Software maintenance tools selection and implementation

A case study based analysis

Authors:

Klara Jakobsson

Matilda Wiklund

Supervisor:

Jordi Olivella

Research Assignment

Universitat Politècnica de Catalunya

Barcelona, 2022/23

Acknowledgements

We would like to thank our supervisor Jordi Ovella for valuable feedback and help with our research. It has been an inspirational and educational journey that opened up new areas of interests for us. We would also like to thank all the participants in our case study for sharing your knowledge and reflections with us.

Abstract

This study investigates the reasoning in selection and implementation of software maintenance methods and tools and identifies potential barriers for successful adoption. This is done through qualitative interviews with practitioners of software maintenance in industry and in accordance with the existing literature. One main reason was found why a certain maintenance tool was chosen, the usability of the tool. However, the popularity of the tool was also a contributing factor for tool selection. Furthermore, three key issues were identified for successful implementation and adoption of the tools: limited awareness of the tool's availability, confusion about its intended usage, and lack of understanding on how to properly utilize it. It is left to the practitioners and stakeholders to weigh and evaluate the significance of the individual reasons. The study is based on empirical data collected over the four months the study lasted and is intended to contribute to the overall knowledge of software maintenance management, processes, and tools.

Keywords: Software Maintenance, Software Maintenance Tools, Software Evolution, Software Maintenance Processes, Software Product Management, SonarQube.

Wordlist

- *Technical debt/Tech debt* - A metaphor to describe the effect of immature artifacts on software maintenance that contributes to short-term benefit to projects in terms of increased productivity and lower cost, but that might have to be paid off with interest later (Alves et al., 2016).
- *Bad smells* - Indicators of bad code in an object-oriented (OO) context. The bad smell is supposed to help developers decide when software needs refactoring (Mantyla et al., 2003).
- *CI* - Continuous Integration, which means frequent merging of several small code changes into a main branch.
- *Cyclomatic Complexity* - McCabe's cyclomatic complexity metric is a measure of the maximum number of linearly independent circuits in a program control graph. One of the primary purposes of the metric is to identify software modules that will be difficult to test, hence, particularly interesting both for researchers and practitioners concerned by software maintenance (Gill & Kemerer, 1991).
- *DevOps* - A software engineering methodology used to facilitate collaboration and shared responsibility between software operations and development by integrating their work. It aims to provide continuous delivery with high quality and can be a complement to agile software development.
- *Integrated Development Environment (IDE)* - A software application that offers several tools to programmers for software development directly in the environment. Source code editor, compiler, debugger, and build automation tools are examples of tools provided by an IDE. The toolbox differs between different IDEs.
- *Bugs* - Errors, flaws or faults in the design, development or operation of a software that lead to unexpected and unwanted results or behaviors.
- *Refactoring* - Improving and restructuring code without changing the external behavior.
- *Regression Testing* - A method within software testing that aims to test the system when new functionality is developed. The regression tests run after every change to deal with the emergence of old bugs when new code is introduced.
- *Software Maintenance* - Working with and developing a software without adding new functionality to increase its quality.

Table of Content

1. Introduction **5**

2. Concepts **6**

2.1 Software Evolution 6

2.1.1 Architecture-Driven Modernization 6

2.2 Software Maintenance 7

2.2.1 Maintenance Tools 9

2.2.2 SonarQube 10

3. Literature **12**

4. Objectives **15**

4.1 Research Questions 15

4.2 Delimitation 15

5. Methodology **17**

5.1 Literature Review 17

5.2 Case Study 17

5.2.1 Practical Maintenance Work 17

5.2.2 Interviews 18

6. Fieldwork findings **20**

6.1 Company overview 20

6.1.1 Description of the company 20

6.1.2 Organizational structure and roles related to software maintenance 21

6.1.3 The company's software maintenance processes and practices 21

6.2 Practical Maintenance Work 21

6.3 Interviews 22

6.3.1 Interviewee 1 22

6.3.2 Interviewee 2 23

6.3.3 Interviewee 3 23

6.3.4 Compilation of interview findings 23

7. Discussion **32**

8. Conclusions **37**

9.1 Recommendations for the company 38

9.2 Future Research 40

9.3 Threats to Validity 40

References **42**

1. Introduction

Software systems need to evolve to stay up to date with current requirements and not lose market share to competitors (Lehman et al., 1997). Faulty software systems cost a lot monetary wise, are time-consuming, and can be a source of security vulnerabilities. Besides this, customer needs are changing over time, the code base gets larger, and technology evolves. Thereby, released software needs to evolve over time and be maintained. Maintaining large systems is difficult, complicated, and time consuming. Adding new features and fixing defects become more complicated as time goes by and as the software system grows (Godfrey & Tu, 2000). A lot of software development resources are put on maintenance, where maintenance tools are helpful for effective maintenance processes. However, the gap between software research proposals and industry practices has been found to be large. Research does not know how industry works with maintenance and the industry does not follow what is suggested by research (Ferreira et al., 2021).

Several tools have been developed to make the maintenance work more effective. These tools are more or less automated and can improve the software quality if a suitable tool is chosen and successfully adopted. The question is why a certain tool is chosen and what barriers there are for successful implementation and adoption? By answering this, many software systems would live longer.

2. Concepts

This section provides the theoretical background of the study. Firstly, the concepts of software evolution and software quality are defined followed by a presentation of the most commonly used software evaluation and maintenance tools of today. Lastly, the software tool mainly focused on in this study, SonarQube, is introduced.

2.1 Software Evolution

Software systems are constantly evolving as the evolution of user requirements can not be predicted. Delivered systems are never complete and will change and grow with time to not get obsolete. Software evolution aims to ensure functional relevance, reliability, and flexibility of a system. Every time a change is made, or a new piece of code is added, the software evolves. The software evolution process can be fully manual or partially to fully automated, depending on how much of the activities are made by human developers or autonomous tools. Lehman (1980) linked the behavior of real-world systems strongly to the environment in which they run, where these systems need to adapt to non-constant requirements and circumstances in their specific environment. The eight observations are referred to as Lehman's Laws and predict that the need for functional change is inevitable. Changing a software system is not a consequence of insufficient requirements analysis or inadequate programming, but fully natural. However, a growing code base also increases in complexity if no refactorization is made (Lehman, 1980).

Lehman's case studies of large software systems suggest that it gets more difficult to add new code as the system grows and that reorganizing the system design is a way of coping with this problem (Lehman et al., 1997). Turski's statistical analysis of Lehman's case studies (Lehman & Beladi, 1985) suggests that software systems grow at a slower pace as it gets more complex and larger (Turski, 1996).

2.1.1 Architecture-Driven Modernization

Architecture-Driven Modernization (ADM) is a concept presented by a task force within the international technology standards consortium named the Object Management Group (OMG). ADM is described as the process of understanding and evolving existing software assets for the purpose of software improvement, modifications, interoperability, refactoring, restructuring, reuse, porting, migration, translation into another language, enterprise application integration, service-oriented architecture, and Model Driven Architecture (MDA) migration. Modernization starts when the current system and processes fail to deliver against the requirements. The mission of the ADM task force is to develop specifications and work towards industry consensus on modernization of software in use (Object Management Group [OMG], 2012).

