MASTER'S THESIS | LUND UNIVERSITY 2014

Filling In Safety Impact Analysis Forms - Quality, Efficiency and Soft Issues

Johan Thiborg-Ericson

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2014-08

Filling In Safety Impact Analysis Forms

(Quality, Efficiency and Soft Issues)

Johan Thiborg-Ericson
`dt07jt4@student.lth.se`

March 14, 2014

Master's thesis work carried out at Anonymous Company.

Supervisor: Anonymous Supervisor

Examiner: Lars Bendix, bendix@cs.lth.se

Abstract

A company wanted a tool to improve the general correctness, quality, efficiency and handling of a form. The form was filled in by developers to document their work with impact analysis during corrective maintenance of the software. There were answers to some of the questions in the form that were missing, inconsistent or contained irony. The most severe problems were caused by bad layout of the questions, disagreement about when to add new regression tests and the fact that developers copied old filled in forms to fill in new forms faster. The biggest efficiency problems were answering the question which got ironic answers (How could this problem been avoided?) and that impact analysis was hard in the old parts of the system. Some developers did not seem comfortable talking about what they thought of the form, for some reason. Four handling problems are identified. The developers copied old forms, implemented bug fixes before the CCB had allowed them to do so and worked with many forms simultaneously. Also, there was long time between when the forms were filled in and when they were reviewed.

The recommended solutions are that the company incorporates a policy on how safety precautions and efficiency of the process should be balanced in the employees' daily work. Meanwhile, they should assign representatives for the developers to whom the other developers might express their thoughts anonymously. The representatives should forward these thoughts to the ones responsible for developing the process at regular scheduled meetings. The question with ironic answers should not be mandatory or be removed since no root cause analysis model suggests collecting proposed process improvements in this way. The time between writing and reviewing the form should be shorter. A new state should be introduced for the forms, to distinguish the forms that are ready to be put in the report to the third part certification organ from those which aren't. It is not recommended to make a solution based on a tool since it could only solve one of the severe problems, and that problem will solve itself eventually anyway.

Keywords: Forms, Quality of Data, Process Improvement, Corrective Maintenance, Impact Analysis.

Acknowledgements

There are many people who can take credit for different aspects of this thesis. My first thank goes to Artour Klevin who never grew tired of providing complementing perspectives. My supervisor has patiently tried to explain the ways of the company to two idealistic students. Lars Bendix has been a great support who generously shared his time and experience and who always believed in me and my work, even when I didn't. I would like to thank Carin Ericson for her tips on academic structure and Leif Thiborg for constantly reminding me of simplicity and pushing me to continue working. Alexandra Larsson has helped me improving the language in parts of the text. I would like to thank Linus Ahlberg for narrow reading, Krystian Hoffman for initiated comments, Martin Johansson for bringing new perspectives and Johannes Persson for overview. My last and greatest thanks go to the employees of the company who showed great patience with my slow understanding. All errors in the text are of course mine.

Contents

1 Introduction **7**

2 Background **11**
- 2.1 Properties and Stakeholders 11
- 2.2 Impact Analysis 12
- 2.3 The Life Cycle of the Answers 13
- 2.4 The Questions of the Form 15
- 2.5 The Uses of the Form 16
- 2.6 Previous Work 17

3 Problem Analysis **21**
- 3.1 Quality and Correctness 21
  - 3.1.1 Question 1 Has Too Little Argumentation 21
  - 3.1.2 Question 4 Contains Nicknames for Files 22
  - 3.1.3 Question 4 and 8 Are Inconsistent 22
  - 3.1.4 Question 5a Is Not Answered 23
  - 3.1.5 Question 12 Contains Irony 23
  - 3.1.6 Current Approaches to Improve the Quality of the Answers 24
- 3.2 Efficiency 24
  - 3.2.1 Finding Documents 24
  - 3.2.2 Everyone Uses the Same Form 24
  - 3.2.3 The Time It Takes To Fill In the Form 25
  - 3.2.4 Increased Demands on the Contents of the Form 25
  - 3.2.5 The Influence of Academic Education 25
- 3.3 Handling 26
  - 3.3.1 The Time of Implementation 26
  - 3.3.2 The Developers Copy Old Forms 27
  - 3.3.3 Quality Assessment Times 27
  - 3.3.4 Efficiency in Different Stages 28
- 3.4 Traceability 30

3.4.1 The Historical Architecture of the System 30
3.4.2 Requirement Documents Become Mandatory 30
3.4.3 Moving to a New Tool 31
3.5 Soft Issues 31
3.5.1 A Global Process 31
3.5.2 Removing Questions 32
3.5.3 Influence over the Questions of the Form 32
3.5.4 Expressing Opinions about the Form 32
3.6 The Results of This Chapter 33

4 Possible Solutions 35
4.1 Agile Development 35
4.2 Speculative Bug-Fixing 37
4.2.1 Adaptive and Corrective Maintenance 37
4.2.2 CCB in Corrective Maintenance 38
4.3 Removing Question 12 40
4.4 Communication 42
4.5 A Fully Done Stage 44
4.6 Software Tool Support 44
4.7 Cheap and Almost Free 45

5 Discussion 47
5.1 Future Work 49

6 Conclusions 51

Index 53

Bibliography 55

A The Form 59

Chapter 1
Introduction

There are many situations where forms may be used in software engineering. [18]Some work has been done to investigate how they are used, e.g. [4]. The author hasn't found any description of forms used for impact analysis. This thesis is a study of a company that uses a form to document impact analysis.

The company investigated in this thesis makes *products* that are supposed to work at high rates, continuously for many years. The products are used in very large factories and often handling dangerous raw material which may cause a major environmental disaster or harm the operators if something goes wrong. It is important that the software detects if something goes wrong so it can shut itself down, before the problem propagates to the hardware. The most *safety critical* parts in the software are the parts detecting problems and shutting the system down safely.

If a bug is found in such software it is important that it is removed as soon as possible. Sometimes however, it happens that the developer who fixes the bug makes a mistake and introduces some other error in the code. The most important task when fixing the bug is ensuring that no new bug has mistakenly been introduced in the safety critical parts of the software. Doing this is called *software change safety impact analysis*, that is, analyze how a change of the software (e.g. fixing a bug) impacts the rest of the product. Note that in safety impact analysis, the main focus is safety. This makes it different from regular *impact analysis*, which is described in Section 2.2.

The company has a defined *process* to ensure that everyone involved in making the product does his very best to make the product safe. One important part of this process is that everyone writes down what they have done so that their work may be reviewed in the future. The company uses a *form* to document the work done during impact analysis. The form is the central subject in this thesis and is included in Appendix A. The answers in the form are assessed by a third party certification organ (among other things, see Section 2.5). Therefore, it is important that the answers are readable and understandable to others than the employees of the company.

The *initial purpose* of this work, according to the company, was to develop a software tool that

- made the forms to require less work,
- helped them to increase the quality of the answers,
- ensure that the forms were handled correctly and
- ensure that the answers contained the correct information.

The subjects Quality of the answers and Correctness of the answers will be handled as one single subject since they are related.

To better understand why the company needed a software tool, the perceived problems were investigated with one *investigation line* for each one of the three subjects: quality, efficiency and handling. The first line is finding questions in the form that sometimes have incorrect or low quality answers. This includes finding out why the answers contain erroneous information or otherwise have low quality. The second line is trying to understand what takes time when filling in the form (efficiency). This line also includes other thoughts that the developers had about the form and how they felt talking about it. The third line is finding if the way the form is currently handled negatively affects how long it takes to fill in or the quality of the answers.

The main *research question* (formulated at the end of this paragraph) is constructed using six partial research questions.

- Which problems does the company perceive regarding the quality, correctness, efficiency and handling of the form?
- How severe are the problems, compared to each other?
- Why have the problems appeared?
- What positive effect will different solutions have on the perceived problems and their causes?
- Will the different solutions negatively influence the safety of the process?
- How much will the different solutions cost?

The fifth research question is added because it is mandatory for each change to the process in this company to preserve or increase the safety of the process. This gives a total of six different research questions, of which the first three investigates three different subjects: quality, efficiency and handling. The main *research question* is finding solutions with good effect (preferably on the problems' causes) and low cost for the severe problems.

The *target audience* and *purpose* of this thesis is threefold. The first is helping the company to improve the way they work. The second is helping other companies who want a tool to improve their efficiency, quality, correctness or handling of a form. The conclusion, in this case, is that a tool isn't a good solution. The third target audience is the computer science research community who may use the findings to better understand the use of forms in the software engineering process in general and corrective maintenance in particular. There is also a motivation of why certain parts of current impact analysis models are inefficient in the context of corrective maintenance.

Doing deeper research in all six partial research questions is way beyond the scope of a master thesis. The chosen *method* was interviews with few questions that could be interpreted in many ways. The plan was that if the interviewees were allowed to speak freely they would come to talk about the most severe problems and the most viable solutions. A more strict investigation method, e.g. questionnaires, would potentially miss important aspects since the respondents would be steered just to think about the questions in the questionnaire. Most interviews were recorded and transcribed, but some interviewees did not allow that. Then, the interviews were reconstructed from the notes and memory. Key-words were identified and gathered into groups that were later formed into the chapters of this text.

The chapters are ordered as follows. Chapter 2 contains the specifics of the company, the product, the process and the form, but also a summary of the related works in the area. Chapter 3 investigates the perceived problems regarding quality, efficiency and handling of with the form. It also tries to find the root causes of the problems. Chapter 4 presents solutions to some of the most severe problems. Chapter 5 will reflect over the quality of the different parts of the earlier chapters, how well the findings generalize to other companies and contexts. It also puts out a roadmap for future work in the area. Some room is given to the discussion about why the company had requested a tool. Chapter 6 summarizes the findings of the previous chapters. The chapter named “Index” contains a list of where different key-words are defined. Appendix A contains a version of the form where the company is made anonymous.

Chapter 2

Background

The software engineering industry produces a wide variety of products. The processes for developing the different products look very different, depending on the specific characteristics of the product produced. This thesis is about a software developing company that sells products that are used for very long periods without being updated and where safety is important. This makes the priorities different from the ones of a standard software development process. The first section explains how and why this process is different from the process used at other companies. The second section of this chapter describes the details of the process connected to the Safety Impact Analysis Form. The third section describes the actual content of the form, i.e. the questions. The fourth section describes earlier scientific findings about forms in the software engineering process.

2.1 Properties and Stakeholders

The product of this company has some extreme attributes (safety requirements, hard to test and low requirement volatility). Therefore the process has other focuses than what is usual. The quality of the process is a very important asset of the company. This section provides a description of the process and the stakeholders who are related to the form. The description is somewhat simplified and altered.

This thesis is done in a company that develops hardware and a software framework for different chemical processes, e.g. producing chemicals or cracking oil. The customers use the framework to program their hardware components. Usually a customer connects many hardware components with wires along the production line to form a system. In such a system, timing is very important. Often, if a computation takes longer than it should, the system will break. This might also happen if there is an erroneous computation. In the best case the system will go to a safe state and wait for an operator to restart it. Then the whole process will have to be restarted, which can take long time, and huge amounts of money will be lost since the machine can't produce anything during the stop. In the worst case, the erroneous computation won't be noticed. Then the software might send faulty signals to the hardware, which may physically harm it, or worse, the operators. In some places the operators might die if failures aren't handled correctly by the software. The ability to handle such situations correctly is called safety.

Besides being safe, the products of this company are supposed to work continuously for decades, without maintenance; they should have high reliability. This makes the requirements change slowly, since the customers think it is more important that the product works than that it has many features.

At this company, safety is implemented by *emergency code* and a *safe state*. The emergency code should be executed at the slightest indication of failure in the system. It should instruct all software and hardware to go to a safe state. The safe state may include stopping engines, closing valves and turning heaters off. The safety of the system depends on all erroneous computations being identified and that the emergency code is executed correctly. All the emergency code in the product is called safety critical. Ideally, the safety critical code should be completely free from bugs. Unfortunately, it is impossible to prove that a program is free from bugs. However, it is possible to estimate how often a failure will occur. Therefore, every code file has a safety integrity level, SIL, indicating how often it will break. Code with SIL-1 will have a probability of failure of 10^{-5} or less in one hour, while code with SIL-4 will only fail with the likelihood 10^{-8}. This gives an expected time between failures of over 10 000 years. The likelihood of failure in the whole system can then be computed from the likelihood of failure in its parts.

It is unfeasible to prove that code fulfills SIL-8. [14] However you can certainly show that you have tried your very best. Hamilton and Beeby argue that "quality has to be seen to be built into the whole development process". [16] Thomas finds that there is a general consensus in the software quality management literature that "[...] a quality process will lead to a quality product". The producing company has invested a lot in a high quality *software engineering process* to ensure the quality of the product.

