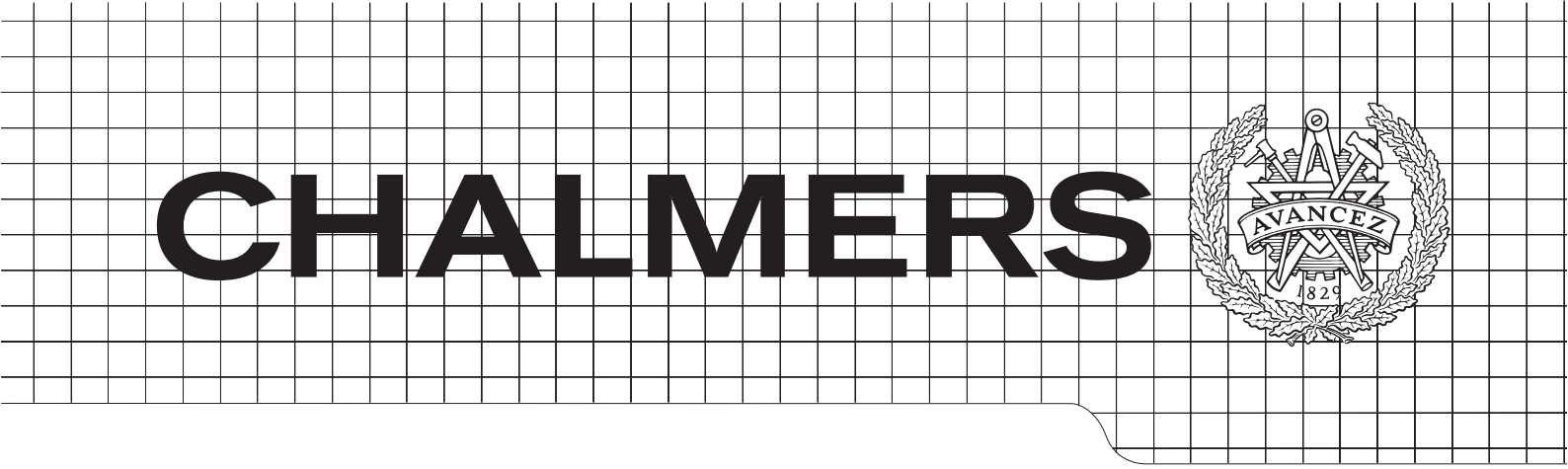

Finding Architectural Debt in Historical Data

Master of Science Thesis in Software Engineering

JOHAN GRUNDÈN

BJÖRN LEXELL

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Gothenburg, Sweden, June 2014

Finding Architectural Debt in Historical Data

Johan Grundén
Björn Lexell

Examiner: Mattias Tichy
Supervisor: Antonio Martini

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Gothenburg, Sweden, June 2014

Abstract

The metaphor Technical Debt (TD) is the description of a sacrifice made in the software development in order to reach a short term goal. For example, implementing a sub-optimal solution in a software product in order to meet a deadline. TD can be created both intentionally and unintentionally and are often hard to identify. This is especially the case when the debts exist in the architecture since they are not as visible as for example badly written code. When left unidentified, the debts are accumulating in unexpected costs such as higher maintenance but more importantly increased lead time in new development. And after a time, expensive and comprehensive refactoring activities are needed. It is often the case that budget constraints prohibit complete refactoring activities. Therefore, it is necessary to focus on fixing the problems that are the worst, i.e. prioritizing the debts. This master thesis has conducted a case study at Ericsson with the goal to find methods that can both identify and prioritize Architectural Technical Debts (ATD). The results from this work includes a Measurement System (MS) developed by the ISO standard 15939 which successfully identifies ATD:s in the form of non-allowed dependencies. Additionally, the MS prioritizes the dependencies based on how high risk they have of being difficult-to-maintain.

Acknowledgements

We want to above all thank our supervisor Antonio Martini for all ideas and the dedication showed throughout all the steps of our research but also for keeping us on track and picking us up in times of trouble. Furthermore we also want to thank Peter, Jesper, Mattias and Patrik at Ericsson for teaching us everything we needed to know about the company and for the time they put aside of work to help us out. Additional thanks to Patrik for letting us conduct this thesis at Ericsson. Special gratitude goes out to the Architects, Anders, Jonas and Suxia who willingly and gladly participated and provided invaluable input for this thesis. Lastly we want to thank Vard Antinyan for your research and for all the time spent with us, completely without needing to, to understand a cornerstone of this work.

Björn Lexell and Johan Grundén, Göteborg 21/5/14

Vocabulary

ATD: Architectural Technical Debt

Build file: Refers to the build.spec file used in the case company that contains all the dependencies to other components needed to build the specific component. Similar to a make file.

DSL: Domain Specific Language

ICT: Information and Communications Technology

Impact estimation: The parts (for example components or source code files) of the software that are estimated to change in a feature development.

Interest: The amount of extra time or resources that have to spend on future development because of the Technical Debt.

LLC: Low Level Component, represents the lowest level of component in the architecture (called Software Unit (SwU) at the case company).

LS: Literature Search, the customized lightweight approach based on SLR used in the thesis.

MLC: Medium Level Component, represents a medium level component in the architecture. It consists of one to many LLC:s (MLC is called Software Block at the case company).

MS: The Measurement System developed in this thesis which identifies and prioritizes non-allowed dependencies.

Non-allowed dependencies: Refers to dependencies between components in the architecture that are forbidden. These dependencies exist in low- and medium level components.

Principal: The cost of fixing a Technical Debt, i.e. removing it from the system.

RCS: Revision Control System RMS: Risk Measurement System, refers to the system developed by Antinyan et. al. [1].

SLR: Systematic literature review

Contents

1

Introduction

Technical Debt (TD) addresses the debt that software developers gain by not doing the quite right solution or producing incomplete development artifacts [2]. In 2010, the TD was estimated to cost the global software industry 500 billion US Dollars. Today, it is still considered to be a severe problem in software development around the world [3]. As the debt increases in software systems, even more resources have to be spent on paying off the interest, i.e. the cost for having the debt. TD is often used for trying to reach short-term goals in sacrifice of long-term [4]. For example, time constraints before a deadline could affect solutions to be *"not optimal"* in order to deliver on time. There also exist different types of TD that are associated with different artifacts of software development. A very problematic type of TD is Architectural Technical Debt (ATD) which includes debts that can be found in the architecture. Studies have been made showing that ATD can be very hard to identify [5] [6]. For example, ATD requires a higher level of abstraction for identification compared to debts that can be found directly in the source code [5]. This means, since the abstraction level for ATDs is not as clear as for source code TD, they are harder to visualize. Moreover, the effects of ATD are not as visible to end-users as for example software defects as they can't be traced to visual components. This causes problems for example when architects are creating arguments for justifying the need of refactoring in the software [7]. Martini, Bosch and Chaudron [6] concludes that non-allowed dependencies between components in the architecture are a severe type of ATD due to the fact that these dependencies might cause ripple effects when changes in the source code are made. This makes the effects of this type of ATD hard to predict. However, identifying and visualizing the ATD is just the first step of solving the problem. When faced with a large amount of ATD:s to solve, a management strategy is needed since budget and time constraints often does not allow to resolve everything at once [8]. Therefore, a prioritization strategy is needed in order to know which ATD:s that are best to resolve first. By both being able to identify and prioritize ATD, the field of software engineering

becomes closer to solving a crucial problem for today's software industry. This thesis address the problems of identifying and prioritizing ATD:s by conducting a case study at one of the largest ICT companies in the world, Ericsson.

1.1 Purpose

The purpose of the thesis is to find a way to identify ATD items in a real life context by designing a method that could be useful for the case company but also with the potential to generalize it to similar or potentially completely different companies. The purpose of the study is also to be able to prioritize ATD in order to get a decision basis for where to start the refactoring work.

1.2 Scope and limitations

ATD can exist in many different ways and in different artifacts of software development. However, this thesis only focuses on non-allowed dependencies as the potential source of an ATD item based on it being the most severe types [6] (theory about non-allowed dependencies and why they are important to address can also be found in Section 2.3). Therefore, the scope of the thesis is to measure non-allowed dependencies in order to identify possible ATD and to measure the risk of the dependencies to be able to prioritize them. The scope is to focus on the dependencies between components in the middle and lower level of the architecture. Therefore, whenever the term *"non-allowed dependencies"* is stated in this thesis, it reflects dependencies between these components.

This thesis is limited to a specific part of the software systems at Ericsson and will not be carried out on any other parts or companies. It is also only concerned with product code and therefore excludes test code.

In order to identify ATD in the form of non-allowed dependencies, the thesis have set up rules that the architecture must follow. However, the thesis is not aiming to extract all architectural rules for a given system, the identified ones could be a subset of the rules. The thesis does not aim to address all rules but rather to focus on the ones that have been proven to be important by the stakeholders. As a result the identified non-allowed dependencies in a system could be a subset of all non-allowed dependencies.

1.3 Research questions

In order to achieve the purpose the thesis will try to answer the following research questions:

RQ1 - How can Architectural Technical Debt in the form of non-allowed dependencies be identified?

RQ2 - How can Architectural Technical Debt in the form of non-allowed dependencies be prioritized?

1.4 Main contributions

The main contributions of this study is:

- A proposed method that analyses features within a product to identify non allowed dependencies. This method answers RQ 1.
- A Measurement System (MS) developed according the ISO standard 15939:2007 which identifies and prioritizes non-allowed dependencies within a system. This contribution answers both RQ 1 and 2.
- A technique for prioritization implemented in the MS which prioritizes non-allowed dependencies based on the risk of adding extra effort for the software development. This concept answers RQ 2.

1.5 Thesis outline

The introduction section is followed by a more elaborated chapter about the background for the terms Technical Debt and Architectural Technical Debt. A section describing the focused type of ATD namely non-allowed dependencies and related work is also presented in more detail. After the background the method chapter will explain how the identification and prioritization methods, including the Measurement System following the ISO standard 15939:2007 [9], were developed and carried out. The result section will present the contributions of the thesis followed by a discussion about the results and what the contributions adds to research. It will also discuss the applicability of the results as well as suggestions for future work. The final chapter will conclude the thesis and present the major findings.

2

Background

This chapter presents the background about Technical Debt as well as Architectural Technical Debt. It also presents theory about non allowed dependencies and more details about the research on prioritization of TD. The Risk Measurement System (RMS) used in the thesis is described as well.

2.1 Technical Debt

Change and evolution is an inevitable fact in today's software industry. Higher and higher demands on more complex systems makes it challenging to manage software projects in a successful manner. Along with the increasing popularity of agile development comes increasing demands on short release cycles and a rapid respond to change to provide value to the customers. As a consequence shortcuts are often taken in development to meet these demands [4]. These shortcuts might meet short term goals but could inflict negatively on future development and maintenance if not addressed properly. This phenomenon is called Technical Debt (TD).

2.1.1 History of TD

The metaphor Technical Debt is a situation where increasing software development costs arises from inadequate development [3]. It refers to the financial world where going into debt means paying interest on a taken loan. In software development it can be seen as that the higher development cost is similar to obtain interest due to the fact that a shortcut has been made at some point [2]. The metaphor was first mentioned by Ward Cunningham [10] where he drew the conclusion of making shortcuts in the source code were similar to going into a financial debt. The main reason for the metaphor was to spell out the need for refactoring to stakeholders that lacked the technical expertise. After this, TD gained increased popularity by both practitioners and researchers, but

there was still a lack of a clear definition [11]. Several studies has been made since then in order to unify the metaphor into definitions and terms [3] [12] [13].

2.1.2 Intentional and unintentional TD

The studies made on TD shows that it can be obtained both intentionally and unintentionally [4]. Intentional TD is based on strategic decisions that are aimed at reaching a certain objective. For example, it could be an intentional shortcut taken in the development to deliver a feature to a customer on time and deal with the consequences of refactoring later. Unintentional TD is often incurred when there is a lack of knowledge or experience. A developer might not be aware of the best solution for a certain task and in that case a TD is incurred that has to be paid back when a more optimal solution is needed. Furthermore, the incurred debt can be seen as reckless or prudent depending on how it is managed [14]. Figure 2.1 describes this taxonomy.

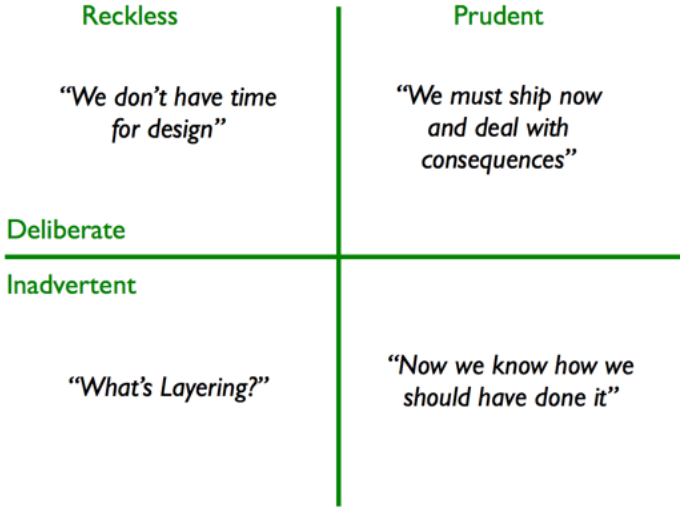

Figure 2.1: Fowler's Technical Debt quadrant [14].

2.1.3 Different types of TD

It is important to distinguish between different dimensions of technical debt due to the fact that it can exist in different ways [3]. For example, a source code debt is gained when the code is of low quality, i.e. it is subject to code decay [15]. This decay leads to more and more complex code and the understandability decreases. Other types of debt include architectural and design debts. The most common definition of this dimension is architectural solutions that are non-optimal [3]. That is, a shortcut has been made in the architecture or it has been left unmanaged over time which for example results in architectural erosion. Additionally, testing, documentation and work processes can also be a subject to debts. All these dimensions of technical debts are incurred in different

ways and it is crucial to separate them since the consequences varies as well as the strategies for repaying.

2.1.4 Identification of TD

A common and crucial condition for managing different types of TD is to properly identify and visualize the items. [4]. There exists many automatic tools for identifying technical debts [16]. For example, one approach is to look for code smells to discover breaches in object-oriented design practices which lowers quality of the software. To more accurately identify TD it is important to take the occurrence of changes into consideration. If a class is identified as a TD, but never changed it does not produce any extra effort, i.e. interest. If no interest is incurred then it is not a TD [13].

As stated in the previous paragraph identification and visualization of TD are crucial for providing information to management decisions. However, tools that focus on source code runs the risk of not being able to identifying different types of debt [11]. Studies has shown that there is a small overlap in what tools and humans detect [17]. This is especially true for Architectural technical debt (ATD). Many practitioners [7] report that since these types of debts are less visible for the customer, they gain less attention and are not prioritized since they are considered to have very little or none customer value. The ATD:s can have more severe effects than source code TD such as higher interests over time on software quality and maintainability [5].

2.2 Architectural Technical Debt

The background study conducted for this thesis shows a gap in understanding and definition of Architectural Technical Debt which is proven by the limited amount of work and published papers. The architecture is recognized to play a major role for the evolution of large software systems [11] and the development of user features should be done with close consideration of the architecture [4]. However, there is a lack of well-established approaches to identify and manage ATD and the problem domain is not as well-known as for example code debt [5]. Previous work on TD, identifying and managing, is mostly focused on source code and based on reckless development or inexperienced developers [4].

2.2.1 Similar terms for ATD

Apart from the term Architectural Technical Debt (also known as architectural debt) the literature review came across some related terms. Design debt for example is according to Zazworka et. al. [18] another definition, one other paper mentions the same term without the explicit connection to ATD but describing the same phenomenon [17]. Mo et. al. [5] talks about architectural decay instances which origin from the architecture beginning to drift or erode. These instances will most likely, if not properly addressed, develop into ATD.

2.2.2 Definition of ATD

The definition for ATD is a sub-optimal solution [3], when a software design cannot satisfy the intended purpose [18], an imperfection in the architecture that have a negative impact on maintenance [17] or when design decisions affects the life cycle properties of a system [5]. Architecture decay is additionally associated with implementation not matching the original architecture, increased resistance to change or the growing effect of negative changes on the quality of a software product [19] [20].

ATD or architectural decay originates from the rapid evolution of software where the architecture cannot follow [13]. In contradiction to user features, which are easy to visualize, architectural changes are mostly invisible but based on intentional decisions. Management and re-work estimations of these debt items are dependent on proper identification [4]. In especially iterative development such as SCRUM, architectural changes are hard to quantify and often assigned zero-value for the customer. This falsifies the view of providing value to the customer as it is only seen as an increased development cost when it in fact could speed up the implementation of new features [5].