2.2 Software Maintenance

Software system quality usually degenerates as the system is subjected to changes during the course of its lifetime, hence maintenance is a necessity. Ideally, maintenance planning should, according to the international software maintenance standard by IEEE (2022) begin during planning for software development. Software evolution and maintenance are characterized by their huge cost and slow speed of implementation but all successful software needs it (Bennett & Rajlich, 2000). For instance, Mantyla et al. (2003) states that “Microsoft uses 20% of its development effort to re-develop the code base of its products.” It is needed to correct faults, adapt the software product to a modified environment and improve performance.

Software maintenance is the collective term for changing, modifying, and updating software after the system or product has been delivered. The objective of the maintenance process is mainly to keep up with requirements derived from customers’ needs. Any successful piece of software needs to be maintained, and over 90 % of the costs of a system typically arises in the maintenance phase (Brooks, 1975). A system’s or product’s maintainability is a measurement of how effective and efficient it can be modified. When proposing change to a service or product, the information and description is called a modification request or a change request. These requests can be classified as different types of maintenance to enable application of problem prioritization methods, root cause analyses and failure recurrence prevention (Swanson, 1980). Swanson (1980) divided software maintenance of a system into three different types, based on the maintenance intentions:

- Adaptive maintenance, which is to adapt to changes in the data environment or processing environment, i.e., keeping the software usable after delivery and in pace with the changing environment.

- Corrective maintenance, which is to correct processing, performance, or implementation failures.
- Perfective maintenance, which is to perfect performance, processing efficiency, or maintainability, as well as providing enhancements for users.

More recently, ISO/IEC/IEEE (2022) have extended this division further, by first splitting up modifications into either a Correction or an Enhancement, to then split each classification into subtypes of maintenance, where Adaptive maintenance can belong to both parent types.

- Correction, which is to make changes that make the software meeting defined operational requirements.
    - Corrective maintenance, which is defined by Swanson above.
    - Preventive maintenance, which is to correct latent faults in the software after delivery before they occur in the live system.
    - Adaptive maintenance, which is defined by Swanson above.
- Enhancement, which is to make changes that make the software meeting new requirements.
    - Additive maintenance, which is to add functionality or features to enhance the usage of the software after delivery.
    - Perfective maintenance, which is defined by Swanson above.
    - Adaptive maintenance, which is defined by Swanson above.

However, ISO/IEC/IEEE (2022) highlights that the practices differ between different organizations. For example, adaptive is not considered an enhancement in some organizations. Each type can also be divided into “scheduled”, “unscheduled” and “emergency”. Another alternative is to classify them as “reactive” or “proactive” maintenance, where corrective and adaptive types fall under reactive, while preventive, perfective, and additive count as proactive. (ISO/IEC/IEEE, 2022) In general, the software maintenance procedures followed in practice lack clear definitions. Reports show that few organizations adopt a separate maintenance process since they struggle with distinguishing software maintenance from software development. (Khan et al., 2001)

The three key elements of software maintenance are the people involved, the supporting tasks, and the knowledge about the software product. These three are interrelated and interdependent and vary in importance between different projects. The supporting tasks should be well defined and each task requires one or more components. A task performs a function and supports activities of other tasks which all aim to achieve the predefined objectives. Each task is implemented by certain methods, and the methods are supported by human interactions and automatic tools (Khan et al., 2001). The tools will be further described in the following section, 2.2.1 Maintenance Tools.

2.2.1 Maintenance Tools

A lot of different tools can be used for software maintenance even though all are not specifically aimed for the maintenance process (Khan et al., 2001). Automated toolsets and manual procedures can be used separately or combined to support the tasks but many environments in software engineering lack functionality for successful integration and coordination of the tools within a project (Sharon et al., 1997). This integration and coordination of tool environments with software projects has been raised as an important challenge by research (Khan et al., 2001).

The applicability of the tools is another critical aspect, since they must be easily adapted to many types of projects if the maintenance process should be kept operative for a long time. Another important aspect is that the maintenance tools must fit into the maintainers' culture and support the techniques and methods used by the programmer. Otherwise, the acceptance for and the adoption of the tools can be difficult (Khan et al., 2001).

There are many different types of activities that fall under the definition of software maintenance. Ferreira et al. (2021) mentions a few: change impact analysis, log of modification requests, modification of code and other artifacts, program comprehension, reverse engineering, measurement, migration, tests, training, and daily support. The high-level software maintenance process shown in Table 2 focuses on how various tasks are performed in the maintenance-related chain. Table 2 is a two-dimensional matrix where the first column shows all components required for a given task. Each of the remaining columns maps to individual tasks and their required components.

Table 2: Applied version of tasks of Software Maintenance Process Infrastructure, Khan et al., 2001

Software Maintenance Tasks					
Components of the tasks	Maintenance Requirements analysis	Determination	Program comprehension	Localisation and impact analysis	Generating test cases
Objectives	Trigger of the process enactment	Examines the technical feasibility	Understanding semantics and architecture of the software	Identifying program location and ripple effects	Tests cases defined for proposed changes
Sources of input	Program execution at the operational suite Real users of the system	Requirements specification, knowledge on software and its nature and characteristics, organizational policy, status of tools and staff ability	Source code, Information from original designer and programmers readable from program documents	Requirements specification, source code, class hierarchy, function call sequences, data structures, data file format	Req.spec. sources code function names, variables used
Output	Refined maintenance requirement specification	Requesting to filter req. spec. termination message, filtered requirements, primary knowledge about the software	Recovered system design artifacts, Program domain	Function names, variables declarations	Test data, program path spec.
Methods	Interviews, Prototyping	Verify requirements	Program walk through, Program slicing, Execution of program	Program walk through, program slicing, execution of program	Regression testing, quality control
Tools	Not specific	Cost estimation software	Reverse engineering, Design recovery, Debugging, Static analyser	Code analyser, Design recovery, Reverse engineering tools	Test tools

2.2.2 SonarQube

SonarQube is the most frequently used software maintenance tool in industry with more than 85.000 organizations using it (Lenarduzzi, 2020). It is an automatic static analysis tool, which is useful for improving internal quality attributes as code violations are revealed in a cost efficient way since there is no need to run the program.

Automatic static analysis tools like SonarQube can be used to automatically highlight performance bottlenecks, identify refactoring opportunities, detect security vulnerabilities, and bad programming practices like code smells (Marcilio et al., 2019). SonarQube analyzes code compliance according to defined rules. If some code violates any of these rules, an estimation of the time needed for refactoring that code is added to the technical debt. Some of the rules are identified as "bugs", which means that they are wrong in some way that soon will be seen as a fault (Lenarduzzi, 2020). SonarQube claims that no false positive bugs are expected at all and is a tool that can analyze more than 30 programming languages and be integrated to the continuous integration (CI) pipeline and DevOps platform (SonarQube, 2023).

Static analysis is included in one of the important principles of CI called continuous inspection. The static analysis should, together with other assessments, be done every time the software changes. There are many benefits with using automatic static analysis tools, but also some challenges. One challenge is the often large amount of false positive code violations, which Johnson et al. (2013) reported to be thousands of. Another challenge is to find the defects considered worth fixing, since developers often ignore violations as a praxis (Wang et al., 2018).

3. Literature

In this literature review section of the report, we will provide an overview of the most relevant and similar works that have been conducted on the topic at hand. The main objective of this section is to present a comprehensive analysis of the existing literature in order to identify the key findings and limitations that have been previously reported. This review will provide a detailed examination of the methods, results and conclusions of the most similar works, and has been used to form the objectives and design of this study.