Unfortunately, it isn't possible for the customers to directly assess the quality of the process since it is a business secret of the producing company. Therefore they hire an independent *third party certification organ* to do this. The certification organ has free access to all the documentation of the company and they regularly do inspections to verify that the process has a high standard and that the process is followed. There is a *process team* at the company who work in close collaboration with the certification organ to improve the process.

The product isn't only developed in Sweden, but also in India and Germany. The different locations where the product is developed will be referred to as sites. The process is required to be the same at all sites, so it is not allowed to make local changes to it.

2.2 Impact Analysis

Impact analysis is an important subject in this thesis. When a program needs to be modified, there are often other things that have to be modified as well. These additional modifications are called ripple effects, like the ripples around a stone dropped in water. As an example, if a feature is added, the corresponding requirement and user's manual have to be changed. The testing department will want to know what tests they will have to run. Finding the code and documents that are connected to a proposed change is called software change impact analysis. The company calls it safety impact analysis. The word "safety" indicates that its purpose is also to assess how the safety of the system is affected by the change.

Impact analysis is much easier to do if there is a traceability matrix. That is a formal documentation on how the different documents (from requirement and design to code and tests) depend on each other. You can also see how the different modules depend on each other. The traceability data is typically stored as fields in a database or in written documents. Still, there might be dependencies that are missed in the traceability matrix.

2.3 The Life Cycle of the Answers

The company uses a *safety impact analysis form* (the form) to report the findings of the safety impact analysis. The form can be found in Appendix A. All problems described in this thesis are somehow connected to that form. This subsection describes how and when the form is filled in, including the whole process around.

If the product stops working a failure report is written.[1] A *case* is opened to find the bug that caused the failure and remove it. A case is all the activities that have to be done before the bug is fixed and the bug fix can be put into the baseline of the product. The process of fixing the bug consists of three *stages*: debugging, implementation and testing. They are described below and in figure 2.1.

In the *debugging stage*, a developer who knows the code that has broken is assigned to find the error. To do this, it is necessary to set up a copy of the *environment* where the bug was found. This includes connecting hardware, installing virtual machines and such. In the debugging stage, the developer also designs (but not implement) a solution that has low impact on the safety of the system. The developer should make an impact analysis of this solution. To do this he uses the traceability matrix (see section 2.2) of the product. It is described in word documents.

There is a manual describing how the developer should fill in the impact analysis form. There is a template for the form in a read only, unformatted text file on the intranet of the company. This file contains the form, that is, a set of questions numbered 1 to 13. Below each question there is a text saying "A:" and that is where the answer to the question should be written. Some of the questions have sub-questions labeled "a", "b", "c" and so on. Each sub-question should be answered separately. When referring to a sub-question, say sub-question "b" under question 4, the format 4b will be used. The version of the form used during the writing of this thesis can be found in appendix A. An explanation of all the questions and their sub-questions can be found in subsection 2.4. When a developer should do safety impact analysis, the text of the latest version of the text file is copied from the intranet to the developers working computer. As the developer learns more about the case, he fills in an answer to each question or sub-question.

The impact analysis form is just one of the many things that the developer has to report in a case. The company has a tool used for creating and saving these reports. This is done using an online form that is posted to a database. The filled in form is a part of the report. It should be pasted in one large field in the online form. It should be noted that the whole

[1]Most failure reports come from the testing department and not from customers, since the testing department uses the product in unusual ways when they test things.

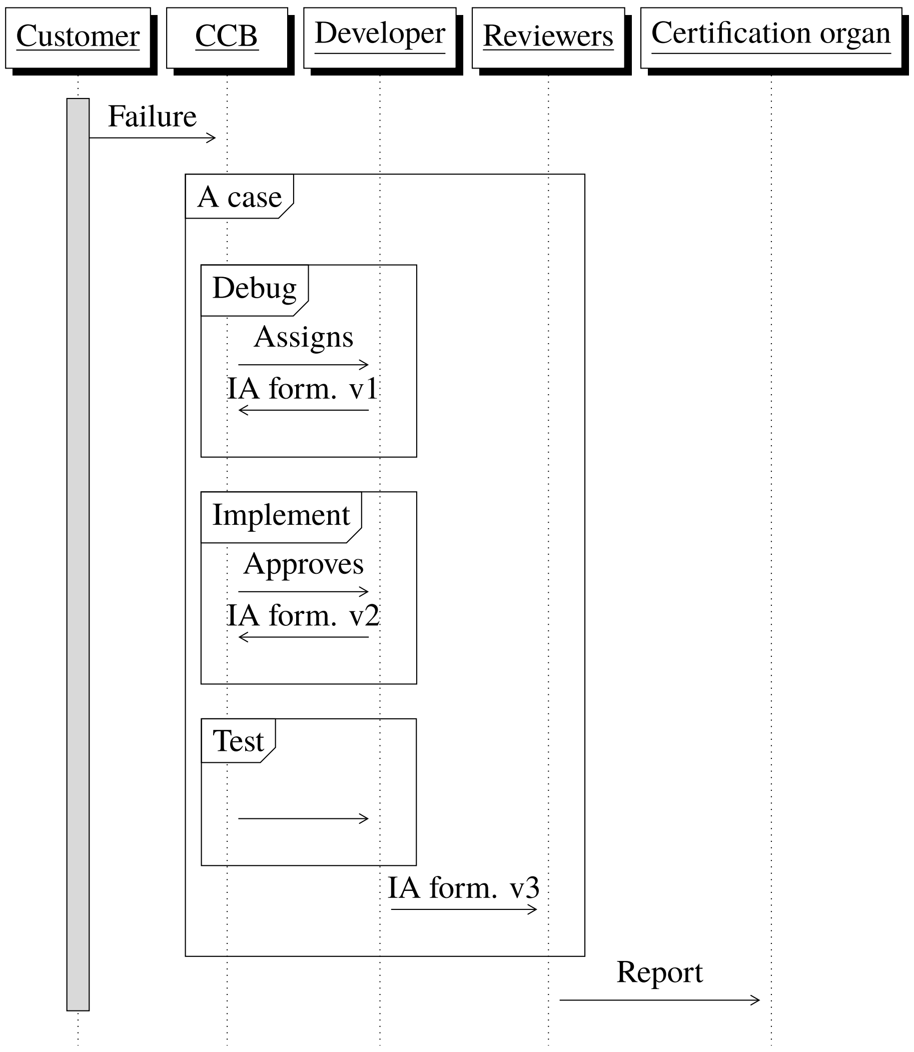

Figure 2.1: The life of a case. The top row shows the actor of the respective column. The case starts at the top row and steps one row at the time to the end of the case at the bottom row.

filled in form is pasted. It is not pasted answer by answer. Upon completion the online form is sent to the database.

The *CCB* (Change/Configuration Control Board) reads the answers in the form and decides if the solution should be implemented. When this thesis was written, the reviewers weren't part of the CCB. Then the developer gets the case back and writes the code in the implementation stage. He also writes tests that verify that his solution works. This requires that the environment from the debugging step has either been kept or it has to be rebuilt. If the developer discovers details of the change that he hadn't thought of earlier, he adds this information to the form. The online form in the reporting tool is filled in again. The CCB reads the new version of the form to decide if the change can be put into the baseline of the product. If it can, the developer makes a local branch of the baseline, merges the bug fix and runs the unit tests in the regression test suite for the affected modules to verify that no other functionality has been inadvertently broken. He also runs the newly written tests in the environment. This is called the (regression) *testing stage*. After this stage the case is handed over to the testing department for further testing.

At the end of the project, a team reviews all the forms, the review team. If a form isn't good enough, the case has to be reopened and the form has to be rewritten. Before the new version of the product is released, the approved forms are put together into a *report* (the safety impact analysis report) by the reviewers. The report is sent to the certification organ so that they can verify that all changes have been done in a safe way. If they aren't happy with something they send the report back for correction or clarification.

2.4 The Questions of the Form

This section gives a short explanation of the questions in the form. It's not strictly needed for understanding the thesis, but just for the curious, narrow reader.

Question 1 tries to establish if the problem at hand can make the system behave in an unsafe way. This is the main indication to how high priority the case should have.

Question 2 lets the developer describe the solution in a more holistic manner than in the rest of the form.

Question 3 is about software versions. When a company buys this kind of product they usually use them for a long time. There is little need to use new versions of the software if the old one works. Therefore, many versions of the software are in use simultaneously. Question 3 is used to verify that the bug is corrected in all the versions.

Question 4 has two main purposes. Firstly it is an indication to what regression tests to run. Secondly it is a fairly objective indication to how "good" the change is. A bug may often be fixed by many different bug fixes. It is preferable to find a bug fix where the impact analysis (see section 2.2) indicates as few safety critical files as possible. Sub-question 4a lists the files that have actually been modified. Sub-question 4b lists safety critical code that depends on the changes in some way.

Question 5 lists the different quality requirements for the product. They have been replaced by dots in appendix A to make the company anonymous. There are eight main requirements and one “other” category in question 5. The exact definition of how the questions should be written is not of interest to this thesis. Note that sub-question 5a is much longer than the others.

Question 6 settles if the change will be visible to the end user when he uses the libraries.

Question 7 is mostly about design documents.

Question 8 contains the unit tests, which are run by the developer. Question 8a should describe how the correction should be verified, that is, a test that fails before, but runs after the correction is implemented. This new test should be added to the test suite for the changed code and should be rerun in the future every time that code might be influenced. Therefore the test should be accompanied by a new test description file in question four, see section 3.1.3. Question 8b lists the test cases associated with the files listed question 4b.

Question 9 is similar to question 6, but is about the manuals rather than the libraries.

Question 10 gives the developer a chance to estimate how much the correction will cost, in terms of man hours. This is the main source for information of the cost of the change.

Question 11 and 12 is a guess of how the problem was caused, and a suggestion to how similar problems could be avoided in the future. The focus of the answer should be on in what phase (requirement, design or implementation) the problem was introduced.

Question 13 is a guiding note to the testing department so they can select what regression tests to run. The difference to question 8 is that the tests are divided between the developers (unit tests) and the testing department (tests at a higher level).

The form is used for several things in the company. The form plays an important role in the stage where the CCB decides if the proposed change should be implemented or not, as described above. It is found in section 3.3.3 mostly uses the questions one, two, three and seven in this stage. The reviewers and the certification organ read the form to verify that the change doesn’t risk influencing the safety of the system. The form is used as a way of giving the developer detailed instructions to follow the process. The second last two questions are used by the process team to improve the process.

2.5 The Uses of the Form

The form is used to give an exact way of communicating the process to the developers and to give a complete list of all the mandatory steps in the process. It is also used as a report to the certification organ, so that they can make sure that the company has done all the safety precautions they have committed to. It is used to document the knowledge about the system gathered by the developer during impact analysis to simplify future impact analyses. I has five versions since its introduction and the questions are updated approximately every second year.

2.6 Previous Work

There is not much previous work specifically focusing on how forms are used to document work in software engineering. However, there is a lot of work done to investigate if it is better to put your trust in the process or in the employees. The former often indicates that forms should be used more than in the latter. The first part of this section outlines the history of software engineering and how the focus of software engineering has changed between developers and the process. Next is a description of different approaches to bug fix documentation and collection of process assessment data.

Every company has to produce valuable software at a competitive price. The value of the product described in this thesis lies in its safety. However, the company must achieve this safety at a reasonable price, or the customers will buy cheaper products from other companies. The process team has a good picture of how to maintain the safety of the process. The developers will have a good insight of how the safety of the process will influence the efficiency of development. Of course, there is a delicate balance between keeping the process both safe and efficient enough, to produce a good product at a reasonable price.

In the authors opinion there are two views of software engineering which are as old as software engineering itself. The first is the one where the developers are in focus. Here the development is viewed as an art, a creative process. The other view has its focus on the process. It is about control, structure and discipline. Here the focus is on collaboration, measurement and establishment of a process (planning, managing, reducing risk). In this view, development is a science.

In 1981 Barry Boehm presented the COnstructive COst MOdel (COCOMO) in the book Software Engineering Economics. [10] Its purpose is to estimate the time a program would take to construct. Fifteen factors are used to estimate how many lines per man-month a team can write. Next to product complexity factor, the analyst capability factor and the programmer capability factor are the most influential. Therefore he concludes that "the personnel areas of staffing, motivation and management offer the biggest payoffs" [10, p. 688] when improving software productivity. Only one factor, modern programming practices, resembles the management view of software engineering. He has chosen to combine factors such as use of structured programming and inspections since those kinds of modern programming practices "[...] tend to be highly correlated on projects [...]" [10, p. 115]. This indicates that projects at this time were either made without modern programming practices or not. In the introduction to the chapter Improving Software Productivity he says that "Frequently, and somewhat misguidedly, the subject of improving the software productivity is interpreted as the process of introducing modern programming practices [...]". This conclusion of the book tries to counterproof the common assertion that the best way to improve productivity was to use modern programming practices. This is typical for the technical view of software engineering.

