Abstract

Code reuse in object oriented software development has been common for some time. A recent study performed by the author revealed that while software developers in small Norwegian companies regard code reuse as important and useful, they are prone to perform ad-hoc reuse. This reduces the positive effects achieved through reuse, and although most of the developers wish to perform more systematic reuse, they do not know how to do this. This thesis aims to help amend this problem by developing a set of guidelines describing the process of making a plan for reuse. To develop the guidelines, a literature study was performed, followed by three phases of writing. Between the three phases of writing, two rounds of three feedback interviews were performed to elicit information on the usability and clarity of the guidelines. Each feedback interview was performed with a developer from a small Norwegian company at the developer's workplace. After each set of interviews, the guidelines were revised and improved. The final set of guidelines presented in this report was considered by the developers to be easily understandable and useful, but further work remains to make the guidelines complete; a set of examples of how the process could be performed is essential to help the developers make the leap from the theoretical descriptions of the guidelines to making their own plan for reuse.

Preface

This thesis ("TDT4900 Datateknikk, masteroppgave" – Computer science, Master's thesis) was completed during the spring semester 2005. The author is a fifth year student at the Norwegian University of Science and Technology (NTNU), Faculty of Information Technology, Mathematics and Electrical Engineering (IME), Department of Computer and Information Science (IDI), where this course is taught.

My sincere thanks to the developers I interviewed; for taking the time to participate in the feedback stages, and sharing their insight and real-world expertise. I would also like to thank Tor Stålhane, for help and guidance throughout the work with this Master's thesis. Finally, I would like to thank Peter Rønning, who always has the time to give a helping hand.

Contents

List of Figures

Chapter 1

Introduction

This chapter contains an introduction to this Master's thesis, starting with a description of my motivation for doing this Master's thesis, followed by the problem definition and the goals I wanted to reach. A description of research methodology is given, before the contents of the rest of this report are outlined.

1.1 Motivation

Since I first started programming four years ago, I have produced a lot of code. Different projects and assignments meant writing different code, but more and more often I found myself thinking "Oh, I've done this before!" This was followed by a intense search through directories and files to find where exactly I had this piece of code. More often than not I came up with nothing and had to program the same code over again. Every time, I thought to myself "If only I had a personal code library". In addition to my own code I have also had access to others' code in group projects and assignments. It would be nice to have this code in a library as well. But the time and efforts needed to create my own library of re-usable code seemed to be greater than I could afford. This seemed to be the case for many of my fellow students as well.

The subject of code reuse indeed seemed intriguing. Last autumn, I therefore did a project entitled "Code Reuse in Object Oriented Software Development"[1]. See chapter 3 for a more detailed description of the project, and appendix A for definitions of the term "code reuse" and other terms used in this report. The project aimed to create an understanding of the nature of code reuse in a group of software developing companies in Norway. I conducted a series of interviews with software developers from 24 software

developing companies of different sizes, locations, and application domains. Through the project work, I learned that the software developers reusing code desired certain effects from reuse, such as improved quality and increased efficiency. Many reported that they felt the achieved effects of reuse were the same as the desired effects. However, most of them seemed not to be very conscious about the issues of reuse, lacking a formulated set of goals and a plan for reuse. Many expressed a desire of improving the way they reused code, but only one (a large, international) company out of the 24 companies had reuse specific procedures and employed a system specifically made for reuse. Mostly, the lack of resources and the fact that the development departments were relatively small (2 to 25 developers) made it difficult to improve the situation.

Based on my interest in code reuse and as it is likely that I myself will be working in a company with a relatively small software development department, I felt a desire to contribute. It seemed like a good thing to provide some tips and hints about how reuse can and should be performed, preferably keeping costs both in time and money to a minimum. Such a set of tips and hints would save the developers a lot of time, as they would not have to search for and summarize the information themselves.

1.2 Problem Definition

I had to be more specific about what I wanted to do, and decided that I would use last autumn's project as a starting point. In the preliminary stages of the project work, I gave some examples of project and thesis ideas. The idea of contributing with tips and hints about how reuse can and should be performed was similar to one of those ideas; the project was to learn about tools and procedures for reuse, while the thesis would be about developing procedures for reuse. I decided to present the tips and hints in the form of a procedure, and decided to use the term "guidelines" to describe this.

As the thesis would be based on the project work, I decided to use the same title: "Code Reuse in Object Oriented Software Development". The project description was as follows: *"Proper reuse of code increases the speed of software development projects." (Rickard Öberg) Code reuse in object oriented software development has been common for some time. A recent study performed by the candidate revealed that while software developers in small Norwegian companies regard code reuse as important and useful, they are prone to perform ad-hoc reuse. This reduces the positive effects achieved through reuse, and although most of the developers wish to perform more systematic reuse, they do not know how to do this. The task is to create a set of guidelines which deals with important issues of reuse, to help software*

developers reuse code in a more proper way.

Reuse is, however, a loose term. As in the project, I wanted to look at the lower end of reuse; the reuse of source code. This in contrast to higher level reuse such as patterns or process reuse/experience databases. There are many definitions of and ways to interpret "code reuse". I wanted the overview of how the software industry reuses code to be as general as possible. Thus, I decided not to exclude any interpretations of reuse. A wide definition which describes the essence of code reuse is: Code is reused when it 1) already exists, and 2) is chosen over the possibility to write new code.

1.3 Goals

The main goal of this work was to produce a set of guidelines to help software developers in small software development departments to reuse code in a more proper way. This main goal consisted of the following sub-goals:

- Get an overview of important issues with regard to code reuse
- Summarize this information and present it in a sensible and easily understandable way
- Get feedback and information from the target group (developers in small software development departments) with regards to:
  - Superfluous or missing information in the guidelines
  - Comprehensibility of the guidelines
  - And ultimately: How useful the guidelines are
- Possibly publish the guidelines in some way

1.4 Research Methodology

To obtain information about important issues of reusing code, I decided to carry out a literature study. The literature study and my own previous project work would be the basis for generating a set of guidelines. This is illustrated in figure 1.1.

As I felt that feedback from experienced developers was important to produce a set of guidelines which would be of any use, I decided to perform two rounds of feedback meetings with software developers from some of the companies I interviewed during my project last autumn. This process is illustrated in figure 1.2. The total time given to complete the thesis was

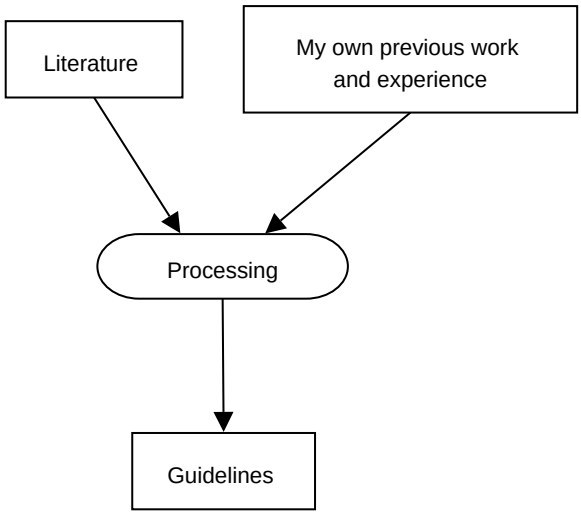

Figure 1.1: Overview of Process - Information Sources

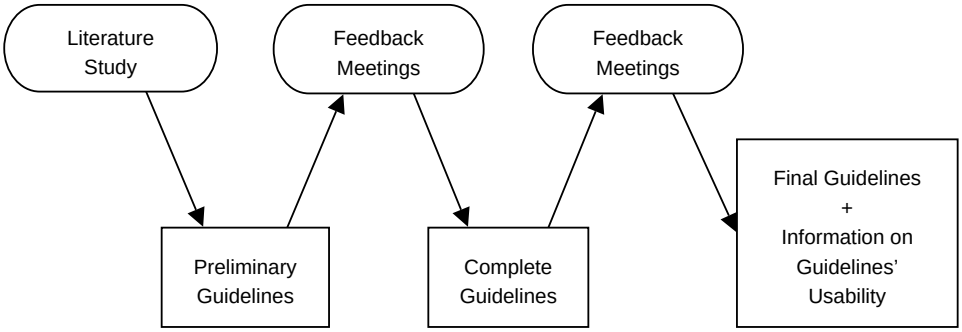

Figure 1.2: Overview of Process - Time

20 weeks. The literature study would be important preparatory work for the rest of the process, and I decided to dedicate about 10 weeks to it. The first few of the remaining 10 weeks I would spend making the preliminary guidelines, while the rest of the time would be spent on feedback interviews and further development of the guidelines. For a description of the detailed plans, see section 5.1.

1.5 Report Outline

The rest of this report is organized as follows:

Literature Study This chapter contains information from the most relevant articles I have read during the literature study, as well as a summary of the state of the art.

Previous Work This chapter describes my previous work on the subject of code reuse in object oriented software development.

Research Method Theory In this chapter, I present research method theory; general theory on research processes, and information about using the interview as a data collection technique.

Making the Guidelines In this chapter, the process of making the guidelines is described: Planning, performing feedback interviews, and writing the guidelines.

Guidelines The content of this chapter is the final version of the guidelines themselves.

Discussion and Conclusion In this chapter, I present a discussion of my work and a conclusion on the usability of the guidelines.

Further Work This chapter contains suggestions for further work.

Chapter 2

Literature Study

In this chapter, I will present the literature I have studied. I will also present a short summary of the state of the art.

2.1 Literature Presentation

This section contains summaries from the most relevant literature I have read, one article at a time. The information in the summaries is the information I felt was the most important and useful to my work. I have grouped the articles by subjects, such as "technology" and "models and strategies". Some articles treat more than one subject, in these cases I have chosen to group the article under the subject which is the most thoroughly treated in the article.

2.1.1 IEEE Standards

The *IEEE Standard for Information Technology – Software Life Cycle Processes – Reuse Processes* (IEEE Std 1517-1999)[2] presents a common framework for extending the software life cycle processes (IEEE/EIA Std 12207.0-1996) to make it include the systematic practice of software reuse. This includes all phases of the software life cycle. The standard specifies processes, activities, and tasks to enable a software product to be constructed from reusable assets. The processes, activities, and tasks for identifying, constructing, maintaining, and managing assets are also specified. An annex to the standard lists types of tools needed to support reuse.

The *IEEE Standard for Information Technology – Software Reuse – Data Model for Reuse Library Interoperability: Basic Interoperability Data Model*

(BIDM) (IEEE Std 1420.1-1995)[3] presents the minimal set of information about assets that reuse libraries should be able to exchange to support interoperability. This minimal set of information is "the information which would enable reuse library users to make quick, intelligent decisions about which assets in other reuse libraries will likely meet their needs".

The *Supplement to IEEE Standard for Information Technology – Software Reuse – Data Model for Reuse Library Interoperability: Asset Certification Framework* (IEEE Std 1420.1a-1996)[4] defines a structure for describing a library's asset certification policy (Asset Certification Framework). That is, the Asset Certification Framework is a technique and an associated data model which is used for organizing, selecting, communicating, and guiding the process of certifying assets.

2.1.2 Success and Failure Factors of Reuse

In *Project-Level Reuse Factors: Drivers for Variation within Software Development Environments*[5], Rothenberger explores the effects of project-level factors in the success of software reuse. He states that "an organization which can successfully identify the factors affecting potential software reuse will be able to better target investments for the improvement of its reuse methodology and thus positively affect its software development productivity and quality." The success factors are sorted into four groups:

- Client influence: Client's budget and time constraints, perceived value of reuse by the client, client's fear of interconnectivity
- Project culture: Degree of promotion/emphasis on reuse by the leader or developing team during project development
- Project attributes: Interaction with other systems, project sequence, project domain
- Developer reuse experience: Experience in recognizing reuse patterns, understanding of the company's reuse model, knowledge of component availability/capability

An analysis showed evidence that the client influence, project culture, and project attributes factors strongly affect the success of systematic reuse projects, while the developer reuse experience had a limited effect on the reuse success.

In the article *Strategies for Software Reuse: A Principal Component Analysis of Reuse Practices*[6], Rothenberger et al. investigate whether the success of software reuse efforts vary with the reuse strategy. Reuse practices are grouped into six dimensions, which cluster into five distinct reuse strate-

gies with different potentials for reuse. The dimensions are Planning and improvement (PI), Formalized process (FP), Management support (MS), Project similarity (PS), Common architecture (CA), and Object technologies (OT). The five strategies have different values in the six dimensions; from 1 (low) to 3 (high). The following list describes the strategies/clusters and their reuse success.

- A: Ad-hoc reuse with high reuse potential: (PI:1 FP:1 MS:1 PS:3 CA:3) *Moderately successful.*
- B: Uncoordinated reuse attempt with low reuse potential: (PI:1 FP:1 MS:2 PS:2 CA:1) *Poorest overall reuse success.*
- C: Uncoordinated reuse attempt with high reuse potential: (PI:2 FP:1 MS:2 PS:2 CA:3) *Slightly more successful than B, due to a common architecture.*
- D: Systematic reuse with low management support: (PI:2 FP:2 MS:1 PS:3 CA:2) *Moderately successful.*
- E: Systematic reuse with high management support: (PI:3 FP:3 MS:3 PS:3 CA:3) *Most successful.*

The article *Success and Failure Factors in Software Reuse*[7] by Morizio et al. identifies key success and failure factors in software reuse. Five high-level factors were shown to be important for a successful reuse program: Top management commitment, introduction of key reuse roles, introduction of reuse processes, modification of non-reuse processes, using a repository, and considering human factors. There is a single attribute over which a company has no control (such as size and application domain) which has an effect on the reuse success: Type of software production. This attribute could be either isolated (the company develops projects which have little or nothing in common) or product family (the company develops a software product that evolves over time, and/or is more or less adapted for each customer). Developing a product family appeared to have a positive impact on reuse success, while the opposite was true for isolated software production. Size of the company did not appear to be a conditioning factor, but indirectly impacts on the ease of achieving top management commitment and its propagation to lower hierarchical levels. Software process maturity is considered to be a useful but not sufficient factor in achieving success. Main causes of failure were the following: Not introducing reuse-specific processes, not modifying non-reuse processes, and not considering human factors. The authors present a decision sequence which highlights issues to consider when starting a reuse program. The authors stress that "the decision sequence merely tries to explain the cases in the data set and does not claim scientific validity as a prediction tool for new cases." There are three main items in this decision sequence, each is presented in the following list.

1. **Reuse potential** Evaluate the reuse potential, which is much higher when similar software products are produced over time (Type of software production = product family). This evaluation is a complex task, because it includes identifying functions likely to be reused and the number of times they could be reused.

2. **Reuse capability** Get commitment of top management to obtain resources and power to change non-reuse-specific processes, add reuse-specific processes (including defining and assigning key reuse roles), address human factors, and set up a repository. When two or more of these issues are not addressed, a failure is likely. A prerequisite is knowing what the processes are, and a small size of the organizational unit and a high process maturity clearly helps in this area.

3. **Reuse implementation** Changing non-reuse-specific processes, adding reuse-specific processes, addressing human factors, and setting up a repository are all activities which have to be addressed through further low-level choices. To make sustainable decisions, the availability of resources in the company (which is related to its size) should be carefully considered.

This article is criticized in the article *More Success and Failure Factors in Software Reuse*[8] by Menzies and Di Stefano, and the authors repudiate the critique in the article *Comments on More Success and Failure Factors in Software Reuse* [9].

Getting to the Goal: Setting Your Sights on Software Reuse[10] by Rhubart argues that it is important to establish specific goals for the level of reuse one wishes to achieve. The goals must be reasonable and realistic. Domain, size, distribution, culture, management and general process maturity of the organization, as well as the size of the application, have an impact on what exactly is reasonable and realistic. The level of interest and enthusiasm of all the people involved is important. A component library which is well-managed and easily accessed is also important to reach the reuse goals. The application domain also has an impact; the reuse can be either vertical (within a domain) or horizontal (across domains). Vertical reuse offers a good reuse potential, but requires either developers with domain-specific experience or a significant learning curve. Horizontal reuse, on the other hand, has a lower reuse potential, but it is more likely that the developers have relevant experience. A reuse repository which provides capability to calculate the reuse level (the portion of the work on a project which is accomplished by reusing components) should be used.

In the article *Managing Software Productivity and Reuse*[11], Boehm introduces a list of the pitfalls most frequently encountered when trying to achieve reuse:

- Simply building a repository of components and assuming it will be used.
- Focusing only on individual components and not on the domain architecture and interface specification.
- Overgeneralizing.
- Low scalability (not planning to scale the solution up).
- Technical obsolescence.

He also presents evidence that reuse works; a proactive reuse strategy by the US Department of Defense could achieve 47 % work avoidance above the normal improvements accomplished with a business-as-usual approach. Finally, he lists critical reuse success factors: Adopt a product line approach, perform a business case analysis to determine the right scope and level of expectation for your product line, focus on achieving black-box reuse, establish an empowered product line manager and stakeholder buy-in, establish reuse-oriented processes and organizations, adopt an incremental approach, employing carefully chosen pilot projects and real-world feedback, use metrics-based reuse operations management, and establish a proactive product-line evolution strategy.

2.1.3 Reuse Models and Strategies

In *Evaluating Software Reuse Alternatives: A Model and Its Application to an Industrial Case Study*[12], Tomer et al. propose a reuse cost model. The model helps identify the basic operations involved in reuse and associates a cost component with each of these. This enables software developers to systematically evaluate and compare all possible alternative reuse scenarios. Their model is based on a model introduced by two of the co-authors (Schach and Tomer), where activities are described along three axes; development, maintenance, and reuse. Thus, the elementary operations are transformation operations (development and maintenance operations) and transition operations (reuse operations).

The article *Assessing the cost-effectiveness of software reuse: A model for planned reuse*[13] by Nazareth and Rothenberger "represents the first step in a series wherein the effects of software reuse on overall development effort and costs are modeled with a view to understanding when it is most effective". The model presented in the article breaks down software reuse costs and contrasts them with the development costs without reuse. The authors computed savings trough reuse, and the results were as follows: Small repositories appear to cost more than they are worth, but the savings attained by a larger repository appear to plateau as the maintenance and search costs

balances the savings. Repositories with small components do not generate savings, as the effort to manage them is larger than the savings generated by reusing them. In larger projects, the savings tend to plateau at higher values than in small projects.

Software Reuse Strategies and Component Markets[14] by Ravichandran and Rothenberger presents three software reuse strategies: White-box reuse, black-box reuse, and black-box reuse with component markets. White-box reuse allows developers to modify the code to suit their needs. This maximizes reuse opportunities, but is also a key source of reuse problems. Black-box reuse avoids these problems by not allowing the developers to modify the reusable components they retrieve. This, however, dramatically reduces the reuse rate. Black-box reuse with component markets (i.e. obtaining components from a marketplace) can increase the reuse rate, as the developers can search from a larger set of components and thereby are more likely to find components fitting their requirements. The authors discuss other advantages and disadvantages of the three strategies, and argue that the latter strategy could be the "silver bullet solution" which makes software reuse a reality and "advances software development to a robust industrial process". They also present a simple decision tree which helps developers decide on which strategy is appropriate for their current situation.

2.1.4 Reuse Technology

In the article *Work-in-Process vs. Finished-Goods: Why a Version Control System is Not a Reuse Repository*[15], Fay specifies the differences between a version control system and a reuse repository; a version control system manages software "work-in-process", while a reuse repository holds completed software ("finished-goods inventory") and should serve as the channel of distribution. The two types of systems should work together to provide an efficient solution for the management of an organization's overall software asset inventory. A reuse repository has to enable the developers to easily find assets, determine their relevance and technical compliance (by access to metadata), and obtain a copy of the asset. The repository should also have functionality which tracks the use of each asset to be able to notify the users of the asset when a new version is provided, and functionality which tracks the savings achieved by reuse.

The article *A Well-Managed Repository*[16] by Harmon discusses the need for a reuse repository and describes the required functionality of such a repository. Significant reuse is not achievable without having a supply of components which are designed for reuse, and developers need to know where to find components and information about their development, functionality, and current use. The article disproves the misconception that developers

can undertake a project and subsequent projects can reuse its components without further adaption; a proper system is needed, and components will have to be rewritten to be more generic. Infrastructure and methodologies needed for rapid, efficient software development is essential, and a well-managed repository is according to the author one of the best signs of the health of an organization's IT process.

