	Sum	**1,1**	**9**

Table 7.4: RMM: Activities in the "Repository structure" factor

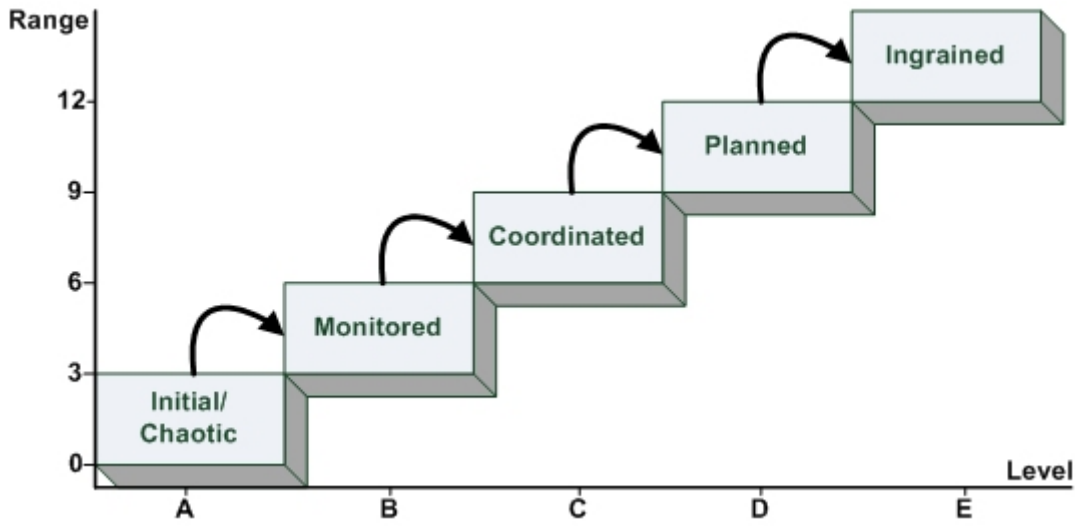

Figure 7.1: Phases in the Reuse Maturity Model (RMM)

2 – Software Development Architecture =s			
2-1 Software Architecture used in the organization		**SKDs points**	**Max points**
2.1.1	Architecture based on structured techniques (exclusive with 2.1.2)	0	0,3
2.1.2	Architecture based on object and events-oriented techniques (excl. 2.1.1)	0	1,2
2.1.3	Use of Graphic User Interface (GUI)	0	0,75
2.1.4	Use of CASE environments	0	0,75
2-2 Types of reuse applied by the organization		**SKDs points**	**Max points**
2.2.1	Reuse of components based on source code, type, functions, etc	0,5	0,5
2.2.2	Reuse of User Interface components (graphic or non-graphic)	0	0,5
2.2.3	Reuse of executable libraries and binary objects with multiple Interfaces	0	0,5
2.2.4	Reuse of components of analysis and design	0,5	0,5
2.2.5	Reuse of components of documentation	0,5	0,5
2.2.6	Reuse of additional components of software project	0,5	0,5
2-3 Testing techniques used by organization		**SKDs Points**	**Max points**
2.3.1	Techniques of verification and validation	0,6	0,6
2.3.2	Unit testing	0,6	0,6
2.3.3	Integration testing	0,6	0,6
2.3.4	System testing	0,6	0,6
2.3.5	Following of ISO standards for testing	0	0,6
	Sum	**4,4**	**9**

Table 7.5: RMM: Activities in the "Development Architecture" factor

3 – Administrative Management =g			
3-1 Administrative aspects of reuse support		**SKDs points**	**Max points**
3.1.1	Human resources (HR) specific to reuse development	0	0,5
3.1.2	HR specific to repository management (librarians)	0	0,5
3.1.3	HR specific to maintenance of components and training	0	0,5
3.1.4	HR specific to identification of reusable components	0	0,5
3.1.5	HR responsible for measuring and evaluation of achieved reuse	0	0,5
3.1.6	HR whose main goal is to encourage reuse	0	0,5
3-2 Incentives and Planning		**SKDs Points**	**Max points**
3.2.1	Existence of incentives for developers	0	0,75
3.2.2	Existence of awards for reusers	0	0,75
3.2.3	Existence of short term planning of reuse (improvements-progress)	0	0,75
3.2.4	Existence of planning to achieve the next level of maturity	0	0,75
3-3 Reuse of previous projects		**Max 3 points**	
3.3.1	Capability for reuse of previous projects	$\frac{98.33}{100}3 = 2.95$	
	Sum	**2,95**	**9**

Table 7.6: RMM: Administrative management

Chapter 8

Survey Results (T3)

The survey was an important source of information on theme T3: "Investigation of SKD's development process (RQ3)". This chapter presents the results of this survey. Further analysis, discussion and validity threats will be discussed in section 10.1 on page 95. The questionnaire was given to the group leader of SG2. Since we wanted as many responds as possible, he encouraged other system groups to participate in answering the questionnaire. For this reason we do not know the exact number of people who were asked to fill in the questionnaire. It took about three weeks from we sent out the first questionnaire, until we had received the last one. We had to remind people to answer, since the developers were busy and had hard to find time to answer. We ended up with a total of 25 answers to the questionnaire. The questionnaires were answered and delivered electronic with the help of Microsoft Word and e-mail.

The variables under the Component section did not have high enough level of measurement to do more powerful statistical analysis. That is why we only chose to do frequencies and cross tabulations. We refer to Appendix I for the complete results from the questionnaire. For the General questions G1 to G3, we chose to display the results in a clustered column chart in the same was as the three previous surveys at Ericsson, Mogul an EDB Business Consulting. We used SPSS version 15.0 for the analysis of the questionnaire. SPSS stands for Statistical Package for the Social Sciences, and it is a program that is used for analyzing data. It can perform a variety of data analysis and presentation functions, including statistical analysis and graphical presentation of data[68].

8.1 Respondents

A total of 25 people answered the questionnaire. The respondents came from various development departments within SKD, and had experience with different computer systems. 17 of the respondents of the questionnaire were system developers, which constitutes 68%. The remaining respondents had different roles as adviser, architect, consultant, team leader or responsible for a system.

Figure 8.1 shows that most of the developers have been working at SKD from 0-4 or 5-9 years, which together makes 64%. 7 of the respondents have been working at SKD for more than 17 years, according to table I.2 in Appendix I.

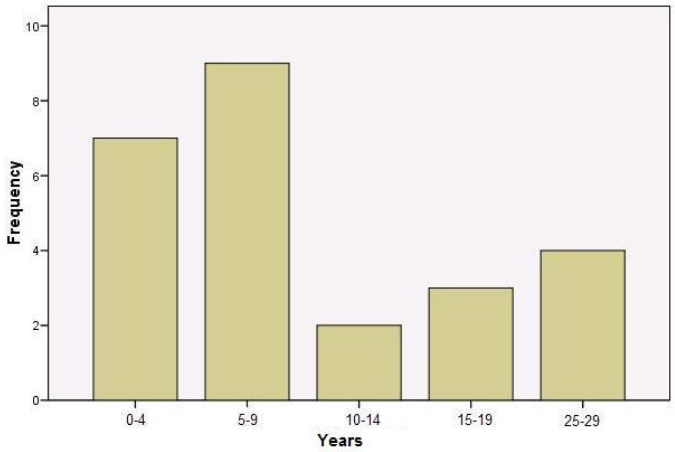

Figure 8.1: Results from question P2: Number of years working at SKD (n=25 respondents)

We asked the respondents which programming and design languages they were familiar with, and currently using. From the answers we could see that 16 of the respondents were familiar with COBOL. Of the 12 respondents familiar with Java, 7 of these were also familiar with COBOL. Most of the developers are currently working with programming and design languages such as COBOL, Java and Oracle Warehouse Builder.

Figure 8.2 shows that 20% of the respondents has high school as their highest completed academic degree, 48% has completed an bachelor degree and 28% has completed an master degree.

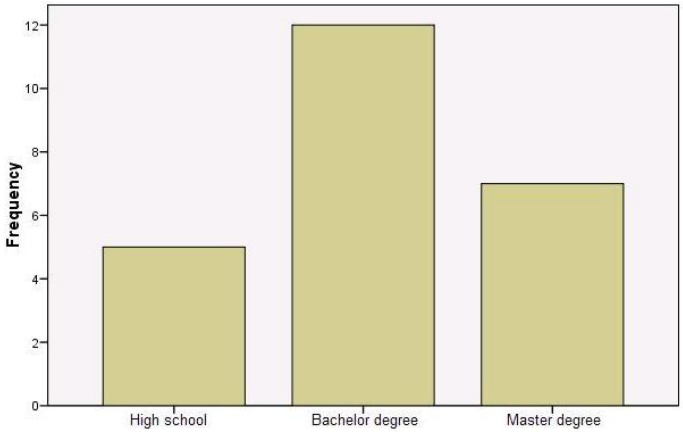

Figure 8.2: Results from question P6: Highest completed academic degree (n=24 respondents)

8.2 General Questions G1

The respondents were asked how important they considered reuse to be in achieving benefits such as lower development costs, shorter development time, higher product quality, more standardized architecture and lower maintenance costs. As figure 8.3 shows, the re-

spondents thinks that reuse is important to achieve all the different benefits. The results does not point out any of the questions as extra important.

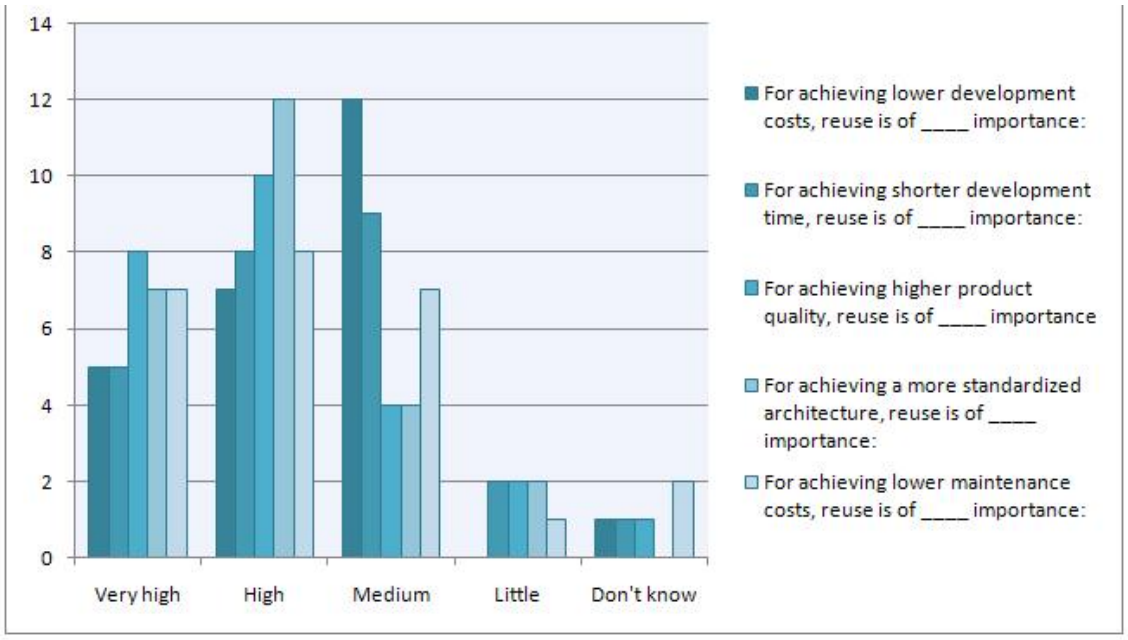

Figure 8.3: Results from questions G1a-d. Columns are in the same sequence as in the description field

Comments:

As mentioned, the respondents were free to add comments to each category of questions. One person said that it is very practical to reuse code by copying code they think is useful when making new systems in COBOL or Easytrieve. They also reuse copy members which are code used by several programs or systems, or use programs who do the same function for several systems by calling up these programs (or also copy members) when needed.

Another person said that for most of the systems he worked on, they make new versions every year because of new tax rules. Most of the tax rules are the same, but some of the rules are new every year. The reuse of code is very important, and makes them able to create new versions easily.

A third person stated that reuse also carries risk in the sense that errors are spread proportionally to the degree of reuse. A too religious approach may be counterproductive: in the pursuit of reusing a module or class (incorporate it in a new system), one may adopt a bad overall design in the new system just to be able to reuse the module. If it is a simple module, writing new code may be the right course. He also said that it is often much more quicker to write the necessary code when one needs it, than to locate a reusable module or class that may or may not exist. Done right, however, reuse is important.

One person commented on that making components reusable require extra resources. Another person stated that in practice, reuse is not achieved very often. It would need a lot more to see real benefits, and he said that it is important to standardize the implementation of a few common tasks.

The last comment was made by a person who did not think that they had opportunity to reuse components here, unless he made a system that was similar to something he had

made before. Then he knew his code and could reuse what he wanted.

8.3 General Questions G2

The respondents were asked how useful or important they found reuse/component based technologies, OO technologies, testing, inspections, formal inspections and configuration management to be. Figure 8.4 shows how the answers were spread out. It seems that the respondents agree that most of the technologies mentioned in the question is of high importance. Testing stands out from the rest; it is of very high importance. OO technologies and configuration management has a higher degree of "I don't know" answers than the rest.

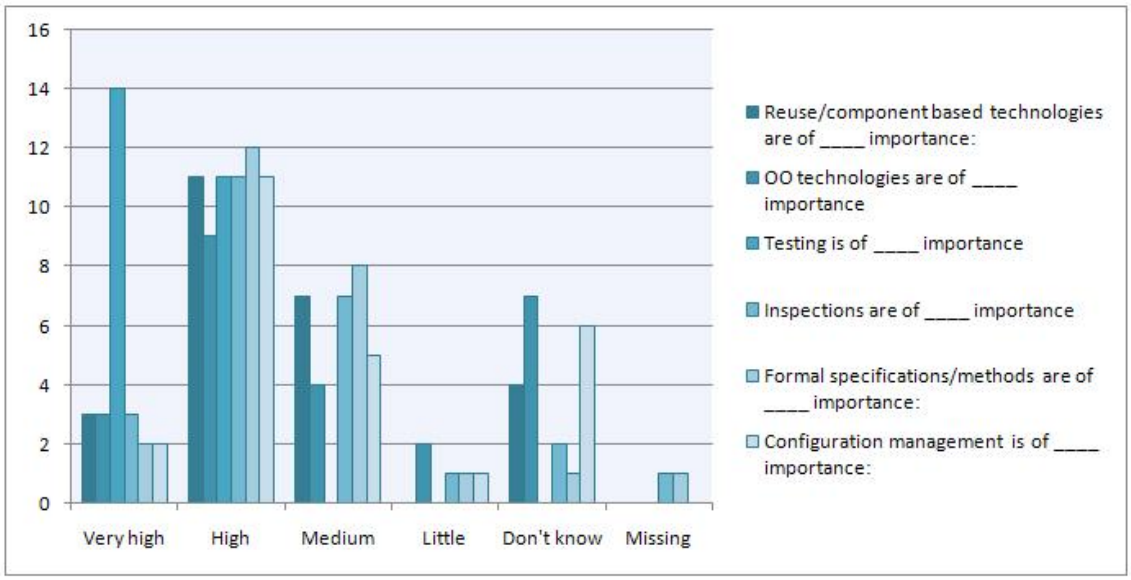

Figure 8.4: Results from questions G2a-f. Columns are in the same sequence as in the description field

8.4 General Questions G3

The respondents were asked how useful or important they considered requirements, use cases, design, code and test data/documentation, with respect to reuse, to be. Figure 8.5 suggest that the participants find most of the artifacts useful and important to reuse. This questions stands out from the two previous general questions, since more respondents answered "little" or "don't know" to this question.

8.5 Component Questions

We asked if the construction of a reuse repository with extra component documentation would or would not be worthwhile. 72% of the respondents stated that it would be worthwhile. 64% of the respondents said that there is no clearly defined way when decideing

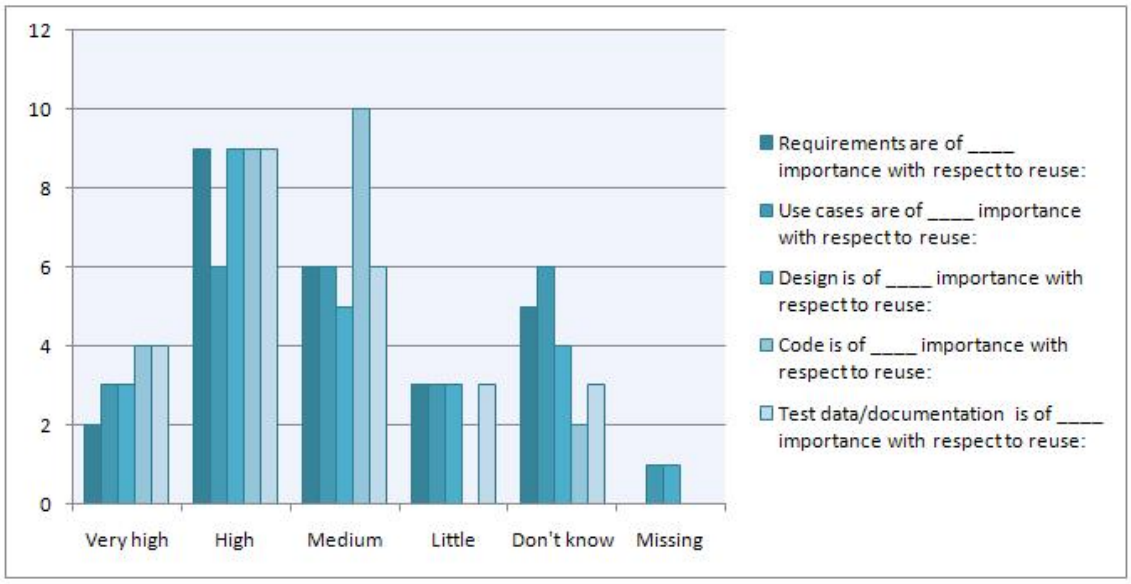

Figure 8.5: Results from questions G3a-e. Columns are in the same sequence as in the description field

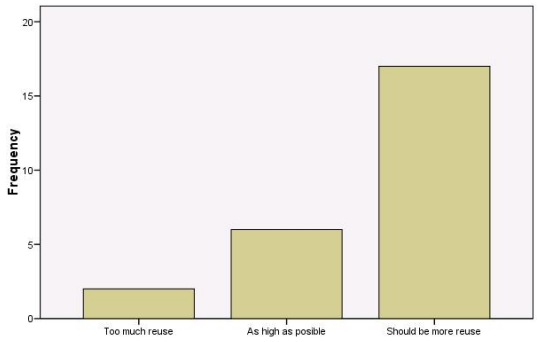

Figure 8.6: Results from question C1:
During development:
(n = 25 respondents)

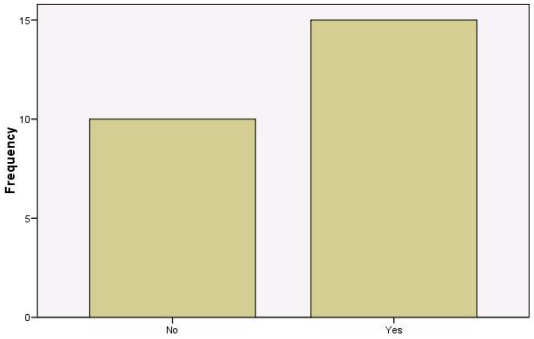

Figure 8.7: Results from question C2:
Is the current process working?
(n = 25 respondents)

whether to reuse a code/design component "as-is" or "with modification", or to make a new component from scratch. 16% follows guidelines and another 16% consult experts as seen in figure 8.8.

