Using Already Existing Data to Answer Questions Asked During Software Change

Daniel Jigin
daniel.jigin@gmail.com

Oscar Gunnesson
ama11ogu@student.lu.se

June 14, 2018

Master's thesis work carried out at Praqma, Malmö.

Supervisors: Lars Bendix, lars.bendix@cs.lth.se
Christian Pendleton, cpe@praqma.net

Examiner: Boris Magnusson, boris.magnusson@cs.lth.se

Abstract

The software development business is always adapting new technologies while trying to keep up with the market and the importance of having the best-suited tools, people and processes is clear. There is a need to understand what needs there might be, so that the configuration managers can supply developers with the correct support to increase the development efficiency. Studies of how developers spend their time have shown that they spend as much time searching for whom to contact in the organization to get answers to their questions, as they do getting the job done.

In traditional software development, configuration managers used to bring a status report about the software to the managers. We have suggested a more modern approach, fit for an agile methodology, where the status of the software is available for any worker at any time. When asking developers how they find answers they say that it is based on gut feeling coming from previous experiences. Providing data will lead to discussions about decisions being data based rather than gut feeling based .

We have investigated CodeScene, a tool that utilizes the version control system GIT to analyze which components that have been committed together and analyzing how it can be utilized when performing an impact analysis and how it can provide the technical debt for a software project. By combining theories from impact analysis and technical debt we have outlined what CodeScene can and can not do. The study resulted in a proof of concept tool that extract the data of interest from CodeScene and combines it with additional data that CodeScene lacks. Three end users tried the tool and found the presented data highly reasonable. One end user said *"If this was wrapped in a nicer user interface and promoted as a agile impact tool then you could make some serious money from it"*.

Keywords: Change Process, Mining Software Repositories, Modern Configuration Status Accounting, Information Needs, Software Tools, Impact Analysis, Technical Debt

DEPARTMENT OF COMUPTER SCIENCE | LUNDS TEKNISKA HÖGSKOLA | PRESENTATION DAY 2018-05-31

MASTER'S THESIS Using Already Existing Data to Answer Questions Asked During Software Change
STUDENTS Daniel Jigin & Oscar Gunnesson
SUPERVISORS Lars Bendix (LTH) & Christian Pendleton (PRAQMA)
EXAMINATOR Boris Magnusson (LTH)

Answering Developer Questions During Software Change

POPULAR SCIENCE SUMMARY **Daniel Jigin & Oscar Gunnesson**

Providing answers to questions developers have is important to avoid frustration and bad decisions. Today, questions about changes to the software are answered by gut feeling and by asking other people; we want to make these decisions data-based. We have investigated tools utilizing already existing information to know what questions we can answer today while not having to introduce new frameworks for project data structuring.

Studies of how developers spend their time have shown that they spend as much time searching for whom to contact in the organization to get answers to their questions, as they do getting the job done. But what is it that developers need and what do they feel that they are lacking and how can we help?

In traditional software development, configuration managers used to bring a status report about the software to the managers. We have suggested a more modern approach, fit for an agile methodology, where the status of the software is available for any worker at any time. When asking developers how they find answers they say that it is based on gut feeling coming from previous experiences. Adding data to the gut feeling would lead to more informed answers.

In our thesis, we have conducted interviews with developers that collected 57 questions. We have focused on questions concerning impact analysis and technical debt, although we found that there were questions concerning testing and implementation as well. Impact analysis is the process of identifying potential consequences of a change, such as finding dependencies in the source code. Technical debt (TD) is a metaphor for describing the cost of taking short-cuts in development. Values on TD can be used to indicate the cost and improve the gut feeling of estimations.

We were introduced to a tool called CodeScene that collects existing data in software repositories and presents metrics like dependencies between configuration items and different code metrics. To know what data that is available today we have investigated and explored CodeScene. We have shown that tools can provide us with all data needed to perform an impact analysis, it is just not structured in a suitable way. We found that being able to select a file to get a set of all dependent files and include TD metrics for each file will help solidify the gut feeling developers have.

To validate the answers to the questions, a proof-of-concept tool was implemented with the purpose of showing that, if the data was presented in a suitable way, it could be used as an impact analysis tool. Furthermore, adding TD metrics of the affected files in the impact set will enable a data-driven discussion of effort estimation.

Acknowledgements

To begin with, we would like to express our gratitude to our supervisor Lars Bendix for his guidance and positive energy through out the whole thesis. His insights into what we were doing were extremely valuable. A big thanks to Praqma and Christian Pendleton for providing the opportunity for us to do our thesis at their company. It has been a pleasure writing our thesis at their office in Malmö. A special thanks to Thierry Lacour, who has been willing to answer all of our questions and for his friendly banter to cheer us on. We also want to say a big thanks to all of the employees at Praqma that have been very welcoming and contributed to fun and interesting discussions. We would also like to thank Adam Tornhill for providing us with licences to access CodeScene.

Lund, May 2018

Daniel Jigin & Oscar Gunnesson

Contents

Chapter 1
Introduction

Every day, decisions are being made by people involved in the software development process to make sure the software is evolving. Decisions can be anything from how the source code should evolve to how the continuous release pipeline should work, some decisions more important than others. The people central to developing software all have their own concerns and areas of expertise. They might be developers, testers, system architects, product managers/product owners and even configuration managers. Developers might have more questions concerning how to implement something while a product manager would like to know how he can prioritize the workload. More than often one person fill more than one of these roles. Not knowing how to find the answers to the questions make them time-consuming. Previous studies of how developers spend their time have shown that they spend as much time searching for whom to contact in the organization to get answers to their questions, as they do getting the job done.

But what is it that developers need and what do they feel that they are lacking and how can we help? To gain insight into this we wish to conduct a study to understand and map the questions developers have. From this, we believe that most of the concerns will be regarding reducing the costs and minimize the pains, e.g making things easier. For example, knowing the overall health of a project and being able to make good effort estimations are desired. Also knowing about certain design flaws and hard pieces in the code-base are sought to make better decisions on when to work with it and to whom the work should be assigned. Having correct effort estimations will help the decision making when planning a development iteration. Therefore there is a need to find out what it is that affects the costs. When asking developers how they do their current effort estimations it is common to get the answer that they base their estimations on gut feeling, which they have required from similar changes and experiences. To find out questions that are asked in software development we will work with the research question:

- **RQ1. What questions are asked during the software development (change) process?**

After having a set of questions that developers want answers to we will explore how they can be answered. Having the answers available with the help of a tool would enable the workers to spend more time implementing functionality or fixing bugs rather than to search for answers.

To find answers to the elicited questions from RQ1 we will conduct research within configuration status accounting (CSA). CSA is the recording and reporting of data about and related to configuration items. Having good status accounting in a project ensures that the development of the project is working on the correct thing. The problem here is that one needs to establish a good way of presenting the information. If it is presented in the wrong way, it will be neglected and therefore a waste of time. Because we also want to have fast feedback we can't have a CSA process that takes to much time; we would like the information to be presented on demand. There are several ways in which one can find data to answer the questions. In this thesis, we will focus on what can be retrieved from historical data, data that already exists in the system. To do this it is possible to get help from the field of Mining Software Repositories (MSR). MSR is the technique to analyze already available data to find interesting information and connections. This data can be found in almost every part of the software project and it can be hard to find good data that one is able to act upon. But only showing the data in its raw format does not help most people. Just because the data is there does not mean it is useful and there is a need to investigate how to turn data into information in software projects. There is a need to investigate how to turn the collected data into useful information that can be presented to gain knowledge about the project. To do this we will investigate what tools can or can not provide answers to the questions. We will therefore work with the research question:

- **RQ2. Which tools can provide the right data needed to answer the questions and how is it provided?**

We have an interest in CodeScene, a tool that mines data in software repositories and presents it, and we will mainly focus our investigation efforts on it to see how it can answer the questions. We will also present alternative tools in the case CodeScene does not fulfill the needs.

We want this thesis to be of an exploratory nature because the problem we will investigate have not been studied clearly and we will need to explore different solutions. We will work with the research questions, trying to gather as much information as possible of the problem. At the beginning of the thesis, we will utilize semi-structured interviews with developers to find questions, followed by an extensive analysis of the results to establish the root cause of the questions. Some of the questions will then be selected and worked with and in this thesis, we will present suggestions to how some of them can be answered. Lastly, we wish to experiment with an implementation of a proof-of-concept tool, showing how a specialized tool can be used to answer some of the questions that we will study.

The first limit to our project is that we only have access to Praqma. During the project, Praqma's employees serve as a sounding board for exploring new ideas as well as providing opportunities for conducting interviews with people involved in the software development process. The employees at Praqma has a varied background in software development and were also used for validating our work.

In the following we will in chapter 2 *Background* present a detailed description of the problem, discussion of the method we used, and research that were a basis for this thesis. In chapter 3 *Questions Asked in Software Development* we will present what questions we found while investigating RQ1. In chapter 4 *Investigation of CodeScene & Other Tools* we present what we found when investigating the tools and how we did it. In chapter 5 *Prototype* we show how we implemented a tool to serve as a proof-of-concept to how some of the questions can be answered. Finally in chapter 6 *Discussion & Related Work* we discuss our results and how they relate to other people's work before we draw our conclusions in chapter 7 *Conclusion*.

Chapter 2
Background

In this chapter a detailed explanation and motivation of the problem we are trying to solve, and its importance is presented. Then we present and motivate the methodology we have chosen. Furthermore, studies related to this thesis and that lay the basis are presented. We then provide the reader with the required background knowledge to understand the content of this paper. Readers that are well educated in the fields of impact analysis or technical debt could skip section 2.3 and 2.4, but it is not recommended, as our definitions may differ from yours.

2.1 Problem Statement

In the beginning, the idea was to investigate how a company could benefit from introducing the MSR-tool CodeScene [1]. CodeScene is a tool that provides metrics on the software, such as dependencies and complexities, by mining software repositories for information. The company believed that the tool would be of use in their planning stage and would point out the biggest risks and the overall health of the project. They also thought that the tool would help them increase the throughput of stories because a healthier source code is easier to change. Due to difficult circumstances, the company decided to delay the introduction of the tool, erasing our opportunity to be at the company to find out how and if the tool was beneficial. CodeScene had at this point already caught our interest and we, therefore, had to change the thesis' setting, resulting in it being carried out at Praqma. Praqma is a software consulting company that specializes in continuous delivery and has smaller software projects, mostly plug-ins to existing applications.

Because we do not know what questions people involved in software processes have, there is a need to investigate what they are and how they can be answered. Identifying peoples needs will help future researchers and tool developers to know where to put their focus. In the area of configuration status accounting (CSA), Praqma suggested that there is unused potential in analyzing already existing data and information in the version control system (VCS) to improve the software development process. There is a belief that questions, that we will find by working with RQ1 *What questions are asked during the software development (change) process?*, can be answered by making data from the VCS available. Retrieving already existing information and presenting it is what is the main concept of MSR, which can be seen as an input to CSA; MSR provides CSA with data and information.

CSA provides software projects with observability, enabling insight into the status of configuration items. CSA is the recording and reporting of information needed to work on a project. The main goal of CSA is to provide the management with the information they need to carry out their work. Without the information, they might still be able to do it but, it would surely be easier if they were provided with answers to their specific questions. An important part of a CSA system is that it can be tailored to specific needs and produce relevant reports. What CSA does is to take raw data and turn it into information; making order out of chaos. The CSA process is therefore only as good as the data coming in. There are requirements on the data: it is stored and possible to manipulate (so that it is presentable). Another important part of CSA is that it should be easy to use and not hinder the development [8]. Because one key aspect of CSA is that it should be tailored to organizational needs and we today see a more agile market, there is a need on CSA to be faster and more flexible. We have decided to call this modern configuration status accounting (M-CSA) where we want everything to be as fast as possible, automated and easy to use. If a developer needs an answer, he should get the answer on demand from M-CSA and not be bottle-necked by any other person or process. There is also a requirement that the answer should be provided using the latest data, i.e the information should be fresh.

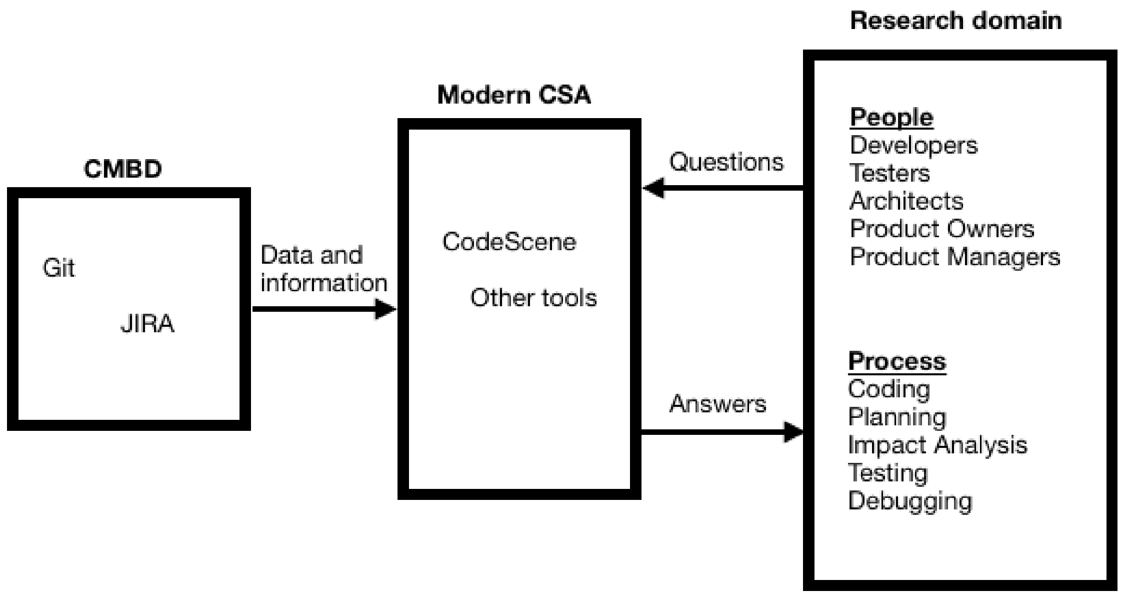


Figure 2.1: A sketch of the problem statement domain

In figure 2.1 we've drawn up a sketch of the problem statement. In the figure, the research domain consists of people with different positions involved in different processes. The questions that these people have collected by RQ1, are fed to the Modern CSA box. The Modern CSA box consists of tools able to provide information about the project. In our case, we will focus on the tool CodeScene. The Modern CSA box will then mine (collect) data and information from the CMDB, process it, and provide the answer to a specific question. The CMDB will in our case be a git repository but could include issue tracking systems such as JIRA. One could even use mail archives and bug tracking systems but that will no be covered in the thesis.

With RQ1 we'll get the questions and with RQ2 *Which tools can provide the right data needed to answer the questions and how is it provided?* we'll fill in the Modern CSA box by investigating tools and motivate why and how they can be used. For RQ1 we will not limit us to any type of question except those outside of the figure, but for RQ2 we will have to focus due to time constraints. The belief is to focus on processes that include dependencies or software health because the introduction of CodeScene showed that it can provide those metrics.

2.2 Method

To give insight into how we addressed the problem, the method for the whole thesis is presented in this section. Here the different options are described and motivated. The outcome of the choices is presented in the chapters 3, 4 and 5, where a more detailed explanation of every step. The method used to answer the research questions specified in chapter 1 was split into three phases, where each phase gave input to the next. The method and motivation of choices inside each phase can be read in their corresponding subsection. The first phase's objective was to investigate and answer RQ1. It was done by first conducting a literature study followed by an interview study that resulted in a list of questions that needed to be answered. The list was too long to analyze every question for RQ2, therefore there was a need to extract the most interesting and relevant. A manageable set of questions were then input to phase two, creating use cases and investigating tools. Phase two began with a literature study, in the domain of technical debt and impact analysis, to find what information is needed to answer the questions. It was followed by a study to find out what data the tools could provide to answer the questions, in conjunction with RQ2. To present the result we decided to create use cases and requirements. In the last phase, we wanted to validate the requirements/use cases produced in phase two. A prototype was created, giving the workers at Praqma the possibility to try out the tool and give their opinions.

2.2.1 Gathering the questions

This phase began with a literature study to gather knowledge about the problem domain, to investigate what has previously been done in the area and what might be missing. It was also meant to inspire us and give us ideas to how questions could be elicited, what might be interesting and what are the kind of areas we need to do interviews within to find questions. As the main purpose of this first step was to answer RQ1, we began the literature study by searching for papers that could be interesting to "What developers ask". Two strategies were used to search for relevant information. The first one was to start with a relevant paper and then look for papers that used it as a reference. The second way was to insert the keywords, describing the problem domain, into databases such as Lub and Google Scholar. Databases containing academic publications was then search using "Google Scholar" combined with those two strategies. Papers with titles that matched our current keywords were saved in a spreadsheet. Only looking at the title was a good first step to sift out the most irrelevant literature, but it still resulted in a fair amount of false positives. The literature that made it into the spreadsheet was therefore ranked based on their relevance to this project. Both authors read the abstract of an article and based on that label it as either relevant or not relevant. The papers could be ranked in three different ways, relevant by both authors, relevant by one and not relevant. Papers which were considered relevant by both authors were carefully read and used as background information[10][12][13][18][22]. Two papers ranked relevant by both were picked out by the academic supervisor to be reviewed. The purpose of doing reviews was to get inspiration for new ideas, get insight into what other researchers have done and to analyze how we could contribute to something new. By finding out what others have missed or could have done better was a good starting point for our thesis. The reviews were done according to Fong's paper on how to conduct a paper review [9]. Relevant parts of the paper reviews can be read in the section literature studies.