2.2.3 Identification of ATD

The identification of ATD has been done with static code tools such as code smells and Automatic Static Analysis issues on the source code [16] with or without a combination of adding interviews [17]. However, some papers propose metrics or approaches that are considering the architecture directly without the use of source code [4] [5]. Mo et. al. [5] suggest transferring architectural models into extended augmented constraint network (EACN) which can be used to determine pairwise dependency relations for automatic detection of decay instances. ATD could also be identified by looking at Modularity violations or Design patterns and grime buildup [16]. Modularity violations happens when change in one module is dependent on change in another, this breaks the principle of modular design. Design patterns are commonly used and recognized to improve maintainability and architectural design. However, changes could lead to code ending up outside the pattern which is known as grime. Furthermore, when changes are introduced, the risk increases for that a design pattern is not suitable to the change, also known as rot. There are more symptoms connected to a higher level of architectural abstraction than source code implementation mentioned as architectural bad smells [21]. They concern architectural elements such as components, connectors, and concerns. Some papers stresses the importance of connecting source level elements with these components and the dependencies between them to properly identify ATD [4] [5]. To answer the first research question, the aim of this thesis is to connect source level elements to components and by looking at modularity violations between them through static analysis to reveal ATD.

2.2.4 Addressing and resolvning ATD

During the literature review a common solution for addressing ATD was found to be refactoring or re-architecting [4] [18] [22] [23]. This solution is often done when developers recognize that an improvement to the architecture is necessary for the maintenance and continued evolution of a system or when it cannot fulfil its purpose anymore. However, decisions about paying of an ATD and the solution to the problem should be done by the architect [21] [22]. One paper states the importance of involving people from the project to decide rework efforts associated with a debt item [4]. Another paper points out business involvement when handling technical debt [22] as it is common for developers or architects *"gold plate"* the solution [11]. This means making the architecture to adaptive or making a solution more *"technically elegant"* than it needs to be. More time is spent then necessary and business consideration should decide when a solution is good enough to maximize the value gained from it.

2.2.5 Different types of ATD

A critical point for when it comes to understanding ATD is to understand the different types that can exist. For example, a common one is code duplication. This type of ATD is also referred to as *"Scattered functionality"* and exists when multiple parts of the software have responsibility over the same matter [21] [5]. Garcia et. al. [21] concludes that this ATD is in direct violation of the design principle of separation of concerns and also states that it can have impacts on the modifiability, testability, reusability and understandability of the system. Many researchers has put a lot of effort into automating tools that can both identify and remove this type of ATD [24]. Non-allowed dependencies are another type of ATD which is recognized as very problematic [6] and will be explained in detail in the next section.

2.3 Non-allowed dependencies between components

A non-allowed dependency is a dependency that lies in conflict with the intended architecture and design of the system [6]. By introducing such dependencies, the architecture of the system will be subject to decay and degradation, i.e. an ATD is introduced in the system [25]. Also, these dependencies might cause ripple effects to other parts of the system that are not known [6]. These effects can be extra changes needed in other components due to the existence of the dependency. If non-allowed dependencies exists that are unknown for the designers, then the effects can be that additional unexpected time has to be put into the unforeseen extra changes. This might cause time estimations to be inaccurate and delay releases of features or make the maintainability of the system worse. There can be many reasons for introducing this type of ATD. For example, the developers might not be aware of that a new dependency is not allowed. As explained in the previous sections of this chapter, time constraints for deadlines can also be a reason for introducing a debt. Hence, decisions might be made to introduce non-allowed dependencies as a part of a shortcut to meet a deadline for a feature.

A common type of dependency is Structural Dependencies where a component is dependent of another component through a method call, class extension or a class aggregation for example [26]. This means that the component which is the dependent one relies on the other component to function properly. Any changes introduced in the other component might produce changes in the dependent component due to the fact that it uses functionality from the other one [27]. So, if the dependency is not allowed, then the dependency could produce extra effort in the form of possible extra updates in the dependent component due to changes in the other component. This can be seen as paying interest on the ATD. The effect becomes even worse if the dependency is circular, i.e. both components are dependent on each other [25]. Per definition, changes in any component would increase the possibility of introducing extra changes in the other one due to the fact that both components are dependent on each other. It could even be the case that additional changes are triggered because of the first one and so on. For a graphic visualization of these ATD types, see Figure 2.2

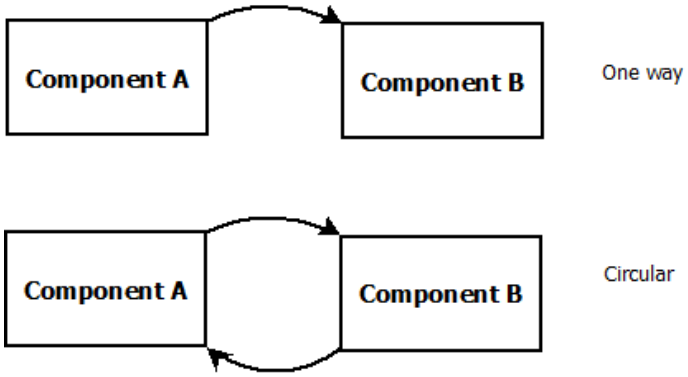

Figure 2.2: A visualization with examples of the effect of the dependencies described.

The extra efforts of these dependencies can be traced to more than just changes in the source code. For instance, more efforts needs to be taken into account when creating tests [25]. Consider that Component A from Figure 2.2 does not have a dependency to Component B initially. When testing A, there is only need for building a test suite for component A. If the dependency to B is introduced, then the test suite for A has to include the test suite for B as well. Due to the fact that this type of ATD is recognized to be very problematic and produce side effects that can be hard to foresee, this thesis will focus on identifying and prioritizing non-allowed dependencies. Methods need to be established in order to systematically achieve this.

2.4 Prioritization of Technical Debt

As with a financial debt, TD has the benefit of supporting short-term goals. But, the drawback is the cost of an accumulating interest over time and the debt also has to be paid back in the form of a principal cost at some point [22]. When an increasing amount

of debts are accumulated in the system, decisions has to be made on how to address the problem. Resource constraints on software projects often have the consequence of making these decisions very difficult since the budget does not allow for fixing every issue [8]. In order to make these decisions as good as possible, a prioritization technique is needed that can identify which debt has the worst effect on the software. Seaman et. al. [8] states that several approaches can be taken when managing the TD. For example, a cost-benefit analysis can be taken to prioritize debts according to how much interest they produce along with how much value that is gained if the principal is paid off (i.e. the cost of extinguish the debt completely). By managing the TD in this way the one with the highest interest and the least principal will be addressed first which is a very efficient managing technique, especially when resource constraints poses as a problem. However, the same study also concludes that there is always the risk of *"overdoing"* the decision making process meaning that the amount of effort is not worth the amount of value that the result provides. Often, it can be hard to estimate the amount of interest and principal that a certain TD generates. This has been shown by Zazworka et. al. [17] where practitioners report that the estimations are indeed a tricky and time-consuming task to perform which spells out the need for automated tool support. Another prioritization approach would be to use the Analytic Hierarchy Process [8]. The process is a simple and effective way that uses pairwise comparisons on different alternatives in order to give each alternative a relative priority among the rest. When applying this process to TD decision making, the various alternatives would be the actual identified TD items. Moreover, when it comes to prioritizing amongst interest and or/principal of TD items, metrics has to be chosen to represent these terms as well. For example, for severe ATD items such as non-allowed dependencies, which is a type of modularity violation, studies has been made that shows that the change proneness is closely related to this ATD [16], which could be seen as a possibly metric Another example of a metric that can be used in the prioritization of interest is a Risk Measurement System (RMS) developed by Antinyan et. al. [1] that can identify source code files that runs the risk of for example being difficult-to-maintain. So, if a file is identified as high-risk it is also identified to generate risk of generating a greater amount of effort to maintain in relation to other files. If the file would be included in a non-allowed dependency for example, then the effort to maintain it would be a part of the interest that the ATD generates. Therefore, in order to address the second research question, the focus in this thesis will be to prioritize the ATD:s after this definition. The RMS is explained in more detail in the following section.

2.5 Risk Measurement System

As the complexity in software products increases, it becomes harder to manage the associated risks. Often the risks are not feasible to manage manually since the size of the products can be very large. For example, assessing millions of lines of code in a product is impossible for a manager for obvious reasons. Therefore, automated tools can be of a great help when trying to identify parts of the source code that are not optimal.

Antinyan et. al. [1] have developed an automated Risk Measurement System (RMS) to identify parts of the source code that are considered as risky. The actual risk is defined as:

> *"The likelihood that a source code file becomes fault prone, difficult-to-manage or difficult-to-maintain"*

The system have been developed with collaboration from both Ericsson AB and Volvo Group Technology and focuses on risky files in lean projects. To be able to measure the files, metrics are needed. The research behind this measurement system showed that two metrics are needed to calculate the relative risk of a file. Firstly, the complexity of the file is measured by McCabe's cyclomatic complexity for functions and files. The definition for McCabe's cyclomatic measurement of functions is the following:

> *"The number of linearly independent paths in the control flow graph of a function, measured by calculating the number of 'if', 'else', 'while', 'for', '||', '&&', 'switch', 'break', 'goto', 'return' and 'continue' tokens."*

The output of the measurement of the function is a value, M. When M $>$ 15 the function is defined as complex. When measuring the complexity on files instead of functions, McCabe's cyclomatic complexity (M) is summed for all functions in that file. With these two measurements, a ratio is calculated between the nr of complex functions and the total number of M in a file. The result is a metric; Effective_M% which represents the complexity of the file. However, a file that is complex does not necessary represent a risk if it is not modified. To take this into account the number of revisions of a file are measured by looking at the Revision Control System(RCS) ClearCase and count the amount of check-ins for a file in a given time period. This metric is called NR. To sum up, the relative risk is calculated by the following formula shown below:

$$\text{Relative risk} = \text{Effective\_M\%} * \text{NR}$$

Moreover, to be able to identify the files that are considered with the most risk, a threshold needs to be defined. This threshold needs to be calibrated to meet the properties of the product that is measured. For instance, the size and number of people working on the product are two properties that needs to be taken into account.

The measurement system has been evaluated by designers from both Ericsson AB and Volvo Group Trucks Technology and it proved that the system could identify all major risks. The system is currently deployed at a department at Ericsson AB were the designers get continuously updated information about the risky files.

3

Method

This chapter presents how the methodologies used in the thesis was developed and carried out. It also includes the research design which describes the reasoning behind doing an exploratory case study. Furthermore, the pre-study done at the company and the case context is presented along with the Literature Search (LS) conducted for Chapter 2, Background. This thesis consists of two different developed methods for identifying and prioritizing Architectural Technical Debt which are presented in different sections.

3.1 Research design

As explained in Chapter 2, ATD is a problem for many software projects today. This thesis aims to find methods for how to handle that problem. In order to find these, knowledge about how the problem behaves will be explored in a real life context. Runeson and Höst [28] explains that in situations like this, a case study is the most appropriate research method to use. For example, it is very flexible in the aspect of how the steps of the research method can to be conducted. Each step can be customizable and adapted to the specific conditions of the case. This benefits the thesis study since the authors started the research with very limited knowledge about the topic and the context. The design of the research could easily be adapted and changed as knowledge grew and the context became clearer. Due to the fact that the thesis workers had no knowledge of how to answer the research questions before hand, the case study was designed to be exploratory. The exploratory case study is aimed to *"seek new insights"* about the specific research problem. This design helps to systematically find out how ATD is a problem and to explore new ideas of how to solve it.

The research process of this case study is based on the principles of the 5 steps defined by Runeson and Höst [28]. The purpose of the steps in this thesis are that they should be stating a systematic approach on how to fulfill the goals of the study. It is

also important to notice that for this study, step 2-5 will be repeated iteratively which will be explained below.

1. *Case study design:* This involves setting up the objectives of the study. To be short and concise the objective is: to find methods that can identify and prioritize ATD in the form of non-allowed dependencies.

2. *Preparation for data collection:* Since the approach of this case study is to explore methods that can identify and prioritize ATD, this step and the following ones will be repeated for each method. For each method, a plan of how the method will work and what information needs to be measured will be established. The concept of the specific methods will be derived from knowledge generated from the pre study mentioned in Section 3.2 along with knowledge and input from the case company and Chalmers. The concepts may also rely on existing standards that defines processes for how to generate methods. This is the case for method 2 explained in Section 3.5. If any interviews would be needed to be conducted, they would be set up and planned in this stage. Also, other components such as source code or revision history needs to be identified if they are necessary to be measured by the specific method.

3. *Collecting data:* The purpose of this step is to realize the plan that has been created in step two. This involves conducting the necessary interviews, measure the components needed.

4. *Analysis of data:* As a part of the analysis step, validations and confirmations has to be done. The methods needs to be validated in a systematic way and be confirmed to be useful in order to draw valid conclusions about them. This will involve setting up an empirical validation method and conducting it by preferably interviewing stakeholders related to the specific method.

5. *Reporting:* As a final step, the methods will be reported through this thesis. This involves all steps from method generation to the analysis.

3.2 Pre study

The pre study for the thesis consisted of a few steps to start of the work, the first one was a Literature Search (LS) process which was followed by a study of the case context. Along with the LS an initial method of identifying ATD:s was formed based on ideas from literature and input and ideas from the department of Software Engineering at Chalmers. Furthermore, the thesis started out with limited knowledge about the case context at Ericsson. Therefore, there was a risk that the initial method, which has not been tested before, would not be applicable in the company. If that risk would occur, the thesis would not be able to produce any results in the form of ATD. In order to mitigate the risk, a second method was also developed when knowledge about the thesis subject and the company context had increased.

3.2.1 Literature Search

To gain knowledge and the necessary understanding required to carry out this thesis a literature search was conducted on academic literature available on TD and ATD, both on identifying and prioritizing. The LS was based on the principles of Systematic Literature Review, SLR [29], but due to time constraints a more lightweight and adapted process was used. Before the start of the thesis some papers had been acquired for the understanding about the problem domain. From these papers an initial set of search terms were derived; Technical debt and Architectural technical debt. The adapted version of SLR, the LS was developed to be as effective as possible and to fit within the planned time frame of two weeks. The process contained the following steps and were done in iterations:

1. Identify search terms. (For the first iteration this was done from the literature gained before the LS)
2. Search lib.chalmers.se and scholar.google.se for the identified terms.
3. Pick the first 10 results from the search.
4. Read the title and abstract, if relevant for the thesis include the paper and add it to a list of possible relevant literature.
5. Read intro and conclusion/discussion. This was done by both thesis authors to determine if the paper would be useful. If not accepted by both a discussion was held until an agreement could be made. If the paper was excluded at this stage a short comment about the paper was noted.
6. Each identified paper was read through more rigorously with the goal to identify new terms, key findings and relevant references. A summary of each paper was written down.
7. *"Snowballing"*, i.e. a common method for continuously pursuing references of references, was made on the relevant sources of the obtained papers.
8. If new search terms were derived from the papers the process was started over from step 2.

The literature search ended up with a total of 30 papers and from the first iteration 5 new terms were found; Architectural Decay, Software Architectural Design, Identifying Technical Debt, Identifying Architectural Technical Debt and Design Debt. The second iteration generated 3 terms; architectural deterioration, design metaphor and architectural bad smells. During the thesis the literature search continued and additional papers were found to support new theory, for example research on non allowed dependencies, the definition and form of expression needed to be considered which resulted in several papers: [6] [26] [27]. As a part of the thesis focused on deliver a measurement system as well as the validation of it, papers about that kind of research were also added, for example [30] and [9].

3.3 Case context

The conducted case study took place at Ericsson which is a world leading ICT-company. They are focused on supplying equipment, software and services for mobile communication. Currently the company has more than 110 000 employees that are situated all over the world.

More specifically, the studied case went on at a site at Lindholmen, Gothenburg, which has around 2000 employees. The specific department for the study works with software for Radio base stations and their product has been in development for half a decade. A couple of years ago they changed their working process to be according to the lean and agile methodology. The developers are about one hundred and are comprised into 15 cross-functional teams. These teams develop features and are responsible for analyzing, designing, implementing and testing the feature. The main release time for the specific product is around 6 months and consists of many features developed by the teams. To support the cross-functional teams there are some other roles as well. For example, there exists 3 Architects which has the main responsibility to ensure that the architecture of the software is kept according to their intended one and to prevent it from erosion and decay. Each Architect has responsibility for its own part of the system and they are the ones who have the best knowledge about the architecture. However, they still have a collective responsibility for the whole system. Furthermore, there exists Technology Specialists that have expert knowledge in the area who are planning future work concerning product care and are often used for consulting when technical problems emerge. The thesis workers supervisors at Ericsson consisted of two software designers which are both scrum masters of cross-functional teams along with one Technology Specialist.

The software architecture of the product developed by the department where the study took place at is both layered and component-based. The product is mainly based of C/C++ source code files. These source files are divided into so called Software Units. They represents the lowest level of architectural components and are therefore called low level components or LLC:s throughout this thesis. The LLC:s are grouped together into Software Blocks which represents a medium level architectural component named MLC in the thesis. Figure 3.1 gives a visualization of this component-based architecture.

