MASTER'S THESIS | LUND UNIVERSITY 2018

Investigating the impact of code sharing and how to manage it

André Alm, Daniel Dornlöv

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2018-15

Investigating the impact of code sharing and how to manage it

André Alm
andre.alm.554@student.lu.se

Daniel Dornlöv
daniel.dornlov.992@student.lu.se

9th November 2018

Master's thesis work carried out at Praqma AB.

Supervisors: Lars Bendix, lars.bendix@cs.lth.se
Albert Rigo, albert.rigo@praqma.net

Examiner: Ulf Asklund, ulf.asklund@cs.lth.se

Abstract

In today's software development climate there is a need to achieve a higher speed of the development process and avoid having to write the same code again, while keeping the costs at a reasonable level. Because of this a rising trend is to share code between projects, teams or developers. The shared code might be source code, binaries, or libraries.

This thesis focuses on what drives a company to start using shared code and also on what can be done both to try to solve the problems caused by code sharing but also to try and see how shared code can be supported so that it becomes as effective and efficient as possible. To get an understanding of what different drivers and problems there are with code sharing interviews were conducted with both developers and persons working in a more overarching role between projects and teams. In addition to the interviews a literature study was conducted and the findings from both of these were weighed together to give a requirements specification on what is needed to share code and avoid the problems found during the interviews and literature study.

The requirements were used as a guide to find solutions to the problems with shared code. It became apparent that there is no silver bullet that can solve all of the problems, instead the trick is in combining solutions to form a package of tools, principles and workflows that together mitigate or lessens the problems. This approach led to a matrix in which it is possible to find a solution based on at what time in the development a problem occurs and on what type of solution it is. This was then used to make a case study at a company and give a recommendation on how they could work with code sharing.

Keywords: Shared code, Duplicated code, Lead time, Inefficiency, Software configuration management, Organisation, Architecture

Acknowledgements

We would like to thank everyone that has helped with making this thesis come together, to the case study company for allowing us to make the case study and sharing their problems with us, to the persons that agreed to be interviewed. We would also like to thank our two supervisors Albert and Lars for their encouragement and willingness to guide us along the way.

Lund, autumn, 2018

André and Daniel

Contents

Chapter 1

Introduction

One of the trends in the software industry today is the desire to effectively share code between products to a large extent. This development is a consequence of the increasing demands for advanced functionality and customisation from customers. In some sectors of the industry, for example in machine heavy industry, hardware has traditionally been responsible for managing most of the functions while software has only been of lesser use. Both the software and the hardware have been highly modelled for the specific purpose and developed separately for the particular machine. This is both expensive and requires long lead times. The economic reality compels more effective development methodologies.

1.1 Problem

This thesis focuses on investigating the shared code practice, it is a practice that seems to be on the rise in the industry as the need for higher adaptability, higher development speed and lowering cost are increasing. Sharing code as a theoretical practice sounds very tempting to most, it promises to cut the costs since you only need to develop a module once and can then reuse it which in turn will increase the speed of the development since some parts are already finished. However in practice code sharing have several problems that companies wanting to use it should be aware of or in some cases have become aware of when the problems hit them.

In this work the different aspects of shared code will be discussed to try and understand questions such as why code sharing is used, what benefits it brings when used, what disadvantages and problems it brings. The thesis will try to analyse and present solutions to the elicited problems and also discuss the specific situation at a case study company who would like to start to use shared code in a major way.

1.2 Purpose

In the industry there is a movement going from hardware specific software to software that has to work as multiple different products or on different hardware. This has led to an increase in software complexity and because of this companies are looking to optimise the development process. The purpose of this thesis is to investigate, discuss and analyse the impact sharing code has on development and also how it can be organised and supported. In addition to this the thesis aims to make a recommendation to a specific company for two specific scenarios.

1.3 Thesis structure

The next chapter, background and methodology, leads the reader into the field and provides all the necessary context to understand the domain and problem area. In addition, the chapter also serves to describe the methodology and motivate the research questions. The third chapter, underlying motivations, advantages and disadvantages, aims to present the found driving forces, advantages and disadvantages of shared code, and the most interesting discoveries from the survey are discussed. In the fourth chapter, possible solutions to the problems are discussed. The fifth chapter discusses the recommendations for the specific case company. In the penultimate chapter, discussion and related work, the validity of the thesis is discussed, proposals for future work are given and reasoning connected to previous research and work is undertaken. The final chapter of the report draws conclusions based on the results from the previous chapters.

Chapter 2

Background and Methodology

This chapter presents the reader with a background to ensure that the rest of the report can be read and understood in the context of the problem. The background is complemented with a problem statement giving a description of why the problem exists, why it should be investigated, and also presents the research questions and the hypothesis. Lastly the reader is given an overview of the methodology used to investigate the questions, and a discussion regarding how the methodology was chosen and why. The chapter follows the same order that the different parts have been mentioned here, beginning with a description of the background followed by a description of the problem statement, and the methodology, respectively.

2.1 Background

In this section the background and context of the thesis is analysed, the concept of shared code is defined and delimited. The section aims to supply the background of the thesis so that the reader better understands the subsequent chapters.

The phenomenon of reuse of units is nothing new in the industry but the shift towards code sharing instead of reuse by copying is becoming more and more widespread. Among other things, the transition from strong hardware dependency in many industries to increasingly software-dependent products has made sharing seem as an increasingly attractive way to go.

As mentioned an alternative to sharing code is to copy code, a copy of the code that exists is taken and the new user may then build their product with this code. If you want

to maintain the code only as one unit, this can be a fateful act. At first, the copied version and the original are equal, but as the development progresses and new functionality is added, and changes are made separately, they will begin to increasingly differ from each other. The versions diverge and a problem Wayne Babich calls double maintenance occurs [1]. This means that the versions now have to be maintained separately because their implementations differ.

The vital part with code sharing is to avoid the double maintenance problem and transfer the code so that all products or teams are accessing and modifying the same data. The problems of double maintenance are eliminated but the shared data problem arise in its place [1]. This means that changes from a product team can directly impact the work of other projects using the shared data. It can of course be a great advantage but also disadvantage, for example, when bugs are introduced into shared code that many rely on, or for that matter when code or interfaces are modified so that their behaviour changes. Other times, it may be that product teams rely, perhaps unknowingly, on behaviour defined by bugs in the shared code, and when this later is corrected, their program stops working.

Code sharing can take place at different levels: libraries, functions, modules, subsystems, classes, binaries and interfaces are all examples of possible approaches. In the library case, commonly used functions are lifted out and standardised in often well-tested collections of high quality. Users of the program can use the library functions to perform its operations in their own programs. For example, many programming languages have built-in libraries to manage commonly used standard operations. In this way, not every programmer is required to manage for example input and output streams. In addition to language libraries, there might be other external third party libraries that are used for some functionality. The same concept can be used by companies to manage general operations that many of their products have in common. The developers' focus can then be placed on the truly product-specific differences.

The sharing of binaries can be motivated and described in much the same way as for libraries. However in the case of the binary the shared component is compiled and available as is. It can be tested and will be highly available for use if produced in a good way. In the case of sharing source code the main purpose is to avoid having to write the same code again and again. It is similar to sharing libraries but the source code bits that are shared are generally much smaller and does not have as much rigorous procedures surrounding them. Sharing in this manner allows developers to pull in some piece of code that for instance deals with some interface handling or does logging of an event.

The thesis is done in cooperation with Praqma and a company engaged in the embedded systems industry and the hope is to highlight general insights about shared code as well as make specific recommendations for code sharing to the embedded systems company. This company will henceforth be known as the case study company. Praqma is a consultancy that helps their customers with configuration management and they have specialised in continuous delivery. The company has relationships with a range of enterprises and has found that many of them have problems with their management of shared code or that they sometimes do not utilise code sharing to the full potential.

In this report, the term shared code is continuously used, since the term is ambiguous an explanation is required. In the scope of this thesis, shared code is the act of sharing source

code, modules, binaries, or libraries between products, projects or teams. The definition should not be confused with that of the extreme programming practice where shared code or collective code ownership means that everyone has the ability to edit all code [2]. This practice is not included in the term shared code since sharing code is more related to the act of actively sharing components, libraries, etc. Collective code ownership does to some extent enable source code sharing but does not deal with the other parts of shared code. Another thing that might be included in shared code is variants, but again this practice is, while in some ways related to code sharing, much more about adapting a component to for instance a different operating system. Here one could argue that the variants are using shared code (depending on how the variant is handled) but in general variant handling is not the same as shared code since there are other ways of handling variants than using shared code and if code sharing is used, it is a tool used in the variant handling process.

2.2 Problem statement

In this section the intentions and purpose of the work are outlined, the three research questions that the thesis seeks to answer and the hypothesis are motivated. The section also serves as a description of why shared code in general does not work as well as it could and many of the benefits outlined in the previous section remains elusive to the companies using it.

There are a lot of companies that can see benefits with sharing code and wants to reduce their costs, what is lacking however is an understanding of what it is that makes a project start to think that they need shared code, and also how can you support and organise it in a good way ensuring that the costs does not outweigh the benefits. It seems as most projects and companies solve the problems as they come and in some cases simply fail to see that there is a problem. This might lead them down a path where they abandon the shared code since they feel that they only get problems from it.

As the work on the thesis was starting the intention were to investigate the phenomena of code sharing and try to understand why a company would like to start doing this or why they chose to start with it in the past. The main reason for wanting to understand this was the fact that Praqma had noted that many of their clients got into problems when they introduced shared code. While discussing the potential drivers and motivations that a company might have it became clear that some aspects of shared code is pure advantages and disadvantages that on their own are not enough to drive a company towards usage of code sharing. Because of this the first research question, RQ1, was split into two parts. The two parts focus on the two different aspects mentioned above, what drives a company to use shared code and what advantages and disadvantages are there when using it:

RQ1.

a. *What are the underlying reasons that makes projects use shared code?*

b. *What advantages and disadvantages does usage of shared code cause?*

As the main interest was focused on the drivers and reasons for using shared code the obvious next question was to see if the problems and disadvantages could be mitigated. It is important to emphasise that there could be problems that does not have a solution that alleviates it and there might be cases where the problem should be left without any attempted solution, since it would only serve to make the problems worse or the problem is simply a manifestation of another underlying problem. The second research question, RQ2, was formulated to try to understand how and if the problems with shared code could be worked with:

RQ2.

What are possible ways to mitigate the problems with shared code?

As alluded to above the question tries to answer what possible ways there are for mitigating a problem with shared code but does not answer if the problem should be mitigated.

When formulating the two first questions it became clear that sharing code for most companies is a fact, and since Praqma was working with a company that was in the beginning of their code sharing journey it was decided that a part of the thesis should focus on investigating how a company could work with shared code. The idea with RQ3 was to make a case study at a company to try and understand their problems and try to use the results from the previous two research questions to make some recommendations as to how the company could work with shared code.

RQ3.

What recommendations for choosing a tool (or tools) can be made to solve the issues with shared code that a specific company have?

This question was originally posed to look at tools and see what they could do to ease the problems of shared code. This soon became only a part of the question which came to be much wider in its scope encompassing things such as processes, architecture, workflows, etc. A factor that contributed to this was that it became clear that no tool alone could solve all of the problems and that the way of dealing with the problems lay in combining many different things, of different type, at different levels.

During the initial discussion about the subject of the thesis there were some discussion about if a driver for shared code was the real driver or if it only was a manifestation of another underlying driver. This discussion led to a feeling that a projects architecture was connected to the project starting to use shared code. Due to this the following hypothesis was formulated:

H1.

A big part of the problems with shared code is due to underlying architecture choices!

The hypothesis was considered to be somewhat related to the first research question and it was decided that the focus of the thesis would lie on the questions, but keeping the hypothesis in mind. This was done so that the thesis would not become to large and because the hypothesis was quite vague, making it easier to focus on the questions and answering the hypothesis should something related to it show up during the work.

2.3 Methodology

Here the methodology used for answering the research questions is discussed and analysed, starting with a brief overview of the methodology and analysis of alternative approaches that was disregarded. Following this the chosen methodology will be discussed and motivated in more detail.

2.3.1 Overview

The work on the thesis was done employing a multitude of different methods to try and cover as much of the shared code concept, from as many angles, as possible. To start of the work on the first research question there was a need to investigate shared code more in depth. To do this a short literature study was conducted, this served as a basis for the first round of interviews that were held after this. To make the most of the time and to get as good responses as possible from the interviews the literature study and interviews were intertwined, effectively iterating the two. This served the purpose of both deepening the knowledge and understanding of shared code by seeing both the industry world and the academic world at the same time.

After this the literature study was extended to elicit solutions to the problems found during the first part of the thesis, as a part of investigating the second research question. This approach was taken since many of the texts studied when eliciting problems and drivers mentioned potential solution to the problems, thereby giving the search a head start when attempting to find more solutions. In combination with this a survey was sent out to see which advantages and disadvantages that are most important in the industry. This was done to try and get some more generality to the results since the literature study mainly provided a focused look on some specific things and the survey took a more general approach. The survey asked the respondents to rate different statements regarding advantages and disadvantages elicited in the first phase of the thesis. Since one of the things that came up during the interviews and literature study was tools, mentioned as the solution to the problems that were found, a tool study was done to see if the available tools could support shared code. This study focused on source code sharing tools since the availability and general maturity of package handlers for sharing binaries was deemed to be much higher.

When both the drivers and problems as well as the potential solutions had been studied in general, the attention turned towards making a case study at a specific company and investigating RQ3. This was done based on interviews that were conducted with employees at the company with the goal of finding out what their experience with shared code was like and what they would like it to be in the future. The results were then analysed and synthesised to create two scenarios that described the two main situations at the case company. Based on these scenarios two recommendations were devised, one for each scenario, based on the collected information during the thesis. These two recommendations were evaluated by presenting them to the company and asking for feedback on the applicability of them. The main reason for doing this was so that the company would get a sense that the recommendation was based on their situation and not based on misunderstandings.

During the work on the thesis there were some alternative approaches that were considered but ultimately disregarded. One of these were the idea to make the work more theoretical and base all problems and drivers on a major literature study and not do any or very few interviews. This approach was decided against since it felt that it would lack a industry perspective. Another approach that was abandoned was to conduct the interviews in one round instead of multiple. The decision to steer away from this was partly due to having some issues with the timing of the work (being partly done during the summer months), partly due to our non disclosure agreement that was delayed due to vacations, and partly due to not being able to analyse and discuss what was said so that the next round of interviews could cover any missing parts or focus in on interesting things that was not noticed during the interviews.

Lastly the original idea was to make some sort of evaluation of the recommendations at another company that was not part of the case study but had some experience with shared code. This was intended to see if the results would hold up in a more general case and not only at the specific case study company. This approach was abandoned partly since the recommendations felt so specific to the company they were tailored to that the decision was made that the other company would not add any useful input on the recommendations. However the main reason due to not following through with the idea was that the time left was prioritised to focus on making the recommendations more in depth since it would have taken considerable time to prepare something that could have been evaluated at another company.

2.3.2 Chosen methodology

Literature study

In order to gain a broader perspective on the chosen subject, shared code, and also to get an understanding of what previous studies had come up with, a literature study was conducted. The main purpose of this was to find out what the prevalent research had discovered in terms of motivations for using shared code but also to get some indication on what advantages and disadvantages and solutions for shared code problems these previous cases have had. The idea was that these ideas could be incorporated and built upon during further work.

The study could have been carried out as a single major literature study, but this methodology was rejected because the idea was to start from something specific and comprehensible and then, if possible, generalise the results. It was decided that method triangulation should be used and that the literature study and interviews in particular should complement each other. In particular, it was decided that they should both be undertaken as an iterative process as both methods would benefit from it. The interviews gave ideas for new literature studies and new areas of focus for it. Similarly, interesting insights from literature reading gave ideas worth exploring during the interviews. This synergy was not considered to be achievable if the methods were conducted in isolation. The literature study was also an important basis for the development of the requirement list for solutions to shared code as well as for the development of the proposed solutions for the problems with shared code.

To find literature a search process was adopted where the initial step was a broad search using Google scholar [3] and LUBsearch [4] with every keyword and search phrases thinkable. After finding some initial literature that had some promise this was used as a stepping stone to find more relevant information. To do this the references of the found text as well as their keywords were used to do further searching. This proved to be a successful method that was repeated several times to find additional material. Because there are vast amounts of information, both printed and digital, an effective selection process was needed to find truly relevant publications. Rapid checks of title, abstract and introductions were made in attempts to select interesting work. This approach was selected since at the beginning the goal was to just find some information that could be built on and then as more and more information was gathered the selection process could be made finer until mostly relevant literature was found and checked for information. This also gave the added benefit of exploring some leads that turned into dead ends which helped with giving the search direction, and fine tuning search word and phrases.

Once a text was read a short summary of the data was written to have something to go back to when analysing the literature. The summaries were focused on what the main points of the text was, and also had a section where all found advantages and disadvantages were listed.

Interviews

In order to get the most accurate description of what problems companies face when sharing code, interviews were conducted with employees at Praqma, the company used in the case study, and with some other developers that were met during the work on the thesis and had experienced code sharing before, thus being able to share insights into the pitfalls of shared code. The persons that were interviewed was chosen both due to them working in a context that gave them some relation to code sharing and due to general expertise. The goal of the interviews was to find out what developers, designers, etc. deem to be problematic and what solutions they have to the problems they face. Other goals with the interviews was to gather material to create the scenarios for the recommendations to the case study company and to complement the literature in order to elicit the requirements that sharing code pose on an organisation.

The initial plan was to conduct the interviews as described in the above section about the literature study, this had to change due to delays in acquiring a non disclosure agreement, with the case study company, and also due to the fact that many of the persons that were to be interviewed were on vacation. Because of this the interviews at the case study company were postponed to a later date while interviews with the others continued.

The interviews were somewhat prepared in advanced by using a semi-structured approach giving them enough flexibility to be adapted to the interviewees responses. Semi-structured interviews was chosen above the structured and loosely-structured. They enabled a interview approach were the questions were not completely firm and gave room for questions adopted to the situation and interesting followups. A set manuscript would probably have become too rigid, and allowing an interview to take any direction may have overlooked interesting areas, and it would be difficult to tie together.

The questions were produced based on initial information from a pre-study and were refined between the interview phases when new insights were obtained. Earlier overlooked questions were added to the script along the way. The questions were thus in a number of versions, and their number varied from about 20 to 30. Some typical questions, like: "At what level do you want to share the code? How much shared code do you want? Who creates shared code? Who decides code should be shared?" were asked. The interviews lasted for about one hour, although some were shorter. They were recorded to simplify succeeding analysis, which consisted of several rounds of listening to the recording followed by a compilation of themes for each interview.

Survey

The main reason the survey was created and sent out was that there was a need to get some sort of feeling for how developers in the industry thinks about shared code with its benefits and drawbacks. The idea was to try and see in what direction the community at large were leaning to when asked which things are most affected by shared code. In this way, the main focus of the company could be preserved at the same time as broader evidence was obtained, since findings could be assumed to be more universal if other people also experienced them.

The chosen approach was to create an online survey with several statements that the respondents were asked to select if they agreed or disagreed with the statement. To ensure that the respondents did not have an alternate idea of shared code a description of the concept was included at the beginning of the survey. The survey was distributed by posting it on a social platform and sharing the post with contacts working with software development.

An alternate approach would have been to create an offline version of the survey that was printed out and handed to developers. It would also have been possible to just ask one or two questions that asked the respondent to list and rank what benefits or drawbacks the respondent knew about in the context of shared code.

Ultimately the online version was selected due to it being simpler to administrate and analyse. The offline version could potentially have garnered more and better responses since the respondents could have been more precisely selected. This is the major drawback of the online version since the tactic of posting the survey on a social media platform and then sharing it with potentially relevant people may result in people not seeing the survey or deciding that they do not have any relevant information to contribute by answering.

Tool study

The tool study was primarily done to further the understanding of what the needs of a developer wanting to share source code are, as well as to get a feel for how well the available tools perform in a source code sharing scenario. When the tool study first was thought of the idea was to try to cover several different types of tools and to see which was most appropriate to use in the context of shared code. This was later reduced to only be a

study of tools that deal with source code sharing. This was done due to time constraints and because of the realisation that there were already several well established tools for sharing binaries and libraries thus leaving source code as the not, so well, covered option for sharing code. The tool study of publicly available tools was seen as a good way to investigate how well tools can support a developer in his work with shared code. It was assumed that a subset of the problems with shared code could be solved by tools and the study would provide some indications of those. The tool study mainly concerned tool functionality with only a minor focus on the process.

The study was done by using the requirements elicited based on the interviews and the literature study to elicit scenarios that a code sharing tool would need to support in order to be considered a tool that supports source code sharing and not only enabling it. Then the individual tools were tested by going through the different scenarios and seeing if the tool could do what the scenario stated. The tools were considered to support the scenario if the solution to it was not to difficult to understand and use, thus slowing down the developer.

Besides the original intent of having many more tools in the study another approach that was considered was to try and make a sort of prototype which would support code sharing in some way. This approach would have allowed for experimenting with shared code much more in depth but was decided against for the same reason as the number of tools were limited, time and wanting to focus on other things.

Evaluation

The results obtained during the work had to be constantly evaluated. In particular, this concerned the four biggest deliveries of the work, the solution matrix, the requirement specification for a solution, the possible solutions and the recommendations for the case study company. All deliveries were evaluated through weekly meetings with the industrial supervisor, during which the obtained results were discussed.

The solution matrix and the recommendation for the case study received slightly more formal evaluations. For the evaluation of the matrix, representatives from the industry were invited to a brief presentation of a draft of the matrix. The presentation were accompanied by a discussion, during which comments were received on elements of the matrix and some possible additions were suggested. This was an important evaluation tool because opinions from a wider audience could be obtained and incorporated.

When the recommendation for the company was produced, a procedure was needed to evaluate its applicability and effectiveness. For that purpose, a presentation was conducted at the case company where the scenarios, needs and recommendations as we perceived them were presented. After the presentation the present employees were given the opportunity to comment on the presentation and give feedback on the correctness of the scenarios and also what they thought of the given recommendations.