Software evolution and software maintenance are broad areas of research that have been studied for more than 30 years. Lehman et al. (1980, 1985, 1987) have built the most extensive and widespread body of research on software evolution for large, long-lived software systems. By several case studies of large software systems, Lehman (1997) could present his laws of software evolution, suggesting that as systems grow, the difficulty of adding new code increases unless the overall design is reorganized. Bennett and Rajlich (2000) made a roadmap for software maintenance and evolution by investigating how software can be designed to easily evolve. Their work contributed to better conceptualization of "maintainability" and how it should be measured. This allows for more effective methods and tools for program comprehension for both code and data. They also discuss that research makes progress regarding self-modifying systems, self-testing systems, and highly adaptive architectures, which calls for a large scope of research in this area.

Previous work has also been done on automatic static analysis tools and how they are used in industry. Based on research suggestions that these tools are underused, Johnson et al. (2013) investigated why developers do not use them and how these tools could be improved. Through interviews with 20 developers, they found that all participants thought the tools were beneficial to use, but that the main barriers were the false positive issues and the way the warnings were presented.

As SonarQube is the most frequently used maintenance tool, some studies have been focused on this specific tool. Lenarduzzi et al. (2020) got in contact with some companies that were dubious about the usefulness of the rules SonarQube proposed and investigated the fault-proneness of these rules. They conducted an empirical study on 21 open-source projects and applied a machine learning algorithm to label the fault-inducing commits. Based on their results, they suggest that the fault-prediction power of SonarQube's own model is extremely low and thereby suggest that companies should carefully consider which rules to apply.

According to Ferreira et al. (2021), there is a lack of understanding among researchers about how practitioners actually perform maintenance tasks. Consequently, practitioners become disconnected with new relevant research and researchers miss important industry information. And ultimately, this results in collaboration difficulties due to differences between different cultures and time perspectives. Hence, Ferreira et al. (2021) investigated the growing gap between software maintenance theory and the practitioners performing the maintenance tasks in practice. The results imply that more effort is needed to develop proper tools and methods for software maintenance, particularly in change impact analysis and software measurement.

Ferreira et al. (2021) employed survey questionnaires as a methodology to gather data from 112 software practitioners from 92 companies and 12 countries. While this approach allowed for a broad range of information to be obtained, it did not permit a detailed examination of the subjects. The use of quantitative methods such as survey questionnaires can lead to limitations in data interpretation, as the respondents may interpret the questions in ways not intended by the researcher. In contrast, the utilization of qualitative methods, such as interviews, would permit a deeper understanding of the subjectively experienced or perceived factors that influence the behavior, motivations, attitudes and priorities of the participants, which may not be captured by quantitative surveys.

This research builds upon Ferreira et al. (2021) examination of the utilization of software maintenance tools by practitioners, and how it compares to the literature. However, in this study, the approach employed will be qualitative interviews, as opposed to the use of questionnaires in the prior research. Similar to Ferreira et al. (2021), this study focuses on the gap between theory and practice in software maintenance, however, this research is further narrowed down by comparing the theory with the most frequently used maintenance tool in practice, namely SonarQube. The comparison generates an overview of what literature suggests regarding maintenance and what the actual practitioner is able to conduct using the SonarQube tool. Furthermore, finding out more about the underlying reasons for choosing a specific maintenance tool together with barriers for successful implementation and adoption of it.

To summarize, the most relevant and similar work will be applied in this report by:

- Comparing Johnson et al. (2013) interview findings about why automatic static analysis tools like SonarQube are not being used. This is done by identifying barriers for successful implementation and adoption for software maintenance tools through qualitative interviews.
- Analyzing if Lenarduzzi et al. (2020) investigation on the fault-proneness of SonarQube also applies in the software company studied in this case. Through interviews the satisfaction rate of SonarQube is examined in addition to the reasoning why or why not the tool is being used.
- Further research the gap between software maintenance theory and the maintenance tasks in practice examined by Ferreira et al. (2021) by conducting qualitative interviews with practitioners. As software maintenance is such a costly and critical affair, it can be of interest for practitioners to minimize the gap to be able to develop the optimal software product.

The limitations of the previous literature stated are synthesized into the objectives of this report and are presented in the following section.

4. Objectives

This research studies the selection and implementation of software maintenance techniques and methods by investigating how developers within a certain software company approach it. The purpose is to find underlying reasons why specific maintenance tools and methods are chosen and identify potential barriers regarding its implementation and adoption. Furthermore, finding out if there are any power balances within the studied organization that affects these processes. This study can contribute to a deeper understanding of how practitioners select and use software maintenance tools in industry today and how it aligns with the software maintenance research.

Through qualitative interviews with people working with software maintenance, this study aims to find key factors associated with successful implementation of methods proposed by maintenance theory. This case study will therefore contribute to a greater understanding of software maintenance tools used in practice and reasons why they are prioritized or not.

4.1 Research Questions

To achieve the purpose of the study, the following research questions were investigated:

Q1: What are the underlying reasons why a software maintenance tool is chosen?
Q2: What are the barriers for successful implementation and adoption of a software maintenance tool?
Q3: How does the software maintenance process in industry compare with the process described in literature when utilizing maintenance tools?

4.2 Delimitation

In this case study, the delimitation were as follows:

- The focus was on the selection and implementation of software maintenance tools in a single software company. The findings of this study may not be generalizable to other organizations or industries.
- The study only considered tools specifically designed for software maintenance, rather than broader project management or development tools.
- The study did not examine the maintenance processes or strategies of the company, only the tools used to support those processes.

- The study did not consider the training of the tools by the company's employees, only the selection and implementation process.
- The case study did not consider the financial cost or return on investment of the selected tools.

5. Methodology

This section describes how the study was conducted. The study was built upon a literature review followed by a case study of how maintenance tools are selected, implemented, and used in a software company. Thereby, the method can be described as consisting of two parts that are interconnected and discussed to reach a conclusion.

5.1 Literature Review

The literature review allowed us to identify gaps in the existing literature and to establish the context for our study. By comparing our results with those of similar works, we will be able to assess the consistency of our findings and to identify any discrepancies that may need further exploration. Ultimately, this literature review serves as a foundation for this study, and helps to ensure that the research is well-informed and relevant to the existing body of knowledge.

The literature was found through the search engines Scopus and Google Scholar by using the following keywords: *software maintenance, software maintenance tools, sonarqube, software evolution, software maintenance process* and *software quality tools*. Additional literature was found by using literature that the literature from our search hits referred to.

5.2 Case Study

The second part is the case study that was conducted in a software company where the external software quality evaluation tool SonarQube was used, which is the tool this study focuses on. The case study consists of two parts, one is practical maintenance work, guided by one of the software developers within the company, and the other part is qualitative interviews with two employed software developers. Another interview was made with a software maintenance researcher who also has industry experience. This respondent contributed with an external perspective and additional insights of the subject.

5.2.1 Practical Maintenance Work

To understand how the studied organization works with software maintenance in practice and how the automatic tools are used, one of the employed software developers showed us how it usually is done in their organization. The practical work with the code was performed by focusing on one code component at the time and investigating its quality state with SonarQube. Based on the tool's analysis, the code components that were graded as of lower quality could be detected and refactored. These parts contained code smells, bugs, high levels of complexity, unused lines of code, or parts that were not covered by any test case. When code like this was detected, we made a refactoring attempt or a general suggestion for how to improve the code quality. Then, the software developer chose what to change and implement or not. We only got a read license to ensure that we would not make any changes that could harm any internal principles or practices.