2.1.5 General Issues of Reuse

In *Software Reuse*[17], Kremer deals with different aspects of reuse; benefits and impediments to reuse, design for and with reuse, and classification of reusable components. Software reuse is an old idea, but the approach to reuse has predominantly been ad-hoc. Today, complex and high-quality systems have to be constructed in very little time, and this demands a more organized approach to reuse. Benefits of reuse are increased productivity, improved quality, reduced cost, and increased flexibility (designing *with* reuse or *for* reuse when demand is respectively high or low). Impediments to reuse are that few organizations have anything which resembles a comprehensive software reusability plan, few developers use the tools and components which provide direct assistance for software reuse, relatively little training is available to help developers understand and employ reuse, many developers consider reuse to be more trouble than it is worth, and few companies provide incentives to reuse. Design *for* reuse require the developers to perform domain engineering and to develop standards which enables components to generalize well to the application domain. Design *with* reuse could be for example module reuse, toolkit reuse, or application framework reuse. Classification of reusable component should follow a 3C Model; the description of each component should contain information on Concept (what the component does), Content (how the component is realized), and Context (under which circumstances the component is supposed to work). Examples of classification schemes are enumerated and faceted classification.

The Software Reuse Initiative of the Program Management Office of the United States Department of Defense has produced a *Software Reuse Executive Primer*[18]. The primer provides a brief introduction to the concept of software reuse, by treating a multitude of questions related to reuse; such as "why is reuse important?" and "how do I avoid failure?". The following summary of the primer mainly consists of direct quotes from the primer: Software reuse provides a basis for dramatic improvements in increased quality and reliability and in long-term decreased costs for software development and maintenance. Software reuse principles have been demonstrated in isolated examples in industry and DOD to provide one of the greatest returns on investment for reducing cost, time and effort throughout the software life-

cycle. The upfront investments are: Creating a separate group of domain engineers, performing domain analysis, developing architectures, developing and maintaining asset management tools, acquiring tools, and redesigning and implementing the proposal evaluation criteria and process. Metrics, algorithms and processes must be developed and historical and current data collected to properly assess the return on investment. The most common failures when transitioning to reuse are inadequate investment, no domain engineering group, no reward for reuse, no reward for architecture population, project-by-project reuse planning, any reuse is good reuse, and technically inadequate metrics used to determine award fees. Failure can be avoided by rewarding reuse, not alternatives, providing leadership and resources, and by reorganizing to facilitate reuse. Things to consider before developing a reuse program are the size of organization and required infrastructure, investment to support needed infrastructure and mechanisms to recoup investment, and tools to facilitate the transition. You get from no reuse to planned reuse with resources, enforced organizational standards, and rewards for reuse and not alternatives.

2.2 State of the Art

During the search for information for the literature study, I found a lot of information on such things as success factors of software reuse and on studies performed to evaluate the reuse processes already in place in different organizations. I only found a single article, however, which contained something resembling a "recipe" for reuse; Morisio et al. [7] describe a decision sequence which highlights issues which should be considered when starting a reuse program. (See the description of the article for information on the decision sequence.) The decision sequence appears to be aimed at larger software development organizations and departments, and I felt some of the activities were too theoretical or at too high a level for developers in small software developments in Norway to embrace the decision sequence. I wanted to concentrate on making a plan for reuse, which corresponds to one of the activities of the second decision point, Reuse capability: Add reuse specific processes, which was only described in a few lines.

This, combined with the results of my previous work (described in the following chapter), led me to conclude that the state of the art was as follows: *Very few, if any, detailed descriptions of the process of making a plan for reuse exist, but such a process description is desired by small software development departments.*

Chapter 3

Previous Work

This thesis is founded on a project I did last autumn, entitled "Code Reuse in Object Oriented Software Development"[1]. This chapter contains information on the project and its results.

3.1 The Project

The project was performed as a part of the graduate level course "TDT4735 Software Engineering, Depth Study" during the fall semester 2004. The research goals were to get an overview of how common code reuse is, to find out what the desired and achieved effects of code reuse are, and to learn about the used tools and procedures which are specifically developed for code reuse. To reach these goals, a series of interviews were conducted. The interviewees were software developers from 24 software developing companies of different sizes, locations, and application domains in Norway.

3.2 The Results

As previously mentioned, a goal was to find out what the desired and achieved effects of code reuse are. The results regarding effects of reuse are not included here, as it is not relevant for this Master's thesis. Included in the following sections are results concerning the extent of code reuse, the use of tools and procedures, and the organization of code reuse.

3.2.1 Extent of Code Reuse

To indicate the extent of code reuse, I used two factors: Whether the developers used a common base of reusable code which was separated from other code, and how high up in the company's hierarchy the decision to reuse was made. The last factor had three options: Reuse was left up to each developer, there was a development department policy on reuse, or there was a company policy on reuse.

14 of the 24 companies/development departments did not have a shared base of reusable code separated from other code, while 10 did. Nine interviewees reported that reuse was left to each developer. Eight development departments had a policy on reuse, while seven companies had a policy on reuse.

The combination which seemed to be the most interesting was a company policy on reuse, but not a common base of reusable code which was separated from other code. My impression before the interviews was that "serious" reuse would include some sort of a separate reuse storage. A company policy on reuse indicated a serious take on reuse, but the lack of a separate base of reusable code might seem a bit less "serious". Only two companies had this combination of policy and storage of code.

A combination which seemed natural was a lack of policies on reuse and a lack of a common, separate base of reusable code. This occurred in nine companies. Another natural combination was a company policy on reuse accompanied by a separate reuse storage. This was the case for five companies, which seemed to be the most conscious of code reuse. There were eight development departments with policies on reuse. Three of those had no common base for reusable code, while there were five where a common base of reusable code existed.

3.2.2 Use of Tools and Procedures

The tools and procedures in question were ones produced specifically for reuse. There was a single company which stood out with regards to tools. The tool used by this company was developed by the company, and imposed certain procedures on code reuse. The company was international, and was also the only company with more than a thousand employees in Norway alone. The company had more than 15000 employees world wide, and the size of the company explained their need for a formalized system for code reuse.

No other companies had tools which were created specifically for code reuse. Some companies, however, used informal procedures for code reuse. There

were three interviewees who stated that the developers followed procedures for reuse.

3.2.3 Organization of Code Reuse

The organization of reuse was classified according to the models described in appendix B. The company which had developed a tool for reuse again stood out. It was the only company which organized reuse following a slightly advanced model; the company had a separate reuse project. The organization did not completely follow this model, described in section B.2; the reusable code was developed as parts of the projects, not by a separate, temporary reuse project. The organization at the company was, however, more complex than the first model, where the projects identify and develop components and submit them to the reuse storage: There was a separate department which dealt with code reuse in the sense of approval or disapproval of the component as fit for reuse and possibly authorizing investments.

There were no companies following the two more advanced models of reuse organization. 14 companies had no separate storage for reusable code, thus not following any of the reuse models listed in appendix B. As mentioned earlier, I felt this indicated a lack of serious commitment to code reuse. Most of the companies with no separate base for reusable code had no company or department policy of reuse, which strengthened the impression of less commitment to code reuse.

Finally, there were nine companies where code reuse followed model 1; project oriented reuse (see section B.1). These companies separated reusable code from other code in a reuse storage. The reusable code was identified and developed by the projects, and it was stored in a separate reuse storage. This is the least complex way of reusing code in the set of models listed in appendix B, and it was the way I beforehand had assumed most of the companies reused code.

3.2.4 Summary

Code was reused by software developers in all the companies, but there were few companies where tools and/or procedures specifically developed for code reuse were employed. The majority of the companies performed reuse in a disorganized manner, without separating reusable code from other code. The interviewees reported that the achieved effects were the same as the desired effects. The effect most commonly mentioned, was improved development efficiency, i.e. saved time and money. The effect which was second most often referred to, was improved quality of software. Other effects

identified were improved stability of software, simplified testing of software, uniformity of how problems are solved, more accurate time and price estimates, and marketing advantages. It seems that many software developers are aware of the positive effects of code reuse, but lack consciousness as to how these effects best can be achieved.

Chapter 4

Research Method Theory

In this chapter I will present general theory on research processes, and information about using the interview as a data collection technique. The first section deals with research processes, and is mainly a summary of the information included in my project report ("Code Reuse in Object Oriented Software Development" [1]). The second section includes a discussion of using interviews as a method of data collection.

4.1 Research Processes

Ringdal [19] presents the research process as consisting of the steps shown in figure 4.1. This is, however, a simplified picture which hides the fact that more often than not, one has to turn and go back one or more steps. Each step is described briefly in the following list. For a more detailed description, see [1], [20], or [19].

1. **Rough problem definition** The rough problem definition consists of an idea which stems from the researcher's own interests and/or from users or customers.

2. **Research questions** The rough problem definition has to be transformed into research questions, which may take the form of questions or hypotheses. The transformation could be based on previous research and theory.

3. **Choice of design** In this step, a rough sketch of how a specific investigation should be formulated is made. A design is based on several choices, such as whether the investigation should be qualitative (results: Text data), quantitative (results: Quantifiable data), or a

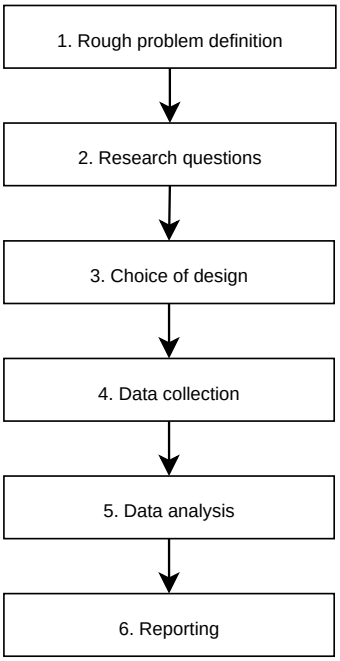

Figure 4.1: Main Steps of the Research Process

combination. The two most common qualitative designs are field observations and conversational interviews, while the two most common quantitative designs are surveys and experiments.

4. **Data collection** The first choice to be made in this step is whether data collection is necessary. Sometimes data has already been collected, removing or reducing the need to collect data. The basic techniques of data collection are:

   - Questionnaire (used in quantitative research)
   - Interview (used in both quantitative and qualitative research. See section 4.2 for a discussion of interviews.)
     - Telephone interview
     - Visitational interview
   - Observation (used in qualitative research)
     - Field observation
     - Laboratory observation

   It is important to make a selection of subjects before data is collected. If it is desirable to have a selection which is representative of the population, the selection has to be performed using statistical criteria.

If, however, only a few cases are to be selected, it is better to make the selection based on strategic reasons.

5. **Data analysis** Before the data can be analyzed, it has to be registered, preferably electronically. How the data should be registered and the degree of detail depends on the type of research and the purpose of the information. The analysis may consist of several methods and techniques, and may vary depending on the design of the study. Ringdal [19] explains that analysis of qualitative data can be difficult, as there are no standardized techniques for this. Fledsberg [20] and Kvale ([21]) present several methods for analyzing data:

   Coding: Categorize the paragraphs of the interview transcriptions.

   Opinion coalescing: Abridge the interviewee's statements, expressing the principal meaning.

   Opinion categorizing: Code the interview in categories, reducing long statements into simple categories (such as "+" or "-").

   Narrative structuring: Organize the text according to time lapse.

   Opinion interpretation: Interpret the text in a deeper, more speculative way.

   Ad-hoc methods: One single standard method is not used. Instead, a free interaction of different techniques is employed.

6. **Reporting** The results may be reported in a journal, report, Master's thesis, dissertation, or book.

4.2 Interviews

An interview is according to Ringdal [19] an exchange of viewpoints between two persons conversing about a subject which interests them. An interview is a specific type of conversation which is often characterized by one person asking questions (the interviewer) and one person answering them (the interviewee). Robson [22] presents advantages and disadvantages of using the interview as a data collection method. The advantages of using interviews are that the researcher has the opportunity to:

- Ask the interviewee directly if he/she has any questions, as opposed to observation, where the researcher is only allowed to observe and interpret events
- Change the direction of the research
- Follow up on interesting answers

- Explore underlying motives in a way which is not possible in self administered questionnaires
- Observe body language in addition to hearing the words spoken by the interviewee

There are two main disadvantages of using interviews as a data collection method:

- The interviews are time consuming
- A lot of preparatory and complementary work is needed, including preparation, organization, notes, and transcribing

Three types of interviews are commonly defined (Seaman [23], Robson [22]):

Structured interview The questions are predefined, and should be asked in a specific order. In a structured interview, the interviewer has very specific objectives for the type of information he/she wants, so the questions may be relatively specific.

Semi-structured interview The questions are predefined, but the order of the questions and the way the questions are asked may be altered during the course of the interview, and the interviewer may ask more or fewer questions than planned. The object of an unstructured interview is to obtain as much information as possible on a broadly defined topic, and the questions should be as open-ended as possible.

Unstructured interview There are no predefined questions, only a general subject of conversation, and the conversation is informal and flows freely.

There are several methods of collecting information from the interviews. Examples are according to Kvale [21] sound recording, video recording, writing notes, and using one's memory. The choice of method is based on the purpose of the researcher. An open-ended study requires a higher degree of detail in notes, as any information could turn out to be relevant. The most common choice is therefore to use sound recording.

When the interviews are completed, speech is transformed to text. The most detailed method is to copy the interview word-for-word, and in addition make detailed descriptions of the setting and the interviewee. This is, however, highly time consuming; Robson [22] states that transcribing an interview would usually take about 10 times the duration of the interview itself. There are other ways of transcribing interviews, depending on the purpose of the interview; if the purpose of the interviews is to obtain a general impression of the interviewee's views, his/her statements may be condensed and rephrased. Such a low degree of detail enables the use of for example a point list.

Chapter 5

Making the Guidelines

This chapter contains a description of how the guidelines were made. The first section covers the planning, while the second section covers organizing the feedback interviews. The next section describes how the first edition of the guidelines were made. This is followed by a section describing the first round of feedback interviews. The next section contains information as to how the second edition of the guidelines were made, followed by a section describing the second round of feedback interviews. The last section describes the final changes made to the guidelines.

5.1 Planning

The research methodology is briefly described in section 1.4. The process I had decided to go through is depicted in figure 1.2. First, I would perform a literature study. Then I would start creating the guidelines, and perform the first round of feedback meetings. I would then revise the guidelines, followed by a second round of feedback interviews, before a final revision of the guidelines. I decided that the literature study should be completed in about 10 weeks, leaving another 10 weeks to make the guidelines and perform the feedback interviews. I wanted to perform the feedback interviews in person at the interviewees' workplace, as this would be the easiest way to communicate freely (as opposed to for example a phone interview). Initially, I had a desire to publish the guidelines in some way, but while planning, I realized that I would not have the time to work on publishing the guidelines.

According to the research method theory in chapter 4, there are six steps to go through when performing research. I planned to do this as follows:

1. **Rough problem definition** This is described in section 1.2, Problem Definition.

2. **Research questions** This corresponds to the third item in the list of goals in section 1.3, Goals: I wanted to get feedback and information from the target group (developers in small software development departments) with regards to:
   - Superfluous or missing information in the guidelines
   - Comprehensibility of the guidelines
   - How useful the guidelines are
3. **Choice of design** This is roughly described in section 1.4, Research Methodology: I wanted to perform two rounds of feedback interviews with software developers from some of the companies I interviewed during my project last autumn, as illustrated in figure 1.2. This meant that the research would be qualitative; the data collected would be descriptive, and the focus would be on the interviewees' perceptions and experiences. The interview type would be visitational.
4. **Data collection** I would perform visitational interviews and collect information by writing notes and sound recording. More information on the further planning of the interviews is given in section 5.1.3, Feedback Interviews.
5. **Data analysis** To register and analyze the data from the interviews, I decided to use opinion coalescing (abridging the interviewee's statements, expressing the principal meaning) which I would do in part during the interviews by writing notes, and in part after the interviews by noting additional key points while listening to the sound recordings from the interviews. In addition, I would use a form of opinion categorizing. I would send the interviewees a list of questions before the meetings, and using these questions as categories, I would reduce long statements made by the interviewees into shorter answers to the questions.
6. **Reporting** The results would be reported in a Master's thesis.

5.1.1 Literature Study

During the project work last fall, I had come across many interesting articles. These articles contained references to other articles, which might also be useful to include in the literature study. I searched the university library's branch databases[1], which led me to such databases of electronic periodicals as Jstor, Association for Computing Machinery (ACM), and Springer. I also

[1] "fagdatabaser"

used the library's system BIBSYS, which provides access to the library's resources through several databases. The university subscribes to a plethora of scientific periodicals on-line, and I found the IEEE database search tool (IEEE Xplore) particularly useful. The IEEE database contains the publications of the IEEE, such as IEEE Software and IEEE Transactions on Software Engineering, as well as the IEEE standards. When searching, I used search terms such as "reuse", "software reuse", "code reuse", "reus*" and the like. This provided me with a long list of possibly relevant literature, a lot more than I could possibly manage to read. So I looked at the abstracts and summaries of the articles and made a prioritized list of the articles I wished to read. I intentionally included more articles in the list than I would have time to read, because I felt they might be useful to use as references in the guidelines even if I had not read the entire article. In the list I included some articles about software reuse in general, as well as articles discussing specific issues of reuse, such as success and failure factors, component storage, and measurement, as well as a few IEEE standards. A description of the most relevant literature is given in chapter 2.

5.1.2 Writing Guidelines

I wanted to work on the guidelines in three phases; before any feedback interviews were performed, after a first set of feedback interviews, and after a second set of feedback interviews. During the first phase of writing, I wanted to concentrate on the overall process of making a plan for reuse. During the second phase, I would add in-depth information about the process. The last phase of writing would consist of making minor changes after the second set of feedback interviews. The first round of interviews would enable me to get feedback on the overall process; does the process seem reasonable and understandable? Later, I would get feedback on the in-depth information as well as the overall process; does it seem relevant and useful?

5.1.3 Feedback Interviews

From the project work last autumn, I had a list of companies and interviewees. I decided to ask some of these interviewees to contribute, as my desire to create the guidelines was based on the results from the interviews with these people. Also, I believed that using the same interviewees would save some time: I already had information about the software development at their companies, I had their contact information, and I believed it would be easier to get the previous interviewees to contribute than to ask a set of new people who knew nothing about my work. From the list of the previous interviewees, i selected the ones fitting the following criteria: First, I wanted

to get information from developers working in small software development departments, as I believed they were the ones who needed such a set of guidelines the most and at the same time had the least time and money to spare for finding this kind of information themselves. I defined "small" as less than 10 developers. Second, I wanted the companies to be located at a place I would be able to visit relatively easily. I felt that performing interviews with several companies in the same city would also be advantageous when it came to traveling time and costs.

There were many cities and towns on the interviewee list, but the two cities with the highest number of interviewed companies were Oslo and Trondheim. These two cities would also be easy for me to visit. I decided to perform one of the two feedback stages in Oslo and the other in Trondheim. To get as much information as possible from the interviews, I wanted to spend about one hour on each of them. I decided that I would interview developers at six companies to be able to get enough information without getting more than I could handle. I would perform three interviews in Oslo and three interviews in Trondheim. I made a prioritized list of the companies in each city, where I preferred the companies with the smallest development departments. I made a plan of what I would say when I contacted the potential interviewees and decided to send an e-mail to the developers who agreed to be interviewed. This e-mail would contain not only the guidelines, but also a set of questions I wanted the interviewees to answer. In this way, they would be able to prepare for the interview, and they would know what I was expecting from them.

When I had completed the planning and was in the process of studying literature, I wanted to know which developers I would be able to interview and when I could visit the company to perform the interview.

5.2 Organizing Feedback Interviews

I selected three weeks during which I would perform the interviews, and started contacting the previous interviewees. I started at the top of the prioritized list for each city, and called the interviewees asking them to contribute. I told them that I had continued the work from last fall's project, and that I needed some real-world feedback to assess the usefulness of a set of guidelines for code reuse in small development departments. I explained that I would send them a draft of the guidelines together with a list of questions which I would like them to answer, so they could prepare for the interview. I only needed to call six people to get six interviews; everyone agreed to be interviewed. The Oslo interviews would be performed first, and the Trondheim interviews would be performed the ensuing week. This left

me little time to revise the guidelines between the first and second round of feedback interviews, but it was the best way to organize the interviews.

One of the interviews (the second interview in Trondheim) was performed at a company where they were using a new system for software development, which included some tools and procedures for code reuse. I was pleased with this, as the experiences of the interviewee at this company probably could be of valuable help – the interviewee might already have been through a process somewhat similar to the one I was trying to formulate.

While I was preparing for the interviews, I decided that I would try to keep the interviews as informal and open as possible, to get as much information as possible on shortages or redundancies, and on the understandability and usefulness of the guidelines. I believed that if the interviews were too strict, I might get less useful information – the interviewee could have many thoughts and opinions, but they might not be expressed. Also, creating an open atmosphere would allow the interviewee to admit it if he/she did not understand something, and thus allow me to become aware of shortages et cetera which caused confusion or made the guidelines hard to understand.

5.3 Writing – First Edition

While writing the first edition of the guidelines, I concentrated on the basic structure of the process; which activities were needed, and in which order. The logical flow of the process was important.