Chapter 3

Underlying reasons – advantages and disadvantages

This chapter is an analysis of the collected data from the literature study, the interviews and the survey. The data and the subsequent analysis of it, resulted in requirements a solution should satisfy. Beyond that, possible driving forces, advantages and disadvantages associated with shared code are analysed and the used methodology is discussed.

The chapter is especially important in the search for answers to research question 1. That is, the investigation of drivers that cause companies to use shared code, as well as possible advantages and disadvantages associated with it. It is of major importance to understand what the problems and advantages with shared code are, since these will dictate the requirements for the tools and processes that will enable usage of shared code.

To get an understanding of the reasons shared code is used, or in the instance of the case study company what their purpose with introducing shared code is, interviews were conducted with employees at the case study company, Praqma and a senior developer from an external company. In addition to these interviews, a literature study was made to give a broader perspective of the subject.

3.1 Interviews, literature and survey

The beginning of this section aims at motivating the interviews, the interview guide, the selection of interviewees, and the survey. The primary goal of these was to obtain data regarding the problems, drivers, advantages, and disadvantages with shared code. In addition, these activities also had as their goal to find potential solutions to the problems and

to get a picture of the situation at the case study company. The goal of the survey was to expand on what was found during the interviews and litterateur study, to try and see if the results were universal and not specific to for instance an interviewee. Following the motivation of the activities the section continues with a discussion of the data obtained from the interviews and the literature study. Lastly the data from the survey is discussed and the most interesting results are analysed.

3.1.1 Interviews

The interviews aimed at getting data on why you want to use shared code, its advantages, disadvantages and its prominent driving forces. They were started early on in the work process and the sessions were conducted in a semi-structured and open-ended manner, because they make a wide collection of qualitative data possible and at the same time lets the interviewee speak relatively freely [5].

A selection of interviewees was made, to allow a broad data material based on the interviewees experience. The interviews were held at the case company, Praqma as well as by video-chat with a senior external developer. At the case company two software developers and one employee with a more overarching role between projects and teams were interviewed and at Praqma consultants with different roles and expertise were interviewed. One worked mainly with automation trains, another with tools and the two remaining were resource consultants. An advantage of the chosen scheme was that it was not limited to obtaining only a picture from one professional role in a specific company. This meant that the data collected became nuanced, especially so since the consultants at Praqma had experience from multiple companies. Another advantage with the iteration of the interviews was that some persons could be interviewed multiple times, this gave the benefit of asking more general question at the fist time and then focusing on more specific questions in the follow up interviews.

During the interviews a series of interesting facts emerged and there are a number of driving forces playing for shared code, one such is increased efficiency requirements. The competition of the market in which the case study company operates has become fierce, while technical height has risen, the market now requires additional and more advanced features at a lower cost. The company is in a transition period where they transform from being a pure hardware company to introducing more and more software into their products, which will perform functions traditionally performed by hardware. A steering platform provider swap, enabling a switch to a higher level programming language has allowed more conventional software development, with associated needs for configuration management. The case study company sees opportunities to meet the ever-increasing customer demands while keeping cost and lead time down by using shared code and they are hoping that duplication of work efforts can be avoided.

The prominent benefits of shared code, discovered, were: efficiency, more users or developers of the code, big winnings after the initial cost and the ready-made modules in development means that a project will get started quickly and get momentum.

Some indicated and possible drawbacks and barriers are: increased demands for dependency management, increased demands for coordination of changes and complicated versioning. Another barrier is how to guarantee and sustain compatibility between all products using a changing shared module. A bug can potentially entail devastating cascade effects and affect the code-base in an unpredictable manner. Further possible drawbacks are how to handle priority conflicts between products and the initial extensive conversion efforts to enable shared code. Finally, the modules that initially created the good momentum, could lead to technical debt (i.e. choosing a easy solution which will not work in the long run causing you additional rework and costs) in the long term which would likely give a slowdown equal or bigger than the created momentum.

3.1.2 Literature

The literature study was done as a complement to the interviews so that the two sources of information could benefit from each other. In addition to eliciting advantages, disadvantages, drivers, and problems from the literature it was also studied to find potential solutions to the problems of shared code. The results from this will be discussed and analysed in chapter 4.

A lot of the literature is not directly about shared code. This is most likely due to shared code meaning different things to different persons. In some cases the texts are more about code reuse which is closely related to shared code. Other text uses shared code in the Extreme Programming (XP) way, meaning collective code ownership [2]. During the reading of these kinds of text some effort was put into trying to discern if the problems or benefits of these practices were applicable to shared code.

Some commonly used keywords during the literature search were "shared code" and "code reuse" preferably combined with "configuration management" to avoid completely unrelated results arising due to ambiguity. When an interesting work was found, two ways were used to derive additional interesting works from it. By checking the references of the work and by looking at the works citing it. The keywords varied over time when different areas were explored. One days insights could lead to a focusing on a specific area for the next days efforts. At times, the overall list of good keywords was developed into a large collection to later be limited to what was really found to be of interest.

Having done the literature search the material was divided into categories based on their expected relevance. This proved to be quite a challenge since the relevance was primarily based on the texts abstracts and in some cases their titles. To try and make the relevance even better the read literature was sorted once more after having been read.

Results

In their papers [6], [7], Danfoss writes about their experience with introducing shared code. The company started with a smaller code platform (at approximately 60 % of the codebase) that was managed by a dedicated team. This platform was worked on and eventually grew to contain 100 % of the codebase. The main reasons Danfoss mentions as to

why they wanted to introduce shared code are: an increasing number of product variants, decreasing time to market, and reducing development cost. From their experience of introducing shared code Danfoss noted that the shared code should have a dedicated team that is responsible for the code. This is to avoid problems with who is allowed to make changes, who decides what to change etc. They also noted that there has to be support in the architecture of the product so that it is flexible enough to be able to be reused in many products. From their experience Danfoss notes that the main advantage with shared code is the speed at which a new product or product variant can be developed, they also note the benefit of not having to constantly reinvent the wheel by re-writing code.

In his blog post [8] Daniel Westheide writes about sharing code between microservices using libraries and the problems that he sees with that. Westheide gives two major reasons as to what the drivers for using shared code are. One of these is "shared domain model" and the other is "infrastructure layer abstraction". Westheide argues that these two both are manifestation of the urge to avoid repetition. The first of the two, the shared domain model, is related to the design of the software on a model level while the other relates to the wanting of commonality in how certain things are done, for example database access. In his post Westheide argues that there are some drawbacks to using shared code, some of which may hit very hard. The major problem according to Westheide is dependencies to the shared libraries, these can cause problems when one is updated and the developers do not want to update causing them to create "workarounds". It might also be the case that in some cases multiple versions of the same library are needed causing what is known as dependency hell, since the dependencies might have dependencies of their own and so on. Especially ominous are diamond dependencies, caused by the fact that a product indirectly requires multiple versions of the same library. This scenario makes it very difficult to change the product because all libraries have to be changed at the same time. To combat the problems of sharing code Westheide suggests designing small libraries, with minimal dependencies, based on emerging similarities and patterns instead of using a top-down approach and pre-design them. He also suggests that library updates should be optional to avoid developers trying to circumnavigate the library.

Koratkar et al. [9] reports about their experiences with introducing shared code (albeit using a slightly different definition of the shared code concept compared to in this thesis) for building virtual Observatory software, citing economics and skill as their main motivations for using shared code. In their case they have come to the conclusion that building the wanted software will require both skilled developers and also large funds. The text names benefits of shared code such as: inexpensiveness (for users), tested, more developers, better standards, better and clearer code. In regards to drawbacks and problems they cite among others: increased demand for coordination, higher initial costs, less control over the code, suspicion towards code not developed by the team or project.

Jeff Whelpley writes in his blog post [10] that sharing code when there is only a few developers are quite easy, and that it is only really when the team starts to grow that the sharing becomes a real problem. In Whelpley's opinion there are four parts to sharing code efficiently at scale: multiple code modules that share some code, multiple team members, high rate of change, little to no loss of individual productivity. These four are in his opinion bound to cause problems when trying to scale up. He later describes a few areas in which his company had problems when going through this process and mentions to name a

few: versioning, refactoring, testing and size. Whelpley's problem with versioning is that updates to shared code becomes difficult to manage and suggests two approaches: force the update now or delay the update to the future.

In the book Software engineering written by Ian Sommerville, advantages and disadvantages for code reused are presented and motivated [11]. Increased dependability is mentioned as an advantage because the reuse code should have been tested and used in past products and its bugs and faults been corrected. Reduced process risk is another mentioned advantage. The cost for the developed software is known and it may be easier to estimate the project cost, if the project is reusing components than to estimate time and money demands for development of a new component. Effective use of specialists is also mentioned, by allowing experts to encapsulate their knowledge into reusable components this work does not need to be repeated in all projects. Lastly, the increased pace of development is addressed and Sommerville argues that code reuse can lead to a higher pace since the development and validation time of the system is reduced when partitions of it are reused components. Some mentioned shortcomings for shared code are the not invented here syndrome, where reused code is given less trust by the developers. He also points out that it must be possible to find the shared code and understand what it is doing in order for other developers to benefit from it [11].

3.1.3 Survey

To get a sense of how developers working in the industry feel about shared code and to try get an understanding of what parts of the problems with shared code that hits hardest a survey was sent out. The respondents were given a short introduction to the concept of shared code and asked about what drives a company to start with shared code. Following this the respondents was presented with a series of statements separated into two parts, one for advantages and one for disadvantages. The statements were of the type: "The importance of traceability and history increases when using shared code", and the respondent was asked to rate their level of agreement on a scale from 1 (I completely disagree) to 6 (I completely agree). After each rating the respondent also had the possibility to comment on their response, stating any difficulties with understanding or interpretations of the statement. This approach to the survey was implemented after a dry run of the survey with a few employees at Praqma who tended to want to elaborate on their responses and also stated that a scale from 1 to say 5 i.e. a scale with a middle would potentially result in most of the respondent picking the neutral response due to indecisiveness.

The survey was sent out on a social media platform and shared with the help of Praqma employees so that it would get some spread amongst people working with software development. After the survey was posted it was left open for close to two months and was closed at the two month mark when no new answers had come in for a while. In total the survey got 21 respondents of which none was removed for not understanding questions or similar reasons. After the survey was closed the results from the statements was compiled into graphs showing the distribution of the answers per statement and the results from the question about drivers and any added comment to statements were compiled and analysed. In the coming sections 3.2.3 and 3.3.1 the most interesting results, are discussed, regard-

ing advantages and disadvantages respectively. All of the surveys result is available as two appendices with appendix A containing the statements and appendix B containing the graphs.

3.2 Drivers and advantages

This section discusses and analyses the driving forces behind the use of shared code. Both those stated in the interviews and those derived from the literature as well as those from the survey. The benefits of shared code for the organisation, project and individual developer are presented and explained.

3.2.1 Drivers

There is a distinction between driving forces and advantages. This distinction is made to see what specifically drives a company to start with shared code and to see what potential gain they can get from using it. A driving force is an underlying motive for introducing shared code. While an advantage is the achieved positive effects compared with other alternatives. Below follows a discussion of the found drivers behind shared code. These drivers were mostly found during the interviews, with supporting material in the literature, and some were discovered by answers to the survey question on what drivers the respondent associated with shared code.

Transition: There are a lot of companies that traditionally have been working exclusively with producing hardware with only minimal software to control the hardware. This is shifting and moving toward a more software focused environment where the same hardware components are reused on several products which then leads the companies to want software abstractions and interface to hardware shared between projects to avoid redoing the same code over and over.

Increased competition and customer demands: Increased competition and customer demands have led to an escalation of technical height and demands for added and more advanced features in the products. There is also a lot of customisation being offered to customers so that they can tailor the product to their own specific needs. This calls for a more flexible software that can adapt to the different configurations while still being cost effective.

Change of control platform: As more and more advanced technology becomes available to the companies new possibilities open up in terms of what is possible for the company to do with the new features of their technology. In the case of the case study company a recent change in control platform supplier enabled them to use a more versatile programming language. This has enabled them to employ more advanced programming practices such as the possibility to share code.

Remove code duplication: One of the main motivations that is mentioned again and again is to remove duplicated code. This ambition is clearly motivated by looking at the

costs of maintaining the same, or almost the same, code in several different products while possibly not knowing all the places where the code is duplicated.

Make all products use a common core: This idea of having a common base for all products where libraries for doing for instance logging or network communication are available to all developers so that a new project can start without having to start from nothing. Instead they can hit the ground running with most of the needed base functions already available to them. This also has the added potential benefit of putting more strict quality control on the common part since it use so widely used.

3.2.2 Advantages

In this section, the advantages for shared code are discussed. They emerged from the interviews and the literature study. To give some hint as to which of these the discussed advantage come from the advantages found in literature are presented with a citation to the source it came from, whereas the ones coming from the interviews are left without citation.

Bugs: The larger number of users increases the chances of detecting bugs. The shared code makes the bug fixes immediately available for all products. In addition, separate bug fixes are avoided for each project or product that had been required if shared code was not used. It is beneficial for, especially, the projects and the organisation.

Higher velocity: The development rate becomes higher when ready-made modules can be shared. It is primarily a benefit for the organisation since it creates competitive advantages. The organisation can quickly adopt to market changes and may release demanded features or products before their rivals [6], [7].

Efficient variant handling: The shared code allows better management of variants and customisation of the products. The common code can be shared among the products and possibly be expanded to support specific features. It is an area where the organisation and projects are the biggest stakeholders [6], [7].

Avoid duplication of code: Efficient sharing of code and previously implemented solutions is desirable. This gives advantages over the entire field, it helps developers in their daily work supporting product variants and they do not have to remake already developed solutions and code sections. For the organisation and the projects there are economic benefits [8]. In some cases when not much code is shared or when the shared code is not used, duplication of code could prove to be more beneficial.

More users of the code: The shared code sections lead to more users of the code, which means that more people develop, test, and otherwise contribute to the code. It raises the quality of the code and leads to a more efficient resource management. This is primarily an advantage for the individual developer and projects [9].

Great return in the long term: Shared code entails potentially high initial costs, but in the long term it gives major benefits and modules can be shared between products

and projects at a very low cost. Primarily it gives advantages to the projects and the organisation.

Momentum early on, for the user: Projects that take in previously developed modules for further refinement to their product quickly gain momentum. It's an advantage for developers because they can focus their work on the truly product-specific areas and it brings cost benefits to the projects [9].

Economy, maximise the investment return: When companies invest in features and products, they naturally strive for a high return. By sharing code, such projects can be implemented with smaller budget appropriations because large parts can be shared multiple times reducing the cost per module [9].

Components can be shared at a low cost: Components can be shared and layered on for adaptation to the various products at a low cost. It is an advantage for the organisation, the individual developers and the projects [9].

Reduced development and maintenance cost: Because code and solutions do not need to be rewritten several times, development and above all maintenance work is less labour intensive. An advantage for the organisation and projects [6], [7], [9].

Establish standards: Once the shared code has been established and achieved a certain degree of maturity, it can evolve into de facto standards. This can simplify the work of the developers since there is a clear way of how to do things [8], [9].

Clearer code: Knowing that the code will be used widely within the organisation can make developers comply better with conventions and design patterns. It raises code quality and will benefit developers and projects.

Specialists can encapsulate their knowledge: Specialists can encapsulate their knowledge in modules that can then be used by others. It benefits the projects and developers who do not need to attempt to implement functions they do not really have expertise in [11].

Reduced cost and more advanced features: The shared code may ultimately lead to more advanced products while keeping costs in check. The hope is to shorten the lead times. It is primarily an advantage for the organisation [6], [7], [9].

3.2.3 Survey

The two statements regarding advantages that got the most interesting graphs were: "Using shared code increases the speed of development." (S2) and "Shared code leads to more efficient variant handling." (S3), shown below in figure 3.1 and figure 3.2 respectively.

As can be seen in figure 3.1 there seems to be a consensus that sharing code increases the speed of development. There are a few answers that does not agree with the statement indicating that not all think that shared code leads to speed increases but most are in agreement with it. This is interesting since there is also much that points to sharing code as a factor that slows down development. Based on what was said during the interviews

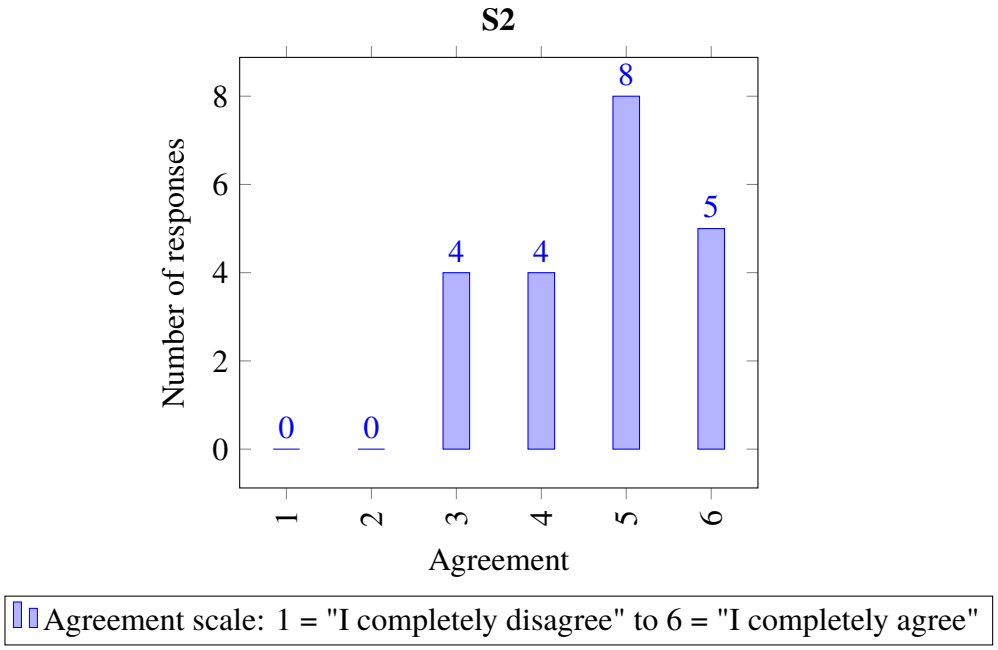

Figure 3.1: Graph showing the response distribution on statement S2: "Using shared code increases the speed of development.".

it seems as if the primary slowdown comes fairly early on in the adoption of shared code and after this initial slow period the speed starts to increase and stays at a increased rate. Because of this it is possible that some of the respondents are referring to this when they have selected to not agree with the statement.

The free text responses supports the theory that it is the initial work that takes time stating that "shared code is an investment", "it does not save time the first or second time it is reused" and "producing it is slow, using it is fast". Some answers also notes that the speed is very much dependent on the quality of the code that is shared stating that it is important to control dependencies and ensure that the code is tested.

From figure 3.2 it can be seen that the answers is very much spread out evenly over the field not pointing in any specific direction. As the statement is quite loosely formulated it is possible that this led to some different interpretations on what was meant with it. It is however interesting to note that despite this potential confusion the problem of variant handling does not seem to be impacted in a clear way by sharing code. The original thought behind the statement was that variant handling could benefit from shared code by having the common parts shared and the specifics separated.

From the free text responses it is clear that there are different views on what constitutes a variant and how they should be dealt with. Some state that there is a risk of the shared code not being flexible enough to be used in a variant context and that it would require too much work to manage the shared code to make sure that the code does not become too complex. In addition the point is made that dependencies would become hard to deal with, having multiple variants depending on the shared code potentially breaking some or all variants with a change to the shared resource.

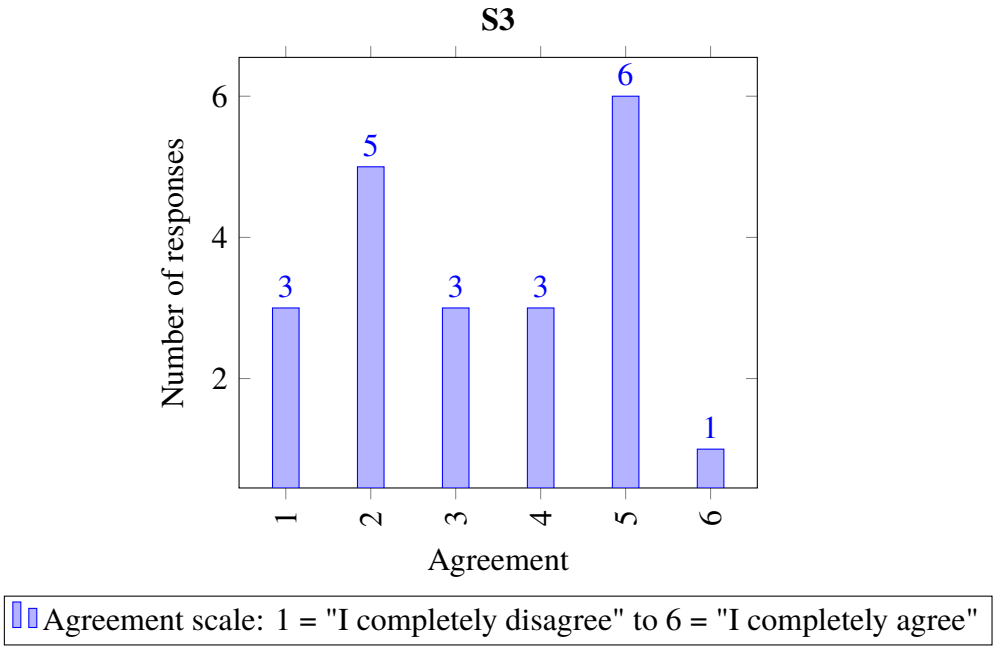

Figure 3.2: Graph showing the response distribution on statement S3: "Shared code leads to more efficient variant handling.".

3.3 Problems and disadvantages

In this section the problems and disadvantages that arises from usage of shared code are discussed. It seems like some of these problems are strictly related to the usage of shared code and some are things in other places that becomes problems when shared code is introduced. From the problems below, coordination, slow change process, prioritisation and infrastructure stands out as problems not directly related to shared code. Instead these problems are amplified and needs more attention when code sharing is used. For instance if a company have some, perhaps unknowingly, problems with coordination this might become a much bigger problem when sharing code since now there needs to be much more communication as to what is done by who and when so that the shared code can work. When not sharing code it would have been possible to not coordinate between teams since they were working on separate things with no relation.