With the insight from the literature study, and with respect to our limitations, it was decided that the main source for the collection of questions that developers ask, was to be by conducting interviews. An interview is normally done by a researcher asking questions to a subject in the area of interest. There were three interview structures to consider when creating the interview guide, see Appendix B. The three structures were structured interview, semi-structured interview and unstructured interview [11]. Because the project is of exploratory nature, the interviews were conducted with semi-structured approach, utilizing pre-defined supporting questions to guide the interviews. This allowed the interview to be flexible and to catch as many questions as possible by asking follow-up questions depending on the subjects answers. The interview guide was established by ideas from the literature study, and then improved by an iterative process. An employee at Praqma acted as a test subject over three iterations before the interview guide had a feasible structure and relevance. To find questions in as many areas as possible there were a need to interview subject with different roles and background. The aim was to conduct interviews with developers, testers, architects, project managers and product owners. During the first interview, one of us took notes and the other person were asking questions. When looking through the notes and discussion what we could take with us from the interview we came to the insight that, from now on there was a need to have an audio recording of the interview. The audio recording allowed us to go through the interviews to analyze them and elicit more questions. There were three ways to elicit questions from the interviews:

- **Direct question:** Stated directly by the subject
- **Indirect question:** Underlying question taken from the context
- **Jeopardy question:** Derived from a suggested solution

Doing observations as a complement to interviews were considered but there was no possibility to do it because of what we described in the context section, lacking a good project and developers to observe. Observations are also more time consuming and would not fit in the scope. In an environment where we could follow subjects around and record their daily questions would perhaps have yielded a bigger set of questions and a more in depth study. By doing interviews we were able to get a wider scope and find interesting areas to investigate and gain more broad knowledge. This also helped with finding a wide set of questions, contributing with research needs in other areas than what we chose to focus on. Only doing the literature study and not holding interviews to find more questions was also taken into consideration. If a previous paper already had found interesting questions to us and Praqma, it would have been sufficient to focus on them. The previous studies did, however, show that those papers did not focus on the same context as we wanted.

It was expected that the outcome of the interviews would yield a lot of questions. To get them down to a manageable number it was necessary to rank the questions and only chose those deemed to be the best. It would have been possible to analyze all the questions, but the result would then just have touched the surface of the result. We chose to analyze a few questions more thoroughly instead, as it would result in a more covering answer and also that we could focus our work on fields we found interesting.

2.2.2 Use cases and investigation of tools

A few of the most interesting questions in the domains of impact analysis and technical debt from the previous phase were selected to begin with. To answer RQ2 and find what data we need to have to answer the selected questions there was a need to analyze them. This was done by performing a new literature study, similar to the one carried out in phase one [4][6][7][14]. The literature study would also give us insight into how the data could be presented, to be useful information. It was decided to present the result from this phase as requirements for a tool, to show how a tool can be constructed to provide answers to the questions. The requirements turned out to be a bit too tool specific and not general. Therefore, use cases were used to clarify what problem the tools are supposed to solve. The requirements and use cases are meant to be complementary, the former more detailed and technical and the latter overall more friendly and easier to have developers verify.

Another approach to finding what data that is necessary to answer the questions could have been to conduct a new interview study, where it would be possible to ask what data the workers of Praqma need to answer those questions. It was deemed to be better to take their questions, turn to academic articles and then provide them with the answers.This would give the interview subject new input of how they could go about the process of finding the needed information, and also the questions found were on such an abstract level that they themselves could not pinpoint what could be parts of the solution.

With the knowledge of what data to look for it was then possible to start investigating what information is in the CMDB, which in our thesis mostly were project repositories, and how tools can retrieve it. We began by investigating if CodeScene had the desirable data, as it was our main tool to study. Google was then used to search for other tools to investigate; if CodeScene wasn't able to provide the answers other tools were researched to see if they could be used as a complement. The other tools that was able to complement CodeScene were EclEmma [2], Stan [19] and JDeodorant [3]. While investigating CodeScene we also took notes if it could provide information that could either answer the questions asked from phase one, or if it had functionality that the literature study did not find. If that was the case, then the new functionality was added as either a requirement or a new use case. An exploratory study wasn't done for the other tools, as exploring a tool is very time consuming, often requiring a big on boarding cost. The data found from other tools were only used as a complement to data CodeScene lacked. Both the analysis of the questions and the investigation of tools was done iterative. Each step answered a few more questions or resulted in additional requirement/use case.

2.2.3 Prototype

The last step of this thesis was the prototype, that had the purpose of validating the requirements/use cases and to be a proof-of-concept that the data from the tools could be turned into useful information. It might have been sufficient to let the workers at Praqma validate the requirements and use cases by reading them, but by implementing a tool it would be easier for them to grasp the ideas if they could try them. This would also give input about the feeling of using such a tool in their daily activities and if it would be of any help. A mock-up with a descriptive manual explaining how the prototype should work was first created. That was created to validate our ideas of how the tool could be implemented, and to be used as a requirement specification, before implementing it. The time restrictions of this phase were very limited, so the technical competence needed to implement the prototype had to already be known. A short study of what we could implement was therefore done, and the tools were implemented in small steps starting with the most basic functionality and then building in more features as long as the time allowed. The validation of the tool started with a short introduction of how the tool worked and what it was meant to be used for. After that, the subjects tried the prototype. The subjects were asked to say what they believed were missing, what was good and what they thought about the tool.

2.3 Literature Study

To gain knowledge about and be inspired by what previous researchers have investigated, what conclusions they have presented and what needs more exploration, a literature study was conducted, as described in section 2.2.1. Here we present what others have come up with and how we plan to extend their research.

In a paper by Andrew J. Ko et al. *Information Needs in Collocated Software Development Teams* [12], they did a study aimed at identifying and characterizing developers' information needs and how developers made decisions while working. By identifying these needs they stated that one will get a better understanding of what tools, processes and practices that are needed to improve and what should be the focus, and in the long term improve the overall quality of software. They claimed that by observing 17 developers they would be able to identify common patterns in their information needs. The focus was on recording goal-oriented events. In the study, they presented a list of 21 questions. The most frequently asked questions were also confirmed by a survey, which indicated that the same questions were also the most important. The three most important questions were: (1) What have my coworkers been doing? (2) What code causes this program state? (3) How have resources that I depend on changed? Common to these questions and to most of the other 18 questions, was that most were answered by asking a coworker.

From the paper, it was clear that the most frequently sought information concerned awareness about coworkers and artifacts. They also stated that information concerning design and program behavior was often deferred because the only source of information was unavailable coworkers. This paper contributed with 21 questions developers might have in a certain context and makes no effort in exploring solutions to the problems. A more interesting approach would be to find a problem and then try to provide a solution. This paper functions as a collection of possible problems that further researchers can pick and choose from to either validate or find solutions to. This paper sets the stage for further research where there won't be a need to conduct seventeen 90 minutes observations to find the questions.

Thomas Fritz et al. conducted a similar study as that paper but tried to extend the study by providing answers to the questions [10]. They interviewed developers instead of observing them to find the questions. The interviews resulted in a list of 78 questions, where 51 of them were listed in the report. Fritz used a model to try to answer the questions, a model that was created in a former paper. The model consists of small fragments with information which can be merged together in different ways. The fragments are sorted by their domain and have different information depending on what fragment it is. They let 18 participants try a prototype that used the model to find the answers and let them answer eight of the questions. The participants were able to find the answers in 94% of the occasions [10].

Silito et al. also conducted a study to find questions developers ask during the implementation of a change task [18]. They also investigated if there were support for the questions in existing tools. They found 44 questions which they found during two observation studies, one with laboratory setting and one in an industrial setting. They elicited questions by transcribing audio recording and analyzing the answer. They only included questions regarding the code base. If they found that questions were roughly the same thy created a generic one. They then continued with a presentation of what tools that could answer the questions [18].

All of the papers focused on developers needs in the context of implementation. This thesis will extend those lists by including more positions than developers and also focus more on the decision parts of the change process. In the paper by Andrew J. Ko et al. they observed what the questions were and how they were answered. This thesis has the approach that the relevant questions were the more abstract ones, not having a simple answer and more need a combination of information and context.

2.4 Impact Analysis

In this section, we introduce the reader to the processes of impact analysis. It is important to understand the scenarios and difficulties of an impact analysis presented here to understand the analysis and investigation that are later presented in this paper. Impact analysis is a widely explored research area and has therefore been defined in many different ways by various authors. Bohner and Arnold defined impact analysis as “identifying the potential consequences of a change, or estimating what needs to be modified to accomplish a change” [6]. The definition focuses on where the changes will take place. Previously Shari Pfleeger defined it, with more focus on the cost of a change, as "the evaluation of the many risks associated with the change, including estimates of the effects on resources, effort, and schedule."[15]. This thesis uses the same definition as Bohner and Arnold, but also add the cost aspect after the impact is estimated, as most of the questions found in phase one could be tied to either dependency analysis or cost estimation. The impact analysis process differs depending on what company and person are doing it. A traditional company using the waterfall method might have a very strict change process where each change must be processed through a control change board where it is decided whether or not the change should be implemented. Every possible outcome, and action, of the change, must be estimated before a decision can be made. A company using an agile approach might not be as strict, as they tend to change smaller tasks and get quicker feedback on if they are going in the wrong direction. It might be sufficient for their developers to perform an impact analysis in their minds, just using their gut feeling as a basis. One reason that the process differs might be that impact analysis usually is not a field being taught in engineer schools and the methodology is therefore designed to fit each developer's personal taste [6]. What they all have in common is that an impact analysis is performed, one way or another, before the change is implemented.

To get an overview of how an impact analysis can be conducted we here present how Bohner and Arnold defined the process. An impact analysis starts with a change request that is described by text is given to an expert. The expert is someone who has knowledge about the system that the change request will impact. From that change request the expert identifies a *starting impact set* (SIS). An impact set consists of objects that could be anything from documentation to source code, testing or even methods/variables in a class. The SIS should contain all objects which will directly be impacted by the wanted change. But changing an object might lead to that other objects must be changed. Take a calculator for instance. A change request has been made that wants to change the number representation that the calculator uses. A direct impact of that would be to change the number representation in the source code. But changing that might break some of the operations which use those numbers. That is called a side effect or the *ripple effect*. The second step is therefore to find an *estimated impact set* (EIS), which are the SIS plus the additional changes caused by the ripple effect. Finding those impacts are usually the most troublesome. Having a tool to find those simplifies the process a lot. Lastly, the change is implemented. That will result in an actual impact set (AIS). The AIS contains every change that actually had to be done in order to implement the change request. The AIS could be used to differ with the EIS to find out how good the estimate was. To estimate the extent of the change Bohner and Arnold suggest that one can perform two separate analysis, dependency analysis, and traceability analysis. Both analyses involve finding connections between parts, but on between different configuration items. Dependency analysis focuses on parts within the source code, which could be files, functions, objects etc, whereas traceability analysis focuses on connections between configuration items such as source code, tests, user manuals, requirements etc and where to find them [6].

Dependency analysis can be brought out on different levels of granularity depending on how exactly the expert can predict the SIS. The levels are file-, function- and object-level. Running the analysis on object-level will result in a more specific EIS than running it on file-level. Figure 2.2 shows a scenario that explains why a more fine-grained level will produce a better EIS. Class Foo and Bar will be dependent if the impact analysis is run at class level. But if the change would need to modify something to fooFunc1(), then the class Bar would not have to change. An impact analysis on function level would have given the right answer. A dependency in source code can show up in a few different scenarios. The analysis and investigation in this thesis have taken the following into consideration:

- **Direct dependency:** An example of this is if class A calls a function from another class B. Changing the function in B would likely trigger a change in A.
- **Indirect dependency:** Same example as the one above, but now C is dependent on A. The change in the function in B would still trigger a change in A, but since C is dependent on A it might also trigger a change in C. C is, therefore, indirect dependent on B.

A tool that finds an estimated impact set should be able to handle all those scenarios. Finding the direct dependencies might seem like an easy task, but manually searching for declarations is time-consuming and not desirable. Indirect dependencies tend to grow large because indirect dependencies can also trigger another indirect dependency. Each time a new dependency is triggered it lowers the probability that it actually has to be modified. Therefore it is good to have an indicator stating how far away the dependency is. It should be clarified that if A is dependent on B it does not mean that B is dependent A. It was however found, during the investigation, that one of the tools treated the dependency as they were both dependent. If they are only dependent in one direction we call them uni-directional and if they are both dependent we call it bi-directional.

Dependency analysis can be done by performing a static analysis, dynamic analysis or a historical analysis. Static means that the source code is evaluated based on its source or binary files. This could be done by looking at the build files or by looking at declarations in a file. In dynamic analysis, it executes the program and traces the program flow and thereby find which parts of the program that uses the others. Historical is done by looking at data from the commit history in the VCS. From that data, it is possible to withdraw which files that have been committed together with each other, e.i which files that are dependent.

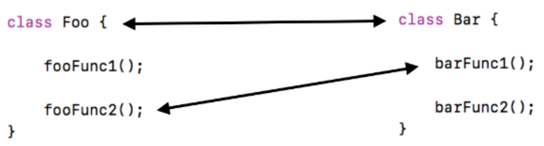


Figure 2.2: Dependencies on class and function level

2.5 Technical Debt

In this section, the reader is provided with an introduction to the term Technical Debt (TD). Later in the thesis, we will focus on TD and it is therefore important to understand. Lately, the metaphor technical debt has become a buzzword in software development and when researching TD to establish a definition to be used in the thesis there were several different papers with different definitions of what should be considered TD. Therefore, this section is important to define what is considered TD in this thesis. This section serves as an introduction to the concept or as quick fresh up.

TD was first coined in a paper by Ward Cunningham where he states that projects build up debt when developers are forced to meet deadlines, requiring them to take short-cuts in development [7]. This debt then incurs interest that later will have to be repaid. The term makes it easier to communicate and talk about consequences, for example, it is easier to tell the customer that if we are implementing this feature this way just to meet the deadline we will take on this amount of TD.

To get a better understanding of TD, Zengyang et al. performed a systematic mapping of the TD field, covering publications between 1992 and 2013 [14]. In this paper, TD is defined as:

> *"Technical Debt is a metaphor reflecting technical compromises that can yield short-term benefit but may hurt the long-term health of a software system."*

In a software project, taking on TD can do both good and harm. The good technical debt can be seen as making an investment, much like taking a loan that will later have to be repaid and that one has to pay interest in the meantime. If the TD is intentionally incurred and it is kept visible and under control, it can help the project obtain some business value, perhaps in the form of a faster release giving them a lead over the competition. This TD is not what should be of concern but rather the unintentional and invisible. If the project developers and management are not aware of the location, how much and possible consequences the TD will increase. Having constantly increasing TD will make future changes much harder, if not impossible [14].

There is a lot that can be considered to be TD, making it easy to classify everything that can make future changes difficult as TD. In the paper, Zengyang et al. state that there exists ten types, namely TD connected to requirements, architecture, design, code, test, build, documentation, infrastructure, versioning, and defects. Out of these, code TD is the most common followed by architectural, test and design. TD connected to code is poorly written code that violates best practices or coding rules. Architectural TD refers to TD being incurred by architecture decisions lowering the internal quality, i.e making the software harder to maintain, by making compromises. TD in testing refers to shortcuts taken in testing, such as not writing a sufficient amount of tests (lack of tests). Design TD they state is incurred by shortcuts in detailed design, such as code smells, which might be very close to Code TD. They also present defects, bugs, and failures, as technical debt [14]. They have found studies that state that defect might not be TD, it all comes down do if one wishes to view something that is unintentional as TD. Only the relevant types of TD have now been presented and to understand the other the reader is referred to [14].

Now that we have a basic understanding of what TD is we need to dig deeper to find out what can be considered as TD. As stated, code smells and best practice violations are both considered to be TD. According to Fowler in his book refactoring some code smells are [4]:

- **Duplicate code:** There is the same code in different places. This is often a result of copy-paste code. Duplicate code results in the software being harder to change due to dependencies between the duplicate code pieces.

- **Long method:** As the name indicates this bad smell is caused by a method being too big. Big methods are hard to understand which is solved by breaking down methods into single responsible and short methods with good names.

- **Large class:** This is a class that has too much responsibility and is too big. To see if a class is bad one can look at the size (lines of code) or the number of instance variables.
- **Long parameter list:** To avoid global data, which is considered evil, one can pass a big amount of parameters to a function. With today's object-oriented programming style this is not needed and it is sufficient to make sure that a method can get the parameters needed by asking an object. A long parameter-list makes the method hard to understand.
- **Divergent change:** When a class or function changes based on different reasons this might be an indicator that it is designed poorly. Often you can solve this by creating an additional class, making both having single purpose.

So what are we to do with TD? Zengyang et al. state that there is a need to manage it and they presents eight different TD management activities: identification, measurement, prioritizing, prevention, monitoring, repayment, representation/documentation, and communication. Out of the activities identification, measuring and repayment have had most attention in the software community. This is because of they each help with the understanding of three important questions: Where is the TD? How much TD do we have? How can we repay it? To identify TD the two most popular methods are code analysis and dependency analysis. When measuring TD the most common ways are to use mathematical formulas or models, use code metrics and human estimations that is based on experience. The most common way to repay TD is by refactoring, making changes without altering the external behavior with the goal to improve quality [14].

Chapter 3

Questions Asked in Software Development

During this chapter the term "Question" is used with two different meanings, one we call *interview questions* which is the questions asked during the interview process and the other one which is only mentioned as *questions* are the resulting questions, i.e the needs in software development.

This chapter starts by explaining how the interview format was produced, how the interview questions were formulated and what literature they were built on. We then present in detail how the interviews were conducted to give other researchers the possibility to perform a similar study. The interviews had the purpose to find as many questions as possible and to give insight into what areas there were a need to investigate further. We present all the questions found, including the ones we have not worked on to provide input to further research. To facilitate the work of future researchers the questions are presented in categories that are based on which research field that could be applied to give an answer to the questions. Some of the questions are provided with a context to eliminate the risk of miss-interpretation and to explain the questions that are hard to interpret on their own. The questions are also analyzed by prioritizing them with respect to this thesis. This was done to make it possible to pick the most relevant questions based on the scope mentioned in section 2.1. After we have presented the questions in Elicited Questions, section 3.2, we have answered *RQ1 What questions are asked during the software development change process?* Finally, we present how the questions were used to create requirements and use cases. The requirements were created to describe what data a tool need to provide and the use cases were created to explain how and why the functionality is needed. They would also be used as input to *RQ2 Which tools can provide the right data needed to answer the questions and how is it provided?* and therefore, section 3.3, must be read to understand why the investigation of CodeScene, described in the next chapter, was conducted like it was To find the requirements and use cases, impact analysis and technical debt were studied to find theories, described in section 2.4 and 2.5, that can be used to answer some of the questions.