The Capability Maturity Model, CMM, was introduced by Watts Humphrey in 1986 in his classic book on process assessment and improvement "Managing the Software Process". [18] It is based on the maturity structure defined by Crosby P.B. (Quality Is Free) which had been used successfully in physics, engineering and manufacturing. In the preface he agrees with Boehm's finding that the most important element in any software organization is talented people. However, he argues that a software process is also needed, since "[...] even the best professionals need a structured and disciplined environment in which to do cooperative work" [18, p. viii]. He argues that organizations with low maturity process might do well temporarily, but they will suffer from periodic crises. [18, p. 55] In this model he wants to emphasize the importance of middle management. He is inspired by the metaphor in W. E. Deming book Out of the Crisis "[...] that process problems are management's responsibility [...]" [18, p. 19]. He believes in the allegory of P. Freeman (Software Perspectives) that "Management's goals and objectives are the fuel [...]" in the complex machinery of an organization. This is in sharp contrast to the view presented by Boehm, that analyst and programmer capability is the most important factor for estimating efficiency. This approach is much about planning.

In the 1980's the software industry had long suffered from an inability to meet commitments, the "Software Crisis". Actually both the COCOMO and the CMM have as primary purpose to remedy this, but in different ways. COCOMO uses a formula using product and project metrics while CMM is more about making a detailed plan.

The book Peopleware (first edition 1987, second edition 1998) by DeMarco and Lister is criticism of the managerial view of software engineering that is written in a rather humoristic tone. This makes it hard to find precise conclusions in it. One of the main subjects is process methodology, a concept that is related to modern programming practices they pinpoint how the managerial view can remove the human factor in both a positive and negative way "The people write the Methodology are smart. The people who carry it out can be dumb. They never have to turn their brains to the ON position". This means that if we have a methodology, we don't have to worry about the shortcomings of the people who execute it, but it also kills the creativity of the work.

The concept of agile development was formulated in the beginning of the century. It argues that a heavy weight process will impose more work to do than the benefit it brings. In terms of algorithm science, it could be seen as a "greedy algorithm" variant of a software process. It proposes that you should believe in the ability of the developers rather than making them following a precisely defined process. This makes it a developer focused view of software engineering.

Agile development does, among other things, "promise [...] lower defect rates" than the waterfall model. [9, p. 57] This should make it a good technique for developing safety critical software. However, it is relatively "untested on safety-critical products" [9, p. 59], possibly because of the prevalent belief that "safety must be designed in a system and dangers must be designed out" [11, p. 190], and agile development contains no such thorough design step. Cozzeti et al. questions if agile is really fit for maintenance settings since they have larger need of documentation. [25] The regression unit tests produced in e.g. extreme programming will, on the other hand, aid maintenance. [8, p. 31] Also, agile methods do look much like the maintenance phase of traditional software development models, in that they work in small increments. Agile and traditional development techniques may be used simultaneously at a company, but this requires up front work. [8] To guide transitions to more agile ways of working, Boehm has developed a model for assessing the applicability of agile best practices. [9] Its main idea is comparing the different characteristics of the product and of the project to the assumed setting in agile and traditional development.

In 1996, Thomas, Hurley and Barnes made a presentation at the Software Engineering: Education and Practice that touch a similar subject. [27] They suggested that "software developers may resist the current approaches to software quality management". They proposed that "Software quality management changes the way software is developed by emphasizing compliance to a defined process, and requiring continual improvement. There is some indication that software developers will resist this new emphasis."

NASA has much experience of using a form for assessing software development methodologies. [17, 4, 12, 23, 3] The forms have much in common with the one described in appendix A. One of the papers from that project investigates how data for understanding the process should be collected. [4] The central point of the method was a document called change report form. One of the results of the report was some lessons they had learned when using the form. In short, their conclusions were as follows. One must build a data collection procedure that considers when and how the data should be gathered, stored and analyzed. It should be investigated if the collection technique influences the data. All stakeholders should be involved so that the goal of data collection becomes agreeable to all. There might be misunderstandings in the communication between the data producers and the data collectors. The last finding is important: assessing the process is expensive.

The book "Managing the Software Process" described above also presents several examples of forms that could be used in the software process. [18] Several situations are mentioned where forms could be used, although forms aren't mentioned explicitly in all those situations. The CCB or management could use the form. It can be saved as a report of the change to baseline, of the bug or of the analysis of the bug. It can be seen as a documented task description of change inspection, task prerequisites and results to control design, coding and inspection. It can be used to find trends in errors and avoid them. It can be used to collect product and process metrics. It can be used for root cause analysis of the bug.

According to Rus the filled in form can be seen as documented knowledge, which is documentation of the knowledge of the system that the developer has gained during impact analysis. [24]

This thesis doesn't focus on the development of new versions of the software, but rather the maintenance of released products. More precisely the context is correcting bugs, Corrective Maintenance. Burton [26] identifies three kinds of maintenance, corrective, adaptive and perfective. Corrective Maintenance is done if the program stops working or doesn't work as specified. Adaptive Maintenance is implementing new features or changing existing functionality. Perfective Maintenance is improving the software, e.g. by using better algorithms or by refactoring.

Chapter 3
Problem Analysis

The developers, reviewers, managers, CCB and the process team were interviewed to give their view of the problems associated with the form. The findings made in a recent "lessons learned" session at the company are also used. The first sections contain the findings regarding quality, efficiency and handling. The remaining sections identify the root causes.

3.1 Quality and Correctness

The company had requested that the quality of the answers in the form should be improved and that the information given in them should be correct. This is important because all the forms are concatenated into a report that is reviewed by the certification organ, see section 2.3.

The purpose of this section is to outline the questions of the form that often have answers that are insufficient, missing or aren't satisfactory for some other reason. It is found that, sometimes, the answer to question 1 has too little argumentation, question 5a isn't answered, the answers to question 4 and 8 is incompatible and the answer to question 12 is ironic.

3.1.1 Question 1 Has Too Little Argumentation

One of the most central part of the safety impact analysis is the first question "Is the reported problem Safety Critical?" since the safety of the system is the most important property of the system (see section 2.1). In most cases the problem isn't safety critical and this thesis only focuses on this case. The answer to this question should be a complete rational to why the problem isn't safety critical. Therefore, the answer should be thorough, but still not too technical.

Question 1 "Is the reported problem Safety Critical?" is pointed out in the Lessons Learned Document as having too little argumentation. In an interview with one of the reviewers the question describing how the problem should be solved also wasn't elaborate enough. The answer to question 1 in 57 forms was printed on paper. This paper was shown to one person in the CCB who was responsible for deciding if a bug fix should be implemented or not. He thought 13 answers were insufficient, and five that might or might not be sufficient, depending on the rest of the answers in the safety impact analysis form. One was too elaborate. Some were not concrete enough. His comments on them were "This describes the problem, not the correction", "Those give me gray hair (the answer was mostly written in C-code)" and "where is the correct place? ('Move variable to the correct place')".

3.1.2 Question 4 Contains Nicknames for Files

Question 4 contains a list of all files that is modified during the change. Some answers include informal names because it is hard to find documents (see section 3.2.1). It is important to use formal names that aren't ambiguous, so that the files can be identified by the certification organ and future developers. The certification organ need the right names to control that everything has been done correctly.

3.1.3 Question 4 and 8 Are Inconsistent

Some of the questions in the form are interdependent in the sense that if you answer a question in a certain way it indicates that the answer to another question should be written in another certain way. If it isn't, there must be something wrong with the second answer and thus the quality of the second answer is low. This sub-section is about two such questions.

The purpose of question 8 is to provide a test that proves that the bug is gone once the fix has been implemented. Since we don't want the bug to reappear in a later release it is a good idea to run this test every time a change is made in the future. Therefore, all tests written by the developer to verify the solution should be written down so that it is possible to notice if the bug has re-appeared. This is called putting the test in the regression test suite. To be able to track why and when the test was added to the suite, question 4, among other things, should list all the new test files that are added to the test suite.

One of the problems the reviewers perceive is when question 8 says that a new test case is needed but no new documents have to be added. Ideally each case should have a new test that should fail with the old code, but pass with the fixed code. This test should be run at the end of the case to prove that the fix is done. Ideally this test should be added to the regression test suite, to see that the particular combination of conditions that caused the error is now safe. This test should be considered as potentially affected in each coming case, and has to be rerun if its module is changed.

This means that the time of doing a case will increase linearly with the number of cases done. The company had tried this ideal way of working, but the cases ended up taking very long, so currently it is not mandatory for the cases with non-sil code. The new guidelines were partly formed by the developers mostly affected by this problem.

One developer who often has to fill in forms said: "When you just have written a fix you want to test everything extra carefully. We usually don't test everything in the functional tests, but if you write an extra paranoid test in a form you always have to make a new test case". It should be noted that most cases comes from the testing department. Those tests often have quite exotic preconditions. Thus, the problem for the developer is to choose between making a careful test to make extra sure that the change have been correctly put into the code, or to make a test that is as small as possible, or non-existent, to reduce future workload. Therefore, the developer describes a test in question 8 to make sure that his implementation works, but avoids mentioning the test in question 4 so that it won't end up in the regression test suite. Then the reviewers have to react, because there is a significant difference between the test suite and the test described in the form. Management and the process team have yet to decide how well a change has to be tested. If it isn't perfectly clear what tests should be in the test suite, the reviewers and the developers will have to settle this for each form. This is an important root cause to the problem.

3.1.4 Question 5a Is Not Answered

Question 5 and its sub-questions are about the inner working of the system and to some of the developers it is seldom relevant. In those cases the question should be answered not applicable. Each sub-question should be answered this way, like so

```
5)  Q: Question 5
    A: a. Sub-question 5a: Not Applicable
       b. Sub-question 5b: Not Applicable
       c. Sub-question 5c: Not Applicable
       ...
```

One common error occurs in this question since many sub-questions in a row are to be answered with "Not Applicable". Sub-question 5a is much longer than the rest of the sub-questions. At a quick glance, it appears that sub-question 5b is the first one, as can be seen in appendix A. This is probably the reason of why it is missed.

The reviewers need answers to each of these sub-questions because otherwise there is no way for them to tell if the developer had intended to fill it in with "Not applicable" or if he had missed to think about the question.

3.1.5 Question 12 Contains Irony

In question 12 the developer should try to think of how future similar problems could be avoided. This is an important step in finding deficiencies and improving the process. The developers sometimes wrote ironic answers to question 12. This is not acceptable since the report will be sent to the certification organ. According to the Lessons Learned developers tend to forget that. The process team has said that the answers to question 11 and 12 have only been used once to improve the process. The developers might have sensed that their answers aren't used much. This might be one of the reasons for them using irony. The process team said "When we introduced the root cause analysis field we discovered that a very large amount of errors could have been avoided if we had better code review. The purpose of this field is to find problematic parts in our development process." By problematic parts they mean stages in the process where improvement will greatly reduce the risk of bugs being introduced. This is exactly what is asked in question 11. It was found that question 11 and 12 contained approximately the same information, by comparing them in many forms.

3.1.6 Current Approaches to Improve the Quality of the Answers

The process team conducts safety education for all personnel to maintain a good safety culture at the company. The goal of the education is that everyone understands what the safety is all about, but it is acceptable if they just know where the information can be found. Once however, the process team had said that some of the staff didn't know about some of the most important safety documents. This indicates that the education hasn't worked as expected.

3.2 Efficiency

The purpose of this section is to find what the developers think is the most time-consuming or annoying activities when doing impact analysis or filling in the form. It is found that the connection between the documents can be hard to find in the old parts of the code. The same goes for the names of the documents. In another case one small impact analysis had to be rewritten to a very large one because of changes to the requirements of what should be reported in the form.

The developers who seldom worked with safety critical parts of the code (see section 2.1) thought that the form was too long for them. The last sub-section is an investigation on if the younger employees have a different view of the need for forms. No conclusive results are found to this question.

3.2.1 Finding Documents

Finding documents and (traceability) links between documents is the central part of ordinary impact analysis (as opposed to safety impact analysis, see section 2.2). The names of the documents are stored in word documents, and a developer might have to look at several documents to find the names he is looking for. One developer thought that, even though it was easy finding the related document, it took too long time. All interviewed developers had worked at the company for many years, so they had learnt where to look for the different documents, if they were working in familiar parts of the code. However, many developers thought safety impact analysis was hard in unfamiliar parts of the code, especially code that was written a long time ago.

3.2.2 Everyone Uses the Same Form

There are many different parts of the code in the software developed by the company (GUI, operating system, libraries, signaling etc). The form is written with the intention of having questions that are relevant for all. For example there is a specific library question. As a consequence, almost all developers have one or more question where they almost always answer "not applicable", see section 3.1.4. For example, a GUI-developer almost never has to fill in any of the sub-questions to question 5. Then he is required to write "not applicable" or "N/A" for short, on each of these sub-questions. One of the most common complaints about the form is that all developers have to answer the same questions.