After the refactorizations, the quality score provided by SonarQube was revisited, to ensure that improvements were made. Insights into the maintenance techniques used in practice and the subjects covered by the tools could then be used for comparison with the framework based on research findings. The discrepancies formed a basis for the discussion and conclusion.

This second part of the method can at first sight be interpreted as action research since we performed practical work with the source code. However, the setting is natural and unobtrusive since we do not control or change any variable that is studied. In this research, the variables are the subjects covered by the maintenance tools and techniques used by the software company, and not the actual code that we work with. Thereby, the method can be classified as a case study that falls in the "field study" concept described by Stol and Fitzgerald (2018). This study will consequently be realistic and aimed for contributing to a greater understanding of software maintenance tools used in practice and on the gap between research suggestions and industry practices. However, the results will not have any precise measurement qualities and the findings will not be generalizable due to the inherent limitations of this research method.

5.2.2 Interviews

Three qualitative, semi-structured interviews were conducted where some open-ended questions were prepared beforehand, and other questions were asked based on previous answers. The interviews opened up for these follow up questions and discussions to allow the interviewee to share their views and to allow for a deeper understanding of the subject. The participants were asked beforehand to give their consent to record the interviews. All participants gave their approval, which enabled the transcription presented in the results. The interviewees were hand-picked based on their working or research area. Another four persons were invited for interviews but either denied our request or did not respond.

In order to get a well-rounded understanding of the subject of software maintenance, it was important to include a diverse range of perspectives. That is why the interviews were conducted with three individuals who research and work with software maintenance in different ways and at different levels. To see an overview of the respondents occupational profession, see table 1.

The first interviewee is a researcher and teacher in software maintenance. Their academic background and experience in teaching others about the subject provided valuable insights into the theoretical and educational aspects of software maintenance. The second interviewee is a team leader who manages a group of developers responsible for maintaining a complex system. They brought a strategic perspective to the discussion, explaining how they prioritize and plan for maintenance tasks within their organization. The third interviewee is a junior developer who works on the front-line of software maintenance, fixing bugs and implementing new features on a daily basis. Their hands-on experience provided valuable insights into the day-to-day challenges and solutions of software maintenance.

Overall, the diverse perspectives of these three interviewees gave a comprehensive understanding of the various aspects of software maintenance, and their insights were valuable in the study.

Table 1: Interview respondents.

Respondents	**Professional role**
Interviewee 1	Software researcher
Interviewee 2	Software developer
Interviewee 3	Software developer

6. Fieldwork findings

This section describes the results from the case study that was made in a mid sized software company to find out more about software maintenance work in practice and how tools are used and viewed upon by software developers within such an organization. The results are divided into three parts, the first part is an overview of the company examined, the second constitutes practical maintenance work and the third part is the result generated from the qualitative interviews.

6.1 Company overview

The following section provides an in-depth look at the anonymous software company that is the focus of this study. The section begins by providing a general description of the company, including its size, industry, and products or services offered. Next, the company's organizational structure and the roles and responsibilities related to software maintenance within the organization are explored. Finally, the company's software maintenance processes and practices, including the tools and methods used to maintain its software, are examined.

6.1.1 Description of the company

The anonymous technology company that is the focus of this study is a medium-sized, fast-growing company that specializes in providing delivery solutions for retail and distribution companies. The company connects, empowers and makes delivery networks available and smarter for its clients, which include some of the most well-known and respected brands and market leaders in the industry. The company is based in Sweden but has expanded to several other countries, with more than 500 employees.

The company places a strong emphasis on creating a human-centric and award-winning culture, and believes that businesses have a key role to play in striving towards a sustainable future. The company's culture is focused on self-leadership, trust, and empowerment. The company's success is explained to be driven by the people who work there, and the organization encourages everyone to lead and make a difference. The company's communicated mission is to be part of the solution for better commerce and a better world.

6.1.2 Organizational structure and roles related to software maintenance

The organization is flat and consists of self-leading agile teams. All teams consist of software developers and more application and customer consultancy related together with project management roles. Most of the teams work with delivering software solutions for customers' supply chains, but there are also teams working with R&D and development of new services and maintenance of existing systems. All the delivering teams work with development of customized solutions for their customers, but do not have any dedicated software maintenance role.

6.1.3 The company's software maintenance processes and practices

Software evolution and maintenance projects are conducted within the company, for example software modernization in terms of migration to new architectures, protocols, and performance optimizations. The organization has licences for the software maintenance tool SonarQube and has ongoing processes with considering new tools to buy. Besides this, there is a mantra for the software developers throughout the organization about leaving the code nicer than you found it, which is commonly known as "the boy scout rule". No explicit specification of their maintenance process was found.

6.2 Practical Maintenance Work

The organization uses the maintenance tool SonarQube, which is the tool described in 3.4.1 SonarQube. A software developer within the studied organization taught us how to use the tool according to how they use it. The way we got taught was by using a local instance of SonarQube and start by looking at an overview of the different code components and their quality grades. From here, we filtered out the ones with a lower quality grade. Out of the components with lowest quality, we looked into one at the time.

At this stage, the tool displayed the issues found in the code together with an estimation of the time effort it would take to fix it. The estimated time was shorter than the time it actually took, but that was expected since we do not work with code on a daily basis. SonarQube divided the issues into bugs, vulnerabilities and code smells. An explanation why the pointed out code part was problematic was also provided by SonarQube, sometimes together with a suggestion for how to fix it.

After finding an issue to fix, we reached the component through the IDE and came up with a fix locally. The software developer taught us that they normally make the code change on their branch and push it to merge into the main development branch that is shared between all the developers within the organization. From there, the code change is pushed to the test and finally to the production environment, but there are stages of inspection in between. We did not get into details about this process, but worth noting is that the practical work we did included the extra step of writing a change suggestion in isolation, and then sending it to the software developer who decided whether to implement it by fixing it in the real code project or not.

The bugs SonarQube found were corrected, and the parts that were analyzed as too complex, were refactored based on the theoretical principles of software quality, i.e., by unnestling loops and methods, and reducing the coupling between classes. Furthermore, the code smells were investigated and most of them were fixed. However, when SonarQube found code smells that were considered as false, by being in line with the company's principles, no action was taken.

We found the interface to be user friendly and easy to learn. The tool started helping us immediately and it was easy to find what we were looking for. Valuable information was easy to find thanks to the interface.

6.3 Interviews

The following section outlines the result of the conducted qualitative interviews by first presenting each respondent and then by summarizing the most relevant data collected from the interviews.

6.3.1 Interviewee 1

The first interviewee, the researcher, has been working with maintenance tools and regression testing and for the past 11-12 years. As a researcher, the main focus has been on analysis and creation of tools rather than using them as a developer. The researchers' main area within software maintenance is quality testing and automated regression testing. Part of their role as researcher is to investigate software tools that developers are using to improve their test processes, making the testing more efficient and effective. The goal is to create tests that make the users better by learning from the tools and becoming more creative developers. This is done by providing useful information and data from the tests to improve development skills.