In this section the drawbacks of shared code usage are presented at three levels: organisation, project, and individual developer. This distinction is made to try to see where the problems hit and also where efforts should be made to help alleviate the problems.

Coordination: When sharing code it is important to coordinate what is done so that everyone is on the same page in terms of how things are going to be done. This is especially important since the larger the company and number of developers the amount of communication paths grows according to: $n(n-1)/2$ [12]. Should there be no coordination, developers that are implementing shared code might do so in a way that are extremely difficult to use by others. There is also the need to separate coordination between developers and coordination between teams or projects, it is easier to

coordinate between developers than between teams. This is a problem mostly affecting projects and developers, it is however a problem that might need organisational changes and therefore it might have an impact on the organisation as well [6], [7], [9], [10].

Initial costs: When introducing shared code it requires, in most cases, a major reorganisation and a lot of work to get the existing code into a state where it can be shared. There is also the question of who will pay? Depending on the company this might be the project that wants to start with shared code, but in some cases the company might have decided that shared code should be introduced company wide and then the company is paying. The problem affects projects and organisation [6], [7].

Slow change process: Should changes take a long time from change request to implementation it is likely that some developers will try and work around the problem or if it is a more critical change they might be stuck making no progress until the change is made. The individual developer has less control over when some of their code is changed. This is in some ways similar to external libraries provided by a third party, updates may take a long time and the individual developer have no influence over how it is changed. This problem affects the developers [6], [7].

Traceability and history: It is important to have something to stand on when attempting to share code, in some instances there might be a need to see where and how a component has changed. If a developer is trying to debug their code they might have a need to know what have changed in the code so that they can narrow down the search for the problem. If the problem lies in a updated library it is important that this can be discovered by the developer in an easy way. This problem is affecting developers.

Maintenance: It is easy to test a change in isolation but if the code is shared it becomes difficult to try the patch since there are a lot of factors affecting the code, this can be a cause for a delay in patching the software. There might also be a problem if the old parts that made up the release that is to be maintained are not readily available, perhaps due to only saving code and not binaries. This problem affects developers and projects.

Compatibility: If a shared library gets an update the modules/projects that use the library needs to be updated to stay compatible with the library. The alternative to this is to stay on an old version which will then cause a version-control problem. This affects developers [8], [10].

Build processes: Sharing code might cause the build process to become harder since in many cases the shared resource is located at another place relative to the project that is using it. If all the code is in one repository it is easier to get all the parts that are needed but the repository can become huge and unmanageable. On the other side if the code is spread out on several repositories then it becomes harder to manage the build processes but easier to manage the code. This problem affects developers and projects.

Bugs: Should there be a bug in the code that is shared it will affect all of the modules/projects that are using it, and depending on the severity of the bug it might

be more or less harmful (there is no avoiding bugs entirely). It might cause a developer that is implementing a feature for a library to hesitate on doing so since it might cause problems for a lot of others. There is also the question of which developer or team that is responsible for fixing the bug if the bug appears in a widely shared component. This affects developers.

Dependencies: With a rising part of the codebase being shared it becomes increasingly more important to manage dependencies so that you suddenly does not end up in a situation where everything is dependent on everything and nothing works. This affects developers and projects [8], [10].

Version-control: Sharing code puts higher demand on version-control so that it is possible to actually have access to the shared code in an easy and effective manner. Each developer would like to have this and that library but not the other and for this library they would not like it to be updated. Alongside this the code, binaries, etc., needs to be organised in some way that enables the artefact to be retrieved and used in a expeditious way. This affects developers and projects [10].

Prioritisation: In some cases it might be necessary to prioritise implementing one feature requested by one project over another project's requested feature, this will cause delays for the other project. This affects projects and developers but it may become an organisational problem if it is handled poorly [6], [7].

Too strongly coupled with libraries: In some cases the strong coupling between a component and a library is not a problem but in others it might be problematic if the library itself have dependencies that might cause strange behaviour or bugs in the component using the library. This affects developers [8].

Infrastructure: There is a need for an infrastructure in the form of tools, automatic tests and processes that can support working with shared code. It can cause problems if the infrastructure is not designed to support the sharing. In some cases it can lead to problems since the needed tools or processes are not in place to support the development, since sharing code likely will put a larger strain on the existing tools or processes. This affects organisation, projects and developers since lack of tools or similar can be a major hindrance to the development and need every part of the company to work together in order to solve it [6], [7].

Discoverability: Should there be a high amount of shared code it might become a problem for a developer to be aware that there is a ready made implementation that they can use. This may casue the developer to repeat work that has already been done by others. This affects developers.

Too much focus on the shared code: In some cases there might be a strong focus on the shared code, leading to the code that is not shared getting less attention and therefore dropping in quality. There is also a risk that the benefits of quickly using a shared library to make something can come back later and cause problems, creating whats known as technical debt. This affects the project in a small way but mainly the developers.

3.3.1 Survey

Regarding disadvantages and problems with shared code the two most interesting results from the survey were: "Sharing code requires a lot of initial work." (S11) and "The build process becomes harder when using shared code." (S15). The graphs for these two are shown below in figure 3.3 and figure 3.4 respectively.

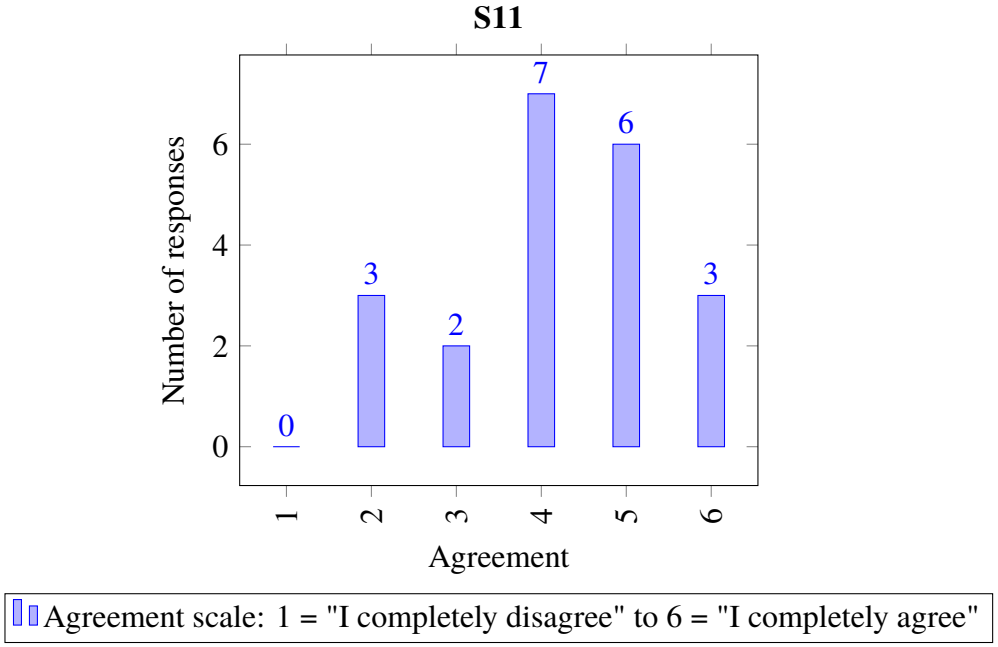

Figure 3.3: Graph showing the response distribution on statement S11: "Sharing code requires a lot of initial work.".

From the responses to statement S11 shown in figure 3.4 there does not seem to be a clear direction to the answers however there might be a slight bias towards the responses agreeing with the statements. This distribution is slightly surprising since one of the main things found in the interview and literature analysis was that shared code would require more work when transitioning to start using it.

This statement is also somewhat related with statement 2 which was presented in section 3.2.3. That statement received many free text responses stating that shared code requires significant initial work to ensure that the speed is kept up at a later stage. In contrast this statement did not receive any free text answer indicating that respondents felt it was clear. The fact that many stated that there are initial work needed in a previous response seems to somewhat contradict what the results here are showing but it is also possible that this is a consequence of the low number of responses making the numbers unreliable.

The graph showing statement S15 in figure 3.4 indicate that this statement was a bit divisive with the responses seemingly tending towards the edges. This might be related to what experience the respondent have with building software that uses shared code. If the respondent have had major problems perhaps due to a mismanaged build process or shared large amounts of code they will most likely answer that the build process becomes very

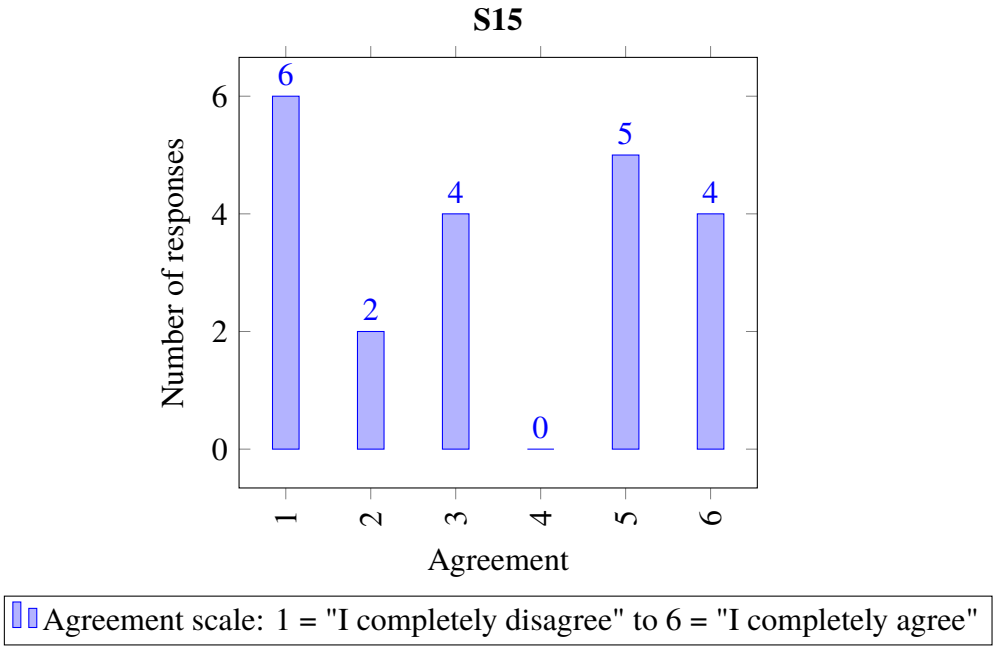

Figure 3.4: Graph showing the response distribution on statement S15: "The build process becomes harder when using shared code.".

difficult. If they on the other hand have shared only some small libraries or similar they would probably not say that the build process becomes harder with shared code.

From the comments in the free text answers the general consensus is that it depends very much on how you are working with the build process in general. One answer mentions that one exception to a more difficult build process would be if the company is using a monolithic repository and thereby simplifying dependency management, while another states that it very much depends on at what extent the builds are dependent on the shared code. This seems to reflect what was said during interviews, namely that when it comes to building it becomes very much a case of handling the dependencies.

3.4 Summary

It seems as if there are quite a lot of drivers for a company to start using shared code, it may bring quite a few benefits and might help the company increase their success. It is however not without its problems and might cause problems that if not dealt with can lead to major disruption and ultimately the company losing more that they would have by just continuing as before.

Sometimes it seems like it would be more beneficial to simply duplicate the code, and if the code is not to be used very rapidly in prototyping and experimental development or if it has more than a few users, it seems to generally be a good idea to avoid the repetition and share code instead. Advantages of joint testing and development can then be taken

advantage of. However it does not make sense to take the extra efforts to produce shared code if the use case for it is very limited, then is it better to keep the repetition.

One of the drivers that was either mentioned or implied in a lot of the literature is efficiency [6], [7]. This driver seems as one of the most logical since the urge to improve would be a fundamental aspect to any company. The main reason for it not being mentioned explicitly during the interviews is probably due to efficiency being a bit of a umbrella term and in most cases people focus on what gave the efficiency boost (lower costs, shorter time to market, less code, and so on). Another important driver that probably is equally or even more important than efficiency is to remove code duplication. This driver seems to be at the core of most people and companies decision process when deciding on if they should or should not use shared code, it was also frequently mentioned as a driver for shared code in the survey responses.

It is interesting to note that by using several methods for finding advantages, drivers, and disadvantages many more drivers have been found. Many interviews and survey responses focused more heavily on negative aspect to shared code while the litterateur was more positive to it, perhaps since the authors to the text was thinking of the concept in a more theoretical view.

From primarily the interviews but also the literature it seems as if no solution to the problems with shared code can last forever. This has a lot to do with companies becoming bigger or smaller, changing direction and so on. In many cases it seems as if the companies are a bit unwilling to change their chosen solution and therefore gets into other problems that could have been avoided, had the solution been changed.

The next chapter, chapter 4, considers solutions for the problems of shared code. It aims at finding possible solutions that suppress the disadvantages as far as possible and the starting point is the list of requirements (see next section) from this chapter.

3.4.1 Requirements for shared code

Organisational support: There should be a commitment from the organisation to support shared code, this is mostly due to the initial cost but also due to the need for organisational support, so that the developers feel that the organisation is ready to adapt to the change in developing practices. The individual project will most likely require new processes and these will take time to get acquainted to, and will therefore lower the projects output for a time when making the switch. There may also be a need for the organisation to step in and dictate procedures and standards that are to be used so that the code sharing is done in a uniform way throughout the organisation. An example of this is that there needs to be a clear way of dealing with changes to the shared code.

Structure of responsibility: It is important to clarify who is responsible for what code, should a situation arise where some change needs to be made to a shared component it needs to be clear who should make this change. This is to prevent a situation where developers either does not change the code since it is not their responsibility

or everyone changes the code even though only one or a few should change it.

Guidelines for code sharing: There must be some guidelines on how products and developers share code. The guidelines should be clear and applied consistently to avoid problems. The level of which the code should be shared must be stipulated and guidelines regarding how to organise and work with the code must be specified.

Tool support: The tools must give a broad support and ease; coordination, build processes, dependency management, traceability, awareness, maintenance, access to code, history and versioning.

Requirements on solutions: A solution that in some way helps alleviate a problem or a part of a problem needs to satisfy some requirements to be considered an effective solution. These are the following:

- **R1:** It should have a small or no impact on individual developer productivity, i.e. give good engineer efficiency.
- **R2:** It should give easy access to shared resources.
- **R3:** It should support large codebases, multiple products and several teams.
- **R4:** It should have good performance.
- **R5:** It should be easy to switch to another solution.
- **R6:** It should be easy to learn and use the solution.
- **R7:** It should be effective, i.e. avoid duplicated code and support sharing of code instead.

Chapter 4

How can shared code be supported?

This chapter builds on the requirements from chapter 3 and uses them to investigate possible solutions to the problems and situations associated with shared code. Each solution's advantages and disadvantages are discussed and analysed to form a solution matrix that can be used to quickly find a solution for a given context. To ensure that the most interesting solutions are covered an iterative approach is used, giving a brief discussion on the wide range of the found solutions and then reducing the solutions down to the most interesting, i.e. the ones that seemed to have the most promise in solving the problems of shared code, and going more in detail on these. In combination with this the results from the conducted tool study are discussed.

4.1 Solutions

In this section different solutions to the problems identified in the previous chapter are discussed and evaluated against the requirements list. The solutions are analysed to find what their respective advantages and disadvantages are. The section begins by outlining all the elicited solutions and then proceeds to discuss the first selection of solutions and discuss these further. Finally the second selection of solutions are discussed and analysed. The selections were in both cases done based on what solutions seemed to have the best possibility to solve problems with sharing code. After the first selection it became clear that two of the selected solutions seemed more apt at solving the problems and these were therefore selected for further investigation.

4.1.1 Overview of solutions

In this section the elicited solutions are discussed and linked to the requirements in section 3.4.1, chapter 3, in some cases stating that the solution fulfils none of the requirements. This is due to the solution having the character of a supportive solution and the requirements are focused on solving parts of shared codes problems. Some advantages and disadvantages are discussed for each solution and possible pre-requisites are analysed. The solutions were found by analysing both literature and the interviews, and in some cases solutions came from discussions amongst the authors. This section follows the same principle from previously when dealing with sources, meaning that some of the solutions come from literature, indicated with a citation, and some come from the interviews.

Conditional compiling: Conditional compiling could be used to enable source code sharing by simply switching in the different part of the code that the product needs. It is done during compile time by indicating to the compiler which parts of the code should be included. This is quite a rough instrument for sharing code and does not allow for anything other than source code to be shared. It may also make the code very difficult to understand when everything is put into one file with switches that the developers have to determine if they are active or not. This solution also has a built in problem in that the language compiler (or interpreter) need to be able to use conditional compiling for this solution to work [6], [7]. This solution could be used in a company producing hardware that have similar software in many products or even uses the same software for most products with only small changes being made for the ones using different software.

This solution complies with the following requirements: R1, R2, R6, R7

Composition model: The composition model as described by Peter Feiler in his report is a model for how a configuration management version control tool works [13]. Using this model the user of the tool can specify a number of selection rules in order to create a specific configuration. This allows the user to simply specify that they want component A, B and C and that they want the latest version that is tested and so on. Using this model can however be tricky since it requires knowledge when setting it up and the user also has to know what different components they want so that they can be specified. It would perhaps, depending on the tool, be possible to configure some kind of user profile to ease some of the burden of using the model, however in a fast moving development this would probably be difficult if the developer needs a tweak to their profile every other day since something new has come up. The solution would require access to a version control tool that supports the model and training in how the tool is used to ensure that developers can work with it. It could potentially be used by a smaller team that have been trained in the model and know how to apply the selection rules.

This solution complies with the following requirements: R2, R3, R4, R7

Design patterns: There are several design patterns that could be helpful when building software that is to be shared. In most cases these patterns are helpful regardless since they help maintain the principles of low coupling and high cohesion. This solution

does not directly solve the problem of sharing code, rather it is more helpful in making the code that is to be shared more tailored to it. For instance a pattern could be used to make a code snippet depend on an abstraction instead of on a implementation thus enabling swapping of the underlying implementation and the possibility for users to share the component via the abstraction. A drawback of using design patterns are that many of the most useful patterns are highly dependent on the language supporting the constructs used to achieve the pattern, usually this implies that the language should be object oriented [14]. Another advantage is that design patterns are easy to learn and can be used in many different situations.

The producer consumer pattern is a pattern that can be used to simulate third party software within a company. Using this pattern the team that is producing a shared resource will act as the third party and release new updates to the resource when these are ready. The consumers of the resource will update their code to use the latest version when they feel ready to do so. The advantage of using this structure is that the producers have a clear responsibility for the code that they produce and the consumer can choose when they would like to update and possibly get compatibility problems [15].

This solution complies with the following requirements: R1, R3, R4, R5, R6

Multi-repositories: Using multi-repos (also known as poly-repos) different projects, components, libraries, or whatever it may be can be separated so that they each live in their own separate repository. This has the advantage of enabling different developer teams to be responsible for their own repository and its contents. It might however lead to the number of repositories skyrocketing out of control and cause the company to lose their overview. This is a problem that is caused primarily by wanting to have a product built from components from many of these repositories. Using multi-repos gives several advantages such as easy responsibility and ownership management, while not requiring anything particular to start using it.

This solution complies with the following requirements: R1, R2, R3, R4, R7

Microservices: As with multi-repos, microservices can serve as a good way to separate things into smaller units which enables easier management in terms of testing and maintenance. Using it also gives the possibility for function encapsulation so that multiple projects or other services can utilise the same service for a function instead of implementing it themselves. The drawback of using microservices is that they do not scale that well and will (like the multi-repo structure) become to many and therefore the developers may lose overview and might end up in a situation where they are only fighting capacity problems in the existing services instead of developing new features [16].

It is possible to generalise the usage of microservices and only rely on some general form of api, such as REST-api. This would allow for deviations from the microservice design principles and only use the basic idea of having modules communicating with each other.

This solution complies with the following requirements: R1, R2, R3, R4, R6, R7

Dependency management: Managing dependencies is always important, especially so when sharing code since shared code often is included as a dependency in another module. This solution is more of a must than a solution since it is something that has to be done regardless if sharing code or not. It is also more of a supportive solution than a solution to problems with shared code.

This solution complies with the following requirements: None

Responsibility structure: As the shared code is used by many it is crucial to have someone responsible for the code so that it is clear to everyone who does changes to the code and who maintains the code. This person or more likely team will serve as a gate keeper ensuring that the quality of the shared code remains intact. This solution does potentially slow down the change process for a shared resource since the responsible team might have a lot to do and not getting to a certain change request that a developer is waiting for, causing the developer to lose momentum.

This solution complies with the following requirements: None

Automatic build, test and release train: This solution is a supportive solution that have the benefit of giving the developers automatic tests that will tell the developers that the code works, or not. Since sharing code can cause a certain module to have unexpected effects due to it being used at a unknown place the tests will catch that something broke and stop the release from going public. The drawback is that having this level of automation requires initial setup and maintaining in order for it to work properly. This drawback is in someways countered by developers spending less time on building, testing and releasing thus freeing up time to maintain the automation instead.

This solution complies with the following requirements: R1, R3, R4

Architecture: When sharing code the architecture becomes more important since the different components need to fit together in perhaps unforeseen of ways. This can be achieved by not designing too much of the details of the architecture up-front and rather let the natural progression of the software dictate what becomes a shared component and what does not. One thing that is important to keep in mind is to ensure that interfaces are properly designed and maintained. This is so that the users of a shared resource will not have to be subjected to more interface changes than necessary and so they can keep their implementation even if the resource's internal implementation changes. This solution requires that there is someone keeping a close eye on the architecture as it evolves and responds to any changes that needs to be done.

This solution complies with the following requirements: R1, R3, R5, R6, R7

Awareness and traceability: Using shared code the developers need to have a good way to find available modules and other shared resources that they can use in their work. If this is not possible they might lose the benefit of the sharing since they will rewrite existing code due to not being able to find it. Another part of the awareness and traceability concept is that a developer that is about to make changes to a piece of code would like to know if someone else is directly dependent on this code and

also if the changes that the developer wants to make will cause the other persons code to break [17], [18].