68% of the respondents stated that a code/design component which is reused (and possibly) modified, is usually more stable and cause less problems, and 28% said that it is about equal to a component created from scratch. 52% of the respondents said that integration may cause some problems when reusing a component, 32% stated that it usually works well, while 8% felt it was difficult. This is shown in figure I.28 in Appendix I.

The next question was to what extend the respondents feel affected by reuse in their work. As seen in figure 8.9, 32% feel that the extend of reuse is high to very high. 28% feel that it is medium, and 32% feel that it is little or no reuse. 68% of the respondents felt that there should be more reuse during development, as seen in figure 8.6, while 8% felt that it was too much reuse during development. 40% of the respondents thinks that the

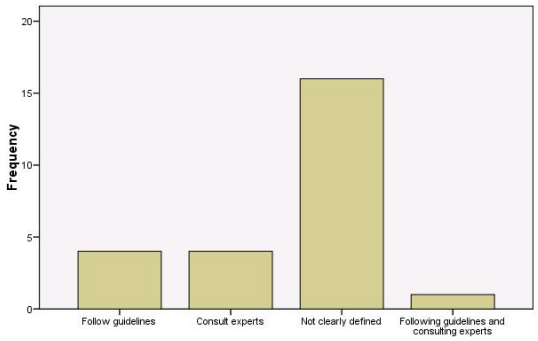

Figure 8.8: Results from question C5: How do you decide to reuse a code/design component?(n = 25 respondents)

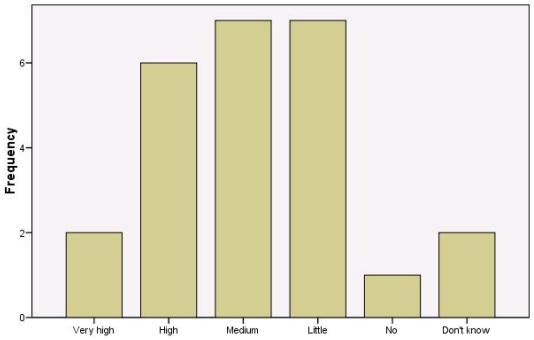

Figure 8.9: Results from question C9: How affected are you by reuse?(n = 25 respondents)

process of finding, assessing and reusing existing code/design is not functioning, while the majority of 60% feel that it is. A total of 60% said that the existing code/design components is sometimes sufficiently documented, while the remaining respondents have split opinions between yes and no. We asked the ones that answered "sometimes" or "no" to this question if they thought that this was a problem. Most of the respondents who felt that the existing code/design components was not sufficiently documented, felt that this was a problem.

The last question was an open question, where the respondents were asked to write down their main source of information about reusable components during implementation.

- Randomly searching code. Consulting older system experts. Reuse has no agenda in my daily work, WHATSOEVER! It's not prioritized by the "bosses"
- Experience
- System documentation
- I ask persons who have worked with similar tasks. I have also, as the years go by, made myself a little "library" of code I think may be useful in the future
- Look at the description in heading and at the code
- The GLD system
- Persons who know the reusable components
- The code itself
- My own knowledge and the experience of my co-workers
- Tips from other developers about specific preexisting modules. Also, reuse is built into our existing practices: A lot of systems are made a new each year, and the reuse in these cases occur when the new code is copied from the last year - this happens regularly
- General system knowledge
- My own experience with those components, and advice from the developers I'm working with. Also some help from Google when third party components is an option

8.6 Requirements

The last category of questions had tree questions involving requirements. 4 of the 25 respondents (16%) did not answer the last category. This was probably because they did not see the last page. The first question was whether the requirement negotiation process at SKD is working sufficiently. 64% of the respondents said it is working sufficiently. In a typical project, 48% stated that it is no particular trend; requirements are sometimes rigid and other times flexible. 32% said that the requirements are usually flexible in a typical project, as seen in table I.34 in Appendix I. The last question was whether requirements is often, sometimes or seldom changed or renegotiated during a development project. 40% of the respondents said that requirements are sometimes changed and 36% said often, during this time.

Comments:

One person wrote that he had answered the questions based on his own experience with the systems he is involved with. These are mainly "bread and butter" systems (systems that are made a new each year), and the changes are thus incremental. The dynamics in these systems are no problem, he states; they are expected and they work very close with the people specifying, often using an approach similar to prototyping. SKD do however at any given time run a lot of projects and his impression is that in these, unexpected changes do happen during the course of the project. The reasons for this are legitimate, and among them are insufficient original specifications, badly chosen technology and subcontractors exploiting things in the specifications that the specifies took for granted; this necessitates new and more detailed specifications, and is accompanied by "quarrels" about whether the new specifications represent changes (for which the subcontractor may charge extra) or if they only represent clarifications (no extra cost).

8.7 Cross Tabulation Analysis of Component Questions

We performed cross tabulation analysis on questions C2 "*Do you feel that the process of finding, assessing and reusing is functioning*" and C9 "*To what extend do you feel affected by reuse in your work*". The results from the cross tabulation analysis are found in Appendix I.

8.7.1 Cross Tabulation Analysis of Question C9

About equal half's of the respondents felt that the process of finding, assessing and reusing is or is not functioning. The majority of respondents stated that the construction of a reuse repository would be worthwhile and that it should be more reuse. The majority of respondents answered that it is not clearly defined how one should decide to reuse a code or design component, and they also stated that code or design components that are reused is more stable and cause less problems that components created from scratch. When integrating a reusable component, the respondents are almost equally split between "usually works well", or it "may cause some problems". The respondents are also split between if there is or is not any extra effort put into testing or documenting potentially reusable components. The majority of the respondents agree on that the design or code of reusable components is sometimes sufficiently documented. The difference between the

respondents answers might be because they work in various development departments and projects.

8.7.2 Cross Tabulation Analysis of Question C2

Question C2 "*Do you feel that the process of finding, assessing and reusing is functioning*" was cross tabulation analyzed with questions C3a, C10a, C4, C5 and C7. To summarize, the majority of respondents who feel that this process is working, thinks that:

- The construction of a reuse repository would be worthwhile
- The integration of reusable components may cause some problems
- It is not a clearly defined way for how to decide whether to reuse a code or design component

8.8 Results from Survey combined with Previous Surveys

We added the results from our survey at SKD to a selection of the results from the three previous surveys on developers view on component reuse. As mentioned in section 6.1 on page 67, the four surveys have some common questions. This section displays the results from the questions that were answered by all four companies. Questions C5 and C9 are included even though they were only answered by three of the companies. The results will be discussed in section 10.2.

There are some differences in the naming of the questions, but we chose to proceed with the naming from our survey at SKD. Survey results and questionnaires from Ericsson, Mogul and EDB Business Consulting can be found in the master thesis of Naalsund and Walseth[37], Wang[75] and Sæhle[52]. The survey at Ericsson had 9 respondents, Mogul had 7 respondents, EDB Business Consulting had 10 respondents and SKD 15 respondents, which makes a total of 51 respondents. Questions C2, C3a, C3b, C4, C5, C9, R1 and R3 is presented in the remainder of this section.

C2: Do you feel that the process of finding, assessing and reusing existing code/design components is functioning? Table 8.1 show how the results were spread, and that 4 respondents did not answer this question.

Company	Yes	No	Number of respondents
Ericsson	4	5	9
Mogul	4	2	6
EDB Business Consulting	7	0	7
Skattedirektoratet	15	10	25
	30	17	47

Table 8.1: Results from question C2 answered by all four companies

C3a: Is the existing code/design components sufficiently documented? Table 8.2 shows how the results were spread. We see that 4 respondents did not answer this question.

Company	Yes	Sometimes	No	Don't know	Number of respondents
Ericsson	0	3	5	X	8
Mogul	0	6	1	0	7
EDB Business Consulting	0	7	0	X	7
Skattedirektoratet	3	17	5	X	25
	3	33	11	0	47

Table 8.2: Results from question C3a answered by all four companies

C3b: If "sometimes" or "no": is this a problem? Table 8.3 shows how the results were spread. We see that 10 respondents did not answer this question.

Company	Yes	No	Number of respondents
Ericsson	7	1	8
Mogul	5	1	6
EDB Business Consulting	6	1	7
Skattedirektoratet	15	5	20
	33	8	41

Table 8.3: Results from question C3b answered by all four companies

C4: Would the construction of a reuse repository, with extra component documentation etc.: Table 8.4 shows how the results were spread. We see that 5 respondents did not answer this question.

Company	Not be worthwhile (No)	Be worthwhile (Yes)	Number of respondents
Ericsson	4	3	7
Mogul	2	5	7
EDB Business Consulting	2	5	7
Skattedirektoratet	7	18	25
	15	31	46

Table 8.4: Results from question C4 answered by all four companies

C5: How would you decide to reuse a component "as-is", reuse with modifications, or make a new component from scratch? Table 8.5 shows how the results were spread. We see that 14 respondents did not answer this question, where 9 of these were from Ericsson.

C9: To what extend do you feel affected by reuse in your work? Table 8.6 shows how the results were spread. We see that 9 respondents did not answer this question, where all of these were from Ericsson.

R1: Is the organizations requirement negotiation process working sufficiently? Table 8.7 shows how the results were spread. We see that 7 respondents did not answer this question.

R2: In a typical project: Was answered by all four companies, but we did not understand the results from Mogul and EDB Business Consulting.

Company	Consulting experts	Following guidelines	Using GSN RUP	Not clearly defined	Number of respondents
Ericsson	-	-	-	-	-
Mogul	3	1	X	3	7
EDB Business Consulting	3	3	X	0	6
Skattedirektoratet	4	4	X	16	24
	10	8	0	19	37

Table 8.5: Results from question C5 answered by three of the companies

Company	Very high	High	Medium	Little	No	Don't know	Number of respondents
Ericsson	-	-	-	-	-	-	-
Mogul	0	1	1	5	0	X	7
EDB Business Consulting	1	4	3	2	0	X	10
Skattedirektoratet	2	6	7	7	1	2	25
	3	11	11	14	1	2	42

Table 8.6: Results from question C9 answered by by three of the companies

Company	Yes	No	Number of respondents
Ericsson	4	4	8
Mogul	2	4	6
EDB Business Consulting	5	4	9
Skattedirektoratet	16	5	21
	27	17	44

Table 8.7: Results from question R1 answered by all four companies

R3: Are requirements often changes/renegotiated during a development project? Table 8.8 shows hot the results were spread. We see that 6 respondents did not answer this question.

Company	Often	Sometimes	Seldom	Number of respondents
Ericsson	6	2	0	8
Mogul	3	4	0	7
EDB Business Consulting	3	6	0	9
Skattedirektoratet	9	10	2	21
	21	22	2	45

Table 8.8: Results from question R3 answered by all four companies

Part V

Discussion

Chapter 9

Current Level of Reuse within the Selected GLD Systems (T2)

In this chapter we discuss the results from the work associated with T2, the investigation of current level of reuse within the selected GLD systems, which aims at answering RQ1. We used three approaches in order to answer this question; system documentation review, analysis of source code, and the Reuse Maturity Model. The first covers black-box reuse between applications within all GLD systems, while the second approach measures identical lines of code (white-box reuse) in a selection of programs from three of the selected GLD systems. The last of the approaches measures the reuse maturity level according to the Koltun and Hudson Reuse Maturity Model. We end the chapter with a discussion on the limitations of the approaches we used.

In section 7.1 we reviewed system documentation for all GLD systems. In section 2.3.1 on page 15 we mention several types of reuse, among them the reuse of software components. Our motivation for executing the system documentation review was to find if programs were reused across different applications, as reusable components. As seen in section 7.1.1 on page 71, only four programs for the GA/LTO system were reused in this manner. For the remaining 14 GLD systems, one program appeared in two applications. We find it interesting that one GLD system had four programs which were used in several applications, while only one turned up in the remaining 14. When we first selected which GLD systems we would investigate closer, we were told that GA/LTO was very different from the other systems; it has a completely different structure, and it is of newer date (Appendix D). Also, GA/LTO has a different system owner than the other GLD systems. We believe this is the reason for the unequal division of reused components between GA/LTO and the other GLD systems. The system documentation describes a rather general purpose for the four reused programs; adding name- and address- information, distribution of salary and deduction statements, retrieving statements from the database, and print. The documentation for the other GLD systems shows that they also have several programs with general purposes, but here each GLD system has its own, modified program for this. In order to reuse a program across different applications, or between GLD systems, the developer must use the program as-is. Even though she has complete access to the programs internals (white-box), modifying the source code might unwillingly affect other programs. This description corresponds to the description of "glass-box" reuse as shown in table 2.2; the developer can view the source code but not modify it.

In section 7.2 we measured the amount of source code reuse from 18 programs. With this approach, as expressed in 5.2, we investigated two forms of reuse. First we analyzed the source code for similarities between programs from different GLD systems in order to find indications of common functionality. Our results, as seen in figure 7.2 on page 73, shows that the median amount of common source code lines was 60%. This indicates that the compared programs has a large multitude of common functionallity, and that these common functionalities could be a subject for reuse. Secondly, we investigateted the similarities in the source code between the annual versions of the programs. We found this analysis very interesting since it showed that up to 99% of the code is common between annual versions. The differences we found was exclusively references to year, database tables etc. Based on our meetings with some of the developers at SKD, we expected these programs to have very few differences. They estimated that approximately 70-80% of the code was identical between the annual versions (see Appendix C).

The approaches for developing the GLD systems can be described as code/design scavenging (2.3.2, p.18). The scavenging technique allows the developers of the GLD systems to decrease both time and keystrokes needed to create a program, if the alternative would be to create each annual version from scratch. One of the problems associated with code and design scavenging is that the developer must know were to find the specific code fragment, and valuable time can be wasted in the search process. This problem does not seem to affect the developers of the GLD systems, since an annual program is basically a "clone" of the previous version, and thus the source is given by the programs name. With the few changes performed each year we made the following assumption; There is a large, stable core in the programs, and bugs and flaws, if any, have been identified and removed several years ago. Altough we think it is plausible, we are not able to confirm this in our current research, but this could become a subject for later studies.

We decided that we would asses the current level of reuse within the selected GLD systems against the Reuse Maturity Model by Koltun and Hudson[17]. This model, as shown in figure 2.3 on page 25, contains five phases of maturity. In order to provide an objective and accurate identification of the precise maturity level, we adopted an approach by Morillo et al.[33]. Although this approach was barely covered in literature in the field of software reuse, we selected it because it provided us with metrics and the ability to calculate the reuse maturity level according to specific criteria. We had no presumptions of which level of reuse to expect; the development at SKD was not supported by reuse technologies such as a software repository or by the development process, but the amount of reused code was considerably high between the annual versions. Our investigation concluded that the current level of reuse within the selected GLD systems is level B, Monitored, in the Koltun and Hudson Reuse Maturity Model. This is the first reuse maturity level assessment conducted for the GLD system, and probably within SKD as well, and the maturity level we identified corresponds with other case studies; Frakes[17] states that most organizations are between the Initial/Chaotic and Monitored phases at the time a reuse program is initiated.

9.1 Limitations of Approaches

The system documentation delivered to us by SKD was supposed to contain all programs in the GLD systems. After performing the review of the documentation and cross-checking of programs and applications, we presented the results to SKD (Appendix E). During this

presentation, one of the developers realized that the system documentation was imperfect, and that there were several programs missing. The developer mentioned some of these, which are listed in table E.1 on page 138.

We also discovered that the documentation did not show possible connections between the different programs, so based on the documentation alone it was impossible to determine the amount of reused components. We could only determine the amount of components reused across different applications, not components reused within the same application. Thus, we chose to do this on an application level, although this does not provide the accuracy we initially wanted. With the lack of complete system documentation and call-graphs, the alternative approach for determining reused components would be to analyze all the source code of all programs in all GLD systems, which would most likely be a too comprehensive task for a Masters' thesis. The cross-checking of programs did not provide us with data which could determine the level of software reuse, but guided us in the selection of which programs we would analyze in the source code analysis.