Moreover, the product employs a layered architecture as well. The architecture consists of 3 layers and one common platform layer which hold all the commonalities between the layers. Each LLC can be a part of one of the different layers and a MLC can have LLC:s from various layers. In Figure 3.2 the layered architecture is described.

Furthermore, the department has specific naming conventions for the source code. This helps keeping a clear structure of the code base. The following example shows the path for a specific source code file:

../product_name/exampleMLC/exampleLLC/examplefile.cc

As seen, the structure of the component-based architecture can be viewed in the path of the file. ExampleMLC refers to the specific medium level component and exampleLLC

refers to the specific low level component. In order to build the product, each LLC has its own build specification called a build.spec(build file in the thesis. This specification describes all the necessary dependencies to other components that are needed for building. The company uses a Revision Control System (RCS) to keep track of the different versions of the code along with each change made to it.

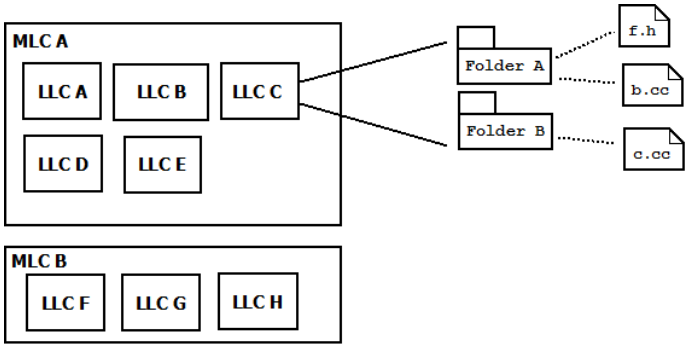

Figure 3.1: The component-based architecture of the product.

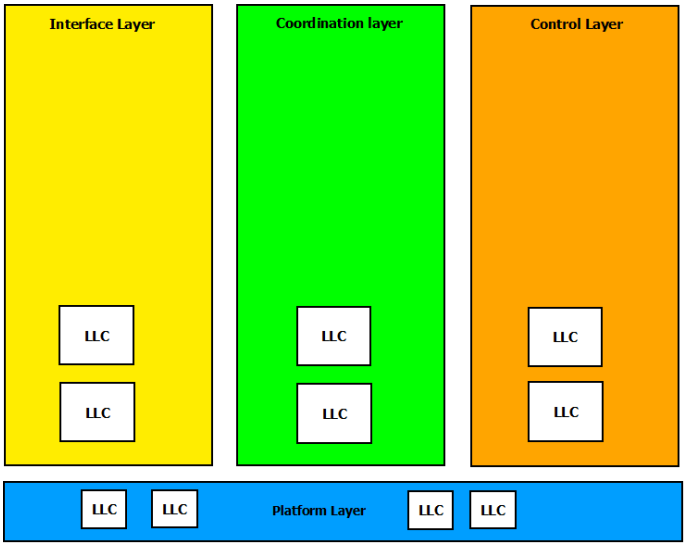

Figure 3.2: The layered architecture of the product.

3.4 Method 1 - Comparing feature impact estimations against changes

This method involves finding non-allowed dependencies by comparing estimations of which parts that should change in development, called impact estimations, with the actual results of the implementation. This would indicate possible ATD items since

an unknown dependency could cause the change in the extra component. The idea for this method originated from the department of Software Engineering at Chalmers which conducted a qualitative study involving the case company. The company develops several features which have documentation about impact estimations called design specifications. This method would look at the documentation for each future individually to identify ATD. This would provide a multiple set of data points to try the method on, which could be used to determine the effectiveness.

3.4.1 Planning

The planned approach for the method consisted in a series of steps. Firstly, documentation needed to be reviewed in order to determine the estimated parts in need of change for a specific feature. When impact estimations were identified, interviews with responsible developers would be held to determine how these estimations were done and by which criteria. The purpose of this was to gain an understanding about the quality of estimations to see if they could be used in the method but also the set up criteria for what was needed for the method overall.

The next planned step in the method was to get the actual changes made for the specific feature from the revision system. These estimations should be traced to the same component level as provided by the documentation so that they could be compared. Furthermore, the next step, comparison between estimations and implementation should be carried out. Potential mismatches, where a change affected a component that was not in the estimations, would be flagged as potential ATD. The unexpected change in that component could be the result of a non-allowed dependency in the system.

To validate the findings from this method a two-step interview process would be conducted. Firstly, an interview would be held with developers where they would be asked to describe ATD items they believed were introduced in the feature. The second interview would include showing the results from the thesis method along with the ATD proposed by the developer to discuss potential mismatches. True positives in the ATD items would be identified by comparing the results and false positives from the method would be able to be discarded by the developers. Items not discarded by the developers would prove an increment in knowledge as they were unknown. A comparison between all items identified, by both parties would work as a way to determine the effectiveness of this method. By doing the same procedure on multiple features even more data points could be collected to strengthen the result of the effectiveness.

The last step of the plan consists of calculating the risk of effort from the Risk Measurement System and connect it to the ATD item identified in the previous step. This would be done to get the risk for each violation in order to prioritize them against each other.

3.4.2 Performing and evaluating

According to the plan the method involved going through a lot of documentation about features. As knowledge grew about the way this was done within the company the

criteria required for a suitable feature was created. This lead to excluding features not providing the required initial estimations and resulted in a subset of suitable features. In order to determine the validity of the estimations two semi-constructed interviews were held with people at the same site. The ones not available for interviews, due to not being at the same location were contacted by mail. A couple of initial questions about the estimations for the feature were asked to determine if a telephone interview was needed. The result for each feature was summarized to determine if they were suitable for the method.

As the thesis played out it revealed that few features matched the criteria for the method and it was therefore concluded to be inefficient and unable to produce any valid results in this specific case context. This is explained in detail in Section 4.1. As the thesis was working with the different methods in parallel all efforts were directed towards the second method explained below.

3.5 Method 2 - Measuring and prioritizing non-allowed dependencies

During the pre-study a possible second method for identifying ATD was elicited from the increased knowledge about the system and the context. This method focused on analyzing the existing dependencies and to check them against rules about architecture in order to identify rule breakers/violations. When the non-allowed dependencies were found they were prioritized according to the risk of extra effort showing up when maintaining the components that were dependent on each other. The derived indicator from this method follows the steps of the ISO standard 15939:2007 which defines a process on how to create Measurement Systems (MS) to addresses a specific information need [9]. By following this standard it was ensured that the MS would provide the stakeholders with correct and valid information. The process is divided into a number of steps that define what is needed to be done in order to meet the requirements of the standard.

3.5.1 Step 1 - Establish and sustain measurement commitment

As a first step the scope of the measurement was defined to be; Measure non-allowed dependencies in order to identify possible ATD and to measure the risk of dependencies to be able to prioritize them. The stakeholders for the measurement were considered to be the architects because they would use the results as ground for refactoring as well as developers who could get indicators about introducing violations in the system. The measurement process involved the thesis writers, responsible for all the steps in the process as well as architects which provided rules and validation of the results. The supervisors at the company provided all access, computers and knowledge on where to find the necessary data needed for the thesis.

3.5.2 Step 2 - Plan the measurement process

This section presents all the steps, according to the ISO standard, carried out to determine the plan of the measurement process such as determine the organization unit, identifying information needs, data collection along with evaluation and validation of the results. A final section describes already existing tools for identifying violations.

3.5.2.1 Determine the organization unit

According to the ISO standard used in the thesis a plan was constructed for the necessary steps involved in the measurement process. Firstly the organization unit was determined to be the department on which the thesis was carried out on. It includes the people working on the product such as architects and development teams but also the actual product and the source code associated with it. The architects are responsible for the architecture of the product and were therefore selected to be the most appropriate ones to identify architectural rules and also the ones most suitable to validate the results of the thesis. The product consist of source code files that belongs to different LLC:s, each one of these units belongs to a MLC. The high level components of this system are not considered in this thesis as they represents an abstraction level not suitable for analyzing dependencies in the way proposed in this method. This is due to unclear connections between high level components and LLC.

3.5.2.2 Identify information needs

Secondly the information needs of the stakeholders for the measurements were identified. This resulted in 2 different information needs;

1. Need to see all the violations and to discover the unknown ones.
2. Need to have a prioritization of violations to know which ones to fix first.

This explains what the stakeholders want out of the indicators, namely a system that can identify and prioritize the non-allowed dependencies. Information need 1 is prioritized over number 2 since the latter relies on having the violations in order to prioritize. Based on the information needs and by following the standard the selected measurements for the thesis were documented as follows:

Name of measurement:	Violation.
Unit of measurement:	Build file, Rule document.
Formal definition:	A dependency between two LLC:s violating an established rule.
Method for data collection:	Interviews with architects to determine rules. Automatic extraction of dependencies from the source code file.
Related information need:	1

Name of measurement:	Severity of violation
Unit of measurement:	Source code and revision history.
Formal definition:	For circular dependencies: The aggregated risk of each involved LLC
	For one way dependencies: The risk of the source LLC in the dependency
Method for data collection:	Measure the risk for files in each involved LLC. Aggregate the risk for each file in a LLC
Related information need:	2

3.5.2.3 Data collection

The procedures for the data collection were planned to involve the following steps: identify rules by interviewing the Architects, create or find a tool which identifies dependencies in the system. Rules should if needed be implemented in the tool so it could check for violations. The RMS described in Section 2.5 would be used to calculate the risk of each source code file which would be manually aggregated for each LLC. The last planned step involved combining the measurements and to validate them with the architects.

3.5.2.4 Evaluation and validation

The criteria for evaluating the results of the measurement process were also established for the plan. Through constant communication with the architects, the thesis supervisor at Chalmers and the company each step should be discussed to identify possible improvements to the process. Data and information products should be confirmed to provide value to the company. The plan for validating the results involved which steps that should be taken at the end to validate the outcome of the measurement process. These steps would determine if results provided new information and/or the correct information about non-allowed dependencies. The purpose was also to verify that the risk prioritization represented the reality.

3.5.2.5 Study of available tools for identifying violations

Before the plan was carried out a study of available tools for measuring dependencies were conducted. This was done for two major reasons; to find out if our intended product existed on the market and to find out the possibility of extending an already existing product with the rules. The study showed similar tools capable of identifying dependencies. By the thesis workers best knowledge the following tools are the most similar:

- *Dependometer* [31]: An open-source tool that is used for analyzing if the architecture of a project is complying with the intended architectural rules. It uses a logical architecture description, defined through an xml file, which is used for verifying the dependency architecture in for example layers or sub components. At the moment the tool can analyze Java, C# and C++ projects. However, a downside with this tool is that it is very time consuming to set up the logical architecture and that it only visualizes dependencies between classes. This means that a lot of time has to be invested in the set-up phase.

- *SonarQube* [32]: Is another open-source tool that monitors many different types of technical debt. This tool is including a feature that is very similar to the one developed by this thesis. It is called an Architectural Rule Engine and it allows the user to set up architectural constraints about which dependencies that are not allowed. This can be done for dependencies between single classes as well as for whole packages. But, this feature is limited to projects that are written in .Net and Java and since the case project is written in C++ this tool was not applicable.

The thesis workers were not familiar with any of the tools found and therefore it would require a lot of time to understand them along with setting up and integrate them into the existing case. Also, possible additional time to develop plugins for these programs in order to implement the architectural rules was considered to be more time consuming than developing a tool from the scratch in a known programing language. Therefore, the decision was taken to develop a new tool that was specific for the case.

3.5.3 Step 3 - Perform the measurement process

This chapter presents the execution of the steps from the planning phase such as the different interviews with the architects about the rules but also the creation of the own-developed tool and the usage of the Risk Measurement System (RMS).

3.5.3.1 Individual rule interviews with architects

According to the established plan the first phase for the measurement process started out with interviews about architectural rules. A combination of the knowledge gained in the pre study along with known good architectural practices were used to create potential ideas for rules. An example of these practices can be circular dependencies which is explained in Section 2.3. The rules were a starting point for constructing interviews with the architects where the goal was to find out which specific ones that existed in their system. All three architects were contacted by email where they were asked for an interview about architectural rules within the system. Thereafter, semi-constructed interviews were held with each one of them at the company. The interviews started out with a short description of the thesis and the planned method. They were then asked to state rules about non-allowed dependencies between LLC:s. If they couldn't answer the question or when they had provided all the answers the potential rules from the pre study were presented followed by a discussion about them. The two later interviewees

were also presented with the rules from the previous one(s) to gain more insight about them. At the end they were asked once again if they could state any other rule that was missed. The reason for that was because they could have gotten more insight about what a rule could be from the examples provided by the thesis workers but also to ensure that no rule would be missed. The interviews were 45-60 minutes long and the results were summarized in an individual rule document with comments and general thoughts.

3.5.3.2 Common interview with architects and final rules

After the individual interviews the architects were contacted once again to decide on a final interview including all architects. The goal of the interview was to present the result and to reach a consensus about which rule(s) that would be used in the thesis. The rules would regard which dependencies that were not allowed between components. A summary of all the findings was presented for the architects and a discussion was held. When a consensus was reached a formal definition was made together with the interview subjects to remove the possibility of misunderstandings. The architects were asked to determine which rules they wanted the thesis to focus on and state the priority among them. The interview was one hour long and took place at the company. The rules that were derived during the interview were the following:

1. *A low level component (LLC) should not have a dependency to a LLC in another medium level component (MLC). However, dependencies to LLC:s in the Platform MLC are OK.* Any dependency between two LLC:s in different MLC:s should go through their corresponding Interface Unit (a type of low level component). The Platform MLC contains LLC:s which has common functionalities and these dependencies are therefore excluded from this rule.

2. *Two LLC:s should not have circular dependencies between them.* An important note is that only direct circular dependencies will be checked here. It could be the case that a circular dependency could exist between three or more LLC:s, but due to time constraints of the thesis, the focus was to check between two LLC:s firstly.

3. *LLC:s in Control Layer should not have dependencies between each other.* A LLC belongs to a specific layer inside the architecture and it shouldn't have a dependency to another LLC in the same layer (even if they are in the same MLC).

4. *A LLC in the platform should not have a dependency to a LLC outside the platform.* The platform consists of library units which should only provide services to others, they should not have a dependency to other components outside the platform.

5. *LLC:s in the Interface Layer should not have dependencies to the Control Layer.* Dependencies should be towards the coordination layer and not between these two.

6. *A LLC should not have a dependency that is directed towards another LLC:s /src folder.* The /src folder is a folder that exists in each LLC. It contains the internal code of the component and other LLC:s should not have dependencies directed

there. Dependencies should rather be directed to interface folders for that LLC (which are located in another folder than /src).

Out of the six rules the architects decided that the first four would be selected for the thesis as they provided the most value for the company.

3.5.3.3 The tool - RuleValidator

In order to measure the non-allowed dependencies a tool was created that could automatically identify rule violations. As a pre step before the creation of the tool the architectural mapping between L- and MLC:s was made to achieve the right level of abstraction needed for the rules. This was done by looking at the actual folder structure of the source code. The results were summarized in an external Microsoft Excel file with the name of the LLC and the MLC to which it belongs. The tool was designed in Java and uses an external Excel library called JXL to create and read Excel files. The main purpose of this tool was to implement the architectural rules and to automatically extract the necessary dependency information from the build file. The implementation time for the tool was approximately one week for one developer with basic knowledge in Java. It uses input in the form of Excel files containing the architectural mapping and special attributes belonging to a rule (rule number 3 and 4 in Section 3.5.3.2). For example, a special attribute could be a layer in the architecture to which an LLC belongs to. The source code is used to extract the dependency files in order to retrieve all the dependencies in the system. Each rule was implemented as an algorithm in the tool for which the dependencies could be checked against. The output of the tool is an Excel file containing information about the source and destination component of each dependency as well as the type of rule that the dependency breaks. To ensure the correctness of the tool, testing was carried out during the development. A test system was created where all dependencies and violations were known. This was used to check if the tool found the correct results. Manual verification was also done several times to check different steps of the application such as reading Excel files and dependency files. Random samples were taken from the results to manually confirm the violations by double-checking the dependency files from the source code.

3.5.3.4 Usage of the Risk Measurement System

The Risk Measurement System (RMS) described in Section 2.5 was used on the same revision of the source code as RuleValidator in order to find out the complexity of each file in the system. Because of a change in revision systems at the company the revision history could not be gained automatically by the RMS and needed to be manually extracted. The time period for the revisions was set to a release period to reflect all stages of development. By selecting that time period the risk would show up as general and stable for each file rather than vast and short variations which could appear with a smaller time span. This is confirmed by the tool creator to be a proper way to address this threat. The result from the RMS was a relative risk for each file showing an

indication of the likelihood that a file becomes hard to maintain or manage i.e. the risk of effort [1]. However, to gain the same abstraction level as the dependencies the result was aggregated for each file in a LLC. This value is a relative value between LLC:s dependent on all files rather than individual risk of files. According to the definition made by Antinyan et. al. [1] more risky code is more error-prone and difficult-to-maintain. Since error-proneness and maintenance time are additive properties, and since the LLC:s are composed by files it was found logical to regard the risk of a LLC as a sum of all files' risks in that component.