In the interviews, the developers mentioned it as an annoying property of the form that some questions are seldom important to them, and in some cases the change is so small that a full safety impact analysis doesn't seem necessary. In the sub-section below it is verified that the code a developer normally works with heavily influences what questions a he answers “not applicable”.

3.2.3 The Time It Takes To Fill In the Form

A small study was performed to see what questions took the longest to fill in, and which questions was the hardest. Since only three developers were interviewed the raw data (each person's answers) won't be shown in its whole, to make the answers anonymous. The conclusions drawn from the data are uncertain since it includes very few participants. However, the answers indicate that some developers spent almost no time on questions that were among the hardest for others. Future investigators should try to find out if tool support is wanted for the easy or for the hard questions in the form.

3.2.4 Increased Demands on the Contents of the Form

Sometimes old safety impact analyses have to be updated due to changes in the process. This could be a very costly process. When asked about a particularly nasty safety impact analysis one developer mentioned a case he had to reopen after the developer who had worked on it before had quit the company. “They had changed the rules of what should be in the form. The other developer had changed an interface, a small piece of functionality in a module. Since the change was in an interface, it influenced the whole code base. The original change was to tidy up the code, the code and interface was too interconnected so they were split. Header files and name spaces had to be changed in all the calling classes. He had mentioned the module and the interface in the original form, and it was tested (Some tests were changed too) but it affected 30 other modules too and the rules had changed so we also had to run tests for all modules affected by the case. These were spread around the code base, both parts that I knew and parts that I didn't knew so I had to look at many tests and test specifications, check logs and stuff.” It should be noted that some tests that had to be re-run because of this added header file had to be performed manually.

3.2.5 The Influence of Academic Education

It is possible that some developers think that the form takes too long to fill in because they don't see the benefits of doing it. The benefits of such process related things can be proved in practice by exposing the improvements of the products to the developers. Another way to convey such a belief is to include it in some kind of education. Over the years, the universities have put more focus on process related issues in the education of software engineers. It is possible that developers who have graduated later will have a better understanding of why the form is needed than the ones who graduated earlier. The part of the company described in this thesis has recently been forced to do some rationalization. Swedish law favors keeping employees who have worked a long time, so currently there are few at the site that have graduated recently. This sub-section tries to find out if this influences the general view of filling in the form. It is found that the employees don't support this theory in general.

In one interview a developer suggested that that the younger developers had a better understanding for the value of the different steps in the process. He thought that it couldn't be directly measurable, but that he could "hear it in how they talked". To investigate this, six developers was asked if there was some group who disliked filling in the form than others. Four of them said that it was mostly a question of what kind of person you are, and one clarified it as, "Some like a formal way of working and some are more 'hands on' ". The last thought that "[...] everyone dislikes [filling in the form] as much". Three groups that were often pointed out was the ones who didn't use it much (often exemplified by the GUI team), the ones who worked in old code (because it is harder to do safety impact analysis in old documents) and the team who was the most affected by the growing regression suite problem mentioned in section 3.1.3.

Since the answers were often somewhat avoiding and careful, a reformulated version of the question was asked to make sure that the answer was avoided because it was a delicate subject. The new question was if there was any type of personality who disliked filling in the forms more. Two developers said that maybe people who had worked there longer could be like that but the second said "I would say that people who have worked here a long time dislike it more, but I have worked here a long time and I don't dislike it." He thought that this might be because he had been part of the group responsible for putting together the safety impact analysis report "You realize that if you don't fill it in in a good way, some other person have to fix it for you at a later stage." In the lessons learned document it is suggested that the developers should be part of writing the Safety impact analysis report. One of the developers was specifically asked if there was a difference between people who had been around a longer and shorter time. He thought that "When you have worked here for a while you learn to accept all the things you need to do", that is, the other way around from the original assumption in this sub-section. It is possible that this applies more to people who are more "hands on" to start with.

Only a few of the asked persons think that the developers view of the form has to do with when they went to school. The results of this section are inconclusive.

3.3 Handling

A form should be filled in at the right time, reviewed properly and otherwise handled correctly. It should be filled in at a time when the information is easy to get. In this section four possible problems with handling are found. Firstly, sometimes coding is done before it is authorized. Secondly some answers are copied from old forms. Thirdly, only some parts of the form are used by the CCB, and hence the form is reviewed in its whole for the first time at the end of the release cycle. Lastly, the developers are working with many cases at the same time, possibly making each case take longer to finish.

3.3.1 The Time of Implementation

One of the main purposes of impact analysis is to give the CCB opportunity to compare its estimated value and its estimated cost. If the bug fix is implemented before the CCB has approved it, the CCB can't manage the cost of implementing it.

Some developers said they sometimes were so sure that the case would be approved they started implementing before they were supposed to. Others said they implemented a bug fix to ensure that they actually had found the bug, so that the CCB shouldn't be bothered by investigating a solution to something that wasn't the problem.

3.3.2 The Developers Copy Old Forms

When someone constructs a process where a form is used, they probably assume that each form will be completely empty at the beginning of the process and completely filled in at the end of that process. However, when the form is a regular digital text document it is possible to start with an old filled in form, thus invalidating the first above assumption. The safety of the product relies on the developer to actively consider each question in the form. The developer will only think of those aspects passively if some answers are already filled in from the start.

Some developers had saved one or more text files (see section 2.3) with old filled in forms. When they work on a case that is similar to a one that they had worked on previously, they used the filled in form from that case, and just edited the parts that differed. In section 3.2.2 it is found that each developer has certain parts of the form he answer more often than others. Thus, keeping old forms enables them to only answer the questions relevant to their part of the code.

This has unfortunately resulted in that some errors have been spread in many documents. The missing answer to question 5a in many documents is most likely a result of this. If some answers are just checked passively by the developer, it is more likely that they will be overlooked.

3.3.3 Quality Assessment Times

The faster the errors in a form are found, the cheaper it is to fix them. Also, it is better from a safety aspect if the developer has the case in fresh memory. Still, with the current process, the form is properly reviewed for the first time at the end of the release cycle.

Only some of the information is used when the CCB decide if a change should be implemented or not: "Well, I only look at parts of the forms, if it affects SIL, how it would be solved and try to figure out what phase it was introduced in. I don't look at what tests they are going to run and what requirements are affected. Sometimes it's hard to see the cases' size by just looking at the form. Then I look at how many tests there are; if there are many it's a big case. I look at questions one to three and [question 7]." Therefore, this thesis recommends keeping the questions one, two, three and seven in the form.

Since only a few of the questions are used by the CCB, the other answers are reviewed for the first time when the case is closed. In the lessons learned document it is suggested that the CCB should have a checklist to ensure that the form is complete. Another problem mentioned in the interviews and the lessons learned document is that the forms are reviewed for the first time at the end of the project, see section 2.3. Some years before the thesis was written, the review team attended the CCB meetings and thus got a chance to review the quality of the answers at an early stage. It has been suggested in the lessons learned document from the latest version that the reviewers should be part of the CCB again.

3.3.4 Efficiency in Different Stages

If a case is lying around not being worked on for a long time, it might take more effort to finish it. It is also a safety problem because the risk that the developer forgets about some implicit safety aspect of the case improves over time. This also influences the developer's ability to explain his answers to the reviewers. A local tool for measuring the state and mean time of all cases was used to investigate how long the different stages of the cases had lasted had changed over the last years. This sub-section contains much data, some efforts to analyze the data but no conclusive findings. At this stage, the data investigation is too immature to be of general use. It is presented anyway, because it might be of interest to the investigated company.

The developers currently have much freedom in choosing which case to work on. This is because most of the cases originates from the testing department, and thus probably don't influence the safety of any real system. However, when a failure report is received from a customer, the case will always have high priority. The developers like the freedom of this liberate approach to choosing what to work with, but it might introduce an efficiency problem. One thing that takes long time when working with a case is setting up the environment, as mentioned in section 2.3. Virtual machines have to be installed and different kinds of hardware have to be collected and connected. If it takes long time between debugging, implementing and testing, this configuration might get lost because someone else needs the hardware or because the virtual machine takes too much space. Also, it gets harder to recall the details of a case if you don't work on it for a long time. Debugging is very much about understanding the code around the bug. In such a big system it is impossible to grasp the details of all the code at once. If you don't work on a bug for a long time you will inevitably forget the details. If you have to work on it again, you will have to do a lot of work to recall the details. This might be less efficient than trying to finish each case as fast as possible by trying to finish the open cases before opening new ones.

Five Developers were asked if they thought that cases took too long from debugging to testing in general. Three of them said maybe, the other two thought they did not. One added that the most inconvenient time gap was the time it took for the CCB to check the implementation and allow for unit testing.

Some years ago the managers noticed that cases suddenly took longer time from start to finish. An effort was made to investigate if there were any bottlenecks in the flow of cases. It was found that many cases were stuck in a stage where the CCB had to approve them for the next stage. To avoid this happening again, a tool was built that analyzed the how long the cases were in the different stages, on average. This tool lets you select version, change type, how severe the change is, case stage and such. The tool computes the shortest, the longest, the mean and the median time for the specified conditions. Often, the longest case had been lying around for several years. The mean times were heavily influenced by this, so the median was used instead in the analysis below. Even though management is currently content with these figures, it might be more efficient if the times are shortened even more.

The statistics tool was used to see if there was any trend in how long time the different stages in the cases took. The time it took for the CCB to approve the change and decide that the case could be tested was investigated for the different sub-versions. The product's

Version number/criticality	Low	Medium	Hard	Critical	All
2.2/3	15	5	4	1	3
2.2/4	60	12	6	4	6
2.2/5	?	18	7	7	13
3.1/1	21	12	18	13	17

Table 3.1: Median of days that the CCB takes to approve a case for testing after it has been marked as implemented by version and severity level. The question mark is caused by an unknown error in the program.

Version/service pack	/1	/2	/3	/4	/5	/6
1.1		4	5	4	7	10
2.2		7	3	6		
3.1	17					

Table 3.2: Median of days that the CCB takes to approve a case for testing after it has been marked as implemented ordered by version and sub-version. Empty cells correspond to small or unreleased versions.

version number has the following form “v.sv/sp” where the “v” is the version, “sv” is sub-version and “sp” is the service pack number. A service pack is an intermediate version that doesn’t include any new functionality, only bug fixes. The version numbers have been altered so that the product can’t be identified.

Table 3.1 and 3.2 presents the time it takes for the CCB to approve a case for testing. They show that the time has increased over the versions. For example, service pack 6 of version 1.1 median time for approving testing is 10, more than twice as long as the four days median time for the second service pack of that version. For the service packs this is expected, since the late service packs are old and have low priority (there are only a few locations where they still are used). Bugs with simpler preconditions will happen more often, and they are easier to fix, so they will be noticed early in the lifecycle of the version, and they will take relatively short time to fix. The bugs with complex preconditions are less likely to occur (because they require a large set of preconditions to be fulfilled simultaneously) and will be hard to debug (because a larger set of preconditions is harder to reproduce). Thus, it is expected that the bugs found in the early service packs will take shorter time to fix. Version 3 does not follow this pattern, but the data-set is so small that this can be considered an outlier. The reason might be that the bugs are found much faster in the first service pack, so that some of them have to be deferred for a couple of weeks. This doesn’t seem to be the case with service pack 2 in version 2 and service pack 1 in version 3. Version 2 had 26 new change requests per week in 2008 whereas 3.1 had 28 in the middle of 2010. The numbers might be misleading because version 2.2 includes cases of an informal type whereas version 3.1 does not. Another factor is that during 3.1/1, the unit testing was only allowed on a “labeled build”, i.e. a build of the software that had a formal name. Then the developers had to wait for such a build before they could test their code. There was one person responsible for making the labeled builds and no clear schedule for when this should be done. Since it took some work for him to do so, it was in his interest to make labeled builds as seldom as possible and the developers had to spend some time convincing him to make a labeled build each time they should test a case. One developer suggested that the dates for the labeled builds should be more formal.

The conclusion is that there is no clear trend indicating that the cases have begun to take longer time, at least not recently. Future work should try to validate these assumptions, using interviews with the employees of the company.

3.4 Traceability

The main task of impact analysis is finding the things that have to be done to accomplish a change, see 2.2. This includes finding the documents related to the code where the bug is removed. In section 3.2.1, the developers identify this as the most time consuming activity when doing impact analysis in the old code. There are three reasons. The first two reasons were that the system had been forced into a more hierarchical design, and that requirements had been enforced. This wasn't always done in a consistent way. The third reason was that not all of the information was moved when a new tool was introduced.

3.4.1 The Historical Architecture of the System

When the product described in this thesis is to be released in a new major version, its architecture is entirely remade. When this is done, much work has to be put into updating the traceability matrix (see section 2.2). It is also important that the architecture neatly captures how the code is really connected. Otherwise it will be harder to do impact analysis.

Before the safety development reform the architecture of the system was flat, it had few levels. Over time, the number of system parts grew larger, and some parts grew very large. When a new major version of the product was planned, a new architecture with more levels was made. Some of the smaller system parts were merged into bigger system parts. Then they were divided into sub-systems which were further divided into components. Unfortunately, this new division wasn't always consistent with the "as built" design of the system.