The programs chosen for the source code analysis belonged to the "paper registration" application. When we presented the results from the analysis, we were told that these programs received little attention from SG2, since more and more deliveries are received electronically these days (Appendix F). Also, the programs we analyzed were all CICS programs. Nevertheless, the approach for developing these programs are the same as with all the other programs in the GLD systems, so the results should represent the other CICS programs as well. We expect that annual versions of Batch-programs have similar reuse rates, but this is purely an assumption. It is difficult to say if our findings from the source code analysis would have been different if we would have analyzed programs from other GLD systems. The validity of our results can be increased by analyzing a larger subset of the GLD systems. The differential tool we used detected only changes in lines and words. With our limited knowledge about the COBOL-language, we were only able to spot changes such as references to year, tables, pointers etc. We showed that the median amount of similarities between two programs from different GLD systems was 60%, but we cannot provide our readers with information on what kind of functionality that differs between the programs.

When we initially started to use the Reuse Maturity Model, we compared the development of the GLD systems against each of the ten corresponding dimensions. The different dimensions can each be in a different phase of maturity, which made it difficult to determine and communicate *one* appropriate phase for the GLD systems. With the paper by Morillo et al. [33] we were able to quantify the different aspects involved in reuse, and we could calculate and locate the current phase in the Reuse Maturity Model. However, a considerable limitation of this approach is that we could not find it in any other literature, and as far as we know it has not been published in any journals (but, the authors have written several other, published articles on software engineering). We became in doubt whether if we should use the approach or not. Our final decision, after inquiring us with our supervisor, was to use the paper.

Chapter 10

Emphasis on Software Reuse in SKD's Development Process (T3)

What is the emphasis on software reuse in the current development process? Our motivation was to find out how well SKD had prepared the organization towards software reuse. Four different approaches was used in order to answer RQ3:

- A survey among 25 developers at SKD
- Meetings with the developers in SG2
- A review of the Framework for System Maintenance
- Interview about SKD's Framework for System Maintenance

This chapter will first go through some of the findings from the survey and its limitations. Then a discussion of the survey in relation to the previous surveys at Ericsson, Mogul and EDB Business consulting is presented, which has no relevance to RQ3, but is presented under theme T3 since it is one of our contributions from the survey. Then a short description of the interview about SKD's Framework for System Maintenance follows, before a discussion of RQ3 which will draw the conclusion of that SKD has not prepared the organization for reuse in their current development process.

10.1 Findings from the Survey at SKD

This section will give a summary of the findings from the General questions G1 to G3 of the survey, go into detail to component question C2 and C9, discuss the surveys validity, and finally compare our results with those the surveys at Ericsson, Mogul, EDB Business Consulting and Statoil.

It seems to be a positive attitude toward software reuse and the developers found reuse to be important for achieving different benefits. Figure 8.3, 8.4 and 8.5 in chapter 8 shows the results of the three categories of general questions. Most of the respondents find reuse to be important for achieving different benefits such as lower development costs, shorter development time, higher product quality, standardized architecture and lower maintenance costs 8.3. For G2 we see that the respondents answers are typically in the range from medium to high. All 25 respondents agree on that testing is very important.

OO technologies and configuration management seemed to be lesser important than the rest as seen in figure 8.4. For question G3 we see that the respondents answers are also typically in the range from medium to high. Several respondents answered don't know on the question involving use cased and requirements. Most of the respondents feel that there should be more reuse, and they have split opinions about whether the process of finding, assessing and reusing is functioning or not. The majority of respondents answered that it is not clearly defined how one should decide to reuse a code or design component. When we asked to what extent they feel affected by reuse in their work, the answers typically ranged from little too high.

10.1.1 Limitations of Survey at SKD

The different validity threats were discussed in section 2.8.3 on page 34.

Internal Validity

The participants' previous knowledge and experience on some approaches to software development can have impact on their answers for questions C1-C3.

External Validity

Since we only had 25 respondents, we do not think that the results are generalizable. But if we add the findings from SKD with those from the surveys at Ericsson, Mogul and EDB Business Consulting we have 51 answers from four companies, we increase our chances of a representable population of the Norwegian IT industry.

Construct Validity

In the survey different concepts such as component, architecture and etc. were used. These lack a clear and proper definition, and may therefore be interpreted differently by the respondents. Also, subjects have different perceptions of the scale used in the answer alternatives in the questionnaire. These may be interpreted differently by the various respondents. A person answering "very high" may be equal to another person's perception of "high". As mentioned in section 8.7 on page 85, several questions were not relevant for all the respondents and should therefore have offered an "don't know" answer option.

When we started to analyze the questions, we discovered that the Component Questions were not relevant for all the respondents. Unfortunately we had forgot to provide question C1, C2, C3a, C4, C5, C6, C7 and C8 with an "I don't know" answer option. Ideally, the respondents should have skipped the questions if they did not know how to answer them, but this was not the case. We take self-criticism for this, and we should have detected this when we decided to use an already existing questionnaire. To make up for this, we decided to make cross tabulation analysis of the questions C1 to C8, against C9. The answer alternatives to question C9 was provided with an "I don't know" answer option, and we thought that this might be a good starting point to eliminate mirepresentative answers. In our opinion respondents who answered that they were little or not affected by reuse in their work, or did not know, may not always be capable of answering questions

involving reuse. This involved 10 of the 25 respondents. For this reason, we removed these 10 from the result presented in section 8.7.

Participants in a survey may have certain expectancies to what the results of the survey should be, which may lead to respondents misrepresenting the answers by trying to look better or worse, or guessing the answers. There can be several reasons why a respondent may give faulty answers; misinterprets the question, does not know the answer but answers anyway, thinks he knows the answer but answers incorrect, or is lying.

10.2 Survey Discussed in Relation to Previous Surveys at NTNU

Our results corresponds to the results of the previous studies at Ericsson, Mogul, EDB Business Consulting and Statoil. Reuse benefits from developers view include lower costs, shorter development time, higher product quality of reusable components and a standardized architecture, which is supported by the findings from Statoil and in the literature. The results also shows that the developers at Ericsson, Mogul, EDB Business Consulting and SKD are positive, but not strongly positive to the value of component repository. The respondents also seems to agree on that the design/code components is not well documented, and that this could sometimes be a problem.

10.2.1 Results from Survey at SKD combined with Previous Studies

When we add our results to the previous three studies, the respondents from SKD pull up the average on all questions. The population get a skewed distribution since there are two-three times as many respondents from SKD, as from the other companies. The results are therefore not generalizable, and the surveys are still a prestudy.

Figure 10.1, 10.2 and 10.1 shows the results of the general questions, from all four surveys. For general questions G1 we see that the answers are typically in the range of medium to very high. The respondents believes that reuse are important in achieving all the different benefits. Lower development costs, shorter development time and a more standardized architecture is distinguished as the most important benefits10.1.

From figure 10.2 that testing and reuse/component based technologies is clearly seen as most useful and important to the respondents.

From figure 10.1 wee see that the respondents find most of the artifacts useful and important with respect to reuse. The majority of the answers range from very high to medium importance.

When we disregard the fact that several of the questions were missing a "don't know" answer option (this concerns the previous surveys too), we see that most of the respondents feels that the process of finding, assessing and reusing existing code/design is functioning. The existing code/design documentation is sometimes sufficiently documented, and this is clearly a problem. Twice as many respondents believes that the construction of a reuse repository would be worthwhile. When it comes to deciding how to reuse a component as-is, with modifications or make a new component from scratch, most of the developers from Mogul, EDB Business Consulting and SKD says that it is not clearly defined, where

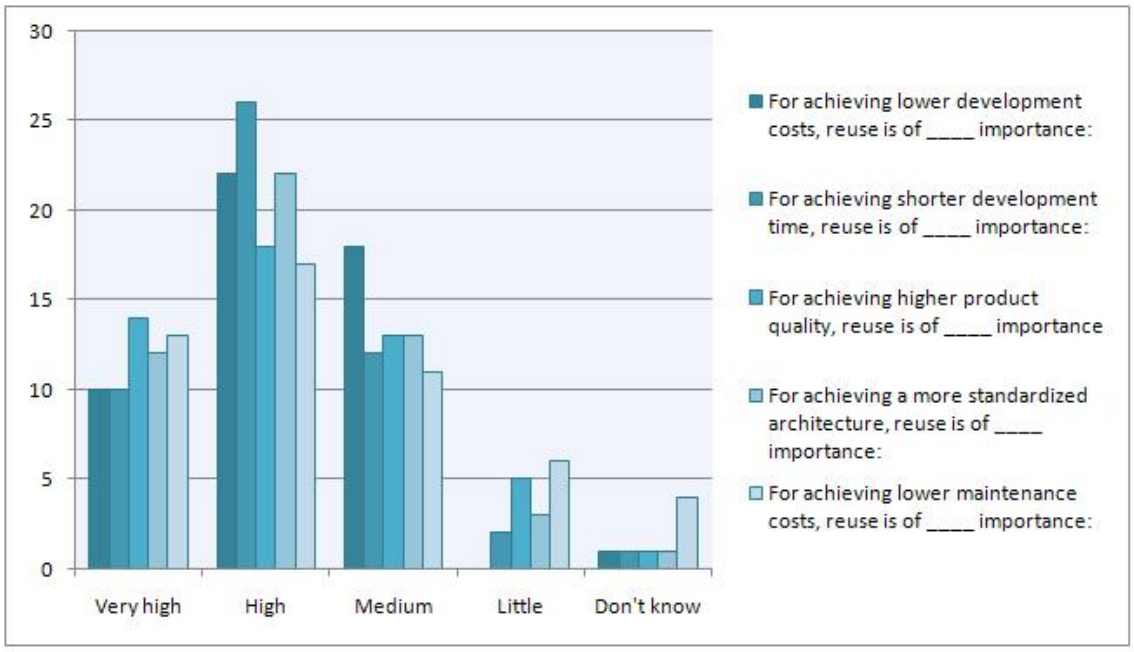

Figure 10.1: General questions G1a-d(n=51 respondents). Columns are in the same sequence as in the description field

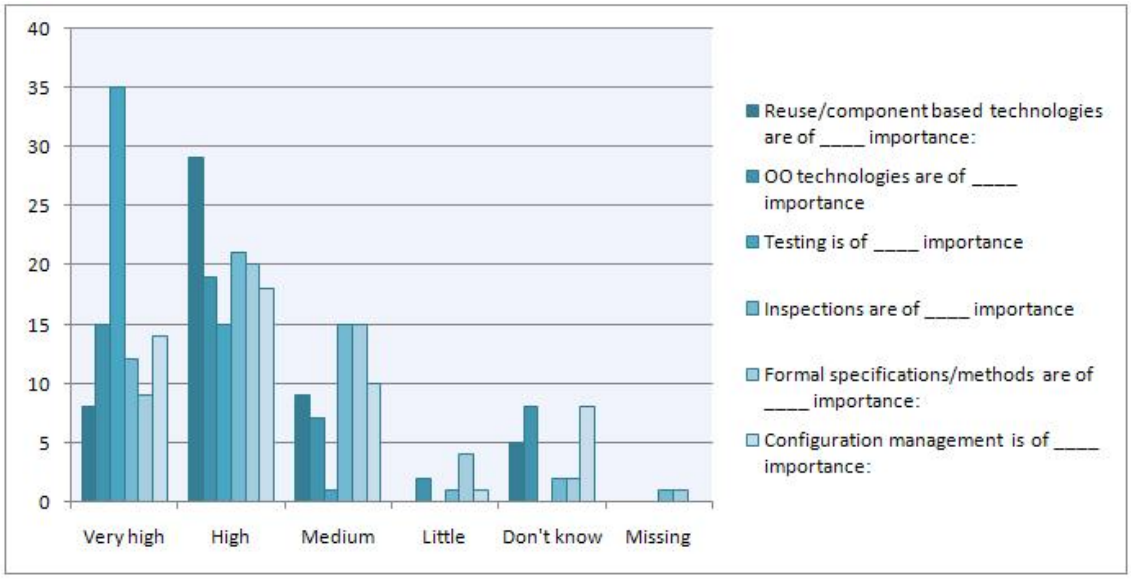

Figure 10.2: General questions G2a-f(n=51 respondents). Columns are in the same sequence as in the description field

the rest is divided between consulting experts or following guidelines. The majority of respondents from Mogul, EDB Business Consulting and SKD feels little affected by reuse in their work. Almost as many respondents are divided between medium to high in this question. The organizations requirement negotiation process is working sufficiently for most of the respondents, and the requirements is often or sometimes changed/renegotiated during a development project.

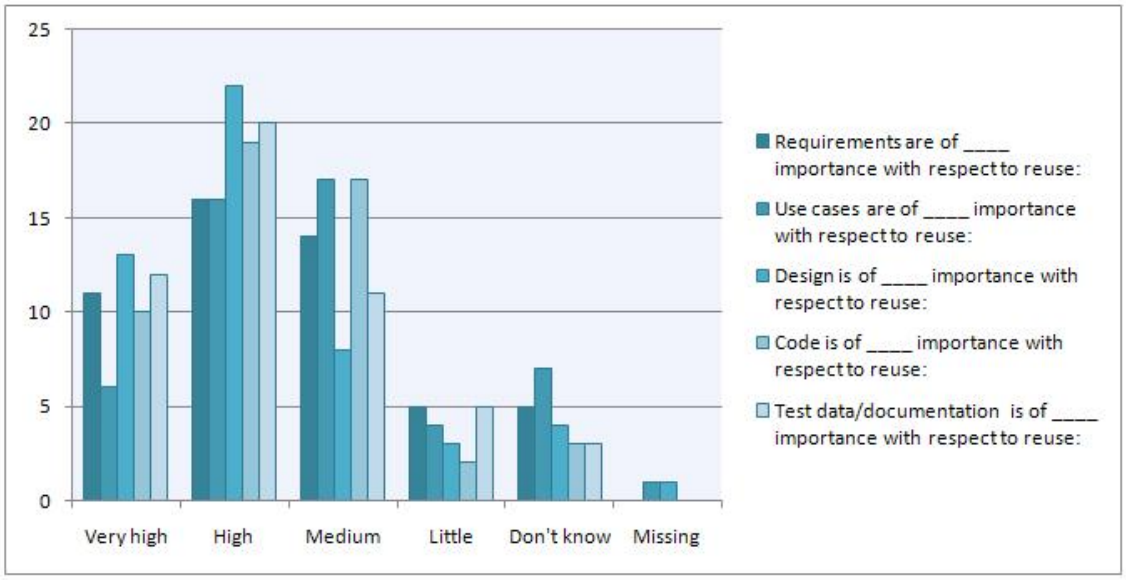

Figure 10.3: General questions G3a-e(n=51 respondents). Columns are in the same sequence as in the description field

10.3 Interview about SKD's Framework for System Maintenance

We interviewed the person in charge of SKD's framework for software maintenance. A resume of the interview can be found in Appendix G. Because we had read and performed a textual search of the framework, before planning the questions, we were sure that the framework did not take software reuse into account.

As mentioned in section 6.2 on page 68, the questions was sent to the interviewee in advance, so she would have time to prepare herself. During the interview an extra person from the "Architecture" group joined her to aid in the questions around SOA.

We were told that SKD had no software maintenance process before the spring 2005, and it was challenging to get people to think new thoughts and change their attitude on how they would develop software. A good deal of information and working methods exist only in individuals minds, not as written knowledge, and SKD is very depended upon specific persons. It also came up during the interview that the process is not at all adjusted for reusing artifacts. This was confirmed by our review of their framework, were we found no indications of software reuse. We asked if the process could be supplemented with routines and templates for software reuse, and were told that it is difficult to know what level of detail such templates should have. Routines and templates would have to be especially adjusted to the different types of systems, because different systems have different requirements.

10.3.1 Limitations of the Interview

Even though the interviewee did not understand all the questions, we felt that our initial assumption were confirmed. She had considerable knowledge about the framework, and

because we had gone through the framework ourselves, we are sure of that the framework does not take reuse into account. Some of the questions was not answered at all, because the interviewee had little experience with software reuse.

10.4 Discussion of RQ3

So back to RQ3, *what is the emphasis on software reuse in the current development process*? As pointed out in the previous section, we have confirmed that the Framework for System Maintenance does not take reuse into account. This is supported by findings from both the survey and meetings.

In the survey, the respondents were able to provide comments to each category of questions. A couple of the developers made the following remarks in our survey:

- "In practice, reuse is not achieved very often"
- "I don't think we have the opportunity to reuse components here"
- "Reuse has no agenda in my daily work WHATSOEVER! It's not prioritized by the 'bosses'"
- "I have also, as the years go by, made myself a little 'library' of code I think may be useful in the future"

These comments supports our assumption on that developers are not encouraged to reuse software artifacts, and that there are no documented routines for how to perform software reuse. In an organization, individual developers can do little about software reuse on their own. Managers must institute the mechanisms needed, and provide organizational support and money to finance them, as described in section 2.2.3.

In an conversation with two of the developers (see Appendix F) we asked them how they got information about reusable components during implementation. The developers told us that they use their own experience, consult other developers or system experts and randomly search for code. This indicated that it is up to each individual developer how they carry out with software reuse. When asked if there were any documented routines on how newly-hired employees could learn the reuse practices performed within the group, we were told that there were no such documented routines. When you have worked on a project for some time, you will learn the routines and they will eventually get institutionalized.

Our conclusion on RQ3 is that SKD has not prepared the organization for reuse in their current development process.

Chapter 11

Opportunities for Systematic Reuse in SKD (T4)

Earlier, we argued that the current state of reuse within the selected GLD systems is "Level B, Monitored" according to the Reuse Maturity Model, and we found that SKD had not prepared the organization for software reuse. This chapter will focus on the future of SKD and the GLD systems; what is the potential for systematic reuse, and how can it be achieved? We will discuss the potential for systematic reuse based on our literature study in chapter 2. To answer how it can be achieved, we focus on changes that are required in SKD's Framework for System Maintenance. Other aspects that must be changed or added to the development process and organization structure will also be suggested, based on the literature study in chapter 2. In the remainder of this chapter, we present three alternatives for how the existing GLD systems can be reengineered.