I started by organizing the information from the literature study by categories such as "reuse repository", "success factors", "measurement", et cetera. I then started drawing up a diagram of the activities that I felt were necessary in the process of making a plan for reuse, drawing lines and arrows between them to envision the flow of information between the activities. The diagram was crude and hand-drawn, but it gave me an overview of the process with its activities and information flow. The diagram is included in figure 5.1. The overall activity was to *make a reuse plan*. This activity would consist of several smaller activities; *classify*, *decide on desired results/goals/effects*, *decide on measurements*, and *decide on reuse policy/rules*. All of the activities would have outputs which could be used as input for other activities. The activity of classifying (the organization which wants to reuse code) would produce a *classification*. This classification would be used in the next activity; deciding on desired results, which would consist of two smaller activities; *make ideal goals* and *make final goals*. The ideal goals from the first sub-activity would be used together with the classification as input to making a set of final goals. The set of final goals would be the output of the activity of deciding on desired results. The fi-

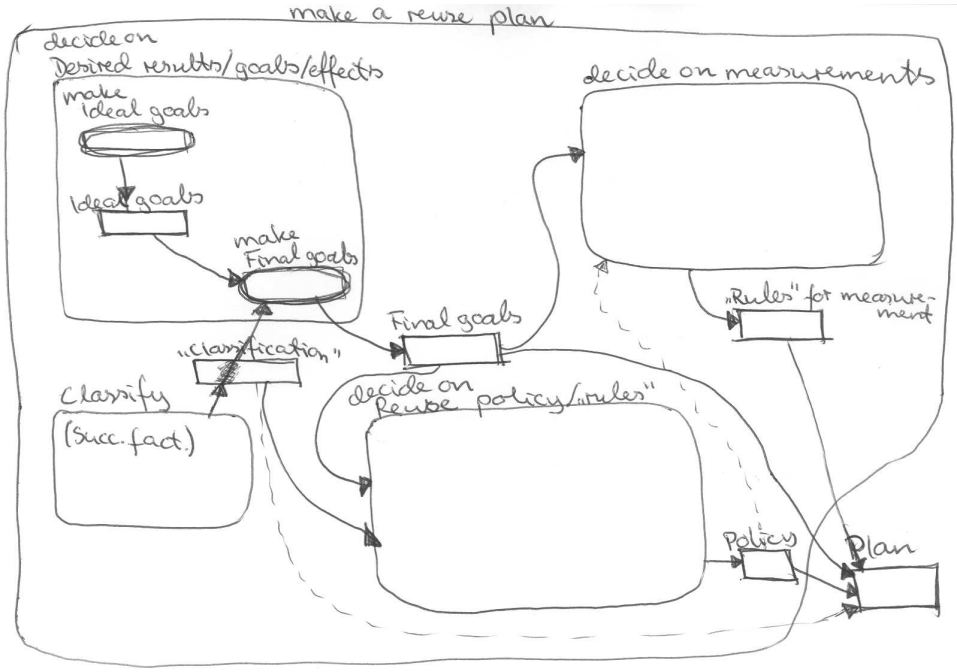

Figure 5.1: Overview of the Process of Making a Reuse Plan – Draft 1

nal goals would be used both in deciding on measurements and deciding on a reuse policy. The activity of deciding on measurements produced a set of rules for measurements, while the activity of deciding on a reuse policy would result in a reuse policy. This reuse policy might also be used as input for the activity of deciding on measurements. All the outputs would be put together to constitute the overall reuse plan, except perhaps the classification. The following paragraph explains what I envisioned that the activities would involve.

The activity *classify* would involve classifying the organization/software development department which is going to make a reuse plan. I wanted to include this activity because it could give the developers valuable insight as to how easy it would be for them to reuse; classifying the reuse organization would help them to gain an understanding of their reuse potential. Since the knowledge and understanding coming from this activity is useful in all the other activities, it is the very first activity. *Decide on desired results* would be the activity where the goals of reusing would be addressed: What do we want to get out of this (i.e. reusing code)? This activity would, as previously mentioned, consist of two sub-activities. In one sub-activity the developers decide on a set of ideal goals: "In a perfect world, we wish to accomplish this and that by reusing code". In the second activity, those ideal goals are moderated to fit their reuse potential: "OK, the world is not

perfect, and we don't have a very high reuse potential, so let's lower our expectations a bit." I put this as the second activity because knowing what your goals are is important through the rest of the process; if you do not know why you are reusing and what you want the results of reuse to be, it is difficult (if not impossible) to make a good plan for reuse. The activity *decide on reuse policy* would be the most extensive activity. In this activity, the developers would decide on rules for reuse, such as whether they wish to use a reuse repository or simply carry on using their version control system, and whether the reuse should be white-box or black-box (i.e. whether the developers reusing the component are allowed to change the component or not). I included this activity as number three, as I believed that the policy could and should have an impact on the next activity: In the activity *decide on measurements*, the developers would decide whether they wish to measure their reuse efforts or not, and if so, make a plan for measuring. Measurement can be a great means of discovering to what extent your goals have been met, but it can also be quite expensive both in time and money.

A few revisions later, the diagram was more clearly set out and re-drawn using diagram software, see figure 5.2. There are four major differences

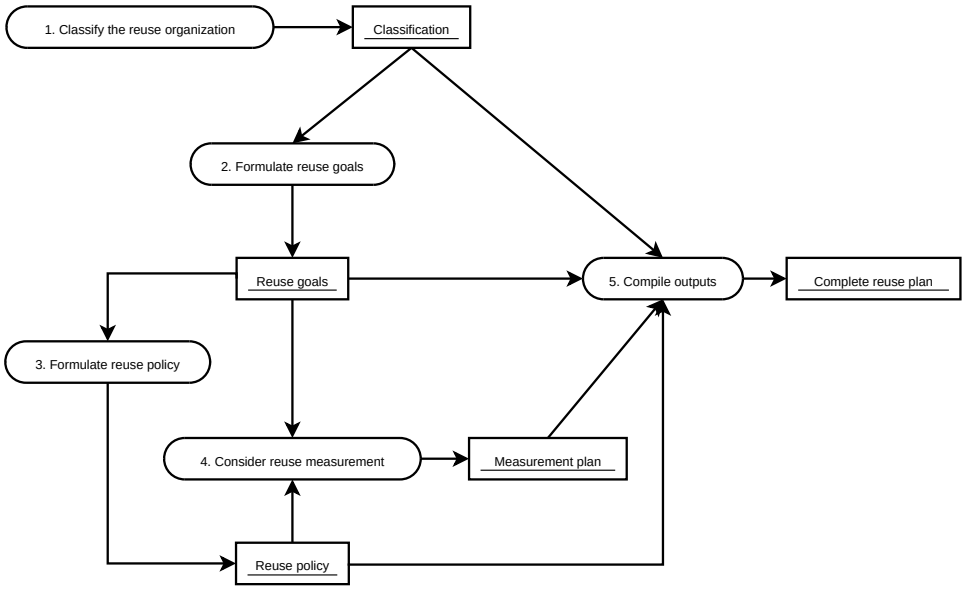

Figure 5.2: Overview of the Process of Making a Reuse Plan – Draft 2

between this version and the hand-drawn one. First, I decided not to include the activity of making a set of ideal reuse goals before deciding on the final reuse goals, as this would probably be superfluous. Second, I felt that the reuse policy should be used as input when making the decisions about reuse measurements, as the policy could lay constraints on the measurement process and some measurements could be made to check whether the policy

was being followed. Third, I decided to include the classification in the complete reuse plan. It would be useful to have this information available, as the information is interesting in itself, and as the rest of the process is at least in part based on this information. The fourth change was that I included a separate activity for compiling the outputs from the first four activities.

The second diagram was still not very easy to follow, so I rearranged the activities and outputs to get a seemingly simpler diagram; the diagram shown in figure 5.3.

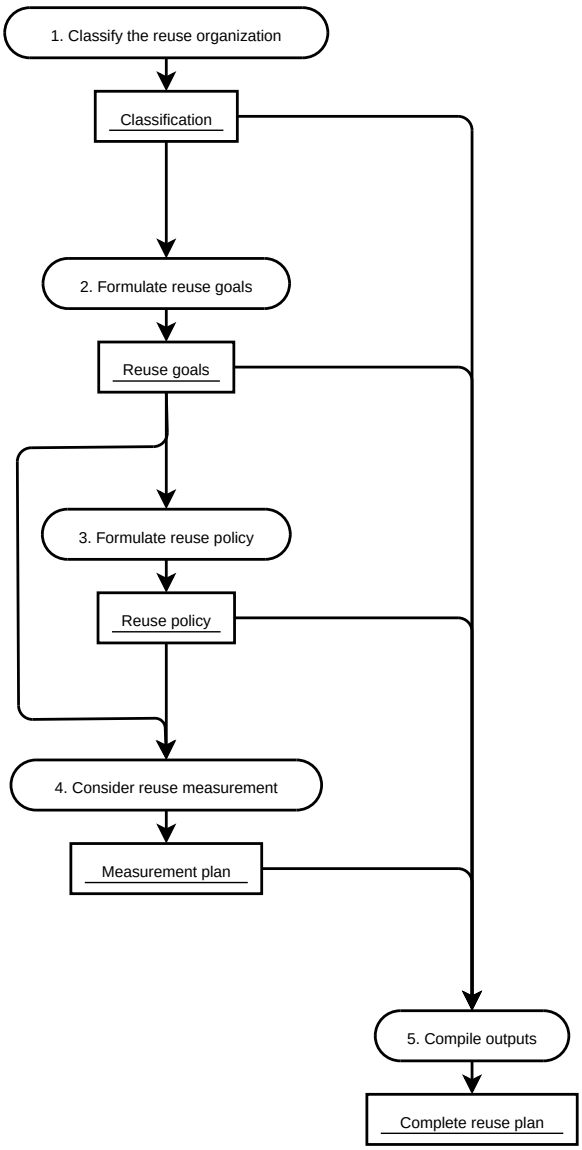

Figure 5.3: Process Diagram for the First Edition

Based on the two last versions of the diagram, I started writing an introduction and a description of the process overview. I wanted this first version/introduction to be short and easily understandable, as I wanted the interviewees' feedback on the overall process. Therefore, I simply wrote a short introduction, followed by an overview of the process; the diagram and a list of the activities with a short description of each activity. The first version of the guidelines is included in appendix F. During this work, I also started writing in-depth descriptions of the activities. As this first phase of writing predominantly was about arriving at an overall process description, the discussion of the in-depth activity descriptions is omitted here, and is instead included in section 5.5.

5.4 Interviews – Feedback 1

The first round of interviews were performed at three companies in Oslo. A few days before the meetings, I sent the interviewees an e-mail containing the introductory version of the guidelines (see appendix F) which they would read before the meetings and a list of questions which I would like them to think about before the meeting. The e-mail text is included in appendix E.

The meetings took place at the interviewees' workplaces. To avoid directing a lot of my attention toward taking notes during the interviews, I wanted to record the interviews with my MP3 player. But I still made brief notes on a printed copy of the guidelines version which I had sent to the interviewee, both to include information or notes which would not be evident from the recordings and to ensure I had at least a little bit of information in case the MP3 player would malfunction during the interview. I started each interview by asking if it was OK that I recorded the interview, and none of the interviewees had any objections.

As planned, I tried to keep the situation rather informal and relaxed, and I let the interviewees talk about what they wanted. Most of the questions I had e-mailed to the interviewees were answered without me having to ask the questions. When there was a pause in conversation or at the end of the interview, I asked the remaining questions. Information about the interviewees, their companies, and the interviews is included in appendix H, with a summary of my notes written during the interviews, additional key points from the recordings, as well as the interviewees' answers to the e-mail questions. The following list contains a summary of those answers:

1. The interviewees were in general left with an overview of what the process of making a plan for reuse involved, but the diagram showing the activities of the process caused some confusion. See the list of results below.

2. None of the activities were considered by the interviewees as being superfluous. An interviewee mentioned research as a possible addition to the activities.
3. The activity of formulating a reuse policy was considered to be the most important and most interesting activity in the process of making a plan for reuse.
4. The interviewees all said that the final set of guidelines could be useful to them and their development departments, predominantly to use as a reference and to actuate a thought process regarding code reuse.

The results of the first three feedback meetings which I considered were the most important for my further work with the guidelines are included in the following list.

1. The diagram showing an overview of the process was not as simple as I had believed it to be. I had overlooked a very important issue: How does the flow appear to go through the diagram? To me it seemed clear, I felt the numbering should at least indicate the flow of activities. But it became evident that the interviewees did not interpret the diagram as I had intended. Two things were pointed out:
   - The arrow from output 2. Reuse goals to activity 4. Consider reuse measurement was mis-interpreted as indicating that you could go right from activity 2 to activity 4, without performing activity 3. Formulate reuse policy.
   - Outputs, i.e. documents resulting from an activity, are obvious. There should always be output from an activity, and therefore it is unnecessary to explicitly include the output in the diagram.
2. Examples are important to aid understanding. A “Toy Store” was suggested; make up a story about one or several development departments going through the process of making a plan for reuse.

5.5 Writing – Second Edition

During the first phase of writing, I had started developing the in-depth descriptions of the activities in the guidelines. During this second phase, I continued that work. I also worked with an issue from the feedback meetings: Improve the diagram which was causing confusion. As there was little time to complete the second stage of writing, I decided not to include a “toy store” or in-depth examples in the second version of the guidelines, although this was generally considered by the interviewees as useful.

To avoid confusion about the diagram, I decided to make it simpler. I split it into several diagrams:

- Activity diagram. This diagram showed an overview of the activities included in the process of making a reuse plan, as well as the process flow. The activity diagram is included in figure 5.4.
- Output diagram. Showing only the outputs from each activity (numbered according to its parent activity), this diagram illustrated that the overall reuse plan is a collection of the outputs from all the activities. The output diagram is included in figure 5.5.
- Complete diagram. This diagram connected activities and outputs, showing which outputs were needed as input to which activities. The complete diagram is included in figure 5.6. There are solid lines without arrows between the activities and their resulting outputs. From each activity there is a solid, bold line with an arrow pointing to the following activity. From each output there are dotted lines with arrows pointing to the activities in which the output is needed.

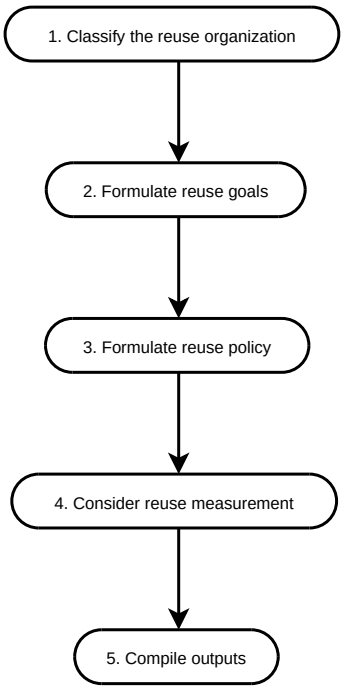

Figure 5.4: Activity Diagram for the Second Edition

In the following sections, I will explain what kind of information I included in the description of the activities and why. I have dedicated one section to each activity:

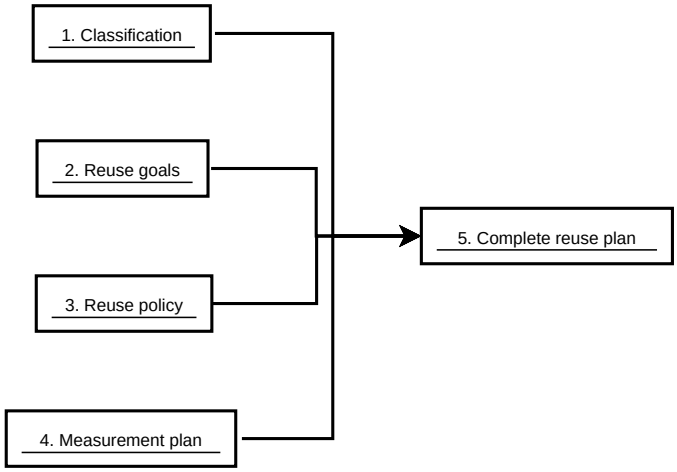

Figure 5.5: Output Diagram for the Second Edition

1. Classify the reuse organization, section 5.5.1.
2. Formulate reuse goals, section 5.5.2.
3. Formulate reuse policy, section 5.5.3.
4. Consider reuse measurement, section 5.5.4.
5. Compile outputs, section 5.5.5.

I decided to make the first part of the activity description similar for all the activities. First, I stated the goal of the activity, followed by a list of the inputs to the activity, and a short description of the output.

As I, partly based on the interviews, considered the subject of tools for reuse to be interesting, I included a separate section discussing tools for reuse. An explanation of what kind of information I included in this section of the guidelines and why is given in section 5.5.6.

The in-depth second version of the guidelines is included in appendix G.

5.5.1 Organization Characteristics

The first activity, Classify the reuse organization, would involve classifying the organization/software development department which is going to make a reuse plan. This activity would help the developers gain an understanding of the organization's reuse potential. It should be simple and not take a lot of time, so I wanted the description of this activity to be relatively short and to the point. In the literature study, I found several discussions of success and failure factors of software reuse. Some of these success factors were

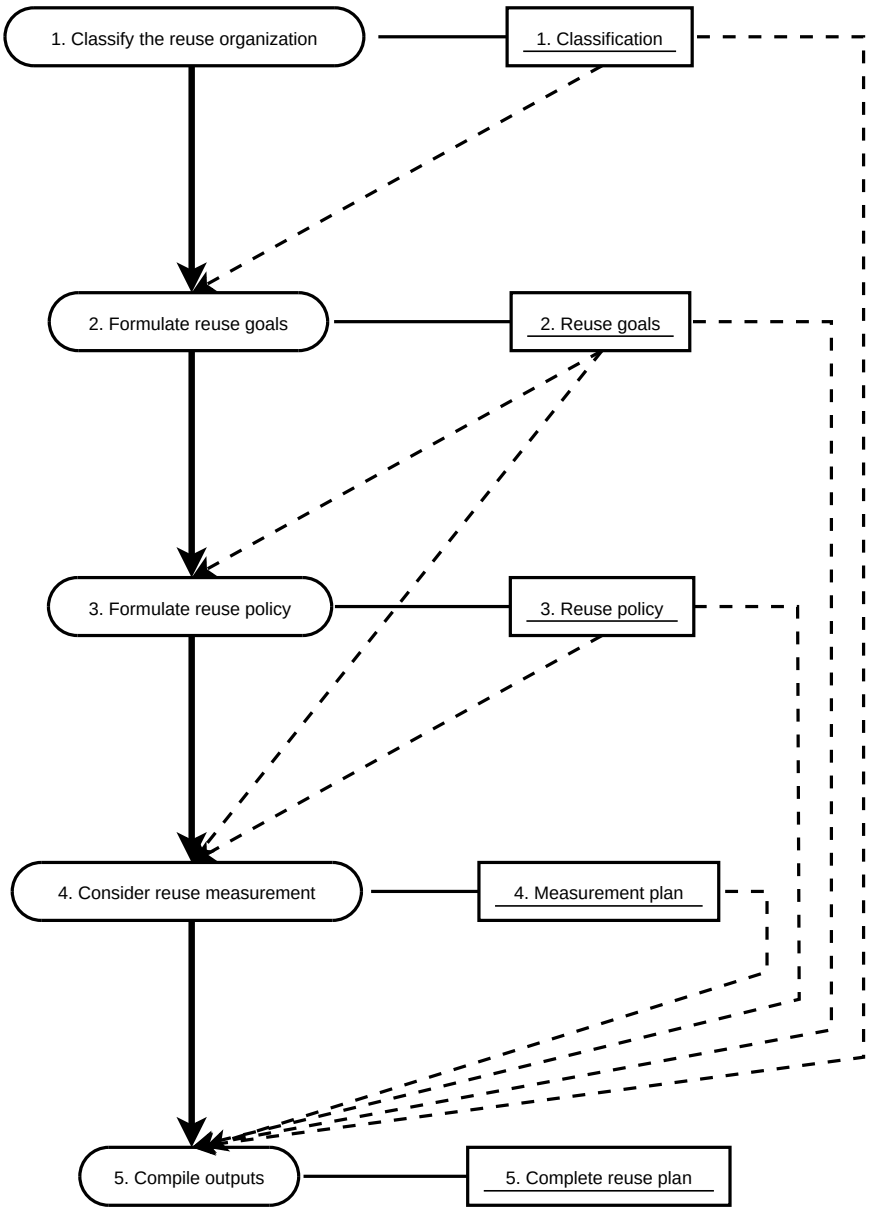

Figure 5.6: Process Diagram for the Second Edition

things that can be changed from project to project, while a few factors were more constant. I felt that these constant factors would be good indicators of an organization's reuse potential.