3.1 Interviews

In this section, the groundwork needed before the interviews are described and motivated. The literature, the processes and the experiences that helped produce an interview guide are presented. As stated in the method, section 2.2.1 the interviews used a semi-structured approach to achieve the breadth needed to cover the scope. Therefore, an interview guide, with some helping questions, was created. It was also good to have questions to inspire the interviewee if he didn't have anything to say. The goal of the interviews was to define questions that arise from employees during a change process in software development, that would provide insight into what questions (information needs) that people would like to have answers to, and thereby answer RQ1 *What questions are asked during the software development (change) process?* The hopes were that by asking general and broad questions the interview subjects would tell us what problems they have, what they struggle with daily or what would be nice to have (improving their daily work). In the interviews, we were looking for both questions that they solve with ease and questions that they have no suggested solution to. By finding and presenting questions with a known solution, we would help others struggling with those questions and by finding questions that they can not answer we would know where we should focus our future efforts. As stated, the interviews aimed at finding questions that are connected to a change; everything from a story's planning phase to implementing and testing was discussed to find questions.

In the section 2.3, literature study, the conclusion were that a lot of questions developers have are solved by asking co-workers. Therefore, a lot of questions in the interview were concerning what data the interviewees needed and how they thought that having that data transformed into information would benefit them. One thing that all the papers we read had in common were that they focused on finding implementations specific questions and that they in their future work section stated that the study could be extended by including more parts of the software process and more roles. From this we decided to not focus on implementation specific questions but on questions about a change that arise before it is carried out. From the related papers we found that a lot of questions concerned dependencies and knowledge about who is working on what. We also found that there were a need to understand how hard a problem is to fix, i.e making estimates.

The literature study together with the first interview made it clear that it was hard to explain the context and what kind of questions we were looking for. This led to that the first part of the interview was to give an introduction where we explain what we meant by the change process and try to figure out if and how the interviewee has been part of that sometime.

The process of creating the final interview guide was that first inspiration from literature studies was used to set up a short list of questions. This list was then extended by trying out some questions on a test subject and discuss what was missing. Discussing the questions with the company supervisor we got input into what's of interest to him and what areas he believed to have unanswered questions. The final version of the interview guide can be read in appendix B. The interview questions found to generate the most questions were:

1. Can you think of a problem when you are executing a change task which you don't know how to solve?
2. How do you plan stories (change tasks)?
3. What information do you need when you implement a change request?
4. What information do you need about tests?

When asked about impact analysis there were almost no answers, but as we'll see in the next section, the most questions can be connected to and solved by impact analysis. There were in total four different interviews with employees at Praqma. Everyone that was interviewed had a background as developer but can be seen as a software generic person, meaning that they have knowledge about most of the parts of software development, including product manager and product owner roles. That allowed us to find questions very central to the planning of change tasks.

3.2 Elicited Questions

This section contains the resulting questions from the interviews. First, we present how the questions were analyzed, i.e how they were put into a category and given a priority. The categorized questions are then presented in sections of its own, each containing a table that gives an overview of the questions. In each section, some relevant questions are further described. The description of the questions is based on discussions in the interviews and in what context they were brought up.

Eliciting questions from the interviews, by using the direct, indirect, and jeopardy method described in section 2.2.1, resulted in a set of *70 questions*. A first analysis had to be done to the questions because some of them were duplicates, they addressed the same problem or needed the same data and information to be answered. For example, the questions *Is this story a must have?* and *Is this story a should have?* were both addressing the same problem, making it possible to transform them into one question as *What is the priority of this question?*.

When categorizing the questions, three different approaches were tried out in the thesis. First, we tried to categorize them from the aspect of who (developer, tester, PO, PM) that was asking the questions. This turned out to be not applicable as the classical roles are not well defined in an agile environment. A developer can for example also have the role as a designer or tester, making it hard to put a question into a single category. Next try was to categorize each question on what configuration item that could provide the answer to the questions, e.g. which document do we need to mine for data to answer it, similar to the one Thomas Fritz used to present their results [10], i.e categorizing with respect to e.g source code, story, or design document. Using this approach made it easy to divide the questions into their corresponding category, but the solution to the questions within a category was not possible to be found in a single research field. In the later part of the thesis a few questions were analyzed by studying literature about what information they need to be answered. There was a chance that when the literature was studied it would also be possible to find solutions to other questions, not chosen to be analyzed, and those other questions would also be noted. By using that categorization it meant that all questions had to be kept in mind while reading, and not only those concerning the material that was currently studied. Therefore, the questions ended up being categorized according to what research field they belonged to. This made it easier to work with finding suitable theories to answer the questions because by investigating one questions in one category gave answers to other questions in the same category, because of them being closely related. We ended up choosing the categories impact analysis, traceability, testing, and technical debt as those were the research fields that we had basic knowledge about and they are big research areas within computer science. The questions that did not fit in a category were put in a new category called other.

After running the initial 70 questions through the first analysis they became 57, where 15 were impact analysis, 8 traceability, 9 test, 6 technical debt, and 19 in the other category. For each category we created a table, tables 3.1, 3.2, 3.3, 3.4 and 3.5. This was done to make it easy to get a good overview of the questions and what research field that can be applied to answer them. Some of the questions were complex and could fit into different categories. These were put into the category best suited to answer the questions.

Not all questions elicited from the interviews were of interest. This was a result of the interviews being set up to gather an as broad variety of questions as possible. This led the questions varying in relevance to the last phases and therefore required new analysis to sift out the less relevant. That was done by coloring each question either green, yellow or red, originating from it being the chosen method when working with the questions on an excel sheet. How we prioritized the questions is described here:

Green means that a question is deemed to be relevant to the thesis. For a question to be relevant they were required to: (1) It should be possible to use data from the CMDB to provide the answer, (2) there is a possibility (gut feeling) that CodeScene can provide the needed information, and (3) be of interest to us or/and Praqma.

Yellow indicates the same as green color. However, since we have limited time for the thesis we will not investigate these any further. Reasons for not investigating any further might be that we believe that analyzing the question will not yield any benefits. This might be because the answer is very trivial or that there already exist very good practices that solve the problem.

Red means that the question did not make the criteria to be categorized as green or yellow. All questions that are a result of a persons lacking in knowledge, "know-how", is considered red. An example: "How can I test what I implemented?" is a result of lacking knowledge about testing. Company guidelines could be found within the CMDB but will not be considered in this thesis and therefore labeled red.

In the following sections, all the tables are presented with the resulting questions. For each question, the table includes what priority the question was given, how it was found from analyzing the interviews, and if we describe it in this thesis. The last column in each table named Analyzed states whether or not we do further work on it, this is done so that the reader can look at the tables to see if a question he is interested in will be described. We have done an analysis of all of the questions but chosen to omit some because we believe that discussing them would lead to the thesis being too long and unfocused to read. We had the time to do the analysis but not the space to present everything.

3.2.1 Impact analysis category

Here we present the questions that were put in the impact analysis category. Questions that concern dependency analysis or cost/time estimations to a change request before it has been carried out was put in the Impact analysis category. A question that was put into this category will at least be of yellow priority or higher because of the fact that IA is based on finding dependencies between configuration items (CI) and that can be mined from the CMDB. The area IA is also of interest to Praqma and one of CodeScene's main function is to find dependencies. Table 3.1 contains the questions that are connected to impact analysis. There are no questions with priority *R* because impact analysis is central in this thesis' scope.

Table 3.1: Elicited questions categorized as impact analysis related.

ID	Question	Priority	Extraction method	Analyzed
1.1	What parts of the source code will this change impact?	G	Implicit	X
1.2	Where is the least risk of breaking anything?	G	Direct	X
1.3	What applications are using my library?	G	Direct	
1.4	Who is dependent on this code?	G	Direct	
1.5	Will this change break backwards compatibility?	Y	Direct	
1.6	How can I implement this story?	Y	Direct	X
1.7	Is this the right way to solve the problem?	Y	Implicit	
1.8	Could this be implemented in a better way?	Y	Implicit	
1.9	How long time will it take to complete this story?	G	Implicit	X
1.10	Can we remove this CI?	Y	Implicit	
1.11	How can we split up repositories?	G	Direct	
1.12	What other stories can be done while working with this one?	Y	Jeopardy	X
1.13	When should we implement this feature?	Y	Implicit	
1.14	What documentation affiliated with this change should I update?	G	Implicit	X
1.15	How can this story be broken down into smaller pieces?	Y	Jeopardy	

What parts of the source code will this change impact? (1.1)
This question was asked because the interviewee required information about where in the code he must make changes if he implements a story. This is achieved by finding the dependencies in the source code. Knowing where in the code the developer must change before he starts implementing will speed up the implementation. The information could also be used to plan when the story should be implemented. If two stories would impact the same part of the source code it might be better to delay one of them to avoid unnecessary merge conflicts.

Where is the least risk of breaking anything? (1.2)
Given that a developer has multiple ways of solving the same story, he was interested in what solution that would have the least impact on the whole system. The interview subject was referring to the number of different classes the story would impact as it would be better if the story would have to change a lot in one class rather than doing small changes in many classes. Doing small changes in many classes is more risky because it is likely to affect other developers, thus increasing the risk for merge conflicts. Asking the same thing as question 1.1 for every single way of implementation and compare the results can answer this question.

How can I implement this story? (1.6)
This question is within the area of IA because IA can help with answering how, or maybe more where, a story should be implemented. Given a set of alternatives one can do an IA on each and implement the one with the least cost.

How long time will it take to complete this story? (1.9)
This question was asked by a product owner that wanted to prioritize a story. In software, development cost is highly related to time or man-hours as immaterial products are produced. The interview subject wanted to know the cost of a story so he could put it in relation to the benefits of the change and with that information know if the story was worth doing.

What documentation affiliated with this change should I update? (1.14)
This address the same problem as question 1.1 but it aims at finding dependencies between source code to other CI:s instead of source code to source code. There are many reasons why other documents might change if the source code is changed. If the change is to add new functionality there might be a user manual that has to be updated. An optimization to an algorithm might trigger a change to a technical document.

3.2.2 Traceability category

A question was put in the traceability category if the information needed to answer the question involved connections between CI:s. Every question concerning traceability should be considered as relevant for this thesis as the CMDB should contain information about a project's CI:s and the connections between them and CodeScene can show dependencies within the repository. If they were labeled yellow or green was just a matter of personal interest. In table 3.2 there are eight questions connected to traceability. Out of these seven are marked as priority *G* but only one is analyzed. Because traceability have strong similarity to impact analysis this is no surprise, however, the time constrains forced the thesis to focus on impact analysis and not traceability.

Table 3.2: Elicited questions categorized as traceability related.

ID	Question	Priority	Extraction method	Analyzed
2.1	Which release was responsible for what bug?	G	Direct	
2.2	Which commits belongs to the same issue?	G	Jeopardy	
2.3	Which commits went into the release?	G	Direct	
2.4	Which parts of the code resulted in bugs in production?	G	Direct	
2.5	What has already been implemented?	G	Implicit	
2.6	Who did what change?	G	Jeopardy	
2.7	Who has knowledge about this piece of code?	G	Jeopardy	X
2.8	What in this class has changed from a previous stage?	Y	Implicit	

Who has knowledge about this piece of code? (2.7)
The question subject wanted the answer to this question in a few different contexts. The first one was that a developer might face the problem of lacking the skill and knowledge about the code base to implement a change and therefore has the need to ask a coworker for help. It would be helpful if he could look up who might know the problem, avoiding him having to run around trying to track down the right person. Another context was in the role of a product owner. Stacking knowledge about a part of the code base on only one person is risky. If that person would leave the company it would mean that there would not be anyone to ask how that part work anymore. Therefore, it is better to have some knowledge distribution in the company. Asking this question would answer if more than one person has the knowledge, thus making it possible to know if it should be redistributed or if it is fine as it is. The last context was aimed to find what developer that was most suited to implement a story. If the story was urgent or hard it would be most suitable to put the developer with the most knowledge of the affected code and if it was an easy change it might be better to put an inexperienced one to distribute the knowledge a bit. Knowing the answer to this question would enable the PO to make that decision. This question was considered as traceability because it aimed at finding the connection between a developer and source code.

3.2.3 Test category

In this section, all the questions about testing are presented. An overall insight from the interviews were that everyone believed that having good tests were crucial to a project. In table 3.3 all the questions are presented. There were nine questions that were categorized to be about testing. The three questions with priority *R* are all solved with standard testing methods. The ones with *G* requires more analysis and need more input to be answered, making them interesting. By looking at the table one can see that the main concern about tests is to be efficient and to focus the efforts in the right places. There were no question about how good the tests are, which might be because the interview subject that wanted to have more knowledge about test, were concerned about the project as a whole an not at unit level. His main concern was to help developers build code faster, which he suggested could be done with getting feedback from the tests faster.

Table 3.3: Elicited questions categorized as test related.

ID	Question	Priority	Extraction method	Analyzed
3.1	How can what has been implemented be tested?	R	Implicit	
3.2	Did the functionality change by implementing this story?	R	Implicit	
3.3	How can the code that changed be tested first?	G	Jeopardy	
3.4	How can adaptive testing be achieved?	G	Jeopardy	X
3.5	How can tests be split up into smaller test suits?	G	Jeopardy	X
3.6	Where should we invest our time?	G	Direct	X
3.7	What tests are we missing?	G	Implicit	
3.8	Did my code pass all the tests?	R	Direct	
3.9	Are we testing the right thing?	Y	Implicit	

How can adaptive testing be achieved? (3.4)
Running tests on a system can be very time consuming and take several hours to complete. The interview subjects described projects they worked on where they had to run tests that took up to eight hours. The idea behind this questions was a result of discussing how one could set up the testing process so that the most optimal test order was achieved. There are several reasons to why one would like to have adaptive testing. It allows the ability to only run certain tests which is desired if we only made a small change and we "know" that it won't affect the whole system. Another reason is that if we know that we have made a lot of changes to a certain part of the code, we might want to test this part first to fail fast because it would be a big difference to fail after one hour compared to seven hours. There is a lot to take into account when deciding what to test first: who has made the changes, how risky is the change, and what parts did we have the most problems within the last release. In order to change the execution order of the tests, they must be modular, which leads to the next question 3.5. Because testing was one of the main question areas and because values can be retrieved from the CMDB this was given the priority green.

How can tests be split up into smaller test suits? (3.5)
Often the testing part in software projects is not as well thought through as the code, but still all architectural principles should be applied to testing. Having modular tests does not only allow the change of execution order, it is also easier to connect test suits to parts of the code. This question was given the priority green because we believe that this can be achieved by doing a dependency analysis to get indications on how to split up the tests.

Where should we invest our time? (3.6)
Imagine that you are joining a new project and you are put in charge of writing test-cases for the source code. You realize that you have to write tests for the whole system which will take a lot of time. In the interviews, they expressed that being able to prioritize which tests to write first would be very beneficial, e.g it is better to start writing unit tests for the part of the code that is more prone to have bugs in it, or maybe a part where there is less experienced developers writing code. This was prioritized as green because CodeScene was believed to be able to help with these decisions.

3.2.4 Technical debt category

In this section the questions about TD are presented, table 3.4, and some are described. These questions were mostly from an interview subject that had a more management role at Praqma. He was responsible for the planning of where work would be carried out in many of the open source projects that Praqma is developing. He himself did not have an academic definition of what TD is and therefore TD was seen as everything in the code that makes future changes difficult and his overall goal was to make sure that the projects are healthy. In table 3.4, an interesting observation is that one question has the priority *R* and extraction method *Direct* while the rest have *G* and *Implicit*. Questions being implicit were a result of the interview subject not knowing that what he talked about could be classified as technical debt. Questions 4.1 has priority *R* because it is outside of the scope and is concerning techniques to reduce TD. Questions 4.2 to 4.6 are all highest priority because they are central to this thesis scope and the CMDB can be mined for answers.

Table 3.4: Elicited questions categorized as technical debt related.

ID	Question	Priority	Extraction method	Analyzed
4.1	How can we decrease technical dept?	R	Direct	
4.2	How generic is the source code?	G	Implicit	X
4.3	Is it possible to decrease technical dept while implementing this story?	G	Implicit	
4.4	What is the technical debt of the source code?	G	Implicit	X
4.5	Is it necessary to decrease the technical debt?	G	Implicit	X
4.6	How much will technical debt impact this story?	G	Implicit	X

How generic is the source code? (4.2)
This questions came from a scenario stated in an interview:

> *"A good example is our git-phlow project. It was targeted on git and had support for GitHub and GitHub issues. We wanted to extend it to Jira and Bitbucket. That should be a rather simple task. But we had to refactor the whole code. The method that talked to issues was GitHub specific and not generic at all. And we found other times where the code was static to GitHub."*

This is a good example where TD was not having a generic code, which can be seen as bad design (not having a modular architecture) and hindered a change task that seemed rather simple. We, therefore, state that how generic the code is can be seen as a measurement of TD. We currently don't know how to solve this but believe that it could be done by searching the code for static variables that are hard-coded in the program and analyze them.

What is the technical debt of the source code? (4.4)
Knowing the TD in the source code will help with estimating the health of the project and the cost estimations for working on the project.

Is it necessary to decrease the technical debt? (4.5)
The importance of TD may vary from company to company and from time to time. One thing to always have in mind when deciding on making an effort in reducing or preventing TD is to have time-to-market in mind. Sometimes taking on TD might be the right choice, if the result otherwise would be that one comes after competitors to market. It would be good to be able to do a Cost/Benefit analysis with respect to TD and to help make a case for the stakeholders that there might be a need to invest in reducing TD. Decisions on whether it is beneficial to reduce the TD needs inputs that can indicate what the results will be if no action is taken to improve.