This solution complies with the following requirements: None

Organisational structure and policies: The main role of the organisation is to support the development process, this is also true when sharing code. If the organisation is well put together some problems can be avoided that would have otherwise hindered the development. For instance if an organisation is very monolithic the architecture of the code will most likely be similar to this, according to Conway's law [19]. This might cause problems when the company wants to share code since as discussed in the architecture solution the design of the software should be designed to help with the sharing. Another example of where the organisation needs to support the development is in the beginning of the process with introducing shared code since it will most likely require resources. Other organisational benefits might be things like company wide processes or policies that ensures that components, functions, etc., are thought of in the same way and therefore easier to share. There is also a need to make some sort of decision on how to finance the production of shared code since it in most cases is code that will not be a standalone product and can therefore not finance itself.

This solution complies with the following requirements: None

Sharing binaries: One of the major advantages with sharing binaries is that they are easy to test and with this a potential added benefit is high quality since the binary can be well tested. However sharing binaries have the drawback that the binaries can not be changed, this might cause a problem if a change needs to be made, leading to a new binary having to be created. Despite the drawback sharing binaries can be achieved fairly easy with the use of a binary repository manager.

This solution complies with the following requirements: R1, R2, R3, R4, R6, R7

Coordination: Coordination can be vital to a company using shared code since as mentioned a developer that is not in the loop on what shared resources that exist will just waste time re-writing things. If teams and developers can coordinate their different activities, shared code can get a higher usage and the teams might discover new things that may be shared and save further time and resources.

This solution complies with the following requirements: None

Change control and prioritisation: When more and more projects and teams depend on a shared resource it becomes increasingly more important to have a rigorous change process that can evaluate incoming change requests so that a selection between necessary and unnecessary changes can be made, especially if the changes will affect the interfaces of the shared resource.

This solution complies with the following requirements: None

Monolithic repository: The main advantage of the monolithic repository is that everything is available in one place. This enables developers to use code from different projects in an easier way. It might cause some problems with dependencies since a bug is

not as easy to isolate as in for instance a multi-repo structure. This does require all projects to share everything and not keep things in separate repositories, which might be a problem if some code is a bit sensitive, which might be solved by sharing binaries instead. This could also potentially be solved by combining the monolithic and multi-repo structures and thereby have some things separated out into their own repositories.

This solution complies with the following requirements: R2, R3, R6, R7

Configuration status accounting and bill of materials: These two solutions can be used in order to get traceabillity and knowledge of what a product is comprised of. Bill of Materials (BoMs) can also be extended to what is known as 150 % BoM which indicates potential other components that can work with a particular configuration [20], [21]. The main disadvantage with doing these things is that it takes time and personnel that the company might not have to spare, but to have some sense of how the shared code is used it would be very beneficial to use these solutions.

This solution complies with the following requirements: None

Do not share code: In some cases sharing code is simply not the answer. It might be that the shared resource is stretched so thin that it is better to split it into several parts and make those fit their purpose instead of trying to make a square peg fit in a triangular hole. Also if the code is just used momentarily to maybe prototype something there is not much point in going into sharing of the code, instead a copy of the code will serve the prototype just as well and it can, if the prototype should lead to something interesting, later be included in the shared code. There is however still the problem of finding the code that can be copied which might lead to re-writing instead.

This solution complies with the following requirements: R1, R3, R4, R5, R6

Sharing libraries: Sharing libraries gives access to already made functions and depending on the library these might cover almost all the wanted functionality. In some cases only one or two functions of the library is used and the rest goes unused which leads to very much dead code in the library. There is also the possibility that the libraries become incompatible with the code, which might be a problem if the program is dynamically linked. This can be avoided by using static linking on sensitive software with the price of the program becoming larger. The ability to push the linking to run-time (i.e. using dynamic linking) might not only be a problem since it can enable bug-fixes to happen in a library without the program having to be recompiled.

This solution complies with the following requirements: R1, R2, R3, R6, R7

Feature toggles: Feature toggles is used to enable or disable functions in an application while running. It allows functions to be included in the code and only activated in those products where it is needed. There are some similarities to condition compilation but this is done during execution. The toggles allow for a step-by-step implementation of features and integration does not have to be tied to periodic realeases. Instead, functions can be included and then tested and later on be activated only when they are ready. A built-in disadvantage is that technical debt can be accumulated, due to bad code not being fixed immediately. The toggles are merged with

permanent code and they are sometimes untested which constitutes a risk factor in the codebase. In addition, there is a risk that the toggles will remain turned on or turned off by mistake [22].

This solution complies with the following requirements: R1, R2, R5, R6 and R7.

4.1.2 Selection of solutions

As can be seen from the section above there are quite a few solutions that in some way contribute to ease the problems of sharing code. However there are also too many to be able to investigate all of them in depth, and because of this a first selection was made in an attempt to be able to investigate some solutions further. In addition some of the listed solution are not really solutions in their own right but rather supportive things that might help but in smaller ways. Following the first selection a second selection was made after some reading on the first selection, this was done to enable a really in depth analysis on a couple of solutions that seemed promising.

This first selection was done based on several factors: the potential of the solution, how well it seemed to solve the problems with shared code, and how interesting it was. Using these factors the selection was made as follows: the composition model, design patterns, repository structures (i.e. the monolithic and multi repository structures), and microservices.

The second selection was made to include repository structures and microservices as these two seemed to be the most promising when it came to actually sharing code practically. The composition model was deemed too be to complex for the average team and design patterns while promising is ultimately dependent on language specifics and not strictly a solution but more of a supportive design workflow. While investigating the repository structures a tool study was conducted to further investigate how a certain tool combined with a structure can be used to share code. This will however be covered in section 4.2, and will not be further discussed in this section.

The composition model

Peter Feilers composition model utilises a system model and version selection rules in order to create a specific configuration [13]. The system model lists all components that are included in the system and the selection rules are used to indicate what version of a specific component should be included in the configuration. The selection rules can be applied on both individual components and on sets of component, for instance a rule could be made indicating that components A, B, C and D should be using the latest version and components E and F should use version 1.2 and 1.5 respectively. It is also possible to select for instance the latest tested component by using multiple rules on a component which can be used for selecting variants. The composition model is an extension of the Checkin/checkout model, also described by Feiler, and thereby relies on locking and checking out the components that are to be worked on. It is important to note that unlike for instance Git [23], which uses another model (the strict long transaction model), the composition model dictates that the

components along with the system models and selection rules are the things under version control.

Using this model in a shared code context would probably work quite well since it would be easy to find a shared component using the system model and then it would only be a matter of updating the selection rules to get the component that is wanted. Since the model does not make a distinction on whether or not the component should be in a certain form the model could be used to share binaries, source code or libraries. However as mentioned above the model uses more of a traditional way of dealing with concurrency control by locking the files that a developer is editing, this might have severe impacts on the speed of the developer and in turn the entire company's speed. Also, having to deal with the selection rules, while giving the developer high control might also impact speed since the rules can become quite complex depending on the context. To summarise the composition model is a good model for sharing code but would most probably become too cumbersome in practice with developers having to be too hands on in their selection of components.

Design patterns

There are several design patterns that may be used to help with shared code. Many of them are however dependant on language features and sometimes requires that the language is object oriented which is a potential deal breaker for a company that are unable to use such a language.

In general the pattern that seems to be the most fitting for easing work with shared code is the producer consumer pattern. This pattern can be used to ensure that the code is maintained and developed in a good and structured way. It enables a team to be responsible for developing the code and releasing it when ready ensuring that there is some structure in the way the shared resource is updated. A team that is using the shared resource (i.e. the consumer) can simply retrieve the resource and use it, not having to worry about maintaining and further develop it. This pattern can be employed on a wide scale throughout the company giving all of the shared resources their own teams, which in turn frees up the product developers time so that they can focus on doing their jobs [15]. Other patterns that might be beneficial, of course depending on the specific company and project, are: the strategy pattern, the dependency inversion pattern, and the command pattern [14]. These three can be used to design the software in a way that simplifies the sharing of it later down the line. For instance the command pattern could be used to switch out different implementations of a function in an easy way. In general the principle is to use whatever patterns that seems to fit the project at hand the best but doing so with some thought so that the patterns does not cause more problems than they solve.

While there are a lot of different patterns a few general overarching ideas that could be used to ease the problems are: hide implementation details behind interfaces and ensuring that the interfaces remain somewhat rigid (so that no unnecessary changes cause a massive update frenzy), avoid creating unnecessary dependencies and ensuring that they are going in the right way (using the dependency inversion pattern might be helpful), designing everything up front will not work out, instead the design should remain fluid and adapt to new situations.

Repository structures

Depending on the repository structure that a company uses there are things that become easier or harder because of this structure. The two structures that are discussed here is the monolithic and the multi-repository structure. Both of these fulfils the the same purpose but they do it in different ways making the decision between the two more of a choice of what a company wants to be easy and what they want to be more difficult [24].

The monolithic structure keeps everything in one big repository with the advantages of being able to search the code for examples of usage, see and edit all code, reducing cognitive load on the developers having easy access to code. The cognitive load argument can however be made in favour of the multi-repo structure as well, but in this case the advantage is the smaller repository size giving the developers an easier time when developing. A disadvantage with this structure is that since everything is collected in one repository it is easy to become lost in the sheer amount of code. The monolithic structure also has the benefit of a new project being easy to create by simply adding a folder to the root folder of the repository [24], [25].

Using the multi-repo structure allows for every team to have their own repository which allows for greater flexibility. The team can use tools, libraries etc. however they wish while not having to worry about how it affects other teams, this allows the team to work in way that they feel is optimal for them, likely increasing their speed. Another advantage of using this structure is that it is possible to be picky with what to download, since you can select only the projects that are interesting (this is in some ways similar to the composition model where you can pick and chose what you want). This structure is not without drawbacks and one of the biggest of these is the fact that a team's product must (in most cases) be consumed as third-party software which can cause problems with for instance bugs or feature handling. If a consumer notices a bug in the product they will have to either ask the producer to fix it or create a pull request, wait for it to be accepted and then they can incorporate the new fixed product in their code [24]. The other major drawback is that if the company is using for instance Git, that uses the strict long transaction model [13], the version control tool will lose much of its functionality when going with a multi-repo structure since the model no longer works when trying to commit to several repositories at once.

Lastly it is quite common to see these two structures combined into a sort of hybrid with several larger repositories that contain several projects usually grouped together after some commonalities allowing them to utilise similarities between projects.

Microservices

Microservices is a architectural design style that takes the principle of localising responsibility to the extreme. The idea is to create small independent services that can interact together to form a system. This idea is a response to having everything collected in one massive system, which is generally referred to as a monolithic architecture. The advantages of using microservices are: each service is independent and can be scaled when needed

based on load, microservices enables good failure isolation and since each service its own contained unit it is easy to assign responsibility for it.

Microservices can be combined with other practices to garner even greater benefits. Examples of this are using the multi-repo structure described above or using Continuous delivery as a way of developing the services. The multi-repo approach would serve to further isolate the code for each service making it very easy for a developer to maintain a good understanding of what a service consists of. Using Continuous delivery enables the services to be deployed automatically so that new changes and fixes become available as soon as possible. This has the benefit of increasing the speed of the development since developers do not have to sit around and wait for the next release to get hold of new features and bug fixes [16], [26].

The drawback of using microservices is that they need to be able to communicate with each other. Should the communication be disrupted the service will need to be able to handle that without breaking. This might cause major disruption for the entire system when one service is unable to deliver to others, sending ripple effects throughout the system. There is also the possibility that the sheer amount of services becomes too much to handle for the developers [27].

4.2 Tool study

It was recognised early that some problems cannot or should not be solved with tools but with well-designed procedures and processes. But it was also clear that for other problems very good tool support could be provided. In addition tools can support processes and make them easier to follow. The tool study aimed to investigate which problems tools could solve and how well the tools could support different scenarios for code sharing. In particular, a tool must support the stipulated requirements from chapter 3.4.1, that is, a tool must give a broad support and ease; coordination, build processes, dependency management, traceability, awareness, maintenance, access to code, history and versioning.

The study focused on tools for source code sharing. Not because it's the only way or because other ways were considered inferior, but because it was considered an interesting area to investigate. Library and packet sharing are also options but they did not make it into this work. The range of tools for library sharing was assessed to be larger and more developed than the source code equivalents. The composition model was a model that came up as a viable option for shared code, with the ability to select components to be included in one composition, it definitely has advantages. Unfortunately, the range of tools for the composition model is rather scarce. IBM's Clearcase [28] suite exists but it was excluded from the tool study for two reasons. First, the tool was deemed to require too much configuration management from users. The tool appears to be very capable but without solid knowledge in the CM-field it becomes cumbersome. Secondly, it is a highly commercial program and the license cost is high, which might be a problem for a smaller company. The Git Subtree [29] and Bit [30] tools were considered but consequently not selected to be included in the study, as subtree merges the history and treats it all as one project, it does not fit in, and Bit emerged as too difficult to use. Finally, two candidates remained. Git

Submodules [31] were chosen due to its built-in support and its ability to separate history and Google Repo [32] was chosen as a promising alternative to Submodules.

To evaluate the applicability of the tools and assess how well they support different activities, a number of scenarios were prepared in which the tools were tested. As a basis for the scenarios, a fictive project was used, which was relative small. Industrial projects are much more extensive and more complex, but for the evaluation purposes a very primitive project structure was adequate. The test environment was set up on two computers with one developer on each and the code was shared between them by a Git repository hosted at GitHub.

The scenarios were divided into three thematic parts. The first part aimed to examine the tool's ability to retrieve and push code to and from other projects. The second part examined how well they can present changes and history. The third was about the management of releases and maintenance of former releases. The total number of scenarios amounted to 15 and they are listed to their entirety in appendix C.

4.2.1 Git Submodules

Git Submodules make it possible to add submodules, which effectively means adding other repositories, to a repository. For shared code, it is particularly advantageous that the same submodule can be added to multiple repositories. In this way, code can be effectively shared [31].

Submodules supports the first thematic part about fetching and pushing code to repositories. When the change status of the whole project were to be shown, some problems were encountered. A simple "diff" only gave the status of the own project not for the submodule. The command "git config –global status.submoduleSummary true " improved the situation slightly but problems remained. To get an adequate history it was necessary to iterate over the files. To view the change history of the submodule, the active folder had to be the submodule. The active folder had to be changed again to see the same status of the own project. Retrieving the latest changes from a specific project went fine as long as the current submodule belonged to the project. If there are many projects, it can quickly degenerate and become unwieldy. Getting specific versions from other projects also went well with the same reservations as above. Removing a submodule no longer needed caused no major problems.

4.2.2 Repo

Repo differs from submodules in some aspects. Firstly, it is not built into Git, but is a python script that runs separately. Secondly, it handles the repositories differently and it is more difficult to integrate separate components because Repo makes them appear to be stored in the same place.

Before code can be downloaded from other projects, a configuration file must be prepared. It can be local, or global so that all users use it. A global file for all users can be used

for all developers, and each individual developer can overshadow the parts they need to in their local file. When the configuration file is in place, it is trivial to get code from other projects.

Useful support to see file status is available. You can add or delete files in multiple projects simultaneously with iteration. And there are commands to get the latest changes from specific projects. If a shared repository is no longer desired, it can be removed from the manifest file.

4.2.3 Summary and comparison

Both tools can be used to share code and the way they do it is similar to each other in several aspects but also differ significantly in other respects.

Google Repo uses configuration files to specify what to include in operations. It can simplify management for users because global files that can then be overshadowed in the places where there is need can be created. The price for this is the reduced ability to distinguish the components with their own life cycles because Repo makes everything appear to be in the same place. If that is not acceptable, submodules are the better choice. Both tools handle fetching, pushing and file status management for projects divided into multiple repositories and both of them can use iterative constructions for this. Similarly, conventional Git commands can be combined with both tools in situations that do not require global operations. However in the global aspect it is important to note that Gits long transaction scheme is broken, since it is not possible to make atomic commits.

The two tools can provide support for a number of problems. First, they form ways to share source code and gives developers access to the shared code. In addition, they contribute to increased awareness of the code because the code is available and the history of commits can be given at all times. In some sense, the build process is simplified because of the tools ability to pull the code together from multirepos.

4.3 Solution matrix

There are many different levels, within which a solution can be categorised. When choosing a solution, it is of value to choose a solution that is at a suitable level for the problem. To this end, a matrix was developed to clearly categorise the solutions and build a structure for overview. The matrix is built up as a 4x3 matrix with tools, process/workflow and culture/mindset as column-headings. The row-headings contain compilation, linking, execution and development. The matrix is two-dimensional because it was considered to be so that code sharing can be handled in mainly three ways. With tool support, through processes and workflows as well as through established cultures and ways of thinking that rules within the company. Subsequently, each solution can operate within one of four steps. It can be about the development phase, which is a broad concept involving the production of code and maintenance of existing code. It can also be a question of compilation of programs, linking or execution, all these distinctions are reflected in the matrix. In some

cells there are solutions followed by colon ":". What follows after a colon explains what aspects of the solution the solution involves.

Some shared code problems can certainly be eased with tools during linking, while others can instead achieve the same goals by, for example, dynamically linking libraries. It is however not always feasible to use tools. For some problems, sound and well-thought-out procedures can be better and even a prerequisite for success. By establishing and maintaining guidelines and a culture around the work of shared code within the organisation, a working approach to shared code can be achieved. The matrix aims to help in the selection of a solution for a specific context by giving an overview of the supply of solutions. It may seem attractive to choose solutions, so that complete coverage of the matrix is achieved. But it is important to point out that one should not overshoot the target and choose unnecessarily many and complex solutions for the problem. Such solution tends to become to cumbersome.

The complete matrix is shown in figure 4.1. Not all cells have been filled in since none of the elicited solutions was deemed to be applicable for them. Other solutions are divided into several cells because different aspects of them are covered in different stages. One such example is monorepos which are found both in tools/compilation but also in process, workflow/development. When compiling, development environments such as integrated developer environments often offer support for loading from many different repositories. At the same time, the monolith is also placed in process, workflow and development because monoliths can simplify development efforts by making all code more easily accessible compared to other options such as multirepos [25]. For example Google basically manages all of its code in a huge monorepo used by thousands of engineers. Almost all the code is visible to all developers and they share a common suite of tools [25]. Another item that may need a more detailed explanation is "no shared code" placed in culture, mindset and development. It may appear to be a contradiction given that the work is about shared code, but can actually be useful under certain conditions, and effectively eliminating problems. For example, in prototype development and rapidly experimental development, setting up shared resources would be foolhardy and too burdensome in relation to the profits. Bill of materials has been placed in the intersection of process/workflow and linking. It is of utmost importance to ensure that everything that went into a component is well documented and specified. Therefore, it is important that the establishment of such a list is part of the routine workflow and that there are elaborated processes to maintain it.

Dynamic linking of libraries is found in the upper right regions of the matrix. It is a phenomenon that is somewhat similar to the classical static linking with the difference that the linking takes place while running the program and that the built-in components can be changed during the execution phase [33]. For this to work, support for this obviously needs to be provided by the tools used. Another tool-aided solution on the same theme is "feature toggles or "feature-flags". By sharing all the code in a common codebase and then enabling just the features that are required in each specific product, feature toggles can be used to support code sharing. The changes take place during execution of the program, which explains feature-toggles location in the matrix.

Conway's law found in the intersection of culture/mindset and development is also an important point. The law says that the architecture that an organisation produces tend to have

a design that reflects the organisation's structure [19]. Since the structure of the organisation is not necessarily the best design for the given program, it can be a very good idea to implement a culture within the company where you pay attention to the problem and try to avoid going into the trap. Microservices and REST-api are found by the intersection of process/workflow and execution. These two techniques are often combined so that a microservice encapsulates all the data for a function. The communication between microservices is then managed using REST-apis. It simplifies the work of communication between microservices by providing a lightweight mechanism and does not even impose any requirements on the language implementation for each microservice [34].

Share binaries found in the intersection of culture/mindset and linking is also an important point. It can many times be beneficial to share binaries because they easier can be tested and quality assured as a unit.

Table 4.1: Solutions matrix, structured as solution: problem

	Tool	**Process / Workflow**	**Culture / Mindset**
Compilation	Conditional compiling, If-def: compiler support, Composition model, Multi-repos.	Conditional compiling: code in compliance with it.	
Linking	Dependency management: link shared resources as dependencies, Static linking of libraries, Composition model, Multi-repo.	Sharing binaries, Configuration status accounting and BOM.	Share binaries.
Execution	Dynamic linking of libraries, Feature-toggle, Automatic test- and release chain.	Microservices, REST-api.	
Development	The composition model, Multi-repos, Monoliths: tool support, Git submodules, Google Repo.	Architecture: design patterns (Object oriented programming), Conditional compiling, If-def: writing code to support it, Monoliths: access to code, Company guidelines, CCB, Prioritisation, Division of responsibility.	No shared code, Architecture: Conways law and overarching design, Monoliths: "Conways law" and Organisational structure, Egoless programming, Coordination, Awareness: developers talking with each other.

4.4 Summary and conclusions

As seen during the chapter there are a lot of different ways of handling problems that arise due to shared code. In this section the main points from the chapter will be summarised and some key points will be discussed.

The selected solutions all have their merits and drawbacks. In the case of the composition model it would seem as it could serve as a good solution to problems with code sharing however the model is quite complex and does not really behave as tools that developers are used to working with. Users of it would need training to be able to fully use all the benefits the model brings. With design patterns it is the opposite case, it is a very easy solution

to understand and use but the patterns have the disadvantage of not enabling shared code, instead they make it easier to share code.

The two repository structures that have been discussed here both have benefits and disadvantages and in some sense the decision to use one or the other have more to do with the company's culture and general philosophy. It could perhaps be useful for many that are just starting out to keep everything separated in a multi-repo structure to ensure that the developers are not overwhelmed by the complexity of the monolith. But then again starting with everything in one repository could also be easier for a starting developer since they will have access to everything and can easily look at examples of how for instance an interface is used.