Morisio et al. [7] used several so-called state variables to describe a company: Size of software staff, Size of overall staff, Type of software production, Software and product (standalone, part of product, or part of process), Software process maturity, Type of application domain, Type of software, Size of baseline, Development approach (object oriented or procedural), and Staff experience. Their study showed that amongst these state variables, only Type of software production could be shown to have a direct effect on the reuse success of the 24 projects they studied. Type of software production in the study was either "isolated" (the company has projects which have little or nothing in common) or "product family" (the company develops a software product that evolves over time, and/or is more or less adapted for each customer). None of the cases with isolated software production were successful, while most of the cases with a product line were successful. Size also has an impact, although indirectly. Size affects the ease of achieving top management commitment, as well as the ease of communication of information and the ease of building a consensus for the reuse program. In this case, smaller appears to be better. The companies considered in their study had reasonably mature processes, and the researchers assumed that software process maturity is a useful but not sufficient factor in achieving success.

Nazareth and Rothenberger [13] state that

> [...] several researchers have noted that reuse is facilitated by well-defined and narrow domains [...] A domain is typically characterized as a set of information systems that possess similar functionality and share the same underlying data. Typically a domain will address related processes and involve a limited set of users.

This indicates that definition and narrowness of domains is a useful characteristic when assessing the reuse potential of an organization.

To summarize, here is a list of the aforementioned characteristics which are useful when assessing the organization's reuse potential.

Type of software production There are two main categories; Product family and Isolated. Product family is when the company develops a software product that evolves over time, and/or is more or less adapted for each customer. Isolated is when the company has projects which have little or nothing in common. In this case, Product family is the better alternative.

Size of organization/software staff Smaller is better when it comes to the ease of achieving top management commitment, as well as the ease of communication of information and the ease of building a consensus for the reuse program.

Software process maturity As high as possible, since a software development department with high process maturity will be better equipped to generate a plan for reuse and, equally important, implement the plan.

Domains As well-defined and narrow as possible, because well-defined and narrow domains facilitate reuse.

5.5.2 Goals

The second activity, Formulate reuse goals, would include addressing the goals of reuse: What do we want to get out of reusing code? Clarifying your expectations makes it easier to develop a good reuse plan for achieving your goals. Nazareth and Rothenberger [13] state that "Reuse programs are likely to be adopted more readily, and with greater conviction, if a clearer understanding of the outcomes of the reuse program were available". As I felt an exhaustive list of all possible types of goals would be too much, I decided to provide the developers with a starting point for formulating their own reuse goals. Through my previous work, interviewing software developers, I had developed a list of positive effects of reuse which the developers mentioned during the interviews:

- Increased efficiency, i.e. saved time, and thereby money
- Improved quality of software
- Improved stability of software
- Simplified testing
- Uniformity: Of how problems are solved, of appearance and functionality
- More accurate time and price estimates
- Marketing advantages; experience and ease of software development

Increased efficiency is a positive effect of reuse which was mentioned by Rothenberger et al. [6] and Nazareth and Rothenberger [13], as well as almost every other article mentioning positive effects of reuse. Improved quality of software was also mentioned in the two aforementioned articles ([6], [13]), and in most of the other articles I read. Anderson [24] mentions standardization, with the following example: "Reuse of UI widgets in MacOS

and Win32 leads to common "look-and-feel" between applications". He also mentions debugging as an activity which profits from reuse; the code which is reused has been tested before. Lastly, Anderson states that "Reuse can lead to a market for component software", and lists some examples: ActiveX components, Hypercard stacks, Java packages, and software tools, such as xerces and xalan from xml.apache.org.

To summarize, this is the list of positive effects of reuse I included in the guidelines:

Increased efficiency Reductions in development cost and time, increased programmer productivity

Better software Improved quality, stability, and maintainability of software

Standardization Uniformity of how problems are solved, of appearance and functionality

Simplified testing Less debugging, reused code is often tested code

Profit Selling reusable components to other companies

In addition to using these positive effects as a basis for formulating goals (which are not easily measurable), it could be useful for the developers to set measurable goals as well, to be able to compare their actual results with their predetermined goals. An example of this is mentioned by Rhubart [10]: The percentage of the work on a specified development project that is accomplished through the use of existing code.

5.5.3 Policy

The activity of deciding on a reuse policy would be the most extensive and, in my eyes, the most important activity. This is the activity where the developers formulate their rules for reuse, such as whether they want to use a separate reuse repository and whether they want to perform white-box or black-box reuse (that is, whether the developers who reuse the components are allowed to make changes to the component or not).

I had reviewed a massive amount of information on issues which I wanted to include in the description of this activity. I tried to sort the bits of information into categories. First, I wondered if I should just go through the issues and discuss them one by one, but it appeared to be too untidy. From an article I read (Rhubart [10]), I got the idea to sort the information into answers to questions. For example, the issue of employing a reuse repository would fall into the category of "where" – "where should the reusable

components be stored". I generated three other categories, "what", "how", and "who":

- *What* kind of reuse do you wish to perform?
- *How* should the reusable components be classified, enabling them to be found and retrieved later?
- *Where* should the reusable components be stored?
- *Who* is responsible for what?

I wanted to discuss these four questions in four subsections of the policy section. What kind of information I wanted to include in each subsection and why, is discussed in the following sections.

What

What kind of reuse do you wish to perform? I chose to present two main decision points in this section. The first one was the aforementioned question of whether to perform black-box or white-box reuse. The other decision point was whether the reuse would be vertical or horizontal. I regarded the first question as the most important one. Should the developer who uses the reusable component be allowed to make changes to it or not? One could write a whole book on this subject, but I wanted the discussion to be short, while still providing the most important information. There was one article in particular which provided me with useful information here; Ravichandran and Rothenberger [14]. The article describes three reuse strategies: White-box reuse, black-box reuse with internal components, and black-box reuse with component markets. It compares features and discusses advantages and disadvantages of each strategy and also includes a reuse decision tree which "depicts the various reuse strategies and the questions developers must answer in choosing each strategy. The model also depicts the cost equations that help developers answer these questions". I considered the article to be so useful that I, in addition to including in the guidelines a brief summary of the main points of the article, wanted to advise the developers to read the whole article themselves. (See the full second version of the guidelines in appendix G for the summary.)

The second question was whether the reuse would be vertical or horizontal, i.e. within a specific domain or across several domains. This is not always a decision the developers can make themselves, as it partially depends on the type of software production of the company (i.e. isolated or product family, as discussed in section 5.5.1 Classifying the reuse organization). I wanted to tell the reader this, as well as provide information on some of the

advantages and disadvantages of each approach. I used Rhubart [10] as my main source of information regarding this issue.

How

How should the reusable components be classified, enabling them to be found and retrieved later? In order to reuse efficiently, it must be possible for the developers to search for and find reusable components. That is, the reusable components have to be organized in some way, and classification of the components is essential. I regarded the information in the IEEE standard "IEEE Standard for Information Technology – Software Reuse – Data Model for Reuse Library Interoperability: Basic Interoperability Data Model (BIDM)" (IEEE Std 1420.1-1995)[3] as a good starting point for developing a classification scheme, as the standard describes "the minimal set of information about assets that reuse libraries should be able to exchange to support interoperability". The minimal set of information is "the information which would enable reuse library users to make quick, intelligent decisions about which assets in other reuse libraries will likely meet their needs". This set of information is not only useful for library information exchange; information enabling developers to make quick, intelligent decisions is just as useful when it comes to the company's own reuse library.

Kremer [17] states that an ideal description of a software component encompasses a so-called 3C Model: Concept, content and context. That is, the description of a component should contain information about what the software does (concept), how the component is realized (content), and under which circumstances the component is supposed to function (context). I included this 3C Model in the discussion, as well as a description of the faceted classification scheme described in the same article.

Where

Where should the reusable components be stored? A separate reuse repository was mentioned as important in several of the articles I read, and in many other articles, it was taken for granted. I wanted to convey to the reader the importance and usefulness of using a separate reuse repository. The article written by Fay [15] explains in an easily understandable way how a reuse repository is different from a version control system, and why a reuse repository should be used. I used Fay's article as a basis for the discussion of where reusable components should be stored.

Even though I personally felt that a reuse repository was the only fitting option for proper code reuse, I wanted to mention the other available op-

tions; storing reusable code and non-reusable code together (i.e. no system for reuse), or storing reusable code separated from non-reusable code (i.e. a reuse library), but in the same system (for example a version control system). A reuse repository system could be expensive and everybody would not be able to afford it, so the other available options would probably be used more often than a separate reuse repository. I also wanted to mention the organization of the components (naming and hierarchy conventions), as this becomes even more important when there is no specialized reuse system. Finally, I wanted to remind the reader that he/she would have to give some thought to how the reuse repository/library should be populated; with already existing components, or with only new components (specifically developed to be reusable).

Who

Who is responsible for what? It is always useful to establish roles and responsibilities. This helps to ensure that things get done; it is a lot harder to avoid doing a task which was especially assigned to you than to avoid doing a task which "somebody should do". It also ensures that there is a single person who can make the ultimate choices, which is especially useful when there are disagreements between the developers. In small software development departments, however, the benefit and need of many roles are relatively small. Therefore, I decided to describe the two roles I regarded as the most important and useful; reuse program manager and library manager. These and other roles are named by Morisio et al. [7].

5.5.4 Measurement

During the activity of deciding on measurements, the developers would decide whether they wish to measure their reuse efforts or not. If they decide to measure reuse, they make a plan for the measurements during this activity. The reason I did not want to present measurements as mandatory, is that while measurement can be a good way of discovering to what extent you have reached your (measurable) goals, it can also be quite expensive. For example, using a measurement such as lines of code would be near to impossible without a tool. Such a tool could be expensive, to the degree that the developers would choose not to buy the tool. In this way, cost (in money or time) could inhibit the measurements ever being made. Automated tools are important aids for measuring reuse, and a good reuse repository tool would enable the developers to make a range of measurements of their code reuse.

While measurements can be very useful, and the developers reading the

guidelines probably would want some practical information on metrics and performing measurements, I felt that these subjects were too complex to be appropriately covered in the guidelines. Therefore, I wanted to achieve two things by the description of this activity: I wanted the reader to understand that measurement is important and useful, and I wanted to give the reader references to useful information about software/reuse metrics and measurement. To advocate reuse measurement, I chose to include some quotes from the articles I had read. I felt that the following list of quotes conveyed the importance of measurement:

Tomer et al. [12] (citing Poulin [25]) "it is widely accepted that the organizational challenges of software reuse outweigh the technical ones. As a result, metrics are needed in order to "make business decisions possible by quantifying and justifying the investment necessary to make reuse happen"[25]"

Nazareth and Rothenberger [13] "Reuse programs are likely to be adopted more readily, and with greater conviction, if a clearer understanding of the outcomes of the reuse program are available"

Boehm [11] includes in his list of eight critical reuse success factors the following point: "Use metrics-based reuse operations management". He goes on to explain that this is important for "tracking progress with respect to expectations and making appropriate adjustments where necessary".

Information on software metrics abound, and I chose to present the reader with the following list:

- Poulin: Metrics for Object-Oriented Reuse [26]
- Poulin: Measuring Software Reuse: Principles, Practices, and Economic Models [25]
- Frakes and Terry: Software Reuse: Metrics and Models [27]
- Devanbu et al.: Analytical and Empirical Evaluation of Software Reuse Metrics [28]
- Pfleeger: Measuring Reuse: A Cautionary Tale [29]

5.5.5 Compilation

The last activity, compilation, would simply involve gathering the output from the preceding activities to make the final, complete plan for reuse. The first section of the plan would contain a description of the organization, the second section would contain a description of the goals which the developers

wish to accomplish by reusing code, the third section would probably be the most extensive section, containing a description of the rules of reuse, while the fourth section would contain a plan for measurement if such a plan was made during the execution of the fourth activity.

The plan could be stored and made available in many ways, e.g. electronically in some predetermined format such as Portable Document Format, PostScript, Word document, even as a simple text file, or on paper. I believed most developers would have their own preferences as to how the plan should be stored and made available, so I decided not to discuss this any further in the guidelines.

5.5.6 Tools

This section of the guidelines would provide some general information on tools for reuse. I did not want to "advertise" for any specific tools/products, and as an exhaustive list of all existing tools for reuse would be impossible for me to make during the work with the guidelines, I wanted only to give information on the types of tools that exist. Here, the IEEE Standard for Information Technology – Software Life Cycle Processes – Reuse Processes (IEEE Std 1517-1999) [2] provided me with a list of types of reuse support tools. Some of the tool categories have been (and may continue to be) manual activities, but I wanted to include them all, to show the reader the possibilities. As the information provided in the standard was already condensed, I chose to give an account of the entire list. (See section G.4 in appendix G for the full list.)

The "Software Reuse Executive Primer", developed by The Software Reuse Initiative of the Program Management Office of the United States Department of Defense [18] provided a list of technologies which was sorted in descending order of ability to easily incorporate reuse principles. (See section G.4 in appendix G for the full list.) I included this list in the tools section as well, because I thought it would be useful for developers to see how "reuse friendly" the technologies they are or will be using are.

To start the tools section a bit more "down to earth", I decided to cite Kremer [17]: "Design for reuse may be augmented by creating an environment that supports component reuse". I also included a list of elements which may be included in such an environment: A component database which is capable of storing software components, a repository management system which allows a client application to retrieve components from the repository, and computer aided software engineering (CASE) tools which support integration of reused components into a new design.

5.6 Interviews – Feedback 2

The second round of interviews were performed at three companies in Trondheim. As with the first round of feedback interviews, I sent the interviewees an e-mail containing the guidelines (but this time it was the in-depth second version, see appendix G) which they would read before the meetings and a list of questions which I would like them to think about before the meeting. This e-mail was sent a few days before the meeting, to give the interviewees some time to digest the information without leaving so much time between the e-mail and the interview that the information would be forgotten in between. The e-mail text is included in appendix E.

The interviews were performed in the same manner as the first round of interviews: The meetings took place at the interviewees' workplaces and I recorded the interviews with an MP3 player while taking notes on a printed copy of the guidelines. I started each interview by asking if it was OK that I recorded the interview, and none of the interviewees had any objections. Again, I tried to keep the situation informal and relaxed, allowing the interviewees to talk freely about what they wanted. I did not have to ask most of the questions I had sent the interviewees in advance, as the interviewees answered these questions while talking freely. The remaining questions were asked when there was a pause in conversation or at the end of the interview. The information from the second round of interviews is included in appendix H: Information about the interviewees, their companies, and the interviews, a summary of my notes written during the interviews, additional key points from the recordings, and the interviewees' answers to the e-mail questions. The following list contains a summary of those answers:

1. The interviewees stated that the overview diagrams and the activity descriptions were easily understandable.
2. None of the interviewees felt that there were other activities which should be included in the guidelines. One interviewee felt that the first activity, classifying the organization, was superfluous for small organizations.
3. All the interviewees regarded the activity of formulating a reuse policy as the most important one.
4. All the interviewees felt that the guidelines could be useful. Two interviewees felt that the guidelines in the present form were too theoretical, and that examples would be necessary for them to use the guidelines. The last interviewee felt that the guidelines were fully usable in their current form, but that examples would enhance them.

The results from the second round of feedback interviews which I considered

were the most important for further work with the guidelines are included in the following list.

1. Examples are essential to make the leap from the theoretical descriptions of the guidelines to making your own plan.
2. Make a list of definitions for words and terms used in the guidelines.
3. Specify scope and limitations of the guidelines, such as how many developers constitute a "small" software development department.
4. The explanation of the classification activity is not clear enough. (Why does it have to be performed before all the others?)
5. The tools section and the information on component classification is for people who are "particularly interested", and should be included as appendixes.
6. The tools section is not easily readable and understandable; the lists of types of tools and technologies do not provide useful information in their current form.
7. It would be useful to have a list of references sorted by category (such as "planning measurement" and "formulating goals").

5.7 Writing – Third Edition

The third phase of writing was the last one. During this phase, I worked with some of the issues which came up during the last round of feedback interviews, as well as making some corrections and changes to wordings. As I did not have much time left to revise the guidelines, some of the issues from the feedback interviews are left as possible further work (see chapter 8).

The following list is a summary of the changes I made to the guidelines:

- Added an appendix for definitions (see appendix A)
- Added a section describing scope and limitations to the introduction (section 6.1.1)
- Removed the paragraph describing the overview diagram in figure 6.3
- Changed the layout of the introductory section of each activity description; presenting the information about goal, inputs, and output in a list

- Added information about the classification activity (section 6.3.1); a more detailed description of what the activity involves and why, and an explanation of why it is the first activity
- Moved the information about faceted classification from the policy activity description (section 6.3.3) to the appendix section (appendix D), and changed the layout a bit to make the text easier to read by adding space
- Added a note on code standard and documentation standard as well as an example of a set of documentation rules to the "Where" section in the description of the activity Formulating a reuse policy (section 6.3.3)
- Added a reminder that measurement should have a purpose, as well as two more literature references to the section on considering reuse measurement (section 6.3.4)
- Moved the section Tools for Reuse from the guideline chapter to the appendix section (appendix C), changed the layout to improve readability, and removed the list of technologies which were sorted by "reuse friendliness"

The third edition, which is the current version, is included in the next chapter (chapter 6: Guidelines).

Chapter 6

Guidelines

This chapter contains the final version of the guidelines for making a plan for code reuse in small software development departments. This is mainly a guide for making a reuse plan for the first time, but each activity can, and probably should, be revisited and the documentation updated as experiences with code reuse are gained. These guidelines will go through several revisions, based on the feedback from developers at Norwegian companies with small software development departments.

6.1 Introduction

This set of guidelines is meant to help people in small software development departments who wish to make a plan for code reuse. There seems to be a relationship between the size of the department and its available resources: The smaller the department, the less resources are available to develop and implement a plan for reuse. Consequently, the process of following these guidelines is meant to be simple and inexpensive, both when it comes to time, and money.

6.1.1 Scope and Limitations

These guidelines are composed primarily with small software development departments in mind (less than ten developers). Most of the activities presented in these guidelines would probably be similar for larger departments, but the process would most likely be more complicated, and involve issues not covered in this report.

The guidelines discuss issues of code reuse (see definition in appendix A)

in general, but they do not contain information specific to reuse of third party code. Reusing third party code can be a great way to save time and effort, but it also introduces numerous challenges and problems which are not encountered when reusing one's own code. Some examples of things to watch out for are: What kind of license the code is published under, what kind of support is available, and whether there is a regular cycle for releases of patches and new versions of the code. Information on using third party software can for example be found in the following resources:

- Basili and Boehm [30] present ten hypothesis regarding the use of commercial off the shelf (COTS) software. For each hypothesis, sources are given and implications are explained.
- Wang and Wang [31] discuss the adoption of open source software (OSS). They present a list of requirements (such as reliability, availability of support, and licensing) to be considered when reusing open source software. They also include information on the most common licenses, and a framework for analyzing OSS.

6.2 Overview

The main activities needed to make a plan for reuse are presented in figure 6.1. The activities should be performed in sequence, as they are presented.

Each activity produces an output which is needed as input to other planning activities. Excluded from this is the output of the last activity, Compilation. This output is the final output; the reuse plan. A separate overview of the outputs is given in figure 6.2. The objects in this figure are numbered according to their corresponding activities (figure 6.1). As the figure shows, output 1, 2, 3, and 4 constitute parts of output 5. A complete overview of the process of making a plan for reuse is given in figure 6.3.

The overviews of activities and inputs/outputs does not include any external inputs or activities. This is done to keep it simple, hopefully making it easier to understand the main concepts. The activities and their outputs are listed below.

1. **Classify the reuse organization.** Gain an understanding of the organization's reuse potential. Output: A description of the organization.
2. **Formulate reuse goals.** Describe the desired effects of code reuse. Output: A set of reuse goals.