The data about violations were combined with the risk to create a priority ranking of the dependencies to determine which one was the most severe. In the case of one-way dependencies, the risk of the violation was considered to be the risk of the source LLC. Circular dependencies were not treated in the same way, the risk was an aggregation of both LLC:s involved to reflect a higher risk for that type of violation. Section 2.3 explains how risk of effort can be introduced in dependencies, this was the reasoning behind the prioritization. The highest risk number would be considered to be the most severe dependency and would be an indication to the stakeholders on where to start refactoring.

3.5.4 Step 4 - Evaluate measurement

This chapter presents the evaluation and validation of the measurement. The ISO standard states the necessity to evaluate the information products which was done according the plan. The evaluation was also extended with a validation of the indicators as it was needed for the research point of view, this is presented in a separate section.

3.5.4.1 Evaluation of measurement

In order to successfully evaluate the Measurement System (MS) several steps were taken. This was done by continuously presenting the progress and process of the MS throughout 4 discussions and the 4 semi-constructed interviews with the architects. 12 weekly meetings during a 3 month period were also held with one Technology Expert and two scrum masters. This was done to in order to confirm that the MS provided them with useful information and to discuss improvements to the process. The result of this was that the evaluation criteria for approval were fulfilled as the indicators were correct, answered what they were looking for and increased their knowledge about the non-allowed dependencies.

3.5.4.2 Validation of measurement

Since the MS is composed of two indicators, both were subject to validation. However, the indicator of the actual violations was considered by the stakeholders to capture all non-allowed dependencies which was discussed continuously through all evaluation sessions described in the previous section. This was due to the fact that in order to create a dependency between two LLC:s, the path to the other LLC needs to be included in the

build file. Only two possibilities existed that would make the indicator of the violations to provide incorrect information. Firstly, it could be possible that the path to the other file in the dependency would be inserted directly in the source code. This is in direct violation of the coding standards within the company and would be identified during their continuous code review sessions. Secondly, it could be possible that a dependency stated in the build file was not actually manifested in the source code, i.e. the dependency was unused. This would require detailed code investigation to see how the actual dependency was manifested. However, this was not seen as a major problem since if the dependency was not manifested in the code, it would only be removed from the build file without any effort when refactoring. The stakeholders were given the chance to exclude such dependencies at the one-time validation interview which focused on validating the prioritization indicator. Additionally, to validate if the indicator of the violations would reveal any unknown non-allowed dependencies, a form was sent out to the stakeholders which is explained in Section 3.5.4.3.

To validate the second indicator, regarding the prioritization of the violations, the thesis had the aim to comply with the method of evaluating measurements proposed by Staron et. al. [33]. The method states an appropriate process for empirical validation of the MS at software organizations and is presented below:

1. *"Develop and deploy a measurement system.*
2. *For a period of time, validate the indicators and measures with the stakeholder: this validation is done by observing the values of the indicators together with the stakeholder and ask the stakeholder to assess the same phenomena without using the measurement system (stakeholder's view).*
3. *After a period of time, if the stakeholder's view is consistent with the indicators values, then we can assume that the measures are empirically valid."*

By conducting this final interview along with a discussion, an empirical validation of the prioritization indicator would make sure that the developed MS was measuring what it was meant to measure. It would in other words validate if the information that the indicator provide would match the stakeholders' information need. The interview was performed in the following way:

1. The interview subjects, i.e. the architects, were presented with the theory behind the concept of risk of adding extra effort to non-allowed dependencies. It was also explained that the risk is calculated based on the number of revisions and the complexity of a component. This was done in order to provide the architects with an understanding about the concept.
2. The architects were given samples of violations for each rule. The samples were taken from the output of the MS and included 3 top prioritized, 3 medium prioritized and 3 randomly selected in between for each rule. The MS' prioritization of each violation was not displayed for the subjects.

3. The architects were asked to prioritize the violations according to how much extra effort that they believed them to generate. They were not asked to prioritize by estimating the exact risk for each component by looking at complexity and number of revisions as this would be infeasible. Instead their decision should be based on their tacit knowledge of how the components in the non-allowed dependencies were considered to generate extra effort.

4. The prioritization from the MS was revealed followed by a discussion about potential differences between the two prioritizations and the reasoning behind the architects' decisions. The purpose was also to determine if the indicator provided them with correct and realistic information even though potential differences would occur.

However, the discussion held in step four revealed that the architects did not prioritize based on the concept of adding extra effort. Instead they prioritized according to if the violation was expected or unexpected for them. For example, a dependency often got a low priority when it was known beforehand, while it received a high priority when it was unknown or they couldn't see any logical explanation for that specific dependency. Sometimes they prioritized based on their *"gut feeling"* or by simply guessing. For some specific dependencies they included their experience about specific components behavior and interactions to determine the severity in order to prioritize. Some of the dependencies were questioned by the architects if they existed or not but they were not so certain about it that they could exclude them. In order to exclude these dependencies, detailed code investigation would have to be done. They also stated that an unknown dependency could be seen as more severe than a known one since it is not considered in development planning and could therefore generate unexpected side effects. Furthermore, they stated that it was hard connect the risk to a LLC during their prioritization since the concept was not well known for them which caused them to rely on other factors for estimating the prioritization. Also, it was hard to manually prioritize a dependency according to the amount of risk of effort since they did not have knowledge of how the actual dependency looked like, only that it existed. Therefore, the process of validating the prioritization indicator through the steps described by Staron et. al. [33] could not be followed completely. This was due to the fact that the architects' criteria for prioritizing manually were not the same that the MS used.

Additionally, the results from the architects' prioritization were used as a comparison with the results from the MS to validate if the prioritization indicator would add any extra knowledge to them. If the comparison would reveal a big difference in the prioritizations, but the architects still would confirm that the indicator provides useful information for them, it would prove that they were provided with new and helpful knowledge about the severity of the existing ATD:s in the system. On the contrary, if the comparison would mostly match, it would reveal that the information about the severity was already known beforehand.

3.5.4.3 Estimation form for architects

The final interview was followed up by sending a form containing all LLC:s and the formal definition of each rule. They were asked to fill in the rule violations that they believed to be in the system for each LLC they had knowledge about. The purpose of this step was to collect information on the awareness of violations in order to compare it against the results of the thesis. This could prove an increment in knowledge and/or a confirmation of correctly identified non-allowed dependencies.

3.6 Technical Solution of the Measurement System - an example

This section will describe how the MS was implemented technically at the case company. The purpose is to provide this information to those who desire to replicate the same method again. The description will be explained in the form of an example. The example will explain how the MS works, all the way from identifying the violations to prioritizing them. Moreover, this specific example will be focused on 3 Low Level Components (LLC) that belongs to the same Medium Level Component (MLC). These three LLC:s has two circular dependencies that are violations that will be identified and then prioritized which can be seen in Figure 3.3.

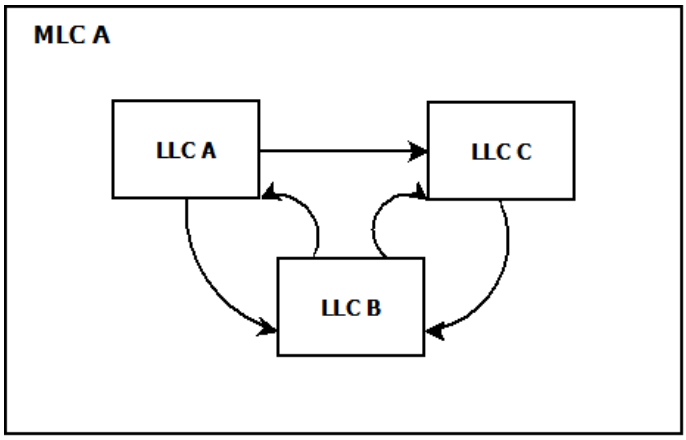

Figure 3.3: An example of dependencies between three different LLC:s in the same MLC.

First of all, the example starts with the build files of the LLC:s. This is where all the dependencies are found. The build files for these three LLC:s are the following:

LLC A

inc /product_name/mlcA/llcB

inc /product_name/mlcA/llcC

LLC B

inc /product_name/mlcA/llcA

inc /product_name/mlcA/llcC

LLC C

inc /product_name/mlcA/llcB

As seen, all dependencies correspond to the dependencies shown in Figure 3.3. The next step is to generate the architectural mapping for the specific system which is needed in order to know which LLC that belongs to which MLC. This is done in an Excel-file and is used as input for the tool that identifies the violations, RuleValidator. The Excel-file is composed of a table with two rows and includes all LLC:s and MLC:s that exists in the system. Table 3.1 shows how it would look for this example.

Table 3.1: an example of the Excel-file that holds the architectural mapping of the system.

	A	B
1	llcA	mlcA
2	llcB	mlcB
3	llcC	mlcC

Now, the input that is needed for RuleValidator has been established. As explained in Section 3.5.3.3, RuleValidator reads all the LLC:s build files from a folder which is located in the top folder of the source code for the system to measure all dependencies. In this case the path to that folder would be *"../product_name/"*. The dependencies are stored in a HashMap, called *"dependencies"* in this example, with a String as a key and an ArrayList of Strings as a value. The key represents the name of the specific LLC and the ArrayList contains all the names of the LLC:s that the specific LLC has dependencies to. Furthermore, the architectural mapping is read via an external library called JXL and stored as a HashMap with String as both key and value. The key represents a LLC and the value represents the specific MLC that it belongs to. For each rule, the tool uses an algorithm with the HashMap *"dependencies"* as input. These algorithms are stored in the class Rules.java. The output for each algorithm is an ArrayList of Pairs which is a tuple and contains two Strings. Each violation is modelled as an element in this ArrayList, with the type Pair<String,String> where the first String represents the source of the dependency and the second String represent the destination. For circular dependencies, both Strings are seen as source and destination. In this example, the algorithm for checking direct circular dependencies is explained by the following java/pseudo code:

```
function checkCircularDependencies(HashMap<String,ArrayList<String>
    dependencies) {
        ArrayList<Pair<String,String>> result

        for each dep in dependencies {

            ArrayList<String> destinations = dep.get()

                for each d in destinations {

                        ArrayList<String> sources = dependencies.get(d)

                    for each s in sources {
                        if ( s equals d)
                            result.add(new Pair(s,d))
                    }
                }
        }
    result = removeDuplicates(result)
    return result
}
```

To explain in text, what the algorithm does is that it retrieves all the destinations of the dependencies that come from a specific LLC. For each of those LLC:s it is checked if it has any destinations to the first LLC. If that is the case, a direct circular dependency exists and it is stored in the results ArrayList. Also, since the algorithm will find duplicates for each circular dependency, a function removeDuplicates removes them. As a side note, RuleValidator is not implemented with the most optimal data structure, a better way would be to model the components as objects with the dependencies as attributes to these objects. Additionally, the algorithms for checking the rules are not the most optimal either, but they suit the purpose of the specific case company.

For this example, the output of the checkCircularDependencies algorithm would be an ArrayList with the two following pairs:

- <llcB,llcC>
- <llcA,llcB>

After RuleValidator has executed all rule checking algorithms, the result is four ArrayLists of the abovementioned kind. One for each rule. The next step is to present the output in an Excel-file that holds all the information about the violations. The Excel-file holds four different sheets, one for each rule, and presents all non-allowed dependencies that have been identified. It also presents empty rows that will be used later for the prioritization process. Table 3.2 shows the sheet of circular dependencies for the example.

As seen, the priority and risk fields are empty at the moment. The following step would be to start the prioritization process. This is done with the Risk Measurement

Table 3.2: Example of the sheet Circular Dependencies output from the RuleValidator.

Priority	LLC 1	LLC 2	Risk
	llcB	llcC	
	llcA	llcB	

System (RMS). The risk calculated is based on the complexity of the source code and the number of changes done to the code. The RMS needs access to the source code and a file containing all the changes of the code. The file that is containing all the changes is retrieved via the RCS for the source code and has the following structure:

/product_name/mlcA/llcA 2014-01-05

/product_name/mlcA/llcB 2014-03-02

/product_name/mlcA/llcA 2014-03-04

One line corresponds to one change made in the source code and they stretch back 6 months in time. The RMS is implemented as an Excel-macro and uses the revision file along with the source code in order to calculate the relative risk for each file and displays it in an Excel sheet. The MS implemented by this thesis then manually aggregates the risk of each source code file to the LLC that it belongs to. For example, if LLC A consists of two files with the risks 12.3 and 5.4, then the risk of LLC A would be 17.7. This is done for all the LLC:s within the system and put as a separate sheet in the Excel file which holds all the violations. Table 3.3 shows the result for this example.

Table 3.3: The relative risk of each LLC within the system for this example

LLC	Relative Risk
llcA	17,7
llcB	19,1
llcC	2,3

Now, the risk has been mapped to the right abstraction level (LLC:s) and will now be mapped to the non-allowed dependencies in order to create a prioritization. For this example, regarding circular dependencies, the risk for the violation is calculated by aggregating the risk of both LLC:s. Since this information should be displayed along with the information from Table 3.2, Excel formulas are introduced in the risk column for automatic calculation of the violations. The violations are then sorted based on the risk and are given the right prioritization. Table 3.4 shows the end result of the MS.

Table 3.4: The two circular dependencies has been prioritized.

Priority	LLC 1	LLC 2	Risk
1	llcA	llcB	36,8
2	llcB	llcC	21,4

3.7 Validity threats

As with all empirical studies there always exists threats to the validity of the study. These should be handled and taken into consideration to increase to trustworthiness [28]. In this section the threats to this thesis are presented as four different categories along with measures taken to mitigate the impact of them.

3.7.1 Construct validity

This concerns that the results are representing what the study is actually trying to investigate and which are needed to answer the research questions. For this study a threat lies in the architectural rules not being understood in the same way by the thesis writers and the architects. This was addressed by presenting the same definition about architectural technical debt and the meaning of a dependency at the start of each interview. Moreover, rules could also be formulated in an ambiguous way or a misinterpretation of the result could lead to faulty rules. Therefore, during the common interview, the rules were agreed on by all architects and a written definition of each one was created together. As revealing Architecture Technical Debt (ATD) could be seen as showing defects or mistakes in architecture a threat lied in the architects intentionally formulating rules to exclude some violations. Except from having a shared common responsibility for the whole system each architect is responsible for one or more MLC:s and they could therefore have a tendency to protect their own territory. As the interview subjects were quite few the threat was mitigated to the best possible extent by holding interviews separately with all architects and to let them talk freely about other areas. The common meeting held with all of them revealed the results for each of the interviews. This proved to work well, for example one rule was formulated even stricter than during the individual interviews. The reasoning for this was stated to be able to get the most violations as possible, this proves the commitment of the architects and eliminates the threat of withholding information. However, this does not eliminate all threats of interview bias as it would be possible for the architects wanting to *"gold plate"* the architecture solution rather than achieving a good enough one. By showing worse results for management more resources could be spent on refactoring then necessary. Due to different level of architectural expertise within the case company this threat could only be handled by clearly stating the rules and presenting the measurement details in a way understandable for everyone involved in the system. The result from all interviews have been validated with all subjects to eliminate the threat of misinterpreting any information, quotes or conclusions drawn from it.

3.7.2 Internal validity

This concerns the risk that an investigated factor is affected by something not considered in the study. During the work of the thesis a couple of factors that could affect the results were identified. One factor was considered to be the ability of introducing dependencies directly in the source code without using the dependency file. This is however stated by the architects to be very unusual and would most likely be found during code reviews. Therefore, the risk of missing dependencies was considered to be very low and the threat was mitigated. Another factor that could influence the result was unused dependencies. As the tool only checks in a specific file and not in the source code there could be cases where a dependency between components is just an include statement without using any parts of the source code. The risk of the dependency could therefore be misleading as the LLC:s risk does not include any extra effort from the violation. However, through discussion with architects, a decision was made to show these fake or unused dependency as they opened up for a rule violation and should be removed. During the final validation it was clearly stated that the results presented could include false dependencies. This was done to give the architects a possibility to question the results.

3.7.3 External validity

This concerns how generalizable the results from the study are. It is fair to assume that most software in the world includes non-allowed dependencies that could be the source of ATD. Therefore, a successful method that is useful for practitioners would provide great value to the community. The generalizability of the thesis method lies in how well each step can be carried out at other cases. The following points describes parts of the Measurement System (MS) that needs to be considered in order for it to be applicable in other contexts:

- *Rules or thoughts about the dependencies in the system needs to be defined:* If there is a lack of architectural thought, it could be tackled by investigating the existing product and determine rules to be followed in the future.
- *The dependencies between the components of the software needs to be extracted:* At the case company, the build files are used to extract the dependencies between components. These are required for the MS to be efficiently implemented. If a build file or any other similar dependency file is not available, a tool that examines source code needs to be used instead. This could require a significant amount of effort and understanding which would make the MS more time consuming to implement.
- *The extraction of the dependencies needs to be at the same abstraction level as the architectural rules:* At the case company, the rules were concerning L- and MLC:s. As the build file reflects the dependencies at the same abstraction level this made the process of verification feasible. This might not always be the case, and then more effort would have to be put in to map the rules with the dependencies available in the source code.