The microservice architecture solution is a very interesting take on the whole code sharing problem. It minimises the services so that they are only doing one single thing and thereby creating a structure where everything can be used by everything since the service is so highly independent.

One of the key takeaways from this chapter is that there does not seem to be one easy solution that can solve all of the problems that shared code has, instead it seems as the best way to go would be to use several solutions together to attempt to alleviate the different problems. Another aspect that is important to keep in mind is that many of these solutions require quite heavy investments in tooling and/or mindsets which means that the company leaders needs to be on board from the beginning, supporting the transition to shared code.

In this chapter the general case of code sharing has been discussed. The next chapter will look into one specific case at a company and give an example of how to apply the matrix and the solutions. This will result in recommendations on how the company can work with their problems and what they should think of when they move forward with their shared code.

Chapter 5

Recommendations for a company

In the previous chapter the focus was on solutions to problems with shared code in the general perspective, in this chapter the focus is made narrower and focuses on how a specific company could work with shared code. This chapter focuses on a case study at a specific company working with embedded software, in order to evaluate the results from chapter 4 by attempting to give some recommendations to the company. The chapter aims at analysing the company's situation and synthesise two scenarios with needs and problems related to shared code that the company have, the two scenarios will be based on the big and small picture respectively to try and capture the two main ways the company wants to share code. These needs and problems are then examined to find solutions that together form recommendations to the two scenarios, as well as an outlook on how these recommendations will work in the future.

5.1 Scenario 1 - The big perspective

This section deals with the broader perspective of shared code at the company. It aims to present the scenario and motivate which of the possible solutions from chapter 4 that might be relevant. In the end, the section will turn to a discussion of the actual recommendations.

5.1.1 Description of the scenario

This scenario deals with the big picture, i.e. the global level at the company where all products are included. The company wants the hardware to be accessible through abstractions and the products should be built from modules so that several products can use, for

example, the same engine module. The goal is for the built products to involve a larger portion of shared code. Several product teams are involved in the development and the total workforce encompassed by the scenario can amount to hundreds. Product teams can be scattered geographically and be located at different districts or even countries. The company operates with long product cycles and a delivered product must be maintained for a long time. Maintenance over decade periods is not infrequent.

Based on the studies conducted at the company, the following prominent needs appeared:

- An ability to abstract hardware.
- Support for modularised products.
- Support for distributed development.
- Support for long product cycles with extensive maintenance phases.
- It should be possible to share libraries in binary form, compiled binaries, as well as source code between products.
- The solution should be scalable.
- It must be possible to share code at different levels. From separate hardware functions to larger platforms.
- The company is bound to an plugin to their IDE that has some shortcomings regarding configuration management. Among other things, changes by Git operations on disk are not always recognised and automation must be carried out via COM objects, regardless it must be supported.

In addition to these needs there are others that were identified but will not be covered by the recommendations since they fall outside of the scope of this thesis, instead these are included as a list of things to keep in mind for the case company (and possibly others that want to start with shared code).

- Prioritisation of tasks.
- There is a need for a clear change process.
- Creating a responsibility structure.
- There is a need to keep tabs on what developers need early on in the development so that the design can adapt to this rather than the developers adapting and working around the design.
- The developers should have some way of seeing the big picture.
- Dependency management on a large scale.

5.1.2 Analysis of potential solutions

Based on the solution matrix from the previous chapter, possible solutions adapted for the scenario will be discussed. The multi-repos found in the development column is a

viable option, the repository structure means that each product or module is organised in its own repository. The structure facilitates ownership, fault isolation and Git operations. In cases where code from other product repositories must be accessed, the solution can be combined with tools such as Git submodules or Google Repo. Repo is not appropriate if separate components with their own life cycles are to exist because Repo makes repositories appear as one. This property disqualifies the tool from the recommendation because the components to be included in the products must have separate records of change history. This is so that the history does not get tied to the product instead of the component since the same component can be used by multiple products that all need access to the history. The composition model [13] undoubtedly has some positive sides. For example, the model offers good opportunities to select constituent components for a product. However, the tool range is very scarce, the Clearcase suite is too difficult to use and requires extensive knowledge of both the tool itself and configuration management at large. This solution would put too much burden on the developers.

If the focus is moved to tools that operate during the compilation phase, the matrix shows that conditional compilation [6] is an option. It is possible to include all code in a codebase using the method and then use conditional statements to include only the relevant features for each product. A major shortcoming is that the code becomes full of these constructions thereby increasing the codes complexity and the overview risks being lost. Sometimes it is beneficial to share binaries and libraries rather than source code. Such sharing can be realised during the linking phase. The binaries are shared via a binary manager and linked into the product. An obvious advantage is that it is easier to test these binaries as one unit which vouch for a high quality.

Dynamic linking [33] is another possibility, as well as feature toggles [22]. Especially feature toggles are very useful for rapid development with continuous integration because features can be included before they are fully tested and only be activated in test cases or for small groups. As the company in question operates with very long product cycles, feature toggles applicability is limited in this case. Automatic tests and release trains that also are found in the execution phase cell are important. To effectively share code between products, there needs to be quick and rational ways that ensures that everything works as intended.

Looking at the process and workflow column of the matrix, it becomes clear that during linking it is possible to share binaries. Traceability of all the components of a product is, of course, important. Bill of materials and configuration status accounting are ways to meet these requirements. Microservices linked together with REST-apis could be used during the execution phase. However, they come with all the shortcomings distributed systems have and are difficult to adapt to the company's architecture.

Moving right in the matrix to the last column, culture/mindset. There's a lot to reason about regarding the architecture. High utilisation of the producer and consumer pattern [15] is desirable because the pattern entails simpler testing and quality control and has a clear built-in separation of responsibility.

5.1.3 Recommendations

Multirepos is recommended because the structure simplifies ownership, fault isolation and Git operations. It also allows for different tools in the different products, a possibility that should not, however, be used as a practice due to reduced ability to knowledge sharing between projects and less direct reuse of, for example building scripts. In addition, a huge flora of tools rarely becomes a success. If a team has very good reasons to use another tool there is a possibility for it.

Libraries tested in binary form provide an effective way to share code between products. The sharing itself is done via some type of package manager. Dependency management is required because a certain lag in the updates is allowed. The tools must provide an overall picture and work globally to give project members good insight into what needs to be updated in the event of a change.

Bugs in shared libraries can have far-reaching negative effects, this is due to them being difficult to find and isolate when sharing code since, for instance a shared library can affect other code in unforeseen ways depending on where and how it is used. It is also possible that the bugs come from developers circumventing the proper change process in order to "just fix this minor thing" causing problems in unintended ways. Extensive automatic test suites can provide a high level of quality and compatibility assurance. To effectively guarantee this, “Test-Driven Development” (TDD) [14] is recommended. Furthermore, a high utilisation of the producer and consumer pattern should be sought. The pattern means that one team (the producer) delivers source code to another team (the consumer). The consumer team then develops new functionality around the delivered code. However, the consumer must not modify the components since this would cause the consumer to have to apply these changes every time a new update is released by the producer. The pattern provides easier testing and quality control and has a clear built-in responsibility division.

For version management, Git is recommended because the support is large among development tools and the built-in “submodules " is well suited for sharing code. The aforementioned plugin shortcomings in which it does not recognise all changes on disk after Git-operations and that COM objects must be used for the automation of builds renounce many of the benefits that Git brings and must eventually be replaced or improved.

It is important to note that submodules is built into Git and it supports sharing of code but it requires an investment in the learning phase and is likely to take time to reach its full potential. Another thing worth mentioning is that it is difficult to make the history work satisfactorily when working with shared code because the commits are not “one” i.e. you have to make several commits per change. This makes you lose one of the great things about Git namely long transactions with atomic commits [13].

With regards to updates the recommendation is to have a update policy where, a product team shall have the opportunity to wait a bit to update and halt at a specific version for example during the final testing before a large ”release". However, backward porting to older versions should not be allowed as it provides a clear incentive for development teams to update without much delay. The scenario requires that versioning of libraries and con-

stituent components is possible in the tools.

5.2 Scenario 2 - The small team

In this section the recommendations to the second scenario will be discussed, following a discussion and analysis on potential solutions for the scenario. This scenario complements the first by looking at how a couple of small teams can work with shared code. In this scenario, that is still a part of the bigger scenario, the most important thing is to work fast and effective.

5.2.1 Description of scenario

This scenario describes the small team that want to share code between a few developers working quite close to each other but not necessarily on the same project. The developers in this scenario would like to share primarily source code but in some cases it could be more beneficial for them to share libraries. The main workflow that the developers want to use regarding sharing code is: they would like to be able to get a snippet of code from some other project that they think would suit theirs, then they would like to adapt the code to their purpose and test to see if it works, after this they would like to be able to either discard the code should it prove to be unusable or they would like to send off the changes to the place where the code originated from. After this the cycle would repeat with another piece of code, and another, and so on.

During the interviews with developers wanting to work like this at the case company, the following needs were elicited as the primary needs for the scenario to work:

- Each developer would like the ability to get code from some place or places, work with it and then send changes back to the respective places that the code came from.
- The developer would like to share source code as much as possible.
- Speed is a big need in this scenario since the developers are doing quite a lot of prototyping and are changing from one thing to another quite rapidly.
- The solution should not take too much focus away from the development, meaning that the developers should be able to get the code that they want in an easy way that does not take too much time.
- The company uses a plugin to their IDE, this is needed to make the code work with their system platform. This plugin requires a certain structure of the project files and is does not recognise changes on disc automatically. There is also some difficulties with automating builds with the plugin.

5.2.2 Analysis of potential solutions

In this scenario there is a need to have some sort of tool that can support the development with sharing source code and looking at the solution matrix from the previous chapter (see table 4.1 for more details) this scenario's needs would place it in the tool/development square when it comes to tooling, this is due to the developers in this scenario primarily wanting to share source code which would not fit under compilation, linking or execution. In this square there are several available solutions that might be used to satisfy the needs of the scenario: the composition model, multi-repos, monolithic repo, Git submodules, and Google Repo.

The first of the solutions is the composition model, which while fully capable of sharing code in the way stipulated by the scenario it will not satisfy the needs of speed and ease of use since the model requires a bit of configuring before it can be used, and since the development is moving at a high pace the proposed solution to this, i.e. using profiles would not work.

The next solution is using a multi-repo structure which would be beneficial since it is not certain that the developers are working on the same projects and the shared code pieces would benefit from being separated out into their own repository making it easy to access all the shared code. On the opposite side of the multi-repo structure there is the monolithic structure which also could work in this context, however since the development should be as fast as possible and be as independent as possible it would be beneficial if all the teams had their own say on how they work which is best done with a multi-repo structure.

Since the multi-repo structure seems to be quite useful, and the developers want to have the ability to get code from several sources, either Git submodules or Google Repo should be included as a recommendation since both of these provide the possibility to get code from multiple repositories. Based on the tool study from chapter 4, both of these tools have strengths and weaknesses, however in the case of this scenario submodules seems to be the better choice since this tool is a bit more convenient for the developers to use due to it having a clearer distinction on what comes from where which is helpful when a developer might be working with code from several places and needs to keep track of what belongs where.

Only using tools will not cover all parts of the needs, for instance when trying to achieve speed in the development it is important to have clear procedures in place for how things are done, for example when splitting out a piece of code and making it shared. Many of these policies can be reused from the recommendations for scenario 1 above, but there is still a need to have some solution to how the given example should work and how to avoid being slowed down when only wanting to create a very quick prototype. One mindset that can be used to achieve the latter is to not share the code in the extreme cases and simply copy the code so that the developer can go ahead with the development without having to retrieve some components and set up their workspace to properly use the shared components. A potential workflow that could be used to solve the given example of how to create a shared code piece, is to when a code snippet that could be shared is discovered put it into a shared repository and rewrite the code so that it can be included in several project by creating good interfaces and so on, using design patterns. After creating the shared

code snippet the developer that created it can either, if they are not aware of anyone using the code, inform the other developers in a coordination meeting like a stand up meeting or similar, or if the developer is aware of someone using the functionality of the newly shared code the developer can create a pull request changing their code to use the shared code instead.

5.2.3 Recommendations

This section list the recommended solutions to the needs of scenario 2 based on the analysis and discussion on different solutions in the section above.

The first recommendation is to use Git and Git submodules with a multi-repo structure with the repositories set up so that every developer can access shared code. As discussed in the previous section a multi-repo structure allows for shared code to live in one or several separate repositories instead of having it mixed in with the different projects. This recommendation covers the need to share source code located at different places and the need for a simple solution that does not take too much focus off the development.

The need for sharing as much source code as possible is not explicitly covered by a solution, rather the solution with multi-repo and Git submodules enables the developers the flexibility to chose for themselves how much source code they would like to share. Should they want to share code in another form, for instance a library, the recommendation is to employ the same principles and tools as in the recommendation for scenario 1.

With regards to the need that is stipulated by the plugin for the IDE there will be no specific recommendation provided other than that the plugin should be improved so that it is easier to work with. This could be done by for example working with the developer of the plugin and suggesting changes or using some of the company's resources to help the plugin developer with making the changes.

In order to achieve a fast development process the recommendation is to have a workflow where developers breaks out a code piece to start sharing it when they notice re-writing or re-using some code piece, doing this would allow developers to work relatively fast and only temporarily be slowed down by the work needed to share the code piece. In addition to this recommendation another recommendation is to actually copy code when working with fast prototyping to avoid all of the slow down of the shared code. However this mindset might become slower if the developer is working on very similar prototypes all the time, in which case it might be more beneficial to start to share code instead. It is also important to note that if a prototype turns out to include a code piece that really would benefit other developers, the developer working on the prototype should be encouraged to put in the work to share that code.

In terms of pre-requisites there are no major ones for these recommendations besides teaching developers how to work with Git submodules and ensuring that the company puts some guidelines in place for what repositories are to be used for sharing. Other than that the main thing to be cautious of when implementing this is that if developers feel that they can not break out code to share it because of time pressure. This could be due to having a close

deadline on their own projects. To avoid this the developers should be actively encouraged to share code that can be shared.

5.3 The future

In this section the recommendations for the two scenarios will be briefly discussed with a future perspective since it is important to be prepared on how a given solution will work when the company starts to grow and needs to scale up their shared code usage.

In the case of scenario 1 the given recommendations are given with the future in mind and will probably work even if the company would scale up. However it is clear that at some point the recommended solutions will start to become problematic, it might be that the amount of repositories grows out of control or that new solutions that are much more effective are developed making the recommendations obsolete. If the problem is the amount of repositories, it is possible to combine repositories to the hybrid version that was discussed previously where the repositories contain multiple projects but not everything is in one repository.

As for scenario 2 the recommendations given here scale to some extent but since the recommendations are tailored to smaller teams working in a tight-knit environment the solutions will not scale that well, instead the recommendation would be to start introducing practices from scenario 1 to scenario 2 as needs arise and problems emerge.

In general it is important to keep a close eye on how the solutions are working and to not be afraid to change things as the company grows. When the requirements of the development start to change the company should be prepared to move away from a solution so that they are not stuck with a old half-working solution. It could be beneficial to start looking at new solutions as soon as the scaling starts, so that the new solution is ready to be used as soon as needed.

Chapter 6

Discussion and related works

In this chapter the results of the thesis will be discussed and related to the research questions posed in chapter 2, further the results will be related to previous work in the field. The feedback from the evaluation of the recommendations will be discussed and analysed. Potential future work will be discussed and the validity of the thesis will be analysed. Finally, some reflections of the thesis work is undertaken to see what can be learnt from the experience.

6.1 Evaluation

In this section the evaluation of the recommendations at the case study company will be discussed. During the evaluation, an assessment of the effects of a prospective implementation of the recommendations was made.

Following a final presentation and subsequent discussions at the case company, it became clear that the thesis scenarios reflected the company's situations fairly well. Likewise, many points in the recommendations were things the company previously had regarded. The fact that some points would appear was obvious because much of the work has revolved around the company. Nevertheless, the evaluation was necessary to investigate misunderstandings or errors due to the lack of perfect information. During the evaluation it emerged, that the company has a need to determine which path it will actually take, so that waypoints can be set up.

The company wishes clear guidelines on how a transition from the small scenario to the larger perspectives scenario can be undertaken because scaling is expected to be difficult but necessary. During the evaluation meeting it was mentioned that tools with the ability

to search in the shared code. It is an important point that unfortunately has not been further investigated due to time constraints.

6.2 Related work

This section puts the work in relation to a handful of previous work in the field. Results and conclusions from them are compared with this thesis.

A paper written by Koratkar et al. [9] presents experiences and tries to give some pointers as to what is needed to share code in a efficient way. They describe the need for code sharing when going towards Virtual Observatories (VO), i.e. observatories that uses software instead of hardware to make observations. Several motivations as to why the need for code sharing arise. In particular decreasing funds, improvements in developer tools and languages, and new development or upgrading of tools. They argue that sharing code reduces the need for developing similar code multiple times and therefore the winnings of shared code can more than compensate for these problems. As an example of this they mention the case of Hewlett-Packard that started reusing code and thereby could save over $ 56 million on a project that cost them around $ 26 million to develop.

Koratkar et al. present their experience from working with a project to improve code sharing and group collaboration. They give examples of practices and the results they had on the sharing of code. Among other things they mention communication and coordination as important parts of sharing code. These two together will take you a long way since you can be on the same page as others and therefore work together to avoid problems.

Other practices that are recommended are: responsibility management, making sure the interfaces are well defined, document the code, change handling, and company support both financially and also guiding. All of these practices can and will take time and also resources and it it therefore important that the leaders in the company are on board from the beginning so that the projects have all the support that they need.

Parts of the results of the work match well with the thesis work. Amongst them, the identified advantages and disadvantages of shared code. Koratkar et al. had a greater focus on the soft values, related to communication and cooperation than this work had. They also placed greater focus on organisational aspects and their study had a different starting position where they looked at an organisation that had introduced shared code.

In a paper by Jaspan et al. [25] the authors presents their findings from exploring what the advantages and disadvantages of a monolithic repository are, compared with a multi-repo variant from a developer perspective. In addition to trying to discern advantages and disadvantages the authors present findings about how the advantages with monoliths are actually utilised. The problem as described by the authors is that there are several bigger companies that are moving towards using monoliths but the authors finds that there is a lack of knowledge about advantages and disadvantages for monoliths and multiple (per projects) repos.

The methodology used by Jaspan et al. to find advantages and disadvantages is a survey sent out to employees at Google. They asked several questions about satisfaction with Google's own monolith but also about if the employee had any previous experience with monoliths or multi-repos and if so which they prefer and why. To try and see how (and if) the engineers at Google actually used the monolith to their advantage the authors looked at the logs from their tooling. To analyse the survey the authors used open coding of the employees answers to group the answers. This allowed the authors to see tendencies in the answers and also to judge which things were major advantages or disadvantages, and what things are minor.

From the survey and log analysis the authors found a few major advantages that most of the engineers agreed on: Visibility, Tools, Dependencies, and developer load. Developers that took the survey listed visibility as a major advantage since they could see examples of how APIs are used and it also enables them to go and look at code in all parts of the company's repository and thereby find examples of implementations and solutions to problems similar to their own.

Jaspan et al. spend quite some time on the tool aspect, they believed that this would be mentioned as a major advantage and tried to investigate if this was the developers liking a specific tool or if they liked the way the tool worked in the context of a monolithic repository. They conclude that tools might be more important than the repo structure, and that you should probably try to have "standard tools" since developers are not used to special inhouse tools and can lose speed while learning the new tool.

The results of the survey at Google showed that the engineers largely preferred the adopted monolithic structure over multirepos. An interesting result is that the few who preferred multirepos stated velocity as an advantage, which is also one of the listed benefits for monorepos. The fact that the monolithic structure was listed as more advantageous differs from the results of this work, at least as far as the recommendation is concerned. Which can be explained by the fact that the situation at a software giant like Google with a long tradition of software development cannot be equated to that of the case company.

In a paper written by Jepsen and Beuche [6], the concept of shared code at Danfoss Drives is treated. The company aims to build larger parts of its products from a common codebase. Danfoss decided that a higher level of reuse would be achieved year 2000. Reuse was first successfully introduced on hardware. Entrusted by those successes, software began to be shared. Initially, this was done through "Clone and Own" but it was not a satisfactory solution. Soon, a feature model with conditional compilation was introduced. Language support came through if-def statements and every feature had make files to activate or deactivate a feature. The product configuration consisted of a selection of the right feature-makefiles.

Jepsen and Beuche state that the changes did not lead to any major changes in the organisational structure, except that an Embedded Software Platform group (ESP) was formed. In some cases where something was developed or changed that would be used almost immediately in several products, the work was carried out in "task forces", where the members came from different product projects and were coordinated by an ESP employee.

The requirements for collaboration and coordination between the project teams are claimed to increase with the introduction of the platform. However, the authors have seen how several interesting features were developed by the projects in such a way that they were more or less ready to use without further work in other products. Thanks to the platform, these became available to all the products almost immediately. The authors also emphasise the freedom of the product projects to decide on upgrades as a balancing between fault isolation from external changes and a need for new fixes and features. Backporting features into older releases was not supported by the platform, giving a incitement to not lag behind to much.

The situation of the company is strongly reminiscent of the case company's and the results of the report are interesting because it is a reflection of how the implementation of shared code has gone at Danfoss. Danfoss introduced a group that became responsible for the shared code (ESP Group). Pronounced ownership of the shared code is something that this report has also found beneficial. Danfoss used if-def statements for handling its shared code, this report has gone another way and recommends other methods, among them library and binary sharing.

In the paper "New Challenges for Configuration Management" written by Larsson and Crnkovic [33] the authors have identified configuration management's lack of adequate support for component development and has presented theoretical solutions. High requirements are imposed on the maintenance of relevant and up to date information about the constituent components and their compatibility and compose a problem area for component-based systems.