11.1 What is the Potential for Systematic Reuse?

The results from RQ1, as seen in chapter 9, concluded that the current level of software reuse within the selected GLD systems is "Level B, Monitored". The conclusion from RQ3 (chapter 10) was that the Framework for Software Maintenance does not emphasize software reuse. This indicates that reuse is not institutionalized into the current development process, and this is therefore an ad-hoc approach towards software reuse. As we discussed in section 2.2.1 on page 13, the ad-hoc approach is an informal process, where no methods for reuse are defined.

Because the various GLD systems had common features, the developers found a chance to save development time and efforts by copying and pasting the existing source code. According to the literature[34] ad-hoc reuse often happens by chance and individuals developers are responsible for identifying and locating reusable components. Copying and pasting code snippets from existing programs into new ones may work fine for a while for individual developers or small groups. However, it does not scale up across business units or enterprises to provide systematic reuse[51]. The productivity gained by copy/paste is only marginal, because it makes the maintenance of the software system more time consuming and complex.

In section 2.2.2 we mentioned some benefits with software reuse, found from literature.

However, it is difficult to achieve the desired benefits without a systematic approach to software reuse[3]. A summary of the benefits is given in the list below:

- Improved quality
- Increased productivity
- Increased interoperability between systems
- Reduced costs

The GLD systems have only experienced minor changes since they were initially developed. The quality of the systems can be debated; the systems performs the activities they are created for, and their defect rate is low. On the other hand, the amount of redundant code and programs that performs basically the same tasks is fairly high.

As mentioned in section 2.6.5, development time is improved when using the copy/paste approach. However, the approach also introduces significant maintenance problems as more products are developed. Multiple copies of the software, each slightly different, has to be managed. The source code must be adapted and modified each time it is reused, and by doing so, new errors may be introduced[50]. Thus, the code must be tested as thoroughly as the first time it was created. Also, defects found in one copy must be found and fixed several times.

In section 3.3 on page 40, we described SKD's new IT-strategy[66] and their need for interaction and communication between the different IT-systems. SKD's software systems should be designed so that the systems can communicate both with systems within SKD, as well as with systems in the public sector. The new IT-strategy for SKD also encourages reuse and development on existing components instead of development from scratch or acquisition of new solutions. The current solution deployed in the GLD systems does not support this desired goal. Software reuse can contribute to the interoperability between software systems if several systems have the same components for the interfaces[50]. Service Oriented Architecture (SOA) has in the recent years become a popular way of increasing legacy systems interoperability and reuseability.

Our investigation has not focused on development costs and resources within SKD or the GLD systems. Due to this, we have not given any estimations on how much time, costs, and effort spend can be reduced. However, we did find that SG2 is concerned about the lack of resources available[60] (this was also mentioned during a meeting with SKD, see Appendix F). More resources is required to handle the increased changes and system initiatives[60]. Development can be conducted by smaller teams with systematic software reuse[50]. Software reuse can reduce the amount of new code and the maintenance of the various GLD systems.

To summarize, we see that the introduction of systematic software reuse in the GLD systems development has potential to improve the overall quality, productivity, interoperability between systems, and reduce costs. The next section will address how this can be achieved.

11.2 How can SKD achieve Systematic Reuse?

In section 2.6.2, we argued that processes varies from both organization to organization, and between projects. One process alone can rarely fulfill all the different requirements from organizations and projects who use them. Recall from RQ3 in chapter 10, we discussed our interview about SKD's Framework for System Maintenance. The interviewee said that their process is of a higher level and meant to capture the overall requirements from different projects within SKD, but it does not specify individual or project specific needs. We will therefore propose a method for how SKD can initiate systematic software reuse. We have followed the six key points for a successful reuse program by Morisio et al.[34], and from Jacobson et al.[20] we used the four concurrent processes, organizational structure, and the ten principles for software reuse.

In order to make systematic software reuse an integrated part of the existing Framework for System Maintenance, we suggest a guideline that should be included into the process. Figure 3.4 on page 41 shows the phases of the Framework for System Maintenance, and that guidelines are available in all phases in the lifecycle. A guideline should include how to plan, specify, model, design, implement and document components and applications in the problem domain.

11.2.1 Management Commitment

In SKD's proposal for its IT-strategy for 2007-2009 [66], they state that software reuse and development on existing solutions should be considered before new development or new acquisition. New components should be create as generic reusable components. With this statement from SKD, we believe that it is fair to assume that a software reuse program would have top management commitment. Management support and commitment are of high importance in a successful reuse program[34].

11.2.2 Changes in Organizational Structure

We recommend that SKD establish a group of *Creators*, who is responsible for developing reusable components and domain engineering. This group would preferably be located under the *Information technology* branch as seen in the organizational chart of SKD's IT-department (figure 11.1). The users of the reusable assets would include the developers of the systems under *Information systems*. The creators needs to be close to the reusers, since they must design and create components that are both useful and feasible.

11.2.3 Changes in Development Process

Jacobson et al.[20] suggested four concurrent processes for software reuse, as described in section 2.6.2, These processes are called create, support, reuse and manage. The *Create process* is performed by the *Creators of reusable assets* as seen in figure 2.3. They are responsible for activities such as inventory and analysis of existing application and assets, domain analysis and engineering, definition of architecture, analysis of reuser needs, technology evolution and testing of reusable assets.

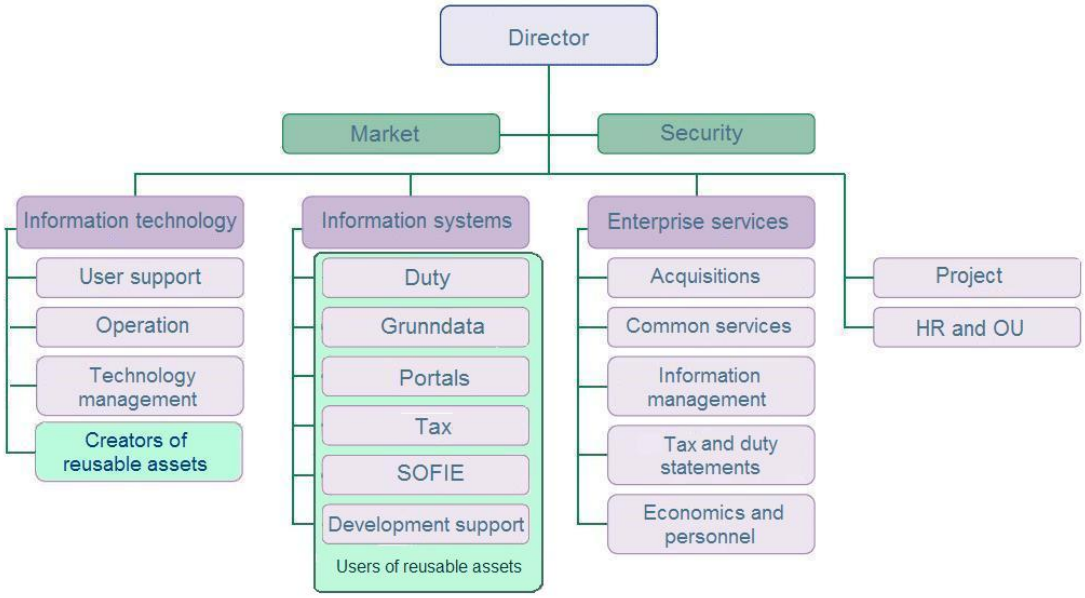

Table 11.1: IT-department organizational chart extended to incorporate software reuse

The *Reuse process* is performed by the systems under the *Information systems* branch in figure 11.1. The reusers are involved in application engineering, and makes use of the reusable assets to produce applications or products. Before developing new code and components for a specific purpose, the developer searches the repository to check if the particular solution already exists. If the solution exists, the developer can either use it as-is or modify it. The reusers are also responsible for activities such as examination of domain models and reusable assets, collection and analysis of end-user needs, design and implementation of additional components, adaption of provided assets, and the construction and testing of complete applications[20].

The *Support function* should help the overall set of processes, and manage and maintain the reusable assets collection. The process may include activities such as classification and indexing of reusable assets in a library or software repository, announcement of the distribution of the assets, provide additional documentation, and also collect feedback and defect reports from reusers. The repository should be managed at organization or enterprise level, since it is an activity that spans across different projects and application systems. Since we are interested in introducing as few changes as possible at a time, we suggest that the support function is performed by the *Creators of the reusable assets*, at least in the beginning. When the reusable software repository have grown considerably, and is used by a number of different systems, this process can be extracted to an own group that is responsible for these activities.

The *Manage process* acts as a superior process, which tracks and coordinate the other processes. Activities includes setting priorities and schedules for new asset construction, analyzing the impact and resolving conflicts concerned alternative solutions when required assets is not available, and establishing a training program. The manage process also have to mediate conflicting interests of both creators and reusers. We believe that this process

can be performed by a subgroup within the *Technology management* group, under the *Information technology* branch.

A software reuse program can only succeed if SKD can prove it to be economical beneficial, as discussed in section 2.4.2 on page 21. It is important to measure the reuse progress with metrics if they are to manage the reuse program, justify the investments and optimize it. The measurement and monitoring of reusable assets and reuse program should be performed by the *Management process*, since it is important that measure and monitoring is performed at the organizational level to ensure reuse.

11.2.4 Training and the use of Champions

Training is one of the most important activities in implementing systematic reuse. Not only does it increase the knowledge about reuse, but it is also a key for gaining acceptance for reuse. This is especially important for SKD, since systematic reuse is quite different from the way reuse is currently performed at SKD, namely ad-hoc. It is important that SKD encourage developers to develop systems with reusable components, and that new components is design with reusability in mind. The introduction of awards or incentives is reported to have a positive effect in other organizations[21][48], and should be introduced at SKD. Incentives helps as a motivation and encourages developers to change their way of developing software. The use of champions, both for the creators and reusers, should encourage to further software reuse within SKD. Champions can act as driving forces behind change processes, and should be people with a strong engineering background that have trust from their fellow workers and are enthusiastic about software reuse.

11.2.5 Pilot Project

The reuse program should be implemented incrementally. By a series of pilot project, taking one system at a time, the program can evolve in smaller increments. SKD should introduce as few changes as possible at a time, and build upon existing knowledge, skills, and tools in the organization. This has several advantages; it helps them build up confidence in the organization and provides immediate returns on investments.

If SKD for example would like to take the GLD systems as a starting point for implementing systematic reuse and the building of reusable assets, it would require substantial efforts and investments. It is important that SKD have a business strategy that looks beyond current projects, and invests in assets that future project can take advantage of. As discussed in section 2.3.5 on page 20, domain engineering is a key concept in systematic reuse, that can help identify potentially reusable assets and an architecture that enables reuse. Domain analysis can be performed in order to locate and record commonalities and variability in the GLD systems. Then, domain implementation can be performed in order to create the reusable assets and new systems. These processes are performed by the *Creators of reusable assets*.

Within the GLD systems it exist potential for vertical, internal and external reuse, meaning that reusable software components can be reused within a single GLD system (for example GA/LTO) or by the different GLD systems. Vertical reuse is harder than horizontal reuse to achieve, but offer greater potential when it comes to the benefits of software reuse.

Horizontal reuse refers to when software components is used across a wide variety of application areas. This can be both COTS components, library components, software drivers and graphical user interface functions. We anticipate that these types of components can be used by other systems outside the GLD systems if they are generic or have a common interface.

Since we only have knowledge about the GLD systems, we are not sure if these are a suitable place to start from. This decision must be made by SKD. Regardless of the system they start with, we assume that great efforts must be made in domain engineering and reengineering of the systems in order to build reusable assets and an architecture that enables reuse. When the pilot project is finished, a new pilot can be initiated. This time a new representable system should go through the same process as the former, only this time SKD has gained more experience, skills and confidence. The new pilot might also be able to use the reusable assets constructed in the previous pilot. It is also fair to assume that processes, guidelines, tools and training is customized gradually as SKD get more experience. As previously mentioned, the entire process must be monitored and measured with metrics in order to manage their reuse progress.

SKD's payoff will eventually come when several applications are build more cost-efficiently and more rapidly using reusable components. This payoff builds up over time, as the reusable components are reused repeatedly and more reusable components are developed.

11.3 Rewriting of Existing GLD Systems, Three Alternative Approaches

In our source code analysis of the GLD systems, we discovered that the annual versions of the programs have between 97% and 99% of unchanged code. This raises the question about how necessary these annual versions really are. Most of the annual systems are required by law to be available for up to 10 years. With 15 different GLD systems, this means that up to 150 systems must be kept alive or stored on tape. As mentioned earlier, the reason for these annual versions relates back to the days where the storage capacity and speed of hard drives was inadequate for storing the data needed for all GLD systems, and for several years. The solution for this was to let each GLD system have its own database, and creating a new GLD and database each year.

In this section we will discuss different approaches to how the GLD systems could be made more reusable while still running on the existing mainframe. The approaches presented here does not consider development of the GLD systems from scratch, but only modifications to the existing systems. The "X-version approach" is specific for the GLD systems, while the two other suggestions could be applied to legacy systems and COBOL programs in general. However, the rewriting of such systems is not to be underestimated, and substantial work and effort is required when considering these approaches.

11.3.1 X-version Approach

The "X version" approach came up during a meeting with one of the developers at SKD (see Appendix F). The programs are created without any reference to which year of the databases they are communicating with. The same program can be used year after year,

and only requires modifications if it is required. Instead, the programs are using database views which can be replaced each year. The view directs the application to the correct year.

This approach has the advantage that the programs would not have to be copied, pasted and modified annually, thus reducing the number of systems which needs to be maintained. This approach is only partial beneficial, since views and databases still needs to be created as they did before, and it does not solve the problem of code redundancies between the different programs.

11.3.2 Separating common Functionality in Separate Modules

The GLD systems had an high percentage of common code lines between the annual versions, but their was also a great amount of commonalities between the different GLD systems. Our code analysis showed that these numbers varied from 36% to 78%. This indicates that there are common functionality between the different systems, and these functionalities are potentially reusable assets. These reusable assets should be extracted and placed into generic components, and made available for other modules. The variation-points are extracted to smaller, specialized components. In object-oriented programming this is recognized as functions in separated classes. We briefly mentioned the concept of a "copybook" earlier. A copybook in COBOL is a file which contains code and can be shared among several programs. By placing the common code in copybooks, the redundancies in the GLD systems would be reduced.

11.3.3 Restructuring and Software Architecture

Our source code analysis revealed that there is a lack of a visible and uniting architecture in the GLD systems. All CICS applications have their "sort of three layer" architecture, but we have not been able to find an higher and superior form of architecture which unifies and structures all the programs. In order to achieve a higher level of reuse, a software architecture is required[20]. And if a software architecture is to be created and deployed, the GLD systems would need severe restructuring. In the remainder of this section we discuss a technique for restructuring COBOL/CICS applications. We must emphasize that we have limited knowledge in this field, and we can not guarantee that this concept will work in the context of the GLD systems.

The CICS applications are written in mixed languages; COBOL and CICS. Restructuring COBOL/CICS applications are far from trivial, and certain problematic CICS constructs should be removed[2]. The CICS HANDLE commands are such constructs. The HANDLE CONDITION, HANDLE ABEND and HANDLE AID has been proved to have counter intuitive behavior under certain conditions, and these should be removed in order to increase the maintainability and reusability of the system. For exception handling, these statements should be replaced with return codes used by CICS. For evaluating input from the user, the HANDLE should be replaced with the COBOL EVALUATE statement.

Sellink et al.[2] also states that the GO TO and GO TO DEPENDING commands should be removed because these commands makes the program unstructured. The GO TO should be replaced by the PERFORM command, and all statements from the point of entry to to where the control is passed back to the teleprocessing monitor should be performed within

the EVALUATE section. Figure 11.1 shows a simple example of how the HANDLE AID commands could be removed.

```
*-------------------*
 CICS-HJELP.
*-------------------*

     EXEC CICS
        HANDLE AID
           ENTER
           PF2     (HENT-PARAMETER-FRA-GX970)
           PF3     (RETUR-HOVEDMENY)
           PF4     (RETUR-KONTROLL-GX05)
           PF12    (INITIER-BILDE)
           PF9     (AVSLUTT)
           ANYKEY (FEILPF)
     END-EXEC

     EXEC CICS
        HANDLE CONDITION
           ERROR (FEIL)
     END-EXEC.

     GO TO RETUR-NAVN-ADR.
```

```
EVALUATE TRUE
   WHEN X-MAPFAIL
        PERFORM FEIL
   WHEN X-PF2
        PERFORM HENT-PARAMETER-FRA-GX970
   WHEN X-PF3
        PERFORM RETUR-HOVEDMENY
   WHEN X-PF4
        PERFORM RETUR-KONTROL-GX05
   WHEN X-PF12
        PERFORM INITIER-BILDE
   WHEN X-PF9
        PERFORM AVSLUTT
   WHEN X-ANYKEY
        PERFORM FEILPF
END EVALUATE
```

Figure 11.1: Simple example of restructured GLD program

After performing the actions mentioned so far, the program should be repartitioned[2]. Subroutines in SECTION levels should be extracted and placed as separated files. The subroutines will then exist as subprograms, which has several obvious payoffs:

- The subprograms becomes reusable since they can be accessed and used by other programs.
- Only the CICS commands and control logic remains in the original program, and the business and database logic is extracted.
- The programs becomes smaller in size, thus more maintainable and testable.

For more detailed information on restructuring COBOL/CICS application and support tools, we recommend the article "Restructuring of COBOL/CICS Legacy Systems"[2]. By restructuring the GLD systems, the architecture would no longer be a "sort of", but rather an actual three layer architecture. The modularization of the system supports the idea of a Service Oriented Architecture (SOA).