- *A Revision Conrol System needs to be used:* In order to prioritize non-allowed dependencies according to the risk of being hard-to-maintain, historical revision of the source code is needed. If this is not the case, the MS will only be able to identify and not prioritize the ATD:s.

This threat was mitigated by trying to use components that are likely to exist in most software developing industries and which are not to complex or time consuming to use and understand. For example, to the best understanding of the thesis workers, most companies have some architect(s) and a planned architecture. At least there exist some rules or thoughts about good practices which can be used in that case. However to extraction of dependencies is the major threat for repeating the method with the same effectiveness. Without the formal build file a much more complex tool could be needed which would increase the amount of effort for the process.

3.7.4 Reliability

This concerns how reliable the results from the study are. It addresses the question if other researchers would produce the same output by conducting the same study. The major threats are associated with the extraction of the rules as they are based on qualitative interviews. The interviews held with the architects increased the knowledge about architectural rules, if the process was repeated the results could be more precise than before. The process for creating a measurement followed a defined ISO standard to the greatest extent eliminating the threat of having an unclear process during the study. By following the standard it would be easier to understand the method and to use it in future or similar research. To increase the reliability a technical example of how to implement all the necessary steps to check one rule are included in this report. This provide a clearer picture for how the study was done at the case company which would help out if someone wanted to repeat it in the same context, or a simliar one.

4

Results

This chapter presents the contribution of the thesis in form of the two evaluated methods for identifying Architectural Technical Debt (ATD) and a new approach for prioritizing violations. Both methods include steps and components that need to be taken into consideration to successfully carry out each process. The new approach connects the risk of adding extra effort with the non allowed dependencies. This chapter also includes the results from the process of validating the idea of prioritization together with the architects and a comparison between violations identified by the previously mentioned and the outcome of the measurement system.

4.1 Method 1 - Comparing feature impact estimations against changes

As explained in section 3.4, one way of finding ATD in the form of non-allowed dependencies would be to compare the estimations of the impact, i.e. the parts that will be affected by change because of the feature, with the actual outcome of the development. A result of this thesis is a lightweight method for identifying this type of ATD:s. The scope of this method is to reveal specific ATD:s that are related to certain features. When the method is applied in the appropriate setting, it could reveal debts without any use of external tools or a heavy setup process. Firstly, the prerequisites needed in order to start the method will be explained. After the prerequisites, the method is divided into certain steps that explain what needs to be done in order to identify the ATD:s. Lastly, Figure 4.1 visualizes the method.

4.1.1 Prerequisites

In order to follow the process, certain prerequisites need to be taken into consideration. The prerequisites explained below are crucial when assessing if a feature is suitable or

not for being a subject for this method.

- *The estimations of the impact have to be documented.* In order to compare the outcome with the estimations, the estimations have to be written down. For many features, documentations were found but it lacked the estimations.
- *The documented impact estimations should reflect intended estimations rather than implementation results.* Some features had documented impact estimations but they were updated along with the progression of the development. As a result the estimations instead described what was actually implemented.
- *The estimations of the impact cannot be at too high abstraction level.* The estimations should preferably be low level components since it provides with a more detailed dependencies. By using a higher level of components the dependencies are not traceable to low level components which could leave out ATD.
- *Knowledge about the estimations of the impact has to be available.* In order to assess the validity of the estimations, interviews with the responsible designers has to be conducted. The reason for this is to determine if the estimations fulfills the criteria described in this section. After investigating, several features proved to be too old to use because the designers had forgotten the details of the estimation process.
- *The outcome of the development needs to be available for measurement.* Since this method relies on revision history for a specific feature it needs to be possible to identify all the development done for that feature and to separate it from others. This is done by looking at the branch(es) used during implementation as they contain all the changes done related to a certain feature. For some features it were not possible to trace back which branch(es) were used for development.
- *The estimations have to be done with the intended architecture or architectural rules in mind.* The estimations need to be based on architectural thought in order to use them for the method. If they are derived without knowing about the architecture the estimations could include non-allowed dependencies which would not be detected by this method.

4.1.2 Steps to conduct the method

Once the prerequisites are met the process of comparing the estimations with resulting changes can be started. The following steps define the necessary actions that need to be taken in each phase of the method:

1. *Select suitable feature and identify impact estimations* - The process begins with the selection of a feature that will be examined and the review of that features impact estimations. The documentation of the initial impact estimations needs to be available. An example of impact estimations is shown in Figure 4.1 where 3 LLC:s are estimated to change

2. *Identify implementation results* - In order to compare the estimations, the impact results of the implementation needs to be identified as well. This is done through accessing the revision history of the feature from the RCS where the source code for the feature is present. An example of implementation results is shown in Figure 4.1 where the revision history reveals changes in 4 LLC:s.

3. *Compare differences* - To detect any differences between the results and the estimations, the next step is to compare them. This is done simply by subtracting the results from the estimations. The difference is used as input for the next step. As seen in Figure 4.1 the estimations difference would be the extra LLC (LLC4) as it is not in the estimation impacts

4. *Identify non-allowed dependencies* - When the difference from the comparison is available, assessment has to be done whether the origin is out of any non-allowed dependencies. This is done through interviews with responsible designers for that specific feature. The goal with the interview is to trace the difference to non-allowed dependencies. In Figure 4.1 the non-allowed dependency shows up to LLC4 as it changed during development but was not included in the impact estimations.

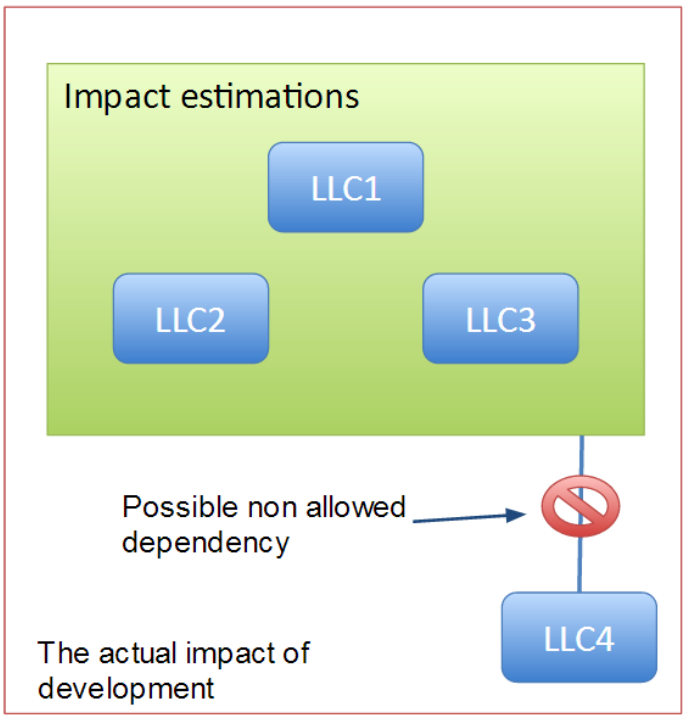

Figure 4.1: Model of method 1

4.1.3 Status of the method

When carrying out the method it was found out that only 3 features out of 33 of the investigated features fulfilled all the prerequisites, therefore it was decided during evaluation to not go ahead with this method. The reason was that it would not be enough data points to draw any valid conclusions. It would not be possible to provide evidence that the method would be useful in identifying ATD. However, the thesis has successfully extracted the necessary steps and prerequisites and if they are true for a certain industrial environment the method could successfully be evaluated. The method has also been subject for static validation by several personnel involved in the case company. Feedback has been provided that shows that this method could be very useful if the prerequisites should be met. For example, one Technology Expert within the company claimed that *"The method would be useful if we would have the required information that is needed to identify our violating dependencies"*. It was also confirmed that if the company would meet these prerequisites, this process could be tested in order to detect debts in their system. Technology Experts and scrum masters stated the need of introducing checklists throughout the development phases that would ensure that the information needed for the prerequisites would be documented.

4.2 Method 2 - Measuring and prioritizing non-allowed dependencies

This section will describe the result of the Measurement System (MS) developed from the ISO standard 15939:2007 [9]. The scope of this MS is to identify and prioritize all the non-allowed dependencies of the system that is being measured. The result is divided up in two subsections. This section is based on the process of identifying and the process of prioritizing the non-allowed dependencies. Each subsection states the necessary steps and components along with critical prerequisites that are needed to start the process. As for method 1 explained in Section 4.1 these subsections have the same structure regarding of how the processes are presented. The structure starts with the prerequisites for the method followed by the necessary steps and ended with a visualization of the components in the process.

4.2.1 Components and process of identifying violations

This thesis aimed to identify ATD in the form of non-allowed dependencies. In order to answer the first research question a process and all the necessary components were created and identified. This contributes to research as it presents everything that is needed to find violations along with a feasible method for the case context. The process and the components for finding violations are presented in this section and seen in Figure 4.2.

4.2.1.1 Prerequisites

The following prerequisites are required in order to start performing the process:

- *Access to source code.* This access is crucial since the dependencies of the system are manifested in the source code.
- *Access to architectural expert knowledge.* In order to create rules about the architecture, access must be obtained to the knowledge of how the architecture is intended to look like. This can be obtained from interviewing architectural experts or by viewing documentation about the architectural guidelines within the system.

4.2.1.2 Steps to conduct the method

As soon as the prerequisites have been confirmed to be ok for the system the identification process can begin. The following steps explain the process:

1. *Architectural rules* - The process of finding violations starts by identifying the rules for the architecture, this is done along with architectural experts. However, the rules, if stated, could be derived from documentation about architecture. For the thesis little documentation existed and only worked as a starting point, rules were mainly formulated together with the architects during interviews. Rules should aim to reflect the intended design of the architecture and therefore work as identifiers of non-allowed dependencies. The architectural experts should be the ones responsible for the architectural design and the future changes to it.
2. *Architectural mapping* - Source code files needs to be connected to architectural components. These components should match the same abstraction level as the rules. The mapping for the thesis was derived from the file structure of the source code but could also, if available, be gained from other software artifacts such as documentation.
3. *Deriving dependencies* - The existing dependencies within the software system should be derived from the source code. For the thesis this was done with the self-developed tool but could be done with similar solutions. The dependencies should be on the same level as the architectural mapping to be able to determine violations to the rules.
4. *Find violations within dependencies* - By checking dependencies against the rules derived in the first step violations were identified. This was done automatically by the self-developed tool, where each rule was implemented as an algorithm and checked against each dependency. The result which is the actual indicator, is a tuple ,<x1,x2>, with the source (x1) and destination component (x2) involved in the non-allowed dependency.

   There exist no strict order of the steps in the process and most of them can be done in parallel. Only the last step is dependent on its predecessors. The first

and the second step is connected since the outcome in form of the architectural abstraction level needs to be the same. This should be taken into consideration when deciding on which step to do first or when working in parallel with them.

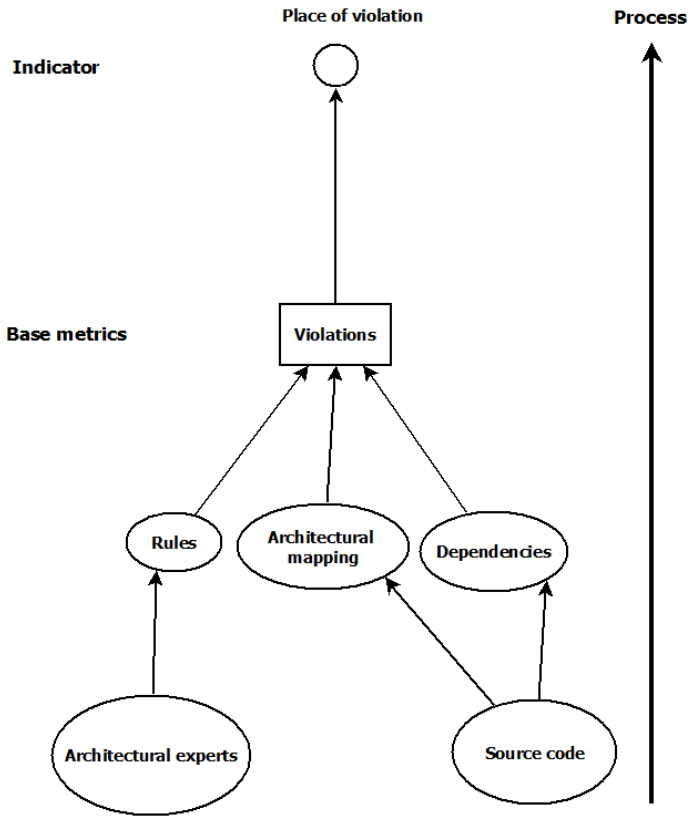

Figure 4.2: Components for the violation indicator.

4.2.2 Components and process of prioritizing

Along with identifying violations this thesis contributes with an additional process and all necessary components to prioritize violations. This is used to answers the second research question in the thesis. The steps in the process along with the components are presented in this section and seen in Figure 4.3

4.2.2.1 Prerequisites

To start the process of prioritizing the violations, two important components need to be available. They are the following:

- *Violations* - In order to start the prioritization process, the process of identifying all violations needs to be finished. The end result from that process is a tuple

of LLC:s <x1,x2> including the source (x1) and the destination (x2) of the non-allowed dependency that is used for prioritizing in this process.

- *Architectural mapping* - When conducting the process of identifying violations, the component of architectural mapping is established. The same component needs to be a part of this process as well. This is important since if a new architectural mapping is derived, it might not be at the same abstraction level, and then it will not be possible to map the risk of adding extra effort to a specific non-allowed dependency.

4.2.2.2 Steps to conduct the method

When the pre required components from the process of identifying violations are in place the following steps explains how the process is performed:

1. *Calculate complexity and number of revisions* - From the source code McCabe's complexity, which is explained in Section 2.5, should be calculated along with the Number of Revisions (NR). The source code needs to be in a Revision Control System (RCS) to get the NR. The time period for the NR should reflect the release time for the product that is being measured. For the thesis the NR was derived manually and the complexity was calculated by the RMS.

2. *Calculate risk* - The risk for each source code file is calculated by the RMS which is more rigorously explained in Section 2.5.

3. *Map risk to the right architectural abstraction level* - The risk for each software source code file should be aggregated for files belonging to the same component. This is done with the architectural mapping so it is possible to connect the risk with a violation.

4. *Map risk to violation* - The component risk should be mapped to the violation depending on the type, one- or two-way. The outcome will be a prioritized list for each rule with the most severe at the top and the least severe at the bottom.

All the steps for this process needs to be done in sequential order and steps 1-2 could be done without the results from the violation process, however step 3 needs the architectural mapping but could be done before having the actual violations.

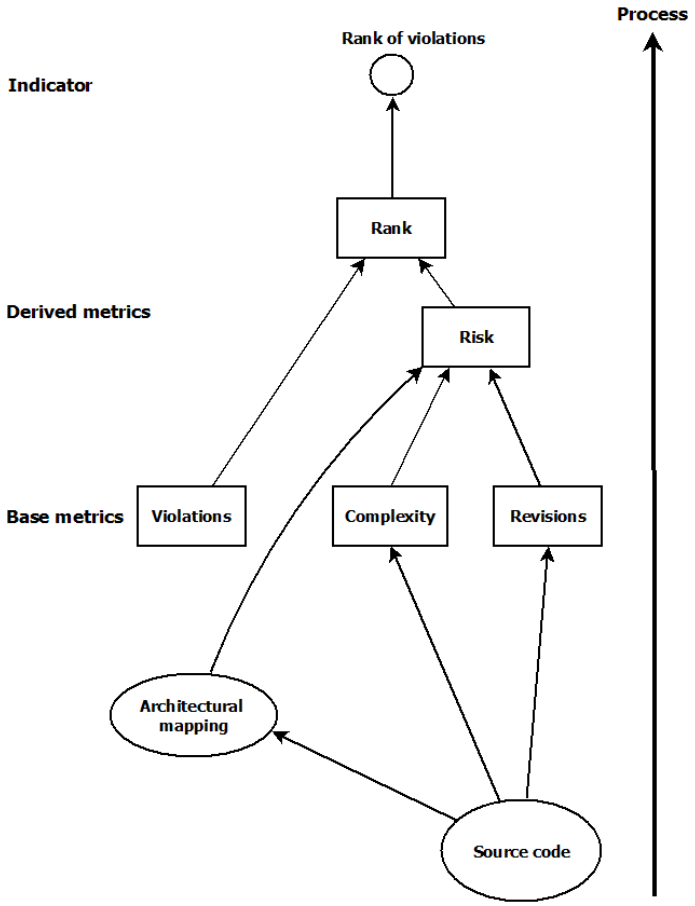

Figure 4.3: Components for prioritization of violations

4.3 New approach of prioritizing

This thesis presents a way to prioritize dependencies by using the research from another study focusing on identifying risky source code files [1]. By establishing an architectural map and connecting the software elements to a higher abstraction level, namely components or the case specific Software Unit (LLC), the risk can be transformed and used to prioritize violations in the architecture. The study establishes a way to connect the risk for each component to an actual non-allowed dependency by defining where the extra effort of a dependency actual lies (explained in Section 2.3). By connecting the risk to a violation it is possible to compare a dependency with others to determine which one is the most severe. This aims to address the problem of not having enough material to convince management on the severity of non-allowed dependencies and to work as an indicator on what to start focus on for refactoring work.