How much will technical debt impact this story? (4.6)
When planning when and what stories to implement it is important to get good time estimations. There are several ways to do this, one way which from the interviews were, that visualizing the TD will help the product manager to better plan work. By knowing that there is a lot of TD in the story will help to estimate the risk and will not take the product owner by surprise when it is hard for a developer to make a fast implementation.

3.2.5 Other category

This section presents all questions that were not able to fit the above categories, see table 3.5. Some of these questions could have been grouped up into a new category but it would have been unnecessary as some were marked as irrelevant to the scope of the thesis. Creating a table for every category would also have impacted the readability of this thesis in a negative way. In this category most of the questions are attached with a priority *R*. This is because answering these questions are most of the time only a result of company standards and there is no need to seek the answer several times, e.g once you know the naming conventions for a branch you will always know how to answer question 5.1. On the other hand, we here have two questions with priority *G*. Question 5.14 is *G* because metrics to support the answer can be found in the CMDM. For instance one can consider something done when all the unit tests have passed or when proper traceability documentation is in place.

Table 3.5: Elicited questions that did not relate to one of the other categories.

ID	Question	Priority	Extraction method	Analyzed
5.1	What should I name a branch?	R	Direct	
5.2	Is this work going in the right direction?	R	Direct	
5.3	When should I stop and evaluate?	R	Implicit	
5.4	What should be standards? Task names, ordering, publishing behaves similarly,	R	Implicit	
5.5	How do I hand over something when I'm done?	R	Implicit	
5.6	What do the customer want?	R	Jeopardy	
5.8	Should I continue implementing this story or should I leave it?	R	Implicit	
5.9	How can I avoid scope creep?	R	Jeopardy	
5.10	What should I do with a bug i find while implementing a story?	Y	Jeopardy	
5.11	What is the context of this [change request]?	Y	Implicit	
5.12	Why am I making this change?	Y	Direct	
5.13	What is this product supposed to solve?	R	Direct	
5.14	How do I know that I'm done?	G	Direct	
5.15	Should i rewrite my local history before pushing to the repository?	R	Direct	
5.16	How can the backlog be cleaned up?	Y	Implicit	X
5.17	How do I make sure my work is understandable?	R	Implicit	X
5.18	Which team/developer are working on what part of the code?	G	Jeopardy	X
5.19	What priority should this story have?	R	Direct	

How can the backlog be cleaned up? (5.16)
The interview subject had worked on a project that had been going on for years. Stories was put into the backlog in a faster phase than could be implemented, so the stories had been piling up. Some stories had become outdated and was not relevant anymore, others that made it into the backlog had so low priority that they would never become relevant. The interview subject felt that having a pile of stories that would never be implemented made it harder to find the relevant ones. It was also time consuming, as finding a relevant story to implement meant a search through the irrelevant ones.

How do I make sure my work is understandable? (5.17)
To give insight into why a question is *R* we will here present a discussion about question 5.17. The questions was referring to if the developer used the right coding convention so that the other team members could understand his way of writing code. Having a common way of doing things in a company is crucial to be able to combine the efforts. Some of

the questions found from the interviews were a result of the company having no document describing their standards (5.1, 5.4, 5.5, 5.9, 5.10, 5.15, 5.17). This question was red because the answer isn't found in the CMDB, but rather in company standards which makes the question uninteresting.

Which team/developer are working on what part of the code? (5.18)
This question aimed at knowing who is working on what. This question is connected to awareness because the goal of answering this was to be able to tell if teams were working on the same parts of a system. If that's the case it could perhaps be good to rearrange the teams or the teams responsibilities.

3.3 Requirements and Use Cases

In the previous section we have presented all the questions that needs to be answered. By creating requirements and use cases we aim to pinpoint the data needed to answer a question. Going from questions to requirements is needed to eliminate ambiguities and to explicitly state what is required by a tool. The data from this section will be used when investigating the tools and thereby answer *RQ2 Which tools can provide the right data needed to answer the questions and how is it provided?* The scope for this thesis required the questions to be narrowed down to a manageable size. The reason behind not turning all the questions into requirements were that it took time to investigate each questions and we tried to start with the ones we believe to be most relevant and that can be built upon in the future.

The questions that we have tried to find answers to are some of the relevant ones in the categories IA and TD from the previous section. Impact analysis was chosen as it was the category that were of most interest to us. In our education we had been introduced to the term and wanted to study it more in depth. It was taken from the interviews that IA is not an important process. This was discarded as they have never done a proper IA and thereby do not know the true benefits of performing one. Technical debt were mainly picked as it was the questions that the workers at Praqma would like to have answers to. In the pre-study it was found that CodeScene possibly could provide the data needed to answer some of the question in both those categories.

We here present some detailed requirements that are meant to be used as if a requirement is full-filled then a specific set of questions can be answered. Later in chapter 5, a prototype is presented as a means to validate the requirements.

The process of creating a requirement was to start with a questions and analyze it by combining the question with the theories presented in section 2.5 and section 2.4. When the question had been reasoned about and requirements created, we took a step back and looked at the big set of questions and tried to see if there were other questions that also could be answered by the requirement.

3.3.1 For impact analysis

In this section three of the questions categorized as IA are analyzed and made into requirements for a tool. The most essential part of an impact analysis is to find the source code that will be affected by the change. Using this as a foundation it is later possible to extend the impact analysis by adding functionality, e.g estimating the cost or adding other configuration items to the impact set. In this section we start by presenting a question, followed by a motivation to why this is a good basis to develop a requirement on. We then present an analysis of the question, which results in a requirement, followed by a discussion of how to interpret it. To make it even more clear we provide some requirements with a use case, which is an easier way of presenting why we have a requirement and what problem it is meant to solve.

Questions 1.1 was chosen as a good starting point because its purpose is to find dependencies between artifacts in the source code, which from the literature study was an essential functionality needed to implement further requirements.

Question 1: *What parts of the source code will this change impact? (1.1)*

The question aims to answer which parts of the source code that will be affected by a change. To find what will be affected an impact analysis is required. In this case, a change request is specified by a user story. In the user story there is a textual explanation of the problem. To understand which parts of the source code that will be affected by the change, the developer must first break down the user story and identify where in the source code he must change, which will be the starting impact set (SIS) as described in section 2.4. The next step is to take account for the ripple effects and find the estimated impact set (EIS), which is what the tool should find.

IA-Req 1: *Given a starting impact set the tool should be able to find dependencies to all other affected source code files.*

Worth noting about this requirement is that it only requires the tool to work on source code. In table 3.6 a use case exemplifies an instance where the requirement provides the answer. As mentioned in section 2.4, impact analysis can be carried out on different levels of granularity. This did not get a requirement of its own, as it is included in this requirement, but a tool that support all three levels (object, function and file) is considered more complete than one that only supports one level. The same applies for different types of dependencies that are described in section 2.4. A tool must at minimum support one of the two (direct or indirect) to pass this requirement, but the tool would be better if it supported both. As described in section 3.2.1, the context of the question was a developer asking where in the code he must change to implement a story. Changing the source code usually also means that some tests must be updated or created. Therefore, source code is also extended to include test files.

Question 2: *Where is the least risk of breaking anything? (1.2)*

As described in section 3.2.1 this question is an extension of question 1.1. The objective of this question was to find where in the code a change should be implemented to impact other parts as little as possible. In an ideal world it would be sufficient to run an impact analysis to get the EIS and then to calculate the number of impacts to find where

Table 3.6: A use case describing an instance of when IA-Req 1 and IA-Req 2 are useful.

Description	A developer is required to make a change to a system. He has identified that the change will require modifications to a class.
Goal	Get a list of dependencies and how likely each dependency is to require additional changes.
Rationale	When making changes the developer needs to know how extensive the change is and what will be impacted. The developer is also interested in the probability that the change will actually affect the dependent file, as it might not be necessary to change the dependent file. This will help him know the risks and cost.

the least risk is. However, not all dependencies results in an actual change meaning that the EIS might not be the same as the actual impact set (AIS). Dependencies which are indirectly connected to each other should not be considered as likely to change as dependencies which are directly connected. A tool should therefore be able to indicate, by weighting the dependencies, how dependent a certain part is on another.

IA-Req 2:*The tool should provide information of how strong the dependency is between every part identified by the tool.*

Question 3: *What documentation affiliated with this change should I update? (1.14)*

Question 1.14 is an extension to the source code analysis and was therefore picked last. To answer this question the tool should not only find dependencies between source code, but also find dependencies between the SIS and other affiliated documents. Affiliated documents could be user manuals, design documents or any other document concerning the project that has to be updated. Table 3.7 presents a use case where a change to the UI triggers an update to the user manual. Unlike IA-Req1, the dependency analysis only has to be run at file level. It is not certain that a document has to be updated even if it is dependent a class so the requirements from question two should also be applied to this.

IA-req 3:*Given a starting impact set the tool should be able to find dependencies to all other affected documents.*

The use cases and theories above only consider dependencies from a starting point to some other object. Finding dependencies from the other objects to the starting point is also interesting as question 1.3 and 1.5 are answered by finding those dependencies. That scenario was not taken into consideration in this thesis as there was no time to also investigate that.

Table 3.7: A use case that describes IA-Req 3.

Description	An impact analysis is being done to a change request which is supposed to change the functionality of the UI. The change will impact how the user interacts with the software. The user manual should, therefore, be updated correspondingly.
Goal	Get a list of all affected documents.
Rationale	Most software development has documents like STLDD (software top-level design document) or User manual connected to their source code. A change to the source code should update those documents accordingly.

3.3.2 For technical debt management

Here we present requirements for TD management. In this thesis the TD activities within the scope are identification, measurement, monitoring and representation/documentation. Similar to the previous section we first states what question we are working with, then a discussion about it followed by requirements and use cases.

Question 1: *What is the technical debt of the source code? (4.4)*

To answer this there is a need to identify and and to measure TD. In section 2.5 metrics describing TD were presented. The identified metrics that are used in this thesis was duplicate code, long method, large class, long parameter list, divergent change, cyclomatic complexity, and code coverage. These metrics will be used in the investigation in the next chapter, but the requirements should not be limited to only those. Technical debt has other definitions so the requirement should also cover those.

TD-Req 1a: *It should be possible quantify the technical debt of the source code.*

To be able to quantify TD we need to know where to measure, which gave the requirement:

TD-Req 1b: *It should be possible to find the technical debt of the source code.*

This requirement has to be handled with precaution and if implemented it is important to understand what it is that one wants to achieve. First of, one should be able to limit where the TD is shown. If you are only responsible for a part of the whole project you only want to get information on that part. Also, technical dept in a part that is "frozen", never changing again, is no use in investigating as long as it works. Making changes in a piece of code that never changes and have high complexity is more harmful than good. Doing this just to improve the readability and ease of understanding the code must be highly thought through. Another thing to take into consideration is that finding TD could be done either by manually specifying what file to find the TD in or have the tool automatically pinpoint where it is.

Question 2: *How much will technical debt impact this story? (4.6)*

Knowing that there exists and to be able to quantify TD in parts of the code where there is planned work will help communicating the cost of implementation. Knowing how TD will impact a story will improve time estimations and help with planning. This leads to the requirement:

TD-Req 2: *It should be possible to specify the files on which to measure the technical debt.*

In a software project, the needed metrics on technical debt should be provided. The tool should be set up to show only the desired information to avoid information overflow. What parameters on TD that one needs to make good estimations depends on the context and what one want. Some parameters are easy to retrieve while others might require more effort, making the effort bigger than the earnings. If only values on cyclomatic complexity, code coverage and number of dependencies are desired one should be able to easily evaluate the task based on those. To better understand what we get from implementing *TD-Req 2* we have described a use case in table 3.8

Table 3.8: Use case for getting an overview of technical debt

Description	A developer is given a change task and wants to know what effort the task will require.
Goal	Get an overview of the technical debt, which will give input into effort estimation.
Rationale	Because TD makes current and future changes more difficult making a change to a story that is coupled with files with high TD will be hard. When planning a story it is therefore important to be able to get values on what TD there is in files that you are required to change. By having the whole picture there will be less surprises. Presenting only values on TD requires the developer to make decisions on his own, but htey are more informed.

Question 3: *Is it necessary to decrease the technical debt? (4.5)*

Getting an answer to the stated questions from a tool, that requires one to reduce TD is not a good idea. There is much that influence the decisions (deadlines, budgets, availability, skill and priorities), thus it is hard for a tool to exactly know when it is a good decision to reduce the TD. A better approach when dealing with TD is to look at metrics and then let an expert evaluate those and make the decisions. This questions should therefor by answered by the first requirement in combination with human effort.

Chapter 4

Investigation of CodeScene & Other Tools

When starting the investigation the goal was to answer *RQ2. Which tools can provide the right data needed to answer the questions and how is it provided?*, i.e. pinpoint what tools that can provide the data with respect to the requirements in section 3.3. While investigating we took note of how the data could be found and how it was presented. As mentioned in the introduction we wanted to look at a tool called CodeScene [1] to see what it can and can not do. We chose CodeScene because Praqma and we had an interest in it. In this chapter, we present the investigation and the outcome. If CodeScene could not present or provide the data, other tools were looked at to see if they could be used as a complement. This was also an expressed wish from Praqma because that they wanted to know how to complements CodeScene's analysis. The other tools treated in this section are EclEmma, JDeoderant, and STAN, which will be further introduced when they are brought up in the following sections. These tools were chosen because they were compatible with the Eclipse platform and were compatible with the projects that were used in the investigation.

Investigating the tools would also give us new ideas of what data that could be useful for impact analysis (IA) or technical debt (TD). In this chapter, there are three sections. The first two presents how CodeScene support IA and TD and how other tools can be used as complements, as well as presenting features CodeScene have that could have been requirements. Some of the "new" features were believed to be of good use in the context of this thesis and we, therefore, present them in each investigation section. To understand the features we conducted experiments as well as reading up on theories from the book *Your Code as a Crime Scene* [21]. In the last section, 4.3, we present use cases or requirements that were found during the investigation and not during the interviews but could have been. We created use cases to easily present them and discuss them. We have chosen to not make all of the use cases into detailed requirements because we believe that would not add anything to the content and understanding.

To do the investigation of CodeScene we set CodeScene up to analyze one of Praqma's open source projects named *Git-Phlow* [16] that is written in *go*. Later in the investigation, it turned out that some of CodeScene's functions were language dependent and we had to change since Go wasn't supported. Instead we used *Pretested-integration-plugin* [17] because it was written in Java, a language that was supported by every function in CodeScene. The project was also good because it was one of the projects that the people at Praqma had recently worked on.

4.1 Support for Doing Impact Analysis

This section presents the result of studying the tools and their ability to find the data needed to answer the questions that were selected in section 3.3. The investigation followed the same order as the requirements were presented in section 3.3 and will also be presented like that here. The investigation revolves around exploring CodeScene's function temporal coupling, which was mentioned in the background chapter, that finds dependencies in a repository. The section below will describe temporal coupling more thoroughly.

4.1.1 Finding dependencies on source code

To answer *What parts of the source code will this change impact? (1.1)* it was stated in section 3.3 that the tool must support *IA-Req 1. Given a starting impact set the tool should be able to find dependencies on all other affected source code file*. To satisfy IA-Req 1. the tool should support the theories about dependencies presented in section 2.4. Therefore, to find out if CodeScene could provide the data, we investigated if it could (1) find direct dependencies, (2) find indirect dependencies, (3) take a starting impact set as input, (4) work on file level, (5) work on function level, and (6) work on object level.

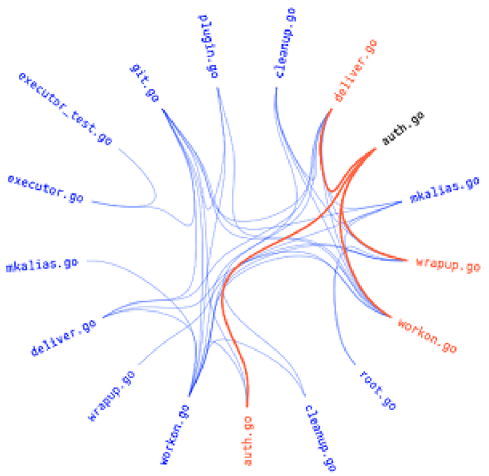


Figure 4.1: A figure showing how CodeScene visualizes the dependencies on file level. In this case *auth.go* is selected and all its dependencies are highlighted in red.

Dependencies (1-2)

Temporal coupling finds the dependencies by looking at the commit history in the repository. If two files have a temporal coupling it means that they have changed together in time. There are two ways in which you can do this in CodeScene, the first way is to look at the coupling by commit, i.e. if the files are committed together they will also have a coupling. The other way is to analyze it by looking at the coupling across commits. When looking for dependencies across commits one can choose to search by *Author and Time* or *Ticked Id*. If the former is set then CodeScene group the commits that a given developer has committed on the same day. All the files that have been included in those commits will have a temporal coupling. Setting the analysis to Ticked ID means that CodeScene group the commits that have the same story ID in its commit message. CodeScene displays the dependencies which can be seen in figure 4.1. If a line connects two files it means that they are related to each other. Because CodeScene bases its analysis on the commits in a project this means that commit strategies in a company influences the results. If commits are squashed together the coupling will be less evident. If a developers works on two files and he makes 20 commits that he push to master he will indicate that the two files have a stronger relation than if he would have squashed to 20 commits into one.

CodeScene has no option to distinguish indirect dependencies, but as CodeScene looks at historical data, if A changed together with C, CodeScene would flag them as dependent. So indirect dependencies are identified, but not distinguished as indirect.

The temporal coupling is calculated by taking the sum of how many times two files have been committed together times two and divide it with the total amount of times the files have been committed. Both the files will therefore be dependent of each other, i.e. the dependency is double linked. This could lead to false dependencies and is something that is discussed more in the discussion chapter.