11.3.4 Issues Related to the proposed Approaches

The annual versions of all GLD systems and databases is still the largest obstacle against achieving a large-scale reuse program. None of the approaches for rewriting the GLD systems solves this particular problem. In one of our meetings, one of the developers stated that it is probably more expensive to manage several databases than to have a single database. Hard drives has decreased in costs, and both capacity and performance has increased since these systems was created in the early 90's. Some modifications would of course be required to the existing tables, among them is the identification of the specific year in each table. All redundancies between the different GLD databases must be identified and replaced with a single table. The developers in SG2 does not agree whether or not it is more feasible with one common database for all the GLD systems, or as the current solution with one database for each GLD system (see Appendix F).

In an internal paper at SKD[61] the group leader for SG2 describe how more modern integration technologies and SOA can be utilized in order to modernize SKD's systems and processes. SOA is a concept of structuring programs in the shape of service. The value

for SKD lays both internal and external to the organization. The internal value is that services can make us of common resources across both mainframe and Unix platforms. The greatest gains is external where SKD's services can be made available for external users and contribute to improved cooperation with citizen and industry. As mentioned earlier, SKD's systems are build on various technical platforms, and with an integration platform these systems could be able to communicate with the help of services. We have not emphasized on Service Oriented Architecture in this thesis since SKD initiated their own project for modernization of the GLD systems. The MAG-project aims at a Service Oriented Architecture for their IT-systems, and as previously mentioned an ideal architecture for the GLD systems has already been proposed. Its still not clear whether the GLD systems will be re-engineered using object-oriented languages, or if they will be customized and wrapped in order to function with the new technology.

Part VI

Conclusion and Further Work

Chapter 12

Conclusion and Further Work

In this chapter we present the conclusion of this thesis, and propose further work.

12.1 Conclusion

The objective of this thesis was to solve four research question and two organization specific goals. We showed that even though there is a considerable amount of reuse within the GLD systems, the extent of this reuse is ad-hoc and performed only on individual basis. Their current phase in the Reuse Maturity Model is *level B, Monitored*, which is common for organizations without a systematic approach to reuse. Our second research question remains inconclusive in this thesis, as we could not answer if reused components are more stable than other components with the available data resources. The currently deployed development process at SKD does not emphasize on software reuse. However, the developers at SKD seems to have a positive attitude toward software reuse, and feels that it is an important approach for lowering development costs, shortening development time, increasing product quality, and lower maintenance costs. Changes must be made to SKD's development process in order to achieve systematic reuse. This requires commitment from the management, and preferably an own unit who is responsible for developing reusable assets. Systematic reuse in the GLD systems could potentially reduce todays maintenance problems, improve the overall quality, increase the productivity and interoperability with other systems.

12.2 Further Work

Our assessments on the current reuse level within the GLD systems can be used as starting point for the introduction of systematic reuse. SKD needs to change their current development process so reusable assets are created and used during development, and they need technology that supports development with reuse. The developers must be encouraged to perform systematic software reuse, and the use of incentives could act as a motivational factor for achieving a higher level of reuse. The guideline we proposed should be tested with pilot-projects in order to gain experience and knowledge, before extending the reuse program to other systems within SKD. When initiating a reuse program, it is important that SKD measures and evaluates the processes and reusable assets frequently. Now that

SKD knows the GLD systems' current level in the Reuse Maturity Model, it should be easier to find where reuse initiatives are required in order to reach a higher phase in the model. Other metrics and models besides the Koltun and Hudson model can also be used.

The source code analysis presented in this thesis shows only the *amount* of similarities between some of the programs. In order to remove the redundancies in the GLD systems, SKD must identify the *content* of the commonalities.

The survey used in this thesis has previously been used by other master thesis' at NTNU. We have contributed with more respondents to NTNU's research on software developers attitude toward software reuse. If later studies include more Norwegian IT-companies, the result could be generalized to concern the Norwegian IT-Industry.

We were not able to answer if reused components are more stable regarding changes and defects than other components, due to the available data resources. The question is nonetheless very interesting, and could be included in case studies were both reused and non-reused components are available, along with statistics on changes and defects.

SKD has initiated a project which focuses on modernization of the GLD systems (the MAG project). When this project becomes well under way, further work could include a second assessment on the state of reuse within the GLD systems. Later studies could also do a closer investigation on the success factors for achieving systematic reuse, such as:

- Which modifications in the development process proved to be beneficial
- The effect of introducing a group of *creators* responsible for creating reusable assets
- The effect of Domain Engineering
- The advantage of incentives for developers
- The advantage of technologies such as a software repository

Bibliography

[1] Silvia Mara Abrah and Antonio Francisco do Prado. Web-enabling legacy systems through software transformations. In *WECWIS '99: Proceedings of the International Workshop on Advance Issues of E-Commerce and Web-Based Information Systems*, page 87, Washington, DC, USA, 1999. IEEE Computer Society.

[2] Harry Sneed Alex Sellink and Chris Verhoef. Restructuring of COBOL/CICS legacy systems. *Software Maintenance and Reengineering, 1999. Proceedings of the Third European Conference on*, pages 72–82, 1999.

[3] Almeida Alvaro and Meira. Towards a software component certification framework. In *In the 7th International Conference on Quality Software (QSIC), Portland, Oregon, USA*, 2007. http://www.ecoop.org/phdoos/ecoop2005phd/EduardoSantanaDeAlmeida.pdf.

[4] Line Kristoffersen Asbjørn Johannesen and Per Arne Tufte. *Forskningsmetode for økonomisk-administrative fag*. Abstrakt forlag as, 2004.

[5] R.D. Banker, R.J. Kauffman, and D. Zweig. Repository evaluation of software reuse. *Software Engineering, IEEE Transactions on*, 19(4):379–389, Apr 1993.

[6] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison Wesley, 2003.

[7] Elizabeth Burd, Malcolm Munro, and Clazien Wezeman. Analysing large cobol programs: the extraction of reusable modules. *icsm*, 00:238, 1996.

[8] Dave Card and Ed Comer. Why do so many reuse programs fail? *Software, IEEE*, 11(5):114–115, Sep 1994.

[9] Reidar Conradi. Process support for reuse. *Software Process Workshop, 1996. Process Support of Software Product Lines., Proceedings of the 10th International*, pages 43–47, 17-19 Jun 1996.

[10] Thomas D. Cook and D. T. Campbell. *Quasi-experimentation: design and analysis issues for field settings*. Boston: Houghton Mifflin, 1979.

[11] Margaret J. Davis. Stars reuse maturity model: Guidelines for reuse strategy formulation. In *Proceedings of the 5th Annual Workshop on Software Reuse*, 1992. ftp://gandalf.umcs.maine.edu/pub/WISR/wisr5/proceedings/ascii/davis_m.ascii.

[12] Ted Davis. Toward a reuse maturity model. In *Proceedings of the 5th Annual Workshop on Software Reuse. University of Maine.*, 1992. http://www.umcs.maine.edu/ larry/latour/WISR/wisr5/proceedings/ascii/davis_t.ascii.

[13] Danielle Fafchamps. Organizational factors and reuse. *IEEE Softw.*, 11(5):31–41, Sep 1994.

[14] William B. Frakes. Systematic software reuse: a paradigm shift. *IEEE*, 1994. http://ieeexplore.ieee.org/iel2/2956/8385/00365817.pdf.

[15] William B. Frakes and Christopher J. Fox. Sixteen questions about software reuse. *Commun. ACM*, 38(6):75–ff., 1995.

[16] William B. Frakes and Sadahiro Isoda. Success factors of systematic reuse. *IEEE Softw.*, 11(5):14–19, September 1994.

[17] William B. Frakes and Carol Terry. Software reuse: metrics and models. *ACM Comput. Surv.*, 28(2):415–435, 1996.

[18] Tore Berg Hansen and Greta Hjertø. *Kvalitet og programvareutvikling.* Tisip og Gyldendal Akademisk, 2003.

[19] Ottar Hellevik. *Forskningsmetode i sosiologi og statsvitenskap.* Oslo: Universitetsforlaget, 2002.

[20] Ivar Jacobson, Martin Griss, and Patrik Jonsson. *Software reuse: architecture, process and organization for business success.* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1997.

[21] Rebecca Joos. Software reuse at motorola. *IEEE Softw.*, 11(5):42–47, Sep 1994.

[22] Philip Koltun. Infrastructure issues for achieving software reuse. In *Fifth Workshop on Institutionalizing Software Reuse, Hewlett-Packard, Palo Alto, California*, October 1992.

[23] Philip Koltun and Anita Hudson. A Reuse Maturity Model. In *Fourth Annual Workshop on Software Reuse*, November 1991.

[24] Det kongelige Fornyings-og administrasjonsdepartementet. Eit informasjonssamfunn for alle, 2006-2007. St.meld.nr.17.

[25] Charles W. Krueger. Software reuse. *ACM Comput. Surv.*, 24(2):131–183, 1992.

[26] Meir M. Lehman and Juan F. Ramil. Rules and tools for software evolution planning and management. *Ann. Softw. Eng.*, 11(1):15–44, 2001.

[27] Wayne C. Lim. Effects of reuse on quality, productivity and economics. *IEEE Software*, 11(5):23 – 30, 1994.

[28] Wayne C. Lim. Why the reuse percent metric should never be used alone. In *Workshop on Institutionalizing Software Reuse (WISR'9)*, January 1999. http://www.umcs.maine.edu/~ftp/wisr/wisr9/final-papers/Lim.html.

[29] Jonathan I. Maletic and Michael L. Collard. Supporting source code difference analysis. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 210–219, Washington, DC, USA, 2004. IEEE Computer Society.

[30] Doug McIlroy. Mass produced software components. *NATO Software Engineering Conference*, pages 138–155, 1968.

[31] Tom Mens. A state-of-the-art survey on software merging. *IEEE Trans. Softw. Eng.*, 28(5), 2002.

[32] Parastoo Mohagheghi and Reidar Conradi. Quality, productivity and economic benefits of software reuse: a review of industrial studies. *Empirical Software Engineering*, 12(5):471–516, 2007.

[33] J. Llorens Morillo, A. Amescua Seco, and V. Martinez Orga. The reuse process and its maturity level in an organization: Rmm, 1997. www.ie.inf.uc3m.es/grupo/Investigacion/LineasInvestigacion/Congresos/RMM97_Docum_Final.doc last visited on 30. May 2008.

[34] Maurizio Morisio, Colin Tully, and Michel Ezran. Diversity in reuse processes. *IEEE Softw.*, 17(4):56–63, 2000.

[35] Maurizio Morisio, Colin Tully, and Michel Ezran. Success and failure factors in software reuse. *IEEE Trans. Softw. Eng.*, 28(4), 2002.

[36] Hausi A. Müller and Mari Georges, editors. *Proceedings of the International Conference on Software Maintenance, ICSM 1994, Victoria, BC, Canada, September 1994*. IEEE Computer Society, 1994.

[37] Erlend Naalsund and Ole Anders Walseth. Decision-making in component-based development. Master's thesis, Norwegian University of Science and Technology, 2002.

[38] Jingyue. Conradi. Mohagheghi. Sæhle. Wang. Naalsund and Walseth. A study of developer attitude to component reuse in three it companies. In *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, pages 538–552. Springer-Verlag Berlin Heidelberg, 2004.

[39] Arbeids og administrasjonsdepartementet. Arkitektur for elektronisk samhandling i offentlig sektor. strategier og tiltak for mer samordning på it-området (forprosjektrapport), 2004.

[40] Wilma M. Osborne and Elliot J. Chikofsky. Guest editors' introduction: Fitting pieces to the maintenance puzzle. *IEEE Softw.*, 7(1):11–12, 1990.

[41] Jeffrey S. Poulin. Measuring reuse. In *5th Annual Workshop on Software Reuse, WISR5*. IBM, 1992. citeseer.ist.psu.edu/poulin92measuring.html.

[42] Jeffrey S. Poulin. Technical opinion: reuse: been there, done that. *Communication of the ACM*, 42(5):98–100, 1999.

[43] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Higher Education, 2001.

[44] Rubén Prieto-Díaz. Making software reuse work: an implementation model. *SIGSOFT Softw. Eng. Notes*, 16(3):61–68, 1991.

[45] Rubén Prieto-Díaz. Systematic reuse: a scientific or an engineering method? *SIGSOFT Softw. Eng. Notes*, 20(SI):9–10, 1995.

[46] Rubén Prieto-Díaz. Status report: software reusability. *Software, IEEE*, 10(3):61–66, May 1993.

[47] Thiagarajan Ravichandran and Marcus A. Rothenberger. Software reuse strategies and component markets. *Commun. ACM*, 46(8):109–114, 2003.

[48] DOD Software reuse initiative falls church va. Software reuse executive primer, 7th edition. http://stinet.dtic.mil/cgi-bin/GetTRDoc?AD=ADA300508&Location=U2&doc=GetTRDoc.pdf.

[49] Slyngstad. Gupta. Conradi. Mohagheghi. Ronneberg and Landre. An empirical study of developers views on software reuse in statoil asa. In *ISESE '06: Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, pages 242–251, New York, NY, USA, 2006. ACM.

[50] Johannes Sametinger. *Software engineering with reusable components*. Springer-Verlag New York, Inc., New York, NY, USA, 1997.

[51] Douglas C. Schmidt. Why software reuse has failed and how to make it work for you. *C++ Report magazine*, January 1999.

[52] Odd Arne Sæhle. Evaluation of software reuse at edb-bc. Master's thesis, Norwegian University of Science and Technology, 2003.

[53] Guttorm Sindre, Reidar Conradi, and Even-Andre Karlsson. The REBOOT approach to software reuse. *The Journal of Systems and Software*, 30(3):201–212, September 1995. citeseer.ist.psu.edu/sindre95reboot.html.

[54] Skattedirektoratet. Strategisk plan for bruk av it i skatteetaten versjon 1.3.

[55] Skattedirektoratet. Veiviser til skatteetaten, 2003.

[56] Skattedirektoratet. Erfaringsrapport for mottaksapparatet 2004. grunnlagsdata fra tredjemann - gld 2003., 2004.

[57] Skattedirektoratet. Rammeverk for skattedirektoratets systemforvaltningsmetode, 2004.

[58] Skattedirektoratet. Rammeverk for skattedirektoratets systemforvaltningsmetode. mars-prosjektet, 2005.

[59] Skattedirektoratet. *Systembeskrivelse Grunnlagsdata 2004*, 2005.

[60] Skattedirektoratet. Erfaringsrapport med forslag til tiltak. grunnlagsdata - glds 2006, 2006.

[61] Skattedirektoratet. S@ts, 2006. Internal paper for the system section at SKD's IT-department.

[62] Skattedirektoratet. *Systembeskrivelse Grunnlagsdata 2006, Vedlegg 1 Programbeskrivelser batch*, 2006.

[63] Skattedirektoratet. *Systembeskrivelse Grunnlagsdata 2006, Vedlegg 2 Programbeskrivelser CICS*, 2006.

[64] Skattedirektoratet. *Systembeskrivelse Grunnlagsdata FLT 2006*, 2006.

[65] Skattedirektoratet. *Systembeskrivelse Grunnlagsdata FLT 2006, Vedlegg 2 - Programbeskrivelser*, 2006.

[66] Skattedirektoratet. Skatteetatens it-strategi 2007-2009, 2007.

[67] Ian Sommerville. *Software engineering*. Addison Wesley, 7 edition, 2001.

[68] SPSS.com. Spss base 14.0 user's guide. http://www.wright.edu/cats/docs/docroom/spss/, 2005. last visited 17.04.2008.

[69] Torstein Talleraas. Arkitekturdokument mag. Technical report, Skattedirektoratet, 2007.

[70] Torstein Talleraas and Knut Botheim. Arkitekturdokument lsa. Technical report, Skattedirektoratet, 2007.

[71] Will Tracz. Software reuse myths. *SIGSOFT Softw. Eng. Notes*, 13(1):17 – 21, 1988.

[72] Jeffrey Voas. Cots software: The economical choice? *IEEE Softw.*, 15(2):16–19, 1998.

[73] Winmerge, an open source visual text file differencing and merging tool for windows (version 2.6.12.0 unicode), 1996-2006. downloaded from http://winmerge.org/ - last visited 01.02.2008.

[74] Myeong-Jae Yi, Jong-Min Park, Jae jung Yang, and Myoung-Joon Lee. Design and implementation of a www-based c source code documentation tool using the re technologies. *Science and Technology, 2001. KORUS '01. Proceedings. The Fifth Russian-Korean International Symposium on*, 1:127–130 vol.1, 26 Jun-3 Jul 2001.

[75] Øyvind Wang. A study of industrial, component-based software engineering, mogul. Master's thesis, Norwegian University of Science and Technology, 2003.

Index

Part VII

Appendices

Appendix A

SEVO - Study of Software Reuse at Sattedirektoratet

The computer systems at Skattedirektoratet (SKD) are mainly based on mainframe (COBOL/ CICS/ DB2) and UNIX (Oracle database, Forms, PL/SQL). It is very likely that SKD will establish a new integration platform (ESB), where reusable services can be made available for both mainframe and UNIX platforms. SKD is interested in an assignment that among other, give answer to the following:

- What is the current state of reuse?

  The study should be based on studies of two systems from each platform. If this is too comprehensive, it is possible to take two systems from the same platform. The study can include an outline over actual reused modules, potentially reusable modules, existing routines that ensures reuse, and knowledge about reuse (to what degree are the employees concerned about reuse, and know about routines and existing modules). Suggest how much of the systems are reused in number of code lines, function points etc.

- Are there differences in reuse (pattern, volume) from mainframe to UNIX?

- Consider benefits and drawbacks with reuse of different types of components, to investigate which components or types of components give the greatest reuse gains. This can be measured after quality factors such as error rate, change rate, architecture standardization etc. The study can also include an estimate of which characteristics give the greatest reusability. Collection of data can happen by the use of data archeology and interview.