6.3.2 Interviewee 2

The second interviewee has a broad developer role within the software company and has worked in the field for more than ten years. Interviewee 2 work with, among other things, coding, software infrastructure and currently in a maintenance project. Interviewee 2 has experience of maintenance in their current position in terms of fixing bugs, making the software components easier to understand and other general improvements. The software Interviewee 2 works with is an application that has been maintained for 7 years, where Interviewee 2 has been involved in the whole lifecycle of the application and experienced the importance of it when seeing how everything changes both inside the system and outside in the world. A lot of work has been made to adapt to the real world and to make the application fit to the new circumstances and requirements alongside with software development in terms of introducing new functionality.

6.3.3 Interviewee 3

The third interviewee is employed as a software developer within the software company and has worked full-time with development for less than a year. Interviewee 3 does not actively work with maintenance and describes an insecurity in how to do it and what it means. Interviewee 3 works mainly with software development in terms of developing new functionality and fixing code that does not work as intended. Besides this, interviewee 3 creates test cases for the code and recycles methods provided by an internal library in the company. Interviewee 3 describes that no education within software maintenance is taught in the organization and that they have not learned any processes for maintaining the code base or assessing the code quality. The general view from the perspective of interviewee 3 is that the code is good as long as it works, but that interviewee 3 is curious about learning more about software maintenance.

6.3.4 Compilation of interview findings

The following tables provide summaries of the findings generated from the three interviews conducted in the case study. Table 3 presents the answers of questions related to the selection of tools, while table 4 provides the answers of questions related to the implementation and adoption of tools. The tables are two-dimensional matrices where the first column shows all questions asked to the corresponding research area. Each of the remaining columns maps out the individual responses to each question.

Table 3: Interview results from selection of tools related questions

Question	Interviewee 1	Interviewee 2	Interviewee 3
How does the company organize their software maintenance? Who is responsible for it?	I don't work in a software company.	All software developers in the company are responsible for maintenance. Developers work actively by acting upon the logs provided by the tool SonarQube for instance. We have a distributed model of maintenance. Maintenance is not dedicated to a single person, team or area - we work according to the boys scout rule "leave things prettier than you found them".	From what I know, I don't think the company works particularly with software maintenance. I ignore the logs, I don't know how to interpret them and since the code is working, I don't see the problem.
Which maintenance tools are the interviewee using?	Mainly two, SonarQube and PITest.	We use both automated and manual tools. For automated tools we use Dependabot and SonarQube.	Eslint and ReSharper.
Who in the organization is responsible for selection of maintenance tools?	I think it depends on the organization and on the tema. Typically the most senior employee or the developer will be the one proposing tools. I think the ultimate decision comes from the manager, but the people who will shape the value	It is a different story for each tool. As far as I know we don't have any specific person or team that decides which tools to use. We have an open attitude to how to do things. It really comes down to you and your team if you want to use	The tool was required in the project, so I guess it is the Project Manager who was responsible for selecting the tool.

	of the tool through the development cycle will be the developers. If the managers are not involved in the code artifact, then I would be surprised if they care at all what maintenance tool is being used, but they might care about which tools that developers are happy with.	something. Based on our organizational structure we are very trusted by the management to know if a tool is cost saving in the end/worth it.	
Has the interviewee been involved in the selection process of a maintenance tool in a company?	Yes, and no. Yes, in the way that I have consulted about maintenance tools for companies, but that was mainly in the role of a researcher, not as a developer - I was never part of the development team in a company and made final decisions.	Yes, I was involved when we changed a tool. We changed from Dependabot to another. I don't remember what the other tool is called, but it is another tool that does the exact same thing but more effectively. **Follow-up question: "Were there any switching costs involved?"** No, not more than the time it took for the person to change it, which probably took about one hour.	No.
Reasons for using the specific maintenance tool	SonarQube: Because it's popular. It has a nice interface. Works with different languages. Comprehensive documentation. No demanding installation. Shallow learning curve.	SonarQube: Makes me a better developer. Easier to know what to do. Helps me to avoid making mistakes. Saves time because you don't have to do it manually. More bug knowledge than me.	Eslint and ReSharper: It was required to use the tool in the project, so I have not thought about the reason why it was choses. Much happens automatically. It saves time, but sometimes gets wrong when it guesses.

The most important criterias when selecting tool	When I give advice, I always try to first listen to their needs and then try to do an investigation on which tool is best suitable for their needs. So, it always depends on their particular need when choosing which tool to use.	Tools should always be chosen from the specific need. Not just use industry standards out of habit. I think that is the most important thing. There are so many different tools, some of them might not be right for the thing you are doing.	I don't know, I have never selected a tool by myself.

Table 4: Interview results from the adoption and implementation of tools related questions

Question	**Interviewee 1**	**Interviewee 2**	**Interviewee 3**
Reasons for successful adoption of tools	The tool needs to be compatible with the processes and tools that the developer already uses. I see more and more that a tool can show a lot of efficiency and effectiveness, meaning that it can be the best tool to make maintenance cheaper, but if people are struggling with using and adopting it, if the usability, understandability, compatibility and comprehensiveness is low, then engineers will not trust it, and then they will not use it, so then it's not adopted. A lot of people overlook the human factor of adopting tools.	Getting people to actually use and learn the tool. Tools are often a big help, but if you don't know how to use them then it might stop you more than it helps you. Therefore, I think if a tool has good usability, the odds of adopting the tool successfully increases.	I don't know, I don't use many maintenance tools.

Is usability important in a maintenance tool?	It is very important. As I said earlier, the barrier to start using it has to be low to make developers actually start using it. For example if you have to do a lot of installing, the effort may not be worth it. I want to be able to use it straight away.	Usability is really important. If a tool is hard to use, they are kind of useless. We hire a lot of newly graduates and they don't know anything about maintenance tools because they don't learn them in school, so tools need a shallow learning curve, otherwise they will not be used. That's why SonarQube and Dependabot is great because they are integrated into the platform we already use, and I think that is really important.	It is important. We have not learned about these tools in school so they have to be easy to learn.
What usability aspects are most important?	Shallow learning curve and compatible with current systems.	Easy to get started I think, if you can get value from the tool from the strat without knowing much about the tool, that's a good starting point for a tool.	I don't know.
What is the common attitude towards working with maintenance?	I don't think there is a particular common view on it, I think it depends. Most likely managers are interested in seeing the outcome whereas developers are interested in seeing the suggestion and learning. Moreover, software maintenance does not have a clear definition. I think that people in	I think the view of maintenance work differs a lot. I think business people, people that are further away from the hardware and software usually tend to ignore it with an attitude: "it already works so why should we fix it?". But that sometimes leads to these types of problems. Personally, I think it's	I don't know, I have not heard much about it but it would be interesting to learn more.

	the industry have different views on what it is. To me, unless you are doing a new software from scratch, you are going to be maintaining the software.	fun, but it is not a word I think other people connect to maintenance. Often it is kind of neglected as something wasteful, like that you put work into something that already works. But for me, maintenance is buying us time for the future. Take a car for example, if you buy a car and never service it, the probability of the car breaking will be higher, and it will be a costly affair to fix it then. That is the point of maintenance. I have seen it happen in the industry so many times.	
Personal view on software maintenance	My view is that maintenance is very important because it is more common to evolve existing code than creating new ones from scratch. Software has a life cycle, and I think, today especially, successful and modern software development focuses much more on creating longer lived software. The longer the software lives, the less time we need to put into creating new software systems.	For me, maintenance is a priority, it is very important. If you don't maintain code, you can end up with code that you can not fix due to lack of maintenance, and the code becomes useless. It is profoundly stupid not to do maintenance. You have the tools to help you which saves a lot of time. If you neglect it, the damage will be huge later on and will cost a lot of time and resources to fix it.	I have not worked with it a lot, but it would be interesting to learn more about it.