STAN, a static data analysis tool, was meant to be used if CodeScene could not find the data needed to fulfill the requirements. It turned out that CodeScene could provide sufficient data for an impact analysis but STAN was briefly explored anyways. The idea of that study was to see if STAN could find dependencies that were not found by CodeScene. The result from STAN was different from the one found by CodeScene, e.g Stan found dependencies from source code to test code but CodeScene did not. This is probably due to the fact that Praqma does not develop the tests and source code at the same time, i.e. commit them together. It is possible in CodeScene to set a minimum of revisions that the two files must have together to be considered as dependent. Therefore, anther reason could be that the two files has too few revisions together. This can be set to remove dependencies that are commit together because the story required them but they are not truly dependent on each other. The scope of this thesis was never to validate the metrics so a thoroughly study why the dependency sets differed was never conducted.

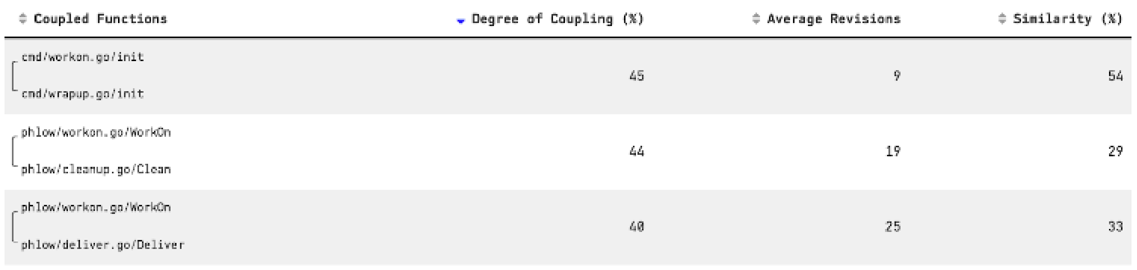
Starting impact set (3)

The names and lines that CodeScene visualize in the temporal coupling view are blue by default. If a name is selected, by hovering the mouse over it, all other files and lines that the selected file is dependent on are colored red. It is, therefore, possible to find the dependencies if the starting impact set only contains one starting point. A story might require changes that have more than one starting point. In order to find all dependencies for multiple starting points, we had to hover over each starting point manually write down the resulting estimated impact set. While hovering over files that have a strong relation we will get approximately the same EIS (exactly the same if they change together 100 of the time). Finding a bigger EIS requires the files in the SIS to not be strongly related, which they might be in the future after the story is implemented.

Levels of granularity (4-6)
In section 2.4 we learned that it is sometimes better to look for dependencies on function or object level rather than on file level. The results above are only showing how CodeScene can provide information about dependencies on file level, but CodeScene can also show the same information but on the function level. While running the analysis on function level it is possible to select an option to show dependencies within the same file, something that has not been discussed in the previous sections. This is a good addition because if a class is big, it is harder to understand it and there might exist dependencies between methods, perhaps as a consequence of copy-paste code.

4.1.2 Probability of impact

A dependency does not always mean that the other file also has to be changed, as described in section 3.3. To get a better estimated impact set, and to answer *IA-Req 2. The tool should provide information about how strong the dependency is between every part identified by the tool*, there is a need to indicate how likely a dependency is to trigger an additional change. CodeScene provides the user with a table of metrics, which can be seen in figure 4.2, in the temporal coupling view described above. *Degree of coupling* (DOC) is a metric that specifies the percentage of times two files have changed together in relation to the total amount of times they have been changed. The metric is calculated as the number of times two files have been committed together divided by the total number of revisions for both files. The metric can be used as a guideline for the experts so they can decide if the impact actually will happen. DOC is a metric that will become more reliable the more commits the files have. *Average revisions* can, therefore, be used as an indicator of how reliant the DOC is. Average revisions are the total amounts of times both files have changed, divided by two. The last metric, *Similarity*, is used as a TD metric and is described in the section below.



Coupled Functions	Degree of Coupling (%)	Average Revisions	Similarity (%)
cmd/workon.go/init cmd/wrapup.go/init	45	9	54
phlow/workon.go/WorkOn phlow/cleanup.go/Clean	44	19	29
phlow/workon.go/WorkOn phlow/deliver.go/Deliver	40	25	33

Figure 4.2: A figure showing the table with metrics that is shown in the temporal coupling view in CodeScene.

4.1.3 Finding dependencies on documents

CodeScene finds dependencies by looking at which files that were committed together which means that documents and source code can be treated the same way. It is, therefore, possible to find dependencies between source code and other documents, as long as they both are in a version-controlled repository. If the project uses a methodology where they are updating and committing the documents after the change to the source code has been committed, they will appear in two different commits. In order to link the two commits together, one must find dependencies across commit and by ticket id, as described above. With an agile approach the documents are updated at the same time which makes CodeScene able to find these relations.

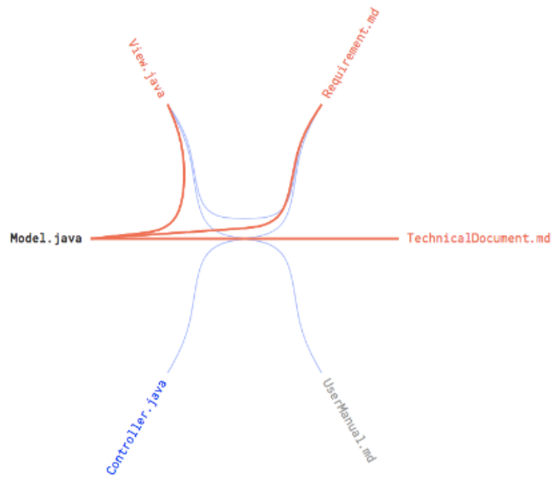


Figure 4.3: A visualization of dependencies between both source code and document files.

None of the above-mentioned projects had a setup where the documentation was in the repository. A mock-up project containing one git repository for the documentation (Requirements, Technical documentation, User manual) and one for the source code (Model, View, Controller) was therefore set up. For each commits the commit message specified what story, by id, it belonged to. Figure 4.3 shows how CodeScene visualizes the dependencies in that project. It shows both the dependencies between source code, but also between source code and other documents. CodeScene provides the same functionality for this as the one described in section 4.1.1.

4.2 Support for Finding Technical Debt

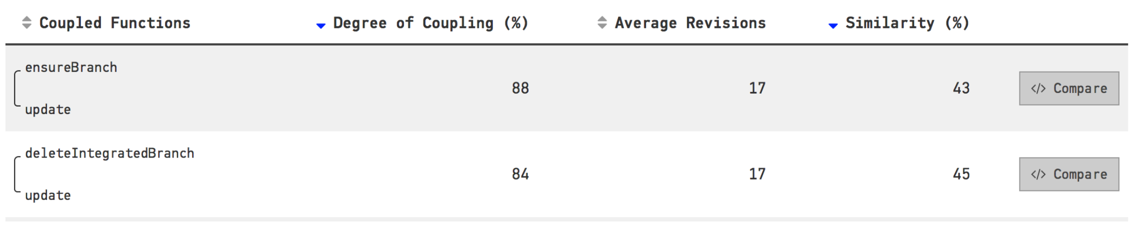
In this section, we start with presenting an investigation of how CodeScene can provide the metrics needed to measure TD, how the metrics are accessed and presented. This we did to prove that CodeScene has or do not has the functionality to fulfill our requirements found in section 3.3.2. We then describe what functionality we found while exploring CodeScene which we did because we wanted to find additional interesting features that we believe could be of use and worth describing. We have chosen to include screen-dumps of CodeScene to give the reader a feeling for how the tool works and what it looks like.

4.2.1 CodeScene's support for TD measurement

To investigate TD we started from the bottom by first only looking at a single file to see what information we could get and then work us up to repository level. This was done because if we could get metrics on every single file, we could argue that the results could be combined to present values for the whole project. We also quickly found that CodeScene only provided most of the metrics on file level. From *TD-Req 1a It should be possible to quantify the technical debt of the source code* in section 3.3.2 we know that for every source code file we need values on duplicate code, long method, large class, long parameter list, divergent change, cyclomatic complexity, and code coverage. Identifying TD is a prerequisite for measuring which makes looking for identification support is done by looking at measurement. Therefore, if the tool has support for *TD-Req 1a* it also has support for *TD-Req 1b It should be possible to find the technical debt of source code*.

Duplicate Code

The first metric we looked at was if CodeScene could tell us about **duplicate code**. There are several different ways to measure similarities between files. What CodeScene does is to detect copy-paste code and present a percentage on how similar they are, but CodeScene only allows you to view this if it has detected a dependency between the files. As shown in figure 4.4, which is an analysis of a file, CodeScene provides a value on how similar two files are, for example, the pair *ensureBranch* and *update* have a similarity of 43%.



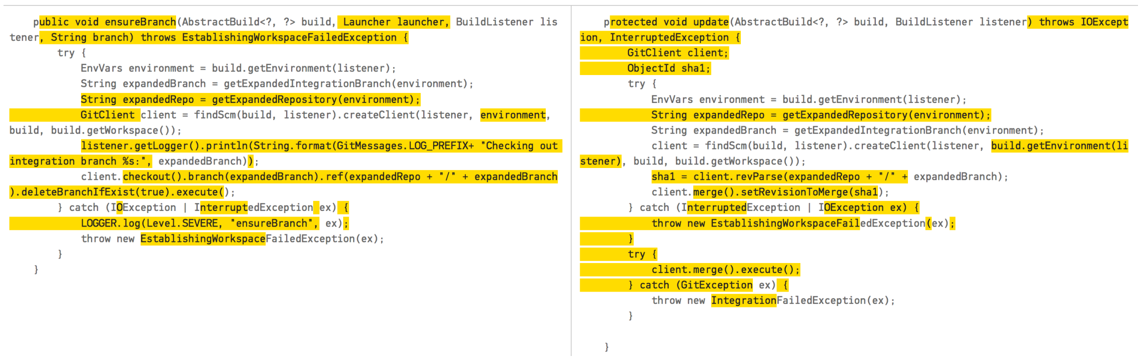
Coupled Functions	Degree of Coupling (%)	Average Revisions	Similarity (%)	
ensureBranch update	88	17	43	Compare
deleteIntegratedBranch update	84	17	45	Compare

Figure 4.4: Duplicate Code

To get a better understanding of the similarities one can compare the files. By comparing the methods *ensureBranch* and *update* one receives the view in figure 4.5 where the highlighted text is what differs. We believe that a more natural way to present copy-paste code would be to highlight the wanted information, the copy-paste code, instead of highlighting what is not duplicate code. In figure 4.5 it can be seen that *String expandedRepo = getExpandedRepository(environment);* and the first catch statement should be considered as duplicate code but are not. This shows that detection of copy-paste code is hard and could be a thesis subject on its own and was therefore not further explored in this thesis.

During the investigation, we've come upon files that had 100% similarity between them which was a result of copy-paste. CodeScene can uncover similarities between methods in the same class and between methods in different classes. CodeScene can not tell you straight up where there is duplicate code in the system, it is merely given when inspecting an item closer. It can however give you a hint where there is duplicate code, since duplicate code usually results in a dependency that can be found by the temporal coupling analysis.

JDeodorant is a static analysis tool that looks for code violations, namely type checking, feature envy (TD we have not discovered before), god class, long method and duplicated code. JDeodorant also has the possibility to identify duplicate code and automatically shows where it is in the system. For this it uses clone detection tools from a third party.



```
    public void ensureBranch(AbstractBuild<?, ?> build, Launcher launcher, BuildListener lis
tener, String branch) throws EstablishingWorkspaceFailedException {
        try {
            EnvVars environment = build.getEnvironment(listener);
            String expandedBranch = getExpandedIntegrationBranch(environment);
            String expandedRepo = getExpandedRepository(environment);
            GitClient client = findScm(build, listener).createClient(listener, environment,
build, build.getWorkspace());
            listener.getLogger().println(String.format(GitMessages.LOG_PREFIX+ "Checking out
integration branch %s:", expandedBranch));
            client.checkout().branch(expandedBranch).ref(expandedRepo + "/" + expandedBranch
).deleteBranchIfExist(true).execute();
        } catch (IOException | InterruptedException ex) {
            LOGGER.log(Level.SEVERE, "ensureBranch", ex);
            throw new EstablishingWorkspaceFailedException(ex);
        }
    }
```

```
    protected void update(AbstractBuild<?, ?> build, BuildListener listener) throws IOExcept
ion, InterruptedException {
        GitClient client;
        ObjectId sha1;
        try {
            EnvVars environment = build.getEnvironment(listener);
            String expandedRepo = getExpandedRepository(environment);
            String expandedBranch = getExpandedIntegrationBranch(environment);
            client = findScm(build, listener).createClient(listener, build.getEnvironment(li
stener), build, build.getWorkspace());
            sha1 = client.revParse(expandedRepo + "/" + expandedBranch);
            client.merge().setRevisionToMerge(sha1);
        } catch (InterruptedException | IOException ex) {
            throw new EstablishingWorkspaceFailedException(ex);
        }
        try {
            client.merge().execute();
        } catch (GitException ex) {
            throw new IntegrationFailedException(ex);
        }

    }
```

Figure 4.5: Duplicate Code Comparison

Large Class

In CodeScene, the most common view is presented in figure 4.6 where the big circle represents the whole repository, the smaller ones represent packages and the circles that are a shade of red represents files. The ones that are both big and red are files that CodeScene deems important bases on how much the file has changed in the past and how big the file is. This is a so-called *Hotspot* map which is a representation of the file system. By looking at a map like this one can get a feeling for how big the different modules are in the project, and from there identify **large classes**. Because the figure makes all the circles' sizes relative to each other CodeScene allows you to mark a file to retrieve the information about it as shown in figure 4.7. This view makes it very easy to spot big files that can be analyzed to see if they are too big but you have to do the work.

Long Method and Cyclomatic Complexity

For each function in any file CodeScene allows you to get values on *lines of code* and *cyclomatic complexity* (CC), figure 4.8. It gets lines of code by counting each line that is not empty or a comment and CC it obtains by calculating each independent path the program can take. Similar to duplicate code, one must manually analyze each file to get these metrics. Visualizing where the longest method or CC is, like the hotspot map, would save time, as one could quickly identify where it is present.

Lack of Test (Code Coverage)

There is no test support in CodeScene. It is, however, possible to analyze if a unit test file change together with the file it tests. If there is no temporal coupling between tests and source code it indicates that the tests are not kept up to speed. That could be an indicator that the tests are not evolving as they should. If the project uses a waterfall methodology then there might be no coupling between the tests and the source code, if analyzed by commits. This could be solved by tying all tests and source code tome the some story id.

To remedy this we have looked at a code coverage tool called EclEmma [2]. EclEmma is an Eclipse specific tool that runs all JUnit tests and marks all the source code that is executed. For each method, it then outputs a percentage of how many lines of code that were covered. This information is easy to extract as a *.csv* file if the data would be needed

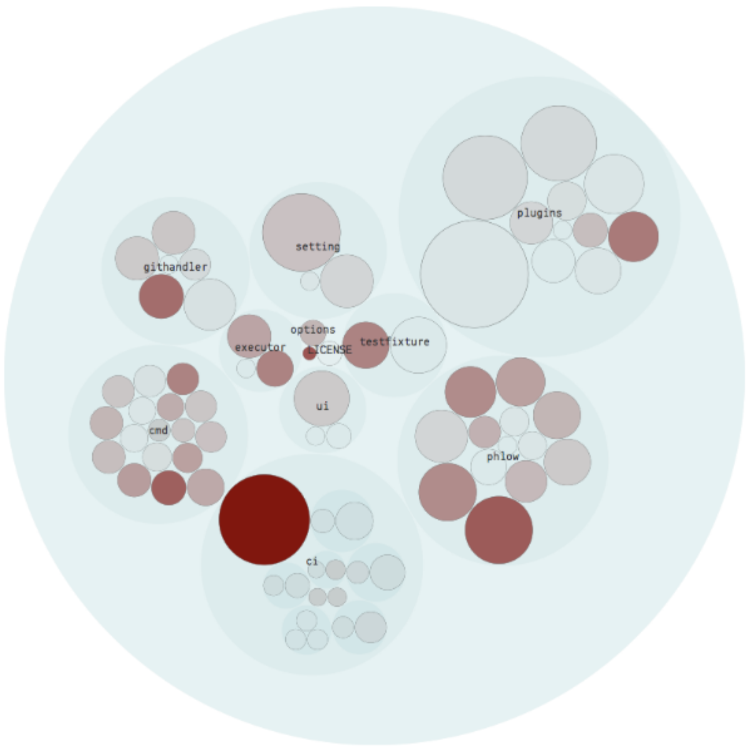
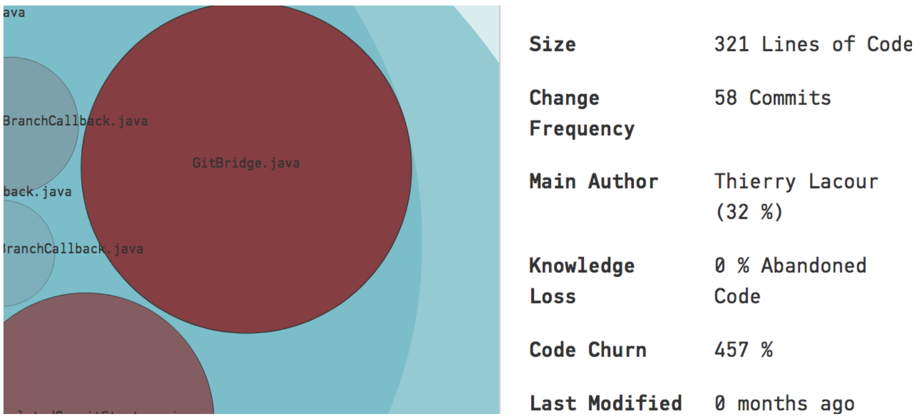


Figure 4.6: An example of a hotspot map.

.