When components are integrated into a system, there are mainly two problems according to the authors. First, it is not possible to predict the behaviour of the system in advance or the effects of an introduction of a new component or updating of an existing one. The second problem is the dynamic behaviour. A new version may work for one product but at the same time break another. To determine whether a component is compatible, there are several types of compatibility levels: in-output compatibility – the internal characteristic of the component is in this case not of interest, but only the in-output relationships. Interface compatibility – during an update, the interface remains unchanged but the implementations may differ. Behavioural compatibility – internal meaning characteristics such as performance, resource requirements, must be sustained, this is the strongest condition and it includes the others. The levels are used to ensure that an update does not compromise the functioning of the entire system. Different situations require particular levels of compatibility, according to the authors.

Larsson and Crnkovic suggest that a list of dependencies are maintained for each system to guarantee compatibility through changes. And suggest a three-step approach. First, the attributes of the current configuration should be saved. Then the installation of the new modules is carried out and a new snapshot of the new configuration is finally taken. These attributes include the compatibility changes allowed, date, time, and size. By comparing the lists with dependencies before and after the change, all changes can be detected.

The authors accentuate that interfaces often are public methods of an object in object-oriented languages. The disadvantage of it is that the implementation language is confined. Separation of the interfaces from the implementation through interfaces such as CORBA

and COM are highlighted as possible solutions. COM resolves the interface version management by defining interfaces as immutable devices. The connection between them is thus very loose and upgrading components in isolation will be easier.

Larsson and Crnkovic, like us, have identified the problem of compatibility of changes. They present levels of compatibility and a methodology to determine whether a change meets them. Such a deep study in this area has not been done in this work. Furthermore, the authors suggest the CORBA and COM standards for communication between the components, the most similar to that from this work is REST-apis. Which also integrate components without strong dependencies.

6.3 Future work

During our work we came across a lot of interesting things that we spent more or less time exploring. However, due to time constraints - and interest - we had to drop further development of these ideas or topics in order to have resources to get in details with what we chose to focus on. In this section we will very briefly present some of these ideas and topics that merit further work.

- During the evaluation of the recommendations to the case company, there was a discussion about how a developer should search and find shared code that they want. Based on this discussion there definitely exist a need for developers to have support from a tool that can somehow search for code based on its functionality.
- Based on the responses from the survey, there are indications that there are some things that affect developers more than other things. It would be interesting to make a larger survey possibly in multiple rounds to really get to the bottom with what problems developers have when they are sharing code. Another approach would be to select the most interesting results from our survey and try to dig deeper into those results.
- Since the tools that have been looked at in this thesis do not deal with history and traceability that well, one possible idea for future work would be to develop a new tool that is specifically tailored to share code.
- Since the thesis have had a broad perspective and not got into details with all topics there is a need to dig deeper in those which was only treated superficially. This can be done by either picking a specific result and digging deeper into that or by redoing a larger part of the thesis with a narrower scope, for instance only focusing on advantages of shared code.

6.4 Validity

In this section the threats to validity of the thesis will be discussed to ensure that any biases or weak points are clear to the reader and to show which conclusions are indicated to be strong and which might be weaker.

Literature study: The main weaknesses of the literature study is that it is difficult to find the right literature and that not all literature used for the work is entirely trustworthy since much of it is blogs or similar that is not peer-reviewed or written to be factual.

Interviews: The interviews suffer mostly from the wrong questions might being asked, this is however mitigated somewhat by iterating the questions that are asked in the interviews. The interviews also have a possible threat to their validity in the selection of the persons being interviewed. This point mostly affects the case study since there are quite a good selection of interviewees for the other topics the interviews cover.

Survey: The main threat to the validity of the survey is the low number of respondents (21) which make it difficult to draw any strong conclusions of how developers and persons working with shared code feel about the elicited advantages, disadvantages, problems, and drivers.

Tool study: Since the tool study is based on the situations described in the literature and the interviews, it is likely that the scenarios used in the study do not cover all scenarios that someone working with shared source code might encounter.

Evaluation: Since most of the evaluation has been informal or semi-formal with only the recommendations being more rigorously evaluated it is not possible to draw the conclusion that the claims made in the thesis would hold in general. Instead they can be claimed to be likely contenders to actual correct claims.

Generalise the case study: The work was carried out at only one case company. In order to draw stronger conclusions, more companies need to be examined.

6.5 Reflection on own work

Here we discuss and reflect on the thesis as a whole and try to explore some of our decisions and analyse what could have been done in a different way now that we know what the outcome of the decision were.

During the work on the thesis a wide perspective has been used to ensure that as many aspects of shared code as possible could be covered. This resulted in some loss of depth compared to what could have been achieved if the scope had been further limited and focused only on certain aspects of code sharing, such as problems with it. In retrospect the taken approach allowed for many interesting paths to be explored which a more limited perspective would not have allowed for, making the chosen perspective seem better suited to discover what shared code has in terms of advantages and disadvantages.

As with most things there are several things that in retrospect seem as the wrong decision, since when looking back all of the information is available to make the call on what would have been best. As an example of this some of the planning of the thesis would have been done differently if all information had been known, an example of this is the evaluations, however since not all information is available the decisions that were made can not be faulted.

One decision that have impacted the generality of the thesis work is the fact that the evaluation of the results was not done at two companies as originally intended. This was done as previously mentioned due to the recommendations to the case company not being relevant to evaluate at another company and due to time constraints hindering a proper evaluation of the rest of the material at another company. On one hand it would have been good for the validity to do more evaluating but on the other hand doing more evaluation would have impacted the amount of time spent on investigating shared code. However it would probably have been possible to do some sort of evaluation at some point had this been planned to take place at another stage of the thesis, instead of at the last part.

During the work the three research questions have been answered. The first question regarding reasons, advantages and disadvantages was found to have a number of aspects to it and these can be found in chapter 3. In the case of the second question it was discovered that there are no single solutions to the problems of shared code instead there is a need to combine solutions depending on the context that the code will be shared in. Potential solutions and a matrix for choosing a solution was discussed in chapter 4. As for the third question the recommendations made for the case company reflects what was found during the work and made use of the solutions matrix in order to give some recommendations to the company for how they could share code in the two scenarios that were identified. The recommendations can be found in chapter 5. No strong evidence for the hypothesis was found, but there are signs indicating that the hypothesis might be correct.

Chapter 7

Conclusion

One of the main themes that has emerged during the work is the plethora of different approaches that you can have when starting with shared code. While most cite speed and economics as their main reasons to start with the sharing, it became obvious that these two are far from the only drivers that motivates a company to start sharing code. While working with the drivers, advantages, disadvantages and problems it became clear that some things need to be in place for code sharing to work in a satisfactory way. This led to a requirement list in which an attempt has been made to try and give an indication as to what should be focused on by a company wanting to start introducing shared code.

Using the requirements a number of potential solutions was found and discussed. These solutions varied in size and nature, with some being smaller workflow changes and others being large tools. The one apparent thing is that there is no such thing as a one size fits all solution, instead the solution lies in the combination of multiple smaller solutions that together fulfils the elicited requirements. In light of this a matrix with the solutions categorised after their nature and timing, during the development, was created. This proved to be an efficient tool to get a quick sense of what solution could be used given a certain situation or scenario.

Based on interviews with employees at both the case company and Praqma, two scenarios describing the case company's needs and problems, regarding shared code, was elicited. These served as a basis for a recommendation on how the problems could be solved. This proved to be a difficult task partly due to the fact that some of these problems are problems that are not covered by this thesis and are therefore left as identified problems that the case company might benefit from looking at themselves. Another difficulty with the recommendation is that it has become clear that a solution that works now will not necessarily work in a few years (or less) time. This prompted the recommendation to try and look a bit forward and take into consideration the switching of solution when that time comes.

Bibliography

[1] W. A. Babich, *Software configuration management: coordination for team productivity*. Addison-Wesley Longman Publishing Co., Inc., 1986.

[2] K. Beck, *Extreme Programming Explained: Embrace Change*, 1st ed. Addison-Wesley Professional, 1999, ISBN: 9780201616415.

[3] (2018). Google scholar, Google, [Online]. Available: https://scholar.google.se/ (visited on 13/07/2018).

[4] (2018). Lubsearch, Lund University, [Online]. Available: http://lubsearch.lub.lu.se (visited on 13/07/2018).

[5] J. Preece and Y. Rogers, *Interaktionsdesign: bortom människa–dator-interaktion*, edition 1:1. Lund: Studentlitteratur, 2016.

[6] H. P. Jepsen and D. Beuche, 'Running a software product line: Standing still is going backwards', in *Proceedings of the 13th International Software Product Line Conference*, Carnegie Mellon University, 2009, pp. 101–110.

[7] H. P. Jepsen, J. G. Dall and D. Beuche, 'Minimally invasive migration to software product lines', in *11th International Software Product Line Conference (SPLC 2007)*, Sep. 2007, pp. 203–211. DOI: 10.1109/SPLINE.2007.30.

[8] D. Westheide. (14th Nov. 2016). The perils of shared code, INNOQ, [Online]. Available: https://www.innoq.com/en/blog/the-perils-of-shared-code/ (visited on 27/06/2018).

[9] A. Koratkar, S. Grosvenor, J. E. Jones, C. Li, J. Mackey, K. Neher and K. R. Wolf, 'Code sharing and collaboration: Experiences from the scientist's expert assistant project and their relevance to the virtual observatory', in *Astronomical Data Analysis*, International Society for Optics and Photonics, vol. 4477, 2001, pp. 208–216.

[10] J. Whelpley. (15th Aug. 2017). The problem with shared code, [Online]. Available: https://medium.com/@jeffwhelpley/the-problem-with-shared-code-124a20fc3d3b (visited on 11/07/2018).

[11] I. Sommerville, *Software engineering*. Addison-Wesley, 2011, ch. 16, ISBN: 978-0-13-703515-1.

[12] F. P. Brooks, *The Mythical Man-Month*. Addison-Wesley, 1995, ISBN: 9780201835953.

[13] P. H. Feiler, *Configuration management models in commercial environments*. Carnegie Mellon University, Software Engineering Institute Pittsburgh, PA, 1991.

[14] R. C. Martin, *Agile Software Development - Principles, Patterns, and Practices*, 1st ed. Pearson Education Limited, 2014, ISBN: 9781292025940.

[15] B. A. White, *Software configuration management strategies and Rational ClearCase: a practical introduction*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[16] J. L. Martin Fowler. (2014). Microservices, A definition of this new architectural term, MartinFowler.com, [Online]. Available: https://martinfowler.com/articles/microservices.html (visited on 19/10/2018).

[17] A. Sarma and A. V. D. Hoek, 'Towards awareness in the large', in *2006 IEEE International Conference on Global Software Engineering (ICGSE'06)*, Oct. 2006, pp. 127–131. DOI: 10.1109/ICGSE.2006.261225.

[18] A. Sarma, D. F. Redmiles and A. van der Hoek, 'Palantir: Early detection of development conflicts arising from parallel code changes', *IEEE Transactions on Software Engineering*, vol. 38, no. 4, pp. 889–908, Jul. 2012, ISSN: 0098-5589. DOI: 10.1109/TSE.2011.64.

[19] M. E. Conway, 'How do committees invent', *Datamation*, vol. 14, no. 4, pp. 28–31, 1968.

[20] (1st Mar. 2018). Why do we need configurable bom or 150 % bom?, [Online]. Available: http://www.practiceplm.com/configurable-bom-or-variant-bom/ (visited on 23/10/2018).

[21] F. Erens, H. Hegge, E. Van Veen and J. C. Wortmann, 'Generative bills-of-material: An overview.', *Integration in Production Management Systems*, vol. 7, 1992.

[22] M. T. Rahman, L.-P. Querel, P. C. Rigby and B. Adams, 'Feature toggles: Practitioner practices and a case study', in *Proceedings of the 13th International Conference on Mining Software Repositories*, ACM, 2016, pp. 201–211.

[23] (2018). Git, The Git Project, [Online]. Available: https://git-scm.com/ (visited on 22/10/2018).

[24] P. Seibel. (19th Nov. 2017). Repo style wars: Mono vs multi, [Online]. Available: http://www.gigamonkeys.com/mono-vs-multi/ (visited on 22/10/2018).

[25] C. Jaspan, M. Jorde, A. Knight, C. Sadowski, E. K. Smith, C. Winter and E. Murphy-Hill, 'Advantages and disadvantages of a monolithic repository: A case study at google', in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, ACM, 2018, pp. 225–234.

[26] S. Newman, *building microservices*, 1st ed. O'Reilly Media Inc, 2015, ISBN: 9781491950357.

[27] A. Noonan. (10th Jul. 2018). Goodbye microservices: From 100s of problem children to 1 superstar, [Online]. Available: https://segment.com/blog/goodbye-microservices/ (visited on 22/10/2018).

[28] (2018). Ibm rational clearcase, IBM, [Online]. Available: https://www.ibm.com/us-en/marketplace/rational-clearcase (visited on 23/10/2018).

[29] N. Paolucci. (Jan. 2017). Git subtree: The alternative to git submodule, Atlassian, [Online]. Available: https://www.atlassian.com/blog/git/alternatives-to-git-submodule-git-subtree (visited on 23/10/2018).

[30] (2018). Bit, Cocycles, [Online]. Available: https://bitsrc.io/ (visited on 23/10/2018).

[31] (2018). Gitsubmodules - mounting one repository inside another, The Git Project, [Online]. Available: https://git-scm.com/docs/gitsubmodules (visited on 22/10/2018).

[32] (2018). Repo - the multiple git repository tool, Google, [Online]. Available: https://gerrit.googlesource.com/git-repo/ (visited on 22/10/2018).

[33] M. Larsson and I. Crnkovic, 'New challenges for configuration management', in *International Symposium on Software Configuration Management*, Springer, 1999, pp. 232–243.

[34] D. Richter, M. Konrad, K. Utecht and A. Polze, 'Highly-available applications on unreliable infrastructure: Microservice architectures in practice', in *Software Quality, Reliability and Security Companion (QRS-C), 2017 IEEE International Conference on*, IEEE, 2017, pp. 130–137.

Appendices

Appendix A

Survey statements

In this appendix the statements the respondents to the survey was asked if they agreed with are listed. The graphs corresponding to the statements can be found in appendix B.

S1: Bugs becomes easier to fix, since the fix only needs to be applied to one place.

S2: Using shared code increases the speed of development.

S3: Shared code leads to more efficient variant handling.

S4: Using shared code reduces duplicated code.

S5: Shared code gives great returns in the long run.

S6: Shared code gives a high momentum early on in the development.

S7: Components can be shared at low cost, once developed.

S8: Shared code gives improved standards and clearer code.

S9: Specialists can encapsulate their knowledge in modules that can then be efficiently used by others.

S10: When using shared code, there is a need for more comprehensive coordination.

S11: Sharing code requires a lot of initial work.

S12: A slow change process is a big problem when using shared code (Here we mean changes as in changes to a shared library's features for instance. A slow change process such as a CCB or other could have an impact on your development.).

S13: The importance of traceability and history increases when using shared code.

S14: Using shared code, maintenance becomes harder since the impact a patch will have has to be tested more thoroughly.

S15: The build process becomes harder when using shared code.

S16: Bugs have bigger consequences when using shared code.

S17: Dependency management becomes more important when using shared code.

S18: Version control becomes harder when using shared code.

S19: Prioritisation becomes more important when using shared code.

S20: Shared code requires extended tool support.

S21: Shared code requires a higher degree of discoverability.

S22: When using shared code there is a too high focus on the shared code.

Appendix B

Survey data

In this appendix the graphs showing the response distribution of all the statements presented in the survey are shown. The statement corresponding to the graph can be found in appendix A.

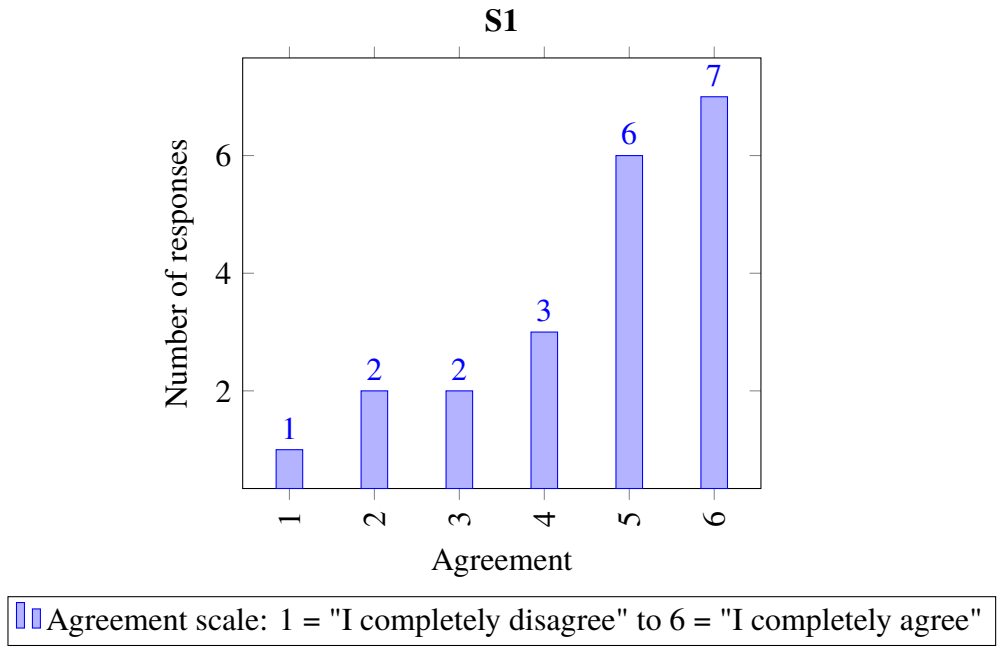

Figure B.1: Graph showing the response distribution on statement S1.

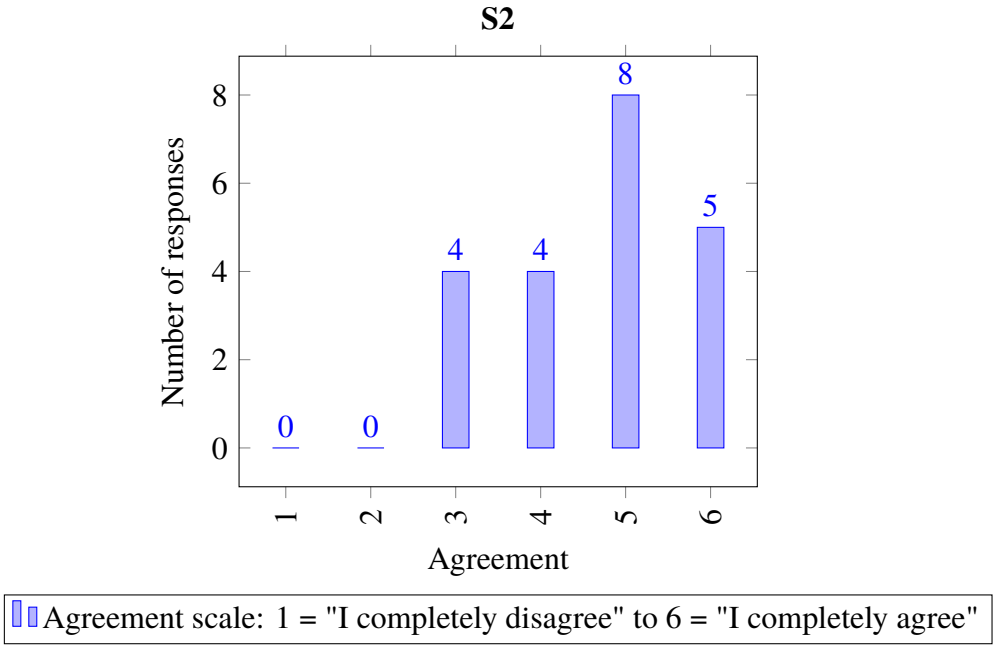

Figure B.2: Graph showing the response distribution on statement S2.

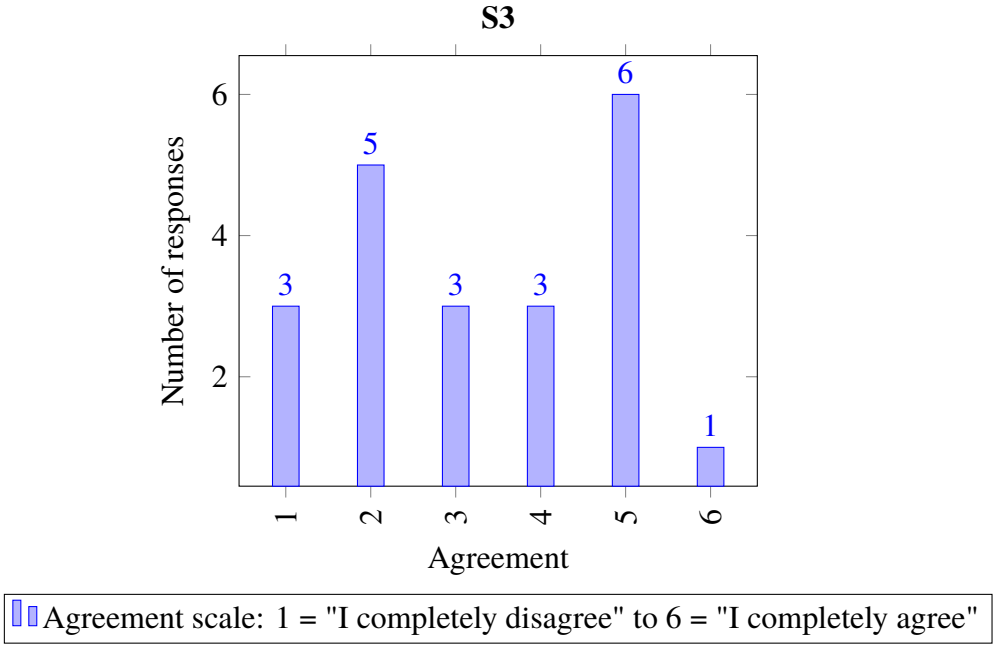

Figure B.3: Graph showing the response distribution on statement S3.