How well does the tool help with the maintenance work?	I don't do maintenance work myself, I only research and develop them.	The tools are always just an assist, it is something that is supposed to help you, hold your hand, but we always have people that do a second review of the code. That is the real value of the maintenance tool, for me at least, that we have more than two eyes on a piece of code. I think for SonarQube and Dependabot and those kinds of tools, those are about not introducing problems in the future - you are stopping yourself in the very beginning - it is very proactive. And that the tools do well I think.	Have not used enough tools to say.
Satisfaction with the current tools used	As I don't use them in practice personally, I can't say. But I think SonarQube has pretty good usability.	Yes, I think most of the tools I use are really good. SonarQube has helped a lot and you actually learn a lot from it as well. You get a lot of comments and information about problems that you can learn from. Moreover, after solving that problem in SonarQube you tend to avoid making the same mistake again. Overall I think it makes you a better developer.	Have not used enough tools to say.
Is maintenance	I would say that they	As said earlier,	Not that I know of, we

prioritized in the company	usually don’t prioritize, managers do prioritize to have new features and expand the product, so I think to me, that is maintenance, but I can see why they might not see it like maintenance.	maintenance is not dedicated to a person, team or an area - it is up to each of us developers to make sure that it happens, we do not have any special requirements from top management about how and when it should happen.	have no requirement to do it as far as I know. If a code works then I don’t do anything to change it, since it works fine I don’t see why we have to change it.
Challenges in the implementation and adoption of maintenance tools	For researchers, one difficulty is not being able to witness the adaptation of tools in companies' software teams. It would generate great usage information about the tool. We do control experiments, but it is difficult to see the long term effect of the tool, how it is adopted. The challenge in the industry is a layer of politics. Since the tool it's part of a bigger organization, you can’t just develop a tool and not tell anyone about it, you need to have agreements, and it needs to be approved, there needs to be some consensus in the development team.	The challenge that I face with tools like SonarQube is that it is a static analyzer, which basically means that it read all the code as texts and analyzes that based on a set of rules, this means that some things might not work exactly according to SonarQube rules, and therefore it suggest the wrong things. So the biggest challenge is that SonarQube does not always give good suggestions and therefore you start mis-trusting it and stop using it. I don’t know how many times I have ignored what SonarQube says - it can break things if you accept everything that the tool suggests.	I don’t know.
Barriers for successful adoption of a maintenance tool	If people are struggling with starting to use the tool, due to low	Poorly performing tools. If a tool would fail when doing the analysis of the code, or	I don't know.

	understandability and steep learning curve, then developers will not use it and won't adopt it. A lot of people overlook the human factor of adopting tools.	updated wrong version, that would be a big problem. If that happened more than one time, I would never use the tool again, I would not trust it anymore. Cost is another barrier. If it is a really expensive tool then you really have to motivate the value it provides for the company. I think both SonarQube and Dependabot are worth it because it saves so much time, and the biggest cost of a company is the cost of an employee.	

7. Discussion

Some discrepancies between the views on software maintenance within the organization were found in the interviews. Interviewee 2 stated that all developers within the company worked actively with maintenance by acting upon the logs provided by SonarQube, while Interviewee 3 said that they ignored the logs since they do not know how to interpret them and why they would need to take action on them. The logs are provided by SonarQube directly in the IDE, which is a part of the process that we did not assess in the practical maintenance work. Thereby, this indicates that the three different software developers work with maintenance in three different ways.

Furthermore, Interviewee 2 stressed the importance of maintenance efforts, in line with what Interviewee 1 said, while Interviewee 3 had almost no knowledge of the subject and put no maintenance efforts in the daily work. Based on this lack of knowledge, together with an expressed curiosity of the subject, one suggestion would be to share the maintenance visions and knowledge better within the organization, and create a mandatory internal course for educating all developers. This could increase the knowledge level and make the maintenance work easier when sharing the efforts put on maintenance work more evenly. As stressed by literature, maintenance is important for assuring long-lived software systems and should be prioritized. If all developers worked with maintenance in parallel with development, for example during the times when not having any development task to do, the software quality would increase and the application would be better maintained without needing additional personnel. Also, the time needed for ensuring good code quality would be lowered, if all developers actually read the logs and took action on the presented issues right away. If the logs are ignored and later on checked by another developer, that person first needs to get familiar with the code component to make sure they understand it. Eliminating this step would save time.

As stated by Interviewee 1, and also found in theory, the usability is crucial for both the choice of the tool and the adoption of it. The tool must be easy to use and quickly show that it helps you as a developer and get trusted enough to become a part of the developers daily development and maintenance process. Hence, the main reason for a maintenance tool to be chosen seems to be if the tool proves a high degree of usability since it helps users to realize the value of the tool more quickly. Related to usability, Interviewee 1 mentions that one reason for selecting SonarQube was due to the tool having a good interface. A software maintenance tool with a good interface is easy and intuitive to use, allowing users to quickly and easily perform maintenance tasks without requiring extensive training or documentation. A good interface will also provide clear and concise feedback to the user, informing them of the status of their tasks and any errors or issues that may have occurred. Overall, a good interface for a software maintenance tool should enable users to effectively and efficiently perform their tasks, without getting in the way or requiring a steep learning curve. Hence, a good interface improves the usability of a maintenance tool and most likely raises the odds for it to be selected.

Furthermore, both Interviewee 1 and 2 emphasized the advantageous attribute that SonarQube was compatible with their existing software systems. The tool was completely integrated in their current platforms. When a software maintenance tool is compatible with an existing software system, it means that the tool is able to access and modify the necessary files and components of the system in order to perform the maintenance tasks. In order for a maintenance tool to be compatible with a software system, it must be designed to work with that specific system, or at least with a similar system that uses the same or similar technologies. This implies that high compatibility with a wide range of different software systems might also be a crucial factor for a tool to be chosen.

The fact that tools are selected due to being an industry standard and popular among developers was also discussed during the interviews. Interviewee 1 mentions that this was indeed one of the reasons why they started using SonarQube in the first place, due to its popularity. In contrast, Interviewee 2 emphases that a tool should never be chosen because it happens to be the industry standard, it should be because it fulfills a certain need. Hence, the selection of tools should be based on what kind of maintenance tasks the user is in need of, and not what others are using. This was mentioned later by Interviewee 1 as well. When consulting firms on tool selection, Interviewee 1 always recommends tools based on the need of the company after doing an investigation. However, it is noteworthy that in selecting SonarQube, Interviewee 1 personally started using the tool due to its popularity as mentioned earlier. This contradicts the researcher's own recommendation to others. Hence, this might be proof that a widespread usage of a tool might be an underlying reason why it is selected.

The barriers for successful implementation and adoption identified in the empirical data are as follows: low usability, steep learning curve, high price of the tool, and lastly, poorly performing tools in terms of too many bad recommendations and failings.

Bad recommendations may harm its trustworthiness and in turn lead to not being used. This was particularly stressed by Interviewee 2 who stated that SonarQube sometimes makes recommendations that are so bad it could even destroy code if one would accept it. Interviewee 2 emphasized the importance of considering the tool as an aid for the developer, providing an additional perspective on the code, but ultimately the final decision regarding actions to be taken rests with the developer. Additionally, the tool can also be helpful in the sense of making maintenance prioritized by reminding developers of fixing issues and bugs when automatically generating logs during the development process.