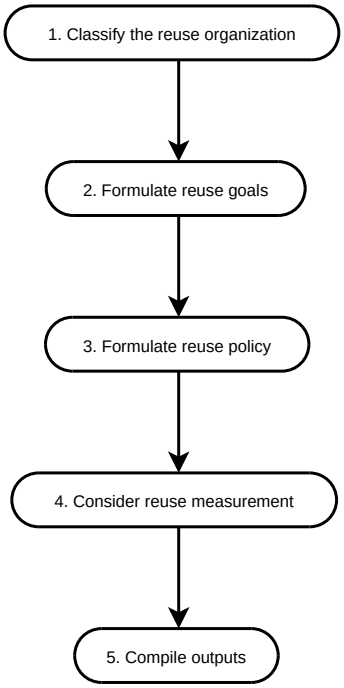

Figure 6.1: Overview of the Activities Included in Making a Reuse Plan

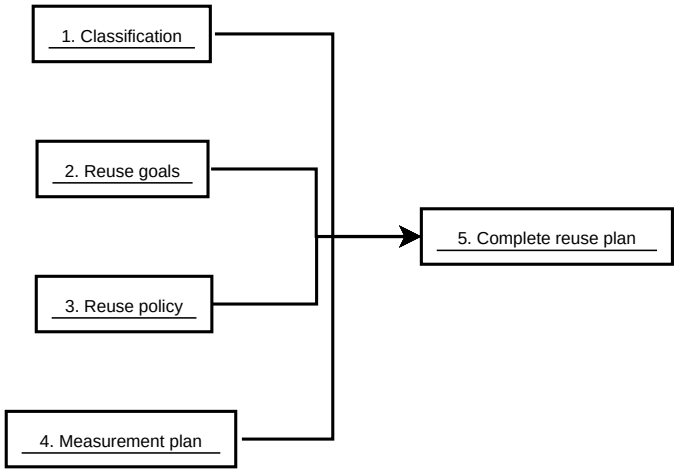

Figure 6.2: Overview of the Outputs Generated by the Planning Activities

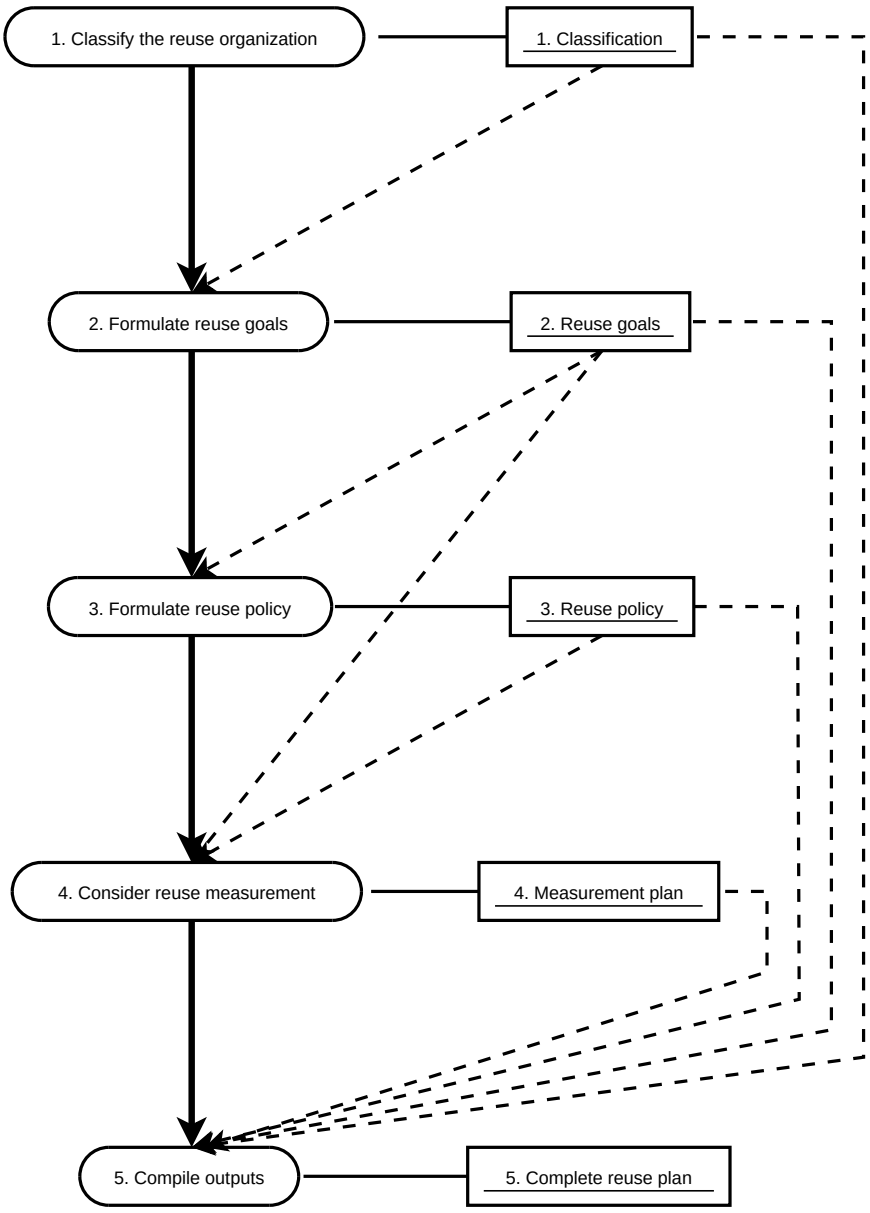

Figure 6.3: Overview of the Process of Making a Reuse Plan

3. **Formulate reuse policy.** Make a set of rules governing your code reuse. This is the main activity. Output: A reuse policy.
4. **Consider reuse measurement.** Decide whether reuse metrics should be used or not, and if yes: Make a plan for reuse measurement. Output: A measurement plan.
5. **Compile output.** Compile the outputs into the final, complete reuse plan. Output: The complete reuse plan.

Note that the third activity, Formulate reuse policy, is the main activity and as such requires more time and effort than the others. Each activity and its output will be described in the following section.

6.3 Description of Activities

In this section, the activities needed to make a reuse plan are described, one at a time. Each description starts by stating the goal of the activity. Then a list of the inputs to the activity is given, followed by a short description of the output. Then the activity itself is described, and references to further information on the subjects at hand are given. For clarity, the descriptions are kept as short as possible, leaving the reader to further investigate the given references as needed in order to gain a deeper understanding and make informed decisions.

6.3.1 Classify the Reuse Organization

Goal: Gain an understanding of the organization's reuse potential.

Input: This activity is the first step in making a plan for code reuse. As such, it has no input.

Output: A description of the organization/company of which the software development department is a part.

This activity is relatively simple, and should take little time to complete. Classification information about the organization is written down and examined in order to better understand the reuse potential of the existing organization. As this understanding is helpful throughout all the other activities in the process of making a plan for reuse, classifying the reuse organization is the first activity. There are many ways to describe and classify the organization, but not all characteristics are useful when we are trying to assess the reuse potential of an organization. A short list of characteristics considered by existing literature to be useful in assessing the reuse potential of an

organization is given below. For a closer description and an explanation of the selection of characteristics, see section 5.5.1.

Type of software production There are two main categories; Product family and Isolated. Product family is when the company develops a software product that evolves over time, and/or is more or less adapted for each customer. Isolated is when the company has projects which have little or nothing in common. Product family indicates a larger reuse potential than Isolated.

Size of organization/software staff Smaller is better when it comes to the ease of achieving top management commitment, as well as the ease of communication of information and the ease of building a consensus for the reuse program.

Software process maturity is the extent to which a process is clearly defined, managed, measured, controlled, and effective, and should be as high as possible, since a software development department with high process maturity will be better equipped to generate a plan for reuse and, equally important, implement the plan.

Domains As well-defined and narrow as possible, because well-defined and narrow domains facilitate reuse.

When you have classified your organization according to these characteristics, you will have a better understanding of the organization's reuse potential and you will hopefully be able to formulate realistic and reasonable reuse goals in the next activity.

6.3.2 Formulate Reuse Goals

Goal: Clarify your expectations by formulating a set of reuse goals, stating the desired effects of reuse, i.e. what you wish to accomplish by reusing code.

Input: The classification from activity 1.

Output: A set of reuse goals. These reuse goals are used as input to several of the ensuing activities.

Before a detailed plan for code reuse is made, it is important to thoroughly consider what you wish to gain by reusing code. Formulating a set of goals raises the general awareness of the reuse process, and hopefully contributes to achieving a "reuse mindset". Having a set of goals to reach for and getting feedback as to whether the goals have been reached also makes it easier to do one's best and to improve the software reuse process. Nazareth et al. [13] state that "Reuse programs are likely to be adopted more readily, and with

greater conviction, if a clearer understanding of the outcomes of the reuse program were available".

While determining the goals of code reuse, consider the information from the classification activity (section 6.3.1). This will give you help in formulating goals which are both reasonable and realistic.

As a starting point for formulating your own goals, you can for example consider some of the positive effects of software reuse:

Increased efficiency Reductions in development cost and time, increased programmer productivity

Better software Improved quality, stability, and maintainability of software

Standardization Uniformity of how problems are solved, of appearance and functionality

Simplified testing Less debugging, reused code is often tested code

Profit Selling reusable components to other companies

None of these effects are easily measurable, but it might be useful to set measurable goals to be able to compare your actual results with your predetermined goals. An example of this is the percentage of the work on a specified development project that is accomplished through the use of existing code [10]. See also section 6.3.4 concerning measurements.

6.3.3 Formulate Reuse Policy (What, How, Where, Who)

Goal: Formulate a reuse policy, i.e. a set of rules governing your code reuse. The set of rules answer the "what, how, where, who" questions of code reuse.

Input: The reuse goals from activity 2.

Output: A description of the reuse policy.

These are the questions which need to be answered:

- *What* kind of reuse do you wish to perform?
- *How* should the reusable components be classified, enabling them to be found and retrieved later?
- *Where* should the reusable components be stored?
- *Who* is responsible for what?

Each of these questions will be discussed in the following sections.

But first, have a look at the reuse models in appendix B. Hauge ([32]) presents four basic models on how reuse could be organized. The choice of model and the answers to several of the above mentioned questions influence each other, so keep the models in mind when you answer the questions, and keep the answers of the questions in mind when you consider the models. Also, during the work with formulating a reuse policy, remember the goals you expressed in the previous activity.

What

What kind of reuse do you wish to perform? In this section, two main decision points are presented:

- Black-box vs. white-box reuse
- Horizontal vs. vertical reuse

The most important decision is whether to perform black-box or white-box reuse. In black-box reuse, the component is reused unchanged, while in white-box reuse, the component is modified to fit the target product [12]. Ravichandran and Rothenberger [14] describe three reuse strategies: White-box reuse, black-box reuse with internal components, and black-box reuse with component markets. [14] is a particularly useful article, as it compares features and discusses advantages and disadvantages of each strategy and also includes a reuse decision tree which "depicts the various reuse strategies and the questions developers must answer in choosing each strategy. The model also depicts the cost equations that help developers answer these questions". The following paragraph contains a brief summary of the main points of the article, but I recommend reading the full article.

An advantage of white-box reuse is the freedom of the developers to modify the code to better suit their current needs. This fitting of existing components to new requirements maximize reuse opportunities, but it is also "a key source of problems encountered during reuse"[14], partly because modification requires a high degree of familiarity with the implementation details. Some inhibitors of white-box reuse mentioned in the article are

- a large up-front investment which is needed to populate a repository with reusable components,
- problems with classification and retrieval of reusable components,
- a potential lack of management support, and
- the need to change the organizational structure and processes.

Black-box reuse does not allow modification of the components which are to be reused. This avoids some of the aforementioned pitfalls, but it also greatly reduces the reuse opportunities. Hence, the possibility of customization through the use of predefined parameters and switches becomes important. Obtaining reusable components from third parties is also a possibility. This might increase the reuse rate as the developers can search through a (potentially) larger set of components, making it more likely that they will find components satisfying their needs.

Another decision is whether the reuse will be vertical, i.e. within a specific domain, or horizontal, i.e. across several domains. This partially depends on the type of software production of your company, see section 6.3.1. Isolated software production implicates horizontal reuse, while a product family implicates vertical reuse. An advantage of horizontal reuse is that "the broader scope of horizontal solutions increases the likelihood that your developers have relevant experience"[10]. A disadvantage is that each application will have less reused code, there is a limit of about 20 % [10]. Vertical reuse has a greater potential – Rhubart [10] states that the upper limits of vertical reuse can reach as high as 90 %. A downside is that vertical reuse calls for developers with domain-specific knowledge and experience, or the developers will be faced with a significant learning curve.

How

How should the reusable components be classified, enabling them to be found and retrieved later? Consider a large repository of reusable components. Although the components are available, they still have to be organized in some way, so developers can find them when they need them. Thus, the classification of reusable components is essential. The components should be described in unambiguous, classifiable terms.

The Institute of Electrical and Electronics Engineers (IEEE) has developed a standard entitled IEEE Standard for Information Technology – Software Reuse – Data Model for Reuse Library Interoperability: Basic Interoperability Data Model (BIDM) (IEEE Std 1420.1-1995)[3]. This standard describes "the minimal set of information about assets that reuse libraries should be able to exchange to support interoperability". The minimal set of information is "the information which would enable reuse library users to make quick, intelligent decisions about which assets in other reuse libraries will likely meet their needs". This set of information is, however, not only useful for library information exchange; information enabling developers to make quick, intelligent decisions is just as useful when it comes to the company's own reuse library. Hence, this standard could also be used as a basis for your own classification scheme, internal to your company.

Another source of information about classification is Kremer [17]. According to [17] an ideal description of a software component encompasses a so-called 3C Model: Concept, content and context.

Concept: This is a description of what the software does; the intent of the component should be conveyed. The description should include the interface to and the semantics of the component.

Content: This is a description of how the component is realized. This information is generally only needed by developers who wish to modify the component.

Context: This is a description of the component's context; "the placement of a component within its domain of applicability"[17]. The description should include conceptual, operational, and implementation features. This enables developers to find a component which meets their requirements.

These descriptions need to be translated into a concrete specification scheme, i.e. there has to be rules as to what kind of information should be provided for each component. There are many such classification schemes, one of which is Faceted classification, described in appendix D.

Ideally, "automated tools should be selected to provide the greatest capabilities in location, selection, use, and control of the reusable components"[18]. (See appendix C for a description of tools for reuse.) This leads us to the next question: "Where should the reusable components be stored?", which is the topic of the following section.

Where

Where should the reusable components be stored? It is generally considered useful and necessary to store reusable components in a separate reuse repository. Fay [15] explains why a reuse repository is different from a version control system, and why it is needed. A version control system manages software "work-in-progress", keeping track of changes to components during development, while the role of a repository is somewhat different. A reuse repository should be a "finished-goods" inventory, and serve as the channel of distribution. This means that the repository should enable developers to easily find the component they are looking for, quickly determine the component's relevance to their needs, effectively evaluate technical compliance, and obtain a copy of the component. Fay states that there should also be a mechanism for tracking information about the components, such as which projects are using the components, and the system should automatically inform users of a component when a new version of the component is available.

The system should also be able to track the savings realized through the use of reusable components from the repository. While I recommend employing a reuse repository, I will not make any recommendations of specific tools. (See appendix C for a description of tools for reuse.)

There are two alternatives to using a repository. Neither are as good as using a reuse repository, but might be the only feasible solution for a software development department with limitations on cost and time. One is to store reusable code together with non-reusable code, i.e. no system. This easily turns into a mess, and does not make it easier to achieve the positive effects of reuse. The other alternative is slightly better; storing reusable code in the same kind of system as non-reusable code, but keeping the reusable code separated from the non-reusable code (reuse library). Either way it is important to consider the organization of the components. Decide on a structure; you can for example establish naming and hierarchy conventions specifically for reusable components. You should also decide on a code standard and a documentation standard, to get a high degree of code and documentation uniformity. This makes it easier to find a certain piece of code, to understand the code, possibly change it, and use it. An example of a set of documentation rules is Sun's Requirements for Writing Java API Specifications[1].

Another topic which needs consideration, is how to populate the reuse repository/library. Do you wish to populate the storage with already existing components? This partially depends on whether you wish to perform black-box or white-box reuse; with white-box reuse, less modification of the existing components is necessary. With black-box reuse, all the components which are put into the storage should be completed and not be changed by the developer wishing to (re)use the component.

Who

Who is responsible for what? When making a plan for reuse, as in other planning activities, it is important to establish roles and responsibilities. In small development departments, however, the benefit and need of many roles, especially full-time roles, are relatively small. The smaller the development department, the less need for many reuse roles. Two important roles are reuse program manager and library manager. A reuse program manager is needed as a driving force in the reuse process; in making a plan for reuse, implementing the plan, checking whether the goals are achieved, and improving the reuse process. A library manager is needed to keep the reuse storage up to date and to ensure the quality of the reusable components.

[1] http://java.sun.com/j2se/javadoc/writingapispecs/index.html

Neither of these two roles have to be full-time, the point is that it is useful to have one person which is responsible for the reuse process in general and one person which is responsible for maintaining the reuse storage. Both of these roles can be assigned to a single person.

6.3.4 Consider Reuse Measurement

Goal: Decide whether you wish to use measurements to supervise the (measurable) results of your code reuse, and make a plan for the measurements you wish to perform.

Input: There are two inputs to this activity; the reuse goals from activity 2, and the reuse policy from activity 3.

Output: A measurement plan.

Using reuse metrics are described in most existing literature as being important. Here is a small collection of opinions:

Tomer et al. [12] (citing Poulin [25]) "it is widely accepted that the organizational challenges of software reuse outweigh the technical ones. As a result, metrics are needed in order to "make business decisions possible by quantifying and justifying the investment necessary to make reuse happen"[25]"

Nazareth and Rothenberger [13] "Reuse programs are likely to be adopted more readily, and with greater conviction, if a clearer understanding of the outcomes of the reuse program are available"

Boehm [11] includes in his list of eight critical reuse success factors the following point: "Use metrics-based reuse operations management". He goes on to explain that this is important for "tracking progress with respect to expectations and making appropriate adjustments where necessary".

Many measurement tasks can, and should, be performed by tools. (For a short discussion of tools for software reuse, see appendix C.) It is, however, important to remember not to go ahead and measure anything and everything; consider what you wish to measure and why – what will you do with the results? There should be a purpose to each measurement you plan to perform. When you decide what to measure and why, keep your reuse goals and your reuse policy in mind; do you reach your goals, and do you follow your policy?

The subject of measuring reuse efforts is too complex to be covered in these guidelines, but information on software reuse metrics abound. If you decide

to measure your reuse efforts, information on how to do this can be found for example in the following resources:

- Poulin: Metrics for Object-Oriented Reuse [26]
- Poulin: Measuring Software Reuse: Principles, Practices, and Economic Models [25]
- Frakes and Terry: Software Reuse: Metrics and Models [27]
- Devanbu et al.: Analytical and Empirical Evaluation of Software Reuse Metrics [28]
- Pfleeger: Measuring Reuse: A Cautionary Tale [29]
- Van Solingen: The Goal/Question/Metric Approach [33]
- Park et al.: Goal-Driven Software Measurement – A Guidebook [34]

6.3.5 Compile Outputs

Goal: Gather the information from the other activities and compile them into the final, complete reuse plan.

Input: This is the final step in creating the complete reuse plan, and the inputs to this activity are the outputs of all the other activities: The classification from activity 1, the reuse goals from activity 2, the reuse policy from activity 3, and the measurement plan from activity 4.

Output: The complete reuse plan.

Gather the documents created during the four previous activities, and compile them into one document. This document is the complete reuse plan. The first section contains a description of your organization. The second section contains a description of the goals you wish to accomplish by reusing code. The third section is probably the longest one, containing the reuse policy. If you have decided to measure you reuse efforts, there will be a fourth section containing a measurement plan. Give some thought to the way you store the finished reuse plan; make sure it is stored in a way which enables all the developers to access it, and which makes revisions possible.

Chapter 7

Discussion and Conclusion

In this chapter, I present a discussion of my work and a conclusion on the usability of the guidelines.

7.1 Discussion

The main goal of this work was to produce a set of guidelines to help software developers in small software development departments to reuse code in a more proper way. I feel that I have come a long way in reaching this goal. However, there is still work to be done before the guidelines are complete. See the following chapter for a description of further work.

The main goal consisted of several sub-goals. In the following list, I will describe each goal and whether I feel I have reached it.

- Get an overview of important issues with regard to code reuse: I have reached this goal. My previous work on the project[1], the literature study, and the feedback interviews have provided me with a wealth of information.
- Summarize this information and present it in a sensible and easily understandable way: I have been able to present the information in a sensible and easily understandable way, although the information could have been made more accessible by dividing it into a theory section and a "how to" section, as well as adding a section of examples.
- Get feedback and information from the target group (developers in small software development departments) with regards to:
  - Superfluous or missing information in the guidelines
  - Comprehensibility of the guidelines

  - And ultimately: How useful the guidelines are

  I have reached this goal. The information in the guidelines is considered to be OK (some interviewees wanted more, some wanted less). The guidelines are considered to be comprehensible. The guidelines are considered to be usable, although an example section is strongly recommended.

- Possibly publish the guidelines in some way: During the planning phase, I discovered that I would not be able to reach this goal, and so I have not tried to reach it.

7.2 Conclusion

The guidelines in their present form are considered easily understandable and useful by the developers I interviewed, but one or several examples of how the process could be performed is essential to help the developers make the leap from the theoretical descriptions of the guidelines to making their own plan. This means that further work on the guidelines is needed before the guidelines will be able to guide the reader through the process of making a plan for reuse. (For a list of further work, see the following chapter.)

Chapter 8

Further Work

During the work with this thesis, several ideas for further work have appeared. The following list of ideas emerged from the feedback interviews. The ideas are either wishes expressed by the interviewees or improvements I suggested and the interviewees thought would be useful.