3.4.2 Requirement Documents Become Mandatory

The company divides the code by requirements. If there is a clear coherent thought in this division, it easy to find what requirement is connected to the part of the code the change described in the form is.

From the interviews, it was clear that most of the work on the requirement documents of the product was done at the end of the release cycle. There were two main reasons for this. The first is that it is easier to write the requirements when they knew what functionality would actually be in the product. Secondly, the additional work required to maintain requirement documents wasn't motivated by the perceived benefits. However, at this point in time, the certification organ demanded that every code file should have an associated requirement document. The new architecture described in section 3.4.1. The Historical Architecture of the System was used as a raw model to write requirements for the code. This proved very hard and some requirement became much larger than the others "And also to find the connections to the requirements. Maybe no requirement exists and maybe one requirement is for this much (makes grand gesture with both hands) because the old parts are so different". What he is trying to say is that you could look for a long time for a requirement and end up realizing that it actually is missing. In some cases you would also have to read several large requirement documents to find the specific requirement you are looking for. If the big documents had been divided and named properly it would be easier to find what you were looking for.

3.4.3 Moving to a New Tool

The introduction of new types of documents 3.4.2 has raised the need for larger and more flexible tools to handle them. When the new tools was introduced effort was made to move all documents from the old tool to the new one, but not all documents were moved. Therefore developers have to use different tools to find the traceability links of a certain part of the code.

One developer thought that using two tools might be a problem for new developers. Another developer had experienced missing a document because he used just one of them. In some cases nicknames for the documents are used in question 4 in the form because the correct names were too hard to find (see also section 3.1.2).

There is a tool called Locator which can find the traceability links in the old system, but it isn't used often. It is unclear why, but one developer said that he did not use it because it didn't work in the hard cases anyway. Incorporating a new tool in how they currently work will take some effort, so it is required that it is significantly easier to use the tool than not.

3.5 Soft Issues

Everyone who was interviewed understood the purpose of the form, but some had ideas of how it should be improved. It would be desirable if the form was easier to improve, especially to remove questions from it. The developers would like to have more influence over the form and the process in general. The developers seemed afraid to be misquoted in the interviews about the form. It is unclear why.

3.5.1 A Global Process

One important aspect of improving the process is making it global, that is, using the same process at all sites (see section 2.1). Then everyone can know what to expect from their coworkers, even if they are on the other side of the earth. But a global process also means that any change to the process have to be assessed by many persons with very different preconditions. Also, changes will take very long from the initial idea to actual implementation. One developer said "Everyone thinks it is hard to change how we work at [this company]". When another developer was asked if he had tried to influence the process he answered "The process never changes, and if it changes it grows." This highlights another perceived problem. A process that is continuously improved from a safety perspective will eventually become very large, as opposed to a process that is improved for efficiency. It is possible that if the developers had more influence over the content of the form, it might be easier and more efficient to work with.

3.5.2 Removing Questions

The developers who work in parts of the code that aren't safety critical would prefer not having to fill in the form each time they are working on a case. However, the purpose of the form is to ensure that when a developer fixes a bug, he will always think about each question. Even if he always gives the same answers to a question, there is an ever so slight chance that once in his career he will answer it differently. This is so important from a safety perspective that it is worth to spend the rest of his career supplying the same answer. Therefore, the process team and the certification organ are extremely conservative when it comes to having different forms for different development teams.

However, this conservative attitude makes it hard to remove questions, even if there is a fairly good motivation for it. Question 12 is an example, see section 4.3.

3.5.3 Influence over the Questions of the Form

At the time of this thesis, the process team was both part of the reviewing team and conducted a dialog with the certification organ about improvement of the process. They had had problems keeping up with these two responsibilities. During the interviews, it was fairly hard to get to talk to the process team and there was seldom time for deeper discussions. However, there are times when the communication did works. For example the developers had been part of forming new guidelines for when new design tests should be added. This might be a reason for the developers feeling that they don't have anything to say about the process, the process team simply is too busy assessing the process and communicating with the certification organ. Any discussion on the subject will be ended promptly due to lack of time to investigate the details. One developer said that "If someone says 'Safety', the discussion is over", when he talked about discussions with the process team. If the developers felt that they could influence the process, or at least were given a chance to discuss their opinions, they might find more reason to follow it. Why should the developer bother to follow a detail in the process they perceive as faulty if no one bothers to assess or fix it. It should be noted that the process team has good intentions in wanting to make the questions easier for the developers; they simply don't have time for unplanned discussions about it.

The conclusion of this chapter is that the developers don't think that they are able to influence the process because the process team seldom has time to give their suggestions serious consideration.

3.5.4 Expressing Opinions about the Form

Interviewing the developers about what they thought about the form wasn't always straight forward. Some of the developers didn't want their interviews to be recorded when they were asked about the process. One interviewee even wanted to see the notes written down during the interview, and one explicitly asked to keep what was said a secret because “If anyone knew I said this, I could get in trouble!”

It seemed that they were careful when talking about what they or other people thought of the form. There are at least two possible reasons, that they were afraid to be misquoted or that some of their opinions were a secret that shouldn’t be told to everyone. Did they worry that someone thought they had said something they hadn’t? Did they think their coworkers would be insulted if only some of what they said was quoted? Were they afraid to seem like gossip starters? All these reasons are acceptable, but they indicate that the subject (to have an opinion about the form) is sensitive. On the other hand, if they want to keep some of their opinions a secret, it is really a problem. Why do they think that some opinions shouldn’t be expressed? There is no clear answer to this. It is important that the company works together for maintaining a culture and spirit that support the safety of the process, because it is possibly their most important sales argument. Still, there might be unforeseen problems if developers are afraid of voicing their opinions.

Whatever the reason, this carefulness is probably an important obstacle to solving many of the perceived problems. Future work should try to investigate this, but much care must be taken to convince the developers of their confidentiality. Such confidentiality must be respected even if it reduces the extent of the research results.

3.6 The Results of This Chapter

The company wished to improve in three areas, quality, efficiency and handling of the forms. The developers, the CCB and the reviewers were interviewed to find problems in those areas. The forms had low quality or incorrect information in the answers to following questions.

- Question 1, argumentation is not elaborate enough.
- Question 4, nicknames for documents are used. The reason is that the correct names are hard to find due to insufficient work during product growth, changes in the process and introduction of tools.
- Question 8 indicates new regression tests that aren’t mentioned in the list of files in question 4. The developers consciously avoid writing this to avoid overzealous work in the future.
- Question 12 is answered ironically because the developer can’t find a good answer.

The developers think that the hardest part in doing impact analysis is when it is done in old parts of the code. This is because of the same reason as the one for question 4 above. The developers think that it is unnecessary for them to fill in some parts of the form. No connection between this and the age of the developers could be found.

There are three potential handling problems with the form. The first is that the developers sometimes implement the change before the change is made. The second is that the developers copy old filled in forms. The third is that the developers work with many forms simultaneously, thereby possibly making them take longer to finish.

The developers feel that they don't have anything to say about the form. It is possible that the developers are afraid of saying anything about the form. This might be a de motivator to following the process. The process team has too many urgent responsibilities to engage in unplanned discussions with the developers about process improvement.

Chapter 4
Possible Solutions

In this chapter future possible ways of working are explored and compared to the company's current way of working. None of them will produce drastic improvement, but that is expected, see [13].

Agile development is a popular method of reducing the need for documents in the software engineering process. However it is untested in safety critical environment and it is possible that it isn't beneficial to use in an environment with such stable requirements. It is discussed if it is motivated to implement bug fixes before the CCB has decided the bug fix should be implemented. It is found that is economically motivated to do so when correcting bugs. The questions 11 and 12 are discussed to see if they belong in this kind of form. It is found in the literature that it is a common practice to collect root cause analyses as part of the configuration management. However, it is suggested that question 12 should be optional since process improvement suggestions should be made in a group. Different possible ways of improving the communication between the developers and the process team are discussed. In the proposed solution, the developers have other developers representing them who meet with the process team regularly.

4.1 Agile Development

This section discuss if the environment of the projects that work with safety critical code is fit for agile practices. The conclusion is that agile practices can be implemented in this environment, if it doesn't interfere with the safety requirements of the process and if there is economical motivation.

The ideas of agile development began forming in the beginning of the century as a reaction to the ever increasing software development processes in companies in the USA. Originating from a manifesto [7], many forms of development have been described, notably SCRUM and extreme programming (XP).

The team who develops the GUI has used agile methods for some years with positive results. There are many reasons for why the rest of the teams should too. Agile development have less need of documents, and a more agile approach could possibly reduce the need for the form altogether. Extreme programming can reduce the complexity of the code by using refactoring.[1] It should also be useful for safety development since it "promise [...] lower defect rates". [9] If Agile development should be introduced in only parts of the organization "maintenance can [...] be a good place to experiment with agile approaches" as "agile development makes the entire development cycle much more like a maintenance phase by providing short, focused iterations". [8] The tests produced during test driven design will help ensure that no functionality is unintentionally broken during maintenance. [8] However, this observation is not restricted to agile development, end the company is already working hard to making regression tests automatic. The lack of other documentation might be a problem. One member of the process team argued that safety impact analysis is hard if you don't have any design documents, because the developer have to rely on his memory to find the ripple effects. Also, documenting the safety impact analysis supports thinking about all different aspects the solution "Sometimes you have thought about it during the investigation and sometimes you don't think about it until you see the question, 'Oh, right, I have check this too!'. It's like a checklist."

In 2003, agile development was still relatively untested in the development of safety critical software [9, p. 59]. This could be because of a long standing belief that "safety must be designed [into] a system". [11] Vuori have compiled a large list of how to tailor agile practices to fit safety development [28], but one should be careful not to tailor agile practices too much, or their benefits are lost. [8]

Boehm and Turner [9] have formed an approach for finding a balance between plan driven and agile development that can be tuned for different projects. It is divided into five steps. In the first step risks associated with agile and plan-driven methods, which, in the second step, are evaluated in the context of the project at hand. If not agile or plan-driven methods clearly is the best choice, the parts of the project best suited for agile are identified, and an architecture is formed that supports using the two methodologies. It should be noted that mixing agile and plan-driven development requires some work up front. [8] The company has already moved the GUI team to agile. In the fourth step an overall project strategy is formed, and in the last step the resulting process is evaluated and revised in parallel with executing the project.

In step one, the risks in [9] are chosen, but as is clarified in that paper "The candidate risks we describe are just that–candidates for consideration". These are called personnel, dynamism, culture, size, and criticality. They are fully described in [9], but a brief description is made here for convenience. The personnel risk describes the relative amount of "programmers who are trained to follow a process" and software developers. The term "trained programmers" is supposed to mean developers who have technical skills but have had trouble following processes and working in a larger group. A "software developer" is defined as a person to whom following processes is more natural and who can aid to a culture of working together. According to [9], an environment with more "trained programmers" fits plan-driven development better than agile. This is in contrast with a manager at the company who claimed that "If you hire a programmer you will be successful in one case of ten but if you hire a software developer you fail in only one case out of ten". The second risk, dynamism, is to be measured by how many of the requirements will change

[1]On the other hand, this study has shown that refactoring reduce maintainability. [29]

per month. If this amount is high, an agile approach might be preferred. The culture is defined as the relative amount of people who likes a “creative chaos” versus them who likes to work under more ordered forms. The size of the project is the fourth risk and it has 30 people as a border between preferred agile and preferred plan-driven. The last risk, criticality, is perhaps the most important in the context of this company. It is graded as the loss a software failure will produce. The grades are comfort, discretionary funds, essential funds, single life, and many lives. Agile is preferable if the criticality is low. Since this product is very safety critical (see section 2.1), plan-driven development is preferred.

Here the risks are evaluated in the context of the different projects (except the GUI team). As is indicated above, the company tries to hire “software developers” rather than “programmers” according to [9]; this should advocate an agile approach. The dynamism of requirements is low in this setting, so the need for agile practices is low. This supports a plan-driven approach. No investigation have been done to investigate the culture precisely, but the findings in section 3.2.5 indicate that there is some amount of both; “Some like a formal way of working and some are more ‘hands on’ ”. The size of the teams is small enough to fit an agile approach. The system is used in some settings where many lives could be lost if the system doesn’t go into a safe state. This means that the criticality risk is high.

One big drawback for agile methodologies is that the safety product industry has a long tradition of process focus. If agile is to be introduced, much of the knowledge will have to be reinvented.

The conclusion is that two things support agile: the teams are small enough, and there are enough developers skilled at following and tailoring a process. The culture might or might not support agile, but the requirement dynamism factor certainly does not. The big problem, though, is the criticality risk. To summarize, agile practices should be possible to adhere to, if they don’t interfere with the safety, and if there is economical motivation in spite of the low requirement dynamism.