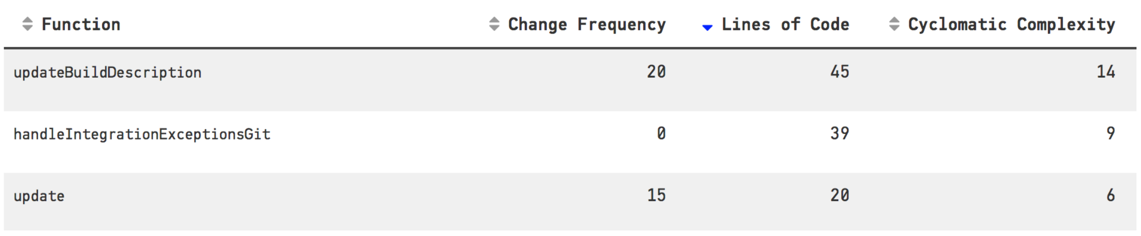


Size	321 Lines of Code
Change Frequency	58 Commits
Main Author	Thierry Lacour (32 %)
Knowledge Loss	0 % Abandoned Code
Code Churn	457 %
Last Modified	0 months ago

Figure 4.7: If a file is marked in the hotspot view one can retrieve the size and other information of the file.

as input in another application.

As for *TD-Req 2: It should be possible to specify the files on which to measure the technical debt* CodeScene only allows you to look at one file at the time. It would be possible to write down TD for each file to get a good overview. This would however not be useful in story planning because it would take to much time.



Function	Change Frequency	Lines of Code	Cyclomatic Complexity
updateBuildDescription	20	45	14
handleIntegrationExceptionsGit	0	39	9
update	15	20	6

Figure 4.8: By doing an x-ray analysis on *GitBridge.java* parameters on lines of code and cyclomatic complexity is given for each method.

4.2.2 Additional features in CodeScene

While exploring CodeScene we came upon two areas that we looked further into. The firsts area was complexity trend analysis and the second knowledge distribution. These features would help us to answer additional questions, that were not further analyzed, from section 3.2 as well as contributing with new ideas to requirements and use cases that we could have found during the interviews.

Complexity trends

In CodeScene, complexity trend analysis is the visualization of how a file's complexity evolves over time. We believe this to be useful because this allows us to know if a file is taking on TD in the form of overly complex code, which will allow us to get input to the questions *4.5 Is it necessary to decrease the technical debt?* for which we previously had no way of answering.

In figure 4.9 there are two graphs, the left is complexity and the right one is complexity to lines of code ratio. CodeScene measures complexity trends by looking at indentation because it is language independent and that an indentation often represent if, else, while and for statements [21]. By looking at figure 4.9 it is easy to see that the trend is based

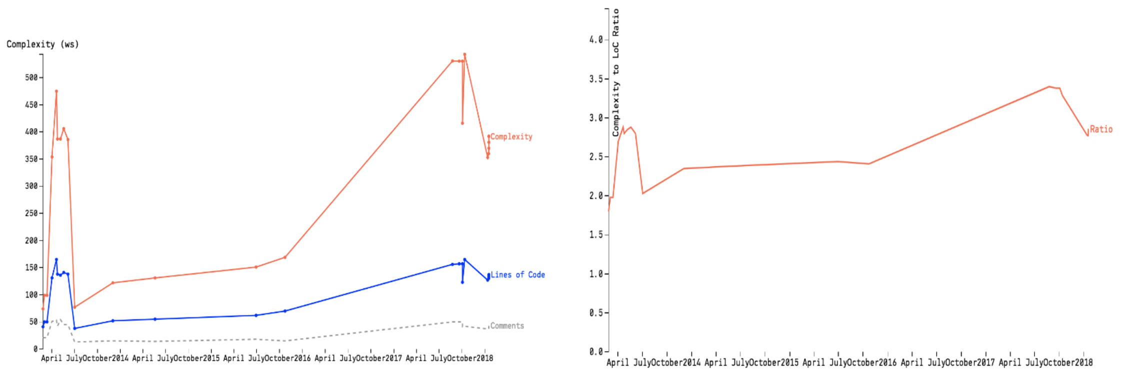


Figure 4.9: Complexity trends

on several years of development and one has to determine from the context how useful the information is. A project that develops faster may only want too look at trends in the recent past. It can, however, be useful to know that the project has evolved in the right direction in the last years. The image is used as an illustration of how it can look and the reason for the big time span is that Praqma had no project with an sufficient amount of commits to evaluate on a short time span and CodeScene's licensing didn't allow us to analyze any bigger projects. The reason that there is a plot showing complexity to lines of code ratio is that it paints a better picture of if the file is a risk. For example, this eliminates the risk of identifying false positives that would arise if many simple functions would be added to a file, increasing the overall complexity value. If the value on complexity to lines of code ratio has an ascending trend it indicates that the code gets harder and harder to understand. Complexity trends are useful because they indicate in what direction the project is heading. One might accept that taking on complexity once is okay but taking on complexity several times will lead to the code being too hard to understand and too complex to make further changes. The trend analysis is also good when investigating problematic areas in the code because it gives answer to if there is currently anything being done to increase or decrease the complexity and TD.

Knowledge distribution

The second area was knowledge distribution. CodeScene can for each file present a percentage for each developer that has made changes to that file. CodeScene shows this as in figure 4.10. These values are retrieved from taking the author and modified lines of code from each commit and sum them up. This can help answering the elicited question, in the traceability category, *2.7 Who has knowledge about this piece of code?*. But how does this connect to technical debt? This does not fit into the definition that we have set up for the thesis but if we see TD as "things that will make future changes difficult" we can argue that having one developer that has almost all knowledge about one part of the code, will make changes there harder. This because the developer might not be available and the code he has written is hard to understand. Another feature that CodeScene provides is a view of the knowledge in the whole project. By specifying the developers in the configuration of the CodeScene analysis, we can get a view of what code that we have good knowledge about and which part we do not. We can also see what happens is a developer should leave the project by setting him as an ex-developer. The code that no one will have knowledge about will then turn red.

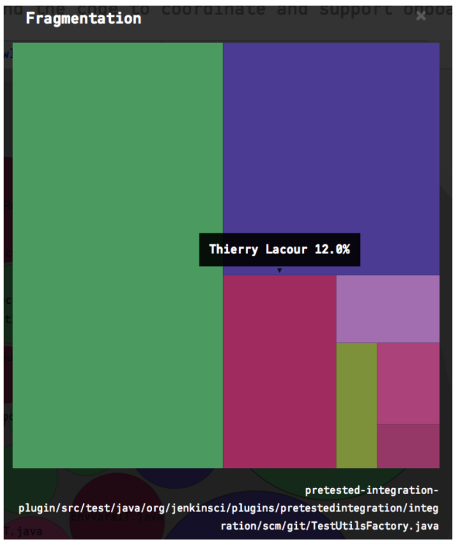

Fragmentation
Thierry Lacour 12.0%
pretested-integration-plugin/src/test/java/org/jenkinsci/plugins/pretestedintegration/integration/scm/git/TestUtilsFactory.java

Figure 4.10: Knowledge distribution of a file. Each color represents a developer and the size represents their knowledge.

4.3 Extended Requirements & Use Cases

The investigation gave new ideas to use cases and requirements that are presented in this section. The layout is the same as the one in section 3.3, but the requirements and use cases will not be connected to a question as these ones are more an extension to the former ones. The idea behind these requirements and use cases is to give future tool developers a foundation if they are to create something new, or maintaining an existing tool. They will not be included in the prototype as they originate from an existing tool and already are implemented.

4.3.1 Impact analysis

Exploring CodeScene gave us ideas for three new requirements that should be considered for an impact analysis tools. The first one was to be able to decide what parts of the project the tool should run the analysis on. In some situations it is unnecessary to find the dependencies as they are always there, e.g a header file will always have an implementation file in C or auto-generated code will always be dependent on the code that generates it. Removing the parts before the analysis was carried out meant that the experts did not have to remove already known dependencies that the tool suggested and thereby save some time.

IA-Req 4:*It should be possible to specify which files the analysis is run on.*

The visualization of the dependencies shown in figure 4.1 made it easier to grasp the size of the change rather than just displaying them in a table. To answer the question *How long time will it take to complete this story? (1.8)* one must first identify the size of the change.

IA-Req 5:*The tool should visualize the dependencies.*

The previous sections have only considered dependencies between source files. A change to an object in a file will most likely also impact other parts of the same file.

IA-Req 6: *The tool should show dependencies within files on function level or lower.*

4.3.2 Technical debt

There were several features concerning TD in CodeScene that were turned into use cases. This was done to easily describe what CodeScene can do and what we think is relevant functionality.

The use case in table 4.1 describes what will happen if someone will become unavailable but this is not the only use. The information can also be used to identify parts of the code that is mainly handled by one single developer and make sure his knowledge is spread out. By spreading out the knowledge the project's progress will not be as dependent on one single person nor will this person risk being a bottleneck if it turns out a lot of work has to be done in "his" part of the code. This can be seen as TD because letting the same developer make changes to a specific class is a short-term benefit (as he has the

Table 4.1: A use case describing knowledge loss of someone leaving a project.

Description	A developer will soon be unavailable, e.g going on paternity leave. There is a need to know how his absence will affect the project.
Goal	Get input on what actions to take when someone is leaving a project.
Rationale	Sometimes it can be hard to become familiar and comfortable with unknown code and architecture. In projects, there is a risk that the person with most knowledge of a part does all the implementation in that part. If that person will be unavailable in the future there is a need to know this because it may result in it being difficult to implement changes in "his" part of the code.

most expertise it will probably go faster than someone without knowledge) but would hurt the long-term health (if he quits, or somehow is not available anymore). The knowledge distribution information can also be used to identify the best-suited developer for a task, which can be the case if something really important needs to be implemented right away, as described in use case in table 4.2.

Table 4.2: A use case describing the need to find the best-suited developer.

Description	There is a need to put the developer with the most knowledge of the code in the story on the story.
Goal	Finding the best-suited developer for a story.
Rationale	The case might be that a deadline is approaching and there is a need to finish a task that the customer wants. The costumer also stresses that the story is of the highest importance.

Another case is the one presented in table 4.3, which uses the exact same feature as the use case in table 4.2. Here there is a need to identify a co-worker that can help with questions regarding the code base.

Table 4.3: A use case describing the need to a developer with good knowledge about the code.

Description	Anna is asked to implement a task and needs to know who to ask for help.
Goal	Finding a developer with knowledge about a specified part of the code.
Rationale	A developer is put on a task to work on a part of the code base that she is not familiar with. She needs to ask for help and therefore would like to know who has knowledge about that piece of code.

The functionality of complexity trend analysis that can be used to get a feeling for if a file will be problematic in the future, an example is the use case in table 4.4. The trends can be used to monitor if the work is going in the correct direction and if a team is doing what they have been asked. Trends could also be used to compare files' trends and thereby be able to prioritize which file that should be taken care of. A story with a steeper upward curve might be prioritized over a one that is growing much slower.

Table 4.4: A use case describing an instance of when a file should be monitored.

Description	A file with "technical debt" has been identified and there is a need to analyze it further. There is a need to know if that file should be watched.
Goal	Find out if a file will be a risk in the future.
Rationale	A file that contains technical debt doesn't have to be a problem. Therefore it's not certain that the cost outweighs the benefit when putting effort into reducing the technical debt. Therefore there is a need to know the trends of a file to guess if it will become a problem.

4.4 Summary of Chapter Four

In this chapter, we mapped the requirements found in section 3.3 to functionality in CodeScene. If CodeScene could not provide the needed data we looked at the tools STAN, EclEmma, and JDeodorant. By combining the tools we were able to provide all the needed information.

A central part of impact analysis is to find dependencies between files in the code. CodeScene did this by mining the git log to connect files to specific commits or task id. CodeScene then presents how many times two files have changed together as well as presenting a percentage of how likely the files are to change together. This is useful as it gives the user an indication of how strong the dependency is, so he can decide whether or not the dependency will carry out a change to both files. An important discovery was that CodeScene only uses double linked dependencies, meaning that if two files are dependent on each other CodeScene will present the probability as the same that the files will require the other one to be changed, which might not be true. We also established that more dependencies can be found by using a static analyzing tool, which is also suggested as a complement to historic dependencies in *Your Code as a Crime Scene* [21].

As for the TD metrics, we found that CodeScene could present values on duplicate code, large class, long method, and cyclomatic complexity. It was only for the metric large class that CodeScene provided a good overview of the whole project. For the other metrics, CodeScene required you to do the work because it would be too time-consuming to have CodeScene analyze everything all the time. These metrics are instead given on demand. As for getting a metric on lack of test we found that CodeScene's ability to provide information is limited by the standards in a project (requires a specific set-up) and it is therefore suggested that one should complement with a code coverage tool.

Other functions in CodeScene that can be of use are trend analysis and knowledge distribution. Trend analysis allows you to monitor files to see if the can become a risk and detect patterns in the development process. CodeScene presents a map of the knowledge distribution on a project. From this one can find out who is the best developer for a task, who I can ask for help, or the risk of someone leaving.

Another insight from working with CodeScene is that the visualization is important. Presenting the right amount of information and in an intuitive way helps the understanding and ease of use. CodeScene has a lot of other interesting functionality that we did not look further into because of time constraints.

Chapter 5
Prototype

We know from previous chapters what is required to perform an impact analysis (IA) and what we need to measure to find technical debt (TD). From the results in section 3.2 *Elicited questions* we chose to dig into questions concerning IA and TD and in section 3.3 *Requirements and Use Cases* we specified requirements needed to answer these questions. In this chapter, we will present the process and outcome of implementing a tool that is shaped by what has been presented in previous chapters, such as requirements and inspiration from CodeScene. There are several reasons to why a tool was implemented. Firstly, it was to serve as a proof-of-concept where we showcased how dependencies and technical debt can be presented in the way we wanted it to. Furthermore, the tool acted as a way to communicate what we wanted to achieve, making it possible to show it to developers and to let them try it out. Moreover, it would show the authors of CodeScene that the current technology could be extended so the tool could be useful as an IA-tool. Lastly, we wanted to do some more practical programming in contrast to all the theoretical work we had previously done. In the following sections, we describe the ideas behind the tool to give a motivation to why we wanted it to be this way, give a description of the implementation and detailed descriptions of some functionality, and explain how we validated the prototype.

By creating a prototype we expected to validate our requirements found in chapter 3 & 4 by showing that they are possible to implement and by having developers test the tool to validate that the tool's functionality is in line with their needs. It is also easier to validate functionality than requirements. By being able to show it to developers we were also able to get input on what functionality that was good and what was missing.

Because of limited time and resources, we did not manage to implement all the features we had in mind. We will still present what we have thought of to give the reader an understanding of what features we found to be useful and could have been included in the prototype. The prototype is meant to serve as a good basis on which one can build more functionality and as input to the creators of CodeScene to inspire them to add new features or tweak current ones.

5.1 Thoughts Behind the Prototype

This section presents the general ideas of why the implementation was created as it is. It will lastly also present a short reminder of the methodology used to construct the prototype, same as the one described in section 2.2.3.

Because we have studied certain questions we naturally wanted to create a proof-of-concept tool that is tailored to these questions. At first, the goal was to implement one tool that would implement the requirements from TD and one for the ones from IA. The TD implementation was meant to be an extension to the hotspot map found in CodeScene by adding the possibility to select on which metric the visualization should show. This would ease the detection of TD by letting the user see where the most TD is present, based on the metrics defined in section 2.5 *Technical Debt*. The idea was to visualize the different data by using the same framework as CodeScene does to visualize, D3 [5]. D3 is an open source framework written in Java that uses comma separated value files (.csv) as input and outputs a visualization like the ones seen in figure 4.6 (a visualization of a system using circles to represent size) and figure 4.1.1 (a list of file names with a line between them if they are connected). We spent a day studying D3 to figure out how to display the data and how to integrate it into a GUI framework. Integrating the visualization with a GUI turned out to be difficult thus making it hard to complete a prototype within the time frame of this phase. It was therefore decided to only focus on the IA implementation.

From the investigation of CodeScene based on IA, section 4.1, it could be concluded that CodeScene could not support multiple starting points and it could not give an estimated impact set (EIS) that only included the affected files. Therefore, the initial goal with the IA tool was to implement those features. It was planned to be done by merging and filtering the data that could be extracted from CodeScene, so the data could be turned into useful information. The proof-of-concept tool should also contain the visualization that was seen in figure 4.1.1, but only with the files that was in a resulting estimated impact set. Having the tool to only work on file level would be enough to prove the points for this thesis. Dependencies on function and object levels were therefore left out but could have been implemented just as easy, as the data could be extracted from the same way as it was done on file level.

With the prototype in place it could be seen that developers could be interested in more information about the change than only the dependencies. In a previous course read by one of the authors they used the XP methodology to develop software. A typical change was carried out in the following way, first the unit tests were implemented, followed with a short refactoring of the existing code. The purpose with that was to gain knowledge of the code pieces that was currently worked on but also to maintain the code base continuously. That followed with an implementation of the story that was carelessly written but good enough to pass the tests. It was eventually refactored once again so to make it more readable for other developers. This methodology was also in line with the one used at Praqma, which was the reason that one of the interviewees wanted to know *How much will technical debt impact this story? (4.6).* One goal of refactoring is to reduce the code smells described in section 2.5. It was therefore decided to include the TD metrics of the affected files from the resulting estimated impact set, to facilitate the decision if a refactorization is necessary or not.

Adding metrics about the software along with the EIS would also help the developer to estimate the effort needed to complete the story, thus being able to answer question *1.9 How long time will it take to complete this story?* The metrics that were chosen to be implemented in the prototype are presented in the next section and are given a context to why it was chosen. It was chosen to only add the metrics that belong together with a change, e.g extracting a method from a long method to improve readability of the code is something that could be done while working on that function but remodel the design of the software structure would probably require stories of its own.

The ideas for the prototype derived from the requirements and investigation of CodeScene was first put onto paper in the form of a mock-up and a sloppy written technical documentation. This served the purpose of quickly being able to validate that the ideas were of use for developers and not something that would be trash. This mock-up was left out from this thesis as it would only take unnecessary space. The mock-up later became a working prototype that can be seen in figure 5.1. The prototype was implemented in small increments, each which added functionality. Only the last increment is shown in this thesis as it shows how the prototype work in its current form. Functions that was not implemented due to the time constrains are presented last in the next section, they include visualizations and tweaking of parameters.