- Include an example ("toy store") as the last section of the guidelines. This was (in some form) a wish from all the interviewees, since it would help the reader to start the thought process for making their own plan. As people read the example, they will probably take a stand as to whether the descriptions fit themselves, and whether they would do the same as the company/companies in the example.
- Add more in-depth information regarding black-box and white-box reuse, as this was considered by the interviewees to be an interesting subject.
- Add more information regarding reuse measurement, or possibly just an example of how measurement could be done.
- Include a list of references sorted by categories which correspond to the activities listed in the guidelines. This would make it easier to find information on each subject.

I originally wanted to publish the guidelines in some way, to make them available to the target group. I did not have the time to do this during the time assigned to working with the thesis, so publishing the guidelines is left as a possibility for further work.

Appendix A

Definitions

This appendix contains definitions of some of the terms used in this report.

Asset: An item of interest, stored in a reuse library (e.g. source code).

Capability Maturity Model: The Capability Maturity Model (CMM) describes the principles and practices underlying software process maturity. Using CMM, software organizations can improve the maturity of their software processes, evolving from ad hoc, chaotic processes to mature, disciplined software processes.

Classification: The manner in which the assets are organized – making it easier to search for and extract assets from a reuse library.

Code reuse: The use of code (source code, binary code, components, or modules) in the solution of different problems. A wide definition which describes the essence of code reuse is: Code is reused when it 1) already exists, and 2) is chosen over the possibility to write new code.

Component: A constituent part of a system (e.g. a module).

Domain: A problem space. A particular area of activity or interest.

Reuse: The use of an asset in the solution of different problems.

Software process maturity: The extent to which a process is clearly defined, managed, measured, controlled, and effective. (See also Capability Maturity Model.)

Systematic reuse: The practice of reusing according to a well-defined, repeatable process.

Appendix B

Reuse Models

Hauge ([32]) presents four basic models on how reuse could be organized, taken from Davenport & Probst ([35]). In this appendix I will describe each model briefly.

B.1 Model 1 — Project Oriented

Shown in figure B.1, model 1 is the most straightforward reuse model. A shared reuse storage is used to exchange reusable components between projects. Each project is responsible for identifying and developing components for this reuse storage.

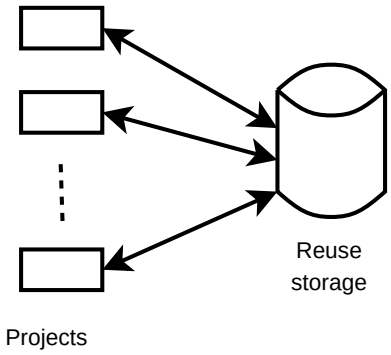

Figure B.1: Reuse Model 1 — Project Oriented

B.2 Model 2 — Reuse Through a Separate Project

Model 2 (figure B.2) is similar to model 1, as the projects identify possibly reusable components to add to a shared storage. Model 2 adds a level of complexity: The development of the reusable components is performed by a temporary reuse project, which then adds the components to the shared storage. When a component is added to the reuse storage, other projects can retrieve it from this shared storage, similarly to the previous model.

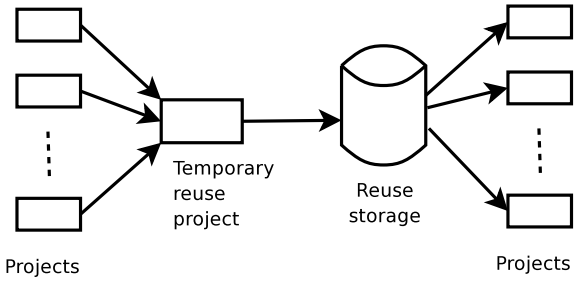

Figure B.2: Reuse Model 2 — Reuse Through Separate Project

B.3 Model 3 — Component Producer

In model 3 (figure B.3), all parts of the development of reusable components are the responsibility of a permanent department; the component producer. As in the two previous models, the reusable components are added to a shared storage and thus made available to the projects.

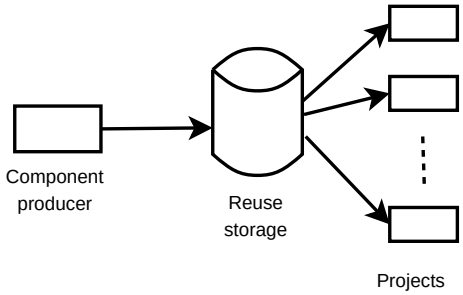

Figure B.3: Reuse Model 3 — Component Producer

B.4 Model 4 — Domain Producers

The most complex reuse model is model 4 (see figure B.4). This model is used in larger organisations who develop products within several domains. Domain specific reusable components are produced by a component development department for one or more domains.

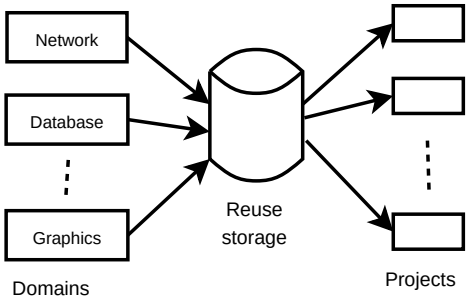

Figure B.4: Reuse Model 4 — Domain Producers

Appendix C

Tools for Reuse

While the organizational and managerial issues generally are considered the most important when reusing software components, technical issues also matter. Tools that provide automated support for reuse make reuse easier to practice and help improve the quality of components. Reuse-oriented tools extend or complement software development tools to the extent that they handle the reuse properties of assets. This section gives a brief summary of some types of tools for reuse.

Kremer [17] states that "Design for reuse may be augmented by creating an environment that supports component reuse". This may include the following elements: A software component database, a repository management system which gives access to the database, a component retrieval system which allows a client application to retrieve components from the repository, and computer aided software engineering (CASE) tools which support integration of reused components into a new design.

The IEEE Standard for Information Technology – Software Life Cycle Processes – Reuse Processes (IEEE Std 1517-1999) [2] includes a list of reuse support tools, divided into four categories. In the standard, it is pointed out that "while some of the categories of reuse tools have been manual activities, and may continue to be, all categories have the potential to benefit from automation". The tools in each of the four categories are listed in the paragraphs below.

The first category is *Analysis and design. Tools for reuse-oriented domain analysis and design* assist domain engineers to recognize similarities among domain elements and to trial-fit elements into existing models and architectures. These tools assist developers to extend and improve their inventory of domain models and architectures. *Legacy-asset salvage analysis tools* analyze legacy assets in order to determine structural and functional patterns

of similarity. *Tools used for applications requirements analysis* cross-match requirements to existing assets in order to minimize the deltas between what is available and what is needed. *Reuse-oriented application design tools* interrogate selected domain architectures in order to give developers a list of options for instantiating the architectures' components. The result is a formal specification of the design of a software product sufficient to drive both documentation and construction tools.

The second category is *Asset constructors*. *Smart editors* find appropriate assets, and parse them so a developer can instantiate them to a particular context. *Generators* construct assets by combining design specifications with domain information contained within the tool. *Assemblers* construct assets by combining design specifications with assets external to the tool. *Legacy-asset reconditioners* package desired patterns, extracted by salvage analysis tools, into assets.

The third category is *Asset testers*. *Adaptability testers* assist domain engineers to determine and improve the ease-of-reuse of given assets. *Generality testers* assist domain engineers to determine and modify the domain of applicability of given assets.

The fourth category is *Reuse management*. There are two areas of *measuring reuse cost and benefits*: Reuse asset life cycle and application life cycle. Tools used in the *reuse asset life cycle* determine costs to manage the asset storage and retrieval mechanisms and to amortize assets over time and over software products. These tools determine the relative costs and benefits of having various assets. Tools used during the *application life cycle* determine project development and maintenance time, as well as effort expended or avoided due to reuse. *Asset configuration and version management tools* keep track of how to access assets, asset ownership, servicing responsibilities, and which version of an asset is used in which software products. *Tools for impact analysis of asset modification* keep track of where assets are reused and dependencies among the assets. *Asset inventory analysis tools* determine the orthogonality (duplication and/or overlap) of the inventory, as well as the age and status of the inventory items. *Tools for asset cataloging* formally register assets into various asset storage and retrieval mechanisms, including updating browsing and retrieval tools with appropriate descriptors and search criteria. *Asset search and retrieval tools* browse and access assets, possibly allowing developers to set appropriate parameters for both construction-time adaptation and runtime execution. *Asset certification tools* support secure certification of an asset's status in terms of where it is reusable (i.e. its reusability) – project, department, enterprise, industry, etc.

Appendix D

Faceted Classification

There are many ways to classify reusable components. One example, described by Kremer [17], is faceted classification. *Facets* are basic descriptive features, for example the function performed by the component, the data manipulated by the component, the context in which the component is applied, or any other feature. The facets are identified by analyzing a domain area. They are then prioritized by importance, and connected to the component. Each component has a set of facets describing it. This set is called the *facet descriptor*, and should according to Kremer generally not contain more than seven or eight facets. An example is a simple scheme with the following facet descriptor:

[function, object type, system type]

Each facet has one or more values which are generally descriptive keywords. The facet "function" might have the following values:

function=(copy, from) or *function=(copy, replace, all)*

Thus, the use of multiple facet values "enables a refined sense of the primitive function copy"[17].

For each component in the reuse repository, keywords are assigned to each facet of the facet descriptor, e.g. the facet "function" has the keywords/-values "copy" and "from". When developers query the repository for possible components, they specify a list of keyword values, and the repository is searched for matches. Automated tools can be used to incorporate a thesaurus, which makes it possible to find matches for technical synonyms of the given keywords. (See appendix C for an overview of tools for reuse.)

Appendix E

E-mail Sent to Companies

This is the e-mail I sent to the three companies located in Oslo and the three companies located in Trondheim. The questions are the same, but the guidelines given to the companies in Trondheim were a complete set, while the companies in Oslo were given only the introductory part of the guidelines. (See appendix F and G for the guidelines sent to the companies.) The full sentence and the sentence in parenthesis followed by an asterisk (*) was only included in the first set of e-mails, which were sent to the interviewees who received the introductory version of the guidelines. Apart from this one sentence, the contents of the first and second set of e-mails were the same.

E.1 Norwegian

Hei,

nå begynner det å nærme seg vårt avtalte møte, og her kommer det jeg ønsker du skal titte på før møtet (vedlagte fil guidelines_only.pdf).

Dette er som du ser kun en innledning til et mer omfattende kapittel.*

Det jeg ønsker feedback fra deg på, er følgende:

- Er prosess-oversikten forståelig? Kommer det tydelig (om enn noe kort beskrevet)* frem hva hver aktivitet innebærer, slik at du sitter igjen med en oversikt over hva prosessen går ut på?
- Er det en/flere aktiviteter som du synes mangler, eller er det en/flere aktiviteter som er overflødige? (Hva, hvorfor?)
- Hvilke aktiviteter synes du virker mest interessante? Hvilke aktiviteter tror du er nyttigst (gjerne basert på hvilken nytteverdi de ville ha for

deg/dere)?

- Tror du de endelige retningslinjene kan være nyttige for deg/utviklingsavdelingen deres? (Hvorfor/hvorfor ikke? Hvordan? For eksempel: For å ha en "huskeliste" å se på selv, for å lage en gjenbruksplan for avdelingen, annet?)

På forhånd tusen takk for at du setter av litt tid til å tenke på dette før møtet vårt!

Ser frem til å møte deg.

Med vennlig hilsen Lisa Wold Eriksen

E.2 English

Hello,

our appointed meeting draws near, and here is the material I wish you look at before the meeting (attached file guidelines_only.pdf).

This is as you can see only an introduction to a more comprehensive chapter.*

I want your feedback on the following:

- Is the process overview comprehensible? Is it clear (although only briefly described)* what each activity includes, so you are left with an overview of what the process involves?
- Is there one or several activities missing, or is there one or several superfluous activities? (What, why?)
- Which activities do you regard as the most interesting? Which activities do you believe are the most useful (this might be based on their utility value to you)?
- Do you think the final set of guidelines could be useful to you/your development department? (Why/why not? How? For example: To have a "memo" to look at yourself, to produce a reuse plan for the department, other?)

Thank you in advance for setting aside some time to think this through before our meeting!

Looking forwards to meeting you.

Kind regards, Lisa Wold Eriksen

Appendix F

Guidelines (First Version)

This chapter contains the latest version of the guidelines for code reuse in small software development departments. These guidelines will go through several revisions, based on the feedback from developers at Norwegian companies with very small software development departments.

F.1 Introduction

This set of guidelines is meant to be a help for people in small software development departments who wish to make a plan for code reuse. There seems to be a relationship between the size of the department and its available resources: The smaller the department, the less resources are available to develop and implement a plan for reuse. Consequently, the process of following these guidelines is meant to be simple and inexpensive, both when it comes to time, and money.

F.2 Overview

The main activities needed to make a plan for reuse are presented in figure F.1. Each activity produces an output which is needed as input to other planning activities. Excluded from this is the output of the last activity, Compilation. This output is the final output; the reuse plan. The overview of activities and inputs/outputs does not include any external inputs or activities. This is done to keep it simple, hopefully making it easier to get a grasp of the main concepts. The activities and their outputs are listed below.

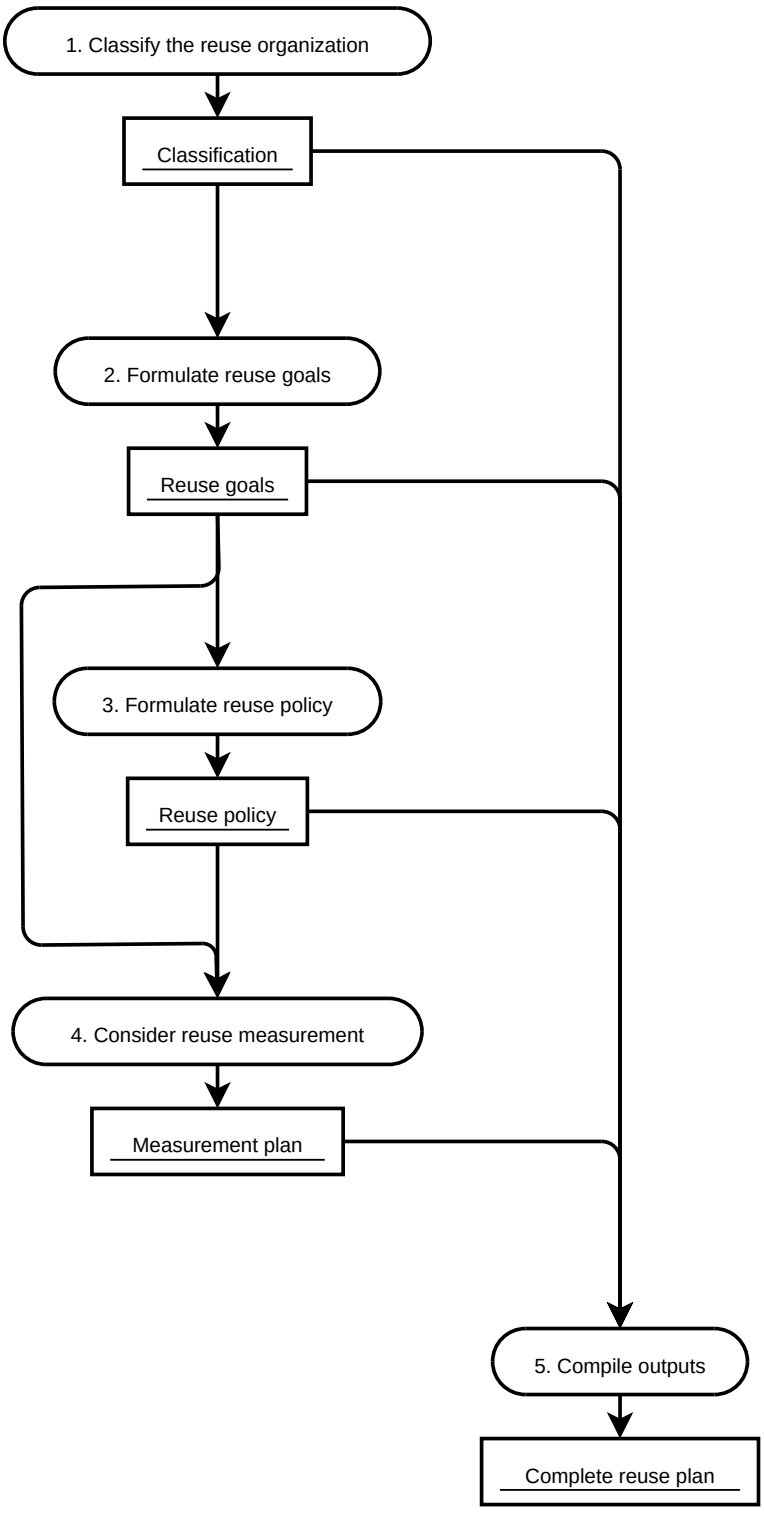

Figure F.1: Overview of the Process of Making a Reuse Plan

1. **Classify the reuse organization.** Gain an understanding of the organization's reuse potential. Output: A description of the organization.
2. **Formulate reuse goals.** Describe the desired effects of code reuse. Output: A set of reuse goals.
3. **Formulate reuse policy.** Make a set of "rules" governing your code reuse. Output: A reuse policy.
4. **Consider reuse measurement.** Decide whether reuse metrics should be used or not, and if yes: Make a plan for reuse measurement. Output: A measurement plan.
5. **Compile output.** Compile the other outputs into the final, complete reuse plan. Output: The complete reuse plan.

Each of these activities and its outputs will be described in the following section. Note that activity 2. Formulate reuse goals does not necessarily have to be fully completed before activity 3. Formulate reuse policy begins, as activity 3 might influence the results of activity 2.

Appendix G

Guidelines (Second Version)

This chapter contains the latest version of the guidelines for making a plan for code reuse in small software development departments. This is mainly a guide for the first-time making of a reuse plan, but each activity can, and probably should, be revisited and the documentation updated as experiences with code reuse are gained. These guidelines will go through several revisions, based on the feedback from developers at Norwegian companies with very small software development departments.

G.1 Introduction

This set of guidelines is meant to be a help for people in small software development departments who wish to make a plan for code reuse. There seems to be a relationship between the size of the department and its available resources: The smaller the department, the less resources are available to develop and implement a plan for reuse. Consequently, the process of following these guidelines is meant to be simple and inexpensive, both when it comes to time, and money.

G.2 Overview

The main activities needed to make a plan for reuse are presented in figure G.1. The activities should be performed in sequence, as they are presented.

Each activity produces an output which is needed as input to other planning activities. Excluded from this is the output of the last activity, Compilation. This output is the final output; the reuse plan. A separate overview of

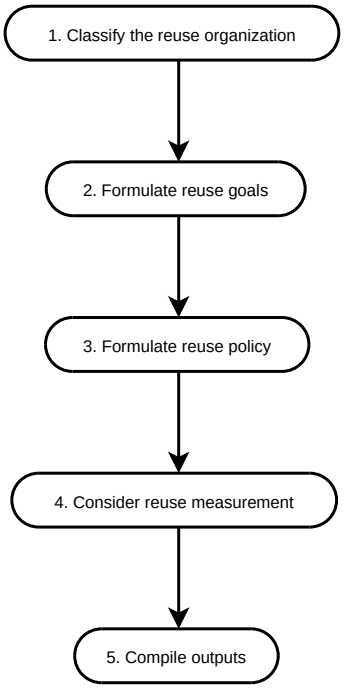

Figure G.1: Overview of the Activities Included in Making a Reuse Plan

the outputs is given in figure G.2. The objects in this figure are numbered according to their corresponding activities (figure G.1). As the figure shows, output 1, 2, 3, and 4 constitute parts of output 5.

A complete overview of the process of making a plan for reuse is given in figure G.3. There are solid lines without arrows between the activities and their resulting outputs. From each activity there is a solid, bold line with an arrow pointing to the following activity. From each output there are dotted lines with arrows pointing to the activities in which the output is needed. Output 2. Reuse goals is for example needed both in activity 3. Formulate reuse policy and activity 4. Consider reuse measurement.

The overviews of activities and inputs/outputs does not include any external inputs or activities. This is done to keep it simple, hopefully making it easier to get a grasp of the main concepts. The activities and their outputs are listed below.

1. **Classify the reuse organization.** Gain an understanding of the organization's reuse potential. Output: A description of the organization.

2. **Formulate reuse goals.** Describe the desired effects of code reuse. Output: A set of reuse goals.