4.2 Speculative Bug-Fixing

This section will argue that it is economically motivated, under some circumstances, to implement a full solution to a bug without having this approved first. The main reasons is that the effort of implementing the change will increase with time, that the cost of moving tasks of a work product back and forth between employees, and that there is higher probability that changes in corrective and perfective maintenance aren’t working, compared to those in adaptive maintenance.

In the analysis chapter it is stated that a bug fix is often implemented in parallel with the debugging activity, even though it is clearly stated in the process that implementation should be done after it is approved by the CCB. This is in accordance with current models of software maintenance and impact analysis.

4.2.1 Adaptive and Corrective Maintenance

This sub-section will discuss if the expenses and risks of the different phases are distributed approximately the same in adaptive and corrective maintenance. The result is that it can neither be proved nor denied that there is any difference.

In 1976 Burton identified three types of software maintenance, namely corrective, adaptive, and perfective. [26] Corrective maintenance is the same as fixing bugs, adaptive is preparing the program for anticipated changes in its environment, e.g. adding new features, and perfective maintenance is improving the program without changing its functionality by, for example, refactoring or by using better data-structures. It is important not to handle adaptive maintenance as the default, and the others as variations. The preconditions for the three are quite different.

It should be noted that the text bellow are mostly guesses that might be heavily influenced by the company where they have been made. It could just be viewed as inspiration for future research to prove or deny its conclusions.

The location of the change is harder to find in corrective maintenance. In adaptive maintenance it is known that the system for some reason isn't fit for its future environment. Maybe a feature could be missing, or a newer version of a library has to be used. The exact location and form of the implementation might not be known, but with some fantasy, and knowledge of the system, some viable solutions shouldn't be too hard to find. In corrective maintenance, on the other hand, finding the location in the code of the bug is a complicated process. To find the hardest bugs, print statements might be needed or even simulating the program execution by hand. [15] Debugging parallel code is even harder because of the probe effect (debugging code might change the behavior of the observed code), it is hard to repeat (the bug only appears in some schedulings) and the fact that there's no global clock. [22] There are tools for debugging operating systems [19], distributed systems [1, 5] and parallel systems [20] but they are not always applicable [15].

Even when the developer has found the piece of code that probably caused the failure, he might not be quite certain. Sometimes the easiest way to verify this is to implement a bug fix and see if the problem remains. Maybe the developer just wants to see what will happen if he makes a certain change in the code. This might be the reason why the developers implemented bug fixes in the debugging stage. Most models suggest that debugging and bug-fixing should be seen as separate activities. [2] It would be interesting to see a field study where this suggestion is evaluated.

Another difference between adaptive and corrective maintenance is that the implementation of the adaptive maintenance probably will meet its goal. If a new feature is designed and implemented it is unlikely that the change actually won't add the desired functionality. In corrective maintenance a maintainer might very well think that he has found the bug, but he can't be sure before the bug fix is actually implemented.

The last difference is that changes in adaptive maintenance are inherently bigger. Adaptive maintenance is done due to a change in the environment of the software that wasn't necessarily planned for. If the design of the system allows for the change to be implemented easily, it is either because of hard work to make a good design to the software, or mere luck. In corrective maintenance, the system is designed to work, and if something goes wrong, there is probably room in the design to fix it.

4.2.2 CCB in Corrective Maintenance

In the Analysis chapter, it is found that developers sometimes implement a bug fix before the CCB have decided that this should be done. This sub-section discuss if the process should be changed to allow this. The conclusion is that the process should allow skipping the CCB step after debugging and the implementation step, if a bug fix has been implemented during the debugging step.

Four possible scenarios are depicted.

- C: the developer is right in thinking that he has found the *Cause* of the failure.
- not C: the bug is somewhere else.[2]
- D: The *Designed* bug fix will remove the bug.
- not D: The failure will still appear after this bug fix is implemented.

Two stochastic variables are defined to model the likelihood that a proposed bug fix will work. Let P_{cause} be the probability of C and let P_{design} be the probability of $D \mid C$, that is, the probability that the solution works, in the cases where C is true. Conceptually we have three interesting cases once a bug candidate is found:

- the candidate bug is actually the sought after bug and the developer succeeds in fixing it,
- the bug is found but the bug fix doesn't work and
- the candidate bug is actually working code and therefore "fixing" it won't solve any problem.

There is a fourth case, when the bug is fixed by chance even though it wasn't found in the first place, but that is unlikely and won't be considered.

Given the terminology of these probabilities, three ways of describing the actual process are depicted. The first way is the canonical way, where everything is done in the way it is supposed to, as described below. The second way describes how it is actually done, and the third way describes how the process could be changed to better mirror how things are done.

In the first way, the developer starts by using debugging to find the cause of the failure. If he finds something, it will be the right thing with likelihood P_{cause}. The next step is designing a bug-fix. The probability that it solves the problem is P_{design}, if we assume C. A preliminary safety impact analysis is done by the developer, and the CCB uses the form to decide if the proposed bug fix should be implemented. The bug fix is implemented by the developer and the form is updated based on what he has learned during the implementation. During this stage, he will probably execute the instructions in the failure report to see that this no longer causes a failure. The probability that everything will work fine is approximately[3] P_{cause} times P_{design}. Otherwise it is necessary to return to the debugging stage to see what went wrong.

A second way to do it would be to implement a bug fix as soon as a possible bug is found, but otherwise to follow the process. This will make the implementation stage symbolic. A third way would be to remove the stage where the CCB decides if the change should be implemented, and combine the debugging and implementation step.

[2]It is of course possible that the developer doesn't find any suspicious parts of the code that is the bug (this is actually quite common). This case won't be investigated since any process model must stop in this case.

[3]I don't count the probabilities that the actual implementation is erroneous or the case $D \mid \text{not } C$

	First	Second	Third
Fast feedback in debugging		X	X
Implementation always works		X	X
Promotes thinking before implementing	X		
Implementation is always approved	X		
Follows a process	X		X

Table 4.1: The benefits of following the process not follow the process and changing the process.

The first way has the benefit that it is the most observable one. It is always clear in what has been done in the case since the process has no ambiguities, and it is followed. It promotes the developer in thinking about the solution before he implements it since the design and implementation step is separated by an impact analysis. If this way is used, the developer will never implement a bug fix without this being approved. The drawback of this way is that sometimes, the solution doesn't work. Then, the work of the CCB will be in vain. The developer will have slower feedback on if the solution works than in the other two ways. On the more personal level, it might be embarrassing for the developer if he has presented a solution that didn't work.

The second way has the apparent drawback that it doesn't follow the process. It has one stage that isn't used, implementation. It has fast debugging feedback and the proposed solution always works. The third way has the main benefit that it mirrors how things are currently done. The benefits of the three ways are depicted in table 4.1.

In extreme programming (a type of agile development) it is recommended to implement a bug fix to be sure that it actually works. [6, p. 32] This is because of the basic philosophy of never doing work based on a guess. Therefore it would be wrong to start the process of writing and review the impact analysis of the change before one knew if the change could be proven to remove the bug.

It is possible that there are other situations when the naive version of a process model isn't the most efficient. Having perfect control of if a step in the process is done might cost more than actually doing it. This is especially true in the case of bug fixing.[4]

4.3 Removing Question 12

One apparent way of making the form faster to fill in is removing questions that don't seem to fit. In the form, there are two questions that have more to do with root cause analysis than impact analysis. Most definitions of impact analysis are looking into the future to find the effects a change, see section 2.2. Root cause analysis, on the other hand, is trying to look back in time to find the events that have caused the need for the change. The form has two questions that have to do with root cause analysis, namely question 11 and 12, see appendix A. Question 11 is about finding the stage in the process where the error was introduced and question 12 lets the developer suggest how similar errors could be avoided

[4]An opponent pointed out to me that this section ignores the probability that the CCB disapproves the implementation. The results in this section should only be used with this taken into account.

	In the form	Somewhere else	Not mentioned
Harjani and Queille		X	
Basili and Weiss	X		
Lehtinen et al.	X	X	

Table 4.2: Places where the literature says the root cause analysis may be documented.

	In the form	Somewhere else	Not mentioned
Harjani and Queille			X
Basili and Weiss		X	
Lehtinen et al.		X	

Table 4.3: Places where the literature says suggestions for process improvement may be documented or in what stage it should be done.

in the future. This section will discuss if the questions 11 and 12 should be removed from the form by looking at process models in the literature. It is found that question 11 should remain but question 12 shouldn't. Then, other ways of collecting suggestions for process improvement are discussed.

Harjani and Queille describe a process model for maintenance of software for space systems software in [17]. It contains examples on how it could be applied in the case of corrective maintenance. It uses root cause analysis as an example of a thing that could be in the localization step (which is its equivalent to the debugging stage of this company). The safety impact analysis is done two steps later. However, it does not have any example that looks like question 12 in any part of the process.

Basili and Weiss [4] describe a method for understanding a software process. In part III, they present a form that both has impact analysis questions and questions about when the error was introduced. They suggest that an interview is conducted with the one who has found the problem as soon as possible if a possible problem with the process is found.

The first part of a paper by Lehtinen et al. [21] includes a literature study of root cause analysis techniques. It suggests that root cause analysis could be done in a questionnaire. It has also looked at how different root cause analysis approaches do "corrective action innovation". This is similar to question 12. Among the reviewed methods, none use questionnaires for corrective action innovation. This supports that question 12 should not be in the form. Table 4.2 and 4.3 depicts where the three papers suggests that root cause analysis and suggestions for process improvement may be documented.

Below, some suggestions to the company are listed and it is evaluated if they can be considered *almost free* or *cheap* as defined in section 4.7.

It seems to be common practice to have root cause analysis questions in a form. Keeping the root cause analysis question is almost free, since the root cause analysis task can be done simultaneously as the mandatory impact analysis task: whenever the developer has time. Therefore the author recommends that the company keeps question 11 in the form.

There might be problems with the quality of question 11 and 12 that have not yet been observed. Other practitioners have found that answers to these kinds of questions have to be verified to be valid. In the third section of [4] forms are filled in by developers to find problematic parts of the process. They found that half of the forms contained errors in one or more questions. The question of where the bug had been introduced had many errors, because the developer was influenced by the environment. Conclusions drawn from that data would be misleading. This could be avoided if the developer was interviewed when the data in the form seemed strange. Such interviews needed to be conducted soon after the form was filled in. Otherwise the developer would have forgotten too much about the case to fix the problem.

A process described Basili and Weiss in [4] has a stage named "collect and validate data". They suggest that one should "Integrate data collection and validation procedures into the configuration control process" to ensure that "[...] data collection is unobtrusive, and collection and validation become a part of the normal development procedures". If the terms of the company of this thesis are used, the stage can be described as below. When the CCB receives the form they try to figure out if it indicates that there is a problem with the process. As soon as possible, the developer who filled in the form should be interviewed by the process team to try to find the root cause of the problem and possibly a solution to this problem. This cannot be done for almost free, since there are currently no natural meeting between the developers and the process team. It can neither be cheap since it is too sporadic to fit in a fix schedule.

In the literature study of [21] it is found that suggestions for process improvement "[...] usually are developed in a meeting". If this activity was made a part of the lessons learned meeting at the end of the versions lifetime, it could be made for almost free, except for preparation of the forms. The drawback of this solution is that the developer might have forgotten the details of the case since it might be over a year between when he has worked with the case and the lessons learned session.

There is a way to compromise between the two above solutions. All developers who have written an impact analysis that indicates problems in the process meet regularly to discuss possible solutions to the problems. The meetings should be scheduled at least twice every year to avoid having too much time between the oldest root cause analysis and the meeting. This solution will only be cheap if the meetings can be kept short, aren't too frequent and don't have too many participants. It might be hard to keep the meetings short since process improvement is a sensitive subject at the company, see section 3.5.4.

Based on the argumentation above, it is suggested that question 11 should remain in the form, but question 12 should not. Instead of question 12 there should be meetings scheduled at regular intervals with all who have written an impact analysis.

4.4 Communication

This section will present different ways to improve the communication between the developers and the process team. The solutions will be evaluated with respect to how much extra work they will imply for the process team, how freely the developers will be able to talk about problems in the process and how thoroughly each suggestion of a change will be discussed. The section concludes that the preferred way to improve the communication is to assign two developers representing the other developers to whom they may present their complaints. They should then anonymously forward the opinions to the process team at regularly scheduled meetings.

Currently the communication between the developers and the process team is mainly the developers pointing out problems with the form (or the process in general) whenever the subject comes up. Since these topics are somewhat infected, the discussion will be held in a tone taking long time to complete. There might be many people around when the subject is raised, e.g. coffee break, education etc. Then, there will be many persons who want to voice their opinions, making the subject taking long time to settle. It also increases the risk of the discussion drifting to neighboring subjects, making it continue indefinitely. Even if this doesn't happen, the decisions regarding safety might be too complex to motivate without preparation. Either way, if the discussions becomes too long, this might be the reason for the discussions being "ended by saying safety", see section 3.5.3.