When presenting the prioritized violations for the architects they confirmed it to be valid and stated that it would be of much value for them to see the associated risk. When

comparing their estimations with the ones produced by the thesis workers one could see some mismatches. However, as the latter estimation was accepted this could be seen as adding knowledge not already in the company. During the interview it was found out that the results would be of value when presenting for technical experts which proves that the aim of providing more material for managerial decisions about refactoring was achieved.

4.4 Results from the developed tool for identifying non-allow dependencies

Table 4.1 describes the number of violations that were identified for each rule using the developed tool. The result is separated per rule due to the fact that they are different violations and the weight, i.e. importance, has not been evaluated for each rule. For example, a violation that breaks rule 2 might be more severe than a violation that breaks rule 1 or vice versa.

Table 4.1: Number of violations found per rule. Rule 4 identified one false violation due to a bug in the developed tool, RuleValidator.

Rule	Nr of violations
1	91
2	19
3	7
4	5 (1 false)

The existing violations that the architects are aware of in the system today are considered to be a problem. Both known and unknown dependencies involve the risk of extra effort but the latter ones are considered as more severe as the estimations won't take the extra time into consideration. It also makes testing harder since, due to the extra dependency, more components need to be built for a test suite, which leads to even more loss of time.

4.5 Evaluation of the Measurement System

When faced with a set of ATD:s in the architecture in the form of non-allowed dependencies it is preferable to have them prioritized so it can be decided which one should be refactored first. This would ease the work when developing and triggering refactoring activities. The prioritization for this MS was based on the risk of adding extra effort the specific non-allowed dependency could produce compared to the other ones. The result was different lists of prioritizations, one list for each rule. The top one would correspond to the most severe dependency and the bottom one to the least severe. Table 4.2 shows an example of how the prioritization could look like for a rule.

Table 4.2: An example of how a prioritized list of violations for a specific rule with a one way violation.

Prioritization	Source LLC	Destination LLC	Relative risk value
1	LLC A	LLC B	10,34
2	LLC A	LLC C	10,34
3	LLC C	LLC B	1,20
4	LLC D	LLC E	0

In order to prove that the prioritization was providing the architects with useful and new knowledge several steps were taken. By comparing the results from the MS with manual estimations by the architects, conclusions could be drawn about the increment of knowledge. The results from these comparisons are presented below in four different tables. Each table shows the prioritization of dependencies according to the MS along with the prioritizations made by the architects and the actual risk value. The same number in the MS prioritization column means that the violations share the same risk. In each table, the last row explains the number of correct matches (represented by a zero) and the average of the difference in the prioritizations. The average of difference shows the difference for the prioritization that a violation got between the architects and the MS estimations. The closer to zero the average is, the less difference between the two prioritizations.

Table 4.3 shows the result from the rule *"A Low Level Component (LLC) should not have a dependency to a LLC in another MLC. However, dependencies to LLC:s in the Platform MLC is OK"*. The nine sample points consists of the 3 top most risky, the 3 least risky and 3 randomly selected in between. For this rule, two estimations were exactly according to the measurement system. The dependency which was identified as most risky by the measurement system got a low priority by the architects, conversely the one of the least risky got the highest priority. The results in between are however closer to the MS prioritization.

Table 4.4 shows the result from the rule *"Two LLC:s should not have circular dependencies between each other"*. The nine samples points were selected in the same way as for the previous rule. Again, two estimations are according to the MS and the most risky one according to the MS is the least risky one according to the architects. The top violation is in the lower bottom half which again shows a mismatch. By comparing the difference from the previous rules the results are quite similar just slightly less accurate for this rule.

Table 4.5 shows the result from the rule *"LLC:s in Control Layer should not have dependencies between each other"*. The results show all the violations for the rule. One result is exactly according to the estimations and the top one from MS gets a low priority by the architects. Otherwise the results are not deviating as much as for the previous rules.

Table 4.6 shows the result from the rule *"A LLC in the platform should not have a dependency to a LLC outside the platform"*. The results show all the violations for the rule. During the interview one violation showed to be faulty as it was a dependency between two platform dependencies. However, only the top one involved a greater amount of risk and the rest shared the same position, since they all originated from the same LLC, which makes the results a bit harder to interpret.

Table 4.3: Results from rule 1: Dependencies between MLC:s

Risk	**MS prioritization**	**Architects prioritization**	**Difference**
36,64	1	6	5
15,54	2	4	2
15,54	2	2	0
10,62	4	3	1
4,09	5	9	4
1,49	6	8	2
0	7	7	0
0	7	1	6
0	7	5	2
		Number of matches: 2	**Average of difference: 2,4**

Table 4.4: Results from rule 2: circular dependencies

Risk	**MS prioritization**	**Architects prioritization**	**Difference**
51,89	1	9	8
48,26	2	2	0
36,64	3	4	1
31,91	4	8	4
24,05	5	3	2
10,88	6	1	5
0	7	7	0
0	7	6	1
0	7	5	2
		Number of matches: 2	**Average of difference: 2,6**

Table 4.5: Results from rule 3: LLC:s in Control Layer

Risk	MS prioritization	Architects prioritization	Difference
15,54	1	5	4
11,63	2	2	0
6,40	3	1	2
3,59	4	2	2
3,59	4	5	1
3,22	6	2	4
3,22	6	5	1
		Number of matches: 1	**Average of difference: 2**

Table 4.6: Results from rule 4: platform dependencies

Risk	MS prioritization	Architects prioritization	Difference
3,55	1	3	2
3,55	1	—	faulty
2,37	3	1	2
2,37	3	2	1
2,37	3	3	0
2,37	3	5	2
		Number of matches: 1	**Average of difference: 2**

4.6 Results from validation interview of the Measurement System

After presenting the results from the MS, discussions were held with the goal to find out if the MS was useful for the architects. For the first rule it was recognized that the high priority dependencies from the MS were LLC:s that were hard-to-maintain. One architect said that *"SwU [LLC] A that has gotten high priority in rule 1 is very "messy" to work with"* confirming that the dependency actually generated extra effort for them. For one specific case, a highly prioritized dependency already had scheduled refactoring activities planned. They explained their low prioritization on a high risk LLC to be that it was a known dependency but also to be because of the source component of the dependency containing functionality that actually should exist somewhere else.

For rule number two, regarding circular dependencies, the top prioritized violations were recognized as tightly coupled and that the included LLC:s often changed together. Some dependencies were already known to be the causes of a *"not natural"* separation

where the components in essence could be the same. This violation was not seen as so severe and would therefore achieve a lower prioritization by the architects' estimations. Some violations were also considered to be more ok than others because of the behavior of the components and risk would not be a good indicator in that case. One of the estimations also broke another rule which affected the prioritization for the current rule, the knowledge of the other rules could therefore affect the results. For rule number three they explained their low prioritization to be an expected one which would be easy to fix. The highly prioritized one was not previously known.

4.6.1 The logic behind one-way and circular dependencies

For violations that included one way dependencies, the architects thought that the logic behind the risk in the source LLC was a good factor for deciding the amount of extra effort that the specific violation added. For circular dependencies, they also agreed that the risk of both LLC:s also needs to be considered due to the fact that changes in any of the components could trigger an extra change in the other one.

4.6.2 Conclusions of the validation interview

To sum up, the architects are aware that there exist non-allowed dependencies in the system and they confirm that it generates extra effort for them both in maintenance and testing. However, the results still shows an increment in knowledge since some violations were not known beforehand. This is proved since the architects confirmed that they were not aware of all non-allowed dependencies that the MS identified. The prioritization made by the architects compared to the one made by the MS showed two different outcomes. The main reasons were that they prioritized by different criteria than the MS and that they had troubles with manually assessing if a LLC was considered as risky or not. Nonetheless, they still agreed that the MS prioritization was a good indicator to show the most effort generating violations regarding to the amount of extra effort generated by them and a useful for knowing where to start refactoring work. They said that the inclusion of theory about the risk of adding extra effort along with the dependencies gave trustworthiness to the results. The general comments about the actual violations were that some were expected but some were completely new. Some of the known were the result of refactoring investigations already done at the company but also because of testing. Much of the extra effort that the non-allowed dependencies generate shows up when they need to include a large amount of LLC:s when testing small parts of the system, this is considered to be a major problem.

5

Discussion

This chapter will discuss the contributions of the thesis and how each of them answers to the research questions stated in Section 1.3. Firstly, the observed qualities of method 1 will be discussed. Secondly, the next two sections will discuss the two indicators of the Measurement System (MS) developed from the ISO standard. After that, the validations for identifying and prioritizing with the MS will be addressed. This is followed by sections that discusses automation and information quality along with a new approach for further research. Lastly, the related work of this thesis is presented.

5.1 Method 1 - Comparing feature impact estimations against changes

The case study conducted at Ericsson aimed to find ways that could identify Architectural Technical Debt (ATD) in the form of non-allowed dependencies. One approach of doing this was to compare initial impact estimations of features with the actual results of the development which resulted in this method. This would identify actual violations or ATD in order to answer the first research question. The extension of data from the Risk Measurement System (RMS) would add the ability to prioritize the identified violations which would address the second research question of the thesis. Furthermore, the results revealed that the method was not applicable in Ericsson's context. The problem was that critical prerequisites that was needed for the method to be successfully conducted was not met. These prerequisites are seen in Section 4.1.1. This section aims to discuss the importance of good estimations in order to identify ATD:s along with the potential benefit of that the method could be very easy to perform. Additionally, when the focus is to examine new features, the method automatically focuses on the debts which are causing problems now. This filters out debts that are not relevant at the moment. The section ends with a discussion about how this method can be combined with the priori-

tization process that this thesis proposes in order to introduce a more customizable way of managing ATD:s.

5.1.1 Rough estimations could hinder the method

An important property of finding non-allowed dependencies by this method is its reliance on estimations of which components are estimated to change for a certain feature. As explained in Section 3.3, the case company works according to agile practices. An agile approach, compared to other practices such as the traditional Waterfall approach, lacks a clear and rigid estimation process in the beginning of the project [34]. Instead agile estimations are often rough and fluent at the initial phases of the feature development in order to become more precise with the increased amount of knowledge that comes over time. Due to the fact that this method relies on comparison between estimations and results a negative effect could be showing more than just non allowed dependencies as not enough effort was put into estimations. For example, such differences could be extra allowed dependencies that just were missed in the estimations because too little effort was put into the research of the impact. This is an important point that needs to be considered and evaluated by further studies.

5.1.2 A lightweight method

Should the prerequisites prove to be valid for a specific case, this method could very well be an efficient way of finding non-allowed dependencies as it would be an easy and time efficient method. This is due to the fact that the method does not rely on any external tool, or an implementation of a tool which can be time-consuming to develop and maintain. In the long run, should the method be fully automatized, it would even be more time saving. An automatized method removes all the time that is put on manually comparing and assessing the potential violations for each feature. This would add to the current pool of research on methods for finding ATD. It is well known fact that different methods for addressing problems works differently well due to the context of where the method is applied. This could very well be the case for this situation and further research has to be made in order to validate if this method could work as an approach to identifying ATD, preferably at a case where the context fulfills the prerequisites.

5.1.3 Properties due to the scope of the method

Since the proposed method is examining features in order to detect ATD:s in the form of non-allowed dependencies it only detects the violations that are related to the investigated features. This means that it does not find every violating dependency in the architecture. If the goal is to find all violations in the system, the complete range of features would have to be examined. And even then, some non-allowed dependencies might still remain undetected since it is not always the case that a dependency is introduced by or related to a feature. Nevertheless, if a certain debt is a problem for the software it means that it is generating a greater amount of extra effort. That debt would most

likely be detected through examining several features and this means that this method could still be argued to identify effort generating ATD:s. Those violations that are left undetected would be less likely to be a problem for the current development since they are not a part of many features and therefore not causing any extra effort to the present progress of the software. If the focus is on examining features related to the current development, a type of prioritization of the debts that exists in the system is included. This prioritization leaves out debts that are not generating any extra effort in the present progress of the software and focuses on those which could be the cause of problems now.

5.1.4 Identifies the same type of ATD as method 2

A positive aspect is that the process for prioritizing violations, which is included in the MS, can be used for prioritizing in this method as well. This is due to the fact that this method identifies the same type of ATD items as the MS developed by the ISO standard [9], namely non-allowed dependencies. Should this method be proven to be more efficient and better suited for another context it would have the benefit of making the process of managing ATD:s more flexible. It would make the process of prioritizing not being reliant on the specific process for identifying violations, explained in Section 4.2.1, and a company in another context could tailor the process from identification to prioritization more after its own needs.

5.2 Method 2 - Measuring and prioritizing non-allowed dependencies

This section is divided into two subsections presenting discussions about each one of the two processes included in the MS; identification of violations and prioritization of violations.

5.2.1 Process for identifying violations

The following section discusses important considerations regarding the steps and components necessary to identify ATD which answers the first research question of this thesis. The discussion starts with the benefits of creating a context specific tool for identifying violations within the case company. Additionally, the discussion focuses on the fact that the components within the process have different strengths in generalizability. The section ends with an elaboration on how creating rules about the architecture can lead to a higher quality of the software product. The developed identification process contributes to the field as it adds another validated way on identifying ATD in the specific form of non-allowed dependencies.

5.2.1.1 Checking violations by implementing a context specific tool

The identification process itself is general but based on common components existing within a software company, this in turn makes it more applicable for a specific company as

components can be adapted to the specific context but the process is kept generalizable. However, the component of checking dependencies against rules probably won't exist within most companies. For this thesis an own tool was developed to get violations based on rules. The reasoning for this was that existing tools for checking dependencies did not include the specific rules. Adding those would take time in understanding the tool, the programing language and possibly in developing a needed extension. An own tool would, based on the knowledge and experience of the thesis workers, not take too much time to developed and could also be done in a familiar programing language. It is the belief that the creation of a such a simple tool is best done within the company in an known programing language as it would be less time-consuming than understanding and extending an existing solution.

5.2.1.2 Customizable components in the process

The components within the process have the advantage of being adjustable for the specific company which the process is applied to. This makes it more generalizable which both architectural mapping and rules are proofs of. For example, rules can be formulated to match the maturity of the company and can be both trivial and complex. Architectural mapping can be on a low level such as simple components or at a much higher abstraction level than used for this thesis. However, the result of this study is somewhat limited to the formal build file as it contains information about dependencies, a more informal system without this file would need a more complex tool for extracting dependencies. This influences the usability of this method which could make the process more difficult to conduct in another case context. Nevertheless, the usage of the formal build file comes with a major advantage as the results are not missing out on ATD:s in form of non-allowed dependencies.

5.2.1.3 Creates awareness of rules

The process of identifying violations contributed with an increased awareness of architectural rules which was a result of one of the components. When creating the rules the architects stated that their knowledge was increased as the rules were formulated and discussed. Since the documentation at the case company did not clearly state architectural rules at the investigated abstraction level the outcome could be a result for the case company. By documenting unspoken as well as spoken rules the awareness and common understanding for them increases but also eliminates the chance of them being misinterpreted due to the previous uncertainty. As an improvement more conscious decisions could be taken during future development to avoid non-allowed dependencies which would lead to a product of higher quality with less debt.

5.2.2 Process for prioritizing violations

To answer the second research question a process for connecting identified violations with risk in order to prioritize was developed. The following section presents a discus-

sion about how creating a unified way of understanding the severity and removing human estimations makes the process easily understood. It also introduces relatively little overhead to the stakeholders. Many researchers states problems associated with gathering the information needed for ATD items and the separation of severity among them [8] [17]. The process addresses crucial problems that needs to be taken into consideration when prioritizing:

- *What should be measured?*
- *Where should the measurements be conducted?*
- *How should the results be interpreted?*

Following the process gives the advantage of breaking down data into small components to easily visualize what is needed and in which order to create the indicator. It clearly defines each component and keeps them at a relatively non-complex level in order to produce a rank for each dependency in the prioritization. This, along with the approach of prioritizing violations discussed in Section 5.3, solves the previously mentioned questions.