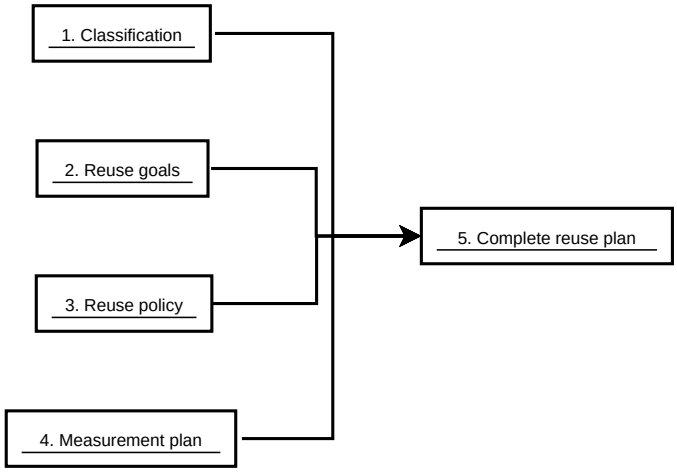

Figure G.2: Overview of the Outputs Generated by the Planning Activities

3. **Formulate reuse policy.** Make a set of "rules" governing your code reuse. Output: A reuse policy.
4. **Consider reuse measurement.** Decide whether reuse metrics should be used or not, and if yes: Make a plan for reuse measurement. Output: A measurement plan.
5. **Compile output.** Compile the other outputs into the final, complete reuse plan. Output: The complete reuse plan.

Each of these activities and its outputs will be described in the following section.

G.3 Description of Activities

In this section, the activities needed to make a reuse plan is described, one at a time. Each description starts by stating the goal of the activity. Then a list of the inputs to the activity is given, followed by a short description of the output. Then the activity itself is described, and references to further information on the subjects at hand are given. For clarity, the descriptions are kept as short as possible, leaving the reader to further investigate the given references as needed in order to gain a deeper understanding and make informed decisions.

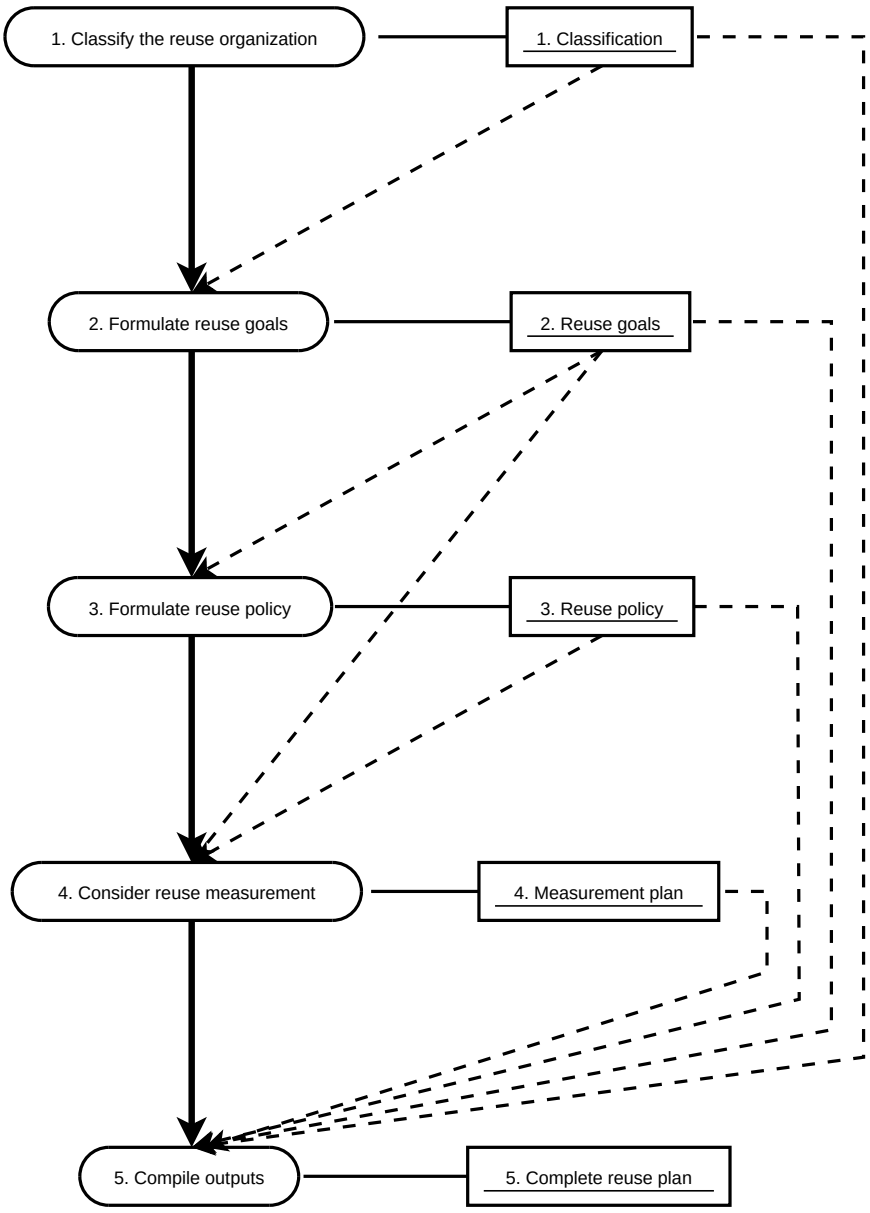

Figure G.3: Overview of the Process of Making a Reuse Plan

G.3.1 Classify the Reuse Organization

The goal of this activity is to gain an understanding of the organization's reuse potential. This activity is the first step in making a plan for code reuse. As such, it has no input. The output of this activity is a description of the organization/company of which the software development department is a part.

There are many ways to describe and classify the organization, but not all characteristics are useful when we are trying to assess the reuse potential of an organization. Below, a short list of characteristics considered by existing literature to be useful in assessing the reuse potential of an organization is given. For a closer description and an explanation of the selection of characteristics, see section 5.3

Type of software production There are two main categories; Product family and Isolated. Product family is when the company develops a software product that evolves over time, and/or is more or less adapted for each customer. Isolated is when the company has projects which have little or nothing in common. In this case, Product family is the better alternative.

Size of organization/software staff Smaller is better when it comes to the ease of achieving top management commitment, as well as the ease of communication of information and the ease of building a consensus for the reuse program.

Software process maturity As high as possible, since a software development department with high process maturity might be better equipped to generate a plan for reuse and, equally important, implement the plan.

Domains As well-defined and narrow as possible, because well-defined and narrow domains facilitate reuse.

When you have classified your organization according to these characteristics, you will have a better understanding of the organization's reuse potential and you will hopefully be able to formulate realistic and reasonable reuse goals in the next activity.

G.3.2 Formulate Reuse Goals

The goal of this activity is to formulate a set of reuse goals, stating the desired effects of reuse, i.e. what you wish to accomplish by reusing code. This activity takes as input the classification from activity 1, and generates

a set of reuse goals as output. These reuse goals are used as input to several of the ensuing activities.

Before a detailed plan for code reuse is made, it is important to thoroughly consider what you wish to gain by reusing code. Formulating a set of goals raises the general awareness of the reuse process, and hopefully contributes to achieving a "reuse mindset". Having a set of goals to reach for and getting feedback as to whether the goals have been reached also makes it easier to do one's best and to improve the software reuse process. Nazareth et al. [13] state that "Reuse programs are likely to be adopted more readily, and with greater conviction, if a clearer understanding of the outcomes of the reuse program were available".

As a starting point for formulating your own goals, you can for example consider some of the positive effects of software reuse:

Increased efficiency Reductions in development cost and time, increased programmer productivity

Better software Improved quality, stability, and maintainability of software

Standardization Uniformity of how problems are solved, of appearance and functionality

Simplified testing Less debugging, reused code is often tested code

Profit Selling reusable components to other companies

None of these effects are easily measurable, but it should be noted that it might be useful to set some measurable goals to be able to compare your actual results with your predetermined goals. An example of this is the percentage of the work on a specified development project that is accomplished through the use of existing code [10]. See also section G.3.4 concerning measurements.

While determining the goals of code reuse, consider the information from the classification activity (section G.3.1). This will give you some help in formulating goals which are both reasonable and realistic.

G.3.3 Formulate Reuse Policy (What, How, Where, Who)

The goal of this activity is to formulate a reuse policy, i.e. a set of "rules" governing your code reuse. The set of rules answer the "what, how, where, who" questions of code reuse. The input to this activity is the reuse goals from activity 2. The output is a description of the reuse policy.

These are the questions which need to be answered:

- *What* kind of reuse do you wish to perform?
- *How* should the reusable components be classified, enabling them to be found and retrieved later?
- *Where* should the reusable components be stored?
- *Who* is responsible for what?

The following sections will discuss each of these questions in turn.

What

What kind of reuse do you wish to perform? In this section, two main decision points are presented:

- Black-box vs. white-box reuse
- Horizontal vs. vertical reuse

The most important decision is whether to perform black-box or white-box reuse. In black-box reuse, the component is reused unchanged, and in white-box reuse, the component is modified to fit the target product [12]. Ravichandran and Rothenberger [14] describe three reuse strategies: White-box reuse, black-box reuse with internal components, and black-box reuse with component markets. [14] is a very useful article, as it compares features and discusses advantages and disadvantages of each strategy and also includes a reuse decision tree which "depicts the various reuse strategies and the questions developers must answer in choosing each strategy. The model also depicts the cost equations that help developers answer these questions". The following paragraph contains a brief summary of the main points of the article, but I recommend reading the full article.

An advantage of white-box reuse is the freedom of the developers to modify the code to better suit their current needs. This fitting of existing components to new requirements maximize reuse opportunities, but it is also "a key source of problems encountered during reuse"[14], partly because modification requires a high degree of familiarity with the implementation details. Some inhibitors of white-box reuse mentioned in the article are

- a large up-front investment which is needed to populate a repository with reusable components,
- problems with classification and retrieval of reusable components,
- a potential lack of management support, and
- the need to change the organizational structure and processes.

Black-box reuse does not allow modification of the components which are to be reused. This avoids some of the aforementioned pitfalls, but it also greatly reduces the reuse opportunities. Hence, the possibility of customization through the use of predefined parameters and switches becomes important. Obtaining reusable components from third parties is also a possibility. This might increase the reuse rate as the developers can search through a (potentially) larger set of components, making it more likely that they will find components satisfying their needs.

Another decision is whether the reuse will be vertical, i.e. within a specific domain, or horizontal, i.e. across several domains. This partially depends on the type of software production of your company, see section G.3.1. Isolated software production implicates horizontal reuse, while a product family implicates vertical reuse. An advantage of horizontal reuse is that "the broader scope of horizontal solutions increases the likelihood that your developers have relevant experience"[10]. A disadvantage is that each application will have less reused code, there is a limit of about 20 % [10]. Vertical reuse has a greater potential – Rhubart [10] states that the upper limits of vertical reuse can reach as high as 90 %. A downside is that vertical reuse calls for developers with domain-specific knowledge and experience, or the developers will be faced with a significant learning curve.

How

How should the reusable components be classified, enabling them to be found and retrieved later? Consider a large repository of reusable components. Although the components are available, they still have to be organized in some way, so developers can find them when they need to. Thus, the classification of reusable components is essential. The components should be described in unambiguous, classifiable terms.

The Institute of Electrical and Electronics Engineers (IEEE) has developed a standard entitled IEEE Standard for Information Technology – Software Reuse – Data Model for Reuse Library Interoperability: Basic Interoperability Data Model (BIDM) (IEEE Std 1420.1-1995)[3]. This standard describes "the minimal set of information about assets that reuse libraries should be able to exchange to support interoperability". The minimal set of information is "the information which would enable reuse library users to make quick, intelligent decisions about which assets in other reuse libraries will likely meet their needs". This set of information is, however, not only useful for library information exchange; information enabling developers to make quick, intelligent decisions is just as useful when it comes to the company's own reuse library. Hence, this standard could also be used as a basis for your own classification scheme, internal to your company.

Another source of information about classification is Kremer [17]. According to [17] an ideal description of a software component encompasses a so-called 3C Model: Concept, content and context.

Concept: This is a description of what the software does; the intent of the component should be conveyed. The description should include the interface to and the semantics of the component.

Content: This is a description of how the component is realized. This information is generally only needed by developers who wish to modify the component.

Context: This is a description of the component's context; "the placement of a component within its domain of applicability"[17]. The description should include conceptual, operational, and implementation features. This enables developers to find a component which meets their requirements.

These descriptions need to be translated into a concrete specification scheme. One possible such scheme is Faceted classification. Facets are basic descriptive features, for example the function performed by the component, the data manipulated by the component, the context in which the component is applied, or any other feature. The facets are identified by analyzing a domain area. They are then prioritized by importance, and connected to the component. Each component has a set of facets describing it. This set is called the facet descriptor, and should according to Kremer generally not contain more than seven or eight facets. An example is a simple scheme with the following facet descriptor: *[function, object type, system type]*. Each facet has one or more values which are generally descriptive keywords. The facet function might have the following values: *function=(copy, from)* or *function=(copy, replace, all)*. Thus, the use of multiple facet values "enables a refined sense of the primitive function copy"[17].

For each component in the reuse repository, keywords are assigned to the set of facets. When developers query the repository for possible components, they specify a list of keyword values, and the repository is searched for matches. Automated tools can be used to incorporate a thesaurus, which makes it possible to find matches for technical synonyms of the given keywords.

Ideally, "automated tools should be selected to provide the greatest capabilities in location, selection, use, and control of the reusable components"[18]. This leads us to the next question: "Where should the reusable components be stored?", which is the topic of the following section.

Where

Where should the reusable components be stored? It is generally considered useful and necessary to store reusable components in a separate reuse repository. Fay [15] explains why a reuse repository is different from a version control system, and why it is needed. A version control system manages software "work-in-progress", keeping track of changes to components during development, while the role of a repository is somewhat different. A reuse repository should be a "finished-goods" inventory, and serve as the channel of distribution. This means that the repository should enable developers to easily find the component they are looking for, quickly determine the component's relevance to their needs, effectively evaluate technical compliance, and obtain a copy of the component. Fay states that there should also be a mechanism for tracking information about the components, such as which projects are using the components, and the system should automatically inform users of a component when a new version of the component is available. The system should also be able to track the savings realized through the use of reusable components from the repository. While I recommend employing a reuse repository, I will not make any recommendations of specific tools.

There are two alternatives other than using a repository. Neither are as good as using a reuse repository, but might be the only feasible solution for a software development department with limitations on cost and time. One is to store reusable code together with non-reusable code, i.e. no system. This easily turns into a mess, and does not make it easier to achieve the positive effects of reuse. The other alternative is slightly better; storing reusable code in the same kind of system as non-reusable code, but keeping the reusable code separated from the non-reusable code (reuse library). Either way it is important to consider the organization of the components. Decide on a structure; you can for example establish naming and hierarchy conventions specifically for reusable components.

Another topic which needs consideration, is how to populate the reuse repository/library. Do you wish to populate the storage with already existing components? This partially depends on whether you wish to perform black-box or white-box reuse; with white-box reuse, less modification of the existing components is necessary. With black-box reuse, all the components which are put into the storage should be completed and in no need of any change now or later.

Who

Who is responsible for what? When making a plan for reuse, as in other planning activities, it is important to establish roles and responsibilities.

But in small development departments, the benefit and need of many roles, especially full-time roles, are relatively small. The smaller the development department, the less need for many reuse roles. Two important roles are reuse program manager and library manager. A reuse program manager is needed as a driving force in the reuse process; in making a plan for reuse, implementing the plan, checking whether the goals are achieved, and improving the reuse process. A library manager is needed to keep the reuse storage up to date and to ensure the quality of the reusable components. Neither of these two roles have to be full-time, the point is that it is useful to have one person which is responsible for the reuse process in general and one person which is responsible for maintaining the reuse storage.

G.3.4 Consider Reuse Measurement

The goal of this activity is to decide whether you wish to use measurements to supervise the (measurable) results of your code reuse, and make a plan for the measurements you wish to perform. There are two inputs to this activity; the reuse goals from activity 2, and the reuse policy from activity 3. The output of this activity is a measurement plan.

Using reuse metrics are described in most existing literature as being important. Here is a small collection of opinions:

Tomer et al. [12] (citing Poulin [25]) "it is widely accepted that the organizational challenges of software reuse outweigh the technical ones. As a result, metrics are needed in order to "make business decisions possible by quantifying and justifying the investment necessary to make reuse happen"[25]"

Nazareth and Rothenberger [13] "Reuse programs are likely to be adopted more readily, and with greater conviction, if a clearer understanding of the outcomes of the reuse program are available"

Boehm [11] includes in his list of eight critical reuse success factors the following point: "Use metrics-based reuse operations management". He goes on to explain that this is important for "tracking progress with respect to expectations and making appropriate adjustments where necessary".

Information on software reuse metrics abound, and if you decide to measure your reuse efforts, information on how to do this can be found for example in the following resources:

- Poulin: Metrics for Object-Oriented Reuse [26]

- Poulin: Measuring Software Reuse: Principles, Practices, and Economic Models [25]
- Frakes and Terry: Software Reuse: Metrics and Models [27]
- Devanbu et al.: Analytical and Empirical Evaluation of Software Reuse Metrics [28]
- Pfleeger: Measuring Reuse: A Cautionary Tale [29]

G.3.5 Compile Outputs

The goal of this activity is to gather the information from the other activities and compile them into the final, complete reuse plan. This is the final step in creating the complete reuse plan, and the inputs to this activity are the outputs of all the other activities: The classification from activity 1, the reuse goals from activity 2, the reuse policy from activity 3, and the measurement plan from activity 4. The output of this activity is the complete reuse plan.

Gather the documents created during the four previous activities, and compile them into one document. This document is the complete reuse plan. The first section contains a description of your organization. The second section contains a description of the goals you wish to accomplish by reusing code. The third section is probably the longest one, containing the reuse policy. If you have decided to measure you reuse efforts, there will be a fourth section containing a measurement plan.

G.4 Tools for Reuse

It should be noted that while the organizational and managerial issues generally are considered the most important when reusing software components, technical issues also matter. Tools that provide automated support for reuse make reuse easier to practice and help improve the quality of components. Reuse-oriented tools extend or complement software development tools to the extent that they handle the reuse properties of assets. This section gives a brief summary of some types of tools for reuse.

Kremer [17] states that “Design for reuse may be augmented by creating an environment that supports component reuse”. This may include the following elements: A component database which is capable of storing software components, a repository management system which gives access to the database, a component retrieval system which allows a client application to retrieve components from the repository, and computer aided software

engineering (CASE) tools which support integration of reused components into a new design.

The IEEE Standard for Information Technology – Software Life Cycle Processes – Reuse Processes (IEEE Std 1517-1999) [2] includes a list of types of reuse support tools. Some of the types of tools have been, and may continue to be, manual activities, all of them have the potential to benefit from automation. There are four categories of tools. The first category is Analysis and design. There are four types of tools in this category:

Reuse-oriented domain analysis and design Assist domain engineers to recognize similarities among domain elements and to trial-fit elements into existing models and architectures. Assist developers to extend and improve their inventory of domain models and architectures.

Legacy-asset salvage analysis Analyze legacy assets in order to determine structural and functional patterns of similarity.

Applications requirements analysis Cross-match requirements to existing assets in order to minimize the deltas between what is available and what is needed.

Reuse-oriented application design Interrogate selected domain architectures in order to present developers with a list of options for instantiating the architectures' components. The result is a formal specification of the design of a software product sufficient to drive both documentation and construction tools.

The second category is Asset constructors:

Smart editors Find appropriate assets, and parse them so a developer can instantiate them to a particular context.

Generators Construct assets by combining design specifications with domain information contained within the tool.

Assemblers Construct assets by combining design specifications with assets external to the tool.

Legacy-asset reconditioners Package desired patterns, extracted by salvage analysis tools, into assets.

The third category is Asset testers:

Adaptability testers Assist domain engineers to determine and improve the ease-of-reuse of given assets.

Generality testers Assist domain engineers to determine and modify the domain of applicability of given assets.

The fourth category is Reuse management:

Measuring reuse cost/benefits *Reuse asset life cycle:* Determine costs to manage the asset storage and retrieval mechanisms and to amortize assets over time and over software products. Determine the relative costs and benefits of having various assets.

Measuring reuse cost/benefits *Application life cycle:* Determine project development and maintenance time and effort expended/avoided due to reuse.

Asset configuration and version management Keep track of how to access needed assets, ownership, and servicing responsibilities, and which version of an asset applies to which software products.

Impact analysis of asset modification Keep track of where assets are reused, and dependencies among assets.

Asset inventory analysis Determine the orthogonality (duplication and-/or overlap) of the inventory, and the age and status of inventory items.

Asset cataloging Formal registration of assets into various asset storage and retrieval mechanisms, including updating browsing and retrieval tools with appropriate descriptors and search criteria.

Asset search and retrieval Browse and access assets, possibly allowing developers to enter appropriate parameter settings for both constructiontime adaptation and runtime execution.

Asset certification Support the secure certification of an asset's status in terms of its scope of reusability–project, department, enterprise, industry, etc.