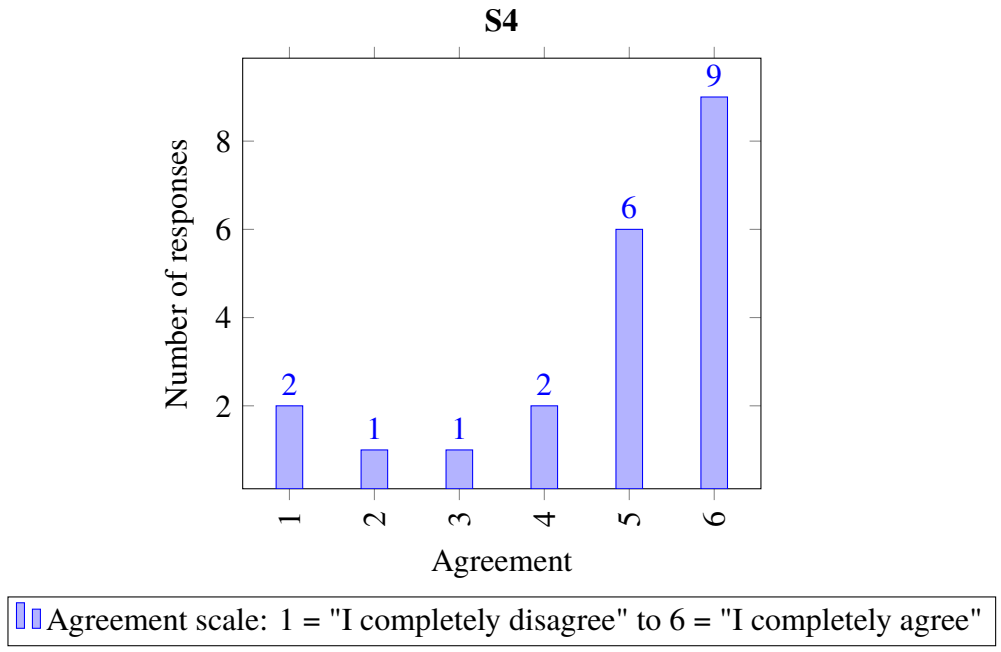

Figure B.4: Graph showing the response distribution on statement S4.

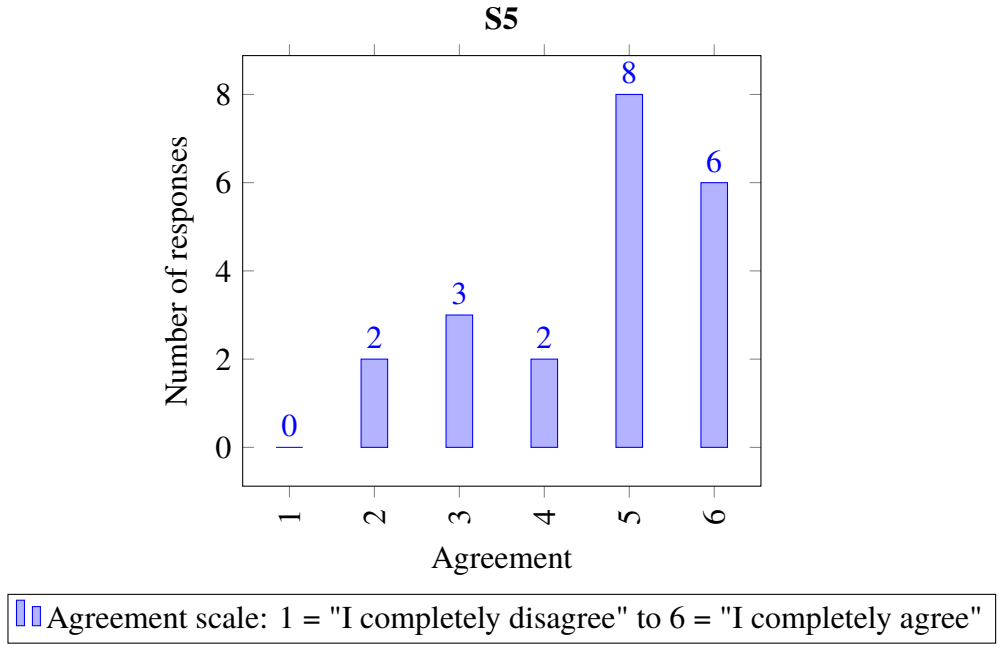

Figure B.5: Graph showing the response distribution on statement S5.

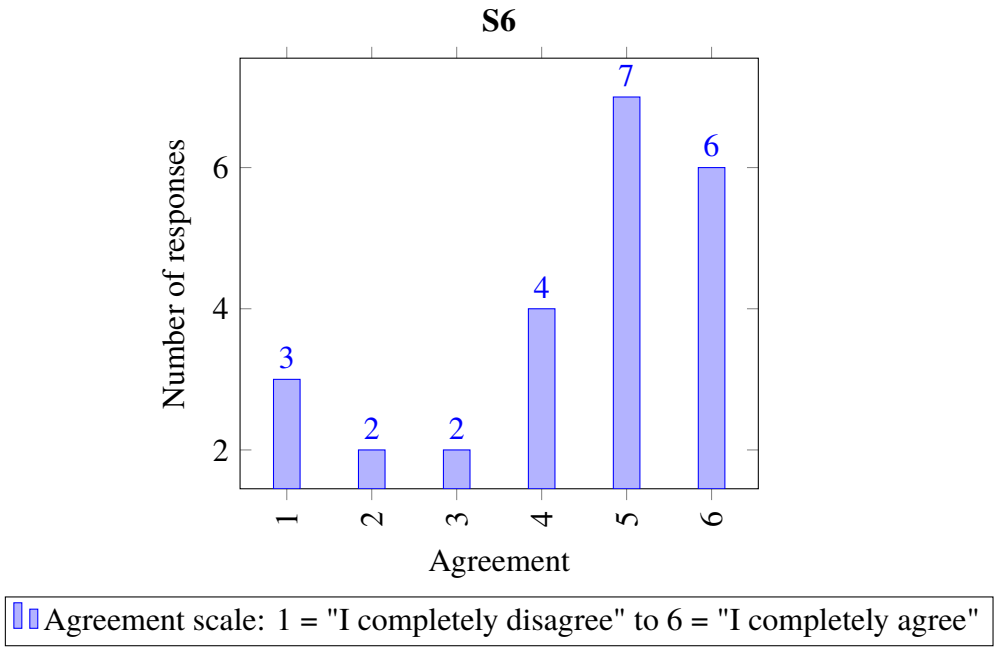

Figure B.6: Graph showing the response distribution on statement S6.

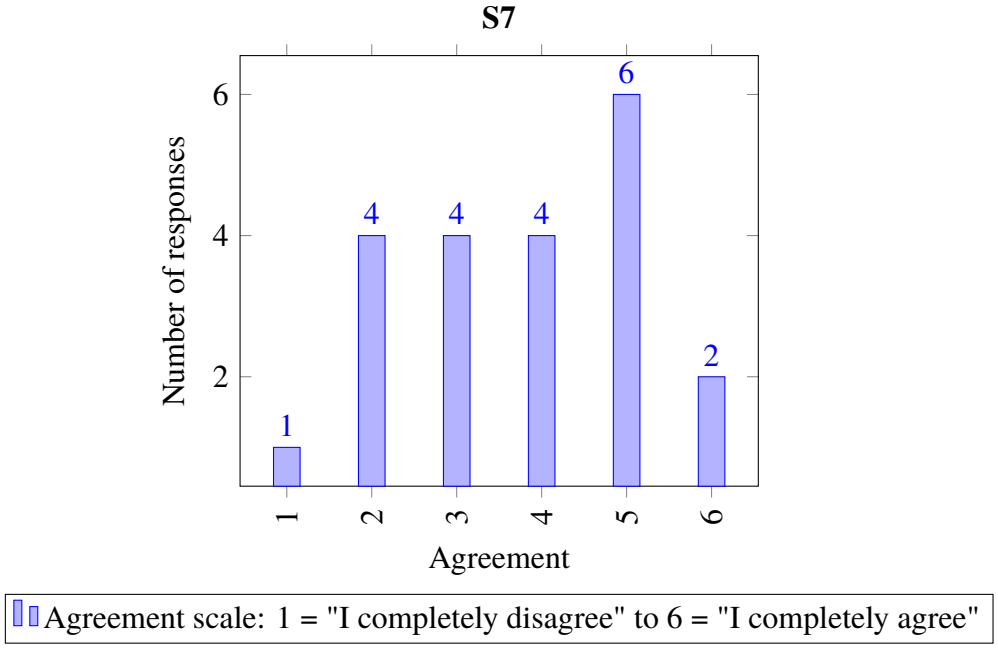

Figure B.7: Graph showing the response distribution on statement S7.

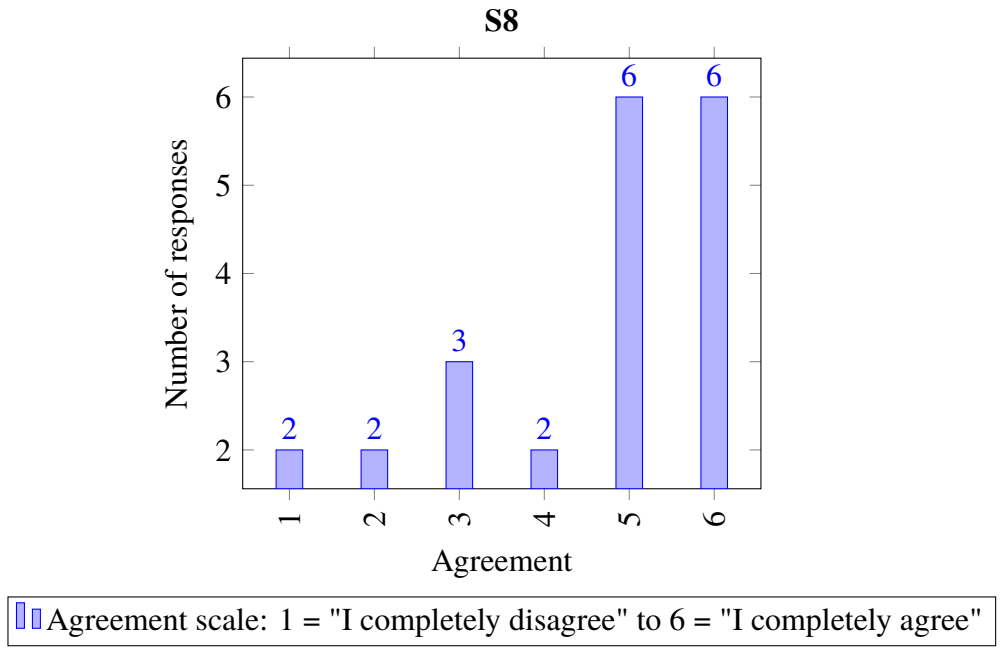

Figure B.8: Graph showing the response distribution on statement S8.

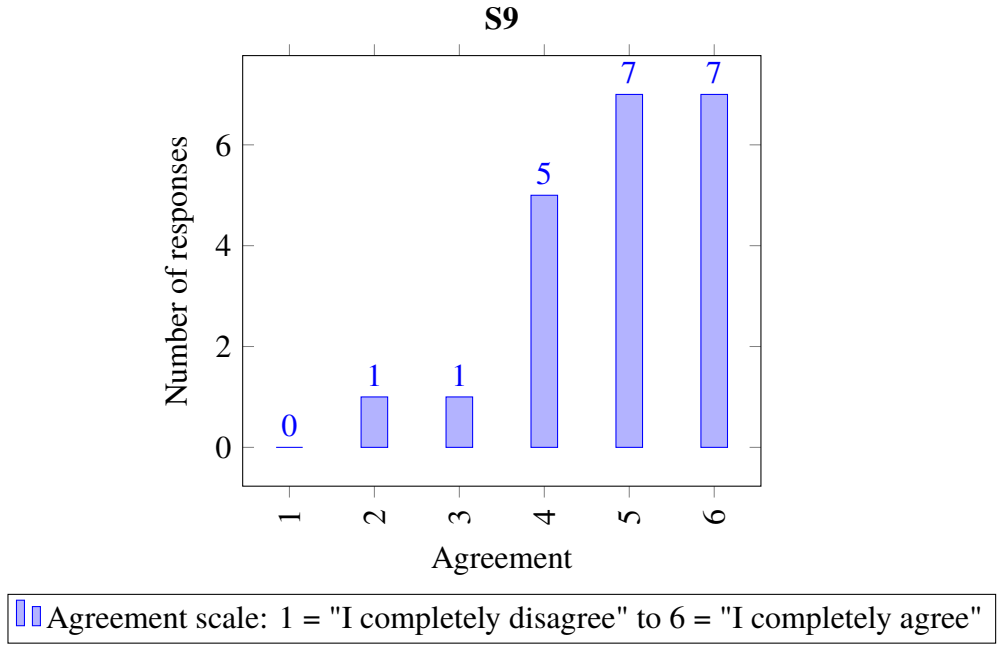

Figure B.9: Graph showing the response distribution on statement S9.

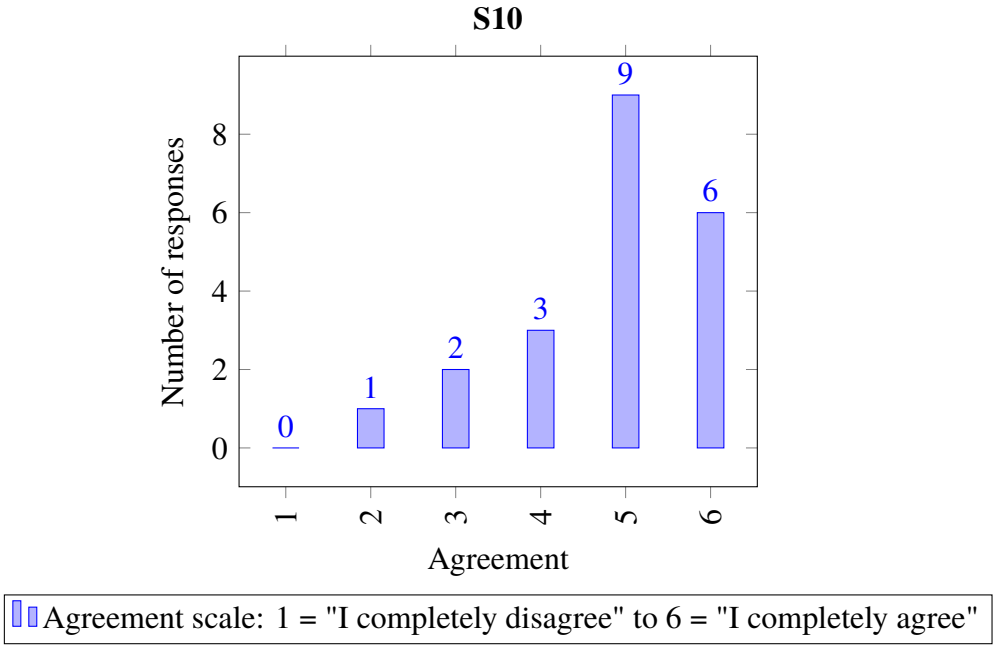

Figure B.10: Graph showing the response distribution on statement S10.

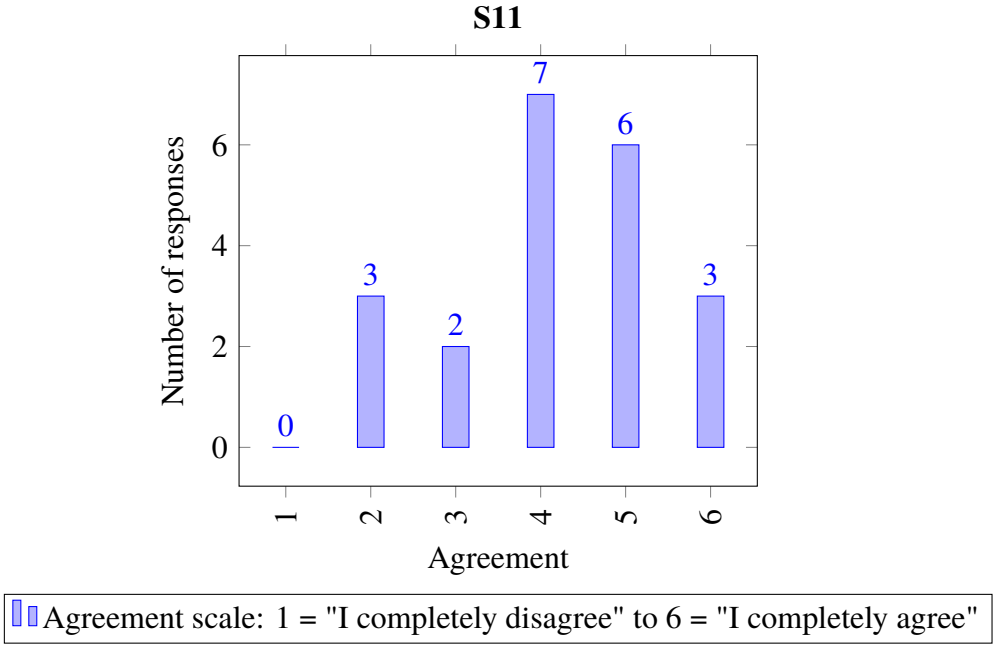

Figure B.11: Graph showing the response distribution on statement S11.

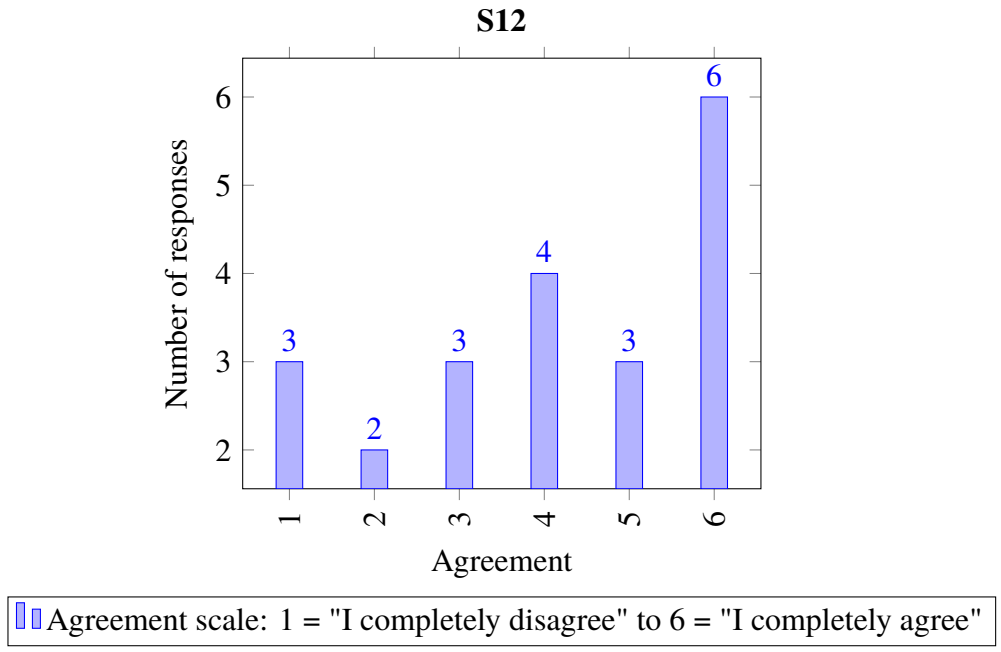

Figure B.12: Graph showing the response distribution on statement S12.

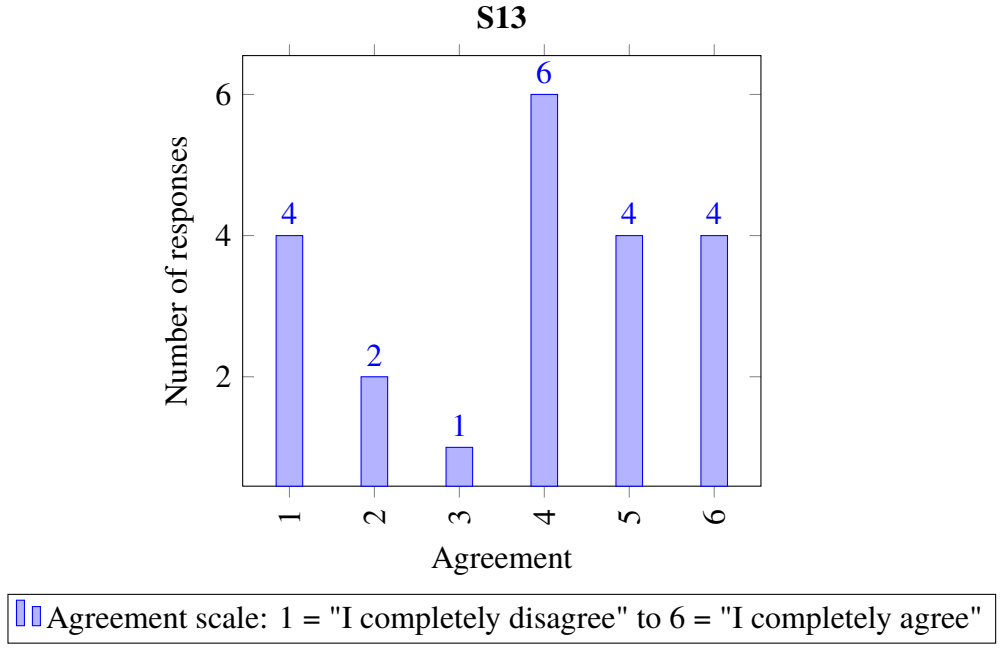

Figure B.13: Graph showing the response distribution on statement S13.