5.2.2.1 Time efficient and easily understood

In order to decide which ATD to fix first Seaman et. al. [8] states some problems that needs to be addressed. First of all, in order for the process to be useful it needs to provide a value greater than the time spent constructing and running it. The method should not introduce too much overhead or be too complex which reduces the cost effectiveness. The presented process takes this into consideration since it is not associated with any time-consuming steps. The process does not have any need of human estimations which removes any time-consuming threat that this would add. However, a downside to leaving out human estimations is that they might more accurately describe the severity of the ATD:s [17]. Also, human stakeholders might provide additional information that involves the context of the debts. For example, rationales about certain decisions might be added that would be impossible for an automatic process to take into consideration. However, even though the positive effects, human elicitations could also lead to the need to interpret the results as the output can vary dependent on the stakeholder. This is not something that needs to be taken into consideration within this process as the indicator has a clear definition of debt.

Another problem regarding prioritization processes is that they need to be easy to understand [8] [17]. This is addressed by following the ISO standard to make the process and all needed components well defined [9]. Each step is explained concretely and the thought is that components should be artifacts known to the case company but also easily understood and/or existing within similar contexts. By including a unified way of interpreting the severity of an ATD, the concept of risk [1], it lowers the chance of misunderstandings.

5.3 New approach of prioritizing

The approach of prioritizing is meant to address the problem of being able to distinguish the amount of extra effort generated from different ATD items. By creating this approach, the thesis provides an answer to research question two. As the prioritization is relative, it is not exact numbers meant for monetary calculations but rather indicators on where to start investigations about refactoring. It is more associated with the established term interest, as it represents the extra effort caused by a dependency. The factor principal is not visualized with this indicator as no concerns are taken about the cost of resolving a violation. This require extra efforts in form of human elicitation or historical data on previously fixed ATD:s [17]. However, the relative prioritization provides enough information since it allows the ATD:s to be sorted in an order of most to least severe and keeps the required efforts in measuring to a reasonable level. It is crucial to keep efforts down since software projects often are suffering from budget and time constraints which has the effect that not all debts can be afford to solve [6] [8]. In those situations the important thing is to fix the most severe debts first which requires difficult decisions to be made [8]. This approach of prioritizing provides more information to the management who is responsible for making those decisions so that they can stand on a more solid ground when facing this kind of problems. Gradually, this idea is a first step to making refactoring activities as efficient as possible by fixing the dependencies that are generating the most amount of extra effort first. The following sections will discuss three different benefits that come from this approach. Firstly, the approach becomes less prone to produce much overhead when human estimations are removed. Secondly, the approach visualizes severity of the debts in a simple way. Lastly, this visualization can in turn lead to that the downside effects of the debts are lowered.

5.3.1 Removing human estimations produces less effort when prioritizing

Several studies have concluded that gathering the information needed to prioritize Technical Debts (TD) is hard and time-consuming [8] [17]. This information is often based on manual estimations that requires time and tacit knowledge about the effects that the debt generates. This thesis presents a way to prioritize specific ATD:s without any need for human estimations. As the approach is based on an empirically validated and well defined research within the same context it enables for less confusion about the prioritization [1]. It is not affected by any human bias or tacit knowledge that can be hard to interpret. Another benefit is that since the need for human estimations is eliminated, the effort for those estimations is eliminated as well. And since the effort for prioritizing is lowered, the risk for that the estimations could produce too little value contra its effort becomes less likely to occur. That is recognized to be a threat to all TD management processes [8]. At the moment, the idea is semi-automatic, and should it be made fully automatic, these benefits would increase even more. However, that is a subject for further research and what should be automated is discussed in Section 5.6.

5.3.2 Connecting the risk to the architecture visualizes the severity of an ATD

The approach combines an empirically validated measurement from another study which includes an automatic tool to determine difficult to maintain parts in a system which is a sign of that those parts are generating extra effort [1]. The study focuses on the calculations for source code files, however this thesis takes the concept one step further by connecting the risk of the files to their respective architectural components. The connection between source code elements and components is a key point for visualizing the effect of an ATD as it handles the problem of defining the abstraction level for where the debt exists [5] [4].

5.3.3 Awareness of the severity obviates the negative effects of hidden ATD

Many studies show that when TD:s are hidden in the software and are left unmanaged, they will become more severe after time [4] [3] [16]. Through prioritizing ATD:s, the severity of how much extra effort the debts are generating will become visible. This would help managers, designers and developers to pay more attention to the debts. By being aware of the severity of the debts in the software it will be easier to plan refactoring activities in order to pay of the debts in time. And, being able to spot these problems before they generate too many negative effects will help to keep costs related to maintenance or developing of new features down so they won't run over budget and help to deliver software on time since time delays also would be lowered [6].

5.4 Validation of identified violations

The results from the second method shows the existence of ATD within the case system. By establishing architectural rules and implementing these rules in a self-developed tool the thesis managed to identify 123 violations. The risk for each Low Level Component (LLC) was successfully calculated by running the Risk Measurement System developed by [1] and aggregating the risk to the required abstraction level. Each violation could thereby be connected to a risk to indicate its priority against others. The next sections will discuss potential threats for that the identified violations could be false. This is important to take into consideration since it could provide false information about the severity to the stakeholders. Additionally, the discussion addresses how the validation interview revealed that unknown non-allowed dependencies existed in the system.

5.4.1 Threats to the process of identifying violations

As dependencies between components cannot exist without the specific build file there is little threat of missing results of the specific type. However, a threat lied in flawed algorithms in the program which was prevented to the greatest possible extent by testing.

During the validation interview one item from rule 4 was discarded for including a dependency that was not necessarily faulty. This was because the platform rule considered dependencies within itself to be incorrect while in reality it is allowed. The remaining 122 violations could not without code inspection be excluded as false positives and were therefore validated as identified possible ATD . However a small threat still lies in dependencies being unused, this can only be addressed by looking at the actual code. In that case the violation would not be ATD as it does not generate any additional effort but still introduces an option for using a non-allowed dependency. During the interview, the architects raised suspicions about some violations but they were not sure enough to exclude them without further investigation. A possibility for not excluding violations could be the simple explanation that they were not aware of the existence rather than them being faulty. The results could however be somewhat skewed due to the fact of that it wasn't possible to easily exclude unused dependencies and no conclusions could be drawn about the existence of them.

5.4.2 Validation revealed the existence of unknown ATD

The presented results were overall considered as good by the architects as it were a mix of known and unknown violations. The unknowns proves that RuleValidator elicits ATD that wouldn't have been found qualitative approaches such as interviews. The fact that violations came as a surprise even to experts shows that ATD is in fact not that visible [13]. However, one can argue that the size of the system affects how much knowledge one can have about the violations within. One architect actually managed to point out all the violations for a small subset of the system before the interview. Unfortunately the same kind of data for the whole system could not be achieved due to time constraints of the architects.

5.5 Validation of the prioritization

This section discusses how the prioritization was validated. The discussion explains how the study has provided extra knowledge about debts for the case company along with suggestions for modifications and extensions of the Measurement System (MS). The results from the comparison showed some mismatches in the prioritization between the architects and the MS. A common recurring pattern for the the architects was to reverse the order for the top and bottom items. For each rule only a few dependencies was exactly right. However, the results in between the top and bottom showed to be more accurate. The approach of using exact numbers for each violation most likely made it even harder, a more fair way of doing it would have been to use high, medium or low. After showing and discussing the architects estimations against the prioritization from the MS, the stakeholders reached the conclusion that the indicated risk showed a good representation of violations. As stated by Staron et. al. [33] one can assume that the results are empirically valid as they were accepted by the stakeholders. This proves that the results gives new insight and contributes to the company. During the

interview the architects stated a threat in unknown dependencies as they could introduce errors in estimations as well as unknown side-effects. This shows a twofold value for the results and gives strength to both indicators for this process. The resulting violations are without taking the risk into account providing value as they could reveal the unknown, which are the most severe ones [6]. This confirms that both research questions in this thesis are answered as the prioritization and the identification provided useful results.

5.5.1 Suggestion to gain knowledge about the increase and decrease of the ATD:s

Something not stated during the interview but addressed by company personnel and supervisors during ad-hoc conversations is the ability to gain the delta for change in number of violations. This would give simple indicators about if the situation is getting worse or better which is recognized as a good metric at the case company. A new proposed MS to meet this information need that could be validated by further research is presented in Section 5.7.

5.5.2 An extension of the prioritization metric

A further outcome from the interview was a suggested addition to the prioritization indicator. As the risk in the source LLC of a dependency is triggered by a change in the destination component the frequency of change could be included in the prioritization. As stated by one architect: *"You want to have a dependency in the direction towards things that are stable, namely things that doesn't change that often".* The extension to the metric would add more precision to the prioritization and thereby provide even greater value for the company. Sommerville [27] states that a change in the destination component (destination LLC for a dependency) may produce extra effort in the source component due to the usage of functionality in the changing unit. Change proneness is also known to be strongly related to modularity violations [16]. These two statements strengthen the beliefs for this new proposed metric.

The calculation of the new indicator was discussed during the interview to be a multiplication of the risk of the source LLC in a violation with the number of revisions for the destination. By including the number of revisions for the destination LLC in the indicator, the architects considered that the likelihood of triggering additional changes in the source LLC due to the dependency would be included. This new metric would lead to violations receiving a more precise value which for example would make it possible to separate violations sharing the same risk. Dependent on the number of changes in the target the prioritizations could be more accurate and more easy to distinguish between.

However, risk is the likelihood of the component to be difficult-to-maintain which is confirmed empirically, while the number of revisions is purely based on historical data [1]. It is therefore not possible to conclude that the target unit is likely to change in the future without carefully validating other aspects that impact the number of changes that will occur. As the time span for the historical data is large, a release period reflecting all stages of development as well as many different feature implementations, it could be the

case that the number of revisions matches the *"reality"* for a component. Therefore, a similar number of revisions should also affect the component in the future. However, this is not proven and further validation and empirical evidence is left for future research before introducing this extension the indicator.

5.6 Automation and information quality of the Measurement System

As the amount of effort put into managing ATD is an important factor, effectiveness of the Measurement System (MS) needs to be considered. And in order for the MS to be effectively used, the process of gathering all the required measurements for the indicators should be automatized. This would enable the tool to run on a repeated basis to collect information which could be used as a snapshot of the system but also to compare historical data with the most recent. This could be a useful indicator the determine if the situation gets better or worse. The scope for this thesis does not include constructing a fully automated tool and due to time constraints the existing solution is only semi-automatic. However, only a few steps involves manual work at the moment, which are:

- Running the MS for calculating non-allowed dependencies and the risk calculation separately.
- Extracting the revision history from the Revision Control System (RCS) that is needed for the risk calculation.
- Calculate the risk for the right abstraction level of LLC:s.
- Mapping the risks to the non-allowed dependencies.

All of the above mentioned points would most likely suit well to be implemented as scripts by an appropriate language. Which specific language that has the most benefits needs to be further investigated. It is the understanding of the thesis workers that a fully automatic system would be achievable and could be done in a reasonable amount of time. It would provide great value to the company to be able to use the MS as a *"one-click"* or a totally automated solution running on a scheduled basis.

5.6.1 How to trust the measurement indicators

An important point to take into consideration to be able to use an automated version of the thesis method is how to establish trust in the indicators i.e. decide on information quality for the base and derived measures. During the thesis all the data were verified manually which would be somewhat time consuming to do for a frequently scheduled task. This should instead be done by automatic checks in all steps of the information gathering process to determine if the indicator is valid. The measures that needs to be validated are presented below:

- *Architectural rules:* Are they up to date or has changes been introduced since last measurement?
- *Source code:* Is the latest version of the source code available?
- *Architectural mapping:* Does the mapping of LLC:s to MLC:s match the architecture of the system or is it outdated?
- *Build files:* Is this file still used for the same purpose/does it contain *"include"* statements?
- *Naming convention:* Does the folder path follow the same name convention? Are the terms MLC, LLC and build file still used in the same way?
- *Revision history:* Is the revision history available for the right time-span?
- *Architectural abstraction for risk:* Are all risk items available and summarized for each LLC?
- *Violations:* Does violations contain a proper destination and source LLC?
- *Rank:* Does the rank contain a proper violation and the associated risk?

By checking these criteria the stakeholders can be sure that the indicators reflects the reality and it would be possible to identify which part of the measurements that needs to be corrected or replaced. The most critical part that could fail is concerning the rules. If a rule needs to be added or changed the source code in RuleValidator needs to be changed. As it is designed each new rule algorithm could just be added with an extra function providing the same output form as the others. However, a proposed way of changning the tool would be to design according to the Strategy design pattern to make it easy to extend with new rules without needing to know or breaking anything in else in the code [35]. Furthermore, it would be preferred to have a Domain Specific Language (DSL) to implement rules without having to change the source code. This is way beyond the scope of the thesis and would probably fit into a master thesis just by itself. To make up for the time spent producing a DSL one would make it generalizable for different contexts, this would however make it more time consuming to set up compared to a quite short implementation time of extending the RuleValidator tool with another algorithm.

5.7 Proposed Measurement System for change in the number of non-allowed dependencies

As explained, an interest in getting an overview of how the amount of non-allowed dependencies changes from week to week was also shown by the company. By following the same ISO standard [9] as the developed Measurement System (MS) by this thesis, a new MS could effectively indicate the change of violations on a weekly basis. The

MS presented in this section has been developed by a researcher at Chalmers from the contributions of this study. The new MS would provide the stakeholders with an indicator that displays the status of the weekly change as shown in Figure 5.1. The stakeholders would in the case company be the architects who are responsible for the architecture along with the developers who would get feedback if they are improving or decreasing the quality of the system. In Figure 5.1, the violations have decreased by 3 since last week, indicating a green arrow pointing downwards. The green OK button indicates that the information quality of the indicator is validated, this will be explained later down in this section.

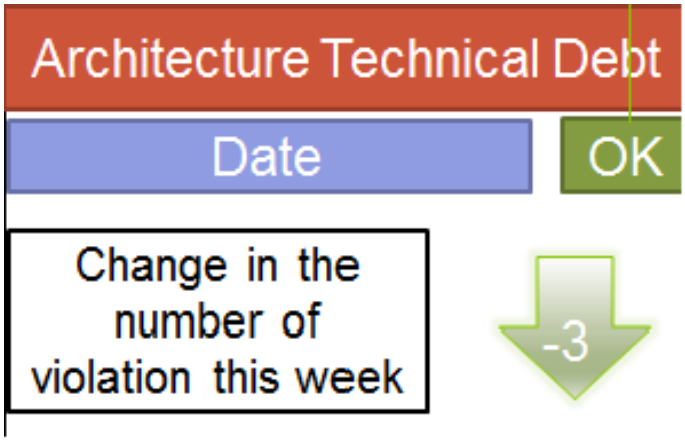

Figure 5.1: An example of how an application of the MS could be display the indicators for the stakeholders

In order to realize the indicator, a model has been proposed which is a modification of the MS used in this thesis. An example output of the model can be seen in Figure 5.1. The base measures which calculates the amount of non-allowed dependencies is still derived the same way as the old MS. That is, by retrieving all dependencies of the system that are located in the build files (explained in Section 3.3) and validate them against the derived architectural rules. The base measures will then be the number of violations of the current week along with the number of violations that existed the previous week. After the base measures are extracted, the derived measure will be the change of number of violations from the last week. This could be implemented with a script that runs these measurements continuously. The change is calculated by subtracting the amount of violations from the previous week from the violations in the current week. The indicator for the stakeholder will use the derived measure in order to display the status of the Change of Violations (CV) which would be stored in a database. As a display for the indicator, the arrow from Figure 5.1 could be used. If the value of CV is smaller than 0, then the ATD has decreased and the arrow will be green. If CV is greater than 0, then the arrow will be red, indicating an increment of the ATD. Should CV be equal to 0, the ATD would be stable and a yellow symbol would be shown. As an action plan for the responsible for the MS which in this case would be the architects

the following interpretation would be made from the indicator; red would mean it would be necessary to check with developers and follow up the necessity of the newly added dependencies, yellow would mean that no action is required since the ATD is stable and a suitable action for the green status would be to commend developers that removed dependencies during the week.

As displayed in Figure 5.2, the green OK label shows the status of the information quality. The purpose of this is to give information about whether the indicator that indicates the change of non-allowed dependencies can be trusted. In order for that indicator to be trusted, certain information quality criteria has to be fulfilled which are listed here:

- The architectural rules used to verify dependencies have to be up to date. If the rules has changed over the last week and nothing has been updated in the MS, this criterion is not fulfilled.
- The architectural mapping has to be consistent. If new elements or structures has been included, the architectural mapping is not valid anymore and has to be updated. If not, this criterion is not fulfilled.
- All functions that are used for calculating the amount of non-allowed dependencies for the previous and current week, along with the change function has to be continually tested. If any of these tests would fail, this criterion would not be fulfilled.
- The application that displays the indicator for the stakeholders has to verify that the correct data has been received from the measures.