- Suggest a number of reusable services for the new integration platform, and estimate cost/benefit with the new integration platform versus keeping the existing architecture. How can SKD make use of OSS/COTS, and which benefits can it provide?

- Suggest improved processes for reuse (technical, project, organization). If possible, a cost/benefit model can be used in the evaluation, to demonstrate concrete effects.

- Look at the effects of outsourcing, mainly with the Altinn system.

The assignment will be further adjusted in dialogue with SKD.

Supervisor: Reidar Conradi, IDI. Co-supervisor: Tore Hovland, SKD

Appendix B

List of GLD systems

These are the 15 GLD systems, presented in Norwegian:

- **GA** - lønns- og trekkoppgaver (også kalt GA/LTO eller FLT-systemet)
- **GB** - Saldo og renter
- **GC** - Beholdning aksjer, obligasjoner og opsjoner fra VPS
- **GD** - Boligselskap
- **GE** - Gaver til forskning og frivillige organisasjoner
- **GF** - Livsforsikring
- **GG** - Skadeforsikring
- **GH** - Beholdning av realisasjoner i aksje- og obligasjonsfond - transaksjoner fra VPS
- **GJ** - Bil og Landbruk(primærnæring)
- **GK** - Barnehager
- **GL** - Boligsameie
- **GM** - Underholdningsbidrag
- **GN** - IPA (Individuell pensjonsavtale)
- **GP** - Avregningsdata
- **GS** - BSU (Boligsparing for ungdom)

Appendix C

Resume of Meeting 25. May 2007, at SKD

We had prepared some questions which we wanted to be answered, before the meeting at SKD. These questions were reviewed by Reidar Conradi, and sent to Tore Hovland in advance of the meeting. It is important to notice that the answers below are just a summary of what was said on the meetings, and not exact quotations. The question was answered by various members of system group 2.

The GLD Systems

1. How many GLD systems are there?

   The number depends on which person is counting. The number usually varies between 14-15 systems. Some people want even more GLD systems, but there are both advantages and disadvantages to this. More GLD systems indicate, among other things, more maintenance.

2. It was mentioned earlier that the first GLD systems were the foundation for new ones. How much of the code is copied and modified to create new GLD systems?

   The earliest GLD systems acted as a base for newer ones. Code was copied and modified to support newer GLD systems need. This indicates that there are about 14 systems that are more or less the same with individual variations, which has to be maintained. In addition to this, there are different versions each year of every GLD system. These annual versions of each system are modified and changed according to new law legislations, faults discovered, new functionality etc. Each GLD system has its own database. This database is also copied each year. About 70-80 percent of the code is the same for the various systems.

   Figure C.1 illustrates how these systems are copied year after year.

3. Are there functions or general code that is the same for all?

   Much of the code is the same, but there are individual modifications to each GLD system. The systems also have different sizes (for example number of programs and code lines).

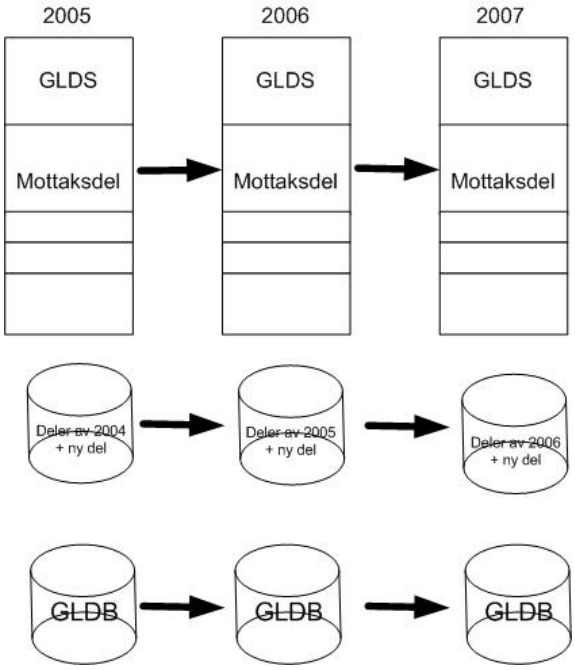

Figure C.1: Annual versions of the systems

4. What are the main challenges attached to this?

   Maintenance; there are several interfaces to deal with. Changes to one system may have consequences on others. Systems and stakeholders have different concerns and needs that must be considered. Figure C.2 shows the different interfaces.

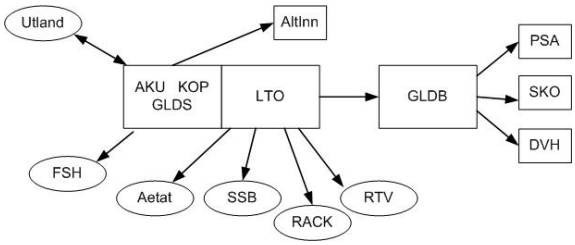

Figure C.2: Interfaces of the systems

5. What kind of changes or change propositions are GLD exposed to and how often are the GLD systems changed?

   Law legislations, faults discovered, new functionality etc. The systems are changed once a year, due to annual revision of the system. The faults are analyzed and corrected if critical errors are discovered.

6. Are changes based solely on the experience reports?

   No, not entirely. Changes are also caused by law legislations.

7. What kind of consequences can a change in a GLD system have for other GLD systems? (Ripple effect and cohesion/coupling between modules)

   A change in one system may have consequences for others. The system owner is responsible for change propositions and notifying responsible parties. It is important

that changes are thoroughly analyzed before implementation starts. An unfortunate incidence occurred when an extensive change imposed changes on other systems and stakeholders. None of the departments wanted to take responsibility for the system test. This indicated that the chain of command were indistinct.

8. Do some GLD systems require more maintenance and fault corrections than other?

   Yes, large systems require more maintenance than smaller once. Some systems are also more exposed to law legislation changes.

9. Are there GLD systems that are more stable regarded to changes, and have lower error rate than others? Yes..

10. Are the GLD systems still under development, or just in maintenance? Maintenance and development of annual versions of each GLD system and corresponding database.

SOA

1. What are the long term goals for introducing SOA? The government wants around the clock electronic administration ("Døgnåpen Elektronisk forvaltning") and co-ordinated services to the public. See the documents for more information:

   - "Eit informasjonssamfunn for alle"
   - "Arkitektur for Elektronisk samhandling i offentlig sector. Strategier og tiltak for mer samordning på IT-området"

2. Profits, cost-effectiveness and users?

   "MinSide" and "Altinn" are examples of web portals that simplify services to the population. An SOA platform for GLD systems might do the same. Other departments might also benefit from SOA. Same software components can be used for different systems, not only the GLD systems. Figure C.3 shows how a possible service oriented architecture might look.

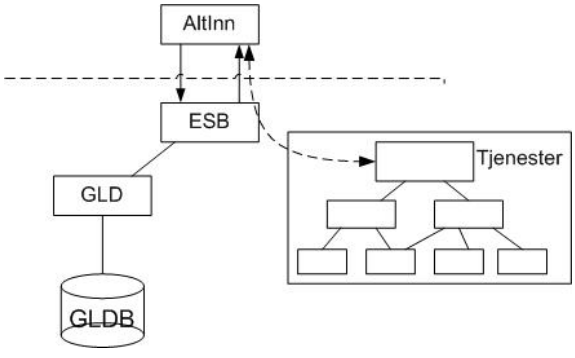

Figure C.3: Service oriented Architecture

3. What kind of services is relevant for SOA?

   Validation and control of schemes that user fills in. Reuse potential of services and components across departments and platforms etc.

4. Possible date for initiation?

   This year.

5. Have SKD internal competences on SOA?

   Yes, some competences. But consultants are also required.

6. Which supplier of ESB is relevant?

   All the major once, for example IBM and BEA.

7. What are the main challenges with introducing SOA?

   The major challenge with GLD is that it is based on batch. SKD wants a new architecture, with transactions and online processing instead of batch (sending files and waiting until the next day for answers).Figure C.4 tries to illustrate this.

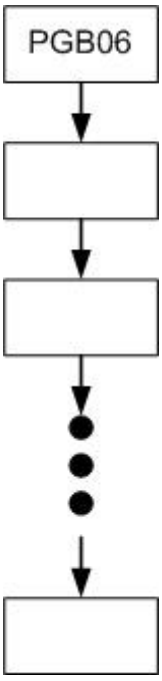

Figure C.4: Sequential batch

Architecture

1. Is there any documentation of the overall architecture of the GLD systems? (Logical view, process view, use cases, deployment view) There is a lot of documentation. We received, among other publications, a copy of "Systembeskrivelse Grunnlagsdata 2004".

2. Is there correspondence between documented architecture and implementation?

   No, architecture erosion has come a long way.

Software Development Method

1. What development method does SKD use?

   SKD has a method for system management. This method is called "Framework for System Maintenance (MARS)", and is designed to help manage all the applications that SKD is responsible for maintenance and further development on.

2. Is this method the same for all system groups?

   Yes, all system groups are supposed to use MARS.

3. Are you satisfied with the method?

   The method can be found at SKD's intranet, where all stakeholders can find it. The different phases are illustrated with relevant activities, guidelines, document templates etc can be found. This is very helpful.

4. What are the test procedures?

   SKD has a test strategy that is described in the MARS development method. System test, acceptance test and production test. TestDirector is used for coordinating and administrating the test activities.

COTS and Open Source

The GLD systems don't use any COTS or open source software, but the government wants all departments to take it into consideration and use it.

Much of the functionality from GLD systems are of generic types and it is likely that a lot of functionality can be bought from somebody else. PDF receipts and encryption standards are examples of functionality that SKD might consider buying and using in the GLD systems.

Appendix D

Resume of Meeting 15. October 2007, at SKD

The agenda for the meeting was the determination of the research questions. In collaboration with Reidar Conradi, who also participated on the meeting, we had suggested four research questions. Each of these research questions were discussed and elaborated at the meeting. To developers from SG2 and one developer from the "Architecture" group from SKD attended the meeting. It is important to notice that this is merely a summary of what was said on the meeting, and not exact quotations.

Organization-specific Goals

Tore Hovland requested that two organization-specific goals should be added to the project.

SKD goal 1: Propose a process which assures software reuse

- SKD's existing software development process; MARS, should be extended to incorporate software reuse
- Ensure that new components are developed with reuse in mind
- How to identify potential reusable assets /components

SKD goal 2: Propose an ideal-architecture for GLD, with focus on reuse

Suggestion: Investigate rule based engines/ systems. This could be appropriate for systems where rules changes, but were part of the system is stable. Search if there are relevant open source components or open standards that might be useful within a new architecture.

Research Questions

RQ1: What is the current state of software reuse in selected GLD systems? Software reuse is defined in IEEE, and concerns not only reuse of code but also documents, architectural patterns, use case etc. There are different levels of functionality and different levels of complexity concerning the changes that needs to be made to the different GLD systems. Each basis data (GLD) has its own set of rules which sets the basic for changes.

Some of these rules are more stable regarding annually changes to the regulations. The data structure can also vary in the different systems and annually versions (example; input from banks etc). An ideal solution for this would be a generic set of rules with a matching data structure.

A practical example regarding software reuse from SKD: The module that calculates taxes is the same that does the calculation for the final tax settlement. The developer from the "Architecture" group showed us a sketch for the GA/LSA component ("7.2 Prinsippskisse for LSA-komponenten), which was discussed.

Four GLD systems were selected. Line and Thor will start by describing the core functionality and subsystems of each of the selected GLD systems. This will result in four different case studies:

1. GA / LTO (Salary- and deduction assignments/ "Lønns- og trekkoppgaver") This system stands out from the rest of the GLDS, and should therefore be included in the study. The whole solution is considered in a new way and there are several potential interviewees. The system has a different owner than the rest of the systems, and is therefore administered and structured in a different way. The functionality is quite similar, but software code and database structural are different. GA has an interface against ALTINN.
2. GB (Credit and interest / "Saldo og renter") This is the system with most functionality.
3. GD (Building association / "Boligselskap") Almost identical with the GL ("Boligsameie") system and has an interface against ALTINN.
4. GK (Kindergartens - Care of children /"Barnehager - pass og stell av barn") GK is a simple and typical GLD system.

A fifth GLD system was mentioned: GJ (agriculture and cars /"Jordbruk og bil"). This system is insufficient and has little value (manual work, lacks validation etc). The system owners want an upgrade with more functionality, so that the system will be at the same level as the rest of the GLD systems. With this system, we have a potential to start from scratch. Reporting can be done via SOA, instead of the old architecture. Maybe this system could act as a basis for the SKD goal 2 - "ideal architecture"?

SCLM - Developer environment used by SKD, and probably also used as configuration management. We shall use a common notation like UML for documenting the architecture, process and products.

RQ2: Is reused components more stable regarding changes and defects compared to other components? Does this research question have to be rephrased since all GLD systems in one or another way are reused? Defect logs, change logs etc can be used as a starting point.

RQ3: What is the potential for reuse? The focus will be on both components and services, with emphasize on a service oriented architecture (SOA). We could see this research question together with SKD goal 1. How difficult will it be to incorporate changes in the current development process and which parts of the development process is ideal for reuse?

Who should be responsible for the reusable components? Is it possible to extend the existing roles at SKD, so that they can support reuse? System architects versus system

maintainers. There is a need for a coordinating group, because of conflicting demands and changes towards the reusable components. SKD is at the current time in acquisition for a new integration platform. Within the next few months there will be a reorganization of the organization (with more focus on architecture and testing).

RQ4: What is the emphasis on software reuse in current development processes? This question is not about inspecting the employee's knowledge or attitude towards software reuse. We want to investigate how well SKD has prepared the organization for reuse, so that the developers can practice and think about reuse in their daily work.

Conradi has a questionnaire that has been used on Ericsson and to smaller companies, to identify this sort of questions. This questionnaire will be the basis for this research question. We will also check out available strategy documents to see for any driving forces for software reuse.

Platforms

To different platforms is currently used:

- UNIX running Oracle
- IBM mainframes running CICS

Upgrading Oracle is more problematic than upgrading the mainframes because of interplay between different systems and versions of systems. There will be a gradual transition to a Java platform (J2EE). The integration platform will connect the different components and services.

Problems:

- Old code running on new platforms
- The architectural dimensions
- Performance

Appendix E

Resume of Meeting 26. November 2007, at SKD

We had prepared some questions which we wanted to be answered, before the meeting at SKD. These questions were reviewed by Reidar Conradi, and sent to Tore Hovland in advance of the meeting. It is important to notice that the answers below are just a summary of what was said on the meetings, and not exact quotations. The question was answered by various members of system group 2.

The day started with a joint meeting between six employees from SKD and us students. We showed the results from our analysis of the program- and system description from the following documents:

- sb-flt-06-b programbeskrivelse 20070430.doc
- sb-flt-06-b systembeskrivelse 20070530.doc
- sb-grld06-gx-z-CICS v.20061009-01.01.doc
- sb-grld06-gz v 20070424-01 01.doc

Summary of our Findings

The following is a summary of our findings:

1. GA/LTO

   Consist of 29 applications or procedures, and 124 programs. Four of the 124 programs was in use by more than one application/procedure: GAnP428, GAnP430, GAnP456 and GAnP510.

2. GLD

   (a) CICS

   The GLD CICS systems consists of 10 applications and 111 programs. None of the programs was in use by more than one procedure. Series of programs occurred often in the GLD systems. We assumed for example that the series GBnP301, GDnP301, GEnP301, GGnP301, GKnP301 and GLnP301 had to be

programs that were almost identical for the GB, GD, GK systems, etc. The following programs occurred in several GLD systems: XXnP301, XXnP302, XXnP303, XXnP304, XXnP305, XXnP380 and XXnP393.

(b) Batch

The GLD batch systems consists of 10 procedures and 52 programs. Only one program were in use by two procedures. This was the GXnP002 program that was used in both the LOAD and Identification routines.

3. Discussion Our results were confirmed under the meeting. One of the employees at system group 2, could tell us that the system documentation for the GLD batch programs was inadequate, since several of the programs were not mentioned in the documentation.

   During the last meeting at Skattedirektoratet, which was on October 15. 2007, it was decided that the GLD systems GA/LTO, GB, GD and GK was to be examined further by us. After the analysis of the programs, we decided to focus on GB, GD and GK because they are built quite similar. GA/LTO is build on a different way, and do not have much in common with the rest of the GLD systems.

   We determined to divide the day into two section. One would focus on CICS and the other one Batch.

Definition of Terms

The terms procedure and applications turned out to be quite confusing for us while reading the program descriptions. Under the meeting, an explanation of the differences between the two were given. For the Batch programs is the program partitioning called procedures, and for the CICS programs it is called application. Every procedure or application has one or more programs.

Batch

As mentioned, we could only find one program that was in use by more than one procedure. One of the developers listed up some an example of programs which were also in use by more than one application, as shown in the table E.1. These programs does not only exist for GB, but also the rest of the GLD systems except from GA/LTO.

Modification routine	Identification routine
GB6p007	GB6p007
GB6p009	GB6p008
GB6p010	GB6p009
GB6p011	GB6p010
GB6p012	GB6p011
GB6p014 *	GB6p012
	GB6p0013

Table E.1: Other programs used by more than one application

The earliest GLD systems were GB and GD. The remaining systems are copied and modified from these through the years.

An example of a program used by several procedures, is the GXnP002 program. This program checks the social security number from input file against the census to see if the number is in use. This program is used by all, since it uses the same database. The read routine is also the same for every GLD system, but the remaining routines has their own variant for each of the GLD systems.

When the GLD systems was developed in the early nineties, the amount of data that had to be stored was considered to be vast. Storing capacity and performance was important factors which lead to that each system got its own database.