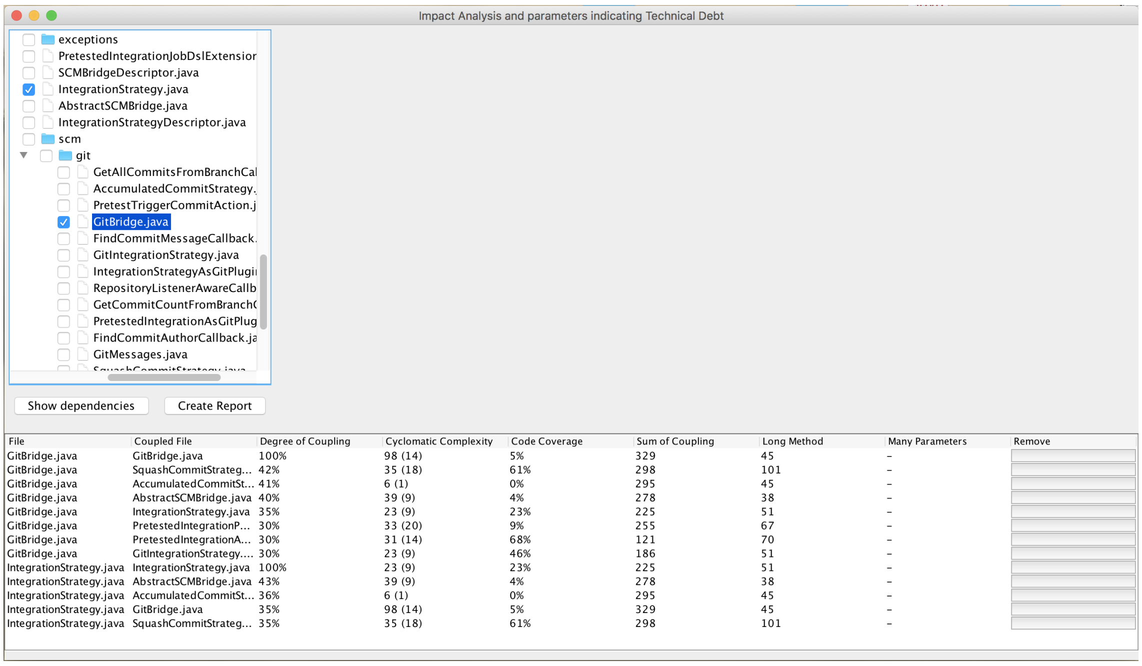
5.2 Implementation Specification

In this section, the current state of the tool is presented. We explain in detail what it can do and how to do it. We explain what limitations there are and what we would have done if we were to develop it further. Because we only spent three weeks on creating a mock-up, implementing the tool, present and validate it, and implement some suggested changes; the functionality is limited. In this section, we also add theories when necessary to explain our choices to aid the reader and to remind why certain metrics are good to have. The tool has good support in our research and we found that it could be a good starting point and addition to CodeScene, to help product owners to make more informed decisions.

The tool is written in Java as it was the language that both authors were the most confident in. It was also the only language that we had experience in a GUI framework, namely Swing. The code was kept under version control by using Git and was developed in the integrated developing environment Eclipse. It turned out that it might have been better to do a web-based solution instead of a desktop application as one of the APIs that were used produced JavaScript files.

The data that was used to provide information about the files were gathered from CodeScene and EclEmma. Luckily both those tools stores their data in .csv files which made the parsing and filtering of the data easier. Neither of the tools had an open API that could be used to run the analysis and collect the data. Therefore, we had to manually run the analysis in the tools and search for the produced analysis files in the file system to put them in the prototypes file system. This is not considered to deem the tool less useful because the tool is a proof-of-concept and manually organizing the files shows that it is possible. In a more permanent project this process could be automated, perhaps by writing some scripts. The data files are read and written to/from the GUI using *Apache commons CSV*, which is an open source API to handle .csv files. The filtering was done by implementing the methods ourselves.

The current version of the implementation can be seen in figure 5.1. The top left corner of the implementation is a representation of the file system that lets the user browse through all files on her hard drive and select which files to include in the starting impact set (SIS). When pressing the button "Show dependencies" all impacted files to the files that are marked are shown at the bottom part of the picture. For each of the impacted files, the values on chosen metrics also appear. The values correspond to the file name in the column "Coupled file". The SIS files are also coupled to themselves and shown in the table. By doing that it was possible to also get information about the SIS files. If the remove button in the last column is pressed the whole corresponding row is deleted, which can be used to trim the EIS. We also implemented the possibility to save the EIS by pressing "Create report" that will create a .csv file that matches the table of values. The purpose of creating a report is so that one can do the impact analysis in the planning phase and then pass the result on to the developer that will implement the task. He then gets the list of dependencies and metrics and can plan accordingly. It is possible to customize which metrics that should be saved when creating a report. This is based on the fact that a CSA tool should be flexible and fitted to the particular needs.


Impact Analysis and parameters indicating Technical Debt
exceptions
PretestedIntegrationJobDslExtensior
SCMBridgeDescriptor.java
IntegrationStrategy.java
AbstractSCMBridge.java
IntegrationStrategyDescriptor.java
scm
git
GetAllCommitsFromBranchCal
AccumulatedCommitStrategy.
PretestTriggerCommitAction.j
GitBridge.java
FindCommitMessageCallback.
GitIntegrationStrategy.java
IntegrationStrategyAsGitPlugi
RepositoryListenerAwareCallb
GetCommitCountFromBranch(
PretestedIntegrationAsGitPlug
FindCommitAuthorCallback.ja
GitMessages.java
Show dependencies
Create Report

File	Coupled File	Degree of Coupling	Cyclomatic Complexity	Code Coverage	Sum of Coupling	Long Method	Many Parameters	Remove
GitBridge.java	GitBridge.java	100%	98 (14)	5%	329	45	-	
GitBridge.java	SquashCommitStrateg...	42%	35 (18)	61%	298	101	-	
GitBridge.java	AccumulatedCommitSt...	41%	6 (1)	0%	295	45	-	
GitBridge.java	AbstractSCMBridge.java	40%	39 (9)	4%	278	38	-	
GitBridge.java	IntegrationStrategy.java	35%	23 (9)	23%	225	51	-	
GitBridge.java	PretestedIntegrationP...	30%	33 (20)	9%	255	67	-	
GitBridge.java	PretestedIntegrationA...	30%	31 (14)	68%	121	70	-	
GitBridge.java	GitIntegrationStrategy....	30%	23 (9)	46%	186	51	-	
IntegrationStrategy.java	IntegrationStrategy.java	100%	23 (9)	23%	225	51	-	
IntegrationStrategy.java	AbstractSCMBridge.java	43%	39 (9)	4%	278	38	-	
IntegrationStrategy.java	AccumulatedCommitSt...	36%	6 (1)	0%	295	45	-	
IntegrationStrategy.java	GitBridge.java	35%	98 (14)	5%	329	45	-	
IntegrationStrategy.java	SquashCommitStrateg...	35%	35 (18)	61%	298	101	-	

Figure 5.1: Picture of the current version of the prototype

The metrics we have chosen to present for each file in the EIS are the metrics that could be useful to decide whether or not the file should be refactored and those that would help the user to estimate the effort needed to complete the story. Noticeable is that duplicate code is not one of them. As stated in the previous chapter, it is difficult to find duplicate code. We would also argue that when looking at copy-paste on file level where you add an abstraction level to fix the "code smell" or splitting up a big class are considered as a "major" refactoring and should have a story on its own. We will now motivate why each of the metrics is included in the tool:

Degree of Coupling tell the user how likely the dependent file is to change. This metric can be used for an expert to filter out dependencies that should not be included in the EIS. A high number means that the dependency is likely to trigger a change to both files.

Cyclomatic Complexity is a way of presenting how difficult a class/function is to understand. We chose to present the total CC for a class as well as the highest CC for a function in that class. The highest CC for a function can be used to decide if the function should be refactored or not while implementing the story, thus lowering the cc for the function. The metric can also be used as an indicator of how hard it will be to understand the code and thereby give a better basis for estimating the effort needed to implement the story. The more complex the function is the harder it will be to change it.

Code Coverage is one way to get a value on if there are a sufficient amount of tests. The metric can be used to decide if it is necessary to write more tests to the affected file or not. It can also be used to get a better basis when estimating the story. If the developer knows that the current code is well tested then he can be more confident while implementing the new additions.

Sum of Coupling is a metric that specifies the total amount of files that a file has changed with. A file that has many dependencies is more complex to change than one with a few. This metric could, therefore, be used as a basis to estimate the effort.

Long Method shows the highest number of lines of codes a single function in a file has. A too long function could mean that the function might have too much responsibility and thereby be violating best practices. This metric should, therefore, be used as a decision maker to decide whether or not the function should be divided into smaller functions.

Many Parameters presents the highest number of parameters a single function in a file has. Having too many parameters indicates the same thing as Long Method and should be used in the same way. This metric was never implemented as no tool that was investigated could provide this data.

Remove can be used to remove dependencies that were found by the tool. A dependency does not mean that a change is needed to both the files. This metric can, therefore, be used to remove dependencies that an expert know won't trigger a change and thereby create an EIS that is closer to the actual impact set.

Not all functionality that was meant to be implemented was done due to the time constraints. Something that we wanted to implement was to customize which metrics to show in the tool. The first idea was to not overflow the user with information and thereby make it hard to use the information. A worker at Praqma did, however, suggest that he would like to see more data about the files and include the possibility to toggle them on and off. This was because he found some metrics misguiding, like if he knows that the developers "cheat" with the code coverage metric it can do more harm than good to include it. The code is however structured to make it is easy to add and remove metrics.

For the big gray area in the upper right corner, we had big plans to present a useful and interactive display that visualize the dependencies of the estimated impact set. We wanted to take inspiration from CodeScene's visualizations by using D3 and tailor them to our needs. We didn’t manage to get it to work with our implementation because D3 produces JavaScript files and that requires third party API’s to be displayed in Swing. There was not enough time to learn those third party API’s. One solution to this would have been to make the whole program as HTML code which D3 is made for.

5.3 Validation of Prototype

By having the core functionality there were a need to validate tool. We could have continued to add functionality but getting feedback that we are working on the right thing would help us to focus our efforts. After the validation, we hoped to have validated the current functionality and to have new ideas to add. We also expected input on what was good and what could be improved. The goal was never to have the developer validate that the values on the metric were correct but merely to have him comment on whether or not they could be of use if they are correct.

The validation was done by asking three developers what they thought of the tool. One developer had the time and interest to sit down for a more in depth demonstration, leading to better discussions while the other ones only gave short comments on the their thoughts. When doing the validation with the developer that were interested, we made him sit down with us as we demonstrated the tool and explained the ideas behind the design and metrics. We explained how it was intended to work and had him try a few functions, while he tried to explain how he interpreted what the tool showed. We had set up the tool to use a project that he had worked on. His immediate reaction when looking at the values from two problematic files was that it aligned with his gut feeling that they were a problem because the tool showed that they had a lot of dependencies with complex classes.

From him, we also received feedback that the metrics need more context. For example, we could state the name of the methods that are troublesome in the sense of lines of code and cyclomatic complexity. It would be nice to know if the tool flags the same method for those. Another metric that we would add directly if given time would have been lines of code for each file. The idea was to have this visualized but could also be useful as a value that one could relate to the other metrics. As we discovered in CodeScene, the cyclomatic complexity to lines of code ratio could come in handy. One of the interviewees found the tool particularly useful for what he called story grooming, i.e breaking down stories into smaller ones. With the information from the tool, dependency and complexity of the change, he could decide if the workload was small enough for a single story and if not the story could be broken down.

The main take away from the validation was that when we showed the tool to developers, three in total, they believed that it would be useful and that the parameters can help them make more informed decision. However, the best way would have been to have them use the tool to influence their real decisions and then have them explain how it affected their reasoning.

Chapter 6
Discussion & Related Work

In this chapter we will discuss the chosen method, our results in relation to related work, what threats to validity there are, and future work describing how our study can be extended. The method will be discussed by evaluating if we took the correct approach based on what we know now. By presenting a discussion of the method we show what we have learned by analyzing what data we had needed to make better decisions. This will also help others to avoid similar mistakes. Results & Related Work presents how our results compares to previous studies where we discuss how our results can strengthen, complete or contradict (or be contradicted by) their results. This will help us motivate if our work has made the society at least a little bit wiser. We then present what threats to validity there are to our results. Be doing this we show which results are well motivated and which need more strengthening. Lastly, we will present how our thesis can be extend by future research to give useful tips on what can be the starting points of a new study.

6.1 Evaluation of Methodology

The overall working method used in the thesis was to have detailed plans for the activities for the next week and have a more loose planning for the long time span. This was a good approach because the work was exploratory and we never knew what direction it would take. The exception to this was when implementing the prototype, where we had a deadline of three weeks, and when writing the report where we worked with a deadline for each chapter. While working, we used working notes to document our findings that would later be used as the data for this thesis. Writing the report after all the work had been carried out allowed us to have a better discussion of what we wanted to put in the report.

The steps taken in the thesis were taken in a good order as the preceding step provided input to the next. At the start, we knew that we were going to (1) elicit questions asked in software development, (2) present answers to them, and (3) validate the answers. As we explained in section 2.2 *Method*, to do (1) we read literature and held interviews. The interviews had a good methodology because they allowed us to find a wide set of questions. Initially we had planned to have more interviews to have the ability to validate questions by having different developers state the same questions. Because of limited resources, we did not have the opportunity to interview more people with different backgrounds and the four subjects we found had to suffice. Alongside the literature study leading up to the interviews we held a configuration management coffee meeting (a meeting for people interested in configuration management) where we hoped to gain better insight into modern configuration status account and get an initial input on what kind of questions we could expect from the interviews. We did not retrieve much information from the meeting but it helped us realize that the scope was a bit to wide and that we had to conduct the interviews to know what there were a need for us to explore further.

To be able to do the second (2) we selected what questions we wanted to focus on and turned them into requirements and use cases to be able to investigate if CodeScene and other tools could provide data to answer them. The selection of the questions to answer (the ones in impact analysis and technical debt) was based what we found interesting and thought we would be able to find good solutions to. An additional approach would have been to compile a list of good questions and have new interviews to have them look and comment on which they find the most interesting and relevant. We could also, after having chosen to work with the questions connected to technical debt and impact analysis, have gone back and asked the interviewees what they believe those are to get a definition to work with. Instead, we chose to find theories in literature because of two reasons: we wanted the definition to be general and not biased, and we knew that they themselves didn't know the root cause to why they weren't able to answer the questions today. Lastly (3) we had to choose how to validate what we had done. To do this we implemented a prototype that we were able to show developers at Praqma. This was a good way to validate the requirements because it is easier to show and discuss functionality. In the initial method plan the idea was to validate the requirements by presenting our results to developers and have them discuss the relevance. This was never done because we found out that implementing a proof-of-concept tool would be easier for the interviewees to understand and thereby validate.

Before we set focus on technical debt and impact analysis we had the problem of having a too big scope because we also wanted to find answers to questions about testing. This led to wasted time trying to do too much at once. We also started reading up on a lot of theories about these areas, leading to very slow progress, because we tried to cover all of our bases. We were then introduced to the "just do it" approach and decided to take the first (but deemed good) definition of technical debt and impact analysis. We learned that doing something is better than reading and doing nothing.

6.2 Results & Related Work

In this section we present how our results relate to other researchers work. Their papers are summarized and followed up with a discussion to how it relates to ours. We have three results that we discuss further, namely our set of questions on comparison to what others have found, how CodeScene finds historical dependencies compared to another way it can be calculated, and how impact analysis can be used in effort estimation.

Comparing results to questions found in related studies

Thomas Fritz and Gail C. Murphy [10] identified that some of the questions developers ask during their daily job are hard to find information about and can be answered by either finding pieces of the needed information in different domains and manually put them together or creating queries on the CMDB. Therefore, they presented a model that aims to provide information about such questions and a study where they investigated how that model performs. To know what questions developers ask they first interview eleven employees from three different sites of one company. To evaluate the model they first created a prototype and then conducted a study where 18 subjects had to answer eight of the found questions with the help of the prototype.

The first study resulted in a list of 78 (46 listed in the report) questions developers ask in their job. One questions were stated by two different interview subject, while the rest were only stated by one. The study of the model showed that 94% of the cases could be answered within the time limit, with a mean time of 2.3 minutes. 16% of the time the subject took more than 5 minutes to answer the question and 6 times the authors had to give a hint to the subjects. The subjects could not find the answer after 10 minutes in only 3 out of the 144 cases.

From their list of questions, it can be seen that the largest set of questions were those regarding changes to the source code, 20 out of the total 46. Most of those 20 questions are asking about changes that have already been carried out to a configuration item and would in our thesis be categorized as traceability. Most of our questions were also connected to changes, naturally, since that was the scope, but rather "what would happen if I did this to X" than "what has happened to X". As can be seen in table 6.1 some of the questions they found could be interpreted as the ones we found. The question *What code is related to a change?* was one of the questions that were used as a basis for the IA requirements in this thesis. That they found the same strengthen the relevance of that question further.

In the paper by Andrew J.Ko et al. [12] they present a result of a study aimed at identifying and characterizing developers' information needs. How developers made decisions while working was also included in the study. By identifying these needs the authors believe that we'll get a better understanding of what tools, processes and practices we need to improve and what we should focus on, and in the long term improve the overall quality of software. The study was carried out by observing 17 developers at Microsoft. These were taken from a pool of 49 (out of surveying 250) developers that agreed to be observed. How these 17 were selected were not mentioned in the report but they stated that they would have liked to observe more but it was not within the time-frame of the project. They claimed that observing 17 developers were enough to see common patterns in their information needs.

Their main contribution is a list of 21 (generalized from 334 instances of information seeking) types of information a developer seeks. As we mentioned in section 2.3 *Literature Study* the questions considered importantly concerned what coworkers had been doing, what is the reason for certain program state, and questions about things their code are dependent on.