If the verification of any of the above mentioned criteria would fail, the symbol displaying the information quality would report ”NOT OK” in red color. Else the status would be *”OK”* in green color. This new MS would help the responsible stakeholders to keep track of the status of this specific ATD. The definition of having technical debt in the system is that it generates interest in the form of extra effort that has to be put in to for example maintenance. This MS could help the architects to get an overview of when the amount of ATD has increased so much that refactoring activities has to be scheduled in order to increase the development velocity. Since the MS also could be fully automated, only a small amount effort has to be put in for which the MS provides an efficient way of tracking and managing the ATD. However, further studies has to be made in order to validate and investigate this MS. The validation has to make sure that the MS provides the stakeholders with useful information and preferably, in-depth case studies has to be made to see what benefits can come out of using the MS.

5.8 Main contributions to the company

The main contributions towards the case company consist of increasing their knowledge about ATD both by presenting the theory behind it but more importantly by identi-

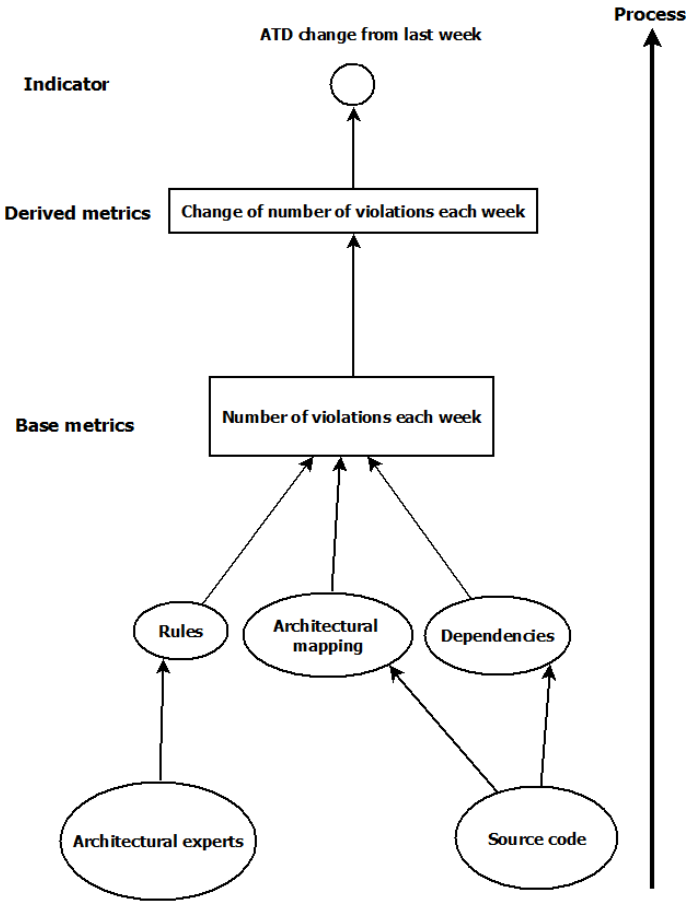

Figure 5.2: The model of the proposed MS

fying concrete debts in their system. The results of this study show an increment in knowledge as all violations were not known beforehand. By prioritizing the non-allowed dependencies the architects have gained more insight about the severity and where to start refactoring work. The thesis provided the case department with a result from the MS, a prioritized list of violations for each rule, which gives them the status of the products health as well as the previously mentioned insights. The list has already been contributed to all people working with the product by the architects in their weekly newsletter.

The thesis also provides the company with a semi-automated process for identifying and prioritizing violations in the future. It includes a manual for the MS along with guidelines on how to completely automate the process and to ensure information reliability in the indicators. The created tool is also a contribution to the case company as it is developed specifically for them with architectural mapping already included. People involved in the thesis have stated that the process and the tool is interesting and useful and that there is a chance that someone will take over the work to develop it further. It

has also been stated that it could be integrated in the company's Continuous Integration process.

The thesis also provides with an external idea from the department of Software Engineering on how to create a new MS (based on the MS from this thesis) for the number of changes of non-allowed dependencies, see Section 5.7. The company has stated the importance to visualize the delta of existing versus historical violations. This could be shown to all involved personnel to get an understanding of the product health and the progress of removing ATD. It also works as a way to create an awareness about introducing new ATD to the system.

Furthermore, another contribution is the first method of the thesis. Even though it did not produce any results the company still showed an interest and proposed that the prerequisites could help to create checklists or templates on how to do estimations in order for the method to work. As a consequence of creating and documenting architectural rules awareness have been gained within the company both for architects but also for all the people that have been in contact with the thesis work or any held presentation.

5.9 Releated work

A few studies have been made which focus on identifying ATD in different ways. Mo et. al. [5] argue that the ATD:s are not as clear and well-defined as code debts and can have more severe consequences. They propose a generic model called Extended Augmented Constraint Network (EACN) that models the architecture. This model is mapped to specific Architectural Decay Instances (ADI) and the purpose is to find an uniformed way to identify ATD:s which then could be automated. The proposed model is still in the work for being tested in real-life context and it does not focus on how to measure the effect of ATD:s as this thesis is doing.

Another similar study tries to identify Architectural Bad Smells that can be a symptom of ATD:s [21]. The study states that in order to identify the bad smells, there needs to be a definition of what is to be identified. This led to a separation of four different bad smells namely; connector envy, scattered functionality, ambiguous interfaces and extraneous connector which could help practitioners in order to identify their ATD:s. However, the purpose is to find generic definitions for all types of ATD:s and not to measure and prioritize a certain type, which this thesis focuses on.

Moreover, a large scale case study has been conducted that included strategies to identify design flaws by using historical data about the design [36]. The source of these design flaws is non-optimal design decisions that contributes to maintainability problems. They conclude that the flaws can be more accurately defined if the historical versions of the software is included in the process. However, the study focuses only on finding high-risk god classes (i.e. classes that performs too much work) and data classes (i.e. classes that only have setters and getters and nothing more) and does not focus on any form of relationships, such as dependencies, between the classes in order to identify debts.

Zazworka et. al. [18] also tries to identify ATD by looking at the impact that the debt items can have on the quality of the system. Here, the focus is also on god classes

which are not the similar type of ATD as this study focuses on.

Nord et. al. [4] conducts a small case study where they try to find a metric for managing Technical Debt (TD) by analyzing two different development paths on an existing system. They focus on an architectural level rather than code by measuring dependencies between elements and change propagation to calculate the rework cost. Since change propagation also is an effect of non-allowed dependencies that can generate extra effort that is used in this thesis, it can be seen as similar. However, as mentioned, the change propagation in that study is used to compare two different development paths and not to compare different ATD items.

Another study compares TD identification techniques and the study includes a tool named CLIO [16]. It identifies modularity violations which means violations in the software that deviates from the intended one. Non-allowed dependencies can by this definition be seen as a modularity violation. However, CLIO identifies the debt items by identifying classes that change a lot together and from there creates and categorizes different types of dependencies [30]. This approach differs from the thesis since the abstraction level is lower, source code files instead of components. The major difference is however that the angle of contact is not the same, the thesis states the rules beforehand and only finds actual violations, CLIO on the other hand does not automatically filter dependencies that are permitted.

Zazworka et. al. [17] has compared many different identification and prioritization techniques for TD. For prioritizing TD, one of these techniques measures the complexity of the source code which is a part of the prioritization process in this thesis as well. Nonetheless, the complexity is as stated only a part of the prioritization and more metrics are included here. Furthermore, this thesis focus on prioritizing non-allowed dependencies rather than actual source code files.

Seaman et. al. [8] has done a study which compares different prioritization strategies for TD. The study includes evaluations on cost-value, Analytic Hierarchy Process, Portfolio- and Option approaches. The purpose is to find a process that can be both efficient and effective. Yet, they conclude that all methods has to be evaluated in more in-depth case studies to gain knowledge about which strategy works best in which situation.

6

Conclusions

Architectural Technical Debt (ATD) in the form of non-allowed dependencies is recognized to be a major problem in software projects today. Often, these types of dependencies are unknown which can cause a lot of unwanted extra maintenance. As the debt increases, more time will be spent on maintaining the system which means that the software project will become less effective and delaying processes such as releasing new features. In order to avoid these effects, projects need to monitor their debt situations. This thesis aimed at finding methods that could help when identifying and prioritizing non-allowed dependencies. The study was conducted in a real life context and, by validation with stakeholders, it successfully identifies and proposes a technique to prioritize non-allowed dependencies. The key findings of this work are:

- *A proposed method that analyses features within a product to identify non-allowed dependencies.* The method does not involve any complex tools and involves straightforward steps that could identify ATD. It clearly describes the necessary prerequisites needed in order to perform the method that is aimed to answer RQ 1. The case company has confirmed that the method could very well be useful if they would meet the prerequisites in the future. Further research should, preferably through in depth case studies, investigate what benefits this method could contribute with in other contexts.

- *A Measurement System (MS) developed according to the ISO standard 15939:2007 that identifies non-allowed dependencies within a system.* The method has defined distinct prerequisites, components and steps to follow which clearly describes all parts needed to identify ATD which answers the first RQ of this thesis. The result of the MS is dependencies that violate architectural rules. The validation shows an increment in knowledge for the case company as it consists of both known and unknown non-allowed dependencies. Further research could use the description of

this method to explore how it works in other contexts. Preferably in case studies with different environmental settings than this case company. Those studies could lead to that more general conclusions about this MS could be drawn.

- *A MS developed according to the ISO standard 15939:2007 that prioritizes non-allowed dependencies within a system.* This is the second indicator of the MS which is calculated after ATD identification in order to prioritize the debts to answer the second RQ. This is based on a new approach of prioritizing ATD:s based on the risk of adding extra effort in the form of increased maintenance and development time as a consequence of the dependency. By connecting the risk of source code files to an architectural component, the study has enabled prioritization of this type of ATD as a part of answering RQ 2. The definition of risk has been established and empirically validated by a previous study. Moreover, this MS also proves to add additional knowledge to architects about the severity of their non-allowed dependencies within their system. The architects suggested that the metric used for prioritizing could be extended. Suggestions for further research involves, among others, to analyze this prioritization approach in cases that holds other conditions and to validate a proposed extension of the metric.

To sum up, the findings made by this work will help the field of Software Engineering in identifying and prioritizing ATD:s. By being able to identify and monitor the debts in the architecture, one will become closer to understand the current debt situation by making them visible. And once the problems are visible, suitable actions such as refactoring activities can be performed if needed. By prioritizing the non-allowed dependencies, these actions will be based on more well-grounded decisions. This will lead to a better way of managing the threats that the ATD:s brings along.

Bibliography

[1] Antinyan V, Staron M, Meding W, et al. Identifying risky areas of software code in Agile/Lean software development: An industrial experience report. In: Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on. IEEE. Antwerp; 2014. pp. 154–163.

[2] Guo Y, Seaman C. A portfolio approach to technical debt management. In: Proceedings of the 2nd Workshop on Managing Technical Debt. ACM. Honolulu; 2011. pp. 31–34.

[3] Tom E, Aurum A, Vidgen R. An exploration of technical debt. Journal of Systems and Software. 2013;86(6):1498–1516.

[4] Nord RL, Ozkaya I, Kruchten P, Gonzalez-Rojas M. In search of a metric for managing architectural technical debt. In: Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), 2012 Joint Working IEEE/IFIP Conference on. IEEE. Helsinki; 2012. pp. 91–100.

[5] Mo R, Garcia J, Cai Y, Medvidovic N. Mapping architectural decay instances to dependency models. In: Managing Technical Debt (MTD), 2013 4th International Workshop on. IEEE. San Fransisco; 2013. pp. 39–46.

[6] Martini A, Bosch J, Chaudron M. Architecture Technical Debt: Understanding Causes and a Qualitative Model. In: 40th EUROMICRO Conference on IEEE Software Engineering and Advanced Applications. Verona; 2014. pp. 1–8.

[7] Lim E, Taksande N, Seaman C. A balancing act: what software practitioners have to say about technical debt. Software, IEEE. 2012;29(6):22–27.

[8] Seaman C, Guo Y, Izurieta C, et al. Using technical debt data in decision making: Potential decision approaches. In: Managing Technical Debt (MTD), 2012 Third International Workshop on. IEEE. Zurich; 2012. pp. 45–48.

[9] Comission ISOE. 15939:2007 - Systems and Software Engineering - Measurement Process. 2007;.

[10] Cunningham W; ACM. The WyCash portfolio management system. 1992;4(2):29–30.

[11] Kruchten P, Nord RL, Ozkaya I. Technical Debt: From Metaphor to Theory and Practice. IEEE Software. 2012;29(6):18–21.

[12] Izurieta C, Vetro A, Zazworka N, Cai Y, Seaman C, Shull F. Organizing the technical debt landscape. In: Third International Workshop on Managing Technical Debt (MTD). IEEE. Zurich; 2012. pp. 23–26.

[13] Kruchten P, Nord RL, Ozkaya I, Falessi D. Technical debt: towards a crisper definition report on the 4th international workshop on managing technical debt. ACM SIGSOFT Software Engineering Notes. 2013;38(5):51–54.

[14] Fowler R. Technical Debt Quadrant; 2009 (accessed 2014-02-03). Available from: `http://www.martinfowler.com/bliki/TechnicalDebtQuadrant.html`.

[15] Eick SG, Graves TL, Karr AF, Marron JS, Mockus A. Does code decay? assessing the evidence from change management data. Software Engineering, IEEE Transactions on. 2001;27(1):1–12.

[16] Zazworka N, Izurieta C, Wong S, et al. Comparing four approaches for technical debt identification. Software Quality Journal. 2013;pp. 1–24.

[17] Zazworka N, Spínola RO, Vetro A, Shull F, Seaman C. A case study on effectively identifying technical debt. In: Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering. ACM. Porto de Galinhas; 2013. pp. 42–47.

[18] Zazworka N, Shaw MA, Shull F, Seaman C. Investigating the impact of design debt on software quality. In: Proceedings of the 2nd Workshop on Managing Technical Debt. ACM. Honolulu; 2011. pp. 17–23.

[19] Riaz M, Sulayman M, Naqvi H. Architectural decay during continuous software evolution and impact of 'design for change'on software architecture. In: Advances in Software Engineering. Springer; 2009. pp. 119–126.

[20] Farid H, Azam F, Iqbal MA. Minimizing the Risk of Architectural Decay by using Architecture-Centric Evolution Process. International Journal of Computer Science, Engineering and Applications. 2011;1(5):1–12.

[21] Garcia J, Popescu D, Edwards G, Medvidovic N. Identifying architectural bad smells. In: Software Maintenance and Reengineering, 2009. CSMR'09. 13th European Conference on. IEEE. Kaiserslautern; 2009. pp. 255–258.

[22] Buschmann F. To pay or not to pay technical debt. Software, IEEE. 2011;28(6):29–31.

[23] Guo Y, Seaman C, Gomes R, et al. Tracking technical debt—An exploratory case study. In: Software Maintenance (ICSM), 2011 27th IEEE International Conference on. IEEE. Williamsburg; 2011. pp. 528–531.

[24] Rieger M, Ducasse S, Lanza M. Insights into system-wide code duplication. In: Reverse Engineering, 2004. Proceedings. 11th Working Conference on. IEEE. Delft; 2004. pp. 100–109.

[25] Martin RC. Design principles and design patterns. Object Mentor. 2000;pp. 1–34.

[26] Beck F, Diehl S. On the congruence of modularity and code coupling. In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering. ACM. Szeged; 2011. pp. 354–364.

[27] Sommerville I. Software engineering. Harlow: Addison-Wesley; 2011.

[28] Runeson P, Höst M. Guidelines for conducting and reporting case study research in software engineering. Empirical software engineering. 2009;14(2):131–164.

[29] Kitchenham B, Pretorius R, Budgen D, et al. Systematic literature reviews in software engineering–a tertiary study. Information and Software Technology. 2010;52(8):792–805.

[30] Wong S, Cai Y, Kim M, Dalton M. Detecting software modularity violations. In: Proceedings of the 33rd International Conference on Software Engineering. ACM. Honolulu; 2011. pp. 411–420.

[31] Dependometer; 2013 (accessed 2014-05-12). Available from: `http://source.valtech.com/display/dpm/Dependometer`.

[32] SonarQube; 2014 (accessed 2014-05-12). Available from: `http://www.sonarqube.org/`.

[33] Staron M, Meding W, Karlsson G, Nilsson C. Developing measurement systems: an industrial case study. Journal of Software Maintenance and Evolution: Research and Practice. 2011;23(2):89–107.

[34] Stober T, Hansmann U. Agile software development: Best practices for large software development projects. Berlin, Springer; 2009.

[35] Christopoulou A, Giakoumakis EA, Zafeiris VE, Soukara V. Automated refactoring to the Strategy design pattern. Information and Software Technology. 2012;54(11):1202–1214.

[36] Rapu D, Ducasse S, Gîrba T, Marinescu R. Using history information to improve design flaws detection. In: Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings. Eighth European Conference on. IEEE. Tampere; 2004. pp. 223–232.