According to the author, this way of communicating is sub-optimal in two ways. Firstly, the process team will have a hard time giving a satisfactory motivation for the safety indications of each developer's proposal. This might make the developers feel that they aren't listened to, thus further influencing the tone of future discussions. The second problem is that the process team will have to spend longer total time motivating, since if a subject isn't settled they will have to start over explaining each new time the subject comes up.

Repeated education is a common way of promoting new ways of working. The idea is that if you tell people the theoretical benefits of working a certain way. If you do this often the developers will eventually think of that way as natural. You could say that the culture has changed. I define culture as the things the employees think are normal or natural. The process team thinks that education haven't had the desired effect (see section 3.1.6), but there is no hard proof that any of the other techniques presented here will have better effect. Also, it is a cheap way, so any new technique will have to be much more effective to be more efficient. The developers won't be able to express their opinions anonymously using this technique, and neither will there be much room for discussing the opinions. If such a subject is discussed, it might be inefficient due to the large amount of attendants.

Morning stand up meetings is a possible solution. It is an idea borrowed from the agile development techniques. It is a meeting held each morning that everyone in the project attends. The participants are supposed to stand up, to encourage short meetings. This technique will be demand much of the process team's time and will possibly be inconvenient also for the developers. It won't be possible to voice opinions anonymously, but since the meetings are held so often, it will be possible to discuss the problems at an earlier stage. Then the conflicts might be less infected.

The idea of letting a group of developers represent the others comes from a model used for communication between the students and the professor in the courses held at Lund University. At the beginning of the course the students are asked to assign two of them who should represent the other students attending the course. If any student has a complaint or suggestion regarding the course they have the option to talk to ones representing instead of talking directly to the professor. This way they don't take a hypothetical risk that the complaint will somehow affect their grade in the course. The fifteen minutes breaks in the lectures are an efficient opportunity for the ones representing the students to talk to the professor. There are always at least two students representing the other students, so that they can discuss ideas internally before talking to the professor. This also evens out the power balance, since each student has less influence.

A similar model could be used at the company. However the process team might not want to use their coffee break to discuss such things. Instead, the representatives and the process team should meet regularly. This technique allows the developer to thoroughly clarify his thoughts to the representatives. They have time to conclude the different opinions before they tell them to the process team. This will be more efficient than the current way of working. This technique gives a communication channel that is anonymous and thorough for the developers and efficient for the process team. It is therefore suggested that the company implements it.

4.5 A Fully Done Stage

This section motivates that the efficiency of impact analysis long term quality of the answers in the form will be improved if the filled in forms are reviewed a shorter time after it is written.

The problem with too little motivation in question in question 1 and missing answers in question 5a would solve themselves fast if the developers were informed about them fast. Then the developers would just have to adjust how they wrote the answer to question 1 and they would soon remember to check 5a. This can only be achieved if the forms are reviewed at an earlier stage than the end of the project.

Currently, the company reviews all forms right before they are put in the report sent to the certification organ. This will require less work for the reviewers as they only have to schedule one meeting. However, if they find a form with a problem, it will be harder for the developer to fix it since it might be a long time since he had written it. Also, he will be under a lot of pressure, because the project is in a late stage. This might make him do mistakes, and that is a safety risk. Making reviews at a late stage might in itself be safer, because the reviewers will have more information. Therefore it might be less safe to review the forms at an earlier time. To review them both early and late would take approximately twice as much of the reviewers' time and is therefore not a good idea. I believe that it is a safety improvement if the developer remembers the case and is less stressed, even if the review team has less information. Also, if it is known which cases have to be reopened the project completion date will be easier to estimate at an earlier time. The suggested solution is therefore to review the forms of the finished cases at regular, scheduled meetings. After this the cases should be set to a new stage called *fully done* where the form is assumed to be able to be put into the report to the certification organ without further editing. This is similar to how the company used to do it, and I don't know why they stopped, so there might be important factors that are missed here.

4.6 Software Tool Support

This section describes the building of a tool that parsed filled in safety impact analysis forms in the company's database how it solved the problem of missing answers and why it isn't a good solution to the problems.

A prototype tool was built to parse the answers of the forms put in the form database of the company. It was able to detect many filled in forms where question 5a had no answer and the questions 5b to 5i were answered with Not Applicable. The existence of such forms was deemed as very badly by the process team and they were very glad that they had been identified so that they could be fixed. The problematic forms were often written by the same persons, indicating that they had copied old forms, and carried the error with it. This problem would solve itself over time, as the developers would have gotten the cases back to fix them. If the developers got faster feedback of that there was a problem with their forms, the problem would solve itself faster. The tool provided a short term solution to a problem that would have solved itself over time. It is probable that the developers will make other simple mistakes in future versions of the form, which the tool will not be able to find. Instead of writing new tools it is recommended to decrease the feedback time of errors in the forms to the developers, as suggested in section 4.5.

4.7 Cheap and Almost Free

Inventors of new revolutionary process steps often claim that their practice can be added in any process for *almost free* with benefits that easily outweighs the cost. When the practice is put to use, it is often found that there are small *invisible costs* around the practice that makes it more expensive than just the time it takes to execute. Here is a sketch of a framework to decide if a practice can really be done for almost free.

A proposed new practice is associated with a recurring activity. An activity has a schedule, a place, a required set of material, required preparations, a required set of persons and an amount of time to do it. Every time the activity is done requires planning to find a place that has the required material, where the required people are free for the time the activity takes that fits the schedule of the activity. The total cost of the activity will be its frequency times finding a time and place that fits all participants plus preparation of the meeting and the material plus the time it takes for the participants to go to the meeting plus the time the activity takes. There is also an indirect cost because this meeting will complicate the planning of other meetings. An activity is almost free if the time it takes is short, it requires no preparation, it has low frequency and if there is a prior activity that has the same schedule that requires exactly the same personnel and the same or more material.

Often, new activities require that people meet more often than in previous activities. Those activities aren't almost free but could still be considered *cheap* if they have a fix schedule. Since everybody has this activity, it won't interfere with the participants other plans, no planning is needed and the activity can have a fix location. The total cost is the cost of preparing the meeting plus the frequency of the activity times the number of participants times the time it takes for the participants to get there, plus the time the meeting takes. A meeting can be called cheap if it requires no preparation, has an easy and fix schedule, it is short and has few participants.

Chapter 5
Discussion

This section assesses the quality and how reliable the results of this thesis are. It also tries to estimate if the results are applicable in other contexts.

All the findings in this thesis are made by only looking at one setting (the current company), so one should be careful to generalize the findings to other settings. If the findings should be used in other contexts it is important to remember that the product of this company has unusually high safety requirement. Also, the functional requirements will change much more slowly than for a regular product.

The interviews have focused on the developers, the reviewers and the process team because they are the main stakeholders of the form. The management has largely been left out of the investigation since they don't come in direct contact with the form. It is, however, possible that management can provide completely new viewpoints, that haven't been thought of in this thesis. Many of the suggested solutions are directed against the management. Future work should investigate if the management knows about the communication issues between the developers and the process team. The role of management is to make the most out of the available personnel. Maybe they don't know about it, or maybe they do, but they don't think it is a problem. Maybe they even think that the continuing discussion between the developers and the process team produces constructive compromises, and therefore want to keep it.

Another important question regarding management that is left unanswered is why the projects run so late, or rather why management fails to make accurate project cost estimates of the projects. One rather cynical possibility is that they deliberately make the schedule too tight to pressure the developers to work faster, by only focusing on the things that are the most important. Then the developers will not prioritize the form which will affect the quality of the answers. It also makes the impact analysis more risky from a cost perspective, since a tight budget suggests reviewing all the forms at the end of the project. As found above, this will lead to less reliable cost estimation for the forms (we don't know if the form is fully done until the very end of the project), it will cost more to fix the remaining faults (e.g. because the developer don't remember the case as well), and the quality of the answer will be lower (for the same reason).

More interviews have been done with developers than with the process team because the developers were more available when the interviews were made. This may influence the analysis to the developers' advantage. But the best way of improving both the quality and efficiency of the process is to better understand the problems perceived by the developers. The developers have the best image of where the inefficiencies in the process are. They also have the best knowledge of why some questions have low quality. That is more complicated subject than knowing which answers have low quality.

When the interviews in section 3.2.5 there was a certain air, it seemed a bit taboo. This might be because criticizing the form is viewed as something bad. The ones who doesn't fully commit to the importance of the form are in a way "less good" employees than those who do. Therefore, in this setting, saying the older employees disliked filling in the form more, would be equivalent to saying that they were worse than the others. It is of course a good thing that the developers are careful of saying degrading things about their co-workers.

It is the authors opinion that communication problems will be increasingly common in the software engineering field (and others), as the maturity of the companies around the world increases. Tackling this problem may be one of the major fields of management science in the future. Many companies will move away from a process made by the developers against an increasingly mature process. The previous is less theoretic, but it is simpler and it is made for just that company. Managers should monitor and manage this development so that the balance between process maturity and simplicity supports the goals of the organization.

The developers had to think for a long time to come up with any big efficiency problems with the form. The tools they suggested would improve the efficiency of the form very little. This raises the question of why they had asked for a tool. Is it because the important problems seem unsolvable? This might stem from the rather one-sided communication between the developers and the process team, see 3.5.3. Also it is found in section 3.5.4 that the developers were careful when talking about the form. Is asking for a tool a subtle way of protesting? Asking for a tool might be interpreted either as saying that it will make the work with the form more efficient (which is a very neutral statement), but also that the developer don't want to fill in the form because he doesn't like it. The developer then leaves the interpretation to the listener, and thus he makes the request safe.

Yet another possibility is that the developers perceive the problems with impact analysis as bigger than they are because they are connected a bigger problem: their lack of influence over the process. If the developer thinks about the form at the same time as he thinks about his lack of influence, he might associate the bad feelings with the form and exaggerate the importance of the problems it causes. Then the developers will exaggerate a tools' ability to solve problems. In a way the form becomes a symbol of a complex problem where they have little power and the tool becomes a viable solution to that symbolic problem.

Humphrey argues that the purpose of process assessment shouldn't be reporting the problems of the organization to higher management. Then "[...] people will learn that they cannot speak in confidence". [18, p. 38] Since the developers knew that this thesis might be read by the managers, they were more careful than they would be otherwise. This problem has probably influenced the results of this thesis but would be hard to avoid, because the thesis must be freely available to everyone, higher management included.

The methodology used in this thesis inherently gives a picture of the company that is less positive than reality. The stakeholders were specifically asked to think of problems. Notably, when this texts talks about a “conflict” between the developers and the process team isn’t as severe as one could imagine, just by reading this text. If a more quantitative method, such as questionnaires with Likert scales, had been used, the problems mentioned in this thesis might just have been additional comments, and the overall result would be that the developers were generally happy with the way things were.

The literature for this thesis is mostly found using a general search engine that indexes several scientific magazines (scholar.google.com). It ranks the results by how many other articles that reference the current article, but also how well it matches the keywords in the search. The choice of literature in this thesis is heavily biased by this ranking, and hence by the keywords chosen. Much of the literature used in this thesis is quite old (in terms of software engineering research). It is possible that other conclusions would be drawn if using newer papers. However, much of this thesis is about software process models similar to the waterfall model, and this area of research has been rather stable over the last decade, according to the author. Therefore, only few important papers on this subject where the ink is still fresh can be expected to be found.

One of the main subjects of this thesis, the form, is very hard to find literature about using keyword searches in digital databases. The keyword form has an abstract meaning e.g. “a form of” and this abstract form is used much more often than the concrete form. Even in the papers containing concrete forms, the forms are seldom the subject investigated, but rather a part of the method. It is much more common to give the developer a form as part of the research process than to ask the developer about a form he uses in his daily work. For this reason, it doesn’t help much searching for words with similar meaning such as questionnaire, the vast majority still are about questionnaires used as part of the method in a study.

It should be noted that the author tends to favor solutions that solves root cause problems at a high cost over solutions that remedy the symptoms at a lower cost. Decision makers preferring the latter should read these suggestions with a critical eye.

5.1 Future Work

Is requirement engineering really essential when developing this kind of product? Clearly the quality requirements are important since it include safety (absence of critical failure), but in the interviews it is clear that there is little need for the functional requirements during system development. Instead, they have added the requirements at the end, mostly because this is mandatory in order to get certified. What we see is a company who has succeeded building a product in world class in avoiding catastrophic failure. There might be other things that are considered a mandatory part of mature process which are, in fact, optional in some cases.

Finding the bug and implementing a bug fix are two different activities that should be separated, according to most models of debugging, see section 4.2.1. I would really like to see work where this model is evaluated against how debugging is really done in reality, and have found no paper on this yet. It would also be interesting to see if the assumptions about the differences between adaptive and corrective maintenance could be verified.

Future work should investigate why the developers are careful when talking about what they and others think of the form, see section 3.5.4.