System Modification

The GB, GD and GK systems are pretty stable regarding changes and the scope of changes between the annual versions. If there for example are five different systems that are affected by a change to the law regulation, this change is only revised in one of the systems source code. This modification is then copied and adjusted to the four remaining systems.

The main difference between the various GLD batch systems are which fields they take as input, sum fields, database definitions and the declarations of cursors. Copy member,which is a description of reusable data fields, are used so that data fields is not hard coded into the source code.

Questions for Developers

We had prepared some questions for the developers in SG2. It is important to notice that the answers below are just a summary of what was said on the meetings, and not exact quotations.

1. What separates the different programs in a series from each other?

   The programs GB, GD etc. uses different databases, and have some differences in the processing rules. They also have different screen menus. It's easier to maintain several systems than few large ones.

2. How much of the code is identical between the systems, for example GBnP301 and GDnP301?

   Much of the code is identical, but it was not possible to get an estimation of this. It was decided that SKD would email us the source code for GB6P301, GD6P301 and GK6P301, this way we could run tools for differencing locally.

3. The documentation states that the GX programs are common for several systems. What is the intention of these?

   The GX programs are general. In CICS for example, they are the main menus.

4. Are there any call graphs for GB, GD, GK or GA? What decides the order in which these programs will be executed? There are no call graphs for the GLD systems. Each program is executed from the main menu, or GX program. The user types a number to identify a particular system, and the program starts running. This is

where the 301 program comes in to play. BATCH programs have their order decided by JCL.

5. Are there any package diagrams for the GLD systems available?

   No.

6. Are the similarities between program with different numbering?

   No, the name of the program basically tells which programs are "copied and modified" from each other.

7. In which way has reusable programs been identified so far? And how can this process be improved?

   It's been a long time since the reused programs was copied and modified, except for the annual versions. There are no routines for this.

   SKD does not have any checklists when it comes to developing new annual versions, the developers have their own routines when performing these tasks. The lack of competence could become a problem in the future, because the lack of routines and since the technology used is rather old. Today, all annual versions are first copied from the preceding year, and then adjusted to fit the needs for this year (tables etc.) The developers are not sure about the future of the GLD systems, but believe they will be phased out eventually as new requirements and services appear. For example, all reporting should be performed using Altinn.

   It was also mentioned that the systems was very stable, and had few changed between the annual versions. However, some of the programs should have been structured in a better way to improve readability and modifiability.

Appendix F

Resume of Meeting 15. February 2008, at SKD

It is important to notice that this is merely a summary of what was said on the meeting, and not exact quotations. Various members from SG2 participated on the meeting.

Presentation of Findings

We presented our findings and results from our code analysis for the developers at SG2 and from the "Architecture" group. When confronted with the high amount of similarities between the different programs, we were told that the developers had tried to separate the commonalities into separate modules. The problem was that each program had different fields that needed validation, and these field often had different length. The programs does not separate logic and presentation. We asked if the programs could be divided into layers, but were told that the programs are build in a "traditional" COBOL-style, and that it is one program.

The Future of the GLD Systems

The developers told us that SKD will not conduct any changes to the COBOL-programs. This has low priority, and SKD has already limited resources.

SKD is about to establish a new integration platform. The MAG-project will propose a new technical solution for the basis-data systems. A developer from the "Architecture group" gave examples of the following ways to rewrite the existing systems:

- Batch
  - By packaging the batch-programs in a different way, it is possible that they can form one or more services
  - A cheap way to open up the systems on, but it still requires maintenance and it is time-consuming
  - It is not certain that this is more cost-beneficial than developing from scratch

- Altinn
  - The Altinn system performs considerable validation.
  - Validation is much of the logic in the GLD systems, and it is reasonable to believe that there will be changes to ghe GLD systems because of the LS and Altinn systems.
- Gradual transition
  - Reuse: new interfaces, web-solutions can be packaged into the presentation layer, closer integration between the systems, make CICS available through services.
  - Remove some of the functionality, rebuild to servers, repackage batch-programs and validation of data etc.

One or several Databases

It was discussed whether one or several databases would be more suitable for the GLD systems. The GLD systems is similar from year to year, where roughly speaking only year and database name is changed. It is therefore not necessary to establish new systems each year, but it is done because the databases is new each year.

The developer from the "Architecture" group was positive to only one database. His arguments were that one database would be more practical. Storage capacity and performance is cheap, and he believed that it is more costly to operate several databases now than before. Maintenance and administration also take up a great deal of time. With the help of XML, it is possible to only use one database for all the different GLD systems. A developer from SG2 did not agree with this proposed solution, because of the low change density to the fields in the databases. The 39 database tables is only altered when new law legislations set in, and it is not that costly to continue with todays approach with separate databases for each GLD system.

The GLD Systems Reuse Approach

Previous version of the different GLD systems is taken as a starting point when developing next years' system. This process is rapid, could one of the developers at the batch-programs tell us. About one day is used when making a new GLD system. The developer therefore question the gains when we proposed scripts or something of the sort, for automating this process (since the script also have to be maintained).

Another developer, working with CICS-programs came up with the following approach, which he referred to as the "X version" approach. The programs are created without any reference to which year of the databases they are communicating with. The same program can be used year after year, and only requires modifications if it is required. Instead, the programs are using database views which can be replaced each year. The view points the application to the correct year. This approach has the advantage that the programs can remain as-is, without being copied, pasted and modified each year. This reduces the total number of systems, thus, fewer systems will need to be maintained. Both views and databases must still be created as they did earlier.

The developers also told us that they use their own experience, consult other developers or system experts and randomly search for code.

The MAG-project

The developer from the "Architecture" group gave us an update on the status of the MAG-project. The MAG-project had become well-established since we started our research, and they had created their own suggestion for an ideal architecture for the GLD systems.

Appendix G

Resume of Interview regarding SKD's Framework for Software Maintenance, 15. February 2008

This is a resume of an interview regarding SKD's framework for software maintenance. At first we only intended to interview the person who was in charge of SKD's software maintenance process, but it turned out that our questions were difficult to understand, so an extra person from the "Architecture" group was called in to help.

1. How would you say that the software maintenance process is functioning?

   This method has been in use since the spring 2005. Before this SKD had no development or maintenance process. It is challenging to get people to think new thoughts - a change of attitude. About 250 people has gotten training in the software maintenance process, and there are constantly more people who decide to use it.

   There are many different systems and needs. This is why the framework is superior and can be adjusted to fit different needs. A good deal of information and working methods is in peoples head, and SKD is very depended upon specific persons. The answer to the question is therefor depended on who you are asking. Altinn and some of the smaller systems do not use the process. Most of the persons that the interviewees has talked to, is positive to the process.

2. How would you say that the software maintenance process is adjusted for reuse of artifacts?

   Not at all. There are no formal structure of this.

   (a) In which phases is it possible to plan for software reuse?

   The interviewee from the "Architecture" group says it is possible to plan for reuse in all of the phases. It is important to consider reuse as early as possible. The software maintenance process is not especially adjusted for this and has little focus on deliveries.

   (b) In which phases is it possible to use existing artifacts as design documents, code etc.?

   Don't know.

(c) Is it possible to supplement the software maintenance process with routines and templates for software reuse?

Yes it is possible, but the questions is on what level should the routines and templates have. After some discussion we conclude that such routines and templates have to be especially adjusted to the different types of systems. Different systems have different needs. The interviewees questions the gains of this.

(d) How would you identify reusable services, with respect to SOA?

No one has asked these questions before. SKD is currently working on establishing a new integration platform, but there are no concrete plans for how to identify reusable services.

i. Does it exist any routines or templates for this?

No, it does not exist any routines or templates for identifying reusable services.

ii. Does it exist any structured catalogue that keep track of reusable services?

No, it does not.

3. Have you accepted any requests from system groups or other units about new routines for reuse?

No one has requested anything - maybe people wants as little as possible.

(a) Do you know if the software developers or system owners are engaged in reuse?

Don't know.

(b) Is reuse systematized as a formal process?

No, it is not.

4. How would you say that the management adjusts for reuse of documents, programs etc.?

It depends on what you mean when you say the "management", because it is different levels of management. Some people has worked on this for a long time, but I would say that the management has to be persuaded.

Reuse has been on the agenda, but it is not until now that is has gotten a break through. The "Platform project" is a pioneer and there are many people who are involved in this. Until now it has been five core groups in addition to reference groups, and more and more are getting involved.

Appendix H

Questionnaire of the Software Development Process and Reuse Aspects at Skattedirektoratet, February 2008

The purpose of this questionnaire is to map how SKD's employees view their development process especially concerning reuse, as well as where they feel changes to the development process may be necessary.

The questionnaire consists of multiple-choice questions. Mark the choices you feel is most correct. Be honest; if you feel the current developments process works perfectly and is in need of no changes, say so.

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.

Personal Information

P1: What is your current role at Skattedirektoratet?

P2: How long have you been working at Skattedirektoratet?

P3: How many projects have you been involved in?

P4: Which programming and design languages are you familiar with?

P5: Which programming and design languages are you currently using?

P6: What is your highest completed academic degree?

General

G1: How important do you consider reuse in achieving the following benefits:

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:

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 artifacts with respect to reuse:

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

Comments:

Components

C1: During development:
[ ] There is too much reuse of code / design components going on.
[ ] The percentage of code/design components reused is as high as possible (optimized).
[ ] There should be more reuse of code / design components.

C2: Do you feel that the process of finding, assessing and reusing existing code / design components is functioning?
[ ] Yes [ ] No

C3a: Is the existing code / design components sufficiently documented?
[ ] Yes [ ] Sometimes [ ] No

C3b: If 'Sometimes' or 'No': Is this a problem?
[ ] Yes [ ] No

C4: 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

C5: How would you decide whether to reuse a code / design component "as-is", reuse "with modification", or make a new component from scratch?
[ ] By following the guidelines
[ ] By consulting experts
[ ] Not clearly defined

C6: A code / design component that is reused (and possibly modified) is usually:
[ ] More stable and cause less problems than a component that is created from scratch

[ ] About equal to a component created from scratch
[ ] Inferior to a component created from scratch (in performance, stability and so on)

C7: Integration when reusing a component
[ ] Usually works well (the components usually fit easily into the architecture)
[ ] May cause some problems
[ ] Is difficult (hard to fit component into architecture)

C8: Is any extra effort put into testing/documenting potentially reusable components?
[ ] Yes [ ] No

C9: To what extend do you feel affected by reuse in your work?
[ ] very high [ ] high [ ] medium [ ] little [ ] no [ ] don't know

C10a: Is the design/code of reusable components sufficiently documented?
[ ] Yes [ ] No [ ] Sometimes [ ] don't know

C10b: If the answer to C10a is 'sometimes' or 'no', is this a problem?
[ ] Yes [ ] No [ ] Sometimes

C10c: What is your main source of information about reusable components during implementation?

Comments:

Requirements

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 fulfill and so on. In such cases it may be necessary for the stakeholders to renegotiate requirements.

R1: Is the requirements renegotiation process at Skattedirektoratet 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:

Appendix I

Results of Questionnaire of the Software Development Process and Reuse Aspect at SKD, February 2008

The purpose of this questionnaire was to map how SKD'a employees view their development process especially concerning reuse, as well as where they feel changes to the development process may be necessary.

Explanation of Terms

The following is a short explanation of the different terms that appears in the tables[4].

- Frequency: Number of respondents who has answered each category
- Percent: Number of respondents who has answered each category given in percent
- Valid Percent: Number of respondents minus missing respondents, given in percent
- Cumulative Percent: The valid percent of that value added to the valid percent of the previous values
- Missing: Number of respondents that has not answered the question

Personal Info

P1: Table I.1 shows the respondents current role at SKD.

P2: Table I.2 shows how long the respondents have been working at SKD.

P3: Table I.3 shows how many projects the respondents have been involved in.

P4: Familiar with the following programming and design languages: Ada, ASP, Assembler, Basic, C, C++, CICS, COBOL, DB2, Easytrieve, HTML, Java, Javascript, JCL, Lisp, Turbo Pascal, Perl, PHP, Prolog SQL, PRO-IV, PL1, Matlab, UML, XML, Oracle Warehouce builder (OWB)etc.

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	Adviser	2	8,0	8,0	8,0
	Architect of data warehouse	1	4,0	4,0	12,0
	IT first consultant	1	4,0	4,0	16,0
	Responsible for a system	2	8,0	8,0	24,0
	Senior Consultant	1	4,0	4,0	28,0
	System developer	17	68,0	68,0	96,0
	Team leader	1	4,0	4,0	100,0
	Total	25	100,0	100,0	

Table I.1: Results from question P1: Current role at SKD

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	1,00	5	20,0	20,0	20,0
	2,00	1	4,0	4,0	24,0
	3,00	1	4,0	4,0	28,0
	5,00	5	20,0	20,0	48,0
	6,00	1	4,0	4,0	52,0
	7,00	1	4,0	4,0	56,0
	8,00	2	8,0	8,0	64,0
	10,00	1	4,0	4,0	68,0
	13,00	1	4,0	4,0	72,0
	17,00	2	8,0	8,0	80,0
	19,00	1	4,0	4,0	84,0
	25,00	3	12,0	12,0	96,0
	27,00	1	4,0	4,0	100,0
	Total	25	100,0	100,0	

Table I.2: Results from question P2: Number of years working at SKD

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	,00	1	4,0	4,3	4,3
	1,00	7	28,0	30,4	34,8
	2,00	3	12,0	13,0	47,8
	3,00	2	8,0	8,7	56,5
	4,00	3	12,0	13,0	69,6
	5,00	3	12,0	13,0	82,6
	6,00	1	4,0	4,3	87,0
	10,00	2	8,0	8,7	95,7
	15,00	1	4,0	4,3	100,0
	Total	23	92,0	100,0	
Missing	System	2	8,0		
Total		25	100,0		

Table I.3: Results from question P3: Number of projects

16 of the respondents are familiar with COBOL. 12 of the respondents were familiar with Java, were 7 of these also were familiar with COBOL.

P5: Currently using the following programming and design languages: CICS, COBOL, Easytrive, JCL, Java, Javascript, PRO-IV, Oracle Designer, Oracle Warehouse Builder etc.

- Java: 5
- COBOL: 7
- Oracle Warehouse Builder: 4
- Javascript: 3

P6: Table I.4 shows the highest completed academic degree.

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	Videregående	5	20,0	20,8	20,8
	Bachelorgrad	12	48,0	50,0	70,8
	Mastergrad	7	28,0	29,2	100,0
	Total	24	96,0	100,0	
Missing	System	1	4,0		
Total		25	100,0		

Table I.4: Results from question P6: Highest completed academic degree

General Questions G1

The respondents were asked how important they considered reuse to be, in achieving benefits such as lower development costs, shorter development time, higher product quality, more standardized architecture and lower maintenance costs.

G1a: For achieving lower development costs, reuse is of 'X' importance according to table I.5.

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	Very high	5	20,0	20,0	20,0
	High	7	28,0	28,0	48,0
	Medium	12	48,0	48,0	96,0
	Don't know	1	4,0	4,0	100,0
	Total	25	100,0	100,0	

Table I.5: Results from question G1a: For achieving lower development costs

G1b: For achieving shorter development time, reuse is of 'X' importance according to table I.6.

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	Very high	5	20,0	20,0	20,0
	High	8	32,0	32,0	52,0
	Medium	9	36,0	36,0	88,0
	Little	2	8,0	8,0	96,0
	Don't know	1	4,0	4,0	100,0
	Total	25	100,0	100,0	

Table I.6: Results from question G1b: For achieving shorter development time

G1c: For achieving higher product quality, reuse is of 'X' importance according to table I.7.

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	Very high	8	32,0	32,0	32,0
	High	10	40,0	40,0	72,0
	Medium	4	16,0	16,0	88,0
	Little	2	8,0	8,0	96,0
	No	1	4,0	4,0	100,0
	Total	25	100,0	100,0	

Table I.7: Results from question G1c: For achieving higher product quality

G1d: For achieving a more standardized architecture, reuse is of 'X' importance according to table I.8.

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	Very high	7	28,0	28,0	28,0
	High	12	48,0	48,0	76,0
	Medium	4	16,0	16,0	92,0
	Little	2	8,0	8,0	100,0
	Total	25	100,0	100,0	

Table I.8: Results from question G1d: For achieving a more standardized architecture

G1e: For achieving lower maintenance costs (including technology updates), reuse is of 'X' importance according to table I.9.

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	Very high	7	28,0	28,0	28,0
	High	8	32,0	32,0	60,0
	Medium	7	28,0	28,0	88,0
	No	1	4,0	4,0	92,0
	Don't know	2	8,0	8,0	100,0
	Total	25	100,0	100,0	

Table I.9: Results from question G1e: For achieving lower maintenance costs

Comments:

In COBOL or Easytrieve programming is very practical to reuse code by copying useful code into the programs when we are making new systems. Or else we reuse copy members which are code used by several programs or systems. We also use programs who do the same function for several systems by calling up these programs (or also the copy members) when needed. This is mostly in COBOL code programming.

For most of the systems I work on, we make a new version every year because of new tax rules every ear. Most of the tax rules are the same, but there are some new rules every year. The reuse of code is very important, and make us able to make new versions easily.

Reuse also carries risk in the sense that errors are spread proportionally to the degree of reuse. And a too religious approach may be counterproductive: In the pursuit of reusing a module or class (incorporate it in a new system), one may adopt a bad overall design in the new system just to be able to reuse the module - if it is a simple module, writing new code may be the right course. And finally it is often much more quicker to write the necessary code when one needs it, than to locate a reusable module/class that may or may not exist. Done right, however, reuse is important.

Making components reusable incurs extra resources.

In practice, reuse is not achieved very often. Would need a lot more to see real benefits. I think it's important to standardize the implementation of a few common tasks .

I don't think we have the opportunity to reuse components here. Unless I make a system that is similar to something I have made before. Then I know my code and can reuse what I want.