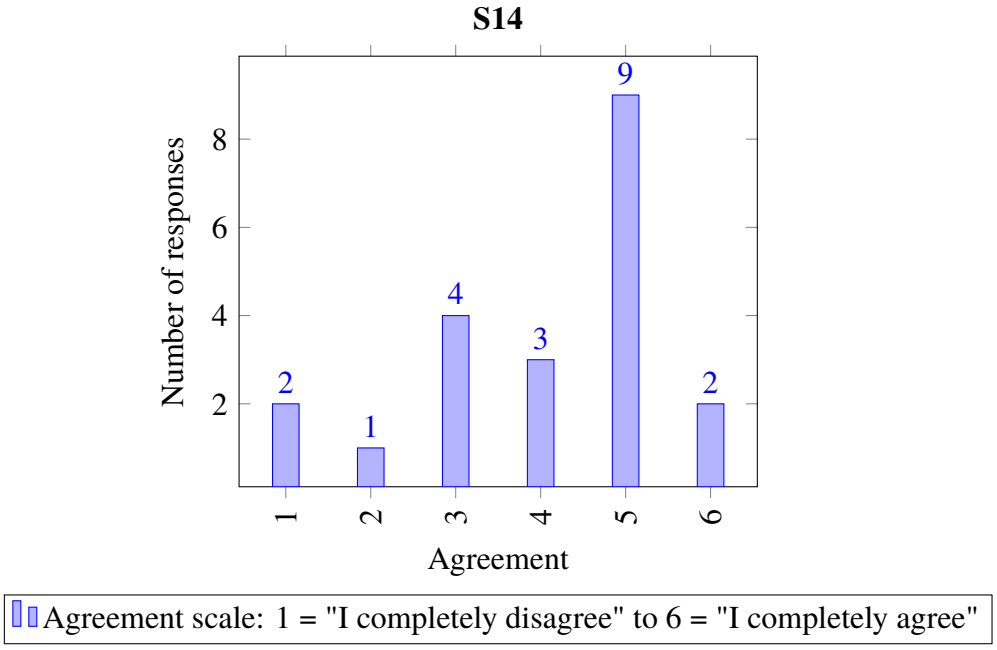

Figure B.14: Graph showing the response distribution on statement S14.

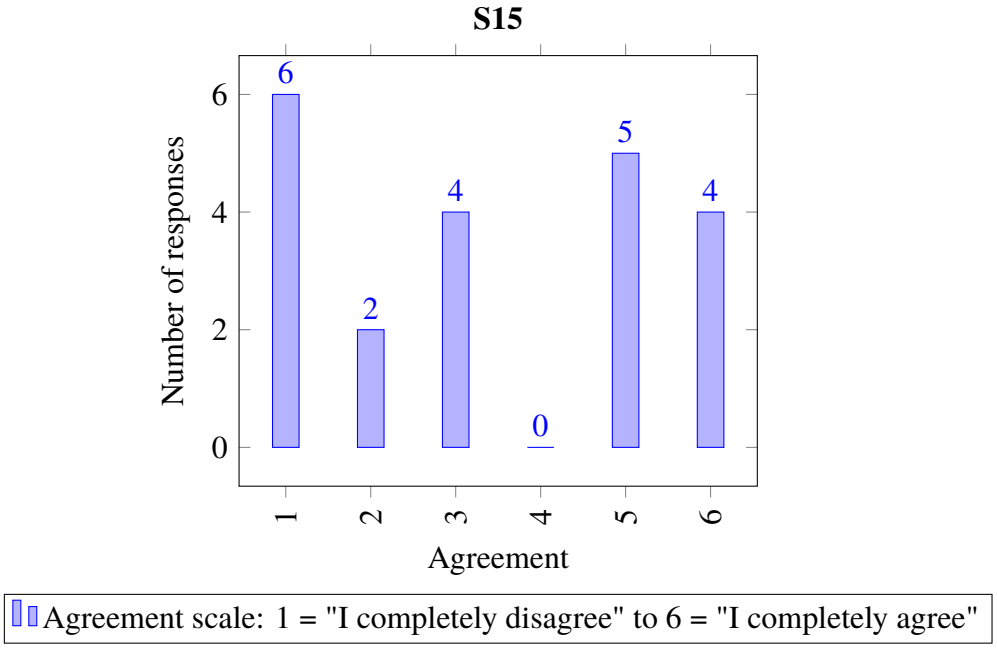

Figure B.15: Graph showing the response distribution on statement S15.

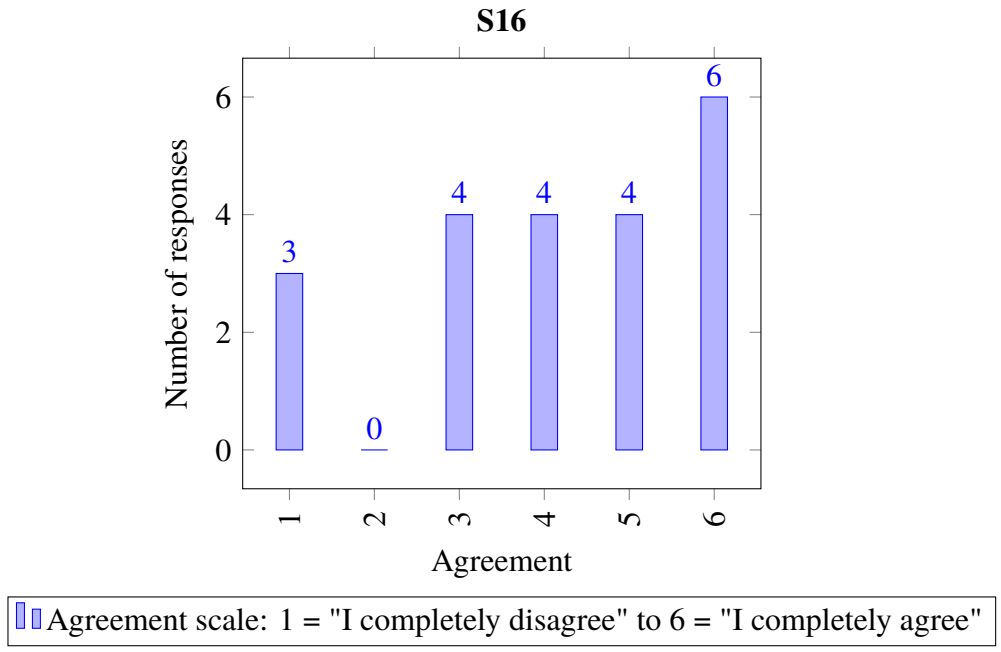

Figure B.16: Graph showing the response distribution on statement S16.

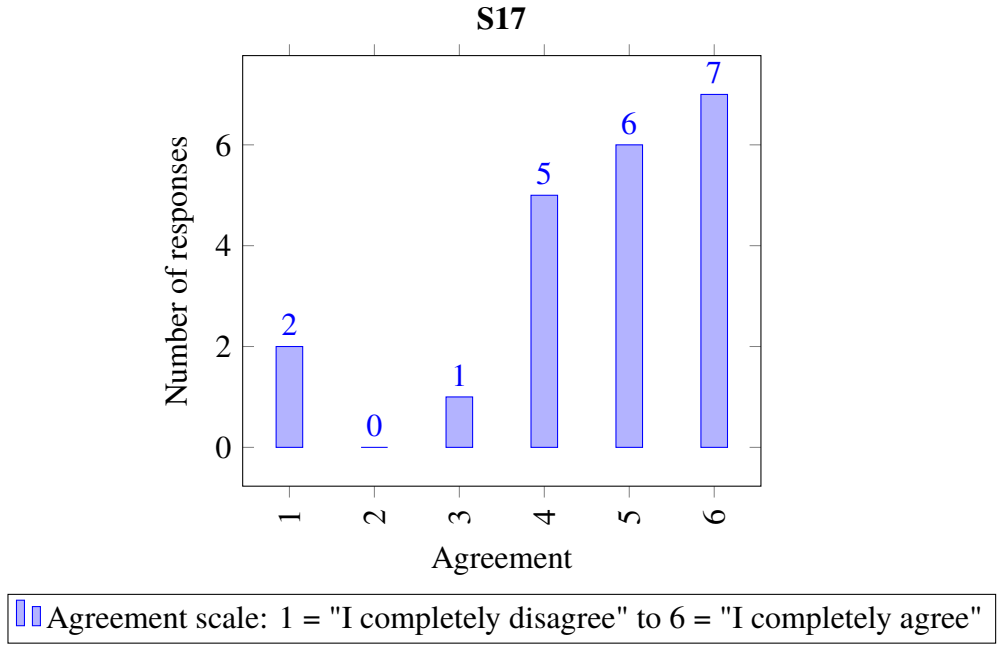

Figure B.17: Graph showing the response distribution on statement S17.

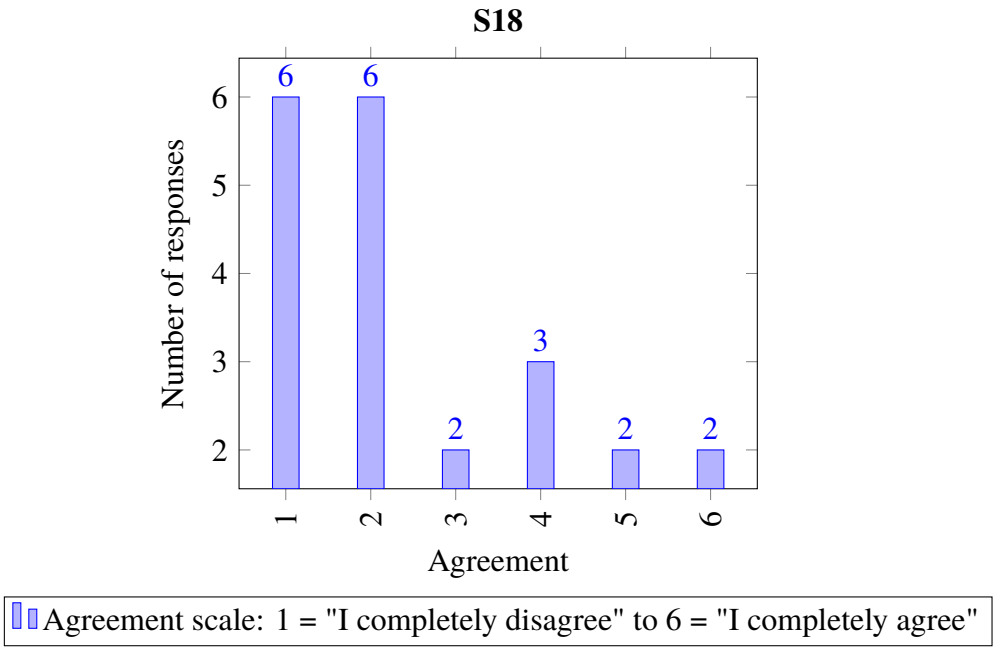

Figure B.18: Graph showing the response distribution on statement S18.

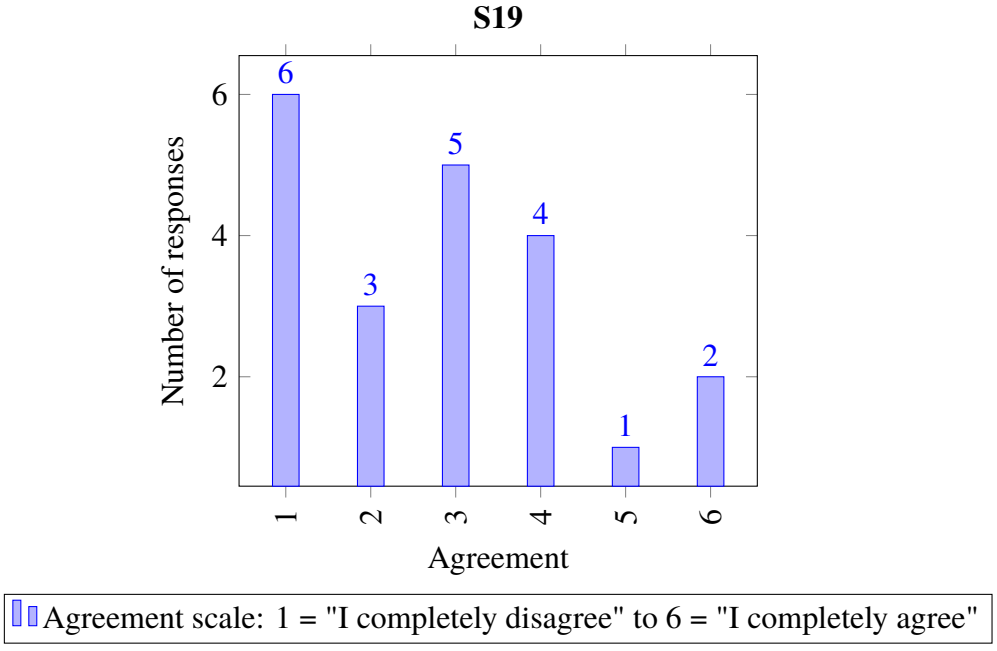

Figure B.19: Graph showing the response distribution on statement S19.

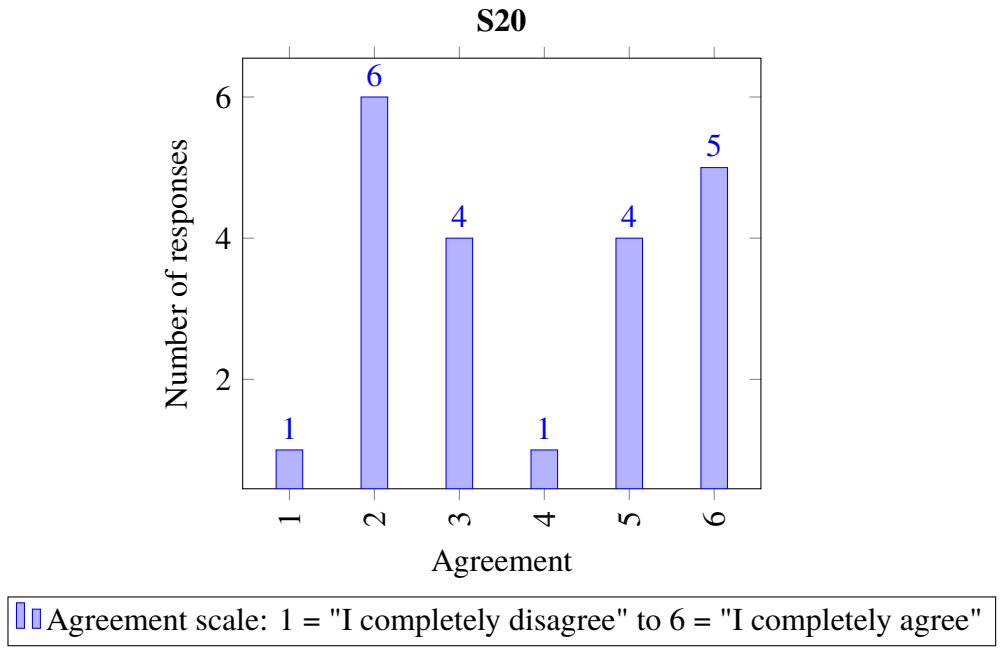

Figure B.20: Graph showing the response distribution on statement S20.

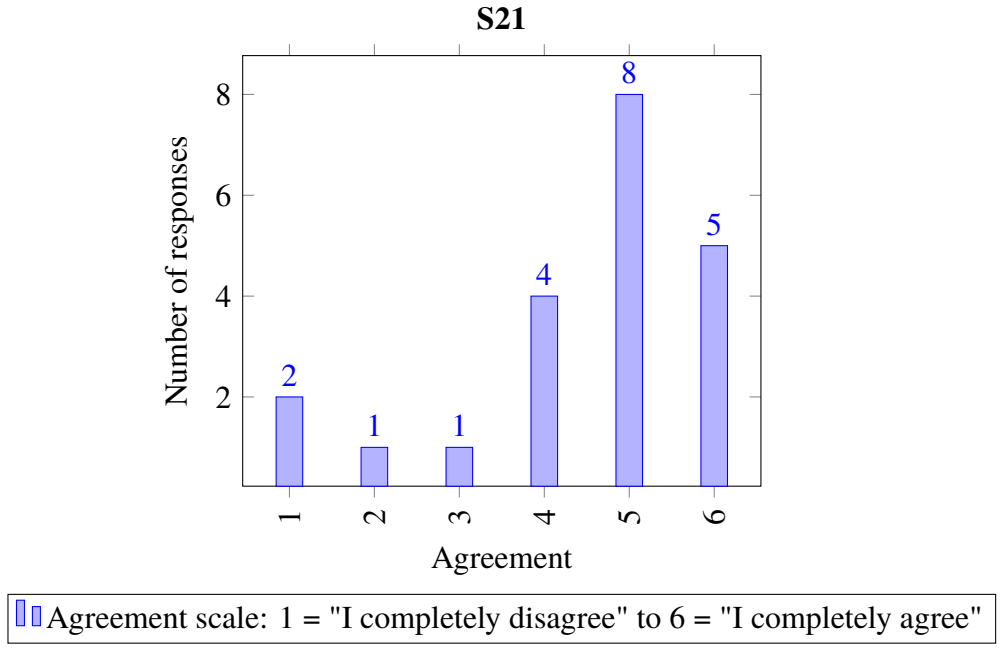

Figure B.21: Graph showing the response distribution on statement S21.

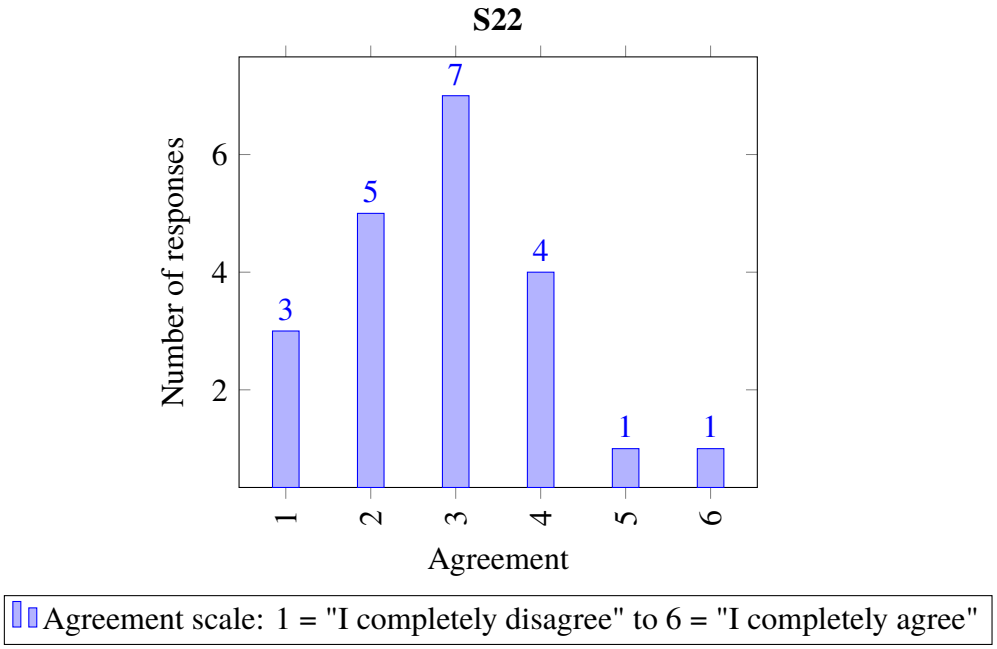

Figure B.22: Graph showing the response distribution on statement S22.

Appendix C

Scenarios used in the tool study

In this appendix the scenarios used in the tool study is listed. The scenarios are using a group of fictive developers A, B and C.

1. A wants to use code written by B in another project.
2. A wants to use a library from another project.
3. A modifies code covering their own project and another external project. The changes should be pushed to both projects.
4. A want to check if all files are up to date or if there exists changes. A has files from several projects and wants to be able be in any folder doing the check.
5. A wants to see if a change has been made and if so see whats have been changed. Independent of active folder.
6. B wants to add some files and remove some others from his project and another external project.
7. B wants to view the change history for some files from his own and another external project independently from active folder.
8. A wants to get the latest changes from specific projects.
9. B wants to get specific versions from particular projects.
10. A and B wants to do parallel development with fetching, modifying of code and thereafter merging of changes.
11. A and B are working on different projects and wants to get a common shared resource maintained by C.

12. A wants to create a release.
13. A is maintaining an old release and at the same time development to a new release is conducted. Some functionality is backported to the earlier release.
14. A do not want to have the shared code anymore and wants to remove it from his workspace.
15. B wants to check if there exists some function from another project that can be useful in his own project.

MASTER'S THESIS Investigating the impact of code sharing and how to manage it
STUDENTS André Alm & Daniel Dornlöv
SUPERVISORS Lars Bendix (LTH) & Albert Rigo (Praqma)
EXAMINER Ulf Asklund (LTH)

How to organize and manage shared code

POPULAR SCIENCE SUMMARY **André Alm & Daniel Dornlöv**

In today's development climate it is becoming more and more popular to share code in order to save on time, rework and costs. This work looks at the motivations and problems that exists for companies in a shared code context, as a part of this a case study at a company is made to get an industry perspective and to give recommendations on how they can work with shared code.

Many companies working with software development today would like to achieve a higher speed of the development process and avoid having to write the same code again. To try and keep the costs at a reasonable level, a rising trend is to share code between projects, teams or developers. The shared code might be source code, binaries, or libraries.

This thesis focuses on what drives a company to start using shared code and also on what can be done, both to try to solve the problems caused by code sharing, but also to try and see how shared code can be supported so that it becomes as effective and efficient as possible. To get an understanding of what different drivers and problems there are with code sharing interviews were conducted with both developers and persons working in a more overarching role between projects and teams. In addition to the interviews a literature study was conducted and the findings from both of these were weighed together to form a requirements specification on what is needed to share code and avoid the problems identified during the interviews and literature study.

The requirements were used as a guide to find solutions to the problems with shared code. It became apparent that there is no silver bullet that can solve all of the problems, instead the trick is in combining solutions to form a package of tools, principles and workflows that together mitigate or lessens the problems. This approach led to a matrix in which it is possible to find a solution based on at what phase in the development a problem occurs and on what type of solution it is. This matrix was then used to make a case study at a company and give recommendations on how they could work with shared code.

For the case study company, two different recommendations were made because mainly two different user scenarios were identified, and these require different measures. The first perspective dealt with sharing code between product teams involving many people and a wide spectrum of products while the second is related to sharing on a smaller scale between a few developers, often in experimental or exploratory purposes. A difficulty with the recommendations is that it become clear that a solution that works now will not necessarily work in a few years (or less) time. This prompted the recommendations to try and look a bit forward and take into consideration the switching of solution when that time comes.