The article Software Reuse Executive Primer [18] provides this list of technologies, in descending order of ability to easily incorporate reuse principles:

- Application generators
- I-CASE tools
- Architecture development tools
- Problem-oriented languages
- Code skeletons
- Composition tools
- CASE tools
- Modeling tools

- Object-oriented knowledge base
- Libraries/repositories
- Natural languages
- Application languages
- Dataflow languages
- Object-oriented languages
- Very high-level languages
- Formal methods
- High-level languages
- Assembly languages

Appendix H

Interviews

This appendix contains information on the feedback interviews with the software developers. The first section describes the first set of interviews (performed in Oslo), while the second section describes the second set of interviews (performed in Trondheim). Each interview is described with the following information: Where the company is located, the position of the interviewee, number of employees (developers and total), date of the interview, and duration of interview. During each interview, I made notes on a printed version of the guidelines I had sent to the interviewee. In addition, I recorded the interview on an MP3 player. To obtain a copy of the notes and/or recordings, please contact the author. Here I include an English summary of the relevant notes taken and additional key points from the recordings, as well as a list of the interviewee's answers to the questions I asked in the e-mail sent to the interviewee before the interview (see appendix E).

I will refer to the e-mail questions only by number. The numbering is as follows:

- 1: Is the process overview comprehensible? Is it clear[1] what each activity includes, so you are left with an overview of what the process involves?
- 2: Is there one or several activities missing, or is there one or several superfluous activities? (What, why?)
- 3: Which activities do you regard as the most interesting? Which activities do you believe are the most useful (this might be based on their utility value to you)?

[1]The following was added in the e-mail to the three first interviewees: (although only briefly described)

- 4: Do you think the final set of guidelines could be useful to you/your development department? (Why/why not? How? For example: To have a "memo" to look at yourself, to produce a reuse plan for the department, other?)

H.1 First Round of Interviews

The first three interviews were performed in Oslo. The interviewees had been sent an e-mail (see appendix E) with the introductory version of the guidelines (see appendix F).

H.1.1 Interview 1

Location	Oslo
Job position of interviewee	Software developer
Employees (developers/total)	2/9
Date	May 19. 2005
Duration of interview	40 minutes (10:00–10:40)

Notes

- The developers use Microsoft SourceSafe, MSDN and Visual Studio.
- The figure (1.1) is not very easily understood.
  - It looks like activity 3 is optional! Change the figure – perhaps by simply moving the arrow (from Reuse goals to 4. Consider reuse measurement) a bit to the right at both ends.
  - Maybe include an arrow with a dotted line from activity 3 to activity 2 to illustrate that it is possible to go back to activity 2 during activity 3?
- Checklists and cookbook recipes would be nice, with references to more detailed information.

Additional Key Points

- This version of the guidelines is pretty general, but the guidelines will be useful with some more in-depth information.

- Yes, it would be nice to have a list of things to consider, with references to more information. It is important to start a thought process, without having to start at scratch.
- On measurements: I think it is important to have automated tools which do these things for you, as it isn't likely anyone would perform measurements manually. But to the extent that there are tools which do it for you, it can be sensible to perform measurements.

Answers to Questions

- *1:* Yes, in my opinion. You have clearly thought more about this than I have. The more I read, the more sensible it seemed to be, both the activities and their sequence. The activities are comprehensible, at least with the list of descriptions. The diagram caused some confusion.
- *2:* I can't think of anything missing or superfluous. Initially, I felt that the first activity might not be useful, but then, after reading about the second and third activity, I thought that it might be useful to go through activity one after all.
- *3:* I feel the second and third activities, and especially the third, are the most important.
- *4:* I think these guidelines could be useful to our development department, of course with more in-depth information. I think these guidelines would be helpful to small organizations with limited resources.

H.1.2 Interview 2

Location	Oslo
Job position of interviewee	Software developer
Employees (developers/total)	3/35
Date	May 19. 2005
Duration of interview	60 minutes (12:00–13:00)

Notes

- A misunderstanding regarding the time aspect: The process is meant to be the first time making of a plan for reuse, while the interviewee had gotten the impression that the developers were supposed to go through the process each time they needed to reuse something. Probably due to a slip of the pen in the very first line of the guidelines ("This chapter contains the latest version of the guidelines for code reuse in small

software development departments"), although it was stated elsewhere that "This set of guidelines is meant to be a help for people in small software development departments who wish to make a plan for code reuse". Correct this!

- Each activity might be (probably should be) revisited at a later time (but this is not covered by these guidelines).
- Change the figure (1.1), the output/document is an implicit part of each activity; don't include the output rectangles in the figure.
- The use of tools is interesting, include some information on this.

Additional Key Points

- A problem with reuse is that you don't know what the other programmers have developed. There usually isn't much time to go through what the others have made, so you might do the same many times.
- Tools are important. At the present time, I use Google[2] a lot to find code I can use, but it's not so usual to reuse our own code. There is no overview of what we have produced, it just gets told from one to another, and that is pretty vulnerable if someone disappears for some reason.
- Getting an overview of what we have developed and what we can reuse would be important to us.
- Documentation is essential, and using proper names for variables and functions in the code is also useful to understand what the code does.
- If I were to start reusing what we've got now, I would primarily try to extract all the functions, and the ones where there is doubt as to what it should be used for would have to be documented better. The next thing would be to make it very searchable, so you can find everything fast. Again, tools are essential. Then, it's the issue of having the goal that the code you write should be reused which is important.

Answers to Questions

- *1:* I thought it was a set of guidelines for reusing, not for making a plan for reuse.
- *2:* Research is important, but that might go into every activity and not be separated as a single activity.

[2] www.google.com

- *3:* The activity of making a reuse policy is important. Also, which tools you should use is an important issue.
- *4:* Yes. This is exciting!

H.1.3 Interview 3

Location	Oslo
Job position of interviewee	Lead developer
Employees (developers/total)	4/9
Date	May 20. 2005
Duration of interview	30 minutes (13:00–13:30)

Notes

- The figure (1.1) could be separated into two separate figures; one for the activities and one for the outputs.
- It is important to mention that activity 3 (Formulate reuse policy) is the most important one.
- Link the theory to practical usage, by using examples, or perhaps a "toy store" (make up a story about a company which goes through the process of making a plan for reuse).
- Refer to typical tools for reuse.

Additional Key Points

- It would be nice with an overview over what kind of tools exist, but the problem is that the specific tools change all the time, and people use different platforms. There is no point in discussing that, but you could include references to typical tools, or at least describe what the different types of tools can do.
- It's more important to have a plan for reuse and a common mindset when the department is large than when there are only 2–3 developers. You should always consider whether you have done this before, or if you can make the code general to enable reuse.

Answers to Questions

- *1:* I thought the diagram was well arranged when I looked at it. The arrow from 2 to 3 seems to indicate that activity three is voluntary

and could be skipped. The division of the process into activities is sensible; I see the value of each activity, and feel that they constitute a natural course through a process for making a reuse plan.

- *2:* I have no expertise, but there doesn't appear to be anything missing or superfluous.
- *3:* The third activity is the one which should be in focus. For example activity one seems dull, there are no immediate effects of it. But activity two and three are exciting, because you will see the effects of performing them, there will be concrete rules. That's where the real work lies.
- *4:* I'm not sure, because this version is only an overview. But I definitely see the value of having the right mindset for reuse.

H.1.4 Summary

Common amongst the interviewees was the opinion that checklists and/or "cookbook recipes" are useful. Also, there was confusion about the process diagram. The information from each interview which I felt would have the greatest impact on my further work was:

- Interview 1: The diagram is confusing and has to be changed.
- Interview 2: Outputs are obvious, there is no need to include the outputs in the activity diagram.
- Interview 3: Examples, such as a "toy store", are important to aid understanding.

H.2 Second Round of Interviews

The second round of three interviews was performed in Trondheim. The interviewees had been sent an e-mail (see appendix E) with an in-depth version of the guidelines (see appendix G).

H.2.1 Interview 1

Location	Trondheim
Job position of interviewee	System architect
Employees (developers/total)	8(From 5 at the turn of the year)/37
Date	May 24. 2005
Duration of interview	90 minutes (12:00–13:30)

Notes

- The guidelines are a bit high-flying, would like them to be more concrete/practical.
- Make a list of definitions for words and terms used in the guidelines (such as "code reuse" and "component").
- It would be nice with some examples.
- The developers have started using a new tool for design/analysis; Enterprise Architect (Sparx System).
- On measurement: Not everything which can be measured should be measured.
- On classifying the reuse organization:
  - "Domains" is at too high a level to be considered at this company.
  - Other things which can affect reuse potential: Different tools, languages and platforms in different projects.
- On formulating reuse goals: "Profit: Selling reusable components to other companies" is quite far-fetched, that means that the reusable component becomes a product in itself.
- Vertical and horizontal reuse depends on the viewpoint; whether it is from a programmer's or a designer's point of view. Designer: Vertical reuse. Programmer: Horizontal reuse.
- On policy, Where:
  - Code standards, error handling are not mentioned.
  - Slip of the pen, last line: Should be "... and not in need of change by the one who's going to use the component".
- On tools:
  - Often you just use what tools you already have, and possibly adding things which are missing (i.e. tools for reuse).
  - Shorten the list from the IEEE, as well as the list from Software Reuse Executive Primer.

Additional Key Points

- These guidelines concentrate on what you should do. It would be better if someone made something based on the guidelines, which you

could use as a starting point, and then get more involved in what you feel you need. Having the guidelines as they are is OK, but it would be much better with a standard example of how to do it.

Answers to Questions

- *1:* Yes, both the overview diagrams and the activity descriptions are understandable.
- *2:* No, I can't see anything missing. Some parts of a few activities might be superfluous, but the activities themselves should be included.
- *3:* Activity three is the most important one, but deciding on measurement is perhaps the most interesting activity.
- *4:* It is nice to have a list of things you need to do, related to goals, measurements et cetera. But the guidelines are a bit too abstract to have a practical use. Adding examples would make it much better, as you can decide whether you would do it the same way as the company in the example, and then you're actually on the way to making your own reuse plan. Then you can do the formal things later, and go through every activity as they are described.

H.2.2 Interview 2

Location	Trondheim
Job position of interviewee	Project coordinator
Employees (developers/total)	4/11
Date	May 26. 2005
Duration of interview	40 minutes (09:00–09:40)

Note: This company had started using a new system for software development, which included functionality for code reuse.

Notes

- Yes, case descriptions/"toy store" would be nice, as well as some real-life examples where appropriate.
- It would be nice to have a bibliography sorted by category (such as "formulating goals" and "measurement") in a separate chapter before the regular bibliography.
- Developers often mis-estimate the time they need to program something themselves (from the bottom up), thinking it will take less time

than it actually does. This makes them discard the possibility of reuse more easily.

- The developers at this company often use documentation in the code to search for things they can reuse. Some rules to govern what kind of information which is given about each component would be useful. TODO: Check the exact wording here!
- On figure 1.1: Activity 2 (Formulate reuse goals) is about clarifying expectations.
- On figure 1.3: The description of the figure is a bit long, not needed – people will understand just by looking at the figure.
- On classifying the reuse organization: There isn't much explanation of why this activity should come first.
- On formulating reuse policy:
  - What: More information on horizontal vs. vertical reuse would be interesting.
  - Who: The reuse program manager and the library manager is often the same person.
  - Who: An ownership feeling is important. In this company, different people have the responsibility for the different application layers (business, logic, data). This makes them feel ownership for development within their own layer.
- On considering reuse measurement: There is little information in the guidelines, but this is an important issue.
- On tools: This section is too difficult to follow; change the formatting.

Additional Key Points

- On roles: We haven't assigned anyone to be a process manager, but there is one developer which is very involved, who sorts code and clears up. I think you're dependent on having a person who likes having that kind of responsibilities.

Answers to Questions

- *1:* Yes. But I feel the first activity, classification, it's not explained clearly why it should be performed before the others. Also, the tools section at the end is a bit disconnected from the rest. The diagrams

are easily understandable, and the description of the overview diagram (last one) is not needed, you intuitively understand this without an explanation.

- *2:* No, I don't think anything is missing. It's a good overview. But I think the issue of measurements is not discussed enough, as it can be difficult to set into practice.
- *3:* The third activity, definitely. That's where I feel the work is. But activity two is very essential, not only for the developers, but to clarify to the management and customers where the resources should be used.
- *4:* We're working on making procedures for reuse, but these guidelines are relevant; these are the things we have discussed. So using the document as a starting point wouldn't be a bad idea. It fits into what we are doing.

H.2.3 Interview 3

Location	Trondheim
Job position of interviewee	Head of development
Employees (developers/total)	3/8
Date	May 27. 2005
Duration of interview	60 minutes (12:00–13:00)

Notes

- These guidelines might be a bit "overkill", at least for this company.
- (Re)using third party code is an interesting subject – do you mean to include this kind of reuse in the guidelines? Amplify whether third party reuse is included or not. It would be nice with at least some short information with references to supplementary information.
- Define the scale – what is "small software development departments"?
- The "not invented here" syndrome is usually just a waste of time and money, as someone else probably already has solved the same problem, possibly in a better way than you could yourself.
- It's important to go through the process described in the guidelines before it's needed.
- On classifying the reuse organization: This is too theoretical for a small company.

- On formulating reuse goals: A positive effect of reuse: More and better quality control of own code.
- On formulating reuse policy:
  - What: It would be interesting with some more information on both black-box and white-box reuse, especially some information on specific techniques for black-box reuse.
  - What: An inhibitor of white-box reuse: Maintenance work.
  - How: The information on faceted classification is for the especially interested; put it in an appendix.
  - Where: It would be interesting with information on the practical aspects of repositories, such as how you make changes to components in the repository (for example bugfixes).
  - Who: It is necessary that someone has responsibility of what gets into the library/repository.
- On considering reuse measurement: This is a bit thin, not very concrete. Perhaps an example would help.
- On tools: This section is also for the especially interested. Put it in an appendix.

Additional Key Points

- The scale should be specified – what is "small development departments"?
- Including a "toy store" is a great idea. It would be nice to be able to read how it actually is done and form you own thoughts around it.
- I think a lot of what's written here is very good and explanatory, but there are some things which are not explained enough; practical information on using repositories, and black-box versus white-box.

Answers to Questions

- *1:* Yes. I found the guidelines to be easily readable and understandable.
- *2:* Well, I don't think there are any missing activities. If I were to change it, I would rather use fewer of the activities, because we're such a small organization, and I don't really see the point in classifying the organization and such, it gets too academical for such a small company.

As a whole, I don't think anything is missing, although I would like to see some information on third party code.

- *3:* Definitely activity three.
- *4:* Today, when we are so few, I feel that parts of these guidelines are a bit uninteresting, but the point of reuse is very interesting. That makes activity three relevant to me. On one side, I think the idea of making a plan is nice, but we're OK as we are, without having to define things on paper. But at the same time, if we for example hire a new programmer which hasn't got the experience we've got, having an overview of reusable code and having a plan is very useful, because it shortens the training process. And it's even more important if someone quits or disappears suddenly. I envision that if you actually do this before you need it, it really pays when the size of the departments increases.

H.2.4 Summary

As in the first round of interviews, all the interviewees regarded examples and/or recipes as helpful in the process of understanding the guidelines. The new diagrams were easily understood by all interviewees. The information from each interview which I felt would have the greatest impact on my further work was:

- Interview 1: Add a list of word definitions (define for example "component" and "code reuse").
- Interview 2: Add a separate chapter of references which are sorted by category.
- Interview 3: Add some information on third party code, with references to more information. Move the theory on tools and component classification into separate appendixes.

Bibliography

[1] Lisa Wold Eriksen. Code reuse in object oriented software development. Master's thesis, Norwegian University of Science and Technology (NTNU), 2004.

[2] Carma McClure. IEEE standard for information technology – software life cycle processes – reuse processes, June 1999. IEEE Std 1517-1999.

[3] The RIG Technical Committee on Asset Exchange Interfaces. IEEE standard for information technology – software reuse – data model for reuse library interoperability: Basic interoperability data model (bidm), December 1995. IEEE Std 1420.1-1995.

[4] The RIG Technical Committee on Asset Exchange Interfaces. Supplement to IEEE standard for information technology – software reuse – data model for reuse library interoperability: Asset certification framework, December 1996. IEEE Std 1420.1a-1996.

[5] Marcus A. Rothenberger. Project-level reuse factors: Drivers for variation within software development environments. *Devision Sciences*, 34(1):83–106, 2003.

[6] Marcus A. Rothenberger, Kevin J. Dooley, Uday R. Kulkarni, and Nader Nada. Strategies for software reuse: A principal component analysis of reuse practices. *IEEE Transactions on Software Engineering*, 29(9):825–837, September 2003.

[7] Maurizio Morisio, Michel Ezran, and Colin Tully. Success and failure factors in software reuse. *IEEE Transactions on Software Engineering*, 28(4):340–357, April 2002.

[8] Tim Menzies and Justin S. Di Stefano. More success and failure factors in software reuse. *IEEE Transactions on Software Engineering*, 29(5):474–477, May 2003.

[9] Maurizio Morisio, Michel Ezran, and Colin Tully. Comments on more success and failure factors in software reuse. *IEEE Transactions on Software Engineering*, 29(5):478, May 2003.

[10] Bob Rhubart. Getting to the goal: Setting your sights on software reuse, March 2002. Published on AWprofessional.com.

[11] Barry Boehm. Managing software productivity and reuse. *Computer*, 32(9):111–113, September 1999.

[12] Amir Tomer, Leah Goldin, Tsvi Kuflik, Esther Kimchi, and Stephen R. Schach. Evaluating software reuse alternatives: A model and its application to an industrial case study. *IEEE Transactions on Software Engineering*, 30(9):601–612, September 2004.

[13] Derek L. Nazareth and Marcus A. Rothenberger. Assessing the cost-effectiveness of software reuse: A model for planned reuse. *The Journal of Systems and Software*, 73:245–255, 2004.

[14] T. Ravichandran and Marcus A. Rothenberger. Software reuse strategies and component markets. *Communications of the ACM*, 46(8):109–114, August 2003.

[15] Sharon Fay. Work-in-process vs. finished-goods: Why a version control system is not a reuse repository, October 2002. Published on Flashline.com.

[16] Paul Harmon. A well-managed repository, August 2001. Published on Flashline.com.

[17] Rob Kremer. Software reuse, 1999. Published on the Software Engineering Research Network at the University of Calgary, sern.ucalgary.ca.

[18] The Software Reuse Initiative of the Program Management Office of the United States Department of Defense. Software reuse executive primer, 1996. Published on Flashline.com.

[19] Kristen Ringdal. *Enhet og mangfold*. Fagbokforlaget Vigmostad og Bjørke AS, 1. edition, 2001.

[20] Camilla Fledsberg. Prosessorientert kvalitetssystem i praksis. Master's thesis, NTNU, June 2003.

[21] Steinar Kvale. *Det kvalitative forskningsintervju*. Ad Notam Gyldendal, 1997.

[22] Colin Robson. *Real World Research*. Blackwell Publishing, 2002.

[23] Carolyn B. Seaman. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on Software Engineering*, 25(4):557–572, July/August 1999.

[24] Kenneth M. Anderson. Software methods and tools – software re-use. Lecture presentation in CSCI3308 at the University of Colorado, 2004.

[25] Jeffrey S. Poulin. *Measuring Software Reuse: Principles, Practices, and Economic Models*. Addison-Wesley, 1997.

[26] Jeffrey S. Poulin. Metrics for object-oriented reuse. Published on Flashline.com.

[27] William Frakes and Carol Terry. Software reuse: Metrics and models. *ACM Computing Surveys*, 28(2):415–436, June 1996.

[28] Prem Devanbu, Sakke Karstu, Walcélio Melo, and William Thomas. Analytical and empirical evaluation of software reuse metrics. In *Proceedings of the 18th international conference on Software engineering*, pages 189–199, 1996.

[29] Shari Lawrence Pfleeger. Measuring reuse: A cautionary tale. *IEEE Software*, pages 118–127, July 1996.

[30] Victor R. Basili and Barry Boehm. COTS-based systems top 10 list. *Computer*, pages 91–93, May 2001.

[31] Huaiquing Wang and Chen Wang. Open source software adoption: A status report. *IEEE Software*, pages 90–95, March/April 2001.

[32] Tor-Erik Hauge. Gjenbruk i it-bedrifter — utvikling og trender. Master's thesis, Høgskolen i Stavanger, June 2003.

[33] R. Van Solingen. *The Goal/Question/Metric Approach*, pages 578–583. Encyclopedia of Software Engineering – 2 Volume Set, 2002.

[34] Robert E. Park, Wolfhart B. Goethert, and William A. Florac. Goal-driven software measurement – a guidebook. HANDBOOK CMU/SEI-96-HB-002, August 1996.

[35] Tom Davenport and Gilbert J. B. Probst. *Knowledge Management Case Book: Siemens best practises*. Wiley, John & Sons, Incorporated, 2002.

Index