Interviewee 1 mentions that an important discovery in their research is the neglection of the human factor. The research showed that even though a tool could show more efficiency and effectiveness than other tools, meaning it could ultimately decrease maintenance costs, it was still no guarantee for successful adoption of the tool. If developers were struggling with using the tool due to poor usability and understandability, then the software engineers would not trust it. This leads to the maintenance tool not being used and therefore, not adopted. Interviewee 1 emphasized that the human factor is often overlooked.

The learning curve of maintenance tools is another aspect discussed by the respondents. A shallow learning curve is generally more user-friendly and easier for people to pick up and use without a lot of training or experience. On the other hand, a product with a steep learning curve may require more time and effort to master, but may also offer more advanced features and functionality. According to Interviewee 2 and 3, software maintenance tools are not typically taught in educational settings. This means that newly graduates employed by software companies lack the basic knowledge of maintenance tools which may lead to difficulties in adopting tools that are not user-friendly. By reviewing the literature and the empirical data of this study, successful implementation and adoption of a maintenance tool is more likely to occur when the learning curve of the tool is shallow.

As found by previous research, there is a gap between research suggestions and industry practices regarding maintenance efforts. What is found in this study is a knowledge gap and a usage of tools gap between two software developers within the same company. On the one hand, this seems natural and inevitable, as knowledge sharing never can or should reach 100 percent within an organization since different people should have expertise within different areas depending on their function and interests. On the other hand, this seems troublesome, as Interviewee 2 describes that everyone works with maintenance in the way they describe, which differs from how the practical work was taught, and how Interviewee 3 describes their work. The practical maintenance work was taught by another developer in a different way, while still using the same tool, SonarQube. Interviewee 3 showed a close to total lack of knowledge within maintenance, and did not know any way they worked with it in their day to day work, which contradicts the way of working with maintenance described by Interviewee 2.

As literature mentions, the applicability of tools is critical in terms of its adoption. Maintenance tools must be easy to use in different types of projects and correlate with the user's working culture, supporting the techniques and methods already used by the developer. This view of the applicability stressed by literature does align to some extent with the view of the practitioners interviewed. Interview 1 mentions that SonarQube is easy to use since it works with multiple different program languages, has comprehensive documentation and interface without any demanding installation process. This implies that SonarQube is quite applicable to the way they are already working with maintenance. Moreover, Interviewee 2 emphasizes that the ability to integrate SonarQube to their platform was particularly important for its adoption. Hence, integration facilitates the applicability of a maintenance tool.

A main concern is however that organizations seldom adopt a separate maintenance process because they struggle with distinguishing software maintenance from software development. We have seen several initiatives from research and software standards organizations towards better defined maintenance processes. One suggestion to software developing organizations is thereby to imitate, and work with defining separate maintenance processes to ensure that all developers contribute to the maintenance work and that the software systems are maintained in the way the management wants. As research testifies that the maintenance practices differ between different organizations and that different parts become more or less important in different projects, it is difficult to define a single, ultimate process for maintenance. The work to be done is rather to clearly define the objectives and the building blocks to allow for customization but avoid absence of maintenance work and tools.

8. Conclusions

This study aimed to answer the research questions presented in 4.1 Research Questions. Based on the discussion of the findings, this section presents the conclusions for each research question.

Q1: What are the underlying reasons why a software maintenance tool is chosen?
One of the underlying reasons why a specific tool was chosen were found to be the usability of the tool. It was found to be important for the developers to quickly see how the tool helps and that the tool was easy to start working with. Another underlying reason seems to be the popularity of the tool. If a tool is well known and used by many, it is more likely to be chosen. However, this is not a valid reason according to research and developers in industry, since it should rather be chosen based on the needs.

Q2: What are the barriers for successful implementation and adoption of a software maintenance tool?
According to our findings, one barrier for successful implementation and adoption of a tool is lack of knowledge about the awareness of the tool, that it should be used, and how it should be used. The tools an organization wishes to implement need to be carefully communicated and teached to all developers to enable proper adoption. As stressed by previous research, it is important to distinguish between software development and software maintenance and define separate processes for software maintenance work. Since different parts of maintenance are important for different software, this must be customized and defined for each software project. These processes should also include a specification of which tools to use and how to use them to assure a proper adoption.

Q3: How does the software maintenance process in industry compare with the process described in literature when utilizing maintenance tools?
As previously stated in literature, there is a gap between research suggestions and industry practice for software maintenance in general. In this study, three different ways of practicing maintenance work were found from instructions or interviews with three software developers within the same company. Thereby, all studied developers worked with maintenance in different ways and we assume that even more different approaches would be found if more developers would be studied. One of the interviewees expressed that they did not work with

maintenance at all, and did not know much about the subject. This contradicts the research suggestions about making maintenance work a top priority to reduce the complexity of the systems, enable continuous growth, and sustainable evolvement of long-lived systems.

9.1 Recommendations for the company

Based on the findings from the interviews and literature insights, implications for the company of this case study are generated in this subsection.

Everyone in the organization working with code is responsible for performing maintenance tasks on the software. However, the view of the maintenance process differed depending on who we talked to in the organization. One viewed maintenance as an absolute necessity to avoid unusable code in the future, leading to huge time and monetary expenses. Another in the company nearly never performed maintenance tasks, and rarely used the maintenance tools when doing them. This discrepancy was identified as one of the main barriers for successfully adopting maintenance tools as stated in the previous section.

Reasons for lack of knowledge about maintenance processes might be due to than in the flat organization structure, everyone is free to work in a way that best fits them, without any specific commands for how to work with maintenance. Hence, the first recommendation is to highlight the importance and value of performing maintenance tasks throughout the company. This can be done by providing regular training sessions on the benefits and best practices of maintenance, setting clear guidelines and expectations for maintenance within the company, and recognizing and rewarding individuals or teams who consistently perform maintenance tasks effectively. Additionally, incorporating maintenance metrics into performance evaluations and setting up a dedicated maintenance team or assigning specific maintenance responsibilities to certain team members can also help to ensure that maintenance is being performed consistently and effectively across the organization. Another way of working towards a more consistent maintenance process is by using an incentive inducing tool, such as the gamification platform Quboo (Quboo Docs, 2021). This tool aims to make software tasks fun by turning them into a game. The user gets scores when performing tasks such as fixing SonarQube issues and improving the software quality. One idea could thereby be to investigate whether this tool would be a successful addition in this organization.

In the selection process of software maintenance tools, the usability of the tools was identified as the primary criterion for selection. Therefore, it is recommended that the company continue to rely on developers as the primary selectors of tools. As developers are the primary users of these tools in their daily work, they are best equipped to evaluate the usability of the tools and make informed decisions. However, as there is a risk of developers selecting tools based on industry standards and popularity rather than their specific needs, it is important to implement a robust evaluation process that includes a thorough examination of the tools' functionalities and a review of their alignment with the organization's and projects' specific requirements. This can be achieved by involving a cross-functional team of developers, project managers, and IT professionals in the selection process and conducting a thorough evaluation of the tools based on predefined criteria such as cost-effectiveness, scalability, and flexibility. Furthermore, it is important to periodically review the tools to assess their continued relevance and effectiveness, as technology and requirements change over time.