Chapter 6
Conclusions

This section contains a summary of the findings from the above chapters. The company wanted a tool that could be used to improve three main areas.

- The quality and correctness of the answers.
- The efficiency when writing the impact analysis form.
- The handling of the form.

This thesis is an investigation of how the correctness, efficiency and handling of the form could be improved. The main research question was finding solutions that had high value/-cost ratio in a long term perspective, where a tool is just one of many possible solutions.

The most severe problem regarding quality was that in some forms was that two answers contained inconsistent information. The answer to one question described how a bug fix could be verified. The answer to another question said that no new test cases had to be written. The developers did this because they thought that the regression test suite would grow too fast if regression tests were added every time a bug was fixed. Another severe problem was that one of the questions sometimes wasn't answered. This happened because of the layout of the form. This problem was remedied by the prototype tool. However, this problem would eventually have resolved itself as the developers learned what question they often missed. A more sustainable solution would be to increase the learning rate of the developers. The question "How could this problem been avoided?" was sometimes answered in an ironic tone, because the developer could not come up with a good answer. No other quality problems were severe.

The most severe problems regarding efficiency were not related directly to the form, but rather to the general task of doing impact analysis. It was hard to find files (code, requirements, tests, etc.) in the old parts of the system. The company does currently try to improve this, so this problem has not been investigated thoroughly. The question "How could this problem been avoided?" took long time to fill in, according to the developers. Other than this, no severe efficiency problem that could be directly related to filling in the form was found. Therefore it was investigated if there could be another reason for the developers to dislike the form. It was found that talking about who liked and who disliked the form was a sensitive subject. It is unclear why it is sensitive.

The developers sometimes implemented bug fixes before that had been allowed by the CCB. This is inconsistent with many models of the impact analysis process. However, it is found that it is economically motivated to do so in correcting maintenance. This suggests a reevaluation of the current impact analysis process models from a corrective maintenance perspective. It currently takes long time between the forms being written and being reviewed. If the developers got faster feedback on their way of writing the forms, the quality of the forms would increase faster. The developers would also be able to work more efficiently and it would improve the safety of the process, since they will remember the case better. This will also make it easier for them to explain their answers. [4] The developers currently work with many impact analyses at the same time. The developers use copies of old forms when they should fill in new ones. This is a safety problem, and errors in the old form will propagate to the new one. No solution is found to this problem.

A tool was built that solved one of the severe problems temporarily. The authors suggested solution is that the company reviews and updates their policies on how to balance efficiency of production and the safety of the process. Meanwhile they should introduce a system with representatives to whom the developers may express their opinions anonymously. No example has been found in the root cause analysis literature where just one person should suggest improvements to the process. Therefore the question "How could this problem been avoided?" should be removed. The company should introduce a stage for the form in which it is completely done in the sense that it don't have to be reviewed again when it is put in the report sent to the third part certification organ. Then the forms shorter time after they have been written. Then the developers will learn from their mistakes faster and they will have the case in fresher memory, which is better from an efficiency and safety perspective.

Index

Almost free, 41, 45

Case, 13
CCB, 15
Certification organ, 12
Change/Configuration Control Board, 15
Cheap, 41, 45
Customers, 11

Debugging stage, 13

Emergency code, 12
Environment, 13

Failure report, 13
Form, 7, 13
Fully done stage, 44

Impact analysis, 12
Implementation stage, 15
Initial purpose, 7
Investigation Lines, 8
Invisible costs, 45

Method, 9

Process, 7, 12
Process team, 12
Product, 7
Purpose, 8

Quality, efficiency and handling, 8

Reliability, 12
Report, 15
Research question, 8
Review team, 15

Safe state, 12
Safety, 12
Safety critical, 7, 12
Safety impact analysis, 13
Safety impact analysis form, 13
Safety impact analysis report, 15
Safety integrity level, 12
SIL, 12
Site, 12
Software change safety impact analysis, 7
Stages of a case, 13
System, 11

Target audience, 8
Testing stage, 15
Third party certification organ, 12
Traceability matrix, 13

Bibliography

[1] Marcos K. Aguilera et al. “Performance Debugging for Distributed Systems of Black Boxes”. In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. SOSP ’03. New York, NY, USA: ACM, 2003, pp. 74–89. ISBN: 1-58113-757-5. DOI: 10.1145/945445.945454. URL: http://doi.acm.org/10.1145/945445.945454.

[2] K. Araki, Z. Furukawa, and Jingde Cheng. “A general framework for debugging”. In: *Software, IEEE* 8.3 (1991), pp. 14–20. ISSN: 0740-7459. DOI: 10.1109/52.88939.

[3] Victor R. Basili et al. “Lessons Learned from 25 Years of Process Improvement: The Rise and Fall of the NASA Software Engineering Laboratory”. In: *Proceedings of the 24th International Conference on Software Engineering*. ICSE ’02. New York, NY, USA: ACM, 2002, pp. 69–79. ISBN: 1-58113-472-X. DOI: 10.1145/581339.581351. URL: http://doi.acm.org/10.1145/581339.581351.

[4] V.R. Basili and D.M. Weiss. “A Methodology for Collecting Valid Software Engineering Data”. In: *Software Engineering, IEEE Transactions on* SE-10.6 (1984), pp. 728–738. ISSN: 0098-5589. DOI: 10.1109/TSE.1984.5010301.

[5] Peter C. Bates and Jack C. Wileden. “High-level debugging of distributed systems: The behavioral abstraction approach”. In: *Journal of Systems and Software* 3.4 (1983), pp. 255–264. ISSN: 0164-1212. DOI: http://dx.doi.org/10.1016/0164-1212(83)90011-0. URL: http://www.sciencedirect.com/science/article/pii/0164121283900110.

[6] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change*. 2nd. Addison-Wesley Professional, 2004. ISBN: 0321278658.

[7] Kent Beck et al. *Manifesto for Agile Software Development*. June 2013. URL: http://www.agilemanifesto.org.

[8] B. Boehm and R. Turner. “Management challenges to implementing agile processes in traditional development organizations”. In: *Software, IEEE* 22.5 (2005), pp. 30–39. ISSN: 0740-7459. DOI: 10.1109/MS.2005.129.

[9] B. Boehm and R. Turner. “Using risk to balance agile and plan-driven methods”. In: *Computer* 36.6 (2003), pp. 57–66. ISSN: 0018-9162. DOI: `10.1109/MC.2003.1204376`.

[10] Barry W. Boehm. *Software Engineering Economics*. 1st. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1981. ISBN: 0138221227.

[11] J. Bowen and V. Stavridou. “Safety-critical systems, formal methods and standards”. In: *Software Engineering Journal* 8.4 (1993), pp. 189–209. ISSN: 0268-6961.

[12] L.C. Briand et al. “A change analysis process to characterize software maintenance projects”. In: *Software Maintenance, 1994. Proceedings., International Conference on*. 1994, pp. 38–49. DOI: `10.1109/ICSM.1994.336791`.

[13] Jr. Brooks F.P. “No Silver Bullet Essence and Accidents of Software Engineering”. In: *Computer* 20.4 (Apr. 1987), pp. 10–19. ISSN: 0018-9162. DOI: `10.1109/MC.1987.1663532`.

[14] R.W. Butler and George B. Finelli. “The infeasibility of quantifying the reliability of life-critical real-time software”. In: *Software Engineering, IEEE Transactions on* 19.1 (1993), pp. 3–12. ISSN: 0098-5589. DOI: `10.1109/32.210303`.

[15] Marc Eisenstadt. “My Hairiest Bug War Stories”. In: *Commun. ACM* 40.4 (Apr. 1997), pp. 30–37. ISSN: 0001-0782. DOI: `10.1145/248448.248456`. URL: `http://doi.acm.org/10.1145/248448.248456`.

[16] V. L. Hamilton and M. L. Beeby. “Issues of traceability in integrating tools”. In: *Tools and Techniques for Maintaining Traceability During Design, IEE Colloquium on*. 1991, pp. 4/1–4/3.

[17] D.-R. Harjani and J.-P. Queille. “A process model for the maintenance of large space systems software”. In: *Software Maintenance, 1992. Proceerdings., Conference on*. 1992, pp. 127–136. DOI: `10.1109/ICSM.1992.242550`.

[18] Watts S. Humphrey. *Managing the Software Process*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1989. ISBN: 0-201-18095-2.

[19] Samuel T. King, George W. Dunlap, and Peter M. Chen. “Debugging operating systems with time-traveling virtual machines”. In: *Proceedings of the 2005 USENIX Annual Technical Conference*. USENIX ’05. 2005, pp. 1–15.

[20] T.J. LeBlanc and J.M. Mellor-Crummey. “Debugging Parallel Programs with Instant Replay”. In: *Computers, IEEE Transactions on* C-36.4 (1987), pp. 471–482. ISSN: 0018-9340. DOI: `10.1109/TC.1987.1676929`.

[21] Timo O.A. Lehtinen, Mika V. Mäntylä, and Jari Vanhanen. “Development and evaluation of a lightweight root cause analysis method (ARCA method) Field studies at four software companies”. In: *Information and Software Technology* 53.10 (2011), pp. 1045–1061. ISSN: 0950-5849. DOI: `http://dx.doi.org/10.1016/j.infsof.2011.05.005`. URL: `http://www.sciencedirect.com/science/article/pii/S0950584911001212`.

[22] Charles E. McDowell and David P. Helmbold. “Debugging Concurrent Programs”. In: *ACM Comput. Surv.* 21.4 (Dec. 1989), pp. 593–622. ISSN: 0360-0300. DOI: `10.1145/76894.76897`. URL: `http://doi.acm.org/10.1145/76894.76897`.

[23] Frank McGarry et al. *Software Process Improvement in the NASA Software Engineering Laboratory*. Technical Report. Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst, 1994.

[24] I. Rus and M. Lindvall. “Knowledge management in software engineering”. In: *Software, IEEE* 19.3 (2002), pp. 26–38. ISSN: 0740-7459. DOI: `10.1109/MS.2002.1003450`.

[25] Sergio Cozzetti B. de Souza, Nicolas Anquetil, and Káthia M. de Oliveira. “A Study of the Documentation Essential to Software Maintenance”. In: *Proceedings of the 23rd Annual International Conference on Design of Communication: Documenting & Designing for Pervasive Information*. SIGDOC ’05. New York, NY, USA: ACM, 2005, pp. 68–75. ISBN: 1-59593-175-9. DOI: `10.1145/1085313.1085331`. URL: `http://doi.acm.org/10.1145/1085313.1085331`.

[26] E. Burton Swanson. “The Dimensions of Maintenance”. In: *Proceedings of the 2Nd International Conference on Software Engineering*. ICSE ’76. Los Alamitos, CA, USA: IEEE Computer Society Press, 1976, pp. 492–497. URL: `http://dl.acm.org/citation.cfm?id=800253.807723`.

[27] S. A. Thomas, S. F. Hurley, and D.J. Barnes. “Looking for the human factors in software quality management”. In: *Software Engineering: Education and Practice, 1996. Proceedings. International Conference*. 1996, pp. 474–480. DOI: `10.1109/SEEP.1996.534036`.

[28] Matti Vuori. *Agile Development of Safety-Critical Software*. Laitosraportti - TUT Publication series. Tampere University of Technology, Department of Software Systems, 2011.

[29] Dirk Wilking, Umar Farooq Kahn, and Stefan Kowalewski. “An Empirical Evaluation of Refactoring.” In: *e-Informatica* 1.1 (2007), pp. 27–42.

Appendix A

The Form

Company secrets within parentheses and sub-questions are replaced with dots (...).

```
1)  Q: Is the reported problem Safety Critical?

    A:

2)  Q: Describe how the problem shall be solved.

    A:

3)  Q: In which versions/revisions does this problem exist?

    A:

4)  Q: List modified code files/modules and their SIL classifications, and/or
affected safety related hardware modules.

    A: a. Directly affected:
       b. Indirectly affected:

5)  Q: How are general system functions and properties affected by the change?

    A: a. Design according to ...................................................:
       b. ...............:
       c. ...:
       d. ............:
       e. ...:
```

```
       f. ..................................:
       g. ..................:
       h. ................:
       i. Any other (if applicable):

6)  Q: SW: Which Library Items are affected by the change?

    A: a. ..................:
       b. .......................:
       c. ........:
       d. ............:

7)  Q: Which documents, ids & titles, need to be modified?
       (.........................................................................
............)

    A:

8)  Q: Which test cases need to be executed?
       (e.g. ................................................................
..........)

    A: a. Directly affected:
       b. Indirectly affected:

9)  Q: Which user documents, including ..........., need to be modified?

    A:

10) Q: How long will it take to correct the problem, and verify the correction?

    A:

11) Q: What is the root cause of this problem?

    A:

12) Q: How could this problem been avoided?

    A:

13) Q: Which requirements and functions need to be retested by .......?

    A: a. Directly affected:
       b. Indirectly affected (based on architecture):
```