General Questions G2

The respondents were asked how useful or important they found reuse/component based technologies, OO technologies, testing, inspections, formal inspections and configuration management to be.

G2a: Reuse/component technologies are of 'X' importance according to table I.10.

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	Very high	3	12,0	12,0	12,0
	High	11	44,0	44,0	56,0
	Medium	7	28,0	28,0	84,0
	Don't know	4	16,0	16,0	100,0
	Total	25	100,0	100,0	

Table I.10: Results from question G2a: Reuse/component technologies

G2b: OO technologies (java, UML, CORBA) are of 'X' importance according to table I.11.

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	Very high	3	12,0	12,0	12,0
	High	9	36,0	36,0	48,0
	Medium	4	16,0	16,0	64,0
	Little	2	8,0	8,0	72,0
	Don't know	7	28,0	28,0	100,0
	Total	25	100,0	100,0	

Table I.11: Results from question G2b: OO technologies

G2c: Testing is of 'X' importance according to table I.12.

G2d: Inspections are of 'X' importance according to table I.13.

G2e: Formal specifications /methods are of 'X' importance according to table I.14.

G2f: Configuration management is of 'X' importance according to table I.15.

Comments: Mainframe are very powerful computers, but they are not built for OO technologies hence cannot make much use of what OO technologies have to offer. So in the mainframe world OO technologies are of little importance.

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	Very high	14	56,0	56,0	56,0
	High	11	44,0	44,0	100,0
	Total	25	100,0	100,0	

Table I.12: Results from question G2c: Testing

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	Very high	3	12,0	12,5	12,5
	High	11	44,0	45,8	58,3
	Medium	7	28,0	29,2	87,5
	Little	1	4,0	4,2	91,7
	Don't know	2	8,0	8,3	100,0
	Total	24	96,0	100,0	
Missing	System	1	4,0		
Total		25	100,0		

Table I.13: Results from question G2d: Inspections

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	Very high	2	8,0	8,3	8,3
	High	12	48,0	50,0	58,3
	Medium	8	32,0	33,3	91,7
	Little	1	4,0	4,2	95,8
	Don't know	1	4,0	4,2	100,0
	Total	24	96,0	100,0	
Missing	System	1	4,0		
Total		25	100,0		

Table I.14: Results from question G2eb: Formal specifications

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	Very high	2	8,0	8,0	8,0
	High	11	44,0	44,0	52,0
	Medium	5	20,0	20,0	72,0
	Little	1	4,0	4,0	76,0
	Don't know	6	24,0	24,0	100,0
	Total	25	100,0	100,0	

Table I.15: Results from question G2f: Configuration management

The code you have tested, you know is ok. If the system abends, you know where you don't have to search for the error. Testing is 'to be or not to be' for a programmer!

Inspections of code produced by juniors is very important (early desk check). Value of formal methods depends upon project size and complexity.

General Questions G3

The respondents were asked how useful or important they considered requirements, use cases, design, code and test data/documentation with respect to reuse to be.

G3a: Requirements are of 'X' importance according to table I.16.

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	Very high	2	8,0	8,0	8,0
	High	9	36,0	36,0	44,0
	Medium	6	24,0	24,0	68,0
	Little	3	12,0	12,0	80,0
	Don't know	5	20,0	20,0	100,0
	Total	25	100,0	100,0	

Table I.16: Results from question G3a: Requirements

G3b: Use cases are of 'X' importance according to table I.17.

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	Very high	3	12,0	12,5	12,5
	High	6	24,0	25,0	37,5
	Medium	6	24,0	25,0	62,5
	Little	3	12,0	12,5	75,0
	Don't know	6	24,0	25,0	100,0
	Total	24	96,0	100,0	
Missing	System	1	4,0		
Total		25	100,0		

Table I.17: Results from question G3b: Use case

G3c: Design is of 'X' importance according to table I.18.

G3d: Code is of of 'X' importance according to table I.19.

G3e: Test data/documentation is of 'X' importance according to table I.20.

Comments: It is good to have a quality, complete requirements but is should not affect the reusability of the system. The design would be the most important phase to ensure that the system components are reusable. Test data are to test out it the system was developed according to the specific requirements. It has little to do with system reuse.

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	Very high	3	12,0	12,5	12,5
	High	9	36,0	37,5	50,0
	Medium	5	20,0	20,8	70,8
	Little	3	12,0	12,5	83,3
	Don't know	4	16,0	16,7	100,0
	Total	24	96,0	100,0	
Missing	System	1	4,0		
Total		25	100,0		

Table I.18: Results from question G3c: Desgin

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	Very high	4	16,0	16,0	16,0
	High	9	36,0	36,0	52,0
	Medium	10	40,0	40,0	92,0
	Don't know	2	8,0	8,0	100,0
	Total	25	100,0	100,0	

Table I.19: Results from question G3d: Code

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	Very high	4	16,0	16,0	16,0
	High	9	36,0	36,0	52,0
	Medium	6	24,0	24,0	76,0
	Little	3	12,0	12,0	88,0
	Don't know	3	12,0	12,0	100,0
	Total	25	100,0	100,0	

Table I.20: Results from question G3e: Test data/documentation

Documentation is very useful at maintenance phase where a coder can use as a reference hence make use of the system component. But the system had to have reusable components at first place.

We work with systems with only a few changes every year, therefore we may run the same tests year after year.

Code quality is of course very important, but the implementation details should not be when reusing an OO component. Not sure how to interpret the question.

Component Questions

C1: During development according to table I.21:

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	Too much reuse	2	8,0	8,0	8,0
	As high as posible	6	24,0	24,0	32,0
	Should be more reuse	17	68,0	68,0	100,0
	Total	25	100,0	100,0	

Table I.21: Results from question C1: During development

C2: Do you feel that the process of finding, assessing and reusing existing code/design components is functioning? Answers in table I.22.

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	No	10	40,0	40,0	40,0
	Yes	15	60,0	60,0	100,0
	Total	25	100,0	100,0	

Table I.22: Results from question C2: How the process of finding, assessing and reusing existing code/design is functioning

C3a: Is the existing code/design components sufficiently documented? Answers in table I.23.

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	Yes	3	12,0	12,0	12,0
	Sometimes	17	68,0	68,0	80,0
	No	5	20,0	20,0	100,0
	Total	25	100,0	100,0	

Table I.23: Results from question C3a: Is the existing code/design components sufficiently documented

C3b: If "Sometimes" or "No": Is this a problem? Answers in table I.24.

		If 'sometimes' or no': is this a problem?		
		Yes	No	Total
Is the existing code/design components sufficiently documented?	Yes	1	0	1
	Sometimes	11	4	15
	No	4	1	5
Total		16	5	21

Table I.24: Results from question C3b: If sometimes or no

C4: Would the construction of a reuse repository, with extra component documentation etc be worthwhile or not? Answers in table I.25.

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	Not worthwhile	7	28,0	28,0	28,0
	Worthwhile	18	72,0	72,0	100,0
	Total	25	100,0	100,0	

Table I.25: Results from question C4: Construction of a reuse repository

C5: How would you decide whether to reuse a code/design component "as-is", reuse "with modification" or make a new component from scratch? Answers in table I.26.

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	Follow guidelines	4	16,0	16,0	16,0
	Consult experts	4	16,0	16,0	32,0
	Not clearly defined	16	64,0	64,0	96,0
	Following guidelines and consulting experts	1	4,0	4,0	100,0
	Total	25	100,0	100,0	

Table I.26: Results from question C5: How to decide whether to reuse a code/design component "as-is", reuse "with modification" or make a new component from scratch

C6: A code/design component that is reused (and possibly modified) is usually more stable, about equal or inferior to a component created from scratch according to table I.27.

C7: Integration when reusing a component usually works well, may cause some problems or is difficult according to table I.28.

C8: Is any extra effort put into testing/documenting potentially reusable components? Answers in table I.29.

C9: To what extend to you feel affected by reuse in your work? Answers in table I.30.

C10a: Is the design/code of reusable components sufficiently documented? Answers in table I.31.

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	More stable and cause less problems	17	68,0	68,0	68,0
	About equal to a component created from scratch	7	28,0	28,0	96,0
	Inferior to a component created from scratch	1	4,0	4,0	100,0
	Total	25	100,0	100,0	

Table I.27: Results from question C6: A code/design component that is reused (and possibly modified) is usually more..

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	Usually works well	8	32,0	34,8	34,8
	May cause some problems	13	52,0	56,5	91,3
	Is difficult	2	8,0	8,7	100,0
	Total	23	92,0	100,0	
Missing	System	2	8,0		
Total		25	100,0		

Table I.28: Results from question C7: Integration

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	Yes	10	40,0	41,7	41,7
	No	14	56,0	58,3	100,0
	Total	24	96,0	100,0	
Missing	System	1	4,0		
Total		25	100,0		

Table I.29: Results from question C8: Extra effort

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	Very high	2	8,0	8,0	8,0
	High	6	24,0	24,0	32,0
	Medium	7	28,0	28,0	60,0
	Little	7	28,0	28,0	88,0
	No	1	4,0	4,0	92,0
	Don't know	2	8,0	8,0	100,0
	Total	25	100,0	100,0	

Table I.30: Results from question C9: Feel affected by reuse

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	Yes	2	8,0	8,0	8,0
	No	2	8,0	8,0	16,0
	Sometimes	15	60,0	60,0	76,0
	Don't know	6	24,0	24,0	100,0
	Total	25	100,0	100,0	

Table I.31: Results from question C10a: Documented

C10b: If the answer to C10a is "sometimes" or "no", is this a problem? Answers in table I.32.

		the answer to C10a is 'sometimes' or 'no', is this a problem			
		Yes	No	Sometimes	Total
the design/code of reusable components sufficiently documented	No	2	0	0	2
	Sometimes	3	3	9	15
	Don't know	1	0	0	1
Total		6	3	9	18

Table I.32: Results from question C10b: If the answer..

C10c: What is your main source of information about reusable components during implementation?

- Randomly searching code. Consulting older system experts. Reuse has no agenda in my daily work WHATSOEVER! It's not prioritized by the 'bosses'
- Experience
- System documentation
- I ask persons who have worked with similar tasks. I have also, as the years go by, made myself a little 'library' of code I think may be useful in the future
- Look at the description in heading and at the code
- The GLD system
- Persons who know the reusable components
- The code itself
- My own knowledge and the experience of my coworkers.
- Tips from other developers about specific preexisting modules. Also, reuse is built into our existing practices: A lot of systems are made a new each year, and the reuse in these cases occur when the new code is copied from the last year - this happens regularly
- General system knowledge

- My own experience with those components, and advice from the developers i'm working with. Also some help from google when third party components is an option
- My own experience with those components, and advice from the developers I'm working with. Also some help from Google when third party components is an option
- Own experience (so I know where to look for code to copy/reuse)

Requirement Questions

R1: Is the requirements renegotiation process at Skattedirektoratet working sufficiently? Answers in table I.33.

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	Yes	16	64,0	76,2	76,2
	No	5	20,0	23,8	100,0
	Total	21	84,0	100,0	
Missing	System	4	16,0		
Total		25	100,0		

Table I.33: Results from question R1: Requirement negotiation process

R2: In a typical project are requirements usually flexible, have no particular trend or very rigid? Answers in table I.34.

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	Usually flexible	8	32,0	38,1	38,1
	No particular trend	12	48,0	57,1	95,2
	Usually very rigid	1	4,0	4,8	100,0
	Total	21	84,0	100,0	
Missing	System	4	16,0		
Total		25	100,0		

Table I.34: Results from question R2: In a typical project..

R3: Are requirements often changed/renegotiated during a development project? Answers in table I.35.

Comments: I have answered the questions here based on my experience with the systems I am involved with; these are mainly "bread and butter" systems (systems that are made a new each year), and the changes are thus incremental). The dynamics in these systems are no problem; they are expected and we work very close with the people specifying, often using an approach akin to prototyping. SKD as such do however at any given time run a lot of projects, and my impression is that in these, unexpected changes do happen during the course of the project. The reasons for this are legio, and among them are insufficient original specifications, badly chosen technology and subcontractors exploiting

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	Often	9	36,0	42,9	42,9
	Sometimes	10	40,0	47,6	90,5
	Seldom	2	8,0	9,5	100,0
	Total	21	84,0	100,0	
Missing	System	4	16,0		
Total		25	100,0		

Table I.35: Results from question R3: How often is requirements changed

things in the specifications that the specifiers took for granted; this necessitates new and more detailed specifications, and is accompanied by "quarrels" about whether the new specifications represent changes (for which the subcontractor may charge extra) or if they only represent clarifications(no extra cost).

Cross Tabulation Analysis

C2 vs. C9 is shown in table I.36.

		what extend do you feel affected by reuse in your work						
		Very high	High	Medium	Little	No	Don't know	Total
Do you feel that the process of finding, assessing and reusing is functioning?	No	2	4	2	1	0	1	10
	Yes	0	2	5	6	1	1	15
Total		2	6	7	7	1	2	25

Table I.36: Cross tabulation analysis of questions C2 and C9

C4 vs. C9 is shown in table I.37.

		what extend do you feel affected by reuse in your work						
		Very high	High	Medium	Little	No	Don't know	Total
Construction of a reuse repository	Not worthwhile	2	0	2	2	0	1	7
	Worthwhile	0	6	5	5	1	1	18
Total		2	6	7	7	1	2	25

Table I.37: Cross tabulation analysis of questions C4 and C9

C1 vs. C9 is shown in table I.38.

C5 vs. C9 is shown in table I.39.

C6 vs. C9 is shown in table I.40.

C7 vs. C9 is shown in table I.41.

C8 vs. C9 is shown in table I.42.

C10a vs. C9 is shown in table I.43.

C2 vs. C3a is shown in table I.44.

		what extend do you feel affected by reuse in your work						
		Very high	High	Medium	Little	No	Don't know	Total
During development	Too much reuse	0	1	0	0	0	1	2
	As high as posible	2	1	2	1	0	0	6
	Should be more reuse	0	4	5	6	1	1	17
Total		2	6	7	7	1	2	25

Table I.38: Cross tabulation analysis of questions C1 and C9

		what extend do you feel affected by reuse in your work						
		Very high	High	Medium	Little	No	Don't know	Total
How to decide whether to reuse a code/design component	Follow guidelines	0	0	2	1	1	0	4
	Consult experts	0	4	0	0	0	0	4
	Not clearly defined	1	2	5	6	0	2	16
	Following guidelines and consulting experts	1	0	0	0	0	0	1
Total		2	6	7	7	1	2	25

Table I.39: Cross tabulation analysis of questions C5 and C9

		what extend do you feel affected by reuse in your work						
		Very high	High	Medium	Little	No	Don't know	Total
the design/code of reusable components sufficiently documented	Yes	1	0	0	1	0	0	2
	No	1	1	0	0	0	0	2
	Sometimes	0	5	6	3	0	1	15
	Don't know	0	0	1	3	1	1	6
Total		2	6	7	7	1	2	25

Table I.40: Cross tabulation analysis of questions C6 and C9

		what extend do you feel affected by reuse in your work						
		Very high	High	Medium	Little	No	Don't know	Total
Integration when reusing a component	Usually works well	1	4	2	1	0	0	8
	May cause some problems	0	2	4	5	1	1	13
	Is difficult	0	0	1	1	0	0	2
Total		1	6	7	7	1	1	23

Table I.41: Cross tabulation analysis of questions C7 and C9

		what extend do you feel affected by reuse in your work						
		Very high	High	Medium	Little	No	Don't know	Total
Is any extra effort put into testing/documenting potentially reusable components	Yes	2	3	3	1	0	1	10
	No	0	3	4	6	1	0	14
Total		2	6	7	7	1	1	24

Table I.42: Cross tabulation analysis of questions C8 and C9

		what extend do you feel affected by reuse in your work						
		Very high	High	Medium	Little	No	Don't know	Total
the design/code of reusable components sufficiently documented	Yes	1	0	0	1	0	0	2
	No	1	1	0	0	0	0	2
	Sometimes	0	5	6	3	0	1	15
	Don't know	0	0	1	3	1	1	6
Total		2	6	7	7	1	2	25

Table I.43: Cross tabulation analysis of questions C10a and C9

		Is the existing code/design components sufficiently documented?			
		Yes	Sometimes	No	Total
Do you feel that the process of finding, assessing and reusing is functioning?	No	2	6	2	10
	Yes	1	11	3	15
Total		3	17	5	25

Table I.44: Cross tabulation analysis of questions C2 and C3a

C2 vs. C10a shown in table I.45.

		the design/code of reusable components sufficiently documented				
		Yes	No	Sometimes	Don't know	Total
Do you feel that the process of finding, assessing and reusing is functioning?	No	2	2	5	1	10
	Yes	0	0	10	5	15
Total		2	2	15	6	25

Table I.45: Cross tabulation analysis of questions C10a and C2

C2 vs. C4 is shown in table I.46.

		Construction of a reuse repository		
		Not worthwhile	Worthwhile	Total
Do you feel that the process of finding, assessing and reusing is functioning?	No	5	5	10
	Yes	2	13	15
Total		7	18	25

Table I.46: Cross tabulation analysis of questions C2 and C4

C2 vs. C5 is shown in table I.47.

C2 vs. C7 is shown in table I.48.

		How to decide whether to reuse a code/design component				
		Follow guidelines	Consult experts	Not clearly defined	Following guidelines and consulting experts	Total
Do you feel that the process of finding, assessing and reusing is functioning?	No	0	2	7	1	10
	Yes	4	2	9	0	15
Total		4	4	16	1	25

Table I.47: Cross tabulation analysis of questions C2 and C5

		Integration when reusing a component			
		Usually works well	May cause some problems	Is difficult	Total
Do you feel that the process of finding, assessing and reusing is functioning?	No	5	4	0	9
	Yes	3	9	2	14
Total		8	13	2	23

Table I.48: Cross tabulation analysis of questions C2 and C7