The company frequently hires newly graduated developers, and as reported by the interviewees, these individuals often possess limited experience in software maintenance. This has resulted in a lack of utilization of maintenance tools among these employees. To address this issue, the implementation of workshops for newly hired employees to familiarize themselves with software maintenance tools may be an effective method of increasing the adoption of these tools within the organization. By providing hands-on training and experience with the tools, newly hired employees will be better equipped to utilize them in their daily work, leading to a reduction in maintenance costs in the long term. This is of paramount importance given that software maintenance is a costly but necessary aspect of software development and the software lifecycle. The utilization of maintenance tools can significantly reduce the time and resources required for maintenance activities.

Lastly, it would be interesting to analyze whether it would be beneficial to have dedicated software maintenance professionals within the delivery teams to ensure that the software is successfully maintained. Another interesting discussion would be to formalize the developers' responsibility to include some percentage of maintenance activities. For example, all developers should aim for 50 % of their time being spent on quality assurance and software maintenance work. Worth noting is that the company does not have any troubles with growing and evolving their code base today, which indicates that their maintenance processes work fine. However, the risk with vaguely defined maintenance processes is that the effects are shown in the future, when it is too late to do any efforts due to the built-in complexity. That captures the whole concept of maintenance, as the approach of “as long as the functionality works, the code is fine ", is dangerous. If the technical debt continously increases, the code eventually is forced to be eliminated or modernized through complex and costly projects, when the situation could have been avoided by continous refactoring. The boy scout rule seems to be known and adopted by all developers within the company, which indicates that all software developers actually work with maintenance to some degree. Interviewee 3 might in fact spend more time on maintenance efforts than they actually believe. What so ever, it would not hurt to be better educated in what maintenance really is and why it is important. Furthermore, to make use of the tools they pay for and gain the profits yielded of using them. There is a reason why licensed tools are bought, but it will be a costly and unprofitable affair if they are paid for but not used.

9.2 Future Research

This research only studied a few software developers within the same company. Future research could be conducted on a larger scope in terms of more developers and/or more companies to get a better understanding and allow for more statistically significant results. This study has too few data points to allow for any discussion of statistical significance, but rather contributes with a sample that opens up for future work. Besides future research on software maintenance tools selection and implementation, more research is needed in how to develop successful maintenance processes, where the maintenance tools play a natural part. Action research on this topic could be suitable to be able to try out and evaluate process development models in industry. By doing it in a real world context, it could contribute to bridge the gap between software research and practices in industry.

9.3 Threats to Validity

There are several threats to the validity of the study. One threat is that we only had a limited amount of time to conduct the research and therefore may not have been able to go as broad as we would have liked. This could mean that we may not have been able to interview a representative sample of people, which could lead to a biased view of the topic.

Another threat to the validity of the study is that we only interviewed a few people. While it is not necessarily a problem to interview a small number of people, it could mean that our findings may not be generalizable to the wider population of software developers and maintenance professionals. Additionally, the fact that we only interviewed people from one software company could also limit the generalizability of our findings. However, since this is a case study, it is to be expected.

Finally, the fact that we only interviewed one software maintenance researcher and two software developers could also be a threat to the validity of our study. While these individuals may have valuable insights and experiences, they may not be representative of the wider software maintenance community.

It is important to consider these threats to the validity of the study and to try to address them as much as possible in our analysis and conclusion. This can help to ensure that the study is as accurate and reliable as possible.

References

Bennett, K. H., & Rajlich, V. T. (2000). Software maintenance and evolution: a roadmap. *Proceedings of the Conference on the Future of Software Engineering*, 73-87.

Brooks, F. (1975). The Mythical Man-Month. *Addison-Wesley*. ISBN 0-201-00650-2.

Ferreira, M., Bigonha, M., & Ferreira, K. A. M. (2021). On The Gap Between Software Maintenance Theory and Practitioners' Approaches. *2021 IEEE/ACM 8th International Workshop on Software Engineering Research and Industrial Practice (SER&IP),* 41-48. doi: 10.1109/SER-IP52554.2021.00015.

Godfrey & Qiang Tu. (2000). Evolution in open source software: a case study. *Proceedings 2000 International Conference on Software Maintenance*, 131-142, doi: 10.1109/ICSM.2000.883030.

ISO/IEC/IEEE. (2022). ISO/IEC/IEEE International Standard - Software engineering - Software life cycle processes - Maintenance. *ISO/IEC/IEEE 14764:2022(E)*, 1-46. doi: 10.1109/IEEESTD.2022.9690131.

Johnson, B., Song, Y., Murphy-Hill, E., & Bowdidge, R. (2013). Why don't software developers use static analysis tools to find bugs?. *Proceedings of the 2013 International Conference on Software Engineering. IEEE Press,* 672–681.

Khan, K., Lo, B., Lo, B., Skramstad, T., & Skramstad, T. (2001). Tasks and Methods for Software Maintenance: a process oriented framework. *Australasian Journal of Information Systems,* 9(1). https://doi.org/10.3127/ajis.v9i1.227.

Lehman, M. M. (1980). "On Understanding Laws, Evolution, and Conservation in the Large-Program Life Cycle". *Journal of Systems and Software.* 1: 213–221. doi:10.1016/0164-1212(79)90022-0.

Lehman, M. M., & Belady, L. A. (1985). Program evolution: processes of software change. *Academic Press Professional, Inc..*

Lehman, M. M., Ramil, J. E., Wemick, P. D., Perry, D. E., & Turski, W. M. (1997). Metrics and laws of software evolution - the nineties view. *Proc. of the Fourth Intl. Software Metrics Symposium (Metrics'97).*

Lenarduzzi, V., Lomio, F., Huttunen, H., & Taibi, D. (2020). Are SonarQube Rules Inducing Bugs?. *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 501-511. doi: 10.1109/SANER48275.2020.9054821.

Lientz, B. P., & Swanson, E. B. (1980). Software Maintenance Management. *Addison-Wesley Publishing Co.: Reading MA*. doi: 0.1049/ip-e.1980.0056.

Mantyla, M., Vanhanen, J., & Lassenius, C. (2003). A taxonomy and an initial empirical study of bad smells in code. *International Conference on Software Maintenance. ICSM 2003. Proceedings,* 381-384. doi: 10.1109/ICSM.2003.1235447.

Marcilio, D., Bonifácio, R., Monteiro, E., Canedo, E., Luz, W., & Pinto, G. (2019). Are Static Analysis Violations Really Fixed? A Closer Look at Realistic Usage of SonarQube. *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, 209-219. doi: 10.1109/ICPC.2019.00040.

Object Management Group. (2012, 11 May). *Architecture-Driven Modernisation Task Force.* https://www.omgwiki.org/admtf/doku.php.

Quboo Docs, Quboo Team. (2021). https://docs.quboo.io/docs/plugin/.

SonarQube Documentation, SonarSource SA. (2023). https://docs.sonarqube.org/latest/.

Stol, K-J., & Fitzgerald, B. (2018). The ABC of Software Engineering Research. *ACM Trans. Softw. Eng. Methodol*. 27, 3. https://doi.org/10.1145/3241743.

Turski, W. M. (1996). Reference model for smooth growth of software systems. *IEEE Trans. on Software Engineering*, 22(8).

Wang, J., Wang, S., & Wang, Q. (2018). Is there a "golden" feature set for static warning identification?. *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement - ESEM'18. ACM Press*. https://doi.org/10.1145/3239235.3239523.