The questions J.Ko et al. elicited are similar to the ones we found, some only different in the phrasing of the question. They did their study by observing which implies that these questions were found by two separate methods. For example, it is easy to see that we have found similar questions by looking at table 6.1. Most of the questions concerning

Table 6.1: Some questions from our thesis compared with the ones in the paper by J.Ko et al [12] and Fritz et al. [10].

Questions from this thesis	Questions from J.Ko et al. [12] and Fritz et al. [10]
What should be the standards?	Did I follow my team's conventions? [12]
How long time will it take to complete this story?	How difficult will this problem be to fix? [12]
Should I continue implementing this story or should I leave it?	Is this problem worth fixing? [12]
What parts of the source code will this change impact?	What code is related to a change? [10]
Who is dependent on this code?	Who is using that API [that I am about to change]?[10]
Who did what change?	Who has made changes to my classes? [10]

technical debt were not found in related studies and are questions we received by expanding our scope to include more than developers. By finding questions relevant to the planning phases is relevant because it can increase the effectiveness of the project. If the planning is done right by doing the stories in a good order, knowing the risk for each story, and having the right developer in the right task the implementation will be easier. We have also continued to work with the questions that we elicited which others have not done. By doing that we showed one way of tackling the problems, by showing how we created requirements that we validated with a prototype. We have also shown that there exist solutions to answering the questions developers have but one needs to consider that nature of the question. The main questions we worked with concerning technical debt are all different. The first question, *What is the technical debt of the source code?* is very different from *Is it necessary to decrease the technical debt?*. The first can be solved by deciding what parameters to measure while the second one is very dependent on the context. The second question can, however, easier be answered by first answering the former because that gives indications on how healthy the code is.

Mining version histories to find dependencies

We have previously described how CodeScene finds the dependencies by mining the version control. The goal of this thesis was never to validate the data that was used in the tools but as we explored CodeScene we out of curiosity also studied how historical analysis works. In a paper by Thomas Zimmermann and Stephan Diehl, *Mining Version Histories to Guide Software Changes* [23] they present their work on a tool named ROSE. In this paper, they describe how ROSE works and evaluate its results. It mines the version histories from CVS and created rules to how files change together, both on file and method level. For each rule it stores the *Support count* which gives the number of transactions a rule has been derived from and the *Confidence* which is the strength of a rule, for example if a change to a file has led to change in another file 10 times and that file has changed 11 times the confidence would be 10/11. The idea behind ROSE is that it is supposed to be used continuously and integrated into your IDE. To evaluate ROSE they analyzed eight open-source projects. To make it simple, they used the tool to see what is recommended to change and then compared it to what actually changed. For example, they mined one month of history to create the rules and then used stories that came later in time as input and looked at how good the tool could predict what would change in that task. They found that the three topmost recommendations contained a correct dependency in 90% of the cases. It cannot predict new functions, which is a limitation for all version mining tools.

They found that the tool is best during the maintenance phase of a project and that ROSE is good at predicting missing changes and a warning of a missing change is only wrong 2% of the times [23].

While investigating dependencies in section 4.1.1 it was found that CodeScene calculates the dependencies by taking the sum of how many times two files have been committed together times two and divide it with the total amount of times the files have been committed. This means that both files will always become dependent of each other. If A has changed 20 times and B has changed 80 times and they both have changed 20 times together. Using CodeScene's equation would mean that they would have a degree of coupling of 40% but in reality, B changes 100% of the times A changes and A only changes 25% of the times B changes. ROSE uses another equation to calculate the degree of coupling (confidence) by taking the sum of how many times two files have been committed together and divide it with the total amount of times the starting file has changed. Using this equation instead of the one CodeScene uses will both produce the right coupling and treat the dependencies as unidirectional. From this paper, we learned that there exists a desire to create tools that use the version histories to find relations. Because ROSE works in the way we want a dependency tool to work it would have been suitable to use its way of finding dependencies in our prototype.

Our investigation of CodeScene was important because it found weaknesses and strengths, and highlighted functionality that we found useful. We have presented scenarios which gives a better understanding of what CodeScene can do. Our scenarios are good because they have been taken from conversations and have been discussed with real developers.

Impact analysis used for effort estimation

At the end of this thesis, it was found that Binish Tanveer et. al has an ongoing research where they aim at improving effort estimation (EE) in an impact analysis in agile software development. They have published a few papers of their current work and the most interesting to this thesis is *Understanding and improving effort estimation in agile software development- an industrial case study* [20]. EE is in most cases done by letting experts use their gut feeling to estimate a story. They, therefore, wanted to explore how EE in an agile context is currently done, what the developers believed to be necessary information to perform the EE and what information that an EE tool should provide. This was done by conducting interviews with three agile developing teams at a German company, SAP SE. It was concluded that the interviewees believed that the accuracy of the estimation highly depended on developer's knowledge and experience and the complexity and impact of the changes. Furthermore, some of the factors in the EE process would benefit by introducing a tool, such as complexity and impact of the change.

In the interviews, they were asked questions about which aspect the developers believed to be relevant for a tool to have. Eight out of the eleven interviewees said that IA (what impact on the system will this change have) would be useful, if a tool could provide that information. They also found that visualizing that information would improve the efficiency of the estimation process. This validates the needs of our impact analysis requirements and shows that it would be useful to add the visualization in a future tool. They found that the persons that did not want to include a tool in their estimations had this opinion because they thought that it would decrease the discussions within the team.

They stated that they preferred a common estimate and understanding instead of a tool. This is not what their tool is intended for however, it is supposed to be used as a basis for a discussion to make the estimations better and to know what parts of the code to discuss. This relates to our tool which has the same purpose.

6.3 Threats to Validity

Most of the questions that were identified from the interviews are only validated to be relevant in the specific context of Praqma. Some were validated in the previous section because other researchers have found similar questions. Questions that were not found in related studies are based on the interviews of three workers at Praqma and they are not representative for the whole field of computer science, but the workers can be seen as having experience and knowledge in most parts of software development. It can be argued that the questions that regard impact analysis and technical debt, in fact, are relevant in a more general context because there exists a research field about them.

The investigation of CodeScene was done from a perspective acquired from the questions and literature which can have led to an analysis of CodeScene from a point of view that it was not intended for. CodeScene's core concept is that change is a measure of where there is technical debt and by looking at where there have been the most changes one knows where to look deeper for problems.

All the data that was used for the implementation of the proof-of-concept tool were manually retrieved from other tools, meaning that the tool did not perform any analysis in real time. Therefore, it is not known how this tool would perform in an actual application. Analyzing code is a process that takes time, running a full analysis of the whole system every day to have fresh data might not be possible because a system with thousands of files and millions of lines of code might take more than a day to analyze. Running the analysis on only the estimated impact set (EIS) that the tool produced in this thesis gave no indication that it would take too long time to analyze, even if the tool would perform the analysis on demand. There is, however, a risk that even that would take too long time. The files that were used in this thesis can be considered as small and the EIS was never larger than 20 files, which led to this not being a problem for us.

6.4 Future Work

We have presented questions asked in software development. This list is not complete and further explorations are needed. By changing the context and conduction more interviews, perhaps supplemented by observations, one can find additional questions. We have also presented questions that we did not explore in this thesis.

The prototype has room for improvements. It can be complemented by adding more metrics or changed to different metrics that fit a certain context. While investigating TD Visualization functionality could also be added. For example could functions that show an overview of a whole project, where one can choose to base the view on certain metrics, be implemented.

One thing that could have been a whole new thesis would be a more in-depth investigation and discussion of how the parameters of technical debt affect effort estimation. It would be interesting to know which parameters are the best to present, e.g is it better to know the cyclomatic complexity or code coverage of a class? It may be that one of the parameters affects the cost much more than the rest.

In the thesis, we did not validate that the metrics shown to the developer during the demonstration of the tool were accurate. In the future, it would be of interest to use a tool like ours or some similar tool to investigate if the metrics given affect the decisions made in projects. In general, there is a need to investigate further how mining software repositories can be used as we have seen that it is very dependent on the context and practices in projects.

Because of the way our tool is intended to work, it would make sense to extend it to take methods as input instead of files. If one is able to specify which methods that one should change in a story the tool should be able to more accurately tell where the dependencies are. Implementing this would require major changes, not just to the functionality but to the GUI as well, e.g the file system chooser would need a complete rework.

Throughout this thesis, there has been a desire to have the tool to recommend when the code needs to be refactored. All stakeholders to this project were interested in such functionality but there was not enough time to investigate it. The idea was to have a threshold on the technical debt and when it raised above a certain value it would recommend that a refactorization is needed. Future researchers could use the metrics for TD used in this thesis and try different threshold to try to find a value where the code would benefit from being refactored.

Chapter 7
Conclusions

The biggest areas that developers had questions in were impact analysis, testing and technical debt. From the interviews it was clear that developers do not believe that they need to perform an impact analysis. This was contradicted by most of the questions they had being answered by performing an impact analysis. This was a direct consequence of the interview subjects not having any experience of doing an IA. To do a good impact analysis it is important to look at more than only dependencies meaning that there is a need to do an effort estimation for a task. From literature studies and from the interviews we have identified that technical debt impacts the cost of implementing a change. Implementing a change in a part of the code that has less technical debt is easier. Therefore, values on technical debt (TD) can be used to indicate the cost and improve the gut feeling of the estimation. Getting values on cyclomatic complexity, lines of code and code coverage gives a better gut feeling on whether or not a change will be hard to implement.

CodeScene's functionality was studied by exploring which of the use cases/requirements it could satisfy, and more important which it could not. Three other tools, STAN, EclEmma and JDeodorant, were also studied, as some of the functionality was not supported by CodeScene, but instead found in those. In impact analysis CodeScene was able to find dependencies on both class level and function level. Indirect dependendencies are also shown, but not explicitly stated. Instead CodeScene provide the user with degree of coupling, which is a parameter that states how many times the two classes have been committed together. All dependencies are bi-directional, meaning that a two classes are always dependent of each other. CodeScene can therefore not find dependencies that are only present in one way. In the area of TD CodeScene can provide us with quantification of TD parameters like long method (by showing lines of code) and overly complex code (cyclomatic complexity). Code coverage was not supported at all and had to be taken from EclEmma.

We have shown by investigating CodeScene, STAN, EclEmma and JDeodorant that MSR- and static analysis-tools can provide us with all information needed to perform an impact analysis, it is just not structured in a suitable way. Therefore, a proof-of-concept tool was implemented with the purpose of showing that, if the data was presented in a suitable way, it could be used as an impact analysis tool. Furthermore, adding TD parameters of the affected files in the impact set will enable a data driven discussion of effort estimation. The tool was presented to developers who confirmed that it correlated with their gut feeling but this will need more research and be tried in a real project with active developers. We can only guess that introducing this tool would help with the estimations.

List of Figures

List of Tables

Bibliography

[1] Codescene. `https://codescene.io/`. Accessed: 2018-07-05.

[2] Eclemma - jacoco java code coverage library. `http://www.eclemma.org/jacoco/`. Accessed: 2018-03-02.

[3] Jdeodorant. `https://github.com/tsantalis/JDeodorant`. Accessed: 2018-02-20.

[4] *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[5] D3: Data-driven documents. `https://github.com/d3/d3`, 2018.

[6] Robert S. Arnold and Shawn A. Bohner. *Software Change Impact Analysis*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.

[7] Ward Cunningham. The wycash portfolio management system. *SIGPLAN OOPS Mess.*, 4(2):29–30, December 1992.

[8] M. A. Daniels. *Principles of Configuration Managment*. Advanced Applications Consultants, Incorporated, 1985.

[9] Philip W.L. Fong. Reading a computer science research paper. *SIGCSE Bull.*, 41(2):138–140, June 2009.

[10] Thomas Fritz and Gail C. Murphy. Using information fragments to answer the questions developers ask. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 175–184, New York, NY, USA, 2010. ACM.

[11] Martin Höst, Björn Regnell, and Per Runeson. *Att genomföra examensarbete*. Studentlitteratur AB, 2006.

[12] A. J. Ko, R. DeLine, and G. Venolia. Information needs in collocated software development teams. In *29th International Conference on Software Engineering (ICSE'07)*, pages 344–353, May 2007.

[13] Thomas D. LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: A study of developer work habits. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 492–501, New York, NY, USA, 2006. ACM.

[14] Zengyang Li, Paris Avgeriou, and Peng Liang. A systematic mapping study on technical debt and its management. *Journal of Systems and Software*, 101:193 – 220, 2015.

[15] Shari Lawrence Pfleeger. *Software Engineering: The Production of Quality Software*. Macmillan Publishing Co., Inc., Indianapolis, IN, USA, 1991.

[16] Praqma. Git phlow. `https://github.com/Praqma/git-phlow`, 2017.

[17] Praqma. pretested-integration-plugin. `https://github.com/Praqma/pretested-integration-plugin`, 2017.

[18] J. Sillito, G. C. Murphy, and K. De Volder. Asking and answering questions during a programming change task. *IEEE Transactions on Software Engineering*, 34(4):434–451, July 2008.

[19] Odysseus Software. Stan - structure analysis for java. `https://stan4j.com/papers/stan-whitepaper.pdf`, Fall. 2008. Accessed: 2018-03-05.

[20] B. Tanveer, L. Guzmán, and U. M. Engel. Understanding and improving effort estimation in agile software development x2014;an industrial case study. In *2016 IEEE/ACM International Conference on Software and System Processes (ICSSP)*, pages 41–50, May 2016.

[21] A. Tornhill. *Your Code as a Crime Scene*. Pragmatic programmers. Pragmatic Bookshelf, 2015.

[22] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, June 2005.

[23] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, June 2005.

Appendices

Appendix A
Interview Guide

A.1 Starting questions

These starting questions will provide a context to the subject and serve as a friendly "get to know" each other dialogue.

1. What can you tell us about yourself?
2. What is your role at the company? Developer, tester, architect, PO, PM, CM?
3. How long have you been at this company?
4. What is your previous experience? Developer, tester, architect, PO, PM, CM? Education?
5. How familiar are you with CM activities? Such as CSA? What do you think CSA is?

A.2 Main interview questions

In this part we tailor the interview depending on the subjects experience and interests. We have some main topics that will start the discussion and then, depending on the answers, decide what to focus on. The questions aims at finding both questions where there is no known and where there is an easy solution.

A.2.1 General questions

These questions is meant to start a general discussion where more questions that we might think of during the interview also will be answered. The listed items are guidelines of what we wish to come up with by asking that question. The context for the interview is that we would like the subject to answer within his or hers specific area of expertise.

1. In which way are you involved with the software evolution?
    - What information do you need to perform that task?
    - How do you go about solving it?
2. Can you think of a problem when you are executing a change task which you don't know how to solve?
    - In what way would solving it help you with your work?
    - What is the downside of not being able to get the information?
3. Have you had any specific change task problems that you've encountered and solved?
    - What was the problem?
    - How do you go about solving it?
    - Do you ask colleagues?
    - Do you google it?
    - Do you use any specific tools?
4. Have you recently asked your co-workers any questions concerning a change task?
    - What?
    - What triggered the question?
    - Did you get what you were looking for?
    - Do you think there is a more efficient way to get the answers?
5. Describe a task in the change process you solve with ease in your daily work.
    - Same follow up questions as 2.

A.2.2 Specific area questions

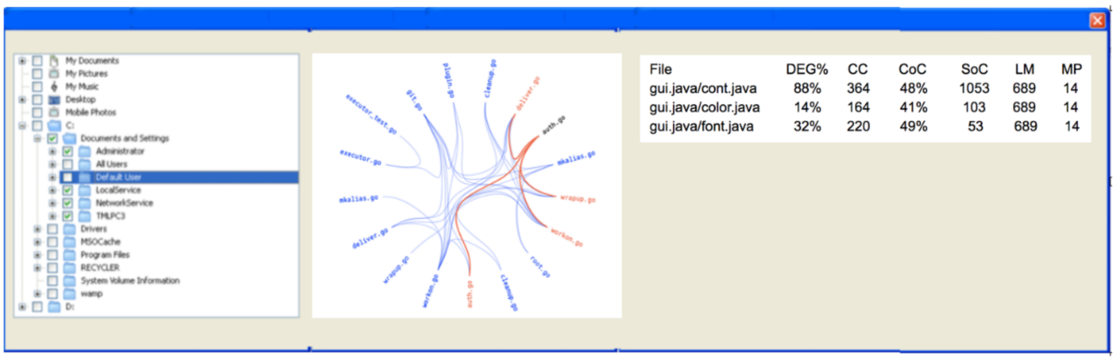
1. Are you manually storing information about CI:s (vital artifacts to you project)?
    - Such as author, traceability, time to finish a task etc...
2. Are you using repository mining to analyze commit history? Why/why not?
    - Repository mining is a technique one uses to extract information from different sources such as commit history, bug reports, mailing list archives
3. Do you perform any impact analysis?
    - When do you do it?
    - What information do you need?

- How do you do it?

4. How do you plan stories?
    - What do you need for cost estimation?
    - Who should do what?
    - How do you prioritize?
5. What major redesign to the source code have you done/thought of lately?
    - What triggered the thought?
    - What do you need to know to execute it?
6. What information do you need to know about tests?
    - Test coverage? Fully covered tests?
    - How do you know what to test?
7. What information do you need when you write a change request?
    - What was the CR about?
    - How did you solve it?
8. What information do you need when you implement a change request?

Appendix B
Mock-up of Prototype

This is an application that is supposed to show how information could be displayed in CodeScene. The best would be to extend the existing functionality, but with our current time frame and resources (three weeks) that would not be possible. Another solution could be to "modify" the files that CodeScene uses to display. That would include the same parsing of the files as the suggested implementation, but we would be limited to the current functionality and therefore not be able to display the information as desired.



File	DEG%	CC	CoC	SoC	LM	MP
gui.java/cont.java	88%	364	48%	1053	689	14
gui.java/color.java	14%	164	41%	103	689	14
gui.java/font.java	32%	220	49%	53	689	14

Figure B.1: The Mock-Up